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 that 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 $\mathbb{U}$, and $\mathbb{S}$ represents the set of all subsets of $\mathbb{U}$.
- With the following operations

all operations supported by the Set ADT, and

$$
\begin{array}{lcll}
\texttt{last}(S) & : & \mathbb{S} \to \mathbb{U} & = \max\ S \\
\texttt{first}(S) & : & \mathbb{S} \to \mathbb{U} & = \min\ S \\
\texttt{split}(S,k) & : & \mathbb{S} \times \mathbb{U} \to \mathbb{S} & = (\{k' \in S \mid k' < k\}, k \stackrel{?}{\in} S, \\
 & & \quad \times \mathit{bool} \times \mathbb{S} & \qquad \{k' \in S \mid k' > k\}) \\
\texttt{join}(S_1, S_2) & : & \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = S_1 \cup S_2, \text{assuming} \\
 & & & \qquad \max S_1 < \min S_2 \\
\texttt{getRange}(S, k_1, k_2) & : & \mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S} & = \{k \in S \mid k_1 \le k \le k_2\}
\end{array}
$$

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.

Improvising Bingle

- 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.
- `rank(S, k)`: How many elements in S are less than k?
- `select(S, i)`: Which element in S has rank i?
- `splitIdx(S,i)`: Split S into two sets: first i keys and the remaining $n - i$ keys.

Augmenting Balanced Trees

$$\texttt{rank}(S,k) : \mathbb{S} \times \mathbb{U} \to \mathit{int} = |\{k' \in S \mid k' < k\}|$$

$$\texttt{select}(S,i) : \mathbb{S} \times \mathit{int} \to \mathbb{U} = k \text{ such that } |\{k' \in S \mid k' < k\}| = i$$

$$\texttt{splitIdx}(S,i) : \mathbb{S} \times \mathit{int} \to \mathbb{S} \times \mathbb{S} = (\{k \in S \mid k < \texttt{select}(S,i)\}, \{k \in S \mid k \geq \texttt{select}(S,i)\})$$

- Without additional information stored with the keys, these operations would take $\theta(|S|)$ work.

Augmenting Balanced Trees

- Let $S = \{1, 2, 3, 4, 5, 6\}$
- `rank(S, 4)` $= |\{1, 2, 3\}| = 3$
- `select(S, 3)` $= 4$ since `rank(S, 4)` $= 3$
- `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!

Select with Augmented Trees

```
1  fun select(T, i) =
2    case expose(T) of
3      NONE ⇒ raise Range
4    | SOME(L, R, k) ⇒
5      case compare(i, |L|) of
6        LESS ⇒ select(L, i)
7      | EQUAL ⇒ k
8      | GREATER ⇒ select(R, i − |L| − 1)
```

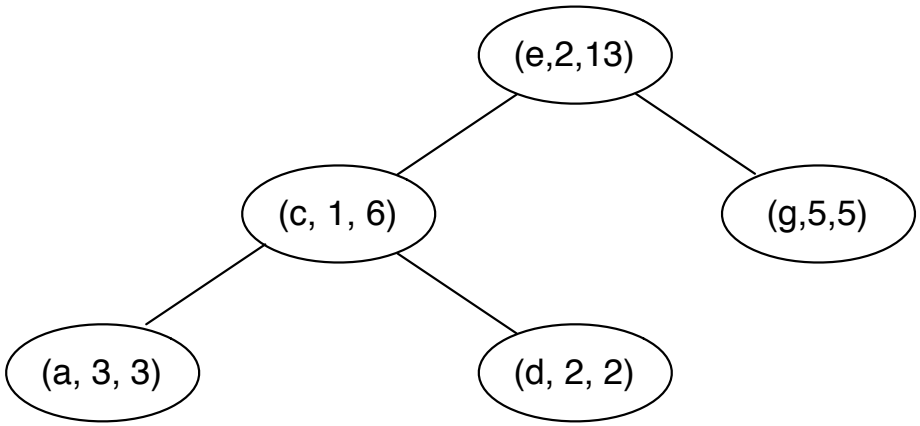
RANK AND SPLITIDX

- `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))

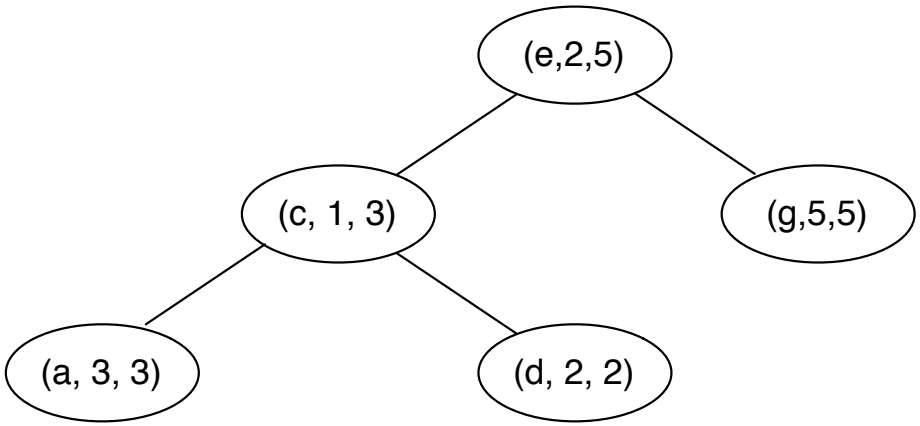
Ordered Tables with Reduced Values

- 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 I_f
  - All operations on ordered tables are supported, and
  - $$\texttt{reduceVal}(A) : \mathbb{T} \to v = \texttt{reduce}\ f\ I_f\ \ A$$
  - 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 is $+$



f is max

Implementation

```
1  datatype Treap = Leaf | Node of (Treap × Treap
2                                   × key × data × data)

3  fun reduceVal(T) =
4     case T of
5        Leaf ⇒ Reduce.I
6      | Node(_, _, _, _, r) ⇒ r

7  fun makeNode(L, R, k, v) =
8     Node(L, R, k, v, Reduce.f(reduceVal(L),
9                               Reduce.f(v, reduceVal(R))))
```

Implementation

```
1  fun join(T1, T2) =
2    case (T1, T2) of
3      (Leaf, _) ⇒ T2
4    | (_, Leaf) ⇒ T1
5    | (Node(L1, R1, k1, v1, s1), Node(L2, R2, k2, v2, s2)) ⇒
6        if (priority(k1) > priority(k2)) then
7          makeNode(L1, join(R1, T2), k1, v1)
8        else
9          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 $O(logn)$ 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 $O(logn)$ work

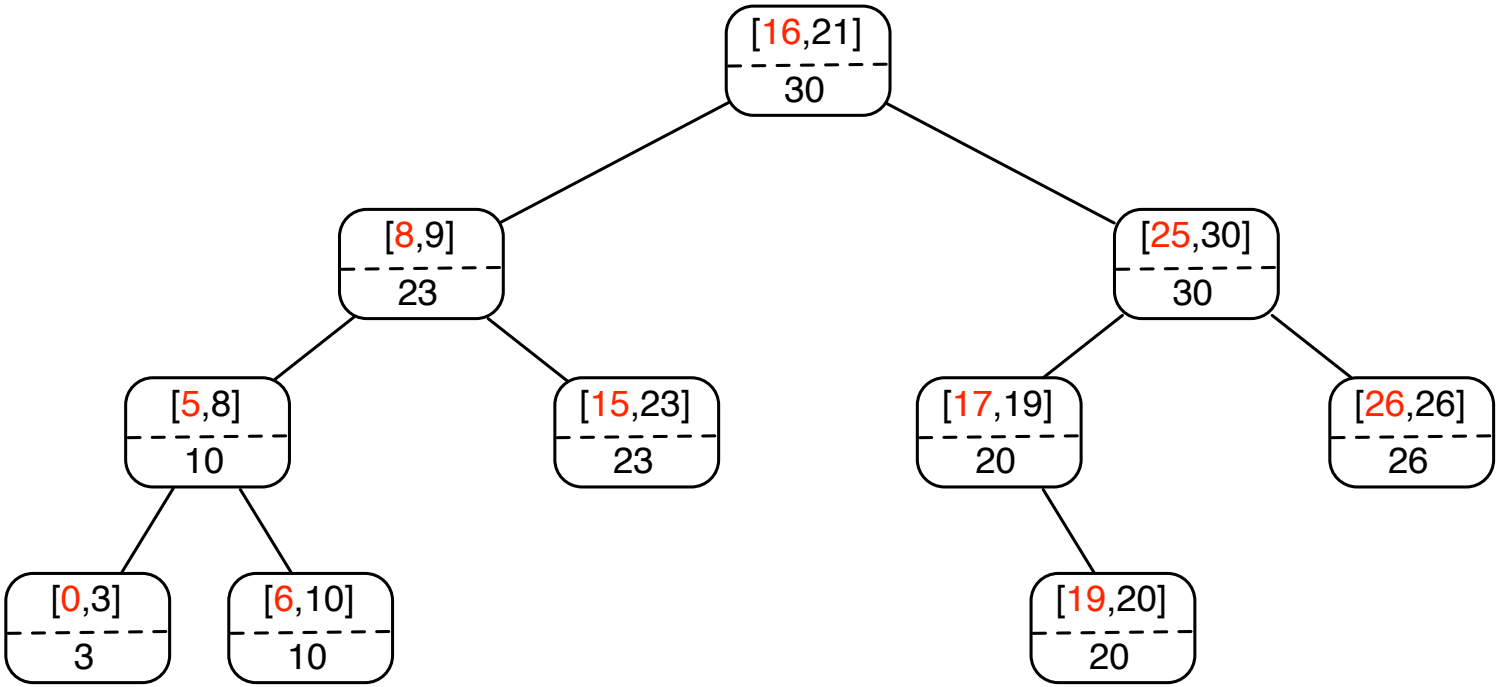
Example Application- Interval Trees

- 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:

$\texttt{insert}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*insert interval I into table A*
$\texttt{delete}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*delete interval I from table A*
$\texttt{count}(A, x)$	:	$\mathbb{T} \times \texttt{real} \to \texttt{int}$	*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
2                                      × real × real × real)

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
```