# Chapter 15

# Binary Search Trees (BSTs)

Search trees are data structures that can be used to efficiently support searches and updates on collections of items that satisfy a total order. Probably, the most common use is to implement sets and tables (dictionaries, mappings), as covered in Chapter 6, and their ordered variants, as covered in the next chapter. You should already know that if all one wants to do is do multiple searches on a static (unchanging) set of ordered elements, then we can presort the elements into an array sequence $S$, and use binary search to find any element in $O(\log |S|)$ work. However, if we want to insert a new element in the sorted array sequence, it requires $\Theta(n)$ work, since many elements have to be moved to make space for the new element. The idea of binary search trees is to allow insertions, and other forms of updates, to be much more efficient. The presentation we give in this chapter is somewhat different than standard presentation, since we are interested in parallel algorithms on trees instead of sequential ones.

## Preliminaries

A binary search tree is a binary tree in which we assign a key to every node, and for each key (node) all keys in its left subtree are less than it, and all keys in its right subtree are greater. The formal definition of rooted trees, binary trees and binary search trees are given in Definitions 15.1, 15.3 and 15.4, respectively. The definitions include some associated terminology we will use. An example of a rooted tree along with the associated terminology is given in Example 15.2, and an example of a binary search tree is given in Example 15.5.

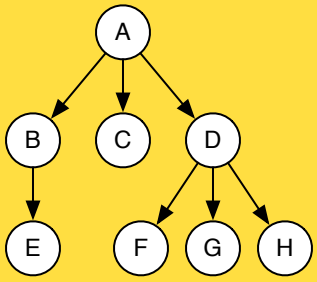
The main idea behind binary search trees is that the keys on the internal nodes allow for us to find if a given key is in our set $S$ by taking a single path through the tree. In particular for a key $k$ we can start at the root $r$ and if $k$ is equal to key there ($k(r)$) then we have found our key, otherwise if $k < k(r)$, then we know that $k$ cannot appear in the right subtree, so we only need to search the left subtree. Continuing the process we will either find the key or reach a leaf and determine the key is not in the tree. In both cases we have followed a single path through the BST. Based on our recursive definition of BSTs, the algorithm for finding a key is summarized

**Definition 15.1** (Rooted Tree)**.** *A* rooted tree *is a directed graph such that*

1. *One of the vertices is the* root *and it has no in edges.*
2. *All other vertices have one in-edge.*
3. *There is a path from the root to all other vertices.*

**Associated Definitions.** When talking about rooted trees, by convention we use the term *node* instead of vertex. A node is a *leaf* if it has no out edges, and an *internal node* otherwise. For each directed edge $(u, v)$, $u$ is the *parent* of $v$, and $v$ is a *child* of $u$. For each path from $u$ to $v$ ($u$ can equal $v$), $u$ is an *ancestor* of $v$, and $v$ is a *descendant* of $u$. For a vertex $v$, its *depth* is the length of the path from the root to $v$ and its *height* is the longest path from $v$ to any leaf. The *height of a tree* is the height of its root. For any node $v$ in a tree, the *subtree rooted at* $v$ is the rooted tree defined by taking the induced subgraph of all vertices reachable from $v$ (i.e. the vertices and the directed edges between them), and making $v$ the root. As with graphs, an *ordered rooted tree* is a rooted tree in which the out edges (children) of each node are ordered.

**Example 15.2.** *An example of a rooted tree:*




| | | |
|---|---|---|
| *root* | : | *A* |
| *leaves* | : | *E, C, F, G, and H* |
| *internal nodes* | : | *A, B, and D* |
| *children of A* | : | *B, C and D* |
| *parent of E* | : | *B* |
| *descendants of A* | : | *all nodes, including A itself* |
| *ancestors of F* | : | *F, D and A* |
| *depth of F* | : | *2* |
| *height of B* | : | *1* |
| *height of the tree* | : | *2* |
| *subtree rooted at D* | : | *the rooted tree consisting of D, F, G and H* |

**Definition 15.3** (Binary Tree)**.** *A* binary tree *is an ordered rooted tree in which every node has exactly two children. We refer to the first child as the* left child *and the second as the* right child*. The* left subtree *of a node is the subtree rooted at the left child, and the* right subtree *the one rooted at the right child.*

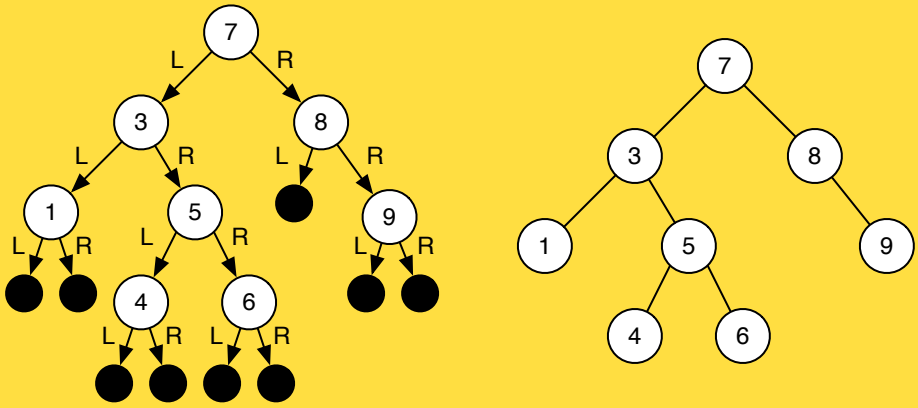
**Definition 15.4** (Binary Search Tree)**.** *A* binary search tree *(BST) over a totally ordered set $S$ is a binary tree that satisfies the following conditions.*

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 $v$, $k(u) > k(v)$*

*Conditions 2 and 3 are referred to as the* BST property*. We often refer to the elements of $S$ in a BST as keys. A BST can equivalently be defined recursively as:*

$$\mathtt{BST}(S) = \begin{cases} \mathtt{Leaf} & S = \emptyset \\ \mathtt{Node}(\mathtt{BST}(S_L), k, \mathtt{BST}(S_R)) & (S = S_L \cup \{k\} \cup S_R) \wedge (S_L < k < S_R) \end{cases}$$

**Example 15.5.** *An example binary search tree over the set $\{1, 3, 4, 5, 6, 7, 8, 9\}$:*




*On the left the $L$ and $R$ indicate the left (first) and right (second) child, respectively. All internal nodes (white) have a key associated with them while the leaves (black) are empty. The keys satisfy the BST property—for every node, the keys in the left subtree are less, and the ones in the right subtree are greater.*

*On the right we have dropped the arrows, since we assume edges go down, the labels $L$ and $R$, since the left and right children will always be placed on the left and right, respectively, and the leaves, since they are implied by a missing child. We will use this convention in future figures.*

by the following:

**Algorithm 15.6** (Finding a key in a BST)**.**

```
fun find(T, k) =
  case T of
    Leaf ⇒ False
  | Node(L, k', R) ⇒
      case compare(k, k') of
        Less ⇒ find(L, k)
      | Equal ⇒ true
      | Greater ⇒ find(R, k)
```

Search trees can be generalized beyond binary trees to work with trees nodes with higher degrees. To do this, for a node with $k$ children, we would need $k-1$ keys so separate each of the children. In this book we only cover binary search trees.

**A Balancing Act.** Since a find only follows a path in the BST, and assuming the comparison at each step takes constant work, then `find` takes work at most proportional to the height of the tree. Therefore, if we want to find keys quickly, we need to keep the height of the tree low. A binary tree is defined to be *balanced* if it has the minimum possible height. For a binary search tree over a set $S$, a balanced tree has height exactly $\lceil \log_2(|S|+1) \rceil$.

**Exercise 15.7.** *Prove that the minimum possible height of a binary search tree with $n$ keys is $\lceil \log_2(n+1) \rceil$.*

Ideally we would always use a balanced tree. Indeed if we never make changes to the tree, we could balance it once up front and it would always be balanced. However, if we want to update the tree by, for example, inserting new keys or combining two trees, then maintaining balance is hard. It turns out to be impossible, for example, to maintain a balanced tree while allowing insertions in $O(\log n)$ work. So, instead, our goal is to use a scheme that keep the trees nearly balanced. We say that a balancing scheme maintains *near balance* if all tress with $n$ elements have height $O(\log n)$, perhaps in expectation or with high probability.

There are many schemes that allow updates while maintain trees that are nearly balanced. Most schemes either try to maintain height balance (the children of a node are about the same height) or weight balance (the children of a node are about the same size, i.e., the number of elements in the subtrees). Here we list a few schemes for maintaining near balance for binary search trees:

1. *AVL trees.* This is the earliest near balance scheme (1962). It maintains the invariant that the two children of each node differ in height by at most 1, which in turn implies near balance.

2. *Red-Black trees.* This scheme maintains 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.

3. *Weight balanced (BB[$\alpha$]) trees.* Maintains the invariant that the left and right subtrees of a node of size $n$ each have size at least $\alpha n$ for $0 < \alpha \leq \frac{1}{2}$. The BB stands for bounded balance, and adjusting $\alpha$ gives a tradeoff between search and update costs.

4. *Treaps.* Associates a random priority with every key and maintains the invariant that the keys are stored in heap order with respect to their priorities (treaps is short for tree-heaps). The scheme guarantees near balance with high-probability.

5. *Splay trees.* An amortized scheme that does not guarantee near balance, but instead guarantees that for any sequence of $m$ insert, search and delete operations each does $O(\log n)$ amortized work.

There are dozens of other schemes on binary trees (e.g. scapegoat trees and skip trees), as well as many that allow larger degrees, including 2–3 trees , brother trees, and B trees. In this chapter we will cover Treaps.

## 15.1 Split and Join

Traditionally, treatments of binary search trees concentrate on three operations: `search`, `insert`, and `delete`. Out of these, search is naturally parallel since any number of searches can proceed in parallel with no conflicts[1]. Insert and delete, however, are inherently sequential, as normally described. For this reason, we'll discuss more general operations that are useful for implementing parallel updates, of which insert and delete are just a special case.

As defined, a BST is either a leaf or an internal node with two children (left and right) and a key associated with it. Each of the left and right children are themselves BSTs. In the discussion in this chapter we find it convenient to also include a value associated with each node. This makes it convenient to define tables. This leads to the following recursive definition:

```
datatype BST =
    Leaf
  | Node of (BST × (key × value) × BST)
```

In addition, depending on the type of tree, we might also keep balance information or other information about the tree stored at each node. We will add such information as we need it. The keys must satisfy the BST property (Definition 15.4).

We'll rely on the following two basic building blocks to construct other functions, such as `search`, `insert`, and `delete`, but also many other useful functions such as intersection and union on sets.

[1] In splay trees and other self-adjusting trees, this is not true since a searches can modify the tree.

$\mathit{split}(T, k) : \mathit{BST} \times \mathit{key} \to \mathit{BST} \times (\mathit{value}\ \mathit{option}) \times \mathit{BST}$
: Given a BST $T$ and key $k$, divide $T$ into two BSTs, one consisting of all the keys from $T$ less than $k$ and the other all the keys greater than $k$. Furthermore if $k$ appears in the tree with associated value $d$ then return $\mathit{SOME}(d)$, and otherwise return $\mathit{NONE}$.

$\mathit{join}(L, m, R) : \mathit{BST} \times (\mathit{key} \times \mathit{value})\ \mathit{option} \times \mathit{BST} \to \mathit{BST}$
: Taking a left BST $L$, an optional middle key-value pair $m$, and a right BST $R$, the function requires that all keys in $L$ are less than all keys in $R$. Furthermore if the optional middle element is supplied, then its key must be larger than any in $L$ and less than any in $R$. It creates a new BST which is the union of $L$, $R$ and the optional $m$.

For both split and join we assume that the BST taken and returned by the functions obey some balance criteria. For example, they might be red black trees. To maintain abstraction over the particular additional data needed to maintain balance (e.g. the color for a red-black tree) we will use the following function to expose the root of a tree without the additional data:

$\mathit{expose}(T) : \mathit{BST} \to (\mathit{BST} \times (\mathit{key} \times \mathit{value}) \times \mathit{BST})\ \mathit{option}$
: Given a BST $T$, if $T$ is empty it returns $\mathit{NONE}$. Otherwise it returns the left child of the root, the right child of the root, and the key and value stored at the root.

With these functions, we can implement `search`, `insert`, and `delete` quite simply:

```
function search(T, k) =
  let val (_, v, _) = split(T, k)
  in v end

function insert(T, (k, v)) =
  let val (L, _, R) = split(T, k)
  in join(L, SOME(k, v), R) end

function delete(T, k) =
  let val (L, _, R) = split(T, k)
  in join(L, NONE, R) end
```

**Exercise 15.8.** *Write a version of* `insert` *that takes a function* $f : \mathit{value} \times \mathit{value}$ *and if the insertion key* $k$ *is already in the tree applies* $f$ *to the old and new value to return the value to associate with the key.*

As we show later, implementing search, insert and delete in terms of $\mathit{split}$ and $\mathit{join}$ is asymptotically no more expensive than a direct implementation. There might be some constant factor overhead, however, so in an optimized implementation search, insert, and delete might be implemented directly. More interestingly, we can use $\mathit{split}$ and $\mathit{join}$ to implement union, intersection, or difference of two BSTs, as described later. Note that union differs from $\mathit{join}$ since it does not require that all the keys in one appear after the keys in the other; the keys may overlap. We describe how to implement $\mathit{union}$ in Section 15.6. The implementation of $\mathit{intersection}$ and $\mathit{difference}$ are very similar.

**Algorithm 15.9** (Split and Join with no balance criteria)**.**

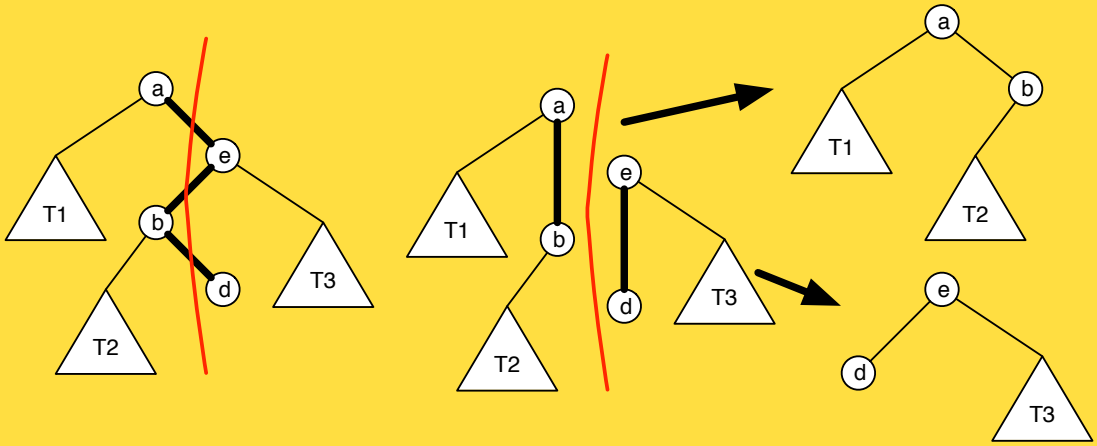
```
1  function split(T, k) =
2    case T of
3      Leaf ⇒ (Leaf, NONE, Leaf)
4    | Node(L, (k', v), R) =
5        case compare(k, k') of
6          LESS ⇒
7            let val (L', m', R') = split(L, k)
8            in (L', m', Node(R', (k', v), R)) end
9        | EQUAL ⇒ (L, SOME(v), R)
10       | GREATER ⇒
11           let val (L', m', R') = split(R, k)
12           in (Node(L, (k', v), L'), m', R') end

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

## 15.2 Implement Split and Join on a Simple BST

We now consider a concrete implementation of `split` and `join` for a particular BST. For simplicity, we consider a version with no balance criteria. The algorithms are described in Algorithm 15.9. The idea of split is to recursively traverse the tree from the root to the key splitting along the path, and then to put the subtrees back together in the appropriate way on when returning from the recursive calls back to the root.

**Example 15.10.** *In the following tree we split on the key $c$. The split traverses the path $\langle a, b, d, e \rangle$ turning right at $a$ and $b$ (line 10 of the Algorithm 15.9) and turning left at $e$ and $d$ (line 6). The pieces are put back together into the two resulting trees on the way back up the recursion.*




We claim that a similar approach can be easily use to implemented `split` and `join` on just about any balanced search tree, although with some additional code to maintain balance.

## 15.3 Quicksort and BSTs

Can we think of binary search trees in terms of an algorithm we already know? As is turns out, the quicksort algorithm and binary search trees are closely related: if we write out the recursion tree for quicksort and annotate each node with the pivot it picks, what we get is a BST.

Let's try to convince ourselves that the function-call tree for quicksort generates a binary search tree when the keys are distinct. To do this, we'll modify the quicksort code from a earlier lecture to produce the tree as we just described. In this implementation, we assume the pivot is selected based on a priority function that maps every key to a unique **priority**:

$$p(k) : \texttt{key} \to \mathcal{R}$$

In particular when selecting the pivot we always pick the key with the highest priority as the pivot. If the priority function is random then this will effectively pick a random pivot.

**Algorithm 15.11.** *Tree Generating Quicksort*

```
1  function qsTree(S) =
2    if |S| = 0 then Leaf
3    else let
4      val pivot = the key from S for which p(k) is the largest
5      val S1 = ⟨ s ∈ S | s < pivot ⟩
6      val S2 = ⟨ s ∈ S | s > pivot ⟩
7      val (TL, TR) = (qsTree(S1) || qsTree(S2))
8    in
9      Node(TL, p, TR)
10   end
```

Notice that this is clearly a binary tree. To show that this is a binary *search* tree, we only have to consider the ordering invariant. But this, too, is easy to see: for `qsTree` call, we compute $S_1$, whose elements are strictly smaller than $p$—and $S_2$, whose elements are strictly bigger than $p$. So, the tree we construct has the ordering invariant. In fact, this is an algorithm that converts a sequence into a binary search tree.

Also notice that the key with the highest priority will be at the root since the highest priority is selected as the pivot and the pivot is placed at the root. Furthermore for any subtree, the highest priority key of that subtree will be at the root since it would have been picked as the pivot first in that subtree. This is important, as we will see shortly.

It should be also clear that the maximum depth of the binary search tree resulting from `qsTree` is the same as the maximum depth of the recursion tree for quicksort using that strategy. As shown in Chapter 7, if the pivots are randomly selected then the recursion tree has depth $O(\log n)$ with high probability. If we assume the priority function $f(k)$ is random (i.e. generates a random priority for each key), then the tree generated by `qsTree` will have depth $O(\log n)$ with high probability.

The surprising thing is that we can maintain a binary search tree data-structure that always has the exact same binary tree structure as generated by `qsTree`. This implies that it will always have $O(\log n)$ depth with high probability.

## 15.4 Treaps

Unlike quicksort, when inserting one-by-one into a BST we don't know all the elements that will eventually be in the BST, so we do not know immediately which one will have the highest priority and will end up at the root.

To maintain the same tree as `qsTree` we first note that we need to maintain the highest priority key at the root. Furthermore, as stated above, within each subtree the highest priority key needs to be at the root. This leads to the key idea of the `treaps` data structure, which is to

maintain the keys in BST order and maintain their priorities in "heap" order. A heap is a tree in which for every subtree, the highest priority value (either largest or smallest) is at the root. The term treap comes from TRee-hEAP. To summarize a treap satisfies the two properties.

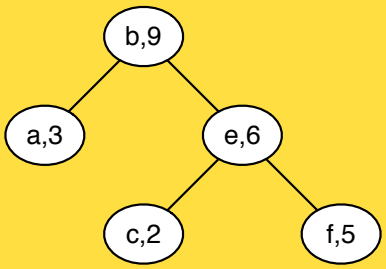
**BST Property:** Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).

**Heap Property:** The associated priorities satisfy the *heap* property. The (max) heap property requires for every node that the value at a node is greater than the value of its two children.

**Example 15.12.** *Consider the following key-priority pairs* $(k, p(k))$*:*

$$(a, 3), (b, 9), (c, 2), (e, 6), (f, 5)$$

*Assuming the keys are ordered alphabetically, these elements would be placed in the following treap.*




**Theorem 15.13.** *For any set $S$ of key-priority pairs with unique keys and unique priorities, there is exactly one treap $T$ containing the key-priority pairs in $S$ which satisfies the treap properties.*

*Proof.* (By induction) There is only one way to represent the empty tree (base case). The key $k$ with the highest priority in $S$ must be the root node, since otherwise the tree would not be in heap order. Only one key has the highest priority. Then, to satisfy the property that the treap is ordered with respect to the nodes' keys, all keys in $S$ less than $k$ must be in the left subtree, and all keys greater than $k$ must be in the right subtree. Inductively, the two subtrees of $k$ must be constructed in the same manner. □

Note, there is a subtle distinction here with respect to randomization. With quicksort the algorithm is randomized. With treaps, none of the functions for treaps are randomized. It is the data structure itself that is randomized.

## Split and Join on Treaps

As mentioned earlier, for any binary tree all we need to implement is split and join and these can be used to implement the other BST operations. Recall that split takes a BST and a key and splits the BST into two BST and an optional value. One BST only has keys that are less than

**Algorithm 15.15** (Join on Treaps)**.**

```
1  function join(T1, m, T2) =
2  let
3     fun singleton(k, v) = Node(Leaf,(k, v),Leaf)
4     fun join2(T1, T2) =
5        case (T1, T2) of
6           (Leaf , _) ⇒ T2
7        | (_ , Leaf) ⇒ T1
8        | (Node(L1, (k1, v1), R1), Node(L2, (k2, v2), R2)) ⇒
9           if (p(k1) > p(k2)) then
10             Node(L1, (k1, v1), join2(R1, T2))
11          else
12             Node(join2(T1, L2), (k2, v2), R2)
13 in
14    case m of
15       NONE ⇒ join2(T1, T2)
16    | SOME(k, v) ⇒ join2(T1, join2(singleton(k, v), T2))
17 end
```

the given key, the other BST only has keys that are greater than the given key, and the optional value is the value of the given key if in the tree. Join takes two BSTs and an optional middle (key,value) pair, where the maximum key on the first tree is less than the minimum key on the second tree. It returns a BST that contains all the keys the given BSTs and middle key.

We claim that the split code given above for unbalanced trees does not need to be modified for treaps.

**Exercise 15.14.** *Convince yourselves than when doing a split none of the priority orders change (i.e. the code will never put a larger priority below a smaller priority).*

The join code, however, does need to be changed. The new version has to check the priorities of the two roots, and use whichever is greater as the new root. The algorithm is given in Algorithm 15.15. In the code recall that $p(k)$ is the priority for the key $k$. The function `join2` is a version of join that has no middle element as an argument. Note that line 20 compares the priorities of the two roots and then places the key with the larger priority in the new root causing a recursive call to join on one of the two sides. This is illustrated in Figure 15.1.

We refer to the *left spine* of the tree as the path from the root to the leftmost node in the tree, and the *right spine* as the path from the root to the rightmost node in the tree. What $\texttt{join}'(T_1, T_2)$ does is to interleave pieces of the right spine of $T_1$ with pieces the left spine of $T_2$, in a way that ensures that the priorities are in decreasing order down the path.

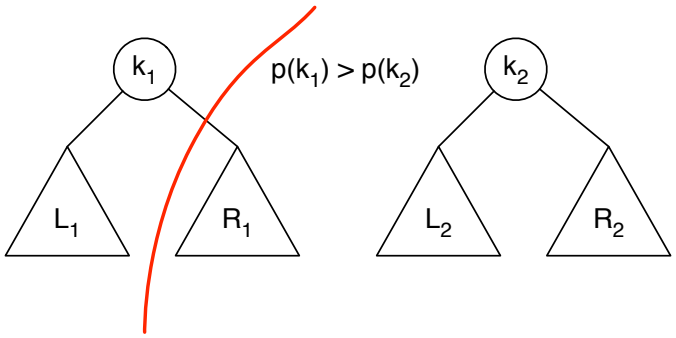
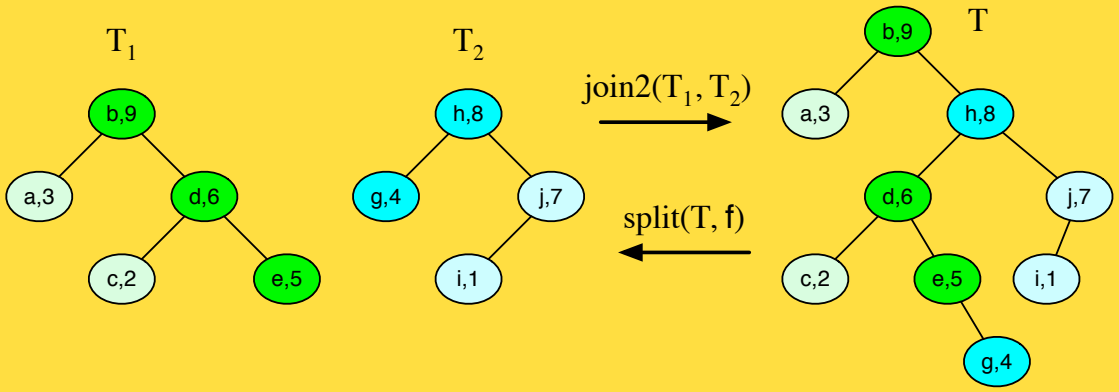



Figure 15.1: Joining two trees $T_1$ and $T_2$. If $p(k_1) < p(k_2)$ then we recurse with $\texttt{join2}(R_1, T_2)$ and make that the right child of $k_1$.

**Example 15.16.** *In the following illustration two treaps are joined. The right spine of $T_1$ consisting of $(b, 9)$, $(d, 6)$ and $(e, 5)$ is effectively merged with the left spine of $T_2$ consisting of $(h, 8)$ and $(g, 4)$. Note that splitting the result with $f$ will return the original two trees.*




Because the keys and priorities determine a treap uniquely, splitting a tree and joining it back together results in the same treap. This property is not true of most other kinds of balanced trees; the order that operations are applied can change the shape of the tree.

Because the cost of `split` and `join` depends on the depth of the $i^{th}$ element in a treap, we now analyze the expected depth of a key in the tree.

## 15.5 Expected Depth of a Key in a Treap

Consider a set of keys $K$ and associated priorities $p : \texttt{key} \to \texttt{int}$. For this analysis, we assume the priorities are unique and random. Consider the keys laid out in order, and as with the analysis

of quicksort, we use $i$ and $j$ to refer to the $i^{th}$ and $j^{th}$ keys in this ordering. Unlike quicksort analysis, though, when analyzing the depth of a node $i$, $i$ and $j$ can be in any order, since an ancestor of $i$ in a BST can be either less than or greater than $i$.


If we calculate the depth starting with zero at the root, the expected depth of a key is equivalent to the number of ancestors it has in the tree. So we want to know how many ancestors a particular node $i$ has. We use the indicator random variable $A_i^j$ to indicate that $j$ is an ancestor of $i$. (Note that the superscript here does not mean $A_i$ is raised to the power $j$; it simply is a reminder that $j$ is the ancestor of $i$.) By the linearity of expectations, the expected depth of $i$ can be written as:

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

To analyze $A_i^j$ let's just consider the $|j-i|+1$ keys and associated priorities from $i$ to $j$ inclusive of both ends. As with the analysis of quicksort in Chapter 7, if an element $k$ has the highest priority and $k$ is less than both $i$ and $j$ or greater than both $i$ and $j$, it plays no role in whether $j$ is an ancestor of $i$ or not. The following three cases do:

1. The element $i$ has the highest priority.

2. One of the elements $k$ in the middle has the highest priority (i.e., neither $i$ nor $j$).

3. The element $j$ has the highest priority.

What happens in each case?

1. If $i$ has the highest priority then $j$ cannot be an ancestor of $i$, and $A_i^j = 0$.

2. If $k$ between $i$ and $j$ has the highest priority, then $A_i^j = 0$, also. Suppose it was not. Then, as $j$ is an ancestor of $i$, it must also be an ancestor of $k$. That is, since in a BST every branch covers a contiguous region, if $i$ is in the left (or right) branch of $j$, then $k$ must also be. But since the priority of $k$ is larger than that of $j$ this cannot be the case, so $j$ is not an ancestor of $i$.

3. If $j$ has the highest priority, $j$ must be an ancestor of $i$ and $A_i^j = 1$. Otherwise, to separate $i$ from $j$ would require a key in between with a higher priority. We therefore have that $j$ is an ancestor of $i$ exactly when it has a priority greater than all elements from $i$ to $j$ (inclusive on both sides).

Therefore $j$ is an ancestor of $i$ if and only if it has the highest priority of the keys between $i$ and $j$, inclusive. Because priorities are selected randomly, there a chance of $1/(|j-i|+1)$ that $A_i^j = 1$ and we have $\mathbf{E}\left[A_i^j\right] = \frac{1}{|j-i|+1}$. (Note that if we include the probability of either $j$ being an ancestor of $i$ or $i$ being an ancestor of $j$ then the analysis is identical to quicksort. Think about why.)

Now we have

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

Recall that the harmonic number is $H_n = \sum_{i=1}^{n} \frac{1}{n}$. It has the following bounds: $\ln n < H_n < \ln n + 1$, where $\ln n = \log_e n$. Notice that the expected depth of a key in the treap is determined solely by it relative position in the sorted keys.

**Exercise 15.17.** *Including constant factors how does the expected depth for the first key compare to the expected depth of the middle ($i = n/2$) key?*

**Theorem 15.18.** *For treaps the cost of* `join`$(T_1, m, T_2)$ *returning* $T$ *and of* `split`$(T, (k, v))$ *is* $O(\log |T|)$ *expected work and span.*

*Proof.* The `split` operation only traverses the path from the root down to the node at which the key lies or to a leaf if it is not in the tree. The work and span are proportional to this path length. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$.

For `join`$(T_1, m, T_2)$ the code traverses only the right spine of $T_1$ or the left spine of $T_2$. Therefore the work is at most proportional to the sum of the depth of the rightmost key in $T_1$ and the depth of the leftmost key in $T_2$. The work of `join` is therefore the sum of the expected depth of these nodes. Since the resulting treap $T$ is an interleaving of these spines, the expected depth is bound by $O(\log |T|)$. □

## Expected overall depth of treaps

Even though the expected depth of a node in a treap is $O(\log n)$, it does not tell us what the expected maximum depth of a treap is. As you have saw in lecture 15, $\mathbf{E}\left[\max_i\{A_i\}\right] \neq$

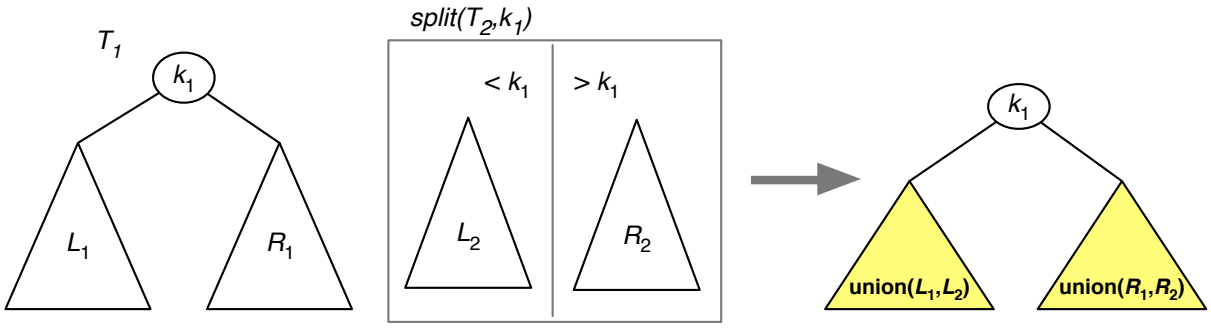



Figure 15.2: Taking the union of the elements of two trees.

$\max_i\{\mathbf{E}[A_i]\}$. As you might surmise, the analysis for the expected depth is identical to the analysis of the expected span of randomized quicksort, except the recurrence uses 1 instead of $c \log n$. That is, the depth of the recursion tree for randomized quicksort is $D(n) = D(Y_n) + 1$, where $Y_n$ is the size of the larger partition. Thus, the expected depth is $O(\log n)$.

It turns out that is possible to say something stronger: For a treap with $n$ keys, the probability that any key is deeper than $10 \ln n$ is at most $1/n$. That is, for large $n$ a treap with random priorities has depth $O(\log n)$ with *high probability*. It also implies that randomized quicksort $O(n \log n)$ work and $O(\log^2 n)$ span bounds hold with high probability.

Being able to put high probability bounds on the runtime of an algorithm can be critical in some situations. For example, suppose my company DontCrash is selling you a new air traffic control system and I say that in expectation, no two planes will get closer than 500 meters of each other—would you be satisfied? More relevant to this class, let's say you wanted to run 1000 jobs on 1000 processors and I told you that in expectation each finishes in an hour—would you be happy? How long might you have to wait?

There are two problems with expectations, at least on their own. Firstly, they tell us very little if anything about the variance. And secondly, as mentioned in an earlier lecture, the expectation of a maximum can be much higher than the maximum of expectations. The first has implications in real time systems where we need to get things done in time, and the second in getting efficient parallel algorithms (e.g., span is the max span of the two parallel calls). Proving these high probability bounds is beyond the scope of this course.

## 15.6 Union

Let's now consider a more interesting operation: taking the union of two BSTs. Note that this differs from `join` since we do not require that all the keys in one appear after the keys in the other. The following algorithm implements the union function using just `expose`, `split`, and `join` and is illustrated in Figure 15.2.

**Algorithm 15.19** (Union of two trees)**.**

```
1  function union(T1, T2) =
2    case expose(T1) of
3      NODE ⇒ T2
4    | SOME(L1, (k1, v1), R1) ⇒
5        let
6          val (L2, v2, R2) = split(T2, k1)
7          val (L, R) = (union(L1, L2) ‖ union(R1, R2))
8        in
9          join(L, SOME(k1, v1), R)
10       end
```

For simplicity, this version returns the value from $T_1$ if a key appears in both BSTs. Notice that *union* uses *split* and *join*, so it can be used for any BST tat support these two operations.

We'll analyze the cost of *union* next. The code for set intersection and set difference is quite similar.

## Cost of Union

In the 15-210 library, *union* and similar functions (e.g., *intersection* and *difference* on sets and *merge*, *extract* and *erase* on tables) have expected $O(m \log(1 + \frac{n}{m}))$ work, where $m$ is the size of the smaller input and $n$ the size of the larger one. We will see how this bound falls out very naturally from the `union` code.

To analyze *union*, we'll first assume that the work and span of *split* and *join* is proportional to the depth of the input tree and the output tree, respectively. In a reasonable implementation, these operations traverse a path in the tree (or trees in the case of `join`). Therefore, if the trees are reasonably balanced and have depth $O(\log n)$, then the work and span of *split* on a tree of $n$ nodes and *join* resulting in a tree of $n$ nodes is $O(\log n)$. Indeed, most balanced trees have $O(\log n)$ depth. This is true both for red-black trees and treaps.

The `union` algorithm we just wrote has the following basic structure. On input $T_1$ and $T_2$, the function $\mathit{union}(T_1, T_2)$ performs:

1. 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 $R_2$.
2. Recursively find $L_u = \mathit{union}(L_1, L_2)$ and $R_u = \mathit{union}(R_1, R_2)$.
3. Now $\mathit{join}(L_u, k_1, R_u)$.

We'll begin the analysis by examining the cost of each `union` call. Notice that each call to *union* makes one call to *split* costing $O(\log |T_2|)$ and one to *join*, each costing $O(\log(|T_1| + |T_2|))$. To ease the analysis, we will make the following assumptions:

1. $T_1$ it is perfectly balanced (i.e., `expose` returns subtrees of size $|T_1|/2$),
2. Each time a key from $T_1$ splits $T_2$, it splits the tree exactly in half, and
3. without loss of generality let $|T_1| \leq |T_2|$.

Later we will relax these assumptions.

With these assumptions, however, we can write a recurrence for the work of `union` as follows:

$$W(|T_1|, |T_2|) \leq 2W(|T_1|/2, |T_2|/2) + O(\log(|T_1| + |T_2|)),$$

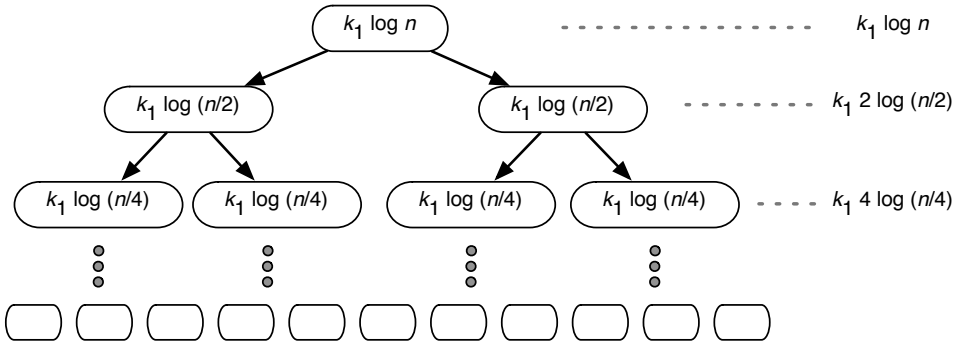
and

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

This recurrence deserves more explanation: When $|T_1| > 1$, `expose` gives us a perfect split, resulting in a key $k_1$ and two subtrees of size $|T_1|/2$ each—and by our assumption (which we'll soon eliminate), $k_1$ splits $T_2$ perfectly in half, so the subtrees `split` that produces have size $|T_2|/2$.

When $|T_1| = 1$, we know that `expose` give us two empty subtrees $L_1$ and $R_1$, which means that both $\texttt{union}(L_1, L_2)$ and $\texttt{union}(R_1, R_2)$ will return immediately with values $L_2$ and $R_2$, respectively. Joining these together with $T_1$ costs at most $O(\log(|T_1| + |T_2|))$. Therefore, when $|T_1| = 1$, the cost of `union` (which involves one `split` and one `join`) is $O(\log(1 + |T_2|))$.

Let $m = |T_1|$ and $n = |T_2|, m < n$ and $N = n + m$. If we draw the recursion tree that shows the work associated with splitting $T_2$ and joining the results, we obtain the following:




There are several features of this tree that's worth mentioning: First, ignoring the somewhat-peculiar cost in the base case, we know that this tree is leaf-dominated. Therefore, excluding the cost at the bottom level, the cost of `union` is $O(\#\text{ of leaves})$ times the cost of each leaf.

*But how many leaves are there? And how deep is this tree?* To find the number of leaves, we'll take a closer look at the work recurrence. Notice that in the recurrence, the tree bottoms out when $|T_1| = 1$ and before that, $T_1$ always gets split in half (remember that $T_1$ is perfectly balanced). Nowhere in there does $T_2$ affects the shape of the recursion tree or the stopping condition. Therefore, this is yet another recurrence of the form $f(m) = f(m/2) + O(...)$, which means that *it has $m$ leaves and is $(1 + \log_2 m)$ deep.*

Next, we'll determine the size of $T_2$ at the leaves. Remember that as we descend down the recursion tree, the size of $T_2$ gets halved, so the size of $T_2$ at a node at level $i$ (counting from 0) is $n/2^i$. But we know already that leaves are at level $\log_2 m$, so the size of $T_2$ at each of the leaves is

$$n/2^{\log_2 m} = \frac{n}{m}.$$

Therefore, each leaf node costs $O(\log(1 + \frac{n}{m}))$. Since there are $m$ leaves, the whole bottom level costs $O(m \log(1 + \frac{n}{m}))$. Hence, if the trees satisfy our assumptions, we have that `union` runs in $O(m \log(1 + \frac{n}{m}))$ work.

**Removing An Assumption:** Of course, in reality, our keys in $T_1$ won't split subtrees of $T_2$ in half every time. But it turns out this only helps. We won't go through a rigorous argument, but if we keep the assumption that $T_1$ is perfectly balanced, then the shape of the recursion tree stays the same. What is now different is the cost at each level. Let's 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 $n_1, \ldots, n_k$, where $\sum_j n_j = n$. Then, the total cost for this level is

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

where we used the fact that the logarithm function is concave[2]. Thus, the tree remains leaf-dominated and the same reasoning shows that the total work is $O(m \log(1 + \frac{n}{m}))$.

Still, in reality, $T_1$ 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. We'll end by remarking that as described, the span of `union` is $O(\log^2 n)$, but this can be improved to $O(\log n)$ by changing the the algorithm slightly.

In summary, `union` can be implemented in $O(m \log(1 + \frac{n}{m}))$ work and span $O(\log n)$. The same holds for the other similar operations (e.g. `intersection`).

# Summary

Earlier we showed that randomized quicksort has worst-case expected $O(n \log n)$ work, and this expectation was independent of the input. That is, there is no bad input that would cause

[2]Technically, we're applying the so-called Jensen's inequality.

the work to be worse than $O(n \log n)$ all the time. It is possible, however, (with extremely low probability) we could be unlucky, and the random chosen pivots could result in quicksort taking $O(n^2)$ work.

It turns out the same analysis shows that a deterministic quicksort will on average have $O(n \log n)$ work. Just shuffle the input randomly, and run the algorithm. It behaves the same way as randomized quicksort on that shuffled input. Unfortunately, on some inputs (e.g., almost sorted) the deterministic quicksort is slow, $O(n^2)$, every time on that input.

Treaps take advantage of the same randomization idea. But a binary search tree is a dynamic data structure, and it cannot change the order in which operations are applied to it. So instead of randomizing the input order, it adds randomization to the data structure itself.

.