**1.** 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>  
**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 swap (T &a, T &b) {  int temp = a;  a = b;  b = temp;  }  static void heapify (T \*start, T \*end, int index) {  int size = end - start;  int largest = index;  int l = 2 \* index + 1;  int r = 2 \* index + 2;  if (l < size && start [l] > start [largest])  largest = l;  if (r < size && start [r] > start [largest])  largest = r;    if (largest != index) {  swap (start [index], start [largest]);  heapify (start, end, largest);  }  }  static void heapSort (T \*start, T \*end) {  // TODO  int size = end - start;  for (int i = size / 2 - 1; i >= 0; i--)  heapify (start, end, i);    for (int i = size - 1; i > 0; i--) {  swap (start [0], start [i]);  heapify (start, start + i, 0);  }  Sorting<T>::printArray(start, end);  } |

**2.** 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) {  if (this -> count == this -> capacity) this -> ensureCapacity (this -> capacity \* 2);  this -> elements [count] = item;  ++this -> count;  this -> reheapUp (this -> count - 1);  }  template <class T>  void Heap <T> :: ensureCapacity (int minCapacity) {  T \*newElements = new T [minCapacity];  for (int i = 0; i < this -> count; ++i) {  newElements [i] = this -> elements[i];  }  delete[] this -> elements;  this -> elements = newElements;  this -> capacity = minCapacity;  }  template <class T>  void Heap <T> :: reheapUp (int position) {  if (position <= 0 || position >= this -> count) return;  int parent = (position - 1) / 2;  if (this -> elements [parent] < this -> elements [position]) {  int temp = this -> elements [parent];  this -> elements [parent] = this -> elements [position];  this -> elements [position] = temp;  reheapUp (parent);  }  return;  } |

**3.** 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) {  if (numberOfElements <= 0 || index < 0 || index > numberOfElements) return;  int left = index \* 2 + 1;  int right = index \* 2 + 2;  int large = 0;  if (left <= numberOfElements - 1) {  if (right <= numberOfElements && maxHeap [right] > maxHeap [left])  large = right;  else  large = left;  if (maxHeap [large] > maxHeap [index]) {  std :: swap (maxHeap [large], maxHeap [index]);  reheapDown (maxHeap, numberOfElements, large);  }  }  }  void reheapUp (int maxHeap[], int numberOfElements, int index) {  if (numberOfElements <= 0 || index < 0 || index > numberOfElements) return;  if (index > 0) {  int parent = (index - 1) / 2;  if (maxHeap [index] > maxHeap [parent]) {  std :: swap (maxHeap [index], maxHeap [parent]);  reheapUp (maxHeap, numberOfElements, parent);  }  }  } |

**4.** 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  if (count > capacity) return -1;  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  if (count > capacity) return;  int index = getItem (item);  if (index > -1) {  elements [index] = elements [count - 1];  count--;  ensureCapacity (-1);  this -> capacity += 1;  reheapDown (index);  }  }  template <class T>  void Heap <T> :: clear() {  // TODO: delete all elements in heap  delete[] elements;  capacity = 10;  count = 0;  elements = new T [capacity];  } |

**5.** 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.  int step = sqrt (n);  int prev = 0;  int i;  for (i = 0; i < n; i += step) {  if (arr [i] <= x) prev = i;  if (arr [i] == x) {  cout << i;  return i;  }  cout << i << " ";  if (arr [i] > x) break;  }  if (prev > n - 1) return -1;  else {  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;  } |

**6.** 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) {  if (left > right || arr [left] > x || arr [right] < x) return -1;  int mid = left + ((x - arr [left]) \* (right - left)) / (arr [right] - arr [left]);  if (mid < left) return -1;  if (mid > right) return -1;  if (arr [mid] == x) {  cout << "We traverse on index: " << mid << endl;  return mid;  }  else if (arr [mid] < x) {  cout << "We traverse on index: " << mid << endl;  left = mid + 1;  }  else {  cout << "We traverse on index: " << mid << endl;  right = mid - 1;  }  return interpolationSearch (arr, left, right, x);  } |

**7.** 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) {  if (left > right) return -1;  int mid = (right + left) / 2;  if (arr [mid] == x) {  cout << "We traverse on index: " << mid << endl;  return mid;  }  else if (arr [mid] < x) {  cout << "We traverse on index: " << mid << endl;  left = mid + 1;  }  else {  cout << "We traverse on index: " << mid << endl;  right = mid - 1;  }  return binarySearch (arr, left, right, x);  } |

**8.** 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) {  long long tmp = key;  int A [100] = {};  int n = 0;  while (tmp != 0) {  A [n] = tmp % 10;  tmp = tmp / 10;  n++;  }  long long sum = 0;  while (n > 0) {  long long s = 0;  for (int i = 0; i < addressSize; i++) {  if (n - i - 1 >= 0) {  s = s \* 10 + A [n - i - 1];  }  }  n -= addressSize;  sum += s;  }  long long t = 1;  for (int i = 0; i < addressSize; i++) t \*= 10;  return sum % t;  }  int rotation (long long key, int addressSize) {  long long tmp = key;  int A [100] = {};  int n = 0;  while (tmp != 0) {  A [n] = tmp % 10;  tmp = tmp / 10; n++;  }  tmp = A [0];  for (int i = 0; i < n - 1; i++) A [i] = A [i + 1];  A [n - 1] = tmp;  long long sum = 0;  while (n > 0) {  long long s = 0;  for (int i = 0; i < addressSize; i++) {  if (n - i - 1 >= 0) {  s = s \* 10 + A [n - i - 1];  }  }  n -= addressSize; sum += s;  }  long long t = 1;  for (int i = 0; i < addressSize; i++) t \*= 10;  return sum % t;  } |

**9.** 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.  map <int, pair <int, int>> Hash;  for (int i = 0; i < n; ++i) {  for (int j = i + 1; j < n; ++j) {  int sum = arr [i] + arr [j];  if (Hash.find(sum) == Hash.end()) {  Hash [sum] = make\_pair (i, j);  }  else {  pair <int, int> temp = Hash [sum];  pair1.first = arr [temp.first];  pair1.second = arr [temp.second];  pair2 = make\_pair (arr [i], arr [j]);  return true;  }  }  }  return false;  } |

**10.** 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) {  int square = seed \* seed;  return (square / 100) % 10000;  }  long int moduloDivision (long int seed, long int mod) {  return seed % mod;  }  long int digitExtraction (long int seed, int \*extractDigits, int size) {  int add = 0;  int e = 0;  int t\_seed = seed;  int s\_size = 0;  while (t\_seed > 0) {  t\_seed /= 10;  s\_size++;  }  for (int i = 0; i < size; i++) {  int modi = pow (10, s\_size - extractDigits [i]);  e = ((seed % modi) - (seed % (modi / 10))) / (modi / 10);  e \*= pow (10, size - i - 1);  add += e;  }  return add;  } |