UNIT-4

Prepared by:

Manjula L, Assistant Professor

Dept. of CSE, MSRIT

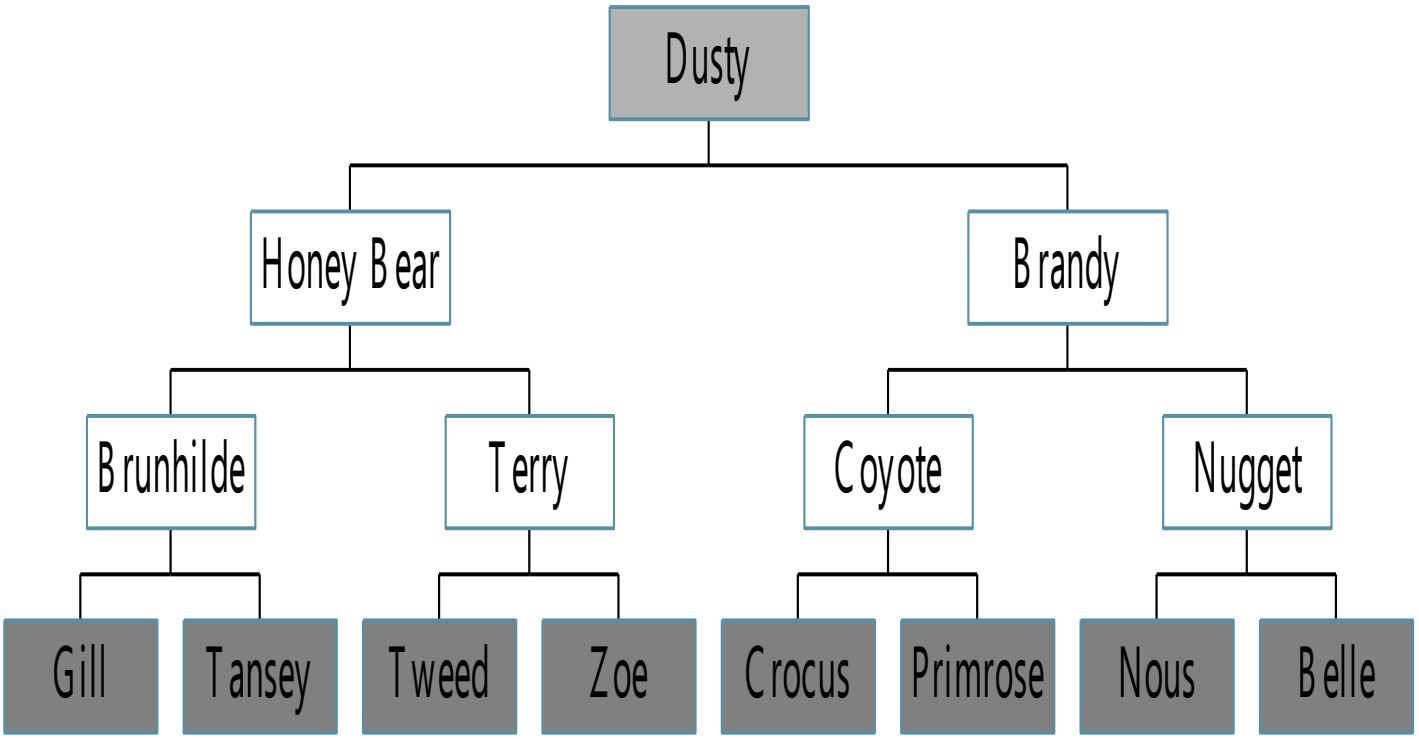
Text Book: Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2008.

Contents

Trees: Introduction

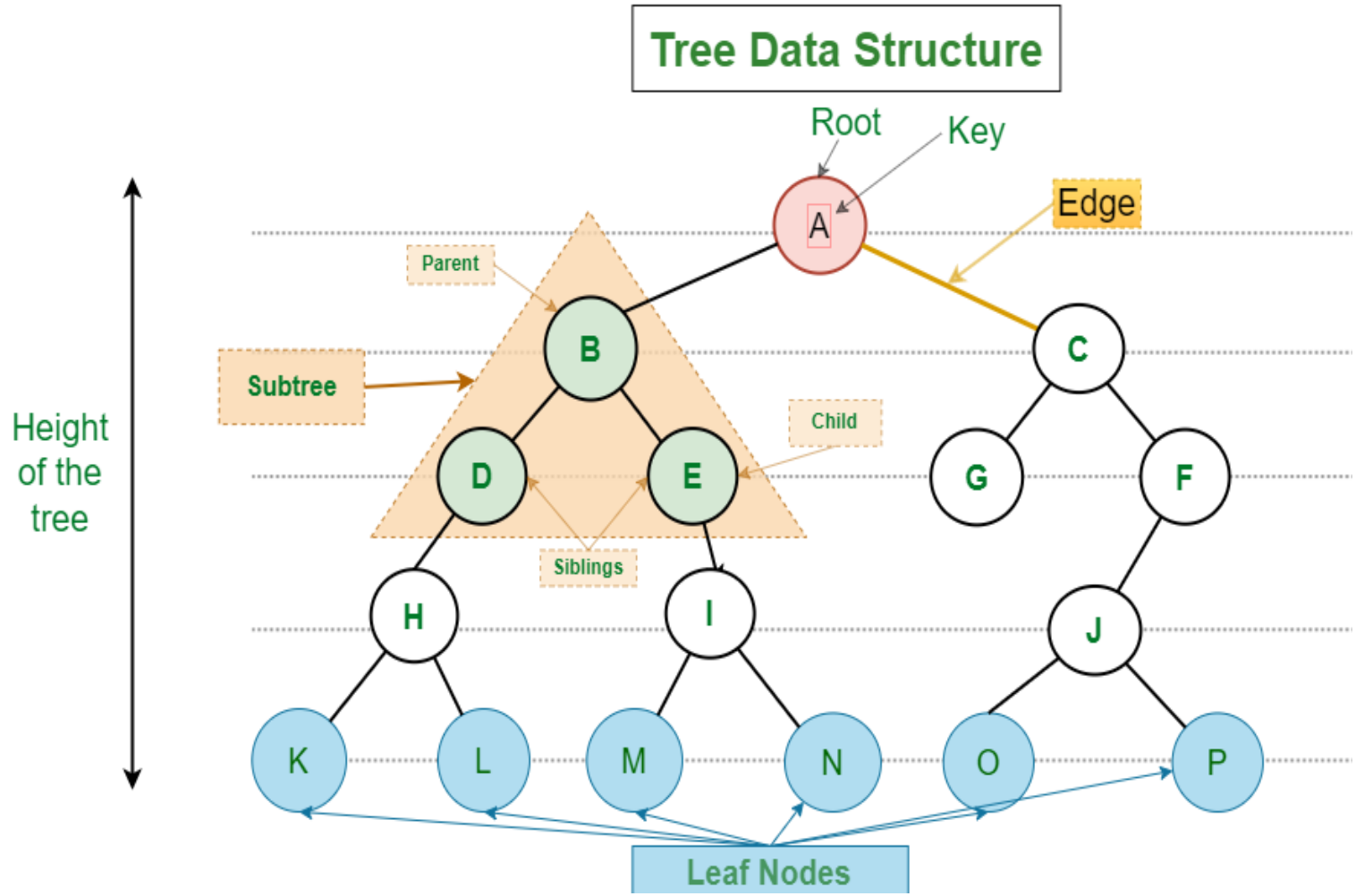
- *T**ree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.*
- *The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.*

Tree Example



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

- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree.
- We call T1, ..., Tn the subtrees of the root.


Tree Data Structure
Root
Key
Edge
A
Parent
B
C
Subtree
Child
D
E
G
F
Siblings
H
I
J
Height of the tree
K
L
M
N
O
P
Leaf Nodes

Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P, G}** are the leaf nodes of the tree.

Basic Terminologies In Tree Data Structure:

- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** Any successor node on the path from the leaf node to that node. **{E,I}** are the descendants of the node **{B}.**
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 1.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

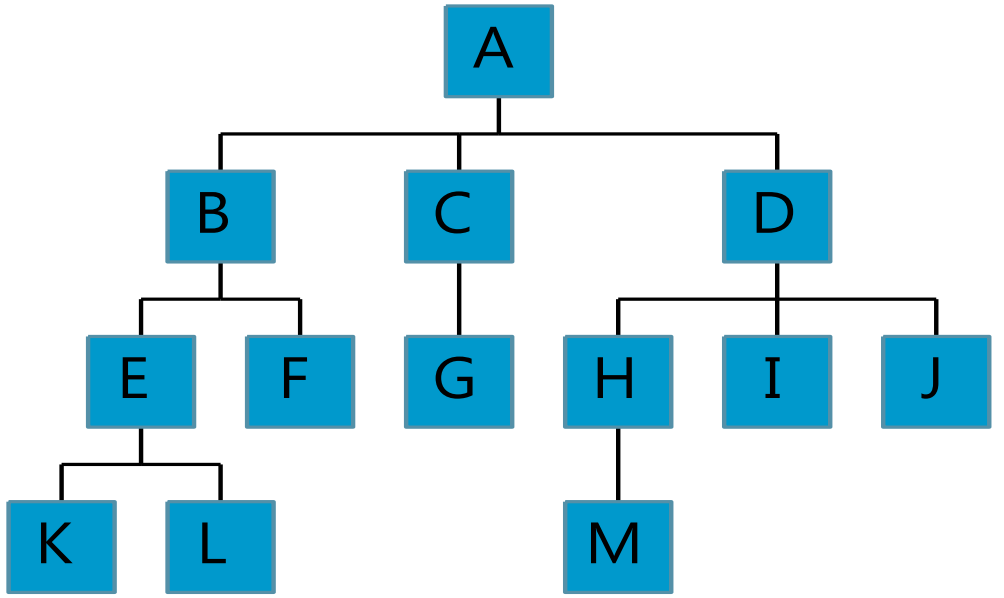
Representation of Trees

1. List Representation
2. Left Child right sibling representation
3. Representation of degree 2 trees

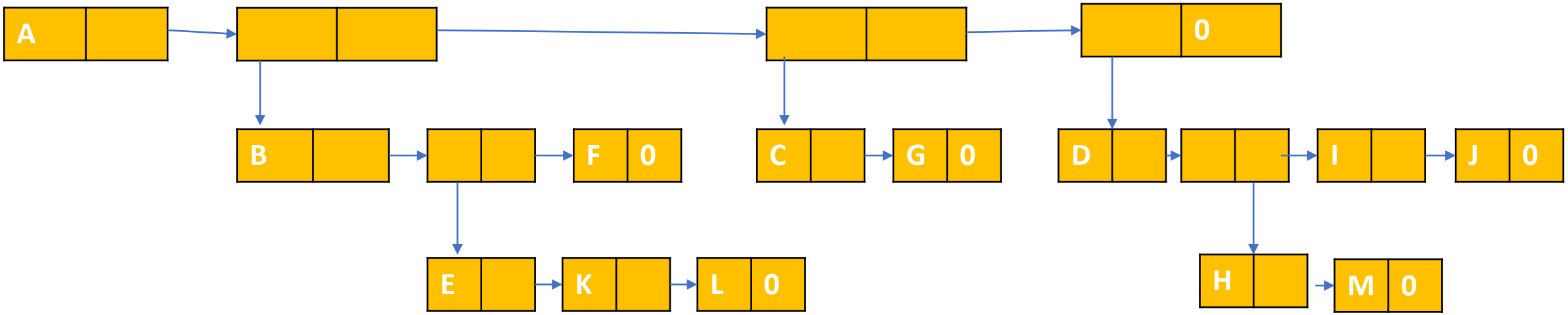
List Representation

List Representation

- (A (B (E (K, L), F), C (G), D (H (M), I, J)))
- The root comes first, followed by a list of sub-trees

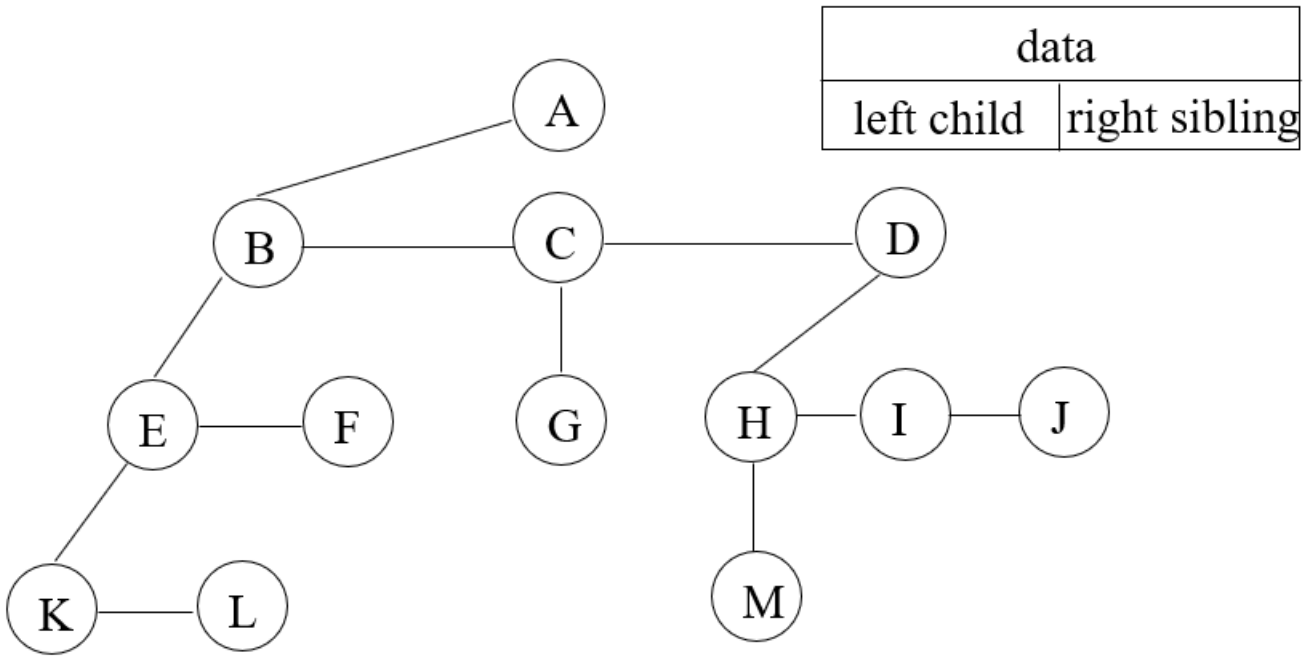


List Representation



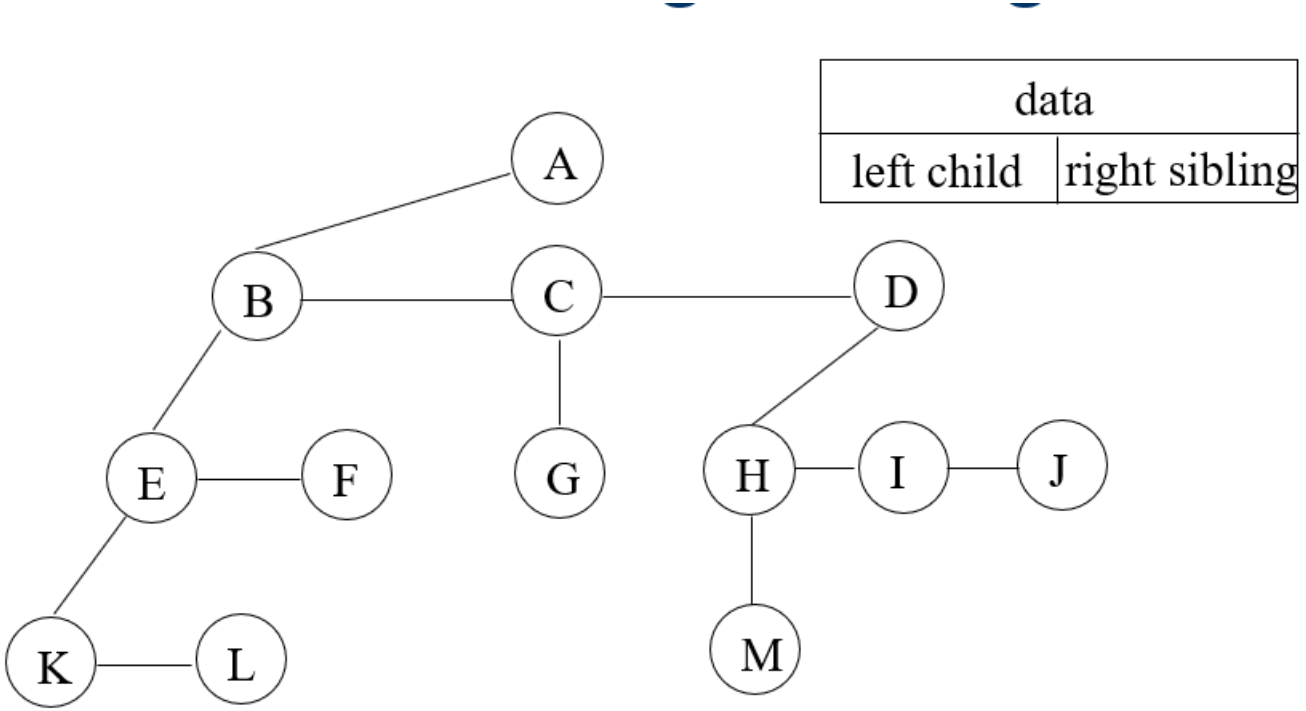
This node structure consumes more memory

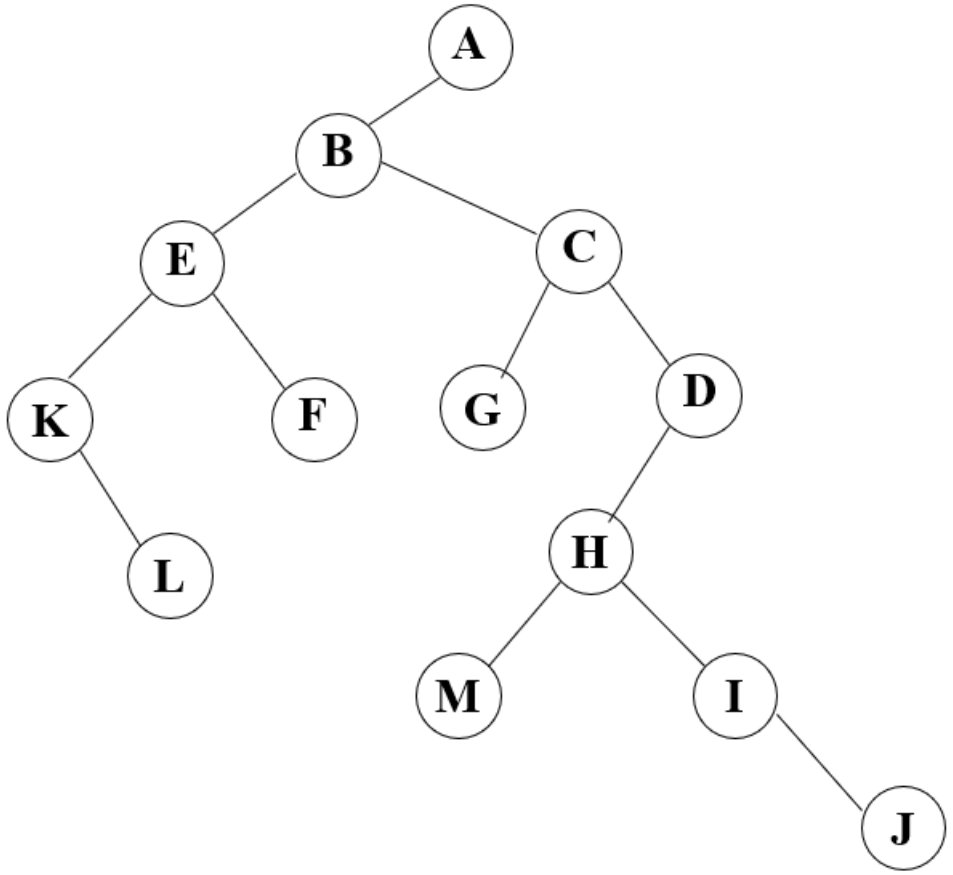
Left Child - Right Sibling representation



Representation of degree- two

- To obtain the degree- two representation of a tree, rotate the right sibling pointers in a left child-right siblings tree clockwise by 45 degrees.





Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.
- A Binary tree is represented by a pointer to the topmost node (commonly known as the “root”) of the tree. If the tree is empty, then the value of the root is NULL.
- Each node of a Binary Tree contains the following parts:
  - Data
  - Pointer to left child
  - Pointer to right child

ADT Binary_tree(abbreviation BinTree) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

Functions:

for all bt, bt1, bt2 $\in$ BinTree, item $\in$ element

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(bt)::= if (bt==empty binarytree) return TRUE else return FALSE

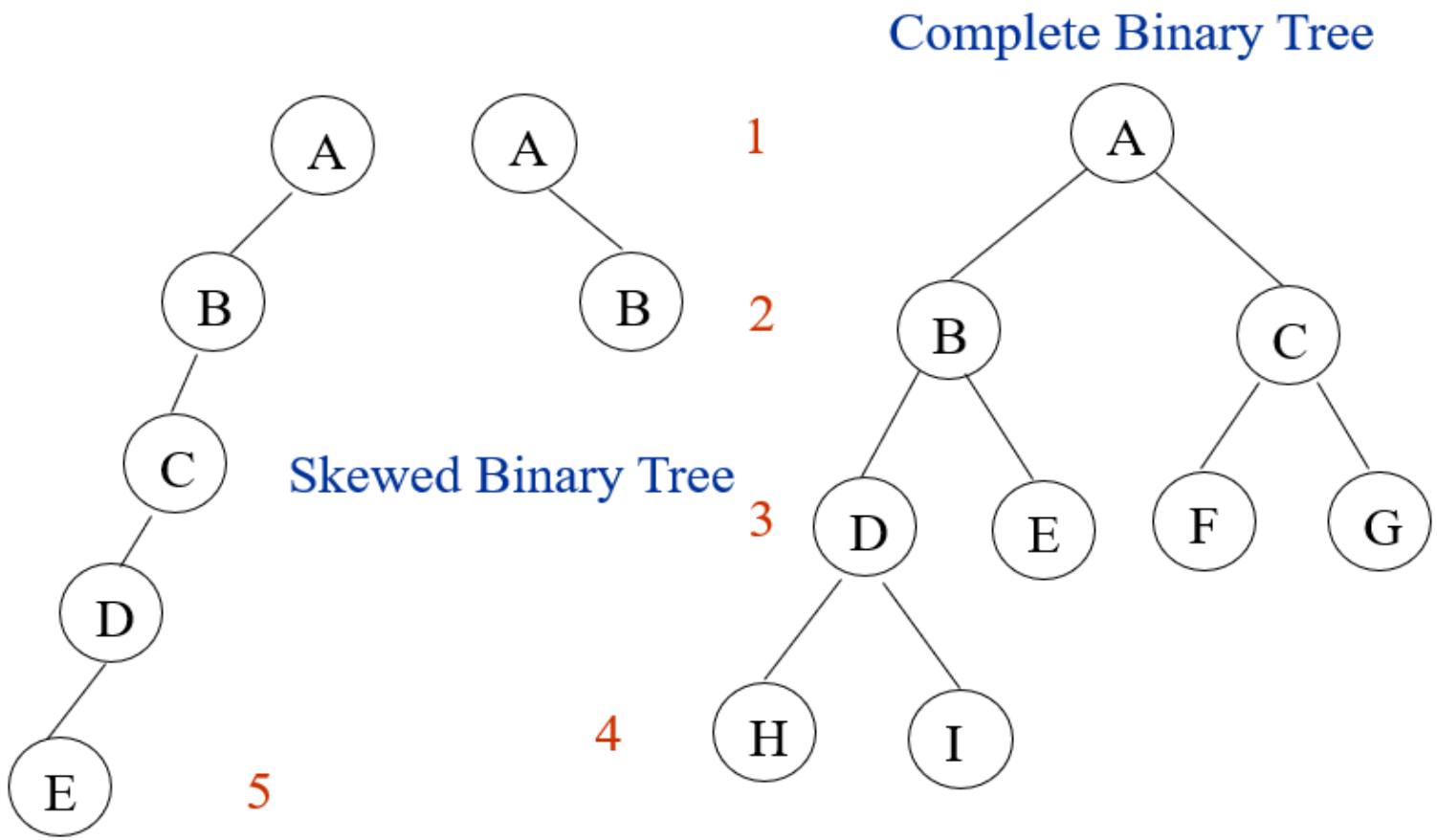
BinTree MakeBT(bt1, item, bt2)::= return a binary tree whose left subtree is bt1, whose right subtree is bt2, and whose root node contains the data item

Bintree Lchild(bt)::= if (IsEmpty(bt)) return error else
return the left subtree of bt

element Data(bt)::= if (IsEmpty(bt)) return error
else return the data in the root node of bt

Bintree Rchild(bt)::= if (IsEmpty(bt)) return error
else return the right subtree of bt

Samples of Trees



Properties of Binary Trees

1. Maximum Number of Nodes in BT
   - The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
   - The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

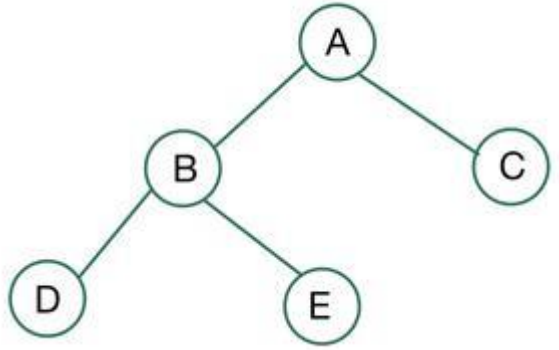
Prove by induction. $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$

2. Relation between number of leaf nodes and degree-2 nodes:

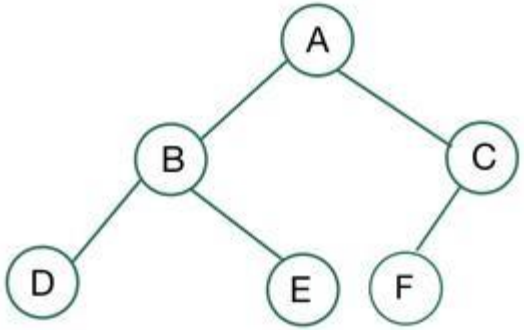
For any nonempty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0=n2+1

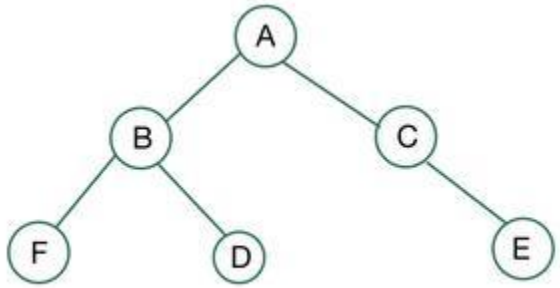
Difference between Full and Complete Binary Tree

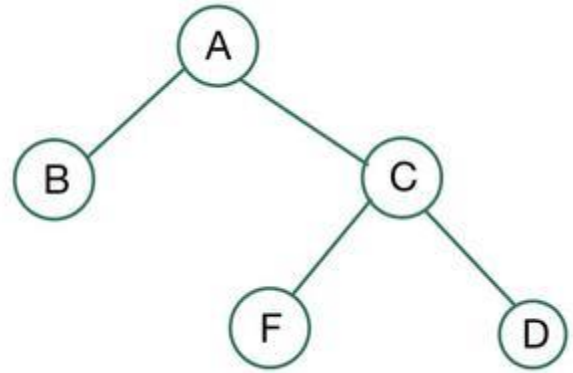
- A full binary tree is a binary tree in which all of the nodes have either 0 or 2 offspring. In other terms, a full binary tree is a binary tree in which all nodes, except the leaf nodes, have two offspring.

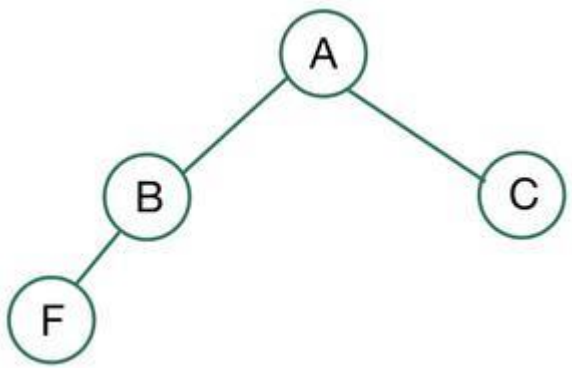


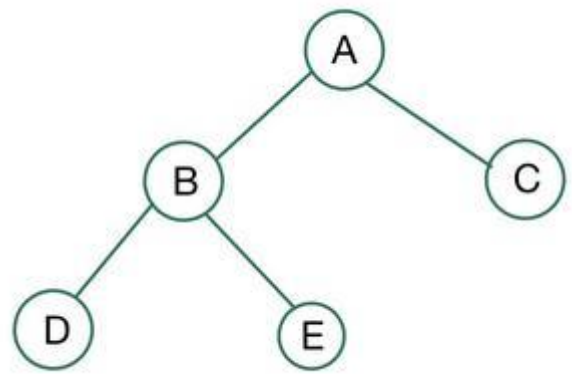
- A binary tree is said to be a complete binary tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible.




A
B
C
F
D
E


A
B
C
F
D


A
B
C
F


A
B
C
D
E

Binary tree Representation

1. Array
2. Linked List

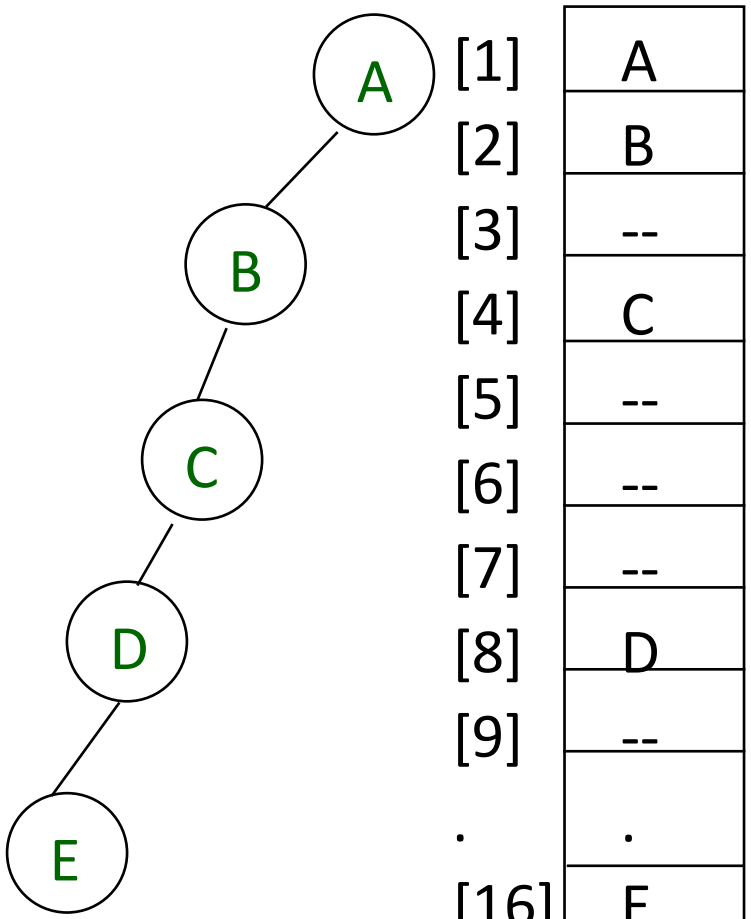
Array

If a complete binary tree with n nodes (depth = log n + 1) is represented sequentially, then for any node with index i, 1<=i<=n, we have:

- parent(i) is at i/2 if i!=1. If i=1, i is at the root and has no parent.
- left_child(i) ia at 2i if 2i<=n. If 2i>n, then i has no left child.
- right_child(i) ia at 2i+1 if 2i +1 <=n. If 2i +1 >n, then i has no right child.

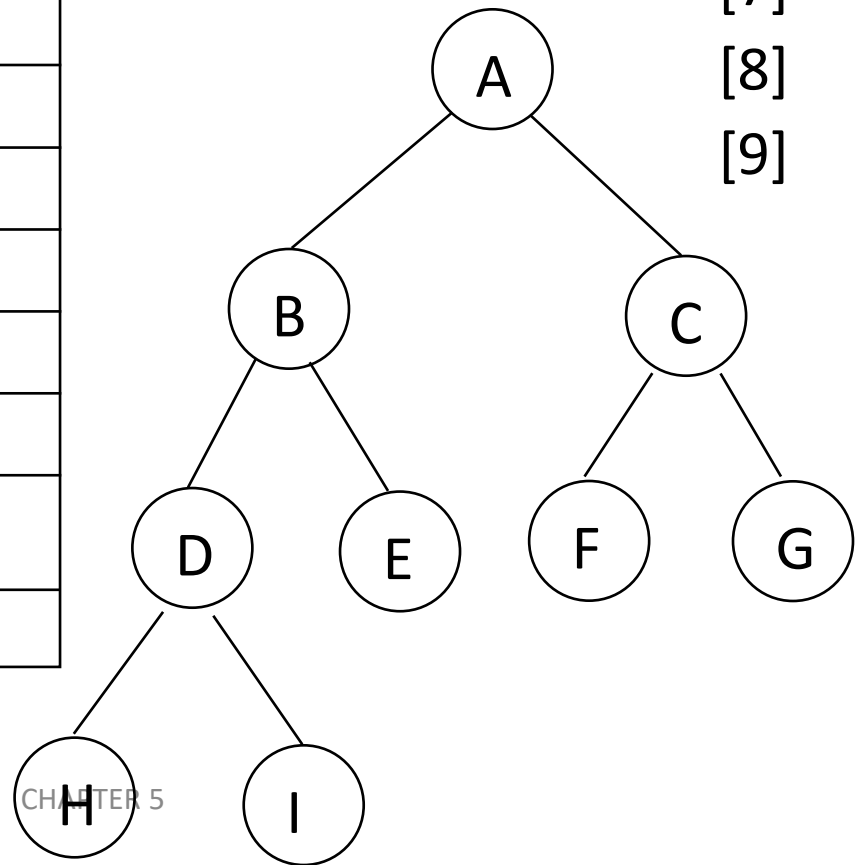
Parent is i/2
Left Child is 2*i+1
Right Chid is 2*i+2

Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

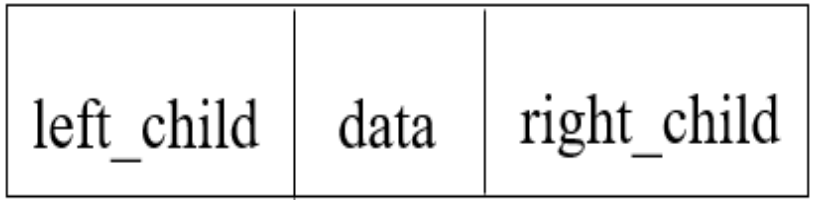
(1) waste space
(2) insertion/deletion problem

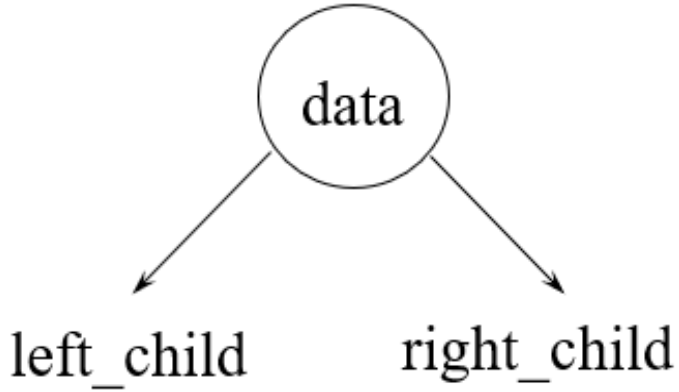


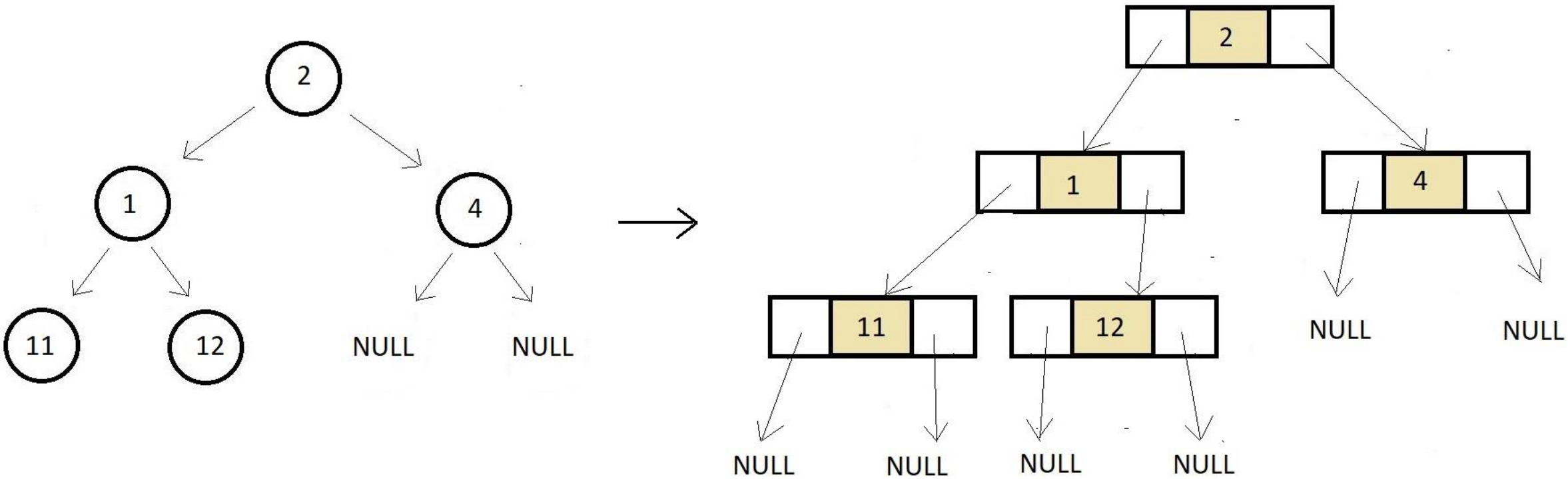
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked List

```
typedef struct node *tree_pointer;
typedef struct node
{
        int data;
        tree_pointer left_child, right_child;
};
```





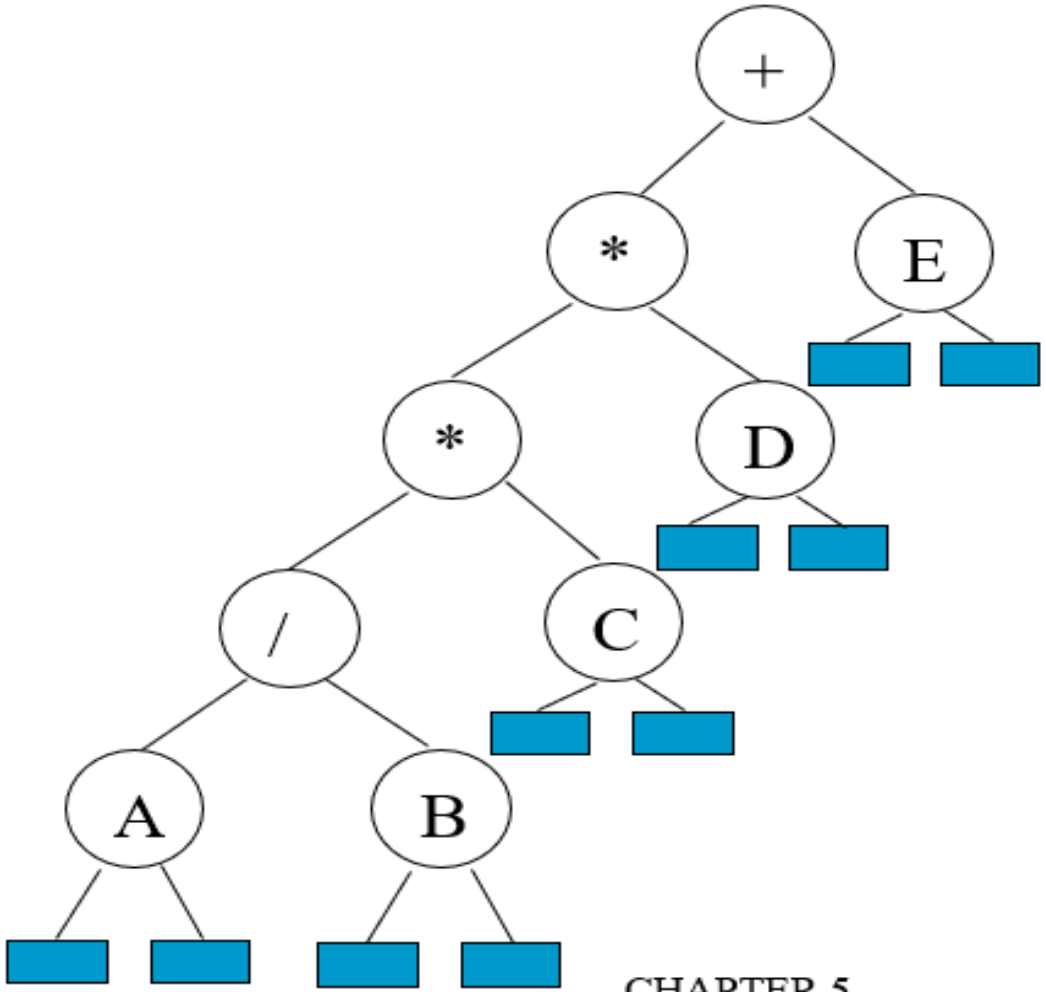

2
1
4
11
12
NULL
NULL
2
1
4
11
12
NULL
NULL
NULL
NULL
NULL
NULL

Binary Tree Traversals

Tree traversal (also known as tree search and walking the tree) is a form of graph traversal and refers to the process of visiting (e.g. retrieving, updating, or deleting) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain LVR, LRV, VLR
- inorder, postorder, preorder

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

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
        if (ptr) {
                inorder(ptr->left_child);
                printf("%d", ptr->data);
                indorder(ptr->right_child);
                }
}
```

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
        if (ptr) {
                printf(“%d”, ptr->data);
                preorder(ptr->left_child);
                predorder(ptr->right_child);
        }
}
```

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
        if (ptr) {
                postorder(ptr->left_child);
                postdorder(ptr->right_child);
                printf("%d", ptr->data);
        }
}
```

Inorder Traversal – Iterative Method

1. Create an empty stack (say **S**).
2. Initialize the **current** node as **root**.
3. Push the **current** node to **S** and set current = current->left until current is NULL
4. If **current** is NULL and the stack is not empty then:
   Pop the top item from the stack.
   Print the popped item and set current = popped_item->right
   Go to step 3.
5. If **current** is NULL and the stack is empty then we are done

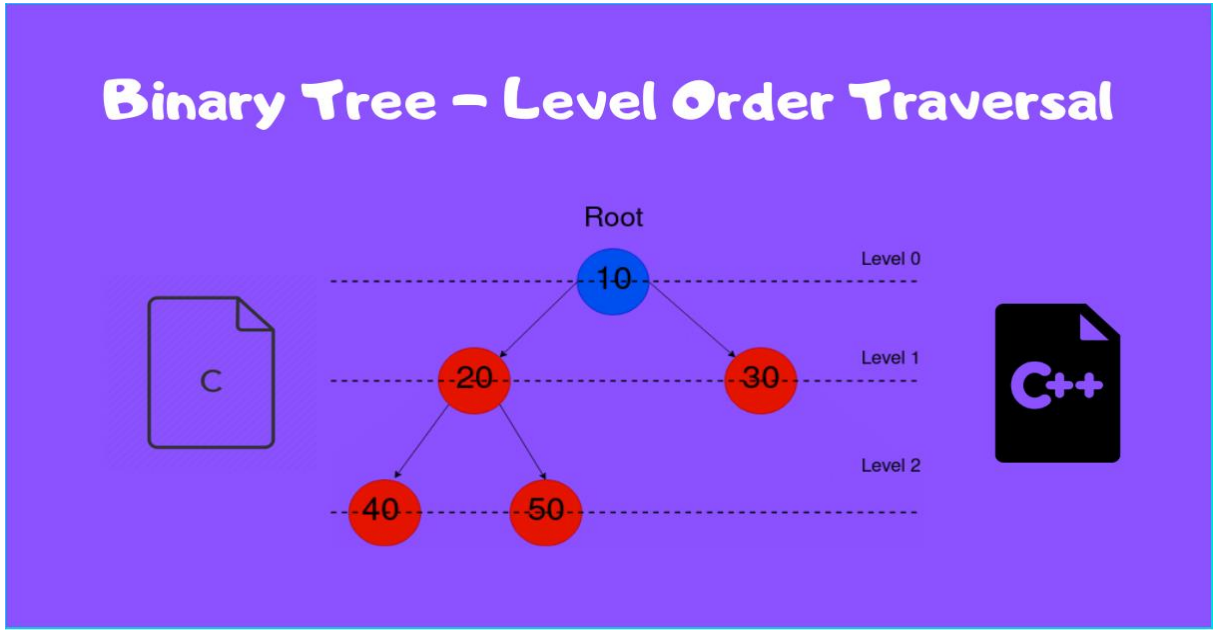
```
void iter_inorder (tree_pointer node)
{
        int top= -1; /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for (;;) {
                for (; node; node=node->left_child)
                        push(node);/* add to stack */
                node= pop(); /* delete from stack */
        if (!node) break; /* empty stack */
                printf("%d", node->data);
        node = node->right_child;
        }
}
```

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

Level order traversal

Level order traversal is actually a Breadth First Search, which is not recursive by nature. Breadth-first search can be used to solve many problems in graph theory, for example: Finding all nodes within one connected component.



```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
        int front = rear = 0;
        tree_pointer queue[MAX_QUEUE_SIZE];
        if (!ptr)
                return; /* empty queue */
        addq(ptr);
         for (;;) {
                ptr = deleteq();
```

```
        if (ptr)
        {
                printf("%d", ptr->data);
                if (ptr->left_child)
                        addq(ptr->left_child);
                if (ptr->right_child)
                        addq(ptr->right_child);
        }
        else
                break;
  }
}
```

Copying Binary Trees

```
tree_poointer copy(tree_pointer original)
{
        tree_pointer temp;
        if (original) {
                MALLOC(temp,sizeof(*temp));
                temp->left_child=copy(original->left_child);
                temp->right_child=copy(original->right_child);
                temp->data=original->data;
                return temp;
                }
        return NULL;
}
```

Equality of Binary Trees

```
int equal(tree_pointer first, tree_pointer second)
{
/* function returns FALSE if the binary trees first and second are not equal,
   otherwise it returns TRUE */


    return ((!first && !second) || (first && second &&(first->data == second->data
   && equal(first->left_child, second->left_child) &&
    equal(first->right_child, second->right_child)))
}
```

Satisfiability Problem

Variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to true.

Is there an assignment to make an expression true?

Ex: x1 ∧ (x2∨ x3)

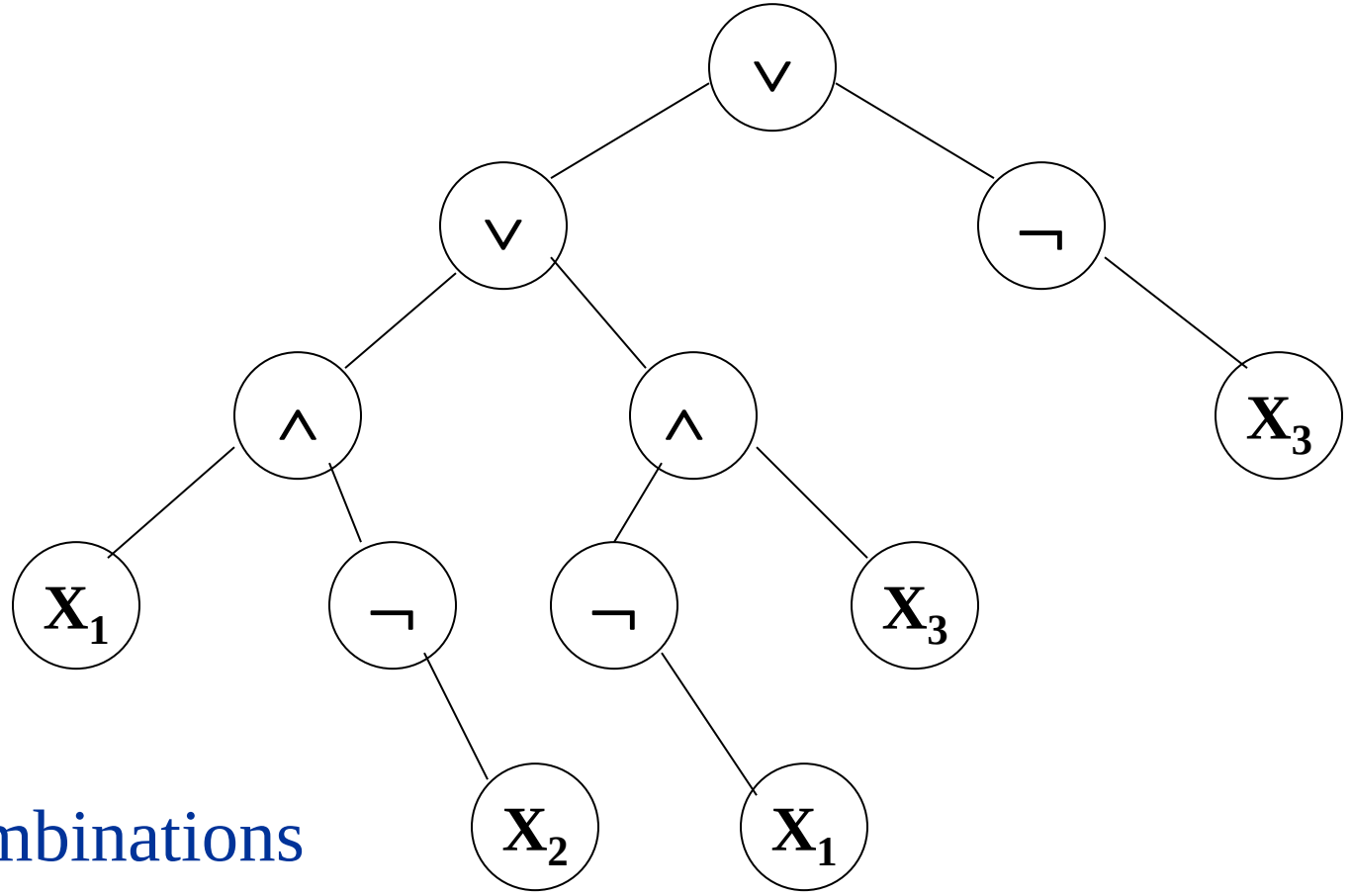
What values of x1,x2 and x3 makes the expression output true.

Set of expressions that we can form using these variables(x1….xn) and operators (¬,∨,∧) is defined by following rules:

- A variable is an expression.
- If x and y are expressions, then ¬x, x∧y, x∨y are expressions.
- Parentheses can be used to alter the normal order of evaluation (¬ > ∧ > ∨).

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

(t,t,t)
(t,t,f)
(t,f,t)
(t,f,f)
(f,t,t)
(f,t,f)
(f,f,t)
(f,f,f)



2^n possible combinations
for n variables

postorder traversal (postfix evaluation)

node structure

left_child	*data*	*value*	*right_child*

```
typedef emun {not, and, or, true, false } logical;
typedef struct node *tree_pointer;
typedef struct node
{
        tree_pointer  list_child;
        logical       data;
        short int     value;
        tree_pointer  right_child;
} ;
```

First version of satisfiability algorithm

```
for (all 2^n possible combinations)
{
      generate the next combination;
      replace the variables by their values;
      evaluate root by traversing it in postorder;
      if (root->value) {
            printf(<combination>);
            return;
   }
}
printf("No satisfiable combination \n");
```

Post-order-eval function

```
void post_order_eval(tree_pointer node)
{
/* modified post order traversal to evaluate a propositional calculus tree */
    if (node) {
        post_order_eval(node->left_child);
        post_order_eval(node->right_child);
        switch(node->data) {
           case not:  node->value =
                    !node->right_child->value;
                    break;
```

```
 case and:     node->value =
                node->right_child->value &&
                node->left_child->value;
                break;
       case or:        node->value =
                node->right_child->value | |
                node->left_child->value;
                break;
        case true:    node->value = TRUE;
                break;
        case false:  node->value = FALSE;
       }
   }
}
```

Threaded Binary Trees

A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.

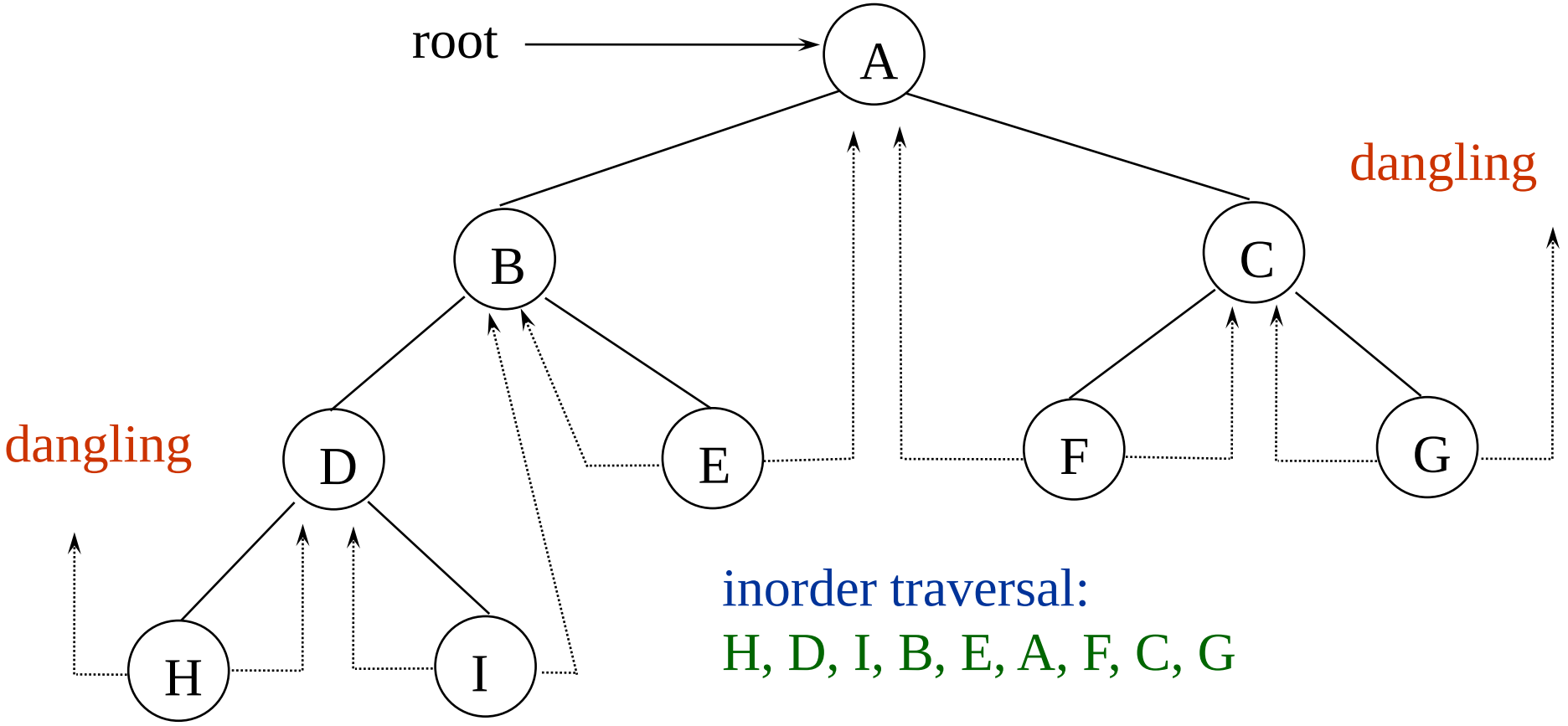
- If ptr->left_child is null,

  replace it with a pointer to the node that would be visited before ptr in an inorder traversal

- If ptr->right_child is null,

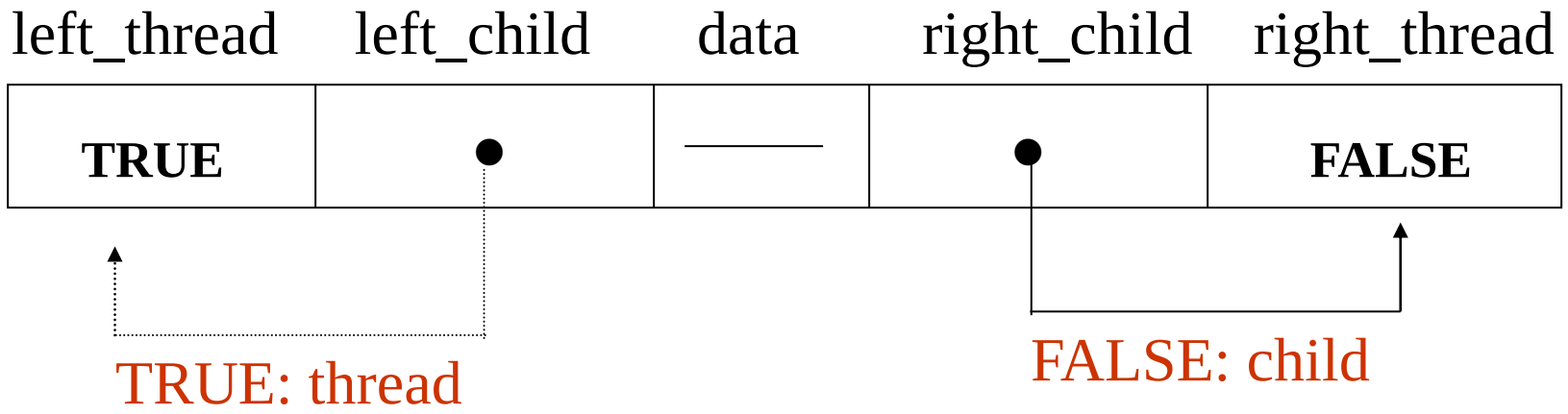
  replace it with a pointer to the node that would be visited after ptr in an inorder traversal

A Threaded Binary Tree



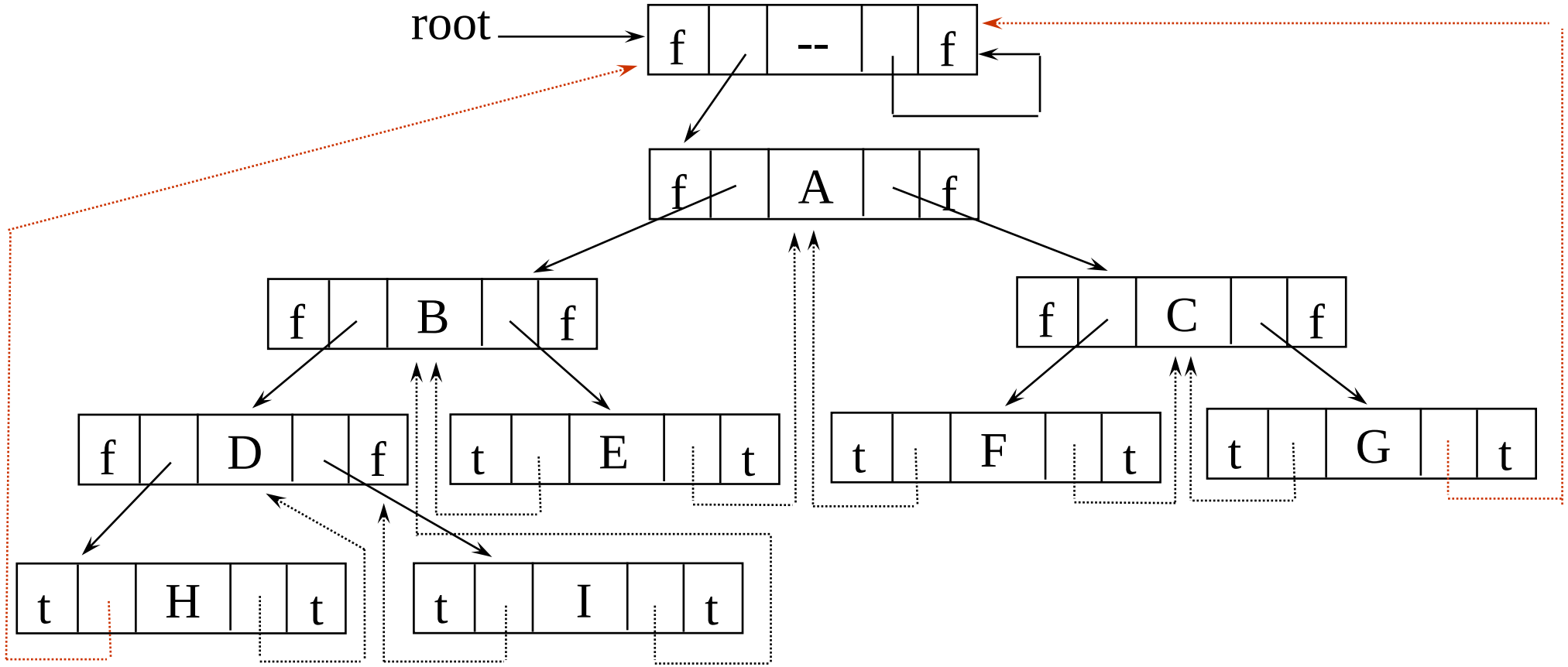
inorder traversal:
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



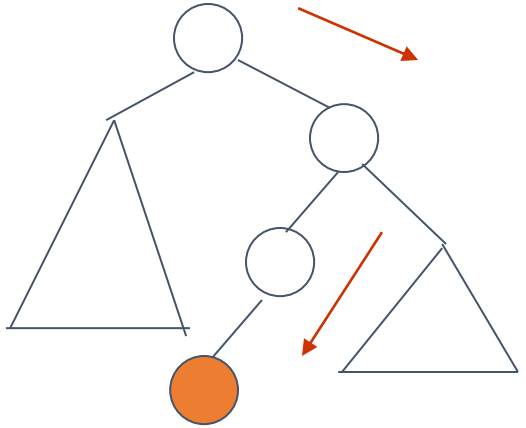

```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree
{    short int left_thread;
   threaded_pointer left_child;
   char data;
   threaded_pointer right_child;
   short int right_thread;  };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer tree)
{
    threaded_pointer temp;
    temp = tree->right_child;
    if (!tree->right_thread)
                while (!temp->left_thread)
                        temp = temp->left_child;
    return temp;
}
```



Inorder Traversal of Threaded BT

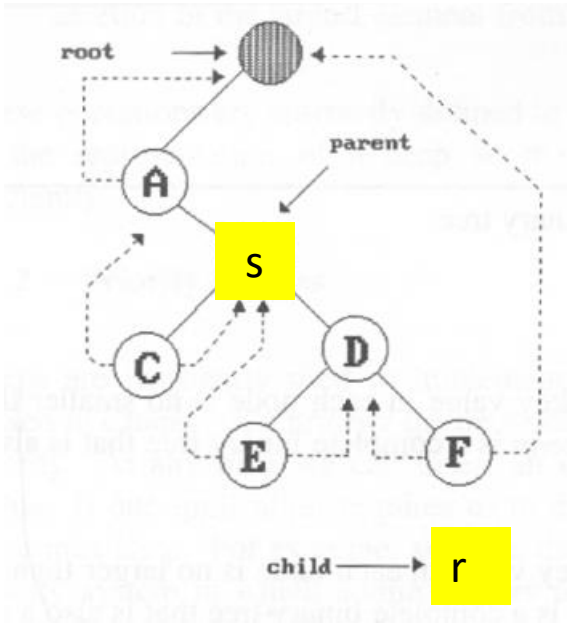
```
void tinorder(threaded_pointer tree)
{
        /* traverse the threaded binary tree inorder */
        threaded_pointer temp = tree;
        for (;;) {
                temp = insucc(temp);
                if (temp==tree) break;
                printf("%3c", temp->data);
        }
}
```

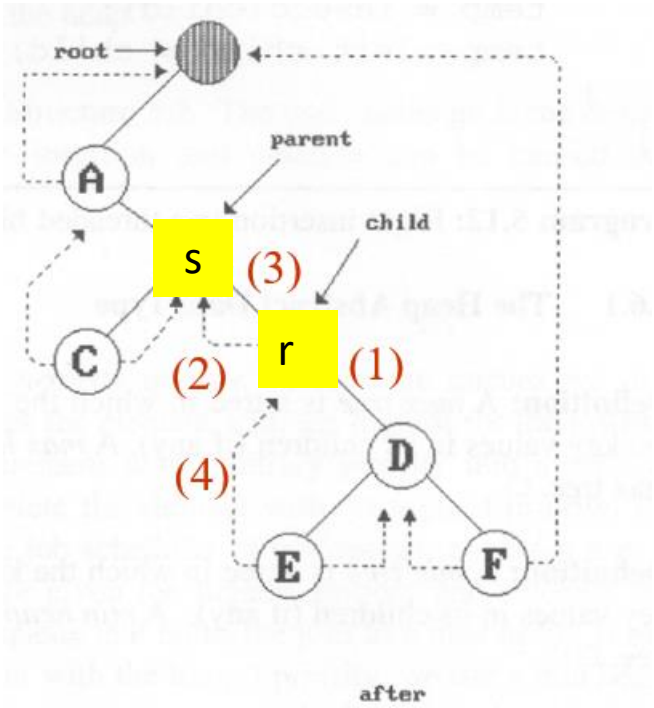
O(n)

Inserting a Node into a Threaded Binary Tree

Consider a case of inserting **node r** as the right child of **node s.**

- **Case A:** If s has empty right subtree, insertion is simple.
- **Case B:** If right subtree of node s is non empty:

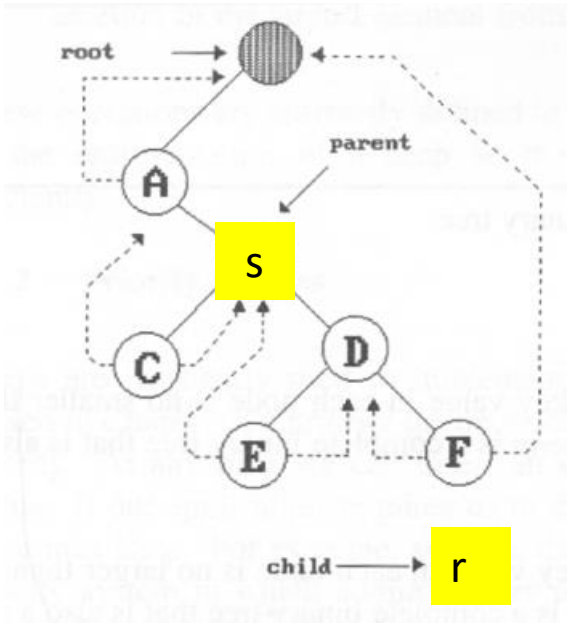


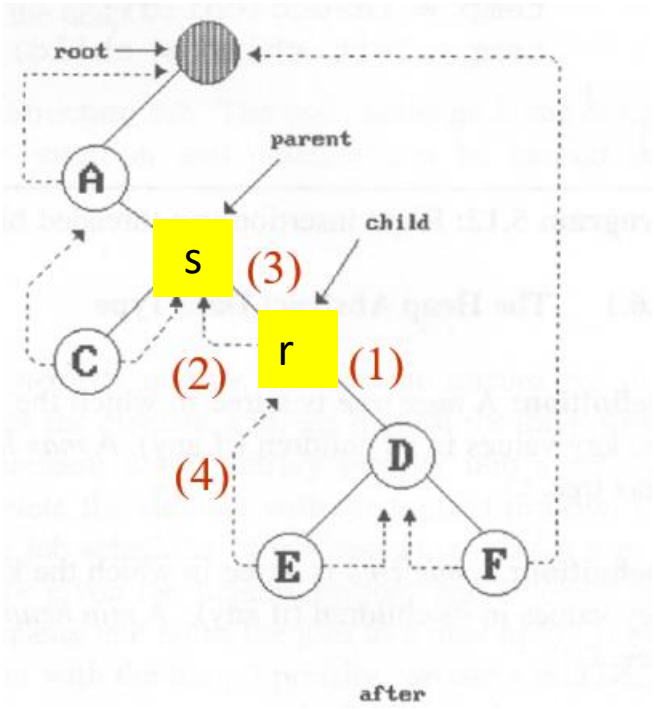


Inserting a Node into a Threaded Binary Tree

Consider a case of inserting **node r** as the right child of **node s.**

- **Case A:** If s has empty right subtree, insertion is simple.
- **Case B:** If right subtree of node s is non empty:



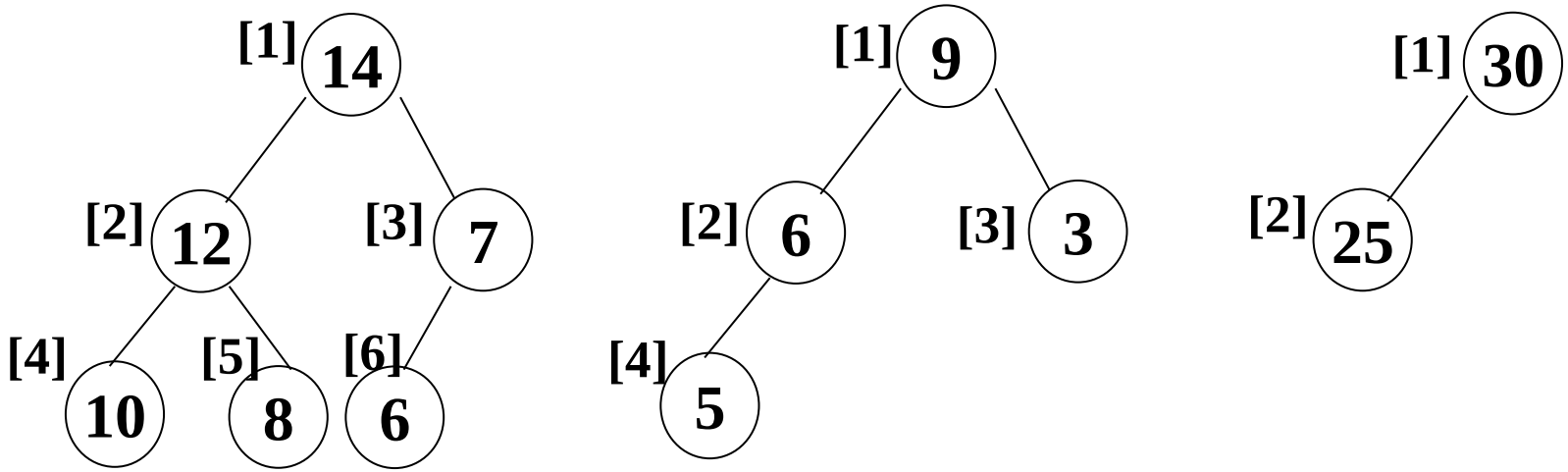



```
1. r ->rightChild=s ->rightChild;
   r->rightThread =s->rightThread;
2. r->leftChild=parent;
   r->leftThread=TRUE;
3. s->rightChild=r;
   s->rightThread=FALSE;
4. if(!r->rightThread)
{
                temp=insucc(r);
                temp->leftChild = r;
}
```

Heap

- A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.
- A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.
- Operations on heaps
  - creation of an empty heap
  - insertion of a new element into the heap;
  - deletion of the largest element from the heap

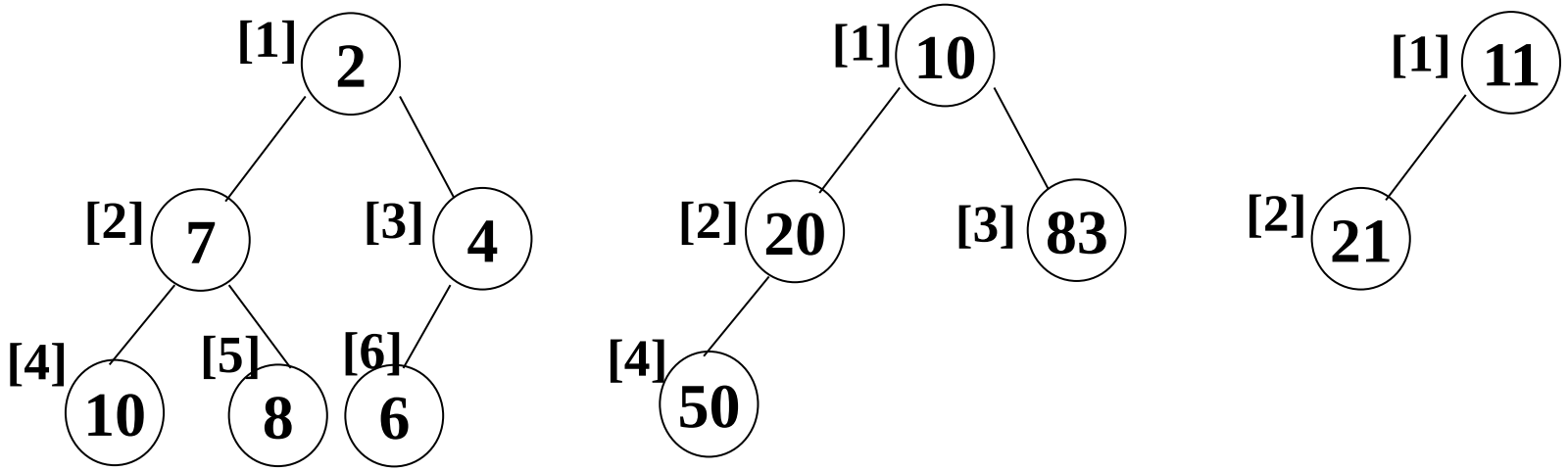
***Figure 5.25:** Sample max heaps (p.219)



Property:

The root of max heap (min heap) contains the largest (smallest).

***Figure 5.26:**Sample min heaps (p.220)



ADT for Max Heap

objects: a complete binary tree of n > 0 elements organized so that the value in each node is at least as large as those in its children

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*, *max_size* belong to integer

MaxHeap Create(max_size)::= create an empty heap that can hold a maximum of max_size elements

Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n))
insert item into heap and return the resulting heap
else return error

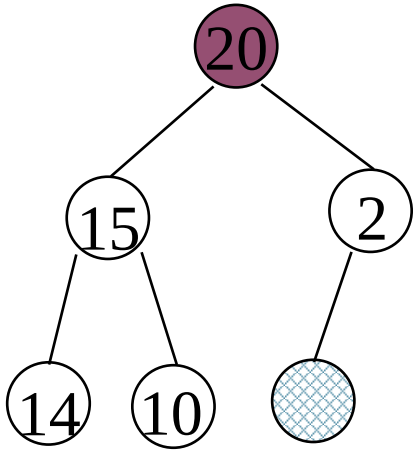
Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE
else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the largest element in the heap and remove it from the heap else return error

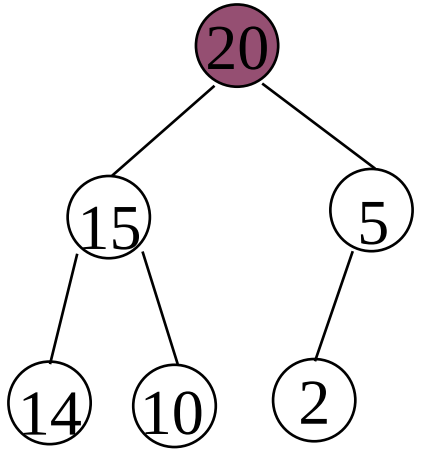
Application: priority queue

- machine service
  - amount of time (min heap)
  - amount of payment (max heap)
- factory
  - time tag

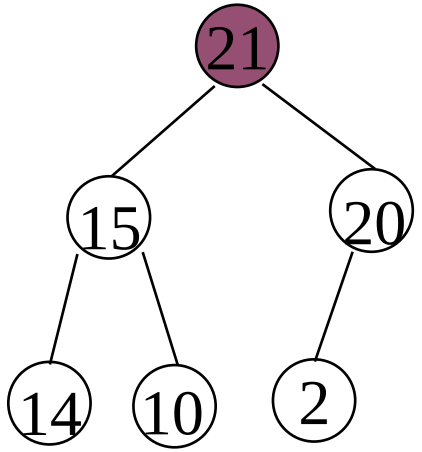
Example of Insertion to Max Heap



initial location of new node



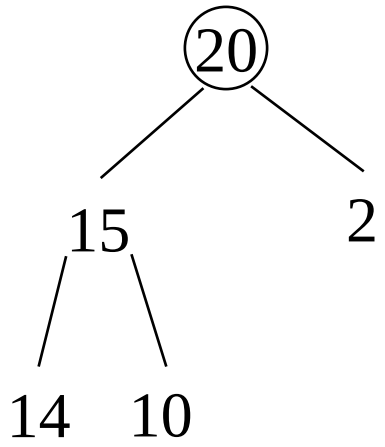
insert 5 into heap



insert 21 into heap

Insertion into a Max Heap


```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



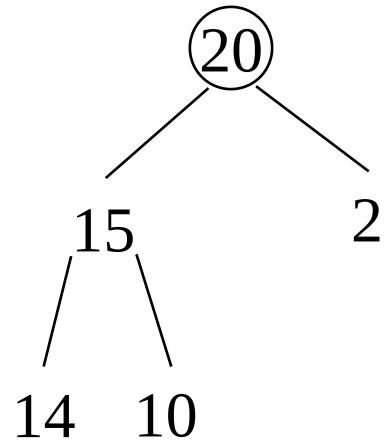
n=5

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap

```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);          //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



n=5

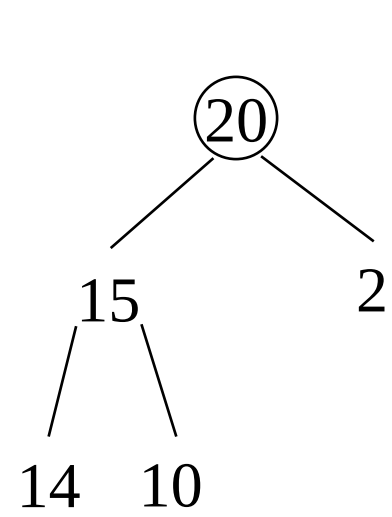
i=6, n=6

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap

```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



i=6, n=6

22>2

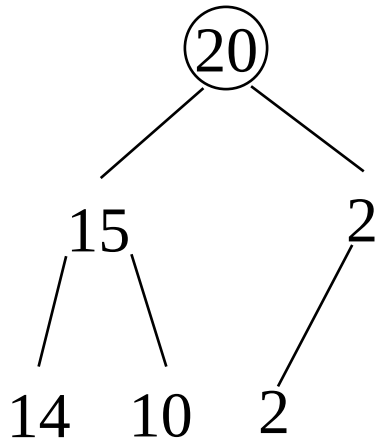
$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap

```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```

22



i=6, n=6

22>2

Heap[6]=heap[3]

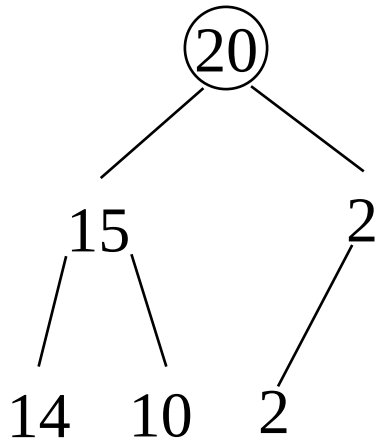
Heap[6]=2

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap


```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



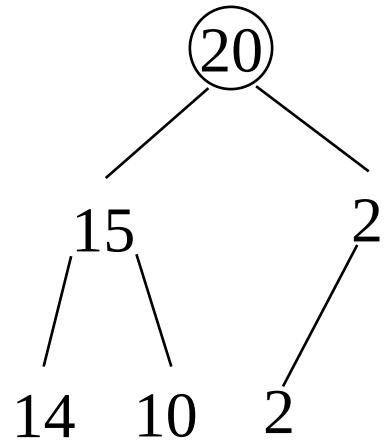
i=3

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap

```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



22

i=3

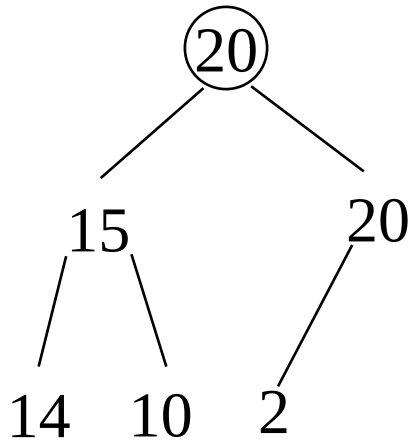
22>20

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap


```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



heap[3]=20

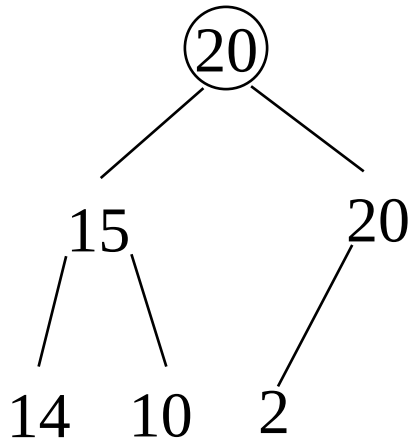
i=1

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Insertion into a Max Heap


```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



heap[3]=20

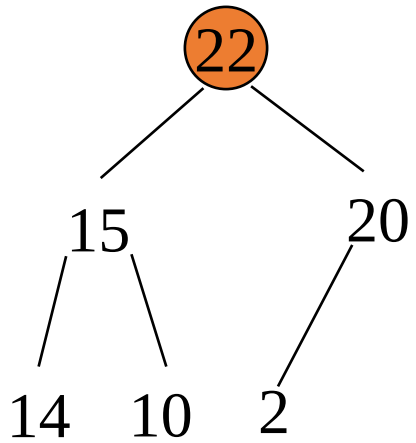
i=1

$2^k-1=n \Rightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

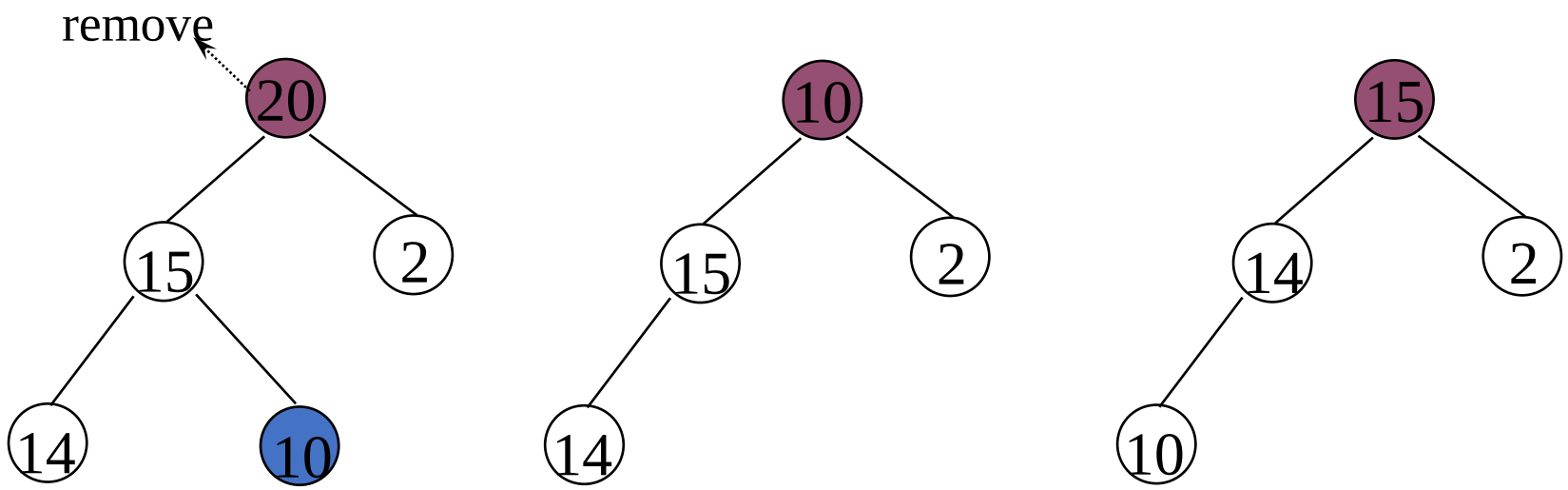
Insertion into a Max Heap

```
void insert_max_heap(element item, int n) //n= number of nodes
 {      int i;
        if (HEAP_FULL(n)) {
                fprintf(stderr, "the heap is full.\n");
                exit(1);
        }
        i = ++(n);        //new position = number of nodes+1
        while ((i!=1)&&(item.key>heap[i/2].key))
        {
                heap[i] = heap[i/2];
                i /= 2;
        }
        heap[i]= item;
}
```



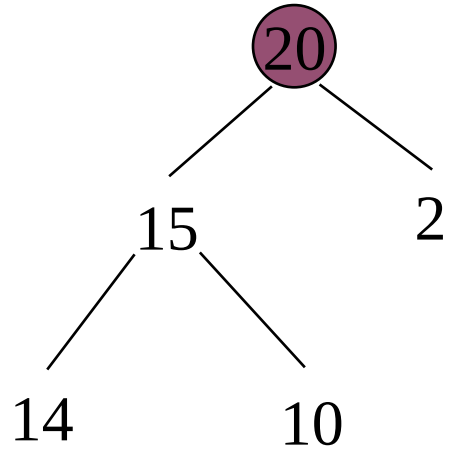
heap[1]=22
i=1

Example of Deletion from Max Heap



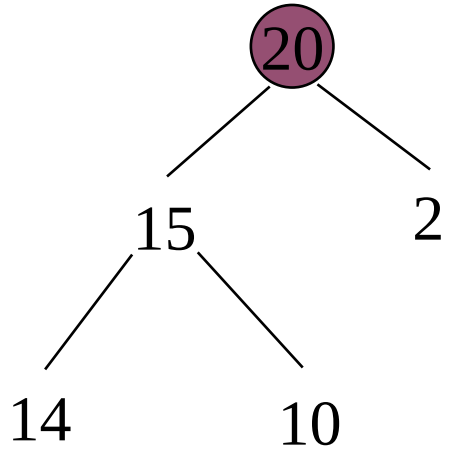
Deletion from a Max Heap

```
element delete_max_heap(int n)
{
        int parent, child;
        element item, temp;
        if (HEAP_EMPTY(n)) {
                fprintf(stderr, "The heap is empty\n");
                exit(1);
        }
                /* save value of the element with the highest key */
         item = heap[1];
                 /* use last element in heap to adjust heap */
        temp = heap[(n)--];
        parent = 1;
        child = 2;
```



Item=20
temp=10,n=4
parent=1
child=2

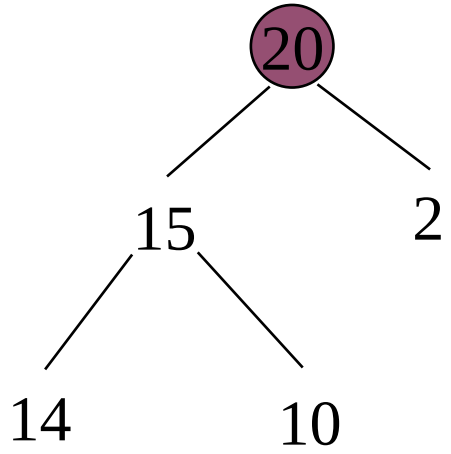
```
while (child <= n)
{
        /* find the larger child of the current parent */
    if ((child < n)&& (heap[child].key<heap[child+1].key))
        child++;
    if (temp.key >= heap[child].key) break;
        /* move to the next lower level */
    heap[parent] = heap[child];
    parent=child;
    child *= 2;
}
heap[parent] = temp;
return item;
}
```



child=2,n=4

1<=4

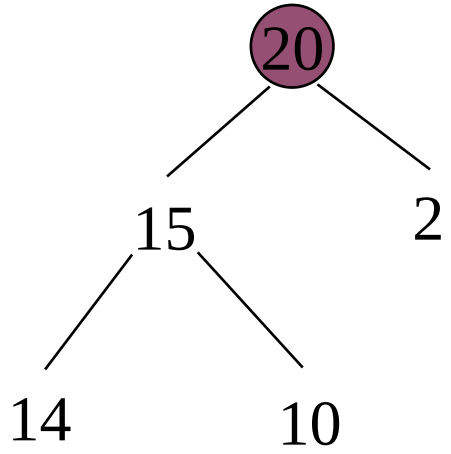
```
        while (child <= n)
        {
                        /* find the larger child of the current parent */
                if ((child < n)&& (heap[child].key<heap[child+1].key))
                        child++;
                if (temp.key >= heap[child].key) break;
                                        /* move to the next lower level */
                heap[parent] = heap[child];
                parent=child;
                child *= 2;
        }
  heap[parent] = temp;
  return item;
}
```



child=2,n=4

2<=4 and 15<2

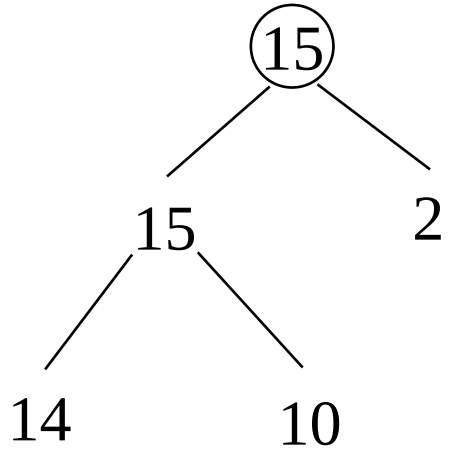
```
        while (child <= n)
        {
                        /* find the larger child of the current parent */
                if ((child < n)&& (heap[child].key<heap[child+1].key))
                        child++;
                if (temp.key >= heap[child].key) break;
                                        /* move to the next lower level */
                heap[parent] = heap[child];
                parent=child;
                child *= 2;
        }
  heap[parent] = temp;
  return item;
}
```



child=2,n=4

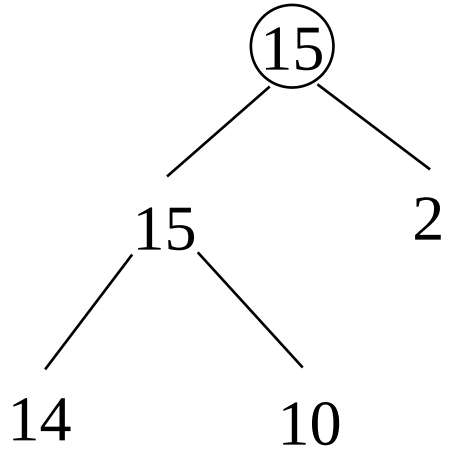
10 >=15

```
        while (child <= n)
        {
                        /* find the larger child of the current parent */
                if ((child < n)&& (heap[child].key<heap[child+1].key))
                        child++;
                if (temp.key >= heap[child].key) break;
                                        /* move to the next lower level */
                heap[parent] = heap[child];
                parent=child;
                child *= 2;
        }
  heap[parent] = temp;
  return item;
}
```



heap[1]=15
Parent=2
Child=4

```
        while (child <= n)
        {
                        /* find the larger child of the current parent */
                if ((child < n)&& (heap[child].key<heap[child+1].key))
                        child++;
                if (temp.key >= heap[child].key) break;
                                        /* move to the next lower level */
                heap[parent] = heap[child];
                parent=child;
                child *= 2;
        }
  heap[parent] = temp;
  return item;
}
```

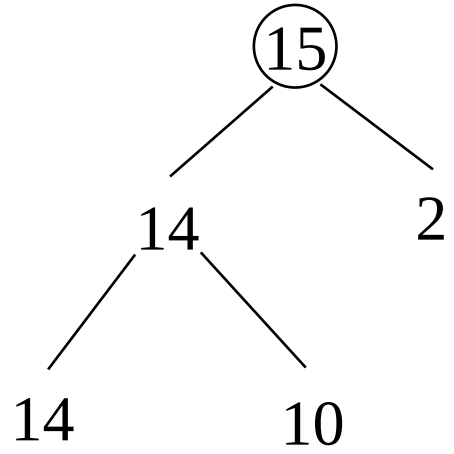


Parent=2

Child=4

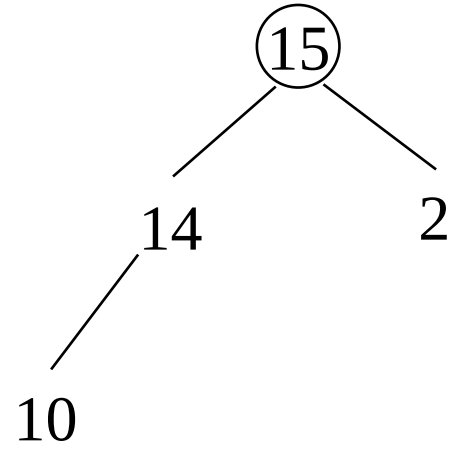
10>14

```
        while (child <= n)
        {
                        /* find the larger child of the current parent */
                if ((child < n)&& (heap[child].key<heap[child+1].key))
                        child++;
                if (temp.key >= heap[child].key) break;
                                        /* move to the next lower level */
                heap[parent] = heap[child];
                parent=child;
                child *= 2;
        }
  heap[parent] = temp;
  return item;
}
```



Heap[2]=heap[4]
parent=4
Child=8

```
while (child <= n)
{
        /* find the larger child of the current parent */
    if ((child < n)&& (heap[child].key<heap[child+1].key))
        child++;
    if (temp.key >= heap[child].key) break;
        /* move to the next lower level */
    heap[parent] = heap[child];
    parent=child;
    child *= 2;
}
 heap[parent] = temp;
 return item;
}
```



Heap[4]=10
parent=4
Child=8
Return 22;

Binary Search Tree

Lab program:

Write a C program for the following operations:

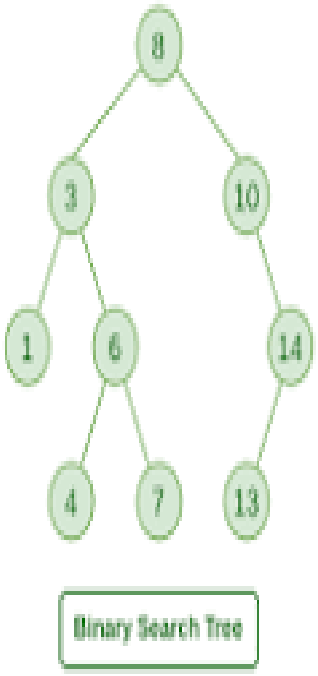
a) Write a function to construct binary search tree.

b) Write a function to perform Binary search tree traversal using:

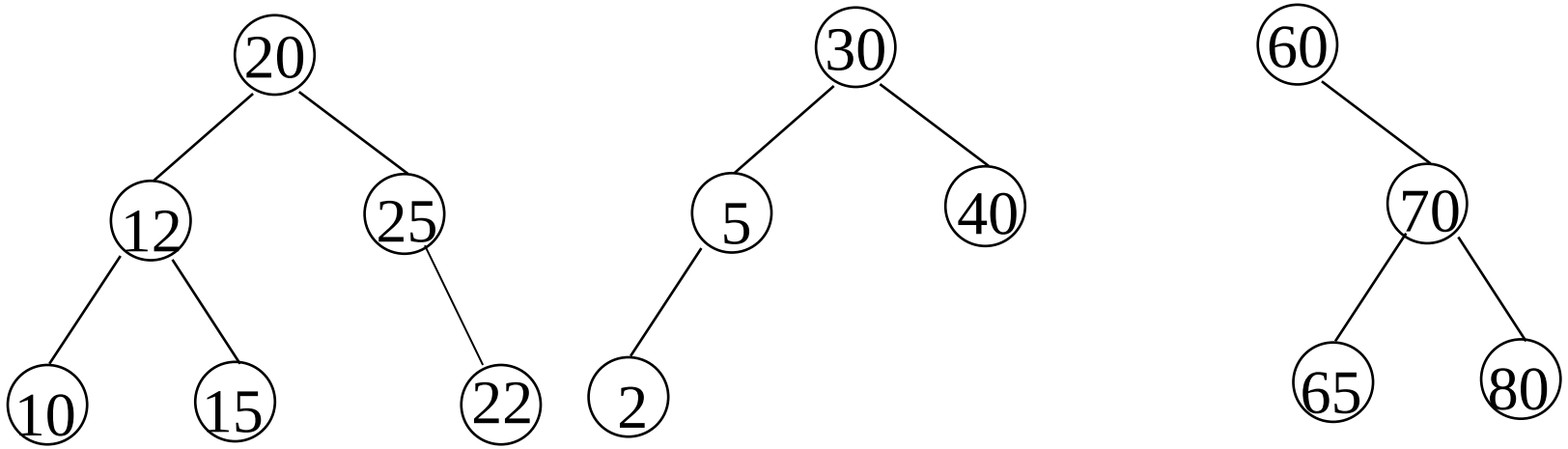
- i) In-order traversal
- ii) Pre-order traversal
- iii) Post-order traversal

Binary Search Tree

In computer science, a binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.

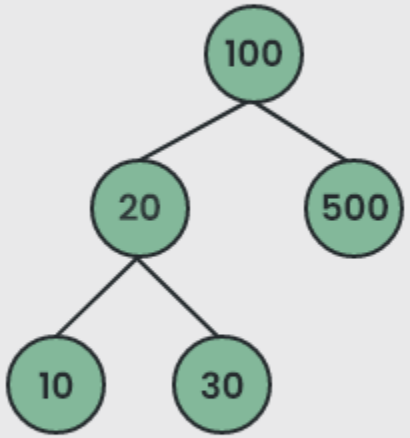


Examples of Binary Search Trees



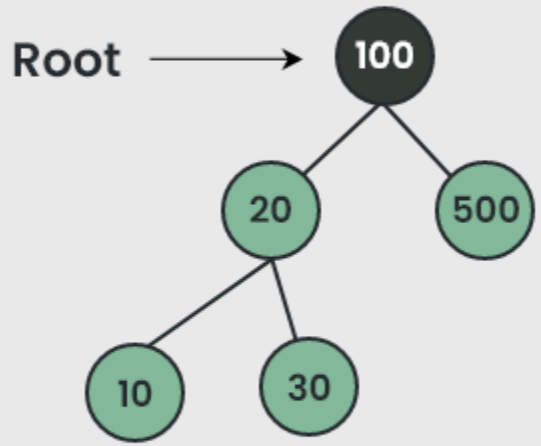
Inserting into Binary tree

Consider The Following BST



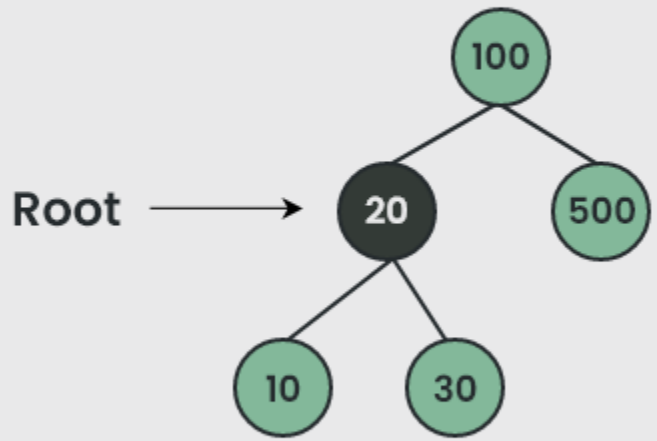
X = 40 (The Node To Be Inserted)

STEP 1 : Comparing X with Root Node



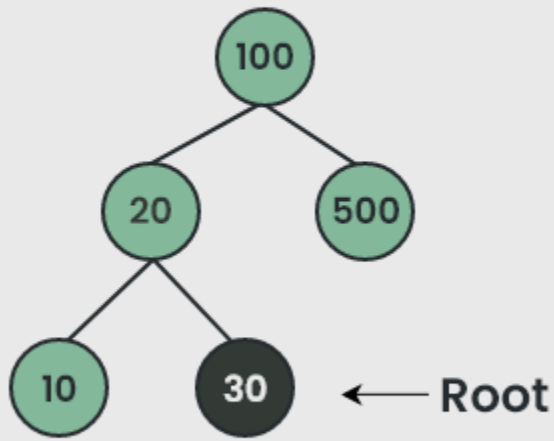
Since 100 Is Greater Than 40.
Move Pointer To The Left Child (20)

STEP 2 : Comparing X with left child of root node



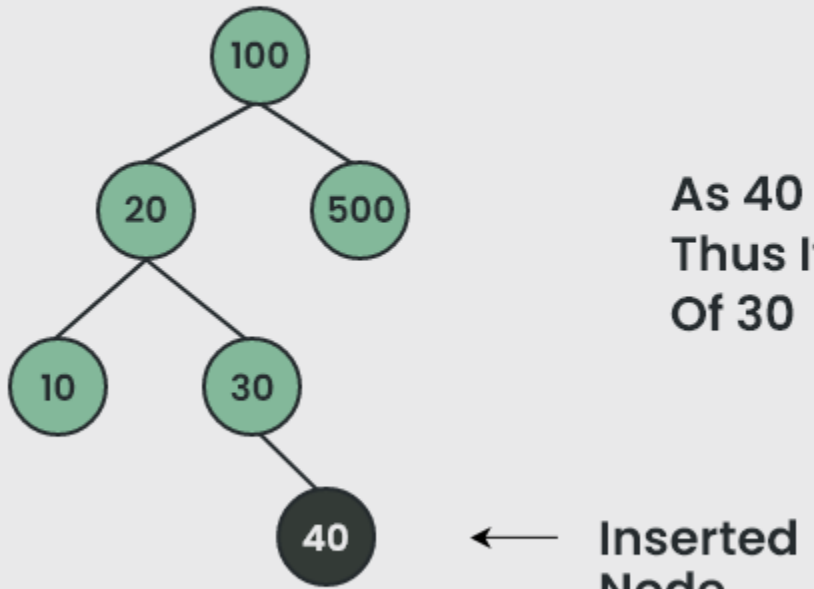
Since 20 Is Less Than 40, Move Pointer To The Right Child (30)

STEP 3 : Comparing x with the right child of 20



Again 40 Is Greater Than 30
Move Pointer To The Right Side
Of 30

STEP 4 :Insert item to the right of 30



As 40 Is Greater Than The Node 30, Thus It Will Be Inserted To The Right Of 30

Searching a Binary Search Tree

```
tree_pointer search(tree_pointer root, int key)
{
/* return a pointer to the node that contains key. If there is no such
   node, return NULL */
        if (!root) return NULL;
        if (key == root->data)
                return root;
        if (key < root->data)
                return search(root->left_child, key);
  return search(root->right_child,key);
}
```

Searching Algorithm –Iterative method

```
tree_pointer search2(tree_pointer tree, int key)
{
        while (tree)
        {
                if (key == tree->data) return tree;
                if (key < tree->data)
                        tree = tree->left_child;
                else tree = tree->right_child;
        }
        return NULL;
}
```

O(h)

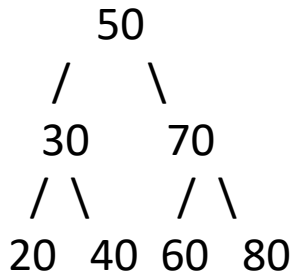
Program Code

```
#include <stdio.h>
#include <stdlib.h>

                                                    //create node structure

struct node
{
        int key;
        struct node *left, *right;
};
```

```
struct node* newNode(int item)        // create node memory
{
        struct node* temp = (struct node*)malloc(sizeof(struct node));
        temp->key = item;
        temp->left = temp->right = NULL;
        return temp;
}
void inorder(struct node* root)       // in-order traversal
{
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->key);
      inorder(root->right);
   }
}
```

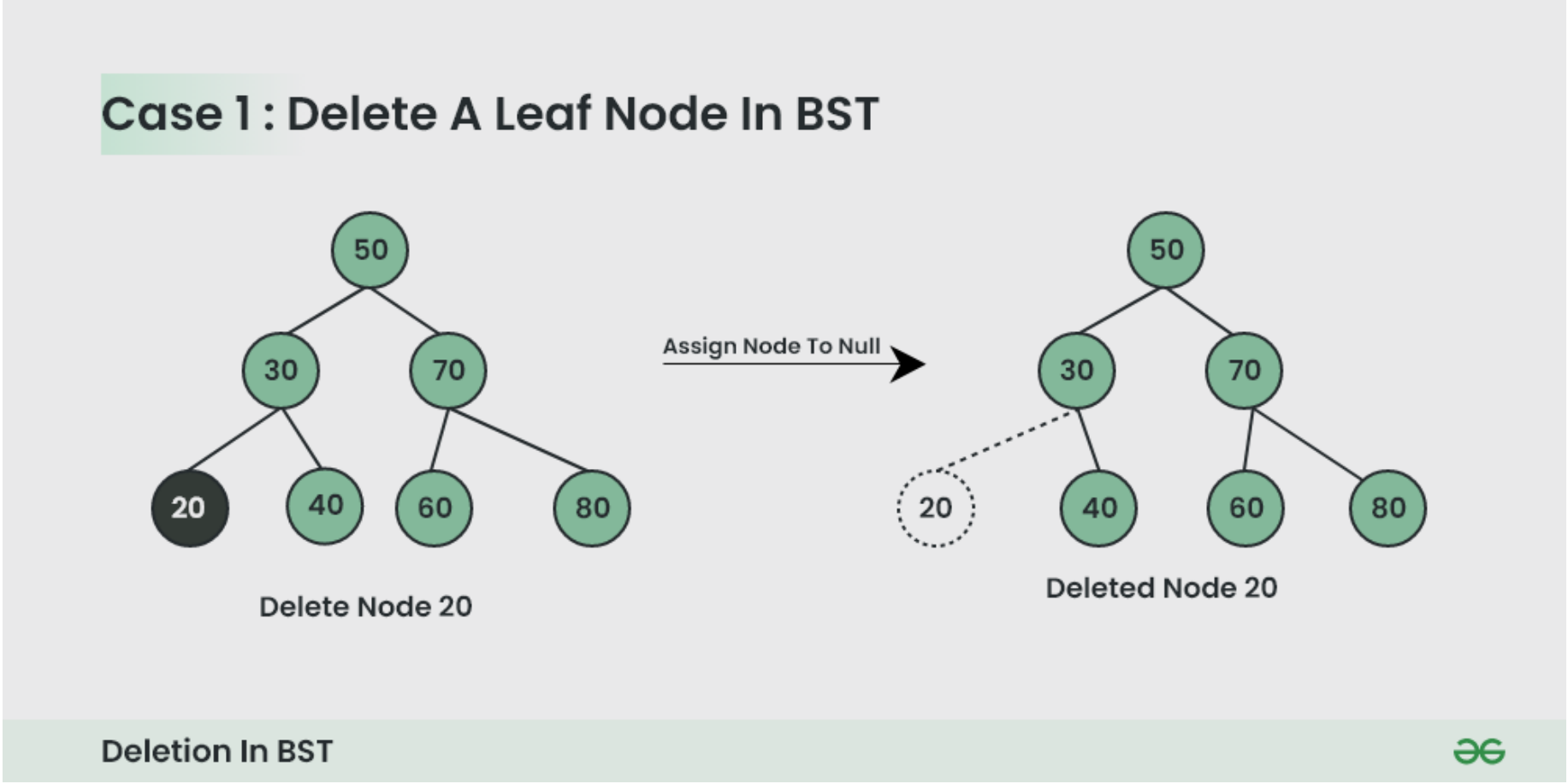
```
// a new node with given key in BST
struct node* insert(struct node* node, int key)
{
        // If the tree is empty, return a new node
        if (node == NULL)
                return newNode(key);

        // Otherwise, recur down the tree
        if (key  <  node->key)
                node->left = insert(node->left, key);
        else if (key > node->key)
                node->right = insert(node->right, key);

        // Return the (unchanged) node pointer
        return node;
}
```

```
int main()
{
    /* Let us create following BST
            struct node* root = NULL;
           root = insert(root, 50);
           insert(root, 30);
           insert(root, 20);
           insert(root, 40);
           insert(root, 70);
           insert(root, 60);
           insert(root, 80);
     // Print inorder traversal of the BST
           inorder(root);
           return 0;
}
```

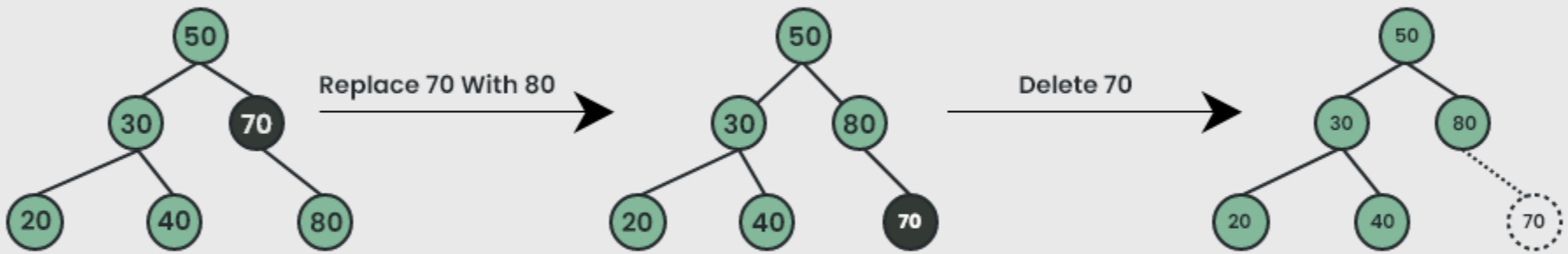


```
struct node* search(struct node* root, int key)
{
                    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
            return root;

                    // Key is greater than root's key
    if (root->key < key)
            return search(root->right, key);

                    // Key is smaller than root's key
    return search(root->left, key);
}
```

Deletion in Binary Search Tree (BST)



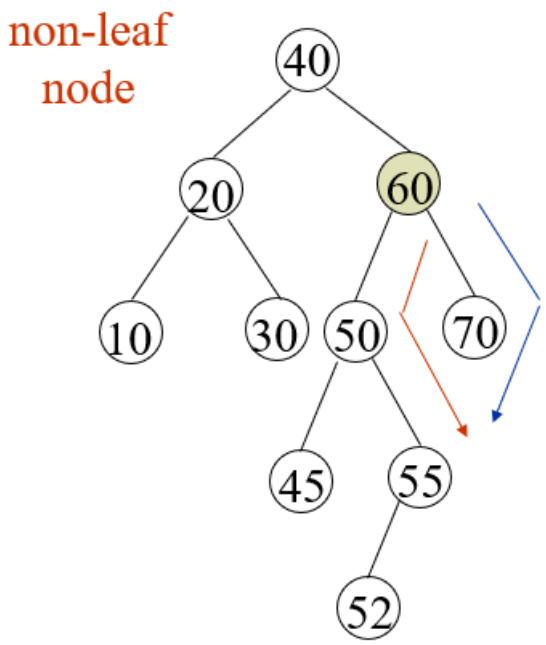
Case 2: Delete A Node With Single Child In BST



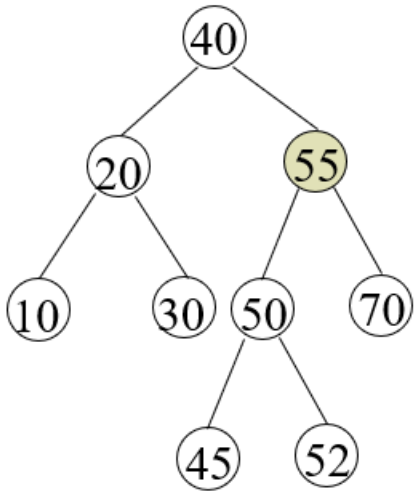
Delete Node 70

After Deletion

Delete a node-with both children



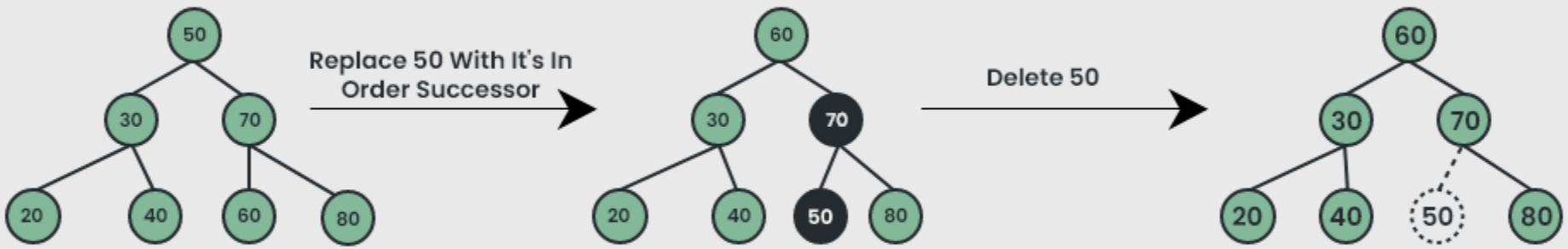
Before deleting 60



After deleting 60

When the non-leaf node with both the children must be deleted , the node should be replaced with either the largest pair in its left subtree or the smallest one in the right sub tree.

Case 3 : Delete A Node With Both Children In BST



Delete Node 50

After Deletion

Selection trees

A Selection tree(tournament tree) is a form of complete binary tree in which each node denotes a player. The last level has n-1 nodes (external nodes) used to represent all the players, and the rest of the nodes (internal nodes) represent either the winner or loser among them.

(1) winner tree

(2) loser tree

winner tree

sequential allocation scheme (complete binary tree)

Each node represents the smaller of its two children.

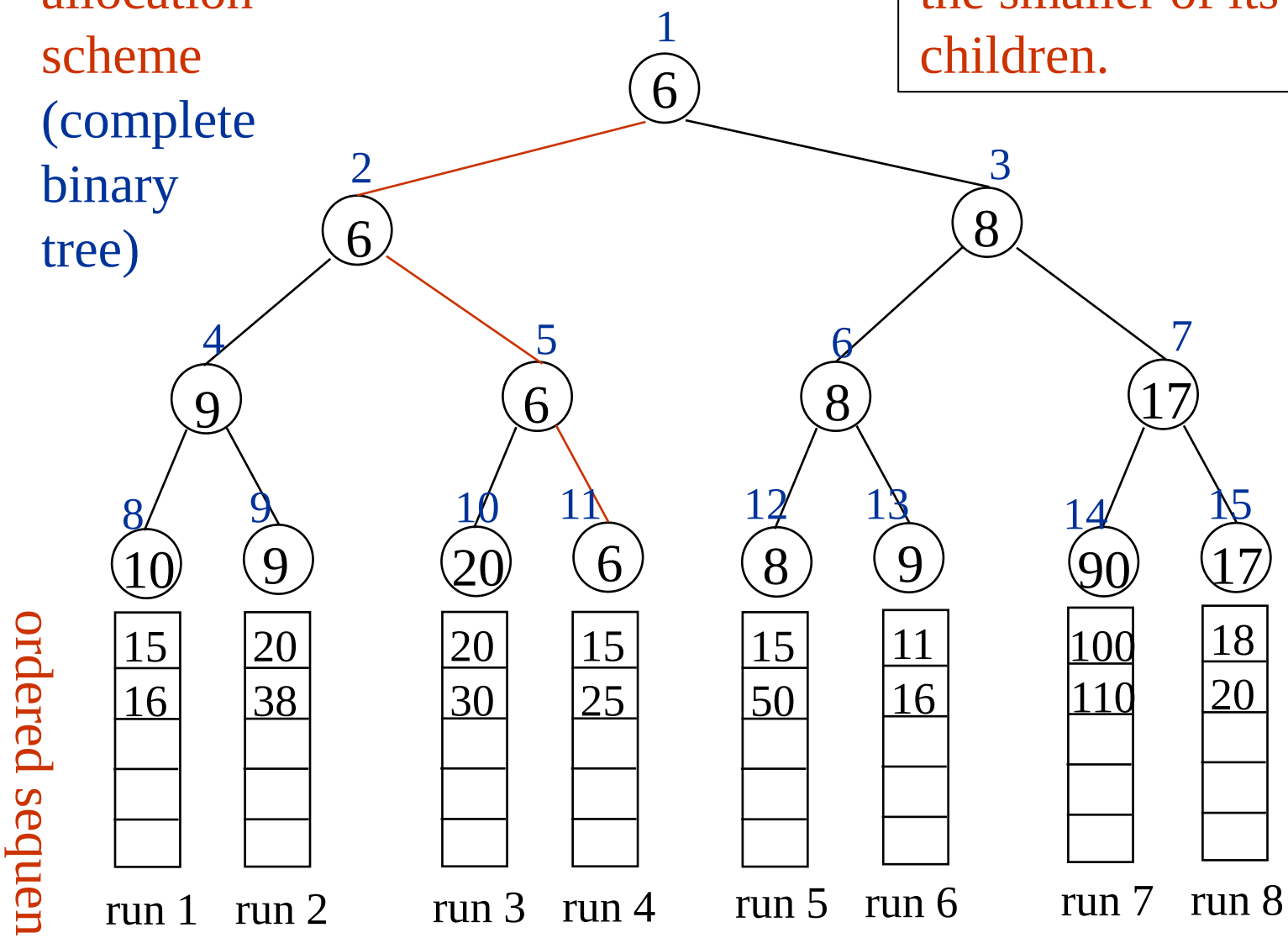


***Figure 5.35:** Selection tree of Figure 5.34 after one record has been output and the tree restructured(nodes that were changed are ticked)

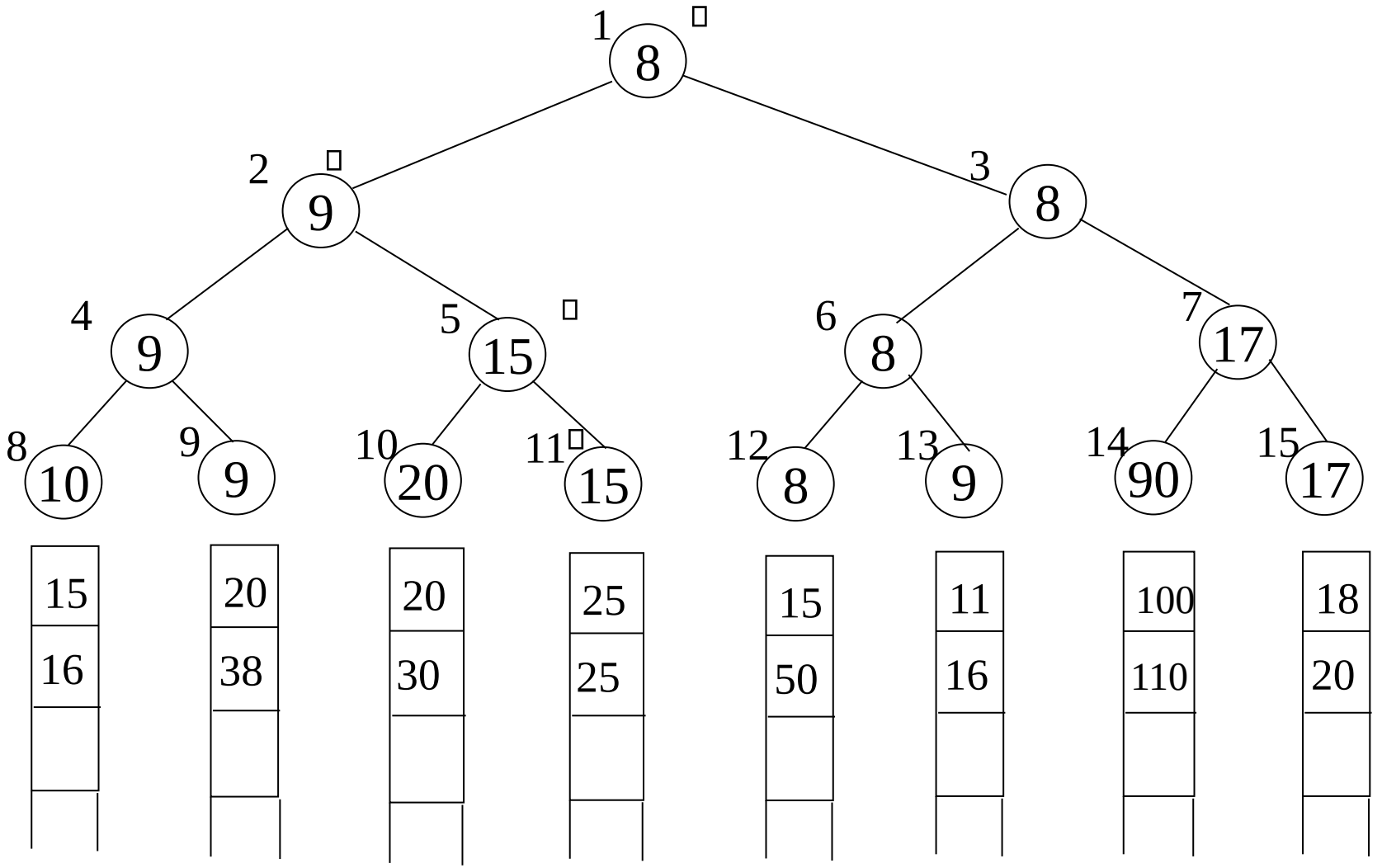
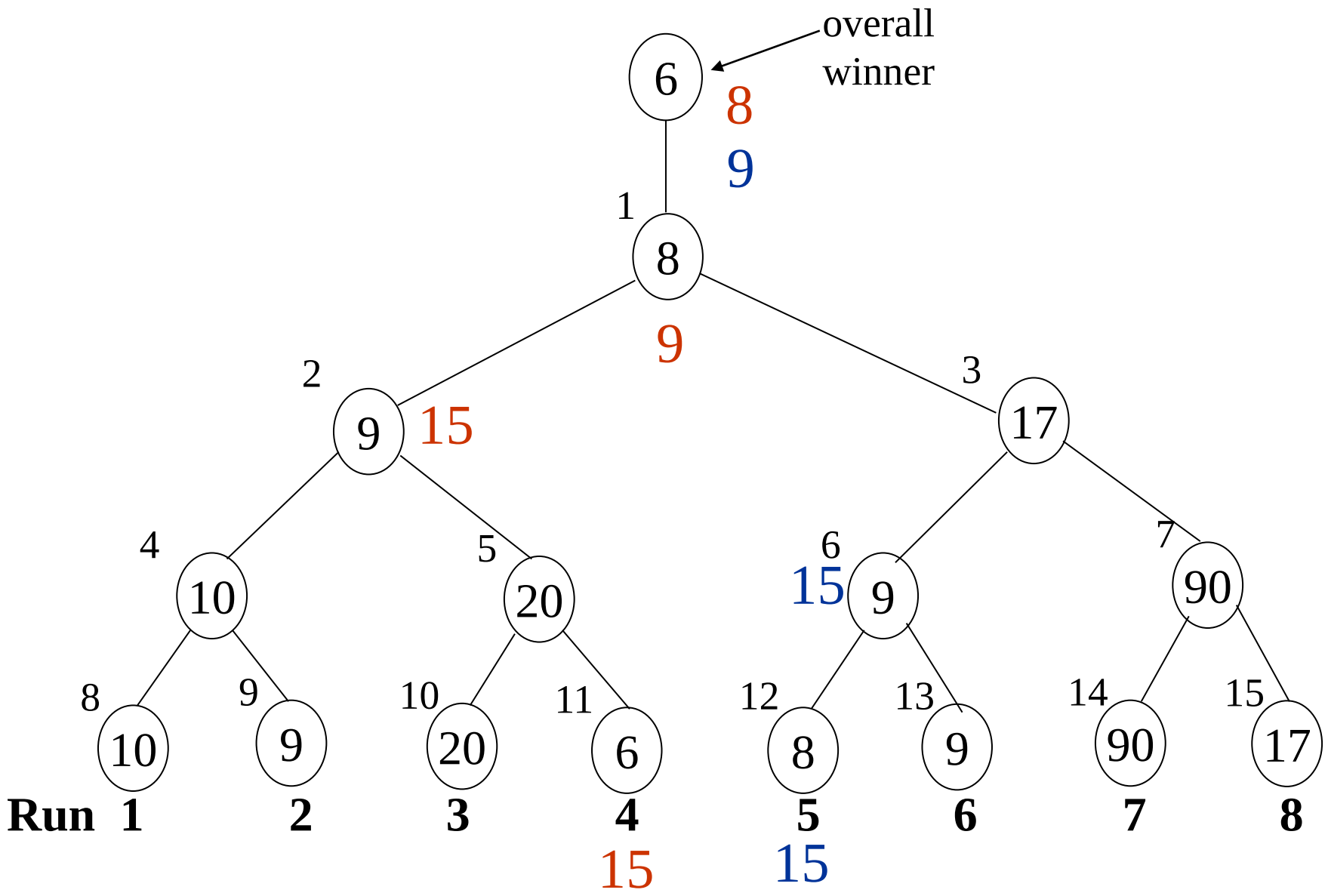
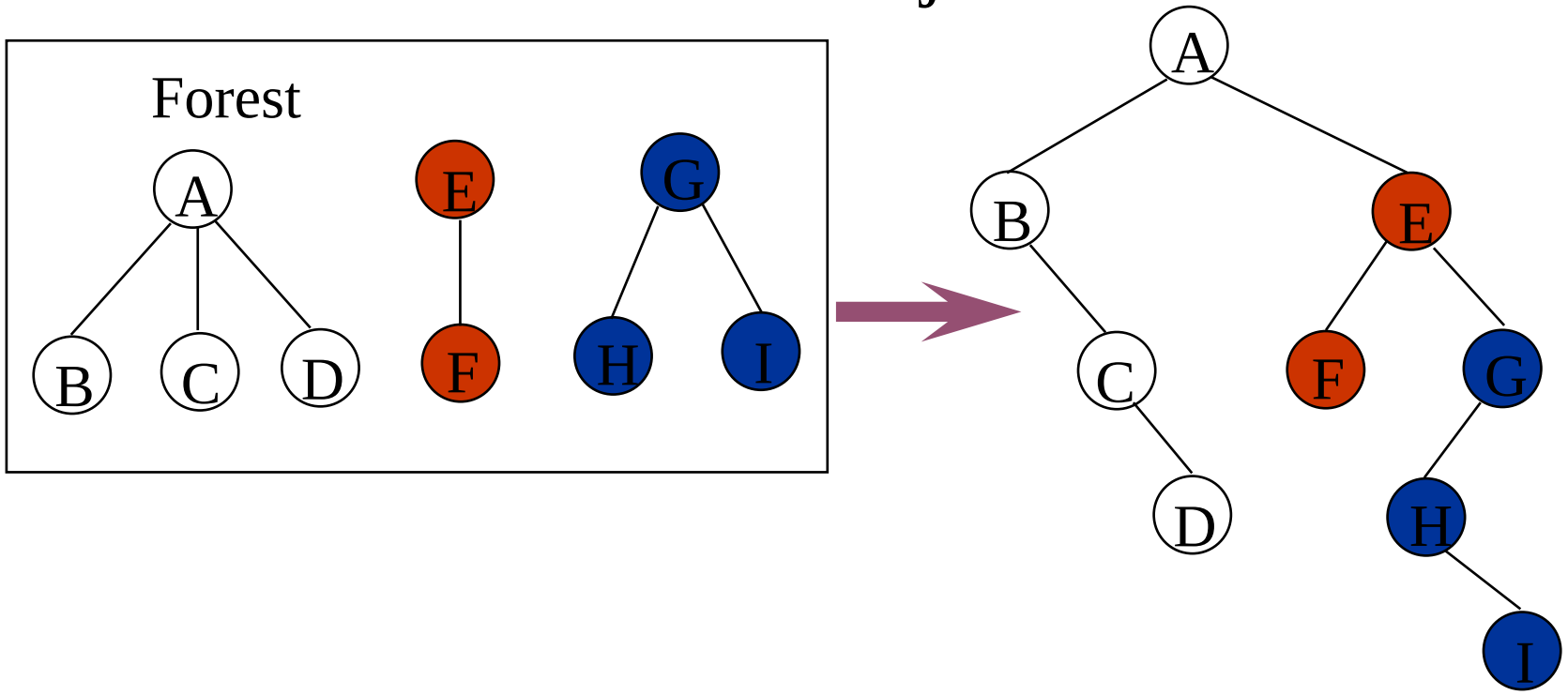


***Figure 5.36:** Tree of losers corresponding to Figure 5.34 (p.235)



Forest

- A forest is a set of n >= 0 disjoint trees



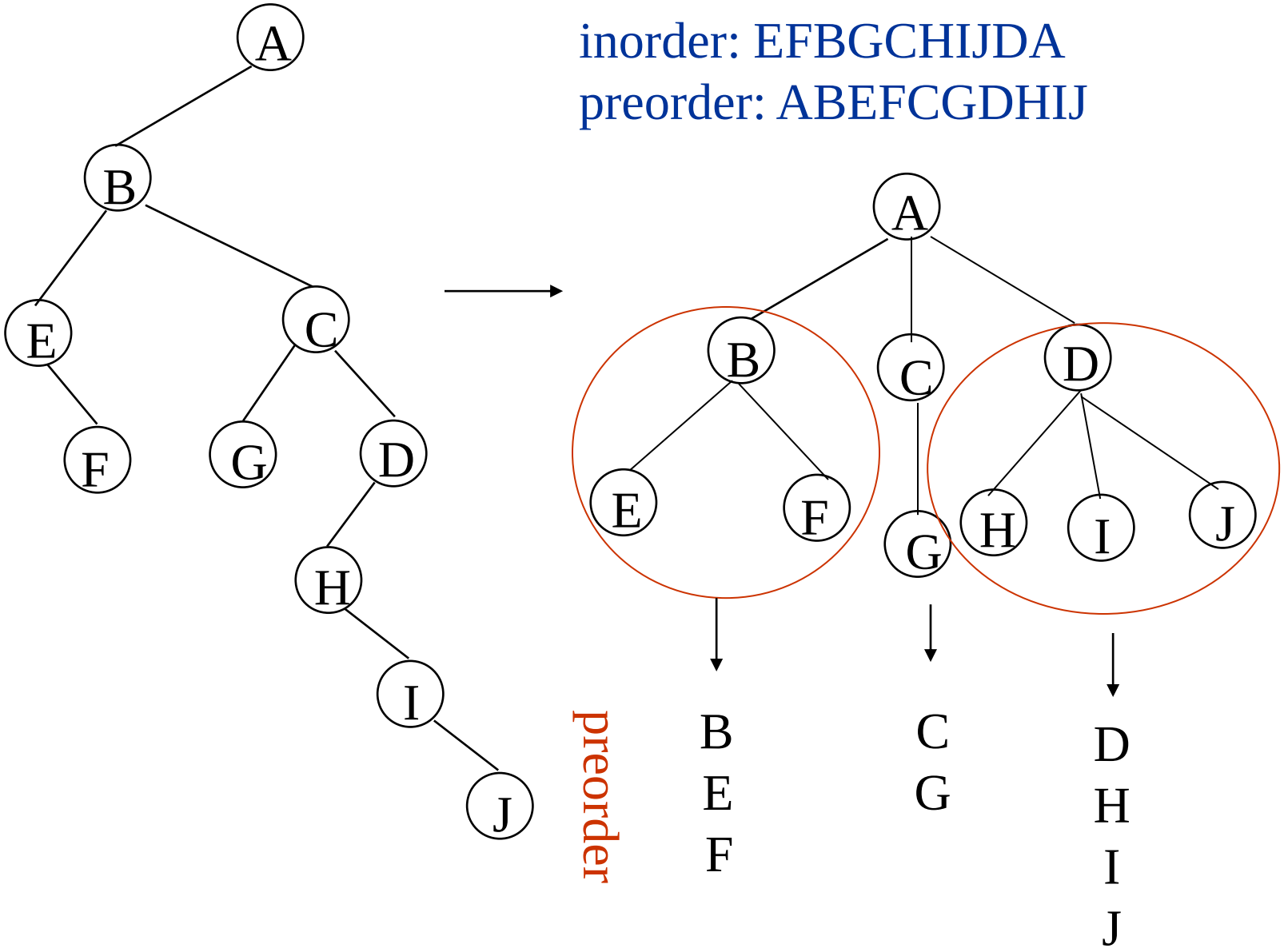
Transform a forest into a binary tree

- T1, T2, …, T*n*: a forest of trees
  B(T1, T2, …, Tn): a binary tree corresponding to this forest algorithm

  (1) empty, if n = 0
  (2) has root equal to root(T1)
      has left subtree equal to B(T11,T12,…,T1*m*)
      has right subtree equal to B(T2,T3,…,Tn)

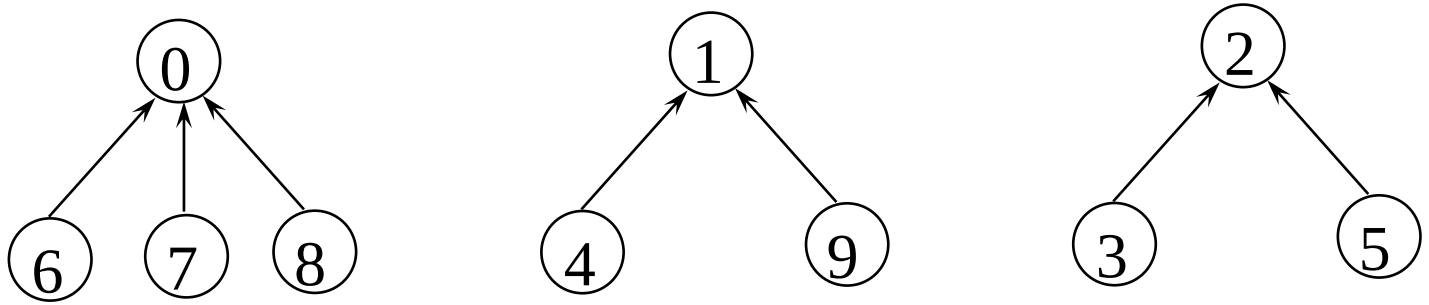
Forest Traversals

- Preorder
  - If F is empty, then return
  - Visit the root of the first tree of F
  - Taverse the subtrees of the first tree in tree preorder
  - Traverse the remaining trees of F in preorder
- Inorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in tree inorder
  - Visit the root of the first tree
  - Traverse the remaining trees of F is indorer


A
B
E
C
F
G
D
H
I
J
inorder: EFBGCHIJDA
preorder: ABEFCGDHIJ
A
B
C
D
E
F
G
H
I
J
preorder
B
E
F
C
G
D
H
I
J

Set Representation

- $S_1=\{0, 6, 7, 8\}$, $S_2=\{1, 4, 9\}$, $S_3=\{2, 3, 5\}$



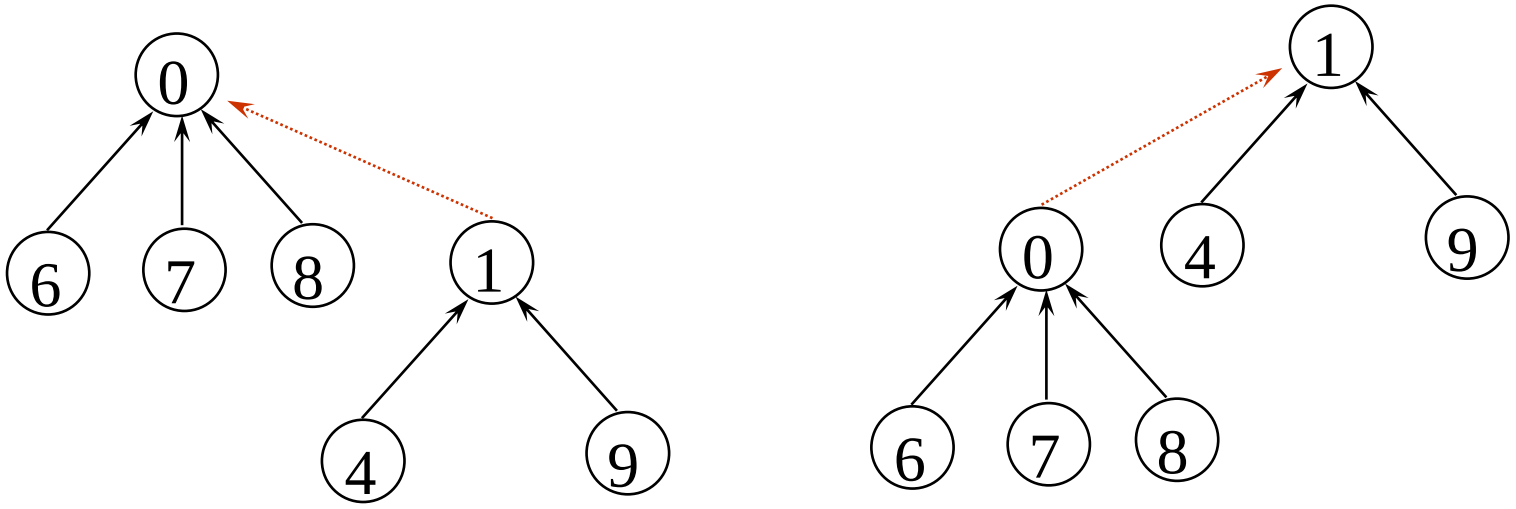
$S_i \cap S_j = \phi$

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

$3 \in S_3$, $8 \in S_1$

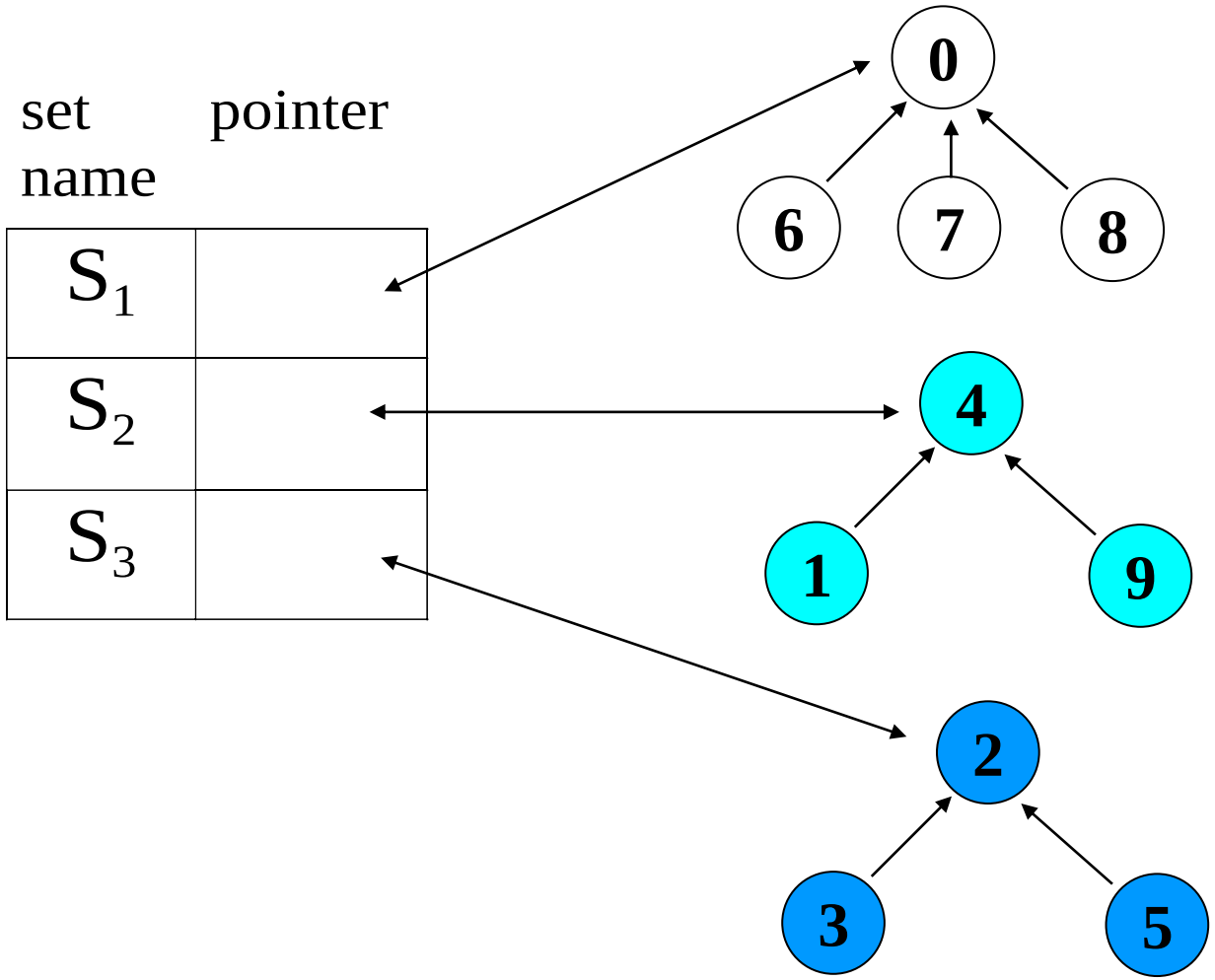
Disjoint Set Union

Make one of trees a subtree of the other



Possible representation for S_1 *union* S_2

***Figure 5.41:** Data Representation of S_1 S_2 and S_3 (p.240)



Array Representation for Set

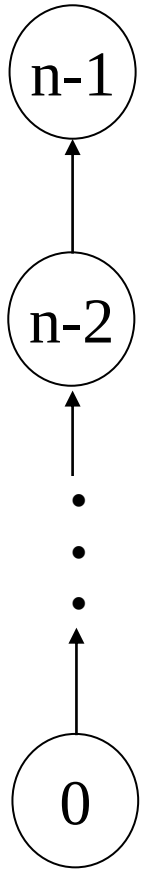
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

```
int find1(int i)
{
    for (; parent[i]>=0; i=parent[i]);
    return i;
}

void union1(int i, int j)
{
    parent[i]= j;
}
```

***Figure 5.43:**Degenerate tree (p.242)

union operation
O(n) n-1

find operation
$O(n^2)$ $\sum_{i=2}^{n} i$



union(0,1), find(0)
union(1,2), find(0)
.
.
.
union(n-2,n-1),find(0)

degenerate tree

***Figure 5.44:**Trees obtained using the weighting rule(p.243)

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

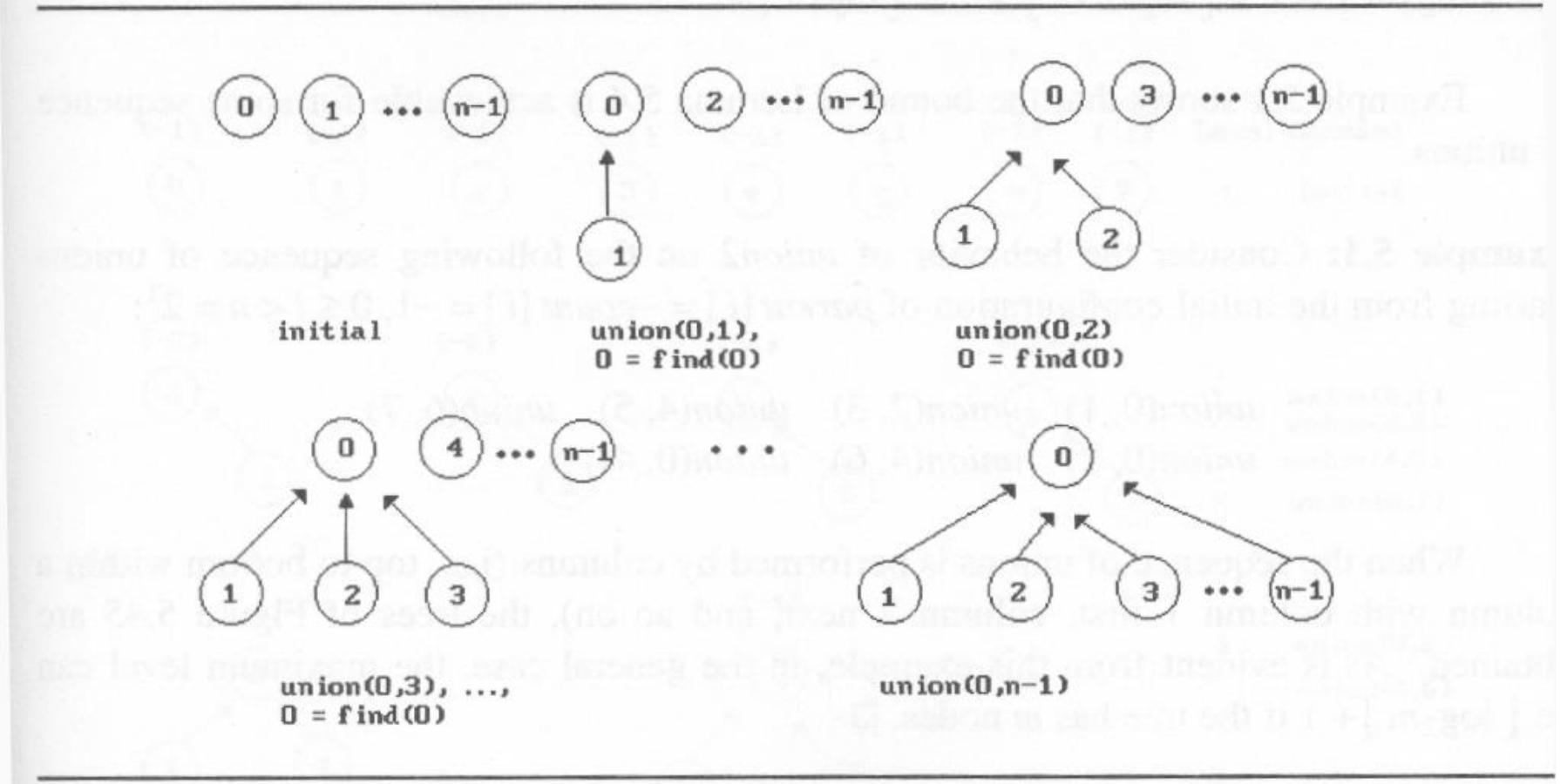


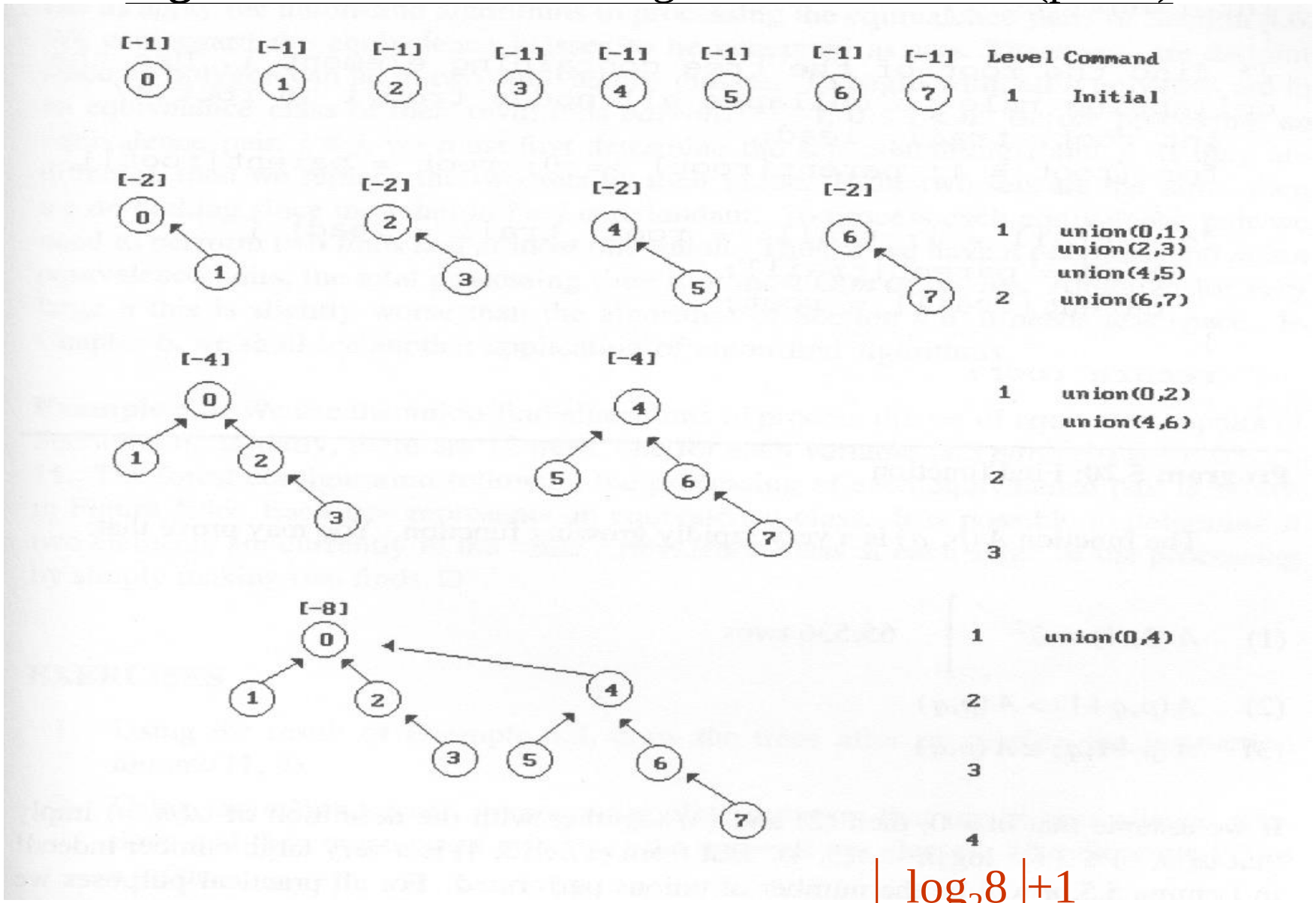
Figure 5.44: Trees obtained using the weighting rule

Modified Union Operation

```
void union2(int i, int j)
{
        int temp = parent[i]+parent[j];          Keep a count in the root of tree
         if (parent[i]>parent[j]) {              i has fewer nodes.
                parent[i]=j;
                parent[j]=temp;
        }
        else {                  j has fewer nodes
                parent[j]=i;
                parent[i]=temp;
        }
}
```

If the number of nodes in tree i is less than the number in tree j, then make j the parent of i; otherwise make i the parent of j.

Figure 5.45: Trees achieving worst case bound (p.245)

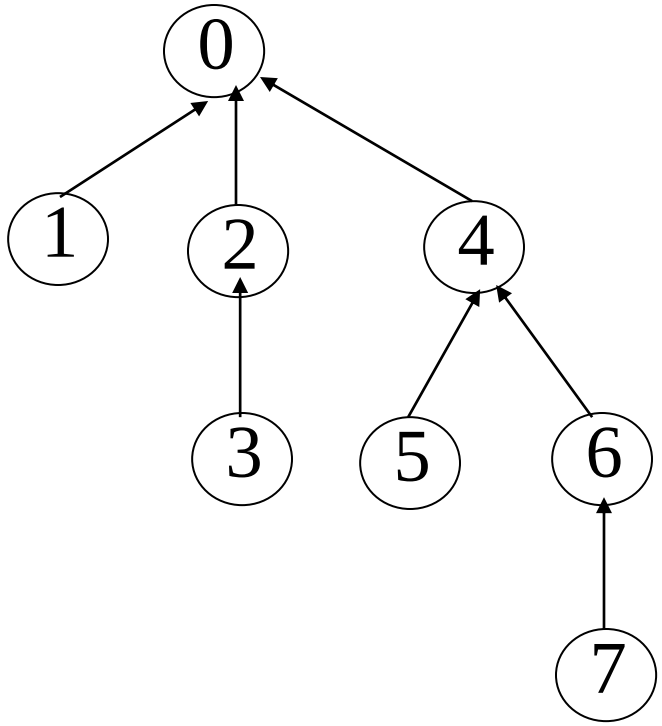


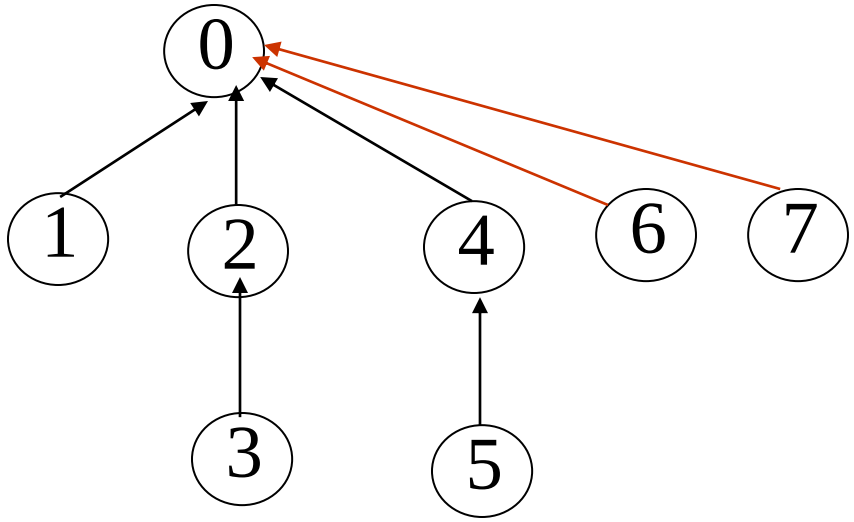
$\lfloor \log_2 8 \rfloor + 1$

Modified *Find*(*i*) Operation

```
int find2(int i)
{
        int root, trail, lead;
        for (root=i; parent[root]>=0; root=parent[root]);
        for (trail=i; trail!=root; trail=lead) {
                lead = parent[trail];
                parent[trail]= root;
        }
    return root:
}
```

If j is a node on the path from i to its root then make j a child of the root





find(7) find(7) find(7) find(7) find(7) find(7) find(7) find(7)

go up	3	1	1	1	1	1	1	1
reset	2							

12 moves (vs. 24 moves)

Applications

- Find equivalence class $i \equiv j$
- 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}

preorder: A B C D E F G H I
inorder: B C A E D G H F I