CS106B Handout 31
Summer 2011 July 18th, 2011

Final Exam

This is an open-note, open-reader exam. You can refer to any course handouts, textbooks, handwritten lecture notes, and printouts of any code relevant to any CS106B assignment. You may not use any laptops, cell phones, or handheld devices of any sort. You will be graded on functionality—but good style helps graders understand what you were attempting. You do not need to include any libraries and you do not need to forward declare any functions. You have 3 hours. We hope this exam is an exciting journey ☺.

Last Name: _____

First Name: _____

Section Leader: _____

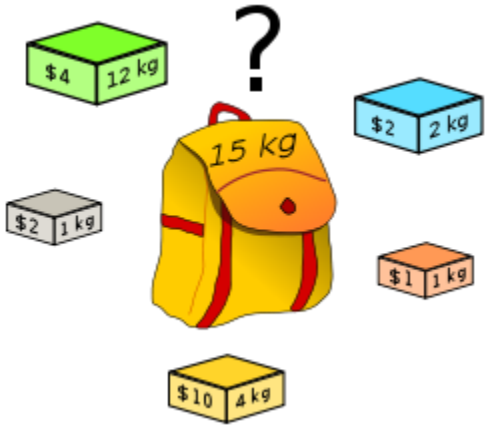
I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

(signed) _____

		Score	Grader
1. Knapsack	[20]	_____	_____
2. DB Trees	[25]	_____	_____
3. Good Will Hunting	[25]	_____	_____
4. Jasmine Revolution	[20]	_____	_____
5. Short Answers	[15]	_____	_____
Total	**[105]**	**_____**	

Problem 1: Knapsack (20 points, Medium Difficulty)

One famous problem in theoretical computer science is the knapsack problem. Given a targest weight and a set of objects in which each object has a *value* and a *weight*, determine a subset of objects such that the sum of their weights is less than or equal to the target weight and the sum of their values is maximized.



(Image courteousy of Wikipedia)

For this problem we will represent an object with the following struct:

```
struct objectT {
   int weight;  //You may assume this is greater than or equal to 0
   int value;   //You may assume this is greater than or equal to 0
};
```

Write the function **`int FillKnapsack(Vector<objectT> objects, int targetWeight)`**, that considers all possible combinations of **`objectT`** from objects (such that the sum of their weights is less than or equal to **`targetWeight`**) and returns the maximum possible sum of object values.

```
int FillKnapsack(Vector<objectT> objects, int targetWeight) {

}
```

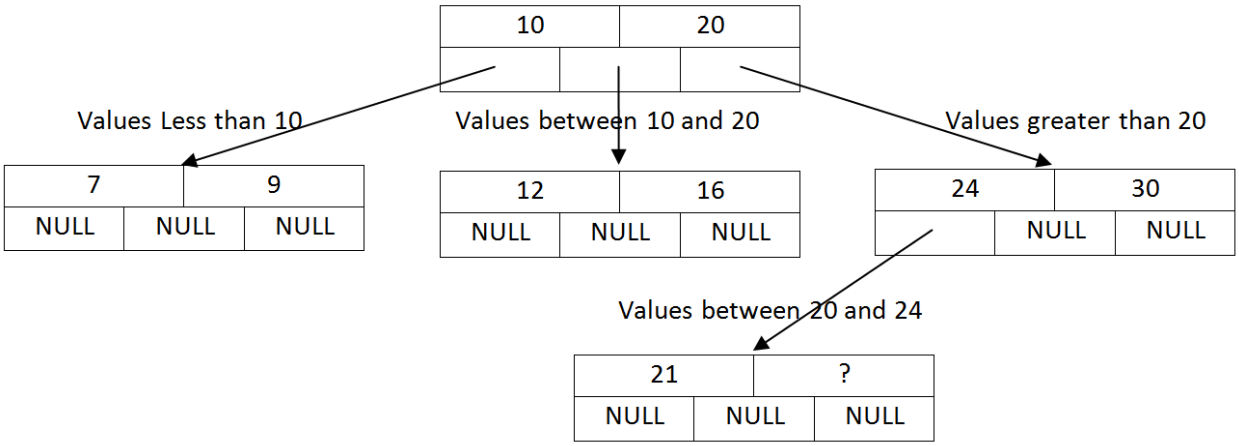
Problem 2: Database Trees (30 points, Very Hard)

In class we learned about Binary Search Trees, a data structure which, when balanced, enable O(log(n)) lookup, insert and removal. While O(log(n)) is ridiculously fast asymptotically, there are situations where it is beneficial to put more information into each node in order to reduce the overall number of nodes that need to be loaded into memory. One common situation where this occurs is with computer file systems. It takes a *long* time to access data from a hard drive relative to the time it takes to access data from memory, so if each node of the tree is stored on a hard drive then we want to keep the number of loads we need to perform as low as possible.

A Database Tree (DB Tree for short) takes the concept of a Binary Search Trees and generalizes it by:

1. Allowing each node to have either 1 or 2 values. If a node has 2 values, then the first value must be less than the second value.
2. Allowing each node to have up to 3 children
3. All values in the left subtree must be less than the first value
4. All values in the right subtree must be greater than the second value
5. All values in the middle subtree must be between the first and second values
6. A node cannot have children unless both its values are populated

Below is a picture of a DB Tree (where "?" indicates the lack of a value):



You are going to write an DB Tree class by implementing the following:

```
class DBTree {
   public:
      DBTree();
      ~DBTree();
      void Insert(int value);
      void Remove(int value);
   private:
      /*
      Helper functions which, the largest or smallest values in the
tree.  You may assume ExtractLargest() is written for you, but you
must implement ExtractSmallest() yourself in part (d).  These
functions will prove to be extremely valuable when implementing
Remove().  For part (c), you may assume that you have working
implementations of these functions.
      */
      int ExtractSmallest();
      int ExtractLargest();

};
```

In the past we have often created private structs to represent the nodes of a tree, but for this problem it makes sense for the DBTree class itself to be the "node" of the tree, so that each node inside of a Database Tree is represented by an instance of the DBTree class.

a) Private implementation definition and constructor

```
private:
```

```
DBTree::DBTree() {
```

```
DBTree::~DBTree() {
```

b) Insert member function

Insertion into a DB-Tree is similar in spirit to insertion into a Binary Search Tree. You follow edges until you find the leaf node in which the value *would be* if the tree contained the value. If the node has fewer than the maximum number of possible values, then you can insert the value into this node. Otherwise, you need to create a new child node which will store the value you are inserting.

```
void DBTree::insert(int value)  {



}
```

c) Remove member function

When removing a value, you first try to find the value in the tree. If the value is found, there are 3 cases to consider:

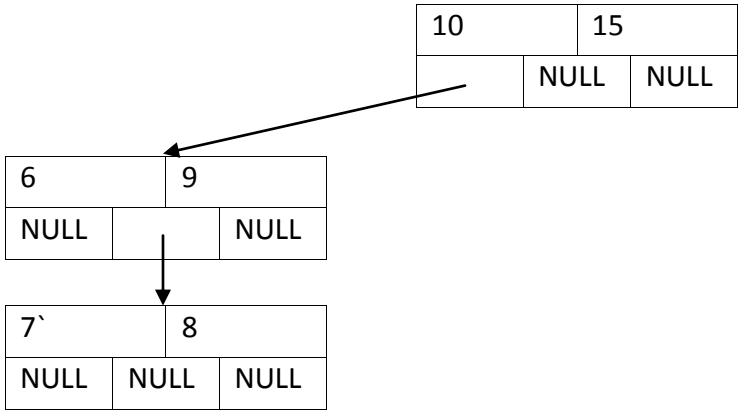
1. If the value is the last remaining value in a leaf node (such as 21 in the example tree), then you remove the node from the tree.
2. If the value is found in a leaf node, but there is another value in the leaf node, then you remove and shift over the remaining values. For instance, if we were removing 7 from the example tree, we would remove it from the node and then shift the 9 over to the first slot.
3. If the value is found in an interior node, then things get a little tricky. When a value is removed from an interior node, it is necessary to "promote" a value from one of its children by searching the tree starting at the interior node, finding a value, removing the value from the tree and then inserting it into the proper location in the interior node. This may require shifting values in the current node. You must promote a value in a manner which maintains all the rules we've declared for DB-Trees. Use `ExtractSmallest()` and `ExtractLargest()` to help you write this. For this question, you may assume they are already given to you, and will properly remove and return the smallest or largest (respectively) value from the tree.

```
void DBTree::remove(int value)  {
```

continued from previous page

d) For `remove()` you assumed you had a working implementations of `ExtranctLargest()` and `ExtranctSmallest()`, but now it is time to write one of them! You only need to write `ExtranctSmallest()`, because the 2 functions are actually quite similar. That said, it is a tricky function. `ExtranctSmallest()` finds the smallest value in a subtree, removes it from the subtree and returns the value.

For `ExtranctSmallest()`, the basic idea is to keep following the leftmost children until you reach a node that doesn't have a left child, then you know that this node's first value will be the smallest value in that subtree. If the node you reach is a leaf node, then things are easy, but what if the node is an interior node? Consider the following picture



By following the leftmost children, from the root (the node with values 10 and 15), you'll eventually find that the smallest value in the tree is 6, but if you just removed the 6, you'll end up with an interior node that only has a single value (which violates rule (6) of our DBTree definition). Keep this issue in mind when working on this problem. This isn't a simple problem to solve, so spend some time planning this one out.

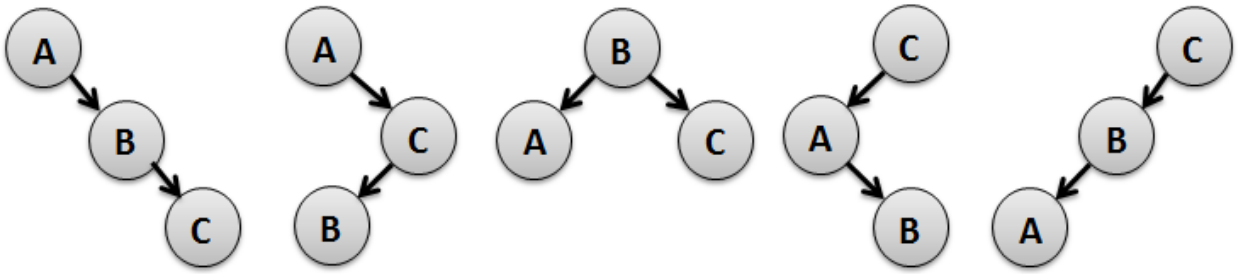
```
int DBTree::ExtractSmallest() {




}
```

Problem 3: Good Will Hunting (25 points, Very Hard)

In the movie Good Will Hunting, Will Hunting (played by Matt Damon) a Janitor at MIT, is a genius. In one scene he amazes Professor Gerald Lambeau when he solves a problem that had been written up in a hallway chalk board. It turns out that problem is actually a very doable graph problem. The problem you are going to solve is almost identical to the one solved by Will in the movie!

Your task is easy to articulate but hard to implement: Write a function that generates all legal Binary Search Trees for a specified collection of values. Each Binary Search Tree must be freshly allocated on the heap.

For example, if you are given the collection of values **{A, B, C}** there are five legal Binary Search Trees that you should create:



In general there are a lot of legal Binary Search Trees that you can make for a given set of values.[1] The collection of values will be represented as a sorted vector where all the values in the vector are unique. You do not need to worry about leaking memory.

(space for the answer to problem 3 appears on the next page)

[1] Interesting: The number of BSTs that can be made from a collection of size *N* is the *N*th Catalan number: ———

Answer to problem 3:

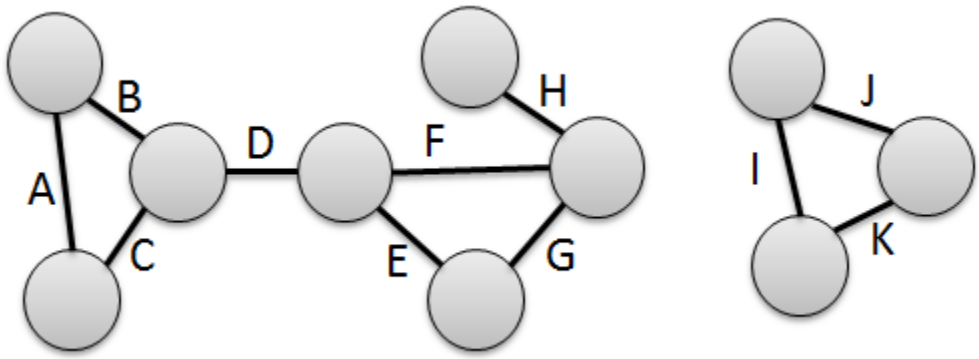
```
struct treeT {
   char value;
   treeT * left;
   treeT * right;
};

Set<treeT *> BuildAllBSTs (Vector<char> & sortedValues) {
```

Problem 4: Jasmine Revolution (20 points, Medium)

On the 28th of January, 2011 the Egyptian Government managed to sever all of Egypt from the internet by cutting a single cable. The internet is structured as an undirected graph where each computer is a node and two computers can communicate as long as there is a single path from one computer to the other. The internet graph has many edges so it was thought that in general it would be unreasonably hard to sever a countries' connection to the outside world. This feet performed by the Egyptian Government was possible because there was a special type of arc called a *bridge* that separated internet in Egypt from the rest of the world.

Your task is to write a function that prints out all bridge's for a given undirected graph. A bridge is defined as an arc with the property that, if that arc did not exist there would no longer be a path in the graph between the two end points of the arc.

For example the follow graph, that has names for arcs not for nodes, has two bridges:



The arcs **D** and **H** are both bridges. Removing the arc **D** will disconnect its two endpoints and similarly removing the arc **H** disconnects the two endpoints that **H** linked. For all other arcs, even if you removed the arc there would still be a path between the endpoints of the arc. For this graph your program should output:

D
H

Each bridge should be output only once. The graph is represented using nodeT and arcT structs which you can assume have been correctly constructed. Your input will be Set containing pointers to all the nodes in the graph and a Set containing pointers to all the arcs in the graph. Your function must not modify the graph. You cannot assume that all nodes in the graph are connected!

(space for the answer to problem 4 appears on the next page)

Answer to problem 4:

```
struct arcT {
   string name;
   nodeT * a;
   nodeT * b
};

struct nodeT {
   Set<arcT *> arcs;
};

void PrintAllBridges(Set<nodeT *> & allNodes, Set<arcT *> & allArcs) {
```

Problem 5: Short Answers (15 points, Medium)

a) Hashing

Consider the following hashing function:

```
int Hash(String str, int numBuckets) {
   int sum = 0;
   for (int i = 0; i < str.length(); i++) {
       sum += str[i]*REALLY_LARGE_NUMBER;
   }
   return sum % numBuckets;
}
```

Argue why this might not be a very good hash function to use if you are hashing English words? You can make your argument either with respect to message authentication or hash maps.

b) Lexicons

What advantages are there to using a Lexicon vs a Set<string> to store the English language?
(Hint: There are 2 big advantages)

c) Graphs

You have a large, densely connected graph. If you wanted to find out if it is possible to get from one node to another in less than *N* hops (moves along an arc), such that *N* is a positive integer, should you use breadth first search or depth first search? Why?