Федеральное государственное автономное образовательное учреждение высшего образования «Пермский национальный исследовательский политехнический университет», ПНИПУ

ЛАБОРАТОРНАЯ РАБОТА

ИНФОРМАЦИОННЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Выполнил: студент группы РИС-23-3б

Артем Владимирович Швецов

Проверила: доцент кафедры ИТАС

Ольга Андреевна Полякова

Пермь 2024

**Постановка задачи**

Написать программу, в которой создаются динамические структуры и выполнить их обработку в соответствии со своим вариантом.

**Для каждого вариант разработать следующие функции:**

* Создание списка.
* Добавление элемента в список (в соответствии со своим вариантом).
* Удаление элемента из списка (в соответствии со своим вариантом).
* Печать списка.
* Запись списка в файл.
* Уничтожение списка.
* Восстановление списка из файла.

**Анализ задачи**

* Для создания элементов списка используем структуру Node, в экземплярах которой будем хранить в указателях адреса соседних элементов;
* Для создания самого списка используем ещё одну структуру – List, в указателях которой будем хранить адреса первого и последнего элементов списка;
* Каждый указатель каждой структуры по умолчанию будет иметь значение nullptr, используем это значение как обозначение начала/конца списка;
* Для вывода, записи в файл и прочих операций со списком будем переходить от элемента к элементу по их адресам, начиная с адреса головного (первого) элемента списка.

**Блок-схема**

Опущена, логика выполнения линейная.

**Код**

#include <iostream>

#include <string>

#include <fstream>

using namespace std;

struct Node {

char& key = \*(new char[1]);

Node\* ptr\_to\_prev\_node = nullptr;

Node\* ptr\_to\_next\_node = nullptr;

};

struct List {

Node\* head\_node = nullptr;

Node\* tail\_node = nullptr;

int size = 0;

};

void delete\_key(List\* list, char key) {

if (list == nullptr) {

cout << "List is not existing" << endl;

return;

}

if (list->head\_node == nullptr) {

cout << "List is empty" << endl;

return;

}

Node\* remove\_node = list->head\_node;

while ((remove\_node->key != key) && (remove\_node != nullptr)) {

remove\_node = remove\_node->ptr\_to\_next\_node;

}

if (remove\_node == nullptr) {

cout << "No key found" << endl;

return;

}

//???

if ((list->head\_node == remove\_node) && (list->tail\_node == remove\_node)) {

list->head\_node = nullptr;

list->tail\_node = nullptr;

}

else if (list->head\_node == remove\_node) {

list->head\_node = remove\_node->ptr\_to\_next\_node;

remove\_node->ptr\_to\_next\_node->ptr\_to\_prev\_node = nullptr;

}

else if (list->tail\_node == remove\_node) {

list->tail\_node = remove\_node->ptr\_to\_prev\_node;

remove\_node->ptr\_to\_prev\_node->ptr\_to\_next\_node = nullptr;

}

else {

remove\_node->ptr\_to\_next\_node->ptr\_to\_prev\_node = remove\_node->ptr\_to\_prev\_node;

remove\_node->ptr\_to\_prev\_node->ptr\_to\_next\_node = remove\_node->ptr\_to\_next\_node;

}

delete remove\_node;

}

void add\_node(List\* list, char str, const int& index = -1) {

if (list == nullptr) {

cout << "List is not existing" << endl;

return;

}

Node\* new\_node = new Node;

new\_node->key = str;

list->size++;

if (list->head\_node == nullptr) {

list->head\_node = new\_node;

list->tail\_node = new\_node;

return;

}

if ((index == -1) || (index == 0)) {

new\_node->ptr\_to\_next\_node = list->head\_node;

list->head\_node->ptr\_to\_prev\_node = new\_node;

list->head\_node = new\_node;

return;

}

int counter = 0;

int from\_tail = list->size - 1 - index;

Node\* current\_node;

if (index < from\_tail) {

current\_node = list->head\_node;

while (counter != index) {

current\_node = current\_node->ptr\_to\_next\_node;

++counter;

}

}

else {

current\_node = list->tail\_node;

while (counter != from\_tail) {

current\_node = current\_node->ptr\_to\_prev\_node;

++counter;

}

}

new\_node->ptr\_to\_next\_node = current\_node;

new\_node->ptr\_to\_prev\_node = current\_node->ptr\_to\_prev\_node;

current\_node->ptr\_to\_prev\_node->ptr\_to\_next\_node = new\_node;

current\_node->ptr\_to\_prev\_node = new\_node;

}

void print\_list(List\* list) {

if (list == nullptr) {

cout << "List is not existing" << endl;

return;

}

Node\* current\_node = list->head\_node;

if (current\_node == nullptr) {

cout << "List is empty" << endl;

return;

}

while (current\_node != nullptr) {

cout << current\_node->key << "; ";

current\_node = current\_node->ptr\_to\_next\_node;

}

cout << endl;

delete current\_node;

}

void list\_filesave(List\* list) {

if (list == nullptr) {

cout << "List is not existing" << endl;

return;

}

Node\* current\_node = list->tail\_node;

ofstream f("list\_savestate.dat");

while (current\_node != nullptr) {

f << current\_node->key << "\n";

current\_node = current\_node->ptr\_to\_prev\_node;

}

f.close();

}

void list\_fileread(List\* list) {

if (list == nullptr) {

cout << "List is not existing" << endl;

return;

}

char buffer\_char[1];

string buffer\_str;

ifstream f("list\_savestate.dat");

while (!f.eof()) {

getline(f, buffer\_str);

buffer\_char[0] = buffer\_str[0];

if (buffer\_str != "") {

add\_node(list, buffer\_str[0]);

}

}

f.close();

}

void list\_delete(List\* list) {

if (list == nullptr) {

cout << "List is not existing" << endl;

return;

}

Node\* remove\_node;

while (list->head\_node != nullptr) {

remove\_node = list->head\_node;

list->head\_node = list->head\_node->ptr\_to\_next\_node;

delete remove\_node;

}

delete list;

}

**Вариант функций для очереди**

struct Node {

char& key = \*(new char);

Node\* ptr\_to\_next\_node = nullptr;

};

struct Queue {

Node\* head\_node = nullptr;

Node\* tail\_node = nullptr;

int size = 0;

};

void pop(Queue\* queue) {

Node\* delete\_node = queue->head\_node;

queue->head\_node = queue->head\_node->ptr\_to\_next\_node;

queue->size--;

delete delete\_node;

}

void push(Queue\* queue, char key) {

Node\* new\_node = new Node;

queue->size++;

new\_node->key = key;

queue->tail\_node->ptr\_to\_next\_node = new\_node;

queue->tail\_node = new\_node;

}

void delete\_key(Queue\* queue, char key) {

if (queue == nullptr) {

cout << "Queue is not existing" << endl;

return;

}

if (queue->head\_node == nullptr) {

cout << "Queue is empty" << endl;

return;

}

int i = 1;

while (i <= queue->size) {

if (queue->head\_node->key == key) {

pop(queue);

}

else

{

push(queue, queue->head\_node->key);

pop(queue);

++i;

}

}

}

void add\_node(Queue\* queue, char str, const int& index = -1) {

if (queue == nullptr) {

cout << "Queue is not existing" << endl;

return;

}

Node\* new\_node = new Node;

new\_node->key = str;

if (queue->head\_node == nullptr) {

queue->head\_node = new\_node;

queue->tail\_node = new\_node;

queue->size++;

return;

}

if ((index == -1) || (index == 0)) {

push(queue, str);

return;

}

int i = 1;

while (i <= queue->size) {

if (i == index) {

push(queue, str);

++i;

}

else

{

push(queue, queue->head\_node->key);

pop(queue);

++i;

}

}

}

void print\_list(Queue\* queue) {

if (queue == nullptr) {

cout << "Queue is not existing" << endl;

return;

}

Node\* current\_node = queue->head\_node;

if (current\_node == nullptr) {

cout << "Queue is empty" << endl;

return;

}

while (current\_node != nullptr) {

cout << current\_node->key << "; ";

current\_node = current\_node->ptr\_to\_next\_node;

}

cout << endl;

}

void list\_filesave(Queue\* queue) {

if (queue == nullptr) {

cout << "Queue is not existing" << endl;

return;

}

Node\* current\_node = queue->head\_node;

ofstream f("list\_savestate.dat");

while (current\_node != nullptr) {

f << current\_node->key << "\n";

current\_node = current\_node->ptr\_to\_next\_node;

}

f.close();

}

void list\_fileread(Queue\* queue) {

if (queue == nullptr) {

cout << "Queue is not existing" << endl;

return;

}

char buffer\_char[1];

string buffer\_str;

ifstream f("list\_savestate.dat");

while (!f.eof()) {

getline(f, buffer\_str);

buffer\_char[0] = buffer\_str[0];

if (buffer\_str != "") {

add\_node(queue, buffer\_str[0]);

}

}

f.close();

}

void list\_delete(Queue\* queue) {

if (queue == nullptr) {

cout << "Queue is not existing" << endl;

return;

}

Node\* remove\_node;

while (queue->head\_node != nullptr) {

remove\_node = queue->head\_node;

queue->head\_node = queue->head\_node->ptr\_to\_next\_node;

delete remove\_node;

}

delete queue;

}

**Вариант функций для стека**

struct Node {

char& key = \*(new char);

Node\* ptr\_to\_next\_node = nullptr;

};

struct Stack {

Node\* head\_node = nullptr;

Node\* tail\_node = nullptr;

int size = 0;

};

void pop(Stack\* stack) {

Node\* delete\_node = stack->head\_node;

stack->head\_node = stack->head\_node->ptr\_to\_next\_node;

stack->size--;

delete delete\_node;

}

void push(Stack\* stack, char key) {

Node\* new\_node = new Node;

stack->size++;

new\_node->key = key;

new\_node->ptr\_to\_next\_node = stack->head\_node;

stack->head\_node = new\_node;

}

void delete\_key(Stack\* stack, char key) {

if (stack == nullptr) {

cout << "Stack is not existing" << endl;

return;

}

if (stack->head\_node == nullptr) {

cout << "Stack is empty" << endl;

return;

}

cout << stack;

Stack\* buffer = new Stack;

while ((stack->size > 0) && (stack->head\_node->key != key)) {

push(buffer, stack->head\_node->key);

pop(stack);

}

if (stack->head\_node != nullptr) pop(stack);

while (buffer->size > 0) {

push(stack, buffer->head\_node->key);

pop(buffer);

}

delete buffer;

}

void add\_node(Stack\* stack, char str, const int& index = -1) {

if (stack == nullptr) {

cout << "Stack is not existing" << endl;

return;

}

Node\* new\_node = new Node;

new\_node->key = str;

if (stack->head\_node == nullptr) {

stack->head\_node = new\_node;

stack->tail\_node = new\_node;

stack->size++;

return;

}

if ((index == -1) || (index == 0)) {

push(stack, str);

return;

}

Stack\* buffer = new Stack;

int i = 1;

while ((stack->size > 0) && (i < index)) {

push(buffer, stack->head\_node->key);

pop(stack);

++i;

}

if (i == index) push(stack, str);

while (buffer->size > 0) {

push(stack, buffer->head\_node->key);

pop(buffer);

}

}

void print\_list(Stack\* stack) {

if (stack == nullptr) {

cout << "Stack is not existing" << endl;

return;

}

Node\* current\_node = stack->head\_node;

if (current\_node == nullptr) {

cout << "Stack is empty" << endl;

return;

}

while (current\_node != nullptr) {

cout << current\_node->key << "; ";

current\_node = current\_node->ptr\_to\_next\_node;

}

cout << endl;

}

void list\_filesave(Stack\* stack) {

if (stack == nullptr) {

cout << "Stack is not existing" << endl;

return;

}

Node\* current\_node = stack->head\_node;

ofstream f("list\_savestate.dat");

while (current\_node != nullptr) {

f << current\_node->key << "\n";

current\_node = current\_node->ptr\_to\_next\_node;

}

f.close();

}

void list\_fileread(Stack\* stack) {

if (stack == nullptr) {

cout << "Stack is not existing" << endl;

return;

}

char buffer\_char[1];

string buffer\_str;

Stack\* buffer = new Stack;

ifstream f("list\_savestate.dat");

while (!f.eof()) {

getline(f, buffer\_str);

buffer\_char[0] = buffer\_str[0];

if (buffer\_str != "") {

add\_node(buffer, buffer\_str[0]);

}

}

Node\* current\_node = buffer->head\_node;

while (current\_node != nullptr) {

add\_node(stack, current\_node->key);

current\_node = current\_node->ptr\_to\_next\_node;

}

delete buffer;

f.close();

}

void list\_delete(Stack\* stack) {

if (stack == nullptr) {

cout << "Stack is not existing" << endl;

return;

}

Node\* remove\_node;

while (stack->head\_node != nullptr) {

remove\_node = stack->head\_node;

stack->head\_node = stack->head\_node->ptr\_to\_next\_node;

delete remove\_node;

}

delete stack;

}

**Main функция (идентична для всех вариантов решения)**

int main() {

int n, k;

string str;

char ch;

Stack\* stack = new Stack;

cout << "Enter number of elements that will be entered on stack creation: ";

cin >> n;

cin.ignore();

for (int i = 1; i <= n; ++i) {

cout << "Enter key for element: ";

getline(cin, str);

ch = str[0];

add\_node(stack, ch);

}

print\_list(stack);

cout << "Enter key for deletion: ";

getline(cin, str);

ch = str[0];

delete\_key(stack, ch);

print\_list(stack);

cout << "Enter index of element for adding and number (K) of entering element: ";

cin >> n >> k;

cin.ignore();

for (int i = 1; i <= k; ++i) {

cout << "Enter key for element: ";

getline(cin, str);

ch = str[0];

add\_node(stack, ch, n);

}

print\_list(stack);

cout << "Stack saved in file" << endl;

list\_filesave(stack);

list\_delete(stack);

stack = nullptr;

print\_list(stack);

stack = new Stack;

cout << "Stack restored from file" << endl;

list\_fileread(stack);

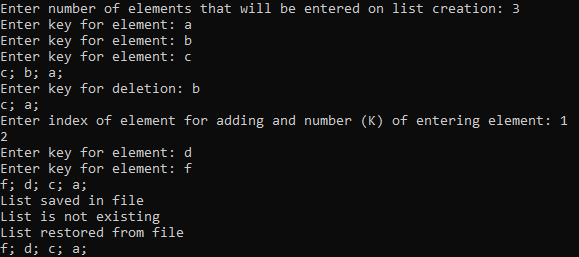
print\_list(stack);

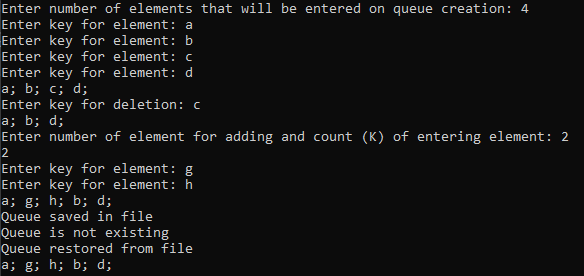
list\_delete(stack);

return 0;

}

**Решение**





**Выводы**

В ходе данной работы были изучены списки, стеки и очереди, и некоторые особенности работы с ними.

**Github**

https://github.com/Hitikov/Lab\_11