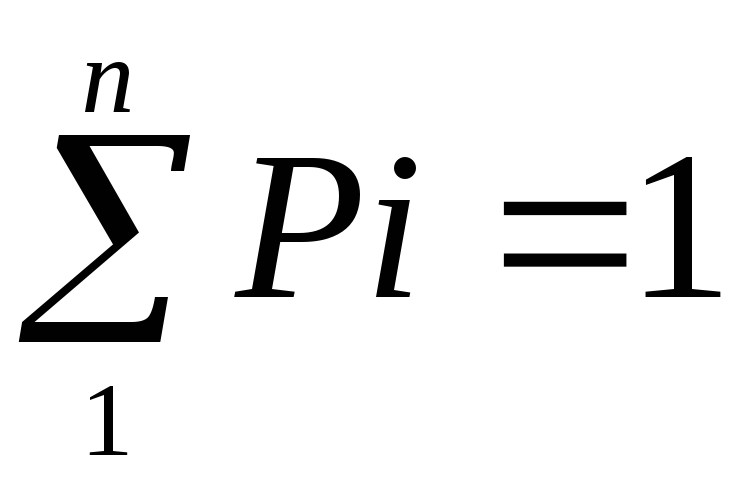
# 

# Деревья оптимального поиска

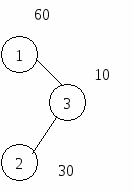
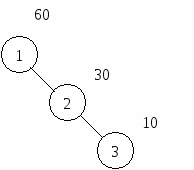
До сих пор предполагалось, что частота обращения ко всем вершинам дерева поиска одинакова. [Однако встречаются ситуации](http://topuch.ru/2-mikrocitogeneticheskie-sindromi/index.html), когда известна информация о вероятностях обращения к отдельным ключам. Обычно для таких ситуаций характерно постоянство ключей, т.е. в дерево не включаются новые вершины и не исключаются старые и структура дерева остается неизменной. Эту ситуацию иллюстрирует сканер транслятора, который определяет, является ли каждое слово программы (идентификатор) служебным. Статистические измерения на сотнях транслируемых программ могут в этом случае дать точную информацию об относительных частотах появления в тексте отдельных ключей.



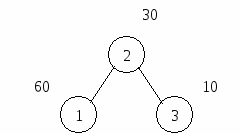
Припишем каждой вершине дерева *Vi* вес *wi*, пропорциональный частоте поиска этой вершины (например, если из каждых 100 операций поиска 15 операций приходятся на вершину *V1*, то *w1*=15). Сумма весов всех вершин дает вес дерева *W*. Каждая вершина *Vi* расположена на высоте *hi*, **корень расположен на высоте 1**. Высота вершины равна количеству операций сравнения, необходимых для поиска этой вершины. Определим средневзвешенную высоту дерева с *n* вершинами следующим образом:

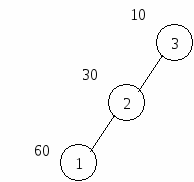
***hср=(w1h1+w2h2+…+wnhn)/W***. Дерево поиска, имеющее минимальную средневзвешенную высоту, называется *деревом оптимального поиска* (ДОП).

**Пример**. Рассмотрим множество из трех ключей *V1*=1, *V2*=2, *V3*=3 со следующими весами: *w1*=60, *w2*=30, *w3*=10, *W*=100. Эти три ключа можно расставить в дереве поиска пятью различными способами.

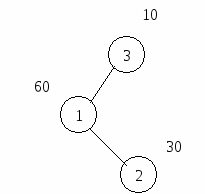


hср=1.5 hср=1.7





hср=1.7 hср=2.5



hср=2.2

Рисунок 57 Различные деревья поиска с вершинами *V1*=1, *V2*=2, *V3*=3

Задача построения ДОП может ставится в двух вариантах:

* Известны вершины и их веса.
* Вес вершины определяется в процессе работы. Например, после каждого поиска вершины ее вес увеличивается на 1. В этом случае необходимо перестраивать структуру дерева при изменении весов.

Далее будем рассматривать задачу построения ДОП с фиксированным набором ключей и их весов

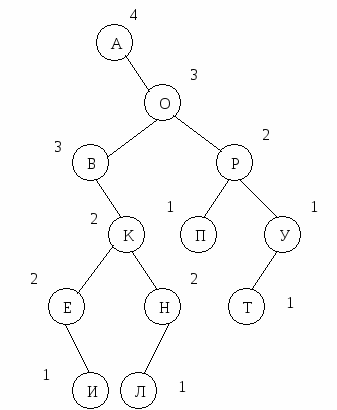
**Алгоритм ПРИБЛИЖЕННОГО построения ДОП 1**

* Отсортировать вершины по вероятностям
* В цикле
* Выбираем элемент с максимальной вероятностью
* Добавляем в обычное БДП

Рассчитаем средневзвешенную высоту построенного дерева

P=4**.**1+3**.**2+3**.**3+2**.**3+2**.**4+1**.**4+1**.**4+2**.**5+ +2**.**5+1**.**5+1**.**6+1**.**6=78

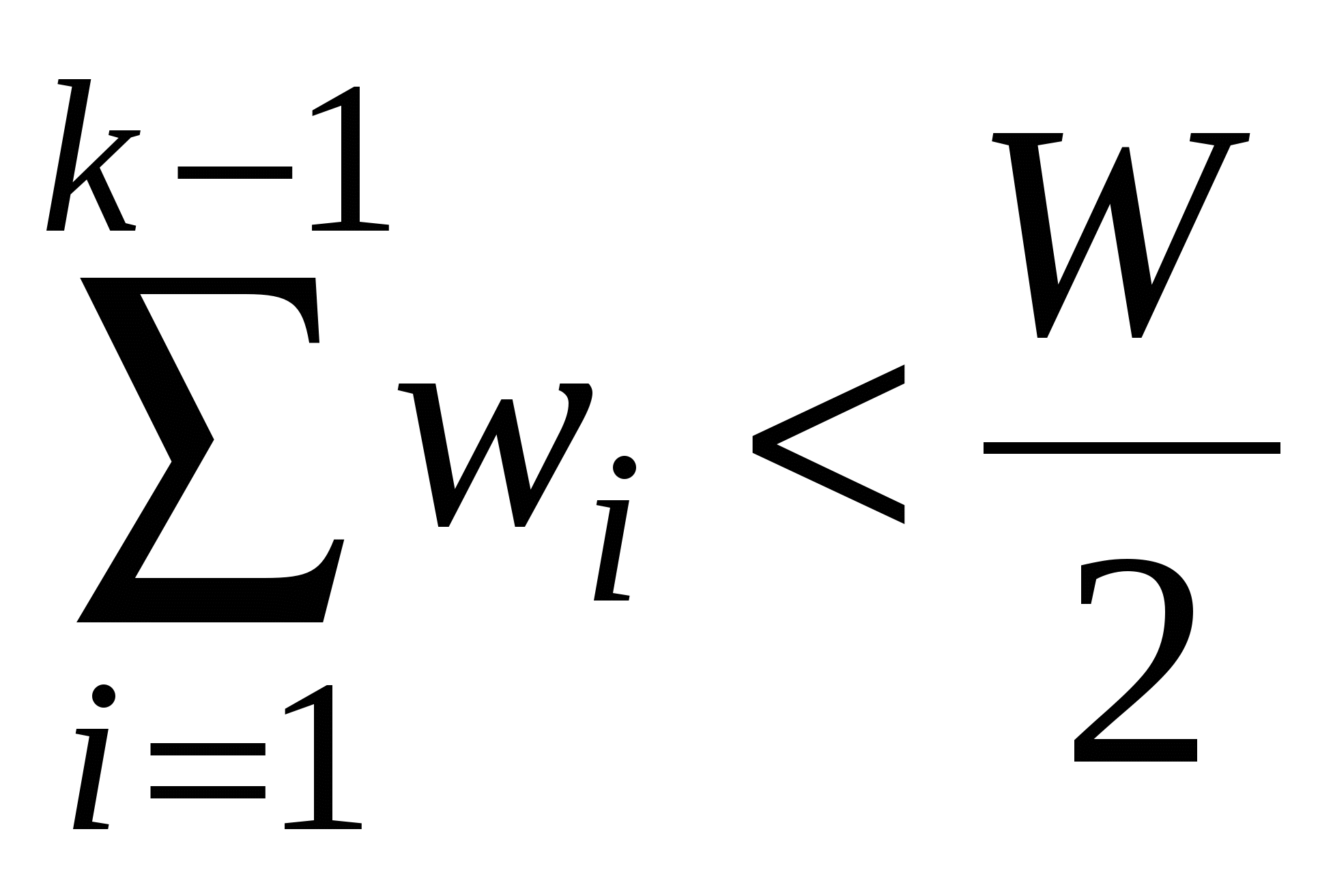
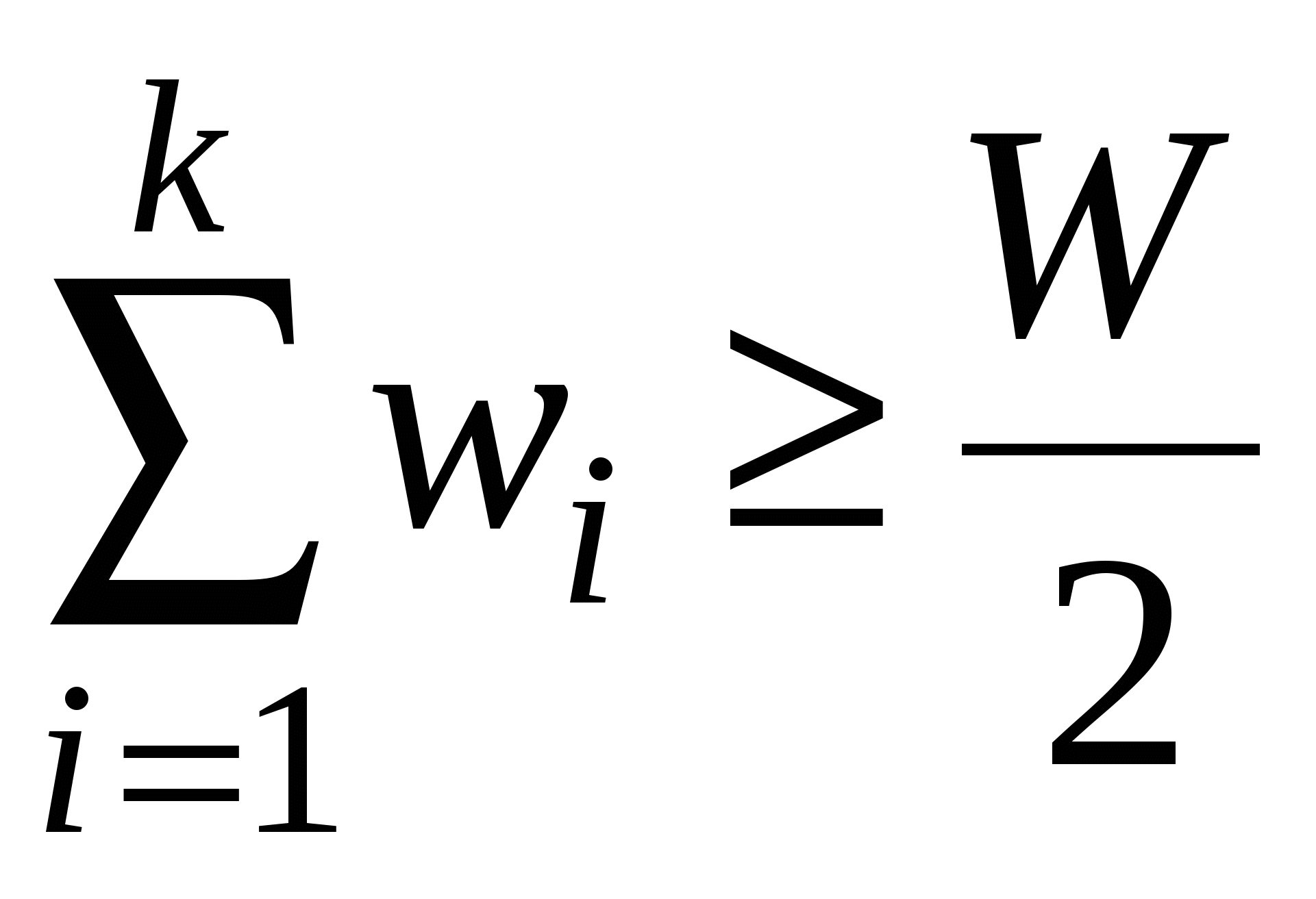
hср=P/W=78/23=3,39

**

*Рисунок 59 Дерево, построенное приближенным алгоритмом А1*

**Алгоритм ПРИБЛИЖЕННОГО построения ДОП 2**

Второй алгоритм использует предварительно упорядоченный по ключу массив вершин. В качестве корня выбирается такая вершина, что разность весов левого и правого поддеревьев была минимальна. Для этого путем последовательного суммирования весов определим вершину Vk, для которой справедливы неравенства:

* и .*

Тогда в качестве "центра тяжести" может быть выбрана вершина Vk, Vk-1 или Vk+1, т. е. вершина, для которой разность весов левого и правого поддерева минимальна.

Далее действия повторяются для каждого поддерева

Алгоритм

Отсортировать(массив)  
дерево = пусто  
ПостроитьДОП(дерево, массив, 0, массив.размер-1)

//--------------------------------//

ПостроитьДОП(ДОП корень, Вершина[] массив, int левая, int правая)

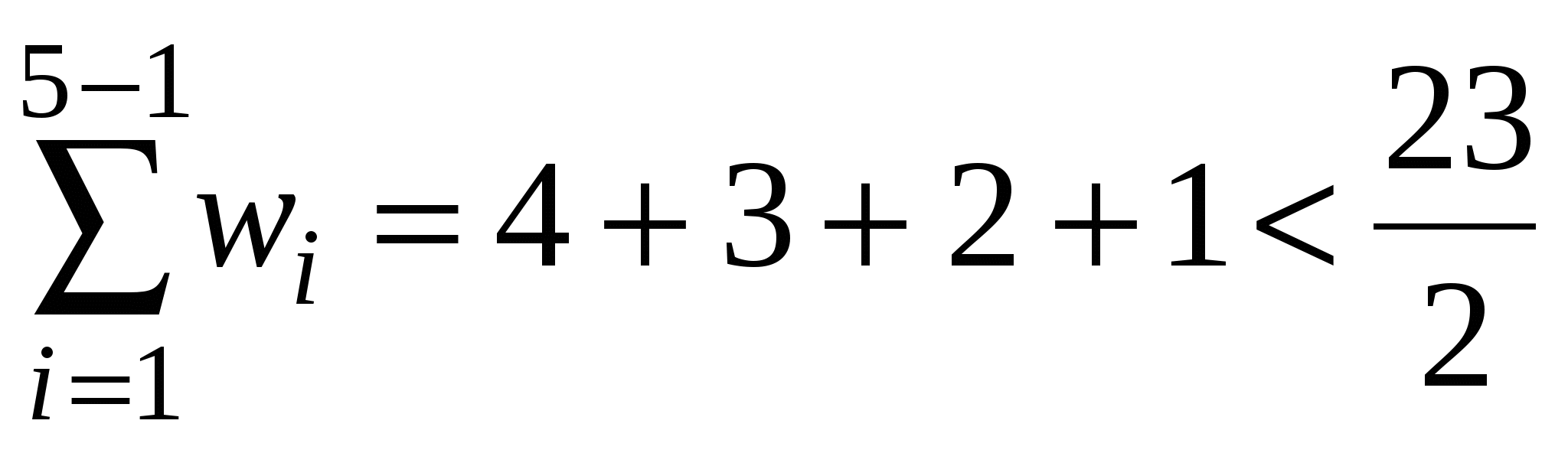
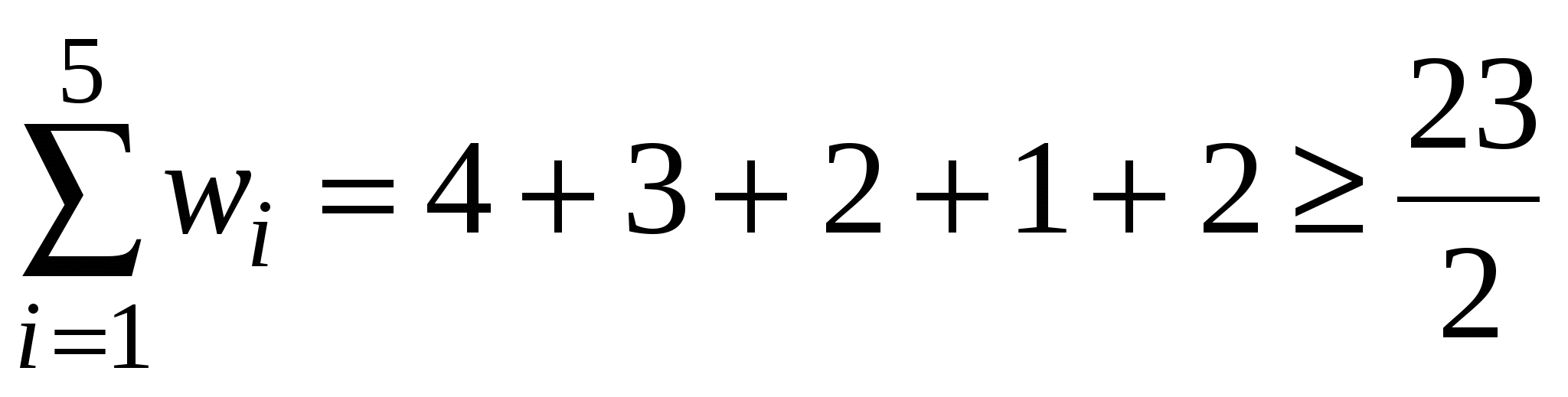
n = НайтиКорень(массив, левая, правая)  
 Вставить(корень, массив[n])  
 ПостроитьДОП(массив[n], массив, левая, n-1)  
 ПостроитьДОП(массив[n], массив, n+1, правая)

**Пример.** Рассмотрим процесс построения приближенного ДОП алгоритмом А2 для строки символов из предыдущего примера. Предварительно упорядочим символы по алфавиту.

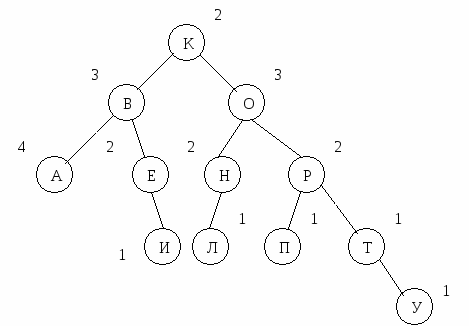
Таблица 4 [Упорядоченный набор вершин](http://topuch.ru/1-ponitie-n-mernogo-vektora-osnovnie-opredeleniya/index.html)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| А | В | Е | И | К | Л | Н | О | П | Р | Т | У |
| 4 | 3 | 2 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 1 |

В качестве корня дерева выбираем вершину *V5=*К, поскольку

, .

Все вершины левее вершины *V5* образуют левое поддерево, вершины правее *V5* – правое поддерево. Далее алгоритм применяется для каждого из поддеревьев в отдельности. Посчитаем средневзвешенную высоту построенного дерева.

P=2**.**1+3**.**2+3**.**2+4**.**3+2**.**3+2**.**3+2**.**3+1**.**4+1**.**4+1**.**4+1**.**4+1**.**5=65

hср=65/23=2.82

Рисунок 60 Дерево, построенное приближенным алгоритмом А2

Приведем некоторые свойства рассмотренных приближенных алгоритмов:

1. Сложность алгоритмов как функция от *n*(количество элементов) зависит следующим образом: время Т = О(*n* log *n*), память М = О(*n*)
2. Дерево, построенное приближенным алгоритмом А1, равносильно случайному (с точки зрения средней высоты) при *n→∞,* т.е. алгоритм А1– плохой*.*
3. Дерево, построенное приближенным алгоритмом А2, асимптотически приближается к оптимальному (с точки зрения средней высоты) при *n→∞,* т.е. алгоритм А2 является хорошим.

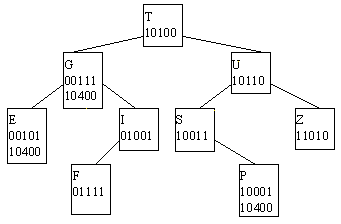
***------------------------------------------------------------------------------------------------------------------------------***

Деревья цифрового поиска

Методы цифрового поиска достаточно громоздки и плохо иллюстрируются. Поэтому мы кратко остановимся на наиболее простом механизме - бинарном дереве цифрового поиска. Как и в деревьях, обсуждавшихся в предыдущих разделах, в каждой вершине такого дерева хранится полный ключ, но переход по левой или правой ветви происходит не путем сравнения ключа-аргумента со значением ключа, хранящегося в вершине, а на основе значения очередного бита аргумента.

Поиск начинается от корня дерева. Если содержащийся в корневой вершине ключ не совпадает с аргументом поиска, то анализируется самый левый бит аргумента. Если он равен 0, происходит переход по левой ветви, если 1 - по правой. Если не обнаруживается совпадение ключа с аргументом поиска, то анализируется следующий бит аргумента и т.д., пока либо не будут проверены все биты аргумента, или мы не наткнемся на вершину с отсутствующей левой или правой ссылкой.

На рисунке 4.16 показан пример дерева цифрового поиска для некоторых заглавных букв латинского алфавита. Считается, что буквы кодируются в соответствии с кодовым набором ASCII, а для их представления и поиска используются 5 младших бит кода. Например, код буквы A равен 41(16), а представляться A будет как последовательность бит 00001.

****

***Рис. 4.16.***

# Матричное дерево

**Матричное дерево - структура поиска данных, самобалансирующееся двоичное дерево с увеличенной скоростью поиска**

Основным недостатком известных самобалансирующихся двоичных деревьев является сохраняющаяся зависимость времени перестановки элементов от высоты дерева. При большой высоте дерева и частом обновлении состава его элементов общая эффективность применения подобных структур данных в процессе поиска информации существенно снижается.

В целях преодоления этого недостатка предлагается новый вид самобалансирующегося двоичного дерева поиска – матричное дерево. Термин «матричное» использован для отражения того факта, что ключ каждого элемента дерева можно разложить на два сомножителя, определяющих место элемента – его уровень относительно уровня листьев дерева и занимаемую позицию относительно самого левого элемента на этом уровне (аналогично индексам элемента матрицы).

Кроме того, матричное дерево разделяется в корневом указателе на четное и нечетное поддеревья, каждое из которых состоит из элементов, связанных в виде двоичного дерева поиска. Четное поддерево включает элементы с четными ключами, нечетное – элементы с нечетными ключами. Каждый элемент матричного дерева содержит ключ, информационное значение, правый и левый указатели на элементы-потомки.

Уровни матричного дерева нумеруются от листьев к корню, позиции элементов на одном уровне нумеруются слева направо. Ключ элемента nxy имеет индексы x и y, где x – уровень элемента относительно листьев дерева и y – позиция элемента относительно первого элемента на уровне.

1. Ключ узла можно разложить на два сомножителя:

*для четного поддерева*

*n* = 2*x* \* *z* (в случае нумерации ключей, начиная с 1)

*n* + 2 = 2*x* \* *z* (в случае нумерации ключей, начиная с 0),

где n – ключ узла, х - уровень, на котором расположен узел, z - сомножитель

*для нечетного поддерева*

*n* + 1 = 2*x* \* *z*

2. Позиция узла на уровне равна *y* = (*z* + 1) / 2

3. Количество узлов, расположенных под общей вершиной, равно *k* = 2*m*, где m – разность по модулю уровней узлов и общей вершины.

4. Позиция последнего узла на уровне, расположенного под общей вершиной, равна *yi* + *k* − 1 = *yvershiny* \* 2*m*

5. Ключ общей вершины равен среднему арифметическому ключей первого и последнего узлов, расположенных под общей вершиной *nvershiny* = (*ni* + *ni* + *k* − 1) / 2

6. Если первый узел имеет общую вершину со вторым узлом, то эта вершина является вышележащей общей вершиной для общей вершины второго узла с третьим узлом.

## **Работа с матричными деревьями**

Алгоритм добавления или удаления узла из состава матричного дерева основан на определении общей вершины для добавляемого (удаляемого) узла и ранее включенного (сохраняемого) узла.

Добавление нового узла начинается с определения характеристики четности/нечетности ключа, уровня и позиции узла в соответствии со свойствами 1 и 2 матричного дерева. После этого в составе четного или нечетного поддерева производится поиск соседнего, ранее включенного узла.

В соответствии со свойствами 3, 4, 5 и 6 матричного дерева определяется общая вершина для нового и соседнего узлов. Если общая вершина совпадает с одним из ранее включенных узлов новый узел присоединяется к нему. В противном случае в состав матричного дерева включается дополнительный узел, являющийся общей вершиной для нового и соседнего узлов.

Дополнительный узел является резервным для присваивания в будущем значения, ассоциированного с его ключом.

Удаление узлов осуществляется в обратном порядке.



## **Сравнение с другими [сбалансированными деревьями](https://dic.academic.ru/dic.nsf/ruwiki/1139611)**

### **Высота дерева**

Степень сбалансированности матричного дерева зависит от вида последовательности, в которой ключи находились перед добавлением в дерево.

* При возрастающей/убывающей ключей последовательности с шагом, равным 1, сбалансированность матричного дерева достигает уровня сбалансированности [АВЛ-дерева](https://dic.academic.ru/dic.nsf/ruwiki/217109).
* При случайной последовательности ключей сбалансированность матричного дерева соответствует уровню сбалансированности [красно-черного дерева](https://dic.academic.ru/dic.nsf/ruwiki/217111).
* При возрастающей/убывающей последовательности ключей вида 2n матричное дерево вырождается в [линейный список](https://dic.academic.ru/dic.nsf/ruwiki/18381).

### **Поиск**

Эффективность поиска в матричном дереве зависит от распределения характеристики четности/нечетности ключей.

* При равномерном распределении характеристики поиск производится в половине от общего числа узлов за время *O*(*log n* / 2), что является лучшим показателем среди всех сбалансированных деревьев. В случае одностороннего распределения характеристики поиск производится в общем числе узлов за время
* *O*(*log n*). Корневой указатель не влияет на сравнительную эффективность поиска в матричном дереве, поскольку входит в состав любого сбалансированного дерева.

### **Вставка**

Производительность алгоритма вставки определяется количеством узлов, участвующих в процессе. У матричного дерева количество узлов, участвующих в определении общей вершины, не превышает четырех, что существенно меньше количества узлов, участвующих в поворотах двоичных поддеревьев [АВЛ-дерева](https://dic.academic.ru/dic.nsf/ruwiki/217109) и [красно-черного дерева](https://dic.academic.ru/dic.nsf/ruwiki/217111).

### **Удаление**

Производительность алгоритма удаления узлов в матричном дереве равна производительности алгоритма вставки узлов и так же эффективна по сравнению с алгоритмами удаления узлов [АВЛ-дерева](https://dic.academic.ru/dic.nsf/ruwiki/217109) и [красно-черного дерева](https://dic.academic.ru/dic.nsf/ruwiki/217111).

### **Память**

Узел матричного дерева не содержит дополнительной информации в отличие от узлов [АВЛ-дерева](https://dic.academic.ru/dic.nsf/ruwiki/217109)(разность высот двоичных поддеревьев) и [красно-черного дерева](https://dic.academic.ru/dic.nsf/ruwiki/217111) (цвет узла).

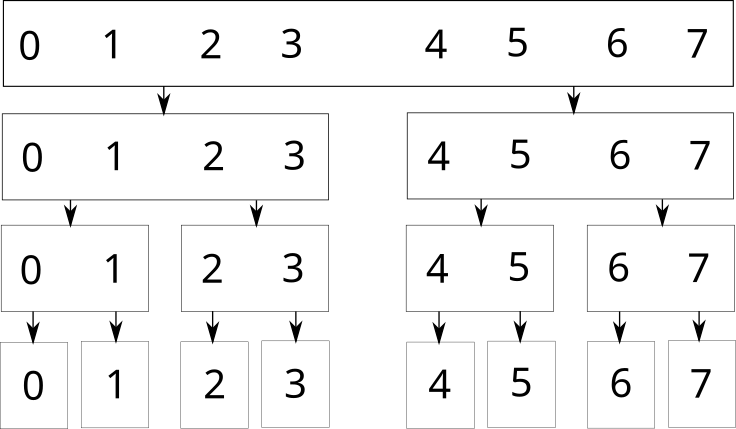
Дерево отрезков

Дерево отрезков - структура данных, позволяющая выполнять многие операции с отрезками массива за O(log ⁡N). Дерево отрезков - универсальная структура данных, для которой можно реализовать неограниченный набор операций (иногда за большую сложность: O(log^2 N)). Самая простая версия дерева отрезков позволяет находить сумму или минимум на отрезке, и изменять отдельные элементы.

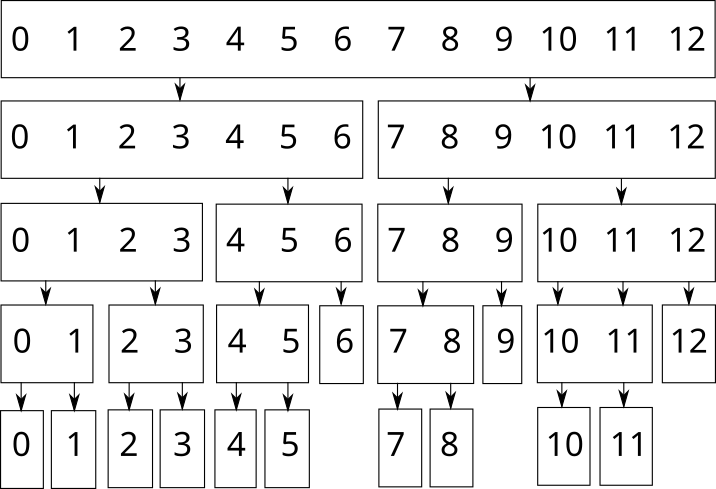
## Построение дерева отрезков

Дерево отрезков - полное бинарное дерево, в котором каждая вершина отвечает за некоторый отрезок в массиве. Корень дерева отвечает за весь массив, его две дочерних вершины - за две половины, и так далее. У каждой вершины, отвечающей за отрезок длиной больше 11, есть две дочерних вершины, отвечающих за левую и правую половины этого отрезка. Листья дерева отрезков отвечают за отдельные элементы (отрезки длиной 11).

Графически это выглядит следующим образом (для массива из 8 элементов):



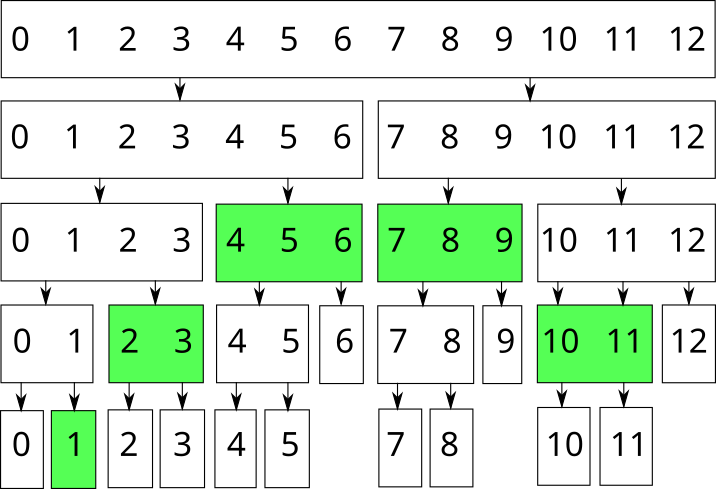
Каждый прямоугольник - вершина дерева отрезков. Если некоторая вершина отвечает за отрезок нечётной длины, то одна из её дочерних вершин будет отвечать за отрезок длиной ⌈n2⌉⌈n2⌉, а другая - за отрезок длиной ⌊n2⌋⌊n2⌋. Например, так выглядит дерево отрезков для массива из 13 элементов:



Для массива из nn элементов дерево отрезков имеет около 2n2n вершин (n+n2+n4+…n+n2+n4+…), а его высота равна порядка lognlog⁡n.

Главное свойство дерева отрезков, на котором и строятся все алгоритмы работы с ним: **любой непрерывный отрезок в массиве из** nn **элементов можно представить с помощью около** 2 logn ⁡n **вершин в дереве отрезков.**

Например, отрезок [1;11][1;11] в массиве длиной 1313 можно представить с помощью следующих вершин:



## Дерево отрезков для поиска суммы

Одна из простейших функций, которую можно считать за O(log⁡N) с помощью дерева отрезков - сумма.

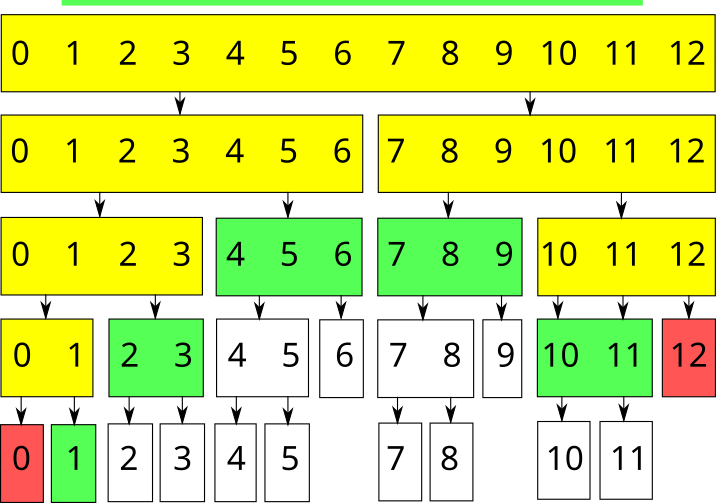
При построении дерева в каждой его вершине сохраним сумму на соответствующем отрезке (построение будет вестись рекурсивно, поэтому достаточно просто сложить суммы на двух дочерних отрезках). Затем каждый запрос суммы на отрезке будем разбивать на 2logN2log⁡N отрезков, и находить сумму на каждом из них за O(1)O(1), используя предпросчитанные значения. Таким образом сложность запроса суммы равна O(logN)O(log⁡N).

Кроме запросов суммы к дереву отрезков также могут поступать запросы на изменение отдельных элементов (модификацию). Заметим, что от изменения одного элемента значение суммы изменится в одной вершине на каждом уровне дерева отрезков (в которую входит этот элемент). Пересчитаем значения суммы (опять же, рекурсивно) только в этих вершинах. Таким образом сложность запроса модификации равна высоте дерева, или O(logN).

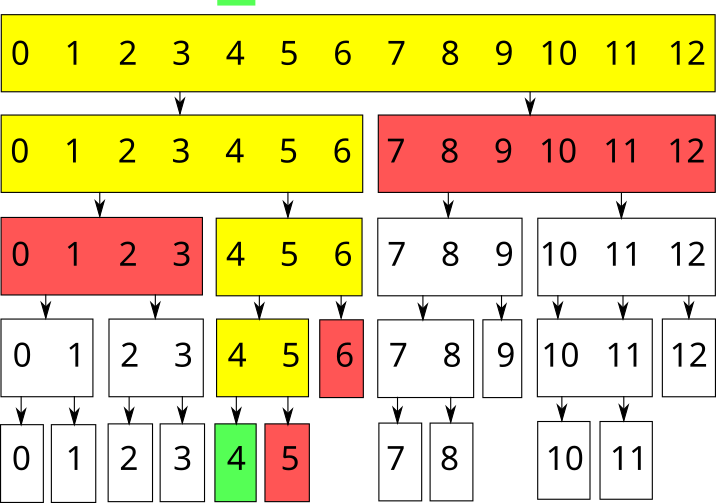
Для реализации запроса суммы и запроса модификации обход дерева реализуется с помощью одного и того же алгоритма, основанного на DFS. Пусть границы нашего запроса [L;R][L;R] (в случае запроса модификации L=RL=R), а границы отрезка, соответствующего текущей вершине [TL;TR][TL;TR]. В зависимости от взаимного положения этих границ, существуют три варианта действий алгоритма:

* Текущий отрезок полностью входит в запрос (L≤TL;TR≤RL≤TL;TR≤R).
* Если это запрос суммы, вернём предпросчитанную сумму на отрезке. Если это запрос модификации, изменим элемент и пересчитаем сумму. В дочерние вершины спускаться нет надобности.
* Текущий отрезок полностью не входит в запрос (TR<LTR<L или R<TLR<TL).
* Никаких действий выполнять не нужно, просто выйдем из функции. Если это запрос суммы, просто вернём 00.
* Текущий отрезок частично входит в запрос.
* Вызовем функцию рекурсивно для двух дочерних отрезков. Если это запрос суммы, вернём сумму двух полученных значений. Если это запрос модификации, пересчитаем значение суммы для текущего отрезка (так как оно могло измениться).

Обозначим эти варианты соответственно зелёным, красным и жёлтым цветом. Тогда запрос суммы на отрезке [1;11][1;11] массива длиной 1313 будет обработан следующим образом:



А запрос модификации 44-го элемента:



## Реализация дерева отрезков для поиска суммы

Полное бинарное дерево представляем, как и, например, кучу, с помощью массива и формул перехода l=2vl=2v и r=2v+1r=2v+1. Для предотвращения всех возможных переполнений размер дерева отрезков для массива из nn элементов принимают равным 4n4n.

build(int v, int l, int r) **if** l == r **then** d[v] = a[l]; **else** int mid = (l + r) / 2;  
 build(v \* 2, l, mid);  
 build(v \* 2 + 1, mid + 1, r);  
 d[v] = d[v \* 2] + d[v \* 2 + 1];

По БД

* Как работает оптимизатор. Индексы vs перпесортировка.
* Цена содержания индексов
* Перенос этого в обычный кодинг