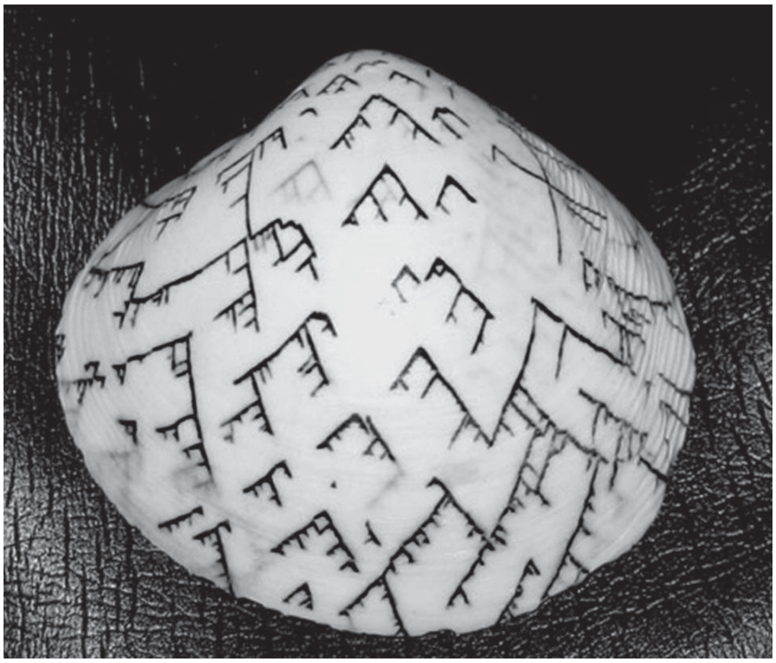
Chapter

3 Binary Search Trees


Lioconcha castrensis shell, 2005. Michael T. Goodrich. Used with permission.

Contents

3.1 Searches and Updates 91
3.2 Range Queries 101
3.3 Index-Based Searching 104
3.4 Randomly Constructed Search Trees 107
3.5 Exercises 110

Figure 3.1: A three-dimensional virtual environment. U.S. Army photo.

In three-dimensional video games and virtual environments, it is important to be able to locate objects relative to the other objects in their environment. For example, if the system is modeling a player driving a simulated vehicle through a virtual war zone, it is useful to know where that vehicle is relative to various obstacles and other players. Such nearby objects are useful for the sake of rendering a scene or identifying targets, for example. (See Figure 3.1.)

Of course, one way to do such a reference check for some object, x, relative to a virtual environment is to compare x to every other object in the environment. For an environment made up of n objects, such a search would require $O(n)$ object-object comparisons for x; hence, doing such a search for every object, x, would take $O(n^2)$ time, which is expensive. Such a computation for a given object x is not taking advantage of the fact that it is likely that there are potentially large groups of objects far from x. It would be nice to quickly dismiss such groups as being of low interest, rather than comparing each one to x individually.

For such reasons, many three-dimensional video games and virtual environments, including the earliest versions of the game ***Doom***, create a partitioning of space using a binary tree, by applying a technique known as ***binary space partitioning***. To create such a partitioning, we identify a plane, P, that divides the set of objects into two groups of roughly equal size—those objects to the left of P and those objects to the right of P. Then, for each group, we recursively subdivide them with other separating planes until the number of objects in each subgroup is small enough to handle as individuals. Given such a binary space partition (or BSP) tree, we can then locate any object in the environment simply by locating it relative to P, and then recursively locating it with respect to the objects that fall on the same side of P as it does. Such BSP tree partitions represent a three-dimensional environment using the data structure we discuss in this chapter, the ***binary search tree***, in that a BSP tree is a binary tree that stores objects at its nodes in a way that allows us to perform searches by making left-or-right decisions at each of its nodes.

3.1 Searches and Updates

Suppose we are given an ordered set, S, of objects, represented as key-value pairs, for which we would like to locate query objects, x, relative to the objects in this set. That is, we would like to identify the nearest neighbors of x in S, namely, the smallest object greater than x in S and the largest object smaller than x in S, if these objects exist. One of the simplest ways of storing S in this way is to store the elements of S in order in an array A. Such a representation would allow us, for example, to identify the ith smallest key, simply by looking up the key of the item stored in cell $A[i]$. In other words, if we needed a method, $\mathsf{key}(i)$, for accessing the key of the ith smallest key-value pair, or $\mathsf{elem}(i)$, the element that is associated with this key, then we could implement these methods in constant time given the representation of S in sorted order in the array A. In addition, if we store the elements of S in such a sorted array, A, then we know that the item at index i has a key no smaller than keys of the items at indices less than i and no larger than keys of the items at indices larger than i.

The Binary Search Algorithm

This observation allows us to quickly "home in" on a search key k using a variant of the children's game "high-low." We call an item I of S a ***candidate*** if, at the current stage of the search, we cannot rule out that I has key equal to k. The algorithm that results from this strategy is known as ***binary search***.

There are several ways to implement this strategy. The method we describe here maintains two parameters, low and high, such that all the candidate items have index at least low and at most high in S. Initially, $\mathsf{low} = 1$ and $\mathsf{high} = n$, and we let $\mathsf{key}(i)$ denote the key at index i, which has $\mathsf{elem}(i)$ as its element. We then compare k to the key of the median candidate, that is, the item with index

$$\mathsf{mid} = \lfloor(\mathsf{low} + \mathsf{high})/2\rfloor.$$

We consider three cases:

- If $k = \mathsf{key}(\mathsf{mid})$, then we have found the item we were looking for, and the search terminates successfully returning $\mathsf{elem}(\mathsf{mid})$.
- If $k < \mathsf{key}(\mathsf{mid})$, then we recur on the first half of the vector, that is, on the range of indices from low to $\mathsf{mid} - 1$.
- If $k > \mathsf{key}(\mathsf{mid})$, we recursively search the range of indices from $\mathsf{mid} + 1$ to high.

This ***binary search*** method is given in detail in Algorithm 3.2. To initiate a search for key k on an n-item sorted array, A, indexed from 1 to n, we call $\mathsf{BinarySearch}(A, k, 1, n)$.

```
Algorithm BinarySearch($A, k$, low, high):
  Input: An ordered array, $A$, storing $n$ items, whose keys are accessed with
    method key($i$) and whose elements are accessed with method elem($i$); a
    search key $k$; and integers low and high
  Output: An element of $A$ with key $k$ and index between low and high, if such
    an element exists, and otherwise the special element null
  if low $>$ high then
      return null
  else
      mid $\leftarrow \lfloor$(low $+$ high)$/2\rfloor$
      if $k =$ key(mid) then
          return elem(mid)
      else if $k <$ key(mid) then
          return BinarySearch($A, k$, low, mid $- 1$)
      else
          return BinarySearch($A, k$, mid $+ 1$, high)
```

Algorithm 3.2: Binary search in an ordered array.

We illustrate the binary search algorithm in Figure 3.3.

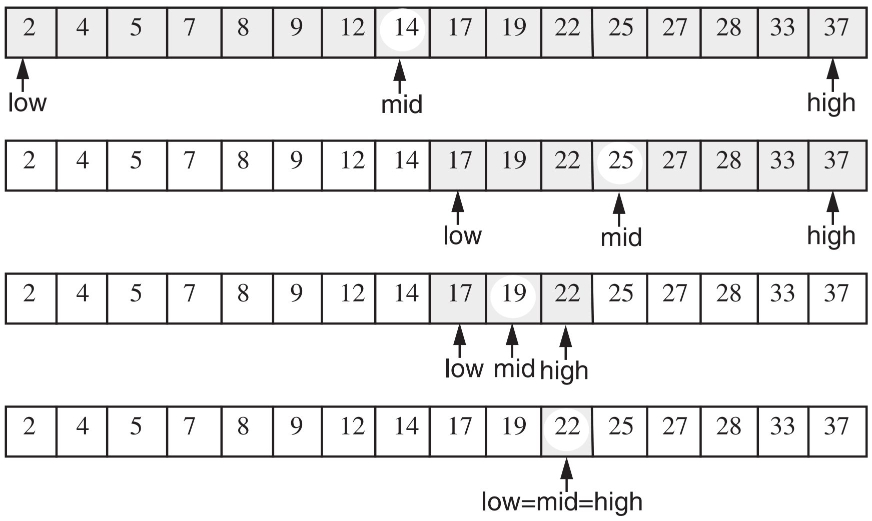


Figure 3.3: Example of a binary search to search for an element with key 22 in a sorted array. For simplicity, we show the keys but not the elements.

Analyzing the Binary Search Algorithm

Considering the running time of binary search, we observe that a constant number of operations are executed at each recursive call. Hence, the running time is proportional to the number of recursive calls performed. A crucial fact is that, with each recursive call, the number of candidate items still to be searched in the array A is given by the value $\mathsf{high} - \mathsf{low} + 1$. Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(\mathsf{mid} - 1) - \mathsf{low} + 1 = \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor - \mathsf{low} \le \frac{\mathsf{high} - \mathsf{low} + 1}{2}$$

or

$$\mathsf{high} - (\mathsf{mid} + 1) + 1 = \mathsf{high} - \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor \le \frac{\mathsf{high} - \mathsf{low} + 1}{2}.$$

Initially, the number of candidate is n; after the first call to $\mathsf{BinarySearch}$, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. That is, if we let a function, $T(n)$, represent the running time of this method, then we can characterize the running time of the recursive binary search algorithm as follows:

$$T(n) \le \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{else,} \end{cases}$$

where b is a constant. In general, this recurrence equation shows that the number of candidate items remaining after each recursive call is at most $n/2^i$. (We discuss recurrence equations like this one in more detail in Section 11.1.) In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate items. Hence, the maximum number of recursive calls performed is the smallest integer m such that $n/2^m < 1$. In other words (recalling that we omit a logarithm's base when it is 2), $m > \log n$. Thus, we have $m = \lfloor \log n \rfloor + 1$, which implies that $\mathsf{BinarySearch}(A, k, 1, n)$ runs in $O(\log n)$ time.

The space requirement of this solution is $\Theta(n)$, which is optimal, since we have to store the n objects somewhere. This solution is only efficient if the set S is static, however, that is, we don't want to insert or delete any key-value pairs in S. In the dynamic case, where we want to perform insertions and deletions, then such updates take $O(n)$ time. The reason for this poor performance in an insertion, for instance, is due to our need to move elements in A greater than the insertion key in order to keep all the elements in A in sorted order, similar to the methods described in Section 2.2.1. Thus, using an ordered array to store elements to support fast searching only makes sense for the sake of efficiency if we don't need to perform insertions or deletions.

3.1.1 Binary Search Tree Definition

The data structure we discuss in this section, the binary search tree, applies the motivation of the binary search procedure to a tree-based data structure, to support update operations more efficiently. We define a binary search tree to be a binary tree in which each internal node v stores an element e such that the elements stored in the left subtree of v are less than or equal to e, and the elements stored in the right subtree of v are greater than or equal to e. Furthermore, let us assume that external nodes store no elements; hence, they could in fact be ***null*** or references to a special NULL_NODE object.

An inorder traversal of a binary search tree visits the elements stored in such a tree in nondecreasing order. A binary search tree supports searching, where the question asked at each internal node is whether the element at that node is less than, equal to, or larger than the element being searched for.

We can use a binary search tree T to locate an element with a certain value x by traversing down the tree T. At each internal node we compare the value of the current node to our search element x. If the answer to the question is "smaller," then the search continues in the left subtree. If the answer is "equal," then the search terminates successfully. If the answer is "greater," then the search continues in the right subtree. Finally, if we reach an external node (which is empty), then the search terminates unsuccessfully. (See Figure 3.4.)

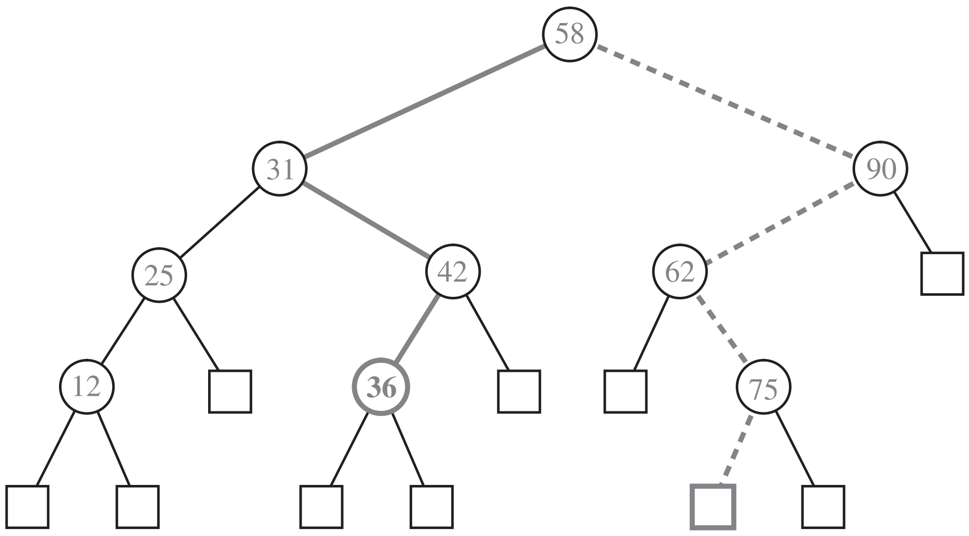


Figure 3.4: A binary search tree storing integers. The thick solid path drawn with thick lines is traversed when searching (successfully) for 36. The thick dashed path is traversed when searching (unsuccessfully) for 70.

3.1.2 Searching in a Binary Search Tree

Formally, a ***binary search tree*** is a binary tree, T, in which each internal node v of T stores a key-value pair, (k, e), such that keys stored at nodes in the left subtree of v are less than or equal to k, while keys stored at nodes in the right subtree of v are greater than or equal to k.

In Algorithm 3.5, we give a recursive method TreeSearch, based on the above strategy for searching in a binary search tree T. Given a search key k and a node v of T, method TreeSearch returns a node (position) w of the subtree $T(v)$ of T rooted at v, such that one of the following two cases occurs:

- w is an internal node of $T(v)$ that stores key k.
- w is an external node of $T(v)$. All the internal nodes of $T(v)$ that precede w in the inorder traversal have keys smaller than k, and all the internal nodes of $T(v)$ that follow w in the inorder traversal have keys greater than k.

Thus, a method find(k), which returns the element associated with the key k, can be performed on a set of key-value pairs stored in a binary search tree, T, by calling the method TreeSearch$(k, T.\text{root}())$ on T. Let w be the node of T returned by this call of the TreeSearch method. If node w is internal, we return the element stored at w; otherwise, if w is external, then we return ***null***.

```
Algorithm TreeSearch($k, v$):
  Input: A search key $k$, and a node $v$ of a binary search tree $T$
  Output: A node $w$ of the subtree $T(v)$ of $T$ rooted at $v$, such that either $w$ is an
     internal node storing key $k$ or $w$ is the external node where an item with key
     $k$ would belong if it existed

   if $v$ is an external node then
       return $v$
   if $k = \text{key}(v)$ then
       return $v$
   else if $k < \text{key}(v)$ then
       return TreeSearch($k, T.\text{leftChild}(v)$)
   else
       return TreeSearch($k, T.\text{rightChild}(v)$)
```

Algorithm 3.5: Recursive search in a binary search tree.

The analysis of the running time of this algorithm is simple. The binary tree search algorithm executes a constant number of primitive operations for each node it traverses in the tree. Each new step in the traversal is made on a child of the previous node. That is, the binary tree search algorithm is performed on the nodes of a path of T that starts from the root and goes down one level at a time. Thus, the number of such nodes is bounded by $h + 1$, where h is the height of T. In other words, since we spend $O(1)$ time per node encountered, the search method runs in $O(h)$ time, where h is the height of the binary search tree T. (See Figure 3.6.)

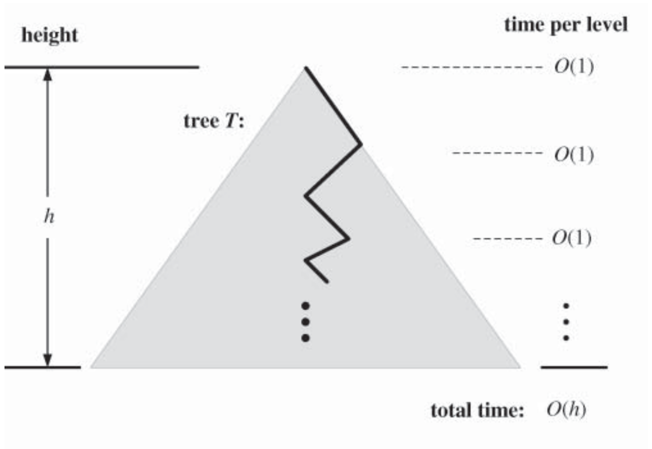


Figure 3.6: Illustrating the running time of searching in a binary search tree. The figure uses standard visualization shortcuts of viewing a binary search tree as a big triangle and a path from the root as a zig-zag line.

Admittedly, the height h of T can be as large as n, but we expect that it is usually much smaller. The best we can do for the height, h, is $\lceil \log(n+1) \rceil$ (see Exercise C-3.5), and we would hope that in most cases h would in fact be $O(\log n)$. For instance, we show below, in Section 3.4, that a randomly constructed binary search tree will have height $O(\log n)$ with high probability. For now, though, consider a binary search tree, T, storing n items, such that, for each node v in T, each of v's children store at most three-quarters as many items in their subtrees as v does. Lots of binary search trees could have this property, and, for any such tree, its height, $H(n)$, would satisfy the following recurrence equation:

$$H(n) \leq \begin{cases} 1 & \text{if } n < 2 \\ H(3n/4) + 1 & \text{else.} \end{cases}$$

In other words, a child of the root stores at most $(3/4)n$ items in its subtree, any of its children store at most $(3/4)^2 n$ items in their subtrees, any of their children store at most $(3/4)^3 n$ items in their subtrees, and so on. Thus, since multiplying by $3/4$ is the same as dividing by $4/3$, this implies that $H(n)$ is at most $\lceil \log_{4/3} n \rceil$, which is $O(\log n)$. Intuitively, the reason we achieve a logarithmic height for T in this case is that the subtrees rooted at each child of a node in T have roughly "balanced" sizes.

We show in the next chapter how to maintain an upper bound of $O(\log n)$ on the height of a search tree T, by maintaining similar notions of balance, even while performing insertions and deletions. Before we describe such schemes, however, let us describe how to do insertions and deletions in a standard binary search tree.

3.1.3 Insertion in a Binary Search Tree

Binary search trees allow implementations of the insert and remove operations using algorithms that are fairly straightforward, but not trivial. To perform the operation insert(k, e) in a binary search tree T, to insert a key-value pair, (k, e), we perform the following algorithm (which works even if the tree holds multiple key-value pairs with the same key):

```
Let $w \leftarrow$ TreeSearch$(k, T.$root$())$
while $w$ is an internal node do
    // There is item with key equal to $k$ in $T$ in this case
    Let $w \leftarrow$ TreeSearch$(k, T.$leftChild$(w))$
Expand $w$ into an internal node with two external-node children
Store $(k, e)$ at $w$
```

The above insertion algorithm eventually traces a path from the root of T down to an external node, w, which is the appropriate place to insert an item with key k based on the ordering of the items stored in T. This node then gets replaced with a new internal node accommodating the new item. Hence, an insertion adds the new item at the "bottom" of the search tree T. An example of insertion into a binary search tree is shown in Figure 3.7.

The analysis of the insertion algorithm is analogous to that for searching. The number of nodes visited is proportional to the height h of T in the worst case, since we spend $O(1)$ time at each node visited. Thus, the above implementation of the method insert runs in $O(h)$ time.

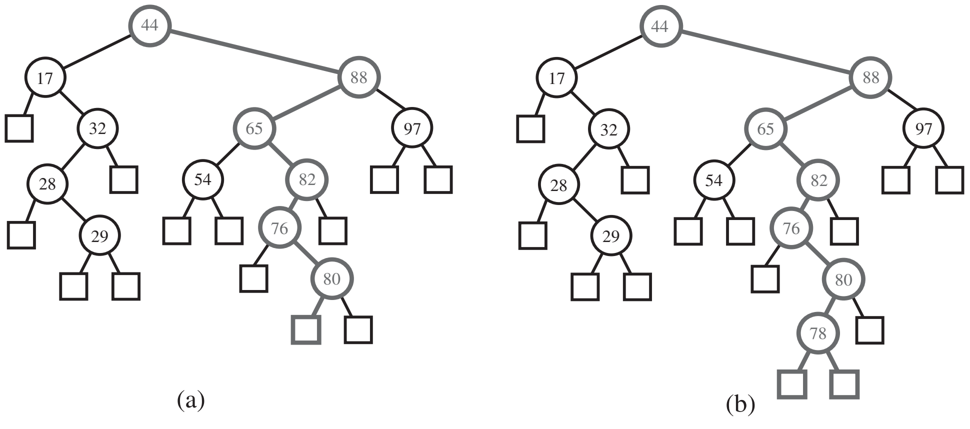


Figure 3.7: Insertion of an item with key 78 into a binary search tree. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

3.1.4 Deletion in a Binary Search Tree

Performing a remove(k) operation, to remove an item with key k from a binary search tree T is a bit more complex than the insertion algorithm, since we do not wish to create any "holes" in the tree T. Such a hole, where an internal node would not store an element, would make it difficult if not impossible for us to correctly perform searches in the binary search tree. Indeed, if we have many removals that do not restructure the tree T, then there could be a large section of internal nodes that store no elements, which would confuse any future searches. Thus, we must implement item deletion to avoid this situation.

The removal operation starts out simple enough, since we begin by executing algorithm TreeSearch$(k, T.\text{root}())$ on T to find a node storing key k. If TreeSearch returns an external node, then there is no element with key k in T, and we return the special element ***null*** and are done. If TreeSearch returns an internal node w instead, then w stores an item we wish to remove.

We distinguish two cases (of increasing difficulty) of how to proceed based on whether w is a node that is easily removed:

- If one of the children of node w is an external node, say node z, we simply remove w and z from T, and replace w with the sibling of z (which is an operation called removeAboveExternal(z) in Section 2.3.4).

This case is illustrated in Figure 3.8.

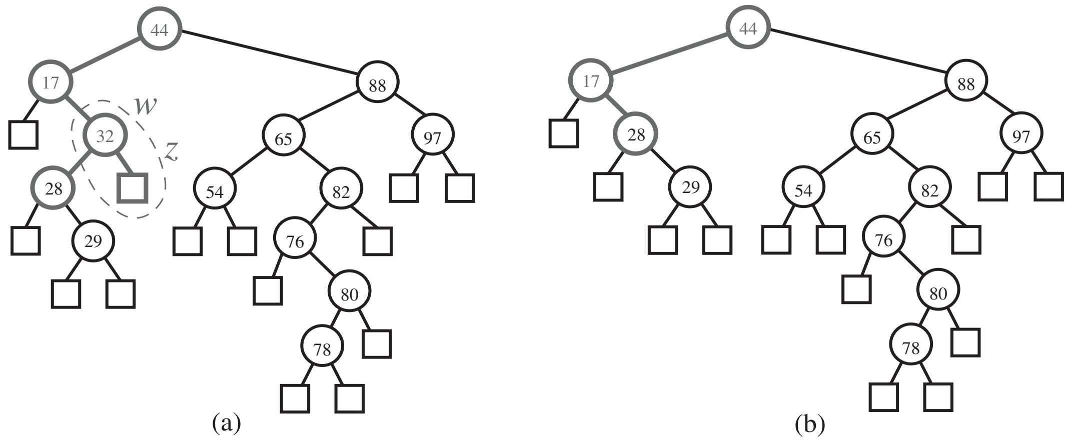


Figure 3.8: Deletion from the binary search tree of Figure 3.7b, where the key to remove (32) is stored at a node (w) with an external child: (a) shows the tree before the removal, together with the nodes affected by the operation that removes the external node, z, and its parent, w, replacing w with the sibling of z; (b) shows the tree T after the removal.

- If both children of node w are internal nodes, we cannot simply remove the node w from T, since this would create a "hole" in T. Instead, we proceed as follows (see Figure 3.9):

    1. We find the first internal node y that follows w in an inorder traversal of T. Node y is the left-most internal node in the right subtree of w, and is found by going first to the right child of w and then down T from there, following left children. Also, the left child x of y is the external node that immediately follows node w in the inorder traversal of T.
    2. We save the element stored at w in a temporary variable t, and move the item of y into w. This action has the effect of removing the former item stored at w.
    3. We remove x and y from T by replacing y with x's sibling, and removing both x and y from T (which is equivalent to operation removeAboveExternal(x) on T, using the terminology of Section 2.3.4).
    4. We return the element previously stored at w, which we had saved in the temporary variable t.

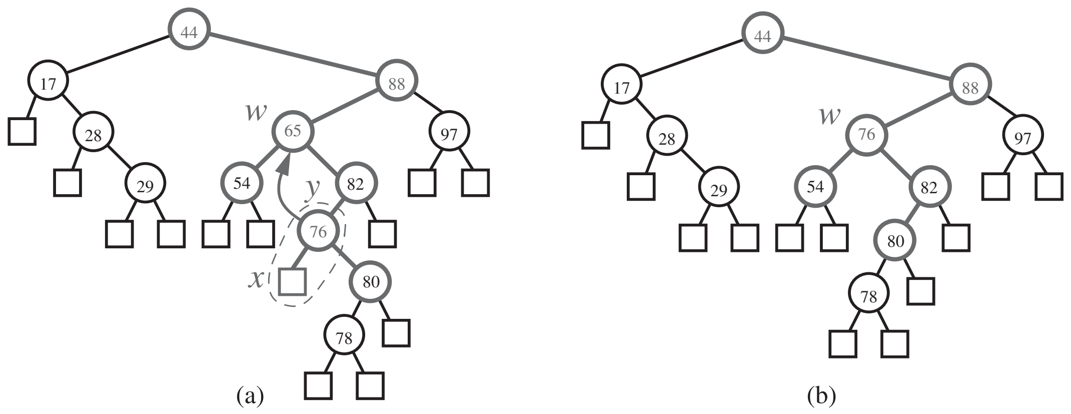


Figure 3.9: Deletion from the binary search tree of Figure 3.7b, where the key to remove (65) is stored at a node whose children are both internal: (a) before the removal; (b) after the removal.

Note that in Step 1 above, we could have selected y as the right-most internal node in the left subtree of w.

The analysis of the removal algorithm is analogous to that of the insertion and search algorithms. We spend $O(1)$ time at each node visited, and, in the worst case, the number of nodes visited is proportional to the height h of T. Thus, in a binary search tree, T, the remove method runs in $O(h)$ time, where h is the height of T.

3.1.5 The Performance of Binary Search Trees

A binary search tree T is an efficient implementation of an ordered set of n key-value pairs but only if the height of T is small. For instance, if T is balanced so that it has height $O(\log n)$, then we get logarithmic-time performance for the search and update operations described above. In the worst case, however, T could have height as large as n; hence, it would perform like an ordered linked list in this case. Such a worst-case configuration arises, for example, if we insert a set of keys in increasing order. (See Figure 3.10.)

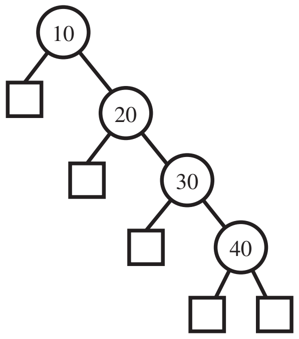


Figure 3.10: Example of a binary search tree with linear height, obtained by inserting keys in increasing order.

To sum up, we characterize the performance of the binary search tree data structure in the following theorem.

Theorem 3.1: *A binary search tree T with height h for n key-element items uses $O(n)$ space and executes the operations* find, insert, *and* remove *each in $O(h)$ time.*

Ideally, of course, we would like our binary search tree to have height $O(\log n)$, and there are several ways to achieve this goal, including several that we explore in the next chapter. For instance, as we explore in this chapter (in Section 3.4), if we construct a binary search tree, T, by inserting a set of n items in random order, then the height of T will be $O(\log n)$ with high probability. Alternatively, if we have our entire set, S, of n items available, then we can sort S and build a binary search tree, T, with height $O(\log n)$ from the sorted listing of S (see Exercise A-3.2). In addition, if we already have a binary search tree, T, of $O(\log n)$ height, and thereafter only perform deletions from T, then each of those deletion operations will run in $O(\log n)$ time (although interspersing insertions and deletions can lead to poor performance if we don't have some way to maintain balance). There are a number of other interesting operations that can be done using binary search trees, however, besides simple searches and insertions and deletions.

3.2 Range Queries

Besides the operations mentioned above, there are other interesting operations that can be performed using a binary search tree. One such operation is the ***range query*** operation, where, we are given an ordered set, S, of key-value pairs, stored in a binary search tree, T, and are asked to perform the following query:

findAllInRange(k_1, k_2): Return all the elements stored in T with key k such that $k_1 \leq k \leq k_2$.

Such an operation would be useful, for example, to find all cars within a given price range in a set of cars for sale. Suppose, then, that we have a binary search tree T representing S. To perform the findAllInRange(k_1, k_2) operation, we use a recursive method, RangeQuery, that takes as arguments, k_1 and k_2, and a node v in T. If node v is external, we are done. If node v is internal, we have three cases, depending on the value of key(v), the key of the item stored at node v:

- key$(v) < k_1$: We recursively search the right child of v.
- $k_1 \leq$ key$(v) \leq k_2$: We report element(v) and recursively search both children of v.
- key$(v) > k_2$: We recursively search the left child of v.

We describe the details of this search procedure in Algorithm 3.11 and we illustrate it in Figure 3.12. We perform operation findAllInRange(k_1, k_2) by calling RangeQuery$(k_1, k_2, T.\text{root}())$.

```
Algorithm RangeQuery($k_1, k_2, v$):
   Input: Search keys $k_1$ and $k_2$, and a node $v$ of a binary search tree $T$
   Output: The elements stored in the subtree of $T$ rooted at $v$ whose keys are in
      the range $[k_1, k_2]$
    if $T$.isExternal($v$) then
        return $\emptyset$
    if $k_1 \leq$ key($v$) $\leq k_2$ then
        $L \leftarrow$ RangeQuery($k_1, k_2, T$.leftChild($v$))
        $R \leftarrow$ RangeQuery($k_1, k_2, T$.rightChild($v$))
        return $L \cup \{$element($v$)$\} \cup R$
    else if key($v$) $< k_1$ then
        return RangeQuery($k_1, k_2, T$.rightChild($v$))
    else if $k_2 <$ key($v$) then
        return RangeQuery($k_1, k_2, T$.leftChild($v$))
```

Algorithm 3.11: The method for performing a range query in a binary search tree.

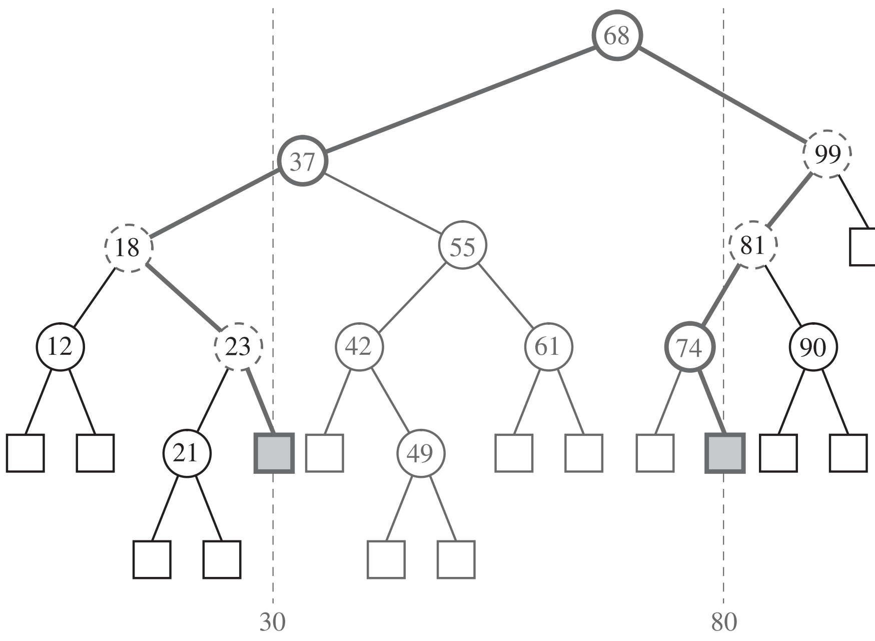


Figure 3.12: A range query using a binary search tree for the keys $k_1 = 30$ and $k_2 = 80$. Paths P_1 and P_2 of boundary nodes are drawn with thick lines. The boundary nodes storing items with key outside the interval $[k_1, k_2]$ are drawn with dashed lines. There are four internal inside nodes.

Intuitively, the method RangeQuery is a modification of the standard binary-tree search method (Algorithm 3.5) to search for the keys between k_1 and k_2, inclusive. For the sake of simplifying our analysis, however, let us assume that T does not contain items with key k_1 or k_2.

Let P_1 be the search path traversed when performing a search in tree T for key k_1. Path P_1 starts at the root of T and ends at an external node of T. Define a path P_2 similarly with respect to k_2. We identify each node v of T as belonging to one of following three groups (see Figure 3.12):

- Node v is a ***boundary node*** if v belongs to P_1 or P_2; a boundary node stores an item whose key may be inside or outside the interval $[k_1, k_2]$.
- Node v is an ***inside node*** if v is not a boundary node and v belongs to a subtree rooted at a right child of a node of P_1 or at a left child of a node of P_2; an internal inside node stores an item whose key is inside the interval $[k_1, k_2]$.
- Node v is an ***outside node*** if v is not a boundary node and v belongs to a subtree rooted at a left child of a node of P_1 or at a right child of a node of P_2; an internal outside node stores an item whose key is outside the interval $[k_1, k_2]$.

Analysis of the Range Query Operation

Consider an execution of the algorithm RangeQuery(k_1, k_2, r), where r is the root of T. We traverse a path of boundary nodes, calling the algorithm recursively either on the left or on the right child, until we reach either an external node or an internal node w (which may be the root) with key in the range $[k_1, k_2]$. In the first case (we reach an external node), the algorithm terminates returning the empty set. In the second case, the execution continues by calling the algorithm recursively at both of w's children. We know that node w is the bottommost node common to paths P_1 and P_2. For each boundary node v visited from this point on, we either make a single call at a child of v, which is also a boundary node, or we make a call at one child of v that is a boundary node and the other child that is an inside node. Once we visit an inside node, we will visit all of its (inside node) descendants.

Since we spend a constant amount of work per node visited by the algorithm, the running time of the algorithm is proportional to the number of nodes visited. We count the nodes visited as follows:

- We visit no outside nodes.
- We visit at most $2h + 1$ boundary nodes, where h is the height of T, since boundary nodes are on the search paths P_1 and P_2 and they share at least one node (the root of T).
- Each time we visit an inside node v, we also visit the entire subtree T_v of T rooted at v and we add all the elements stored at internal nodes of T_v to the reported set. If T_v holds s_v items, then it has $2s_v + 1$ nodes. The inside nodes can be partitioned into j disjoint subtrees $T_1, \ldots, T_j$ rooted at children of boundary nodes, where $j \leq 2h$. Denoting with s_i the number of items stored in tree T_i , we have that the total number of inside nodes visited is equal to

$$\sum_{i=1}^{j}(2s_i + 1) = 2s + j \leq 2s + 2h.$$

Therefore, at most $2s + 4h + 1$ nodes of T are visited and the operation findAllInRange runs in $O(h + s)$ time. We summarize:

Theorem 3.2: *A binary search tree of height h storing n items supports range query operations with the following performance:*

- *The space used is $O(n)$.*
- *Operation* findAllInRange *takes $O(h + s)$ time, where s is the number of elements reported.*
- *Operations* insert *and* remove *each take $O(h)$ time.*

3.3 Index-Based Searching

We began this chapter by discussing how to search in a sorted array, A, which allows us to quickly identify the ith smallest item in the array simply by indexing the cell $A[i]$. As we mentioned, a weakness of this array representation is that it doesn't support efficient updates, whereas a binary search tree allows for efficient insertions and deletions. But when we switched to a binary search tree, we lost the ability to quickly find the ith smallest item in our set. In this section, we show how to regain that ability with a binary search tree.

Suppose, then, that we wish to support the following operation on a binary search tree, T, storing n key-value pairs:

select(i): Return the item with ith smallest key, for $1 \leq i \leq n$.

For example, if $i = 1$, then we would return the minimum item, if $i = n$, then we would return the maximum, and if $i = \lceil n/2 \rceil$, then we would return the median (assuming n is odd).

The main idea for a simple way to support this method is to ***augment*** each node, v, in T so as to add a new field, n_v, to that node, where

- n_v is the number of items stored in the subtree of T rooted at v.

For instance, see Figure 3.13.

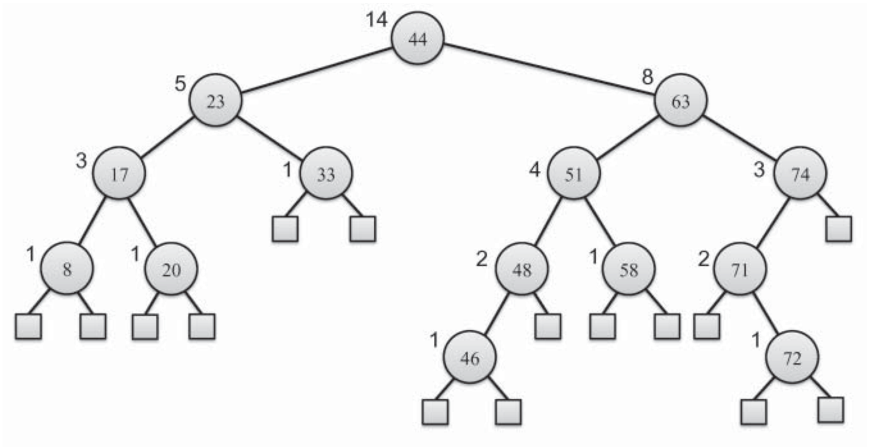


Figure 3.13: A binary search tree augmented so that each node, v, stores a count, n_v, of the number of items stored in the subtree rooted at v. We show the key for each node inside that node and the n_v value of each node next to the node, except for external nodes, which each have an n_v count of 0.

Searching and Updating Augmented Binary Search Trees

Having defined the n_v count for each node in a binary search tree, T, we need to maintain it during updates to T. Fortunately, such updates are easy.

First, note that the n_v count for any external node is 0, so we don't even need to store an actual n_v field for external nodes (especially if they are ***null*** objects). To keep the n_v counts for internal nodes up to date, we simply need to modify the insertion and deletion methods as follows:

- If we are doing an insertion at a node, w, in T (which was previously an external node), then we set $n_w = 1$ and we increment the n_v count for each node v that is an ancestor of w, that is, on the path from w to the root of T.
- If we are doing a deletion at a node, w, in T, then we decrement the n_v count for each node v that is on the path from w's parent to the root of T.

In either the insertion or deletion case, the additional work needed to perform the updates to n_v counts takes $O(h)$ time, where h is the height of T. This updating takes an additional amount of time that is $O(h)$, where h is the height of T, because we spend an additional amount of $O(1)$ time for each node from w to the root of T in either case. (See Figure 3.14.)

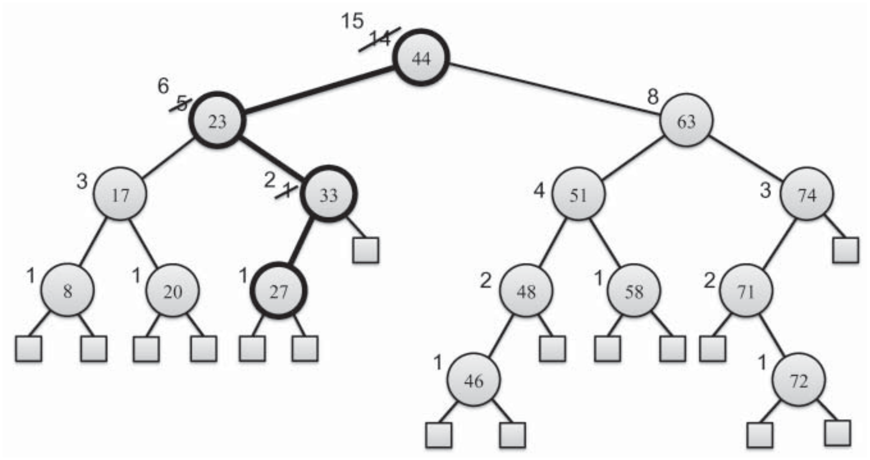


Figure 3.14: A update in an a binary search tree augmented so that each node, v, stores a count, n_v, of the number of items stored in the subtree rooted at v. We show the path taken, along with n_v updates, during this update, which is an insertion of a node with key 27 in the tree from Figure 3.13.

Let us consider how to perform method select(i) on a tree, T, augmented as described above. The main idea is to search down the tree, T, while maintaining the value of i so that we are looking for the ith smallest key in the subtree we are still searching in. We do this by calling the TreeSelect(i, r, T), shown in Algorithm 3.15, where r is the root of T. Note that this algorithm assumes that the n_v count for any external node is 0. (See Figure 3.16.)

```
Algorithm TreeSelect($i, v, T$):
  Input: Search index $i$ and a node $v$ of a binary search tree $T$
  Output: The item with $i$th smallest key stored in the subtree of $T$ rooted at $v$
  Let $w \leftarrow T$.leftChild($v$)
  if $i \le n_w$ then
      return TreeSelect($i, w, T$)
  else if $i = n_w + 1$ then
      return (key($v$), element($v$))
  else
      return TreeSelect($i - n_w - 1, T$.rightChild($v$), $T$)
```

Algorithm 3.15: The TreeSelect algorithm.

The correctness of this algorithm follows from the fact that we are always maintaining i to be the index of the ith smallest item in the subtree we are searching in, so that the item returned will be the correct index. In particular, note that when we recursively search in a right subtree, we first subtract the count of the number of items stored in the left subtree and the parent. The running time for performing this query is $O(h)$, where h is the height of T, since we spend $O(1)$ time per level of T in our search to the node storing the ith smallest key.

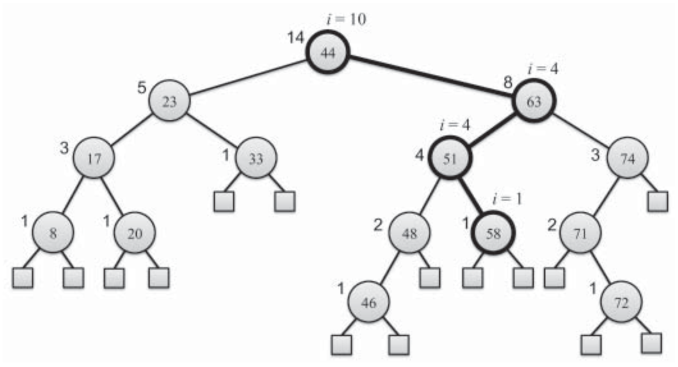


Figure 3.16: A search for the 10th smallest item in a binary search tree augmented so that each node, v, stores a count, n_v, of the number of items stored in the subtree rooted at v. We show the path taken during this search, along with the value, i, that is maintained during this search.

3.4 Randomly Constructed Search Trees

Suppose we construct a binary search tree, T, by a sequence of insertions of n distinct, random keys. Since the only thing that impacts the structure of T is the relative order of the keys, we can assume, without loss of generality, that the keys involved are the integers from 1 to n. That is, we can assume that we are given a random permutation, P, of the keys in the set $\{1, 2, \ldots, n\}$, where all permutations are equally likely, and are asked to build the binary search tree, T, by inserting the keys in P in the given order.

Let v_j denote the node in T that holds the item with key j, and let $D(v_j)$ denote the depth of v_j in T. Note that $D(v_j)$ is a random variable, since T is constructed at random (based on P). Define $X_{i,j}$ to be a 0-1 indicator random variable that is equal to 1 if and only if v_i is an ancestor of v_j, where, for the sake of this analysis, we consider a node to be an ancestor of itself. That is, $X_{i,i} = 1$. We can write

$$D(v_j) = \sum_{i=1}^{n} X_{i,j} \; - \; 1,$$

since the depth of a node is equal to the number of its proper ancestors. Thus, to derive a bound on the expected value of $D(v_j)$, we need to determine the probability that $X_{i,j}$ is 1.

Lemma 3.3: *The node v_i is an ancestor of the node v_j in T if and only if i appears earliest in P of any integer in $R_{i,j}$, the range of integers between i and j, inclusive.*

Proof: If i appears earliest in P of any integer in $R_{i,j}$, then, by definition, it is inserted into T before any of these integers. Thus, at the time when j is inserted into T, and we perform j in T, we must encounter the node v_i as our depth $D(v_i)$ comparison, since there is no other element from $R_{i,j}$ at a higher level in T.

Suppose, on the other hand, that i appears earliest in P of any integer in $R_{i,j}$. Then i is inserted into T before any item with a key in this range; hence, for the search for j, we will encounter v_i at some point along the way, since j has to be located into this interval and, at the depth of v_i, there is no other element from $R_{i,j}$ to use as a comparison key. ■

This fact allows us to then derive the probability that any $X_{i,j}$ is 1, as follows.

Lemma 3.4: *Let $X_{i,j}$ be defined as above. Then* $\Pr(X_{i,j} = 1) \; = \; 1/(|i-j|+1)$.

Proof: There are $|i - j| + 1$ items in the range from i to j, inclusive, and the probability that i appears earliest in P of all of them is 1 over this number, since all the numbers in this range have an equal and independent probability of being earliest in P. The proof follows, then, from this fact and Lemma 3.3. ■

An Analysis Based on Harmonic Numbers

In addition to the above lemmas, our analysis of a randomly constructed binary search tree also involves harmonic numbers. For any integer $n \geq 1$, we define the nth ***harmonic number***, H_n, as

$$H_n = \sum_{i=1}^{n} \frac{1}{i},$$

for which it is well known that H_n is $O(\log n)$. In fact,

$$\ln n \leq H_n \leq 1 + \ln n,$$

as we explore in Exercise C-3.11.

Harmonic numbers are important in our analysis, because of the following.

Lemma 3.5:

$$E[D(v_j)] \quad \leq \quad H_j + H_{n-j+1} - 1.$$

Proof: To see this relationship, note that, by the linearity of expectation,

$$\begin{aligned} E[D(v_j)] &= \sum_{i=1}^{n} E[X_{i,j}] \; - \; 1 \\ &= \sum_{i=1}^{j} E[X_{i,j}] \; + \sum_{i=j+1}^{n} E[X_{i,j}] \; - \; 1 \\ &= \sum_{i=1}^{j} \frac{1}{j-i+1} \; + \sum_{i=j+1}^{n} \frac{1}{i-j+1} \; - \; 1 \\ &\leq \sum_{k=1}^{j} \frac{1}{k} \; + \sum_{k=1}^{n-j+1} \frac{1}{k} \; - \; 1 \\ &= H_j + H_{n-j+1} - 1. \end{aligned}$$

■

So, in other words, the expected depth of any node in a randomly constructed binary search tree with n nodes is $O(\log n)$. Or, put another way, the average depth of the nodes in a randomly constructed binary search tree is $O(\log n)$.

Bounding the Overall Height with High Probability

In addition to the above bound for the average depth of a randomly constructed binary search tree, we might also be interested in bounding the overall height of such a tree, T, which is equal to the maximum depth of any node in T. In order to analyze this maximum depth of a node in T, let us utilize the following Chernoff bound (see Exercise C-19.14).

Let $X_1, X_2, \ldots, X_n$ be a set of mutually independent indicator random variables, such that each X_i is 1 with some probability $p_i > 0$ and 0 otherwise. Let $X = \sum_{i=1}^{n} X_i$ be the sum of these random variables, and let μ' denote an upper bound on the mean of X, that is, $E(X) \le \mu'$. Then, for $\delta > 0$,

$$\Pr(X > (1+\delta)\mu') < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu'}.$$

Using this bound, we can derive the following theorem.

Theorem 3.6: *If T is a randomly constructed binary search tree with $n > 4$ nodes, then the height of T is $O(\log n)$ with probability at least $1 - 1/n$.*

Proof: Recall that $D(v_j) = \sum_{i=1}^{n} X_{i,j} - 1$. Let $L_j = \sum_{i=1}^{j-1} X_{i,j}$ denote the "left" part of this sum and $R_j = \sum_{i=j+1}^{n} X_{i,j}$ denote the "right" part, with both leaving off the term $X_{j,j}$. Then $D(v_j) = L_j + R_j$. So, by symmetry, it is sufficient for us to bound R_j, since a similar bound will hold for L_j. The important observation is that all of the $X_{i,j}$ terms in the definition of R_j are independent 0-1 random variables. This independence is due to the fact that whether i is chosen first in P from $\{j, j+1, \ldots, i\}$ has no bearing on whether $i+1$ is chosen first in P from $\{j, j+1, \ldots, i, i+1\}$. Moreover, $E[R_j] = H_{n-j+1} \le H_n$. Thus, by the above Chernoff bound, for $n > 4$,

$$\Pr(R_j > 4H_n) < \left[\frac{e^3}{4^4}\right]^{H_n} \le \frac{1}{n^{2.5}},$$

since $H_n \ge \ln n$. Therefore, $\Pr(D(v_j) > 8H_n) \le 2/n^{2.5}$, which implies that the probability that ***any*** node in T has depth more than $8H_n$ is at most $2n/n^{2.5} = 2/n^{1.5}$. Since $n > 4$ and H_n is $O(\log n)$, this establishes the theorem. ■

So, if we construct a binary search tree, T, by inserting a set of n distinct items in random order, then, with high probability, the height of T will be $O(\log n)$.

The Problem with Deletions

Unfortunately, if we intersperse random insertions with random deletions, using the standard insertion and deletion algorithms given above, then the expected height of the resulting tree is $\Theta(n^{1/2})$, not $O(\log n)$. Indeed, it has been reported that a major database company experienced poor performance of one its products because of an issue with maintaining the height of a binary search tree to be $O(\log n)$ even while allowing for deletions. Thus, if we are hoping to achieve $O(\log n)$ depth for the nodes in a binary search tree that is subject to both insertions and deletions, even if these operations are for random keys, then we need to do more than simply performing the above standard insertion and deletion operations. For example, we could use one of the balanced search tree strategies discussed in the next chapter.

3.5 Exercises

Reinforcement

R-3.1 Suppose you are given the array $A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$, and you then perform the binary search algorithm given in this chapter to find the number 8. Which numbers in the array A are compared against the number 8?

R-3.2 Insert items with the following keys (in the given order) into an initially empty binary search tree: 30, 40, 50, 24, 8, 58, 48, 26, 11, 13. Draw the tree that results.

R-3.3 Suppose you have a binary search tree, T, storing numbers in the range from 1 to 500, and you do a search for the integer 250. Which of the following sequences are possible sequences of numbers that were encountered in this search. For the ones that are possible, draw the search path, and, for the ones that are impossible, say why.

a. (2, 276, 264, 270, 250)

b. (100, 285, 156, 203, 275, 250)

c. (475, 360, 248, 249, 251, 250)

d. (450, 262, 248, 249, 270, 250)

R-3.4 Suppose T is a binary search tree of height 4 (including the external nodes) that is storing all the integers in the range from 1 to 15, inclusive. Suppose further that you do a search for the number 11. Explain why it is impossible for the sequence of numbers you encounter in this search to be $(9, 12, 10, 11)$.

R-3.5 Draw the binary search trees of minimum and maximum heights that store all the integers in the range from 1 to 7, inclusive.

R-3.6 Give a pseudocode description of an algorithm to find the element with smallest key in a binary search tree. What is the running time of your method?

R-3.7 Draw the binary search tree that results from deleting items with keys 17, 28, 54, and 65, in this order, from the tree shown in Figure 3.7b.

R-3.8 A certain Professor Amongus claims that the order in which a fixed set of elements is inserted into a binary search tree does not matter—the same tree results every time. Give a small example that proves Professor Amongus wrong.

R-3.9 Suppose you are given a sorted set, S, of n items, stored in a binary search tree. How many different range queries can be done where both of the values, k_1 and k_2, in the query range $[k_1, k_2]$ are members of S?

R-3.10 Suppose that a binary search tree, T, is constructed by inserting the integers from 1 to n in this order. Give a big-Oh characterization of the number of comparisons that were done to construct T.

R-3.11 Suppose you are given a binary search tree, T, which is constructed by inserting the integers in the set $\{1, 2, \ldots, n\}$ in a random order into T, where all permutations of this set are equally likely. What is the average running time of then performing a $\mathsf{select}(i)$ operation on T?

R-3.12 If one has a set, S, of n items, where n is even, then the median item in S is the average of the ith and $(i+1)$st smallest elements in S, where $i = n/2$. Describe an efficient algorithm for computing the median of such a set S that is stored in a binary search tree, T, where each node, v, in T is augmented with a count, n_v, which stores the number of items stored in the subtree of T rooted at v.

R-3.13 What is H_5, the 5th harmonic number?

Creativity

C-3.1 Suppose you are given a sorted array, A, of n distinct integers in the range from 1 to $n+1$, so there is exactly one integer in this range missing from A. Describe an $O(\log n)$-time algorithm for finding the integer in this range that is not in A.

C-3.2 Let S and T be two ordered arrays, each with n items. Describe an $O(\log n)$-time algorithm for finding the kth smallest key in the union of the keys from S and T (assuming no duplicates).

C-3.3 Describe how to perform the operation findAllElements(k), which returns every element with a key equal to k (allowing for duplicates) in an ordered set of n key-value pairs stored in an ordered array, and show that it runs in time $O(\log n + s)$, where s is the number of elements returned.

C-3.4 Describe how to perform the operation findAllElements(k), as defined in the previous exercise, in an ordered set of key-value pairs implemented with a binary search tree T, and show that it runs in time $O(h + s)$, where h is the height of T and s is the number of items returned.

C-3.5 Prove, by induction, that the height of a binary search tree containing n items is at least $\lceil \log(n+1) \rceil$.

C-3.6 Describe how to perform an operation removeAllElements(k), which removes all key-value pairs in a binary search tree T that have a key equal to k, and show that this method runs in time $O(h + s)$, where h is the height of T and s is the number of items returned.

C-3.7 Let S be an ordered set of n items stored in a binary search tree, T, of height h. Show how to perform the following method for S in $O(h)$ time:

countAllInRange(k_1, k_2): Compute and return the number of items in S with key k such that $k_1 \leq k \leq k_2$.

C-3.8 Describe the structure of a binary search tree, T, storing n items, such that T has height $\Omega(n^{1/2})$ yet the average depth of the nodes in T is $O(\log n)$.

C-3.9 Suppose n key-value pairs all have the same key, k, and they are inserted into an initially empty binary search tree using the algorithm described in Section 3.1.3. Show that the height of the resulting tree is $\Theta(n)$. Also, describe a modification to that algorithm based on the use of random choices and show that your modification results in the binary search tree having height $O(\log n)$ with high probability.

C-3.10 Suppose that each row of an $n \times n$ array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in $O(n \log n)$ time (not $O(n^2)$ time!) for counting the number of 1's in A.

C-3.11 (For readers familiar with calculus): Use the fact that, for a decreasing integrable function, f,

$$\int_{x=a}^{b+1} f(x)dx \;\le\; \sum_{i=a}^{b} f(i) \;\le\; \int_{x=a-1}^{b} f(x)dx,$$

to show that, for the nth harmonic number, H_n,

$$\ln n \;\le\; H_n \;\le\; 1 + \ln n.$$

C-3.12 Without using calculus (as in the previous exercise), show that, if n is a power of 2 greater than 1, then, for H_n, the nth harmonic number,

$$H_n \le 1 + H_{n/2}.$$

Use this fact to conclude that $H_n \le 1 + \lceil \log n \rceil$, for any $n \ge 1$.

Applications

A-3.1 Suppose you are asked to automate the prescription fulfillment system for a pharmacy, MailDrugs. When an order comes in, it is given as a sequence of requests, "x_1 ml of drug y_1," "x_2 ml of drug y_2," "x_3 ml of drug y_3," and so on, where $x_1 < x_2 < x_3 < \cdots < x_k$. MailDrugs has a practically unlimited supply of n distinctly sized empty drug bottles, each specified by its capacity in milliliters (such 150 ml or 325 ml). To process a drug order, as specified above, you need to match each request, "x_i ml of drug y_i," with the size of the smallest bottle in the inventory than can hold x_i milliliters. Describe how to process such a drug order of k requests so that it can be fulfilled in $O(k \log(n/k))$ time, assuming the bottle sizes are stored in an array, T, ordered by their capacities in milliliters.

A-3.2 Imagine that you work for a database company, which has a popular system for maintaining sorted sets. After a negative review in an influential technology website, the company has decided it needs to convert all of its indexing software from using sorted arrays to an indexing strategy based on using binary search trees, so as to be able to support insertions and deletions more efficiently. Your job is to write a program that can take a sorted array, A, of n elements, and construct a binary search tree, T, storing these same elements, so that doing a binary search for any element in T will run in $O(\log n)$ time. Describe an $O(n)$-time algorithm for doing this conversion.

A-3.3 Suppose you work for a computer game company, which is designing a first person shooting game. In this game, players stand just outside of a circular playing field and shoot at targets inside the circle. There are a lot of players and targets, however, so, for any given player, it only makes sense to display the targets that are close by that player. Thus, whenever a new target, t, appears, the game should only make it visible to the players that are in the same "zone" as t. In order to support the fast processing of such queries, you offer to build a binary space partitioning (BSP) tree, T, for use in the game engine. You are given the (x, y) coordinates of all n players, which are in no particular order, but are constrained to all lie at different locations on a circle, C. Your job is to design an efficient algorithm for building such a BSP tree, T, so that its height is $O(\log n)$. Thus, the root, r, of T is associated with a line, L, that divides the set of players into two groups of roughly equal size. Then, it should recursively partition each group into two groups of roughly equal size. Describe an $O(n \log n)$-time algorithm for constructing such a tree T. (See Figure 3.17.)

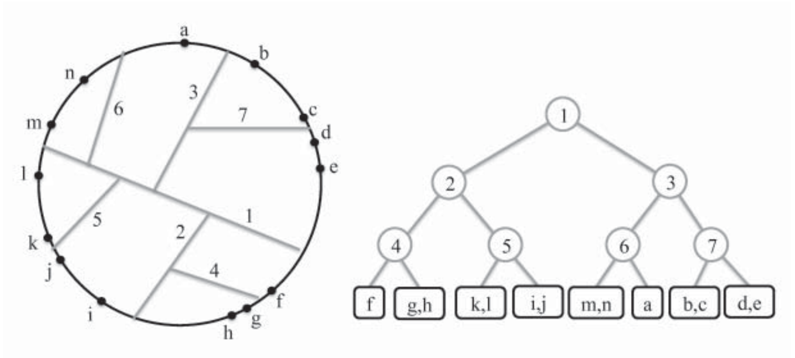


Figure 3.17: A circular environment for a first-person shooter game, where players are represented as points (labeled with letters), together with a two-dimensional BSP tree for this configuration, where dividing lines and nodes are similarly numbered.

A-3.4 It is sometimes necessary to send a description of a binary search tree in text form, such as in an email or text message. Since it is a significant challenge to draw a binary search tree using text symbols, it is useful to have a completely textural way of representing a binary search tree. Fortunately, the structure of a binary search tree is uniquely determined by labeling each node with its preorder and postorder numbers. Thus, we can build a textural representation of a binary search tree T by listing its nodes sorted according to their preorder labels, and listing each node in terms of its contents and its postorder label. For example, the tree of Figure 3.10 would be represented as the string,

$$[(10, 9), (\emptyset, 1), (20, 8), (\emptyset, 2), (30, 7), (\emptyset, 3), (40, 6), (\emptyset, 5), (\emptyset, 6)].$$

Describe an $O(n)$ time method for converting an n-node binary search tree, T, into such a textural representation.

A-3.5 Consider the reversal of the problem from the previous exercise. Now you are the recipient of such a message, containing a textural representation of a binary search tree as described in the previous exercise. Describe an algorithm running in $O(n \log n)$ time, or better, for reconstructing the binary search tree, T, that is represented in this message.

A-3.6 Suppose you are building a first-person shooter game, where virtual zombies are climbing up a wall while the player, who is moving left and right in front of the wall, is trying to knock them down using various weapons. The position of each zombie is represented with a pair, (x, y), where x is the horizontal position of the zombie and y is its height on the wall. The player's position is specified with just a horizontal value, x_p. One of the weapons that a player can use is a bomb, which kills the zombie that is highest on the wall from all those zombies within a given horizontal distance, r, of x_p. Suppose the zombies are stored in a binary search tree, T, of height h, ordered in terms of their horizontal positions. Describe a method for augmenting T so as to answer maximum-zombie queries in $O(h)$ time, where such a query is given by a range $[x_p - r, x_p + r]$ and you need to return the coordinates of the zombie with maximum y-value whose horizontal position, x, is in this range. Describe the operations that must be done for inserting and deleting zombies as well as performing maximum-zombie queries.

A-3.7 The first-century historian, Flavius Josephus, recounts the story of how, when his band of 41 soldiers was trapped by the opposing Roman army, they chose group suicide over surrender. They collected themselves into a circle and repeatedly put to death every third man around the circle, closing up the circle after every death. This process repeated around the circle until the only ones left were Josephus and one other man, at which point they took a new vote of their group and decided to surrender after all. Based on this story, the ***Josephus problem*** involves considering the numbers 1 to n arranged in a circle and repeatedly removing every mth number around the circle, outputting the resulting sequence of numbers. For example, with $n = 10$ and $m = 4$, the sequence would be

4, 8, 2, 7, 3, 10, 9, 1, 6, 5.

Given values for n and m, describe an algorithm for outputting the sequence resulting from this instance of the Josephus problem in $O(n \log n)$ time.

Chapter Notes

Interestingly, the binary search algorithm was first published in 1946, but was not published in a fully correct form until 1962. For some lessons to be learned from this history, please see the related discussions in Knuth's book [131] and the papers by Bentley [28] and Levisse [142]. Another excellent source for additional material about binary search trees is the book by Mehlhorn in [157]. In addition, the handbook by Gonnet and Baeza-Yates [85] contains a number of theoretical and experimental comparisons among binary search tree implementations. Additional reading can be found in the book by Tarjan [207], and the chapter by Mehlhorn and Tsakalidis [160].

Our analysis of randomly constructed binary search trees is based on the analysis of randomized search trees by Seidel and Aragon [191]. The analysis that a random sequence of insertions and deletions in a standard binary search tree can lead to it having $\Theta(n^{1/2})$ depth is due to Culberson and Munro [53]. The report of a database company that experienced poor performance of one of its products due to an issue with how to maintain a balanced binary search tree subject to insertions and deletions is included in a paper by Sen and Tarjan [193].