Computer-Science-Exam

11.01.2026 14:11 #MSU/SS3 #CS

Основные конструкции языка C++. Полиморфизм, инкапсуляция, наследование. Конструкторы и деструкторы. Перегрузка унарных и бинарных операторов. Утечка памяти. Исключения. Ключевые слова (что означают, зачем используются, примеры использования): try, catch, const, static, public, private, protected, friend, template.

Основа (объект, класс)

В понятие объекта будем вкладывать следующий смысл:

1. Способ организации данных (деление данных на элементы, определение взаимодействий между отдельными элементами данных, задание *дисциплины доступа* к данным)
2. Правила модификации данных (*алгоритмы*), обеспечивающие обработку и преобразование данных
3. Правила взаимодействия данного объекта с другими объектами (*интерфейс*).

Итак,

> **Def**
>
> *Объект (object)* — это некоторая конструкция, которая обладает внутренним состоянием (набором данных) и набором функций, позволяющих это состояние модифицировать. Функции из этого набора могут быть доступны внешнему использованию (public-интерфейс), но также могут быть закрыты от внешнего воздействия и использоваться только как внутренние (private) алгоритмы объекта.

На C++ объект реализуется с помощью классов.

> **Def**
>
> *Класс (class)* — это структура, которая содержит некоторый набор данных и также набор методов (функций), которые могут выполнять работу с данными этого класса, преобразуя эти данные и тем самым меняя состояние конкретного экземпляра класса.
> По Шилдту: класс — это механизм для создания объектов.

Короче говоря, класс — это такой шаблон, а объект — это его конкретная реализация, класс — это кусок исходного кода, а объект — это кусок памяти в компьютере (надеюсь, питонисты не прочитают этот текст..).

Концепции ООП (полиморфизм, инкапсуляция, наследование)

Все языки объектно-ориентированного программирования (в том числе C++) основаны на трёх основных концепциях, называемых полиморфизмом, инкапсуляцией и наследованием.

Инкапсуляция

Def

Инкапсуляция (encapsulation) — механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства и неправильного использования.

Tip

Когда данные и код объединяются согласно инкапсуляции, создаётся *объект* (объект — это то, что поддерживает инкапсуляцию).

Короче говоря, этот принцип гласит, что вам не надо знать что происходит внутри объекта, когда вы жмёте на какие-то кнопки (используете публичные методы) («чёрный ящик»).
Таким образом, благодаря этому принципу, у нас и данные защищены, и не надо думать над тем, как там всё устроено внутри объекта.

Полиморфизм

Def

Полиморфизм (polymorphism) — это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

Например, в C были функции `fabs()`, `labs()`, `abs()` для конкретного типа данных, а в объектно-ориентированном C++ каждая из них могла быть названа просто `abs()`.
Например, практически везде ограниченно применяется полиморфизм в арифметических операторах: в C символ «+» используется и для целых чисел, и для чисел с плавающей точкой.
Полиморфизм позволяет манипулировать объектами разной степени сложности путём создания для них стандартного интерфейса для реализации похожих действий. Таким образом, благодаря этому принципу, достаточно помнить и использовать общий интерфейс объектов.

Дополнение

Есть такие понятия как *перегрузка функций* (function overloading) и *перегрузка операторов* (operator overloading). Это такой полиморфизм, при котором компилятор сам «по контексту» выберет какую функцию (оператор) выбрать в данном случае.

Наследование

Def

> *Наследование (inheritance)* — это процесс, посредством которого один объект может приобретать свойства другого. Более точно, объект может наследовать основные свойства родительского объекта и добавлять к ним черты, которые характерны только для него.

Например, я — объект класса «студент мехмата», но студент мехмата — это часть более общего класса — «студент МГУ» и так далее.
Данный принцип важен для поддержания иерархии классов (hierarchical classification).

Классы в C++

```
class MyClass {
    // закрытые функции и переменные класса
public:
    // открытые функции и переменыне класса
} //список объектов (кто-то хоть раз тут что-то писал?)
;
```

Пример:

```
class MyClass {
    int a;
public:
    void set_a (int num);
    int get_a ();
};

void MyClass::set_a (int num) {
a = num;
}

int MyClass::get_a () {return a;}

int main() {
    MyClass ob1, ob2;
    ob1.set_a(10);
    ob2.set_a(99);
    cout << ob1.get_a() << " " << ob2.get_a() << endl;
    return 0;
}
```

В Шилдте переводчик гений и общепринятый перевод member function в виде метода заменил на какую-то функцию-член. Короче, метод — это функция, которая принадлежит классу или структуре. Она работает с конкретным объектом и имеет доступ к его внутреннему состоянию. Вызывается точкой после названия объекта.

В отличие от структур, классы обеспечивают инкапсуляцию, наследование и полиморфизм.

ляляляляляяяя ляляя ляляяяяяяя

Права доступа

Кто может обращаться к членам класса?

- **Public** — доступ открыт всем.
- **Protected** — доступ открыт только классу и его наследникам
- **Private** — доступ есть только внутри класса, права доступа не наследуются.

Пример:

```
class Base {
public:
    int public_member;
protected:
    int protected_member;
private:
    int private_member;
};

class Derived : public Base {
public:
    void access_members() {
        public_member = 1; // Доступно
        protected_member = 2; // Доступно
        // private_member = 3; // Ошибка: private_member недоступен
    }
};

int main() {
    Derived d;
    d.public_member = 4; // Доступно
    // d.protected_member = 5; // Ошибка: protected_member недоступен извне
    return 0;
}
```

Шаблоны

Тут всё просто: `template<typename T>` или `template<class T>`.

Конструкторы

Конструктор отвечает за то, в каком состоянии будет создаваться класс при его объявлении в программе.
Пример:

```
class MyClass {
    int a;
    int b;
public:
```

```
    MyClass() = default; // конструктор называется так же, как и класс
    MyClass() {a = b = 0;}
    MyClass(int n, int m) {a = n; b = m;}
    MyClass(int n = 0, int m = 0) {a = n; b = m;}
    MyClass(int n = 0, int m = 0) : a(n), b(m) {}
    void set_a (int num);
    int get_a ();
};
```

Деструкторы

Деструктор отвечает за уничтожение класса. По сути он освобождает ресурсы, захваченные классом во время его жизни. Пример:

```
class MyClass {
    int a;
public:
    MyClass() = default;
    ~MyClass() = default; // обозначается с тильдой
    void set_a (int num);
    int get_a ();
};
```

Конструктор копирования

```
class MyClass {
public:
    int val;
    MyClass(int v) : val(v) {}
    // конструктор копирования
    MyClass(const MyClass& other) : value(other.val) {}
};
```

Он вызывается при инициализации объекта другим объектом:

```
MyClass obj1(10);      // обычный конструктор
MyClass obj2(obj1);    // конструктор копирования
MyClass obj3 = obj1;   // тоже конструктор копирования (не присваивание!)
```

и при всяких передачах объекта в функцию или возвратах объекта из функции по значению.

Перегрузка унарных и бинарных операторов

Def

Перегрузка (overloading) операторов — это процедура определения встроенных операторов (+, -, *, /, ++, =, //, ** и т.д.) для объектов класса.

Def

Оператор присваивания копированием — оператор, который копирует состояние одного объекта в другой существующий объект.

Его можно вызвать двумя способами: как функцию со своим специфическим именем, либо как выражение с кратким значком операции:

```
const MyClass& MyClass::operator= (const MyClass &c);
// вызовы:
a.operator=(b);
// или
a = b;
```

Примеры:

```
Complex Complex::operator +(const Complex &a){
  Complex tmp;
  tmp.re=re+a.re;
  tmp.im=im+a.im;
  return tmp;
}
```

```
// оператор присваивания копированием (всегда возвращают *this)
const NumberA & operator=(const NumberA & v) { value = v.value; error = v.error; return *this;
}
const NumberA & operator=(double v) { return *this = NumberA(v); }

// а вообще надо делать по-умному:
class My_Array {
    int * array;
    int count;
public:
    My_Array & operator = (const My_Array & other) {
        if (this != &other) { // защита от неправильного самоприсваивания
            // 1: выделяем "новую" память и копируем элементы
            int * new_array = new int[other.count];
            std::copy(other.array, other.array + other.count, new_array);
            // 2: освобождаем "старую" память
            delete [] array;
            // 3: присваиваем значения в "новой" памяти объекту
            array = new_array;
            count = other.count;
        } // по соглашению всегда возвращаем *this
        return *this;
    }
};
```

```cpp
class Vector {
private:
    double x, y;
public:
    // 1. Бинарный оператор + (как метод класса)
    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }

    // 2. Унарный оператор - (префиксный, как метод класса)
    Vector operator-() const {
        return Vector(-x, -y);
    }

    // 3. Бинарный оператор += (возвращает ссылку на себя)
    Vector& operator+=(const Vector& other) {
        x += other.x;
        y += other.y;
        return *this;
    }

    // 4. Унарный оператор ++ (постфиксный). Фиктивный int - признак постфиксной формы.
    Vector operator++(int) {
        Vector temp = *this; // Сохраняем старое значение
        x++;
        y++;
        return temp; // Возвращаем старое значение
    }
    // Префиксный ++
    Vector& operator++() {
        ++x; ++y;
        return *this;
    }
};

// 5. Бинарный оператор << для вывода (обычно внешняя дружественная функция)
std::ostream& operator<<(std::ostream& os, const Vector& v) {
    os << "(" << v.x << ", " << v.y << ")";
    return os;
}
```

Правило трёх

Вообще если программист не определит конструктор копирования, присваивания (копированием) и деструктор, то компилятор создаст их по своему разумению. В простых случаях этого и хватит. Однако в более запущенных случаях считается, что эти три покемона должны быть вместе, а именно:

Правило трёх

Если программист явно определяет хотя бы одну из функций: конструктор копирования, присваивание (копированием), деструктор, то он должен явно определить и остальные, поскольку если для одной из функций не хватает дефолтной функциональности, то, скорее всего, её не хватает и для оставшихся.

Утечка памяти

Def

Утечка памяти (memory leak) — ситуация, когда динамически выделенная память (через `new` или `malloc`) не освобождается (с помощью `delete` или `free`) после окончания её использования.

Important

RAII (Resource Acquisition Is Initialization) — принцип, который гласит, что жизненный цикл ресурса должен быть связан с жизненным циклом объекта: ресурс должен выделяться в конструкторе и освобождаться в деструкторе, обеспечивая автоматическое управление.

Проблема в том, что программа будет потреблять много оперативной памяти и может аварийно завершить работу.
В классах прописываются деструкторы, поэтому при работе с классами можно не волноваться по поводу утечек памяти.
Как проверить? Ну, можно использовать Valgrind (на Linux) или AddressSanitizer.

Исключения

Def

Исключения — механизм обработки ошибок и исключительных ситуаций, который позволяет отделить код, генерирующий ошибку, от кода, который её обрабатывает.

Принцип работы:

1. При возникновении ошибки код **генерирует (`throws`)** исключение — объект любого типа (чаще всего классы, унаследованные от `std::exception`).
2. Управление немедленно передаётся вверх по стеку вызовов.
3. Исключение может быть **перехвачено (`catch`)** в любом подходящем месте стека, где есть соответствующий обработчик.
4. Если исключение не перехвачено, программа аварийно завершается.

Пример:

```cpp
#include <iostream>
#include <stdexcept>

double divide(int a, int b) {
    if (b == 0) {
        throw std::invalid_argument("Division by zero!"); // ГЕНЕРАЦИЯ
    }
    return static_cast<double>(a) / b;
}

int main() {
    int x, y;
    std::cin >> x >> y;
    try { // БЛОК, ГДЕ МОЖЕТ ВОЗНИКНУТЬ ИСКЛЮЧЕНИЕ
        double result = divide(x, y);
        std::cout << "Result: " << result << std::endl;
    }
    catch (const std::invalid_argument& e) { // ОБРАБОТЧИК
        std::cerr << "Error: " << e.what() << std::endl;
    }
    catch (const std::exception& e) { // Более общий обработчик
        std::cerr << "Standard exception: " << e.what() << std::endl;
    }
    catch (...) { // Перехват ЛЮБОГО исключения
        std::cerr << "Unknown exception!" << std::endl;
    }
    return 0;
}
```

Ключевые слова

try

Начало блока кода, в котором могут быть сгенерированы исключения.

catch

Начало блока-обработчика для исключения определённого типа.

const

1. **Константность объекта:** запрещает изменение
`const int size;`
2. **Константный метод:** гарантирует, что метод не меняет состояние объекта
`void print() const;`
3. **Константный параметр:** гарантирует, что функция не изменит аргумент
`void func(const BigObject& obj);`

static

1. **Статическая переменная в функции:** сохраняет значение между вызовами
`int counter() { static int c=0; return ++c;}`

2. **Статический член класса:** один на весь класс, а не на объект
`static int totalCount;`
3. **Статический метод:** принадлежит классу, а не объекту, может обращаться только к статическим членам
`static int getTotal() { return totalCount; }`

public

Модификатор доступа. Члены, объявленные после `public:`, доступны из любого места программы.

private

Модификатор доступа. Члены доступны **только** методам этого же класса и `friend`-элементам. Основа инкапсуляции.

protected

Модификатор доступа. Члены доступны методам этого класса и его наследникам, а также `friend`-элементам.

friend

Объявляет функцию или класс **"другом"**, предоставляя ему доступ к `private` и `protected` членам. Нарушает инкапсуляцию, но полезен для перегрузки операторов.
`friend std::ostream& operator<<(...);`

template

см. раздел «Шаблоны».

Структуры

Так-то это должно было быть на первом курсе.
Грубо говоря, структура — это класс, у которого все поля по умолчанию public.
Пример:

```
struct Point {
    int x;
    int y;
    void print() {
        cout << "Point(" << x << ", " << y << ")" << endl;
    }
};
```

Класс vs структура

```
struct ExampleStruct {
    int data;  // по умолчанию public
};

class ExampleClass {
```

```
    int data;  // по умолчанию private
};
```

```
struct BaseStruct {
    int publicData;
};

struct DerivedStruct : BaseStruct {  // наследование по умолчанию public
    // можем напрямую обращаться к publicData
};

class BaseClass {
    int privateData;
public:
    int publicData;
};

class DerivedClass : BaseClass {  // наследование по умолчанию private
    // не можем напрямую обращаться к publicData из BaseClass
    // (так как оно унаследовано как private)
};

// чтобы исправить:
class CorrectDerived : public BaseClass {  // явно указываем public
    // теперь publicData доступно
};
```

Стек, дек, очередь, приоритетная очередь. Непрерывные реализации на базе массива. Примеры алгоритмов на основе структур данных.

Там дальше будут случаи, когда в комментариях пишу «+ аналогичный с const» — это значит, что надо написать два метода: const T& <метод> () const {...} и T& <метод> () {...} для того, чтобы можно было вызывать метод для const-объектов.

Стек

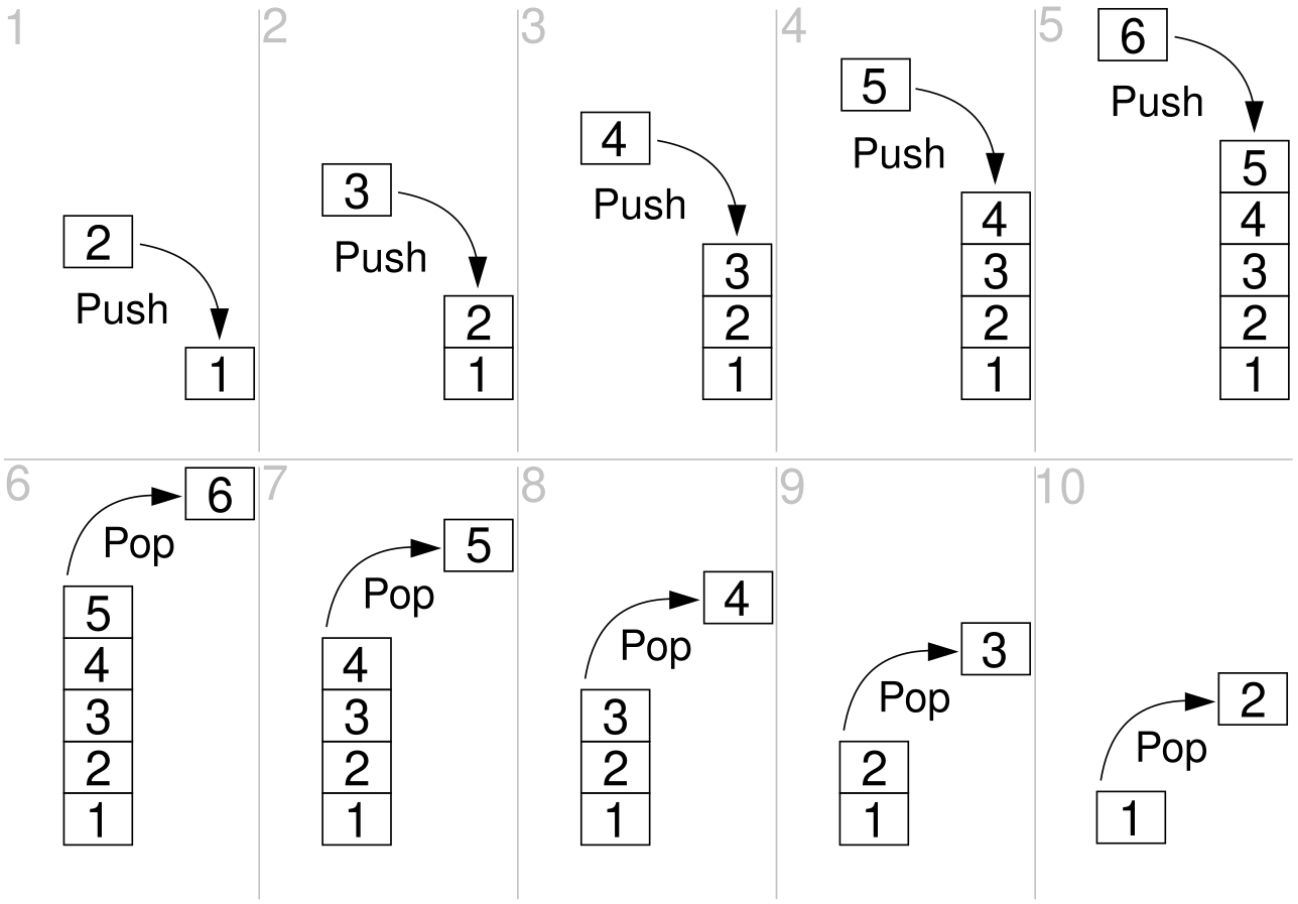
Def

Стек (stack) — структура данных, организованная по принципу LIFO (last in — first out). То есть такая структура данных, которая умеет:

1. добавлять элемент в конец;
2. удалять последний элемент;
3. смотреть последний.
   Каждую из этих операций умеет за $O(1)$.

Функционал:

1. Создание стека
2. Уничтожение стека
3. Добавление элемента в стек
4. Удаление вершины из стека
5. Изъятие элемента из стека
6. Чтение/изменение вершины стека
7. Очистка стека
8. Проверка наличия места в стеке
9. Проверка стека на пустоту



Краткий вариант:

```
class Stack {
    int* data;
    int capacity;
    int topIndex;
public:
    Stack(int size) : capacity(size), topIndex(-1) {
        data = new int[capacity];
    }
    void push(int x) {
        if (topIndex < capacity - 1) {
            data[++topIndex] = x;
        }
    }
    void pop() {
        if (topIndex >= 0) topIndex--;
```

```
    }
    int top() { // + аналогичный с const
        if (topIndex >= 0) {
            return data[topIndex];
        }
        throw std::underflow_error("Stack is empty");
    }
};
```

Реализация от В.Д. Валединского:

```
template <class T>
class Stack {
private:
    T *mem, *top;
    size_t sz;
public:
    Stack (size_t mxsz) : sz(mxsz) {
        mem = new T[sz];
        top = mem - 1;
    }
    Stack (const Stack<T>& st) : sz(st.sz) {
        mem = new T[sz];
        size_t sz2 = st.size();
        for (size_t i = 0; i < sz2; i++) {mem[i] = st.mem[i];}
        top = mem + sz2 - 1;
    }
    ~Stack () {delete [] mem;}

    void push (const T& x);
    void pop ();
    T& top ();
    const T& top() const;
    size_t size() const {return top - mem + 1;}
    size_t capacity () const {return sz;}
};

template <class T>
void Stack<T>::push (const T& x) {
    if (size() < sz) {
        *(++top) = x;
    }
}

template <class T>
void Stack<T>::pop () {
    if (size() > 0) top--;
}

template <class T>
T& Stack<T>::top () {
    if (size() == 0) throw std::range_error("Stack::top: empty stack");
```

```
    return *top;
}
```

Дек

> **Def**
>
> *Дек (deque — double-ended queue)* — структура данных, в которой элементы можно добавлять (удалять, смотреть) как в начале, так и в конце.

Функционал:

1. Создание дека
2. Уничтожение дека
3. Добавление элемента в начало / конец дека
4. Изъятие элемента из начала / конца дека
5. Удаление элемента из начала / конца дека
6. Чтение начала / конца дека
7. Изменение начала / конца дека
8. Очистка дека
9. Проверка наличия места в деке
10. Проверка дека на пустоту

Простой пример:

```
class Deque {
    int* data;
    int capacity;
    int front, rear, size;
public:
    Deque(int cap) : capacity(cap), front(0), rear(0), size(0) {
        data = new int[capacity];
    }
    void push_back(int x) {
        if (size < capacity) {
            data[rear] = x;
            rear = (rear + 1) % capacity;
            size++;
        }
    }
    void push_front(int x) {
        if (size < capacity) {
            front = (front - 1 + capacity) % capacity;
            data[front] = x;
            size++;
        }
    }
```

```
    int front_element() { // + аналогичный с const
        if (size > 0) {
            return data[front];
        }
        throw std::underflow_error("Deque is empty");
    }
    int back_element() { // + аналогичный с const
        if (size > 0) {
            int back_pos = (rear - 1 + capacity) % capacity;
            return data[back_pos];
        }
        throw std::underflow_error("Deque is empty");
    }
};
```

Реализация от В.Д. Валединского:

```
template <class T>
class Deque {
private:
    T *mem, *endmem, *front, *back;
    size_t sz;
    T *Next (T *p) {
        return (p == endmem) ? 0 : p + 1;
    }
    T * Prev (T *p) {
        return (p == 0) ? endmem : p - 1;
    }
public:
    Deque (size_t mxsz) {
        mem = new T[mxsz];
        endmem = mem + mxsz - 1;
        back = mem;
        front = Next(back);
        sz = 0;
    }
    ~Deque () {delete [] mem;}
    void push_front (const T& x) {
        if (sz == max_size()) return;
        front = Prev(front);
        *front = x;
        sz++;
    }
    void push_back (const T& x) {
        if (sz == max_size()) return;
        back = Next(back);
        *back = x;
        sz++;
    }
    bool pop_back () {
        if (sz == 0) return false;
        back = Prev(back);
        sz--;
```

```
    }
    bool pop_front () {
        if (sz == 0) return false;
        front = Next(front);
        sz--;
    }
    bool del_head () {
        if (sz == 0) return false;
        head = Prev(head);
        sz--;
        return true;
    }
    T& front () {
        if (sz == 0) throw std::range_error("Deque::front() : empty queue\n");
        return *front;
    }
    T& back ();
    size_t size() const {return size;}
    size_t max_size() const {return endmem - mem + 1;}
};
```

Очередь

> **Def**
>
> *Очередь (queue)* — структура данных, организованная по принципу FIFO (first in — first out).

Функционал:

1. Создание очереди
2. Уничтожение очереди
3. Добавление элемента в конец очереди
4. Изъятие элемента из начала очереди
5. Удаление элемента из начала очереди
6. Изменение конца очереди
7. Очистка очереди
8. Проверка наличия места в очереди
9. Проверка очереди на пустоту

По факту полностью повторяет дек, только убираем функцию «добавить в начало» и «удалить в конце»
Прототип:

```
template <class T>
class Queue {
private:
    T *mem, *endmem, *head, *tail;
    int sz;
public:
```

```
    Queue();
    ~Queue();
    int put(T val); // push_back из дека
    int take(T *val); // pop_front из дека
    int del(); // del_head из дека
    // и прочее барахло
};
```

```
class Queue {
    int* data;
    int capacity;
    int front, rear, size;
public:
    Queue(int cap) : capacity(cap), front(0), rear(-1), size(0) {
        data = new int[capacity];
    }
    void enqueue(int x) {
        if (size < capacity) {
            rear = (rear + 1) % capacity;
            data[rear] = x;
            size++;
        }
    }
    void dequeue() {
        if (size > 0) {
            front = (front + 1) % capacity;
            size--;
        }
    }
    int front_element() { // + аналогичный с const
        if (size > 0) {
            return data[front];
        }
        throw std::underflow_error("Queue is empty");
    }
    int back() { // + аналогичный с const
        if (size > 0) {
            return data[rear];
        }
        throw std::underflow_error("Queue is empty");
    }
};
```

Очередь с приоритетом

Здесь расскажу только концепцию: суть заключается в том, что данная структура поддерживает только две операции: добавить элемент и извлечь максимум (или минимум). То есть есть два метода:
`insert(ключ, значение)` — добавляет пару (ключ, значение) в хранилище
`extract_min()` — возвращает пару (ключ, значение) с минимальным значением ключа, удаляя её из хранилища.

В качестве примера очереди с приоритетом можно рассмотреть список задач работника. Когда он заканчивает одну задачу, он переходит к очередной — самой приоритетной (ключ будет величиной, обратной приоритету) — то есть выполняет операцию извлечения максимума. Начальник добавляет задачи в список, указывая их приоритет, то есть выполняет операцию добавления элемента.

Общее

Очередь с приоритетом выкидываем, дальше разговор без неё.
В среднем все вышеприведённые структуры умеют добавлять, удалять и вытаскивать крайний для себя элемент за $O(1)$ и требуют $O(n)$ памяти.

Примеры алгоритмов

Стек

- Задача проверки баланса скобок
- DFS

Очередь

- BFS

Дек

- Скользящее окно максимумов (дан массив и длина отрезка, для каждого отрезка найти максимум на нём)

Приоритетная очередь

- Алгоритм Дейкстры

Ссылочные реализации списков. Однонаправленные и двунаправленные списки.

Скорее для общего развития:

Def

Последовательность — это способ организации данных, смысл которого заключается в том, что в конкретный момент времени мы имеем доступ к текущему элементу последовательности, обработав который, можем перейти к следующему элементу. Если снова хотим вернуться к текущему, то надо заново пройти всю последовательность.

Def

Однопроходный алгоритм — это алгоритм, который вычисляет все необходимые характеристики прямо по ходу чтения последовательности, сохраняя минимально возможное количество элементов.

Примеры задач:

1. Подсчёт количества локальных минимумов
2. Среднее арифметическое

Общее развитие закончилось.

Def

Список — это структура данных, которая обладает следующими свойствами:

1. Элементы данных образуют линейную цепочку, то есть для каждого элемента существует «следующий» и «предыдущий» (кроме, конечно же, крайних, у них только по одному соседу)
2. В каждый момент времени в этой линейной цепочке определена некоторая текущая позиция и нам разрешён доступ к элементу в этой текущей позиции (или к его непосредственным соседям)
3. Положение текущей позиции можно изменять и тем самым получать доступ к значению любого элемента данной структуры, не извлекая из неё элементов
4. Добавление и удаление элементов может происходить в окрестности текущей позиции, причём эти операции не должны приводить к массовым пересылкам данных в памяти.

Из-за четвёртого пункта нам придётся отказаться от непрерывности и размещать новые элементы там, где им найдётся место, а не по соседству. Для выполнения первого и второго пункта нам теперь придётся *запоминать* кто у кого сосед. Раньше (хаха, куда я ссылаюсь? ну типа в прошлый вопрос, да) для этого были функции `prev` и `next`, которые могли непосредственно посчитать кто для кого кто.

Def

Двунаправленный список — список, в котором каждый элемент (кроме первого и последнего) ссылается на два соседних с ним элемента.

Соглашения:

1. Каждый элемент списка хранит содержательную информацию и два указателя — на следующий и предыдущий элементы.
2. В каждый момент времени имеется доступ только к двум соседним элементам списка.
3. Удалять можно только соседей.
4. Новые элемент можно добавлять только по соседству.

А что делать с крайними элементами? Ну либо можно обнулить отсутствующих соседей (указатели), либо можно закольцевать список: возьмём элемент `base` и сделаем его предыдущим для первого и последующим для последнего. Зная адрес размещения `base`, знаем краевой ли рассматриваемый элемент.

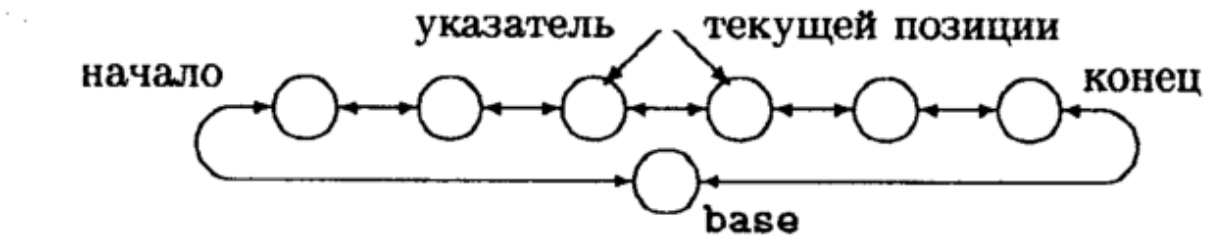


Рис. 6.1. Схема закольцованного двунаправленного списка.

Функционал (система предписаний, как пишет всегда Валединский):

1. Создать список
2. Уничтожить список
3. Добавить элемент до (или за) указателем
4. Удалить элемент до (или за) указателем
5. Прочитать элемент до (или за) указателем
6. Изменить элемент до (или за) указателем
7. Передвинуть указатель вперёд (или назад)
8. Встать в начало (конец) списка
9. Очистить список
10. Список пуст?
11. Указатель в конце (в начале)?

> **Def**
>
> *Однонаправленный список* — список, в котором каждый элемент (кроме первого и последнего) ссылается только на следующий элемент.

Как добавлять и удалять элементы из однонаправленного списка? Есть два варианта:

1. Удаляется текущий элемент, а новым текущим становится следующий за ним. Добавляемый элемент вставляется перед текущим и становится новым текущим. (тут нужен двойной указатель)

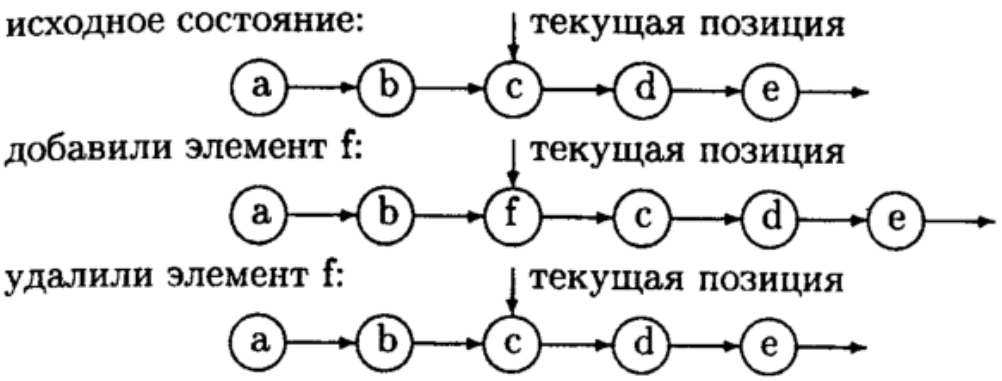


Рис. 6.2. Добавление и удаление элементов при первом подходе.

2. Удаляем не текущий элемент, а следующий за ним, добавляемый элемент ставится за текущим, текущая позиция не меняется.

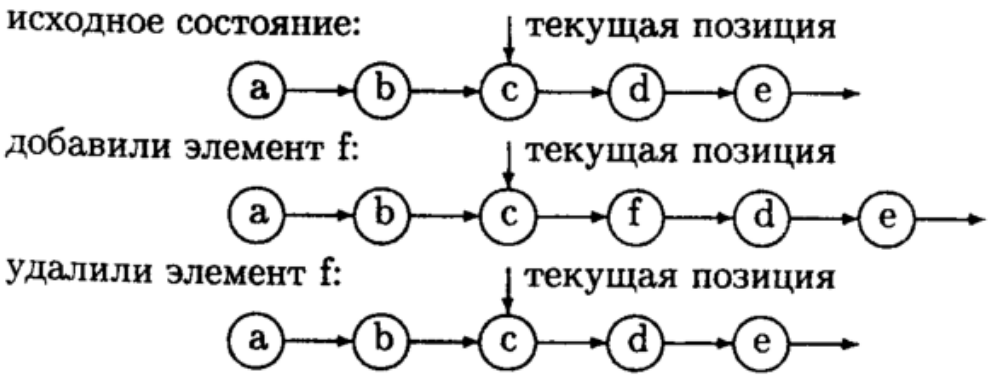


Рис. 6.3. Добавление и удаление элементов при втором подходе.

Система предписаний (для второй реализации):

1. Создать список
2. Уничтожить список
3. Добавить элемент за текущим
4. Удалить элемент за текущим
5. Прочитать текущий элемент
6. Изменить текущий элемент
7. Доступ к элементу за текущим
8. Передвинуть текущую позицию вперёд
9. Встать в начало списка
10. Очистить список
11. Список пуст?
12. Текущая позиция в конце списка?

Деревья. Определения и обходы. Реализация. Бинарные и произвольные (сильноветвящиеся) деревья. Представление произвольного дерева как бинарного.

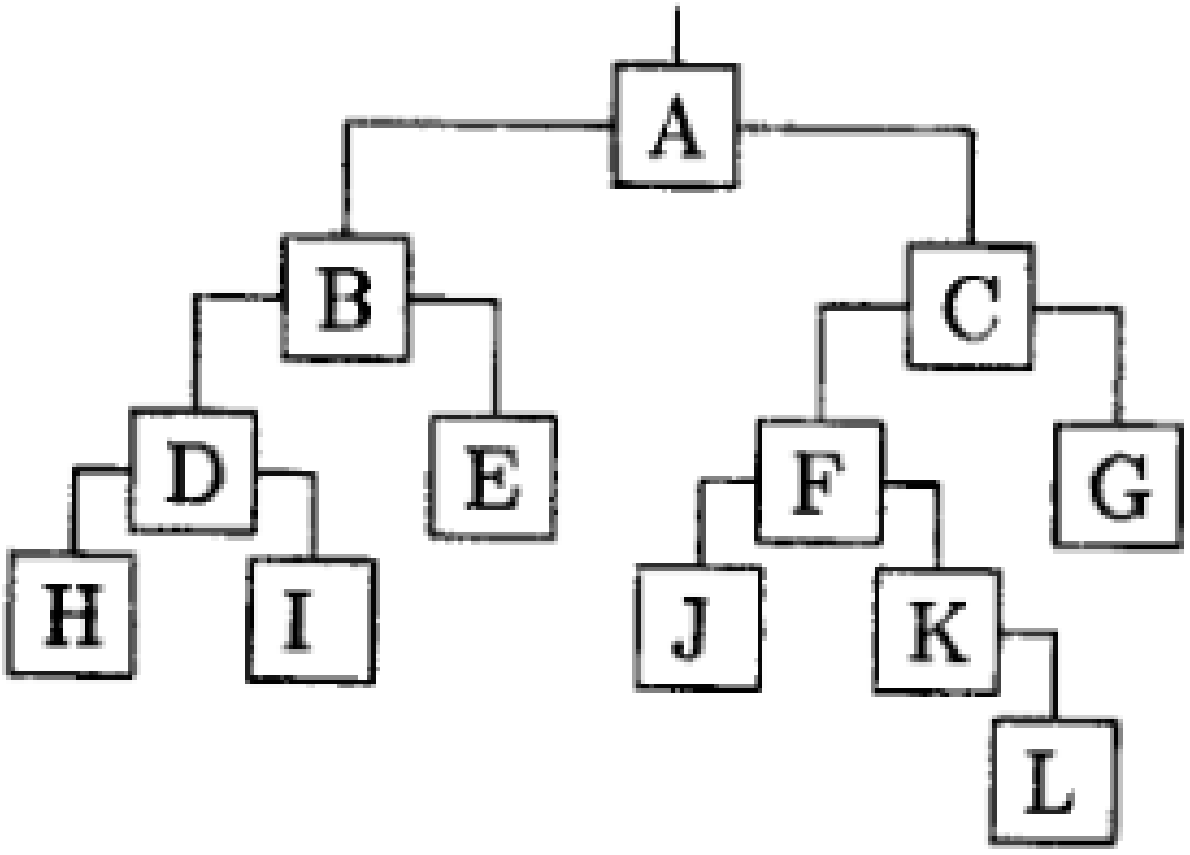
Def

Дерево (tree) — связный граф без циклов.
Расстояние между двумя вершинами — количество рёбер графа, содержащихся в связной цепочке рёбер.
Вершина B — *потомок* вершины A, если расстояние между A и B равно 1 и расстояние от A до корня меньше, чем расстояние от B до корня дерева.
Вершина A — *родитель* вершины B.
Будем говорить, что вершина A принадлежит k-му уровню дерева, если расстояние от A до корня дерева равно k.
Ветвь дерева — связная последовательность вершин, начинающаяся в корне и оканчивающаяся на вершине, не имеющей потомков.

Def

Дерево, в котором каждая вершина имеет не более двух потомков, называется *бинарным*, в противном случае будем дерево называть *произвольным*.

Обход бинарного дерева




```
class TreeNode {
public:
    Type val;
```

```
    TreeNode *prev; // ссылка на родителя (на самом деле почти никогда не используется, если дерево имеет рекурсивное представление)
    TreeNode *left, *right;
};
```

Особый интерес представляет задача *обхода графа* (надо обойти все вершины, а вы что думали). Ею и займёмся для бинарного дерева. Вообще дерево можно определить рекурсивно, тогда и обходить можно рекурсивно:

```
// сверху вниз
void Up_Down (TreeNode *pos, Type *max) {
    if (!pos) return;
    if (*max < pos->val) *max = pos->val;
    Up_Down(pos->left, max);
    Up_Down(pos->right, max);
}
// слева направо
void Left_Right (TreeNode *pos, Type *max) {
    if (!pos) return;
    Left_Right(pos->left, max);
    if (*max < pos->val) *max = pos->val;
    Left_Right(pos->right, max);
}
// снизу вверх
void Down_Up (TreeNode *pos, Type *max) {
    if (!pos) return;
    Down_Up(pos->left, max);
    Down_Up(pos->right, max);
    if (*max < pos->val) *max = pos->val;
}

int main() {
    ...
    // поиск максимума на примере Up_Down
    Type max = root->val;
    Up_Down(root, &max);
    ...
}
```

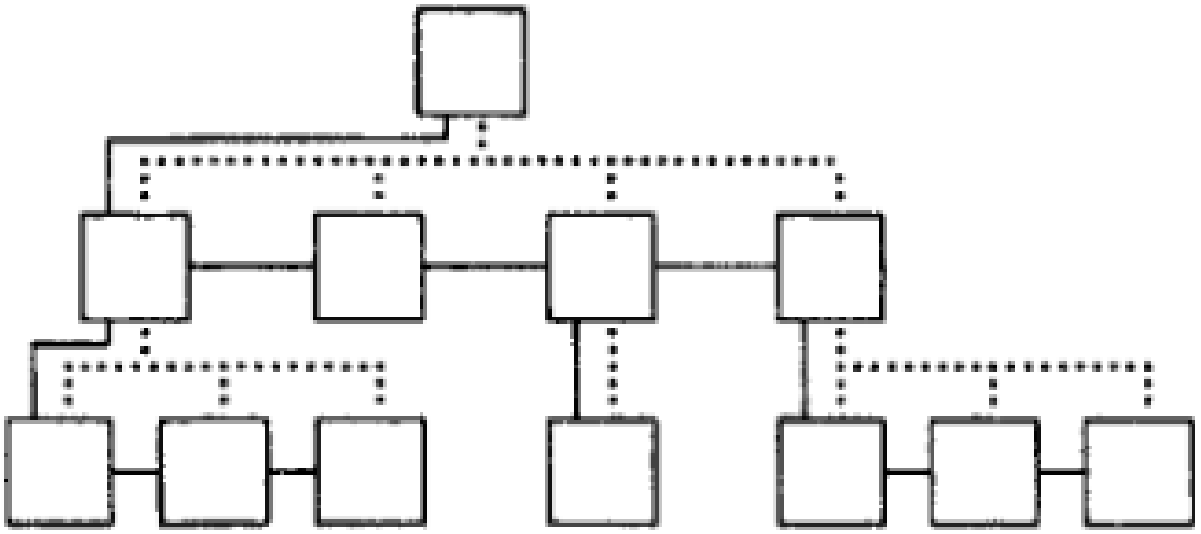
На примере рисунка:

- Сверху вниз: ABDHIECFJKLG
- Снизу вверх: HIDEBJLKFGCA
- Слева направо: HDIBEAJFKLCG

Обход произвольного дерева

Мы же не знаем сколько изначально будет потомков у текущей вершины, поэтому предлагается всех потомков текущей вершины связать в однонаправленный список в том порядке, в котором они появляются в дереве, а в родительской вершине хранить два указателя: на следующий элемент в списке братьев и на

начало списка потомков данной вершины. Можно также добавить указатель на родительскую вершину, если необходимо перемещаться назад к корню.



Здесь пунктирными линиями обозначены связи между вершинами исходного дерева, а сплошными линиями — связи, определяемые указателями, хранящимися в реализации дерева.
Для такого случая:

```
class TreeNode {
public:
    Type val;
    TreeNode *next;
    TreeNode *down;
};

class TreeNode2 { // на векторах
public:
    Type val;
    vector<TreeNode2> sons;
};

class Tree {
    TreeNode* root; // указатель на корень дерева
public:
    Tree () {root = nullptr;}
};
```

Дальше методом чайника (только слева направо смысла не имеет):

```
void Up_Down (TreeNode *pos, Type *max) {
    if (!pos) return;
    if (*max < pos->val) *max = pos->val;
    pos = pos->down;
    while (pos) {
        Up_Down(pos, max);
        pos = pos->next;
```

```
    }
}
```

Таким образом, каждый элемент имеет всего две ссылки на «последующие» элементы, поэтому в тех задачах, где не надо отслеживать принадлежность элементов определённому уровню, можно применять алгоритмы обходов бинарных деревьев.

Бинарное дерево поиска. Реализация. Процедуры поиска, добавления и удаления элемента.

Def

Бинарное дерево поиска (BST — binary search tree) — бинарное дерево, ключ любой вершины которого не меньше ключа любой вершины левого поддерева и строго меньше ключа любой вершины правого поддерева, то есть:
`pos->left->val <= pos->val < pos->right->val`

Ясен пень, что вряд ли на экзамене надо будет прям писать код, тут я имею в виду, что и так понятно что надо делать, если посмотреть на код. Мне понятно)

Поиск элемента (из Валединского)

```
class Node {
public:
    Type id;
    Node* l, *r;
};

class BST {
public:
    Node* root;
    BST () {root = nullptr;}
};

Node *RecurSearch (Node* root, Type x) { // рекурсивно
    if (!root) return 0;
    if (x == root->id) return root;
    if (x < root->id) return RecurSearch(root->l, x);
    else return RecurSearch(root->r, x);
}

Node *DirectSearch (Node* root, Type x) { // нерекурсивно
    while (root) {
        if (x == root->id) break;
        root = (x < root->id) ? root->l : root->r;
    }
```

```
    return root;
}
```

Добавление элемента (из Кошелева)

```
Node *_insert (Node* root, Type x) {
    if (v == nullptr) { // новый корень поддерева
        Node* nnode = new Node();
        nnode->id = x;
        return nnode;
    }
    if (root->id == x) {return v;}
    if (root->id < x) {
        root->r = _insert(root->r, x);
        return root;
    }
    if (root->id > x) {
        root->l = _insert(root->l, x);
        return root;
    }
}

void insert (Type x) {
    root = _insert(root, x);
}
```

Удаление элемента (из Валединского)

```
Node *DelElement (Node* root, Type x) {
    Node *pos;
    if (!root) return 0; // пустое дерево
    if (x == root->id) {
        // не более одного потомка
        if (!root->left) {
            pos = root->l;
            delete root;
            return pos;
        }
        if (!root->right) {
            pos = root->r;
            delete root;
            return pos;
        }
        // два потомка
        for (pos = root->l; pos->r; ) pos = pos->r;
        root->id = pos->id;
        // удаляем ненужную вершину после подмены
        root->l = DelElement(poot->l, pos->id);
    }
    else {
        if (x < root->id) {
            root->l = DelElement(root->l; x);
```

```
        }
        else {
            root->r = DelElement(root->r, x);
        }
    }
    return root;
}
```

Полный пример (из ДЗ Кошелева):

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

struct Node {
    int id;
    Node *l, *r;
    Node () {l = r = nullptr;}
};

struct BST {
    Node* root;
    BST () {root = nullptr;}
    ~BST() {
        _clear(root);
    }
    void _clear(Node* v) {
        if (v == nullptr) return;
        _clear(v->l);
        _clear(v->r);
        delete v;
    }
    int _find(Node* v, int x) {
        if (v == nullptr) return 0;
        if (v->id == x) return 1;
        if (x < v->id) return _find(v->l, x);
        return _find(v->r, x);
    }
    Node* _insert(Node* v, int x, bool& inserted) {
        if (v == nullptr) {
            inserted = true;
            Node* nnode = new Node();
            nnode->id = x;
            return nnode;
        }
        if (v->id == x) {
            inserted = false;
            return v;
        }
        if (v->id < x) {
            v->r = _insert(v->r, x, inserted);
        }
```

```
        if (v->id > x) {
            v->l = _insert(v->l, x, inserted);
        }
        return v;
    }
    Node* _erase(Node* v, int x, bool& removed) {
        if (v == nullptr) {
            removed = false;
            return nullptr;
        }
        if (v->id < x) {
            v->r = _erase(v->r, x, removed);
            return v;
        }
        if (v->id > x) {
            v->l = _erase(v->l, x, removed);
            return v;
        }
        if (v->id == x) {
            removed = true;
            Node* l = v->l;
            Node* r = v->r;
            delete v;
            return _merge(l, r);
        }
        return v;
    }

    Node* _merge (Node* l, Node* r) {
        if (l == nullptr) {return r;}
        if (r == nullptr) {return l;}
        l->r = _merge(l->r, r);
        return l;
    }

    int find(int x) {
        return _find(root, x);
    }

    int insert(int x) {
        bool inserted = 0;
        root = _insert(root, x, inserted);
        return inserted;
    }

    int erase(int x) {
        bool removed = 0;
        root = _erase(root, x, removed);
        return removed;
    }
};

int main() {
    int q;
    cin >> q;
```

```
    BST tree;

    for (int i = 0; i < q; ++i) {
        int n, x;
        cin >> n >> x;
        if (n == 1) {
            cout << tree.insert(x) << endl;
        } else if (n == 2) {
            cout << tree.erase(x) << endl;
        } else if (n == 3) {
            cout << tree.find(x) << endl;
        }
    }
    return 0;
}
```

АВЛ-дерево. Процедуры добавления и удаления элемента. Утверждение о длине АВЛ-дерева.

Вообще BST может выродиться в линейный список и тогда поиск будет иметь сложность $O(n)$, а так-то хорошее бинарное дерево имеет глубину $O(\log_2 n)$. Как добавлять новые элементы так, чтобы дерево росло равномерно?
Накинем базы:

Def

- Идеально сбалансированное бинарное дерево — дерево, длины любых двух ветвей которого отличаются не более чем на единицу, считая от корня.
- Длина дерева — длина его максимальной ветви.

- Сбалансированное бинарное дерево — дерево, для любой вершины которого левое и правое поддеревья отличаются не более чем на единицу.

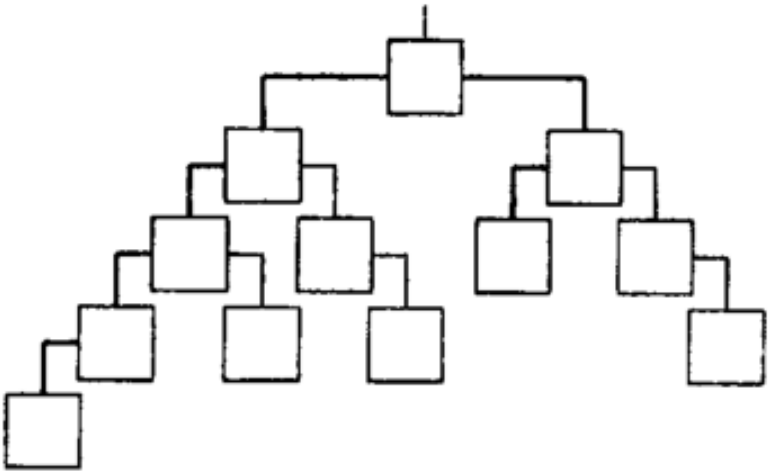


Рис. 7.7. Сбалансированное, но не идеально сбалансированное, дерево.

Тут надо сказать, что Валединский сразу решил, что сбалансированное дерево — это АВЛ-дерево.
Также надо сказать, что далее все элементы в деревьях имеют различные ключи.

Theorem

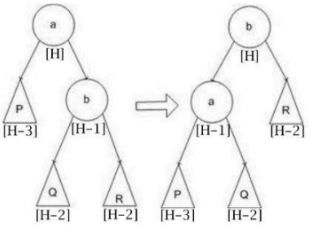
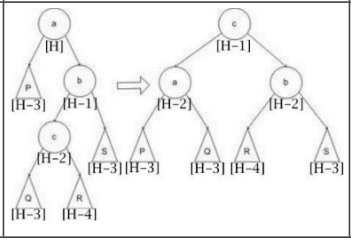
Длина сбалансированного дерева превосходит длину идеально сбалансированного дерева с таким же числом вершин не более чем в 1,5 раза.

Построим наихудший вариант дерева: в любой вершине, кроме концевых, левое поддерево длиннее правого на единицу. Пусть $N(h)$ — число элементов в худшем дереве глубины h.
Тогда $N(0) = 0$ — пустое дерево.
$N(1) = 1$ — дерево с одним элементом.
$N(k) = N(k-1) + N(k-2) + 1$ — похоже на числа Фибоначчи.
Тогда $N(h) = F_{h+2}$. При $h \to \infty:\ F_h \sim \frac{\varphi^n}{\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$.
Тогда

$$k \sim \log_2 N / \log_2 \varphi \leqslant 1,45 \log_2 N.$$

Балансировка, добавление, удаление

Нам придётся хранить показатель сбалансированности вершины `balance`. Если значение равно -1, 0, 1, то вершина сбалансирована, иначе — нет.
Пусть $h(T)$ — высота поддерева T. Если $|h(L) - h(R)| = 2$, то вершина разбалансирована, если меньше двух, то сбалансирована. Также пусть $d[i] = h(L) - h(R)$. Тогда имеются 4 типа вращения:

Тип вращения	Иллюстрация	Когда используется	Расстановка балансов
Малое левое вращение		$diff[a] = -2$ и $diff[b] = -1$ или $diff[a] = -2$ и $diff[b] = 0$.	$diff[a] = 0$ и $diff[b] = 0$ $diff[a] = -1$ и $diff[b] = 1$
Большое левое вращение		$diff[a] = -2$, $diff[b] = 1$ и $diff[c] = 1$ или $diff[a] = -2$, $diff[b] = 1$ и $diff[c] = -1$ или $diff[a] = -2$, $diff[b] = 1$ и $diff[c] = 0$.	$diff[a] = 0$, $diff[b] = -1$ и $diff[c] = 0$ $diff[a] = 1$, $diff[b] = 0$ и $diff[c] = 0$ $diff[a] = 0$, $diff[b] = 0$ и $diff[c] = 0$

Аналогично с правыми.

Ну я так понимаю тут просто картинки надо будет нарисовать и соотношения балансировки написать + идею добавления и удаления. Опишу кратко: тут всё как в BFS, только надо ещё снизу вверх проводить балансировку на выходе из рекурсий.

```
struct Node {
    int id;
    Node* l;
    Node* r;
    int h;
    Node () = default;
    Node (int x) {
        id = x;
        h = 1;
        l = r = nullptr;
    }
};

struct AVL {
    Node* root;
    AVL () {root = nullptr;}
    ~AVL () {clear(root);}

    Node* _insert (Node* v, int x) { // как в BST + балансировка
        if (v == nullptr) {
            Node* nnode = new Node();
            nnode->id = x;
            return nnode;
        }
        if (v->id == x) {return v;}
        if (v->id < x) {
            v->r = _insert(v->r, x);
        }
        if (v->id > x) {
            v->l = _insert(v->l, x);
        }
        return balance(v);
    }

    Node* _erase (Node* v, int x) {
```

```
    if (!v) return nullptr;
    if (x < v->id)
        v->l = _erase(v->l, x);
    else if (x > v->id)
        v->r = _erase(v->r, x);
    else { // нашли элемент, который хотим удалить
        if (!v->l || !v->r) { // 1 или 0 потомков
            Node* temp = v->l ? v->l : v->r;
            if (!temp) { // нет потомков
                temp = v;
                v = nullptr;
            }
            else *v = *temp; // один потомок
            delete temp;
        }
        else { // 2 потомка
            // нам надо найти элемент с как можно более близким значением к x
            // находим минимум в правом поддереве — это и будет нужный элемент
            Node* temp = _find_min(v->r);
            v->id = temp->id;
            v->r = _erase(v->r, temp->id);
        }
    }
    return balance(v);
}

int _search (Node* v, int x) {
    if (!v) return 0;
    if (x == v->id) return 1;
    return x < v->id ? _search(v->l, x) : _search(v->r, x);
}

Node* _find_min (Node* v) {
    return v->l ? _find_min(v->l) : v;
}

int height (Node* v) {
    if (v) return v->h;
    return 0;
}

void upd_height (Node* v) {
    if (v) {
        v->h = 1 + std::max(height(v->l), height(v->r));
    }
}

int diff (Node* v) {
    return v ? height(v->l) - height(v->r) : 0;
}

Node* rotate_left (Node* v) {
    if (!v || !v->r) return v;
    Node* w = v->r
    v->r = w->l;
```

```
    w->l = v;
    upd_height(v)
    upd_height(w);
    return w;
}

Node* rotate_right (Node* v) {
    if (!v || !v->r) return v;
    Node* w = v->l;
    v->l = w->r;
    w->r = v;
    upd_height(v);
    upd_height(w);
    return w;
}

Node* big_rotate_left (Node* v) {
    if (!v) return nullptr;
    v->r = rotate_right(v->r);
    return rotate_left(v);
}

Node* big_rotate_right (Node* v) {
    if (!v) return nullptr;
    v->l = rotate_left(v->l);
    return rotate_right(v);
}

Node* balance (Node* v) {
    if (!v) return nullptr;
    upd_height(v);
    if (diff(v) == -2) { // правое поддерево тяжелее
        if (diff(v->r) <= 0) { // правое поддерево правого поддерева тяжелее
            return rotate_left(v);
        }
        return big_rotate_left(v); // левое поддерево правого поддерева тяжелее
    }
    if (diff(v) == 2) { // левое поддерево тяжелее
        if (diff(v->l) >= 0) { // левое поддерево левого поддерева тяжелее
            return rotate_right(v);
        }
        return big_rotate_right(v); // правое поддерево левого поддерева тяжелее
    }
    return v;
}
void clear (Node* v) {
    if (v) {
        clear(v->l);
        clear(v->r);
        delete v;
    }
}
//-----------------------------------------
void insert (int x) {root = _insert(root, x);}
void erase (int x) {root = _erase(root, x);}
```

```
    int search (int x) {return _search(root, x);}
};
```

Битовое множество. Поиск, добавление, удаление; операции объединения, пересечения, инвертирования.

Для задач, в которых не важна упорядоченность, удобно пользоваться математическим понятием множества. Над множествами можно выполнять следующие операции: добавлять / удалять / искать элемент, а также объединять / пересекать / инвертировать сами множества.
Какой хотим функционал:

1. Создать пустое множество
2. Уничтожить множество
3. Добавить элемент
4. Удалить данный элемент
5. Принадлежит ли элемент множеству?
6. Взять элемент из множества
7. Очистить множество
8. Множество пусто?
9. Итератор по элементам множества.

Question

Реализовать подмножество множества целых чисел в диапазоне от 0 до $N_{max} - 1$.

Надо выделить массив целых чисел, каждое из которых рассматривается как набор из фиксированного количества бит. Определение местоположения конкретного бита сводится к вычислению порядкового номера элемента базового массива чисел и определении номера нужного нам бита в этом элементе.

`unsigned long int` занимает 4 байта или 32 бита, поэтому за один такт процессор будет делать сразу 32 вычисления, отчего скорость выполнения множественных операций вырастет в 32 раза.

Если k — интересующий нас элемент числового множества, то номером элемента в базовом числовом массиве будет целая часть от деления на 32 (`cell_num(k)`), а номером бита в этом элементе — остаток от деления k на 32.

```
#include <string.h>
// делим x на 32:
#define cell_num(x)    ((x) >> 5) // номер ячейки (5 потому что ul)
// остаток от деления на 32:
#define cell_bit(x)    ((x) & 0x1FL) // номер бита в ячейке
using ul = unsigned long;

class BitSet {
private:
```

```
    ul* mem; // указатель на массив базовых ячеек
    ul n;
public:
    BitSet (ul maxsize) {
        n = cell_num(maxsize) + 1; // количество ячеек
        mem = new ul[n];
        memset(mem, 0, n * sizeof(ul));
    }
    ~BitSet () {if (mem) delete [] mem;}
    // (1 << cell_bit(x)) означает сдвиг на номер бита, который
    // отвечает за данный x
    bool Find (unsigned long x) {
        return (mem[cell_num(x)] & (1 << cell_bit(x))) != 0;
    }
    void Put (ul x) {
        *(mem + cell_num(x)) |= (1 << cell_bit(x));
    }
    void Del (ul x) {
        *(mem + cell_num(x)) &= ~(1 << cell_bit(x));
    }
    BitSet& Union (BitSet& oth, ul n) {
        for (ul i = 0; i < n; i++) {
            mem[i] |= oth.mem[i];
        }
        return *this;
    }
    BitSet& Intersection (BitSet& oth, ul n) {
        for (ul i = 0; i < n; i++) {
            mem[i] &= oth.mem[i];
        }
        return *this;
    }
    BitSet& Diff (BitSet& oth, ul n) {
        for (ul i = 0; i < n; i++) {
            mem[i] &= ~oth.mem[i];
        }
        return *this;
    }
    BitSet& Inv () {
        for (ul i = 0; i < n; i++) {
            mem[i] = ~mem[i];
        }
        return *this;
    }
};
// здесь по-хорошему при побитовом сдвиге надо написать
// вместо единицы 1UL или закастовать (ul)1
// и надо везде проверки границ, но это и так понятно.
```

Асимптотика на множественных операциях здесь улучшается с $O(n)$ для массивов до $O(n/32)$ для битового множества (да, константы можно выносить в O-нотации, но тут это важно). А ещё в STL уже давно написано то же самое и называется `bitset`.

Хеширование. Примеры хеш-функций. Хеш-множество на основе массива списков.

Задача состоит в обработке небольшого подмножества некоторого необозримого множества. Хочется разделить исходное множество на несколько классов эквивалентности относительно некоторой функции. Более формально:

$$f : M \to \{0, 1, \ldots, p-1\}$$
$$N \subset M$$
$$M_k = \{x \in M : f(x) = k\}$$

Функция f разбивает множество M на p классов эквивалентности M_k. Тогда при работе с подмножеством N достаточно вычислить значение функции на элементе из этого подмножества и работать с пересечением

$$N_k = M_k \cap N.$$

Если удачно выбрать функцию, то количество элементов в N_k будет в среднем в p раз меньше, чем во всём множестве N.

Реализация хеш-множества на основе массива списков

(также ещё называется хеш-таблицей с закрытой адресацией)
Каждая ячейка хеш-таблицы — это указатель на некоторый список. Элементы, которые имеют один и тот же хеш, попадают в этот список.

- Вставка: Вычисляется хеш элемента, находится нужная ячейка, элемент добавляется в список из этой ячейки.
- Поиск: Вычисляется хеш элемента, в нужном списке как-то ищется элемент.
- Удаление: Вычисляется хеш элемента, в нужном списке удаляется элемент, если он есть.

Вместо списков можно использовать любую другую структуру данных: хотите бинарное дерево, хотите битовое множество, хотите динамический массив — короче тут под конкретную задачу можно подгонять производительность.

Затраты памяти

Если резервировать место для хранения n элементов с размером a байт, а хеш-функция принимает p значений, то для хранения множества с помощью списков потребуется

$$(a + b)n + pb$$

байт, где b — количество байт в представлении адреса.
Для дерева, соответственно:

$$(a + 2b)n + pb$$

Здесь второе слагаемое отвечает за, так скажем, заголовок хеш-таблицы, а вторая часть за каждый класс эквивалентности.

Хеш-множество по методу последовательных (линейных) проб. Оценка среднего количества проб при добавлении элемента.

Def

Коллизия — это совпадение значений хеш-функции для различных элементов.

Можно ли как-то без ссылочной структуры оформить хеш-таблицу? Такой способ называется хеш-таблицей с открытой адресацией (если что, так на лекциях не говорилось, только на семинарах). Короче, берём массив длиннее, чем количество принимаемых хеш-функцией значений.

Добавление: $h[k] = x$, где $k = f(x)$. Если произошла коллизия, то пусть $f(x) = f(t)$, тогда следующей ячейкой будет ячейка с номером $f(t) + g(t)$, где $g(t)$ можно взять какой-нибудь константной функцией ($g(t) = 1$)). Если снова попали в занятую ячейку, то смотрим $f(t) + 2g(t)$ и т.д. Складывать надо, конечно, по модулю длины массива.

Поиск: аналогичный добавлению алгоритм.

Удаление: просто так очистить ячейку нельзя: прервётся цепочка для поиска, поэтому обычно создаётся дополнительный массив, который имеет три значения: ячейка пустая, ячейка занята и ячейка была занята. Если ячейка была занята, то на добавлении её можно перезаписать, а для поиска просто пропустить данную итерацию.

Theorem

Пусть в методе проб коэффициент заполнения хеш-таблицы есть $s = \frac{m}{n}$, где n — размер таблицы, m — количество занятых ячеек (мощность рабочего множества). Пусть хеш-функция обеспечивает равномерное распределение элементов рабочего множества по позициям в таблице. Тогда среднее число проб при работе с множеством (при поиске, добавлении, удалении) составляет $\frac{1}{1-s}$.

Доказательство:
Вероятность попасть на занятую ячейку есть $s_1 = s$, а на незанятую — $(1 - s_1)$. При второй пробе эти вероятности есть $s_2 = \frac{m-1}{n-1}$ и $(1 - s_2)$.
Для третьей пробы $s_3 = \frac{m-2}{n-2}$ и $(1 - s_3)$.
Вероятность сделать ровно k проб есть:

$$q_k = s_1 s_2 \dots s_{k-1}(1 - s_k)$$

— тут $k - 1$ неудачных и последняя удачная (неважно сколько раз ты падал, важно сколько раз ты поднялся после падения уу)
блин, серьёзно? а как без матожидания? ну ладно, тут вроде очев.
Среднее число проб — это математическое ожидание

$$\sum_{k=1}^{m} k q_k = 1(1 - s_1) + 2s_1(1 - s_2) + 3s_1 s_2(1 - s_3) + \dots =$$

$$= 1 + s_1 + s_1 s_2 + s_1 s_2 s_3 + \dots <$$

$$< 1 + s_1 + s_1^2 + \dots \leqslant \frac{1}{1 - s}.$$

Минимальная совершенная хеш-функция. Определение. Построение.

Задача состоит в том, что у нас есть m слов (ключей). Надо построить такую хеш-функцию, которая не даёт коллизий на этих словах.

Пусть $\mathcal{M}$ — множество всех возможных ключей, $M \subseteq \mathcal{M}$ — подмножество заданных фиксированных ключей, то есть $|M| = m$.

Def

- Хеш-функция $h(x)$ называется *совершенной*, если для любых $x_1, x_2 \in M$:

$$x_1 \neq x_2 \Longrightarrow h(x_1) \neq h(x_2).$$

- Совершенная хеш-функция $h(x)$ называется *минимальной*, если

$$h : M \to \{0, \ldots, m-1\}.$$

- Пусть на множестве ключей введено отношение порядка «<». Совершенная хеш-функция $h(x)$ *сохраняет порядок*, если для любых $x_1, x_2 \in M$:

$$x_1 < x_2 \Longrightarrow h(x_1) < h(x_2).$$

щащащаща пойду в разнос, скриньте!