Data Structure and Algorithms

Chapter 7 Tree and Binary Trees

Dr. Zhiqiang Liu

School of Software and Microelectronics, Northwest Polytechnical University

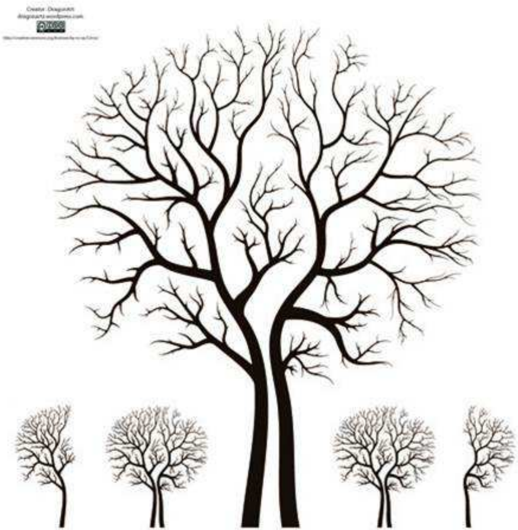


Table of Contents

Next Section

7.1 Tree Definition and Concepts

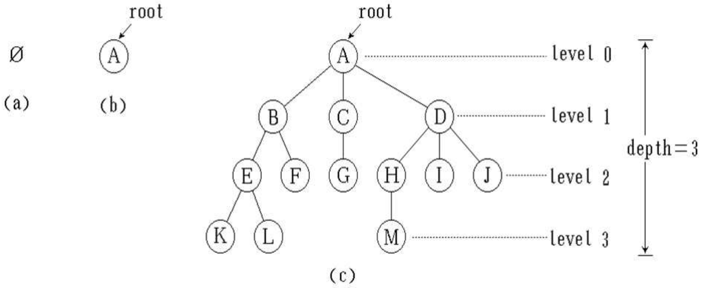
- Tree Definition: Trees consist of **n** nodes such that the following conditions:
  - **Only one node has no predecessor: the root.**
  - **Every node other than the root has a unique predecessor.**
  - **Starting at any node, one can reach the root by repeatedly stepping from a node to its predecessor.**
  - **Except the root, the remainders can be partitioned into $m(m > 0)$ un-overlapped subsets $T_0, T_1, \ldots, T_{m-1}$, each of which is also a tree. They are called as the sub trees of the root.**
  - **If $n = 0$, it is a NULL tree.**

Note: The root of the tree has no predecessor and 0 to many successors (sub trees).

7.1 Tree Definition and Concepts

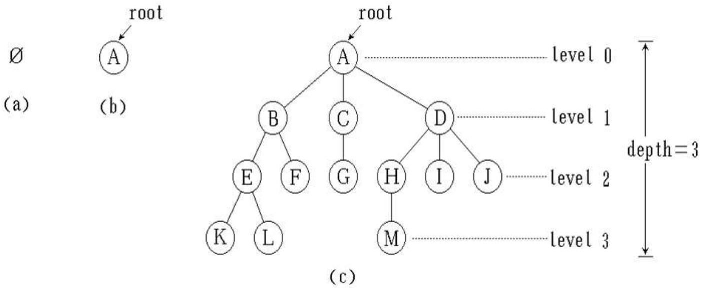
- **Tree**
  - NULL tree
  - Root tree
  - Generic tree

- Node
- Branch
- Degree of the node
- Degree of tree
- Branch node
- Leaf node
- Child node
- Parent node



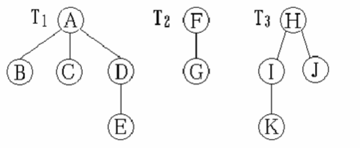
7.1 Tree Definition and Concepts

- Sibling node
- Ancestor node
- Descendent node
- Level
- Depth of the tree



7.1 Tree Definition and Concepts

- Ordered tree
- Non-ordered tree
- Forest
  - A set of trees.
- Orchard
  - An ordered set of ordered trees.
  - Ordered forest.



Next Section

Next Subsection

7.2.1 Binary Tree Definition

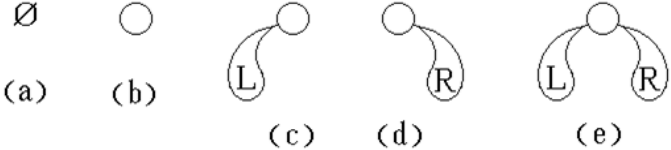
- A **binary tree** is either empty, or it consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree** of the root.

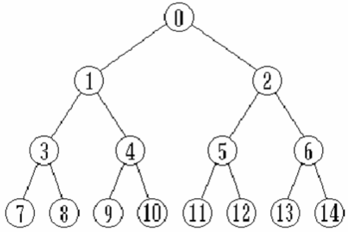
Binary tree = (Root,
Left sub binary tree,
Right sub binary tree)

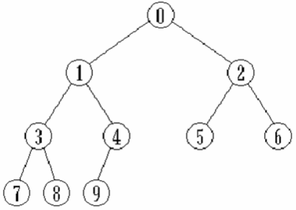
7.2.1 Binary Tree Definition

Examples

- Binary tree





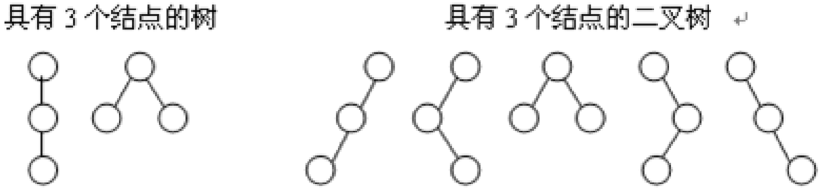


7.2.1 Binary Tree Definition

- Draw all different forms of the trees and binary trees with three nodes respectively

7.2.1 Binary Tree Definition

- Draw all different forms of the trees and binary trees with three nodes respectively

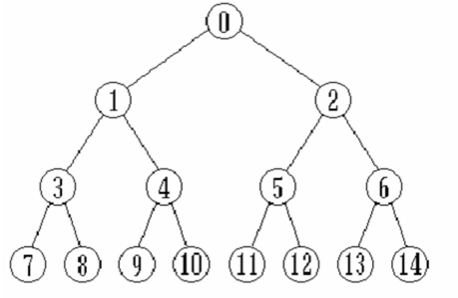


Next Subsection

7.2.2 Characteristics of Binary Tree

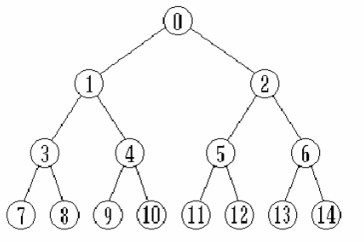
1. There are at most 2^i nodes at level $i(i \geq 0)$.
2. If the height of binary tree is $h(h \geq$-1$)$, the number of nodes is at most $2^{h+1} - 1$
3. In binary tree, if the number of leaves is n_0,and the number of nodes with left and right children is n_2, then:

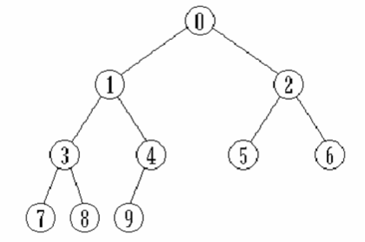
$$n_0 = n_2 + 1$$



7.2.2 Characteristics of Binary Tree

- Full Binary Tree
  - A binary tree in which every level, except possibly the deepest, is completely filled.
- Complete Binary Tree





7.2.2 Characteristics of Binary Tree

4. The depth of the complete binary tree with n nodes is:

$$\lceil \log_2(n+1) \rceil - 1$$

Proof: Assuming the depth of the complete binary tree is h, it is obvious that:

$$2^h - 1 < n \le 2^{h+1} - 1 \qquad 2^h < n+1 \le 2^{h+1}$$

$$h < \lceil \log_2(n+1) \rceil < h+1$$

So that the depth h satisfies:

$$\lceil \log_2(n+1) \rceil - 1$$

7.2.2 Characteristics of Binary Tree

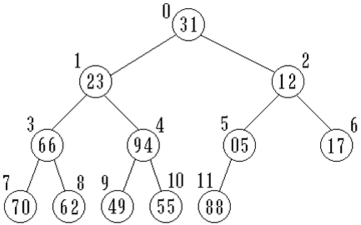
5. If we number all the nodes of the complete binary tree from top to the bottom and from left to right at the same level, the nodes have been numbered as $0, 1, 2, \ldots (n-1)$
   - If $i == 0$, then i has no parent. (it is the root); If $i > 0$, then its parent is $\lfloor (i-1)/2 \rfloor$
   - If $2*i+1 < n$, then the left child of i is $2*i+1$.
     If $2*i+2 < n$, then the right child of i is $2*i+2$.
   - If i is even and i!=0, then the left child of i is $i-1$.
     If i is odd and i!=$n-1$, then the right child of i is $i+1$.
   - The Level of node i is $\lfloor \log_2(n+1) \rfloor$ $\square$

ADT of Binary Tree

```
template <class Type>
class BinaryTree{
public:
    BinaryTree();
    BinaryTree(BinTreeNode<Type> * lch,
        BinTreeNode<Type> * rch, Type item);
    int IsEmpty();
    BinTreeNode<Type> *Parent();
    BinTreeNode<Type> *LeftChild();
    BinTreeNode<Type> *RightChild();
    int Insert(const Type &item);
    int Find(const Type &item) const;
    Type GetData() const;
    const BinTreeNode<Type> *GetRoot() const;
}
```

Next Subsection

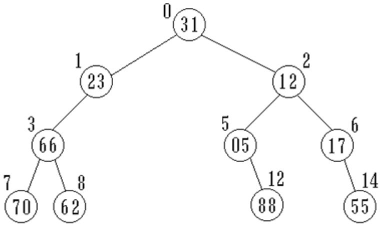
7.2.3 Binary tree represented as array and linked list



(a)

0	1	2	3	4	5	6	7	8	9	10	11
31	23	12	66	94	05	17	70	62	49	55	88

(b)



(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
31	23	12	66		05	17	70	62				88		55

(b)

7.2.3 Binary tree represented as array and linked list

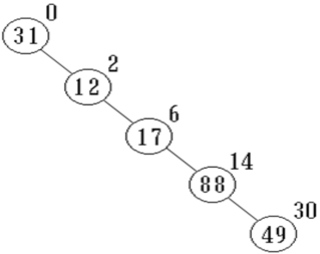
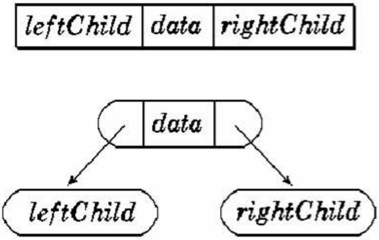


Figure 2.1: Single Branch Tree

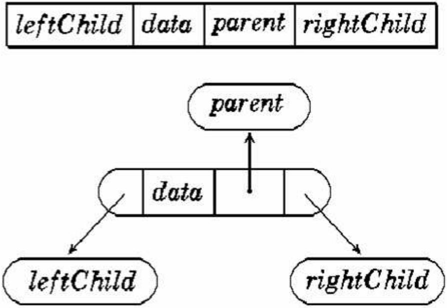
31		12				17								88																30

7.2.3 Binary tree represented as array and linked list

- Linked list
  - Bi-branch list
  - Tri-branch list



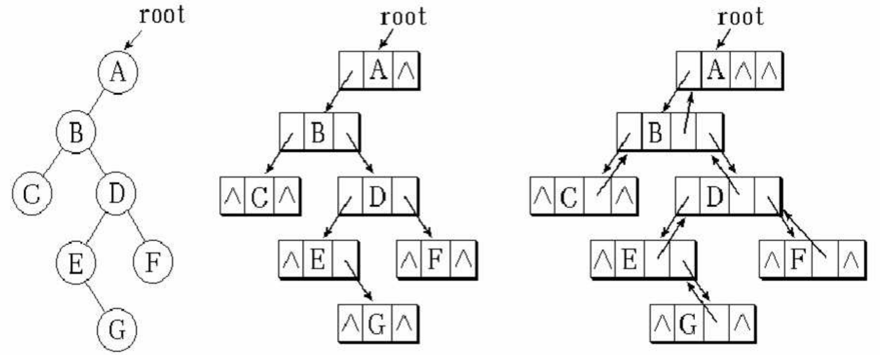
(a) Bi-branch list



(b) Tri-branch list

7.2.3 Binary tree represented as array and linked list

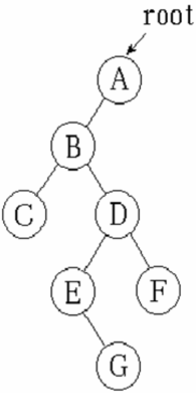
Examples of linked list for binary tree



(a) Binary tree (b) Bi-branch linked list (c) Tri-branch linked list

7.2.3 Binary tree represented as array and linked list

Binary tree represented in static linked list array



(a) Binary tree

	data	*parent*	*leftChild*	*rightChild*
0	A	−1	1	−1
1	B	0	2	3
2	C	1	−1	−1
3	D	1	4	5
4	E	3	−1	6
5	F	3	−1	−1
6	G	4	−1	−1

Class of Binary Tree

```
template<class Type> class BinaryTree;
template<class Type> class BinTreeNode
{
    friend class BinaryTree<Type>;
private:
    BinTreeNode<Type> *leftChild, *rightChild;
    Type data;
public:
    BinTreeNode():leftChild(NULL),rightChild(NULL){}
    BinTreeNode(Type item, BinTreeNode<Type> *left=NULL,
        BinTreeNode<Type> *right=NULL)
        :data(item), leftChild(left),
        rightChild(right){}
    Type GetData()const{ return data; }
    BinTreeNode<Type> *GetLeft()const{
        return leftChild; }
    BinTreeNode<Type> *GetRight()const{
        return rightChild;}
```

Class of Binary Tree

```
    void SetData(const Type& item){
        data = item;}
    void SetLeft(BinTreeNode<Type> *L){
        leftChild = L;}
    void SetRight(BinTreeNode<Type> *R){
        rightChild = R;}
};

template <class Type> class BinaryTree{
public:
    BinaryTree():root(NULL){}
    BinaryTree(Type value):RefValue(value),root(NULL){}
    virtual ~BinaryTree(){destroy(root);}
    virtual int IsEmpty(){return root==NULL?1:0;}
    virtual BinTreeNode<Type> *Parent(BinTreeNode<Type>
        *current){
        return root==NULL||root==current?NULL:Parent(
            root,current);}
```

Class of Binary Tree

```
    virtual BinTreeNode<Type> *LeftChild(BinTreeNode<
        Type> *current){
        return root!= NULL?current->leftChild:NULL;}
    virtual BinTreeNode<Type> *RightChild(BinTreeNode<
        Type> *current){
        return root!=NULL?current->rightChild:NULL;}
    virtual int Insert(const Type& item);
    virtual int Find(const Type&item)const;
    const BinTreeNode<Type> *GetRoot()const{return root
        ;}
    friend istream &operator >>(istream &in, BinaryTree<
        Type> &Tree)
    friend ostream &operator <<(ostream &out, BinaryTree
         <Type> &Tree)
private:
    BinTreeNode<Type> *root;
    Type RefValue;
```

Class of Binary Tree

```
    BinTreeNode<Type> *Parent(BinTreeNode<Type> *start, BinTreeNode<Type> *current);
    int Insert(BinTreeNode<Type> * &current, const Type &item);
    void Traverse(BinTreeNode<Type> *current, ostream &out)const
    int Find(BinTreeNode<Type> *current, const Type &item)const
    void destroy(BinTreeNode<Type> *current);
}
```

QUIZ

- 一棵深度为6的满二叉树有__________个分支结点和__个叶子。

- 一棵具有257个结点的完全二叉树，它的深度_。

- 设一棵完全二叉树具有1000个结点，则此完全二叉树有___个叶子结点，有___个度为2的结点，有_个结点只有非空左子树，有_个结点只有非空右子树。

QUIZ

- 一棵深度为6的满二叉树有$\underline{n_0 - 1 = 31}$个分支结点和$\underline{32}$个叶子。
  - 注：深度为K的二叉树最多有$2^K - 1$个结点
  - 对于任意一棵二叉树，如果度为0的结点个数为n_0，度为2的结点个数为n_2，则$n_0 = n_2 + 1$。
  - 满二叉树没有度为1的结点，所以分支结点数就是二度结点数。
- 一棵具有257个结点的完全二叉树，它的深度_。
- 设一棵完全二叉树具有1000个结点，则此完全二叉树有___个叶子结点，有___个度为2的结点，有_个结点只有非空左子树，有_个结点只有非空右子树。

QUIZ

- 一棵深度为6的满二叉树有$\underline{n_0 - 1 = 31}$个分支结点和$\underline{32}$个叶子。
  - 注：深度为K的二叉树最多有$2^K - 1$个结点
  - 对于任意一棵二叉树，如果度为0的结点个数为n_0，度为2的结点个数为n_2，则$n_0 = n_2 + 1$。
  - 满二叉树没有度为1的结点，所以分支结点数就是二度结点数。
- 一棵具有257个结点的完全二叉树，它的深度$\underline{9}$。
  - 注：用$\lfloor \log_2 n \rfloor + 1 = \lfloor 8.xx \rfloor + 1 = 9$□
- 设一棵完全二叉树具有1000个结点，则此完全二叉树有___个叶子结点，有___个度为2的结点，有_个结点只有非空左子树，有_个结点只有非空右子树。

QUIZ

- 一棵深度为6的满二叉树有$\underline{n_0 - 1 = 31}$个分支结点和$\underline{32}$个叶子。
  - 注：深度为K的二叉树最多有$2^K - 1$个结点
  - 对于任意一棵二叉树，如果度为0的结点个数为n_0，度为2的结点个数为n_2，则$n_0 = n_2 + 1$。
  - 满二叉树没有度为1的结点，所以分支结点数就是二度结点数。
- 一棵具有257个结点的完全二叉树，它的深度$\underline{9}$。
  - 注：用$\lfloor \log_2 n \rfloor + 1 = \lfloor 8.xx \rfloor + 1 = 9$□
- 设一棵完全二叉树具有1000个结点，则此完全二叉树有$\underline{500}$个叶子结点，有$\underline{499}$个度为2的结点，有$\underline{1}$ 个结点只有非空左子树，有$\underline{0}$个结点只有非空右子树。
  - 答：最快方法：用叶子数$= \lceil n/2 \rceil = 500, n_2 = n_0 - 1 = 499$。另外，最后一结点为$2i$属于左叶子，右叶子是空的，所以有1 个非空左子树。完全二叉树的特点决定不可能有左空右不空的情况，所以非空右子树数$= 0$.

QUIZ

- 一棵度为2的树与一棵二叉树有何区别?

- 二叉树是非线性数据结构，所以____ 。
  - （A）它不能用顺序存储结构存储;
  - （B）它不能用链式存储结构存储;
  - （C）顺序存储结构和链式存储结构都能存储;
  - （D）顺序存储结构和链式存储结构都不能使用

- Proof: For a Full k Tree, the number of leaf node n_0 and the number of Non-leaf node n_1 satisfy: $n_0 = (k-1)n_1 + 1$

QUIZ

- 一棵度为2的树与一棵二叉树有何区别?
  - 答：度为2的树从形式上看与二叉树很相似，但它的子树是无序的，而二叉树是有序的。即，在一般树中若某结点只有一个孩子，就无需区分其左右次序，而在二叉树中即使是一个孩子也有左右之分。
- 二叉树是非线性数据结构，所以___ 。
  - （A）它不能用顺序存储结构存储;
  - （B）它不能用链式存储结构存储;
  - （C）顺序存储结构和链式存储结构都能存储;
  - （D）顺序存储结构和链式存储结构都不能使用
- Proof: For a Full k Tree, the number of leaf node n_0 and the number of Non-leaf node n_1 satisfy: $n_0 = (k-1)n_1 + 1$

QUIZ

- 一棵度为2的树与一棵二叉树有何区别?
  - 答：度为2的树从形式上看与二叉树很相似，但它的子树是无序的，而二叉树是有序的。即，在一般树中若某结点只有一个孩子，就无需区分其左右次序，而在二叉树中即使是一个孩子也有左右之分。
- 二叉树是非线性数据结构，所以 C 。
  - （A）它不能用顺序存储结构存储;
  - （B）它不能用链式存储结构存储;
  - （C）顺序存储结构和链式存储结构都能存储;
  - （D）顺序存储结构和链式存储结构都不能使用
- Proof: For a Full k Tree, the number of leaf node n_0 and the number of Non-leaf node n_1 satisfy: $n_0 = (k-1)n_1 + 1$

Next Section

Next Subsection

7.3.1 Preorder, Inorder, Postorder traversal

- Definition
- Parts

•Root	**V**	
•Left sub-tree	**L**	**Lc=Left child**
•Right sub-tree	**R**	**Rc=Right child**

- Rules

•Preorder	**VLR**
•Inorder	**LVR**
•Postorder	**LRV**

7.3.1 Preorder, Inorder, Postorder traversal

Inorder traversal

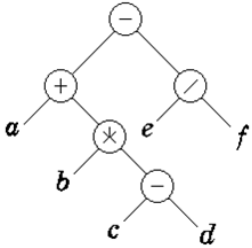
- Framework of Inorder traversal
  - If binary tree T is NULL, NULL operation else

    Inorder traverse the left sub-tree

    Visit root

    Inorder traverse the right sub-tree

Result of Inorder traversal:



7.3.1 Preorder, Inorder, Postorder traversal

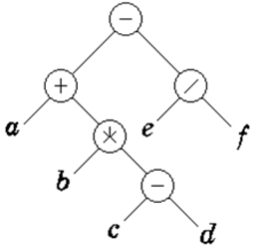
Inorder traversal

- Framework of Inorder traversal
  - If binary tree T is NULL, NULL operation else

    Inorder traverse the left sub-tree
    Visit root
    Inorder traverse the right sub-tree

Result of Inorder traversal:

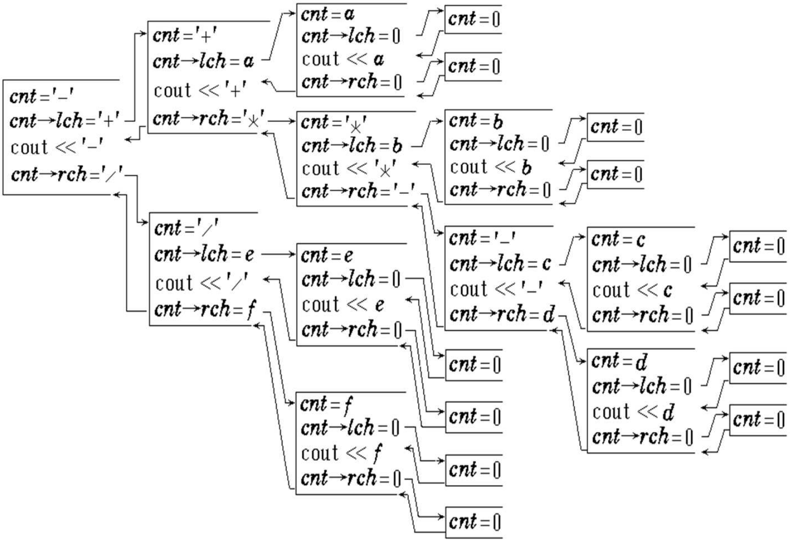
$a + b * c - d - e / f$



Recursive algorithm of Inorder Traversal

```
template <class Type>
void BinaryTree <Type>::InOrder(){
    InOrder(root);
}
template <class Type> void BinaryTree<Type>::InOrder(
    BinTreeNode<Type> *current){
        if(current!=NULL){
            InOrder(current->leftChild);
            cout<<current->data;
            InOrder(current->rightChild);
        }
}
```

7.3.1 Preorder, Inorder, Postorder traversal



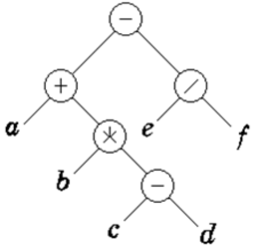
7.3.1 Preorder, Inorder, Postorder traversal

Preorder traversal

- Framework of Preorder traversal

  If binary tree T is NULL, NULL operation
  else

  Visit root
  Preorder traverse the left sub-tree
  Preorder traverse the right sub-tree



Result of Preorder traversal

$- + a * b - cd / ef$

7.3.1 Preorder, Inorder, Postorder traversal

Recursive algorithm of Preorder Traversal

```
template <class Type>
void BinaryTree <Type>::PreOrder(){
    PreOrder(root);
}
template<class Type> void BinaryTree<Type>::
    PreOrder(BinTreeNode<Type> *current){
    if (current!=NULL){
       cout<<current->data;
       PreOrder(current->leftChild);
       PreOrder(current->rightChild);
    }
}
```

7.3.1 Preorder, Inorder, Postorder traversal

Postorder traversal

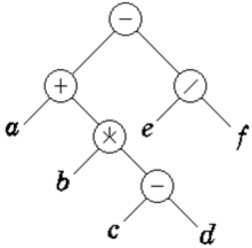
- Framework of Postorder traversal

  If binary tree T is NULL, NULL operation
  else

  Postorder traverse the left sub-tree
  Postorder traverse the right sub-tree
  Visit root

  Result of Postorder traversal

  $abcd - * + ef / -$



7.3.1 Preorder, Inorder, Postorder traversal

Recursive algorithm of Preorder Traversal

```
template <class Type> void
    BinaryTree <Type>::PostOrder ( ) {
    PostOrder ( root );
}

template <class Type> void BinaryTree<Type>::
PostOrder ( BinTreeNode <Type> *current ) {
    if ( current != NULL ) {
       PostOrder ( current->leftChild );
       PostOrder ( current->rightChild );
       cout << current->data;
    }
}
```

Applications

- Compute **the number of nodes** in BT

```
template <class Type>
int BinaryTree<Type>::Size(
const BinTreeNode<Type> *t) const{
    if(t == NULL)
        return 0;
    else
        return 1 + Size(t->leftChild)
                 + Size(t->rightChild);
}
```

Applications

- Compute **the depth of** BT

```
template <class Type>
int BinaryTree<Type>::Depth(const BinTreeNode <Type> *t
    ) const{
    if(t == NULL)
        return -1;
    else
        return 1 +
            Max(Depth(t->leftChild),
            Depth(t->rightChild));
}
```

Next Subsection

7.3.2 Non recursive algorithm for traversal

```c
//Node structure
typedef struct BitreeNode {
     TreeDataType  data;
     struct BitreeNode *
         leftChild;
     struct BitreeNode *
         rightChild;
} BitreeNode,  * Bitree;
```

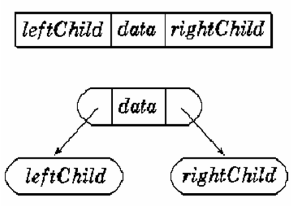
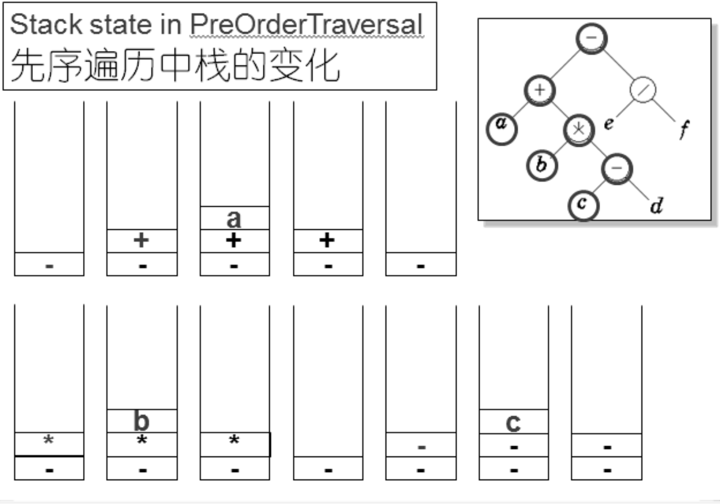


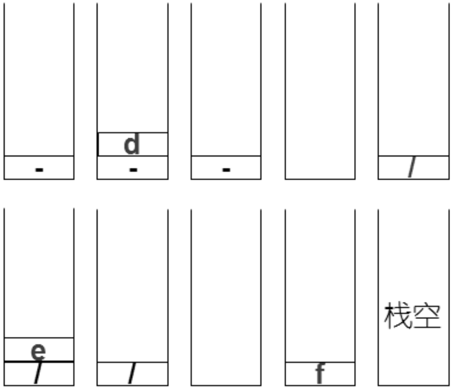
Figure 3.1: Stack

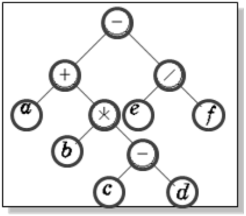
7.3.2 Non recursive algorithm for traversal



7.3.2 Non recursive algorithm for traversal

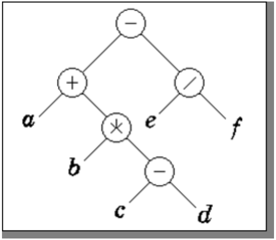
Stack state in PreOrderTraversal
先序遍历中栈的变化





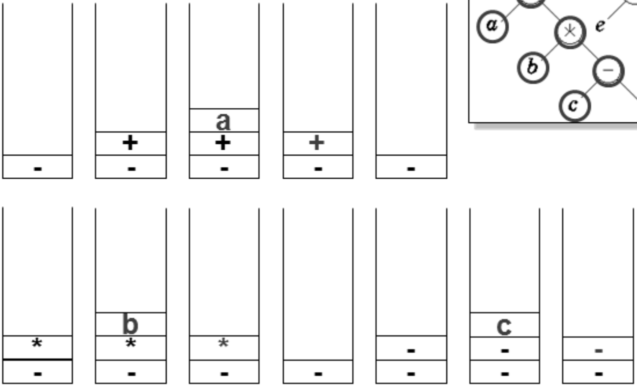
7.3.2 Non recursive algorithm for traversal

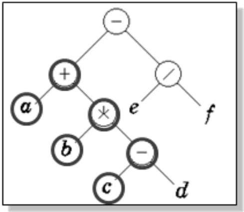
```
//Preorder Traversal
void PreOrderTraverse(Bitree T){
    StackType S; BitreeNode *p;
    S.makeEmpty(); p = T;
    do{ while(p){
            printf(p->data);
            S.Push(p);
            p = p->leftChild;
        }
        if(!S.IsEmpty()){
            p = S.getTop(); S.Pop();
            p = p->rightChild;
        }
    }while(p || !S.IsEmpty());
}
```



7.3.2 Non recursive algorithm for traversal

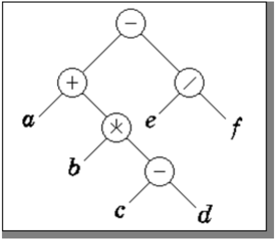
Stack state in InOrderTraversal
中序遍历中栈的变化



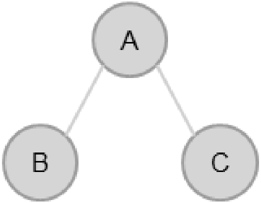


7.3.2 Non recursive algorithm for traversal

```
//InOrder Traversal
void InOrderTraverse(Bitree T){
    StackType S; BitreeNode *p;
    S.makeEmpty(); p = T;
    do{
       while(p){
          S.Push(p); p=p->leftChild;
       }
       if(!S.IsEmpty()){
          p = S.getTop(); S.Pop();
          printf(p->data);
          p = p->rightChild;
       }
    }while(p||!S.IsEmpty());
}
```

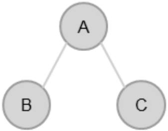


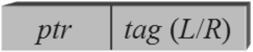
Postorder Traversal with Non-Recursive



7.3.2 Non recursive algorithm for traversal

```
typedef struct StackNode
{
    enum tag{L,R};
    BitreeNode *ptr;
} StackNode;
```





Consider push stack twice!!!

7.3.2 Non recursive algorithm for traversal

```
//Postorder Traversal
typedef struct StackNode {
    enum tag{L,R};
    BitreeNode * ptr;
} StackNode;
void PostOrderTraverse(Bitree T){
    StackType S; BitreeNode *p;
        StackNode w;
    S.makeEmpty(); p=T;
    do{
        while(p){
            w.ptr=p; w.tag=L;
            S.Push(w);
            p=p->leftChild;
        }
```

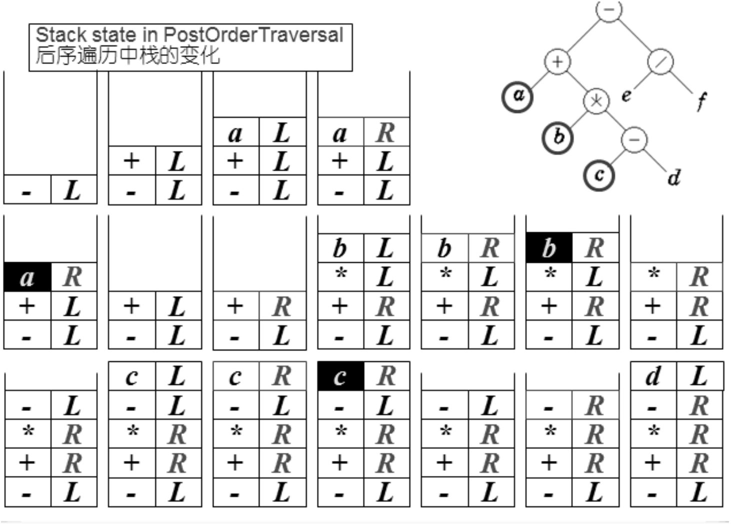
ptr	$tag\ (L/R)$

7.3.2 Non recursive algorithm for traversal

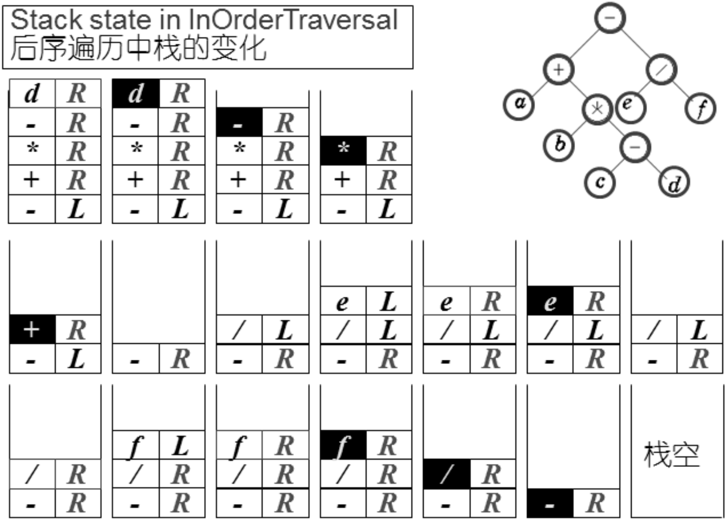
```
        int continue = 1;
          while(continue&&!S.IsEmpty()){
            w = S.getTop(); S.Pop();
            p = w.ptr;
            switch(w.tag){
            case L: w.tag=R; S.Push(w);
                continue = 0;
                p = p->rightChild;
                break;
            case R: printf(p->data);break;
            }
        }
    }while(p||!S.IsEmpty());
}
```

需要考虑二次进栈！！！

7.3.2 Non recursive algorithm for traversal



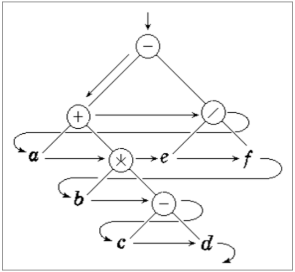
7.3.2 Non recursive algorithm for traversal



7.3.2 Non recursive algorithm for traversal

Level order traversal

- From the root, visit the node from the top to the bottom, and from the left to the right at the same level.
- Queue involved



7.3.2 Non recursive algorithm for traversal

Levelorder Iterator Class

```
template <class Type> class LevelOrder :
    public TreeIterator <Type> {
public:
    LevelOrder(const BinaryTree<Type> &BT);
    ~LevelOrder(){}
    void First();
    // Advance an item
    void operator++();
protected:
    Queue<const BinTreeNode<Type>*> qu;
};
```

7.3.2 Non recursive algorithm for traversal

```
template<class Type> LevelOrder<Type>::
LevelOrder(const BinaryTree<Type> &BT):
TreeIterator<Type> (BT)
{
    qu.EnQueue(T.GetRoot());
}

template <class Type>
void LevelOrder<Type>::First(){
    qu.MakeEmpty();
    if(T.GetRoot())
        qu.EnQueue( T.GetRoot() );
    operator++();
}
```

7.3.2 Non recursive algorithm for traversal

```
// Advance an item, EnQueue its left and right child
template <class Type>
void LevelOrder<Type>::operator++(){
    if(qu.IsEmpty()){
       if(current==NULL)
         {cout“<<End of traversal”!<<endl;exit(1);}
       current=NULL; return;
    }
    current=qu.DeQueue(); //DeQueue
    if(current->GetLeft()!=NULL) //Left child
        qu.EnQueue(current->GetLeft()); //EnQueue
    if(current->GetRight()!=NULL) //Right child
        qu.EnQueue(current->GetRight()); //EnQueue
}
```

Tree Iterator

4个功能：定位到第一个节点；定位到下一个节点；判定是否到达最后一个节点；访问当前节点。

```
//Base class for BT Iterator
template<class Type> class TreeIterator {
public:
    TreeIterator(const BinaryTree<Type>& BT)
        : T(BT),current(NULL){}
    virtual ~TreeIterator(){}
    //Located to First (Implemented in derived classes)
    virtual void First()=0;
    // Next node
    virtual void operator ++()=0;
    // Judge validity of current node
    int operator +()const{return current!=NULL;}
    // Return the value of current
    const Type& operator()()const;
```

7.3.2 Non recursive algorithm for traversal

```
protected:
    const BinaryTree<Type>& T; //BT
    const BinTreeNode<Type>* current;
private:
    TreeIterator(const TreeIterator&){}
    // Assignment
    TreeIterator& operator=(const TreeIterator&)const;
};
template<class Type> const
  Type& TreeIterator<Type>::operator()const {
    if(current==NULL){
        cout<<"Invalid visit!"<<endl;exit(1);
    }
    return current->GetData();
}
```

7.3.2 Non recursive algorithm for traversal

Postorder Iterator

- Stack for active record

Address: **Node*	**Counter:** ***S.PopTime***

- S.PopTime = 0, 1 or 2,
- Push the root of sub-tree and set

  S.PopTime = 0
- Before postorder traversing the left sub-tree, change

  S.PopTime = 1

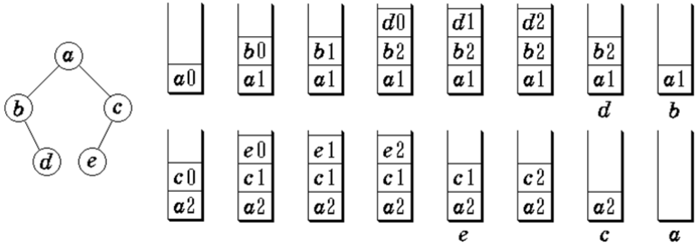
  and push the left child into the stack.
- After postorder traversing the left sub-tree completely, it is necessary to traverse the right sub-tree.

7.3.2 Non recursive algorithm for traversal

- Before postorder traversing the right sub-tree, change

  S.PopTime = 2

  and push the right child into the stack.
- After traversing the right sub-tree, it is turn to visit the root of sub-tree and pop it from the stack.



7.3.2 Non recursive algorithm for traversal

Postorder Iterator Class

```
template<class Type> struct stkNode {
    //Stack node definition
    const BinTreeNode<Type> *Node;//Node Address
    int PopTime //Counter
    stkNode(const BinTreeNode<Type> *N=NULL):Node(N),
        PopTime(0){}
}
```

Address: **Node*	**Counter:** *S.PopTime*

7.3.2 Non recursive algorithm for traversal

```
template<class Type> class PostOrder:public TreeIterator
    <Type>{
    public:
        PostOrder(const BinaryTree <Type>& BT);
        ~PostOrder(){}
        //Seek to the first node in postorder traversal
        void First();
        //Seek to the successor
        void operator ++ ( );
    protected:
        //Active record stack
        Stack <StkNode<Type>> st;
}
```

7.3.2 Non recursive algorithm for traversal

```
template <class Type> PostOrder <Type> ::
PostOrder(const BinaryTree<Type> &BT):
    TreeIterator<Type> (BT){
        st.Push(StkNode<Type>(T.GetRoot()));
}
template <class Type> void  PostOrder <Type> :: First(){
    st.MakeEmpty();
    if(T.GetRoot()!=NULL)
      st.Push(StkNode<Type>(T.GetRoot()));
    operator ++();
}
```

7.3.2 Non recursive algorithm for traversal

```
template<class Type> void PostOrder<Type>::operator++(){
    if (st.IsEmpty()){
        if (current == NULL){
           cout << “End of traversal”!<< endl; exit(1);}
        current=NULL;  return;  //finish traversal
     }
     StkNode<Type> Cnode;
     for(;;){  //find out the successor
        Cnode = st.Pop();
        if (++Cnode.PopTime == 3){ //exit from right sub-tree
            current = Cnode.Node;   return;
        }
     st.Push(Cnode);  //push Cnode into stack
     if(Cnode.PopTime==1){//push leftchild of Cnode into stack
        if(Cnode.Node->GetLeft()! = NULL)
            st.Push(StkNode<Type> (Cnode.Node->GetLeft()));
     } else {//push the right child of Cnode into stack
        if(Cnode.Node->GetRight()!=NULL)
            st.Push(StkNode<Type>(Cnode.Node->GetRight()));
     }
}
```

Inorder Iterator Class

```
template<class Type> class InOrder :
    public PostOrder<Type>{
public:
    InOrder(BinaryTree<Type>& BT):PostOrder<Type>(BT){}
    void First();
    void operator++();
};
```

7.3.2 Non recursive algorithm for traversal

```
template<class Type>
void InOrder<Type>::operator++(){
    if(st.IsEmpty()){
        if(current==NULL){cout<< “End of Traversal”!<<endl;exit
            (1);}
        current=NULL; return;}
    StkNode <Type> Cnode;
    for (;;){//find out the inorder successor
        Cnode=st.Pop(); //pop
        if(++Cnode.PopTime==2){//exit from the left sub-tree
            current = Cnode.Node;
            if(Cnode.Node->GetRight()!=NULL)
                st.Push(StkNode<Type>(Cnode.Node->GetRight()));
            return;
    }
    st.Push(Cnode);//+1 and push cnode into stack
    if(Cnode.Node->GetLeft()!=NULL)
        //push the left child of Cnode in stack
        st.Push(StkNode<Type> (Cnode.Node->GetLeft()));
    }
}
```

Preorder Iterator Class

```
template<class Type> class PreOrder:public TreeIterator<
    Type>{
public:
    PreOrder(const BinaryTree<Type>& BT );
    ~PreOrder(){}
    void First();
    //Seek to the first node in preorder traversal
    void operator++(); //Seek to the successor
protected:
    Stack<const BinTreeNode<Type> *> st;//Active stack
}
```

7.3.2 Non recursive algorithm for traversal

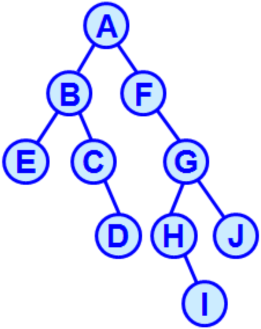
```
template <class Type>PreOrder<Type>::PreOrder(const
    BinaryTree<Type> &BT):
    TreeIterator<Type>(BT){
        st.Push(BT.GetRoot()));
}
template<class Type> void PreOrder<Type>::First(){
    st.MakeEmpty();
    if(BT.GetRoot())
        st.Push(BT.GetRoot()));
    operator++();
}
```

7.3.2 Non recursive algorithm for traversal

```
template<class Type> void PreOrder<Type>::operator ++(){
    if(st.IsEmpty()){
        if(current==NULL){
            cout<<"End␣of␣traversal"<<endl;
            exit(1);
        }
        current=NULL; return; //finish traversal
    }
    current=st.Pop();
    if(current->GetRight()!=NULL)
        st.Push(current->GetRight());
    if(current->GetLeft()!=NULL)
        st.Push(current->GetLeft());
}
```

QUIZ

- Please write down the node sequence with Pre-order, Inorder and Postorder traversal.
- Describe the stack status of non-recursive algorithm in above steps.



找出所有满足下列条件的二叉树：

1）它们在先序遍历和中序遍历时，得到的结点访问序列相同；

2）它们在后序遍历和中序遍历时，得到的结点访问序列相同；

3）它们在先序遍历和后序遍历时，得到的结点访问序列相同。

Next Section

7.4 Binary Tree Reconstruction

Problem 1

Given a binary tree, is Preorder sequence, or Inorder sequence, or Postorder sequence Unique?

Problem 2

- Given a preorder sequence of BT, can you reconstruct an unique binary tree?
- How about Inorder sequence, postorder sequence?

7.4 Binary Tree Reconstruction

Problem 3

- Given preorder and inorder sequences of BT, can you reconstruct this binary tree?

  Preorder: abcd;   Inorder: badc;

- Given postorder and inorder sequences of BT, can you reconstruct this binary tree?

  Postorder: bdca;   Inorder: badc;

- Given preorder and Postorder sequences of BT, can you reconstruct this binary tree?

  Preorder: abcd;   Postorder: dcba;

7.4 Binary Tree Reconstruction

Example

- Suppose that the preorder sequence of BT is

  ABECDFGHIJ

  and the inorder sequence of BT is

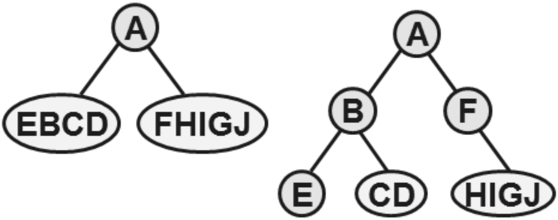
  EBCDAFHIGJ

  try to reconstruct this binary tree.

7.4 Binary Tree Reconstruction

Preorder sequence: <u>A</u>BECDFGHIJ

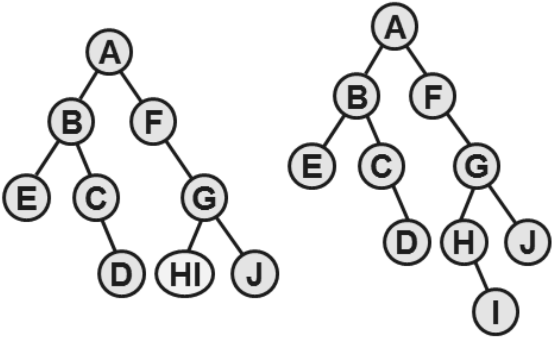
Inorder sequence: EBCD<u>A</u>FHIGJ



7.4 Binary Tree Reconstruction

Preorder sequence: ABECDFGHIJ

Inorder sequence: EBCDAFHIGJ



7.4 Binary Tree Reconstruction

Theorem

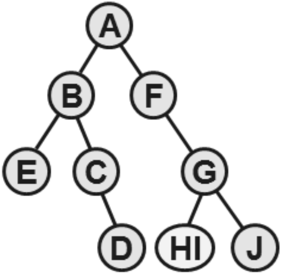
- Given the preorder sequence pre$[1,\ldots,n]$ and inorder sequence in$[1,\ldots,n]$ of a binray tree, the topological structure of the binary tree is unique.

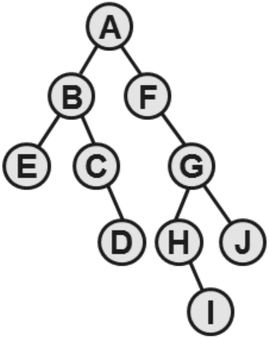
  Proof: Suppose the nodes are not same.if $n=1$, if is obvious to generate a root.if $n<k$, the proposition is correct, we now see the case for $n=k$.the pre and in arrays can be partitioned into two sub-arrays of pre $[2,\ldots,m]$, in$[1,\ldots,m-1]$ and pre$[m+1,\ldots,n]$, in$[m+1,\ldots,n]$.

7.4 Binary Tree Reconstruction

Preorder sequence: ABECD FGHIJ

Inorder sequence: EBCD A FHIGJ

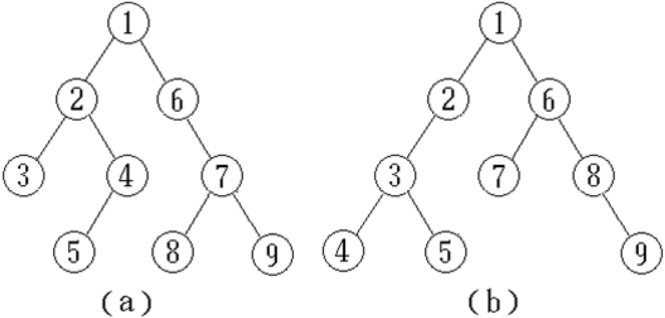




7.4 Binary Tree Reconstruction

Theorem

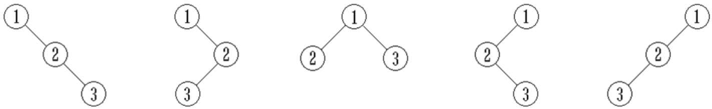
- Given a preorder sequence of BT, we can reconstruct a lot of BTs according to the corresponding inorder sequences.



7.4 Binary Tree Reconstruction

Example

- 1, 2, 3
  Preorder sequence 123
  Inoder traversed sequences are 123, 132, 213, 231, and 321



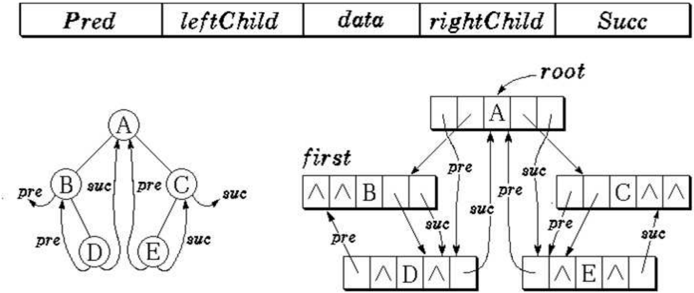
Exercise

- 已知一棵二叉树的后序遍历序列为EICBGAHDF，同时知道该二叉树的中序遍历序列为CEIFGBADH，求前序遍历？
- FCIEDAGBH
- 已知二叉树按中序排列为 BFDAEGC，按前序排列为 ABDFCEG，要求画出该二叉树。
- (思考？)画出和下列已知序列对应的树T，树的先根次序访问序列为RAEFDGBMKHIC，树的后根次序访问序列为EAFGDJBKHIMCR。

Next Section

7.5 Threaded Binary Tree

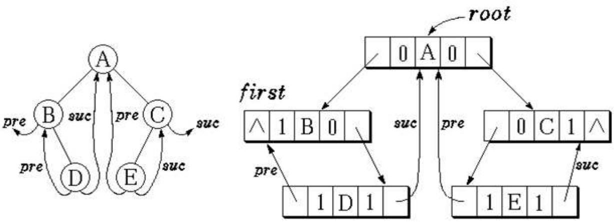
- Thread
  - Concept
  - Implementation



7.5 Threaded Binary Tree

Threaded binary tree and its representation

leftChild	leftThread	data	rightThread	rightChild



if LeftThread = 0, LeftChild points to the left child

LeftThread = 1, LeftChild points to the predecessor

if RightThread = 0, RightChild points to the right child

RightThread = 1, RighChild points to the successor

7.5 Threaded Binary Tree

Threaded binary tree

```
template<class Type> class ThreadNode {
    friend class ThreadTree;
    friend class ThreadInorderIterator;
    private:
        int leftThread, rightThread;
        ThreadNode<Type> *leftChild, *rightChild;
        Type data;
    public:
        ThreadNode(const Type item):data (item),
        leftChild (NULL), rightChild (NULL),
        leftThread (0), rightThread (0) { }
};
```

7.5 Threaded Binary Tree

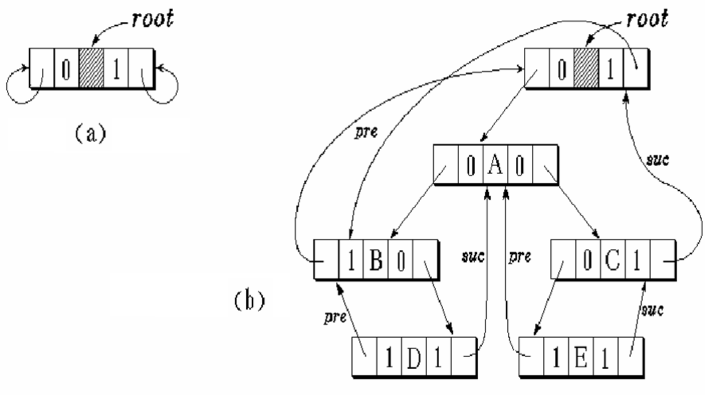
```
template <class Type> class ThreadTree {
    friend class ThreadInorderIterator;
    public:
        ......
    private:
        ThreadNode<Type> *root;
};

template<class Type>class ThreadInorderIterator {
public:
    ThreadInorderIterator(ThreadTree<Type>&Tree)
        :T(Tree){ current = T.root; }
```

7.5 Threaded Binary Tree

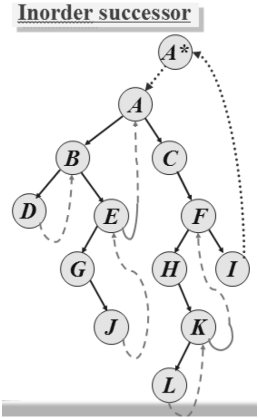
```
    ThreadNode<Type> *First();
    ThreadNode<Type> *Last();
    ThreadNode<Type> *Next();
    ThreadNode<Type> *Prior();
private:
    ThreadTree<Type> &T;
    ThreadNode<Type> *current;
}
```

7.5 Threaded Binary Tree

Inorder threaded binary linked list with dummy node



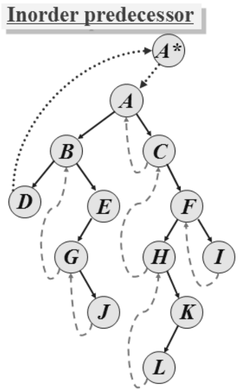
7.5 Threaded Binary Tree




```
if(current->rightThread==1)
    if(current->rightChild!=T.root)
    successor:current->rightChild
    else no successor
else//current->rightThread!=1
    if(current->rightChild!=T.root)
        successor: the first
            inorder visited node
            in the right sub-tree
    else error
```

DBGJEACHLKFI

7.5 Threaded Binary Tree




```
if(current->leftThread==1)
   if(current->leftChild!=T.root)
      predecessor: current->
          leftChild
   else no predecessor
else//current->leftThread==0
   if(current->leftChild!=T.root)
     predecessor: the last inorder
          visited node in the left
          sub-tree
   else error
```

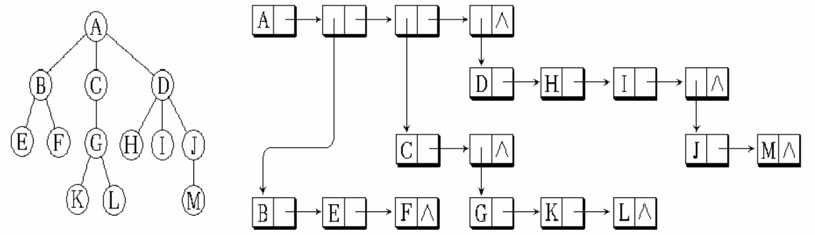
DBGJEACHLKFI

Next Section

Next Subsection

7.6.1 Storage for Tree

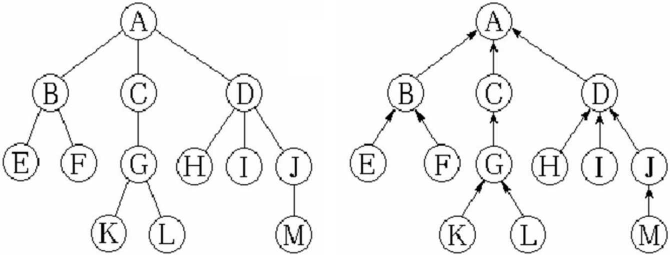
- (1) General list



(a) Tree　　(b) General list representation

7.6.1 Storage for Tree

- (2) Sequential list of parent



NodeList	1	2	3	4	5	6	7	8	9	10	11	12	13
data	A	B	E	F	C	G	K	L	D	H	I	J	M
parent	0	1	2	2	1	5	6	6	1	9	9	9	12

(b) Sequential list of Parent

7.6.1 Storage for Tree

- (3) Linked list of children – multiple linked list

data	$child_1$	$child_2$	$child_3$	....	$child_d$

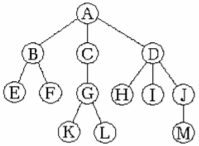
7.6.1 Storage for Tree

- (3) Linked list of children – multiple linked list

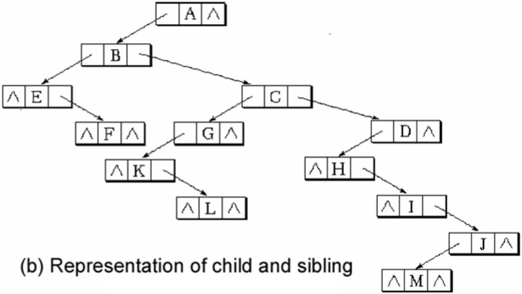
data	$child_1$	$child_2$	$child_3$	....	$child_d$

- (4)Left child–right sibling representation

data	*firstChild*	*nextSibling*



(a) Tree



(b) Representation of child and sibling

Left child–right sibling representation

```
//Tree class
template <class Type> class Tree;
template<class Type> class TreeNode {
friend class<Type> Tree;
private:
    Type data;
    TreeNode<Type> *firstChild, *nextSibling;
    TreeNode(Type value=0,
        TreeNode<Type> *fc=NULL,
        TreeNode<Type> *ns=NULL ):data(value),
        firstChild(fc), nextSibling(ns){}
};
```

Left child–right sibling representation

```
template<class Type> class Tree{
public:
    Tree(){root=current=NULL;}
    //......
private:
    TreeNode<Type> *root, *current;
    void PreOrder(ostream& out,TreeNode<Type> *p );
    int Find(TreeNode<Type> *p,Type target);
    void RemovesubTree(TreeNode<Type> *p);
    int FindParent(TreeNode<Type> *t,TreeNode<Type> *p);
}
```

Left child–right sibling representation

```
//Implementation
template<class Type> void Tree<Type>::BuildRoot(Type rootVal){
    //Create root node
    root=current=new TreeNode<Type>(rootVal);
}

template<class Type> int Tree<Type>::Root(){
    //Set root as current node
    if(root==NULL){
        current=NULL; return 0;
    }
    else{
        current=root; return 1;
    }
}
```

Left child–right sibling representation

```
//Find out the parent of current node and set it as the
    new current node
template<class Type>int Tree<Type>::Parent(){
    TreeNode<Type> *p=current, *t;
    if (current==NULL||current==root){
        current=NULL; return 0;
    }
    t=root;
    int k=FindParent(t,p);
    return k;
}
```

Left child–right sibling representation

```
template<class Type>int Tree<Type>::
FindParent(TreeNode<Type> *t, TreeNode<Type> *p){
    //find out the parent of  p in T
    TreeNode<Type> *q=t->firstChild;
    while(q!=NULL&&q!=p){
       //find out the node along sibling link
       if((int i=FindParent(*q,*p))!=0)return i;
       q=q->nextSibling;
    }
    if(q!=NULL&&q==p){
        current=t; return 1;
    }
    else return 0;
}
```

7.6.1 Storage for Tree

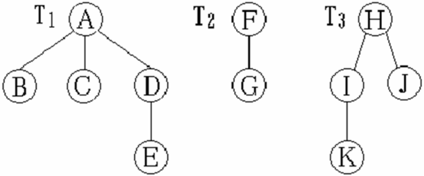
```
template<class Type> int Tree<Type>::FirstChild()
{
    if(current!=NULL&&current->firstChild!=NULL){
        current=current->firstChild; return 1;
    }
    current=NULL; return 0;
}
template<class Type>int Tree<Type>::NextSibling()
{
    if (current!=NULL&&current->nextSibling!=NULL){
        current=current->nextSibling; return 1;
    }
    current=NULL; return 0;
}
```

7.6.1 Storage for Tree

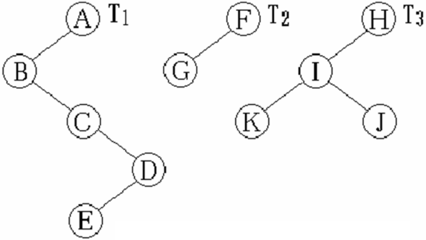
```
template<class Type>int Tree<Type>::Find(Type target){
    if(IsEmpty()) return 0;
    return Find(root,target);
}

template<class Type> int Tree<Type>::
Find(TreeNode<Type> *p,Type target){
    int result=0;
    if(p->data==target){result=1; current=p;}
    else{
        TreeNode<Type> *q=p->firstChild;
        while(q!=NULL&&!(result=Find(q,target)))
            q=q->nextSibling;
    }
    return result;
}
```

Next Subsection

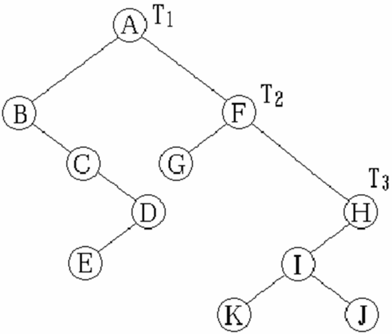
7.6.2 Conversion between Forest and BT



(a) Forest (including 3 trees)



(b) Binary tree representation of each tree



(c) BT representation of forest

7.6.2 Conversion between Forest and BT

- Forest $\Longrightarrow$ BT
  - If $F == NULL(n = 0)$, corresponding binary tree BT is $NULL$
  - If $F != NULL$, then
    - **The root of corresponding BT is the root of T_1, which is the first tree in the forest**
    - **The left subtree of BT is converted from the subtrees of T_1,** $B(T_1 1, T_1 2, \ldots, T_1 m)$
    - **The right subtree of BT is converted from the remainders of the forest,** $B(T_2, T_3, \ldots, T_n)$

7.6.2 Conversion between Forest and BT

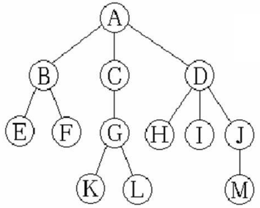
BT$\Longrightarrow$ Forest

- If$BT == NULL(n = 0)$, corresponding Forest F is $NULL$
- If$BT != NULL$, then
  - **The root of the first tree T_1,in converted forest F is the root of BT;**
  - **The subtrees of T_1, $T_1, \{T_1 1, T_1 2, \ldots, T_1 m\}$, are converted from the left subtree of BT;**
  - **The other trees in corresponding forest, $\{T_2, T_3, \ldots, T_n\}$,are converted from the right subtree of BT;**

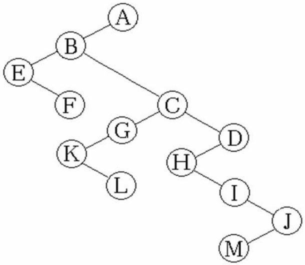
7.6.2 Conversion between Forest and BT

Traversal of Tree

- Depth-first traversal
  - Preorder (root first visited) traversal: ABEFCGKLDHIJM
  - Postorder (root last visited) traversal: EFBKLGCHIMJDA
  - Tree preorder = BT preorder Tree postorder = BT inorder



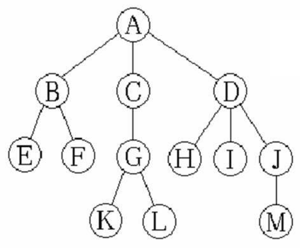
(a) Tree



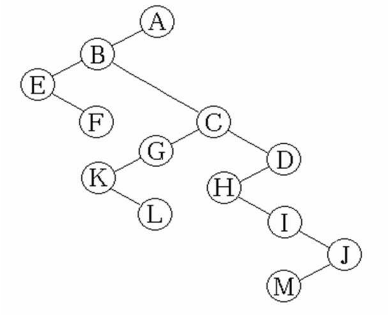
7.6.2 Conversion between Forest and BT

Traversal of Tree

- Breath-first traversal



(a) Tree



7.6.2 Conversion between Forest and BT

(1) Preorder traversal (Recursive algorithm)

```
template <class Type>
void Tree<Type>::PreOrder(){
     if(!IsEmpty()){
        visit();
        TreeNode<Type> *p=current;
        int i=FirstChild();
        while(i){
            PreOrder(); i=NextSibling();
        }
        current=p;
     }
}
```

7.6.2 Conversion between Forest and BT

(2) Postorder traversal (Recursive algorithm)

```
template<class Type>
void Tree<Type>::PostOrder(){
    if(!IsEmpty()){
        TreeNode<Type> *p=current;
        int i=FirstChild();
        while(i){
             PostOrder(); i=NextSibling();
        }
        current=p;
        visit();
    }
}
```

7.6.2 Conversion between Forest and BT

(3) Preorder traversal (Iterative algorithm)

```
template <class Type>
void Tree<Type>::NorecPreOrder(){
    Stack<TreeNode<Type>*> st(DefaultSize);
    TreeNode<Type> *p=current;
    do{
        while(!IsEmpty()){
            visit(); st.Push(current); FirstChild();
        }
        while(IsEmpty()&&!st.IsEmpty()){
            current=st.Pop(); NextSibling();
        }
    } while(!IsEmpty());
    current=p;
}
```

7.6.2 Conversion between Forest and BT

(4) Postorder traversal (Iterative algorithm)

```
template <class Type>
void Tree<Type>::PostOrder1(){
    Stack<TreeNode<Type>*> st;
    TreeNode<Type> *p=current;
    do{
        while(!IsEmpty()){
             st.Push(current); FirstChild();}
        while(IsEmpty()&&!st.IsEmpty()){
            current=st.Pop(); visit(); NextSibling();}
    } while(!IsEmpty());
    current = p;
}
```

7.6.2 Conversion between Forest and BT

//Breadth-first traversal

```
template<class Type> void Tree<Type>::LevelOrder(){
    Queue<TreeNode<Type>*>Qu(DefaultSize);
    TreeNode<Type> *p
    if(!IsEmpty()){
       p=current; Qu.EnQueue(current);
       while(!Qu.IsEmpty()){
            current=Qu.DeQueue(); visit();
            FirstChild();
            while(!IsEmpty()){
                 Qu.EnQueue(current); NextSibling();}
       }
       current=p;}
}
```

Postorder traversal(Iterative algorithm)

```
template <class Type>
void Tree<Type>::PostOrder1(){
    Stack<TreeNode<Type>*> st;
    TreeNode<Type>*p=current;
    do {
        while(!IsEmpty()){
             st.Push(current); FirstChild();
        }
        while(IsEmpty()&&!st.IsEmpty()){
  current=st.Pop(); visit(); NextSibling();
        }
    } while(!IsEmpty());
    current=p;
}
```

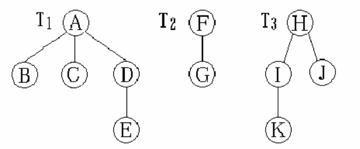
7.6.2 Conversion between Forest and BT

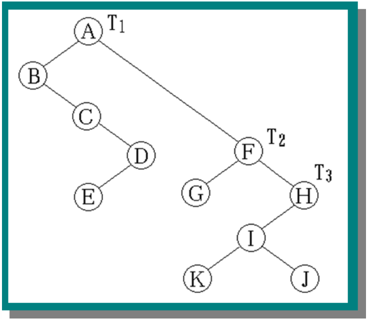
```
//Breadth-first traversal
template<class Type> void Tree<Type>::LevelOrder() {
    Queue<TreeNode<Type>*> Qu(DefaultSize);
    TreeNode<Type> *p
    if(!IsEmpty()){
       p=current; Qu.EnQueue(current);
       while(!Qu.IsEmpty()){
            current=Qu.DeQueue();
            visit(); FirstChild();
            while(!IsEmpty()){
                 Qu.EnQueue(current); NextSibling();}
       }
       current=p;}
}
```

Next Subsection

7.6.3 Traversal of Forest

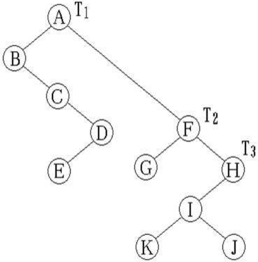
- Three order traversals based on BT
  - Preorder traversal
  - Inorder traversal
- Level order traversal





7.6.3 Traversal of Forest

- (1) Preorder
  - If F==NULL, return; else
  - Visit the root of the first tree of F;
  - Preorder traverse the specific forest of sub-trees of the first tree;
  - Preorder traverse the specific forest consisting of the remainder trees;

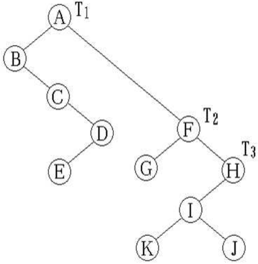


A B C D E F G H I K J

7.6.3 Traversal of Forest

Traversal of Forest

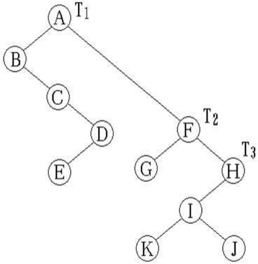
- (2) Inorder
  - If F==NULL, return; else
  - Inorder traverse the specific forest of sub-trees of the first tree;
  - Visit the root of the first tree of F;
  - Inorder traverse the specific forest of the remainder trees;



B C E D A G F K I J H

7.6.3 Traversal of Forest

- (3) Postorder
  - If F==NULL, return; else
  - Postorder traverse the specific forest of sub-trees of the first tree;
  - Postorder traverse the specific forest of the remainders trees;
  - Visit the root of the first tree of F;

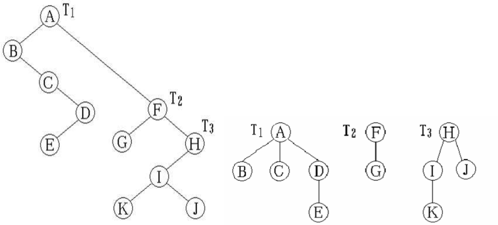


E D C B G K J I H F A

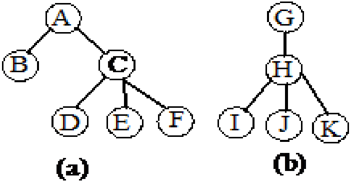
7.6.3 Traversal of Forest

- (4) Levelorder
  - If F==NULL, return; else
  - Visit the root of each tree in the forest one by one;
  - Visit the children of the root of each trees according to the order of the roots;
  - Visit the children of these children of these sub-roots;

A F H B C D G I J E K

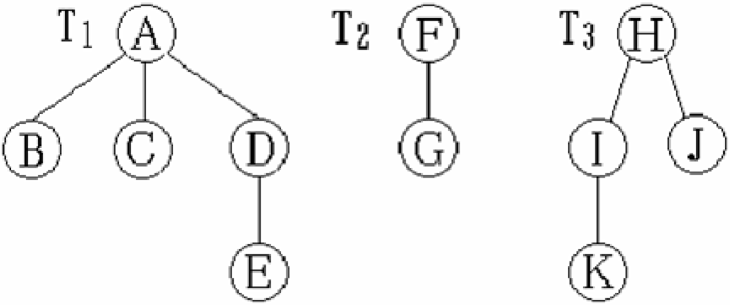


Exercise



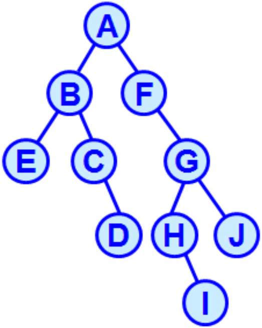
- Tree traversal.
- Please covert forest to binary tree.

Exercise



- Tree traversal.
- Please covert forest to binary tree.

Exercise



- Please covert binary tree to forest.
- Tree traversal.

Next Section

7.7 Huffman Tree and Coding

- Background

Score	0-59	60-69	70-79	80-89	90-100
Grade	E	D	C	B	A
Percent	0.05	0.15	0.40	0.30	0.10

7.7 Huffman Tree and Coding

- Background

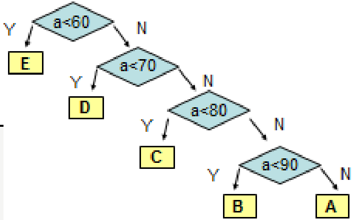
Score	0-59	60-69	70-79	80-89	90-100
Grade	E	D	C	B	A
Percent	0.05	0.15	0.40	0.30	0.10

```
if (a<60) b‘=’E;
    else if (a<70) b‘=’D;
        else if (a<80) b‘=’C;
            else if (a<90) b‘=’B;
                else b‘=’A;
```

7.7 Huffman Tree and Coding

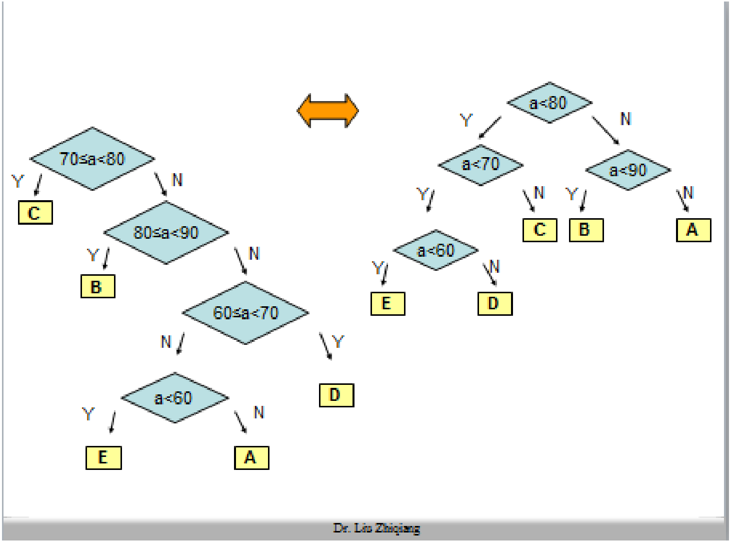
- Background

Score	0-59	60-69	70-79	80-89	90-100
Grade	E	D	C	B	A
Percent	0.05	0.15	0.40	0.30	0.10




```
if (a<60) b‘=’E;
    else if (a<70) b‘=’D;
        else if (a<80) b‘=’C;
            else if (a<90) b‘=’B;
                else b‘=’A;
```

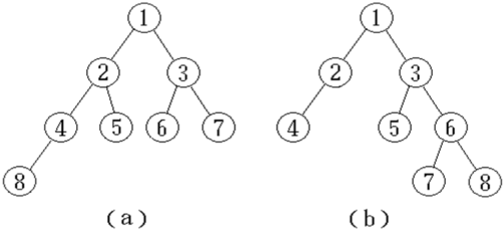
7.7 Huffman Tree and Coding



7.7 Huffman Tree and Coding

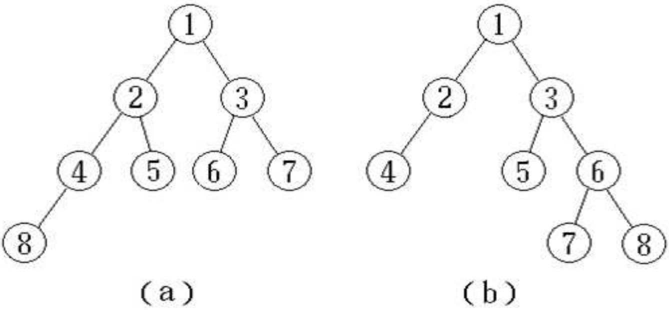
- Path length of node
  - The number of the branches between two specific nodes
    $PL(n_1, n_2) =$ number of branches
    $PLn(i) =$ number of branches from root to node i
- Path length of tree
  - The summarization of path lengths from every nodes to the root

$$PL = \sum_{i=0}^{n-1} PLn(i)$$



7.7 Huffman Tree and Coding

Examples of binary trees with different path length



PL=0+1+1+2+2+2+2+3=13

PL=0+1+1+2+2+2+3+3=14

7.7 Huffman Tree and Coding

- In general

$$PL \geq \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor = 0+1+1+2+2+2+2+3+3+\cdots$$

- Minimum

$$PL = \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor$$

7.7 Huffman Tree and Coding

- In general

$$PL \geq \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor = 0+1+1+2+2+2+2+3+3+\cdots$$

- Minimum

$$PL = \sum_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor$$

相同节点数的二叉树，完全二叉树的路径最小（只有完全二叉树吗？唯一吗？）

7.7 Huffman Tree and Coding

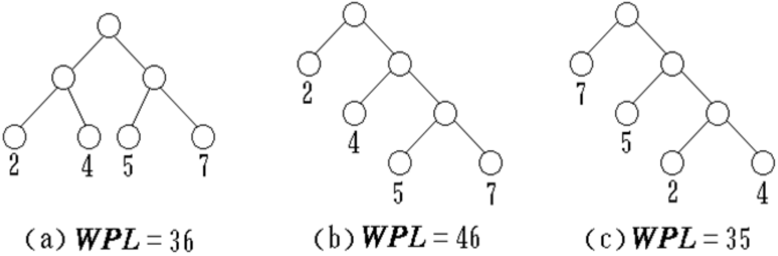
Weighted Path Length

- Weighted Path Length – WPL
  - Weighted leaves
  - WPL
    - The summarization of path length from root to the leaves multiply the weights

$$WPL = \sum_{i=0}^{n-1} w_i * l_i$$

7.7 Huffman Tree and Coding

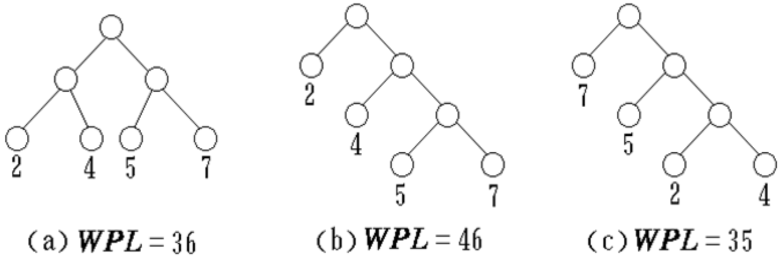
Examples



- 若T是一颗有n个叶节点的二叉树，并将权值分别赋给n个叶节点，则称T为带权值的扩充二叉树。带权值的叫做外节点，不带权值的叫做内节点

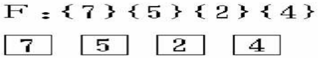
7.7 Huffman Tree and Coding

Examples

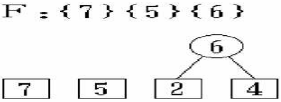


- 若T是一颗有n个叶节点的二叉树，并将权值分别赋给n个叶节点，则称T为带权值的扩充二叉树。带权值的叫做外节点，不带权值的叫做内节点
- Huffman tree
- 带权路径长度最小的二叉树应该是权值越大的外节点离根节点越近的二叉树，这种二叉树叫做 霍夫曼树。

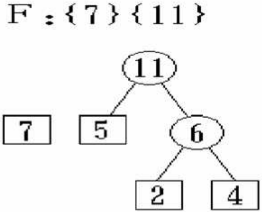
7.7 Huffman Tree and Coding



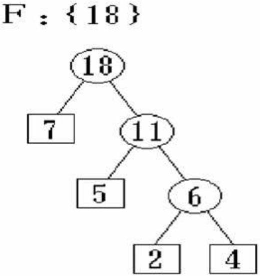
(a) Initilization



(b) Merge {2} and {4}



(c) Merge {5} and {6}



(d) Merge {7} and {11}

Huffman algorithm

- 设给定的一组权值为W1，W2，W3，......Wn，据此生成森林F=T1，T
  中的每棵二叉树只有一个带权为W1的根节点（i=1，2，......n）。
- 在F中选取两棵根节点的权值最小和次小的二叉树作为左右构造一棵新的二叉树，新二叉树根节点的权值为其左、右子树根节点的权值之和。
- 在F中删除这两棵最小和次小的二叉树，同时将新生成的二叉树并入森林中。
- 重复（2）（3）过程直到F中只有一棵二叉树为止。

Exercise

有七个带权结点,其权值分别为3, 7, 8, 2, 6, 10, 14, 试以它们为叶结点构造一棵哈夫曼树（请按照每个结点的左子树根结点的权小于等于右子树根结点的权的次序构造）, 并计算出带权路径长度WPL及该树的节点总数。

Huffman algorithm, Concepts of coding

- Encode：编码是信息从一种形式或格式转换为另一种形式的过程。
- Decode：是编码的逆过程。
- Code book
- 霍夫曼编码是最小冗余编码问题，是数据压缩学的基础。
- 用预先规定的方法将文字、数字或其他对象编成数码，或将信息、数据转换成规定的电脉冲信号。

Examples of coding

CAST CAST SAT AT A TASA

{ C, A, S, T }

C:00 A:01 S:10 T:11

00011011 00011011 100111 0111 01 11011001

(2+7+4+5) * 2 = 36

{ 2/18, 7/18, 4/18, 5/18 }

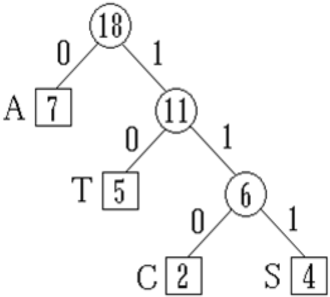
{ 2, 7, 4, 5 }

CAST CAST SAT AT A TASA

A : 0 T : 10 C : 110 S : 111

110011110 110011110 111010

7*1+5*2+(2+4)*3 = 35



7.7 Huffman Tree and Coding

Example

- A communication system has eight symbols

  c1, c2, c3, c4, c5, c6, c7, c8

  and the probability of each symbol is

  5, 25, 3, 6, 10, 11, 36, 4

  Try to design a coding method

7.7 Huffman Tree and Coding

Huffman Tree establishment

Node	Weight	Parent	lchild	rchild
1 C1	5	0	0	0
2 C2	25	0	0	0
3 C3	3	0	0	0
4 C4	6	0	0	0
5 C5	10	0	0	0
6 C6	11	0	0	0
7 C7	36	0	0	0
8 C8	4	0	0	0
9	-	0	0	0
10	-	0	0	0
11	-	0	0	0
12	-	0	0	0
13	-	0	0	0
14	-	0	0	0
15	-	0	0	0

Node	Weight	Parent	lchild	rchild
1 C1	5	10	0	0
2 C2	25	14	0	0
3 C3	3	9	0	0
4 C4	6	10	0	0
5 C5	10	11	0	0
6 C6	11	12	0	0
7 C7	36	14	0	0
8 C8	4	9	0	0
9	7	11	3	8
10	11	12	1	4
11	17	13	9	5
12	22	13	10	6
13	39	15	11	12
14	61	15	2	7
15	100	0	13	14

7.7 Huffman Tree and Coding

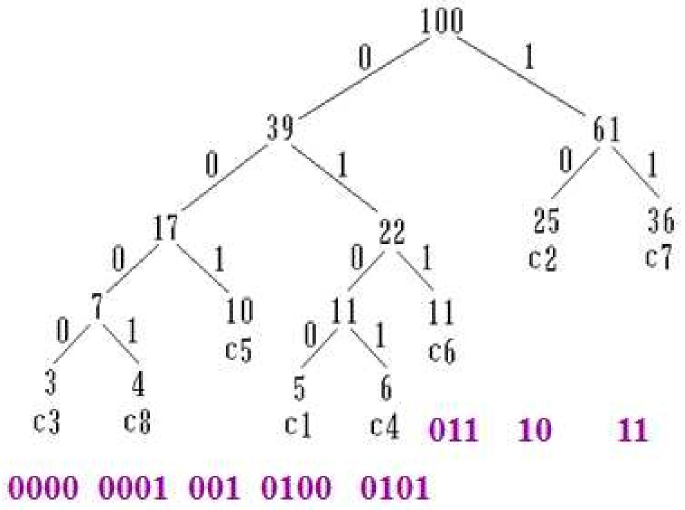
Huffman Tree

Huffman Coding

Node	Weight	Parent	lchild	rchild
1 C1	5	10	0	0
2 C2	25	14	0	0
3 C3	3	9	0	0
4 C4	6	10	0	0
5 C5	10	11	0	0
6 C6	11	12	0	0
7 C7	36	14	0	0
8 C8	4	9	0	0
9	7	11	3	8
10	11	12	1	4
11	17	13	9	5
12	22	13	10	6
13	39	15	11	12
14	61	15	2	7
15	100	0	13	14

C1	0	1	0	0
C2	1	0		
C3	0	0	0	0
C4	0	1	0	1
C5	0	0	1	
C6	0	1	1	
C7	1	1		
C8	0	0	0	1

7.7 Huffman Tree and Coding



7.7 Huffman Tree and Coding

Huffman coding:

c1	c2	c3	c4	c5	c6	c7	c8
0100	10	0000	0101	001	011	11	0001

Total length of these symbols is

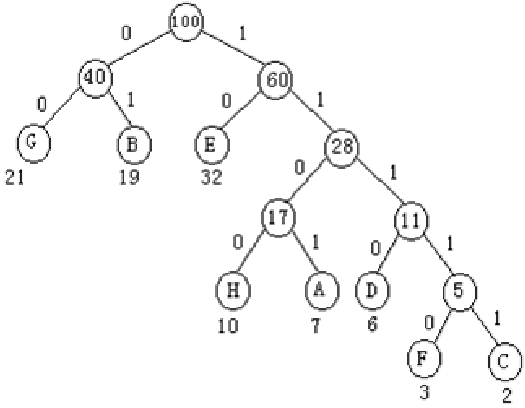
$$4 * 5 + 2 * 25 + 4 * 3 + 4 * 6 + 3 * 10 + 3 * 11 + 2 * 36 + 4 * 4 = 257$$

7.7 Huffman Tree and Coding

- 假设用于通信的电文仅由8个字母组成，字母在电文中出现的频率分别为 0.07，0.19，0.02，0.06，0.32，0.03，0.21，0.1。试为这8个字母设计Huffman Code。使用0 7的二进制表示形式是另一种编码方案。对于上述实例，比较两种方案的优缺点。

7.7 Huffman Tree and Coding

- 假设用于通信的电文仅由8个字母组成，字母在电文中出现的频率分别为 0.07，0.19，0.02，0.06，0.32，0.03，0.21，0.1。试为这8个字母设计Huffman Code。使用0 7的二进制表示形式是另一种编码方案。对于上述实例，比较两种方案的优缺点。



Summary

1. Tree Definition and Concepts
2. Binary Tree
   - Binary Tree Definition
   - Characteristics of Binary Tree
   - Binary tree represented as array and linked list
3. Traversal of Binary Tree
   - Preorder, Inorder, Postorder traversal
   - Non recursive algorithm for traversal
4. Binary Tree Reconstruction
5. Threaded Binary Tree
6. Tree and Forest
   - Storage for Tree
   - Conversion between Forest and BT
   - Traversal of Forest
7. Huffman Tree and Coding