Глава 1

Использование Стека Вызовов

1.1. Проверка скобочных выражений

При решении задач с использованием стека, или при помощи автоматов со стековой памятью, часто можно обойтись без организации стека как самостоятельной структуры данных. В качестве примера рассмотрим две реализации программы проверки правильности скобочного выражения (с несколькими типами скобок).

Для начала рассмотрим *синтаксис* скобочных выражений. Правильное скобочное выражение S подчиняется следующим правилам:

$$\begin{array}{rcl} S & \leftarrow & \emptyset \\ & | & '(' \ S \ ')' \ S \\ & | & '\{' \ S \ '\}' \ S \\ & | & '[' \ S \ ']' \ S \end{array}$$

Этот набор правил назвается *грамматикой* и его следует читать так:

> Правильное скобочное выражение S является либо пустой строкой, либо правильным скобочным выражением, заключенным в парные скобки (различных типов), за которым следует правильное скобочное выражение.

S в данном наборе называется нетерминалом, а все варианты скобок – алфавитом или терминалами – символами, которые не допускают дальнейшего определения.

Обычно, для проверки алгоритму отдается строка, и на выходе алгоритм выдает бинарный ответ — ДА или НЕТ (является ли строка правильным скобочным выражением или нет). Классическим решением данной задачи является организация автомата со стековой памятью, который последовательно считывает символы строки и сохраняет в стековой памяти путь из открытых скобок. Это решение следует из того, что указанный набор правил можно прочитать так:

> Правильное скобочное выражение — это такая последовательность символов, в которой каждая открывающая скобка должна быть закрыта либо сразу, либо после правильного скобочного выражения, а каждая закрывающая скобка должна следовать либо сразу после парной открывающей, либо после правильного скобочного выражения.

1.2. Использование внешнего стека

Рассмотрим классическое решение. Сначала опишем стековые процедуры `push` и `pop`:

1 ⟨*Реализация стека* 1⟩≡
⟨*Глобальные переменные стека* 2a⟩
⟨*Реализация* `push` 2b⟩
⟨*Реализация* `pop` 2c⟩

Для простоты положим стек как массив символов ограниченной, но достаточной длины, подходящей под лимиты задачи (если они есть, конечно):

2a ⟨*Глобальные переменные стека* 2a⟩≡ (1)

```
#define MAX_STACK 1024
char stack[MAX_STACK];
int stack_ptr = 0;
```

Defines:
MAX_STACK, used in chunk 2b.
stack, used in chunk 2.
stack_ptr, used in chunk 2.

Функция push кладет значение на вершину стека и увеличивает указатель stack_ptr на единицу. На всякий случай, проверяем, что мы не переполнили стек:

2b ⟨*Реализация* push 2b⟩≡ (1)

```
int push(char x) {
    if ( stack_ptr >= MAX_STACK)
        return -1;
    stack[stack_ptr++] = x;
    return 0;
}
```

Defines:
push, used in chunk 3b.
Uses MAX_STACK 2a, stack 2a, and stack_ptr 2a.

Функция pop проверяет, что стек не пуст, возвращает значение с вершины стека (через указатель) и уменьшает stack_ptr.

2c ⟨*Реализация* pop 2c⟩≡ (1)

```
int pop(char* x) {
    if ( stack_ptr <= 0 )
        return -1;
    *x = stack[--stack_ptr];
    return 0;
}
```

Defines:
pop, used in chunk 3.
Uses stack 2a and stack_ptr 2a.

Теперь, допустим, у нас уже есть строка `s` для проверки. Просто перемещаемся последовательно по строке и, в зависимости от читаемого символа, если он является открывающей скобкой – кладем (парную ей) на стек, если закрывающей — сверяем с вершиной. При любом несовпадении, либо по досрочном окончании строки или стека — делаем заключение о неправильности скобочного выражения.

3a ⟨*Функция проверки выражения со стеком* 3a⟩≡

```
int check(char* s) {
    int err = 0;
    char x;
    while(*s) {
        switch (*s) {
        ⟨Открывающие скобки - кладем на стек 3b⟩
        ⟨Закрывающие скобки сверяем с вершиной 3c⟩
        default:
            break;
        }
        if ( err ) return -1;
        s++;
    }
    ⟨Проверяем, что стек пуст 3d⟩
    return 0;
}
```

Defines:
check, never used.

В случае, когда читаемый символ `*s` является открывающей скобкой, кладем на стек (с проверкой выхода за пределы) закрывающую пару.

3b ⟨*Открывающие скобки - кладем на стек* 3b⟩≡ (3a)

```
        case '(':
            err = push(')'); break;
        case '{':
            err = push('}'); break;
        case '[':
            err = push(']'); break;
```

Uses push 2b.

Если читаемый символ — закрывающая скобка, сверяем с вершиной (если она доступна). Выходим сразу, если что-то не так.

3c ⟨*Закрывающие скобки сверяем с вершиной* 3c⟩≡ (3a)

```
        case ')': case '}': case ']':
        if ( (err = pop(&x)) || x != *s )
            return -1;
        break;
```

Uses pop 2c.

Все остальные символы мы просто игнорируем, перемещаясь по строке дальше. Как только мы дошли до конца строки:

3d ⟨*Проверяем, что стек пуст* 3d⟩≡ (3a)

```
    if ( !pop(&x) ) return -1;
```

Uses pop 2c.

Данный код, а точнее код фнукций `push` и `pop` будет естественным образом усложняться, если размер стека заранее неизвестен и ограничений тоже никаких нет.

1.3. Использование стека вызовов

Посмотрим, как можно обойтись без использования внешней структуры и методов `push` и `pop`. Оказывается, древовидная структура рекурсивных вызовов фуукций языка Си соответствует древовидному синтаксису наших выражений, а переменные, хранящиеся в стеке вызовов соответствуют стеку открытых скобок. Если в строке встречается открывающая скобка — запоминаем парную ей в локальной переменной нашей функции. Если закрывающая — возвращаемся, а вызывающая сторона проверит совпадение.

4a ⟨*Рекурсивная проверка* 4a⟩≡ (5a)
```
char check_rec(char** s) {
    char wait_for;
    char c = *(*s)++;
    switch(c) {
    ⟨Открывающие скобки - запоминаем 4b⟩
    ⟨Закрывающие скобки - возвращаемся 4c⟩
    default:
        return check_rec(s);
    }
    ⟨Проверяем совпадение 4d⟩
}
```
Defines:
`check_rec`, used in chunks 4d and 5a.

В этой реализации мы используем указатель на указатель `char** s` для передачи между рекурсивными вызовами “потока“ символов. Каждый вызов может модифицировать указатель на символ потока. Чаще всего нам потребуется читать следюущий символ при рекурсивном вызове, поэтому мы сразу перемещаем указатель после чтения в переменную `c`.

4b ⟨*Открывающие скобки - запоминаем* 4b⟩≡ (4a)
```
    case '{': wait_for = '}'; break;
    case '[': wait_for = ']'; break;
    case '(': wait_for = ')'; break;
```

Закрытием считаем не только скобки, но и конец строки (символ `'\0'`). Поскольку фнукция проверки должна возвращать 0 в случае правильности выражения и прочие значения — в противном случае, этот вариант вполне нам подойдет.

4c ⟨*Закрывающие скобки - возвращаемся* 4c⟩≡ (4a)
```
    case '}': case ']': case ')': case '\0': return c;
```

После успешной проверки — двигаемся дальше. Общим результатом проверки будет являться результат проверки выражения, начинающегося со следующего символа (поток `s` уже подготовлен). Если `check_rec` вернул не то, что мы ожидаем — возвращаемся с ненулевым результатом, который будет интерпретирован, как ошибка.

4d ⟨*Проверяем совпадение* 4d⟩≡ (4a)
```
    return check_rec(s) != wait_for ? -1 : check_rec(s);
```
Uses `check_rec` 4a 5b.

Как видим, второе решение более изящно и ближе к оригинальному описанию проверяемого синтаксиса.

Сведем все это воедино, с простой оберткой для тестирования:

5a ⟨*skobkicheck.c* 5a⟩≡
```
#include <stdio.h>
⟨Рекурсивная проверка 4a⟩
int main() {
    char buf[8192];
    char* s = fgets(buf, sizeof(buf), stdin);
    if ( !check_rec(&s) )
        printf("OK\n");
    else
        printf("NOT OK\n");
    return 0;
}
```
Defines:
`main`, never used.
Uses `check_rec` 4a 5b.

Возможно, более наглядным (но не самым лучшим) и приближенным к математическому определению будет вариант, когда мы не будем выносить проверку совпадения за оператор `switch`, а будем проверять и двигаться дальше непосредственно в `case`-е:

5b ⟨*Еще одна реализация* `check_rec` 5b⟩≡
```
char check_rec(char** s) {
    char c = *(*s)++;
    switch (c) {
        case '{': return check_rec(s) != '}' ? -1 : check_rec(s);
        case '[': return check_rec(s) != ']' ? -1 : check_rec(s);
        case '(': return check_rec(s) != ')' ? -1 : check_rec(s);
        case '}': case ']': case ')': case '\0': return c;
        default: return check_rec(s);
    }
}
```
Defines:
`check_rec`, used in chunks 4d and 5a.

Красота!

Задание

1. Дополните решение возможностью анализировать выражение из четырех типов скобочных пар: {}, (), <>, [].

2. Дополните грамматику и анализатор так, чтобы в строке могли встечаться строки, заключенные в двойные кавычки, между которыми текст проверяться не должен. Пример правильного выражения: `(")())"><>""()`. Подсказка: набор правил теперь должен включать еще вспомогательный нетерминал T, обозначающий строку без кавычек внутри:

$$\begin{array}{rcl} T & \leftarrow & \emptyset \\ & | & A\ T \\ S & \leftarrow & \emptyset \\ & | & '"'\ T\ '"'\ S \\ & | & '('\ S\ ')'\ S \\ & | & '\{'\ S\ '\}'\ S \\ & | & '['\ S\ ']'\ S \end{array}$$

где A – произвольный символ, не равный ". Для дополнительного символа нетерминала T нужно будет сделать отдельную анализирующую фукнцию.

Глава 2

Потоковая обработка данных

Достаточно часто встречается задача обработки данных, которые поступают на вход вычислительной системы последовательно. Причём возможности повторного их получения нет, либо она существенно ограничена. В этом случае говорят, что имеют дело с *потоком* данных. Примерами потока могут служить символы, введенные с клавиатуры, сигнал, принимаемый из эфира или микрофона, ровно как и с любых других датчиков, сетевые пакеты и т.п. Обработка потока происходит т.н. *однопроходными алгоритмами*, поскольку они проходят всю последовательность входных данных ровно один раз и второй проход (или произвольный доступ) к данным им не требуется.

Однопроходный алгоритм также может быть применен, конечно, и к файлам с произвольным доступом, однако "однопроходность" подчеркивает относительную простоту модели вычислений.

Результатом работы однопроходного алгоритма может быть также *поток* данных и/или выдача результата вычислений по окончании входного потока.

Обычно выделяют три класса обработчиков: *фильтры* (filter), *отображения* (map) и *редукцию* (reduce), хотя все они являются вариациями *свёртки* (fold). Все потоковые вычисления можно свести к композиции таких обработчиков.

Фильтры принимают на вход поток и возвращают новый поток, но содержащий только элементы удовлетворяющие некоторому условию. Например, фильтр всех четных чисел:

$$f(\{1,2,3,4,5,6,\ldots\}) = \{2,4,6,\ldots\}$$

Отображения возвращают новый поток, который содержит результат применения некоторой функции к каждому элементу входного потока. Например, возведение в квадрат:

$$m(\{1,2,3,4,5,6,\ldots\}) = \{1,4,9,16,25,36,\ldots\}$$

Редукция строит *агрегат* из элементов потока, если задана операция "сложения" $a \oplus b$ на множестве элементов потока, обладающая свойством дистрибутивности. например, суммирует числа:

$$r(\{1,2,3,4,5,6\}) = 31$$

Можно, конечно, заставить редукцию выдавать поток (она готова для каждого входного элемента предоставить результат), но обычно она "ждет" окончания входного потока и выдает агрегат.

Обобщенная свертка похожа на редукцию, но она может накапливать результат (строить агрегат) типа отличного от типа элементов потока. Вот такую, наиболее общую, форму алгоритмов мы и рассмотрим.

2.1. Однопроходные алгоритмы без памяти

Потоком S назовем однородную последовательность данных $S = \{a_0, a_1, \ldots, a_i, \ldots\}$, принадлежащих некоторому типу $a_i \in A$.

Подразумевается, что длина потока заранее неизвестна, хотя конечна.

Считается, что для потока заданы две функции, имеющие семантику *головы* и *хвоста* — h и t. Взятие элемента из потока $h : S \to A$, которая возвращает элемент A, если поток не пуст, и функция перехода к следующему состоянию потока $t : S \to S$. Для полноты описания лучше иметь эти

функции разделенными, хотя, часто оказывается, что h и t совмещены (t вызывается автоматически, а состояние потока S скрыто, как, например, при использовании `stdin` в `getchar` или `scanf`).

Поскольку длина потока неизвестна, и он может закончиться в любой момент, результат нужно готовить сразу, с момента поступления первого элемента. Материал для построения результата называется состоянием Q, которое может меняться после почтения каждого элемента. Сам результат R, который вообще-то может быть менее богат, чем состояние, должен получаться из него посредством функции $g : Q \to R$.

Рассмотрим типичный обработчик потока. Чаще всего он имеет форму $p : S \times Q \to R$. На входе мы имеем поток S (для которого заданы функции h и t), и, важно заметить, какое-то состояние Q. Само тело обработчика выражается в виде функции $f : Q \times A \to Q$, которая, собственно, и занимается модификацией состояния в зависимости от обрабатываемого элемента.

$$p(s,q) = \begin{cases} s = \emptyset : & g(q) \\ \text{иначе} : & p(t(s), f(q, h(s))) \end{cases} \tag{2.1}$$

Т.е. в случае, если поток пуст (h не может больше взять элемент из потока), вычисляем результат исходя из накопленного состояния q. Иначе, если элемент в потоке существует, то производим вычисление и рекурсивно продолжаем обработку потока. Запуск обработчика подразумевает, помимо потока, также и начальное состояние. Назовем его q_0. В каких-то случаях оно может оказаться тривиальным, а где-то его придется еще специальным образом "готовить".

Также существует другой вариант обработчика:

$$\begin{aligned} p(s,q) &= g(p'(s,q)) \\ p'(s,q) &= \begin{cases} s = \emptyset : & q \\ \text{иначе} : & f(p'(t(s),q), h(s)) \end{cases} \end{aligned} \tag{2.2}$$

Здесь нам пришлось ввести вспомогательную функцию p', которая, собственно, делает вычисления, а сама p является *оберткой*.

Дело в том, что находясь в середине потока, можно смотреть на него в двух направлениях. С одной стороны, можно считать себя первым и смотреть на оставшийся "хвост" потока (2.1). Такой обработчик называется *право-рекурсивным* или *хвостовым*. С другой стороны, можно смотреть в начало и считать, что на нас обработка потока должна закончиться, и основная работа уже проделана. В этом случае обработка (2.2) называется *лево-рекурсивной*.

Заметьте, что в обоих случаях последовательность чтения элементов из потока одинакова, и левая рекурсия не означает, что поток читается с конца.

Многие потоки не имеют семантики "хвоста". Из них можно только брать элементы, а хвост сдвигается сам. Т.е. для потока определена только функция $n : S \to A$ (next, следующий), а факт окончания потока как-то должен отражаться в возвращаемом значении. Например, можно дополнить множество A пустым значением $\emptyset$ и считать его концом потока. В этом случае обработчик выглядит чуть проще:

$$p(s,q) = \begin{cases} n(s) = \emptyset : & g(q) \\ \text{иначе} : & p(s, f(q, n(s))) \end{cases} \tag{2.3}$$

однако, теряется т.н. *ссылочная прозрачность*, когда вызов функции $n(s)$ можно заменить ее возвращаемым значением, посколько она меняет *внутреннее состояние* потока.

Алгоритм называется *однопроходным без памяти*, если он имеет хвостовую форму и размер состояния $\|Q\| = O(1)$ – является константой и не зависит количества прочитанных элементов. Хвостовая форма в данном определении важна, поскольку левая рекурсия подразумевает, что функция f "ждет" результата рекурсивного вызова p', а для этого потребуется стековая память (вспомните функцию переворота потока символов).

Например, простой подсчет суммы потока действительных чисел можно разобрать на следующие "запчасти":

- В качестве функции чтения элемента из потока n задействуем фукнцию `scanf`. При работе со стандартным потоком ввода `stdin` сам поток таскать с собой не надо, функиця `scanf` и так о нем знает, поэтому его можно из нашей конструкции иключить.

- Состояние Q – это действительное число, которое, собственно, и является суммой.
- Функция $f(q, a) = q + a$ – простая сумма двух чисел.
- Функция $g(q)$ – пусть печатает результат. Будем использовать для этого `printf`.

Давайте оформим это в виде программы на Си.

10a ⟨*обработчик-суммирование* 10a⟩≡ (10b)

```
int sum(double q) {
    double a;
    if ( scanf("%lf", &a) == 1 )
        return sum(a+q);
    return printf("%lf\n", q);
}
```

Defines:
`sum`, used in chunks 10 and 11b.

Заметили, где здесь прячутся функции n, p, f, g? Поскольку `scanf` возвращает количество прочитанных шаблонов, можно использовать ее для проверки окончания потока (если `scanf` не может прочитать число двойной точности, то вернет 0, а если поток `stdin` закончится (`EOF`), то -1). Сами прочитанные значения она записывает по переданным адресам. В нашем случае, пришлось выделить переменную `double a` для того, чтобы хранить результат чтения из потока.

Целиком программа выглядит так:

10b ⟨*sum.c* 10b⟩≡

```
#include <stdio.h>
⟨обработчик-суммирование 10a⟩

int main() {
    return sum(0);
}
```

Defines:
`main`, never used.
Uses `sum` 10a.

Начальное состояние суммирования, в нашем случае, — это 0.

Попробуем что-нибудь посложенее. Давайте посчитаем минимум, максимум и среднее от введенных чисел. Сначала разберем, что входит в состояние Q. Очевидно, это композиция из текущих минимума, максимума, а также пары – суммы и количества прочитанных чисел, из которых g потом посчитает среднее. Однако, если поток совершенно пустой, мы не сможем посчитать ничего. Т.е. g в случае, если количество прочитанных чисел равно нулю, должна напечатать ошибку. Состояние Q можно формить в виде структуры `struct`, а можно передавать его между вызовами в виде нескольких аргументов:

10c ⟨*подсчет статистики* 10c⟩≡ (11a)

```
int stat(double min, double max, double sum, int n) {
    double a;
    if ( scanf("%lf", &a) == 1 )
        return stat(a < min ? a : min,
                    a > max ? a : max,
                    sum + a,
                    n + 1);
    if ( n == 0 )
        return printf("Error: Empty stream!\n");
    else
        return printf("min=%lf, max=%lf, avg=%lf\n", min, max, sum/n);
}
```

Defines:
`stat`, used in chunk 11a.
Uses `sum` 10a.

Остается вопрос, что считать начальным состоянием q_0? Если с суммой и количеством все понятно, что в качестве начального минимума можно предложить аналог $+\inf$, а в качестве начального максимума $-\ -\inf$. Поскольку любое другое число будет, соответственно, меньше начального минимума и больше начального максимума. К сожалению, таких стандартных констант для типа `double` нет, хотя тип `double` включает в себя эти значения и они могут появляться в процессе вычислений. Но, можно воспользоваться свойством функции стандартной библиотеки `atof`, которая преобразовывает строку в число с плавающей точкой двойной точности, но может также принимать значения `"inf"` и `"-inf"`, возвращая нам соответствующие ”бесконечности”.

11a ⟨*stat.c* 11a⟩≡

```
#include<stdio.h>
// нужно для atof
#include<stdlib.h>

⟨подсчет статистики 10c⟩

int main() {
    return stat(atof("inf"), atof("-inf"), 0, 0);
}
```

Defines:
`main`, never used.
Uses `stat` 10c.

Другим способом избежать проблем с поиском подходящего q_0 для минимума и максимума является является его неявная ивалидация. Т.е. не будем полагаться на q_0 вообще, а начнем считать минимум/максимум с первого элемента потока:

11b ⟨*подсчет статистики с невалидным начальным состоянием* 11b⟩≡ (11c)

```
int stat2(double min, double max, double sum, int n) {
    double a;
    if ( scanf("%lf", &a) == 1 )
        return stat2( (!n || a < min)? a : min,
                      (!n || a > max)? a : max,
                      sum + a,
                      n + 1);
    if ( n == 0 )
        return printf("Error: Empty stream!\n");
    else
        return printf("min=%lf, max=%lf, avg=%lf\n", min, max, sum/n);
}
```

Defines:
`stat2`, used in chunk 11c.
Uses `sum` 10a.

Соответственно, вся программа может выглядеть так:

11c ⟨*stat2.c* 11c⟩≡

```
#include <stdio.h>

⟨подсчет статистики с невалидным начальным состоянием 11b⟩

int main() {
    return stat2(-100313, 231987, 0, 0);
}
```

Defines:
`main`, never used.
Uses `stat2` 11b.

Здесь намеренно указаны произвольные начальные значения для минимума и максимума, чтобы показать, что они ни на что не влияют.

Задание

1. Пусть тип элементов потока – это тройка действительных чисел $< x, y, m >$ обозначающих координату и массу точечного тела. Найти центр масс и общую массу точек.

2. Пусть тип элементов потока – это множество измерений пары значений напряжения и тока через резистор $< U, I >$. Найдите сопротивление резистора и точность (средне-квадратичное отклонение).

2.2. Однопроходные алгоритмы с памятью

Но, бывают задачи, когда $O(1)$ для Q явно недостаточно. Рассмотрим, например, подсчет значения медианы множества введенных чисел. Медиана – это такое число из множества, которое делит данное множество на 2 равные (приблизительно) части – числа, которые больше медианы и числа, которые меньше (или равны) медиане. Очевидно, что расчет требует хранения всех чисел (или, хотя бы, их распределения), а на это уже потребуется $O(n)$ памяти. Если мы на каждом этапе чтения потока будем хранить отсортированный список уже введенных значений, то медианой будет элемент из середины этого списка. Поддерживать отсортированный список достаточно просто – нужно вставлять новый элемент в положенное ему порядком место. Вставка проще всего производится если список односвязный.

Односвязный список – это чуть более сложная структура чем массив, является динамической, но позволяет быструю вставку элемента в начало списка, а также предоставляет возможности "неограниченного" роста. Т.е. список, в отличие от массива, не занимает памяти, если в нем нет элементов и не требует априорных знаний о возможном размере.

Каждый элемент списка представляется в виде структуры:

12 ⟨*определение списка* 12⟩≡ (14b)

```
typedef struct _list {
    double e;
    struct _list* next;
} List;
```

Defines:
List, used in chunks 13, 14, 38–41, and 44–48.

которая содержит в себе сам элемент (поле `e`) и указатель на следующий элемент списка (поле `next`). Если элемент является последним, то указатель `next` принимает специальное значение `NULL`, т.е. *нулевой указатель*. Заметьте, что односвязный список является рекурсивной структурой.

Идея поддержания отсортированного списка очень проста. Допустим, у нас есть указатель на голову списка `lst` и нам нужно вставить в список число e. Если список пуст (`lst == NULL`), либо когда в голове списка содержится элемент больший или равный e (`lst->e >= e`), мы создаем новый элемент `new` и заполняем его поле `new->e = e`. Новый элемент `new` теперь должен стать новой головой, а старый список (если он был) необходимо прицепить ему в качестве хвоста: `new->next = lst`. Т.е. в хвосте списка содержатся элементы со значением $\geq e$ Если же в голове списка элемент меньше вставляемого, то нужно подыскать ему место в хвосте списка. Для этого нужно вызвать процедуру вставки еще раз, но уже для хвоста. Поскольку у хвоста может "отрасти" новая голова, то процедура вставки всегда должна возвращать голову нового списка.

13a ⟨*вставка элемента в отсортированный список* 13a⟩≡ (14b)

```
List* insert_sort(List* lst, double e) {
    if ( lst == NULL || lst->e >= e ) {
        List* new = malloc(sizeof(List));
        new->e = e;
        new->next = lst;
        return new;
    } else {
        lst->next = insert_sort(lst->next, e);
        return lst;
    }
}
```

Defines:
`insert_sort`, used in chunk 14b.
Uses `List` 12 38a 44a.

Интересной выглядит задача поиска середины списка для вычисления медианы. Можно заняться подсчетом общего числа элементов в списке, потом поделить его пополам и пробежаться от начала списка, отсчитывая половину от общего числа, по достижении – вернуть полученный элемент. Но можно поступить проще, так сказать, однопроходно. Если ввести два указателя и первый, назовем его `slow` каждый раз смещать на один элемент, а второй – `fast` на два элемента списка вперед, то к тому моменту, когда `fast` доберется до конца списка, `slow` как раз окажется посередине:

13b ⟨*поиск середины* 13b⟩≡ (14b)

```
List* median(List* slow, List* fast) {
    if ( !fast || !fast->next)
        return slow;
    return median(slow->next, fast->next->next);
}
```

Defines:
`median`, used in chunk 14b.
Uses `List` 12 38a 44a.

Ну и, поскольку, память у нас под список выделяется динамическая, неплохо бы ее в конце всех вычислений очистить:

14a ⟨*очистка списка* 14a⟩≡ (14b)

```
void list_free(List* lst) {
    if (lst) {
        List* next = lst->next;
        free(lst);
        list_free(next);
    }
}
```

Defines:
`list_free`, used in chunk 14b.
Uses `List` 12 38a 44a.

Давайте теперь соединим все запчасти в программу:

14b ⟨*stat3.c* 14b⟩≡

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

⟨определение списка 12⟩
⟨вставка элемента в отсортированный список 13a⟩
⟨поиск середины 13b⟩
⟨очистка списка 14a⟩

int stat3(List* lst) {
    double a;
    if ( scanf("%ld", &a) == 1 )
        return stat3(insert_sort(lst, a));
    if ( ! lst )
        printf("Error: list is empty!\n");
    else
        printf("median: %lf\n", median(lst, lst)->e)
    list_free(lst) // освобождаем память
    return 0;
}

int main() {
    return stat3(NULL);
}
```

Defines:
`main`, never used.
`stat3`, never used.
Uses `insert_sort` 13a, `List` 12 38a 44a, `list_free` 14a, and `median` 13b.

Задание

1. Немного оптимизируйте список, сохраняя *распределение* элементов. Т.е. в структуру добавьте поле с количеством – если элемент в потоке встречался n раз. Понятно, что поиск медианы будет уже не таким красивым, а в состоянии придется "тащить" общее количество прочитанных элементов.

2. Напишите "фильтр" уникальных элементов потока. Программа должна печатать число из потока, если оно встретилось в первый раз. Придется хранить список всех встреченных чисел и каждый раз проверять, не находится ли следующий элемент из потока уже в данном списке. Если число уже есть в списке – читать следущий элемент и ничего не печатать.

Глава 3

Численные методы

Немного потренируемся писать алгоритмы численных методов решения двух видов задач, которые часто возникают на практике. Поиск корня нелинейного уравнения и поиск минимума нелинейной функции.

Сразу оговорюсь, что данная глава не претендует на полноту математического описания данных методов и даже не на обзор. Если с поиском корней (нулей) нелинейной функции более-менее все понятно, то поиск минимума – это практически творческая задача.

3.1. Поиск корня нелинейного уравнения

Существует несколько распространенных способов решения уравнения вида $f(x) = 0$. Все они *итерационные*, т.е. получают значение x в результате повторяющихся итераций, на каждой итерации x_n постепенно приближается к искомому значению. Это означает, что за конечное время можно получить только приближенный результат. Поэтому количество итераций определяется необходимой точностью ответа. Методы различаются требованиями к функции f и *сходимостью* — скоростью, с которой увеличивается точность на каждой итерации.

Но, так или иначе, **все методы** требуют аналитической подготовки к решению. В любом случае, требуется выбора окрестности δ и доказательства, что эта окрестность содержит корень, а также, что функция достаточно хорошо себя ведет в этой окрестности, возможно даже потребуется аналитическое вычисление производной функции $f'(x)$.

Например, известный метод Ньютона (касательных) обладает самой лучшей сходимостью. Однако, налагает серьезные ограничения на вид функции и ее производные первого и второго порядка (не говоря уже об их существовании) внутри окрестности корня. И в некоторых случах, например, когда производная в корне обращается в 0 (кратный корень), скорость теряется.

Другие методы хуже в плане сходимости, но уже менее требовательны к виду функции. Самым простым и "робастным" является метод поиска корня делением пополам – *дихотомии*. Для его успешной работы только требуется, чтобы в начальной окрестности поиска — некотором интервале $[a, b]$ — производная $f'(x)$ не меняла бы знак, а на концах этого интервала знак функции был бы разным $f(a) \cdot f(b) < 0$.

Вот его мы и рассмотрим.

Итак, нам для вычисления корня потребуется тело функции f, начальный интервал $[a, b]$ и предельная точнось ϵ. Максимальное количество итераций можно прикинуть сразу – это $log_2|b - a| - log_2\epsilon$, поскольку идея метода состоит в том, чтобы сокращать область поиска каждую итерацию в 2 раза, а минимальным размером области является ϵ.

Для того чтобы понять, где может быть корень, нам потребуется значение фукнции f в трех точках: $f(a)$, $f(b)$ и $f(c)$, где $c = (a + b)/2$ – центр интервала $[a, b]$. Обозначим `fa`, `fb` и `fc` – соответствующие значения фукнции Тогда суть метода можно выразить следующими строками:

17a ⟨*суть метода* 17a⟩≡ (17b)
```
if ( b - a < epsilon ) return c;
if ( fa * fc < 0 ) ⟨ищем корень в левой половине 17c⟩
else if ( fc * fb <  0) ⟨ищем корень в правой половине 17d⟩
else if ( fc == 0) return c;
else if ( fa == 0) return a;
else if ( fb == 0) return b;
else ⟨корень недостижим 17e⟩
```

Вариант `конень недостижим` возникает, когда знаки `fa`, `fb` и `fc` совпадают. В этом случае мы перестаем "бороться" за корень и просим у пользователя уточнить область поиска.

Сама функция поиска (алгоритм) будет оформлена в виде рекурсивной функции `sol`. Она должна зависеть от функции `f`, которая, в свою очередь, принимает на вход `double` и возвращает `double`: `double f(double)`, начала и конца области поиска `double a, b`, и точности `epsilon`.

Рекурсия, собственно, будет происходить в ветках `ищем корень в левой половине` и `ищем корень в правой половине`. Но стоит заметить, что в рекурсивном вызове из трех вычисленных значений `fa`, `fb` и `fc`, по крайней мере, два потребутся точно. Поэтому имеет смысл их не вычислять заново, а передать в виде параметров в функцию. Т.е. добавим еще аргументы `double fa, fb`:

17b ⟨*функция поиска* 17b⟩≡ (18b)
```
double sol(double f(double), double a, double b, double epsilon,
           double fa, double fb) {
    /* вычисляем только c и fc  остальное дано "свыше" */
    double c = (a+b)/2;
    double fc = f(c);

    ⟨суть метода 17a⟩
}
```
Defines:
`sol`, used in chunks 17 and 18b.

Теперь распишем ветки с рекурсивным вызовом для поиска в интервалах $[a, c]$ и $[c, b]$. Они выглядят достаточно просто:

17c ⟨*ищем корень в левой половине* 17c⟩≡ (17a)
```
return sol(f, a, c, epsilon, fa, fc);
```
Uses `sol` 17b.

17d ⟨*ищем корень в правой половине* 17d⟩≡ (17a)
```
return sol(f, c, b, epsilon, fc, fb);
```
Uses `sol` 17b.

Тут мы не забываем передать уже вычисленные значения f.

Чтобы указать вызывающей стороне, что корень не найден (точнее, это мы подозреваем, что корня может не быть, хотя ему никто не запрещает находиться в интервале), будем возвращать специальное значение `nan`.

17e ⟨*корень недостижим* 17e⟩≡ (17a)
```
return atof("nan");
```

Теперь попробуем применить наш алгоритм к какому-нибудь уравнению, которое не имеет алгебраических решений (например, с участием тансцендентных функций):

$$e^x - 5x = 0$$

Оформим его в виде функции:

18a ⟨*исследуемая функция* 18a⟩≡ (18b)

```
double func(double x) {
    return exp(x) - 5*x;
}
```

Defines:
`func`, used in chunk 18b.

Напишем программу-обертку, которая будет принимать в качестве аргументов диапазон поиска и точность:

18b ⟨*sol.c* 18b⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

⟨исследуемая функция 18a⟩

⟨функция поиска 17b⟩

int main(int argc, char* argv[]) {
    double a = atof(argv[1]);
    double b = atof(argv[2]);
    double epsilon = atof(argv[3]);
    printf("Result = %.20lf\n", sol(func, a, b, epsilon, func(a), func(b)));

}
```

Defines:
`main`, never used.
Uses `func` 18a and `sol` 17b.

Для линковки программы нам потребуется математическая библиотека `libm`, в которой даны определения функций `exp`, `sin`, `cos` и многих других. Они не входят в стандартную библиотеку `libc`, их придется подключать в явном виде Линковка с математической библиотекой выглядит вот так:

```
gcc -o sol -O3 -lm sol.c
```

Здесь опция `-lm` добавляет тело нужной нам библиотеки. Опция `-O3` относится не к линковке, а к компиляции и указывает компилятору строить оптимизированный код по скорости и по размеру.

Как раньше говорилось, поиск корня всегда требует дополнительной аналитической работы по определению начальной области поиска. Для этого нужно себе хорошо представлять, как ведет себя функция $f(x)$. Например, если мы рассматриваем $f(x) = e^x - 5x$, то при $x \to \pm\infty$ значение $f(x) \to +\infty$. Таже можно легко посчитать, что $f(x)$ достигает минимума в точке $\log 5$, а само значение минимума – $5 - 5\log 5 < 0$. Следовательно, функция имеет 2 корня слева и справа от $x = \log 5 \approx 1.6$.

Попробуем поискать первый корень с точностью до 10^{-6}:

```
sol 0 1.6 1e-6
```

Получаем вот такой результат:

```
Result = 0.25917091369628908470
```

Задание

1. Добавьте в функцию поиска `sol` подсчет числа шагов, требующихся для достижения нужной точности. Сверьте с теоретическими оценками.

2. Реализуйте функцию поиска методом Ньютона `solnewton`: $x_{n+1} = x_n - f(x_n)/f'(x_n)$. Для этого добавьте реализацю функции $f'(x)$ и передайте ее в качестве дополнительного параметра в функцию `solnewton`. При этом, в отличие от `sol`, можно избавиться от параметров `b`, `fa` и `fb`, поскольку данному методу достаточно одной начальной точки приближения. Сравните количество шагов, требующихся для достижения нужной точности функциям `sol` и `solnewton` на одинаковых $f(x)$.

3.2. Поиск минимума

Обобщить методы поиска корня на уравнения с несколькими переменными практически невозможно, поскольку в многомерном случае "корнем", вообще говоря, может являться целая поверхность.

Но вот минимум (экстремум) функции $f(x)$ вполне можно (численно) искать в случае, если x – вектор. Если же функция однопараметрическая и при том дифференцируема аналитически, то поиск минимума можно свести к поиску корня $f'(x) = 0$.

Одним из простых (но не самых эффективных) способов поиска экстремума является метод *градиентного спуска*. Идея метода проста. Если мы стоим на склоне горы, то чтобы попасть в долину, скорее всего, нужно двигаться в сторону, противоположную направлению *градиента*. Градиентом функции $\nabla f(\mathbf{x})$ в точке $\mathbf{x}$ называется вектор, составленный из частных производных функции по компонентам вектора:

$$\nabla f(\mathbf{x}) = \left\{ \frac{\partial}{\partial x_1} f, \frac{\partial}{\partial x_2} f, \dots \right\}$$

Этот вектор лежит в (гипер)плоскости касательной к функции f в точке x. И направление этого вектора соответствует направлению наибольшего роста функции в данной точке.

Признаком окончания движения может служить тот факт, что градиент обнуляется. Т.е. в точке, по крайней мере локального минимума $\nabla f(\mathbf{x}) = 0$.

Понятно, что как и в случае с корнями, данный метод может привести только к одному (локальному) минимуму, в долине которого находится наше начальное приближение. А поиск абсолютного (глобального) минимума (экстремума) в общем случае — задача вообще нерешаемая. Обычно ищется несколько минимумов с множеством случайно (равномерно, на сколько это возможно) разбросанных начальных приближений и из них выбирается самый минимальный. Но чем больше пространство поиска, тем меньше вероятность, что найдется "глобальный" минимум.

Итак, направление движения приблизительно определено, остается вопрос: на сколько шагать? С одной стороны, шагать нужно быстро, чтобы скорее добраться до минимума. С другой стороны, если шагать далеко, то можно "проскочить" минимум. Поэтому выбирается достаточно маленький коэффициент α, а координаты следующего приближения вычисляются как

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \nabla f(\mathbf{x}_n)$$

Значение α можно подобрать эмпирически в качестве константы, а можно вычислять на каждом шаге. Стоит заметить, что размерность $[\alpha]$ совпадает с размерностью обратной второй производной $(\partial^2 f/\partial x^2)^{-1}$, Поэтому было бы неплохо заручиться оценкой предельных значений *Гессиана* функции f, которые могут достигаться в точках экстремума. И чем они выше, тем острее пики и впадины и тем меньше должны быть α, чтобы обеспечить сходимость. Иначе, мы рискуем постоянно проскакивать минимумы.

Методов вычисления значений α существует множество, однако ни один из них не лишен "произвола" в выборе параметров и так или иначе связан с предположительными знаниями о виде $f(\mathbf{x})$.

Для простоты остановимся пока на постоянном коэффициенте спуска $\alpha = const$

Давайте пока представим итерацию метода градиентного спуска в виде функции `grad`. Для вычисления итерации нам потребуется вектор `x`. Если мы хотим иметь дело с произвольной размерностью, то придется ее передавать, скажем, в параметре `nx`. Также нам будут нужны параметры нашего

”произвола” `alpha`, `epsilon` – предельная точность, `delta` – размер области, на которой будет вычисляться градиент, и, собственно, сама функция `f`, только она уже будет принимать на вход вектор (массив `double*` и размерность `int`)

20a ⟨*градиентный спуск* 20a⟩≡ (23)
```
int grad(double *x, int nx, double alpha, double delta,
        double epsilon, double f(double*, int)){
    ⟨локальные переменные 20b⟩
    ⟨вычисляем градиент в точке x 20d⟩
    ⟨проверка условий останова 20e⟩
    ⟨вычисляем следующее приближение 20f⟩
    printf("x[0]=%lf, x[1]=%lf, error=%lf\n", x[0], x[1], f(x, nx));
    return grad(x, nx, alpha, delta, epsilon, f);
}
```
Defines:
`grad`, used in chunk 23.

Перед тем, как вычислить градиент, нам потребуется вычислить значение $f(x)$.

20b ⟨*локальные переменные* 20b⟩≡ (20a) 20c ▷
```
    double fx = f(x, nx);
```

Сам градиент нам нужно где-то запомнить, для этого аллоцируем массив (динамически), а также будем считать квадрат модуля градиента:

20c ⟨*локальные переменные* 20b⟩+≡ (20a) ◁ 20b
```
    double *g = calloc(sizeof(double), nx);
    double g_module = 0;
```

20d ⟨*вычисляем градиент в точке x* 20d⟩≡ (20a)
```
    int i;
    for (i = 0; i < nx; i++) {
        x[i] += delta;
        g[i] = (f(x, nx) - fx)/delta;
        x[i] -= delta;
        g_module += g[i]*g[i];
    }
```

Понятно, что мы вычисляем не градиент, а *приближение градиента* и точность его зависит от выбора параметра `delta`. Мы переиспользуем компоненты вектора `x` для выбора шагов в направлении каждой из коодинат.

20e ⟨*проверка условий останова* 20e⟩≡ (20a)
```
    if (g_module < epsilon*epsilon) {
        free(g);
        return 0;
    }
```

Если модуль градиента достигает некоторого предельного значения, то останавливаемся. Можно остановиться и по другим критериям, например, когда $\alpha\|\nabla f\| < \epsilon$.

20f ⟨*вычисляем следующее приближение* 20f⟩≡ (20a)
```
    for (i = 0; i < nx; i++) {
        x[i] -= alpha * g[i];
    }
    free(g);
```

Отлично. Приступаем к самому интересному. Предположим, что мы ставим некоторый эксперимент и хотим подобрать параметры модели, описывающей какое-то явление. В распоряжении у нас есть параметризованная модель $= F(\vec{\theta}, \mathbf{x})$ которая предсказывает скалярное значение y от некоторого вектора значений наблюдаемых параметров $\mathbf{x}$ и (ненаблюдаемых) параметров модели $\vec{\theta}$. А также некоторое количество измерений (естественно, с погрешностью):

$$S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$$

Задача подбора параметров модели, наилучшим образом описывающей зависимость y_i от $\mathbf{x}_i$ обычно сводится к минимизации ошибки ε, которая определяется как

$$\varepsilon_{\vec{\theta}} = \frac{1}{N} \sum_i (F(\vec{\theta}, \mathbf{x}_i) - y_i)^2$$

Это средний квадрат отклонения значений, полученных моделью, от реально измеряемых.

Собственно, набор оптимальных параметров $\vec{\theta} : \varepsilon \to \min$ мы и будем искать методом градиентного спуска.

Для этого нам потребуются результаты `N` измерений наблюдаемых параметров x_i и величины y. Их можно хранить в виде нескольких массивов: `double x1[N]`, `double x2[N]`, ... `double y[N]`, а можно сделать и массив структур. Для простоты измерения будут храниться глобально.

Нужна будет функция вычисления ошибки `mse` (от Mean Squared Error – квадрат стандартной ошибки), которая будет принимать на вход массив параметров модели `double* p` их количество `int np`

21a ⟨*функция вычисления ошибки* 21a⟩≡ (23)

```
double mse(double* p, int np) {
    double e = 0;
    int i;
    for ( i = 0; i < N; i++ ) {
        double d;
        ⟨вычислить отклонение d 22b⟩
        e += d*d;
    }
    return e/N;
}
```

Defines:
`mse`, used in chunk 23.
Uses `N` 22a.

Сама модель нам потребуется исключительно здесь и больше нигде, поэтому ее можно не передавать в качестве аргумента и реализовать на месте, прямо внутри `mse`, либо оформить в виде глобальной функции.

Функцию `mse`, в свою очередь, мы будем передавать в качестве аргумента `f` нашей универсальной функции `grad`, которая будет заниматься подбором параметров.

Теперь пришло время определиться с моделью. Допустим, мы измеряем ток I, протекающий через какой-то элемент цепи, в зависимости от напряжения U. В результате измерений мы получили множество пар точек (I_i, U_i). Занесем их в нашу структуру:

21b ⟨*структура измерения* 21b⟩≡ (23)

```
typedef struct {
    double I;
    double U;
} Measure;
```

Defines:
`Measure`, used in chunk 22a.

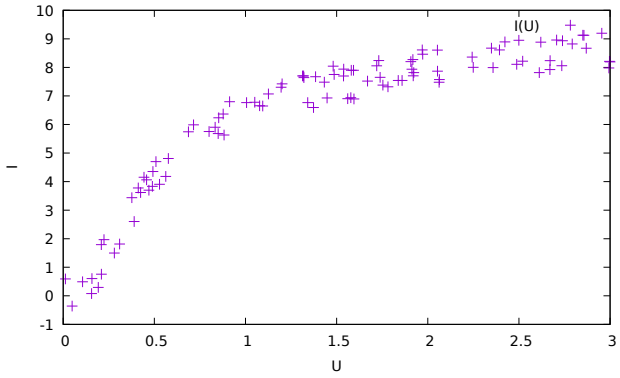
Аллоцируем массив для наших измерений:

22a ⟨*массив измерений* 22a⟩≡ (23)

```
#define MAX_MEASURES 100
int N;
Measure measures[MAX_MEASURES];
```

Defines:
`MAX_MEASURES`, used in chunk 23.
`measures`, used in chunks 22b and 23.
`N`, used in chunks 21a and 23.
Uses `Measure` 21b.

Значения измерений на плоскости ложатся как-то так:



На графике видно, что ток растет нелинейно с увеличением u, и при некоторых значениях u наступает ”насыщение”. Попробуем апроксимировать данную зависимость моделью $I(u) = I_0(1 - e^{-u/U_0})$ (если она, конечно, имеет физический смысл). Эта модель двухпараметрическая. Здесь параметр I_0 определяет ”предельный” ток, а параметр U_0 – начало наступления режима насыщения. Добавим вычисления модели в функцию вычисления ошибки для i-го измерения:

22b ⟨*вычислить отклонение d* 22b⟩≡ (21a)

```
//d = p[0]*(1 - exp(-measures[i].U/p[1])) - measures[i].I;
d = p[0]*exp(-p[1]/measures[i].U) - measures[i].I;
```

Uses `measures` 22a.

Здесь мы используем параметры `p[0]` и `p[1]` из массива для описания модели.

Напишем программу-обертку для ввода значений измерений и начальных параметров модели, с которых мы начнем спуск, а также параметры градиентного спуска α и ϵ. Параметры модели I_0 и U_0 мы будем передавать в качестве аргументов, а экспериментальные данные будем подавать на `stdin` в виде множества пар действительных значений, разделенных пробелом.

23 ⟨*grad.c* 23⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

⟨структура измерения 21b⟩
⟨массив измерений 22a⟩

⟨функция вычисления ошибки 21a⟩

⟨градиентный спуск 20a⟩

int main(int argc, char* argv[]) {
    double alpha, epsilon;
    double p[2]; /* параметры модели */
    p[0] = atof(argv[1]);
    p[1] = atof(argv[2]);
    alpha = atof(argv[3]);
    epsilon = atof(argv[4]);
    /* Читаем измерения */
    N = 0;
    while (N < MAX_MEASURES
        && scanf("%lf %lf", &measures[N].U, &measures[N].I) == 2) N++;
    /* Запускаем градиентный спуск */
    grad(p, 2, alpha, epsilon, epsilon, mse);
    /* печатаем результат и финальную ошибку */
    printf("Params: %0.10lf, %0.10lf, error=%lf\n", p[0], p[1], mse(p, 2));

}
```

Defines:
`main`, never used.
Uses `grad` 20a, `MAX_MEASURES` 22a, `measures` 22a, `mse` 21a, and `N` 22a.

Компилировать данную программу нужно также с опцией `-O3`:

```
gcc -o grad -O3 grad.c -lm
```

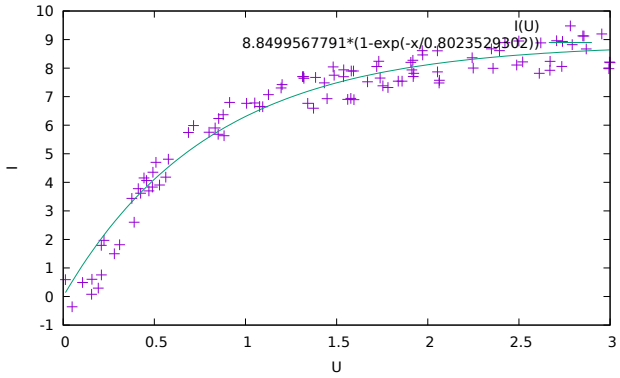
Попробуем запустить ее с приблизительным начальными параметрами (в файле `gen.dat` находятся наши измерения).

```
./grad 0.5 0.5 0.00001 0.000001 <gen.dat
```

В результате получаем:

```
Params: 8.8499567794, 0.8023529303, error=0.274655
```

Смотрим на график:



Задание

- Попробуйте различные данные и различные модели. Оцените количество шагов до получения нужной точности, всегда ли она достигается? Как на сходимость влияют параметры α, δ, ϵ?
- Добавьте в алгоритм `grad` "отжиг" – постепенное уменьшение α с каждой итерацией. Введите параметр отжига m: $\alpha_{n+1} = m\alpha_n$ и проверьте сходимость с ним.

Глава 4

Двоичные деревья поиска

Рассмотрим новую структуру данных. *Двоичное Дерево Поиска.* Данная структура позволяет хранить (упорядоченное) множество элементов со временем доступа на запись, поиск, удаление *в среднем* $O(\log N)$.

Каждый узел дерева N представляет собой либо пустоту $\emptyset$, либо тройку $\langle L\ x\ R\rangle$, где x — значение узла, L и R — левое и правое поддеревья соответственно.

Высота дерева. Кажый узел дерева характеризуется своей высотой h:

$$\begin{aligned} h(N) &= h(\langle L\ x\ R\rangle) = 1 + \max(h(L), h(R)) \\ h(\emptyset) &= 0 \end{aligned}$$

Узел N с соответствуюей ему высотой h будем обозначать N_h.

Множество узлов. Множеством узлов дерева T называется такое множество элементов x, которое состоит из значения узла T и значений его поддеревьев.

$$x \in T \Leftrightarrow T = \langle L\ y\ R\rangle \Rightarrow x = y \vee x \in L \vee x \in R$$

4.1. АВЛ - Дерево

Определение. АВЛ-деревом называется двоичное дерево поиска T, в котором для каждого поддерева $N = \langle L_{h_l}\ x\ R_{h_r}\rangle$ верно неравенство $|h_l - h_r| \leq 1$. АВЛ-дерево является самобалансирующимся. Т.е. все функции его модификации сохраняют это условие.

Функция вставки. Вставку элемента x в дерево T будем обозначать функцией $I(T, x)$. Обычная вставка, без балансировки может выглядеть следующим образом:

$$\begin{aligned} I(T,x):\ & \\ & I(\emptyset, x) = \langle \emptyset\ x\ \emptyset\rangle \\ & I(\langle L\ y\ R\rangle, x) = \\ & \qquad x \leq y \Rightarrow \langle I(L,x)\ y\ R\rangle \\ & \qquad x > y \Rightarrow \langle L\ x\ I(R,x)\rangle \end{aligned}$$

В самобалансирующемся дереве функция вставки должна в себя включать балансировку. Случай, когда $T = \emptyset$ является тривиальным, мы его рассматривать не будем. Расмотрим те случаи, когда условие баланса нарушается и требуется перебалансировка. Вариантов разбалансировки немного. Она может возникать когда высоты двух поддеревьев уже различаются на единицу, например $T = \langle L_h\ R_{h+1}\rangle$ (для простоты будем опускать значение узла дерева), и добавление нового узла увеличивает дисбаланс: $T \to \langle L_h\ R_{h+2}\rangle$. В этом случае, для восстановления баланса, должен быть выбран новый корень из правого поддерева R.

Для этого допустим, что правое поддерево состоит из двух своих поддеревьев: $R_{h+2} = \langle P_m\ Q_n \rangle$, где, по условию, $\max(m, n) = h + 1$. Тогда, функция балансировки $\mathcal{B}(T)$ — трансформация дерева T в новое дерево, в корне которого будет узел R будет выглядеть так:

$$\begin{aligned}\mathcal{B}(\langle L_h\ R_{h+2}\rangle) = \quad & \mathcal{B}(\langle L_h\ \langle P_m\ Q_n\rangle\rangle) = \\ & = \langle\langle L_h\ P_m\rangle\ Q_n\rangle\end{aligned}$$

Если $m < n$, тогда $m = h$, а $n = h+1$. В этом случае, выбор узла R в качестве корня нового дерева сохраняет баланс:

$$\big\langle\langle L_h\ P_h\rangle_{h+1}\ Q_{h+1}\big\rangle_{h+2}$$

В противном случае, дерево останется разбалансированным:

$$\big\langle\langle L_h\ P_{h+1}\rangle_{h+2}\ Q_h\big\rangle_{h+3}$$

Этот случай указывает нам, что мы “отхватили” от правого поддерева R слишком большой кусок и отдали его в левое поддерево. Попробуем забрать поменьше. Разделим дополнительно дерево $P_{h+1} = \langle M_m\ N_n\rangle$ и передадим в левое поддерево только его левую часть M:

$$\begin{aligned}\mathcal{B}(T) = \mathcal{B}(\langle L_h\ R_{h+2}\rangle) = \\ = \mathcal{B}(\langle L_h\ \langle P_{h+1}\ Q_h\rangle\rangle) = \\ = \mathcal{B}(\big\langle L_h\ \big\langle\langle M_m\ N_n\rangle_{h+1}\ Q_h\big\rangle\big\rangle) = \\ = \langle\langle L_h\ M_m\rangle\ \langle N_n\ Q_h\rangle\rangle\end{aligned}$$

Поскольку $\max(m, n) = h$, то и $\max(h, m) = h$ и $\max(n, h) = h$, а следовательно, высоты левого $\langle L\ M\rangle$ и правого $\langle N\ Q\rangle$ поддеревьев стали равны $\max(h, m) + 1 = \max(n, h) + 1 = h + 1$ и все дерево оказадлось опять сбалансированным.

Если обобщить все варианты разбалансировки слева и справа, то получится 4 варианта реализации функции $\mathcal{B}(T)$:

$$\begin{aligned}\mathcal{B}(T) = \\ T = \langle L_h\ R_{h+2}\rangle \Rightarrow \\ R = \langle P_h\ Q_{h+1}\rangle \Rightarrow \langle\langle L_h\ P_h\rangle\ Q_{h+1}\rangle \\ R = \langle\langle M_m\ N_n\rangle\ Q_h\rangle \Rightarrow \langle\langle L_h\ M_m\rangle\ \langle N_n\ Q_h\rangle\rangle \\ T = \langle L_{h+2}\ R_h\rangle \Rightarrow \\ L = \langle P_{h+1}\ Q_h\rangle \Rightarrow \langle P_{h+1}\ \langle Q_h\ R_h\rangle\rangle \\ L = \langle K_h\ \langle M_m\ N_n\rangle\rangle \Rightarrow \langle\langle K_h\ M_m\rangle\ \langle N_n\ R_h\rangle\rangle\end{aligned}$$

Примечательным является то, что можно осуществлять балансировку, имея только балансы деревьев. Будем считать баланс дерева b как разницу высот поддеревьев: положительным, если высота правого поддерева больше левого, отрицалельным в обратном случае, и равным нулю, если высоты равны. Обозначать баланс дерева будем верхним индексом T^b, чтобы не путать с высотой.

$$\begin{aligned}\mathcal{B}(T) = \\ T = \langle L\ R\rangle^2 \Rightarrow \\ R = \langle P\ Q\rangle^1 \Rightarrow \Big\langle\langle L\ P\rangle^0\ Q\Big\rangle^0 \\ R = \Big\langle\langle M\ N\rangle^b\ Q\Big\rangle^{-1} \Rightarrow \Big\langle\langle L\ M\rangle^{\min(-b,0)}\ \langle N\ Q\rangle^{\max(-b,0)}\Big\rangle^0 \\ T = \langle L\ R\rangle^{-2} \Rightarrow \\ L = \langle P\ Q\rangle^{-1} \Rightarrow \Big\langle P\ \langle Q\ R\rangle^0\Big\rangle^0 \\ L = \Big\langle K\ \langle M\ N\rangle^b\Big\rangle^1 \Rightarrow \Big\langle\langle K\ M\rangle^{\min(-b,0)}\ \langle N\ R\rangle^{\max(-b,0)}\Big\rangle^0\end{aligned}$$

4.2. Реализация

Будем делать нашу структуру максимально абстрактной, так, чтобы она могла хранить объекты произвольной природы. Главное, чтобы на множестве этих объектов была определена операция сравнения, а также, если эти объекты сложные, то еще потребуются операции копирования и освобождения памяти (конструктор и деструктор).

Итак, вставка узла. Функция `_tree_insert` принимает на вход Структуру дерева с интерфейсными функциями, указатель на узел и указатель на значение, которое должно быть помещено в это дерево. Данная функция возвращает 0, если высота дерева не изменилась и 1, если высота дерева увеличилась. Это поможет изменить баланс вышестоящему узлу. Результат рекуррентного вызова `_tree_insert` будем сохранять в переменную `dbal` модифицировать при помощи нее баланс получившегося узла. В этой функции мы будем использовать функциональную структуру, которая не будет модицифироваться, а будет замешаться на новую, через конструктор `_Tr`

28a ⟨*Вставка нового узла* 28a⟩≡ (35b)
```
int _tree_insert(Tree* t, Node** n, void* data) {
    Node* x = *n;
    int dbal;
    if ( !x ) {
        *n = _Tr(NULL, t->cpy(data), NULL, 0);
        return 1;
    }
    if ( t->cmp(data, x->data) <= 0 )
        dbal = -_tree_insert(t, &x->l, data);
    else
        dbal =  _tree_insert(t, &x->r, data);

    *n = _Tr(x->l, x->data, x->r, x->bal + dbal);
    dbal = ABS((*n)->bal) > ABS(x->bal) ? 1 : 0;
    free(x);
    return dbal;

}
```
Defines:
`_tree_insert`, used in chunk 28b.
Uses `_Tr` 28c, `ABS` 35b, `Node` 29b, and `Tree` 29c.

Саму эту функцию использовать напрямую не очень удобно, поэтому мы её обернем интерфейсным методом:

28b ⟨*Интерфейс* 28b⟩≡ (35b) 30b ▷
```
int tree_insert(Tree* t, void* data) {
    return _tree_insert(t, &t->root, data);
}
```
Defines:
`tree_insert`, used in chunks 35a and 36.
Uses `_tree_insert` 28a and `Tree` 29c.

Конструктор дерева выглядит, с одной стороны, просто, поскольку заполняет поля структуры узла `Node`. Но этот же конструктор также должен следить за балансом. Идея такая, что если требуется балансировка, то конструктор будет рекурсивно вызывать себя для перестроения поддеревьев.

28c ⟨*Создание нового узла с балансировкой* 28c⟩≡ (35b)
```
Node* _Tr(Node* l, void* data, Node* r, int bal) {
    Node* res = NULL;
    ⟨Какая-то балансирующая магия 29a⟩

    res = malloc(sizeof(Node));
    *res = (Node){l, r, data, bal};
    return res;
}
```
Defines:
`_Tr`, used in chunks 28a, 29a, and 32–34.
Uses `Node` 29b.

Собственно, сама балансировка, по большей части, повторяет структуру функции $\mathcal{B}$:

29a ⟨*Какая-то балансирующая магия* 29a⟩≡ (28c)
```
if ( bal == 2 ) {
    Node R = *r;
    if ( R.bal < 0 ) {
        Node RL = *R.l;
        res = _Tr(_Tr(l, data, RL.l, MIN(-RL.bal, 0)), RL.data,
                  _Tr(RL.r, R.data, R.r, MAX(-RL.bal, 0)), 0);
        free(R.l);
    } else
        res = _Tr(_Tr(l, data, R.l, 0), R.data, R.r, 0);
    free(r);
} else if ( bal == -2 ) {
    Node L = *l;
    if ( L.bal > 0 ) {
        Node LR  = *L.r;
        res = _Tr(_Tr(L.l, L.data, LR.l, MIN(-LR.bal, 0)), LR.data,
                  _Tr(LR.r, data, r, MAX(-LR.bal, 0)), 0);
        free(l->r);
    } else
        res = _Tr(L.l, L.data, _Tr(L.r, data, r, 0), 0);
    free(l);
}
if (res) return res;
```
Uses `_Tr` 28c, `MAX` 35b, `MIN` 35b, and `Node` 29b.

Заметье, что балансировщик удаляет старые варианты узлов и строит новые, которые используют существующие поддеревья. Дадим теперь определения.

29b ⟨*Определение узла* 29b⟩≡ (35b)
```
typedef struct _Node {
    struct _Node* l;
    struct _Node* r;
    void* data;
    int bal;
} Node;
```
Defines:
`Node`, used in chunks 28–34.

Сами интерфейсные функции по работе со значенями в узлах (копирование `cpy`, сравнение `cmp` и освобождение `free`) ложатся в структуру `Tree`, заодно в ней будем хранить корень дерева.

29c ⟨*Определение интерфейса дерева* 29c⟩≡ (35a)
```
typedef struct _Tree {
    void* (*cpy)(void* data);
    int (*cmp)(void* x, void* y);
    void (*free)(void* data);
    struct _Node* root;
} Tree;
```
Defines:
`Tree`, used in chunks 28, 30, 31, and 34–36.

4.3. Поиск и разрушение

Поиск осуществляется рекурсивно, реузльтат получается через обратный вызов `cb`. Если поиск нужно прекратить `cb` должен вернуть 1.

30a ⟨*Функция поиска* 30a⟩≡ (35b)
```
int _tree_search(Tree* t, Node* n, void* key, int (*cb)(void*)) {
    int cmp;
    if ( !n ) return 0;
    cmp = t->cmp(key, n->data);
    if ( cmp == 0 )
        if ( cb(n->data) ) return 1;
    if ( cmp <= 0 )
        return _tree_search(t, n->l, key, cb);
    else
        return _tree_search(t, n->r, key, cb);

}
```
Defines:
`_tree_search`, used in chunk 30b.
Uses `Node` 29b and `Tree` 29c.

30b ⟨*Интерфейс* 28b⟩+≡ (35b) ◁28b 30d▷
```
int tree_search(Tree* t, void* key, int (*cb)(void*)) {
    return _tree_search(t, t->root, key, cb);
}
```
Defines:
`tree_search`, used in chunk 35a.
Uses `_tree_search` 30a and `Tree` 29c.

Также нам будет полезна функция обхода всего дерева (для отладки и печати). Также делаем ее в паре с интерфейсной. Останавливаемся, если `cb` возвращает ненулевое значение. Дополнительно в `cb` передается уровень узла дерева:

30c ⟨*Функция обхода дерева* 30c⟩≡ (35b)
```
int _tree_walk(Tree* t,  Node* n, int level, int(*cb)(void*, int, int)) {
    if ( !n ) return 0;
    if (_tree_walk(t, n->l, level+1,  cb)) return 1;
    if ( cb(n->data, level, n->bal) ) return 1;
    return _tree_walk(t, n->r, level+1, cb);
}
```
Defines:
`_tree_walk`, used in chunk 30d.
Uses `Node` 29b and `Tree` 29c.

30d ⟨*Интерфейс* 28b⟩+≡ (35b) ◁30b 31b▷
```
int tree_walk(Tree* t, int (*cb)(void*, int, int)) {
    return _tree_walk(t, t->root, 0, cb);
}
```
Defines:
`tree_walk`, used in chunks 35a and 36.
Uses `_tree_walk` 30c and `Tree` 29c.

Разрушение дерева (деструктор) строится по все тому же рекуррентному принципу.

31a ⟨*Деструктор* 31a⟩≡ (35b)

```
void _tree_free(Tree* t, Node* n) {
    Node* l, *r;
    if ( !n ) return;
    t->free(n->data);
    l = n->l;
    r = n->r;
    free(n);
    _tree_free(t, l);
    _tree_free(t, r);
}
```

Defines:
`_tree_free`, used in chunk 31b.
Uses `Node` 29b and `Tree` 29c.

31b ⟨*Интерфейс* 28b⟩+≡ (35b) ◁30d 34b▷

```
void tree_free(Tree* t) {
    _tree_free(t, t->root);
}
```

Defines:
`tree_free`, used in chunks 35a and 36.
Uses `_tree_free` 31a and `Tree` 29c.

4.4. Удаление узла

Удаление узла из дерева, тем более, из самобалансирующегося — это, как и любая другая немонотонность — достаточно сложная операция. Опять попробуем построить теоретическую модель. Начнем строить функцию удаления $\mathcal{D}(T, x)$ с простых и очевидных с случаев:

$$\begin{aligned} \mathcal{D}(\emptyset, x) &= \emptyset, 0 \\ \mathcal{D}(\langle \emptyset\ x\ \emptyset \rangle, x) &= \emptyset, 1 \end{aligned} \tag{4.1}$$

Забегая вперед, мы, по аналогии с функцией вставки, будем помимо результирующего дерева возвращать также факт изменения высоты дерева, чтобы вызывающая сторона могла бы поправить баланс. В данном случае, высота дерева не меняется. Теперь посмотрим, что будет в случае, если дерево не пусто:

$$\begin{aligned} \mathcal{D}(\langle L\ y\ R \rangle^b, x) &= \\ & x < y \Rightarrow \\ & \quad L', d \leftarrow \mathcal{D}(L, x); \\ & \quad \mathcal{B}(\langle L'\ y\ R \rangle^{b+d}), |b+d| < |b| \\ & x > y \Rightarrow \\ & \quad R', d \leftarrow \mathcal{D}(R, x); \\ & \quad \mathcal{B}(\langle L\ y\ R' \rangle^{b-d}), |b-d| < |b| \end{aligned} \tag{4.2}$$

Случай удаления, когда $x = y$, собственно, представляет основную сложность. Что делать с двумя поддеревьями, когда у них нет общего корня. Оказывается, нужно избрать новый корень – это либо самый левый узел правого поддерева R, либо самый правый узел левого поддерева, у которого может быть только один потомок. Введем вспомогательные функции для выбора нового корня. Обозначим функции отбора самого левого правого узлов соответственно $\mathcal{E}_l$ и $\mathcal{E}_r$. Эти функции будут возвращать отобранный узел, новое дерево и факт изменения высоты дерева (для корректировки баланса вызываемой стороной).

$$\begin{aligned} \mathcal{E}_l(T) &= \\ & T = \langle \emptyset\ x\ R \rangle \Rightarrow x, R, 1 \\ & T = \langle L\ x\ R \rangle^b \Rightarrow e, L', d \leftarrow \mathcal{E}_l(L); \\ & \quad e, \mathcal{B}(\langle L'\ x\ R \rangle^{b+d}), |b+d| < |b| \\ \mathcal{E}_r(T) &= \\ & T = \langle L\ x\ \emptyset \rangle \Rightarrow x, L, 1 \\ & T = \langle L\ x\ R \rangle^b \Rightarrow e, R', d \leftarrow \mathcal{E}_r(R); \\ & \quad e, \mathcal{B}(\langle L\ x\ R' \rangle^{b-d}), |b-d| < |b| \end{aligned} \tag{4.3}$$

Теперь случай функии удаления, когда удаляется корень, будет выглядеть следующим образом:

$$\begin{aligned} \mathcal{D}(\langle L\ x\ R \rangle^b, x) &= \\ & b > 0 \Rightarrow e, R', d \leftarrow \mathcal{E}_l(R); \\ & \quad \mathcal{B}(\langle L\ e\ R' \rangle^{b-d}), |b-d| < |b| \\ & b \leq 0 \Rightarrow e, L', d \leftarrow \mathcal{E}_r(L); \\ & \quad \mathcal{B}(\langle L'\ e\ R \rangle^{b+d}), |b+d| < |b| \end{aligned} \tag{4.4}$$

4.5. Реализация удаления

Определим функции отбора самого левого и правого значений $\mathcal{E}_l$ и $\mathcal{E}_r$ дерева, как показано в (4.3):

32 ⟨*Функция отбора левого знаения* 32⟩≡ (35b)

```
int elect_l(Node** n, void** e) {
    Node* x = *n;
    int dbal =  1;
    if ( !x->l ) {
        *n = x->r;
        *e = x->data;
```

```
    } else {
        dbal = elect_l(&x->l, e);
        *n = _Tr(x->l, x->data, x->r, x->bal+dbal);
        dbal = ABS((*n)->bal) < ABS(x->bal) ? 1: 0;
    }
    free(x);
    return dbal;
}
```

Defines:
elect_l, used in chunk 34a.
Uses _Tr 28c, ABS 35b, and Node 29b.

33 ⟨*Функция отбора правого значения* 33⟩≡ (35b)

```
int elect_r(Node** n, void** e) {
    Node* x = *n;
    int dbal = 1;
    if ( !x->r) {
        *n = x->l;
        *e = x->data;
    } else {
        dbal = elect_r(&x->r, e);
        *n = _Tr(x->l, x->data, x->r, x->bal-dbal);
        dbal = ABS((*n)->bal) < ABS(x->bal) ? 1: 0;
    }
    free(x);
    return dbal;
}
```

Defines:
elect_r, used in chunk 34a.
Uses _Tr 28c, ABS 35b, and Node 29b.

Теперь сама функция удаления. Она, по сути, является полным отражением (4.1), (4.2) и (4.4) и использует вспомогательные функции выбора нового корня `elect_l` и `elect_r` в случае, если он удаляется.

34a ⟨*Функция удаления* 34a⟩≡ (35b)

```
int _tree_delete(Tree* t, Node** n, void* data) {
    Node* x = *n;
    int cmp, dbal;
    if ( !x) return 0;
    cmp = t->cmp(data, x->data);

    printf("Comparing %d and %d => %d\n", (int)data, (int)x->data, cmp);
    if ( !x->l && !x->r && !cmp ) {
        printf("Deleted node found, no chidren\n");
        t->free(x->data);
        free(x);
        *n = NULL;
        return 1;
    }
    if ( cmp < 0 )
        dbal = _tree_delete(t, &x->l, data);
    else if ( cmp > 0 )
        dbal = - _tree_delete(t, &x->r, data);
    else {
        t->free(x->data);
        dbal = x->bal > 0 ? -elect_l(&x->r, &x->data)
                          : elect_r(&x->l, &x->data);
    }
    *n = _Tr(x->l, x->data, x->r, x->bal+dbal);
    dbal = ABS((*n)->bal) < ABS(x->bal) ? 1 : 0;
    free(x);
    return dbal;
}
```

Defines:
`_tree_delete`, used in chunk 34b.
Uses `_Tr` 28c, `ABS` 35b, `elect_l` 32, `elect_r` 33, `Node` 29b, and `Tree` 29c.

34b ⟨*Интерфейс* 28b⟩+≡ (35b) ◁31b

```
int tree_delete(Tree* t, void* data) {
    return _tree_delete(t, &t->root, data);
}
```

Defines:
`tree_delete`, used in chunks 35a and 36.
Uses `_tree_delete` 34a and `Tree` 29c.

Соберем прототипы интерфейсных функций в заголовочный файл:

35a ⟨*tree.h* 35a⟩≡
```
#ifndef _TREE_H_
#define _TREE_H_
struct _Node;
⟨Определение интерфейса дерева 29c⟩
int tree_insert(Tree* t, void* data);
int tree_delete(Tree* t, void* data);
int tree_walk(Tree* t, int (*cb)(void*, int, int));
int tree_search(Tree* t, void* key, int (*cb)(void*));
void tree_free(Tree* t);
#endif /* _TREE_H_ */
```
Defines:
`_TREE_H_`, never used.
Uses `Tree` 29c, `tree_delete` 34b, `tree_free` 31b, `tree_insert` 28b, `tree_search` 30b, and `tree_walk` 30d.

А их реализации — в библиотечный файл. Заметьте, что мы не раскрыли определение `struct _Node` — это внутренности библиотеки.

35b ⟨*tree.c* 35b⟩≡
```
#include<stdio.h>
#include<stdlib.h>
#include "tree.h"
#define ABS(x) ((x)> 0? (x): -(x))
#define MIN(x, y) ((x) < (y)? (x) : (y))
#define MAX(x, y) ((x) > (y)? (x) : (y))
⟨Определение узла 29b⟩
⟨Создание нового узла с балансировкой 28c⟩
⟨Вставка нового узла 28a⟩
⟨Функция обхода дерева 30c⟩
⟨Функция поиска 30a⟩
⟨Деструктор 31a⟩
⟨Функция отбора левого знаения 32⟩
⟨Функция отбора правого значения 33⟩
⟨Функция удаления 34a⟩
⟨Интерфейс 28b⟩
```
Defines:
`ABS`, used in chunks 28a and 32–34.
`MAX`, used in chunk 29a.
`MIN`, used in chunk 29a.

Для тестирования сделаем небольшую программу, которая будет вставлять несколько узлов в дерево и печатать его, затем удалять и печатать еще раз:

36 ⟨*testtree.c* 36⟩≡

```
#include<stdio.h>
#include<stdlib.h>
#include "tree.h"
void* int_copy(void* a) {
    return a;
}
void int_free(void* a) {
    return;
}
int int_cmp(void* a, void* b) {
    return (a < b)? -1 : (a > b)? 1: 0;
}

int tree_print(void* data, int level, int bal) {
    int i;
    for ( i = 0; i < level; i++ )
        printf("  ");
    printf("%d (%d)\n", (int)data, bal);
    return 0;
}

int main(int argc, char* argv[]) {
    Tree t;
    int k;
    int m;
    int i;
    t.cpy = int_copy;
    t.free = int_free;
    t.cmp = int_cmp;
    t.root = NULL;
    k = atoi(argv[1]);
    m = atoi(argv[2]);
    for (i = 0; i < k; i++ ) {
        tree_insert(&t, (void*)i);
    }
    tree_walk(&t, tree_print);
    printf("Deleting %d\n", m);
    tree_delete(&t, (void*)m);

    printf("--------\n");
    tree_walk(&t, tree_print);

    tree_free(&t);
    return 0;
}
```

Defines:
`int_cmp`, never used.
`int_copy`, never used.
`int_free`, never used.
`main`, never used.
`tree_print`, never used.
Uses `Tree` 29c, `tree_delete` 34b, `tree_free` 31b, `tree_insert` 28b, and `tree_walk` 30d.

4.6. Сортировка слиянием односвязного списка

Рассмотрим реализацию быстрой сортировки односвязного списка. Она хороша тем, что имеет наихудшее время работы $O(N \log N)$, не требует дополнительной памяти и является стабильной (сохраняет порядок равных элементов).

Определение односвязного списка. Это структура, которая включает в себя полезную нагрузку (в нашем случае — строка `str`) и указатель на хвост списка `next`.

38a ⟨*linked list definition* 38a⟩≡ (42)
```
typedef struct _List {
    char* str;
    struct _List* next;
} List;
```
Defines:
`List`, used in chunks 13, 14, 38–41, and 44–48.

Общая идея быстрой сортировки односвязного списка `lst` выглядит вот так:

38b ⟨*sort definition* 38b⟩≡ (42)
```
void llsort(List** lst) {
    List* a = NULL;
    List* b = NULL;
    if( !*lst || !(*lst)->next )
        return;
    ⟨split 38c⟩
    ⟨sort splits 38d⟩
    ⟨merge 38e⟩
}
```
Defines:
`llsort`, used in chunks 38d and 41d.
Uses `List` 12 38a 44a.

Для начала мы убеждаемся, что список `lst` имеет смысл сортировать: он не пуст и имеет более одного элемента. В противном случае — выходим, список уже отсортирован.

Затем идет разделение списка на 2 (почти) равные части `a` и `b`:

38c ⟨*split* 38c⟩≡ (38b)
```
    llsplit(&a, &b, *lst);
```
Uses `llsplit` 39b.

Затем, эти части сортируются

38d ⟨*sort splits* 38d⟩≡ (38b)
```
    llsort(&a);
    llsort(&b);
```
Uses `llsort` 38b.

И уже отсортированные части склеиваются с сохранением порядка:

38e ⟨*merge* 38e⟩≡ (38b)
```
    llmerge(lst, a, b);
```
Uses `llmerge` 39a.

Теперь введем определения функций. Начнем со склейки двух односвязных списоков с сохранением порядка `llmerge`. Сначала действуем в предположении, что оба списка не пусты, поэтому мы сравниваем их головные элементы. В результирующий список `lst` записываем наменьшую голову одного из списков и рекусивно продолжаем склейку между с его хвостом и другим списком, в качестве результата подставляя адрес следующего элемента результирующего списка. Если один из списков пуст, то просто оставляем в качестве результата оставшийся список.

39a ⟨*merge definition* 39a⟩≡ (42)

```
void llmerge(List** lst, List* a, List* b) {
    if ( a && b) {
        if ( strcmp(a->str, b->str) <= 0 ) {
            *lst = a;
            llmerge(&a->next, b, a->next);
        } else {
            *lst = b;
            llmerge(&b->next, b->next, a);
        }
    } else {
        *lst = a ? a : b;
    }
}
```

Defines:
`llmerge`, used in chunk 38e.
Uses `List` 12 38a 44a.

Разберем теперь идею разделения односвязного списка на две равные части. Опять будем решать задачу рекурсивно. Преположим, что у нас уже есть два разделенных списка и `a` и `b` — адреса их концов. Если разделяемый список `lst` не пуст, поставляем его голову в конец оного из списков (для определенности, пусть это будет `a`), и далее, рекурсивно вызываем разеление оставшегося хвоста, но при этом меняем `a` и `b` местами. Теперь концом первого списка `a` будет конец второго, а концом второго — конец первого (естественно, с учетом того, что он уже немножко подрос).

39b ⟨*split definition* 39b⟩≡ (42)

```
void llsplit(List** a, List** b, List* lst) {
    if ( lst ) {
        *a = lst;
        llsplit(b, &(*a)->next, lst->next);
    } else {
        *a = NULL;
        *b = NULL;
    }
}
```

Defines:
`llsplit`, used in chunk 38c.
Uses `List` 12 38a 44a.

Есть одно замечание, касающееся порядка списков `a` и `b` в рекурсивных вызовах `llsplit` и `llmerge`. Если при разделении мы постоянно меняли `a` и `b` местами, то тоже самое следует сделать и при склейке, чтобы сохранялся порядок равных элементов. Вот, в принципе, и все. Merge-sort на односвязных списках в трех простых функциях.

Не забываем про функции для создания и удаления списков. При добавлении элемента в список потребуется дополнительная память, поэтому нужно обработать ситуации, когда памяти не хватает.

40a ⟨*insert definition* 40a⟩≡ (42)
```
int llinsert(List** lst, const char* str) {
    List* new;
    if ( new = malloc(sizeof(List)) ) {
        if ( new->str = strdup(str) ) {
            new->next = *lst;
            *lst = new;
            return 0;
        }
        free(new);
        return -1;
    }
    return -2;
}
```
Defines:
`llinsert`, used in chunk 41c.
Uses `List` 12 38a 44a.

Освобождение памяти, занятой списком происходит в обратном порядке. Мы сохраняем значения поля `next` на случай, если после освобождении головы `lst` оно будет разрушено. Можно, конечно, сначала рекусивно вызвать `llfree` на хвосте, потом уже освободить голову, но при этом рекурсия перестанет быть хвостовой и может быть неоптимизирована компиляторами.

40b ⟨*free definition* 40b⟩≡ (42)
```
void llfree(List* lst) {
    if ( lst ) {
        List* next = lst->next;
        free(lst->str);
        free(lst);
        llfree(next);
    }
}
```
Defines:
`llfree`, used in chunk 41f.
Uses `List` 12 38a 44a.

Заметьте, что `llfree` хоть и удаляет, но не делает полезную модификацию `lst`, поэтому, в отличие от других функций - модификаторов, мы передаем ей только указатель на `List`.

Тестирование Для тестирования нам потребуется функция вывода элементов односвязного списка `llprint`. Также выполним ее в виде рекурсии.

40c ⟨*llprint definition* 40c⟩≡ (42)
```
void llprint(List* lst) {
    if (lst) {
        printf("%s", lst->str);
        llprint(lst->next);
    }
}
```
Defines:
`llprint`, used in chunk 41e.
Uses `List` 12 38a 44a.

Основая программа для тестирования состоит из четырех основных этапов:

1. ⟨*read text into list* 41c⟩ — чтение из стандарного потока ввода,
2. ⟨*sort list* 41d⟩ — сортировака,
3. ⟨*output sorted list* 41e⟩ — вывод отсортированного списка в стандартный поток вывода
4. ⟨*free list* 41f⟩ – освобождение ресурсов, связанных со списком.

41a ⟨*main function* 41a⟩≡ (42)
```
int main() {
    ⟨variables 41b⟩
    ⟨read text into list 41c⟩
    ⟨sort list 41d⟩
    ⟨output sorted list 41e⟩
    ⟨free list 41f⟩
    return -1;
}
```
Defines:
`main`, never used.

Для работы нам потребуется указатель на список и буфер для чтения строк. Выбор размера буфера, конечно, зависит от природы строк. Здесь мы предполагаем, что размер строки (в байтах) не превышает 4 Кб. Для *произвольных строк* с неограниченной длиной такая реализация **не подойдет!**

41b ⟨*variables* 41b⟩≡ (41a)
```
char buf[4096];
List* lst = NULL;
```
Uses `List` 12 38a 44a.

Чтение текста из потока по строкам — это функция `fgets`. Она принимает на вход помимо адреса буфера также максимальный размер этого буфера. Если строка превысит этот размер, она обрежется и следующий вызов `fgets` вернет в буфере строку, начинающуюся с места, в котором прервался предыдущий вызов:

41c ⟨*read text into list* 41c⟩≡ (41a)
```
while( fgets(buf, sizeof(buf), stdin) ) {
    llinsert(&lst, buf);
}
```
Uses `llinsert` 40a.

Сортировка — это просто вызов `llsort`:

41d ⟨*sort list* 41d⟩≡ (41a)
```
llsort(&lst);
```
Uses `llsort` 38b.

Выводим отсортированный список на экран:

41e ⟨*output sorted list* 41e⟩≡ (41a)
```
llprint(lst);
```
Uses `llprint` 40c.

И освобождаем его:

41f ⟨*free list* 41f⟩≡ (41a)
```
llfree(lst);
```
Uses `llfree` 40b.

Целиком программа выглядит вот так:

42 ⟨*llsort.c* 42⟩≡
```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```
⟨*linked list definition* 38a⟩
⟨*merge definition* 39a⟩
⟨*split definition* 39b⟩
⟨*sort definition* 38b⟩
⟨*insert definition* 40a⟩
⟨*free definition* 40b⟩
⟨*llprint definition* 40c⟩
⟨*main function* 41a⟩

4.7. Задания

1. Проверьте на практике и докажите, что данный алгоритм является стабильным (сохраняет порядок равных элементов). Подсказка: *модифицируйте функцию сравнения так, чтобы она сравнивала первые n элементов.*

2. Докажите, что время сортировки не зависит от значений элементов и составляет $O(N \log N)$.

3. Измените алгоритм разделения `llsplit` так, чтобы он делал только одно изменение (разрезал список пополам в середине). Сохранится ли в этом случае свойство стабильности?

Глава 5

Очередь

5.1. Реализация очереди на односвязных списках

Определение. Очередью (queue, pipe или FIFO) называется абстрактная структура данных, хранящая упорядоченную последовательность (список) элементов, возможно, различной природы и позволяющая удалять элементы только с начала последовательности и добавлять только в конец (либо наоборот). Очередь нам потребуется для реализации других алгоритмов (обход графа в ширину).

Для начала нам потребуется определение элемента списка произвольной природы. Это структура, которая содержит абстрактный указатель `elt` и указатель на следующий элемент списка `next`.

44a ⟨*определение абстрактного списка* 44a⟩≡ (47 48a)
```
typedef struct _List {
    void* elt;
    struct _List* next;
} List;
```
Defines:
List, used in chunks 13, 14, 38–41, and 44–48.

Основная проблема реализации очереди связана с тем что добавлять (enqueue) и удалять (dequeue) элементы из нее нужно за константное время $O(1)$, т.е. не зависящее от размера очереди. Это возможно сделать, если нам известно два указателя — на первый и последний элементы очереди, назовем их `head` и `last`.

Например, функции добавления `enqueue` и удаления `dequeue` могут выглядеть следующим образом:

44b ⟨*enqueue реализация 1* 44b⟩≡
```
int enqueue(List** head, List** last, void*** elt) {
    ⟨создать новый элемент 44c⟩
    ⟨обнулить хвост 44d⟩
    ⟨вернуть указатель на полезную нагрузку 44e⟩
    ⟨добавить элемент в существующий хвост last 44f⟩
    ⟨заместить новым элементом старый last 44g⟩
    ⟨случай пустой очереди 44h⟩
    return 0;
}
```
Defines:
enqueue, used in chunks 45a, 47, and 48a.
Uses List 12 38a 44a.

Очередь сама заботится об аллокации памяти для своей структуры. Но может случиться, что аллокация невозможна — в этом случае выходим с ошибкой.

44c ⟨*создать новый элемент* 44c⟩≡ (44b 46b)
```
    List* new = malloc(sizeof(List));
    if ( ! new ) return -1;
```
Uses List 12 38a 44a.

44d ⟨*обнулить хвост* 44d⟩≡ (44b)
```
    new->next = NULL;
```

44e ⟨*вернуть указатель на полезную нагрузку* 44e⟩≡ (44b 46b)
```
    *elt = &new->elt;
```

Данная строчка может показаться странной. Объяснение дано ниже.

44f ⟨*добавить элемент в существующий хвост* last 44f⟩≡ (44b)
```
    if (*last)
        (*last)->next = new;
```

44g ⟨*заместить новым элементом старый* last 44g⟩≡ (44b)
```
    *last = new;
```

44h ⟨*случай пустой очереди* 44h⟩≡ (44b)
```
    if ( ! *head ) *head = new;
```

Эта реализация требует пояснений. Помимо двух двойных указателей в аргументах функции используется тройной(!) указатель на `void`. На самом деле это проявление “абстрактности” данной структуры. Фнукция `enqueue` не занимается аллокацией памяти для элемента очереди а *возвращает указатель* на ту область памяти, куда вызывающая сторона сможет записать *указатель* на реальные данные.

Чтобы не ошибиться в количестве символов `` `*' `` можно прибегнуть к мнемоническому правилу: сколько раз в описании аргумента встречается слов *‘возвращает’*, *адрес* или *‘указатель’* столько `` `*' `` и надо ставить после типа (перед именем). Т.е. в нашем случае — это `` `void***' ``. Можно это также прочитать так: нам требуется, чтобы функция *вернула* [1] *указатель* [2] на ту область памяти, куда мы смогли бы записать значение `void*`. В результате, опять поучаем тип `void***`. Он же и подсказывает нам использование данной функции:

45a ⟨*возможное использование enqueue* 45a⟩≡
```
List* head = NULL;
List* tail = NULL;
void** pdata;
...
if(!enqueue(&head, &tail, &pdata)) {
    *pdata = malloc(...);
}
...
```
Uses `enqueue` 44b 46b and `List` 12 38a 44a.

Таким образом, данная функция *возвращает* нам данные типа `void**`, которые мы будем использовать как *адрес*, по которому мы положим реальные данные.

Это же правило применимо и первым двум аргументам `head` и `last`. Функция `enqueue` *возвращает* нам *указатели* на `List`.

45b ⟨*dequeue реализация 1* 45b⟩≡
```
int dequeue(List** head, List** last, void** elt) {
    List* next, *old = *head;
    ⟨обработать случай пустого списка 45c⟩
    ⟨вернуть полезную нагрузку 45d⟩
    ⟨сохранить следующий элемент очереди 45e⟩
    ⟨обработать случай последнего элемента 45f⟩
    ⟨освободить память 45g⟩
    ⟨сместить начало очереди на следующий элемент 46a⟩
    return 0;
}
```
Defines:
`dequeue`, used in chunks 47 and 48a.
Uses `List` 12 38a 44a.

45c ⟨*обработать случай пустого списка* 45c⟩≡ (45b)
```
if ( ! old ) return -1;
```

45d ⟨*вернуть полезную нагрузку* 45d⟩≡ (45b 46e)
```
*elt = old->elt;
```

45e ⟨*сохранить следующий элемент очереди* 45e⟩≡ (45b)
```
next = old->next;
```

45f ⟨*обработать случай последнего элемента* 45f⟩≡ (45b)
```
if ( old == *last )
    *last = NULL;
```

45g ⟨*освободить память* 45g⟩≡ (45b 46e)
```
free(old);
```

46a ⟨*сместить начало очереди на следующий элемент* 46a⟩≡ (45b)
```
*head = next;
```

Хранить два указателя не совсем удобно, да и не нужно. Можно оставить себе один указатель, “замкнув” список так, чтобы последний элемент указывал не на `NULL`, а на голову списка, при этом достаточно хранить только `last`, что и будет нашей очередью. Соответственно, наши функции немного видоизменятся:

46b ⟨*enqueue простая реализация* 46b⟩≡ (47 48b)
```
int enqueue(List** queue, void*** elt) {
    ⟨создать новый элемент 44c⟩
    ⟨вернуть указатель на полезную нагрузку 44e⟩
    ⟨вставить элемент между последним и первым 46c⟩
    ⟨сместить указатель очереди на новый элемент 46d⟩
    return 0;
}
```
Defines:
`enqueue`, used in chunks 45a, 47, and 48a.
Uses `List` 12 38a 44a.

Изменяется только следующие части:

46c ⟨*вставить элемент между последним и первым* 46c⟩≡ (46b)
```
    if ( ! *queue ) {
        new->next = new;
    } else {
        new->next = (*queue)->next;
        (*queue)->next = new;
    }
```

Если очередь состоит из одного элемента — замкнуть его на себя.

46d ⟨*сместить указатель очереди на новый элемент* 46d⟩≡ (46b)
```
    *queue = new;
```

Соответствующим образом изменяется фукнция `dequeue`:

46e ⟨*dequeue простая реализация* 46e⟩≡ (47 48b)
```
int dequeue(List** queue, void** elt) {
    List* old;
    ⟨обработать случай пустой очереди 46f⟩
    ⟨изъять голову из списка 46g⟩
    ⟨вернуть полезную нагрузку 45d⟩
    ⟨случай последнего элемента 46h⟩
    ⟨освободить память 45g⟩
    return 0;
}
```
Defines:
`dequeue`, used in chunks 47 and 48a.
Uses `List` 12 38a 44a.

46f ⟨*обработать случай пустой очереди* 46f⟩≡ (46e)
```
    if (! *queue ) return -1;
```

46g ⟨*изъять голову из списка* 46g⟩≡ (46e)
```
    old = (*queue)->next;
    (*queue)->next = old->next;
```

46h ⟨*случай последнего элемента* 46h⟩≡ (46e)
```
    if ( old == *queue) *queue = NULL;
```

Вот и вся реализация очереди.

В качестве примера использования очереди приведем алгоритм синтеза звука Карплуса-Стронга.

47 ⟨*karplus.c* 47⟩≡

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
⟨определение абстрактного списка 44a⟩
⟨enqueue простая реализация 46b⟩
⟨dequeue простая реализация 46e⟩
int main() {
    List *queue = NULL;
    void** pdata;
    int v, f = 0, n = 20;
    while(!feof(stdin) && fscanf(stdin, "%d ", &v)) {
        if ( enqueue(&queue, &pdata) )
            break;
        *pdata = (void*)v;
    }

    while(!dequeue(&queue, (void**)&v) && n) {
        printf("%d\n", v);
        if ( v == f ) n--;
        else n = 20;

        f = f/2 + v/2;
        if ( enqueue(&queue, &pdata) )
            break;
        *pdata = (void*)f;
    }

    while (!dequeue(&queue, (void**)&v) ) {
        printf("%d\n", v);
    }
}
```

Defines:
`main`, never used.
Uses `dequeue` 45b 46e, `enqueue` 44b 46b, and `List` 12 38a 44a.

5.2. Задание

Использовать рекурсию.

1. Упростите функцию `enqueue` в предположении, что полезная нагрузка `elt` уже аллоцирована.

2. Разберите реализацию алгоритма синтеза звука удара по струне Карплуса-Стронга. Модифицируйте его для 1) чисел с плавающей точкой, 2) для строк текста так, чтобы буквы из двух строк перемежались между собой (случайным образрм) в пропорции 1:1, 3*) для строк текста с перемежающимися словами.

5.3. Реализаци очереди в виде библиотеки

Поскольку, в будущем, нам потребуется реализация очереди (для обхода графа в глубину), офофрмим ее в виде отдельного модуля, который состоит из заголовочного файла `llqueue.h` и файла `llqueue.c`

48a ⟨*llqueue.h* 48a⟩≡

```
#ifndef _LLQUEUE_H_
#define _LLQUEUE_H_
⟨определение абстрактного списка 44a⟩
int enqueue(List** q, void*** data);
int dequeue(List** q, void** data);
#endif
```

Defines:
  `_LLQUEUE_H_`, never used.
Uses `dequeue` 45b 46e, `enqueue` 44b 46b, and `List` 12 38a 44a.

48b ⟨*llqueue.c* 48b⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include "llqueue.h"

⟨enqueue простая реализация 46b⟩
⟨dequeue простая реализация 46e⟩
```

Глава 6

Поиск в ширину

Рассмотрим группу задач, которые сводятся к задаче поиска *в ширину* (breadth-first search). Эта группа задач так или иначе связана с нахождением кратчайшего или оптимального пути, минимального числа ходов в какой-либо игре или распространение фронта в неоднородных средах. В любом случае, пространство поиска представляется в виде графа, а решение задачи – это цепочка ребер, отвечающая какому-то заданному условию минимума. Сама природа значений, связанных с ребрами или с узлами не так важна. Важно, чтобы эти значения были неизменны в процессе решения.

Пусть задан граф $G = (V, E)$, состоящий из вершин V и рёбер E. Предположим, что заданы некоторые значения на ребрах $f(E) \to X$ из некоторого множества, на котором определен порядок элементов, а также агрегирующий оператор $X \oplus X \to X$, обладающий свойствами дистрибутивности и коммутативности и монотонности на множестве X: $\forall a, b \in X a \oplus b \geq a$.

И пусть заданы начальная и конечная вершины $v_0, v_t \in V$. Требуется найти такую неповторяющуюся последовательность вершин, последовательно соединенных ребрами:

$$P_{0 \to t} = \{v_{p_i}, i = 0 \dots k\}, v_{p_0} = v_0, v_{p_k} = v_t : \forall i < k e_{p_i} = (v_{p_i}, v_{p_{i+1}}) \in E,$$

которая дает минимальное значение $P_{\min} = \operatorname{argmin} Q(P)$, где Q вычисляется по формуле:

$$Q(P) = f(v_{p_0}, v_{p_1}) \oplus f(v_{p_1}, v_{p_2}) \oplus \dots \oplus f(v_{p_{k-1}}, v_t), v_{p_i} \in P$$

Идея. Основная идея поиска пути, при котором достигается минимум какого-то агрегата (и при этом чтобы вершины пути не повторялись) заключается в сохранении промежуточного значения этого агрегата в вершинах графа, а сами вершины обходить способом, похожим на принцип распространения фронта волны.

Допустим, у нас есть некоторый *фронт*, состоящий из вершин графа v_j, для которых вычислено значения агрегата $Q(v_j)$. Для каждой вершины фронта v_j выбираются смежные вершины v_i такие, что на них еще значение агрегата не задано, ($Q(v_i) = +\infty$), либо $Q(v_i) > Q(v_j) + f(v_j, v_i)$. Каждая такая вершина добавляется к вершинам фронта, а v_j – удаляется. Причем первой удаляется вершина, добавленная первой. В результате получается новый фронт, который, по сути, является очередью вершин. Изначально фронт состоит из одной вершины v_0, и процесс обхода графа и вычисления агрегата продолжается пока фронт не пуст. Результатом процесса будет либо найденное минимальное значение $Q(v_t)$, либо отсутствие пути $v_0, \dots v_t$, в этом случае, $Q(v_t)$ не будет вычислено ($Q(v_t) = +\infty$). Этот алгоритм называется алгоритмом Дейкстры.

6.1. Лабиринт

Одной из наиболее распространенных (учебных) задач, которые позволяют продемонстрировать подход поиска в ширину — это задача поиска кратчайшего пути в плоском лабиринте между входом и выходом. Лабиринт – это поле, состоящее из клеток, каждая из которых может быть либо пустой, либо стеной. Путь может состоять только из пустых клеток. Вход и выход, разумеется, также являются пустыми клетками. Размер лабиринта конечен, он окружен (невидимыми) стенами, функция “стоимости” перехода между клетками — единица, а агрегирующий оператор – простое суммирование, так что путь в лабиринте характеризуется просто числом клеток в нем.

Будем читать структуру лабиринта из текстового файла, в котором будет m строк по n символов. Символ `'.'` (точка) будет означать пустую клетку, символ `'#'` (решетка) — стену, а символы `'i'` и `'o'` — соответственно, вход и выход. Результатом решения задачи поиска кратчайшего пути будет все то же представление лабиринта в виде матрицы символов, но с обозначенным кратчайшим путем (одним из) при помощи символов `'p'` и обозначаением количества клеток в нем.

Вот пример задания файла с лабиринтом

```
################################################
#...#..........#...........#........#.....o#
#...#..........#.....#.....#........#.....#
#.#.######.....####..###.....#....##...#.....#
#.#..###...#.....#.....###....#...##....#.....#
#i#..#.....#.....#.....##.....#....##...#.....##
#.##.....###....#.....####..##.....##..###....#
#.#.......#...........#.........##.........##
################################################
```

Общая структура программы будет выглядеть так:

51a ⟨*labyrinth.c* 51a⟩≡

```
⟨Необходимые библиотеки 53b⟩
⟨Определения 53a⟩
⟨Глобальные переменные 51b⟩

⟨Вспомогательные функции 52c⟩
int main(int argc, char* argv[]) {
    char* filename;
    int err = 0;
    if ( argc < 2 ) {
        fprintf(stderr, "usage: labyrinth <file.lab>\n");
        exit(1);
    }
    filename = argv[1];
    ⟨Прочитать файл лабиринта 52a⟩
    ⟨Найти кратчайший путь 54g⟩
    if( ⟨Путь найден 55a⟩ ) {
        ⟨Напечатать лабиринт с кратчайшим путем 55c⟩
    } else {
        ⟨Решения не существует 56b⟩
    }
    ⟨Деинициализация 56c⟩
    return 0;
}
```

Defines:
`main`, never used.

Теперь потихонечку начнем.

Сначала определим формат представления лабиринта. Пусть это будет массив целых числел `Lab`, который мы будем использовать одновременно как информацию о геометрии лабиринта (стены и пустые клетки), так и о длине пути от входа.

51b ⟨*Глобальные переменные* 51b⟩≡ (51a) 51c▷

```
unsigned int* Lab;
```

Defines:
`Lab`, used in chunks 52c, 53d, and 56c.

Также нам потребуется размер лабиринта, координаты входа и выхода. Пусть это будут целочисленные ширина `W` и высота `H`. Вход и выход будут хранится в парах `IX,IY` и `OX,OY` соответственно.

51c ⟨*Глобальные переменные* 51b⟩+≡ (51a) ◁51b 52b▷

```
unsigned int W, H;
unsigned int IX, IY;
unsigned int OX, OY;
```

Defines:
`H`, used in chunks 52c, 53d, and 56a.
`IX`, used in chunks 52c, 54f, and 56a.
`IY`, used in chunks 52c, 54f, and 56a.
`OX`, used in chunks 52c, 55, and 56a.
`OY`, used in chunks 52c, 55, and 56a.
`W`, used in chunks 52c, 53d, and 56a.

Прочитаем лабиринт из файла. Если не удастся это сделать – напечатаем ошибку и выйдем. Будем читать файл построчно, длина первой строки будет определять ширину лабиринта, а количество строк – высоту. Каждый раз, когда строка будет зачитана, будем заполнять массив `Lab` в соответствии со следующими правилами: Всякий символ, не равный `'.'`, `'i'` или `'o'` будет восприниматься как клетка со стеной и иметь значение `0`, в противном случае — это пустая клетка, которая будет иметь значение `-1` (аналог $+\infty$ для `unsigned int`). Соответственно, встретившиеся символы `'i'` и `'o'` дадут нам координаты `IX,IY` и `OX,OY`.

52a ⟨*Прочитать файл лабиринта* 52a⟩≡ (51a)
```
if ( err = lab_read(filename) ) {
    fprintf(stderr, "Error while reading labyrinth: %d\n", err);
    exit(1);
}
```
Uses `lab_read` 52c.

Сама реализация функции `lab_read` содержит в себе чтение буфера `BUF`, его анализ и заполнение массива `Lab`.

52b ⟨*Глобальные переменные* 51b⟩+≡ (51a) ◁51c 53c▷
```
char BUF[4096];
```
Defines:
`BUF`, used in chunks 52c and 56a.

52c ⟨*Вспомогательные функции* 52c⟩≡ (51a) 54e▷
```
int lab_read(char* filename) {
    int i, k;
    FILE *f = fopen(filename, "r");
    if ( !f ) return -1;
    W = 0;
    H = 0;
    Lab = NULL;
    while(!feof(f)) {
        if ( !fgets(BUF, sizeof(BUF), f)) break;
        if ( !H ) W = strlen(BUF);
        Lab = realloc(Lab, sizeof(*Lab)*W*(H+1));
        for ( i = 0; i < W && BUF[i]; i++ ) {
            switch(BUF[i]) {
            case 'i': case 'o': case '.':
                *(Lab + W*H + i) = -1; break;
            default:
                *(Lab + W*H + i)  = 0;
            }
            if (BUF[i] == 'i') { IX = i; IY = H; }
            else if ( BUF[i] == 'o') { OX = i; OY = H; }
        }
        while ( i < W ) *(Lab + W*H + i++) = 0;
        H++;
    }
    fclose(f);
    return 0;
}
```
Defines:
`lab_read`, used in chunk 52a.
Uses `BUF` 52b, `H` 51c, `IX` 51c, `IY` 51c, `Lab` 51b, `OX` 51c, `OY` 51c, and `W` 51c.

В результате у нас получится массив `Lab` размера `W * H` с элементами `0` и `-1`.

Теперь перейдем к самому главному — нахождению кратчайшего пути. Для этого нам потребуется очередь. Воспользуемся реализацией из предыдущего раздела. В очереди будем хранить пару координат клетки `x` и `y`, для которых мы создадим структуру

53a ⟨*Определения* 53a⟩≡ (51a) 53d▷
```
typedef struct _cell {
    int x, y;
} Cell;
```
Defines:
`Cell`, used in chunks 53 and 54.

Подключим необходимые заголовочные файлы

53b ⟨*Необходимые библиотеки* 53b⟩≡ (51a)
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "llqueue.h"
```

Создадим глобальную переменую с очередью.

53c ⟨*Глобальные переменные* 51b⟩+≡ (51a) ◁52b
```
List* Queue;
```
Defines:
`Queue`, used in chunks 53 and 54.

Теперь определим рекурсивную функцию обхода лабиринта с фронтом в очереди `Queue`. Если очередь пуста – прекращаем работу и проверяем, дошли ли мы до выхода. Немного упростим себе задачу, добавив макросы для проверки правильности шага из текущей клетки `IS_VALID`, нахождения внутри границ лабиринта `IN_BOUNDS`, а также доступ к значению в клетке `CELL`:

53d ⟨*Определения* 53a⟩+≡ (51a) ◁53a
```
#define IS_VALID(x, y) ((x)*(x) + (y)*(y) > 0 && (x)*(y) == 0)
#define IN_BOUNDS(X, Y)  ((X)>= 0 && (X) < W && (Y)>=0 && (Y) < H)
#define CELL(X, Y) *(Lab + W*(Y) + (X))
```
Defines:
`CELL`, used in chunks 54–56.
`IN_BOUNDS`, used in chunks 54c and 55b.
`IS_VALID`, used in chunks 54c and 55b.
Uses `H` 51c, `Lab` 51b, and `W` 51c.

53e ⟨*Функция обхода* 53e⟩≡ (54e)
```
int walk() {
    Cell* c;
    Cell** front;
    int dx, dy;
    unsigned int p;
    if ( ! Queue ) {
        printf("Front is empty!\n");
        return 0;
    }
    ⟨Читаем текущую клетку из фронта 54a⟩
    ⟨Запоминаем число шагов до текущей клетки 54b⟩
    ⟨Перебираем возможные варианты путей и формируем новый фронт 54c⟩
    return walk();

}
```
Defines:
`walk`, used in chunk 54g.
Uses `Cell` 53a and `Queue` 53c.

54a ⟨*Читаем текущую клетку из фронта* 54a⟩≡ (53e)
```
dequeue(&Queue, (void**)&c);
```
Uses Queue 53c.

54b ⟨*Запоминаем число шагов до текущей клетки* 54b⟩≡ (53e)
```
p = CELL(c->x, c->y);
```
Uses CELL 53d.

При проверке возможного шага воспользуемся тем, что `-1` для `unsigned int` — это максимальное число, а стена – это 0. Число шагов `p` у нас строго положительно, поэтому перезаписывать значение в клетке будем только если оно больше чем `p+1`. Т.е. это точно не стена, а клетка, либо еще не пройденная, либо с большим числом шагов (хотя, в нашем случае такая ситуация невозможна):

54c ⟨*Перебираем возможные варианты путей и формируем новый фронт* 54c⟩≡ (53e)
```
for( dx = -1; dx <= 1; dx++ ) {
    for ( dy = -1; dy <= 1; dy++ ) {
        if ( IS_VALID(dx, dy) && IN_BOUNDS(c->x + dx, c->y + dy) ) {
            if ( CELL(c->x + dx, c->y + dy) > p + 1 ) {
                CELL(c->x + dx, c->y + dy) = p + 1;
                ⟨Добавить новую клетку во фронт 54d⟩
            }
        }
    }
}
free(c);
```
Uses CELL 53d, IN_BOUNDS 53d, and IS_VALID 53d.

Далее следует немного Си-шной магии. Это получение адреса области памяти в которую нужно будет записать указатель на нашу структуру `Cell` и заполнение ее новыми значениями. Заметьте, что заполнить структуру, доступную по указателю можно не прибегая к серии присваиваний отдельных ее полей, а сразу приравнять известным значениям. Следим за количеством “звезд”:

54d ⟨*Добавить новую клетку во фронт* 54d⟩≡ (54c)
```
enqueue(&Queue, (void***)&front);
*front = malloc(sizeof(Cell));
**front = (Cell){c->x + dx, c->y + dy};
```
Uses Cell 53a and Queue 53c.

Добавляем нашу функцию в список вспомогательных функций.

54e ⟨*Вспомогательные функции* 52c⟩+≡ (51a) ◁52c 54f▷
⟨*Функция обхода* 53e⟩

А само нахождение кратчайшего пути выразим в функции инициализации с последующим вызовом функции обхода `walk`:

54f ⟨*Вспомогательные функции* 52c⟩+≡ (51a) ◁54e 55b▷
```
void init_walk() {
    Cell** init;
    CELL(IX, IY) = 1;
    enqueue(&Queue, (void***)&init);
    *init = malloc(sizeof(Cell));
    **init = (Cell){IX, IY};
}
```
Defines:
init_walk, used in chunk 54g.
Uses CELL 53d, Cell 53a, IX 51c, IY 51c, and Queue 53c.

54g ⟨*Найти кратчайший путь* 54g⟩≡ (51a)
```
init_walk();
walk();
```
Uses init_walk 54f and walk 53e.

Как понять, что выход достигнут? Значение в этой клетке должно быть “конечным”.

55a ⟨*Путь найден* 55a⟩≡ (51a)
```
CELL(OX, OY) < (unsigned int)(-1)
```
Uses CELL 53d, OX 51c, and OY 51c.

Теперь не менее сложная часть — это рисование найденного пути. Для его маркировки будем использовать еще одно “близкое к бесконечности” значение `(unsigned int)(-2)`, которое мы врядли достигнем. Строить найденный путь, очевидно, лучше всего с конца:

55b ⟨*Вспомогательные функции* 52c⟩+≡ (51a) ◁54f 56a▷
```
int walk_back(int x, int y, int p) {
    int dx, dy;
    if (p == 1) return 0;
    for (dx = -1; dx <= 1; dx++ ) {
        for (dy = -1; dy <=1; dy++ ) {
            if ( IS_VALID(dx, dy) && IN_BOUNDS(x+dx, y+dy) ) {
                if ( CELL(x+dx, y+dy) == p - 1 ) {
                    CELL(x+dx, y+dy) = (unsigned int)(-2);
                    return walk_back(x+dx, y+dy, p - 1);
                }
            }
        }
    }
    return -1;
}
```
Defines:
walk_back, used in chunk 55c.
Uses CELL 53d, IN_BOUNDS 53d, and IS_VALID 53d.

55c ⟨*Напечатать лабиринт с кратчайшим путем* 55c⟩≡ (51a)
```
walk_back(OX, OY, CELL(OX, OY));
print_lab();
```
Uses CELL 53d, OX 51c, OY 51c, print_lab 56a, and walk_back 55b.

Функция печати будет выглядеть обратной функции чтения. Читаем построчно массив `Lab` и заполняем буфер `BUF` символами, соответствующими либо пустой клетке, либо стене, либо пути, либо точкам входа и выхода.

56a ⟨*Вспомогательные функции* 52c⟩+≡ (51a) ◁55b

```
void print_lab() {
    int x, y;
    for ( y = 0; y < H; y++ ) {
        for ( x = 0; x < W; x++ ) {
            switch(CELL(x, y)) {
            case 0:
                BUF[x] = '#'; break;
            case (unsigned int)(-2):
                BUF[x] = '.'; break;
            default:
                if ( x == IX && y == IY )
                    BUF[x] = 'i';
                if ( x == OX && y == OY )
                    BUF[x] = 'o';
                else
                    BUF[x] = ' ';
            }
        }
        BUF[W] = '\0';
        puts(BUF);
    }
}
```

Defines:
`print_lab`, used in chunk 55c.
Uses `BUF` 52b, `CELL` 53d, `H` 51c, `IX` 51c, `IY` 51c, `OX` 51c, `OY` 51c, and `W` 51c.

В случае отсутствия пути, напечатаем сообщение:

56b ⟨*Решения не существует* 56b⟩≡ (51a)

```
fprintf(stderr, "Path not found!\n");
```

Не забудем подчистить за собой массив.

56c ⟨*Деинициализация* 56c⟩≡ (51a)

```
free(Lab);
```

Uses `Lab` 51b.

Ну вот, опять все. Кстати, результат решения задачи будет выглядеть как-то так:

```
Front is empty!
##################################################
#    #          #  .....     #         #  ...o##
#... #          #  .  #..    #.......  #  .   ##
#.#.######...   ####. ###..   #.   ##.  #  .   ##
#.#. ###...#.    #...  ###.   #.  ## .  #  .   ##
#.#..#...  #..   #.    ## ..  #.   ##.. #  .  ###
# ##...   ###.   #.    ####. ##.    ##. ###.   ##
# #      #  ......    #   .....  ##  ......  ###
##################################################
```

6.2. Задания

1. Разрешите шагать в ближайшие клетки по диагонали.

2. Разрешите шагать в ближайшие клетки по диагонали, только измените стоимость шага так, чтобы хождения по осям прибаляло к пути 10, а хождение по диагонали – 14.