

Это репозиторий курс "Структуры данных" для КЛШ-24.

План по занятиям:

1. [Урок 1](#) - знакомство с ООП.
2. [Урок 2](#) - многофайловые программы, простая реализация Vector.
3. [Урок 3](#) - доработка Vector.
4. [Урок 4](#) - шаблоны, наследование и структуры на основе Vector.
5. [Урок 5](#) - методы монеток и потенциалов.
6. [Урок 6](#) - связный список и итераторы.
7. [Урок 7](#) - разные варианты дека и интерфейс для бинарных деревьев поиска.
8. [Урок 8](#) - rbst, больше наследования.
9. [Урок 9](#) - декартово дерево, обычное и по неявному ключу.
10. [Урок 10](#) - Splay Tree: амортизированный логарифм в бинарном дереве поиска.
11. [Урок 11](#) - 2-3 дерево: строгий логарифм в дереве поиска.
12. [Урок 12](#) - пары и set, map, multi- с итераторами.

Урок 1: Знакомство с ООП.

[К главному описанию](#)

Краткий план

1. Знакомство с ООП (классы, объекты, конструкторы).
2. Перегрузка операторов.
3. Пример: класс для дробей.

Мотивация

Идея того, что всё вокруг классы и объекты очень жизненна: например понятно, что многие машины имеют примерно одинаковый функционал и поэтому их можно описать какими-то общими параметрами (класс), но при этом каждая машина имеет что-то своё (цвет, количество бензина в баке, имя хозяина) - поэтому нужны экземпляры классов (объекты).

Если же говорить о программировании, то классом может быть что-то и более абстрактное, например сегодня мы займёмся классом дробей.

Модификаторы доступа

Некоторые данные внутри класса хотелось бы защищать, чтобы никто посторонний к ним не имел доступа - для решения этой проблемы придумали модификаторы доступа:

- `public` - эти данные известны и снаружи класса и внутри;
- `private` - эти данные известны только внутри класса;
- `protected` - эти данные известны внутри класса и в классах наследниках (про наследование поговорим позже);
- `friend` - не является модификатором доступа, но с помощью этого ключевого слова можно разрешать доступ желаемым функциям и классам к `private` полям;

Для дроби хочется чтобы числитель и знаменатель были `private` так как негоже кому попало их менять.

Конструкторы

Конструкторы вызываются в момент создания объекта класса и нужны чтобы присвоить полям объекта какие-то значения (по умолчанию или переданные в конструктор).

Перегрузка операторов

Не во всех языках программирования существуют перегрузки, но в C++ они реализованы так:

- Пусть мы хотим перегрузить оператор `+`, чтобы имело смысл выражение `a + b`, где `a` и `b` дроби.
- Тогда внутри класса определяем:

```
Frac operator+(Frac other);
```

Где `operator+` - название перегружаемой функции, а `other` - правый аргумент суммы.

- Внутри функция должна работать так, чтобы вычислить значение выражения `(*this) + other`.

Перегрузка операторов ввода-вывода

Как было описано выше, когда мы пишем `l + r`, то фактически вызывается `l.operator+(r)`.

Теперь представим, что мы хотим вывести дробь, то есть написать `cout << f`, тогда должен вызваться `cout.operator<<(f)`. Но мы ведь не можем залезть внутрь класса для `cout` чтобы определить там функцию вывода! В таких случаях перегрузку выполняют в виде свободной функции принимающей два аргумента, для нашего примера это будет:

```
ostream& operator<<(ostream& os, Frac f);
```

Так как `ostream` это название класса для `cout`.

Кроме того часто операторы ввода-вывода приходится делать `friend`, так как они меняют значения полей класса, как раз такой пример и есть в коде.

Константность и ссылки

В перегрузке оператора вывода мы зачем-то принимали что-то с `&` и возвращали что-то с таким же непонятным типом. На самом деле символ амперсанда всего лишь означает, что мы не хотим копировать объект при передаче его в функцию.

То есть обычно при вызове функции передаваемые в неё аргументы копируются и их можно редактировать внутри функции независимо от копии вне функции. Но с операторами ввода-вывода копирование было бы чем-то странным, ведь консоль для ввода-вывода всего одна и не хорошо создавать копии, поэтому используется `&`.

Кроме того иногда объект у вас слишком большой чтобы его копировать и тогда логично использовать `&`. Если при этом вы не хотите внутри функции как-то объект поменять, то стоит использовать `const` чтобы показать неизменяемость объекта:

```
ostream& operator<<(ostream& os, Frac const& f);
```

Урок 2: Многофайловые программы, простая реализация Vector.

[К главному описанию](#)

Краткий план

1. Необходимость многофайловых программ и как это реализовать на C++.
2. Пространства имён.
3. Простая версия класса Vector.

Мотивация

Когда кода становится много, то хочется разбивать его на несколько файлов. Причём разбиение стоит делать не абы как, а согласно какой-то логике, например один файл на один класс (как это делается в Java).

В C++ же разбиение происходит ещё более тонко: на файлы с кодом (.cpp) и заголовочные (.h). В заголовочном файле хранится только описание классов и функций (объявление) тогда как сама реализация делается в файлах с кодом (определение).

Когда вы пишите `#include <smth>`, то компилятор подставляет вместо этой строки весь код из smth. Но также у компилятора есть правило одного определения (one definition rule), согласно которому объявлений может быть несколько, а определение должно быть только одно. Именно поэтому в заголовочные файлы выносятся объявления и они подключаются с помощью `#include`. А файлы с кодом никуда не подключаются, поэтому определение получается ровно одно.

Пространства имён

Так случается, что названия чего-то начинают пересекаться (например вы захотели переделать встроенную функцию `sqrt`, так как иногда она бывает неточна). В таких случаях прибегают к namespace (пространство имён). Их цель в том, чтобы во-первых сгруппировать функции и классы (например пространство имён `std::` содержит все встроенные функции), а во-вторых для того, чтобы иметь возможность сделать функцию с таким же названием, как уже существующая.

Как устроен Vector

Нам нужно менять размер массива, для этого будем пользоваться указателями и операторами `new[]` и `delete[]`. Эти операторы позволяют выделять и удалять память под массив. В текущем варианте Vector будет хранить `int` и устроен будет так:

- Внутри себя хранит `int* data` - указатель на массив, `size_t cur_size`, `cur_capacity` - текущий размер и размер выделенного массива (все поля `private`).
- Если хотим добавить элемент и `cur_size = cur_capacity`, то выделяем массив

в два раза большего размера и всё туда копируем, иначе добавляем в конец выделенного места и сдвигаем `cur_size`.

- Ещё будем поддерживать операции получения последнего элемента и вывод всего контейнера.

Урок 3: Доработка Vector.

[К главному описанию](#)

Краткий план

1. Аккуратная работа с памятью (копирование).
2. Конструктор от размера.
3. Больше методов: получение размера, удаление из конца; константные методы.
4. Сравнение объектов: оператор `<=>`.
5. Обращение по индексу и итераторы: оператор `[]`, методы `begin` и `end`.

Мотивация

Казалось бы, мы всё уже написали и оно отлично работает. Но вот нет, мы сегодня рассмотрим такие примеры, по которым будет видно наличие ошибок.

Копирование

Рассмотрим вот такой код:

```
Vector v1;
v1.push_back(2);
v1.push_back(3);
v1.push_back(5);
v1.push_back(7);
cout << v1; // 2 3 5 7
Vector v2 = v1;
cout << "v1: " << v1; // 2 3 5 7
cout << "v2: " << v2; // 2 3 5 7
v1.push_back(11);
v2.push_back(13);
cout << "v1: " << v1; // 2 3 5 7 13
cout << "v2: " << v2; // 2 3 5 7 13
```

Хмм, кажется что-то не так!

Придумано много разных способов обойти такие ошибки, мы рассмотрим Swap and Copy. В нём нужно сделать функции `swap` и `copy` и потом уже аккуратно реализовать остальные функции через них:

```
void copy(Vector const& v);
void swap(Vector& v);
Vector(Vector const& v); // конструктор копирования, выр
Vector& operator=(Vector const& v); // оператор присваивания, выраж
```

Больше конструкторов

Представим, что мы хотим сделать новый конструктор от количества элементов, в

таком случае нам нужен:

```
Vector(size_t size = 0);
```

Где параметр по умолчанию нужно указывать в заголовочном файле, но не нужно указывать в файле с кодом.

Но теперь внимание, "незаметная" разница, у нас будут работать оба варианта:

```
Vector v3(2);           // круглые скобки
Vector v4 = 5;          // присваивание
```

Второй синтаксис какой-то странный, лучше его явно запретить, для этого придумали слово `explicit` (запретить неявные преобразования):

```
explicit Vector(size_t size = 0);
```

Больше методов

Предположим, что нам понадобились методы, говорящие размер вектора, тогда понятно, что хочется написать:

```
size_t Vector::size() {
    return cur_size;
}
```

Но, представим, что кто-то захотел написать такой код:

```
const Vector v3(v2);
cout << "v3: " << v3;
cout << v3.size() << "\n";
```

В целом все хорошо, понятно чего хотел автор и видно, что константу изменять он не собирался. Но такой код не скомпилируется, ведь метод `size` не помечен как константный. Чтобы это работало нужно написать:

```
size_t Vector::size() const {
    return cur_size;
}
```

Аналогично константными должны быть `get_back` и `capacity`.

Теперь реализуем операцию удаления из конца вектора: удаляем (уменьшаем `cur_size`) и если получилось, что `cur_size * 4 < cur_capacity`, то выделяем заново память размера `cur_size * 2` и копируем туда данные. Почему константы выбраны именно так, мы поговорим на следующем занятии.

Сравнение объектов

В старых стандартах для определения сравнения между объектами, нужно было перегружать все операторы вручную (на самом деле можно написать только оператор `<` и остальные выразить через него, можете на досуге подумать как именно). В новом же стандарте появился оператор `<=`, и можно определить его, оператор `==` и остальные операторы получатся сами.

Для этого определяем операторы:

```
std::strong_ordering operator<=>(Vector const& other) const;
bool operator==(Vector const& other) const;
```

Внутри первого можно пользоваться операторами `<=>` для встроенных типов, или возвращать что-то из списка: `less` (левое меньше), `greater` (левое больше), `equal` (равны). Математики придумали не только привычные по жизни порядки `std::strong_ordering`, но и некоторые другие, о них можете почитать сами в свободное время.

Функцию равенства можно определять как-то по умному (именно поэтому стандарт заставляет определять её отдельно), но можно и через `<=>`:

```
bool Vector::operator==(Vector const& other) const {
    return (*this <=> other) == std::strong_ordering::equal;
}
```

Обращение по индексу и итераторы

Ну все же уже давно хотят получать доступ к элементу вектора по произвольному индексу, давайте научимся!

```
int& operator[](int pos);                                // чтобы смоchь присвоить зн
int const& operator[](int pos) const;                      // чтобы получить значение
```

Познакомимся со способом сокращения не удобных названий типов:

```
using iterator = int*;                                  // теперь вместо int* можно
using const_iterator = const int*;                      // теперь вместо const int*
```

Ну и теперь мы готовы определить функции `begin` и `end`, чтобы по нашему вектору можно было делать `range-based for`:

```
iterator begin();                                     // возвращаем data
iterator end();                                       // возвращаем data + cur_si
const_iterator begin() const;
const_iterator end() const;
```

Теперь оператор вывода можно реализовать без использования `friend`!

Урок 4: Шаблоны, наследование и структуры на основе Vector.

[К главному описанию](#)

Краткий план

1. Зачем нужны шаблоны и как они разбиваются на файлы.
2. Stack на основе Vector: наследование.
3. Queue с помощью двух Stack как полей класса.

Мотивация

Вроде бы мы справились сделать Vector, хранящий int. А что если нам понадобится хранить string, float, double, long long, char - под всё это заново класс писать что-ли?! К счастью нет, и в C++ для этих целей придуманы шаблоны.

Как работать с шаблонами

В объявлении класс (.h файл) сразу над классом добавляем и над функцией вывода:

```
template<typename T>
```

И внутри объявления класс меняем используем T как тип, хранимый в векторе.

В реализации класса (.cpp файл) перед каждым методом нужно добавить:

```
template<typename T>
```

И заменить Vector на Vector<T> везде, где имеется ввиду тип данных (то есть все места, кроме названия конструктора и деструктора).

И наконец переименовываем vector.cpp в vector.tpp и добавляем в конце vector.h строку:

```
#include "vector.tpp"
```

Подключать файл в заголовочный нужно так как шаблонный класс должен подключаться в другие файлы целиком (вместе с реализацией всех методов), а поменяли расширение мы для того, чтобы отличать файл с настоящим кодом от файла с шаблоном кода (у MSWord сделано также: для обычных файлов и файлов с шаблоном стиле используются разные расширения).

Стек

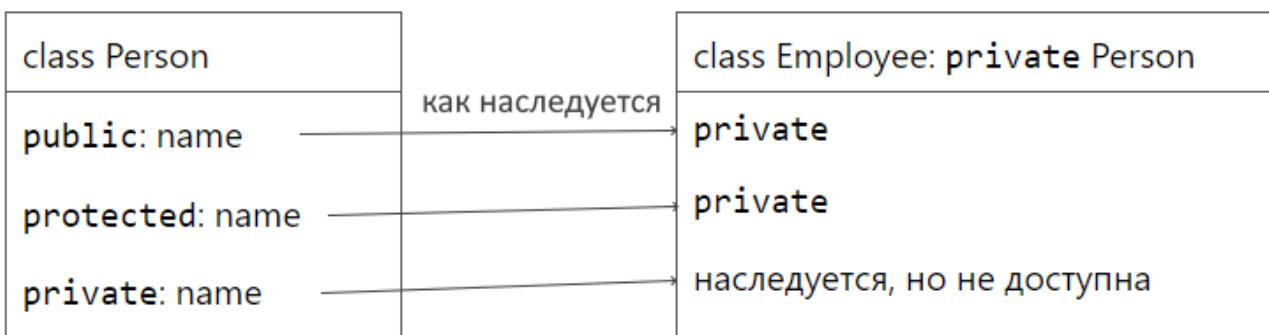
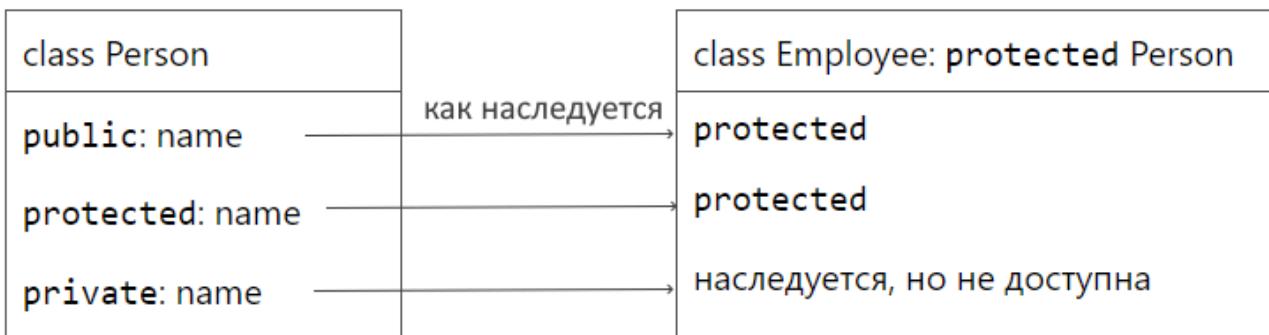
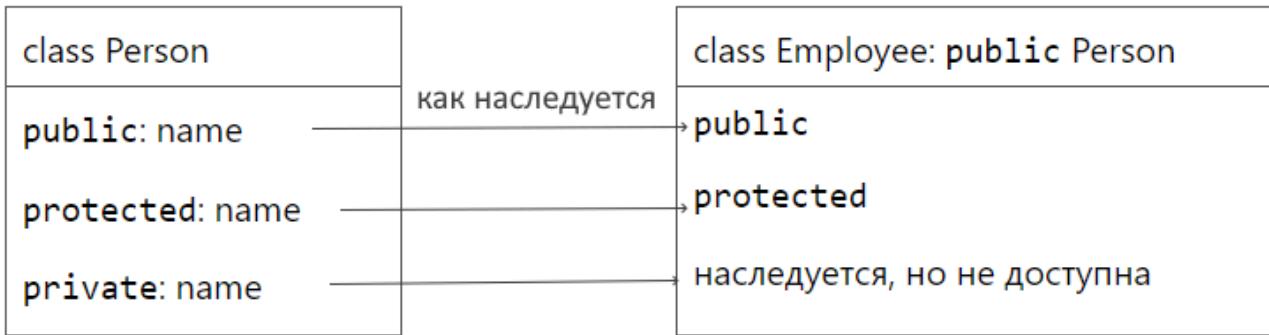
Стек - это такая структура данных, которая позволяет только добавлять и удалять элемент из конца, обращаться к последнему элементу. По сути это только часть функционала, которую мы уже сделали в Vector, и зачем мы вообще останавливаемся на стеке? Это может показаться странным, но на практике люди любят искусственно

ограничить себе возможности: если вы хотите использовать только функционал стека, то хочется использовать отдельный класс для стека, а не использовать вектор - так код упрощается для чтения и потенциально может стать эффективнее от использования структуры данных умеющей меньше. Но хоть структура и простая, но мы постараемся максимально усложнить себе жизни!

Мы уже познакомились с инкапсуляцией, когда рассматривали модификаторы доступа. Теперь настало время рассмотреть наследование. Представим, что мы хотим в классе Child сделать примерно тоже самое, что делали в классе Parent но с какими-то отличиями. Тогда мы можем унаследовать Child от Parent и поменять только нужные функции. Это хорошая практика, так как позволяет уменьшить объём кода в отличие от полного копирования всех методов. В C++ придумали несколько видов наследования:

```
class Child : public Parent;
class Child : protected Parent;
class Child : private Parent;
```

Если не указать модификатор, то по умолчанию используется `private` (а для `struct` используется `public`). Общая схема того, как наследуются поля:



Когда в Child мы хотим использовать метод method унаследованный из Parent, то его можно вызвать через Parent::method.

Для стека мы хотим приватно унаследоваться от вектора и переименовать методы в push, top, pop. Деструктор у нас ничего делать не будет, так как C++ сам вызывает деструктор родительского класса. Обращение по индексу, методы begin и end, операцию вывода тоже можно определить, но они не очень осмыслены к структуре с возможностью доступа только к последнему элементу.

Ещё из приятных фактов стоит отметить, что конструктор копирования и оператор присваивания можно определять, а довериться в их создании компилятору. Компилятор справляется сам сгенерировать эти методы, если в самом классе не хранится что-то сложное (в нашем случае в Vector хранилась data и мы следили чтобы она правильно копировалась, в Stack же ничего не хранится и компилятор сгенерирует функции через копирования в Vector).

Но если очень хочется, то можно определить конструктор и вызывать в нём родительский конструктор, например вот такой код допустим:

```
template<typename T>
Stack<T>::Stack() : Vector<T>::Vector() {} // Вызываем родительски
```

Очередь

Теперь мы перейдём к такой структуре, как очередь (Queue). Функционал у очереди тоже довольно маленький: можно добавить элемент в конец и взять из начала. И очередь можно реализовать с помощью двух стеков `left` и `right`

- При добавлении элемента добавляем его в `right`.
- Если мы удаляем элемент из `left` пустой, то перекладываем все элементы из `right` в `left` и переходим к следующему шагу.
- Если мы удаляем элемент из `left` не пустой, то удаляем из `left`.

Понятно, что очередь не нужно не от чего наследовать, так как левая и правая части достаточно равнозначны. Поэтому в очереди будем хранить два стека. Все методы определяются по описанию выше, разве что вынести перекладывание всех элементов из `right` в `left` можно в отдельную функцию `move_stack` (она должна быть `private`), чтобы кто попало её не вызывал.

Кроме того стоит отметить, что метод `front` пришлось сделать неконстантным, так как если `left` пуст, то мы не можем достать элемент снизу `right`. В целом это можно обойти, если никогда не оставлять `left` пустым, или добавить метод в стек. То есть ограничения на некоторые методы бывают вызваны внутренним устройством структуры и это нормально.

Урок 5: Методы монеток и потенциалов.

[К главному описанию](#)

Краткий план

1. Сложность алгоритмов.
2. Метод монеток.
3. Применение монеток: Stack и Queue через элементарные операции с Vector.
4. Метод потенциалов.
5. Применение потенциалов: операции с Vector.

Мотивация

В прошлые разы мы что-то написали, но почему именно это? Почему Vector нужно расширять в 2 раза, а сжимать в 2 раза только когда можем даже в 4? Сегодня мы разберёмся с этими вопросами и обоснуем, что написанные нами структуры достаточно хороши.

Сложность алгоритмов

Программисты люди ленивые и поэтому хотят оценивать только порядок времени работы. Например t , $2t$, $0.5t$ и $100t$ для них одинаковы (потому что можно найти компьютер чуть помощнее или подождать подольше), а вот $3t^2$ это уже существенно больше, так как при росте t придётся брать очень много новых компьютеров.

Формальное определение. Мы говорим, что функция $t(n)$ имеет сложность $O(f(n))$, если: $\exists C > 0 \forall n: \exists N: \forall n \geq N \quad t(n) \leq C f(n)$

Метод монеток

Пусть мы хотим оценить время работы какой-то структуры данных. Тогда будем давать каждому элементу этой структуры какое-то фиксированное число монеток и в моменты работы с этим элементом будем эти монетки забирать. Тогда в сумме мы сделаем количество операций не большее, чем количество выданных монеток, а следовательно на один элемент потратим количество операций равное количеству монеток. Так мы можем доказать, что у какой-то структуры данных линейная сложность.

Применение монеток

Будем считать, что все операции с Vector работают за $O(1)$, то есть посчитаем сложность работы стека и очереди в терминах операций с вектором. Потом, когда мы узнаем, за сколько выполняются операции с Vector, то сможем более детально узнать и сложности стека и очереди, если захотим.

Стек. Здесь всё просто: при добавлении элемента даём ему одну монетку, а при удалении элемента этой монеткой оплачиваем операцию его удаления. Итого на n запросов мы потратим $O(n)$ монеток, значит сложность на все запросы будет $O(n)$,

а на один - $O(1)$. Такую же оценку можно получить и из других рассуждений: пусть мы сделали n добавлений и n удалений, тогда каждый элемент один раз добавился и один раз удалился, значит в сумме сделали $O(n)$ операций и $O(1)$ на одну операцию.

Очередь. Здесь уже придётся поступить чуть хитрее: каждому элементу при добавлении даём по три монетки. В момент перекладывания элемента из одного стека в другой мы забираем две монетки (так как это удаление из правого стека и добавление в левый стек - итого две операции), а ещё одну монетку забираем в момент удаления элемента из левого стека. Итого потратили $O(n)$ монет на n запросов, значит один запрос делаем за $O(1)$. И как и в случае со стеком можно было просто посмотреть и заметить, что один элемент два раза добавляется в стеки и два раза удаляется, а отсюда получится та же сложность.

Метод потенциалов

Пусть мы ввели "потенциал" Φ_i - какая-то величина, характеризующая нашу структуру в момент времени i (Φ_0 - потенциал до всех запросов, Φ_i - потенциал после обработки i запросов). И пусть произошло q запросов к структуре, каждый запрос занял время t_1, t_2, \dots, t_q . Тогда алгоритм работает амортизировано за $O(f)$, если:

- $t_i + \Phi_i - \Phi_{i-1} \leq f$, где $1 \leq i \leq q$;
- $0 \leq \Phi_i \leq q \cdot f$, где $0 \leq i \leq q$;

Доказательство. Сложим все q неравенств первого типа, тогда получим: $\left(\sum_{i=1}^q t_i\right) + \Phi_m - \Phi_0 \leq q \cdot f \Rightarrow \sum_{i=1}^q t_i \leq 2qf = q \cdot O(f)$

Применение потенциалов

Хотим доказать, что операции с `Vector` работают амортизировано за $O(1)$. Рассмотрим потенциал $\Phi_i = |2sz_i - cap_i|$, где sz_i - величина `size` в момент времени i , а `cap_i` - величина `capacity` в момент времени i (обе величины перед запросом i). Тогда проверяем оба условия:

- Второе условие: $sz_i \leq cap_i \leq 4 \cdot sz_i \Rightarrow 2sz_i - cap_i \geq 2sz_i - 4sz_i \geq -2sz_i \Rightarrow 0 \leq \Phi_i = |2sz_i - cap_i| \leq 2sz_i \leq 2i \leq 3q$;
- Добавление без релокации: $t_i = 1, \Phi_i - \Phi_{i-1} \leq 2 \Rightarrow t_i + \Phi_i - \Phi_{i-1} \leq 3$;
- Удаление без релокации: $t_i = 1, \Phi_i - \Phi_{i-1} \leq 2 \Rightarrow t_i + \Phi_i - \Phi_{i-1} \leq 3$;
- Расширение вектора: $t_i = sz_{i-1}, \Phi_i - \Phi_{i-1} = |2sz_{i-1} - 2sz_{i-1}| = 0 \leq 3$;
- Сжатие вектора: $t_i = sz_i, \Phi_i - \Phi_{i-1} = |2sz_i - 2sz_i| = 0 \leq 3$;

Итого показали, что `Vector` работает амортизировано за $O(3) = O(1)$.

Из этого следует, что во-первых наша реализация хорошая (можно посмотреть какие

оценки получатся, если например уменьшать размер при $2\text{size} < \text{capacity}$), а во-вторых, при оценки сложности стека и очереди мы принимали время работы операций с вектором за $O(1)$, а теперь знаем, что это было честно.

Урок 6: Связный список и итераторы.

[К главному описанию](#)

Краткий план

1. Кто такие итераторы.
2. Односвязный и двусвязный списки.

Мотивация

Мы уже реализовали почти все линейные структуры данных, которые есть в стандартной библиотеке, теперь дело осталось за малым.

Итераторы

Помнится мы реализовали у `Vector` методы `begin` и `end`, которые возвращали указатели. Но бывают более сложные структуры, у которых переход к следующему элементу не такой тривиальный, как у указателей. Для решения этой проблемы придумали итераторы, которые должны уметь:

- Делать операцию `*` - разыменование итератора, нужно вернуть элемент контейнера.
- Делать операцию `++` - переход итератора вперёд.
- Опционально делать операцию `--` - переход итератора назад.
- Делать операцию сравнения `==` - чтобы проверять, дошёл ли итератор до конца.
- Иногда оператор `<` также имеет смысл.
- Определить все псевдонимы (не обязательно, без них тоже будет работать):

```
using iterator_category = std::forward_iterator_tag;
// также бывают: input_iterator_tag, output_iterator_tag,
// bidirectional_iterator_tag, random_access_iterator_tag, contiguous_i
using value_type = T;
using element_type = T;
using pointer = T*;
using reference = T&;
```

Связные списки

Концепция связных списков достаточно простая: в каждом элементе будет хранить само значение и ссылку на соседний элемент. Если мы храним ссылку только на одного соседа (следующего) - то это односвязный список, `forward_list` в C++; а если храним ссылку и на следующий элемент и на предыдущий - то это двусвязный список, `list` в C++.

Для связных списков нам понадобится определить по три класса: для элемента, итератора и собственно самого связного списка. Приведём здесь класс для элемента и итератора односвязного списка, так как раньше мы такого не писали:

```

template<typename T> class ForwardListIterator;
template<typename T> class ForwardList;

template<typename T>
class ForwardListItem {
private:
    T value;
    ForwardListItem<T> *next;
public:
    ForwardListItem(T value, ForwardListItem<T> *next) : value(value),
        friend class ForwardListIterator<T>;
        friend class ForwardList<T>;
};

template<typename T>
class ForwardListIterator {
private:
    ForwardListItem<T> *it;
public:
    ForwardListIterator(ForwardListItem<T> *it = nullptr) : it(it) {}

    T& operator*() {
        return it -> value;
    }

    ForwardListIterator<T> operator++() {
        it = it -> next;
        return *this;
    }

    bool operator==(ForwardListIterator<T> const& other) const {
        if (it == nullptr && other.it == nullptr) return true;
        if (it == nullptr || other.it == nullptr) return false;
        return it -> next == other.it -> next;
    }

    friend class ForwardList<T>;
};

```

Сам же односвязный список поддерживает операции вставки и удаления элемента из начала, и по итератору (точнее вставляет или удаляет элемент после переданного итератора).

В двусвязном списке в каждом элементе есть указатели на следующий и предыдущий, поэтому вставку и удаление можно делать с обоих концов, удалять элементы можно по самому итератору, а вставлять как до, так и после переданного итератора. Кроме этого итератор для двусвязного списка должен уметь делать --.

Замечание: реализация двусвязного списка может оказаться сложнее чем думается :worried:

Урок 7: Разные варианты дека и интерфейс для бинарных деревьев поиска.

[К главному описанию](#)

Краткий план

1. Дек на двусвязном списке.
2. Дек на трёх стеках.
3. Дек на закольцованным буфере.
4. Бинарные деревья поиска.

Мотивация

Кажется, что последняя оставшаяся линейная структура - это дек, его мы реализуем сегодня. И дальше уже перейдём к деревьям.

Дек на двусвязном списке

Дек - это такая структура данных, которая умеют добавлять, брать и удалять элемент из своих начала и конца.

В прошлый раз мы уже сделали двусвязный список, у которого очевидно хорошая асимптотика, а следовательно если дек сделать на двусвязном списке, тоже получим асимптотику $O(1)$ на запрос. Как и в случае со стеком, приватно наследуемся и меняем названия для методов.

Дек на трёх стеках

Если мы вспомним реализацию очереди на двух стеках, то там была левая и правая половина и элементы перекладывались из одной части в другую. Когда мы хотим сделать дек, то тоже заведём левую и правую половину и будем отвечать на запросы так:

- Если нужно добавить с какого-то конца, то добавляем на вершину соответствующей половины.
- Если нужно удалить с левого конца и левый стек пуст, то переложим с помощью третьего стека половину правого стека в левый (верхнюю половину откладываем во временный стек, нижнюю перекладываем справа налево, элементы из временного стека возвращаем в правый стек). Аналогично если нужно удалить с правого конца и правый стек пуст.
- Если нужно удалить с какого-то конца и его стек не пуст, то удаляем элемент с вершины.

Если аккуратно порисовать, то станет понятно, что порядок в деке сохраняется правильным и следовательно такая структура точно работает.

Значит осталось только доказать асимптотику, для этого введём потенциал $\Phi_i = |lsz_i - rsz_i|$, где lsz_i и rsz_i - размеры левого и правого стека соответственно после ответа на i запросов. Проверяем условия потенциалов:

- Второе условие: $lsz_i, rsz_i \leq i \Rightarrow \Phi_i = |lsz_i - rsz_i| \leq i \leq 2q$;
- Добавление: $t_i = 1, \Phi_i - \Phi_{i-1} \leq 1 \Rightarrow t_i + \Phi_i - \Phi_{i-1} \leq 2$;
- Удаление без перебрасываний: $t_i = 1, \Phi_i - \Phi_{i-1} \leq 1 \Rightarrow t_i + \Phi_i - \Phi_{i-1} \leq 2$;
- Перебрасыванием (слева направо): $t_i = lsz_{i-1}, \Phi_i - \Phi_{i-1} = |x| \in \{0, \pm 1\} - |lsz_{i-1} - 0| \leq -lsz_{i-1} + 1 \Rightarrow t_i + \Phi_i - \Phi_{i-1} \leq 1 \leq 2$;

Где в пункте с перебрасыванием мы не считаем константу при lsz_{i-1} , так как от неё можно избавить умножением потенциала на такую же константу. То есть доказали, что такой дек тоже работает за $O(1)$ на запрос.

Дек на закольцованным буфере

И, наконец, последний хорошо известный способ реализации дека - на закольцованным буфере. К такой реализации не написан код, но это не отменяет его важности для описания.

Храним внутри дека вектор и два индекса: позицию начала и позицию конца дека. При этом индексы мы считаем закользованным, то есть после последнего элемента идёт первый, а перед первым - последний. Тогда, если вектор ещё заполнился не полностью, то добавляем элемент в свободную ячейку, соседнюю с нужным концом, и двигаем нужный индекс. Аналогично при удалении просто сдвигаем индекс. Если же мы хотим добавить элемент и при этом весь вектор уже заполнен, то заводим новый вектор в два раза больше и копируем все данные туда (или можно много раз сделать `push_back` и переместить туда наши данные).

Понятно, что такой способ тоже будет работать за амортизировано $O(1)$ на запрос, так как по сути мы упираемся в асимптотику вектора.

Почему же у нас описано так много способов реализовать дек? Во-первых способы выше упорядочены по сложности написания: от простого к сложному. А во-вторых поговаривают, что на практике третий способ будет быстрее второго, а второй будет быстрее первого - именно поэтому нам интересно придумывать как можно больше способов даже с одинаковой асимптотикой, чтобы на практике смотреть какой оказался быстрее (то есть тут мы уже можем гнаться за константой, той самой, которую мы так нещадно отбросили в асимптотике).

Бинарные деревья поиска

Теперь настала пора переходить к бинарным деревьям поиска. Их придумано много разных видов и остальные много занятий мы посвятим им. А сейчас коротки опишем, что будет общего у всех бинарных деревьев поиска:

- Структура хранится в виде дерева, где в каждой вершине хранится ключ для поиска (`value`) и указатель на левого и правого ребёнка (каких-то может не быть, тогда они `nullptr`).

- Для каждой вершины выполняются $\$t\$$ условия: $\$\\forall x \\in t.L: \\ x.value < t.value \\land \\forall x \\in t.R: \\ x.value < t.value \$$, где за $\$t.L\$$ и $\$t.R\$$ обозначены левое и правое поддерево соответственно.
- Заметим, что в вершинах можно дополнительно хранить ещё какую-то информацию, у нас это размер поддерева (он пока не пересчитывается, но возможность всё же есть).

Тогда для всех таких деревьев поиска операция поиска элемента реализуется очень легко:

- Стартуем с корня, а если дошли до `nullptr`, то останавливаемся.
- Если в текущей вершине значение ровно такой, как мы ищем, то нужная вершина нашлась.
- Иначе если значение в вершине больше, то спускаемся в левое поддерево и повторяем рекурсивно; а если меньше, то спускаемся в правое поддерево и тоже повторяем рекурсивно.

Прекрасная же структура! Единственный её минус, что пока наше дерево может получиться не сбалансированным и до разных вершин придётся идти разные пути. Но ничего, эту проблему люди научились решать разными способами и мы тоже научимся.

Урок 8: Rbst, больше наследования.

[К главному описанию](#)

Краткий план

1. Split и Merge.
2. Рандомизированное бинарное дерево поиск (rbst).
3. Как написать rbst аккуратно.
4. Виртуальные методы.
5. Приведение типов.
6. Неопределённое поведение.

Мотивация

Настала пора уменьшить глубину наших деревьев поиска, для этого попробуем использовать рандом. Сразу заметим, что глубина любого дерева поиска будет хотя бы логарифм, а значит наша цель - достичь именно его.

Split и Merge

Сегодняшние структуры будут выражаться через две простых операции: `Split` и `Merge`. Первая разделяет бинарное дерево поиска по переданному ключу на два дерева поиска (в первом все элементы меньше, а во втором - больше либо равные). А вторая операция склеивает два бинарных дерева поиска при условии, что ключи одного меньше чем ключи другого.

Через эти две операции можно выразить остальные:

- `find x`: сделаем `Split` сначала по `x`, а потом у правой части по `x + 1` (если тип не поддерживает операцию `+1`, то придётся определять несколько разных `Split`). Итого получили три разных куска, средний из них отвечает за `x`: если он `nullptr` то таких элементов не было, а иначе - были.
- `insert x`: делаем такие же `Split` как и в операции `find` и потом делаем `Merge` всех частей в порядке `<x, x, >x`.
- `erase x`: делаем такие же `Split` как и в `find`, а оставляем только результат `Merge` от `<x` и `>x`.

Обе операции, `Split` и `Merge` можно реализовать рекурсивно. Также заметим, что операции `Split` по сути занимается тем же самым, чем раньше занимался `find`, следовательно `Split` можно написать общий для всех деревьев поиска, а менять `Merge` - в нашем случае на какое-то использование рандома.

Рандомизированное бинарное дерево поиска

В каждой вершине мы будем подсчитывать размер её поддерева. И при операции `Merge(L, R)` корнем будет становиться корень `L` с вероятностью $\frac{|L|}{|L| + |R|}$, а `R` будет корнем с вероятностью $\frac{|R|}{|L| + |R|}$. Можно показать, что тогда итоговая глубина получится логарифмическая, то есть оптимальная.

Как это написать

Во-первых начнём с безжалостных изменений в интерфейсе для бинарного дерева поиска, теперь он превратится в:

```
template<typename Child> using bst_t = Child;
template<typename Child> using bst_ptr_t = Child*;

template<typename T, typename Child>
class BinarySearchTree {
private:
    void update_info();
protected:
    T value;
    bst_ptr_t<Child> left, right;
    virtual void update() = 0;
public:
    BinarySearchTree();
    BinarySearchTree(T value);
    BinarySearchTree(T value, bst_ptr_t<Child> l, bst_ptr_t<Child> r);

    std::pair<bst_ptr_t<Child>, bst_ptr_t<Child>> split_lt_geq(T key);
    std::pair<bst_ptr_t<Child>, bst_ptr_t<Child>> split_leq_gt(T key);

    bst_ptr_t<Child> insert(T value);
    std::pair<bool, bst_ptr_t<Child>> find(T value);
    bst_ptr_t<Child> erase(T value);

    virtual ~BinarySearchTree();
    void print();
};
```

А для rbst объявление будет таким:

```
template<typename T> class RandomBinarySearchTree;
template<typename T> using rbst_t = bst_t<RandomBinarySearchTree<T>>;
template<typename T> using rbst_ptr_t = bst_ptr_t<RandomBinarySearchTree<T>>;

template<typename T>
class RandomBinarySearchTree : public BinarySearchTree<T, RandomBinaryS
private:
    static size_t get_size(rbst_ptr_t<T> t);
    static rbst_ptr_t<T> merge(rbst_ptr_t<T> l, rbst_ptr_t<T> r);
    size_t cur_size;
    void update() override;
public:
    RandomBinarySearchTree();
    RandomBinarySearchTree(T value);
    RandomBinarySearchTree(T value, rbst_ptr_t<T> left, rbst_ptr_t<T> r

        friend BinarySearchTree<T, RandomBinarySearchTree<T>>;
};
```

С реализацией всё должно быть примерно понятно, достаточно написать по описанию структуры. осталось только пояснить некоторые ключевые слова используемые в объявлениях.

Ключевое слово `static` в этом контексте означает, что это объявляется метод класса и вызывать его надо будет через `<Класс>::<Метод>`.

Виртуальные методы

Представим, что у вас есть базовый класс, который хочет, чтобы в дочерних классах была определена какая-то функция. Тогда если объявить её виртуальной (`virtual`), то её можно будет переопределить в дочерних классах и при её вызове от указателя будет вызываться именно дочерняя функция, а не родительская. Это проявление полиморфизма, и можно привести такой поясняющий пример:

```
class Parent {  
public:  
    void f() const {  
        std::cout << "Parent::f" << std::endl;  
    }  
    virtual void g() const {  
        std::cout << "Parent::g" << std::endl;  
    }  
};  
class Child: public Parent {  
public:  
    void f() const {  
        std::cout << "Child::f" << std::endl;  
    }  
    void g() const override {  
        std::cout << "Child::g" << std::endl;  
    }  
};  
  
Child c;  
Parent* p1 = &c;  
p1 -> f(); // Parent::f  
p1 -> g(); // Child::g  
Parent& p2 = c;  
p2.f(); // Parent::f  
p2.g(); // Child::g
```

При этом слово `override` опционально, то есть и без него всё будет работать. Но всё же его лучше указывать, ведь так явно понятно что вы делаете переопределение, а компилятор будет ругаться, если переопределять такое вы не могли. Также кроме этих слов ещё используется `final`, чтобы показать, что дочерним классам нельзя переопределять функцию.

Приведение типов

При реализации методов `bst` и `rbst` нам понадобится несколько раз делать приведение типов, для этого придумано много разных способов:

- `const_cast` - добавление и удаление константности;
- `static_cast` - приведение типов, корректность которых проверяется во время компиляции;
- `dynamic_cast` - приведение типов, можно делать более хитрые преобразование, чем в `static_cast`;
- `reinterpret_cast` - можно делать совсем странные вещи, например строку (из произвольных символов) в число;
- `C-style cast` - в каком-то порядке выбирается приведение типов из списка выше;

Нам хватит `static_cast`, да и в целом его почти всегда достаточно. `C-style cast` считается плохим синтаксисом и его лучше избегать.

Неопределённое поведение

В текущей реализации есть UB (`undefined behaviour`) в моменты, когда мы делаем вызовы вида `nullptr -> split ...`. Но на практике кажется, что функцию вызывать получается и пока не будет разыменования нулевого указателя - всё будет работать. Но опять же повторюсь, что это не по стандарту и лучше бы как-то переписать этот фрагмент кода.

Урок 9: Декартово дерево, обычное и по неявному ключу.

[К главному описанию](#)

Краткий план

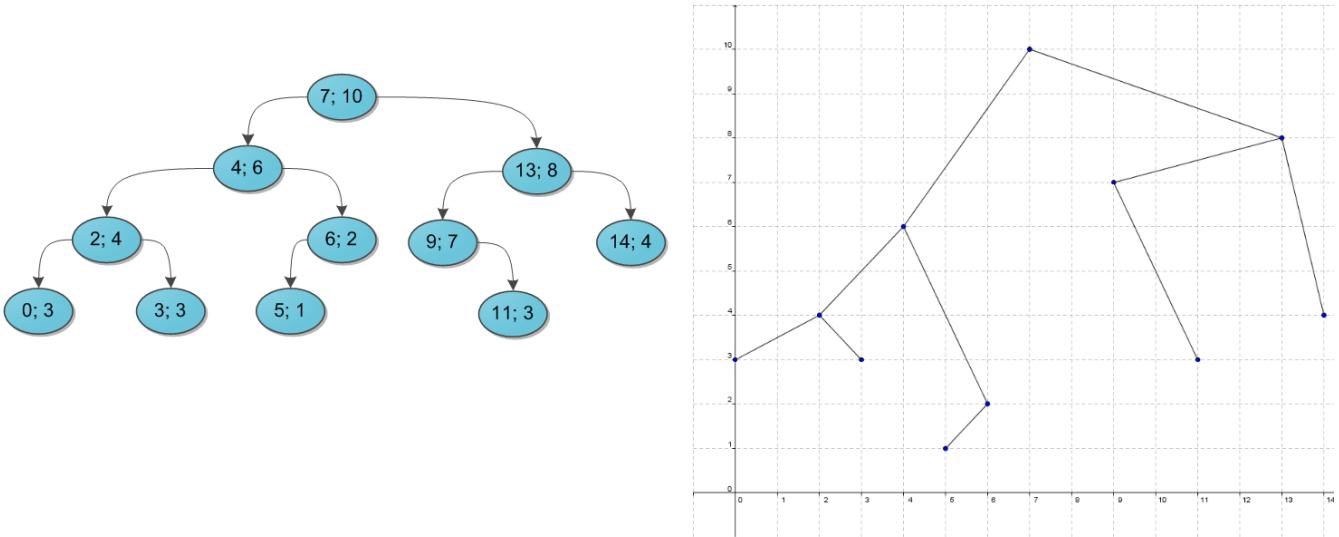
1. Как работает Декартово дерево.
2. Декартово дерево по неявному ключу (с операцией суммы на отрезке).

Мотивация

Конечно rbst структура хорошая, но в тех же олимпиадах люди предпочитают писать декартовые деревья (ДД). Не очень понятно, с чем связаны именно такие предпочтения, наверное у ДД ожидаемая глубина меньше (но опять же только в константу раз, не асимптотически).

Как работает Декартово дерево

Раз уж мы хотим где-то применить рандом, то для этого давайте каждой вершине присваивать свой случайный приоритет. И потом с помощью этого приоритета реализуем Merge: корнем станет та вершина, у которой приоритет больше. Такую структуру принято называть Декартовым деревом, так как она очень удачно изображается на обычной декартовой плоскости:



Пишется декартово дерево в целом точно так же, как и rbst: в конструкторах добавляем генерацию приоритета, а при слиянии двух ДД используем эти приоритеты.

Декартово дерево по неявному ключу (с операцией суммы на отрезке)

Представим, что перед нами стоит такая задача: дан какой-то массив и поступают

запросы добавления и удаления элемента по индексу, а также запрос суммы на отрезке. При этом разрешается все операции реализовывать за $O(\log n)$.

Идея такая: откажемся от ключей и вместо них будем хранить в каждой вершине размер её поддерева. Тогда при операции `Split` будем пользоваться этими размерами, а `Merge` будем делать точно так же, как и раньше, с помощью приоритетов, сохранённых в вершинах.

К сожалению (или к счастью), данная модификация достаточно сильно отличается от остальных реализованных структур, поэтому придётся снова всё скопировать и немного исправить: часть из `bst` (`split`), часть из `rbst` (`update`) и часть из `treap` (`merge`).

Кроме того замети, что в декартово дерево очень легко можно добавить обработку каких-то запросов на отрезке. Для этого в вершине будем хранить нужную величину (например сумму в поддереве), и внутри функции `update` будем эту величину пересчитывать (а `update` вызывается, когда в поддереве что-то поменялось).

Урок 10: Splay Tree: амортизованный логарифм в бинарном дереве поиска.

[К главному описанию](#)

Краткий план

1. Как работает Splay Tree.
2. Операции со Splay деревом.
3. Асимптотика операций со Splay деревом.

Мотивация

Мы уже прошли rbst и декартово дерево, но всё же в них используется рандом, так что иногда они могут давать плохие результаты. Но есть и детерминированные деревья поиска с логарифмической глубиной, сегодня познакомимся с Splay Tree - одной из таких структур.

Splay Tree

Основная логика у структуры такая: пусть мы сделали модификацию в какой-то вершине, тогда давайте поднимем эту вершину в корень. Так мы получаем быстрый доступ к недавно модифицированным вершинам, а если подъём сделать правильно то ещё и логарифмическую сложность, правда только амортизировано (то есть в среднем за все запросы).

Всё строится на операции поворота ребра `rotate_edge`. Пусть у нас были две вершины $A\$$ и $B\$$ соединённые ребром и $A\$$ - родитель $B\$$. Тогда если аккуратно выписать всех детей в порядке сортировки, то можно довольно понятным способом сделать $B\$$ родителем $A\$$, чтобы порядок сортировки сохранился (нарисуйте картинку сами).

Теперь мы хотим научиться поднимать вершину в корень, тогда возможны три ситуации:

- У вершины есть только родитель и нет деда (`zig`) - тогда просто выполним поворот ребра.
- У вершины $A\$$ есть и родитель $B\$$ и дед $C\$$, причём родство в разные стороны (`zig-zag`) - тогда выполним сначала поворот ребра $A - B\$$, а потом ребра $A - C \$$.
- У вершины $A\$$ есть и родитель $B\$$ и дед $C\$$, причём родство в одну сторону (`zig-zig`) - тогда выполним сначала поворот ребра $B - C\$$, а потом ребра $A - B \$$.

Именно такие повороты и подъёмы позволяют достичь амортизированного логарифма.

Операции со Splay деревом

Выше описали как работает операция `expose(x)` - поднятие вершины x в корень. Теперь выразим обычные операции, которые мы хотим делать с деревьями поиска через уже известные нам:

- `find x`: спускаемся как по бинарному дереву поиска, в конце делаем операцию `expose` от вершины, куда пришли.
- `merge L, R`: находим самый большой элемент в L (спускаемся в право если можем, иначе останавливаемся) и делаем его `expose`. После этого подвешиваем R в качестве правого сына к L .
- `split x`: делаем поиск элемента x и его подъём с помощью `expose`. В качестве меньших элементов возвращаем левого сына, а в качестве больших - правого.
- `insert x`: сначала делаем `split x`, а потом результаты подвешиваем как детей к x .
- `erase x`: сначала делаем `split x`, а потом делаем `merge` его результатов.

Асимптотика операций со Splay деревом

Для каждой вершины x заведём её ранг: $r(x) = \log_2 sz(x)$, где $sz(x)$ - количество вершин в поддереве с корнем в x .

Пусть теперь t - корень дерева и мы делаем `expose x`, тогда хотим доказать, что он будет выполняться не дольше, чем $3r(t) - 3r(x) + 1$.

Здесь нужно аккуратно разобрать все виды подъёмов и показать, что неравенство верно во всех случаях. Если это будет сделано, то когда-то потом :)

Урок 11: 2-3 дерево: строгий логарифм в дереве поиска.

[К главному описанию](#)

Краткий план

1. Структура В-дерева.
2. Операции с В-деревом.
3. Как работает 2-3 дерево.

Мотивация

Кажется, что по прошлым урокам можно предсказать, какую структуры данных мы хотим написать сегодня. В самом деле сначала у нас был логарифм рандомизированный, потом амортизированный, значит сейчас хотим добиться строго логарифма. То есть сегодняшняя структура будет отвечать на любой запрос всегда за логарифм от количества хранимых элементов. С теоретической точки зрения уже нет смысла придумывать структуры лучшее, так как быстрее логарифма отвечать на запросы не получится (иначе бы мы научились сортировать быстрее чем за $O(n \log n)$), и ничего лучше логарифма на каждый запрос тоже не придумано.

Структура В-дерева

Раньше мы познакомились с бинарным деревом поиска: там в каждой вершине хранится какой-то элемент, в детях левее значения меньше, а в детях правее -- больше. Но пусть мы хотим, чтобы из вершины было не два ребёнка, а больше, например $C\$$. Тогда давайте поделем вершины дерева на два типа: листовые и внутренние. Во внутренних вершинах нужно как-то научиться переходить к какому-то ребёнку, поэтому давайте если у вершины $C\$$ детей, то в вершине будем хранить $C - 1\$$ разделяющее число (например максимум во всех детях кроме самого правого). Тогда во внутренних вершинах сможем переходить к правильным вершинам детям и дальше всё аналогично бинарным деревьям поиска.

Заметим, что все данные мы храним только в листьях, а время на запрос к любым данным хотим получить одинаковое. Следовательно нужно как-то сделать так, чтобы все листья были на одной глубине. И в общем понятно, что если рассмотреть полное бинарное дерево и добавить в него один элемент, то понадобится произвести довольно много перестройки; аналогично, если добавить что-то между уже существующими числами, то тоже не ясно, как всё хорошо перестроить. Поэтому давайте наложим меньше ограничений на структуру дерева, а именно сделаем у вершины не фиксированное количество детей, а диапазон $[B; 2B]$ (по факту в бинарном дереве у нас тоже $0, 1, 2$ детей, так что идея нам знакома), исключением будет корень, там $[0; 2B]$ вершин.

Операции с В-деревом

Искать данные в В-дереве мы вроде бы научились, поэтому надо научиться их

добавлять и удалять.

- `insert x`: пусть хотим вставить какой-то элемент в структуру, тогда поиском можно найти место вставки и добавить элемент туда. Но может так произойти, что после этого у вершины станет $2B$ детей -- в таком случае делим её пополам. После этого проблемы с излишком дочерних вершин поднялись на один слой выше, в общем дальше повторяем такие операции. Если на самом верхнем слое кроме корня добавили ещё что-то, то создаём для них общего родителя и назначаем его корнем.
- `erase x`: пусть хотим удалить какой-то элемент из структуры, тогда поиском можно найти место удаления и убрать его оттуда. После этого у вершины может стать слишком мало детей ($B - 1$), тогда можно или попросить одного ребёнка у брата, или если ни у кого нет лишних детей, то можно отдать брату всех своих. Во втором случае мы удаляем текущую вершину, поэтому проблемный слой переместился на один вверх и дальше продолжаем аналогично.

Казалось бы, какая ужасная структура. Но нет, она применяется в жизни, говорят в алгоритмах во внешней памяти (в таких алгоритмах мы можем считывать данные блоками и хотим минимизировать количество обращений ко внешней памяти). Заметим, что теперь глубина дерева стала $\log_B n$, а это даже несколько меньше, чем $\log_2 n$. Но на самом деле мы платим тем, что при спуску по дереву выбор ребёнка будет происходить дольше, за $O(\log_2 B)$. Но опять же в алгоритмах во внешней памяти это всё приемлемо.

2-3 дерево

У нас память не внешняя, поэтому B можем выбрать сами какое хотим, и поэтому выберем $B = 2$, получим структуру, называемую 2-3 деревом. Её основные параметры:

- У каждой вершины кроме корня 2 или 3 ребёнка, во внутренних вершинах кроме корня храним 1 или 2 разделяющих элемента.
- Для `find x` просто спускаемся по дереву.
- Для `insert x` спускаемся по дереву, добавляем x куда нужно и если это четвёртый ребёнок, то делим вершину на две и поднимаем проблемы на один слой вверх.
- Для `erase x` спускаемся по дереву, удаляем x откуда нужно и если ребёнок остался всего один, то пытаемся его отдать кому-нибудь, или взять у других ребёнка взаймы. Если пришлось отдать то убираем вершины и поднимаем потенциальную проблему на слой выше.

Урок 12: Пары и set, map, multi- с итераторами.

[К главному описанию](#)

Краткий план

1. Реализация set и map.
2. Реализация multiset и multimap.
3. Пары.
4. Итераторы.

Мотивация

Кто знаком со встроенными в C++ структурами данных знает, что всего пройденного нами разнообразия структур данных там нет. А есть только set, map, multiset и multimap. Так давайте сегодня научимся делать эти структуры на основе изученных нами деревьев.

Во всех этих структурах нам нужен доступ данным за логарифм (такое у нас есть в изученных структурах) и хранение данных в отсортированном порядке (такое тоже есть). Ещё у всех структур есть итераторы, с ними видимо разберёмся отдельно.

Реализация set и map

В зависимости от реализации все изученные нами структуры хранили обычный set. Но давайте всё же выделим пункты, которые нам наверняка гарантируют, что у нас получится именно set (то есть элементы не повторяются):

- При добавлении элемента не добавляем новый, если уже есть (или сначала удаляем старую версию, а потом добавляем новую).
- При удалении элемента просто удаляем эту единственную версию.

Теперь перейдём к map. По сути map<A, B> мы можем заменить на set<pair<A, B>>, где сравнение в паре происходит только по первой координате. Это очень удобная замена, потому что позволяет сводить одну структуру к другой. Если же хочется скопировать реализацию чего-то уже сделанного, то можно добавить в каждую вершину с данными (типа A) поле info типа B.

Реализация multiset и multimap

В этих структурах мы хотим иметь возможность хранить несколько элементов с одинаковым ключом. В целом, изученным нами структуры позволяют хранить в них одинаковые элементы (если несколько аккуратнее отнестись к коду), но мне не нравится такой путь, придётся какие-то равенства отдельно рассматривать, ужас. Поэтому предлагается сделать замены виды:

- multiset<A> на set<pair<A, size_t>> или на set<pair<A, vector<A>>> --

где в первом случае мы для элемента храним количество его добавлений в структуру, а во втором -- все добавления. Мотивация у второго способа в том, что равные элементы для множества могут иметь разную информацию (например, если оператор сравнения не смотрит на какие-то поля).

- `multimap<A, B>` на `map<A, vector>` -- для каждого ключа храним список из всех его значений.

В целом можно поступать довольно творчески: например если В дорого копировать, то вместо `vector` использовать `list`; если мы знаем, что A в `multiset` какой-то тривиальный встроенный тип со сравнение по умолчанию (например числа или строки), то можно использовать `pair<A, size_t>`, а не всякие `pair<A, vector<A>>`.

И замечу, что типы используемые выше: `vector` и `list` мы уже научились писать, следовательно можно пользоваться нашими версиями, а не версиями из стандартной библиотеки. Единственное, что у нас раньше не было `pair`, стоит его реализовать.

Пары

Кажется, что с парами всё достаточно просто: нужно создать шаблонный класс от двух аргументов (разные поля пары могут потенциально иметь разные типы данных). Так что объявление и определение достаточно понятные:

```
template<typename F, typename S>
class Pair {
public:
    F first;
    S second;
    Pair();
    Pair(F first, S second);
};
```

Кроме того заметим, что в таком случае валидны все три способа создания пар:

```
Pair<int, string> pis1;
Pair<int, string> pis2(1, "one");
Pair<int, string> pis3{2, "two"};
```

Итераторы

По хорошему множества, словари и их мультиварианты (да и просто бинарное дерево поиска) должны поддерживать итераторы. В целом довольно понятно, как делать `++` и `--` итератора: нужно как-то найти следующий элемент в структуре, который больше или меньше данного.

Пусть мы стоим в какой-то вершине `v` бинарного дерева поиска и хотим найти минимальный элемент, больше текущего. Тогда если есть правый сын, то нужно спустить в него и найти его самого левого ребёнка (минимальный элемент). Если же ребёнка справа нет, то нужно подняться к родителю `p`: если `v` -- левый сын `p`, то как раз `p` будет больше `v`; а если `v` правый сын `p`, то значит всё поддерево `p` мы обошли, следовательно придётся подниматься выше. Аналогично можно найти максимальный элемент, который больше текущего. Чтобы найти минимальный (или максимальный) элемент в структуре достаточно всё время спускаться пока можем

влево (вправо).

В общем понятно, как это написать, но кодовая реализация появится только после появления множеств, словарей и их мультиверсий.