Chapter 37

Introduction

This chapter presents the motivation behind Binary Search Trees (***BSTs***) and an Abstract Data Type (ADT) for them. It also briefly discusses the well-known implementation techniques for BSTs.

1 Motivation

Searching Dynamic Collections. Searching efficiently is one of the most important goals in the design of data structures. Of the many search data structures that have been designed and are used in practice, search trees, more specifically balanced binary search trees (BSTs), occupy a coveted place because of their broad applicability to many different problems. For example, in this book, we rely on binary search trees to implement set and table (dictionary) abstract data types. Sets and tables are instrumental in many algorithms, including for example graph algorithms, many of which we cover later in the book.

A Total Order. BSTs require that the keys being stored come from a total order so we can store the keys in "sorted order". This makes BSTs particularly useful when our searches are based on the order, such as finding all keys between two values (a range search), or given a key, finding the next larger key. They are also useful if the only access we have to keys is the ability to compare them. In these cases other search structures such as hash tables (discussed in a later chapter) are of little utility. Even when we do not need ordering, BSTs have some other benefits over hash tables. An important one is that it is easy to make them "persistent" so that an update leaves the old tree unmodified while creating a new tree.

Dynamic versus Static. If we are interested in searching a static or unchanging collection of elements, then BSTs are not necessary and we can use sequences instead. More specifically, we can represent a collection as a sorted sequence and use binary search to implement searches. Using array sequences such a binary search requires logarithmic work.

If we wish to support dynamic collections, which allow, for example, inserting and deleting keys, then a sequence based implementation would require linear work for these updates since all, or many, keys might need to be moved. BSTs support various updates, including insertions and deletions, in logarithmic work.

Sequential versus Parallel Aggregate 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 operations, such as union, intersection, set-difference, filter, map, and reduce. All of these can be implemented in parallel. Then instead of inserting one element at a time, for example, we can use a union to insert a whole set of values.

Roadmap. We first define binary search trees. We then present an ADT for binary search trees , and describe a parametric implementation of the ADT. The parametric implementation uses only one non-trivial operation, $joinMid$, which joins together two trees with a key in the middle. As a result, we are able to reduce the problem of implementing the BST ADT to the problem of implementing just the function $joinMid$. We then present a specific instance of the parametric implementation using Treaps .

2 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 37.1 (Full Binary Tree)**.** A ***full binary tree*** is an ordered rooted tree where each node is either a ***leaf***, which has no children, or an ***internal node***, which has a ***left child***, a value, and a ***right child***. This can be defined as the recursive type:

$$\begin{array}{lcl} \texttt{type}\ \alpha\ \mathit{tree} & = & \mathit{Leaf} \\ & | & \mathit{Node}\ \texttt{of}\ (\mathit{tree} \times \alpha \times \mathit{tree}) \end{array}$$

where α is the type of the value stored at each internal node, and for an internal node $Node(L, v, R)$, L is the left child, v is the value, and R is the right child.

For a given node in a binary tree, we define the ***left subtree*** of the node as the subtree rooted at the left child, and the ***right subtree*** of the node as the subtree rooted at the right child.

Here we only consider storing values on the internal nodes and assume leaves have no values associated with them.

It is useful to define different traversal orders on binary trees. A traversal starts at the root and inductively (recursively) traverses each subtree. The traversal order that is most important to us is the in-order traversal. It can be defined as follows:

Definition 37.2. The ***in-order*** traversal of a full binary tree is given by the order of elements in the sequence returned by:

```
$inOrder\ T =$
  case $T$ of
    $Leaf \Rightarrow \langle\,\rangle$
  | $Node(L, k, R) \Rightarrow inOrder(L) +\!\!+ \langle\, k \,\rangle +\!\!+ inOrder(R)$
```

i.e., it recursively puts the values from the left subtree first, then the key k at the node, and then the values from the right subtree. We assume leaves have no values.

Another common order is the pre-order, which visits keys in the order given by:

```
$preOrder\ T =$
  case $T$ of
    $Leaf \Rightarrow \langle\,\rangle$
  | $Node(L, k, R) \Rightarrow \langle\, k \,\rangle +\!\!+ preOrder(L) +\!\!+ preOrder(R)$
```

i.e., first the key, then the left subtree, and finally the right subtree.

When the values (keys) we store at each node have a total ordering defined by a comparison $<$, we will use the following notation: For complete binary trees T, T_1 and T_2, and a value k, we use:

$$\begin{array}{lcl} T < k & \equiv & \text{for all } k' \in T, k' < k \\ k < T & \equiv & \text{for all } k' \in T, k < k' \\ T_1 < T_2 & \equiv & \text{for all } k_1 \in T_1, k_2 \in T_2, k_1 < k_2 \end{array}$$

We are now ready to define binary search trees.

Definition 37.3 (Binary Search Tree (BST))**.** Consider a set S taken from a total order defined by the comparison $<$. A ***binary search tree*** (BST) over S is a full binary tree T (with no leaf values) that satisfies the following conditions.

1. There is a one-to-one mapping $k(v)$ from internal tree nodes of T to elements in S, and

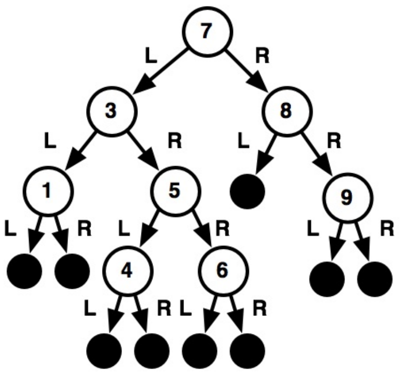
2. $inOrder(T)$ is sorted by $<$.

In the definition, the second condition is referred to as the ***BST property***. It can be equivalently stated as: for all internal nodes $(L, k, R) \in T, L < k < R$ (using our notation above).

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.

We define the ***size*** of a BST S as the number of keys in the tree, and also write it as $|S|$. We define the ***depth*** of either a leaf or node in a BST, as the length of the path from the root to that leaf or node. The root always has depth 0. We define the ***height*** of a BST, denoted as $h(T)$, as the maximum depth of any leaf. An empty tree has height 0, and a tree with a single node has height 1.

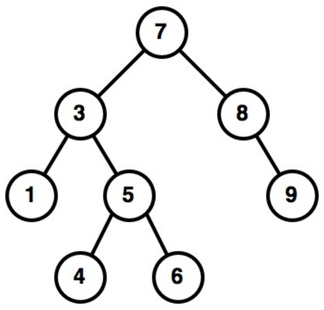
Example 37.1. An example binary search tree over the set of natural numbers $\{4, 1, 7, 9, 83, 6, 5\}$, and defined by the standard total ordering over the natural numbers, is given by the following tree T:



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— the in-order ordering, given by $inOrder(T)$ is $\langle\, 1, 3, 4, 5, 6, 7, 8, 9 \,\rangle$, is sorted. Or, equivalently, for every node, the keys in the left subtree are less, than the key at the root, and the ones in the right subtree are greater.

The size of the tree is 8, because there are 8 keys in the tree. The height of the tree is 4, e.g., the path from root 7 to a child of node 4.

In the illustration of the tree, the edges are oriented away from the root, to indicate the direction we can search from starting at the root. When illustrating binary search trees, we usually replace the directed arcs with undirected edges, leaving the orientation to be implicit. We also drop the leaves since they contain no information, and implicitly assume the left branch (L) is on the left and the right branch (R) is on the right. Given these conventions, we get the following simpler illustration of a BST, which we will be using in the rest of the book.



Here we are assuming that we just store keys at the nodes of the trees, and that the keys represent an ordered set. In practice we often store a value associated with each key at each node, and perhaps other information. These do not change the fundamental ideas we discuss here, so we will ignore them at first and then come back to them.

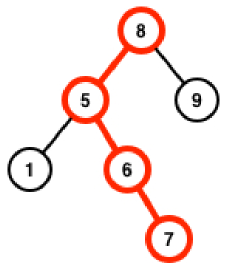
3 Searching a BST

The primary goal of a BST is to do fast searches for a particular key. Fortunately, it is relatively easy to search for a key in a BST, and even to find the next larger or smaller key in the tree. To find a particular key we can start at the root r and if k equals the key at the root, call it k', then we have found our key, otherwise if $k < k'$, then we know that k cannot appear in the right subtree, so we only need to search the left subtree, and if $k > k'$, 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. Based on this idea the following algorithm will return whether a key k is in a BST T.

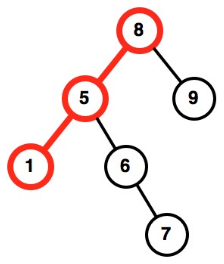
Algorithm 37.4 (Searching a BST)**.**

```
$\mathit{find}\ T\ k =$
  case $T$ of
    $\mathit{Leaf} \Rightarrow \mathit{false}$
  $| \mathit{Node}(L, k', R) \Rightarrow$
      if $(k = k')$ then $\mathit{true}$
      else if $(k < k')$ then $\mathit{find}\ L\ k$
      else $\mathit{find}\ R\ k$
```

Example 37.2. A successful search (find) for 7. The search path is highlighted.



An unsuccessful search (find) for 4.



It is important to note that the search only visits a path from the root to some node (either a leaf or internal node). Therefore the search will visit at most as many nodes as the height of the tree, $h(T)$. Assuming the comparison $<$ takes constant work, this means the work for *find* is $O(h(T))$.

4 Balancing BSTs

Given that the work to find a key (and also for other operations) is proportional to the height of a BST, our goal should therefore be to keep the height low. In the worst case the height could be equal to the size. This would be true if every node had one child that is a leaf. Such a tree is clearly very unbalanced. Our goal therefore implies we should keep the tree "balanced" such that all leaves of the tree are at approximately at the same depth. We formalize this notion as follows.

Definition 37.5 (Perfectly Balanced BSTs)**.** A binary tree is ***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. There are various schemes that keep trees nearly balanced for any sized tree. These schemes maintain invariants at each node that ensure this near balance.

Definition 37.6 (Nearly Balanced BSTs)**.** We refer to a balancing scheme as maintaining ***near balance***, or simply ***balance***, if all trees with n elements that satisfy the scheme's invariants have height $O(\lg n)$. In some cases this is satisfied in expectation or with high probability.

Balanced BST Data Structures. There are many balancing schemes for BSTs. 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 balancing schemes:

1. ***AVL trees*** are the earliest nearly balanced BST data structure (1962). AVL trees maintain the invariant that the two children of each node differ in height by at most one, which 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 $0 < \alpha \leq 1 - \frac{1}{\sqrt{2}}$. 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 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 $O(\lg n)$ amortized work.

There are several other balancing schemes for BST data structures (e.g. scapegoat trees and AA trees), as well as many that allow larger degrees, including 2–3 trees, brother trees, and B trees.

Remark. Many of the existing BST data structures were developed for sequential computing. Some of these data structures such as Treaps, which we describe here, generalize naturally to parallel computing. But some others, such as data structures that rely on amortization techniques can be challenging to support in the parallel setting.

5 An Interface for Sets

As discussed, BSTs are useful for representing sets of keys that have a total ordering, and require dynamic changes. Let us consider what functions could be useful for such "dynamic" sets. Certainly we will require some basic operations such as creating an empty set, creating a singleton set, or returning the size of the set. Also, we would like to find an element in a set, insert an element into a set, and delete an element from a set. You might have studied how to do this with BSTs with particular balancing schemes (e.g. AVL trees, or red-black trees) in previous courses.

However, at a higher level we would like to supply bulk operations on sets, such as taking the union of two sets, filtering a set so that only elements that satisfy a predicate remain, or summing the elements of a set with respect to some associative operation. For tables (i.e., when we associate a value with each key), we might also want to map some function over the values to generate a new table, or filter based on the values. Such bulk operations are very useful in programming with sets and tables. They are also important for taking advantage of parallelism since individual inserts and deletes are inherently sequential, but bulk operations can often be parallelized.

To implement these bulk operations it is useful to build them on top of some primitives. Building them on top of insertion, deletion will not be effective since they are inherently sequential. Instead, as we will see, it is useful to build them on top of three other operations `split`, `joinM`, and `joinPair`. These are described below. As described in the next chapter, `split` and `joinPair` can be implemented in terms of `joinM` so all we will really need is `joinM` (actually a slight variant, called `joinMid`).

Here we present an abstract data type that supplies a collection of useful functions over ordered sets. We will extend this list with further functions in the following chapters. An important aspect of this interface is that it is designed to support parallelism.

Data Type 37.7 (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 whose types are specified as:

$$\begin{array}{lcl}
empty & : & \mathbb{T} \\
singleton & : & \mathbb{K} \to \mathbb{T} \\
size & : & \mathbb{T} \to \mathbb{N} \\
\mathit{find} & : & \mathbb{T} \to \mathbb{K} \to \mathbb{B} \\
delete & : & (\mathbb{T} \times \mathbb{K}) \to \mathbb{T} \\
insert & : & (\mathbb{T} \times \mathbb{K}) \to \mathbb{T} \\
union & : & (\mathbb{T} \times \mathbb{T}) \to \mathbb{T} \\
intersection & : & (\mathbb{T} \times \mathbb{T}) \to \mathbb{T} \\
\mathit{difference} & : & (\mathbb{T} \times \mathbb{T}) \to \mathbb{T} \\
split & : & (\mathbb{T} \times \mathbb{K}) \to (\mathbb{T} \times \mathbb{B} \times \mathbb{T}) \\
joinPair & : & (\mathbb{T} \times \mathbb{T}) \to \mathbb{T} \\
joinM & : & (\mathbb{T} \times \mathbb{K} \times \mathbb{T}) \to \mathbb{T} \\
\mathit{filter} & : & (\mathbb{K} \to \mathit{bool}) \to \mathbb{T} \to \mathbb{T} \\
reduce & : & (\mathbb{K} \times \mathbb{K} \to \mathbb{K}) \to \mathbb{K} \to \mathbb{T} \to \mathbb{K}
\end{array}$$

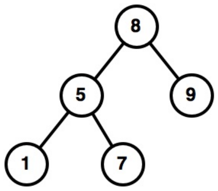
and functionality is defined below.

The ADT supports two constructors: $empty$ and $singleton$. As their names imply, the function $empty$ creates an empty BST and the function $singleton$ creates a BST with a single key.

The function *find* searches for a given key and returns a boolean indicating success.

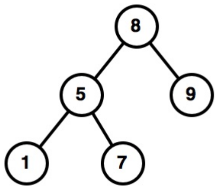
The functions $insert$ and $delete$ insert and delete a given key into or from the BST.

Example 37.3. Consider the following tree.

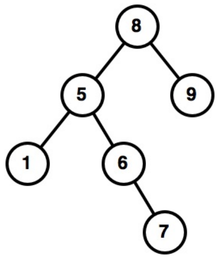


- Searching for 5 in the tree above returns `true`.
- Searching for 6 in the tree above returns `false`.

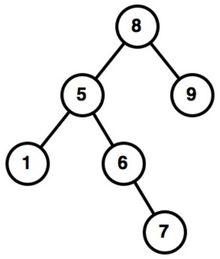
Example 37.4 (Insertion)**.** Consider the following tree.



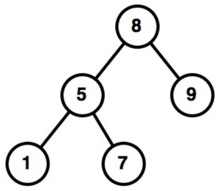
Inserting the key 6 into this tree returns the following tree.



Example 37.5 (Deletion)**.** Consider the following tree.



Deleting the key 6 from this tree returns the following tree.

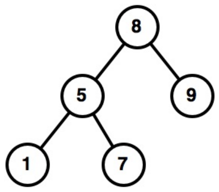


Union, Intersection, and Difference. The function *union* takes two BSTs and returns a BST that contains the union of the keys in the two BSTs. The function *intersection* takes two BSTs and returns a BST that contains the keys that appear in both (i.e., the intersection of the sets). The function *difference* 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.

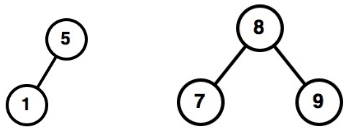
Note that *union* can be thought of as a "parallel" insert since it can add multiple keys at once. Similarly *difference* can be thought of as a "parallel" delete, since it will remove multiple keys at once.

Split. 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.

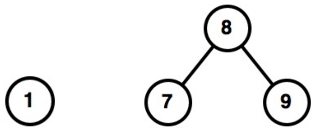
Example 37.6. Consider the following input tree.



- Splitting the input tree at 6 yields two following trees, consisting of the keys less that 6 and those greater that 6, and returns `false` to indicate that 6 is not in the input tree.

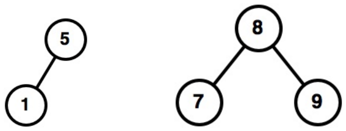


- Splitting the input tree at 5 yields the following two trees, consisting of the keys less than 5 and those greater than 5, and also returns `true` to indicate that 5 is found in the input tree.

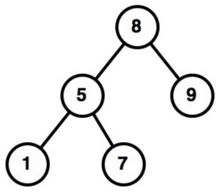


JoinPair. The function *joinPair* takes two trees T_1 and T_2 such that all the keys in T_1 are less than the keys in T_2. The function returns a tree that contains all the keys in T_1 and T_2. The exact structure of the tree returned by *joinPair* can differ from one implementation to another: the specification only requires that the resulting tree is a valid BST and that it contains all the keys in the trees being joined.

Example 37.7. Joining the two trees below using the function *joinPair*



yields the following three.



JoinM. The function $joinM$ takes a tree T_1, a key k, and another tree T_2 such that $T_1 < k < T_2$. A returns a tree containing all of the three. It is similar to $joinPair$ except it includes the middle value (hence the M instead of $Pair$). As with $joinPair$ the exact structure of the tree returned can differ from one implementation to another.