211lab13_S11

This week's lab is a bit of a potpourri of topics:

- **Quicksort Algorithm**
- **Variable Argument Lists**
- **Two Visitor Styles**
- **Binary Search Algorithm**
- **Insertion into a Doubly-linked list**

Quicksort

Quicksort is the most common sorting algorithm in use today. In light of its technological impact, it's good to know how it works.

Fundamentally, Quicksort is akin to merge sort, in that it seeks to partition the set into two pieces and then recursively sort each piece. Mergesort simply whacks the set into two equal-sized pieces, which has the side effect of requiring a non-trivial merging process to re-join the sorted sub-pieces.

Rather than split the set (array) into two arbitrary and equal-sized pieces, Quicksort divides the set into subsets where one subset contains values that are all *smaller* than a given "pivot value" while the other subset are all values that are *greater* than the pivot. Some implementations put the pivot in the set of smaller numbers and some put it in the set of larger numbers and still others keep it completely separate.

Joining the sorted subsets is trivial because we already know that one set is completely smaller than the other set.

Which value to use as the pivot? Consider this: on average, for any randomly chosen element in the set (array), the value of that element is the mean of all the values. Thus, on average, half of the values in the set are smaller than any randomly chosen value and half are thus larger. What does this all mean? It means that it doesn't matter which element we choose to be our pivot because on average, half of the values will be larger than our pivot and half will be smaller, which is what we want. So, why not pick the first element? It's statistically no different than any other elenent and its real easy to find!

On average, quicksort will be faster than mergesort because the partitioning process of breaking the set into "smaller" and a "larger" subsets is less computationally expensive than mergesort's generalized merge operation. On the other hand, if the set is pathological, e.g. it's already sorted, Quicksort will degenerate into a O(N*N) behavior while mergesort will always be O(N*log(N)).

Here are some links to some animations of the Quicksort algorithm (all imperative/procedural code):

- http://www.animatedrecursion.com/intermediate/quicksort.html
- http://www.cs.auckland.ac.nz/software/AlgAnim/qsort.html
- http://pages.stern.nyu.edu/~panos/java/Quicksort/index.html

Here's a very basic implementation of Quicksort in an imperative coding style:

```
import java.util.*;
class QuicksortBasic implements QuickSort {
  public ArrayList<Integer> sort(ArrayList<Integer> a) {

    if (a.isEmpty()) return new ArrayList<Integer>();

    ArrayList<Integer> left = new ArrayList<Integer>();
    ArrayList<Integer> mid = new ArrayList<Integer>();
    ArrayList<Integer> right = new ArrayList<Integer>();

    for (Integer i : a)
      if ( i < a.get(0) )
        left.add(i); // Use element 0 as pivot
      else if ( i > a.get(0))
        right.add(i);
      else
        mid.add(i);

    ArrayList<Integer> left_s = sort(left);
    ArrayList<Integer> right_s = sort(right);

    left_s.addAll(mid);
    left_s.addAll(right_s);

    return left_s;
  }
}
```

It is possible to optimize this code further to make it even faster.

From an OO sorting perspective, Quicksort is a hard-split, easy join example of Merritt's sorting theorem.

```
/**
 * A concrete sorter that uses the QuickSort method.
 */
public class QuickSorter extends ASorter
{

 /**
  * The constructor for this class.
  * @param iCompareOp The comparison strategy to use in the sorting.
  */
 public QuickSorter(AOrder iCompareOp)
 {
  super(iCompareOp);
 }
 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value of this function.
  * This method places all values greater than the key at A[0] at indices above the split index
  * and all values below the key at indices less than the split index.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  * @return The split index.
  */
 protected int split(Object[] A, int lo, int hi)
 {
      Object key = A[lo];
      int lx = lo; // left index.
      int rx = hi; // right index.
      // Invariant 1: key <= A[rx+1:hi].
      // Invariant 2: A[lo:lx-1] <= key.
      // Invariant 3: there exists ix in [lo:rx] such that A[ix] <= key.
      // Invariant 4: there exists jx in [lx:hi] such that key <= A[jx].
      while (lx <= rx)
      {
```

```
         while (aOrder.lt(key, A[rx])) // will terminate due to invariant 3.
         {
            rx--;  // Invariant 1 is maintained.
         } // A[rx] <= key <= A[rx+1:hi]; also  invariant 0, lx <= rx.

         while (aOrder.lt(A[lx], key)) // will terminate due to invariant 4.
         {
            lx++;  // Invariant 2 is maintained.
         } // A[lo:lx-1] <= key <= A[lx]

         if (lx <= rx)
         {
            // swap A[lx] with A[rx]:
            Object temp = A[lx];
            A[lx] = A[rx]; // invariant 3 is maintained.
            A[rx] = temp;  // invariant 4 is maintained.
            rx--;  // invariant 1 is maintained.
            lx++;  // invariant 2 is maintained.
         }
      } // rx < lx, A[lo:lx-1] <= key <= A[rx+1:hi], and key = A[lx].
  return lx;

 }

 /**
  * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].
  * This method does nothing, as the sub-arrays are already in proper order.
  * @param A A[lo:s-1] and A[s:hi] are sorted.
  * @param lo the low index of A.
  * @param s
  * @param hi the high index of A.
  */
 protected void join(Object[] A, int lo, int s, int hi)
 {
 }
}
```

The split operations is just the partitioning of the array into two parts, one smaller than the pivot and one larger than the pivot. The split operation corresponds to the "partition" function that many Quicksort implementations identify. The join operation is simply a no-op because the two sorted subsets are disjoint and contiguous, so they are already in order.

Also download the lecture notes on this subject that includes an interesting decomposition of the Quicksort problem into an architecture that can be efficiently run in parallel on multi-core processors.

==========

Variable Argument Lists

We discussed this before in lab and lecture, but it's worth revisiting to make sure everyone understands it.

A parameter defined as "P... params" is a vararg input parameter of type P. A vararg input parameter allows you to call the method with zero, one, two, or however many values as you want.

For instance, suppose I have a method define as such:

```
// returns the sum of all the input values supplied
int sum(int...params) {
   int result = 0;
   for(int i=0; i< params.length; i++) {
       result += params[i];
   }
   return result;
}
```

(Note: there are more efficient ways of writing this method, but I wanted to show you several features at once)

A vararg parameter comes in as an array, so one can figure out how many parameters were supplied (params.length) and one can access each supplied value using regular array syntax, e.g. params[i].

The following are all valid calls to the above method:

- `sum()` --> returns zero.
- `sum(42)` --> returns 0+42 = 42.
- `sum(10, 20)` --> returns 0+10+20 = 30.
- `sum(100, 1000, 10000)` --> returns 11100 etc

Varargs and Visitors

Vararg input parameters are very useful when you might want to call a method, such as the `accept` method of a host (and hence a case on the visitor), with varying numbers of input parameters, or perhaps, none at all.

If you don't need to use the input parameter, I suggest using the "`Void`" object type and calling your parameter something like "`nu`" (for "not used"):

```
Void... nu
```

When you call the visitor, simply don't supply an input parameter.

If you need an input parameter, just set the `P` (the generic type of the input parameter) to the type you need and call the `accept` method with as many values as you need. Note that the one restriction of a vararg input parameter is that all the input values have to be of the same type.

The visitor system that HW11 uses allows you to give parameters to the visitor at the time that you use the visitor rather than having to imbed additional parameters inside the visitor as was done in many examples in class. This means that the same visitor instance can be used in multiple situations with different input parameters. To note however, when using anonymous inner classes, input parameter values are often not needed because the anonymous inner class already has access, via its closure, to most of the values it needs. Input parameters are particularly useful for things such as accumulators however, which cannot be curried in.

==========

A Tale of Two Visitors

You were shown two different styles of writing visitors, with and without input parameters. Why two different techniques?

The main reason: because different people view the world in different manners and everyone's viewpoint has their strengths and weaknesses.

Here's a more technical breakdown:

Dr. Cartwright's No-input Parameter Visitor Style

Any necessary parameters for a visitor are handed to the constructor of the visitor and are saved in field(s) of the visitor to be used as the visitor executes.

Pros:

- Parameters of multiple types are easily handled.
- The host's `accept` method is simplified and maintains exactly the same call structure for any visitor.
- When using anonymous inner class implementations of visitors, most parameters are curried in via the anonymous inner class's closure, so parameter storage fields are often not necessary.

Cons:

- If the parameter value changes, e.g. for accululator algorithms, either one has to instantiate an entirely new instance of the visitor or take the risk of mutating the parameter storage field and hope that no one else is using the visitor.
- Anonymous inner class implementations treat parameters that are invariant over the lifetime of the visitor equivalently to those that are varying from call to call.
- The design does not cleanly separate the variant nature of the parameters from the invariant nature of the processing algorithm that the vistior represents.

Dr. Wong's Vararg Input Parameter Visitor Style

Parameters for a visitor are handed to the vistior at execution time (when accepted by the host) via a variable input argument list (vararg).

Pros:

- The variant input parameters are cleanly separated from the invariant processing algorithm the vistior represents. This enables many more visitors to be written as singletons.
- Input parameters with varying values, e.g. accumulator algorithms are easily handled without instantiating a new visitor instance or worrying about multiple usages of the visitor.
- The same visitor instance can be used simultaneously, e.g. in parallel taskings and in branching recursive algorithms.
- Varargs allows varied numbers of parameter values to be handed to the same vistior, including none at all.
- Using anonymous inner class implementations allow the separation of parameters that are invariant over the lifetime of the visitor (curried-in values) vs. parameters that vary from call to call (input parameters).

Cons:

- Parameters of different types are difficult to handle.
- Since the host's `accept` method allows any number of input parameters to be provided, it is impossible to tell if the correct number of input parameters has been passed.
- No parameter visitors are still required to declare an input parameter type (e.g. "`Void...nu`").

Who's "right"? That depends on where you stand and what's important to you. Perhaps neither. You choose what's right for what you're trying to do at that moment. There is no gospel, only defensible but non-absolute positions.

==========

Binary Search in a Sorted Array

The recursive technique used to find a value in a sorted array is very useful in many situations.

Consider the following sorted array of values:

`[1, 2, 4, 7, 10, 11, 12, 13, 20, 25, 27, 28, 30]`

Supposed you're give a value, `x`, what's the fastest way to either find that value in the array or determine if it is not present?

The trick is to use the same technique many people use to find a name in a phonebook, a technique called a "binary search" because it is continually making the searched space smaller by a factor of two:

1. Open the book in the middle `mid = (max+min)/2` (min, max and mid are the page numbers here)
2. If the name you want is on that page, you're done.
3. If the name is before the names where you opened the book, re-open the book halfway on the smaller side and repeat the process.
4. If the name is after the names where you opened the book, re-open the book halfway on the larger side and repeat the process.

For the above array, we would look for the number 11 by doing the following:

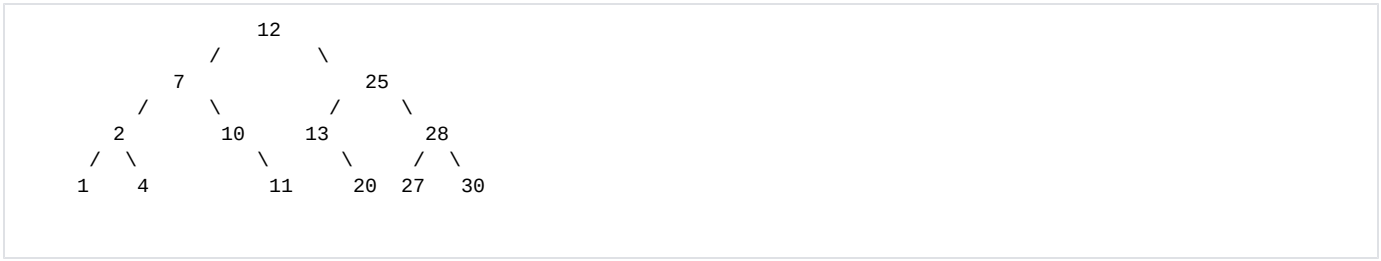
1. min = 0, max = 13, so mid = 6 ==> so look at the #12 (min, max and mid are indices into the array here)
2. min= 0, max = 6, and 11 < 12, so look at mid = (6+0)/2 = 3, i.e. the #7.
3. min= 3, max = 6, and 11 > 7, so look at the mid = (6+3)/2 = 4, i.e. the #10
4. min = 4, max = 6 and 11>10 so look at the mid=(4+6)/2 = 5, i.e. the #11
5. min = 5, max = 6, there's only one element left and since 11 = 11, we found it!

Here's some Java code to do the same:

```
/**
 * Uses binary search to find x in the given array within the given min (inclusive) and max (exclusive) bounds.
 * returns true if found, false otherwise.
 */
boolean find(int x, int[] anArray, int min, int max) {
  if(0 >= (max-min) {
    return false;   // safety catch, should never hit this.
  }
  else if(1 == (man-min)) {   // only one element to look at
    return anArray[min] == x;   // return true is equal, false otherwise
  }
  else {   // still looking
    int mid = (max+min)/2;  // integer arithmetic will round down.
    if(anArray[mid] <= x) {  // careful here, need to make sure midpoint is included if x = anArray[mid]
       return find(x, anArray, mid, max);  // recur with smaller bounds = upper half of the original bounds.
    }
    else {
       return find(x, anArray, min, mid);  // recur with smaller bounds = lower half of the original bounds
    }
  }
}
```

Such as search is very fast, running in O(log N) time because you are reducing the size of the space by a constant multiplicative factor every time your recurse.

One way to look at what is going on here is to think of the sorted array as already being in the form of a binary search tree, where each mid-point is the top node of a sub-tree. All the above algorithm is doing is walking down from the top of the binary search tree, i.e the path 12 => 7 => 10 => 11.



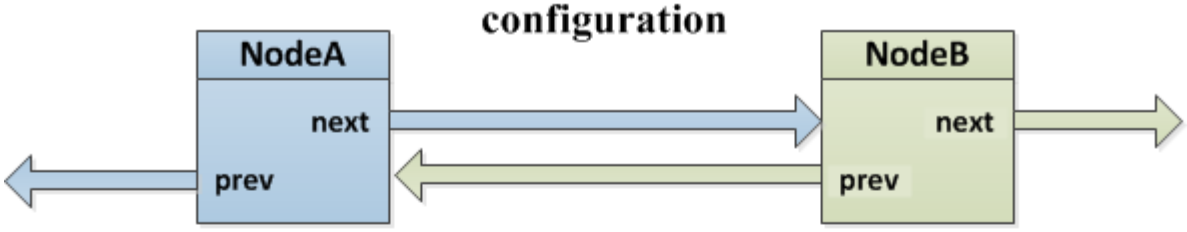
It is a very useful notion to conceptually view an array as a tree. This enables one to leverage the power of recursive data structures with the speed and compactness of an array.

==========

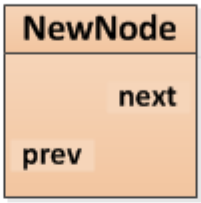
Insertion into a Doubly-linked List

Inserting a node into a doubly-linked list, ala `BiList`, isn't difficult, but it does take care to make sure that one has the right values for the node references at any given moment. Here's a pictoral explanation of the process:

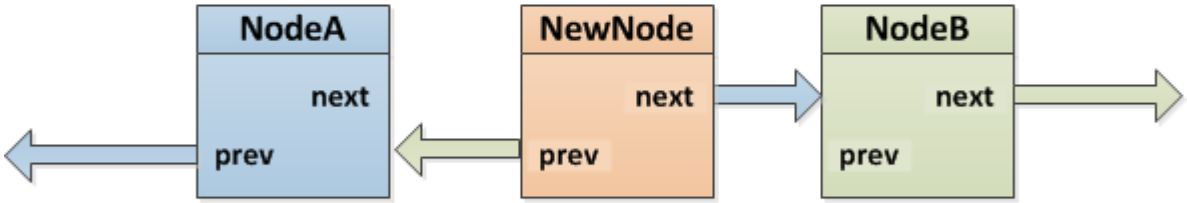
Original configuration



Insert NewNode between NodeA and NodeB



Copy next from NodeA and prev from NodeB and put them into NewNode



Set next in NodeA and prev in NodeB to reference NewNode