18 Binary Search and Binary Search Trees

18.1 Introduction

Binary search is a much faster alternative to sequential search for sorted lists. Binary search is closely related to binary search trees, which are a special kind of binary tree. We will look at these two topics in this chapter, studying the complexity of binary search, and eventually arriving at a specification for a BinarySearchTree class.

18.2 Binary Search

When people search for something in an ordered list (like a dictionary or a phone book), they do not start at the first element and march through the list one element at a time. They jump into the middle of the list, see where they are relative to what they are looking for, and then jump either forward or backward and look again, continuing in this way until they find what they are looking for, or determine that it is not in the list.

Binary search takes the same tack in searching for a key in a sorted list: the key is compared with the middle element in the list. If it is the key, the search is done. If the key is less than the middle element, then the process is repeated for the first half of the list. If the key is greater than the middle element, then the process is repeated for the second half of the list. Eventually, either the key is found in the list, or the list is reduced to nothing (the empty list), at which point we know that the key is not present in the list.

This approach naturally lends itself to a recursive algorithm, which we show in Ruby below.

```
def rb_search(array, key)
  return nil if array.empty?
  m = array.size/2
  return m if key == array[m]
  return rb_search(array[0…m],key) if key < array[m]
  index = rb_search(array[m+1..-1],key)
  index ? m+1+index : nil
end
```

Figure 1: Recursive Binary Search

Search algorithms traditionally return the index of the key in the list or -1 if the key is not found; in Ruby we have a special value for undefined results, so we return `nil` if the key is not in the array. Note that although the algorithm has the important precondition that the array is sorted, checking this would take far too much time, so it is not checked.

The recursion stops when the array is empty and the key has not been found. Otherwise, the element at index `m` in the middle of the array is checked. If it is the key, the search is done and index `m` is returned; otherwise, a recursive call is made to search the portion of the list before or after `m` depending on whether the key is less than or greater than `array[m]`.

Although binary search is naturally recursive, it is also tail recursive. Recall that a tail recursive algorithm is one in which at a recursive call is the last action in each activation of the algorithm, and that tail recursive algorithms can always be converted to non-recursive algorithms using only a loop and no stack. This is always more efficient and often simpler as well. In the case of binary search, the non-recursive algorithm is about equally complicated, as the Ruby code in Figure 2 below shows.

```
def binary_search(array, key)
  lb, ub = 0, array.size-1
  while (lb <= ub)
    m = (ub+lb)/2
    return m if key == array[m]
    if key < array[m]
      ub = m-1
    else
      lb = m+1
    end
  end
  return nil
end
```

Figure 2: Non-Recursive Binary Search

To analyze binary search, we will consider its behavior on lists of size n and count the number of comparisons between list elements and the search key. Traditionally, the determination of whether the key is equal to, less than, or greater than a list element is counted as a single comparison even though it may take two comparisons in most programming languages.

Binary search does not do the same thing on every input of size n. In the best case, it finds the key in the middle of the list, doing only a single comparison. In the worst case, the key is not in the list, or is found when the sub-list being searched has only one element. We can easily generate a recurrence relation and initial conditions to find the worst case complexity of binary search, but we will instead use a binary search tree to figure this out.

Suppose that we construct a binary tree from a sorted list as follows: the root of the tree is the element in the middle of the list; the left child of the root is the element in the middle of the first half of the list; the right child of the root is the element in the middle of the second half of the list, and so on. In other words, the vertices of the binary tree are filled according to the order in which the values would be encountered during a binary search of the list. To illustrate, consider the binary tree in Figure 3 made from <*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*> in the way just described.

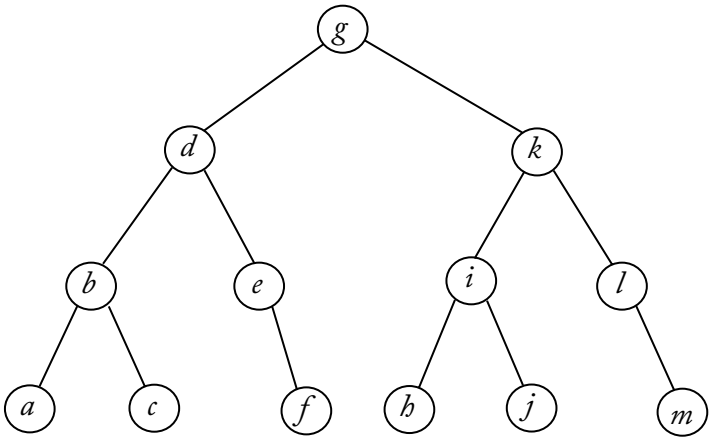

Figure 3: A Binary Tree Made from a List

A tree built this way has the following interesting properties:

- It is a full binary tree so its height is always floor(lg n).
- For every vertex, every element in its left sub-tree (if any) is less than or equal to the element at the vertex, and every element in its right sub-tree (if any) is greater than or equal to the element at the vertex.
- If we traverse the tree inorder, we visit the vertices in the order of the original list, that is, in sorted order.

The first property tells us the worst case performance of binary search because a binary search will visit each vertex from the root to a leaf in the worst case. The number of vertices on these paths is the height of the tree plus one, so $W(n) = floor(\lg n) + 1$. We can also calculate the average case by considering each vertex equally likely to be the target of a binary search and figuring out the average length of the path to each vertex. This turns out to be approximately lg n for both successful and unsuccessful searches. Hence, on average and in the worst case, binary search makes $O(\lg n)$ comparisons, which is very good.

18.3 Binary Search Trees

The essential characteristic of the binary tree we looked at above is the relationship between the value at a vertex and the values in its left and right sub-trees. This is the basis for the definition of binary search trees.

> **Binary search tree**: A binary tree whose every vertex is such that the value at each vertex is greater than the values in its left sub-tree, and less than the values in its right sub-tree.

Binary search trees are an important data type that retains the property that traversing them in order visits the values in the vertices in sorted order. However, a binary search tree may not be full, so its height may be greater than floor(lg n). In fact, a binary search tree whose every vertex but one has only a single child will have height n-1.

Binary search trees are interesting because it is fast both to insert elements into them and fast to search them (provided they are not too long and skinny). This contrasts with most collections, which are usually fast for insertions but slow for searches, or vise versa. For example, elements can be inserted into an (unsorted) LinkedList quickly, but searching a LinkedList is slow, while a (sorted) ArrayList can be searched quickly with binary search, but inserting elements into it to keep it sorted is slow.

The *binary search tree of T* ADT has as its carrier set the set of all binary search trees whose vertices hold a value of type T. It is thus a subset of the carrier set of the binary tree of T ADT. The operations in this ADT includes all the operations of the binary tree ADT, with the addition of a precondition on *buildTree()*, shown below. The list below also includes two operations added to the binary search tree ADT.

> *buildTree*(v,t_l,t_r)—Create and return a new binary tree whose root holds the value v and whose left and right subtrees are t_l and t_r. Its precondition is that v is greater than any value held in t_l and less than any value held in t_r.
>
> *add*(t,v)—Put v into a new vertex added as a leaf to t, preserving the binary search tree property, and return the resulting binary search tree. If v is already in t, then t is unchanged.
>
> *remove*(t,v)—Remove the vertex holding v from t, if any, while preserving the result as a binary search tree, and return the resulting binary search tree.

This ADT is the basis for a BinarySearchTree class.

18.4 The Binary Search Tree Class

A BinarySearchTree is a kind of BinaryTree, so the BinarySearchTree class is a sub-class of BinaryTree. Its constructor needs a precondition to make sure that trees are constructed properly. It can also override the contains() operation to be more efficient. Otherwise, it only needs to implement the three operations pictured in Figure 4 below.

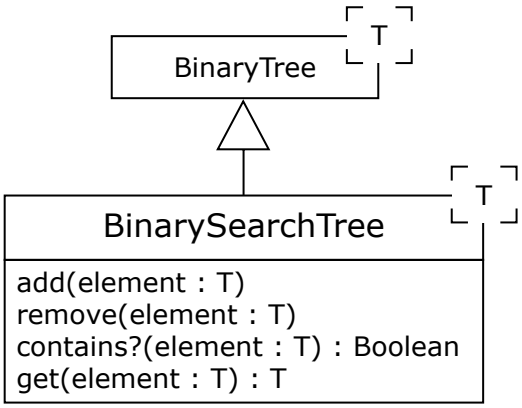

Figure 4: The BinarySearchTree Class

The add() operation puts an element into the tree by making a new child node at a spot that preserves the binary search tree's integrity. If the element is already in the tree, then the element passed in replaces the value currently stored in the tree. In this way, a new record can replace an old one with the same key (more about this in later chapters).

The get() operation returns the value stored in the tree that is "equal" to the element sent in. It is intended to fetch a record from the tree with the same key as a dummy record supplied as an argument, thus providing a retrieval mechanism (again, we will discuss this more later).

The contains?() and get() operations both search the tree by starting at its root and moving down the tree, mimicking a binary search. If the desired value is at the root (the best case), this requires only one comparison, so $B(n)$ is in $O(1)$. In the worst case, when the tree is effectively a list, this requires $O(n)$ comparisons. Empirical studies have shown that when binary search trees are built by a series of insertions of random data, they are more or less bushy, and their height is not too much more than lg n, so the number of comparisons is in $O(\lg n)$ on average.

The add() operation takes a path down the tree to the spot where the new element would be found during a search and adds a new leaf node to hold it. This again requires $O(1)$ comparisons in the best case, $O(\lg n)$ comparisons in the average case, and $O(n)$ comparisons in the worst case. Finally, the remove() operation must first find the deleted element and then manipulate the tree to remove the node holding the element in such a way that it is preserved as a binary search tree. This operation also takes $O(1)$ time in the best case, $O(\lg n)$ time in the average case, and $O(n)$ time in the worst case.

Binary search trees thus provide very efficient operations except in the worst case. There are several kinds of balanced binary search trees whose insertion and deletion operations keep the tree bushy rather than long and skinny, thus eliminating the poor worst case behavior. We will not study balanced binary search trees.

18.5 Summary and Conclusion

Binary search is a very efficient algorithm for searching ordered lists, with average and worst case complexity in $O(\lg n)$. We can represent the workings of binary search in a binary tree to produce a full binary search tree. Binary search trees have several interesting properties and provide a kind of collection that features excellent performance for insertion, deletion, and search, except in the worst case. We can also traverse binary search trees inorder to access the elements of the collection in sorted order.

18.6 Review Questions

1. Why can recursion be removed from the binary search algorithm without using a stack?
2. If a binary tree is made from an ordered list of 100 names by placing them into the tree to mimic a binary search as discussed in the text, what is the height of the resulting tree?
3. Approximately how many comparisons would be made by binary search when searching a list of one million elements in the best, worst, and average cases?
4. What advantage does a binary search tree have over collections like ArrayList and LinkedList?

18.7 Exercises

1. A precondition of binary search is that the searched array is sorted. What is the complexity of an algorithm to check this precondition?
2. Write and solve a recurrence relation for the worst case complexity of the binary search algorithm.
3. *Interpolation search* is like binary search except that it uses information about the distribution of keys in the array to choose a spot to check for the key. For example, suppose that numeric keys are uniformly distributed in an array and interpolation search is looking for the value *k*. If the first element in the array is *a* and the last is *z*, then interpolation search would check for *k* at location (*k*-*a*)/(*z*-*a*) * (*array.length*-1). Write a non-recursive linear interpolation search using this strategy.
4. Construct a binary search tree based on the order in which elements of a list containing the numbers one to 15 would be examined during a binary search, as discussed in the text.
5. Draw all the binary search tree that can be formed using the values *a*, *b*, and *c*. How many are full binary trees?
6. The `BinarySearchTree` `add()` operation does not attempt to keep the tree balanced. It simply work its way down the tree until it either finds the node containing the added element or finds where such a node should be added at the bottom of the tree. Draw the binary search tree that results when values are added to the tree in this manner in the order *m*, *w*, *a*, *c*, *b*, *z*, *g*, *f*, *r*, *p*, *v*.
7. Write the `add()` operation for the `BinarySearchTree` class using the strategy explained in the last exercise.
8. Write the `remove()` operation for the `BinarySearchTree` class. This operation must preserve the essential property of a binary search tree, namely that the value at each node is greater than or equal to the values at the nodes in its left sub-tree, and less than or equal to the values at the nodes in its right sub-tree. In deleting a value, three cases can arise:
   - The node holding the deleted value has no children; in this case, the node can simply be removed.
   - The node holding the deleted value has one child; in this case, the node can be removed and the child of the removed node can be made the child of the removed node's parent.
   - The node holding the deleted value has two children; this case is more difficult. First, find the node holding the successor of the deleted value. This node will always be the left-most descendent of the right child of the node holding the deleted value. Note that this node has no left child, so it has at most one child. Copy the successor value over the deleted value in the node where it resides, and remove the redundant node holding the successor value using the rule for removing a node with no children or only one child above.

   a) Use this algorithm to remove the values *v*, *a*, and *c* from the tree constructed in exercise 6 above.

   b) Write the `remove()` operation using the algorithm above.

18.8 Review Question Answers

1. Recursion can be removed from the binary search algorithm without using a stack because the binary search algorithm is tail recursive, that is, it only calls itself once as its last action on each activation.
2. If a binary tree is made from an ordered list of 100 names by placing them into the tree to mimic a binary search as discussed in the text, the height of the resulting tree is *floor*(lg 100) = 6.
3. When searching a list of one million elements in the best case, the very first element checked would be the key, so only one comparison would be made. In the worst case, *floor*(lg 1000000)+1 = 20 comparison would be made. In the average case, roughly *floor*(lg 1000000) = 19 comparison would be made.
4. An ArrayList and a LinkedList allow rapid insertion but slow deletion and search, or rapid search (in the case of an ordered ArrayList) but slow insertion and deletion. A binary search tree allows rapid insertion, deletion, and search.