PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 12
Binary Search Trees

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

BST is important?

- Searching is one of the most important operations in computer science
- What data structure can we use for searching?
  - Sequence?
  - Search tree?
  - Binary search tree?
  - Balanced binary search tree?

Difference?
Which one is better?

SYNOPSIS

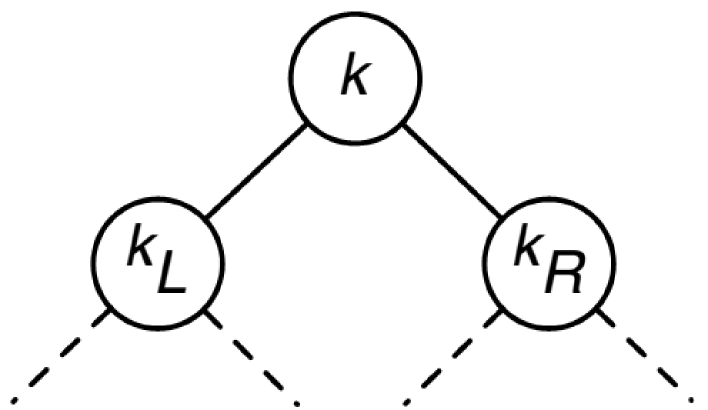
- Preliminaries
- The BST Abstract Data Type
- Implementation via Balancing
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

華中科技大學

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

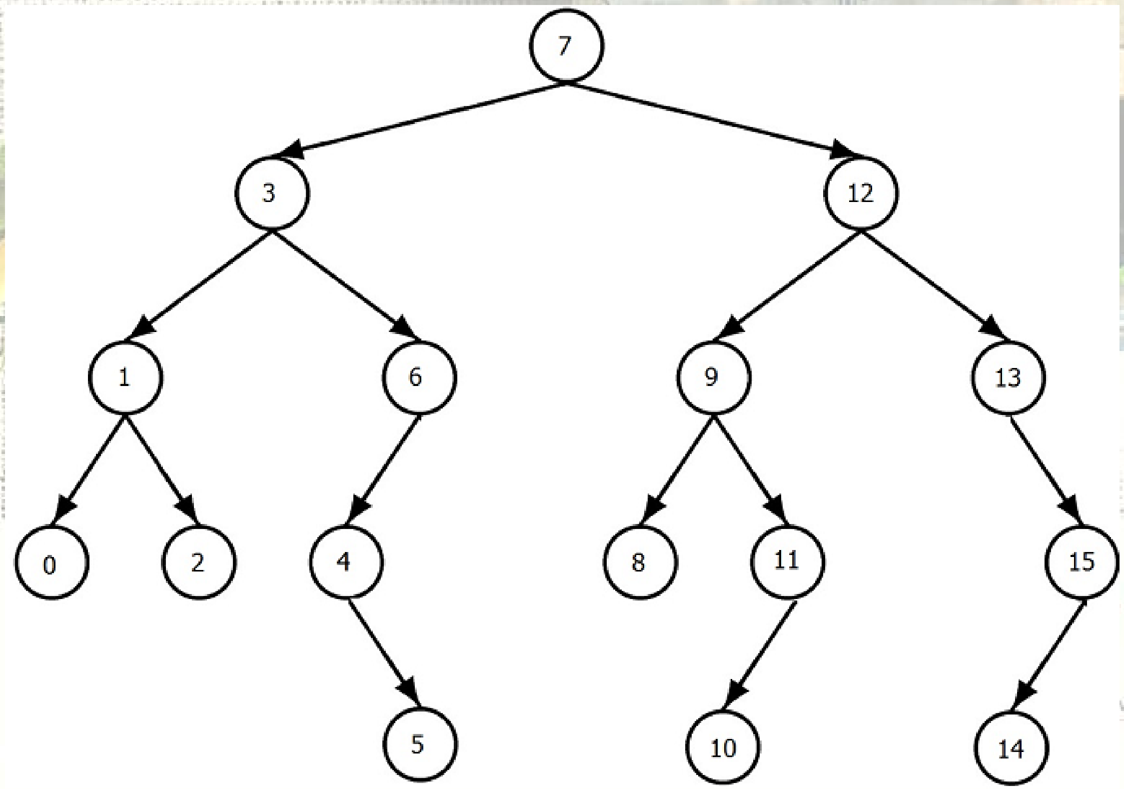
Binary Trees

- **A rooted tree** is a tree with a distinguished root node that can be used to access all other nodes
- **A full binary tree** is an ordered rooted tree in which every node has **exactly** two children
  - ➢ 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 $k_R>k$

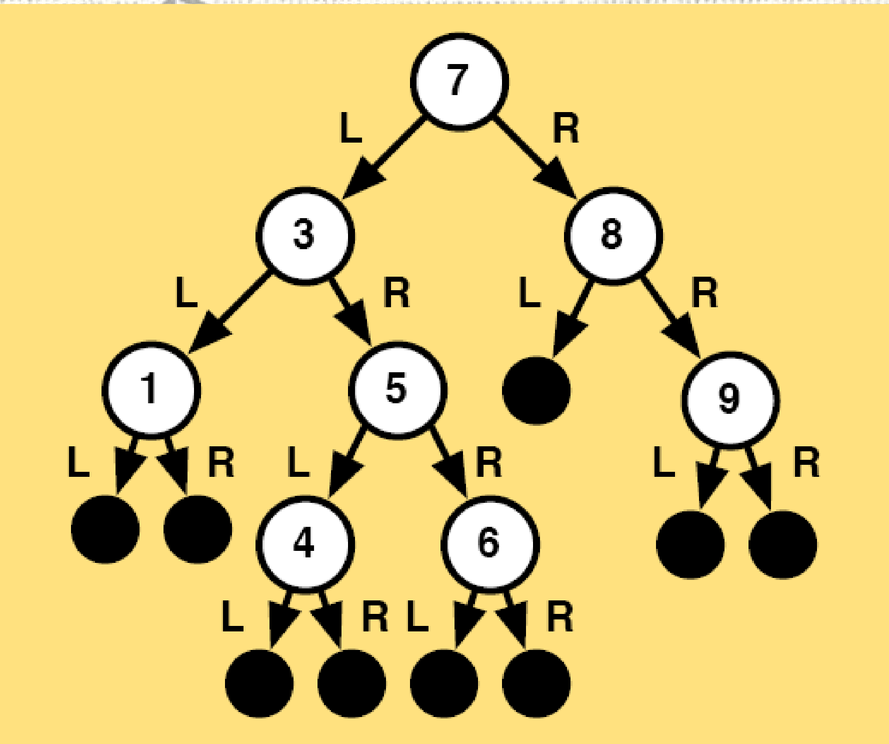


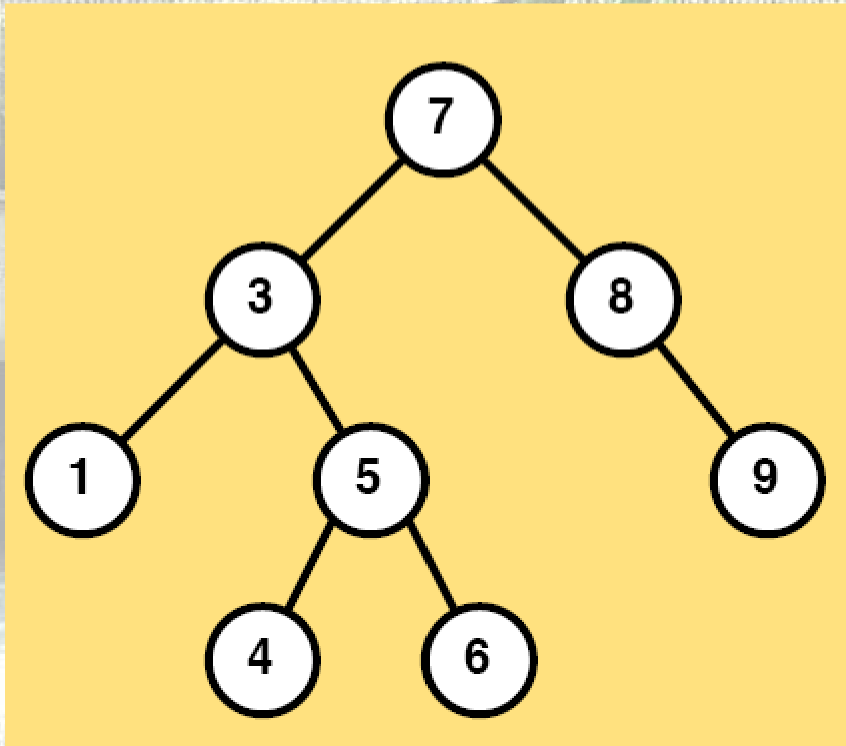
Binary Search Trees

- **A binary search tree (BST) over a totally ordered set S is a full binary tree that satisfies the following conditions.**
  - **1. There is a one-to-one mapping k(v) from internal tree nodes to elements in S**
  - **2. for every u in the left subtree of v, k(u) < k(v)**
  - **3. for every u in the right subtree of v, k(u) > k(v)**

Can you write a binary search tree over the set {1, 4, 7, 9, 13, 17} ?

Binary Search Trees





SYNOPSIS

- Preliminaries
- The BST Abstract Data Type
- Implementation via Balancing
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

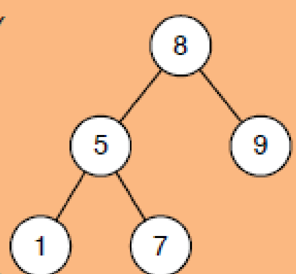
EMPTY, SINGLETON

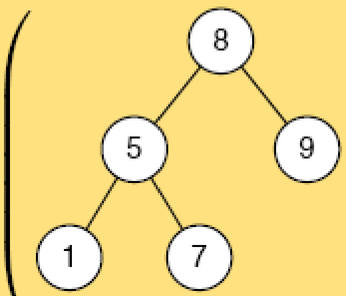
- $\mathbb{K}$ & $\mathbb{T}$
  - For a universe of totally ordered keys $\mathbb{K}$, the BST ADT consists of a type $\mathbb{T}$ representing a power set of keys
- $[\![T]\!]$
  - for a tree T, $[\![T]\!]$ denotes the set of keys in the tree

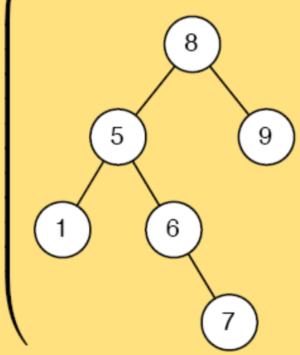
$$
\begin{array}{lcl}
\mathtt{empty} & : & \mathbb{T} \\
\mathtt{empty} & = & T \textit{ where } [\![T]\!] = \emptyset \\
\mathtt{singleton} & : & \mathbb{K} \to \mathbb{T} \\
\mathtt{singleton}(k) & = & T \textit{ where } [\![T]\!] = \{k\} \,.
\end{array}
$$

FIND, INSERT, DELETE

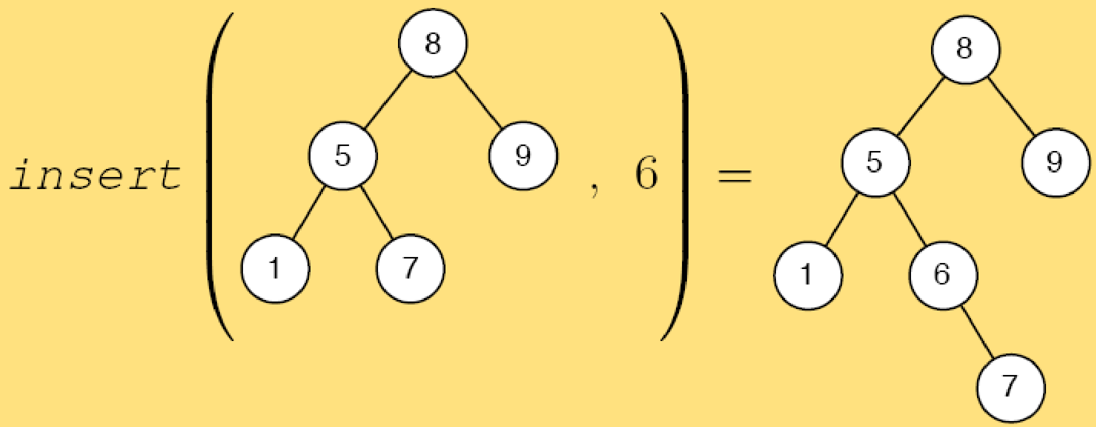
$$
\begin{aligned}
\mathtt{find} &: (\mathbb{T} \times \mathbb{K}) \to \mathbb{B} \\
\mathtt{find}(T, k) &= \mathtt{true}\ \textit{if and only if}\ k \in [\![T]\!] \\[1ex]
\mathtt{insert} &: (\mathbb{T} \times \mathbb{K}) \to \mathbb{T} \\
\mathtt{insert}(T, k) &= T'\ \textit{where}\ [\![T']\!] = [\![T]\!] \cup \{k\} \\[1ex]
\mathtt{delete} &: (\mathbb{T} \times \mathbb{K}) \to \mathbb{T} \\
\mathtt{delete}(T, k) &= T'\ \textit{where}\ [\![T']\!] = [\![T]\!] \setminus \{k\}\,.
\end{aligned}
$$

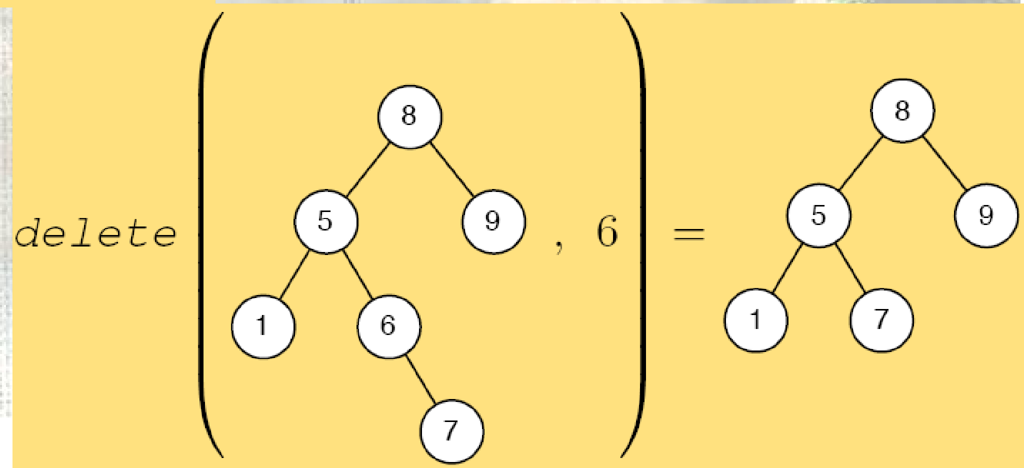
$\mathtt{find}\Bigg($  $,\ 5 \Bigg) = \boxed{?}$

$\mathtt{insert}\Bigg($  $,\ 6 \Bigg) = \boxed{?}$

$\mathtt{delete}\Bigg($  $,\ 6 \Bigg) = \boxed{?}$

FIND, INSERT, DELETE





12-1 Insert
Design an algorithm for inserting a given key into a BST.

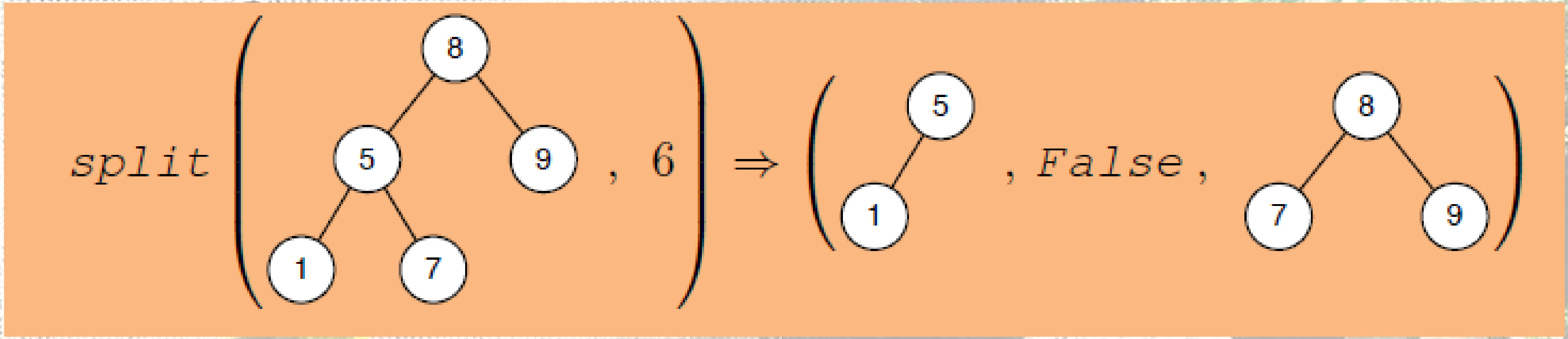
12-2 Delete
Design an algorithm for deleting a given key from a tree.

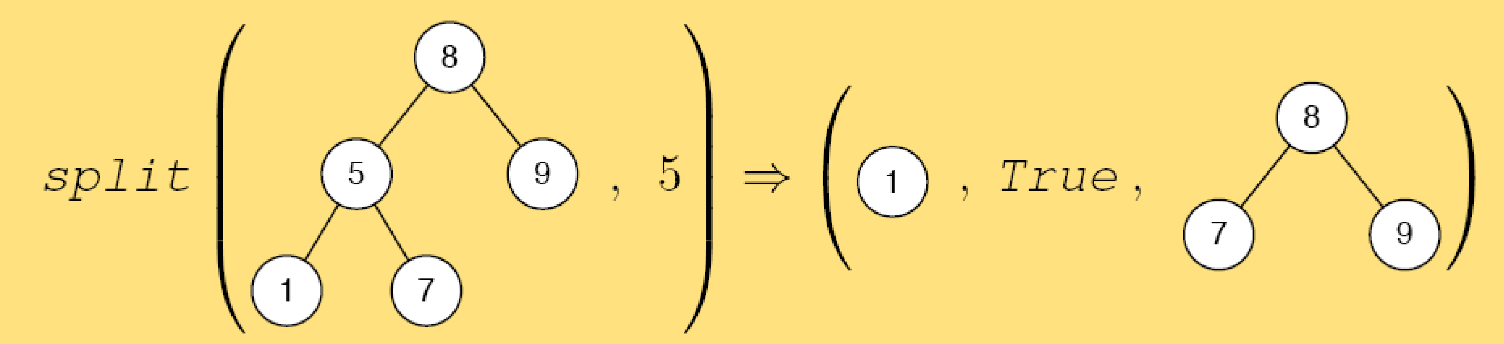
UNION, INTERSECTION, DIFF

$$
\begin{aligned}
\texttt{union} &: (\mathbb{T} \times \mathbb{T}) \to \mathbb{T} \\
\texttt{union}(T_1, T_2) &= T \textit{ where } [\![T]\!] = [\![T_1]\!] \cup [\![T_2]\!] \\[1em]
\texttt{intersection} &: (\mathbb{T} \times \mathbb{K}) \to \mathbb{T} \\
\texttt{intersection}(T_1, T_2) &= T \textit{ where } [\![T]\!] = [\![T_1]\!] \cap [\![T_2]\!] \\[1em]
\texttt{difference} &: (\mathbb{T} \times \mathbb{K}) \to \mathbb{T} \\
\texttt{difference}(T_1, T_2) &= T \textit{ where } [\![T]\!] = [\![T_1]\!] \setminus [\![T_2]\!]
\end{aligned}
$$

SPLIT

split $: \mathbb{T} \to \mathbb{K} \to (\mathbb{T} \times \mathbb{B} \times \mathbb{T})$

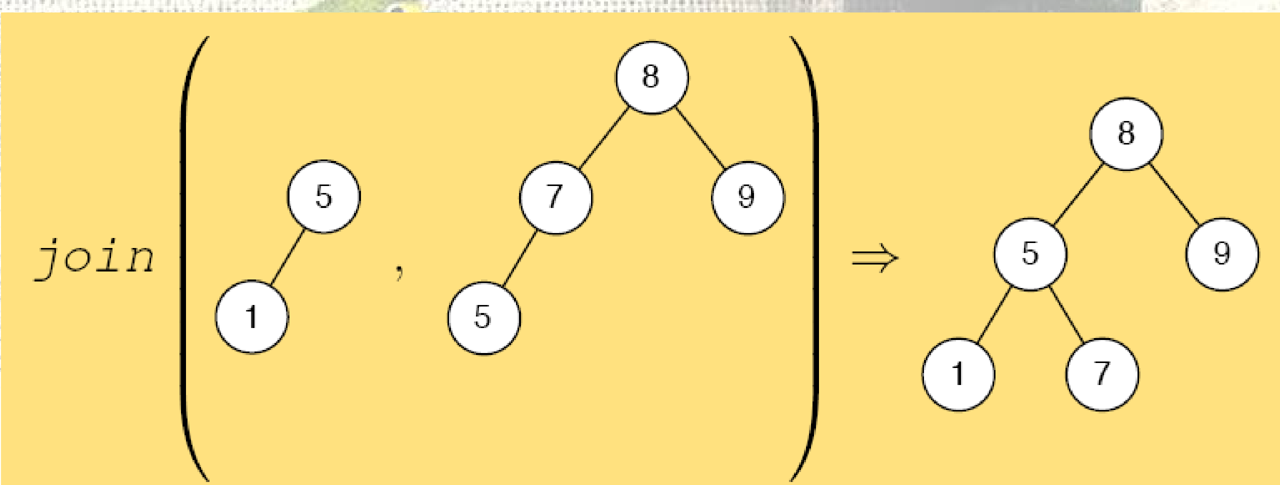




JOIN

- The function join(T_1, T_2) takes two trees T_1 and T_2 such that **all the keys in T_1 are less than the keys in T_2.**

$$
\begin{array}{lcl}
\mathtt{join} & : & (\mathbb{T} \times \mathbb{T}) \rightarrow \mathbb{T} \\
\mathtt{join}(T_1, T_2) & = & T \ \mathit{where}\ [\![T]\!] = [\![T_1]\!] \cup [\![T_2]\!]
\end{array}
$$



SPLIT, JOIN

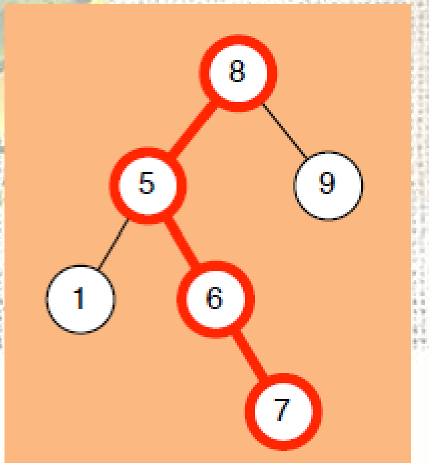
- The exact structure of the trees returned by split can differ from one implementation to another
  - the specification only requires that the resulting trees to be valid BST's and that they contain the keys less than k and greater than k, leaving their structure otherwise unspecified
- The exact structure of the tree returned by join can differ from one implementation to another
  - the specification only requires that the resulting tree is a valid BST and that it contains all the keys in the trees joined.

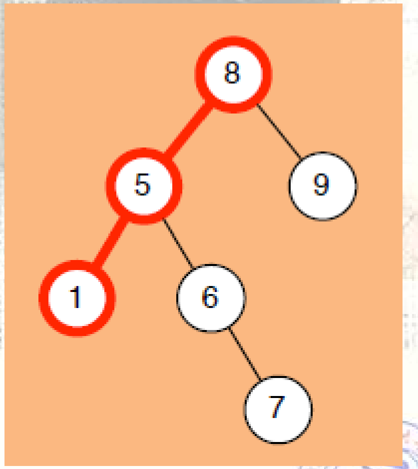
SYNOPSIS

- Preliminaries
- The BST Abstract Data Type
- **Implementation via Balancing**
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

Search tree

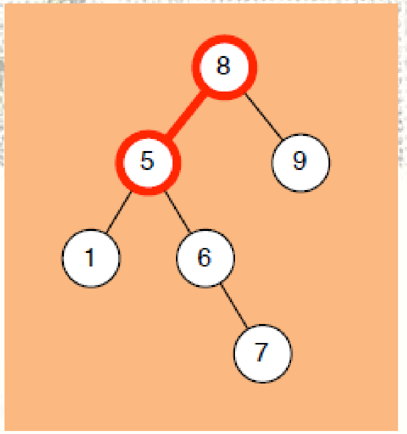
- The main idea behind BST's is to organize the keys such that
  - 1. a specific key can be searched by following a branch in the tree by doing key comparisons along the way
  - Search 7 & 4

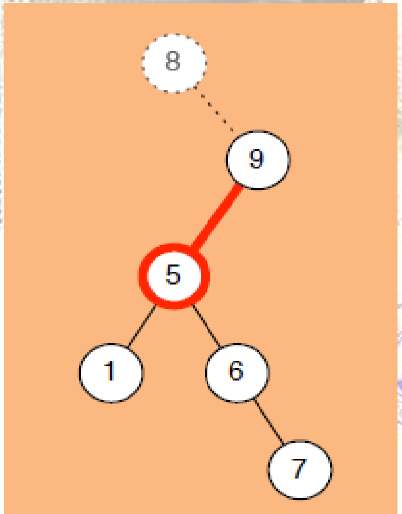




Search tree

- The main idea behind BST's is to organize the keys such that
  - 2. a range of keys in a subtree can be operated on (e.g., moved) by performing constant work
    - each subtree in a binary tree contains the keys that all the keys within a specific range, e.g., all keys less than 8
    - Once we find a range of keys, we can operate on them as a group by handling the root





Balanced tree

- The find & search operations in the BST ADT depend on the paths that we have to walk in the tree
- A binary tree is defined to be perfectly balanced if it has the minimum possible height
  - ➢ Both children are about the same height
  - ➢ Both subtrees are about the same size
- For a binary search tree over a set S, a perfectly balanced tree has height exactly $\lceil \log_2(|S| + 1) \rceil$

BALANCED BST

计算机科学中的树				
二叉树	二叉树	二叉查找树	笛卡尔树	Top tree
	T树			
自平衡二叉查找树	AA树	**AVL树**	红黑树	伸展树
	树堆	节点大小平衡树		
B树	B树	B+树	B*树	Bx树
	UB树	2-3树	2-3-4树	(a,b)-树
	Dancing tree	H树		
Trie	前缀树	后缀树	基数树	
空间划分树	四叉树	八叉树	k-d树	vp-树
	R树	R*树	R+树	X树
	M树	线段树	希尔伯特R树	优先R树
非二叉树	Exponential tree	Fusion tree	区间树	PQ tree
	Range tree	SPQR tree	Van Emde Boas tree	
其他类型	堆	散列树	Finger tree	Metric tree
	Cover tree	BK-tree	Doubly-chained tree	iDistance
	Link-cut tree	树状数组		

BALANCED BST

- There are many balanced BST data structures
  - **AVL trees** are the earliest near-balance BST data structure (1962). It maintains the invariant that the two children of each node differ in height by at most one
  - **Red-Black trees** maintain the invariant that all leaves have a depth that is within a factor of 2 of each other.
    - ✓ The depth invariant is ensured by a scheme of coloring the nodes red and black
  - **Weight balanced** (BB[α]) trees maintain the invariant that the left and right subtrees of a node of size n each have size at least αn for $0 < \alpha \leq 1/2$.
    - ✓ The BB stands for bounded balance, and adjusting gives a tradeoff between search and update costs

BALANCED BST

- There are many balanced BST data structures
  - **Treaps** associate a random priority with every key and maintain the invariant that the keys are stored in heap order with respect to their priorities (treaps is short for tree-heaps)
    - ✓ Treaps guarantee near balance with high-probability
  - **Splay trees** are an amortized data structure that does not guarantee near balance, but instead guarantees that for any sequence of m insert, find and delete operations each does O(log n) amortized work

SYNOPSIS

- Preliminaries
- The BST Abstract Data Type
- Implementation via Balancing
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

A Parametric Implementation

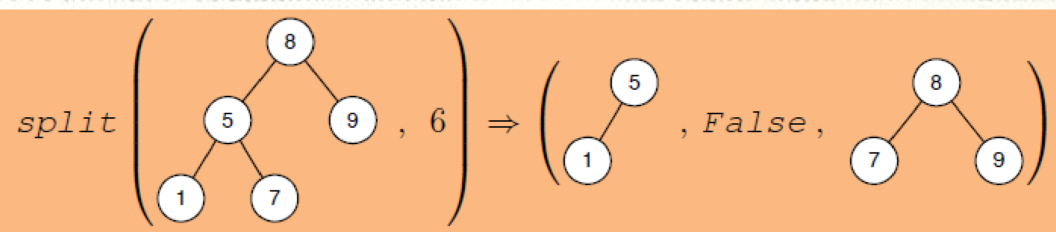
- **Implementing the BST ADT with split and join**

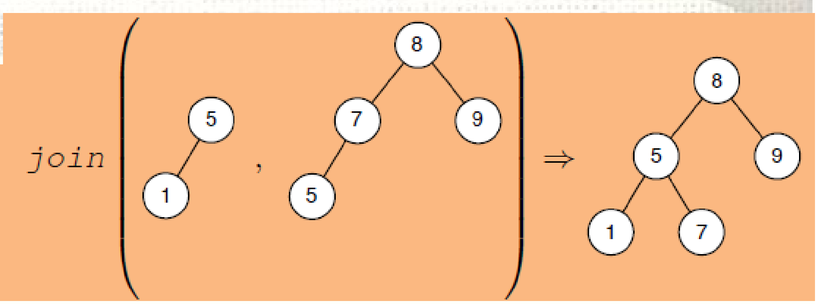
$$\textbf{type}\ \mathbb{T} = \mathit{Leaf} \mid \mathit{Node\ of}\ (\mathbb{T} \times \mathbb{K} \times \mathbb{T})$$

$$\begin{aligned}
&\mathit{split}\ (T, k) = \ldots\ (\star\ \textbf{as}\ \mathit{given}\ \star)\\
&\mathit{join}\ (T_1, T_2) = \ldots\ (\star\ \textbf{as}\ \mathit{given}\ \star)\\
&\mathit{joinM}\ (T_1, k, T_2) = \mathit{join}\ (T_1,\ \mathit{join}\ (\mathit{singleton}\ k,\ T_2))
\end{aligned}$$

$$\mathit{empty} = \mathit{Leaf}$$

$$\mathit{singleton}\ (k) = \mathit{Node}(\mathit{Leaf}, k, \mathit{Leaf})$$





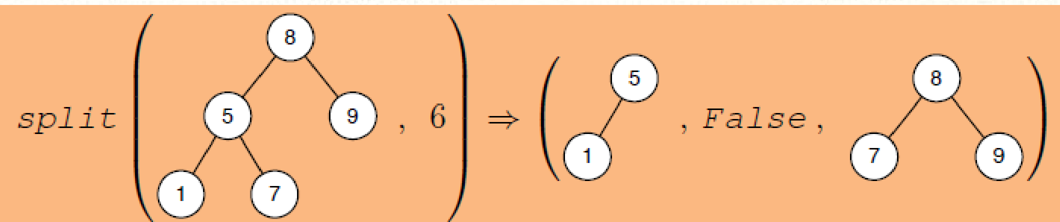
A Parametric Implementation

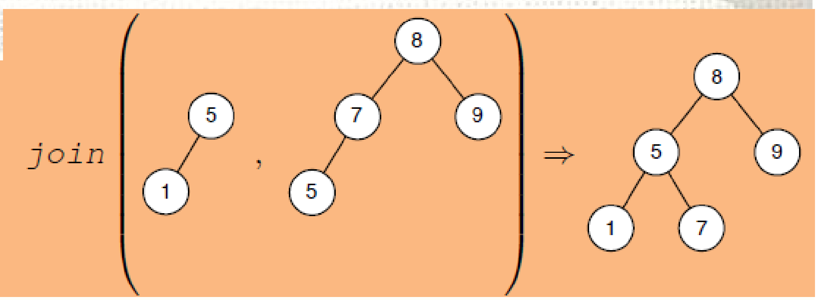
- **Implementing the BST ADT with split and join**
  - **find, insert, delete?**

$$\mathit{find}\ (T,k)\ =\ \textbf{let}\ (\_,v,\_)\ =\ \mathit{split}\ (T,k)\ \ \textbf{in}\ \ v\ \ \textbf{end}$$

$$\mathit{insert}\ (T,k)\ =\ \textbf{let}\ (L,\_,R)\ =\ \mathit{split}\ (T,k)\ \textbf{in}\ \mathit{joinM}\ (L,k,R)\ \textbf{end}$$

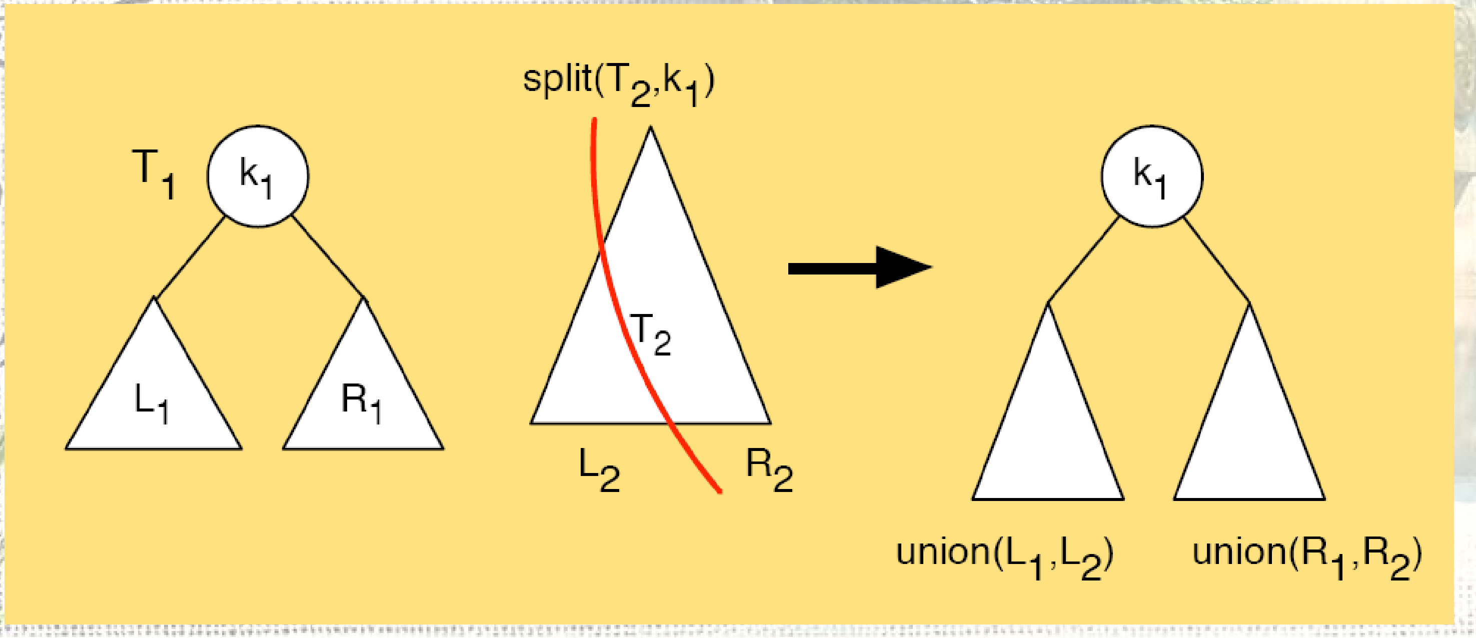
$$\mathit{delete}\ (T,k)\ =\ \textbf{let}\ (L,\_,R)\ =\ \mathit{split}\ (T,k)\ \textbf{in}\ \mathit{join}\ (L,R)\ \textbf{end}$$





A Parametric Implementation

- **Implementing the BST ADT with split and join**
  - **union?**



A Parametric Implementation

- **Implementing the BST ADT with split and join**
  - **union?**

```
union t1 t2 =
  case (t1, t2)
  | (Leaf, _) => t2
  | (_, Leaf) => t1
  | (Node (l1, k1, r1), _) =>
    let (l2, _, r2) = split t2 k1
        (l, r) = (union l1 l2) || (union r1 r2)
    in joinM l k1 r end
```

A Parametric Implementation

- **Implementing the BST ADT with split and join**
  - **Intersect?**

```
intersect t1 t2 =
  case (t1, t2)
  | (Leaf,_) => Leaf
  | (_,Leaf) => Leaf
  | (Node (l1, k1, r1),_) =>
    let (l2, b, r2) = split t2 k1
        (l, r) = (intersect l1 l2) || (intersect r1 r2)
    in if b then joinM l k1 r else join l r end
```

A Parametric Implementation

- **Implementing the BST ADT with split and join**
  - **Diff?**

```
difference t1 t2 =
  case (t1, t2)
  | (Leaf,_) => Leaf
  | (_,Leaf) => t1
  | (Node (l1,k1,r1),_) =>
    let (l2,b,r2) = split t2 k1
        (l,r) = (difference l1 l2) || (difference r1 r2)
    in if b then join l r else joinM L k1 r end
```

Exercise 12.13. Prove correct the functions `intersection`, `difference`, and `union`.

SYNOPSIS

- Preliminaries
- The BST Abstract Data Type
- Implementation via Balancing
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

NOTICES

- These implementations all use balancing techniques to ensure that the **depth of the BST** remains **O(log n)**
- Our cost-specifications can be viewed as **worst-case bounds**
- variables n and m are defined as $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$

	Balanced BST	
	Work	Span
`empty`	$O(1)$	$O(1)$
`singleton v`	$O(1)$	$O(1)$

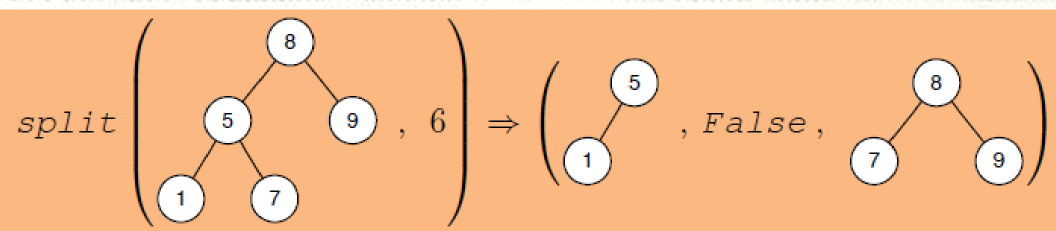
華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

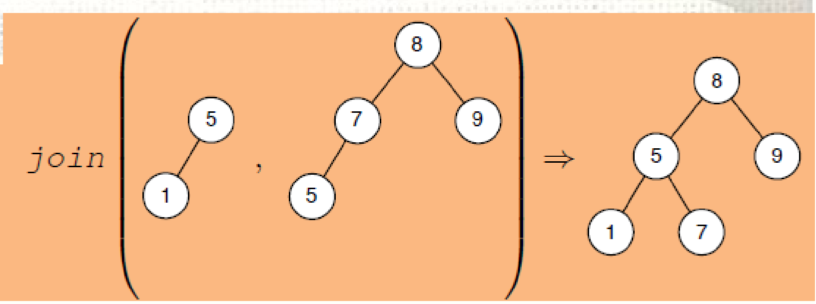
Cost Specification

- Split(T, k) & join(T_1, T_2)
  - W=? S=?

	Work	Span
`split` $t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
`join` $t_1\ t_2$	$O(\lg(\lvert t_1 \rvert + \lvert t_2 \rvert))$	$O(\lg(\lvert t_1 \rvert + \lvert t_2 \rvert))$

how?





Cost Specification

- find(*T*, *k*), insert(*T*, *k*), delete(T,k)
  - W=?, S=?

$$\mathtt{find}\ (T,k) = \textbf{let}\ (\_,v,\_) = \mathtt{split}\ (T,k)\ \textbf{in}\ v\ \textbf{end}$$

$$\mathtt{insert}\ (T,k) = \textbf{let}\ (L,\_,R) = \mathtt{split}\ (T,k)\ \textbf{in}\ \mathtt{joinM}\ (L,k,R)\ \textbf{end}$$

$$\mathtt{delete}\ (T,k) = \textbf{let}\ (L,\_,R) = \mathtt{split}\ (T,k)\ \textbf{in}\ \mathtt{join}\ (L,R)\ \textbf{end}$$

	Work	*Span*
$\mathtt{find}\ t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
$\mathtt{insert}\ t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$
$\mathtt{delete}\ t\ k$	$O(\lg \lvert t \rvert)$	$O(\lg \lvert t \rvert)$

Why?

Cost Specification

- union(T_1, T_2), intersect(T_1, T_2), diff(T_1, T_2)
  - W=?, S=?

how?

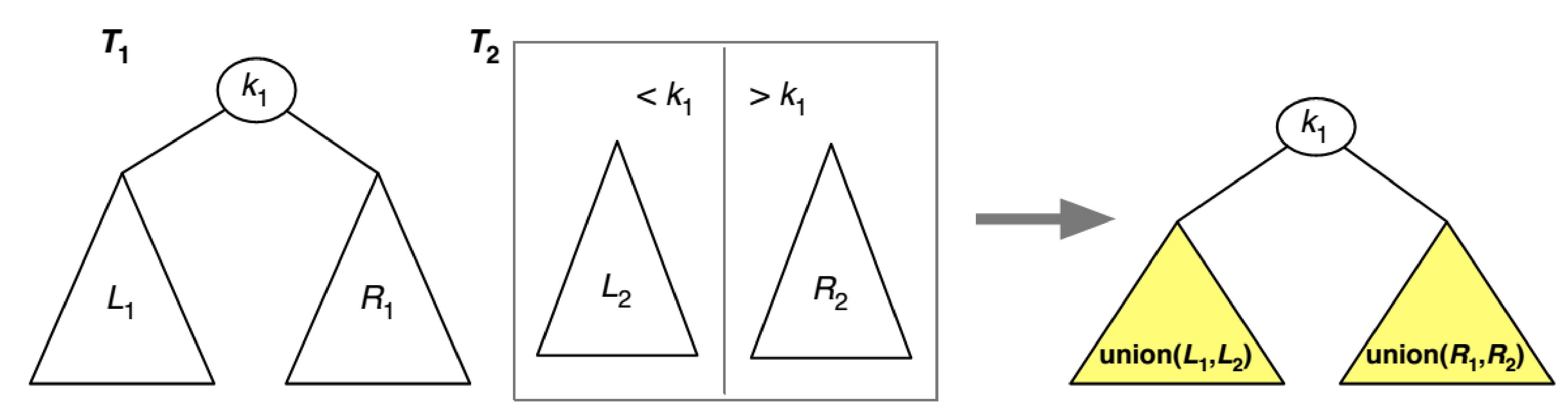
	Work	Span
`intersect` t_1 t_2	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$
`difference` t_1 t_2	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$
`union` t_1 t_2	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$

Concrete Implementations: Union

- **Recall union implement**

$$
\begin{array}{l}
\texttt{union}\ t_1\ t_2\ = \\
\quad \textbf{case}\ (t_1\texttt{,}t_2) \\
\quad |\ \texttt{(Leaf,\_)}\ \texttt{=>}\ t_2 \\
\quad |\ \texttt{(\_,Leaf)}\ \texttt{=>}\ t_1 \\
\quad |\ \texttt{(Node}\ (l_1, k_1, r_1)\texttt{,\_)}\ \texttt{=>} \\
\qquad \textbf{let}\ (l_2, \_, r_2)\ =\ \texttt{split}\ t_2\ k_1 \\
\qquad\qquad (l, r)\ =\ (\texttt{union}\ l_1\ l_2)\ ||\ (\texttt{union}\ r_1\ r_2) \\
\qquad \textbf{in}\ \texttt{joinM}\ l\ k_1\ r\ \textbf{end}
\end{array}
$$

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 =** *union*(**L_1,L_2**) **and R_u=***union***(R_1,R_2)**
- **Now** *join***(L_u,k_1,R_u)**

Analysis of Union

$$
\begin{array}{l}
\texttt{union}\ t_1\ t_2\ = \\
\quad \textbf{case}\ (t_1\texttt{,}t_2) \\
\quad \mid\ (\texttt{Leaf,\_})\ \texttt{=>}\ t_2 \\
\quad \mid\ (\texttt{\_,Leaf})\ \texttt{=>}\ t_1 \\
\quad \mid\ (\texttt{Node}\ (l_1,k_1,r_1)\texttt{,\_})\ \texttt{=>} \\
\qquad \textbf{let}\ (l_2,\_,r_2)\ =\ \texttt{split}\ t_2\ k_1 \\
\qquad\qquad (l,r)\ =\ (\texttt{union}\ l_1\ l_2)\ \|\ (\texttt{union}\ r_1\ r_2) \\
\qquad \textbf{in}\ \texttt{joinM}\ l\ k_1\ r\ \textbf{end}
\end{array}
$$

- split **costs** $O(\lg|T_2|)$
- **Two recursive calls to** union
- join **costs** $O(\lg(|T_1|+|T_2|)$

Analysis of Union - Assumptions

- To simplify the analysis, we will make the following assumptions
  - T_1 is perfectly balanced
  - Each a key from T_1 splits T_2, it splits exactly in half
  - without loss of generality let $|T_1| \le |T_2|$
  - Then, m = $|T_1|$, n=$|T_2|$

$$
\begin{aligned}
W_{\text{union}}(m,n) &\le 2W_{\text{union}}(m/2,n/2) + W_{\text{split}}(n) + W_{\text{join}}(n+m) + O(1) \\
&\le 2W_{\text{union}}(m/2,n/2) + O(\lg n)\ .
\end{aligned}
$$

Why?

華中科技大學

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Analysis of Union

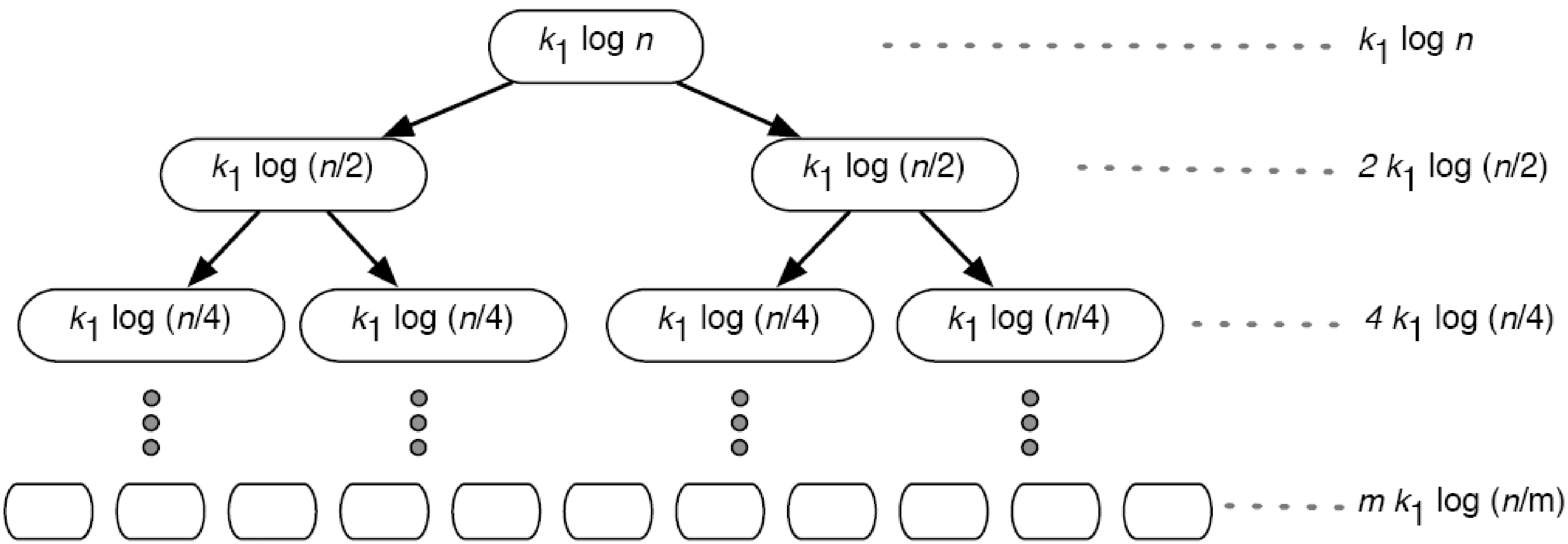
- When $|T_1|=1$, *case* give us two empty subtrees L_1 and R_1
- *union*(L_1,L_2) returns L_2, *union*(R_1,R_2) returns R_2 immediately!
- Joining these costs at most

$$O(\log(|T_1|+|T_2|)) = O(\log(1+|T_2|))$$

$$\begin{aligned} W_{\text{union}}(1,n) &\le 2W_{\text{union}}(0,n/2) + W_{\text{split}}(n) + W_{\text{join}}(n) + O(1) \\ &\le O(\lg n) . \end{aligned}$$

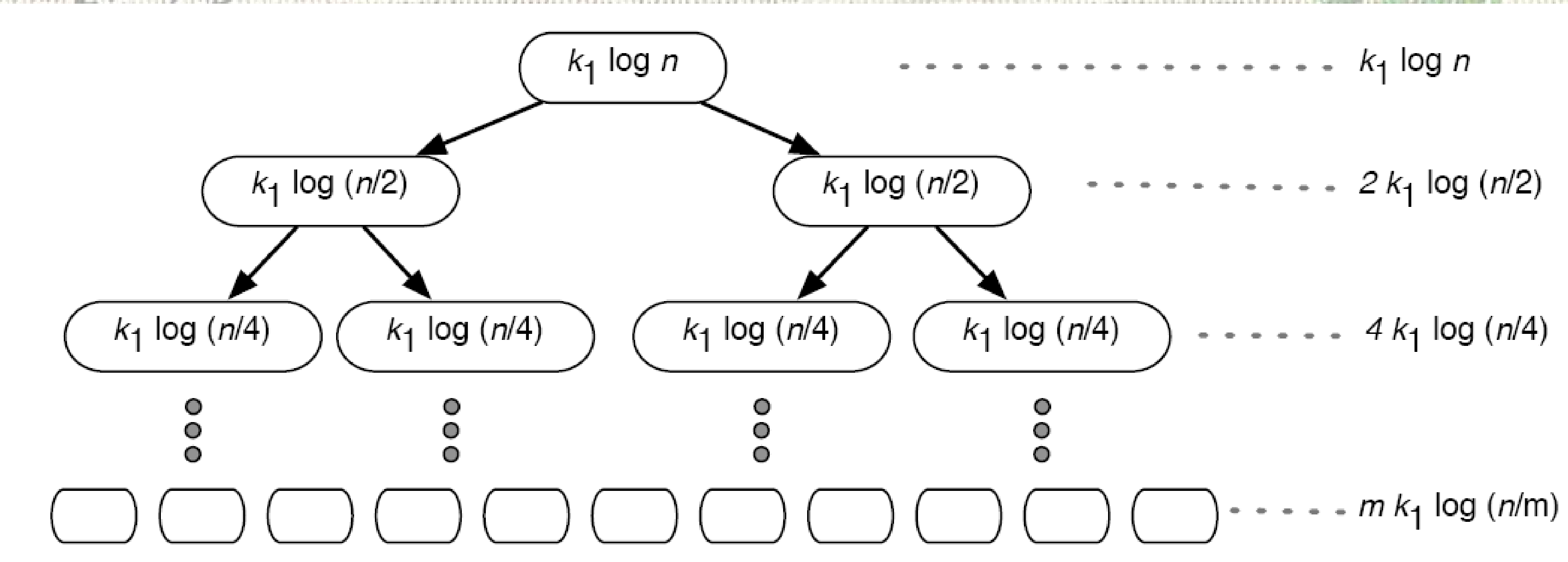
Analysis of Union

- **If we draw the recursion tree that shows the work associated with splitting T_2 and joining the results, we obtain the following**



$$W_{\text{union}}(1, n) \leq \boxed{2W_{\text{union}}(0, n/2)} + W_{\text{split}}(n) + W_{\text{join}}(n) + O(1)$$
$$\leq O(\lg n) \ .$$

Analysis of Union



- **How many leaves are there in this recursion tree?**
- **How deep is the tree?**
- **What is the size of T_2 at the leaves?**

Analysis of Union

- **How many leaves are there in this recursion tree?**
  - T_2 has no impact
  - We get $m=|T_1|$ leaves **Why?**
- **How deep is the tree?**
  - $1+\log_2 m$ **Why?**
- **What is the size of T_2 at the leaves?**
  - $n/2^{\log_2 m} = n/m$ **Why?**
- **Total cost at the leaves = $O(m\log(n/m))$** **Why?**

Analysis of Union

- **We will now prove that the cost at the bottom level is indeed asymptotically the same as the total work.**
  - **It is possible to prove that the tree is leaves-dominated by computing the ratio of the work at adjacent levels,**

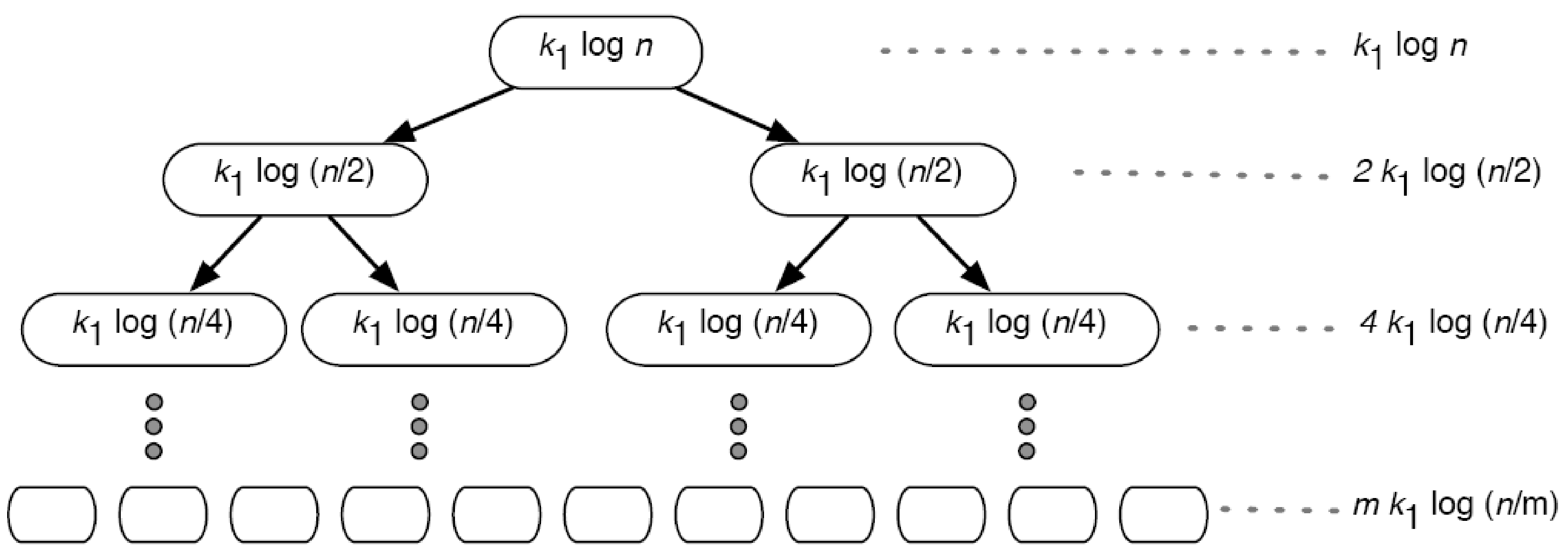
$$\frac{2^{i-1}k_1 \lg n/2^{i-1}}{2^i k_1 \lg n/2^i} = \frac{1}{2}\frac{\lg n - i + 1}{\lg n - i}$$

    - ✓ **where i ≤ logm < lg n**
  - **This ratio is less than 1 for all levels except for the last level, where by taking i = lg n-1 we have**

$$\frac{1}{2}\frac{\lg n - i + 1}{\lg n - i} \leq \frac{1}{2}\frac{1}{\lg n - logn + 1 + 1}\lg n - \lg n + 1 = \frac{1}{1}.$$

Analysis of Union

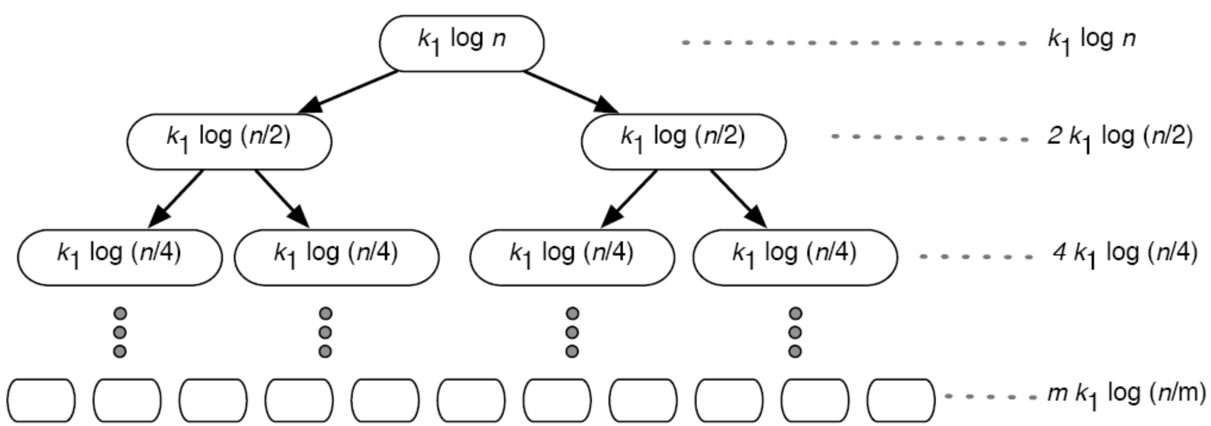
- Thus the total work is asymptotically dominated by the total work of the leaves, which is
  - O (mlg n/m)



Analysis of Union

- **Direct derivation**
  - **We can establish the same fact more precisely.**
  - **Let's start by writing the total cost across all levels, omitting for simplicity the constant factor k_1, and assuming that $n = 2^a$ and $m = 2^b$**

$$\sum_{i=0}^{b} 2^i \lg \frac{n}{2^i}.$$



Analysis of Union

- We can rewrite this sum as

$$\sum_{i=0}^{b} 2^i \lg \frac{n}{2^i} = \lg n \sum_{i=0}^{b} 2^i - \sum_{i=0}^{b} i\, 2^i. = a \sum_{i=0}^{b} 2^i - \sum_{i=0}^{b} i\, 2^i$$

- Let's now focus on the second term

$$\sum_{i=0}^{b} i\, 2^i = \sum_{i=0}^{b} \sum_{j=i}^{b} 2^j = \sum_{i=0}^{b} \left(\sum_{j=0}^{b} 2^j - \sum_{k=0}^{i-1} 2^k \right)$$

Analysis of Union

- **We will now substitute the close form for each inner summation and continue simplifying**

$$\sum_{i=0}^{b} i\,2^i = \sum_{i=0}^{b}\sum_{j=i}^{b} 2^j = \sum_{i=0}^{b}\left(\sum_{j=0}^{b} 2^j - \sum_{k=0}^{i-1} 2^k\right)$$

$$\begin{aligned}
&= \textstyle\sum_{i=0}^{b}\left((2^{b+1}-1)-(2^i-1)\right).\\
&= (b+1)(2^{b+1}-1) - \textstyle\sum_{i=0}^{b}(2^i-1)\\
&= (b+1)(2^{b+1}-1) - \left(2^{b+1}-1-(b+1)\right)\\
&= (b+1)(2^{b+1}-1) - \left(2^{b+1}-1-(b+1)\right)\\
&= b\,2^{b+1}+1.
\end{aligned}$$

Analysis of Union

- **Let's now go back and plug this into our original work bound, i.e.,**

$$\sum_{i=0}^{b} i\,2^i = \sum_{i=0}^{b}\sum_{j=i}^{b} 2^j = \sum_{i=0}^{b}\left(\sum_{j=0}^{b} 2^j - \sum_{k=0}^{i-1} 2^k\right)$$

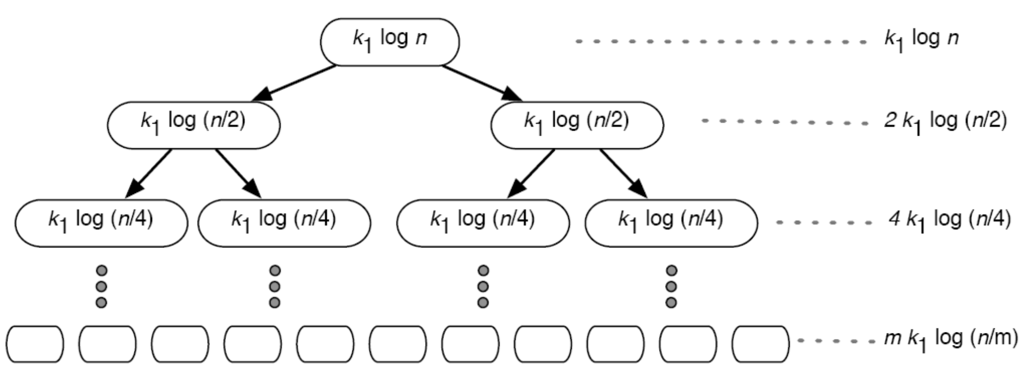
$$\begin{aligned}
&= a\sum_{i=0}^{b} 2^i - \sum_{i=0}^{b} i\,2^i. \\
&= a\,(2^{b+1}-1) - (b\,2^{b+1}+1) \\
&= a\,2^{b+1} - a - b\,2^{b+1} - 1 \\
&= 2m(a-b) - a - 1 \\
&= 2m(\lg n - \lg m) - a - 1 \\
&= 2m(\lg \tfrac{n}{m} - a - 1 \\
&= O(m \lg \tfrac{n}{m}).
\end{aligned}$$

Analysis of Union

- While the direct method may seem complicated, it is **more robust than the brick method**
  - It can be applied to analyze essentially any algorithm, whereas the Brick method requires establishing a geometric relationship between the cost terms at the levels of the tree.

Analysis of Union

- **Removing the Assumptions**
  - **Of course, in reality, our keys in T_1 won't split subtrees of T_2 in half every time**
- **1. keep the assumption that T_1 is perfectly balanced**
  - **the shape of the recursion tree stays the same**
    - **Let us try to analyze the cost at level i**
    - **At this level, there are $k = 2^i$ nodes in the recursion tree**
    - **Say the sizes of T_2 at these nodes are $n_1, \dots, n_k$, where $\sum_j n_j = n$**



Analysis of Union

- 1. keep the assumption that T_1 is perfectly balanced
  - the total cost for this level is

$$c \cdot \sum_{j=1}^{k} \lg(n_j) \leq c \cdot \sum_{j=1}^{k} \lg(n/k) = c \cdot 2^i \cdot \lg(n/2^i),$$

  - used the fact that the logarithm function is concave

?

  - Thus, the tree remains leaf dominated and the same reasoning shows that the total work is O(mlg(n/m))

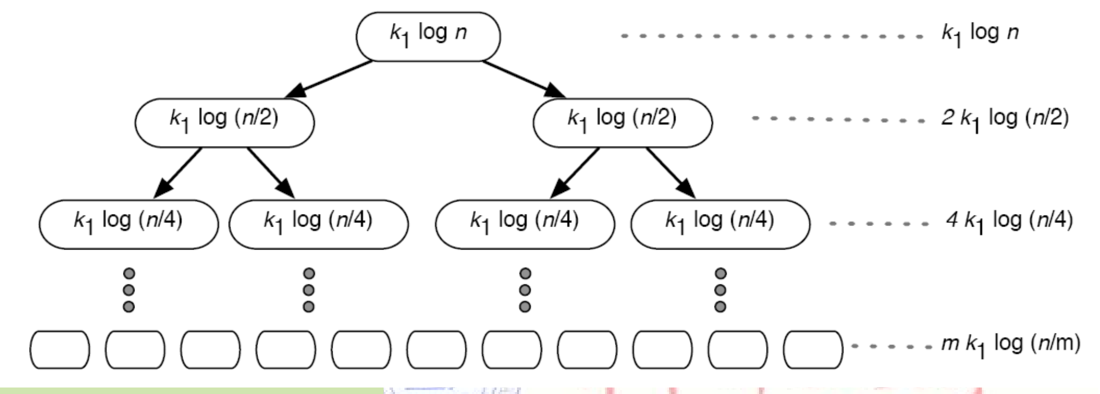
Analysis of Union

- 2. T_1 doesn't have to be perfectly balanced as we assumed
  - A similar reasoning can be used to show that T_1 only has to be approximately balanced.
  - We will leave this case as an exercise

Analysis of Union

- **Span**
  - the span of union is $O(\lg^2 n)$ **how?**
  - but this can be improved to $O(\lg n)$ by changing the algorithm slightly
- **In summary, union can be implemented in**
  - $O(m\lg(n/m))$ work and span $O(\lg n)$

```
union t1 t2 =
  case (t1,t2)
  | (Leaf,_) => t2
  | (_,Leaf) => t1
  | (Node (l1,k1,r1),_) =>
    let (l2,_,r2) = split t2 k1
        (l,r) = (union l1 l2) || (union r1 r2)
    in joinM l k1 r end
```

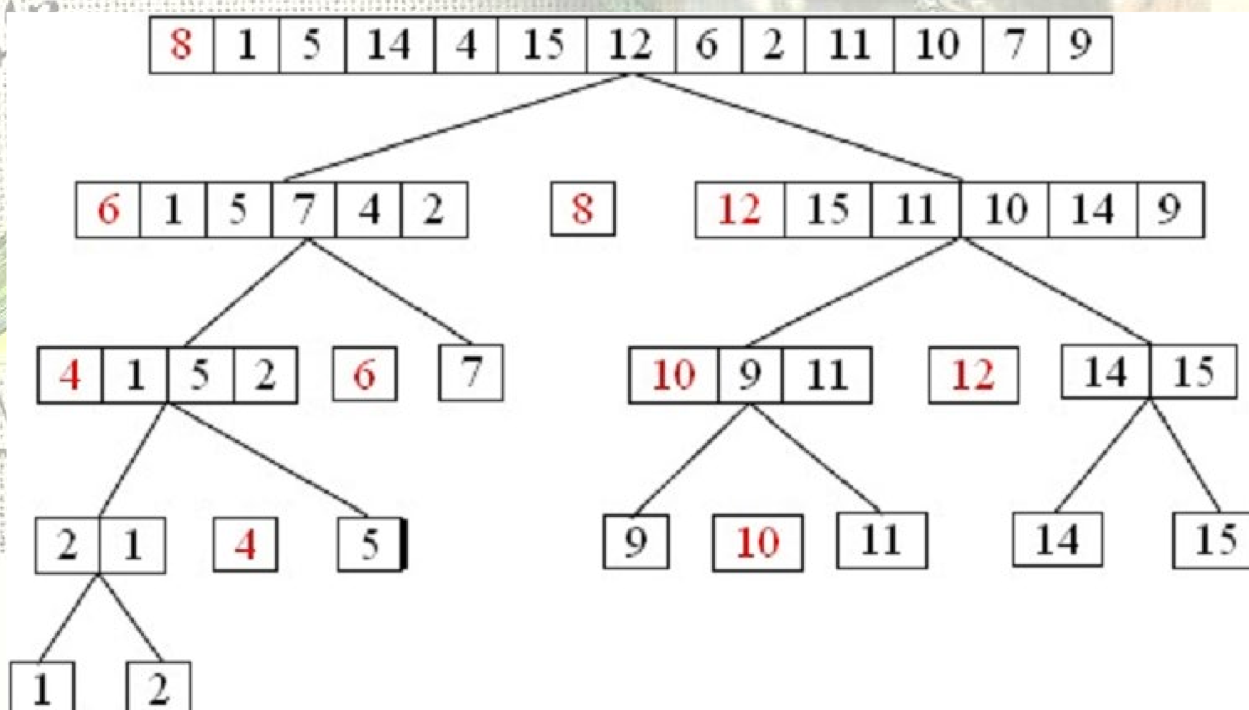


SYNOPSIS

- Preliminaries
- The BST Abstract Data Type
- Implementation via Balancing
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

Quicksort And BSTs

- Write out the recursion tree for quicksort
  - ➢ Assume distinct keys
- Annotate each node with the pivot picked at the stage
- You get a BST



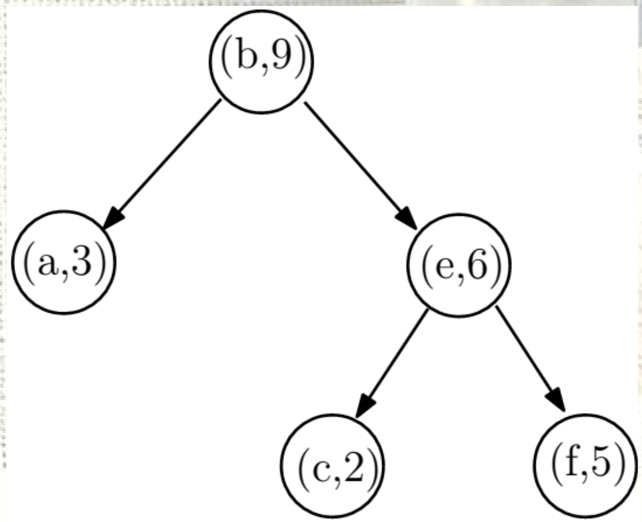
Treaps

- 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)



treaps

- A treap is a binary search tree **T** over a set **S** along with a priority for each key given by

$$p : \mathbb{K} \rightarrow \mathbb{Z}$$

- that in addition to satisfying the BST property on the keys **S**, satisfies the heap property on the priorities p(s), $s \in S$, i.e., for every node v:

$$p(k(v)) \geq p(k(L(v))) \text{ and } p(k(v)) \geq p(k(R(v)))$$

- where **k(v)** denotes the key of a node.

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)

Exercise 12.17. Prove that if the priorities are unique, then there is exactly one tree structure that satisfies the Treap properties.

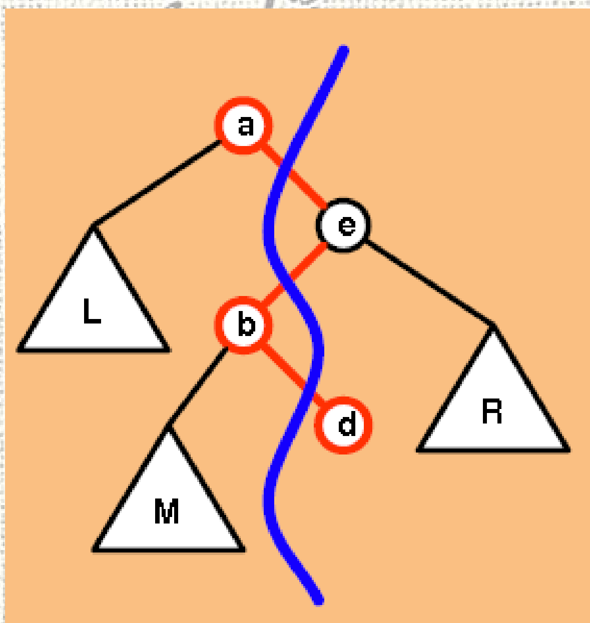
Implementing BST with Treaps

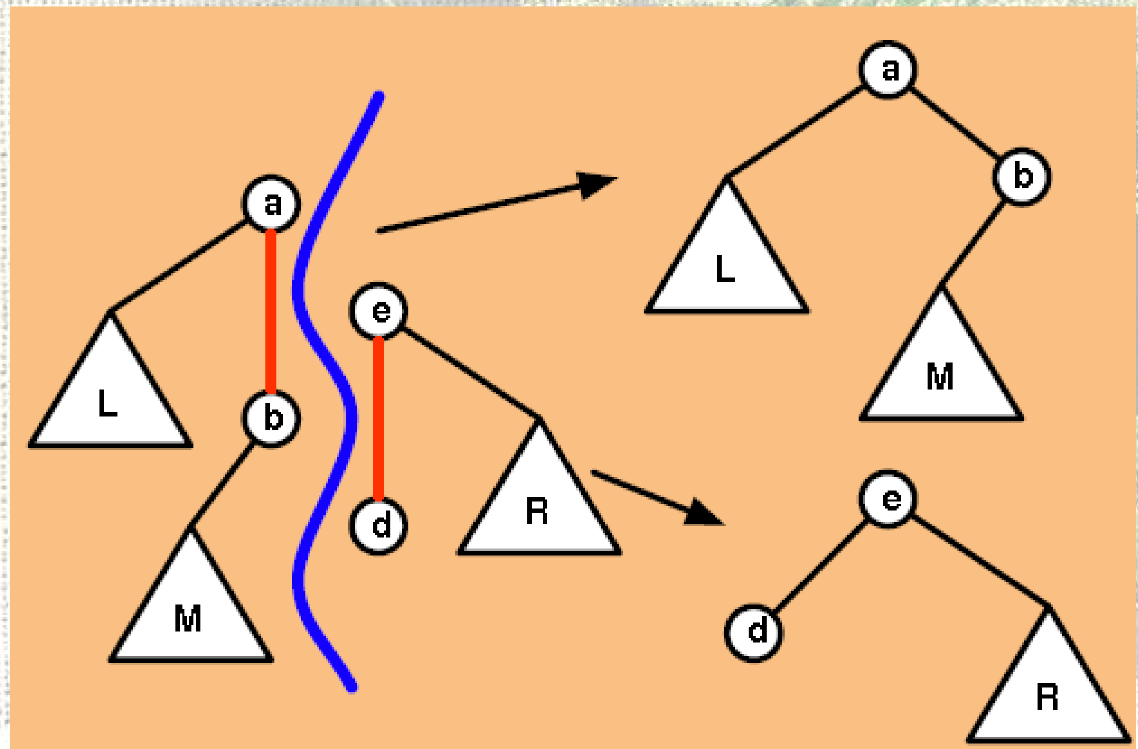
```
type 𝕋 = Leaf | Node of (𝕋 × 𝕂 × ℤ × 𝕋)

let empty = Leaf

singleton(k) = Node(Leaf,k,randomInt(),Leaf)
```

- **randomInt**
  - **returns a (pseudo-)random number**

Implementing split with Treaps

- Split (T, c)





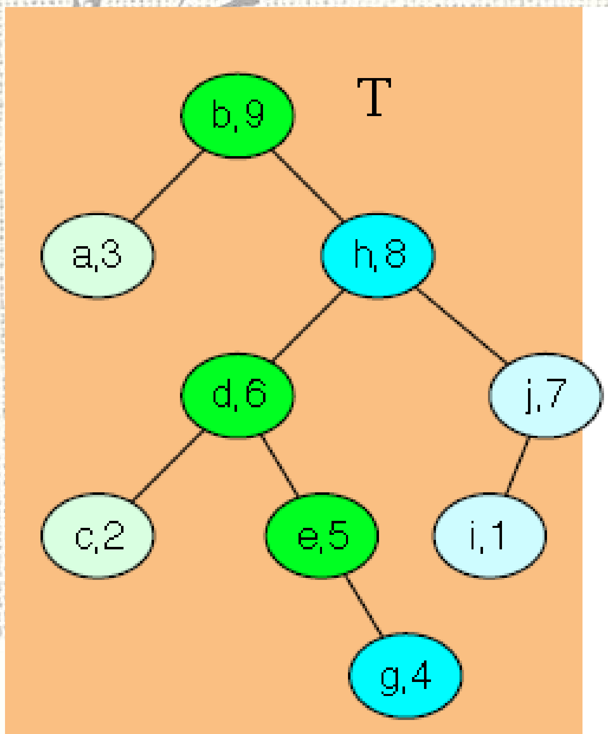
Implementing split with Treaps

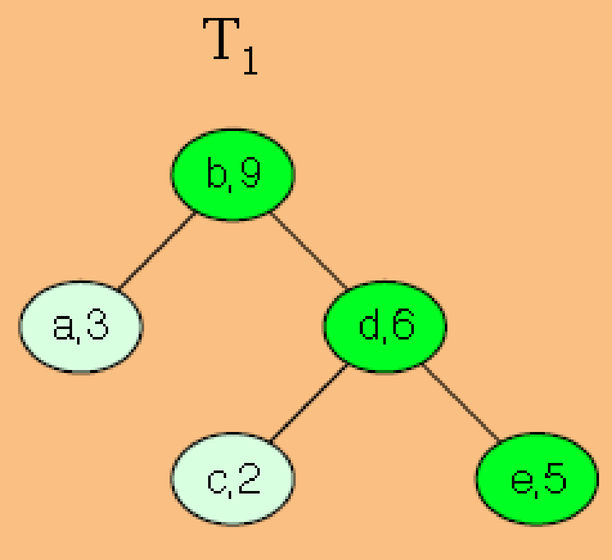
```
split(T,k) =
   case T
   | Leaf ⇒ (Leaf, False, Leaf)
   | Node (L,k',p',R) =
         case compare (k,k')
         | LESS ⇒
               let (L',x,R') = split (L,k)
               in (L',x,Node(R',k',p',R)) end
         | EQUAL ⇒ (L,True,R)
         | GREATER ⇒
               let (L',x,R') = split (R,k)
               in (Node (L,k',p',L'),x,R') end
```

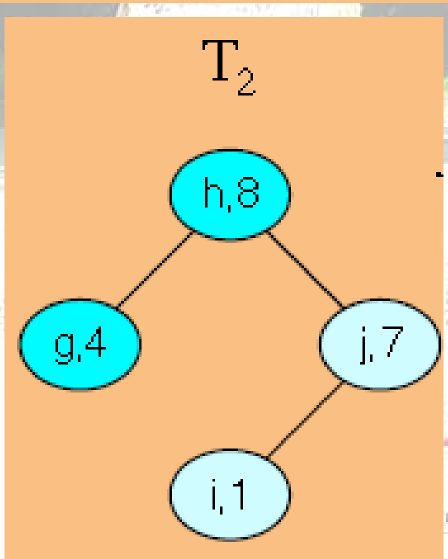
- T={(G,50),(C,35),(E,33),(H,29),(I,25),(B,24),(A,21),(L,16),(J,13),(K,9),(D,8)}
- split(t,l)

Implementing split with Treaps

- **Split(T, f)**

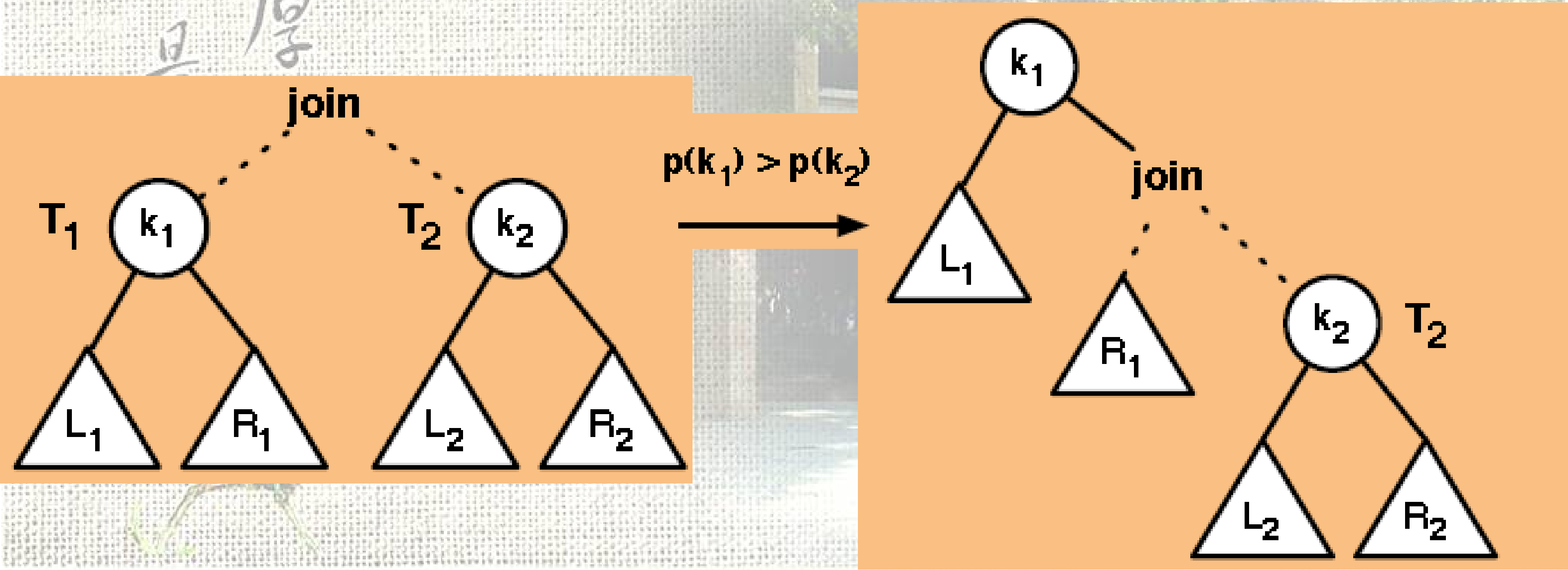






Implementing join with Treaps

- Join

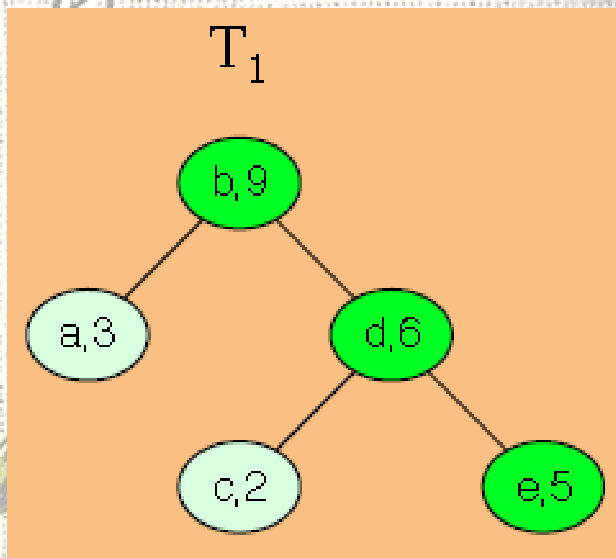


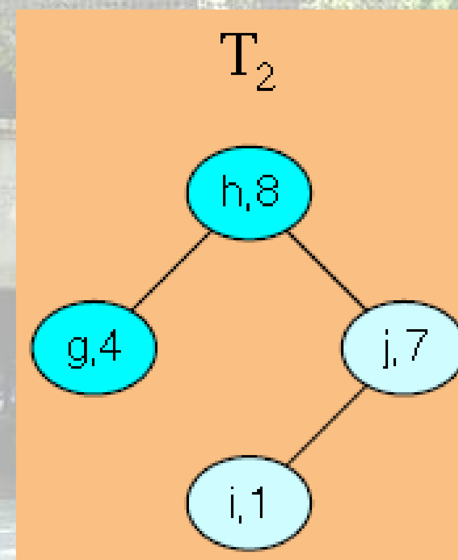
Implementing join with Treaps

```
join(T1, T2) =
    case (T1, T2)
    | (Leaf, _) => T2
    | (_ , Leaf) => T1
    | (Node (L1, k1, p1, R1), Node (L2, k2, p2, R2)) =>
        if (p1 > p2) then
            Node (L1, k1, p1, join (R1, T2))
        else
            Node (join(T1, L2), k2, p2, R2)
end
```

Implementing join with Treaps

- Join(T_1, T_2)





- **Thinking: can we implement split and join just with BST?**

Analysis of randomized treaps

- **analyze the height of a treap assuming that the priorities are picked at random**
- **To do this we will relate treaps to quicksort**

Algorithm 11.21. *Treap Generating Quicksort*

```
qsTree(S) =
   if |S| = 0 then Leaf
   else let
      val pivot = the key k ∈ S for which p(k) is the largest
      val S1 = ⟨ s ∈ S | s < pivot ⟩
      val S2 = ⟨ s ∈ S | s > pivot ⟩
      val (L, R) = (qsTree(S1) || qsTree(S2))
   in
      Node(L, pivot, R)
   end
```

Analysis of randomized treaps

- The tree generated by qsTree(S) is the treap for S, *why? Can you prove?*
- What does this tell us about the height of treaps?
  - the height of a treap is identical to the recursion depth of quicksort
  - if we pick the priorities at random, the recursion depth is O(log n) with high probability

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

SYNOPSIS

- Preliminaries
- The BST Abstract Data Type
- Implementation via Balancing
- A Parametric Implementation
- Cost Specification
- Treaps
- Augmenting Trees
- Exercises

Augmenting Balanced Trees

- We can add other additional values to help with other search operations
  - Track key positions and certain subset sizes
- rank(T, k)
  - How many elements in T are less than k *or equal to* k?
- select(T, i)
  - Returns the key with the rank *i* in *T*
- splitIdx(S, i)
  - Split *S* into two sets: first *i* keys and the remaining *n*–*i* keys

Augmenting Balanced Trees

- **Let T ={1,2,3,4,5,6}**
- rank(T, 4) = **|{1,2,3,4}| = 4**
- rank(T, 4) = **|{1,2,3}| = 3**

Which one is right?

- select(T, 4) **=4 since** rank(S, 4) **=4**
- select(T, 3) **=4 since** rank(S, 4) **=3**

Which one is right?

- splitIdx(T, 3) = **({1,2,3},{4,5,6})**

Augmenting Balanced Trees

- How to implement **Rank (T, k)**?

```
rank (T, k) =
   case T
   | Leaf => 0
   | Node (L, k', R) =>
        case compare (k, k')
        | LESS => rank (L, k)
        | EQUAL => |L|
        | GREATER => |L| + 1 + rank (R, k)
```

Augmenting Balanced Trees

- How to implement **select (T, i)**?

```
select (T,i) =
   case (T)
   | Leaf => raise exception OutOfRange
   | Node (L,k,R) =>
        case compare (i,|L|) of
           LESS => select (L,i)
           EQUAL => k
           GREATER => select (R,i-|L|-1)
```

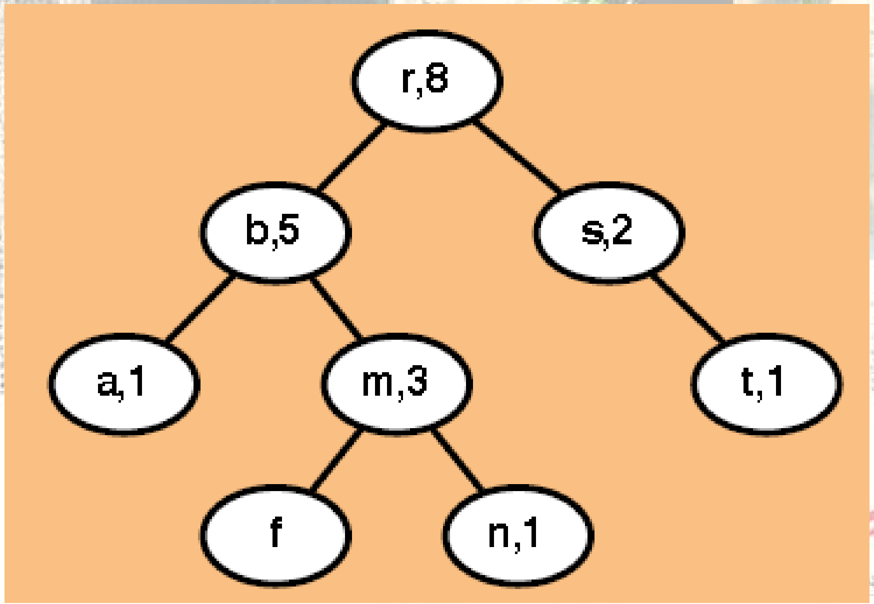
Augmenting Balanced Trees

- **What is the work and span of these functions?**
  - ➢ **Rank (T, k): W=? S=?**
  - ➢ **Select (T, i): W=? S=?**
  - ➢ **Both: W=O(n), S= O(log(n))** — **how?**

$$
\begin{array}{llcl}
\texttt{rank}(S,k) & : \; \mathbb{S} \times \mathbb{U} \rightarrow \mathit{int} & = & |\{k' \in S \mid k' < k\}| \\
\texttt{select}(S,i) & : \; \mathbb{S} \times \mathit{int} \rightarrow \mathbb{U} & = & k \text{ such that } |\{k' \in S \mid k' < k\} \\
\texttt{splitIdx}(S,i) & : \; \mathbb{S} \times \mathit{int} \rightarrow & = & (\{k \in S \mid k < \texttt{select}(S,i)\}, \\
 & \quad \mathbb{S} \times \mathbb{S} & & \{k \in S \mid k \geq \texttt{select}(S,i)\})
\end{array}
$$

Augmenting Balanced Trees

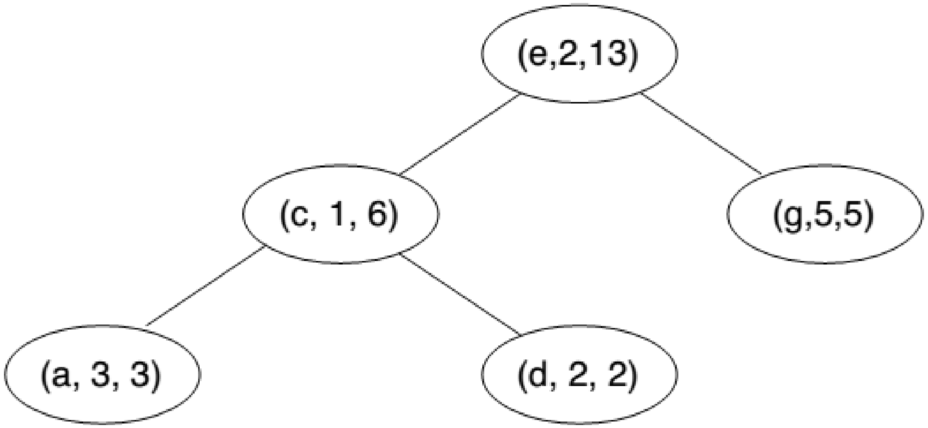
- **Can we compute size of subtrees more efficiently?**
  - **At each node keep the size of the subtree**
  - **This allows** size **and the three other operations in *O(d)* work with *d* as the depth of the tree**
  - **Size can be computed on the fly by adding 1 to the sum of the subtree sizes!**



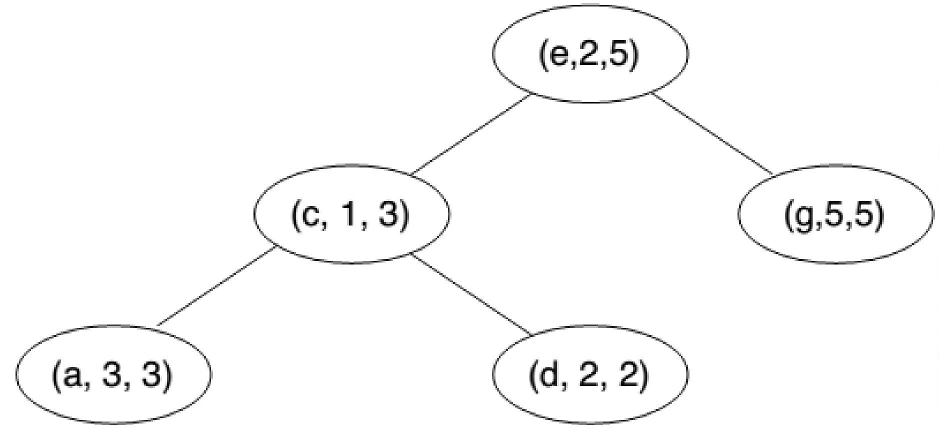
Pairing nodes with reduced values

- **Maintain at each node a "sum" based on an associative operator *f***
  - **Updated during insert/delete, merger, extract, etc**
- **Given $f : v \times v \rightarrow v$, and I_f**
  - **All operations on ordered tables are supported, and**
  - *reduceVal*(A): T→v = *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!**

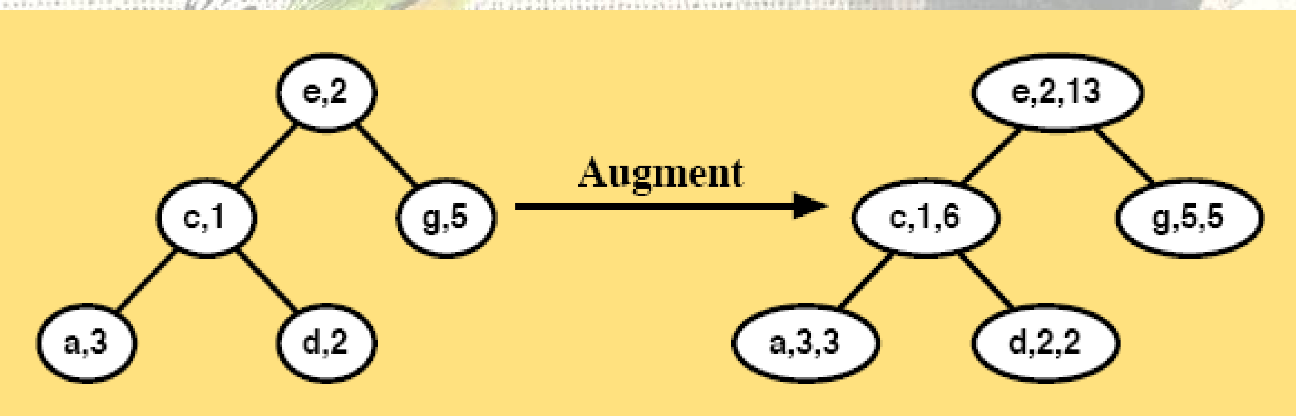
Pairing nodes with reduced values



f is $+$



f is max



Implementation

```
(* type of the reduced value, as specified. *)
type rv = ...
(* associative reducer function, as specified. *)
f(x: rv, y: val, z: rv): rv = ...
(* identity for the reducer function, as specified. *)
id_f : rv = ...

type treap =
    Leaf
  | Node of (Treap × key × priority × (val × rv) × Treap)
```

Implementation

- The only difference in the implementation of split and join functions is the use of **mkNode** instead of **Node**

$$
\begin{array}{l}
\texttt{rvOf}\ t\ = \\
\quad \textbf{case}\ t \\
\quad |\ \texttt{Leaf}\ \texttt{=>}\ id_f \\
\quad |\ \texttt{Node}\ (\_,\_,\_(\_,\texttt{w}),\_)\ \texttt{=>}\ \texttt{w} \\
\\
\texttt{mkNode}\ (l,k,p,v,r)\ =\ \texttt{Node}\ (l,k,p,(v,f\ (\texttt{rvOf}\ l,v,\texttt{rvOf}\ r)),r)
\end{array}
$$

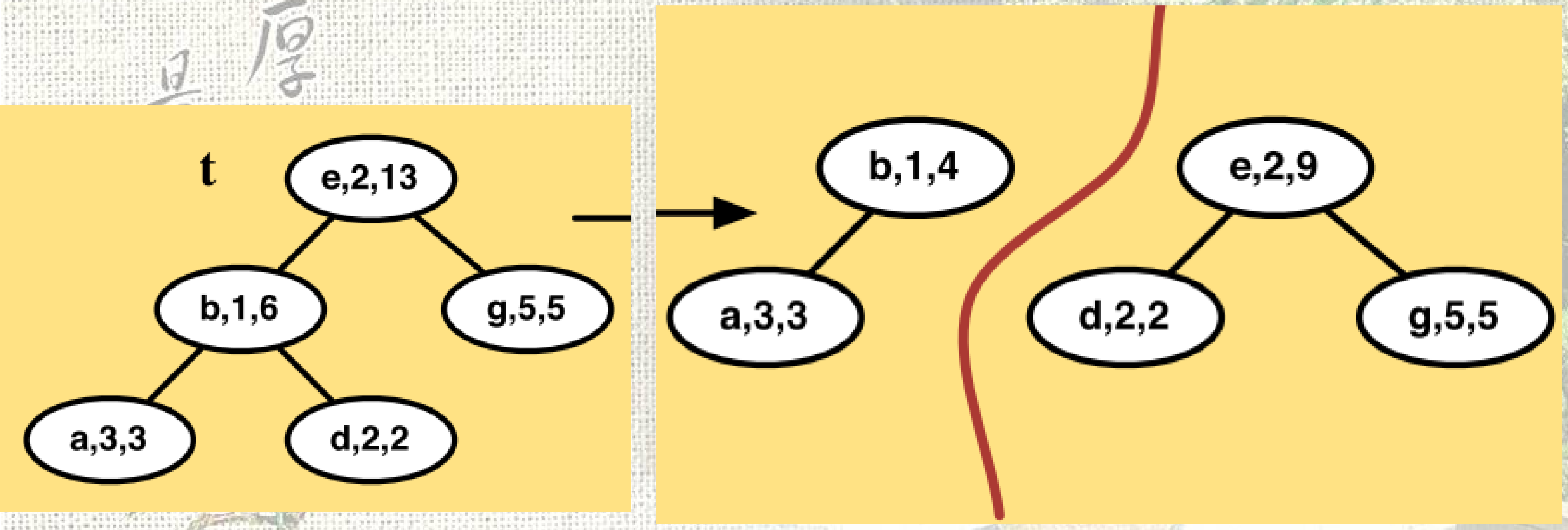
Implementation

- The only difference is the use of **mkNode** instead of **Node**

```
split t k =
  case t
  | Leaf => (Leaf, false, Leaf)
  | Node (l, k', p', (v', w'), r) =
      case compare (k, k')
      | LESS =>
          let (l', x, r') = split l k
          in (l', x, mkNode (r', k', p', v', r)) end
      | EQUAL => (l, true, r)
      | GREATER =>
          let (l', x, r') = split r k
          in (mkNode (l, k', p', v', l'), x, r') end
```

Implementation

- **Split t c**



Implementation

- The only difference is the use of **mkNode** instead of **Node**

$$
\begin{array}{l}
\texttt{join } t_1 \ t_2 = \\
\quad \textbf{case } (t_1, t_2) \texttt{ of} \\
\qquad \texttt{(Leaf, \_) => } t_2 \\
\quad | \ \texttt{(\_, Leaf) => } t_1 \\
\quad | \ \texttt{(Node } (l_1, k_1, p_1, (v_1, w_1), r_1)\texttt{, Node } (l_2, k_2, p_2, (v_2, w_2), r_2)\texttt{) =>} \\
\qquad\quad \textbf{if } p_1 > p_2) \textbf{ then } \texttt{mkNode } (l_1, k_1, p_1, v_1, \texttt{join } r_1 \ t_2) \\
\qquad\quad \textbf{else } \texttt{mkNode } (\texttt{join } t_1 \ l_2, k_2, v_2, r_2)
\end{array}
$$

华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

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**

how?

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**

how?

Example Application – Interval Trees

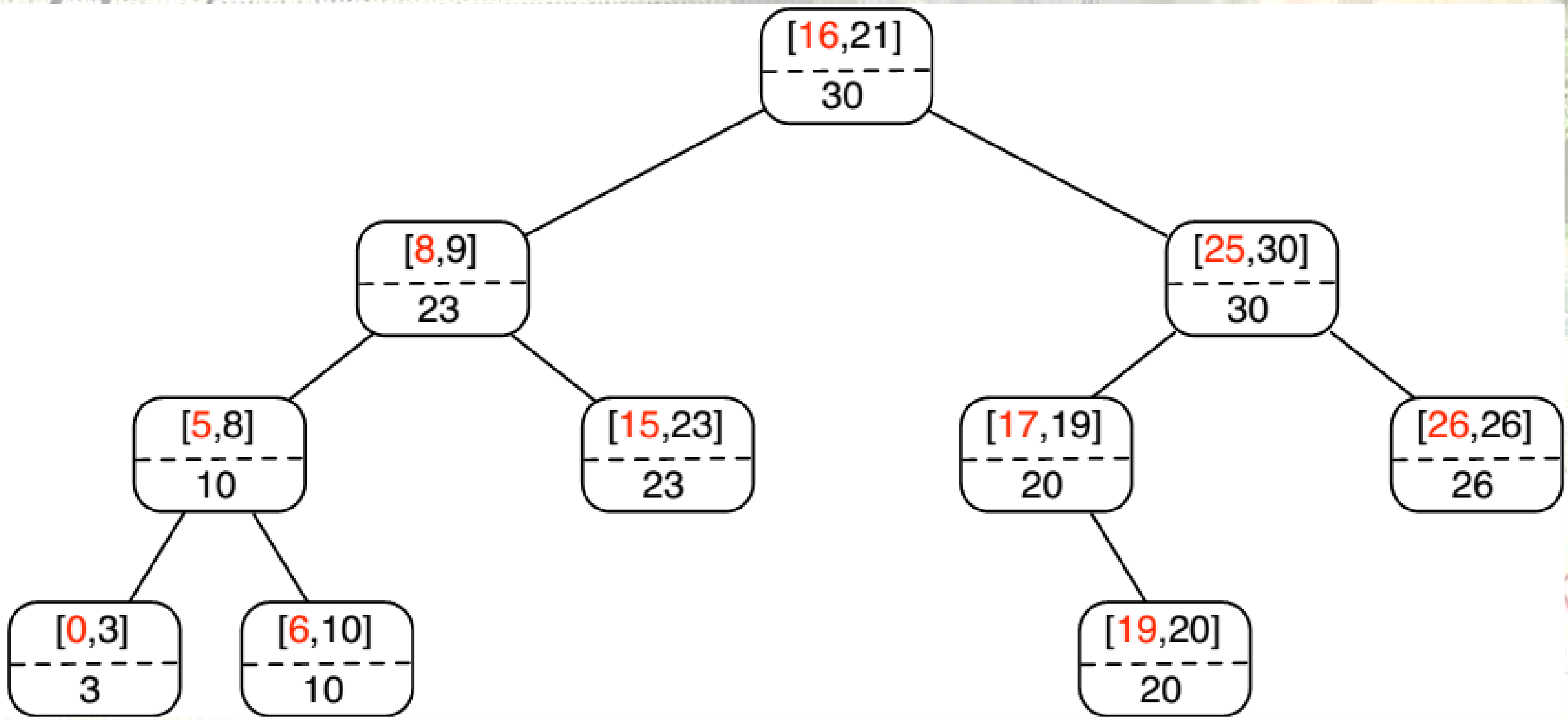
- An interval is a region on the real number line starting at x_l and ending at x_r
- an interval table supports the following operations on intervals:

$\texttt{insert}(A, I)$	:	$\mathbb{T} \times (\texttt{real} \times \texttt{real}) \to \mathbb{T}$	*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*

- How to implement?

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

- How about the **Work and Span**?

```
datatype intTree = Leaf | Node of (intTree × intTree
                                   × real × real × real)

fun overlap(x, low, high) =
      if (x ≥ low & x ≤ high) then 1 else 0

fun countInt(T, x) =
   case T of
      Leaf ⇒ 0
    | Node(L, R, low, high, max) ⇒
        if (x > max) then 0
        else countInt(L, x)+
             overlap(x, low, high)+
             if (x > low) then countInt(R, x) else 0
```

Exercises

12-3 Minimum height
Prove that the minimum possible height of a binary search tree with n keys is $\lceil \log_2(n+1) \rceil$.

12-4 Finding Ranges
Given a BST T and two keys $k_1 \leq k_2$ return a BST T' that contains all the keys in T that fall in the range $[k_1, k_2]$.

12-5 Tree rotations
In a BST T where the root v has two children, let u and w be the left and right child of v respectively. You are asked to reorganize T. For each reorganization design a constant work and span algorithm.

- **Left rotation.** Make w the root of the tree.
- **Right rotation.** Make u the root of the tree.

Exercises

12-6 Size as reduced value

Show that size information can be computed as a reduced value. What is the function to reduce over?

12-7 Implementing `splitRank`

Implement the `splitRank` function.

12-8 Implementing `select`

Implement the `select` function using `splitRank`.

SUMMARY

- BST
- Balanced BST
- Treaps = Balanced BST? = QuickSort?
- Augmenting balanced BST