CS106B Instructor: Cynthia Lee

Autumn 2019

FINAL EXAM

---

1. 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 order 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:  *This one is completed for you as a node formatting example.*	Diagram after inserting 10:
Diagram after inserting 13:	Diagram after inserting 8:

Diagram after inserting 2:	Diagram after inserting 9:

(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:	Diagram after calling dequeue a second time:

(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										

Capacity: ☐

Size: ☐

2. **Pointers and Memory (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 (outer box vs. inner 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, and do not change/remove any pointer arrows or other values unless they are actually changed.

```
Slayer* vamp = new Slayer[2];
vamp[1].angel = 2;
vamp[1].buffy = new Slayer;
vamp[0].buffy = vamp;
vamp[0].angel = 3;
delete vamp[1].buffy;
vamp[1].buffy = NULL;
//Draw the state of memory now
```

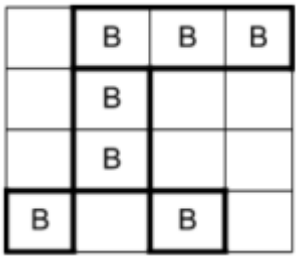
```
struct Slayer {
   int angel;
   Slayer * buffy;
};
```

DRAWING:

Stack:

Heap:

3. Recursive Backtracking (18pts). The game of Battleship is a time-honored competition amongst friends. Each person has a board (which we'll 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 friend's 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 there's any legal way to place the ships that avoids all the locations you've targeted so far.

(a) (6pts) First, you'll need a helper function `placeHoriz` that attempts to place one ship on the board in a specified location. We represent your friend's board (from your perspective) as a `Grid<char>`, where '?' represents a spot you know nothing about, 'M' represents a location you have already targeted (and that your friend said was a miss), and 'B' represents a placed ship (placed tentatively as part of backtracking exploration). The function takes the current Grid, the length of the ship, and a row-col where you should place the ship. As the function's name suggests, you should try to place the ship <u>horizontally</u> on the board with the leftmost part of the ship at row and col. If the ship fits (<u>does not overlap any 'M' or 'B' cells, and stays in bounds of the board</u>), 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.

Before:

?	M	?	?
?	?	M	?
M	?	?	M
?	M	?	?

After (with size=2, row=2, col=1):

?	M	?	?
?	?	M	?
M	B	B	M
?	M	?	?

```
bool placeHoriz (Grid<char>& board, int size, int row, int col){

}
```

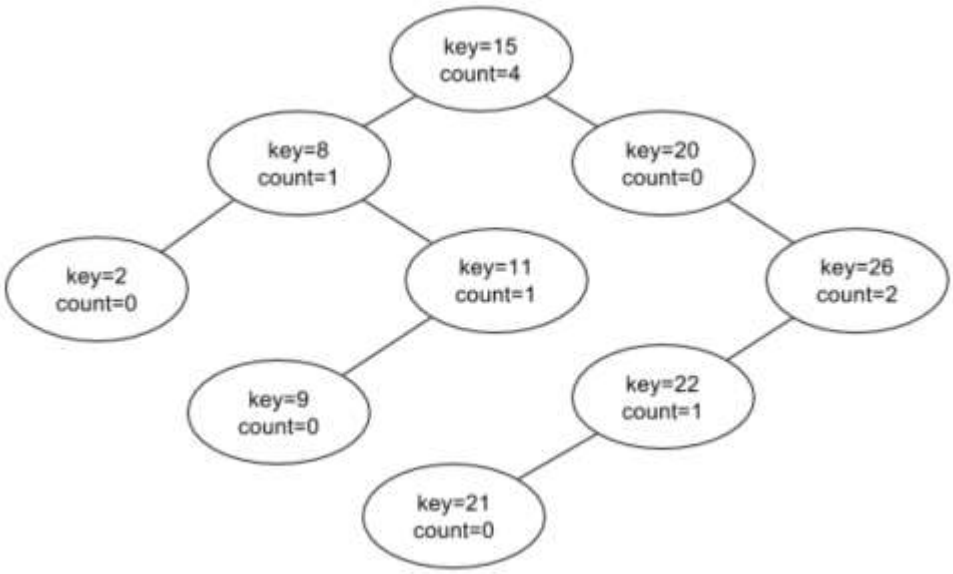
(b) (12pts) Now write a recursive backtracking function `canPlaceShips` that checks if a collection of ships can all be placed on the board such that they do not overlap each other or any cell marked 'M.' The collection of ships is provided as a `Vector<int>` of sizes, where each `int` represents one ship's size. The function returns true if it's 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 friend's self-reporting!—so you would return false in that case.
- You'll 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.

```
bool canPlaceShips(Grid<char>& board, Vector<int> shipSizes){

}
```

4. **Trees (26pts).** 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.
    - **Example:** 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;
};
```

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).

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




}

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




}
```

```
// (7pts) Inserts key into the tree in the proper place, and updates all tree
// counts appropriately. Your solution must be recursive, using the provided
// helper.
void KTree::addKey(int key)
{









}

// 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
// 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);
                  return true;
            } else {
                  return addKeyHelper(curr->left);
            }
      } else if (key > curr->key) {
            if (curr->right == NULL) {
                  curr->right = new Node(key);
                  return true;
            } else {
                  return addKeyHelper(curr->right);
            }
      } else {
            return false;
      }
}
```

```
// (11pts) Returns the kth smallest key in the tree (numbered starting
// with 0). In the example tree above, for k=0 return 2; k=3 return 11; k=5
// return 20. You may assume that is valid (0 <= k < N for tree with N nodes).
// Your solution must be recursive (see recursive helper below). For full
// credit, your solution must be O(logN). As a fallback option, O(N) solutions
// that do not use auxiliary data structures will incur a small 2pt deduction.
int KTree::getKthKey(int k)
{









}

// Recursive helper for getKthKey(). For the O(N) solution, you may add
// argument(s) and change the return value if you like. For the O(logN)
// solution, this is the best header.
int kthKeyHelper(int k, Node* curr                                        )
{









}
```