CS106B Instructor: Cynthia Lee

Autumn 2019 Practice Exam

FINAL EXAM

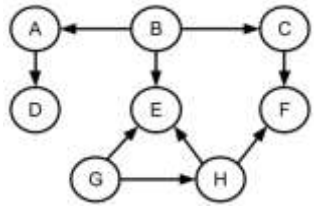
---

1. **Graphs.** You are given a directed, unweighted graph as an NxN dimension `Grid<bool>`, where the vertices are numbered 0 to N-1, and `Grid[row][col]` is `true` when there is an edge from vertex `row` to vertex `col`. The graph is not necessarily connected. Your task is to find subsets of the vertices in the graph that are actually a tree *(for a special definition of tree, given below)*. The function signature should be as follows:

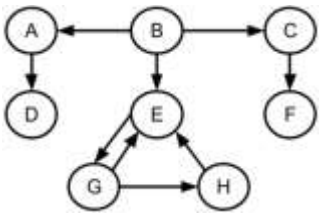
   ```
   int findLargestTree(Grid<bool>& graph)
   ```

   - **`graph`** is the input graph. You should not change its structure (vertex set, edges).
   - **Return the index of a graph vertex that is the root of the largest tree in the graph**. If there is no such vertex in the graph (no valid trees), then return **`-1`**. If there is a tie for largest tree, return either.
   - You may want to create a helper function with parameters that hold additional information about the vertices during your algorithm.
   - **What is a tree?** A tree in this case is defined as follows:
     - A tree consists of **a root node** (one vertex) and all vertices reachable from that root.
     - We interpret outgoing edges from a graph node as basically children of that node when we think of it as a tree (so nodes can have any number of children).
     - No vertex in the tree may be reachable by more than one path from the root (no cycles).

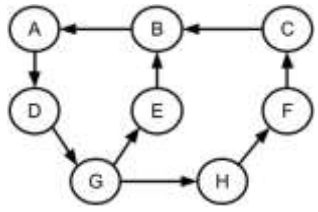
   Examples:

   Return a pointer to Vertex B.
   Vertex B is the root of a tree that includes vertices {A, B, C, D, E, F}. Note that G is **not** even under consideration by size, because it is not the root of a valid tree (E is reachable by more than one path: G->E and G->H->E).

   Return a pointer to Vertex A *or* Vertex C.
   Note that Vertex B is **not** the root of a valid tree because nodes reachable from it include a cycle.

   Answer: Return NULL.

```
Vertex* findLargestTree(BasicGraph& graph) {

}
```

2. **Pointers and Linked Lists.** Write the **`contains`** function that given two linked lists will determine whether the second list is a subsequence of the first. To be a subsequence, every value of the second must appear within the first list and in the same order, but there may be additional values interspersed in the first list. A list contains itself; the NULL list is contained in any list.

Examples:

list	sub	contains(list, sub)
1 -> 4 -> 2 -> 9	1 -> 4	true
1 -> 4 -> 2 -> 9	9 -> 4	false
1 -> 4 -> 2 -> 9	4 -> 9	true
1 -> 4 -> 2 -> 9	1 -> 1 -> 4	false
1 -> 4 -> 2 -> 9	2 -> 9 -> 10	false

```
struct listnode {
   int val;
   listnode * next;
};

bool contains(listnode * list, listnode * sub) {




}
```

3. **Recursion.** A ternary (or 3-ary) tree is one where each node has up to three children. A node for this tree is defined as:

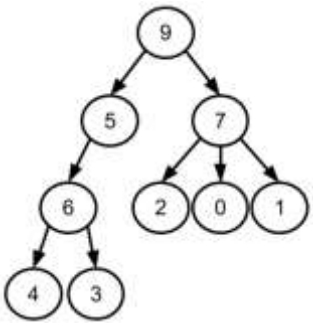
```
struct treenode {
   treenode(int k, treenode * l, treenode * m, treenode * r) {
       key= k; left = l; middle = m; right = r;
   };
   int key;
   treenode * left;
   treenode * middle;
   treenode * right;
};
```

Given a ternary tree, identify the subset of its nodes with the maximum sum of their **key** fields, subject to the constraint that no pair of the selected nodes may be ancestors/descendants of each other. Your function signature should be as follows:

```
Set<int> maxSumSubset (treenode * root);
```

- As the title of this problem suggests, you should solve this using recursion.
- You may assume that all keys in the tree are unique and non-negative.
- Unlike Heaps or BSTs, the keys have no ordering rule governing them.

Example:



Return a Set containing {4, 3, 7}.

Some of the reasoning involved is as follows: We would rather choose 4 and 3 than their parent 6 because 4+3>6. We would rather choose 7 than its children 2+0+1.

Write your solution on the next page. As usual, you are free to write helper function(s).

```
Set<int> maxSumSubset (treenode* root) {

}
```

4. **BSTs and Heaps.**

(a) We have implemented the Map ADT using a plain Binary Search Tree (no special balancing measures). Our node structure is defined as "`struct node { int key; int value; node* left; node* right; };`" Draw a diagram of the BST that results from inserting the following (key, value) pairs in the order given. Please label each BST node with **both the key and value**. (25,2), (17,1), (29,0), (55,1), (45,7), (29,3)

Diagram after inserting (25,2):  *This one is completed for you.*	Diagram after inserting (17,1):
Diagram after inserting (29,0):	Diagram after inserting (55,1):
Diagram after inserting (45,7):	Diagram after inserting (29,3):

(b) We have implemented the Priority Queue ADT using a binary min-heap. Draw a diagram of the heap's tree structure that results from inserting the following priority values in the order given: 25, 37, 28, 12, 30, 3

Diagram after inserting 25:  *This one is completed for you.*	Diagram after inserting 37:
Diagram after inserting 28:	Diagram after inserting 12:
Diagram after inserting 30:	Diagram after inserting 3:

5. **Pointers and Memory.** 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 values by writing NULL or drawing a slash through the box.
    - Draw the components in the appropriate stack and heap areas marked for you.
    - Draw struct fields adjacent to each other (held in one subdivided box), and label each field with the field name for clarity.

```
int eleven = 11;
int season = 2;
Things* stranger = new Things;
stranger->el = eleven;
stranger->barb = &season;
stranger->joyce = new Things;
stranger->joyce->el = stranger->el;
stranger->joyce->barb = stranger->barb;
int* eggos = new int[3];
eggos[2] = eleven;
```

```
struct Things {
   int el;
   int* barb;
   Things* joyce;
};
```

DRAWING:

Stack:

Heap:

6. **Algorithms.** When asked for Big-O analysis, give a tight bound of the nearest runtime complexity class.

(a) The function **Binky** is defined as follows:

```
int Binky(int n)
{
    if (n <= 1) return 1;
    if ((n % 2) == 0)
       return 2 * Binky(n/2);
    else
       return Binky(n + 1);
}
```

Give the computational complexity of **Binky** expressed in big-O notation, where *N* is the value of the argument **n**, assumed to be a nonnegative integer. Briefly justify your answer.

(b) Removing a cell from a singly-linked list typically requires not only a pointer to the cell but also to its previous cell. You propose getting around this by overwriting the contents of the cell with the value of the cell that follows and then deleting the following cell instead when you need to delete a cell. Using this idea, you've written a new version of **deleteCell**:

```
void deleteCell(Cell *ptr) {
   Cell *toDelete = ptr->next;
   *(ptr) = *(ptr->next); // struct assignment copies over all fields
   delete toDelete;
}
```

Does this strategy work? YES NO (circle) Briefly explain why or why not:

(c) Given an unsorted input of N integers, you wish to print the median element. Assume N is odd. Three different algorithms are proposed to finding the median. Each is correct, but they have different performance profiles

1. Run the first N/2 passes of **`SelectionSort`** and print the lastmost element swapped.

2. Sort the array using **`MergeSort`** and print out the middlemost element.

3. Follow this algorithm starting with K = N/2 + 1. Choose the first element as the pivot and use the **`Quicksort`** partition function to divide into a left (all elements smaller than pivot) and a right section (all elements larger than pivot). Let L be the number of elements in left section. If L = K, print the pivot and you're done. If K < L, recursively apply algorithm to find Kth element in left section. If K > L, recursively apply the algorithm to find the (K - L)th element within right section.

Give the big-O running time (tight bound) of each algorithm in the worst case.

	Worst-case big-O
1.	
2.	
3.	