数据结构:二叉树

Data Structure

主讲教师：屈卫兰

Office number：基地203

tel:　13873195964

二叉树

- 定义及主要特性
- 周游二叉树
- 二叉树的实现
- 二叉查找树
- 堆与优先队列
- huffman编码树

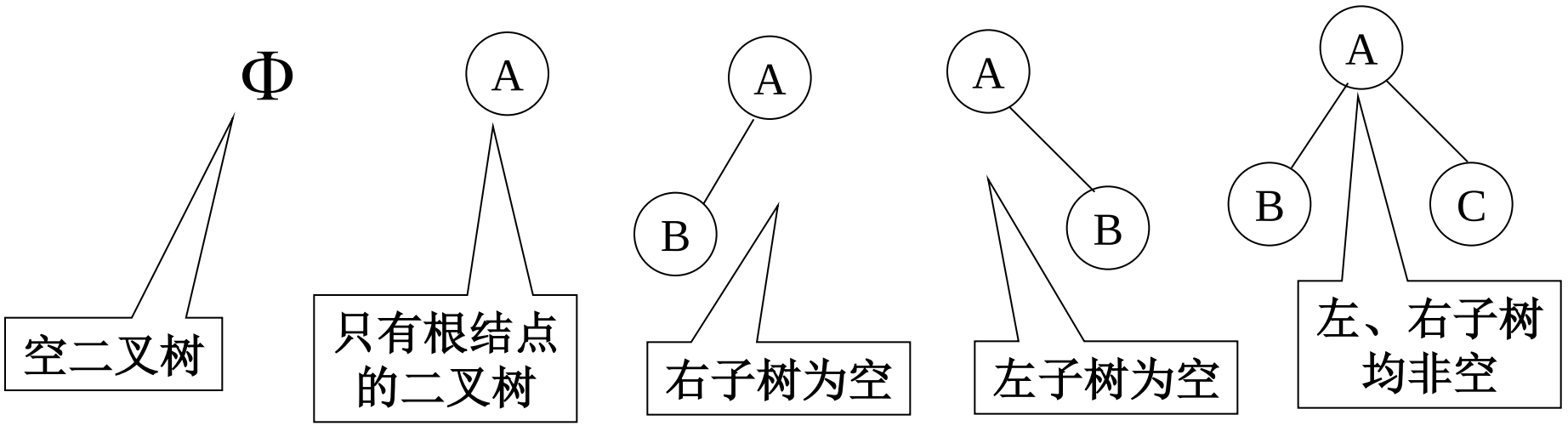
二叉树定义及主要特性

- 递归定义：

  二叉树由结点的有限集合组成，这个集合或者为空，或者由一个根结点及两棵不相交的，分别称作这个根的左子树和右子树的二叉树组成。

- 特点：
  - 每个结点至多有二棵子树。
  - 二叉树的子树有左、右之分，且其次序不能任意颠倒。

基本形态



二叉树的相关术语

- 从一个结点到它的两个子结点都有边(edge)相连，这个结点称为它的子结点的父结点(parent)。
- 如果一棵树的一串结点$n_1, n_2, \ldots, n_k$有如下关系：结点n_i是n_{i+1}的父结点$(1 \le i < k)$，就把$n_1, n_2, \ldots, n_k$称为一条由n_1至n_k的路径(path)。这条路经的长度(length)是k-1(因为k个结点是用k-1条边连接起来的)。如果有一条路径从结点R至结点M，那么R就称为M的祖先(ancestor)，而M称为R的子孙(descendant)。

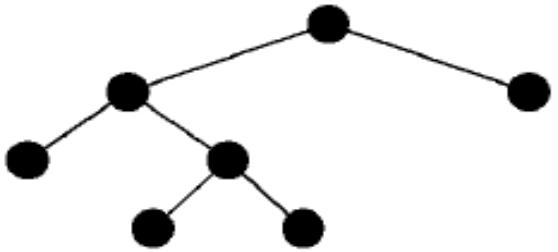
二叉树的相关术语

- 结点M的深度(depth)就是从根结点到M的路径的长度。树的高度(height)等于最深的结点的深度+1。任何深度为d的结点的层数(level)都为d。根结点深度为0，层数也为0。
- 没有非空子树的结点称为叶结点(leaf)或终端结点。至少有一个非空子树的结点称为分支结点或内部结点(internal node)。

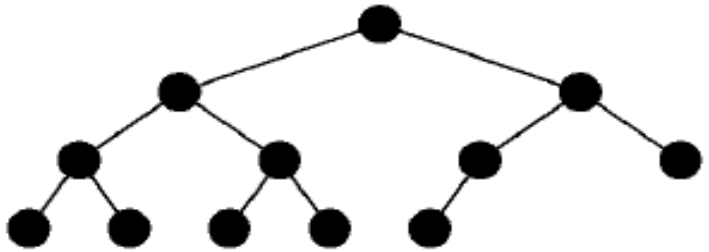
二叉树的相关术语

- 满二叉树

  如果一颗二叉树的任何结点，或者是树叶，或者恰有两个非空子女的分支结点，则此二叉树称为满二叉树。



(a)



(b)

(a)满二叉树(非完全二叉树) (b)完全二叉树(非满二叉树)

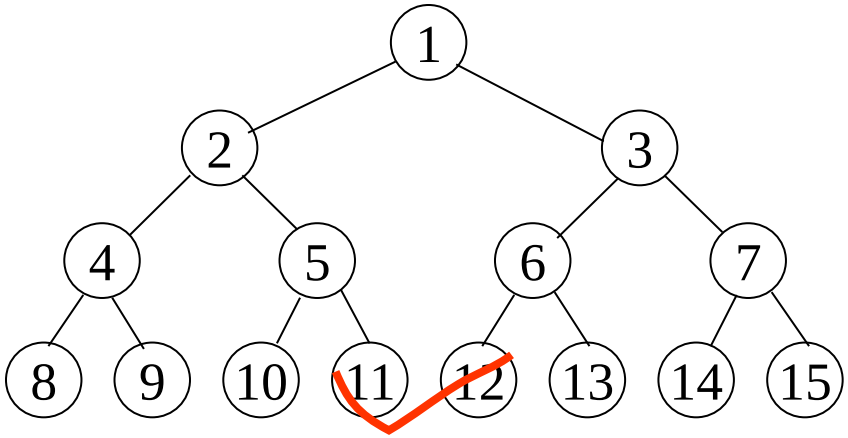
二叉树的相关术语

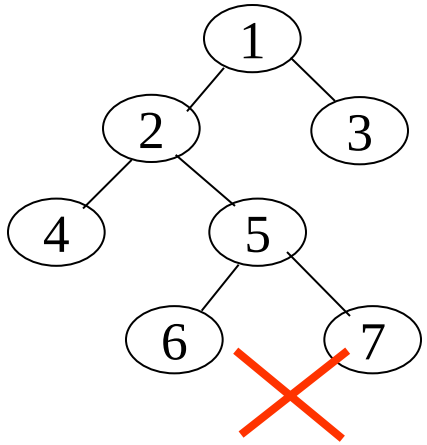
- 完全二叉树

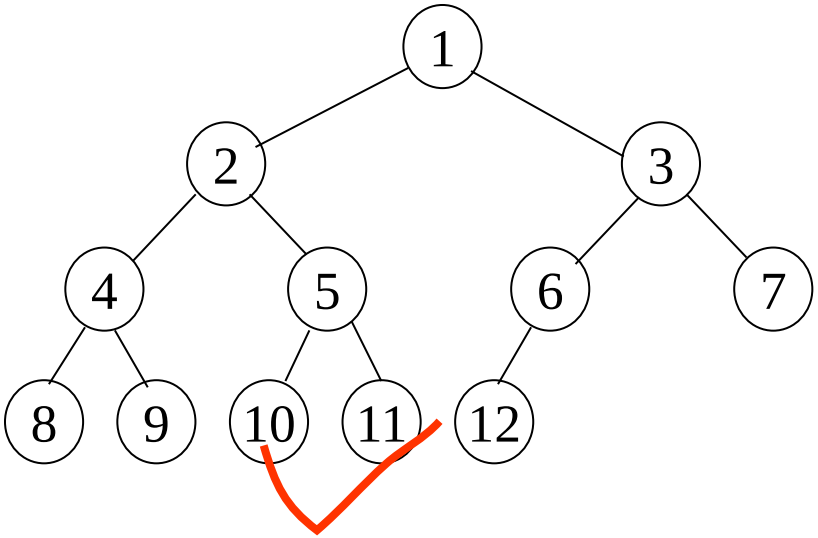
  若一颗二叉树最多只有最下面的两层结点度数可以小于2，并且最下面一层的结点都集中在该层最左边的若干位置上，则称此二叉树为完全二叉树。

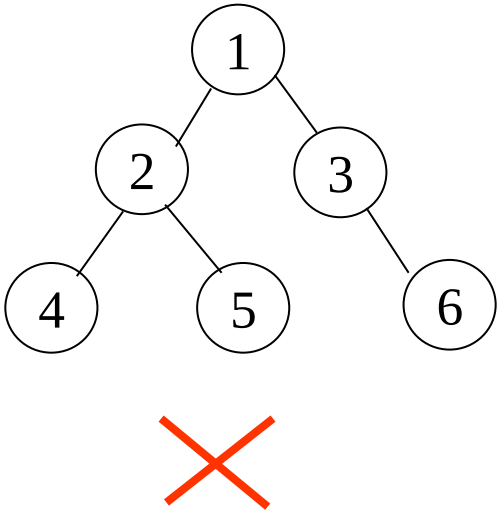
  自根结点起每一层从左至右地填充。一棵高度为d的完全二叉除了d-1层外，每一层都是满的。底层叶结点集中在左边的若干位置上。

完全二叉树









二叉树性质

1. 满二叉树定理：非空满二叉树树叶数等于其分支结点数加1。

证明：设二叉树结点数为n，叶结点数为m，分支结点数为b。

有n（总结点数= m（叶)+b（分支）(1)

∵每个分支，恰有两个子结点（满），故有2*b条边一颗二叉树，除根结点外，每个结点都恰有一条边联接父结点，故共有n-1条边。即n-1=2*b(2)

∴由(1)(2)得n-1=m+b-1=2*b，

得出m(叶) =b（分支）+ 1

二叉树性质

2、满二叉树定理的推论：一棵非空二叉树空子树的数目等于其结点数目加1。

证明1：设二叉树T，将其所有空子树换成叶结点，把新的二叉树记为T‘。所有原来树T的结点现在是树T’的分支结点。

根据满二叉树定理，新添加的叶结点数目等于树T的结点数目加1，

而每个新添加的叶结点对应树T的一棵空子树，因此树T中空子树的数目等于树T中结点数目加1。

二叉树性质

证明2：根据定义，二叉树T中每个结点都有两个子结点指针（空或非空）。

因此一个有n个结点的二叉树有2n个子结点指针。

除根结点外，共有n-1个结点，它们都是由其父结点中相应指针指引而来的，换句话说就有n-1个非空子结点指针。

既然子结点指针数为2n，则其中有n+1个为空(指针)。

二叉树性质

3. 任何一颗二叉树，度为0的结点比度为2的结点多一个。

证明：设有n个结点的二叉树的度为0、1、2的结点数分别为=n0，n1，n2，n=n0 +n1 +n2（公式1）

设边数为e。因为除根以外，每个结点都有一条边进入，故n=e+1。

由于这些边是有度为1和2的结点射出的，因此e=n1+2*n2，于是n=e+1= n1 +2*n2 +1（公式2）

因此由公式（1）（2）得

n0+n1+n2=n1+2*n2+1　即n0 =n2 +1

二叉树性质

4. 二叉树的第i层（根为第0层）最多有2^i个结点
5. 高度为k(深度为k-1。只有一个根结点的二叉树的高度为1，深度为0)的二叉树至多有2^k-1个结点
6. 有n个结点的完全二叉树的高度为$\log_2(n+1)$(深度为$\log_2 n$)

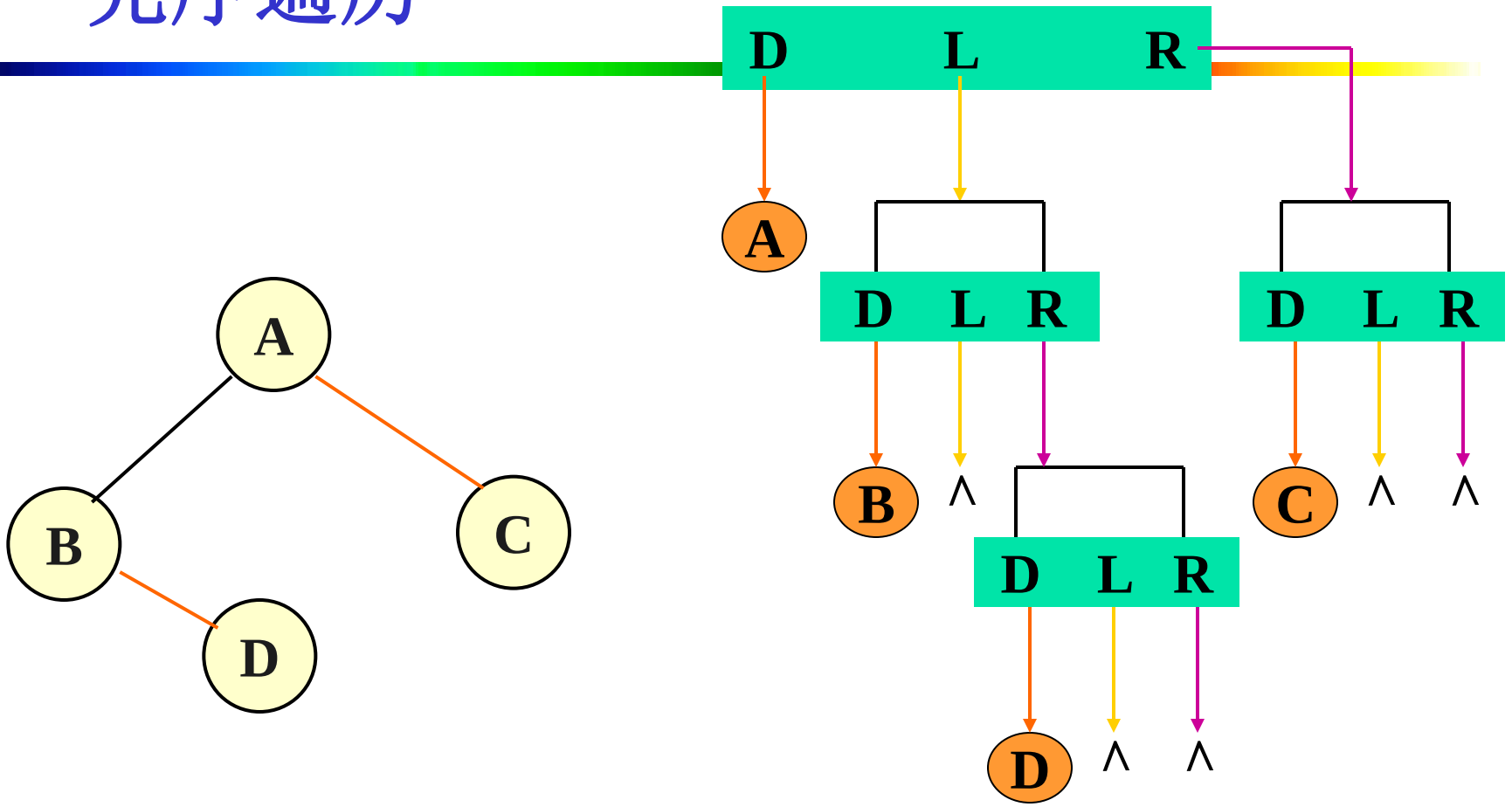
二叉树的抽象数据类型

```
template <class Elem> class BinNode {
public:
   virtual Elem& val( ) = 0 ;
   virtual void setVal (const Elem&) = 0;
  virtual BinNode* left() const = 0;
  virtual BinNode* right() const = 0;
 virtual void setLeft(BinNode*  ) = 0;
  virtual void setRight(BinNode* ) = 0;
  virtual bool isLeaf() = 0;
};
```

遍历二叉树

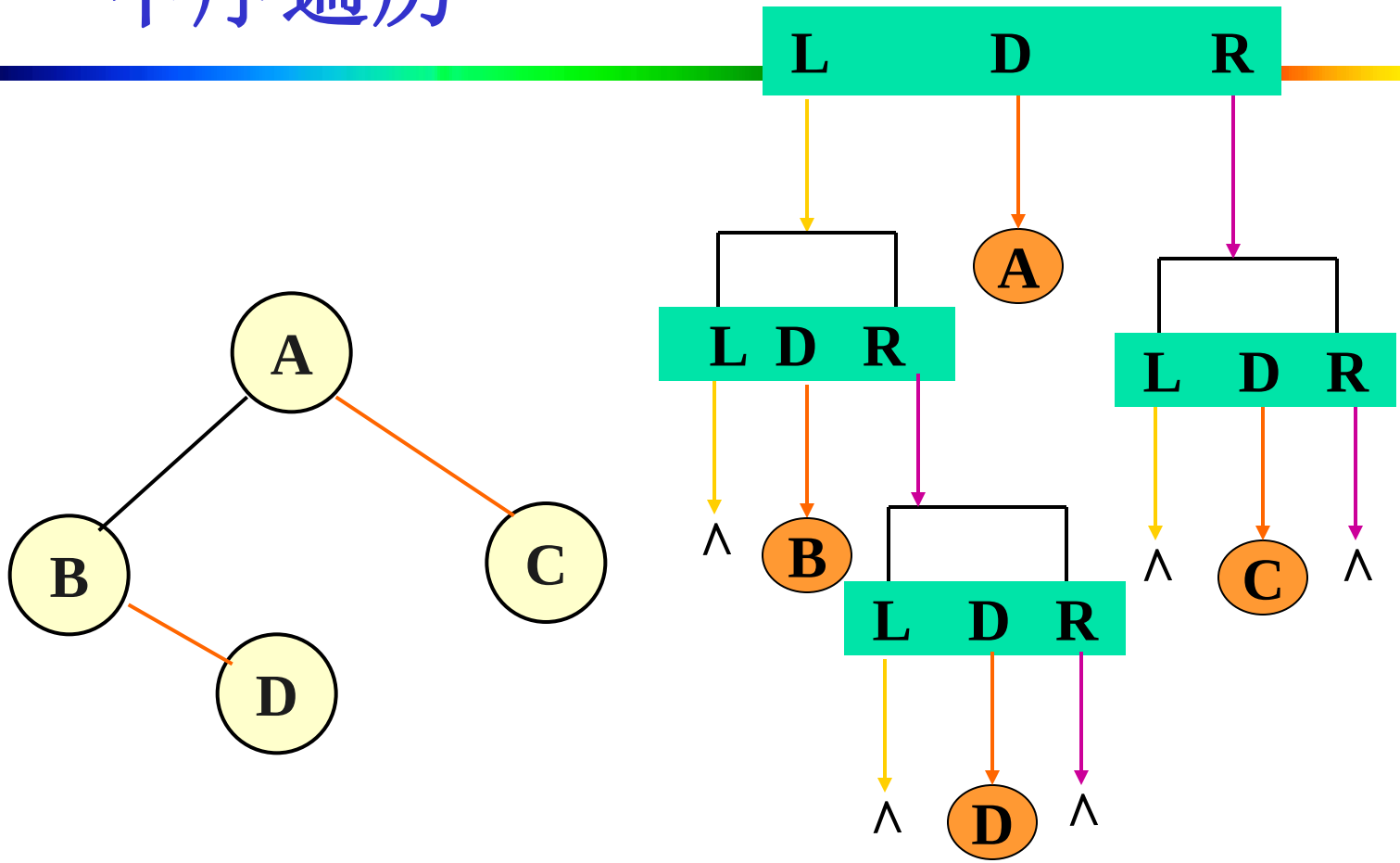
- 遍历：系统地访问二叉树中的结点。每个结点都正好被访问到一次。
- 方法：
  - 前序遍历(preorder traversal)：访问根结点；前序遍历左子树;前序遍历右子树。
  - 中序遍历(inorder traversal)：中序遍历左子树;访问根结点;中序遍历右子树。
  - 后序遍历(postorder traversal)：后序遍历左子树;后序遍历右子树;访问根结点。

先序遍历



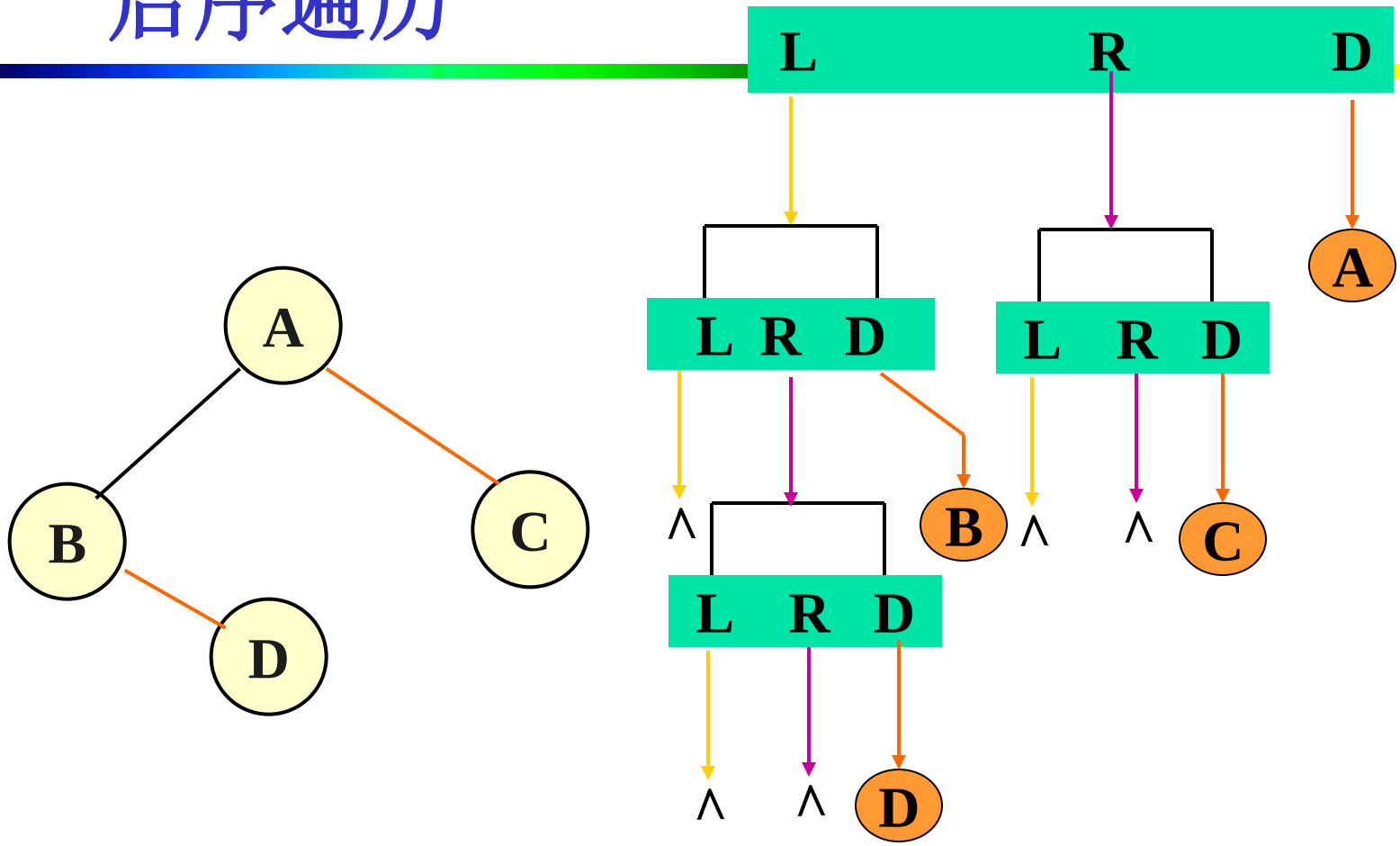
先序遍历序列：A B D C

中序遍历



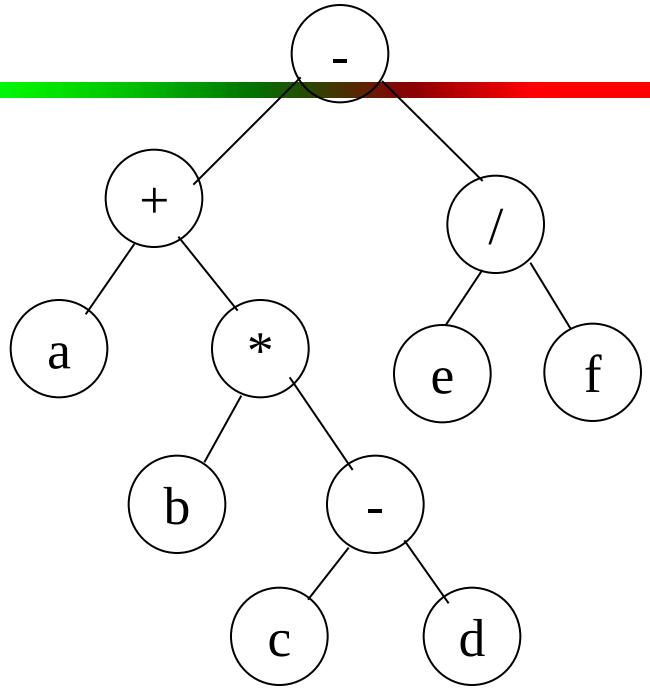
中序遍历序列：B D A C

后序遍历



后序遍历序列： D B C A

举例



先序遍历： - + a * b - c d / e f

中序遍历： a + b * c - d - e / f

后序遍历： a b c d - * + e f / -

层次遍历： - + / a * e f b - c d

前序遍历算法

```
template < class Elem >
void preorder(BinNode<Elem>* subroot)
{
  if (subroot == NULL) return;  //Empty subtree,do
    nothing
  visit(subroot);  //Perform whatever action is desired
  preorder(subroot->left());
  preorder(subroot->right());
}
```

由二叉树的先序和中序序列建树

仅知二叉树的先序序列“abcdefg” 不能唯一确定一棵二叉树，

如果同时已知二叉树的中序序列“cbdaegf”，则会如何？

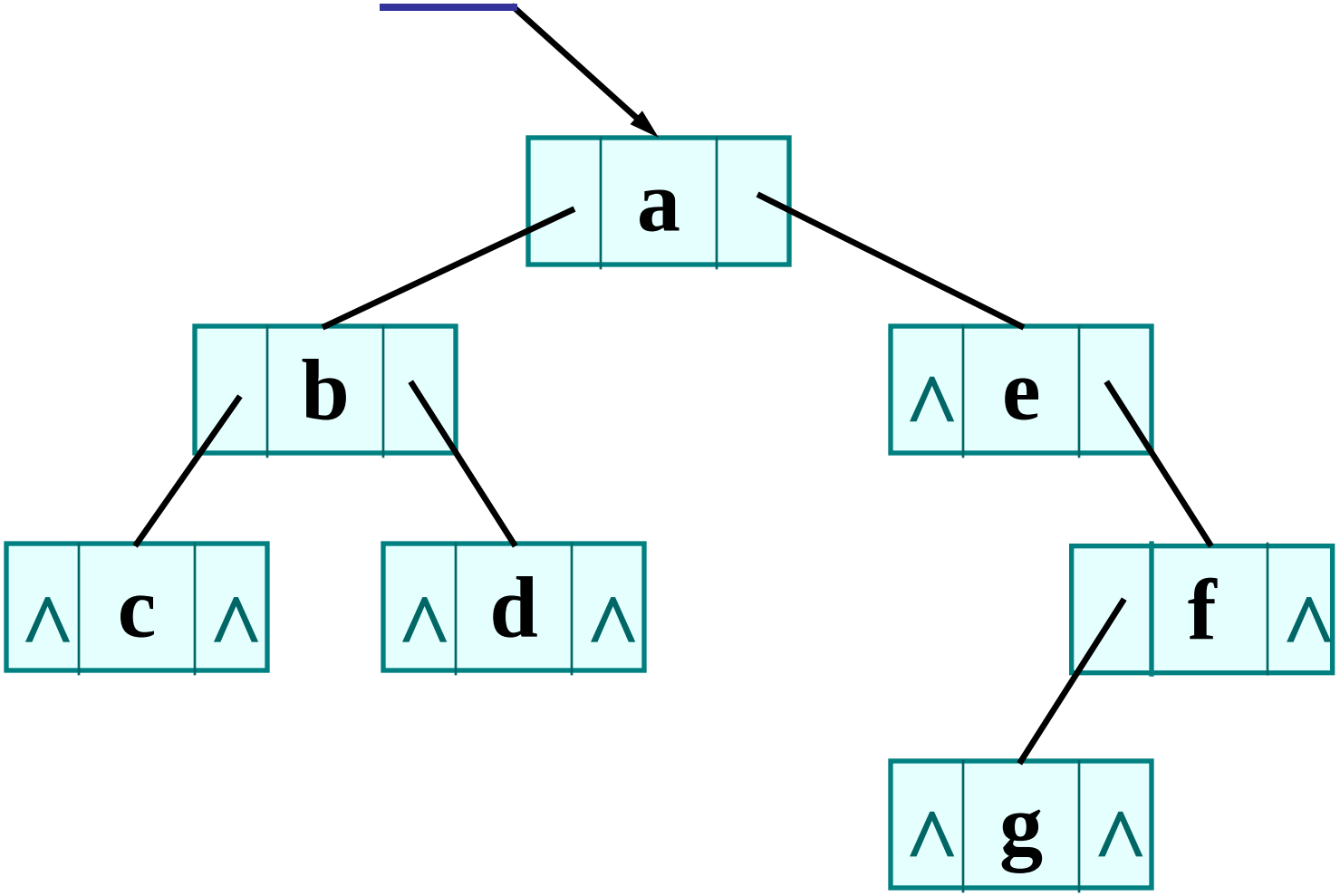
二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

例如：

a b c d e f g　先序序列

c b d a e g f　中序序列



遍历算法应用

计算二叉树的结点数：

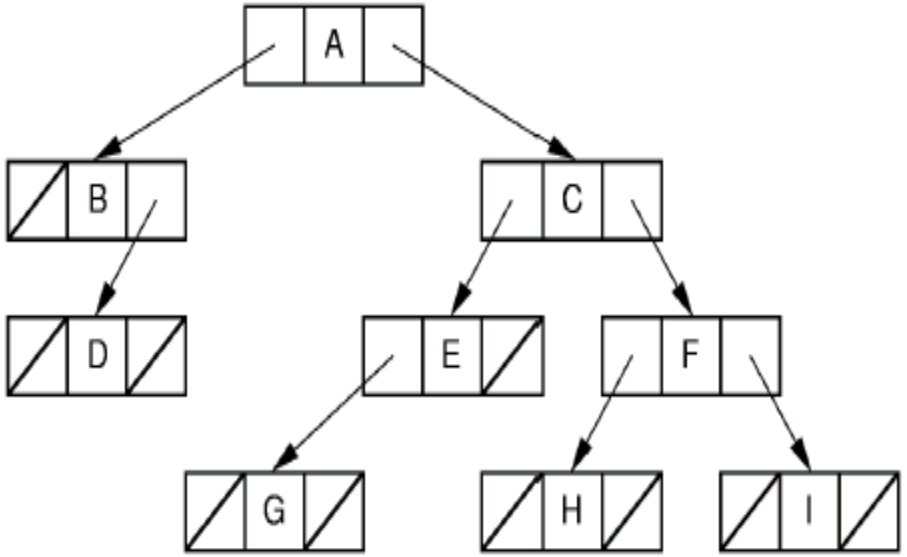
```
template < class Elem >
int count(BinNode<Elem>* subroot)
{
  if (subroot == NULL) return 0;
  return 1 + count(subroot->left())
           + count(subroot->right());
}
```

二叉树的实现

使用指针实现二叉树

- 二叉链表(最常用)

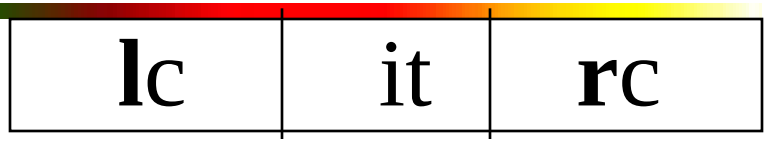
```
class BSTNode:public BinNode<E> {
private:
  Key k;
  E it;
  BSTNode* lc;
  BSTNode* rc;
...};
```

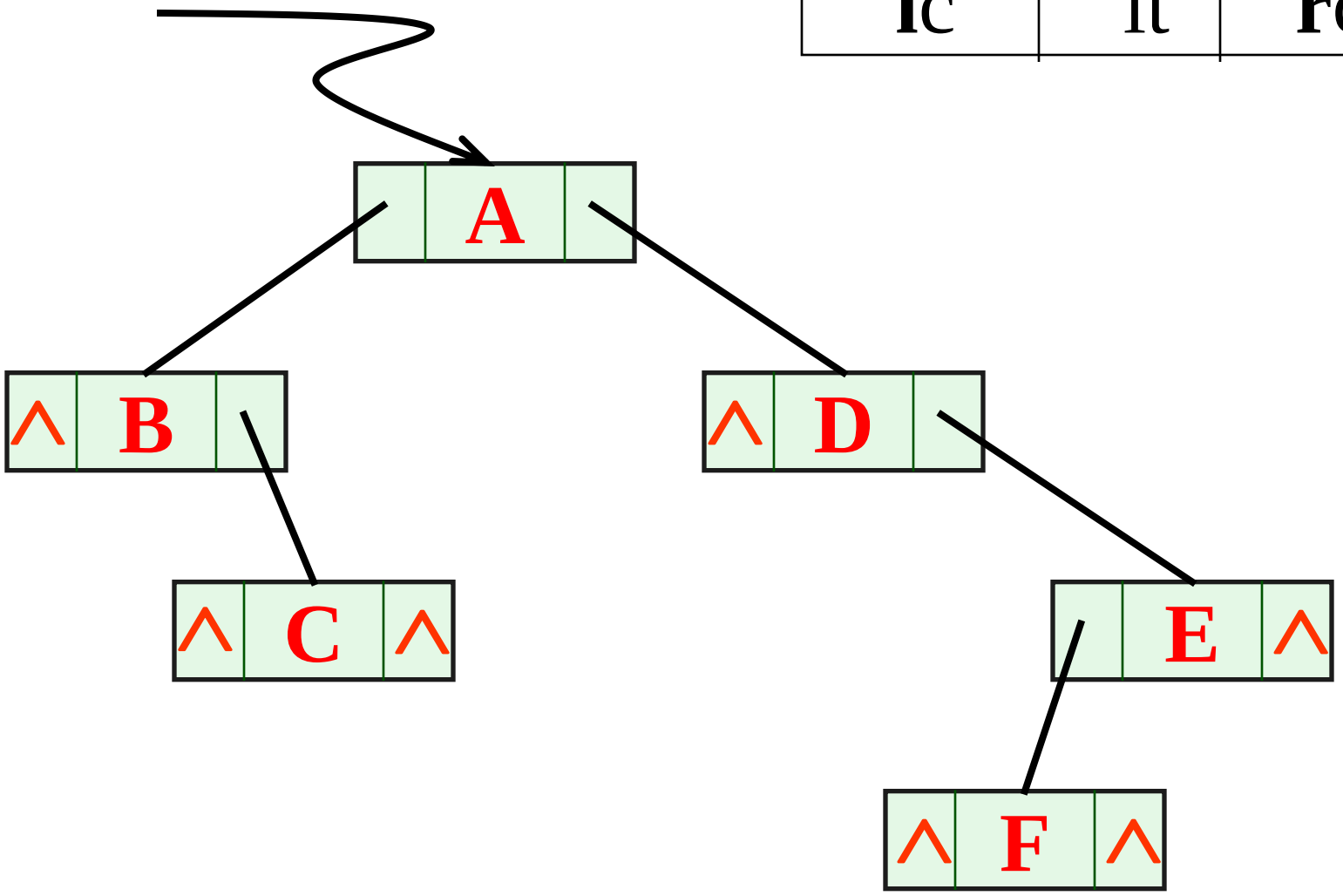


好处：运算方便;问题：空指针太多

二叉链表

结点结构:





二叉树的存储

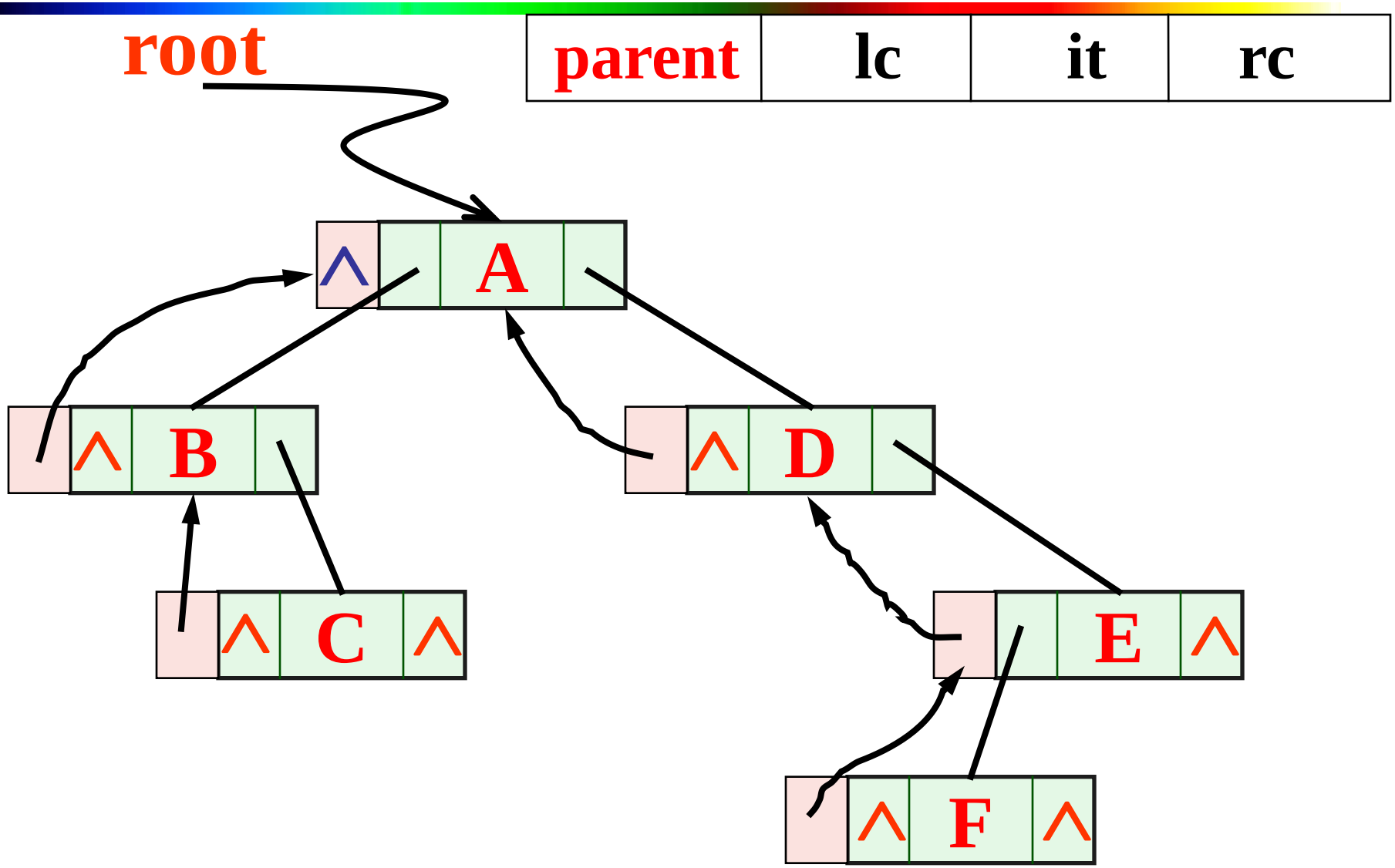
- 带父指针的三重链表

在某些经常要回溯到父结点的应用中很有效。

```
class BSTNode:public BinNode<E> {
  private:
    Key k;
    E it;
    BSTNode* lc;
    BSTNode* rc;
     BSTNode* father;
  …};
```

三重链表

结点结构:



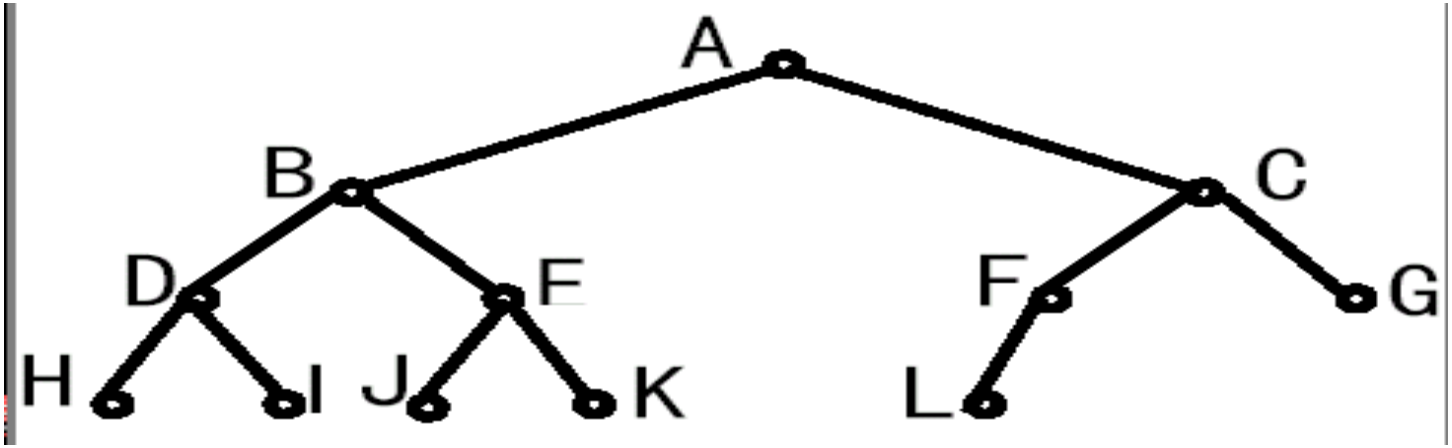
使用数组实现完全二叉树

完全二叉树的顺序存储

ABCDEFGHIJKL

按照二叉树的层次周游次序存储在一个数组中

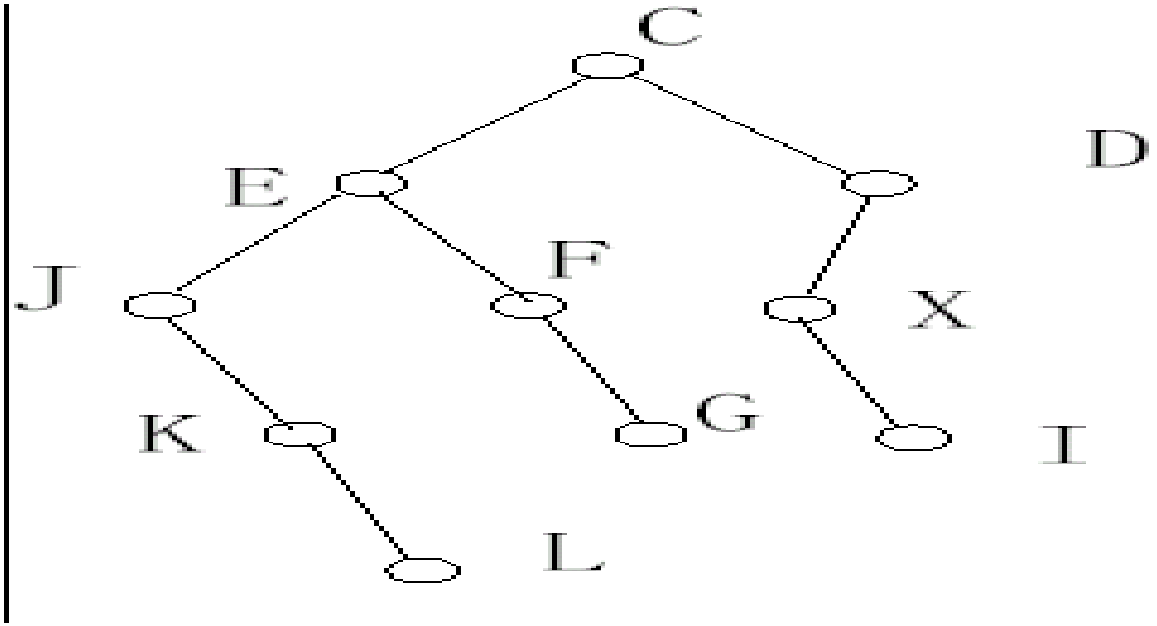
简单，省空间



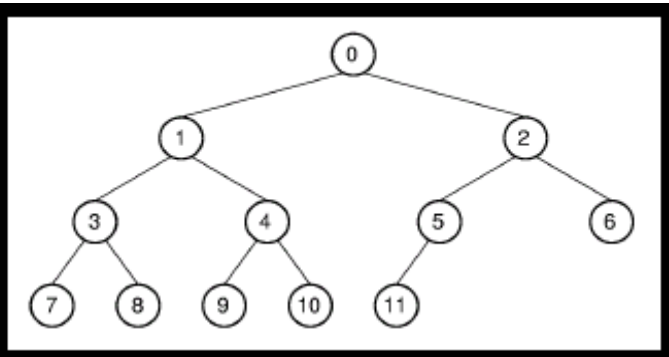
顺序存储

- 非完全二叉树在置空值而转换为完全二叉树存储

CEDJFX//K/G/I////L



完全二叉树的下标对应关系



Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

完全二叉树的下标公式

公式中r表示结点的索引， n表示二叉树结点总数。

Parent(r) = $\lfloor (r-1)/2 \rfloor$ **，当r!=0时。**

Leftchild(r) =2r +1，当2r+1<n时。

Rightchild(r) = 2r+ 2 ，当2r+2 < n 时。

Leftsibling(r) =r-1，当r为偶数且0<=r<=n-1。

Rightsibling(r) =r+1，当r为奇数且r+1<n。

二叉检索树

定义：二叉检索树或者为空，或者是满足下列条件的非空二叉树：

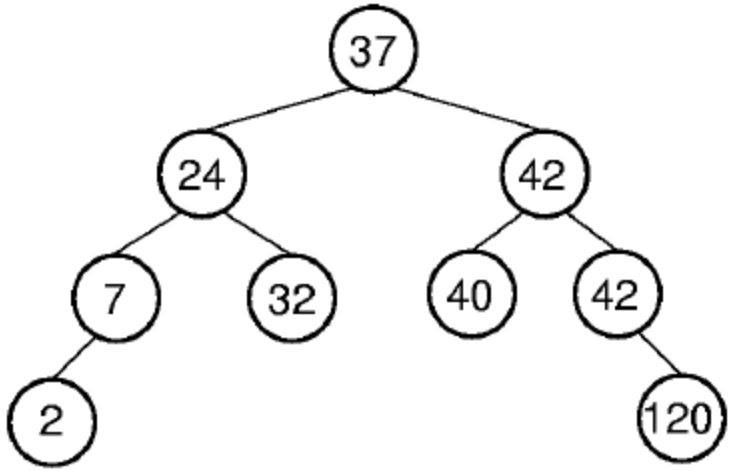
(1)若它的左子树非空，则左子树上所有结点的值均小于根结点的值；

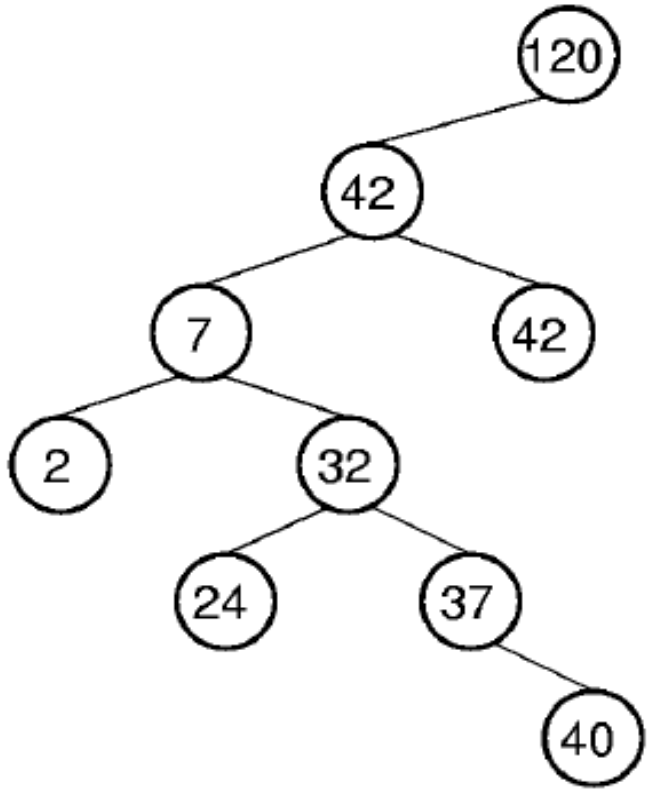
(2)它的右子树非空，则右子树上所有结点的值均大于或等于根结点的值；

(3)左右子树本身又各是一棵二叉检索树。

性质：按照中序周游将各结点打印出来，将得到按照由小到大的排列。

BST图示





检索

二叉检索树的效率就在于只需检索二个子树之一。

- 从根结点开始，在二叉检索树中检索值K。如果根结点储存的值为K，则检索结束。
- 如果K小于根结点的值，则只需检索左子树
- 如果K大于根结点的值,就只检索右子树
- 这个过程一直持续到K被找到或者我们遇上了一个树叶。
- 如果遇上树叶仍没有发现K，那么K就不在该二叉检索树中。

二叉检索树类定义

```
template<class Key,class Elem,class KEComp,class EEComp>
class BST:public Dictionary<Key,Elem,KEComp,EEComp> {
private:
  BinNode<Elem>* root;  //Root of the BST
  int nodecount;  //Number of nodes in the BST
  void clearhelp(BinNode <Elem>* );
  BinNode<Elem>* inserthelp(BinNode<Elem>*,const Elem&);
  BinNode<Elem>* deletemin(BinNode <Elem>*,BinNode<Elem>*&);
  BinNode<Elem>* removehelp(BinNode<Elem>*, const Key&,
                                        BinNode<Elem>*&);
  bool findhelp(BinNode<Elem>*,  const Key&, Elem&)  const;
  void printhelp(BinNode <Elem>* , int)  const;
```

二叉检索树类定义

```
public:
  BST() { root = NULL;nodecount=0; } //Contructor
  ~BST() { clearhelp(root); }                  //Destructor
  void clear() {clearhelp(root);root=NULL; nodecount=0;}
  bool insert(const Elem&e){root=inserthelp(root,e);
       nodecount++; return true;}
  bool remove(const Key& k,Elem& e) {
     BinNode<Elem>* t=NULL;
     root = removehelp(root,K,t);
     if(t==NULL) return false;   //Nothing done
     e = t->val();  nodecount--;delete t;return true;}
  bool removeAny(Elem& e){  //Delete min value
     if(root==NULL) return false;  //Empty tree
     BinNode<Elem>* t;
     root=deletemin(root,t); e=t->val( );delete t;
     nodecount--;return true;}
  bool find(const Key& K, Elem& e) const {return findhelp(root,K,e);}
  int size( ) {return nodecount; }
  void print( ) const{
      if (root==NULL) cout<<"The BST is empty.\n";
     else printhelp(root,0);}
};
```

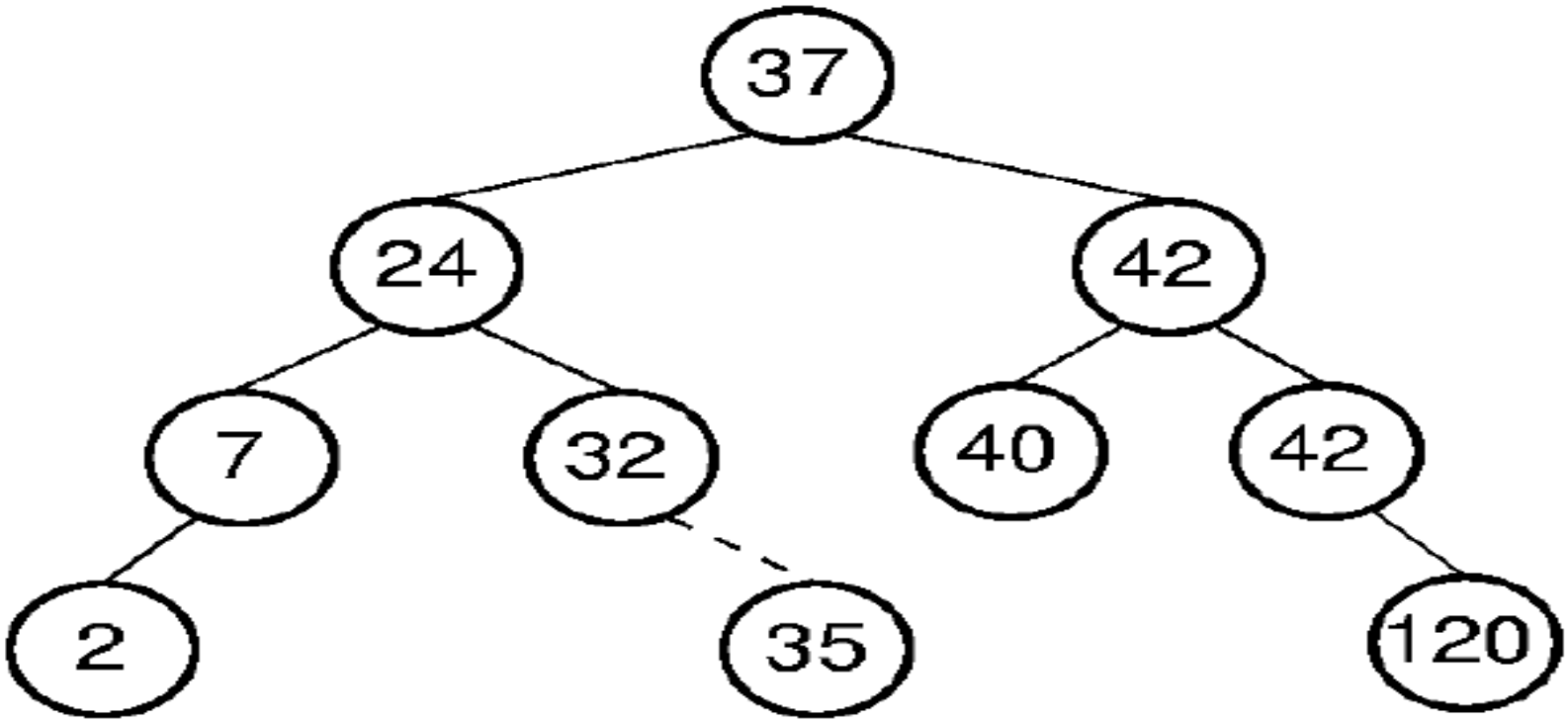
检索

```
template<class Key,class Elem,class KEComp,class EEComp>
bool BST<Key,Elem,KEComp,EEComp>::findhelp(
        BinNode<Elem>* subroot, const Key& k,Elem& e) const {
  if (subroot == NULL) return false;  //Empty tree
  else if (KEComp::lt(K, subroot->val( ))  //Check left
     return findhelp(subroot->left( ), K, e);
  else if (KEComp::gt(K, subroot->val( ))   //Check roght
     return findhelp(subroot->right(), K, e);
   else {e=subroot->val( ); return true;} //Found it
}
```

插入

```
template <class Key, class Elem,class KEComp, class
EEComp>
BinNode<Elem>* BST<Key,Elem,KEComp,EEComp>::
 inserthelp(BinNode<Elem>* subroot,const Elem& val) {
 if (subroot == NULL) // Empty: create node
   return new BinNodePtr<Elem>(val,NULL,NULL);
 if (EEComp::lt(val, subroot->val()))
   subroot->setLeft(inserthelp(subroot->left(), val));
 else subroot->setRight(
        inserthelp(subroot->right(), val));
 // Return subtree with node inserted
 return subroot;
}
```

BST插入图示



删除

从二叉检索树中删除一个任意的结点R，首先必须找到R，接着将它从二叉树中删除掉。

如果R是一个叶结点(没有儿子)，那么只要将R的父结点指向它的指针改为NULL就可以了。

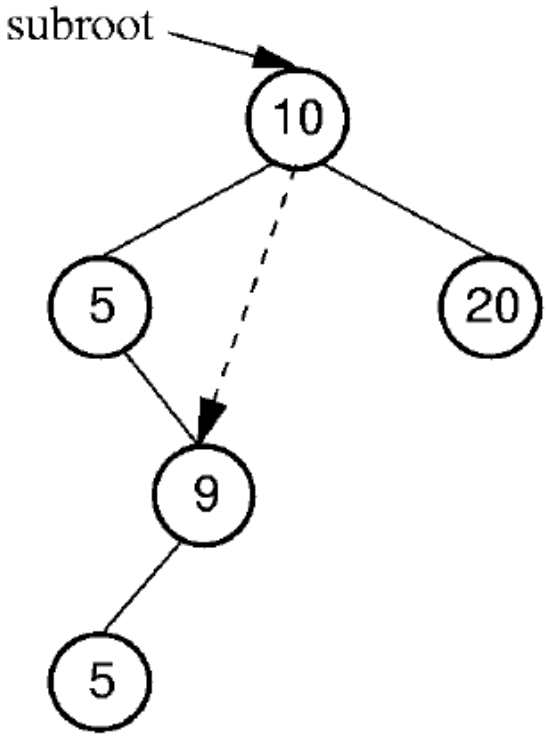
如果R是一个分支结点，我们就不能简单地删除这个结点，因为这样做会破坏树的连通性。

如果R只有一个儿子，就将R的父结点指向它的指针改为指向R的子结点就可以了。

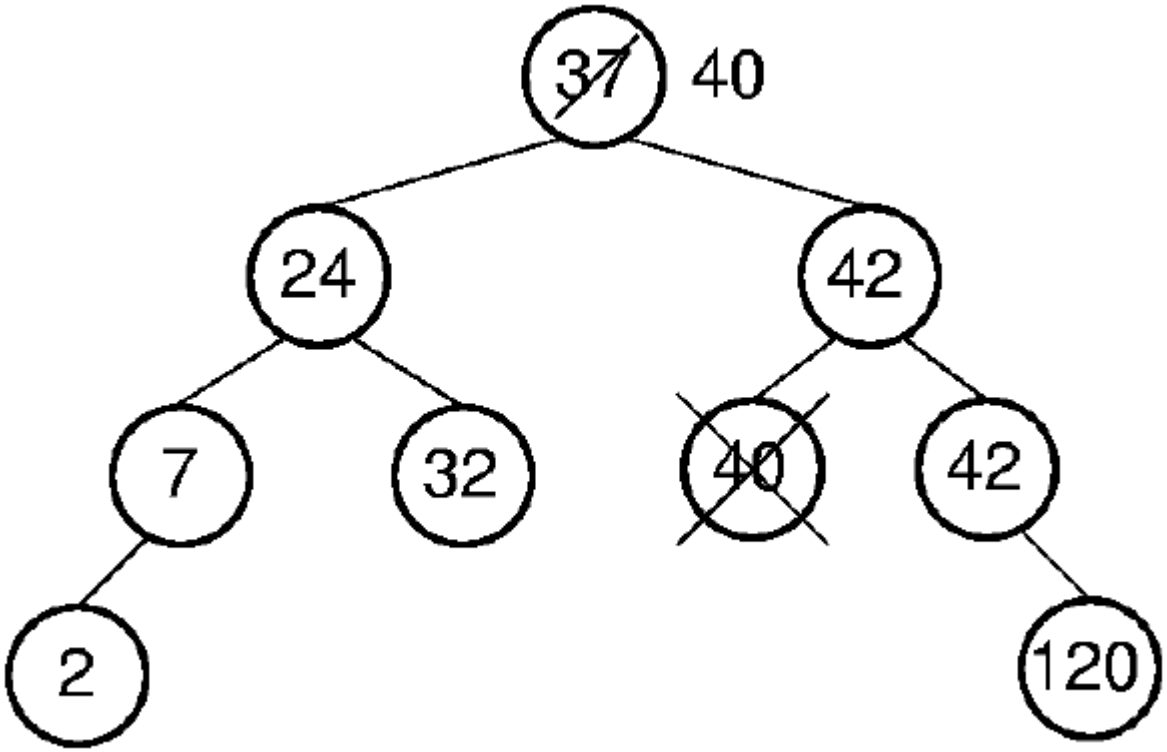
如果R有两个儿子，为了保持二叉检索树的性质，可以用R的中序后继结点来代替它。

删除子树中最小值图示

```
template<class Key,class Elem,class KEComp,class EEComp>
BinNode<Elem>*BST<Key,Elem,KEComp,EEComp>::
deletemin (BinNode<Elem>* subroot,BinNode<Elem>*& min){
  if (subroot->left() == NULL) { //Found min
     min subroot;
     return subroot->right();
  else { // Continue left
     subroot->setLeft(
     deletemin(subroot->left(),min));
     return subroot;
  }
}
```



删除右子树中最小值结点



BST Remove (1)

```
template<class Key,class Elem,class KEComp,class EEComp>
BinNode<Elem>*BST<Key,Elem,KEComp,EEComp>::
removehelp(BinNode<Elem>* subroot, const Key& K,
                    BinNode<Elem>*& t){
  if (subroot == NULL) return NULL;       //Val is not in tree
  else if (KEComp::lt(K,subroot->val( )))  //check left
     subroot->setLeft(removehelp(subroot->left(), k, t));
  else if (KEComp::gt(K,subroot->val( )))  //check right
     subroot->setRight(removehelp(subroot->right(), k, t));
```

BST Remove (2)

```
else {            // Found it: remove it
    BinNode<Elem>* temp ;
    t=subroot;
    if (subroot->left() == NULL)   //Only a right child
        subroot = subroot->right( );//So point to right
    else if (subroot->right() == NULL)   subroot=subroot->left( );
    else {  // Both children are non-empty
        subroot->setRight(deletemin(subroot->right( ),temp));
        Elem te=subroot->val( );
        subroot->setVal(temp->val( ));
        temp->setval( te);
        t=temp; }
      }
  return subroot;
}
```

堆与优先队列

定义：

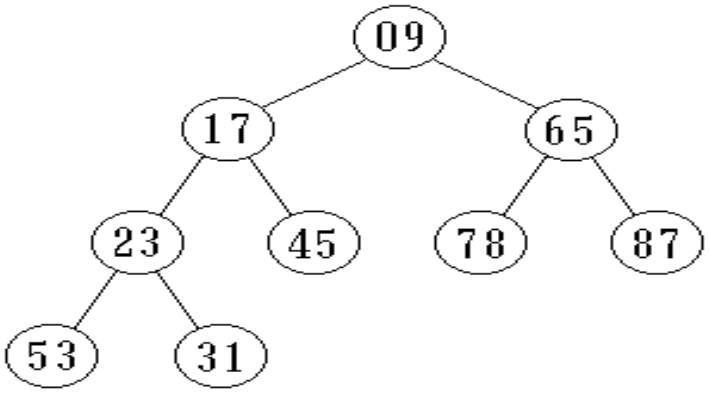
对于一个关键码序列{ K_0, K_1, …, K_{n-1}}，如果满足$K_i \geqslant K_{2i+1}$，$K_i \geqslant K_{2i+2}$，(i=0, 1, …, n/2-1)，则称其为堆，而且这是最大值堆。

性质：

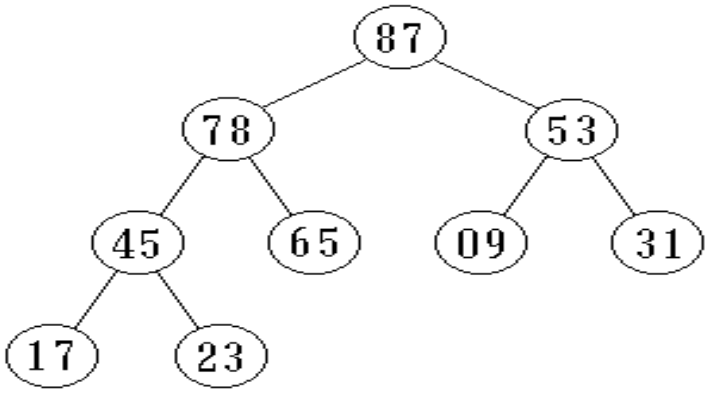
是任意一个结点的值都大于或者等于其任意一个子结点存储的值。由于根结点包含大于或等于其子结点的值，而其子结点又依次大于或等于各自子结点的值，所以根结点存储着该树所有结点中的最大值。

最小值堆

最小值堆(min-heap)的性质是每一个结点存储的值都小于或等于其子结点存储的值。由于根结点包含小于或等于其子结点的值，而其子结点又依次小于或等于各自子结点的值，所以根结点存储了该树所有结点的最小值。



（a）最小堆



（b）最大堆

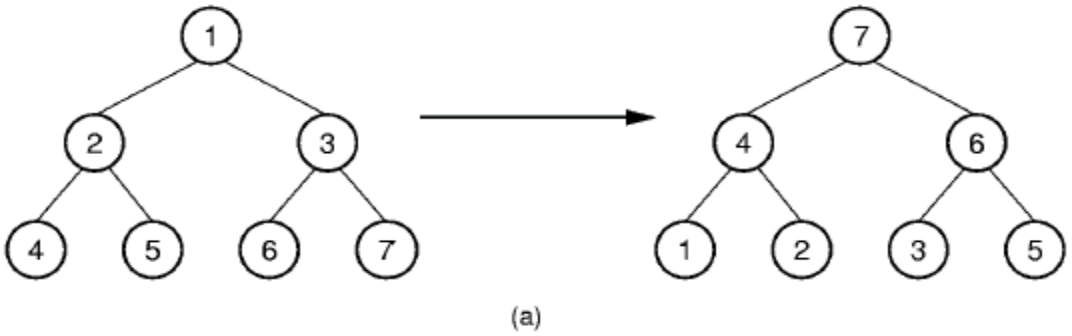
最大值堆的实现

```
template<class Elem,class Comp> class maxheap{
private:
  Elem* Heap;   // Pointer to the heap array
  int size;     // Maximum size of the heap
  int n;        // Number of elems now in heap
  void siftdown(int); // Put element in place
public:
  maxheap(Elem* h, int num, int max)
    {Heap=h; n=num; size=max; buildHeap(); }
  int heapsize() const
     { return n;}
  bool isLeaf(int pos) const
     { return (pos>=n/2) && (pos<n);  }
```

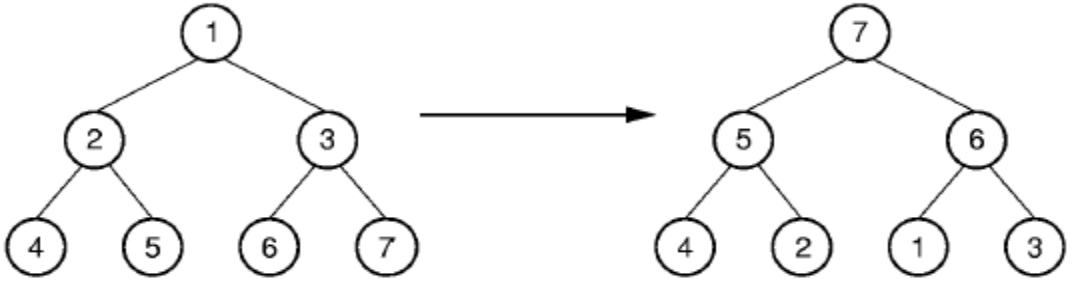
最大值堆的实现

```
  int leftchild(int pos) const
    { return 2*pos+1; }
  int rightchild(int pos) const
    { return 2*pos+2; }
  int parent(int pos) const
    { return (pos-1)/2; }
  bool insert(const Elem&);
  bool removemax(Elem&);
  bool remove(int , Elem&);
  void buildHeap()
    { for (int i=n/2-1;i>=0;i--)  shiftdown(i); }
};
```

建堆图示



(a)



(b)

(a) (4-2) (4-1) (2-1) (5-2) (5-4) (6-3) (6-5) (7-5) (7-6)
(b) (7-3), (5-2), (7-1), (6-1)

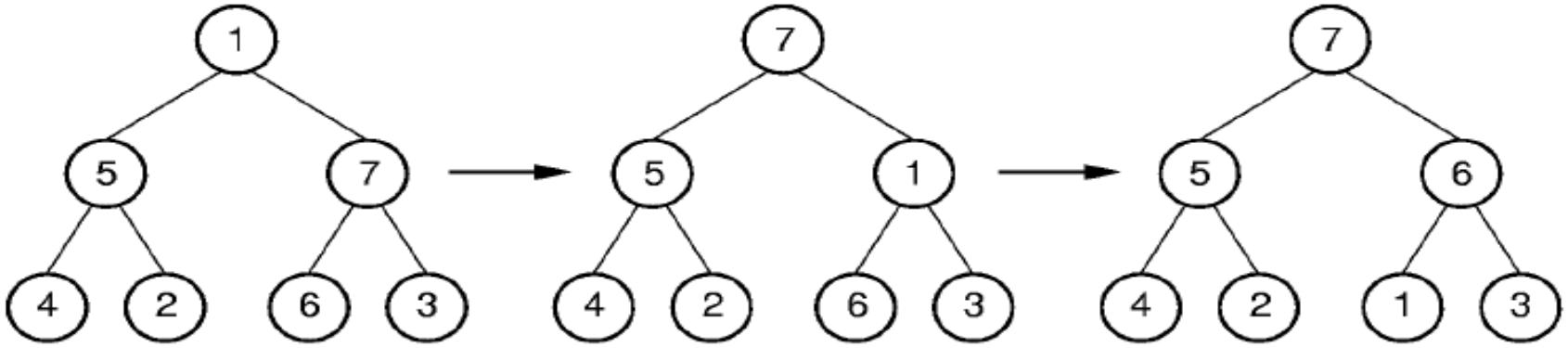
堆的形成

- 不必将值一个个地插入堆中，通过交换形成堆。
- 假设根的左、右子树都已是堆，并且根的元素名为R。能这种情况下，有两种可能：

  (1) R的值大于或等于其两个子女，此时堆已完成

  (2) R的值小于其某一个或全部两个子女的值，此时R应与两个子女中值较大的一个交换，结果得到一个堆，除非R仍然小于其新子女的一个或全部的两个。这种情况下，我们只需简单地继续这种将R“拉下来来”的过程，直至到达某一个层使它大于它的子女，或者它成了叶结点。

Shiftdown操作



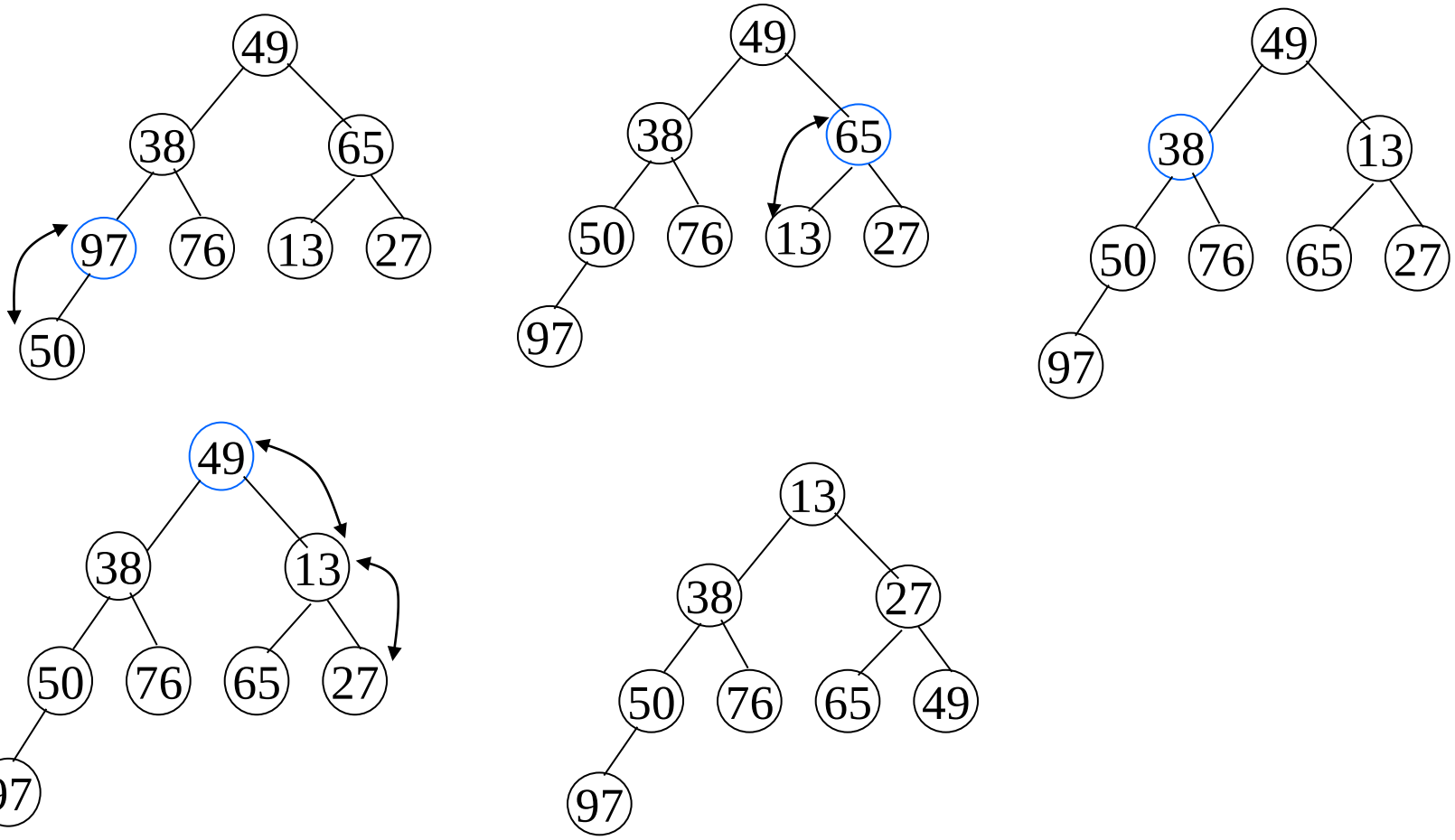
(a) (b) (c)

从堆的第一个分支结点（**heap[n/2-1]**，自底向上逐步把以各分支结点为根的子树调整成堆。

当调整到某一个位置**heap[pos]**时，由于其子树都已经是堆，

- 如果它也正好满足堆定义，则不用调整；
- 否则取其子树结点中较大的与**heap[pos]**交换；
- 交换后此子树根可能不满足堆定义（但其子树还是堆）；这样，可以继续一层层交换下去，最多到叶停止（过筛）

举例


49
38
65
97
76
13
27
50
49
38
65
50
76
13
27
97
49
38
13
50
76
65
27
97
13
38
27
50
76
65
49
97

Siftdown

```
template<class Elem,class Comp>
void maxheap<Elem,Comp>::siftdown(int pos) {
  while (!isLeaf(pos)) {  //stop if pos is a leaf
    int j = leftchild(pos);
    int rc = rightchild(pos);
    if ((rc<n) && Comp::lt(Heap[j],Heap[rc]))
      j = rc;  set j to greater child's value
    if (!Comp::lt(Heap[pos], Heap[j])) return;
    swap(Heap, pos, j);
    pos = j;  //move down
     }
}
```

Remove Max Value

```
template<class Elem,class Comp>
bool maxheap<Elem,Comp>:: removemax(Elem& it) {
  if(n==0)  return false; // Heap is empty
  swap(Heap, 0, --n);   // Swap max with end
  if (n != 0) siftdown(0);
  it= Heap[n];
  return  true;
 }
```

Insert element

```
template<class Elem,class Comp>
bool maxheap<Elem,Comp>:: insert(const Elem& val) {
  if(n>=size) return false; // Heap is full
  int curr=n++;
  Heap[curr]=val;  //start at end of heap
//now sift up until curr's parent>curr
  while ((curr!=0) && (Comp::gt(Heap[curr],Heap[parent(curr)])))  {
      swap(Heap, curr, parent(curr));
      curr=parent(curr);
    }
return  true;
 }
```

建堆操作的效率

n个结点的堆，高度d = floor($\log_2 n$ + 1)。根为第0层，则第i层结点个数为2^i，

考虑一个元素在堆中向下移动的距离。

- 大约一半的结点深度为d-1，不移动（叶）。
- 四分之一的结点深度为d-2，而它们至多能向下移动一层。
- 树中每向上一层，结点的数目为前一层的一半，而子树高度加一。因而元素移动的最大距离的总数为

$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} = o(n)$$

所以，这种算法时间代价为O(n)。

由于堆有log n层深，插入结点、删除普通元素和删除最大元素的平均时间代价和最差时间代价都是O(log n)。

Huffman编码树

1.固定长度编码

- 设所有代码都等长，则表示n个不同的代码需要$\log_2 n$位称为固定长度编码(a fixed-length coding scheme)。
- ASCII 码就是一种固定长度编码。
- 如果每个字符的使用频率相等的话，固定长度编码是空间效率最高的方法。

2.数据压缩和不等长编码

- 频率不等的字符

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

数据压缩和不等长编码

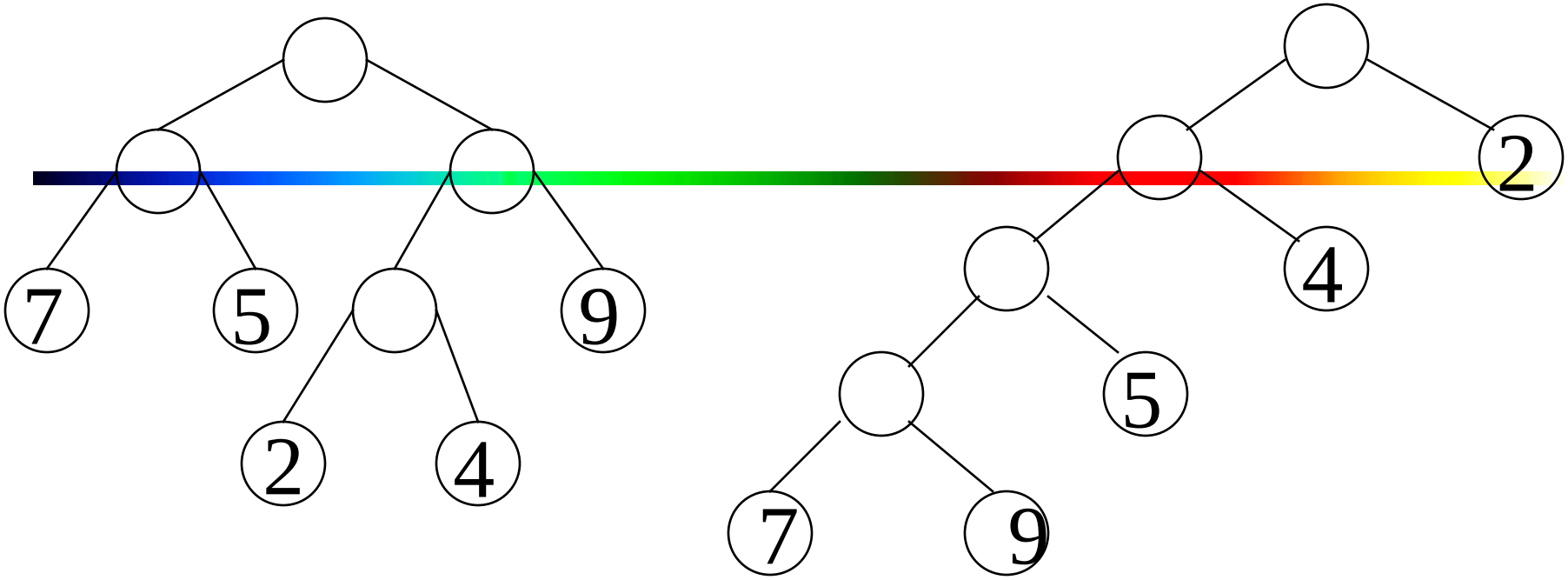
- 可以利用字母的出现频率来编码，使得经常出现的字母的编码较短，反之不常出现的字母编码较长。
- 数据压缩既能节省磁盘空间，又能提高运算速度。（外存时空权衡的规则）
- 不等长编码是今天广泛使用的文件压缩技术的核心
- Huffman 编码是最简单的文件压缩技术，它给出了这种编码方法的思想。

建立Huffman编码树

- 对于n个字符K_0，K_1，…，K_{n-1}，们它的使用频率分别为w_0，w_1，…，w_{n-1}，请给出它们的编码，使得总编码效率最高。
- 定义一个树叶的带权路径长度(Weighted path length)为权乘以它的路径长度(即树叶的深度）。
- 这个问题其实就是要求给出一个具有n个外部结点的扩充二叉树， 该二叉树每个外部结点Ki有一个wi与之对应，作为该外部结点的权

这个扩充二叉树的叶结点带权外部路径长度总和 $\sum_{i=0}^{n-1} w_i \cdot l_i$ 最小(注意不管内部结点，也不用有序)。

权越大的叶结点离根越近；如果某个叶的权较小，可能就会离根较远。



WPL(T)=

7×2+5×2+2×3+

4×3+9×2

=60

WPL(T)=

7×4+9×4+5×3+

4×2+2×1

=89

如何构造最优树

(赫夫曼算法)以二叉树为例：

(1) 根据给定的 n 个权值 $\{w_1, w_2, \ldots, w_n\}$，

构造 n 棵二叉树的集合

$$F = \{T_1, T_2, \ldots, T_n\},$$

其中每棵二叉树中均只含一个带权值为 w_i 的根结点，其左、右子树为空树；

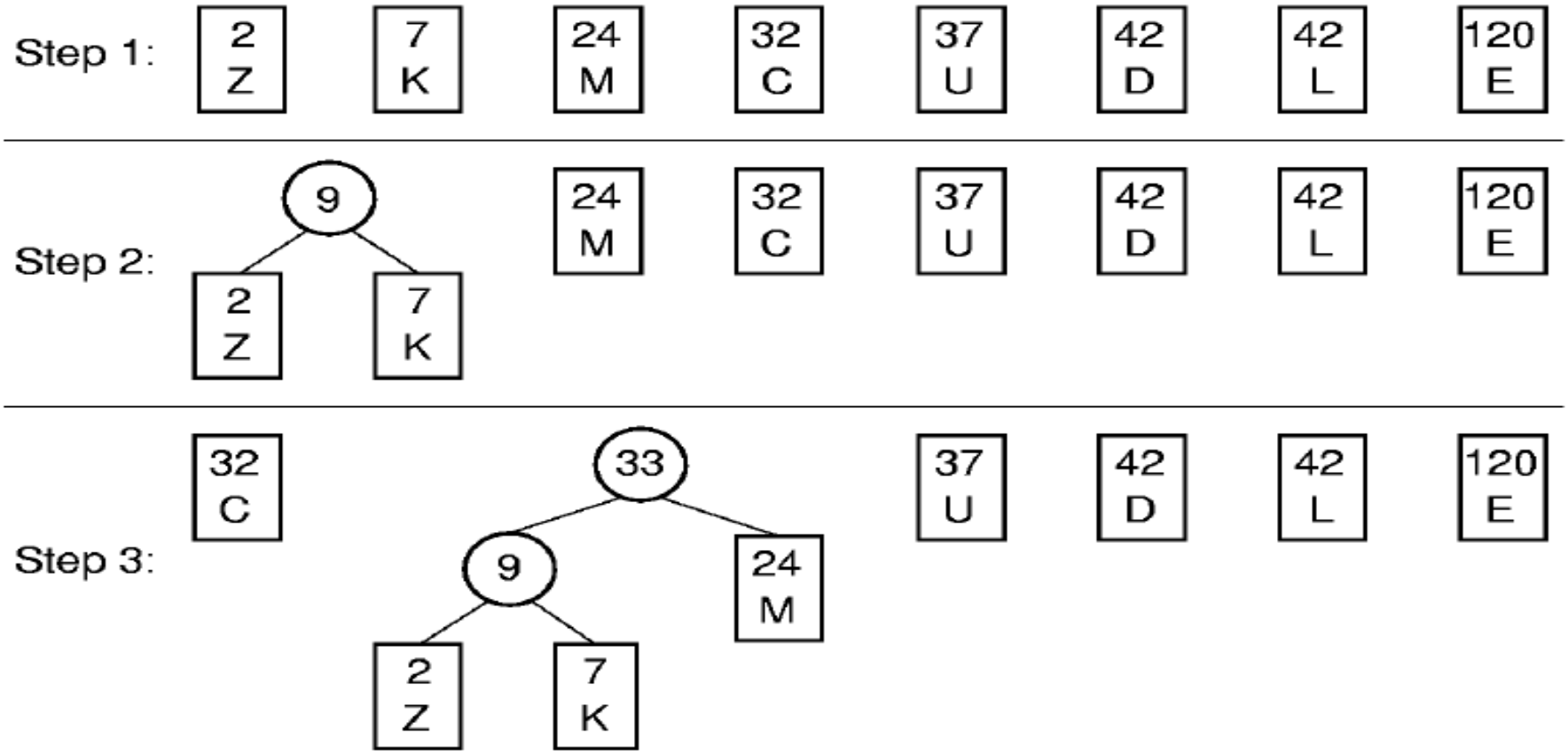
如何构造最优树

(2) 在F中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；

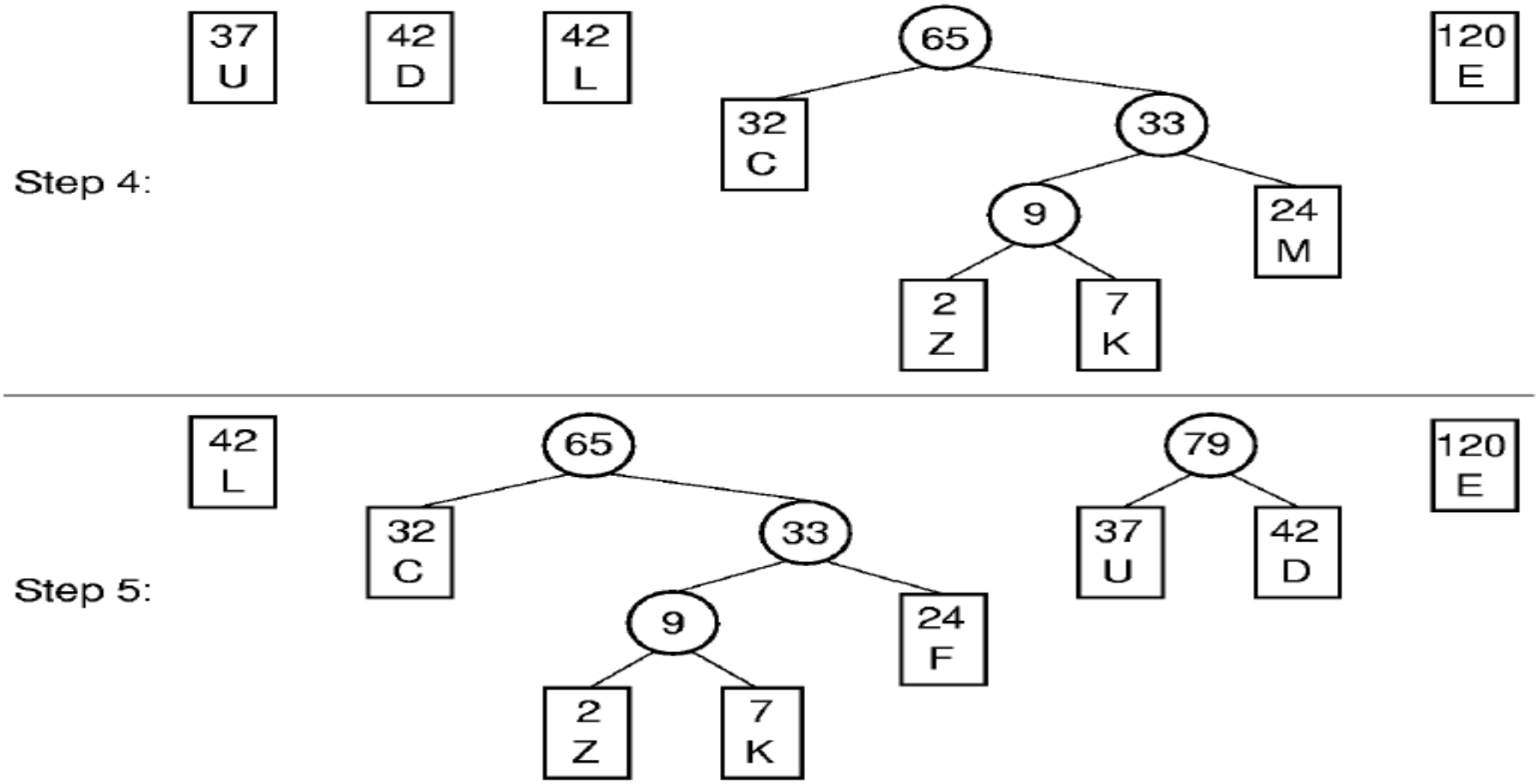
(3) 从F中删去这两棵树，同时加入刚生成的新树；

(4) 重复(2)和(3)两步，直至F中只含一棵树为止。

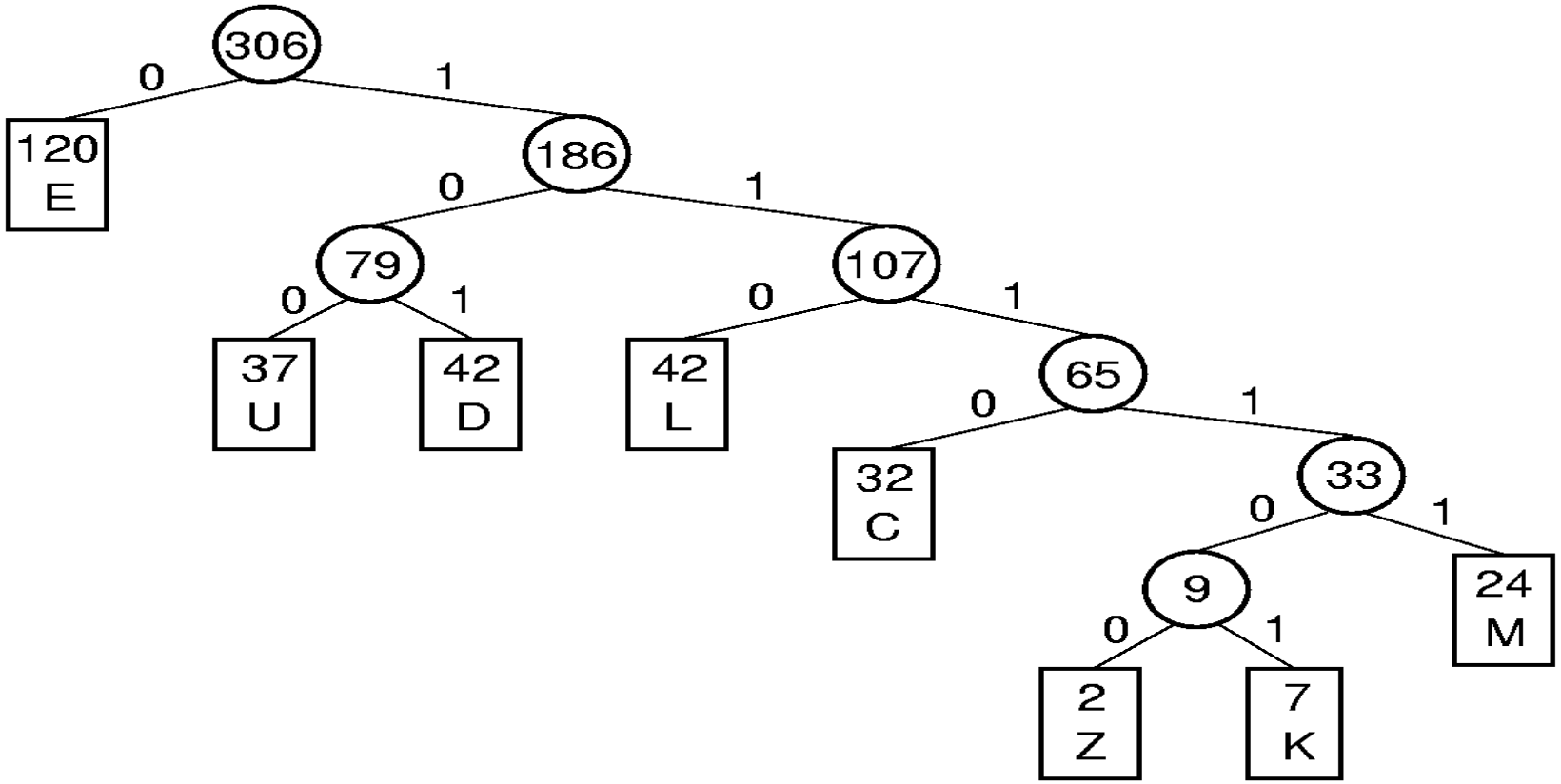
Huffman建树图示



Huffman建树图示



Huffman建树图示



Haffman树结点的实现

```
template <class Elem>
class HuffNode {        //Node abstract base class
public:
  virtual int weight() = 0;
  virtual bool isLeaf() = 0;
  virtual HuffNode* left() const = 0;
  virtual void setLeft(HuffNode*) = 0;
  virtual HuffNode* right() const = 0;
  virtual void setRight(HuffNode*) = 0;
};
```

Haffman树结点的实现

```
template <typename Elem>   // Leaf node subclass
class LeafNode : public HuffNode<Elem> {
private:
  FreqPair<Elem>* it;  //Frequency pair
public:
  LeafNode(const Elem& val, int freq)   // Constructor
    { it = new FreqPair<Elem>(val,freq); }
  int weight() { return it->weight(); }
  FreqPair<Elem>* val()  { return it; }
  bool isLeaf() { return true; }
  virtual HuffNode* left() const { return NULL; }
  virtual void setLeft(HuffNode*) { }
  virtual HuffNode* right() const { return NULL; }
  virtual void setRight(HuffNode*) {}
};
```

Haffman树结点的实现

```
template <class Elem>  //Internal node subclass
class IntlNode : public HuffNode<Elem> {
private:
  HuffNode<Elem>* lc;    // Left child
  HuffNode<Elem>* rc;   //Right child
  int wgt;                          //Subtree weight
public:
  IntlNode(HuffNode<Elem>* l , HuffNode<Elem>* r)
     { wgt = l->weight( ) + r->weight( ) ; lc = l; rc = r; }
  int weight( ) { return wgt; }    //Return frequency
  bool isLeaf() { return false; }
  HuffNode<Elem>* left ( ) const { return lc; }
  void setLeft (HuffNode<Elem>* b) { lc = (HuffNode*)b; }
  HuffNode<Elem>* right( ) const { return rc; }
  void setRight (HuffNode<Elem>* b) { rc = (HuffNode*)b; }
};
```

元素/频率对的类申明

```
template <class Elem>
class FreqPair { // An element/frequency pair
private:
  Elem it;       // An element of some sort
  int freq;      // Frequency for the element
public:
  FreqPair(const Elem& e, int f) // Constructor
    { it = e; freq = f; }
  ~FreqPair() { }                // Destructor
  int weight() { return freq; }  // Return the weight
  Elem& val() { return it; }     // Return the element
};
```

Haffman树

```
template <class Elem>
class HuffTree {
private:
  HuffNode<Elem>* theRoot;
public:
  HuffTree(Elem& val, int freq)
    { theRoot = new LeafNode<Elem>(val, freq); }
  HuffTree(HuffTree<Elem>* l, HuffTree<Elem>* r)
    { theRoot = new IntlNode<Elem>(l->root(), r->root()); }
  ~HuffTree() {}
  HuffNode<Elem>* root() { return theRoot; }
  int weight() { return theRoot->weight(); }
};
```

Haffman树

```
//Compare two Huffman trees by total weight
template <class Elem> class HHcompare{
public:
  static bool lt(HuffTree<Elem>* x, HuffTree<Elem>* y)
        { return x->weight( ) < y->weight( ); }
  static bool eq(HuffTree<Elem>* x, HuffTree<Elem>* y)
        { return x->weight( ) == y->weight( ); }
  static bool gt(HuffTree<Elem>* x, HuffTree<Elem>* y)
        { return x->weight( ) > y->weight( ); }
};
```

Haffman树

```
template <class Elem>HuffTree<Elem>*
buildHuff(SLList<HuffTree<Elem>*, HHCompare<Elem> >*fl ) {
    HuffTree<Elem> *temp1, *temp2, *temp3;
    for(fl->setStart( ); fl->leftLength( )+fl->rightLenght( )>1;
          fl->setStart( )){ //while at least two items left
       fl->remove(temp1);  //Pull first two trees
       fl->remove(temp2);  //off the list
       temp3 = new HuffTree<Elem> (temp1, temp2 );
       fl->insert(temp3);  //put the new tree back on  list
       delete temp1;  //Must delete the remnants
       delete templ2;  //of the trees we created
    }
   retuen temp3;
}
```