M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Data Structures
Course Code: CS32
Credits: 3:1:0

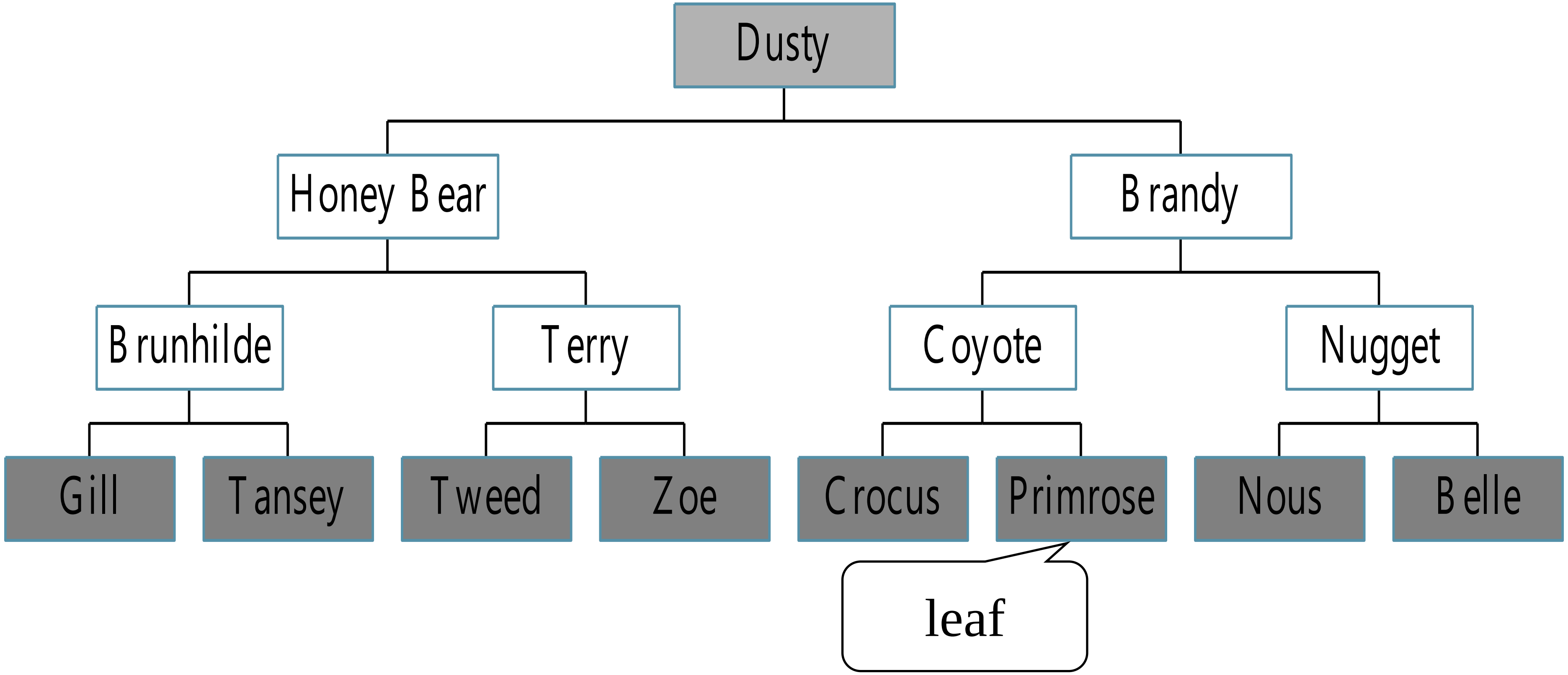
Term: September – December 2020

Faculty:
Vandana S Sardar
Mamatha Jadhav V
A Parkavi

Unit 4: Trees

The contents in this presentation are selected from
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”, Universities Press, 2008

Trees


Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

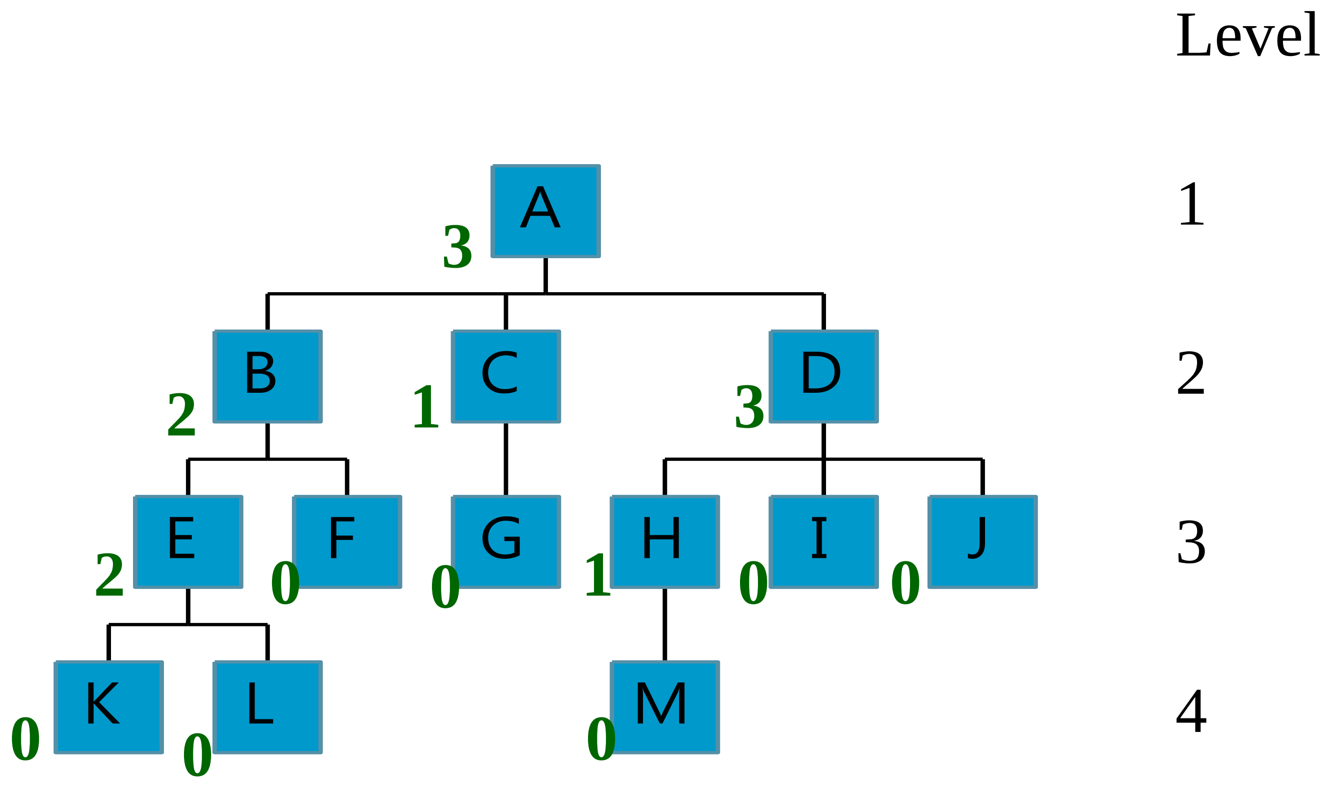
Definition of Tree

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 $T_1, \ldots, T_n$, where each of these sets is a tree.
- We call $T_1, \ldots, T_n$ the subtrees of the root.

Level and Depth

node (13)
degree of a node
leaf (terminal)
nonterminal
parent
children
sibling
degree of a tree (3)
ancestor
level of a node
height of a tree (4)



Terminology

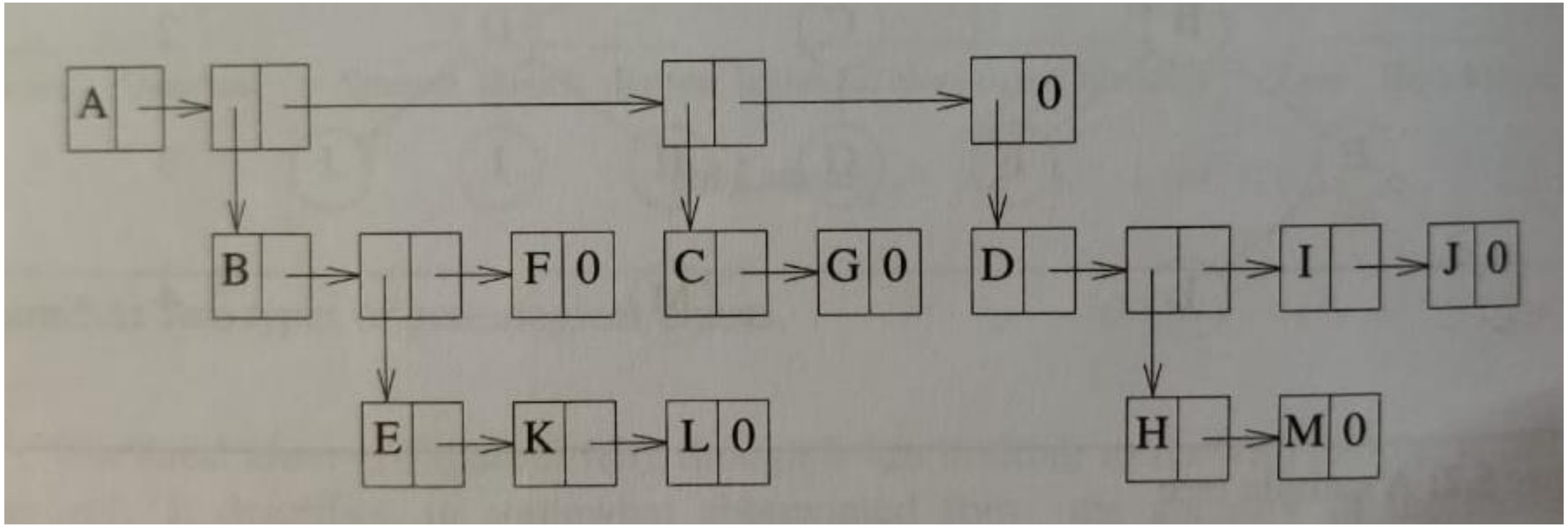
- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.

Representation of Trees

- 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

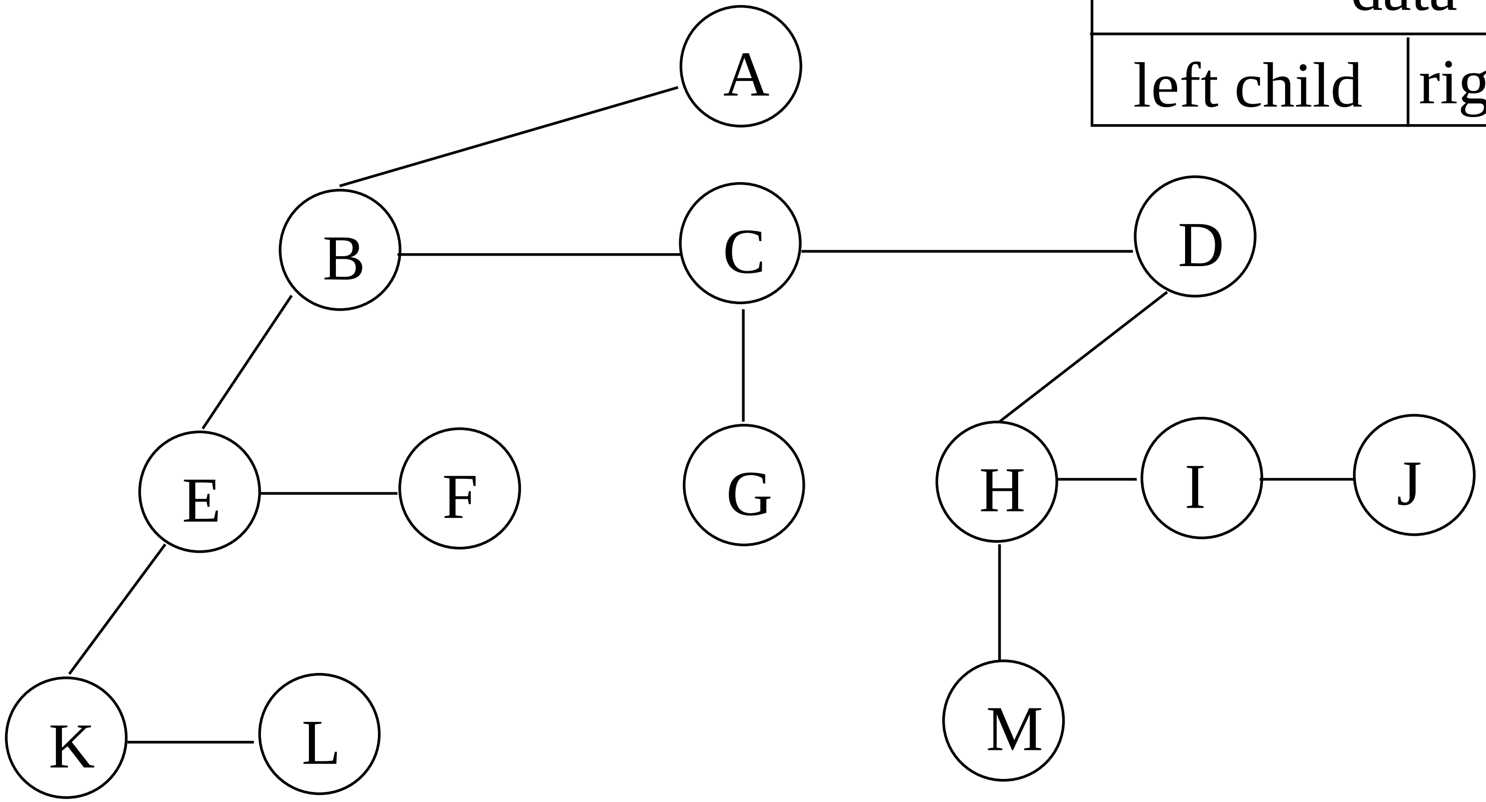
data	child 1	child 2	...	child n

List Representation of the tree



Left Child - Right Sibling

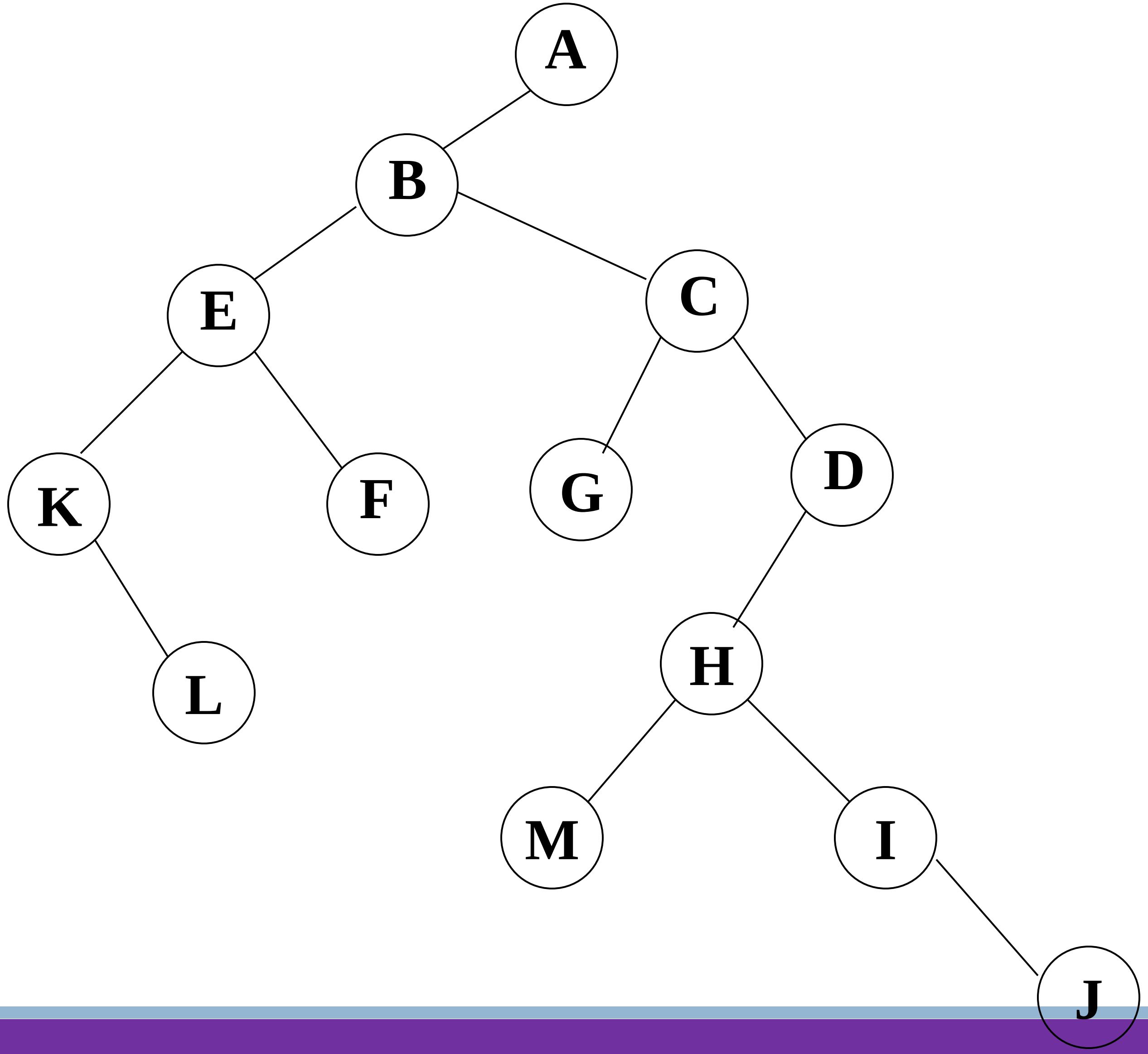
data	
left child	right sibling



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.

Left child-right child tree representation of a tree



Abstract Data Type Binary_Tree

ADT *Binary_Tree*(abbreviated *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 binary tree) return *TRUE*
else return *FALSE*

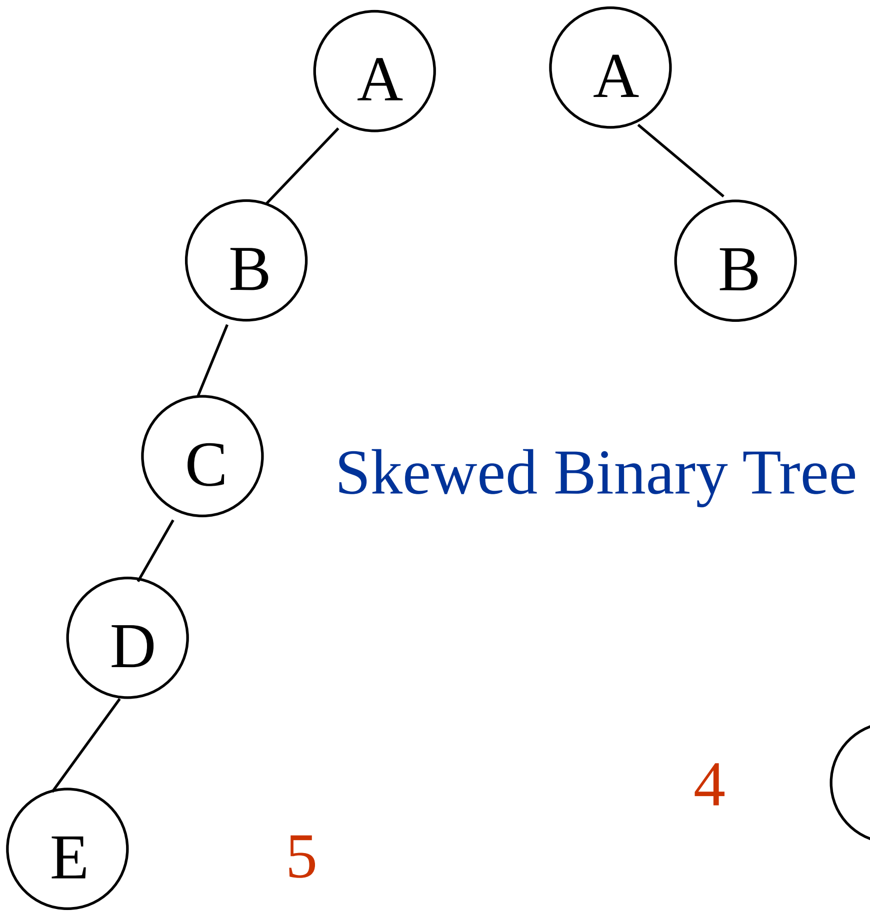
BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*

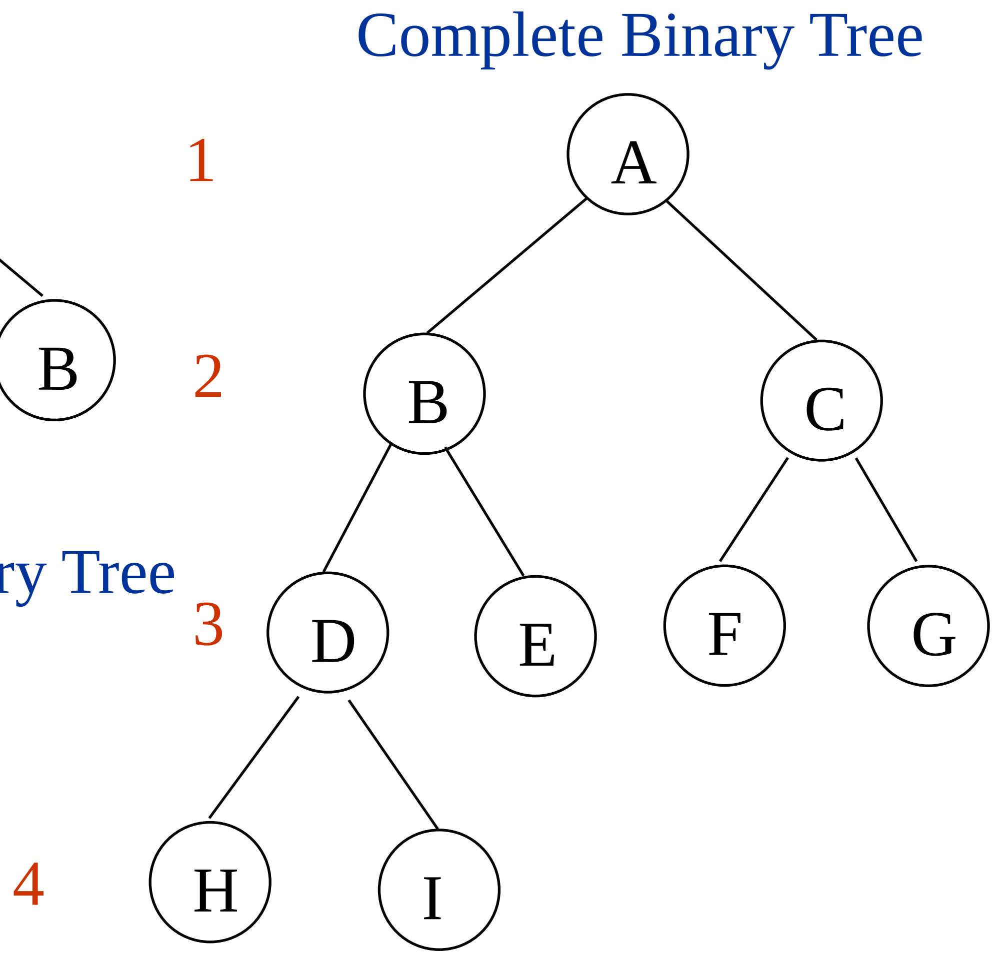
Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the left subtree of *bt*

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





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 nubmer of nodes in a binary tree of depth k is 2^k-1, $k>=1$.

Prove by induction.

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

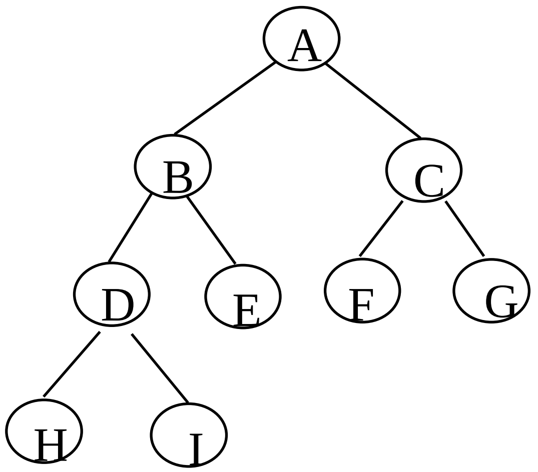
Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n_0 *is the number of leaf nodes and* n_2 *the number of nodes of degree 2, then* $n_0 = n_2 + 1$

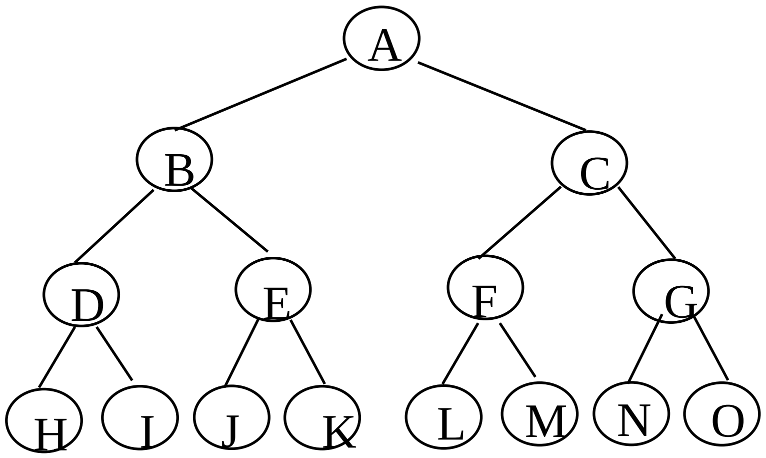
Full BT VS Complete BT

- A full binary tree of depth *k* is a binary tree of depth *k* having 2^k -1 nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

Complete binary tree



Full binary tree of depth 4

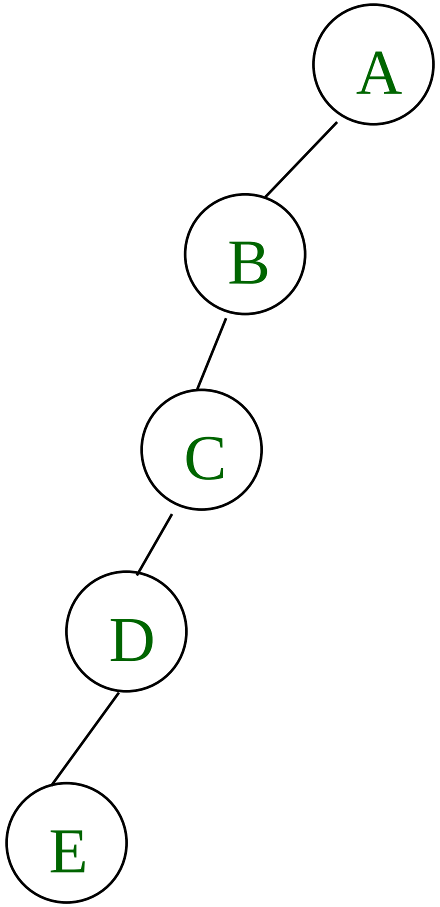


RAMAIAH
Institute of Technology

Binary Tree Representations

- 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.
  - *leftChild*(i) ia at 2i if 2i<=n. If 2i>n, then i has no left child.
  - *rightChild*(i) ia at 2i+1 if 2i +1 <=n. If 2i +1 >n, then i has no right child.

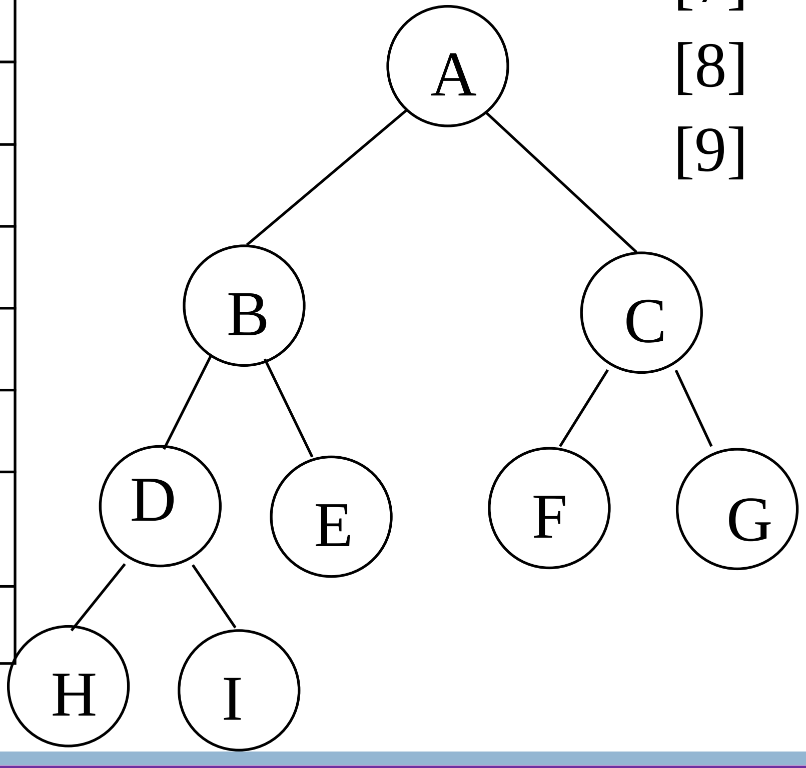
Sequential Representation



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

(1) waste space

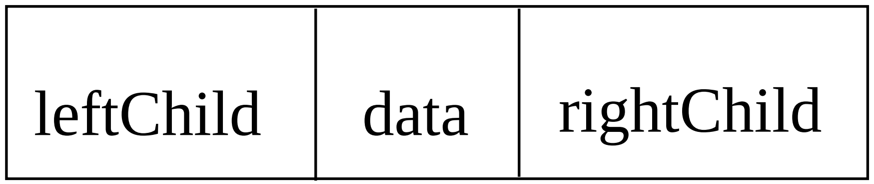
(2) insertion/deletion problem

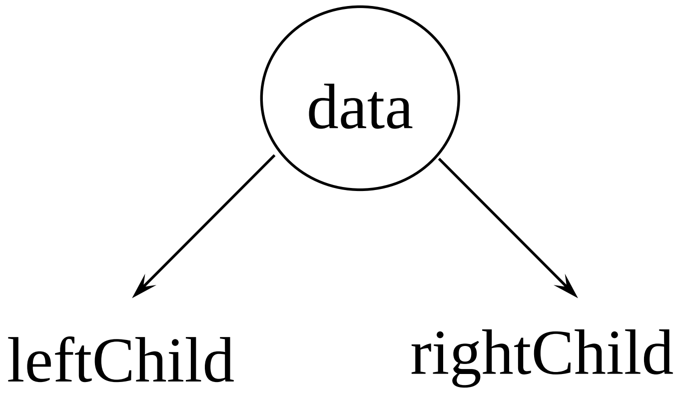


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

```
typedef struct node *treePointer;
typedef struct node {
 int data;
 treePointer leftChild, rightChild;
};
```

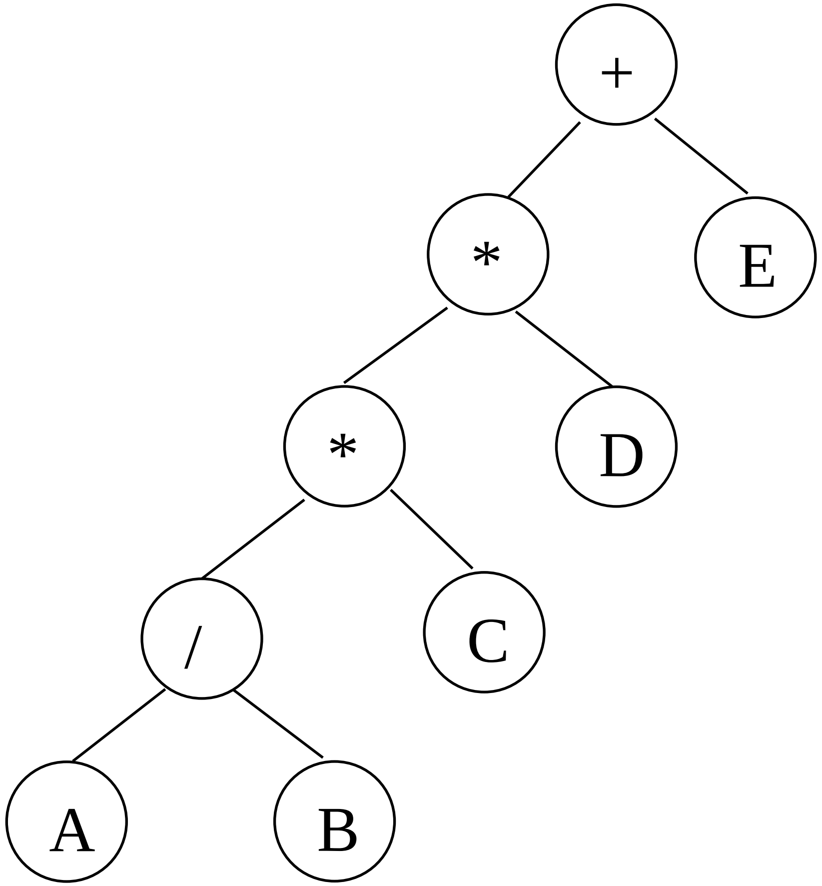




Binary Tree Traversals

- 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(treePointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->leftChild);
        printf("%d", ptr->data);
        indorder(ptr->rightChild);
    }
}
```

A / B * C * D + E

Preorder Traversal (recursive version)

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

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(treePointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->leftChild);
        postdorder(ptr->rightChild);
        printf("%d", ptr->data);
    }
}
```

A B / C * D * E +

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	

Iterative Inorder Traversal

(using stack)

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

Level Order Traversal

(using queue)

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

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

Copying Binary Trees

```
treePointer copy(treePointer original)
{
treePointer temp;
if (original) {
 temp=(treePointer) malloc(sizeof(node));
 temp->leftChild=copy(original->leftChild);
 temp->rightChild=copy(original->rightChild);
 temp->data=original->data;
 return temp;
}
return NULL;
}
```

Equality of Binary Trees

the same topology and data

```
int equal(treePointer first, treePointer 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->leftChild, second->leftChild) &&
       equal(first->rightChild, second->rightChild))
}
```

Propositional Calculus Expression

A variable is an expression.

If x and y are expressions, then $\neg x$, $x \wedge y$,
$x \vee y$ are expressions.

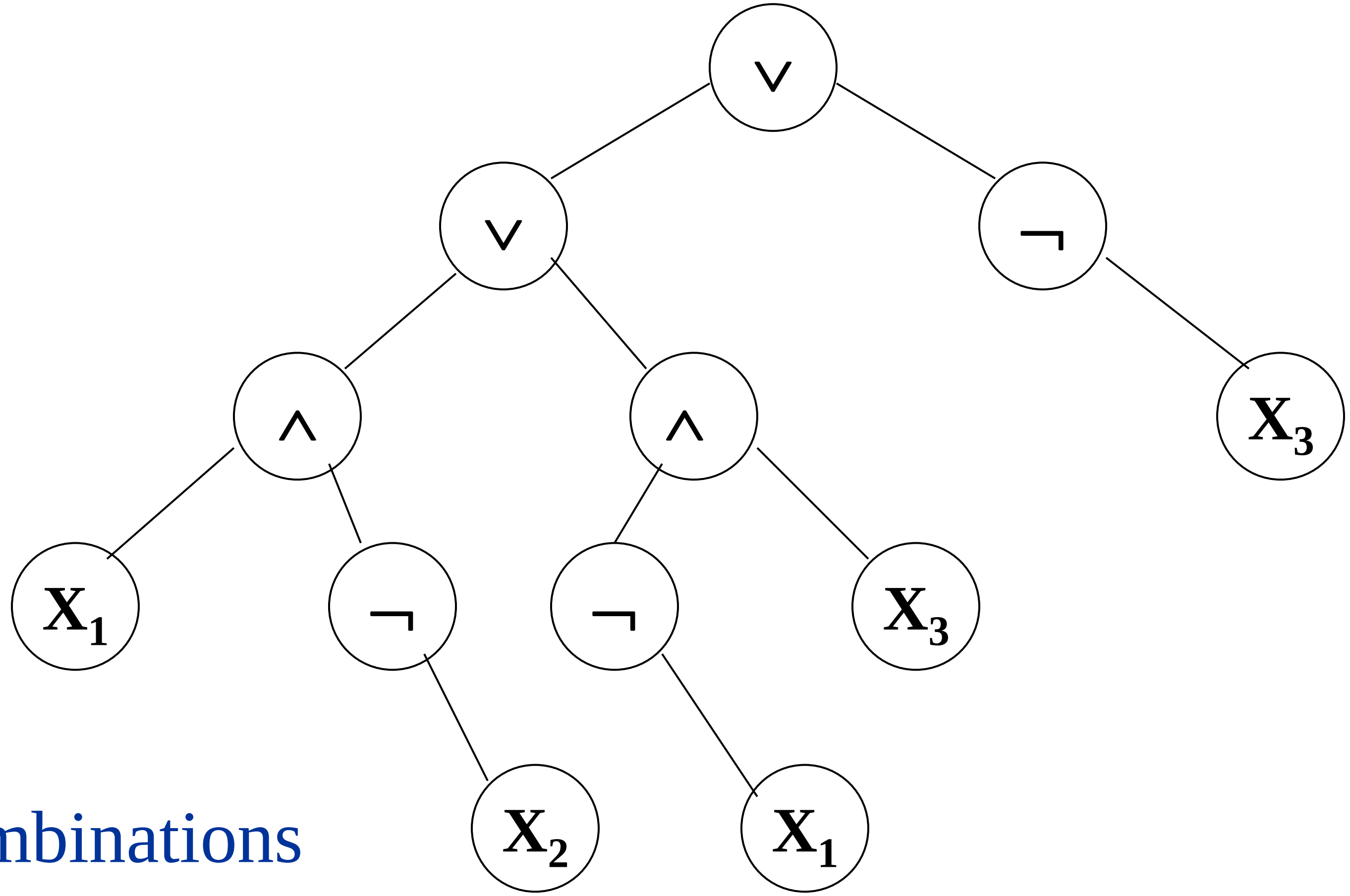
Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$).

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

satisfiability problem: Is there an assignment to make an expression true?

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

(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

leftChild	***data***	***value***	***rightChild***

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

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");
```

RAMAIAH
Institute of Technology

Post-order-eval function

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

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

Threaded Binary Trees

- Two many null pointers in current representation of binary trees

  n: number of nodes

  number of non-null links: n-1

  total links: 2n

  null links: 2n-(n-1)=n+1

- Replace these null pointers with some useful “threads”.

Threaded Binary Trees *(Continued)*

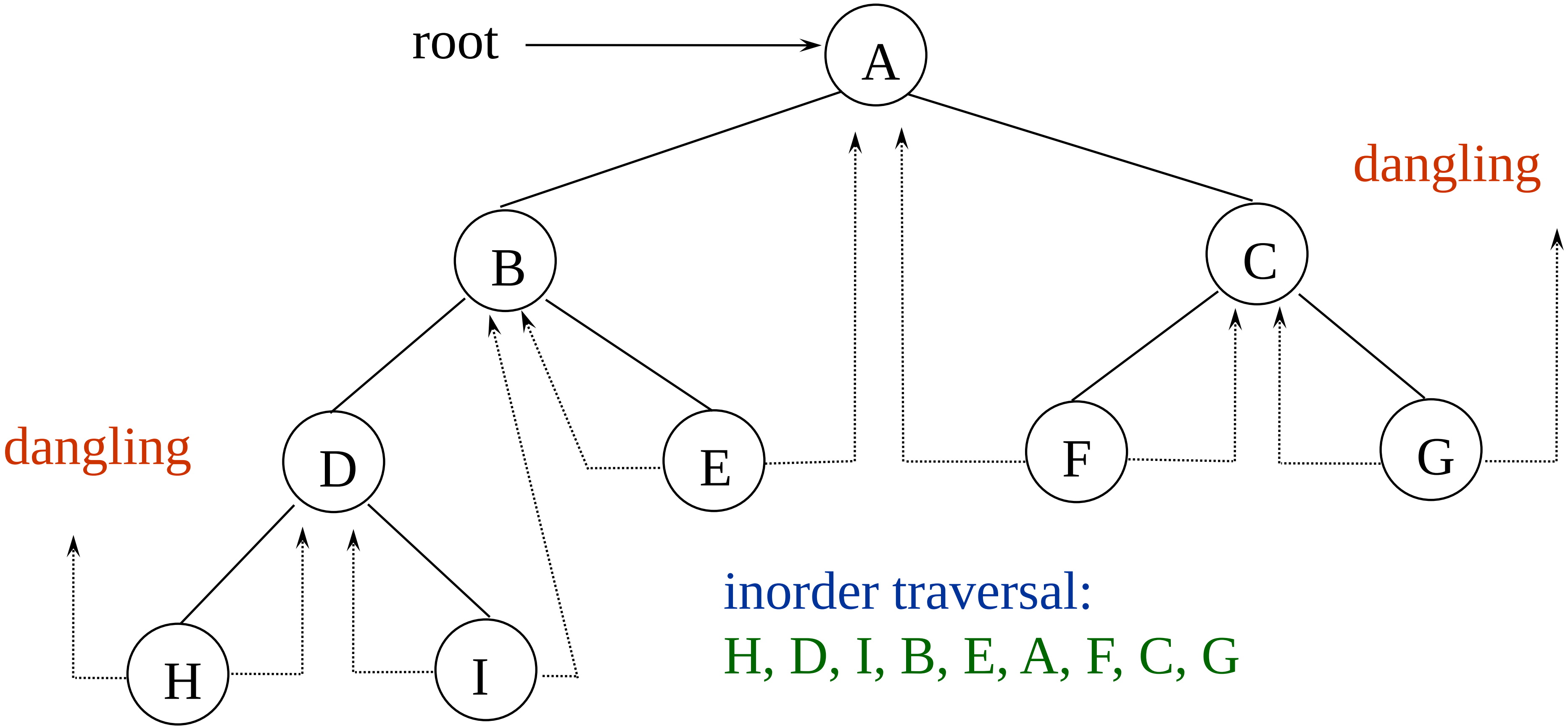
If `ptr->leftChild` is null,

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

If `ptr->rightChild` is null,

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

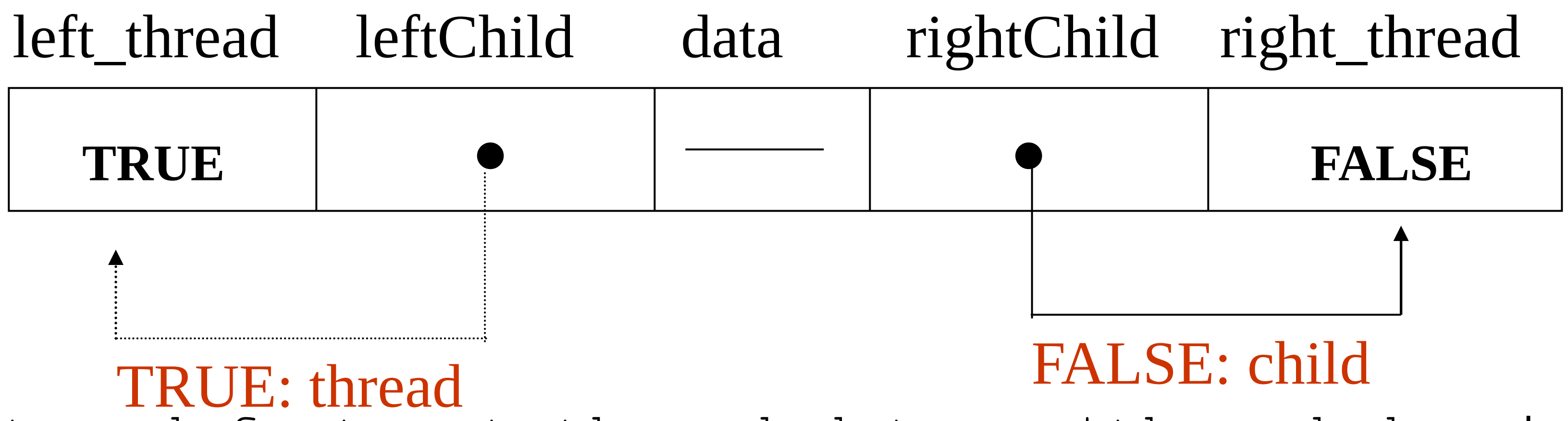
A Threaded Binary Tree



inorder traversal:

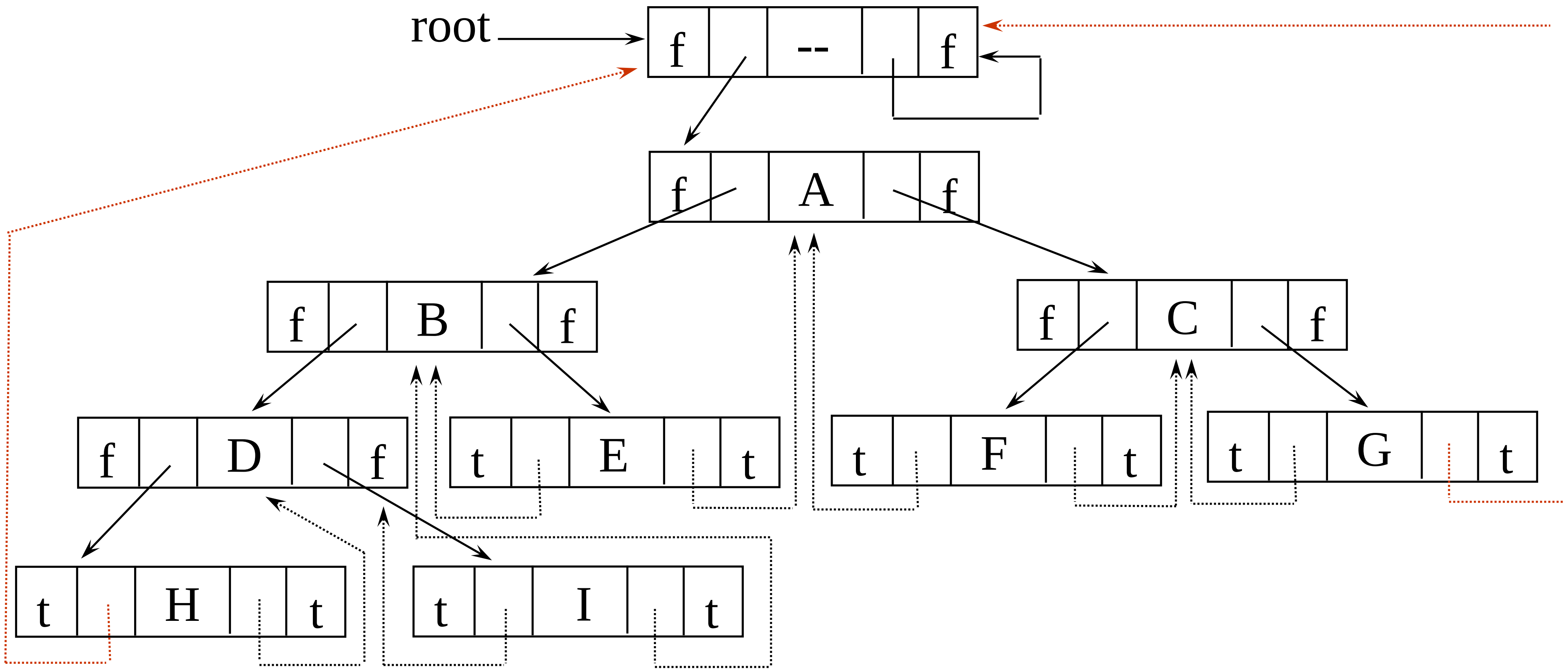
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



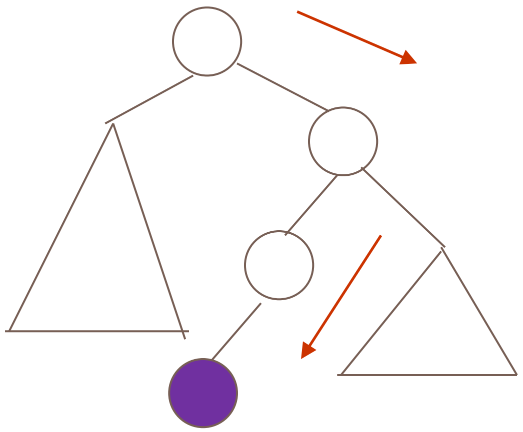

```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer leftChild;
    char data;
    threaded_pointer rightChild;
    short int right_thread;  };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer tree)
{
  threaded_pointer temp;
  temp = tree->rightChild;
  if (!tree->right_thread)
    while (!temp->left_thread)
      temp = temp->leftChild;
  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);
    }
}
```

Inserting Nodes into Threaded BTs

- Insert `child` as the right child of node `parent`
  - change `parent->right_thread` to FALSE
  - set `child->left_thread` and `child->right_thread` to TRUE
  - set `child->leftChild` to point to `parent`
  - set `child->rightChild` to `parent->rightChild`
  - change `parent->rightChild` to point to `child`

Examples

Insert a node D as a right child of B.

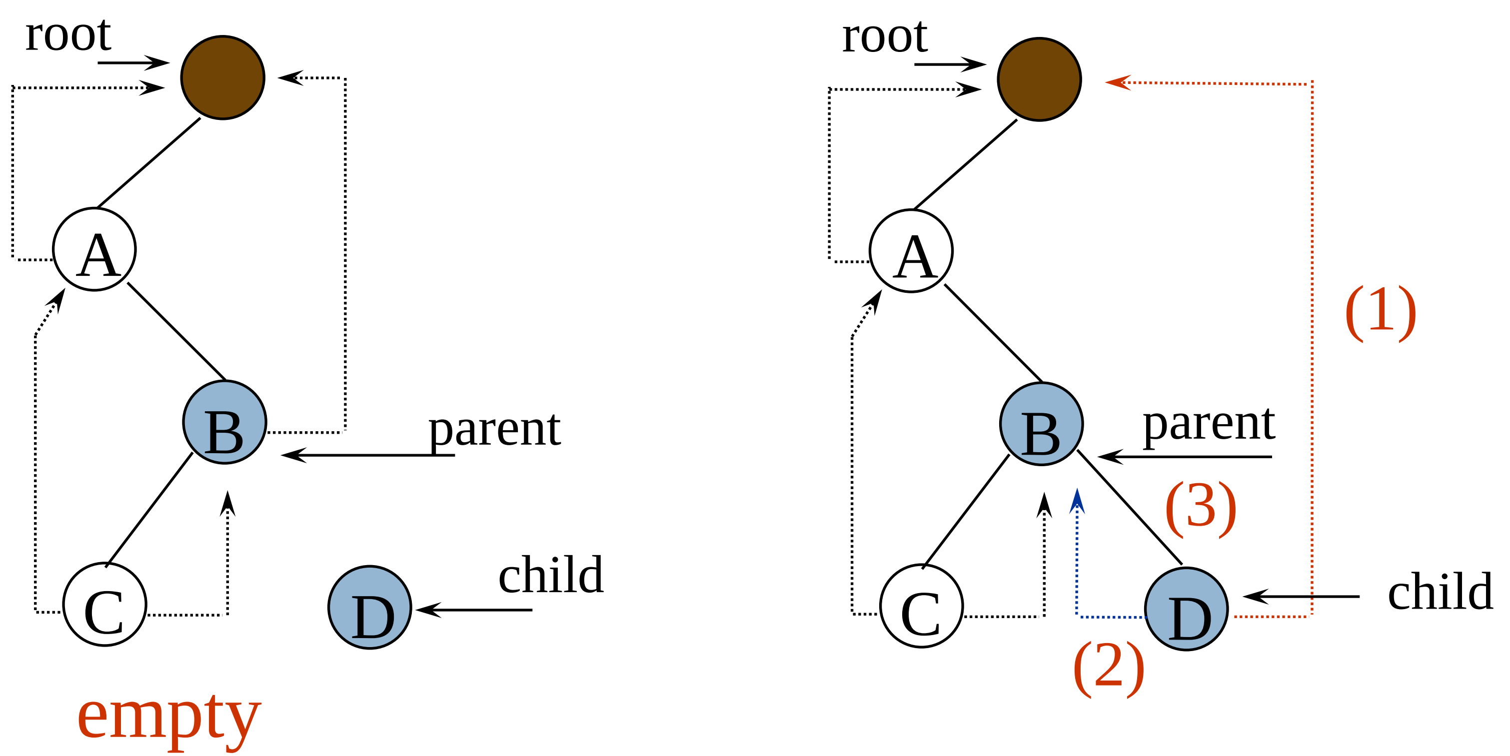
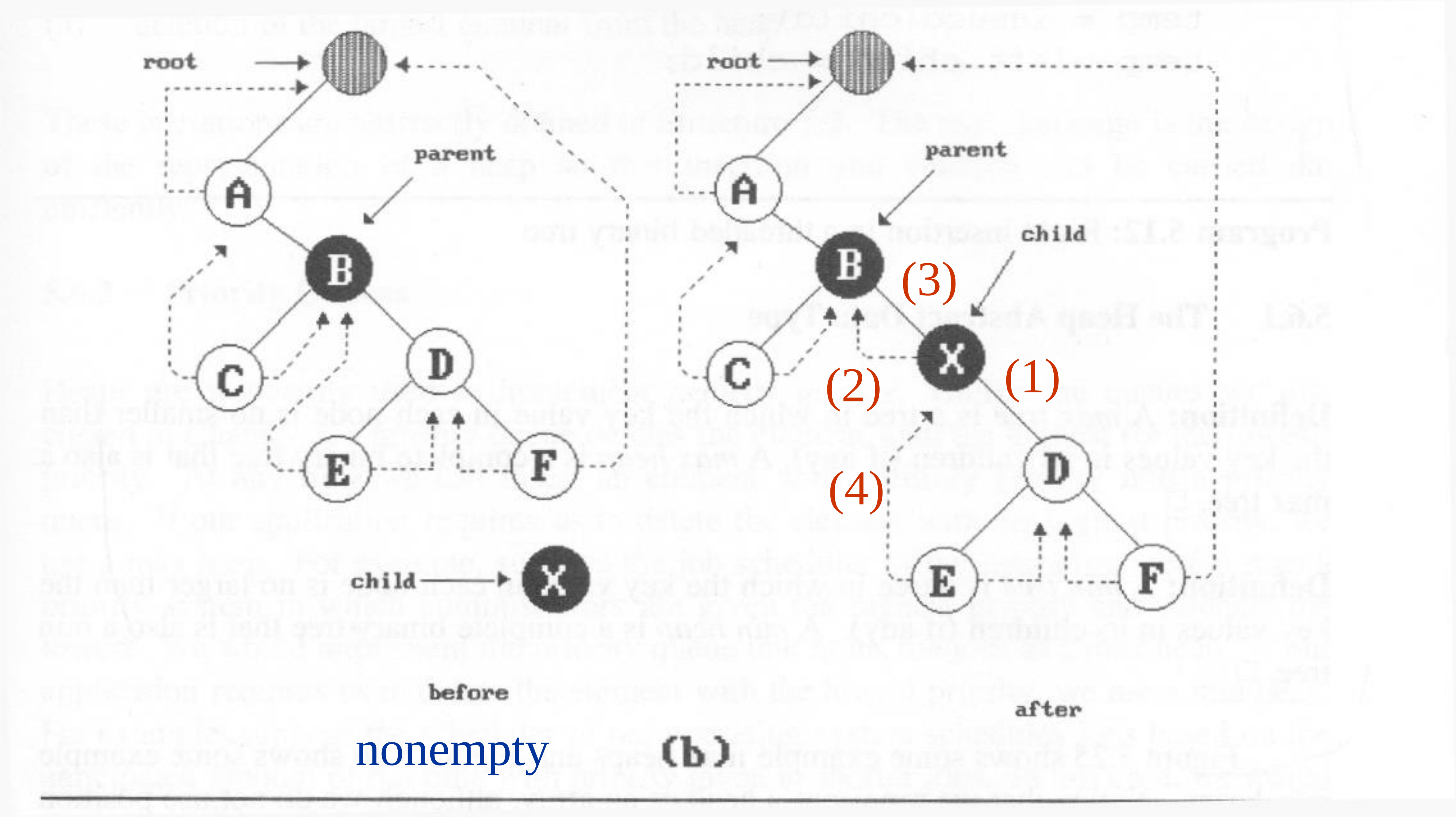


***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.217)



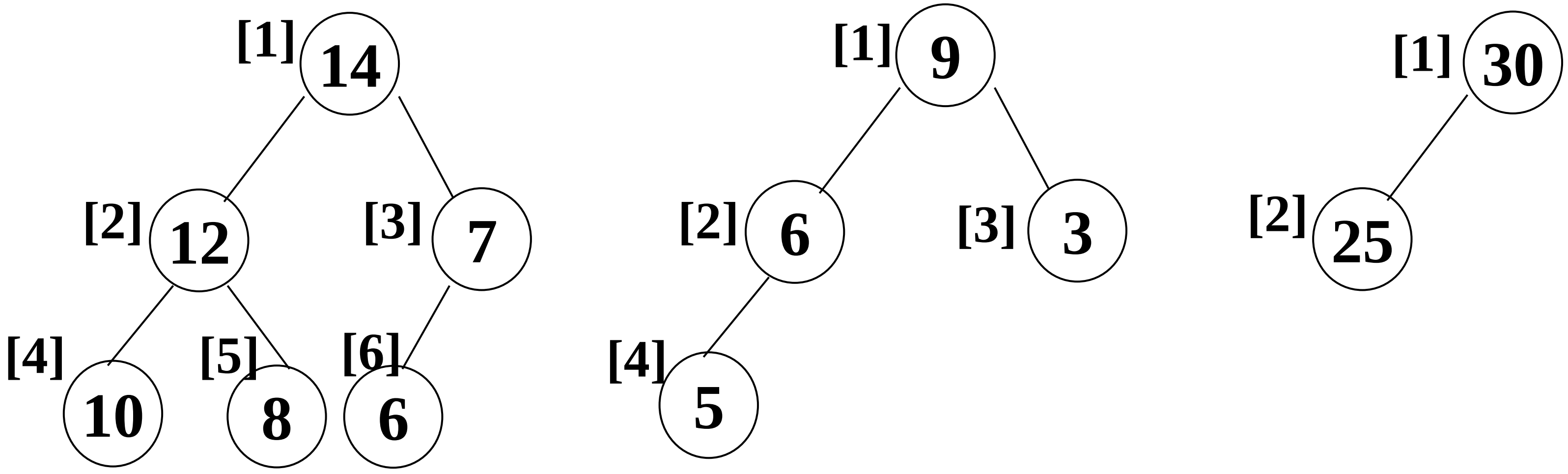
Right Insertion in Threaded BTs

```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
  threaded_pointer temp;
(1) child->rightChild = parent->rightChild;
  child->right_thread = parent->right_thread;
  child->leftChild = parent;  case (a)
(2) child->left_thread = TRUE;
  parent->rightChild = child;
(3) parent->right_thread = FALSE;
  if (!child->right_thread) { case (b)
(4)  temp = insucc(child);
     temp->leftChild = child;
  }
}
```

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

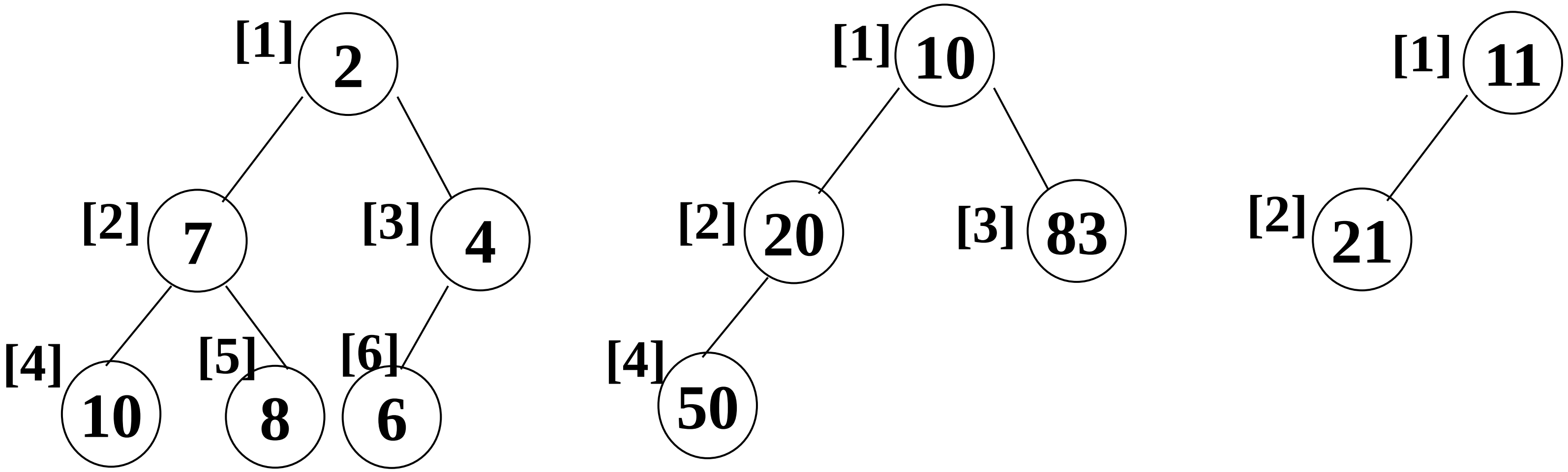
Sample max heaps



Property:

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

Sample min heaps



ADT for Max Heap

structure MaxHeap

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

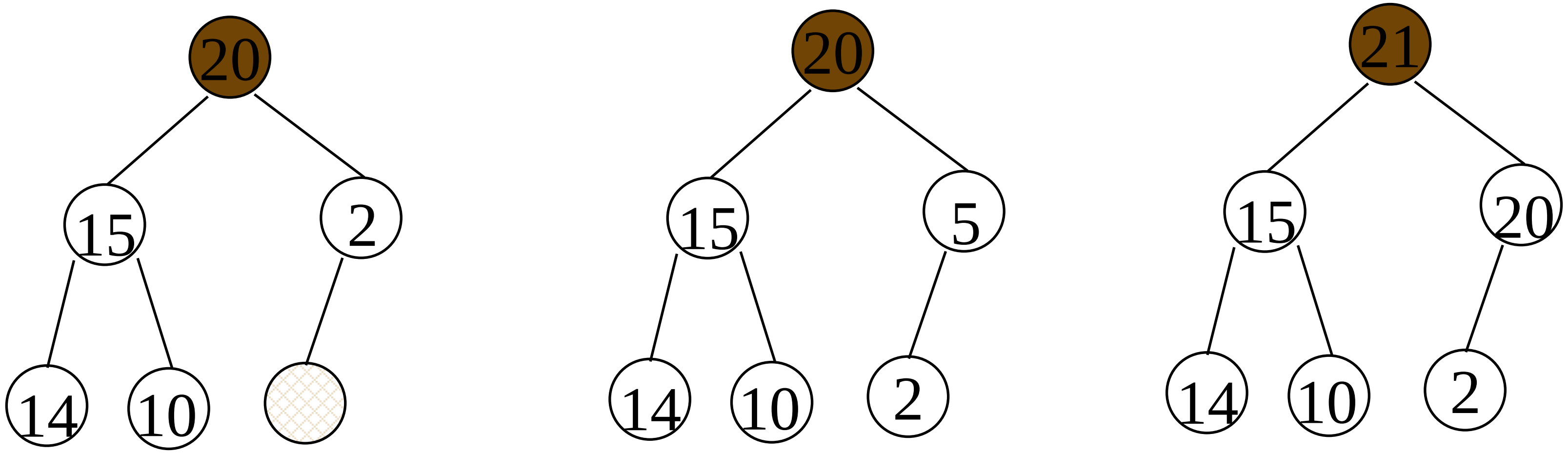
Machine service

- 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

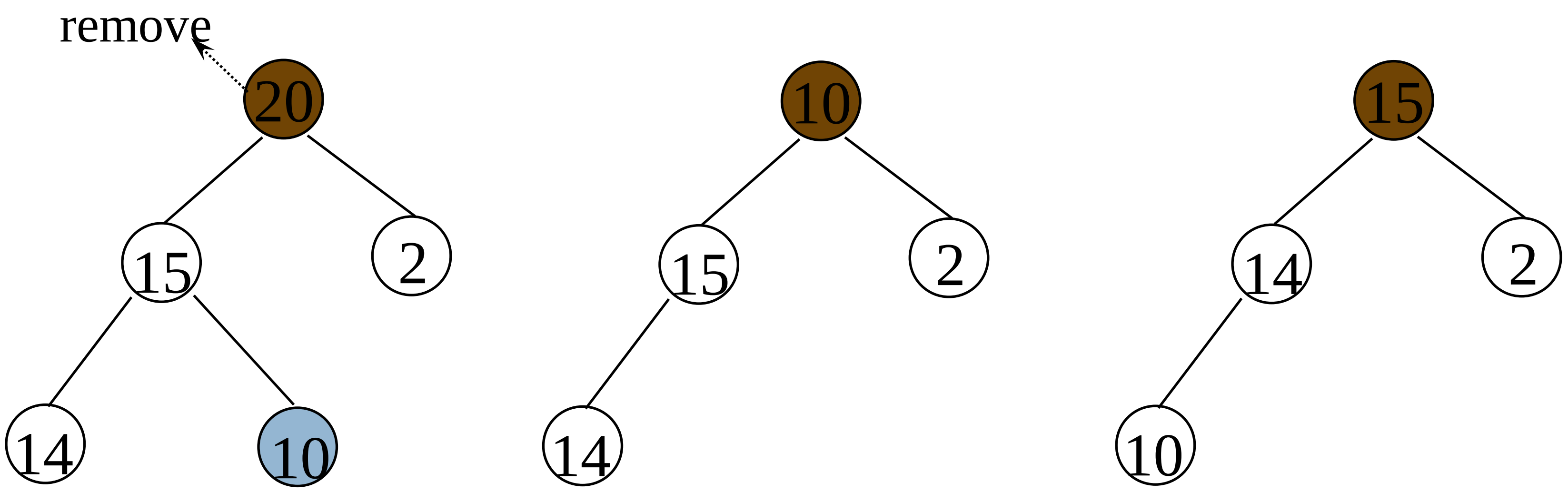
Insertion into a Max Heap

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

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

$O(\log_2 n)$

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;
}
```

RAMAIAH
Institute of Technology

```
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];
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

Binary Search Tree

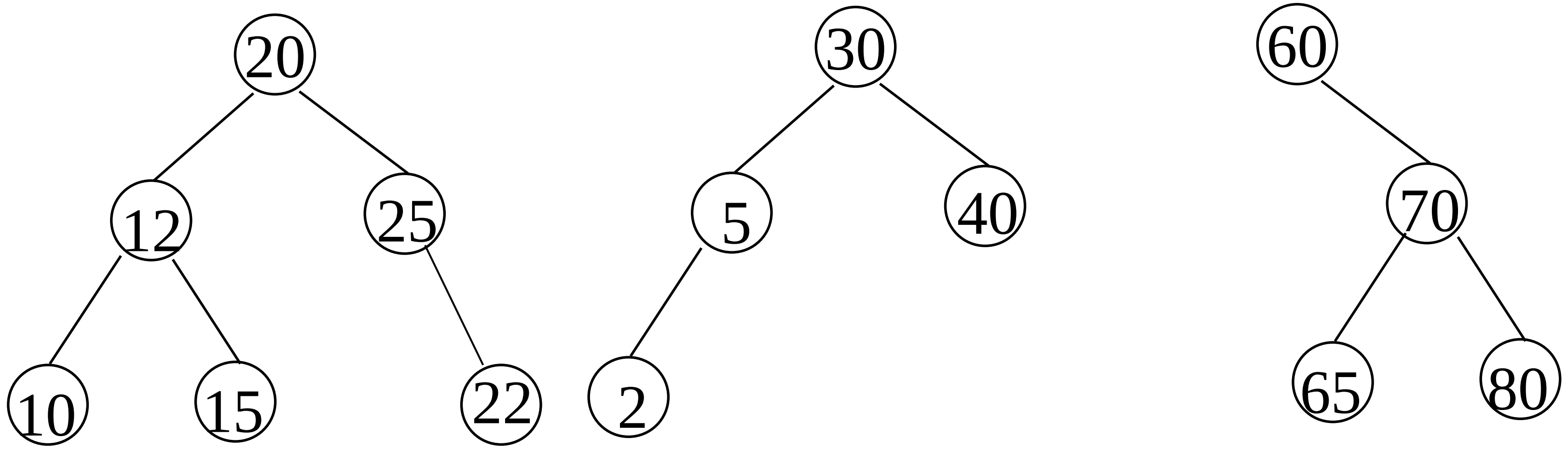
Heap

- a min (max) element is deleted. $O(\log_2 n)$
- deletion of an arbitrary element $O(n)$
- search for an arbitrary element $O(n)$

Binary search tree

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

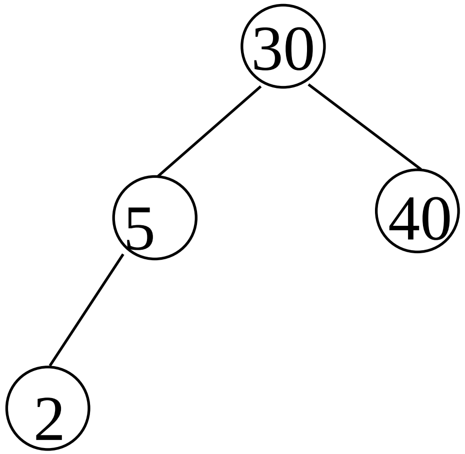
```
treePointer search(treePointer root,
             int key)
{
/* return a pointer to the node that contains key. If there
  is no such
  node, return NULL */

  if (!root) return NULL;
  if (key == root->data) return root;
  if (key < root->data)
      return search(root->leftChild,
                    key);
  return search(root->rightChild,key);
}
```

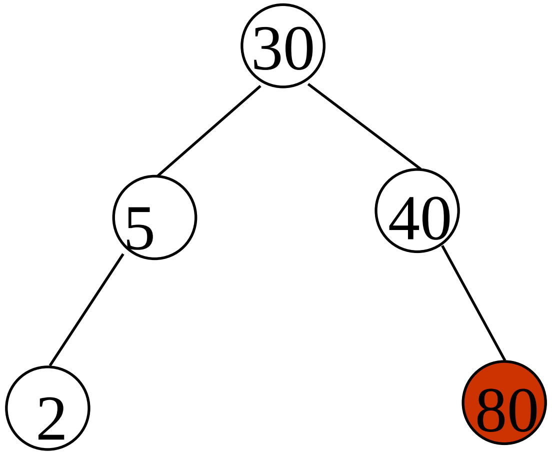
Another Searching Algorithm

```
treePointer search2(treePointer tree, int key)
{
  while (tree) {
    if (key == tree->data) return tree;
    if (key < tree->data)
        tree = tree->leftChild;
    else tree = tree->rightChild;
  }
  return NULL;
}
```

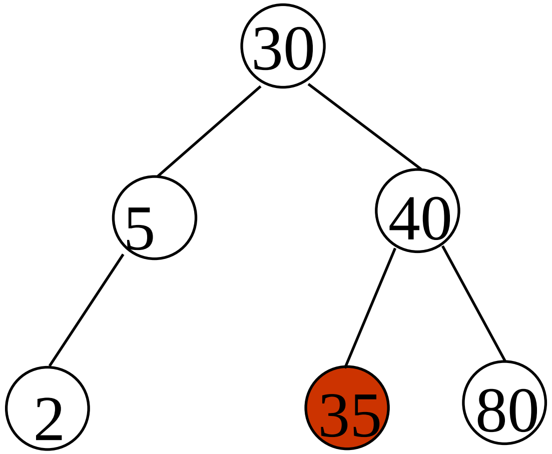
O(h)

Insert Node in Binary Search Tree





Insert 80

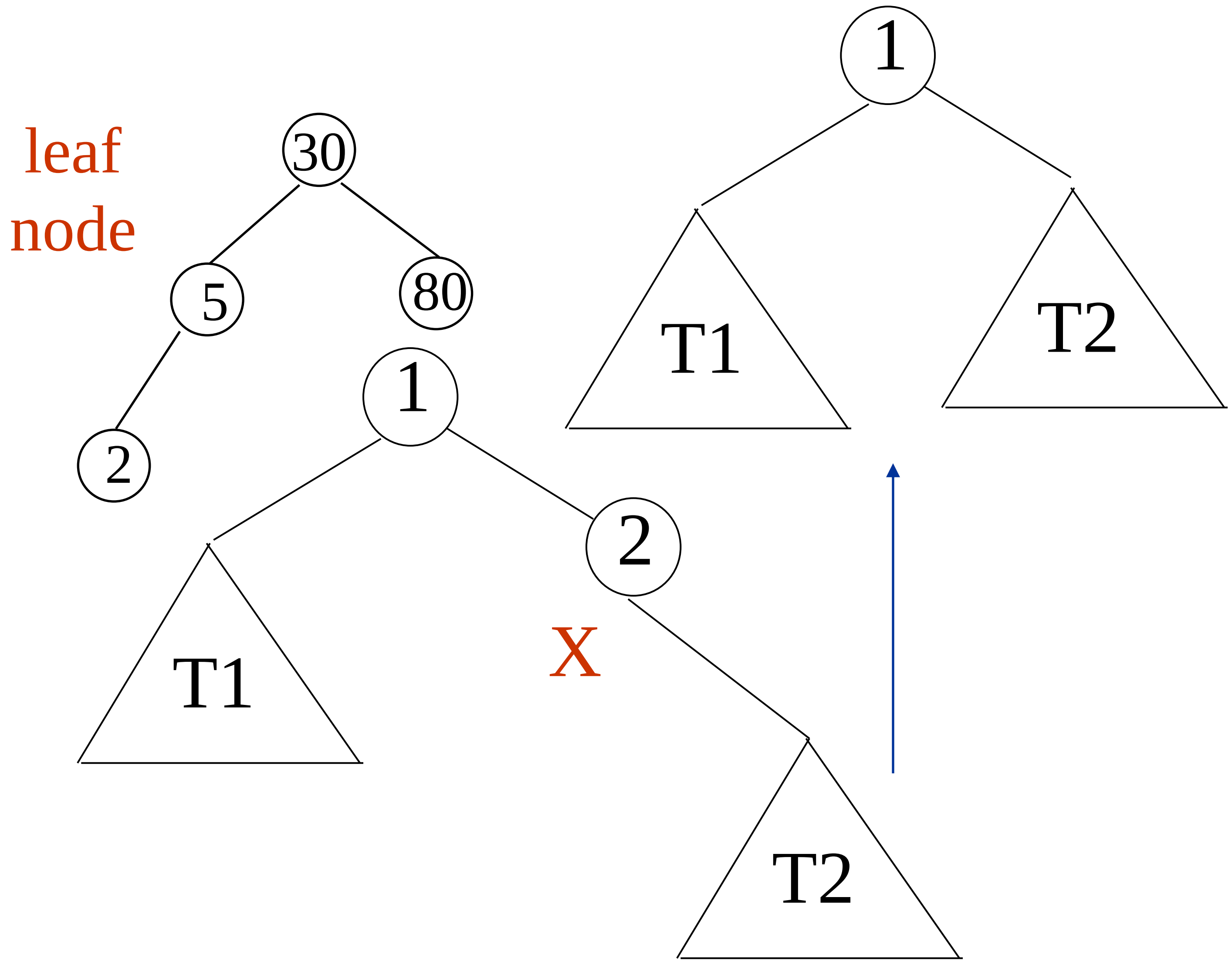


Insert 35

Insertion into A Binary Search Tree

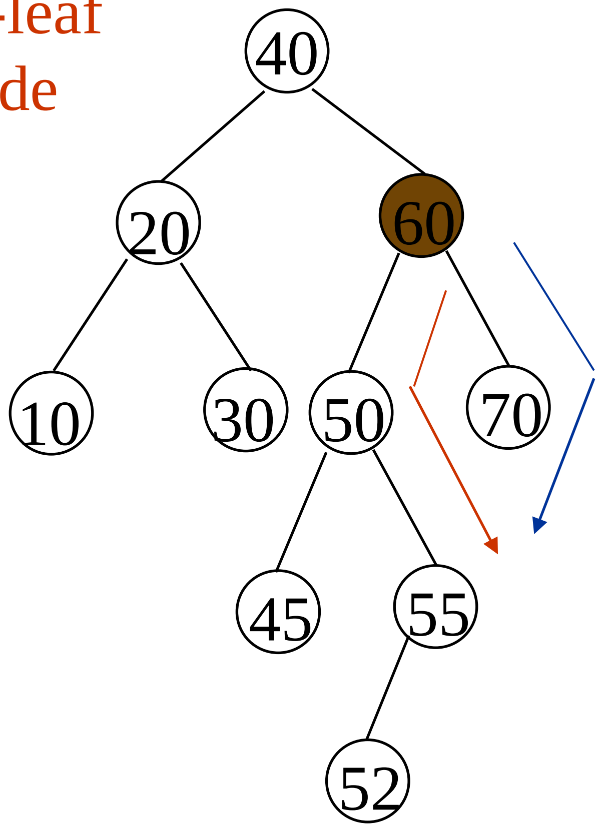
```
void insert_node(treePointer *node, int num)
{treePointer ptr,
      temp = modified_search(*node, num);
  if (temp || !(*node)) {
   ptr = (treePointer) malloc(sizeof(node));
   if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->data = num;
   ptr->leftChild = ptr->rightChild = NULL;
   if (*node)
     if (num<temp->data) temp->leftChild=ptr;
        else temp->rightChild = ptr;
   else *node = ptr;
  }
}
```

Deletion for A Binary Search Tree

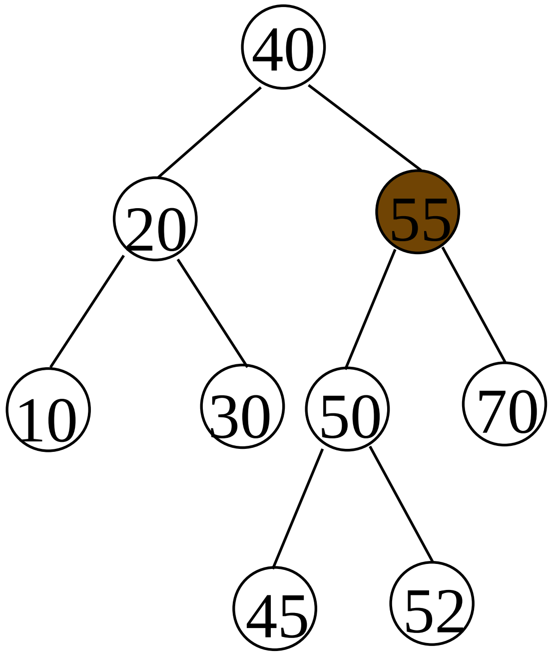


Deletion for A Binary Search Tree

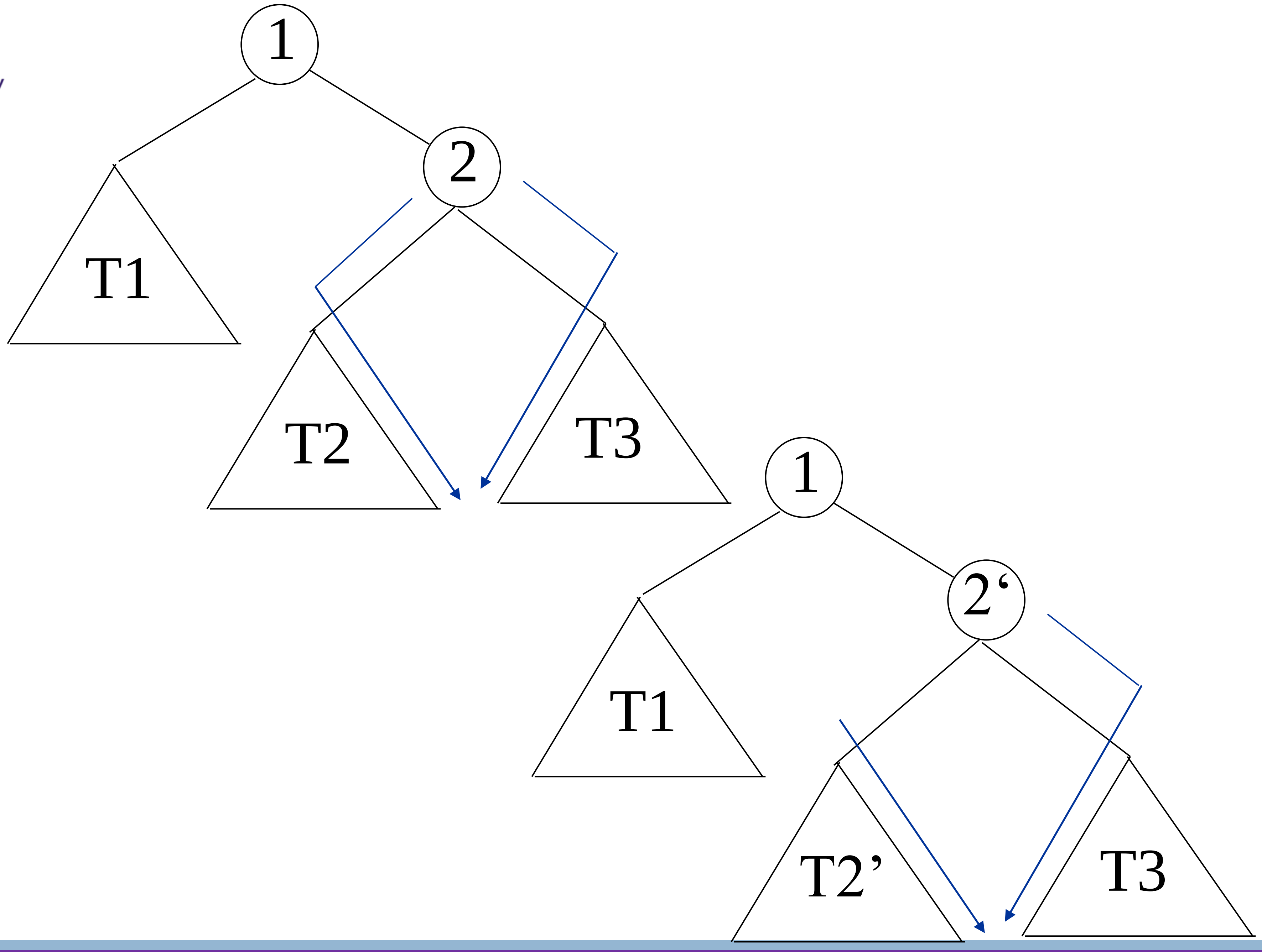
non-leaf node



Before deleting 60



After deleting 60


1
2
T1
T2
T3
1
2'
T1
T2'
T3

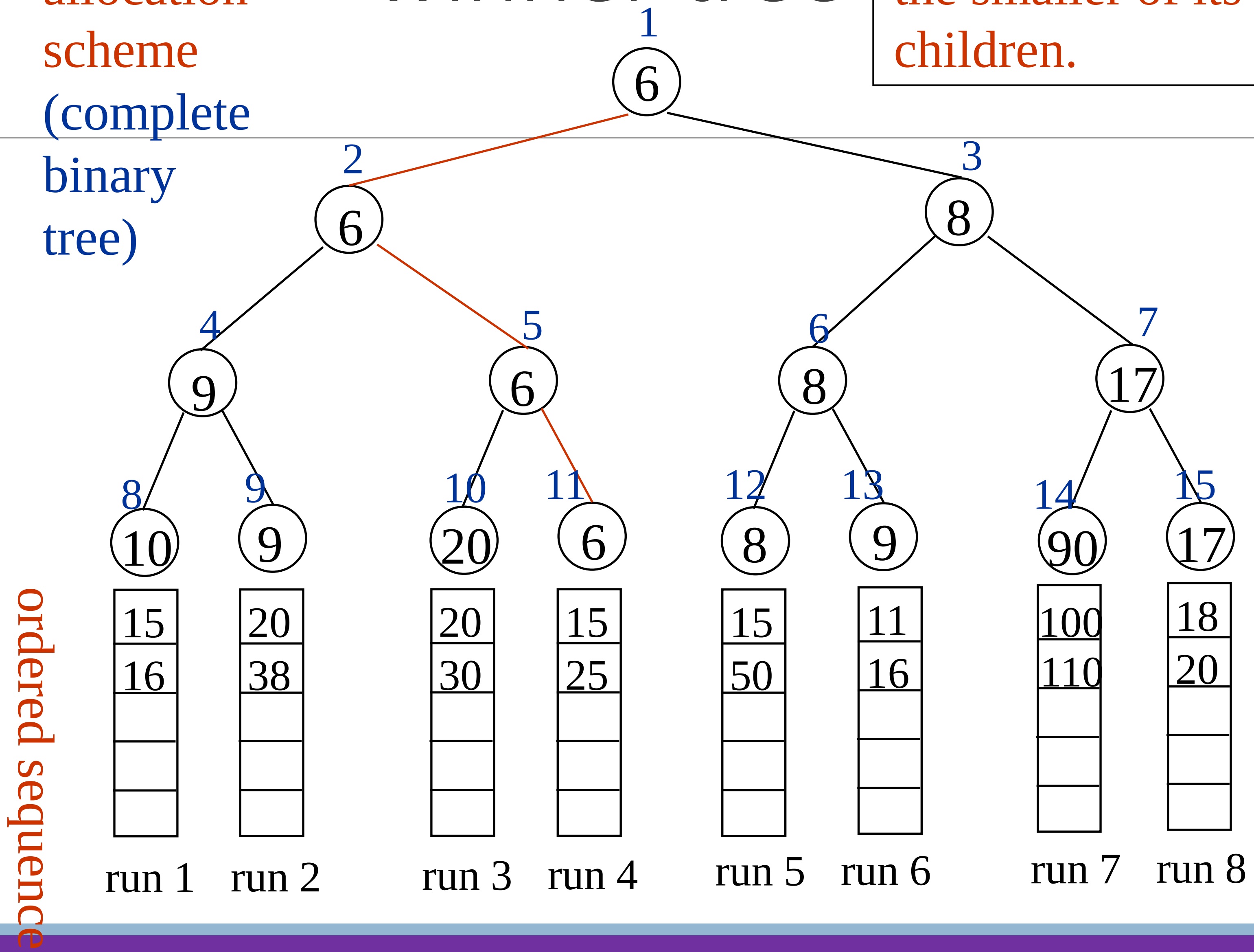
Selection Trees

(1) Winner tree
(2) Loser tree

winner tree

sequential allocation scheme (complete binary tree)

Each node represents the smaller of its two children.



RAMAIAH Institute of Technology

Selection tree of after one record has been output and the tree restructured

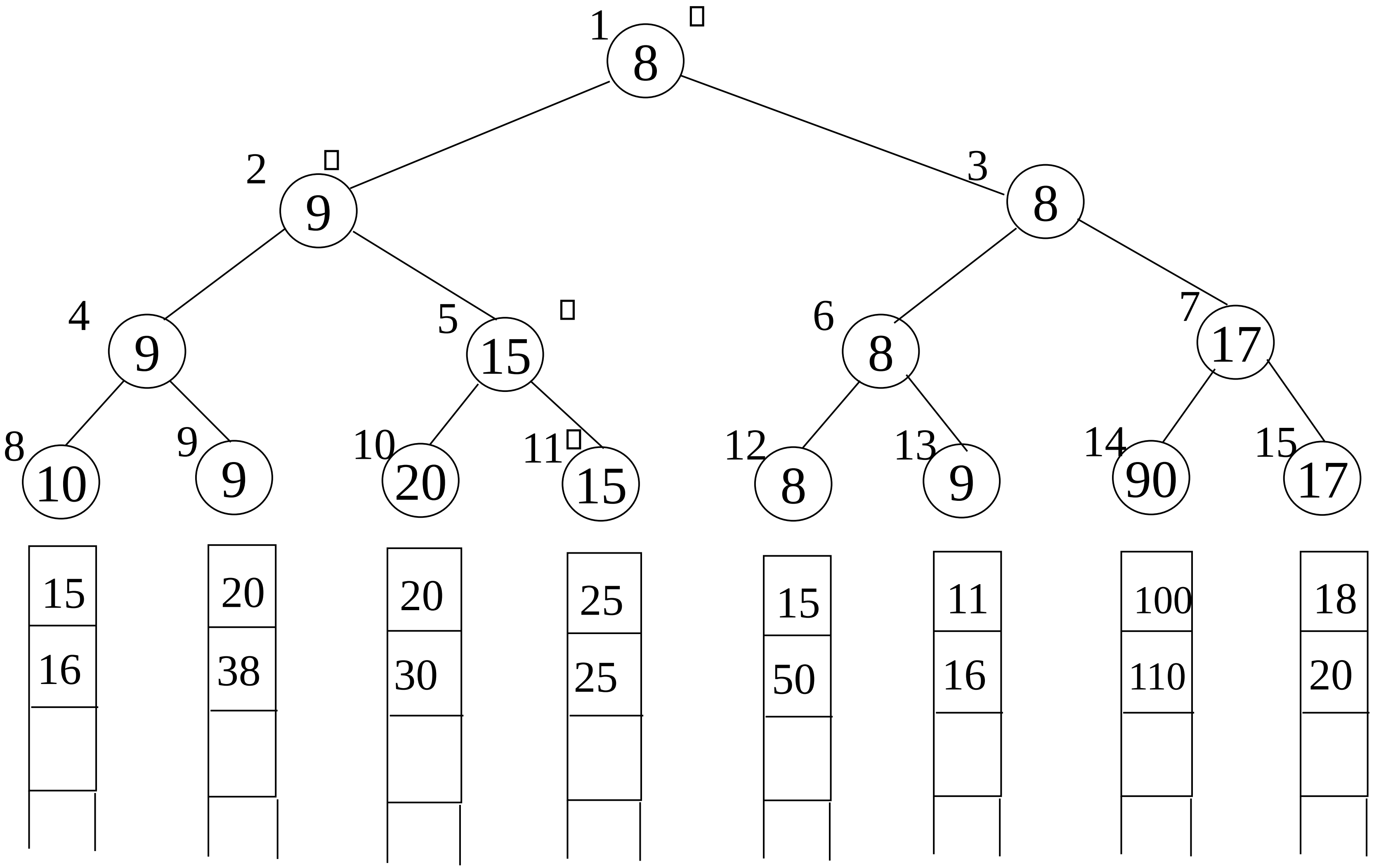
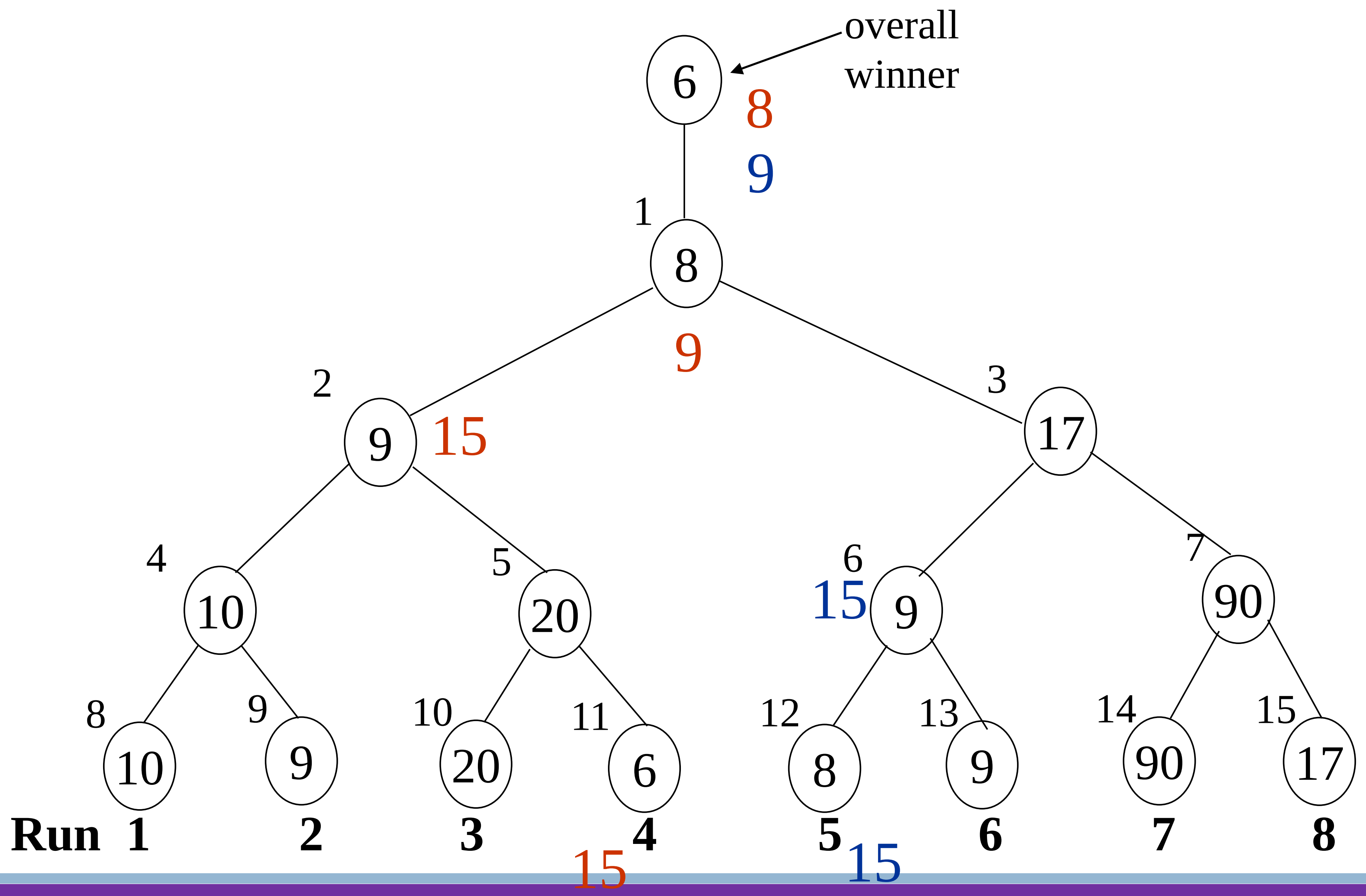
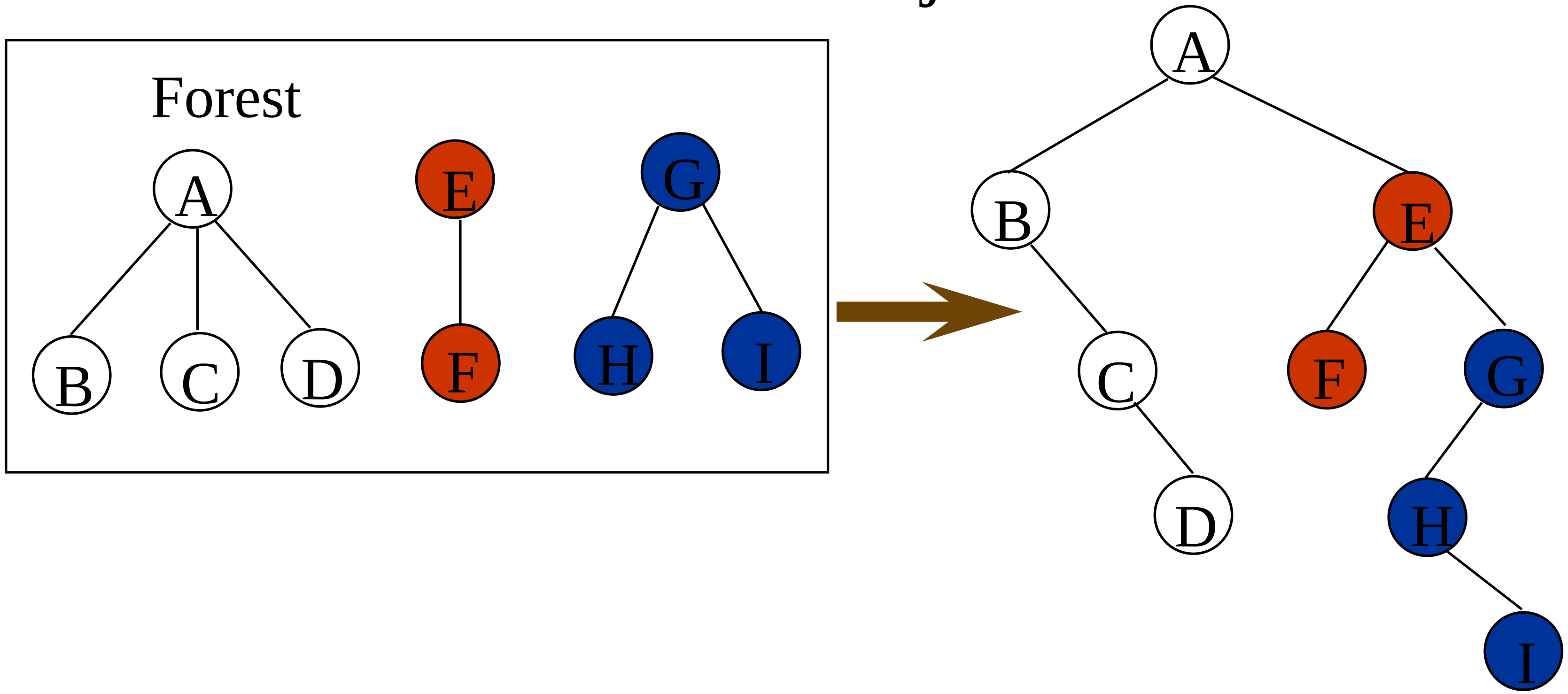


*Figure 5.36: Tree of losers corresponding to Figure 5.34 (p.235)



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

preorder: A B C D E F G H I

inorder: B C A E D G H F I