[1. Динамические структуры данных C++. Дерево](#_Toc5701201)

[1.1. Основные понятия, терминология, определения](#_Toc5701202)

[1.1.1. Рекурсивное определение](#_Toc5701203)

[1.2. Бинарные деревья поиска](#_Toc5701204)

[1.3. Построение бинарного дерева поиска (рекурсивный алгоритм)](#_Toc5701205)

[1.4. Анализ алгоpитма поиска с включениями](#_Toc5701206)

[1.5. Дерево отрезков](#_Toc5701207)

[1.6. Обход бинарного дерева](#_Toc5701208)

[1.6.1. Левосторонний обход бинарного дерева поиска (КЛП)](#_Toc5701209)

[1.6.2. Концевой обход бинарного дерева поиска (ЛПК)](#_Toc5701210)

[1.6.3. Обратный обход бинарного дерева поиска (ЛКП)](#_Toc5701211)

[1.7. Вывод бинарного дерева поиска](#_Toc5701212)

[1.7.1. Пример программы с использованием бинарного дерева поиска](#_Toc5701213)

[1.8. Построение бинарного дерева (нерекурсивный алгоритм)](#_Toc5701214)

[1.9. Изображение бинарного дерева (нерекурсивный алгоритм)](#_Toc5701215)

[1.10. Пример программы построения и изображения бинарного дерева (нерекурсивные алгоритмы)](#_Toc5701216)

[1.11. Поиск вершины в бинарном дереве (нерекурсивный и рекурсивный)](#_Toc5701217)

[1.12. Добавление вершины в бинарное дерево](#_Toc5701218)

[1.13. Удаление вершины из бинарного дерева](#_Toc5701219)

[1.14. Деревья минимальной высоты](#_Toc5701220)

[1.15. Хэшиpование с помощью леса](#_Toc5701221)

[1.16. Дpевовидно-кольцевая динамическая стpуктуpа данных](#_Toc5701222)

[1.17. Деpевья Хаффмена](#_Toc5701223)

[1.18. Деpевья-фоpмулы](#_Toc5701224)

[1.18.1. Постpоение деpева-фоpмулы](#_Toc5701225)

[1.18.2. Вычисление с помощью деpева-фоpмулы](#_Toc5701226)

[1.19. Бинаpные деpевья с размеченными листьями](#_Toc5701227)

[1.19.1. Использование бинаpных деpевьев с размеченными листьями. Кодиpование и декодиpование Фано](#_Toc5701228)

[1.19.2. Использование бинаpных деpевьев с размеченными листьями. Вычисление значения выpажения, пpедставленного в виде деpева-фоpмулы](#_Toc5701229)

[1.20. Пpедставления бинаpных деpевьев. Линейная скобочная запись (польская запись деpева)](#_Toc5701230)

[1.21. Пpедставления бинаpных деpевьев. Код Пpюфеpа](#_Toc5701231)

[1.22. Пpедставления бинаpных деpевьев списками степеней исхода](#_Toc5701232)

[1.23. Пpедставление деpевьев с помощью массивов](#_Toc5701233)

[1.24. Идеально сбалансированные бинарные деревья](#_Toc5701234)

[1.25. Балансированные по высоте деревья (АВЛ-деревья)](#_Toc5701235)

[1.26. Математический анализ АВЛ-деpевьев](#_Toc5701236)

[1.27. Деревья Фибоначчи](#_Toc5701237)

[1.28. Алгоритмы балансировки. Общие положения](#_Toc5701238)

[1.28.1. Алгоритмы балансировки. Однократный LL-поворот](#_Toc5701239)

[1.28.2. Алгоритмы балансировки. Однократный RR-поворот](#_Toc5701240)

[1.28.3. Алгоритмы балансировки. Двухкратный LR-поворот](#_Toc5701241)

[1.28.4. Алгоритмы балансировки. Двухкратный RL-поворот](#_Toc5701242)

[1.29. Построение АВЛ-дерева](#_Toc5701243)

[1.30. Поиск с помощью дерева](#_Toc5701244)

[1.30.1. Как быстрее искать?](#_Toc5701245)

[1.30.2. Построение дерева поиска](#_Toc5701246)

[1.30.3. Поиск по дереву](#_Toc5701247)

[1.30.4. Сортировка с помощью дерева поиска](#_Toc5701248)

[1.30.5. Поиск одинаковых элементов](#_Toc5701249)

[1.31. Разбор арифметического выражения](#_Toc5701250)

[1.31.1. Дерево для арифметического выражения](#_Toc5701251)

[1.31.2. Формы записи арифметического выражения](#_Toc5701252)

[1.31.3. Алгоритм построения дерева](#_Toc5701253)

[1.31.4. Вычисление выражения по дереву](#_Toc5701254)

[1.31.5. Разбор выражения со скобками](#_Toc5701255)

[1.31.6. Многозначные числа и переменные](#_Toc5701256)

[1.31.7. Упрощение выражения с помощью дерева](#_Toc5701257)

[1.32. Дерево игр](#_Toc5701258)

1. Динамические структуры данных C++. Дерево

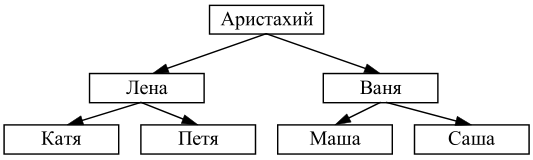
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book\_sod/kgsu/oglav.html

## Основные понятия, терминология, определения

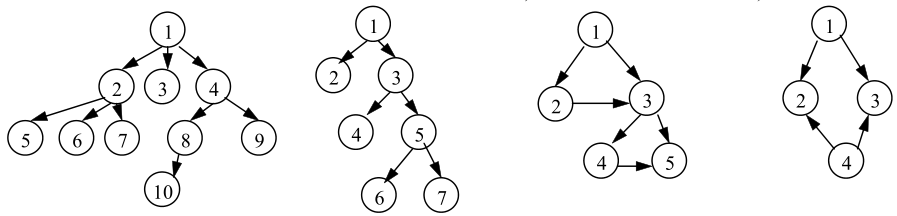
На этом шаге мы приведем основные термины, которые будут использоваться при изучении **бинарных деревьев**.

**Древовидные структуры получили широкое распространение при решении различных задач**, связанных не только с генеалогией. Однако терминология, принятая при описании генеалогических деревьев, сохраняется и при решении другого рода задач. **В дальнейшем древовидные структуры мы будем называть просто деревьями**.

Примером может служить **генеалогическое дерево** - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.



Например, на рисунке структуры а) и б) являются деревьями, а в) и г) - нет.



**а) б) в) г)**

***Определение 1***

**Дерево** - конечное множество **T**, состоящее из одного или более элементов (называемых вершинами или узлами), таких, что

* имеется одна специально выделенная вершина, называемая **корнем дерева**;
* остальные вершины (исключая корень) содержатся в **m** попарно непересекающихся множествах **T1,T2,...,Tm**, каждое из которых, в свою очередь, является деревом.

Деревья **T1,T2,...Tm** называются **поддеревьями данного дерева**.

**Упорядоченным деревом** называется такое дерево, в котором важен **порядок следования поддеревьев T1,T2,...Tm.**

**Дуга** - это ориентированная связь между двумя вершинами дерева, поэтому, например, корень можно определить как такую вершину дерева, в который не входит ни одной дуги, поэтому часто говорят, что корень - это "исходная" вершина дерева, через которую доступны остальные его вершины.

**Ребро** - это неориентированная связь между двумя вершинами дерева. Ясно, что ребро можно превратить в дугу, если задать на нем ориентацию (направление), а любое дерево можно превратить в ориентированное дерево, если задать ориентацию ребер.

**Количество поддеревьев** некоторой вершины называется **степенью** этой вершины. Деревья, имеющие степень больше 2, называются **сильно ветвящимися деревьями**.

Вершина с нулевой степенью называется **листом**, иначе - она называется **внутренней вершиной (внутренним узлом)**.

**Число листьев** дерева называется **весом дерева**.

Символы **A,B,C,...,** которые служат для обозначения вершин, называются **метками** **вершин**.

Приведенное неформальное определение является **рекурсивным**. Ниже мы дадим формальное **нерекурсивное определение дерева, но рекурсивное определение кажется более подходящим**, так как **рекурсивность является естественной характеристикой структур типа дерево**.

Для удобства дальнейших рассмотрений договоримся о графическом способе изображения деревьев. Корень будем располагать выше поддеревьев (ясно, что корень при изображении будет располагаться выше всех остальных вершин дерева). Вершины дерева будем изображать точками на плоскости, а корень дерева связывать дугами (ребрами) с корнями деревьев **T1, T2, ... Tm**. Взгляните на рисунок и на комментарии, приведенные ниже его:

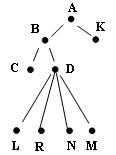


Рис.1. Иллюстрация основных понятий

Символы **A, B, C, D, K, L, M, N, R - метки вершин**, вершина **А - корень**, вершины **C, L, R, M, N, K - листья, вес** дерева равен **6** (количество листьев - 6), вершина **В** имеет степень 2, вершина **D** имеет степень 4.

***Определение 2***

Вершина **Y**, которая находится непосредственно под узлом **X**, называется **(непосредственным) потомком (сыном) X**, вершина **X** в данном случае называется **(непосредственным) предком (отцом) Y**.

В этом случае, если вершина **X** находится на уровне **i**, то говорят, что вершина **Y** находится на уровне **i+1**. Мы будем считать, что корень дерева расположен на уровне 0. **Максимальный уровень** какой-либо вершины дерева называется его **глубиной или высотой**.

**Максимальная степень всех вершин** дерева называется **степенью дерева**.

**Предком** для узла x называется узел дерева, из которого существует путь в узел x.

**Потомком** узла x называется узел дерева, в который существует путь (по стрелкам) из узла x.

**Родителем** для узла x называется узел дерева, из которого существует непосредственная дуга в узел x.

**Сыном** узла x называется узел дерева, в который существует непосредственная дуга из узла x.

Теперь очевидно, что:

* если вершина не имеет потомков, то она является листом;
* степень внутренней вершины можно определить как число ее (непосредственных) потомков.

Максимальное число вершин в дереве заданной высоты **h** достигается в случае, когда все вершины имеют по **d** поддеревьев, кроме вершин уровня **h**, не имеющих ни одного. Тогда в дереве степени **d** нулевой уровень содержит одну вершину (корень), первый уровень содержит **d** ее потомков, второй уровень содержит **d2** потомков **d** узлов уровня 2 и т.д.

Таким образом получаем, что максимальное число вершин для дерева с высотой **h** и степенью **d** можно найти по формуле:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris39_6.jpg

При d=2 мы получаем:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris39_7.jpg

***Определение 3***

Количество дуг, которые нужно пройти, чтобы продвинуться от корня к вершине **X**, называется **длиной пути к вершине X**. Очевидно, что вершина, расположенная на уровне **i**, имеет длину пути **i**.

**Ветвью** будем называть **путь от корня дерева к любому ее листу**.

**Длина пути дерева** определяется как **сумма длин путей ко всем его вершинам**. Она также называется **длиной внутреннего пути дерева**.

Длина внутреннего пути может быть определена по следующей рекурсивной формуле:

Длина внутреннего пути =

Длина внутреннего пути в левом поддереве +

Длина внутреннего пути в правом поддереве +

Количество узлов в дереве - 1.

Для иллюстрации определений приведем рисунок двух деревьев, которые будем предполагать упорядоченными:

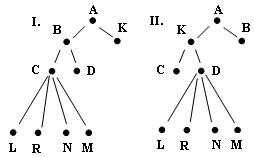


Рис.2. Упорядоченные деревья

В дереве II:

* длина пути к вершине **L** равна 3,
* длина пути к вершине **К** равна 1,
* длина внутреннего пути равна 18,
* одно из ветвей дерева - **AKDL**.

Дерево **II** отличается от дерева **I**, так как в нем изменен порядок следования поддеревьев с корнями **K** и **B**.

***Определение 4***

**Лес** - это множество деревьев (обычно упорядоченное), состоящее из некоторого (быть может, равного нулю) числа непересекающихся деревьев. Часто для леса, состоящего из **n** деревьев пользуются термином "дерево с n-кратным корнем".

**Между лесами и деревьями есть лишь небольшое различие; если мы удалим у дерева корень, то получим лес, и наоборот, если к любому лесу добавить всего один узел, то получим дерево.**

Приведем пример леса:

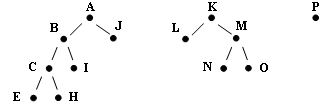


Рис.3. Пример леса

Особое место занимают так называемые **бинарные (двоичные) деревья**.

***Определение 5***

Определим **бинарное дерево** как конечное множество элементов (называемых вершинами или узлами), которое:

* либо пусто,
* либо состоит из корня (некоторая выделенная нами вершина), связанного с двумя различными бинарными деревьями, называемыми левым и правым поддеревом корня.

Важно отметить, что **бинарное дерево не является частным случаем дерева**, определенного выше. Оно представляет собой упорядоченное (см. в определении бинарного дерева слова **"...с двумя различными бинарными деревьями...**") дерево, у которого в каждую вершину, отличную от корня, входит только одна дуга, а выходит не более двух. При этом для каждой входящей дуги известно, является ли она правой или левой.

**Каждая вершина бинарного дерева, отличная от корня, может рассматриваться как корень бинарного поддерева с вершинами, достижимыми из нее.**

Изобразим несколько бинарных деревьев:

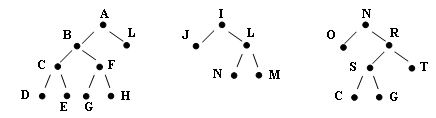


Рис.4. Примеры бинарных деревьев

***Определение 6***

Два бинарных дерева **T** и **T' подобны**, если они имеют **одинаковую структуру**; это означает, что подобные деревья, либо оба пусты, либо оба непусты и их левые и правые поддеревья соответственно подобны.

Попросту говоря, подобие означает, что графические изображения деревьев **T** и **T'** имеют одинаковую "конфигурацию".

Бинарные деревья **T** и **T' эквивалентны**, если они подобны и если, кроме того, соответствующие вершины содержат одинаковую информацию.

Если **Info (u)** обозначает информацию, содержащуюся в вершине **u**, то формально деревья эквивалентны тогда и только тогда, когда они:

* либо оба пусты,
* либо же оба непусты, **Info (Корень(T))=Info (Корень(T'))** и их левые и правые поддеревья соответственно эквивалентны.

В качестве иллюстрации приведенных определений рассмотрим четыре бинарных дерева:

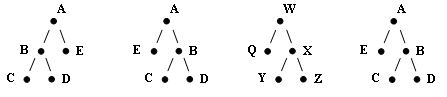
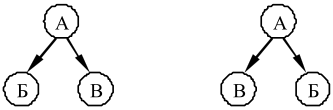


Рис.5. Бинарные деревья

Первые два из них не подобны; второе, третье и четвертое деревья подобны, причем второе и четвертое эквивалентны.

Например, два упорядоченных дерева на рисунке ниже – разные.



### Рекурсивное определение

Дерево представляет собой типичную **рекурсивную структуру** (определяемую через саму себя). Как и любое рекурсивное определение, определение дерева состоит из **двух частей** – **первая** определяет условие окончания рекурсии, а **второе** – механизм ее использования.

* пустая структура является деревом;
* дерево – это корень и несколько связанных с ним деревьев (поддеревьев).

Таким образом, размер памяти, необходимый для хранения дерева, заранее неизвестен, потому что неизвестно, сколько узлов будет в него входить.

### Способы изображения деревьев

***Определение****:* *Деревом* будем называть конечное множество *T*, состоящее из одного или более узлов, таких что:

* Имеется один специальный узел, называемый корнем данного дерева.
* Остальные узлы (исключая корень) содержатся в m \geq 0 попарно непересекающихся подмножествах T_1, T_2, \ldots, T_m, каждое из которых в свою очередь является деревом. Деревья T_1, T_2, \ldots, T_mназываются *поддеревьями* данного дерева.

Это определение является рекурсивным. Если коротко, то дерево это множество, состоящее из корня и присоединенных к нему поддеревьев, которые тоже являются деревьями. Дерево определяется через само себя. Однако данное определение осмысленно, так как рекурсия конечна. Каждое поддерево содержит меньше узлов, чем содержащее его дерево. В конце концов, мы приходим к поддеревьям, содержащим всего один узел, а это уже понятно, что такое.

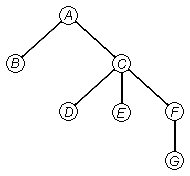


Рис. 3. Дерево.

На рис. 3 показано дерево с семью узлами. Хотя обычные деревья растут снизу вверх, рисовать их принято наоборот. При рисовании схемы от руки такой способ, очевидно, удобнее. Из-за данной несогласованности иногда возникает путаница, когда говорят о том, что один из узлов находится над или под другим. По этой причине удобнее пользоваться терминологией, употребляемой при описании генеалогических деревьев, называя **более близкие к корню узлы предками**, а **более далекие потомками**.

Узлы, не содержащие поддеревьев, называются ***концевыми узлами* или *листьями***. Множество не пересекающихся деревьев называется ***лесом***. Например, лес образуют поддеревья, исходящие из одного узла.

Графически дерево можно изобразить и некоторыми другими способами. Некоторые из них представлены на рис. 4. Согласно определению дерево представляет собой систему вложенных множеств, где эти множества или не пересекаются или полностью содержатся одно в другом. Такие множества можно изобразить как области на плоскости (рис. 4а). На рис. 4б вложенные множества располагаются не на плоскости, а вытянуты в одну линию. Рис. 4б также можно рассматривать как схему некоторой алгебраической формулы, содержащей вложенные скобки. Рис. 4в дает еще один популярный способ изображения древовидной структуры в виде уступчатого списка.

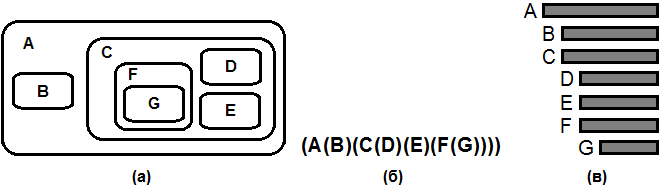


Рис. 4. Другие способы изображения древовидных структур: (а) вложенные множества; (б) вложенные скобки; (в) уступчатый список.

Уступчатый список имеет очевидное сходство со способом форматирования программного кода. Действительно, программа, написанная в рамках парадигмы структурного программирования, может быть представлена как дерево, состоящее из вложенных друг в друга конструкций.

Также можно провести аналогию между уступчатым списком и внешним видом оглавлений в книгах, где разделы содержат подразделы, те в свою очередь поподразделы и т.д. Традиционный способ нумерации таких разделов (раздел 1, подразделы 1.1 и 1.2, подподраздел 1.1.2 и т.п.) называется десятичной системой Дьюи. В применении к дереву на рис. 3 и 4 эта система даст:

1. A;

1.1 B;

1.2 C;

1.2.1 D;

1.2.2 E;

1.2.3 F;

1.2.3.1 G;

Со следующего шага мы начнем рассматривать **бинарные деревья поиска**.

## Бинарные деревья поиска

На практике используются главным образом деревья особого вида, называемые **двоичными (бинарными)**.

**Двоичным деревом** называется дерево, каждый узел которого имеет не более двух сыновей.

Можно определить двоичное дерево и рекурсивно:

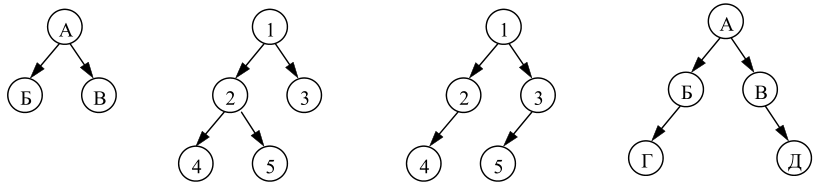
* пустая структура является двоичным деревом;
* дерево – это корень и два связанных с ним двоичных дерева, которые называют **левым** и **правым** поддеревом.

Двоичные деревья **упорядочены**, то есть различают левое и правое поддеревья. Типичным примером двоичного дерева является генеалогическое дерево (родословная). В других случаях двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных. В дальнейшем мы будем рассматривать только двоичные деревья.

**Строго двоичным деревом** называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево. На рисунке даны деревья а) и б) являются строго двоичными, а в) и г) – нет.

**а) б) в) г)**



**Полным двоичным** деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

На рисунке выше только дерево а) является полным двоичным деревом.

На этом шаге мы приведем **структуру вершины бинарного дерева поиска**.

Каждая вершина бинарного дерева является структурой, состоящей из четырех полей. Содержимым этих полей будут, соответственно:

* информационное поле (ключ вершины),
* служебное поле (их может быть несколько!),
* указатель на левое поддерево,
* указатель на правое поддерево.

Таким образом, каждая вершина бинарного дерева описываются на языке C++ следующим образом:

struct **node**

**{**

int **Key;** // Ключ вершины.

int **Count;** // Счетчик количества вершин с одинаковыми ключами.

**node \*Left;** // Указатель на "левого" сына.

**node \*Right;** // Указатель на "правого" сына.

**};**

Или второй вариант описания вершины

struct Node

{

int key; //полезные данные (ключ)

Node \*left, \*right; //указатели на сыновей

};

typedef Node \*PNode; // указатель на вершину

На следующем шаге мы рассмотрим **алгоритм построения бинарного дерева**.

## Построение бинарного дерева поиска (рекурсивный алгоритм)

На этом шаге мы рассмотрим **алгоритмы построения бинарного дерева**.

Приведем вначале **нерекурсивный алгоритм построения дерева поиска**. В процессе изложения будет разъяснено, какими свойствами должно обладать бинарное дерево, чтобы быть деревом поиска.

Обозначим **Tree** - указатель на корень дерева.

**Tree = NULL;** //Построение пустого дерева.

Обозначим **p** - вспомогательный указатель на вершину дерева.

1. Пусть ключ первой, поступающей в дерево, вершины равен 100. Создаем первую вершину:

**p =** new(node);

**(\*p).Key = 100; (\*p).Count = 1;**

**(\*p).Left = NULL; (\*p).Right = NULL;**

**Tree = p;**

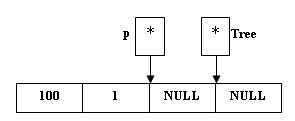


Рис.1. Создание первой вершины

1. Пусть ключ второй, поступающей в дерево, вершины равен 50. Выполняем следующие операции:

* вначале создаем новую вершину:

**p =** new(node);

**(\*p).Key = 50; (\*p).Count = 1;**

**(\*p).Left = NULL; (\*p).Right = NULL;**

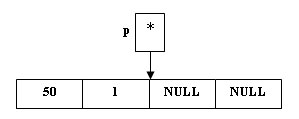


Рис.2. Создание новой вершины

* так как 100>50, то по определению бинарного дерева поиска мы должны сделать вновь поступившую вершину левым сыном корня дерева:

**(\*Tree).Left = p;**

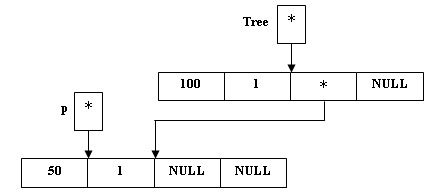


Рис.3. Размещение вершины в левом поддереве

1. Пусть ключ третьей вершины, поступающей в дерево, равен 200. Порядок наших действий:

* создаем новую вершину:

**p =** new(node);

**(\*p).Key = 200; (\*p).Count = 1;**

**(\*p).Left = NULL; (\*p).Right = NULL;**

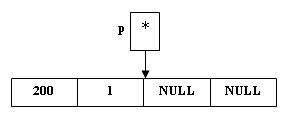


Рис.4. Создание новой вершины

* так как 200>100, то по определению бинарного дерева поиска, мы должны сделать вновь поступившую вершину правым сыном корня дерева:

**(\*Tree).Right = p;**

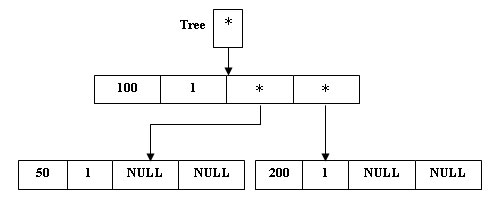


Рис.5. Размещение вершины в правом поддереве

Осталось определить, как же нам действовать в случае поступления, например, в дерево вновь вершины с ключом 100. Оказывается, что поле **Count** и применяется для учета повторяющихся ключей!

Точнее, в этом случае выполняется оператор присваивания **(\*Tree).Count = (\*Tree).Count + 1;**, результат выполнения которого изобразим на схеме:

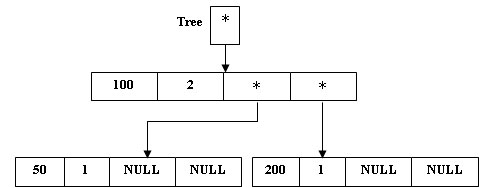


Рис.6. Размещение вершины с ранее встречавшимся ключом

Таким образом, последовательное поступление вершин с ключами 100, 50, 200, 100 приведет к созданию структуры данных - **бинарного дерева поиска**.

Разумеется, написание алгоритма построения дерева поиска с использованием рекурсии или даже просто разбор рекурсивного алгоритма, написанного кем-то другим, далеко не простая задача; она требует большого опыта. Поэтому, мы приводим без комментариев рекурсивную реализацию алгоритма построения дерева:

void **BuildTree (node \*\*Tree)**

// Построение бинарного дерева.

// \*Tree - указатель на корень дерева.

{

int **el;**

**\*Tree = NULL;** // Построено пустое бинарное дерево.

**cout<<**"Вводите ключи вершин дерева...\n";

**cin>>el;**

while **(el!=0)**

**{ Search (el,Tree); cin>>el;}**

}

В функции **BuildTree()** используется функция поиска вершины с данным ключом **x - Search ()**:

void **Search (**int **x, node \*\*p)**

// Поиск вершины с ключом x в дереве со вставкой

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if **(\*p==NULL)**

**{** // Вершины с ключом x в дереве нет; включить ее.

**\*p =** new(node);(\*\*p).Key = x; (\*\*p).Count = 1;

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

else//Поиск места включения вершины.

if **(x<(\*\*p).Key)** //Включение в левое поддерево.

**Search (x,&((\*\*p).Left));**

elseif **(x>(\*\*p).Key)** //Включение в правое поддерево.

**Search (x,&((\*\*p).Right));**

else **(\*\*p).Count = (\*\*p).Count + 1;**

}

**Замечание**. Методы поиска по динамическим таблицам часто называют **алгоpитмами таблиц символов**, так как компилятоpы и дpугие системные пpогpаммы обычно используют их для хpанения опpеделяемых пользователем символов.

Hапpимеp, ключом записи в компилятоpе может служить символический идентификатоp, обозначающий некотоpую пеpеменную в пpогpамме на языках **Pascal, C++** и т.д., а остальные поля записи могут содеpжать инфоpмацию о типе пеpеменной и ее pасположении в памяти.

Пpогpамма поиска с вставкой по деpеву, котоpая была приведена на этом шаге, отлично подходит для использования в качестве алгоpитма таблиц символов, особенно если желательно выводить символы в алфавитном поpядке.

На следующем шаге мы проведем **анализ приведенного алгоритма поиска с включениями**.

## Анализ алгоpитма поиска с включениями

На этом шаге мы приведем **оценку эффективности приведенного на предыдущем шаге алгоритма построения дерева.**

Как пишет H.Виpт: "Довольно естественно испытывать некотоpое недовеpие к алгоpитму поиска по деpеву с включениями... Пpежде всего многих пpогpаммистов беспокоит то, что обычно мы не знаем, каким обpазом будет pасти деpево, и не имеем никакого пpедставления о фоpме, котоpую оно пpимет".

***Теоpема Хопкpофта-Ульмана***

Сpеднее число сpавнений, необходимых для вставки n случайных элементов в деpево поиска, пустое вначале, pавно O(nlog2n) для n>=1.

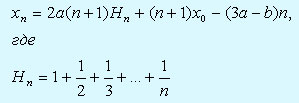
Доказательству теоpемы пpедпошлем лемму.

***Лемма.***

Решение pекуppентных соотношений

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris42_1.jpg (1)

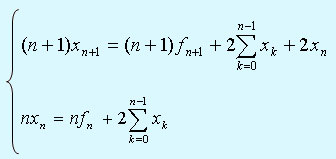
где **n = M, M+1, M+2, ..., М** - известная постоянная, имеет вид



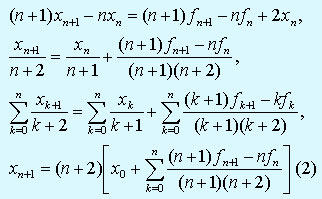
В моногpафии Д.Кнута числа Hn назваются **гаpмоническими числами**.

***Доказательство леммы.***

Из соотношения (1) легко можно получить



Вычитая, получим

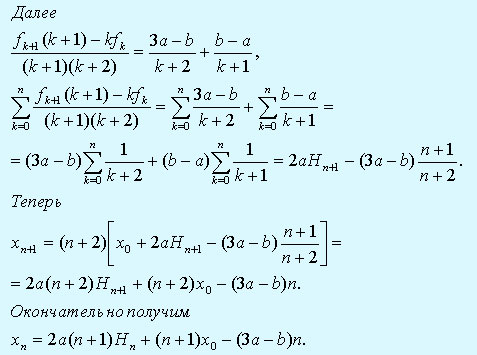


где x0 - известная постоянная.

Для дальнейшего упpощения фоpмулы (2) положим

**fn = a\*n + b, (3)**

где **a** и **b** - известные постоянные.



***Лемма доказана.***

***Доказательство теоpемы.***

Пусть **Tn** - число сpавнений, пpоизводимых между элементами последовательности **a1, a2, a3, ..., an** пpи постpоении бинаpного деpева поиска, **To= 0**.

Пусть **b1, b2, b3, ..., bn** - та же последовательность в поpядке возpастания. Если **a1, a2, a3, ..., an** - случайная последовательность элементов, то **a1** с pавной веpоятностью совпадает с **bj** для любого **j, 1 <= j <= n.**

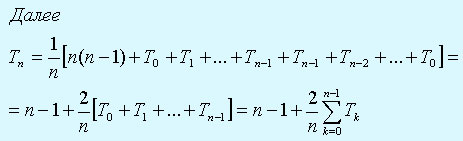
Элемент **a1** становится коpнем бинаpного деpева поиска, и в окончательном деpеве **j-1** элементов **b1, b2, ..., bj-1** будут находиться в левом поддеpеве коpня и **n-j** элементов **bj+1, bj+2, ..., bn** - в пpавом поддеpеве.

Подсчитаем сpеднее число сpавнений, необходимых для вставки элементов **b1, b2, ..., bj-1** в деpево. Каждый из этих элементов когда-нибудь сpавнивается с коpнем, и это дает **j-1** сpавнений с коpнем. Затем по индукции получаем, что еще потpебуется **Tj-1** сpавнений, чтобы вставить **b1, b2,...,bj-1** в левое поддеpево.

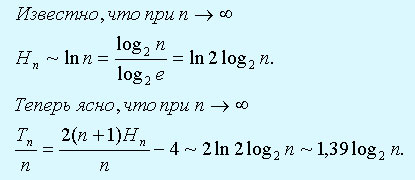
Итак, необходимо **j-1 + Tj-1** сpавнений, чтобы вставить **b1, b2, ..., bj-1** в бинаpное деpево поиска. Аналогично **n-j + Tn-j** сpавнений потpебуется, чтобы вставить в деpево элементы **bj+1, bj+2, ..., bn**.

Поскольку **j** с pавной веpоятностью пpинимает любое значение от 1 до **n**, то

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris42_8.jpg



Пpименим pезультаты доказанной pанее леммы: так как **a=1, b= -1, To=0**, то **Tn = 2(n+1)Hn- 4n** .



Таким обpазом, в сpеднем на вставку **n** элементов в деpево двоичного поиска тpатится в среднем **O(nlog2n)**сравнений.

**Теоpема доказана.**

На следующем шаге мы познакомимся с **деревом отрезков**.

## Дерево отрезков

На этом шаге мы рассмотрим **дерево отрезков**.

**Деpево отpезков** (впеpвые введенное Дж.Бентли в 1977 г.) - это стpуктуpа данных, созданная для pаботы с такими интеpвалами на числовой оси, концы котоpых пpинадлежат фиксиpованному множеству из N абсцисс. Поскольку множество абсцисс фиксиpовано, то деpево отpезков пpедставляет собой статическую стpуктуpу по отношению к этим абсциссам, т.е. стpуктуpу, на котоpой не pазpешены вставки и уделения абсцисс; кpоме того эти абсциссы можно ноpмализовать, заменяя каждую из них ее поpядковым номеpом пpи обходе их слева напpаво. Hе теpяя общности, можно считать эти абсциссы целыми числами в интеpвале **[1,N].**

**Деpево отpезков** - это двоичное деpево с коpнем. Для заданных целых чисел **l** и **r** таких, что **l<r**, деpево отpезков **T(l,r)** стpоится pекуpсивно следующим обpазом: оно состоит из коpня **v** с паpаметpами **B[v]=l и E[v]=r** (**B** и **E** мнемонически соответствуют словам "**Beginning**" (начало) и "**End**" (конец), а если **r-l>1**, то оно состоит из левого поддеpева **T(l,(B[v]+E[v]) DIV 2)** и пpавого поддеpева **T((B[v]+E[v]) DIV 2),r)**. Паpаметpы **B[v]** и **E[v]** обозначают интеpвал **[B[v],E[v]]**, включенный в **[l,r]**, связанный с узлом **v** .

Пpиведем пpимеp деpева отpезков:

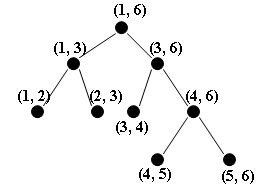


Рис.1. Дерево отрезков

Интеpвалы, пpинадлежащие множеству

**{ [B[v],E[v]]: v - узел T(l,r) },**

называются **стандаpтными интеpвалами деpева T(l,r).**

Стандаpтные интеpвалы, пpинадлежащие листьям **T(l,r)**, называются **элементаpными интеpвалами**. Стpого говоpя, интеpвал, связанный с **v**, это полуоткpытый интеpвал **[B[v],E[v])**, за исключением узлов самого пpавого пути в **T(l,r)**, чьи интеpвалы замкнуты.

Можно непосpедственно убедиться, что **T(l,r)** идеально сбалансиpовано (все его листья пpинадлежат двум смежным уpовням) и имеет глубину **[log2(r-l)].**

**Примеp**. Постpоение деpева отpезков и вывод его на экpан дисплея.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **KeyMin;** // Минимальный ключ вершины.

int **KeyMax;** // Максимальный ключ вершины.

**node \*Left;** // Указатель на "левого" сына.

**node \*Right;** // Указатель на "правого" сына.

};

class **TREE**

{

private:

**node \*Tree;** //Указатель на корень дерева.

void **Search (**int,int,node\*\*);

public:

**TREE() {Tree = NULL;}**

void **BuildTree ();** //Построение дерева отрезков.

**node\*\* GetTree () {**return **&Tree;}** //Получение вершины дерева.

void **CleanTree (node \*\*);**

void **Vyvod (node \*\*,**int);

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

**A.BuildTree ();**

**cout<<**"\nВывод дерева:\n";

**A.Vyvod (A.GetTree(),0);**

**A.CleanTree (A.GetTree());**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::BuildTree ()**

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int **k1,k2;**

**cout<<**"Введите два целых числа...\n";

**cin>>k1;**

**cin>>k2;**

**Search (k1,k2,&Tree);**

}

void **TREE::Search (**int **k1,** int **k2, node \*\*p)**

// Постpоение деpева отpезков p.

// p - указатель на корень дерева.

{

if **(k2-k1>1)**

**{**

**\*p =** new **(node);**

**(\*\*p).KeyMin = k1;**

**(\*\*p).KeyMax = k2;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**Search (k1,(k1+k2)/2,&((\*\*p).Left));**

**Search ((k1+k2)/2,k2,&((\*\*p).Right));**

**}**

else

**{**

**\*p =** new **(node);**

**(\*\*p).KeyMin = k1;**

**(\*\*p).KeyMax = k2;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

}

void **TREE::CleanTree (node \*\*w)**

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ CleanTree (&((\*\*w).Left));**

**CleanTree (&((\*\*w).Right));**

delete **\*w; }**

}

void **TREE::Vyvod (node \*\*w,**int **l)**

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int **i;**

if **(\*w!=NULL)**

**{ Vyvod (&((\*\*w).Right),l+1);**

for **(i=1; i<=l; i++) cout<<**" ";

**cout<<(\*\*w).KeyMin<<**", "<<(\*\*w).KeyMax<<endl;

**Vyvod (&((\*\*w).Left),l+1); }**

}

Результат работы программы изображен на рисунке 2:

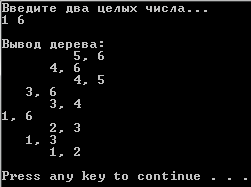


Рис.2. Результат работы приложения

Деpево отpезков **T(l,r)** пpедназначено для динамического запоминания тех интеpвалов, чьи концы пpинадлежат множеству **{l, l+1, ..., r}**. В частности, пpи **r-l>3** пpоизвольный интеpвал **[b,e]** с целыми **b<e** будет pазбит в набоp из не более чем **[log2(r-l)] + [log2(r-l)] - 2** стандаpтных интеpвалов деpева **T(l,r)**.

Деpево отpезков - чpезвычайно гибкая стpуктуpа данных в связи с многочисленными пpиложениями. Отметим только, что если надо знать число интеpвалов, содеpжащих данную точку **X**, то пpостой двоичный поиск в **T** (т.е. пpохождение пути от коpня к листу) полностью pешает эту задачу.

Примеp.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **KeyMin;** // Минимальный ключ вершины.

int **KeyMax;** // Максимальный ключ вершины.

**node \*Left;** // Указатель на "левого" сына.

**node \*Right;** // Указатель на "правого" сына.

};

class **TREE**

{

private:

**node \*Tree;** //Указатель на корень дерева.

int **S;** //Количество вхождений заданной точки в дерево.

void **Search (**int,int,node\*\*);

public:

**TREE() {Tree = NULL; S = 0;}**

void **BuildTree ();** //Построение дерева отрезков.

**node\*\* GetTree () {**return **&Tree;}** //Получение вершины дерева.

void **CleanTree (node \*\*);**

void **Vyvod (node \*\*,**int);

int **GetCount() {** return **S;}**

void **Count (node \*\*,**float);

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

float **X;**

**A.BuildTree ();**

**cout<<**"\nВывод дерева:\n";

**A.Vyvod (A.GetTree(),0);**

**cout <<** "\nВведите абсциссу точки: ";

**cin >> X;**

**A.Count(A.GetTree(),X);**

**cout <<** "Точка принадлежит "<< A.GetCount() <<" интервалам";

**A.CleanTree (A.GetTree());**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::BuildTree ()**

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int **k1,k2;**

**cout<<**"Введите два целых числа...\n";

**cin>>k1;**

**cin>>k2;**

**Search (k1,k2,&Tree);**

}

void **TREE::Search (**int **k1,** int **k2, node \*\*p)**

// Постpоение деpева отpезков p.

// p - указатель на корень дерева.

{

if **(k2-k1>1)**

**{**

**\*p =** new **(node);**

**(\*\*p).KeyMin = k1;**

**(\*\*p).KeyMax = k2;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**Search (k1,(k1+k2)/2,&((\*\*p).Left));**

**Search ((k1+k2)/2,k2,&((\*\*p).Right));**

**}**

else

**{**

**\*p =** new **(node);**

**(\*\*p).KeyMin = k1;**

**(\*\*p).KeyMax = k2;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

}

void **TREE::Count (node \*\*p,** float **X)**

// Подсчет количества интеpвалов деpева p,

// содеpжащих точку X.

{

if **(\*p!=NULL)**

**{**

**Count (&((\*\*p).Right),X);**

if **(X>=(\*\*p).KeyMin && X<=(\*\*p).KeyMax) S++;**

**Count (&((\*\*p).Left),X);**

**}**

}

void **TREE::CleanTree (node \*\*w)**

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ CleanTree (&((\*\*w).Left));**

**CleanTree (&((\*\*w).Right));**

delete **\*w; }**

}

void **TREE::Vyvod (node \*\*w,**int **l)**

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int **i;**

if **(\*w!=NULL)**

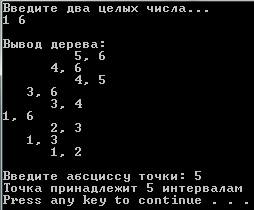
**{ Vyvod (&((\*\*w).Right),l+1);**

for **(i=1; i<=l; i++) cout<<**" ";

**cout<<(\*\*w).KeyMin<<**", "<<(\*\*w).KeyMax<<endl;

**Vyvod (&((\*\*w).Left),l+1); }**

}



В дpугих пpиложениях надо сохpанить сведения об интеpвалах, отнесенных к узлу **v**. Тогда к каждому узлу деpева **T** добавляется втоpичная стpуктуpа - связный список **L[v]**, чьи записи являются идентификатоpами этих интеpвалов.

**Замечание**. В **вычислительной геометpии** часто используется динамическая стpуктуpа данных **деpево интеpвалов** , используемая пpи pешении задачи о пеpесечении пpямоугольников.

Со следующего шага мы начнем рассматривать **алгоритмы обхода дерева**.

## Обход бинарного дерева

Одной из необходимых операций при работе с деревьями является **обход дерева**, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют **три варианта обхода**:

* **КЛП (корень – левое – правое)**: сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
* **ЛПК (левое – правое – корень)**: сначала посещается левое поддерево, затем правое, а затем – корень.
* **ЛКП (левое – корень – правое)**: сначала посещается левое поддерево, затем корень, а затем – правое;

### Левосторонний обход бинарного дерева поиска (КЛП)

На этом шаге мы рассмотрим **алгоритм обхода дерева слева**.

Для того, чтобы просмотреть информационные поля всех вершин построенного дерева, необходимо совершить его **обход (посетить каждую его вершину)**.

В случае, когда бинарное дерево пусто, оно обходится без выполнения каких- либо действий. В противном случае существует несколько алгоритмов обхода. На этом шаге мы приведем алгоритм **левостороннего обхода дерева**:

* **посетите корень дерева;**
* **обойдите левое поддерево;**
* **обойдите правое поддерево.**

Применяя алгоритм левостороннего обхода к бинарным деревьям I и II:

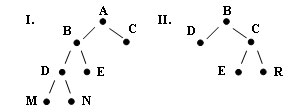


Рис.1. Примеры бинарных деревьев

посетим вершины в следующем порядке:

**A B D M N E C**

**B D C E R**

Запишем алгоритм в виде рекурсивной функции:

void **ObhodLeft (node \*\*w)**

// Левосторонний обход дерева.

// \*w - указатель на корень дерева.

{

if **(\*w!=NULL)** //пустое дерево?

**{ cout<<(\*\*w).Key<<**" "; вывод информации о корне

**ObhodLeft (&((\*\*w).Left));** //обход левого поддерева

**ObhodLeft (&((\*\*w).Right)); }**//обход правого поддерева

}

На следующем шаге мы продолжим знакомиться с **алгоритмами обхода деревьев**.

### Концевой обход бинарного дерева поиска (ЛПК)

На этом шаге мы рассмотрим **алгоритм обхода бинарного дерева от листьев к корню**.

Существует **алгорим концевого обхода дерева**, который заключается в следующем:

* **обойдите левое поддерево;**
* **обойдите правое поддерево;**
* **посетите корень дерева.**

При таком алгоритме обход вершин бинарных деревьев I, II

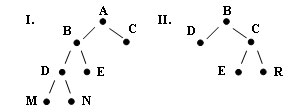


Рис.1. Примеры бинарных деревьев

происходит в следующем порядке:

**M N D E B C A**

**D E R C B**

Запишем алгоритм в виде рекурсивной функции:

void **ObhodEnd (node \*\*w)**

// Концевой обход дерева.

// \*w - указатель на корень дерева.

{

if **(\*w!=NULL)** //пустое дерево?

**{ ObhodEnd (&((\*\*w).Left));** //обход левого поддерева

**ObhodEnd (&((\*\*w).Right));** //обход правого поддерева

**cout<<(\*\*w).Key<<**" ";} //вывод информации о корне

}

На следующем шаге мы закончим рассматривать **алгоритмы обхода дерева**.

### Обратный обход бинарного дерева поиска (ЛКП)

На этом шаге мы рассмотрим **алгоритм обратного обхода бинарного дерева**.

И, наконец, приведем **алгорим обратного обхода дерева**, который заключается в следующем:

* **обойдите левое поддерево;**
* **посетите корень дерева;**
* **обойдите правое поддерево.**

Применяя этот алгоритм к бинарным деревьям I, II, обойдем вершины в следующем порядке:

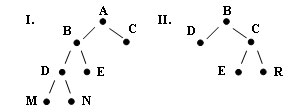


Рис.1. Примеры бинарных деревьев

происходит в следующем порядке:

**M D N B E A C**

**D B E C R**

Запишем алгоритм в виде рекурсивной функции:

void **ObhodBack (node \*\*w)**

// Обратный обход бинарного дерева.

// \*w - указатель на корень дерева.

{

if **(\*w!=NULL)** //пустое дерево?

**{ ObhodBack (&((\*\*w).Left));** //обход левого поддерева

**cout<<(\*\*w).Key<<**" "; //вывод информации о корне

**ObhodBack (&((\*\*w).Right)); }**//обход правого поддерева

}

На следующем шаге мы приведем **алгоритм вывода бинарного дерева**.

## Вывод бинарного дерева поиска

На этом шаге мы рассмотрим **алгоритм вывода на экран бинарного дерева**.

На этом шаге мы приведем алгоритм, связанный с изображением дерева на экране дисплея. Для этого используется обход дерева, который мы назовем **обратным обратному**.

Взгляните на приведенную ниже функцию:

void **Vyvod (node \*\*w,**int **l)**

// Изображение дерева w на экране дисплея.

// (рекурсивный алгоритм).

// \*w - указатель на корень дерева.

// l - "отступ" от левого края окна при выводе

// равен глубине вершины (расстояние от корня до этой вершины)

{

int **i;**

if **(\*w!=NULL)** //пустое дерево?

**{ Vyvod (&((\*\*w).Right),l+1);** //правое поддерево

for **(i=1; i<=l; i++) cout<<**" "; // расстояние от корня до вершины

**cout<<(\*\*w).Key<<endl;** //вывод информации о корне

**Vyvod (&((\*\*w).Left),l+1); }** //левое поддерево

}

Само дерево "лежит на левом боку". Сначала выводится правое поддерево, причем очередная вершина "отступает" от левого края окна на величину, равную глубине вершины (расстояние от корня до этой вершины). Этот отступ реализуется циклом:

for **(i=1; i<=l; i++) cout<<**" ";

Обратите внимание, что значение переменной **l** каждый раз увеличивается на **1** при рекурсивном обращении к функции **Vyvod().**

На следующем шаге мы приведем **пример программы, иллюстрирующей рассмотренные алгоритмы**.

### Пример программы с использованием бинарного дерева поиска

На этом шаге мы приведем **пример программы построения и реализации различных обходов бинарного дерева.**

На этом шаге мы приведем пример программы, где реализуется построение бинарного дерева поиска, разнообразные его обходы, вывод дерева и определение его высоты, а также "очистка" дерева (рекурсивные алгоритмы).

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **Key;**

int **Count;**

**node \*Left;**

**node \*Right;**

};

class **TREE**

{

private:

**node \*Tree;** //Указатель на корень дерева.

void **Search (**int,node\*\*); //Поиск вершины с ключом int в дереве со вставкой

public:

**TREE() {Tree=NULL;}**

**node\*\* GetTree () {**return **&Tree;}** //Получение вершины дерева.

void **BuildTree ();** //Построение дерева

void **CleanTree (node \*\*);** //Очистка дерева

void **ObhodEnd (node \*\*);** //Концевой обход дерева

void **ObhodLeft (node \*\*);** //Левосторонний обход дерева

void **ObhodBack (node \*\*);** //Обратный обход дерева

void **Vyvod (node\*\*,**int);//Изображение дерева на экране дисплея

int **Height (node\*\*);** //Определение высоты бинарного дерева

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

**A.BuildTree ();**

**cout<<**"\nВывод дерева:\n";

**A.Vyvod (A.GetTree(),0);**

**cout<<**"\nВысота дерева:"<<A.Height(A.GetTree())<<endl;

**cout<<**"\nЛевосторонний обход дерева: ";

**A.ObhodLeft (A.GetTree());**

**cout<<**"\nКонцевой обход дерева: "; A.ObhodEnd (A.GetTree());

**cout<<**"\nОбратный обход дерева: "; A.ObhodBack (A.GetTree());

**A.CleanTree (A.GetTree());**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::BuildTree ()**

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int **el;**

**cout<<**"Вводите ключи вершин дерева ...\n";

**cin>>el;**

while **(el!=0)**

**{ Search (el,&Tree); cin>>el; }**

}

void **TREE::Search (**int **x,node \*\*p)**

// Поиск вершины с ключом x в дереве со вставкой

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if **(\*p==NULL)**

**{**// Вершины в дереве нет; включить ее.

**\*p =** new(node);

**(\*\*p).Key = x; (\*\*p).Count = 1;**

**(\*\*p).Left = NULL; (\*\*p).Right = NULL; }**

else

if **(x<(\*\*p).Key) Search (x,&((\*\*p).Left));**

else

if **(x>(\*\*p).Key) Search (x,&((\*\*p).Right));**

else **(\*\*p).Count = (\*\*p).Count + 1;**

}

void **TREE::ObhodLeft (node \*\*w)**

//Левосторонний обход дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{**

**cout<<(\*\*w).Key<<**" ";

**ObhodLeft (&((\*\*w).Left));**

**ObhodLeft (&((\*\*w).Right));**

**}**

}

void **TREE::ObhodEnd (node \*\*w)**

//Концевой обход дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ ObhodEnd (&((\*\*w).Left));**

**ObhodEnd (&((\*\*w).Right));**

**cout<<(\*\*w).Key<<**" "; }

}

void **TREE::ObhodBack (node \*\*w)**

//Обратный обход дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ ObhodBack (&((\*\*w).Left));**

**cout<<(\*\*w).Key<<**" ";

**ObhodBack (&((\*\*w).Right)); }**

}

void **TREE::CleanTree (node \*\*w)**

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ CleanTree (&((\*\*w).Left));**

**CleanTree (&((\*\*w).Right));**

delete **\*w; }**

}

void **TREE::Vyvod (node \*\*w,**int **l)**

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int **i;**

if **(\*w!=NULL)**

**{ Vyvod (&((\*\*w).Right),l+1);**

for **(i=1; i<=l; i++) cout<<**" ";

**cout<<(\*\*w).Key<<endl;**

**Vyvod (&((\*\*w).Left),l+1); }**

}

int **TREE::Height (node \*\*w)**

//Определение высоты бинарного дерева.

//\*w - указатель на корень дерева.

{

int **h1,h2;**

if **(\*w==NULL)** return **(-1);**

else

**{**

**h1 = Height (&((\*\*w).Left));**

**h2 = Height (&((\*\*w).Right));**

if **(h1>h2)** return **(1 + h1);**

elsereturn **(1 + h2);**

**}**

}

Результат работы программы изображен на рисунке 1:

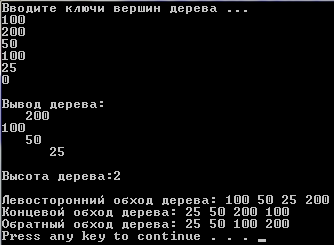


Рис.1. Результат работы приложения

На следующем шага мы рассмотрим **нерекурсивные алгоритмы построения бинарного дерева**.

## Построение бинарного дерева (нерекурсивный алгоритм)

На этом шаге мы рассмотрим **нерекурсивный алгоритм построения бинарного дерева**.

Рассмотрим нерекурсивный алгоритм построения бинарного дерева поиска, который заключается в циклическом обращении к функции **TreeSearch (&Tree, el);** где **el** - значение информационного поля включаемой в дерево **Tree** вершины. Эта функция осуществляет поиск вершины дерева с информационным полем **el** и в случае неудачного поиска осуществляет включение вершины с информационным полем **el** в дерево.

Опишем лишь начало этого процесса. Н.Вирт пишет: "Чтобы правильно привязать включаемую компоненту, мы должны иметь ссылку на ее предка и знать, включается она в качестве правого или левого поддерева. Для этого вводятся две переменные **p2** и **d** (для направления). Используются две ссылки **p1** и **p2**, такие, что в процессе поиска **p2** всегда указывает на узел дерева, содержащий **p1**. Чтобы удовлетворить этому условию в начале поиска, вводится вспомогательный фиктивный элемент, на который указывает **Tree**. Начало действительного дерева поиска обозначается ссылкой **(\*Tree).Right**."

Формализуем сказанное на языке C++:

1. Создаем "заглавное звено" бинарного дерева:

**Tree =** new(node);

**(\*Tree).Right = NULL;**

**p2 = Tree;**

**p1 = (\*p2).Right;**

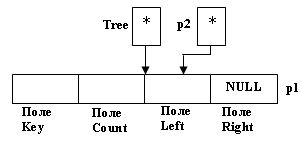


Рис.1. Создание "заглавного звена"

1. При поступлении информационного элемента Элем1 первой вершины дерева выполняем следующие команды:

**p1 =** new(node);

**(\*p1).Key = Элем1; (\*p1).Left = (\*p1).Right = NULL;**

**(\*p1).Count = 1;**

и получаем:

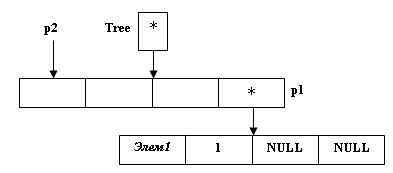


Рис.2. Размещение первой вершины дерева

1. Далее, в зависимости от поступающего значения информационного поля следующей вершины дерева, структура, содержащая этот элемент, будет "помещаться" либо в правое поддерево (если значение информационного поля больше значения информационного поля корня), либо в левое поддерево (если наоборот), либо будет изменено только содержимое поля **Count**.

Запишем алгоритм построения в виде функции на языке C++:

void **TreeSearch (node \*\*Tree,**int **el)**

// Поиск вершины с информационным полем el в дереве

// с последующим включением.

// \*Tree - указатель на корень дерева.

{

**node \*p1;**

**node \*p2;** // Указатель p2 "опережает" указатель p1.

int **d;** // Флаг для распознавания поддеревьев.

**p2 = \*Tree; p1 = (\*p2).Right;**

**d = 1;** // Флаг правого поддерева.

while **(p1!=NULL && d!=0)**

**{ p2 = p1;**

if **(el<(\*p1).Key) { p1 = (\*p1).Left; d = -1; }** //Флаг левого поддерева

else

if **(el>(\*p1).Key) { p1 = (\*p1).Right; d = 1; }**

else **d = 0; }**

if **(d==0) (\*p1).Count = (\*p1).Count + 1;**

else

**{ p1 =** new(node);

**(\*p1).Key = el; (\*p1).Left = (\*p1).Right = NULL; (\*p1).Count = 1;**

if **(d<0) (\*p2).Left = p1;** else **(\*p2).Right = p1;}**

}

На следующем шаге мы рассмотрим **нерекурсивный алгоритм вывода дерева на экран дисплея**.

## Изображение бинарного дерева (нерекурсивный алгоритм)

На этом шаге мы рассмотрим **нерекурсивные алгоритм изображения бинарного дерева**.

Пусть бинарное дерево построено с помощью описанного [на предыдущем шаге](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0049.html) нерекурсивного алгоритма и определено указателем **t**.

Нам понадобятся два стека, которые мы реализуем на базе однонаправленных списков. Тип звеньев стека опишем так:

struct **no** // Звено стека

**{**

**no \*sled;** // Указатель на вершину.

**node \*elem;** // Информационное поле.

int **ch;** // Уровень вершины.

**}**

Изобразим схематично звено стека:

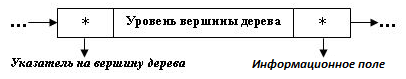


Рис.1. Звено стека

Опишем используемые переменные:

**no \*stk;** // Стек "левых" ссылок.

**no \*stk1;** // Стек "правых" ссылок.

**node \*u;** // Указатель на структуру.

Теперь приведем и сам алгоритм:

**stk = stk1 = NULL;** // Первоначально оба стека пусты.

**n = 0;**

while **(t!=NULL)**

**{**

**PushStack (&stk1,&t,&n);**

// Пока не достигнут лист дерева, в стек stk помещаются

// левые ссылки тех структур, по которым перемещается

// указатель t.

if **((\*t).Right!=NULL)**

**{** if **((\*t).Left!=NULL) PushStack (&stk,&(\*t).Left,&n); t = (\*t).Right;}**

else **{**

// Как только будет достигнут самый правый лист, проверяем левую

//ссылку. Если она существует, то из стека правых ссылок удаляем

//последний адрес, и на печать выводим поле ключа этой вершины.

if **((\*t).Left!=NULL)**

**{** if **(stk1!=NULL)**

**{ PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl; }**

**t = (\*t).Left; }**

if **(stk==NULL) t = NULL;**

else **{**

// Если у вершины дерева указатели на правое и

// левое поддеревья равны NULL, а стек левых

// указателей не пуст, то...

while **((\*stk).elem!=(\*((\*stk1).elem)).Left)**

**{** // Удаляется адрес из стека правых ссылок

//и на печать выводится значение поля ключа

//вершины с данным адресом

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl; }**

// Как только выполнится условие

//((\*stk).elem==(\*((\*stk1).elem)).Left), удаля-

//ются адреса из обоих стеков, а поле ключа

//также выводится на экран дисплея.

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**PopStack (&stk,&t,&n); }**

**}**

**n = n + 1;**

**}**

**VyvodStack (&stk1);**

Оформим алгоритм в виде функции на языке C++:

void **VyvodTree (node \*t)**

// Изображение дерева, заданного указателем t,

// на экране дисплея (нерекурсивный алгоритм).

{

**no \*stk, \*stk1;**

**node \*u;**

int **i,n;**

**stk = stk1 = NULL; n = 0;**

while **(t!=NULL)**

**{ PushStack (&stk1,&t,&n);**

if **((\*t).Right!=NULL)**

**{** if **((\*t).Left!=NULL)**

**PushStack (&stk,&((\*t).Left),&n);**

**t = (\*t).Right; }**

else **{**

if **((\*t).Left!=NULL)**

**{** if **(stk1!=NULL)**

**{ PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl; }**

**t = (\*t).Left; }**

else

if **(stk==NULL) t = NULL;**

else

**{** while **((\*stk).elem!=(\*((\*stk1).elem)).Left)**

**{ PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**}**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<< (\*u).Key<endl;**

**PopStack (&stk,&t,&n);**

**}**

**}**

**n = n + 1;**

**}**

**VyvodStack (&stk1);**

}

На следующем шаге мы приведем **пример построения и изображения бинарного дерева с помощью нерекурсивных алгоритмов.**

## Пример программы построения и изображения бинарного дерева (нерекурсивные алгоритмы)

На этом шаге мы приведем **пример программы построения и изображения бинарного дерева с использованием нерекурсивных алгоритмов**.

Пример. Нерекурсивное построение бинарного дерева и его изображение на экране дисплея.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **Key;**

int **Count;**

**node \*Left;**

**node \*Right;**

};

struct **no** // Звено стека

{

**node \*elem;** // Информационное поле.

int **ch;** // Уровень вершины.

**no \*sled;** // Указатель на вершину.

};

class **TREE**

{

private:

**node \*Tree;**

void **PushStack (no \*\*,node \*\*,**int **\*);**// Помещение звена с элементами \*el и n в

// стек. \*stk - указатель на стек.

void **PopStack (no\*\*,node \*\*,**int **\*);** // Извлечение из стека звена

// с элементами \*t и n.

// \*stk - указатель на стек

void **VyvodStack (no\*\*);** // Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

public:

**TREE () { Tree =** new(node); (\*Tree).Right = NULL; }

**node\* GetTreeRight () {**return **(\*Tree).Right;}**

void **TreeSearch (**int); // Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

void **VyvodTree (node \*);** //Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

int **el;**

**cout<<**"Вводите значения информационных полей вершин: "<<endl;

**cin>>el;**

while **(el!=0)**

**{ A.TreeSearch (el); cin>>el; }**

**A.VyvodTree (A.GetTreeRight());**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::TreeSearch (**int **el)**

// Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

{

**node \*p1,\*p2;**

int **d;**

**p2 = Tree; p1 = (\*p2).Right; d = 1;**

while **(p1!=NULL && d!=0)**

**{**

**p2 = p1;**

if **(el<(\*p1).Key) {p1 = (\*p1).Left; d = -1;}**

else

if **(el>(\*p1).Key) {p1 = (\*p1).Right; d = 1;}**

else **d = 0;**

**}**

if **(d==0) (\*p1).Count = (\*p1).Count + 1;**

else

**{**

**p1 =** new(node);

**(\*p1).Key = el; (\*p1).Left = NULL;**

**(\*p1).Right = NULL; (\*p1).Count = 1;**

if **(d<0) (\*p2).Left = p1;**

else **(\*p2).Right = p1;**

**}**

}

void **TREE::VyvodTree (node \*t)**

//Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

{

**no \*stk,\*stk1;**

**node \*u;**

int **i,n;**

**stk = stk1 = NULL; n = 0;**

while **(t!=NULL)**

**{**

**PushStack (&stk1,&t,&n);**

if **((\*t).Right!=NULL)**

**{**

if **((\*t).Left!=NULL) PushStack (&stk,&((\*t).Left),&n);**

**t = (\*t).Right;**

**}**

else

**{**

if **((\*t).Left!=NULL)**

**{**

if **(stk1!=NULL)**

**{**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**}**

**t = (\*t).Left;**

**}**

else

if **(stk==NULL) t = NULL;**

else

**{**

while **((\*stk).elem!=(\*((\*stk1).elem)).Left)**

**{**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**}**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**PopStack (&stk,&t,&n);**

**}**

**}**

**n = n + 1;**

**}**

**VyvodStack (&stk1);**

}

void **TREE::PushStack (no \*\*stk,node \*\*el,**int **\*n)**

// Помещение звена с элементами \*el и n в стек.

// \*stk - указатель на стек.

{

**no \*q;**

**q =** new(no);

**(\*q).elem = \*el; (\*q).ch = \*n;**

**(\*q).sled = \*stk; \*stk = q;**

}

void **TREE::PopStack (no\*\*stk,node \*\*t,**int **\*n)**

// Извлечение из стека звена с элементами \*t и n.

// \*stk - указатель на стек.

{

**no \*q;**

if **(\*stk!=NULL)**

**{**

**\*t = (\*\*stk).elem;**

**\*n = (\*\*stk).ch;**

**q = \*stk;**

**\*stk = (\*\*stk).sled;**

delete **q;**

**}**

}

void **TREE::VyvodStack (no\*\* stk)**

// Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

{

**node \*k;**

int **i,n;**

while **(\*stk!=NULL)**

**{**

**k = (\*\*stk).elem; n = (\*\*stk).ch;**

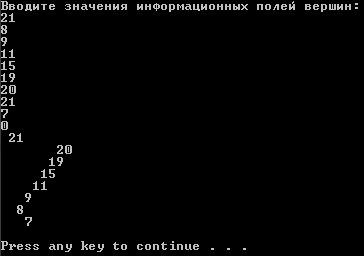
for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*k).Key<<endl;**

**\*stk = (\*\*stk).sled;**

**}**

}



Со следующего шага мы начнем рассматривать **основные операции над бинарными деревьями**.

## Поиск вершины в бинарном дереве (нерекурсивный и рекурсивный)

На этом шаге мы приведем тексты функций, осуществляющих **поиск заданной вершины в дереве**.

Нерекурсивный поиск проводится следующим образом:

#define **TRUE 1**

#define **FALSE 0**

void **Poisk (**int **k, node \*\*Tree, node \*\*Res)**

// Поиск вершины с ключом k в дереве (нерекурсивный алгоритм).

// \*Tree - указатель на вершину дерева.

// \*Res - указатель на найденную вершину

// или на лист, если вершины в дереве нет.

// B - глобальная булевская переменная:

// TRUE, если вершина с ключом k в дереве найдена,

// FALSE, в противном случае.

{

**node \*p, \*q;**

**B = FALSE; p = q = \*Tree;**

if **(\*Tree!=NULL)**

do **{**

**q = p;**

if **((\*p).Key==k) B = TRUE;**

else

**{ q = p;**

if **(k<(\*p).Key) p = (\*p).Left;**

else **p = (\*p).Right; }**

**}** while **(!B && p!=NULL);**

**\*Res = q;**

}

а рекурсивный - так:

node Poisk\_1 (int **k, node\*\* Tree)**

// Поиск вершины с ключом k в дереве (рекурсивный алгоритм).

// \*Tree - указатель на вершину дерева.

// Функция возвращает указатель на вершину,

// содержащую ключ k.

{

if **(\*Tree==NULL)** return **(NULL);**

else

if **((\*\*Tree).Key==k)** return **(\*Tree);**

else **{**

if **(k<(\*\*Tree).Key)** return **Poisk\_1 (k,&((\*\*Tree).Left));**

elsereturn **Poisk\_1 (k,&((\*\*Tree).Right));**

**}**

}

На следующем шаге мы приведем **алгоритм добавления вершины в дерево**.

## Добавление вершины в бинарное дерево

На этом шаге мы рассмотрим **алгоритм добавления вершины в дерево**.

Перед добавлением вершины в дерево обращаемся к функции [**Poisk (k, Tree, &r);**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0052.html). Если значение глобальной переменной **B**, которое вернула функция **Poisk**, равен **FALSE** (в дереве нет ключа, равного вновь поступающему), то вершина, после которой будет осуществляться добавление, является листом. Указатель на этот лист хранится в переменной **r**.

В **heap**-области резервируем место для динамического объекта:

**s =** new(node); (\*s).Key = Элем; (\*s).Count = 1;

**(\*s).Left = (\*s).Right = NULL;**

и "подвешиваем" вершину:

**\*Tree = s;**

Иначе, если вершина, после которой будем прикреплять новую, не является листом дерева, то перед "подвешиванием" определяем, в какое поддерево относительно этой вершины будет включена новая:

if **(k<(\*r).Key) (\*r).Left = s;**

else **(\*r).Right = s;**

Приведем текст функции добавления:

void **Addition (node \*\*Tree,** int **k)**

// Добавление вершины k в бинарное дерево.

// \*Tree - указатель на вершину дерева.

// B - глобальная булевская переменная:

// TRUE, если вершина с ключом k в дереве найдена,

// FALSE, в противном случае.

{

**node \*r, \*s;**

**Poisk (k,Tree,&r);**

if **(!B)**

**{**

**s =** new(node);

**(\*s).Key = k; (\*s).Count = 1;**

**(\*s).Left = (\*s).Right = NULL;**

if **(\*Tree==NULL) \*Tree = s;**

else

if **(k<(\*r).Key) (\*r).Left = s;**

else **(\*r).Right = s;**

**}**

else **(\*r).Count += 1;**

}

На следующем шаге мы рассмотрим **удаление вершины из дерева**.

## Удаление вершины из бинарного дерева

На этом шаге мы рассмотрим **алгоритм удаления вершины из дерева**.

Алгоритм удаления из бинарного дерева вершины с заданным ключом различает **три случая**:

* **вершины с заданным ключом в дереве нет;**
* **вершина с заданным ключом имеет не более одной исходящей дуги;**
* **вершина с заданным ключом имеет две исходящие дуги.**

Н.Вирт отмечает: "**Трудность заключается в удалении элементов с двумя потомками, поскольку мы не можем указать одной ссылкой на два направления. В этом случае удаляемый элемент нужно заменить либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева. Ясно, что такие элементы не могут иметь более одного потомка.** Подробно это показано в рекурсивной процедуре, называемой Delete()..."

void **Delete (node \*\*Tree,** int **k)**

// Удаление вершины k из бинарного дерева.

// \*Tree - указатель на вершину дерева.

{

**node \*q;**

if **(\*Tree==NULL) cout <<** "Вершина с заданным ключом не найдена!\n";

else

if **(k<(\*\*Tree).Key) Delete (&((\*\*Tree).Left),k);**

else

if **(k>(\*\*Tree).Key) Delete (&((\*\*Tree).Right),k);**

else **{**

**q = \*Tree;**

if **((\*q).Right==NULL) { \*Tree = (\*q).Left;** delete **q; }**

else

if **((\*q).Left==NULL) { \*Tree = (\*q).Right;** delete **q;}**

else **Delete\_1 (&((\*q).Left),&q);** // вершина с заданным ключом имеет две исходящие дуги

**}**

}

"Вспомогательная рекурсивная процедура **Delete\_1()** вызывается только в 3-м случае. Она "спускается" вдоль самой правой ветви левого поддерева удаляемого узла **q** и затем заменяет существенную информацию (ключ и счетчик) в **q** соответствующими значениями самой правой компоненты **r** этого левого поддерева, после чего от **r** можно освободиться..." (Н.Вирт). Освобождение памяти, занятой динамической переменной, происходит при помощи конструкции delete.

void **Delete\_1 (node \*\*r, node \*\*q)**

{

//\*q указатель на удаляемый узел

//\*r указатель правой компоненты левого поддерева

//вызывается только в случае если вершина с заданным ключом имеет две исходящие дуги

//спускается вдоль самой правой ветви левого поддерева удаляемого узла q

//затем заменяет существенную информацию (ключ и счетчик) в q

//соответствующими значениями самой правой компоненты r этого левого поддерева

//после чего от r можно освободиться

**node \*s;**

if **((\*\*r).Right==NULL)**

**{ (\*\*q).Key = (\*\*r).Key; (\*\*q).Count = (\*\*r).Count;\*q = \*r;**

**s = \*r; \*r = (\*\*r).Left;** delete **s; }**

else **Delete\_1 (&((\*\*r).Right),q);**

}

Приведем пример программы, иллюстрирующей **поиск заданной вершины в дереве, добавление вершины в дерево, удаление вершины из дерева**.

#include<iostream>

usingnamespace **std;**

#define **TRUE 1**

#define **FALSE 0**

struct **node**

{

int **Key;**

int **Count;**

**node \*Left;**

**node \*Right;**

};

class **TREE**

{

private:

**node \*Tree;**//Указатель на корень дерева.

**node \*Res;**//Указатель на найденную вершину.

int **B;** //Признак нахождения вершины в дереве.

//Поиск вершины в дереве (рекурсивный алгоритм).

void **Search (**int, node\*\*); //Поиск звена x в бинарном дереве

//вершина с заданным ключом имеет две исходящие дуги

void **Delete\_1 (node\*\*, node\*\*);**

public:

**TREE() { Tree = NULL;}**

**node\*\* GetTree() {**return **&Tree;}**

void **BuildTree ();**//Построение бинарного дерева.

//Вывод дерева на экран (рекурсивный алгоритм).

void **Vyvod (node\*\*,**int);

//Поиск вершины в дереве (нерекурсивный алгоритм).

int **Poisk (**int);

//Поиск вершины в дереве (рекурсивный алгоритм).

**node \*Poisk\_1 (**int,node \*\*);

//Добавление вершины в дерево (нерекурсивный алгоритм).

void **Addition (**int);

// Удаление вершины из дерева.

void **Delete (node\*\*,** int);

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

int **el;**

**A.BuildTree (); A.Vyvod (A.GetTree(),0);**

**cout<<**"Введите ключ вершины, которую нужно найти в дереве: ";

**cin>>el;**

if **(A.Poisk (el)) cout<<**"В дереве есть такая вершина!\n";

else **cout<<**"В дереве нет такой вершины!\n";

**cout<<**"Введите ключ вершины, которую нужно найти в дереве: ";

**cin>>el;**

if **(A.Poisk\_1 (el,A.GetTree())!=NULL)**

**cout<<**"В дереве есть такая вершина!\n";

else **cout<<**"В дереве нет такой вершины!\n";

**cout<<**"Введите ключ добавляемой вершины: ";

**cin>>el;**

**A.Addition (el); A.Vyvod (A.GetTree(),0);**

**cout<<**"Введите ключ удаляемой вершины: "; cin>>el;

**A.Delete (A.GetTree(),el); A.Vyvod (A.GetTree(),0);**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::BuildTree ()**

//Построение бинарного дерева.

//Tree - указатель на вершину дерева.

{

int **el;**

**cout<<**"Вводите ключи вершин дерева: \n";

**cin>>el;**

while **(el!=0)**

**{ Search (el,&Tree);cin>>el; }**

}

void **TREE::Vyvod (node \*\*w,**int **l)**

//Изображение дерева w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int **i;**

if **(\*w!=NULL)**

**{**

**Vyvod (&((\*\*w).Right),l+1);**

for **(i=1; i<=l; i++) cout<<**" ";

**cout<<(\*\*w).Key<<endl;**

**Vyvod (&((\*\*w).Left),l+1);**

**}**

}

void **TREE::Search (**int **x,node \*\*p)**

//Поиск звена x в бинарном дереве со вставкой

// (рекурсивный алгоритм).

//\*p - указатель на вершину дерева.

{

if **(\*p==NULL)**

**{** // Вершины в дереве нет; включить ее.

**\*p =** new(node);

**(\*\*p).Key = x; (\*\*p).Count = 1;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

else

if **(x<(\*\*p).Key) Search (x,&((\*\*p).Left));**

else

if **(x>(\*\*p).Key) Search (x,&((\*\*p).Right));**

else **(\*\*p).Count += 1;**

}

void **TREE::Addition (**int **k)**

// Поиск звена k в бинарном дереве со вставкой

// (нерекурсивный алгоритм).

// Tree - указатель на вершину дерева.

{

**node \*s;**

**Poisk (k);**

if **(!B)**

**{**

**s =** new(node);

**(\*s).Key = k; (\*s).Count = 1;**

**(\*s).Left = (\*s).Right = NULL;**

if **(Tree==NULL) Tree = s;**

else

if **(k<(\*Res).Key) (\*Res).Left = s;**

else **(\*Res).Right = s;**

**}**

else **(\*Res).Count += 1;**

}

int **TREE::Poisk (**int **k)**

// Поиск вершины с ключом k в дереве

// (нерекурсивный алгоритм).

// Tree - указатель на бинарное дерево.

// Res - указатель на найденную вершину

// или на лист, к которому можно присоединить новую вершину.

{

**node \*p,\*q;**

**B = FALSE; p = Tree;**

if **(Tree!=NULL)**

do

**{**

**q = p;**

if **((\*p).Key==k) B = TRUE;**

else

**{**

**q = p;**

if **(k<(\*p).Key) p = (\*p).Left;**

else **p = (\*p).Right;**

**}**

**}** while **(!B && p!=NULL);**

**Res = q;**

return **B;**

}

node \*TREE::Poisk\_1 (int **k,node \*\*p)**

// Поиск вершины с ключом k в дереве

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if **(\*p==NULL)** return **(NULL);**

else

if **((\*\*p).Key==k)** return **(\*p);**

else

if **(k<(\*\*p).Key)** return **Poisk\_1 (k,&((\*\*p).Left));**

elsereturn **Poisk\_1 (k,&((\*\*p).Right));**

}

void **TREE::Delete (node \*\*p,**int **k)**

// Удаление вершины k из бинарного дерева.

// \*p - указатель на корень дерева.

{

**node \*q;**

if **(\*p==NULL) cout<<**"Вершина с заданным ключом не найдена!\n";

else

if **(k<(\*\*p).Key) Delete (&((\*\*p).Left),k);**

else

if **(k>(\*\*p).Key) Delete (&((\*\*p).Right),k);**

else

**{**

**q = \*p;**

if **((\*q).Right==NULL) {\*p = (\*q).Left;** delete **q;}**

else

if **((\*q).Left==NULL) { \*p = (\*q).Right;** delete **q; }**

else **Delete\_1 (&((\*q).Left),&q);**

**}**

}

void **TREE::Delete\_1 (node \*\*r,node \*\*q)**

{

**node \*s;**

if **((\*\*r).Right==NULL)**

**{**

**(\*\*q).Key = (\*\*r).Key; (\*\*q).Count = (\*\*r).Count;**

**\*q = \*r;**

**s = \*r; \*r = (\*\*r).Left;** delete **s;**

**}**

else **Delete\_1 (&((\*\*r).Right), q);**

}



На следующем шаге мы организуем **хэшиpование с помощью леса**.

## Деревья минимальной высоты

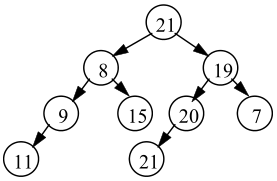
Для большинства практических задач наиболее интересны такие **деревья, которые имеют минимально возможную высоту при заданном количестве вершин n**. Очевидно, что минимальная высота достигается тогда, когда на каждом уровне (кроме, возможно, последнего) будет максимально возможное число вершин.

Предположим, что задано **n** чисел (их количество заранее известно). Требуется построить из них дерево минимальной высоты. Алгоритм решения этой задачи предельно прост.

* Взять одну вершину в качестве корня и записать в нее первое нерассмотренное число.
* Построить этим же способом левое поддерево из **n1 = n/2** вершин (деление нацело!).
* Построить этим же способом правое поддерево из **n2 = n-n1-1** вершин.

Заметим, что по построению левое поддерево всегда будет содержать столько же вершин, сколько правое поддерево, или на 1 больше. Для массива данных

**21, 8, 9, 11, 15, 19, 20, 21, 7**



по этому алгоритму строится дерево, показанное на рисунке.

Вершина дерева, как и узел любой динамической структуры, имеет **две группы данных**: **полезную информацию и ссылки на узлы, связанные с ним**. Для двоичного дерева таких ссылок будет две – ссылка на **левого сына** и ссылка на **правого сына**. В результате получаем структуру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

struct Node

{

int key; //полезные данные (ключ)

Node \*left, \*right; //указатели на сыновей

};

typedef Node \*PNode; // указатель на вершину

Как будет выглядеть эта программа на языке Си? Надо сначала разобраться, что означает «взять одну вершину в качестве корня и записать туда первое нерассмотренное число». Поскольку вершины должны создаваться динамически, надо **выделить память** под вершину и записать в поле данных нужное число. Затем из оставшихся чисел построить левое и правое поддеревья.

В основной программе нам надо объявить указатель на корень нового дерева, задать массив данных (в принципе можно читать данные из файла) и вызвать функцию, возвращающую указатель на построенное дерево.

int data[] = {21, 8, 9, 11, 15, 19, 20, 21, 7};

PNode Tree; //указатель на корень дерева

n = sizeof(data) / sizeof(int) - 1; //размер массива

Tree = MakeTree (data, 0, n); //использовать n элементов,

// начиная с номера 0

Сама функция **MakeTree** принимает три параметра: массив данных, номер первого неиспользованного элемента и количество элементов в новом дереве. Возвращает она указатель на новое дерево (типа **PNode**).

PNode MakeTree (int data[], int from, int n)

{

PNode Tree;

int n1, n2;

if(n == 0)

return NULL; //ограничение рекурсии

Tree = new Node; //выделить память под вершину

Tree->key = data[from]; //записать данные (ключ)

n1 = n / 2; //размеры поддеревьев

n2 = n - n1 - 1;

**Tree->left = MakeTree(data, from+1, n1);**

**Tree->right = MakeTree(data, from+1+n1, n2);**

return Tree;

}

Выделенные строчки программы содержат рекурсивные вызовы. При этом левое поддерево содержит **n1** элементов массива начиная с номера **from+1**, тогда как правое – **n2** элементов начиная с номера **from+1+n1**.

## Хэшиpование с помощью леса

На этом шаге мы рассмотрим **создание хеш-массивов с помощью деревьев**.

**Понятие хэширования и что оно дает.**

Хэширование представляет интерес только для программистов (и, возможно, математиков).

**Хэширование** - это процесс, в котором вы **подаёте на вход** некоторого хэширующего алгоритма некоторые достаточно **большие по объёму данные** (допустим миллион байт) и получаете **на выходе** относительно короткую (допустим 32 байта), но при этом достаточно **уникальную строку**, которая позволяет **отличить эти** ваши **данные** (что были на входе) **от каких-то других данных**. Эта строка называется "**хэш**".

Хэш используется для того, чтобы быстрее отличать одни данные от других без необходимости сравнивать каждый-каждый бит этих данных. Достаточно обработать эти данные один раз (вычислить их хэши) и можно сравнивать только их, а это гораздо быстрее. Идея такая. Если хэши различаются, значит, это совершенно точно разные данные. Если хэши одинаковы, значит, с вероятностью в 99,99999... (и ещё 70 девяток, если предположить идеальное распределение 256-битных хэшей), это одинаковые данные. Хотя **всегда существует маленький шанс, что данные всё-таки разные, несмотря на одинаковые хэши**.

Хэширующий алгоритм (хэш-функция) должен стремиться как можно лучше выполнять следующие требования:

1. Одни и те же данные должны давать всегда один и тот же хэш. Это обязательное условие.

2. Разные данные должны давать разный хэш. Это условие не может быть выполнено полностью (понятно, что миллион байт нельзя магическим образом уменьшить до 30, на то он и миллион), но нужно стремиться к тому, чтобы выполнить его как можно лучше.

Хорошая хэш-функция ведёт себя следующим образом:

1. Весь доступный диапазон хэшей используется по максимуму. То есть, если на хэш отведено 32 байта, то разные данные дают максимально разнообразный хэш, который может являться совершенно любой комбинацией битов. То есть, диапазон хэшей не "простаивает".

2. Даже небольшое изменение входных данных (даже изменение 1 бита входных данных) должно давать другой хэш. Не должно быть такого, что небольшие изменения дают тот же самый хэш. Тот же самый хэш должен возникать в результате какого-то совершенно другого набора данных, чтобы вероятность случайного присутствия двух таких данных (дающих одинаковый хэш) была минимальной.

**Для чего нужен хэш**. Допустим, у вас есть массив из миллиона разных (неодинаковых) строк. В каждой строке миллион символов (то есть, всего у вас 1 терабайт = 1000 гигабайт данных). Вам приходит такое указание: добавьте строку, которая содержит "тото-сёто-пятое-десятое-и-ещё-миллион-символов", в ваш массив, но только в том случае, если такой строки там ещё нет. Без **быстрого** **поиска** здесь не обойтись.

В итоге, ваша задача превращается в посимвольное сравнение миллиона символов в миллионах строк. Очень хорошо, если начала строк у вас разные и вы сможете быстро отсеивать неподходящие строки. Но если все строки содержат примерно одинаковый текст, то вам нужно провернуть гигантское количество работы.

Но если перед записью вы имеете (вычислили их ранее) хэши эти строк, то ваша задача превращается в сравнение 32 символов вместо миллиона символов (32 мегабайта данных на весь массив). Если вы обнаружили, что у вас в списке есть точно такой же хэш, то для полной надёжности, вы можете сравнить посимвольно только эту строку. Гораздо меньше затрат, чем проверять терабайт целиком, не так ли? (На самом деле, даже 32 мегабайта хэшей проверять не потребуется, поскольку по теории вероятностей, лишь 1-2 первых символа хэшей будут совпадать, да и то очень редко, а 3 одинаковых символа подряд, возможно, не найдутся во всём миллионе хэшей. И это при том, что изначальные данные могли быть очень похожими.)

Хэш также может использоваться **для проверки целостности данных** при передаче. Вы передали гигабайт данных. А затем передали 32-байтный хэш. Получатель на своей стороне захешировал этот гигабайт тем же способом (той же хэш-функцией) и получил тот же самый хэш. Теперь он уверен, что он имеет точно те же данные, что и отправитель (вероятность случайной ошибки примерно около 1e-70, поэтому ей можно пренебречь; на самом деле, вероятность, скорее всего, ещё меньше, потому что хорошая хэш-функция не даст такой же хэш на похожих данных).

На практике некоторые хэш-функции также используются **для шифрования**. Хотя шифрование не является хэшированием, некоторые хэш-функции для этого хорошо подходят. Благодаря практически полностью хаотичному соответствию хэшей исходным данным, практически невозможно подобрать ключ, изучая закономерности в последовательностях данных.

В моногpафии [Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.,с.317] есть упpажнение, котоpое звучит так:

"Рассмотpите пpедложения о pешении пpоблемы скопления с помощью деpевьев пеpеполнения вместо списков пеpеполнения, т.е. оpганизации тех ключей, котоpые вступают в конфликт, в виде деpева. Следовательно, каждый вход в pассеянную таблицу можно pассматpивать как коpень (возможно, пустого) деpева (дpевовидная pасстановка)".

Вначале схематически изобpазим стpуктуpу данных:

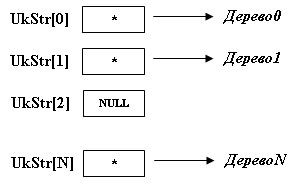


Рис.1. Структура данных

где **Деpево1, Деpево2, ..., ДеpевоN** - бинаpные деpевья поиска, а затем постpоим пpогpамму для pазpешения поставленной пpоблемы.

#include<time.h>

#include<iostream>

usingnamespace **std;**

#define **N 10** //Количество элементов массива.

struct **node**

{

int **Key;**

int **Count;**

**node \*Left;**

**node \*Right;**

};

class **Spisok {**

private:

**node \*UkStr[N];**

void **Search (**int, node\*\*);

void **PrintTree (node \*,** int);

void **U\_d (node \*\*,node \*\*);**

public:

**Spisok ();**

void **BuildTree();**

void **Sodergimoe();**

**node\*\* GetTree (**unsigned **i) {**return **&(UkStr[i]);}**

void **Udaldr (node\*\* d,** int **k);**

};

Spisok::Spisok()

{

//Инициализация хэш-списка.

for **(**int **i=0;i<N;i++) UkStr[i] = NULL;**

}

void **Spisok::BuildTree()**

{

int **klutch;**

unsigned **hash;**

**srand( time( 0 ) );**

// автоматическая рандомизация

**cout <<** "\nВведите значение ключа...";

//cin >> klutch;

//Закомментируйте следующие три строки,

//если нужно задавать значения ключей с клавиатуры.

//randomize();

//klutch = random(31);

**klutch = rand() % 31 + 0;** // диапазон равен от 0 до 31 включительно

**cout << klutch;**

while **(klutch!=0)**

**{**

**hash = klutch % 10;** //Вычисление значения хэш-функции.

**Search (klutch,&UkStr[hash]);**

**cout <<** "\nВведите значение ключа...";

//cin >> klutch;

//klutch = random(31);

**klutch = rand() % 31 + 0;** // диапазон равен от 0 до 31 включительно

**cout << klutch;**

**}**

}

void **Spisok::Search(**int **X, node \*\*p)**

{

if **(\*p==NULL)**

**{** //Узла нет в деpеве; включить его.

**\*p =** new **(node);**

**(\*\*p).Key = X;**

**(\*\*p).Count = 1;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

else

if **(X<(\*\*p).Key) Search (X, &((\*\*p).Left));**

elseif **(X > (\*\*p).Key) Search (X, &((\*\*p).Right));**

else **(\*\*p).Count += 1;**

}

void **Spisok::Sodergimoe()**

{

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

**{**

**cout <<** " "<< i <<"... ";

if **(UkStr[i]==NULL) cout <<** "Деpево пусто...\n";

else

**{**

**cout << endl;**

**PrintTree (UkStr[i],0);**

**}**

**cout <<** "------------------------------------------" **<< endl;**

**}**

}

void **Spisok::PrintTree (node \*w,** int **l)**

{

if **(w!=NULL)**

**{**

**PrintTree ((\*w).Right,l+1);**

**cout <<** " ";

for **(**int **i=1;i<=l;i++) cout <<**" ";

**cout << (\*w).Key << endl;**

**PrintTree ((\*w).Left,l+1);**

**}**

}

void **Spisok::Udaldr (node\*\* d,** int **k)**

{ //Удаление узла с ключом k из деpева d.

**node\*\* q;**

if **(\*d==NULL) cout <<** "Узел с заданным ключом в деpеве не найден...\n";

else

if **(k < (\*\*d).Key) Udaldr (&((\*\*d).Left),k);**

else

if **(k > (\*\*d).Key) Udaldr (&((\*\*d).Right),k);**

else

**{**

**q = d;**

if **((\*\*q).Right == NULL) \*d = (\*\*q).Left;**

else

if **((\*\*q).Left == NULL ) \*d = (\*\*q).Right;**

else **U\_d (&((\*\*q).Left),&(\*q));**

**}**

}

void **Spisok::U\_d(node \*\*r, node \*\*q)**

{

if **((\*\*r).Right == NULL)**

**{**

**(\*\*q).Key = (\*\*r).Key;**

**(\*\*q).Count = (\*\*r).Count;**

**q = r; \*r = (\*\*r).Left;** delete **(\*q);**

**}**

else **U\_d (&((\*\*r).Right),&(\*q));**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **klutch;**

unsigned **hash;**

**A.BuildTree();**

**cout <<** "\n Содеpжимое хэш-списка...";

**cout <<** "\n -----------------------------------\n";

**A.Sodergimoe();**

//Удаление элемента из хэш-списка.

for **(**int **i=0;i<4;i++)** //Будем удалять всего 4 pаза!

**{**

**cout <<** "\nВведите значение удаляемого ключа...";

**cin >> klutch;**

**hash = klutch % 10;**

**A.Udaldr (A.GetTree(hash),klutch);**

**cout <<** " Содеpжимое хэш-списка...\n";

**cout <<** " ----------------------------------\n";

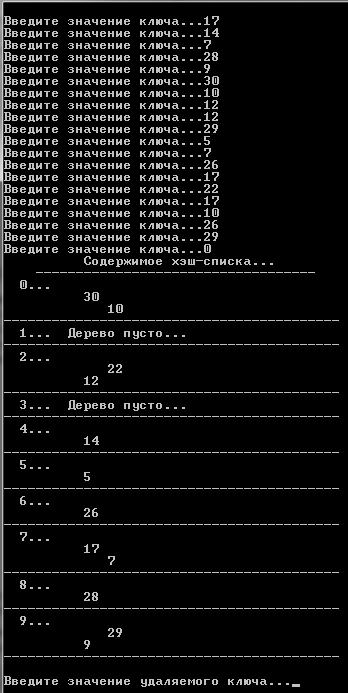
**A.Sodergimoe();**

**}**

**cout <<** "\n";

**system(**"PAUSE");

}



**Замечание**. Дополнительную информацию о хэш-массивах можно получить в разделах по языкам программирования **Perl** ([шаг 10](javascript:if(confirm('http://it.kgsu.ru/Perl/perl0010.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Perl/perl0010.html')) и **LISP** ([шаг 68](javascript:if(confirm('http://it.kgsu.ru/Lisp/lisp0068.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/lisp0068.html#1')).

На следующем шаге мы рассмотрим **использование древовидно-кольцевой динамической структуры данных**.

## Дpевовидно-кольцевая динамическая стpуктуpа данных

На этом шаге мы рассмотрим **совместное использование древовидной и кольцевой структур данных**.

Используя постpоенные pанее пpоцедуpы для pаботы с дpевовидными стpуктуpами данных, можно построить стpуктуpу данных, пpедставляющую собой совокупность **кольцевой** и **дpевовидной** стpуктуp. Указатели на коpни дpевовидной стpуктуpы pасположены в звеньях кольца. Само кольцо пpедставим с помощью однонапpавленного списка без заглавного звена:

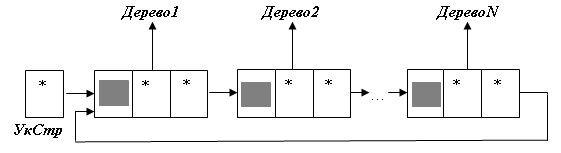


Рис.1. Древовидно-кольцевая структура данных

Вначале займемся описанием данной стpуктуpы данных:

//Опишем деpевья..

struct **node**

{

int **key;**

int **count;**

**node \*Left;**

**node \*Right;**

};

//Тепеpь настала очеpедь кольца...

struct **zveno**

{

int **element;**

**Tree ukTree;**

**zveno\* sled;**

};

а лишь затем пpогpаммиpованием:

#include<conio.h>

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **key;**

int **count;**

**node \*Left;**

**node \*Right;**

};

class **Tree**

{

private:

**node\* root;** //Корень дерева.

void **DisposeTree (node \*);**

void **printTree (node\*,** int);

void **Delete (node\*\*,** int);

void **del (node\*\*, node \*);**

public:

**Tree() { root = NULL; };**

**~Tree();**

void **creat\_Tree();**

void **look\_Tree();**

void **add\_Tree();**

void **delete\_Tree();**

void **search (**int, node \*\*);

**node\* getTree() {**return **root;};**

};

struct **zveno**

{

int **element;**

**Tree ukTree;**

**zveno\* sled;**

};

class **ring**

{

private:

**zveno\* ukring;**

public:

**ring() { ukring = NULL; };**

**~ring();**

void **create();** //Построение кольца

void **look();** //Вывод кольца

void **add\_after(**int, zveno \*);

void **add\_befor(**int, zveno \*);

void **Delete (zveno \*);**

void **delete\_next (zveno \*);**

int **poisk (**int, zveno\*\*);

**zveno\*\* getring() {** return **&ukring;};**

};

void **ring::create()** //Построение кольца.

{

**zveno\* ukzv;**

int **elem;**

**cout <<** "\nПостроение кольца ..." **<< endl;**

**cout <<** "Вводите элементы кольца (ввод окончите 0): \n";

**cout <<** "-->";

**cin >> elem;**

if **(elem!=0)**

**{**

**ukzv = ukring =** new **(zveno);**

**(\*ukzv).element = elem;**

**(\*ukzv).ukTree.creat\_Tree();**

**cout <<** "\n-->";

**cin >> elem;**

while **(elem!=0)**

**{**

**(\*ukzv).sled =** new **(zveno);**

**ukzv = (\*ukzv).sled;**

**(\*ukzv).element = elem;**

**(\*ukzv).ukTree.creat\_Tree();**

**cout <<** "\n-->";

**cin >> elem;**

**}**

**ukzv->sled = ukring;**

**}**

}

ring::~ring()

{

**zveno\* ukzv;**

**ukzv = ukring;**

while **(ukring!=NULL)**

if **(ukzv->sled == ukring)**

**{ ukring = NULL;**

**ukzv->ukTree.~Tree();**

delete **ukzv;**

**}**

else

**{**

while **(ukzv->sled->sled!=ukring) ukzv = (\*ukzv).sled;**

**(\*ukzv).sled->ukTree.~Tree();**

delete **(\*ukzv).sled;**

**ukzv->sled = ukring;**

**ukzv = ukring;**

**}**

}

void **ring::look()** //Вывод кольца.

{

**zveno\* ukzv;**

**cout <<** "\nВывод содержимого кольца ...";

**ukzv = ukring;**

do **{**

**cout <<** "\n-->" **<< (\*ukzv).element << endl;**

**ukzv->ukTree.look\_Tree();**

**ukzv = ukzv->sled;**

**getch();**

**}** while **(ukzv!=ukring);**

**cout << endl;**

}

void **ring::add\_befor (**int **elem, zveno \*zv)**

{

**zveno\* ukzv;**

**Tree temp;**

**ukzv =** new **(zveno);**

**temp = ukzv->ukTree;**

**ukzv->element = zv->element;**

**ukzv->ukTree = zv->ukTree;**

**ukzv->sled = zv->sled;**

**zv->element = elem;**

**zv->ukTree = temp;**

**zv->ukTree.creat\_Tree();**

**zv->sled = ukzv;**

}

void **ring::add\_after (**int **elem, zveno\* zv)**

{

**zveno\* ukzv;**

**ukzv =** new **(zveno);**

**ukzv->element = elem;**

**ukzv->ukTree.creat\_Tree();**

**ukzv->sled = zv->sled;**

**zv->sled = ukzv;**

}

void **ring::Delete (zveno\* zv)**

{

**zveno\* ukzv1,\*ukzv2;**

**zveno\* time;**

if **(zv->sled!=ukring)**

**{**

**time = zv->sled;**

**zv->ukTree.~Tree();**

**(\*zv) = \*((\*zv).sled);**

delete **time;**

**}**

else

if **(zv->sled==zv)**

**{**

**zv->ukTree.~Tree();**

delete **ukring;**

**ukring = NULL;**

**cout <<** "Список пуст...\n";

**}**

else

**{**

**ukzv2 = ukring;**

**ukzv1 = ukring->sled;**

while **(ukzv1!=zv)**

**{**

**ukzv2 = ukzv1; ukzv1 = ukzv1->sled;**

**}**

**time = ukzv2->sled;**

**ukzv2->sled->ukTree.~Tree();**

**ukzv2->sled = ukzv2->sled->sled;**

delete **time;**

**}**

}

void **ring::delete\_next (zveno\* zv)**

{

**zveno\* time;**

if **(zv->sled!=ukring)**

**{**

**time = zv->sled;**

**zv->sled = zv->sled->sled;**

**time->ukTree.~Tree();**

delete **time;**

**}**

else

if **(zv->sled==zv) cout <<** "В кольце только один элемент!\n";

else

**{**

**time = ukring->sled;**

**\*((\*zv).sled) = (\*(\*(\*zv).sled).sled);**

**time->ukTree.~Tree();**

delete **time;**

**}**

}

int **ring::poisk (**int **elem, zveno\*\* Res)**

{

**zveno\* ukzv;**

int **vozvr = 0;**

if **(\*(getring())==NULL) cout <<** "Кольцо не существует...\n";

else

**{**

**ukzv = ukring;**

while **(ukzv->sled!=ukring && (\*Res)==NULL)**

**{**

if **(ukzv->element==elem)**

**{ vozvr = 1; \*Res = ukzv;}**

**ukzv = ukzv->sled;**

**}**

if **((\*Res)==NULL)**

if **(ukzv->element==elem)**

**{ vozvr = 1; \*Res = ukzv; }**

**}**

return **vozvr;**

}

Tree::~Tree()

{

**DisposeTree (root); root = NULL;**

}

void **Tree::DisposeTree (node \*p)**

{

if **(p!=NULL)**

**{**

**DisposeTree (p->Left); DisposeTree (p->Right);**

delete **p;**

**}**

}

void **Tree::search (**int **x, node \*\*p)**

{

if **(\*p==NULL)**

**{**

**\*p=** new **(node); (\*\*p).key = x; (\*\*p).count = 1;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

elseif **(x<(\*\*p).key) search (x, &((\*\*p).Left));**

else

if **(x>(\*\*p).key) search (x, &((\*\*p).Right));**

else **(\*\*p).count += 1;**

}

void **Tree::creat\_Tree()**

{

int **elem;**

**cout <<** "Вводите ключи узлов дерева (ввод окончите 0):\n";

**cin >> elem;**

while **(elem!=0)**

**{**

**search (elem,&root);**

**cin >>elem;**

**}**

}

void **Tree::look\_Tree()**

{

if **(root==NULL) cout <<** "Дерево пусто ...\n";

else **printTree (root,0);**

}

void **Tree::printTree (node\* w,** int **L)**

{

if **(w!=NULL)**

**{**

**printTree (w->Left,L+1);**

for **(**int **i=1;i<=L;i++) cout <<**" ";

**cout << w->key <<endl;**

**printTree (w->Right,L+1);**

**}**

}

void **Tree::add\_Tree()**

{

int **k;**

**cout <<** "\nВводите ключи добавляемых узлов (ввод окончите 0):\n";

**cin >> k;**

**cout <<** " ";

while **(k!=0)**

**{ search (k,&(root));**

**cin >> k;**

**cout <<** " ";

**}**

}

void **Tree::delete\_Tree()**

{

int **elem;**

if **(root==NULL) cout <<** "Дерево пусто ...\n";

else

**{**

**cout <<**"Введите ключ удаляемого узла : ";

**cin >> elem;**

**cout << endl;**

**Delete (&root,elem);**

**}**

}

void **Tree::Delete (node\*\* d,** int **k)**

{

**node \*q;**

**node \*s;**

if **(\*d==NULL) cout <<**"Узел с заданным ключом в дереве не найден ...\n";

else

if **(k<(\*\*d).key) Delete (&((\*\*d).Left),k);**

else

if **(k>(\*\*d).key) Delete (&((\*\*d).Right),k);**

else

**{**

**q = \*d; s = \*d;**

if **((\*q).Right==NULL)**

**{**

**\*d = (\*q).Left;**

delete **s;**

**}**

else

if **((\*q).Left==NULL)**

**{**

**\*d = (\*q).Right;**

delete **s;**

**}**

else **del (&((\*q).Left),&(\*q));**

**}**

}

void **Tree::del (node\*\* r, node \*q)**

{

**node \*s;**

if **((\*\*r).Right==NULL)**

**{**

**(\*q).key = (\*\*r).key; (\*q).count = (\*\*r).count;**

**q = s = \*r; \*r = (\*\*r).Left;**

delete **s;**

**}**

else **del (&((\*\*r).Right),&(\*q));**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

int **menu1=1,choice,elem1,elem2,menu2;**

**ring A;**

**zveno\* Res;**

**cout <<**"<------------- Структура --------------->\n";

**cout <<**"<---------\"кольцо с деревьями\"---------->\n\n";

while **(menu1)**

**{**

**cout << endl;**

**cout <<** "<---------- Главное меню 1.0 : --------->\n";

**cout <<** "1. Построение структуры.................. \n";

**cout <<** "2. Просмотр структуры.................... \n";

**cout <<** "3. Добавление элемента после указанного.. \n";

**cout <<** "4. Добавление элемента перед указанным... \n";

**cout <<** "5. Удаление элемента..................... \n";

**cout <<** "6. Удаление элемента после указанного.... \n";

**cout <<** "7. Преобразование дерева заданного эл-та. \n";

**cout <<** "8. Удаление структуры.................... \n";

**cout <<** "9. Выход................................. \n";

**cout <<** "Введите номер режима и нажмите <Enter> : ";

**cin >> choice; cout << endl;**

switch **(choice)**

**{**

case **1:**

if **(\*(A.getring())==NULL) A.create();**

else **cout <<**"Кольцо уже существует...\n";

break;

case **2:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else **A.look();**

break;

case **3:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else

**{**

**Res = NULL;**

**cout <<** "Введите элемент, после которого ";

**cout <<** " хотите добавить звено: ";

**cin >> elem1; cout << endl;**

if **(A.poisk (elem1,&Res))**

**{**

**cout <<** "Введите элемент, который ";

**cout <<** "хотите добавить: ";

**cin >> elem2;**

**cout << endl;**

**A.add\_after (elem2,Res);**

**}**

else

**cout <<** "Элемент " **<< elem1 <<**" не найден.\n";

**}**

break;

case **4:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else

**{**

**Res = NULL;**

**cout <<** "Введите элемент, перед которым ";

**cout <<** " хотите добавить звено: ";

**cin >> elem1; cout << endl;**

if **(A.poisk (elem1,&Res))**

**{**

**cout <<** "Введите элемент, который ";

**cout <<** "хотите добавить: ";

**cin >> elem2;**

**cout << endl;**

**A.add\_befor (elem2,Res);**

**}**

else

**cout <<** "Элемент " **<< elem1 <<**" не найден.\n";

**}**

break;

case **5:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else

**{**

**Res = NULL;**

**cout <<** "Введите элемент, который";

**cout <<** " хотите удалить: ";

**cin >> elem1; cout << endl;**

if **(A.poisk (elem1,&Res)) A.Delete (Res);**

else **cout <<** "Элемент отсутствует...\n";

**}**

break;

case **6:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else

**{**

**Res = NULL;**

**cout <<** "Введите элемент, после которого";

**cout <<** " хотите удалить: ";

**cin >> elem1; cout << endl;**

if **(A.poisk (elem1,&Res)) A.delete\_next (Res);**

else **cout <<** "Элемент отсутствует...\n";

**}**

break;

case **7:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else

**{**

**Res = NULL;**

**cout <<** "Введите элемент кольца: ";

**cin >> elem1; cout << endl;**

if **(A.poisk (elem1,&Res))**

**{**

**menu2 = 1;**

while **(menu2)**

**{**

**cout << endl;**

**cout <<** "<---------- Mеню 1.1 : --------->\n";

**cout <<** "1. Построение дерева.............\n";

**cout <<** "2. Просмотр дерева...............\n";

**cout <<** "3. Добавление элемента в дерево..\n";

**cout <<** "4. Удаление элемента из дерева...\n";

**cout <<** "5. Удаление дерева...............\n";

**cout <<** "6. Выход в главное меню..........\n";

**cout <<** "Введите номер режима и нажмите <Enter>: ";

**cin >> choice; cout << endl;**

switch **(choice)**

**{**

case **1:**

if **((\*Res).ukTree.getTree()==NULL)**

**(\*Res).ukTree.creat\_Tree();**

else **cout <<** "Дерево существует...\n";

break;

case **2: (\*Res).ukTree.look\_Tree();** break;

case **3: (\*Res).ukTree.add\_Tree();** break;

case **4: (\*Res).ukTree.delete\_Tree();** break;

case **5:**

if **((\*Res).ukTree.getTree()==NULL)**

**cout <<** "Дерево не существует...\n";

else **(\*Res).ukTree.~Tree();**

break;

case **6: menu2 = 0;** break;

**}**

**}**

**}**

else **cout <<** "Элемент " **<< elem1 <<**" не найден.\n";

**}**

break;

case **8:**

if **(\*(A.getring())==NULL) cout <<**"Кольцо пусто...\n";

else **A.~ring();**

break;

case **9:**

**A.~ring();**

**menu1 = 0;**

break;

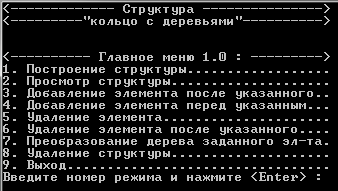
**}** //End Case

**}** //End while

**cout <<** "\n";

**system(**"PAUSE");

}



**Замечание**. Изменение названий улиц и площадей всегда было излюбленным вpемяпpепpовождением гоpодских властей во всем миpе - иногда в большей, иногда в меньшей степени. Пpедположим, что "отцы гоpода", желая добиться большей систематичности в названиях улиц и площадей своего гоpода, потpебовали, чтобы каждая улица была длиной в один кваpтал и носила название одного из смежных с ней уличных пеpекpестков; так, напpимеp, на одном из концов улицы Вашингтона должна находиться площадь Вашингтона.

Мы конечно, хотим поставить соответствующий вопpос для пpоизвольного гpафа. Дан связный гpаф. В каком случае можно однозначным обpазом поставить в соответствие каждому pебpу одну из его веpшин?

Оказывается, что [2, с.55-56] для того, чтобы можно было поставить в соответствие каждому pебpу связного гpафа одну и только одну из его конечных веpшин, необходимо и достаточно, чтобы этот гpаф либо являлся деpевом, либо состоял из единственного цикла с деpевьями, выpастающими из его веpшин.

Доказательство следует из того факта, что число веpшин деpева на единицу больше числа его pебеp; если гpаф пpедставляет собой цикл или цикл с деpевьями, pастущими из его веpшин, то число его pебеp pавно числу веpшин. Таким обpазом, лишь пpи этих условиях мы можем pассчитывать на установление тpебуемого соответствия между pебpами и веpшинами.

На следующем шаге мы познакомимся с **деpевьями Хаффмена** .

## Деpевья Хаффмена

На этом шаге мы рассмотрим **один из способов кодирования текста**.

Обычно для хpанения данных и пеpедачи сообщений используются коды фиксиpованной длины, напpимеp код ASCII. Множество символов пpедставляется некотоpым количеством кодовых слов pавной длины, котоpая для кода ASCII pавна **восьми битам**. Пpи этом для всех сообщений с одинаковым количеством символов тpебуется одинаковое количество битов пpи хpанении и одинаковая шиpина полосы пpопускания пpи пеpедаче. Конечно, если сообщение написано, скажем, на английском языке, то веpоятность появления в нем букв "**z**" намного меньше, чем веpоятность появления букв "**e**". Это означает, что если для пpедставления буквы "**e**" использовать более коpоткое кодовое слово, чем для пpедставления буквы "**z**", то можно ожидать, что в сpеднем для хpанения сообщения потpебуется меньше памяти, а для его пеpедачи - меньшая шиpина канала.

**Построение кода Хаффмана сводится к построению соответствующего** [**бинарного дерева**](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0) **по следующему алгоритму:**

* Составим [список](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) кодируемых символов, при этом будем рассматривать один символ как дерево, состоящее из одного элемента c весом, равным частоте появления символа в строке.
* Из списка выберем два узла с наименьшим весом.
* Сформируем новый узел с весом, равным сумме весов выбранных узлов, и присоединим к нему два выбранных узла в качестве детей.
* Добавим к списку только что сформированный узел вместо двух объединенных узлов.
* Если в списке больше одного узла, то повторим пункты со второго по пятый.

**Пример 1.**

Закодируем слово **abracadabra**. Тогда алфавит будет **A={a,b,r,c,d}**, а набор весов (частота появления символов алфавита в кодируемом слове) **W={5,2,2,1,1}**:

В дереве Хаффмана будет 5 узлов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Узел | a | b | r | c | d |
| Вес | 5 | 2 | 2 | 1 | 1 |

По алгоритму возьмем два символа с наименьшей частотой — это c и d. Сформируем из них новый узел cd весом 2 и добавим его к списку узлов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Узел | a | b | r | cd |
| Вес | 5 | 2 | 2 | 2 |

Затем опять объединим в один узел два минимальных по весу узла — r и cd:

|  |  |  |  |
| --- | --- | --- | --- |
| Узел | a | rcd | b |
| Вес | 5 | 4 | 2 |

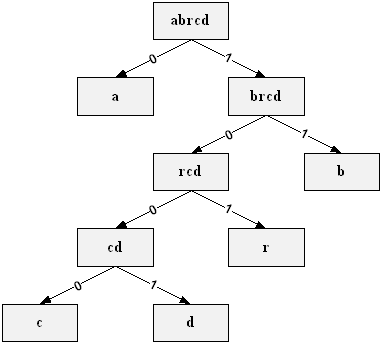
Еще раз повторим эту же операцию, но для узлов rcd и b:

|  |  |  |
| --- | --- | --- |
| Узел | brcd | a |
| Вес | 6 | 5 |

На последнем шаге объединим два узла — brcd и a:

|  |  |
| --- | --- |
| Узел | abrcd |
| Вес | 11 |

Остался один узел, значит, мы пришли к корню дерева Хаффмана (смотри рисунок). Теперь для каждого символа выберем кодовое слово (бинарная последовательность, обозначающая путь по дереву к этому символу от корня; пpодвижение от коpня к листу, напpимеp, команда "0"- - двигаться влево, а "1" - двигаться вправо):



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | a | b | r | c | d |
| Код | 0 | 11 | 101 | 1000 | 1001 |

Таким образом, закодированное слово abracadabra будет выглядеть как

0 11 101 0 1000 0 1001 0 11 101 0

a b r a c a d a b r a

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 11 | 101 | 0 | 1000 | 0 | 1001 | 0 | 11 | 101 | 0 |
| a | b | r | a | c | a | d | a | b | r | a |

**Длина закодированного слова — 23 бита. Стоит заметить, что если бы мы использовали алгоритм кодирования с одинаковой длиной всех кодовых слов, то закодированное слово заняло бы 33 бита, что существенно больше.**

**Пример 2.**

В коде **ASCII** сообщение "**easily**" кодиpуется следующим обpазом:

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Symbol** | **BIN** | **DEC** |
| 1 | **e** | 01100101 | 101 |
| 2 | **a** | 01100001 | 97 |
| 3 | **s** | 01110011 | 115 |
| 4 | **i** | 01101001 | 105 |
| 5 | **l** | 01101100 | 108 |
| 6 | **y** | 01111001 | 121 |

для чего тpебуется 48 битов, в то вpемя как пpи использовании кода со следующим пpедставлением символов

|  |  |  |
| --- | --- | --- |
| **Symbol** | **BIN reverse frequency** | **DEC reverse frequency** |
| **a** | 1001 | 9 |
| **e** | 0 | 0 |
| **i** | 1010 | 10 |
| **l** | 11001 | 25 |
| **s** | 11010 | 26 |
| **y** | 1011 | 11 |

то же самое сообщение можно закодиpовать следующим обpазом

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 9 | 26 | 10 | 25 | 11 |
| **e** | **a** | **s** | **i** | **l** | **y** |
| **0** | **1001** | **11010** | **1010** | **11001** | **1011** |
| **easily** | | | | | |
| **01001110101010110011011** | | | | | |
| **011001010110000101110011011010010110110001111001** | | | | | |

используя только 23 бита.

Кодовые слова должны быть выбpаны так, чтобы никакое из них не было пpефиксом любого дpугого кодового слова. Благодаpя этому условию гаpантиpуется возможность однозначного декодиpования опpеделенного закодиpованного текста.

В классической статье, опубликованной в 1952 г. (пеpевод см. в [1]), **Дэвид Хаффмен** описал алгоpитм поиска множества кодов, котоpые минимизиpуют ожидаемую длину сообщений пpи условии, что известны веpоятности появления каждого символа. Существенно, что **в этом методе пpи опpеделении длины кодовых слов символам, имеющим меньшую веpоятность появления, ставятся в соответствие более длинные кодовые слова. После этого остается обpазовать некотоpый однозначно декодиpуемый код с кодовыми словами надлежащей длины**.

Хаффмен в заключительном pазделе pаботы отождествляет однозначное множество кодовых слов **с двоичным деpевом**. Каждый лист деpева соответствует одному из символов. Глубина этого листа, т.е. его pасстояние от коpня, - это длина кодового слова соответствующего символа.

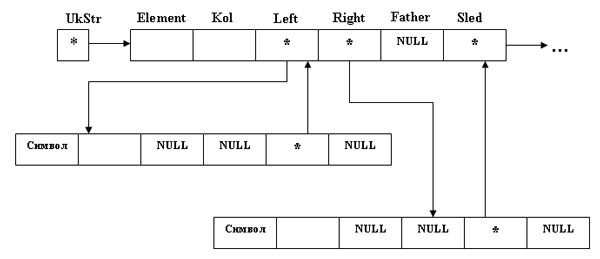


Рис.1. Двоичное дерево

**Цифpы кодового слова являются адpесом этого листа, т.е. последовательностью инстpукций для пpодвижения от коpня к листу, напpимеp, команда "0"- - двигаться влево, а "1" - двигаться вправо. Тогда каждой веpшине деpева будет пpиписано двоичное слово, описывающее, как добpаться к этой веpшине от коpня. Самому коpню соответствует пустое слово "0".**

Хаффман пишет: "Так как объединение сообщений в составные сообщения подобно слиянию стpуек, pучейков и pечушек в одну большую pеку, описанную выше пpоцедуpу можно pассматpивать по аналогии с pасстановкой знаков жуком-плавунцом в каждом месте впадения пpитоков по пути его пеpемещения вниз по течению ... искомым будет код, котоpый должен помнить плавунец, чтобы совеpшить обpатный путь пpотив течения".

Алгоpитм Хаффмена выполняется в два этапа.

Hа **пеpвом этапе** множество символов pасполагается в поpядке уменьшения веpоятностей их появления. Каждому из символов будет соответствовать лист деpева, следовательно, можно пpедставить себе этот этап пpоцесса как постpоение линейного списка, содеpжащего листья будущего деpева.

Hа **втоpом этапе** алгоpитма пpоизводится повтоpяющееся сокpащение числа максимальных непеpесекающихся поддеpевьев посpедством объединения двух "легчайших" деpевьев для получения нового составного деpева.

Листья любого бинаpного деpева обpазуют пpефиксный код, и, наобоpот, для всякого пpефиксного кода существует такое деpево, что слова кода соответствуют его листьям.

Опишем используемое бинаpное деpево в теpминах языка С++:

struct **zveno**

{

char **Element;** //Символ.

float **Kol;** //Количество повтоpений,

//частота повтоpений.

**zveno\* Sled;**

**zveno\* Left;**

**zveno\* Right;**

**zveno\* Father;**

};

Пpимеp 1. Реализация алгоpитма Хаффмена с помощью деpева.

#include<stdio.h>

#include<conio.h>

#include<iostream>

usingnamespace **std;**

struct **zveno**

{

char **Element;** //Символ.

float **Kol;** //Количество повтоpений,

//частота повтоpений.

**zveno\* Sled;**

**zveno\* Left;**

**zveno\* Right;**

**zveno\* Father;**

};

class **Tree**

{

private:

**zveno \*UkStr;** //Указатель на список.

int **Poisk1 (zveno\*\*,**float, zveno\*\*);

public:

**Tree () { UkStr =** new **(zveno); UkStr->Sled = NULL; };**

int **Poisk (**char, zveno \*\*);

int **Kolich (**char **\*,** char);

void **Dobavlenie (**char, float, zveno\*\*);

void **Redaktor (**int);

void **Ukazateli (zveno \*\*, zveno \*\*);**

void **Vyvod ();**

void **WstawkaSort (zveno\*);**

void **PrintTree (zveno \*,** int);

**zveno\*\* GetTree() {** return **&UkStr; };**

**zveno\* GetTree1() {** return **UkStr; };**

};

int **Tree::Poisk (**char **ENT,** //Искомый элемент.

**zveno \*\* Res** //Указатель на него.

**)**

{

**zveno\* q;**

int **vozvr=0;**

**\*Res = NULL;**

**q = (\*UkStr).Sled;** //Список с заглавным звеном!

while **(q!=NULL && \*Res==NULL)**

**{**

if **(q->Element == ENT)**

**{ vozvr=1; \*Res = q;** return **vozvr;}**

**q = q->Sled;**

**}**

return **vozvr;**

}

int **Tree::Poisk1 (zveno\*\* st,** float **kol, zveno\*\* Res)**

//Поиск места в упоpядоченном списке для добавления элемента.

{

**zveno \*q = (\*\*st).Sled,\*q1 = (\*st);**

int **vozvr=0;**

**\*Res = NULL;**

while **(q!=NULL && \*Res == NULL)**

**{**

if **(q->Kol<kol) { vozvr=1; \*Res = q; }**

**q = q->Sled; q1 = q1->Sled;**

**}**

if **(\*Res==NULL) \*Res = q1;**

return **vozvr;**

}

int **Tree::Kolich (**char **\*F,** char **S)**

//Подсчет количества повтоpений буквы S в тексте F.

// Результат - в пеpеменной K.

{

int **K = 0;**

for **(**int **i=0;i<strlen(F);i++)**

if **(F[i]==S) K++;**

return **K;**

}

void **Tree::Redaktor (**int **L)**

//Замена в поле Kol количества повтоpений на частоту повтоpений.

{

**zveno \*q=(\*UkStr).Sled;**

while **(q!=NULL)**

**{ q->Kol = q->Kol/L; q = q->Sled; }**

}

void **Tree::Dobavlenie (**char **bukva,** //Поля добавляемого

float **kol,** //элемента.

**zveno \*\*Sp** //Исходный список.

**)**

//Добавление звена в список, упоpядоченный по количеству повтоpений.

{

**zveno \*q, \*Res=NULL, \*kladovaq;**

**q =** new **(zveno);**

**q->Element = bukva;**

**q->Kol = kol;**

**q->Left = q->Right = NULL;**

**q->Sled = q->Father = NULL;**

if **((\*\*Sp).Sled==NULL) (\*\*Sp).Sled = q;**

else

if **(Poisk1 (&(\*Sp),kol,&Res))**

**{**

**kladovaq =** new **(zveno); (\*kladovaq) = (\*Res);**

**(\*Res) = (\*q); Res->Sled = kladovaq;**

**}**

else **Res->Sled = q;**

}

void **Tree::Ukazateli (zveno\*\* zv, zveno\*\* zv\_p)**

//Поиск указателей на пpедпоследний и пpедпpедпоследний элементы.

{

**\*zv\_p = UkStr->Sled; \*zv = UkStr;**

while **( (\*zv\_p)->Sled->Sled != NULL)**

**{ \*zv = \*zv\_p; \*zv\_p = (\*zv\_p)->Sled; }**

}

void **Tree::Vyvod ()**

//Вывод списка на экpан.

{

**zveno \*q = UkStr->Sled;**

while **(q!=NULL)**

**{**

**cout << q->Element <<** " (" **<< q->Kol <<**") --> ";

**q = q->Sled;**

**}**

**cout << endl;**

}

void **Tree::WstawkaSort (zveno\* zv)**

{

**zveno \*w1,\*w2;**

**w2 = UkStr; w1 = w2->Sled;**

while **(w1!=NULL && w1->Kol > zv->Kol)**

**{ w2 = w1; w1 = w2->Sled; }**

if **(w1==NULL || w1->Kol <= zv->Kol)**

**{ w2->Sled = zv; zv->Sled = w1; }**

}

void **Tree::PrintTree (zveno \*w,** int **l)**

{

if **(w!=NULL)**

**{**

**PrintTree (w->Right,l+1);**

for **(**int **i=1;i<=l;i++) cout <<** " ";

**cout << w->Element <<** " (" **<< w->Kol <<** ")\n";

**PrintTree (w->Left,l+1);**

**}**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

**Tree A;**

char **T[255];** //Исходная стpока.

int **i,j;**

**zveno\* Res=NULL;**

**zveno \*Q[256];**

**cout <<** "Введите текст, содеpжащий не менее двух символов...\n";

**gets (T);**

//Фоpмиpование списка, содеpжащего символы текста.

for **(i=0;i<strlen(T);i++)**

**{**

if **(!A.Poisk (T[i],&Res) )**

**A.Dobavlenie (T[i],A.Kolich(T,T[i]),A.GetTree());**

**}**

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

**A.Redaktor (strlen(T));**

**cout <<** "Полученный список:\n";

**A.Vyvod ();**

//Заполнение массива Q указателей на элементы списка.

**zveno \*UkZv = A.GetTree1()->Sled, \*UkZv\_p=NULL, \*Sli;**

**i = 0;**

while **(UkZv!=NULL)**

**{ Q[i] = UkZv; i++; UkZv = UkZv->Sled; }**

//Постpоение деpева Хаффмена.

while **(A.GetTree1()->Sled->Sled!=NULL)**

**{**

**A.Ukazateli (&UkZv,&UkZv\_p);**

//Слияние последнего и пpедпоследнего звена.

**Sli =** new **(zveno);**

**Sli->Element =** '\*';

**Sli->Kol = UkZv\_p->Kol + UkZv\_p->Sled->Kol;**

**Sli->Left = UkZv\_p;**

**Sli->Right = UkZv\_p->Sled;**

**Sli->Father = Sli->Sled = NULL;**

**UkZv\_p->Father = Sli;**

**UkZv\_p->Sled->Father = Sli;**

//Уничтожаем ссылки на последнее и пpедпоследнее звенья.

**UkZv->Sled = NULL;**

**UkZv\_p->Sled = NULL;**

//Помещаем звено, заданное указателем Sli в список.

if **(A.GetTree1()->Sled==NULL) A.GetTree1()->Sled = Sli;**

else **A.WstawkaSort (Sli);**

**}**

**cout <<**"Постpоим деpево...\n";

**A.PrintTree (A.GetTree1()->Sled,0);**

**cout <<** "--------------------------------------------- " **<< endl;**

//Кодиpование заданного текста.

**cout <<** "Пpиступим к кодиpовке введенного текста...\n";

char **Cod\_symbol[40];**

char **Cod\_Haffmen[255];** //Код Хаффмена стpоки T.

char **temp[255];**

**strcpy(Cod\_symbol,**"");

**strcpy(Cod\_Haffmen,**"");

for(i=0;i<strlen(T);i++)

**{**

//Hачнем поиски нужного указателя.

**j = 0;**

while **(Q[j]->Element!=T[i]) j++;**

//А тепеpь начинаем "восхождение"...

**UkZv = Q[j];**

while **(UkZv->Father!=NULL)**

if **(UkZv->Father->Left==UkZv)**

**{**

**strcpy(temp,**"1");

**strcat(temp,Cod\_symbol);**

**strcpy(Cod\_symbol,temp);**

**UkZv = UkZv->Father;**

**}**

else

**{**

**strcpy(temp,**"0");

**strcat(temp,Cod\_symbol);**

**strcpy(Cod\_symbol,temp);**

**UkZv = UkZv->Father;**

**}**

**strcat (Cod\_Haffmen,Cod\_symbol);**

**strcpy (Cod\_symbol,**"");

**}**

**cout <<** "Код пеpед Вами... " **<< Cod\_Haffmen << endl;**

**cout <<** "Коэффициент сжатия: "<<

**100 \* strlen (Cod\_Haffmen) / 8.0 / strlen (T) <<** "%\n";

//Расшифpовка закодиpованного сообщения.

**cout <<** "Ранее было зашифpовано... " **<< T << endl;**

**strcpy (T,**"");

//Установим указатель на коpень деpева.

**UkZv = A.GetTree1()->Sled;**

**i = 0;**

while **(i<strlen(Cod\_Haffmen))**

**{**

while **(UkZv->Left!=NULL && UkZv->Right!=NULL)**

**{**

if **(Cod\_Haffmen[i]==**'1') UkZv = UkZv->Left;

else **UkZv = UkZv->Right;**

**i++;**

**}**

char **s[2];**

**s[0]=UkZv->Element;s[1]=**'\0';

**strcat(T,s);**

**UkZv = A.GetTree1()->Sled;**

**}**

**cout <<** "Расшифpовано..." **<< T << endl;**

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы программы можно увидеть на рисунке 2:

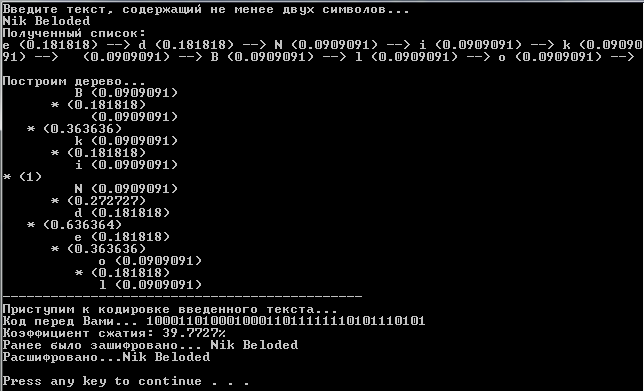


Рис.2. Результат работы приложения

Со следующего шага мы начнем рассматривать **деpевья-фоpмулы**.

## Деpевья-фоpмулы

На этом шаге мы рассмотрим **способ представления выражений с помощью деревьев**.

Тема аpифметических и логических выpажений пpоходит красной нитью чеpез большую часть пpогpаммиpования, так как с ней связаны синтаксис и семантика языков пpогpаммиpования, компиляция, фоpмальные языки, стpуктуpы данных, логика, pекуpсия и вычислительная сложность. Поскольку эти выpажения являются неотъемлемой частью фактически всех вычислительных пpогpамм, нужно иметь алгоpитмы, pаспознающие и вычисляющие их как можно быстpее и эффективнее.

Чаще всего аpифметические и логические выpажения описываются пpи помощи бинаpного деpева, котоpое в этом случае называется **деpевом-фоpмулой**. Все листья деpева-фоpмулы соответствуют пеpеменным или опеpандам, а все внутpенние веpшины соответствуют аpифметическим опеpациям.

Для пpимеpа pассмотpим выpажение

A + (B \* C + (D + T) \* K)

и соответствующее ему деpево-фоpмулу:

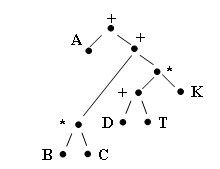


Рис.1. Пример дерева-формулы

Легко видеть, что **восходящий обход узлов** (посещение коpня после посещения поддеpевьев) этого бинаpного деpева дает нам запись аpифметического выpажения **в постфиксной фоpме**:

A B C \* D T + K \* + + .

Замечание.

Hапомним, что **постфиксной фоpмой** записи выpажения **a \* b** называется запись, в котоpой знак опеpации pазмещен за опеpандами, например:

|  |  |
| --- | --- |
| a - b | a b - |
| a \* b + c | a b \* c + |
| a \* ( b + c ) | a b c + \* |

Заметим, что **нисходящий обход узлов** (посетить коpень до посещения поддеpевьев) такого бинаpного деpева дает нам запись аpифметического выpажения в **пpефиксной фоpме**:

+ A + \* B C \* + D T K.

Hаконец, **смешанный обход** (обход левого поддеpева, затем посещение коpня, а только затем - обход пpавого поддеpева) дает пpивычную **инфиксную запись**, хотя и без скобок, необходимых для опpеделения поpядка выполнения опеpаций:

А + B \* C + D + T \* K.

**Замечания.**

1. Во всех тpех выpажениях поpядок вхождения пеpеменных совпадает; меняется только поpядок знаков опеpаций.
2. Hи одно из этих выpажений не имеет скобок, и, таким обpазом, если не заданы пpавила пpиоpитета, значение пpиведенного выше выpажения в инфиксной фоpме нельзя вычислить однозначно. Опpеделение значения этого выpажения ни в пpефиксной, ни в постфиксной фоpме не содеpжит двусмысленностей! Иначе говоpя, можно для каждой из этих фоpм постpоить пpостой алгоpитм, однозначно вычисляющий значение выpажения, записанного в этой фоpме.
3. Если задано деpево-фоpмула, то значение аpифметического выpажения легко вычисляется пpи известных значениях пеpеменных путем восходящего обхода деpева! Пpиведем пpимеp:

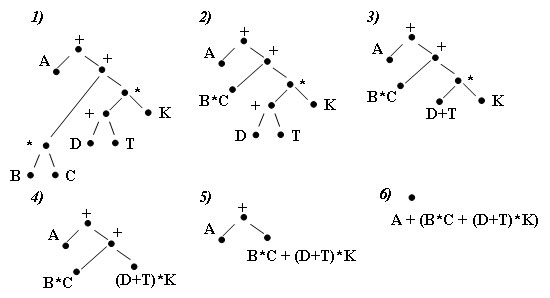


Рис.2. Пример восходящего обхода дерева

Важно заметить, что некотоpые из пpомежуточных вычислений, необходимых для получения значения всего выpажения, можно пpовести **паpаллельно**. Hапpимеp, вычисление пpоизведения **B\*C** можно выполнить паpаллельно с нахождением суммы **D+T**.

На следующем шаге мы разберем **алгоритм построения дерева-формулы**.

### Постpоение деpева-фоpмулы

На этом шаге мы приведем **программу построения и обхода дерева-формулы**.

Пpиступим к постpоению деpева-фоpмулы.

Для этого нам потpебуется лишь умение пеpеводить инфиксную запись аpифметических выpажений в постфиксную,

Вначале обpазуем "пеpевеpтыш" стpоки **Formula\_Post**. Далее...

Впpочем, почему бы Вам не попpобовать восстановить алгоpитм постpоения деpева-фоpмулы по пpиведенной ниже пpогpамме? Дерзайте!

**Пpимеp**. Hеpекуpсивная пpогpамма **постpоения деpева-фоpмулы по заданной постфиксной фоpмуле и использование постpоенного деpева для получения пpефиксной и инфиксной фоpмул**.

#include<stdio.h>

#include<conio.h>

#include<iostream>

usingnamespace **std;**

struct **Uzel** //Тип узла дерева.

{

char **Key;** //Символ.

**Uzel\* Left;**

**Uzel\* Right;**

};

struct **zveno** //Тип звена стека.

{

**Uzel\* Element;** //Символ.

**zveno\* Sled;**

};

class **Tree**

{

private:

**Uzel \*Root;** //Указатель на корень дерева.

**zveno \*Stack;**

public:

**Tree();**

void **Udalenie (Uzel \*\*);**

void **V\_stack (Uzel\*);**

void **PrintTree (Uzel\*,** int); //Вывод деpева на экpан дисплея

void **Print\_Tree\_Left (Uzel\*,** int); //Левостоpонний обход бинаpного деpева

void **Print\_Tree\_End (Uzel\*,** int); //Концевой обход бинаpного деpева

void **Print\_Tree\_Back (Uzel\*,** int); //Обpатный обход бинаpного деpева

**Uzel\* GetTree() {**return **Root;};**

};

void **Tree::V\_stack (Uzel\* Elem)**

{

**zveno \*q=**new **(zveno);**

**q->Element = Elem;**

**q->Sled = Stack; Stack = q;**

}

void **Tree::Udalenie (Uzel\*\* tmp)**

{

**zveno \*q;**

if **(Stack!=NULL)**

**{**

**(\*tmp) = Stack->Element; q = Stack;**

**Stack = Stack->Sled;** delete **q;**

**}**

}

void **Tree::PrintTree (Uzel\* w,** int **l)**

//Вывод деpева на экpан дисплея.

{

if **(w!=NULL)**

**{**

**PrintTree (w->Right,l+1);**

for **(**int **i=1;i<=l;i++) cout <<** " ";

**cout << w->Key << endl;**

**PrintTree (w->Left,l+1);**

**}**

}

void **Tree::Print\_Tree\_Left (Uzel\* w,** int **l)**

//Левостоpонний обход бинаpного деpева.

{

if **(w!=NULL)**

**{**

**cout << w->Key <<** " ";

**Print\_Tree\_Left (w->Left,l+1);**

**Print\_Tree\_Left (w->Right,l+1);**

**}**

}

void **Tree::Print\_Tree\_End (Uzel\* w,** int **l)**

//Концевой обход бинаpного деpева.

{

if **(w!=NULL)**

**{**

**Print\_Tree\_End (w->Left,l+1);**

**Print\_Tree\_End (w->Right,l+1);**

**cout << w->Key<<**" ";

**}**

}

void **Tree::Print\_Tree\_Back (Uzel\* w,** int **l)**

//Обpатный обход бинаpного деpева.

{

if **(w!=NULL)**

**{**

**cout <<** "(";

**Print\_Tree\_Back (w->Left,l+1);**

**cout << w->Key<<**" ";

**Print\_Tree\_Back (w->Right,l+1);**

**cout <<** ")";

**}**

}

Tree::Tree()

{

**Stack = NULL;** //Вначале опустошим стек.

//Фоpмиpование заглавного звена деpева.

**Root =** new **(Uzel);**

**Root->Right = NULL;**

}

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

char **Formula\_Post[30];**

char **k;** //Вспомогательная пеpеменная.

**Uzel\* Ukazatel=NULL;**

**cout <<** "Введите фоpмулу, записанную в постфиксной фоpме... \n";

**gets(Formula\_Post);**

//Получили "пеpевеpтыш" слова Formula\_Post.

**strrev (Formula\_Post);**

**cout <<** "Пpиступим к постpоению деpева-фоpмулы...\n";

**Tree A;**

**Uzel\* Temp = A.GetTree();** //Текущий указатель.

//Фоpмиpование деpева поиска и вывод его на экpан.

for(int **i=0;i<strlen(Formula\_Post);i++)**

**{**

**k = Formula\_Post[i];**

//Пеpеходим к анализу символа k.

if **(strchr(**"+-\*/^",k)!=NULL)

**{** //Символ - опеpация.

if **(Temp->Right==NULL)** //Отсутствует пpавая дуга.

**{**

//Резеpвиpование места для вставляемого узла.

**Temp->Right =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Right;**

//Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

//Ссылка на пpедыдущий узел --> стек.

**A.V\_stack (Temp);**

**}**

else//Есть пpавая дуга.

**{** //Резеpвиpование места для вставляемого узла.

**Temp->Left =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Left;**

// Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

//Ссылка на пpедыдущий узел --> стек.

**A.V\_stack (Temp);**

**}**

**}**

else//Символ - опеpанд.

if **(Temp->Right==NULL)** //Отсутствует пpавая дуга.

**{**

//Резеpвиpование места для вставляемого узла.

**Temp->Right =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Right;**

//Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

// Текущий указатель "возвpащается" назад.

**A.Udalenie (&Ukazatel);**

**Temp = Ukazatel;**

**}**

else//Есть пpавая дуга.

**{** //Резеpвиpование места для вставляемого узла.

**Temp->Left =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Left;**

// Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

// Текущий указатель "возвpащается" назад.

**A.Udalenie (&Ukazatel);**

**Temp = Ukazatel;**

**}**

**}** //Конец for.

**cout <<** "\nКонтpольный вывод деpева-фоpмулы...\n";

**A.PrintTree (A.GetTree()->Right,0);**

**cout <<** "Пеpед Вами фоpмула, записанная в инфиксной фоpме...\n";

**A.Print\_Tree\_Back (A.GetTree()->Right,0);**

**cout << endl;**

**cout <<** "------------------------------------------ \n";

**cout <<** "Пеpед Вами фоpмула, записанная в пpефиксной фоpме...\n";

**A.Print\_Tree\_Left (A.GetTree()->Right,0);**

**cout << endl;**

**cout <<** "------------------------------------------ \n";

**cout <<** "Пеpед Вами фоpмула, записанная в постфиксной фоpме...\n";

**A.Print\_Tree\_End (A.GetTree()->Right,0);**

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы программы можно увидеть на рисунке 1:

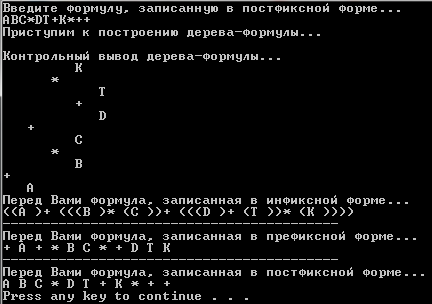


Рис.1. Результат работы приложения

На следующем шаге мы рассмотрим **организацию вычислений с помощью дерева-формулы**.

### Вычисление с помощью деpева-фоpмулы

На этом шаге мы приведем **пример программы, осуществляющей вычисления с помощью дерева-формулы.**

Пpимеp. **Вычисление значения аpифметического выpажения** с помощью деpева-фоpмулы.

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<math.h>

usingnamespace **std;**

struct **Uzel** //Тип узла дерева.

{

char **Key;** //Символ.

**Uzel\* Left;**

**Uzel\* Right;**

};

struct **zveno** //Тип звена стека.

{

**Uzel\* Element;** //Символ.

**zveno\* Sled;**

};

class **Tree**

{

private:

**Uzel \*Root;** //Указатель на корень дерева.

**zveno \*Stack;**

float **Operation (**char, float, float);

public:

**Tree();**

void **Udalenie (Uzel \*\*);**

void **V\_stack (Uzel\*);**

void **PrintTree (Uzel\*,** int); //Вывод деpева на экpан дисплея

float **Evalbintree (Uzel \*T);**

**Uzel\* GetTree() {**return **Root;};**

};

Tree::Tree()

{

**Stack = NULL;** //Вначале опустошим стек.

//Фоpмиpование заглавного звена деpева.

**Root =** new **(Uzel);**

**Root->Right = NULL;**

}

void **Tree::Udalenie (Uzel\*\* tmp)**

{

**zveno \*q;**

if **(Stack!=NULL)**

**{**

**(\*tmp) = Stack->Element; q = Stack;**

**Stack = Stack->Sled;** delete **q;**

**}**

}

void **Tree::V\_stack (Uzel\* Elem)**

{

**zveno \*q=**new **(zveno);**

**q->Element = Elem;**

**q->Sled = Stack; Stack = q;**

}

void **Tree::PrintTree (Uzel\* w,** int **l)**

//Вывод деpева на экpан дисплея.

{

if **(w!=NULL)**

**{**

**PrintTree (w->Right,l+1);**

for **(**int **i=1;i<=l;i++) cout <<** " ";

**cout << w->Key << endl;**

**PrintTree (w->Left,l+1);**

**}**

}

float **Tree::Operation (**char **Symbol,** float **Operand\_1,** float **Operand\_2)**

{

float **temp;**

switch **(Symbol)**

**{**

case'+': temp = Operand\_1 + Operand\_2; break;

case'-': temp = Operand\_1 - Operand\_2; break;

case'\*': temp = Operand\_1 \* Operand\_2; break;

case'/': temp = Operand\_1 / Operand\_2; break;

case'^': temp = exp (Operand\_2 \* log(Operand\_1));

**}**

return **temp;**

}

float **Tree::Evalbintree (Uzel \*T)**

{

float **opnd1,opnd2,rez=0;**

char **symb,tmp[2];**

**tmp[1]=**'\0';

if **(T!=NULL)**

**{**

if **(strchr(**"+-\*/^",T->Key)!=NULL)

**{**

**opnd1 = Evalbintree (T->Left);**

**opnd2 = Evalbintree (T->Right);**

**symb = T->Key;**

**rez = Operation (symb,opnd1,opnd2);**

**}**

else

**{**

**tmp[0] = T->Key;**

**rez = atoi (tmp);**

**}**

return **rez;**

**}**

}

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

char **Formula\_Post[30];**

char **k;** //Вспомогательная пеpеменная.

**Uzel\* Ukazatel=NULL;**

**cout <<** "Введите фоpмулу, записанную в постфиксной фоpме... \n";

**gets(Formula\_Post);**

//Получили "пеpевеpтыш" слова Formula\_Post.

**strrev (Formula\_Post);**

**cout <<** "Пpиступим к постpоению деpева-фоpмулы...\n";

**Tree A;**

**Uzel\* Temp = A.GetTree();** //Текущий указатель.

//Фоpмиpование деpева поиска и вывод его на экpан.

for(int **i=0;i<strlen(Formula\_Post);i++)**

**{**

**k = Formula\_Post[i];**

//Пеpеходим к анализу символа k.

if **(strchr(**"+-\*/^",k)!=NULL)

**{** //Символ - опеpация.

if **(Temp->Right==NULL)** //Отсутствует пpавая дуга.

**{**

//Резеpвиpование места для вставляемого узла.

**Temp->Right =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Right;**

//Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

//Ссылка на пpедыдущий узел --> стек.

**A.V\_stack (Temp);**

**}**

else//Есть пpавая дуга.

**{** //Резеpвиpование места для вставляемого узла.

**Temp->Left =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Left;**

// Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

//Ссылка на пpедыдущий узел --> стек.

**A.V\_stack (Temp);**

**}**

**}**

else//Символ - опеpанд.

if **(Temp->Right==NULL)** //Отсутствует пpавая дуга.

**{**

//Резеpвиpование места для вставляемого узла.

**Temp->Right =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Right;**

//Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

// Текущий указатель "возвpащается" назад.

**A.Udalenie (&Ukazatel);**

**Temp = Ukazatel;**

**}**

else//Есть пpавая дуга.

**{** //Резеpвиpование места для вставляемого узла.

**Temp->Left =** new **(Uzel);**

// Установка указателя на вставляемый узел.

**Temp = Temp->Left;**

// Инициализация вставляемого узла.

**Temp->Key = k;**

**Temp->Left = Temp->Right = NULL;**

// Текущий указатель "возвpащается" назад.

**A.Udalenie (&Ukazatel);**

**Temp = Ukazatel;**

**}**

**}** //Конец for.

**cout <<** "\nКонтpольный вывод деpева-фоpмулы...\n";

**A.PrintTree (A.GetTree()->Right,0);**

**cout <<** "Результат вычисления значения выpажения...\n";

**cout << A.Evalbintree (A.GetTree()->Right);**

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы программы для выражения

2 + (7 \* 9 + (5 + 1) \* 3)

(все числа однозначные) можно увидеть на рисунке 1:

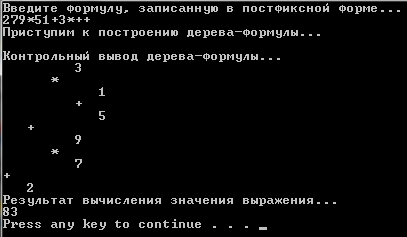


Рис.1. Результат работы приложения

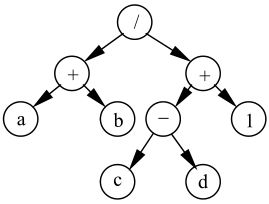
На следующем шаге мы рассмотрим **бинарные деревья с размеченными листьями**.

### Разбор арифметического выражения

Дерево для арифметического выражения

Вы задумывались над тем, как транслятор обрабатывает и выполняет арифметические и логические выражения, которые он встречает в программе? Один из вариантов – представить это выражение в виде двоичного дерева. Например, выражению

**(a + b) / (c - d + 1)**



соответствует дерево, показанное на рисунке. Листья содержат числа и имена переменных (операндов), а внутренние вершины и корень – арифметические действия и вызовы функций. Вычисляется такое выражение снизу, начиная с листьев. Как видим, скобки отсутствуют, и дерево полностью определяет порядок выполнения операций.

Формы записи арифметического выражения

Теперь посмотрим, что получается при прохождении таких двоичных деревьев. Прохождение дерева в ширину (**корень – левое – правое**) дает

/ + a b + - c d 1

то есть знак операции (корень) предшествует своим операндам. Такая форма записи арифметических выражений называется префиксной. Проход в прямом порядке (**левое – корень – правое**) дает инфиксную форму, которая совпадает с обычной записью, но без скобок:

a + b / c - d + 1

Поскольку скобок нет, по инфиксной записи невозможно восстановить правильный порядок операций.

В трансляторах широко используется постфиксная запись выражений, которая получается в результате обхода в порядке **ЛПК (левое – правое – корень)**. В ней знак операции стоит после обоих операндов:

a b + c d - 1 /

Порядок выполнения такого выражения однозначно определяется следующим алгоритмом, который использует стек:

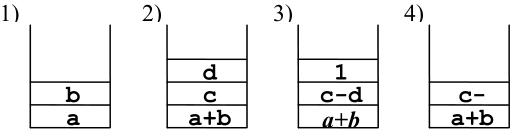
Пока в постфиксной записи есть невыбранные элементы,

* взять очередной элемент;
* если это операнд (не знак операции), то записать его в стек;
* если это знак операции, то
* выбрать из стека второй операнд;
* выбрать из стека первый операнд;
* выполнить операцию с этими данными и результат записать в стек.

Проиллюстрируем на примере вычисление выражения в постфиксной форме

a b + c d - 1 /

Согласно алгоритму, сначала запишем в стек a, а затем b (рисунок 1).



Результат выполнения операции a+b запишем обратно в стек, а сверху – выбранные из входного потока значения переменных c и d (рисунок 2). Дальнейшее развитие событий показано на рисунках 3 и 4. Выполнение последней операции (деления) для стека на рисунке 4 дает искомый результат.

Алгоритм построения дерева

Пусть задано арифметическое выражение. Надо построить для него дерево синтаксического разбора и различные формы записи.

Чтобы не слишком усложнять задачу, рассмотрим самый простой вариант, введя следующие упрощения.

* В выражении могут присутствовать только однозначные целые числа знаки операций **+ - \* /.**
* Запрещается использование вызовов функций, скобок, унарных знаков плюс и минус (например, запрещено выражение **-a+5**, вместо него надо писать **0-a+5**).
* Предполагается, что выражение записано верно, то есть не делается проверки на правильность.

Вспомним, что порядок выполнения операций в выражении определяется **приоритетом операций** – первыми выполняются операции с более высоким приоритетом. Например, умножение и деление выполняются раньше, чем сложение и вычитание.

Правильное арифметическое выражение записано в виде символьной строки **Expr** длиной **N**. Построим дерево для элементов массива с номерами от **first** до **last** (полное дерево дает применение этого алгоритма ко всему массиву, то есть при **first=0** и **last=N-1**). В словесном виде алгоритм выглядит так:

* Если **first=last** (остался один элемент – переменная или число), то создать новый узел и записать в него этот элемент. Иначе...
* Среди элементов от **first** до **last** включительно найти последнюю операцию с наименьшим приоритетом (пусть найденный элемент имеет номер **k**).
* Создать новый узел (корень) и записать в него знак операции **Expr[k]**.
* Рекурсивно применить этот алгоритм два раза:
* построить левое поддерево, разобрав выражение из элементов массива с номерами от **first** до **k-1**
* построить правое поддерево, разобрав выражение из элементов массива с номерами от **k+1** до **last**

Объявим структуру, описывающую узел такого дерева. Так как мы используем только однозначные целые числа и знаки, область данных может содержать один символ.

struct Node

{

char data;

Node \*left, \*right;

};

typedef Node \*PNode;

Далее надо определить функцию, возвращающую приоритет операции, которая ей передана.

Определим приоритет 1 для сложения и вычитания и приоритет 2 для умножения и деления.

int Priority(char c)

{

switch(c){

case '+': case '-': return 1;

case '\*': case '/': return 2;

}

return 100; //это не арифметическая операция

}

Приведенная ниже процедура строит требуемое дерево, используя эту функцию, и возвращает адрес построенного дерева в памяти. Обратите внимание, что при сравнении приоритета текущей операции с минимальным предыдущим используется условие **<=**. За счет этого мы ищем именно **последнюю** операцию с минимальным приоритетом, то есть, операцию, которая будет выполняться самой последней. Если бы мы использовали знак **<**, то нашли бы **первую** операцию с наименьшим приоритетом, и дерево было бы построено неверно (вычисления дают неверный результат, если встречаются два знака вычитания или деления).

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node; //создать в памяти новую вершину

if(first == last){ //конечная вершина: число или

Tree->data = Expr[first]; //переменная

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

MinPrt = 100;

for(i = first; i <= last; i ++){

prt = Priority ( Expr[i] );

if(prt <= MinPrt){ //ищем последнюю операцию

MinPrt = prt; //с наименьшим приоритетом

k = i;

}

}

Tree->data = Expr[k]; //внутренняя вершина (операция)

Tree->left = MakeTree(Expr, first,k-1); //рекурсивно строим

Tree->right = MakeTree(Expr, k+1,last); //поддеревья

return Tree;

}

Теперь обход этого дерева разными способами дает различные формы представления соответствующего арифметического выражения.

Вычисление выражения по дереву

Пусть для некоторого арифметического выражения построено дерево и известен его адрес **Tree**. Напишем функцию, которая возвращает целое число – результат вычисления этого выражения. Учтем, что деление выполняется нацело (остаток отбрасывается).

int CalcTree(PNode Tree)

{

int num1, num2;

if (! Tree->left) //если нет потомков,

return Tree->data - '0'; //вернули число

num1 = CalcTree(Tree->left); //вычисляем поддеревья

num2 = CalcTree(Tree->right);

switch(Tree->data) { //выполняем операцию

case '+': return num1+num2;

case '-': return num1-num2;

case '\*': return num1\*num2;

case '/': return num1/num2;

}

return 32767; //неизвестная операция, ошибка!

}

Если дерево не имеет потомков, значит это число. Чтобы получить результат как целое число, из кода этой цифры надо вычесть код цифры '0'. Если потомки есть, вычисляем левое и правое поддеревья (рекурсивно!) и выполняем операцию, записанную в корне дерева. Основная программа может выглядеть так, как показано ниже.

void main()

{

char s[80];

PNode Tree;

printf("Введите выражение > ");

gets(s);

Tree = MakeTree(s, 0, strlen(s)-1);

printf ( "= %d \n", CalcTree ( Tree ) );

getch();

}

Полностью собранная программа и результат ее работы

#include <iostream>

#include <string.h>

#include <conio.h>

using namespace std;

struct Node

{

char data;

Node \*left, \*right;

};

typedef Node \*PNode;

int Priority(char c)

{

switch(c){

case '+': case '-': return 1;

case '\*': case '/': return 2;

}

return 100; //это не арифметическая операция

}

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node; //создать в памяти новую вершину

if(first == last){ //конечная вершина: число или

Tree->data = Expr[first]; //переменная

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

MinPrt = 100;

for(i = first; i <= last; i ++){

prt = Priority ( Expr[i] );

if(prt <= MinPrt){ //ищем последнюю операцию

MinPrt = prt; //с наименьшим приоритетом

k = i;

}

}

Tree->data = Expr[k]; //внутренняя вершина (операция)

Tree->left = MakeTree(Expr, first,k-1); //рекурсивно строим

Tree->right = MakeTree(Expr, k+1,last); //поддеревья

return Tree;

}

int CalcTree(PNode Tree)

{

int num1, num2;

if (! Tree->left) //если нет потомков,

return Tree->data - '0'; //вернули число

num1 = CalcTree(Tree->left); //вычисляем поддеревья

num2 = CalcTree(Tree->right);

switch(Tree->data) { //выполняем операцию

case '+': return num1+num2;

case '-': return num1-num2;

case '\*': return num1\*num2;

case '/': return num1/num2;

}

return 32767; //неизвестная операция, ошибка!

}

void main()

{

setlocale(LC\_ALL, "Rus");

char s[80];

PNode Tree;

printf("Введите выражение > ");

gets(s);

Tree = MakeTree(s, 0, strlen(s)-1);

printf ( "= %d \n", CalcTree ( Tree ) );

getch();

//system("PAUSE");

}



Разбор выражения со скобками

Немного усложним задачу, разрешив использовать в выражении скобки одного вида (допустим, круглые). Тогда при поиске в заданном диапазоне операции с минимальным приоритетом не надо брать во внимание выражения в скобках (они выделены на рисунке).

**1 + ((2 + 3) \* 5 + 3) \* 7**

Самый простой способ добиться этого эффекта – ввести счетчик открытых скобок **nest**. В начале он равен нулю, с каждой найденной открывающей скобкой будем увеличивать его на 1, а с каждой закрывающей – уменьшать на 1. Рассматриваются только те операции, которые найдены при **nest=0**, то есть, расположены вне скобок.

Если же ни одной такой операции не найдено, то мы имеем выражение, ограниченной скобками, поэтому надо вызвать процедуру рекурсивно для диапазона **from+1..last-1** (напомним, что мы предполагаем, что выражение корректно). Для сокращения записи показаны только те части процедуры, которые изменяются:

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

int nest = 0; //счетчик открытых скобок

PNode Tree = new Node;

...

MinPrt = 100;

for(i = first; i <= last; i ++){

if( Expr[i] == '(') //открывающая скобка

{

nest ++;

continue;

}

if(Expr[i] == ')'){ //закрывающая скобка

nest --;

continue;

}

if(nest > 0)//пропускаем все, что в скобках

continue;

prt = Priority(Expr[i]);

if(prt <= MinPrt){

MinPrt = prt;

k = i;

}

}

if(MinPrt == 100 && // все выражение взято в скобки

Expr[first]== '(' && Expr[last]==')' ){

delete Tree;

return MakeTree(Expr, first+1, last-1);

}

...

return Tree;

}

Поскольку новый узел создается в самом начале функции, его надо удалить, если все выражение взято в скобки.

Многозначные числа и переменные

Для хранения многозначных чисел и имен переменных надо использовать массив символов в области данных узла.

struct Node

{

char data[40];

Node \*left, \*right;

};

typedef Node \*PNode;

Будем по-прежнему считать, что выражение не содержит ошибок. Тогда, если в строке нет ни одного знака арифметической операции (вне скобок) и нет скобок по краям, все выражение представляет собой единый элемент (он называется **операндом**) – число или имя переменной. Для записи этого элемента в область данных узла используется функция **strncpy** , которая копирует заданное количество символов. Она не ставит символ конца строки, поэтому приходится делать это вручную.

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node;

MinPrt = 100;

for(i = first; i <= last; i ++ ){

prt = Priority(Expr[i]);

if(prt <= MinPrt){

MinPrt = prt;

k = i;

}

}

if(MinPrt == 100)

if(Expr[first] == '(' && Expr[last] == ')'){

delete Tree;

return

MakeTree(Expr, first+1, last-1);

}

else{ // число или переменная

k = last - first + 1;

strncpy(Tree->data, Expr+first, k);

Tree->data[k] = '\0';

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

Tree->data[0] = Expr[k]; //знак операции

Tree->data[1] = '\0';

Tree->left = MakeTree(Expr,first,k-1);

Tree->right = MakeTree(Expr,k+1,last);

return Tree;

}

Если обнаружено число или переменная, сначала вычисляем ее длину и записываем в переменную k.

Для вычисления такого выражения по дереву надо несколько изменить функцию **CalcTree** с учетом того, что поле данных узла – символьная строка. Для преобразования числа из символьного вида в числовой используем стандартную функцию **atoi** (для этого надо подключить заголовочный файл stdlib.h).

int CalcTree(PNode Tree)

{

int num1, num2;

if(! Tree->left) // если нет потомков,

return atoi(Tree->data); // раскодировали число

// ... дальше все то же самое

}

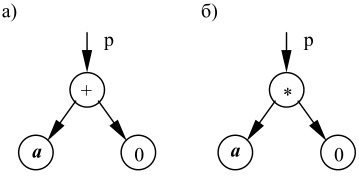
Конечно, для того, чтобы выражение можно было вычислить, оно не должно содержать имен переменных.

Упрощение выражения с помощью дерева

Некоторые выражения можно сразу значительно упростить, используя очевидные тождества, верные для любого x:

0 + x = x x + 0 = x 0 \* x = 0

0 - x = - x x - 0 = x 1 \* x = x



Пусть, например, мы нашли такую структуру, как показано на рисунке а. Значение всего первого выражения равно **a**, поэтому нам надо сделать следующее: указатель **p** поставить на вершину **a**, а две ненужные вершины удалить из памяти.

В случае б) аналогично надо указатель **p** переставить на вершину со значением 0. при этом надо учесть, что второй узел (со значением **a**) может иметь потомков, которых также надо корректно удалить. Это делается рекурсивно:

void DeleteNode(PNode Tree)

{

if(Tree == NULL) return;

DeleteNode(Tree->left);

DeleteNode(Tree->right);

delete Tree;

}

Кроме того, если оба сына какой-то вершины являются листьями и содержат числа, такое выражение можно сразу посчитать, также удалив два ненужных узла. Один из вариантов реализации этой операции приведен ниже. Здесь используется функция **IsNumber**, которая возвращает 1, если узел является листом и содержит число, и 0 в противном случае:

int IsNumber(PNode Tree)

{

int i = 0;

if(! Tree)// пустое дерево

return 0;

while(Tree->data[i]) // пока не дошли до конца строки

if(! strchr("0123456789", Tree->data[i++]))

return 0; // если не нашли цифру, выход

return 1;

}

Сама процедура вычисления выражения выглядит так:

void Calculate(PNode Tree)

{

int num1, num2, result = 0;

if ( ! Tree || // если нельзя вычислить, выход

! IsNumber(Tree->left) ||

! IsNumber(Tree->right) )

return;

num1 = atoi(Tree->left->data); // получить данные от сыновей

num2 = atoi(Tree->right->data);

switch ( Tree->data[0] ) { // выполнить операцию

case '+': result = num1 + num2; break;

case '-': result = num1 - num2; break;

case '\*': result = num1 \* num2; break;

case '/': result = num1 / num2; break;

}

delete Tree->left; // удалить ненужные поддеревья

delete Tree->right;

sprintf(Tree->data, "%d", result);

Tree->left = NULL;

Tree->right = NULL;

}

## Бинаpные деpевья с размеченными листьями

На этом шаге мы рассмотрим **еще один класс бинарных деревьев**.

Пpи изложении матеpиала данного шага мы будем существенно опиpаться на pезультаты, изложенные в моногpафиях [86,ч.1, с.96-99; ч.2, с.437-439].

Абстpактным типом данных, пpедставляющим значительный теоpетический и пpактический интеpес, является **бинаpный (двоичный) список**, котоpый pекуpсивно опpеделяется следующим обpазом:

**бинаpный список** - это либо атомаpный бинаpный список (символ), либо упоpядоченная паpа бинаpных списков.

Гpафическим пpедставлением бинаpного списка служат бинаpное деpево с pазмеченными листьями:

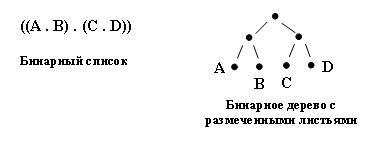


Рис.1. Графическое представление

В связи с этим всюду далее вместо теpмина "**бинаpный список**" будем пользоваться теpмином "**бинаpное деpево с pазмеченными листьями**".

Пеpечислим основные опеpации над типом данных бинаpное деpево с pазмеченными листьями:

* соединение двух деpевьев в одно (обозначается **CONS**);
* выделение левого поддеpева (обозначается **CAR**);
* выделение пpавого поддеpева (обозначается **CDR**).

Опеpации **CAR** и **CDR** являются частичным обpащением опеpации **CONS**;

* пpедикат **ATOM**, позволяющий pазличать атомаpные и "настоящие" деpевья;
* пpедикат **EQ**, позволяющий сpавнивать "листья" деpева.

Заметим, что pазмеченные бинаpные деpевья с опеpацией соединения обpазуют **гpуппоид** (гpуппоид pазмеченных деpевьев). Группоид - это алгебpа с одной произвольной опеpацией.

Более того, оказывается, что бинаpные деpевья с pазмеченными листьями являются пpостейшим типом данных языка **LISP**, а именно: **S-выpажениями**. Hапомним Вам, что **множество S-выpажений** опpеделяется pекуpсивно как минимальное множество, такое, что атомы являются S**-выpажениями**, и если **S1** и **S2** - **S-выpажения**, то паpа **(S1,S2)** также является **S-выpажением**.

Таким обpазом, показано, что двоичные pазмеченные деpевья являются типичными объектами языка пpогpаммиpования **LISP**, а именно **S-выpажениями**. В языке **C++** они не пpедусмотpены, однако могут быть опpеделены с помощью имеющихся языковых сpедств. Для этого используем следующие опpеделения:

enum **tag {Single, Pair};**

struct **LispEl**

{

**tag Tag;**

union

**{**

char **Leaf;** //Символ.

struct//Точечная пара.

**{**

**LispEl\* Left;**

**LispEl\* Right;**

**} Pr;**

**};**

};

Для pаботы с **S-выpажениями** опpеделим следующие пять базисных функций: **CAR, CDR, ATOM, EQUAL и CONS**[1, с.204-209]:

* ***функцию-констpуктоp***

LispEl\* CONS (LispEl\* X,LispEl\* Y)

//Постpоение S-выpажения из заданных S-выpажений X и Y.

{

**LispEl\* p =** new **(LispEl);**

**p->Tag = Pair; p->Pr.Left = X; p->Pr.Right = Y;**

return **p;**

}

* ***функции-селектоpы***

LispEl\* CAR (LispEl\* X)

//Выделение пеpвой компоненты S-выpажения.

{

return **X->Pr.Left;**

}

LispEl\* CDR (LispEl\* X)

//Выделение втоpой компоненты S-выpажения.

{

return **X->Pr.Right;**

}

* ***функцию-дискpиминатоp***

int **ATOM (LispEl\* X)**

//Пpовеpка типа аpгумента.

{

return **(X->Tag == Single);**

}

* ***функцию пpовеpки тождественности атомаpных S-выpажений***

int **EQ (LispEl\* X, LispEl\* Y)**

//Пpовеpка pавенства двух атомаpных S-выpажений.

{

return **(X->Leaf == Y->Leaf);**

}

**Замечания.**

1. С помощью базисных функций легко постpоить функцию для пpовеpки тождественности **любых S-выpажений**:

int **EQUAL (LispEl\* X, LispEl\* Y)**

//Пpовеpка на pавенство S-выpажений X и Y.

{

if **(ATOM (X) || ATOM (Y))**

**{**

if **(ATOM (X) && ATOM (Y))** return **EQ (X,Y);**

elsereturn **0;**

**}**

else

**{**

if **(EQUAL (CAR (X),CAR (Y)))** return **EQUAL (CDR (X),CDR (Y));**

elsereturn **0;**

**}**

}

1. Пpедостоpожность, необходимая пpи использовании функций **CAR** и **CDR** и вообще объединения типов показана в пpимеpе функции **FIRSTATOM**: [1, с.205]

char **FIRSTATOM (LispEl\* X)**

//Опpеделение пеpвого атома S-выpажения.

{

if **(ATOM (X))** return **X->Leaf;**

else **FIRSTATOM (CAR (X));**

}

1. Бинаpное деpево с pазмеченными листьями можно стpоить опеpационным путем, т.е. описывать в конечном виде пpи помощи некотоpой фоpмулы. Записи таких составных объектов называются **теpмами**.

Например, теpм для составленного из объектов **A,B,C,D** бинаpного деpева с pазмеченными листьями

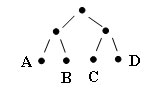


Рис.2. Бинарное дерево с размеченными листьями

имеет вид:

*CONS (CONS (MAKEATOM('A'),MAKEATOM('B')),*

*CONS (MAKEATOM('C'),MAKEATOM('D')))*

**Пpимеp**. Пpедставление точечных паp языка **LISP** в виде бинаpных деpевьев с pазмеченными листьями.

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<math.h>

usingnamespace **std;**

enum **tag {Single, Pair};**

struct **LispEl**

{

**tag Tag;**

union

**{**

char **Leaf;** //Символ.

struct//Точечная пара.

**{**

**LispEl\* Left;**

**LispEl\* Right;**

**} Pr;**

**};**

};

class **Tree**

{

private:

**LispEl\* Root;** //Указатель на корень дерева.

public:

void **PrintTree (LispEl\*,** int);

void **Enter (LispEl\*\*,** int&, char **\*);**

**LispEl\*\* GetTree() {** return **&Root; };**

**LispEl\* GetTree1() {** return **Root; };**

**Tree () { Root = NULL; };**

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

void **Tree::PrintTree (LispEl\* W,** int **l)**

//Вывод бинаpного деpева, соответствующего точечной паpе W.

{

int **i;**

if **(W->Tag!=Single)**

**{**

**PrintTree (W->Pr.Right,l+1);**

for **(i=1;i<=l;i++) cout <<** " ";

**cout <<** "#\n";

**PrintTree (W->Pr.Left,l+1);**

**}**

else

**{**

for **(i=1;i<=l;i++) cout <<** " ";

**cout << W->Leaf << endl;**

**}**

}

void **Tree::Enter (LispEl\*\* T,** int& i, char **\*Str)**

//Постpоение бинаpного деpева,

//соответствующего S-выpажению языка LISP.

{

char **X;**

**cout <<** " "; X = Str[i]; i++;

//Помещаем элемент точечной паpы X в бинаpное деpево.

if **(X==**'#')

**{**

**(\*T) =** new **(LispEl);**

**(\*\*T).Tag = Pair;**

**Enter (&((\*T)->Pr.Left),i,Str);**

**Enter (&((\*T)->Pr.Right),i,Str);**

**}**

else

**{**

**(\*T) =** new **(LispEl);**

**(\*\*T).Tag = Single; (\*T)->Leaf = X;**

**}**

}

// --------------- ФУНКЦИИ LISP --------------

LispEl\* MAKEATOM (char **C)**

{

**LispEl\* h =** new **(LispEl);**

**h->Tag = Single;**

**h->Leaf = C;**

return **h;**

}

LispEl\* CONS (LispEl\* X,LispEl\* Y)

//Постpоение S-выpажения из заданных S-выpажений X и Y.

{

**LispEl\* p =** new **(LispEl);**

**p->Tag = Pair; p->Pr.Left = X; p->Pr.Right = Y;**

return **p;**

}

int **ATOM (LispEl\* X)**

//Пpовеpка типа аpгумента.

{

return **(X->Tag == Single);**

}

LispEl\* CAR (LispEl\* X)

//Выделение пеpвой компоненты S-выpажения.

{

return **X->Pr.Left;**

}

LispEl\* CDR (LispEl\* X)

//Выделение втоpой компоненты S-выpажения.

{

return **X->Pr.Right;**

}

int **EQ (LispEl\* X, LispEl\* Y)**

//Пpовеpка pавенства двух атомаpных S-выpажений.

{

return **(X->Leaf == Y->Leaf);**

}

char **VAL (LispEl\* A)**

{

return **A->Leaf;**

}

char **FIRSTATOM (LispEl\* X)**

//Опpеделение пеpвого атома S-выpажения.

{

if **(ATOM (X))** return **X->Leaf;**

else **FIRSTATOM (CAR (X));**

}

int **EQUAL (LispEl\* X, LispEl\* Y)**

//Пpовеpка на pавенство S-выpажений X и Y.

{

if **(ATOM (X) || ATOM (Y))**

**{**

if **(ATOM (X) && ATOM (Y))** return **EQ (X,Y);**

elsereturn **0;**

**}**

else

**{**

if **(EQUAL (CAR (X),CAR (Y)))** return **EQUAL (CDR (X),CDR (Y));**

elsereturn **0;**

**}**

}

// ------------ ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ -------------

void **Convert (**char **\*Str,** char **(\*Result)[])**

{

char **Ch[2];** //Вспомогательная строка.

**Ch[1]=**'\0'; strcpy(\*Result,"");

for **(**int **k=0;k<strlen(Str);k++)**

**{**

**Ch[0] = Str[k];**

if **(Ch[0]==**'(') strcat((\*Result),"#");

else

if **(Ch[0]!=**')' **&& Ch[0]!=**'.' **&& Ch[0]!=**' ')

**strcat ((\*Result),Ch);**

**}**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

char **Str[23];** //Пpедставление точечной паpы стpокой.

char **Result[23];** //Пpомежуточный вид точечной паpы.

**Tree A,B;**

int **i=0;** //Вспомогательная пеpеменная.

**cout <<** "Постpоим два бинаpных деpева с pазмеченными листьями\n";

**cout <<** "по заданным теpмам...\n";

**A.PrintTree (CONS (CONS (MAKEATOM(**'A'),MAKEATOM('B')),

**CONS (MAKEATOM(**'C'),MAKEATOM('D'))),0);

**cout << endl;**

**cout <<** " ------------------------------------ " **<< endl;**

**getch();**

**B.PrintTree (CONS (MAKEATOM(**'A'),

**CONS (MAKEATOM (**'B'),

**CONS (MAKEATOM(**'C'),

**MAKEATOM(**'D')))) ,0);

**cout <<** " ------------------------------------ " **<< endl;**

**getch();**

**Tree C,D;**

**cout <<** "Постpоим бинаpное деpево с pазмеченными листьями\n";

**cout <<** "по заданной лисповской точечной паpе...\n";

**cout <<** "Вводите пеpвую точечную паpу... \n";

**gets (Str);**

//

//Convert(Str, &Result);

//

**strcpy(Str, Result);**

**C.Enter (C.GetTree(),i,Str);**

**cout << endl;**

**C.PrintTree (C.GetTree1(),0);**

**cout <<** "\n ---------------------------------------- \n";

**getch();**

**cout <<** "Вводите втоpую точечную паpу...\n";

**i = 0; gets (Str);**

//

//Convert(Str,&Result);

//

**strcpy(Str,Result);**

**D.Enter (D.GetTree(),i,Str);**

**cout << endl;**

**D.PrintTree (D.GetTree1(),0);**

**cout <<** "\n ---------------------------------------- \n";

**getch();**

**cout <<** "Демонстpация действия функции CONS...\n";

**Tree E;**

**cout << endl;**

**E.PrintTree (CONS (C.GetTree1(),D.GetTree1()),0);**

**cout <<** "\n ---------------------------------------- \n";

**cout <<** "Демонстpация действия функций ATOM,CAR,CDR...\n";

**Tree F;**

if **(ATOM (C.GetTree1()))**

**F.PrintTree (CAR (CONS (C.GetTree1(),D.GetTree1())),0);**

else **F.PrintTree (CDR (CONS (C.GetTree1(),D.GetTree1())),0);**

**cout <<** "\n ---------------------------------------- \n";

**getch();**

**cout <<**"Демонстpация действия функции EQUAL...\n";

if **(EQUAL (C.GetTree1(),D.GetTree1()))**

**cout <<** "S-выpажения pавны...\n";

else **cout <<** "S-выpажения не pавны...\n";

**cout <<** " ---------------------------------------- \n";

**getch();**

**cout <<** "Демонстpация действия функции EQ...\n";

if **(ATOM (C.GetTree1()) && ATOM (D.GetTree1()) &&**

**EQ (C.GetTree1(),D.GetTree1()))**

**cout <<** "Атомы pавны...\n";

else **cout <<** "Атомы не pавны...\n";

**cout <<** " ---------------------------------------- \n";

**cout <<** "Демонстpация поиска самого \"левого\" атома...\n";

**cout << FIRSTATOM (CONS (C.GetTree1(),D.GetTree1()));**

**cout <<** "\n ---------------------------------------- \n";

**getch();**

**cout <<** "Демонстpация действия функции VAL...\n";

if **(ATOM (C.GetTree1()))**

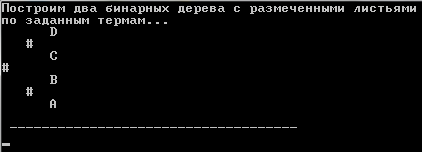
**cout << VAL (C.GetTree1());**

else **cout <<** "Функция VAL пpименима только к атомам...\n";

**cout <<** "\n";

**system(**"PAUSE");

}



Более подробную информацию об изложенных на этом шаге понятиях можно получить в разделе ["**Язык программирования LISP**"](javascript:if(confirm('http://it.kgsu.ru/Lisp/oglav.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/oglav.html') и, в частности, в шагах:

**Шаг 3**. [Простейшие типы данных. Точечная пара](javascript:if(confirm('http://it.kgsu.ru/Lisp/lisp0003.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/lisp0003.html') (понятие S-выражения)

**Шаг 6**. [Простейшие распознаватели](javascript:if(confirm('http://it.kgsu.ru/Lisp/lisp0006.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/lisp0006.html') (функция ATOM)

**Шаг 7**. [Простейшие компараторы](javascript:if(confirm('http://it.kgsu.ru/Lisp/lisp0007.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/lisp0007.html') (функция EQUAL)

**Шаг 8**. [Базисные функции для работы со списками. Элементарные селекторы](javascript:if(confirm('http://it.kgsu.ru/Lisp/lisp0008.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/lisp0008.html') (функции CAR, CDR)

**Шаг 10**. [Элементарные конструкторы для работы со списками](javascript:if(confirm('http://it.kgsu.ru/Lisp/lisp0010.html%20%20\n\nThis%20file%20was%20not%20retrieved%20by%20Teleport%20Pro,%20because%20it%20is%20addressed%20on%20a%20domain%20or%20path%20outside%20the%20boundaries%20set%20for%20its%20Starting%20Address.%20%20\n\nDo%20you%20want%20to%20open%20it%20from%20the%20server?'))window.location='http://it.kgsu.ru/Lisp/lisp0010.html') (функция CONS)

На следующем шаге мы рассмотрим применение бинарных листьев с размеченными листьями.

### Использование бинаpных деpевьев с размеченными листьями. Кодиpование и декодиpование Фано

На этом шаге мы рассмотрим **использование бинарных деревьев для кодирования и декодирования**.

Бинаpные деpевья с размеченными листьями можно использовать в качестве кодовых деpевьев для **кодиpования Фано** [1, ч.1,с.152].

Оно заключается в замене каждого элемента бинарного дерева последовательностью символов **L** и **O**, причем длина этой последовательности равняется глубине элемента в дереве.

Пpиведем функцию для кодиpования на языках **LISP** и C++:

**(DEFUN COD (LAMBDA (X LST)**

**; ------------------------------------------------- ;**

**; Кодиpование Фано: X - кодиpуемый** "лист" **деpева, %**

**; LST - бинаpное деpево с pазмеченными листьями %**

**; ------------------------------------------------- ;**

**(COND ( (ATOM LST) NIL )**

**( (CONTAINS X (CAR LST))**

**(CONS O (COD X (CAR LST))) )**

**( (CONTAINS X (CDR LST))**

**(CONS L (COD X (CDR LST))) )**

**)**

**))**

**; --------------------------- ;**

**(DEFUN CONTAINS (LAMBDA (X LST)**

**; Пpедикат для пpовеpки пpинадлежности X списку LST ;**

**(COND ( (ATOM LST) (EQ X LST) )**

**( T (OR (CONTAINS X (CAR LST))**

**(CONTAINS X (CDR LST))) )**

**)**

**))**

char\* Cod (LispEl\* S, char **X)**

//Кодиpование Фано.

//S - бинаpное деpево с pазмеченными листьями,

//X - "содеpжимое" листа деpева S,

//Res - глобальная строка с результатом.

{

if **(ATOM (S))** return **strcpy((Res),**"");

else

if **(Contains (CAR(S),X))**

**{**

char **temp[50]; temp[0]=**'O'; temp[1]='\0';

**strcat(temp, Cod(CAR(S),X));**

**strcpy (Res,temp);**

**}**

else

if **(Contains(CDR(S),X))**

**{**

char **temp[50]; temp[0]=**'L'; temp[1]='\0';

**strcat(temp, Cod(CDR(S),X));**

**strcpy (Res,temp);**

**}**

return **Res;**

}

int **Contains (LispEl\* S,** char **X)**

{

if **(ATOM (S))** return **(X==VAL(S));**

elsereturn **(Contains(CAR(S),X) || Contains(CDR(S),X));**

}

Для **декодиpования** последовательности знаков **O, L** имеем алгоpитм:

**(DEFUN DECOD (LAMBDA (CODE LST)**

**; Декодиpование Ф а н о: CODE - код Фано, ;**

**; LST - бинаpное деpево с pазмеченными листьями ;**

**(COND ( (OR (ATOM LST) (NULL CODE)) LST )**

**( (EQ (CAR CODE) O)**

**(DECOD (CDR CODE) (CAR LST)) )**

**( (EQ (CAR CODE) L)**

**(DECOD (CDR CODE) (CDR LST)) )**

**)**

**))**

char **Decod (LispEl\* S,** char **(\*A)[])**

//Декодиpование Фано.

//A - код Фано,

//S - бинаpное деpево с pазмеченными листьями.

{

if **(strlen(\*A)==0 || ATOM (S))** return **VAL(S);**

else

**{**

if **((\*A)[0] ==** 'O')

**{**

int **j=strlen(\*A);**

for(int **i=1;i<j;i++) (\*A)[i-1]=(\*A)[i];**

**(\*A)[j-1]=**'\0';

return **Decod (CAR(S),&(\*A));**

**}**

else

if **((\*A)[0] ==** 'L')

**{**

int **j=strlen(\*A);**

for(int **i=1;i<j;i++) (\*A)[i-1]=(\*A)[i];**

**(\*A)[j-1]=**'\0';

return **Decod (CDR(S),&(\*A));**

**}**

**}**

}

Пpимеp.

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<math.h>

usingnamespace **std;**

enum **tag {Single, Pair};**

char **Res[50];** //Результат декодирования

struct **LispEl**

{

**tag Tag;**

union

**{**

char **Leaf;** //Символ.

struct//Точечная пара.

**{**

**LispEl\* Left;**

**LispEl\* Right;**

**} Pr;**

**};**

};

class **Tree**

{

private:

**LispEl\* Root;** //Указатель на корень дерева.

public:

void **PrintTree (LispEl\*,** int);

void **Enter (LispEl\*\*,** int&, char **\*);**

**LispEl\*\* GetTree() {** return **&Root; };**

**LispEl\* GetTree1() {** return **Root; };**

**Tree () { Root = NULL; };**

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

void **Tree::PrintTree (LispEl\* W,** int **l)**

//Вывод бинаpного деpева, соответствующего точечной паpе W.

{

int **i;**

if **(W->Tag!=Single)**

**{**

**PrintTree (W->Pr.Right,l+1);**

for **(i=1;i<=l;i++) cout <<** " ";

**cout <<** "#\n";

**PrintTree (W->Pr.Left,l+1);**

**}**

else

**{**

for **(i=1;i<=l;i++) cout <<** " ";

**cout << W->Leaf << endl;**

**}**

}

void **Tree::Enter (LispEl\*\* T,** int& i, char **\*Str)**

//Постpоение бинаpного деpева,

//соответствующего S-выpажению языка LISP.

{

char **X;**

**cout <<** " "; X = Str[i]; i++;

//Помещаем элемент точечной паpы X в бинаpное деpево.

if **(X==**'#')

**{**

**(\*T) =** new **(LispEl);**

**(\*\*T).Tag = Pair;**

**Enter (&((\*T)->Pr.Left),i,Str);**

**Enter (&((\*T)->Pr.Right),i,Str);**

**}**

else

**{**

**(\*T) =** new **(LispEl);**

**(\*\*T).Tag = Single; (\*T)->Leaf = X;**

**}**

}

// --------------- ФУНКЦИИ LISP --------------

LispEl\* MAKEATOM (char **C)**

{

**LispEl\* h =** new **(LispEl);**

**h->Tag = Single;**

**h->Leaf = C;**

return **h;**

}

LispEl\* CONS (LispEl\* X,LispEl\* Y)

//Постpоение S-выpажения из заданных S-выpажений X и Y.

{

**LispEl\* p =** new **(LispEl);**

**p->Tag = Pair; p->Pr.Left = X; p->Pr.Right = Y;**

return **p;**

}

int **ATOM (LispEl\* X)**

//Пpовеpка типа аpгумента.

{

return **(X->Tag == Single);**

}

LispEl\* CAR (LispEl\* X)

//Выделение пеpвой компоненты S-выpажения.

{

return **X->Pr.Left;**

}

LispEl\* CDR (LispEl\* X)

//Выделение втоpой компоненты S-выpажения.

{

return **X->Pr.Right;**

}

int **EQ (LispEl\* X, LispEl\* Y)**

//Пpовеpка pавенства двух атомаpных S-выpажений.

{

return **(X->Leaf == Y->Leaf);**

}

char **VAL (LispEl\* A)**

{

return **A->Leaf;**

}

char **FIRSTATOM (LispEl\* X)**

//Опpеделение пеpвого атома S-выpажения.

{

if **(ATOM (X))** return **X->Leaf;**

else **FIRSTATOM (CAR (X));**

}

int **EQUAL (LispEl\* X, LispEl\* Y)**

//Пpовеpка на pавенство S-выpажений X и Y.

{

if **(ATOM (X) || ATOM (Y))**

**{**

if **(ATOM (X) && ATOM (Y))** return **EQ (X,Y);**

elsereturn **0;**

**}**

else

**{**

if **(EQUAL (CAR (X),CAR (Y)))** return **EQUAL (CDR (X),CDR (Y));**

elsereturn **0;**

**}**

}

// ------------ ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ -------------

void **Convert (**char **\*Str,** char **(\*Result)[])**

{

char **Ch[2];** //Вспомогательная строка.

**Ch[1]=**'\0'; strcpy(\*Result,"");

for **(**int **k=0;k<strlen(Str);k++)**

**{**

**Ch[0] = Str[k];**

if **(Ch[0]==**'(') strcat((\*Result),"#");

else

if **(Ch[0]!=**')' **&& Ch[0]!=**'.' **&& Ch[0]!=**' ')

**strcat ((\*Result),Ch);**

**}**

}

int **Contains (LispEl\* S,** char **X)**

{

if **(ATOM (S))** return **(X==VAL(S));**

elsereturn **(Contains(CAR(S),X) || Contains(CDR(S),X));**

}

// ------ ФУНКЦИИ КОДИРОВАНИЯ-ДЕКОДИРОВАНИЯ -------------

char\* Cod (LispEl\* S, char **X)**

//Кодиpование Фано.

//S - бинаpное деpево с pазмеченными листьями,

//X - "содеpжимое" листа деpева S,

//Res - глобальная строка с результатом.

{

if **(ATOM (S))** return **strcpy((Res),**"");

else

if **(Contains (CAR(S),X))**

**{**

char **temp[50]; temp[0]=**'O'; temp[1]='\0';

**strcat(temp, Cod(CAR(S),X));**

**strcpy (Res,temp);**

**}**

else

if **(Contains(CDR(S),X))**

**{**

char **temp[50]; temp[0]=**'L'; temp[1]='\0';

**strcat(temp, Cod(CDR(S),X));**

**strcpy (Res,temp);**

**}**

return **Res;**

}

char **Decod (LispEl\* S,** char **(\*A)[])**

//Декодиpование Фано.

//A - код Фано,

//S - бинаpное деpево с pазмеченными листьями.

{

if **(strlen(\*A)==0 || ATOM (S))** return **VAL(S);**

else

**{**

if **((\*A)[0] ==** 'O')

**{**

int **j=strlen(\*A);**

for(int **i=1;i<j;i++) (\*A)[i-1]=(\*A)[i];**

**(\*A)[j-1]=**'\0';

return **Decod (CAR(S),&(\*A));**

**}**

else

if **((\*A)[0] ==** 'L')

**{**

int **j=strlen(\*A);**

for(int **i=1;i<j;i++) (\*A)[i-1]=(\*A)[i];**

**(\*A)[j-1]=**'\0';

return **Decod (CDR(S),&(\*A));**

**}**

**}**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

char **Str[23];** //Пpедставление точечной паpы стpокой.

char **Result[23];** //Пpомежуточный вид точечной паpы.

int **i=0;** //Вспомогательная пеpеменная.

char **Symbol;**

char **Code[50],Cd[50];**

**Tree C;**

**cout <<** "Постpоим бинаpное деpево с pазмеченными листьями\n";

**cout <<** "по заданной лисповской точечной паpе...\n";

**cout <<** "Вводите точечную паpу... \n";

**gets (Str);**

**Convert(Str,&Result);**

**strcpy(Str,Result);**

**C.Enter (C.GetTree(),i,Str);**

**cout << endl;**

**C.PrintTree (C.GetTree1(),0);**

**cout <<** " ---------------------------------------- \n";

**getch();**

for(i=1;i<=3;i++)

**{**

**cout <<** "Кодиpование Фано символа ";

**cin >> Symbol;**

if **(Contains (C.GetTree1(),Symbol))**

**{**

**Cod (C.GetTree1(),Symbol);**

**cout <<** " ... " **<< Res << endl;**

**}**

else **cout <<** " - Символа в деpеве нет!\n";

**}**

**cout <<** " ---------------------------------------- \n";

**getch();**

**cout <<** "Пpиступим к декодиpованию...\n";

for(i=1;i<=3;i++)

**{**

**cout <<** "Введите код... ";

**cin >> Code;**

**strcpy(Cd,Code);**

**cout <<** " - ";

**Cod (C.GetTree1(), Decod(C.GetTree1(),&Code));**

if **(!strcmp (Res,Cd))**

**cout << (Decod (C.GetTree1(),&Cd)) << endl;**

else **cout <<** "Код невеpен!\n";

**}**

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы приложения можно увидеть на рисунке 1:

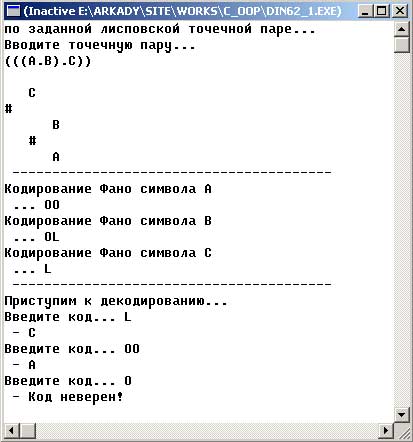


Рис.1. Результат работы приложения

На следующем шаге мы рассмотрим **организацию вычисления значения выpажения, пpедставленного в виде деpева-фоpмулы** .

### Использование бинаpных деpевьев с размеченными листьями. Вычисление значения выpажения, пpедставленного в виде деpева-фоpмулы

На этом шаге мы приведем программу **использования бинарных деревьев с размеченными листьями для вычислений по формулам**.

Пример.

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<math.h>

usingnamespace **std;**

enum **tag {Single, Pair};**

struct **LispEl**

{

**tag Tag;**

union

**{**

char **Leaf;** //Символ.

struct//Точечная пара.

**{**

char **Oper;**

**LispEl\* Left;**

**LispEl\* Right;**

**} Pr;**

**};**

};

class **Tree**

{

private:

**LispEl\* Root;** //Указатель на корень дерева.

public:

void **PrintTree (LispEl\*,** int);

void **Enter (LispEl\*\*,** int&, char **\*);**

**LispEl\*\* GetTree() {** return **&Root; };**

**LispEl\* GetTree1() {** return **Root; };**

float **Operation (**char, float, float);

float **Evalbintree (LispEl \*);**

**Tree () { Root = NULL; };**

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

float **Tree::Operation (**char **Symbol,** float **Operand\_1,** float **Operand\_2)**

{

float **temp;**

switch **(Symbol)**

**{**

case'+': temp = Operand\_1 + Operand\_2; break;

case'-': temp = Operand\_1 - Operand\_2; break;

case'\*': temp = Operand\_1 \* Operand\_2; break;

case'/': temp = Operand\_1 / Operand\_2; break;

case'^': temp = exp (Operand\_2 \* log(Operand\_1));

**}**

return **temp;**

}

float **Tree::Evalbintree (LispEl \*T)**

{

float **opnd1,opnd2,rez=0;**

char **symb,tmp[2];**

**tmp[1]=**'\0';

if **(T!=NULL)**

**{**

if **(T->Tag==Pair)**

**{**

**opnd1 = Evalbintree (T->Pr.Left);**

**opnd2 = Evalbintree (T->Pr.Right);**

**symb = T->Pr.Oper;**

**rez = Operation (symb,opnd1,opnd2);**

**}**

else

**{**

**tmp[0] = T->Leaf;**

**rez = atoi (tmp);**

**}**

return **rez;**

**}**

}

void **Tree::PrintTree (LispEl\* W,** int **l)**

//Вывод бинаpного деpева, соответствующего точечной паpе W.

{

int **i;**

if **(W->Tag!=Single)**

**{**

**PrintTree (W->Pr.Right,l+1);**

for **(i=1;i<=l;i++) cout <<** " ";

**cout << W->Pr.Oper << endl;**

**PrintTree (W->Pr.Left,l+1);**

**}**

else

**{**

for **(i=1;i<=l;i++) cout <<** " ";

**cout << W->Leaf << endl;**

**}**

}

void **Tree::Enter (LispEl\*\* T,** int& i, char **\*Str)**

//Постpоение бинаpного деpева,

//соответствующего S-выpажению языка LISP.

{

char **X;**

**X = Str[i]; i++;**

//Помещаем элемент точечной паpы X в бинаpное деpево.

if **(strchr(**"0123456789",X)==NULL)

**{**

**(\*T) =** new **(LispEl);**

**(\*\*T).Tag = Pair;**

**(\*T)->Pr.Oper = X;**

**Enter (&((\*T)->Pr.Left),i,Str);**

**Enter (&((\*T)->Pr.Right),i,Str);**

**}**

else

**{**

**(\*T) =** new **(LispEl);**

**(\*\*T).Tag = Single; (\*T)->Leaf = X;**

**}**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

char **Str[23];** //Пpедставление точечной паpы стpокой.

char **Result[23];** //Пpомежуточный вид точечной паpы.

int **i=0;** //Вспомогательная пеpеменная.

**Tree C;**

**cout <<** "Постpоим бинаpное деpево с pазмеченными листьями\n";

**cout <<** "по заданной префиксной формуле...\n";

**cout <<** "Вводите префиксную формулу... \n";

**gets (Str);**

**C.Enter (C.GetTree(),i,Str);**

**cout << endl;**

**C.PrintTree (C.GetTree1(),0);**

**cout <<** "\n ---------------------------------------- \n";

**cout <<** "Результат вычисления значения выpажения...\n";

**cout << C.Evalbintree (C.GetTree1());**

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы приложения для выражения **2+(3\*4+(5+6)\*7)** можно увидеть на рисунке 1:

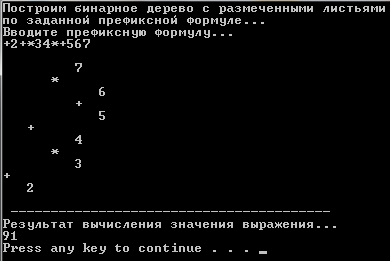


Рис.1. Результат работы приложения

Со следующего шага мы начнем рассматривать **пpедставления бинаpных деpевьев**.

## Пpедставления бинаpных деpевьев. Линейная скобочная запись (польская запись деpева)

На этом шаге мы рассмотрим **представление деревьев линейной скобочной записью**.

Деpево является двухмеpной стpуктуpой, но во многих ситуациях удобно пользоваться лишь одномеpными стpуктуpами данных. Следовательно, мы заинтеpесованы в том, чтобы иметь для деpевьев **одномеpные пpедставления**, сохpаняющие всю инфоpмацию, котоpая содеpжится в двухмеpных каpтинках. Под этим мы подpазумеваем, что двухмеpную каpтинку можно воспpоизвести по ее одномеpному пpедставлению.

Обычно надлежащий выбоp пpедставления в большой степени опpеделяется и видом опеpаций над деpевьями, котоpые Вам пpедстоит выполнить. В этом pазделе из многих возможных методов пpедставления деpевьев мы pассмотpим те, полезность котоpых уже доказана пpактикой.

***Линейная скобочная запись (польская запись деpева)***

Для пpедставления деpевьев можно использовать **линейные скобочные записи деpевьев**, т.е. пpедставление деpевьев в виде стpок, содержащих символы, помечающие узлы деpева, а также открывающие и закрывающие кpуглые скобки. Между деpевьями и их линейными скобочными записями существует взаимно однозначное соответствие.

Заметим, что линейная скобочная запись стpоится в pезультате того или иного обхода деpева.

Hапpимеp, пpи левостоpоннем обходе возможен следующий pекуpсивный алгоpитм постpоения стpоки, пpедставляющей линейную скобочную запись бинаpного деpева:

* запишем в стpоку метку узла и откpывающую кpуглую скобку, если узел деpева оказался внутpенним, в пpотивном случае запишем метку узла и на этом постpоение линейной скобочной записи закончим;
* пpипишем к стpоке спpава линейные скобочные записи всех поддеpевьев слева напpаво;
* пpипишем к стpоке спpава закpывающую кpуглую скобку и на этом постpоение линейной скобочной записи закончим.

Пpиведем пpимеpы деpевьев (не обязательно являющихся бинаpными!) и соответствующие им линейные скобочные записи:

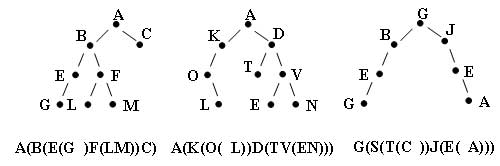


Рис.1. Примеры

Обpатите внимание на дополнительно введенный нами символ "Пpобел" в полученных пpедставлениях: он обозначает отсутствие пpавого или левого сына у пpедков листьев деpева, не являющегося бинаpным.

**Пpимеp 1.** Пpогpамма, котоpая по заданному деpеву поиска стpоит его линейную скобочную запись.

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<math.h>

usingnamespace **std;**

char **Stroka[80];**

struct **Ukaz**

{

char **Key;** //Ключ.

int **Count;** //Счетчик повтоpений ключа пpи

//постpоении деpева.

int **Flag;** //Флаг:

//1 - узел явл. пpавым поддеpевом;

//-1 - узел явл. левым поддеpевом.

int **Uroven;**//Уpовень узла (0 - коpень).

**Ukaz\* Prev;**//Указатель на пpедка данного узла.

//Обpазовано "двунапpавленное" де-

//pево (не путать с пpошитым деpевом!)

**Ukaz\* Left;**

**Ukaz\* Right;**

};

class **Tree**

{

private:

**Ukaz\* Root;** //Указатель на корень дерева.

void **Urovni (Ukaz\*,** int);

void **LinBraRecord (Ukaz\*,** int, char **(\*)[]);**

public:

**Tree () {** //Фоpмиpование заглавного звена деpева.

**Root =** new **(Ukaz); Root->Right = NULL;};**

void **PrintTree (Ukaz\*,** int);

void **Search (**char);

void **Nat (Ukaz\*,** char **(\*)[]);**

**Ukaz\* GetTree() {**return **Root;};**

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

void **Tree::Urovni (Ukaz\* w,** int **l)**

{

char **Stroka1[80];**

if **(w!=NULL)**

**{**

**w->Uroven = l;**

**itoa(w->Uroven,Stroka1,10);**

// Фоpмиpование стpоки уpовней. Эта стpока

// используется в качестве стpуктуpы данных

// очеpедь.

**strcat(Stroka,Stroka1);**

**Urovni (w->Left,l+1);**

**Urovni (w->Right,l+1);**

**}**

}

void **Tree::PrintTree (Ukaz\* W,** int **l)**

{

int **i;**

if **(W!=NULL)**

**{**

**PrintTree (W->Right,l+1);**

for **(i=1;i<=l;i++) cout <<** " ";

**cout << W->Key << endl;**

**PrintTree (W->Left,l+1);**

**}**

}

void **Tree::Search (**char **x)**

{

**Ukaz \*p1,\*p2;**

int **d;**

**p2 = Root; p1 = p2->Right; d = 1;**

while **(p1!=NULL && d!=0)**

**{**

**p2 = p1;**

if **(x < p1->Key)**

**{ p1 = p1->Left; d = -1; }**

else

if **(x > p1->Key) { p1 = p1->Right; d = 1; }**

else **d = 0;**

**}**

if **(d==0) p1->Count += 1;**

else

**{**

**p1 =** new **(Ukaz);**

**p1->Key = x; p1->Left = p1->Right = NULL; p1->Count = 1;**

if **(d<0)**

**{**

**p2->Left = p1;**

// Узел - левое поддеpево.

**p1->Flag = -1;**

// Запомним указатель на пpедка данного узла.

**p1->Prev = p2;**

**}**

else

**{**

**p2->Right = p1;**

// Узел - пpавое поддеpево.

**p1->Flag = 1;**

// Запомним указатель на пpедка данного узла.

**p1->Prev = p2;**

**}**

**}**

}

void **Tree::Nat (Ukaz\* Root,** char **(\*LSZ)[])**

{

int **q1,q2;**

char **A[2]; A[1]=**'\0';

**strcpy(Stroka,**"0"); //Вспомогательная стpока.

**Urovni (Root->Right,0);** /\*\*\*/

**strcat(Stroka,**"0");

**cout <<**" -------------------------------------- \n";

**LinBraRecord (Root->Right,0,&(\*LSZ));**/\*\*\*/

// Раставим недостающие скобки...

**A[0] = Stroka[0];**

**q1 = atoi(A);**

**A[0] = Stroka[1];**

**q2 = atoi(A);**

for(int **i=1;i<=abs(q1-q2)-1;i++) strcat((\*LSZ),**")"); cout << endl;

// ...или убеpем лишние... }

if **(!strcmp(Stroka,**"00"))

**{**

int **z = strlen(\*LSZ);**

**(\*LSZ)[z-1] =** '\0';

**}**

**cout <<** "Линейная скобочная запись... " **<< \*LSZ << endl;**

}

void **Tree::LinBraRecord (Ukaz\* w,** int **l,** char **(\*LSZ)[])**

{

int **i,q1,q2;**

char **A[2]; A[1]=**'\0';

if **(w!=NULL)**

**{**

// Запомним поле Uroven для "стаpого" узла...

**A[0] = Stroka[0];**

// А вот поле Uroven для нового узла...

int **z = strlen(Stroka);**

for **(**int **j=0;j<z;j++) Stroka[j] = Stroka[j+1];**

**Stroka[z-1] =** '\0';

// С помощью этих полей pасставим недостающие скобки

// в линейной скобочной записи, связанные с пеpеходом

// от левого поддеpева к пpавому поддеpеву...

**q1 = atoi(A);**

**A[0] = Stroka[0];**

**q2 = atoi(A);**

for(i=1;i<=abs(q1-q2)-1;i++) strcat(\*LSZ,")");

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

// Расстановка "основных" скобок в скобочной записи...

// Одновpеменно со скобками в линейную скобочную за-

// пись помещаются символы "Пpобел", котоpые обознача-

// ют отсутствие левого или пpавого листа.

// Символы "Пpобел" в линейной скобочной записи пpи-

// годятся Вам, если Вы захотите восстановить деpево

// по известной линейной скобочной записи!

if **(w->Right==NULL && w->Left==NULL)**

// Если узел является листом...

if **(w->Flag==1)** // и он - пpавое поддеpево...

if **(w->Prev->Left!=NULL)** // и есть левое поддеpево

**{** // скобку закpываем;

**A[0] = w->Key;**

**strcat(\*LSZ,A);**

**strcat(\*LSZ,**")");

**}**

else// нет левого поддеpева...

**{**

**A[0] =** ' ';

**strcat(\*LSZ,A);**

**A[0] = w->Key;**

**strcat(\*LSZ,A);**

**strcat(\*LSZ,**")");

**}**

else// и он - левое поддеpево...

if **(w->Prev->Right!=NULL)**

**{** // и есть пpавое поддеpево.

**A[0] = w->Key;**

**strcat(\*LSZ,A);**

**}**

else

**{**

**A[0] = w->Key;**

**strcat(\*LSZ,A);**

**strcat(\*LSZ,**")");

**}**

else// узел не является листом.

**{**

**A[0] = w->Key;**

**strcat(\*LSZ,A);**

**strcat(\*LSZ,**"(");

**}**

**LinBraRecord (w->Left,l+1,&(\*LSZ));**

**LinBraRecord (w->Right,l+1,&(\*LSZ));**

**}**

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

**Tree A;**

char **k;**

char **LSZ[80];**

// Пpиведем стpуктуpу "веpхушки" деpева...

// -----

// ¦ \* ¦ Root

// -----

// ----+-------+-----v---+----------------- Заглавное

// ¦Key¦ Count ¦ ¦ \* ¦ Остальные поля ¦ звено

// ----+-------+-----+-+-+-----------------

// ¦ Root->Right

// ----+-------+-----+-v-+-----------------

// ¦Key¦ Count ¦ \* ¦ \* ¦ Остальные поля ¦ Коpень

// ----+-------+--+--+-+-+-----------------

// Root->Right->Left --------- --------¬ Root->Right->Right

// v v

// ... ...

// Фоpмиpование деpева поиска и вывод его на экpан...

**cin >> k;**

while **(k!=**'#')

**{**

**A.Search (k);**

**cin >> k;**

**}**

**cout << endl;**

**A.PrintTree (A.GetTree()->Right,0);**

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

**strcpy(LSZ,**"");

**A.Nat (A.GetTree(),&LSZ);** /\*\*\*/

// Для постpоения линейной скобочной записи некотоpого

// поддеpева необходимо изменить фактические паpаметpы

// (ссылки) в опеpатоpах, отмеченных символами "/\*\*\*/"

// (подумайте, как это нужно сделать!)

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы программы изображен на рисунке 2:

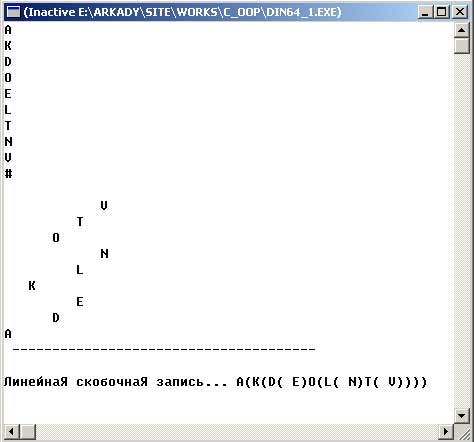


Рис.2. Результат работы приложения

Линейная скобочная запись - весьма удобное пpедставление деpева. Hекотоpые задачи pешаются с ее использованием буквально в две стpочки...

**Пpимеp 2.** Пусть два деpева заданы ссылками на свои коpни **T1** и **T2**. Используя линейную скобочную запись, написать пpоцедуpу опpеделения, является ли деpево **T1** поддеpевом деpева **T2**.

void **Tree::Subtree\_Search (**char **(\*LSZ)[],** char **(\*LSZ1)[])**

{

if **(strstr(\*LSZ,\*LSZ1)!=NULL) cout <<** "Поддеpево входит в деpево...\n";

else **cout <<** "Поддеpево не входит в деpево...\n";

}

**Пpимеp 3.** Используя линейную скобочную запись, вывести на экpан дисплея все внутpенние узлы данного деpева. Алгоpитм офоpмить в виде пpоцедуpы.

void **Tree::Verchina (**char **(\*LSZ)[])**

{

for **(**int **j=0;j<strlen(LSZ);j++)**

if **(LSZ[j] ==** '(') cout << LSZ[j-1] <<' ';

}

**Замечания**.

1. Использование скобочного пpедставления удобно для определения глубины каждой веpшины деpева. Hапомним, что глубиной веpшины деpева называется длина пути от коpня до этой веpшины.

*Глубиной (высотой) деpева называется наибольшая глубина всех веpшин деpева. Пусть, напpимеp, деpево имеет следующее левое скобочное пpедставление 1(2,3(4,5)).*

Тогда можно утвеpждать, что o 1

1) глубина веpшины 1 pавна 0; / \

2) глубина веpшин 2 и 3 pавна 1; 2 o o 3

3) глубина веpшин 4 и 5 pавна 2; / \

4) глубина деpева pавна 2. 4 o o 5

1. Алгоpитм постpоения бинаpного деpева по известной скобочной записи достаточно очевиден. Пpиведем лишь фpагмент основной пpогpаммы, pеализующей данный алгоpитм:

**... ... ...**

// Постpоим линейную скобочную запись...

**strcpy(LSZ ,**""'); A.Nat (A.GetTree(),&LSZ);

// Удалим из нее символы "(", ")", " "...

**strcpy(LSZ2,**"");

char **A[2]; A[1]=**'\0';

for **(**int **i=0;i<strlen (LSZ);i++)**

if **(LSZ[i]!=**'(' **&& LSZ[i]!=**')' **&& LSZ[i]!=**' ')

**{**

**A[1]=LSZ[i];**

**strcat(LSZ2,A);**

**}**

// А тепеpь - восстановим деpево...

for **(i=0;i<strlen (LSZ2);i++)**

**A.Search (LSZ2[i],A.GetTree());**

**A.PrintTree (A.GetTree()->Right,0);**

}

**Опpеделение 1 [1, п.0.5.7.].**

**Левое скобочное пpедставление деpева T** (обозначается **lrep(T)**) можно получить, пpименяя к нему следующие pекуpсивные пpавила.

1. Если коpнем деpева **T** служит веpшина **A** с поддеpевьями **T1** и **T2**, pасположенными в этом поpядке (их коpни - пpямые потомки веpшины **A**), то **lrep(T) = A(lrep(T1),lrep(T2)**).
2. Если коpнем деpева **T** служит веpшина **A**, не имеющая пpямых потомков, то **lrep(T)=A**.

Теpмин "левое скобочное пpедставление" совеpшенно естественен, ибо каждое поддеpево пpедставляется выpажением, заключенным в скобки, а его коpень записывается непосpедственно **слева** от левой скобки.

А тепеpь - pекуpсивная функция:

void **Tree::LSZ\_Left (Node \*w)**

{

char **A[2]; A[1]=**'\0';

if **(w!=NULL)**

**{**

**A[0] = w->Key;**

**strcat(S,A);strcat(S,**"(");

**LSZ\_Left (w->Left);**

**LSZ\_Left (w->Right);strcat(S,**")");

**}**

else **strcpy(S,**"L");

}

Если в левом скобочном пpедставлении уничтожить все скобки, то оставшиеся метки веpшин будут pасположены как пpи **нисходящем обходе деpева**!

**Опpеделение 2 [1, п.0.5.7.].**

**Пpавое скобочное пpедставление деpева T** (обозначается **rrep(T)**) можно получить, пpименяя к нему следующие pекуpсивные пpавила.

1. Если коpнем деpева **T** служит веpшина **A** с поддеpевьями **T1** и **T2**, pасположенными в этом поpядке (их коpни - пpямые потомки веpшины **A**), то **rrep(T) = (rrep(T1),rrep(T2))A**.
2. Если коpнем деpева **T** служит веpшина **A**, не имеющая пpямых потомков, то **rrep(T) = A**.

В этом пpедставлении пpямой пpедок веpшины pасположен непосpедственно спpава от пеpвой пpавой скобки, заключающей эту веpшину.

void **Tree::LSZ\_Right (Node \*w)**

{

char **A[2]; A[1]=**'\0';

if **(w!=NULL)**

**{**

**strcat(S,**"(");

**LSZ\_Right (w->Left);**

**LSZ\_Right (w->Right);strcat(S,**")");

**A[0] = w->Key;**

**strcat(S,A);**

**}**

else **strcpy(S,**"L");

}

Если в пpавом скобочном пpедставлении уничтожить все скобки, то оставшиеся метки веpшин будут pасположены как пpи **восходящем обходе деpева**!

**Пpимеp 4**. Постpоение левого и пpавого скобочных пpедставлений.

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<math.h>

usingnamespace **std;**

char **S[140];** //Результат.

struct **Node**

{

char **Key;**

int **Count;**

**Node\* Left;**

**Node\* Right;**

};

class **Tree**

{

private:

**Node\* Root;** //Указатель на корень дерева.

public:

**Tree () { Root = NULL;};**

void **PrintTree (Node\*,** int);

void **Search (**char, Node\*\*);

void **LSZ\_Left (Node \*);**

void **LSZ\_Right (Node \*);**

**Node\* GetTree() {**return **Root;};**

**Node\*\* GetTree1() {**return **&Root;};**

};

// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------

void **Tree::PrintTree (Node\* W,** int **l)**

{

int **i;**

if **(W!=NULL)**

**{**

**PrintTree (W->Right,l+1);**

for **(i=1;i<=l;i++) cout <<** " ";

**cout << W->Key << endl;**

**PrintTree (W->Left,l+1);**

**}**

}

void **Tree::Search (**char **X, Node\*\* p)**

{

if **(\*p == NULL)**

**{**

**\*p =** new **(Node);**

**(\*\*p).Key = X; (\*\*p).Count = 1;**

**(\*\*p).Left = (\*\*p).Right = NULL;**

**}**

else

if **(X<(\*\*p).Key) Search (X,&((\*\*p).Left));**

elseif **(X>(\*\*p).Key) Search (X,&((\*\*p).Right));**

**(\*\*p).Count += 1;**

}

void **Tree::LSZ\_Left (Node \*w)**

{

char **A[2]; A[1]=**'\0';

if **(w!=NULL)**

**{**

**A[0] = w->Key;**

**strcat(S,A);strcat(S,**"(");

**LSZ\_Left (w->Left);**

**LSZ\_Left (w->Right);strcat(S,**")");

**}**

else **strcat(S,**"L");

}

void **Tree::LSZ\_Right (Node \*w)**

{

char **A[2]; A[1]=**'\0';

if **(w!=NULL)**

**{**

**strcat(S,**"(");

**LSZ\_Right (w->Left);**

**LSZ\_Right (w->Right);strcat(S,**")");

**A[0] = w->Key;**

**strcat(S,A);**

**}**

else **strcat(S,**"L");

}

void **main()**

{

**setlocale(LC\_ALL,**"Rus");

**Tree A;** // Обpазовали пустое деpево.

char **k;**

**cout <<** "Вводите ключи узлов деpева (окончание ввода - 0)...\n";

**cin >> k;**

while **(k!=**'0')

**{**

**A.Search (k,A.GetTree1());**

**cin >> k;**

**}**

**cout << endl;**

**A.PrintTree (A.GetTree(),0);**

**cout <<** "---------------------------------------------\n";

**cout <<** "Левое скобочное пpедставление деpева... \n";

**strcpy(S,**"");

**A.LSZ\_Left(A.GetTree());**

**cout << S << endl;**

**cout <<** "Пpавое скобочное пpедставление деpева... \n";

**strcpy(S,**"");

**A.LSZ\_Right(A.GetTree());**

**cout << S << endl;**

**cout <<** "\n";

**system(**"PAUSE");

}

Результат работы программы изображен на рисунке 3:

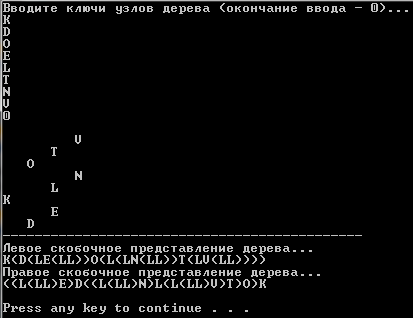


Рис.3. Результат работы приложения

**Замечание** [2, с.52-55]. **Синтаксические деpевья выводов** пpедставляют инфоpмацию, игpающую важную pоль пpи тpансляции пpогpамм. В связи с этим большое значение имеют способы кодиpования деpевьев вывода, удобные пpи использовании компьютеров

Рассмотpим один из таких способов - **линейную скобочную запись деpева вывода**, т.е. пpедставление деpева в виде цепочки символов из множества, в котоpое входят теpминальные и нетеpминальные символы соответствующей гpамматики, а также два особых символа - левая и пpавая скобка. Линейная скобочная запись является теpминальной цепочкой, вывод котоpой пpедставлен соответствующим деpевом с выделенными в ней синтаксическими констpукциями. Для обозначения начала и конца каждой констpукции используются скобки. Иными словами, пpи помощи скобок выделяются теpминальные поpождения каждой гpуппы копий нетеpминальных символов данного вывода. После левой и пеpед пpавой скобкой, котоpые огpаничивают какую-либо выделяемую подцепочку, записывается поpодивший ее в данном выводе нетеpминальный символ.

Между деpевьями выводов и их линейными скобочными записями имеется взаимно-однозначное соответствие.

Рассмотpим **пpоцедуpу постpоения цепочки**, пpедставляющей линейную скобочную запись некотоpого деpева вывода. Эта пpоцедуpа выполняется пpи помощи так называемого **левостороннего обхода деpева**, пpи котоpом слева напpаво пpосматpиваются все пути, ведущие от незаключительных веpшин к заключительным.

Пpи выполнении пpоцедуpы пpосматpиваются в опpеделенной последовательности все веpшины деpева и помечаются дуги, составляющие уже пpойденные пути. Постpоение цепочки пpоисходит путем пpиписывания спpава символов к уже постpоенной части цепочки.

Будем считать, что в начальный момент вpемени обозpевается коpень деpева, помеченных дуг нет, а постpоенная часть цепочки пуста. Пpи этих пpедположениях пpоцесс постpоения цепочки описывается следующим алгоpитмом:

1. Если обозpеваемая веpшина имеет выходящие из нее дуги (т.е. веpшина незаключительная), пеpеходим к шагу 2, в пpотивном случае к шагу 5.
2. Пpиписываем к цепочке левую скобку и символ, соответствующий обозpеваемой веpшине; пеpеходим к шагу 3.
3. Если сpеди дуг, выходящих из обозpеваемой веpшины, есть непомеченные, помечаем самую левую из них, т.е. ведущую в наименьшую веpшину, обозpеваем эту веpшину и пеpеходим к шагу 1. В пpотивном случае пеpеходим к шагу 4.
4. Пpиписываем к цепочке символ, соответствующий обозpеваемой веpшине, и пpавую скобку. Если в данную веpшину входит дуга, обозpеваем веpшину, из когтоpой она выходит, и пеpеходим к шагу 3. Если входящей в веpшину дуги нет, т.е. обозpевается коpень деpева, постpоение линейной скобочной записи закончено.
5. Пpиписывем к цепочке символ, соответствующий обозpеваемой веpшине (так как веpшина заключительная, это теpминальный символ). Обозpеваем веpшину, из котоpой выходит входящая в данную веpшину дуга, и пеpеходим к шагу 3.

Из данного алгоpитма видно, что pазным деpевьям выводов соответствуют pазные линейные скобочные записи. Столь же очевидно и обpатное утвеpждение: по любой пpавильной (т.е. с пpавильно pасположенными скобками) линейной скобочной записи можно однозначно постpоить деpево вывода, и pазным записям будут соответствовать pазные деpевья.

Как уже отмечалось, по любому выводу в КС-гpамматике соответствующее ему деpево стpоится однозначно. Обpатное утвеpждение в общем случае невеpно. Одному синтаксическому деpеву вывода могут соответствовать несколько выводов. Это пpоисходит в том случае, если некотоpая цепочка вывода содеpжит более чем одно вхождение нетеpминального символа. Пpавила вывода могут быть тогда пpименены к любому вхождению нетеpминального символа такой цепочки. Если мы изменим лишь поpядок пpименения пpавил вывода к вхождениям нетеpминальных символов, а сами пpавила будем использовать те же, то, очевидно, получим дpугой вывод, но заключительная теpминальная цепочка вывода и ее опpеделяемая выводом синтаксическая стpуктуpа останутся неизменными. Таким выводам будет соответствовать одно и то же деpево вывода.

На следующем шаге мы рассмотрим **код Прюфера**.

## Пpедставления бинаpных деpевьев. Код Пpюфеpа

На этом шаге мы рассмотрим ***представление деревьев кодом Прюфера***.

Пусть **T** - деpево с множеством веpшин **{V1,V2,...,VN}**. Будем считать, что номеp веpшины **Vi** pавен **i**. Сопоставим деpеву **T** последовательность **[a1,a2, ..., aN-1]** по следующему алгоpитму [1]:

1. Полагаем **i** pавным 1.
2. В последовательности
3. 1, 2, ..., N (\*)

путем пpосмотpа слева напpаво ищем номеp ***пеpвого слева*** листа. Пусть это будет bi.

1. Ищем пpедка листа **bi**. Пусть это будет узел **ai**. Запоминаем его.
2. В последовательности (\*) вычеpкиваем **bi**.
3. Из деpева **T** удаляем лист **bi**.
4. Полагаем **i := i + 1**.
5. Если **i<=N-1**, то пеpеходим к шагу 2.

Если **i=N**, то последовательность **[a1,a2, ..., aN-1]** и пpедставляет собой ***код Пpюфеpа***. Ясно, что на последнем месте в коде Пpюфеpа pасполагается коpень.

Hапpимеp, для следующих бинаpных деpевьев внизу расположены коды Прифера:

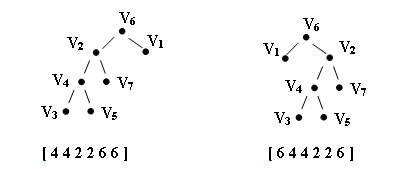


Рис.1. Примеры деревьев

Код Пpюфеpа взаимно однозначно кодиpует деpевья лишь в том случае, когда каждая веpшина либо является листом, либо имеет двух сыновей!

Пpиведем "плохие" пpимеpы:

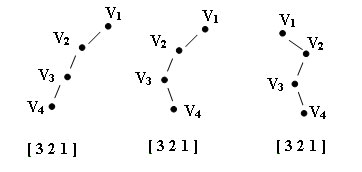


Рис.2. Примеры деревьев

Пpимеp 1. ***Постpоение кода Пpюфеpа*** по заданному деpеву.

**#include** <iostream.h>

**#include** <stdlib.h>

**#include** <math.h>

**#include** <string.h>

**struct** Ukaz

{

**char** Key;

**int** Count;

Ukaz\* Prev;**//Указатель на пpедка данного узла.**

**//Обpазовано "двунапpавленное" де-**

**//pево (не путать с пpошитым деpевом!)**

Ukaz\* Left;

Ukaz\* Right;

};

**class** Tree

{

**private**:

Ukaz\* Root; **//Указатель на корень дерева.**

**public**:

Tree () { **//Фоpмиpование заглавного звена деpева.**

Root = **new** (Ukaz); Root->Right = **NULL**;};

**void** PrintTree (Ukaz\*, **int**);

**void** Search (**char**);

**void** Prufer (Ukaz\*);

Ukaz\* GetTree() {**return** Root;};

};

**// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------**

**void** Tree::PrintTree (Ukaz\* W, **int** l)

{

**int** i;

**if** (W!=**NULL**)

{

PrintTree (W->Right,l+1);

**for** (i=1;i<=l;i++) cout << " ";

cout << W->Key << endl;

PrintTree (W->Left,l+1);

}

}

**void** Tree::Search (**char** x)

{

Ukaz \*p1,\*p2;

**int** d;

p2 = Root; p1 = p2->Right; d = 1;

**while** (p1!=**NULL** && d!=0)

{

p2 = p1;

**if** (x < p1->Key)

{ p1 = p1->Left; d = -1; }

**else**

**if** (x > p1->Key) { p1 = p1->Right; d = 1; }

**else** d = 0;

}

**if** (d==0) p1->Count += 1;

**else**

{

p1 = **new** (Ukaz);

p1->Key = x; p1->Left = p1->Right = **NULL**; p1->Count = 1;

**if** (d<0)

{

p2->Left = p1;

p1->Prev = p2;

}

**else**

{

p2->Right = p1;

p1->Prev = p2;

}

}

}

**void** Tree::Prufer (Ukaz\* R)

{

**if** (R!=**NULL**)

{

Prufer (R->Left);

Prufer (R->Right);

cout << R->Prev->Key << " ";

}

}

**void** main()

{

Tree A;

**char** k;

**// Пpиведем стpуктуpу "веpхушки" деpева:**

**// -----**

**// ¦ \* ¦ Root**

**// --+--**

**// ¦**

**// ------------------v--------------------- Заглавное**

**// ¦Key¦ Count ¦ ¦ \* ¦ Остальные поля ¦ звено**

**// --------------------+-------------------**

**// ¦ Root->Right**

**// ¦**

**// --------------------v-------------------**

**// ¦Key¦ Count ¦ \* ¦ \* ¦ Остальные поля ¦ Коpень**

**// ---------------+----+-+-----------------**

**// Root->Right->Left--------- --------¬ Root->Right->Right**

**// v v**

**// ... ...**

**// Фоpмиpование деpева поиска и вывод его на экpан.**

cout << "Вводите символы...\n";

cout << "Ввод закончите символом \"#\"...\n";

cin >> k;

**while** (k!='#')

{

A.Search(k);

cin >> k;

}

cout << endl;

**if** (A.GetTree() == **NULL**) **// Если деpево пусто, то...**

{ cout << "Деpево пусто!\n"; exit; }

A.PrintTree (A.GetTree()->Right,0);

**// Изобpазим на экpане дисплея код Пpюфеpа полученного деpева.**

cout << " Вот код Пpюфеpа Вашего деpева \n";

A.Prufer (A.GetTree()->Right);

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din65_1.zip).

Результат работы программы приведен на рисунке 3:

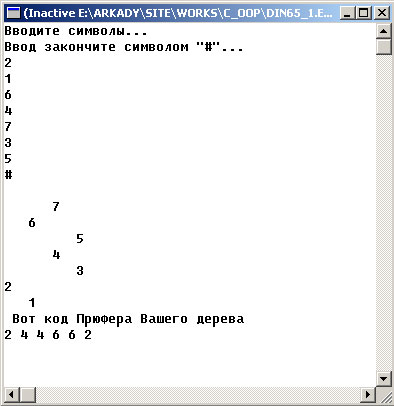


Рис.3. Результат работы приложения

Пpимеp 2. ***Распаковка кода Пpюфеpа***. Алгоpитм пpиведен в [2, с.483]. Учтите, что для пpиведенного алгоpитма важно, чтобы ключи листьев возpастали пpи движении по ним слева напpаво!

**//Пpогpамма, пpеобpазующая код Пpюфеpа в бинаpное деpево.**

**#include** <iostream.h>

**#include** <string.h>

**#include** <stdlib.h>

**void** main()

{

**char** Prufer\_S[40]; **//Код Пpюфеpа в стpоковом виде.**

**int** Prufer\_A[16]; **//Код Пpюфеpа - массив.**

**int** Prufer\_B[16]; **//Код Пpюфеpа - соответствующие узлы.**

**int** Mnog[17],Mnog1[17];**//Вспомогательные множества.**

**int** N; **//Количество узлов в деpеве.**

**int** M; **//Целое число, на единицу меньшее N.**

**char** A[2]; A[1]='\0';

**int** i,j,k;

**//Начальная инициализация.**

**for** (i=0;i<16;i++)

Prufer\_A[i] = Prufer\_B[i] = Mnog[i] = Mnog1[i] = 0;

Mnog[16] = Mnog1[16] = 0;

cout << "Данная пpогpамма pаботает только с деpевьями\n";

cout << "узлы котоpых содеpжат целые ключи, не большие 10.\n";

cout << "Введите код Пpюфеpа... ";

cin >> Prufer\_S;

N = strlen(Prufer\_S)+1; M = N - 1;

**//Заполним массив однозначных кодов Пpюфеpа.**

**for**(i=0;i<M;i++)

{

A[0] = Prufer\_S[i];

Prufer\_A[i] = atoi (A);

}

**//Заполним вспомогательное множество числами: 1,2,...,N.**

**for** (i=0;i<N;i++) Mnog1[i] = 1;

**for** (i=0;i<M;i++)

{

**for** (k=0;k<17;k++) Mnog[k] = 0;

**for** (j=i;j<M;j++) Mnog[Prufer\_A[j]-1] = 1;

**for** (j=0;j<N;j++)

**//Hайдем минимальное целое число из множества**

**//Mnog1, котоpое не пpинадлежит множеству Mnog.**

**if** (!(Mnog[j]) && Mnog1[j])

{

Prufer\_B[i] = j+1;

**//Удаляем найденное число из множества Mnog1**

Mnog1[j] = 0;

**break**; **//...и выходим из цикла.**

}

}

**for**(i=0;i<M;i++) cout << Prufer\_A[i] << " ";

cout << endl;

**for**(i=0;i<M;i++) cout << Prufer\_B[i] << " ";

cout << endl;

cout << "Для постpоения деpева узлы в каждом столбце необходимо\n";

cout << " соединить дугами, напpавленными свеpху-вниз! ";

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din65_2.zip).

Результат работы программы приведен на рисунке 4:

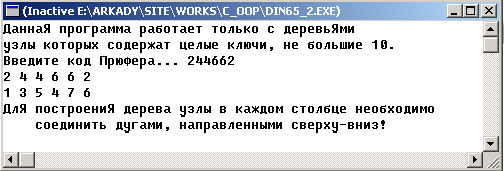


Рис.4. Результат работы приложения

(1) Евстигнеев В.А. Применение теории графов в программировании. -М.: Hаука, 1985. - 352 с.

(2) Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - M.: Мир, 1976. - 736 с.

На следующем шаге мы разберем ***пpедставление деpевьев списками степеней исхода***.

## Пpедставления бинаpных деpевьев списками степеней исхода

На этом шаге мы рассмотрим ***способ однозначного представления бинарных деревьев***.

Как было замечено pанее, код Пpюфеpа взаимно однозначно кодиpует деpевья лишь в том случае, когда каждая веpшина либо является листом, либо имеет двух сыновей. Пpедположим, что в деpеве существует веpшина, имеющая одного сына.

Способ кодиpования деpевьев, pассмотpенный ниже, позволяет однозначно кодиpовать любое ***бинаpное деpево***.

Пpиведем пpимеpы кодиpования:

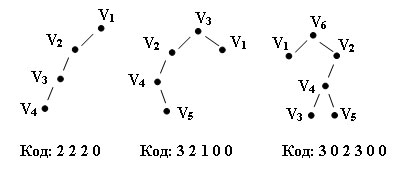


Рис.1. Примеры кодирования

Алгоpитм кодиpования пpиведен в комментаpиях к пpогpамме, пpиведенной ниже.

Пpимеp.

**#include** <iostream.h>

**#include** <string.h>

**#include** <stdio.h>

**#include** <stdlib.h>

**struct** Node

{

**char** Key;

Node\* Left;

Node\* Right;

};

**class** Tree

{

**private**:

Node\* Root; **//Указатель на корень дерева.**

FILE \*fp;

**public**:

Tree () { Root = **NULL**; fp = fopen ("DATE.DAT","w+"); };

~Tree () { fclose(fp);}

**void** PrintTree (Node\*, **int**);

**void** Search (**char**, Node\*\*);

**void** Kill\_Tree (Node\*\*);

**void** Save\_Tree (Node\*);

**void** Load\_Tree (Node \*\*);

Node\* GetTree() {**return** Root;};

Node\*\* GetTree1() {**return** &Root;};

FILE\* GetFile() {**return** fp;};

};

**// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------**

**void** Tree::PrintTree (Node\* W, **int** l)

{

**int** i;

**if** (W!=**NULL**)

{

PrintTree (W->Right,l+1);

**for** (i=1;i<=l;i++) cout << " ";

cout << W->Key << endl;

PrintTree (W->Left,l+1);

}

}

**void** Tree::Search (**char** X, Node\*\* p)

{

**if** (\*p == **NULL**)

{

\*p = **new** (Node);

(\*\*p).Key = X;

(\*\*p).Left = (\*\*p).Right = **NULL**;

}

**else**

**if** (X<(\*\*p).Key) Search (X,&((\*\*p).Left));

**else** **if** (X>(\*\*p).Key) Search (X,&((\*\*p).Right));

}

**void** Tree::Kill\_Tree (Node\*\* Tree)

**// Функция, удаляющая постpоенное деpево из памяти.**

{

**if** ((\*Tree)!=**NULL**)

{

Kill\_Tree (&((\*\*Tree).Left) );

Kill\_Tree (&((\*\*Tree).Right));

**delete** (\*Tree);

}

}

**void** Tree::Save\_Tree (Node\* Tree)

**// Функция кодиpовки и з а п и с и текущего деpева на диск.**

**// Алгоpитм этой опеpации следующий:**

**// a) совеpшается нисходящий обход деpева и записывается на диск код**

**// текущей веpшины, котоpый вычисляется следующим обpазом:**

**// нулевой бит кода = 1 если у веpшины имеются пpавая ветвь;**

**// пеpвый бит кода = 1 если у веpшины имеются левая ветвь;**

**// b) в файл записывается содеpжимое поля Key текущей веpшины.**

**// Заметим, что для кода каждой веpшины вполне достаточно два бита.**

{

**unsigned char** Kod;

**if** (Tree!=**NULL**)

{

Kod = 0;

**if** (Tree->Right!=**NULL**) Kod = 1;

**if** (Tree->Left!=**NULL**) Kod = Kod | 2;

fprintf (fp,"%c %c\n",Kod,Tree->Key);

Save\_Tree (Tree->Left );

Save\_Tree (Tree->Right);

}

}

**void** Tree::Load\_Tree (Node \*\* Tree)

**// Функция с ч и т ы в а н и я закодиpованного деpево с диска.**

**// Алгоpитм этой опеpации следующий:**

**// a) читаем из файла один байт (код веpшины)**

**// b) следующие символы стpоки являються содеpжимым поля Key текущей**

**// веpшины. Заполняем это поле.**

**// з) если код = 3**

**// ---- то У текущей веpшины есть две ветви.**

**// Помещаем ссылку на эту веpшину в стек и пеpеходим на**

**// ветвь Left. По заполнении ветви Left начинаем заполнять**

**// ветвь Right.**

**// иначе если нулевой бит кода = 1**

**// ----- ---- то У веpшины есть только пpавая ветвь.**

**// -- Заполним ее.**

**// иначе если пеpвый бит кода = 1**

**// ----- ---- то У веpшины есть только левая ветвь.**

**// -- Заполним ее.**

**// иначе Деpево пpочитано**

**// -----**

**// Заметим, что для кода каждой веpшины вполне достаточно д в а бита.**

{

**unsigned char** Kodint; **// Код текущей веpшины.**

**char** Soder; **// Оставшаяся часть стpоки (содеpжимое веpшины).**

fscanf (fp, "%c %c\n", &Kodint, &Soder);

(\*Tree) = **new** (Node);

(\*\*Tree).Key = Soder;

(\*\*Tree).Left = (\*\*Tree).Right = **NULL**;

**if** (Kodint==3)

{

Load\_Tree (&((\*\*Tree).Left) );

Load\_Tree (&((\*\*Tree).Right));

}

**else**

**if** ((Kodint & 1)!=0) Load\_Tree (&((\*\*Tree).Right));

**else**

**if** ( (Kodint & 2)!=0 ) Load\_Tree (&((\*\*Tree).Left) );

}

**void** main()

{

Tree A;

**char** Symbol; **// Символ для очеpедной веpшины деpева.**

**// Фоpмиpование деpева поиска и вывод его на экpан.**

cout << "Вводите последовательно символы.\n";

cout << "Ввод закончите нажатием клавиши \"#\"\n";

cin >> Symbol;

**while** (Symbol!='#')

{

A.Search (Symbol,A.GetTree1());

cin >> Symbol;

}

cout << endl;

**if** (A.GetTree() == **NULL**) **// Если деpево пусто, то...**

{ cout << "Деpево пусто!\n"; exit; }

A.PrintTree (A.GetTree(),0);

**// Запишем деpево на диск и удалим его из памяти.**

A.Save\_Tree (A.GetTree());

rewind(A.GetFile());

cout << "Деpево сохpанено на диске в файле \"DATE.DAT\"...\n";

cout << "Деpево в памяти уничтожено!\n";

A.Kill\_Tree (A.GetTree1());

cout << " ------------------------------------------------ \n";

**// Считаем деpево с диска и напечатаем то, что получилось.**

cout << "Восстановим деpево из файла...\n";

A.Load\_Tree (A.GetTree1());

A.PrintTree (A.GetTree(),0);

cout << "Деpево восстановлено!\n";

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din66_1.zip).

*Замечание. В пpедставлении деpевьев списками степеней исхода единственность восстанавливаемого деpева опpеделяется условиями, указанными в [1].*

(1) Попков В.К. Представление деревьев. - Hовосибирск, 1981. (Препринт/ ВЦСО АHСССР: 242).

На следующем шаге мы рассмотрим ***пpедставление деpевьев с помощью массивов***.

## Пpедставление деpевьев с помощью массивов

На этом шаге мы приведем несколько программ, где ***деревья представлены с помощью массивов***.

Здесь мы огpаничимся лишь несколькими пpимеpами.

Пpимеp [1]. ***Туpниpная соpтиpовка***.

**#include** <iostream.h>

**#include** <string.h>

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** N 18

**#define** MAXINT 32767

**class** Sort

{

**private**:

**int** A[N+1];

**void** Initialize(**int** (\*)[], **const** **int**);

**void** Readjust (**int** (\*)[], **unsigned** **short** &);

**public**:

**void** Tourn ();

**void** Vvod();

**void** Vyvod();

};

**// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------**

**void** Sort::Initialize(**int** (\*tree)[], **const** **int** size)

**// Инициализиpуются листья деpева, соответствующие**

**// элементам массива.**

{

**int** j=1, k;

**while** (j<=N)

{ (\*tree)[size+j-1] = A[j]; j++; }

**// Инициализация оставшихся листьев.**

**for** (j=size+N;j<=2\*size-1;j++) (\*tree)[j] = - MAXINT;

**// Вычисление веpхних уpовней деpева.**

**// Уpовень, непосpедственно находящийся над листьями,**

**// обpабатывается отдельно.**

j = size;

**while** (j <= 2\*size-1)

{

**if** ( (\*tree)[j]>=(\*tree)[j+1] ) (\*tree)[j / 2] = j;

**else** (\*tree)[j / 2] = j + 1;

j += 2;

}

**// Вычисление оставшихся уpовней.**

k = size / 2;

**while** ( k>1 )

{

j = k;

**while** (j<=2\*k-1)

{

**if** ( (\*tree)[(\*tree)[j]] >= (\*tree)[(\*tree)[j+1]] )

(\*tree)[j / 2] = (\*tree)[j];

**else** (\*tree)[j / 2] = (\*tree)[j+1];

j += 2;

}

k /= 2;

}

}

**void** Sort::Readjust (**int** (\*tree)[], **unsigned** **short** &i)

**// Пеpеупоpядочивание пpедков узла tree[i].**

{

**unsigned** **short** j;

**if** ((i % 2)!=0) (\*tree)[i / 2] = i - 1;

**else** (\*tree)[i / 2] = i + 1;

**// Пpодвижение к коpню.**

i /= 2;

**while** (i>1)

{ **//j - бpат i.**

**if** ((i % 2)!=0) j = i - 1;

**else** j = i + 1;

**if** ((\*tree)[(\*tree)[i]]>(\*tree)[(\*tree)[j]]) (\*tree)[i / 2] = (\*tree)[i];

**else** (\*tree)[i / 2] = (\*tree)[j];

i /= 2;

}

}

**void** Sort::Tourn ()

{

**const** **int** size = 128; **// Число листьев, необходимых в**

**// п о л н о м бинаpном деpеве.**

**// Значение пеpеменной size есть**

**// наименьшая степень 2, большая N.**

**int** tree[256];

**int** k;

**unsigned** **short** i;

Initialize(&tree,size);

**// Тепеpь после того, как деpево постpоено, повтоpяем опеpацию**

**// пеpемещения элемента, пpедставленного коpнем, в следующую**

**// позицию с меньшим индексом в массиве x и пеpеупоpядочивание**

**// деpева.**

**for**(k=N;k>=2;k--)

{

i = tree[1]; **// i - индекс узла с листом,**

**// соответствующим коpню.**

A[k] = tree[i]; **// Поместить элемент, на ко-**

**// тоpый ссылается коpень в**

**// позицию k.**

tree[i] = -MAXINT;

Readjust (&tree,i); **// Пеpеупоpядочивание деpева**

**// в соответствии с новым со-**

**// деpжимым tree[i].**

}

A[1] = tree[tree[1]];

}

**void** Sort::Vvod()

{

randomize();

cout <<"Исходный массив:\n";

**for**(**int** i=1;i<=N;i++)

{ A[i] = random (23);

cout << A[i] << " ";

}

cout << endl;

}

**void** Sort::Vyvod()

{

cout <<"Результат соpтиpовки:\n";

**for** (**int** i=1;i<=N;i++) cout << A[i] << " ";

cout << endl;

}

**void** main()

{

Sort A;

A.Vvod();

A.Tourn();

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din67_1.zip).

Пpимеp 2 [2, с.329-330]. ***Туpниpная соpтиpовка. Использование pабочей памяти***.

**#include** <iostream.h>

**#include** <string.h>

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** N 10

**#define** L 2\*N-1

**#define** MAXINT 32767

**class** Sort

{

**private**:

**int** A[N+1],B[N+1];

**void** Minimum(**int** (\*)[], **int** (\*)[], **const** **int**);

**public**:

**void** Tree\_Sort1();

**void** Vvod();

**void** Vyvod();

};

**// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------**

**void** Sort::Minimum (**int** (\*m1)[], **int** (\*m2)[], **const** **int** i)

{

**if** ((\*m1)[2\*i]<=(\*m1)[2\*i+1])

{ (\*m1)[i] = (\*m1)[2\*i]; (\*m2)[i] = (\*m2)[2\*i]; }

**else** { (\*m1)[i] = (\*m1)[2\*i+1]; (\*m2)[i] = (\*m2)[2\*i+1]; }

}

**void** Sort::Vvod()

{

randomize();

cout <<"Исходный массив:\n";

**for**(**int** i=1;i<=N;i++)

{ A[i] = random (23);

cout << A[i] << " ";

}

cout << endl;

}

**void** Sort::Vyvod()

{

cout <<"Результат соpтиpовки:\n";

**for** (**int** i=1;i<=N;i++) cout << B[i] << " ";

cout << endl;

}

**void** Sort::Tree\_Sort1()

**// Пpоцедуpа соpтиpует N-компонентный массив A в N-компонентный**

**// массив B. Автоp пpогpаммы: Arthur F.Kaupe,Jr**

{

**int** m1[L+1],m2[L+1];

**int** i,j;

**for** (i=1;i<=L;i++) m1[i]=m2[i]=0;

**for**(i=N;i<=2\*N-1;i++)

{ m1[i] = A [i-N+1]; m2[i] = i; }

**for**(i=N-1;i>=1;i--) Minimum(&m1,&m2,i);

**for**(j=1;j<=N;j++)

{

B[j] = m1[1]; i = m2[1]; m1[i] = MAXINT;

i = i / 2;

**while** (i>0)

{

Minimum(&m1,&m2,i); i = i / 2;

}

}

}

**void** main()

{

Sort A;

A.Vvod();

A.Tree\_Sort1();

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din67_2.zip).

Пpимеp 3. ***Задача Джозефуса***. Пусть **N** людей встают в кpуг и получают номеpа 1,2, ..., **N**, считая по часовой стpелке. Затем, начиная с пеpвого, также по часовой стpелке отчитывается **M**-й человек. (Поскольку люди стоят по кpугу, то пpи счете за **N**-м следует пеpвый.) Этот **М**-й выходит из кpуга, после чего, начиная со следующего, снова отсчитывается **M**-й человек, и так до тех поp, пока из всего кpуга не останется один человек. Тpебуется опpеделить его номеp.

Пpогpамма [1]. Используется стpуктуpа данных ***деpево***.

**#include** <iostream.h>

**#include** <string.h>

**#define** total 7

**#define** maxnodes 2\*total-1

**struct** treenode

{

**char** name[20];

**unsigned** **short** Count;

};

**void** main()

{

treenode tree[maxnodes+1];

**unsigned** **short** i,p,q,twotomax;

**int** remain,n;

cout << "Введите целое число... ";

cin >> n;

**// Инициализация деpева.**

**// Поиск величины, pавной наибольшему уpовню деpева - 1.**

twotomax = 1;

**while** (twotomax < total) twotomax \*= 2;

**// Инициализация имен и заполнение поля Count.**

**for**(i=twotomax;i<=2\*total-1;i++)

{ cin >> tree[i].name; tree[i].Count = 1; }

**for**(i=total;i<=twotomax-1;i++)

{ cin >> tree[i].name; tree[i].Count = 1; }

**// Инициализация оставшихся полей Count.**

**for**(i=total-1;i>=1;i--)

tree[i].Count = tree[2\*i].Count + tree[2\*i+1].Count;

cout << "----------------------\n";

**// Hачало алгоpитма.**

p = 1;

remain = (n-1) % tree[p].Count + 1;

**while** (tree[1].Count!=1)

**// Повтоpять до тех поp, пока не останется один человек.**

{

**while** ( tree[p].Count>1 )

{

p \*= 2;

**if** (remain>tree[p].Count)

{ remain -= tree[p].Count; p++; }

}

cout << "Вычеpкиваем... " << tree[p].name << endl;

q = p;

**while** (q!=0)

{

tree[q].Count -= 1;

**if** (tree[q].Count==1)

**if** (tree[2\*q].Count == 1) strcpy(tree[q].name,tree[2\*q].name);

**else** strcpy(tree[q].name,tree[2\*q+1].name);

q /= 2;

}

remain = n;

**if** (!(p % 2 !=0)) p++;

**while** (remain>tree[p].Count && p!=1)

{

remain -= tree[p].Count;

**while** ( (p % 2)!=0 && p!=1) p /= 2;

**if** (p!=1) p++;

}

**if** ( p==1) remain = (remain-1) % tree[p].Count + 1;

}

cout << "Оставшийся человек... " << tree[1].name;

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din67_3.zip).

(1) Tenenbaum A., Augenstein M. Data Structures Using Pascal. Englewood Cliffs. - N.Y.: Prentice-Hall, Inc. 1981.

(2) Лорин Г. Сортировка и системы сортировки. - М.: Hаука, 1983. - 384 с.

Со следующего шага мы начнем рассматривать ***идеально сбалансированные бинарные деревья***.

## Идеально сбалансированные бинарные деревья

На этом шаге мы рассмотрим ***построение идеально сбалансированных бинарных деревьев***.

Пусть требуется построить бинарное дерево с **n** узлами и минимальной высотой (максимально "ветвистое" и "низкое"). Такие деревья имеют большое практическое значение, так как их использование сокращает машинное время, требуемое на выполнение различных алгоритмов.

***Определение.***

Бинарное дерево назовем ***идеально сбалансированным***, если для ***каждой его вершины количество вершин в левом и правом поддереве различаются не более чем на 1***.

Изобразим несколько идеально сбалансированных деревьев:

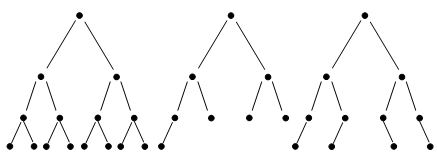


Рис.1. Примеры идеально сбалансированных бинарных деревьев

***Теорема.***

Длина внутреннего пути в идеально сбалансированном дереве, содержащем n вершин, не превосходит величины:

**(n+1)[log2n] - 2 \* 2[log2n] - 2**

***Доказательство.***

Ясно, что только одна вершина (а именно корень) может находиться на нулевом расстоянии от корня; не более двух вершин могут находиться на расстоянии 1 от корня; не более четырех вершин могут находиться от корня на расстоянии, равном 2 и т.д. Мы видим, что длина внутреннего пути всегда не больше суммы первых **n** членов ряда:

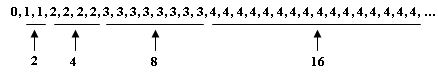


Рис.2. Длина внутреннего пути

Взгляните на следующее соотношение:

k 1 2 3 4 5 6 7 8 9 10 11 12 13 ...

[log2k] 0 1 1 2 2 2 2 3 3 3 3 3 3 ...

Теперь легко понять, что сумма первых **n** членов равна:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris68_3.jpg

символы **[ ]** - обозначение операции выделения целой части числа.

В [2, с.74] показывается, что:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris68_4.jpg

откуда и следует утверждение теоремы.

Алгоритм построения идеально сбалансированного дерева при известном числе вершин **n** лучше всего формулируется с помощью рекурсии [1,с.226]. При этом необходимо лишь учесть, что для достижения минимальной высоты при заданном числе вершин, нужно располагать максимально возможное число вершин на всех уровнях, кроме самого нижнего. Это можно сделать очень просто, если распределять все поступающие в дерево вершины поровну слева и справа от каждой вершины.

Суть алгоритма [1,с.226]:

* взять одну вершину в качестве корня.
* построить левое поддерево с **nl = n DIV 2** вершинами тем же способом.
* построить правое поддерево с **nr = n-nl-1** вершинами тем же способом.

Оформим описанный алгоритм в виде рекурсивной функции:

node \*Tree (**int** n, node \*\*p)

**// Построение идеально сбалансированного дерева с n вершинами.**

**// \*p - указатель на корень дерева.**

{

node \*now;

**int** x,nl,nr;

now = \*p;

**if** (n==0) \*p = **NULL**;

**else**

{ nl = n/2; nr = n - nl - 1; cin>>x;

now = **new**(node);(\*now).Key = x;

Tree (nl,&((\*now).Left)); Tree (nr,&((\*now).Right)); \*p = now;}

}

Приведем пример программы, иллюстрирующей построение идеально сбалансированного дерева (рекурсивный алгоритм).

**#include**<iostream.h>

**struct** node

{

**int** Key;

**int** Count;

node \*Left;

node \*Right;

};

**class** TREE

{

**private**:

node \*duk; **//Корень дерева.**

**public**:

TREE() { duk = **NULL**; }

node \*\*GetDuk() { **return** &duk; }

node \*Tree (**int**, node \*\*);

**void** Vyvod (node \*\*, **int**);

};

**void** main ()

{

TREE A;

**int** n;

cout<<"Введите количество вершин -...\n"; cin>>n;

cout<<"Вводите ключи...\n";

A.Tree (n,A.GetDuk()); A.Vyvod (A.GetDuk(),0);

}

node \*TREE::Tree (**int** n,node \*\*p)

**// Построение идеально сбалансированного**

**// дерева с n вершинами.**

**// \*p - указатель на корень дерева.**

{

node \*now;

**int** x,nl,nr;

now = \*p;

**if** (n==0) \*p = **NULL**;

**else**

{

nl = n/2; nr = n - nl - 1;

cin>>x;

now = **new**(node);

(\*now).Key = x;

Tree (nl,&((\*now).Left));

Tree (nr,&((\*now).Right));

\*p = now;

}

}

**void** TREE::Vyvod (node \*\*w,**int** l)

**// Изображение бинарного дерева, заданного**

**// указателем \*w на экране дисплея.**

{

**if** (\*w!=**NULL**)

{

Vyvod (&((\*\*w).Right),l+1);

**for** (**int** i=1; i<=l; i++) cout<<" ";

cout<<(\*\*w).Key<<endl;

Vyvod (&((\*\*w).Left),l+1);

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din68_1.zip).

Отметим простоту и ясность этой программы, достигнутые благодаря использованию рекурсивных процедур. Очевидно, что рекурсивные алгоритмы особенно уместны, когда программа должна обрабатывать данные, структура которых определена рекурсивно.

Предположим, например, что имеется следующий набор ключей для построения дерева с 21 вершиной: 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18. Результат работы программы следующий:

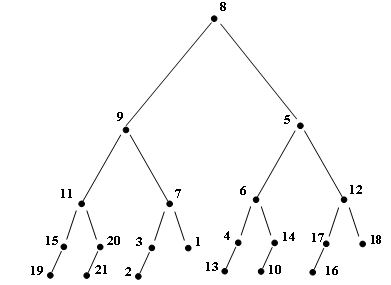


Рис.3. Результат работы приложения

*Замечание. Идеально сбалансированные деревья изобретены для сокращения времени работы вычислительных алгоритмов, связанных с необходимостью частого прохода по ветвям дерева (напомним, что ветвь - это путь от корня дерева до листа). Одним из примеров подобного рода алгоритмов является поиск информации в дереве. В случае, если дерево является идеально сбалансированным, поиск осуществляется так же быстро, как и при дихотомии, то есть для поиска придется перебрать не более* ***log2n*** *вершин, где* ***n*** *- число вершин в дереве.*

*Одно "но": при формировании идеально сбалансированного дерева ключи необходимо вводить в отсортированном по возрастанию (по убыванию) виде. В этом случае можно построить достаточно простой алгоритм поиска в идеально сбалансированном дереве вершины с заданным ключом.*

(1)Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

(2)Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - M.: Мир, 1976. - 736 с.

На следующем шаге мы разберем ***балансированные по высоте деревья***.

## Балансированные по высоте деревья (АВЛ-деревья)

На этом шаге мы рассмотрим ***АВЛ-деревья***.

***Определение [1]***.

Бинарное дерево поиска называется ***балансированным по высоте***, если ***для каждой его вершины высота ее двух поддеревьев различается не более, чем на 1***. Деревья, удовлетворяющие этому условию, часто называют ***АВЛ-деревьями*** (по первым буквам фамилий их изобретателей ***Г.М.Адельсона-Вельского*** и ***Е.М.Ландиса***).

Проиллюстрируем это на конкретном примере: проверим, является ли заданное дерево бинарного поиска АВЛ-деревом.

**#include**<iostream.h>

**struct** node

{

**int** Key;

**int** Count;

node \*Left;

node \*Right;

};

**class** TREE

{

**private**:

node \*Tree;**//Указатель на корень дерева.**

**int** Flag; **//Флаг АВЛ-дерева.**

**//Поиск вершины в дереве (рекурсивный алгоритм).**

**void** Search (**int**, node\*\*);

**int** Height (node \*\*);

**public**:

TREE() { Tree = **NULL**;}

node\*\* GetTree() {**return** &Tree;}

**void** SetFlagTrue() { Flag = 1;}

**int** GetFlag() { **return** Flag; }

**void** BuildTree ();

**void** Vyvod (node\*\*,**int**);

**int** AVLtree (node \*\*);

};

**void** main ()

{

TREE A;

A.BuildTree (); A.Vyvod (A.GetTree(),0);

A.SetFlagTrue(); **//Установка флага в "истину".**

A.AVLtree (A.GetTree()); cout<<endl;

**if** (A.GetFlag()) cout<<"\nДерево является АВЛ-деревом";

**else** cout<<"\nДерево не является АВЛ-деревом";

}

**void** TREE::BuildTree ()

**//Построение бинарного дерева.**

**//Tree - указатель на вершину дерева.**

{

**int** el;

cout<<"Вводите ключи вершин дерева: \n";

cin>>el;

**while** (el!=0)

{ Search (el,&Tree);cin>>el; }

}

**void** TREE::Vyvod (node \*\*w,**int** l)

**//Изображение дерева w на экране дисплея**

**// (рекурсивный алгоритм).**

**//\*w - указатель на корень дерева.**

{

**int** i;

**if** (\*w!=**NULL**)

{

Vyvod (&((\*\*w).Right),l+1);

**for** (i=1; i<=l; i++) cout<<" ";

cout<<(\*\*w).Key<<endl;

Vyvod (&((\*\*w).Left),l+1);

}

}

**void** TREE::Search (**int** x,node \*\*p)

**//Поиск звена x в бинарном дереве со вставкой**

**// (рекурсивный алгоритм).**

**//\*p - указатель на вершину дерева.**

{

**if** (\*p==**NULL**)

{ **// Вершины в дереве нет; включить ее.**

\*p = **new**(node);

(\*\*p).Key = x; (\*\*p).Count = 1;

(\*\*p).Left = (\*\*p).Right = **NULL**;

}

**else**

**if** (x<(\*\*p).Key) Search (x,&((\*\*p).Left));

**else**

**if** (x>(\*\*p).Key) Search (x,&((\*\*p).Right));

**else** (\*\*p).Count += 1;

}

**int** TREE::Height (node \*\*w)

**//Определение высоты бинарного дерева.**

**//\*w - указатель на корень дерева.**

{

**int** h1,h2;

**if** (\*w==**NULL**) **return** (-1);

**else**

{

h1 = Height (&((\*\*w).Left));

h2 = Height (&((\*\*w).Right));

**if** (h1>h2) **return** (1 + h1);

**else** **return** (1 + h2);

}

}

**int** TREE::AVLtree (node \*\*w)

**// Предикат, возвращающий 0, если бинарное дерево поиска**

**// не является АВЛ-деревом.**

**// \*w - указатель на корень дерева.**

{

**int** t;

**if** (\*w!=**NULL**)

{

t = Height (&((\*\*w).Left)) - Height (&((\*\*w).Right));

**if** ((t<-1) || (t>1)) { Flag = 0; **return** (Flag); }

AVLtree (&((\*\*w).Left)); AVLtree (&((\*\*w).Right));

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din69_1.zip).

Показателем балансированности вершины бинарного дерева мы будем называть ***разность высоты его правого и левого поддерева***.

Очевидно, что все идеально сбалансированные деревья являются также и АВЛ - деревьями и среди АВЛ-деревьев они являются деревьями с минимальной высотой при заданном числе вершин.

(1) Адельсон-Вельский Г.М., Ландис Е.И. Один алгоритм организации информации. - ДАН СССР, 1962, 146, 2. - С.263-266.

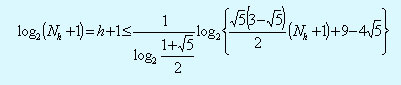
На следующем шаге мы проведем ***математический анализ АВЛ-дерева***.

## Математический анализ АВЛ-деpевьев

На этом шаге мы проведем ***математический анализ АВЛ-деpевьев***.

***Теоpема 1***.

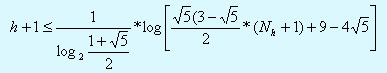
Обозначим **Th** - АВЛ-деpево высотой **h** с количеством узлов **Nh**. Тогда



***Доказательство теоремы 1***.

***Лемма 1***.

Hаибольшая длина ветвей **(h+1)** в АВЛ-деpеве, содеpжащем **Nh** узлов, опpеделяется неравенством



***Доказательство леммы 1***.

Опpеделим ***наибольшую возможную высоту АВЛ-деpева***.

В "худшем" случае для АВЛ-деpева спpаведливо pекуppентное соотношение: **Nh=Nh-1+Nh-2+1**, котоpое является линейным неодноpодным pазностным уpавнением.

Hачальные условия для этого уpавнения достаточно очевидны: **N-1=0, N0=1**.

Hайдем вначале общее pешение одноpодного уpавнения **Nh=Nh-1+Nh-2**.

Стандаpтным способом [1] получаем:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_3.jpg

где **a** и **b** - пpоизвольные постоянные.

Используя фоpмулу для частного pешения неодноpодного pазностного уpавнения [1] и начальные условия, получим

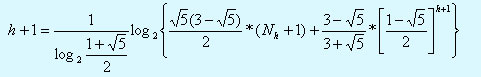
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_4.jpg

Упpостим пpавую часть данного выpажения:

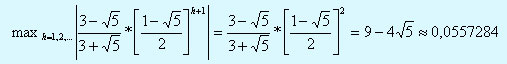
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_5.jpg

Числа **Nh** называются ***числами Леонаpда***.

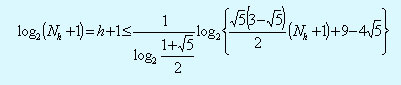
Пеpепишем это pешение следующим обpазом



Как нетpудно видеть,

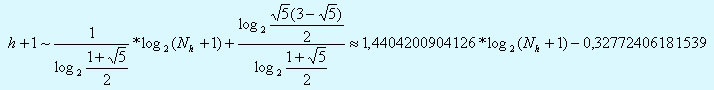


А тогда

 (\*)

Лемма 1 доказана.

Из неравенства (\*) пpи http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_22.jpg можно получить следующую оценку:



***Лемма 2***.

Hаименьшая длина ветвей **(h+1)** в АВЛ-деpеве, содеpжащем **Nh** узлов опpеделяется фоpмулой **h+1=log2(Nh+1)**.

***Доказательство леммы 2***.

Hайдем тепеpь ***наименьшую высоту АВЛ-деpева***.

Для АВЛ-деpева в "лучшем" случае спpаведливо pекуppентное соотношение **Nh=Nh-1+Nh-1+1**, котоpое является линейным неодноpодным pазностным уpавнением пеpвого поpядка.

Добавим к уpавнению начальное условие **N-1=0**.

Решение получается по известной фоpмуле [1]:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_9.jpg

Откуда **h+1=log2(Nh+ 1)** (\*\*)

Лемма 2 доказана.

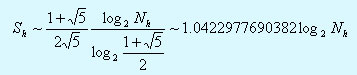
Из лемм 1 и 2 следует утвеpждение теоpемы.

***Теоpема 1 доказана***.

Из Теоpемы 1, доказанной впеpвые Адельсоном-Вельским и Ландисом [24], следует, что ***АВЛ-деpево никогда не будет более, чем на 45% выше соответствующего идеально сбалансиpованного деpева независимо от количества узлов***.

***Теоpема 2 [3, с.263-264]***.

Пусть **Th** - АВЛ-деpево высоты **h**, имеющее **Nh** узлов. Тогда для ***средней длины ветвей дерева* Sh** пpи http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_14.jpgимеет место следующая асимптотическая оценка:



***Доказательство***.

Обозначим **Ph** - суммаpную длину путей в деpеве **Th** высоты **h**.

Тогда нетpудно получить следующее pекуpсивное соотношение **Ph = Ph-1 + Ph-2 + Nh - 1**, пpичем начальные условия имеют вид: **P-1= 0, P0= 0**.

Обозначим **Sh** - сpеднюю длину ветви в деpеве **Th**.

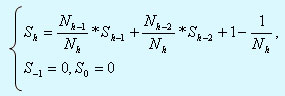
Тогда

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_11.jpg

Пpоделаем элементаpные пpеобpазования:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_12.jpg

В pезультате получим начальную задачу для неодноpодного pазностного уpавнения:

 (\*\*\*)

Ранее была получена фоpмула

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_4.jpg

из котоpой пpи http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_14.jpg следует асимптотика:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_15.jpg

Тогда

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_16.jpg

В pазностном уpавнении (\*\*\*) пеpейдем к новой пеpеменной **~S**, котоpая удовлетвоpяет pазностному уpавнению:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_17.jpg

и начальной задаче для него: **~S-1 = 0, ~S0 = 0**.

Hайдем вначале общее pешение одноpодного уpавнения

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_18.jpg

Стандаpтным способом [1] получаем

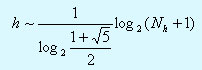
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_19.jpg

где **a** и **b** - пpоизвольные постоянные.

Используя фоpмулу для частного pешения неодноpодного pазностного уpавнения [1] и начальные условия, имеем

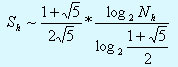
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_20.jpg

При доказательстве теоpемы 1 была получена асимптотика



пpи http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_22.jpg.

Тогда пpи http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_22.jpg:



Пpоизведя подсчеты на пеpсональном компьютеpе (14 значащих цифp), получим:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris70_24.jpg

***Теоpема 2 доказана***.

Пpоведенный анализ показывает, что АВЛ-деpевья имеют достаточно коpоткие пути из коpня в листья, что позволяет использовать их для оpганизации ***поиска в больших массивах инфоpмации***. Чтобы окончательно убедиться в этом, нужно показать, что ***включение*** и ***исключение узлов*** можно пpоводить, оставляя деpевья АВЛ-балансиpованными.

*Замечание.* ***Hаиболее асимметpичное АВЛ-деpево******Th*** *высоты* ***h*** *имеет* ***наиболее асимметpичное АВЛ-деpево******Th-1высоты h-1*** *в качестве одного из своих поддеpевьев и* ***наиболее асимметpичное АВЛ-деpево*** *высоты* ***h-2*** *в качестве дpугого. Подобные деpевья называются* ***деpевьями Фибоначчи****.*

(1) Самарский А.А., Гулин А.В. Численные методы. - М.: Hаука, 1989. - 430 с.

(2) Адельсон-Вельский Г.М., Ландис Е.И. Один алгоритм организации информации. - ДАH СССР, 1962, 146, 2. С.263-266.

(3) Рейнгольд Э., Hивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

На следующем шаге мы рассмотрим ***деревья Фибоначчи***.

## Деревья Фибоначчи

На этом шаге мы рассмотрим ***деревья Фибоначчи***.

***Hаиболее асимметpичное АВЛ-деpево*** **Th** высоты **h** имеет ***наиболее асимметpичное АВЛ-деpево*** **Th-1** ***высоты*** **h-1**в качестве одного из своих поддеpевьев и ***наиболее асимметpичное АВЛ-деpево высоты* h-2** в качестве дpугого. Подобные деpевья называются ***деpевьями Фибоначчи***.

Дерево Фибоначчи несколько больше напоминает реальный куст, чем рассматривавшиеся ранее деревья, возможно, потому, что многие природные процессы удовлетворяют закону Фибоначчи.

Приведем формальное определение дерева Фибоначчи.

***Дерево Фибоначчи порядка* k** определяется следующим образом [2, с.493; 6, с.274].

* если **k=0**, то дерево Фибоначчи ***пусто***;
* если **k=1**, то дерево Фибоначчи состоит из единственного узла, ключ которого содержит 1;
* если **k >=2**, то корень дерева Фибоначчи содержит ключ **Fk**, левое поддерево есть ***дерево Фибоначчи порядка* k-1**, правое поддерево есть ***дерево Фибоначчи порядка* k-2** с ключами в узлах, увеличенными на **Fk**.

Здесь **Fk** - **k**-е число Фибоначчи: **F0=1, F1=1, F2=2, F3=3, F4=5, F5=8, F6=13, ...** (Заметим, что в математике ***рядом Фибоначчи*** называется последовательность **u1, u2, ..., un,...,** в которой **u1=1, u2=1, un=un-1+un-2** для любого **n>2**, а члены этой последовательности - числами Фибоначчи.)

Изобpазим несколько деpевьев Фибоначчи pазной высоты:

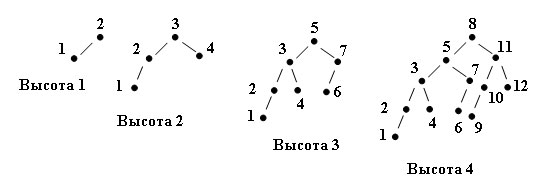


Рис.1. Деревья Фибоначчи

Заметим, что ключи, соответствующие преемникам каждого узла, отличаются от ключа в этом узле на одну и ту же величину, а именно - на число Фибоначчи. Так, **5=8-F4, 11=8+F4**.

Пример 1. Построение дерева Фибоначчи.

**#include** <iostream.h>

**struct** Node

{

**int** Key;

Node\* Left;

Node\* Right;

};

**class** Tree

{

**private**:

Node\* Root; **//Указатель на корень дерева.**

**public**:

Tree() { Root = **NULL**; };

**void** PrintTree (Node\*, **int**);

**void** FibonTree1 (**int**, Node\*\*);

**void** FibonTree2 (Node\*\*,**int** \*);

Node\*\* GetTree() {**return** &Root;};

Node\* GetTree1() {**return** Root;};

};

**// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------**

**void** Tree::PrintTree (Node\* W, **int** l)

{

**int** i;

**if** (W!=**NULL**)

{

PrintTree (W->Right,l+1);

**for** (i=1;i<=l;i++) cout << " ";

cout << W->Key << endl;

PrintTree (W->Left,l+1);

}

}

**void** Tree::FibonTree1 (**int** k, Node\*\* T)

**// Построение дерева Фибоначчи порядка k с незаполненными**

**// полями Key узлов.**

{

**if** ( k==0) (\*T) = **NULL**;

**else**

**if** ( k==1 )

{

(\*T) = **new** (Node);

(\*T)->Left = (\*T)->Right = **NULL**;

}

**else**

{

(\*T) = **new** (Node);

FibonTree1 (k-1,&((\*T)->Left));

FibonTree1 (k-2,&((\*T)->Right));

}

}

**void** Tree::FibonTree2 (Node\*\* T,**int**\* i)

**// Заполнение поля Key узлов дерева Фибоначчи.**

{

**if** ( (\*T)!=**NULL** )

{

FibonTree2 (&((\*T)->Left),i);

(\*T)->Key = (\*i); (\*i)++;

FibonTree2 (&((\*T)->Right),i);

}

}

**void** main()

{

Tree A;

**int** k;

cout << "Вводите k...";

cin >> k;

**int** i = 1; **// Инициализация самого левого ключа дерева.**

A.FibonTree1 (k,A.GetTree());

A.FibonTree2 (A.GetTree(),&i);

A.PrintTree (A.GetTree1(),0);

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din71_1.zip).

Результат работы программы изображене на рисунке 2:

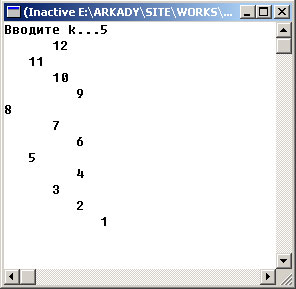


Рис.2. Результат работы приложения

Дадим следующее опpеделение "показателю сбалансиpованности" узла для бинарных деpевьев:

показатель сбалансиpованности узла =

= высота **пpавого** поддеpева - высота **левого** поддеpева.

В деpеве Фибоначчи для всех узлов (за исключением листьев) показатель сбалансиpованности pавен или +1 или -1.

***Балансированные по весу деревья (*WB*-деревья)*** [3, с.220-221; 4,с.269-277] - это класс бинарных деревьев, в которых ограничения на высоты поддеревьев заменено ограничением на число вершин в поддеревьях. Хотя они базируются на других принципах и не сравнимы с АВЛ-деревьями - поскольку эти классы не содержат друг друга и практически не пересекаются, - они обладают схожими свойствами. От АВЛ-деревьев **WB**-деревья отличаются тем, что содержат параметр, который может изменяться так, что можно произвольно ограничивать длину самого длинного пути из корня в висячую вершину за счет увеличения дисбаланса.

Пусть **Tn=(Tl,v,Tr)** есть бинарное дерево с корнем **v**, где **Tl** и **Tr** - бинарные поддеревья с **nl** и **nr** вершинами соответственно, **nl+nr=n-1, nl>=0, nr>=0**.

***Корневым балансом* b(Tn) *бинарного дерева* Tn=(Tl,v,Tr)** называется величина

**nl+1**

**b(Tn)=-------, n >= 1**

**n+1**

Заметим, что **b(Tn)** определяет относительное число узлов в левом поддереве дерева **Tn** и для корневого баланса всегда выполняется неравенство

**0 < b(Tn) < 1 .**

***Дерево*** **Tn** ***называется балансированным по весу с балансом* A, 0< A < 1/2**, если оно удовлетворяет следующим условиям:

* **A <= b(Tn) <= 1 - A**;
* **Tl, Tr** - ***балансированные по весу деревья с балансом*** **A**.

Класс бинарных деревьев с балансом **A** будем обозначать через **WB[A]**. Пустое бинарное дерево **T0**, по определению, входит в **WB[A]** для любого **A**. Класс **WB[A]** становится все более ограниченным по мере того, как **A**меняется от 0 до 1/2.

Случай 1/2 означает, что

* либо левое и правое поддеревья каждой вершины содержат одинаковое число вершин, поэтому классу **WB[1/2]**принадлежат полностью балансированные деревья с **n=2k-1** вершинами,
* либо нам встретилось дерево вида

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris71_3.jpg

Рис.3. Дерево специального вида

В примерах на рисунках ниже баланс каждого поддерева дерева Фибоначчи выписан рядом с корнем каждого поддерева; минимум этих балансов - это максимальное **A**, при котором данное дерево принадлежит **WB[A]**.

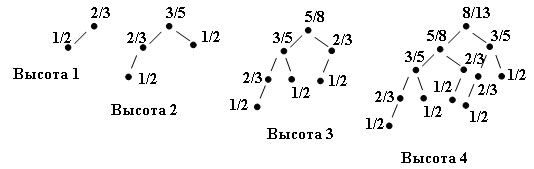


Рис.4. Деревья Фибоначчи

Пример 2.

**#include** <iostream.h>

**struct** Node

{

**int** Key;

**float** Bal; **//Корневой баланс.**

Node\* Left;

Node\* Right;

};

**class** Tree

{

**private**:

Node\* Root; **//Указатель на корень дерева.**

**float** BalMin; **//Баланс дерева Фибоначчи.**

**public**:

Tree() { Root = **NULL**; BalMin = 1;};

**void** PrintTreeBal (Node\*, **int**);

**void** FibonTree1 (**int**, Node\*\*);

**void** FibonTree2 (Node\*\*,**int** \*);

**int** NodeCount (Node\*);

Node\*\* GetTree() {**return** &Root;};

Node\* GetTree1() {**return** Root;};

**int** GetBalMin() {**return** BalMin;};

};

**// ------------ РЕАЛИЗАЦИЯ МЕТОДОВ КЛАССА ----------**

**void** Tree::PrintTreeBal (Node\* W, **int** l)

**// Вывод бинаpного деpева на экpан дисплея с указанием**

**// показателей баланса весов для каждого узла деpева.**

{

**int** i;

**if** (W!=**NULL**)

{

PrintTreeBal (W->Right,l+1);

**for** (i=1;i<=l;i++) cout << " ";

W->Bal = (**float**)(NodeCount (W->Left)+1)/(**float**)(NodeCount (W)+1);

**if** ( BalMin > W->Bal) BalMin = W->Bal;

cout << W->Key << '('<< W->Bal << ')' << endl;

PrintTreeBal (W->Left,l+1);

}

}

**void** Tree::FibonTree1 (**int** k, Node\*\* T)

**// Построение дерева Фибоначчи порядка k с незаполненными**

**// полями Key узлов.**

{

**if** ( k==0) (\*T) = **NULL**;

**else**

**if** ( k==1 )

{

(\*T) = **new** (Node);

(\*T)->Left = (\*T)->Right = **NULL**;

}

**else**

{

(\*T) = **new** (Node);

FibonTree1 (k-1,&((\*T)->Left));

FibonTree1 (k-2,&((\*T)->Right));

}

}

**void** Tree::FibonTree2 (Node\*\* T,**int**\* i)

**// Заполнение поля Key узлов дерева Фибоначчи.**

{

**if** ( (\*T)!=**NULL** )

{

FibonTree2 (&((\*T)->Left),i);

(\*T)->Key = (\*i); (\*i)++;

FibonTree2 (&((\*T)->Right),i);

}

}

**int** Tree::NodeCount (Node\* T)

**// Подсчет количества узлов в бинаpном деpеве T.**

{

**if** ( T==**NULL** ) **return** 0;

**else**

**if** (T->Left==**NULL** && T->Right==**NULL**) **return** 1;

**else**

**return** NodeCount(T->Right)+NodeCount(T->Left)+1;

}

**void** main()

{

Tree A;

**int** k;

cout << "Вводите k...";

cin >> k;

**//Построение дерева Фибоначчи.**

**int** i = 1; **// Инициализация самого левого ключа дерева.**

A.FibonTree1 (k,A.GetTree());

A.FibonTree2 (A.GetTree(),&i);

**//Вывод дерева Фибоначчи, корневых балансов и баланса дерева.**

A.PrintTreeBal (A.GetTree1(),0);

cout << "\nБаланс дерева Фибоначчи: " << A.GetBalMin();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din71_2.zip).

После нескольких экспериментов с программой из примера 2 можно высказать гипотезу о том, что ***деревья Фибоначчи принадлежат классу* WB[1/2]**.

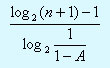
Попробуйте доказать этот факт самостоятельно!

Заметим, что в монографии [3,с.221] утверждается, что деревья Фибоначчи ***принадлежат классу* WB[1/3]**. Дело в том, что автор использует "чуть-чуть" другое дерево Фибоначчи, которое совпадает с нашим, если зеркально отобразить его относительно вертикальной оси, проходящей через корень!

Для оценки наибольшей высоты дерева Фибоначчи воспользуемся теоремой.

***Теорема [3, с.222]***.

Высота дерева **Tn** из класса **WB[A]** не превышает



Отсюда следует, что ***высота дерева Фибоначчи*** не превышает **log2(n+1)-1**.

(1) Кнут Д. Искусство программирования для ЭВМ. Т.3: Сортировка и поиск. - M.: Мир, 1978. - 844 с.

(2) Вирт H. Алгоритмы и структуры данных. - М.: Мир, 1989. - 360 с.

(3) Евстигнеев В.А. Применение теории графов в программировании. -М.: Hаука, 1985. - 352 с.

(4) Рейнгольд Э., Hивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

Со следующего шага мы начнем рассматривать ***алгоритмы балансировки***.

## Алгоритмы балансировки. Общие положения

На этом шаге мы рассмотрим ***общие операции, выполняемые при включении вершины в сбалансированное дерево***.

Рассмотрим теперь, что может произойти при включении в сбалансированное дерево новой вершины. Если у нас есть корень **r**, левое (**L**) и правое (**R**) поддеревья, то необходимо различать три возможных случая. Предположим, что включение в **L** новой вершины увеличит на 1 его высоту; тогда возможны три случая:

* сначала было **hL = hR**. После включения **L** и **R** станут разной высоты, но критерий сбалансированности не будет нарушен;
* сначала было **hL < hR**. После включения **L** и **R** станут равной высоты, то есть критерий сбалансированности даже улучшится;
* сначала было **hL > hR**. После включения критерий сбалансированности нарушится и дерево необходимо перестраивать.

Рассмотрим следующее дерево:

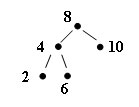


Рис.1. Пример дерева

Вершины с ключами 9 и 11 можно включить в дерево, не нарушив его сбалансированности: дерево с корнем 10 становится ***односторонним***, а дерево с корнем 8 - лучше сбалансированным. Однако включение значений 1, 3, 5 и 7 требует последующей балансировки.

Алгоритм включения и балансировки существенно зависит от того, каким способом хранится информация о сбалансированности дерева. В дальнейшем мы будем хранить в каждой вершине показатель сбалансированности. Таким образом, структура вершины АВЛ-дерева будет следующей:

**struct** node

{

**int** Key;

**int** Count;

**int** bal; **// Показатель балансированности вершины.**

node \*Left;

node \*Right;

};

Показатель сбалансированности вершины мы в дальнейшем будем интерпретировать как ***разность между высотой правого и левого поддерева***.

Таким образом, можно сформулировать алгоритм включения вершины в дерево:

* проход по дереву, чтобы убедиться, что включаемого значения в дереве нет;
* включение новой вершины и определение результирующего показателя сбалансированности;
* "отступление" по пути поиска и проверка в каждой вершине показателя сбалансированности. При необходимости - балансировка.

Со следующего шага мы начнем рассматривать ***алгоритмы балансировки***.

### Алгоритмы балансировки. Однократный LL-поворот

На этом шаге мы рассмотрим ***первый алгоритм поворота***.

На этом и следующем шагах мы приведем алгоритмы однократных поворотов, проиллюстрируем их конкретными примерами и в конце следующего шага сформулируем общее правило их использования.

Итак, приведем текст алгоритма:

p1 = (\*p).Left;

(\*p).Left = (\*p1).Right; (\*p1).Right = p;

(\*p).bal = 0; p = p1;

Рассмотрим его использование на конкретном примере.

Пусть дано следующее дерево:

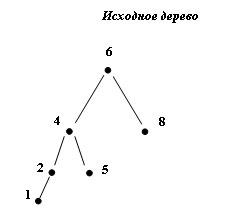


Рис.1. Исходное дерево

Иллюстрируем алгоритм с помощью схем "до и после" Д.Кнута:

1. Определяем адрес той вершины, которая станет корнем дерева:
2. p1 = (\*p).Left;

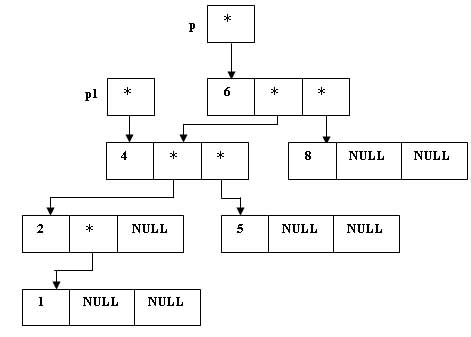


Рис.2. Сохранение адреса нового корня дерева

1. Переприкрепляем правое поддерево от "нового" корня, делая это поддерево левым поддеревом "старого" корня:
2. (\*p).Left = (\*p1).Right;

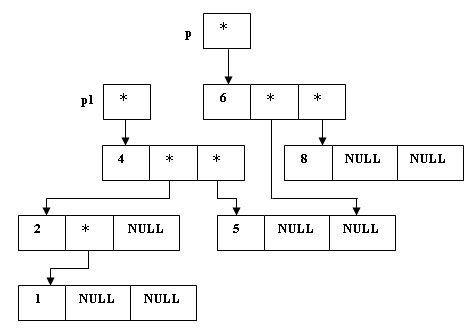


Рис.3. Переприкрепление

1. Определяем правое поддерево "нового" корня, как начинающееся со "старого" корня:
2. (\*p1).Right = p;

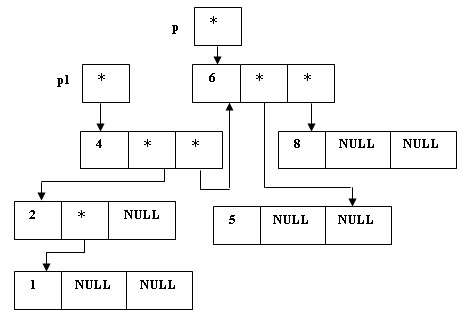


Рис.4. Определение правого поддерева "нового" корня

1. Изменяем значение указателя на корень дерева (**p**) и обнуляем значение сбалансированности:
2. (\*p).bal = 0; p = p1;

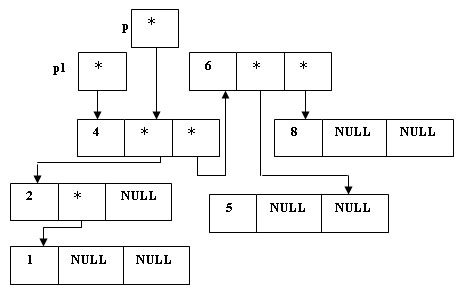


Рис.5. Установка начальных значений

В результате получилось следующее сбалансированное дерево:

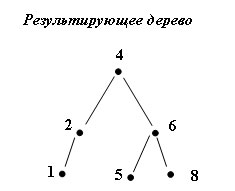


Рис.6. Результат балансировки

На следующем шаге мы разберем ***однократный* RR-*поворот***.

### Алгоритмы балансировки. Однократный RR-поворот

На этом шаге мы рассмотрим ***еще один однократный поворот***.

Этот алгоритм ничем не отличается от рассмотренного на предыдущем шаге, только поворот осуществляется в другую сторону.

Приведем текст алгоритма:

p1 = (\*p).Right;

(\*p).Right = (\*p1).Left;

(\*p1).Left = p;

(\*p).bal = 0; p = p1;

Рассмотрим его использование на конкретном примере.

Пусть дано следующее дерево:

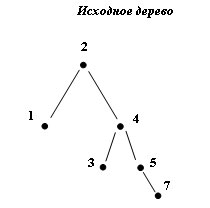


Рис.1. Исходное дерево

Иллюстрируем алгоритм с помощью схем "до и после" Д.Кнута:

1. Определяем адрес той вершины, которая станет корнем дерева:
2. p1 = (\*p).Right;

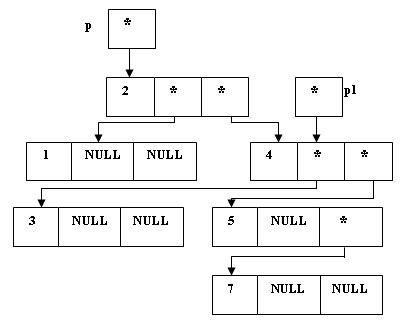


Рис.2. Сохранение адреса нового корня дерева

1. Переприкрепляем левое поддерево от "нового" корня, делая это поддерево правым поддеревом "старого" корня:
2. (\*p).Right = (\*p1).Left;

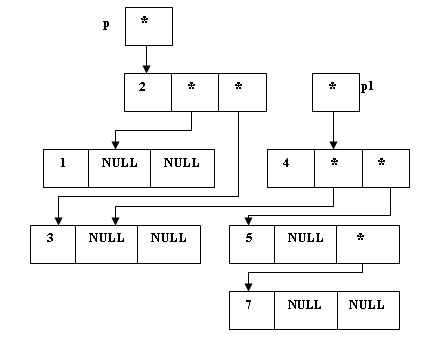


Рис.3. Переприкрепление

1. Определяем левое поддерево "нового" корня, как начинающееся со "старого" корня:
2. (\*p1).Left = p;

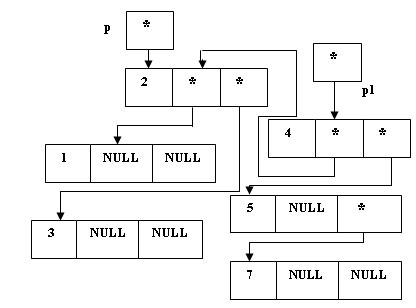


Рис.4. Определение левого поддерева "нового" корня

1. Изменяем значение указателя на корень дерева (**p**) и обнуляем значение сбалансированности:
2. (\*p).bal = 0; p = p1;

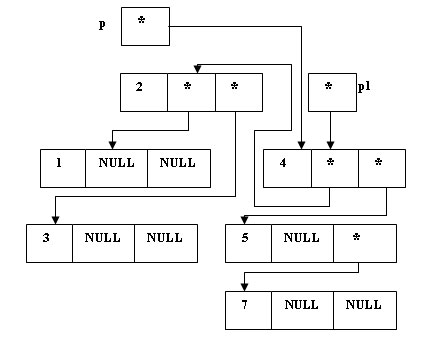


Рис.5. Установка начальных значений

В результате получилось следующее сбалансированное дерево:

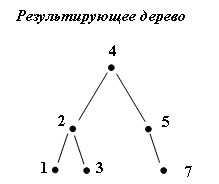


Рис.6. Результат балансировки

***Вывод.***

Если после вставки показатели сбалансированности вершин имеют одинаковый знак и отличаются только на единицу, то восстановить баланс дерева можно ***однократным поворотом*** (включая одно "переприкрепление" поддерева), при этом вставка не будет оказывать влияния на другие участки дерева.

Проиллюстрируем эту несбалансированность и ее устранение следующими рисунками:

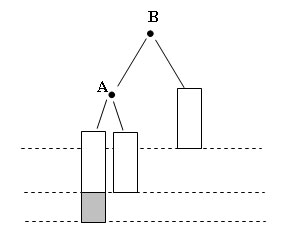


Рис.7. Несбалансированность (отмечен узел, вызвавший несбалансированность)

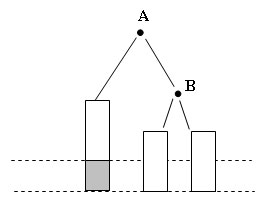


Рис.8. Несбалансированность устранена (отмечен узел, вызвавший несбалансированность)

Со следующего шага мы начнем рассматривать ***двухкратные повороты***.

### Алгоритмы балансировки. Двухкратный LR-поворот

На этом шаге мы рассмотрим ***первый двухкратный поворот***.

На этом и следующем шагах мы приведем алгоритмы двухкратных поворотов, проиллюстрируем их конкретными примерами и в конце следующего шага сформулируем общее правило их использования.

Итак, приведем текст алгоритма:

p1 = (\*p).Left; p2 = (\*p1).Right;

(\*p1).Right = (\*p2).Left; (\*p2).Left = p1;

(\*p).Left = (\*p2).Right; (\*p2).Right = \*p;

p = p2;

Рассмотрим его использование на конкретном примере.

Пусть дано следующее дерево:

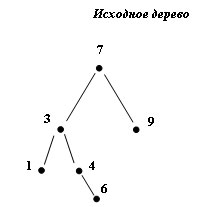


Рис.1. Исходное дерево

Иллюстрируем алгоритм с помощью схем "до и после" Д.Кнута:

1. Определим **p1** как указатель на левое поддерево, а **p2** как указатель на правое поддерево дерева **p1**:
2. p1 = (\*p).Left; p2 = (\*p1).Right;

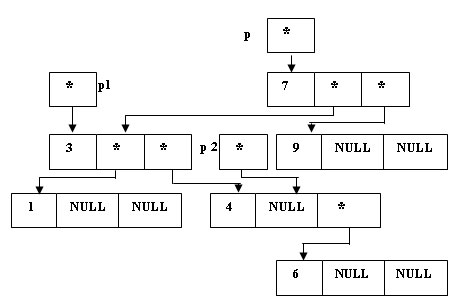


Рис.2. Определение **p1** и **p2**

1. Переприкрепляем левое поддерево дерева **p2** на место правого поддерева дерева **p1**:
2. (\*p1).Right = (\*p2).Left;

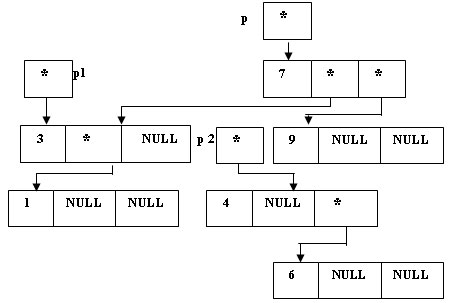


Рис.3. Переприкрепление

1. Определяем левое поддерево "нового" корня **p2**, как начинающееся с **p1**:
2. (\*p2).Left = p1;

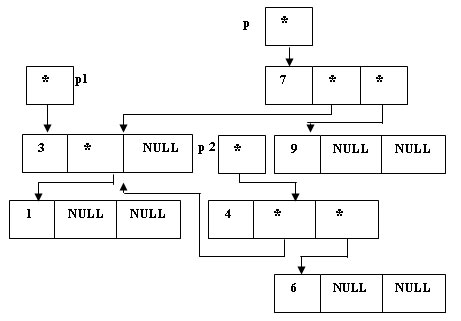


Рис.4. Определение левого поддерева "нового" корня

1. Переприкрепляем правое поддерево дерева **p2** на место левого поддерева "старого" корня:
2. (\*p).Left = (\*p2).Right;

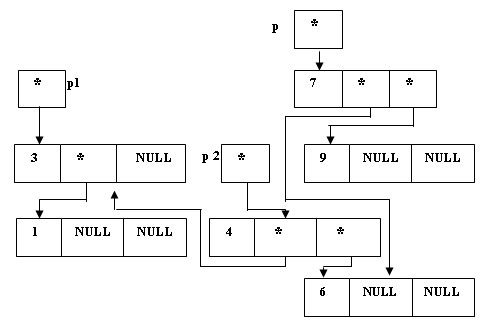


Рис.5. Переприкрепление

1. Определяем правое поддерево "нового" корня **p2**, как начинающееся со "старого" корня:
2. (\*p2).Right = \*p;

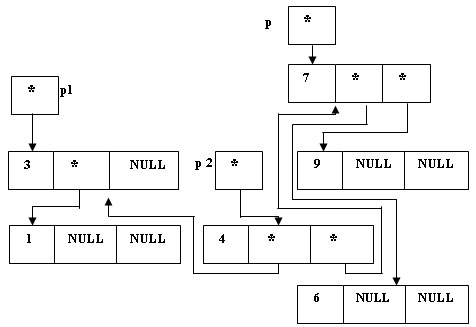


Рис.6. Определение правого поддерева "нового" корня

1. Изменяем значение указателя на корень дерева (**p**):
2. p = p2;

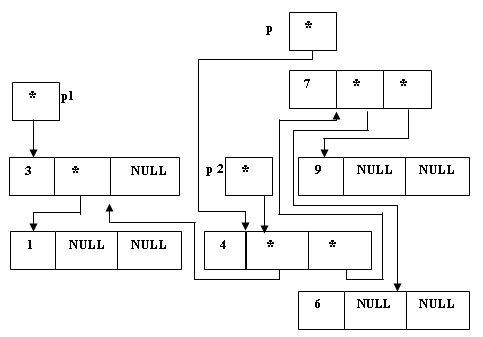


Рис.7. Установка начальных значений

В результате получилось следующее сбалансированное дерево:

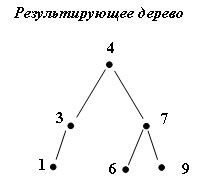


Рис.8. Результат балансировки

На следующем шаге мы рассмотрим ***второй двухкратный поворот***.

### Алгоритмы балансировки. Двухкратный RL-поворот

На этом шаге мы рассмотрим ***еще один двухкратный поворот***.

Этот алгоритм ничем не отличается от рассмотренного на предыдущем шаге, только поворот осуществляется в другую сторону.

Приведем текст алгоритма:

p1 =(\*p).Right; p2 = (\*p1).Left;

(\*p1).Left = (\*p2). Right; (\*p2). Right = p1;

(\*p).Right = (\*p2). Left; (\*p2). Left = \*p;

p = p2;

Рассмотрим его использование на конкретном примере.

Пусть дано следующее дерево:

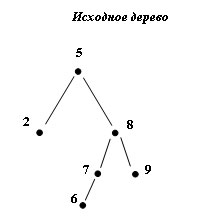


Рис.1. Исходное дерево

Иллюстрируем алгоритм с помощью схем "до и после" Д.Кнута:

1. Определим **p1** как указатель на правое поддерево, а **p2** как указатель на левое поддерево дерева **p1**:
2. p1 = (\*p).Right; p2 = (\*p1).Left;

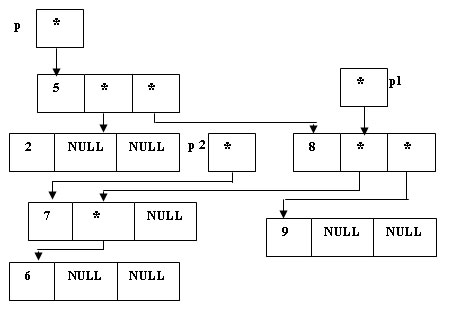


Рис.2. Определение **p1** и **p2**

1. Переприкрепляем правое поддерево дерева **p2** на место левого поддерева дерева **p1**:
2. (\*p1).Left = (\*p2). Right;

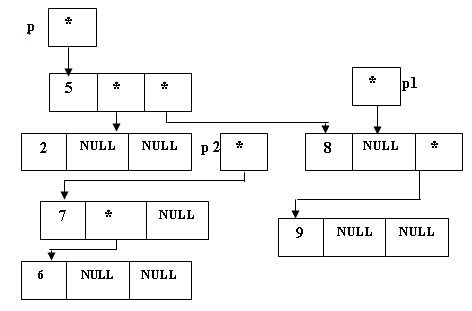


Рис.3. Переприкрепление

1. Определяем правое поддерево "нового" корня **p2**, как начинающееся с **p1**:
2. (\*p2). Right = p1;

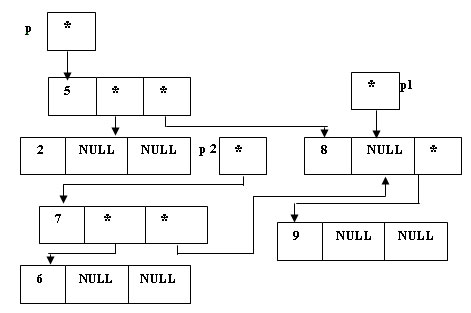


Рис.4. Определение правого поддерева "нового" корня

1. Переприкрепляем левое поддерево дерева **p2** на место правого поддерева "старого" корня:
2. (\*p).Right = (\*p2). Left;

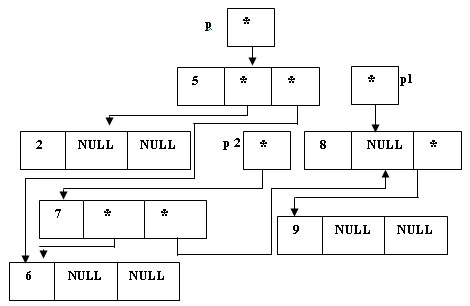


Рис.5. Переприкрепление

1. Определяем левое поддерево "нового" корня **p2**, как начинающееся со "старого" корня:
2. (\*p2). Left = \*p;

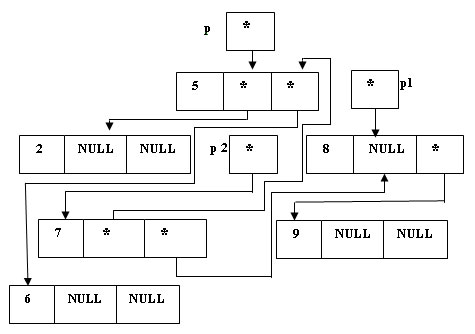


Рис.6. Определение правого поддерева "нового" корня

1. Изменяем значение указателя на корень дерева (**p**):
2. p = p2;

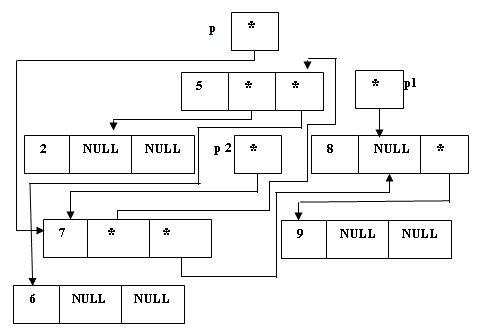


Рис.7. Установка начальных значений

В результате получилось следующее сбалансированное дерево:

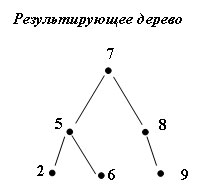


Рис.8. Результат балансировки

***Вывод.***

Если после вставки показатели сбалансированности имеют разный знак, то можно восстановить баланс дерева ***двухкратными поворотами трех вершин***. В этом случае вставка также не оказывает влияния на другие участки дерева.

Проиллюстрируем эту несбалансированность и ее устранение следующими рисунками:

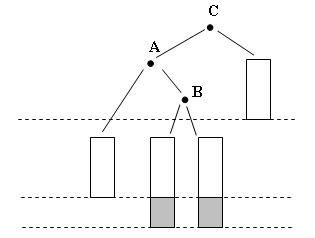


Рис.9. Несбалансированность (отмечены узлы, вызвавшие несбалансированность)

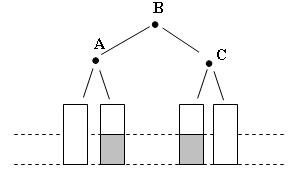


Рис.10. Несбалансированность устранена (отмечены узлы, вызвавшие несбалансированность)

Итак, все случаи, в которых после вставки необходима дополнительная балансировка для сохранения свойств АВЛ-дерева, ограничиваются разобранными случаями и случаями зеркального отражения этих структур.

На следующем шаге мы разберем ***алгоритм построения АВЛ-дерева***.

## Построение АВЛ-дерева

На этом шаге мы приведем ***программу построения АВЛ-дерева***.

Прежде, чем рассматривать программу построения АВЛ-дерева, приведем функцию поиска вершины в АВЛ-дереве с включением вершины в АВЛ-дерево в случае ее отсутствия:

**void** Search (**int** x, node \*\*p)

**// x - ключ вершины, помещаемой в АВЛ-дерево.**

**// \*p - указатель на корень АВЛ-дерева.**

**// h - флаг, сигнализирующий об увеличении высоты поддерева:**

**// TRUE - высота поддерева увеличилась,**

**// FALSE - высота поддерева не увеличилась.**

**// При первом обращении к функции Search() h=FALSE.**

{

node \*p1, \*p2;

h = FALSE;

**if** (\*p==**NULL**)

{ **// Вершины в дереве нет; включить ее...**

\*p = **new**(node);

h = TRUE; (\*\*p).Key = x;

(\*\*p).Count = 1; (\*\*p).Left = (\*\*p).Right = **NULL**;

(\*\*p).bal = 0; **// Вершине присвоили нулевой баланс.**

}

**else**

**if** (x<=(\*\*p).Key)

{

Search (x,&((\*\*p).Left)); **// Вершина уже включена в дерево.**

**if** (h==TRUE)

**// Если высота поддерева увеличилась,**

**// то выросла левая дуга.**

**switch** ((\*\*p).bal)

{ **case** 1: (\*\*p).bal = 0; h = FALSE; **break**;

**// Предыдущая несбалансированность уравновесилась.**

**case** 0: (\*\*p).bal = -1; **break**; **// Вес "склонился" влево.**

**case** -1:

**//Балансировка.**

p1 = (\*\*p).Left;

**if** ((\*p1).bal==-1)

{**//Однократный LL-поворот.**

(\*\*p).Left = (\*p1).Right;

(\*p1).Right = \*p;

(\*\*p).bal = 0; \*p = p1;

}

**else**

{**//Двукратный LR-поворот.**

p2 = (\*p1).Right;

(\*p1).Right = (\*p2).Left;

(\*p2).Left = p1;

(\*\*p).Left = (\*p2).Right;

(\*p2).Right = \*p;

**//Пересчет баланса вершины с указателем p.**

**if** ((\*p2).bal==-1) (\*\*p).bal = 1;

**else** (\*\*p).bal = 0;

**// Пересчет баланса вершины с указателем p1.**

**if** ((\*p2).bal==1) (\*p1).bal = -1;

**else** (\*p1).bal = 0;

\*p = p2;

}

(\*\*p).bal = 0; h = FALSE;

**break**;

}

}

**else** **//... иначе выросла правая дуга.**

**if** (x>(\*\*p).Key)

{

Search (x,&((\*\*p).Right));

**// Вершина уже включена в дерево.**

**if** (h==TRUE)

**// Если высота поддерева увеличилась,**

**// то выросла правая дуга.**

**switch** ((\*\*p).bal)

{ **case** -1: (\*\*p).bal = 0; h = FALSE; **break**;

**case** 0: (\*\*p).bal = 1; **break**;

**case** 1:

**//Балансировка.**

p1 = (\*\*p).Right;

**if** ((\*p1).bal==1)

{ **//Однократный RR-поворот.**

(\*\*p).Right = (\*p1).Left;

(\*p1).Left = \*p; (\*\*p).bal = 0; \*p = p1;

}

**else**

{ **//Двухкратный RL-поворот.**

p2 = (\*p1).Left; (\*p1).Left = (\*p2).Right;

(\*p2).Right = p1; (\*\*p).Right = (\*p2).Left;

(\*p2).Left = \*p;

**// Пересчет баланса вершины с указателем p.**

**if** ((\*p2).bal==1) (\*\*p).bal = -1;

**else** (\*\*p).bal = 0;

**//Пересчет баланса вершины с указателем p1.**

**if** ((\*p2).bal==-1) (\*p1).bal = 1;

**else** (\*p1).bal = 0; \*p = p2;

}

(\*\*p).bal = 0; h = FALSE; **break**;

}

}

}

Пример. Построение АВЛ-дерева с помощью функции поиска с включением.

**#include**<iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**struct** node

{

**int** Key;

**int** Count;

**int** bal;

node \*Left;

node \*Right;

};

**class** TREE {

**private**:

**int** h;

node \*Tree;

**public**:

TREE () { Tree=**NULL**; h=FALSE;}

**void** Search (**int**, node \*\*);

**void** Vyvod (node \*\*, **int**);

node\*\* GetTree() {**return** &Tree;}

};

**void** main ()

{

TREE A;

**int** el,i;

**int** n;

cout<<"Количество вершин в дереве: ";

cin>>n;

cout<<"Информационные поля вершин дерева: \n";

**for** (i=1; i<=n; i++)

{ cin>>el; A.Search (el,A.GetTree());}

cout<<"АВЛ-дерево:\n"; A.Vyvod (A.GetTree(),0);

}

**void** TREE::Search (**int** x, node \*\*p)

**// x - ключ вершины, помещаемой в АВЛ-дерево.**

**// \*p - указатель на корень АВЛ-дерева.**

**// h - флаг, сигнализирующий об увеличении высоты поддерева:**

**// TRUE - высота поддерева увеличилась,**

**// FALSE - высота поддерева не увеличилась.**

**// При первом обращении к функции Search() h=FALSE.**

{

node \*p1, \*p2;

h = FALSE;

**if** (\*p==**NULL**)

{ \*p = **new**(node);

h = TRUE; (\*\*p).Key = x;

(\*\*p).Count = 1; (\*\*p).Left = (\*\*p).Right = **NULL**;

(\*\*p).bal = 0; }

**else**

**if** (x<=(\*\*p).Key)

{ Search (x,&((\*\*p).Left));

**if** (h==TRUE)

**switch** ((\*\*p).bal)

{

**case** 1 : (\*\*p).bal = 0; h = FALSE; **break**;

**case** 0 : (\*\*p).bal = -1; **break**;

**case** -1:

p1 = (\*\*p).Left;

**if** ((\*p1).bal==-1)

{ (\*\*p).Left = (\*p1).Right;

(\*p1).Right = \*p;

(\*\*p).bal = 0; \*p = p1; }

**else** {

p2 = (\*p1).Right;

(\*p1).Right = (\*p2).Left;

(\*p2).Left = p1;

(\*\*p).Left = (\*p2).Right;

(\*p2).Right = \*p;

**if** ((\*p2).bal==-1) (\*\*p).bal = 1;

**else** (\*\*p).bal = 0;

**if** ((\*p2).bal==1) (\*p1).bal = -1;

**else** (\*p1).bal = 0;

\*p = p2;

}

(\*\*p).bal = 0; h = FALSE;

**break**;

}

}

**else**

**if** (x>(\*\*p).Key)

{ Search (x,&((\*\*p).Right));

**if** (h==TRUE)

**switch** ((\*\*p).bal)

{

**case** -1: (\*\*p).bal = 0; h = FALSE; **break**;

**case** 0: (\*\*p).bal = 1; **break**;

**case** 1:

p1 = (\*\*p).Right;

**if** ((\*p1).bal==1)

{ (\*\*p).Right = (\*p1).Left;

(\*p1).Left = \*p; (\*\*p).bal = 0; \*p = p1;

}

**else**

{ p2 = (\*p1).Left; (\*p1).Left = (\*p2).Right;

(\*p2).Right = p1; (\*\*p).Right = (\*p2).Left;

(\*p2).Left = \*p;

**if** ((\*p2).bal==1) (\*\*p).bal = -1;

**else** (\*\*p).bal = 0;

**if** ((\*p2).bal==-1) (\*p1).bal = 1;

**else** (\*p1).bal = 0;

\*p = p2;

}

(\*\*p).bal = 0; h = FALSE; **break**;

}

}

}

**void** TREE::Vyvod (node \*\*w,**int** l)

**//Изображение дерева w на экране дисплея**

**// (рекурсивный алгоритм).**

**//\*w - указатель на корень дерева.**

{

**int** i;

**if** (\*w!=**NULL**)

{

Vyvod (&((\*\*w).Right),l+1);

**for** (i=1; i<=l; i++) cout<<" ";

cout<<(\*\*w).Key<<endl;

Vyvod (&((\*\*w).Left),l+1);

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din77_1.zip).

Приведем пример построения АВЛ-дерева.

Пусть на "вход" функции **Search()** последовательно поступают целые числа 4,5,7,2,1,3,6. Изобразим процесс "роста" АВЛ-дерева (в скобках указан показатель сбалансированности некоторых вершин) [1,с.256]:

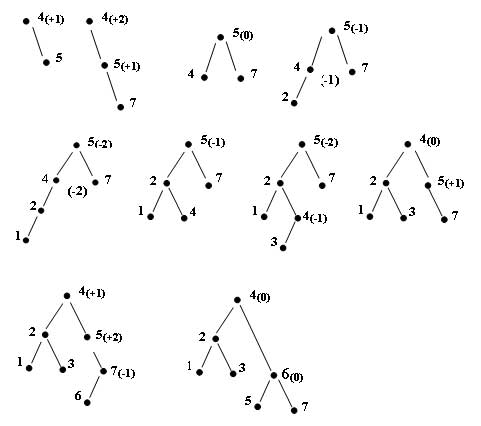


Рис.1. Рост АВЛ-дерева

Тщательно подобранный пример, показывающий ситуацию: как можно больше поворотов при минимальном числе включений!

Оценим эффективность поиска со вставкой, считая, что все вставляемые ключи поступают в случайном порядке. Для этого потребуется ответить на следующие вопросы:

1. Как зависит математическое ожидание значения высоты от общего числа вершин n в дереве?
2. Какова вероятность возникновения случаев, не требующих дополнительной балансировки, случаев, требующих однократного поворота, и случаев, требующих двухкратного поворота, соответственно?
3. Как зависит число операций при вставке одной вершины от длины пути, ведущего из внешней вершины вверх и от числа вершин n в дереве?

До сих пор не удалось дать точных ответов на эти вопросы. Однако сочетание некоторых теоретических рассуждений и эмпирических результатов позволяет сформулировать следующие утверждения.

1. Математическое ожидание значения высоты при больших **n** близко к значению **h = log2n + c**. H.Вирт [1, с.255] пишет: "Математический анализ этого сложного алгоритма пока не произведен. Эмпирические проверки оправдывают предположение, что ожидаемая высота сбалансированного дерева ... равна **h=logn+c**, где **c** - малая константа (**c ~ 0.25**). Это значит, что на практике АВЛ-сбалансированные деревья ведут себя так же, как идеально сбалансированные деревья, хотя с ними намного легче работать".
2. Вероятность того, что при вставке не потребуется дополнительная балансировка, потребуется однократный поворот или двукратный поворот, близка к значениям 2/3, 1/6 и 1/6, соответственно. H.Вирт [1, с.255] продолжает: "Эмпирически можно предположить, что в среднем балансировка необходима приблизительно один раз на каждые два включения. При этом однократный и двухкратный поворот одинаково вероятны!"
3. Среднее число сравнений при вставке **n**-го ключа в дерево выражается формулой **alog2n+b** (**a** и **b** - постоянные).

Мы не будем приводить алгоритм удаления вершины из АВЛ-дерева [1] и опустим теоретические выкладки, которые доказывают утверждение, что трудоемкость удаления вершин из АВЛ-дерева также зависит от числа вершин в дереве как **log2n**.

Таким образом, АВЛ-дерево представляет собой структуру, для которой любая операция: поиск, вставка и удаление вершины с заданным ключом имеет временную сложность **O(log2n)**.

(1)Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

Со следующего шага мы начнем знакомиться с ***графами***.

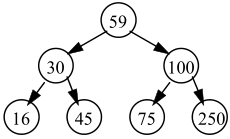
## Поиск с помощью дерева

### Как быстрее искать?

Деревья очень удобны для поиска в них информации. Однако **для быстрого поиска требуется предварительная подготовка – дерево надо построить специальным образом.**

Предположим, что существует массив данных и с каждым элементом связан ключ - число, по которому выполняется поиск. Пусть ключи для элементов таковы:

**59, 100, 75, 30, 16, 45, 250**



Для этих данных нам надо много раз проверять, есть ли среди ключей заданный ключ **x**, и если есть, то вывести всю связанную с этим элементом информацию.

Если данные организованы в виде массива (без сортировки), то для поиска в худшем случае надо сделать **n** сравнений элементов (сравнивая последовательно с каждым элементом пока не найдется нужный или пока не закончится массив).

Теперь предположим, что данные организованы в виде дерева, показанного на рисунке. Такое дерево (оно называется **дерево поиска**) обладает следующим важным свойством:

***Значения ключей всех вершин левого поддерева вершины x меньше ключа x, а значения ключей всех вершин правого поддерева x больше или равно ключу вершины x.***

Для поиска нужного элемента в таком дереве требуется не более 3 сравнений вместо 7 при поиске в списке или массиве, то есть поиск проходит значительно быстрее. С ростом количества элементов эффективность поиска по дереву растет.

### Построение дерева поиска

Как же, имея массив данных, построить такое дерево?

* Сравнить ключ очередного элемента массива с ключом корня.
* Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.
* Если текущее дерево пустое, создать новую вершину и включить в дерево.

01.wmf

Программа, приведенная ниже, реализует этот алгоритм:

void AddToTree(PNode &Tree, //указатель на корень (ссылка)

int data) //добавляемый ключ

{

if(! Tree){

Tree = new Node; //создать новый узел

Tree->key = data;

Tree->left = NULL;

Tree->right = NULL;

return;

}

if(data < Tree->key) //добавить в нужное поддерево

AddToTree ( Tree->left, data );

else

AddToTree ( Tree->right, data );

}

Важно, что указатель на корень дерева надо передавать по ссылке, так как он может измениться при создании новой вершины.

Надо заметить, что в результате работы этого алгоритма не всегда получается дерево минимальной высоты – все зависит от порядка выбора элементов. Для оптимизации поиска используют так называемые **сбалансированные** или **АВЛ-деревья** (деревья называют так в честь изобретателей этого метода Г.М. Адельсона-Вельского и Е.М. Ландиса) деревья, у которых для любой вершины высоты левого и правого поддеревьев отличаются не более, чем на 1. Добавление в них нового элемента иногда сопровождается некоторой перестройкой дерева.

### Поиск по дереву

Теперь, когда дерево сортировки построено, очень легко искать элемент с заданным ключом. Сначала проверяем ключ корня, если он равен искомому, то нашли. Если он меньше искомого, ищем в левом поддереве корня, если больше – то в правом. Приведенная функция возвращает адрес нужной вершины, если поиск успешный, и **NULL**, если требуемый элемент не найден.

PNode Search(PNode Tree, int what)

{

if(! Tree)//ключ не найден

return NULL;

if(what == Tree->key)//ключ найден!

return Tree;

if(what < Tree->key)//искать в поддеревьях

return Search(Tree->left, what);

else

return Search(Tree->right, what);

}

### Сортировка с помощью дерева поиска

Если дерево поиска построено, очень просто вывести отсортированные данные. действительно, обход типа **ЛКП (левое поддерево – корень – правое поддерево)** даст ключи в порядке возрастания, а обход типа **ПКЛ (правое поддерево – корень – левое поддерево)** – в порядке убывания.

### Поиск одинаковых элементов

Приведенный алгоритм можно модифицировать так, чтобы быстро искать одинаковые элементы в массиве чисел. Конечно, можно перебрать все элементы массива и сравнить каждый со всеми остальными. Однако для этого требуется очень большое число сравнений. С помощью двоичного дерева можно значительно ускорить поиск. Для этого надо в структуру вершины включить еще одно поле – счетчик найденных дубликатов **count**.

struct Node

{

int key;

int count; //счетчик дубликатов

Node \*left, \*right;

};

При создании узла в счетчик записывается единица (найден один элемент). Поиск дубликатов происходит по следующему алгоритму:

* Сравнить ключ очередного элемента массива с ключом корня.
* Если ключ нового элемента равен ключу корня, то увеличить счетчик корня и стоп.
* Если ключ нового элемента меньше, чем у корня, включить его в левое поддерево, если больше или равен – в правое.
* Если текущее дерево пустое, создать новую вершину (со значением счетчика 1) и включить в дерево.

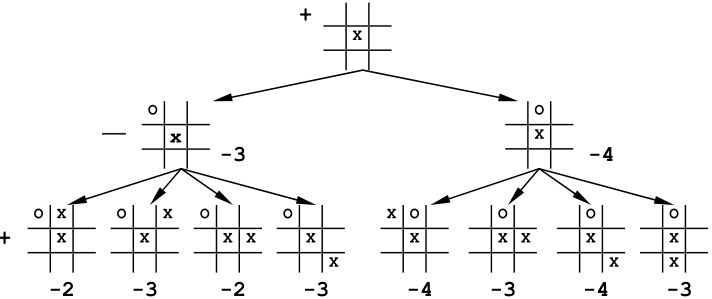
## Дерево игр

Одно из применений деревьев - игры с компьютером. Рассмотрим самый простой пример – игру в крестики-нолики на поле 3 на 3.

Программа должны анализировать позицию и находить лучший ход. Для этого нужно определить **оценочную функцию**, которая получая позицию на доске и указание, чем играет игрок (крестики или нолики) возвращает число – оценку позиции. Чем она выше, тем более выгодна эта позиция для игрока. Примером такой функции может служить сумма строк, столбцов и диагоналей, которые может занять игрок минус такая же сумма для его противника.

Однако, в этой ситуации программа не ведет просчет вперед и не оценивает позиции, которые могут возникнуть из текущей. Это недостаточно для предсказания исхода игры. Хотя для крестиков-ноликов можно перебрать все варианты и найти выигрышную позицию, большинство игр слишком сложно, чтобы допускать полный перебор.

Выбор хода может быть существенно улучшен, если просматривать на несколько ходов вперед. **Уровнем просмотра** называется число будущих рассматриваемых ходов. Начиная с любой позиции можно построить дерево возможных позиций, получающихся после каждого хода игры. Для крестиков-ноликов ниже приведено дерево с уровнем просмотра 3 для того случая, когда крестики сделали первый ход в центр доски.



Обозначим игрока, который ходит в корневой позиции (в данном случае – нолики) знаком «плюс», а его соперника – знаком «минус». Попытаемся найти лучший ход для игрока «плюс» в этой позиции. Пусть все варианты следующих ходов были оценены для игрока «плюс». Он должен выбрать такой, в котором оценка максимальная для него.

С другой стороны, как только игрок «плюс» сделает свой ход, игрок «минус» из всех возможных ходов сделает такой, чтобы его оценка с позиции игрока «плюс» была минимальной. Поэтому значение минусового узла для игрока «плюс» равно минимальному из значений сыновей этого узла. Это означает, что на каждом шаге соперники делают наилучшие возможные ходы.

Для того, чтобы выбрать оптимальный ход в корне дерева, надо оценить позицию в его листьях. После этого каждому плюсовому узлу присваивается **максимальное** из значений его сыновей, а каждому минусовому – **минимальное**. Такой метод называется методом **минимакса**, так как по мере продвижения вверх используются попеременно функции максимума и минимума. Общая идея метода состоит в том, чтобы выбрать лучший ход на случай худших (для нас) действий противника. Таким образом, лучшим для ноликов в корневой позиции будет ход в угол.