Mid-Term 2: 2nd Practice Coding Questions

Directions

Read these directions carefully.

You will be solving two of the three programming problems detailed below. ***Problem 1*** is **mandatory**, but you only have to do **either** ***Problem 2*** **or** ***Problem 3*****.** For each problem you must submit three documents:

1. A C++ program that solves the given problem.
2. A text file that contains the output of your program when it is run.
3. A short document that contains any issues or concerns you have about the given problem, as well as any information that we need to understand, compile, or run your solution.

The text file you submit will help us understand your thought process. Mention anything you have done to write, test, and debug your code. Incomplete code can still receive points if you show that you have identified the errors and tried to debug them.

Your submission should be valid C++ 11 code. The solutions should use similar types and functions to those in the given starter code, but you are welcome to make modifications as you see fit. We will be running this code on other computers, so make sure to avoid **any** undefined behavior such as uninitialized variables.

PROBLEM 1 (MANDATORY)

Task:

Given 2 arrays, write a program to check if the two arrays are similar or not.

Requirements:

1. Implement a **checkSimilar** function:

```
bool checkSimilar(int* arr1, int* arr2); // example declaration
```

2. The function takes 2 arguments which are addresses of the 2 arrays **arr1** and **arr2** and return a boolean value indicating whether the arrays are similar or not.

3. Two arrays are said to similar if they have the same length and same set of elements but not necessarily in the same order.

4. **EXAMPLES:**

Example 1:

ARRAY 1

5	1	2	4	3

ARRAY 2

3	1	4	5	2

ARRAY 1 and *ARRAY 2* are **similar** since they both are of length 5 and they have the same set of elements {1, 2, 3, 4, 5}

Example 2:

ARRAY 1

3	5	1	2	4

ARRAY 2

5	6	3	2	1

ARRAY 1 and *ARRAY 2* are **not similar** since they have different set of elements. *ARRAY 1* has set of elements {1, 2, 3, 4, 5} and *ARRAY 2* has set of elements {1, 2, 3, 5, 6}

5. Write a main function that creates 2 arrays. Call the function **checkSimilar** by sending the address of the 2 arrays. Create arrays of different set of elements and differing lengths to show that it works in all cases, and print out the results.

6. **Test your function** to make sure that it works in every case.

PROBLEM 2

Task:

Given a binary tree, write a program to print out all the boundary nodes of the binary tree **in no particular order.**

Requirements:

1. Implement a **printBoundary** function

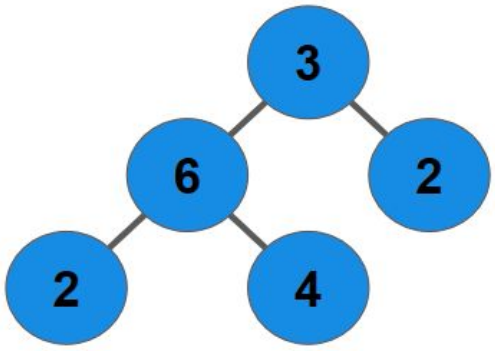
```
void  printBoundary(struct Node* root); //example declaration
```

2. The function takes the root node address of the tree as the argument. The functions prints out all the boundary nodes of the tree.

3. Boundary nodes of a tree are nodes which the leftmost and rightmost nodes in the tree

4. **EXAMPLES:**

Example 1:



In this tree, the boundary nodes are **3, 6, 2 and 2.**
The output expected is printing these elements in any order.
Example outputs are ***3, 6, 2, 2*** or ***3, 2, 6, 2*** or ***2, 3, 6, 2***.

5. Write the main function that creates a binary tree. Call the function **printBoundary** by passing the address of the root node of the tree. The **printBoundary** function prints the boundary nodes of the tree. Create different types of trees like *full binary tree, right skewed tree, left skewed tree, complete binary tree, trees where every node has only one child, with different tree heights* and verify if the function works in all cases, and print out the results.

6. **Test your function** to make sure that it works in every case irrespective of the height or type of the tree.

PROBLEM 3

Task:

Given an undirected graph, write a program to check if there is a cycle in the graph or not.

Requirements:

1. Implement a **checkCycle** function

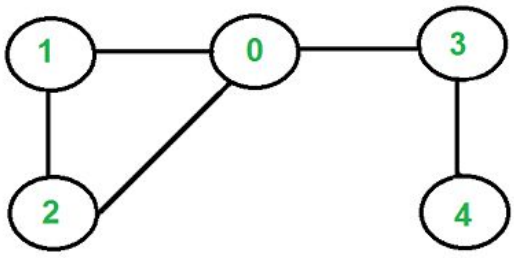
```
bool checkCycle(int v, int visited[], int parent); //example declaration
```

2. The function **checkCycle** is a recursive function and takes a vertex number, the visited array and the parent of the vertex 'v'.

3. **checkCycle** function returns ***True*** if there is a cycle present and returns ***False*** if there is no cycle.

4. **EXAMPLES**

Example 1:



For this graph, the **checkCycle** function returns **True** since there is a ***cycle 1, 0, 2.***

5. Write a main function that creates a simple tree as shown in the above image, then iterates through every vertex of the graph, then calls the function **checkCycle for each vertex**. Even if one of the checkCycle calls return True, that means that a cycle is present otherwise there's no cycle. Test this on different graph configurations to verify that it works on multiple graphs.

6. **Test your function** to make sure that it works in every case, no matter what the configuration of the graph is.