1. Binary heap operations

int MinHeap::extractMin()

{

if (heap\_size <= 0)

return -1;

if (heap\_size == 1)

{

heap\_size--;

return harr[0];

}

else

{

int root = harr[0];

harr[0] = harr[heap\_size-1];

heap\_size--;

MinHeapify(0);

return root;

}

}

//Function to delete a key at ith index.

void MinHeap::deleteKey(int i)

{

if(i>=heap\_size)

{

return;

}

decreaseKey(i, INT\_MIN);

extractMin();

}

//Function to insert a value in Heap.

void MinHeap::insertKey(int k)

{

heap\_size++;

int i = heap\_size - 1;

harr[i] = k;

while (i != 0 && harr[parent(i)] > harr[i])

{

swap(harr[i], harr[parent(i)]);

i = parent(i);

}

}

//Function to change value at ith index and store that value at first index.

void MinHeap::decreaseKey(int i, int new\_val)

{

harr[i] = new\_val;

while (i != 0 && harr[parent(i)] > harr[i]) {

swap(harr[i], harr[parent(i)]);

i = parent(i);

}

}

/\* You may call below MinHeapify function in

above codes. Please do not delete this code

if you are not writing your own MinHeapify \*/

void MinHeap::MinHeapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && harr[l] < harr[i]) smallest = l;

if (r < heap\_size && harr[r] < harr[smallest]) smallest = r;

if (smallest != i) {

swap(harr[i], harr[smallest]);

MinHeapify(smallest);

}

}

1. Heap sort

void heapify(int arr[], int n, int i)

{

int largest = i;

int l = 2\*i + 1;

int r = 2\*i + 2;

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i)

{

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

public:

//Function to build a Heap from array.

void buildHeap(int arr[], int n)

{

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

heapify(arr, n, i);

}

public:

//Function to sort an array using Heap Sort.

void heapSort(int arr[], int n)

{

buildHeap(arr,n);

for (int i=n-1; i>0; i--)

{

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

1. K largest elements

vector<int> kLargest(int arr[], int n, int k)

{

sort(arr, arr+n);

vector<int> ans;

for(int i=n-1;i>=n-k;i--)

ans.push\_back(arr[i]);

return ans;

}

1. Kth largest element

int KthLargest(int arr[], int n, int k) {

sort(arr, arr+n);

return arr[n-k];

}

1. Kth smallest element

int kthSmallest(int arr[], int n, int k)

{

sort(arr, arr+n);

return arr[k-1];

}

1. Kth largest element in stream

void kthLargest(int arr[], int n, int k)

{

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

for(int i = 0; i<k-1 ;i++)

{

cout<<"-1 ";

pq.push(arr[i]);

}

pq.push(arr[k-1]);

cout<<pq.top()<<" ";

for(int i=k;i<n;i++)

{

if(arr[i]>pq.top()){

pq.pop();

pq.push(arr[i]);

}

cout<<pq.top()<<" ";

}

}

1. K most occurring elements

int kMostFrequent(int arr[], int n, int k)

{

int res=0;

unordered\_map<int,int> m;

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

m[arr[i]]++;

priority\_queue<int> pq;

for(auto it:m){

int freq = it.second;

pq.push(freq);

}

while(k--){

res += pq.top();

pq.pop();

}

return res;

}

1. Minimum cost of ropes

long long minCost(long long arr[], long long n) {

priority\_queue <long long, vector<long long>, greater<long long> > pq;

for(long long i = 0; i<n; i++)

pq.push(arr[i]);

long long cost = 0;

while(pq.size()>1)

{

long temp = pq.top();

pq.pop();

temp+=pq.top();

pq.pop();

pq.push(temp);

cost+=temp;

}

return cost;

}

1. Nearly sorted

vector <int> nearlySorted(int arr[], int num, int k){

priority\_queue<int,vector<int>,greater<int>>pq(arr,arr+num);

vector<int>v;

while(!pq.empty()){

v.push\_back(pq.top());

pq.pop();

}

return v;

}