Practical 7

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

The Aim of this practical was to implement the heap using the linked data structure. The version of heap that is implemented here is called leftist heap. This datastructure Is implemented like. Every node has ofcourse left and right pointers but one extra number which is “rank” which is distance of node from nearest leaf. The insertion and deletion takes 𝒪

The build heap can be done in 𝒪

# Code

* Leftist Heap (LeftistHeap.h)

#ifndef LEFTIST\_HEAP\_H

#define LEFTIST\_HEAP\_H

#include <iostream>

#include <cassert>

template<typename Key>

struct heap\_node {

private:

constexpr static size\_t space\_block = 7;

public:

Key key;

size\_t rank;

heap\_node<Key>\* left;

heap\_node<Key>\* right;

heap\_node(const Key& key) : key(key), rank(1), left(nullptr), right(nullptr) { }

friend heap\_node<Key>\* merge(heap\_node<Key>\* l, heap\_node<Key>\* r) {

if (l == nullptr)

return r;

if (r == nullptr)

return l;

if (l->key < r->key) {

l->right = merge(l->right, r);

assert(l->right);

if (!l->left || l->left->rank < l->right->rank) {

std::swap(l->left, l->right);

}

l->rank = l->right ? l->right->rank + 1 : 1;

return l;

}

else {

r->right = merge(r->right, l);

assert(r->right);

if (!r->left || r->left->rank < r->right->rank) {

std::swap(r->left, r->right);

}

r->rank = r->right ? r->right->rank + 1 : 1;

return r;

}

assert(false);

}

friend std::ostream& print\_helper(std::ostream& stream, const heap\_node<Key>\* ptr, size\_t spaces = 0) {

if (!ptr)

return stream;

spaces += space\_block;

print\_helper(stream, ptr->right, spaces);

stream << '\n';

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

stream << ' ';

stream << ptr->key << " (" << ptr->rank << ")";

print\_helper(stream, ptr->left, spaces);

}

~heap\_node() {

if (left) {

delete left;

}

if (right) {

delete right;

}

}

};

template<typename Key>

struct leftist\_heap {

private:

heap\_node<Key>\* m\_root;

heap\_node<Key>\* \_insert(heap\_node<Key>\* root, const Key& key) {

if (!root) {

return new heap\_node<Key>(key);

}

if (key < root->key) {

heap\_node<Key>\* new\_node = new heap\_node<Key>(key);

new\_node->left = root;

return new\_node;

}

if (root->right && key > root->right->key) {

root->right = \_insert(root->right, key);

assert(root->left && root->right);

if (root->left->rank < root->right->rank)

std::swap(root->left, root->right);

root->rank = root->right->rank + 1;

return root;

}

heap\_node<Key>\* new\_node = new heap\_node<Key>(key);

if (root->left == nullptr) {

root->left = new\_node;

return root;

}

assert(root->right == nullptr);

assert(root->rank == 1);

root->right = new\_node;

root->rank = 2;

assert(root->left && root->right);

if (root->left->rank < root->right->rank) {

std::swap(root->left, root->right);

}

return root;

}

public:

leftist\_heap() : m\_root(nullptr) { }

void insert(const Key& key) {

m\_root = \_insert(m\_root, key);

// We could also create new heap with one node and merge with existing tree.

// I wrote this merge function later so i already had insert both implementations are fine

// auto new\_node = new heap\_node<Key>(key);

// m\_root = merge(m\_root, new\_node);

}

void pop() {

heap\_node<Key>\* left\_child = m\_root->left;

heap\_node<Key>\* right\_child = m\_root->right;

m\_root->left = nullptr;

m\_root->right = nullptr;

delete m\_root; // resetting left, right was necessary because delete operator deletes childs recursively

m\_root = merge(left\_child, right\_child);

}

const Key& top() const {

return m\_root->key;

}

bool empty() const { return m\_root == nullptr; }

void vertical\_print() const {

print\_helper(std::cout, m\_root);

}

~leftist\_heap() {

delete m\_root;

m\_root = nullptr;

}

};

#endif // LEFTIST\_HEAP\_H

* LeftistHeap Test (main function)

#include <iostream>

#include "LeftistHeap.h"

using namespace std;

int main() {

leftist\_heap<int> hp;

for (int i = 0; i < 10; i++)

hp.insert(i + 1);

std::cout << "After Inserting [0-10]\n";

hp.vertical\_print();

std::cout << "\n";

cout << "Current Min Element : " << hp.top() << "\n";

hp.pop();

cout << "After Poping Min Element\n";

hp.vertical\_print();

return 0;

}

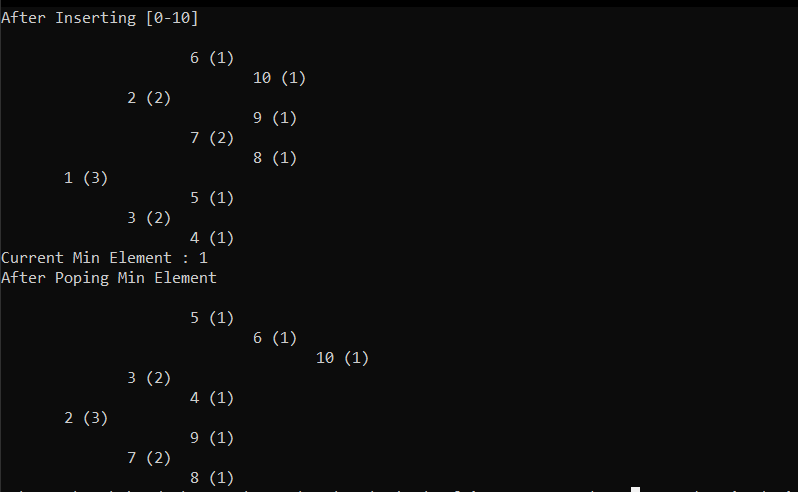
# Inputs

This Program Doesn’t take any inputs.

The Program itself inserts 1 to 10 Numbers in heap and displays them.

Then deletes one element and prints modified heap. The Number in bracket in output is “rank” of node.

# Output



# Conclusion

Here we learned about the merge, insert, delete functions of leftist heap. It takes other approach for leftist heap is it tries to be unbalanced and then rebalances itself. Here right child’s “rank” is always lower than “rank” of left child for every immediate node.