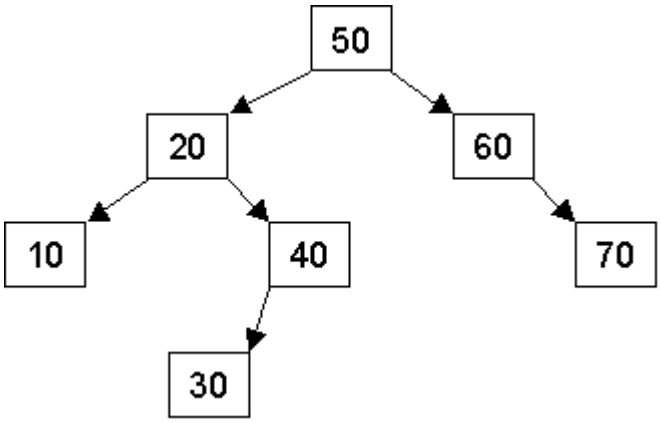
Winter 2020 CS 32

Homework 5

Time due: 11:00 PM Thursday, March 12

This homework is a good study guide for the final. The final will be open book, open notes. We don't expect you to memorize every last detail of every algorithm, so a skill you should develop is the ability to locate an algorithm in a reference source, trace through it, and understand it. If it's not in exactly the form your application requires, you should be able to adapt it.

1. Consider the following binary search tree, ordered using the < relationship:



   a. Using the simplest binary search tree (BST) insertion algorithm (no balancing), show the tree that results after inserting into the above tree the nodes 80, 65, 76, 15, 34 and 25 in that order. (If you're not skilled with a drawing tool, use a simple text form of the tree. For example, the tree depicted above could be shown as

```
                50

        20              60

10          40              70

          30
```

   Use enough space to distinguish left children from right children. Another way to represent the tree in text form (that distinguishes left children from right children) is

```
50
  20
    10
    40
      30
      xx
  60
    xx
    70
```

   b. After inserting the nodes mentioned in part a, what would be printed out by in-order, pre-order, and post-order traversals of the tree (assume your traversal function prints out the number at each node as it is visited)?

c. After inserting the nodes mentioned in part a, what is the resulting BST after you delete the node 30, then the node 20? (Again, just use a simple deletion algorithm with no balancing. If you have an option of making a choice, any correct choice is acceptable.)

2. In some binary search trees, each node has a left child pointer, a right child pointer and a parent pointer. The parent pointer of a node points to its parent (duh!), or is nullptr if the node is the root node. This problem will examine such trees.

   a. Show a C++ structure/class definition for a binary tree node that has both child node pointers and a parent node pointer. Assume the data stored in each node is an int.

   b. Write pseudocode to insert a new node into a binary search tree with parent pointers. (Hint: You can find binary search tree insertion code on pp. 471-473).

3. *Either wait until after Monday's lecture to do this problem, or watch the heaps online lecture.*

   Consider the following operations on an initially empty heap h; this heap is a maxheap, so the biggest item is at the top. The heap is represented as a binary tree:

```
h.insert(3);
h.insert(5);
h.insert(2);
h.insert(0);
h.insert(10);
h.insert(4);
int item;
h.remove(item); // Removes the biggest item from the heap, and puts it in item
h.insert(9);
h.insert(7);
h.remove(item);
```

   a. Show the resulting heap (As in problem 1a, show the tree in some recognizable form.)

   b. Show how your heap from part a would be represented in an array.

   c. After executing `h.remove(item);` one more time, show the array that results.

4. Note: A `pair<T1, T2>` is a simple struct with two data members, one of type T1 and one of type T2. A `set<K>` and a `map<K, V>` are organized as approximately balanced binary search trees; an `unordered_set<K>`and an `unordered_map<K, V>` are organized as hash tables that never allow the load factor to exceed some constant, and a loop that visits every item in a hash table of N items is O(N). For the keys to be hashed, the hash function used produces uniformly distributed results.

   Suppose UCLA has **C** courses each of which has on average **S** students enrolled. For this problem, courses are represented by strings (e.g. "CS 32"), and students by their int UIDs. We will consider a variety of data structures, and for each determine the big-O time complexity of the appropriate way to use that data structure to determine whether a particular student *s* is enrolled in course *c*. For example, if the data structure were `vector<pair<string, vector<int>>>`, where each pair in the outer vector represents a course and all the students in that course, with those students being sorted in order, then if the pairs are in no particular order in the outer vector, the answer would be O(C + log S). (The reason is that we'd have to do a linear search through the outer vector to find the course, which is O(C), and then after that do a binary search of the S students in the sorted vector for that course, which is O(log S).) In these problems, we're just looking for the answer; you don't need to write the reason.

   a. `vector<pair<string, list<int>>>`, where each pair in the outer vector represents a course and all the students in that class, with those students being sorted in order. The pairs are in no particular

order in the outer vector. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?

b. `map<string, list<int>>`, where the students in each list are in no particular order. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?
c. `map<string, set<int>>`. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?
d. `unordered_map<string, set<int>>`. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?
e. `unordered_map<string, unordered_set<int>>`. What is the big-O complexity to determine whether a particular student *s* is enrolled in course *c*?
f. Suppose we have the data structure `map<string, set<int>>` and we wish for a particular course *c* to write the id numbers of *all* the students in that course in sorted order. What is the big-O complexity?
g. Suppose we have the data structure `unordered_map<string, unordered_set<int>>` and we wish for a particular course *c* to write the id numbers of *all* the students in that course in sorted order (perhaps using an additional container to help with that). What is the big-O complexity?
h. Suppose we have the data structure `unordered_map<string, set<int>>` and we wish for a particular student *s* to write *all* the courses that student is enrolled in, in no particular order. What is the big-O complexity?

5. A class has a *name* (e.g., `Actor`) and zero or more *subclasses* (e.g., the class with name `Flame` or the class with name `Bacterium`). The following program reflects this structure:

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Class
{
  public:
    Class(string nm) : m_name(nm) {}
    string name() const { return m_name; }
    const vector<Class*>& subclasses() const { return m_subclasses; }
    void add(Class* d) { m_subclasses.push_back(d); }
    ~Class();
  private:
    string m_name;
    vector<Class*> m_subclasses;
};

Class::~Class()
{
    for (size_t k = 0; k < m_subclasses.size(); k++)
        delete m_subclasses[k];
}

void listAll(string path, const Class* c)  // two-parameter overload
{
    You will write this code.
}

void listAll(const Class* c)  // one-parameter overload
{
    if (c != nullptr)
        listAll("", c);
}

int main()
{
    Class* d1 = new Class("Salmonella");
```

```
    listAll(d1);
    cout << "====" << endl;
    d1->add(new Class("AggressiveSalmonella"));
    Class* d2 = new Class("Bacterium");
    d2->add(new Class("EColi"));
    d2->add(d1);
    Class* d3 = new Class("Goodie");
    d3->add(new Class("RestoreHealthGoodie"));
    d3->add(new Class("FlamethrowerGoodie"));
    d3->add(new Class("ExtraLifeGoodie"));
    listAll(d3);
    cout << "====" << endl;
    Class* d4 = new Class("Actor");
    d4->add(d2);
    d4->add(new Class("Flame"));
    d4->add(d3);
    listAll(d4);
    delete d4;
}
```

This main routine should produce the following output (the first line written is `Salmonella`, not an empty line):

```
Salmonella
====
Goodie
Goodie=>RestoreHealthGoodie
Goodie=>FlamethrowerGoodie
Goodie=>ExtraLifeGoodie
====
Actor
Actor=>Bacterium
Actor=>Bacterium=>EColi
Actor=>Bacterium=>Salmonella
Actor=>Bacterium=>Salmonella=>AggressiveSalmonella
Actor=>Flame
Actor=>Goodie
Actor=>Goodie=>RestoreHealthGoodie
Actor=>Goodie=>FlamethrowerGoodie
Actor=>Goodie=>ExtraLifeGoodie
```

Each call to the one-parameter overload of `listAll` produces a list, one per line, of the inheritance path to each class in the inheritance tree rooted at `listAll`'s argument. An inheritance path is a sequence of class names separated by "=>" (no spaces). There is no "=>" before the first name in the inheritance path.

a. You are to write the code for the two-parameter overload of `listAll` to make this happen. You must not use any additional container (such as a stack), and the two-parameter overload of `listAll` must be recursive. You must not use any global variables or variables declared with the keyword `static`, and you must not modify any of the code we have already written or add new functions. You may use a loop to traverse the vector; you must not use loops to avoid recursion.

   Here's a useful function to know: The standard library string class has a + operator that concatenates strings and/or characters. For example,

   ```
   string s("Hello");
   string t("there");
   string u = s + ", " + t + '!';
   // Now u has the value "Hello, there!"
   ```

   It's also useful to know that if you choose to traverse an STL container using some kind of iterator, then if the container is const, you must use a `const_iterator`:

```
void f(const list<int>& c)  // c is const
{
    for (list<int>::const_iterator it = c.begin(); it != c.end(); it++)
        cout << *it << endl;
}
```

(Of course, a vector can be traversed either by using some kind of iterator, or by using operator[] with an integer argument).

For this problem, you will turn a file named `list.cpp` with the body of the two-parameter overload of the `listAll` function, from its "void" to its "}", no more and no less. Your function must compile and work correctly when substituted into the program above.

b. We introduced the two-parameter overload of `listAll`. Why could you not solve this problem given the constraints in part a if we had only a one-parameter `listAll`, and you had to implement *it* as the recursive function?

Turn it in

By Wednesday, March 11, there will be a link on the class webpage that will enable you to turn in this homework. Turn in one zip file that contains your solutions to the homework problems. The zip file must contain two files:

- hw.docx, hw.doc, or hw.txt, a Word document or a text file with your solutions to problems 1, 2, 3, 4, and 5b.
- `list.cpp`, a C++ source file with the implementation of the two-parameter overload of the `listAll`function for problem 5a.