CS106X
Winter 2017

Chris Gregg

Section Handout #7

This week has continued practice with binary trees, in particular binary search trees, heaps, and hashing. When working on your solutions, remember that you must not leak memory. Assume the following structure been declared:

TreeNode

```
struct TreeNode {
  int data;
  TreeNode *left;
  TreeNode *right;
};
```

You can also assume that there is a helper function, `deleteTree`, that takes in the pointer to a tree and frees all the memory associated with that tree.

1. Binary Search Tree Insertion

Draw the binary search tree that would result from inserting the following elements in the given order. Remember that strings will be stored in lexicographic order.

a. Jaques, Sunny, Klaus, Violet, Beatrice, Bertrand, Kit, Lemony
b. Leslie, Ron, Tom, Jerry, Larry, Garry, April, Andy
c. Aaron, Anu, Chris, Colin, Jason, Leslie, Wesley

2. Is It a BST?

Write a function that takes in the root of a tree of integers and returns whether or not the binary tree is in valid BST order.

3. Tree Rotations

A "rotation" is an operation used on BSTs to reposition elements; this is frequently used to improve the performance of the BST by moving elements that are requested more frequently closer to the root, or to take an unbalanced tree and make it balanced. A left-rotation pivots around the link between a node and its parent, so that the child of the link rotates counterclockwise to become the new parent of its former parent. An example of this transformation is below, where we pivot around the link between the 5 node and its parent, the 3 node:

Note that a constant number of pointers need to be updated. The former parent of the 5 node becomes its left child, and the former left child of the 5 node becomes the right child of its previous parent. Because of these operations, the BST property is held constant.

First, write a function that takes in the pointer to the parent of the link you're pivoting around (that is, a pointer to a TreeNode pointer) and changes a constant number of pointers so that the right child of

Thanks to Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106 instructors for contributing content on this handout. Thanks to Wesley Rodriguez and Colin Kincaid for proofreading.

the given node becomes the parent.

Second, write a function that given a pointer to the root of a binary search tree (that is, a pointer to a TreeNode pointer) and an integer value, and uses both your left rotation function as well as a corresponding right rotation function (which you can assume exists) and bubbles the specified value up to the root. You may assume that the value is present, although it's possible to leave the tree unaltered if the value is missing.

4. Level-Order Heaps
In lecture, we learned about level order traversals of binary trees – that is, printing out the root, then its children, then its grandchildren, and so on. Write a function that, given an integer heap array and its size, prints out an level-order traversal of the heap.

5. On the Level
Write a function that, given an integer heap array, a level within the heap (where 0 is the root), and the size of the heap, prints out the elements within that level.

6. Hash Functions
Let's say we have a class `StRiNg` that works like the string class that you're used to, with one exception: equality is case insensitive (meaning that two `StRiNG`s are considered equal if they contain the same characters ignoring upper and lower cases). Which of the following functions are legal hash functions? Which of the following functions are *good* hash functions?

```
int hash1(StRiNg &s) {
  return 0;
}

int hash2(StRiNg &s) {
  int sum = 0;
  for (int i = 0; i < s.length(); i++) {
    sum += s[i];
  }
  return sum;
}
```

```
int hash3(StRiNg &s) {
  return (int) &s;
}

int hash4(StRiNg &s) {
  int product = 1;
  for (int i = 0; i < s.length; i++) {
    product *= tolower(s[i]);
  }
  return product;
}
```

7. Hash It Out
Simulate the behavior of a hash map as described in lecture. Assume the following:

- the hash table array has an **initial capacity of 10**
- the hash table uses **separate chaining** to resolve collisions
- the **hash function** returns the absolute value of the integer key, mod the capacity of the hash table
- **rehashing** occurs at the *end* of an add where the load factor is **>= 0.5** and **doubles the capacity** of the hash table

Draw an array diagram to show the final state of the hash table after the following operations (on the following page) are performed.

```
HashMap map;
map.put(18, 22);
map.put(6, 40);
map.put(16, 19);
map.put(6, 999);
map.put(276, 55);
map.put(8, 33);
map.put(31, 19);
map.remove(19);
map.remove(16);
map.put(100, 14);
if (map.containsKey(276)) {
  map.remove(55);
}
map.put(-18, 4);
map.put(26, 5);
```

8. Hashing and Rehashing

Assume you have a hash table with 6 buckets. Diagram the result of putting the following values into the hash table, using a hash function that adds the values of each letter in the string (where 'a' is 1, 'b' is 2, and so on). Handle collisions appropriately. Then, diagram the resulting bucket arrangement after rehashing it to have 12 buckets. Assume you rehash by iterating through each bucket in order and insert each element into the new array as you encounter them.

```
cabbage, baggage, deadbeaf, cafe, badcab, feed
```

9. Open Addressing

In lecture, we talked about handling hashing collisions by putting elements that hash into the same bucket into a linked list. This isn't the only way to deal with collisions, however. Open addressing (also called closed hashing) handles collisions by finding another location in the array to store the new element. Take, for instance, the linear probing technique.

- On **insertion**, if there's a collision, you advance in the array one element at a time until you find an empty location, and you store the element there. For example, if an element hashes to index 3, but that element is in use, you first consider 4, then 5, and so on, until you find an empty element (including possibly wrapping around to index 0)
- On **lookup**, you do a linear search starting from the hash value until you find the key, or until you find an empty cell, indicating that the key is not in the map.
- On **removal**, you find and remove the entry from its current position. However, since this might introduce gaps in the table, you might need to fill in those gaps. Upon removing cell *i*, you must search forward until you find another empty cell or a key that can be moved to cell *i* (that is, a key whose hash value is equal to or earlier than *i*). If an empty cell is found, then emptying *i* is safe. Otherwise, you must empty the new cell *j* and repeat this process from *j*.

Implement the public interface of the below class (on the following page), using linear probing to handle collisions.

Add whatever private members you find necessary. Your hash table should be able to hold at least 10 elements, but you don't need to worry about rehashing; once your hash table has exhausted its initial capacity, you should throw an exception.

```
class OpenHashTable {
public:
  OpenHashTable();
  ~OpenHashTable();

  // puts the key/value pair into the map
  void put(string key, string value);
  // returns the value associated with the key, or "" if the key doesn't exist
  string get(string key);
  // removes the key/value pair from the map, or does nothing if it doesn't exist
  void remove(string key);
private:
  int hash(string key); // this is implemented for you

  struct OpenEntry {
    string key;
    string value;
    bool inUse;
  };
};
```