CS106B

Instructor: Cynthia Lee

Autumn 2019

December 9, 2019

FINAL EXAM

1. Pointers and Memory (13pts). Consider the following code.

```
char* tony = new char;
*tony = 'A';
Endgame captmarvel[2];
captmarvel[0].label = *tony;
captmarvel[1].label = 'B';
captmarvel[0].ptr = new Endgame;
captmarvel[1].ptr = nullptr;
Endgame* tchalla = new Endgame;
tchalla->label = 'C';
tchalla->ptr = &captmarvel[1];
captmarvel[0].ptr->label = 'D';
captmarvel[0].ptr->ptr = tchalla;
delete tony; // ☹
//Draw the state of memory now
```

```
struct Endgame {
   char label;
   Endgame* ptr;
};
```

(a) (10pts) Draw the state of memory at the end of the execution of this code. Be careful in showing where your pointers originate and terminate (starting in a box vs. pointing to the border of a box). Leave uninitialized or unspecified areas blank, and clearly mark null (draw a slash through the box). Draw the components in the appropriate stack and heap areas marked for you. Mark memory that has been deleted by enclosing it in a circle with a slash through it, like this: ⊘ but leave it where it is.

Stack:	Heap:

SUNet: @stanford.edu

(b) (3pts) What would the following code print, if executed immediately following the code above (within the same function)? DO NOT MODIFY YOUR DRAWING on the previous page to show any effects of this code or to add any additional variables declared here. Just analyze what would print.

```
Endgame* curr = &captmarvel[0];
while (curr->ptr->ptr != nullptr) {
    cout << curr->ptr->label << " ";
    curr = curr->ptr;
}
cout << "Done!" << endl;
```

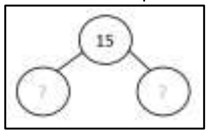
Output:

[scratch space]

2. Binary Heap (20pts). Imagine that we have changed our implementation of the Priority Queue ADT so that it uses a binary **max**-heap (largest number has the highest priority).

(a) (12pts) Draw a diagram of the tree shape of the heap after enqueuing the following priorities in the order given: **15, 3, 2, 20, 1, 5** (for this priority queue we don't have a separate value, just the priority). Your diagram should be a tree shape with circles for nodes (containing the `int` key for that node), as shown in the format example at right.

Format example:



i. Tree after inserting 15:	ii. Tree after inserting 3:
iii. Tree after inserting 2:	iv. Tree after inserting 20:
v. Tree after inserting 1:	vi. Tree after inserting 5:

SUNet: @stanford.edu

(b) (4pts) Recall that in lecture we learned a trio of tree traversal algorithms: pre-order, in-order, and post-order. In the box below, write a comma-separated list of the priority keys of your final binary max heap (as shown in part (a), box vi), in the order in which the nodes would be visited during an in-order traversal. (*Hint and reminder:* this tree is a binary heap, not a BST.)

(c) (4pts) Continuing from the final heap in part (a) (part (a), box vi), draw a diagram of the tree shape of the heap after calling dequeue twice.

vii. Tree after calling dequeue once:	viii. Tree after calling dequeue a second time:

[scratch space]

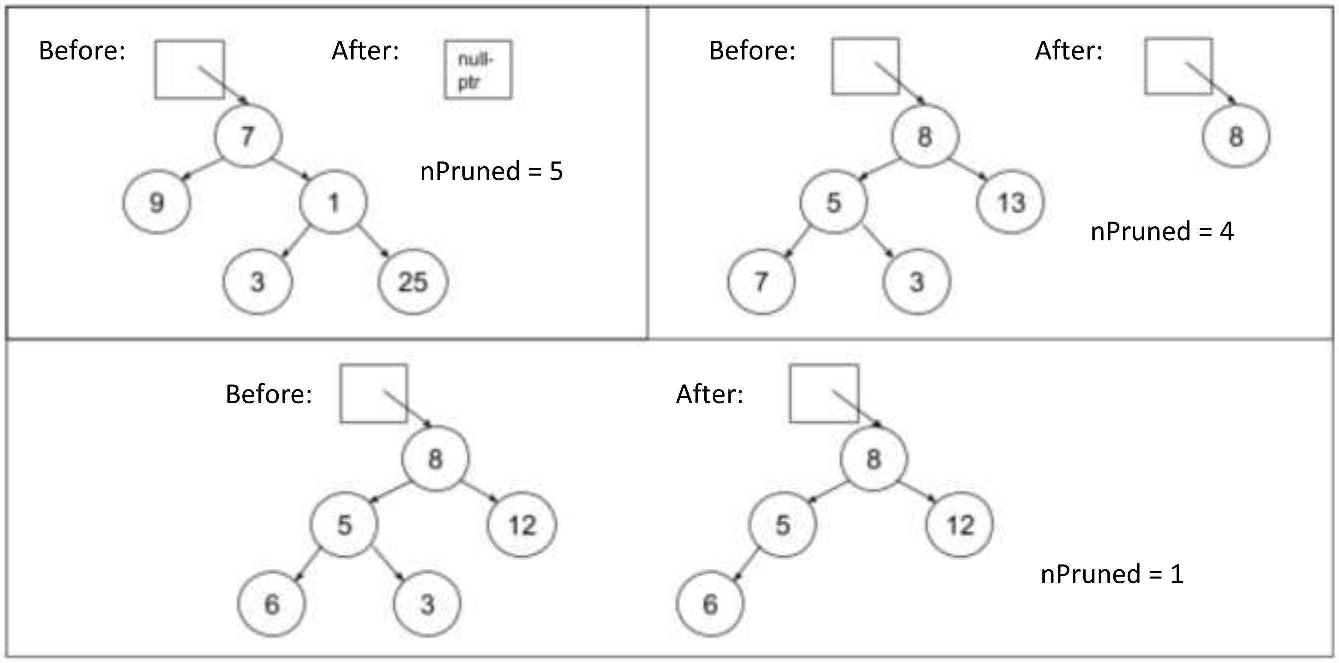
3. Trees (10pts). Expert gardeners will tell you that winter is the best time to prune summer-flowering trees, so this problem asks you to prune a binary tree. Your function will examine a given tree and prune off all the subtrees of that tree when all the numbers in that subtree happen to be odd. (The ordering of the numeric keys in the tree is arbitrary, not necessarily BST-style order.)

- A full-credit solution must visit each node only once (*hint:* using one of our standard traversals).
- To prune a node, `delete` it to free the memory (you may assume all nodes were dynamically allocated) and set any pointer pointing to it to `nullptr`.
- You will keep track of the total number of pruned nodes via a reference parameter.
- Here are the node struct for the tree and the function header:

```
struct Node {
    int key;
    Node* left;
    Node* right;
};
```

```
/* @param  root is a pointer to the root node of a binary tree. It is passed by reference
 *         in case you need to prune the root itself.
 * @param  nPruned is an int reference that is 0 on the initial function call, and by the
 *         end should hold the total number of pruned nodes.
 */
void pruneOddSubtrees(Node*& root, int& nPruned);
```

Examples:



SUNet: @stanford.edu

Complete the function as described on the previous page.

```
void pruneOddSubtrees(Node*& root, int& nPruned) {

}
```

4. **Graph Class (20pts).** In this problem, you will write a very simple Graph class. The implementation will be a *weighted, directed* graph using an *adjacency matrix* stored as a `Grid<int>`.

- The N vertices of the graph are represented by integers from 0 to N – 1.
- The layout of the Grid must work as follows: the value at row vertex1 and column vertex2 (i.e., in the Grid at [vertex1][vertex2]) is the weight of the edge **from** vertex1 **to** vertex2, or zero if there is no edge.
- The addEdge function will not add an edge from a vertex to itself, so Grid entries at [vertex][vertex] will always be zero.

More explanation of what this means and how the class will work in the contents of `finalgraph.h`, below. On the following page, you are asked to implement some of these functions as you would in `finalgraph.cpp`.

```
class FinalGraph {
public:
    /* Constructor takes the number of vertices to be in the graph. Sets up the Grid with
     * nVertex rows and nVertex columns, and all entries set to 0.
     * @param nVertex must be greater than zero
     */
    FinalGraph(int nVertex);

    /* Adds an edge from vertex start to vertex end, with given weight. If the edge is already
     * present, changes weight to given weight value. If start or end does not exist (is
     * not an int between 0 and N - 1), throws an exception. If start is equal to end,
     * returns without doing anything.
     * @param weight must be greater than zero (implementation will assume it is)
     */
    void addEdge(int start, int end, int weight);

    /* Returns a set containing all the vertices v such that there is an edge FROM the given
     * vertex TO v. The set will not include vertex itself (since self-loop edges are not
     * allowed). If vertex does not exist (is not an int between 0 and N - 1), throws an
     * exception.
     */
    Set<int> getNeighbors(int vertex);

    /* If a path *of length exactly 2 edges* exists from vertex start to vertex end, returns
     * the sum of the two edges of that path (if more than one such path, choose any). If
     * start or end does not exist (is not an int between 0 and N - 1), throws an exception.
     * If there is no 2-edge path from start to end, returns -1.
     */
    int twoEdgePathWeight(int start, int end);

private:
    /* Helper function to check if a vertex exists (is a value between 0 and N – 1, inclusive).
     * If not, throws an exception with error message "Invalid vertex".
     */
    void assertExists(int vertex);

    Grid<int> _matrix;
};
```

SUNet: @stanford.edu

```
/* (3pts) Helper function to check if a vertex exists (is a value between 0 and N – 1,
 * inclusive). If not, throws an exception with error message "Invalid vertex".
 * NOTE: use this any time you need to check that a vertex exists. */
void FinalGraph::assertExists(int vertex) {







}
```

```
/* (4pts) Adds an edge from vertex start to vertex end, with given weight. If the edge is
 * already present, changes weight to given weight value. If start or end does not exist (is
 * not an int between 0 and N - 1), throws an exception. If start is equal to end, returns
 * without doing anything.
 * @param weight must be greater than zero (implementation will assume it is) */
void FinalGraph::addEdge(int start, int end, int weight) {











}
```

```
/* (6pts) Returns a set containing all the vertices v such that there is an edge FROM the
 * given vertex TO v. The set will not include vertex itself (since self-loop edges are not
 * allowed). If vertex does not exist (not an int between 0 and N - 1), throws an exception.*/
Set<int> FinalGraph::getNeighbors(int vertex) {
















}
```

```
/* (7pts) If a path *of length exactly 2 edges* exists from vertex start to vertex end,
 * returns the sum of the two edges of that path (if more than one such path, choose any).
 * If start or end does not exist (is not an int between 0 and N - 1), throws an exception.
 * If there is no 2-edge path from start to end, returns -1.
 * IMPORTANT NOTE: because path length is limited, there are ways to approach it that don't
 * run a full-blown traditional DFS or BFS. Do something much simpler. You'll want to use
 * the getNeighbors() function as a helper. */
int FinalGraph::twoEdgePathWeight(int start, int end) {

}
```

SUNet: @stanford.edu

---

5. Sorting (6pts). Below is the code for two sorting algorithms, exactly as we saw them in lecture.

Selection Sort

```
void sort(Vector<int>& vec) {
  int n = vec.size();
  // already-fully-sorted section grows
  // 1 at a time from left to right
  for (int lh = 0; lh < n; lh++) {
    int rh = lh;
    // find the min element in the
    // entire unsorted section
    for (int i = lh + 1; i < n; i++) {
       // found new min?
       if (vec[i] < vec[rh]) rh = i;
    }
    // swap min into sorted section
    int tmp = vec[lh];
    vec[lh] = vec[rh];
    vec[rh] = tmp;
  }
}
```

Insertion Sort

```
void sort(Vector<int>& vec) {
  int n = vec.size();
  // already-sorted section grows 1 at a
  // time from left to right
  for (int i = 1; i < n; i++) {
    int j = i;
    // does this item needs to move
    // left to be in order?
    while (j > 0 && vec[j-1] > vec[j]) {
       // keep swapping this item with
       // its left neighbor if it is
       // smaller than the left neighbor
        int tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
       j--;
    }
  }
}
```

(a) (2pts) Of these two algorithms, which would you choose if you knew that your input data would often already be in sorted order?

○ Selection sort would have better Big-O time cost

○ Insertion sort would have better Big-O time cost

○ Either—relevant Big-O time costs equal

(b) (2pts) Of these two algorithms, which would you choose if you knew that your input data would often be in sorted—in the opposite (descending)—order?

○ Selection sort would have better Big-O time cost

○ Insertion sort would have better Big-O time cost

○ Either—relevant Big-O time costs equal

(c) (2pts) Briefly explain your responses to (a) and (b), including stating what the Big-O cost(s) would be.