**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

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

Тема: Алгоритмы сортировки

|  |  |  |
| --- | --- | --- |
| Студент гр. 8381 |  | Киреев К.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

## Цель работы.

Ознакомиться с основными понятиями алгоритмов сортировки на языке программирования C++. Разработать программу, реализующую сортировку Шелла с разными способами задания длин промежутков.

## Задание.

Вариант №15

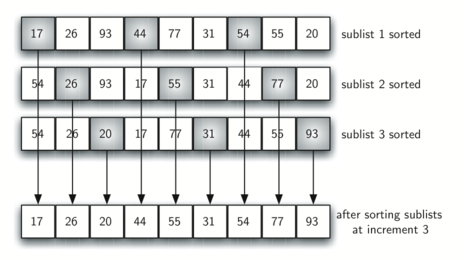
Реализовать сортировку Шелла с графическим интерфейсом.

## Основные теоретические положения.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d{\displaystyle d}. После этого процедура повторяется для некоторых меньших значений d{\displaystyle d}, а завершается сортировка Шелла упорядочиванием элементов при {\displaystyle d=1} d = 1(то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

* отсутствие потребности в памяти под стек;
* отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до O(n²), что хуже, чем худшее гарантированное время для сортировки Шелла.

Рис.1.Графическая схема иерархического списка.

## Выполнение работы.

Для решения задачи используются следующие функции:

void calculate(const H\_list x, L\_list& first\_el, int& n); – Проверка иерархического списка на пустоту

void calc\_expression(const H\_list x, int& n, L\_list& first\_el, L\_list& el); – Работа со списком с обрамляющими скобками

void calc\_brackets(const H\_list x, int& n, L\_list& first\_el, L\_list& el); – Работа со списком без обрамляющих скобок

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тип | Имя | Назначение |
| 1 | const H\_list | x | Хранение иерархического списка |
| 2 | int & | n | Количество атомов в иерархическом списке |
| 3 | L\_list& | first\_el | Указатель на первый элемент создающегося линейного списка |
| 4 | L\_list& | el | Указатель на текущий элемент создающегося линейного списка |

Алгоритм работы:

*Шаг 1.*

Вызов функции calculate(h, l, n). Если переданный ей иерархический список пуст, то указатель на первый элемент линейного списка получает значение NULL и анализ завершается; иначе переход к шагу 2.

*Шаг 2.*

Вызов функции calc\_expression(x, n, first\_el, el). Работу функции описывает Шаг 3. Присваивание значение NULL полю next текущего элемента линейного списка. Завершение анализа.

*Шаг 3.*

Если текущий элемент иерархического списка является атомом, то увеличение переменной n, отвечающей за количество атомов в списке, на 1, затем запись текущего элемента иерархического списка в линейный список; иначе вызов функции calc\_brackets(x, n, first\_el, el). Работу функции описывает Шаг 4.

*Шаг 4.*

Если иерархический список не пуст, то вызов функции calc\_expression(head(x), n, first\_el, el) (см. Шаг 3), а затем функции calc\_brackets(tail(x), n, first\_el, el) (см. Шаг 4).

Примечание: после завершения анализа иерархический и линейный списки удаляются.

## Оценка эффективности алгоритма.

Алгоритм, реализованный в программе, имеет линейную зависимость от количества элементов иерархического списка, то есть сложность оценивается как O(n). Ввиду рекурсивного алгоритма рост занимаемой памяти растет линейно из-за создаваемых в функциях временных переменных.

## Тестирование программы.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Набор тестовых данных | | Предполагаемые результаты, высчитанные вручную | Результаты выполнения программы | Линейный список | Сравнительный анализ |
| № | Данные |
| 1 | ((((r)))) | 1 | 1 | r | Ошибок нет |
| 2 | (a s y) | 3 | 3 | asy | Ошибок нет |
| 3 | (d y (t I) o p) | 6 | 6 | dytlop | Ошибок нет |
| 4 | (((y I u p) k y n w r) k o) | 11 | 11 | ylupkynwrko | Ошибок нет |
| 5 | () | 0 | 0 | - | Ошибок нет |
| 6 | E | 1 | 1 | E | Ошибок нет |
| 7 | (e) | 1 | 1 | e | Ошибок нет |
| 8 | ((a p r o () (r o (g h) v k) e n) h g o r) | 16 | 16 | aprorohvkenhgor | Ошибок нет |

## Выводы.

В ходе выполнения лабораторной работы была изучена такая структура данных как иерархические списки, а также рекурсивные методы ее обработки. Была реализована программа на С++, использующая иерархические списки, которая анализирует строку, подсчитывает количество атомов и составляет линейный список атомов.

# Приложение А Исходный код программы

#include <iostream>

#include <cstdlib>

#include <fstream>

using namespace std;

typedef char type;

struct list{

    type atom;

    list\* next;

};

struct s\_expr;

struct hd\_and\_tl{

    s\_expr\* hd;

    s\_expr\* tl;

};

struct s\_expr{

    bool tag;

    union{

        type atom;

        hd\_and\_tl pair;

    }node;

};

typedef s\_expr\* H\_list;

typedef list\* L\_list;

bool isNull(const H\_list x);

H\_list head(const H\_list x);

H\_list tail(const H\_list x);

H\_list cons(const H\_list x, const H\_list y);

bool isAtom(const H\_list x);

H\_list make\_atom(const type x);

void destroy\_h(H\_list s);

void destroy\_l(L\_list b);

void read\_expression(type prev\_symbol, H\_list& y, ifstream &infile);

void read\_brackets(H\_list& y, ifstream &infile);

void write\_expression(const H\_list x, ofstream &outfile);

void write\_brackets(const H\_list x, ofstream &outfile);

void output(H\_list& s, L\_list& b, ofstream &outfile, int &n);

void calculate(const H\_list x, L\_list& b, int &n);

void calc\_expression(const H\_list x, int & n, L\_list& first\_el, L\_list& el);

void calc\_brackets(const H\_list x, int & n, L\_list& first\_el, L\_list& el);

int main(){

    ifstream infile("input.txt");

    ofstream outfile("output.txt");

    H\_list h;

    L\_list l;

    int n = 0;

    type x;

    do

        infile >> x;

    while (x == ' ');

    read\_expression(x, h, infile);

    calculate(h, l, n);

    output(h, l, outfile, n);

    destroy\_h(h);

    destroy\_l(l);

    return 0;

}

bool isNull(const H\_list x){

    return x == NULL;

}

H\_list head(const H\_list x){

    if(x == NULL){

        cerr << "! Head(Nil)\n";

        exit(1);

    }

    else

        if(isAtom(x)){

            cerr << "! Head(Atom)\n";

            exit(1);

        }

        else

            return x->node.pair.hd;

}

H\_list tail(const H\_list x){

    if(x == NULL){

        cerr << "! Tail(Nil)\n";

        exit(1);

    }

    else

        if(isAtom(x)){

            cerr << "! Tail(Atom)\n";

            exit(1);

        }

        else

            return x->node.pair.tl;

}

H\_list cons(const H\_list x, const H\_list y){

    if(isAtom(y)){

        cerr << "! Cons(x,Atom)\n";

        exit(1);

    }

    else{

        H\_list p = new s\_expr;

        if(p != NULL){

            p->tag = false;

            p->node.pair.hd = x;

            p->node.pair.tl = y;

            return p;

        }

        else{

            cerr << "! Exhausted memory\n";

            exit(1);

        }

    }

}

bool isAtom(const H\_list x){

    if(x == NULL)

        return false;

    else

        return (x->tag);

}

H\_list make\_atom(const type x){

    H\_list p = new s\_expr;

    if(p != NULL){

        p -> tag = true;

        p->node.atom = x;

    }

    else{

        cerr << "! Exhausted memory\n";

        exit(1);

    }

    return p;

}

void destroy\_h(H\_list s){

    if(s != NULL){

        if(!isAtom(s)){

            destroy\_h(head(s));

            destroy\_h(tail(s));

        }

    delete s;

    }

}

void destroy\_l(L\_list b){

    L\_list tmp;

    while(b != NULL){

        tmp = b->next;

        delete b;

        b = tmp;

    }

    delete b;

}

void read\_expression(type prev\_symbol, H\_list& y, ifstream &infile){

    if(prev\_symbol == ')'){

        cerr << " ! First symbol can't be )\n";

        exit(1);

    }

    else

        if(prev\_symbol != '(')

            y = make\_atom(prev\_symbol);

        else

            read\_brackets(y, infile);

}

void read\_brackets(H\_list& y, ifstream &infile){

    type x;

    H\_list H1, H2;

    if (!(infile >> x)){

        cerr << " ! Missing characters after (" << endl;

        exit(1);

    }

    else{

        if(x == ')')

            y = NULL;

        else{

            read\_expression(x, H1, infile);

            read\_brackets(H2, infile);

            y = cons(H1, H2);

        }

    }

}

void write\_expression(const H\_list x, ofstream &outfile){

    if (isNull(x)){

        cout << " ()";

        outfile << " ()";

    }

    else

        if (isAtom(x)){

            cout << ' ' << x->node.atom;

            outfile << ' ' << x->node.atom;

        }

        else{

            cout << " (" ;

            outfile << " (" ;

            write\_brackets(x, outfile);

            cout << " )";

            outfile << " )" ;

        }

}

void write\_brackets(const H\_list x, ofstream &outfile){

    if (!isNull(x)){

        write\_expression(head(x), outfile);

        write\_brackets(tail(x), outfile);

    }

}

void output(H\_list &s, L\_list &b, ofstream &outfile, int& n){

    cout << "Hierarchical list:";

    outfile << "Hierarchical list:";

    write\_expression(s, outfile);

    cout << endl << "Atoms: " << n ;

    outfile << endl << "Atoms: " << n ;

    cout << endl << "Linked list: ";

    outfile << endl << "Linked list:";

    if(b != NULL)

        while(b != NULL){

            outfile << b->atom << " ";

            cout << b->atom << " ";

            b = b->next;

        }

    else{

        cout << "()" << endl;

        outfile << "()" << endl;

    }

}

void calculate(const H\_list x, L\_list& first\_el, int& n){

    L\_list el;

    if (isNull(x))

        first\_el = NULL;

    else{

        calc\_expression(x, n, first\_el, el);

        el->next = NULL;

    }

}

void calc\_expression(const H\_list x, int& n, L\_list& first\_el, L\_list& el){

    if(isAtom(x)){

        n++;

        if(n == 1){

            el = new list;

            el->atom = x->node.atom;

            el->next = NULL;

            first\_el = el;

        }

        else{

            el->next = new list;

            el = el->next;

            el->atom = x->node.atom;

        }

    }

    else

        calc\_brackets(x, n, first\_el, el);

}

void calc\_brackets(const H\_list x, int& n, L\_list& first\_el, L\_list& el){

    if (!isNull(x)){

        calc\_expression(head(x), n, first\_el, el);

        calc\_brackets(tail(x), n, first\_el, el);

    }

}