25 It's a Long Way To Tipparery...

Introduction

Until now we paid little attention to how much work is involved in performing the computations that our methods represent. However, now that we have seen several different ways of solving a particular problem we need to understand how to choose which technique is better suited for the problem at hand.

In general, when reasoning about computational processes, we are concerned with the amount of time needed to perform a task, the amount of computer memory needed while the computation runs, or the combination of the two measures. We will first explore the *time complexity of algorithms*, that is, the measure of the time it takes to perform a particular computation. For most problems, the more data the computation consumes, the longer it takes to produce the result. Therefore, we represent the time complexity as a map from the size of the data to the estimated time it takes to perform the task for data of that size.

Three Search Algorithms

Let us first consider the problem of determining whether the given collection of data contains the given item. We first recall three solutions of this problem, then we will reason about their advantages and shortcomings. The first solution comnsumes an IRange iterator and traverses over the data ina sequential order. The second solution is searching for an item in a binary search tree. The third solution, the *binary search*, is new to us. It takes advantage of the fact that in an *ArrayList* we can access data at any *index* location directly. It also requires that the data be stored in the *ArrayList* in a sorted order.

For now, we assume these data structures already exist and contain the data in the desired form. Later, we may ask about the time needed to both construct the data structure and to then determine whether it contains the given item.

We assume that all objects that represent our data items can be compared using the given *Comparator*.

Linear Search of Data Generated by IRange

Find the given item in the collection of data generated by the given *IRange* iterator:

```
// does the given collection contain the given item,
// using the Comparator to compare two objects
boolean contains( Object obj, IRange it, Comparator comp){
  if (it.hasMore()){

    // is the current item the one we are looking for
    if (comp.compare(it.current(), obj) == 0)
      return true;

    else
      // continue the search in the rest of the data
      return this.contains(it.next() , comp, obj);
  }

  // no more data to search through
  else
    return false;
}
```

Binary Tree Search

Assume the binary search tree is represented by three classes, *ABST*, *Leaf*, and *Node*, where *Node* contains three fields: *Object data*, *ABST left*, and *ABST right*. The method *contains* is then defined as:

```
// in the class ABST:
// does this binary search tree contain the given item,
// using the Comparator to compare two objects
abstract boolean contains(Object obj, Comparator comp);

// in the class Leaf:
boolean contains(Object obj, Comparator comp){
  return false;
}
```

```
// in the class Node:
boolean contains(Object obj, Comparator comp){
   if (comp.compare(this.data, obj) == 0)
      return true;
   else
      if (comp.compare(this.data, obj) < 0)
         return this.right.contains(obj, comp);
      else
         return this.left.contains(obj, comp);
}
```

Binary Search of a Sorted ArrayList

This algorithms is a true *divide and conquer* kind of algorithm. To find out whether the item appears in a sorted *ArrayList* we first ask whether it is the middle item. If it is, the search is over. If the given item is smaller than the item in the middle, the search continues only in the *lower half*. Otherwise, the search continues in the *upper half* of the *ArrayList*.

Because we need to search in only a part of the given *ArrayList*, our method will consume the lower and upper bound for the search, in addition to the given *Object* and *Comparator*:

```
// determine whether the given ArrayList contains the given object
// among the elements at index locations between low and high
// assert: high − low >= 0, low >= 0, high < arlist.size()
boolean contains(ArrayList arlist,
                 int low, int high,
                 Object obj,
                 Comparator comp){


   // only one or two elements left
   if (high − low <= 1)
      return comp.compare(obj, arlist.get(low) == 0) ||
             comp.compare(obj, arlist.get(high) == 0);
   else
      // compare with the middle for equality
      if (comp.compare(obj, arlist.get((low + high)/2)) == 0)
         return true;
```

```
        else
          // obj smaller than the middle item
          if (comp.compare(obj, arlist.get((low + high)/2)) < 0)
            return contains(arlist, low, (low + high)/2 − 1, obj, comp);

          else
            // obj larger than the middle item
            return contains(arlist, (low + high)/2 + 1, high, obj, comp);
    }
```

Exercise.

Make examples of the use of the binary search, including the expected outcome. Run your examples as tests.

Complexity of Search Algorithms

When measuring the time complexity of computation, we typically count the number of basic operations that need to be performed to sove a problem of the given size. When performing a search, we consider each comparison to be the basic operation. The reasoning is that although for each comparison we also need to perform some other operations, such as advancing the iterator, finding the index of the middle item, recursing to the left or right subtree of a binary search tree, these operations are performed as a consequence of a (possibly previous) comparison.

Linear Search of Data Generated by IRange

Suppose we want to find an item in a list of 10 items generated by an iterator. In the *best case*, the item we are searching for is at the begining of the list and our search algorithm produces a true result after just *one* comparison. In the *worst case* the item is at the end of the list, or is not in the list at all. In that case, we need to compare the given item with every item in the list for a total of 10 comparisons. If the list had n items, we would need n comparisons for the worst case. On the average, to find whether an item is in the list takes about $n/2$ steps.

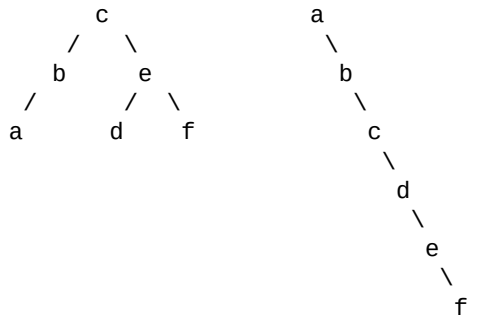
Rather than writing a mathematical function, the complexity of algorithms is defined in terms of *Big-O*. We say, **the worst case of the linear search is on the order of n, and write it as $O(n)$.** (Please, read chapter 29 of HtDP for an explanation of the meaning of *Big-O*.) It will be clear that the complexity of the search in the average case is also $O(n)$. However, the

best case is different. Here we can produce the answer after just one comparison, and so the best case is on the order of $O(1)$ - a constant time. That means, it takes the same amount of time regardless of the size of the data we search.

Binary Tree Search

For the binary tree search, the best case is the same as for the data generated by an iterator. If the item we are searching for is in the root node of the tree, we will know after just one comparison, so it is again on the order of $O(1)$

The rest of the analysis is much harder. We already know that the shape of a binary search tree can vary widely. We have seen the following shapes for the trees with six nodes:



In the best case the nodes in the top levels all have both, the left and the right subtree, and only at the last level are there nodes without both children. The height of the tree (the number of levels) is on the order of $O(\log(n))$. The average time needed to determine whether a binary search tree contains the given item is on the order of $O(n)$. Intuitively we believe this. However, a formal proof is much harder and is not given here.

In the worst case, when the binary tree has the degenerate shape, the search requires $n - 1$ comparisons, and so the worst case search is on the order of $O(n)$.

Binary Search of a Sorted ArrayList

As before, the best case happens when the item is found on the first try. In the worst case, we keep narrowing down the size of the interval between *low* and *high*, until the difference is either 0 or 1. After each comparison, the size of the interval becomes one half of the current size. That means that in

the worst case we only need log(n) comparisons. The average case is also on the order of log(n).

The following table summarizes our results:

Algorithm \ Case	Best Case	Average Case	Worst Case
Linear Search of IRange Data	O(1)	O(n)	O(n)
Binary Search Tree Search	O(1)	O(log(n))	O(n)
Binary Search of ArrayList	O(1)	O(log(n))	O(log(n))

Complexity of Insertions

We now consider the problem of inserting a new item into an existing structure. There are several variants of this problem. We can select whether to insert an item at the beginning or at the end of the structure, provided the structure is organized in a linear fashion (a list or an *ArrayList*). Inserting at the begining of a recursively defined list consists of only one step: constructing a new list in which the new item is the first and the original list is the rest. Inserting at the end of an *ArrayList* is usually also done in constant time. Please, read the documentation of ArrayList for the explanation of how the capacity of an *ArrayList* is handled.

It is much harder to insert a new item at the end of a recursively defined list. We must traverse the entire list before we get to the end. In this case, the operation that helps us estimate the time needed to perform the task is the traversal, the advancing to the next element of the list. It is clear that this insertion is on the order of $O(n)$.

Inserting a new item at the begining of an *ArrayList* presents another problem. The implementation of the *ArrayList* keeps the references to all objects in a sequence of slots, so that the reference to the element at k is in the *k-th* slot in the sequence of references. If a new element is inserted at index 0, the references to all objects in the *ArrayList* have to be moved to the next slot. (The last reference is moved to a new slot, then the next to last is moved to the last slot, etc.) The time it takes to move the references to the new slots represents most of the work in this algorithm. So, in this case, we measure the time complexity by estimating the number of move operations. Again, this algorithm is on the order of $O(n)$.

Another situation to consider is when the data is already sorted and we wish for the insertion to preserve the *sortedness* property. If the data is in a linear list, we need to compare it with the items in the list in a sequential order to find where to insert. This would require on the order of $O(n)$ steps. In an *ArrayList* we would traverse the first part of the list, comparing the given item with the elements of the *ArrayList*, and then, in order to insert the element in the middle, the remaining references wold have to be moved over to the 'right'. Again, the algorithms is on the order of $\{O(n)$.

Finally, inserting a new item into a binary search tree is on the average on the order of $O(\log(n)$.

Our results are summarized in the following table:

Algorithm \ Case	Complexity
Insert at the front of a list	O(1)
Insert at the end of a list	O(n)
Insert into a sorted list	O(n)
Insert into a Binary Search Tree	O(log(n))
Insert at the front of an ArrayList	O(n)
Insert at the end of an ArrayList	O(1)
Insert into a sorted ArrayList	O(n)