**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text;**

**using System.Threading.Tasks;**

**namespace ConsoleApp3**

**{**

**class priorityqueue**

**{**

**node[] Arr;-> θ(1)**

**int length = 0;-> θ(1)**

# public void enqueue(node x)

**{**

**insert\_value(x); O(log|V|)**

**}**

# public node dequeue()

**{**

**return extract\_min();O(log|V|)**

**}**

public bool empty() total= θ(1)

**{**

**return (length == 0);-> θ(1)**

**}**

# public priorityqueue()

**{**

**Arr = new node[10000];-> θ(1)**

**}**

public int count() **{ return length; }** -> O(1)

void insert\_value(node val) total= O(log|V|)

**{**

**length = length + 1;-> θ(1)**

**Arr[length] = null; //assuming all the numbers greater than 0 are to be inserted in queue.**

**increase\_value(length, val); = O(log|V|)**

**}**

public void increase\_value(int i, node val) total= O(log|V|)

**{**

**Arr[i] = val;//1**

**while (i > 1 && Arr[i / 2].F >= Arr[i].F)**

**total =#of itrations\*O(Body)= O(log|V|)\* O(1)=. O(log|V|)**

**{**

**swap(ref Arr[i / 2], ref Arr[i]); ->O(1).**

**i = i / 2;**

**}**

**}**

void min\_heapify(int i, int N) total= O(log|V|)

**{**

**int left = 2 \* i; ];-> θ(1)**

**int right = 2 \* i + 1; -> θ(1)**

**int smallest; ];-> θ(1)**

**if (left <= N && Arr[left].F < Arr[i].F) ];-> θ(1)**

**smallest = left; ];-> θ(1)**

**else**

**smallest = i; ];-> θ(1)**

**if (right <= N && Arr[right].F < Arr[smallest].F) ];-> θ(1)**

**smallest = right; ];-> θ(1)**

**if (smallest != i) ];-> θ(1)**

**{**

**swap(ref Arr[i], ref Arr[smallest]); ];-> O(1)**

**min\_heapify(smallest, N); ];-> O(log(|V|)**

**}**

**}**

void build\_minheap(int N) Total =O(V)

**{**

**for (int i = N / 2; i >= 1; i--)**

**min\_heapify(i, N);**

**}**

void swap(ref node x, ref node y) ->O(1)

**{**

**node t = x; ->O(1)**

**x = y; ->O(1)**

**y = t; ->O(1)**

**}**

node extract\_min() Total=O(log|V|)

**{**

**if (length == 0) -> θ(1)**

**{**

**throw new InvalidOperationException("Can’t remove element as queue is empty");**

**}**

**node min = Arr[1]; -> θ(1)**

**Arr[1] = Arr[length]; -> θ(1)**

**length = length - 1; -> θ(1)**

**min\_heapify(1, length); O(log|V|)**

**return min; -> θ(1)**

**}**

**}**

**class Program**

**{**

static priorityqueue pq; **-> θ(1)**

# public static void astar(node start)

# **{**

**pq = new priorityqueue();-> θ(1)**

**node root = start; -> θ(1)**

**pq.enqueue(root); O(log(V))**

**while (!pq.empty()) total =#of itrations\*O(Body)=O(E)\*O( log(V))=O(E log(v))**

**{**

**node top = pq.dequeue();O()**

**if (top.H == 0) -> θ(1)**

**{**

**Console.WriteLine(top.Level); -> θ(1)**

**return; -> θ(1)**

**}**

**top.getadj();-> O(1)**

**//c ->as a constant**

**//counter of adj list max will be 4**

**for (int i = 0; i < top.adj.Count(); i++)) total =#of itrations\*O(Body)=O(C)\*O(log|v|)=O( clog|v|)**

**{**

**node front = top.adj[i]; -> θ(1)**

**front.F = front.Level + front.H; -> θ(1)**

**pq.enqueue(front); O(log|v|)**

**}**

**}**

**}**

**class node**

**{**

**public int[,] borad; -> θ(1)**

**public List<node> adj; -> θ(1)**

**public int index0i; -> θ(1)**

**public int index0j; -> θ(1)**

**public int indexPi; -> θ(1)**

**public int indexPj; -> θ(1)**

**public int Level; -> θ(1)**

**public int F; -> θ(1)**

**public int H; -> θ(1)**

**public int N; -> θ(1)**

# public node(int[,] tmp, int N, int ii, int jj, int H, int Level)

{

**borad = tmp; -> θ(1)**

**indexPi = -1; -> θ(1)**

**indexPj = -1; -> θ(1)**

**this.N = N; -> θ(1)**

**this.H = H; -> θ(1)**

**this.Level = Level; -> θ(1)**

**this.index0i = ii; -> θ(1)**

**this.index0j = jj; -> θ(1)**

**adj = new List<node>();-> θ(1)**

**}total = -> θ(1)**

# public node(int[,] tmp, int N, int ii, int jj, int H, int indexPi, int indexPj, int Level)

{

**borad = tmp; -> θ(1)**

**this.N = N; -> θ(1)**

**this.index0i = ii; -> θ(1)**

**this.index0j = jj; -> θ(1)**

**this.Level = Level; -> θ(1)**

**this.indexPi = indexPi; -> θ(1)**

**this.indexPj = indexPj; -> θ(1)**

**this.H = H; -> θ(1)**

**adj = new List<node>();-> θ(1)**

**} total=-> θ(1)**

# int ham(node tmp, int n, int iOfLastCell, int jOfLastCell)

**{**

**int h = tmp.H;**

**if (((tmp.index0i \* tmp.N + tmp.index0j) + 1) == n) -> θ(1)**

**{**

**h--;-> θ(1)**

**}**

**if (((iOfLastCell \* tmp.N + jOfLastCell) + 1) == n) -> θ(1)**

**{**

**h++;-> θ(1)**

**}**

**return h; -> θ(1)**

**}**

# void swap(ref int x, ref int y)

**{**

**int t = x; -> θ(1)**

**x = y; -> θ(1)**

**y = t; -> θ(1)**

**}**

# public bool issame(node tmp, int i, int j)

**{**

**if (i == tmp.indexPi && j == tmp.indexPj) -> θ(1)**

**{**

**return true; -> θ(1)**

**}**

**return false; -> θ(1)**

**}**

# public void getadj()

**{**

**// node tt;**

**node parent = this;**

**if (index0i > 0) -> θ(1)**

**{//swap**

**int[,] temp = new int[N, N]; -> θ(1)**

**Array.Copy(borad, temp, borad.Length); O(borad.length))**

**int nn = temp[index0i - 1, index0j]; -> θ(1)**

**int hh = ham(parent, nn, index0i - 1, index0j); -> θ(1)**

**swap(ref temp[index0i, index0j], ref temp[index0i - 1, index0j]); -> θ(1)**

**node t1 = new node(temp, N, index0i - 1, index0j, hh, index0i, index0j, parent.Level + 1); -> θ(1)**

**if (!issame(parent, index0i - 1, index0j)) O(1)**

**{**

**adj.Add(t1);  O(1)**

**}**

**}**

**if (index0i + 1 < N) -> θ(1)**

**{**

**int[,] temp = new int[N, N]; -> θ(1)**

**Array.Copy(borad, temp, borad.Length); O(borad.length)**

**int nn = temp[index0i + 1, index0j]; -> θ(1)**

**int hh = ham(parent, nn, index0i + 1, index0j); -> θ(1)**

**swap(ref temp[index0i + 1, index0j], ref temp[index0i, index0j]); -> θ(1)**

**node t2 = new node(temp, N, index0i + 1, index0j, hh, index0i, index0j, parent.Level + 1); -> θ(1)**

**if (!issame(parent, index0i + 1, index0j)) O(1)**

**{**

**adj.Add(t2);  O(1)**

**}**

**}**

**if (index0j > 0)**

**{**

**int[,] temp = new int[N, N]; -> θ(1)**

**Array.Copy(borad, temp, borad.Length); O(borad.length))**

**int nn = temp[index0i, index0j - 1]; -> θ(1)**

**int hh = ham(parent, nn, index0i, index0j - 1); -> θ(1)**

**swap(ref temp[index0i, index0j], ref temp[index0i, index0j - 1]); -> θ(1)**

**node t3 = new node(temp, N, index0i, index0j - 1, hh, index0i, index0j, parent.Level + 1); -> θ(1)**

**if (!issame(parent, index0i, index0j - 1)) O(1)**

**{**

**adj.Add(t3);O(1)**

**}**

**}**

**if (index0j + 1 < N) -> θ(1)**

**{**

**int[,] temp = new int[N, N]; -> θ(1)**

**Array.Copy(borad, temp, borad.Length); O(borad.length))**

**int nn = temp[index0i, index0j + 1]; -> θ(1)**

**int hh = ham(parent, nn, index0i, index0j + 1); -> θ(1)**

**swap(ref temp[index0i, index0j], ref temp[index0i, index0j + 1]); -> θ(1)**

**node t4 = new node(temp, N, index0i, index0j + 1, hh, index0i, index0j, parent.Level + 1); -> θ(1)**

**if (!issame(parent, index0i, index0j + 1)) O(1)**

**{**

**adj.Add(t4);O(1)**

**}**

**}**

**}**

**}**

# Analysic of build\_minheap:-

Let *h = log n* represent the height of the heap. The work required for the siftDown approach is given by the sum

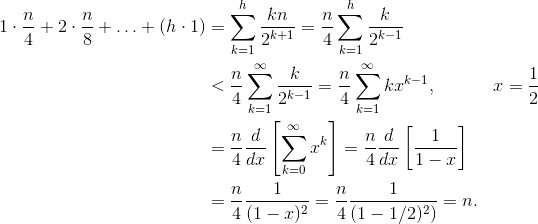
(0 \* n/2) + (1 \* n/4) + (2 \* n/8) + ... + (h \* 1).

Each term in the sum has the maximum distance a node at the given height will have to move (zero for the bottom layer, h for the root) multiplied by the number of nodes at that height. In contrast, the sum for calling siftUp on each node is

(h \* n/2) + ((h-1) \* n/4) + ((h-2)\*n/8) + ... + (0 \* 1).

It should be clear that the second sum is larger. The first term alone is *hn/2 = 1/2 n log n*, so this approach has complexity at best *O(n log n)*.

One method (there are other analyses that also work) is to turn the finite sum into an infinite series and then use Taylor series. We may ignore the first term, which is zero:

[](https://i.stack.imgur.com/959f6.png)

**#O(N)**