Chapter 10

Binary Search Trees

Searching is one of the most important operations in computer science. Of the many search data structures that have been designed and are used in practice, search trees, more specifically balanced binary search trees, occupy a coveted place because of their broad applicability to many different sorts of problems. For example, in this course , we rely on binary search trees to implement set and table abstract data types (Chapter 11), which are then used in the implementation of many algorithms, including for example graph algorithms.

Question 10.1. *Can't we use sequences for searching?*

If we are interested in searching a static or unchanging collection of elements, then we can use a simpler data structure such as sequences. For example, if we use array-based implementation and cost specification, we can implement an efficient search function by representing our collection as a sorted sequence and by using binary search, which would require indexing into the sequence logarithmically many number of times. If, however, we want to support a dynamic collection, where for example, we insert new elements and delete existing elements, sequences would require linear work. Binary search trees (BSTs) make it possible to compute with dynamic collection by using insertions, deletions, as well as searches all in logarithmic number of tree operations.

In courses on sequential algorithms, presentation of binary search trees focus on functions for inserting elements, for deleting elements, and for searching. While these functions are important, they are not sufficient for parallelism, because they perform singleton updates. In this course , we also consider aggregate update operations, such as union and difference, which can be used to insert and delete (respectively) many elements at once.

The rest of this chapter is organized as follows. We first define binary search trees (Section 10.1) and present an ADT for them (Section 10.2). We then present a parameteric implementation of the ADT (Section 10.4) that implements the BST ADT by using only two operations that `split` and `join` that respectively split a tree at a given key and join two trees.

Insj Section 10.5, we present a cost specification based on the parameterized implementation, which achives strong bounds as long as the `split` and `join` operations have logarithmic work and span. As a result, we are able to reduce the problem of implementing the BST ADT to the problem of implementing just the function `split` and `join`. We finish the chapter by presenting one approach to implementing `split` and `join` by using treaps (Section 10.6). We mention other possible implementation techniques in Section 10.3

10.1 Preliminaries

We start with some basic definitions and terminology involving rooted and binary search trees. Recall first that a rooted tree is a tree with a distinguished root node that can be used to access all other nodes (Definition 2.6). A full binary tree is a rooted tree, where each node is either a leaf or an internal node with a left and a right child (Definition 10.2).

Definition 10.2 (Binary Tree)**.** *A full* 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.*

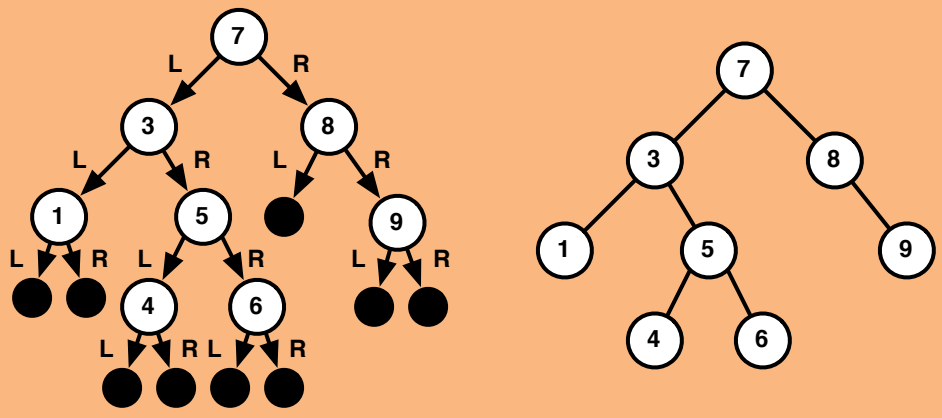
A binary search tree is a full binary tree, where each internal node u has a unique key k such that each node in its left subtree has a key less than k and each node in its right subtree has a key greater that x (Definition 10.3). Formally, we can define binary search trees as follows.

Definition 10.3 (Binary Search Tree (BST))**.** *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, $k(u) > k(v)$*

In the definition, conditions 2 and 3 are referred to as the BST property*. We often refer to the elements of S in a BST as keys, and use $\mathsf{dom}(T)$ to indicate the domain (keys) in a BST T. The* size *of a BST is the number of keys in the tree, i.e. $|S|$.*

Example 10.4. *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 use this convention in future figures.*

10.2 The BST Abstract Data Type

ADT 10.5 describes an ADT for BSTs parameterized by a totally ordered key set. We briefly describe this ADT and present some examples. As we describe in the rest of this section, the BST ADT can be implemented in many ways. In order to present concrete examples, we assume an implementation but do not specify it.

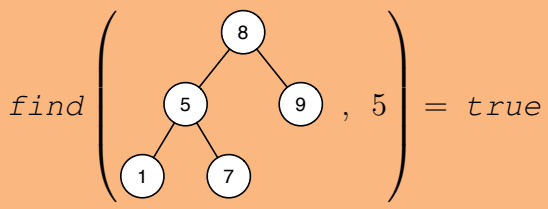
The ADT supports the constructor operations `empty` and `singleton` for creating an empty BST and BST with a single key. The function `find` searches for a given key and returns a boolean indicating success.

Abstract Data Type 10.5 (BST). *For a universe of totally ordered keys $\mathbb{K}$, the BST ADT consists of a type $\mathbb{T}$ representing a power set of keys and the functions specified as follows. For the specification, for a tree T $[|T|]$ denotes the set of keys in the tree.*

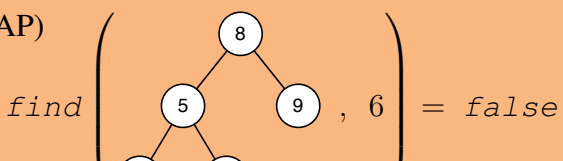
`empty`	:	$\mathbb{T}$
`empty`	=	T *where* $[|T|] = \emptyset$
`singleton`	:	$\mathbb{K} \to \mathbb{T}$
`singleton`(k)	=	T *where* $[|T|] = \{k\}$.
`find`	:	$(\mathbb{T} \times \mathbb{K}) \to \mathbb{B} \times \mathbb{T})$
`find`(T, k)	=	`true` *if and only if* $k \in [|T|]$
`insert`	:	$(\mathbb{T} \times \mathbb{K}) \to \mathbb{T}$
`insert`(T, k)	=	T' *where* $[|T|] = [|T|] \cup \{k\}$
`delete`	:	$(\mathbb{T} \times \mathbb{K}) \to \mathbb{T}$
`delete`(T, k)	=	T' *where* $[|T'|] = [|T|] \setminus \{k\}$.
`union`	:	$(\mathbb{T} \times \mathbb{T}) \to \mathbb{T}$
`union`(T_1, T_2)	=	T' *where* $[|T'|] = [|T_1|] \cup [|T_2|]$
`intersection`	:	$(\mathbb{T} \times \mathbb{K}) \to \mathbb{T}$
`intersection`(T_1, T_2)	=	T' *where* $[|T'|] = [|T_1|] \cap [|T_2|]$
`diff`	:	$(\mathbb{T} \times \mathbb{K}) \to \mathbb{T}$
`diff`(T_1, T_2)	=	T' *where* $[|T'|] = [|T_1|] \setminus [|T_2|]$
`split`	:	$(\mathbb{T} \times \mathbb{K}) \to (\mathbb{T} \times \mathbb{B} \times \mathbb{T})$
`split`(T, k)	=	(T_1, b, T_2) *where* $[|T_1|] = \{k' : k' \in [|T|], k' < k\}$, $[|T_2|] = [|T|] \setminus [|T_1|]$, *and* $b =$ `true` *if and only if* $k \in [|T|]$
`join`	:	$(\mathbb{T} \times \mathbb{T}) \to \mathbb{T}$
`join`(T_1, T_2)	=	T *where* $[|T|] = [|T_1|] \cup [|T_2|]$

Example 10.6. *Searching in BST's illustrated.*

- *Searching for* 5 *in the input tree returns* `true`.

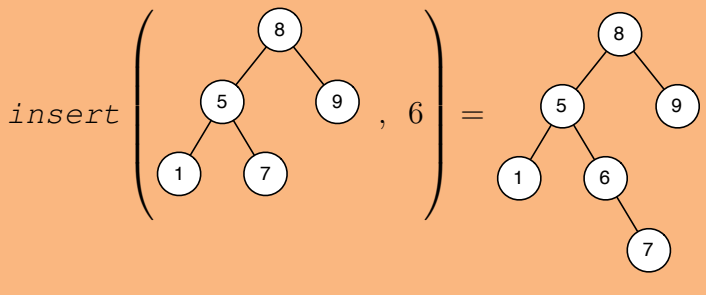


- *Searching for* 6 *in the input tree returns* `false`.

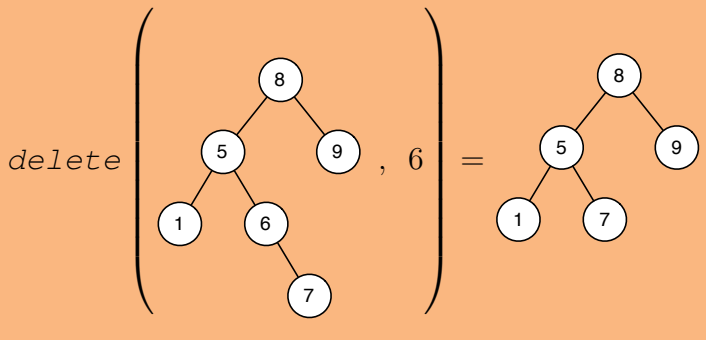


The functions `insert` and `delete` insert and delete a given key into or from the BST.

Example 10.7. *Inserting the key* 6 *into the input tree returns a new tree including* 5.



Example 10.8. *Deleting the key* 6 *from the input tree returns a tree without it.*

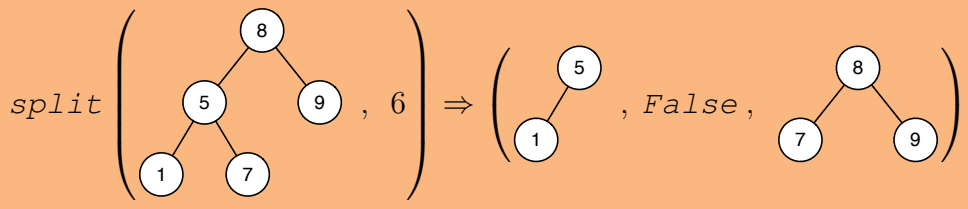


The function `union` takes two BST's and returns a BST that contains all the keys in them; it can be viewed as an aggregate insert operation. The function `intersection` takes two BST's and returns a BST that contains the keys common in both. The function `diff` takes two BST's T_1 and T_2 and returns a BST that contains the keys in T_1 that are not in T_2. The function `diff` is effectively an aggregate delete operation.

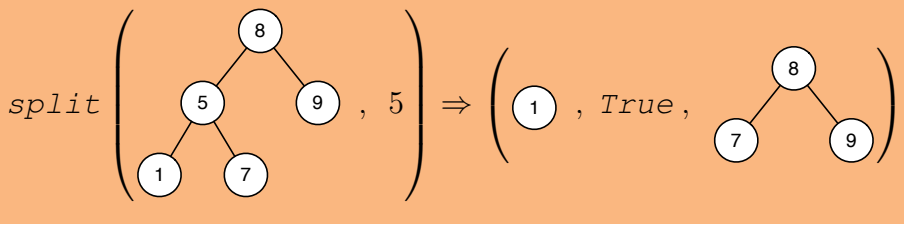
The function `split`(T, k) takes a tree T and a key k and splits T into two trees: one consisting of all the keys of T less than k, and another consisting of all the keys of T greater than k. It also returns a Boolean indicating whether k appears in T. Exactly how `split` reorganizes the input tree to produce its output can differ from on implementation to another: as long as the resulting trees contain the keys less than k and greater than k, exactly how the trees are organized remains unspecified in the ADT.

Example 10.9. *The function* `split` *illustrated.*

- *Splitting the input tree at* 6 *yields two trees, consisting of the keys less that* 6 *and those greater that* 6*, indicating also that* 6 *is not in the input tree.*

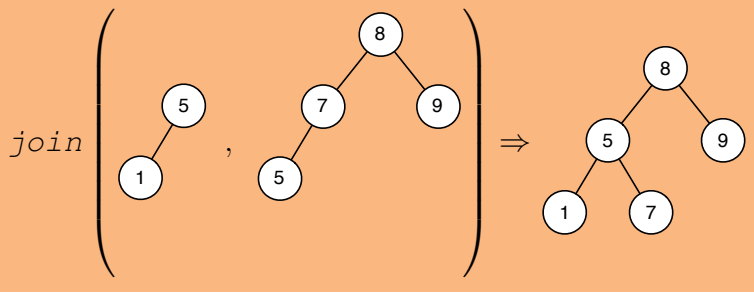


- *Splitting the input tree at* 5 *yields two trees, consisting of the keys less than* 5 *and those greater than* 5*, indicating also that* 5 *is found in the input tree.*



The function $\texttt{join}(T_1, T_2)$ takes two trees T_1 and T_2 such that all the keys in T_1 are less than the keys in T_2. The function returns a tree that contains all the keys in T_1 and T_2. Exactly how `join` constructs the result tree could differ between different implementations, as long as the result is a valid BST and it contains the union of the keys.

Example 10.10. *The function* `join` *illustrated.*



10.3 Implementation via Balancing

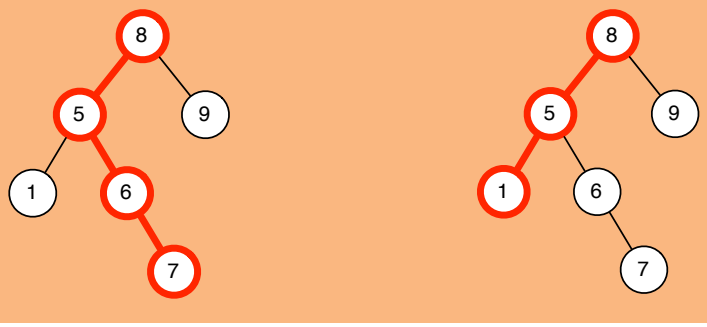
The main idea behind BST's is to organize the keys such that

1. a specific key can be searched by following a branch in the tree by doing key comparisons along the way, and

2. a range of keys in a subtree can be operated on (e.g., moved) by performing constant work.

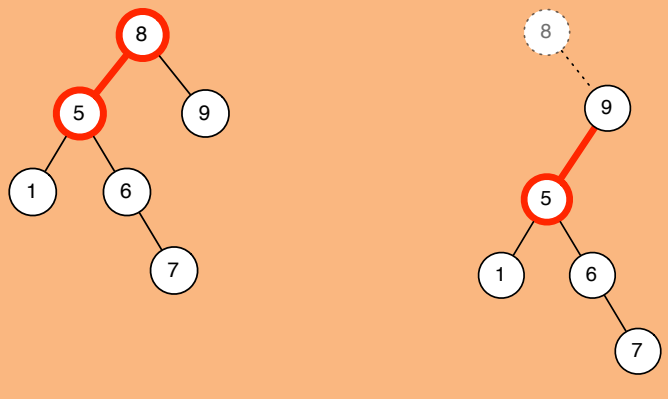
To see how we can perform a search in the tree, consider searching for a key k. We can start at the root r and if k equals the key at the root, $k(r)$, then we have found our key, otherwise if $k < k(r)$, then we know that k cannot appear in the right subtree, so we only need to search the left subtree, and if $k > k(r)$, then we only have to search the right subtree. Continuing the search 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 starting at the root.

Example 10.11. *A successful search for* 7 *and an unsuccessful search for* 4 *in the given tree. Search paths are highlighted.*



To see how we can operate on a range of keys note first that each subtree in a binary tree contains the keys that all the keys within a specific range. We can find such a range by performing a search as described— in fact, a search as described identifies a possibly empty range. Once we find a range of keys, we can operate on them as a group by handling the root. For example, we can move the whole subtree to another location by linking the root to another parent.

Example 10.12. *In the given example tree we can handle all the keys that are less that* 8 *by holding the subtree rooted at* 5*, the left child of* 8*. For example, we can make* 5 *the left child of* 9 *and delete* 8 *from the tree. Note that if* 8 *remains in the tree, the resulting tree would not be a valid BST.*



Exercise 10.13. *Design an algorithm for inserting a given key into a BST.*

Exercise 10.14. *Design an algorithm for deleting a given key from a tree.*

By taking advantage of the ability to find keys and more generally ranges of keys by walking a path in the BST and the ability to move ranges by performing constant work, it turns out to be possible to implement all the operations in the BST ADT efficiently as long as the paths that we have to walk are not too long. One way to guarantee absence of long paths is to make sure that the tree remains balanced, i.e., the longest paths have approximately the same length. A binary tree is defined to be *perfectly balanced* if it has the minimum possible height. For a binary search tree over a set S, a perfectly balanced tree has height exactly $\lceil \log_2(|S| + 1) \rceil$.

Ideally we would use a perfectly balanced tree. Indeed if we never make changes to the tree, we could balance it once up front and it would remain balanced. However, if we want to update the tree by, for example, inserting new keys or combining two trees, then maintaining such perfect balance is costly. It turns out to be impossible, for example, to maintain a perfectly balanced tree while allowing insertions in $O(\log n)$ work. BST data structures therefore aim to keep approximate or near balanced instead of perfect balance. We refer to a BST data structure as *balanced* or *near balanced* if all trees with n elements have height $O(\log n)$, perhaps in expectation or with high probability.

There are many balanced BST data structures. Most 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). Here we list a few such data structures:

1. *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, which in turn implies near balance.

2. *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.

3. *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 $0 < \alpha \leq \frac{1}{2}$. The BB stands for bounded balance, and adjusting α gives a tradeoff between search and update costs.

4. *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.

5. *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.

There are dozens of other balanced BST data structures (e.g. scapegoat trees and skip lists), as well as many that allow larger degrees, including 2–3 trees , brother trees, and B trees. In this chapter we will cover treaps, red-black trees, and briefly AVL trees.

10.4 A Parametric Implementation

We describe what might be called a minimalist implementation of the BST ADT based on just two functions, `split` and `join`. Since the implementation depends on just these two functions, we think of it as a parametric implementation.

Data Structure 10.15 illustrates an implementation of the BST ADT using by just split and join. We define the tree as a data type indicating that the tree is either a leaf or an internal node with a left and right subtree and a key. For the implementation, we assume that `split` and `join` are given.

As an auxilary function, we first define function `joinM` which takes two trees T_1 and T_2 and a "middle" key k that is sandwiched between the two trees—that is k is greater than all the keys in T_1 and less than all the keys in T_2—and returns a tree that contais all the keys in T_1 and T_2 as well as k.

We implement `find` with a `split` by taking advantage of the fact that `split` indicates whether the key used for splitting is found in the tree or not. To implement `insert` of a key k into a tree T, we first `split` the tree T at k and then join the two returned trees along with key k using `joinM`. To implement `delete` of a key k from a tree T, we first `split` the tree T at k and then join the two returned trees with `join`. If the key k was found, this gives us

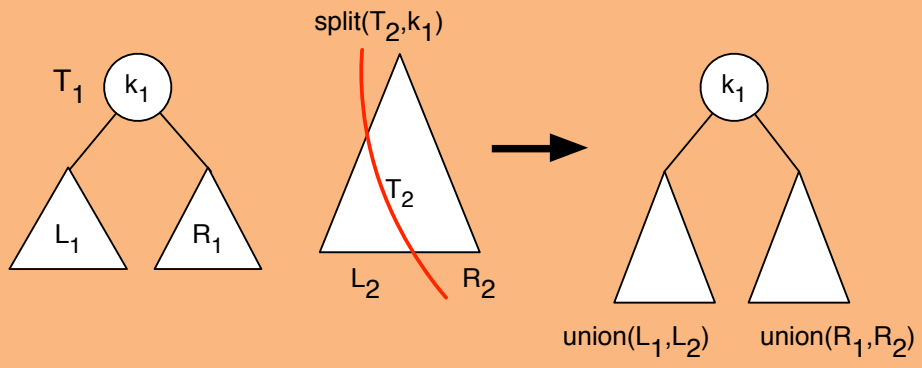
Data Structure 10.15. *Implementing the BST ADT with* `split` *and* `join`*]*

```
datatype 𝕋 = Leaf | Node of (𝕋 × 𝕂 × 𝕋)

function split (T, k) = ... (* as given *)
function join (T1, T2) = ... (* as given *)
function joinM (T1, k, T2) =  join (T1, join (singleton k, T2))

let empty = Leaf
function singleton(k) = Node(Leaf,k,Leaf)

function find(T, k) = let (_,v,_) = split(T, k)  in v end

function insert(T, k) = let (L, _, R) = split(T, k) in joinM(L, k, R) end

function delete(T, k) = let (L, _, R) = split(T, k) in join(L, R) end

function union(T1, T2) =
  case (T1, T2) of
    (Leaf,_) ⇒ T2
  | (_,Leaf) ⇒ T1
  | (Node(L1, k1, R1),_) ⇒
    let  val (L2, _, R2) = split(T2, k1)
         val (L, R) = (union(L1, L2) || union(R1, R2))
    in joinM (L, k1, R) end

function intersect(T1, T2) =
  case (T1, T2) of
    (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


function diff(T1, T2) =
  case (T1, T2) of
    (Leaf,_) ⇒ Leaf
  | (_,Leaf) ⇒ T1
  | (Node(L1, k1, R1),_) ⇒
    let (L2, b, R2) = split(T2, k1)
        (L, R) = (diff (L1, L2) || diff (R1, R2))
    in if b then join (L, R) else joinM (L, k1, R) end
```

a tree that does not contain the k; otherwise we obtain a tree of the same set of keys (though the structure of the tree may be different internally depending on the implementation of `split` and `join`).

To implement `union`(T_1, T_2) in parallel we use a divide-and-conquer approach. The idea is to split both trees at some key k, recursively union the two parts with keys less than k, and the two parts with keys greater than k and then join them. How do we find the key k? One way is simply to use the key at the root of one of the two trees, for example the first tree, and split the second tree with it. This idea leads to the implementation given in Data Structure 10.15.

Example 10.16. *`union`(T_1, T_2) by using the key k_1 at the root of T_1 to split T_2 and recursively unioning the parts less than k_1 and the parts greater than k_1.*



To implement `intersection`(T_1, T_2) in parallel we use a divide-and-conquer approach similar to the implementation of `join`. As in `join`, we split both trees by using the key at the roof of one, and compute intersections recursively. We then compute the result by joining the results from the recursive calls and including in the join the key used for splitting if the key is contained in both trees.

Question 10.17. *How do we know that we account for all possible shared keys when we divide the inpet trees into two and calculate intersections recursively? What if a key on the left matches a key on the right half of the other tree?*

Since the trees are BST's, checking for the intersections of left and right subtrees recursively and computing the intersection based on their results is guaranteed to find all common keys because a key on the left side of one tree cannot be on the right side of the tree obtained by splitting the other tree.

Cost Specification 10.18 (BST's). *We define the* **BST** *cost specification as follows. The variables n and m are defined as $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$ when applicable.*

	Balanced BST	
	Work	Span
`empty`	1	1
`singleton`(v)	1	1
`find`(T, k)	$1 + \log \lvert T \rvert$	$1 + \log \lvert T \rvert$
`insert`(T, k)	$1 + \log \lvert T \rvert$	$1 + \log \lvert T \rvert$
`delete`*(T,k)*	$1 + \log \lvert T \rvert$	$1 + \log \lvert T \rvert$
`union`(T_1, T_2)	$m \cdot \log \frac{n}{m}$	$1 + \log n$
`intersect`(T_1, T_2)	$m \cdot \log \frac{n}{m}$	$1 + \log n$
`diff`(T_1, T_2)	$m \cdot \log \frac{n}{m}$	$1 + \log n$
`split`(T, k)	$1 + \log \lvert T \rvert$	$1 + \log \lvert T \rvert$
`join`(T_1, T_2)	$\log(\lvert T_1 \rvert + \lvert T_2 \rvert)$	$\log(\lvert T_1 \rvert + \lvert T_2 \rvert)$

10.5 Cost Specification

We now consider the cost specifications for the BST ADT. Even though there are many ways in which we can implement an efficient BST data structure that mathes our ADT, many of these implementation more or less match the same cost specification, with the main difference being whether the bounds are worst-case, expected case (probabilistic), or amortized. These implementations all use balancing techniques to ensure that the depth of the BST remains $O(\log n)$, where n is the number of keys in the tree. For the purposes specifying the costs, we don't distinguish between worst-case, amortized, and probabilistic bounds, because we can always rely on the existence of an implementation that matches the desired cost specification. In other words, our cost-specifications can be viewed as worst-case bounds. When using specific data structures that match the specified bounds in an amortized or randomized sense, we will try to be careful when specifying the bounds.

Cost Specification 10.5 shows the costs for the BST ADT as can be realized by several balanced BST data structures such as treaps (in expectation), red-black trees (in the worst case), and splay trees (amortized). As may be expected the cost of `empty` and `singleton` are constant.

For the rest of the operations, we will justify the cost bounds by assuming the existince of

logarithmic time `split` and `join` operations, and by using our parametric implementation described above. The work and span costs of `find`, `insert`, and `delete` are determined by the `split` and `join` operation and are thus logarithmic in the size of the tree.

The cost bounds on `union`, `intersection`, and `diff`, which are similar are more difficult to see. Let's analyze the cost for `union` based on our parametric implementation. It is easy to apply a similar analysis to `intersection` and `diff`.

Consider now a call to `union` with parameters T_1 and T_2. To simplify the analysis, we will make the following assumptions:

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

Later we will relax these assumptions. Let us define $m = |T_1|$ and $n = |T_2|$ (recall the size of a tree is the number of keys in it). With these assumptions and examining the algorithm we can then write the following recurrence for the work of `union`:

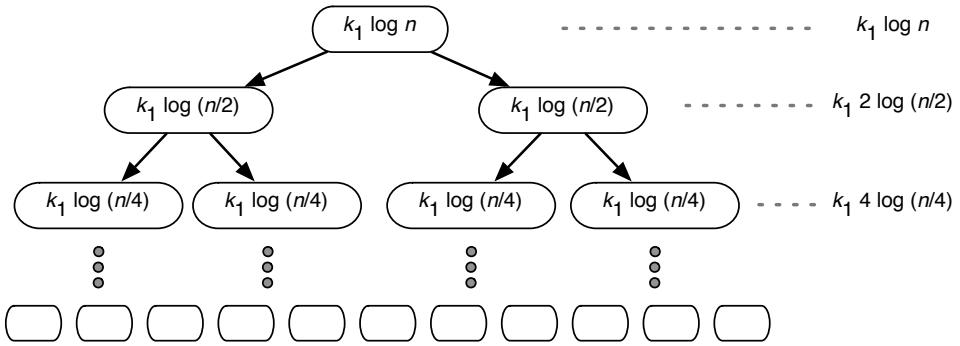
$$\begin{aligned} W_{\text{union}}(m,n) &\leq 2W_{\text{union}}(m/2,n/2) + W_{\text{split}}(n) + W_{\text{join}}(n+m) + O(1) \\ &\leq 2W_{\text{union}}(m/2,n/2) + O(\log n)\ . \end{aligned}$$

The size for join is the sum of the two sizes, $m+n$, but since $m \leq n$, $O(\log(n+m))$ is equivalent to $O(\log n)$. We also have the base case

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

The final inequality is since $2W_{\text{union}}(0,n) = O(1)$.

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 $O(\#$ 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 will take a closer look at the work recurrence. Notice that in the recurrence, the tree bottoms out when $m = 1$ and before that, m 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 will determine the size of T_2, i.e. n, at the leaves. Well we had m keys in T_1 to start with, and they chopped T_2 evenly, so each part at the base case will have size $\frac{n}{m}$. Therefore, each leaf node costs $O(\log(1 + \frac{n}{m}))$ (the $1+$ is needed to deal with the case that $n = m$). 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.

Of course, in reality, our keys in T_1 won't split subtrees of T_2 in half every time. But it turns out that any unevenness in the splitting only helps reduce the work—i.e., the perfect split is the worst case. 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 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 $n_1, \ldots, n_k$, 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[1]. 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 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 algorithm slightly.

In summary, `union` can be implemented in $O(m \log(1 + \frac{n}{m}))$ work and span $O(\log n)$.

Essentially the same analysis applies to `intersection` and `diff`, whose structure is essentially the same as `union` expect for an additional constant work and span for the conditional (`if`) expression.

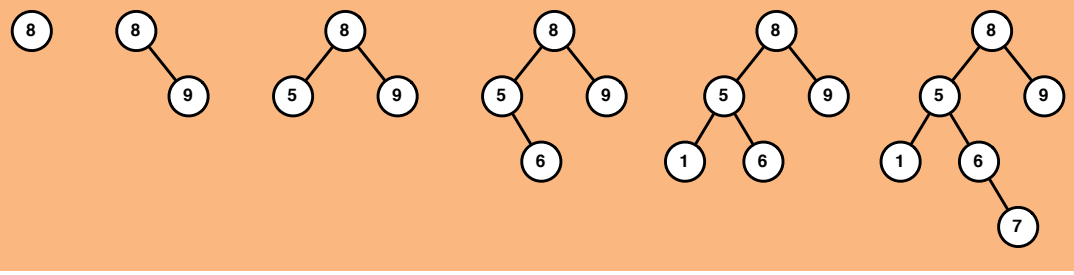
[1] Technically, we're applying the so-called Jensen's inequality.

10.6 Treaps

Our parametric implementation established an interesting fact: to implement the BST ADT efficiently, we only need to provide efficient `split` and `join` operations. In this section, we present a data structure called *treaps* that can support `split` and `join` operations in expected logarithmic work and span. Treaps achieve their efficiency by maintaining BST's that are probabilistically balanced. Of the many balanced BST data structures, treaps are likely the simplest, but, since they are randomized, they only guarantee near balance with high probability.

To see the idea behind treaps, let's first discuss how we can map a sequence of keys to a BST. Given S, a sequence of unique keys, let's start with an empty BST and insert the keys in S into the BST one by one (from left to right) by using the following algorithm. For each k, perform a search for the key k in the current BST and let u be the leaf note where the (unsuccessful) search terminates. To insert k into the tree replace u with a new node with the key k.

Example 10.19. *We can map the sequence* $\langle 8, 9, 5, 6, 1, 7 \rangle$ *to a BST by inserting the keys from left to right.*



Question 10.20. *What can we say about the height of such a tree?*

Note now that the heght of the BST will be directly determined by the input sequence. Specifically, the specific order—or permutation—in which the keys are inserted will determine the height. For most permutations, the tree will be reasonably well balanced because we get an unbalanced tree only in cases where an element partitions the following keys unevenly. Given that we have many more even partitions for a given set of keys (many "middle" keys), many permutations create balanced trees.

Question 10.21. *Can we take advantage of this observation somehow?*

Recall that the BST ADT does not care about the specific structure of the BST but only the set of keys in the tree. We can take advantage of this by selecting a (uniformly) random permutation of the keys and constructing the BST based on this permutation. Since most permutations give us reasonably balanced trees, this should work well.

Question 10.22. *Suppose that we were given the keys in the sequence one by one instead of all at once. Can you think of a way to select a uniformly random permutation?*

Observe that we can select a uniformly random permutation even if we don't have all the keys: all we have to do is to assign a random priority to the keys as they arrive and build our BST by considering the keys in the priority order. By picking priorities randomly we essentially guarantee that we always build our tree on a uniformly randomly selected permutation. This is one of the main ideas behind treaps: treaps can be viewed as maintaining BST's on a uniformly random permutation of the keys in the tree. To achieve this "imitation of randomly ordered insertions" we associate a priority with each key and require the BST to be "heap ordered" with respect to priorities.

More precisely, we define treaps as follows.

Definition 10.23 (Treap)**.** *A treap is a binary search tree T over a set S along with a* priority *for each key given by*

$$p : \mathbb{K} \to \mathbb{Z} \, ,$$

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:

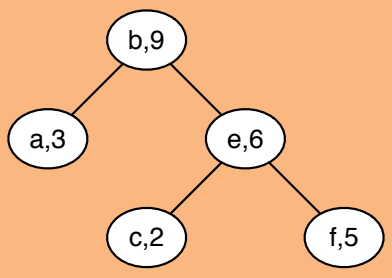
$$p(k(v)) \geq p(k(L(v))) \text{ and } p(k(v)) \geq p(k(R(v))),$$

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

Example 10.24. *The following key-priority pairs $(k, p(k))$,*

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

where the keys are ordered alphabetically, form the following treap:



since 9 is larger than 3 and 6, and 6 is larger than 2 and 5.

Exercise 10.25. *Prove that if the priorities are unique, then there is exactly one tree structure that satisfies the treap properties.*

So how do we assign priorities? As we briefly suggested in the informal discussion above, it turns out that if the priorities are selected uniformly randomly then the tree is guaranteed to be near balanced, i.e. $O(\log |S|)$ height, with high probability. We will show this shortly.

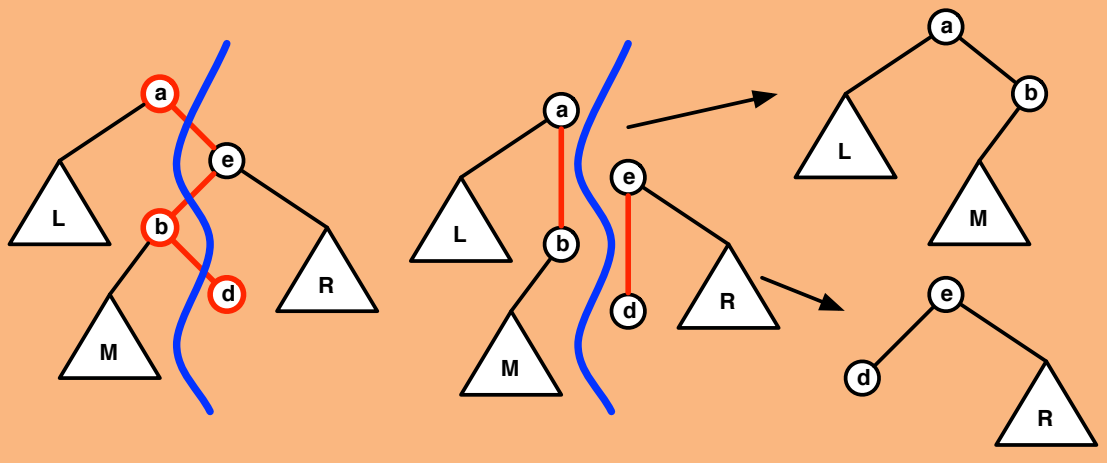
Question 10.26. *How can we maintain the heap order as we modify the tree?*

The second idea behind Treaps is to update the tree structure according to new priorities efficiently by performing local reorganizations. Based on our parameterized implementation, we can give an implementation for the BST ADT with treaps simple by implementing the `split` and `join` functions. Data Structure 10.27 shows an treap-based implementation for the BST ADT based on this idea. For the implementation we assume, without loss of generality, that the priorities are integers numbers. We present only the code for `split` and `join`; the rest of the implementation is essentially the same as in Data Structure 10.15 with the only exception that since the nodes now carry priorities, we will need to account for them as we pattern match on nodes and create new ones. In implementing the rest of the functions, there is no interesting operations on priorities: they simply follow the key of the node that they belong to.

To implement the function `singleton`, we rely on a function `randomInt`, which when called returns a (pseudo-)random number. Such functions are broadly provided by the basic libraries of many programming languages.

The `split` algorithm recursively traverses the tree from the root to the key k splitting along the path, and then when returning from the recursive calls, it puts the subtrees back together. When putting back the trees along the path being split through, the function does not have to compare priorities because `Node` on Lines 14 and 18, the priority p' is the highest priority in the input tree T and is therefore larger than the priorities of either of the subtrees on the left and right. Hence `split` maintains the heap property of treaps.

Example 10.28. *A `split` operation on a treap ad key c, which is not the treap. The `split` traverses the path $\langle a, e, b, d \rangle$ turning right at a and b (Line 16 of the Data Structure 10.27) and turning left at e and d (Line 12). The pieces are put back together into the two resulting trees on the way back up the recursion.*

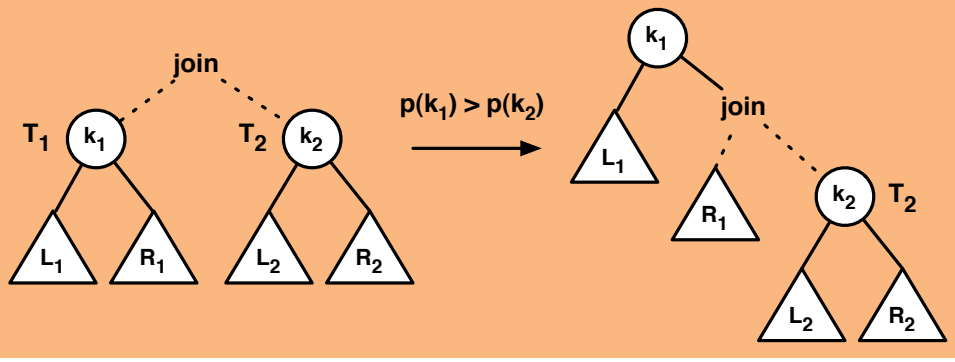


Data Structure 10.27 (Implementing BST with Treaps)**.**

```
1  datatype 𝕋 = Leaf | Node of (𝕋 × 𝕂 × ℤ × 𝕋)
2
3  let empty = Leaf
4
5  function singleton(k) = Node(Leaf,k,randomInt(),Leaf)
6
7  function split(T,k) =
8    case (T) of
9      Leaf ⇒ (Leaf, False, Leaf)
10   | Node(L,k′,p′,R) =
11       case compare(k,k′) of
12         LESS ⇒
13           let (L′,x,R′) = split(L,k)
14           in (L′,x,Node(R′,k′,p′,R)) end
15       | EQUAL ⇒ (L, True,R)
16       | GREATER ⇒
17           let (L′,x,R′) = split(R,k)
18           in (Node(L,k′,p′,L′),x,R′) end
19
20
21 function join(T1,T2) =
22   case (T1,T2) of
23     (Leaf, _) ⇒ T2
24   | (_ , Leaf) ⇒ T1
25   | (Node(L1,k1,p1,R1), Node(L2,k2,p2,R2)) ⇒
26       if (p1 > p2) then
27         Node(L1,k1,p1,join(R1,T2))
28       else
29         Node(join(T1,L2),k2,p2,R2)
30 end
```

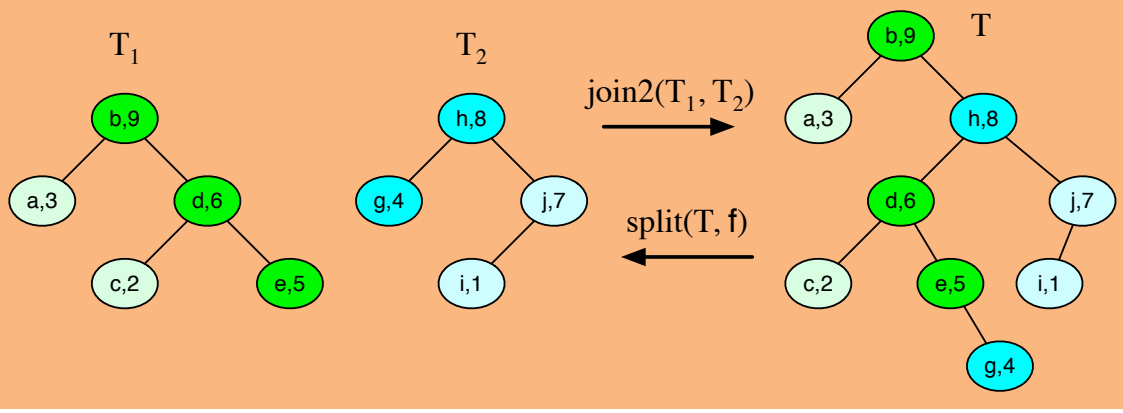
Unlike the implementation of *split*, the implementation of *join*(L, R) operates on priorities in order to ensure that the resulting treap satisfies the heap priority of treaps. Specifically, given two trees, *join* first compares the priorities of the two roots, making the larger priority the new root. It then recursively joins the treaps consisting of the other tree and the appropriate side of the new root. This is illustrated in the following example:

Example 10.29. *$\texttt{join}(T_1, T_2)$ on treaps. If $p(k_1) > p(k_2)$, then the function recurs with $\texttt{joinPair}(R_1, T_2)$ and the result becomes the right child of k_1.*



The path from the root to the leftmost node in a BST is called the *left spine*, and the path from the root to the rightmost node is called the *right spine*. The function $\texttt{join}(T_1, T_2)$ merges the right spine of T_1 with the left spine of T_2 based on the priority order. This ensures that the priorities are in decreasing order down the path.

Example 10.30. *$\texttt{join}(T_1, T_2)$ on treaps in more detail. The right spine of T_1 consisting of $(b, 9)$, $(d, 6)$ and $(e, 5)$ is merged by priority with the left spine of T_2 consisting of $(h, 8)$ and $(g, 4)$. Note that splitting the result with the key f will return the original two trees.*



We are now interested in the work for split and join. Each one does constant work on each recursive call. For $\texttt{split}$ each recursive call goes to one of the children, so the number of recursive calls is at most the height of T. For $\texttt{join}$ each recursive call either goes down one level in T_1 or one level in T_2. Therefore the number of recursive calls is bounded by the sum of the heights of the two trees. Hence the work of $\texttt{split}(T, k)$ is $O(h(T))$ and the work of $\texttt{join}(T_1, m, T_2)$ is $O(h(T_1) + h(T_2))$. This leaves us with the question of what is the height of a treap?

Analysis of randomized treaps. We now analyze the height of a treap assuming that the priorities are picked at random. To do this we will relate treaps to quicksort, which we analyzed in Chapter 9. In particular consider the following variant of quicksort.

Algorithm 10.31. *Treap Generating Quicksort*

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

This algorithm is almost identical to our previous quicksort except that it uses `Node` instead of `append` on line 9, `Leaf` instead of an empty sequence in the base case, and, since it is generating a set, it needs only keep one copy of the keys equal to the pivot.

The tree generated by `qsTree`(S) is the treap for S. This can be seen by induction. It is true for the base case. Now assume by induction it is true for the trees returned by the two recursive calls. The tree returned by the main call is then also a treap since the `pivot` has the highest priority, and therefore is correctly placed at the root, the subtrees and in heap order by induction, and because the keys in T_L are less than the pivot, and the keys in T_R are greater than the pivot, the tree has the BST property.

What does this tell us about the height of treaps? Well it tells us that the height of a treap is identical to the recursion depth of quicksort. In Chapter 9 we proved that if we pick the priorities at random, the recursion depth is $O(\log n)$ with high probability. Therefore we know that the height of a treap is $O(\log n)$ with high probability.

10.7 Augmenting Trees

Thus far in this chapter, the only interesting information that we stored in BST's were keys. While such trees can be interesting in many applications, we often wish to augment trees with more information. For example, we may want to pair a value with a key, or to pair a value with each node in the binary tree. In the latter case, the value associated with the node may change as the tree is re-organized.

Question 10.32. *How can we change the BST ADT to associate values with keys?*

To associate values with keys, we may need to change the arguments and return values for each function in the ADT. These changes are relatively small and straightforward. For example, operations such an `singleton` and `insert` will now take a value to be associated with the key under consideration. Similarly, `find` and `split` will need to return the value along with the key if the key is found, perhaps returning an optional value.

Question 10.33. *How can we change a BST data structure to associate values with keys?*

Similarly, storing values inside BST's in addition to keys requires relatively small changes to a data structure implementing the ADT. For example, to accommodate the key, we can change the BST data type to include a key-value pair instead of just a key. We then change the implementation of the other functions to pass the value around with the key as needed, making sure that a key-value pair is never separated. For function such as `find` and `split` that may need to return the value, we make sure to do so.

Associating a value with each key is probably the simplest form of augmentation. As a more complex augmentation, in addition to a value, we might want to associate with each node a value that reflects some property of the BST, for example, to perform additional operations.

As a motivating example, suppose that we wish to extend the BST ADT (ADT 10.5) with the following additional functions.

- Function `rank`(T, k) returns the rank of the key k in the tree, i.e., the number of keys in T that are less than or equal to k.

- Function `select`(T, i) returns the key with the rank i in T.

Question 10.34. *Can you implement these functions by using a BST?*

For the implementation, we want a way to count the number of nodes in a subtree. Let $|T|$ denote the size of (the number of nodes in) in a tree. We can implement these operations as follows.

Algorithm 10.35 (Rank)**.**

```
1  function rank(T, k) =
2    case (T) of
3      Leaf ⇒ 0
4    | Node(L, k', R) ⇒
5        case compare(k, k') of
6          LESS ⇒ rank (L, k)
7        | EQUAL ⇒ |L| + 1
8        | GREATER ⇒ |L| + 1 + rank(R, k)
9
10 function select (T, i) =
11   case (T) of
12     Leaf ⇒ raise exception OutOfRange
13   | Node (L, k, R) ⇒
14       case compare(i, |L| + 1) of
15         LESS ⇒ select (L, i)
16         EQUAL ⇒ k
17         GREATER ⇒ select (R, i − |L| − 1)
```

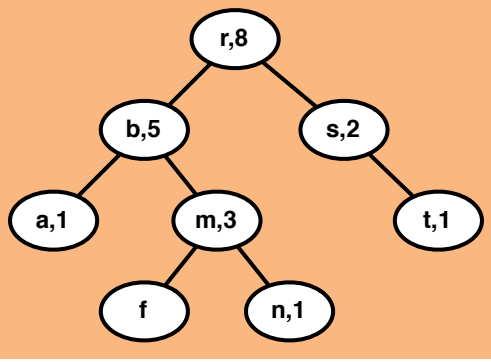
Question 10.36. *What is the work and span of these functions?*

For balanced trees such as with treaps, these functions require logarithmic span but without augmenting a BST, these functions require linear work because computing the size of a subtree takes linear time in the size of the subtree.

Question 10.37. *Can we compute size of subtrees more efficiently?*

If, however, we augment the tree so that each node in the tree keeps track of the number of nodes that are descendants of that node—in other words the size of the (subtree) rooted at that node—then the work and span both become logarithmic because computing the size of a subtree is now constant work operation.

Example 10.38. *An example BST, where keys are ordered lexicographically and the nodes are augmented with the sizes of subtrees.*



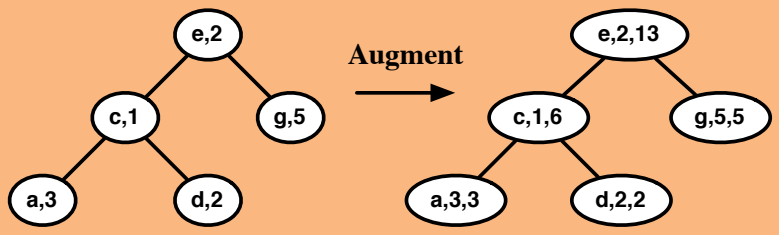
Question 10.39. *But how can we maintain the sizes of the subtrees as we perform various operations on the BST such as possibly aggregate insertions, deletions, splits, and joins?*

Maintaining the size of each subtree rooted at a node in a binary-search-tree implementation is not difficult. All we have to do is to update the size values as we reorganize the tree. Specifically, using our parametric implementation all we have to do is to update the `split` and `join` functions so that as they construct new nodes, they compute the size of the subtree rooted at that node by adding the sizes of the subtrees (plus one). To do all of this, we of course have to change the definition of a node so that it has an additional `size` field, which is initialized when a new singleton tree is created.

In addition to the `rank` and `select` functions, you can also define the function `splitRank`(T, i), which splits the tree into two by returning the trees T_1 and T_2 such that T_1 contains all keys with rank less than i and T_2 contains all keys with rank is greater or equal to i. Such a function can be used for example to write divide-and-conquer algorithms on imperfectly balanced trees.

Pairing nodes with reduced values. To compute rank-based properties of keys in a BST, we augmented the BST so that each node stores the size of its subtree. More generally, we might want to associate with each node a *reduced value* that is computed by reducing over the subtree rooted at the node by a user specified function. In general, there is no restriction on how the reduced values may be computed, they can be based on keys or additional values that the tree is augmented with. To compute reduced values, we simply store with every node v of a binary search tree, the reduced value of its subtree (i.e. the sum of all the reduced values that are descendants of v, possibly also the value at v itself).

Example 10.40. *The following diagram shows a tree with key-value pairs on the left, and the augmented tree on the right, where each node additionally maintains the sum (the function f is addition) of it subtree.*



The sum at the root (13) is the sum of all values in the tree ($3+1+2+2+5$). It is also the sum of the reduced values of its two children (6 and 5) and its own value 2.

The value of each reduced value in a tree can be calculated as the sum of its two children plus the value stored at the node. This means that we can maintain these reduced values by simply taking this "sum" of three values whenever creating a node. We can thus change a data structure to support reduced values by changing the way we create nodes. In such a data structure, if the function that we use for reduction performs constant work, then the work and the span bound for the data structure remains unaffected.

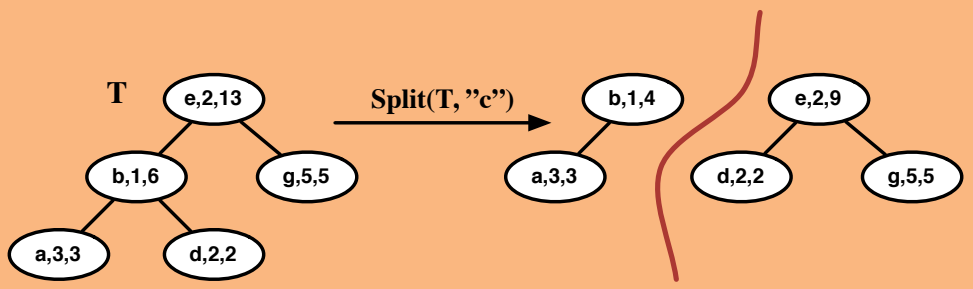
As an example, Figure 10.41 describes an extension of the parametric implementation of treaps to support reduced values. The description is parametric in the values paired with keys and the function `f` used for reduction. The datatype for treaps is extended to store the value paired with the key as well as the reduced value. Specifically, in a `Node`, the first data entry is the value paired by the key and the second is the reduced value.

To compute reduced values as the structure of the tree changes, the implementation relies on an auxilary function `mkNode` (read "make node") that takes the key-value pair as well as the left and right subtrees and computes the reduced value by applying reducer function to the values of the left and right subtrees as well as the value. The only difference in the implementation of `split` and `join` functions from Chapter 10 is the use of `mkNode` instead of `Node`.

Data Structure 10.41 (Treaps with reduced values)**.**

```
1  % type of the reduced value, as specified.
2  type rval = ...
3
4  % associative reducer function, as specified.
5  function f(x: rval, y: rval): rval = ...
6
7  % identity for the reducer function, as specified.
8  I_f : rval = ...
9
10 datatype treap =
11     Leaf
12   | Node of (Treap × key × priority × (rval × rval) × Treap)
13
14 function rvalOf(T) =
15   case T of
16     Leaf ⇒ I_f
17   | Node(_,_,_(_,r),_) ⇒ r
18
19 function mkNode(L,k,p,v,R) =
20   Node(L,k,p,(v,f(rvalOf(L),f(v,rvalOf(R)))),R)
21
22 function split(T,k) =
23   case T of
24     Leaf ⇒ (Leaf, False, Leaf)
25   | Node(L,k',p',(v',s'),R) =
26       case compare(k,k') of
27         LESS ⇒
28           let (L',x,R') = split(L,k)
29           in (L',x,mkNode(R',k',p',v',R)) end
30       | EQUAL ⇒ (L, True, R)
31       | GREATER ⇒
32           let (L',x,R') = split(R,k)
33           in (mkNode(L,k',p',v',L'),x,R') end
34
35 function join(T1,T2) =
36   case (T1,T2) of
37     (Leaf, _) ⇒ T2
38   | (_, Leaf) ⇒ T1
39   | (Node(L1,k1,p1,(v1,s1),R1), Node(L2,k2,p2,(v2,s2),R2)) ⇒
40       if ((p1) > (p2)) then
41         mkNode(L1,k1,p1,v1,join(R1,T2))
42       else
43         mkNode(join(T1,L2),k2,v2,R2)
```

Example 10.42. *The following diagram shows an example of splitting an augmented tree.*



The tree is split by the key c, and the reduced values on the internal nodes need to be updated. This only needs to happen along the path that created the split, which in this case is e, b, and d. The node for d does not have to be updated since it is a leaf. The `makeNode` *for e and b are what will update the reduced values for those nodes.*

We note that this idea can be used with any binary search tree, not just treaps. In all cases one needs only to replace the function that creates a node with a version that also sums the reduced values from the children and the value from the node to create a reduced value for the new node.

Remark 10.43. *We note that in an imperative implementation of binary search trees in which a child node can be side affected, then the reduced values need to be recomputed on all nodes in the path from the modified node to the root.*

10.8 Exercises

Exercise 10.44. *Prove that the minimum possible height of a binary search tree with n keys is $\lceil \log_2(n+1) \rceil$.*

Exercise 10.45 (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 $[k_1, k_2]$.*

Exercise 10.46 (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.*

Exercise 10.47 (Size as reduced value)**.** *Show that size information can be computed as a reduced value. What is the function to reduce over?*

Exercise 10.48. *Implement the* `splitRank` *function.*

Exercise 10.49. *Implement the* `select` *function using* `splitRank`*.*

.