CS61B Summer 2006
Exam 2
31 July 2006

Your name:

Your login: cs61b-_______

This exam is worth 40 points (out of a possible 230 for your course grade).

This exam is open book, open notes - you may use any references on paper that you wish, but no electronic devices (e.g. laptop computers) are permitted.

This booklet contains XX numbered pages (including this page). Please make sure that you have all of the pages before you begin. You will have 80 minutes to complete the exam.

Please turn off your cell phone now.

1 Asymtotic Analysis (5 points)

Circle "true" or "false" for each of the following statements.

a. $15n^3 + 10^{99}n^2 + 5n \log n$ is in $\Theta(n^3)$.

TRUE - The dominating term is the n^3 term.

b. $2^n + 7n^2 + 9n$ is in $O(n^{99})$.

FALSE - 2^n is too big to be dominated by n^{99}

c. The running time of insertion sort on a list of length n is in $\Omega(n)$.

TRUE - Remember that Ω is a lower bound, so anything that dominates n is in $\Omega(n)$

d. The memory use of quicksort on an array of size n is in $O(n^2)$.

TRUE - Remember that O gives an *upper* bound, so anything dominated by n^2 is in $O(n^2)$.

e. The time of `find` on an AVL tree with n nodes is in $\Theta(n)$

FALSE - it's $\Theta(log(n))$

2 Choosing a sorting algorithm (4 points)

For each situation described below, say which sorting algorithm *from this list* (insertion sort, selection sort, mergesort, quicksort, bucket sort) is the best to use. You will not use every possible answer and some answers may be used more than once.

a. A linked list of several thousand integers between 0 and 100.

Ans: Bucket sort, since there are many more items than possible keys, and we are starting with a linked list, so we will not need to create nodes for each element to make the lists in each bucket - we already have them.

b. A linked list of several thousand integers, when there is a 99% chance that the list is already sorted.

Ans: Insertion sort, since it runs in linear time on an already sorted list, and none of the other choices do (except bucket sort, but you don't know enough about the distribution of the keys to make that a good idea).

c. An array of several thousand integers between 0 and 1,000,000.

Ans: Quicksort - it's the fastest and most memory efficient to use on arrays.

d. An array of several thousand integers between 0 and 100, *using only a constant amount of additional memory.*

Ans: Quicksort - bucket sort might be faster, since there are far more items to sort than possible keys, but bucket sort on an array requires making a list node for each item being sorted (so they can be put into the linked lists at each bucket), so it uses an amount of additional memory proportional to (n + q), where n is the number of items being stored and q is the number of possible keys (101 in this case). Had the input been a linked list instead, bucket sort would have been an acceptable answer, since that would only use q additional memory.

3 Array Duplicates (10 points)

Given an array A of objects of type `O`, we would like to decide if A has any duplicate elements.

a. Suppose that object of type `O` do not have a total ordering - so we can test them for equality (using `.equals`, but cannot say if one is less than or greater than another. Give the most efficient algorithm you can for determining if A has any duplicates and write Java code for it. The only method you can call on `O` objects is `.equals`.

```
public boolean hasDuplicates(O[] A) {
   for(int i = 0; i < A.length; i++) {
     for(int j = i+1; j < A.length; j++) {
       if (A[i].equals(A[j]) {
         return true;
       }
     }
   }
   return false;
}
```

We loop through the array, comparing each pair of objects exactly once, returning true as soon as we find a duplicate. Note that you have to be careful about comparing an object to itself - if we had started both i and j at 0, then we would return true on the first call every time, since an object is always .equals to itself.

b. What is the running time of your algorithm? Give a Θ bound.

Ans: $\Theta(n^2)$ - we can see this from our double-nested for loop.

c. Suppose now that `O` implements the `Comparable` interface - so we can use the `compareTo` method to determine if one is less than, equal to, or greater than another. Is it possible to solve the problem faster than the algorithm you gave in part (a)? If yes, describe the algorithm in *one short sentence* and state its running time. If no, *briefly* explain why not.

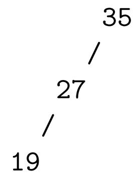
Ans: If O implements `compareTo`, then we can sort the array of O objects using one of the (n log n) sorting algorithms we know (e.g. quicksort), and then search for duplicates by just walking down the sorted array once. The running time of this is $\Theta(n \log n + n)$, which is $\Theta(n \log n)$.

4 AVL Trees (6 points)

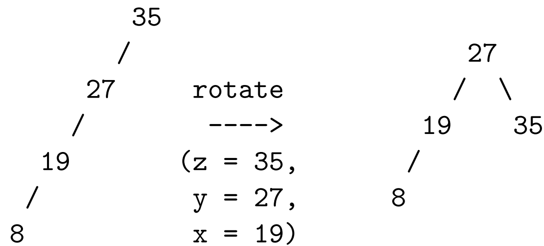
Construct an AVL tree by inserting nodes as indicated, starting with the given "start" tree. Draw the tree after each insertion and label the nodes clearly. (If you need to draw anything extra while you are figuring out your solution for an item, please draw a box around the tree that you want graded as your final answer.)

Note: The height of the empty tree is 0.

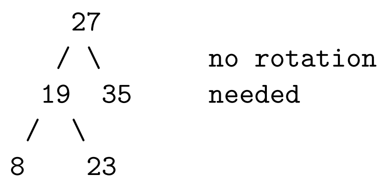
Start with this tree:



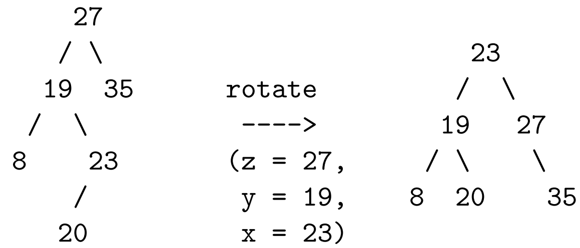
- insert 8



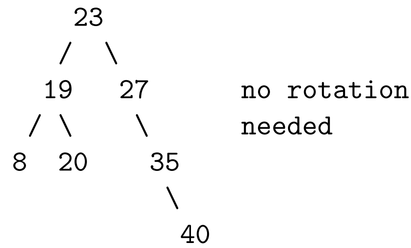
- insert 23



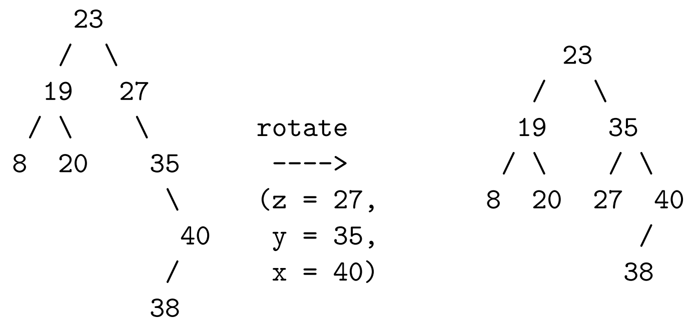
- insert 20



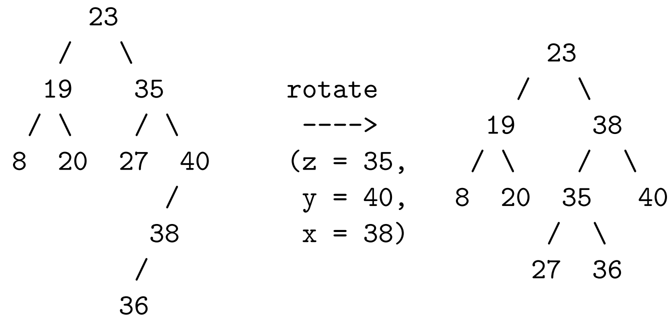
- insert 40



- insert 38



- insert 36



The height of the empty tree being 0 seemed to cause difficulties for a lot of people. Some textbooks/implementations choose 0, others choose -1... it's an arbitrary convention and each has its own advantages and disadvantages. But since it was clearly stated (in bold!) at the begining of the question to use 0, that's what had to be done.

Some people seemed to forget that an AVL tree is a kind of binary search tree, and didn't respect the BST invariant when adding items. The tree is essentially useless as a search tree if it doesn't maintain the search invariant!

5 Search Trees (8 points)

What node is at the root of each of the following trees after running (a) `insert(17)` or (b) `remove(36)`? You don't have to draw the whole tree after the operation, *just give the root node.* Do each operation on the tree as it is given to you, separately, not one after the other (so the 2 parts are independent).

Ans:

- For the basic BST, inserting and removing never change the root, so the answer is just the "12" node for both.
- For the splay tree, when we insert a new node, we splay it to the root, so the answer for the insert(17) part is the "17" node. When we try to remove a node, if it's not in the tree, we splay the node where the search ended to the root, so the answer for the remove(36) part is the "26" node.
- For this 2-3-4 tree, insert finds a 3-key node on its way down, so it pushes that nodes middle key up to its parent, so the answer for the insert(17) part is the 2-key node `|22|27|`. Remove does not encounter any 1-key nodes except for the root on its way to removing an item from a leaf, so the root doens't change - the answer for the remove(36) part is the 1-key node `|27|`.
- For this 2-3-4 tree, insert does not find any 3-key nodes on its way down, so the root does not change - the answer for the insert(17) part is the 1-key node `|40|`. Remove encounters a 1-key node on its way down, so it tries to steal a key from its sibling. The sibling also has only 1 key, and also, their parent is the root and has only 1 key. So these 3 1-key nodes are merged to become the new root, so the answer for the remove(36) part is the 3-key node `|37|40|44|`.

6 Tree traversal (7 points)

Traversing a tree "level order" means that we visit all the nodes at depth 0, then at depth 1, then at depth 2, etc., going from left to right at each depth. For example, supposing that each node contained a character, and visiting a node printed that character, if T is the tree below, then inorder(T) prints ILOVETREES.

```
          I
        / | \
       L  O  V
       |  |  | \
       E  T  R  E
      /|
     E S
```

Assume that lists implement the following interfaces:

```
public interface List {

     //Returns the head of the list, which will be null if the list is empty.
     public ListNode getHead();
}

public interface ListNode {

     //Returns this ListNode's piece of data, which is a TreeNode.
     public TreeNode getItem();

     //Returns the next node in the list (which is null if we are at the end)
     public ListNode getNext();
}
```

Recall some of the simpler data structures that we looked at, namely stacks, queues, and priority queues. You may want to use one or more of these in your solution; assume that you have implementations of all of them available that implement the standard interfaces described in class.

Write the `levelorder()` method to traverse a tree in level order. Some other parts of the TreeNode class that you may need are provided for you.

To recieve full credit, your algorithm must run in $\Theta(n)$ time, where n is the number of nodes in the tree.

```
public class TreeNode  {

   //Visits the node
   public void visit() {
       //visit code
   }

   //Returns a list of the children of this TreeNode.
   public List getChildren() {
       //getchildren code
   }

   public void levelorder() {

       Queue q = new Queue();
       q.enqueue(this);

       while(!q.isEmpty()) {

           TreeNode n = (TreeNode) q.dequeue();
           n.visit();

           TreeNode c = n.getChildren().getHead();
           while (c != null) {
               q.enqueue(c);
               c = c.getNext();
           }
       }
   }
```