**Динамическая структура «дерево»**

**Основные понятия**

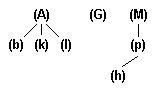
Структуры данных типа “дерево” исключительно широко используются в программной индустрии. В отличие от списковых структур деревья относятся к **нелинейным** структурам. Любое дерево состоит из элементов – **узлов** или **вершин**, которые по определенным правилам связаны друг с другом рёбрами. В списковых структурах за текущей вершиной (если она не последняя) всегда следует только одна вершина, тогда как в древовидных структурах таких вершин может быть **несколько**. Математически дерево рассматривается как частный случай графа, в котором отсутствуют замкнутые пути (циклы).

**Дерево - это** граф, который характеризуется следующими свойствами:

1. Существует единственный элемент (узел или вершина), на который не ссылается никакой другой элемент - и который называется **корнем** (рис. - *A,G,M - корни*).

2. Начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры.

3. На каждый элемент, кроме корня, имеется единственная ссылка, т.е. каждый элемент адресуется единственным указателем.

****

****

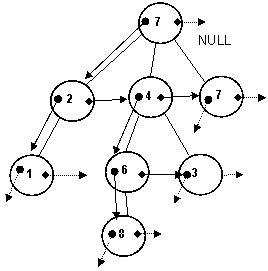
Рис 1. Дерево и лес

Каждый элемент данных называется **вершиной или узлом**  дерева (**node**), а любой фрагмент дерева называется **поддеревом (subtree-b,c,d**). **Вершина, к которой не присоединены поддеревья, называется заключительным узлом (terminalnode) или листом (leaf-k,l).Высота (height)** дерева равняется максимальному количеству уровней от корня до листа. При использовании деревьев часто встречаются такие понятия как путь между начальной и конечной вершиной (последовательность проходимых ребер или вершин).

Для неформального описания корневых деревьев часто используется генеалогическая терминология, согласно которой каждая ветвь отражает отношение **потомок-предок** между инцидентными ей узлами**. Корень дерева-**это узел, который не имеет предка. Узлы дерева, которые не имеют потомков называются**листьями**. Остальные узлы (не листья и не корень) называются разветвлениями.

Рисунок иллюстрирует классическое изображение корневого дерева средствами теории графов, где вершины и ребра графа представляют узлы и ветви дерева.

Здесь заглавные буквы латинского алфавита обозначают узлы, а строчные- ветви корневого дерева. Конфигурация ветвей этого корневого дерева такова, что узел**А** является корнем, узлы **В С и D- разветвлениями**, а узлы **E, F, G, H, и K - листьями.**



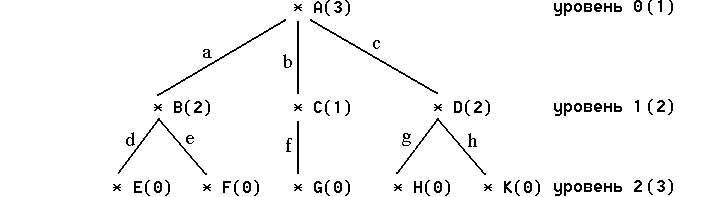


Рис.3. Изображение корневогоРис.4. Представление дерева на основе

дерева в теории графов ветвящегося списка

Важными метрическими характеристиками корневого дерева является **степень и уровень узла**. **Степенью узла корневого дерева считается число поддеревьев, для которых он является корнем.** Для приведенного примера корневого дерева: корень**А**имеет степень **3**, степени разветвлений **B** и **D** - равны **2**, а степень разветвления **С** равна **1**. Степени остальных узлов равны 0, потому что они являются листьями, т. е. не имеют поддеревьев. Уровень узла корневого дерева определяется длиной пути, образованного чередующейся последовательностью узлов и ветвей, который соединяет его с корнем. Длина пути измеряется числом узлов в нем. Для рассмотренного примера корень**А** имеет уровень **1**, разветвления   **B, C и D** имеют уровень **2**, а листья **E, F, G, H и K** - уровень **3**. При измерении длины пути числом ветвей в нем, указанные уровни узлов надо уменьшить на 1.

Узел**С** называется **предком** (или **отцом**), а узлы **k** и **l** называются **потомками(наследниками или сыновьями)** их соответственно между собой называют **братьями**. Причем левый потомок является старшим, а правый - младшим. Число поддеревьев данной вершины называется **степенью** этой вершины. ( В данном примере **k** имеет **2** поддерева, следовательно**степень** вершины **k** равна 2).

Если из дерева убрать корень и ребра, соединяющие корень с вершинами первого яруса, то получится некоторое множество несвязанных деревьев. Множество несвязанных деревьев называется **лесом**.

**Классификация деревьев**

**Типы деревьев**:

* двоичные (бинарные),
* ориентированные,
* упорядоченные,
* мультивариантные (дерево Байера).

***Классификацию деревьев можно провести по разным признакам***.

1. По числу возможных потомков у вершин различают **двоичные (бинарные**) или **недвоичные (сильноветвящиеся)** деревья.  
Двоичное дерево: каждая вершина может иметь не более двух потомков.  
Недвоичное дерево: вершины могут иметь любое число потомков.

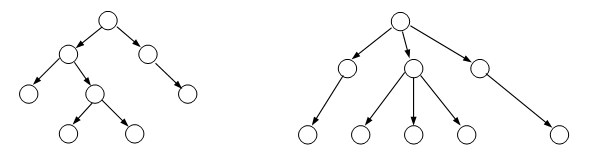


Рис.2.Двоичное и сильно ветвящееся деревья

1. Если в дереве важен порядок следования потомков, то такие деревья называют **упорядоченными.** Для них вводится понятие левый и правый потомок (для двоичных деревьев) или более левый/правый (для недвоичных деревьев). В этом смысле два следующих простейших упорядоченных дерева с**одинаковыми**лементами считаются **разными**:
2. 

При рассмотрении дерева как структуры данных необходимо четко понимать следующие два момента:

1. Все вершины дерева, рассматриваемые как переменные языка программирования, должны быть одного и того же типа, более того –структурами с некоторым информационным наполнением и необходимым количеством связующих полей
2. В силу естественной логической разветвленности деревьев (в этом весь их смысл!) и отсутствия единого правила выстраивания вершин в порядке друг за другом, их логическая организация не совпадает с физическим размещением вершин дерева в памяти. При работе с деревьями можно допустить, что в памяти они существуют в том же виде, что и на бумаге. Но помните, что дерево — всего лишь способ логической организации данных в памяти, а память линейна.

#### Логическое представление и изображение деревьев.

Имеется ряд способов графического изображения деревьев.

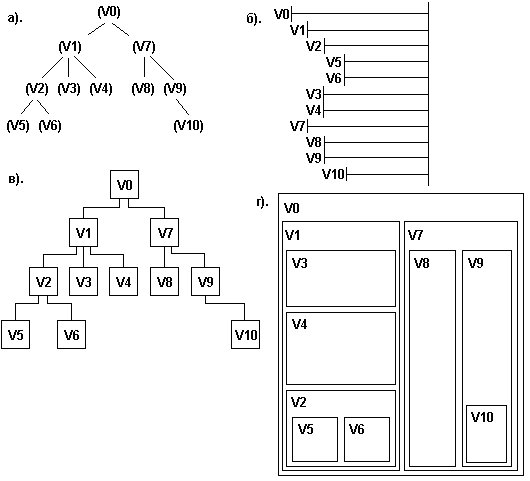
**1-ый способ** заключается в использовании для изображения поддеревьев известного метода **диаграмм Венна**,

**2-ой** - метод **вложенных скобок**,

**3-ий способ** - это способ, применяемый при составлении оглавлений книг. Последний способ, базирующийся на формате с нумерацией уровней, сходен с методами, используемыми в языках программирования. При применении этого формата каждой вершине приписывается числовой номер, который должен быть меньше номеров, приписанных корневым вершинам присоединенных к ней поддеревьев. Отметим, что корневые вершины всех поддереьев данной вершины должны иметь один и тот же номер.

**4-ий способ**-**граф**

**Метод вложенных скобок(V0(V1(V2(V5)(V6))(V3)(V4))(V7(V8)(V9(V10))))**



**Рис.6. Представление дерева : а)- исходное дерево, б)- оглавление книг, в)- граф, г)- диаграмма Венна**

Все эти представления демонстрируют одну и ту же структуру и поэтому эквивалентны. С помощью графа можно наглядно представить разветвляющиеся связи, которые по понятным причинам привели к общеупотребительному термину "дерево".

Деревья нужны для описания любой структуры с иерархией. **Традиционные примеры таких структур**: генеалогические деревья, десятичная классификация книг в библиотеках, иерархия должностей в организации, алгебраическое выражение, включающее операции, для которых предписаны определенные правила приоритета.

**Дерево - динамическая нелинейная структура данных, каждый элемент которой содержит собственно информацию (или ссылку на то место в памяти ЭВМ, где хранится информация) и ссылки на несколько (не менее двух) других таких же элементов**

Из теории графов вытекает определение, которое позволяет интерпретировать корневое дерево как рекурсивный объект, который содержит сам себя и определяется с помощью себя самого (т. е. дерево определяется в терминах дерева). Определение корневого дерева как определение любого рекурсивного объекта содержит базисную и рекурсивную части. Рекурсивное определение корневого дерева позволяет более простым способом формализовать его структуру и алгоритмы обработки.

**Дерево как абстрактная структура данных должна включать следующий набор операций:**

* **добавление новой вершины**
* **удаление некоторой вершины**
* **обход всех вершин дерева**
* **поиск заданной вершины**

**Бинарное дерево**

Двоичные деревья образуют широко распространенный в программировании класс деревьев. Двоичные деревья (ДД) используются наиболее часто и поэтому представляют наибольший практический интерес. Каждая вершина ДД должна иметь **два связующих поля** для адресации двух своих возможных потомков.

В некотором смысле двоичное дерево является особым видом связанного списка. Элементы можно вставлять, удалять и извлекать в любом порядке. Кроме того, операция извлечения не является разрушающей. Несмотря на то, что деревья легко представить в воображении, в теории программирования с ними связан ряд сложных задач. В данном разделе деревья затрагиваются лишь поверхностно.

**Бинарное (двоичное) дерево (binarytree)** - это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве содержатся только ключи, имеющие значения, меньшие, чем значение данного узла, а в правом поддереве содержатся только ключи, имеющие значения, большие, чем значение данного узла.  
Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева.

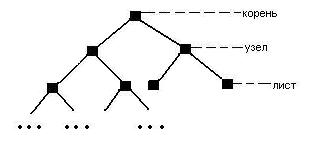


Рис.3. Схематичное изображение бинарного дерева

Бинарное дерево может представлять собой пустое множество.  
Бинарное дерево может выродиться в список:  


Рис.4.Вырожденное дерево

Дерево **T** называется полным, если  все его уровни, за исключением возможно последнего, имеют максимальное число узлов, и если все узлы в последнем уровне имеют хотя бы левого потомка. Глубина **D** полного дерева с **n** узлами равна

**Dn= log2 n +1.** Это относительно малое число. Например, если дерево имеет

n= 1 000000 узлов, то его глубина Dn =21.  
Двоичное дерево поиска *может быть либо пустым, либо оно обладает таким свойством, что корневой элемент имеет большее значение узла, чем любой элемент в левом поддереве, и меньшее или равное, чем элементы в правом поддереве. Указанное свойство называется* характеристическим свойством двоичного дерева *поиска и выполняется для любого узла такого дерева, включая корень. Далее будем рассматривать только двоичные деревья поиска. Такое название двоичные деревья поиска получили по той причине, что* скорость поиска в них примерно такая же, что и в отсортированных массивах: O(n) = C • log2n (в худшем случае O(n) = n).

ДД можно реализовать двумя способами:

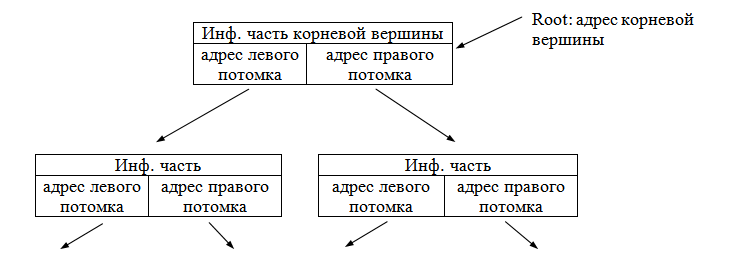
* на основе массива записей с использованием индексных указателей
* на базе механизма динамического распределения памяти с сохранением в каждой вершине адресов ее потомков (если они есть)

Второй способ является значительно более удобным и поэтому используется наиболее часто. В этом случае каждая вершина описывается как запись, содержащая как минимум три поля: информационную составляющую и два ссылочных поля для адресации потомков:

**struct TREE {  
   intInfo;  
  TREE \*Right;  
  TREE \*Left;  
};**

Для обработки дерева достаточно знать адрес корневой вершины. Для хранения этого адреса надо ввести ссылочную переменную:  
**TREE \*Root;**

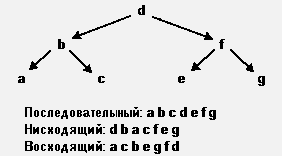
Тогда пустое дерево просто определяется установкой  переменной  **Root**  в  нулевое значение:  
**Root = NULL;**



Реализацию основных операций с ДД удобно начать с **процедуробхода**. Поскольку дерево является нелинейной структурой, то НЕ существует **единственной** схемы обхода дерева.

Способ упорядочивания дерева зависит от того, как к нему впоследствии будет осуществляться доступ. Процесс поочередного доступа к каждой вершине дерева называется *обходом (вершин) дерева* (treetraversal). Существует три способа прохождения дерева:

1. **Последовательный (***обход симметричным способом* ) **inorder** - дерево проходится, начиная с левой ветви вверх к корню, затем к правой ветви. (**a b c d e f g**)
2. **Нисходящий (***обход в прямом порядке, прямой обход, упорядоченный обход, обход сверху*, или *обход в ширину* )**preorder** - от корня к левой ветви, затем к правой.( **d b a c f e g**)
3. **Восходящий** (обход в *обратном порядке, обход в глубину, обратный обход, обход снизу*) **postorder** - проходится левая ветвь, затем правая, затем корень.(**a c b e g f d**)



При симметричном обходе обрабатывается сначала левое поддерево, затем корень, а затем правое поддерево. При прямом обходе обрабатывается сначала корень, затем левое поддерево, а потом правое. При обходе снизу сначала обрабатывается левое поддерево, затем правое и, наконец корень.

Последовательность доступа при каждом методе обхода показана ниже:

Симметричный обход **a b c d e f g**

Прямой обход **d b a c f e g**

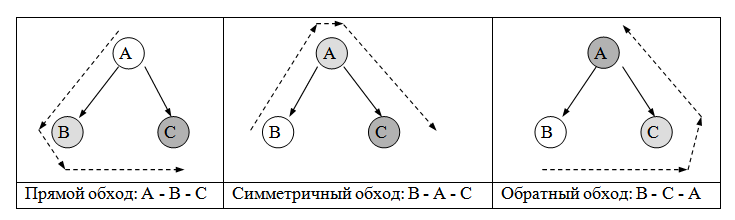
Обход снизу **a c b e g f d**

Несмотря на то, что дерево не должно быть обязательно упорядоченным, в большинстве задач используются именно такие деревья. Конечно, структура упорядоченного дерева зависит от способа его обхода. В оставшейся части данной главы предполагается симметричный обход. Поэтому упорядоченным двоичным деревом будет считаться такое дерево, в котором левое поддерево содержит вершины, меньшие или равные корню, а правое содержит вершины, большие корня.

Для объяснения каждого из этих правил удобно воспользоваться простейшим ДД из трех вершин. Обход всего дерева следует проводить за счет последовательного выделения в дереве подобных простейших поддеревьев и применением к каждому из них соответствующего правила обхода. Выделение начинается с корневой вершины.

Сами правила обхода носят рекурсивный характер и формулируются следующим образом:

1. **Обход в прямом направлении(нисходящий) preorder:**
   * обработать корневую вершину текущего поддерева
   * перейти к обработке левого поддерева таким же образом
   * обработать правое поддерево таким же образом
2. **Симметричный обход(последовательный) inorder:**
   * рекурсивно обработать левое поддерево текущего поддерева
   * обработать вершину текущего поддерева
   * рекурсивно обработать правое поддерево
3. **Обход в обратном направлении (восходящий) postorder:**
   * рекурсивно обработать левое поддерево текущего поддерева
   * рекурсивно обработать правое поддерево
   * затем – вершину текущего поддерева



**Каждый рекурсивный вызов отвечает за обработку своего текущего поддерева. П**осле полной обработки текущего поддерева происходит возврат к поддереву более высокого уровня, а для этого надо запоминать и в дальнейшем восстанавливать адрес корневой вершины этого поддерева. Рекурсивные вызовы позволяют выполнить это запоминание и восстановление автоматически, если описать адрес корневой вершины поддерева как формальный параметр рекурсивной функции.

Каждый рекурсивный вызов прежде всего должен проверить переданный ей адрес на **NULL**. Если этот адрес равен **NULL**, то очередное обрабатываемое поддерево является пустым и никакая его обработка не нужна, поэтому просто происходит возврат из рекурсивного вызова. В противном случае в соответствии с реализуемым правилом обхода производится  либо обработка вершины, либо рекурсивный вызов для обработки левого или правого поддерева. Обращение к рекурсивной процедуре для обработки левого потомка надо заменить помещением в стек адреса текущей корневой вершины и переходом к левому потомку этой вершины. Обработка правого потомка заключается в извлечении из стека адреса некоторой вершины и переходе к её правому потомку.

В принципе, достаточно легко реализовать **нерекурсивный** вариант процедур обхода, если учесть, что рекурсивные вызовы и возвраты используют **стековый** принцип работы. Например, рассмотрим схему реализации нерекурсивного симметричного обхода. В соответствии с данным правилом сначала надо обработать всех левых потомков, т.е. спустится влево максимально глубоко. Каждое продвижение вниз к левому потомку приводит к запоминанию в стеке адреса бывшей корневой вершины. Тем самым для каждой вершины в стеке запоминается путь к этой вершине от корня дерева. Для нерекурсивного обхода дерева необходимо объявить вспомогательную структуру данных – стек. В информационной части элементов стека должны храниться адреса узлов этого дерева, поэтому ее надо описать с помощью соответствующего ссылочного типа Node.

Еще одним интересным применением процедур обхода является уничтожение всего дерева с освобождением занимаемой вершинами памяти. Ясно, что в простейшем поддереве надо сначала удалить левого и правого потомка, а уже затем – саму корневую вершину. Здесь наилучшим образом подходит правило обхода в обратном направлении. Наличие нескольких правил обхода дерева вполне обоснованно, и в каждой ситуации надо выбирать подходящее правило.

**Операции над абстрактным бинарным деревом**

1. Создать пустое бинарное дерево.
2. Создать бинарное дерево, содержащее один узел, по заданному элементу.
3. Создать бинарное дерево по заданному корню и двум бинарным поддеревьям этого корня.
4. Присоединить к корню бинарного дерева левое или правое поддерево.
5. Обойти узлы бинарного дерева в прямом, симметричном или обратном порядке Отсоединить от корня бинарного дерева левое или правое поддерево
6. Уничтожить бинарное дерево.

**Работа с бинарным деревом**

Для работы с деревьями имеется множество алгоритмов. К наиболее важным относятся задачи построения и обхода деревьев.

* **Создать бинарное дерево**
* **Добавить узел в дерево**
* **Вывести на печать бинарное дерево**
* **Обойти дерево 3 методами**
* **Удалить элемент в бинарном дереве**
* **Уничтожить бинарное дерево**

**Построение дерева**

Пусть двоичное дерево поиска описывается следующим типом

**struct TREE {  
   intinf;  
  TREE \*right;  
  TREE \*left;  
};**

Определим переменные и операции для работы с деревом:

Начальные данные:

**Root**– корень дерева

**X** – переменная для вставки элемента в дерево

**Level** – количество уровней дерева

Промежуточные данные:

**Root->left** – указатель на левое поддерево

**Root->right** – указатель на правое поддерево

**K** – Переменная для выбора действия

**Функции**

**tree \*insert (tree \*root, intitem)** - Создание первого элемента и добавление элементов

**intcount\_elem (tree \*root)** -Подсчет количества узлов дерева

**void print (tree \*root, int level)-**Выводдереванаэкран

**int height (tree \*root)-**Вычислениевысотыдерева

**voiddeleteTree(tree \*root)-**Уничтожениедерева

**voidpreorder (tree \*root) -** обход в прямом порядке(Нисходящий)

**inorder(tree \*root) -** обход симметричным способом (Последовательный )

**postorder(tree \*root) -** обход в обратном порядке(Восходящий)

**Добавление, обход, поиск и удаление узлов дерева производятся рекурсивно.**

В функции добавления элемента число сравнивается со значением в корне. Если оно не больше, то функция вызывается для добавления этого числа к левому поддереву, если больше, то к правому. В результате, происходит спуск к пустому поддереву, вместо которого формируется новый элемент и добавляется к дереву. Аналогично работает подпрограмма поиска.