Insertion Sort: an algorithm for sorting an array

Let's use arrays to solve a problem that comes up often in programming, namely *sorting*. Suppose we have an array of objects that is in no particular order and the objects are such that it is meaningful to talk about an ordering e.g. the objects might be numbers, or they might be strings which can be sorted alphabetically. Given the under-ordered items in an array, we would like to rearrage the items in the array such that they are sorted.

There are several possible algorithms for doing so. "Insertion sort" is one of the simplest to describe. The basic idea of the algorithm is to assume that the k elements of the array (indices $0, \ldots, k-1$) are already in the correct order, and then insert element at index k into its correct position with respect to the first k elements. We start with $k = 0$. The first element is clearly in its correct position if we only have one element, so there is nothing to do.

Now suppose that the first k elements are correctly ordered relative to each other. How do we put the element at index k into its correct position? The idea is to look backwards from index k until we find the right place for it. Element $a[k-1]$ is the largest of all $a[0], \ldots, a[k-1]$ by assumption, since the first k elements are in their correct order. When stepping backwards, if the element in the next position is greater than $a[k]$, then we move that element forward in the array to make room for the $a[k]$. If, on the other hand, we find an element that is less than (or equal to) $a[k]$ then we go no further.

The algorithm is listed below. You should step through it and make sure you follow it. I suggest you also have a look at an applet that allows you to visualize how the algorithm works, such as `http://tech-algorithm.com/articles/insertion-sort`.

ALGORITHM: INSERTION SORT
INPUT: array $a[\,]$ with N elements that can be compared $(<, =, >)$
OUTPUT: array $a[\,]$ containing the same elements but in increasing order

```
for k = 1 to N − 1 do
    tmp ← a[k]
    i ← k
    while (i > 0) & (tmp < a[i − 1]) do
        a[i] ← a[i − 1]
        i ← i − 1
    end while
    a[i] = tmp
end for
```

Analysis of insertion sort

Suppose you are given an array $a[\,]$ which is of size N, and you step through this algorithm line by line. How many steps will you take? Interestingly, the answer depends on the data, not only on the size N.

Suppose the operations inside the **for** loop but outside the **while** loop take c_1 time steps, and the operations inside the **while** loop take c_2 times steps. Then, in the worst case, you need to take

about $c_1N + c_2(1 + 2 + \cdots + N - 1)$, where the latter expression occurs in the case that, for each k, the **while** loop decrements i all the way from k back to 0. But you should recall from high school math that

$$1 + 2 + \ldots N - 1 = \frac{N(N-1)}{2}$$

so you can see that in the worst case the algorithm takes time that depends on N^2. This worst case scenario occurs in the case that the array is already sorted, but it is sorted in the wrong direction, namely *from largest to smallest.*

In the "best" case, the array is already correctly sorted from smallest to largest. Then the condition tested in the **while** loop will be false every time (since $a[i-1] < tmp$), and so each time we hit the **while** statement, it will take a *constant* amount of time. This is the *best case* scenario, in the sense that the algorithm executes the fewest operations in this case. Since there are N passes through the **for** loop, the time taken is proportional to N.

array as a "list"

In the next part of the course, we consider data structures that hold an ordered set of elements. By "ordered", I mean that we can talk about the first element, the second element, etc. Such a set of ordered objects is often called a *list.* One also refers to list data structures as "linear data structures".

The *array* is a natural data structure for representing a list. If we have *size* elements in the list, then we store these elements in positions $0, 1, \ldots, size - 1$. An array allows us to access any of the elements in the list by providing the position (index) of the element. Arrays can sometimes be an awkward way to represent a list, however. If you want to insert a new element into the array then you have to make room for that element in the array. For example, if you want to insert a new element at the front (into array slot 0), then you need to move all the elements ahead one position ($i \to i + 1$). We saw this type of problem with the insertion sort algorithm.

```
//  add new element to front of the list
//  assuming that there is room left in the array
//
    for (i = size; i > 0; i--)
       a[i] = a[i-1]
    a[0] = new element
    size = size + 1
```

Note that you need the for loop to go backwards. Think what happens if you go forwards!

A similar issue arise if you want to remove an element, namely you need to shift all elements back one position ($i \to i - 1$).

```
//  remove the element at front of the list
//
    for (i = 1; i < size-1; i++)
       a[i-1] = a[i]
    a[size-1] = null
    size = size - 1
```

If you have *size* elements in the array, then adding or removing the element with index 0 takes *size* operations. This is obviously inefficient, especially if we are doing alot of adding (also called "insertions") and removals (also called "deletions") at the front of the list.

Compare the above to the problem of adding or removing at the last element in the list:

```
//  add new last element to the list
//  assuming that there is room left in the array
//
    a[size] = new element
    size = size + 1


//  remove the last element from the list
//
    a[size-1] = null
    size = size - 1
```

These algorithms take constant time, i.e. independent of the number of elements in the array.

One note about the above "algorithms" is that, when we remove an element, we have been careful to set the slot to have a `null` value. This is not strictly speaking necessary, since we have a `size` variable which keeps track of how big the list is and hence it keeps track of where the elements are, namely in slots 0 to `size-1`.

Singly Linked lists

Let's look at an alternative data structure for representing a list. Define a list *node* which contains:

- an element of a list (this could be the element itself or it could be a reference to an element)
- a reference to the `next` node in the list.

In Java, we could have a node class defined as follows:

```
class  SNode{
   Type         element;
   SNode        next;
}
```

where `Type` is the type of the object in the list. To define the list itself, we would define another class:

```
class  SLinkedList{
   SNode        head;
   SNode        tail;
   integer      size;
}
```

The field `head` points to the first node in the list and the field `tail` points to the last node in the list. If there is only one node in the list, then `head` and `tail` would point to the same node.

Let's look at a few methods for manipulating a linked list:

```
addFirst( newNode ){
   newNode.next = head;
   head = newNode;
   size = size + 1;
}
```

Here we assume the input parameter is a node rather than an element. If we were to write this "algorithm" in Java, we would need to be more careful about such things. Don't concern yourself with this detail here. Instead, notice the order of the two instructions. The order matters!

```
removeFirst(){
   tmp = head;
   head = head.next;
   tmp.next = null;
   size = size - 1;
}
```

Notice that we have used `tmp` here. We could have just had one instruction (`head = head.next`) but then the old first node in the list would still be pointing to the new first node in the list, even though it isn't part of the list. (You might argue that since the old first node is not part of the list anymore, then you don't care if it points to the new first node. In that case, the first and third instructions above which involve `tmp` would be unnecessary.)

Also note that the `removeFirst()` method ignores certain cases. For example, if there is only one element in the list, then removing the first means that we are also removing the last. In that case, we should set the `tail` reference to `null`.

Adding a node at the tail can be done in a small number of steps.

```
addLast( newNode ){
   tail.next = newNode;
   tail = tail.next;
   size = size + 1;
}
```

Removing the last node from a list, however, requires many steps. The reason is that you need to modify the `next` reference of the node that comes before the *tail* node which you want to remove. But you have no way to directly access the node that comes before `tail`, and so you have to find this node by searching from the front of the list.

The algorithm begins by checking if the list has just one element. If it does, then the last node is the first node and this element is removed. Otherwise, it scans the list for the next-to-last element.

```
removeLast(){
   if (head == tail){
      head = null;
      tail = null;
      size = 0;
   }
   else{
     tmp = head;
     while (tmp.next != tail){
        tmp = tmp.next;
     }
     tmp.next = null;
     tail = tmp;
     size = size - 1;
   }
}
```

This method requires about `size` steps. This is significantly more expensive than what we had with an array implementation, where we had a constant cost in removing the last element from a list. We will come back to this comparison at the end of next lecture, when we compare arrays with singly and doubly linked lists.