**АЛГОРИТМЫ**

**ПОИСКИ**

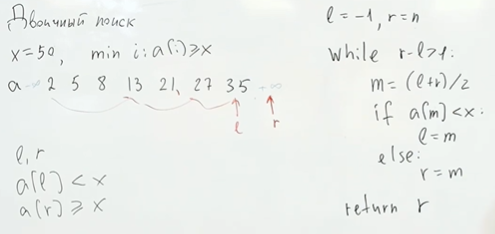
**Бинарный поиск (двоичный поиск)**

Главные условия – конечный массив в котором все элементы отсортированы по возрастанию.

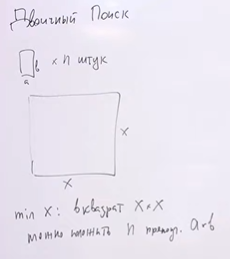
Суть алгоритма в том, чтобы брать центральный элемент массива и сравнивать с искомым значением, в зависимости от результата сдвигать левую или правую границу на выбранный центральный элемент, и рекурентно проделываем это пока не получим результат.

Правильный код бинарного поиска нужно писать следующим образом:

Допустим у нас задача найти ближайшее наибольшее число к заданному, тогда к исходному массиву мы добавляем первым и последним элементом виртуальную +- бесконечность, это будут левая и правая границы поиска. А далее будем также выбирать центральный элемент, относительно которого, левые элементы будут меньше значения, а правые больше или равны.



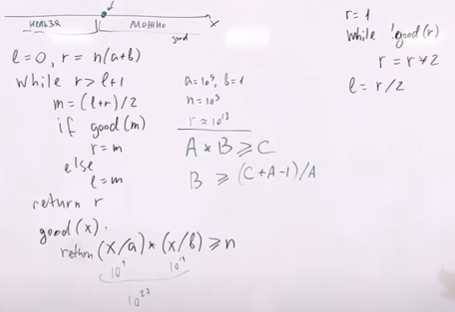
Задача 1: найти минимальную сторону квадратной доски, на которую можно пове сить n постеров размером a на b.



Решение: определить данную задачу как бинпоиск – у нас на отрезке размеров стороны квадрата можно найти точку с которой начнется удовлетворять условие возможности разместить все постеры, а все что до нее не будет удовлетворять.

В первую очередь нужно определить границы поиска, левая граница будет всегда нулевой, то с правой все сложнее. Для правой границы нужно не выйти за пределы типа, из-за того, что число может быть большим, то переполнения возникнут проблемы. Для этого установим правую границу 1, и пока она не будет удовлетворять good, будем увеличивать ее в два раза. Таким образом добьёмся того, что правая граница будет не больше чем 4n.

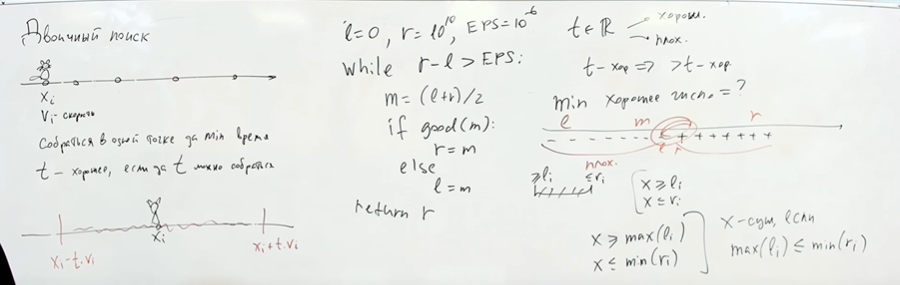
Следующим шагом пишем обычный бинпоиск, в котором мы функцией проверки good будем проверять можно ли центральный выбранный элемент назвать удовлетворяющим. Для этого будем проверять что ((x/a)\*(x/b) >= n).



Задача 2: Существует отрезок на котором в рандомном порядке расставлены люди, имеющие координату и скорость передвижения. Найти наименьшее время, за которое все могут оказаться в одной точке.

Решение: во-первых, т.к. числа вещественные, нужно помнить, что точности нужной можно добиться только, указав промежуток, во-вторых из-за нюанса с вещественными числами в ЯП, можно уйти в бесконечный цикл.

Такой же алгоритм бинпоиска, с тем нюансом, что он будет выполнятся пока разница между правой и левой границей (временем сбора хорошим и плохим) будет больше некого отрезка эпсилон. А проверка условия good будет для всех считать промежутки, в которых может оказаться человек, после чего посчитать точки пересечения, найти min и max. Также необходимо не забыть дописать условие на зацикливание double.



Выполняется за О (log n)

int **funk**(int \*arr, int f)

//плохая реализация

{

int rightLine = 11;

int leftLine = -1;

while (rightLine - leftLine != 1)

{

int midline;

midline = (rightLine + leftLine)/2;

if(arr[midline] < f)

leftLine = midline;

else

{

rightLine = midline;

}

}

cout << rightLine;

return 0;

}

int **main**()

{

int mas[11] = {3,6,7,9,23,23,56,57,67,98,100};

int foun = 23;

funk(mas,foun);

**СОРТИРОВКИ**

**Пузырьковая сортировка**

Сложность: O(N^2)

Выполняется некоторое количество проходов по массиву — начиная от начала массива, перебираются пары соседних элементов массива. Если 1-й элемент пары больше 2-го, элементы переставляются (выполняется обмен). При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива (как бы «всплывает» до нужной позиции, как пузырёк в воде — откуда и название алгоритма).

int passages = 0;

for(int i = 0; i < n - 1; i++){

swapped = false;

passages++;

for(int j = 0; j < n – i – 1; j++){

if(arr[j]>arr[j+1]{

int temp = arr[j+1];

arr[j+1] = arr[j];

arr[j] = temp;

swapped = true;

passeges++;

}

}

if(!swapped) break;

}

**Сортировка вставками (Insertion Sort)**

Сложность: O(N^2)

Алгоритм сортировки, на каждом шаге которого массив постепенно перебирается слева направо. При этом каждый последующий элемент размещается так, чтобы он оказался между ближайшими элементами с минимальным и максимальным значением.

for(int i = 1; i < n ; ++i){

int k = i;

while(k>0 && arr[k-1]>arr[k]){

if(arr[j]>arr[j+1]{

int temp = arr[k-1];

arr[k-1] = arr[k];

arr[k] = temp;

k -= 1;

}

}

}

**Сортировка подсчетом (counting sort)**

Специальная сортировка.

Доп. память: O(M).

Сложность: O(N + M) , где М – мощность сортируемого контейнера.

Изначальный имеем неотсортированный массив ARR и вспомогательный массив AN, в котором индексы от 0 до M являются значениями неотсортированного массива. Проходим последовательно ARR и при нахождении числа i увеличиваем счетчик в ячейке дополнительного массива AN на 1. Таким образом при прохождении всего массива ARR в массиве AN будет содержаться статистика частоты появления числа I по порядку. ВАЖНО чтобы количество и разброс значений i был невелик, иначе массив AN будет содержать в себе много ненужной инфы.

void **counting\_sort**(int arr[], int N, int M)

{

int tmp\_arr[M];

for (int i = 0; i < M; i++) {

tmp\_arr[i] = 0;

}

for(int i = 0; i < N; i++){

tmp\_arr[arr[i]]++;

}

int counter = 0;

for (int i = 0; i < M; i++) {

for (int j = 0; j < tmp\_arr[i]; j++) {

arr[counter] = i;

counter++;

}

}

}

*Плюсы:* скорость работы алгоритма при маленькой мощности входного контейнера.

*Минусы:* обязательство знать диапазон значений в изначальном массиве (мощность), также выделение дополнительной памяти.

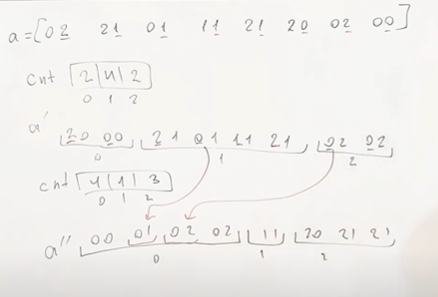
**Поразрядная сортировка (radix sort)**

Специальная сортировка.

Доп. память: O(N).

Сложность: O(k\*(n+m)).

Пусть в качестве аргумента сортировке передается массив, в котором содержатся n m-значных чисел, и каждая цифра может принимать значения от 0 до k−1. Тогда цифровая сортировка позволяет отсортировать данный массив за время O(m(n+k)), если устойчивая сортировка имеет время работы O(n+k). Если k небольшое, то оптимально выбирать в качестве устойчивой сортировки сортировку подсчетом.Алгоритм поразрядной сортировки гениален тем, что сортирует не числа целиком, а значения разрядов. Получается, что он как бы разбирается с числами на уровне единиц, десятков, сотен и т. д. и только потом он делает общую сортировку. Это позволяет ему не бегать по всем сравниваемым числам и не делать миллион сравнений. Отсюда и экономия времени.



void **radix\_sort**(int arr[], const int& N, const int& M)

{

int arr\_null[N];

int arr\_unit[N];

int size\_null = 0;

int size\_unit = 0;

int radix = 0;

for (int i = 0; i < N; i++)

arr\_null[i]=0;

for (int i = 0; i < N; i++)

arr\_unit[i]=0;

while(radix < M) // проходим по массиву M раз, где М - количество бит в наибольшем числе массива

{

for (int i = 0; i < N; i++)

{

if(arr[i] & (1 << radix)) // побитовая маска для вычленения нужных битов числа

{

arr\_unit[size\_unit++] = arr[i]; }

else

{

arr\_null[size\_null++] = arr[i]; }

}

// копирование информации из вспомогательных массивов в основной

for (int i = 0; i < size\_null; i++)

{

arr[i] = arr\_null[i];

}

for (int i = 0; i < size\_unit; i++)

{

arr[i + size\_null] = arr\_unit[i];

}

size\_null = 0;

size\_unit = 0;

radix++;

}

}

*Плюсы: имеет линейную временную сложность, эффективна для сортировки большого количества целых чисел или строк.*

*Минусы: неэффективна для сортировки чисел с плавающей запятой или других типов данных, требуется значительный объем памяти, требуется знать количество разрядов.*

**Пирамидальная сортировка (сортировка кучей – Heap sort)**

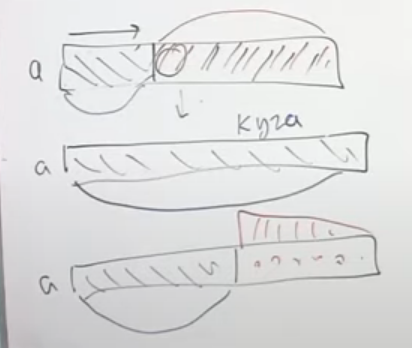
Доп. память: не требует.

Сложность: O(n logn).

Пирамидальная сортировка основана на такой структуре данных как двоичная куча. Чтобы отсортировать массив, нужно переложить элементы один за другим из массива в кучу insert(), а после также поочередно вытащить их с помощью метода get\_min() (после этого удалить элемент remove\_min()).

Однако при таком раскладе добавляются дополнительные издержки по памяти. Для избавления от доп. памяти будем делать сортировку в исходном массиве.

Заметим, что при последовательном копировании элементов, часть исходного массива до текущего элемента уже не используется. В куче же занята эта же часть скопированных элементов. Делаем вывод о том, что мы всегда используем только O(n) памяти, тогда можно это делать в одном массиве.



Мы берем i-ый элемент исходного массива и просеиваем его сверху относительно левой части исходного массива (левая часть массива уже куча). Так до последнего элемента, при этом исходный массив полностью станет кучей. Процесс трансформации массива в кучу – heapify.

Чтобы расставить элементы кучи по убыванию в этом же массиве, будем вычленять минимальный элемент кучи (т.е. первый) и добавлять его в конец массива. Таким образом получим отсортированный массив без использования дополнительной памяти.

Реализация в файле Heap\_sort.

**Быстрые рекуррентные сортировки**

Построены на принципе разделяй и властвуй. Например, есть неотсортированный массив, который мы делим пополам и последовательно сортируем эти массивы.

**1) Сортировка Тони Хоара (Quick sort)**

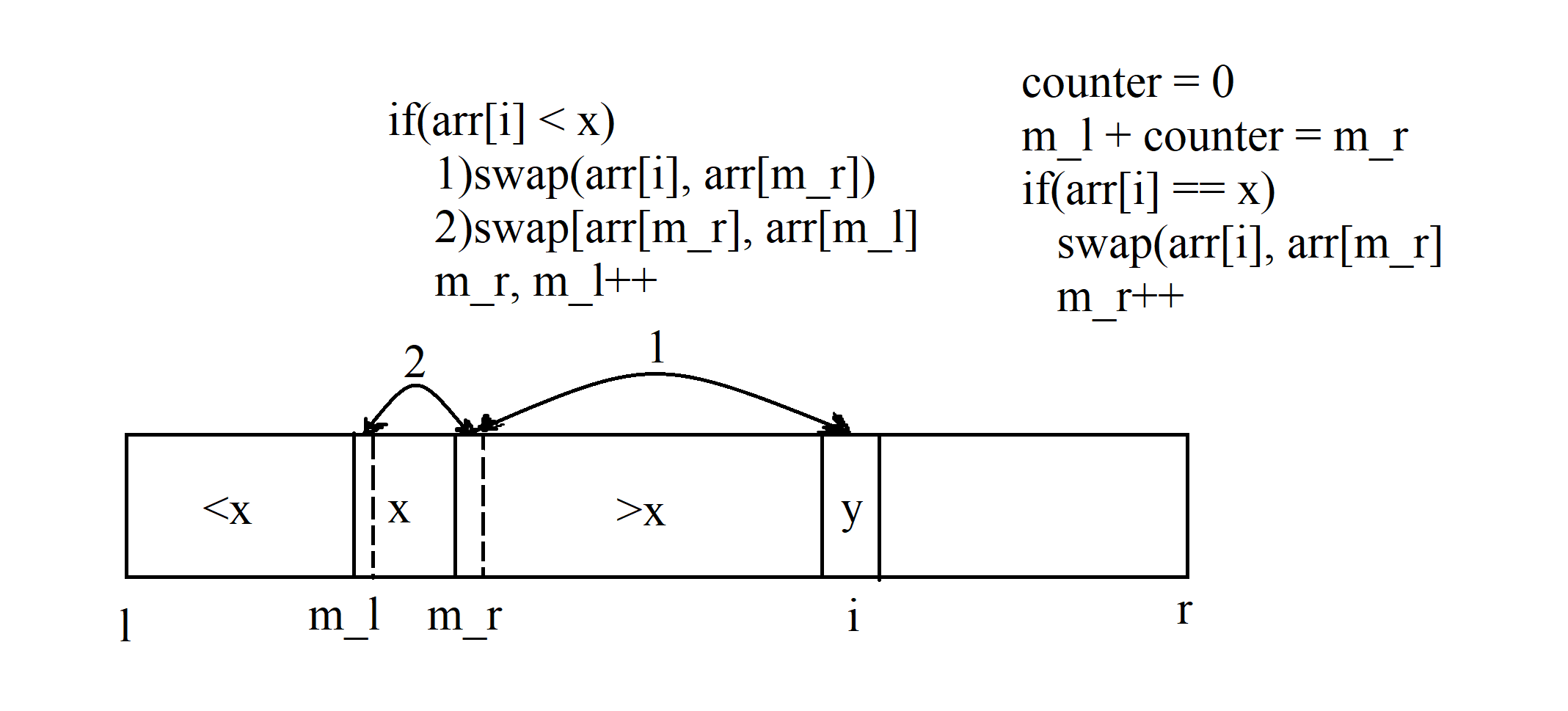
Доп. память: logn для стека рекурсии.

Сложность: среднее время - O(n\*log n), худшее - O(n\*n),

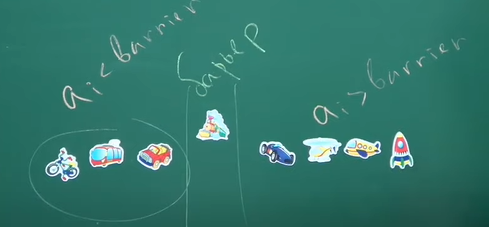
Применяем методику разделяй и властвуй уже осуществив предварительную подготовку. Ключевая идея: сортирующее действие – подготовительное (осуществляется до входа в рекурсию).

Мы берем “барьерный элемент” (рандомный элемент) и проходим исходный массив за O(n) сравнивая элементы, помещая элементы в два разных массива в зависимости от того больше или меньше барьерного. Таким образом обернув данный алгоритм в рекурсию получим отсортированные массивы, которые останется только объединить.

Для реализации без дополнительной памяти, а также с возможностью сортировать контейнеры с одинаковыми элементами, понадобится более сложная реализация. Суть в том, чтобы хранить слева от текущего итерируемого элемента отсортированную часть. Которая в свою очередь делится на три промежутка: та что меньше рандомного элемента, равна рандомному элементу, больше рандомного элемента. После сортировки, уход в рекурсию осуществляется от двух частей - большей и меньшей, часть с равными элементами остается на месте.



Данная сортировка является быстрой вероятностно. В наихудшем случае выполнится за O(n\*n), но вероятностно(среднее) работает за (n\*logn).



Реализация в файле QuickSort.

*Плюсы: является быстрой вероятностно, не использует дополнительной памяти, сортировка происходит "на месте".*

*Минусы: если барьерный элемент max или min, падает скорость сортировки.*

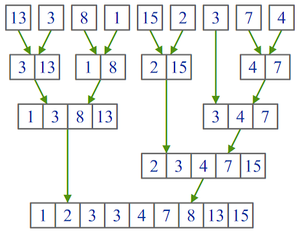
**2) Сортировка слиянием (Merge sort)**

Доп. память: O(N) для массива и O(1) для списка.

Сложность: худшее, лучшее, среднее время - O(n\*log n)

С помощью арифметики указателей разделяем массив на две части которые и будут сравниваться.

Сортирующее действие после ухода в рекурсию. Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу. Иначе массив разбивается на две части, которые сортируются рекурсивно. После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

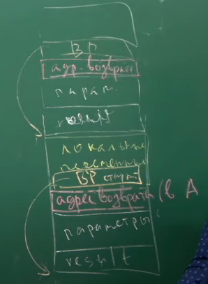


Реализация в файле Merge\_sort.

*Плюсы: устойчивая - сохраняет порядок равных элементов (принадлежащих одному классу эквивалентности по сравнению); хорошо параллелится; работает даже на структурах данных последовательного доступа.*

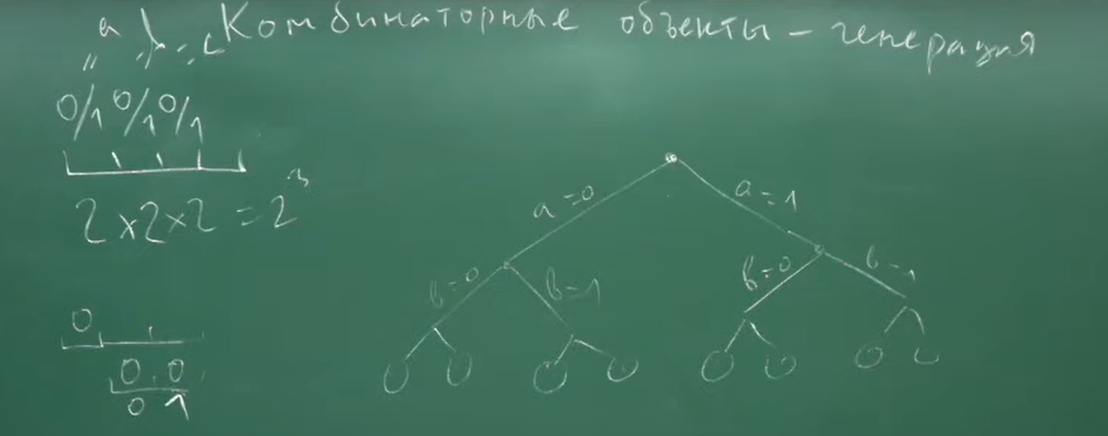
*Минусы: на «почти отсортированных» массивах работает столь же долго, как на хаотичных; требует дополнительной памяти по размеру исходного массива.*

**РЕКУРСИЯ**

Способ решения задачи через сведение ее к подзадаче (-ам), аналогичной (-ым) исходной, но проще.

На стеке рекурсия выполняется следующим образом (синхронный вызов):

При вызове функции на стеке заготавливается ячейка под возвращаемое значение. Передаваемые в функцию параметры также ложатся на стек. После этого на стек помещается адрес возврата в эту функцию, функция вызывает следующую функцию и “засыпает”, стек поинтер улетает выше. Вызванная функция кладет на стек состояние старого бэйс поинтера, раскладывает на стеке свои локальные переменные, место под результат. Так происходит до n-ой функции с которой все начнет раскручиваться. Функция n вычисляет и записывает результат, снимает со стека бэйс поинтер и возвращает его туда куда он указывал, снимает адрес возврата и будит функцию нижестоящую.



Алгоритмическая рекурсия - это по определению алгоритм, построенный по стратегии Разделяй-и-Властвуй (Divide-and-Conquer), в котором для хранения подзадач использована LIFO структура данных, т.е., попросту выражаясь, стек.

У таких алгоритмов натуральным образом есть прямой ход, когда задача разбивается на более мелкие подзадачи, которые заносятся в стек. Затем подзадачи извлекаются из стека по одной и решаются по такому же принципу, т.е. разбиваются на еще более мелкие подзадачи. Разбиение продолжается до тех пор, пока подзадача не станет тривиально решаемой.

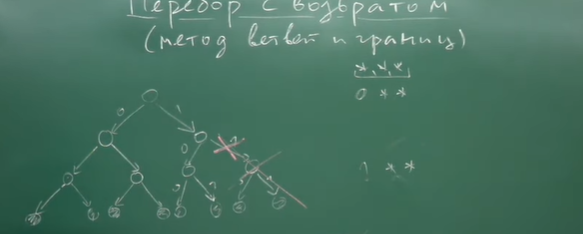
А также у рекурсивных алгоритмов есть обратный ход, когда все более мелкие подзадачи уже решены и их решения теперь можно объединить в решение более крупной задачи. Наличие обратного хода (backtracking) - отличительная черта именно рекурсивных алгоритмов. Т.е. в рекурсии всегда есть обратный ход. Вот, собственно, "при чем здесь рекурсия".

Реализация обычной рекурсии и с двумя рекуррентными вызовами в файле.

**1) Перебор с возвратом (метод ветвей и границ)**

Пример: допустим нужно сгенерировать числа размером n, при том выводить такую битовую последовательность, в которой число встретится всего один раз. Тогда на стадии рекуррентных вызовов можно начать отсекать ненужные ветви, тем самым существенно укоротив дерево.

Основная идея в том, чтобы спилить ненужную ветвь как можно раньше, тем самым не тратить время на перебор заведомо неправильных вариантов.

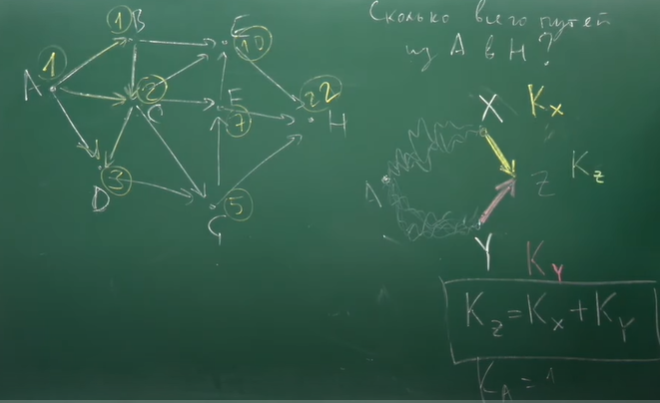


**ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

Т.к. рекурсия очень ресурсоемка и в ней возникают повторения вычислений, для оптимизации рекуррентной идеи приходит на помощь динамическое программирование. В этой парадигме мы идем как бы от обратного – от крайнего случая. Зная начальные параметры задачи (крайние случаи рекурсии), мы начинаем высчитывать следующие пока не достигнем интересующего нас.

Пример задачи с подсчетом путей из А в Х.

Рассмотрев отельный город, мы поймем, что кол-во путей в него, зависит от количества путей городов, пути которых приходят в него, а точнее их суммой. Таким образом, исходя из того, что кол-во путей в город А равно 1 (остаться в А), начнем расчеты по формуле, пока не достигнем города Х. Таким образом в отличии от рекурсии мы идем с малого и “захватываем” неизвестную область.



**1) Одномерное динамическое программирование(снизу-вверх)**

Задача 1: Сколько программ получения числа 15 из 1 операциями +1 и +2?

Решение: наподобие прошлой задачи представим числа от 1 до 15 как таблицу городов. В каждое число (город) можно попасть либо из предшествующего прибавив 1, либо из предпредшеств. прибавив к нему 2. И считая, что количество способов попасть в самый первый город равняется 1 (остаться на месте), способ получить двойку только путем +1 (т.е. одним способом), а вот тройку уже двумя, составим формулу получения n-го числа – K(n) = K(n-1) + K(n-2). Таким образом, зная начальные параметры, в отличии от рекурсии, мы начнем считать от них и идти до искомого числа.

Решение в файле dynamic programming.

Задача 2 (Задача о кузнечике): Есть кузнечик, прыгающий на +1 или на +2 за раз. У каждого числа есть своя стоимость. Найти наименьший по стоимости путь из 1 в n.

Решение: Начнем как обычно с начальных значений (крайних случаев). Кузнечик стоит на первой клетке, соответственно стоимость первого числа p1. Стоимость второй клетки рассчитывается как p1+p2. Перейдем к конечным случаям. Возьмем i-ю клетку (число) и поймем, что в нее можно попасть, прыгнув с i-2 или i-1 клетки. Соответственно при выборе с какой клетки прыгать будет важна стоимость. Получается, что стоимость i-го числа будет равняться минимальной стоимости между i-1 и i-2 клетками с учетом стоимости самой i-ой клетки – С(i) = min(C(i-1), C(i-2)) + p(i).

Решение в файле dynamic\_progr\_hopper.

**2) Двумерное динамическое программирование**

Задача 1: на шахматной доске размером n на m в координате (0;0) стоит фигура которая может ходить только на +1 вправо или на +1 вниз. Рассчитать количество путей из стартового поля в координату (n, m).

Решение: пойдем от обратного, чтобы попасть в последнюю ячейку нужно сделать +1 шаг с одной из двух соседних ячеек. Соответственно количество путей последней ячейки будет зависеть от суммы количества путей двух ячеек с которых можно на нее попасть (К(i,j) = K(i,j-1) + K(i-1,j)). Теперь определим крайние случаи – посмотрев на доску поймем, что крайние поля доски имеют только одного соседа, соответственно способов попасть в них всегда 1 (K(1,j) = 1; K(i,1) = 1).

Сложность рекурсии: O(Cnk) – кол-во сочетаний

Решение в файле dynamic\_progr\_2ddp

Задача 2 (Задача о рюкзаке со стоимостями): имеется рюкзак с ограниченной грузоподъемностью W, и n количество вещей с массой wi и ценностью ui. Нужно наполнить рюкзак вещами с максимально возможной ценностью, при этом не превысив массу рюкзака.

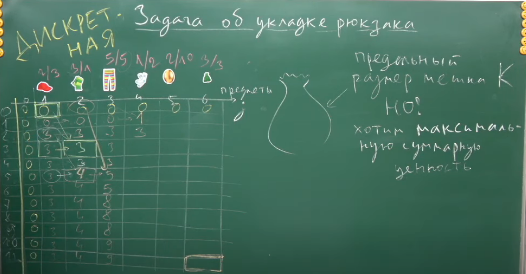
Ответом на задачу будет являться подмножество из исходных предметов с не превышающей суммарной массой и максимальной суммарной стоимостью.

Логично что раз это подмножество из множества, то мы можем начать перебирать варианты пока не найдем подходящий, однако в таком случае сложность будет O(2N).

Подумав, можно прийти к тому чтобы брать самый дорогой (удельный) по массе предмет который влезет в рюкзак. Такой алгоритм носит название жадного алгоритма. Но такой алгоритм не дает точного (максимально оптимального) решения, он приближается к ней.

Решение: Если задача с дискретными значениями масс предметов, тогда строим таблицу в которой горизонтальными значениями будут предметы, вертикальными – вместимость рюкзака от 0 до W. Искомый результат максимальной стоимости будет находится в последней ячейке (n, W).

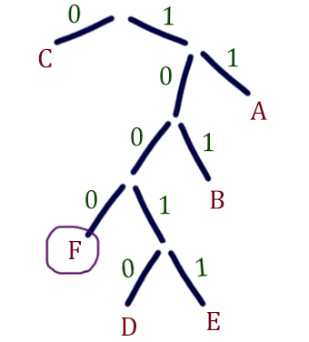
В этой матрице, будем как бы открывать доступные элементы по одному начиная с первого и высчитывать ценность которую мы унесем, по второй оси представим грузоподъемность рюкзака с нулевого до настоящего значения.



**АЛГОРИТМЫ НА СТРОКАХ**

Строка, в которой размер кодового слова постоянен – равномерный код (ASCII). Так как размер кодов символов постоянный, то хранение можно представить как массив. Соответственно доступ к элементам O(1).

Строка, представленная как префиксное дерево (по условию Фано) – неравномерный код (UTF-8). Так, например, если в байте первый бит 1, значит символ занимает еще и следующий байт. Из-за неравномерности размера кода символа доступ последовательный О(N).



Не пользоваться С-строками и функциями, а стандартным stringом плюсов!!!

Расстояние Левенштейна – алгоритм вычисления расстояния между строчками. Для начала рассмотрим расстояние Хаффмана – между строчками АБВТ и БВВТ = 2. В расстоянии Левенштейна будут рассматриваться расстояния с учетом добавления, удаления, замены символа. Из строчки АБВТ добавление АББВТ, удаление ББВТ, замена БВВТ.

Расстояние Хаффмана для строк одинаковой длины, расстояние Левенштейна показывает минимальное расстояние между двумя строчками разной длины.

**1) Задача на сравнение идентичности двух строк (алгоритмы поиска редакционных изменений)**.

Используется для нахождения вероятного правильного слова при опечатках. Может решаться способами вероятностного совпадения (для избранных символов проверить идентичность в разных местах).

Найдем расстояние между строками “string” и “master”. Сразу укажем крайние случаи – расстояние между пустой строкой a и стройкой b = size(b).

Начнем сравнивать буквы двух слов по отдельности с конца. Если эти буквы идентичны, тогда расстояние между словами будет равняться как расстояние между strin\_ и maste\_. Если же буквы не совпадают тогда расстояние будет равняться минимуму от (strin\_ и master, string и maste\_) + 1. Тогда итоговое расстояние будет min(от трех расстояний) + 1.

Сводя к формуле получим, что редакционное расстояние Lij между словом a и b размером 6 символов (L66):

Реализация в файле Levenst\_Vagner-Fisher.

**2) Задача поиска подстроки в строке (алгоритм Кнута-Мориса-Пратта)**

Допустим, перед нами стоит задача найти подстроку “ааааб” в длинной строке. Тогда алгоритм наивного поиска заключается в том, что мы сравниваем с первой буквы подстроку и букву строки, и при нахождении несостыковки перемещаем указатель на первую букву строки на 1 вправо. Сложность будет произведением размера строки на подстроку.

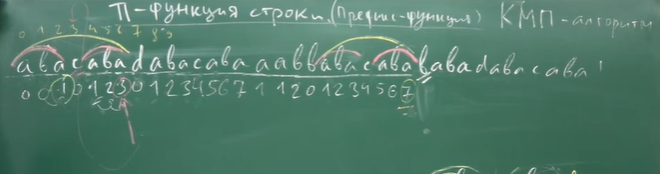
Оптимальным решением задачи будет нахождение П-функции алгоритмом КМП. Введем для строки понятие собственного суффикса – конец строки, не полностью совпадающий со строкой. Например, в строке “abc” суффиксом является “c”, “bc”, но не “abc”.

Тогда определение П-функции будет: максимальная длина собственного суффикса, совпадающего с префиксом. Это означает что в строке “aba” суффикс и префикс 'a' совпадают, а значит П-функция = 1.

Допустим есть подстрока “abacaba”, пф-ция =3, тогда допустим, что после нее у нас есть еще символы. Возникает вопрос что будет с пф-цией при различных вариантах встречи символов. 1 – abacabac – пф-ция в данном случае увеличилась на 1 и стала равна 4.

2 – abacabab – сначала мы проверили не равняется ли новая буква центральной ‘c’, и только после этого вычислили новую п-фцию.

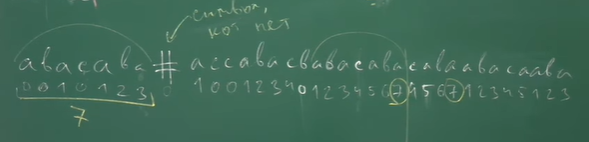
Таким образом выполним пример. Будем идти по строке слева-направо, захватывая таким образом все большую часть строки и вычислять пф-цию для нее. Снизу пишем получившееся значения пф-ции, сверху отсчитываем позицию символа.



Алгоритм в том, что мы при каждом новом символе заглядываем в “верхний” префикс и смотрим у него совпадение буквы, если нет, спускаемся в нижний уровень префикса, и так пока не найдем совпадение, либо просто обнуляем пф-цию.

РЕАЛЬНОЕ ПРИМЕНЕНИЕ.

Найти подстроку в строке, все вхождения. Тогда мы первым делом вписываем подстроку, после нее символ, который однозначно не встретится в тексте (чтобы пф-ция подстроки const). Далее пойдет сама строка в которой производится поиск. Начинаем вычислять пф-цию для вообще всей строки. Далее все как в примере выше. После вычисления пф-ции для всей строки, зная размер искомой подстроки, находим все значения пф-ции равные этому значению, это и будут искомые подстроки в строке.



Таким образом скорость алгоритма будет O(N+M).

Реализация алгоритма в файле KMP\_algo.

**ХЭШИРОВАНИЕ**

1) Хэш-функция - это математический алгоритм, который отображает данные произвольного размера в битовый массив фиксированного размера.

Результат, производимый хеш-функцией, называется «хеш-суммой» или же просто «хешем», а входные данные часто называют «сообщением».

Для идеальной хеш-функции выполняются следующие условия:

а) хеш-функция является детерминированной, то есть одно и то же сообщение приводит к одному и тому же хеш-значению

b) значение хеш-функции быстро вычисляется для любого сообщения

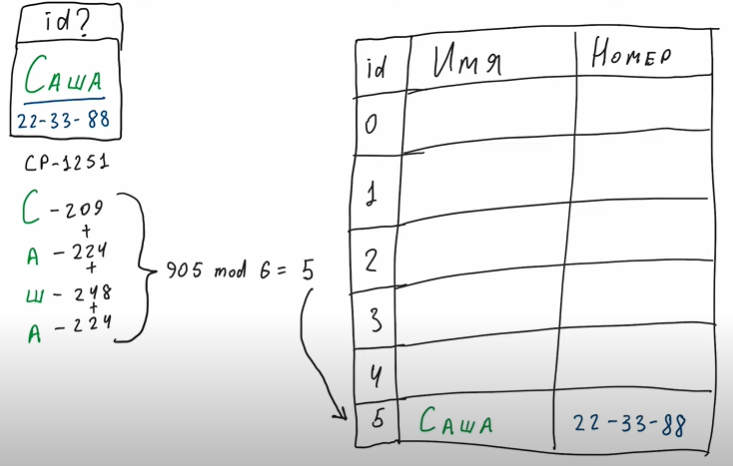
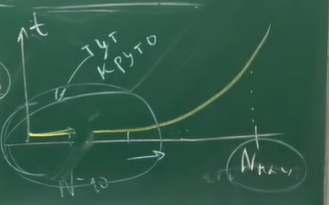
c) невозможно найти сообщение, которое дает заданное хеш-значение

d) невозможно найти два разных сообщения с одинаковым хеш-значением

e) небольшое изменение в сообщении изменяет хеш настолько сильно, что новое и старое значения кажутся некоррелирующими

2) Хеш-таблица — это контейнер, который используют, если хотят быстро выполнять операции вставки/удаления/нахождения. В языке C++ хеш-таблицы скрываются под флагом unoredered\_set и unordered\_map.

Мы определяем функцию хеширования, которая по каждому входящему элементу будет определять натуральное число. А уже дальше по этому натуральному числу мы будем класть элемент в (допустим) массив. Тогда имея такую функцию мы можем за *ОЖИДАЕМОЕ* O(1) обработать элемент.

Мы хотим пользоваться хэш-таблицей когда она почти пустая, потому что при увеличении значений, увеличивается время работы. Т.е брать размер таблицы с запасом относительно предполагаемого размера количества значений. Для этого при достижении половины занимаемой памяти будем осуществлять перехэширование (выделения x2-x10 памяти от исходной). В этот момент добавление элемента будет занимать значительное время, т.к. выделится новая память, перенесение таблицы.

3) Проблема коллизии

Естественно, возникает вопрос, почему невозможно такое, что мы попадем дважды в одну ячейку массива, ведь представить функцию, которая ставит в сравнение каждому элементу совершенно различные натуральные числа просто невозможно. Именно так возникает проблема коллизии, или проблемы, когда хеш-функция выдает одинаковое натуральное число для разных элементов.

Существует несколько решений данной проблемы: *метод* *цепочек* (по сути хранить по ключу сразу список значений и в случае коллизий, просто добавлять еще одни элемент) и метод *открытой* *адресации* (если ячейка по ключу занята, искать до нахождения свободной и записать туда результат).

Метод открытой адресации заключается в том, что если мы берем хэш ключа и переходим в ячейку, а там занято, тогда мы идем вправо до того момента пока не найдем пустую ячейку, соответственно кладем туда ключ значение.

Поиск же будет осуществляться так, берем хэш от ключа, и идем в ячейку, если значения нет, идем вправо до первой пустой ячейки, соответственно значения нет. Однако при этом теперь мы не можем удалять из ячеек пары, т.к. нарушится логика. Поэтому добавим параметр флага удаления значения, если флаг 1 – значение удалено.

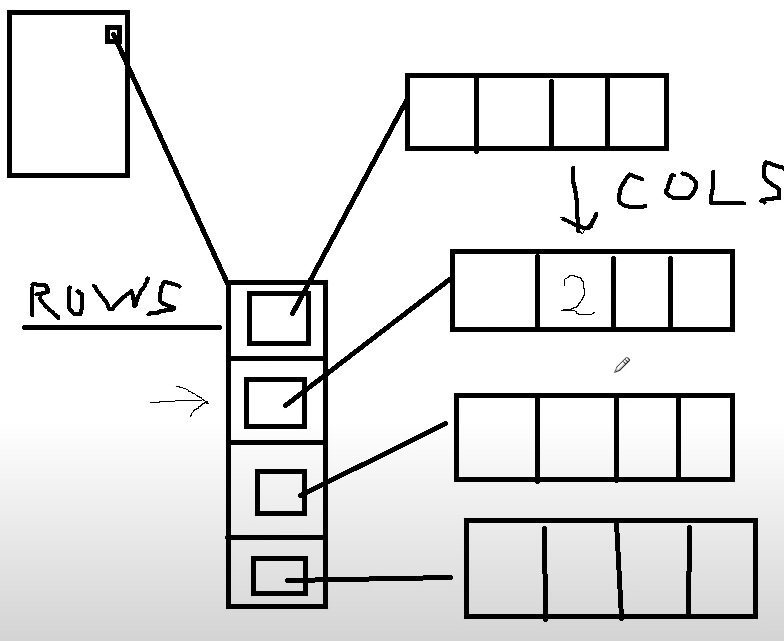
Время работы хэш-таблицы при перехэшировании без учета – O(1) для добавления, удаления, поиска.

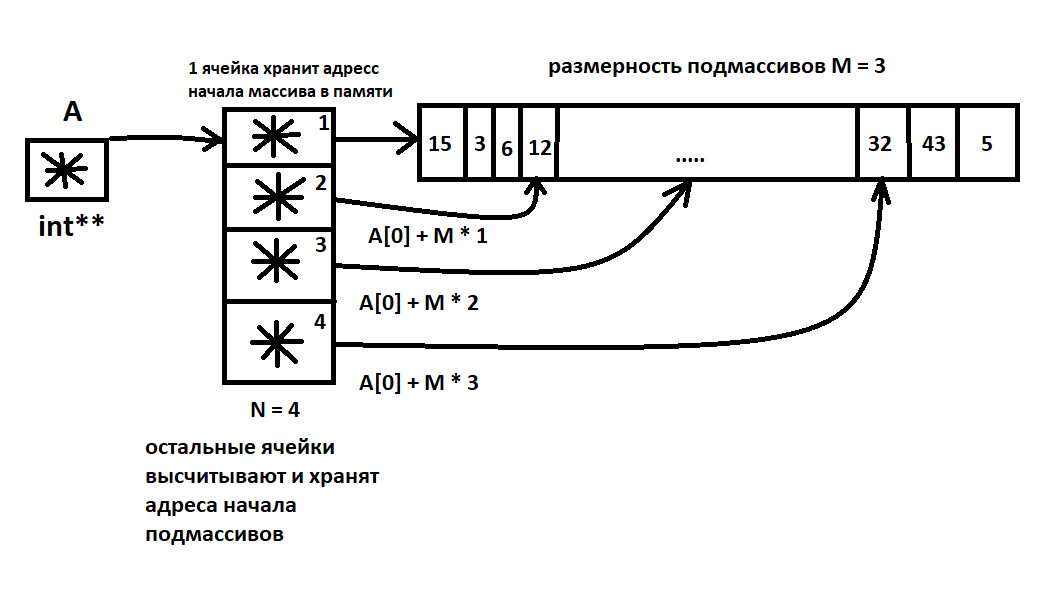
Реализация хэш-таблицы - <https://habr.com/ru/articles/509220/>.

**СТРУКТУРЫ ДАННЫХ**

**1) Динамический двумерный массив**

По сути в памяти представлен как указатель на массив указателей на одномерные массивы данных в первом случае или как указатели на части одномерного массива в памяти во втором. Не забывать чистить память при удалении.

****

****

Реализация создания динам двумер массива на С++:

1.

Int N = 3, M = 5;

Int\*\* A = new int\*[N]

For(int I = 0; I < N; i++){

A[i] = new int[M]; }

2.

Int N = 3, M = 5;

Int\*\* A = new int\*[N]

A[0] = new int[N\*M]; // Сразу выделяем в памяти цельный блок под двумерный массив, адрес которого будет хранится в нулевой ячейке

For(int I = 1; I < N; i++){

A[i] = A[0] + M\*i; // в остальные ячейки высчитываем и кладем указатели на “виртуальные подмассивы”

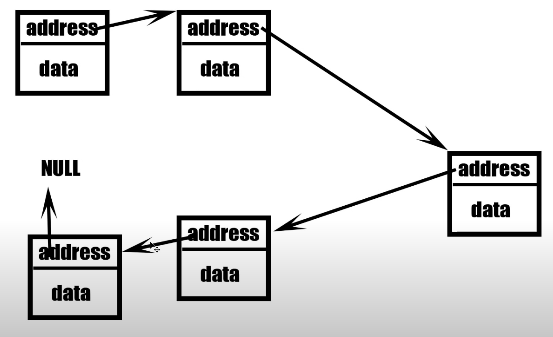
} // плюсом создания такого динам двумерн массива является то, что можно обращаться к элементам арифметикой указателей, т.к. массив хранится в динам памяти цельным куском, в отличии от первой реализации

*Минусы: при работе с новыми элементами в массиве, придется пересоздавать и копировать массив для расширения/уменьшения.*

*Плюсы: быстрый доступ к содержимому ячеек за счет арифметики указателей.*

**2) Односвязный (однонаправленный) список**

Совокупность элементов в котором каждый элемент (Node) знает адрес следующего элемента. В отличии от массива, Node хранятся хаотично в памяти.



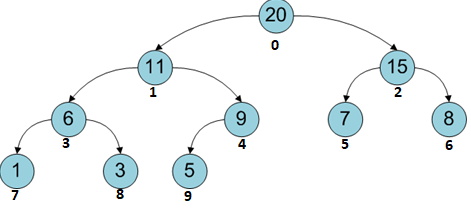
Адрес первого элемента (head) хранится в реализуемом списке в private. С него и начинается список. Чтобы обойти весь список нужно пройтись по каждой Node и перейти по хранимому в ней адресу следующего элемента. Последняя Node списка хранит указатель на nullptr. Реализация в файле LinkedList.

*Плюсы: быстрая работа с элементами (удаление, добавление элементов) т.к. просто изменяем поля адреса элементов в Node.*

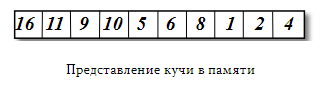
*Минусы: медленное получение доступа (в сравнении с массивом) к конкретному элементу т.к. приходится обходить список с head’а.*

**3) Двоичная куча (Binary Heap)**

Двоичная куча представляет собой полное бинарное дерево логарифмической высоты, для которого выполняется основное свойство кучи: приоритет каждой вершины больше приоритетов её потомков. Слои заполняются последовательно сверху вниз и слева направо, без «дырок».



Хранить кучу будем в виде массива arr, где у корня индекс равен 0, а у вершины k индексы ее детей равны (2k+1) и (2k+2).



Поддерживает следующие операции:

1. Нахождение минимума за O(1).

Всегда нулевой элемент.

2. Удаление минимума за O(log n).

При удалении минимального элемента – нулевого, структура дерева нарушается, поэтому мы берем самый последний элемент дерева и переносим его в вершину. Теперь нужно вставить его в правильном порядке. Мы сравниваем новую вершину с потомками, свапаемся местами с наименьшим, переходим на уровень ниже и так пока не выполнится неравенство. Такой метод называется просеивание сверху-вниз (sift down).

3. Добавление нового элемента в кучу за O(log n)

Элемент будет добавляться в крайнюю нижнюю свободную ячейку, при этом будет нарушено свойство того что элемент больше(меньше) родителя. Для того чтобы восстановить структуру, будем делать просеивание дерева снизу-вверх (sift up). Берем новое число и сравниваем его с родителем, пока неравенство не будет выполнено.

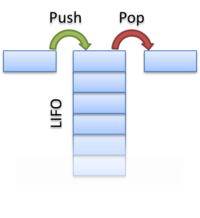
*Плюсы: нахождение минимального(максимального) константное время, добавление элемента за логарифм. Не требует дополнительной памяти.*

*Минусы: нельзя искать элементы, точнее это будет долго O(n).*

Реализация класса и функции heapify в файлах Heap и heapify.

**4) Стек (stack)**

Cтруктура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (LIFO). Для стека с n элементами требуется O(n) памяти, так как она нужна лишь для хранения самих элементов.



Обладает операциями:

- push (запись в стек) — операция вставки нового элемента за O(1);

- pop (снятие со стека) – операция удаления верхнего элемента за O(1);

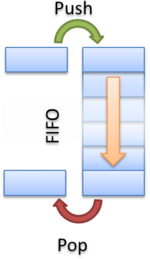
- empty – проверка стека на наличие в нем элементов.

Лучше реализовывать стек на динамическом массиве, в результате чего появляется существенное преимущество над обычной реализацией: при операции push мы никогда не сможем выйти за границы массива, тем самым избежим ошибки переполнения.

Создадим вектор и определим операции стека на нём. В функции push перед тем, как добавить новый элемент, будем проверять, не нужно ли расширить массив вдвое, а в pop, перед тем, как изъять элемент из массива, — не нужно ли вдвое сузить размер вектора.

**5) Очередь (Queue)**

Структура данных, добавление и удаление элементов в которой происходит путём операций push и pop соответственно. Притом первым из очереди удаляется элемент, который был помещен туда первым, то есть в очереди реализуется принцип «первым вошел — первым вышел» (англ. first-in, first-out — FIFO). У очереди имеется голова (англ. head) и хвост (англ. tail). Когда элемент ставится в очередь, он занимает место в её хвосте. Из очереди всегда выводится элемент, который находится в ее голове.



Обладает операциями:

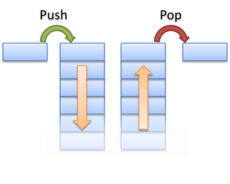
- push (запись в стек) — операция вставки нового элемента за O(1);

- pop (снятие со стека) – операция удаления верхнего элемента за O(1);

- empty – проверка стека на наличие в нем элементов;

- size - получение количества элементов в очереди.

Очередь можно реализовать на двух стеках leftStack и rightStack. Поступим следующим образом: leftStack будем использовать для операции push, rightStack для операции pop. При этом, если при попытке извлечения элемента из rightStack он оказался пустым, просто перенесем все элементы из leftStack в него (при этом элементы в rightStack получатся уже в обратном порядке, что нам и нужно для извлечения элементов, а leftStack станет пустым).



**6) Вектор**

<https://education.yandex.ru/handbook/cpp/article/vectors-and-strings>

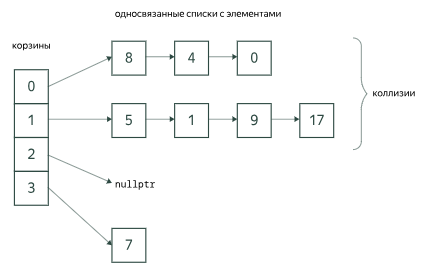
**АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ**

Ассоциативные контейнеры сопоставляют ключам некоторые значения. В стандартной библиотеке есть ассоциативные контейнеры, основанные на сбалансированных деревьях поиска (map, set) и контейнеры, основанные на хеш-таблицах (unordered\_map, unordered\_set). В этих контейнерах ключи уникальны, то есть, не могут повторяться. Также существуют и multi-версии этих контейнеров, в которых допускаются повторы ключей.

**7) Хэш-таблица (unordered\_map, unordered\_set)**

Само название этого класса подчёркивает, что данные будут храниться не упорядоченными по ключу. Предполагается, что для каждого ключа определена хеш-функция (по умолчанию std::hash<Key>()), а по ней вычисляется номер корзины (bucket), в которую должен попасть ключ.

Случай, когда два разных ключа оказываются в одной корзине, называется коллизией. В С++ для разрешения коллизий используется метод цепочек, то есть, внутри одной корзины все элементы выстраиваются в односвязный список.



Если хеш-функция достаточно равномерна и корзин достаточно много, то в среднем время поиска, добавления и удаления элементов для unordered\_map будет константным (O(1)).

Интерфейс unordered\_map специально сделан похожим на интерфейс map. Нам будет достаточно заменить только заголовочный файл и имя контейнера.

У контейнера unordered\_map есть функция max\_load\_factor, которая задаёт максимально допустимое соотношение между числом элементов и количеством корзин. По умолчанию эта величина равна единице, так что unordered\_map пытается в среднем вообще избежать коллизий. Но это не означает отсутствия коллизий в отдельных корзинах.

Если при вставке очередного элемента среднее число элементов в корзинах превышает этот порог, число корзин автоматически увеличивается и происходит рехеширование. Чем-то это напоминает реаллокацию у вектора.

Если нам заранее известно финальное количество ключей, то можно вызвать заранее функцию reserve и избежать лишних рехеширований при вставках.

Контейнер std::unordered\_set похож на unordered\_map по внутреннему устройству, но он хранит только ключи, без ассоциированных значений.

**ДЕРЕВЬЯ**

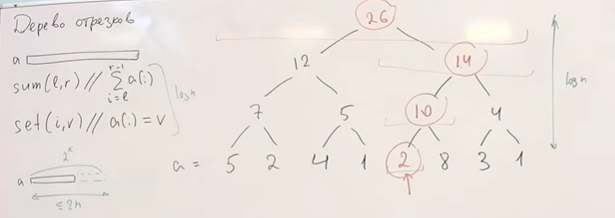
**1) Дерево отрезков**

Допустим, мы хотим вычислять сумму элементов на отрезке. В случае статического массива это легко реализуется префиксной суммой. Вкратце заводим дополнительный массив сумм, где new\_arr[i] = new\_arr[i-1] + arr[i];

<https://silvertests.ru/GuideView.aspx?id=31909>

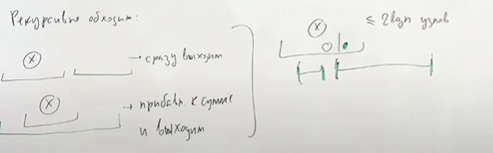
Однако, если массив динамический, то нам понадобится уже реализация дерева отрезков.

Допустим у нас есть массив значений, для каждой соседней пары значений посчитаем их сумму, для получившихся пар сумм посчитаем их сумму, таким образом корнем дерева будет являться сумма всего массива. В такой реализации вставка и подсчет суммы будет занимать log(N).



Операция вставки будет осуществляться с пересчетом родительских листьев. Операция вычисления суммы на отрезке имеет несколько вариантов.

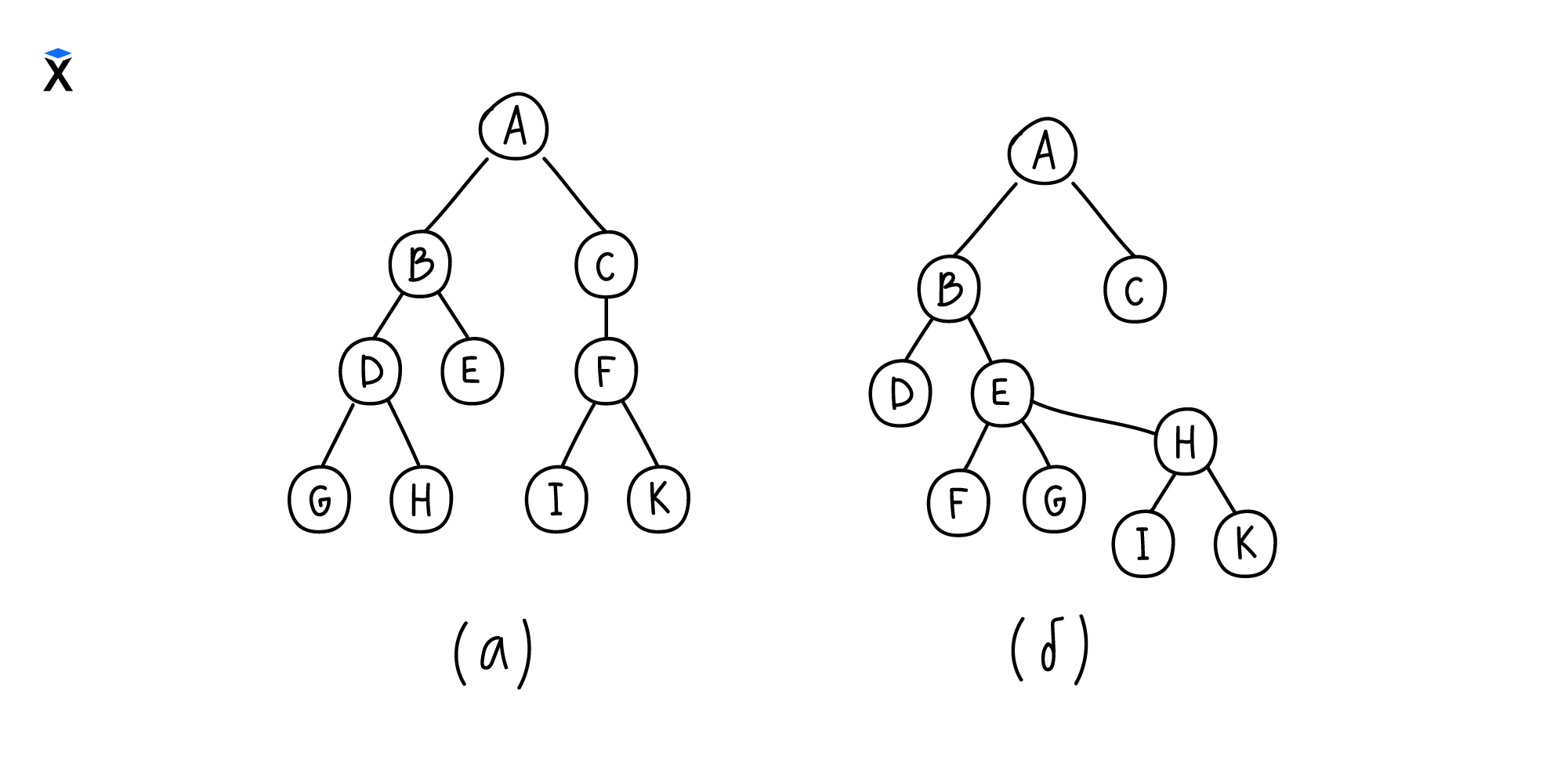
Будем рекурсивно идти вниз по дереву и искать принадлежащие отрезку суммы. Спускаться вниз будем пока лист либо полностью лежит в отрезке, либо полностью не принадлежит отрезку, тогда мы либо добавляем сумму, либо нет, переходим в другую ветку рекурсии. Если же не сработало ни то ни другое, тогда отрезок лежит частично. Если такое произошло, тогда отрезок листов содержит одну из границ искомого отрезка.



Реализацию можно писать как на ООП с Node’ами, так и массивом как с кучей. В нашем случае будет куча. Обе операции будут выполняться рекурсивно. В set будем передавать позицию, новое значение, а также текущий узел и границы текущего отрезка.

**8) Сбалансированное (Бинарное) дерево поиска (BST)**

В С++ std::set и std::map реализованы на данной структуре данных. Бинарное дерево или двоичное дерево — это дерево, в котором у каждого из его узлов не более двух дочерних узлов. При этом каждый дочерний узел тоже представляет собой бинарное дерево.



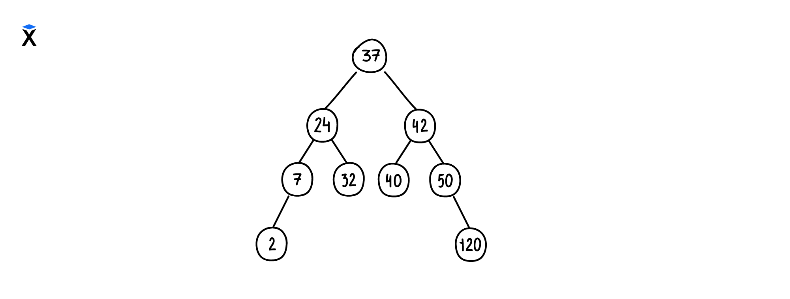
Дерево (а) — бинарное. У каждого его узла не более двух дочерних узлов, у каждого из которых тоже не более двух дочерних. Например, узлы E, G, H, I и K — листовые, значит, у них ноль дочерних узлов. Как только правило двух дочерних нарушается, то дерево перестает относиться к классу бинарных. Так, дерево (б) не является бинарным, так как у узла E три дочерних узла.

Бинарные деревья поиска отличаются от обычных бинарных деревьев тем, что хранят данные в отсортированном виде. Хранение значений внутри бинарного дерева поиска организовано в следующем виде:

- Все значения в узлах левого дочернего поддерева меньше значения родительского узла;

- Все значения в узлах правого дочернего поддерева больше значения родительского узла;

- Каждый дочерний узел тоже является бинарным деревом поиска.

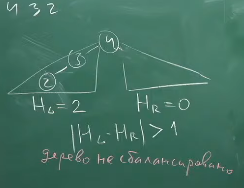
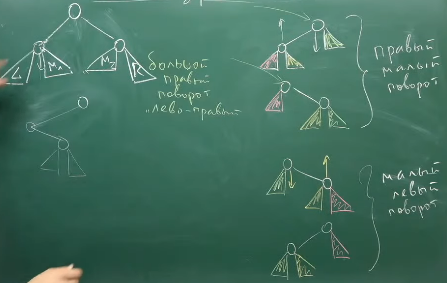


Благодаря такой структуре хранения данных поиск узла в бинарном дереве поиска занимает O(logn). Это значительно меньше, если хранить значения в списках — O(N).

Если использовать отсортированный массив для хранения данных, скорость поиска элементов сравняется. Но при оценке времени вставки хранение в массиве значительно проигрывает работе с деревьями – O(N) против O(logn) соответственно.

Такая высокая эффективность поиска в бинарном дереве поиска наблюдается только при сохранении его в сбалансированном состоянии — когда все уровни, кроме последнего полностью заполнены. Это значит, что любое добавление или удаление вершины может потребовать полное перестроение дерева.

Балансировка дерева – ключевая составляющая, поддерживающая асимптотику BST стабильной logn. Если высота правого и левого поддерева отличаются больше чем на 1, то дерево не сбалансировано. Следовательно, нужны алгоритмы балансировки деревьев.

[Бинарные деревья | Алгоритмы на деревьях (hexlet.io)](https://ru.hexlet.io/courses/algorithms-trees/lessons/binary/theory_unit?ysclid=ltcwm1g5li699748520)