15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 20

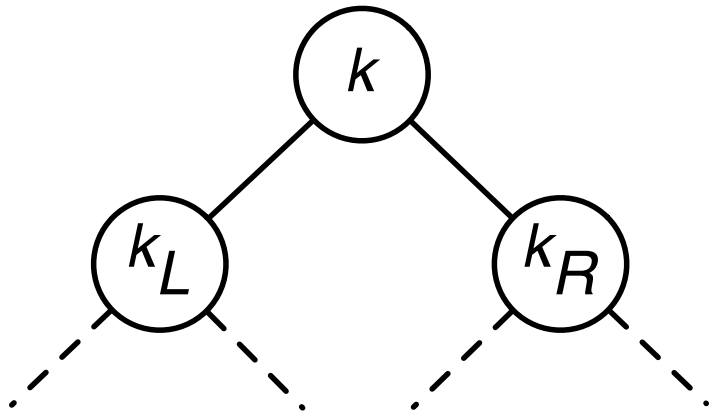
Search Trees I: BSTs Split, Join, and Union

Synopsis

- Binary Search Trees
- Basic Structural Operations on BSTs
- Basic Operations on BSTs
- Concrete Implementations
- Cost Analysis

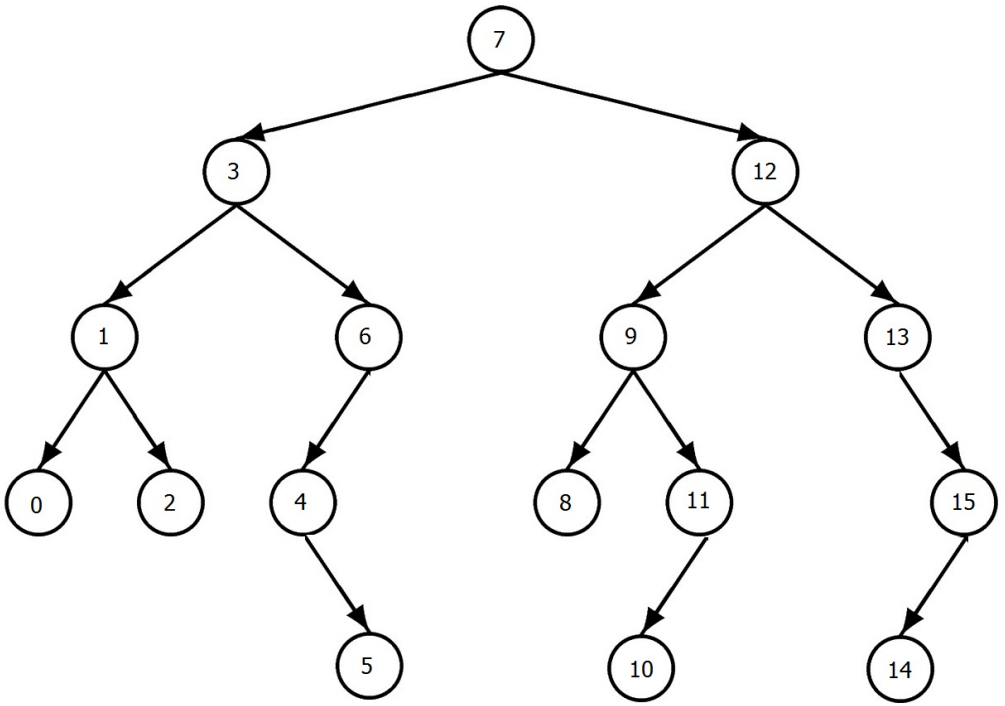
Binary Trees

- Trees where each node has at most 2 children each of which is a binary tree.
  - Left child / Left subtree
  - Right child / Right subtree

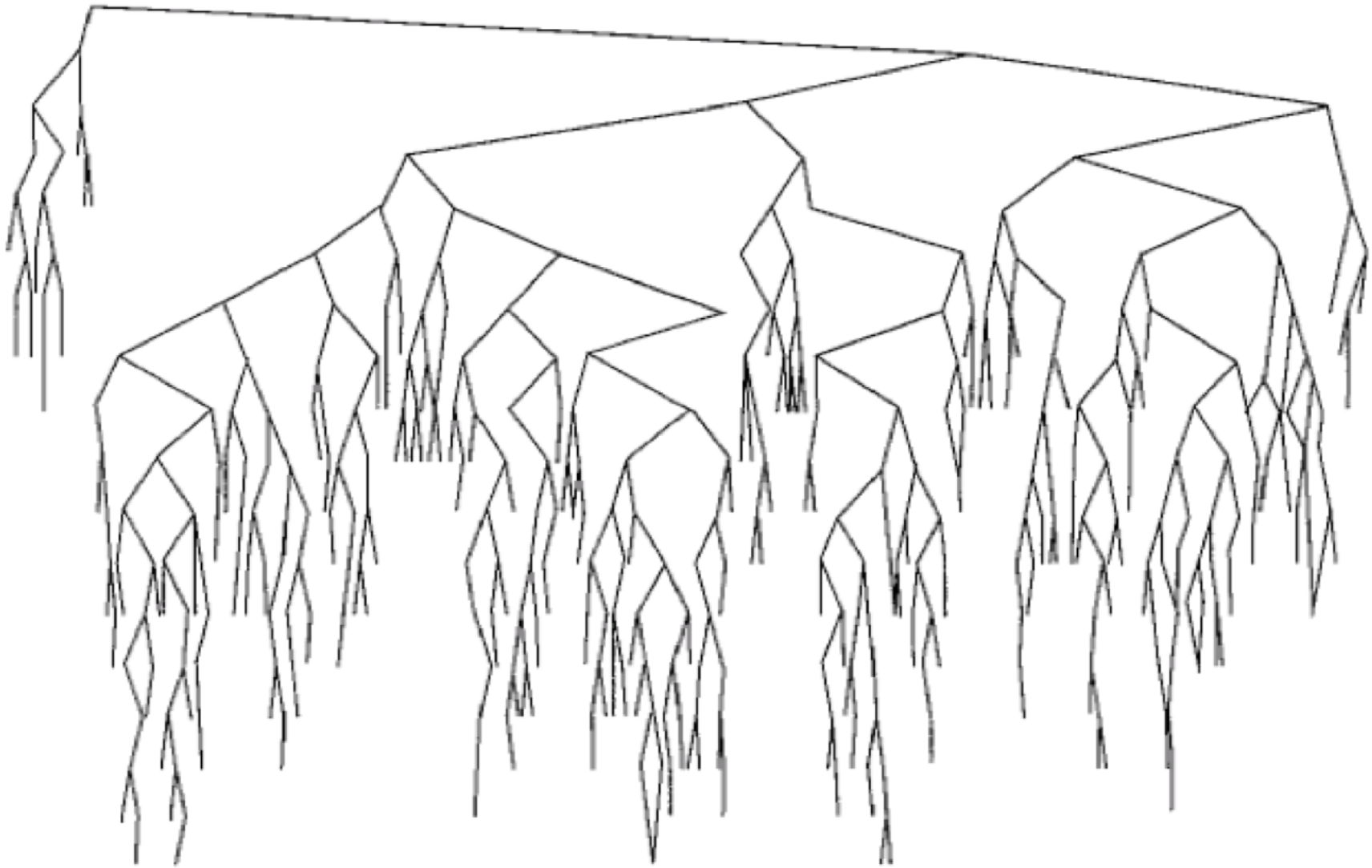


Binary Search Trees

- 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 $k_R > k$



Things can get pretty bad



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.

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

Basic BST Operations

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

- `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 $\texttt{SOME}(d)$
- Otherwise it returns `NONE`.

Basic BST Operations

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

- 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.
- We also assume both split and join maintain balance.

Basic BST Operations

- $\texttt{expose}(T) : \texttt{BST} \rightarrow (\texttt{BST} \times \texttt{BST} \times \texttt{key} \times \texttt{data})\ \texttt{option}$

- Returns the components if BST T is not empty.

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
```

Concrete Implementations: Split

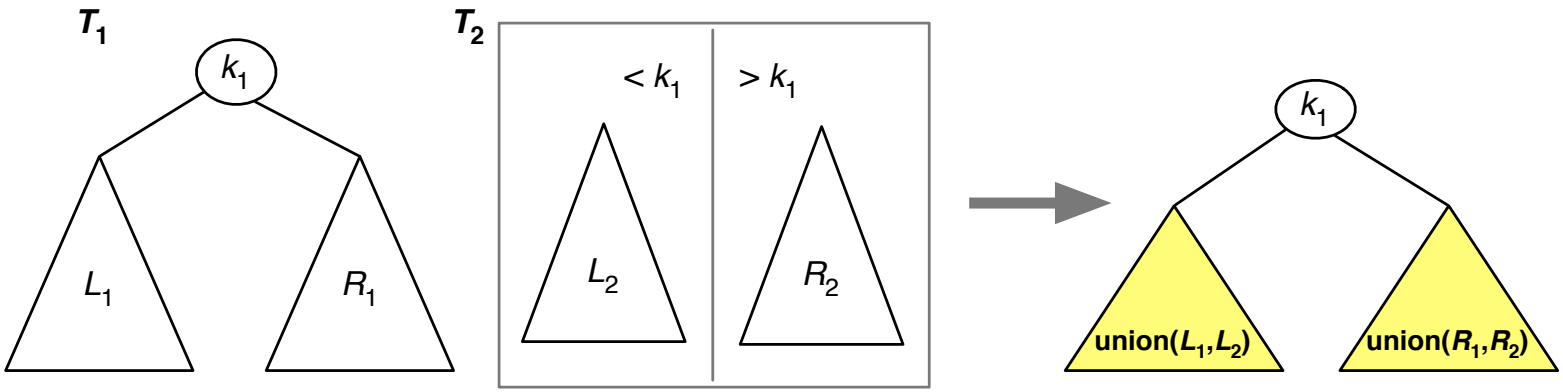
```
datatype BST = Leaf |
               Node of (BST * BST * key * data)
```

```
 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
```

Concrete Implementations: Join

```
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))
```

Concrete Implementations: Union



- 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 $L_u = \texttt{union}(L_1, L_2)$ and $R_u = \texttt{union}(R_1, R_2)$.
- Now $\texttt{join}(L_u, k_1, R_u)$.

Concrete Implementations: Union

```
1  fun union(T1, T2) =
2     case expose(T1) of
3        NONE ⇒ T2
4      | SOME(L1, R1, k1, v1) ⇒
5        let   (L2, v2, R2) = split(T2, k1)
6              (L, R) = union(L1, L2) || union(R1, R2)
7        in join(L, SOME(k1, v1), R)
8        end
```

- Returns the value from T_1 if a key appears in both trees.

Analysis of Union

```
1  fun union(T1, T2) =
2     case expose(T1) of
3        NONE ⇒ T2
4      | SOME(L1, R1, k1, v1) ⇒
5        let   (L2, v2, R2) = split(T2, k1)
6              (L, R) = union(L1, L2) || union(R1, R2)
7        in  join(L,  SOME(k1, v1),  R)
8        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

$$W(|T_1|, |T_2|) = \underbrace{2W(|T_1|/2, |T_2|/2)}_{\text{recursive union calls}} + \underbrace{O(\log(|T_1| + |T_2|))}_{\text{split and join}},$$

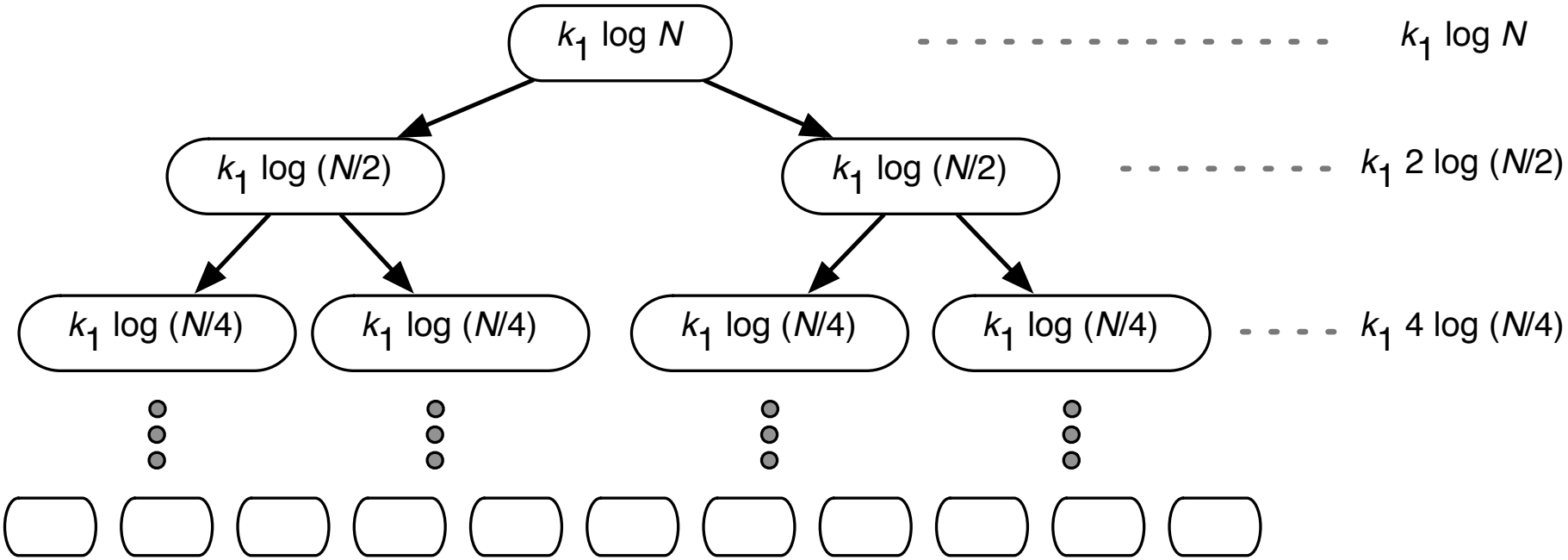
and

$$W(1, |T_2|) = O(\log(1 + |T_2|)).$$

- When $|T_1| = 1$, `expose` give us two empty subtrees L_1 and R_1
- `union`(L_1, L_2) returns L_2, `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 $n = |T_2|$ and $N = n + m$



Bottom level: each costs log (1+ (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 $= O(m \log(1 + \frac{n}{m}))$