

CS 32 Solutions Week 9

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. Although exams are online this quarter, it is still in your best interest to practice these problems by hand and not rely on a compiler.

Solutions are written in red. The solutions for **programming problems** are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please contact your LA or go to any of the LA office hours.

Concepts: Hash Tables, Heaps

For this worksheet, “using a hash table” means you’re allowed to use a STL implementation of one (`unordered_map`, `unordered_set`).

1. Given a string, find the first non-repeating character in it and return its index. If every character repeats, return -1. You may assume the string contains only lowercase letters. Use a hash table to solve this problem.

Examples:

Input: `s = "leetcode"`

Output: `0`

Input: `s = "loveleetcode"`

Output: `2`

Solution

```
int firstUniqueChar(std::string s) {  
    // Map character to the frequency of occurrence  
    unordered_map<char, int> counter;  
    for(int i = 0; i < s.size(); i++) {  
        counter[s[i]]++;  
    }  
    for (int i = 0; i < s.size(); i++) {  
        if (counter[s[i]] == 1) return i;  
    }  
    return -1;  
}
```

```
}
```

2. Given an array of integers and a target sum, determine if there exists two integers in the array that can be added together to equal the sum.

The time complexity of your solution should be $O(N)$, where N is the number of elements in the array. In other words, the brute force method of comparing each element with every other element using nested for loops will not satisfy this requirement.

Examples:

Input: `arr[] = [4, 8, 3, 7, 9, 2, 5], target = 15`

Output: `true`

Explanation: 8 and 7 add up to the target sum 15

Input: `arr[] = [1, 3, 5, 2, 4], target = 10`

Output: `false`

Explanation: No combination of two numbers in the array sum to 10

```
bool twoSum(int arr[], int n, int target);
```

Solution

```
bool twoSum(const int arr[], int n, int target) {
    unordered_set<int> numsFound;

    // We will add every number to a set as we iterate
    // through the array. If our set ever contains the
    // 'complement' of the number we are looking at, we
    // have found a pair of numbers whose sum is the target
    // and we will return true. Otherwise, if we reach the
    // end of the array, return false.
    for (int i = 0; i < n; i++) {
        int complement = target - arr[i];
        if (numsFound.find(complement) != numsFound.end()) {
            return true;
        }
        else {
            numsFound.insert(arr[i]);
        }
    }
}
```

```

        }
    }
    return false;
}

```

3. Implement the following function:

```
bool isMaxHeap(const int arr[], int len);
```

This function takes in an array *arr* of length *len* and returns whether or not that array represents a binary max heap. In other words, *arr* must follow the max heap property, where a parent is greater than or equal to its children.

Solution

```

bool isMaxHeap(const int arr[], int len) {
    for (int x = 0; x < len; x++) {
        int left = 2 * x + 1;
        int right = left + 1;

        if (left < len && arr[left] > arr[x])
            return false;
        if (right < len && arr[right] > arr[x])
            return false;
    }
    return true;
}

```

4. Implement the following function, given the following data structure:

```

struct Node {
    int val;
    Node* left;
    Node* right;
};

```

```
bool isMinHeap(const Node* head);
```

This function takes in the head of a binary tree and returns whether or not that binary tree represents a binary min heap. In other words, this tree must follow the min heap property, where a parent is less than or equal to its children.

Solution

```
// Precondition: head points to a complete tree
bool isMinHeap(const Node* head) {
    if (head == nullptr)
        return true;

    Node* left = head->left;
    Node* right = head->right;
    if (left != nullptr && head->val > left->val)
        return false;
    if (right != nullptr && head->val > right->val)
        return false;

    return isMinHeap(head->left) && isMinHeap(head->right);
}
```

5. You are working at a credit card company and need to store account balances. Each account holder has an integer `userId` number. Each user/userid can have as many bank accounts as they want specified by an integer `accountId`. Write a class called `Bank` that supports insertion of a deposit and search of a given user and account id. Insert should update the balance of an account, if the given `accountId` and `userId` already exist. If it does not exist, it should create a new entry with the given parameters. Search should return the balance of the account, if the given `accountId` and `userId` exists. If it does not exist, the function must return `-1`. The company wants to process a high volume of transactions so they demand search and insertion work in $O(1)$ time, i.e they do not depend on the number of users or bank accounts. Hint: Consider an STL container in an STL container

```
class Bank {
public:
    void insert (int amount, int userId, int accountId);
```

```
int search (int userId, int accountId);
...
}
```

i.e

Bank B;

B.insert(10, 765, 937)

B.search(765, 937) *// returns 10*

Solution

```
class Bank {
public:
void insert(int amount, int userId, int accountId);
int search(int userId, int accountId);
Private:
    // userId -> {accountId -> account}
    unordered_map<int, unordered_map<int, int>> database;
};

void Bank::insert(int amount, int userId, int accountId) {
    auto userMap = database.find(userId);
    if(userMap == database.end()) {
        unordered_map<int, int> userAccountMap;
        userAccountMap.insert({accountId, amount});
        database.insert({userId, userAccountMap});
    } else {
        auto userAccountMap = userMap->second;
        //unspecified what to do when inserting multiple accounts with same id
        userAccountMap.insert({accountId, amount});
        //alternate way of interacting with unordered_map that updates and creates
        database[userId] = userAccountMap;
    }
}

int Bank::search(int userId, int accountId) {
    auto userMap = database.find(userId);
    if(userMap == database.end()) {
        return -1;
    }
    unordered_map<int, int> userAccountMap = userMap->second;
```

```

auto account = userAccountMap.find(accountId);
if(account != userAccountMap.end()) {
    return account->second;
} else {
    //unspecified what to do here
    return -1;
}
}

```

6. Given an array of n integers that is guaranteed to satisfy the max heap property, write a function that constructs a binary tree representing the same binary max heap as the array and returns its root.

```

struct Node {
    int val;
    Node* left;
    Node* right;
};

```

```

Node* makeMaxHeap(const int a[], int n);

```

Solution

```

Node* makeMaxHeap(const int a[], int n) {
    return makeMaxHeapHelper(a, n, 0);
}

Node* makeMaxHeapHelper(const int a[], int n, int i) {
    if (i >= n) {
        return nullptr;
    }
    Node* root = new Node;
    root->val = a[i];
    root->left = makeMaxHeapHelper(a, n, 2 * i + 1);
    root->right = makeMaxHeapHelper(a, n, 2 * i + 2);
    return root;
}

```

7. Given an array of distinct elements and a range [low, high], use a hash table to output all numbers in the range that are not in the array. Print the missing elements in sorted order.

Example:

Input: arr[] = {10, 12, 11, 15}, low = 10, high = 15

Output: 13, 14

Input: arr[] = {1, 14, 11, 51, 55}, low = 50, high = 55

Output: 50, 52, 53, 54

Solution

```
#include <unordered_set>
#include <iostream>
using namespace std;

void inRange(int arr[], int size, int low, int high)
{
    // Insert all elements of arr[] in set
    unordered_set<int> set;
    for (int i=0; i<size; i++)
        set.insert(arr[i]);

    // Traverse through the range and print all
    // missing elements
    for (int x=low; x<=high; x++)
        if (set.find(x) == set.end()) //or if (set.count(x) == 0)
            cout << x << " ";
}
```

We use unordered_set here because it stores unique values not in a particular order to allow fast retrieval of values.

```
#include <unordered_map>
#include <iostream>
using namespace std;

void inRange(int arr[], int size, int low, int high)
{
    // Insert all elements of arr[] in set
    unordered_map<int, bool> map;
    for (int i=0; i<size; i++)
```

```

        map[arr[i]]=true;

// Traverse through the range and print all
// missing elements
for (int x=low; x<=high; x++)
    if (map.find(x)== map.end())
        cout << x << " ";
}

//version with map

```

8. Write a function, `sum3`, that takes in an array of integers and determines whether there exists exactly three elements in the array that sum to 0. Return `true` if three such elements exist and `false` if not. No repeated elements are allowed. Your function must run faster than the brute force $O(N^3)$.

i.e `[1,2,3,4,5,6]` -> `False`
 `[1,-1,2,-2]` -> `False`
 `[1,2,-3, 6, 8]` -> `True`

```
bool sum3(const int arr[], int n);
```

Solution

```

#include <iostream>
#include <unordered_set>
using namespace std;

bool sum3(const int arr[], int n) {
    //create hash table
    unordered_set<int> hashedArr;
    for(int i = 0; i < n; i++){
        hashedArr.insert(arr[i]);
    }

    //search for opposite of every pair
    for(int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
            int oppSum = (arr[i] + arr[j])*-1;
            if(oppSum != arr[i] && oppSum != arr[j] && hashedArr.find(oppSum) !=
hashedArr.end()) {
                return true;
            }
        }
    }
    return false;
}

```

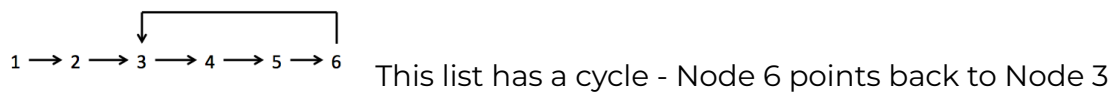


```

    }
}
return false;
}

```

9. Given a linked list, determine if it has a cycle in it. This can be done by starting from the head of the linked list and traversing it until you reach a node you have already seen or the end of the list. The time complexity of your solution should be $O(N)$ where N is the number of nodes in the list. Example:



Use the following Node definition and function header to get started:

```

struct Node {
    int val;
    Node* next;
}

```

```

bool hasCycle(const Node* head);

```

Solution

```

bool hasCycle(const Node* head) {
    unordered_set<const Node*> nodesSeen;
    Node* temp = head;

    while (temp != nullptr)
    {
        if (nodesSeen.find(temp) != nodesSeen.end()) {
            // we've seen this node already, there is a cycle!
            return true;
        }
        else {
            // add this node to the set of ones we've already seen
            nodesSeen.insert(temp);
            temp = temp->next;
        }
    }
}

```

```
    }  
}  
// we saw all nodes only once, so there is no cycle  
return false;  
}
```