CSCI-1200 Data Structures — Fall 2016
Test 3 — Solutions

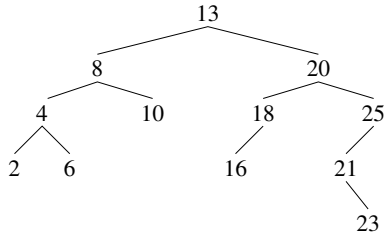
1 Upside-Down Binary Search Tree [/ 36]

1.1 Binary Search Tree Diagram Warmup [/ 7]

Draw the tree that results when this sequence of 12 numbers is inserted (in this order) into a binary search tree using the algorithm covered in lecture and lab.

```
13  8  20  4  18  25  2  10  16  21  6  23
```

Solution:



Which numbers are the leaves of this tree? *Hint: There are five.*

Solution: `2 6 10 16 23`

Upside-Down Binary Search Tree

Ben Bitdiddle has come up with another wacky tree scheme (with questionable usefulness). He proposes to represent a binary search tree not with a single pointer to the tree root, but instead with an STL list of the leaf nodes. And then it follows that each `Node` will store only a pointer to its parent.

```
class Node {
 public:
  Node(int v) : value(v), parent(NULL) {}
  int value;
  Node* parent;
};
```

Ben is sure that because his new representation only has one pointer per `Node` this structure will be much more memory efficient than the typical binary tree. Here's how he proposes to construct the tree you drew above:

```
std::list<Node*> leaves;
Insert(leaves,13);   Insert(leaves,8);    Insert(leaves,20);   Insert(leaves,4);
Insert(leaves,18);   Insert(leaves,25);   Insert(leaves,2);    Insert(leaves,10);
Insert(leaves,16);   Insert(leaves,21);   Insert(leaves,6);    Insert(leaves,23);
assert (leaves.size() == 5);
```

1.2 `BelongsInSubtree` [/ 14]

Rather than jumping straight into the implementation of the `Insert` function, Alyssa P. Hacker suggests that Ben start by implementing the `BelongsInSubtree` function. This *recursive* function takes in two arguments: `node`, a pointer to a `Node` already in the upside down tree, and `value`, an element we would like to add. The function returns false if placing `value` within a subtree of `node` violates the binary search tree property of the whole tree and true otherwise. Note: Ben's tree does not allow duplicate elements.

Solution:

```
bool BelongsInSubtree(Node* node, int value) {
  if (node == NULL) return false;
  // check for duplicate
  if (node->value == value) return false;
  // made it to the root!  this value fits on this branch
  if (node->parent == NULL) return true;
  // doesn't belong to the left of the grandparent
  if (node->value < node->parent->value && value > node->parent->value) return false;
  // doesn't belong in the right subtree of the grandparent
  if (node->value > node->parent->value && value < node->parent->value) return false;
  return BelongsInSubtree(node->parent,value);
}
```

The implementation of `Insert` will call the `BelongsInSubtree` function on each `Node` in the tree. Note: This function will return true for at least one, but possibly many nodes in the tree! Of these possible choices, `Insert` will select the node that is furthest (in number of parent pointer links) from the root of the tree. For example, if we'd like to insert the value `15` into our example tree, there are four nodes that will return true for the `BelongsInSubtree` function above. What values are stored in those nodes? Which of these nodes will be selected by `Insert` as the immediate parent for '`15`'?

Solution: The Nodes storing 13, 20, 18, and 16 all return true.
15 will be placed as the left child of 16.

1.3 Destroy Tree [/ 15]

Now, let's write the `DestroyTree` function, which cleans up all of the dynamically allocated memory associated with Ben's upside-down tree leaving a valid empty tree.

Solution:

```
void DestroyTree(std::list<Node*> &leaves) {
  // use an STL set to collect all tree nodes (removes duplicates)
  std::set<Node*> nodes;
  for (std::list<Node*>::iterator itr = leaves.begin(); itr != leaves.end(); itr++) {
    Node* tmp = *itr;
    while (tmp != NULL) {
      if (!nodes.insert(tmp).second)
        break;
      tmp = tmp->parent;
    }
  }
  // now delete everything
  for (std::set<Node*>::iterator itr = nodes.begin(); itr != nodes.end(); itr++) {
    delete *itr;
  }
  // set the tree to the empty tree
  leaves.clear();
}
```

If the tree contains n elements, and is approximately balanced, what is the order notation of your implementation of destroy tree? Write 2-3 sentences justifying your answer.

Solution: Walking from each leaf to root: # of leaves * tree height = n/2 * log(n) = O(n log(n)). But by checking the return value of set insertion, it's only O(n) total set insertions. Each set insertion costs log(n), so it's O(n log(n)) to collect the nodes without duplicates. It only costs O(n) to iterate over the nodes and delete them. Final answer: O(n log(n)).

2 Halloween History Maps [/ 34]

```
r Bob Williams pirate
r Chris Smith doctor
r Chris Smith doctor
r Bob Williams zombie
r Alice Jones zombie
r Chris Smith pirate
r Bob Williams pirate
r Chris Smith zombie
r Chris Smith elf
r Bob Williams doctor

h pirate
h elf
h doctor
```

The costume shop owner from Homework 7 has asked for help predicting what costumes their indecisive customers might choose in the future. Looking at the history of costume rentals they suspect there might be a pattern when a customer changes their mind about their Halloween costume.

For the example data on the left, we can see two instances where a customer switched from a pirate costume to a doctor costume, and only once did a customer switch from a pirate costume to a zombie costume. Here is the output we expect from the sample '`r`' = *rental* and '`h`' = *print costume history* commands:

```
history for pirate
   next rental was doctor 1 time(s)
   next rental was zombie 2 time(s)
no next rental history for elf
history for doctor
   next rental was pirate 1 time(s)
```

The shop owner emphasizes the need for fast performance in this implementation, since the system will be handling the records for thousands of customers and costumes in many different cities.

2.1 Data Structure Sketch [/ 6]

Let's store this data in two variables, one with the current customer information, and the second with the costume history. (You'll specify the typedefs in the next part). Sketch the contents of these variables after the rental commands above. Follow the conventions from lecture for your diagrams.

`PEOPLE_TYPE people;`

Alice Jones	zombie
Bob Williams	doctor
Chris Smith	elf

`HISTORY_TYPE history;`

doctor	pirate	1
pirate	doctor	1
	zombie	2
zombie	elf	1
	pirate	1

Solution:

2.2 The `typedefs` [/ 4]

Next, fill in these `typedef` declarations.

Solution:
```
typedef std::map<std::string,std::string> PEOPLE_TYPE;
typedef std::map<std::string,std::map<std::string,int> > HISTORY_TYPE;
```

2.3 Implementation of the Rental Command [/ 9]

Now, complete the implementation:

```
int main() {
  PEOPLE_TYPE people;
  HISTORY_TYPE history;
  std::string first, last, costume;
  char c;
  while (std::cin >> c) {
    if (c == 'r') {
      std::cin >> first >> last >> costume;
```
Solution:
```
      std::string name = first + " " + last;
      PEOPLE_TYPE::iterator itr = people.find(name);
      if (itr != people.end() && itr->second != costume) {
        history[itr->second][costume]++;
      }
      people[name] = costume;
    }
```

NOTE: main function code continued on next page...

2.4 Implementation of the History Command [/ 10]

```
    else {
      assert (c == 'h');
      std::cin >> costume;
```
Solution:
```
      HISTORY_TYPE::const_iterator itr = history.find(costume);
      if (itr != history.end()) {
        std::cout << "history for " << costume << std::endl;
        std::map<std::string,int>::const_iterator itr2;
        for (itr2 = itr->second.begin(); itr2 != itr->second.end(); itr2++) {
          std::cout << "   next rental was " << itr2->first << " " << itr2->second << " time(s)" << std::endl;
        }
      } else {
        std::cout << "no next rental history for " << costume << std::endl;
      }
    }
  }
}
```

2.5 Order Notation [/ 5]

If the shop has p customers, c costumes, and r total rental events, what is the order notation for performing a single rental (the `'r'` command)? Write 1-2 sentences justification.
Solution: $O(\log p)$ to find this customer in the people map. $O(\log c)$ to find the old costume in the history map. $O(\log c)$ to find the new costume in the interior history map. Overall: $O(\log p + \log c)$.

What is the order notation for performing a history query (the `'h'` command)? (justify your answer)
Solution: $O(\log c)$ to the costume in the history map. $O(c)$ to loop over all of the "next" costumes in the interior history map. Overall: $O(c)$.

3 Allergic to `for` and `while` [/ 17]

Complete the functions below *without using any additional* `for` *or* `while` *expressions.* Given an STL `vector` of words, find all pairs of those words that share at least one common letter. For example, if `words` contains: `apple boat cat dog egg fig` then `common(words)` should return:
`(apple,boat) (apple,cat) (apple,egg) (boat,cat) (boat,dog) (dog,egg) (dog,fig) (egg,fig)`

```
typedef std::set<std::pair<std::string,std::string> > set_of_word_pairs;

bool common(const std::string &a, const std::string &b) {
  for (int i = 0; i < a.size(); i++)
    for (int j = 0; j < b.size(); j++)
      if (a[i] == b[j]) return true;
  return false;
}

void common(set_of_word_pairs &answer, const std::vector<std::string>& words, int a, int b) {
```

Solution:

```
  if (common(words[a],words[b])) {
    answer.insert(std::make_pair(words[a],words[b]));
  }
  if (b < words.size()-1)
    common(answer,words,a,b+1);
  }


}

void common(set_of_word_pairs &answer, const std::vector<std::string>& words, int a) {
```

Solution:

```
  if (a < words.size()-2) {
    common(answer,words,a+1);
  }
  common(answer,words,a,a+1);


}

set_of_word_pairs common(const std::vector<std::string>& words) {
```

Solution:

```
  set_of_word_pairs answer;
  if (words.size() >= 2) {
    common(answer,words,0);
  }
  return answer;


}
```

4 Wait, was this an iClicker Question? [/ 10]

Grading Note: -2pts each iclicker unanswered or incorrect.

4.1 For a *balanced* binary search tree with n elements what is the order notation for the worst case single call to `operator++`, the average (or amortized) single call to `operator++`, the total running time for a loop increment from `begin()` to `end()`.

Solution: C

(A) Single worst: $O(1)$ Single average/amortized: $O(1)$ Total: $O(n)$

(B) Single worst: $O(log\ n)$ Single average/amortized: $O(log\ n)$ Total: $O(n)$

(C) Single worst: $O(log\ n)$ Single average/amortized: $O(1)$ Total: $O(n)$

(D) Single worst: $O(n)$ Single average/amortized: $O(log\ n)$ Total: $O(n\ log\ n)$

(E) Single worst: $O(log\ n)$ Single average/amortized: $O(log\ n)$ Total: $O(n\ log\ n)$

4.2 Which of the following statements about STL container types is *true*?

Solution: A

(A) A program that uses an STL `set` can easily be changed to use an STL `map` instead, with little or no performance impact.

(B) A program that uses an STL `map` can easily be changed to use an STL `set` instead, with little or no performance impact.

(C) A program that uses an STL `set` can easily be changed to use an STL `list` instead, with little or no performance impact.

(D) A program that uses an STL `vector` can easily be changed to use an STL `list` instead, with little or no performance impact.

(E) A program that uses an STL `map` can easily be changed to use an STL `vector` of STL `pairs` instead, with little or no performance impact.

4.3 Which of the following is *true* for the STL map iterators?

Solution: E

(A) STL list and STL map iterators are typedef-ed as simple pointers to a data element in the container.

(B) Since STL map has an `operator[]`, it is like STL vector and thus I can not only move a map iterator forward just one spot (using `itr++`), but I can also jump forward an arbitrary number of spots (e.g., 25) using `itr + 25`.

(C) Visiting every element in an STL map is faster than visiting every element in an STL vector.

(D) Data is accessed in the order it was inserted.

(E) None of the above.

4.4 What is the purpose of Red-Black Trees?

Solution: C

(A) Ensure that the tree is exactly balanced.

(B) Minimize the memory usage of the tree.

(C) Guarantee that the tree height is no more than 2 * log(n).

(D) Allow the tree to store duplicates.

(E) None of the above.

4.5 Which of the following guidelines for designing operator overloading is *false*?

Solution: D

(A) Only use friend functions if a function cannot be implemented as a member function and implementation as a non-member function would require writing accessors/modifiers to private data that are otherwise unnecessary and poor class design.

(B) Don't change the intuitive meaning of an operator.

(C) The input & output stream operators should return the stream object by reference so calls may be nested/chained.

(D) You can overload operators for every symbol on your keyboard!

(E) Do not attempt to return a local variable by reference. The compiler will give you a warning that *should not be ignored.*

4.6 Which of the following is *false* for the STL pair class?

Solution: C

(A) Pairs are cool, they can be used to return two values from a function (more intuitive than that weird trick using pass-by-reference arguments).

(B) For a previous homework I have (or could have easily) written my own STL pair-like class gluing together two related items.

(C) The first item in an STL pair is always const and cannot be changed.

(D) `first` and `second` are public member variables of the STL pair struct; therefore, they can be accessed and edited directly by the user (we don't need to use a `get` or `set` member function).

(E) I don't need to put `#include <utility>` if I'm using an STL map in the same file (because it is indirectly included with `#include <map>`).