15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 1

Overview – The Genome Sequencing Problem
MAJOR THEMES

- Defining precise **problem** and **data abstractions**, 

- Designing and programming
  - correct and efficient algorithms and data structures
  - for given problems and data abstractions

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions Data</td>
<td>Problem Abstract Data Type</td>
</tr>
<tr>
<td></td>
<td>Algorithm Data Structure</td>
</tr>
</tbody>
</table>
**Problem vs. Algorithm**

- Sorting, string matching, finding shortest paths in graphs, . . . , are **problems**
  - **Input:** A sequence \([a_1, a_2, \ldots, a_n]\)
  - **Output:** A permutation of the sequence \([a_{i_1}, a_{i_2}, \ldots, a_{i_n}]\) such that \(\forall j, 1 \leq j < n, a_{i_j} \leq a_{i_{j+1}}\)

- Quicksort, Mergesort, Insertion Sort, . . . , are **algorithms** for sorting.
Abstract Data Types vs. Data Structures

- A set is an abstract data type (ADT)
  - Test membership, intersect, union, difference, . . .

- Sequences, trees, hash-tables are examples of data structures.

- ADT’s determine functionality, data structures determine costs.
Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Multi-core Chips
**Multi-core Chips**

### Intel Core i7 Processor Series

**Features & Specifications**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel Core i7-965 Extreme Edition</th>
<th>Intel Core i7-940</th>
<th>Intel Core i7-920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Speed (GHz)</td>
<td>3.20</td>
<td>2.93</td>
<td>2.66</td>
</tr>
<tr>
<td>QPI Speed (GT/sec)</td>
<td>6.4</td>
<td>4.8</td>
<td>4.8</td>
</tr>
<tr>
<td>Socket</td>
<td>1366-pin LGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td>8 Megabytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Speed Support</td>
<td>DDR3-1066</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDP</td>
<td>130 Watts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overspeed Protection Removed</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Processor Architecture</td>
<td>New Intel Core micro architecture (Nehalem) 45nm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key Platform Features**

- Intel Hyper-Threading Technology delivers 8-threaded performance on 4 cores
- Intel Turbo Boost Technology
- 8M Intel Smart Cache
- Integrated Memory Controller with support for 3 channels of DDR3 1066 memory
- Intel QuickPath interconnect to Intel X58 Express Chipset
## Parallel Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-core</td>
<td>8-core</td>
</tr>
<tr>
<td>Sorting 10M strings</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove dupl. 10M strings</td>
<td>0.66</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min. span. tree 10M edges</td>
<td>1.60</td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS 10M edges</td>
<td>0.82</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Running times in seconds
15-210 vs. a Traditional Course

- Emphasis on parallel thinking at a high level
  - Parallel algorithms and parallel data structures

- Purely functional model of computation
  - Safe for parallelism
  - Higher level of abstraction

- Ideas still relevant for imperative computation
  - Lot of overlap, but covered differently!
SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms
Sequencing the Genome

- The human DNA molecule encodes the complete set of genetic information using 4 bases
  - Adenine (A), Cytosine (C), Guanine (G) and Thymine (T)

- A sequence of about
  - 3 billion base pairs
  - arranged into 46 chromosomes

makes up the human genome.
A chromosome is a sequence of genes

A gene is a sequence of the base pairs
  - But there seem to be a lot of base-pairs with no apparent functions.
SEQUENCING THE GENOME

SEQUENCING THE GENOME

SEQUENCING THE GENOME


OVERVIEW – THE GENOME SEQUENCING PROBLEM
CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS
Fall 2013
Sequencing the Genome

- Determining the complete DNA sequence is a grand challenge.
- Very hard to do in one go with wet lab techniques.
- The Shotgun Technique has been found work quite well.
SHOTGUN SEQUENCING

- Break up multiple DNA strands into short segments
  - Chemistry!

- Short segments are sequenced.
  - Chemistry!

- Stitch short sequences computationally.
  - This is where CS comes in.
# Shotgun Sequencing

<table>
<thead>
<tr>
<th>Strand</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>AGCATGCTGCGAGCTGCTTAGGCTA</td>
</tr>
<tr>
<td>First shotgun sequence</td>
<td>AGCATGCTGCGAGCTGCT-----------TAGGCTA</td>
</tr>
<tr>
<td>Second shotgun sequence</td>
<td>AGCATG----------------------------------------</td>
</tr>
<tr>
<td></td>
<td>-----CTGAGCTGCTTTAGGCTA</td>
</tr>
<tr>
<td>Reconstruction</td>
<td>AGCATGCTGCGAGCTGCTTAGGCTA</td>
</tr>
</tbody>
</table>

Suppose you have three strands sequenced:

```
catt ag gagtat
cat tagg ag tat
cat tta gga gtat
```

But they really come in a messy way, e.g.,

```
catt ag tta cat tagg ag gagtat
tat ca gga gtat
```

So how do we stitch them?

- Given a set of overlapping genome subsequences, construct the “best” sequence that includes them all.
SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms
The Abstract Problem

The Shortest Superstring Problem

Given
- an alphabet of symbols $\Sigma$, and
- a set of finite strings $S \subseteq \Sigma^+$,

return
- a shortest string $r$ that contains every $s \in S$ as a substring of $r$.

- $\Sigma, \Sigma^+$
- $\Sigma = \{A, C, G, T\}$
SOME OBSERVATIONS

- Ignore strings that are already in other strings. Why?

  \{catt, ag, gagtat, cat, tagg, ag, tat, ca, tta, gga, gtat\}

  ↓

  \{catt, gagtat, tagg, tta, gga,\}

- Each string must start at a distinct position in the result. Why?
SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms:
  - The Brute Force Algorithm
The Brute Force Algorithm

The Brute Force Technique
Enumerate all possible candidate solutions for a problem
- score each solution, and/or
- check each satisfies the problem constraints
Return the best solution.

How does this apply to the SS Problem?
- Generate permutations
- Remove overlaps
- Stitch strings
- Select the shortest resulting string
The Brute Force Algorithm

- catt tta tagg gga gagtat
- catt tta tagg gga gagtat
- cattaggagatag

**Lemma**

Given a finite set of strings $S \subseteq \Sigma^+$, the brute force technique finds the shortest superstring.

- See handout.

So what is the problem with this technique?
There are just too many permutations!

So, \( n = 100 \rightarrow 100! \approx 10^{158} \) permutations.

Testing at \( 10^{10} \) permutations/sec, you need

- \( \approx 10^{148} \) seconds
- \( \approx 10^{143} \) days (\( \approx 10^5 \) seconds/day)
- \( \approx 2.7 \times 10^{140} \) years
- \( \approx 2.7 \times 10^{138} \) centuries

Not bloody likely you will test each permutation before hell freezes over!

- Even if every subatomic particle in the universe was a processor
PROSPECTS FOR A FASTER ALGORITHM?

- SS belongs to very important class of problems called NP (for Nondeterministic Polynomial).
- For such problems, no algorithm with polynomial work is known.
- But solutions can be verified in polynomial work!
- Wait for 15-451 and 15-453 for the gory details!
- But usually there are approximation algorithms
  - with bounds on the quality of results, and
  - perform better in practice.
SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms:
  - The Brute Force Algorithm
  - Reducing SS to TSP
**Problem Reduction**

- A *reduction* is a mapping from one problem \((A)\) to another problem \((B)\), so that the solution \(B\) problem can be used to solve \(A\).
  - Solving a set of linear equations, reduces to inverting a matrix.
- Map the instance of problem \(A\) to an instance of \(B\),
- Solve using algorithms for \(B\)
- Map the resulting solution back.
Reducing SS to TSP

The (Asymmetric) Traveling Salesperson Problem (TSP)

Given a weighted directed graph
- find the shortest path that starts at vertex $s$, and
- visits each vertex once, and
- returns to $s$.

≡ Hamiltonian path with the lowest total sum of weights

So, how is this related to SS?
REDUCING SS TO TSP

If $s_i$ is followed by $s_j$ in how much will the SS length increase?

- $s_i = \text{tagg}$ followed by $s_j = \text{gga} \rightarrow \text{tagga}$

General case?

- $w_{i,j} = |s_j| - \text{overlap}(s_i, s_j)$
- $\text{overlap}("\text{tagg}", "\text{gga}") = 2$
- $|"\text{gga}"| - 2 = 1$
**Reducing SS to TSP**

Build a graph \( D = (V, A) \)
- One vertex for each \( s_i \) and one for special “null” node, \( \Lambda \)
- A directed edge from \( s_i \) to \( s_j \) has weight \( w_{i,j} = |s_j| - \text{overlap}(s_i, s_j) \)
- \( w_{\Lambda,i} = |s_i| \rightarrow \text{no overlap, maximal increase} \)
- \( w_{i,\Lambda} = 0 \rightarrow \text{, no overlap, no increase} \)
**Reducing SS to TSP**

\[ S = \{ \text{catt}, \text{tta}, \text{acat} \} \]
Reducing SS to TSP

This tour ≡ \textit{cattacat\_ta}

Length 10

This tour ≡ \textit{catt\_acat}

Length 8
REDUCING SS TO TSP

- TSP considers all Hamiltonian paths (hence is brute force)

- TSP finds the minimum cost Hamiltonian path.
  - Total cost is the length of the SS

- TSP is also NP-hard.
SYNOPSIS

- A real world problem: Gene sequencing.
- The computational problem.
- Algorithms:
  - The Brute Force Algorithm
  - Reducing SS to TSP
  - The Greedy Algorithm
**The Greedy Technique**

Given a sequence of steps to be made, at each decision point

- make a **locally optimal** decision
- without ever backtracking on previous decisions.

- Greedy is a quite general algorithmic paradigm.
- In general, it does not get the best solution.
  - But it does work for some other problems (e.g., Huffman Encoding, MST)
THE GREEDY APPROXIMATION TO SS

- Start with a pair of strings with maximal overlap (Why?)
- Continue with strings that adds the least extension every time.
  - This is the locally optimal decision!

- We already defined $\text{overlap}(s_i, s_j)$
- $\text{join}(s_i, s_j) \equiv$ concatenate $s_j$ to $s_i$ and remove overlap.
  - $\text{join}($"tagg","gga") = ”tagga"
THE GREEDY APPROXIMATION TO SS

GREEDY APPROX SS

fun greedyApproxSS(S) =
if |S| = 1 then s_0
else let
val O = {(overlap(s_i, s_j), s_i, s_j) : s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
val (o, s_i, s_j) = maxval <#1 O
val s_k = join(s_i, s_j)
val S' = (\{s_k\} ∪ S) \{s_i, s_j\}
in
  greedyApproxSS(S')
end

• S’ gets smaller by one string after each recursion.
The Greedy Approximation to SS

- GreedyApproxSS returns a string with length within 3.5 times the shortest string.
- Conjectured to return within a factor of 2.
- Does much better in practice.
The Greedy Approximation to SS

- Let's do an example.
- \( S = \{ \text{catt}, \text{gagtat}, \text{tagg}, \text{tta}, \text{gga}, \} \)
SUMMARY

- Interfaces vs Implementations
  - Precise interfaces are key.

- The Shortest Superstring Problem
  - The brute-force approach
  - Reduction to TSP
  - Approximate solution using greedy paradigm
SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm
Cost Models

- **Sequential**: the Random Access Machine (RAM) model
- **Parallel**: the Parallel RAM model
- **Parallel**: the 15-210 model
  - Tied to high-level programming constructs – operational semantics
  - Think parallel!
15-210 Cost Model

- $W(e)$: Work needed to evaluate $e$
- $S(e)$: Span of the evaluation of $e$

Parameterized with relevant problem size measures.

Asymptotic Models
  - How do algorithms scale to large problems!
**PARAMETERIZATION**

- We measure the size of representation of the input.

- **Sorting**: Number of items to sort
- **Map, Reduce**: Number of items in the sequence
- **Graph Problems**: Number of Nodes, Edges
- **Searching**: Number of items in the database
- **Matrix operations**: Number of rows and columns
- **Prime number testing**: Size – number of bits to represent the number (not the value!)
- **Computing $n^{th}$ Fibonacci number**: Size – number of bits to represent the number (not the value!)
**Rules of Composition**

- \((e_1, e_2)\): Sequential Composition
  - Add work and span
- \(e_1 || e_2\): Parallel Composition
  - Add work but take the maximum span

![Diagram](image-url)
# Rules of Composition

<table>
<thead>
<tr>
<th>e</th>
<th>W(e)</th>
<th>S(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>op e</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(e₁, e₂)</td>
<td>(1 + W(e₁) + W(e₂))</td>
<td>(1 + S(e₁) + S(e₂))</td>
</tr>
<tr>
<td>(e₁</td>
<td></td>
<td>e₂)</td>
</tr>
</tbody>
</table>

\[
\text{let val } x = e₁ \\
\text{in } e₂ \text{ end}
\]

\[
\begin{align*}
&1 + W(e₁) + W(e₂) \\
&\quad \quad \quad \quad \quad \quad W(e₂[\text{Eval}(e₁)/x]) \\
&\quad \quad \quad \quad \quad \quad S(e₂[\text{Eval}(e₁)/x])
\end{align*}
\]

\[
\begin{align*}
\{f(x) | x \in A\} & \quad 1 + \sum_{x \in A} W(f(x)) \quad 1 + \max_{x \in A} S(f(x))
\end{align*}
\]
RULES OF COMPOSITION

- \( \{ f(x) \mid x \in A \} \equiv \text{map } f \ A \)

- \( W(\text{map } f \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \sum_{i=0}^{n-1} W(f(s_i)) \)

- \( S(\text{map } f \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \max_{i=0}^{n-1} S(f(s_i)) \)
**Upper and Lower Bounds**

- **Upper bound**: The maximum asymptotic work (and span) that a given algorithm needs for all inputs of size $n$.

- **Lower bound**: The minimum asymptotic work (and span) that any algorithm for a problem needs for all inputs of size $n$. 
SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm
PARALLELISM

For a given $W$ and $S$, what is the maximum number of processors you can utilize?

- $\mathbb{P} = \frac{W}{S}$

Why?

Mergesort has $W = \theta(n \log n)$ and $S = \theta(\log^2 n)$

- $\mathbb{P} = \theta\left(\frac{n}{\log n}\right)$

The larger the problem is, the higher the parallelism
DESIGNING PARALLEL ALGORITHMS

- Keep work as low as possible
  - No unnecessary computation

- Keep span as low as possible
  - Hence get high-parallelism
SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm
Mapping from a computation graph to processors
A greedy scheduler will schedule a ready task on an available processor.
A Lower Bound

Let $T_p$ be the “time” needed when using $p$ processors,

$$\max\left(\frac{W}{p}, S\right) \leq T_p$$

Why?
AN UPPER BOUND

- With $p$ processors

$$T_p < \frac{W}{p} + S$$

- Why?
Tying Things Together

- Speed-up is $\frac{W}{T_p}$
  - Maximum possible speed-up is $p$.

\[
T_p < \frac{W}{p} + S \\
= \frac{W}{p} + \frac{W}{P} \\
= \frac{W}{p} \left(1 + \frac{p}{P}\right) \\
\]

- $P \gg p \rightarrow$ near perfect parallelism
SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm
Costs for the Brute Force SS Algorithm

- The brute-force algorithm
  - For each permutation
    - Remove overlaps
    - Stitch strings
  - Output (one of) the shortest string(s)

- \( \text{overlap}(s_i, s_j) \) will be needed many times.
  - Preprocess \( S \) once and store overlaps as a table
    - What prefix to remove
    - Increase in length
Preprocessing – Inputs

- A set $S$ is $n$ strings, $s_1, s_2, \cdots, s_n$
- Define

$$m = \sum_{i=1}^{n} |s_i|$$

and observe $n \leq m$. 
Preprocessing a Pair

- Work and span for preprocessing one pair, $s_i$ and $s_j$?
  - $W = O(|s_i| \cdot |s_j|)$ Why?
  - $S = O(\log(|s_i| + |s_j|))$ Why?
\[ W_{ov} \leq \sum_{i=1}^{n} \sum_{j=1}^{n} W(\text{overlap}(s_i, s_j)) \]
\[ = \sum_{i=1}^{n} \sum_{j=1}^{n} O(|s_i||s_j|) \]
\[ \leq \sum_{i=1}^{n} \sum_{j=1}^{n} (k_1 + k_2 |s_i||s_j|) \]
\[ = \sum_{i=1}^{n} \sum_{j=1}^{n} k_1 + \sum_{i=1}^{n} \sum_{j=1}^{n} (k_2 |s_i||s_j|) \]
\[ = k_1 n^2 + k_2 \sum_{j=1}^{n} |s_j|(\sum_{i=1}^{n} |s_i|) = k_1 n^2 + k_2 m^2 \in O(m^2) \]
All $s_i, s_j$ pairs can be processed in parallel.

\[
S_{ov} \leq \max_{i=1}^{n} \max_{j=1}^{n} S(\text{overlap}(s_i, s_j)) \\
\in O(\log m)
\]
Brute Force SS Algorithm

- **Work:**
  - $O(n)$ lookups each with $O(1)$ work. Why?
  - $n!$ permutations
  - $O(n \cdot n!) = O((n + 1)!)$
  - $W_{ov}$ can be ignored!

- **Span:**
  - All permutations can be done in parallel, but!
    
    ```
    func permutations S =
    if $|S| = 1$ then {S}
    else
      {append([s], p) : s in S, p in permutations(S\s)}
    ```
  - This has span $O(n)$. Why?
  - $S_{ov}$ can be ignored.
SYNOPSIS

- Cost Models
- Parallelism
- Scheduling
- Cost Analysis for the Shortest Super String Problem
  - The Brute Force Algorithm
  - The Greedy Algorithm
The Greedy SS Algorithm

1 fun greedyApproxSS(S) =
2   if |S| = 1 then s0
3   else let
4     val O = {(overlap(s_i, s_j), s_i, s_j) : s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
5     val (o, s_i, s_j) = maxval <#1 O
6     val s_k = join(s_i, s_j)
7     val S' = ({s_k} ∪ S)\{s_i, s_j}
8     in
9       greedyApproxSS(S')
10   end
The Greedy SS Algorithm

1 fun greedyApproxSS(S) =
2 if |S| = 1 then s0
3 else let
4  val O = {(overlap(s_i, s_j), s_i, s_j) : s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
5  val (o, s_i, s_j) = maxval <#1 O
6  val s_k = join(s_i, s_j)
7  val S' = ({s_k} ∪ S)\{s_i, s_j}
8  in
9   greedyApproxSS(S')
10 end

W_{ov} = O(m^2), \ S_{ov} = O(\log m)
### The Greedy SS Algorithm

1. \[
\text{fun greedyApproxSS}(S) =
\]
2. \[
\text{if } |S| = 1 \text{ then } s_0
\]
3. \[
\text{else let}
\]
4. \[
\text{val } O = \{(\text{overlap}(s_i, s_j), s_i, s_j) : s_i \in S, s_j \in S, s_i \neq s_j\}
\]
5. \[
\text{val } (o, s_i, s_j) = \text{maxval} < \#_1 O
\]
6. \[
\text{val } s_k = \text{join}(s_i, s_j)
\]
7. \[
\text{val } S' = (\{s_k\} \cup S)\backslash\{s_i, s_j\}
\]
8. \[
\text{in}
\]
9. \[
\text{greedyApproxSS}(S')
\]
10. \[
\text{end}
\]

- \(W_{\text{maxval}} = O(m^2), \ S_{\text{maxval}} = O(\log m)\)
- Why?
THE GREEDY SS ALGORITHM

fun greedyApproxSS(S) =
  if |S| = 1 then s0
  else let
    val O = {(overlap(s_i, s_j), s_i, s_j): s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
    val (o, s_i, s_j) = maxval <#1 O
    val s_k = join(s_i, s_j)
    val S' = ({s_k} ∪ S)\{s_i, s_j}
  in
  greedyApproxSS(S')
end

- No more than \( W = O(m^2) \), \( S = O(\log m) \)
- Why?
**The Greedy SS Algorithm**

```plaintext
fun greedyApproxSS(S) =
  if |S| = 1 then s0
  else let
    val O = {(overlap(s_i, s_j), s_i, s_j) : s_i ∈ S, s_j ∈ S, s_i ≠ s_j}
    val (o, s_i, s_j) = maxval <#1 O
    val s_k = join(s_i, s_j)
    val S' = ({s_k} ∪ S)\{s_i, s_j}
  in
  greedyApproxSS(S')
end
```

- At most \( n \) (sequential) calls to \( \text{greedyApproxSS} \)
  - Each with \( W = O(m^2), \ S = O(\log m) \)
- \( W_{\text{greedy}} = O(nm^2) \) and \( S_{\text{greedy}} = O(n \log m) \)
- Why?
SUMMARY

- Cost Models: Rules of Composition
- Parallelism and Scheduling
- Cost Analysis for the Shortest Super String Problem
  - Preprocessing for overlaps
  - The Brute Force Algorithm
  - The Greedy Algorithm
15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

Lecture 3
ALGORITHMIC TECHNIQUES AND DIVIDE-AND-CONQUER
SYNOPSIS

- Algorithmic Techniques
- Divide-and-Conquer
  - Analysis of Costs
- The Maximum Contiguous Subsequence Sum Problem
ALGORITHMIC TECHNIQUES

Brute Force

- Try all possibilities
- Almost always intractable
- Useful for testing small cases
- Code usually easy to write

Reducing one problem to another

- Transform the structure or the instance of a problem.
- Shortest Superstring $\rightarrow$ Traveling Salesperson Problem
- Apply algorithms for the new problem
**Inductive Techniques**

- Solve **one or more smaller problems** to solve the large problem.
- Techniques differ on
  - The number of subproblems
  - How subproblem solutions are used

- Divide-and-Conquer
- Greedy
- Contraction
- Dynamic Programming
Divide-and-Conquer

- Divide a problem of size $n$ into $k > 1$ problems
  - Sizes $n_1, n_2, \ldots, n_k$

- Solve each problem recursively.

- Combine the subproblem solutions.
**Greedy**

- Given a problem of size $n$

- Remove one (or more) elements using a **greedy** approach
  - Smallest, two smallest, nearest, lowest, etc.

- Solve the remaining smaller problem
  - Usually smaller by 1 or 2 items.
**CONTRACTION**

- Given a problem of size \( n \)
- Generate a significantly smaller (contracted) instance
  - e.g., of size \( n/2 \)
- Solve the smaller instance
- Use the result to solve the original problem.
- One recursive call instead of multiple!
Dynamic Programming

- Like Divide-and-Conquer

- Solutions to subproblems used multiple times!

- Compute once and store, and then reuse.
ADTs and Data Structures

- Techniques rely on Abstract Data Types (for functionality)
  - and on data structures that implement them (for costs)

- Sequences, Sets, Tables, Priority Queues, Graphs, Trees, ...
RANDOMIZATION

- Introduce randomness at a choice point
  - Quicksort: choose a pivot randomly

- Testing for primality
  - Miller-Rabin primality test
  - $\frac{3}{4}$ of numbers $< n$ are “witnesses” to $n$’s compositeness.
  - Randomly choose 100 numbers $< n$
  - $P(\text{Failing to find a witness}) = 1 - \left(\frac{1}{4}\right)^{100}$
  - $P(n \text{ is prime}) = 1 - \left(\frac{1}{4}\right)^{100} = 0.9999\ldots 9327\ldots$
SYNOPSIS

- Algorithmic Techniques
- Divide-and-Conquer
  - Analysis of Costs
- The Maximum Contiguous Subsequence Sum Problem
**Divide-and-Conquer**

- Very versatile.
- Easy to implement.
- Parallelizable
- Code follows the structure of a proof.
- Cost reasoning follows code structure.
  - Recurrences
STRENGTHENING THE PROBLEM

- Compute more than “superficially” needed.
- No increase to work or span.
- More efficient combine step.
- At the end, this extra information can be discarded.
**General Structure**

- **Base case(s)**
  - When problem small enough, use a different technique.
  - For example, in quicksort, switch to insertion sort to sort < 30 elements.

- **Inductive Step**
  - Divide into parts
    - Sometimes quite simple: e.g., mergesort
    - Sometimes a bit tricky: e.g., quicksort
  - Solve subproblems (in parallel)
  - Combine results
    - Sometimes quite simple: e.g., quicksort
    - Sometimes a bit tricky: e.g., mergesort

- Costs can be in the divide or combine steps or in both.
**General Structure**

\[ W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combine}}(n) \]

\[ S(n) = S_{\text{divide}}(n) + \max_{i=1}^{k} S(n_i) + S_{\text{combine}}(n) \]
SOLVING RECURRENCES

- Tree method (Brick method)
- Substitution method
**The Tree Method**

- Expand recurrence into a tree structure.

```
  \hspace{4cm} \begin{array}{c}
   \text{Cost of level 0} \\
   \text{Cost of level 1} \\
   \text{Cost of level 2}
  \end{array}
```

- Add/Max costs at levels.
The Tree Method

- Solve $W(n) = 2W(n/2) + O(n)$

- In general, solve

  $W(n) = 2W(n/2) + g(n)$

  where $g(n) \in O(f(n))$
### The Tree Method

- \( g(n) \in O(f(n)) \Rightarrow g(n) \leq c \cdot f(n) \)
  - For some \( c > 0, N_0 > 0 \) and \( n \geq N_0 \)

- \( g(n) \leq k_1 \cdot f(n) + k_2 \) for some \( k_1, k_2 \) and \( n \geq 1 \)
  - e.g., \( k_1 = c \) and \( k_2 = \sum_{i=1}^{N_0} |g(i)| \) (Why?)

- Solve \( W(n) \leq 2W(n/2) + k_1 \cdot n + k_2 \)
  - \( f(n) = n \) in our case.
The Tree Method

- Solving $W(n) \leq 2W(n/2) + k_1 \cdot n + k_2$

Questions:
- Number of levels in the tree?
- Problem size at level $i$?
- Cost for each node at level $i$?
- Number of nodes at level $i$?
- Total cost at level $i$?
The Tree Method

- Total cost at level $i$ is at most
\[ 2^i \cdot \left( k_1 \frac{n}{2^i} + k_2 \right) = k_1 \cdot n + 2^i \cdot k_2 \]

- Total cost over all levels is
\[
W(n) \leq \sum_{i=0}^{\log_2 n} (k_1 \cdot n + 2^i \cdot k_2) \\
= k_1 n(1 + \log_2 n) + k_2 (2^0 + 2^1 + \cdots + 2^{\log_2 n}) \\
\leq k_1 n(1 + \log_2 n) + 2k_2 n \quad \text{(Why?)} \\
\in O(n \log n)
\]
THE BRICK METHOD

- Look at the cost structure at the levels of the cost tree
  - Leaves dominated
  - Balanced
  - Root dominated
**Leaves-Dominated Cost Trees**

- For some $\rho > 1$, for all levels $i$

$$\text{cost}_{i+1} \geq \rho \cdot \text{cost}_i$$

- Overall cost is $O(\text{cost}_d)$ where $d$ is the depth.
**Balanced Cost Trees**

- All levels have about the same cost

  ++++++++++
  ++++++++++
  ++++++++++
  ++++++++++

- Overall cost is $O(d \cdot \max_i \text{cost}_i)$ where $d$ is the depth.
ROOT-DOMINATED COST TREES

- For some $\rho < 1$, for all levels $i$

$$\text{cost}_{i+1} \leq \rho \cdot \text{cost}_i$$

- Overall cost is $O(\text{cost}_0)$ where $d$ is the depth.
The Brick Method

What type of a cost tree is this?

\[ k_1 n + k_2 \]
\[ k_1 (n/2) + k_2 \]
\[ k_1 (n/4) + k_2 \]
\[ k_1 (n/4) + k_2 \]
\[ k_1 (n/4) + k_2 \]
\[ k_1 (n/4) + k_2 \]

\[ k_1 n + k_2 \]
\[ k_1 n + 2k_2 \]
\[ k_1 n + 4k_2 \]
SYNOPSIS

- Algorithmic Techniques
- Divide-and-Conquer
  - Analysis of Costs
- The Maximum Contiguous Subsequence Sum Problem
THE MCSS PROBLEM

THE MAXIMUM CONTIGUOUS SUBSEQUENCE SUM PROBLEM

- Given a sequence of numbers $S = \langle s_1, \ldots, s_n \rangle$,
- Find

$$mcss(S) = \max_{1 \leq i \leq j \leq n} \left\{ \sum_{k=i}^{j} s_k \right\}$$

- $S = \langle 0, -1, 2, -1, 4, -1, 0 \rangle$, $mcss(S) = 5$
- How many possible subsequences are there?
- All positive numbers?

ALGORITHMIC TECHNIQUES AND DIVIDE-AND-CONQUER
CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS
FALL 2013
**Brute Force Algorithm**

- Compute the sum of all $O(n^2)$ possible subsequences (in parallel)
  - Use plus reduce
- Subsequence $(i, j)$ needs
  - $O(j - i)$ work (Why?)
  - $O(\log(j - i))$ span (Why?)

\[
W(n) = 1 + \sum_{1 \leq i \leq j \leq n} W_{\text{reduce}}(j - i) \leq 1 + n^2 \cdot W_{\text{reduce}}(n)
\]
\[
= 1 + n^2 \cdot O(n) \in O(n^3)
\]
\[
S(n) = 1 + \max_{1 \leq i \leq j \leq n} S_{\text{reduce}}(j - i) \leq 1 + S_{\text{reduce}}(n) \in O(\log n)
\]
Brute Force Algorithm

- Compute maximum over all $O(n^2)$ sums
  - Use max reduce
  - Needs $O(n^2)$ work and $O(\log n)$ span
  - Can be ignored (Why?)

- Total costs for brute force are:
  - $O(n^3)$ work
  - $O(\log n)$ span
Let’s solve \( S = \langle -2, -1, 2, 3, 2, -2 \rangle \)

Is this right?

How do we combine subproblem results?
Recursion handles

- When mcss(S) subsequence is in the left.
- When mcss(S) subsequence is in the right.

What happens when mcss(S) spans across the divide point?

Largest Sum Suffix  Largest Sum Prefix

Largest Sum Prefix

Maximum sum across the divide
fun \( mcss(s) = \)

\[
\begin{align*}
\text{case } (\text{showt } s) \quad &\text{of} \\
\text{EMPTY } = -\infty \quad &| \\
\text{ELT}(x) = x \quad &| \\
\text{NODE}(L, R) = \\
\quad \text{let val } (m_L, m_R) = (mcss(L) \| mcss(R)) \\
\quad \text{val } m_A = \text{bestAcross}(L, R) \\
\quad \text{in } \max\{m_L, m_R, m_A\}
\end{align*}
\]

- \( W(n) = 2W(n/2) + O(n) \) (Why?) \( \rightarrow W(n) \in O(n \log n) \)
- \( S(n) = S(n/2) + O(\log n) \) (Why?) \( \rightarrow S(n) \in O(\log^2 n) \)
**DIVIDE-AND-CONQUER – II**

**IMPORTANT QUESTIONS**

- Can we do better than $O(n \log n)$ work?

- What part of the divide-and-conquer is the bottleneck?
  - Combine takes linear work? (Why?)

- How can we improve?
Divide-and-Conquer – II

- The answers lie here

Largest Sum Suffix  Largest Sum Prefix

L  R

Maximum sum across the divide

- Strengthen the subproblems
  - Compute additional information
**DIVIDE-AND-CONQUER – II**

Left Subproblem

- $\text{mps}_L$ = maximum prefix sum
- $\text{mss}_L$ = maximum suffix sum
- $\text{mcss}_L$ = maximum cross sum
- $\text{Total}_L$

Diagram illustrates the components of a subproblem within the context of divide-and-conquer algorithmic techniques.
**DIVIDE-AND-CONQUER – II**

- **Left Subproblem**
  - $mcss_L$
  - $mps_L$
  - $mss_L$
  - Total $L$

- **Right Subproblem**
  - $mcss_R$
  - $mps_R$
  - $mss_R$
  - Total $R$
DIVIDE-AND-CONQUER – II

Left Subproblem

Right Subproblem

mcss

Total

mps

mss

Algorithmic Techniques and Divide-and-Conquer
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
**DIVIDE-AND-CONQUER – II**

Left Subproblem

- \( mcss_L \)
- \( mps_L \)
- \( mss_L \)
- \( \text{Total}_L \)

Right Subproblem

- \( mcss_R \)
- \( mps_R \)
- \( mss_R \)
- \( \text{Total}_R \)

Total

- \( mcss \)
- \( mps \)
- \( mss \)
- \( \text{Total} \)

\[
\text{Total} = \text{Total}_L + \text{Total}_R
\]

\[
mcss = \max ( mcss_L, mcss_R, mss_L + mps_R )
\]

\[
mps = \max ( mps_L, \text{Total}_L + mps_R )
\]

\[
mss = \max ( mss_L + \text{Total}_R, mss_R )
\]
fun \(\text{mcss}(a) = \) let
fun \(\text{mcss}'(a)\) case (\(\text{showt} a\)) of
   EMPTY = (\(-\infty, -\infty, -\infty, 0\))
   | ELT(\(x\)) = (\(x, x, x, x\))
   | NODE(\(L, R\)) =
      let
         val \((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2)\) = \(\text{mcss}(L) \parallel \text{mcss}(R)\) in
         (\(\max(s_1 + p_2, m_1, m_2)\), \(\max(p_1, t_1 + p_2)\), \(\max(s_1 + t_2, s_2)\), \(t_1 + t_2\))
      end
   val \((m, p, s, t) = \text{mcss}'(a)\) in \(m\) end
**Cost Analysis**

1. `fun mcss(a) =`  
2. `let`  
3. `fun mcss′(a)`  
4. `case (showt a)`  
5. `of EMPTY = (−∞, −∞, −∞, 0)`  
6. `| ELT(x) = (x, x, x, x)`  
7. `| NODE(L, R) =`  
8. `let`  
9. `val ((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2)) = (mcss(L) || mcss(R))`  
10. `in`  
11. `(max(s_1 + p_2, m_1, m_2), max(p_1, t_1 + p_2), max(s_1 + t_2, s_2), t_1 + t_2)`  
12. `end`  
13. `val (m, p, s, t) = mcss′(a)`  
14. `in m end`  

- **Assuming** `showt` **has** \(O(\log n)\) **work and span.**  
  - \(W(n) = 2W(n/2) + O(\log n)\)  
  - \(S(n) = S(n/2) + O(\log n)\)
**Cost Analysis**

- \( W(n) = 2W(n/2) + O(\log n) \)

- \( W(n) \leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i) \)
**Substitution Method**

- Solve $W(n) \leq 2W(n/2) + k \cdot \log n$
  - $k > 0$
  - $W(n) \leq k$ for $n \leq 1$

- Guess $W(n) \leq \kappa_1 n - \kappa_2 \log n - \kappa_3$
  - Need to find $\kappa_1, \kappa_2, \text{ and } \kappa_3$.

- Base case: $W(1) \leq k \Rightarrow \kappa_1 - \kappa_3 \leq k$
**Inductive Step**

\[
W(n) \leq 2W\left(\frac{n}{2}\right) + k \cdot \log n
\]

\[
\leq 2\left(\kappa_1 \frac{n}{2} - \kappa_2 \log \left(\frac{n}{2}\right) - \kappa_3\right) + k \cdot \log n
\]

\[
= \kappa_1 n - 2\kappa_2 (\log n - 1) - 2\kappa_3 + k \cdot \log n
\]

\[
= (\kappa_1 n - \kappa_2 \log n - \kappa_3) + (k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3)
\]

\[
\leq \kappa_1 n - \kappa_2 \log n - \kappa_3
\]

**Choose** \(\kappa_2 = k\) and \(2\kappa_2 - \kappa_3 \leq 0\) (Why?)

**For example,** \(\kappa_2 = k, \kappa_1 = 3k, \kappa_3 = 2k\) satisfies the constraints.
SUMMARY

- Algorithmic Paradigms
- Divide-and-Conquer
  - General Form
  - Cost Analysis
  - Tree and Brick Methods
  - Substitution Method
- Maximum Contiguous Subsequence Problem
  - Brute Force
  - Divide-and-Conquer
  - Divide-and-Conquer with Subproblem Strengthening
SYNOPSIS

- The Euclidian Travelling Salesperson Problem
- Divide-and-Conquer Heuristic Algorithm
- Analysis of Costs
THE EUCLIDIAN TSP

- Given a set of points in a $n$-dimensional Euclidian space.
  - What is a Euclidian space?

- Find the shortest Hamiltonian cycle.
  - What is a Hamiltonian cycle?

- We get a planar Euclidian Traveling Salesperson Problem when the points are in 2-dimensional space.
The Planar TSP
SYNOPSIS

The Euclidian Travelling Salesperson Problem
Divide-and-Conquer Heuristic Algorithm
Analysis of Costs
What is a heuristic?

Approximation algorithm
- Resulting tour length is guaranteed to be close to the actual minimum tour length
- If you spend enough work (but polynomial).

The Divide-and-Conquer does work both before and after the recursive calls.
Assume $P_\ell$ and $P_r$ have tour lengths $T_\ell$ and $T_r$.

Tour length for the combination?
A DIVIDE-AND-CONQUER HEURISTIC

\[ T_{\ell} + T_{r} + \| \mathbf{u}_{\ell} - \mathbf{u}_{r} \| + \| \mathbf{v}_{\ell} - \mathbf{v}_{r} \| - \| \mathbf{u}_{\ell} - \mathbf{v}_{\ell} \| - \| \mathbf{u}_{r} - \mathbf{v}_{r} \| \]

Add these

Subtract these
A **DIVIDE-AND-CONQUER HEURISTIC**

\[ T_{\ell} + T_r + \| u_{\ell} - v_r \| + \| v_{\ell} - u_r \| - \| u_{\ell} - v_{\ell} \| - \| u_r - v_r \| \]

Add these

Subtract these
A Divide-and-Conquer Heuristic

- Try all pairs of edges $e_\ell$ from $P_\ell$ and $e_r$ from $P_r$
  - How many pairs are there?

- For each pair of edges, find the smallest increase.

- Then combine the small tours into a large tour.
A DIVIDE-AND-CONQUER HEURISTIC

fun eTSP(P) = 
case (|P|) of 
    0, 1 ⇒ raise TooSmall
    2 ⇒ {(P[0], P[1]), (P[1], P[0])}
    n ⇒ let 
        val (P₁, Pᵣ) = splitLongestDim(P) 
        val (L, R) = (eTSP(P₁) || eTSP(Pᵣ)) 
        val (c, (e'L, e'r)) = 
            minval <#1 {(swapCost(e'L, e'r), (e'L, e'r)) : e'L ∈ L, e'r ∈ R} 
        in 
        swapEdges(append(L, R), e'L, e'r) 
    end
A DIVIDE-AND-CONQUER HEURISTIC

fun eTSP(P) =
  case (|P|)
  of  0, 1 => raise TooSmall
       |  2  => {(P[0], P[1]), (P[1], P[0])}
       | n  => let
         val (P_ell, P_r) = splitLongestDim(P)
         val (L, R) = (eTSP(P_ell) || eTSP(P_r))
         val (c, (e'_ell, e'_r)) =
           minval <#1 {(swapCost(e_ell, e_r), (e_ell, e_r)) : e_ell \in L, e_r \in R}
          in
           swapEdges(append(L, R), e'_ell, e'_r)
        end
**Splitting the Points**

- Split at the median along the longer spread dimension.

---

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013

Divide-and-Conquer Continued
**Swap Cost**

- Given $e_ℓ = (u_ℓ, v_ℓ) \in L$ and $e_r = (u_r, v_r) \in R$

\[
\text{swapCost}((u_ℓ, v_ℓ), (u_r, v_r)) = \text{Cost Added} - \text{Cost Removed}
\]

\[
\text{Cost Added} = \min(\|u_ℓ - u_r\| + \|v_ℓ - v_r\|, \\
\|u_ℓ - v_r\| + \|v_ℓ - u_r\|)
\]

\[
\text{Cost Removed} = \|u_ℓ - v_ℓ\| + \|u_r - v_r\|
\]
Swapping Edges

- \texttt{swapEdges(append(L,R), e'_\ell, e'_r)}
- Appends the Tour edge lists from subproblems
- Then removes and adds appropriate edges.
SYNOPSIS

- The Euclidian Travelling Salesperson Problem
- Divide-and-Conquer Heuristic Algorithm
- Analysis of Costs
fun eTSP(P) =
  case (|P|)
    of 0, 1 ⇒ raise TooSmall
    | 2 ⇒ {(P[0], P[1]), (P[1], P[0])}
    | n ⇒ let
      val (Pℓ, Pr) = splitLongestDim(P) O(n) work O(log n) span (Why?)
      val (L, R) = (eTSP(Pℓ) || eTSP(Pr)) 2W(n/2) work S(n/2) span
      val (c, (e′ℓ, e′r)) =
        minval <#1 {swapCost(eℓ, e_r, (e′ℓ, e_r)) :
          eℓ ∈ L, e_r ∈ R} O(n²) work O(log n) span (Why?)
      in
        swapEdges(append(L, R), e′ℓ, e′r) O(log n) span (Why?)
    end
Cost Analysis

\[ W(n) = 2W(n/2) + O(n^2) \]

\[ S(n) = S(n/2) + O(\log n) \]

\[ S(n) \in O(\log^2 n) \]
Cost Analysis

- Solve (directly)

\[
W(n) = 2W(n/2) + k \cdot n^{1+\varepsilon}
\]

for constant \( \varepsilon > 0 \).
- Depth is \( \log_2 n \) (Is this technically right?)
- At level \( i \), we have \( 2^i \) nodes each costing \( k \cdot (n/2^i)^{1+\varepsilon} \)

\[
W(n) = \sum_{i=0}^{\log n} k \cdot 2^i \cdot \left( \frac{n}{2^i} \right)^{1+\varepsilon}
\]

\[
= k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\log n} 2^{-i \cdot \varepsilon}
\]

\[
\leq k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\infty} 2^{-i \cdot \varepsilon}
\]

\[
W(n) \in O(n^{1+\varepsilon}) \quad \text{(Why?)}
\]
Euclidian Traveling Salesperson Problem
  ▶ Divide-and-Conquer Heuristic
  ▶ Processing before and after the subproblem solutions.

Cost Analysis
15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

Lecture 5
DATA ABSTRACTION AND SEQUENCES
SYNOPSIS

- Abstractions and Implementations
  - Meldable Priority Queues
- The Sequence ADT
- The \textit{scan} operation
- Introduction to \textit{contraction}
## Abstractions and Implementations

<table>
<thead>
<tr>
<th>Functions</th>
<th>Data</th>
<th>Abstraction</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Problem</td>
<td>Algorithm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abstract Data Type</td>
<td>Data Structure</td>
</tr>
</tbody>
</table>
MELDABLE PRIORITY QUEUES

- Priority Queues
  - Insert an item – *insert*
  - Return and delete the item with the minimum priority – *deleteMin*
- Meldable Priority Queue
  - Join two priority queues into one – *meld*
**Meldable Priority Queues**

- $S$ is a totally ordered set (integers, strings, reals, ...).
- $T$ is a type representing *subsets* of $S$.

\[
\begin{align*}
\text{empty} & : T = \{\} \\
\text{insert}(S, e) & : T \times S \rightarrow T = S \cup \{e\} \\
\text{deleteMin}(S) & : T \rightarrow T \times (S \cup \{\bot\}) = \begin{cases} 
(S, \bot) & S = \{\} \\
(S \setminus \{\min S\}, \min S) & \text{otherwise}
\end{cases} \\
\text{meld}(S_1, S_2) & : T \times T \rightarrow T = S_1 \cup S_2
\end{align*}
\]
MPQ Definition in SML

signature MPQ
sig
  struct S : ORD
  type t
  val empty : t
  val insert : t * S.t -> t
  val deleteMin : t -> t * S.t option
  val meld : t * t -> t
end

- No semantics, only the types.
MPQ: COST SPECIFICATIONS

Implementation 1:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(S, e)</td>
<td>O(</td>
</tr>
<tr>
<td>deleteMin(S)</td>
<td>O(1)</td>
</tr>
<tr>
<td>meld(S₁, S₂)</td>
<td>O(</td>
</tr>
</tbody>
</table>

What is the underlying data structure? Sorted Array
meld is actually an array merge.
MPQ: COST SPECIFICATIONS

- Implementation 2:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(S, e)</td>
<td>$O(\log</td>
</tr>
<tr>
<td>deleteMin(S)</td>
<td>$O(\log</td>
</tr>
<tr>
<td>meld(S₁, S₂)</td>
<td>$O(</td>
</tr>
</tbody>
</table>

- What is the underlying data structure? Heaps
### MPQ: Cost Specifications

- **Implementation 3:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(S, e)</code></td>
<td>`O(log</td>
</tr>
<tr>
<td><code>deleteMin(S)</code></td>
<td>`O(log</td>
</tr>
<tr>
<td><code>meld(S_1, S_2)</code></td>
<td>`O(log(</td>
</tr>
</tbody>
</table>

- Later!
ABSTRACTIONS AND IMPLEMENTATIONS

- The Abstract Data Type
  - Functionality
  - Correctness
- The Cost Specification
  - Multiple Cost Specifications
  - We only need these to do cost analysis.
- Underlying Data Structure
  - Multiple Data Structures
A relation is a set of ordered pairs.
   - First from set $A$, second from set $B$

A relation $\rho \subseteq A \times B$.

A function is a relation $\rho$, where for every $a \in A$ there is only one $b$ such that $(a, b) \in \rho$.

A sequence is a function where 
$A = \{0, \ldots, n - 1\}$ for some $n \in \mathbb{N}$. 
The Sequence ADT – Functionality

A sequence is a type $S_\alpha$ representing functions from $\{0, \ldots, n-1\}$ to $\alpha$.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>$S_\alpha$</td>
<td>${}$</td>
</tr>
<tr>
<td>length($A$)</td>
<td>$S_\alpha \rightarrow \mathbb{N}$</td>
<td>$</td>
</tr>
<tr>
<td>singleton($v$)</td>
<td>$\alpha \rightarrow S_\alpha$</td>
<td>${(0, v)}$</td>
</tr>
<tr>
<td>nth($A, i$)</td>
<td>$S_\alpha \rightarrow \alpha$</td>
<td>$A(i)$</td>
</tr>
<tr>
<td>map($f, A$)</td>
<td>$(\alpha \rightarrow \beta) \times S_\alpha \rightarrow S_\beta$</td>
<td>${(i, f(v)) : (i, v) \in A}$</td>
</tr>
<tr>
<td>tabulate($f, n$)</td>
<td>$(\mathbb{N} \rightarrow \alpha) \times \mathbb{N} \rightarrow S_\alpha$</td>
<td>${(i, f(i)) : i \in {0, \ldots, n-1}}$</td>
</tr>
<tr>
<td>take($A, n$)</td>
<td>$S_\alpha \times \mathbb{N} \rightarrow S_\alpha$</td>
<td>${(i, v) \in A \mid i &lt; n}$</td>
</tr>
<tr>
<td>drop($A, n$)</td>
<td>$S_\alpha \times \mathbb{N} \rightarrow S_\alpha$</td>
<td>${(i - n, v) : (i, v) \in A \mid i \geq n}$</td>
</tr>
<tr>
<td>append($A, B$)</td>
<td>$S_\alpha \times S_\alpha \rightarrow S_\alpha$</td>
<td>$A \cup {(i +</td>
</tr>
</tbody>
</table>
## The Sequence ADT – Cost Specs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work Complexity</th>
<th>Span Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>length(T)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>nth(T)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>append(S_1, S_2)</code></td>
<td>$O(</td>
<td>S_1</td>
</tr>
</tbody>
</table>
# The Sequence ADT – Cost Specs

<table>
<thead>
<tr>
<th>ArraySequence</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabulate $f \ n$</td>
<td>$O\left(\sum_{i=0}^{n} W(f(i))\right)$</td>
<td>$O\left(\max_{i=0}^{n} S(f(i))\right)$</td>
</tr>
<tr>
<td>map $f \ S$</td>
<td>$O\left(\sum_{s \in S} W(f(s))\right)$</td>
<td>$O\left(\max_{s \in S} S(f(s))\right)$</td>
</tr>
</tbody>
</table>
## The Sequence ADT – Cost Specifications

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>length($T$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>nth($T$)</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>append($S_1$, $S_2$)</td>
<td>$O(\log(</td>
<td>S_1</td>
</tr>
</tbody>
</table>
### The Sequence ADT – Cost Specifications

<table>
<thead>
<tr>
<th>TreeSequence</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tabulate</strong> $f \ n$</td>
<td>$O\left(\sum_{i=0}^{n} W(f(i))\right)$</td>
<td>$O\left(\log n + \max_{i=0}^{n} S(f(i))\right)$</td>
</tr>
<tr>
<td><strong>map</strong> $f \ S$</td>
<td>$O\left(\sum_{s \in S} W(f(s))\right)$</td>
<td>$O\left(\log</td>
</tr>
</tbody>
</table>
**Some Notational Conventions**

- $S_i$: The $i^{th}$ element of sequence $S$
- $|S|$: The length of sequence $S$
- $\langle \rangle$: The empty sequence
- $\langle v \rangle$: A sequence with a single element $v$
- $\langle i, \ldots, j \rangle$: A sequence of integers starting at $i$ and ending at $j \geq i$.
- $\langle e : p \in S \rangle$: Map the expression $e$ to each element $p$ of sequence $S$.
  The same as "\texttt{map (fn p ⇒ e) S}" in ML.
- $\langle p \in S \mid e \rangle$: Filter out the elements $p$ in $S$ that satisfy the predicate $e$.
  The same as "\texttt{filter (fn p ⇒ e) S}" in ML.

- More examples are given in the “Syntax and Costs” document.
The Scan Operation

- Related to reduce.

\[ \text{scan } f \ | \ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha \text{ seq} \]

\[ \to (\alpha \text{ seq } \times \alpha) \]

- \( I \) is the identity value
- \( f \) is an (associative) function
- \( S \) is a sequence
- Produces \( \langle I, f(I, S_0), f(f(I, S_0), S_1), \ldots \rangle \) and \( \text{reduce } f \ | \ S \)
  - \( \text{scan } + 0 \langle 2, 1, 4, 6 \rangle = (\langle 0, 2, 3, 7 \rangle, 13) \)
The Scan Operation

- \textit{scan} computes prefix sums.

\begin{verbatim}
1 fun scan f I S =
2   (⟨reduce f I (take(S,i)) : i ∈ ⟨0,... n − 1⟩⟩,
3    reduce f I S)
\end{verbatim}

- \(S\) has \(n\) elements
- Apply \textit{reduce} to each prefix of \(S\) of \(i\) elements, \(0 \leq i \leq n − 1\)
  - Gives you the \(α\) \textit{seq} part
- Apply \textit{reduce} to \(S\)
  - Gives you the \(α\) part
- So you get \((α\texttt{ seq} → α)\)
**The Scan Operation**

\[
\text{scan } + 0 \langle 2, 1, 3 \rangle = (\langle \text{reduce } + 0 \langle \rangle, \\
\text{reduce } + 0 \langle 2 \rangle, \\
\text{reduce } + 0 \langle 2, 1 \rangle \rangle \\
\text{reduce } + 0 \langle 2, 1, 3 \rangle) \\
= (\langle 0, 2, 3 \rangle, 6)
\]

- This is obviously not efficient!
- We will see how to do this with

\[
W(\text{scan } f / S) = O(|S|) \\
S(\text{scan } f / S) = O(\log |S|)
\]
The Inclusive Scan Operation

- reduce all prefixes ending at position $i$, $0 \leq i \leq n - 1$

$$\text{scan}I + 0 \langle 2, 1, 3 \rangle = \langle 2, 3, 6 \rangle$$
**THE MAXIMUM CONTIGUOUS SUBSEQUENCE SUM PROBLEM**

- Given a sequence of numbers $S = \langle s_1, \ldots, s_n \rangle$,
- Find

$$mcss(S) = \max_{1 \leq i \leq j \leq n} \left\{ \sum_{k=i}^{j} s_k \right\}$$

- $S = \langle 0, -1, 2, -1, 4, -1, 0 \rangle$, $mcss(S) = 5$
Using Scan in the MCSS Prob.

Consider $S = \langle 1, -2, 3, -1, 2, -3 \rangle$

Let $X = scan I + 0 S = \langle 1, -1, 2, 1, 3, 0 \rangle$

What is $X_j - X_i$ for $j > i$?

- $\sum_{k=i+1}^j S_k$
- $X_4 - X_0 = 3 - 1 = 2$
Define $R_j$ as the maximum sum that starts at some $i$ and ends at $j > i$.

\[
R_j = \max_{i=0}^{j} \sum_{k=i}^{j} S_k \\
= \max_{i=0}^{j} (X_j - X_{i-1}) \\
= X_j + \max_{i=0}^{j} (-X_{i-1}) \\
= X_j + \max_{i=0}^{j-1} (-X_i) = X_j - \min_{i=0}^{j-1} X_i \quad \text{(Why?)}
\]
Using `scan` in the MCSS Prob.

\[ R_j = X_j - \min_{i=0}^{j-1} X_i \]

- You need \( X_j \) and the minimum previous \( X_i, i < j \)
  - can be done by a minimum `scan`

\[
(M, \_ ) = \operatorname{scan} \min 0 X = (\langle 0, 0, -1, -1, -1, -1 \rangle, -1)
\]

\[
R = \langle X_j - M_j : 0 \leq j < \lvert S \rvert \rangle = \langle 1, -1, 3, 2, 4, 1 \rangle
\]
Let’s Recap

- Given $S = \langle 1, -2, 3, -1, 2, -3 \rangle$
- We computed $X$ with a $+ \text{scan}_I$.
  - $X = \langle 1, -1, 2, 1, 3, 0 \rangle$
- We computed $M$ with a $\text{min scan}$
  - $M = \langle 0, 0, -1, -1, -1, -1 \rangle$
- We computed $R = \langle X_j - M_j : 0 \leq j < |S| \rangle$
  - $R = \langle 1, -1, 3, 2, 4, 1 \rangle$
- A final max $\text{reduce}$ on $R$ gives us the MCSS, 4.
Using \texttt{Scan} in the MCSS Prob.

\begin{verbatim}
fun MCSS(S) =
  let
    val X = scanI + 0 S
    val (M, _) = scan min 0 X
  in
    max \langle X_j - M_j : 0 \leq j < |S| \rangle
  end
\end{verbatim}

- Work? $O(n)$
- Span? $O(\log n)$
Copy Scan

- Scan can also be used to pass information along a sequence.

\[ \langle \text{NONE}, \text{SOME(7)}, \text{NONE}, \text{NONE}, \text{SOME(3)}, \text{NONE} \rangle \]

\[ \downarrow \]

\[ \langle \text{NONE}, \text{NONE}, \text{SOME(7)}, \text{SOME(7)}, \text{SOME(7)}, \text{SOME(3)} \rangle \]

- Each element receives the nearest previous \text{SOME()} value.
- Easy to do sequentially with \text{iter}. 
Can we do this with \textit{scan}?

\[ f : \alpha \text{ option} \times \alpha \text{ option} \rightarrow \alpha \text{ option} \]

1. \textbf{fun} \; \texttt{copy}(a, b) =
2. \textbf{case} \; b \; \textbf{of}
3. \quad \texttt{SOME}(_, \_\_) \Rightarrow b
4. \quad \mid \texttt{NONE} \Rightarrow a

- Passes its right argument if it is \textit{SOME}, else passes its left argument.
- How do you show \texttt{copy} is associative.
IMPLEMENTING SCAN – CONTRACTION

- **scan** looks inherently sequential.
  - Naive implementation needs $O(n^2)$ work.
  - Slightly clever sequential implementation needs $O(n)$ work.
  - Divide and Conquer approaches do not break the sequentiality. (Why?)

- **Contraction**
  1. Construct a much smaller instance of the problem
  2. Solve the smaller instance recursively
  3. Construct solution to the original instance.
IMPLEMENTING REDUCE WITH CONTRACTION

- Given \langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle
- Apply + pairwise and (in parallel) to get \langle 3, 5, 7, 5 \rangle
  - This is the contracted instance!
- Apply + pairwise to get \langle 8, 12 \rangle
- Finally apply + pairwise to get \langle 20 \rangle
- The 3rd step of the contraction does nothing in this case.
IMPLEMENTING SCAN WITH CONTRACTION

Given $S = \langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$

- $scan + 0 S = (\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20)$

First do pairwise $+$ on $S$ to get $\langle 3, 5, 7, 5 \rangle$

Now (recursively) do scan on this to get $(\langle 0, 3, 8, 15 \rangle, 20)$

- What is the relation to the final scan?

We have every other element of the final scan!

How do we fill in the rest?
IMPLEMENTING SCAN WITH CONTRACTION

Input = \( \langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle \)

Partial Output = \( \langle 0, 3, 8, 15, 20 \rangle \)

Desired Output = \( \langle 0, 2, 3, 6, 8, 10, 15, 19, 20 \rangle \)
Implementing Scan with Contraction

% implements: the Scan problem on sequences that have a power of 2 length

fun scanPow2 f i s =
  case |s| of
    0 ⇒ (⟨⟩, i)
    1 ⇒ (⟨i⟩, s[0])
    n ⇒
      let
        val s' = ⟨f(s[2i], s[2i + 1]) : 0 ≤ i < n/2⟩
        val (r, t) = scanPow2 f i s'
      in
        (⟨pi : 0 ≤ i < n⟩, t), where pi = \[
          \begin{align*}
            r[i/2] & \quad \text{if } \text{even}(i) \\
            f(r[i/2], s[i - 1]) & \quad \text{otherwise.}
          \end{align*}
        \]
  end

General case is in the course notes.
SUMMARY

- Abstractions and Implementations
  - Meldable Priority Queues
- The Sequence ADT
- The $\text{scan}$ operation
- Introduction to contraction
- Implementing $\text{scan}$ with contraction.
SYNOPSIS

- The *reduce* operation
- Implementing divide and conquer with *reduce*
  - Implementing MCSS with *reduce*
- Analyzing cost of higher order functions
  - Cost analysis for *reduce*
The Reduce Operation

\[ \text{reduce } f / S : (\alpha \times \alpha \rightarrow \alpha) \rightarrow \alpha \]
\[ \rightarrow \alpha \text{ seq } \rightarrow \alpha \]

- When \( f \) is associative, \text{reduce} returns sum with respect to \( f \).
- Same result as \text{iter } f / S
  - \text{iter} is sequential and allows more general \( f \) (e.g., \( \beta \times \alpha \rightarrow \beta \))
- If \( f \) is not associative, \text{reduce} and \text{iter} results may differ.
THE REDUCE OPERATION

Specific combination based on a perfect binary tree.

= combine  = "dummy" elements

\[ x_0 \times \ldots \times x_1 \times x_2 \times x_3 \times x_4 \times x_5 \]

= combine  = "dummy" elements

\[ x_0 \times \ldots \times x_1 \times x_2 \times x_3 \times x_4 \times x_5 \]
Many divide and conquer have the following structure

```plaintext
1 fun myDandC(S) =
2   case showt(S) of
3     EMPTY ⇒ emptyVal
4     | ELT(v) ⇒ base(v)
5     | NODE(L, R) ⇒ let
6       val (L', R') = (myDandC(L) || myDandC(R))
7       in
8       somemessyCombine(L', R')
9     end
```

This corresponds to a binary tree combination scheme.
DIVIDE AND CONQUER WITH REDUCE


### Divide and Conquer with Reduce

```plaintext
fun myDandC(S) =
case showt(S) of
  EMPTY ⇒ emptyVal
| ELT(v) ⇒ base(v)
| NODE(L, R) ⇒ let
    val (L', R') = (myDandC(L) || myDandC(R))
in
  someMessyCombine(L', R')
end

reduce someMessyCombine emptyVal (map base S)
```

---

**Sequences - II**

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
MCSS USING REDUCE

\[
mcss(s) = \max_{1 \leq i \leq j \leq n} \left\{ \sum_{k=i}^{j} s_k \right\}
\]

Left Subproblem

Right Subproblem

**Total** = **Total**\(_L\) + **Total**\(_R\)

**mcss** = max ( **mcss**\(_L\), **mcss**\(_R\) )

**mps** = max ( **mps**\(_L\), **Total**\(_L\) )

**mss** = max ( **mss**\(_L\), **Total**\(_R\) )
MCSS USING REDUCE

\[ \text{mcss}(s) = \max_{1 \leq i \leq j \leq n} \left\{ \sum_{k=i}^{j} s_k \right\} \]

\[
\text{fun } \text{combine}((M_L, P_L, S_L, T_L), (M_R, P_R, S_R, T_R)) = \\
\quad (\max(S_L + P_R, M_L, M_R), \max(P_L, T_L + P_R), \max(S_R, S_L + T_R), T_L + T_R) \\
\text{fun } \text{base}(v) = (v, v, v, v) \\
\text{val } \text{emptyVal} = (-\infty, -\infty, -\infty, 0) \\
\]

\[
\text{fun } \text{mcss}(S) = \\
\quad \text{reduce combine emptyVal (map base S)}
\]
SOME OBSERVATIONS

- Which code to use is a matter of taste
  - reduce version is shorter
  - Divide and Conquer version exposes the inductive structure.
- reduce version not applicable when split is complicated
  - e.g., in Quicksort
How do we implement the divide and conquer scan with this template?

- \( \text{scan } f \mid S \equiv \) 
  \( \text{reduce combine emptyVal (map base } S) \)

- \( \text{emptyVal} = ? (\langle \rangle, I) \)
- \( \text{fun base}(v) = ? (\langle I \rangle, f(I, v)) \)
- \( \text{fun combine} = ? \)
  \( \text{fun combine}(\langle S_1, T_1 \rangle, \langle S_2, T_2 \rangle) = \) 
  \( \text{append}(S_1, (\text{map } (\text{fn } x \Rightarrow f(x, T_1)) S_2), f(T_1, T_2)) \)

- Is this right?

- \( \text{fun combine}(\langle S_1, T_1 \rangle, \langle S_2, T_2 \rangle) = \) 
  \( (\text{append}(S_1, (\text{map } (\text{fn } x \Rightarrow f(T_1, x)) S_2), f(T_1, T_2)) \)
Cost of Higher Order Functions

- We assume that $f$ runs in $O(1)$ work and span.
  - $\Rightarrow$ reduce has $O(n)$ work and $O(\log n)$ span
- Easy for map and tabulate

$$W(\text{map } f S) = 1 + \sum_{s \in S} W(f(s))$$

$$S(\text{map } f S) = 1 + \max_{s \in S} S(f(s))$$

- How about reduce?
Mergesort via Reduce

Remember the reduce template for divide and conquer?

\[
\text{reduce} \ \text{combine} \ \text{emptyVal} \ (\text{map} \ \text{base} \ S)
\]

\[
\text{val} \ \text{combine} = \text{merge} < \\
\text{val} \ \text{base} = \text{singleton} \\
\text{val} \ \text{emptyVal} = \text{empty}() \\
\text{fun} \ \text{reduceSort}(S) = \\
\quad \text{reduce} \ \text{combine} \ \text{emptyVal} \ (\text{map} \ \text{base} \ S)
\]
Cost of ReduceSort

- $\text{merge}_<$ is an associative function with costs:
  \[
  W(\text{merge}_<(S_1, S_2)) = O(n_1 + n_2) \\
  S(\text{merge}_<(S_1, S_2)) = O(\log(n_1 + n_2))
  \]

- $f$ has no longer $O(1)$ work and span.
- What is the cost of $\text{reduceSort}$.
For costs, reduction sequence matters!
Sequential order
On input $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$, we get

$$merge_<(\ldots merge_<(merge_<(l, \langle x_0 \rangle), \langle x_1 \rangle), \langle x_2 \rangle), \ldots)$$

- Left arg. has length 0 through $n-1$
- Right arg. always has length 1.
- Work:

$$W(\text{reduceSort } S) \leq \sum_{i=0}^{n-1} c \cdot (1 + i) \in O(n^2)$$

Why?
Mergesort with Reduce

- Equivalent to `iter` version
  
  \[\text{fun } \text{reduceSort}'(S) = \text{iter merge}_< \text{emptyVal} \text{ (map base } S)\]

- Works really as an Insertion Sort.
  - Inserts each element into a sorted prefix!
- No parallelism except in `merge`

\[S(\text{reduceSort } S) \leq \sum_{i=0}^{n-1} c \cdot \log(1 + i) \in O(n \log n)\]
MERGESORT WITH REDUCE

- The reduction tree is very unbalanced!
- Suppose $n = 2^k$ and merge tree is as follows

\[ x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \]

\[ = \text{merge} \]
Mergesort with Reduce

- \( n \) nodes at constant cost at each leaf (\( i = \log_2 n \))
- \( n/2 \) nodes one level up, each costing \( c(1 + 1) \) (\( i = \log_2 \frac{n}{2} \)) (Why?)
- In general, for level \( i \), we have \( 2^i \) nodes merging two sequences each length \( \frac{n}{2^{i+1}} \)
For level $i$, we have $2^i$ nodes merging two sequences each length $\frac{n}{2^{i+1}}$

$$W(\text{reduceSort } x) \leq \sum_{i=0}^{\log n} 2^i \cdot c \left( \frac{n}{2^{i+1}} + \frac{n}{2^{i+1}} \right)$$

$$= \sum_{i=0}^{\log n} 2^i \cdot c \left( \frac{n}{2^i} \right) \in O(n \log n)$$
**Mergesort with Reduce**

- \( W(\text{reduceSort}\ S) \in O(n \log n) \Rightarrow \text{mergeSort} \).
- \text{mergeSort} and \text{insertionSort} are special cases of \text{reduceSort} using different reduction orders.

\[ x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \]

\( \oplus = \text{merge} \)
**Reduce Order**

- Result of `reduce` depends on the order when $f$ is not associative.
- When $f$ is associative, different orders result in different costs.
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 7
Collect, Sets and Tables
SYNOPSIS

- The collect operation
- The map-collect-reduce paradigm
- Sets
- Tables
THE COLLECT OPERATION

- Group items that share a **common key**.

```plaintext
val Data = ⟨("jack sprat", "15-210"),
("jack sprat", "15-213"),
("mary contrary", "15-210"),
("mary contrary", "15-251"),
("mary contrary", "15-213"),
("peter piper", "15-150"),
("peter piper", "15-251"),
...⟩

↓

val rosters = ⟨("15-150", {"peter piper", ...}),
("15-210", {"jack sprat", "mary contrary", ...}),
("15-213", {"jack sprat", ...}),
("15-251", {"mary contrary", "peter piper"}),
...⟩
```
The Collect Operation

- Very common operation in Relational Databases
- Usually called the Group by operation.

```scala
val rosters = (("15-150", ("peter piper", . . .)))
              (("15-210", ("jack sprat", "mary contrary", . . .)))
              (("15-213", ("jack sprat", . . .)))
              (("15-251", ("mary contrary", "peter piper")))
              . . .
```

- **Students** are grouped by **Course Numbers**.
The Collect Operation

\[ \text{collect} : (\alpha \times \alpha \rightarrow \text{order}) \rightarrow (\alpha \times \beta) \text{ seq} \]
\[ \rightarrow (\alpha \times \beta \text{ seq}) \text{ seq} \]

1. \( \alpha \times \alpha \rightarrow \text{order} \) is a function for comparing keys of type \( \alpha \)
2. \((\alpha \times \beta) \text{ seq} \) is a sequence of key-value pairs
3. \((\alpha \times \beta \text{ seq}) \text{ seq} \) is the resulting sequence:
   - each unique \( \alpha \) value is paired with a sequence of all \( \beta \) values it appears with
The Collect Operation

val collectStrings = collect String.compare
val rosters = collectStrings(⟨(n, c):(c, n) ∈ Data⟩)

val rosters = ⟨("15-150", ⟨"peter piper", ...⟩)
("15-210", ⟨"jack sprat", "mary contrary", ...
("15-213", ⟨"jack sprat", ...⟩)
("15-251", ⟨"mary contrary", "peter piper"⟩)
...
⟩

⟨(n, c) : (c, n) ∈ Data⟩ arranges the data appropriately.
How would you implement \texttt{collect}?

- Sort the items on their keys
- Partition the resulting sequence
- Pull out pairs between each key change
The dominant cost of \texttt{collect} is in sorting.

Work is $O(W_c n \log n)$, Span is $O(S_c \log^2 n)$

- $W_c$ work bound for the comparison function
- $S_c$ span bound for the comparison function

A $O(n)$ work can be implemented with hashing.

- Need a separate hash function
- Output not in sorted order
Using Collect in Map-reduce

- The **map-reduce paradigm** is used to process very large collection of **documents**.
  - A document is a collection of **words/strings**.
  - Not the `mapReduce` of 15-150!

- **map-reduce paradigm** $\equiv$ map-collect-reduce(s).
USING COLLECT IN MAP-REDUCE

- $f_m$ maps each document to a sequence of key-value pairs.
  - $f_m : (\text{document} \rightarrow (\text{key} \times \alpha) \text{ seq})$

- All key-value pairs in a document are collected.

- $f_r$ is applied to the keys to get a single value for a key.
  - $f_r : \text{key} \times \alpha \text{ seq} \rightarrow \beta$
**An Example**

\[
docs = \langle \text{“this is a document”}, \\
\text{“this is is another document”}, \\
\text{“a last document”} \rangle
\]

\[
\downarrow
\]

\[
\langle \text{“this”}, 1 \rangle, \langle \text{“is”}, 1 \rangle, \langle \text{“a”}, 1 \rangle, \langle \text{“document”}, 1 \rangle, \\
\langle \text{“this”}, 1 \rangle, \langle \text{“is”}, 1 \rangle, \langle \text{“is”}, 1 \rangle, \langle \text{“a”}, 1 \rangle, \langle \text{“another”}, 1 \rangle, \\
\langle \text{“document”}, 1 \rangle, \langle \text{“a”}, 1 \rangle, \langle \text{“last”}, 1 \rangle, \langle \text{“document”}, 1 \rangle
\]

\[
\downarrow
\]

\[
\langle \text{“a”}, 2 \rangle, \langle \text{“another”}, 1 \rangle, \langle \text{“document”}, 3 \rangle, \langle \text{“is”}, 3 \rangle, \langle \text{“last”}, 1 \rangle, \\
\langle \text{“this”}, 2 \rangle
\]
fun mapCollectReduce \( f_m \ f_r \) docs =

\[
\text{let}
\begin{align*}
\text{val} & \quad \text{pairs} = \text{flatten} \langle f_m(s) : s \in \text{docs} \rangle \\
\text{val} & \quad \text{groups} = \text{collect String.compare \ pairs} \\
\text{in} \\
\langle f_r(g) : g \in \text{groups} \rangle
\end{align*}
\text{end}
\]

\[
\text{flatten} \langle \langle a, b, c \rangle, \langle d, e \rangle \rangle \Rightarrow \langle a, b, c, d, e \rangle
\]
MapReduce in SML

```sml
fun mapCollectReduce f_m f_r docs = 
  let
    val pairs = flatten ⟨f_m(s) : s ∈ docs⟩
    val groups = collect String.compare pairs
  in
    ⟨f_r(g) : g ∈ groups⟩
  end

fun f_m(doc) = ⟨(w, 1) : tokens doc⟩
fun f_r(w, s) = (w, reduce + 0 s)
```
**MapReduce Example in SML**

fun \( f_m(doc) = \langle (w, 1) : tokens \ doc \rangle \)

fun \( f_r(w, s) = (w, reduce + 0 \ s) \)

val countWords = mapCollectReduce \( f_m \ f_r \)

countWords \( \langle \text{“this is a document”}, \) \n\text{“this is another document”}, \) \n\text{“a last document”} \rangle \)

\( \Rightarrow \langle \langle \text{“a”}, 2 \rangle, \langle \text{“another”}, 1 \rangle, \langle \text{“document”}, 3 \rangle, \langle \text{“is”}, 3 \rangle, \) \n\langle \text{“last”}, 1 \rangle, \langle \text{“this”}, 2 \rangle \rangle \)
**Sets**

- Sets play a very important role in math.
- Often needed in many algorithms.
- Many languages either support sets directly or have libraries for sets.
- In 15-210 we use a purely functional definition for sets:
  - When updates are done, a new set is returned.
**Sets as an ADT**

- $\mathcal{U}$ is a universe of elements.
- The SET ADT is a type $\mathcal{S}$ that represents the power set of $\mathcal{U}$.

\[
\begin{align*}
\text{empty} &: \mathcal{S} \
\text{size}(S) &: \mathcal{S} \to \mathbb{Z}_{\geq 0} \
\text{singleton}(e) &: \mathcal{U} \to \mathcal{S} \
\text{filter}(f, S) &: ((\mathcal{U} \to \{T, F\}) \times \mathcal{S}) \to \mathcal{S}
\end{align*}
\]
SETS AS AN ADT

\[
\text{find}(S, e) : \quad S \times U \quad = \quad \{|s \in S \mid s = e|\} = 1
\rightarrow \{T, F\}
\]

\[
\text{insert}(S, e) : \quad S \times U \rightarrow S \quad = \quad S \cup \{e\}
\]

\[
\text{delete}(S, e) : \quad S \times U \rightarrow S \quad = \quad S \setminus \{e\}
\]

\[
\text{intersection}(S_1, S_2) : \quad S \times S \rightarrow S \quad = \quad S_1 \cap S_2
\]

\[
\text{union}(S_1, S_2) : \quad S \times S \rightarrow S \quad = \quad S_1 \cup S_2
\]

\[
\text{difference}(S_1, S_2) : \quad S \times S \rightarrow S \quad = \quad S_1 \setminus S_2
\]

- What is the relationship between these two groups?
**Sets as an ADT**

- **We do not really need** `find`, `insert`, `delete`!

\[
\begin{align*}
\text{find}(S, e) & = \text{size}(\text{intersection}(S, \text{singleton}(e))) = 1 \\
\text{insert}(S, e) & = \text{union}(S, \text{singleton}(e)) \\
\text{delete}(S, e) & = \text{difference}(S, \text{singleton}(e))
\end{align*}
\]

- **intersection, union, and difference**
  - can operate on multiple elements, and
  - are suitable for parallelism
Cost Model for Sets

Underlying data structure can be
- hash-tables
- balanced trees

We will assume a balanced-tree implementation.

We will assume comparison of two set elements take
- $W_c$ work and $S_c$ span.
## Cost Model for Sets

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>size($S$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>singleton($e$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filter($f, S$)</td>
<td>$O\left(\sum_{e \in S} W(f(e))\right)$</td>
<td>$O\left(\log</td>
</tr>
<tr>
<td>find($S, e$)</td>
<td>$O(W_c \cdot \log</td>
<td>S</td>
</tr>
<tr>
<td>insert($S, e$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete($S, e$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cost Models for Sets

intersection($S_1, S_2$) \hspace{1cm} \text{Work} = O(W_c \cdot m \cdot \log(1 + \frac{n}{m}))

union($S_1, S_2$) \Rightarrow \hspace{1cm} \text{Span} = O(S_c \cdot \log(n + m))

difference($S_1, S_2$)

\[ n = \max(|S_1|, |S_2|) \hspace{1cm} \text{and} \hspace{1cm} m = \min(|S_1|, |S_2|) \]

- Sets are equal size ($n = m$)
  - Work = $O(W_c \cdot m \cdot \log(1 + 1)) = O(W_c \cdot n)$
  - Span = $O(S_c \cdot \log n)$

- One of the sets is a singleton ($m = 1$)
  - Work = $O(W_c \cdot \log(1 + n)) = O(W_c \cdot \log n)$
  - Span = $O(S_c \cdot \log(n + 1)) = O(S_c \cdot \log n)$
Tables

- Table is an ADT for sets of **key-value** pairs.

  \[ \{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\} \]
  \[ \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\} \]

- Each **key** appears only once
- Many languages provide either built-in support or libraries.
**Tables**

- $K$ is the universe of keys.
- $V$ is the universe of values.
- $T$ is a type that represents the power set of $K \times V$
  - restricted so that each key appears at most once.
  - $S$ is the power set of $K$.
  - $\mathbb{Z}_{\geq 0}$ denotes the positive integers.
Table Functions

(empty) : $T = \emptyset$

(size) : $T \rightarrow \mathbb{Z}_{\geq 0} = |T|$

(singleton($k, v$)) : $K \times V \rightarrow T = \{(k, v)\}$

(filter($f, T$)) : $((V \rightarrow \{T, F\}) \times T) \rightarrow T = \{(k, v) \in T | f(v)\}$

(map($f, T$)) : $((K \times V \rightarrow V) \times T) \rightarrow T = \{(k, f(k, v)) | ((k, v) \in T)\}$

(insert($f, T, (k, v)$)) : $(V \times V \rightarrow V) \times T \times (K \times V) \rightarrow T$

$= (T \setminus \{(k, v)\}) \cup \{(k, f(v, v'))\}$

if $(k, v') \in T$

$T \cup \{(k, v)\}$

otherwise

(delete($T, k$)) : $T \times K \rightarrow T = \{(k', v) \in T | k \neq k'\}$
TABLE FUNCTIONS

\[ \text{find}(T, k) : T \times K \rightarrow (V \cup \perp) = \begin{cases} v & (k, v) \in T \\ \perp & \text{otherwise} \end{cases} \]

\[ \text{merge}(f, T_1, T_2) : (V \times V \rightarrow V) \times T \times T \rightarrow T = \bigcup_{k \in K} \begin{cases} \{ (k, f(v_1, v_2)) \} & (k, v_1) \in T_1 \land (k, v_2) \in T_2 \\ \{ (k, v_1) \} & (k, v_1) \in T_1 \land (k, v_2) \notin T_2 \\ \{ (k, v_2) \} & (k, v_2) \in T_2 \land (k, v_1) \notin T_1 \end{cases} \]

\[ \text{extract}(T, S) : T \times S \rightarrow T = \{ (k, v) \in T \mid k \in S \} \]

\[ \text{erase}(T, S) : T \times S \rightarrow T = \{ (k, v) \in T \mid k \notin S \} \]
Table Examples

Suppose we have the two tables:

- **Summer**: \( \{ \text{tree} \mapsto \text{green}, \; \text{sky} \mapsto \text{blue}, \; \text{cmuq} \mapsto \text{tan} \} \)
- **Fall**: \( \{ \text{grass} \mapsto \text{gray}, \; \text{tree} \mapsto \text{brown} \} \)

```
merge (fn (a, b) \Rightarrow b) Summer Fall

\{ \text{grass} \mapsto \text{gray}, \; \text{tree} \mapsto \text{brown}, \; \text{sky} \mapsto \text{blue}, \; \text{cmuq} \mapsto \text{tan} \}
```
Suppose we have the two tables:

- \( \text{Summer} = \{\text{tree} \mapsto \text{green}, \ \text{sky} \mapsto \text{blue}, \ \text{cmuq} \mapsto \text{tan}\} \)
- \( \text{Fall} = \{\text{grass} \mapsto \text{gray}, \ \text{tree} \mapsto \text{brown}\} \)

\[ \text{extract}(\text{Summer}, \{\text{sky}, \ \text{grass}\}) \]

- \( \{\text{sky} \mapsto \text{blue}\} \)
Table Examples

Suppose we have the two tables:

- **Summer** = \{ tree \mapsto green, sky \mapsto blue, cmuq \mapsto tan \}
- **Fall** = \{ grass \mapsto gray, tree \mapsto brown \}

\[ \text{erase}(\text{Summer}, \{sky, grass\}) \]

\[ \{ tree \mapsto green, cmuq \mapsto tan \} \]
**Table Examples**

- Other useful functions from the library

- \( \text{collect}: (\text{key} \times \alpha) \ \text{seq} \rightarrow (\alpha \ \text{seq}) \ \text{table} \)

- \( \text{fromSeq}: (\text{key} \times \alpha) \ \text{seq} \rightarrow \alpha \ \text{table} \)
  - \( \text{fromSeq}(A) = \{ k \mapsto s_0 : (k \mapsto S) \in \text{collect}(A) \} \)
Table Functions

- Major differences from sets:
  - find returns the value if key is in the table else returns ⊥ (NONE).
  - insert/merge need a function to combine if the key is already in the/both table(s).

- Just as with sets, there is symmetry between
  - extract and find
  - merge and insert
  - erase and delete
## Cost Models for Tables

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>size($T$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>singleton($k, v$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filter($f, T$)</td>
<td>$O\left(\sum_{(k,v)\in T} W(f(v))\right)$</td>
<td>$O\left(\log</td>
</tr>
<tr>
<td>map($f, T$)</td>
<td>$O\left(\sum_{(k,v)\in T} W(f(k,v))\right)$</td>
<td>$O\left(\max_{(k,v)\in T} S(f(k,v))\right)$</td>
</tr>
</tbody>
</table>
## Cost Models for Tables

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>find $S, k$</td>
<td>$O(W_c \log</td>
<td>T</td>
</tr>
<tr>
<td>insert $T, (k, v)$</td>
<td>$O(W_c \log</td>
<td>T</td>
</tr>
<tr>
<td>delete $T, k$</td>
<td>$O(W_c \log</td>
<td>T</td>
</tr>
<tr>
<td>extract $T_1, T_2$</td>
<td>$O(W_c m \log (1 + \frac{n}{m}))$</td>
<td>$O(S_c \log (n + m))$</td>
</tr>
<tr>
<td>merge $T_1, T_2$</td>
<td>$O(W_c m \log (1 + \frac{n}{m}))$</td>
<td>$O(S_c \log (n + m))$</td>
</tr>
<tr>
<td>erase $T_1, T_2$</td>
<td>$O(W_c m \log (1 + \frac{n}{m}))$</td>
<td>$O(S_c \log (n + m))$</td>
</tr>
</tbody>
</table>

$n = \max(|T_1|, |T_2|) \quad m = \min(|T_1|, |T_2|)$
SUMMARY

- Collect
- Map-Collect-Reduce
- Sets
- Tables
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 8
Sets and Tables–II
SYNOPSIS

- How search engines work
- Single-threaded sequences
How do search engines work?

- What are the inputs?
  - (Billions and billions of) documents consisting of “words”.

- How do we interact with the search engine
  - (Boolean) Keyword queries
  - List of matching documents (URLS)
How does the search really work?

- User inputs a query (say a couple of words)
- SE starts searching for the words in each document one-by-one
- Returns documents when they match.
- Not really!
  - Not scalable (even for one user)
- Preprocessing
Preprocessing
Plan

- What kinds of queries we want to have.
- What is the interface we want to have.
- How do we implement all these
**Queries**

- **Bingle (:-)** supports logical queries on words involving
  - And: “15210” And “course” And “slides”
  - Or: “15210” Or “15150”
  - AndNot: “15210” AndNot “Pittsburgh”

- “CMU” And “fun” And (“courses Or “clubs”)

- Result would be a list of webpages/documents that match.
THE INTERFACE

signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

val makeIndex : (docId * string) seq -> index

val find : index -> word -> docList
val And : docList * docList -> docList
val AndNot : docList * docList -> docList
val Or : docList * docList -> docList
val size : docList -> int
val toSeq : docList -> docId seq
Documents

- Indexing a tweet database.

\[ T = \langle \text{"jack"}, \text{"chess club was fun"}, \]
\[ \text{"mary"}, \text{"I had a fun time in 210 class today"}, \]
\[ \text{"nick"}, \text{"food at the cafeteria sucks"}, \]
\[ \text{"sue"}, \text{"In 217 class today I had fun reading my email"}, \]
\[ \text{"peter"}, \text{"I had fun at nick’s party"}, \]
\[ \text{"john"}, \text{"tiddlywinks club was no fun, but more fun than 218"} \rangle \]

- "jack" is a document id
- "chess club was fun" is a document
Using the Interface

\( T = \langle (\text{"jack"}, \text{"chess club was fun"}), \\
(\text{"mary"}, \text{"I had a fun time in 210 class today"}), \\
(\text{"nick"}, \text{"food at the cafeteria sucks"}), \\
(\text{"sue"}, \text{"In 217 class today I had fun reading my email"}), \\
(\text{"peter"}, \text{"I had fun at nick’s party"}), \\
(\text{"john"}, \text{"tiddlywinks club was no fun, but more fun than 218"}) \rangle \\
\) 

\textbf{val} \quad f = (\text{find } (\text{makeIndex}(T))): \text{word} \rightarrow \text{doclist} \\

\text{toSeq}(\text{And}(f \ "fun", \text{Or}(f \ "class", f \ "club"))) \\
\Rightarrow \langle \text{"jack"}, \text{"mary"}, \text{"sue"}, \text{"john"} \rangle
Using the Interface

\[ T = \langle \text{"jack"}, \text{"chess club was fun"}), \\
\text{"mary"}, \text{"I had a fun time in 210 class today"}), \\
\text{"nick"}, \text{"food at the cafeteria sucks"}), \\
\text{"sue"}, \text{"In 217 class today I had fun reading my email"}), \\
\text{"peter"}, \text{"I had fun at nick's party"}), \\
\text{"john"}, \text{"tiddlywinks club was no fun, but more fun than 218"}), \rangle \\
\]

\[ \text{size(AndNot(f "fun", f "tiddlywinks"))} \]

\[ \Rightarrow 4 \]
THE MAKEINDEX FUNCTION

fun makeIndex(docs) =
let
  fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
  val Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
  val Words = Table.collect(Pairs)
in
  {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
end

What does tagWords do?

tagWords(“jack”, “chess club was fun”)
⇒ ⟨(“chess”, “jack”), (“club”, “jack”), (“was”, “jack”), (“fun”, “jack”)⟩
THE PAIRS FUNCTION

fun makeIndex(docs) =
  let
  fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
  val Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
  val Words = Table.collect(Pairs)
  in
  {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
  end

- What does Pairs do?

Pairs = ⟨("chess", "jack"), ("club", "jack"), ("was", "jack"),
         ("fun", "jack"), ("I", "mary"), ("had", "mary"),
         ("fun", "mary"), . . .⟩
THE COLLECT FUNCTION

fun makeIndex(docs) =
  let
    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str) ⟩
    val Pairs = flatten{tagWords(d) : d ∈ docs}
    val Words = Table.collect(Pairs)
  in
    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
  end

What does collect do?

Words = { (“a” ↦ {“mary”}),
          (“at” ↦ {“mary”, “peter”}),
          (“fun” ↦ {“jack”, “mary”, “sue”, “peter”, “john”}),
          ...
}
fun makeIndex(docs) =
let
  fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
  val Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
  val Words = Table.collect(Pairs)
in
  {w ↦ Set.fromSeq(d):(w ↦ d) ∈ Words}
fun makeIndex(docs) =
  let
  fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
  val Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
  val Words = Table.collect(Pairs)
  in
  {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
  end

Assuming tokens have a upper bound on length
  - \( W_{\text{makeIndex}}(n) \in O(n \log n) \), \( S_{\text{makeIndex}} \in O(\log^2 n) \)
  - What does \( n \) represent?
REST OF THE INTERFACE

fun find $T$ $v$ = Table.find $T$ $v$
fun And($s_1$, $s_2$) = $s_1$ ∩ $s_2$
fun Or($s_1$, $s_2$) = $s_1$ ∪ $s_2$
fun AndNot($s_1$, $s_2$) = $s_1$ \ $s_2$
fun size($s$) = $|s|$
fun toSeq($s$) = Set.toSeq($s$)
**Single-threaded Array Sequences**

- Updating an array sequence in an imperative language takes $O(1)$ work.
- In a functional setting, everything is persistent.
- An update to a sequence of $n$ elements needs:
  - $O(n)$ work for `arraySequence` implementation to copy and update.
  - $O(\log n)$ work for `treeSequence` implementation (because of substructure sharing)
- Interfaces do not provide functions for updating a single position.
  - to discourage sequential (and expensive) computation.
A map can be implemented as follows

\[
\text{fun } \text{map } f \ S = \\
\quad \text{iter } (\text{fn } ((i, S'), \nu) \Rightarrow (i + 1, \text{update } (i, f(\nu)) \ S')) \\
\quad (0, S)
\]

- Iterates \( n \) times \((i = 0, \ldots n - 1)\)
- and updates the value \( S_i \) with \( f(S_i) \).
- arraySequence: Each update will do \( O(n) \) work for a total \( O(n^2) \) work
- treeSequence: Each update will do \( O(\log n) \) work for a total \( O(n \log n) \) work.
SINGLE-THREADED SEQUENCES

- A new ADT: **Single Threaded Sequence**: `stseq`
- Useful when a bunch of items need to be updated.
- Straightforward interface
- Cost specification imply non-functional stuff under the hood!
## STSEQ Interface and Costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fromSeq(S)</code> : $\alpha$ seq $\rightarrow$ $\alpha$ stseq</td>
<td>$O(</td>
<td>S</td>
</tr>
<tr>
<td>Converts from a regular sequence to a stseq.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>toSeq(ST)</code> : $\alpha$ stseq $\rightarrow$ $\alpha$ seq</td>
<td>$O(</td>
<td>S</td>
</tr>
<tr>
<td>Converts from a stseq to a regular sequence.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nth ST i</code> : $\alpha$ stseq $\rightarrow$ int $\rightarrow$ $\alpha$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Returns the $i^{th}$ element of ST. Same as for seq.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>update (i, v) S</code> : (int $\times$ $\alpha$) $\rightarrow$ $\alpha$ stseq $\rightarrow$ $\alpha$ stseq</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Replaces the $i^{th}$ element of $S$ with $v$.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>inject I S</code> : (int $\times$ $\alpha$) seq $\rightarrow$ $\alpha$ stseq $\rightarrow$ $\alpha$ stseq</td>
<td>$O(</td>
<td>I</td>
</tr>
<tr>
<td>For each $(i, v) \in I$ replaces the $i^{th}$ element of $S$ with $v$.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cost bounds for `nth` and `update` only valid for the “current” version of the sequence.
Map with stseq

```ml
fun map f S = let
  val S' = StSeq.fromSeq(S)
  val R = iter
      (fn ((i, S''), v) ⇒ (i + 1, StSeq.update (i, f(v)) S''))
      (0, S')
  in
  StSeq.toSeq(R)
  end
```

- Overall work and span is $O(n)$ (Why?)
- Multiple updates can be done in $O(n)$ time.
IMPLEMENTING STSEQ

- Keep two full copies of the sequence
  - Original and Current
  - We keep a change log: updates to the original to get Current.
- When Current is updated
  - We create a “new” Current with the update, and update change log.
  - Mark the previous version as old, remove its Current and but keep the old change log.
- Any item from the current version is accessible in constant work.
- Any item from the any previous version is accessible but needs more work.
IMPLEMENTING STSEQ

Change Log

| Original | ( ) | Current |

CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS

SETS AND TABLES–II

FALL 2013
IMPLEMENTING STSEQ

Change Log

<table>
<thead>
<tr>
<th>Original</th>
<th>( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td></td>
</tr>
</tbody>
</table>

update(3, 5)

<table>
<thead>
<tr>
<th>Original</th>
<th>( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Version1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original</th>
<th>((3,5) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Current</td>
</tr>
</tbody>
</table>

- There really is only one copy of the **Original**.
- All point to that copy.
IMPLEMENTING STSEQ

Original ( )   Old Version1

Original ((3,5) )

Original ( )
Original ((3,5) )

update(6, 7)

Original ( )   Old Version1

Original ((3,5) )   Old Version2

Original ((6, 7)(3,5) )   5  Current  7
IMPLEMENTING STSEQ

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )

Original ((3,5) )

Original ((6, 7)(3,5) )

Original ( )
**Summary**

- How search engines work
- Single-threaded sequences
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 9
Graphs
SYNOPSIS

- Graphs
- Graph terminology/definitions
- Graph representations/costs.
- Graph search
Graphs

- Most versatile ADT in the study of algorithms
- Captures relationships between pairs of items
- A graph consists of
  - a set of $V$ vertices/nodes
  - a set edges $E \subseteq V \times V$
- Edges represent relationships between nodes.
  - directed edges (asymmetric relationships)
  - undirected edges (symmetric relationships)
- Nodes or edges can have additional weights or values associated.
Social Networks - Questions

- Who is popular?
- What is the largest “clique”?  
- Do I know somebody who knows X?  
- What is the “diameter”?  

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
**Transportation Networks - Questions**

- What is the shortest route from NYC to Los Angeles?
  - without Toll Roads?
  - without any state roads?
- What is the expected driving time from Boston to Atlanta?
  - considering traffic congestion?
Flow Networks
Flow Networks - Questions

- Is it possible to send 1M cubic meters of gas to Paris daily?
- What is the maximum gas that can be pumped from Azerbaijan to Italy?
OTHER EXAMPLES OF GRAPHS

- Course prerequisite relation graphs (directed-acyclic)
- Web-page linkage graph
- Protein-protein interaction graph
- Neural networks
- Semantic networks
**Directed Graphs**

- A directed graph (digraph) is $G = (V, E)$
  - $V$ is a set of vertices (or nodes), and
  - $E \subseteq V \times V$ is a set of directed edges (or arcs).
- Each arc is an ordered pair $e = (u, v)$
  - Arcs represent asymmetric relationships
  - A graph can have self loops $(u, u)$
An undirected graph is \( G = (V, E) \)
- \( V \) is a set of vertices (or nodes), and
- \( E \subseteq V \times V \) is a set of edges

Each edge is an unordered pair \( e = \{u, v\} \)
- Edges represent symmetric relationships
- A undirected graphs do not have self-loops.
In an undirected graph, $G = (V, E)$, a vertex $v$ is a neighbor of $u$ if $\{u, v\} \in E$.

In an undirected graph, $N_G(v) = \{u \mid \{u, v\} \in E\}$ is the neighborhood of $v$.

If $U$ is a set of nodes,
- $N_G(U) = \bigcup_{v \in U} N_G(v)$ is the neighborhood of $U$.
In a directed graph, $G = (V, E)$,
- $u$ is an in-neighbor of $v$ if $(u, v) \in E$
- $u$ is an out-neighbor of $v$ if $(v, u) \in E$

In a directed graph
- $N^{-}_G(u)$ is the set of in-neighbors of $u$.
- $N^{+}_G(u)$ is the set of out-neighbors of $u$.
- When we use $N_G(v)$, we mean out-neighbors.

If $U$ is a set of nodes,
- $N^{+}_G(U) = \bigcup_{u \in U} N^{+}_G(u)$ is the out-neighborhood of $U$. 
NODE DEGREES

- **Undirected graphs**: degree $d_G(v)$ of a vertex $v$ is $|N_G(v)|$
- **Directed graphs**:
  - in-degree of a vertex $v$ is $d^-_G(v) = |N^-_G(v)|$
  - out-degree of a vertex $v$ is $d^+_G(v) = |N^+_G(v)|$
- We will remove subscript $G$ if it is clear from context.
**Paths**

- A *path* is a sequence of adjacent vertices.
- For a graph $G = (V, E)$

$$Paths(G) = \{ P \in V^+ | 1 \leq i < |P|, (P_i, P_{i+1}) \in E \}$$

- $V^+$ denotes a sequence of length 1 or more.
- Repeats are allowed.
- The *length* of a path is the number of edges.
- A path may have an infinite length.
- A *simple path* has no repeated vertices.
  - Often “simple” will be dropped.
**Reachability**

- A vertex $v$ is **reachable** from a vertex $u$ in $G$ if there is a path starting at $u$ and ending at $v$ in $G$.
- $R_G(u)$ is the set of vertices reachable from $u$.
- An undirected graph is **connected** if all vertices are reachable from all other vertices.
- A directed graph is **strongly connected** if all vertices are reachable from all other vertices.
Cycles

- A cycle is a path that starts and ends at the same vertex.
- In a directed graph a cycle can have length 1 (i.e. a self loop).
- In an undirected graph we require that a cycle must have length at least three.
  - Going from $u$ to $v$ and back to $u$ does not count.
- A simple cycle is a cycle that has no repeated vertices other than the start vertex being the same as the end.
Trees, Forests and DAGs

- An undirected graph with no cycles is a forest.
- If it is connected then it is a tree.
- A directed graph is a forest or tree if it becomes a forest or tree when all arcs are made undirected.
- In a rooted tree one node is the root.
- For a directed graph, all edges are either towards the root or away from the root.
- A directed graph with no cycles is a directed acyclic graph (DAG)
The distance $\delta_G(u, v)$ from a vertex $u$ to a vertex $v$ in a graph $G$ is the shortest path (minimum number of edges) from $u$ to $v$.

The diameter of a graph is the maximum shortest path length over all pairs of vertices: $\text{diam}(G) = \max \{\delta_G(u, v) : u, v \in V\}$. 
Multi-graphs allow multiple edges between the same pair of vertices.
Sparse and Dense Graphs

- Let $n = |V|$ and $m = |E|$.
- A directed graph can have at most $n^2$ edges.
- An undirected graph can have at most $\frac{n(n - 1)}{2}$ edges.
- A graph is sparse if $m \ll n^2$. Otherwise it is called dense.
- In most applications, that graphs are sparse.
  - Nobody on Twitter has $10^9$ followers
  - Though some have very large number— but still small when compared to $n$. 

Graphs
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
**Operations on Graphs**

1. Map over the vertices \( v \in V \).
2. Map over the edges \( (u, v) \in E \).
3. Map over the neighbors of a vertex \( v \in V \), or in a directed graph the in-neighbors or out-neighbors.
4. Return the degree of a vertex \( v \in V \).
5. Determine if an edge \( (u, v) \) is in \( E \).
6. Insert or delete vertices.
7. Insert or delete edges.
**Adjacency Matrix Representation**

- Assume vertices are numbered 1, 2, ..., n (or 0, 1, ..., n−1).
- Graph is represented by an $n \times n$ matrix of binary values in which location $(i, j)$ is 1 if $(i, j) \in E$ and 0 otherwise.
  - For undirected graphs, matrix is symmetric and has 0’s along the diagonal.

![Graph Example](image)

\[
\begin{bmatrix}
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0
\end{bmatrix}
\]
Adjacency List Representation

- Graph is represented by an array $A$ of length $n$ where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex $i$.
  - In an undirected graph edge $\{u, v\}$ will appear in the adjacency list for both $u$ and $v$ (not always necessary!)

```
1 3
2 4
```

```
1 2 3 4
3 4
1
2
3
4
1
2
3
```

Graphs
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
Other Representations

- Adjacency Array

- Edge List

\[ ((1, 3), (1, 4), (2, 4), (3, 1), (3, 4), (4, 1), (4, 2), (4, 3)) \]
MORE ABSTRACT REPRESENTATIONS

Edge Sets

- Directed graphs: Set items are pairs \((u, v)\) representing arcs.
- Undirected graphs: Set items are sets \(\{u, v\}\) representing edges.

Edge Tables

- Directed graphs: Table items are pairs \((u, v) \mapsto w_{u,v}\) representing arcs and associated values.
- Undirected graphs: Set items are pairs \(\{u, v\} \mapsto w_{u,v}\) representing edges and associated values.
**Edge Sets and Tables**

- Similar to edge lists but abstracts from underlying representation.
- Search for an edge needs $O(\log m)$ work.
- Searching for neighbors is not efficient: $O(m)$ work but $O(\log m)$ span. (Why?)
Adjacency Tables

- Table items are \((key, value)\) pairs.
- Keys are vertex/node labels.
- Values are either sets or tables
  - Sets: All neighbors node labels or out-neighbor node labels.
  - Tables: All pairs of neighbors node labels and associated edge values.
- Accessing neighbors needs \(O(\log n)\) work and span.
- (Constant work) Map over neighbors needs \(O(d_G(u))\) work and \(O(\log d_G(u))\) span.
- Looking up an edge needs \(O(\log n)\) work and span.
### Cost Summary

<table>
<thead>
<tr>
<th></th>
<th>edge set</th>
<th></th>
<th>adj table</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>work</td>
<td>span</td>
<td>work</td>
<td>span</td>
</tr>
<tr>
<td>isEdge(G, (u, v))</td>
<td>O(log m)</td>
<td>O(log m)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>map over all edges</td>
<td>O(m)</td>
<td>O(log m)</td>
<td>O(m)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>map over neighbors of v</td>
<td>O(m)</td>
<td>O(log m)</td>
<td>O(log n + d_G(v))</td>
<td>O(log n)</td>
</tr>
<tr>
<td>d_G(v)</td>
<td>O(m)</td>
<td>O(log m)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

**Graphs**

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
**Graph Search**

- Fundamental operation of graphs
  - Start at some (set of) node(s)
  - Systematically visit all reachable nodes (only once)
- Used for determining properties of graphs/nodes
  - Connected?
  - Bipartite?
  - Node \( v \) reachable from node \( u \)?
  - Shortest path from \( u \) to \( v \)?
GRAPH SEARCH

For all graph search methods vertices can be partitioned into three sets at any time during the search:

1. vertices already \textit{visited} \((X)\),
2. the unvisited neighbors of the visited vertices, called the \textit{frontier} \((F)\),
3. and the rest.
Graph Search Methods

- Breadth-first Search (BFS)
  - Parallelizable but for shallow graphs!
- Depth-first Search (DFS)
  - Inherently sequential!
- Priority-first Search (PFS)
- All reachable nodes from a source are visited, but in different orders.
GRAPH SEARCH TREES

- Each search starting from a source node creates a search tree.
- We refer to the source node as the root.

- Which search schemes do these correspond to?
Graphs
Graph terminology/definitions
Graph representations/costs.
Graph search
15-210
Paralle and Sequential Algorithms and Data Structures

Lecture 10
Breadth-first Search
SYNOPSIS

- Breadth-first search
- BFS Extensions
- BFS Costs
- BFS with Single-threaded Sequences
**Graph Search**

- Fundamental operation of graphs
  - Start at some (set of) vertex(s)
  - Systematically visit all reachable vertices (only once)
- Used for determining properties of graphs/vertices
  - Connected?
  - Bipartite?
  - Vertex $v$ reachable from vertex $u$?
  - Shortest path from $u$ to $v$?
Graph Search Methods

- Breadth-first Search (BFS)
  - Parallelizable but for shallow graphs!
- Depth-first Search (DFS)
  - Inherently sequential!
- Priority-first Search (PFS)
- All reachable vertices from a source are visited, but in different orders.
Breadth-first Search

Applicable to a variety of problems
- Connectedness
- Reachability
- Shortest path
- Diameter
- Bipartiteness

Applicable to both directed and undirected graphs
- For digraphs, we only consider outgoing arcs.
For all graph search methods vertices can be partitioned into three sets at any time during the search:

1. vertices already *visited* \((X \subseteq V)\),
2. the unvisited neighbors of the visited vertices, called the *frontier* \((F)\),
3. the rest; unseen vertices.

The search essential goes as follows:

while vertices remain

- visit some unvisited neighbors of the visited set

Web navigation analogy.
**Breadth-first Search**

- Starting from a source vertex $s$
  - Visit all vertices that are (out-)neighbors of $s$ (at distance 1)
  - Visit all vertices at distance 2 from $s$
  - Visit all vertices at distance 3 from $s$, etc.

- A vertex at distance $i + 1$ must have a (in-)neighbor at distance $i$. 
Breadth-first Search

- BFS needs to keep track of vertices already visited.
- $X_i$: all vertices visited at start of level $i$
  - Vertices in $X_i$ have distance less than $i$.
- $F_i$: all unvisited neighbors of vertices in $X_i$
  - Vertices in $F_i$ have distance exactly $i$.
- “Visit” ⇒ Do something with the vertices (e.g., print it).
- $X_{i+1} = X_i \cup F_i$
- $F_{i+1} = N_G(F_i) \setminus X_{i+1}$ ($N_G(F_i) = \bigcup_{v \in F_i} N(v)$)
Breadth-first Search

1 fun BFS(G = (V, E), s) =
2   let
3     fun BFS′(X, F, i) =
4       if |F| = 0 then (X, i)
5       else let
6         val X′ = X ∪ F % Visit the Frontier
7         val N = NG(F) % Determine the neighbors
8         val F′ = N \ X′ % Remove vertices that have
9            % been visited
10        in BFS′(X′, F′, i + 1) % Next level
11      end
12     end
13   in BFS′({}, {s}, 0)
14 end
Some Details

- Adjacency table representation
  - Entries of the sort \((\text{Vertex}, \{\text{Neighbors}\})\).
- Remember \(N_G(F) = \bigcup_{v \in F} N(v)\)

```haskell
fun \(N_G(F) = \text{Table.reduce Set.Union} \{\}
\text{Table.extract}(G, F)\)
```

Breadth-first Search
Proving BFS Correct

- State and prove an invariant.
- All reachable vertices are returned.
- Algorithm terminates.
Proving BFS Correct

Lemma

In algorithm BFS when calling BFS′(X, F, i), we have

- \( X = \{ v \in V_G \mid \delta_G(s, v) < i \} \), and
- \( F = \{ v \in V_G \mid \delta_G(s, v) = i \} \)

By induction on levels \( i \)

- For base case \((i = 0)\) \( X_0 = \{ \}, F_0 = \{ s \} \)
  - Only \( s \) has distance 0 from \( s \)
  - No vertex has distance < 0 from \( s \).

So base case is true!
Proving BFS Correct

- Assume claims are true for \( i \), show for \( i + 1 \).
- \( X_{i+1} \) is the union of
  - \( X_i \): all vertices at distance \(< i\)
  - \( F_i \): all vertices at distance \( = i \)
- Hence \( X_{i+1} \) must have all vertices at distance \(< i + 1\)
- \( F_{i+1} = N_G(F_i) \setminus X_{i+1} \)
  - Vertices in \( F_i \) have distance exactly \( i \)
  - Vertices in \( N_G(F_i) \) have distance no more than \( i + 1 \)
  - Vertices in \( N_G(F_i) \) are reachable from a vertex at distance \( i \)
  - When we remove \( X_{i+1} \) from \( N_G(F_i) \) only unvisited vertices at distance exactly \( i + 1 \) remain.
**Additional Observations**

- If $v$ is reachable from $s$ and has distance $d$, there must be a vertex $u$ at distance $d - 1$.
  - BSF will not terminate without finding $v$.

- For any vertex $\delta(s, v) < |V|$, so algorithm will terminate in at most $|V|$ rounds/levels.
**EXTENSIONS TO BFS**

- Finding shortest distances
- What do we need to keep?

```ocaml
fun BFS(G, s) = let
  fun BFS′(X, F, i) =
    if |F| = 0 then X
    else let
      val X′ = X ∪ \{v ↦ i : v ∈ F\}
      val F′ = N_G(F) \ domain(X′)
    in BFS′(X′, F′, i + 1) end
  in BFS′({}, {s}, 0) end
```
EXTENSIONS TO BFS

Finding BFS trees.

There could be multiple BFS trees.
Finding BFS Trees

- What do we need to keep for each vertex?
- Record a parent
  - If \( v \) is in a frontier, then there should be one or more visited vertices \( u \) such that \((u, v) \in E\).
  - Any of those could be the parent of \( v \).
IDENTIFYING PARENTS

- Post-process the BFS distance table
- Identify one (in-)neighbor vertex in $N^{-}(v)$ whose distance is one less.
- Another way is to keep a table of vertices mapping to parents.
  - For each $v \in F$, generate a table \( \{u \mapsto v : u \in N(v)\} \)
  - Maps each neighbor of $v$ back to $v$.
- Merge these tables to $X$
  - Choose one if you have multiple parents.
Cost Analysis for BFS

Most graph algorithms do NOT use divide and conquer.
  ▶ So no natural way to develop recurrences and solve them.
  ▶ Instead, we just count steps
Cost Analysis for BFS

- BFS works in a sequence rounds (one per level)
- We can add up work and span in each round.
  - But work at a level depends on number of outgoing edges from the frontier!
- Take a more global view
  - Each vertex appears exactly once in some frontier.
  - All their (out-)edges are processed once.
- \( W_{BFS}(n, m) = W_v n + W_e m \)
  - \( n = |V| \) and \( m = |E| \)
Costs Analysis for BFS

- Span is a bit more tricky!
- \( S_{BFS}(n, m, d) = S_l d \) where \( d \) is the maximum distance \( (d = \max_{v \in V} \delta(s, v)) \)
- Assuming \( W_v = O(\log n) \) and \( W_e = O(\log n) \) and span/level \( S_l = O(\log^2 n) \)

\[
W_{BFS}(n, m) = O(n \log n + m \log n) \\
= O(m \log n) \quad \text{(Why?)}
\]

\[
S_{BFS}(n, m, d) = O(d \log^2 n)
\]
Costs per Vertex and Edge

- Nontrivial operations are
  1. \( X' = X \cup F \)
  2. \( N = N_G(F) \)
  3. \( F' = N \setminus X' \).

- These all depend on size of \( F \) and number of outgoing edges from \( F \).

- Let \( \|F\| = \sum_{v \in F} (1 + d_G^+(v)) \)
  - Vertices and outgoing edges in \( f \).
## Costs per Vertex and Edge

<table>
<thead>
<tr>
<th></th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \cup F$</td>
<td>$O(</td>
<td>F</td>
</tr>
<tr>
<td>$N \setminus X'$</td>
<td>$O(</td>
<td>F</td>
</tr>
</tbody>
</table>

These come from set cost specs.

\[
Work = O(W_c \cdot |F| \log(1 + \frac{n}{|F|})) = O(|F| \log n)
\]

\[
Span = O(S_c \cdot \log(n + |F|)) = O(\log n)
\]
Graph is represented as a table mapping vertices to a set of their outneighbors.

```ocaml
fun \( N_G(F) = Table.reduce \text{Set.Union} \{ \}
\text{Table.extract}(G, F) \)
```

Extract vertices from table: Work is \( O(|F| \log n) \)
fun \( N_G(F) = \text{Table.reduce Set.Union} \{\} \)  
\( (\text{Table.extract}(G,F)) \)

\( \mathcal{R}(\text{reduce } f \parallel S) = \{ \text{all function applications } f(a, b) \text{ in the reduction tree} \} \).

\[
W(\text{reduce } f \parallel S) = O \left( n + \sum_{f(a, b) \in \mathcal{R}(f \parallel S)} W(f(a, b)) \right)
\]

\[
S(\text{reduce } f \parallel S) = O \left( \log n \max_{f(a, b) \in \mathcal{R}(f \parallel S)} S(f(a, b)) \right)
\]
**Lemma**

For any combine function $f : \alpha \times \alpha \to \alpha$ and a monotone size measure $s : \alpha \to \mathbb{R}^+$, if for any $x, y$,

1. $s(f(x, y)) \leq s(x) + s(y)$ and
2. $W(f(x, y)) \leq c_f (s(x) + s(y))$ for some universal constant $c_f$ depending on the function $f$,

then

$$W(\text{reduce } f \parallel S) = O\left(\log |S| \sum_{x \in S} (1 + s(x))\right)$$
In our case $\alpha$ is the set type, $f$ is $\text{Set.union}$, $s$ the size of a set.

1. Size of the union $\leq$ sum of the sizes.
2. Work of a union $\leq$ is at most proportional to size of the sets!

So $\text{Set.union}$ satisfies the conditions of the lemma.

$F_{\text{ ngh}} = \text{Table.extract}(G, F)$

- $F_{\text{ ngh}}$ is a set of neighbor sets.

$$W(\text{reduce union } \emptyset F_{\text{ ngh}}) = O\left(\log|F_{\text{ ngh}}| \sum_{ngh \in F_{\text{ ngh}}} (1 + |ngh|)\right)$$

$$= O(\log n \cdot ||F||)$$
Back to Costs

\[ S(\text{reduce union } \{\} F_{\text{ ngh}}) = O(\log^2 n) \]

- Each union has span \( O(\log n) \)
- The reduction tree is bounded by \( \log n \) depth.
- So at level \( i \), \( W = O(||F_i|| \cdot \log n) \) and each edge is processed once, \( \Rightarrow \)
  - work per edge is \( O(\log n) \).
- Span depends on \( d \)
  \[
  (S_{\text{ BFS}}(n, m, d) = O(d \log^2 n))
  \]
  - In worst \( d \in O(n) \) \( \Rightarrow \) BFS is sequential.
BFS WITH ST SEQUENCES

- BFS Costs revisited

\[
\begin{align*}
W_{BFS}(n, m) &= O(m \log n) \\
S_{BFS}(n, m, d) &= O(d \log^2 n)
\end{align*}
\]

- Using single-threaded sequences reduces costs to

\[
\begin{align*}
W_{BFS}(n, m) &= O(m) \\
S_{BFS}(n, m, d) &= O(d \log n)
\end{align*}
\]
BFS with ST Sequences

- Vertices are labeled with integers:
  - \( V = \{0, 1, \ldots, n - 1\} \)
  - Integer labeled (IL) graphs.
- We use (array) sequences to represent graphs.
  - Constant work access to vertices.
  - Neighbors also stored as integer indices
- IL graphs are implemented with type
  \((\text{int seq}) \text{ seq}\)
BFS with ST Sequences

- BFS returns a mapping from each vertex to its parent in the BFS tree.
- Visited vertices are maintained as
  (int option) stseq
    - NONE: Vertex has not been visited.
    - SOME(v): Vertex visited from parent v.
BFS WITH ST SEQUENCES

fun BFS(G : (int seq) seq, s : int) =
  let
    fun BFS′(XF : int option stseq, F : int seq) =
      if |F| = 0 then stSeq.toSeq XF
      else let
        % compute neighbors of the frontier
        val N = flatten(⟨(u, SOME(v)) : u ∈ G[v] & XF[u] = NONE⟩ : v ∈ F)
        % add new parents
        val XF′ = stSeq.inject(N, XF)
        % remove duplicates
        val F′ = ⟨u : (u, v) ∈ N | XF′[u] = v⟩
        in BFS′(XF′, F′) end
    in
      BFS′(stSeq.update(s, SOME(s), X₀), ⟨s⟩)
  end

BREADTH-FIRST SEARCH
CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS
FALL 2013
## Costs

<table>
<thead>
<tr>
<th></th>
<th>$XF: stseq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>work span</td>
</tr>
<tr>
<td>flatten</td>
<td>$O(</td>
</tr>
<tr>
<td>inject</td>
<td>$O(</td>
</tr>
<tr>
<td>remove dup.</td>
<td>$O(</td>
</tr>
<tr>
<td>total across all $d$ rounds</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>
SUMMARY

- Breadth-first search
- BFS Extensions
- BFS Costs
- BFS with Single-threaded Sequences
15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

Lecture 11
DEPTH-FIRST SEARCH
SYNOPSIS

- Depth-first search
- Cycle-detection in directed and undirected graphs
- Topological Sorting
- Generalizing DFS
- DFS with Single-threaded Sequences
**Graph Search**

- Fundamental operation of graphs
  - Start at some (set of) node(s)
  - Systematically visit all reachable nodes (only once)
- Used for determining properties of graphs/nodes
  - Connected?
  - Bipartite?
  - Node \( v \) reachable from node \( u \)?
  - Shortest path from \( u \) to \( v \)?
Graph Search Methods

- Breadth-first Search (BFS)
  - Parallelizable but for shallow graphs!
- Depth-first Search (DFS)
  - Inherently sequential!
- Priority-first Search (PFS)
- All reachable nodes from a source are visited, but in different orders.
Breadth-first Search

- Applicable to a variety of problems
  - Connectedness
  - Reachability
  - Shortest path
  - Diameter
  - Bipartedness

- Applicable to both directed and undirected graphs
  - For digraphs, we only consider outgoing arcs.
Graph Search

For all graph search methods vertices can be partitioned into three sets at any time during the search:

1. vertices already visited \( (X \subseteq V) \),
2. the unvisited neighbors of the visited vertices, called the frontier \( (F) \),
3. the rest; unseen vertices.

The search essentially goes as follows:

while vertices remain
   - visit some unvisited neighbors of the visited set

Web navigation analogy.
**Taking CS Courses**

- Take the following courses – but one per semester

```
122 150 210 213 251
451 410 359
412 859 712 750
```

- What are some possible orders?

**Depth-first Search**

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
This problem is known as topological sorting.
- Put vertices in a linear order that respects the graph precedence relationships.

How can we know if a schedule is even possible?
- There should be no cycles!

Both these problems can be solved by depth-first search (DFS)
- DFS looks at any edge at most twice.
DFS vs BFS

BFS

- Explores vertices one level at a time.
  - Increases breadth
  - No backtracking
- Can solve/generate
  - reachability
  - connectedness
  - spanning tree
- Not suitable for topological sort

DFS

- Explores vertices one vertex at a time.
  - Increases depth
  - Backtracking when it can’t go deeper
- Can solve/generate
  - reachability
  - connectedness
  - spanning tree
- Not suitable for shortest unweighted path
DFS vs. BFS

Depth-first Search

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
DFS vs. BFS

Depth-first Search (DFS) and Breadth-first Search (BFS) are two fundamental graph search algorithms. DFS explores the graph depth-wise, visiting as far as possible along each branch before backtracking. BFS explores the graph breadth-wise, visiting all the neighbors of the current node before moving to the next level of nodes.
The DFS Algorithm

: fun DFS(G, s) = let
: fun DFS′(X, v) =
: if (v ∈ X)
TOUCH v :
: then X
: else let
ENTER v :
val X′ = X ∪ {v}
: val X′′ = iter DFS′ X′ (NG(v))
EXIT v :
in X′′ end
: in DFS′({}, s) end

Each iter does a mapping of the sort \( f : \alpha \times \beta \rightarrow \alpha \)

\[
S = s_0 \\
\text{foreach } a \in A : \\
S = f(S, a) \\
\text{return } S
\]
SOME OBSERVATIONS

- **iter** goes sequentially
  - Sets are unordered, ordering depends on implementation!
- **When a vertex** $v$ **is entered (ENTER $v$) in code**
  - it picks the “first” outgoing edge $(v, w_1)$
  - through **iter** calls **DFS’($X \cup \{v\}, w_1$)**
- **When DFS’($X \cup \{v\}, w_1$) returns**
  - All vertices reachable from $w_1$ are explored
  - Vertex set returned is
    \[ X_1 = X \cup \{v\} \cup \{\text{All vertices reachable from } w_1\} \]
- **iter** picks next edge $(v, w_2)$ and continues
- **When iter** is done
  \[ X'' = X \cup \{v\} \cup \{\text{All vertices reachable from } v\} \]
TOUCHING, ENTERING AND EXITING

\[
\text{fun } \text{DFS}(G, s) = \text{let } \\
\text{fun } \text{DFS}'(X, v) = \\
\text{if } (v \in X) \text{ TOUCH } v : \text{ then } X \\
\text{else let } \\
\text{ENTER } v : \text{ val } X' = X \cup \{v\} \\
\text{val } X'' = \text{iter DFS'} X' (N_G(v)) \\
\text{EXIT } v : \text{ in } X'' \text{ end } \\
\text{in DFS'({}, s) end }
\]

- We try to visit a vertex \(v\)
- We process \(v\) and its outgoing edges.
- We are done with \(v\).
DFS with Parallelism

Can we do all outgoing edges in parallel?

- Yes - if parallel searches never meet up (then we really have a tree!)
- No - otherwise.
**LEMMA**

For a graph $G = (V, E)$ with $m$ out edges and $n$ vertices:

- DFS' will be called at most $m$ times.
- There will be at most $\min(n, m)$ “enters”.
- $v \in X$ can fail at most $m$ times.
- we make call to DFS', when we have an edge (total $m$ times)
  - But we can enter a vertex a most once per DFS'
- So number of enters $\leq \min(n, m)$
**Cost of DFS**

**Corollary**

The DFS algorithm on a graph with $m$ out edges, and $n$ vertices, and using the tree-based cost specification for sets, runs in $O(m \log n)$ work and span.

- Using ST sequences reduces work and span to $O(m)$
Cycle Detection in Undirected Graphs

- DFS’ arrives at \( v \) a second time and this time from \( u \). What can we conclude?
  - There must be two paths between \( u \) and \( v \)! (Why?)
- Not really! In undirected graphs cycles should have length at least 3.
Cycle Detection in Undirected Graphs

```haskell
fun undirectedCycle(G, s) = let
  fun DFS' p ((X, C), v) =
    if (v ∈ X)
      TOUCH v
    then (X, true)
    else let
      ENTER v:
        val X' = X ∪ {v}
        val (X'', C') = iter (DFS' v) (X', C) (N_G(v) \ {p})
      EXIT v:
        in (X'', C') end
    in DFS' s ({}, false, s) end

C keeps tracks of cycles.

p is the parent – removed from neighbors and curried!
```
**Topological Sorting**

- Order the vertices so that the ordering respects reachability.
  - If \( u \) is reachable from \( v \), \( v \) must come earlier in the ordering.

![Graph with vertices and edges]

**Depth-first Search**
Partial Orders

- A DAG defines a partial order on the vertices.
- For vertices $a, b \in V$, $a \leq_p b$ if and only if there is a directed path from $a$ to $b$
- Partial order is a relation $\leq_p$ that obeys
  1. reflexivity — $a \leq_p a$,
  2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$,
  and
  3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$. 
Topological Sort

- If $\leq_t$ is the total ordering then
  
  $a \leq_p b \Rightarrow a \leq_t b$

  but not reverse is not necessarily true!

\[
a \leq_t b \leq_t c \leq_t d \leq_t e \leq_t f \leq_t g \leq_t h
\]
**Topological Sort with DFS**

- Augment with a new source vertex $s$
  
  $$G = (V, E) \rightarrow G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$$

- Why do we need to do this?

```ml
fun topSort(G = (V, E)) = let
  val s = a new vertex
  val G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})
  fun DFS'((X, L), v) =
    if (v \in X)
      TOUCH v
      then (X, L)
    else let
      ENTER v:
      val X' = X \cup \{v\}
      val (X'', L') = iter DFS' (X', L) (NG'(v))
      EXIT v:
      in (X'', v :: L') end
    in DFS'((\{\}, []), s) end
```

Depth-first Search 23/33
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms Fall 2013
**Topological Sort with DFS**

**Theorem**

On a DAG when exiting a vertex $v$ in DFS all vertices reachable from $v$ have already exited.

- Assume $u$ is reachable from $v$.
  - $u$ is entered before $v$. $u$ must exit before $v$ is entered (otherwise there is a cycle!)
  - $v$ is entered before $u$. $u$ will exit first.
Cycle Detection in Directed Graphs

- Important preprocessing step in Topological Sort
  - Topological sort will return garbage when graph has cycles.

- We augment the graph with a node $s$ with an edge to every other vertex the graph.
  - This can not add cycles. Nothing comes into $s$.  
  

Depth-first Search
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
Cycle Detection in Directed Graphs

fun directedCycle(G = (V, E)) = let
  val s = a new vertex
  val G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
  fun DFS′((X, Y, C), v) = 
    if (v ∈ X)
    then (X, Y, v ∈ Y)
    else let
      ENTER v:
        val X' = X ∪ {v}
        val Y' = Y ∪ {v}
        val (X'', Y'', C') = iter DFS' (X', Y', C) (NG(v))
      EXIT v:
        in (X'', Y'' \ {v}, C') end
    in (X, Y, C) = DFS'(({}, {}, false), s)
  in C end
**Back edges in a DFS Search**

- A back edge goes from a vertex \( v \) to an ancestor \( u \) in the DFS tree.

**Theorem**

A directed graph \( G = (V, E) \) has a cycle if and only if for \( G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\}) \) a DFS from \( s \) has a back edge.
**Generalizing DFS**

- All DFS code seem very much alike.
- They do work on
  - Touching,
  - Entering, and
  - Exiting
- We need to keep some state $\sigma$ around
  - and update it appropriately!

$$
\sum_0 : \alpha \\
touch : \alpha \times \text{vertex} \times \text{vertex} \rightarrow \alpha \\
enter : \alpha \times \text{vertex} \times \text{vertex} \rightarrow \alpha \\
exit : \alpha \times \text{vertex} \times \text{vertex} \rightarrow \alpha
$$
Generic DFS Algorithm

fun DFS(G, Σ₀, s) = let
  fun DFS' p ((X, Σ), v) =
    if (v ∈ X) then (X, touch(Σ, v, p))
  else let
    val Σ' = enter(Σ, v, p)
    val (X', Σ'') = iter (DFS' p) (X ∪ {v}, Σ') N⁺(v)
    val Σ''' = exit(Σ, Σ'', v, p)
  in (X', Σ''') end
  in DFS' s (({}, Σ₀), s) end
**Undirected Cycle Detection**

1. \( \text{fun } \text{DFS}(G, \Sigma_0, s) = \text{let} \)
2. \( \text{fun } \text{DFS}' \ p \ ((X, \Sigma), v) = \)
3. \( \text{if } (v \in X) \text{ then } (X, \text{touch}(\Sigma, v, p)) \)
4. \( \text{else let} \)
5. \( \text{val } \Sigma' = \text{enter}(\Sigma, v, p) \)
6. \( \text{val } (X', \Sigma'') = \text{iter}(\text{DFS}' \ p) (X \cup \{v\}, \Sigma') \ N^+_G(v) \)
7. \( \text{val } \Sigma''' = \text{exit}(\Sigma, \Sigma'', v, p) \)
8. \( \text{in } (X', \Sigma'') \text{ end} \)
9. \( \text{in } \text{DFS}' \ s ((\{\}, \Sigma_0), s) \text{ end} \)

\( \Sigma_0 = ([s], \text{false}) : \text{vertex list} \times \text{bool} \)

\( \text{fun } \text{touch}((L \text{ as } h :: T, fl), v, p) = (L, h \neq p) \)

\( \text{fun } \text{enter}((L, fl), v, p) = (v :: L, fl) \)

\( \text{fun } \text{exit}((L \text{ as } h :: T, fl), v, p) = (T, fl) \)
TOPOLOGICAL SORT

1 fun DFS(G, Σ₀, s) = let
2 fun DFS' p ((X, Σ), v) =
3   if (v ∈ X) then (X, touch(Σ, v, p))
4     else let
5       val Σ' = enter(Σ, v, p)
6       val (X', Σ'') = iter (DFS' p) (X ∪ {v}, Σ') \ N_G⁺(v)
7       val Σ''' = exit(Σ, Σ'', v, p)
8     in (X', Σ'''') end
9   in DFS' s (({}, Σ₀), s) end

Σ₀ = [] : vertex list
fun touch(L, v, p) = L
fun enter(L, v, p) = L
fun exit(L, v, p) = v :: L
fun \( \text{DFS}(G, \Sigma_0, s) = \) let
fun \( \text{DFS}' \ p \ ((X, \Sigma), v) = \)
   if \( v \in X \) then \( (X, \text{touch}(\Sigma, v, p)) \)
   else let
      val \( \Sigma' = \text{enter}(\Sigma, v, p) \)
      val \( (X', \Sigma'') = \text{iter}(\text{DFS}' \ p) \ (X \cup \{v\}, \Sigma') \ N_G^+(v) \)
      val \( \Sigma''' = \text{exit}(\Sigma, \Sigma'', v, p) \)
   in \( (X', \Sigma''') \) end
in \( \text{DFS}' \ s \ ((\{}{}, \Sigma_0\), s) \) end

\( \Sigma_0 = (\{}{}, \ false): \text{Set} \times \text{bool} \)
fun \( \text{touch}((S, f1), v, p) = (S, v \in S) \)
fun \( \text{enter}((S, f1), v, p) = (S \cup \{v\}, f1) \)
fun \( \text{exit}((S, f1), v, p) = (S \setminus \{v\}, f1) \)
DFS WITH ST SEQUENCES

fun DFS(G: (int seq) seq, s: int) =
  let
    fun DFS′ p ((X: bool stseq, Σ), v: int) =
      if (X[v]) then (X, touch(Σ, v, p))
      else let
        val X′ = update(v, true, X)
        val Σ′ = enter(Σ, v, p)
        val (X′′, Σ′′) = iter (DFS′v) (X′, Σ′) (G[v])
      in (X′′, exit(Σ′′, v, p))
    in
      X_init = stSeq.fromSeq(⟨false: v ∈ ⟨0, ..., |G| − 1⟩⟩)
      stSeq.toSeq(DFS′((X_init, Σ₀), s))
  end

O(m) work and span.
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 13
Shortest Weighted Paths
SYNOPSIS

- Representing weighted graphs.
- Priority-first Search
- Shortest weighted paths
- Dijkstra’s Algorithm
WEIGHTED GRAPH REPRESENTATION

- \( G = (V, E, w) \) where \( w : E \rightarrow \text{eVal} \)
- \( \text{eVal} \) is a set (type) of possible values
  - Typically real numbers, but could be anything!
- Table of \((\text{edge} \mapsto \text{weight})\).

\[
W = \{(0, 1) \mapsto 0.7, (1, 2) \mapsto -2.0, (0, 2) \mapsto 1.5\}
\]

- We could use \( \text{find} \ W \ e \) to find \( w(e) \).
**Weighted Graph Representation**

- Table of \((vertex \mapsto \text{table of } (vertex \mapsto weight))\)

  \[ G = \{0 \mapsto \{1 \mapsto 0.7, \ 2 \mapsto 1.5\}, \ 1 \mapsto \{2 \mapsto -2.0\}, \ 2 \mapsto \{}\} \] .

- With one lookup, we can get to the neighbors and weights.
- We will mostly use this representation.
Priority-first Search

- Generalization of BFS and DFS – also called best-first search
- Visits vertices in some priority order
  - Static - decided ahead of time
  - Dynamic – decided on the fly– while things change during the search
**Priority-first Search**

1. \( \text{fun} \ pfs(G, s) = \text{let} \)
2. \( \quad \text{fun} \ pfs'(X, F) = \)
3. \( \quad \text{if} \ (F = \{\}) \ \text{then} \ X \)
4. \( \quad \text{else} \ \text{let} \)
5. \( \quad \quad \text{val} \ M = \text{highest priority vertices in} \ F \)
6. \( \quad \quad \text{val} \ X' = X \cup M \)
7. \( \quad \quad \text{val} \ F' = (F \cup N(M)) \setminus X' \)
8. \( \quad \text{in} \ pfs'(X', F') \ \text{end} \)
9. \( \text{in} \ pfs'({}, \{s\}) \ \text{end} \)
Several famous graph algorithms are instances of priority-first search.

- Dijkstra’s Algorithm for single-source shortest paths (SSSP).
- Prim’s Algorithm for minimum spanning trees (MST).

PFS is a greedy algorithm.

- It greedily adds vertices from the frontier based on a cost function.
- It never backs up!
**Shortest Weighted Paths**

- \( G = (V, E, w) \) with \( w : E \to \mathbb{R} \)
  - \( w(u, v) = \infty \) if \( (u, v) \notin E \)
- The weight of a path is the sum of the weights of the edges on it.

**The SSSP Problem**

- Given a graph \( G \) and a source vertex \( s \), find the shortest weighted path to every other vertex.
  - \( \delta_G(u, v) \) is the weight of the shortest path from \( u \) to \( v \).
Dijkstra’s Algorithm

- Dijkstra’s Algorithm solves SSSP when the weights are non-negative \( w : E \to \mathbb{R}_+ \cup \{0\} \).
  - Greedy
  - Finds optimal solutions to a nontrivial task.
- Why do we need a new algorithm? Why not use BFS?

![Graph 1]

![Graph 2]
Observations

Which vertices can we definitely claim we know the shortest path from $s$?

- $s$ itself (Why?)
- The vertex $v$ nearest to $s$ (Why?)

In general

- if we know the shortest path distances to a set of vertices
- how can we determine the shortest path to another vertex?
**Dijkstra’s Algorithm**

- At any point in time we know the exact shortest path weight from $s$
  - to vertices in $X \subseteq V$ ($s \in X$), and
  - to some vertex $v \in T$ ($= V \setminus X$) that is closest to some vertex in $X$

  based on paths going through only vertices in $X$.
- Thus, expand $X$ by considering only the nearest neighbors of the vertices visited!
- Define $\delta_{G,X}(s, v)$ to be the shortest path length from $s$ to $v$ in $G$ that only goes through vertices in $X$ (except for $v$)
Dijkstra’s Property

Consider a graph
- \( G = (V, E), \) \( w : E \to \mathbb{R}_+ \cup \{0\}, \)
- a source vertex \( s \in V \)

For any partitioning of the vertices \( V \) into \( X \) and \( T = V \setminus X \) with \( s \in X \),

\[
\min_{t \in T} \delta_{G,X}(s, t) = \min_{t \in T} \delta_G(s, t).
\]

What is this assertion saying?
- The actual shortest distance to the vertex in \( T \) that is closest to \( s \), has to go through vertices in \( X \)!
Consider a vertex $v_m \in T$ such that $\delta_G(s, v_m) = \min_{t \in T} \delta_G(s, t)$, and a shortest path from $s$ to $v_m$ in $G$. The path must cross from $X$ to $T$ at some point using some edge $(v_X, v_T)$. (Prefix) Subpaths of shortest paths are also shortest paths! (Why?)
Dijkstra’s Property

- The subpath from $s$ to $v_T$ is the shortest path to $v_T$
- Since edges weights are $\geq 0$
  \[ \delta_G(s, v_T) \leq \delta_G(s, v_m) \]
  - It could be that $v_T = v_m$.
- Also, $\delta_{G,X}(s, v_T) = \delta_G(s, v_T)$ (Why?)
Dijkstra's Property

\[
\min_{t \in T} \delta_{G,X}(s, t) \leq \delta_{G,X}(s, v_T) = \delta_G(s, v_T) \leq \delta_G(s, v_m) = \min_{t \in T} \delta_G(s, t).
\]

But

\[
\min_{t \in T} \delta_{G,X}(s, t) \geq \min_{t \in T} \delta_G(s, t) \quad \text{Why?}
\]

\[
\Rightarrow \min_{t \in T} \delta_{G,X}(s, t) = \min_{t \in T} \delta_G(s, t).
\]
Dijkstra’s Property

- We can find the shortest path to a node in $T = V \setminus X$, by just considering neighbors of $X$.
- Pick a vertex $t \in T$ that minimizes priority $\delta_{G,X}(s, t)$.
- At that point
  - $\delta_{G,X}(s, t)$ becomes $\delta_G(s, t)$ - we now know the exact shortest path length to $t$.
  - $X = X \cup \{t\}$, that is, $t$ is now visited.
  - $T = T \setminus \{t\}$
  - $\delta_{G,X}(s, u)$ where $u \in T$ and $(t, u) \in E$ must be updated. (Why?/How?)
Dijkstra’s Property – Updates

Before

\[ \partial_{G,X}(s, u) \]

\[ w(t, u) \]

\[ \partial_{G,X}(s, t) \]

After

\[ \text{new } \partial_{G,X}(s, u) = \min_{x \in X} (\partial_G(s, x) + w(x, u)) \]
Dijkstra’s Algorithm

Given a weighted graph \( G = (V, E, w) \) and a source \( s \), Dijkstra’s algorithm is priority first search on \( G \)

- starting at \( s \), with \( d(s) = 0 \) (and \( d(\nu) = \infty, \nu \neq s \))
- using priority \( P(\nu) = \min_{t \in V} (d(t) + w(t, \nu)) \) (to be minimized)
- setting \( d(\nu) = P(\nu) \) when \( \nu \) is visited.
Dijkstra’s Algorithm

**Lemma**

- When Dijkstra’s Algorithm returns $d(v) = \delta_G(s, v)$ for all reachable $v$,

- Base case is true: $d(s) = 0$.
- Assume true for $|X| = i$, then add vertex that minimizes $P(v) = \delta_{G, X}(s, v)$.
- By Dijkstra’s Property we know $\min_{v \in T} \delta_{G, X}(s, v) = \min_{v \in T} \delta_G(s, v)$
- So $d(v) = \delta_G(s, v)$ for $|X| = i + 1$. 
**Dijkstra's Algorithm**

```haskell
1 fun dijkstra(G, s) =
2 let
3   fun dijkstra′(X, Q) =
4     case PQ.deleteMin(Q) of
5       (NONE, _) ⇒ X
6       | (SOME(d, v), Q′) ⇒
7         if (v, _) ∈ X then dijkstra′(X, Q′)
8         else let
9           val X′ = X ∪ {(v, d)}
10          fun relax (Q, (u, w)) = PQ.insert(d + w, u) Q
11          val Q″ = iter relax Q′ N_G(v)
12           in dijkstra′(X′, Q″) end
13   in
14     dijkstra′({}, PQ.insert(0, s) {})
15 end
```

Shortest Weighted Paths
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
Dijkstra Variants

- Update the neighbors in the priority queue instead of adding duplicates.
  - PQ needs to support `decreaseKey` function.
- Visit all equally closest vertices in parallel (like BFS)
  - Potentially not much parallelism!
  - PQ needs to return all such vertices.
Dijkstra in Action

Dijkstra's algorithm

Shortest Weighted Paths

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Dijkstra in Action

Dijkstra's algorithm

\[
\begin{array}{l}
X \\
\text{s} \rightarrow 0 \\
Q \\
2 \rightarrow b \\
4 \rightarrow c \\
5 \rightarrow a
\end{array}
\]

Shortest Weighted Paths

CMU-Q  15-210 Parallel and Sequential Data Structures and Algorithms
Dijkstra in Action

Shortest Weighted Paths
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Dijkstra in Action

Dijkstra’s algorithm

S

HORTEST
WEIGHTED
PATHS

CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS

FALL 2013

Q

4→c
4→a
5→a
7→e

X

s→0
b→2
d→3

SHORTEST WEIGHTED PATHS 25/32

CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS

FALL 2013
Dijkstra in Action

Shortest Weighted Paths
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms Fall 2013
Dijkstra in Action

Dijkstra's algorithm 

\[ X \]
- \( s \rightarrow 0 \)
- \( b \rightarrow 2 \)
- \( d \rightarrow 3 \)
- \( c \rightarrow 4 \)
- \( a \rightarrow 4 \)

\[ Q \]
- \( 5 \rightarrow a \)
- \( 5 \rightarrow b \)
- \( 6 \rightarrow f \)
- \( 7 \rightarrow e \)

Graph:
- Nodes: \( s, a, b, d, e, f, g \)
- Edges with weights:
  - \( s \rightarrow a: 5 \)
  - \( a \rightarrow s: 1 \)
  - \( s \rightarrow b: 2 \)
  - \( b \rightarrow s: 1 \)
  - \( b \rightarrow d: 5 \)
  - \( d \rightarrow b: 1 \)
  - \( d \rightarrow f: 2 \)
  - \( f \rightarrow d: 2 \)
  - \( e \rightarrow b: 5 \)
  - \( f \rightarrow e: 0 \)

Formulas:

- X
- Q

Shortest Weighted Paths

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
**Dijkstra in Action**

![Graph with nodes and edges labeled with weights]

- **X**
  - s → 0
  - b → 2
  - d → 3
  - c → 4
  - a → 4
  - f → 6

- **Q**
  - 6 → e
  - 7 → e

**Shortest Weighted Paths**
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
Dijkstra in Action

Shortest Weighted Paths
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
Dijkstra’s Algorithm

1 fun dijkstra(G, s) =
2 let
3   fun dijkstra′(X, Q) =
4     case PQ.deleteMin(Q) of
5         (NONE, _) ⇒ X
6         | (SOME(d, v), Q′) ⇒
7             if ((v, _) ∈ X) then dijkstra′(X, Q′)
8             else let
9                 val X′ = X ∪ {(v, d)}
10                fun relax (Q, (u, w)) = PQ.insert(d + w, u) Q
11                val Q″ = iter relax Q′ N_G(v)
12                in dijkstra′(X′, Q″) end
13           in
14           dijkstra′({}, PQ.insert(0, s) {})
15 end

Shortest Weighted Paths
COST ANALYSIS

- Priority Queue with $O(\log n)$ work and span.
- Graphs: tree-based table or arrays
- Table of distances: tree-based table, array sequence, or ST sequence.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Line</th>
<th># of calls</th>
<th>PQ</th>
<th>Tree Table</th>
<th>Array</th>
<th>ST Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteMin</td>
<td>4</td>
<td>$O(m)$</td>
<td>$O(\log m)$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>insert</td>
<td>10</td>
<td>$O(m)$</td>
<td>$O(\log m)$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Priority Q total</td>
<td></td>
<td></td>
<td>$O(m \log m)$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>find</td>
<td>7</td>
<td>$O(m)$</td>
<td>-</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>9</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Distances total</td>
<td></td>
<td></td>
<td>-</td>
<td>$O(m \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>$N_G(v)$</td>
<td>11</td>
<td>$O(n)$</td>
<td>-</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>-</td>
</tr>
<tr>
<td>iter</td>
<td>11</td>
<td>$O(m)$</td>
<td>-</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>-</td>
</tr>
<tr>
<td>Graph access total</td>
<td></td>
<td></td>
<td>-</td>
<td>$O(m + n \log n)$</td>
<td>$O(m)$</td>
<td>-</td>
</tr>
</tbody>
</table>

Using a tree table work is $O(m \log n)$. 
Representing weighted graphs.
Priority-first Search
Shortest weighted paths
Dijkstra’s Algorithm
SYNOPSIS

- Graphs with negative edge weights.
- Bellman Ford Algorithm
- Cost Analysis
What is a problem with this graph?
Graphs with Negative Weights

Dijkstra fails! (Why?)
What is a problem with this graph?
Negative cost cycle!
There is no shortest path from $v_3$ to $v_5$
**Graphs with Negative Weights**

- Currency Exchange Arbitrage

![Graph Diagram]

- 100 USD → 365 QAR → 177.5 TL → 80.68 EUR → 104.9 USD

  - You just made 5 USD out of thin air!
I have USDs but I want to buy BPs.
- I can buy directly, or
- I can buy through some intermediate currencies!

Which way will get me more BPs?
- I need to do this fast!
Shortest Paths

- How does this problem relate to the shortest problem?
  - Where are the negative weights?

Weights are $-\log$ of the exchange rates!
**Shortest Paths with Negative Weights**

- Define $\delta^l_G(s, t)$ the shortest weighted path from $s$ to $t$ using at most $l$ edges.
  - so the unweighted path length is $l$
- Base cases:
  - $\delta^0_G(s, s) = 0$
  - $\delta^0_G(s, v) = \infty$ for all $v \neq s$.
- Induction
  \[
  \delta^{k+1}(v) = \min_{x \in N^-(v)} (\delta^k(x) + w(x, v)).
  \]
- Minimum of $\delta^k(x) + w(x, v)$ over the in-neighbors.
THE BELLMAN FORD ALGORITHM

1 fun BellmanFord(G = (V, E), s) =
 2 let
 3   fun BF(D, k) =
 4     let
 5       val D' = {v ↦→ min_{u ∈ N^-_G(v)}(D_u + w(u, v)) : v ∈ V}
 6     in
 7       if (k = |V|) then ⊥
 8       else if (∀{D_v = D'_v : v ∈ V}) then D
 9       else BF(D', k + 1)
10   end
11  val D = {v ↦→ if v = s then 0 else ∞ : v ∈ V}
12  in BF(D, 0) end
How Bellman Ford Algorithm Works

path lengths = 0
HOW BELLMAN FORD ALGORITHM WORKS

path lengths ≤ 1
How Bellman Ford Algorithm Works

![Graph diagram with nodes and edges labeled with weights.](image)

Path lengths \( \leq 2 \)
How Bellman Ford Algorithm Works

path lengths ≤ 3
**How Bellman Ford Algorithm Works**

Path lengths ≤ 4

![Graph](image-url)
**Bellman Ford Correctness**

**Theorem**

Given a directed weighted graph $G = (V, E, w)$, $w : E \to \mathbb{R}$, and a source $s$, the BellmanFord algorithm returns the shortest path length from $s$ to every vertex or indicates that there is a negative weight cycle in $G$ reachable from $s$. 

---

**Shortest Weighted Paths–II**

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
Use induction on the number of edges $k$ in the path.

Base case is correct, $D_s = 0$.

On each step, for all $v \in V \setminus \{s\}$, a shortest path with at most $k + 1$ edges
- must consist of a path of at most $k$ edges for vertex $u$
- followed by a single edge $(u, v)$.

Taking the minimum combination, gives us the shortest path with at most $k + 1$ edges.
NEGATIVE Cost CYCLES

- This can go at most for \( n = |V| - 1 \) rounds.
- If we reach round \( n \), there must be reachable negative cost cycle.
- Otherwise, Bellman Ford will stop earlier with all simple shortest paths.
Cost Analysis

- Graph represented as a table.
  - $(\mathbb{R} \text{ vtxTable}) \text{ vtxTable}$, where first $\text{ vtxTable}$ maps vertices to their in-neighbors

  $G = \{0 \mapsto \{1 \mapsto 0.7, 2 \mapsto 1.5\}, 1 \mapsto \{2 \mapsto -2.0\}, 2 \mapsto \{\}\}.$

- Graph represented as a sequence of sequences.
  - $((\text{int } \times \text{eVal}) \text{ seq}) \text{ seq}$
Bellman-Ford Algorithm (Again)

1 fun BellmanFord(G = (V, E), s) =
2 let
3 fun BF(D, k) =
4 let
5 val D′ = \{ v ↦ min_{u \in N_{G}^{-}(v)}(D_u + w(u, v)) : v ∈ V \}
6 in
7 if (k = |V|) then ⊥
8 else if (all\{D_v = D'_v : v ∈ V\}) then D
9 else BF(D′, k + 1)
10 end
11 val D = \{ v ↦ if v = s then 0 else ∞ : v ∈ V \}
12 in BF(D, 0) end

- Line 5 is tabulate over the vertices
- Line 8 is tabulate with a reduction over the vertices
Cost Analysis

\[ \text{val } D' = \{ v \mapsto \min_{u \in N^{-}_G(v)} (D_u + w(u, v)) : v \in V \} \]

- Sum work and max span over vertices.
- \( n = |V| \) and \( m = |E| \)
- For each vertex we have the following costs:
  - Find the neighbors \( \text{find } G \ v \): \( O(\log n) \) work and span.
  - Map over neighbors – find distance \( D_u \) and add: \( O(\log n) \) work and span for each \( u \) in the in-neighborhood.
  - Min reduce: \( O(1 + N_G(v)) \) work and \( O(\log |N_G(v)|) \) span.
**Work per Stage-1**

\[
\text{val } D' = \{ v \mapsto \min_{u \in N_G^-(v)} (D_u + w(u, v)) : v \in V \}
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Over one vertex v</th>
<th>Over graph G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>(O(\log n))</td>
<td>(O(n \log n))</td>
</tr>
<tr>
<td>Map</td>
<td>(O(1 +</td>
<td>N_G^-(v)</td>
</tr>
<tr>
<td>Min Reduce</td>
<td>(O(1 +</td>
<td>N_G^-(v)</td>
</tr>
</tbody>
</table>

- **Total work is** \(O((n + m) \log n)\) **and assuming** \(m > n, O(m \log n)\)
**Span per Stage -1**

\[ \text{val } D' = \{ v \mapsto \min_{u \in N^-_G(v)} (D_u + w(u, v)) : v \in V \} \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Over one vertex v</th>
<th>Over graph G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Map</td>
<td>( O(1 + \log n) )</td>
<td>( O(1 + \log n) )</td>
</tr>
<tr>
<td>Min Reduce</td>
<td>( O(\log</td>
<td>N^-_G(v)</td>
</tr>
</tbody>
</table>

- Total span is \( O(\log n) \)
else if (all \( \{ D_v = D'_v : v \in V \} \)) then \( D \)

- This involves a tabulate and an and-reduction.
- Work = \( O(n \log n) \), Span = \( O(\log n) \)

- \( n \) sequential calls to \( BF \), so total costs are:

\[
\begin{align*}
W(n, m) & = O(n \cdot m \log n) \\
S(n, m) & = O(n \log n)
\end{align*}
\]
Costs with ST Sequences

- We use IL (integer labeled) graphs.
- $\text{find} \rightarrow \text{nth}: O(1)$ work.
- Similar improvements for looking up neighbors and distance table.

\[
W(n, m) = O(nm) \\
S(n, m) = O(n)
\]
SUMMARY

- Graphs with negative edge weights.
- Bellman Ford Algorithm
- Analysis
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 15
Probability and Randomized Algorithms
SYNOPSIS

- Overview of Discrete Probability
- Finding the two largest elements
- Find the $k^{th}$ smallest element.
Randomized Algorithms

- Exploit randomness during computation
  - Pivot selection in Quicksort
  - Average case analysis
  - Primality testing

- **Question**: How many comparisons are needed to find the second largest number on a sequence of $n$ numbers?
  - Naive algorithm: $2n – 3$ comparisons
  - Divide and Conquer algorithm: $3n/2$ comparisons
  - Simple randomized algorithm: $n – 1 + 2 \log n$ comparisons on the average.
Overview of Discrete Probability

- **Probabilistic Experiment**: outcome is probabilistic.
- **Sample Space** ($\Omega$): arbitrary and possibly countably infinite set of possible outcomes.
  - Tossing a coin
  - Throwing a die/pair of dice.
- **Primitive Event**: Any one of the elements of $\Omega$.
- **Event**: Any subset of $\Omega$
  - First die is a 5
  - Dice sum to 7
  - Any die is even.
Probability Function

- Probability Function: $\Omega \rightarrow [0, 1]$
  \[ \sum_{e \in \Omega} \Pr[e] = 1 \]

- Probability of an event $A$:
  \[ \sum_{e \in A} \Pr[e] \]

- Probability of “first die is 4”?
- Probability of “dice sum to 4”?
Random Variables

- Random Variable: $X : \Omega \rightarrow \mathbb{R}$
  - $X$ is the sum of the two die rolls
- Indicator Random Variable: $Y : \Omega \rightarrow \{0, 1\}$
  - $Y$ is 1 if the dice are the same, 0 otherwise
  - $Y$ is 1 if the total is larger than 7, 0 otherwise
- For $a \in \mathbb{R}$, the event “$X = a$” is the set

$$\{\omega \in \Omega \mid X(\omega) = a\}$$
**EXPECTATION**

- The expectation of a random variable

\[
\mathbb{E}_{\Omega, \text{Pr}}[X] = \sum_{e \in \Omega} X(e) \cdot \text{Pr}[e].
\]

- The expectation of an *indicator* random variable:

\[
\mathbb{E}[Y] = \sum_{e \in \Omega, p(e) = \text{true}} \text{Pr}[e] = \sum_{e \in \Omega} \text{Pr}[\{e \in \Omega \mid p(e)\}].
\]

- \(p : \Omega \rightarrow \text{bool}\)
Events $A$ and $B$ are **independent** if the occurrence of one does not affect the probability of the other

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

- $A = \{(d_1, d_2) \in \Omega \mid d_1 = 1\}$ and $B = \{(d_1, d_2) \in \Omega \mid d_2 = 1\}$ are independent.
- $C = \{(d_1, d_2) \in \Omega \mid d_1 + d_2 = 4\}$ is NOT independent of $A$ (Why?)
Events $A_1, \ldots, A_k$ are mutually independent if and only if for any non-empty subset $I \subseteq \{1, \ldots, k\}$,

$$\Pr[\bigcap_{i \in I} A_i] = \prod_{i \in I} \Pr[A_i].$$

Random variable $X$ and $Y$ are independent if fixing one does NOT affect the probability distribution of the other.

- $X = \text{“value of the first die”}$ is independent of $Y = \text{“value of the second die”}$.
- $X$ is NOT independent of $Z = \text{“sum of the dice”}$.
**Linearity of Expectations**

- Important Theorem: given two random variables $X$ and $Y$
  \[ E[X] + E[Y] = E[X + Y] \]
- Easy to show!
  \[ \sum_{e \in \Omega} \Pr[e]X(e) + \sum_{e \in \Omega} \Pr[e]Y(e) = \sum_{e \in \Omega} \Pr[e](X(e) + Y(e)) \]
- Expected sum of two dice
  - Consider 36 outcomes and take average
  - Sum expectations for each dice ($3.5 + 3.5 = 7$)
In general, for a binary function $f$ the equality

$$f(\mathbb{E}[X], \mathbb{E}[Y]) = \mathbb{E}[f(X, Y)]$$

is not true in general.

- $\max(\mathbb{E}[X], \mathbb{E}[Y]) \neq \mathbb{E}[\max(X, Y)]$
- What is $\mathbb{E}[\max(X, Y)]$?

$\mathbb{E}[X] \times \mathbb{E}[Y] = \mathbb{E}[X \times Y]$ is true if $X$ and $Y$ are independent.
Toss $n$ coins with probability of heads, $p$. What is the expected value of $X$, the number of heads?

\[
E[X] = \sum_{k=0}^{n} k \cdot \Pr[X = k]
\]

\[
= \sum_{k=1}^{n} k \cdot p^k (1 - p)^{n-k} \binom{n}{k} \quad \text{(Why?)}
\]

\[
= \sum_{k=1}^{n} \frac{n}{k} \binom{n-1}{k-1} p^k (1 - p)^{n-k} \quad \text{[because } \binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1} \text{]}
\]

\[
= n \sum_{k=1}^{n} \binom{n-1}{k-1} p^k (1 - p)^{n-k}
\]
EXAMPLES

Toss $n$ coins with probability of heads, $p$. What is the expected value of $X$, the number of heads?

$$
E[X] = \sum_{k=0}^{n} k \cdot \Pr[X = k]
$$

\[
\begin{align*}
&= n \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j+1} (1 - p)^{n-(j+1)} \\
&= n \cdot p \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j} (1 - p)^{(n-1)-j} \\
&= n \cdot p \cdot (p + (1 - p))^{n-1} \\
&= n \cdot p
\end{align*}
\]

[ Binomial Theorem ]
Toss \( n \) coins with probability of heads, \( p \). What is the expected value of \( X \), the number of heads?

Using linearity of expectations.

\[
X_i = \mathbb{I}\{i\text{-th coin turns up heads}\}
\]

\[
X = \sum_{i=1}^{n} X_i
\]

\[
\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p = n \cdot p
\]

because \( \mathbb{E}[X_i] = p \).
A coin has a probability $p$ of coming up heads. What is the expected value of $Y$ representing the number of flips until we see a head? Write a recurrence!

- With probability $p$, we’ll get a head and we are done,
- With probability $1-p$, we’ll get a tail and we’ll go back to square one

\[
E[Y] = p \cdot 1 + (1-p) \left( 1 + E[Y] \right)
\]

\[
\]
FINDING THE TOP TWO ELEMENTS

1 fun max2(S) = let
2   fun replace((m1,m2),v) =
3      if v ≤ m2 then (m1,m2)
4      else if v ≤ m1 then (m1,v)
5      else (v,m1)
6   val start = if S1 ≥ S2 then (S1,S2) else (S2,S1)
7   in iter replace start S⟨3,…,n⟩
8 end

- We will do exact analysis.
- $1 + 2(n - 2) = 2n - 3$ comparisons in the worst case. (Why?)
- A Divide and Conquer algorithm gives $3n/2 - 2$
Worst Case Analysis

1 fun max2(S) = let
2   fun replace((m1,m2),v) =
3     if v ≤ m2 then (m1,m2)
4     else if v ≤ m1 then (m1,v)
5     else (v,m1)
6   val start = if S1 ≥ S2 then (S1,S2) else (S2,S1)
7   in iter replace start S⟨3,...,n⟩
8 end

- An already sorted sequence (e.g., ⟨1,2,3,...,n⟩) will need exactly 2n – 3 comparisons.
- But this happens with 1/n! chance!
A RANDOMIZED ALGORITHM

- The worst-case analysis is overly pessimistic.
- Consider the following variant:
- On input of a sequence $S$ of $n$ elements:
  1. Let $T = \text{permute}(S, \pi)$, where $\pi$ is a random permutation (i.e., we choose one of the $n!$ permutations).
  2. Run the naïve algorithm on $T$.
- No need to really generate the permutation!
  - Just pick an unprocessed element randomly until all elements are processed.
  - It is convenient to model this by one initial permutation!
**ANALYSIS**

1. \[ \text{fun max2}(S) = \text{let} \]
2. \[ \quad \text{fun replace}((m_1, m_2), v) = \]
3. \[ \quad \text{if } v \leq m_2 \text{ then } (m_1, m_2) \]
4. \[ \quad \text{else if } v \leq m_1 \text{ then } (m_1, v) \]
5. \[ \quad \text{else } (v, m_1) \]
6. \[ \quad \text{val start} = \text{if } S_1 \geq S_2 \text{ then } (S_1, S_2) \text{ else } (S_2, S_1) \]
7. \[ \quad \text{in } \text{iter replace start } S(3, \ldots, n) \]
8. \[ \quad \text{end} \]

- \( X_i = 1 \) if \( T_i \) is compared in Line 4, 0 otherwise.
- \( Y \) is the number of comparisons

\[
Y = \left( \begin{array}{c}
1 \\
\text{Line 6}
\end{array} \right) + \left( \begin{array}{c}
2n - 2 \\
\text{Line 3}
\end{array} \right) + \sum_{i=3}^{n} X_i; \\
\text{Line 4}
\]
This expression in true regardless of the random choice we’re making.

We’re interested in computing the expected value of $Y$.

By linearity of expectation,

$$
E[Y] = E \left[ 1 + (n - 2) + \sum_{i=3}^{n} X_i \right]
$$

$$
= 1 + (n - 2) + \sum_{i=3}^{n} E[X_i].
$$
Analysis

- Problem boils down to computing $E [X_i]$, for $i = 3, \ldots, n$!
- What is the probability that $T_i > m_2$?
  - $T_i > m_2$ holds when $T_i$ is either the largest or the second largest in $\{T_1, \ldots, T_i\}$
- So, what is the probability that $T_i$ is one of the two largest elements in a randomly permuted sequence of length $i$?
  - $\frac{1}{i} + \frac{1}{i} = \frac{2}{i}$
- $E [X_i] = 1 \cdot \frac{2}{i} = 2/i$
ANALYSIS

\[ E[Y] = 1 + (n - 2) + \sum_{i=3}^{n} E[X_i] \]

\[ = 1 + (n - 2) + \sum_{i=3}^{n} \frac{2}{i} \]

\[ = 1 + (n - 2) + 2\left(\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}\right) \]

\[ = n - 4 + 2\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}\right) \]

\[ = n - 4 + 2H_n \]

- \( H_n \) is the \( n^{th} \) Harmonic number
- \( H_n \leq 1 + \log_2 n \)
- \( E[Y] \leq n - 2 + 2\log_2 n \)
FINDING THE $k^{th}$ SMALLEST ELEMENT

- **Input:** a sequence of $n$ numbers (not necessarily sorted)
- **Output:** the $k^{th}$ smallest value in $S$ (i.e., $(\text{nth (sort } S) k)$).
- **Requirement:** $O(n)$ expected work and $O(\log^2 n)$ span.

- We can’t really sort the sequence!
FINDING THE $k^{th}$ SMALLEST ELEMENT

1 fun $kthSmallest(k, S)$ = let
2    val $p$ = a value from $S$ picked uniformly at random
3    val $L = \{x \in S \mid x < p\}$
4    val $R = \{x \in S \mid x > p\}$
5    in if $(k < |L|)$ then $kthSmallest(k, L)$
6       else if $(k < |S| - |R|)$ then $p$
7       else $kthSmallest(k - (|S| - |R|), R)$

Let $X_n = \max\{|L|, |R|\}$

$$W(n) = W(X_n) + O(n)$$
$$S(n) = S(X_n) + O(\log n)$$
FINDING THE $k^{th}$ SMALLEST ELEMENT

We want to find $E[X_n]$?

$$E[X_n] = \sum_{i=1}^{n-1} \max\{i, n-i\} \cdot \frac{1}{n} \leq \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \leq \frac{3n}{4}$$
FINDING THE $k^{th}$ SMALLEST ELEMENT

- $E[X_n] \leq \frac{3n}{4} \Rightarrow$ geometrically decreasing sum 
  $\Rightarrow O(n)$ work.
- What is $Pr[X_n \leq \frac{3}{4}n]$?
- Since $|R| < n - |L|$, 
  $$X_n \leq \frac{3}{4}n \Leftrightarrow \frac{n}{4} < |L| \leq \frac{3n}{4}$$
  and the probability is 
  $$\frac{3n/4 - n/4}{n} = \frac{n/2}{n} = \frac{1}{2}$$
FINDING THE $k^{th}$ SMALLEST ELEMENT

$$W(n) = \sum_i \Pr[X_n = i] \cdot W(i) + c \cdot n$$

Using stepwise approximation

$$\leq \Pr[X_n \leq \frac{3n}{4}] W(3n/4) + \Pr[X_n > \frac{3n}{4}] W(n) + c \cdot n$$

$$= \frac{1}{2} W(3n/4) + \frac{1}{2} W(n) + c \cdot n$$

$$\implies (1 - \frac{1}{2}) W(n) = \frac{1}{2} W(3n/4) + c \cdot n$$

$$\implies W(n) \leq W(3n/4) + 2c \cdot n$$

Root Dominated hence solves to $O(n)$.
FINDING THE $k^{th}$ SMALLEST ELEMENT

$$S(n) = S(X_n) + O(\log n)$$

$$S(n) \leq \sum_i \Pr[X_n = i] \cdot S(i) + c \log n$$

$$\leq \Pr[X_n \leq \frac{3n}{4}] S(3n/4) + \Pr[X_n > \frac{3n}{4}] S(n) + c \cdot \log n$$

$$\leq \frac{1}{2} S(3n/4) + \frac{1}{2} S(n) + c \cdot \log n$$

$$\implies (1 - \frac{1}{2}) S(n) \leq \frac{1}{2} S(3n/4) + c \log n$$

$$\implies S(n) \leq S(3n/4) + 2c \log n$$

This solves to $O(\log^2 n)$. 
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 16
Graph Contraction
SYNOPSIS

- Graph Contraction
- Finding Connected Components
- Edge Contraction
- Star Contraction
Most graph search algorithms were either
- sequential, or
- had span dependent on the diameter.

Can we make these algorithms more parallel?
- Polylogarithmic span: span is bounded by a polynomial in \( \log n \)

We will look at contraction as a way to build parallel algorithms for some graph problems:
- Graph Connectivity
- Spanning Trees
**Graph Connectivity**

- Two vertices in an undirected graph are connected if there is a path between them.
- A graph is connected if all pairs of vertices are connected.
- The graph connectivity problem partitions a graph into its maximal connected subgraphs.

The graph has two connected subgraphs: \{a, b, c, d\} and \{e, f\}.
Graph Connectivity

- BFS or DFS
  - Identify vertices of a connected component
  - Identify all connected components!
- BFS could be parallel but has span $\propto$ diameter $d$
- Each connected component needs to be done sequentially!
Graph Contraction

- Problem → Smaller Problem
- Shrink the size of the graph and solve the connectivity problem on the small graph.
  - Different components can be handled in parallel!
- Applicable to other problems
  - Spanning Trees
  - Minimum Spanning Trees
**Graph Contraction**

\[ \text{contract} : \text{graph} \rightarrow \text{partition} \]

- Takes a graph \( G(V, E) \) and returns a partitioning of \( V \) into connected subgraphs.
  - Not necessarily maximally connected subgraphs (yet)
  - But vertices in a partition are connected.

\[
\{\{a, b, c\}, \{d\}, \{e, f\}\}
\]
If the graph contracts on each round, eventually each maximal connected component will shrink down to a single vertex!
**Representing Partitions**

\[
\begin{align*}
\{\{a, b, c\}, \{d\}, \{e, f\}\} & \\
\downarrow & \\
(\{a, d, e\}, \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\})
\end{align*}
\]
Counting Components

1 fun numComponents((V, E), i) =
2 if |E| = 0 then |V|
3 else let
4 val (V', P) = contract((V, E), i)
5 val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6 in
7 numComponents((V', E'), i + 1)
8 end

- Ignore i for the time being!
- V' is the set of representative vertices
- P maps every v ∈ V to a v' ∈ V'.
- E' is the set of edges in the contracted graph.
  - Self-loops are removed!
fun components((V, E), i) = 
if |E| = 0 then {v ↦→ v : v ∈ V}
else let
val (V', P) = contract((V, E), i)
val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
val P' = components((V', E'), i + 1)
in {v ↦→ P'[P[v]] : v ∈ V}
end
**Computing Components**

```
fun components((V, E), i) =
  if |E| = 0 then {v ↦ v : v ∈ V}
  else let
    val (V', P) = contract((V, E), i)
    val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
    val P' = components((V', E'), i + 1)
  in
    {v ↦ P'[P[v]] : v ∈ V}
  end

Base case: Every vertex maps to itself!
```
Computing Components

fun components((V, E), i) = 
   if |E| = 0 then {v ↦→ v : v ∈ V}
   else let
      val (V′, P) = contract((V, E), i)
      val E′ = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
      val P′ = components((V′, E′), i + 1)
   in
   {v ↦→ P′[P[v]] : v ∈ V}
end

(Recursively) find components of the contracted graph
fun components((V, E), i) =  
if |E| = 0 then {v ↦→ v : v ∈ V}  
else let  
val (V′, P) = contract((V, E), i)  
val E′ = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}  
val P′ = components((V′, E′), i + 1)  
in  
{v ↦→ P′[P[v]] : v ∈ V}  
end

Map each vertex to the representative vertex of its partition!
fun components((V, E), i) = 
  if |E| = 0 then {v ↦→ v : v ∈ V}
  else let
  val (V', P) = contract((V, E), i)
  val E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
  val P' = components((V', E'), i + 1)
  in
  {v ↦→ P'[P[v]] : v ∈ V}
  end

After 4: V' = {a, d, e}          
P = {a ↦→ a, b ↦→ a, c ↦→ a, d ↦→ d, e ↦→ e, f ↦→ e}

After 6: P' = {a ↦→ a, d ↦→ a, e ↦→ a}

8 returns: {a ↦→ a, b ↦→ a, c ↦→ a, d ↦→ a, e ↦→ a, f ↦→ a}
IMPLEMENTING CONTRACT

- **Edge Contraction**: Only pairs of vertices connected by an edge are contracted.
- **Star Contraction**: Vertices around a “center star” collapse to the “star”
- **Tree Contraction**: Disjoint trees within the graph are identified and vertices in a tree are collapsed to the root.
- **Parallel**
- **Reduce graph size** (vertices/edges?) by a constant factor every round.
  - Will lead to $O(\log n)$ rounds!
Edge Contraction

- Find disjoint edges – edges can not share vertices.

- Vertex matching problem
- Can be done in parallel
  - Each edge picks a random priority in $[0, 1]$
  - Any edge which has highest priority for both vertices gets selected.
- It turns out this has some problems!
**Edge Contraction**

- Consider a graph like

  ![Graph Diagram]

  - How many edges can be contracted each round?
  - How many rounds are needed to contract to 1 node?
  - Not very parallel!
Star Contraction

- Star subgraphs can be contracted in parallel!
- How do we find disjoint stars?
Each vertex throws a coin
  ▶ Heads → vertex is a star-center
  ▶ Tails → vertex is a potential satellite (Why potential?)

Each satellite then selects a center.
**Random Coin Tosses**

- Pretend each vertex has a potentially infinite sequence of random coin flips
- \( \text{heads}(v, i) : \text{vertex} \times \text{int} \rightarrow \text{bool} \) provides access to these coin tosses.
- This can be implemented with a pseudorandom number generator.
Star Contraction

fun starContract(G = (V, E), i) =
let

  % select edges that go from a tail to a head
  val TH = \{(u, v) ∈ E | ¬heads(u, i) ∧ heads(v, i)\}

  % make mapping from tails to heads, removing duplicates
  val P = \bigcup_{(u, v) ∈ TH} \{u ↦ v\}

  % remove vertices that have been remapped
  val V’ = V \ domain(P)

  % Map remaining vertices to themselves
  val P’ = \{u ↦ u : u ∈ V’\} ∪ P

in (V’, P’) end

---

Graph Contraction

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
**Star Contraction**

- Coin flips (heads(v, i))
- Find potential centers (TH)
- Compute "hook" edges (P)

\[
\text{val } TH = \{(u, v) \in E \mid \neg \text{heads}(u, i) \land \text{heads}(v, i)\}
\]

\[TH = \{(c, a), (c, b), (e, b)\}.
\]
**Star Contraction**

- Coin flips (heads(v,i))
- Find potential centers (TH)
- Compute "hook" edges (P)

\[
\text{val} \ P = \bigcup_{(u,v) \in \text{TH}} \{u \mapsto v\}
\]

- \(\text{TH} = \{(c, a), (c, b), (e, b)\}\)
- \(P = \{c \mapsto b, e \mapsto b\}\)
**Star Contraction**

- **Coin Flips (Heads(v,i))**
- **Find Potential Centers (TH)**
- **Compute "Hook" Edges (P)**

\[
\text{val } V' = V \setminus \text{domain}(P)
\]

- \( P = \{c \mapsto b, e \mapsto b\} \)
- \( \text{domain}(P) = \{c, e\} \)
- \( V' = \{a, b, d\} \)
Star Contraction

```plaintext
val \( P' = \{ u \mapsto u : u \in V' \} \cup P \)

- \( P = \{ c \mapsto b, e \mapsto b \}, \ V' = \{ a, b, d \} \)
- \( P' = \{ a \mapsto a, b \mapsto b, c \mapsto b, d \mapsto d, e \mapsto b \} \)
```
**Analysis of Star Contraction**

**Lemma**

For a graph $G$ with $n$ non-isolated vertices, let $X_n$ be the random variable indicating the number of vertices removed by $\text{starContract}(G, \_).$ Then, $E[X_n] \geq n/4.$

- $H_v$: vertex $v$ comes up heads, $T_v$: vertex $v$ comes up tails
- $R_v$: vertex $v$ is removed in contraction
- $v$ has at least one neighbor $u.$
- $T_v \land H_u$ implies $R_v$
  - If $v$ is a tail, join $u$’s star or some other star.
- $\Pr[R_v] \geq \Pr[T_v]\Pr[H_u] = 1/4$
- Expected total $\geq n/4$
fun starContract(G = (V, E), i) =

let

% select edges that go from a tail to a head – O(m) work, O(1) span
val TH = \{(u, v) ∈ E | ¬ heads(u, i) ∧ heads(v, i)\}
% make mapping from tails to heads, removing duplicates
val P = \{(u, v) ∈ TH | u → v\}
% remove vertices that have been remapped
% O(n) work, O(log n) span
val V' = V \ domain(P)
% Map remaining vertices to themselves - O(n) work, O(log n) span
val P' = \{u → u : u ∈ V'\} ∪ P

in (V', P') end

• n nodes, m edges
• O(n + m) work, O(log n) span.
Analysis of Connectivity

1. fun numComponents((V, E), i) =
2. if |E| = 0 then |V|
3. else let
4. val (V’, P) = starContract((V, E), i)
5. val E’ = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6. in
7. numComponents((V’, E’), i + 1)
8. end

• S(n) = S(n’) + O(log n)
• n’ = n – Xn and E [Xn] = n/4, so E [n’] = 3n/4
• S(n) ∈ O(log² n)
ANALYSIS OF CONNECTIVITY

- We can remove a constant fraction of vertices every round.
- For each vertex removed, we remove at least one edge.
- Consider a hypothetical contraction

<table>
<thead>
<tr>
<th>round</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>n/2</td>
<td>m − n/2</td>
</tr>
<tr>
<td>3</td>
<td>n/4</td>
<td>m − 3n/4</td>
</tr>
<tr>
<td>4</td>
<td>n/8</td>
<td>m − 7n/8</td>
</tr>
</tbody>
</table>

- Number of edges does not go below \( m − n \).
Analysis of Connectivity

\[ W(n, m) \leq W(n', m) + O(n + m), \]

As before, \( E[n'] = 3n/4 \), so

\[ E[W(n, m)] \in O(n + m \log n) \]
**Tree Contraction**

- Identify disjoint trees and contract them.
- For every tree of $t$ vertices contracted, $t - 1$ edges are removed.
- Number of edges also go down geometrically at every round.
- Leads to $O(m)$ work and $O(\log^2 n)$ span.
There is no lecture 17
15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 18
MINIMUM SPANNING TREES
SYNOPSIS

- Minimum Spanning Trees
- Kruskal’s and Prim’s Algorithms
- Using Star Contraction for MST
Minimum Spanning Trees
Minimum Spanning Trees
Minimum Spanning Trees

- Given a connected undirected graph $G = (V, E)$
  - Each edge $e$ has $w_e \geq 0$
- Find a spanning tree, $T$ that minimizes
  \[ w(T) = \sum_{e \in E(T)} w_e. \]

- Sequential algorithms:
  - Kruskal’s Algorithm
  - Prim’s Algorithm
Minimum Spanning Trees

A - B: 6
B - C: 8
C - D: 7
C - G: 12
C - F: 5
F - E: 14
F - H: 3
H - I: 11
I - D: 9
H - G: 19
Light Edge Rule

- Given \( G = (V, E) \), \( U \subseteq V \) partitions the graph into two parts with vertices \( U \) and \( V \setminus U \).
- The edges between \( U \) and \( V \setminus U \) are called the cut edges \( E(U, \overline{U}) \).
**Light Edge Rule**

**Theorem**

Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights.

- For any nonempty $U \subset V$
- the minimum weight edge $e$ between $U$ and $V \setminus U$ is in the minimum spanning tree $\text{MST}(G)$ of $G$. 
**Light Edge Rule**

- Assume \( e = (u, v) \) is the minimum edge in the cut but not in the MST.
- MST should have at least another edge in the cut.
- Adding \( e \) to the path between \( u \) and \( v \) creates a cycle.
- Removing the max edge from path (blue line) and adding \( e \) should give a ST with less weight.
- Original (claimed) MST (through blue line) cannot be a MST!
Kruskal’s Algorithm

- Greedy
- Each vertex is a subtree by itself initially
- Combine the two sub-trees on both sides of the next smallest edge (if they are different)
- Uses the union-find data structure.
- \(O(m \log n)\) work and span!
Kruskal’s Algorithm

Minimum Spanning Trees

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
Prim’s Algorithm

- Greedy
- Based on Priority-based Search – Variant of Dijsktra’s Algorithm
- Maintain visited \( X \) and frontier \( F \) vertices.
- Visit the nearest unvisited vertex in the frontier.
- \( O(m \log n) \) work and span!
**Prim’s Algorithm**

1. `fun prim(G) =`
2. `let`
3. `fun enqueue v (Q,(u,w)) = PQ.insert (w,(v,u)) Q`
4. `fun proper(X, Q, T) =`
5. `case PQ.deleteMin(Q) of`
6. `| (NONE, _) ⇒ T`
7. `| (SOME(d,(u,v)), Q') ⇒`
8. `if (v ∈? X) then proper(X, Q', T)`
9. `else let`
10. `val X' = X ∪ {v}`
11. `val T' = T ∪ {(u,v)}`
12. `val Q'' = iter (enqueue v) Q' N_G(v)`
13. `in proper(X', Q'', T') end`
14. `val s = an arbitrary vertex from G`
15. `val Q = iter (enqueue s) {} N_G(s)`
16. `in`
17. `proper({s}, Q, {})`
18. `end`
Prim’s Algorithm
**Observation**

- The minimum weight edge out of every vertex of a weighted graph \( G \) belongs to its MST.

- Why should this be the case?

- MST can contain other edges!
Parallel MST - Idea #1

- Throw all minimum weight edges into MST
- **Tree contract** the vertices for all these edges
- Repeat until no edges remain!

Each rounds removes at least 1/2 of the vertices (Why?)
**Parallel MST - Idea #2**

- Let $minE$ be the set of minimum weight edges.
- Let $H = (V, minE)$ be a subgraph of $G$
- We apply (modified) star contraction to $H$
  - The tails hook up through the minimum weight edge!

```ml
fun minStarContract(G = (V, E), i) =
  let
  val minE = minEdges(G)
  val P = {u ↦ (v, w) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
  val V' = V \ domain(P)
  in (V', P) end
```
**Parallel MST - Idea #2**

- Even though we are working with a subgraph, the star contract lemma still applies.

**Lemma**
For a graph $G$ with $n$ non-isolated vertices, let $X_n$ be the random variable indicating the number of vertices removed by $\text{minStarContract}(G, r)$. Then, $E(X_n) \geq n/4$.

- MST will take expected $O(\log n)$ rounds.
As the graph contracts, the end point of each edge changes!

At the end, the edges may not have the original end points.

Associate a unique label to each edge initially:

- $(\text{vertex} \times \text{vertex} \times \text{weight} \times \text{label})$
- The end points change but the label does not!
Modifed Star Contract

```
fun minStarContract(G = (V, E), i) =
    let
        val minE = minEdges(G)
        val P = {(u ↦→ (v, w, ℓ)) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
        val V' = V \ domain(P)
    in (V', P) end
```

- Line 3: Finds min edge for each vertex.
  - All these belong to the MST
- Line 4: Picks tails and heads, and the creates mapping from tails to heads.
- Line 5: Removes all tail vertices from the vertex set.
**The MST Algorithm**

```
1 fun MST((V, E), T, i) =
2     if |E| = 0 then T
3     else let
4         val (V′, PT) = minStarContract((V, E), i)
5         val P = \{u \mapsto v : u \mapsto (v, w, ℓ) \in PT\} \cup \{v \mapsto v : v \in V′\}
6         val T′ = \{ℓ : u \mapsto (v, w, ℓ) \in PT\}
7         val E′ = \{(P[u], P[v], w, ℓ) : (u, v, w, ℓ) \in E | P[u] \neq P[v]\}
8     in
9         MST((V′, E′), T \cup T′, i + 1)
10    end

Invoked by \(MST(G, \{\}, 1)\).
```
**IMPLEMENTING $\text{minEdges}(G)$**

fun joinEdges((v1, w1, l1), (v2, w2, l2)) =
  if (w1 ≤ w2) then (v1, w1, l1) else (v2, w2, l2)

fun minEdges(E) =
  let
    val ET = \{ u ↦ (v, w, l) : (u, v, w, l) ∈ E \}
  in
    (merge joinEdges) {} ET
  end
SYNOPSIS

- Quicksort
- Work and Span Analysis of Randomized Quicksort
- Lower Bound for Comparison-based Sorting
- Lower Bound for Merging
QUICKSORT

- Originally invented and analyzed by Hoare in 1960’s.
- I strongly urge to watch Jon Bentley on “Three beautiful Quicksorts” at
  - www.youtube.com/watch?v=QvgYAQzg1z8.
int i, j;
for( i = low, j = high - 1; ; )
{
    while( a[ ++i ] < pivot );
    while( pivot < a[ --j ] );
    if( i >= j )
        break;
    swap( a, i, j );
}
// Restore pivot
swap( a, i, high - 1 );
quicksort( a, low, i - 1 ); // Sort small elements
quicksort( a, i + 1, high ); // Sort large elements
fun quicksort(S) =
  if |S| = 0 then S
  else let
    val p = pick a pivot from S
    val S1 = ⟨s ∈ S | s < p⟩
    val S2 = ⟨s ∈ S | s = p⟩
    val S3 = ⟨s ∈ S | s > p⟩
    val (R1, R3) = (quicksort(S1) || quicksort(S3))
  in
    append(R1, append(S2, R3))
  end
QuickSort

- Each call to QuickSort either makes
  - No recursive calls (base case), or
  - Two recursive calls
- Call tree is a binary
- Depth the call tree determines the span of the algorithm.
Picking the Pivot

- Always pick the first element
  - Worst case $O(n^2)$ work.
  - In practice, almost sorted inputs are not uncommon.
- Pick the median of 3 elements (e.g., first, middle and last elements)
  - could possibly divide evenly
  - worst case is still bad
- Pick an element at random
  - we hope this divides evenly in expectation
  - leading to expected $O(n \log n)$ work and $O(\log^2 n)$ span.
PICKING THE PIVOT

- Pick first element
  - Worst case $O(n^2)$ work.
  - Expected $O(n \log n)$ work
    - Averaged over all possible orderings.
  - Work well on the average
  - Slow on some, possibly common, cases.

- Pick a random element
  - Expected worst-case $O(n \log n)$ work.
    - For input in any order, the expected work is $O(n \log n)$
  - No input has expected $O(n^2)$ work.
  - With a small probability, we could be unlucky and have $O(n^2)$ work.
**Randomized Quicksort**

- Assign a uniformly random priority to each number in $[0, 1]$.

```plaintext
fun quicksort(S) =
  if |S| = 0 then S
  else let
    val p = pick as pivot the highest priority element from S
    val S_1 = \{ s \in S | s < p \}
    val S_2 = \{ s \in S | s = p \}
    val S_3 = \{ s \in S | s > p \}
    val (R_1, R_3) = (quicksort(S_1) ∥ quicksort(S_3))
  in
    append(R_1, append(S_2, R_3))
  end
```

- Once the priorities are assigned, the algorithm is deterministic.
RANDOMIZED QUICKSORT

- Count comparisons made!
  - Almost all the work is comparisons.
  
  \[ X_n = \# \text{ of comparisons } \text{quicksort} \]
  makes on input of size \( n \)

- Find \( \mathbb{E} [X_n] \) for any input sequence \( S \)

- Notation:
  - Let \( T = \text{sort}(S) \)
  - \( T_i \) and \( T_j \) refer to elements in the final sorted order and \( i < j \) and \( T_i \leq T_j \).
  - \( p_i \) refers to priority chosen for \( T_i \).
  - \( A_{i,j} = 1 \) if \( T_i \) and \( T_j \) were ever compared during the sort.
Analyzing Quicksort

Crucial point is how to model $A_{i,j}$.

In any one call to quicksort, there are three cases:

- Pivot $p$ is either $T_i$ or $T_j \Rightarrow A_{i,j} = 1$
- $T_i < p < T_j \Rightarrow T_i \in S_1$, $T_j \in S_3$, $A_{i,j} = 0$
- Either $p < T_i$ or $p > T_j \Rightarrow T_i, T_j \in S_1$ or $T_i, T_j \in S_3$

If two elements are compared in a quicksort call, they will never be compared again in any other call!
Analyzing Quicksort

\[ X_n \leq 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} A_{ij} \]

- The non-optimized code compares each element to pivot 3 times.

  1 \[ \ldots \]
  2 \[ \text{val } S_1 = \langle s \in S | s < p \rangle \]
  3 \[ \text{val } S_2 = \langle s \in S | s = p \rangle \]
  4 \[ \text{val } S_3 = \langle s \in S | s > p \rangle \]
  5 \[ \ldots \]

- By linearity of expectation

\[ E[X_n] \leq 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[A_{ij}] = 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[A_{ij} = 1] \]
**Analyzing Quicksort**

- Consider first when the pivot is one of $T_i, T_{i+1}, ..., T_j$
- $T_i$ and $T_j$ are compared $\iff p_i$ or $p_j$ is the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.
  - Assume $T_k, i < k < j$ has higher priority.
  - For any subdivision $\cdots, T_i, \cdots, T_k, \cdots, T_j$ will become a pivot and separate $T_i$ and $T_j$
  - $T_i$ and $T_j$ will never be compared!
ANALYZING QUICKSORT

\[ E[A_{ij}] = \Pr[A_{ij} = 1] \]
\[ = \Pr[p_i \text{ or } p_j \text{ is the maximum in } \{p_i, \ldots, p_j\}] \]
\[ = \frac{2}{j - i + 1} \quad \text{(Why ?)} \]

- \(j - i + 1\) elements between \(p_i\) and \(p_j\) and each is equally likely to be the maximum.
- We want either \(p_i\) or \(p_j\), hence \(\frac{2}{j - i + 1}\).
- \(T_i\) is compared to \(T_{i+1}\) with probability 1.
ANALYZING QUICKSORT

\[ E[X_n] \leq 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[A_{ij}] \]

\[ = 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \]

\[ = 3 \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \quad \text{(change variables)} \]

\[ \leq 6 \sum_{i=1}^{n} H_n \]

\[ \leq 6 \cdot n \cdot H_n \in O(n \log n) \]
Analyzing Quicksort

- Indirectly, average work for basic deterministic quicksort is $O(n \log n)$.
  - Just shuffle data randomly and apply the basic algorithm
  - $\equiv$ to picking random priorities
ALTERNATIVE ANALYSIS

Write a recurrence for the number of comparisons:

\[ X(n) = X(Y_n) + X(n - Y_n - 1) + n - 1 \]

Random variable \( Y_n \) is the size of \( S_1 \).

\[
\mathbb{E}[X(n)] = \mathbb{E}[X(Y_n) + X(n - Y_n - 1) + n - 1] \\
= \mathbb{E}[X(Y_n)] + \mathbb{E}[X(n - Y_n - 1)] + n - 1 \\
= \frac{1}{n} \sum_{i=0}^{n-1} (\mathbb{E}[X(i)] + \mathbb{E}[X(n - i - 1)]) + n - 1
\]
**Alternative Analysis**

\[
E[X(n)] = \frac{1}{n} \sum_{i=0}^{n-1} (E[X(i)] + E[X(n-i-1)]) + n-1 \\
= \frac{2}{n} \sum_{i=0}^{n-1} E[X(i)] + n-1
\]

- With telescoping, this also solves as \(O(n \log n)\)
**Expected Span**

- $S$ is split into $L(ess)$, $E(qual)$ and $(g)R(eater)$.
- Let $X_n = \max\{|L|, |R|\}$.
- We use *filter* to partition.

$$S(n) = S(X_n) + O(\log n)$$
**EXPECTED SPAN**

- Let $\bar{S}(n)$ denote $E[S(n)]$
- We bound $\bar{S}(n)$ by considering $Pr[X_n \leq 3n/4]$ and $Pr[X_n > 3n/4]$.
- $Pr[X_n \leq 3n/4] = 1/2$
  - As with SmallestK, 1/2 of the randomly chosen pivots results in larger partition of at most size $3n/4$ elements.
**EXPECTED SPAN**

\[
S(n) = \sum_i \Pr[X_n = i] \cdot S(i) + c \log n
\]

\[
\leq \Pr[X_n \leq \frac{3n}{4}]S(\frac{3n}{4}) + \Pr[X_n > \frac{3n}{4}]S(n) + c \cdot \log n
\]

\[
\leq \frac{1}{2}S(\frac{3n}{4}) + \frac{1}{2}S(n) + c \cdot \log n
\]

\[
\implies (1 - \frac{1}{2})S(n) \leq \frac{1}{2}S(\frac{3n}{4}) + c \log n
\]

\[
\implies S(n) \leq S(\frac{3n}{4}) + 2c \log n
\]

\[
\implies S(n) \in O(\log^2 n)
\]
Lower Bound for Sorting

- What is asymptotically the minimum number comparisons any sorting algorithm has to make?
- Lower-bounds apply to problems not to algorithms.
  - Algorithms provide upper bounds!
- We say sorting is $\Omega(n \log n)$
- No (comparison-based) sorting algorithm has work asymptotically lower than $n \log n$. 
**Decision Trees**

- If there are $N$ outcomes, the number of questions is at least $\log_2 N$. 

---

**QuickSort Analysis and Sorting Lower Bounds**

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Fall 2013
For $n$ items, how many possible outcomes can there be?

- $n! \Rightarrow$ we need at least $\log_2(n!)$ “questions”. 
Sorting as a Decision Problem

\[ \log(n!) = \log n + \log(n - 1) + \cdots + \log(n/2) + \cdots + \log 1 \]

\[ \geq \log n + \log(n - 1) + \cdots + \log(n/2) \]

\[ \geq \frac{n}{2} \cdot \log(n/2) \in \Omega(n \log n) \]
LOWER BOUND FOR MERGING

- We have sorted sequences $A$, $|A| = n$ and $B$, $|B| = m$ and $m \leq n$.
  - Assume all elements are unique.
- All interleavings are possible
- We need to choose $m$ positions out of $n + m$ to place the elements of $B$ amongst elements of $A$.
- This can be done in $\log_2 \binom{n+m}{m}$ ways.
LOWER BOUND FOR MERGING

\[ \binom{n}{r} \geq \left(\frac{n}{r}\right)^r \]

\[ \text{See Lemma in the notes.} \]

\[ \log_2 \left( \binom{n + m}{m} \right) \geq \log_2 \left( \frac{n + m}{m} \right)^m = m \log_2 \left( 1 + \frac{n}{m} \right) \]
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 20
Search Trees I: BSTs Split, Join, and Union
SYNOPSIS

- Binary Search Trees
- Basic Structural Operations on BSTs
- Basic Operations on BSTs
- Concrete Implementations
- Cost Analysis
**Binary Trees**

- Trees where each node has at most 2 children each of which is a binary tree.
  - Left child / Left subtree
  - Right child / Right subtree
**Binary Search Trees**

- Binary trees with the “search” property
- For each node \(v\) with key \(k\)
  - The key of the left child \(k_L < k\)
  - The key of the right child \(k_R > k\)
**Balanced Trees**

- We try to keep binary search trees balanced.
  - Both children are about the same height
  - Both subtrees are about the same size
- **AVL Trees**
  - Left and right subtree heights differ by at most 1.
  - $O(\log n)$ root height maintained after each insertion and deletion.
- **Splay Trees**
  - Balanced in the amortized sense
  - A sequence of $n$ find, insert, or delete operations take $O(n \log n)$ work.
  - So average is $O(\log n)$ work.
Basic BST Operations

- Data type is defined by **structural induction**
  - Leaf
  - Node with a left child, a right child, a key, optional additional data.

```plaintext
datatype BST = Leaf |
              Node of (BST * BST * key * data)
```

Search Trees I: BSTs Split, Join, and Union
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
**Basic BST Operations**

- **split**\((T, k) : \text{BST} \times \text{key} \rightarrow \text{BST} \times (\text{data option}) \times \text{BST}\)

  - **split** divides \(T\) into two BSTs,
    - one consisting of all the keys from \(T\) less than \(k\)
    - the other all the keys greater than \(k\)
  - If \(k\) appears in the tree with associated data \(d\) then **split** returns **SOME**\((d)\)
  - Otherwise it returns **NONE**.
Basic BST Operations

- $\text{join}(L, m, R) : \text{BST} \times (\text{key} \times \text{data}) \text{ option} \times \text{BST} \rightarrow \text{BST}$

- Takes a left subtree ($L$) an optional key-data pair $m$ and a right subtree ($R$)
  - Assumes all keys in $L$ are less than all keys in $R$.
  - If present, the optional key is also larger than all keys in $L$ and smaller than all keys in $R$.

- Creates a new BST that is the union of $L$ and $R$ and $m$.
- We also assume both split and join maintain balance.
Basic BST Operations

- **expose\( (T) : BST \rightarrow (BST \times BST \times key \times data) \text{ option} \)**

  Returns the components if BST \( T \) is not empty.
Basic BST Operations - Search

fun search T k =
  let val (_, v, _) = split(T, k)
  in v
end
Basic BST Operations - Insert

1 fun insert \( T \) \((k, v)\) =
2 let val \((L, v', R)\) = split\(T, k\)
3 in join\(L, \text{SOME}(k, v), R\) 
4 end
Basic BST Operations - Delete

1 fun delete T k =
2 let val (L, , R) = split(T, k)
3 in join(L, NONE, R)
4 end
Concrete Implementations: Split

datatype BST = Leaf |
    Node of (BST * BST * key * data)

fun split(T, k) =
    case T of
        Leaf  ⇒ (Leaf, NONE, Leaf)
    | Node(L, R, k', v) ⇒
        case compare(k, k') of
            LESS ⇒
                let val (L', r, R') = split(L, k)
                in (L', r, Node(R', R, k', v)) end
            EQUAL ⇒ (L, SOME(v), R)
            GREATER ⇒
                let val (L', r, R') = split(R, k)
                in (Node(L, L', k', v), r, R') end
**Concrete Implementations: Join**

```haskell
fun join(T1, m, T2) =
  case m of
    SOME(k, v) ⇒ Node(T1, T2, k, v)
    | NONE ⇒
    case T1 of
      Leaf ⇒ T2
      | Node(L, R, k, v) ⇒ Node(L, join(R, NONE, T2), k, v))
```
For $T_1$ with key $k_1$ and children $L_1$ and $R_1$ at the root, use $k_1$ to split $T_2$ into $L_2$ and $R_2$.

Recursively find $L_u = \text{union}(L_1, L_2)$ and $R_u = \text{union}(R_1, R_2)$.

Now $\text{join}(L_u, k_1, R_u)$. 
Concrete Implementations: Union

```haskell
fun union(T1, T2) =
  case expose(T1) of
    NONE ⇒ T2
  | SOME(L1, R1, k1, v1) ⇒
    let val (L2, v2, R2) = split(T2, k1)
    val (L, R) = union(L1, L2) || union(R1, R2)
    in join(L, SOME(k1, v1), R)
  end
```

- Returns the value from $T_1$ if a key appears in both trees.
fun union(T1, T2) =
case expose(T1) of
  NONE ⇒ T2
| SOME(L1, R1, k1, v1) ⇒
    let val (L2, v2, R2) = split(T2, k1)
    val (L, R) = union(L1, L2) || union(R1, R2)
in join(L, SOME(k1, v1), R)
end

- split costs $O(\log |T_2|)$.
- Two recursive calls to union
- join costs $O(\log(|T_1| + |T_2|))$
ANALYSIS OF UNION - ASSUMPTIONS

• $T_1$ is perfectly balanced.
  ▶ expose return subtrees of size $|T_1|/2$
  ▶ Each a key from $T_1$ splits $T_2$, it splits exactly in half.
Analysis of Union

\[ W(|T_1|, |T_2|) = 2W\left(\frac{|T_1|}{2}, \frac{|T_2|}{2}\right) + O(\log(|T_1| + |T_2|)), \]

and

\[ W(1, |T_2|) = O(\log(1 + |T_2|)). \]

- When \(|T_1| = 1\), expose give us two empty subtrees \(L_1\) and \(R_1\).
- \(\text{union}(L_1, L_2)\) returns \(L_2\), \(\text{union}(R_1, R_2)\) returns \(R_2\) immediately!
- Joining these costs at most \(O(\log(|T_1| + |T_2|)) = O(\log(1 + |T_2|))\).
**ANALYSIS OF UNION**

- Let $m = |T_1|$ and $n = |T_2|$

  ![Diagram showing a binary tree with cost calculations for each level]

  - **Bottom level:** Each box costs $\log (n/m)$
  - **Leaf dominated (Why?)**
Analysis of Union

- How many leaves are there in this recursion tree?
  - $T_2$ has no impact.
  - We get $m = |T_1|$ leaves.
- How deep is the tree?
  - $1 + \log_2 m$
- What is the size of $T_2$ at the leaves?
  - $n/2^{\log_2 m} = \frac{n}{m}$
- Total cost at the leaves = $O(m \log(1 + \frac{n}{m}))$
- Union cost = $O(m \log(1 + \frac{n}{m}))$
SYNOPSIS

- Overview of Binary Search Trees
- Relationship between Quicksort and BSTs
- Treaps
- Expected Depth of a Treap
There are many options for keeping trees balanced.

- **Split** and **join** are the main structural operations to implement **find**, **insert**, **delete**, **union**, etc.

- **Cost of split and join** are logarithmic in the size of the input and output trees.

- Union needs $O(m \log(1 + \frac{n}{m}))$ work ($m \leq n$).
Quicksort and BSTs

- Write out the recursion tree for quicksort.
  - Assume distinct keys.
- Annotate each node with the pivot picked at that stage.
- You get a BST.

![Recursion Tree for Quicksort]

**SearchTree II: Treaps**
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Fall 2013
SEQUENCE TO BST

fun qs_tree(S) =
  if |S| = 0 then LEAF
  else let
      val p = pick a pivot from S
      val S1 = {s ∈ S | s < p}
      val S3 = {s ∈ S | s > p}
      val (T_L, T_R) = (qs_tree(S1) || qs_tree(S3))
  in
      NODE(T_L, p, T_R)
  end

Unlike Quicksort, we do not know what elements will be in the tree, when we start.
  ▶ We can not select a (n) (future?) element to be the root.
**Treaps**

- Treap = TRee + hEAP
- A treap is a randomized BST that maintains balance in a probabilistic way.
- Each element/key gets a unique random priority
- The nodes in the treap satisfy BST property.
  - Keys are stored in-order in the tree.
- The associated priorities satify the (max) heap property.
**The Max-Heap Property**

- Priority at each node is greater than the priorities of the children.
- Suppose we have
  \[ S = (a, 3), (b, 9), (c, 2), (e, 6), (f, 5) \]
LET’S DO AN EXAMPLE

- Draw the treap for the following \((key, priority)\) sequence.

\[(G,50), (C,35), (E,33), (H,29), (I,25), (B,24), (A,21), (L,16), (J,13), (K,9), (D,8)\]
THEOREM

For any set $S$ of unique key-priority pairs, there is exactly one treap $T$ containing the key-priority pairs in $S$ which satisfies the treap properties.

- Key $k$ with highest priority must be at the root.
- All keys $< k$ must be in the left subtree
- All keys $> k$ must be in the right subtree
- Subtrees of $k$ are constructed inductively in the same manner.
Basic BST Operations - Search

```plaintext
fun search T k =
  let val (_, v, _) = split(T, k)
  in v
  end
```
Basic BST Operations - Insert

1 fun insert T (k, v) =
2  let val (L, v', R) = split(T, k)
3  in join(L, SOME(k, v), R)
4  end
Basic BST Operations - Delete

fun delete T k = 
  let val (L, _, R) = split(T, k)
  in join(L, NONE, R)
  end

So if split and join are implemented the other more useful operations are covered.
**JOIN AND SPLIT**

- **split**\((T, k)\): BST × key → BST × (data option) × BST

- **split** divides \(T\) into two BSTs,
  - one consisting of all the keys from \(T\) less than \(k\)
  - the other all the keys greater than \(k\)

- If \(k\) appears in the tree with associated data \(d\) then **split** returns SOME\((d)\)

- Otherwise it returns NONE.
JOIN AND SPLIT

\[ \text{join}(L, m, R) : \text{BST} \times (\text{key} \times \text{data}) \text{ option} \times \text{BST} \rightarrow \text{BST} \]

- Takes a left subtree (\(L\)) an optional \textbf{key-data pair} \(m\) and a right subtree (\(R\))
  - Assumes all keys in \(L\) are less than all keys in \(R\).
  - If present, the optional key is also larger than all keys in \(L\) and smaller than all keys in \(R\).
- Creates a new BST that is the union of \(L\) and \(R\) and \(m\).
**Split on Treaps**

- Split code does not have to change.
- Priority orders do not change.
- Split does not put a larger priority *below* a smaller priority.
**Split on Treaps**

```ML
datatype BST = Leaf | Node of (BST * BST * key * data)

fun split(T, k) =
  case T of
  Leaf ⇒ (Leaf, NONE, Leaf)
  | Node(L, R, k', v) ⇒
    case compare(k, k') of
    LESS ⇒
      let val (L', r, R') = split(L, k)
      in (L', r, Node(R', R, k', v)) end
    EQUAL ⇒ (L, SOME(v), R)
    GREATER ⇒
      let val (L', r, R') = split(R, k)
      in (Node(L, L', k', v), r, R') end
```
JOIN ON TREAPS

- Join needs to change!
  - The priorities of the roots of two trees need to be compared.
  - The root with the larger priority becomes the new root.
- Basic join took the root of the first tree or the new node as the root.

```haskell
fun join(T1, m, T2) =
case m of
  SOME(k, v) ⇒ Node(T1, T2, k, v)
  | NONE ⇒
  case T1 of
    Leaf ⇒ T2
    | Node(L, R, k, v) ⇒ Node(L, join(R, NONE, T2), k, v)
```

SEARCH TREES II: TREAPS 17/26
CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS FALL 2013
JOIN ON TREAPS

fun join(T1, m, T2) =
   let
      fun singleton(k, v) = Node(Leaf, Leaf, k, v)
      fun join'(T1, T2) =
         case (T1, T2) of
            (Leaf, _) ⇒ T2
            | (_, Leaf) ⇒ T1
            | (Node(L1, R1, k1, v1), Node(L2, R2, k2, v2)) ⇒
               if (priority(k1) > priority(k2)) then
                  Node(L1, join'(R1, T2), k1, v1)
               else
                  Node(join'(T1, L2), R2, k2, v2)
            end
         in
         case m of
            NONE ⇒ join'(T1, T2)
            SOME(k, v) ⇒ join'(T1, join'(singleton(k, v), T2))
         end
Cost of split and join depend on the expected depth of a key.

Given a set of keys \( K \) and priorities \( p : key \rightarrow \text{int} \):

- Priorities are unique!

Consider the elements of the tree laid out in order:

- \( key_i < key_j \Rightarrow \cdots, key_i, \cdots, key_j, \cdots \)
- \( key_j < key_i \Rightarrow \cdots, key_j, \cdots, key_i, \cdots \)

\( A^j_i \) is an indicator variable:

- \( A^j_i = 1 \) if \( key_j \) is an ancestor of \( key_i \) in the treap.
- \( A^j_i = 0 \) otherwise.
**Expected Depth of a Key**

\[ \cdots, key_i, \cdots, key_j, \cdots \]

\[ key_i < key_j \]

\[ p_i = \max(p_i, \ldots, p_j) \quad p_k = \max(p_i, \ldots, p_j) \quad p_j = \max(p_i, \ldots, p_j) \quad i < k < j \]

\[ A_{i,j} = 0 \quad A_{i,j} = 0 \quad A_{i,j} = 1 \]
**Expected Depth of a Key**

\[
\cdots, \text{key}_j, \cdots, \text{key}_i, \cdots
\]

\[
\text{key}_i > \text{key}_j
\]

\[
p_i = \max(p_j, \ldots, p_i)
\]

\[
p_k = \max(p_j, \ldots, p_i) \quad \text{for } i < k < j
\]

\[
p_j = \max(p_j, \ldots, p_i)
\]

\[
A_{i,j} = 0
\]

\[
A_{i,j} = 0
\]

\[
A_{i,j} = 1
\]
**Expected Depth of a Key**

\[
E \left[ \text{depth of } i \text{ in } T \right] = E \left[ \sum_{j=1, j \neq i}^{n} A_i^j \right] = \sum_{j=1, j \neq i}^{n} E \left[ A_i^j \right].
\]

\[
E \left[ A_i^j \right] = \frac{1}{|j - i| + 1} \quad (\text{Why?})
\]

*Search Trees II: Treaps*

*CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms*

*Fall 2013*
**Expected Depth of a Key**

\[
\mathbb{E}[\text{depth of } i \text{ in } T] = \sum_{j=1, j \neq i}^{n} \frac{1}{|j-i| + 1}
\]

(Split \mid \Rightarrow) \quad \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1}

(Change variables \Rightarrow) \quad \sum_{k=2}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k}

\begin{align*}
&= H_i - 1 + H_{n-i+1} - 1 \\
&< \ln i + \ln(n-i+1) \\
&= O(\log n)
\end{align*}

Relative (sorted) position of a key determines expected depth in treap.
COST OF SPLIT AND JOIN

**Theorem**

For treaps

- $\text{join}(T_1, m, T_2)$ returning $T$
- $\text{split}(T, (k, v))$

have $O(\log |T|)$ expected work and span.

- See notes for short proofs.
Expected Max Depth of a Treap

- Expected depth of treap node is $O(\log n)$
  - Find takes on the average $O(\log n)$ work and span.
- What is the expected maximum depth of a treap?
  - Why is this important?
  - Expected worst-case cost!
- But $\mathbb{E}[\max_i\{A_i\}] \neq \max_i\{\mathbb{E}[A_i]\}$!
- It turns out this is almost the same problem as the expected span of the quicksort.
\( Y_n \) is the size of the larger partition.

\[ D(n) = D(Y_n) + 1 \Rightarrow D(n) \in O(\log n) \]
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 23
More with Trees
SYNOPSIS

- Ordered Sets and Tables
- Bingle Revisited
- Augmenting Balanced Trees
- Ordered Tables with Reduced Values
- Application Examples
Ordered Sets and Tables

- So far, we did not worry about the ordering of the values/keys in sets and tables.
  - Find, union, intersect, merge, etc.
- For many applications, exploiting any order is very important!
  - Find all elements between 3 and 17.
  - Find all customers who bought more than 5 of one item.
  - Find all emails in the week of March 31st.
- Ordered sets and tables.
**Ordered Set ADT**

- We have a totally ordered universe $U$, and $S$ represents the set of all subsets of $U$.
- With the following operations

  all operations supported by the Set ADT, and

\[
\text{last}(S) : S \rightarrow U = \max S \\
\text{first}(S) : S \rightarrow U = \min S \\
\text{split}(S,k) : S \times U \rightarrow S \times \text{bool} \times S = (\{k' \in S \mid k' < k\}, k \in S, \{k' \in S \mid k' > k\}) \\
\text{join}(S_1,S_2) : S \times S \rightarrow S = S_1 \cup S_2, \text{assuming max } S_1 < \min S_2 \\
\text{getRange}(S,k_1,k_2) : S \times U \times U \rightarrow S = \{k \in S \mid k_1 \leq k \leq k_2\}
\]
Ordered Set ADT

- Underlying implementation uses trees.
- `first` and `last` are easy
  - `first` traverses down the left spine to the minimum value.
  - `last` traverses down the right spine to the maximum value.
- `getRange` involves two splits.
IMPROVISING BINGLE

signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index
  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
end

- docList is a set.
- index is a table.
We want to limit the search to certain domains (e.g., cmu.edu)
  ▶ or docs with a certain name.
We want to add
val inDomain : domain * docList -> docList
For example
inDomain("cs.cmu.edu",
  and(find idx "cool", find idx "TAs"))
IMPROVISING BINGLE

- Assume doc ids are URLs.
- Assume they are “reverse” lexicographically ordered.
  - The last character is the most important!

```
1  fun inDomain(domain, L) =
2      getRange(L, domain, string.prepend(domain, "$"))
```

- $ is a character that is greater than any character.
Augmenting Balanced Trees

- Sets (and underlying trees) hold the key and any associated values.
- We can add other additional values to help with other search operations.
  - Track key positions and certain subset sizes.
- \( \text{rank}(S, k) \): How many elements in \( S \) are less than \( k \)?
- \( \text{select}(S, i) \): Which element in \( S \) has rank \( i \)?
- \( \text{splitIdx}(S, i) \): Split \( S \) into two sets: first \( i \) keys and the remaining \( n - i \) keys.
Augmenting Balanced Trees

rank(S, k) : \( S \times U \to \text{int} \) = \(|\{k' \in S | k' < k\}|\)

select(S, i) : \( S \times \text{int} \to U \) = \( k \) such that \(|\{k' \in S | k' < k\}| = i\)

splitIdx(S, i) : \( S \times \text{int} \to S \times S \)

Without additional information stored with the keys, these operations would take \( \theta(|S|) \) work.
Let $S = \{1, 2, 3, 4, 5, 6\}$

- $\text{rank}(S, 4) = |\{1, 2, 3\}| = 3$
- $\text{select}(S, 3) = 4$ since $\text{rank}(S, 4) = 3$
- $\text{splitIdx}(S, 3) = (\{1, 2, 3\}, \{4, 5, 6\})$
Augmenting Balanced Trees

- At each node keep the size of the subtree.
- This allows size and the three other operations in $O(d)$ work with $d$ as the depth of the tree.
- Size can be computed on the fly by adding 1 to the sum of the subtree sizes!
fun select(T, i) =
  case expose(T) of
    NONE ⇒ raise Range
  | SOME(L, R, k) ⇒
    case compare(i, |L|) of
      LESS ⇒ select(L, i)
    | EQUAL ⇒ k
    | GREATER ⇒ select(R, i − |L| − 1)
**Rank and splitIdx**

- **rank** is easy: just split and return the size of the left tree!
- **splitIdx** is just like split (or you navigate using sizes (as opposed to key values))
**Ordered Tables with Reduced Values**

- Maintain at each node a “sum” based on an associative operator $f$.
  - Updated during insert/delete, merge, extract, etc.
- Given $f : v \times v \to v$, and $l_f$
  - All operations on ordered tables are supported, and
  - $reduceVal(A) : T \to v = reduce f l_f A$
  - We want to be able to do $reduceVal$ in $O(1)$ work (assuming $f$ needs $O(1)$ work).
  - $f$ is known beforehand!
Ordered tables with reduced values

\[ f \text{ is } + \]

\[ f \text{ is max} \]
datatype Treap = Leaf | Node of (Treap × Treap × key × data × data)

fun reduceVal(T) = 
case T of
  Leaf ⇒ Reduce.I
  | Node(_, _, _, _, r) ⇒ r

fun makeNode(L, R, k, v) =
  Node(L, R, k, v, Reduce.f(reduceVal(L),
  Reduce.f(v, reduceVal(R))))
fun join′(T₁, T₂) =
case (T₁, T₂) of
  (Leaf, _) ⇒ T₂
  | (_ , Leaf) ⇒ T₁
  | (Node(L₁, R₁, k₁, v₁, s₁), Node(L₂, R₂, k₂, v₂, s₂)) ⇒
    if (priority(k₁) > priority(k₂)) then
      makeNode(L₁, join(R₁, T₂), k₁, v₁)
    else
      makeNode(join(T₁, L₂), R₂, k₂, v₂)
Example Application - Sales Data

- Sales information are kept by the time stamp in an ordered table.
  - (2/3/2013 – 12:30, $120)
- Find the total sales between $t_1$ and $t_2$
- $f$ is +
- $reduceVal(getRange(T, t_1, t_2))$ takes $O(\log n)$ work
**EXAMPLE APPLICATION - STOCK DATA**

- Stock prices information are kept by the time stamp in an ordered table.
  - (2/3/2013 – 12:30, $120/share)
- Find the maximum price between $t_1$ and $t_2$
- $f$ is max
- $reduceVal(getRange(T, t_1, t_2))$ takes $O(\log n)$ work
**Example Application - Interval Trees**

- An interval is a region on the real number line starting at $x_l$ and ending at $x_r$.
- An interval table supports the following operations on intervals:

  - $\text{insert}(A, I) : T \times (\text{real} \times \text{real}) \to T$
    - insert interval $I$ into table $A$
  - $\text{delete}(A, I) : T \times (\text{real} \times \text{real}) \to T$
    - delete interval $I$ from table $A$
  - $\text{count}(A, x) : T \times \text{real} \to \text{int}$
    - return the number of intervals crossing $x$ in $A$
Interval Trees

- Organize intervals as a BST based on lower-boundary as key
- Use the max upper boundary in the subtree as additional information.
COUNTING INTERVALS

1  datatype intTree = Leaf | Node of (intTree × intTree × real × real × real)
2
3  fun overlap(x, low, high) =
4      if (x ≥ low & x ≤ high) then 1 else 0

5  fun countInt(T, x) =
6  case T of
7      Leaf ⇒ 0
8      | Node(L, R, low, high, max) ⇒
9      if (x > max) then 0
10     else countInt(L, x)+
11        overlap(x, low, high)+
12     if (x > low) then countInt(R, x) else 0
15-210
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 24
DYNAMIC PROGRAMMING
SYNOPSIS

- Dynamic Programming
- Subset Sum Problem
- Minimum Edit Distance Problem
- Additional example applications
# Algorithmic Paradigms Contrasted

- Inductive Paradigms combine solutions to smaller subproblem(s).

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Subproblems</th>
<th>Reuse of Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide and Conquer</td>
<td>&gt; 1</td>
<td>NO</td>
</tr>
<tr>
<td>Contraction</td>
<td>= 1</td>
<td>NO</td>
</tr>
<tr>
<td>Greedy</td>
<td>= 1</td>
<td>NO</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>&gt; 1</td>
<td>YES</td>
</tr>
</tbody>
</table>
You can save some work if you remember the solutions to the smaller subproblems.
How much work does this code need?

(fun fib(n) =
  if (n ≤ 1) then 1
  else fib(n − 1) + fib(n − 2)
)

It turns out \( W_{fib}(n) = O(c^n) \) (Why?)
It also turns out that \( \text{fib}(n) \) can be computed with \( O(n) \) work.

- Note that \( n \) is not the right measure for modeling work here (Why? ) but it is convenient!
SOLUTION COMPOSITION GRAPH

- DAG
- Each node is a subproblem instance
- Edges model dependences
- Edges go from smaller to larger subproblems
- Vertices with no in-edges are base cases
- Vertices with no out edges are the instance we are trying to solve.
Dynamic programming can be seen as evaluating a DAG by navigating from the leaves to the root.
- Computing the subsolutions at each node as needed and when possible.

Work and span fall out of the DGA structure!
- Work: sum over nodes
- Span: Find the longest path!

Many DP solutions have significant parallelism, but some do not.
The challenge is to find the appropriate DAG structure for a given problem.

DP is most suitable for optimization problems.
  ▶ Solution optimizes (minimizes/maximizes) some criteria.

DP is also suitable for decision problems.
  ▶ Is there a solution to this instance?
Dynamic Programming

Top-down approach
- Starts at the root
- Uses recursion to solve the subproblems
- But remembers the solutions – memoization.
- Usually elegant and evaluates only the needed subproblems.

Bottom-up approach
- Starts at the leaves
- Traverses the DAG in some fashion.
- All subproblems may need to be computed.
- More parallelizable.

Coming up with the abstract inductive structure is important.
- Sharing and coding comes later.
### The Subset Sum (SS) Problem

Given a multiset of positive integers $S$ and a positive integer value $k$, determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

- **Given $S = \{1, 4, 2, 9, 9\}$**
  - No solution for $k = 8$
  - For $k = 7$ \{1, 4, 2\} is a solution.
- **NP-hard** if $k$ is unconstrained.
- We will include $k$ in the work bounds.
- $k$ is polynomial in $|S|$, work is polynomial in $|S|$.
- *Pseudo-polynomial work* solution.
THE SUBSET SUM PROBLEM

- Brute force: Consider all $2^n$ subset for a total work of $O(n2^n)$.
- Divide and Conquer: also ends up being exponential work.
- Sharing solutions however works.
THE SUBSET SUM PROBLEM

- To solve $SS(S, k)$, pick some element $a \in S$
- Solve (recursively) $SS(S \setminus \{a\}, k - a)$
  - If there is a solution, we are done.
- If not, solve $SS(S \setminus \{a\}, k)$.

```plaintext
1   fun ss(S, k) =
2     case (showl(S), k) of
3       (_, 0) ⇒ true
4       | (NIL, _) ⇒ false
5       | (CONS(a, R), _) ⇒
6         if (a > k) then ss(R, k)
7         else (ss(R, k - a) orelse ss(R, k))
```

DYNAMIC PROGRAMMING
CMU-Q 15-210 PARALLEL AND SEQUENTIAL DATA STRUCTURES AND ALGORITHMS
Fall 2013
The Subset Sum Problem DAG

\[ SS(\emptyset, 0) \rightarrow SS(\{1\}, 1) \rightarrow SS(\{1,1\}, 2) \rightarrow SS(\{1,1,1\}, 3) \]

\[ SS(\emptyset, 1) \rightarrow SS(\{1\}, 2) \rightarrow SS(\{1,1\}, 3) \]

\[ SS(\emptyset, 2) \rightarrow SS(\{1\}, 2) \rightarrow SS(\{1,1\}, 3) \]

\[ SS(\emptyset, 3) \rightarrow SS(\{1\}, 2) \rightarrow SS(\{1,1\}, 3) \]
How many distinct subproblems do we need to solve?
The Subset Sum Problem

- For \( SS(S, k) \), there are only \(|S|\) distinct lists ever used.
- The second argument decreases down to 0, so has at most \( k + 1 \) values.
- So we have at most \(|S|(k + 1) = O(k|S|)\) instances.
- Each instance has constant work \( \Rightarrow \) total \( O(k|S|) \) work.
- Longest path in DAG is \(|S| \Rightarrow \) span is \( O(|S|) \)
  - \( O(k) \) parallelism.
**The Subset Sum Problem**

- Why *pseudo-polynomial*?
  - For \( k \), the input size is \( \log k \), but the work is \( O(2^{\log k |S|}) \)
    - *Exponential* in input size!
  - If \( k \leq |S|^c \) for some constant \( c \), then work is \( O(k|S|) = O(|S|^{c+1}) \) on input of size \( c \log |S| + |S| \)
**Minimum Edit Distance**

**Minimum Edit Distance (MED)**

Given a character set \( \Sigma \) and two sequences of characters \( S = \Sigma^* \) and \( T = \Sigma^* \), determine the minimum number of insertions and deletions of single characters required to transform \( S \) to \( T \).

- Start with \( S = \langle A, B, C, A, D, A \rangle \)
  - Delete \( C \)
  - Delete last \( A \)
  - Insert a \( C \)
- You get \( T = \langle A, B, A, D, C \rangle \)
- So \( MED(S, T) = 3 \)
Applications of MED

- Spelling correction
  - What is an English word close to *Ynglisd*?
- Storing multiple versions of files efficiently.
- Approximate matching of genome sequences
**Minimum Edit Distance**

- Given $S = s :: S'$ and $T = t :: T'$
- If $s = t$, $MED(S, T)$ is determined by $S'$ and $T'$
- Otherwise we have two subproblems:
  - Find $MED(S, T')$ – consider a deletion from $T$ to get $T'$
  - Find $MED(S', T)$ – consider a deletion to $S$ to get $S'$
- Find the minimum and add 1.
**Minimum Edit Distance**

```
fun MED(S, T) =
case (showl(S), showl(T)) of
  (_, NIL) ⇒ |S|
  (NIL, _) ⇒ |T|
  (CONS(s, S′), CONS(t, T′)) ⇒
    if (s = t) then MED(S′, T′)
    else 1 + min(MED(S, T′), MED(S′, T))
```

- If run recursively, this would take exponential work.
  - Binary tree with linear depth!
- But there is significant sharing!
MINIMUM EDIT DISTANCE

ABC = <A,B,C>

MED(ABC, DBC)  
MED(BC, DBC)  
MED(C, DBC)  
MED(ϕ, DBC)  

MED(BC, BC)  
MED(C, BC)  
MED(ϕ, BC)  

MED(ABC, BC)  
MED(ABC, C)  
MED(BC, C)  
MED(ϕ, BC)  

MED(ϕ, ϕ)  

MED(ϕ, ϕ)
**Minimum Edit Distance**

- There are at most $|S| + 1$ possible values for the first argument.
- There are at most $|T| + 1$ possible values for the second argument.
- So we have $(|S| + 1) \times (|T| + 1) = O(|S| |T|)$ possible subproblems, each of constant work.
  - Total work is $O(|S| |T|)$.
- Total span is $O(|S| + |T|)$ (Why?)
A longest common subsequence of strings $S_1$ and $S_2$ is a longest subsequence shared by both.

$LCS(ABCDEF, EBCEG) = BCE$

May be empty or not necessarily unique.

$LLCS(S_1, S_2)$ computes the length of the LCS.

Subproblem structure is very similar to MED. (Work it out!)
**Optimal Change**

- For a currency with coins $C_1, C_2, \ldots C_n = 1$ (cents), what is the minimum number of coins needed to make $K$ cents of change.
- US Currency has 25, 10, 5, 1 cent coins.
- To give back 63 cents, you need to give 25+25+10+1+1+1, a total of 6 coins.
  - Greedy works in this case, but not always
  - If you had a 21 cent coin (for some strange reason), greedy would not work.
- DP solutions solves two subproblems $K_1 = i$ and $K_2 = K - i$ for all $i = 1, \ldots \lfloor K/2 \rfloor$
- Then chooses $i$ that minimizes the sum of the solutions
0-1 Knapsack

- Items with “benefit” $p_i$ and cost $w_i$
  - $x_i = 1$ or $0$ – take item $i$ or not.
- Maximize $\sum_{j=1}^{n} p_j \cdot x_j$
- Subject to $\sum_{j=1}^{n} w_j \cdot x_j \leq c$
- Optimal Exam Strategy Problem (:-)
  - Questions 1 through $n$, worth $p_1, \ldots, p_n$ points.
  - Time estimate for solving question $j$ is $w_j$
  - You have $T$ units of time.
  - Which questions do you solve to maximize your grade?
  - Subproblem structure is resembles the thinking for subset sum problem
Optimal Matrix Multiplication

- We need to multiply $n$ matrices $A_1 \times A_2 \times \cdots A_n$
  - $A_i$ has sizes $p_{i-1} \times p_i$ and $A_{i+1}$ has sizes $p_i \times p_{i+1}$
  - Multiplying $A_i$ and $A_{i+1}$ needs $O(p_{i-1} \cdot p_i \cdot p_{i+1})$ work

- What is the best way to “parenthesize” the sequence to minimize the number of scalar multiplications?

- $m[i, j]$ is the minimum number of scalar multiplications for multiplying $A_i \times \cdots \times A_j$
  - A subproblem
**Optimal Matrix Multiplication**

\[
m[i, j] = \begin{cases} 
0 & i = j \\
\min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j\} & i < j
\end{cases}
\]

- Find that \( k \) that minimizes the cost of multiplying \( A_i \times \cdots \times A_j \)
- We need to compute \( m[1, n] \) and how we got that (the choice of \( k \)'s when we are minimizing subproblems)
SYNOPSIS

- Top-down Dynamic Programming
- Bottom-up Dynamic Programming
- Optimal Binary Search Trees
**TOP-DOWN DP**

- Run the recursive code as is:
  - Start with the root
  - Work down to the leaves

- **Memoization**: We need to avoid redundant computation.
  - If we encounter the same arguments, we just look up the solution
  - If not, we compute once and store in a memo table.

- Checking for equal arguments could be costly.
  - We use simple surrogates for actual arguments (e.g., integers)
**Top-down DP for MED**

- MED takes two sequences and on each recursive call, uses suffixes of the original sequences.
  - There is a one-to-one mapping from non-negative integers to suffixes (rather to suffix lengths!)
  - Could also use prefixes!
  - This makes indexing a bit easier.
fun MED(S, T) = 
  case (showl(S), showl(T)) of 
    (_, NIL) ⇒ |S|
    | (NIL, _) ⇒ |T|
    | (CONS(s, S'), CONS(t, T')) ⇒
      if (s = t) then MED(S', T')
    else 1 + min(MED(S, T'), MED(S', T))
MED with Surrogates

1 fun MED(S, T) = let
2     fun MED′(i, 0) = i
3         | MED′(0, j) = j
4         | MED′(i, j) = case (S_i = T_j) of
5             true ⇒ MED′(i - 1, j - 1)
6             | false ⇒ 1 + min(MED′(i, j - 1), MED′(i - 1, j))
7     in
8     MED′(|S|, |T|)
9 end

- MED′ has i and j, instead of S and T
  - i represents S_{0, \ldots, i - 1}
  - j represents T_{0, \ldots, j - 1}

- No memo table yet!
We can now add a memo table, accessed with \((i,j)\)

- We can also use a two dimensional array!

```ocaml
1  fun memo f (M, a) =
2   case find(M, a) of
3     SOME(v) ⇒ (M, v)
4   | NONE ⇒ let
5      val (M', v) = f(M, a)
6    in
7       (update(M', a, v), v)
8   end
```
**Memoized MED**

1. \( \text{fun } \text{MED}(S, T) = \text{let} \)
2. \( \quad \text{fun } \text{MED}'(M, (i, 0)) = (M, i) \)
3. \( \quad \quad \text{|MED}'(M, (0, j)) = (M, j) \)
4. \( \quad \quad \text{|MED}'(M, (i, j)) = \text{case } (S_i = T_j) \text{ of} \)
5. \( \quad \quad \quad \text{true } \Rightarrow \text{MED}''(M, (i - 1, j - 1)) \)
6. \( \quad \quad \quad \text{|false } \Rightarrow \text{let} \)
7. \( \quad \quad \quad \quad \text{val } (M', v_1) = \text{MED}''(M, (i, j - 1)) \)
8. \( \quad \quad \quad \quad \text{val } (M'', v_2) = \text{MED}''(M', (i - 1, j)) \)
9. \( \quad \quad \quad \quad \text{in } (M'', 1 + \min(v_1, v_2)) \text{ end} \)
10. \( \quad \quad \text{and } \text{MED}''(M, (i, j)) = \text{memo } \text{MED}'(M, (i, j)) \)
11. \( \text{in} \)
12. \( \quad \text{MED}'(\{\}, (|S|, |T|)) \)
13. \( \text{end} \)

- Purely functional
- but highly sequential
**BOTTOM-UP DP**

- Start with the leaves
- Works through the subproblems consistent with the DAG
  - if \((u, v)\) is a dependency edge in the DAG, compute \(u\) before \(v\), for all such \(u\).
  - All values will be available for \(v\) when they are needed!
- Uses a memo table.
- Understanding the DAG structure is important
Bottom-Up DP for MED

Dag for MED("tcat", "atc")
We can go by diagonals.
We can go by rows.
We can go by columns.
Bottom-up DP for MED

```ml
fun MED(S, T) = let
  fun MED'(M, (i, 0)) = i
  | MED'(M, (0, j)) = j
  | MED'(M, (i, j)) = case (S_i = T_j) of
    true ⇒ M_{i-1,j-1}
  | false ⇒ 1 + min(M_{i,j-1}, M_{i-1,j})

  fun diagonals(M, k) =
    if (k > |S| + |T|) then M
    else let
      val s = max(0, k - |T|)
      val e = min(k, |S|)
      val M' = M \cup \{(i, k-i) ↦ MED'(M, (i, k-i)) : i ∈ {s, ..., e}\}
    in
      diagonals(M', k + 1)
    end

  in
    diagonals({}, 0)
  end
```

Dynamic Programming – II
CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Spring 2013
In Round 0, we compute $M_{0,0}$
In Round 1, we compute $M_{0,1}$ and $M_{1,0}$
In Round 2, we compute $M_{0,2}$, $M_{1,1}$, $M_{2,0}$
In Round 3, we compute $M_{0,3}$, $M_{1,2}$, $M_{2,1}$, $M_{3,0}$
... 
How about parallelism?
Let’s revisit BSTs
- The cost of finding a key is proportional to the depth of the key in the tree.
- Fully balanced BST with $n$ nodes $\Rightarrow$ average depth is $\log n$

Suppose you have a (fixed/static) dictionary and you know the probability that a given key will be accessed

What is the BST structure with the lowest overall cost?
The optimal binary search tree (OBST) problem is given an ordered set of keys $S$ and a probability function $p : S \rightarrow [0 : 1]$, to find $\hat{T}$

$$\hat{T} = \arg \min_{T \in \text{Trees}(S)} \left( \sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\text{Trees}(S)$ is the set of all BSTs on $S$, and $d(s, T)$ is the depth of the key $s$ in the tree $T$ (Assume the root has depth 1).
**Optimal Binary Search Trees**

<table>
<thead>
<tr>
<th>key</th>
<th>$k_1$</th>
<th>$k_2$</th>
<th>$k_3$</th>
<th>$k_4$</th>
<th>$k_5$</th>
<th>$k_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(key)$</td>
<td>$1/8$</td>
<td>$1/32$</td>
<td>$1/16$</td>
<td>$1/32$</td>
<td>$1/4$</td>
<td>$1/2$</td>
</tr>
</tbody>
</table>

\[
\text{Cost} = \frac{1}{8} \times 2 + \frac{1}{32} \times 4 + \frac{1}{16} \times 3 + \frac{1}{32} \times 4 + \frac{1}{4} \times 1 + \frac{1}{2} \times 2 = \frac{31}{16}
\]
How many binary search trees of $n$ distinct keys are there?

- Hint: Think of matrix chain multiplication!

In DP, an optimal solution should be based on optimal subproblem solutions.

One of the keys ($S_r$) must be at the root of the optimal tree.

- Both subtrees must be optimal.

How do we select $S_r$?

- Pick the key with highest probability and put it at the root, and recurse?
- Does not really work!
**Optimal Binary Search Trees**

- Try all elements as a potential root
- For each, recursively find their optimal solutions
- Pick the best among the $|S|$ possibilities.
- All elements under a root are contiguous in the sorted sequence.
Optimal Binary Search Trees

- Use \((i, j)\) as a surrogate for the tree spanning \(S_i, \ldots, S_j\).
- Let \(T\) be the tree covering \(S_i, \ldots, S_j\) with root \(S_r, i \leq r \leq j\), with \(T_L T_R\) as the subtrees.

\[
\text{Cost}(T) = \sum_{s \in T} d(s, T) \cdot p(s)
\]

\[
= p(S_r) + \sum_{s \in T_L} (d(s, T_L) + 1) \cdot p(s) + \sum_{s \in T_R} (d(s, T_R) + 1) \cdot p(s)
\]

\[
= \sum_{s \in T} p(s) + \sum_{s \in T_L} d(s, T_L) \cdot p(s) + \sum_{s \in T_R} d(s, T_R) \cdot p(s)
\]

\[
= \sum_{s \in T} p(s) + \text{Cost}(T_L) + \text{Cost}(T_R)
\]

- Find the \(r, i \leq r \leq j\) that minimizes this cost.
**Optimal Binary Search Trees**

1. \[ \text{fun } OBST(S) = \]
2. \[ \text{if } |S| = 0 \text{ then } 0 \]
3. \[ \text{else } \left( \sum_{s \in S} p(s) \right) + \min_{i \in \langle 1 \ldots |S| \rangle} \left( OBST(S_{i-1}) + OBST(S_{i+1}|S|) \right) \]

- How many possible subproblems are there?
  - A subsequence can end at \( n \) different positions
  - For the \( i^{th} \) end position there are \( i \) possible start positions.

\[ \sum_{i=1}^{n} i = n(n + 1)/2 \in O(n^2) \] possible subproblems.

- Longest path of dependences in the DAG is \( O(n) \) since recursion can go down for \( n \) levels (Why?)
WORK AND SPAN

- Cost of each subproblem is not uniform! (Why?)
- Each subproblem has $O(n)$ work and $O(\log n)$ span (Why?)
- We get total $O(n^3)$ work and $O(n \log n)$ span. (Why?)
CODE FOR OPTIMAL BST

```plaintext
fun OBST(S) = let
  fun OBST'(i, l) =
    if l = 0 then 0
    else \( \sum_{k=0}^{l-1} p(S_{i+k}) + \min_{k=0}^{l-1} (OBST'(i, k) + OBST'(i + k + 1, l - k - 1)) \)
    in
    OBST'(1, |S|)
  end
end
```
For a bottom up version, a triangular table is sufficient

\[
c_{ij} = \text{optimal cost of the tree covering } S_{ij}
\]
SYNOPSIS

- Hashing and Hash Tables
- Handling Collisions
  - Linear Probing
  - Quadratic Probing
Hash Tables – Basic Ideas

- Data structure that allows you to quickly insert, delete, and retrieve items with expected $O(1)$ work.
- Relies on
  - a fixed size array data structure (of some size $m$), and
  - a hash function that can map from a potentially infinite space of keys to integer indexes $[0, \ldots, m-1]$
- Disadvantages
  - Collisions
  - Increased memory use to avoid collisions
  - Not work efficient for $findmin$, $findmax$, or extracting keys in sorted order
Hash Table - Basic Ideas

Space of possible values of the data elements (possibly infinite)

\[ x_1, x_2, x_3, \ldots, x_4 \]

Hash: \( X \rightarrow \{0, \ldots, m-1\} \)

Mapping

Space of possible values of the indices (0 to m-1)

\[ 0, \ldots, \ldots, m-1 \]
Hash Functions

- There is a deep theory behind hash functions.
- We will be interested in some simple functions.
- We will assume hash functions have the idealized property of *simple uniform hashing*:
  - The hash function uniformly distributes keys in range $[0, \ldots, m - 1]$
  - Hash value for one key is independent of the hash value for another key.
For integers key we can use a linear congruential hash function

\[ h(x) = (ax + b) \mod m \]

where \( a \in [1, \ldots, m - 1] \), \( b \in [0, \ldots, m - 1] \), and \( m \) is prime.
Hash Functions

For strings, we can use a polynomial like

$$h(S) = \left( \sum_{i=1}^{\left| S \right|} s_i a^i \right) \mod m$$
Support *insert*, *find* and *delete*.

Can implement abstract data types *Set* and *Table*.

Do not require total ordering on the universe of keys.

*Collision* is the main issue

- Two keys hash to the same location.
- Impossible to avoid if we do not know the keys in advance
  - Size of key universe >> size of table.
COLLISIONS

For a table size of 365, one needs 23 keys for a 50% chance of collision and 66 for a 99% chance of collision (Why?)

- Birthday paradox
Handling Collisions

- **Separate chaining**
  - Store elements not in a table, but in linked lists (containers, bins) hanging off the table.

- **Open addressing**:
  - Put everything into the table, but not necessarily into cell $h(k)$.

- **The perfect hash**:
  - When you know the keys in advance, construct hash functions that avoids collisions entirely.

- **Multiple-choice hashing/Cuckoo hashing**:
  - Consider exactly two locations $h_1(k)$ and $h_2(k)$ only.
Handling Collisions

- We will only consider the first two.
- We will assume we have a set \( n \) keys \( K \) and a hash function \( h : \text{key} \rightarrow [0, \ldots, m - 1] \) for some \( m \).
Separate Chaining

- Maintain an array of linked lists (buckets).
- Keys that hash to the same value live in the same list at location $h(k)$
- **Insertion**: Insert at the beginning
  - Multiple inserts for the same key $\Rightarrow$ traverse the list
  - May as well insert at the end.
- **Find**: hash to $h(k)$ and search in the list.
- **Delete**: remove from the list.
Separate Chaining

Costs depend on the load factor $\lambda = n/m$ which is also the average length of a list.
SEPARATE CHAINING

- Assume $h(k)$ takes $O(1)$ work and we have simple uniform hashing
- *Unsuccessful search* takes expected $\Theta(1 + \lambda)$ work.
  - $O(1)$ for $h(k)$ and $\lambda$ for traversing the list.
**Separate Chaining**

- *Successful search* takes expected $\Theta(1 + \lambda)$ work.
- Cost of *Successful search* = Cost of unsuccessful search at the time of insertion (Why?)
- With $i$ keys, the unsuccessful search would take $(1 + i/m)$ work.
- Averaging over $i$ we get

$$\frac{1}{n} \sum_{i=0}^{n-1} (1+i/m) = 1 + (n-1)/2m = 1 + \lambda/2 - \lambda/2m = \Theta(1 + \lambda)$$

- Considering constant factors, successful search looks at 1/2 the list on the average.
**Open Addressing**

- No lists – everything is stored in the array directly
- The arrays is some constant factor larger than the maximum number of keys we want to store.
AN EXAMPLE

Initially the hash table is empty

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
An Example

Hash Tables

Insert 100
An Example

Hash Tables

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Spring 2013
**An Example**

Hash Tables

Insert 144
An Example

Hash Tables

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Spring 2013

Insert 169
An Example

Insert 196
An Example

Insert 225

<table>
<thead>
<tr>
<th>0</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>225</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>169</td>
</tr>
</tbody>
</table>
**An Example**

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>225</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>169</td>
</tr>
</tbody>
</table>

Insert 256  COLLISION because location 6 is full.  Try location 6 + 1 = 7
Insert 289. COLLISION because location 9 is full. Try location \((9+1)\mod 10 = 0\)
### An Example

<table>
<thead>
<tr>
<th>0</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>225</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>169</td>
</tr>
</tbody>
</table>

Insert 289  COLLISION because location 9 is full.

Try location \((9+1) \mod 10 = 0\) FULL
An Example

Hash Tables

Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10 = 0 FULL

Try location (9+2)mod 10 = 1 FULL
An Example

Hash Tables

Insert 289  COLLISION because location 9 is full.

Try location \((9+1)\mod 10 = 0\) FULL

Try location \((9+2)\mod 10 = 1\) FULL

Try location \((9+3)\mod 10 = 2\) AVAILABLE
AN EXAMPLE

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>121</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>289</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>225</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>196</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>169</td>
<td></td>
</tr>
</tbody>
</table>

Insert 324  COLLISION because location 4 is full.

Try location \((4+1)\mod 10 = 5\) FULL
AN EXAMPLE

Insert 324  COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL
AN EXAMPLE

<table>
<thead>
<tr>
<th>0</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td>2</td>
<td>289</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>225</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>169</td>
</tr>
</tbody>
</table>

Insert 324 COLLISION because location 4 is full.

Try location \((4+1)\mod 10 = 5\) FULL

Try location \((4+2)\mod 10 = 6\) FULL

Try location \((4+3)\mod 10 = 7\) FULL
AN EXAMPLE

Hash Tables

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Spring 2013

0 100
1 121
2 289
3
4 144
5 225
6 196
7 256
8 324
9 169

Insert 324 COLLISION because location 4 is full.

Try location \((4+1)\mod 10= 5\) FULL

Try location \((4+2)\mod 10= 6\) FULL

Try location \((4+3)\mod 10= 7\) FULL

Try location \((4+4)\mod 10= 8\) AVAILABLE
AN EXAMPLE

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>289</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>324</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>169</td>
<td></td>
</tr>
</tbody>
</table>

Insert 361 COLLISION because location 1 is full.
Try location (1+1)mod 10 = 2 FULL
### An Example

#### Insert 361

COLLISION because location 1 is full.

Try location \((1+1)\mod 10= 2\) FULL

Try location \((1+2)\mod 10= 3\) AVAILABLE

<table>
<thead>
<tr>
<th>0</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td>2</td>
<td>289</td>
</tr>
<tr>
<td>3</td>
<td>361</td>
</tr>
<tr>
<td>4</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>225</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td>324</td>
</tr>
<tr>
<td>9</td>
<td>169</td>
</tr>
</tbody>
</table>
Open Addressing

- Open addressing uses an ordered sequence of locations.
- $h(k, i)$ gives us the $i^{th}$ location for key $k$.
- $\langle h(k, 0), h(k, 1), h(k, 2), \ldots \rangle$ is the probe sequence.
- Try these locations in order until an empty cell is found and insert there.
Open Addressing - Insert

fun insert(T, k) = let
  fun insert'(T, k, i) =
    case nth T h(k, i) of
      NONE ⇒ update(h(k, i), k) T |
      _ ⇒ insert'(T, k, i + 1)
  in
    insert'(T, k, 1)
  end

- $T$ must be an ST array - otherwise work and span are not constant.
- Need to check if table is full and the key is already in the table or not.
function find(T, k) =
  let
    function find'(T, k, i) =
      case nth T h(k, i) of
        NONE ⇒ false
        SOME(k') ⇒ if eq(k, k') then true
                      else find'(T, k, i + 1)
      in
    find'(T, k, 1)
  end
We can not just delete an items and set its cell to \textit{NONE}! (Why ?)

\textit{find} will stop searching if it encounters an empty cell.

Use \textit{lazy delete}

- Instead of deleting, use a special value \textit{HOLD}.

1. \textbf{datatype} $\alpha \text{ entry} = \text{EMPTY} \mid \text{HOLD} \mid \text{FULL}$ of $\alpha$

- Find and Insert will need to be changed accordingly.
- Lazy delete effectively increases load factor.
- Rehashing to the rescue!
Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing
LINEAR PROBING

- We check cell at $h(k, i) = (h(k) + i) \mod m$ in $i^{th}$ probe.
- $m$ possible probe sequences.
- Keys tend to cluster – *primary clustering*.
  - Inserts add to a cluster
  - Probe sequences get longer and longer
Impact of Clustering

- Assume table is half full ($\lambda = 1/2$)
- Minimum clustering when every other cell is empty!
- Average probes for insert is $3/2$
  - One probe to check cell $h(k)$
  - With $1/2$ chance try the next cell (which by design should be empty)
**IMPACT OF CLUSTERING**

- Worst case: all keys are clustered to the second half of the array. (Remember $\lambda = 1/2 \Rightarrow m = 2n$)
- How many probes for positions 0 through $n - 1$?
  - 1 (Why?)
- How many probes when initial hash is to cell $n$?
  - $n$ (Why?)
- How many probes when initial hash is to cell $n + 1$?
  - $n - 1$ (Why?)
- Average is
  \[
  \frac{(n+[n+(n-1)+(n-2)+\ldots+1])}{m} = \frac{n}{m} + \frac{n(n+1)}{2m} \approx \frac{n}{4}
  \]
- Even though though the average cluster length is 2, the cost is about $n/4$ probes.
Costs for Linear Probing

- Given a hash table of size $m$ and with $n = \lambda m$ keys.
- The cost of an unsuccessful search/insert is
  \[
  \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda^2} \right)
  \]
- The cost of an successful search is
  \[
  \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right)
  \]
Expected Probes for Insertion and Unsuccessful Search
Costs for Linear Probing

Expected Probes for Insertion and Unsuccessful Search

Number of Probes

Load factor

0.00 0.10 0.20 0.30 0.40 0.50 0.60

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms
Spring 2013
Expected Probes for Successful Searches

Number of Probes

Load factor

Costs for Linear Probing
Expected Probes for Successful Searches

Costs for Linear Probing
We check cell at \( h(k, i) = (h(k) + i^2) \mod m \) in \( i^{th} \) probe.

- Makes longer jumps
- Avoids primary clustering
- But has secondary clustering.
- Since there are \( m \) possible positions there are \( m \) probe sequences.
- Not all available cells get probed (Why?)
Quadratic Probing

- If $m$ is prime and the table is at least half empty, then quadratic probing will always find an empty location.
- Furthermore, no locations are checked twice.
QUADRATIC PROBING

- Consider two probe locations $h(k) + i^2$ and $h(k) + j^2$, $0 \leq i, j < \lceil m/2 \rceil$.
- Suppose the locations are the same but $i \neq j$.

\[
\begin{align*}
  h(k) + i^2 &\equiv (h(k) + j^2) \mod m \\
  i^2 &\equiv j^2 \mod m \\
  i^2 - j^2 &\equiv 0 \mod m \\
  (i - j)(i + j) &\equiv 0 \mod m
\end{align*}
\]

- Therefore, either $i - j$ or $i + j$ are divisible by $m$.
- But since both $i - j$ and $i + j$ are less than $m$ and $m$ is prime, they cannot be divisible by $m$.
- Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location.
**Quadratic Probing**

- Computing the next hash value is only slightly more expensive

\[
h_i - h_{i-1} \equiv (i^2 - (i - 1)^2) \mod m
\]

\[
h_i \equiv (h_{i-1} + 2i - 1) \mod m
\]

- If the table gets too full, one can resize and rehash
  - Constant additional overhead
**Double Hashing**

- Uses two hash-functions:
  - initial location
  - size of the jump

- $i^{th}$ probe is

\[ h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m. \]

- Different keys are likely to have different values jump function if they collide.
- Avoids secondary clustering
- $h_2(k)$ should be relatively prime to $m$ to probe each locations.
  - $m$ prime and $0 < h_2(k) < m$ is one option.
Double Hashing

- The average number of probes for an unsuccessful search or an insert is at most
  \[ 1 + \lambda + \lambda^2 + ... = \left( \frac{1}{1 - \lambda} \right) \]

  - Why?

- The average number of probes for a successful search is
  \[ \frac{1}{\lambda} \left( 1 + \ln \left( \frac{1}{1 - \lambda} \right) \right) \]

  - Same argument of averaging over probes at insertion time.
## Double Hashing

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>1/4</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>successful</strong></td>
<td>1.2</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td><strong>unsuccessful</strong></td>
<td>1.3</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
</tbody>
</table>

- Allows for smaller tables than linear or quadratic probing
- Higher cost for hash function
**Parallel Hashing**

- \( \text{injectCond}(\text{IV}, S) : (\text{int} \times \alpha)\text{seq} \times (\alpha\text{option})\text{seq} \rightarrow (\alpha\text{option})\text{seq}. \)

- Conditionally writes each value \( v_j \) into location \( i_j \) of \( S \)
  - if the location is set to NONE

```ocaml
fun insert(T, K) = let
  fun insert'(T, K, i) =
    if |K| = 0 then T
    else let
      val T' = injectCond([(h(k, i), k) : k ∈ K], T)
      val K' = {k : k ∈ K | T[h(k, i)] ≠ k}
      in
      insert'(T', K', i + 1) end
    in
    insert'(T, k, 1)
  end
```

---

**Hash Tables**

CMU-Q 15-210 Parallel and Sequential Data Structures and Algorithms

Spring 2013
15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 27
Priority Queues
Synopsis

- Priority Queues
- Heaps
- Meldable Priority Queues
- Leftist Heaps
Priority Queues

- Abstract Data Type supporting
  - deleteMin/deleteMax
  - insert
- Used in many useful algorithms
  - Dijkstra’s Algorithm
  - Prim’s Algorithm for MST
  - Constructing Huffman Codes
  - Heapsort
Heapsort

1 fun sort S =
2 let
3   val pq = iter Q.insert Q.empty S
4   fun sort' pq =
5     let
6       case (PQ.deleteMin pq) of
7         NONE ⇒ []
8         | SOME(v, pq') ⇒ v :: sort'(pq')
9     in
10    Seq.fromList(sort'pq)
11 end
**Underlying Implementations**

- **Sorted and Unsorted Lists/Arrays**
  - One of `deleteMin` and `insert` is fast \(O(1)\)
  - The other is slow. \(O(n)\)

- **Balanced binary search trees**
  - Both operations have \(O(\log n)\) work and span.

- **Binary heaps**
  - Both operations have \(O(\log n)\) work and span.
  - But binary heaps provide a \(O(1)\) work `findMin` operation.
Heaps

- A *min-heap* (*max-heap*) is a rooted tree
- Key at every node is \( \leq \) (*\( \geq \)*) all descendants.
- A *binary heap* is a heap which has
  - *Shape property*: The tree is a complete binary tree
    - All levels of the tree are completely filled except the bottom level, which is filled from the left
  - *Heap Property*
A complete tree
Binary Heaps

An incomplete tree
**Binary Heaps**

- Shape Property $\Rightarrow$ binary heap can be maintained in an array.
- Index of a parent or a child is very easy to compute
- Operations first restore shape property, then heap property.
Binary Heaps and Arrays
We can insert elements one-by-one
  - With balanced binary trees and binary heaps, work is $O(n \log n)$
  - Can we do better?

Build the heap recursively
  - If left and right sides are already heaps, just shift down the root element.
fun sequentialFromSeqS =
let
  fun heapify(S, i) =
    if (i >= |S|/2) then S
    else let
      val S' = heapify(S, 2 * i + 1)
      val S'' = heapify(S', 2 * i + 2)
      in shiftDown(S'', i) end
    in heapify(S, 0) end
Cost Analysis

- **shiftDown** does $O(\log n)$ work on subtree of size $n$
- $W(n) = 2W(n/2) + O(\log n) \in O(n)$
- Opportunities for parallelism?
**Parallel Heapify**

- Green cells are OK
- All the pinks cells can be shifted down in parallel
- Then all purple cells can be shifted down in parallel
- (All) Red cell(s) can be shifted down in parallel
Parallel Heapify
Parallell Heapify

We use Single-threaded sequences

```plaintext
fun fromSeq S : 'a seq =
  let
    fun heapify (S, d) =
      let
        val S' = shiftDown (S, ⟨2^d - 1, ..., 2^{d+1} - 2⟩, d)
      in
        if (d = 0) then S'
        else heapify (S', d - 1)
      end
    in
      heapify (S, ⌊log_2 n⌋ - 1)
    end
```

- $S(n) = S(n/2) + O(\log n) \in O(\log^2 n)$
**Parallel Heapify**

- \( d = 2 \Rightarrow \text{shiftDown} (S, <3, 4, 5, 6>, 2) \)
- \( d = 1 \Rightarrow \text{shiftDown} (S, <1, 2>, 1) \)
- \( d = 0 \Rightarrow \text{shiftDown} (S, <0>, 0) \)
# Priority Queues – Summary

<table>
<thead>
<tr>
<th>Data. Str.</th>
<th>findMin</th>
<th>deleteMin</th>
<th>insert</th>
<th>fromSeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted linked list</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>balanced search tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Meldable Priority Queues

- Priority Queues with an additional *meld* operation
  - Just like the union in BSTs
  - Takes two meldable PQs and returns the union as a meldable PQ
- Implementations uses *leftist heaps*
  - Same work and span as binary heaps for insert, deletemin
  - Meld has $O(\log n + \log m)$ work and span where $m$ and $n$ are the heap sizes
Min Heaps

- Binary tree
- Maintains the heap property
- But does *not* maintain the complete binary tree property
- Here is an example:

```
  3
 / \
7 8
/ \
11 15
/ \
22 16
```
To implement deleteMin

- Remove the root

```
    7   o     o 8
  / \                   
11 o   o 15
  / \                 
22 o   o 16
```

We can then use meld to union the heaps.
To implement insert
  ▶ We create a single node heap
  ▶ meld it with the original heap
fromSeq is also easy using reduce

val pq = Seq.reduce Q.meld Q.empty
  (Seq.map Q.singleton S)
**THE MELD OPERATION**

- So we only need the *meld* operation
- Consider

```
  4 o
 / \ / \ o 3
11 o o 7 8 o o 5
 / \ / 
19 o 23 14 o
```

- Which element goes to the root?
**The Meld Operation**

- Select the tree with the smaller root and recursively **meld** with one of its children.

```
      o 3
     /  \
   8    o = meld ( 4    ,  o 5 )
     /    /
  14  o 11 o 7
     /  /
19  o 23
```
The Meld Operation

- Applying recursively

```
        o 3
       / \  
      8 o o 4
     /   / \
    14 o 11 o = meld ( o 7 o 5)
     / \ 
    19 o o 23
```
**The Meld Operation**

- Applying recursively

```
    3
   / \
  8   4
 /   / \
14  11   5
 /   /   \
19   23   = meld(7, empty)
```

- Melding $A$ with an empty heap gives $A$
THE MELD OPERATION

1 datatype $PQ = Leaf \mid Node \ (key \times PQ \times PQ)$

2 fun meld($A$, $B$) =

3 case ($A$, $B$) of

4 (__, Leaf) ⇒ $A$

5 | (Leaf, __) ⇒ $B$

6 | ($Node(k_a, L_a, R_a)$, $Node(k_b, L_b, R_b)$) ⇒

7 case Key.compare ($k_a$, $k_b$) of

8 LESS ⇒ $Node(k_a, L_a, meld(R_a, B))$

9 | __ ⇒ $Node(k_b, L_b, meld(A, R_b))$

- Traverses the right spines of the trees
- Could be $\Theta(|A| + |B|)$ in the worst case.
**LEFTIST HEAPS**

- When melding, keep trees *deeper* on the left.
- Define
  
  \[ \text{rank}(x) = \# \text{ of nodes on the right spine of the subtree rooted at } x, \]

- For all nodes, rank can be inductively defined
  
  \[ \text{rank}(\text{leaf}) = 0 \]
  \[ \text{rank}(\text{node}(_, _, R)) = 1 + \text{rank}(R) \]
**Leftist Property**

- For all node $x$ in a leftist heap,
  \[ \text{rank}(L(x)) \geq \text{rank}(R(x)) \]
  - $L(x)$ and $R(x)$ are the left and child children of $x$
- Allows for
  
  \[
  \begin{array}{c}
  \circ 1 \\
  \circ 2 \\
  \circ 3 \\
  \vdots \\
  \circ n \\
  \end{array}
  \]

- But this is OK (Why?)
**Leftist Heaps**

- Most items pile to the left
- Right spine is relatively short!

**Lemma**

In a leftist heap with $n$ entries, the rank of the root node is at most $\log_2(n + 1)$. 
**Leftist Heaps**

1. **datatype** \( PQ = \text{Leaf} | \ Node \ of \ (\text{int} \times \text{key} \times PQ \times PQ) \)
2. **fun** \( \text{rank} \ \text{Leaf} = 0 \)
   3. \( \text{rank} \ (\text{Node}(r, _, _, _)) = r \)
4. **fun** \( \text{makeLeftistNode} (v, L, R) = \)
   5. \( \text{if} \ (\text{rank}(L) < \text{rank}(R)) \)
   6. \( \text{then} \ \text{Node}(1 + \text{rank}(L), v, R, L) \)
   7. \( \text{else} \ \text{Node}(1 + \text{rank}(R), v, L, R) \)

*Puts lower rank subtree to the right!*
Leftist Heaps

```haskell
fun meld (A, B) =
  case (A, B) of
    (_, Leaf) ⇒ A
    | (Leaf, _) ⇒ B
    | (Node(_, ka, La, Ra), Node(_, kb, Lb, Rb)) ⇒
      case Key.compare(ka, kb) of
        LESS ⇒ makeLeftistNode (ka, La, meld(Ra, B))
        | _ ⇒ makeLeftistNode (kb, Lb, meld(A, Rb))
```
**Leftist Heaps**

**Theorem**

If $A$ and $B$ are leftists heaps then

- the $meld(A, B)$ algorithm runs in $O(\log(|A|) + \log(|B|))$ work, and
- returns a leftist heap containing the union of $A$ and $B$.

- Code traverses the right spines, one node at a time
  - so needs at most $\text{rank}(A) + \text{rank}(B)$ steps
  - Each step needs constant work
- $\text{makeLeftistNode}$ guarantees leftist result
### Proving the Lemma

**Claim**
If a heap has rank $r$, it contains at least $2^r - 1$ entries.

- $n(r) \equiv$ nodes in the smallest heap of rank $r$
  - Monotone: if $r' \geq r$, then $n(r') \geq n(r)$
  - $n(0) = 0$
- $\text{rank}(L(x)) \geq \text{rank}(R(x)) = r - 1$
  
  \[
  n(r) = 1 + n(\text{rank}(L(x))) + n(\text{rank}(R(x))) \\
  \geq 1 + n(r - 1) + n(r - 1) = 1 + 2 \cdot n(r - 1).
  \]
- $n(r) \geq 2^r - 1$
Proving the Lemma

- Apply the claim
- Suppose leftist heap of $n$ nodes has rank $r$
- $n \geq n(r) \geq 2^r - 1$
- $2^r \leq n + 1 \Rightarrow r \leq \log_2(n + 1)$
- Rank of a leftist node of $n$ nodes is at most $\log_2(n + 1)$
# Summary of Priority Queues

<table>
<thead>
<tr>
<th>Implementation</th>
<th>insert</th>
<th>findMin</th>
<th>deleteMin</th>
<th>meld</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Unsorted) Sequence</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>Sorted Sequence</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>Balanced Tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(m \log (1 + \frac{n}{m}))$</td>
</tr>
<tr>
<td>Leftist Heap</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log m + \log n)$</td>
</tr>
</tbody>
</table>