Chapter 12

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 book, we rely on binary search trees to implement set and table abstract data types (Chapter 13), which are then used in the implementation of many algorithms, including for example graph algorithms.

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, we can use a sequence with the array-based cost specification to implement an efficient search function by representing the collection as a sorted sequence and by using binary search. Such an implementation would yield a logarithmic-work search operation. If, however, we want to support dynamic collections, where for example, we insert new elements and delete existing elements, sequences would require linear work. Binary search trees, or ***BSTs*** for short, make it possible to compute with dynamic collections by using insertions, deletions, as well as searches all in logarithmic number of tree operations.

In the traditional treatment of algorithms, which focuses on sequential algorithms, binary search trees revolve around three operations: insertion, deletion, and search. While these operations are important, they are not sufficient for parallelism, since they perform a single update at a time. We therefore 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 12.1) and present an ADT for them (Section 12.2). We then present a parametric implementation of the ADT (Section 12.4) by using only two operations, `split` and `join`, which respectively split a tree at a given key and join two trees. In Section 12.5, we present a cost specification based on the parametric implementation, which achieves 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 functions `split` and `join`. We finish the chapter by presenting a specific instance of the parametric implementation using Treaps (Section 12.6). Other possible implementation techniques are described in Section 12.3

12.1 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 (Definition 2.18). A ***full binary tree*** is a rooted tree, where each node is either a ***leaf***, which has no children, or an ***internal node***, which have a left and a right child (Definition 12.1).

Definition 12.1. [Full Binary Tree] A ***full binary tree*** is an ordered rooted tree in which every internal node has exactly two children: the first or the ***left child*** and the second or 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.

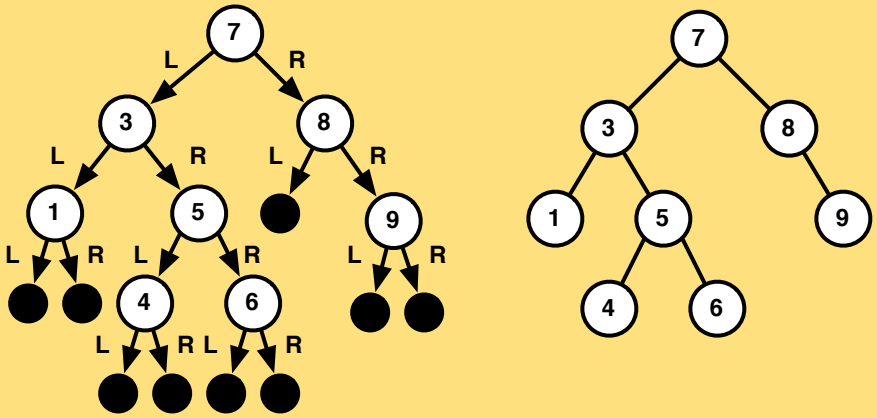
A binary search tree is a full binary tree, where each internal node u has a unique key k such that each node in its left subtree has a key less than k and each node in its right subtree has a key greater that x. Formally, we can define binary search trees as follows.

Definition 12.2. [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 12.3. An example binary search tree over the set of natural numbers $\{1, 3, 4, 5, 6, 7, 8, 9\}$ is shown below.



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.

In the illustration on the left, the edges are oriented away from the root. They could have also been oriented towards the root. When illustrating binary search trees, we usually replace the directed arcs with undirected edges, leaving the orientation to be implicit. We also draw the left and right subtrees of a node on its left and right respectively. Following this convention, we can draw the tree on the left above as shown an the right. We use this convention in future figures.

12.2 The BST Abstract Data Type

ADT 12.4 describes an ADT for BSTs parametrized by a totally ordered key set. We briefly describe this ADT and present some examples. As we shall see, the BST ADT can be implemented in many ways. In order to present concrete examples, we assume an implementation but do not specify it.

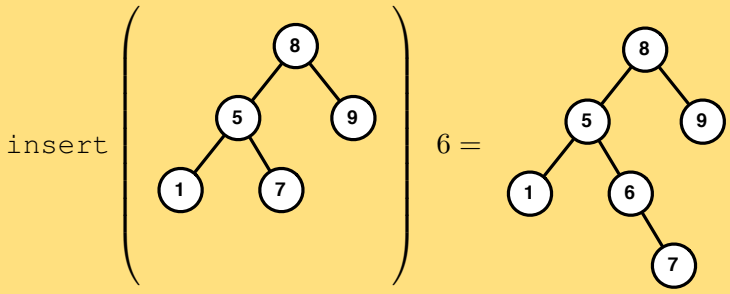
The ADT supports the constructor operations `empty` and `singleton` for creating an empty BST and BST with a single key. The function `find` searches for a given key and returns a boolean indicating success.

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

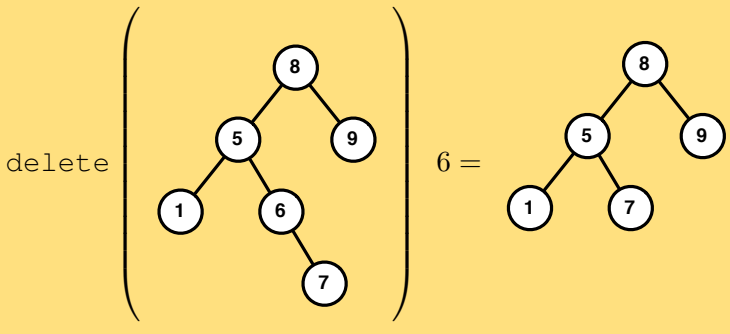
Abstract Data Type 12.4. [BST] For a universe of totally ordered keys $\mathbb{K}$, the BST ADT consists of a type $\mathbb{T}$ representing a power set of keys and the functions specified as follows. In the specification, $[\![T]\!]$ denotes the set of keys in the tree T.

empty	:	$\mathbb{T}$
singleton	:	$\mathbb{K} \to \mathbb{T}$
find	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{B}$
delete	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{T}$
insert	:	$\mathbb{T} \to \mathbb{K} \to \mathbb{T}$
intersection	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
difference	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
union	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
split	:	$\mathbb{T} \to \mathbb{K} \to (\mathbb{T} \times \mathbb{B} \times \mathbb{T})$
join	:	$\mathbb{T} \to \mathbb{T} \to \mathbb{T}$

Example 12.5. Inserting the key 6 into the input tree returns a new tree including 6.



Example 12.6. Deleting the key 6 from the input tree returns a tree without it.



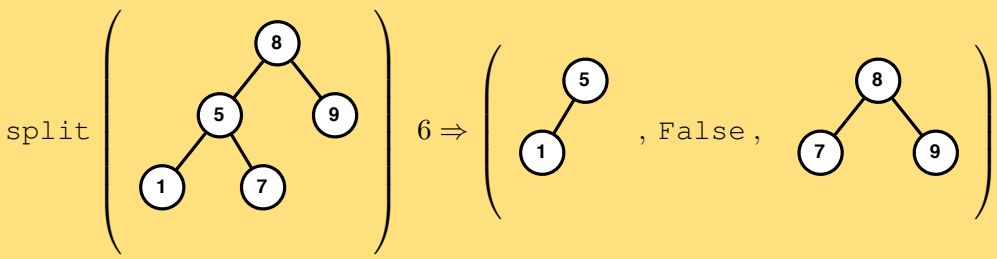
The function `union` takes two BSTs and returns a BST that contains all the keys in them; `union` is an aggregate insert operation. The function `intersection` takes two BSTs and returns a BST that contains the keys common in both. The function `difference` takes two

BSTs t_1 and t_2 and returns a BST that contains the keys in t_1 that are not in t_2; `difference` is an aggregate delete operation.

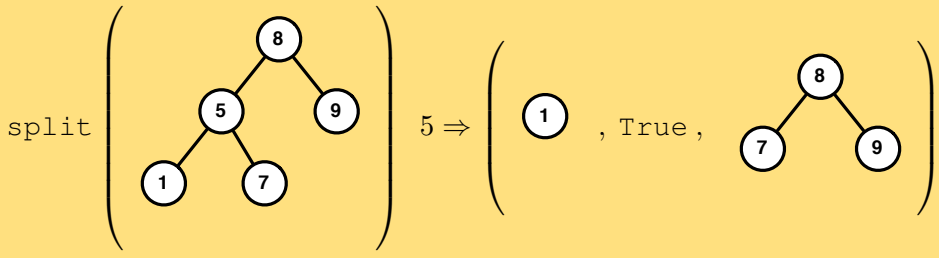
The function `split` takes a tree t and a key k and splits t into two trees: one consisting of all the keys of t less than k, and another consisting of all the keys of t greater than k. It also returns a Boolean value indicating whether k appears in t. The exact structure of the trees returned by `split` can differ from one implementation to another: the specification only requires that the resulting trees to be valid BSTs and that they contain the keys less than k and greater than k, leaving their structure otherwise unspecified.

Example 12.7. 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.

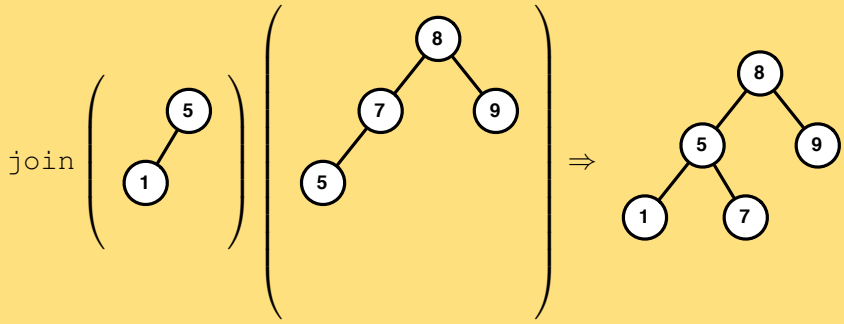


- Splitting the input tree at 5 yields two trees, consisting of the keys less than 5 and those greater than 5, indicating also that 5 is found in the input tree.



The function `join` 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. The exact structure of the tree returned by `join` can differ from one implementation to another: the specification only requires that the resulting tree is a valid BST and that it contains all the keys in the trees joined.

Example 12.8. The function `join` illustrated.



12.3 Implementation via Balancing

The main idea behind the implementation of BSTs is to organize the keys such that

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

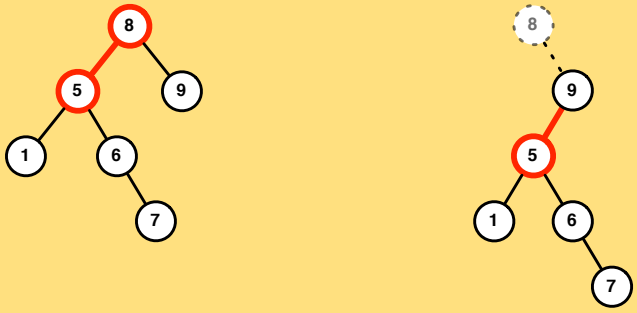
2. a set of keys that constitute a contiguous range in a sorted order of keys in the tree can be moved as a chunk by performing constant work.

To see how we can search in a tree, consider searching for a key k in a tree t whose root is r. 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 conclude that the key is not in the tree. In both cases we have followed a single path through the BST starting at the root.

Example 12.9. 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 all the keys that are 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 their root. For example, we can move the whole subtree to another location by linking the root to another parent.

Example 12.10. Consider the tree shows below on the left, we can handle all the keys that are less than 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.



By finding a range of keys by traversing a path in the BST, and by moving ranges with constant work, it turns out to be possible to implement all the operations in the BST ADT efficiently as long as the paths traversed 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 with n keys, a perfectly balanced tree has height exactly $\lceil \lg(n+1) \rceil$.

Ideally we would like to use only perfectly balanced trees. If we never make changes to the tree, we could balance it once and for all. If, however, we want to update the tree by, for example, inserting new keys, then maintaining such perfect balance is costly. In fact, it turns out to be impossible to maintain a perfectly balanced tree while allowing insertions in $O(\lg n)$ work. BST data structures therefore aim to keep approximate balance instead of a perfect one. We refer to a BST data structure as *nearly balanced* or simply as *balanced* if all trees with n elements have height $O(\lg n)$, 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 nearly balanced 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 approximate 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 $0 < \alpha \leq 1 - \frac{1}{\sqrt{2}}$. 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 (the term "Treap" is short for "tree heap"). Treaps guarantee approximate balance with high-probability.

5. ***Splay trees*** are an amortized data structure that does not guarantee approximate balance, but instead guarantees that for any sequence of m insert, find and delete operations each does $O(\lg n)$ 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.

12.4 A Parametric Implementation

We describe a minimalist implementation of the BST ADT based on two functions, `split` and `join`. Since the implementation depends on just these two functions, we refer to it as a parametric implementation. Data Structure 12.11 illustrates the parametric implementation, assuming that the implementation of `split` and `join` are supplied. The implementation defines the tree type as consisting of leaves or internal nodes with left and right subtrees and a key. The auxiliary function, `joinM` 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 contains all the keys in t_1 and t_2 as well as k.

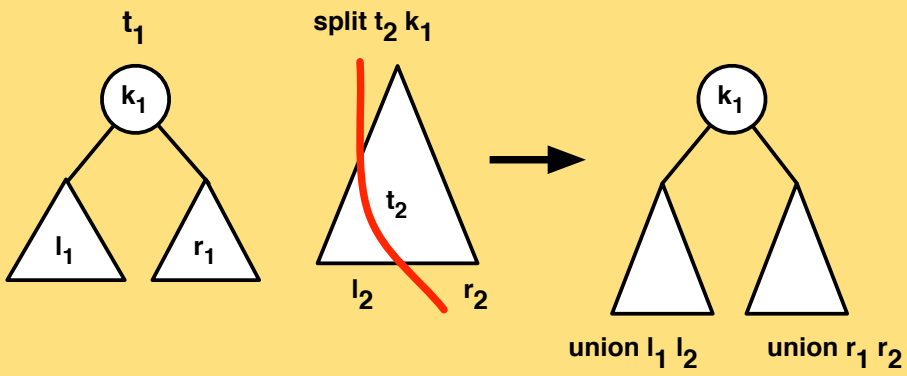
The function `find` is easily implementable with a `split`, which indicates whether the key used for splitting is found in the tree or not. To implement `insert` of a key k into a tree, we first `split` the tree at k and then join the two returned trees along with key k using `joinM`. To implement `delete` of a key k from a tree, we first `split` the tree at k and then join the two returned trees with `join`. If the key k was found, this gives us 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`).

The implementation of `union` uses divide and conquer. 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. There are different ways to select the key k used to split the tree. One way is 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 is the approach take in the parametric implementation.

Data Structure 12.11. [Implementing the BST ADT with `split` and `join`]

```
type 𝕋 = Leaf | Node of (𝕋 × 𝕂 × 𝕋)
split t k = ... (* as given *)
join t1 t2 = ... (* as given *)

joinM t1 k t2 = join t1 (join (singleton k) t2)
empty = Leaf
singleton (k) = Node(Leaf,k,Leaf)
find t k = let (_,v,_) = split t k  in v end
delete t k = let (l,_,r) = split t k in join l r end
insert t k = let (l,_,r) = split t k in joinM l k r end

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

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

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

Example 12.12. The union of tree t_1 and t_2 illustrated.



The implementation of `intersection` uses a divide-and-conquer approach similar to that of `union`. As in `union`, we split both trees by using the key k_1 at the root of the first tree, and compute intersections recursively. We then compute the result by joining the results from the recursive calls and including the key k_1 if it is found in both trees. Note that since the trees are BSTs, checking for the intersections of left and right subtrees recursively and is guaranteed to find all shared keys because the `split` operation places all keys less than and greater than the given key to two separate trees.

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

12.5 Cost Specification

There are many ways to implement an efficient data structure that matches our BST 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 $O(\lg n)$, 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. 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 12.14 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.

Cost Specification 12.14. [BSTs] The ***BST*** cost specification is defined 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.

	Work	*Span*
`empty`	$O(1)$	$O(1)$
`singleton` k	$O(1)$	$O(1)$
`split` $t\ k$	$O(\lg \|t\|)$	$O(\lg \|t\|)$
`join` $t_1\ t_2$	$O(\lg(\|t_1\| + \|t_2\|))$	$O(\lg(\|t_1\| + \|t_2\|))$
`find` $t\ k$	$O(\lg \|t\|)$	$O(\lg \|t\|)$
`insert` $t\ k$	$O(\lg \|t\|)$	$O(\lg \|t\|)$
`delete` $t\ k$	$O(\lg \|t\|)$	$O(\lg \|t\|)$
`intersect` $t_1\ t_2$	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$
`difference` $t_1\ t_2$	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$
`union` $t_1\ t_2$	$O\left(m \cdot \lg \frac{n+m}{m}\right)$	$O(\lg n)$

For the rest of the operations, we justify the cost bounds by assuming the existence 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 `difference`, which are similar are more difficult to see. Let's analyze the cost for `union` as implemented by the parametric implementation. It is easy to apply a similar analysis to `intersection` and `difference`.

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 is perfectly balanced (i.e., the left and right subtrees of the root have 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. $|t_1| < |t_2|$.

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`:

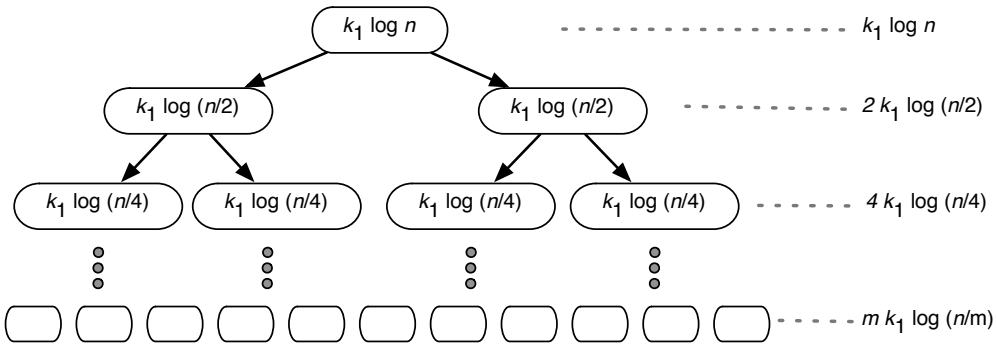
$$\begin{aligned} W_{\text{union}}(m,n) &\leq 2W_{\text{union}}(m/2,n/2) + W_{\text{split}}(n) + W_{\text{join}}(n+m) + O(1) \\ &\leq 2W_{\text{union}}(m/2,n/2) + O(\lg n)\ . \end{aligned}$$

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

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

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

We can draw the recursion tree showing the work performed by the splitting of t_2 and by the joining of the results as follows. For simplicity of the argument, let's assume that the leaves of the tree correspond to the case for $m = 1$.



Brick method. Let's analyze the structure of the recursion tree shown above. We can find the number of leaves in the tree by examining 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). The tree t_2 does not affects the shape of the recursion tree or the stopping condition. Thus, there are exactly m leaves in the tree. In fact, the recursion can be rewritten as a recursion of the form $W(m) = 2W(m/2) + \ldots$, which means that there are m leaves. By the same reasoning, we can see that the leaves are $(1 + \lg m)$ deep.

Let's now determine the size of t_2 at the leaves. We have m keys in t_1 to start with, and they split t_2 evenly all the way down to the level of the leaves (by assumption). Thus, the leaves have all the same size of $\frac{n}{m}$. Therefore, each leaf adds a $O(\lg(1 + \frac{n}{m}))$ term to the work (the $1+$ is needed to deal with the case that $n = m$). Since there are m leaves, the whole bottom level costs $O(m \lg(1 + \frac{n}{m}))$.

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

$\frac{2^{i-1}\lg n/2^{i-1}}{2^i \lg n/2^i} = \frac{1}{2}\frac{\lg n - i + 1}{\lg n - i}$, where $i \leq \lg m < \lg n$. This ratio is less than 1 for all levels except for the last level, where by taking $i = \lg n - 1$ we have

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

Thus the total work is asymptotically dominated by the total work of the leaves, which is $O(m \lg n/m)$.

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

$$W(n,m) = \sum_{i=0}^{b} 2^i \lg \frac{n}{2^i}.$$

We can rewrite this sum as

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

Let's now focus on the second term. Note that

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

Substituting the closed form for each inner summation and simplifying leads to

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

Let's now go back and plug this into the original work bound and simplify

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

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

Removing the Assumptions. 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

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

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 $O(m \lg(1 + \frac{n}{m}))$.

Still, in reality, t_1 doesn't have to be perfectly balanced as we assumed. To generalize the analysis, we just need a tree with $O(\lg m)$ height. Thus, we only need t_1 to be approximately balanced.

Finally, we assumed that t_1 is larger that t_2. If it is smaller, then we can reverse the order of arguments, so in this case, there is no loss of generality. If they are the same size, we need to be a bit more precise in our handling of the base case in our summation but this is all.

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

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

Essentially the same analysis applies to the functions `intersection` and `difference`, whose structures are the same as `union`, except for an additional constant work and span for the conditional (**if**) expression.

12.6 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 BSTs that are probabilistically balanced. Of the many balanced BST data structures, Treaps are likely the simplest, but, since they are randomized, they only guarantee approximate balance with high probability.

The idea behind Treaps is to associate a uniformly randomly selected priority to each key and maintain a priority order between keys in addition to the binary-search-tree order. The priority order between keys resemble the order used in binary heaps, leading to the name "Tree Heap" or "Treap." We define Treaps as follows.

[1] This is also known as Jensen's inequality.

Definition 12.15. [Treap] A Treap is a binary search tree over a set K 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 K, satisfies the heap property on the priorities $p(k), k \in K$, i.e., for every internal node v with left and right children u and w:

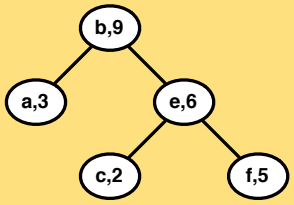
$$p(k(v)) \geq p(k(u)) \text{ and } p(k(v)) \geq p(k(w)),$$

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

Example 12.16. 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 12.17. 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. $O(\lg |S|)$ height, with high probability. We will show this shortly.

The second idea behind Treaps is to update the tree structure according to new priorities efficiently by performing local reorganizations. Based on our parametrized implementation, we can give an implementation for the BST ADT with Treaps simply by implementing the `split` and `join` functions. Data Structure 12.18 shows such an implementation. For the implementation we assume, without loss of generality, that the priorities are integers. We present only the code for `split` and `join`; the rest of the implementation is essentially the same as in Data Structure 12.11 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 that they belong to.

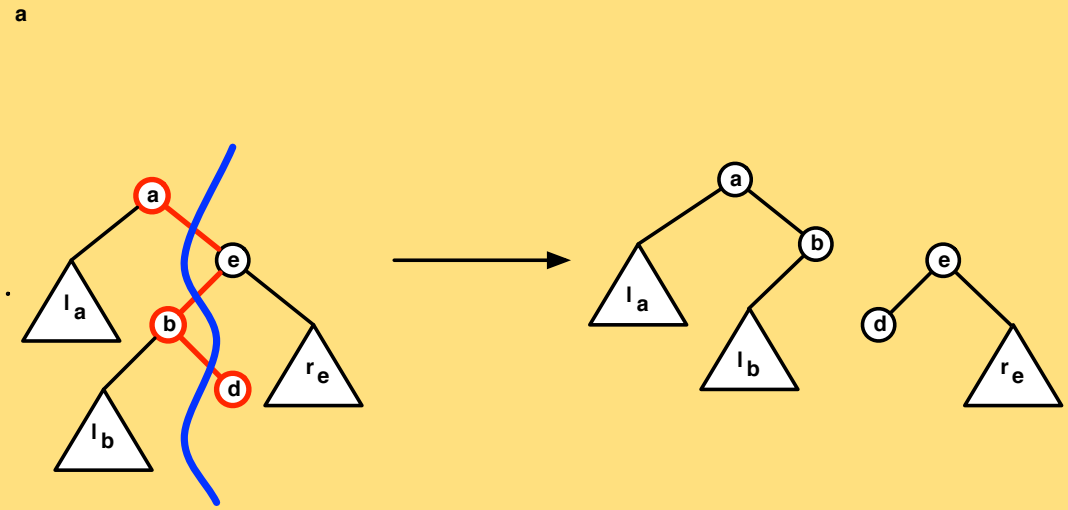
Data Structure 12.18. [Implementing BST with Treaps]

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

let empty = Leaf

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

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

join t1 t2 =
  case (t1,t2)
  | (Leaf, _) => t2
  | (_ , Leaf) => t1
  | (Node (l1,k1,p1,r1), Node (l2,k2,p2,r2)) =>
      if (p1 > p2) then
        Node (l1,k1,p1, join r1 t2)
      else
        Node (join t1 l2, k2,p2,r2)
end
```

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 programming languages.

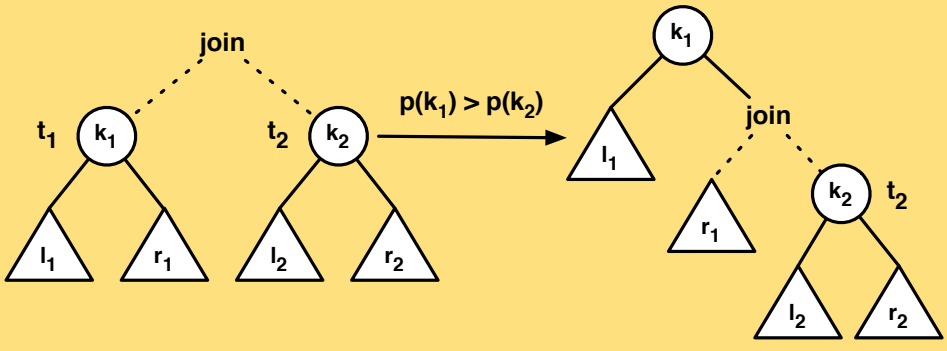
The `split` algorithm recursively traverses the tree from the root to the key k splitting along the path, and then when returning from the recursive calls, it puts the subtrees back together. When putting back the trees along the path being split through, the function does not have to compare priorities 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 12.19. A `split` operation on a Treap and key c, which is not in 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 12.18) 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.



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:

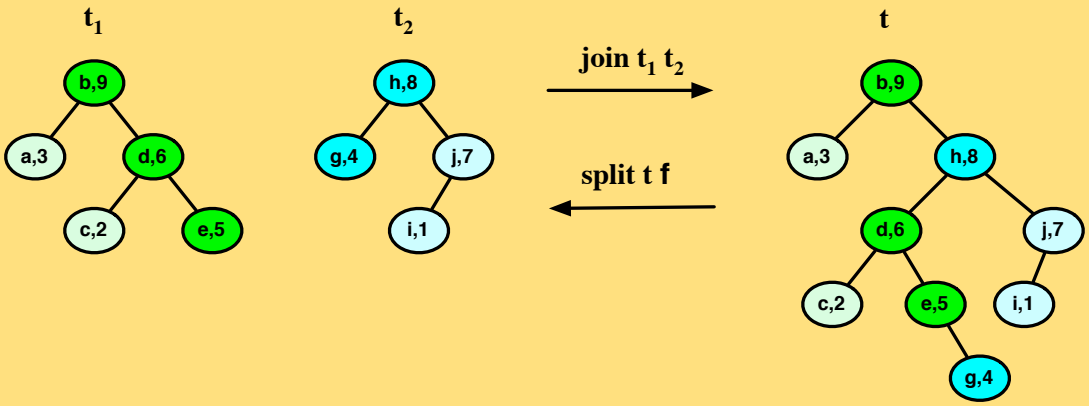
Example 12.20. An illustration of `join`t_1 t_2 on Treaps. If $p(k_1) > p(k_2)$, then the function recurs with `join`(R_1, t_2) and the result becomes the right child of k_1.



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 `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 12.21. An illustration of `join` for Treaps applied to t_1 and t_2 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.



Let's bound now the work for `split` and `join`. Each one does constant work on each recursive call. For `split` each recursive call goes to one of the children, so the number of recursive calls is at most the height of t. For `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 `split` $t\ k$ is $O(h(t))$ and the work of `join`(t_1, m, t_2) is $O(h(t_1) + h(t_2))$. Thus all that is left to do is to bound the height of a Treap.

Analysis of randomized Treaps. We can analyze the height of a Treap by relating them to quicksort, which we analyzed in Chapter 11. In particular consider the following variant of quicksort.

Algorithm 12.22. Treap Generating Quicksort

```
qsTree a =
  if |a| = 0 then Leaf
  else let
    x = the key k ∈ a for which p(k) is the largest
    a1 = ⟨ y ∈ a | y < x ⟩
    a2 = ⟨ y ∈ a | y > x ⟩
    (l, r) = (qsTree a1 ) || (qsTree a2)
  in
    Node (l, x, r)
  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`(a) is the Treap for the sequence a. 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 x 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 l are less than the pivot, and the keys in r are greater than the pivot, the tree has the BST property.

Based on this isomorphism, we can bound the height of a Treap by the recursion depth of quicksort. In Chapter 11, we proved that if we pick the priorities at random, the recursion depth is $O(\lg n)$ with high probability. Therefore we know that the height of a Treap is $O(\lg n)$ with high probability.

12.7 Augmenting Trees

Thus far in this chapter, the only interesting information that we stored in BSTs were keys. While such trees can be useful, we sometimes wish to augment trees with more information. In this section, we describe how we might augment BSTs with additional information such as key-value pairs, subtree sizes, and reduced values in general.

12.7.1 Augmenting with Key-Value Pairs

Perhaps the simplest form of augmentation involves storing in the BST a key-value pair instead of just a key. Implementing BSTs augmented with key-value pairs is relatively straightforward by updating the relevant parts of the ADT. For example, to accommodate the key, we can change the BST data type to a key-value pair, and update the implementation of the functions to pass the value around with the key as needed, making sure that a key-value pair is never

Algorithm 12.23. [Rank]

```
rank t k =
  case t
  | Leaf => 0
  | Node (l, k', r) =>
      case compare (k, k')
      | LESS => rank l k
      | EQUAL => |l|
      | GREATER => |l| + 1 + rank r k

select t i =
  case t
  | Leaf => raise exception OutOfRange
  | Node (l, k, r) =>
      case compare (i, |l|) of
        LESS => select l i
        EQUAL => k
        GREATER => select r i - |l| - 1)
```

separated. For functions such as `find` and `split` that may return the value, we make sure to do so.

12.7.2 Augmenting with Size

As a more complex augmentation, we might want to associate with each node in the tree a size field that tells us how large the subtree rooted at that node is. As a motivating example for this form of augmentation, suppose that we wish to extend the BST ADT (ADT 12.4) with the following additional functions.

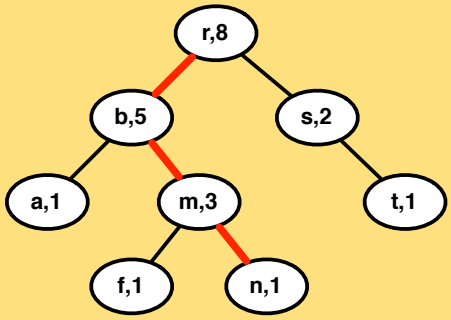
- Function `rank` t k returns the rank of the key k in the tree, i.e., the number of keys in t that are less than or equal to k.
- Function `select` T i returns the key with the rank i in t.

Such functions arise in many applications. For example, we can use them to implement the sequence interface discussed in Chapter 6.

If we have a way to count the number of nodes in a subtree, then we can easily implement these functions. Algorithm 12.23 shows such an implementation by using a size operation for computing the size of a tree, written $|t|$ for tree t. With balanced trees such as Treaps, the `rank` and `select` functions require logarithmic span but linear work, because computing the size of

a subtree takes linear time in the size of the subtree. If, however, we augment the tree so that at each node, we store the size of the subtree rooted at that node, then work becomes logarithmic, because we can find the size of a subtree in constant work.

Example 12.24. An example BST, where keys are ordered lexicographically and the nodes are augmented with the sizes of subtrees. The path explored by `rank (T,n)` and `select (T,4)` is highlighted.



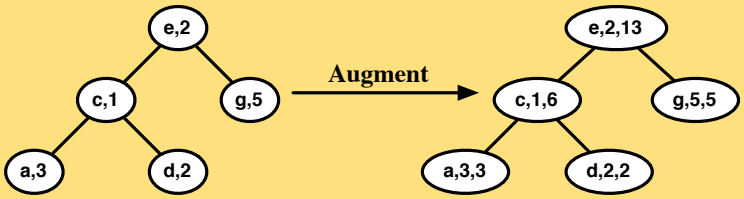
To implement a size-augmented tree, we need to keep `size` field at each node and compute the size of the nodes as they are created. In our parametric implementation, we can incorporate the `size` field by changing the definition of a node and initializing it to 1, when a singleton tree is created. When `split` and `join` functions create a new node, they can compute its size by summing the sizes of its children.

In addition to the `rank` and `select` functions, we 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.

12.7.3 Augmenting 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 u of a binary search tree, the reduced value of its subtree (i.e. the sum of all the reduced values that are descendants of u, possibly also the value at u itself).

Example 12.25. The following drawing shows a tree with key-value pairs on the left, and the augmented tree on the right, where each node additionally maintains the sum 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 the "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.

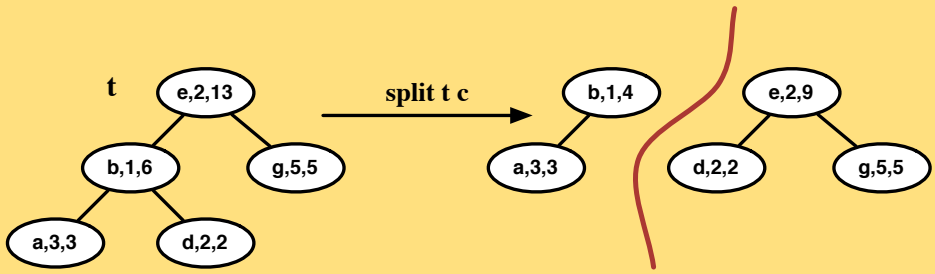
As an example, Figure 12.26 describes an extension of the parametric implementation of Treaps to support reduced values. The description is parametric in the values paired with keys and the function `f` used for reduction. The type for Treaps is extended to store the value paired with the key as well 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 auxiliary 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 12 is the use of `mkNode` instead of `Node`.

Data Structure 12.26. [Treaps with reduced values]

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

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

rvOf t =
  case t
  | Leaf => id_f
  | Node (_,_,_(_,w),_) => w

mkNode (l,k,p,v,r) = Node (l,k,p,(v,f (rvOf l,v,rvOf r)),r)

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

join t1 t2 =
  case (t1,t2) of
    (Leaf, _) => t2
  | (_, Leaf) => t1
  | (Node (l1,k1,p1,(v1,w1),r1), Node (l2,k2,p2,(v2,w2),r2)) =>
      if p1 > p2) then mkNode (l1,k1,p1,v1,join r1 t2)
      else mkNode (join t1 l2,k2,v2,r2)
```

Example 12.27. 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. We only need to replace the function for creating a node so that as it creates the node, it also computes a reduced value for the node by summing the reduced values of the children and the value of the node itself.

Remark 12.28. In an imperative implementation of binary search trees, when a child node is side affected, the reduced values for the nodes on the path from the modified node to the root must be recomputed.

12.8 Problems

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

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

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

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

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

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

12-6 Size as reduced value
Show that size information can be computed as a reduced value. What is the function to reduce over?

12-7 Implementing `splitRank`
Implement the `splitRank` function.

12-8 Implementing `select`
Implement the `select` function using `splitRank`.

.