Practice Final Exam Solutions

Your name: CS106B Rockstar

Sunet ID: cs106b-rockstar@stanford.edu

Section leader: Don't have one

Honor Code: I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I am submitting are my own work. I agree not to talk about the exam contents to anyone until a solution key is posted by the instructor.

Signature: _____ ← YOU MUST SIGN HERE!

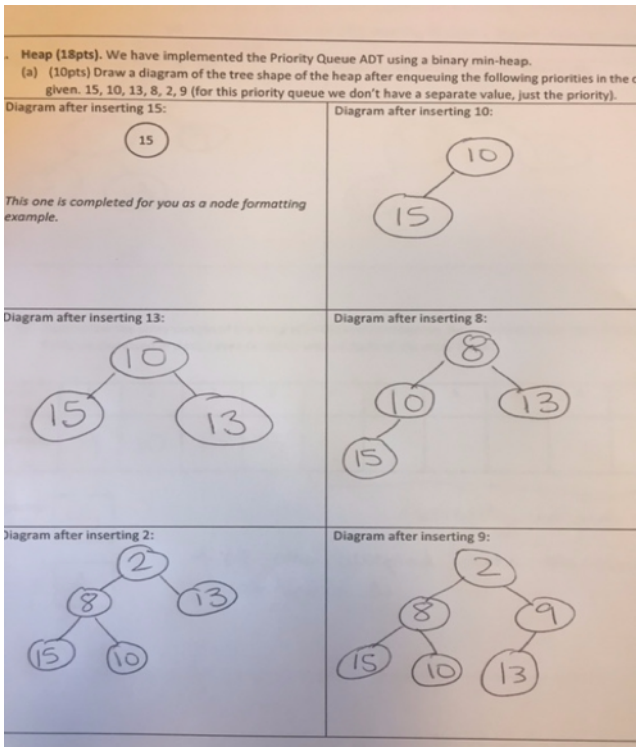
Rules: (same as posted previously to class web site)

- This exam is to be completed by each student individually, with no assistance from other students.
- You have 2 hours (120 minutes) to complete this exam.
- This test is **closed-book, closed-notes**. You may only have one 8.5x11" double-sided sheet of notes.
- You may not use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- On code-writing problems, you do not need to write a complete program, prototypes, nor #include statements. Write only the code (function, etc.) specified in the problem statement.
- Unless otherwise specified, you can write helper functions to implement the required behavior. When asked to write a function, do not declare any global variables.
- Do not abbreviate code, such as writing ditto ("") or dot-dot-dot marks (...). Pseudo-code will be given no credit.
- If you wrote your answer on a back page or attached paper, please label this clearly to the grader.
- Do NOT staple or otherwise insert new pages into the exam. Changing the order of pages in the exam confuses our automatic scanning system.
- **You should write your name at the top of every page in the exam.** You will get 1 point for this.
- **Tear off the last page (reference sheet)** before submission. You will get 1 point for this.
- Follow the Stanford Honor Code on this exam and correct/report anyone who does not do so.

#	Description	Earned	Max
0	Names and removing last page		2
1	Heap		14
2	Linked Lists		10
3	Recursive Backtracking		18
4	Graphs		20
5	Trees		26
	Total		**90**

Name: _____

1. Heap



Heap (18pts). We have implemented the Priority Queue ADT using a binary min-heap.

(a) (10pts) Draw a diagram of the tree shape of the heap after enqueuing the following priorities in the given. 15, 10, 13, 8, 2, 9 (for this priority queue we don't have a separate value, just the priority).

Diagram after inserting 15:

15

This one is completed for you as a node formatting example.

Diagram after inserting 10:

10, 15

Diagram after inserting 13:

10, 15, 13

Diagram after inserting 8:

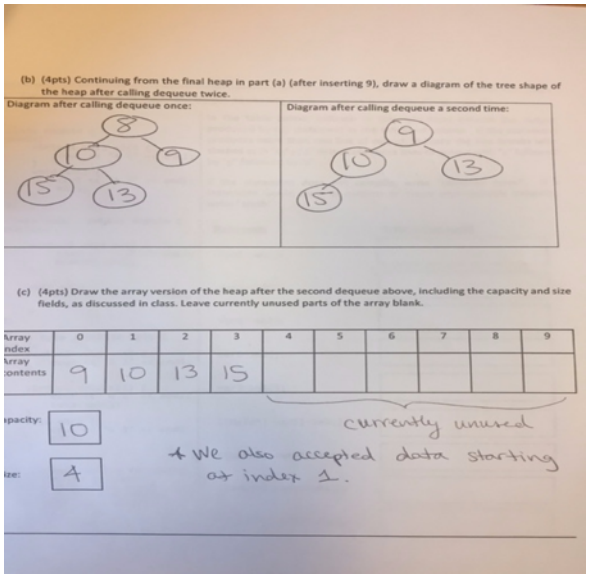
8, 10, 13, 15

Diagram after inserting 2:

2, 8, 13, 15, 10

Diagram after inserting 9:

2, 8, 9, 15, 10, 13



(b) (4pts) Continuing from the final heap in part (a) (after inserting 9), draw a diagram of the tree shape of the heap after calling dequeue twice.

Diagram after calling dequeue once:

8, 10, 9, 15, 13

Diagram after calling dequeue a second time:

9, 10, 13, 15

(c) (4pts) Draw the array version of the heap after the second dequeue above, including the capacity and size fields, as discussed in class. Leave currently unused parts of the array blank.

Array index	0	1	2	3	4	5	6	7	8	9
Array contents	9	10	13	15						

currently unused

capacity: 10

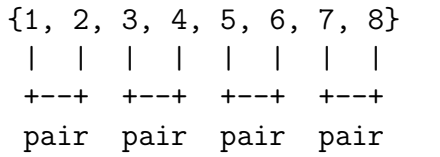
size: 4

+ We also accepted data starting at index 1.

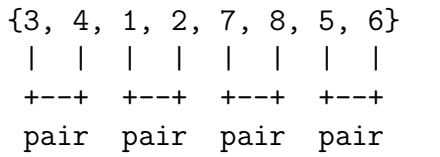
Name: _____

2. Linked Lists

Write a function `switchPairsOfPairs` that rearranges a linked list of integers by switching the order of each two neighboring pairs of elements in the sequence. Your function is passed a reference to a pointer to the front of the list as a parameter. Suppose, for example, that a variable named list points to the front of a list storing the following values:



This list has four pairs. If we make the call of `switchPairsOfPairs(front);` the list's state should become:



Notice that the pair (1, 2) has been switched with the pair (3, 4) and that the pair (5, 6) has been switched with the pair (7, 8). This example purposely used sequential integers to make the rearrangement clear, but you should not expect that the list will store sequential integers. It also might have extra values at the end that are not part of a group of four. Such values should not be moved. For example, if the list had stored this sequence of values:

{3, 8, 19, 42, 7, 26, 19, -8, 193, 204, 6, -4, 99, 2, 7}

Then a call on the function should modify the list to store the following. Note that 99, 2, 7 at the end are not switched:

{19, 42, 3, 8, 19, -8, 7, 26, 6, -4, 193, 204, 99, 2, 7}

Your function should work properly for a list of any size. Note: The goal of this problem is to modify the list by modifying pointers. It might be easier to solve it in other ways, such as by changing nodes' data values or by rebuilding an entirely new list, but such tactics are forbidden.

Constraints: For full credit, obey the following restrictions in your solution. A violating solution can get partial credit.

- Do not modify the data field of any existing nodes.
- Do not create any new nodes by calling `new ListNode(...)`. You may create as many `ListNode*` pointers as you like, though.
- Do not use any auxiliary data structures such as arrays, vectors, queues, maps, sets, strings, etc.

Name: _____

- Your code must make a single pass over the list (not multiple passes) and must run in no worse than O(N) time, where N is the length of the list.
- Note that the list has no "size" function, nor are you allowed to loop over the whole list to count its size.

Recall the `ListNode` structure:

```
struct ListNode {
    int data;
    ListNode* next;
};
```

Name: _____

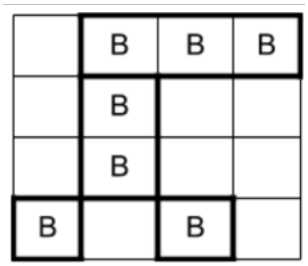
2. Linked Lists (Writing Space)

```
void switchPairsOfPairs(ListNode*& front) {
    if (front && front->next && front->next->next && front->next->next->next) {
        // have: front -> 1 -> 2 -> 3 -> 4 -> 5 ...
        // want: front -> 3 -> 4 -> 1 -> 2 -> 5 ...
        ListNode* curr = front->next->next; // curr -> 3
        front->next->next = curr->next->next; // 2 -> 5
        curr->next->next = front; // 4 -> 1
        front = curr; // front -> 3
        curr = curr->next->next->next; // curr -> 2

        // have: curr -> 2 -> 5 -> 6 -> 7 -> 8 -> 9 ...
        // want: curr -> 2 -> 7 -> 8 -> 5 -> 6 -> 9 ...

        while (curr->next && curr->next->next && curr->next->next->next
            && curr->next->next->next->next) {
            ListNode* temp = curr->next->next->next; // temp -> 7
            curr->next->next->next = temp->next->next; // 6 -> 9
            temp->next->next = curr->next; // 8 -> 5
            curr->next = temp; // 2 -> 7
            curr = temp->next->next->next; // curr -> 6
        }
    }
}
```

Name: _____

3. Recursive Backtracking *(18 points)*

The game of Battleship is a time-honored competition amongst friends. Each person has a board (which well represent with a Grid) where they secretly place several ships (1xN rectangles) so that they do not overlap other ships or go off the board. The picture to the right is an example of a Grid with 4 ships placed on it: size 3 (placed horizontally), size 2 (placed vertically), and two of size 1. Each cell with B represents part of a ship, and complete ships are outlined in black.



To win, you try to sink your friends ship by naming a row/col location to target with a cannon. Your friend self-reports whether you hit on a part of one of their ships or not (miss). If the locations you name result in many consecutive misses, you might begin to wonder whether your opponent is cheating in their self-reporting! So you decide to write a backtracking recursion program to determine whether theres any legal way to place the ships that avoids all the locations youve targeted so far.

Part a ***(6 points)***

First, youll need a **(non-recursive)** helper function `placeHoriz` that attempts to place one ship on the board in a specified location. We represent your friends board (from your perspective) as a `Grid<char>`, where ? represents a spot you know nothing about and M represents a location you have already targeted (and that your friend said was a miss). The function takes the current Grid, the length of the ship, and a row-col where you should place the ship. As the functions name suggests, you should try to place the ship horizontally on the board with the leftmost part of the ship at row and col. If the ship fits (does not overlap any M or B cells, and stays in bounds of the board), fill in the designated cells with B (to indicate a tentative guess at a possible ship placement) and return true. Otherwise return false. If your function returns false, no changes should be made to the board. You may assume that the provided row and col are in bounds of the Grid (though the ship might go out of bounds from there).

Write your answer on the following page.

Name: _____

3. Part a (Writing Space)

```
bool placeHoriz(Grid<char>& board, int size, int row, int col){
    for (int i = 0; i < size; i++) {
        if (!board.inBounds(row, col + i) || board[row][col + i] != '?') {
            return false;
        }
    }
    for (int i = 0; i < size; i++) {
        if (board.inBounds(row, col + i)) {
            board[row][col + i] = 'B';
        }
    }
    return true;
}
```

Name: _____

3. Part b ***(12 points)***

Now write a **recursive backtracking** function `canPlaceShips` that checks if a collection of ships can all be placed on the board such that they do not overlap each other or any cell marked M. The collection of ships is provided as a `Vector<int>` of sizes, where each int represents one ships size. The function returns true if its possible to place all the ships on the board, and false otherwise.

- Example: It would be impossible to place four ships of sizes 3, 2, 1, and 1 in any configuration on the before example board in part (a)something fishy (ha) is going on with your friends self-reporting!so you would return false in that case.
- Youll want to use your `placeHoriz` helper function, and you may assume a corresponding `placeVert` also exists, which does the same except that it places the ship vertically.
- You may also assume you have helpers and `unplaceHoriz` and `unplaceVert`, which remove a ship of size `size` from the specified location by writing ? in all its cells (the unplace functions have the same input parameter list as the place functions, but void return type).
- Your function should use backtracking recursion. Your code is not required to have any particular Big-O cost, but you may lose points if your code is extremely inefficient, such as exploring obviously invalid paths rather than stopping and backtracking.

Write your answer on the following page.

Name: _____

3. Part b (Writing Space)

```
bool canPlaceShips(Grid<char> & board, Vector<int> shipSizes) {
    if (shipSizes.size() == 0) {
        return true;
    }
    int shipSize = shipSizes[0];
    shipSizes.remove(0);
    for (int row = 0; row < board.numRows(); row++) {
        for (int col = 0; col < board.numCols(); col++) {
            if (placeHoriz(board, shipSize, row, col)) {
                if (canPlaceShips(board, shipSizes)) {
                    return true;
                }
                unplaceHoriz(board, shipSize, row, col);
            }
            if (placeVert(board, shipSize, row, col)) {
                if (canPlaceShips(board, shipSizes)) {
                    return true;
                }
                unplaceVert(board, shipSize, row, col);
            }
        }
    }
    return false;
}
```

Name: _____

4. Graphs *(20 points)*

Noticing the popularity of the Stanford Marriage Pact, Stanfords housing office has decided to use a similar algorithm to match roommates next year. They want to force students currently in Roble and Wilbur to mix next year, so each roommate pair must include one student from each. Based on a proprietary deep learning algorithm that examines your social media and grades (serious privacy issues in this hypothetical!), ResEd has created a weighted, directed graph of all students in R[oble] and W[ilbur], where an en edge exists from each student in R to each student in W, with weight indicating how much the R student should want to match with the W student. Corresponding reverse edges and weights exist from W students to R students. Smaller weights indicate more favorable pairing. You will write a function to help ResEd create roommate matches!

Part a *(5 points)*

Before we tackle the main algorithm, write a helper function that takes a `Vector` of the roommate ratings and returns a Map from Wilbur students to a `PriorityQueue` of Roble students, where the priority is the weight of the edge from the W student to that R student. To do so, you have will need the following structs (you may assume that comparison operators exist for both):

```
struct Student {
    string name;
    string dorm; // either Wilbur or Roble
};

struct Preference {
    Student *start;
    Student *end;
    int ranking;
};
```

Write your answer on the following page.

Name: _____

4. Part a (Writing Space)

```
Map<Student*, PriorityQueue<Student*>> getWPrefs(Vector<Preference *> &preferences) {
  Map<Student*, PriorityQueue<Student*>> wPrefs;
  for (Preference *preference : preferences) {
    if (preference->start->dorm == "Wilbur") {
      wPrefs[preference->start].enqueue(preference->end, preference->ranking);
    }
  }
  return wPrefs;
}
```

Name: _____

4. Part b (15 points)

Now we will write the main SMP algorithm. First, call the helper from (a) to get a Map `wPrefs`. Also create an empty `Set<Student*> isMatched` to track which W students already have matches, and an empty `Map<Student*, Student*>` matches to map each R student to their current W student match.

The algorithm repeats the following actions in a loop until all students have a match (i.e., all `wPrefs` keys are also in `isMatched`):

- Loop over wPrefs keys and, for each W student that is not already matched, attempt to match them with the next highest-priority roommate for them (next in Ws PriorityQueue).
- A match attempt will succeed if the R student is currently not matched to anyone.
- A match attempt will also succeed if the R student is currently matched to someone else, but the R student would prefer this new W student to their current match (the weight of the edge R-¿W is less than the weight of the edge R-¿[current match]).
- To perform a match, update `isMatched` and matches accordingly, including breaking the previous match if the R student had one.
- If the first match attempt for W fails, W stays unmatched this round. Continue the loop to the next W student in `wPrefs`.
- Return the matches.

Write your answer on the following page.

Name: _____

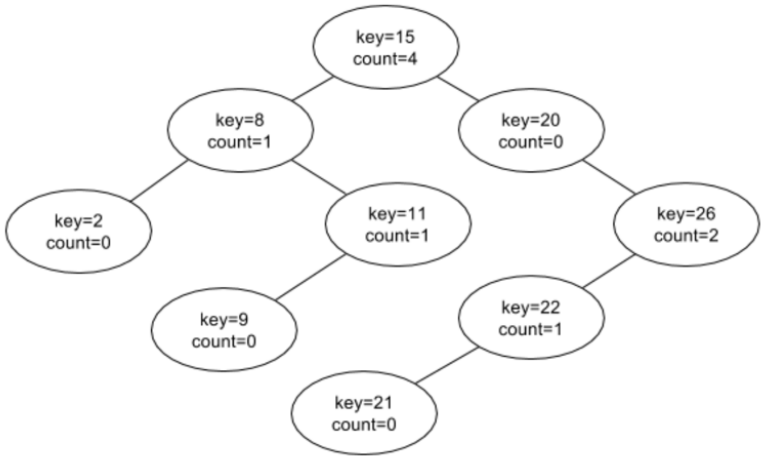
4. Part b (Writing Space)

```
Map<Student*, Student*> matchRoommates(Vector<Preference *> &preferences) {
    Map<Student*, PriorityQueue<Student*>> wPrefs = getWPrefs(preferences);
    Set<Student*> isMatched; // Wilbur students that are currently matched
    Map<Student*, Student*> matches; // from Roble to Wilbur

    while (isMatched.size() < wPrefs.size()) {
        for (Student *w: wPrefs) {
            if (!isMatched.contains(w)) {
                Student *r = wPrefs[w].dequeue(); // get potential roommate
                if (!matches.containsKey(r)) { // roommate is unmatched
                    matches[r] = w;
                    isMatched.add(w);
                } else {
                    Preference *currentMatch = nullptr;
                    Preference *potentialMatch = nullptr;
                    for (Preference *preference : preferences) {
                        if (preference->start == r && preference->end == w) {
                            potentialMatch = preference;
                        }
                        if (preference->start == r && preference->end == matches[r]) {
                            currentMatch = preference;
                        }
                    }

                    if (currentMatch->ranking > potentialMatch->ranking) {
                        // r prefers w to current match
                        isMatched.remove(matches[r]);
                        matches[r] = w;
                        isMatched.add(w);
                    }
                }
            }
        }
    }

    return matches;
}
```

Name: _____

5. Trees *(26 points)*

A k-ordered statistic tree is a Binary Search Tree where each node has an additional field that stores the number of nodes in its left subtree. The k-ordered statistic tree can use this information to quickly locate the kth element in the tree (kth if all elements were listed in ascending sorted order). We will use this to implement a Set ADT, so keys in the tree are all unique. Below is a valid k-ordered statistic tree. Notice the keys follow the usual BST ordering.



The file `korder.h` is as follows (do not edit this code):

```
struct Node {
    Node(int key) { this->key = key; count = 0; left = right = NULL; }
    int key; // the usual BST key
    int count; // count of nodes in left subtree
    Node* left; // left child
    Node* right; // right child
};


class KTree {
public:
    KTree();
    ~KTree();
    void addKey(int key);
int getKthKey(int k);

private:
    Node* root;
};
```

Name: _____

Functions in the korder.cpp file are shown below and on the following 2 pages. Complete the code for them. You are welcome to add additional helper function(s) if you want (do not add them to the class in .h file, just add them below).

```
// Constructor (2pts)
KTree::KTree()
{

    root = NULL; // or this->root = NULL;

}

// Destructor (6pts)
KTree::~KTree()
{

    deleteHelper(root); // they add this, can be named anything

}

deleteHelper(Node* curr) // they add this, can be named anything
{
    if (curr != NULL) {
        deleteHelper(curr->left);
        deleteHelper(curr->right);
        delete curr;
    }
}


// Inserts key into the tree in the proper place, and updates all tree
// counts appropriately. Your solution must be recursive, using the
// provided helper. (7pts)
void KTree::addKey(int key)
{
    if (root == NULL) {
       root = new Node(key);
    } else {
       addKeyHelper(key, root);
    }
}

// Recursive helper function for addKey. Returns true if node was added, false
// if key was duplicate so no add was done. The code for a standard BST insert
// is already provided here for you. You should edit this code to make it
// work for k-ordered tree. Write your additional line(s) of code to the right
```

Name: _____

```
// and use arrows to indicate where to insert your addition(s). Cross out any
// code you want to delete.

bool addKeyHelper(int key, Node* curr)
{
    if (key < curr->key) {
        if (curr->left == NULL) {
            curr->left = new Node(key);
            curr->count++;
            return true;
        } else {
            if (addKeyHelper(key, curr->left)) {
                curr->count++;
                return true;
            } else {
                return false;
            }
        }
    } else if (key > curr->key) {
        if (curr->right == NULL) {
            curr->right = new Node(key);
            return true;
        } else {
            return addKeyHelper(curr->right);
        }
    } else {
        return false;
    }
}

// 3 possible solutions for kthKey

// O(logN) (full credit)
int kthKeyHelper(int k, Node* curr)
{
   if (k == curr->count) {
       return curr->key;
   }
   if (k < curr->count) {
       return kthKeyHelper(k, curr->left);
   }
   if (k > curr->count) {
       return kthKeyHelper(k - curr->count - 1, curr->right);
   }
}
```

Name: _____

```
// O(N) no aux data structs (small deduction)
void kthKeyHelper(int k, Node* curr, int& countSoFar, int& retVal){
   if (curr != NULL) {
       kthKeyHelper(k, curr->left, countSoFar, retval);
       if (k == countSoFar) {
           retVal = curr->key;
       }
       countSoFar++;
       kthKeyHelper(k, curr->right, countSoFar, retVal);
   }
   return 0;
}

// O(N) with aux data structs (larger deductionin this version the wrapper
// will receive the Vector populated with keys of the tree, and then just
// pull out nums[k])
void kthKeyHelper(int k, Node* curr, Vector<int>& nums){
   if (curr != NULL) {
       kthKeyHelper(k, curr->left, countSoFar);
       nums.add(curr->key);
       // "optimization" if (nums.size() < k)
       kthKeyHelper(k, curr->right);
   }
   return 0;
}
```