

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: Рекурсивная обработка иерархических списков

Студент гр. 9382

Докукин В.М.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Познакомиться с такой часто используемой на практике нелинейной конструкцией, как иерархический список, способами её организации и рекурсивной обработки, а также научиться применять иерархические списки для решения различных задач, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Основные теоретические положения.

Иерархический список — нелинейная структура данных, представляющая из себя набор структурных единиц — элементов списка, каждый из которых содержит указатель на следующий по порядку элемент списка. Основным отличием иерархического списка от обычного связного списка является наличие у каждого элемента указателя на подсписок. Таким образом, элементами иерархического списка могут быть не только элементы, но и другие списки.

Написание кода производилось на базе системы Linux Ubuntu 18.04. Код был написан на языке C++ с использованием стандартных библиотек. Работа с кодом происходила в текстовом редакторе Notepad++.

Задание.

Пусть выражение (логическое, арифметическое, алгебраическое*) представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме ((<операция> <аргументы>)), либо в постфиксной форме (<аргументы> <операция>). В задании даётся один из следующих вариантов требуемого действия с выражением: проверка синтаксической корректности, упрощение (преобразование), вычисление.

17) логическое, упрощение, префиксная форма

Описание структур данных для реализации иерархических списков.

1. class Node – класс элемента списка. Класс содержит 3 поля — char key для хранения ключа, Node* next для хранения указателя на следующий элемент и Node* child для хранения указателя на начало подсписка.

Кроме того, реализованы следующие функции работы с иерархическим списком:

1. bool areIdentical(Node* node1, Node* node2) – рекурсивная функция. Проверяет, являются ли два списка идентичными. Два списка считаются идентичными, если они содержат одинаковые элементы в одинаковой последовательности.

2. void simplifyNode(Node* node) – упрощает список согласно основным логическим законам(идемпотентности, исключённого третьего, непротиворечия, исключения констант, двойного отрицания, де Моргана).

3. simplifyExpression(Node* head) – рекурсивная функция. Упрощает выражение, переданное в виде списка.

4. bool isCorrectExpression(std::string expr) – проверяет, является ли введённое выражение корректным. Выражение считается корректным, если в нём присутствуют только символы «1», «0», «(», «)», «&», «|», «!».

5. int exprToList(std::string expr, Node* head, int i = 0) – рекурсивная функция. Преобразует выражение в список, первым элементом которого является объект по адресу в переменной head.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 — Результаты тестирования

№ теста	Входные данные	Выходные данные	Комментарии
1	(& (a b) (a b))	isCorrectExpression() call You've inserted the next expression: (& (a b) (a b)) simplifyExpression() call for node & simplifyExpression() call for node simplifyExpression() call for node a simplifyNode() call for a simplifyExpression() call for node b simplifyNode() call for b simplifyNode() call for areIdentical() call for a b simplifyExpression() call for node simplifyExpression() call for node a simplifyNode() call for a	Проверка на считывание выражения с лишними пробелами

		simplifyExpression() call for node b simplifyNode() call for b simplifyNode() call for areIdentical() call for a b simplifyNode() call for & areIdentical() call for areIdentical() call for a a areIdentical() call for b b Simplified expression: (a b)	
2	[пустая строка]	isCorrectExpression() call Incorrect expression format.	Проверка на взаимодействие с пустой строкой
3	()	isCorrectExpression() call Incorrect expression format.	Проверка на взаимодействие с пустыми скобками
4	(& (a b) (& c d))))	isCorrectExpression() call You've inserted the next expression: (& (a b) (& c d)) simplifyExpression() call for node & simplifyExpression() call for node simplifyExpression() call for node a simplifyNode() call for a simplifyExpression() call for node b simplifyNode() call for b simplifyNode() call for areIdentical() call for a b simplifyExpression() call for node & simplifyExpression() call for node c simplifyNode() call for c simplifyExpression() call for node d simplifyNode() call for d simplifyNode() call for & areIdentical() call for c d simplifyNode() call for & areIdentical() call for & Simplified expression: (& (a b) (& c d))	Проверка на взаимодействие с лишними скобками

Выводы.

В ходе выполнения лабораторной работы:

1. Была изучена структура данных — иерархический список, а также особенности и принципы работы с ней.
2. Была написана программа, решающая поставленную задачу.

3. Была написана серия тестов, позволяющих качественно оценить работу программы(тесты находятся в файле tests.txt в директории с исходным кодом программы).

ПРИЛОЖЕНИЕ 1.

ИСХОДНЫЙ КОД ПРОГРАММЫ.

Название файла: main.cpp

```
#include<iostream>
#include<string>
#include<fstream>

class Node{ // Ўёâíâíò ñîèñêà
public:
    char key;
    Node* next;
    Node* child;
    Node(char key = '\0', Node* next = nullptr, Node* child = nullptr){
        this->key = key;
        this->next = next;
        this->child = child;
    }
};

bool areIdentical(Node* node1, Node* node2){ // Ĭôîââôÿâò, ÿâëÿpòñÿ èè äââ
ñîèñêà èââíðè-íûìè
    std::cout<<"areIdentical() call for "<<node1->key<<" "<<node2->key<<'\n';
    if (node1->key != node2->key) return false;
    if (node1->child != nullptr && node2->child != nullptr){
        if (node1->child->next != nullptr && node2->child->next != nullptr)
return areIdentical(node1->child, node2->child) & areIdentical(node1->child-
>next, node2->child->next);
        else return areIdentical(node1->child, node2->child);
    }
    return true;
}

void simplifyNode(Node* node){ // Óîôîùâíèâ ñîèñêà
    std::cout<<"simplifyNode() call for "<<node->key<<'\n';
    if (node->key == '&'){
        if (node->child->key == '0' || node->child->next->key == '0'){
            node->key = '0';
            node->child = nullptr;
            return;
        }
    }
}
```

```

    }
    if (node->child->key == '1'){
        node->key = node->child->next->key;
        node->child = node->child->next->child;
        return;
    }
    if (node->child->next->key == '1'){
        node->key = node->child->key;
        node->child = node->child->child;
        return;
    }
    if (areIdentical(node->child, node->child->next)){
        node->key = node->child->key;
        node->child = node->child->child;
        return;
    }
    if (node->child->key == '!' && node->child->next->key == '!'){
        node->key = '|';
        node->child->key = node->child->child->key;
        node->child->next->key = node->child->next->child->key;
        node->child->child = node->child->child->child;
        node->child->next->child = node->child->next->child->child;
        return;
    }
    if (node->child->key == '!'){
        if (node->child->next != nullptr && node->child->child !=
nullptr)

            if (areIdentical(node->child->next, node->child->child)){
                node->key = '0';
                node->child = nullptr;
            }
        return;
    }
    if (node->child->next->key == '!'){
        if (node->child->next->child != nullptr && node->child !=
nullptr)

            if (areIdentical(node->child->next->child, node->child)){
                node->key = '0';
                node->child = nullptr;
            }
        return;
    }
}

```

```

    }
    if (node->key == '|'){
        if (node->child->key == '1' || node->child->next->key == '1'){
            node->key = '1';
            node->child = nullptr;
            return;
        }
        if (node->child->key == '0'){
            node->key = node->child->next->key;
            node->child = node->child->next->child;
            return;
        }
        if (node->child->next->key == '0'){
            node->key = node->child->key;
            node->child = node->child->child;
            return;
        }
        if (areIdentical(node->child, node->child->next)){
            node->key = node->child->key;
            node->child = node->child->child;
            return;
        }
        if (node->child->key == '!' && node->child->next->key == '!'){
            node->key = '&';
            node->child->key = node->child->child->key;
            node->child->next->key = node->child->next->child->key;
            node->child->child = node->child->child->child;
            node->child->next->child = node->child->next->child->child;
            return;
        }
        if (node->child->key == '!'){
            if (node->child->next != nullptr && node->child->child !=
nullptr)

                if (areIdentical(node->child->next, node->child->child)){
                    node->key = '1';
                    node->child = nullptr;
                }
            return;
        }
        if (node->child->next->key == '!'){
            if (node->child->next->child != nullptr && node->child !=
nullptr)

```



```

        if (areIdentical(node->child->next->child, node->child)){
            node->key = '1';
            node->child = nullptr;
        }
        return;
    }
}

if (node->key == '!'){
    if (node->child->key == '0'){
        node->key = '1';
        node->child = nullptr;
        return;
    }
    if (node->child->key == '1'){
        node->key = '0';
        node->child = nullptr;
        return;
    }
    if (node->child->key == '!'){
        node->key = node->child->child->key;
        node->child = node->child->child->child;
        return;
    }
}

}

void simplifyExpression(Node* head){ // Óíðíùáíèå åñåãî âúðàæåíèÿ(íáõíä ñíèñèå à
åëóáèó)
    std::cout<<"simplifyExpression() call for node "<<head->key<<'\n';
    if (head->child != nullptr) {
        simplifyExpression(head->child);
        if (head->child->next != nullptr) simplifyExpression(head->child-
>next);
    }
    simplifyNode(head);
}

bool isCorrectExpression(std::string expr){ // Ĭđîââđÿâò, ÿâëÿâòñÿ èè åååå,ííîå
âúðàæåíèå êîðäâèùî
    std::cout<<"isCorrectExpression() call\n";
    int open = 0;
    for (int i = 0; i < expr.length(); i++){

```

```

        if (expr[i] >= '2' && expr[i] <= '9') return false;
        if (expr[i] == '(') open++;
        if (expr[i] == ')') open--;
    }
    if (open > 0) return false;
    return true;
}

int exprToList(std::string expr, Node* head, int i = 0){ // Ĩðâĭáðâçóâò
âûðâæâĭââ â èâðâððè-âñêèè ñĭèñĭê
    while (expr[i] == ' '){
        i++;
    }
    if (expr[i] == '('){
        head->key = expr[++i];
        head->child = new Node();
        i = exprToList(expr, head->child, i + 1);
        if (head->key == '|' || head->key == '&'){
            head->child->next = new Node();
            i = exprToList(expr, head->child->next, i);
        }
    }
    else if (isalpha(expr[i])){
        head->key = expr[i];
        head->child = nullptr;
    }
    else if (isdigit(expr[i])){
        head->key = expr[i];
        head->child = nullptr;
    }
    return i + 1;
}

void printList(Node* head){ // Ĩâ-àòââò ñĭèñĭê ĭà ýêðâĭ
    if (head->key == '&' || head->key == '|' || head->key == '!'){
        std::cout<<"("<<head->key<<" ";
        printList(head->child);
        if (head->child->next != nullptr){
            std::cout<<" ";
            printList(head->child->next);
        }
        std::cout<<")";
    }

```

```

    }
    else{
        std::cout<<head->key;
    }
}

void process(std::string expr){ // íáðàáíðèà áááá,íííâí âûðàæáíèÿ
    int c = isCorrectExpression(expr);
    if (!c) {
        std::cout<<"Incorrect expression format.\n";
        return;
    }

    Node* head = new Node();

    exprToList(expr, head);

    std::cout<<"You've inserted the next expression: ";
    printList(head);
    std::cout<<'\\n';

    simplifyExpression(head);
    std::cout<<"Simplified expression:\\n";
    printList(head);
    std::cout<<'\\n';

    delete head;
}

int func(){
    int a;
    std::string expr;

    std::cout<<"Choose input option(0 - file input, 1 - console input):\\n";
    std::cin>>a;
    if (a){
        std::cin.ignore();
        std::getline(std::cin, expr);
        process(expr);
        return 0;
    }
}

```

```

std::ifstream f("tests.txt");
if (!f){
    std::cout<<"Couldn't open file!\n";
    return 1;
}

while (!f.eof()){
    std::getline(f, expr);
    process(expr);
    std::cout<<"-----\n";
}
return 0;
}

int main(){
    try{
        func();
    }
    catch(...){
        std::cout<<"An unexpected error occurred.\n";
    }
    return 0;
}

```