**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе 1**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Ассоциативный массив

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Касаткин А.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2022

Задача: Реализовать шаблонный ассоциативный массив (map) на основе красно-

черного дерева.

Описание методов:

**insert**(ключ, значение) // добавление элемента с ключом и значением

Если в дереве есть элементы, то двигаемся до null(элемент дерева которые имеет ключ 0 и

значение 0 и является висящими вершинами). Если ключ больше, то вправо, иначе - влево.

Вместо null ставит новый элемент с ключем, значением и красным цветом. И все его дети

становятся null.

Потом происходит восстановление дерева.

1. Выполняем следующие действия, пока родитель p узла newNode

красный

2. Если p является левым потомком grandParent для newNode,

то выполняем действия в соответствии с 3 разными случаями

1 случай:

a) Если цвет правого потомка grandParent узла newNode красный, перекрашиваем обоих потомков grandParent в

черный, а цвет grandParent меняем на красный

b) Обозначаем за newNode узел grandParent

Случай-II: проверяем условие из п.1. Если условие НЕ выполняется, то переходим к п. 4.

в) если newNode является правым потомком p. Если да, то назначаем p в качестве newNode

г) выполняем левый поворот для newNode

Случай-III: проверяем условие из п.1. Если условие НЕ выполняется, топереходим к п. 4.

д) устанавливаем цвет p как черный, а grandParent перекрашиваем в красный

е) выполняем правый поворот для grandParent

3. Иначе (если p является правым потомком grandParent для newNode) выполняем следующее:

a) Если цвет левого потомка grandParent узла newNode - красный, то устанавливаем цвет обоих

потомков grandParent как черный, а цвет grandParent - как красный (симметрично случаю I). Назначаем grandParent в

качестве newNode

b) Иначе, если newNode является левым потомком p, то назначаем p в качестве newNode и

выполняем правый поворот для newNode

c) Перекрашиваем p в черный, а grandParent в красный

d) Выполняем левый поворот grandParent

4. Этот шаг выполняется после выхода из цикла п. 1-3 устанавливаем цвет корня дерева как черный

**remove**(ключ) // удаление элемента по ключу

Если в дереве есть элементы, то двигаемся до нужного элемента. Если ключ больше, то вправо,

иначе - влево.

1. Пусть узел nodeToBeDeleted будет удаляться из дерева

2. Сохраняем цвет nodeToBeDeleted в origrinalColor

3. Если левый потомок nodeToBeDeleted равен NULL

a) Обозначаем правый потомок nodeToBeDeleted за x

б) Перемещаем x на место nodeToBeDeleted

4. Иначе если правый потомок nodeToBeDeleted равен NULL (левый потомок nodeToBeDeleted

не равен NULL, а правый равен NULL):

а) Обозначаем за x левого потомка nodeToBeDeleted

б) Перемещаем x на место nodeToBeDeleted

5. Иначе (оба потомка nodeToBeDeleted не равны NULL)

а) Обозначаем за y минимальное значение из правого поддерева nodeToBeDeleted (самый

левый элемент в правом поддереве)

б) Сохраняем цвет y в origrinalColor

в) Обозначаем за x правого потомка y

г) если y это ребенок nodeToBeDeleted, тогда устанавливаем y родителем x

д) иначе (y не ребенок nodeToBeDeleted) перемещаем правого потомка y на место y

е) перемещаем y на место nodeToBeDeleted

ж) устанавливаем для y цвет origrinalColor

6. Если цвет origrinalColor черный, то вызываем алгоритм восстановления свойств после

удаления

1. Выполняем следующие действия, пока x не станет корнем дерева, а цвет x станет черным

2. Если x - левый потомок, то

a) Обозначаем за w брата x

b) Если w красный

Случай-I:

- Перекрашиваем w в черный

- Перекрашиваем родителя x в красный

-Выполняем левый поворот для родителя x

-Обозначаем rightChild родителя x за w

c) Если цвет правого и левого потомка w черный

Случай-II:

- Перекрашиваем w в красный

- Назначаем в качестве x родителя x

d) Иначе, если цвет rightChild узла w черный

Случай-III:

- Перекрашиваем leftChild узла w в черный

- Перекрашиваем w в красный

-Выполняем правый поворот для w

-Обозначаем rightChild родителя x за w

e) Если из вышеперечисленных случаев не происходит ни один

случай, то выполняем

Случай-IV:

- Перекрашиваем w в цвет родителя x

- Перекрашиваем родителя x в черный

- Перекрашиваем правого потомка w в черный

- Выполняем левый поворот для родителя x

- Обозначаем за x корень дерева

3. Иначе к п.2. (x - правый потомок) выполняем все те же действия (начиная со слайда 35), что

и в предыдущих пунктах, только изменяем направление вращений левое на правое и наоборот

4. Перекрашиваем x в черный

**find**(ключ) // поиск элемента по ключу

Если в дереве есть элементы, то двигаемся до нужного элемента. Если ключ больше, то вправо,

иначе - влево.

Если искомый элемент null, то ошибка, иначе возвращает значение искомого элемента.

**clear** // очищение ассоциативного массива

Пока голова есть и не равна null, то удаляем ключ головы.

**get\_keys** // возвращает список ключей

Производит метод обхода в глубину и каждый раз, когда указатель спускается его ключ

добавляется в список

**get\_values** // возвращает список значений

Производит метод обхода в глубину и каждый раз, когда указатель спускается его значение

добавляется в список

**print** // вывод в консоль

Выводит в консоль ключ и значение.

Оценка временной сложности:

|  |  |  |
| --- | --- | --- |
| insert(ключ, значение) | добавление элемента с ключом и значением | O(log n) |
| remove(ключ) | удаление элемента по ключу | O(log n) |
| find(ключ) | поиск элемента по ключу | O(log n) |
| clear | очищение ассоциативного массива | O(n) |
| get\_keys | возвращает список ключей | O(n^2) |
| get\_values | возвращает список значений | O(n^2) |
| print | вывод в консоль | O(n^2) |

Описание unit-тестов:

**Test\_insert**

Просто добавляет элемент.

Пытается добавить уже существующий.

Проверяет все элементы с помощью метода find.

**Test\_remove**

Просто удаляет.

И пытается удалить элемент с пустого дерева.

**Test\_find**

Находит существующий элемент.

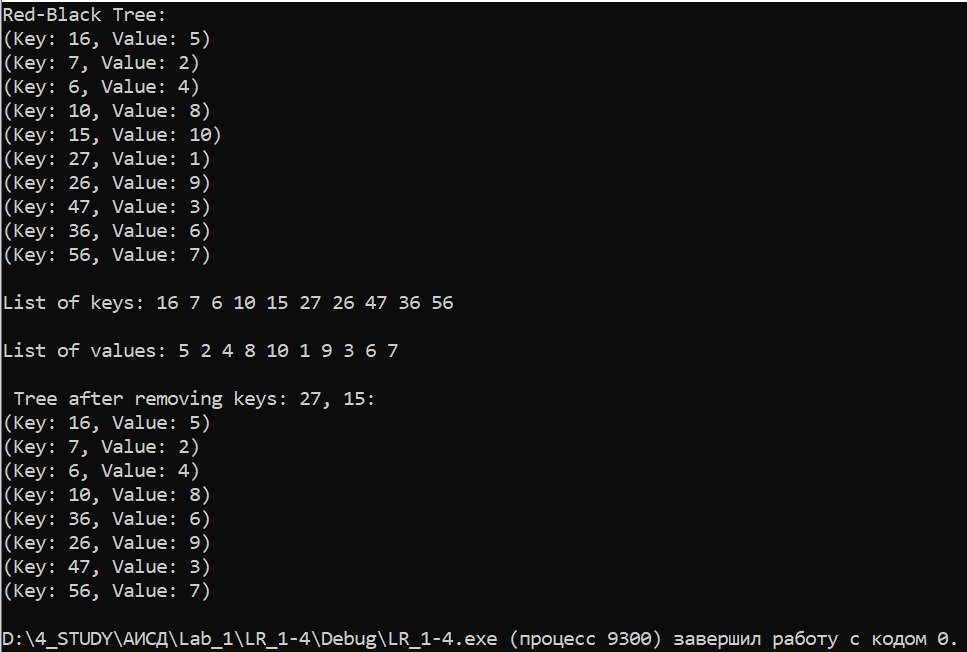
И пытается найти несуществующий.

**Test\_getvalues**

Создает дерево с одинаковыми ключом и значением.

И смотрит есть ли они в списке.

Результат работы программы:



Листинг:

**Main.cpp**

#include <iostream>

#include "RBTree.h"

using namespace std;

int main() {

RedBlackTree<int, int> tree;

tree.insert(27, 1);

tree.insert(7, 2);

tree.insert(47, 3);

tree.insert(6, 4);

tree.insert(16, 5);

tree.insert(36, 6);

tree.insert(56, 7);

tree.insert(10, 8);

tree.insert(26, 9);

tree.insert(15, 10);

cout << "Red-Black Tree:\n";

tree.printTree();

cout << "\nList of keys: ";

Element\_List<int>\* temp = tree.getKeys()->head;

while (temp != NULL) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

cout << "\nList of values: ";

temp = tree.getValues()->head;

while (temp != NULL) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

cout << "\n Tree after removing keys: 27, 15:\n";

tree.remove(27);

tree.remove(15);

tree.printTree();

tree.clear();

}

**List.h**

#pragma once

#include <iostream>

using namespace std;

template<class T>

class Element\_List {

public:

T data;

Element\_List\* prev;

Element\_List\* next;

Element\_List() {

this->data = 0;

this->prev = nullptr;

this->next = nullptr;

}

};

template<class T>

class List {

public:

Element\_List<T>\* head;

Element\_List<T>\* tail;

List() {

this->head = nullptr;

this->tail = nullptr;

}

bool isEmpty();

void pushback(T data);

void pushfront(T data);

void popback();

void popfront();

void clear();

};

template<class T>

inline bool List<T>::isEmpty()

{

if (head == nullptr && tail == nullptr)

return true;

else if (head != nullptr && tail != nullptr)

return false;

else

throw logic\_error("Error");

}

template<class T>

inline void List<T>::pushback(T data)

{

if (isEmpty()) {

Element\_List<T>\* Node = new Element\_List<T>();

Node->data = data;

head = Node;

tail = Node;

}

else {

Element\_List<T>\* Node = new Element\_List<T>();

Node->data = data;

tail->next = Node;

Node->prev = tail;

tail = Node;

}

}

template<class T>

inline void List<T>::pushfront(T data)

{

if (isEmpty()) {

Element\_List<T>\* Node = new Element\_List<T>();

Node->data = data;

head = Node;

tail = Node;

}

else {

Element\_List<T>\* Node = new Element\_List<T>();

Node->data = data;

head->prev = Node;

Node->next = head;

head = Node;

}

}

template<class T>

inline void List<T>::popback()

{

if (isEmpty())

throw logic\_error("List is empty.");

else {

if (tail->prev != nullptr) {

tail = tail->prev;

delete tail->next;

tail->next = nullptr;

}

else {

delete tail;

head = nullptr;

tail = nullptr;

}

}

}

template<class T>

inline void List<T>::popfront()

{

if (isEmpty())

throw logic\_error("List is empty.");

else {

if (head->next != nullptr) {

head = head->next;

delete head->prev;

head->prev = nullptr;

}

else {

delete head;

head = nullptr;

tail = nullptr;

}

}

}

template<class T>

inline void List<T>::clear()

{

Element\_List<T>\* Node = head;

while (Node != nullptr) {

Node = Node->next;

delete head;

head = Node;

}

}

**RBTree.h**

#pragma once

#include<iostream>

#include "List.h"

#include "Stack.h"

using namespace std;

enum class RBColor {

RED,

BLACK

};

template<class T1, class T2>

class Element\_RBtree {

public:

RBColor color;

T1 key;

T2 value;

Element\_RBtree\* parent;

Element\_RBtree\* left;

Element\_RBtree\* right;

Element\_RBtree() {

this->color = RBColor::RED;

this->key = 0;

this->value = 0;

this->parent = nullptr;

this->left = nullptr;

this->right = nullptr;

}

};

template<class T1, class T2>

class RedBlackTree {

public:

Element\_RBtree<T1, T2>\* root;

Element\_RBtree<T1, T2>\* nil;

RedBlackTree() {

nil = new Element\_RBtree<T1, T2>();

nil->color = RBColor::BLACK;

root = nil;

root->left = nil;

root->right = nil;

}

void leftTurn(Element\_RBtree<T1, T2>\* xNode);

void rightTurn(Element\_RBtree<T1, T2>\* yNode);

void insert(T1 key, T2 value);

void rebalancingTreeAfterInsertion(Element\_RBtree<T1, T2>\* Node);

Element\_RBtree<T1, T2>\* find(T1 key);

void remove(T1 key);

void rebalancingTreeAfterRemove(Element\_RBtree<T1, T2>\* Node);

List<Element\_RBtree<T1, T2>\*>\* allNodesToList();

void printTree();

List<T1>\* getKeys();

List<T2>\* getValues();

void clear();

};

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::leftTurn(Element\_RBtree<T1, T2>\* xNode)

{

Element\_RBtree<T1, T2>\* yNode = xNode->right;

Element\_RBtree<T1, T2>\* bNode = yNode->left;

xNode->right = bNode;

if (bNode != nil)

bNode->parent = xNode;

Element\_RBtree<T1, T2>\* pNode = xNode->parent;

yNode->parent = pNode;

if (pNode == nullptr)

root = yNode;

else {

if (pNode->left == xNode)

pNode->left = yNode;

else if (pNode->right == xNode)

pNode->right = yNode;

}

yNode->left = xNode;

xNode->parent = yNode;

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::rightTurn(Element\_RBtree<T1, T2>\* yNode)

{

Element\_RBtree<T1, T2>\* xNode = yNode->left;

Element\_RBtree<T1, T2>\* bNode = xNode->right;

yNode->left = bNode;

if (bNode != nil)

bNode->parent = yNode;

Element\_RBtree<T1, T2>\* pNode = yNode->parent;

xNode->parent = pNode;

if (pNode == nullptr)

root = xNode;

else {

if (pNode->left == yNode)

pNode->left = xNode;

else if (pNode->right == yNode)

pNode->right = xNode;

}

xNode->right = yNode;

yNode->parent = xNode;

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::insert(T1 key, T2 value)

{

if (root == nil) {

Element\_RBtree<T1, T2>\* Node = new Element\_RBtree<T1, T2>;

Node->key = key;

Node->value = value;

Node->color = RBColor::BLACK;

Node->left = nil;

Node->right = nil;

root = Node;

}

else {

Element\_RBtree<T1, T2>\* Node = root;

while (Node->left != nil || Node->right != nil) {

if (key == Node->key)

throw logic\_error("This key is exist.");

else if (key < Node->key) {

if (Node->left != nil)

Node = Node->left;

else break;

}

else {

if (Node->right != nil)

Node = Node->right;

else break;

}

}

Element\_RBtree<T1, T2>\* newNode = new Element\_RBtree<T1, T2>;

newNode->key = key;

newNode->value = value;

newNode->left = nil;

newNode->right = nil;

newNode->parent = Node;

if (newNode->key > Node->key)

Node->right = newNode;

else

Node->left = newNode;

newNode->color = RBColor::RED;

rebalancingTreeAfterInsertion(newNode);

}

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::rebalancingTreeAfterInsertion(Element\_RBtree<T1, T2>\* Node)

{

while (Node->parent->color == RBColor::RED) {

Element\_RBtree<T1, T2>\* Parent = Node->parent;

Element\_RBtree<T1, T2>\* grandParent = Parent->parent;

if (grandParent->left == Parent) {

if (grandParent->right->color == RBColor::RED) {

grandParent->left->color = RBColor::BLACK;

grandParent->right->color = RBColor::BLACK;

grandParent->color = RBColor::RED;

Node = grandParent;

}

else {

if (Parent->right == Node) {

Node = Parent;

leftTurn(Node);

Parent = Node->parent;

grandParent = Parent->parent;

}

Parent->color = RBColor::BLACK;

grandParent->color = RBColor::RED;

rightTurn(grandParent);

}

}

else if (grandParent->right == Parent) {

if (grandParent->left->color == RBColor::RED) {

grandParent->left->color = RBColor::BLACK;

grandParent->right->color = RBColor::BLACK;

grandParent->color = RBColor::RED;

Node = grandParent;

}

else {

if (Parent->left == Node) {

Node = Parent;

rightTurn(Node);

Parent = Node->parent;

grandParent = Parent->parent;

}

Parent->color = RBColor::BLACK;

grandParent->color = RBColor::RED;

leftTurn(grandParent);

}

}

if (Node == root)

break;

}

root->color = RBColor::BLACK;

}

template<class T1, class T2>

inline Element\_RBtree<T1, T2>\* RedBlackTree<T1, T2>::find(T1 key)

{

if (root == nil) {

throw logic\_error("Tree is empty.");

}

else {

Element\_RBtree<T1, T2>\* Node = root;

bool check = 0;

while (Node != nil) {

if (key == Node->key) {

check = 1;

break;

}

else if (key < Node->key)

Node = Node->left;

else

Node = Node->right;

}

if (check == 0)

throw invalid\_argument("Key doen't exist in the tree.");

else

return Node;

}

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::remove(T1 key)

{

if (root == nil)

throw logic\_error("Tree is empty.");

else {

Element\_RBtree<T1, T2>\* Node = find(key);

Element\_RBtree<T1, T2>\* xNode;

RBColor originalColor = Node->color;

if (Node->left == nil) {

delete Node->left;

xNode = Node->right;

if (Node->parent->left == Node) {

Node->parent->left = xNode;

}

else if (Node->parent->right == Node) {

Node->parent->right = xNode;

}

xNode->parent = Node->parent;

}

else if (Node->right == nil) {

delete Node->right;

xNode = Node->left;

if (Node->parent->left == Node) {

Node->parent->left = xNode;

}

else if (Node->parent->right == Node) {

Node->parent->right = xNode;

}

xNode->parent = Node->parent;

}

else {

Element\_RBtree<T1, T2>\* yNode = Node->right;

while (yNode->left != nil) {

yNode = yNode->left;

}

if (yNode == Node->right) {

originalColor = Node->color;

}

else {

originalColor = yNode->color;

}

xNode = yNode->right;

if (yNode->parent == Node) {

xNode->parent = yNode;

yNode->left = Node->left;

yNode->right = xNode;

}

else {

if (yNode->parent->right == yNode) {

yNode->parent->right = xNode;

}

else if (yNode->parent->left == yNode) {

yNode->parent->left = xNode;

}

yNode->left = Node->left;

yNode->right = Node->right;

}

if (Node->parent->right == Node) {

Node->parent->right = yNode;

}

else if (Node->parent->left == Node) {

Node->parent->left = yNode;

}

yNode->parent = Node->parent;

yNode->color = originalColor;

}

if (originalColor == RBColor::BLACK) {

rebalancingTreeAfterRemove(xNode);

}

delete Node;

}

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::rebalancingTreeAfterRemove(Element\_RBtree<T1, T2>\* Node)

{

Element\_RBtree<T1, T2>\* wNode;

while (Node != root && Node->color != RBColor::BLACK) {

if (Node->parent->left = Node) {

wNode = Node->parent->right;

if (wNode->color == RBColor::RED) {

wNode->color = RBColor::BLACK;

Node->parent->color = RBColor::RED;

leftTurn(Node->parent);

wNode = Node->parent->right;

}

else if (wNode->left->color == RBColor::BLACK && wNode->right->color == RBColor::BLACK) {

wNode->color = RBColor::RED;

Node = Node->parent;

}

else if (wNode->right->color == RBColor::BLACK) {

wNode->left->color = RBColor::BLACK;

wNode->color = RBColor::RED;

rightTurn(wNode);

wNode = Node->parent->right;

}

else {

wNode->color = Node->parent->color;

Node->parent->color = RBColor::BLACK;

wNode->right->color = RBColor::BLACK;

leftTurn(Node->parent);

Node = root;

}

}

else if (Node->parent->right = Node) {

wNode = Node->parent->left;

if (wNode->color == RBColor::RED) {

wNode->color = RBColor::BLACK;

Node->parent->color = RBColor::RED;

rightTurn(Node->parent);

wNode = Node->parent->left;

}

else if (wNode->left->color == RBColor::BLACK && wNode->right->color == RBColor::BLACK) {

wNode->color = RBColor::RED;

Node = Node->parent;

}

else if (wNode->left->color == RBColor::BLACK) {

wNode->right->color = RBColor::BLACK;

wNode->color = RBColor::RED;

leftTurn(wNode);

wNode = Node->parent->left;

}

else {

wNode->color = Node->parent->color;

Node->parent->color = RBColor::BLACK;

wNode->left->color = RBColor::BLACK;

rightTurn(Node->parent);

Node = root;

}

}

Node->color = RBColor::BLACK;

}

}

template<class T1, class T2>

inline List<Element\_RBtree<T1, T2>\*>\* RedBlackTree<T1, T2>::allNodesToList()

{

List<Element\_RBtree<T1, T2>\*>\* list = new List<Element\_RBtree<T1, T2>\*>;

Stack<Element\_RBtree<T1, T2>\*>\* stack = new Stack<Element\_RBtree<T1, T2>\*>;

Element\_RBtree<T1, T2>\* temp = root;

while (temp != nil) {

list->pushback(temp);

if (temp->right != nil)

stack->push(temp->right);

if (temp->left != nil)

temp = temp->left;

else {

if (!stack->isEmpty())

temp = stack->pop();

else

temp = nil;

}

}

return list;

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::printTree()

{

if (root == nil)

cout << "Tree is empty.";

else {

List<Element\_RBtree<T1, T2>\*>\* listNode = allNodesToList();

while (listNode->head != nullptr) {

cout << "(Key: " << listNode->head->data->key << ", Value: " << listNode->head->data->value << ")\n";

listNode->popfront();

}

}

}

template<class T1, class T2>

inline List<T1>\* RedBlackTree<T1, T2>::getKeys()

{

List<T1>\* listKey = new List<T1>;

List<Element\_RBtree<T1, T2>\*>\* listNode = allNodesToList();

Element\_List<Element\_RBtree<T1, T2>\*>\* temp = listNode->head;

while (temp != nullptr) {

listKey->pushback(temp->data->key);

temp = temp->next;

}

return listKey;

}

template<class T1, class T2>

inline List<T2>\* RedBlackTree<T1, T2>::getValues()

{

List<T2>\* listValue = new List<T2>;

List<Element\_RBtree<T1, T2>\*>\* listNode = allNodesToList();

Element\_List<Element\_RBtree<T1, T2>\*>\* temp = listNode->head;

while (temp != nullptr) {

listValue->pushback(temp->data->value);

temp = temp->next;

}

return listValue;

}

template<class T1, class T2>

inline void RedBlackTree<T1, T2>::clear()

{

if (root == nil)

throw logic\_error("The tree is already empty.");

else {

Element\_RBtree<T1, T2>\* current = root;

Stack<Element\_RBtree<T1, T2>\*>\* stack = new Stack<Element\_RBtree<T1, T2>\*>;

while (current != nil) {

if (current->right != nil)

stack->push(current->right);

if (current->left != nil) {

Element\_RBtree<T1, T2>\* temp = current->left;

delete current;

current = temp;

}

else if (!stack->isEmpty()) {

delete current;

current = stack->pop();

}

else {

delete current;

current = nil;

}

}

root = current;

}

}

**Stack.h**

#pragma once

using namespace std;

template<class T>

class Stack {

private:

class Element\_Stack {

public:

T data;

Element\_Stack\* next;

};

Element\_Stack\* top;

size\_t size;

public:

Stack() {

top = nullptr;

size = 0;

}

~Stack() = default;

void push(T x);

T pop();

bool isEmpty();

size\_t getsize();

};

template<class T>

inline void Stack<T>::push(T x)

{

if (isEmpty()) {

Element\_Stack\* Node = new Element\_Stack;

Node->data = x;

Node->next = nullptr;

top = Node;

}

else {

Element\_Stack\* Node = new Element\_Stack;

Node->data = x;

Node->next = top;

top = Node;

}

size++;

}

template<class T>

inline T Stack<T>::pop()

{

if (isEmpty())

throw runtime\_error("Stack is empty.");

else {

Element\_Stack\* temp = top;

if (top->next == nullptr)

top = nullptr;

else

top = temp->next;

T result = temp->data;

delete temp;

size--;

return result;

}

}

template<class T>

inline bool Stack<T>::isEmpty()

{

if (top == nullptr)

return true;

else

return false;

}

template<class T>

inline size\_t Stack<T>::getsize()

{

return size;

}