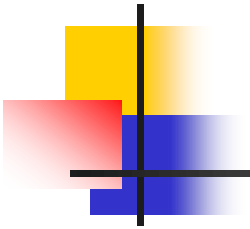
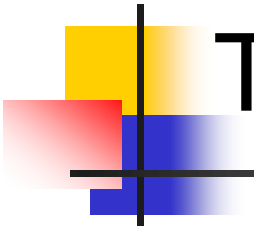


Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal



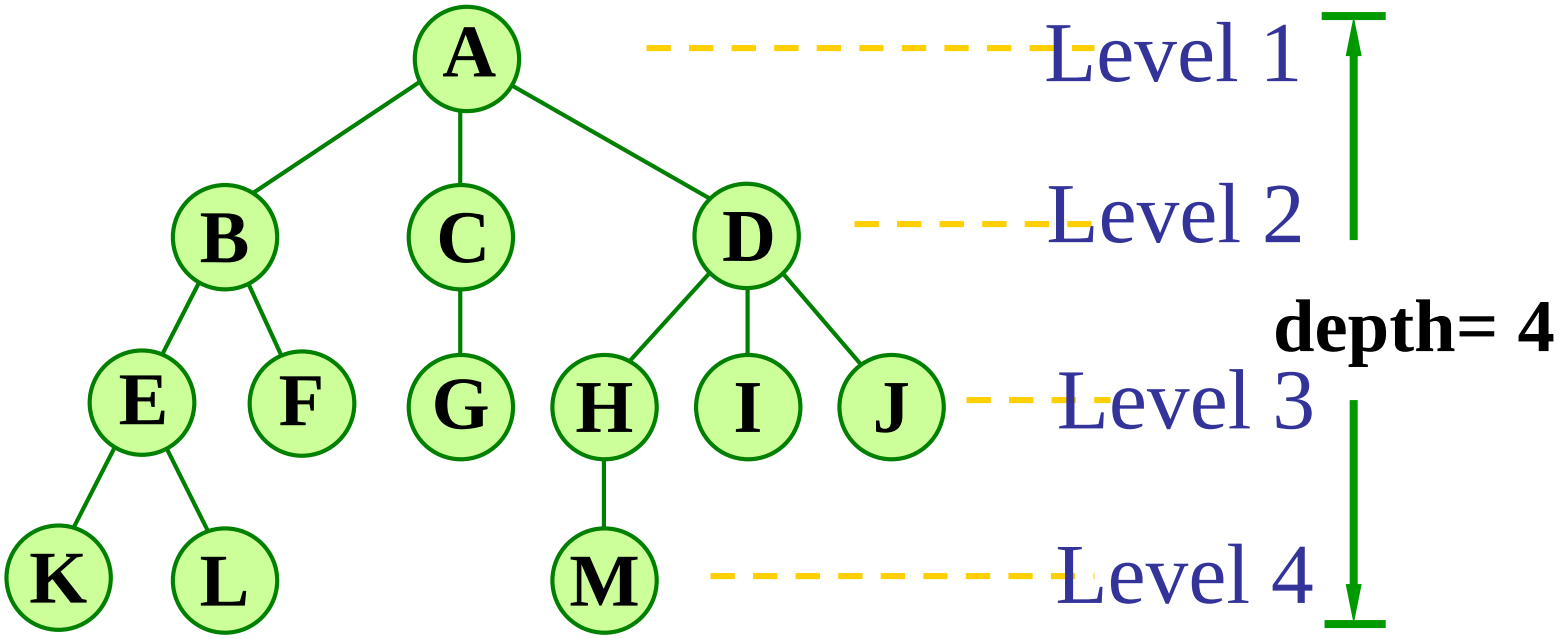
Terminology

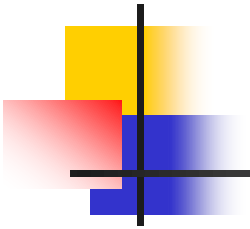
- Definition: A tree is a finite set of one or more nodes such that

(1) There is a specially designated node called root.

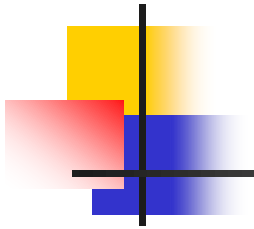
(2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called subtrees of the root.

Terminology

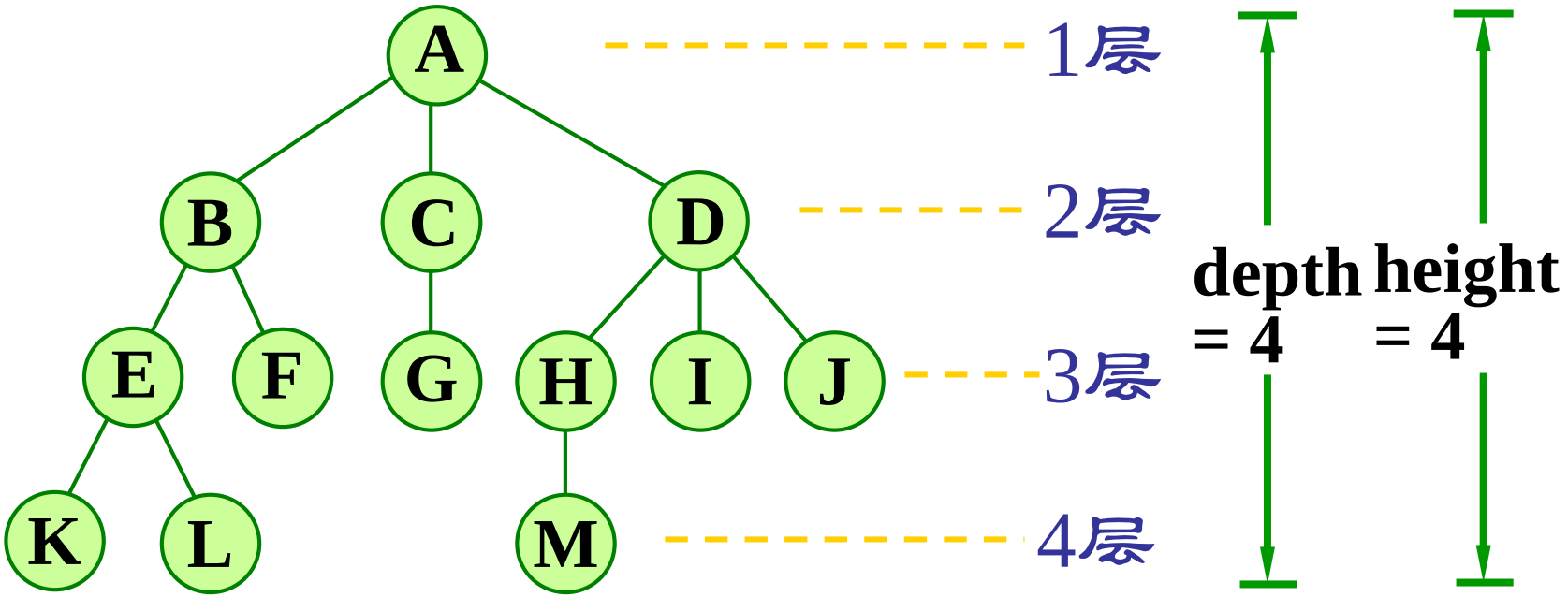


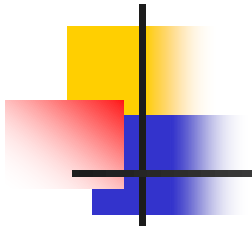


- 子女(child)
  - 若结点的子树非空，结点子树的根即为该结点的子女
- 双亲(parent)
  - 若结点有子女，该结点是子女双亲
- 兄弟(sibling)
  - 同一结点的子女互称为兄弟
- 祖先(ancestor)
  - 某结点到根结点的路径上的各个结点都是该结点的祖先
- 子孙(descendant)
  - 某结点的所有下属结点，都是该结点的子孙



- 度(degree)
  - 结点的子女个数，即为该结点的度
  - 树的度：树中各个结点的度的最大值
- 分支结点(branch) / 非终端结点
  - 度不为0的结点
- 叶结点(leaf)
  - 度为0的结点
- 高度(height)
  - 规定：叶结点的高度为1
  - 其双亲结点的高度等于它的高度加1
- 树的高度
  - 等于根结点的高度，即根结点所有子女高度的最大值加1


A
B
C
D
E
F
G
H
I
J
K
L
M
1层
2层
3层
4层
depth = 4
height = 4



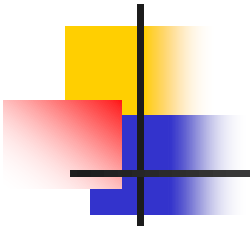
- 有序树
  - 树中结点的各棵子树 $T_0, T_1, \ldots$是有次序的
- 无序树
  - 树中结点的各棵子树之间的次序不重要，可以互相交换位置
- 森林(forest)
  - 森林是m（$m \geq 0$）棵树的集合

ADT of Trees

```
template <class T>
class Tree {
/*
对象: 树是由n (≥0) 个结点组成的有限集合。
    在类界面中的 position 是树中结点的地址。
    在顺序存储方式下是下标型, 在链表存储方式下是指针型。
    T 是树结点中存放数据的类型, 要求所有结点的数据类型一致
*/
public:
   Tree ();
   ~Tree ();
```

```
BuildRoot (const T& value);
  //建立树的根结点
position FirstChild(position p);
  //返回 p 第一个子女地址, 无子女返回 0
position NextSibling(position p);
  //返回 p 下一兄弟地址, 若无下一兄弟返回 0
position Parent(position p);
   //返回 p 双亲结点地址, 若 p 为根返回 0
T getData(position p);
  //返回结点 p 中存放的值
bool InsertChild(position p, T& value);
  //在结点 p 下插入值为 value 的新子女, 若插
  //入失败, 函数返回false, 否则返回true
```

```
    bool DeleteChild (position p, int i);
      //删除结点 p 的第 i 个子女及其全部子孙结
      //点, 若删除失败, 则返回false, 否则返回true
    void DeleteSubTree (position t);
      //删除以 t 为根结点的子树
    bool IsEmpty ();
      //判树空否, 若空则返回true, 否则返回false
    void Traversal (void (*visit)(position p));
      //遍历以 p 为根的子树
};
```

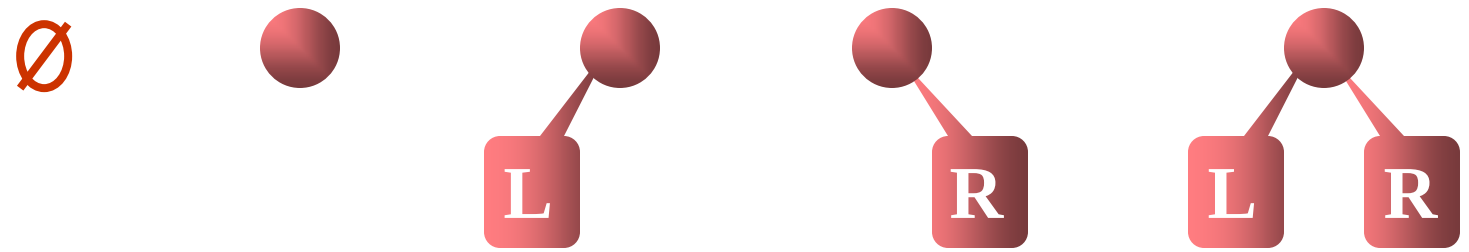


Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

Binary Trees

- A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.



Five Types of Binary Trees

Properties of Binary Trees

- Lemma 5.2 [Maximum number of nodes]:
- (1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

Proof.

当i=1时， $2^{1-1} = 2^0 = 1$ ，结论成立；

假定对所有j (1=<j<i)，结论成立，即第j层最多有2^{j-1}个结点；

则，在第i-1层最多有2^{i-2}个结点；

由于，二叉树每个结点最多有2个子女，

因而，第i层上的最大结点数为第i-1层上最大结点数的2倍: $2*2^{i-2} = 2^{i-1}$

- (2) The maximum number of nodes in a binary tree of depth k is 2^k-1, k≥1.

Proof.

因为每一层最少要有1个结点，

因此，最少结点数为 k。

最多结点个数借助性质1用求等比级数前k项和的公式:

$$2^0+2^1+2^2+\ldots+2^{k-1}=2^k-1$$

- Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]:
- For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2+1$.

Proof：设度为 1 的结点有 n_1 个，总结点数为n，

总边数为e，则根据二叉树的定义，

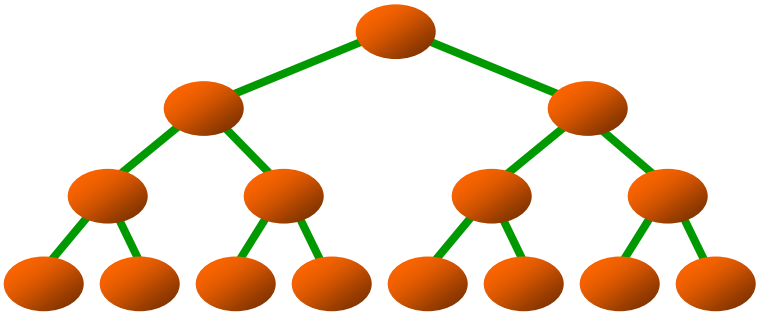
$$n = n_0+n_1+n_2 \qquad e = 2n_2+n_1 = n-1$$

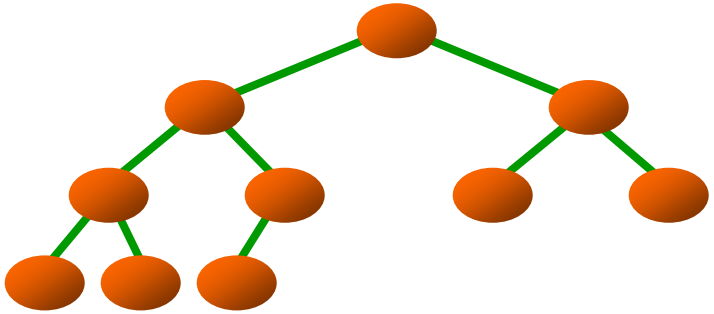
因此，有 $2n_2+n_1 = n_0+n_1+n_2-1$

$$n_2 = n_0-1 \Rightarrow n_0 = n_2+1$$

- Full Binary Tree: : A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, $k \geq 0$.
- Complete Binary Tree

A binary tree with n nodes and depth k is complete iff its nodes corresponding to the nodes numbered from 1 to n in the full binary tree of depth k.





- The height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$

  设完全二叉树的深度为k，则有

  $$\underbrace{2^{k-1}-1}_{\text{上面}k-1\text{层结点数}} < n \leqslant \underbrace{2^{k}-1}_{\text{第}k\text{层的最大结点数}}$$

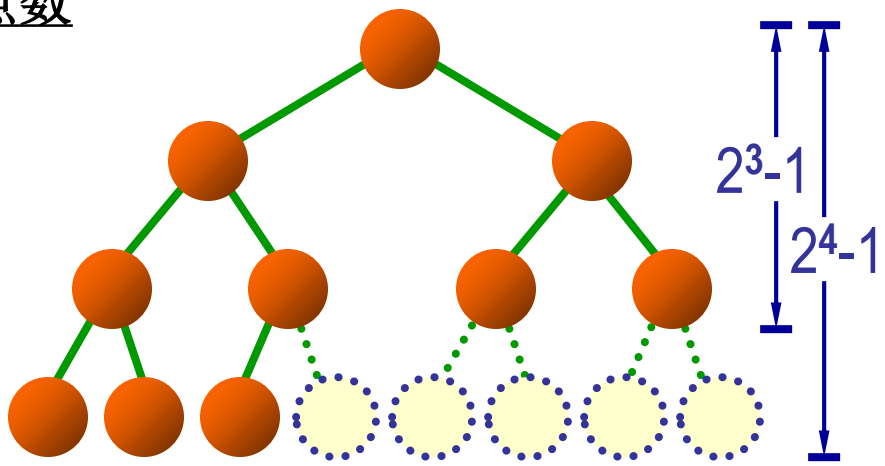
  变形 $2^{k-1} < n+1 \leqslant 2^{k}$

  取对数

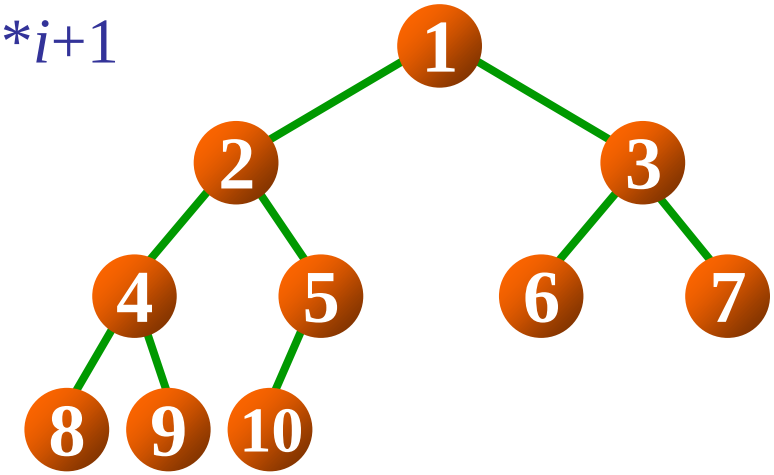
  $k-1 < \log_2(n+1) \leqslant k$

  有

  $\lceil \log_2(n+1) \rceil = k$

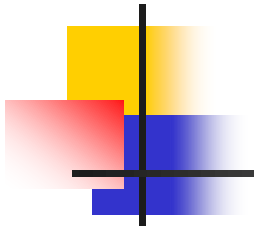


- 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号1, 2, …, n，则有以下关系：
  - ✓ 若$i = 1$, 则 i 无双亲
  - ✓ 若$i > 1$, 则 i 的双亲为$\lfloor i / 2 \rfloor$
  - ✓ 若$2*i <= n$, 则 i 的左子女为 $2*i$,
    若$2*i+1 <= n$, 则 i 的右子女为$2*i+1$
  - ✓ 若 i 为奇数, 且$i \neq 1$,
    则其左兄弟为$i-1$,
  - ✓ 若 i 为偶数, 且$i \neq n$,
    则其右兄弟为$i+1$



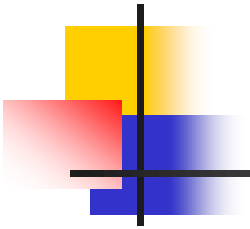
ADT of Binary Trees

```
template <class T>
class BinaryTree {
//对象: 结点的有限集合, 二叉树是有序树
public:
    BinaryTree ();                              //构造函数
    BinaryTree (BinTreeNode<T> *lch,
                BinTreeNode<T> *rch, T item);
      //构造函数, 以item为根, lch和rch为左、右子
      //树构造一棵二叉树
    int Height ();                              //求树深度或高度
    int Size ();                          //求树中结点个数
```



```
BinTreeNode<T> *Parent (BinTreeNode<T> *t);//求结点 t 的双亲
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t);
                                    //求结点 t 的左子女
BinTreeNode<T> *RightChild (BinTreeNode<T> *t);
                                          //求结点 t 的右子女

bool Insert (T item);              //在树中插入新元素
bool Remove (T item); //在树中删除元素
bool Find (T& item);               //判断item是否在树中
bool getData (T& item);            //取得结点数据
bool IsEmpty ();                   //判二叉树空否？
```

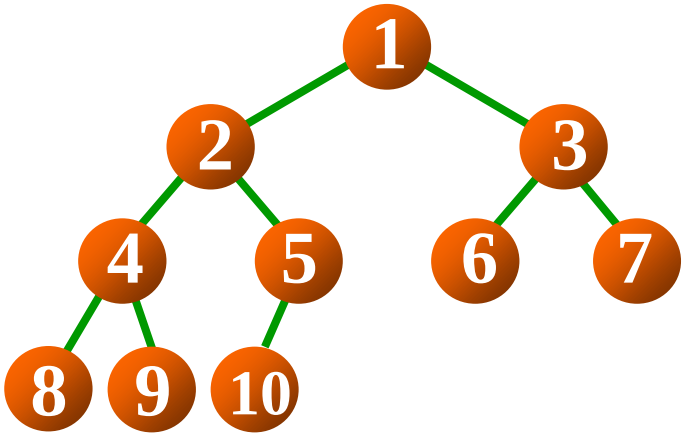
```
    BinTreeNode<T> *getRoot ();     //取根

    void preOrder (void (*visit) (BinTreeNode<T> *t));
        //前序遍历, visit是访问函数
    void inOrder (void (*visit) (BinTreeNode<T> *t));
        //中序遍历, visit是访问函数
    void postOrder (void (*visit) (BinTreeNode<T> *t));
        //后序遍历, (*visit)是访问函数
    void levelOrder (void (*visit)(BinTreeNode<T> *t));
        //层次序遍历, visit是访问函数
};
```



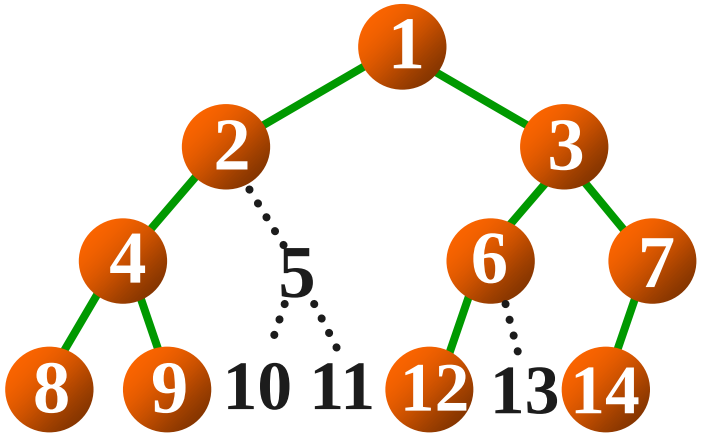
Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

Binary Tree Representations

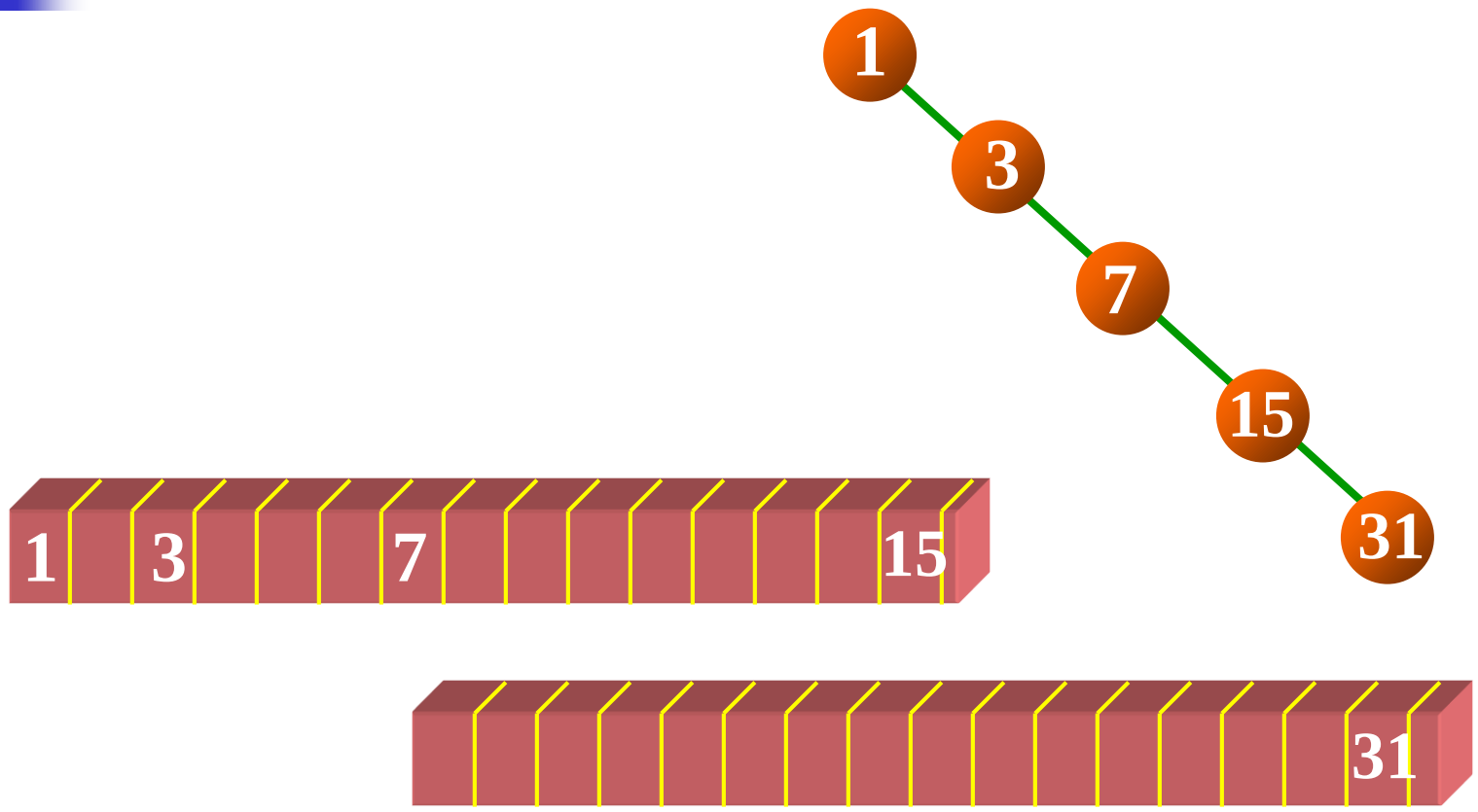


Array Representation of Complete Binary Trees



Array Representation of Binary Trees

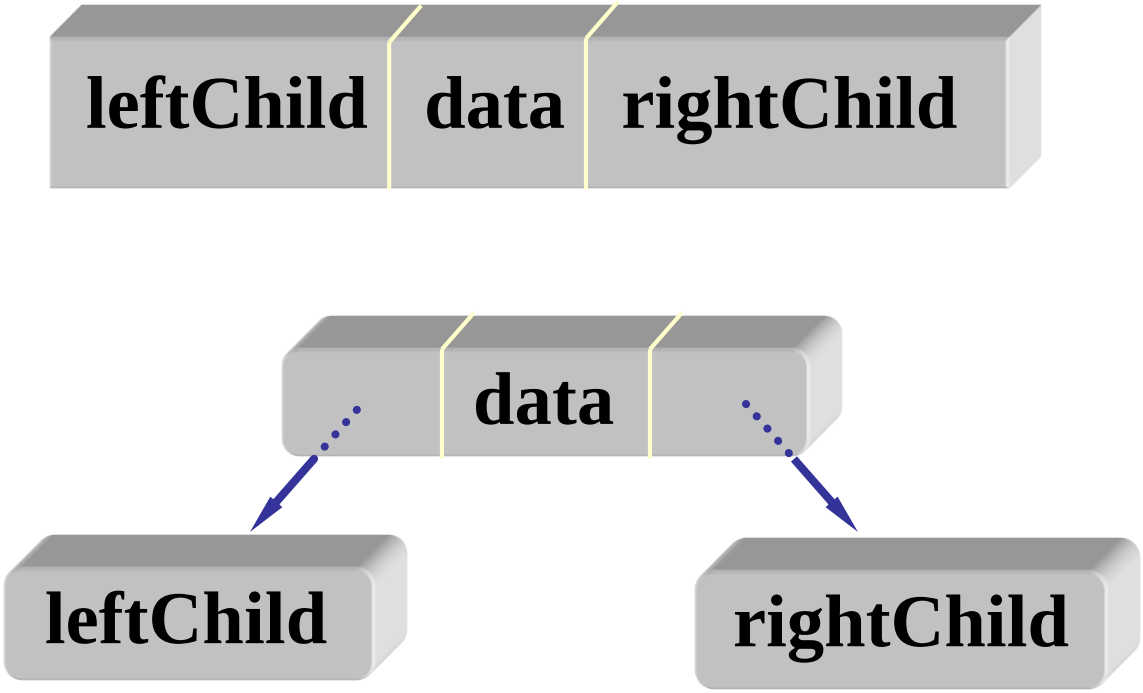
A Special Case



Linked Representation

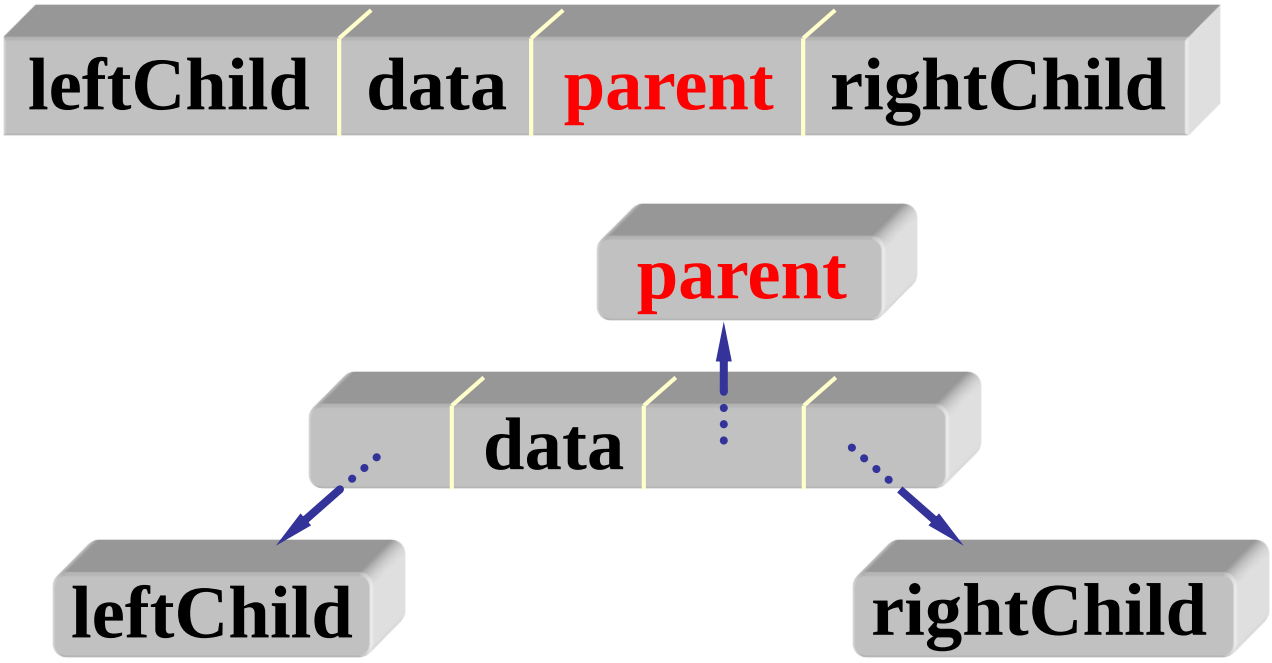
- 二叉树结点定义：每个结点有3个数据成员
  - data域存储结点数据
  - leftChild存放指向左子女的指针
  - rightChild存放指向右子女的指针

Linked Representation

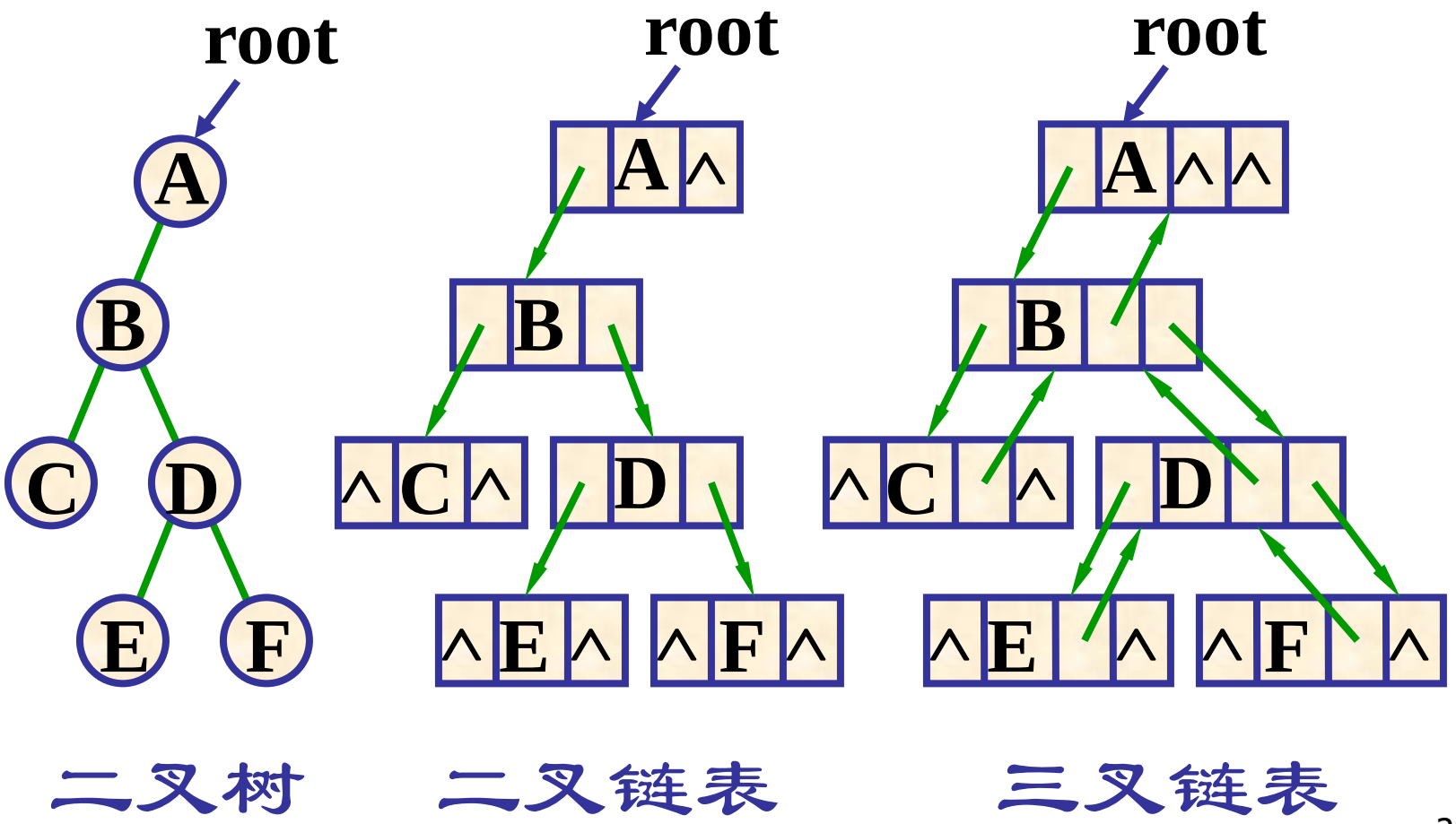


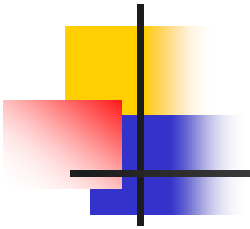
Linked Representation

- 每个结点：增加一个指向双亲的指针parent，使得查找双亲也很方便。



Examples





Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

Binary Tree Traversal

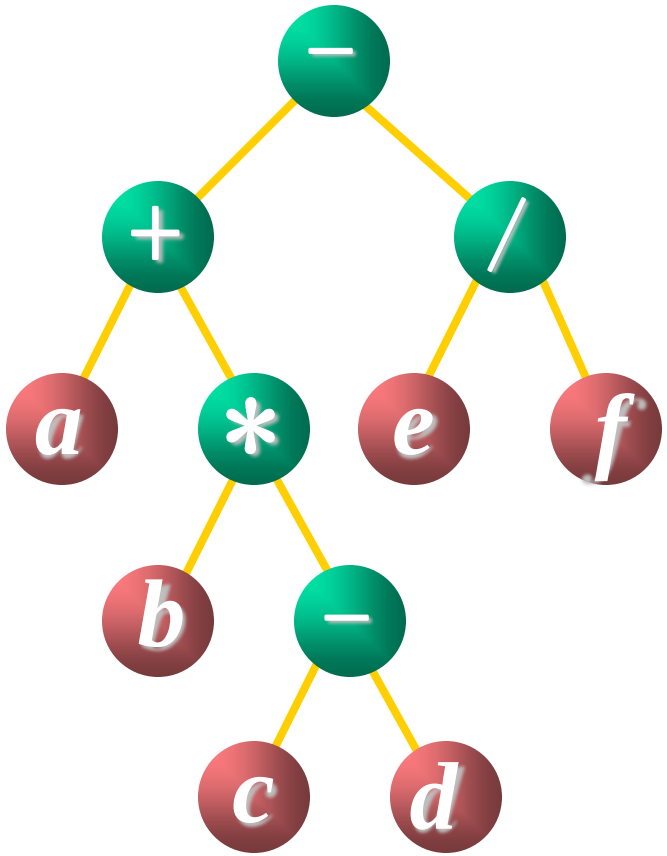
- 二叉树的遍历就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次。
- 设访问根结点记作 V
  遍历根的左子树记作 L
  遍历根的右子树记作 R
- 则可能的遍历次序有

前序	VLR	镜像	VRL
中序	LVR	镜像	RVL
后序	LRV	镜像	RLV

Inorder Traversal

中序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 中序遍历左子树 (L)；
  - 访问根结点 (V)；
  - 中序遍历右子树 (R)。



遍历结果

$a + b * c - d - e / f$

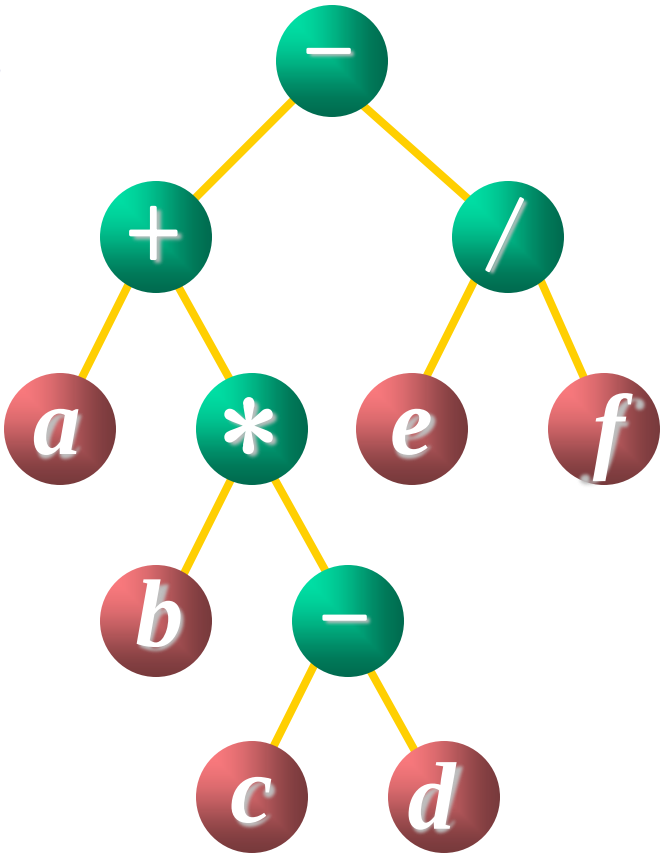
Algorithm of Inorder Traversal

```
template <class T>
void BinaryTree<T>::InOrder (BinTreeNode<T> *
    subTree, void (*visit) (BinTreeNode<T> *t))
{
     if (subTree != NULL)
     {
          InOrder (subTree->leftChild, visit);
                                        //遍历左子树
          visit (subTree);              //访问根结点
          InOrder (subTree->rightChild, visit);
                                        //遍历右子树
     }
}
```

Preorder Traversal

前序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 访问根结点 (V)；
  - 前序遍历左子树 (L)；
  - 前序遍历右子树 (R)。



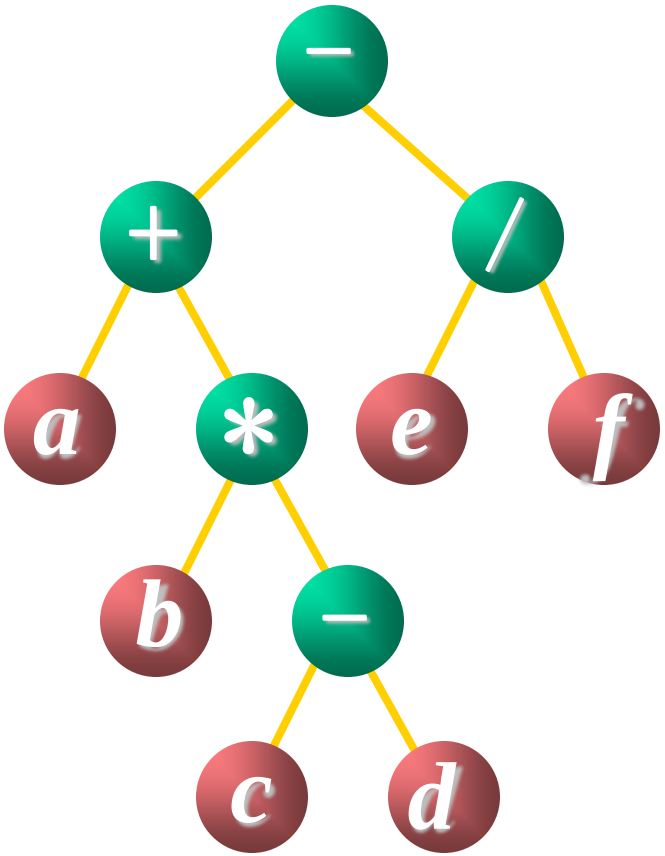
遍历结果

$$- + a * b - c\ d / e\ f$$

Postorder Traversal

后序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 后序遍历左子树 (L)；
  - 后序遍历右子树 (R)；
  - 访问根结点 (V)。

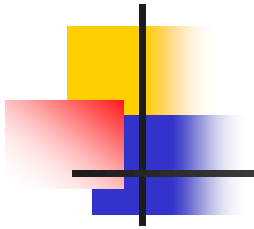


遍历结果

$a\ b\ c\ d\ -\ *\ +\ e\ f\ /\ -$

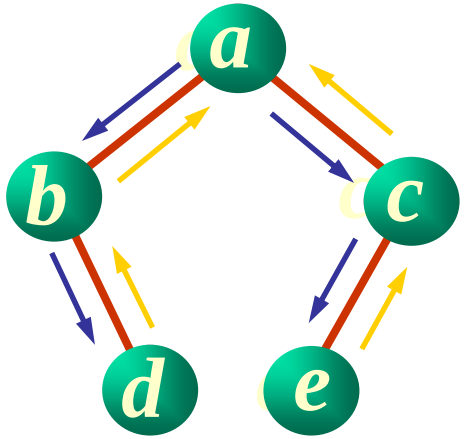
Algorithm of Postorder Traversal

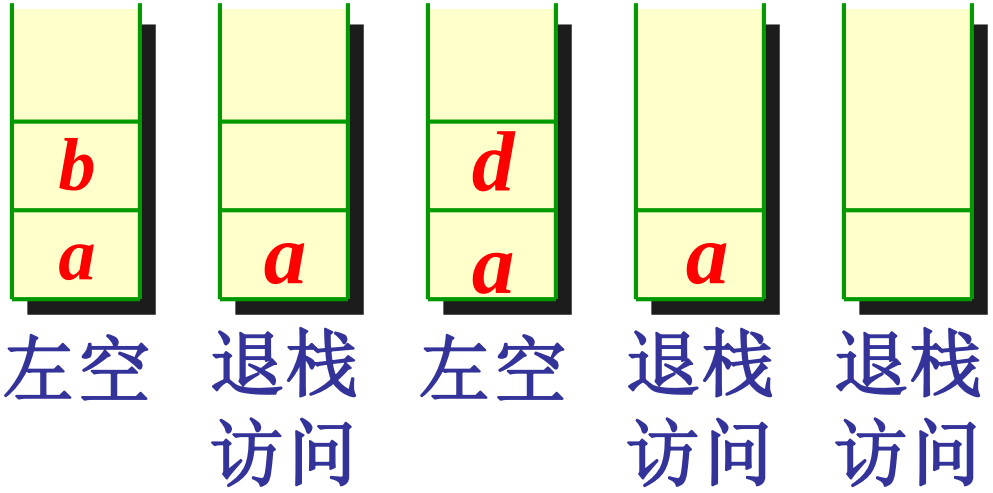
```
template <class T>
void BinaryTree<T>::PostOrder (BinTreeNode<T> * subTree,
    void (*visit) (BinTreeNode<T> *t )
{
    if (subTree != NULL )
    {
      PostOrder (subTree->leftChild, visit);    //遍历左子树
      PostOrder (subTree->rightChild, visit); //遍历右子树
      visit (subTree);              //访问根结点
    }
}
```

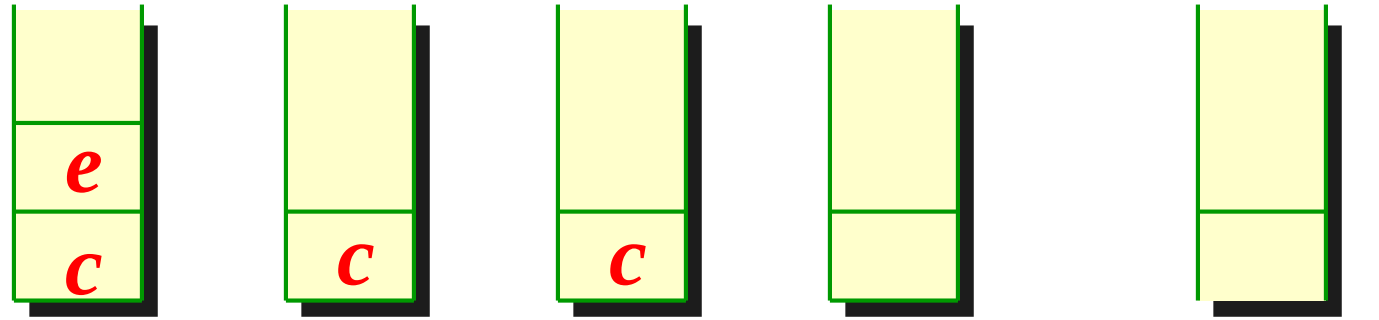


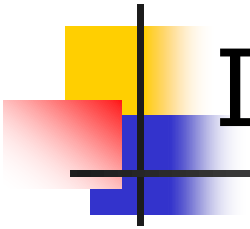
Traversal with a Stack

Inorder Traversal with a Stack







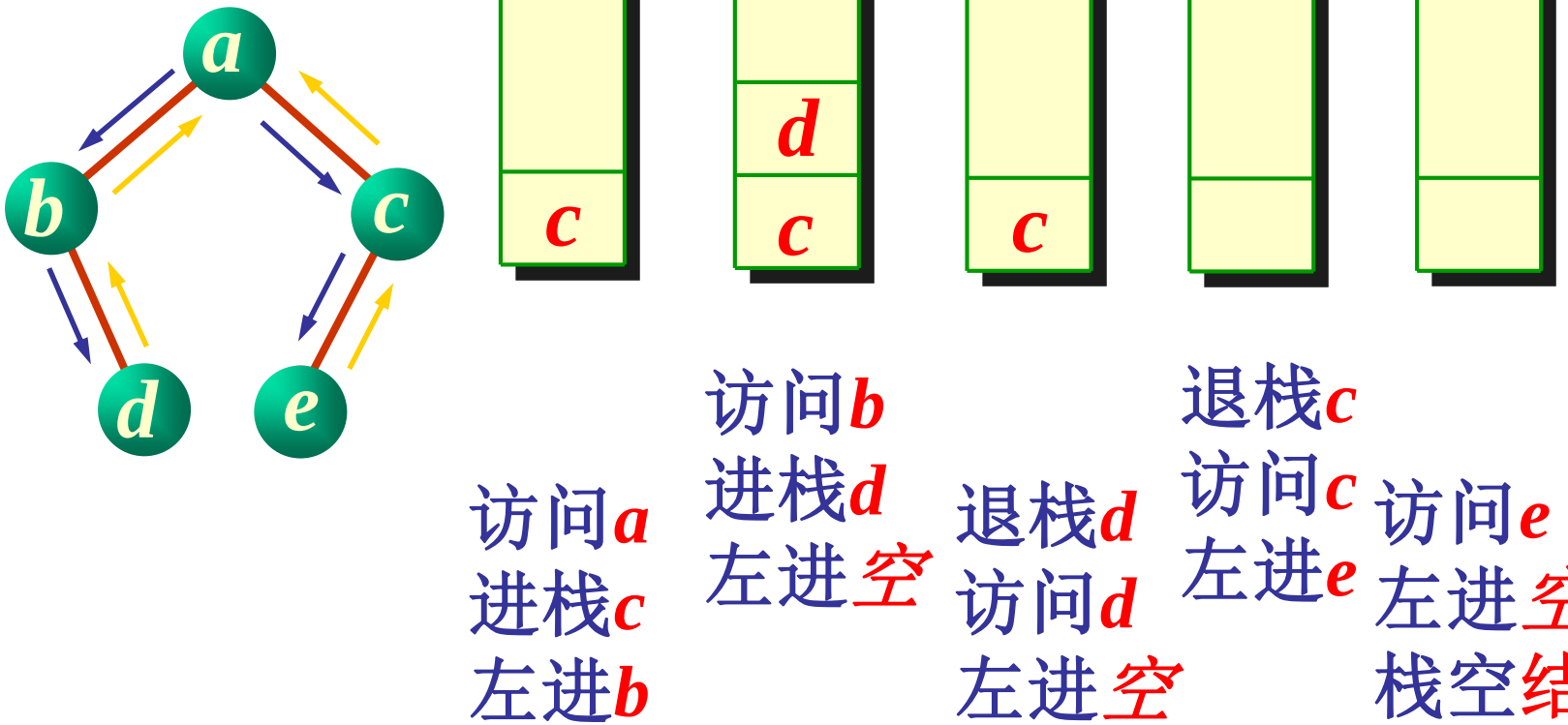


Inorder Traversal with a Stack

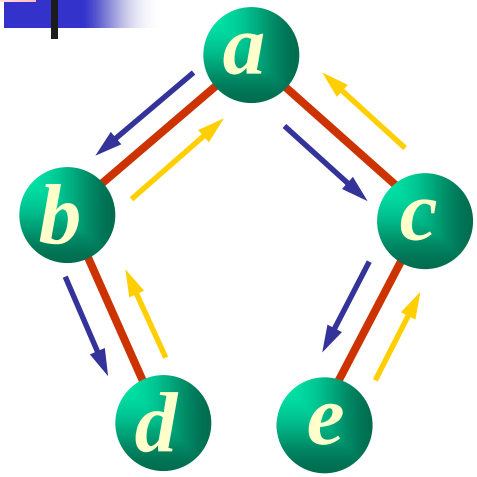
```
template <class T>
void BinaryTree<T>::
InOrder (void (*visit) (BinTreeNode<T> *t)) {
    stack<BinTreeNode<T>*> S;
```

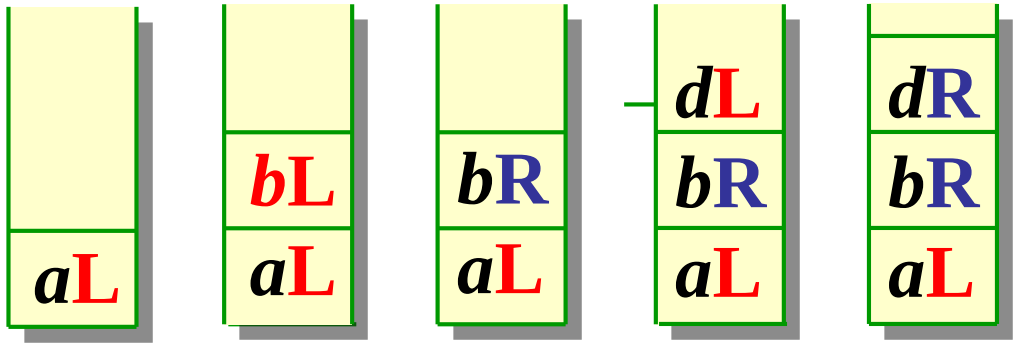
```
    BinTreeNode<T> *p = root;
    do {
        while (p != NULL) {     //遍历指针向左下移动
            S.Push (p);         //该子树沿途结点进栈
            p = p->leftChild;
        }
        if (!S.IsEmpty()) {                 //栈不空时退栈
            S.Pop (p);  visit (p); //退栈, 访问
            p = p->rightChild;   //遍历指针进到右子女
        }
    } while (p != NULL || !S.IsEmpty ());
};
```

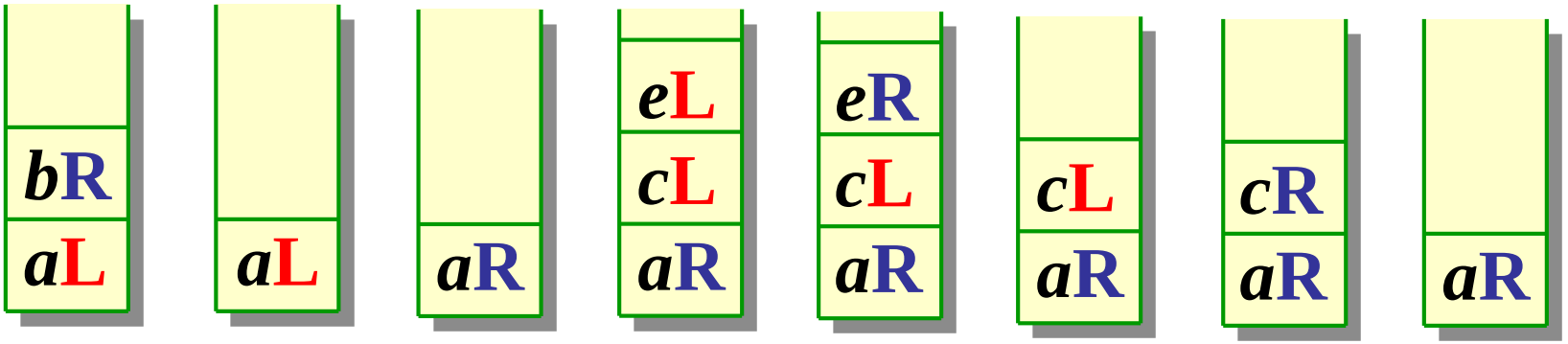
Preorder Traversal with a Stack



Postorder Traversal with a Stack

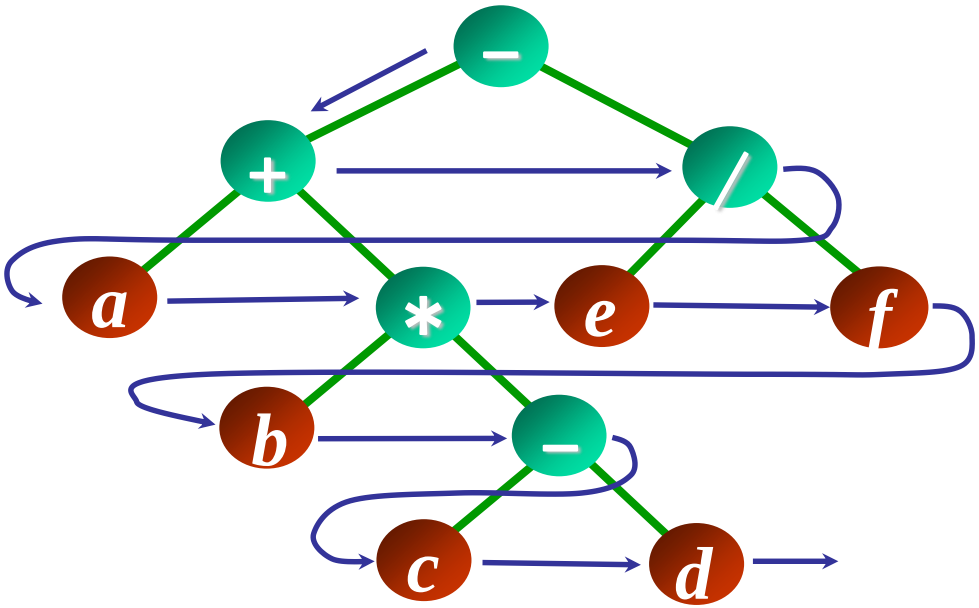




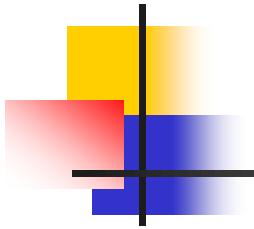


Level-Order Traversal

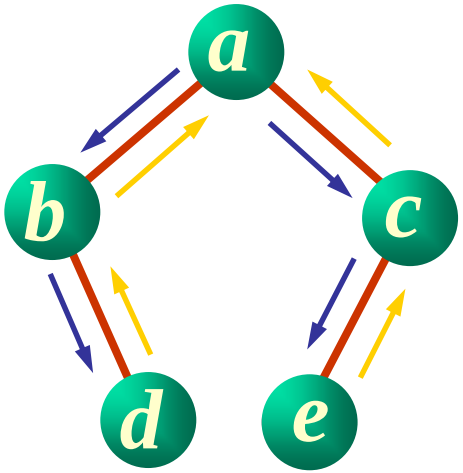
- 层次序遍历二叉树就是从根结点开始，按层次逐层遍历，如图：



遍历顺序



- 算法是非递归的
- 这种遍历需要使用一个先进先出的队列，
- 在处理上一层时，将其下一层的结点直接进到队列（的队尾）
- 在上一层结点遍历完后，下一层结点正好处于队列的队头，可以继续访问它们



Q | *a* | | | | |　访问*a*, 进队

Q | | *b* | *c* | | |　*a*出队
访问*b*, 进队
访问*c*, 进队

Q | | | *c* | *d* | |　*b*出队
访问*d*, 进队

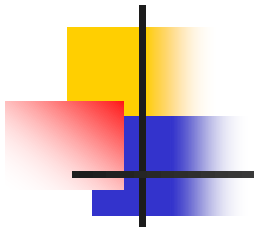
Q | | | | *d* | *e*　*c*出队
访问*e*, 进队

Q | | | | | *e*　*d*出队

Q | | | | | |　*e*出队

Level-Order Traversal

```
template <class T>
void BinaryTree<T>::
levelOrder (void (*visit) (BinTreeNode<T> *t)) {
    if (root == NULL) return;
    Queue<BinTreeNode<T> * > Q;
    BinTreeNode<T> *p = root;
    visit (p);   Q.EnQueue (p);
    while (!Q.IsEmpty ()) {
        Q.DeQueue (p);
        if (p->leftChild != NULL) {
```



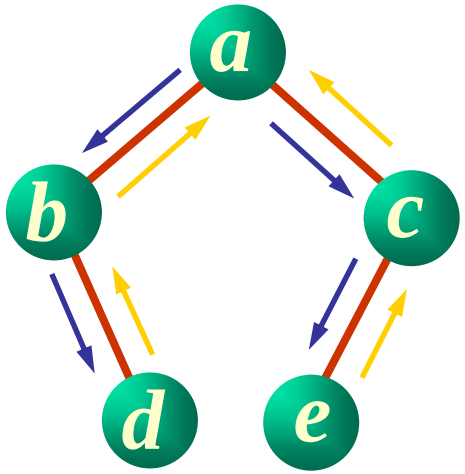
```
            visit (p->leftChild);
            Q.EnQueue (p->leftChild);
        }
        if (p->rightChild != NULL) {
            visit (p->rightChild);
            Q.EnQueue (p->rightChild);
        }
    }
};
```

二叉树的计数

- 二叉树遍历的结果是将一个非线性结构中的数据通过访问排列到一个线性序列中

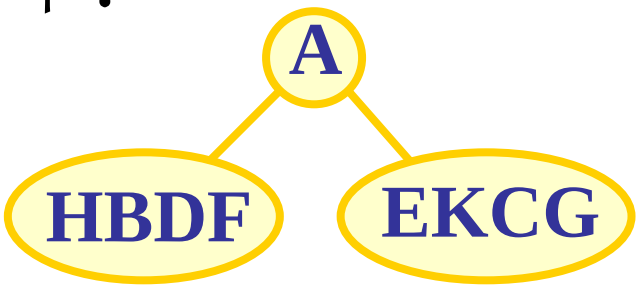
- 前序序列：*a b d c e* 特点是第一个访问的*a*一定是树根，只要左子树非空，后面紧跟的*b* 一定是根的左子女，…

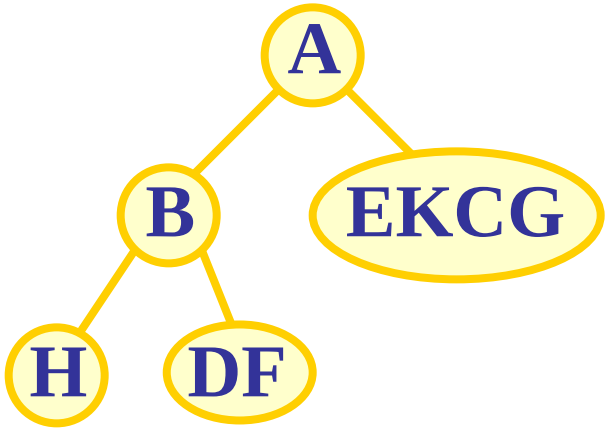
- 中序序列：*b d a e c* 特点是树根 *a* 把整个中序分成两部分，*a* 左侧子序列是根的左子树上的结点数据，右侧子序列是根的右子树上的结点数据

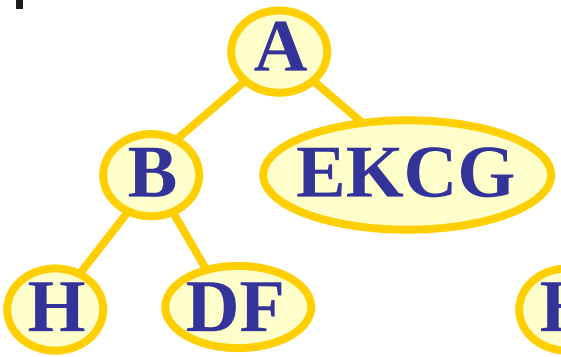


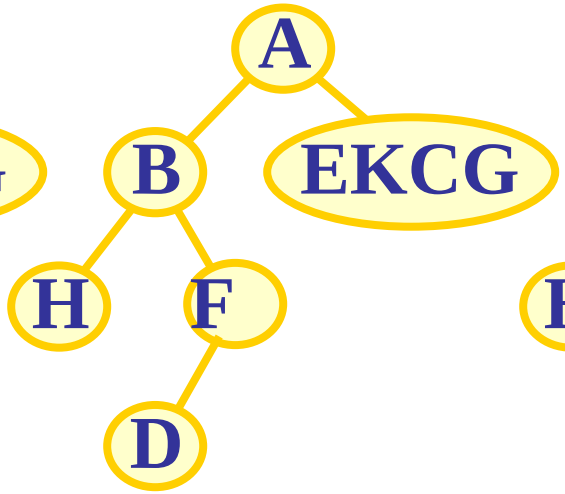
! 由二叉树的前序序列和中序序列可唯一地确定一棵二叉树。

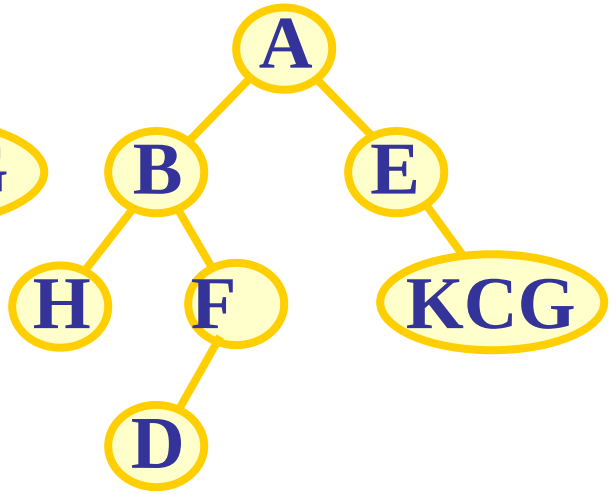
- 例, 前序序列 { A B H F D E C K G } 和中序序列 { H B D F A E K C G }, 构造二叉树过程如下：

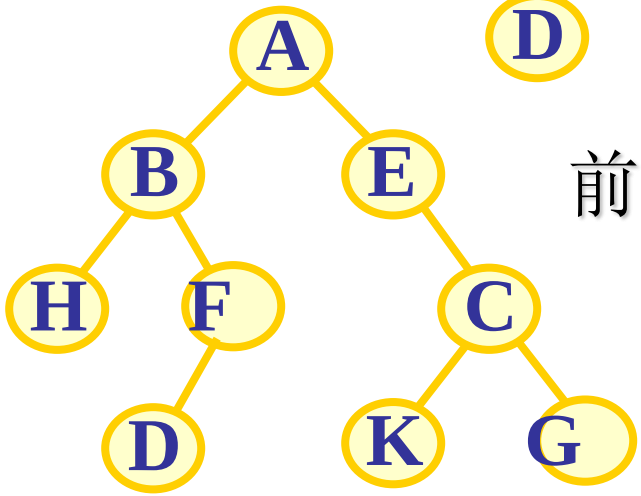






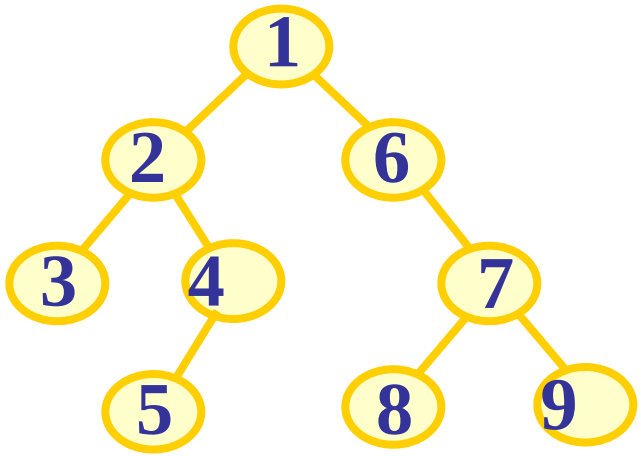


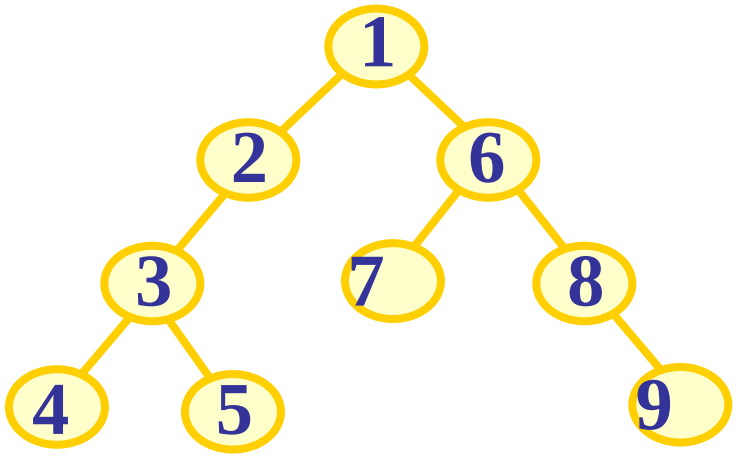


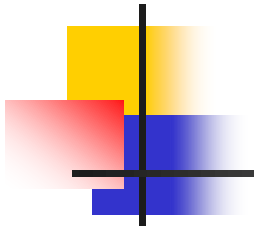


前序序列{ A B H F D E C K G }

- 如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。

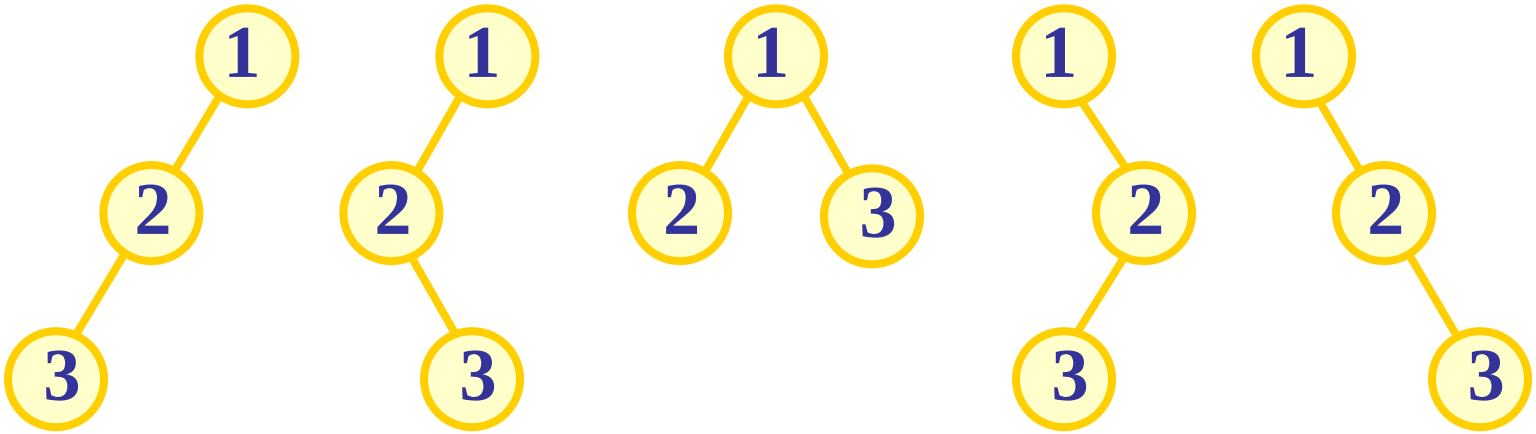






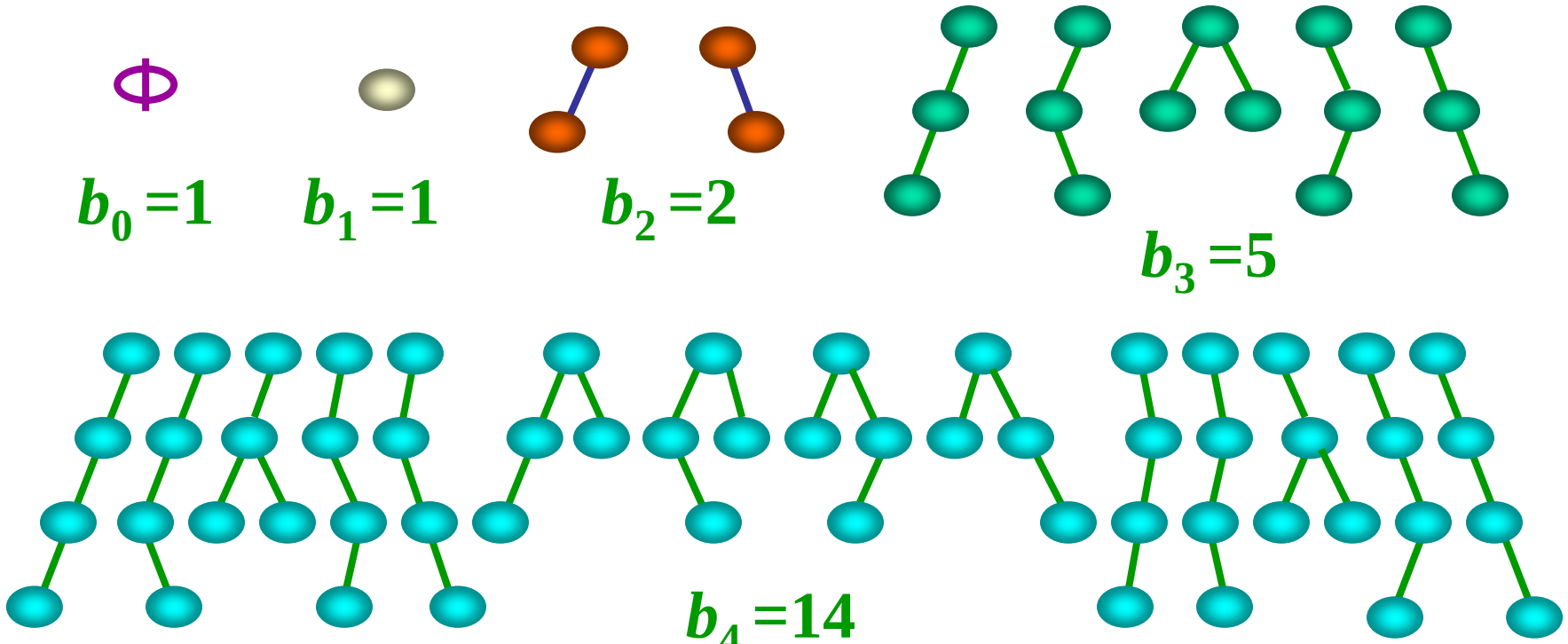
- 固定前序排列，选择所有可能的中序排列，可能构造多少种不同的二叉树？

- 例如，有 3 个数据 { 1, 2, 3 }，可得 5 种不同的二叉树。它们的前序排列均为 123，中序序列可能是 321，231，213，132，123。



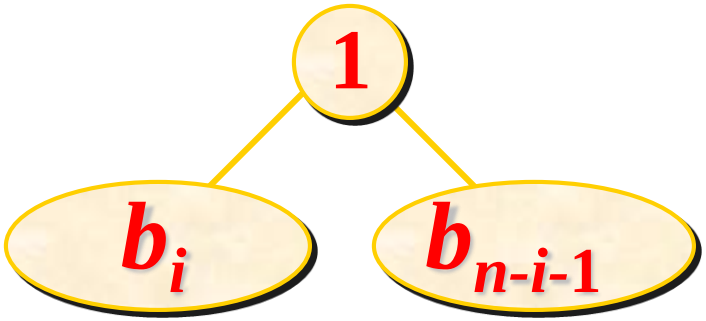
- 前序序列为 123，中序序列为 312 的二叉树不存在

有0个, 1个, 2个, 3个结点的不同二叉树如下



计算具有n个结点的不同二叉树的棵数

$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$



Catalan函数

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n!\cdot n!}$$

$$b_3 = \frac{1}{3+1} C_6^3 = \frac{1}{4} \frac{6\cdot 5\cdot 4}{3\cdot 2\cdot 1} = 5 \qquad b_4 = \frac{1}{4+1} C_8^4 = \frac{1}{5} \frac{8\cdot 7\cdot 6\cdot 5}{4\cdot 3\cdot 2\cdot 1} = 14$$

练习

- 给定二叉树如图所示。设N代表二叉树的根，L代表根节点的左子树，R代表根节点的右子树。若遍历后的结点序列是3175624，则其遍历方式是（　）。

A. LRN　B. NRL　C. RLN　D.RNL

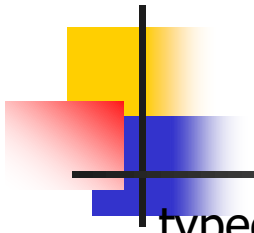
练习

- 一棵二叉树的前序遍历序列为1234567，它的中序遍历序列可能是（ ）。

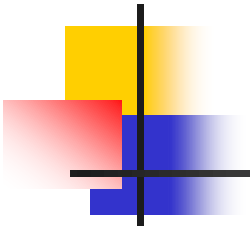
A. 3124567　　B. 1234567

C. 4135627　　D. 1463572

```
Typedef  int  ElemType;          //链表数据的类型定义
Typedef  struct  Lnode{          //链表结点的结构定义
           ElemType  data;       //结点数据
           struct Lnode  *link;  //结点链接指针
}Lnode, *LinkList;
Int Search_k(LinkList  list, int   k) {
    Lnode  *p=list->link, *q=list->link;
    int count=0;
    while (p!=NULL){
           if (count<k)  count++;
           else q=q->link;
           p=p->link;
    }
    if (count<k)
        return 0;
    else { printf("%d", q->data); return 1;}
}
```

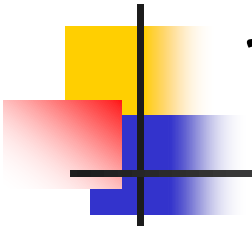


```
typedef  struct Node{
        char data;
        struct Node *next;
}Snode;
int listlen(Snode  *head) {
     int len=0;
     while (head->next!=NULL) {
             len++; head=head->next;}
Snode*  find_addr(Snode *strl, Snode  *str2){
     int m, n;
     Snode  *p, *q;
     m=listlen(str1);   n=listlen(str2);
     for (p=str1; m>n; m--)    p=p->next;
     for (q=str2; m<n; n--)     q=q->next;
     while (p->next !=NULL && p->next !=q->next) {
             p=p->next;  q=q->next;}
     return p->next;
}
```



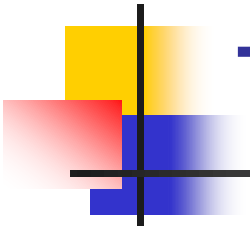
Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal



Threaded Binary Tree

- Note that in the linked representation of binary tree, there are n +1 0 links, which is more than actual pointers.
- A clever way to make use of these 0 links is to replace them by pointers, called threads, to other nodes in the tree.



Threads

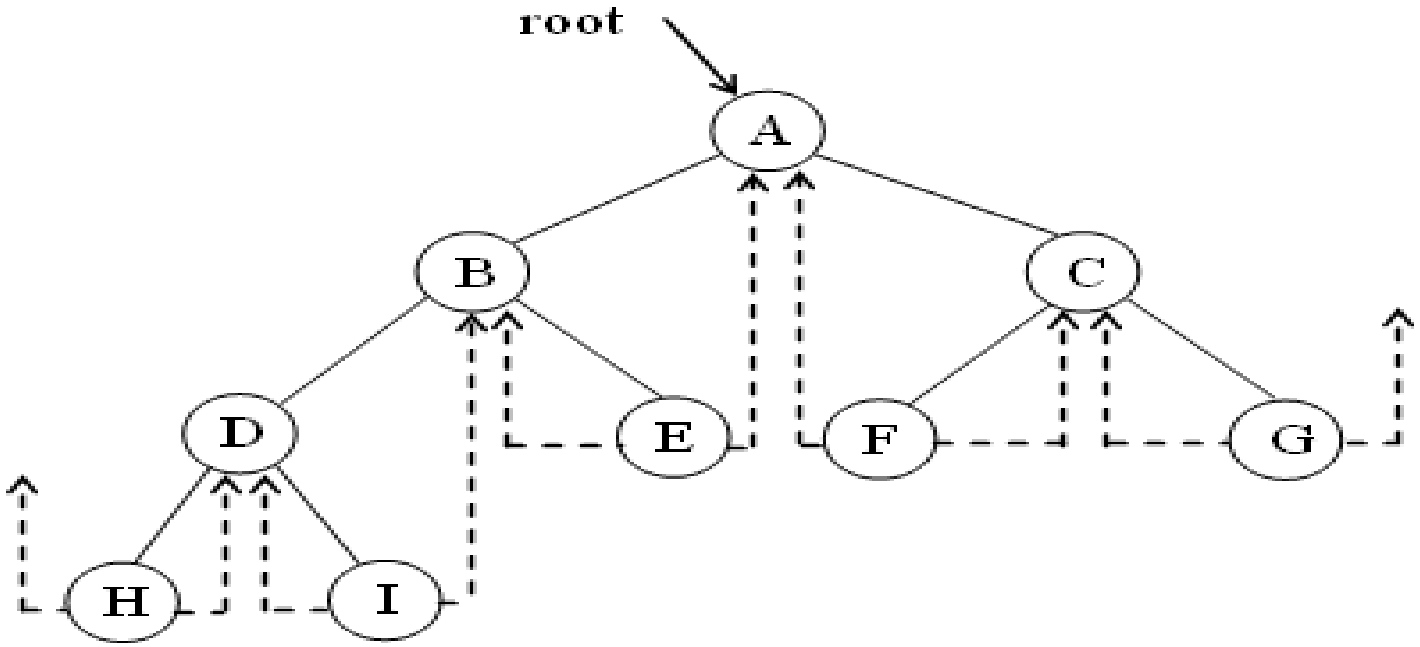
- The threads are constructed using the following rules:

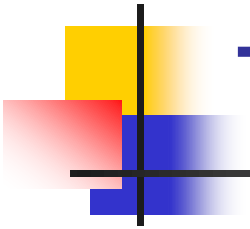
(1) A 0 rightChild field at node p is replaced by a pointer to the inorder successor of p.

(2) A 0 leftChild field at node p is replaced by a pointer to the inorder predecessor of p.

An Example

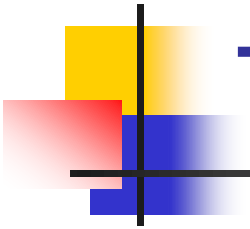
- The following is a threaded tree, in which node E has a predecessor thread pointing to B and a successor thread to A.





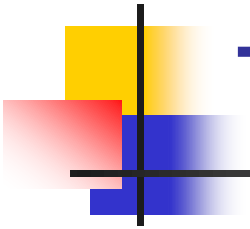
Threads

- To distinguish between threads and normal pointers, add two bool fields:
  - leftThread
  - rifgtThread
- If t→leftThread == true, then t→leftChild contains a thread, otherwise a pointer to left child. Similar for t→rightThread.



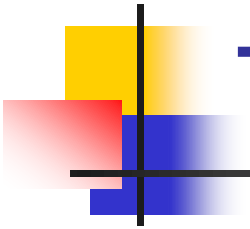
Threads

```
template <class T>
class ThreadedNode {
friend class ThreadedTree;
private:
    bool  leftThread;
    ThreadedNode * leftChild;
    T data;
    ThreadedNode * rightChild;
    bool  rightThread;
};
```



Threads

```
template <class T>
class ThreadedTree {
public:
    // Tree operations
…
private:
    ThreadedNode *root;
};
```



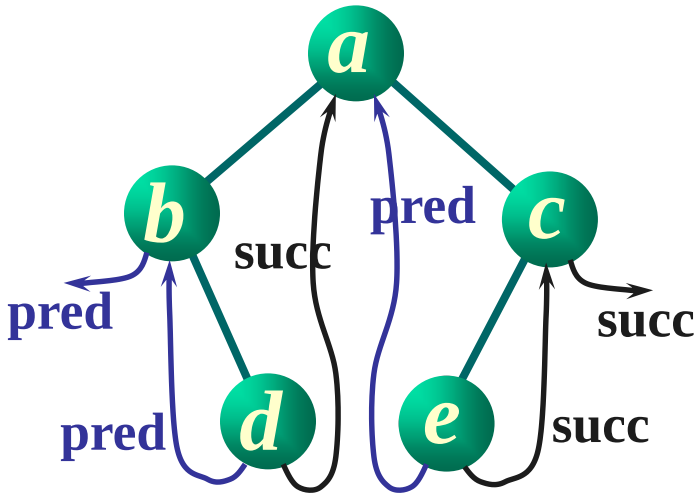
Threads

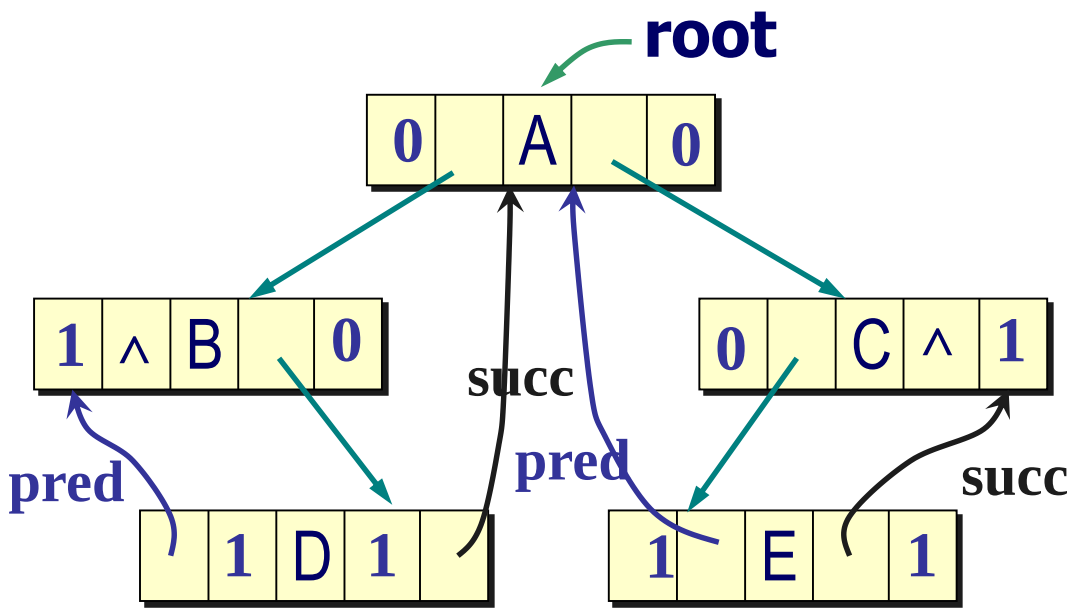
Let ThreadedInorderIterator be a nested class of ThreadedTree:

```
class ThreadedInorderIterator {
public:
    T* Next();
    ThreadedInorderIterator()
        { currentNode = root; }
private:
    ThreadedNode<T>* currentNode;
};
```

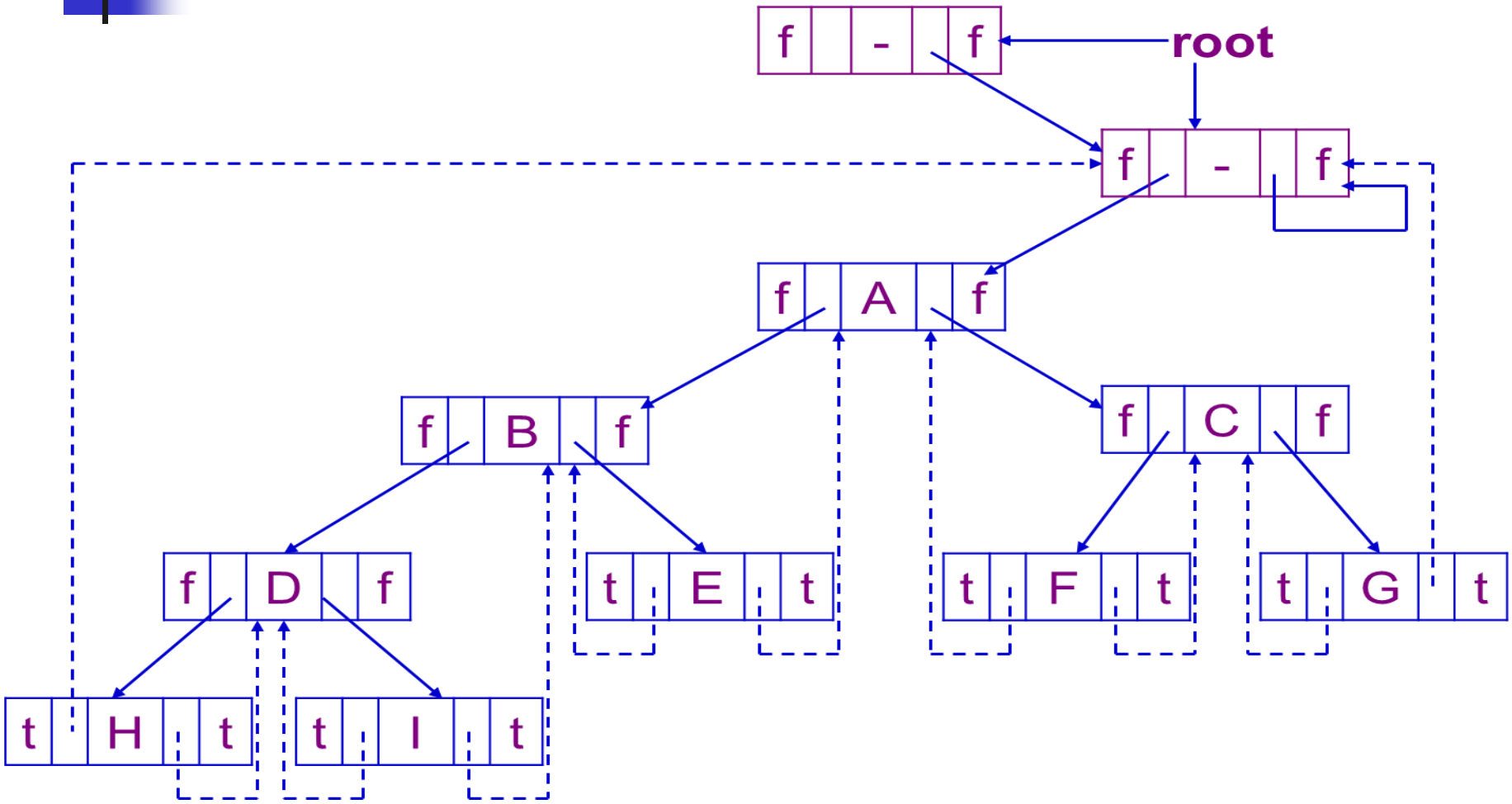
Linked Representation

Ltag	leftChild	data	rightChild	Rtag

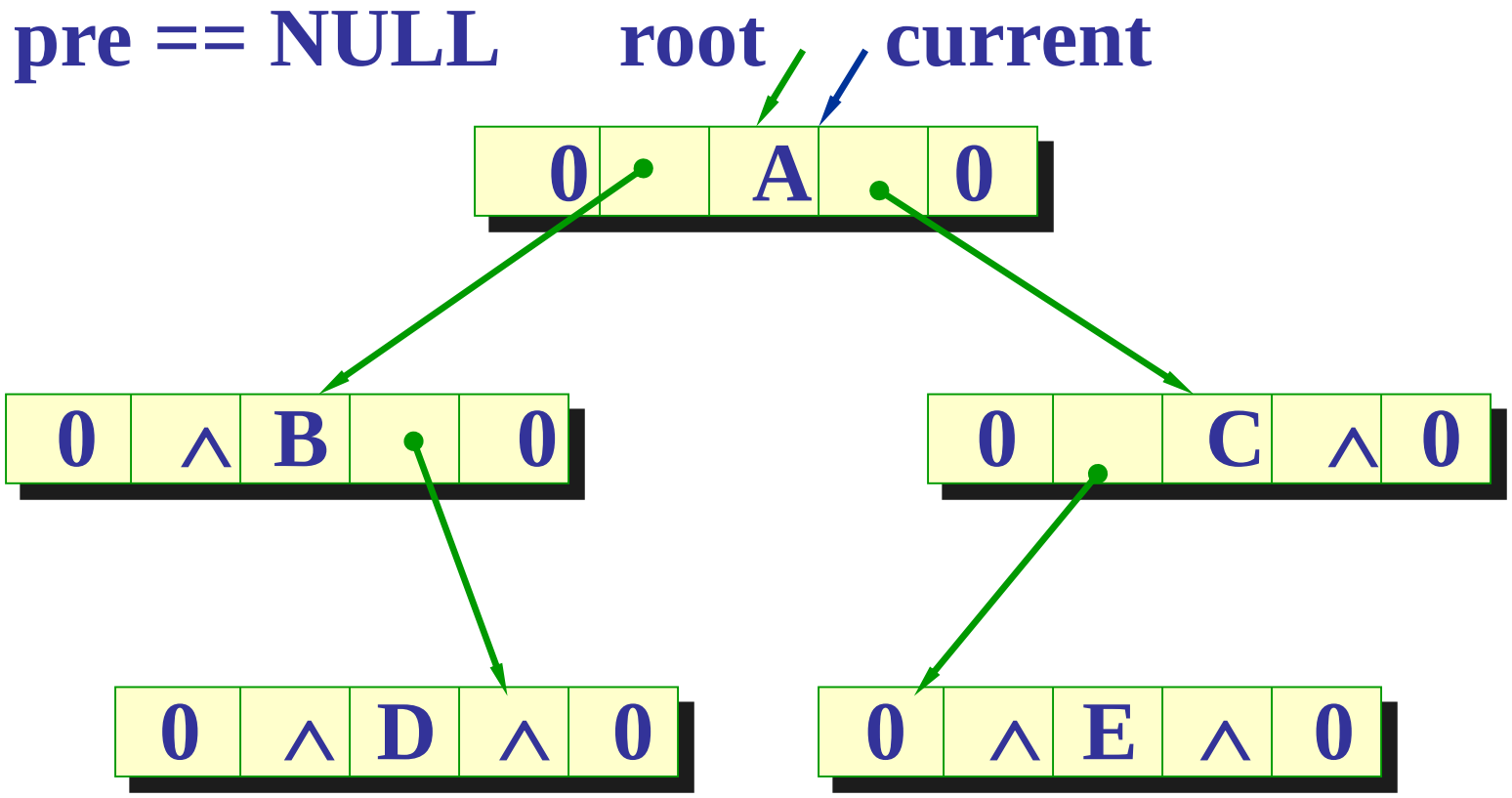


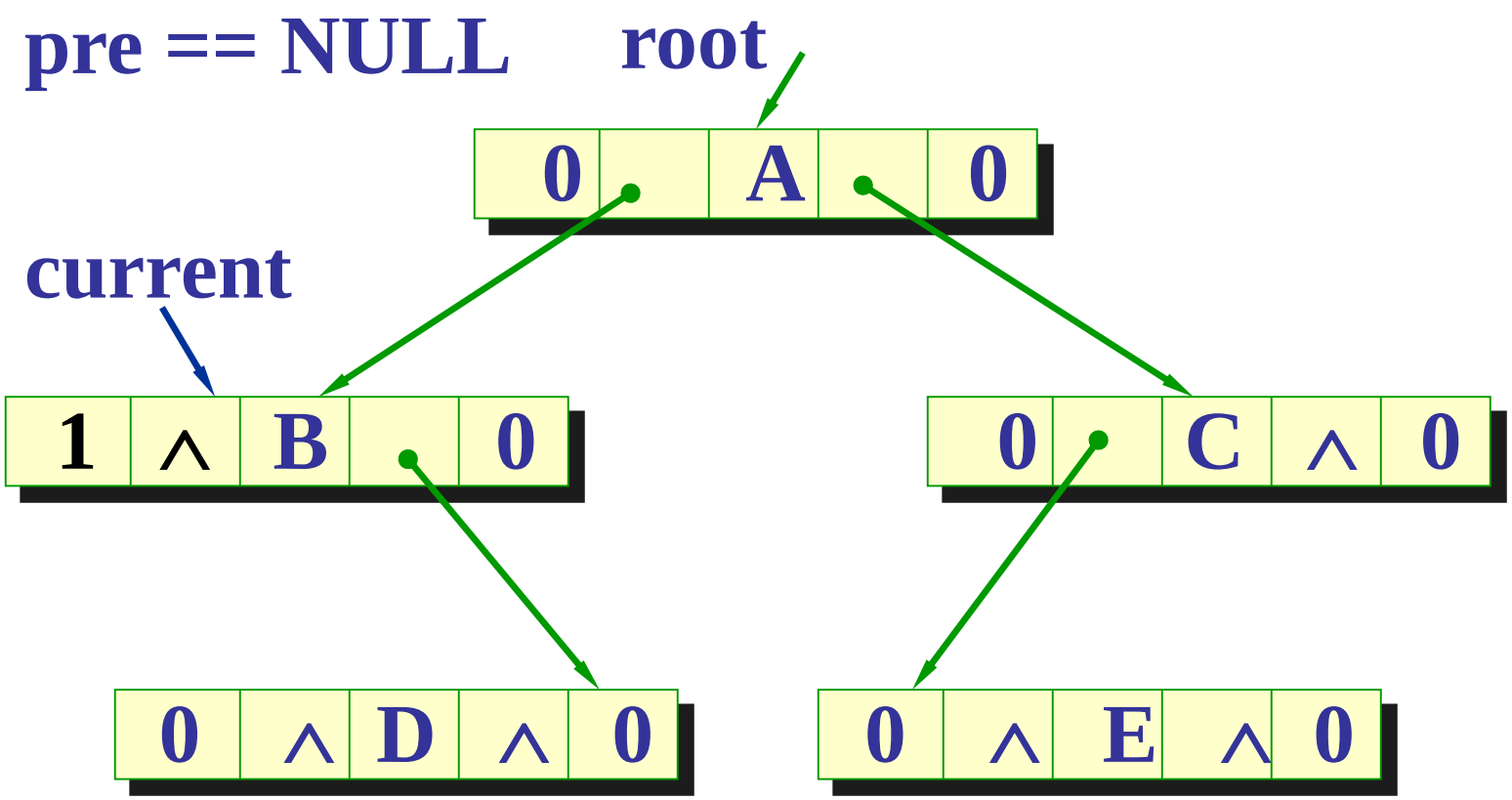


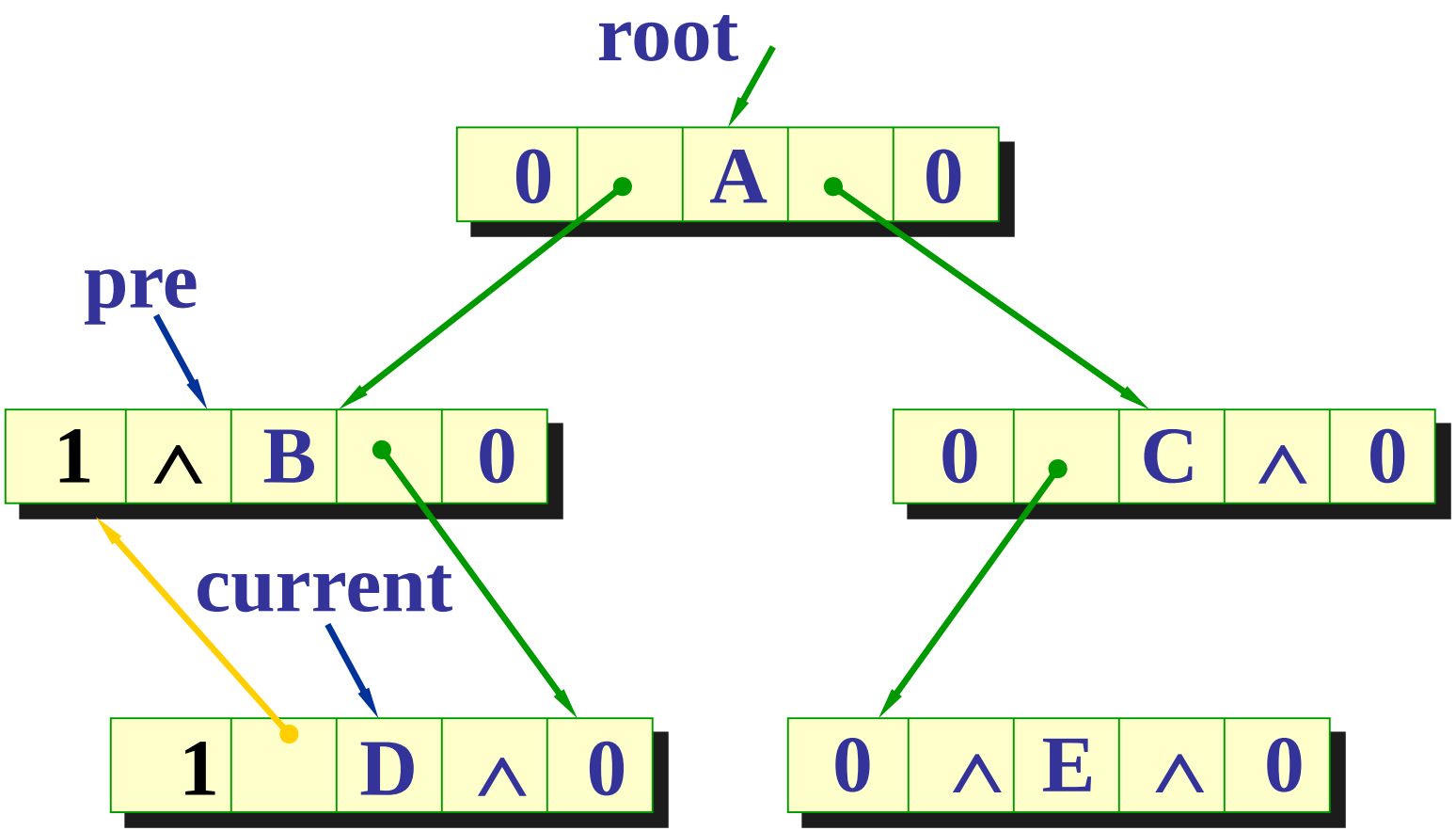
Memory representation of threaded tree

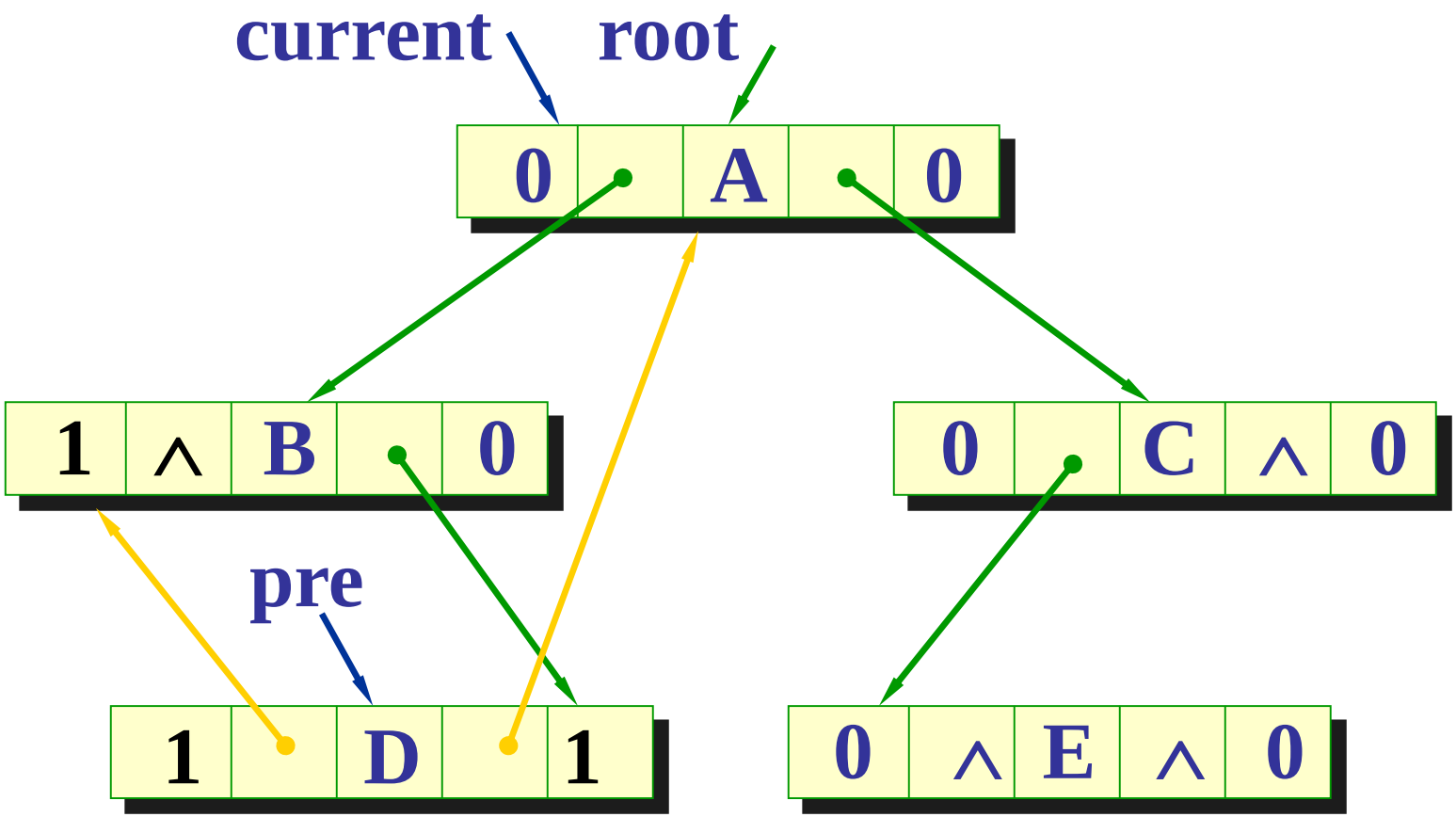


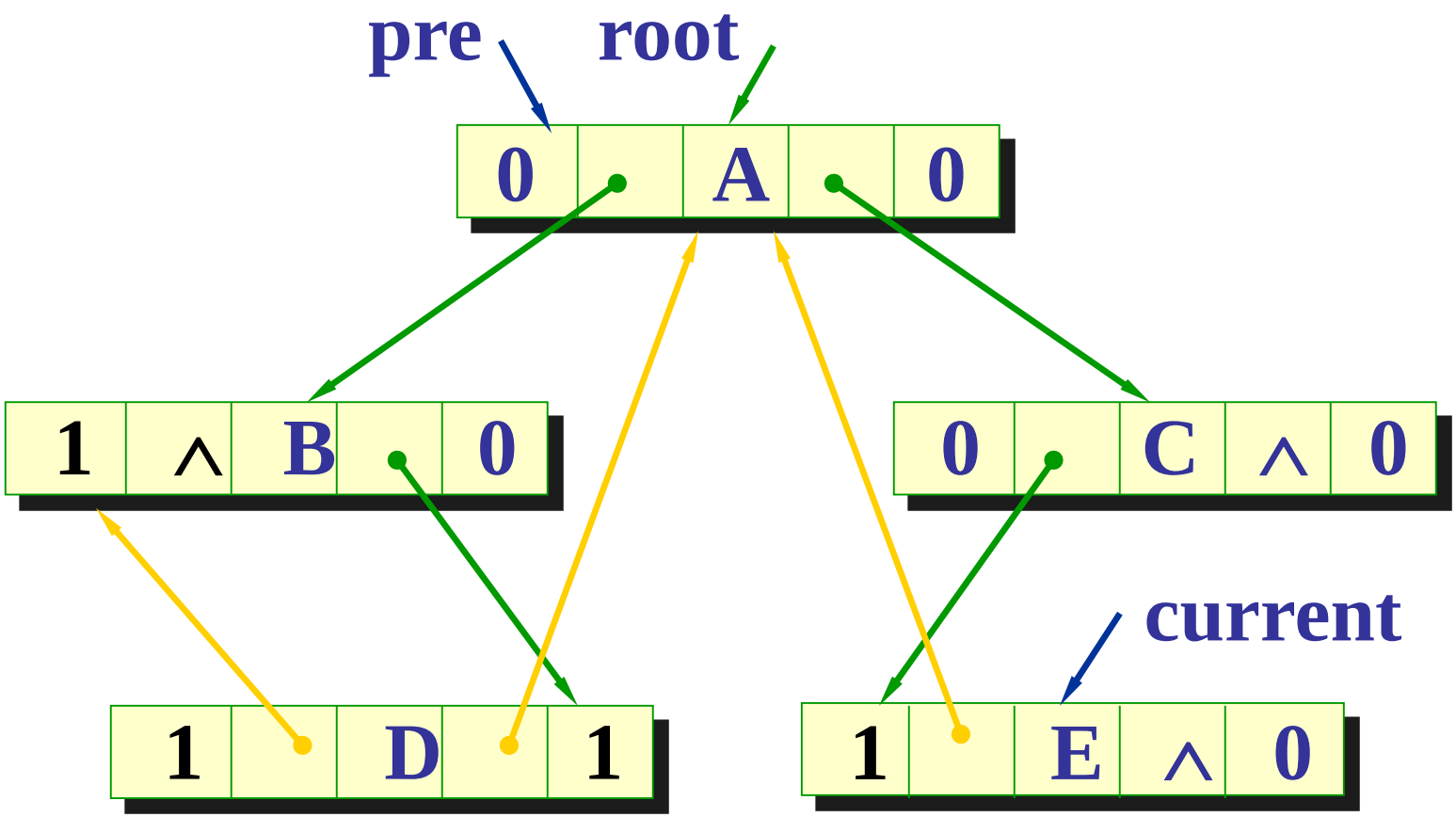
通过中序遍历建立中序线索化二叉树

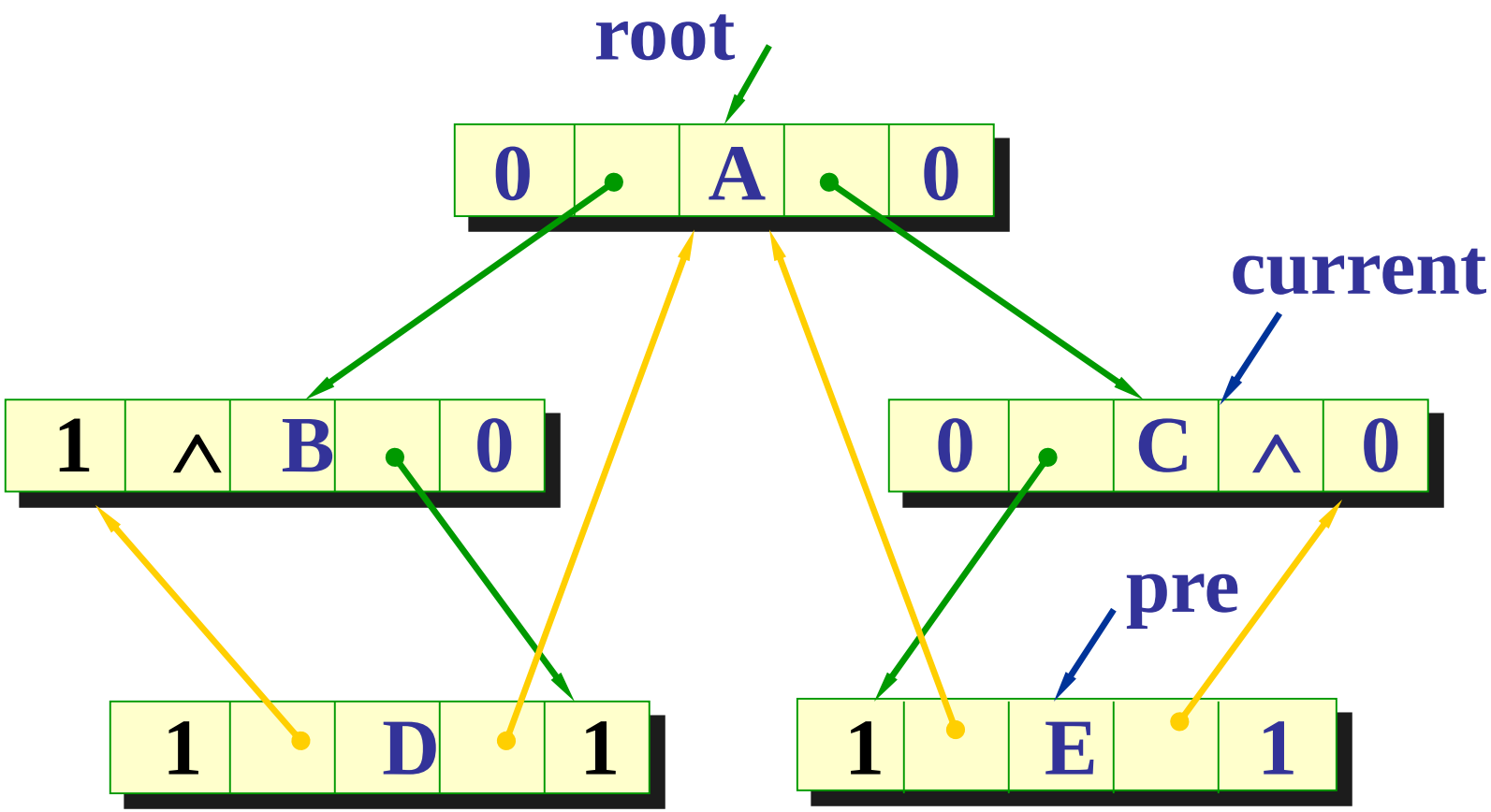


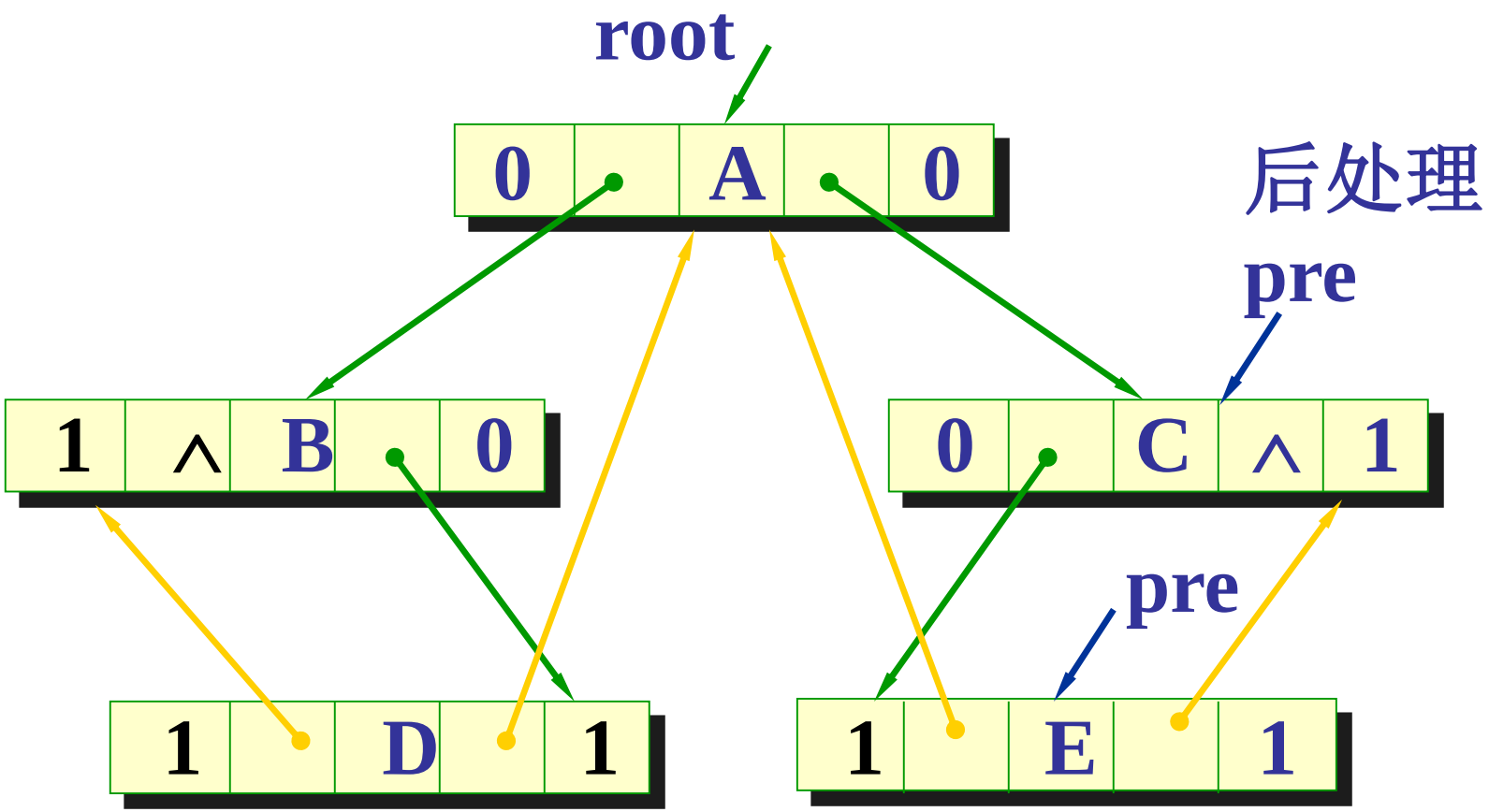

pre == NULL
root
0
A
0
current
1
∧
B
0
0
C
∧
0
0
∧
D
∧
0
0
∧
E
∧
0


root
0
A
0
pre
1
B
0
0
C
0
current
1
D
0
0
E
0


current
root
0
A
0
1
B
0
0
C
0
pre
1
D
1
0
E
0


pre
root
0
A
0
1
∧
B
0
0
C
∧
0
current
1
D
1
1
E
∧
0


root
0
A
0
current
1
B
0
0
C
0
pre
1
D
1
1
E
1

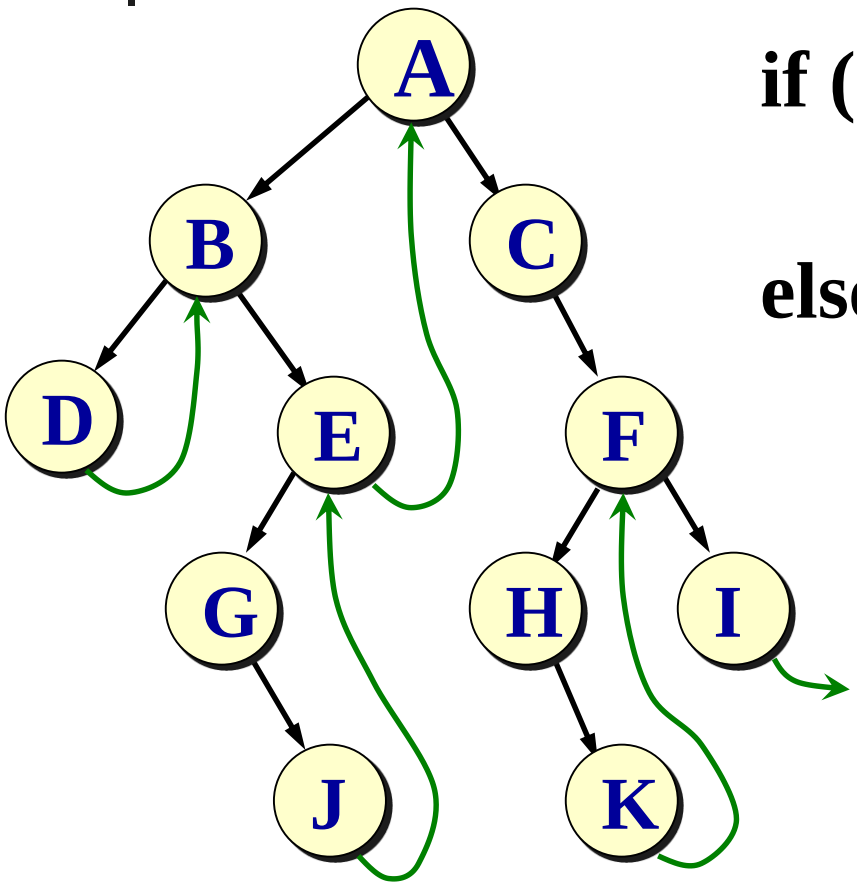

root
0
A
0
后处理
pre
1
∧
B
0
0
C
∧
1
pre
1
D
1
1
E
1

```
template <class T>
void ThreadTree<T>::createInThread () {
    ThreadNode<T> *pre = NULL;    //前驱结点指针
    if (root != NULL) {           //非空二叉树,线索化
        createInThread (root, pre);
                    //中序遍历线索化二叉树
        pre->rightChild = NULL;  pre->rtag = 1;
  //后处理中序最后一个结点
    }
}
```

```
template <class T>
void ThreadTree<T>::
createInThread (ThreadNode<T> *current,
    ThreadNode<T> *& pre) {
//通过中序遍历,对二叉树进行线索化
    if (current == NULL) return;
    createInThread (current->leftChild, pre);
        //递归,左子树线索化
    if (current->leftChild == NULL) {
        //建立当前结点的前驱线索
        current->leftChild = pre;   current->ltag = 1;
    }
```

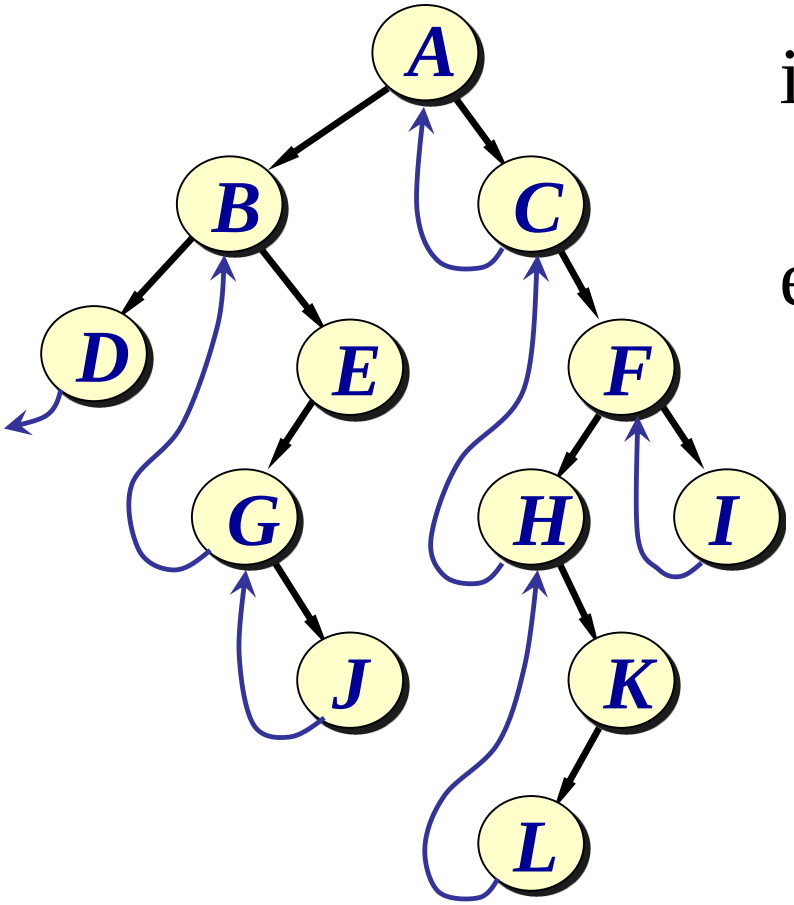
```
    if (pre != NULL && pre->rightChild == NULL)
            //建立前驱结点的后继线索
       { pre->rightChild = current;   pre->rtag = 1; }
    pre = current;
            //前驱跟上,当前指针向前遍历
    createInThread (current->rightChild,  pre);
            //递归,右子树线索化
}
```

Finding Successor




```
if (current->rtag ==1)
    后继为current->rightChild
else   //current->rtag == 0
    后继为当前结点右子树
    的中序下的第一个结点
```

Finding Predecessor



if (current->ltag == 1)

前驱为current->leftChild

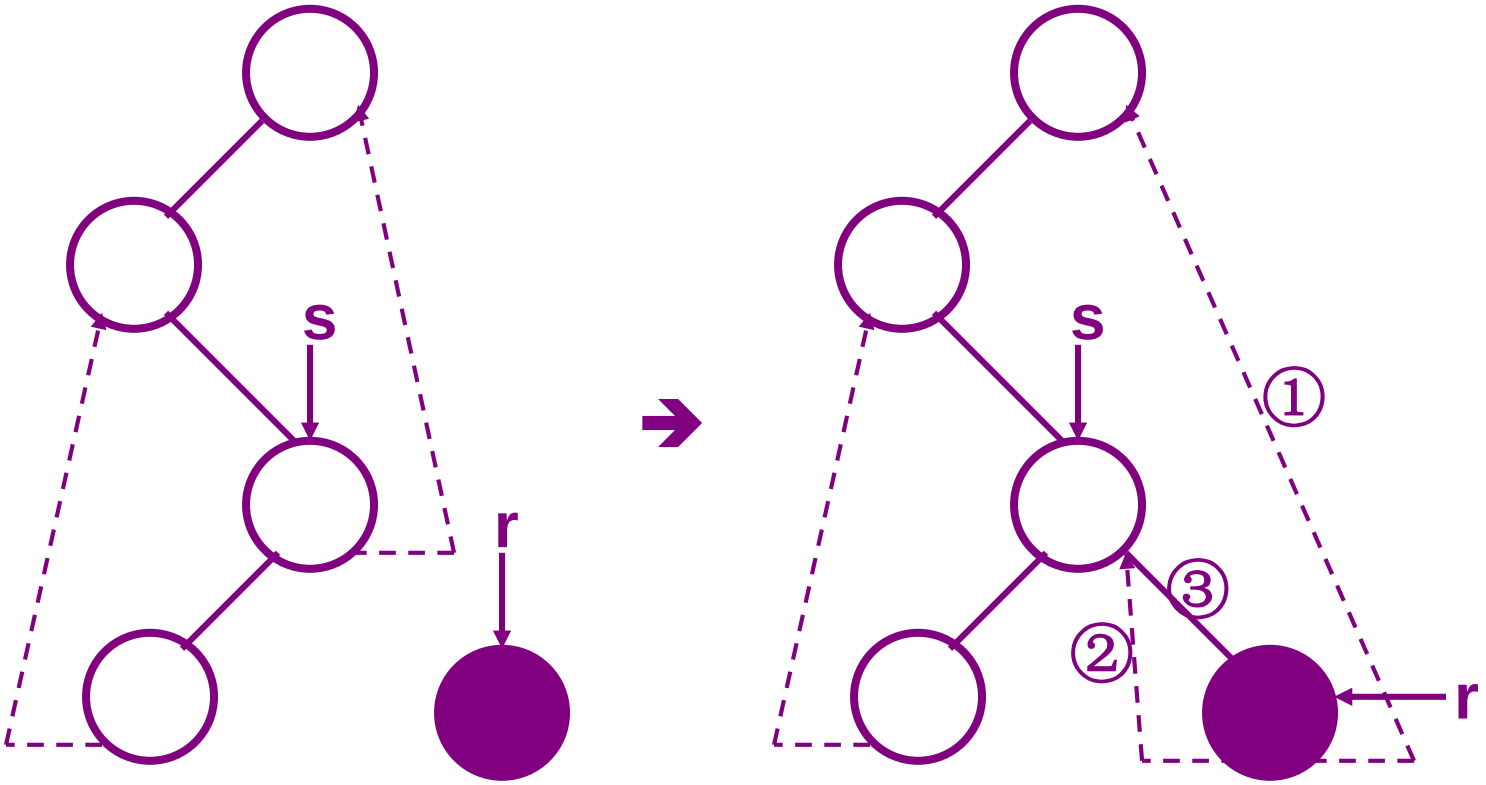
else //current->ltag == 0

前驱为当前结点左子树中序下的最后一个结点

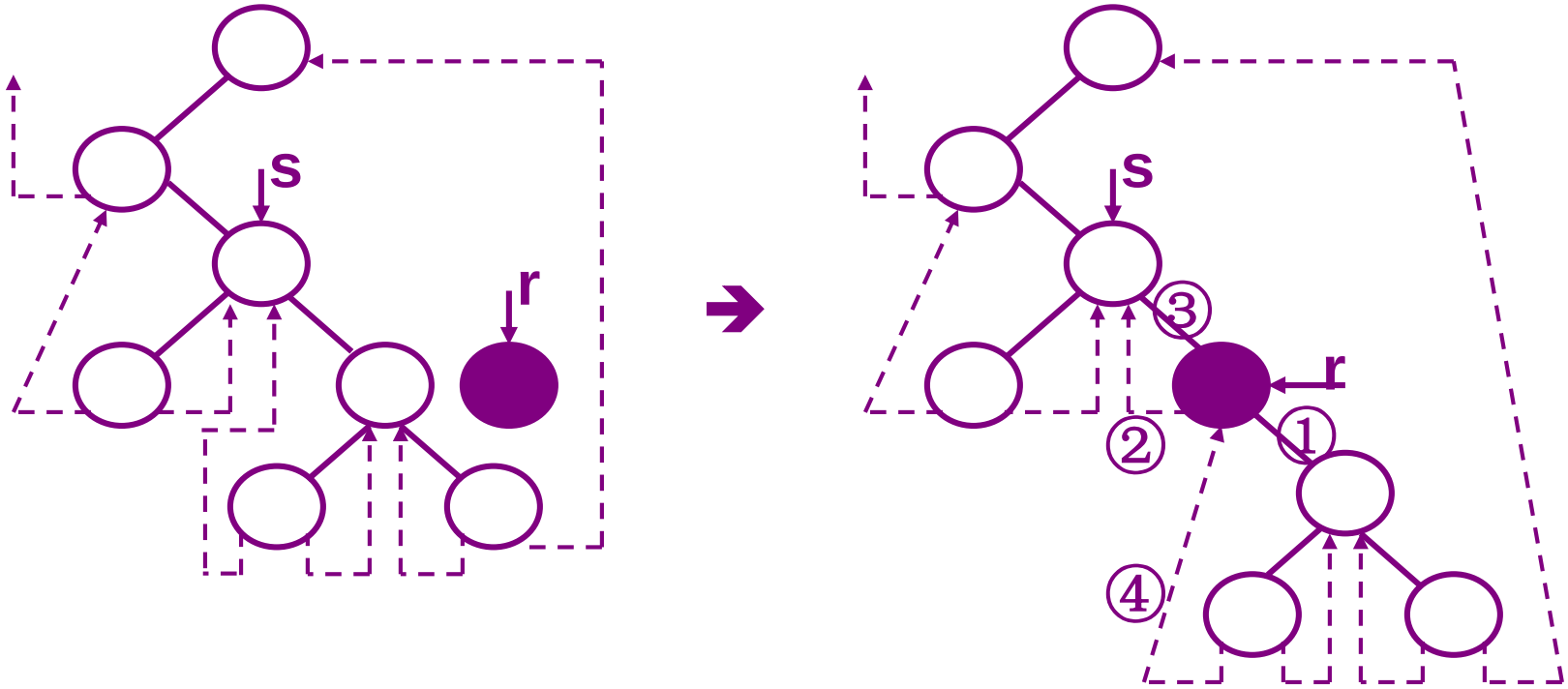
Inserting a Node into a Threaded Binary Tree

- Insertion into a threaded tree provides the function for growing a threaded tree.
- We shall study only the case of inserting r as the right child of s. The left child case is similar.

(1) If s→rightThread==true, as:

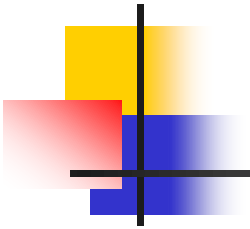


(2) If s→rightThread==false, as:



In both (1) and (2), actions ①,②, ③ are the same, ④ is special for (2).

```
template <class T>
void  ThreadedTree<T>::InsertRight(ThreadedNode<T>* s,
                                   ThreadedNode<T>* r)
{ // insert r as the right child of s
    r→rightChild=s→rightChild;                    // ①
    r→rightThread=s→rightThread;   // ① note s!=t.root,
    r→leftChild=s;                                // ②
    r→leftThread=true;                            // ②
    s→rightChild=r;                               // ③
    s→rightThread=false;                          // ③
    if (! r→rightThread) {     // case (2)
       ThreadedNode<T>* temp=InorderSucc(r);  // ④
       temp→leftChild=r;                                // ④
    }
}
```



Trees

- Trees
- Binary Trees
  - Definition and Features
  - Binary Tree Representations
  - Traversal
  - Threaded Binary Trees
- Heaps and Binary Search Trees
  - Heaps
  - Huffman Trees
- Forrests
  - Tree Representations
  - Transforming between Trees and Forrests
  - Traversal

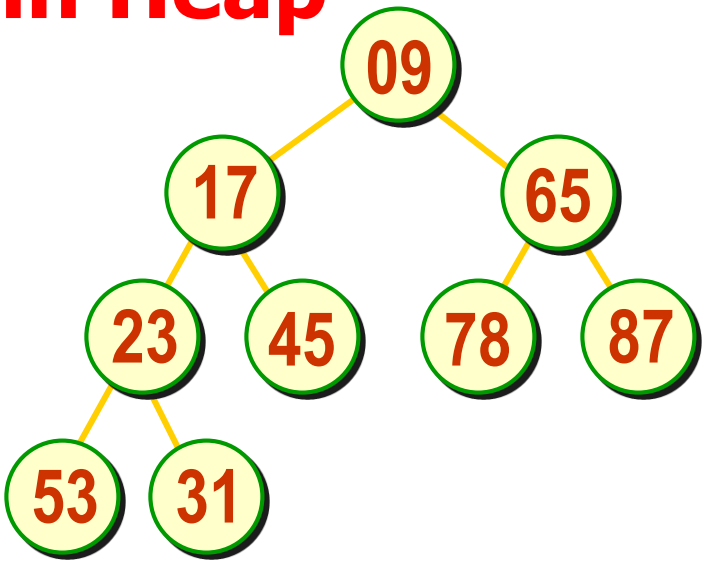
Priority Queue

- In a priority queue, the element to be deleted is the one with highest (or lowest) priority.
- 用堆实现其存储表示，能够高效运作

```
template <class T, class E>
class MinPQ {          //最小优先级队列类的定义
public:
    Virtual bool Insert (E& d) = 0;
    Virtual bool Remove (E& d) = 0;
};
```

Definition of Heap

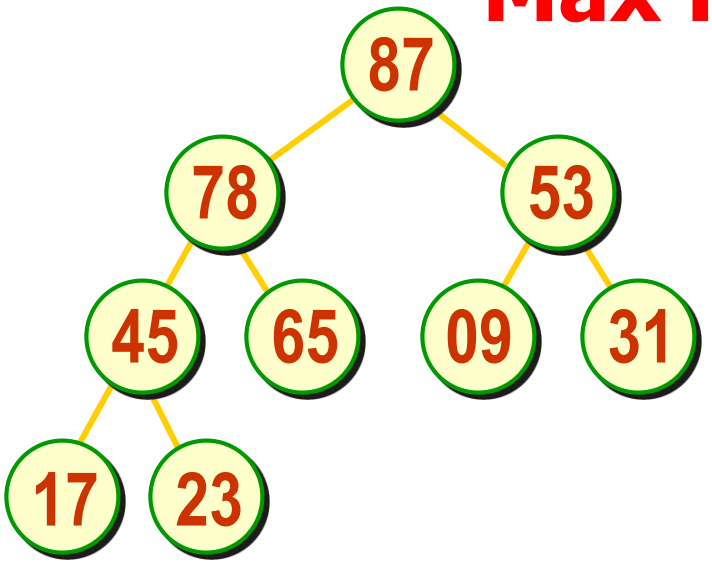
Min Heap



完全二叉树顺序表示

$K_i \leqslant K_{2i+1}$ &&$K_i \leq K_{2i+2}$

Max Heap



完全二叉树顺序表示

$K_i \geqslant K_{2i+1}$ &&$K_i \geq K_{2i+2}$

Computing Index in a Heap

- 堆存储在下标从 0 开始计数的一维数组中，在堆中给定下标为 i 的结点时
  a) 如果 $i = 0$，结点 i 是根结点，无双亲；否则结点 i 的父结点为结点 $\lfloor (i-1)/2 \rfloor$；
  b) 如果 $2i+1 > n-1$，则结点 i 无左子女；否则结点 i 的左子女为结点 $2i+1$；
  c) 如果 $2i+2 > n-1$，则结点 i 无右子女；否则结点 i 的右子女为结点 $2i+2$。

Definition of Min Heap

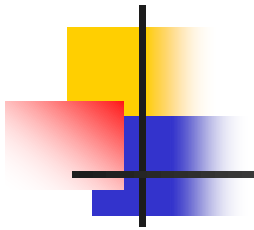
```
template <class T, class E>
class MinHeap : public MinPQ<T, E> {
//最小堆继承了（最小）优先级队列
public:
    MinHeap (int sz = DefaultSize);     //构造函数
    MinHeap (E arr[], int n);           //构造函数
    ～MinHeap() { delete [ ] heap; }    //析构函数
    bool Insert (E& d);                 //插入
    bool Remove (E& d);                 //删除
```

```
    bool IsEmpty () const                    //判堆空否
      { return  currentSize == 0; }
    bool IsFull () const                     //判堆满否
      { return currentSize == maxHeapSize; }
     void MakeEmpty () { currentSize = 0; }  //置空堆
private:
    E *heap;                                 //最小堆元素存储数组
    int currentSize;       //最小堆当前元素个数
    int maxHeapSize;   //最小堆最大容量
    void siftDown (int start, int m);     //调整算法
    void siftUp (int start);                 //调整算法
};
```

Create a Heap

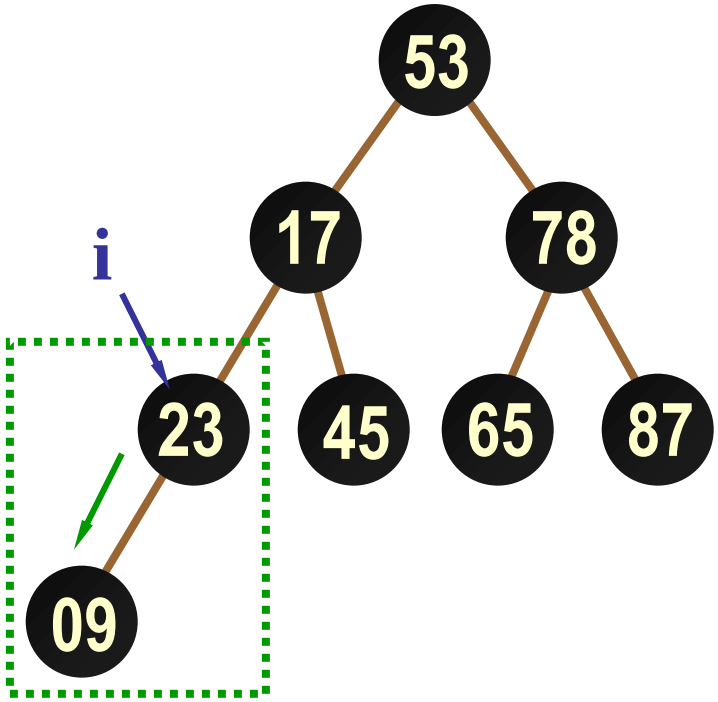
```
template <class T, class E>
MinHeap<T>::MinHeap (int sz)
{
    maxHeapSize = (DefaultSize < sz) ? sz : DefaultSize;
    heap = new E[maxHeapSize];      //创建堆空间
    if (heap == NULL) {
        cerr << "堆存储分配失败！" << endl;  exit(1);
    }
    currentSize = 0;     //建立当前大小
}
```

```
template <class T, class E>
MinHeap<T>::MinHeap (E arr[], int n)
{
    maxHeapSize = (DefaultSize < n) ? n : DefaultSize;
    heap = new E[maxHeapSize];
     if (heap == NULL) {
       cerr << "堆存储分配失败！" << endl;  exit(1);
    }
    for (int i = 0; i < n; i++) heap[i] = arr[i];
    currentSize = n;  //复制堆数组, 建立当前大小
     int currentPos = (currentSize−2)/2;
                        //找最初调整位置:最后分支结点
```

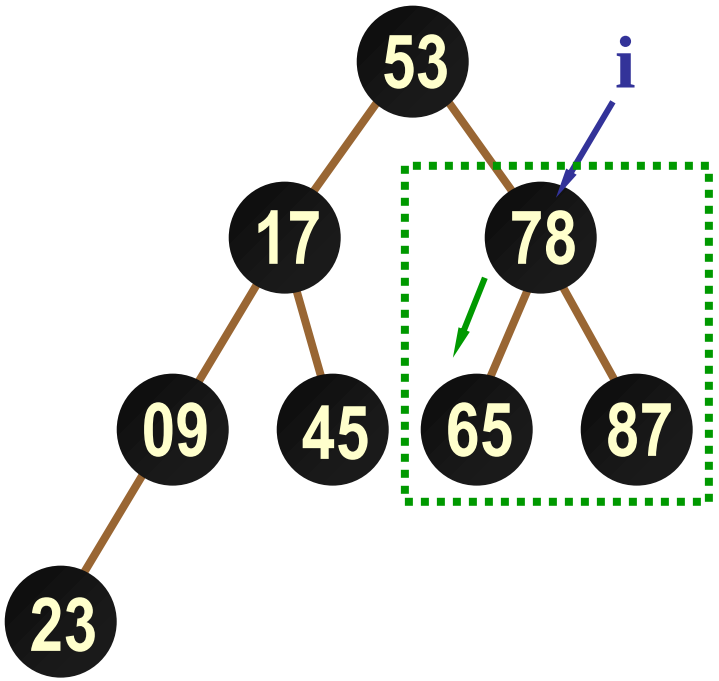


```
        while (currentPos >= 0) {     //逐步向上扩大堆
            siftDown (currentPos, currentSize-1);
             //局部自上向下下滑调整
            currentPos--;
        }
    }
```

SiftDown

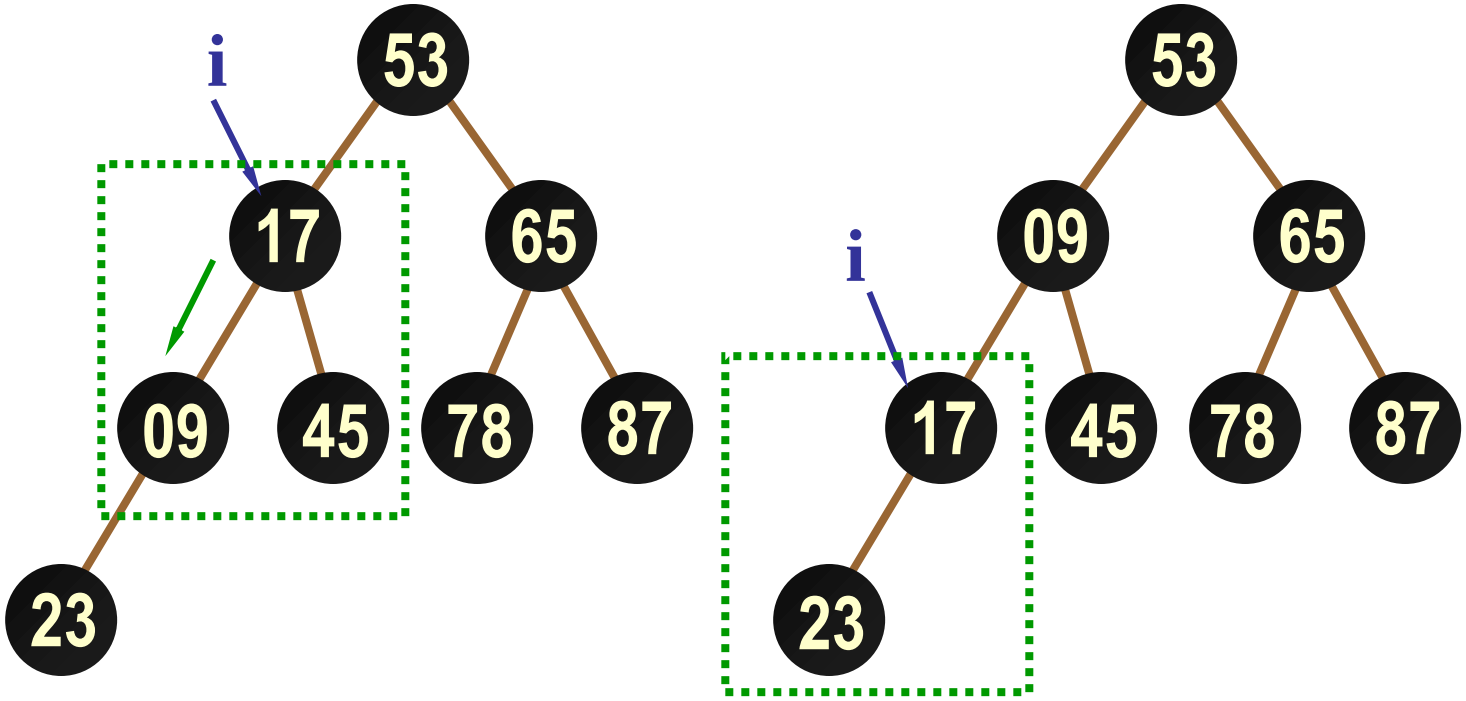


currentPos = i = 3

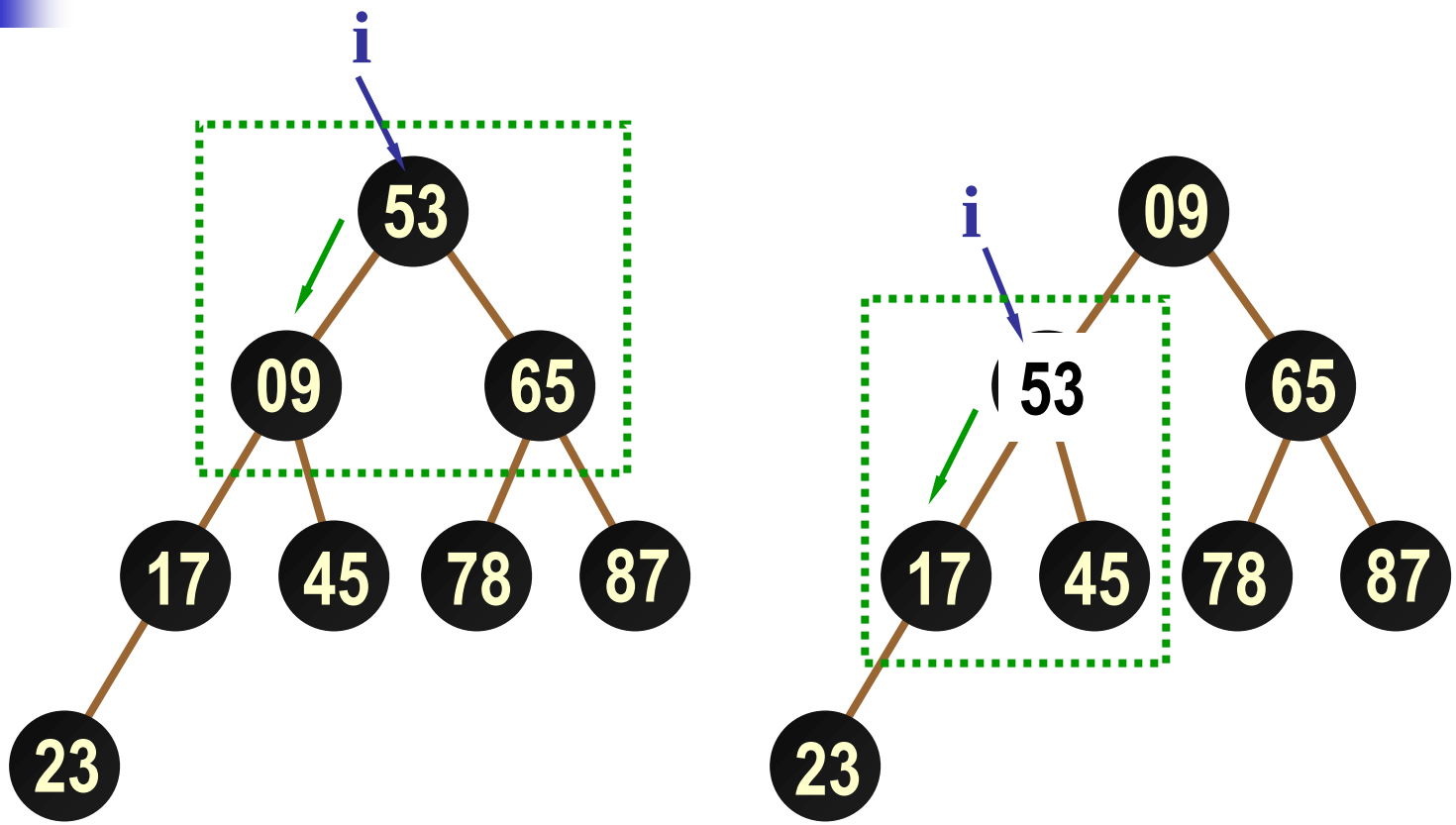


currentPos = i = 2

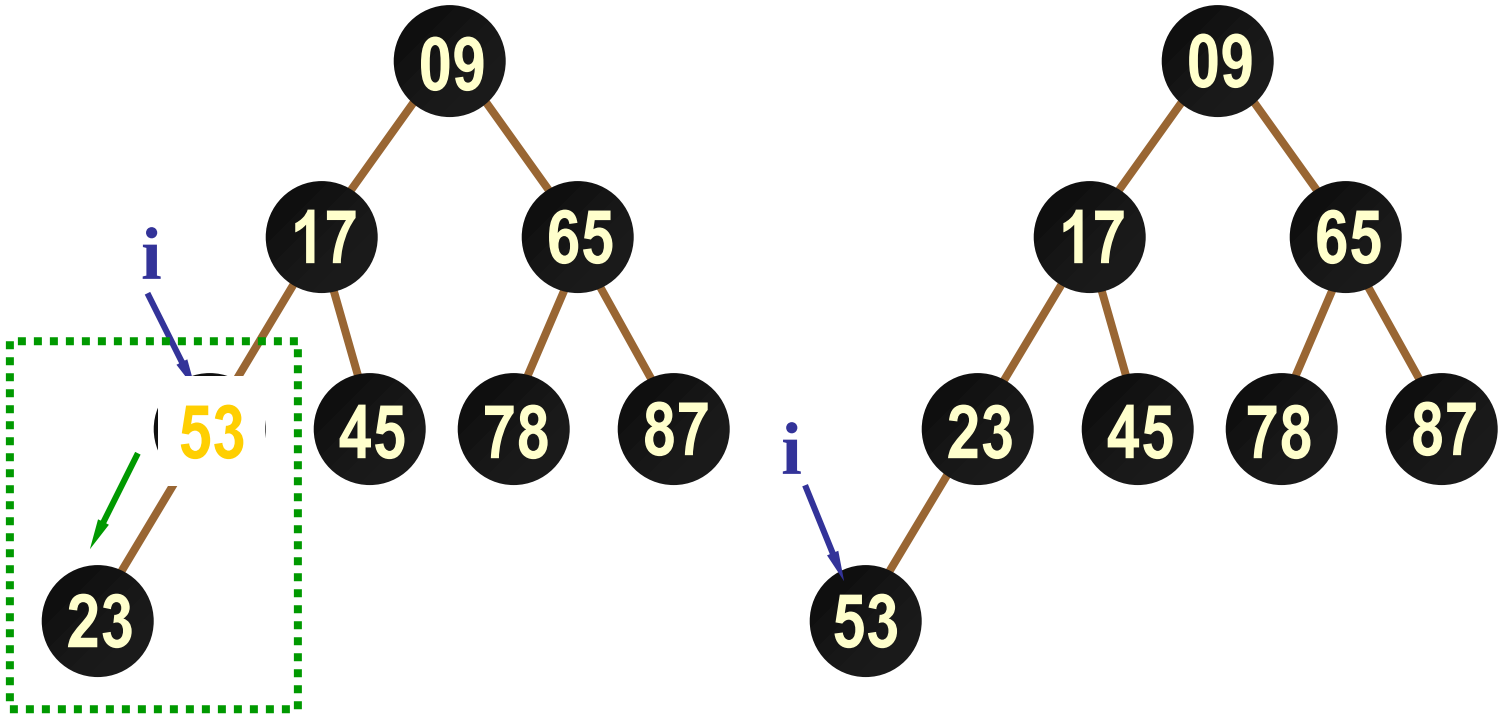
自下向上逐步调整为最小堆



currentPos = i = 1

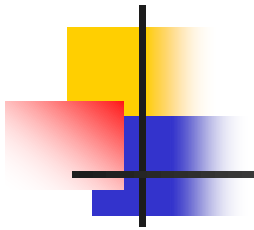


currentPos = i = 0

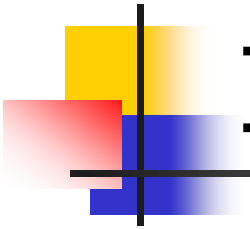

09
17
65
i
53
45
78
87
23
09
17
65
23
45
78
87
i
53

SiftDown Algorithm

```
template <class T, class E>
void MinHeap<T>::siftDown (int start, int m ) {
//私有函数: 从结点start开始到m为止, 自上向下比较,
//如果子女的值小于父结点的值, 则关键码小的上浮,
//继续向下层比较, 将一个集合局部调整为最小堆。
    int i = start,   j = 2*i+1;      //j是i的左子女位置
    E temp = heap[i];
    while (j <= m) {                          //检查是否到最后位置
        if ( j < m && heap[j] > heap[j+1] ) j++;
                                          //让j指向两子女中的小者
```



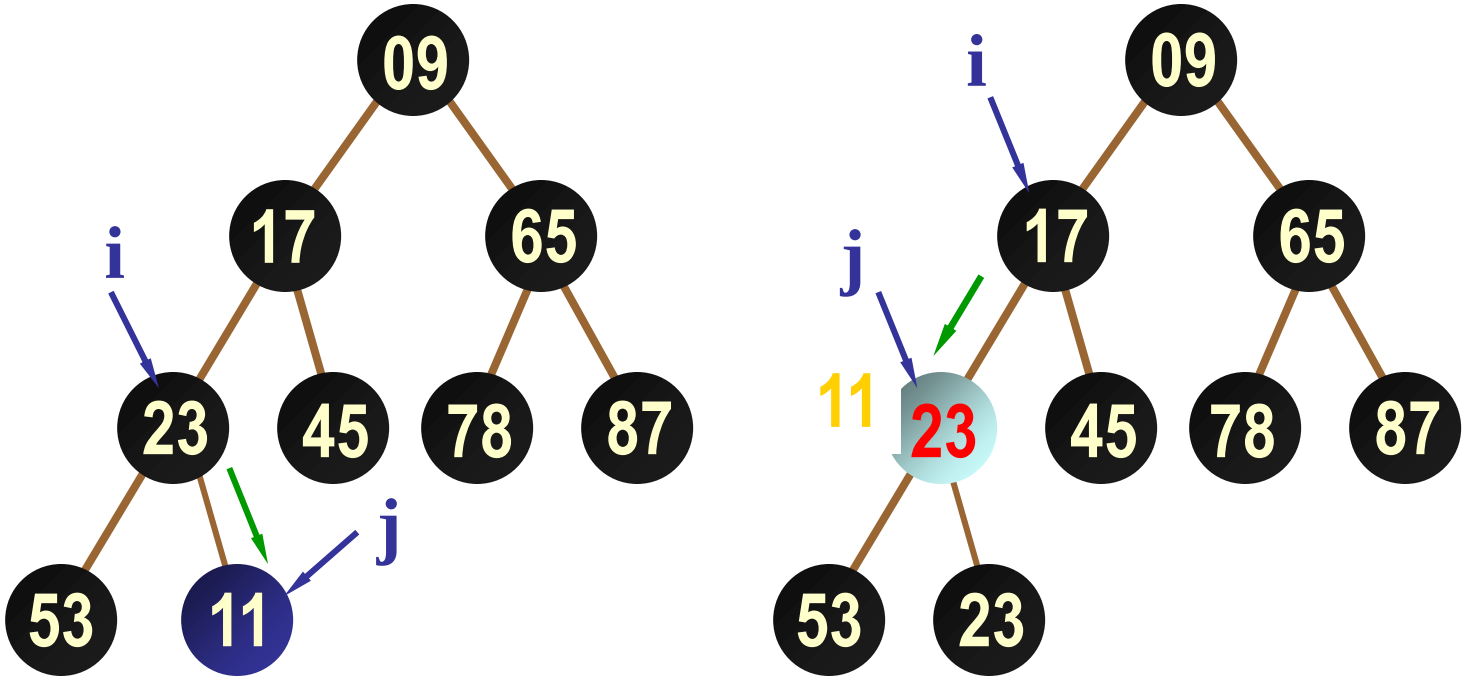
```
        if ( temp <= heap[j] ) break;   //小则不做调整
        else { heap[i] = heap[j];  i = j; j = 2*j+1; }
                                        //否则小者上移, i, j下降
    }
    heap[i] = temp;                     //回放temp中暂存的元素
};
```



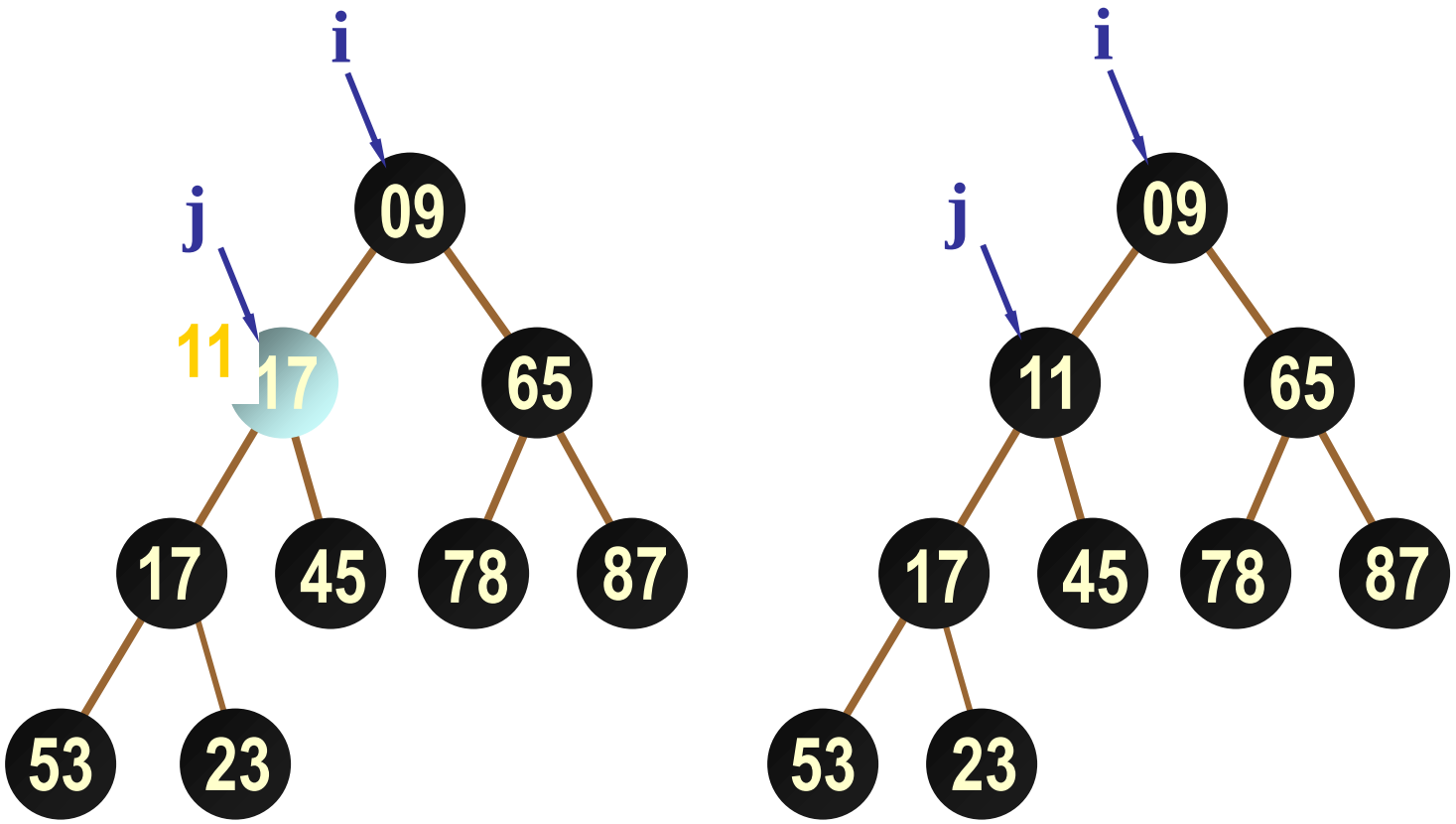
Insert Algorithm

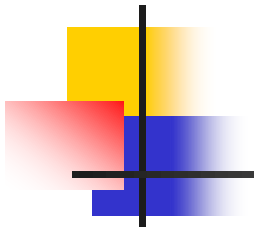
- 每次插入都加在堆的最后，再自下向上执行调整，使之重新形成堆
- 时间复杂性$O(\log_2 n)$

最小堆的向上调整



在堆中插入新元素11


i
j
11
09
17
65
17
45
78
87
53
23
i
j
09
11
65
17
45
78
87
53
23



```
template <class T, class E>
bool MinHeap<T>::Insert (const E& x )  {
//公共函数: 将X插入到最小堆中
    if ( currentSize == maxHeapSize ) //堆满
        { cerr << "Heap Full" << endl;  return false; }
    heap[currentSize] = x;                    //插入
    siftUp (currentSize);                              //向上调整
    currentSize++;                                     //堆计数加1
    return true;
};
```

```
template <class T, class E>
void MinHeap<T>::siftUp (int start) {
//私有函数: 从结点start开始到结点0为止, 自下向上比较,
//如果子女的值小于父结点的值, 则相互交换,
//这样将集合重新调整为最小堆。
//关键码比较符<=在E中定义。

    int j = start,  i = (j−1)/2;   E temp = heap[j];
    while (j > 0) {                          //沿父结点路径向上直达根
        if (heap[i] <= temp) break;
                    //父结点值小, 不调整
          else { heap[j] = heap[i];  j = i;  i = (i−1)/2; }
                                             //父结点结点值大, 调整
     }
heap[j] = temp;                                        //回送
};
```

Remove Algorithm

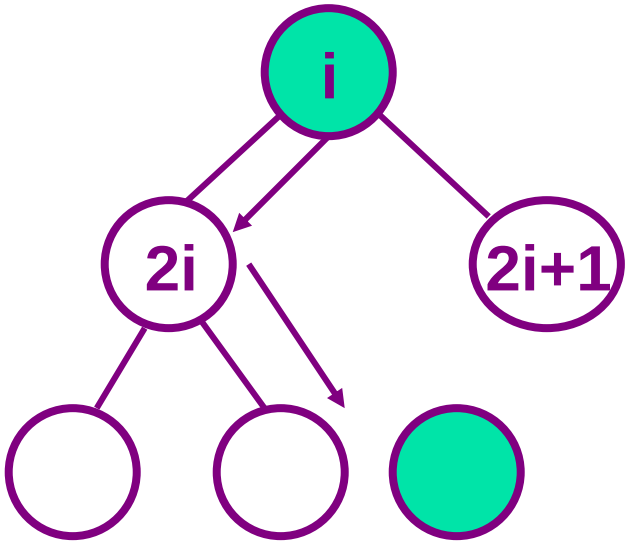
```
template <class T, class E>
bool MinHeap<T>::Remove (E& x) {
    if ( !currentSize ) {          //堆空, 返回false
        cout << "Heap empty" << endl;  return false;
    }
    x = heap[0];
    heap[0] = heap[currentSize−1];
    currentSize−−;
    siftDown(0, currentSize−1);        //自上向下调整为堆
    return true;                       //返回最小元素
};
```

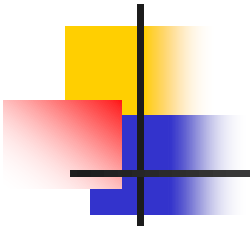
Max Heap

```
template <class T>
void MaxHeap<Type>::Push(const T& e)
{ // insert e into the max heap
   if (heapSize == capacity)  { // double the capacity
      ChangeSize1D(heap, capacity, 2*capacity);
      capacity *= 2;
   }
   int currentNode = ++heapSize;
   while (currentNode != 1 && heap[currentNode/2] < e)
   {  // bubble up
      heap[currentNode] = heap[currentNode/2];
      currentNode /=2;
   }
   heap[currentNode] = e;
}
```

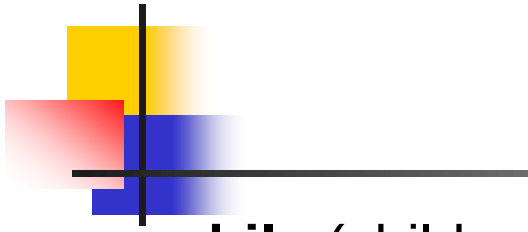
Deletion from a Max heap

- The max element is to be deleted from node 1
- Compare its key with that of its larger child
- All the way down until it is no smaller or reach the leaf---sift down.





```
template <class T>
void MaxHeap<T>::Pop()
{ // delete the max element.
   if (IsEmpty()) throw “Heap is empty. Cannot delete.”;
   heap[1].~T(); // delete the max

   // remove the last element from heap
   T lastE = heap[heapSize--];

   // trickle down
   int currentNode = 1;  // root
   int child = 2;      // left child of currentNode
```



```
    while (child <= heapSize)
    {
       // set child to the larger child of currentNode
       if (child<heapSize && heap[child]<heap[child+1]) child++;

       // can we put lastE in currentNode?
       if (lastE>=heap[child]) break;  // yes

       // no
       heap[currentNode]=heap[child]; // move child up
       currentNode=child; child*=2; // move down a level
     }
    heap[currentNode]=lastE;
}
```