Lab 1: Implementation of priority queue, stack and binary search trees using C++

Objective:

To implement priority queue, stack and binary search tree using C++.

Theory:

1. Stack : A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO (Last in First Out) or FILO (First in Last Out). LIFO implies that is inserted first, comes out last.
    a. Insertion Algorithm:
        i. Checks if the stack is full.
        ii. If the stack is full, produces an error and exit.
        iii. If the stack is not full, increments top to point next empty space.
        iv. Adds data element to the stack location, where top is pointing.
        v. Returns success.
    b. Deletion Algorithm:
        i. Checks if the stack is empty.
        ii. If the stack is empty, produces an error and exit.
        iii. If the stack is not empty, accesses the data element at which top is pointing.
        iv. Decreases the value of top by 1.
        v. Returns success.
2. Priority Queue: A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.
   In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value.
    a. Inserstion Algorithm:
        i. Add the element to the bottom level of the heap at the leftmost open space.
        ii. 2.Compare the added element with its parent; if they are in the correct order, stop.
        iii. 3.If not, swap the element with its parent and return to the previous step 2.
    b. Deletion Algorithm:
        i. 1. Replace the root of the heap with the last element on the last level.
        ii. 2. Compare the new root with its children; if they are in the correct order, stop.
        iii. 3. If not, swap the element with one of its children and return to the previous step 2.
3. Binary Search Tree: A Binary Search tree is a data structure used in computer science for organizing and storing data in a sorted manner. Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing

values less than the parent node and the right child containing values greater than the parent node. This hierarchical structure allows for efficient searching, insertion, and deletion operations on the data stored in the tree.

a. Insertion Algorithm:
    i. Create a node to be inserted (newNode) and assign it the value.
    ii. Start from the root node and compare the values of newNode and root node.
    iii. If the newNode is less than root, go to left child of root, otherwise go to the right child of the root.
    iv. Continue the step 2 (each node is a root for its sub tree) until a null pointer (or leaf node) is found.
    v. Insert the newNode as left child if newNode is less than leaf node.
    vi. Insert the newNode as a right child if newNode is greater than leaf node.

Observation:

To measure time taken to run the functions we created an utility function, getTime which accepts a function lambda as an argument and measures the time taken to run the function in nanoseconds. This function will also be included in other lab works.

```
#pragma once
#include <functional>
#include <chrono>
long long getTime(std::function<void()> f)
{
    auto start = std::chrono::high_resolution_clock::now();
    f();
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
start);
    return elapsed.count();
}
```

1. Stack:

```
#include <iostream>
#include <functional>
#include <vector>
#include "gettime.h"
using namespace std;

class Stack
{
    int *elements;

public:
    size_t size = 0;
    size_t max = 0;

    Stack(size_t size)
    {
        elements = (int *)(calloc(size, sizeof(int)));
        max = size;
    }
    int top()
    {
        if (size > 0)
        {
            return elements[size - 1];
        }
        throw out_of_range("Nothing on top");
    }
```

```
    int pop()
    {
        if (size == 0)
        {
            throw out_of_range("Nothing to pop");
        }
        int ret = elements[size - 1];
        size--;
        return ret;
    }
    int &operator[](size_t index)
    {
        if (index > size - 1)
        {

            throw out_of_range("Index out of range");
        }
        return elements[index];
    }
    void push(int value)
    {
        if (size + 1 <= max)
        {
            elements[size] = value;
            size++;
        }
        else
        {
            throw out_of_range("Stack is full");
        }
    }
};

int main()
{
    vector<int> sizes;
    sizes.push_back(10000000);
    for (size_t i = 0; i < 4; i++)
    {
        sizes.push_back(sizes.back() + 5000);
    }
    for (size_t i = 0; i < sizes.size(); i++)
    {
        int size = sizes[i];
        Stack s(size);
        auto push_items = [&]()
        {
            for (int i = 0; i < s.max; i++)
```

```
            {
                s.push(i);
            }
        };

        cout << "Time taken to push " << size << " items: " <<
getTime(push_items) << " nanoseconds\n";
    }
    for (size_t i = 0; i < sizes.size(); i++)
    {
        int size = sizes[i];
        Stack s(size);

        for (int i = 0; i < s.max; i++)
        {
            s.push(i);
        }

        auto pop_items = [&]()
        {
            while (s.size > 0)
            {
                s.pop();
            }
        };

        cout << "Time taken to pop " << size << " items: " <<
getTime(pop_items) << " nanoseconds\n";
    }
    return 0;
}
```

Output:

```
Time taken to push 10000000 items: 63698600 nanoseconds
Time taken to push 10005000 items: 60155100 nanoseconds
Time taken to push 10010000 items: 71405200 nanoseconds
Time taken to push 10015000 items: 77423500 nanoseconds
Time taken to push 10020000 items: 78233600 nanoseconds
Time taken to pop 10000000 items: 54558200 nanoseconds
Time taken to pop 10005000 items: 50655500 nanoseconds
Time taken to pop 10010000 items: 61593500 nanoseconds
Time taken to pop 10015000 items: 47190800 nanoseconds
Time taken to pop 10020000 items: 45748000 nanoseconds
```

2. Priority Queue:

```
#include <iostream>
#include <vector>
#include "gettime.h"
```

```
using namespace std;

class PriorityQ
{
public:
    vector<pair<int, int>> queue;

    int getParent(int idx)
    {
        if (idx % 2 == 0)
            return idx / 2 - 1;
        return idx / 2;
    }

    int leftIdx(int idx)
    {
        return 2 * idx + 1;
    }
    int rightIdx(int idx)
    {
        return 2 * idx + 2;
    }

    void shiftUp(int idx)
    {
        while (idx > 0 && queue[getParent(idx)].second < queue[idx].second)
        {
            swap(queue[getParent(idx)], queue[idx]);
            idx = getParent(idx);
        }
    }

    void shiftDown(int idx)
    {
        int maxIdx = idx;
        int left = leftIdx(idx);
        if (left < queue.size() && queue[left].second > queue[maxIdx].second)
        {
            maxIdx = left;
        }

        int right = rightIdx(idx);

        if (right < queue.size() && queue[right].second >
queue[maxIdx].second)
        {
            maxIdx = right;
        }
```

```
        if (idx != maxIdx)
        {
            swap(queue[idx], queue[maxIdx]);
            shiftDown(maxIdx);
        }
    }

    int extractMax()
    {
        int res = queue[0].first;
        queue[0] = queue[queue.size() - 1];
        queue.pop_back();
        shiftDown(0);
        return res;
    }

    void remove(int idx)
    {
        queue[idx].second = queue[0].second + 1;
        shiftUp(idx);
        extractMax();
    }

    void push(int value, int priority)
    {
        queue.push_back({value, priority});
        shiftUp(queue.size() - 1);
    }
};

int main()
{
    vector<int> sizes;
    int start_size = 1000000;
    int increment = 500000;
    sizes.push_back(start_size);
    srand(time(NULL));
    for (size_t i = 0; i < 4; i++)
    {
        sizes.push_back(sizes.back() + increment);
    }
    for (size_t i = 0; i < sizes.size(); i++)
    {
        int size = sizes[i];
        PriorityQ pq;
        long long time = getTime([&]()
                                 {
```

```
            for (size_t i = 0; i < size; i++)
            {
                //insert random values with random priority
                pq.push(rand(), rand());
            } });
        cout << "Insertion of " << size << " elements took " << time << "
nanoseconds" << endl;
        time = getTime([&]()
                       {
            for (size_t i = 0; i < size; i++)
            {
                //remove random
                pq.remove(rand() % pq.queue.size());
            } });
        cout << "removal of " << size << " elements took " << time << "
nanoseconds" << endl;
    }
    return 0;
}
```

Output:

```
Insertion of 1000000 elements took 155615700 nanoseconds
removal of 1000000 elements took 1622739000 nanoseconds
Insertion of 1500000 elements took 249995900 nanoseconds
removal of 1500000 elements took 2432241800 nanoseconds
Insertion of 2000000 elements took 317752100 nanoseconds
removal of 2000000 elements took 3407677500 nanoseconds
Insertion of 2500000 elements took 420124400 nanoseconds
removal of 2500000 elements took 4389531700 nanoseconds
Insertion of 3000000 elements took 479193600 nanoseconds
removal of 3000000 elements took 5547313400 nanoseconds
```

3. Binary Search Tree

```
#include <iostream>
#include <vector>
#include "gettime.h"
using namespace std;

class BSTnode
{
public:
    int value;
    BSTnode *left = NULL;
    BSTnode *right = NULL;

    BSTnode(int value) { this->value = value; }

    void addChild(int v)
```

```
{
    BSTnode *child = new BSTnode(v);
    addChild(child);
}
void addChild(BSTnode *child)
{
    if (child)
    {
        if (child->value == this->value)
        {
            return;
        }
        if (child->value < this->value)
        {
            if (this->left)
            {
                this->left->addChild(child);
            }
            else
            {
                this->left = child;
            }
        }
        else
        {
            if (this->right)
            {
                this->right->addChild(child);
            }
            else
            {
                this->right = child;
            }
        }
    }
}
BSTnode *getMin()
{
    if (this->left != NULL)
    {
        return this->left->getMin();
    }
    else
    {
        return this;
    }
}
BSTnode *getMax()
```

```
    {
        if (this->right)
        {
            return this->right->getMax();
        }
        else
        {
            return this;
        }
    }
    void deleteChild(int v)
    {
        BSTnode *node = search_iter(v);
        BSTnode *parent = getParent(node);
        if (node && parent)
        {
            // node is leaf
            if (node->left == NULL && node->right == NULL)
            {
                if (parent->right == node)
                {
                    parent->right = NULL;
                }
                else
                {
                    parent->left = NULL;
                }
                free(node);
                return;
            }
            // node has both left and right
            if (node->left != NULL && node->right != NULL)
            {
                BSTnode *minNode = node->right->getMin();
                BSTnode *minParent = this->getParent(minNode);
                if (minParent->left == minNode)
                {
                    minParent->left = NULL;
                }
                else
                {
                    minParent->right = NULL;
                }
                BSTnode *nodeRight = node->right == minNode ? NULL : node-
>right;
                BSTnode *nodeLeft = node->left == minNode ? NULL : node-
>left;
                if (parent->right == node)
```

```
                {
                    parent->right = NULL;
                }
                else
                {
                    parent->left = NULL;
                }
                minNode->addChild(nodeRight);
                minNode->addChild(nodeLeft);
                parent->addChild(minNode);
                free(node);
                return;
            }
            // node has only one child
            if (node->left == NULL || node->right == NULL)
            {
                BSTnode *nodeChild = node->left ? node->left : node->right;
                if (parent->right == node)
                {
                    parent->right = NULL;
                }
                else
                {
                    parent->left = NULL;
                }
                parent->addChild(nodeChild);
                free(node);
                return;
            }
        }
    }
    BSTnode *getParent(BSTnode *n)
    {
        if (n)
        {
            if (n == this)
            {
                return NULL;
            }
            if (this->left == n || this->right == n)
            {
                return this;
            }
            if (this->left)
            {
                BSTnode *checkLeft = this->left->getParent(n);
                if (checkLeft)
                {
```

```
                return checkLeft;
            }
        }
        if (this->right)
        {
            BSTnode *checkRight = this->right->getParent(n);
            if (checkRight)
            {
                return checkRight;
            }
        }
    }
    return NULL;
}
BSTnode *search(int v)
{
    if (this->value == v)
    {
        return this;
    }
    if (v > this->value)
    {
        if (this->right)
        {
            BSTnode *rSearch = this->right->search(v);

            return rSearch;
        }
    }
    if (v < this->value)
    {
        if (this->left)
        {
            BSTnode *lSearch = this->left->search(v);

            return lSearch;
        }
    }
    return NULL;
}
BSTnode *search_iter(int v)
{
    BSTnode *current = this;
    while (current != NULL)
    {
        if (current->value == v)
        {
            return current;
```

```
            }
            if (v > current->value)
            {
                current = current->right;
            }
            else
            {
                current = current->left;
            }
        }
        return NULL;
    }

    int size(int initSize = 0)
    {
        int s = initSize + 1;
        if (this->left)
        {
            s = this->left->size(s);
        }
        if (this->right)
        {
            s = this->right->size(s);
        }
        return s;
    }
};

int main()
{
    vector<int> sizes;
    int start_size = 1000000;
    int increment = 500000;
    sizes.push_back(start_size);
    // srand(time(NULL));
    for (size_t i = 0; i < 4; i++)
    {
        sizes.push_back(sizes.back() + increment);
    }
    // insert random values
    for (size_t i = 0; i < sizes.size(); i++)
    {
        int size = sizes[i];
        BSTnode *root = new BSTnode(rand());
        long long time = getTime([&]()
                                 {
                                     for (size_t i = 0; i < size; i++)
                                     {
```

```
                                        root->addChild(rand());
                                    } });
        cout << "BST insert " << size << " nodes: " << time << "ns" <<
endl;
        time = getTime([&]()
                        {
                            for (size_t i = 0; i < size; i++)
                            {
                                root->search(rand());
                            } });
        cout << "BST search recursive " << size << " nodes: " << time <<
"ns" << endl;
        time = getTime([&]()
                        {
                            for (size_t i = 0; i < size; i++)
                            {
                                root->search_iter(rand());
                            } });
        cout << "BST search iter " << size << " nodes: " << time << "ns" <<
endl;
        time = getTime([&]()
                        {
                            for (size_t i = 0; i < size; i++)
                            {
                                root->deleteChild(rand());
                            } });
        cout << "BST delete " << size << " nodes: " << time << "ns" <<
endl;

        cout << "----------" << endl;
    }
}
```

Output:

```
BST insert 1000000 nodes: 987869200ns
BST search recursive 1000000 nodes: 275361200ns
BST search iter 1000000 nodes: 277612500ns
BST delete 1000000 nodes: 8908388500ns
----------
BST insert 1500000 nodes: 1636738400ns
BST search recursive 1500000 nodes: 613827600ns
BST search iter 1500000 nodes: 530228600ns
BST delete 1500000 nodes: 11334035400ns
----------
BST insert 2000000 nodes: 2199068100ns
BST search recursive 2000000 nodes: 861690600ns
BST search iter 2000000 nodes: 797249600ns
BST delete 2000000 nodes: 13220617800ns
----------
BST insert 2500000 nodes: 2920991400ns
BST search recursive 2500000 nodes: 1130717200ns
BST search iter 2500000 nodes: 1035156600ns
BST delete 2500000 nodes: 13116876100ns
----------
BST insert 3000000 nodes: 3422782200ns
BST search recursive 3000000 nodes: 1563222300ns
BST search iter 3000000 nodes: 1269990500ns
BST delete 3000000 nodes: 14304810900ns
----------
```

Conclusion:

In this way, we implemented priority queue, stack and binary search trees using C++.