15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 20

Search Trees II: Treaps

Synopsis

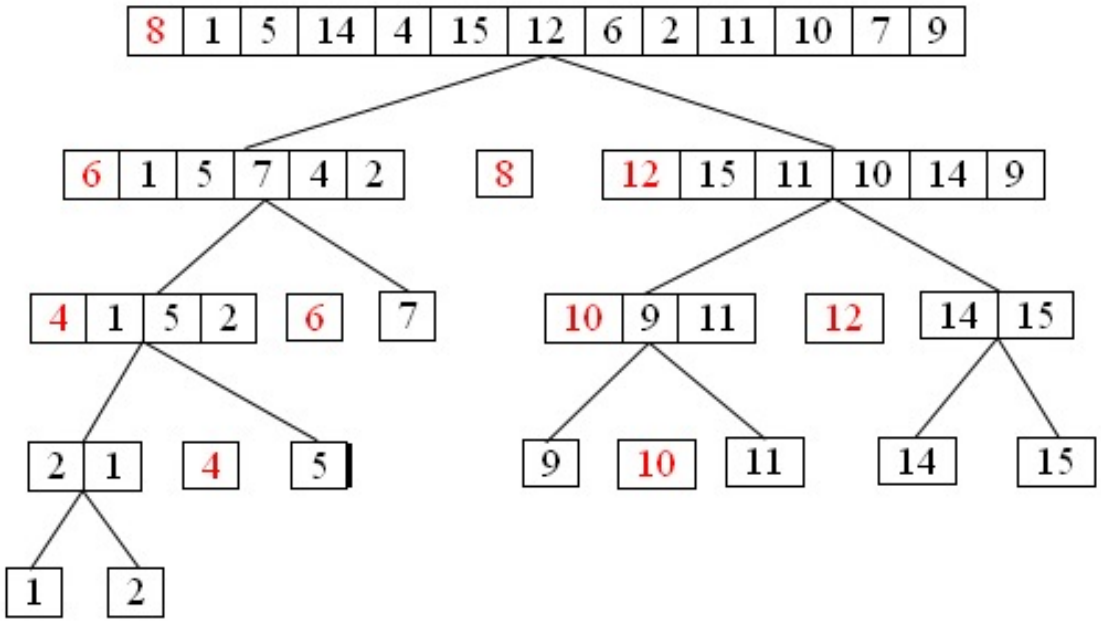
- Overview of Binary Search Trees
- Relationship between Quicksort and BSTs
- Treaps
- Expected Depth of a Treap

BST Overview

- 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.



Sequence to BST

```
1   fun qs_tree(S) =
2     if |S| = 0 then LEAF
3     else let
4           p = pick a pivot from S
5           S1 = ⟨ s ∈ S | s < p ⟩
6           S3 = ⟨ s ∈ S | s > p ⟩
7           (TL, TR) = ( qs_tree(S1) || qs_tree(S3) )
8       in
9         NODE(TL, p, TR)
10      end
```

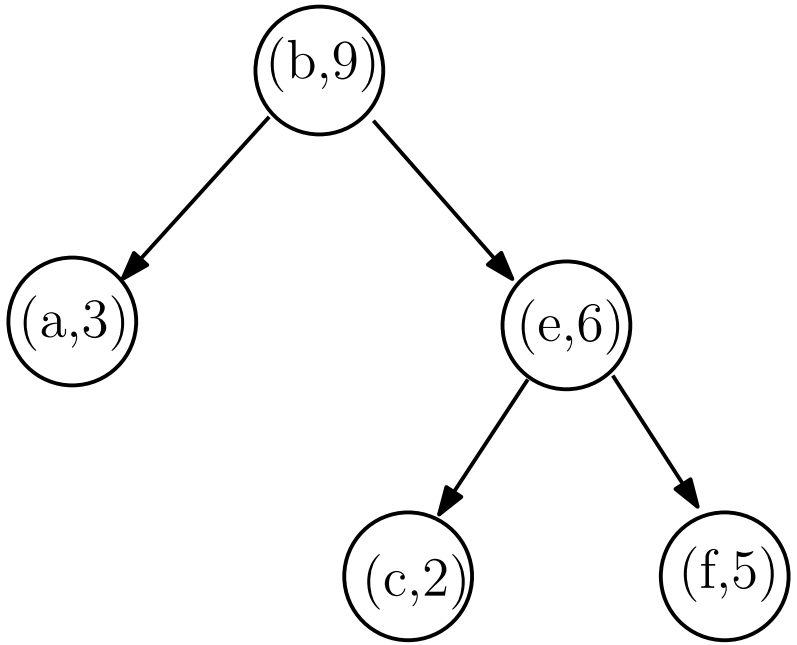
- Unlike Quicksort, we do not know what elements will be in the tree, when we start.
  - We can not select a (n) (future?) element to be the root.

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 satisfy 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)

Treaps

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

```
1  fun search T k =
2  let (_, v, _) = split(T, k)
3  in v
4  end
```

Basic BST Operations - Insert

```
1  fun insert T (k, v) =
2  let   (L, v', R) = split(T, k)
3  in  join(L, SOME(k, v), R)
4  end
```

Basic BST Operations - Delete

```
1  fun delete T k =
2  let   (L, _, R) = split(T, k)
3  in  join(L, NONE, R)
4  end
```

- So if `split` and `join` are implemented the other more useful operations are covered.

Join and Split

- $\mathtt{split}(T, k) : \mathtt{BST} \times \mathtt{key} \rightarrow \mathtt{BST} \times (\mathtt{data\ option}) \times \mathtt{BST}$

- $\mathtt{split}$ 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 $\mathtt{split}$ returns $\mathtt{SOME}(d)$
- Otherwise it returns $\mathtt{NONE}$.

Join and Split

- $join(L, m, R) : BST \times (key \times data)\ option \times BST \to BST$

- 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.

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

```
datatype BST = Leaf |
               Node of (BST * BST * key * data)
```

```
 1  fun split(T, k) =
 2    case T of
 3      Leaf  ⇒ (Leaf, NONE, Leaf)
 4    | Node(L, R, k′, v) ⇒
 5        case compare(k, k′) of
 6          LESS ⇒
 7            let  (L′, r, R′) = split(L, k)
 8            in  (L′, r, Node(R′, R, k′, v)) end
 9          EQUAL ⇒ (L, SOME(v), R)
10          GREATER ⇒
11            let  (L′, r, R′) = split(R, k)
12            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 of the new node as the root.

```
1  fun join(T1, m, T2) =
2    case m of
3      SOME(k, v) ⇒ Node(T1, T2, k, v)
4    | NONE ⇒
5      case T1 of
6        Leaf ⇒ T2
7      | Node(L, R, k, v) ⇒ Node(L, join(R, NONE, T2), k, v)
```

Join on Treaps

```
1   fun join(T1, m, T2) =
2   let
3     fun singleton(k, v) = Node(Leaf, Leaf, k, v)
4     fun join'(T1, T2) =
5       case (T1, T2) of
6         (Leaf, _) ⇒ T2
7       | (_, Leaf) ⇒ T1
8       | (Node(L1, R1, k1, v1), Node(L2, R2, k2, v2)) ⇒
9           if (priority(k1) > priority(k2)) then
10            Node(L1, join'(R1, T2), k1, v1)
11          else
12            Node(join'(T1, L2), R2, k2, v2)
13  in
14    case m of
15      NONE ⇒ join'(T1, T2))
16    | SOME(k, v) ⇒ join'(T1, join'(singleton(k, v), T2))
17  end
```

Expected Depth of a Key

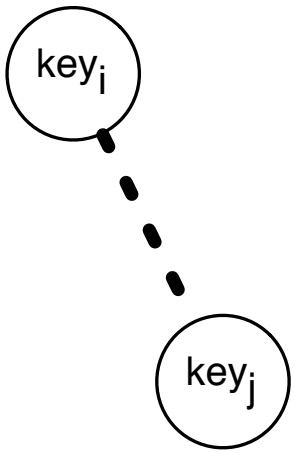
- Cost of split and join depend on the expected depth of a key.
- Given a set of keys K and priorities $p : \texttt{key} \to \mathit{int}$
  - Priorities are unique!
- Consider the elements of the tree laid out in order
  - $key_i < key_j \Rightarrow \cdots, key_i, \cdots, key_j, \cdots$
  - $key_j < key_i \Rightarrow \cdots, key_j, \cdots, key_i, \cdots$
- A_i^j is an indicator variable:
  - $A_i^j = 1$ if key_j is an ancestor of key_i in the treap.
  - $A_i^j = 0$ otherwise.

Expected Depth of a Key

$$\ldots, key_i, \ldots, key_j, \ldots$$

$$key_i < key_j$$

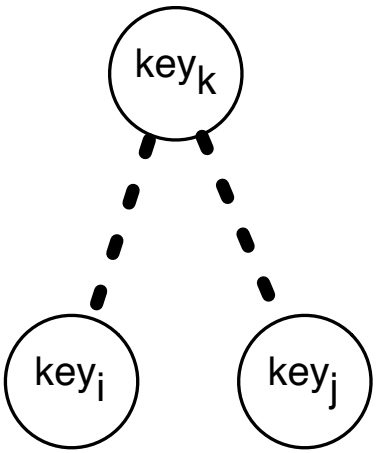
$p_i = \max(p_i, \ldots, p_j)$



$A_i^j = 0$

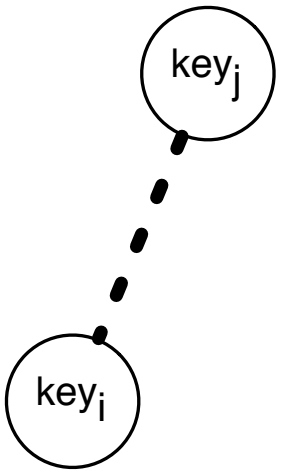
$p_k = \max(p_i, \ldots, p_j)$

$i<k<j$



$A_i^j = 0$

$p_j = \max(p_i, \ldots, p_j)$



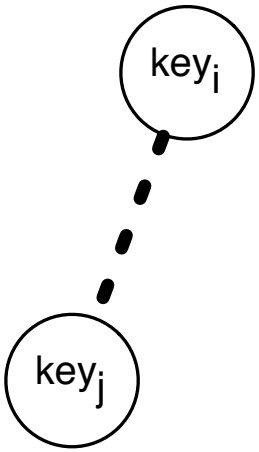
$A_i^j = 1$

Expected Depth of a Key

$$\ldots, key_j, \ldots, key_i, \ldots$$

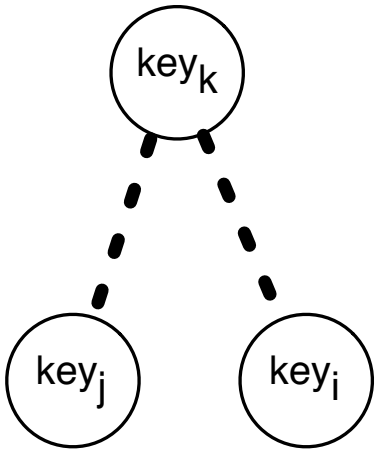
$$key_i > key_j$$

$p_i = \max(p_j, \ldots, p_i)$



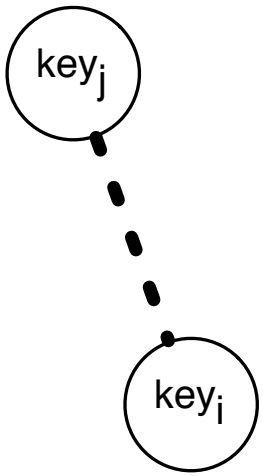
$A_i^j = 0$

$p_k = \max(p_j, \ldots, p_i)$, $i<k<j$



$A_i^j = 0$

$p_j = \max(p_j, \ldots, p_i)$



$A_i^j = 1$

Expected Depth of a Key

$$\mathbf{E}\left[\text{depth of } i \text{ in } T\right] = \mathbf{E}\left[\sum_{j=1, j\neq i}^{n} A_i^j\right] = \sum_{j=1, j\neq i}^{n} \mathbf{E}\left[A_i^j\right].$$

$$\mathbf{E}\left[A_i^j\right] = \frac{1}{|j-i|+1} \quad \text{(Why?)}$$

Expected Depth of a Key

$$
\begin{aligned}
\mathbf{E}[\text{depth of } i \text{ in } T] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\
(\text{Split} \mid\ \mid \Rightarrow) &= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
(\text{Change variables } \Rightarrow) &= \sum_{k=2}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&= H_i - 1 + H_{n-i+1} - 1 \\
(\ln n < H_n < \ln n + 1 \Rightarrow) &< \ln i + \ln(n-i+1) \\
&= O(\log n)
\end{aligned}
$$

- Relative (sorted) position of a key determines expected depth in treap.

Cost of Split and Join

Theorem

For treaps

- `join`(T_1, m, T_2) returning T
- `split`$(T, (k, v))$

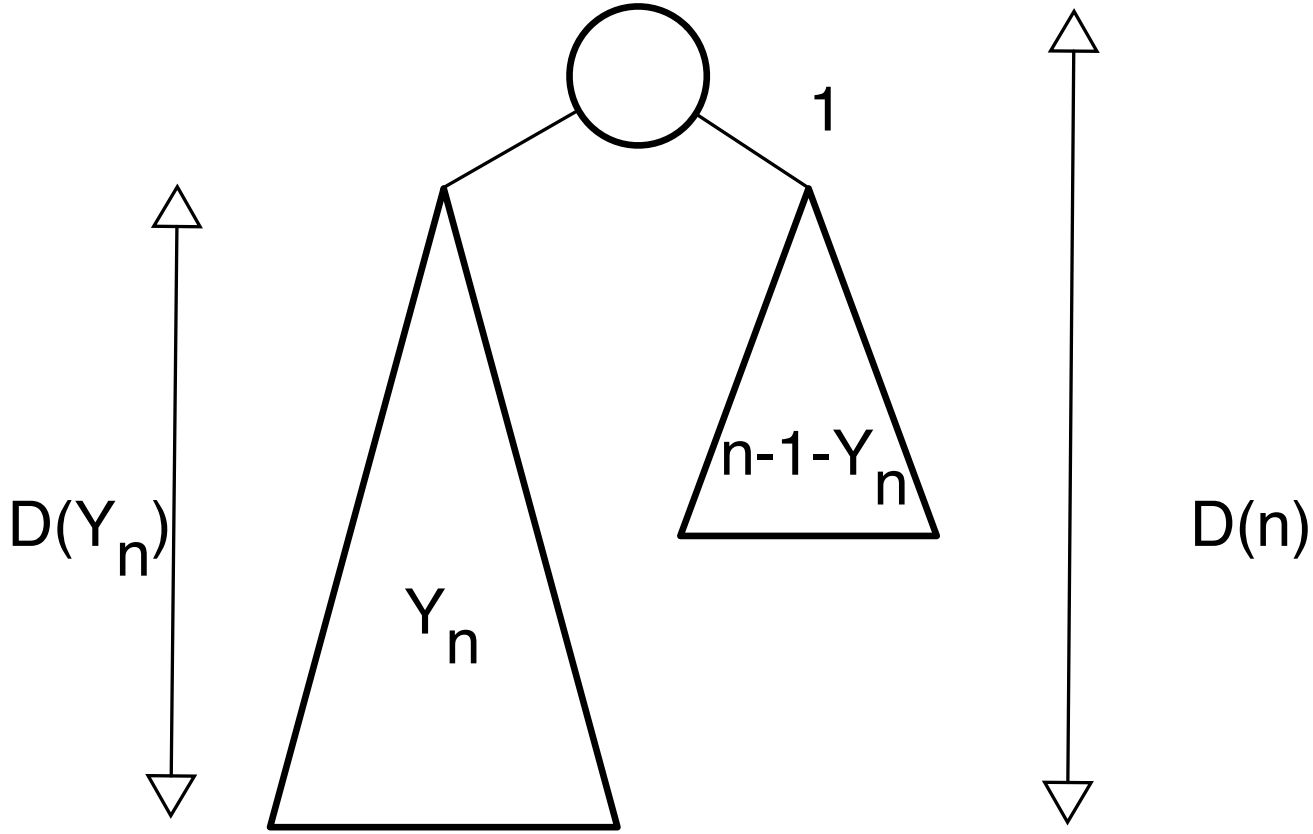
have $O(\log |T|)$ expected work and span.

- 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 $\mathbf{E}[\max_i\{A_i\}] \neq \max_i\{\mathbf{E}[A_i]\}$!
- It turns out this is almost the same problem as the expected span of the quicksort.

Expected Max Depth of a Treap



- Y_n is the size of the larger partition.
- $D(n) = D(Y_n) + 1 \Rightarrow D(n) \in O(\log n)$