CS104: Data Structures & Object-Oriented Programming
Summer 2021 - Midterm Exam
06/30/21, 10:00 AM – 12:00 PM + 15 min. to upload
(Submit on Gradescope by 12:15 PM)

[Complete all the information in the box below.]

Name:_**(No need to fill this out since you are on Gradescope)**_

Student ID: _____ **Email:** _____@usc.edu

Lecture section (Circle One):

Redekopp		
9:30 a.m.		

Ques.	Your score	Max score	Time
1		8	10 min.
2		9	12 min.
3		8	13 min.
4		7	20 min.
5		8	25 min.
6		10	40 min.
Total		50	

Note: The last page is scratch paper. Submit it with your exam

Data structures and algorithms that may be used for certain problems

In each problem you may assume the use of the following:
- `std::sort(vector<T>& values)` which will perform an n*log(n) sort in ascending order'

- `std::find(iterator first, iterator last, T target)` which finds the target in Θ(n)

Assume the various STL containers are available:
- **vector<T>:** Traditional array-based list with O(1) operations:
  - push_back(), back(), operator[]
- **singly<T>:** A singly-linked list (with tail pointer) with O(1) operations:
  - back(), front(), push_back(), push_front(), and pop_front()
- **queue<T>:** Implements a queue with O(1) operations:
  - front(), push(), and pop()
- **stack<T>:** Implements a stack with O(1) operations:
  - top(), push(), and pop()
- **set<T>**: Standard set implementation with O(log n) operations:
  - insert(), erase(), find()
- **map<K,V>**: Standard map implementation with O(log n) operations:
  - insert(pair<K,V>), erase(key), find(key)

1. **(8 pts.) Abstract Data Types**: Choose the best abstract data type (List, Set, Map, PQ, Queue, Stack) given the description of the desired data structures below. Be as specific as possible (don't answer List if a Queue is applicable). Also show what types would be used as template arguments (e.g. `map<string, int>` or `stack<double>`). If multiple "best" options exist, choose either. Give a **SHORT** 1-2 sentence justification for your choice (don't waste time on a long explanation...it won't help you get any more credit than a short answer).

    1.1. A structure to support the following: an academic advisor wants to track the `students` she advises each day over the course of a semester. Given a day, she'd like to quickly check if she advised a particular student and also be able to quickly go from one day's students to the next or previous day's students.

    1.2. A structure for storing the percentage of people vaccinated in each US state (and quickly be able to update that percentage).

    1.3. A structure to store the answers on a website like StackOverflow.com. The answers should be displayed in order of "upvotes" from highest number of upvotes to least.

    1.4. A structure to track who is in each checkout/cashier lane and their order for an entire grocery store where `customers` may choose a checkout line (given by an integer) to then wait in.

2. **Heaps (9 pts.)**

   Given the following array numbers: **[6, 12, 3, 11, 14, 8, 1, 7, 15, 9, 2]** (the number 6 is in position 0 and 2 in position 10), find the **min-heap** that is a result of calling the make-heap (build-heap) algorithm that uses *heapify* on the array and that runs in linear time.

   In the area below show the final array contents (similar to the format we showed above). For full credit, you must also show your work at each step in the algorithm by drawing the complete **tree** representation (not array form, but tree form) after each heapify step until the algorithm is complete. **Hint:** You don't need to call heapify on EVERY node but can start a certain location. For each call, you need only show the relevant part of the tree that heapify operats on. Gradescope: Upload a picture or PDF of your work in the file upload area below.

   Final array contents after the algorithm completes:

   [__, __, __, __, __, __, __, __, __, __, __]

   Diagrams of intermediate trees after each heapify call **(indicate what location you start your heapify call on)**.

3. Linked Lists and Recursion (8 pts.)

Given the code in the image below answer the following questions regarding a call to `llmystery(head, 2)` assuming head points to the linked list of values:

1	2	3	4	5

Answer the following questions:

3.1. **True/False:** Will the original linked list be modified (values or pointers)? _____

3.2. On the next page, show what will be printed by the code, and further show a call tree (box diagram) of the recursive calls (with relevant arguments) made during execution.
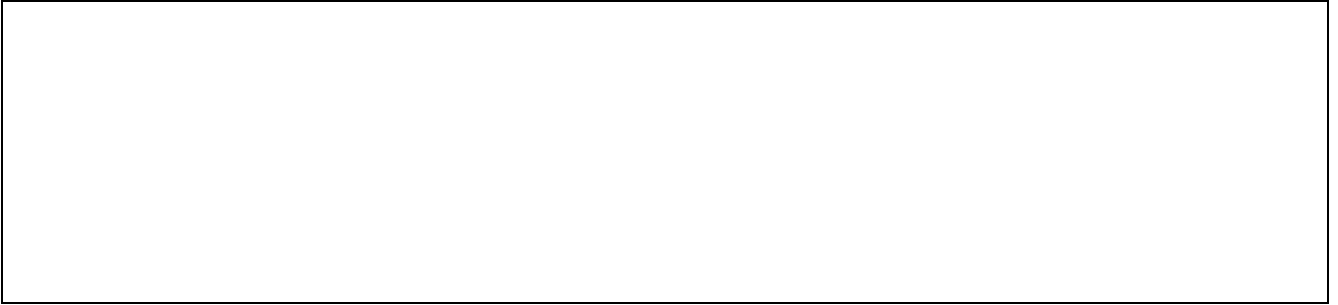
```cpp
struct IntItem {
  int val;
  IntItem *next;
  IntItem(int v, IntItem* n) { val = v; next = n; }
};

void llmystery(IntItem* head, size_t loc);
int llmysteryHelper(IntItem* head, size_t loc, IntQueue& v);

void llmystery(IntItem* head, size_t loc)
{
  IntQueue q;
  int t = llmysteryHelper(head, loc, q);
  cout << t << endl; // *** COUT PRINT HERE ***
  while(!q.empty()) {
    cout << q.front() << " ";  // *** COUT PRINT HERE ***
    q.pop_front();
  }
  cout << endl;
}

int llmysteryHelper(IntItem* head, size_t loc, IntQueue& q)
{
  if(head == nullptr) {  return 0; }
  else if(loc > 0) {
    int t = llmysteryHelper(head->next, loc-1, q);
    q.push_back(head->val);
    return t;
  }
  else {
    q.push_back(head->val);
    return head->val + llmysteryHelper(head->next, loc, q) ;
  }
}
```

Show what will be printed by the call to `llmystery(head, 2)`

Upload a diagram of the call tree (box diagram) of the recursive calls made during execution.

4. Class Organization, Heaps, and BSTs (7 pts.)

> **Complete your code in the provided `bst-heap.h` at the bottom of the file. You will **NOT** be able to compile and test this code because we do not provide the implementation of the base `BST<Key>` class.** Assume it is provided and works. We will visually grade your MaxHeap<Key> class assuming the BST class works.

Suppose you are provided a complete, templated binary search tree (BST) class: **BST<Key>.** It does not necessarily guarantee balance. We will assume type Key supports all the basic comparison operators: <, >, ==, !=, etc.

You are now asked to write a **MaxHeap<Key>** class to implement a max heap (priority queue). It should use the **BST<Key>** class to implement it (though how to structure the relationship between these two is part of the question and left to you to decide). You may assume no duplicate keys are added to the heap.

Your MaxHeap<Key> must implement the following public interface using the standard definitions of the push, pop, and top operations. If **top()** or **pop()** is called on an empty heap, **throw std::out_of_range exception.**

```
template <typename Key>
class Heap /* your choice */
{
public:
    Heap();
    ~Heap();
    void push(const Key& newKey);
    void pop();
    const Key& top() const;  // throws std::out_of_range if empty
private:
    // add any data members or helper functions as necessary
};
```

4.1. Finish the implementation of the class / functions in the provided `bst-heap.h`.

- Your implementation should utilize/re-use as much (as reasonable) of the `BST<Key>` implementation to avoid unnecessary code. (There may be some duplication in your Heap implementation, but be judicious...if a BST operation can already accomplish a task, try to use it)
- Your runtime does not need to match that of a traditional Heap

4.2. Analyze the runtime of your `top()` implementation (show a very short justification or work).

5. Binary Trees and Recursion (8 pts.)

> **Use the provided skeleton file `remdepth.cpp` to complete the code.**

Write a recursive function: **`Node* remDepth(Node* root, int depth)`** to delete all nodes **BELOW** a certain depth from a **binary** tree (not necessarily BST), and return the pointer to the (potentially now NULL) root. Your implementation **MAY NOT** use loops anywhere. You may define helper function(s), as necessary.

An example, showing how we define depth and how the function should work is illustrated below.
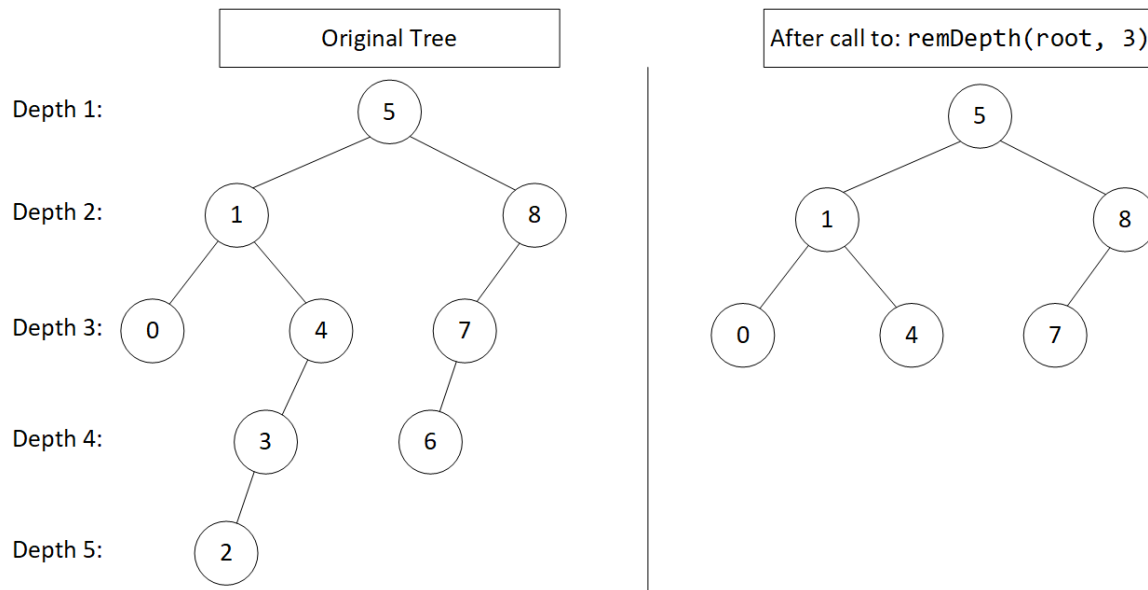
Other examples: A call to **`remdepth(root, 0)`** would cause the entire tree to be deleted, in which case you should return **`nullptr`**. A call to **`remdepth(root, 1)`** would cause all nodes **except the root** to be deleted.

If no nodes are below the specified depth, simply do nothing (i.e. do not alter the tree in any way).

Your code should run in **Theta(n)**, where **n** is the number of nodes in the tree.

6. STL and ADTs (10 pts.)

> **Use the provided skeleton file `travel.cpp` to complete the code.**

In this problem you may only use: `map<K,V>`, `set<K>`, `stack<T>`, and `queue<T>`. **No vector, deque, or list**.

Tommy, the forgetful tourist, enjoys travelling around the country. You will write a class **Tourist**, to help him plan and track his travels. This class will implement 4 primary functions:

```
addPlaceToVisit()
visitNextLocation()
currentLocation()
previousVisit()
```

Tommy talks to people who tell him about fun **locations** he should visit. Locations are represented by the name of the place and its x,y coordinates (e.g. latitude and longitude). He adds them to his "places to visit itinerary" via a call to **addPlaceToVisit()** which should quickly check if they are already on his itinerary of places to visit, and if not, store the location. When he's ready to travel to the next location he calls **visitNextLocation()** which chooses a new place to visit, removes it from his places to visit, and updates his current location. By default you can choose any place on his itinerary to go next, but for bonus points, you should ensure that he visits places on his itinerary in order from **west to east** (i.e. **visitNextLocation()** should always give the *west-most (lowest x-value)* unvisited location.

Further, he wants to track where he's visited because he often leaves one of his belongings somewhere he visited and needs to keep going back to previous locations to find his lost belongings. This process is accomplished by (potentially many) calls to **previousVisit()** which should retrace the locations he visited before his current location from **most-to-least** recently visited. To keep things simple, once he finds his item at a location, the next call to **visitNextLocation()** should resume his travels by going to the next **new** place to visit (rather than returning back through the same locations he was just at looking for his items).

As an example, suppose Tommy hears about locations: **B, A, C, A, E, D** and adds them to his places to visit (removing the duplicate **A**). By making multiple calls to **visitNextLocation** he visits to **A, B, C**, then **D** (let's assume these are in west-to-east order). At location **D** he realizes he left his USC hat somewhere. By making calls to **previousVisit()** he will retrace his steps to **C** and then **B** at which point he finds his hat. From there a call to **visitNextLocation** would yield the only unvisited location that remains on his places to visit: **E**. But if he arrives at E and realizes he also lost his earbuds, a new sequence of calls to **previousVisit()** should yield **D**, **C**, **B**, and so on. Thus, when we return to previously visited locations we cannot forget that order but must restore those locations when we start traveling to new locations via calls to **visitNextLocation**.

Any other behavior not specified here is left to your discretion. (You need not add any features we have not specified.) However, you must adhere to the runtime requirements given in the function prototypes in the **Tourist** class declaration.

Intentionally blank for scratch work. Please turn it in with your exam:

Name: _____ Section time: _____