Тема 7.2. Нелинейные структуры данных. Деревья

- Деревья и рекурсивные алгоритмы
- Общий вид алгоритма полного рекурсивного обхода дерева
- Виды алгоритмов, работающих с деревом
- Способы представления деревьев
- Эффективность алгоритмов, работающих с деревьями
- Алгоритмы, основанные на полном обходе дерева
- Линейные структуры данных на деревьях

ДЕРЕВЬЯ И РЕКУРСИВНЫЕ АЛГОРИТМЫ

Дерево – иерархическая структура некоторой совокупности элементов.

Массивы, массивы указателей и списки имеют линейную структуру, единственный порядок обхода, который и определяет порядок следования (перечисления, логической нумерации) элементов. Деревья и графы представляют собой структуры, которые не допускают подобной «линеаризации»: их **невозможно «вытянуть в линию»** и для их изображения необходима плоскость. С точки зрения организации данных это дает разнообразие вариантов размещения одного и того же набора данных и различные варианты обхода одной и той же структуры.

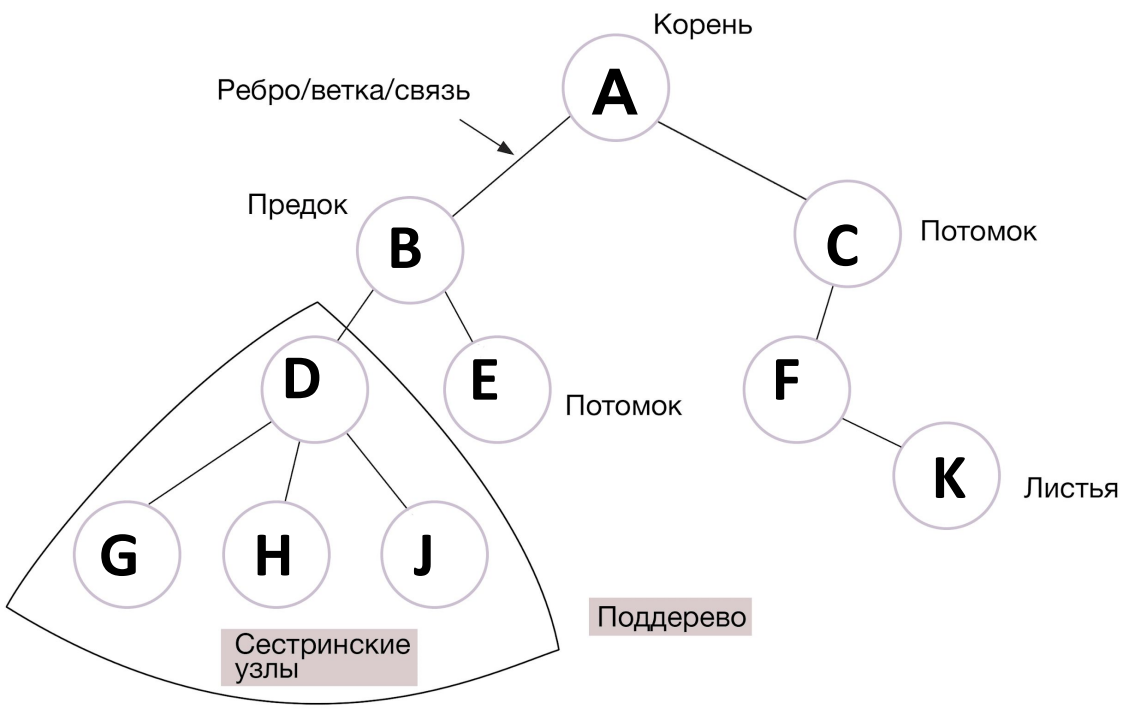
ОПРЕДЕЛЕНИЕ:

Дерево – конечное множество T, состоящее из одного или более узлов, таких что:

- имеет один специально обозначенный узел, называемый корнем данного дерева;
- остальные узлы (исключая корень) содержаться в попарно непересекающихся множествах $T_1, T_2, \ldots, T_n$, каждое из которых в свою очередь является деревом. Деревья $T_1, T_2, \ldots, T_n$ называются поддеревьями данного дерева;
- это определение является рекурсивным, т. е. мы определили дерево в терминах самих же деревьев.

ДЕРЕВЬЯ И РЕКУРСИВНЫЕ АЛГОРИТМЫ

Определение дерева имеет рекурсивную природу. Элемент этой структуры данных называется **вершиной**. Дерево представляет собой вершину, имеющую ограниченное число связей (**ветвей**) к другим деревьям. Нижележащие деревья для текущей вершины называются **поддеревьями**, а их головные вершины - **потомками**. По отношению к потомкам текущая вершина называется **предком**. Вершины, не имеющие потомков, называются **листьями**, головная вершина всего дерева называется **корневой**. **Высота дерева** определяется количеством уровней, на которых располагаются его узлы.



Число поддеревьев данного узла называется степенью этого узла. Узел с нулевой степенью называется листом. **Рекурсивное определение дерева ведет к тому, что алгоритмы работы с ним тоже являются рекурсивными.** На самом деле возможны и циклические алгоритмы, но они являются следствием линейной рекурсии, основанной на выборе.

ПРИМЕРЫ ДРЕВОВИДНОЙ СТРУКТУРЫ

1. Генеалогическое древо. Представьте себе генеалогическое древо отношений между поколениями: бабушки и дедушки, родители, дети, братья и сестры и т.д. Мы обычно организуем семейные деревья иерархически.
2. Организационные диаграммы, например структура организации имеет иерархический вид.
3. Деревья используются для представления синтаксических структур в компиляторах программ. В HTML, объектная модель документа (DOM) представляется в виде дерева. HTML-тег содержит другие теги. У нас есть тег заголовка и тег тела. Эти теги содержат определенные элементы. Заголовок имеет мета теги и теги заголовка. Тег тела имеет элементы, которые отображаются в пользовательском интерфейсе, например, h1, a, li и т.д.
4. Деревья используются для организации информации в системах управления базами данных. B-дерево применяется для структурирования (индексирования) информации на жёстком диске (как правило, метаданных).
5. И т.д.

ОБЩИЙ ВИД АЛГОРИТМА ПОЛНОГО РЕКУРСИВНОГО ОБХОДА ДЕРЕВА

Алгоритм не зависит от формы представления дерева.

Идея: любое действие, выполняемое над вершиной, должно быть выполнено также и по отношению ко всем его поддеревьям, а значит, алгоритм должен быть рекурсивно выполнен по отношению ко всем потомкам этой вершины. В качестве параметра обязателен идентификатор текущей вершины (индекс, указатель, ссылка).

```
void ScanTree( текущая вершина)
{
if (текущая вершина==NULL) return;
for( перебор потомков) ScanTree( i-ый потомок)
}
```

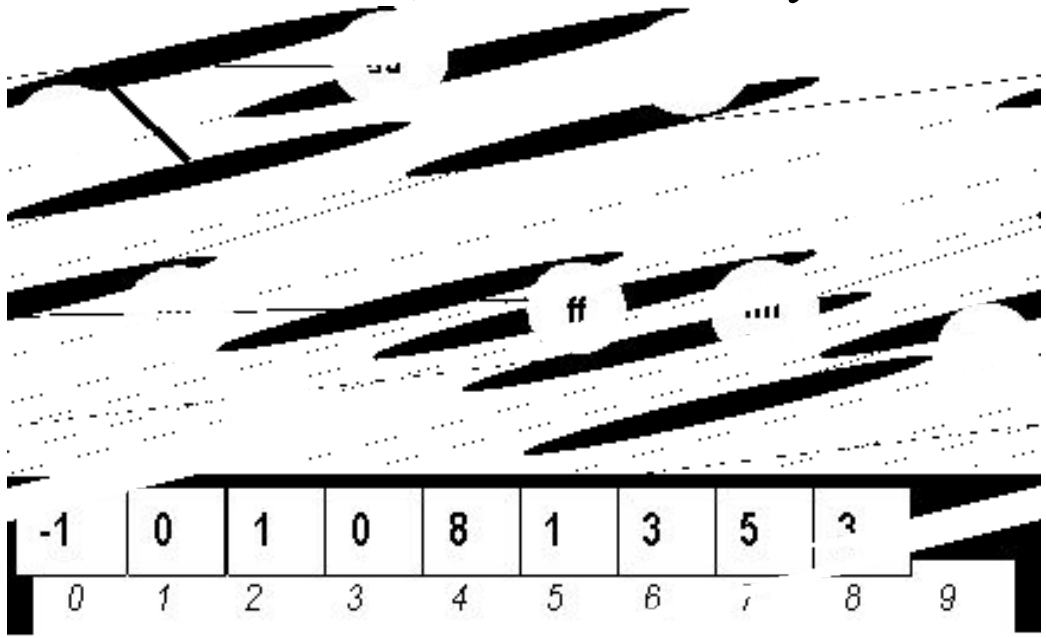
ВИДЫ АЛГОРИТМОВ, РАБОТАЮЩИХ С ДЕРЕВОМ

Когда речь идет о древовидных структурах, следует отличать их абстрактное определение от конкретного способа их реализации в памяти. Последнее зависит также от вида алгоритмов, работающих с деревом:

- если используется рекурсивный или циклический алгоритм, начинающий работать с корневой вершины дерева, то необходимы только прямые ссылки от предка к потомкам;
- если алгоритм предполагает навигацию по дереву во всех направлениях, как вверх, так и вниз по дереву (например, в древовидной системе каталогов), то предполагается наличие как прямых, так и обратных ссылок от потомков к предкам (в системе каталогов – ссылка на родительский каталог);
- возможны алгоритмы, которые работают с деревом, начиная с терминальных вершин. Тогда кроме ссылок от потомков к предкам необходима еще структура данных, объединяющая терминальные вершины (например, массив указателей).

СПОСОБЫ ПРЕДСТАВЛЕНИЯ ДЕРЕВЬЕВ

Составными частями физического представления дерева могут быть массивы, списки, массивы указателей. Представление дерева в виде массива с индексами предков. Поскольку у каждого потомка один единственный предок, то, разместив вершины в массиве, можно в каждую из них поместить индекс предка.

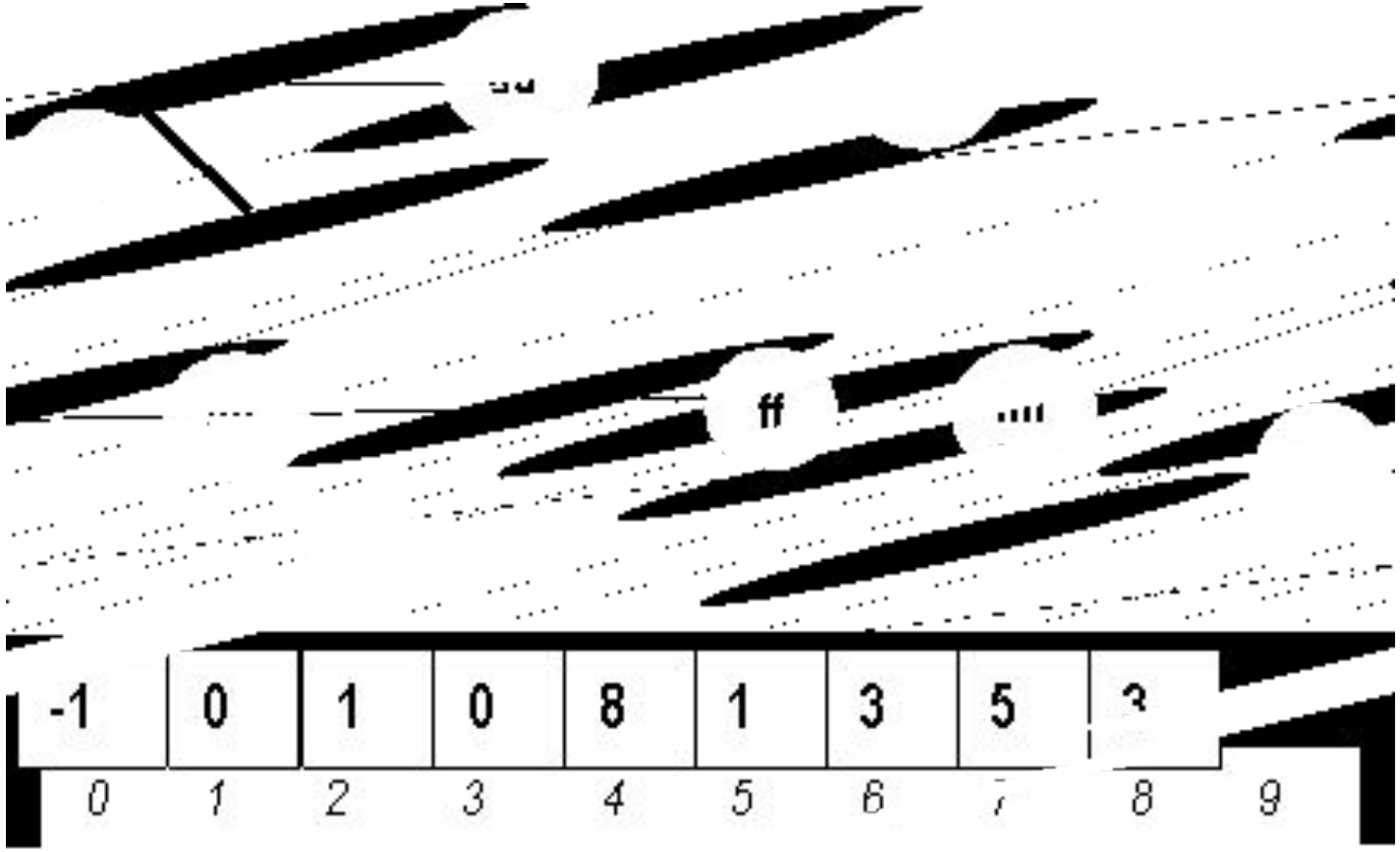



```
#include <iostream>
using namespace std;
struct mtree{
        string s;
        int parent;
        };
// k – индекс предка
void scan_m(mtree A[], int n, int k, int level){
cout<<"l="<<level <<" node="<< k<<" s="<<A[k].s <<endl;
        for (int i=0;i<n;i++)                                   // Цикл выбора потомков вершины
                if (A[i].parent==k) scan_m(A,n,i,level+1);}
int main(){ mtree A1[]={{"aa",-1}, {"bb",0}, {"cc",1}, {"dd",0}, {"ee",8}, {"ff",1}, {"gg",3},
{"hh",5}, {"ii",3}};
   scan_m(A1,11,0,0);
  return 0; }
```

СПОСОБЫ ПРЕДСТАВЛЕНИЯ ДЕРЕВЬЕВ

Это не слишком эффективный способ. Ведь в рекурсивном алгоритме для каждой вершины делается цикл по всему массиву в поисках потомков. Действительно, трудоемкость алгоритма получается T=N*N или N2. Все-таки этому способу можно найти применение, например, если алгоритмы используют просмотр от потомков к предкам. Или, например, в таблицах баз данных, где имеются внутренние эффективные механизмы селекции данных.

```
l=0 node=0 s=aa
l=1 node=1 s=bb
l=2 node=2 s=cc
l=2 node=5 s=ff
l=3 node=7 s=hh
l=4 node=9 s=jj
l=4 node=10 s=kk
l=1 node=3 s=dd
l=2 node=6 s=gg
l=2 node=8 s=ii
l=3 node=4 s=ee
```

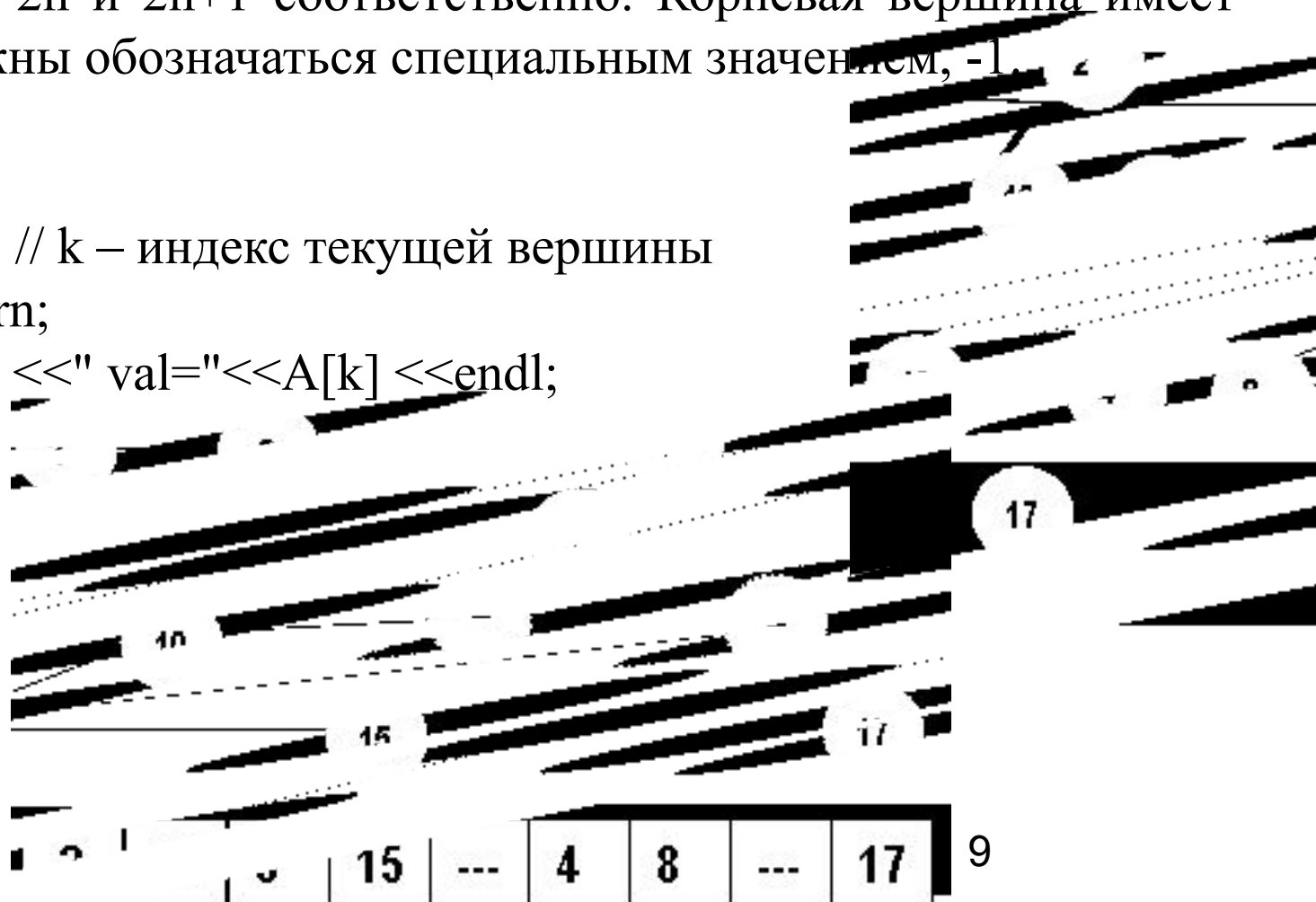


ПРЕДСТАВЛЕНИЕ ДЕРЕВА В МАССИВЕ С ВЫЧИСЛЯЕМЫМИ АДРЕСАМИ ПОТОМКОВ.

Если не искать, как было сделано выше, потомков, то, может быть, их адреса (или индексы) можно вычислить?

Для некоторого вида деревьев, как например, с двумя потомками, принять способ размещения, в котором адреса (индексы) потомков вычисляются через адрес (индекс) предка. Если предок имеет индекс n, то два его потомка - 2n и 2n+1 соответственно. Корневая вершина имеет индекс 1. Отсутствующие потомки должны обозначаться специальным значением, -1.

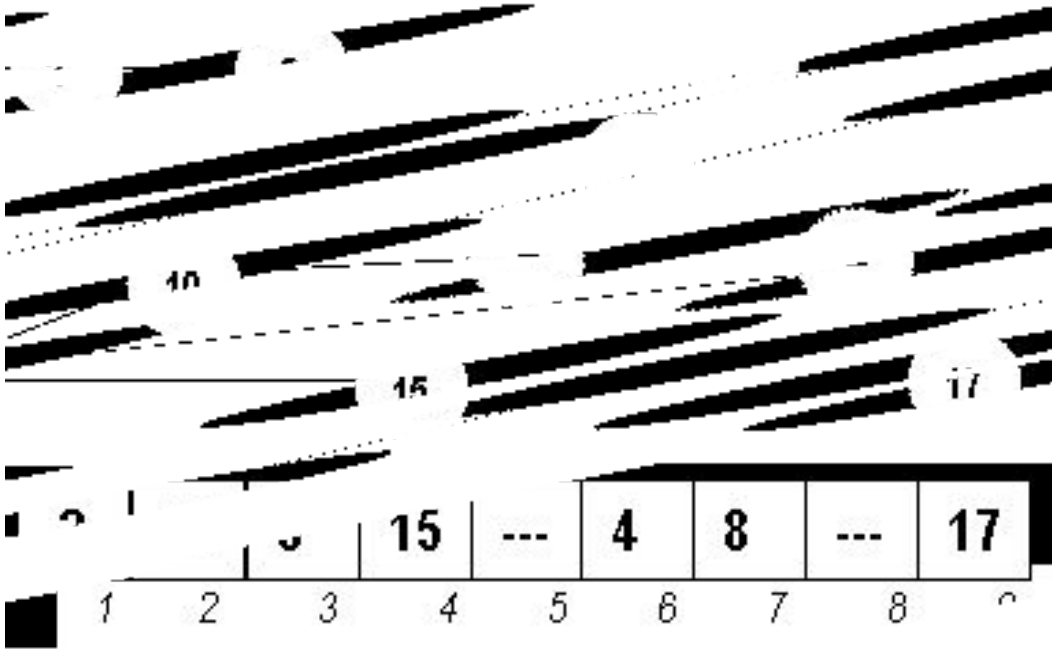
```
#include <iostream>
using namespace std;
void scan_2(int A[], int n, int k,int level){   // k – индекс текущей вершины
        if (k>=n) return; if (A[k]==-1) return;
        cout<<"l="<<level <<" node="<<k <<" val="<<A[k] <<endl;
        scan_2(A,n,2*k,level+1);
        scan_2(A,n,2*k+1,level+1);
}
int A2[]={-1,2,10,3,15,-1,4,8,-1,17};

int main(){ scan_2(A2,10,1,0); return 0; }
```



ПРЕДСТАВЛЕНИЕ ДЕРЕВА В МАССИВЕ С ВЫЧИСЛЯЕМЫМИ АДРЕСАМИ ПОТОМКОВ.

Получается быстро, а главное, без дополнительной информации, индекс массива однозначно определяет положение вершины. Но за это приходится расплачиваться. Каждый следующий уровень требует удвоения размерности массива, вне зависимости от того, сколько вершин этого уровня используются. Поэтому основное требование – сбалансированность. Если есть хотя бы одна ветвь, сильно отличающаяся по длине, то эффективность использования памяти резко снижается. Если же дерево вырождается в список, то размерность массива растет экспоненциально $W=2^N$.

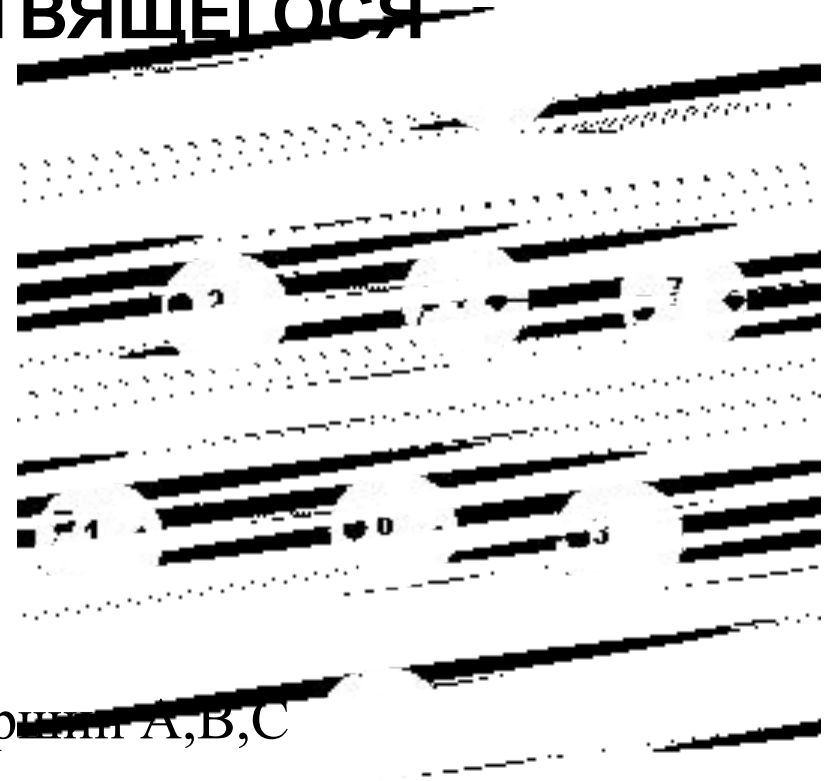
```
l=0 node=1 val=2
l=1 node=2 val=10
l=2 node=4 val=15
l=3 node=9 val=17
l=1 node=3 val=3
l=2 node=6 val=4
l=2 node=7 val=8
```



ПРЕДСТАВЛЕНИЕ ДЕРЕВА В ВИДЕ ВЕТВЯЩЕГОСЯ СПИСКА.

Наиболее близка «по духу» к дереву списковая структура, однако цепочка элементов в данном случае является не линейной, а разветвляющейся. Каждая вершина содержит два указателя – на «старшего сына» – заголовок списка следующего уровня, и на «следующего брата» - ссылка в списке вершин текущего уровня.

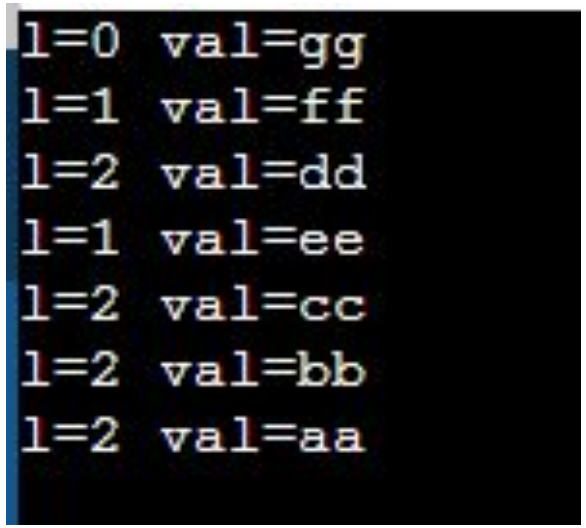
ПРЕДСТАВЛЕНИЕ ДЕРЕВА В ВИДЕ ВЕТВЯЩЕГОСЯ СПИСКА.



```
#include <iostream>
using namespace std;
// Представление дерева в виде разветвляющегося списка
struct ltree{
        string s;
        ltree *son,*bro;   // Указатели на старшего сына
        };                 // и младшего брата
ltree   A={"aa",NULL,NULL}, // Последняя в списке
        B={"bb",NULL,&A},
        C={"cc",NULL,&B},      // Список потомков - концевых вершин A,B,C
        D={"dd",NULL,NULL}, E={"ee",&C,NULL},
        F={"ff",&D,&E},        // Список потомков G - вершин F,E
        G={"gg",&F,NULL}, *ph = &G;
void scan_l(ltree *p, int level){
        if (p==NULL) return;
        cout<<"l="<<level <<" val="<<p->s <<endl;
        for (ltree *q=p->son;q!=NULL;q=q->bro) scan_l(q,level+1); }
int main(){ scan_l(ph,0); return 0;}
```

ПРЕДСТАВЛЕНИЕ ДЕРЕВА В ВИДЕ ВЕТВЯЩЕГОСЯ СПИСКА.

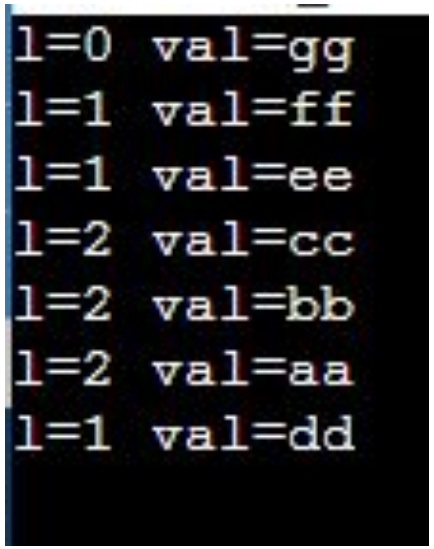
Определение ltree поразительно напоминает двухсвязный список. Ничего удивительного. Ведь определение структуры задает только факт наличия двух указателей, а каким образом они будут связаны – это определяется либо инициализацией, либо алгоритмически. На самом деле при наличии ошибок в программах, работающих со списками, могут получиться похожие «несанкционированные» нелинейные структуры.



ПРЕДСТАВЛЕНИЕ ДЕРЕВА С ИСПОЛЬЗОВАНИЕМ МАССИВА УКАЗАТЕЛЕЙ НА ПОТОМКОВ.

Можно подобрать способ представления, в котором физическая структура максимально соответствует логической структуре дерева, т.е. ее внешнему виду: корень, ветви, потомки. Если ветвь считать указателем, то вершина – это структура, содержащая массив указателей на потомков.

```
#include <iostream>
#define N 4
struct tree{
      string s;
      int n;                          // Количество потомков в МУ
      tree *ch[N]; };
tree  H1={"aa",0},  B1={"bb",0}, C1={"cc",0}, D1={"dd",0},
            E1={"ee",3,&C1,&B1,&H1}, F1={"ff",0},
            G1={"gg",3,&F1,&E1,&D1}, *ph1 = &G1;
void scan(tree *p, int level){
      if (p==NULL) return;
      std::cout<<"l="<<level <<" val="<<p->s <<std::endl;
      for (int i=0; i<p->n; i++)   scan(p->ch[i],level+1);        }
int main(){ scan(ph1,0); return 0;}
```



ЭФФЕКТИВНОСТЬ АЛГОРИТМОВ, РАБОТАЮЩИХ С ДЕРЕВЬЯМИ

Можно провести аналогии между парой «деревья – рекурсивные алгоритмы» и «пространство-время». При работе рекурсивной программы происходит развертке дерева вызовов функции во времени, а дерево, как структура данных, выглядит как отображенный в памяти результат выполнения рекурсивного алгоритма. Именно поэтому к деревьям применимы выводы относительно эффективности рекурсивных алгоритмов:

- полный рекурсивный обход дерева имеет линейную трудоемкость;
- эффективными являются **жадные** алгоритмы. Применительно к дереву жадность состоит в выборе в каждой вершине единственного потомка. Вместо цикла рекурсивного вызова для всех потомков должен быть один вызов. Можно также заменить рекурсивный алгоритм циклическим, переходя на каждом шаге к выбранному потомку. Основанием для однозначного жадного выбора является либо введение в дерево избыточности (дополнительные данные в вершинах), либо упорядочение данных в нем.

Жадный алгоритм (англ. Greedy algorithm) — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным. Известно, что если структура задачи задается матроидом, тогда применение жадного алгоритма выдаст глобальный оптимум. Например задача о размене монет.

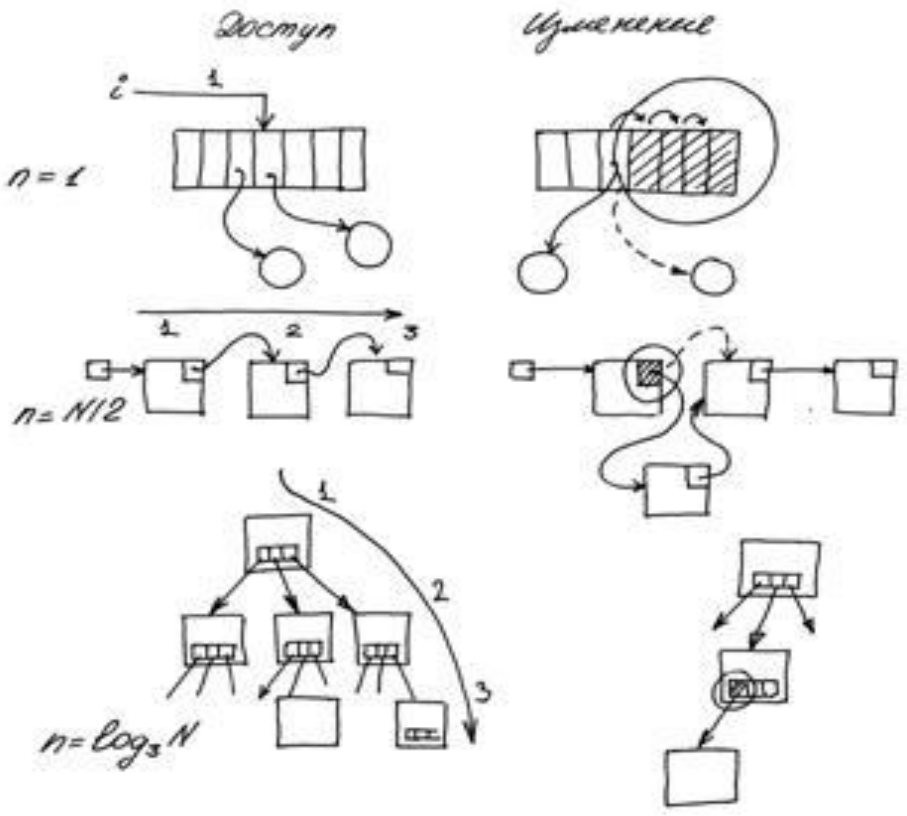
АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ РЕКУРСИВНОМ ОБХОДЕ ДЕРЕВА.

Для начала рассмотрим простейшие алгоритмы безотносительно к способам организации данных в дереве. Полный рекурсивный обход дерева предполагает просмотр всех вершин дерева и с целью получения общих характеристик всей древовидной структуры. Сразу же следует остановиться на технологических способах формирования результата обхода:

- явный результат рекурсивной функции предполагает его накопление в процессе выполнения цепочки возвратов из рекурсивной функции (т.е. накопление результат идет в обратном направлении – от потомков к предку). При этом каждая вершина, получая результаты от потомков, вносит собственную «ложку дегтя», т.е. объединяет результаты поддеревьев с собственным;
- возможно использование формального параметра – ссылки, которая передается по цепочке рекурсивных вызовов. В этом случае все рекурсивные вызовы ссылаются на общую переменную, которая играет роль глобальных данных, используемых для накопления результата.

ПРЕДВАРИТЕЛЬНОЕ СРАВНЕНИЕ СО СПИСКАМИ И МАССИВАМИ.

Даже не вдаваясь в подробности организации данных в дереве, можно сделать предварительные выводы, основываясь на известных нам формах его представления. Во-первых, **в алгоритмическом аспекте дерево реализует известную поговорку «дальше в лес – больше дров»**. «Дрова» - вершины, для которых наблюдается экспоненциальный рост количества с ростом «глубины» дерева. Если при этом удается организовать эффективное отсечение «лишних» поддеревьев, то можно надеяться на **эффективные алгоритмы поиска элементов по значению и доступа к ним по логическому номеру**.

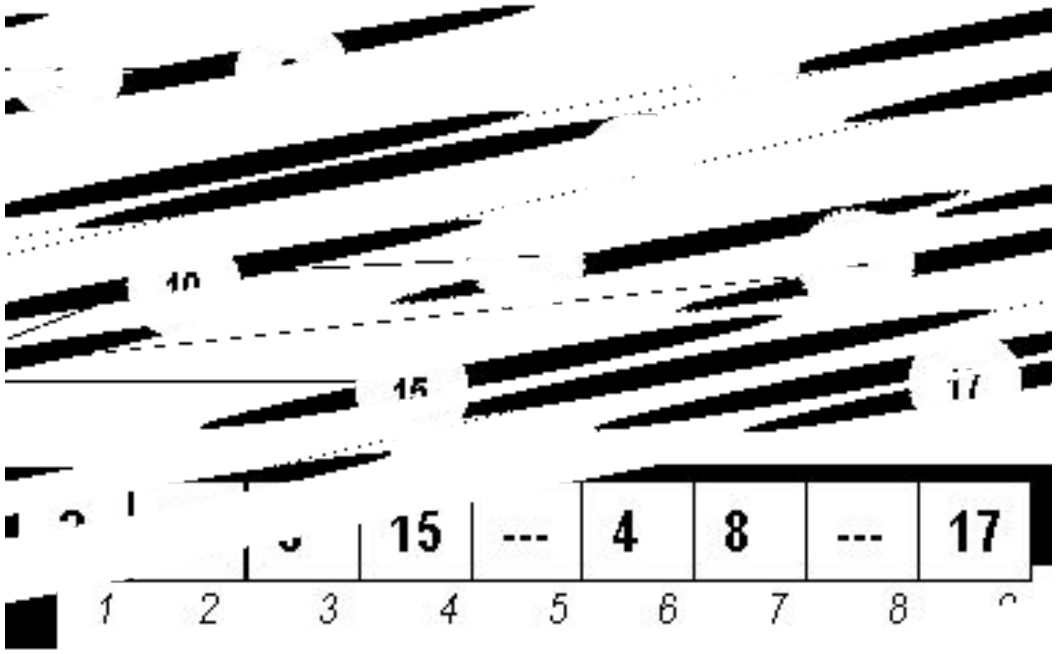


Здесь имеется **явное преимущество перед списками, где все подобные алгоритмы основаны на полном переборе** (линейном поиске). Во-вторых, в технологическом аспекте **изменение порядка следования или размещения вершин в деревьях может быть достигнуто переустановкой связей (ветвей) у отдельных вершин**, так же, как это делается в списках. Здесь имеется явное **преимущество перед массивами**, для которых требуется массовое перемещение (сдвиг) элементов. Таким образом, с точки зрения **эффективности работы дерево представляет собой компромисс между двумя крайностями: массивом и списком.**

АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ ОБХОДЕ ДЕРЕВА

Рекурсивное определение дерева и рекурсивный же алгоритм его обхода позволяют выполнить просмотр всех вершин дерева и получить общие характеристики всего дерева. Естественным выглядит здесь обратное накопление результата в рекурсивной функции: потомки возвращают значения, которые интегрируются с результатом текущей вершины и возвращаются к предку.

```
// Алгоритмы, основанные на полном обходе дерева
struct tree1{
        int val;
        int n;
        tree1 *ch[10];};
//-------- Количество вершин в дереве
int F1(tree1 *p){
        int s=1;
        for (int i=0;i < p->n; i++) s+=F1(p->ch[i]);
        return s;}
//--------- Сумма значений в вершине дерева
int F2(tree1 *p){
        int s=p->val;
        for (int i=0;i < p->n; i++)  s+=F2(p->ch[i]);
        return s;}
```



АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ ОБХОДЕ ДЕРЕВА

Рекурсивное определение дерева и рекурсивный же алгоритм его обхода позволяют выполнить просмотр всех вершин дерева и получить общие характеристики всего дерева. Естественным выглядит здесь обратное накопление результата в рекурсивной функции: потомки возвращают значения, которые интегрируются с результатом текущей вершины и возвращаются к предку.

```
// Алгоритмы, основанные на полном обходе дерева
struct tree1{
        int val;
        int n;
        tree1 *ch[10];};
//-------- Количество вершин в дереве
int F1(tree1 *p){
        int s=1;
        for (int i=0;i < p->n; i++) s+=F1(p->ch[i]);
        return s;}
//--------- Сумма значений в вершине дерева
int F2(tree1 *p){
        int s=p->val;
        for (int i=0;i < p->n; i++)  s+=F2(p->ch[i]);
        return s;}
```

Из приведенных примеров видно, что обе функции накапливают сумму некоторых значений, получаемых от потомков. Ответ на вопрос «Сумму чего?» следует искать в другом месте: результат зависит от того, что добавляет в сумму текущая вершина.

АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ ОБХОДЕ ДЕРЕВА

В первом примере в вычислении максимума из потомков участвует значение в текущей вершине, что однозначно определяет общий результат. Во втором случае не все так очевидно. Логический анализ не позволяет сформулировать результат: Функция возвращает максимальное значение «чего-то», полученного от потомков, увеличивая его в текущей вершине на 1. Только исторический анализ и непосредственное наблюдение за программой позволяют догадаться, что результатом функции (а также инвариантом рекурсивного алгоритма) является максимальная длина ветви: текущая вершина удлиняет путь на 1.

```
//-------  Максимальное значение в вершине дерева
int F3(tree1 *p){
        int s=p->val;
        for (int i=0;i < p->n; i++)
                { int vv=F3(p->ch[i]); if (vv > s)  s=vv; }
        return s;}
//---------- Максимальная длина ветви дерева
int F4(tree1 *p){
        int s=0;
        for (int i=0;i < p->n; i++)
                { int vv=F4(p->ch[i]); if (vv > s)  s=vv; }
        return s+1;}
```

АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ ОБХОДЕ ДЕРЕВА

Рекурсивный обход позволяет получить другие характеристики дерева, например, передавая в качестве формального параметра текущую «глубину» вершины, можно подсчитать сумму длин ветвей (расстояний до корня), что служит характеристикой сбалансированности дерева: в сбалансированном дереве оно близко к log2(N), в вырожденном списке – к N/2.

```
//----- Суммарное расстояние до корня - степень сбалансированности
int F6(tree1 *p, int L){
        int s=L;
        for (int i=0;i < p->n; i++)
                s+=F6(p->ch[i],L+1);
        return s;}
double main6(tree1 *p){ return ((double)F6(p,1))/F1(p); }
```

АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ ОБХОДЕ ДЕРЕВА

При поиске в дереве вершины, значение в которой удовлетворяет заданному условию, кроме непосредственно обнаружения вершины нужно еще оборвать процесс рекурсивного обхода у всех вершин – предков, которым передается указатель на найденную вершину.

```
//--------  Поиск первого значения, удовлетворяющего условию
tree1 *F7(tree1 *p, int vv){                          // Действие, выполняемое потомком
if (p->val ==vv) return p;            // Найдено в текущей вершине - вернуть
        for (int i=0;i < p->n; i++){
tree1 *q=F7(p->ch[i]);                // Найдено у потомка – прекратить обход
if (q!=NULL) return q; }              // Обработка результата предком
        return NULL;}
```

Кажущийся парадокс алгоритма, работающего с деревом. Действие, выполняемое в вершине – предке, связанное с получением результата, записано в теле функции после действия, вызвавшее его в потомке.

Для сохранения результата явном виде используется результат рекурсивной функции. Даже для вершин, не принимающих участия в его формировании, его надо явно передавать от потомков к предку. Бывает удобнее использовать в качестве формального параметра ссылку на общую переменную – результат, которая играет роль глобальной для множества экземпляров рекурсивной функции, соответствующих вершинам дерева (хотя синтаксически может таковой и не являться).

АЛГОРИТМЫ, ОСНОВАННЫЕ НА ПОЛНОМ ОБХОДЕ ДЕРЕВА

```
// Поиск свободной вершины с min глубиной. Ссылки на параметры, общие для всех вершин
// lmin - минимальная глубина, pmin - вершина минимальной глубины
void find_min(tree *p, int level, int &lmin, tree *&pmin){
        if (p==NULL) return;
        if (lmin!=-1 && level >=lmin) return;                  // Заведомо худший вариант
        if (p->n!=N && (lmin==-1 || level<=lmin))               // Вершина ближе
                { lmin=level; pmin=p; return; }
        for (int i=0; i<p->n;i++)                              // Количество потомков = N
                find_min(p->ch[i],level,lmin,pmin);
        }}
void main(){
        tree *pp2;…..                          // Корневая вершина
tree *q;                              // Результат – указатель на найденную
        int lm=-1;                             // Результат – минимальная глубина
        find_min(pp2,0,lm,q);…}
```

Обратите внимание на элемент оптимизации: если обнаружена вершина со свободной ветвью на некоторой глубине, то обход дерева ограничивается этой глубиной: заведомо худшие варианты уже не просматриваются.

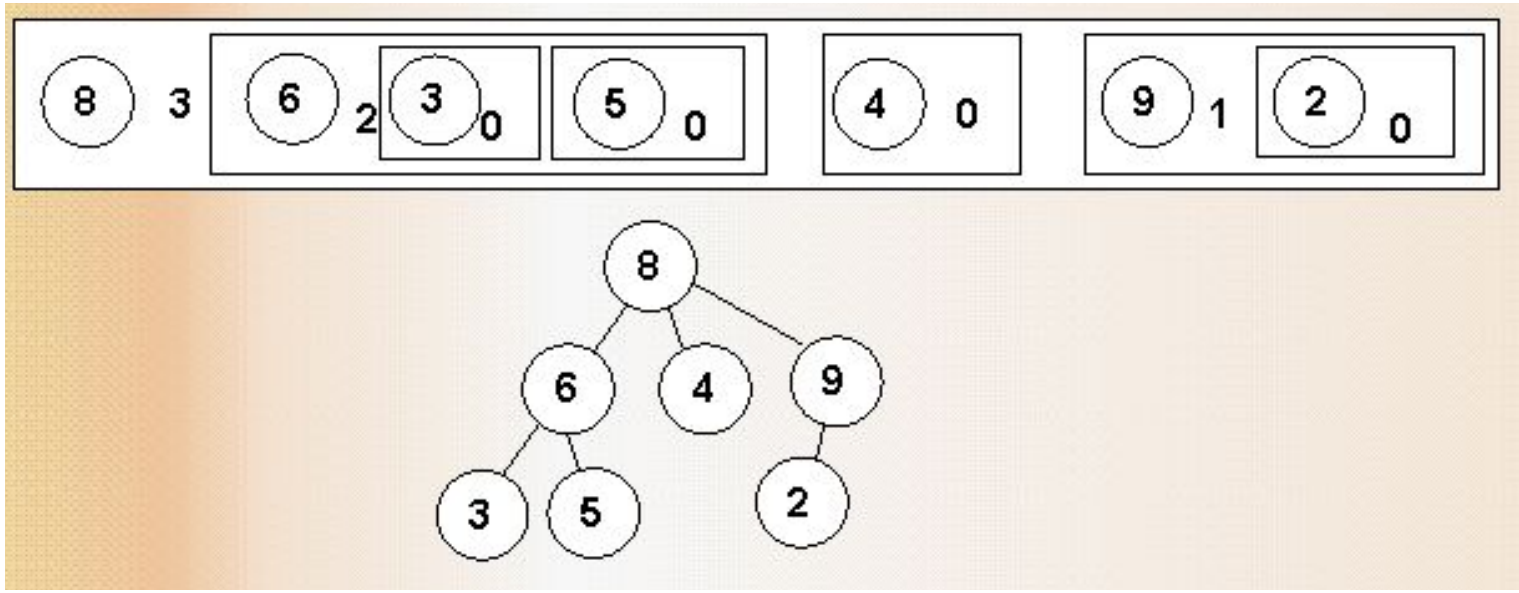
СОХРАНЕНИЕ ДЕРЕВА В ПОСЛЕДОВАТЕЛЬНЫЙ ПОТОК

До сих пор мы рассматривали сохранение в последовательном текстовом потоке данных, хранимых в линейных структурах. Для дерева возможны два варианта:

- сохранение данных, хранящихся в вершинах дерева. В этом случае структура (топология) дерева теряется и при загрузке этих же данных в новое дерево его топология будет уже другой. Возможны и парадоксы. При сохранении данных сбалансированного двоичного дерева в файле получится упорядоченная последовательность, обратная загрузка которой простейшим алгоритмом приведет к вырождению дерева в линейный список;
- для сохранения структуры (топологии) дерева достаточно использовать рекурсивный последовательный формат, например, записывать количество прямых потомков в вершины и вызывать для них рекурсивно функцию сохранения.

СОХРАНЕНИЕ ДЕРЕВА В ПОСЛЕДОВАТЕЛЬНЫЙ ПОТОК

```
//-------------Сохранение в последовательный поток
void save(tree *p,  ofstream &fd){
        fd<<p->val<<" "<<p->n<<endl;
        for (int i=0;i<p->n;i++)
              save(p->ch[i],fd);          }
//-------------Загрузка из последовательного потока
tree *load(ifstream &fd){
        tree *p=new tree;
        fd>>p->val;
        fd>>p->n;
        for (int i=0;i<p->n;i++)
                p->ch[i]=load(fd);
        return p;          }
```



Сохранение дерева в последовательный поток

ГОРИЗОНТАЛЬНЫЙ ОБХОД ДЕРЕВА

Рекурсивный обход дерева связан со стеком, который используется рекурсивным алгоритмом для сохранения вызовов. В принципе, стек можно сделать и явным, последовательность обхода от этого не изменится. В стек помещаются указатели на вершины-потомки, причем в обратном порядке (для их извлечения в прямом). Рекурсия превращается в цикл: на каждом шаге из явного стека извлекается очередная вершина, при ее обработке стек дополняется указателями на потомков.

```
// Полный рекурсивный обход дерева с явным использованием стека
void scan1(tree *p){
        tree **stack=new tree*[1000];                // Явный стек
        int sp=-1;
        stack[++sp]=p;                              // В стеке есть вершины
        while(sp!=-1){
                tree *q=stack[sp--];                 // Извлечь очередную
                printf("cnt=%d val=%d\n",q->cnt,q->val);
                for (int i=q->n-1; i>=0; i--)        // Запись в стек в обратном порядке
                        stack[++sp]=q->ch[i];
                }
        delete stack;}
```

ГОРИЗОНТАЛЬНЫЙ ОБХОД ДЕРЕВА

Если же вместо стека применить очередь, то обход дерева будет происходить «по горизонтали». Тогда можно естественным образом реализовать алгоритмы, использующие свойство «близости» к корню, например, построить сбалансированное дерево, размещая вершины на ближайшие свободные места. С практической точки зрения эта задача является чисто умозрительной: на самом деле включение вершин в дерево осуществляется по правилам, сохраняющим определенные свойства, установленные для дерева (например, упорядочение или нумерация в порядке обхода).

```
// Поиск ближайшей к корню вершины со свободной ветвью с использованием очереди вершин
tree *find_first(tree *ph,int sz){
int fst=0,lst=0;
tree **Q=new tree *[sz];  // Циклическая очередь
Q[lst++]=ph;                                  // Поместить исходную в очередь
while(fst!=lst){                               // Пока очередь не пуста
        tree *q=Q[fst++];                      // Извлечь указатель на очередную вершину
        if (fst==sz) fst=0;
        if (q->n !=N) { delete Q; return q; } // Найдена первая со свободными ветвями
                for (int i=0;i<q->n;i++){ Q[lst++]=q->ch[i]; if (lst==sz) lst=0; } // Помещение всех потомков в очередь
        } delete Q; return NULL;} // Очередь пуста – вершина не найдена
```

ГОРИЗОНТАЛЬНЫЙ ОБХОД ДЕРЕВА

Аналогичный алгоритм на основе рекурсивного обхода был рассмотрен выше. Забегая вперед, рассмотрим более эффективный (жадный) алгоритм, основанный на выборе единственного потомка с минимальным числом (счетчиком) вершин в его поддереве. Этот счетчик является элементом избыточности, и его необходимо корректировать в процессе движения по всем проходимым вершинам.

```
// Вставка в поддерево с минимальным количеством вершин
tree *create(int vv){
        tree *q=new tree;
        q->val=vv; q->n=0; q->cnt=1; return q; }
void insert_min(tree *&p, int vv){            // Ссылка на указатель на текущую вершину
        if (p==NULL) { p=create(vv); return; }
        p->cnt++;                             // Наращивать счетчик в промежуточных вершинах
        if (p->n!=N) {                        // Есть свободная ветка – создать вершину
                p->ch[p->n++]=create(vv); return; }
        for (int i=1,k=0; i<N;i++)            // Количество потомков = N
                if (p->ch[i]->cnt < p->ch[k]->cnt)
                        k=i;                  // Искать потомка с min
        insert_min(p->ch[k],vv); }            // числом вершин в поддереве и выбрать его
```

ДЕРЕВЬЯ И РЕКУРСИВНЫЕ АЛГОРИТМЫ

Путем из узла n_1 в узел n_k называется последовательность узлов $n_1, n_2, \dots, n_k$, где $\forall\, i, 1 \le i < k$, узел n_i является родителем узла n_{i+1}. Если существует путь из узла n_1 в узел n_k, то n_1 называется предком n_k, n_k – потомком n_1. Длиной пути называется число на 1 меньше числа узлов, составляющих этот путь.

- Бинарное дерево – дерево, в котором каждый узел имеет не более двух поддеревьев. В этом случае будем различать левое и правое поддерево.
- Дерево двоичного поиска – бинарное дерево, узлы которого помечены элементами множества. Определяющее свойство дерева двоичного поиска заключается в том, что все элементы, хранящиеся в узлах левого поддерева любого узла x, меньше элемента