# Hash

## Q1

Implement function

int foldShift(long long key, int addressSize);  
int rotation(long long key, int addressSize);

to hashing key using Fold shift or Rotation algorithm.

Review Fold shift:

The **folding method** for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| cout << rotation(600101, 2); | 26 |

int foldShift(long long key, int addressSize)

{

}

int rotation(long long key, int addressSize)

{

}

//REMEMBER: ĐẶT ÁC BIẾN TẠM CÙNG DATATYPE VỚI IMPUT VALUE (LONG LONG -> XÀI BIẾN TẠM LONG LONG (no int))

int foldShift(long long *key*, int *addressSize*)

{

    //special case:

    if(*key* == 0) return 0;

    //normal case:

    Long long res = 0;

    long long pow10 = 1;

    while(pow10 < *key*){

        pow10 \*= 10;

    }

    while(pow10 >1)

    {

        res += *key* / pow10;

*key*  = *key* % pow10;

        for(int i = 0; i < *addressSize*; i++) pow10 /= 10;

    }

    res += *key*;

    long long powSize = 1;

    for(int i = 0; i < *addressSize*; i++) powSize \*= 10;

    if(res >= powSize) return res%powSize;

    return res;

}

int rotation(long long *key*, int *addressSize*)

{

    //normal case:

    long long modVal = *key* % 10;

*key* = *key* / 10;

    long long pow10 = 1;

    while(pow10 < *key*) pow10 \*= 10;

*key* += modVal \* pow10;

    return foldShift(*key*, *addressSize*);

}

## Q2

Implement three following hashing function:

**long** **int** midSquare(**long** **int** seed);  
**long** **int** moduloDivision(**long** **int** seed, **long** **int** mod);  
**long** **int** digitExtraction(**long** **int** seed, **int**\* extractDigits, **int** size);

Note that:

In midSquare function: we eliminate 2 last digits and get the 4 next digits.

In digitExtraction: extractDigits is a sorted array from smallest to largest index of digit in seed (index starts from 0). The array has size **size.**

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int a[]={1,2,5};  cout << digitExtraction(122443,a,3); | 223 |
| cout <<midSquare(9452); | 3403 |

long int midSquare(long int seed)

{

}

long int moduloDivision(long int seed, long int mod)

{

}

long int digitExtraction(long int seed,int\* extractDigits,int size)

{

}

long int midSquare(long int *seed*)

{

    return (*seed*\**seed* / 100) % 10000;

}

long int moduloDivision(long int *seed*, long int *mod*)

{

    return *seed* % *mod*;

}

long int digitExtraction(long int *seed*,int\* *extractDigits*,int *size*)

{

*string* str = to\_string(*seed*);

    long int sizestr = str.size();

    long int res = 0;

    long int count = 0;

    for(long int i=0; i<sizestr; i++)

    {

        if(i == *extractDigits*[count])

        {

            res = res\*10 + (str[i] - '0');

            count++;

        }

    }

    return res;

}

## Q3 – lam lai, so voi reference code

There are n people, each person has a number between 1 and 100000 (1 ≤ n ≤ 100000). Given a number target. Two people can be matched as a **perfect pair** if the sum of numbers they have is equal to target. A person can be matched no more than 1 time.

**Request:** Implement function:

int pairMatching(vector<int>& nums, int target);

Where nums is the list of numbers of n people, target is the given number. This function returns the number of **perfect pairs** can be found from the list.

**Example:**

The list of numbers is {1, 3, 5, 3, 7} and target = 6. Therefore, the number of **perfect pairs** can be found from the list is 2 (pair (1, 5) and pair (3, 3)).

**Note:**

In this exercise, the libraries iostream, string, cstring, climits, utility, vector, list, stack, queue, map, unordered\_map, set, unordered\_set, functional, algorithm has been included and namespace std are used. You can write helper functions and classes. Importing other libraries is allowed, but not encouraged, and may result in unexpected errors.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| vector<int>items{1, 3, 5, 3, 7};  int target = 6;  cout << pairMatching(items, target); | 2 |
| int target = 6;  vector<int>items{4,4,2,1,2};  cout << pairMatching(items, target); | 2 |

int pairMatching(vector<int>& nums, int target) {

}

int pairMatching(vector<int>& *nums*, int *target*) {

    int countPair = 0;

    while(*nums*.size()){

        int first = *nums*.front();

*nums*.erase(*nums*.begin());

        int find = *target* - first;

        for(auto i = *nums*.begin(); i<= *nums*.end(); i++){

            if(\*i == find){

                countPair++;

*nums*.erase(i);

                break;

            }

        }

    }

    return countPair;

}

code tham khao

int pairMatching(vector<int>& nums, int target) {

int count=0;

//vector<pair<int,bool>> v;

sort(nums.begin(),nums.end());

//for(unsigned int i=0;i<nums.size();i++){

// v.push\_back(make\_pair(nums[i],0));

//}

int start=0;

int end=nums.size()-1;

while(start<end){

if(nums[start]+nums[end]>target){

end--;

}

else if(nums[start]+nums[end]<target){

start++;

}

else{

count++;

start++;

end--;

}

}

//while(start<end){

// if(v[start].first+v[end.first])

//}

return count;

}

# search

## Q4

Implement function

int binarySearch(int arr[], int left, int right, int x)

to search for value x in array arr using recursion.

After traverse an index in array, we print out this index using cout << "We traverse on index: " << index << endl;

Note that middle of left and right is floor((right-left)/2)

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {1,2,3,4,5,6,7,8,9,10};  int x = 10;  int n = sizeof(arr) / sizeof(arr[0]);  int result = binarySearch(arr, 0, n - 1, x);  (result == -1) ? cout << "Element is not present in array"  : cout << "Element is present at index " << result; | We traverse on index: 4  We traverse on index: 7  We traverse on index: 8  We traverse on index: 9  Element is present at index 9 |

int binarySearch(int arr[], int left, int right, int x)

{

}

int binarySearch(int *arr*[], int *left*, int *right*, int *x*){

    if(*left* <= *right*){

        int mid = (*right*+*left*)/2;

        cout << "We traverse on index: " << mid << endl;

        if(*arr*[mid] == *x*){

            return mid;

        }

        else if(*arr*[mid] > *x*){

            return binarySearch(*arr*, *left*, mid-1, *x*);

        }

        else return binarySearch(*arr*, mid+1, *right*, *x*);

    }

    return -1;

}

## Q5

Given an array of distinct integers, find if there are two pairs (a, b) and (c, d) such that a+b = c+d, and a, b, c and d are distinct elements. If there are multiple answers, you can find any of them.

Some libraries you can use in this question:

#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <algorithm>  
#include <iostream>  
#include <utility>  
#include <map>  
#include <vector>  
#include <set>

**Note**: The function checkAnswer is used to determine whether your pairs found is true or not in case there are two pairs satistify the condition. You don't need to do anything about this function.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = { 3, 4, 7, 1, 2, 9, 8 };  int n = sizeof arr / sizeof arr[0];  pair<int, int> pair1, pair2;  if (findPairs(arr, n, pair1, pair2)) {  if (checkAnswer(arr, n, pair1, pair2)) {  printf("Your answer is correct.\n");  }  else printf("Your answer is incorrect.\n");  }  else printf("No pair found.\n"); | Your answer is correct. |
| int arr[] = { 3, 4, 7 };  int n = sizeof arr / sizeof arr[0];  pair<int, int> pair1, pair2;  if (findPairs(arr, n, pair1, pair2)) {  if (checkAnswer(arr, n, pair1, pair2)) {  printf("Your answer is correct.\n");  }  else printf("Your answer is incorrect.\n");  }  else printf("No pair found.\n"); | No pair found. |

bool findPairs(int arr[], int n, pair<int,int>& pair1, pair<int, int>& pair2)

{

// TODO: If there are two pairs satisfy the condition, assign their values to pair1, pair2 and return true. Otherwise, return false.

}

bool findPairs(int *arr*[], int *n*, pair<int,int>& *pair1*, pair<int, int>& *pair2*)

{

    // TODO: If there are two pairs satisfy the condition, assign their values to pair1, pair2 and return true. Otherwise, return false.

    // Create an empty Hash to store mapping from sum to

    // pair indexes

    map<int, pair<int, int> > Hash;

    // Traverse through all possible pairs of arr[]

    for (int i = 0; i < *n*; ++i)

    {

        for (int j = i + 1; j < *n*; ++j)

        {

            // If sum of current pair is not in hash,

            // then store it and continue to next pair

            int sum = *arr*[i] + *arr*[j];

            if (Hash.find(sum) == Hash.end())

                Hash[sum] = make\_pair(i, j);

            else // Else (Sum already present in hash)

            {

                // Find previous pair

                pair<int, int> pp = Hash[sum];// pp->previous pair

                // Since array elements are distinct, we don't

                // need to check if any element is common among pairs

*pair1* = make\_pair(*arr*[pp.first], *arr*[pp.second]);

*pair2* = make\_pair(*arr*[i], *arr*[j]);

                return true;

            }

        }

    }

    return false;

}

Reference:

<https://www.geeksforgeeks.org/find-four-elements-a-b-c-and-d-in-an-array-such-that-ab-cd/>

<https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/>

<https://www.geeksforgeeks.org/pair-in-cpp-stl/>

## Q6

Implement function

int interpolationSearch(int arr[], int left, int right, int x)

to search for value x in array arr using recursion.

After traverse to an index in array, before returning the index or passing it as argument to recursive function, we print out this index using cout << "We traverse on index: " << index << endl;

Please note that you can't using key work for, while, goto (even in variable names, comment).

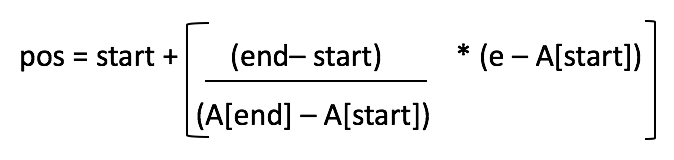
For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = { 1,2,3,4,5,6,7,8,9 };  int n = sizeof(arr) / sizeof(arr[0]);  int x = 3;  int result = interpolationSearch(arr, 0, n - 1, x);  (result == -1) ? cout << "Element is not present in array"  : cout << "Element is present at index " << result; | We traverse on index: 2  Element is present at index 2 |
| int arr[] = { 1,2,3,4,5,6,7,8,9 };  int n = sizeof(arr) / sizeof(arr[0]);  int x = 0;  int result = interpolationSearch(arr, 0, n - 1, x);  (result == -1) ? cout << "Element is not present in array"  : cout << "Element is present at index " << result; | Element is not present in array |

int interpolationSearch(int arr[], int left, int right, int x)

{

}



int interpolationSearch(int *arr*[], int *left*, int *right*, int *x*)

{

    int pos;

    // Since array is sorted, an element present

    // in array must be in range defined by corner

    if (*left* <= *right* && *x* >= *arr*[*left*] && *x* <= *arr*[*right*]) {

        // Probing the position with keeping

        // uni---m distribution in mind.

        pos = *left*

              + (((double)(*right* - *left*) / (double) (*arr*[*right*] - *arr*[*left*]))

                 \* (*x* - *arr*[*left*]));

        // Condition of target found

        cout << "We traverse on index: " << pos << endl;

        if (*arr*[pos] == *x*)

            return pos;

        // If x is larger, x is in right sub array

        if (*arr*[pos] < *x*)

            return interpolationSearch(*arr*, pos + 1, *right*, *x*);

        // If x is smaller, x is in left sub array

        if (*arr*[pos] > *x*)

            return interpolationSearch(*arr*, *left*, pos - 1, *x*);

    }

    return -1;

}

## Q7

In computer science, a jump search or block search refers to a search algorithm for ordered lists. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements. For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x. The optimal value of m is √n, where n is the length of the list.

In this question, we need to implement function jumpSearch with step √n to search for value x in array arr. After searching at an index, we should print that index until we find the index of value x in array or until we determine that the value is not in the array.

int jumpSearch(int arr[], int x, int n)

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 };  int x = 55;  int n = sizeof(arr) / sizeof(arr[0]);  int index = jumpSearch(arr, x, n);  if (index != -1) {  cout << "\nNumber " << x << " is at index " << index;  }  else {  cout << "\n" << x << " is not in array!";  } | 0 4 8 12 9 10  Number 55 is at index 10 |
| int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 };  int x = 144;  int n = sizeof(arr) / sizeof(arr[0]);  int index = jumpSearch(arr, x, n);  if (index != -1) {  cout << "\nNumber " << x << " is at index " << index;  }  else {  cout << "\n" << x << " is not in array!";  } | 0 4 8 12  Number 144 is at index 12 |

int jumpSearch(int arr[], int x, int n) {

// TODO: print the traversed indexes and return the index of value x in array if x is found, otherwise, return -1.

}

Code dung mot phan (self-code)

int jumpSearch(int *arr*[], int *x*, int *n*) {

    // TODO: print the traversed indexes and return the index of value x in array if x is found, otherwise, return -1.

    int step = (int)sqrt((double)*n*);

    //NOTE: double sqrt(double num);

    int prev=0;

    while(prev < *n*){

        cout<< prev<<" ";

        if(*arr*[prev] == *x*) return prev;

        else if(*arr*[prev] < *x*) prev += step;

        else break;

    }

    prev -= step;

    int next = prev + step;

    if(next >= *n*) next = *n*-1;

    for(int i=prev+1; i < next; i++){

        cout<< i<<" ";

        if(*arr*[i] == *x*) return i;

        else if(*arr*[i] > *x*) break;

    }

    return -1;

}

Testcase that I’m wrong (I want ignore it)

|  |  |  |
| --- | --- | --- |
| int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 611, 612, 613 };  int x = 614;  int n = sizeof(arr) / sizeof(arr[0]);  int index = jumpSearch(arr, x, n);  if (index != -1) {  cout << "\nNumber " << x << " is at index " << index;  }  else {  cout << "\n" << x << " is not in array!";  } | 0 4 8 12 16 17 18 19  614 is not in array! | 0 4 8 12 16 17  614 is not in array! |

Code đúng 100%

Code đúng 100%

int jumpSearch(int *arr*[], int *x*, int *n*) {

    // TODO: print the traversed indexes and return the index of value x in array if x is found, otherwise, return -1.

    int step = sqrt(n);

    int prev = 0;

    int i;

    for (i = 0; i < n; i += step) {

        if (arr[i] == x) {

            cout << i;

            return i;

        }

        cout << i << " ";

        if (arr[i] < x) prev = i;

        if (arr[i] > x) break;

    }

    for (int j = prev; j < i; j++) {

        if (arr[j] == x) {

            cout << j;

            return j;

        }

        if (j != prev) cout << j << " ";

        if (arr[j] > x) break;

    }

    return -1;

}

# Heap - heap sort

## Q8

Your task is to implement heap sort (in ascending order) on an unsorted array.

#define SEPARATOR "#<ab@17943918#@>#"  
#ifndef SORTING\_H  
#define SORTING\_H  
#include <iostream>  
#include <queue>  
**using** **namespace** std;  
**template** <**class** T>  
**class** Sorting {  
**public**:  
    /\* Function to print an array \*/  
    **static** **void** printArray(T \*start, T \*end)  
    {  
        **long** size = end - start;  
        **for** (**int** i = 0; i < size - 1; i++)  
            cout << start[i] << ", ";  
        cout << start[size - 1];  
        cout << endl;  
    }  
      
    **//Helping functions go here**  
    **static** **void** heapSort(T\* start, T\* end){  
        //TODO  
        Sorting<T>::printArray(start,end);  
    }  
      
};  
#endif /\* SORTING\_H \*/

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[4]={4,2,9,1};  Sorting<int>::heapSort(&arr[0],&arr[4]); | 1, 2, 4, 9 |
| int arr[4]={-1,0,2,3};  Sorting<int>::heapSort(&arr[0],&arr[4]); | -1, 0, 2, 3 |

//Helping functions go here

static void heapSort(T\* start, T\* end){

//TODO

Sorting<T>::printArray(start,end);

}

//Helping functions go here

    static void swap(*T*\* *start*, int *i*, int *j*){

*T* temp = \*(*start* + *i*);

        \*(*start* + *i*) = \*(*start* + *j*);

        \*(*start* + *j*) = temp;

    }

    static void Delete(*T*\**start*, *T*\* *end*){

*T* x = \**start*;

        int size = *end* - *start* +1;

        \**start* = \**end*;

        int i =0;

        int j= 2\*i +1;

        while(j < size-1){

            if(\*(*start*+j) < \*(*start*+j+1)){

                j = j+1;

            }

            if(\*(*start*+i) < \*(*start*+j)){

                Sorting<*T*>::swap(*start*,i,j);

                i = j;

                j=2\*i+1;

            }

            else break;

        }

        \**end* = x;

    }

    static void Heapify(*T*\* *start*, *T*\* *end*){

        int size = *end* - *start* +1;

        for(int i = (size/2)-1; i >= 0; i--){

            int j = 2\*i + 1;

            while(j < size){

                if(\*(*start*+j) < \*(*start*+j+1) && j+1 < size) j++;

                if (\*(*start*+i) < \*(*start*+j)){

                    Sorting<*T*>::swap(*start*,i,j);

                    i = j;

                    j = 2\*i + 1;

                }

                else break;

            }

        }

    }

    static void heapSort(*T*\* *start*, *T*\* *end*){

        //TODO

        Sorting<*T*>::Heapify(*start*, *end*-1);

*T*\* temp = *end*-1;

        while(*start* < temp){

            Sorting<*T*>::Delete(*start*, temp);

            temp = temp -1;

        }

        Sorting<*T*>::printArray(*start*,*end*);

    }

## Q9

In a fast food restaurant, a customer is served by following the first-come, first-served rule. The manager wants to minimize the total waiting time of his customers. So he gets to decide who is served first, regardless of how sooner or later a person comes.

Different kinds of food take different amounts of time to cook. And he can't cook food for two customers at the same time, which means when he start cooking for customer A, he has to finish A 's order before cooking for customer B. For example, if there are 3 customers and they come at time 0, 1, 2 respectively, the time needed to cook their food is 3, 9, 6 respectively. If the manager uses first-come, first-served rule to serve his customer, the total waiting time will be 3 + 11 + 16 = 30. In case the manager serves his customer in order 1, 3, 2, the total waiting time will be 3 + 7 + 17 = 27.

**Note**: The manager does not know about the future orders.

In this question, you should implement function **minWaitingTime** to help the customer find minimum total waiting time to serve all his customers. You are also encouraged to use data structure **Heap** to complete this question. You can use your own code of **Heap**, or use functions related to Heap in library <algorithm>.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int n = 3;  int arrvalTime[] = { 0, 1, 2 };  int completeTime[] = { 3, 9, 6 };  cout << minWaitingTime(n, arrvalTime, completeTime); | 27 |
| int n = 4;  int arrvalTime[] = { 0, 4, 2, 5 };  int completeTime[] = { 4, 2, 3, 4 };  cout << minWaitingTime(n, arrvalTime, completeTime); | 21 |

int minWaitingTime(int n, int arrvalTime[], int completeTime[]) {

// YOUR CODE HERE

}

int minWaitingTime(int *n*, int *arrvalTime*[], int *completeTime*[]) {

    // YOUR CODE HERE

    int sumWait = 0;

    int preTime = 0;

    while (*n* != 0) {

        int index = 0;

        bool isWaiting = false;

        for (int i = 0; i < *n*; ++i) {

            if (*arrvalTime*[i] <= preTime) {

                if (isWaiting == false) {

                    index = i;

                    isWaiting = true;

                }

                else {

                    if (*completeTime*[i] < *completeTime*[index]) {

                        index = i;

                    }

                }

            }

        }

        if (isWaiting == true) {

            preTime = preTime + *completeTime*[index];

            sumWait = sumWait + preTime - *arrvalTime*[index];

            swap(*completeTime*[index], *completeTime*[*n* - 1]);

            swap(*arrvalTime*[index], *arrvalTime*[*n* - 1]);

*n*--;

        }

        else {

            preTime = *arrvalTime*[0];

            for (int i = 1; i < *n*; i++) {

                if (*arrvalTime*[i] < preTime) {

                    preTime = *arrvalTime*[i];

                }

            }

        }

    }

    return sumWait;

}

## Q10

Implement functions: **Peek, Pop, Size, Empty, Contains** to a maxHeap. If the function cannot execute, **return -1**.

#include <iostream>  
#include <fstream>  
#include <string>  
#include <cstring>  
#include <cmath>  
#include <vector>  
#include <algorithm>  
**using** **namespace** std;  
#define SEPARATOR "#<ab@17943918#@>#"  
**template**<**class** T>  
**class** Heap {  
**protected**:  
    T\* elements;  
    **int** capacity;  
    **int** count;  
**public**:  
    Heap()  
    {  
        **this**->capacity = 10;  
        **this**->count = 0;  
        **this**->elements = **new** T[capacity];  
    }  
    ~Heap()  
    {  
        **delete**[]elements;  
    }  
    **void** push(T item);  
      
    **bool** isEmpty();  
    **bool** contains(T item);  
    T peek();  
    **bool** pop();  
    **int** size();  
      
    **void** printHeap()  
    {  
        cout << "Max Heap [ ";  
        **for** (**int** i = 0; i < count; i++)  
            cout << elements[i] << " ";  
        cout << "]\n";  
    }  
**private**:  
    **void** ensureCapacity(**int** minCapacity);  
    **void** reheapUp(**int** position);  
    **void** reheapDown(**int** position);  
};

//Your code goes here

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| Heap<int> maxHeap;  for (int i=0;i<10;i++){  maxHeap.push(i);  }  cout << maxHeap.size(); | 10 |
| Heap<int> maxHeap;  for (int i=0;i<10;i++){  maxHeap.push(i);  }  cout << maxHeap.isEmpty(); | 0 |

template<class T>

int Heap<T>::size(){

}

template<class T>

bool Heap<T>::isEmpty(){

}

template<class T>

T Heap<T>::peek(){

}

template<class T>

bool Heap<T>::contains(T item){

}

template<class T>

bool Heap<T>::pop(){

}

template<class *T*>

int Heap<*T*>::size(){

    return count;

}

template<class *T*>

bool Heap<*T*>::isEmpty(){

    return count == 0;

}

template<class *T*>

*T* Heap<*T*>::peek(){

    if(count == 0) return -1;

    return elements[0];

}

template<class *T*>

bool Heap<*T*>::contains(*T* *item*){

    for(int i = 0; i < count; i++){

        if(elements[i] == *item*) return true;

    }

    return false;

}

template<class *T*>

bool Heap<*T*>::pop(){

    if(count){

        elements[0] = elements[count-1];

        count--;

*this*->reheapDown(0);

        return true;

    }

    return false;

}

## Q11 – lam lai, so voi reference code

Cho template của class Printer Queue có 2 phương thức bắt buộc:

1. addNewRequest(int priority, string fileName)

Phương thức đầu tiên sẽ thêm 1 file vào danh sách hàng đợi của máy in (bao gồm độ ưu tiên và tên file). Test case sẽ có tối đa 100 file cùng lúc trong hàng đợi

2. print()

Phương thức thứ hai sẽ in tên file kèm xuống dòng và xóa nó ra khỏi hàng đợi. Nếu không có file nào trong hàng đợi, phương thức sẽ in ra "No file to print" kèm xuống dòng.

PrinterQueue tuân theo các quy tắc sau:

* fileName có độ ưu tiên cao nhất sẽ được in trước.
* Các fileName có cùng độ ưu tiên sẽ in theo thứ tự FIFO (First In First Out) order.

Nhiệm vụ của bạn là hiện thực class PrinterQueue thỏa mãn các yêu cầu dữ liệu trên

**Lưu ý:** Bạn có thể thay đổi mọi thứ, thêm thư viện cần thiết ngoại trừ thay đổi tên class, prototype của 2 public method bắt buộc.

**Giải thích testcase 1:** File goodbye.pdf có độ ưu tiên là 2 và được thêm vào sớm hơn file goodnight.pdf (độ ưu tiên = 2) nên sẽ được in trước, sau đó đến file goodnight.pdf và cuối cùng là hello.pdf có độ ưu tiên thấp nhất (1)

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| PrinterQueue\* myPrinterQueue = new PrinterQueue();  myPrinterQueue->addNewRequest(1, "hello.pdf");  myPrinterQueue->addNewRequest(2, "goodbye.pdf");  myPrinterQueue->addNewRequest(2, "goodnight.pdf");  myPrinterQueue->print();  myPrinterQueue->print();  myPrinterQueue->print(); | goodbye.pdf  goodnight.pdf  hello.pdf |
| PrinterQueue\* myPrinterQueue = new PrinterQueue();  myPrinterQueue->addNewRequest(1, "hello.pdf");  myPrinterQueue->print();  myPrinterQueue->print();  myPrinterQueue->print(); | hello.pdf  No file to print  No file to print |

class PrinterQueue

{

// your attributes

public:

// your methods

void addNewRequest(int priority, string fileName)

{

// your code here

}

void print()

{

// your code here

// After some logic code, you have to print fileName with endline

}

};

class *Item*{

    private:

        int prior;

*string* filename;

    public:

        Item(*string* *filename*="", int *prior*=0){

*this*->prior = *prior*;

*this*->filename = *filename*;

        }

        void setdata(*string* *filename*, int *prior*){

*this*->prior = *prior*;

*this*->filename = *filename*;

        }

        int getPrior(){

            return *this*->prior;

        }

*string* getFilename(){

            return *this*->filename;

        }

        ~Item(){}

};

class *PrinterQueue*

{

    // your attributes

private:

*Item*\* item = **new** *Item*[100];

    int count = 0;

public:

    // your methods

    void addNewRequest(int *priority*, *string* *fileName*)

    {

        // your code here

        if(count == 100) return;

        item[count].setdata(*fileName*,*priority*);

        count++;

    }

    int findIndex(){

        int highestPriority = INT8\_MIN;

        int ind = -1;

        for(int i = 0; i < count;i++){

            if(item[i].getPrior() > highestPriority){

                ind = i;

                highestPriority = item[i].getPrior();

            }

        }

        return ind;

    }

    void print()

    {

        // your code here

        // After some logic code, you have to print fileName with endline

        int ind = *this*->findIndex();

        if(ind != -1){

            cout<<item[ind].getFilename()<<endl;

            for(int i =ind; i < count; i++){

                item[ind] = item[ind+1];

            }

            count--;

        }

        else{

            cout<<"No file to print"<<endl;

        }

    }

};

code tham khảo

#include <queue>

#include <vector>

#include <iostream>

using namespace std;

struct huh {

bool operator() (pair<int, string> a, pair<int, string> b) {

return a.first < b.first;

}

};

class PrinterQueue

{

public:

// your attributes

class Node{

public:

Node\*next;

int pri;

string s;

Node(Node\*next,int pri,string s){

this->next=next;

this->pri=pri;

this->s=s;

}

};

Node\*root;

PrinterQueue(){

root=nullptr;

}

public:

// your methods

void addNewRequest(int priority, string fileName)

{

if(root==nullptr) {

root=new Node(nullptr,priority,fileName);

return ;

}

Node\*temp=root;

Node\*prev=nullptr;

while(temp!=nullptr&&temp->pri>=priority){

prev=temp;

temp=temp->next;

}

if(prev==nullptr){

Node\* newroot=new Node(nullptr,priority,fileName);

newroot->next=root;

root=newroot;

}

else{

Node \*newnode=new Node(prev->next,priority,fileName);

prev->next=newnode;

}

}

void print()

{

// your code here

// After some logic code, you have to print fileName with endline

//cout<<pq.size()<<endl;

if (root==nullptr) {

cout << "No file to print" << endl;

return;

}

cout<<root->s<<endl;

Node\*temp=root->next;

delete root;

root=temp;

}

};

## Q12

Implement function push to push a new item to a maxHeap. You also have to implement ensureCapacity and reheapUp to help you achieve that.

template

class Heap{

protected:

T \*elements;

int capacity;

int count;

public:

Heap()

{

this->capacity = 10;

this->count = 0;

this->elements = new T[capacity];

}

~Heap()

{

delete []elements;

}

void push(T item);

void printHeap()

{

cout << "Max Heap [ ";

for (int i = 0; i < count; i++)

cout << elements[i] << " ";

cout << "]";

}

private:

void ensureCapacity(int minCapacity);

void reheapUp(int position);

};

// Your code here

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| Heap<int> maxHeap;  for(int i = 0; i <5;i++)  maxHeap.push(i);  maxHeap.printHeap(); | Max Heap [ 4 3 1 0 2 ] |

template<class T>

void Heap<T>::push(T item){

}

template<class T>

void Heap<T>::ensureCapacity(int minCapacity){

}

template<class T>

void Heap<T>::reheapUp(int position){

}

template<class *T*>

void Heap<*T*>::push(*T* *item*){

    ensureCapacity(count +1);

    elements[count] = *item*;

    count++;

    reheapUp(count-1);

}

template<class *T*>

void Heap<*T*>::ensureCapacity(int *minCapacity*){

    if(*minCapacity* >= capacity){

        //re-allocate

        int old\_capacity = capacity;

        capacity = old\_capacity + (old\_capacity >> 2);

        try{

*T*\* new\_data = **new** *T*[capacity];

            //OLD: memcpy(new\_data, elements, capacity\*sizeof(T));

            memcpy(new\_data, elements, old\_capacity\*sizeof(*T*));

            delete []elements;

            elements = new\_data;

        }

        catch(std::*bad\_alloc* e){

            e.what();

        }

    }

}

template<class *T*>

void Heap<*T*>::reheapUp(int *position*){

*T* temp = elements[*position*];

    int i = *position*;

    while(i>0 && temp > elements[(i-1)/2]){

        elements[i] = elements[(i-1)/2];

        i = (i-1)/2;

    }

    elements[i] = temp;

}

## Q13 lam lai, so voi reference code

Given an integer array nums. In each operation, two numbers are removed from array, and their sum is pushed back to the array. The cost of the operation is the sum of the two removed numbers.

**Request:** Implement function:

int reduceSum(vector<int>& nums);

Where nums is the mentioned array (the number of elements is between 1 and 100000). This function returns the minimum sum of cost after repeating the operation until the array is reduced to one element.

**Example:**

Given the array: [1, 2, 3, 4]

In the first operation, 1 and 2 are removed, and their sum is pushed back to the array. The cost of the operation is 3. After the first operation, the array is: [3, 3, 4].

Repeat the process, the minimum total cost should be 19.

**Note:**

In this exercise, the libraries iostream, string, cstring, climits, utility, vector, list, stack, queue, map, unordered\_map, set, unordered\_set, functional, algorithm has been included and namespace std are used. You can write helper functions and class. Importing other libraries is allowed, but not encouraged.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| vector<int> nums {1, 2, 3, 4};  cout << reduceSum(nums); | 19 |

int reduceSum(vector<int>& nums) {

// STUDENT ANSWER

}

struct cmp{

bool operator()(int a,int b){

return a>b;

}

};

int reduceSum(vector<int>& nums) {

// STUDENT ANSWER

priority\_queue<int,vector<int>,cmp> pq;

for(unsigned int i=0;i<nums.size();i++){

pq.push(nums[i]);

}

int sum=0;

//return 1;

while(pq.size()!=1){

int t1=pq.top();

pq.pop();

int t2=pq.top();

pq.pop();

sum=sum+t1+t2;

pq.push(t1+t2);

}

return sum;

}

## Q14

Given an array which the elements in it are random. Now we want to build a Max heap from this array. Implement functions Reheap up and Reheap down to heapify element at index position. We will use it to build a heap in next question.

To keep things simple, this question will separate the heap array, not store it in the class heap

void reheapDown(int maxHeap[], int numberOfElements, int index);  
void reheapUp(int maxHeap[], int numberOfElements, int index);

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {1,2,3,4,5,6,7,8};  int size = sizeof(arr)/sizeof(arr[0]);  reheapDown(arr,size,0);  cout << "[ ";  for(int i=0;i<size;i++)  cout << arr[i] << " ";  cout << "]"; | [ 3 2 7 4 5 6 1 8 ] |
| int arr[] = {1,2,3,4,5,6,7,8};  int size = sizeof(arr)/sizeof(arr[0]);  reheapUp(arr,size,7);  cout << "[ ";  for(int i=0;i<size;i++)  cout << arr[i] << " ";  cout << "]"; | [ 8 1 3 2 5 6 7 4 ] |

void reheapDown(int maxHeap[], int numberOfElements, int index)

{

}

void reheapUp(int maxHeap[], int numberOfElements, int index)

{

}

void reheapDown(int *maxHeap*[], int *numberOfElements*, int *index*)

{

    if(*index* > (*numberOfElements*)/2 || *index*<0)   return;

    int i= *index*;

    int largest\_e=0;

        if((2\*i + 1) < *numberOfElements*){

            int left\_e= 2\*i +1;

            if((2\*i +2) < *numberOfElements*){

                int right\_e= 2\*i +2;

                if(*maxHeap*[left\_e] > *maxHeap*[right\_e])    largest\_e= left\_e;

                else    largest\_e= right\_e;

            }

            else    largest\_e= left\_e;

        }

        else    return;

        if(*maxHeap*[i] < *maxHeap*[largest\_e]){

            swap(*maxHeap*[i], *maxHeap*[largest\_e]);

            reheapDown(*maxHeap*, *numberOfElements*, largest\_e);

            //break;

        }

    return;

}

void reheapUp(int *maxHeap*[], int *numberOfElements*, int *index*)

{

    if(*index*<=0 || *index* >= *numberOfElements*)   return;

    int parent\_i= (*index*-1)/2;

    if(*maxHeap*[parent\_i] < *maxHeap*[*index*]){

        swap(*maxHeap*[parent\_i], *maxHeap*[*index*]);

        reheapUp(*maxHeap*, *numberOfElements*, parent\_i);

    }

    return;

}

## Q15 – dung mot phan

Implement method remove to **remove** the element with given value from a **maxHeap**, **clear** to remove all elements and bring the heap back to the initial state.  You also have to implement method **getItem** to help you. Some given methods that you don't need to implement again are **push**, **printHeap**, **ensureCapacity**, **reheapUp**, **reheapDown**.

class Heap {  
protected:  
    T\* elements;  
    int capacity;  
    int count;  
public:  
    Heap()  
    {  
        this->capacity = 10;  
        this->count = 0;  
        this->elements = new T[capacity];  
    }  
    ~Heap()  
    {  
        delete[]elements;  
    }  
    void push(T item);  
    int getItem(T item);  
    void remove(T item);  
    void clear();  
    void printHeap()  
    {  
        cout << "Max Heap [ ";  
        for (int i = 0; i < count; i++)  
            cout << elements[i] << " ";  
        cout << "]\n";  
    }  
private:  
    void ensureCapacity(int minCapacity);  
    void reheapUp(int position);  
    void reheapDown(int position);  
};  
  
// Your code here

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| Heap<int> maxHeap;  int arr[] = {42,35,30,15,20,21,18,3,7,14};  for (int i = 0; i < 10; i++)  maxHeap.push(arr[i]);  maxHeap.remove(42);  maxHeap.remove(35);  maxHeap.remove(30);  maxHeap.printHeap(); | Max Heap [ 21 20 18 15 14 7 3 ] |
| Heap<int> maxHeap;  int arr[] = {78, 67, 32, 56, 8, 23, 19, 45};  for (int i = 0; i < 8; i++)  maxHeap.push(arr[i]);  maxHeap.remove(78);  maxHeap.printHeap(); | Max Heap [ 67 56 32 45 8 23 19 ] |
| Heap<int> maxHeap;  int arr[] = { 13, 19, 20, 7, 15, 12, 16, 10, 8, 9, 3, 6, 18, 2, 14, 1, 17, 4, 11, 5 };  for (int i = 0; i < 20; ++i)  maxHeap.push(arr[i]);  maxHeap.clear();  maxHeap.printHeap(); | Max Heap [ ] |

template<class T>

int Heap<T>::getItem(T item) {

// TODO: return the index of item in heap

}

template<class T>

void Heap<T>::remove(T item) {

// TODO: remove the element with value equal to item

}

template<class T>

void Heap<T>::clear() {

// TODO: delete all elements in heap

}

// Your code here

template<class *T*>

int Heap<*T*>::getItem(*T* *item*) {

    // TODO: return the index of item in heap

    for(int i = 0; i < count; i++){

        if(elements[i] == *item*) return i;

    }

    return -1;

}

template<class *T*>

void Heap<*T*>::remove(*T* *item*) {

    // TODO: remove the element with value equal to item

    int index = getItem(*item*);

    if(index == -1) return;

    elements[index] = elements[count - 1];

    count--;

    reheapDown(index);

}

template<class *T*>

void Heap<*T*>::clear() {

    // TODO: delete all elements in heap

    while(count){

        remove(elements[0]);

    }

    delete[]elements;

    elements = **new** *T*[capacity];

}

## Q16 lam lai, so voi reference code

Given an array of non-negative integers. Each time, we can take the smallest integer out of the array, multiply it by 2, and push it back to the array.

**Request:** Implement function:

int leastAfter(vector<int>& nums, int k);

Where nums is the given array (the length of the array is between 1 and 100000). This function returns the smallest integer in the array after performing the operation k times (k is between 1 and 100000).

**Example:**

Given nums = [2, 3, 5, 7].

In the 1st operation, we take 2 out and push back 4. The array is now nums = [3, 4, 5, 7].

In the 2nd operation, we take 3 out and push back 6. The array is now nums = [4, 5, 6, 7].

In the 3rd operation, we take 4 out and push back 8. The array is now nums = [5, 6, 7, 8].

With k = 3, the result would be 5.

**Note:**

In this exercise, the libraries iostream, string, cstring, climits, utility, vector, list, stack, queue, map, unordered\_map, set, unordered\_set, functional, algorithm has been included and namespace std are used. You can write helper functions and classes. Importing other libraries is allowed, but not encouraged, and may result in unexpected errors.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| vector<int> nums {2, 3, 5, 7};  int k = 3;  cout << leastAfter(nums, k); | 5 |

int leastAfter(vector<int>& nums, int k) {

// STUDENT ANSWER

}

struct mycmp{

bool operator()(int a,int b){

return a>b;

}

};

int leastAfter(vector<int>& nums, int k) {

// STUDENT ANSWER

priority\_queue<int,vector<int>,mycmp> pq;

for(unsigned int i=0;i<nums.size();i++){

pq.push(nums[i]);

}

for(int i=0;i<k;i++){

int temp=pq.top();

pq.pop();

pq.push(temp\*2);

}

return pq.top();

}