МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ»

КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ

Лабораторная работа №3

по дисциплине «Структуры данных и алгоритмы»

Факультет: ПМИ

Группа: ПМИ-03

Студенты: Сидоров Д.И., Малыгин С. А.

Преподаватель: Еланцева Е.Л.

НОВОСИБИРСК

2021

*1)Условие задачи:*

Формулу вида:

< формула > ::= < терминал >⏐(< формула > < знак > < формула >)

< знак > ::= + ⏐ - ⏐ \*

< терминал > ::= 0⏐1⏐2⏐3⏐4⏐5⏐6⏐7⏐8⏐9

можно представить в виде двоичного дерева (‘дерева – формулы ‘):

* формула из одного терминала (цифры) представляется деревом из одной вершины (корнем) с этим терминалом;
* формула вида ( f1 s f2 ) - деревом, в котором корень – это знак s, а

левое и правое поддеревья – это соответствующие представления f1 и f2 :

по заданной формуле построить дерево – формулу, обходя дерево – формулу в 1)прямом, 2)обратном, 3)концевом порядке, напечатать его элементы и вычислить (как целое число) значение;

*2)Анализ данных:*

* + *Входные данные:* Выражение, записанное в файле.
  + *Выходные данные:* результат обходов и результат выражения.
  + *Метод решения:* Запишем данное выражение в стек. Далее при помощи алгоритма построим бинарное ориентированное дерево, где корень - знак, а поддеревья - цифры.

Выведем результаты обхода дерева при помощи трех рекурсивных процедур: pre\_order(прямой обход), post\_order(концевой обход), in\_order(обратный обход). Подсчитаем значение выражения при помощи рекурсивной подпрограммы answer.

Алгоритм построения дерева: Создаем корень. Если встречается “)”, создаем правое поддерево и вызываем функцию answer с указателем на созданную вершину. Если встречается цифра, записываем ее в вершину и говорим, что у этой вершины нет потомков. Если встречается знак, создаем левое поддерево и вызываем функцию answer с указателем на созданную вершину.

* + *Основные подпрограммы:*

output\_from\_File - ввод выражения в стек.

PushFront – добавить элемент в начало списка.

PopFront – взятие первого элемента в динамическом деке.

IsEmpty – проверка динамического дека на пустоту.

add\_btree – добавить ветку дерева.

build\_btree– построение дерева.

Clear- удаление стека.

post\_order- концевой обход.

pre\_order- прямой обход.

in\_order- обратный обход.

IsEmpty- проверка на пустоту стека.

First – ввод первого элемента в стек.

answer – нахождение значения выражения.

3)Структура входных и выходных данных

Внешнее представление входных данных:

Выражение, записанное в файле.

Внутреннее представление входных данных:

Входные данные записываются в стек - линейный двунаправленный ациклический список. Каждое звено списка реализовано структурой

struct List

{

List\* prev;

List\* next;

char Data;

};

List\* begin;

Далее по алгоритму строится бинарное ориентированное дерево. Дерево реализовано следующей структурой:

struct bintree

{

char data;

bintree\* l;

bintree\* r;

};

Внешнее представление выходных данных:

Результат обходов и результат нахождения значения выражения, выведенные на консоль

Внутреннее представление выходных данных:

Выходные данные берутся в результате обхода бинарного ориентированного дерева.

*4)Алгоритм*

*Класс STACK:*

#include " STACK.h"

#include <iostream>

using namespace std;

namespace Program

{

STACK:: STACK (void)

{

}

STACK::~ STACK (void)

{

}

Процедура STACK::First(char data)

{

Если (IsEmpty())

{

begin = new List;

Вывод << "Enter first element: ";

Ввод >> begin->Data;

begin->next = NULL;

begin->prev = NULL;

end = begin;

}

Иначе

{

Вывод << "Dec is not empty" << endl;

}

}

Процедура STACK::PushFront(символ data)

{

Если (IsEmpty())

{

First(data);

}

Иначе

{

List\* temp = new List;

temp->Data = data;

temp->prev = NULL;

temp->next = begin;

begin->prev = temp;

begin = temp;

}

}

Функция STACK::PopFront(char& temp)

{

Если (!IsEmpty())

{

символ temp = begin->Data;

List\* del = begin;

begin = begin->next;

Удалить del;

begin->prev = NULL;

Возвращаем true;

}

Иначе

{

Возвращаем false;

}

}

Функция STACK::IsEmpty()

{

Если (begin == NULL)

{

Возвращаем true;

}

Иначе

{

Возвращаем false;

}

}

Процедура STACK::Clear()

{

Если (!IsEmpty())

{

List\* f = begin->next;

List\* fg = begin;

Пока (f != NULL)

{

Удалить fg;

fg = f;

f = f->next;

}

Удалить f;

begin = NULL;

}

}

Процедура STACK::Show\_ALL()

{

Если (IsEmpty())

{

Вывод << endl << "Dec is empty" << endl;

}

Иначе

{

for (List\* flag = begin; flag != NULL; flag = flag->next)

{

Вывод << flag->Data;

Если (flag->next != NULL)

{

Вывод << "->";

}

}

Вывод << endl;

}

}

}

*Программа*:

using namespace Program;

using namespace std;

struct bintree

{

символ data;

bintree\* l;

bintree\* r;

};

Функция output\_from\_File(struct STACK& List)

{

ifstream fin;

fin.open("Текст.txt");

символ ch = NULL;

Если (!fin.is\_open())

{

Вывод << "Ошибка открытия файла!" << endl;

Возвращаем false;

}

for (; !fin.eof();)

{

fin.get(ch);

Если (ch != '(')

{

List.PushFront(ch);

}

}

List.PopFront(ch);

fin.close();

Возвращаем true;

}

Функция add\_btree(struct bintree\*\* tree, STACK& List)

{

символ ch;

List.PopFront(ch);

Если (ch == ')')

{

(\*tree)->r = new bintree;

bintree\* flag\_r = (\*tree)->r;

add\_btree(&flag\_r, List);

}

Если (('0' < ch) && (ch <= '9'))

{

(\*tree)->data = ch;

(\*tree)->r = NULL;

(\*tree)->l = NULL;

Возвращаем true;

}

List.PopFront(ch);

Если (('\*' <= ch) && (ch <= '-'))

{

(\*tree)->data = ch;

(\*tree)->l = new bintree;

bintree\* flag\_l = (\*tree)->l;

add\_btree(&flag\_l, List);

}

Возвращаем true;

}

Функция build\_btree(struct bintree\*\* zero\_btree)

{

STACK List;

Если (output\_from\_File(List))

{

Если (!List.IsEmpty())

{

(\*zero\_btree) = new bintree;

add\_btree(zero\_btree, List);

List.Clear();

Возвращаем true;

}

}

Возвращаем false;

}

Процедура post\_order(bintree\* d)

{

Если (d != NULL)

{

post\_order(d->l);

post\_order(d->r);

Вывод << d->data;

}

}

Процедура pre\_order(bintree\* d)

{

Если (d != NULL)

{

Вывод << d->data;

pre\_order(d->l);

pre\_order(d->r);

}

}

Процедура in\_order(bintree\* d)

{

Если (d != NULL)

{

pre\_order(d->l);

Вывод << d->data;

pre\_order(d->r);

}

}

Функция answer(bintree\* d)

{

символ symbol = d->data;

Если (('\*' <= symbol) && (symbol <= '-'))

{

Оператор ветвления (symbol)

{

case '\*' :

Возвращаем (answer(d->l) \* answer(d->r));

break;

case '+' :

Возвращаем (answer(d->l) + answer(d->r));

break;

case '-' :

Возвращаем (answer(d->l) - answer(d->r));

break;

}

}

Иначе

{

Возвращаем (Целое число(symbol) - '0');

}

}

Функция main()

{

setlocale(LC\_ALL, "rus");

bintree\* btree = NULL;

Если (build\_btree(&btree))

{

Вывод << endl << "Концевой обход: ";

post\_order(btree);

Вывод << endl << "Прямой обход: ";

pre\_order(btree);

Вывод << endl << "Обратный обход: ";

in\_order(btree);

Вывод << endl << "Ответ: " << answer(btree);

}

Иначе

{

Вывод << "Выражение не дано!";

}

Возвращаем 0;

}

*5)Структура программы:*

1) Процедура ввод первого элемента:

First(char data)

Входные данные: data – первый элемент.

3) Процедура добавления элемента в начало списка:

PushFront(char data)

Входные данные: data – элемент, добавляемый в начало списка.

5) Функция взятия первого элемента в динамическом деке:

PopFront(char& temp)

Входные данные: temp – элемент, значение которого изменится на значение элемента из начала списка.

Выходные данные: true – если Дек не пуст, false – если Дек пуст.

6) Функция проверки динамического дека на пустоту:

IsEmpty()

Выходные данные: False или True

7) Процедура очистки динамического дека:

Clear()

9)Функция ввода выражения в стек:

output\_from\_File(struct STACK& List)

Входные данные:

List – указатель на начало списка.

Выходные данные: True

10)Функция создания ветки дерева:

add\_btree(struct bintree\*\* tree, STACK& List)

Входные данные:

Tree- указатель на корень дерева.

List- указатель на начало списка.

Выходные данные: True

11)Функция построения дерева:

build\_btree(struct bintree\*\* zero\_btree)

Выходные данные:

zero\_btree – указатель на корень дерева.

12)Процедура концевого обхода

post\_order(bintree\* d)

Входные данные: d- указатель на корень дерева.

12)Процедура прямого обхода

pre\_order(bintree\* d)

Входные данные: d- указатель на корень дерева.

12)Процедура обратного обхода

in\_order(bintree\* d)

Входные данные: d- указатель на корень дерева.

13)Функция нахождения значения выражения

answer(bintree\* d)

Входные данные: d- указатель на корень дерева

Выходные данные: целое число – значение выражения

First

IsEmpty

PushFront

output\_from\_File

PopFront

Clear

add\_btree

build\_btree

Pre\_order

answer

main

In\_order

Post\_order

build\_btree

build\_btree

build\_btree

add\_btree

add\_btree

*6)Текст программы:*

*Класс STACK:*

#include "STACK.h"

#include<iostream>

using namespace std;

namespace Program

{

STACK::STACK(void)

{

}

STACK::~STACK(void)

{

}

void STACK::First(char data)

{

if (IsEmpty())

{

begin = new List;

begin->Data = data;

begin->next = NULL;

begin->prev = NULL;

}

}

void STACK::PushFront(char data)

{

if (IsEmpty())

{

First(data);

}

else

{

List\* temp = new List;

temp->Data = data;

temp->prev = NULL;

temp->next = begin;

begin->prev = temp;

begin = temp;

}

}

bool STACK::PopFront(char& temp)

{

if (!IsEmpty())

{

if (begin->next == NULL)

{

temp = begin->Data;

begin = NULL;

}

else

{

temp = begin->Data;

List\* del = begin;

begin = begin->next;

delete del;

begin->prev = NULL;

}

return true;

}

else

{

return false;

}

}

bool STACK::IsEmpty()

{

if (begin == NULL)

{

return true;

}

else

{

return false;

}

}

void STACK::Clear()

{

if (!IsEmpty())

{

List\* f = begin->next;

List\* fg = begin;

while (f != NULL)

{

delete fg;

fg = f;

f = f->next;

}

delete f;

begin = NULL;

}

}

void STACK::Show\_ALL()

{

if (IsEmpty())

{

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

}

else

{

for (List\* flag = begin; flag != NULL; flag = flag->next)

{

cout << flag->Data;

}

cout << endl;

}

}

}*Программа*:

#include<iostream>

#include<fstream>

#include"STACK.h"

using namespace Program;

using namespace std;

struct bintree

{

char data;

bintree\* l;

bintree\* r;

};

bool output\_from\_File(struct STACK& List)

{

ifstream fin;

fin.open("Текст.txt");

char ch = NULL;

if (!fin.is\_open())

{

cout << "Ошибка открытия файла!" << endl;

return false;

}

for (; !fin.eof();)

{

fin.get(ch);

if (ch != '(')

{

List.PushFront(ch);

}

}

List.PopFront(ch);

fin.close();

return true;

}

bool add\_btree(struct bintree\*\* tree, STACK& List)

{

char ch;

List.PopFront(ch);

if (ch == ')')

{

(\*tree)->r = new bintree;

bintree\* flag\_r = (\*tree)->r;

add\_btree(&flag\_r, List);

}

if (('0' < ch) && (ch <= '9'))

{

(\*tree)->data = ch;

(\*tree)->r = NULL;

(\*tree)->l = NULL;

return true;

}

List.PopFront(ch);

if (('\*' <= ch) && (ch <= '-'))

{

(\*tree)->data = ch;

(\*tree)->l = new bintree;

bintree\* flag\_l = (\*tree)->l;

add\_btree(&flag\_l, List);

}

return true;

}

bool build\_btree(struct bintree\*\* zero\_btree)

{

STACK List;

if (output\_from\_File(List))

{

if (!List.IsEmpty())

{

(\*zero\_btree) = new bintree;

add\_btree(zero\_btree, List);

List.Clear();

return true;

}

}

return false;

}

void post\_order(bintree\* d)

{

if (d != NULL)

{

post\_order(d->l);

post\_order(d->r);

cout << d->data;

}

}

void pre\_order(bintree\* d)

{

if (d != NULL)

{

cout << d->data;

pre\_order(d->l);

pre\_order(d->r);

}

}

void in\_order(bintree\* d)

{

if (d != NULL)

{

pre\_order(d->l);

cout << d->data;

pre\_order(d->r);

}

}

int answer(bintree\* d)

{

char symbol = d->data;

if (('\*' <= symbol) && (symbol <= '-'))

{

switch (symbol)

{

case '\*' :

return (answer(d->l) \* answer(d->r));

break;

case '+' :

return (answer(d->l) + answer(d->r));

break;

case '-' :

return (answer(d->l) - answer(d->r));

break;

}

}

else

{

return (int(symbol) - '0');

}

}

int main()

{

setlocale(LC\_ALL, "rus");

bintree\* btree = NULL;

if (build\_btree(&btree))

{

cout << endl << "Концевой обход: ";

post\_order(btree);

cout << endl << "Прямой обход: ";

pre\_order(btree);

cout << endl << "Обратный обход: ";

in\_order(btree);

cout << endl << "Ответ: " << answer(btree);

}

else

{

cout << "Выражение не дано!";

}

return 0;

}

*7)Тесты:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Входные данные | Выходные данные | | Примечание |
| Результат обходов | Результат вычисления выражения |
| 1 | (4+8) | Концевой обход 48+  Прямой обход +48  Обратный обход 4+8 | 12 | Простое выражение |
| 2 | (((4+8)-(5\*4))+(7\*1)) | Концевой обход  48+54\*-71\*+  Прямой обход  +-+48\*54\*71  Обратный обход  4+8-\*54+\*71 | -1 | Более сложное выражение |
| 3 | 5 | Концевой обход 5  Прямой обход 5  Обратный обход 5 | 5 | выражение, которое состоит из одной цифры |
| 4 | \*Пусто\* | Выражение не дано! | | Выражения нет в файле |

8) *Результат работы программы*:

Программа работает правильно, что подтверждают тесты.