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.
    - **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.
    - **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.
    - **Insertion 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.
    - **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.
    - **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.

```
long long getTime(std::function<void()> f){
        auto start = clock();
```

```
        f();
        auto end = clock();
        long double duration = end - start;
        return (duration/CLOCKS_PER_SEC) * 1000000000;
}
```

1. Stack

```
#include <iostream>
#include <chrono>
#include <functional>
using namespace std;

class Stack{
    int *data;
    int top;
    int size;

    public:
    Stack(int size){
        this->size = size;
        data = (int *)(calloc(this->size, sizeof(int)));
        top = 0;
    }

    void push(int item){
        if(top + 1 <= size){
            data[top] = item;
            top++;
        }
        else{
            cout << "Stack is full" << endl;
        }

    }

    int pop(){
        if(top == 0){
            cout << "Stack is empty" << endl;
            return 0;
        }
        int tdata = data[top-1];
        top--;
        return tdata;
    }

    int peek(){
        return data[top-1];
    }
};

int main(){
    Stack s(10000000);
    auto push = [&](){
        for(int i = 0; i < 10000000; i++){
            s.push(i);
        }
    };

    auto pop = [&](){
        for(int i = 0; i < 10000000; i++){
            s.pop();
        }
    };
    cout << "Time taken to push 10000000 elements:" << getTime(push) << "ns" <<
    endl;
    cout << "Time taken to pop 10000000 elements:" << getTime(pop) << "ns" <<
    endl;
}
```

Output:

```
● ➜ lab1 git:(main) ✗ g++ stack.cpp -o stack.out
● ➜ lab1 git:(main) ✗ ./stack.out
  Time taken to push 10000000 elements:57015000ns
  Time taken to pop 10000000 elements:28071000ns
```

2. Priority Queue:

```cpp
#include <iostream>
#include <ctime>
#include <vector>
#include <functional>

using namespace std;

template <typename T>


class PriorityQueue {
    vector<T> data;

    public:
    PriorityQueue(){
    }

    void enqueue(T item){
        data.push_back(item);
        int c = data.size() - 1;

        while (c > 0){
            int p = (c - 1) / 2;
            if (data[c] <= data[p])
                break;
            T temp = data[p];
            data[p] = data[c];
            data[c] = temp;
            c = p;
        }
    }

    T dequeue(){
        int l = data.size() - 1;
        T fi = data[0];
        data[0] = data[l];
        data.pop_back();
        --l;

        int p = 0;

        while (true){
            int c = 2 * p + 1;
            if (c > l)
                break;

            int rchild = c + 1;
            if (rchild <= l && data[rchild] > data[c])
                c = rchild;

            if (data[p] >= data[c])
                break;

            T temp  = data[p];
            data[p] = data[c];
            data[c] = temp;
            p = c;
        }
        return fi;
    }
};
```

```
int main(){
    PriorityQueue<int> pq;
    auto enqueue = [&](){
        for(int i = 0; i < 1000000; i++){
            pq.enqueue(i);
        }
    };

    auto dequeue = [&](){
        for(int i = 0; i < 1000000; i++){
            pq.dequeue();
        }
    };

    cout << "Time taken to enqueue 1000000 elements: " << getTime(enqueue) <<
    "ns" << endl;
    cout << "Time taken to dequeue 1000000 elements: " << getTime(dequeue) <<
    "ns" << endl;
}
```

Output:

```
➜ lab1 git:(main) ✗ g++ priorityQueue.cpp -o queue.out
➜ lab1 git:(main) ✗ ./queue.out
Time taken to enqueue 10000000 elements: 2007420000ns
Time taken to dequeue 10000000 elements: 2886036000ns
```

3. Binary Search Tree:

```
#include <iostream>
#include <functional>

using namespace std;

class BSTree{
    public:
    struct Node{
        int data;
        Node *left;
        Node *right;> [!PDF|] [[lab1.pdf#page=1&selection=10,1,19,6|lab1, p.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. Che
>
>
    };

    Node *createNode(int data){
        Node *newNode = new Node();
        newNode->data = data;
        newNode->left = newNode->right = NULL;
        return newNode;
    }

    Node *insert(Node *root, int data){
        if(root == NULL){
            root = createNode(data);
            return root;
        }
        if(data < root->data){
            root->left = insert(root->left, data);
        }
        else if(data > root->data){
            root->right = insert(root->right, data);
        }
        return root;
    }

    Node *search(Node *root, int data){
        if(root == NULL || root->data == data){
```

```
            return root;
        }
        if(root->data < data){
            return search(root->right, data);
        }
        return search(root->left, data);
    }

    Node *searchIterative(Node *root, int data){
        while(root != NULL && root->data != data){
            if(root->data < data){
                root = root->right;
            }
            else{
                root = root->left;
            }
        }
        return root;
    }
};

int main(){
    auto insertAndSearch = [&](){
        BSTree bst;
        BSTree::Node *root = NULL;
        bst.createNode(4);
        bst.insert(root, 2);
        bst.insert(root, 10);
        bst.insert(root, 1);
        bst.insert(root, 5);
        bst.insert(root, 6);
        bst.insert(root, 7);
        bst.insert(root, 8);
        BSTree::Node *search = bst.search(root, 3);
        BSTree::Node *searchIterative = bst.searchIterative(root, 3);
        searchIterative ? cout << "Element found" << endl : cout << "Element not
         found" << endl;
    };
    cout << "Time taken to insert 8 elements and search: \n" <<
    getTime(insertAndSearch) << "ns" << endl;
}
```

Output:

```
➜ lab1 git:(main) ✗ g++ binarySearchTree.cpp -o bst.out
➜ lab1 git:(main) ✗ ./bst.out
Time taken to insert 100 elements: 11000ns
Time taken to search for an element: Element not found
2000ns
```

Conclusion:

In this way we can implement stack, priority queue and binary search tree.