СЛИВЫ ИНФЫ

№	Задания
1	1.1 Ссылочной тип В языках со строгой дисциплиной описаний (Паскаль) динамический объект не может иметь собственного имени, так как все идентификаторы языка должны быть описаны, поэтому принято не именовать, а обозначать динамический объект посредством ссылки на него. Переменная-ссылка должна быть описана в разделе объявлений программы как переменная ссылочного типа. При этом сама ссылка является статическим объектом. Ссылка занимает всего лишь одно машинное слово, что совсем немного (0(1)) по сравнению с возможным размером растущего динамического объекта. Ссылочный тип - это такой же простой скалярный тип, как целый, вещественный и логический, также имеющий прямую аппаратную поддержку. Элементами множества значений этого типа являются конкретные ссылки на объекты указанного типа, созданные в основной памяти в процессе выполнения программы. Разные указанные типы порождают разные ссылочные типы, множества значений которых не пересекаются. Однако пустое ссылочное значение nil принадлежит любому из ссылочных типов. Для переменных в точности одного и того же ссылочного типа определены операции присваивания и разыменования и отношение равенства. Операция разыменования обеспечивает доступ к значению обозначаемого ссылкой объекта. В Си обозначается звездочкой слева от указателя (*p). Операция разыменования имеет аппаратную поддержку в любом современном процессоре (косвенная адресация). «Важно делать различие между переменной-указателем и указуемым объектом и быть очень осторожным при присваивании и сравнении указателей» 1.2 Алгоритм кмп 1977м году Д.Кнут, В.Морис и В.Пратт опубликовали алгоритм, требующий только H сравнений даже в самом плохом случае, вместо **H*M** (Морис разработал алгоритм отдельно от Кнута и Пратта, но печатались они вместе) **Идея:** начиная каждый раз сравнение объекта со строкой мы забываем ценную информацию (как в прямолинейном методе поиска) о тех элементах, которые уже просмотрели и сравнили. Технически это просмотр объекта на самоподобие (фрактальность) и составление таблицы D, в которой для каждого случая неполного прохода по объекту (нашли первое неполное вхождение, в котором на ***j*-том** месте расхождения) было число, на которое нужно было вернуться назад (с ***j*-того** места) для нового сравнения (возможно и “-1”, т.е. идти вперёд). Вычисление D само является поиском в строке, для чего можно использовать КМП алгоритм. **Алгоритм:** **а)** вычисление таблицы D **б)** ищем вхождение первого элемента **в)** поэлементно сравниваем, если не совпадает на *j*-той позиции → подставляем ***j*** в таблицу D и продолжаем проверять, сдвинув начало первого вхождения на ***j* – *d*[*j*]** **г)** если в строке ещё может быть слово **(*j*!= H – M)** возвращаемся к п.б, ELSE – вхождений нет. **Простыми словами:** как только мы получили несоответствие при сравнении образца со строкой, мы забываем об этом, и начинаем сравнивать уже со следующего элемента строки, и так до победного. **Например**: Нужно найти слово Hooligan в предложении HuliHooluganHooliganHolk HuliHooluganHooligan Hooligan - на 2-м элементе совпадения прекратились - начинаем сравнивать с 3-го элемента

бю

……

- на 12-м элементе совпадения прекратились - начинаем сравнивать с 13-го. На 20м элементе часть сроки полностью совпала с образом - мы нашли 1-е вхождение. Проверяем дальше до конца строки.

Достоинством КМП является однопроходность, ведь по строке мы не возвращаемся назад, что выгодно для электромеханических устройств запоминания (не нужен реверс) и для удалённого поиска по сети.

Минус – подлинный выигрыш только, если перед неудачей было некоторое число совпадений (скорее исключение, чем правило).

1.3 Степень дерева

Замечание:

Находим вершину с максимальным кол-вом детей

Решение:

Пусть у нас заданы следующие структуры:

```
typedef struct _tree_node {
    char name;
    struct vector * children;
} tree_node;

typedef struct _vector_item {
    struct _vector_item * next;
    struct _vector_item * prev;
    struct _tree_node * node;
} dequeue_item;

typedef struct _vector {
    struct _vector_item * head;
    struct _vector_item * back;
    int size;
} vector;
```

то функция определяющую степень выглядит вот так:

```
int degree_tree(tree_node * root) {
    int max;
    vector_item * curr;

    vector queue;
    vector_create(&queue);

    max = root->children->size;

    curr = root->children->head;
    while (curr != NULL) {
        vector_push_back(&queue, curr);
        curr = curr->next;
    }

    while (queue.size > 0) {
        curr = vector_pop_front(&queue);

        if (curr->node->children->size > max) {
            max = curr->node->children->size;
        }

        curr = curr->node->children->head;
        while (curr != NULL) {
            vector_push_back(&queue, curr);
            curr = curr->next;
        }
    }
    dequeue_destroy(&queue);
    return max;
}
```

2

2.1 Деревья выражений (стр 288)

В синтаксическом анализе определяют формальные грамматики, которые представляют собой множество правил подстановки. Так, выражение можно определить как множество термов, разделяемых знаками «+» или « - »; в свою очередь, терм есть совокупность множителей, разделяемых знаками «*» или «/»; наконец, множитель может быть либо идентификатором, либо константой. Иерархическая структура выражения может быть описана древовидной схемой.

Обходя дерево выражения разными способами, мы получим три различные очереди вершин, которые представляют различные формы записи выражений: префиксную, инфиксную, но без скобок, задающих порядок выполнения операций, и постфиксную (обратную польскую).

2.2 Таблицы с прямым доступом (387 стр)

В 2х словах: ускорение работы с таблицей за счёт хеширования (перевода ключей элементов из литерной (символьной) формы в число, которое является указателем в памяти на этот элемент).
{ведь обращение к элементу (чтение, запись, удаление) в конечном итоге есть обращение к памяти, так почему бы не начать сразу обращаться к памяти, пропуская сравнения ключей и всякое остальное}
{для такого финта ушами нужно поместить таблицу с массивом (доступ за постоянное время)}
Для этого перевода нужна хеш-функция (отображение H *ключей* элементов в *адреса* памяти).

Простейший пример хеш-функции: функция получения числового кода литера
(если литеров несколько, то можно воспользоваться схемой Горнера: A = б + (i – 1) * sizeof(T), где б – адрес начала массива, i – индекс компоненты вектора, отсчитываемый от “1”, sizeof(T) – размер компоненты вектора).
Проблема в том, что разные ключи могут соответствовать одному адресу. Такие случаи называются коллизиями.
На такой случай нужно предусмотреть операцию рехеширования – переадресация, в случае занятости адреса (запись нового элемента) или в случаях, когда по данному адресу находится элемент с другим ключом (чтение).
2 способа:
1) организовать список строк с идентичным первичным ключом H(k) (этот список может расположить где душе угодно, но такой способ вызовет увеличение расхода памяти);
2) или, при занятости данной ячейки памяти, “впихнуть” наш элемент куда-нибудь рядом}

Достоинства таблицы с прямым доступом (хеш-таблицы) в том, что при линейных сложностях операций хеширования и рехеширования можно обрабатывать таблицу (читать, искать, добавлять новый элемент и удалять старые) за постоянное время.

Недостаток – нельзя напечатать так быстро заполненную таблицу в упорядоченном виде, да ещё и сохраняя естественный хронологический порядок равнозначных элементов (нужна сортировка → что было до хеша).

2.3 Составить программу сравнения линейных списков

main.c

```c
#include <stdio.h>

#include "list.h"

int main(void) {
    int n, m, i, v;
    list xs1, xs2;

    printf("Введите кол-во элементов в первом списке: ");
    scanf("%d", &n);

    printf("Введите кол-во элементов во втором списке: ");
    scanf("%d", &m);

    list_create(&xs1);
    list_create(&xs2);

    printf("Заполняем первый список:\n");
    for (i = 0; i < n; ++i) {
        printf("> ");
        scanf("%d", &v);
        list_push(&xs1, v);
    }
```

```
    printf("Заполняем второй список:\n");
    for (i = 0; i < m; ++i) {
        printf("> ");
        scanf("%d", &v);
        list_push(&xs2, v);
    }

    if (list_compare(&xs1, &xs2)) {
        printf("Списки равны.\n");
    } else {
        printf("Списки не равны.\n");
    }

    list_destroy(&xs1);
    list_destroy(&xs2);

    return 0;
}
```

list.h

```
#ifndef LIST_H
    #define LIST_H

    #include <stdlib.h>
    #include <stdbool.h>

    typedef struct _list_item {
        struct _list_item * next;
        int data;
    } list_item;

    typedef struct _list {
        int size;
        struct _list_item * head;
    } list;

    void list_create(list * xs);
    void list_push(list * xs, int value);
    bool list_compare(list * xs1, list * xs2);
    void list_destroy(list * xs);

#endif
```

list.c

```
#include "list.h"
void list_create(list * xs) {
    xs->head = malloc(sizeof(list_item));
    xs->head->next = NULL;
    xs->size = 0;
}

void list_push(list * xs, int value) {
    if (xs->size == 0) {
        xs->head->data = value;
    } else {
        list_item * curr;
        list_item * tmp = malloc(sizeof(list_item));
        tmp->data = value;

        curr = xs->head;
        while (curr->next != NULL) {
            curr = curr->next;
        }

        curr->next = tmp;
    }
```

```
    xs->size++;
}

bool list_compare(list * xs1, list * xs2) {
    if (xs1->size != xs2->size) {
        return false;
    } else {
        int i;
        list_item * curr1, * curr2;

        curr1 = xs1->head;
        curr2 = xs2->head;
        for (i = 0; i < xs1->size; i++) {
            if (curr1->data != curr2->data) {
                return false;
            }

            curr1 = curr1->next;
            curr2 = curr2->next;
        }

        return true;
    }
}

void list_destroy(list * xs) {
    list_item * curr,
              * prev,
              * tmp;

    curr = xs->head;

    if (curr->next == NULL) {
        free(curr);
        curr = NULL;
        return;
    }

    while (curr != NULL) {
        prev = curr;
        tmp = curr->next;
        free(prev);
        curr = tmp;
    }
}
```

3

3.1Вектор. Функциональная спецификация. Логическое описание и физическое представление.

Вектор

Вектор - это массив, который располагаются в куче, и его размер может быть задан произвольно. Недостатком таких массивов является большее среднее время доступа к элементу. Это связано с иерархичностью памяти современных ЭВМ. Главным источником критики векторов является обязанность программиста вручную очищать память, но в современных ЯП это делается автоматически.

Функциональная спецификация

Вектор является последовательностью динамической длины, и время доступа к элементам этой последовательности постоянно и не зависит от длины последовательности. Количество элементов вектора не фиксировано и всегда может быть изменено.

Как правило, операционные системы, выделяя блок памяти, возвращают адрес начала этого блока. Для доступа к компоненте вектора необходимо прибавить к адресу блока размер одной компоненты, умноженной на ее индекс.

Логическое описание и физическое представление

Примечание: Описать вектор на массиве и сделать следующие операции:
create, size, load, resize, equal, destroy.

Структура такая:

```
typedef struct _vector {
        T* data;
        int size;
} vector;
```

3.2Алгоритм Рутисхаузера

Алгоритм Рутисхаузера — один из ранних формальных алгоритмов разбора выражений со скобками, его особенностью является предположение о правильной скобочной структуре выражения, также алгоритмом не учитывается неявный приоритет операции. Впервые описан в 1951 году швейцарским математиком Хайнцем Рутисхаузером (нем. Heinz Rutishauser), был охарактеризован как «танцевальная процессия вокруг скобочных скал».

При обработке выражения: D:=((C-(B*L))+K)

алгоритм присваивает каждому символу из строки номер уровня по следующему правилу:

1. если это открывающаяся скобка или переменная, то значение увеличивается на 1;

2. если знак операции или закрывающаяся скобка, то уменьшается на 1.
Для выражения (A+(B*C)) присваивание значений уровня будет происходить следующим образом:

Номер символа	1	2	3	4	5	6	7	8	9
Символы строки	(	A	+	(	B	*	C	)	)
Номера уровней	1	2	1	2	3	2	3	2	1

Алгоритм складывается из следующих шагов:

1. выполнить расстановку уровней;
2. выполнить поиск элементов строки с максимальным значением уровня;
3. выделить тройку — два операнда с максимальным значением уровня и операцию, которая заключена между ними;
4. результат вычисления тройки обозначить вспомогательной переменной;
5. из исходной строки удалить выделенную тройку вместе с её скобками, а на её место поместить вспомогательную переменную, обозначающую результат, со значением уровня на единицу меньше, чем у выделенной тройки;
6. выполнять шаги 2 — 5 до тех пор, пока во входной строке не останется одна переменная, обозначающая общий результат выражения.

Символы строки	(	A	+	(	B	*	C	)	)
Номера уровней	1	2	1	2	3	2	3	2	1

Символы строки	(	A	+	BC	)
Номера уровней	1	2	1	2	1

Символы строки	A + BC результат!
Номера уровней	1

3.3Программа реверс дека

```c
#include <stdio.h>

#include "dequeue.h"

int main(void) {
    int val, n, i;
    dequeue d, tmp;

    dequeue_create(&d);
    dequeue_create(&tmp);

    printf("Введите кол-во чисел: ");
    scanf("%d", &n);

    printf("Вводите числа:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &val);
        dequeue_push_back(&d, val);
    }

    printf("Исходный дек:\n");
    dequeue_print(&d);

    while (d.size > 0) {
        val = dequeue_pop_back(&d);
        dequeue_push_back(&tmp, val);
    }

    while (tmp.size > 0) {
        val = dequeue_pop_front(&tmp);
        dequeue_push_back(&d, val);
    }
```

```
        printf("Перевёрнутый дек:\n");
        dequeue_print(&d);

        dequeue_destroy(&d);
        dequeue_destroy(&tmp);

        return 0;
    }
```

4

4.1Стек. Функциональная спецификация

Стек

Стек — тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Операции:

- `push`
- `is_empty`
- `top`
- `pop`

Функциональная спецификация

Тип S_T или стек объектов типа T, характеризуется операциями:

- СОЗДАТЬ: Ø ➞ S_T
- ПУСТО: S_T➞ boolean
- ДЛИНА: S_T ➞ N
- В_СТЕК: S_Tx T➞ S_T
- ИЗ_СТЕКА: S_T➞S_T
- ВЕРХ:S_T ➞T
- УНИЧТОЖИТЬ: S_T➞ Ø

4.2Деревья поиска

Если дерево организовано так, что для каждой вершины t_i справедливо утверждение, что все ключи левого поддерева t_j меньше ключа *C*, а все ключи правого поддерева *C* больше его, то такое дерево называют деревом поиска.

По построению дерева поиска, переходя к одному из поддеревьев, мы автоматически исключаем из рассмотрения другое поддерево, содержащее половину узлов.

Поиск по дереву с включениями

Требуется определить частоту вхождения каждого из слов в последовательность. Надо либо найти слово в дереве и добавить 1 к находящемуся в соответствующем узле счётчику его вхождений, либо, в случае неуспеха, включить новый элемент в дерево с сохранением нелинейного иерархического порядка и поисковой структуры и установить для него единичное значение счётчика.

Процедура поиска с включением search() сначала должна осуществлять поиск в глубину в дереве. В случае неудачи поиск оканчивается на одном из листьев дерева, к которому привела попытка поиска. Поскольку все предыдущие проверки на пути к этому листу пройдены, то мы имеем готовый маршрут поиска нового элемента, а данный лист представляет собой искомое место для вставки этого элемента в поисковое дерево.

```
typedef struct _word {
    char key;
    int count;
    struct _word * left, * right;
} word;

void search(char x, word *p) {
    if (!p) {
        p = malloc(sizeof(word));
        p->key = x;
        p->count = 1;
        p->left = p->right = 0;
```

```
    } else {
        if (x < p->key) {
            search(x, p->left);
        } else {
            if (x > p->key) {
                search(x, p->right);
            } else {
                p->count++;
            }
        }
    }
}
```

4.3Поиск глубины дерева на Си

Замечание: Данная реализация работает для бинарного дерева

```
typedef struct _tree_node {
    int val;
    struct _tree_node * left;
    struct _tree_node * right;
} tree_node;

int max(int a, int b) {
    return a >= b ? a : b;
}

int max_depth(tree_node * root) {
    if (root != NULL) {
       return max(max_depth(root->left), max_depth(root->right)) + 1;
    }

    return 0;
}
```

5

5.1Линейный список. Физическое представление. Итераторы.

Для обхода сложных динамических структур данных цепного или сплошного характера принято использовать так называемые итераторы.

Для удобства обхода списка, доступа к нему, определим объекты, обладающие функциями перехода от данного элемента списка к соседним. Зададим для них отношения равенства и неравенства. Два таких объекта равны тогда и только тогда, когда они указывают на один и тот же элемент списка. Также предоставим возможность чтения и записи элемента списка посредством введенных объектов. Такие объекты принято называть итераторами, и, конечно же, для них надо определить соответствующий тип данных.

Структура элемента списка:

```
struct Item
{
        struct Item* prev;
        struct Item* next;
        T data;
};
typedef struct
{
        Item* node;
} Iterator;
```

Равенство:

```
bool Equal(const Iterator* lhs, const Iterator* rhs)
{
        return lhs->node == rhs->node;
}
```

Переход на следующий элемент:

```
Iterator Next(Iterator* i)
{
        i->node = i->node->next;
        return i;
}
```

Переход на предыдущий элемент:

```
Iterator Prev(Iterator* i)
{
        i->node = i->node->prev;
        return i;
}
```
Чтение текущего элемента:
```
T Fetch(const Iterator* i)
{
        return i->node->data;
}
```
Запись данных в элемент списка:
```
void Store(const Iterator* i, const T t)
{
        i->node->data = t;
}
```
Во всех функция отсутствуют проверки, потому что они загромождают код, поэтому все проверки должны быть осуществлены пользователем.

5.2Алгоритм Бойера-Мура

.Боер и Д.Мур предложили алгоритм, который не только улучшал (относительно КМП) обработку самого худшего случая (когда в образце и строке многократно повторяется небольшое число букв), но и давал выигрыш в промежуточных ситуациях.
БМ алгоритм поиска строки считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке.

Идея: сканирование ***слева направо***, сравнение ***справа налево*** (характерно для семитских языков). Совмещается начало строки и образца, проверка начинается с последнего символа образца.

Если символы совпадают, производится сравнение предпоследнего символа шаблона и т.д.
Если все символы образца совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен.

Если же какой-то символ образца не совпадает с соответствующим символом строки, шаблон сдвигается на "**несколько**"символов вправо, и проверка снова начинается с последнего символа (эти "**несколько**" обычно равно **разнице** между [количеством литеров в образце] и [первой встречи (справа) данного символа в образце]).

Простым языком (я пытался) + пример:
1) Создаем **таблицу смещений** (одномерный массив) по следующему принципу. Допустим, у нас есть **образец: Hooligan**. Таблица для него будет выглядеть так: |7|5|5|4|3|2|1|8| (объясняю, таблица заполняется справа налево, номер самой правой литеры в начале пропускается. **Нумеруем литеры в образце по принципу**: номер литеры = на сколько она далеко от конца образца. Если имеются **две одинаковые литеры**, то они **нумеруются одинаково**, **по значению наименее удаленной от конца образца** (у нас тип дважды "о" встречается, и тк в первый раз она удалена на 5, то и в следующие разы тож пятерочку ставим). **Касаемо первой литеры справа**: если она встретилась еще раз в образце, то нумеруем так же, как и ту, которая встретилась. Если же эта литера у нас в образце единственная, то ее номер = длина образца. **Символам, не вошедшим в образ сопоставляется длина образца** (в нашем случае 8)**.**

2) Дана строка:
Holydans GoolygansHoolygans ,сравниваем ее с образом **Hooligan справа налево**
Hooligan

На первом же символе **"n" не совпало с "s"**. Мы берем символ **"s" (из строки, а не образца),** и смотрим его номер по таблице. **Его номер = 8 (тк "s" - символ, не вошедший в образец). →**
→ Мы двигаем образец вдоль строки на 8 позиций

_____Hooligan
Тут у нас сразу "a" не совпадает с "n". a = 1 по таблице, двигаем образ на один. А вот дальше уже у нас совпадает часть образца со строкой, но только часть . **В этом случаем мы двигаем строку на значение крайнего правого элемента образца aka первого совпадения (в нашем случае на 8). И так далее до либо нахождения всех вхождений, либо до момента, когда ниче не войдет, и вообще F.**
(тут заебатый урок кстати https://www.youtube.com/watch?v=KIUHWMwavQg)

Кстати таблицу смещений можно делать по таблице ASCII

Достоинствами БМ является то, почти всегда, кроме специально построенных случаев (**худший - строка ааа...аа, образ баа……аа, временная сложность O(N*M), лучший - строка аааа...аааа, образ бббб…..бббб)**, он требует значительно меньше H сравнений, проходя “саженью образа” по строке. Лучшая оценка, когда последний символ образа всегда не совпадает с соответствующими литерами строки, проходимой аршином M, равно H/M.

В своей публикации (которая была после Кнута, Морриса и Пратта) Боер и Мур приводили соображения о дальнейшем улучшение алгоритма. Одно из которых – объединение стратегий БМ (больший сдвиг во время несовпадений) и КМП (существенный сдвиг при частичном совпадение), которая основывалась на 2х таблицах, из которых выбиралось большее смещение.

5.3Нахождение одинаковых элементов дерева

Под поиском в ширину понимается описание способа, позволяющего выполнить обход вершин графа. Например, имеем граф G = (V, E), у которого выполнено выделение исходной вершины s. Согласно алгоритма обхода в ширину, нужно последовательно обойти каждое ребро графа G, чтобы побывать на каждой его вершине, которую возможно достигнуть из вершины s. Параллельно необходимо определять самое маленькое число рёбер (длину маршрута) от s до всех достижимых из неё вершин.

Решение:

Делаем обход в ширину и заносим уникальные вершины в новый вектор

При поиске в ширину:

(1) поместить в пустую очередь корень дерева
(2) если очередь узлов пуста, то конец обхода
(3) извлечь первый элемент из очереди и поместить в ее конец всех его сыновей по старшинству
(4) повторить поиск, начиная с п.2

```
typedef struct _vector_item {
    struct _vector_item * next;
    struct _vector_item * prev;
    struct _tree_node * node;
} vector_item;

typedef struct _vector {
    int size;
    struct _vector_item * front;
    struct _vector_item * back;
} vector;

typedef struct _tree_node {
    char id;
    struct _vector * children;
} tree_node;

vector * find_same_nodes(tree_node * root) {
    vector_item * curr;
    vector queue;
    vector result;

    vector_create(&queue);
    vector_create(&result);

    curr = root->children->front;
    while (curr != NULL) {
        vector_push_back(&queue, curr);
        curr = curr->next;
    }

    while (queue.size > 0) {
        curr = vector_pop_front(&queue);

        if (result.size == 0 || vector_in(&result, curr)) {
            vector_push_back(&result, curr);
        }

        curr = curr->node->children->front;
        while (curr != NULL) {
            vector_push_back(&queue, curr);
```

```
            curr = curr->next;
        }
    }

    vector_destroy(&queue);

    return &result;
}
```

6

6.1 Двоичное дерево. Физическое представление. Прошивка.

Физическое представление (массив)

Введём жёсткое размещение элементов дерева в массиве. В первом элементе массива разместим корень дерева, во 2-ом и 3-ем - его левого и правого потомков. Далее поместим пары потомков потомков и т. д. Сыновья элемента дерева с индексом i хранятся в элементах массива с индексами $2i$ и $2i + 1$. Согласно данной схеме размещения, j-ый элемент i-ого уровня имеет индекс $2^{i-1} + j - 1$.

Этот метод весьма экономичен по памяти. Основным неудобством сплошного представления дерева является высокая цена вставки и удаления элементов.

Для деревьев можно использовать рекурсивные ссылочные представления, которые были разработаны для списков, но теперь направляются к левому и к правому поддеревьям соответственно. Тип для вершины дерева, конечно же, фиксирован, но имеет двойное рекурсивное разветвление.

Прошивка

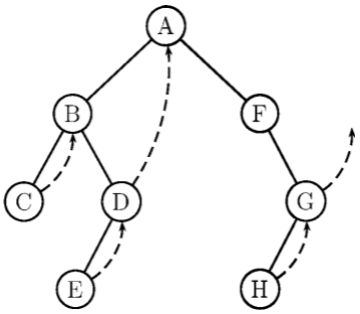
В прошитых бинарных деревьях вместо пустых указателей используются специальные связи-нити и каждый указатель в узле дерева дополняется однобитовым признаком ltag и rtag, соответственно. Признак определяет, содержится ли в соответствующем указателе обычная ссылка на поддерево или в нем содержится связь-нить.

Связь-нить в поле l указывает на узел - предшественник в обратном порядке обхода (inorder), а связь-нить в поле r указывает на узел - преемник данного узла в обратном порядке обхода.

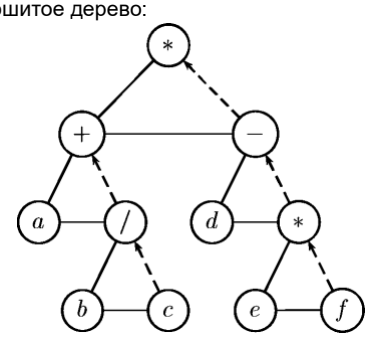
Введение признаков ltag и rtag не приводит к сколько-нибудь значительному увеличению затрат памяти, зато упрощает алгоритм обхода деревьев, так как для прошитых деревьев можно выполнить нерекурсивный обход без использования стека.

Также существуют правопршитые (левопрошитые), в которых есть только rtag (ltag), для экономии памяти.

Правопршитое дерево:



Левопрошитое дерево:



6.2 Турнирные сортировки.

Существует модификация сортировки выборкой, оставляющая после каждого прохода гораздо больше порядковой информации, чем просто минимальное значение. В сортируемом множестве можно сравнить пары соседних элементов. В результате $n/2$ сравнений мы получим такое же число победителей - элементов с меньшими ключами. Если среди победителей провести такое же сравнение, мы получим n/4 меньших элементов. Продолжая процесс мы получим дерево выбора минимального элемента.

Для практической реализации сортировки с помощью дерева выбора осталось лишь указать способ эффективного построения и использования этого дерева.

Пирамида определяется как последовательность ключей, h_L, h_{L+1},... , h_R такая, что $h_i \leq h_{2i}$ и $h_i \leq h_{2i+1}$для i = L, … , R/2. Пирамида отличается от турнирного дерева отсутствием дубликатов элементов сортируемой последовательности.

Сформулируем алгоритм вставки элемента в пирамиду. Как уже говорилось, первоначально элемент помещается в ее вершину, i = 1. Рассмотрим тройку элементов с индексами i, 2i, 2i + 1. Во-первых, по построению пирамиды, элементы 2i и 2i + 1 являются потомками i-того элемента. Выберем наименьший из этих трех элементов. Если он находится на i-той позиции, то вставка элемента

завершается. Иначе обозначим позицию наименьшего элемента буквой j, где j = 2i или j = 2i + 1. Поменяем местами элементы i и j, положим i := j и продолжим процесс сравнений и обменов с i-той позиции, куда в результате итерации был помещен вставляемый элемент.

```
void sift(int l, int r)
{
        int i, j, z, x;
        i = l;
        j = 2*l;
        x = a[l];
        if(j <= r && a[j] < a[j+1])
                j++;
        while(j <= r && x < a[j])
        {
                z = a[i];
                a[i] = a[j];
                a[j] = z;
                i = j;
                j = 2*j;
                if(j < r && a[j] < a[j+1])
                        j++;
        }
}

void heapsort()
{
        int l, r, x;
        l = n/2 + 1;
        while(l > 1)
        {
                l--;
                sift(l, n);
        }
        r = n;
        while(r > 1)
        {
                x = a[0];
                a[0] = a[r];
                a[r] = x;
                sift(0, r);
        }
}
```

Максимальное число перестановок: $M = \frac{n \log n}{2}$;

6.3Реверс файла на си.

С помощью fseek

С помощью стека. Помещаем каждый символ в стек и после прочтения файла, извлекаем элементы стека в результирующий файл.

Рекурсивный

```
#include "stdio.h"

void reverse_file(FILE * input, FILE * output){
    char c;

    if(feof(input)) {
        return;
    } else {
        c = fgetc(input);
        reverse_file(input, output);
        fputc(c, output);
    }
```

```
}

int main(int argc, char * argv[]) {
    FILE * input, * output;

    input = fopen(argv[1], "r");
    output = fopen(argv[2], "w");

    reverse_file(input, output);

    return 0;
}
```

7

7.1Динамические и статические объекты

Свойства объекта, которые остаются неизменными при любом исполнении называются **статическими**. Их можно определить по тексту программы. **Пример:** тип объекта.

Свойства объекта, изменяемые во время работы программы называются **динамическими**.

Пример: конкретное значение переменной.

Одна крайняя позиция представлена концепцией неограниченного динамизма, когда по существу любая характеристика обрабатываемого объекта может быть изменена при выполнении программы. Такая концепция не исключает прогнозирования и контроля, но и не связывает их жестко со структурой текста программы.

Другая крайняя позиция выражена в стремлении затруднить программисту всякое изменение характеристик объектов.

Если память выделяется в процессе трансляции и ее объем не меняется от начала до конца выполнения программы, то такой объект является статическим.

Если же память выделяется во время выполнения программы и ее объем может меняться, то такой объект является динамическим.

7.2Адресный тип. Полиморфизм с Си при помощи адресного типа

Адресный тип имеет множеством значений диапазон допустимых адресов ЭВМ. Также адресный тип совместим с целым и к нему применимы арифметические операции. Адресный тип полностью устраняет контроль типов, выполняемый компилятором, и его следует употреблять только для разработки родовых модулей низкого уровня.

Для реализации полиморфных функций, зависящих от типа передаваемых аргументов, в C++ используется понятие шаблона функции. Полиморфизм полезно применять для создания функции общего назначения:

Пример:

```
template<class T>
T max(T a, T a) {
    return a >= b ? a : b;
}
```

Здесь функция max возвращает максимальное значение при условии, если оператор >= перегружен.

7.3Сортировка стека

```
void stack_sort(stack * s) {
    int curr_val;
    stack t1, t2;

    stack_init(&t1);
    stack_init(&t2);

    curr_val = stack_pop(s);
    stack_push(&t1, curr_val);

    while (s->size > 0) {
        curr_val = stack_pop(s);

        if (stack_top(&t1) <= curr_val) {
            stack_push(&t1, curr_val);
        } else {
            while (stack_top(&t1) > curr_val && t1.size > 0) {
                stack_push(&t2, stack_pop(&t1));
            }

            stack_push(&t1, curr_val);
```

```
                while (t2.size > 0) {
                    stack_push(&t1, stack_pop(&t2));
                }
            }
        }

        stack_destroy(s);
        s = &t1;
        stack_destroy(&t2);

    }
```

8

8.1 Линейный список. Функциональная спецификация.

Линейный список

Линейные списки позволяют представить последовательность элементов так, чтобы каждый элемент был бы доступен вне зависимости от положения.

Линейный список - это представление в ЭВМ конечного упорядоченного динамического мультимножества. Элементы этого множества линейно упорядочены, но порядок определяется не индексами, а относительным расположением элементов. Линейные списки естественно использовать всякий раз, когда встречаются упорядоченные множества переменного размера.

Добавление/удаление элемента в конец: O(1), а в произвольное место O(n)

Функциональная спецификация

Тип L_T или двусвязный список объектов типа T, характеризуется операциями:

- СОЗДАТЬ: Ø ➞ L_T
- ПУСТО: L_T➞ boolean
- ДЛИНА: L_T ➞ N
- ПЕРВЫЙ:L_T ➞T
- ПОСЛЕДНИЙ:L_T ➞T
- СЛЕДУЮЩИЙ:L_Tx T➞ L_T
- ПРЕДЫДУЩИЙ:L_Tx T➞ L_T
- ВСТАВКА: L_Tx T x T➞ L_T
- УДАЛЕНИЕ: L_Tx T➞ L_T
- УНИЧТОЖИТЬ: L_T➞ Ø

8.2Сортировка Хоара.

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void qs(int *a, int start, int end) {
    if (start < end) {
        int left = start,
            right = end,
            middle = a[(left + right) / 2];

        do {
            while (a[left] < middle) left++;
            while (a[right] > middle) right--;

            if (left <= right) {
                swap(&a[left], &a[right]);
                left++;
                right--;
            }
        } while (left <= right);

        qs(a, start, right);
```

```
        qs(a, left, end);
    }
}
```

8.3Транспонирование матрицы из файла

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_DIMENTION 1000

int main(void) {
    int num, // текущее число
        offset, // отступ для sscanf (https://stackoverflow.com/questions/3975236/how-to-use-sscanf-in-loops)
        // счётчики размерности матрицы
        i,
        j,
        //
        n,
        m,
        matrix[MAX_DIMENTION][MAX_DIMENTION]; // массив для матрицы

    char * line; // текущая строка
    size_t len = 0; // длина считанной строки
    FILE * input_file = NULL;

    input_file = fopen("input.txt", "r");

    // заполняем массив
    i = 0;
    while (getline(&line, &len, input_file) != -1) {
        j = 0;
        while (sscanf(line, " %d%n", &num, &offset) == 1) {
            matrix[i][j++] = num;
            line += offset;
        }
        i++;
    }

    // выводим результат
    for (n = 0; n < j; n++) {
        for (m = 0; m < i; m++) {
            printf("%d ", matrix[m][n]);
        }
        printf("\n");
    }

    return 0;
}
```

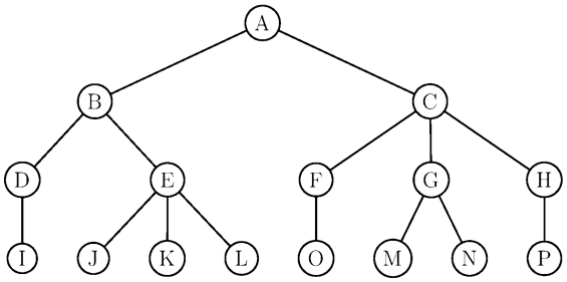
9

9.1 Представление и обработка дерева общего вида

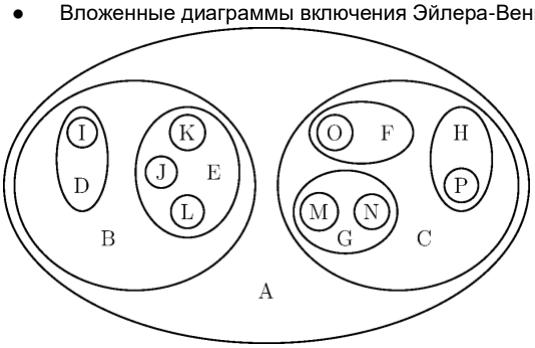
Деревья - структура данных, наиболее приспособленная для решения задач искусственного интеллекта и синтаксического анализа. Пусть T - некоторый тип данных. Деревом типа T называется структура, которая образована элементом типа T, называемым корнем дерева, и конечным, возможно пустым, множеством с переменным числом элементов - деревьев типа T, называемых поддеревьями этого дерева. Это рекурсивное определение похоже на определения файла, очереди, стека и списка, но отличается от них нелинейностью ввиду наличия разветвлений. Виды

Виды представления деревьев:

- Стандартное



- Вложенные диаграммы включения Эйлера-Венна



- иерархические скобочные структуры:

(A (B (D (I), E (J, K, L)), C (F (O), G (M, N),H (P))))

- Матрица смежности

Для описания взаимного расположения узлов принята терминология генеалогических деревьев,
Определенные нами таким образом деревья называются деревьями общего вида или сильно ветвящимися.

9.2Алгоритм Рабина-Карпа

Рабин и Карп изобрели еще один алгоритм поиска подстрок, который эффективен на практике и к тому же обобщается на многомерный случай (например, поиск на двумерной решетке). Хотя в худшем случае время работы алгоритма Рабина-Карпа 0(M(N-M +1)), в среднем он работает достаточно быстро.

Предположим, что алфавит последовательности и образца имеют простую числовую интерпретацию. Если p[0..M-1] - образец, то через ρ обозначим число, десятичной записью которого он является. В последовательности s = [0..N-1] обозначим через S_jчисло, десятичным изображением которого является подстрока s[j..j+M-1].

Очевидно, j является допустимым сдвигом при ускоренном поиске тогда и только тогда, когда соответствующая подстрока S_j совпадает с образцом ρ. ρ вычисляется по схеме Горнера.

9.3 Написать программу на Си, которая подсчитывает количество различных элементов дерева

```
typedef struct _vector_item {
    struct _vector_item * next;
    struct _vector_item * prev;
    struct _tree_node * node;
} vector_item;

typedef struct _vector {
    int size;
    struct _vector_item * front;
```

```
    struct _vector_item * back;
} vector;

typedef struct _tree_node {
    char id;
    struct _vector * children;
} tree_node;

int count_various_nodes(tree_node * root) {
    int count;
    vector_item * curr;
    vector queue;
    vector result;

    vector_create(&queue);
    vector_create(&result);

    curr = root->children->front;
    while (curr != NULL) {
        vector_push_back(&queue, curr);
        curr = curr->next;
    }

    while (queue.size > 0) {
        curr = vector_pop_front(&queue);

        if (result.size == 0 || !vector_in(&result, curr)) {
            vector_push_back(&result, curr);
        }

        curr = curr->node->children->front;
        while (curr != NULL) {
            vector_push_back(&queue, curr);
            curr = curr->next;
        }
    }
    count = result.size;

    vector_destroy(&queue);
    vector_destroy(&result);

    return count;
}
```

10

10.1 Алгоритм Дейкстры

Способ разбора математических выражений, представленных в обычной инфиксной форме. Результат - обратная польская запись или дерево выражений.

Алгоритм работает при помощи стека.

- Пока не все токены обработаны:
- Прочитать токен.
- Если токен — *число*, то добавить его в очередь вывода.
- Если токен — *функция*, то поместить его в стек.
- Если токен — *разделитель аргументов функции* (например запятая):
- Пока токен на вершине стека не *открывающая скобка*, перекладывать операторы из стека в выходную очередь. Если в стеке не было *открывающей скобки*, то в выражении пропущен *разделитель аргументов функции* (запятая), либо пропущена *открывающая скобка*.
- Если токен — *оператор* op1, то:
- Пока присутствует на вершине стека токен *оператор* op2, и
  Либо *оператор* op1 лево-ассоциативен и его приоритет меньше, чем у *оператора* op2 либо равен,
  или *оператор* op1 право-ассоциативен и его приоритет меньше, чем у op2,
  переложить op2 из стека в выходную очередь;
- (Иначе, когда стек операторов пуст или содержит открывающую скобку)
  - положить op1 в стек.

- Если токен — *открывающая скобка*, то положить его в стек.
- Если токен — *закрывающая скобка*:
- Пока токен на вершине стека не является *открывающей скобкой*, перекладывать операторы из стека в выходную очередь.
- Выкинуть *открывающую скобку* из стека, но не добавлять в очередь вывода.
- Если токен на вершине стека — функция, добавить её в выходную очередь.
- Если стек закончился до того, как был встречен токен *открывающая скобка*, то в выражении пропущена скобка.
- Если больше не осталось токенов на входе:
- Пока есть токены операторы в стеке:
- Если токен оператор на вершине стека — скобка, то в выражении присутствует незакрытая скобка.
- Переложить оператор из стека в выходную очередь.
- Конец.

Каждый токен-число, функция или оператор выводится только один раз, а также каждый токен-функция, оператор или круглая скобка будет добавлен и удален из стека по одному разу. Постоянное количество операций на токен, линейная сложность алгоритма O(*n*).

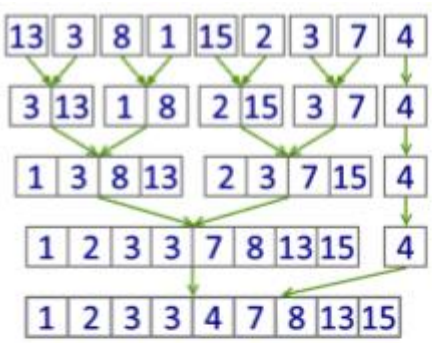
Также Дейкстра придумал алгоритм поиска кратчайших путей от начала до всех вершин в графе. Алгоритм:

```
while queue.size > 0:
    node = queue.dequeue()

    if node == 'goal':
        break

    for child in graph[node]:
        new_cost = costs[node] + graph.cost(node, child)
        if (child not in costs) or (new_cost < costs[child]):
            costs[node] = new_cost
            priority = new_cost
            queue.enqueue(child, priority)
```

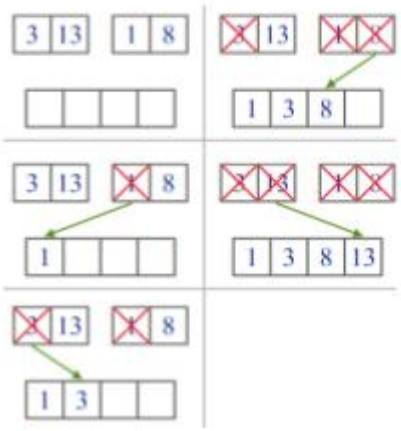
10.2 Сортировка слиянием

Решения задачи сортировки:

1) последовательность a разбивается на 2 половины b и c
2) каждая из получившихся частей сортируется отдельно тем же самым алгоритмом
3) два упорядоченных массива половинного размера соединяются в один.
4) части b и c сливаются; при этом одиночные элементы из разных частей образуют упорядоченные пары в выходной последовательности, т.е. первым в ней оказывается меньший из 2х первых элементов
5) полученная последовательность более упорядочена чем изначальная и она снова подвергается разделению и слиянию; при этом упорядоченные пары переходят в 4ки, те – в 8ки и т.д.



Объединение:



ример: (40 47 10 38 80 20 01 50)→(40 47 10 38) (80 20 01 50) →(40 80 20 47 01 10 38 50) →(40 80 20 47)(01 10 38 50) →(01 10 40 80 20 38 47 50) →(01 10 40 80)(20 38 47 50) →(01 10 20 38 40 47 50 80)

ограмма (мб и для примера выше подходит):

```
void merge(int a[]; long lb, long split, long ub){
    it1 = 0;
    it2 = 0;
    int result[ub - lb];

    while (lb + it1 < split && split + it2 < ub){
        if (a[lb + it1] < a[split + it2]){
            result[it1 + it2] = a[lb + it1];
            it1 += 1;
        } else{
            result[it1 + it2] = a[split + it2];
            it2 += 1;
          }
    }
    while (left + it1 < split){
        result[it1 + it2] = a[left + it1];
        it1 += 1;
    }
    while (split + it2 < rb){
        result[it1 + it2] = a[split + it2];
        it2 += 1;
    }
    for (int i = 0; i < it1 + it2; ++i)
        a[lb + i] = result[i];
     }


void mergeSort(int a[], long lb, long ub) {
```

```
  long split;                    // индекс, по которому делим массив

  if (lb < ub) {                 // если есть более 1 элемента
    split = (lb + ub)/2;

    mergeSort(a, lb, split);       // сортировать левую половину
    mergeSort(a, split+1, last);// сортировать правую половину
    merge(a, lb, split, ub);    // слить результаты в общий массив
  }
}
```

10.3 Удаление листов

```
typedef struct _vector_item {
    struct _vector_item * next;
    struct _vector_item * prev;
    struct _tree_node * node;
} vector_item;

typedef struct _vector {
    int size;
    struct _vector_item * front;
    struct _vector_item * back;
} vector;

typedef struct _tree_node {
    char id;
    struct _tree_node * parent;
    struct _vector * children;
} tree_node;


void delete_leaf(tree_node * root) {
    vector_item * curr, * parent;
    vector queue;

    vector_create(&queue);

    curr = root->children->front;
    while (curr != NULL) {
        vector_push_back(&queue, curr);
        curr = curr->next;
    }

    while (queue.size > 0) {
        curr = vector_pop_front(&queue);

        if (curr->node->children->size == 0) {
            free(curr);
            curr = NULL;
        } else {
            curr = curr->node->children->front;
            while (curr != NULL) {
                vector_push_back(&queue, curr);
                curr = curr->next;
            }
        }

    }

    vector_destroy(&queue);
}
```

11

11.1Двоичное дерево. Функциональная спецификация.

Тип BT_T или двоичное дерево типа T, определяется так:

- СОЗДАТЬ: Ø ➞ BT_T
- ПОСТРОИТЬ: BT_Tx T x BT_T ➞ BT_T
- ПУСТО: BT_T ➞ boolean

- КОРЕНЬ: BT_T ➞T
- СЛЕВА: BT_T➞ BT_T
- СПРАВА: BT_T➞ BT_T
- УНИЧТОЖИТЬ: BT_T➞ Ø

Свойства:

1. ПУСТО(СОЗДАТЬ) = true
2. ПУСТО(ПОСТРОИТЬ(btl, t, btr)) = false
3. КОРЕНЬ(ПОСТРОИТЬ(btl, t, btr)) = t
4. СЛЕВА(ПОСТРОИТЬ(btl, t, btr)) = btl
5. СПРАВА(ПОСТРОИТЬ(btl, t, btr)) = btr
6. ПОСТРОИТЬ(СЛЕВА(bt), КОРЕНЬ(bt), СПРАВА(bt)) = bt

Операции над деревьями:

- чтение данных из узла дерева;
- создание дерева, состоящего из одного корневого узла;
- построение дерева из заданных корня и нескольких поддеревьев;
- присоединение к узлу нового поддерева;
- замена поддерева на новое поддерево;
- удаление поддерева;
- получение узла, следующего за данным в определённом порядке.

11.2 Алгоритм Бауэра-Замельзона.

Из ранних стековых методов рассматривается алгоритм Бауэра и Замельзона. Алгоритм использует два стека и таблицу функций перехода. Один стек используется при трансляции выражения, а второй - во время интерпретации выражения.
Обозначения: T - стек транслятора, E - стек интерпретатора.
В таблице переходов задаются функции, которые должен выполнить транслятор при разборе выражения. Возможны шесть функций при прочтении операции из входной строки:
- f1 - заслать операцию из входной строки в стек T; читать следующий символ строки;
- f2 - выделить тройку - взять операцию с вершины стека T и два операнда с вершины стека E; вспомогательную переменную, обозначающую результат, занести в стек E; заслать операцию из входной строки в стек T; читать следующий символ строки;
- f3 - исключить символ из стека T; читать следующий символ строки;
- f4 - выделить тройку - взять операцию с вершины стека T и два операнда с вершины стека E; вспомогательную переменную, обозначающую результат, занести в стек E; по 0-таблице определить функцию для данного символа входной строки;
- f5 - выдача сообщения об ошибке;
- f6 - завершение работы.

Таблица переходов для алгебраических выражений будет иметь вид(символ $ является признаком пустого стека или пустой строки):

Операция на вершине стека T	Операция из входной строки $	(	+	-	*	/	)
$	6	1	1	1	1	1	5
(	5	1	1	1	1	1	3
+	4	1	2	2	1	1	4
-	4	1	2	2	1	1	4
*	4	1	4	4	2	2	4
/	4	1	4	4	2	2	4

Алгоритм просматривает слева направо выражение и циклически выполняет следующие действия: если очередной символ входной строки является операндом, то он безусловно переносится в стек E; если же операция, то по таблице функций перехода определяется номер функции для выполнения. Для выражения A+(B-C)*D ниже приводится последовательность действий алгоритма.

стек E	стек T	Входной символ	Номер функции	Тройка
$	$	A		
$A	$	+	1	
$A	$+	(	1	
$A	$+(	B		
$AB	$+(	-	1	
$AB	$+(-	C		
$ABC	$+(-	)	4	- B C ->R
$AR	$+(	)	3	
$AR	$+	*	1	
$AR	$+*	D		

```
|$ARD  | $+*   | $   |   4      |* R D ->Q |
|$AQ    | $+    | $   |   4      |+ A Q ->S |
|$S     | $     | $   |Конец |        |
```

11.3Конкатенация трех файлов

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    FILE * file1,
         * file2,
         * file3,
         * output;

    char c;

    // открываем файлы на чтение
    file1 = fopen(argv[1], "r");
    file2 = fopen(argv[2], "r");
    file3 = fopen(argv[3], "r");

    output = fopen(argv[4], "w");

    while ((c = fgetc(file1)) != EOF)
        fputc(c, output);

    while ((c = fgetc(file2)) != EOF)
        fputc(c, output);

    while ((c = fgetc(file3)) != EOF)
        fputc(c, output);

    fclose(file1);
    fclose(file2);
    fclose(file3);
    fclose(output);

    return 0;
}
```

12

12.1Дек. Сравнительное описание. Примеры задач

Дек - двухсторонняя очередь, динамическая структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец.

Операции над деком:

- включение элемента справа и слева
- исключение элемента справа и слева
- определение размера
- очистка

Пример задачи: Птицы, сидевшие на проводе, и разозлившиеся от поднятой пыли начали бежать по проводу в разные стороны (право и лево). Птицы бегут с одинаковой скоростью, а встретившись, начинают бежать в противоположном направлении. Если птица добегает до конца провода, то она взлетает. Необходимо написать программу, которая при данных длине провода, начальным позициям и направлениям бега выяснит, в какой момент времени каждая улетит с провода.

Решение: Введение двух деков. При взлете птицы из дека удаляется крайний элемент, а сами деки меняются местами. Храня пары <дек; сколько надо добавить к числам в нем>, можно находить через какое время и в какую сторону улетит очередная птица. Поскольку порядок птиц не меняется, дополнительно следует хранить сортированный массив координат начальных положений птиц, чтобы определить, какая из них покинула провод.

12.2Алгоритмы обхода деревьев

Для бинарного дерева определены три варианта обхода: прямой (сверху вниз, preorder), обратный (слева направо, inorder) и концевой (снизу вверх, posrorder). Рассмотрим их поподробнее:

1. КЛП (preorder)
    (a) если дерево пусто, то конец обхода;
    (b) берется корень;
    (c) выполняется обход левого поддерева;
    (d) выполняется обход правого поддерева.
2. ЛКП (inorder)
    (a) если дерево пусто, то конец обхода;
    (b) выполняется обход левого поддерева;

(c) берется корень;
(d) выполняется обход правого поддерева.

3. ЛПК (postorder)
(a) если дерево пусто, то конец обхода;
(b) выполняется обход левого поддерева;
(c) выполняется обход правого поддерева.
(d) берется корень;

При обходе деревьев могут использоваться либо рекурсивные процедуры, либо программы с циклом, зависящим от стека.

12.3 Задача - не рекурсивная сортировка Хоара

```
void Quick(int arr[], int n)
{
  int base, left, right, i, j;
  base = left = right = i = j = 0;
  Stack stack;

  stack.Push(n - 1);
  stack.Push(0);
  do {                // while (stack.GetSP() != NULL)
    left  = stack.Pop();
    right = stack.Pop();
    if (((right - left) == 1) && (arr[left] > arr[right]))
        swap(arr[left], arr[right]);
    else {

      base = arr[(left + right) / 2];
      i = left;
      j = right;
      // цикл продолжается, пока индексы i и j не сойдутся
      do {        // while (i > j)
        // пока i-ый элемент не превысит опорный
        while ((base > arr[i]))
          ++i;
        // пока j-ый элемент не окажется меньше опорного
        while (arr[j] > base)
          --j;
        if (i <= j)
          swap(arr[i++], arr[j--]);
      } while (i <= j);
    }
    if (left < j) {
      stack.Push(j);
      stack.Push(left);
    }
    if (i < right) {
      stack.Push(right);
      stack.Push(i);
    }
  } while (stack.GetSP() != NULL);
}
```

13	

13.1Файл. функциональная спецификация (208 стр)

Файл

Динамические объекты могут размещаться не только в основной, но и во внешней памяти ЭВМ. Файлы могут быть внешними (долгосрочное хранение информации) и внутренними (память для них выделена временно).

Функциональная спецификация

Обозначим через F_T файловый тип с компонентами типа T. Множество значений файлового типа строго определяется следующими правилами:

1. {} есть файл типа F_T(пустая последовательность или пустой файл);
2. если f есть файл F_T и t есть объект типа T, то f + {t} есть файл типа F_T (+ - конкатенация);
3. никакие другие значения не являются файлами типа F_T;

Базовое множество атрибутов файлового типа:

- операция конкатенации;
- операция присваивания;
- отношения равенства;
- функции: создание, доступ, модификация, уничтожение.

13.2Дерево поиска (303 стр)

Если дерево организовано так, что для каждой вершины t_i справедливо утверждение, что все ключи левого поддерева t_j меньше ключа *C*, а все ключи правого поддерева *C* больше его, то такое дерево называют деревом поиска.

По построению дерева поиска, переходя к одному из поддеревьев, мы автоматически исключаем из рассмотрения другое поддерево, содержащее половину узлов.

13.3Вычислить константное булевое выражение с помощью обратной польской записи

Примечание: выражение вводится в постфиксной форме.

```
#include <stdio.h>

#include "stack.h"

int main() {
    char c, a, b;    stack s;

    stack_create(&s);

    while ((c = getchar()) != EOF) {
        if (c == '0' || c == '1') {
            stack_push(&s, c - '0');
        } else if (c == '+') {
            a = stack_pop(&s);
            b = stack_pop(&s);

            if (a && b)
                stack_push(&s, a);
            else
                stack_push(&s, a + b);
        } else if (c == '*') {
            a = stack_pop(&s);
            b = stack_pop(&s);
            stack_push(&s, a * b);
        }
    }

    printf("%d\n", stack_top(&s));
    return 0;
}
```

14	

14.1Уровни описания структур данных.

- Функциональная спецификация типа данных — внешнее формальное определение, не зависящее от языка,

оборудования. Также описывается допустимое мно-во значений. Дать формальное определение типа данных - это значит задать множество значений этого типа и множество изображений этих значений вместе с правилом их интерпретации.

- Логическое описание — отображение функциональной спецификации на средства конкретного языка программирования. Если тип данных не атомарный, то необходима декомпозиция объекта на такие составные части, которые могут быть описаны средствами выбранного языка программирования.
- Физическое представление — конкретное отображение на память машины объёма программы в соответствии с логическим описанием. Конструктивные особенности памяти имеют два вида физического представления:
  - Сплошное - это представление, при котором объект размещается в памяти машины в непрерывной последовательности единиц хранения. Данное представление имеет эффективную аппаратную поддержку и используется для таких структур как векторы, массивы, деки.
  - Цепное представление - это такое представление, при котором значение объекта разбивается на отдельные части, которые могут быть расположены в разных участках памяти машины, причем эти участки тем или иным способом связаны «в цепочку» с помощью указателей. Цепное представление используется, как правило, для динамических структурных объектов (списки, деревья, очереди, стеки и деки).

14.2 Процедурный тип. Полиморфизмы на си.

Процедурный тип представляет собой класс типов, отдельные типы которого есть множества глобально определенных процедур с идентичной спецификацией, включая и тип результата для функций. Константами процедурных типов являются имена глобальных процедур. Переменные процедурных типов принимают значения из соответствующего спецификации заголовка множества процедур. Процедурную переменную можно сравнить с однотипной или присвоить ей допустимое значение другой переменной или константы того же типа.

Поскольку для вызова процедуры в конечном счете нужно знать ее адрес, его роль может играть ссылка на процедуру.

Процедурный тип позволяет просто и систематически описывать более универсальные родовые модули с хранимыми процедурами.

В Си и C++ таких тонких понятий, как процедурный тип, просто нет. Вместо этого предусмотрены указатели на функции.

```
typedef double (*binary_function)(double, double); /*Указатель на функцию двух переменных*/
double f(double x, double y) {
        return x + y;
}
binary_function pf = f;
double z0 = pf(1, 2); // Вызов функции через указатель
double z1 = (*pf)(3, 4); // Тоже вариант вызова
```

14.3Проверка вложенности скобок

```
bool test_brackets_sequence(char * str) {
    int i;
    char c, top;
    bool result;
    stack s;

    stack_create(&s);

    for (i = 0, c = str[i]; c != '\0'; c = str[++i]) {
        if (c == '(' || c == '{' || c == '[') {
            stack_push(&s, c);
        } else {
            if (s.size == 0) {
                return false;
            }

            top = stack_pop(&s);
            if ((c == '[' && top != ']') || (c == '{' && top != '}') || (c == '(' && top != ')'))
{
                return false;
            }
        }
    }

    result = s.size == 0;
    stack_destroy(&s);

    return result;
}
```

15

15.1Абстракции в языках программирования (409 стр)

Абстракция - это не только отвлечение от чего-то несущественного ,но также инструмент познавательной деятельности человека, приводящий к абстрактным понятиям.

АТД - это определение некоторого понятия в виде класса объектов с некоторыми свойствами и операциями.

В определение АТД входят следующие четыре части:

- внешность (видимая часть, сопряжение, интерфейс), содержащая имя определяемого типа (понятия), имена операций с указанием типов их аргументов и значений и т. п.
- абстрактное описание операций и объектов, с которыми они работают
- конкретное (логическое) описание этих операций на обычном распространенном языке программирования
- описание связи между 2 и 3, объясняющее, в каком смысле часть 3 корректно представляет часть 2

15.2Сортировка Шелла (360 стр)

Сортировка Шелла является улучшением сортировки вставками. Для ускорения он предложил выделять в сортируемой последовательности периодические подпоследовательности регулярного шага, в каждой из которых отдельно выполняется обычная сортировка вставкой. Эти подпоследовательности дают большие перемещения элементов. После каждого прохода шаг подпоследовательностей уменьшается и сортировка повторяется с новыми прыжками. Последней выполняется сортировка с шагом 1, подчищающая огрехи предыдущих проходов. Данная сортировка является устойчивой. Реализация на Си:

```
void shell_sort(int * arr, int n)  {
    int gap, i, j, tmp;

    for (gap = n/2; gap > 0; gap /= 2)  {
        for (i = gap; i < n; i++) {
            temp = arr[i];

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            arr[j] = temp;
        }
    }
}
```

15.3 Составить прогу вычисляющую арифметическое выражение из чисел с обратной польской записью

Удобство обратной польской нотации заключается в том, что выражения, представленные в такой форме, очень **легко вычислять**, причём за линейное время. Заведём стек, изначально он пуст. Будем двигаться слева направо по выражению в обратной польской нотации; если текущий элемент — число или переменная, то кладём на вершину стека её значение; если же текущий элемент — операция, то достаём из стека два верхних элемента (или один, если операция унарная), применяем к ним операцию, и результат кладём обратно в стек. В конце концов в стеке останется ровно один элемент - значение выражения.

Очевидно, этот простой алгоритм выполняется за $O(n)$, т.е. порядка длины выражения.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
const int N = 20;

typedef struct _stack_item
{
   struct _stack_item * next;
   struct _stack_item * prev;
   char data;
} stack_item;
typedef struct _stack
{
   int size;
   struct _stack_item * head;
} stack;
void stack_create(stack * s)
{
   s->head = NULL;
   s->size = 0;
}
void stack_push(stack * s, char value)
{
   stack_item * tmp;
```

```
    tmp = malloc(sizeof(stack_item));
    tmp->data = value % 48;//******ВАЖНО******

    if (s->size == 0)

        tmp->prev = NULL;
        s->head = tmp;

    else

        tmp->prev = s->head;
        s->head->next = tmp;
        s->head = tmp;

    s->size++;
}
char stack_pop(stack * s)
{
    char val;
    stack_item * tmp;

  val = s->head->data;
    tmp = s->head->prev;
    free(s->head);
    s->head = tmp;
    s->size--;

    return val;
}
void stack_destroy(stack * s)
{
    while (s->size > 1)

        stack_pop(s);


    free(s->head);
    s->head = NULL;
    s->size = 0;
}
int isNotEmpty(stack * s)
{
    if (s->size == 0)
        return 0;
    else
        return 1;
}

int main()
{
    setlocale(LC_ALL, "rus");
    char s[N];
    scanf("%s",s);
    int i = 0;
    printf("Введенное выражение: ");
    while (s[i] != '\0')

        printf("%c",s[i]);
        i += 1;

    printf("\n");
    i = 0;
    stack st;
    stack_create(&st);
    while (s[i]!='\0')

        if (s[i]>='0' && s[i]<='9')

            stack_push(&st, s[i]);

        if (s[i] == '+' || s[i] == '-' || s[i] == '*' || s[i] == '/')

            char tmp1;
```

```
                char tmp2;
                tmp2 = stack_pop(&st);//читаем ведь с лева на право
                tmp1 = stack_pop(&st);
                if (s[i] == '+')

                    stack_push(&st, tmp1 + tmp2);

                else if (s[i] == '-')

                    stack_push(&st, tmp1 - tmp2);

                else if (s[i] == '*')

                    stack_push(&st, tmp1 * tmp2);

                else if (s[i] == '/')

                    stack_push(&st,tmp1 / tmp2);


            i += 1;

        char result = stack_pop(&st);
        printf("Значение выражения: %d", result);
        return 0;
    }
```

16

16.1 Абстрактные типы данных

АТД - это, по существу, определение некоторого понятия в виде класса (одного или более) объектов с некоторыми свойствами и операциями.

В самой развитой форме в определение АТД входят следующие четыре части:

- внешность (видимая часть, сопряжение, интерфейс), содержащая имя определяемого типа (понятия), имена операций с указанием типов их аргументов и значений и т. п.
- абстрактное описание операций и объектов, с которыми они работают
- конкретное (логическое) описание этих операций на обычном распространенном языке программирования предыдущего поколения
- описание связи между 2 и 3, объясняющее, в каком смысле часть 3 корректно представляет часть 2

16.2Обменные сортировки

В данном разделе мы опишем простой метод сортировки, в котором обмен местами двух элементов производится не с целью вставки или выборки, а непосредственно для упорядочения. Так называемый алгоритм прямого обмена, основывается на сравнении и перестановке пар соседних элементов до тех пор, пока таким образом не будут упорядочены все элементы. Часто эту сортировку называют пузырьковой.

Реализация на Си:

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubble_sort(int *a, int size) {
    int j;
    for (int i = 0; i < size; ++i) {
        for (j = size; j > i; --j) {
            if (a[j] < a[j - 1]) {
                swap(&a[j], &a[j - 1]);
            }
        }
    }
}
```

Если на каждом проходе вести подсчет числа состоявшихся обменов, то можно улучшить пузырьковую сортировку. Еще одно улучшение - не только фиксировать наличие обмена, но и запоминать индекс последнего состоявшегося обмена k. Поскольку все пары соседних элементов выше индекса k уже упорядочены, просмотры можно заканчивать на этом индексе, большем нижнего предела параметра цикла i. Если чередовать направления прохода, то сортировка станет Шейкерной:

```
void shaker_sort(int a[], int n) {
    int p, i;
    for (p = 1; p <= n / 2; p++) {
        for (i = p - 1; i < n - p; i++)
            if (a[i] > a[i+1])
                swap(&a[i], &a[i + 1]);
        for (i = n - p - 1; i >= p; i--)
            if (a[i] < a[i-1])
                swap(&a[i], &a[i - 1]);
    }
}
```

16.3 Написать программу на на Си, которая из заданного дерева удаляет +0 и *1

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "stack.h"

typedef struct _Node
{
        char _varOp;
        double _num;
        struct _Node *_left;
        struct _Node *_right;
        struct _Node *_parent;
} Node;

Node *treeNodeCreate(Node *parent);
Node *treeCopy(Node **node);
void treeBuild(Node **node, Stack *st, Node *parent);
void treeDestroy(Node **node);
void zeroDeleteRight(Node **node);
void zeroDeleteLeft(Node **node);
void oneDeleteRight(Node **node);
void oneDeleteLeft(Node **node);
void PKL(Node **node, const int level);
void PKLprint(Node **node, const int level);


int isLetter(const char ch);
int isNumber(const char ch);
int isOp(const char ch);
int isOpHigh(const char op1, const char op2);
void postOrder(const char *str, Stack *st);

int main(void)
{
        int action;
        char expr[255];
        Node *root = NULL, *root2 = NULL;
        Stack stPost;

        while (1)
        {
                printf("Options:\n");
                printf("1. Enter the expression\n");
                printf("2. Print source expression\n");
                printf("3. Print transformed expression\n");
                printf("4. Print source tree\n");
                printf("5. Print transformed tree\n");
                printf("6. Quit\n");
                printf("Choose option: ");
                scanf("%d", &action);

                switch (action)
                {
                        case 1:
                        {
                                printf("Enter expression: ");
```

```
                                        scanf("%s", expr);

                                        treeDestroy(&root);
                                        treeDestroy(&root2);
                                        stackCreate(&stPost);
                                        postOrder(expr, &stPost);
                                        treeBuild(&root, &stPost, NULL);
                                        stackDestroy(&stPost);

                                        root2 = treeCopy(&root);

                                        PKL(&root2, 0);

                                        break;
                                }

                                case 2:
                                {
                                        printf("Source expression: %s\n", expr);

                                        break;
                                }

                                case 3:
                                {
                                        printf("\n");

                                        break;
                                }

                                case 4:
                                {
                                        if (root != NULL)
                                        {
                                                printf("Source tree\n");
                                                PKLprint(&root, 0);
                                        }
                                        else
                                                printf("Empty source tree\n");

                                        break;
                                }

                                case 5:
                                {
                                        if (root2 != NULL)
                                        {
                                                printf("Transformed tree\n");
                                                PKLprint(&root2, 0);
                                        }
                                        else
                                                printf("Empty transformed tree\n");

                                        break;
                                }

                                case 6: break;

                                default:
                                {
                                        printf("Error. Wrong option\n");

                                        break;
                                }
                        }

                        if (action == 6)
                                break;
                }

                treeDestroy(&root);
                treeDestroy(&root2);

                return 0;
```

```
}

Node* treeNodeCreate(Node *parent)
{
	Node *tmpNode = (Node *)malloc(sizeof(Node));

	tmpNode->_varOp = '\0';
	tmpNode->_num = 0.0;
	tmpNode->_left = NULL;
	tmpNode->_right = NULL;
	tmpNode->_parent = parent;

	return tmpNode;
}

Node *treeCopy(Node **node)
{
	Node *tmpNode = NULL;

	if (*node == NULL)
		return NULL;

	tmpNode = treeNodeCreate(&(*node)->_parent);
	tmpNode->_varOp = (*node)->_varOp;
	tmpNode->_num = (*node)->_num;
	tmpNode->_left = treeCopy(&(*node)->_left);
	tmpNode->_right = treeCopy(&(*node)->_right);

	return tmpNode;
}

void treeBuild(Node **node, Stack *st, Node *parent)
{
	Token token;

	if (stackEmpty(st))
		return;

	token = stackTop(st);

	stackPop(st);

	(*node) = treeNodeCreate(parent);
	(*node)->_varOp = token._varOp;
	(*node)->_num = token._num;

	if (isOp((*node)->_varOp))
	{
		treeBuild(&(*node)->_right, st, node);
		treeBuild(&(*node)->_left, st, node);
	}
}

void treeDestroy(Node **node)
{
	if (*node == NULL)
		return;

	if ((*node)->_left != NULL)
		treeDestroy(&(*node)->_left);

	if ((*node)->_right != NULL)
		treeDestroy(&(*node)->_right);

	free(*node);

	*node = NULL;
}


void zeroDeleteRight(Node **node){
	if (!isOp((*node)->_left->_varOp)){
		(*node)->_num = (*node)->_left->_num;
		(*node)->_varOp = (*node)->_left->_varOp;
```

```
                        (*node)->_left = NULL;
                        (*node)->_right = NULL;
                }
                else {
                        if ((*node) == (*node)->_parent->_right){
                                (*node)->_parent->_right = (*node)->_left;
                                (*node)->_left->_parent = (*node)->_parent;
                                (*node)->_left = NULL;
                                treeDestroy(node);
                        }
                        else if ((*node) == (*node)->_parent->_left){
                                (*node)->_parent->_left = (*node)->_left;
                                (*node)->_left->_parent = (*node)->_parent;
                                (*node)->_left = NULL;
                                treeDestroy(node);
                        }
                }
}

void zeroDeleteLeft(Node **node){
                if (!isOp((*node)->_right->_varOp)){
                        (*node)->_num = (*node)->_right->_num;
                        (*node)->_varOp = (*node)->_right->_varOp;
                        (*node)->_left = NULL;
                        (*node)->_right = NULL;
                }
                else {
                        if ((*node) == (*node)->_parent->_right){
                                (*node)->_parent->_right = (*node)->_right;
                                (*node)->_right->_parent = (*node)->_parent;
                                (*node)->_right = NULL;
                                treeDestroy(node);
                        }
                        else if ((*node) == (*node)->_parent->_left){
                                (*node)->_parent->_left = (*node)->_right;
                                (*node)->_right->_parent = (*node)->_parent;
                                (*node)->_right = NULL;
                                treeDestroy(node);
                        }
                }
}

void oneDeleteRight(Node **node){
                if (!isOp((*node)->_left->_varOp)){
                        (*node)->_num = (*node)->_left->_num;
                        (*node)->_varOp = (*node)->_left->_varOp;
                        (*node)->_left = NULL;
                        (*node)->_right = NULL;
                }
                else {
                        if ((*node) == (*node)->_parent->_right){
                                (*node)->_parent->_right = (*node)->_left;
                                (*node)->_left->_parent = (*node)->_parent;
                                (*node)->_left = NULL;
                                treeDestroy(node);
                        }
                        else if ((*node) == (*node)->_parent->_left){
                                (*node)->_parent->_left = (*node)->_left;
                                (*node)->_left->_parent = (*node)->_parent;
                                (*node)->_left = NULL;
                                treeDestroy(node);
                        }
                }
}

void oneDeleteLeft(Node **node){
                if (!isOp((*node)->_right->_varOp)){
                        (*node)->_num = (*node)->_right->_num;
                        (*node)->_varOp = (*node)->_right->_varOp;
                        (*node)->_left = NULL;
                        (*node)->_right = NULL;
                }
                else {
                        if ((*node) == (*node)->_parent->_right){
```

```
                                (*node)->_parent->_right = (*node)->_right;
                                (*node)->_right->_parent = (*node)->_parent;
                                (*node)->_right = NULL;
                                treeDestroy(node);
                        }
                        else if ((*node) == (*node)->_parent->_left){
                                (*node)->_parent->_left = (*node)->_right;
                                (*node)->_right->_parent = (*node)->_parent;
                                (*node)->_right = NULL;
                                treeDestroy(node);
                        }
                }
}

void PKL(Node **node, const int level)
{
        if (*node == NULL)
                return;

        if ((*node)->_right != NULL)
                PKL(&(*node)->_right, level + 1);

        if ((*node)->_varOp == '\0')
                return;
        if ((*node)->_varOp == '+' && (*node)->_right->_num == 0.0 && (*node)->_right->_varOp == '\0')
                zeroDeleteRight(node);
        if ((*node)->_varOp == '+' && (*node)->_left->_num == 0.0 && (*node)->_left->_varOp == '\0')
                zeroDeleteLeft(node);
        if ((*node)->_varOp == '*' && (*node)->_right->_num == 1.0 && (*node)->_right->_varOp == '\0')
                oneDeleteRight(node);
        if ((*node)->_varOp == '*' && (*node)->_left->_num == 1.0 && (*node)->_left->_varOp == '\0')
                oneDeleteLeft(node);
        if ((*node)->_left != NULL)
                PKL(&(*node)->_left, level + 1);
}

void PKLprint (Node **node, const int level)
{
        if (*node == NULL)
                return;

        if ((*node)->_right != NULL)
                PKLprint(&(*node)->_right, level + 1);

        if ((*node)->_varOp != '\0')
                printf("%*s%c\n", level * 4, "", (*node)->_varOp);
        else
                printf("%*s%f\n", level * 4, "", (*node)->_num);

        if ((*node)->_left != NULL)
                PKLprint(&(*node)->_left, level + 1);
}

int isLetter(const char ch)
{
        return ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'));
}

int isNumber(const char ch)
{
        return (ch >= '0' && ch <= '9');
}

int isOp(const char ch)
{
        return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^' || ch == '(' || ch == ')');
}

int opPrior(const char op)
{
        if (op == '^')
                return 4;

        if (op == '*' || op == '/')
```

```
                return 3;

        if (op == '+' || op == '-')
                return 2;

        return 1;
}

int isOpHigh(const char op1, const char op2)
{
        if (op1 == '(' || op2 == '(' || op2 == ')')
                return 0;

        if (op1 == op2 && op2 == '^')
                return 0;

        return (opPrior(op1) >= opPrior(op2));
}

void postOrder(const char *str, Stack *st)
{
        int i = 0, step = -1, isBracket = 0, isDot = 0;
        char tmpCh;
        Token tk;
        Stack stOp;

        stackCreate(&stOp);

        tk._varOp = '\0';
        tk._num = 0.0;

        while (str[i] != '\0')
        {
                if (str[i] == '.')
                        isDot = 1;
                else if (isLetter(str[i]))
                {
                        tk._varOp = str[i];

                        stackPush(st, tk);
                }
                else if (isNumber(str[i]))
                {
                        tk._varOp = '\0';

                        if (!isDot)
                                tk._num = tk._num * 10.0 + str[i] - '0';
                        else
                        {
                                tk._num = tk._num + pow(10.0, step) * (str[i] - '0');
                                step--;
                        }

                        if (str[i + 1] != '.' && !isNumber(str[i + 1]))
                        {
                                stackPush(st, tk);

                                tk._num = 0.0;
                                step = -1;
                                isDot = 0;
                        }
                }
                else if (isOp(str[i]))
                {
                        tk._varOp = str[i];

                        if (str[i] == ')')
                                isBracket = 1;
                        else if (str[i] == '-' && (i == 0 || str[i - 1] == '('))
                        {
                                tmpCh = tk._varOp;
                                tk._varOp = '\0';
                                tk._num = 0.0;
```

```
                                        stackPush(st, tk);

                                        tk._varOp = tmpCh;
                                }

                                while (!stackEmpty(&stOp) && (isOpHigh(stackTop(&stOp)._varOp, str[i]) || isBracket))
                                {
                                        if (stackTop(&stOp)._varOp == '(')
                                                isBracket = 0;
                                        else
                                                stackPush(st, stackTop(&stOp));

                                        stackPop(&stOp);
                                }

                                if (str[i] != ')')
                                        stackPush(&stOp, tk);
                        }

                        i++;
                }

                while (!stackEmpty(&stOp))
                {
                        stackPush(st, stackTop(&stOp));
                        stackPop(&stOp);
                }

                stackDestroy(&stOp);
}
```

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	
49	
50	
51	
52	
53	
54	
55	
56	
57	
58	
59	
60	
61	

62	
63	
64	