Data Structures and Algorithms

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

tianyi.zang@gmail.com

Tree & Binary Trees

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

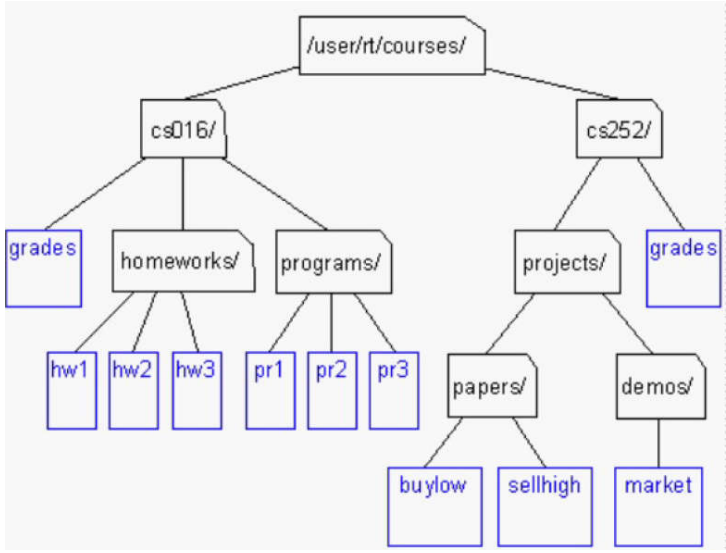
tianyi.zang@gmail.com

Outline

- Tree Definitions and Terminology
- Tree ADT
- Binary Tree Definitions
- Binary Tree Properties
- Binary Tree Implementation
- Binary Tree Traverse Algorithms
- Binary Tree Applications

Tree Example

- Representing File Structures:
  - Consider the Unix file system
  - Hierarchically arranged so that each file (including directories) belongs to some directory (except the / file which is the root)
  - Each directory must be able to tell what files are in it
- Other Trees
  - Family Trees
  - Organization Structure Charts
  - Program Design
  - Structure of a chapter in a book

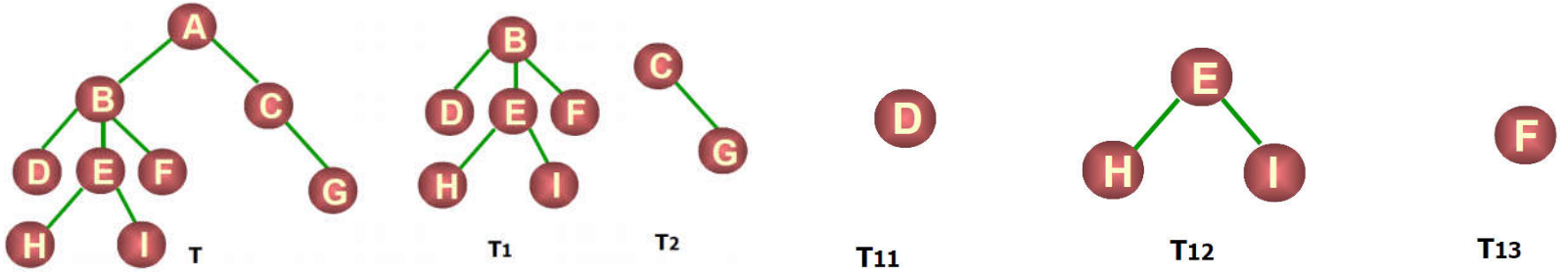


Definition of Tree

- A tree T is a finite set of one or more elements called nodes such that:
  - There is one designated node R, called the root of T.
  - If the set (T- {R}) is not empty, these nodes are partitioned into n > 0 disjoint subsets T_0, T_1, ..., T_{n-1}, each of which is a tree, and whose roots R_1, R_2, ..., R_n, respectively, are children of R
  - The subsets T_i ($0 < i < n$) are said to be subtrees of T.

Definition of Tree

- root: no predecessor
- children: no successor
- others: only one predecessor and one or more successor
- **Definition is recursive**

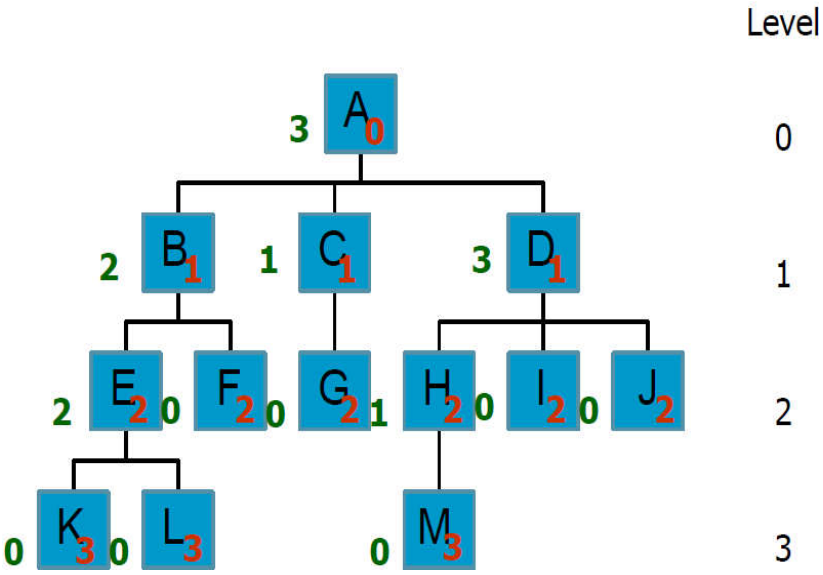


Terminology

- Root of subtree only have a direct previous, but can have 0 or more direct successor.
- There is an *edge* from a node to each of its children, and a node is said to be the *parent* of its children.
- If $n_1, n_2, ..., n_k$ is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a *path* from n_1 to n_k. The *length* of the path is k -1.
- The *depth* of a node M in the tree is the length of the path from the root of the tree to M.
- The *height* of a tree is one more than the depth of the deepest node in the tree.

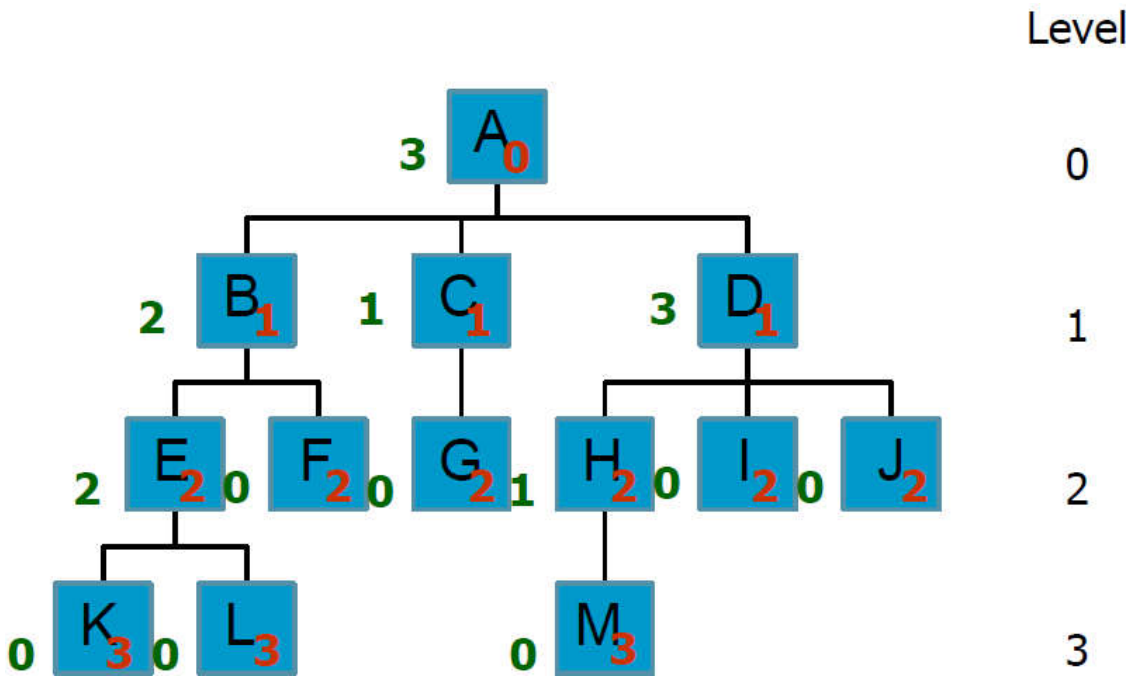
Terminology

- The *degree* of a node is the number of subtrees of the Node
  - The degree of A is 3, the degree of C is 1.
- The *degree of a tree* is the maximum degree of the nodes in the tree.
- The node with degree 0 is a *leaf or terminal* node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The *ancestors* of a node are all the nodes along the path from the root to the node.
- A *forest* is a collection of one or more trees



Terminology

- node (13)
- degree of a node
- root
- leaf (terminal)
- internal node
- parent
- children
- sibling
- degree of a tree (3)
- ancestor
- descendant
- level of a node
- height of a tree (4)
- forest



Terminology: comparision

Linear structure

First element

No precedeed

Last element

No succedeed

Other element

One precedeed and succedeed

One to one

Tree structure

Root node (only one)

No parents

Leave node (many)

No childred

Other node

One parent and multi children

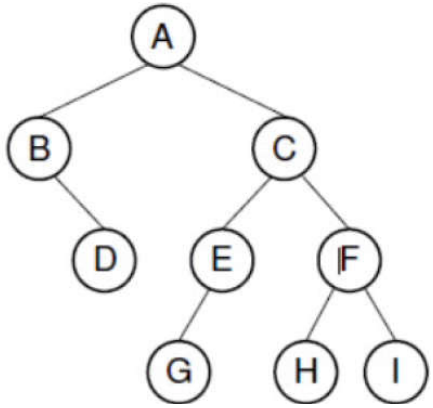
one to many

General Tree ADT

```
// General tree ADT
template <typename E> class GenTree {
    // Send all nodes to free store
    void clear();
    // Return the root of the tree
    GTNode<E>* root();
    // Combine two subtrees
    void newroot(E&, GTNode<E>*, GTNode<E>*);
    // Print a tree
    void print();
};
```

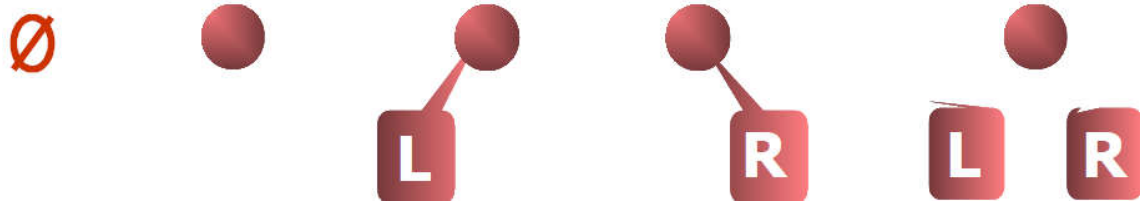
Definition of Binary Tree

- *A binary tree* is made up of a finite set of nodes.
- This set either is empty or
- Consists of a node called the root together with two binary trees, called *the left subtree and/or right subtree*, which are disjoint from each other.
- A binary tree example：

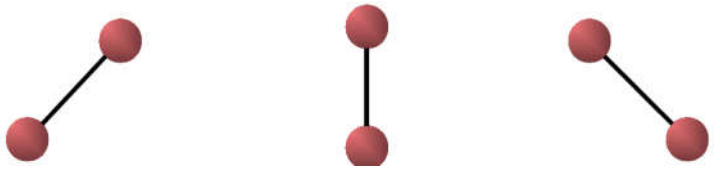


Shapes of Binary Tree

- Binary Tree has five different shapes



- left and right are important for binary trees
- The following three trees are the same or not?



Properties of Binary Tree

(1) **The maximum number of nodes on i^{th} level of a binary tree is 2^{i-1}, $i \geq 1$.**

- Proof: The proof is by induction on i.

  Induction base: The root is the only node on first level.

  The maximum number of nodes on 1^{th} level is $2^{i-1} = 2^0 = 1$.

  Induction hypothesis: For all j, $1 \leq j < i$, the maximum number of nodes on j^{th} level is 2^{j-1}.

  Induction step: Since each node has a maximum degree of 2, the maximum number of nodes on i^{th} level is two times the maximum number of nodes on $(i-1)^{th}$ level or $2 * 2^{(i-1)-1} = 2^{i-1}$

Properties of Binary Tree

(2) The maximum number of nodes in a binary tree of height k is 2^k-1, $k \geq 1$.

- Proof: The maximum number of nodes in a binary tree of height k is:

$$\sum_{i=1}^{k} (\text{max number of nodes on ith level})$$

$$= \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Properties of Binary Tree

(3) **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:

– Let n and B denote the total number of nodes & branches in T.

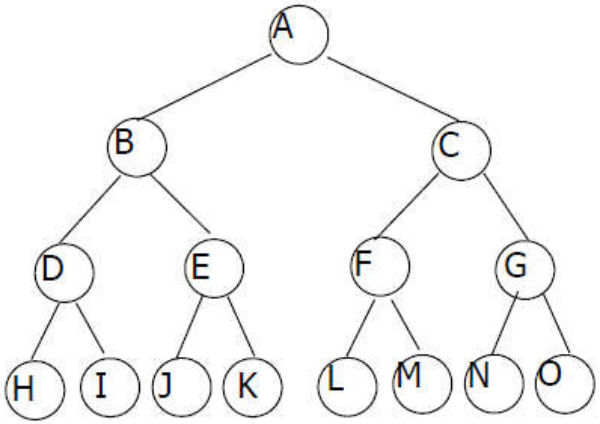
– Let n_0, n_1, n_2 represent the number of nodes with no children, single child, and two children respectively.

– $n= n_0+n_1+n_2$ $\quad B+1=n,\ B=n_1+2n_2 \Longrightarrow n_1+2n_2+1= n$

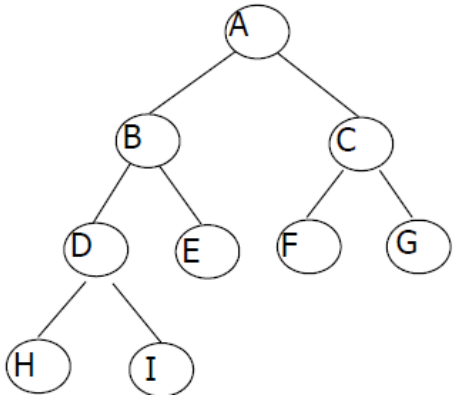
$n_1+2n_2+1= n_0+n_1+n_2 \Longrightarrow n_0=n_2+1$

Full BT & Complete BT

- Each node ***in a full binary tree*** is either (1) an internal node with exactly two non-empty children or (2) a leaf.
- In the ***complete binary tree*** of height d, all levels except possibly the bottom level are completely full. The bottom level has its nodes filled in from the left side.



Full binary tree



Complete binary tree

Properties of Complete Binary Tree

- (4)The height k of the complete binary tree with n nodes is: $k=\lceil \log_2(n+1) \rceil$ or $\lfloor \log_2 n \rfloor + 1$
- Proof:

  By (2) and definition of complete BT, there is:

  $2^{k-1}-1 < n \leq 2^k-1$

  $2^{k-1} \leq n < 2^k$

  $k-1 \leq \log_2 n < k$

  k is integer, so $k = \lfloor \log_2 n \rfloor + 1$

Properties of Binary Tree: sequential storage structure

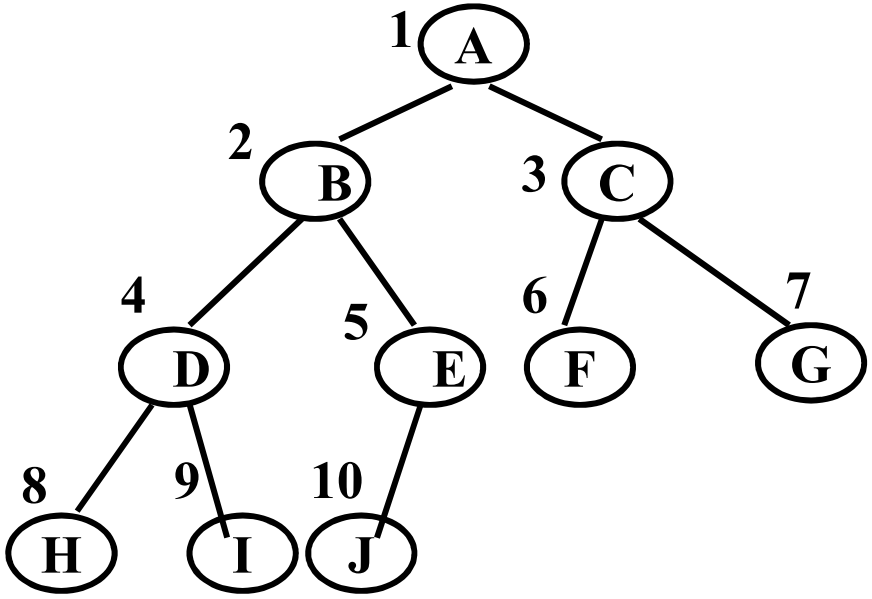
(5) If a complete binary tree with n nodes (height $=\log_2 n+1$) is represented sequentially, then for any node with index i, $1 \le i \le n$, we have:

- parent(i) is at $\lfloor i/2 \rfloor$ if i!=1. If i=1, i is at the root and has no parent.
- leftChild(i) is at 2i if $2i \le n$. If 2i>n, then i has no left child.
- rightChild(i) is at 2i+1 if $2i+1 \le n$. If 2i +1 >n, then i has no right child.

Proof:

Binary Tree: sequential storage structure

- Example:



	A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9	10

Full Binary Tree Theorem

- **Theorem 1:** The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.
- **Proof:**

  By definition, all internal nodes of full binary tree are nodes of degree 2. And $n_0=n_2+1$

Full Binary Tree Theorem

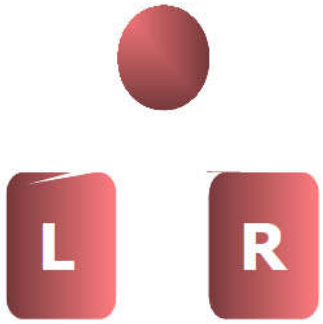
- **Theorem 2:** The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.
- **Proof:**
  - Let n denotes the number of nodes in binary tree T.
  - By definition, every node in binary tree T has two children. So, there are 2n children in T (including empty nodes).
  - Every node except the root node has one parent, for a total of n-1 nodes with parents. In other words, there are n-1 non-empty children.
  - The number of empty children is :2n-(n-1)=n+1

Binary Tree Traversals

- Preorder
- Inorder
- Postorder
- Level Order

Binary Tree Traversals

- Traversal: Each node is visited once and can only be visited once.
- Traversal is easy to linear structure. But to nonlinear structure, it is needed to **linearize** nonlinear structure according to certain rules
- The binary tree consists of three basic units: root, left subtree and right subtree.



Binary Tree Traversals

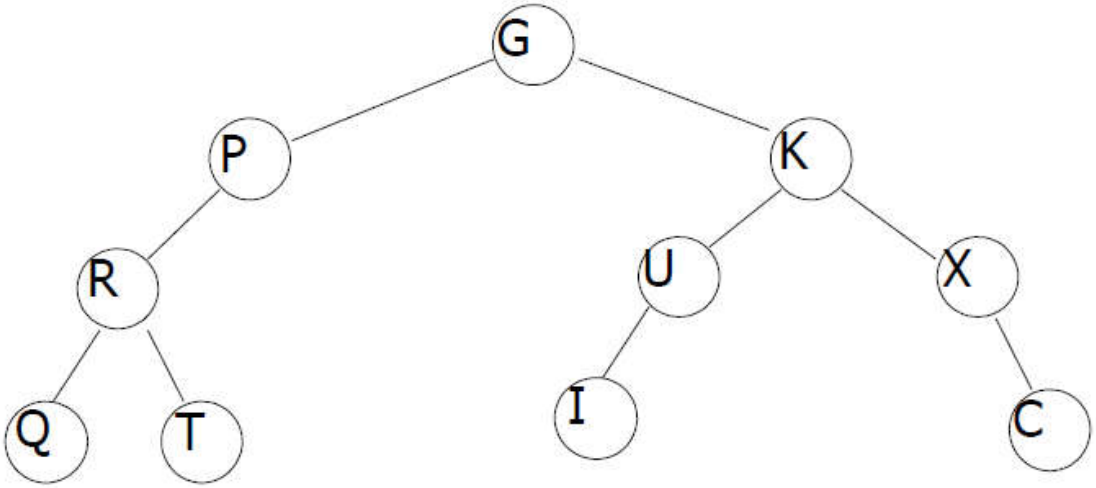
- Let L, D, and R stand for moving left, visiting the root node, and moving right.
- There are six possible combinations of traversal
  - DLR, LDR, LRD, DRL, RDL, RLD
- Adopt convention that we traverse left before right, only 3 traversals remain
  - DLR, LDR, LRD
  - preorder, inorder , postorder

Binary Tree Traversals

- Suppose that we need to visit all of the nodes in a binary tree. In what order can this be done? The most common:
  - Preorder
  - Inorder
  - Postorder
  - Level Order
- Level order is breadth-first traversal, the other three are depth-first traversal.

Preorder Traversal

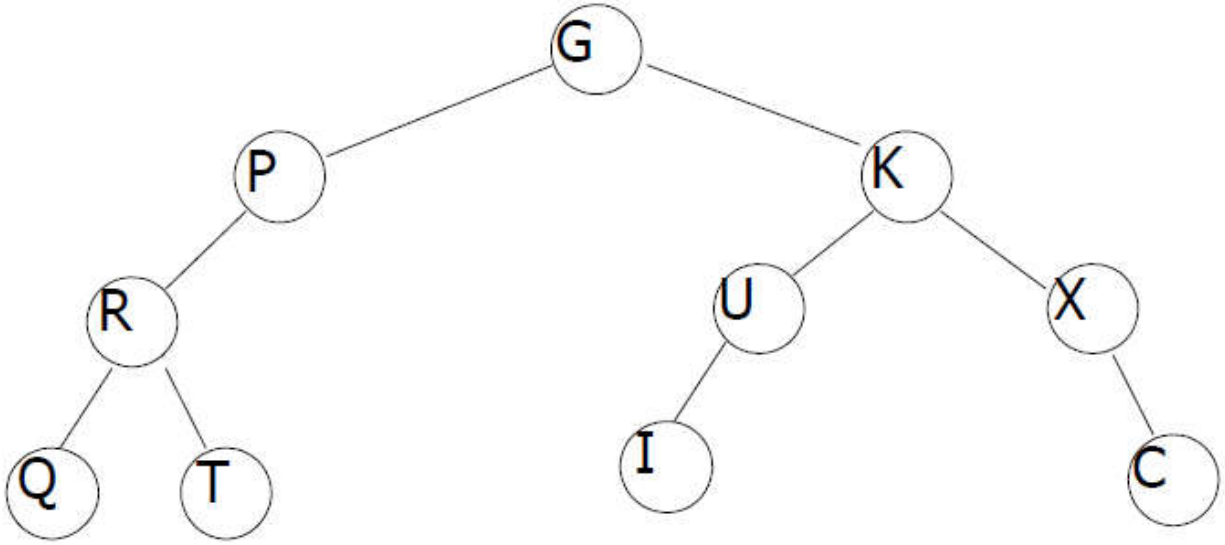
- Visit the root node.
- Traverse the left subtree.
- Traverse the right subtree.



G, P, R, Q, T, K, U, I, X, C

Inorder Traversal

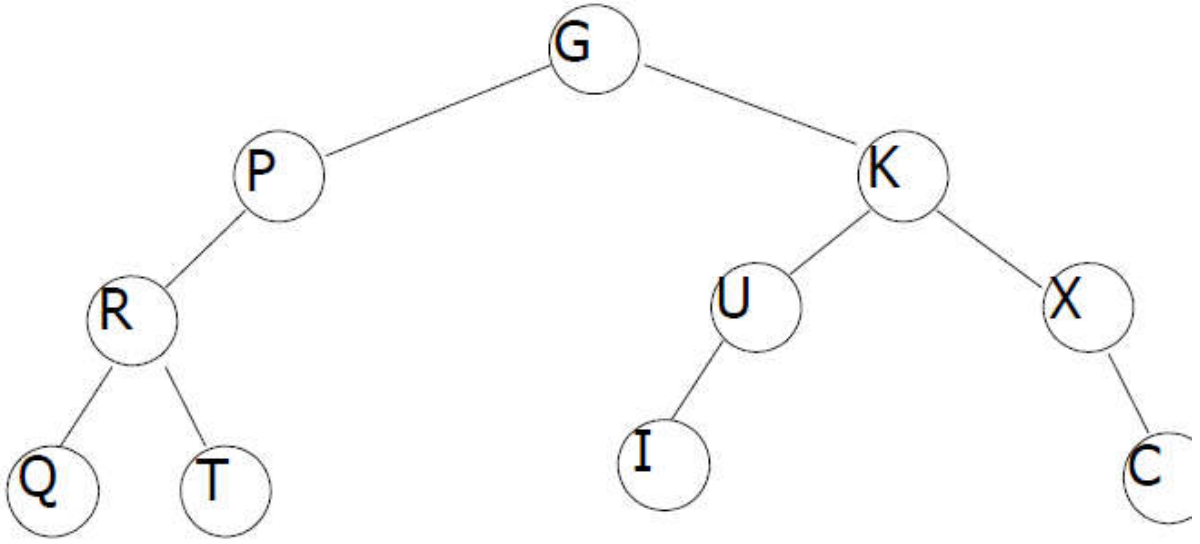
- Traverse the left subtree.
- Visit the root node.
- Traverse the right subtree.



Q, R, T, P, G, I, U, K, X, C

Postorder Traversal

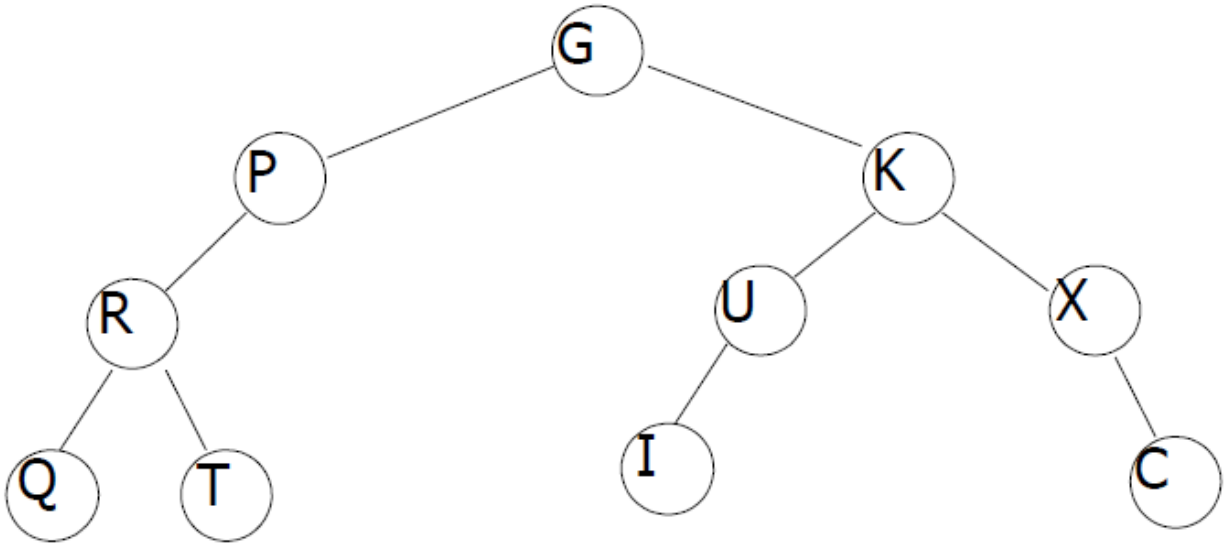
- Traverse the left subtree.
- Traverse the right subtree.
- Visit the root node.



Q, T, R, P, I, U, C, X, K, G

Level Order Traversal

- Visit the nodes from level to level,
- beginning with the root node.



G, P, K, R, U, X, Q, T, I, C

Basic operations of BT

① Empty (BT) : //construct an empty BT

② IsEmpty (BT) : // return TRUE if empty, otherwise FALSE

③ CreateBT (V, LT , RT) : // create a new BT, the value of the data field of the root node is V, and its left and right subtree are LT and RT respectivly

④ Lchild (BT): Return the node's left child, empty if no left child.

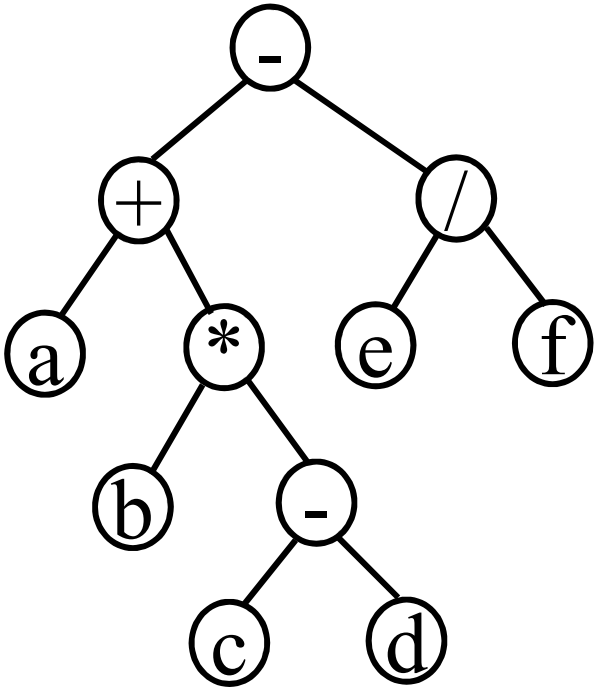
⑤ Rchild (BT) : Return the node's right child , empty if no right child.

⑥ Data (BT) : Return the node's value

Using basic operations: Preorder traverse

```
void  PreOrder (BTREE BT )
{
    if ( ! IsEmpty ( BT ) ) // do nothing
    {  // Perform desired action
         visit ( Data ( BT ) ) ;
        PreOrder ( Lchild ( BT ) ) ;
        PreOrder ( Rchild ( BT ) ) ;
     }
}
```

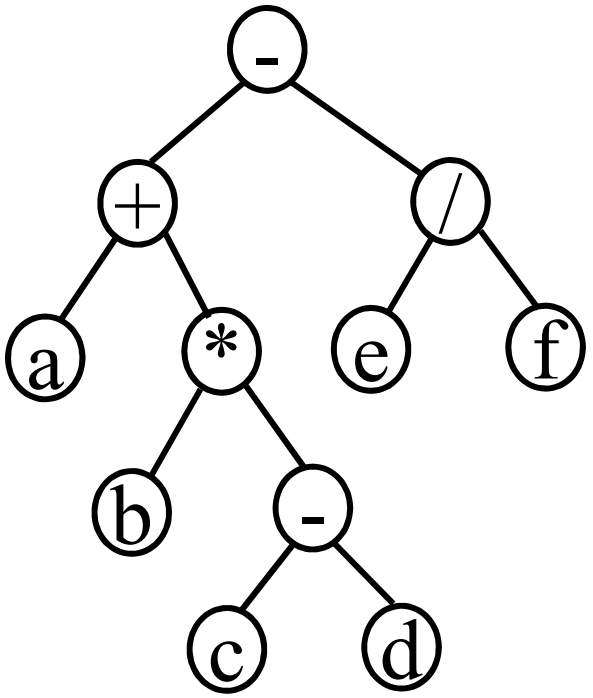
Preorder sequence: - + a * b – c d / e f



a+b*(c-d)-e/f

Using basic operations: Inorder traverse

```
void  InOrder (BTREE BT )
{
    if ( ! IsEmpty ( BT ) ) // do nothing
    {    InOrder ( Lchild ( BT ) ) ;
          // Perform desired action
         visit ( Data ( BT ) ) ;
         InOrder ( Rchild ( BT ) ) ;
     }
}
```

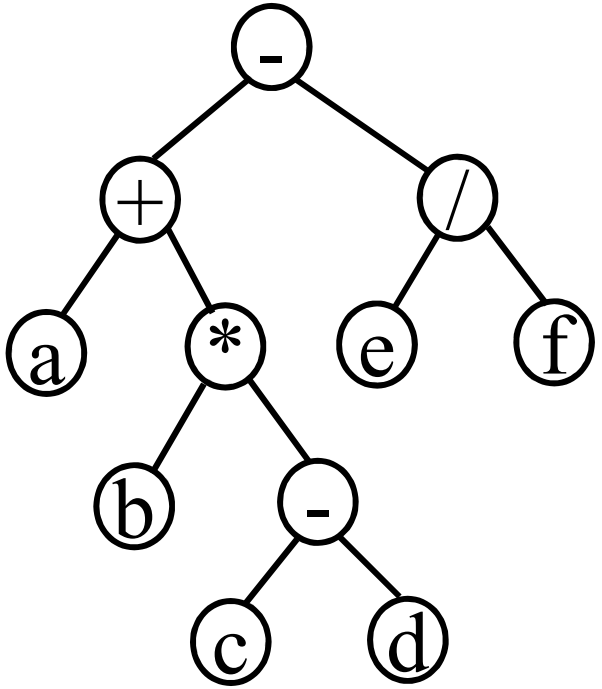


a+b*(c-d)-e/f

Inorder sequence: a + b * c – d – e / f

Using basic operation: Postorder traverse

```
void  PostOrder (BTREE BT )
{
    if ( ! IsEmpty ( BT ) ) //do nothing
   {      PostOrder ( Lchild ( BT ) ) ;
          PostOrder ( Rchild ( BT ) ) ;
          // Perform desired action
           visit ( Data ( BT ) ) ;
     }
}
```



a+b*(c-d)-e/f

Postorder sequence: a b c d - * + e f / -

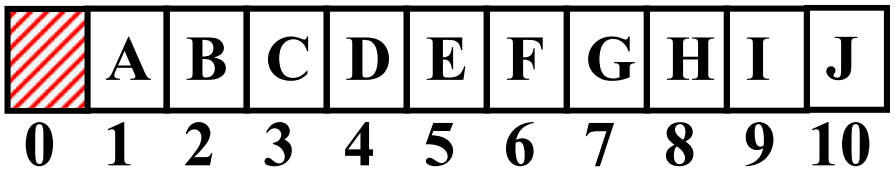
Binary Tree Implementations

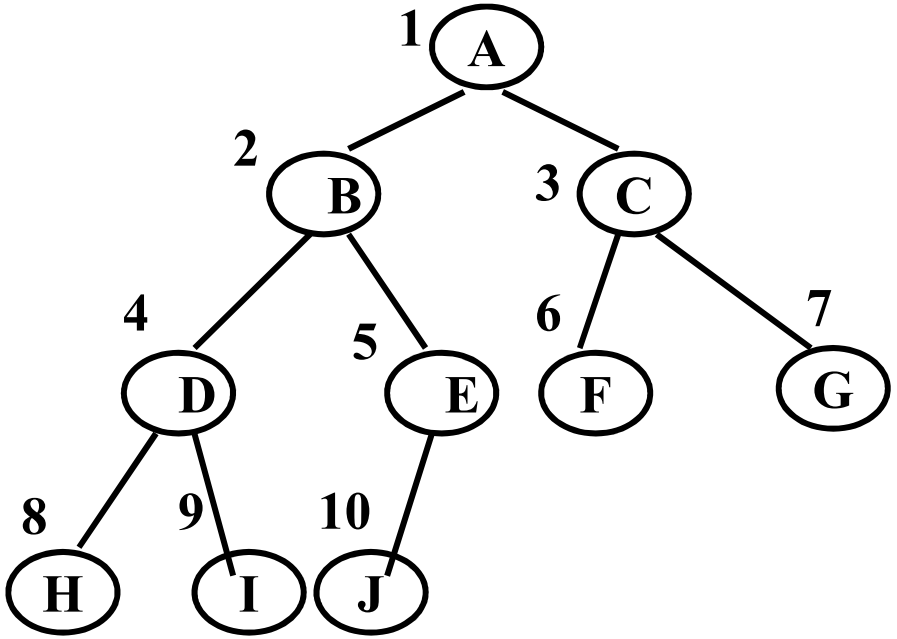
Binary Tree Implementations

- Storage structure
- There are two implementations:
  - array-based implementation
    // sequential storage stucture
  - pointer-based implementation

Array-Based Implementation

- How to use an array representation for binary trees?
- **For Complete Binary Trees**
- Use one-dimension array to store the value of nodes in Level Order





Array-Based Implementation

- How to calculate the array indices of the various relatives of a node?
- The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to n-1

Parent(r) = $\lfloor (r-1)/2 \rfloor$ if $r \neq 0$.

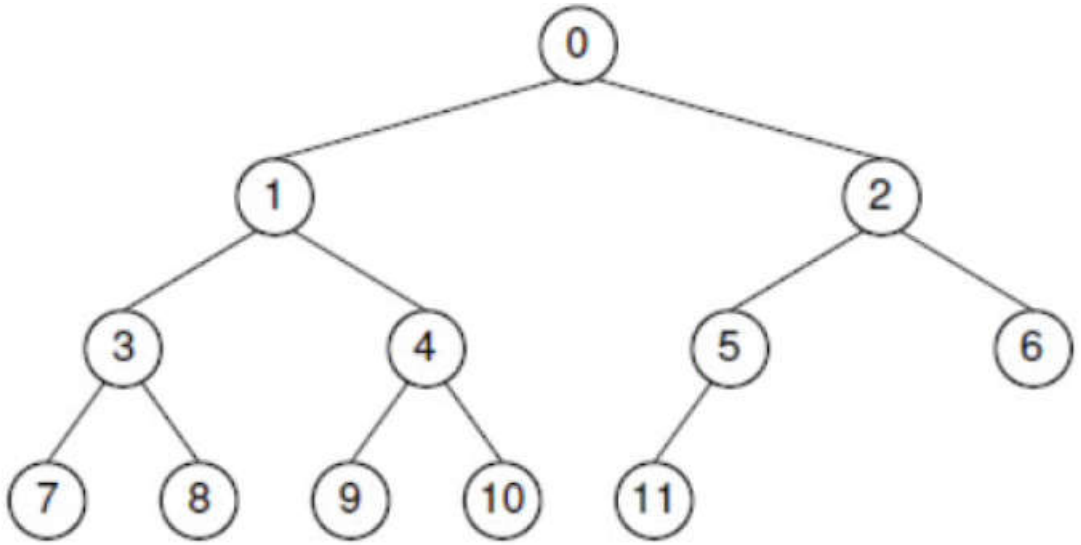
Left child(r) = $2r + 1$ if $2r + 1 < n$.

Right child(r) = $2r + 2$ if $2r + 2 < n$.

Left sibling(r) = $r - 1$ if r is even.

Right sibling(r) = $r + 1$ if r is odd and $r + 1 < n$.

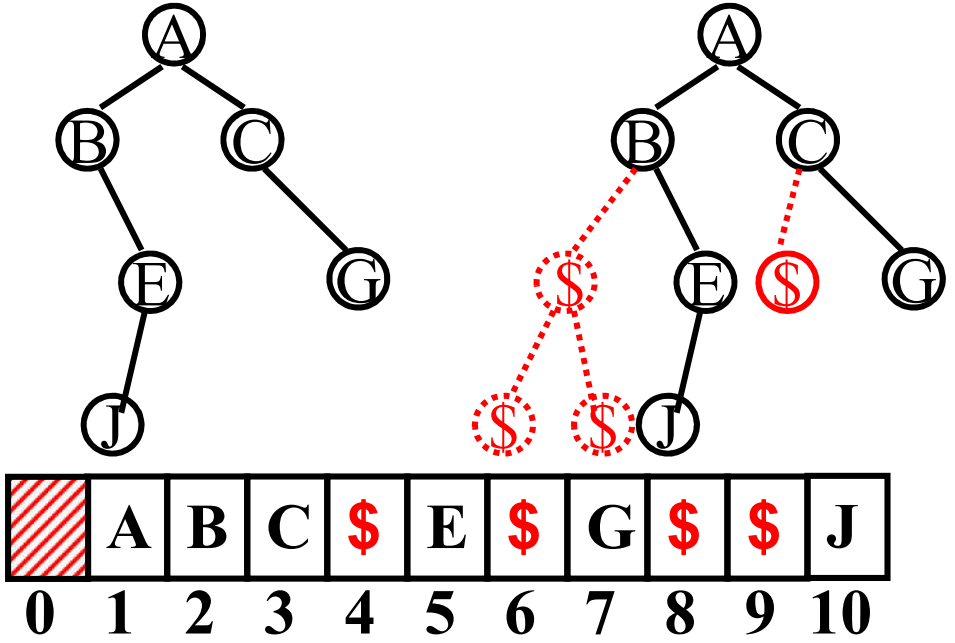
Array-Based Implementation: Example



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

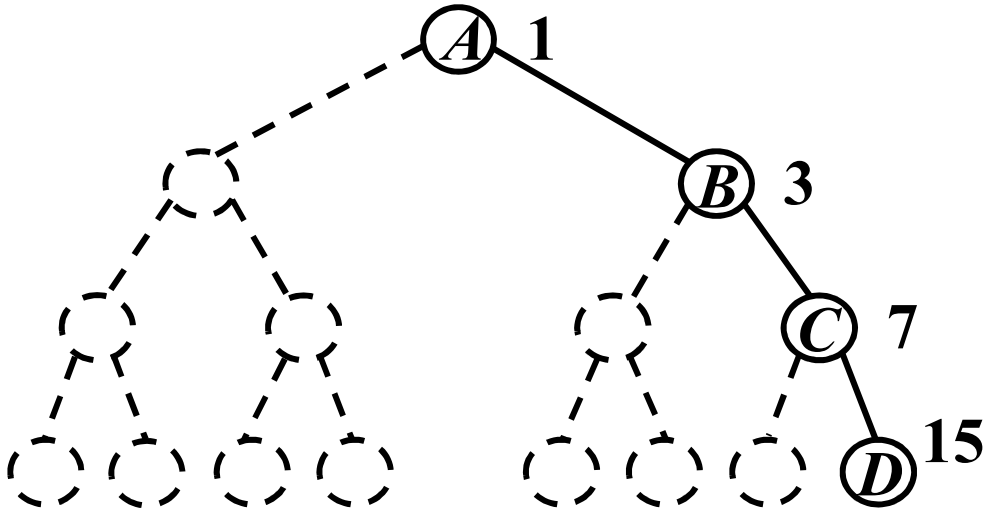
Array-Based Implementation

- For general BT, make it CBT by appending *virtual node* with value of special symbol
- Given level number i of a node, you can know where is the parent, left child and right child of the node



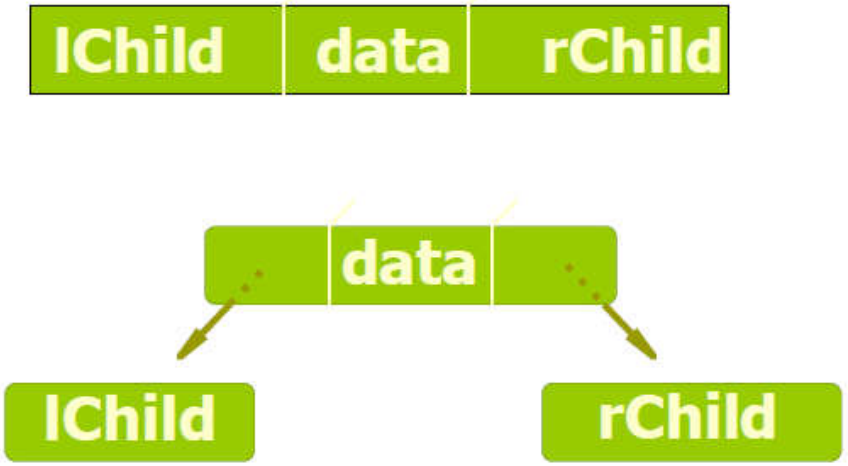
Array-Based Implementation

- For general Binary Trees
- Waste space for some cases

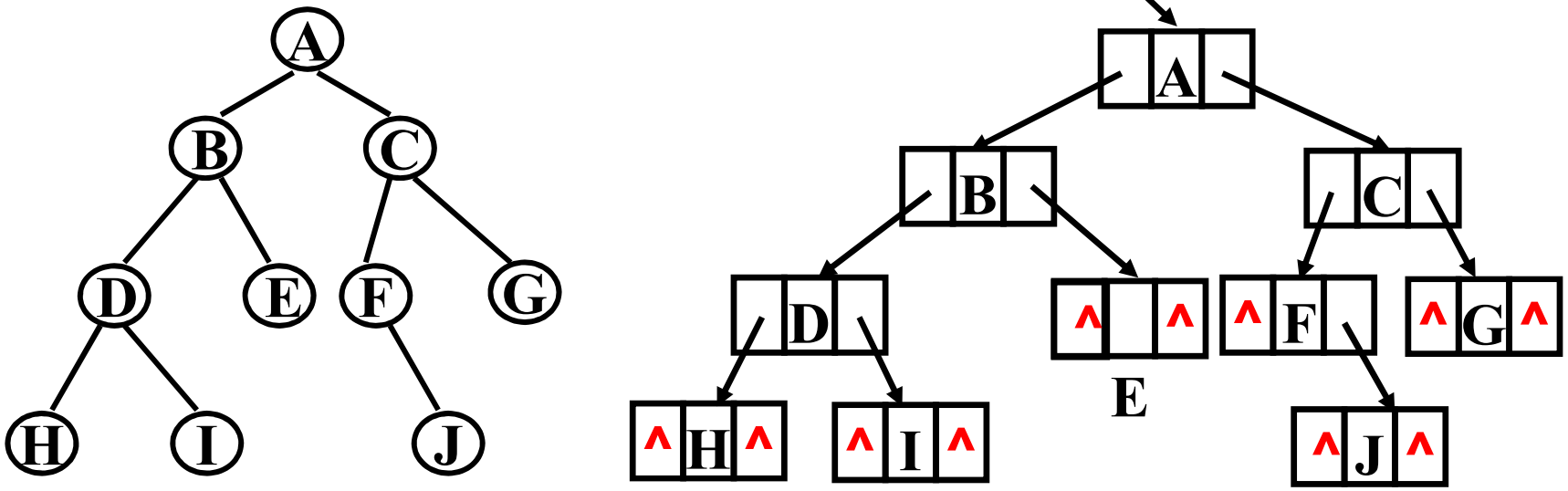


Pointer-Based Implementation

- Dynamic binary linked list:
- All binary tree nodes have two children.
- The most common node implementation includes a value field and pointers to the two children



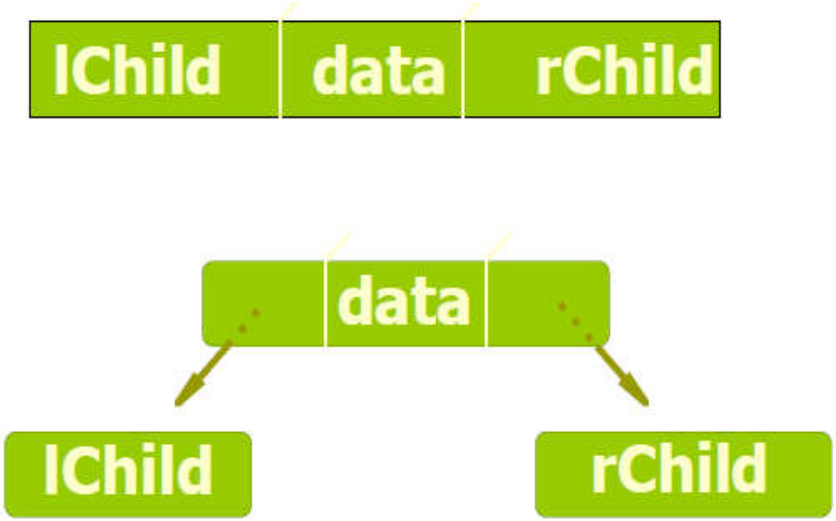
Example



How many null pointers? How many pointers to child nodes?

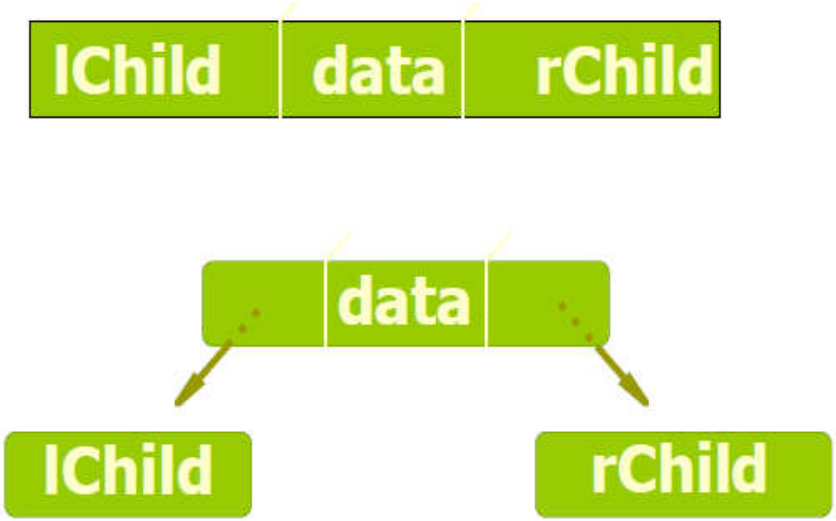
BT: Storage structure

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
          datatype  data ;
} ;
typedef  struct  node * BTREE ;
```



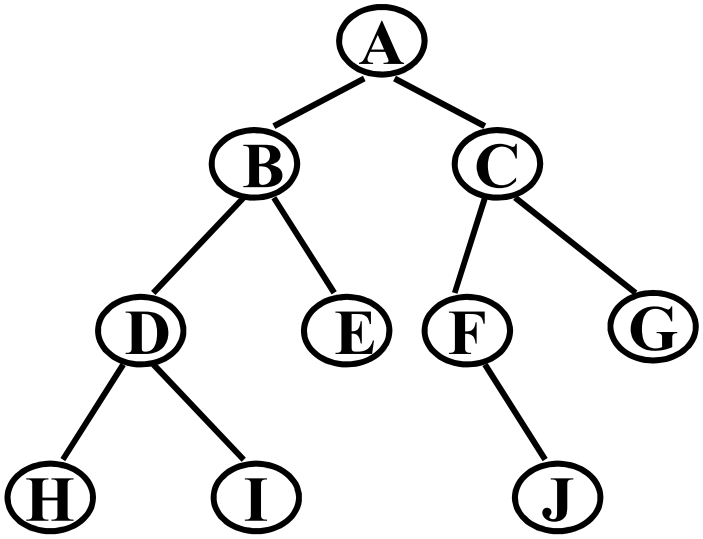
BT: Storage structure

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
          datatype  data ;
} ;
typedef  struct  node * BTREE ;
```



CreateBT: operations

```
BTREE CreateBT(datatype v, BTREE ltree , BTREE rtree )
{
    BTREE  root ;
    root = new node ;
    root →data = v ;
    root →lchild = ltree ;
    root →rchild = rtree ;
    return  root ;
}
```

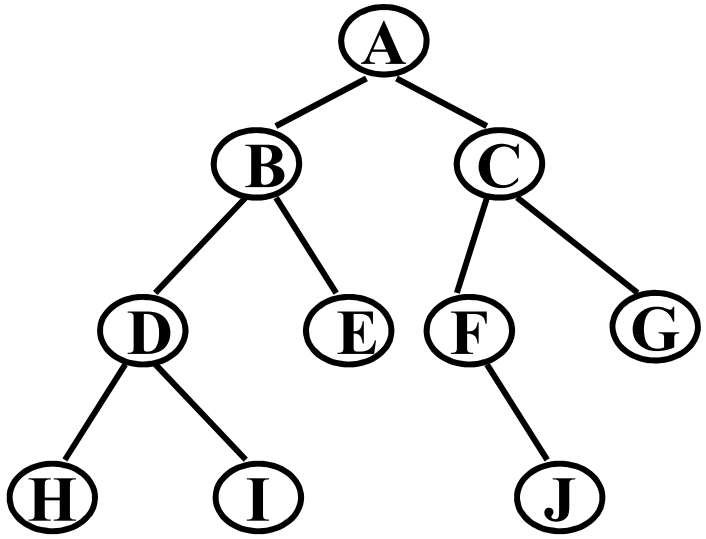


Create a BT in preorder

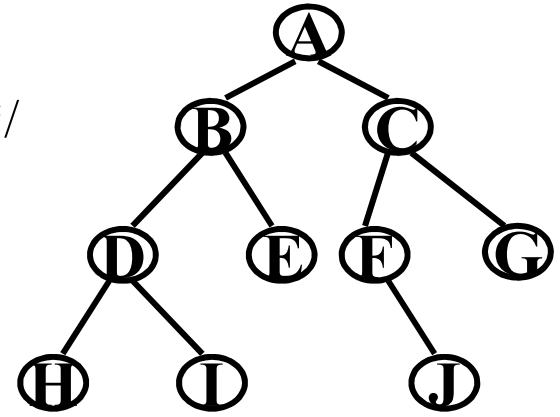
- Input: ABDH##I##E##CF#J##G##, # is null

// review the BT traversal in preorder

```
void CreateBT(BTREE & T)
{  cin >> ch ;
   if  ( ch == '#' ) T = NULL ;
   else{
        T →data = ch ;
        CreateBT ( T →lchild ) ;
        CreateBT ( T →rchild ) ;
   }
}
```



CreateBT: none-recursive




```
struct node *s[max];  /*tempory array of node pointers */
BTREE CreateBT (  )
{  int i , j; datatype ch;
   struct node *bt, *p; /* bt  is root，  p is for new node*/
   cin >> i>>ch ;
   while ( i != 0&&ch != 0) {
         p =new node;          p → data=ch;
         p → lchild=NULL;  p → lchild=NULL;
         s[ i ]=p;
         if  ( i == 1 ) bt = p ;
         else {    j =i /2;   /*parent number */
             if ( i %2==0 )   s[ j ]→lchild=p;  /* i  is left son of  j */
             else                 s[ j ]→rchild=p; /* i is right son of j */
        }cin >> i>>ch ;}
}
```

Position	0
Parent	–
Left Child	1
Right Child	2
Left Sibling	–
Right Sibling	–

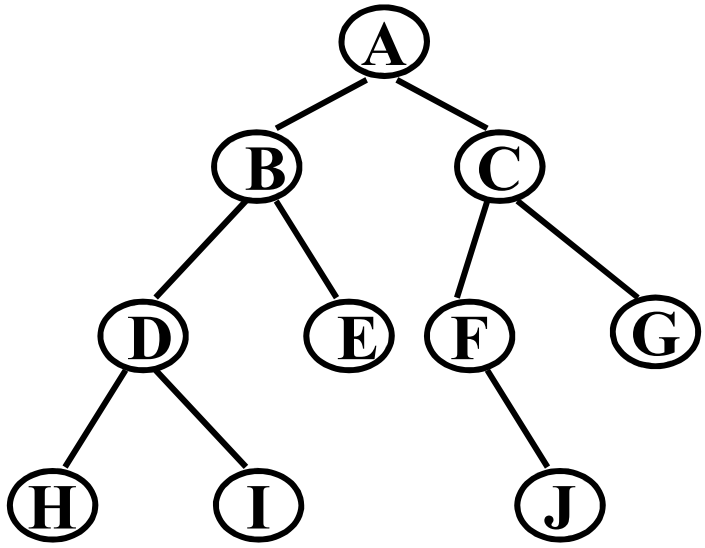
Traversal algorithm

- Recursive algorithm
- Non recursive algorithm

Recursive traverse algorithms: Preorder

A traversal routine is naturally written as a recursive function.

```
void  PreOrder (BTREE BT )
{
    if ( BT != NULL)
    {
        cout<< BT->data  ;
        PreOrder ( BT->lchild ) ;
        PreOrder ( BT->rchild ) ;
     }
}
```



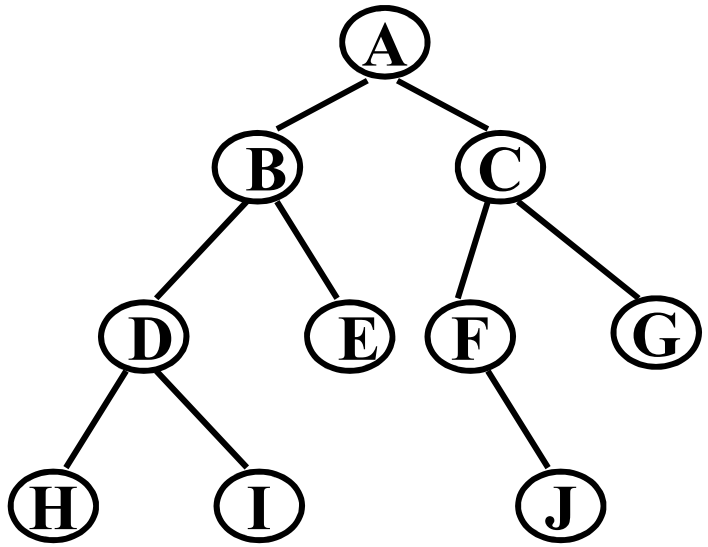
Recursive traverse algorithms: Preorder

- An important decision in the implementation of any recursive function on trees is when to check for an empty subtree.
- An alternate design as follows:

```
void preorder2(BTREE  root ) {
    visit(root); // Perform whatever action is desired
    if (root->left() != NULL) preorder2(root->left());
    if (root->right() != NULL) preorder2(root-right());
}
```

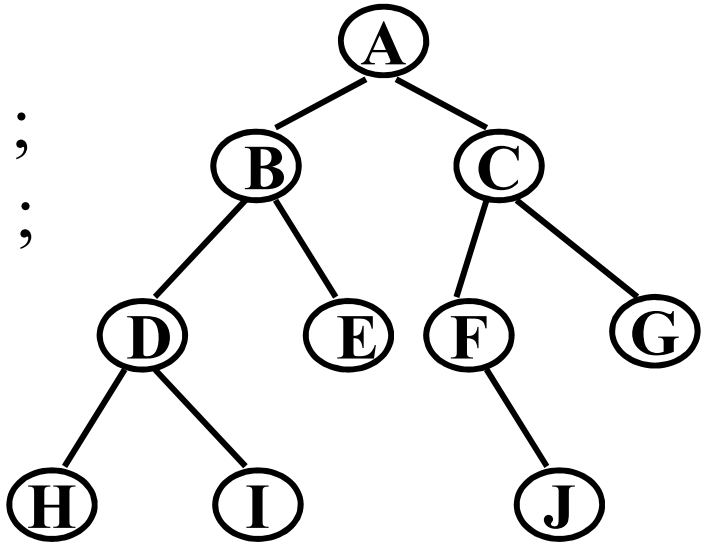
Recursive traverse algorithms: inorder

```
void  InOrder (BTREE BT )
{
    if ( BT != NULL)
    {
        InOrder ( BT->lchild ) ;
        cout<< BT->data  ;
        InOrder ( BT->rchild ) ;
     }
}
```



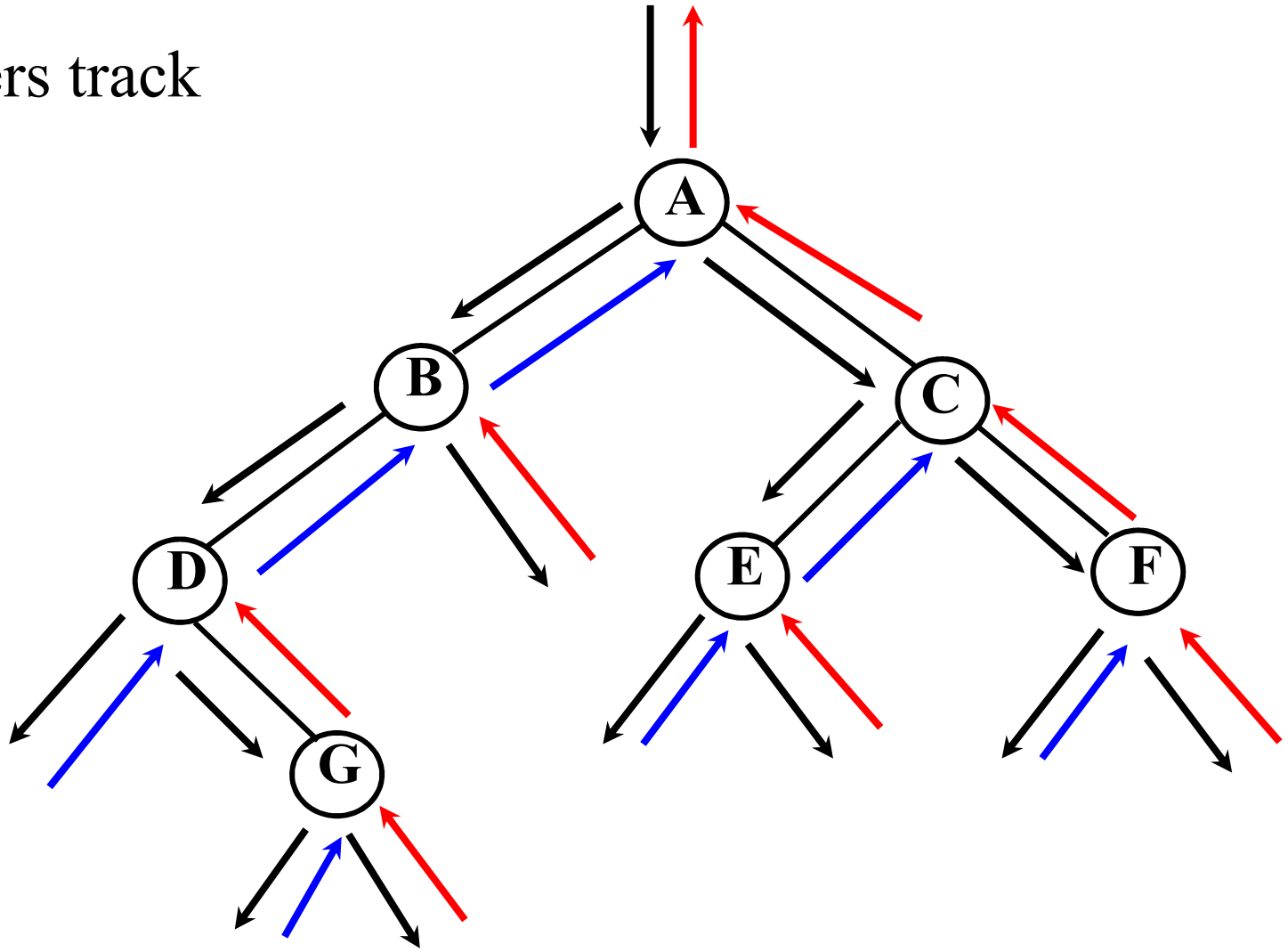
Recursive traverse algorithms: postorder

```
void  PostOrder (BTREE BT )
{
    if ( BT != NULL)
    {
        PostOrder ( BT->lchild ) ;
        PostOrder ( BT->rchild ) ;
        cout<< BT->data  ;
     }
}
```



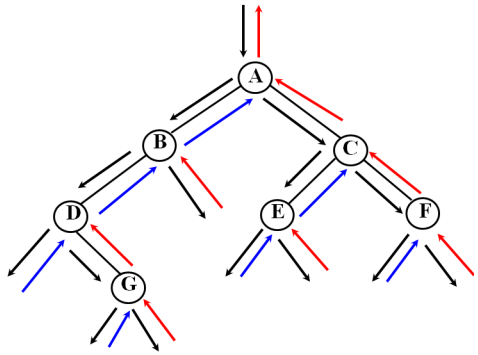
Non recursive traverse

- Travers track



Non recursive traverse algorithms: preorder

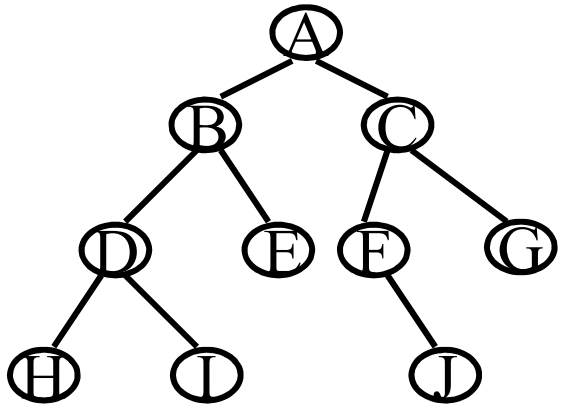
Using stack: FILO




```
0 void PreOrder(BTREE root)
1 {   top= -1;      //if stack is empty, using sequential stack
2     while (root!=NULL || top!= -1) {
2.1           while (root!= NULL) {
2.1.1                 cout<<root->data;  // visit root
2.1.2                 s[++top]=root;  // the pointer of root enters stack
2.1.3                 root=root->lchild;  // traverse the left subtree of the root
                  }
2.2           if (top!= -1) {
2.2.1                 root=s[top--];   // pop stack up to root;
2.2.2                 root=root->rchild;  //get ready to go to right subtree
                  }
      }
}
```

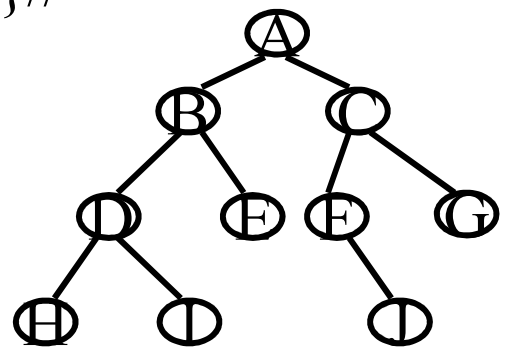
Non recursive traverse algorithms: inorder

```
0 void InOrder(BTREE  root)
1 {   top= -1;      // using sequential stack
2    while (root!=NULL | | top!= -1) {
2.1          while (root!= NULL) {
2.1.1                s[++top]=root;  // enter stack
2.1.2                root=root->lchild;   //traverse left subtree
             }
2.2           if (top!= -1) {
2.2.1                root=s[top--];   // pop top to root
2.2.2                cout<<root->data;
2.2.3                root=root->rchild;   // get ready to go right
             }
      }
   }
```



Non recursive traverse algorithms: postorder

```
0 void PostOrder(BTREE root)
1{    top= -1; //sequential stack
2     while (root!=NULL | | top!= -1)  {
2.1         while (root!=NULL)  {
2.1.1               top++;
                    s[top].ptr=root;
                    s[top].flag=1;
2.1.2               root=root->lchild;
            }//*
      }
```

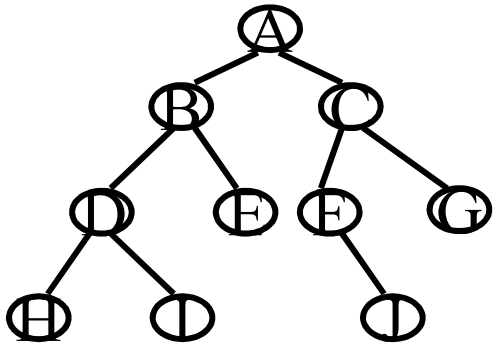



```
2.2 while (top!= -1 && s[top].flag==2) {
          root=s[top--].ptr;
          cout<<root->data;  // say, node H
       }
 2.3 if (top!= -1) {
          s[top].flag=2;  // already done, twice
          root=s[top].ptr->rchild; // go to right
       }
}
```

Non recursive traverse algorithms: level order

Using queue: FIFO

```
0 void LeverOrder (BTREE root)
1 {   front=rear=0;  //  initialization
2    if (root==NULL) return;
     Q[++rear]=root;  // enter queue
3    while (front!=rear) {
3.1            q=Q[++front];
3.2            cout<<q->data;
3.3            if (q->lchild!=NULL)  Q[++rear]=q->lchild; // enter queue
3.4            if (q->rchild!=NULL)  Q[++rear]=q->rchild; // enter queue
      }
  }
```



Applications: count nodes

Based on the core algorithms of traverse algorithms

```
int Count ( BTREE T )
{    if ( T == NULL )   return 0;
     else  return 1 + Count ( T->lchild )
                    + Count ( T->rchild );
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
        datatype  data ;
} ;
typedef node  * BTREE ;
```

Applications: height & removal

Caculate the height of a tree

```
int Height (BTREE T )
{  if ( T == NULL ) return 0;
   else {int m = Height ( T->lchild );
         int n = Height ( T->rchild ) );
         return (m > n) ? (m+1) : (n+1);
        }
}
```

Remove a tree

```
void Destroy (BTREE  T)
{
   if ( T != NULL ) {
      Destroy ( T->lchild );
      Destroy ( T->rchild );
      delete T;
   }
}
```

Applications: Exchange subtree

Exchange the subtrees of all node of a BT

```
void  Exchange ( BTREE T )
{   Node *p = T, *tmp;
    if ( p != NULL ) {
        temp = p->lchild;
        p->lchild = p->rchild;
        p->rchild = tmp;
        Exchange ( p->lchild );
        Exchange ( p->rchild );
    }
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
            datatype  data ;
} ;
typedef node  * BTREE ;
```

Application: output node

Printing the leaf nodes in preorder

```
void PreOrder(BTREE T )
{
   if (T) {
      if (!T->lchild && !T->rchild)
            cout<<T->data;
      PreOrder(T->lchild);
      PreOrder(T->rchild);
   }
}
```

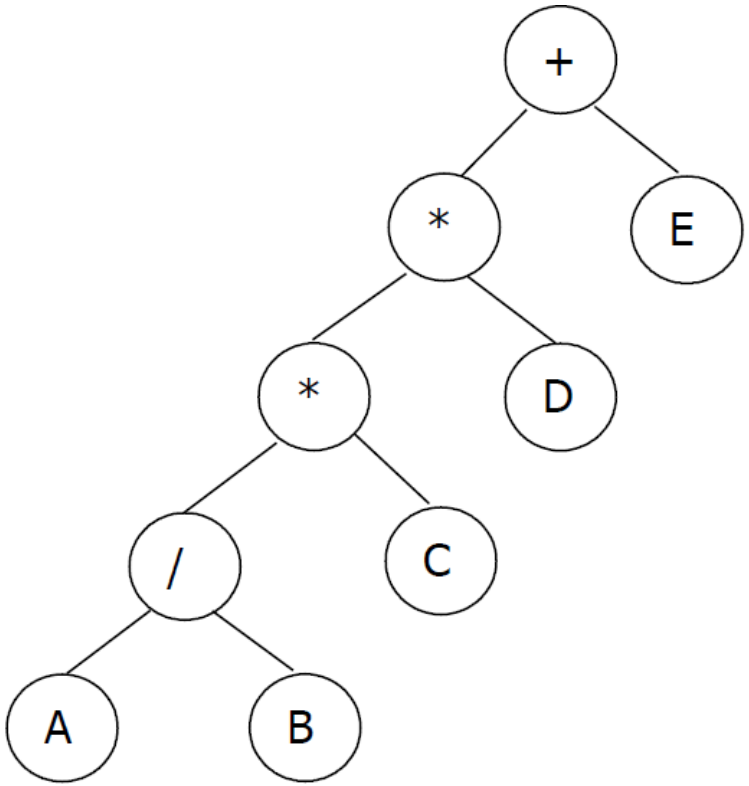
```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
          datatype  data ;
} ;
typedef node  * BTREE ;
```

Expression Tree

- A Binary Tree built with operands and operators.
- Also known as a parse tree.
- Used in compilers.
- Notation
  - Preorder
    - Prefix Notation
  - Inorder
    - Infix Notation
  - Postorder
    - Postfix Notation

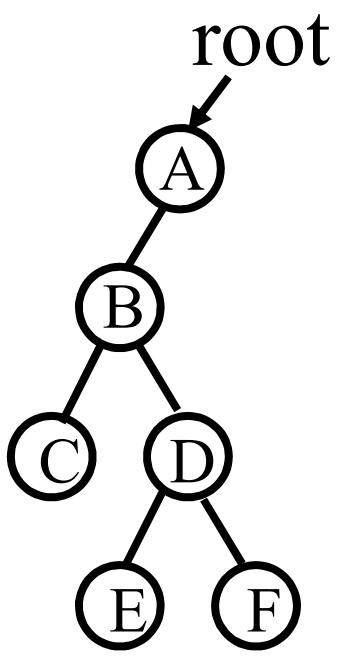
Arithmetic Expression Using BT

- inorder traversal
  A / B * C * D + E
  infix expression
- preorder traversal
  + * * / A B C D E
  prefix expression
- postorder traversal
  A B / C * D * E +
  postfix expression
- level order traversal
  + * E * D / C A B

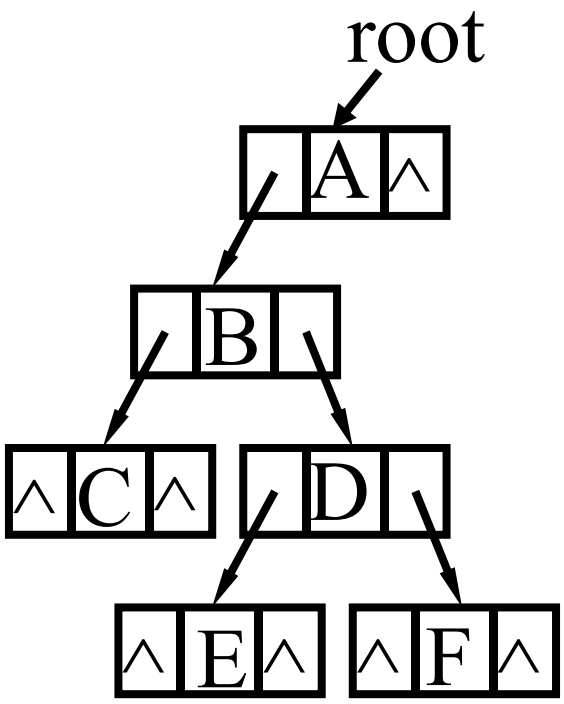


Linked storage structure of a BT

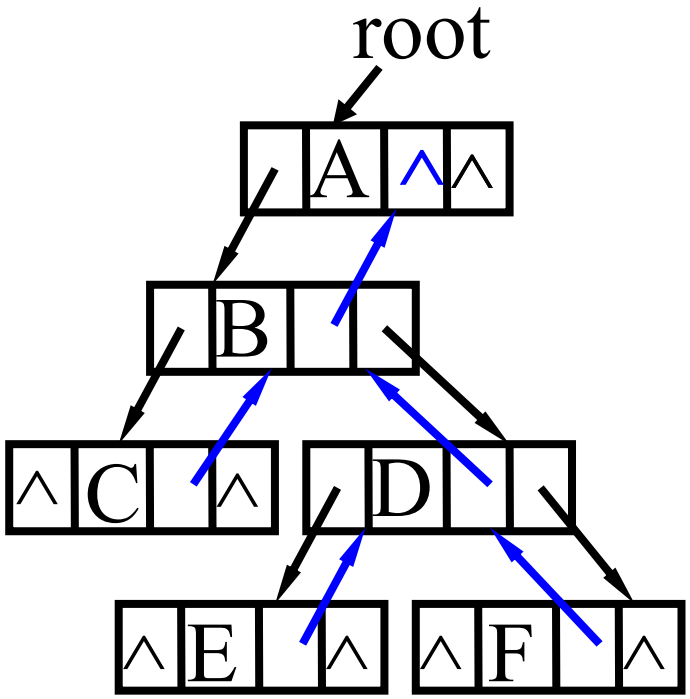
- Linked storage structure of a BT: dynamic Ternary Linked List
- A pointer pointing to its parent is involved.



BT



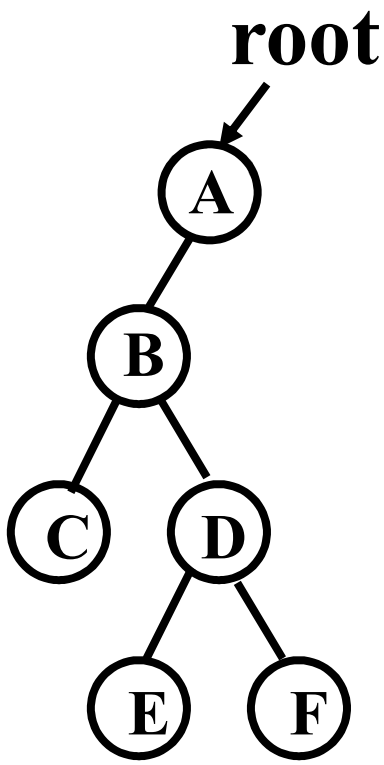
Dynamic BLL



Dynamic TLL

Linked storage structure of a BT

- **Static** binary linked list and ternary linked list



BT

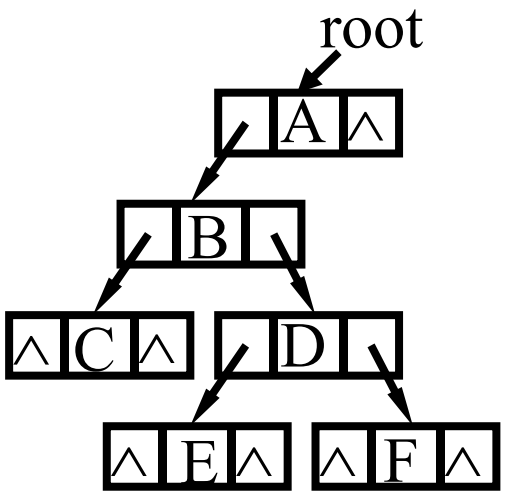
	data	parent	lchild	rchild
0	A	-1	1	-1
1	B	0	2	3
2	C	1	-1	-1
3	D	1	4	5
4	E	3	-1	-1
5	F	3	1	1

Static linked list

Threaded Binary Tree

Linked list: threaded BT

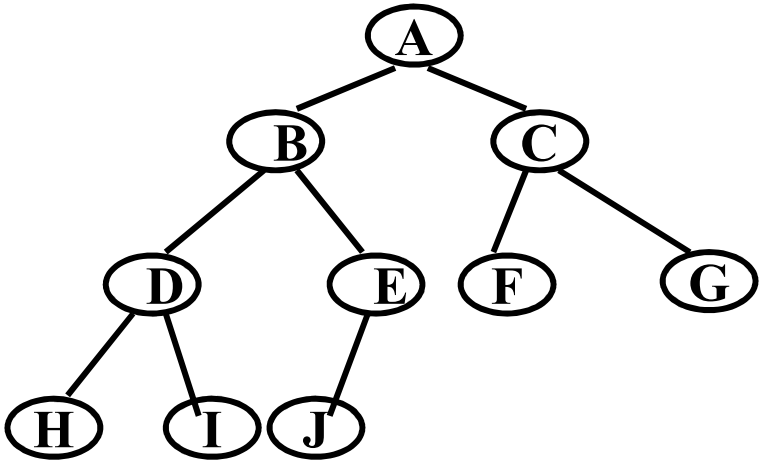
- Binary trees have lots of wasted space ?
  - A Linked list of a BT with n（n≥1）nodes, there are n-1 pointers to its subtrees, and n+1 null pointers （2n-(n-1)： total links - non-null links）
  - the leaf nodes each have 2 null pointers
- Given a node in a BT, how to find its predecessor and successor in some traversal order?
- How to traverse linked list of BT?
  - If we don't use stack and queue
- How to express the traversal order of a BT?
- Use the null pointers ? Replace these null pointers with some useful "threads".

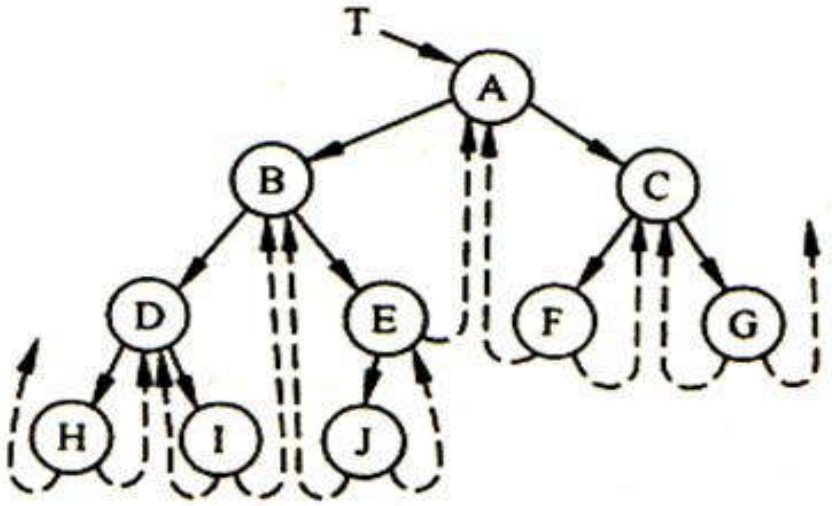


Linked list: threaded BT

- We can use the pointers to help us in traversals
  - If node p has left child, then p->lchild points to it，Otherwise, points to its predecessor (in preorder, inorder, and postorder), which is called *thread*
  - If node p has right child, then p->rchild points to it，Otherwise, points to its successor (in preorder, inorder, and postorder), which is called *thread*
- We need to know if a pointer is an actual link or a thread
  - The two tags are used to keep a boolean for each pointer

Example: threaded BT in inorder traversal





The data structure of a node

lchild	ltag	data	rchlid	rtag

$$p\text{->ltag} = \begin{cases} \text{TRUE} & p\text{->lchild} \ \text{//pointing to left child} \\ \text{FALSE} & p\text{->lchild} \ \text{//pointing to the predecessor} \end{cases}$$

$$p\text{->rtag} = \begin{cases} \text{TRUE} & p\text{->rchild} \ \text{//pointing to right child} \\ \text{FALSE} & p\text{->lchild} \ \text{// pointing to the successor} \end{cases}$$

Example: threaded BT in inorder traversal

- There are 4 kinds of threaded BT corresponding the 4 ways to traversal

  (1) preorder threaded BT; (2) in order threaded BT;

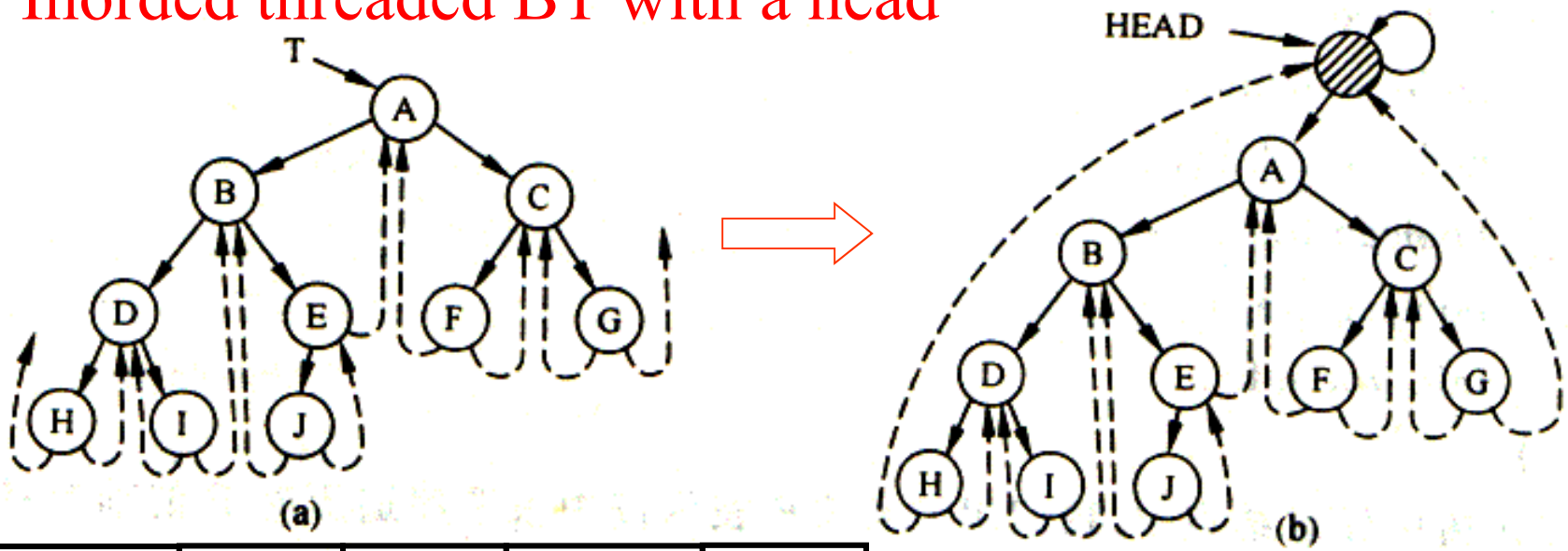
  (3) postorder threaded BT; (4) level order threaded BT

- Definition of the storage structure of a node

```
struct  node {
        datatype  data ;
        struct  node *lchild, *rchild;
                bool   ltag, rtag;
};
typedef  struct node  *  THTREE;
```

Example: threaded BT in inorder traversal

- Inorded threaded BT with a head



lchild	ltag	data	rchlid	rtag

Nonempty threaded BT：
head->lchild = T //root
head->ltag = TRUE ;
head->rchild = head ;
head->rtag = TRUE ;

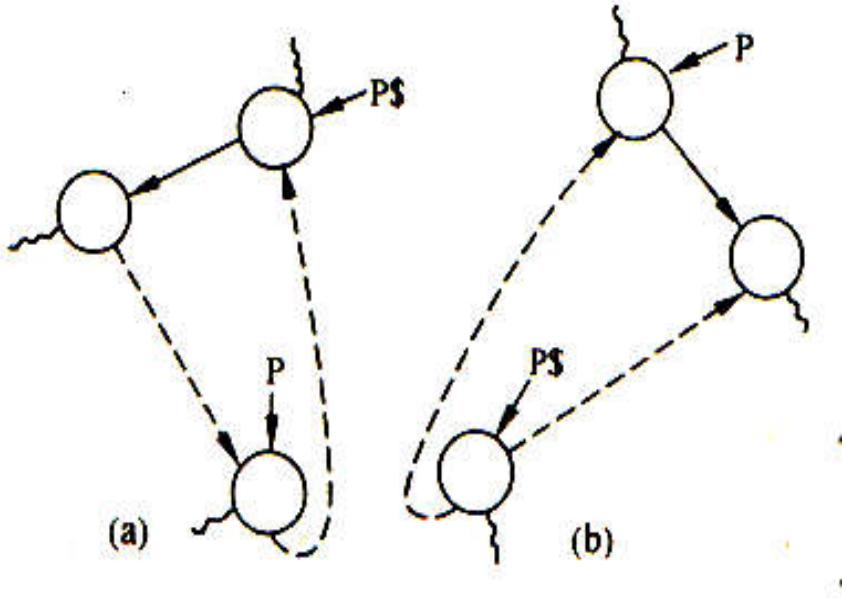
Empty threaded BT：
head->lchild = head ;
head->ltag = FALSE ;
head->rchild = head ;
head->rtag = TRUE ;

Algorithm1: Next Node in Threaded BT

a) If p->rtag==FALSE，p->rchild is the next node (thread);

b) If p->rtag==TRUE，p$ is the mostleft node of p.

```
THTREE  InNext( THTREE  p)
{  THTREE  Q ;
    Q=p->rchild ;
    if (p->rtag = = TRUE )
       while( Q->ltag = = TRUE )
         Q = Q->lchild ;
    return ( Q ) ;
}
```

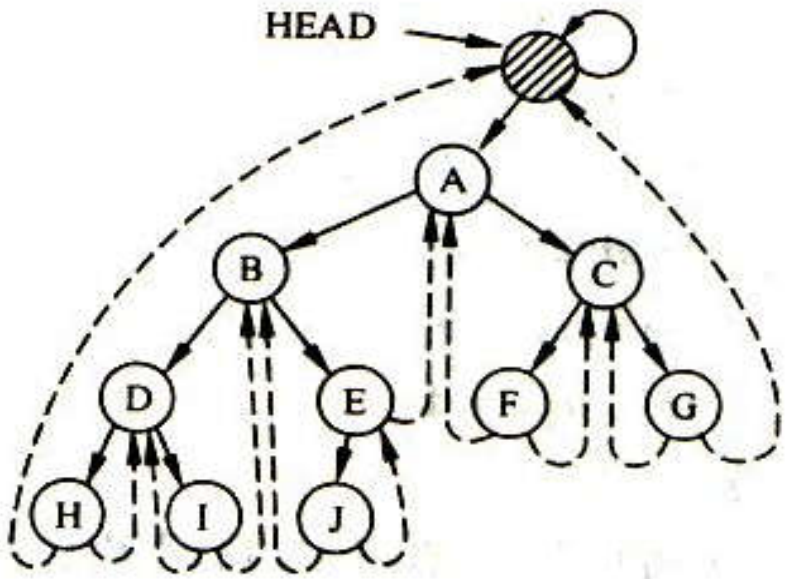


Algorithm2:
Inorder Traversal of Threaded BT

- Using InNext

```
void ThInOrder(THTREE HEAD)
{   THTREE   tmp ;
    tmp = HEAD ;
    do {
        tmp = InNext ( tmp ) ;
        if ( tmp != HEAD )
            visit ( tmp -> data ) ;
    } while ( tmp != HEAD ) ;
}
```

```
head->lchild = T
head->rchild = head ;
head->ltag = TRUE ;
head->rtag = TRUE ;
```



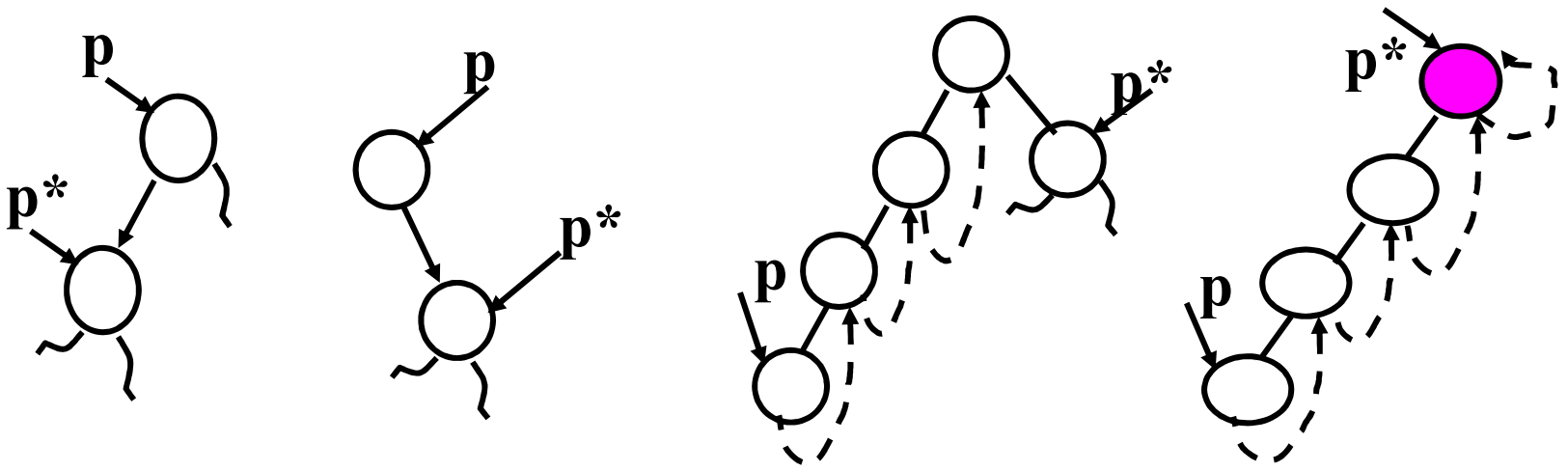
Algorithm3: Sucessor p*in preorder of node p in inorder threaded BT

- Analysis：LDR, DLR

(1) if p->lchild is non empty, then p->lchild is p*;

(2) if p->lchild is empty and p->rchild is non empty, then p->rchild is p*;

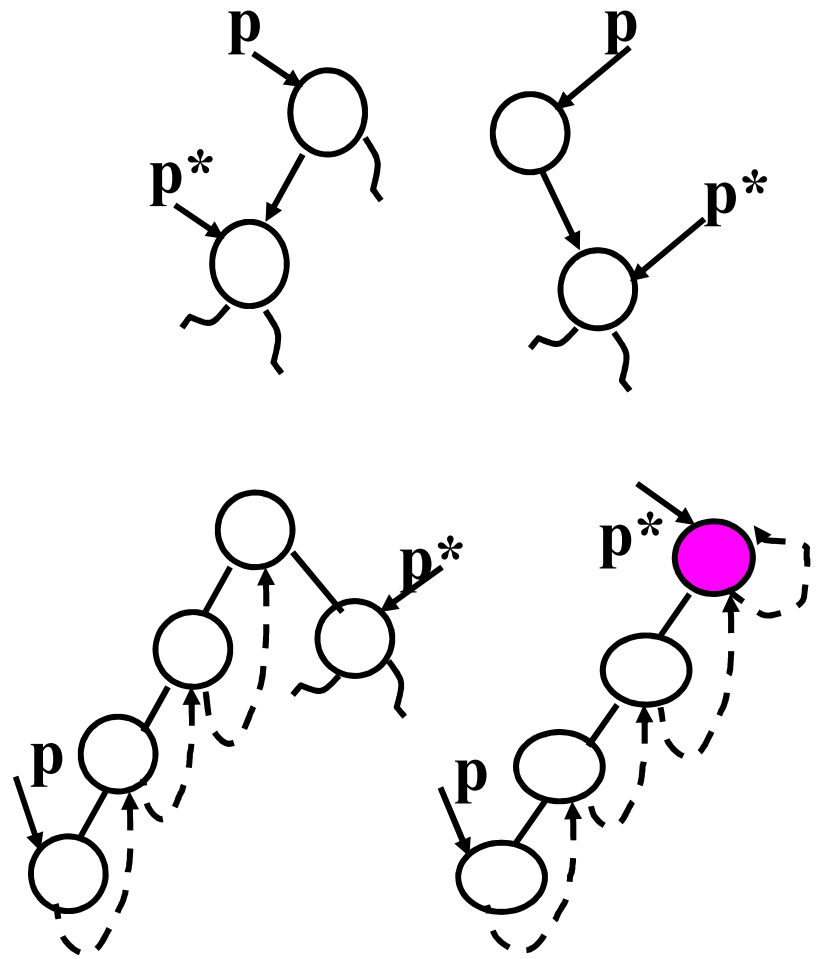
(3) if p->lchild is empty and p->rchild is empty, then it is the right child which parent is the first one has right child, or the head.



DLR

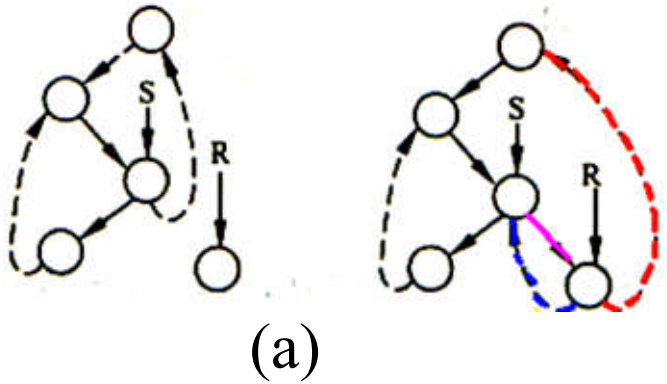
Algorithm3: Successor p*in preorder of node p in inorder threaded BT

```
THTREE   PreNext( THTREE  p)
{   THTREE   Q ;
    if (p->ltag = = TRUE )
            Q=p->lchild ;
     else{ Q = p;
            while(Q->rtag = = FALSE)
                Q = Q->rchild ;
            Q = Q->rchild ;
       }  return ( Q ) ;
}
```

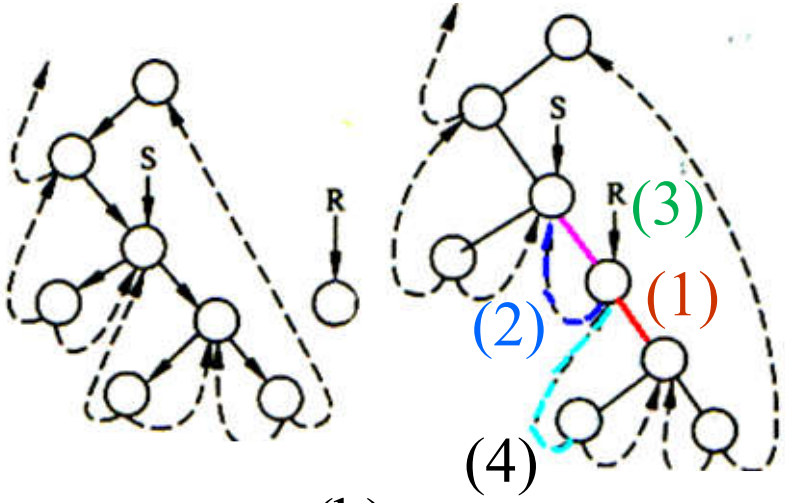


Algorithm 4:
Inserting Nodes into InordedThreaded BTs

```
void RInsert (THTREE S ,THTREE R)
{  THTREE   W ;
   R->rchild = S->rchild;
   R->rtag = S->rtag ;
   R->lchild = S ;//--
   R->ltag = FALSE ;//--
   S->rchild = R ;
   S->rtag = TRUE ;
   if (R->rtag==TRUE) { //case b
         w = InNext( R ) ;
         w->lchild = R ;  }
}
```



(a)



(b)

Algorithm 5: Threadize (inorder) a BT (recursive)

```
BTREE *pre=NULL;  //global varible of the precedcessor
void  InOrderTh(BTREE *p)
1 {   if ( p ){ //if root is not empty, otherwise return.
2   InOrderTh( p->lchild ); // threadize the left subtree of root p
3   //threadize the root
        p->ltag=( p->lchild ) ? TRUE : FALSE; // if non empty, then tag=1,
        p->rtag=( p->rchild )? TRUE : FALSE; //otherwise 0.
        if ( pre ) { // if there is predecessor *pre for node*p
          if ( pre->rtag ==FASLE) //  rtag is the thread
               pre->rchild=p; // set right thread of *pre to the inorder successor
          if ( p->ltag ==FASLE)  // ltag is the thread
               p->lchild=pre;         //set inorder predecessor
        }
        pre = p;   // set pre to be the predecessor of next visiting node
4       InOrderTh( p->rchild ); // threadize the right subtree of root p
  }
}
```

Two BTs are the same?

```
// both shape and data of a node.
int  Equal( BTREE firstbt, BTREE secondbt )
{    int  x ;
     x = 0 ;
     if ( IsEmpty(firstbt) && IsEmpty(secondbt) )
          x = 1 ;
     else if ( !IsEmpty( firstbt ) && ! IsEmpty( secondbt ) )
          if (  Data( firstbt ) == Data( secondbt )  )
               if ( Equal( Lchild( firstbt ) , Lchild( secondbt ) ) )
                    x= Equal( Rchild( firstbt ) , Rchild( secondbt) )
     return( x ) ;
} /* Equal */
```

General Tree

General Tree

Basic Operations

- Parent(n , T) // n ‘s parent
- LeftMostChild(n , T)
- RightSibling(n , T) // return the right sibling
- Data(n , T) // return value of data field
- CreateK r (v , T_1 , T_2 , …… , T_k) , k = 1 , 2 , ……
  // create a tree r, v is the value of which data field, and has k subtrees, from left to right are T_1 , T_2 , …… , T_k ; return r。
- Root(T) // return root node

Copy a BT

```
BTREE  Copy( BTREE  oldtree )
{
   BTREE   temp ;
   if ( oldtree != NULL ) {
         temp = new  Node ;
         temp -> data = oldtree->data ;
         temp -> lchild = Copy( oldtree->lchild ) ;
         temp -> rchild = Copy( oldtree->rchild ) ;
         return ( temp );
   }
   return ( NULL ) ;
}  /* Copy */
```

General Tree Traversals

Basic Operations: Traversals

- For general trees, preorder, postorder traversals and level are defined with meanings similar to their binary tree counterparts.
- Preorder traversal of a general tree:
  - first visits the root of the tree, then performs a preorder traversal of each subtree from left to right.
- A postorder traversal of a general tree:
  - performs a postorder traversal of the root's subtrees from left to right, then visits the root.
- Inorder traversals are generally not useful with general trees.
- Levelorder traversal :
  - From top to bottom , from left to right.

General Tree Traversals

- Preorder traversal:

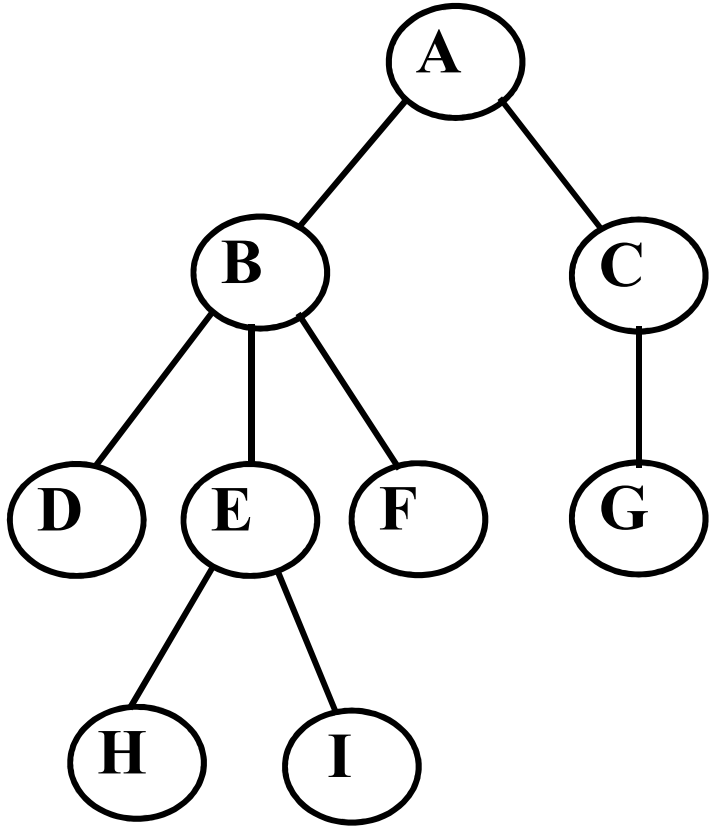
  A B D E H I F C G

- Postorder traversal:

  D H I E F B G C A

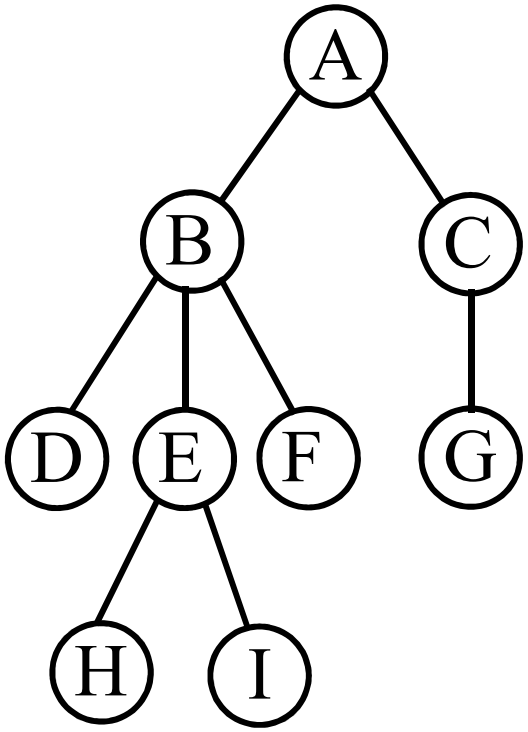
- Levelorder traversal:

  A B C D E F G H I



Algorithm: preorder traversal

```
void  PreOrder(node n, TREE  T )
{   node  c ;
    visit( Data( T ) ) ;
    c = LeftMostChild( n , T ) ;
    while ( c != NULL ) {
        PreOrder( c , T ) ;
        c = RightSibling( c , T ) ;
    }
}
```

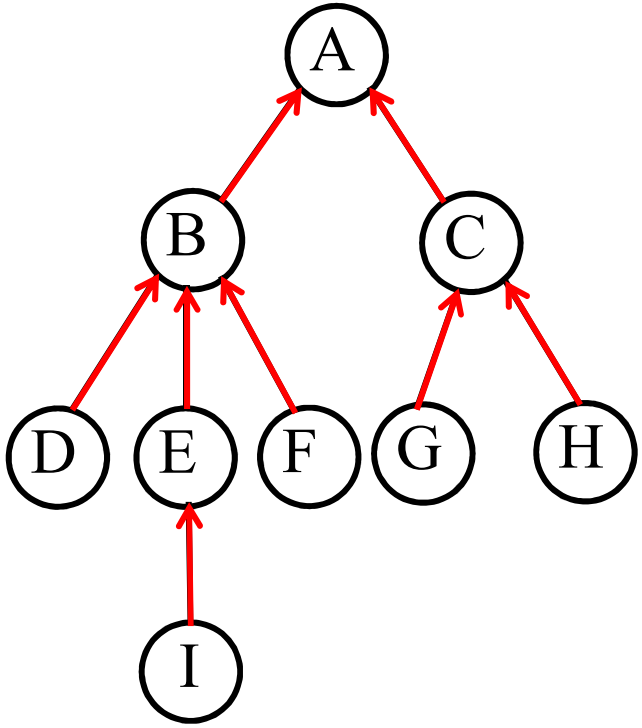


PreOrder（Root(T)， T）

General Tree implementation

- Parent pointer implementation
  - Each node has only one parent node
  - One dimension array

	1	2	3	4	5	6	7	8	9
data	A	B	C	D	E	F	G	H	I
parent	0	1	1	2	2	2	3	3	5

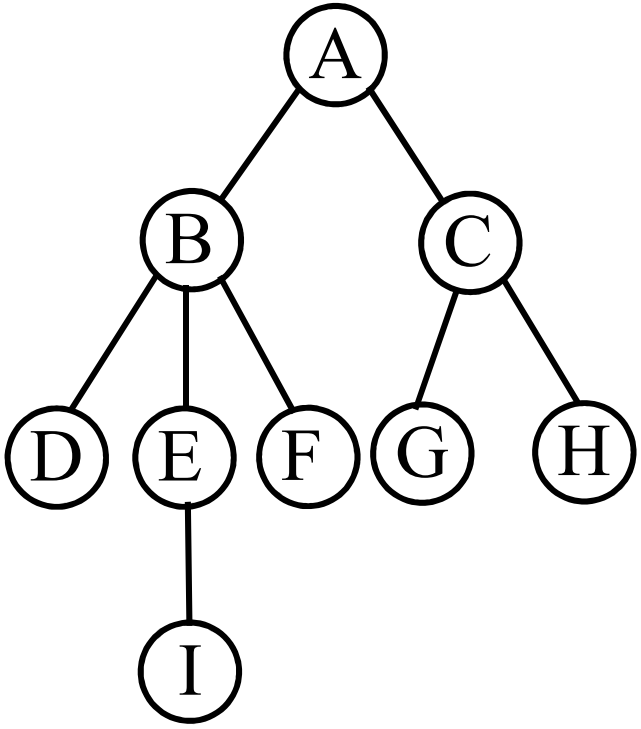



```
struct node {
        T data;    //data field
         int parent; //pointer field, index in the array
} ;    //a dynamic linked list
```

Parent pointer implementation

- How to find its parent and ancestor? performance?
- How to find its child? performance?
- How to find its sibling? performance?

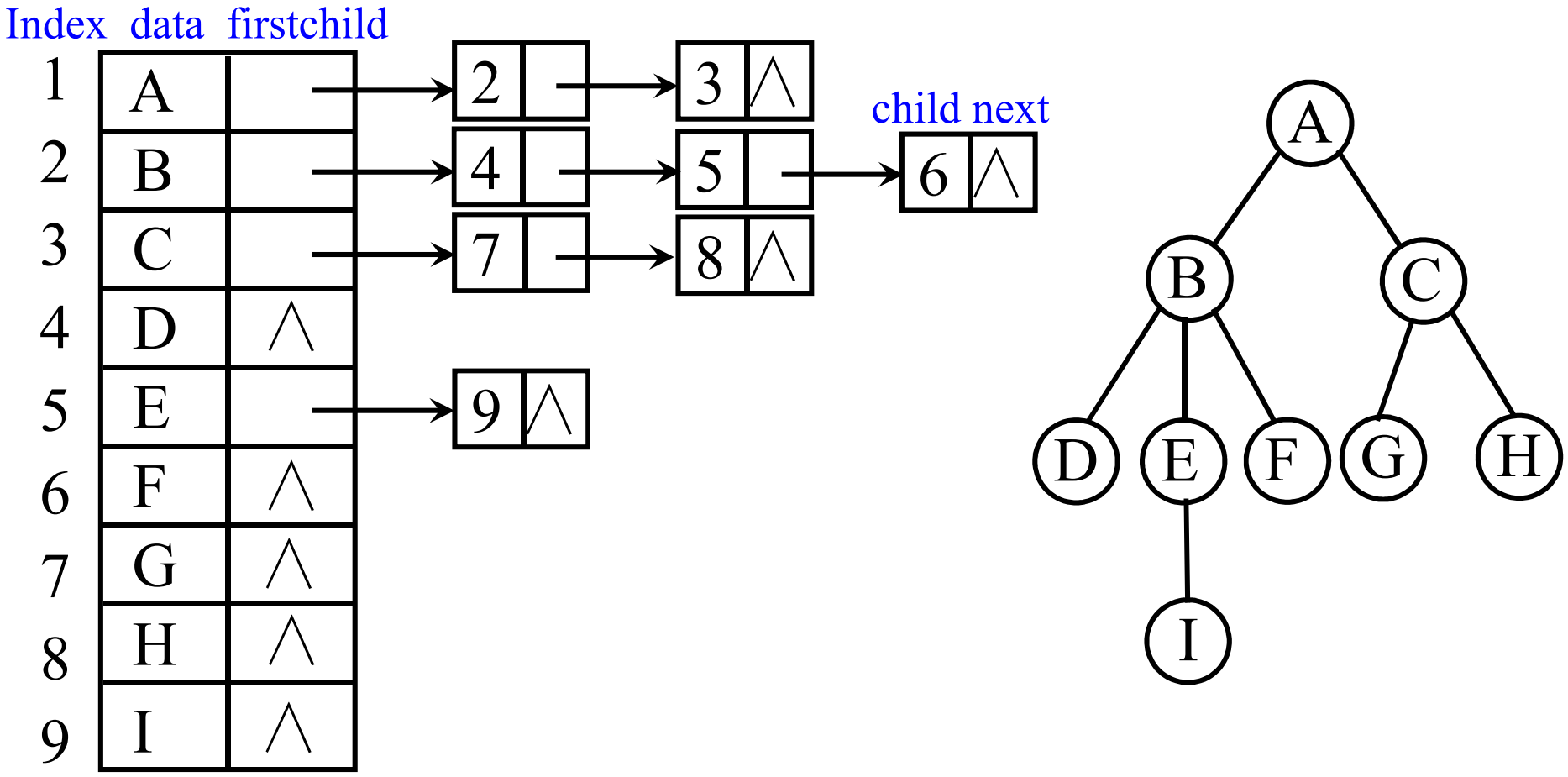
	1	2	3	4	5	6	7	8	9
data	A	B	C	D	E	F	G	H	I
parent	0	1	1	2	2	2	3	3	5
firstchild	2	4	7	0	9	0	0	0	0
rightsib	0	3	0	5	6	0	8	0	0



Children Implementations

- It simply stores with each internal node a linked list of its children.
- Each node contains a value,
  - a pointer (or index) to its parent, and
  - a pointer to a linked list of the node's children, stored in order from left to right.
- Array of n heads of linked lists

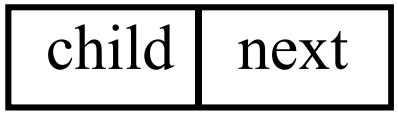
Children Implementations



Children Implementations

```
struct  CTNode  {
    int    child ;
    CTNode  *next ;
 };
struct CTBox{
     DataType    data ;
     CTNode  * firstchild ;
} ;
struct  {
     CTBox  nodes[MaxSize] ;
     int   n , r ;
} CTree ;
```

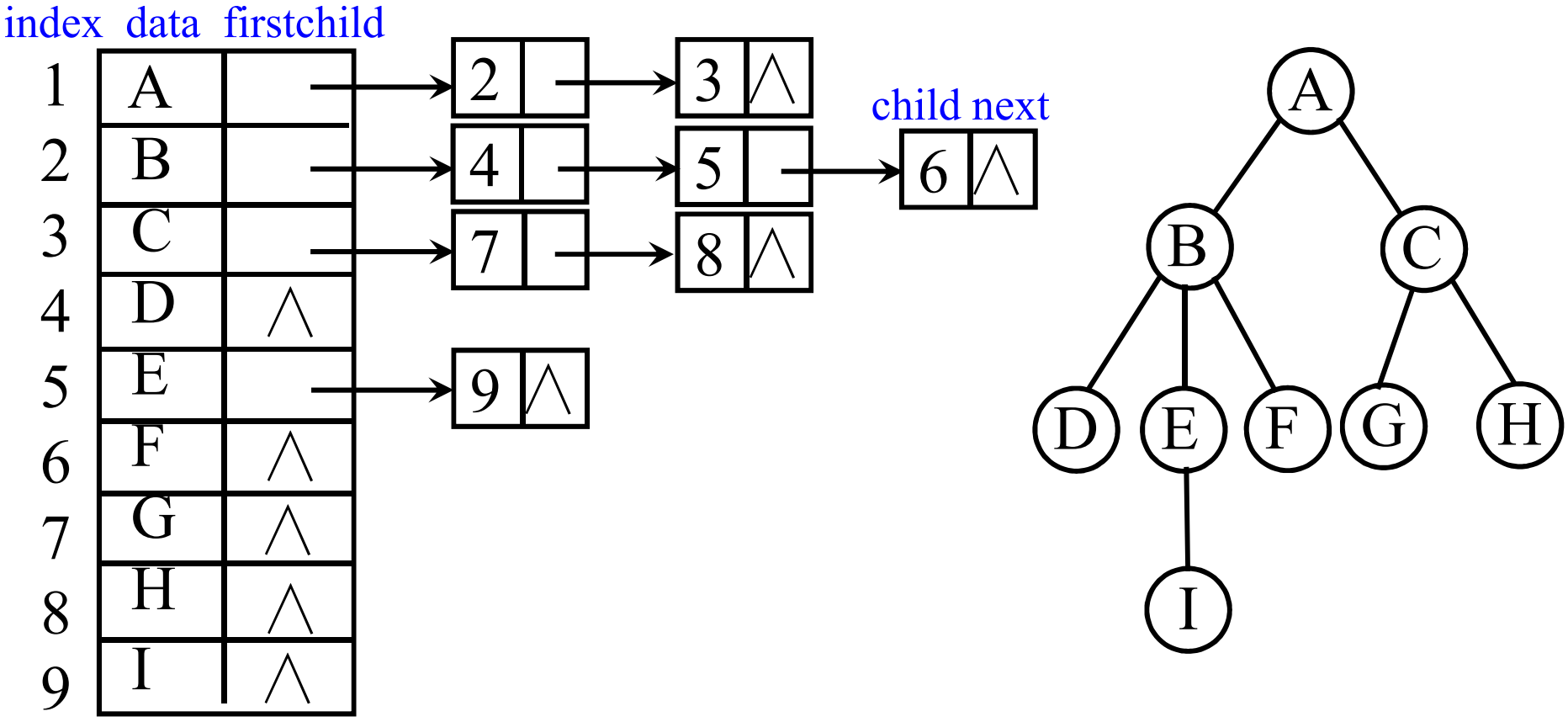
Children node



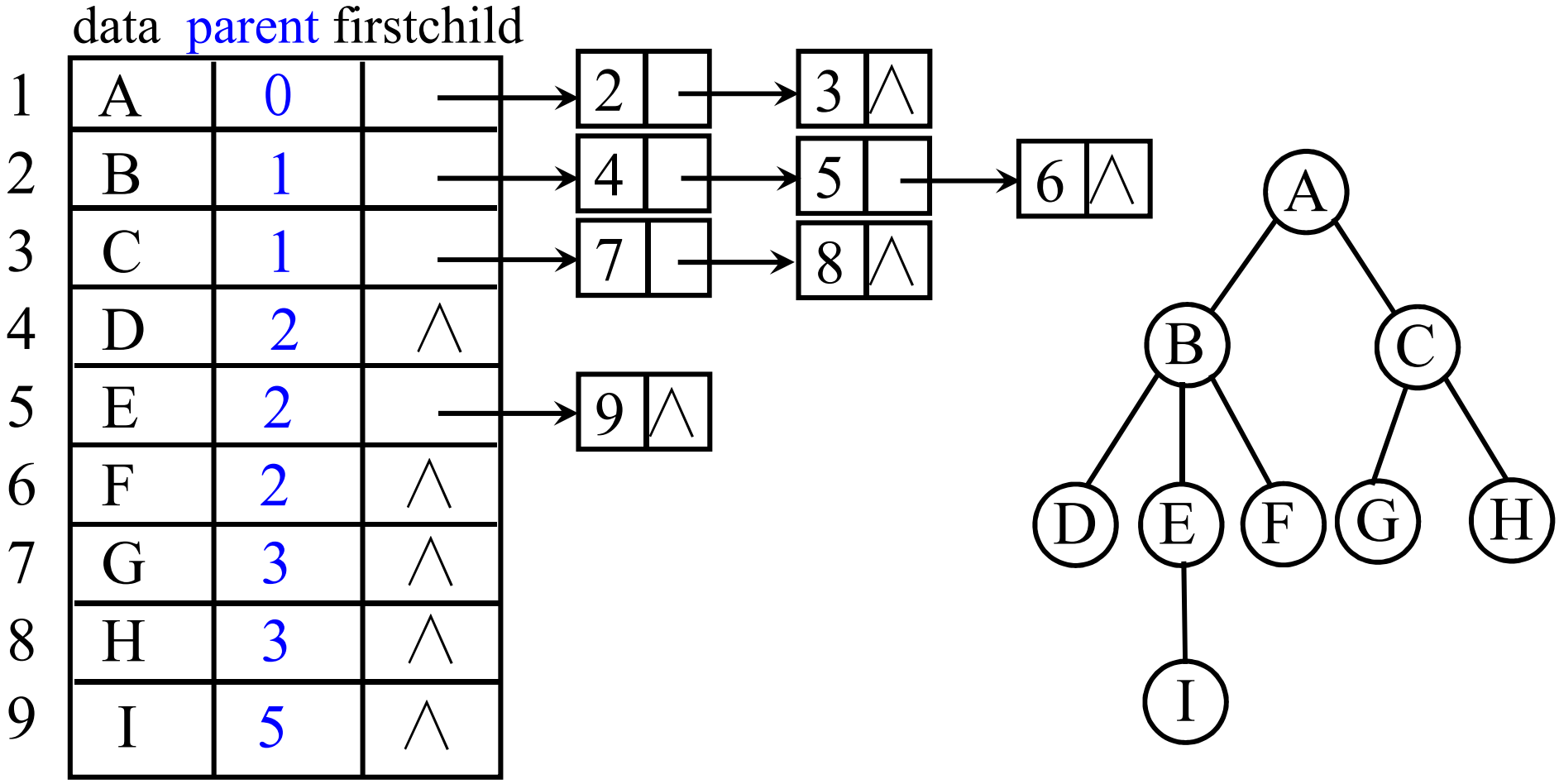
Head node

Children Implementations

- How to find children? performance?
- How to find parent? performance?

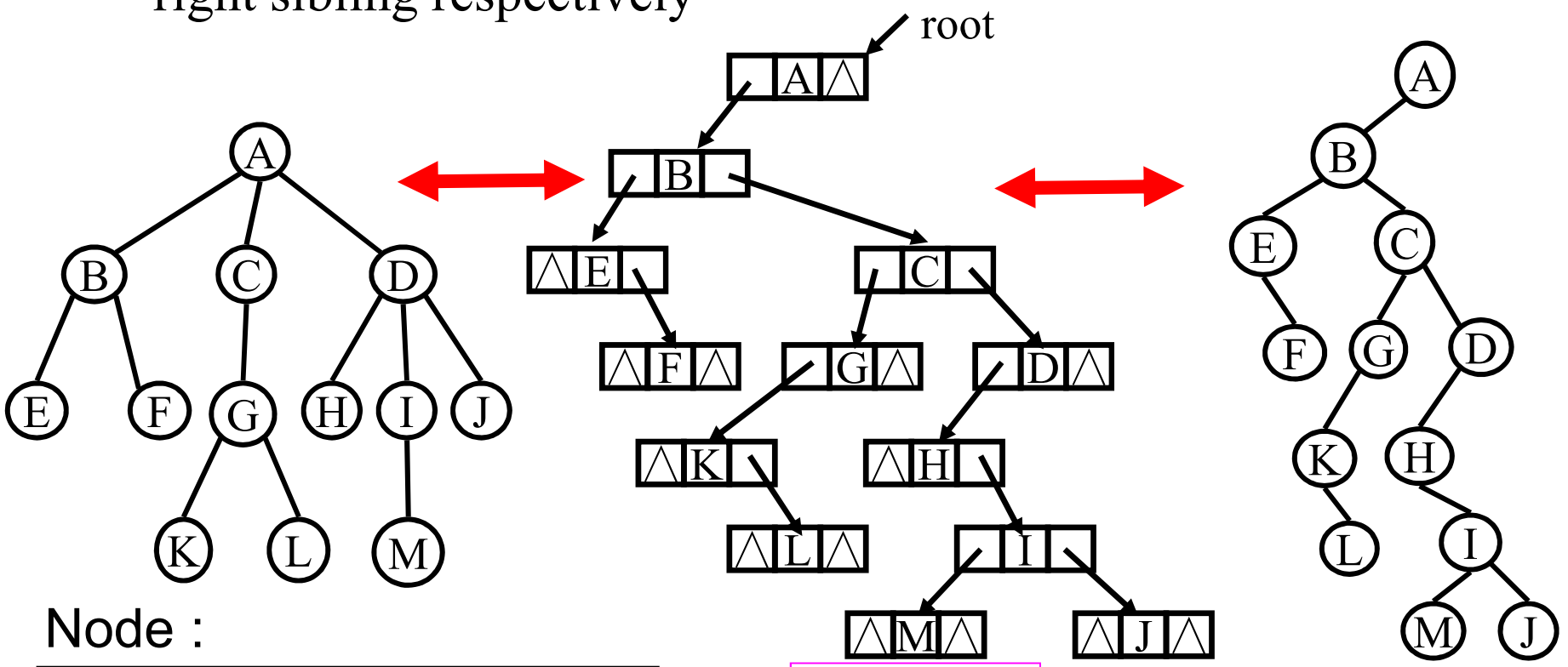


Parent-children implementation

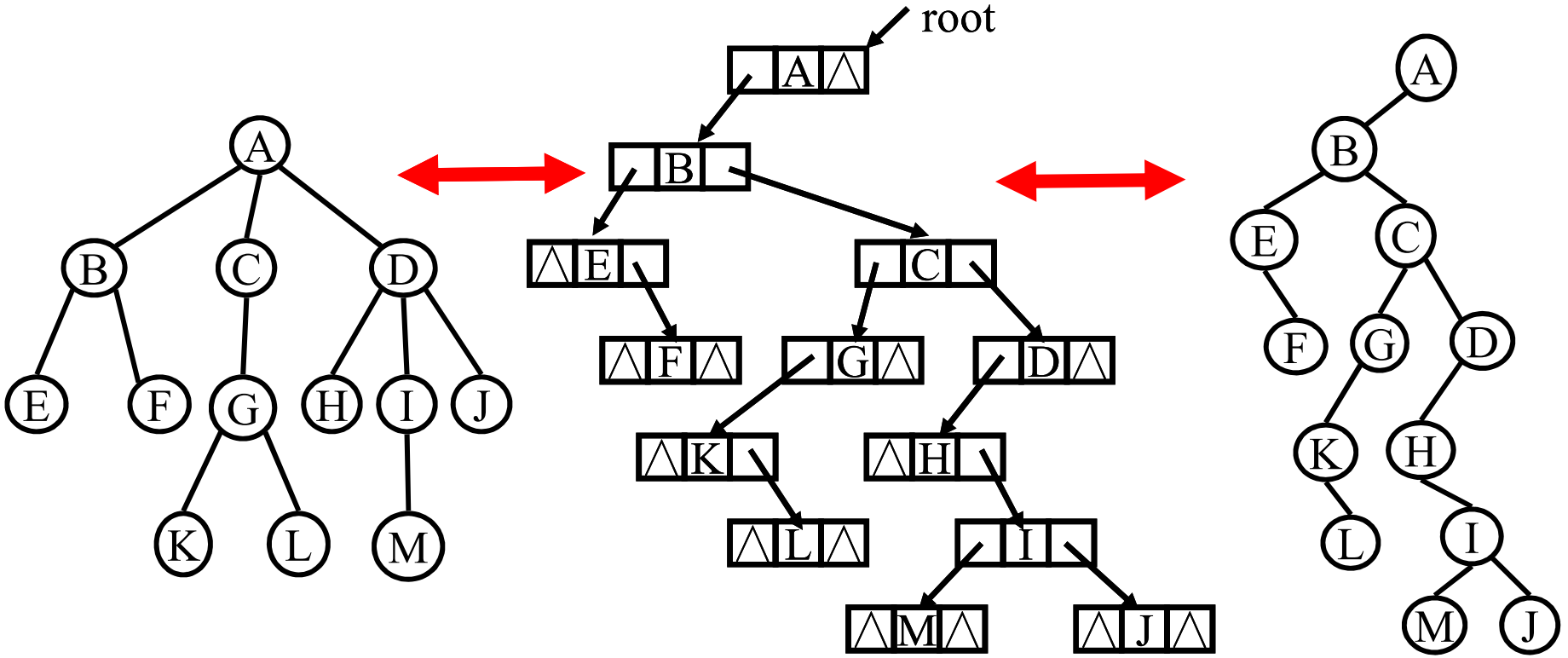


BT implementation

- Two pointers are used, which pointing to the first child and right sibling respectively



BT implementation



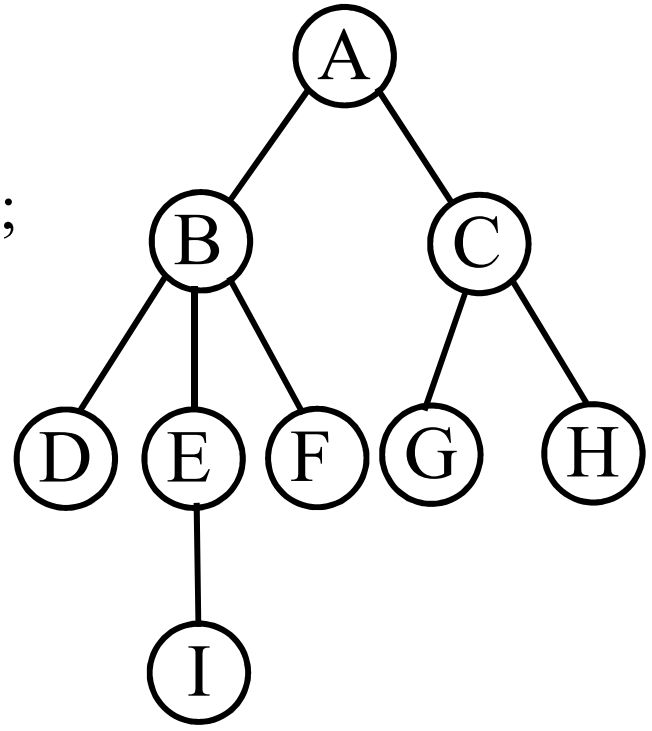
traversal	tree	bt
preorder	ABEFCGKLDHIMJ	ABEFCGKLDHIMJ
inorder	EBFAKGLCHDMIJ	EFBKLGCHMIJDA
postorder	EFBKLGCHMIJDA	FELKGMJIHDCBA

BT implementation

Node:

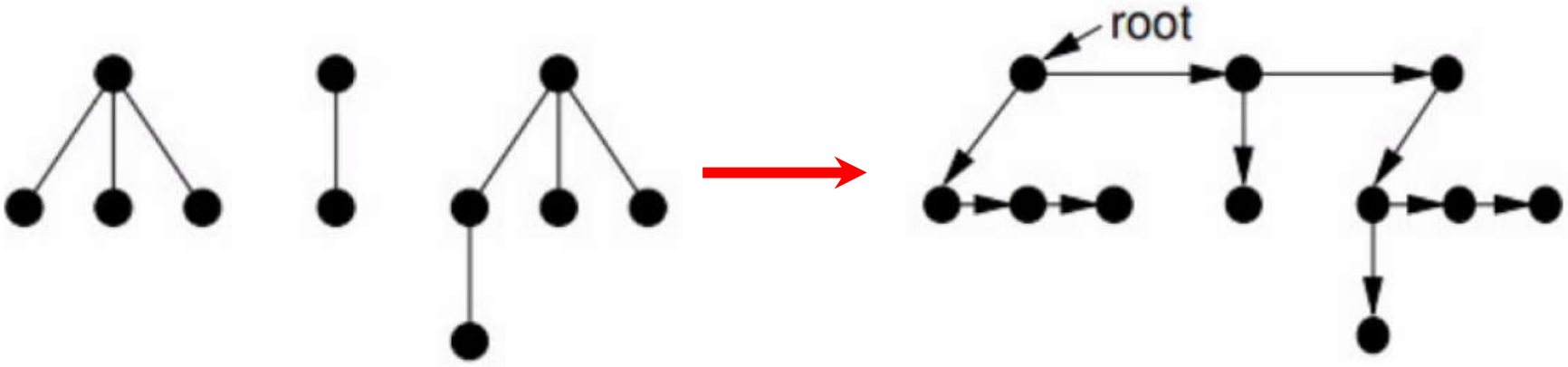
firstchild	data	rightsib

```
struct  CSNode {  //dynamic
        DataType     data;
        CSNode   *firstchild ,  *rightsib ;
} ;
typedef struct  CSNode   *CSTREE ;
```



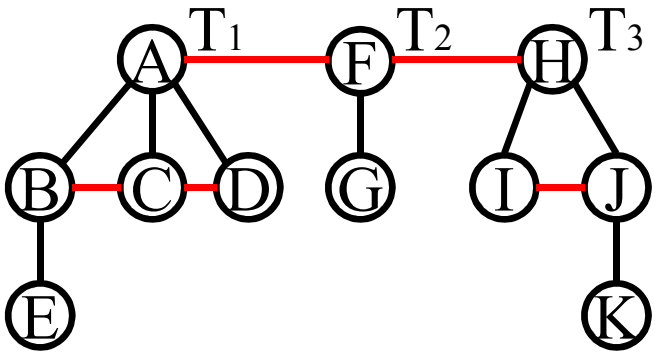
Converting forest to binary tree

- Converting form a forest of general trees to a single binary tree
- Each node stores pointers to its left child and right sibling,
- The tree roots are assumed to be sibling for the purpose of converting.

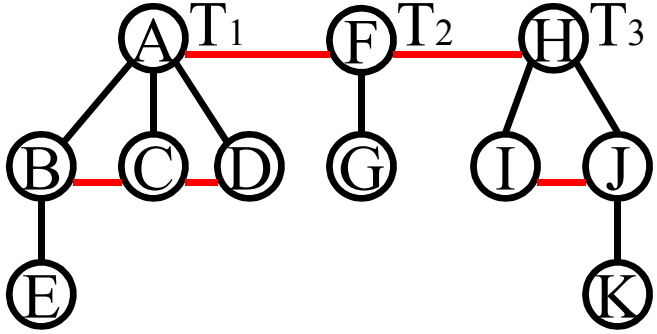


Converting forest to binary tree

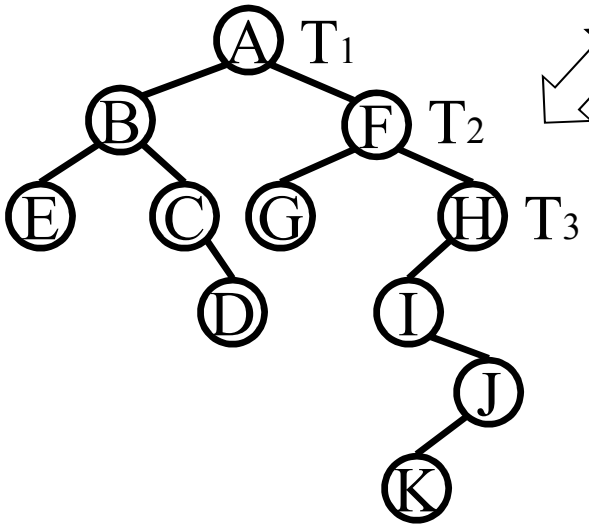
Include links：



Remove links：



rotation：



Algorithm: Transforming a forest into a binary tree

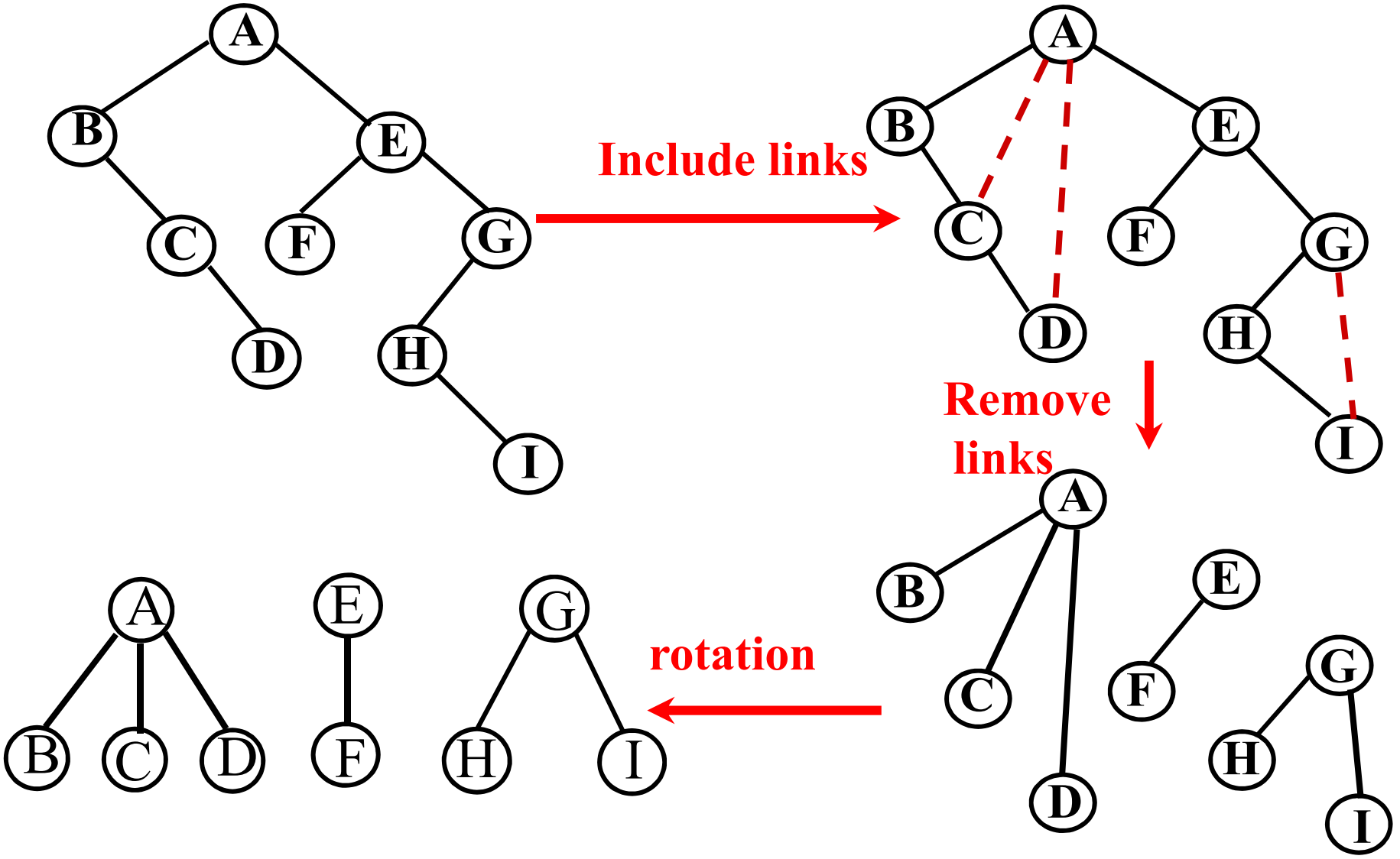
- Definition:
  - A *forest* is a set of $n \geq 0$ disjoint trees

- If $T_1,\ldots,T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1,\ldots,T_n)$;
- B is empty, if $n = 0$
- B has root equal to root(T_1); has left subtree equal to B($T_{11},T_{12},\ldots,T_{1m}$); and has **right subtree equal to $B(T_2,T_3,\ldots,T_n)$**
  - where $T_{11},T_{12},\ldots,T_{1m}$ are the subtrees of root (T_1)

Converting binary trees to forest

- **Include links:**
  - If k is left child of node R， then R to be the parent of k's right child, and
  - Apply the same rule to all right child
- **Remove links：**
  - Remove the link from k to its right child.
  - Apply the same rule to all right child
- **rotation ：**
  - Rotate the tree anticlockwise by 45 degrees
  - This is level order of the nodes

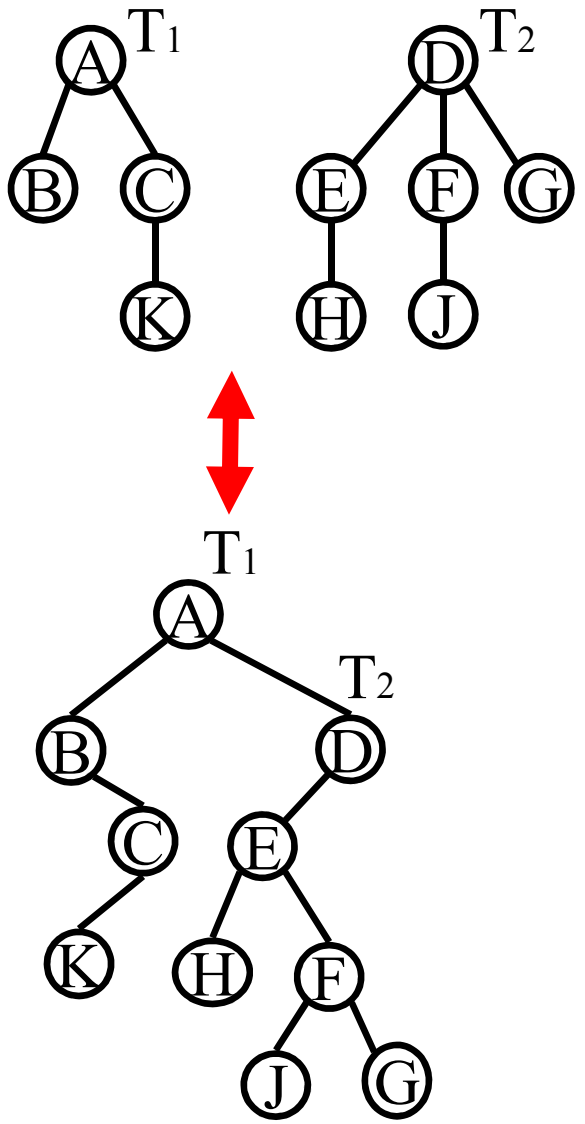
Converting binary trees to forest



Forest traversals

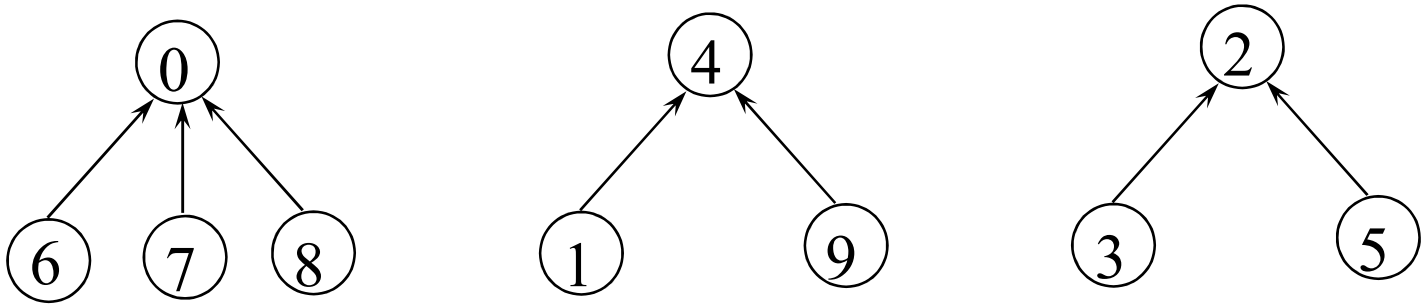
- Forest preorder traversal
  (1)If *F* is empty, then return.
  (2)Visit the root of the first tree of *F*.
  (3)Traverse the subtrees of the first tree in tree preorder.
  (4)Traverse the remaining tree of *F* in preorder.

- Forest postorder traversal
  (1)If *F* is empty, then return
  (2)Traverse the subtrees of the first tree in tree postorder
  (3)Traverse the remaining tree of *F* in postorder
  (4)Visit the root of the first tree of *F*



App: Set Representation

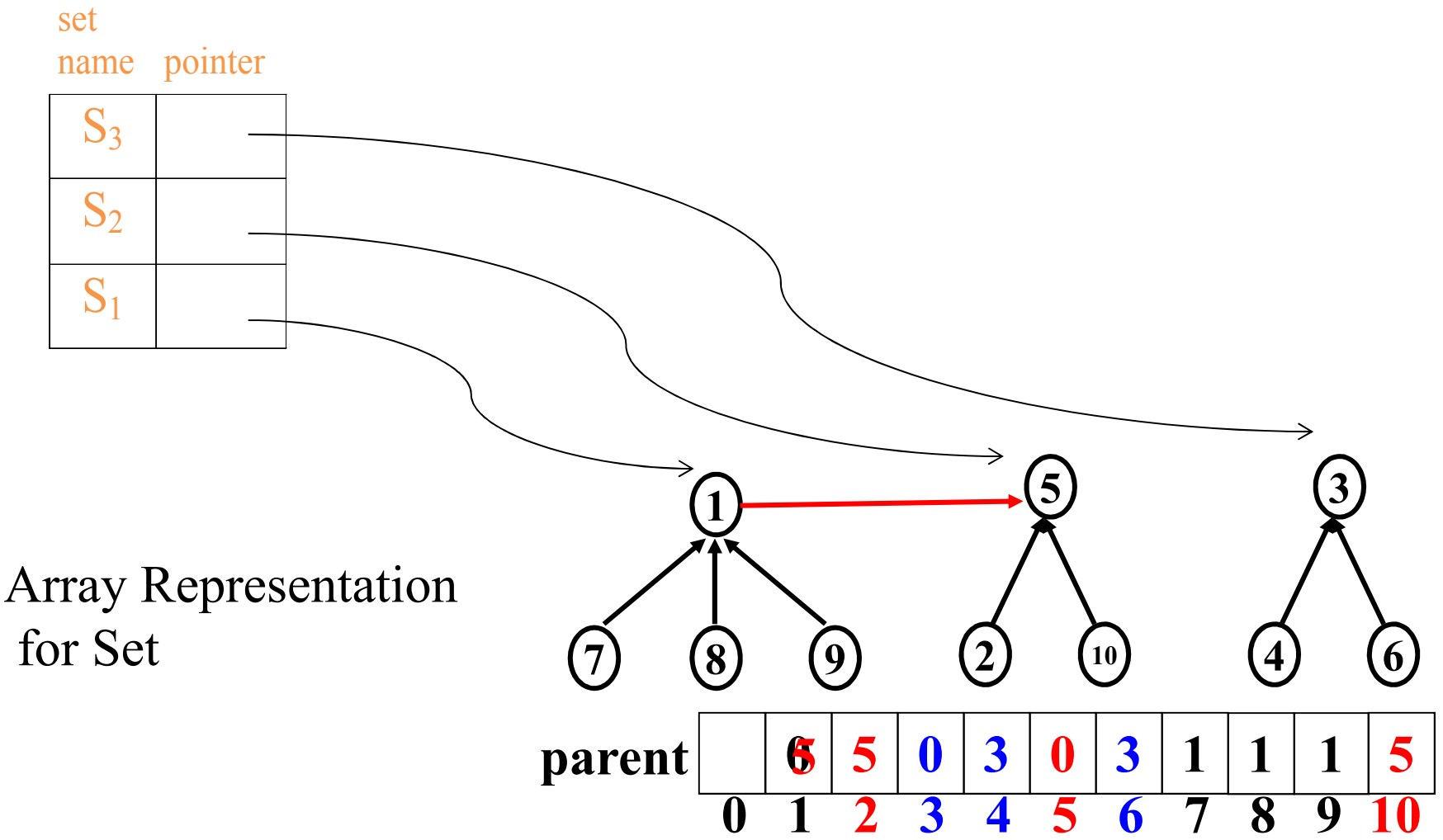
- S1={0, 6, 7, 8}, S2={1, 4, 9}, S3={2, 3, 5}



- Operations considered here
  - *Disjoint set union* $S_1 \cup S_2=\{0,6,7,8,1,4,9\}$
  - *Find*(*i, S*): Find the set containing the element *i*.

    $3 \in S_3$, $8 \in S_1$
  - *Initial(A, x)*

Implementation



Array Representation
for Set

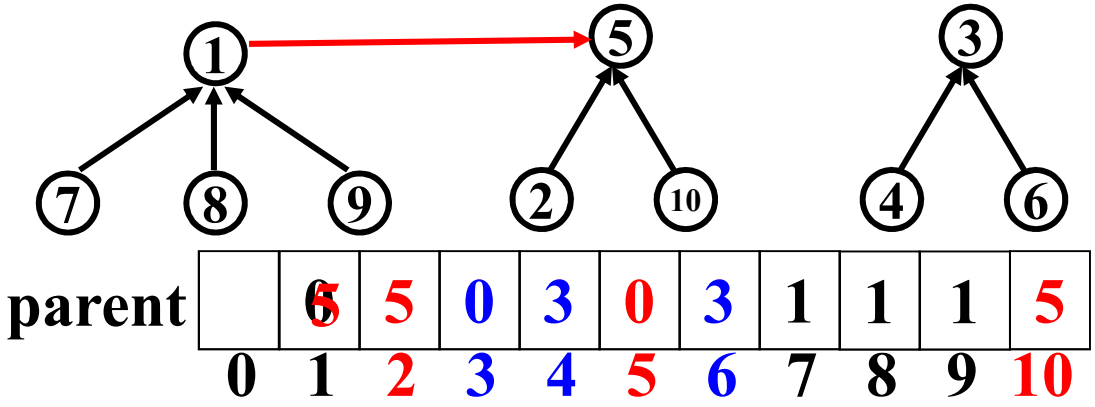
App: Set Representation

- Storage Structure

```
#define n // number of elements
typedef int MFSET[n+1];
/* the type of set is MFSET, the type of element is int */
```

- Basic operations

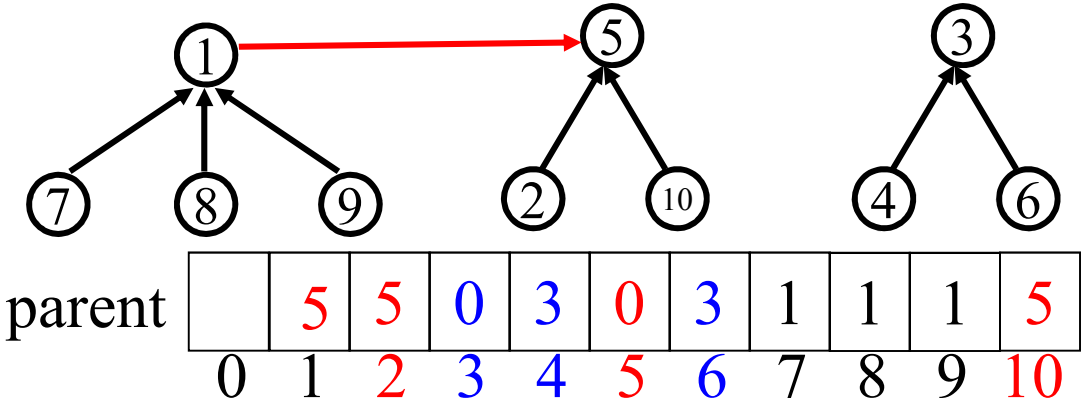
```
void Union(int i, int j ,MFSET parent)
{    parent[i]=j;/* the root of union is j */
}//O(1)
```



App: Set Representation

- Basic operations

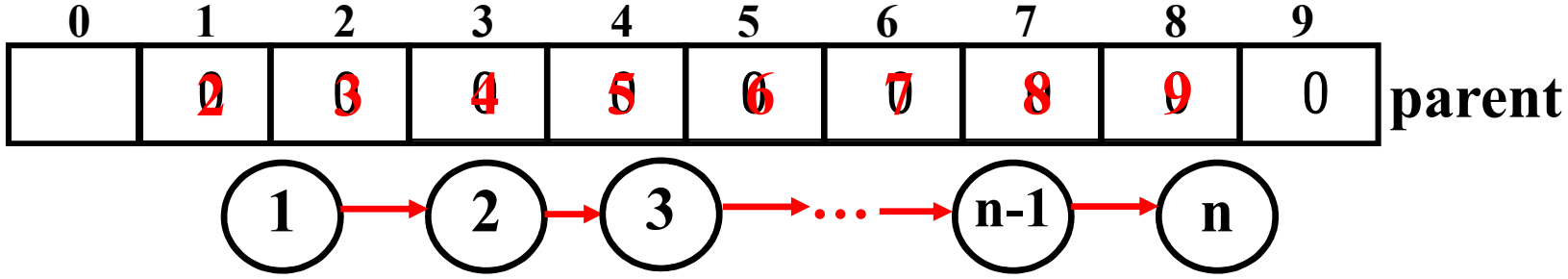
```
void Initial(int x ,MFSET parent)
{    parent[x]=0;
} //O(1)

int Find(int i, MFSET parent)
{   int tmp=i;
    while(parent[tmp]!=0)/* >0, not root */
       tmp=parent[tmp]; /* next upstream */
    return tmp;
}//O(n)
```



App: Set Representation

- Performance analysis：
  - Union(1，2，parent)， Find(1，parent)
  - Union(2，3，parent)， Find(1，parent)
  - Union(3，4，parent)， Find(1，parent)
  - ……
  - Union(n-1,n，parent)，Find(1，parent)

degenerate tree



one Union is O(1)， n-1 times in total， total is O(n);
one Find(1, parent)， if 1 is on i level， Find(1, parent) takes O(i)， n-2 times， $O(\sum i) = O(n^2)$

App: Set Representation

- Improve the implementation
  - Basic idea：

weighting rule for union(i,j): if # of nodes in i < # in j
then j is the parent of i

  - Storage structure

```
typedef struct{
        int father;
        int count; /* weighting */
} MFSET[ n+1 ];
```

App: Set Representation

Improved algorithms

```
void Union(int A,int B,MFSET C)
{   if(C[A].count > C[B].count) { /* |B|<|A| */
        C[B].father = A; /* merge to A */
        C[A].count += C[B].count;
    }
    else {   /*|A|<|B|*/
        C[A].father = B;  /* merge to B */
        C[B].count += C[A].count;
    }
}
```

App: Set Representation

Improved algorithms

```
int Find(int x, MFSET C)
{     int tmp=x;
      while(C[tmp].father!=0)/*>0, not root*/
            tmp=C[tmp].father; /* next up stream */
      return tmp;
}
void Initial(int A ,MFSET C)
{     C[x].father=0;
      C[x].count=1;
}
```

App: Set Representation

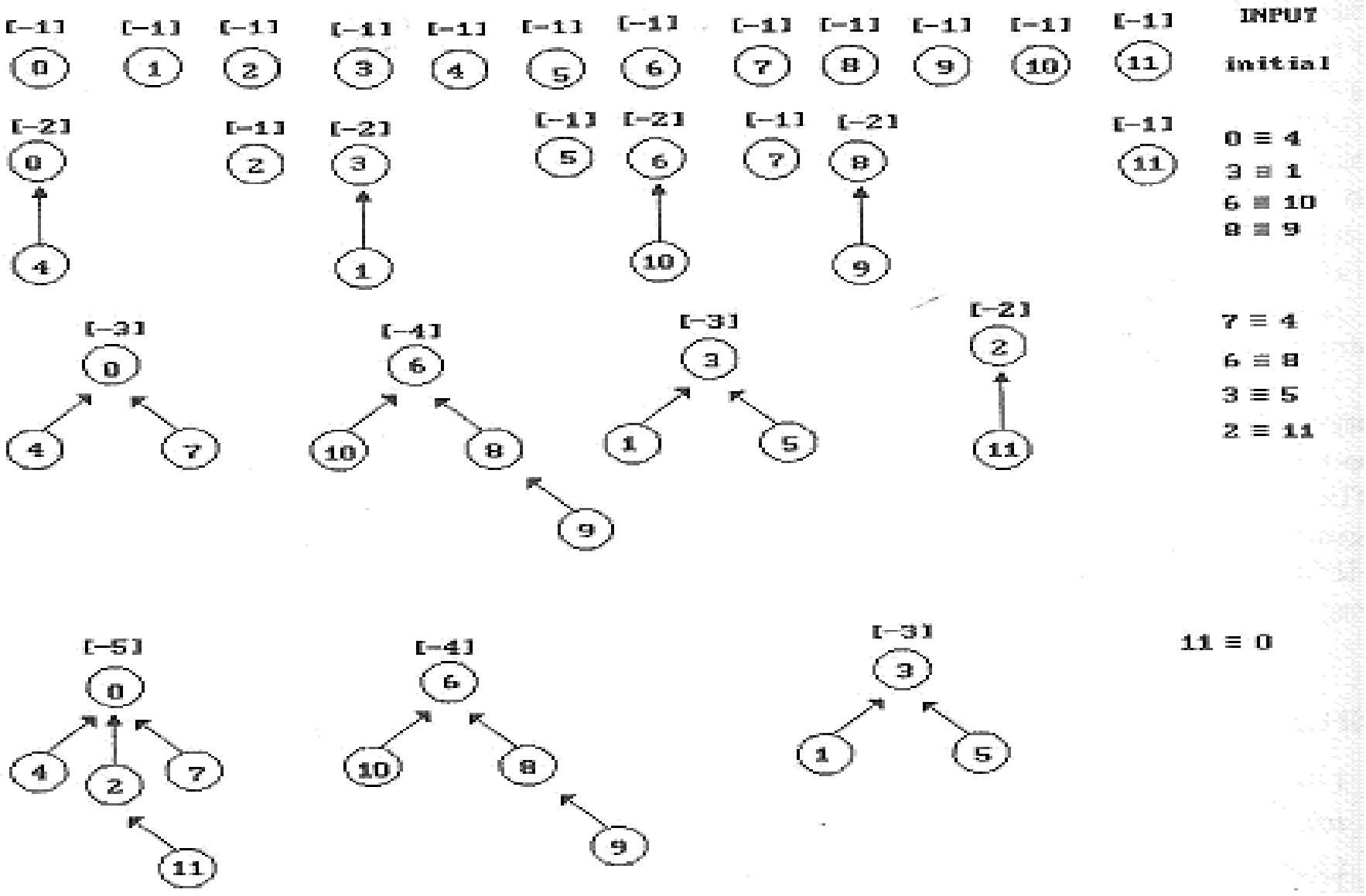
- Equivalence classification of sets
  - Find equivalence classification $i \equiv j$
  - Find S_i and S_j such that $i \in S_i$ and $j \in S_j$ (two finds)
    - $S_i = S_j$ do nothing
    - $S_i \neq S_j$ union(S_i , S_j)
  - example

    $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8,$

    $3 \equiv 5, 2 \equiv 11, 11 \equiv 0$

    $\{0, 2, 4, 7, 11\}, \{1, 3, 5\}, \{6, 8, 9, 10\}$

Example：



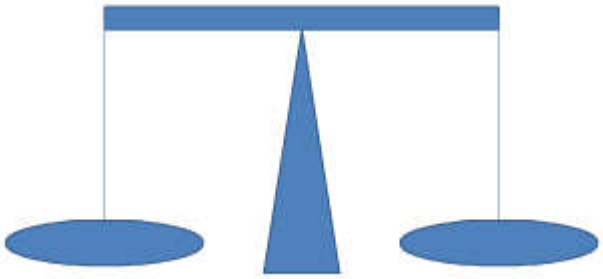
App: Set Representation

Equivalence algorithm

```
void Equivalence (MFSET S)
{     int i ,j , k ,m;
      for(i=1; i<=n+1;i++)
           Initial(i,S);              /*S  only contains i */
      cin>>i>>j;                      /* input the equivalence of i and j */
      while(!(i==0&&j==0){ /* for all equivalences*/
           k=Find(i,S);               /*root of i*/
           m=Find(j,S);               /* root of j*/
           if(k!=m)                 /*if k==m,i,j ,  already in the tree*/
                Union(i,j,S);
          cout<<i<<j;
      }
}
```

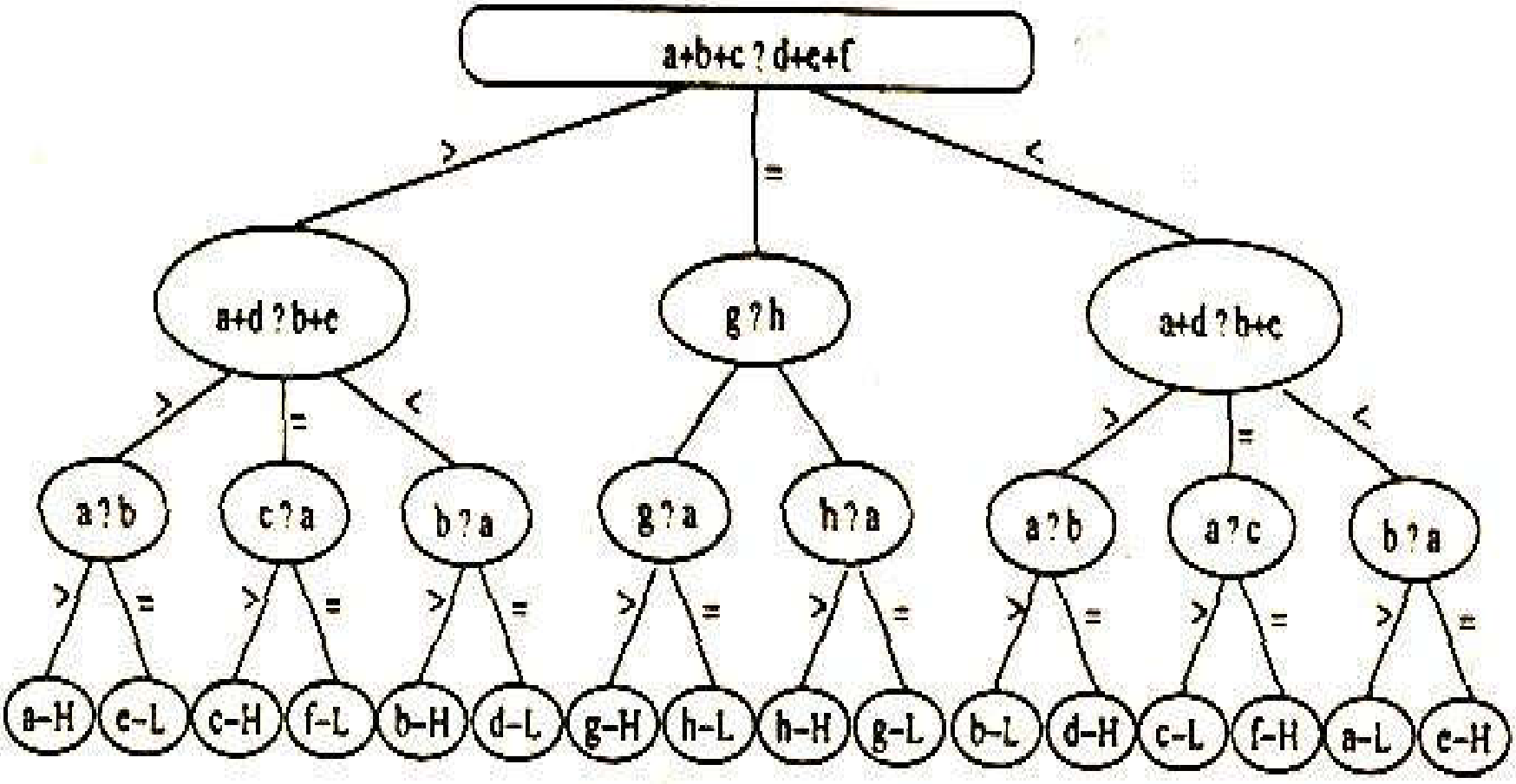
App: Decision tree

- 8-coin problem：
  - You are given 8 coins (a、b、c、d、e、f、g、h), one of them is fake coin, which is lighter or heavier than the others. By using only the balance weighing scale at most ？ times, explain the steps to identify the fake coin



App: Decision tree

- Decision tree for 8-coin problem
- The set of all possible solutions



Huffman Tree

App: Huffman tree: Introduction

- We usually encode strings by assigning **fixed- length codes** to all characters in the alphabet (for example, 8-bit coding in ASCII).
- However, if different characters occur with different frequencies, we can save memory and reduce transmittal time by using **variable length encoding**.
- The idea is to assign shorter codes to characters that occur more often.

App: Huffman tree: Introduction

- Relative frequencies of the letters of the alphabet:

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

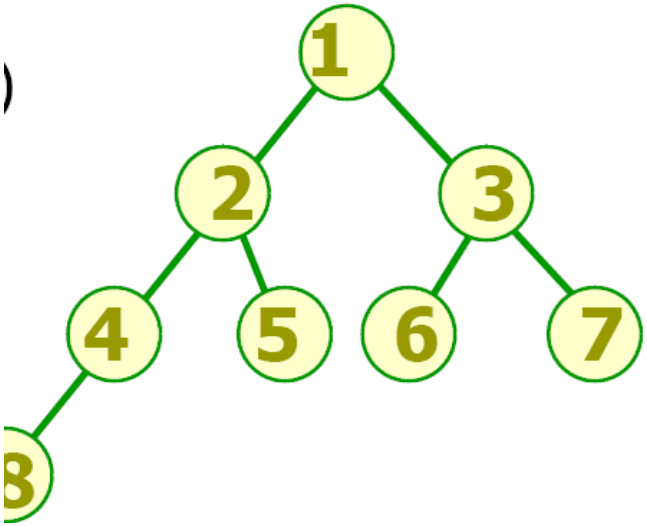
The letter ‘E’appears about 60 times more often than the letter ‘Z.’

App: Huffman tree

Path Length (PL)

- If $n_1, n_2, ..., n_k$ is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a path from n_1 to n_k.
- The length of the path is k -1.

PL = 3*1+2*3 = 9



App: Huffman tree

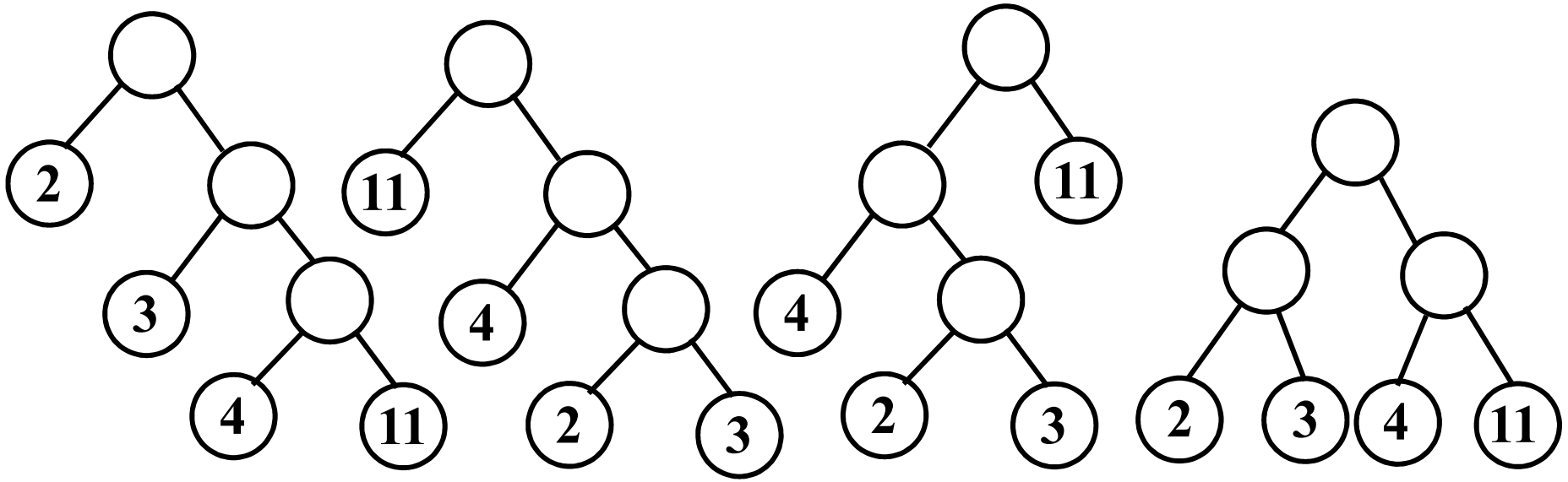
Weighted Path Length

- weighted path length of a leaf is its weight times its depth.
- weighted path length of a tree is the sum of weighted path lengths of every leaf.

$$WPL = \sum_{k=1}^{n} w_k * PL_k$$

- **Huffman tree has the minimum WPL**

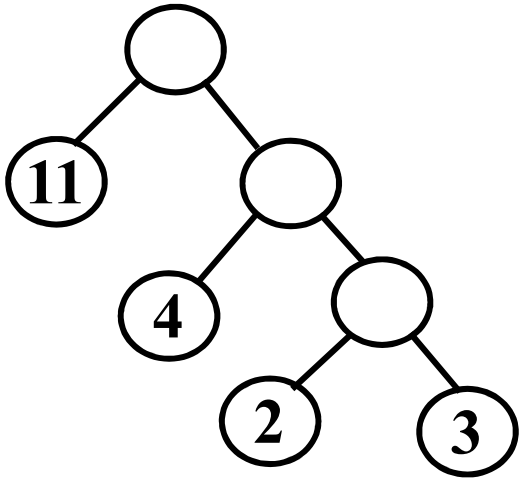
Examples:

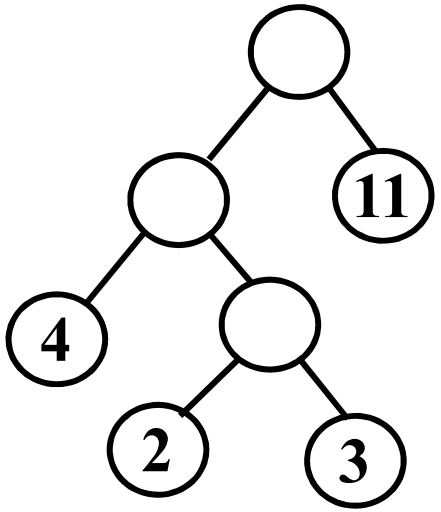


WPL=53 **WPL=34** **WPL=34** **WPL=40**

App: Huffman tree

- Characters of Huffman tree：
  - The two letters with least frequency are stored in siblings nodes, whose depth is at least as deep as any other leaf nodes in the tree.
  - Nodes with 0 and 2 degree, without nodes with 1 degree.
  - If there are *n* leaf nodes, then the number of all nodes is *2n-1*
  - The value of WPL is minimum





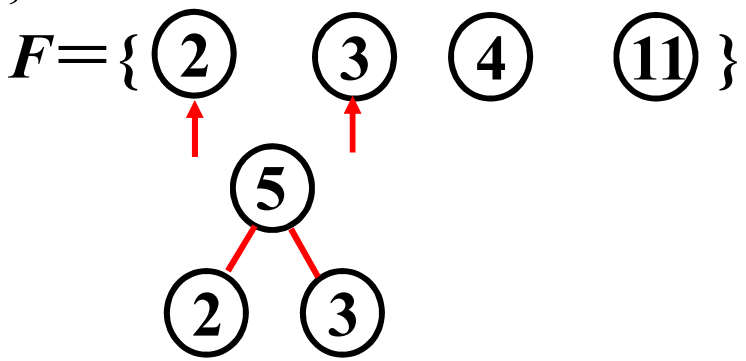
App: Huffman tree

How to build Human Trees

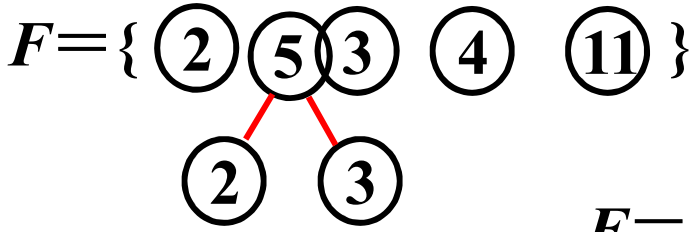
- Create a collection of *n* initial Huffman trees
  - Each of them is a single leaf node containing one of the letters.
  - Put the *n* partial trees onto a priority queue organized by weight (frequency), the one with lowest weight in front.
- Remove the first two trees (ones with lowest weight) from the priority queue.
  - Join the two trees together to create a new tree whose root has the two trees as children, and
  - whose weight is the sum of the weights of the two trees.
  - Put this new tree back into the priority queue.
- This process is repeated until all of the partial Huffman trees have been combined into one.

App: Huffman tree

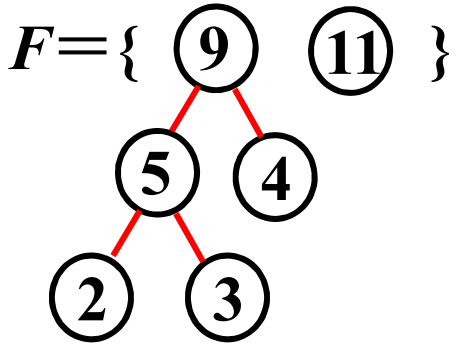
- Demo： $W=\{2, 3, 4, 11\}$
  - initialization：
  - Select and union：

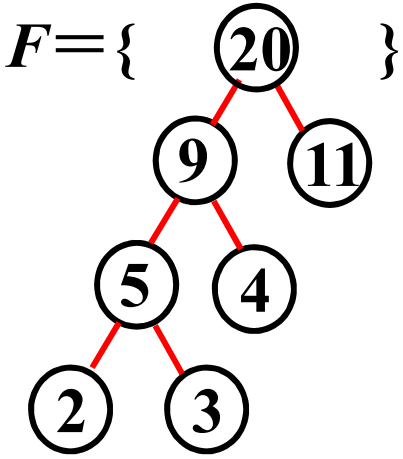


  - Remove and join:



  - repeat:

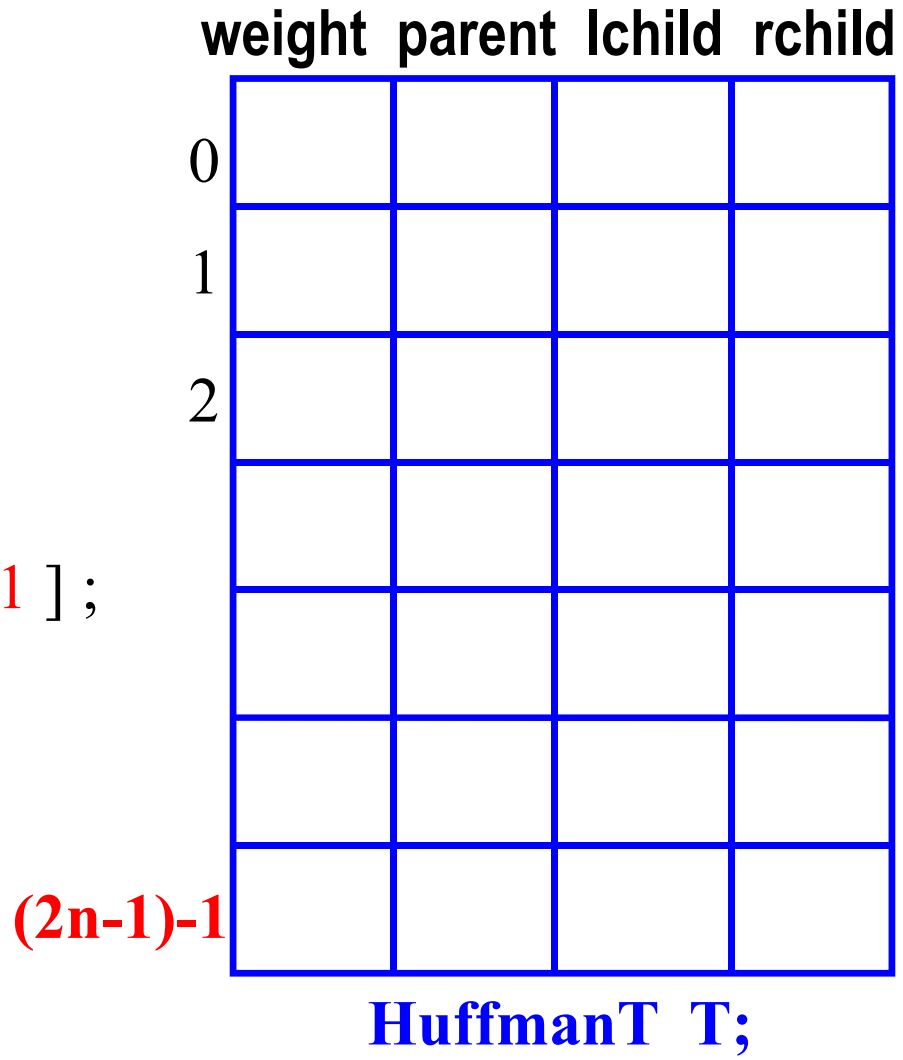




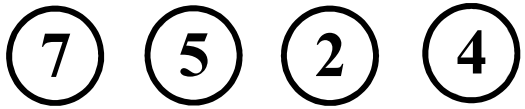
App: Huffman tree

- Storage structure----static ternary linked list

```
typedef  struct { //
      double  weight ;
       int    lchild ;
       int    rchild ;
       int    parent;
}HTNODE ;
typedef   HTNODE  HuffmanT[ 2n-1 ] ;
```



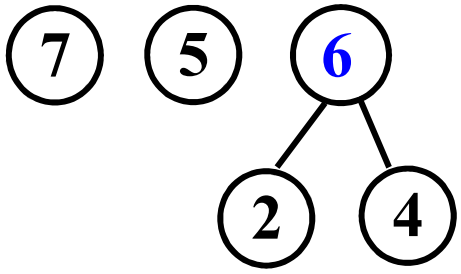
App: Huffman tree: Demo



	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	-1	-1	-1
3	4	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

initialization

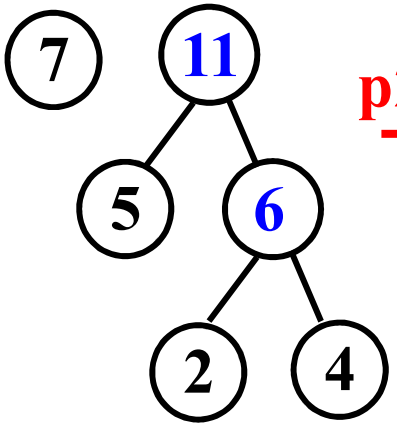
App: Huffman tree



	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
p1 → 2	2	4 ~~-1~~	-1	-1
p2 → 3	4	4 ~~-1~~	-1	-1
i → 4	6	-1	2 ~~-1~~	3 ~~-1~~
5		-1	-1	-1
6		-1	-1	-1

process

App: Huffman tree



p1 →

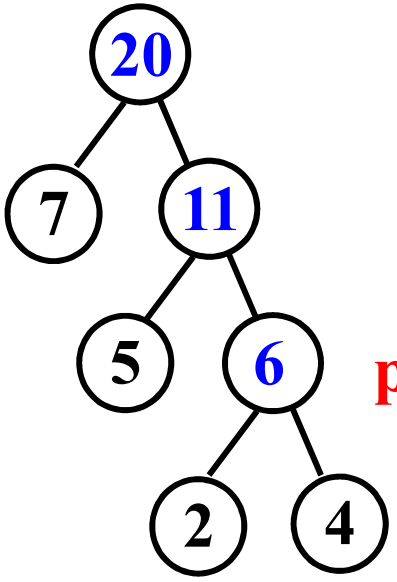
p2 →

i →

	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	5 ~~-1~~	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5 ~~-1~~	2	3
5	11	-1	1 ~~-1~~	4 ~~-1~~
6		-1	-1	-1

process

App: Huffman tree



	weight	parent	lchild	rchild
p1 → 0	7	6 ~~-1~~	-1	-1
1	5	5	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5	2	3
p2 → 5	11	6 ~~-1~~	1	4
i → 6	20	-1	0 ~~-1~~	5 ~~-1~~

App: Huffman tree

- Algorithm

```
void  CreatHT(HuffmanT T)//T[2n-2] is the root
{  int i ,p1 ,p2;
   InitHT(T);                                  //1.initialization
   InputW(T);                                  //2. input weight
   for（i = n; i < 2n-1; i++）{                //3. n-1 times union
      SelectMin(T, i-1, &p1, &p2);
      T[p1].parent =T[p2].parent = i;
      T[i].lchild= p1；
      T[i].rchild= p2；
      T[i].weight =T[p1].weight + T[p2].weight；
    }
}
```

Huffman encode

Data compression

Q1. Given a text that uses 32 symbols (26 different letters, space and some punctuation characters), how can we encode this text in bits?

A. We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called a **fixed length encoding**.

Q2. Some characters (e, t, a, o) are used more often than others, how can we use this to reduce our encoding?

A. Encode these characters with fewer bits, and the others with more bits.

Q3. How do we know when the next symbol begins?

A. Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that **no** code is a **prefix** of another one.

ex: c(a) = 01, c(b) = 010, c(c) = 1 => 0101 is? aa

Prefix codes

Definition. A **prefix code** for a set S is a function *c* that maps each $x \in S$ to 1s and 0s in such a way that for $x, y \in S, x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Ex. c(a)=11, c(e)=01, c(k)=001, c(l)=10, c(u)=000

Q. What is the meaning of 1001000001?

A. "leuk".

Q. Suppose frequencies are know in a text of 1G:
f_a=0.4, f_e=0.2, f_k=0.2, f_l=0.1, f_u=0.1

Q. What is the original size of the encoded text?

A. $2*f_a + 2*f_e + 3*f_k + 2*f_l + 3*f_u = 2.4G$

Optimal prefix codes

Definition: The Average Bits per Letter (ABL) of a prefix code c is the sum over

all symbols of its frequency times the number of bits of its encoding

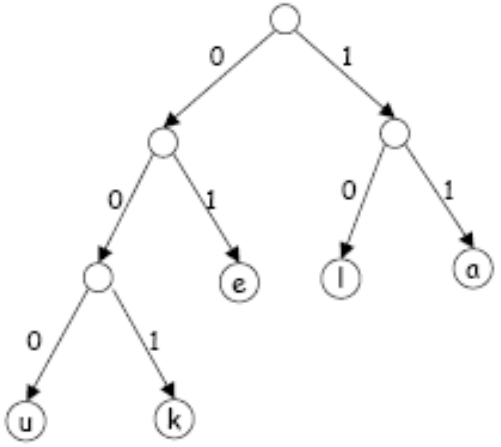
$$ABL(c) = \sum_{x \in S} f_x . |c(x)|$$

We would like to find a prefix code that has the **lowest** possible ABL.

Suppose we model a code in a binary tree…

Representing prefix codes using binary trees

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Q. How does the tree of a prefix code look like?

A. Only the leaves have a label.

Proof. An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.

Representing prefix codes using binary trees

Q. What is the meaning of

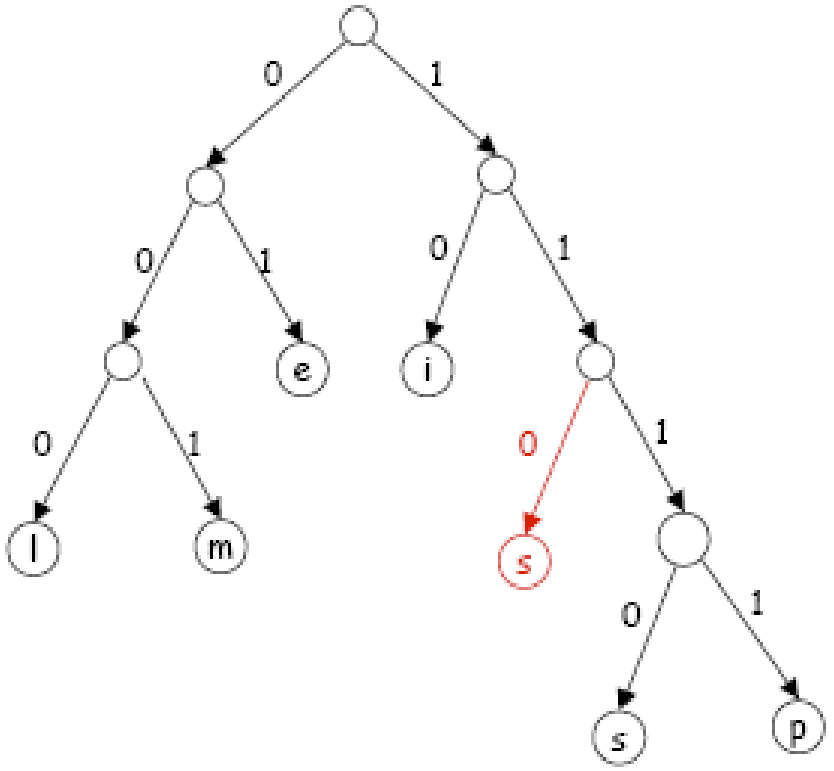
11101000111111010**00**?

$$ABL(T) = \sum_{x \in S} f_x . | depth_r(x) |$$

A. “simpel”.

Q. How can this prefix be more efficient?

A. Change encoding of *p* and *s* to a shorter one. This tree is not full.

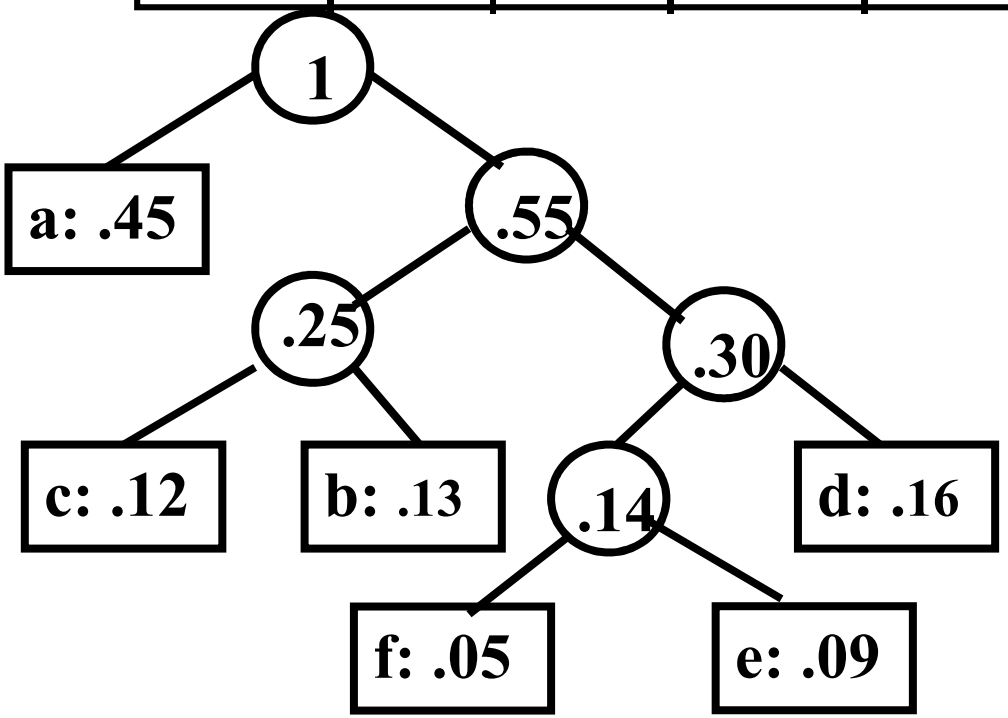


App: Huffman tree

Letter	a	b	c	d	e	f	avr.
P	0.45	0.13	0.12	0.16	0.09	0.05	
fix	000	001	010	011	100	101	3
unfix	0	101	100	111	1101	1100	2.24

$= \lceil \log_2 |C| \rceil$

$= \sum p_j \cdot l_j$



	ch	bits
0	a	0
1	b	101
2	c	100
3	d	111
4	e	1101
5	f	1100
6		

encode table

App: Huffman tree

- Storage structure of encode table

```
typedef struct{
   char  ch;   //
   char  bits[n+1]; //string of bits
}CodeNode;
typedef CodeNode HuffmanCode[n];
HuffmanCode    H;
```

	ch	bits
0	a	0 \0
1	b	1 0 1 \0
2	c	1 0 0 \0
3	d	1 1 1 \0
4	e	1 1 0 1 \0
5	f	1 1 0 0 \0
6		

- Huffman tree algorithm is based on Huffman algorithm
- Set 0 to left child, and 1 to right child
- Huffman code of a leaf code is the sequences of {0,1} on the path.

App: Huffman tree

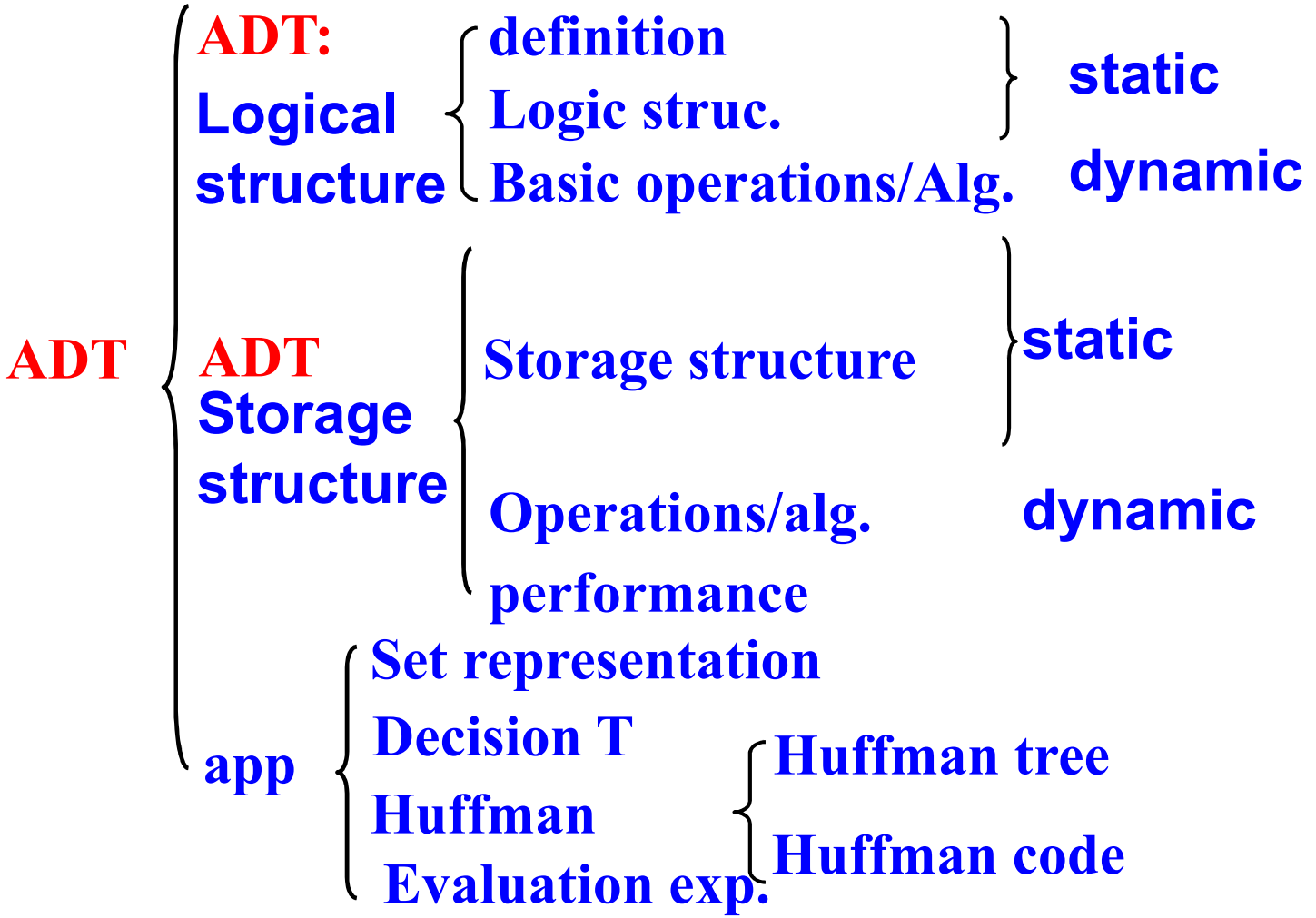
- Algorithm

```
void CharSetHuffmanEncoding( HuffmanT T, HuffmanCode H)
{  /*based on Huffman T, get the Huffman encode table H*/
   int c, p, i;          /* c and p are pointers  */
   char cd[n+1];    /* temp for keep the encode */
   int start;
   cd[n]='\0';
   for( i =0; i <n;  i++){  /* get the T[i] of leaf node */
      H[i].ch=getchar();  /* get T[i] character*/
      start=n;                  /* */
      c =i;                      /* upstream from leaf of T[i] */
      while( (p=T[c].parent)>=0){ /* T[c] is the root */
         cd[--start]=(T[p].lchild==c)? '0' : '1';
         /* T[c] is the left child of T[p] ，encode by 0，otherwise 1*/
         c=p;      /* go on*/
      }
      strcpy(H[i].bits,&cd[start]);  /*copy onto table*/
   }
}
```

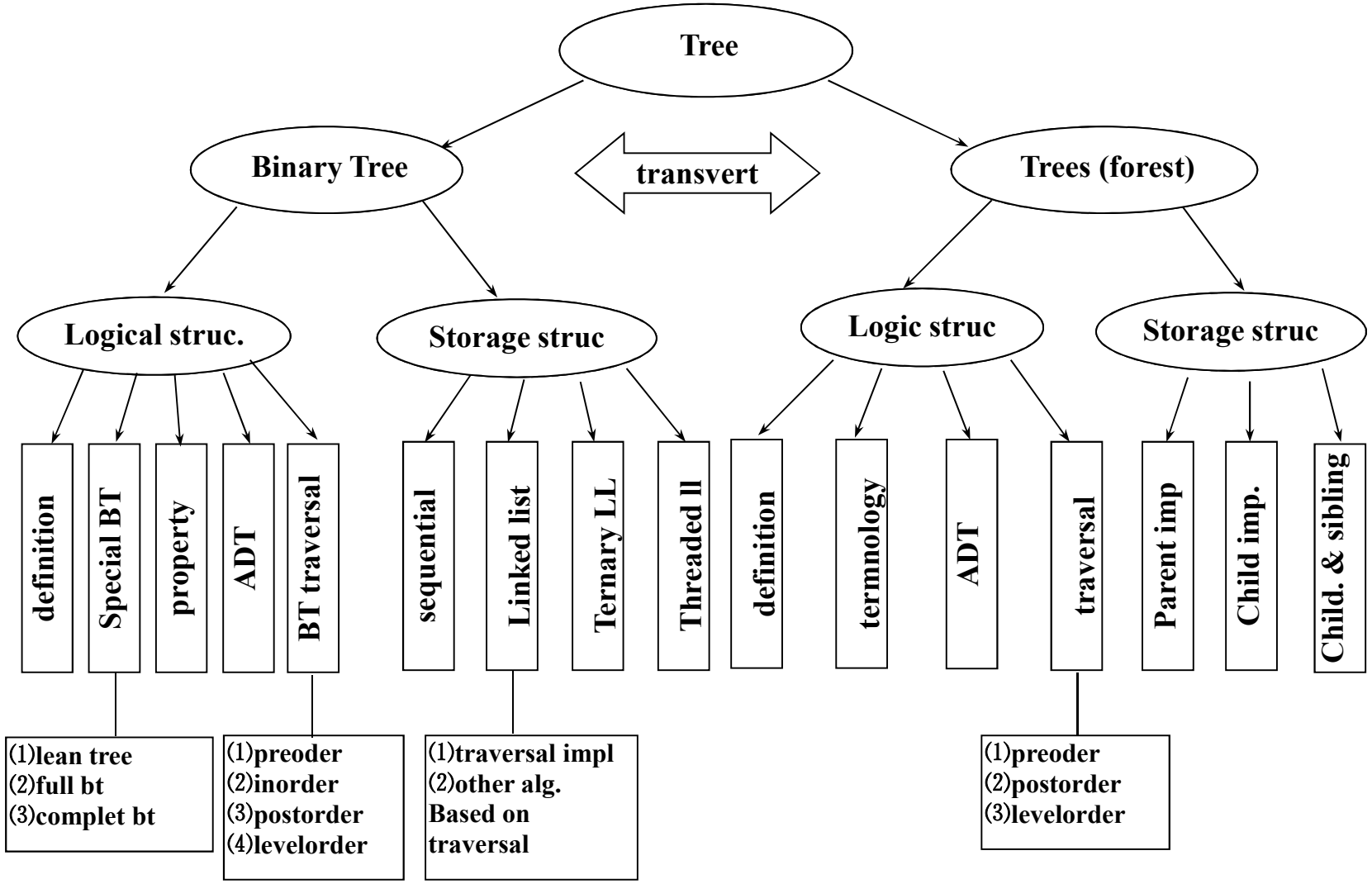
	ch	bits
0	a	0 \0
1	b	1 0 1 \0
2	c	1 0 0 \0
3	d	1 1 1 \0
4	e	1 1 0 1 \0
5	f	1 1 0 0 \0
6		

Summary

- concepts：BT、T



Summary


Tree
Binary Tree
transvert
Trees (forest)
Logical struc.
Storage struc
Logic struc
Storage struc
definition
Special BT
property
ADT
BT traversal
sequential
Linked list
Ternary LL
Threaded ll
definition
termnology
ADT
traversal
Parent imp
Child imp.
Child. & sibling
(1)lean tree
(2)full bt
(3)complet bt
(1)preoder
(2)inorder
(3)postorder
(4)levelorder
(1)traversal impl
(2)other alg.
Based on
traversal
(1)preoder
(2)postorder
(3)levelorder