CS61B Fall 2015 Guerrilla Section 2 Worksheet Solutions

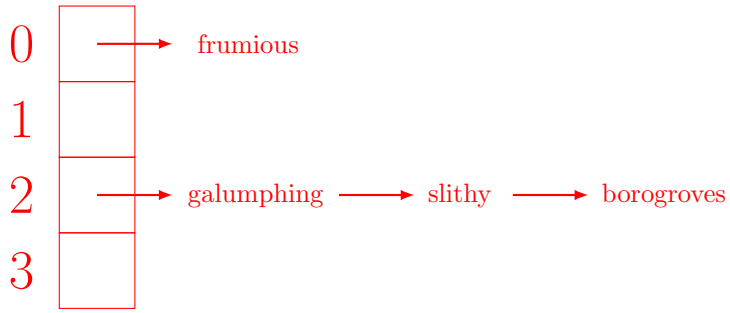
31 October 2015

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is*. Of course, a topic appearing on this worksheet does not imply that the topic will appear on the midterm, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

1 Hashing Mechanics

Insert the following words into a hash table (in the same order that they are listed): galumphing, frumious, slithy, borogroves, mome, bandersnatch. Assume that the hash code of a `String` is just its length (note that this is not actually the hash code for `String`s in Java). Use external chaining to resolve collisions, use 4 as the initial size of the array and double the size when the load factor is equal to 1.

After the first 4 insertions, the hash table looks like this (note that the order of the items in each linked list is not important and depends on the implementation):



Before the next insertion, the array is resized. After all insertions are completed, the hash table looks like this (shown on next page):

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

2 More Hashing

Describe a potential problem with each of the following:

(a) An implementation of the `hashCode` method of the `String` class that simply returns the length of the string (i.e. the hash code used in problem 1).

This will likely cause many collisions since many distinct strings have the same length. Since the efficiency of hash tables depends on a low number of collisions, this would result in inserting, removing, and finding objects in the hashtable to be slow.

(b) An implementation of the `hashCode` method of the `String` class that simply returns a random number each time.

Hash functions must be deterministic, so this hashCode is not even valid. If random numbers are used, then hashCode may return different values for the same object when it is called repeatedly.

(c) Overriding the `equals` method of a class without overriding the `hashCode` method.

If equals is overriden and hashCode is not, then it is possible that two objects will be equal (according to the equals method) but have different hashCodes. This may cause a HashSet to report that an object is not present even if some identical object is in the HashSet.

(d) Overriding the `hashCode` method of a class without overriding the `equals` method.

This causes the same potential problem as part (c).

(e) Modifying an object after inserting it into a `HashSet`.

This may prevent us from being able to find the object again since the HashSet uses the object's hashCode to check if the object is present (and if the object is modified, its hashCode may also change).

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

3 Using Hash Tables

Given an array `A` of integers and an integer `x` we want to find out if `A` contains two integers `y` and `z` such that $y + z = x$. Write a function that accomplishes this efficiently in most cases (the worst case running time is allowed to be slow). For a hint, look at the title of this question.

```
public static boolean twoSum(int[] A, int x) {
    HashSet<Integer> B = new HashSet<Integer>();
    for (int y : A) {
        B.add(x);
    }
    for (int z : A) {
        if (B.contains(x - z)) {
            return true;
        }
    }
    return false;
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

4 Heap Mechanics

Fill in the following class to implement the insert method for a min heap. Assume that the heap is implemented with an array where the root is stored at index 1 (rather than index 0). You do not need to handle resizing the array.

```
public class MinHeap {

    /** Initializes an empty min heap. */
    public MinHeap() {
        //Implementation not shown.
    }

    /** Removes and returns the minimum element stored in this heap. */
    public int removeMin() {
        //Implementation not shown.
    }

    /** Returns the minimum integer stored in this heap. */
    public int findMin() {
        return data[1];
    }

    /** Inserts the integer X into this heap. */
    public void insert(int x) {
        size += 1;
        data[size] = x;
        int i = size;
        while (i > 1 && data[i] < data[i/2]) {
            swap(i, i/2);
            i = i/2;
        }
    }

    //You may add other instance methods here.

    /** Swaps the elements of data at positions I and J. */
    private void swap(int i, int j) {
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }

    /** The data used to represent this heap. */
    private int[] data;
    /** The number of elements currently being stored in this heap. */
    private int size;
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

5 Lowest Common Ancestor (CS61B Fall 2013 Midterm 2)

The lowest common ancestor of two nodes in a tree is the deepest node in the tree (furthest from the root) that has both of the nodes as descendants. It can be one of the two nodes (if one is an ancestor of the other). Write a function that takes the root of a binary search tree along with two integers that are in that tree and returns the lowest common ancestor of those two integers.

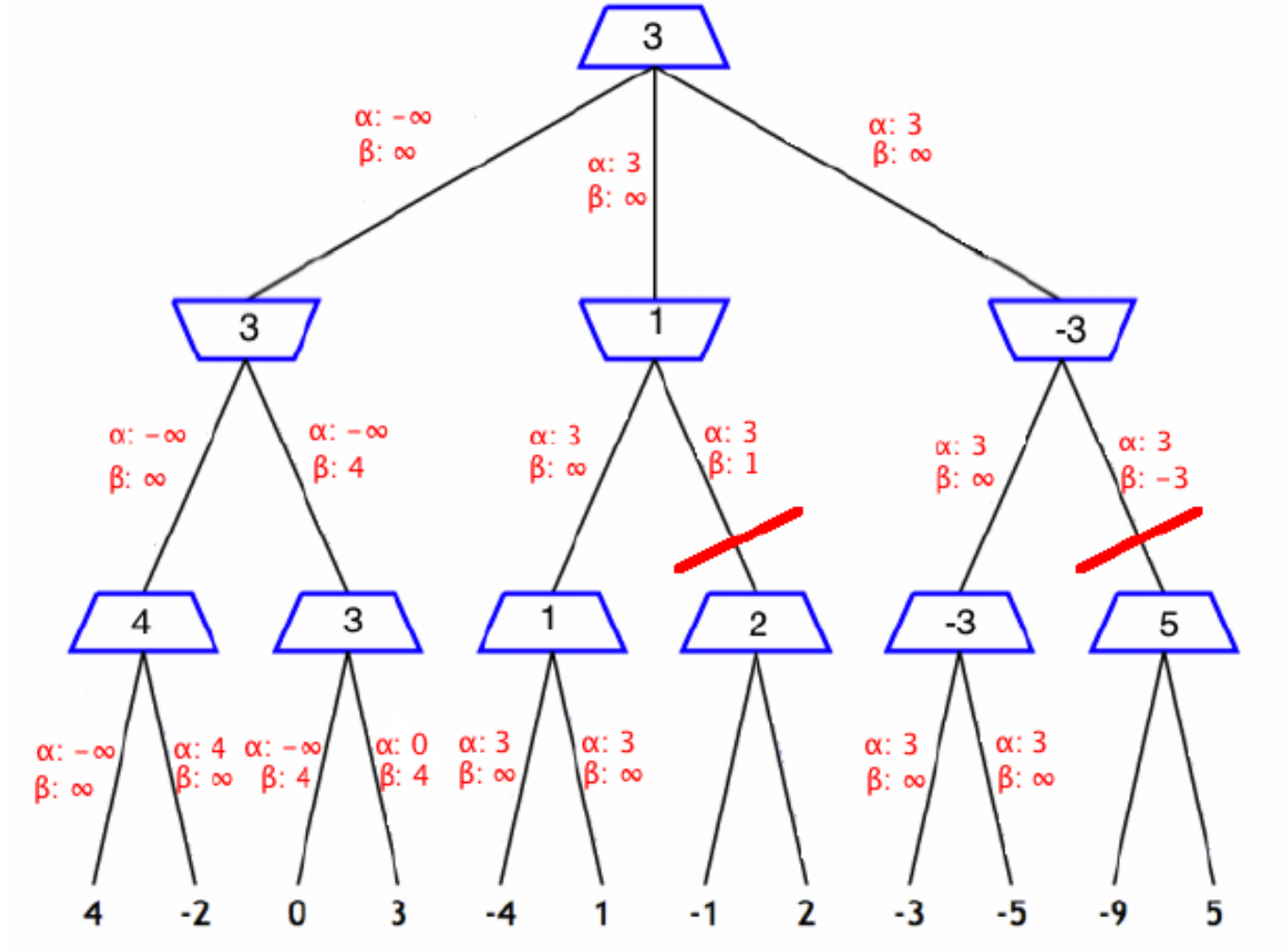
```
1  public class TreeNode {
2      public int val;
3      public TreeNode left;
4      public TreeNode right;
5
6      /** Assuming that T is a binary search tree and that KEY1 and
7       *  KEY2 both appear in T, returns the value of the lowest common
8       *  ancestor of the nodes containing KEY1 and KEY2. */
9      public static int findLCA(TreeNode T, int key1, int key2) {
10         if (key1 > key2) {
11             return findLCA(T, key2, key1);
12         } else if (T.val > key2) {
13             return findLCA(T.left, key1, key2);
14         } else if (T.val < key1) {
15             return findLCA(T.right, key1, key2);
16         } else {
17             return T.val;
18         }
19     }
20 }
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

6 Game Trees

Consider the game tree shown below. Trapezoids that 'point' up, represent the player seeking to maximize the heuristic evaluation (this is you); trapezoids that 'point' down represent the player seeking to minimize the heuristic evaluation (your opponent).



(a) Fill out the values in the 'maximizer' and 'minimizer' node for the above game tree after applying the minimax algorithm to it.

(b) Cross out (with an X) all branches that would be pruned by a Minimax implementation that utilizes the pruning strategy discussed in class at the end of lecture #22 (alpha-beta pruning).

(c) According to the minimax algorithm, which move should we make for the above game? Say, your opponent was a 3-year old, would you still use the minimax algorithm? According to the minimax algorithm we should make the move corresponding to the left-most child of the root. If we were playing a 3-year-old however, we would not necessarily want to avoid the minimax algorithm since the minimax algorithm simply finds the move that has the best guaranteed outcome, rather than the move with the highest possible reward. Since 3-year-olds are not masters of strategy, we might want to opt for a potentially riskier move that has a chance of a higher reward if our opponent makes a suboptimal move.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

7 Generics and Binary Trees

(a) Suppose that we have a binary tree where the nodes contain objects that can be compared to each other. Then we could define an order on such trees in the following way: a tree T_1 is considered greater than a tree T_2 if one of the following holds:

- T_2 is empty and T_1 is not.
- The value in the root of T_1 is greater than the value in the root of T_2.
- The values in the roots are equal and the left subtree of T_1 is greater than the right subtree of T_2.
- The values in the roots are equal and the left subtrees are equal and the right subtree of T_1 is greater than the right subtree of T_2.

Fill in the class `ComparableTree` below so that the `compareTo` method will give the ordering described above. You may assume that the values stored in the tree are never null.

```
1  public class ComparableTree<T extends Comparable<T>>
2      implements Comparable<ComparableTree<T>> {
3      public T val;
4      public ComparableTree<T> left;
5      public ComparableTree<T> right;
6
7      @Override
8      public int compareTo(ComparableTree<T> t2) {
9          return compareTrees(this, t2);
10     }
11
12     private static int compareTrees(ComparableTree<T> t1,
13                                     ComparableTree<T> t2) {
14         if (t1 == null) {
15             if (t2 == null) {
16                 return 0;
17             } else {
18                 return -1;
19             }
20         }
21         if (t2 == null) {
22             return 1;
23         }
24         int compareVal = t1.val - t2.val;
25         if (compareVal != 0) {
26             return compareVal;
27         }
28         int compareLeft = compareTrees(t1.left, t2.left);
29         if (compareLeft != 0) {
30             return compareLeft;
31         }
32         return compareTrees(t1.right, t2.right);
33     }
34 }
```

(b) What type of tree traversal does the above ordering correspond to?

A preorder traversal since we first compare the values at the root.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

8 Assorted Heap Questions

(a) Describe a way to modify the usual max heap implementation so that finding the minimum element takes constant time without incurring more than a constant amount of additional time and space for the other operations.

Simply add a variable that keeps track of the minimum value in the heap. When inserting a new value, simply update this variable if the new value is smaller than it. Since the max heap only supports removing the largest element, rather than arbitrary elements, the minimum element will only be removed when the heap becomes empty, at which point we will need to reset the variable keeping track of the minimum value.

(b) (Fall 2014 Final exam) In class, we looked at one way of implementing a priority queue: the binary heap. Recall that a binary heap is a nearly complete binary tree such that any node is smaller than all of its children. There is a natural generalization of this idea called a d-ary heap. This is also a nearly complete tree where every node is smaller than all of its children. But instead of every node having two children, every node has d children for some fixed constant d.

1. Describe how to insert a new element into a d-ary heap (this should be very similar to the binary heap case). What is the running time in terms of d and n (the number of elements)?

   To insert, we add the new element on the last level of the tree and then bubble it up, much as in a binary heap. When bubbling up, we only need to compare the node to its parent. Since the tree has $\Theta(\log_d(n))$ levels and we have to do at most one comparison (compare the node to its parent) and one swap at each level, insertion takes $\Theta(\log_d(n))$ time.

2. What is the running time of finding the minimum element in a d-ary heap with n nodes in terms of d and n?

   The minimum element is simply the root, just as with a binary min-heap. So finding it takes $\Theta(1)$ time.

3. Describe how to remove the minimum element from a d-ary heap (this should be very similar to the binary heap case). What is the running time in terms of d and n?

   To remove the minimum element, we first replace it with the last element on the last level of the heap and then bubble this element dowm, just as in a binary heap. When bubbling a node down, we have to compare the node to all of its children to determine if it is larger than any of them and to find the smallest one (since we must swap with the smallest child to preserve the heap property). So at each level we have to do at most $\Theta(d)$ comparisons and 1 swap. Since there are $\Theta(\log_d(n))$ levels, removing the minimum takes $\Theta(d \log_d(n))$ time.