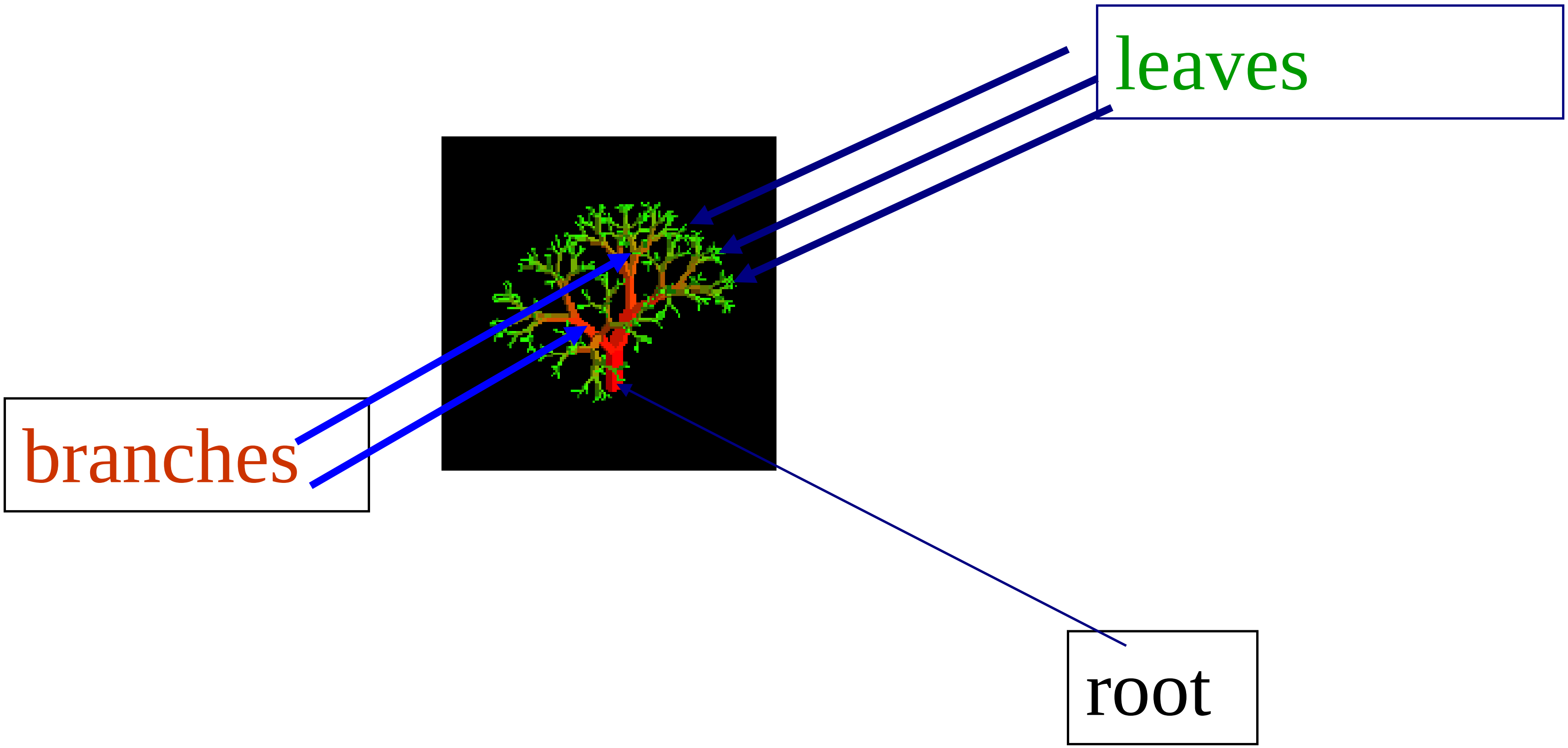
CHAPTER 5

Trees

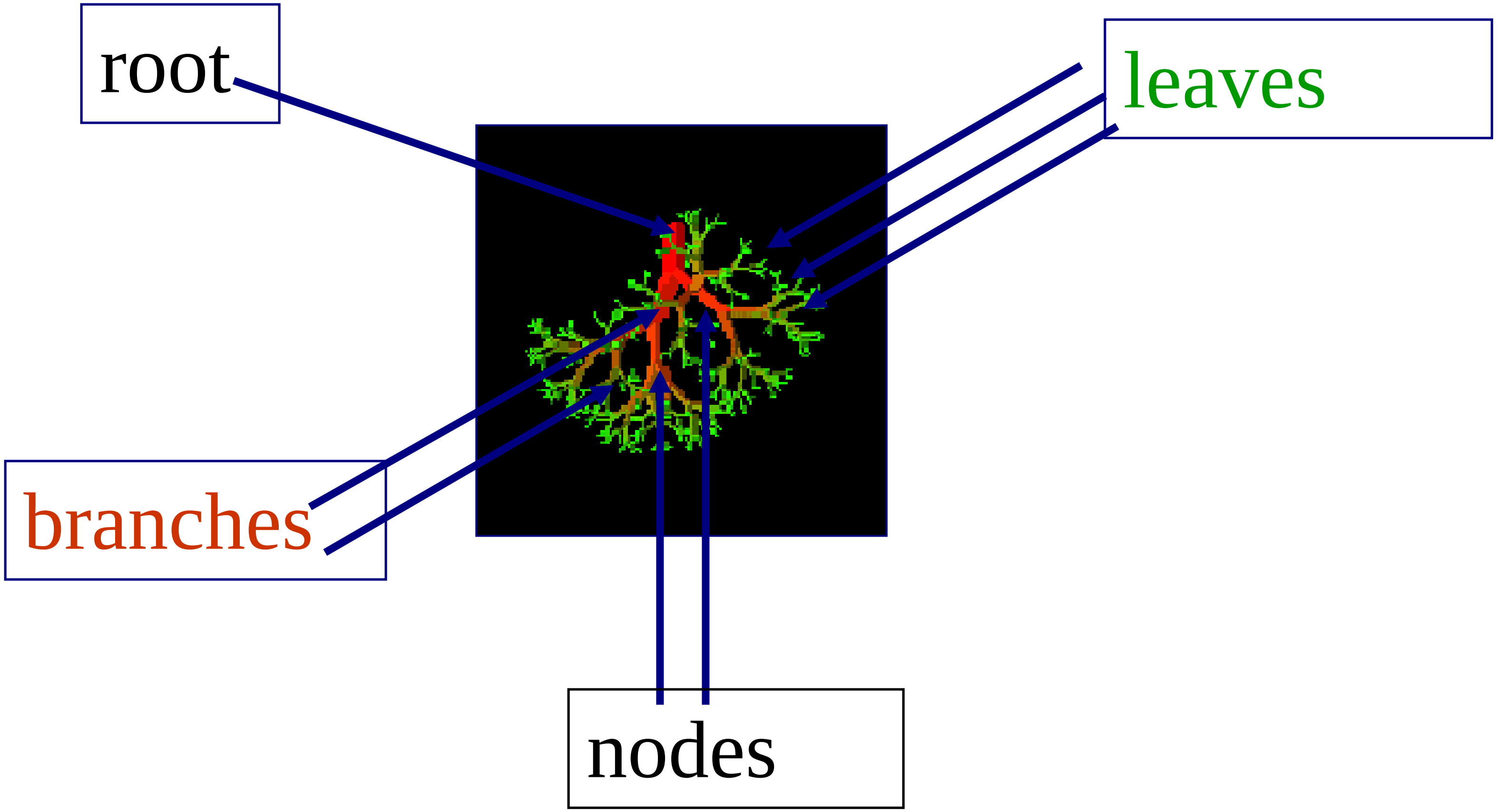
All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C /2nd Edition",
Silicon Press, 2008.

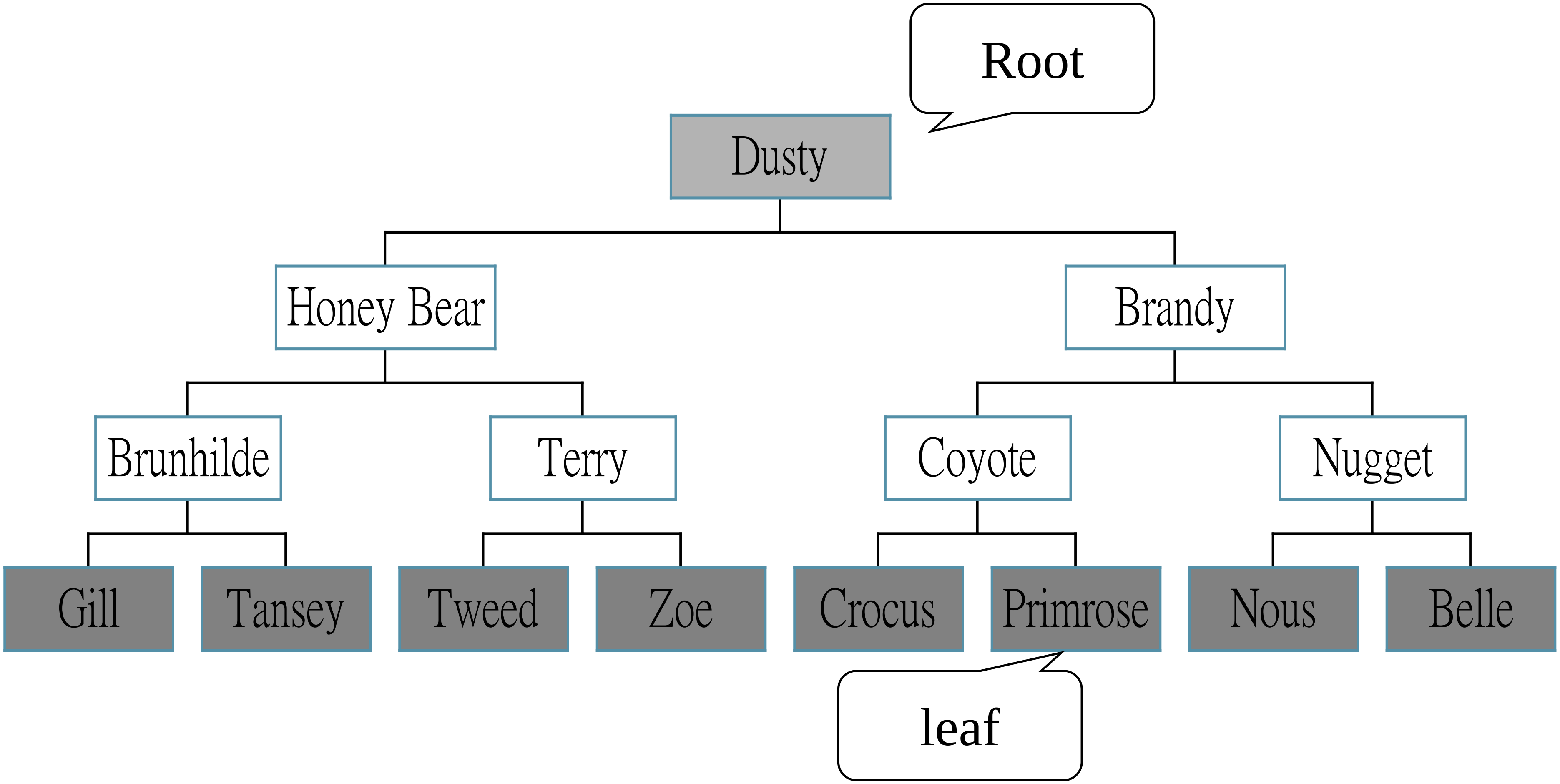
Nature Lover's View Of A Tree



Computer Scientist's View



Trees

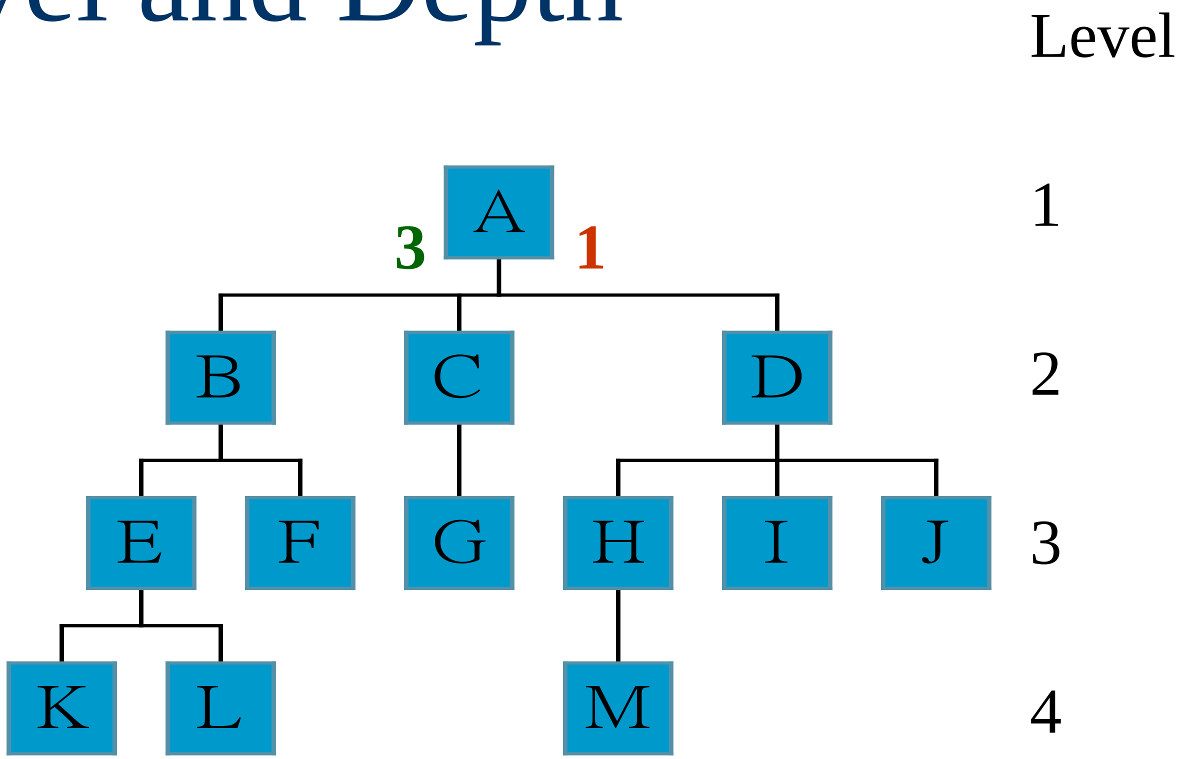

Root
Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the .
- The remaining nodes are partitioned into n>=0 disjoint sets T_1, ..., T_n, where each of these sets is a tree.
- We call T_1, ..., T_n the of the root.

Level and Depth

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



Terminology(1/2)

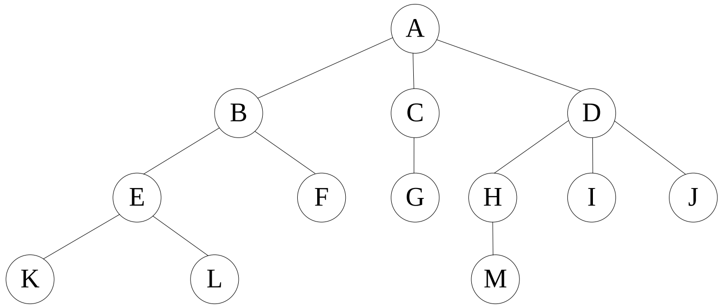
- The   of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is   or terminal node.
- A node that has subtrees is the   of the roots of the subtrees.

Terminology(2/2)

- The roots of these subtrees are the 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

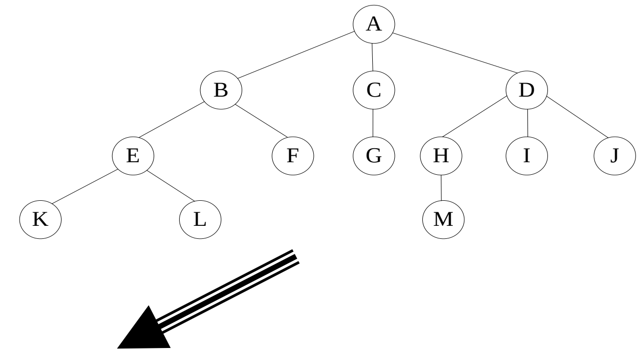


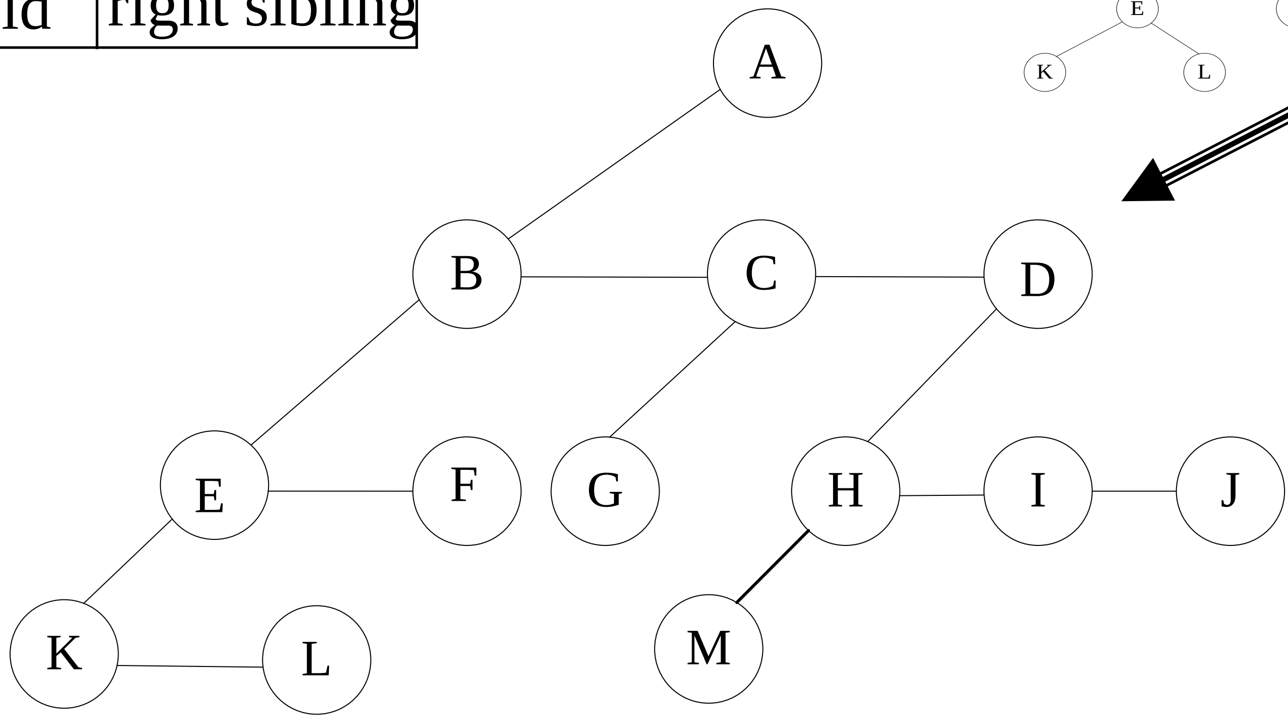
  – (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  – The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link n

Left Child - Right Sibling

data	
left child	right sibling





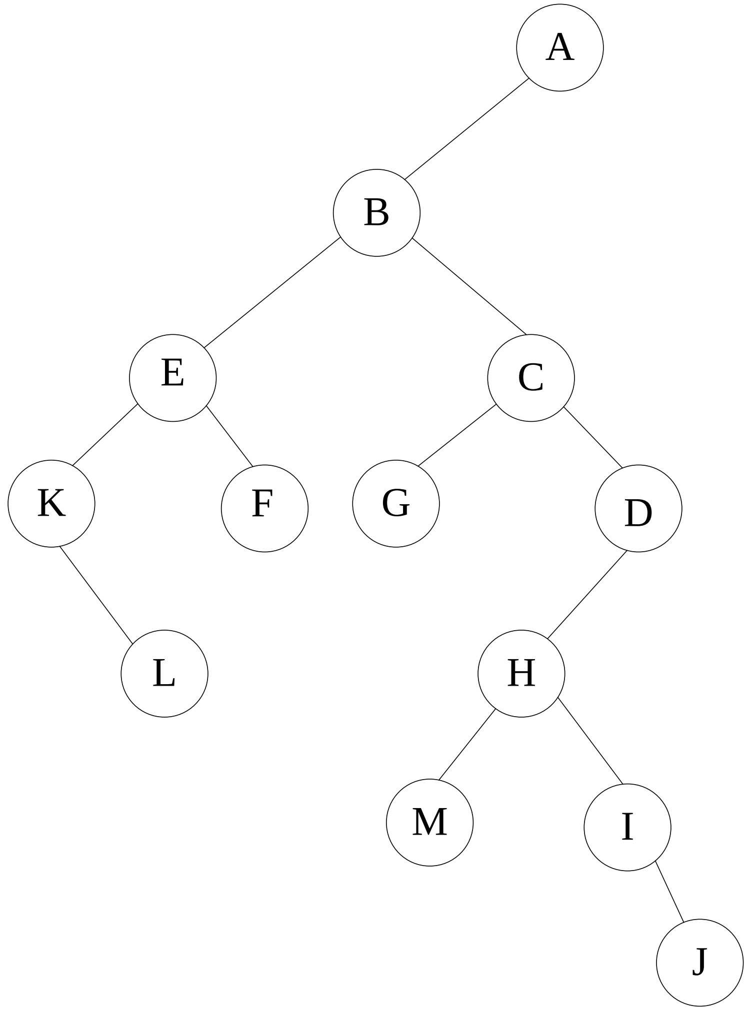
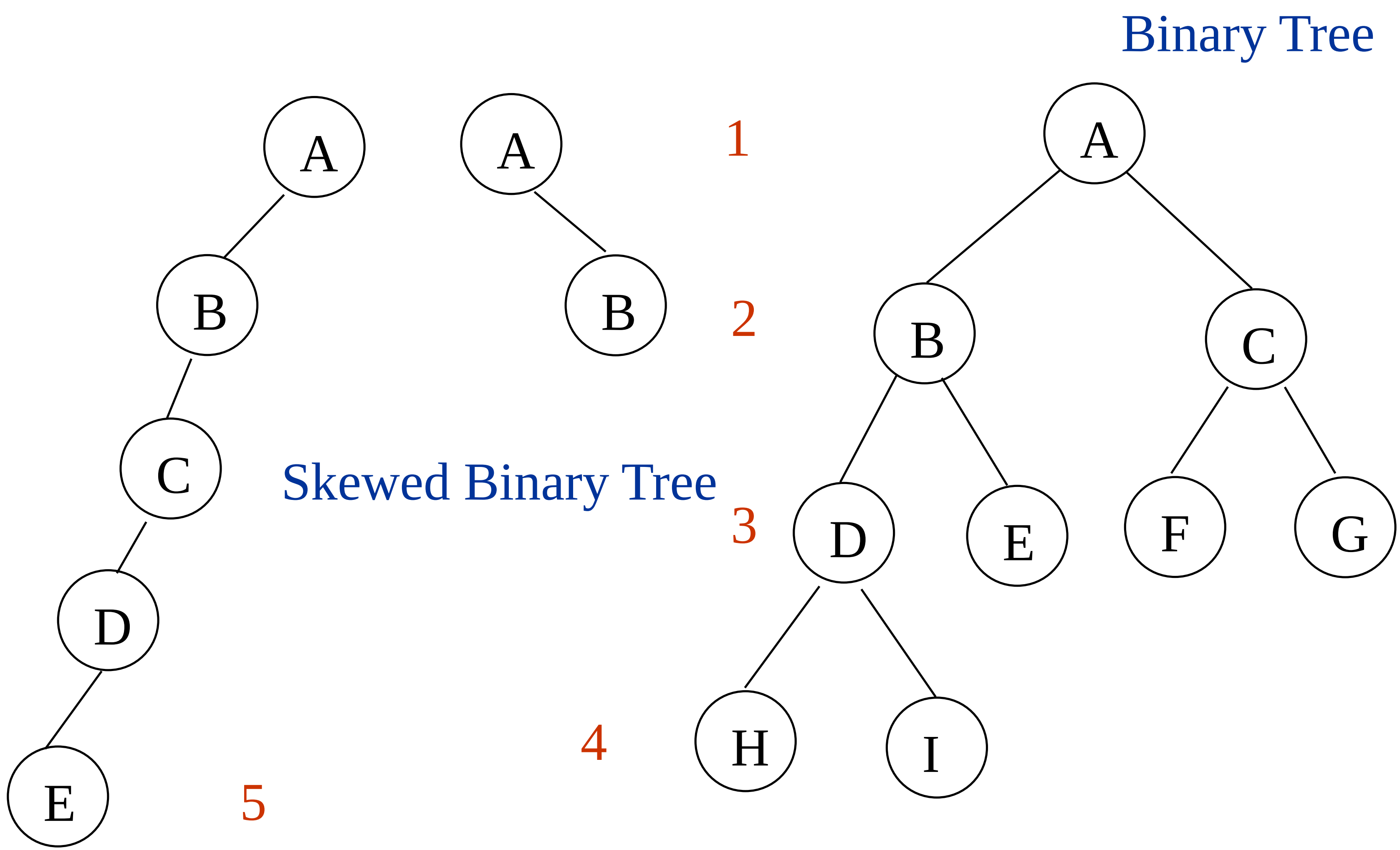


***Figure 5.7:** Left child-right child tree representation of a tree (p.197)

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
and .
- Any tree can be transformed into binary tree.
  - by representation
- The left subtree and the right subtree are distinguished.

Samples of Trees



Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is , i>=1.
- The maximum nubmer of nodes in a binary tree of depth k is , 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

proof:

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

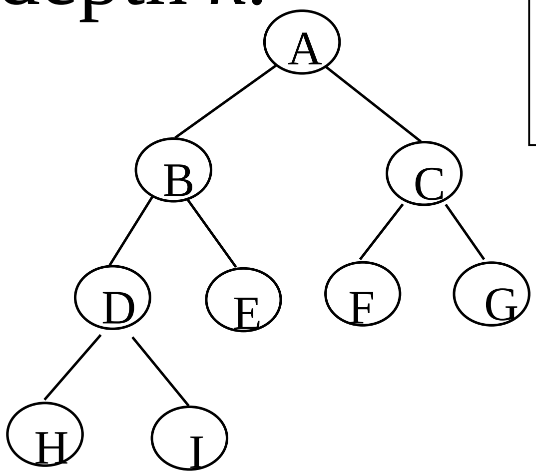
Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

$n = n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$,
$n_1+2n_2+1= n_0+n_1+n_2$ ==>

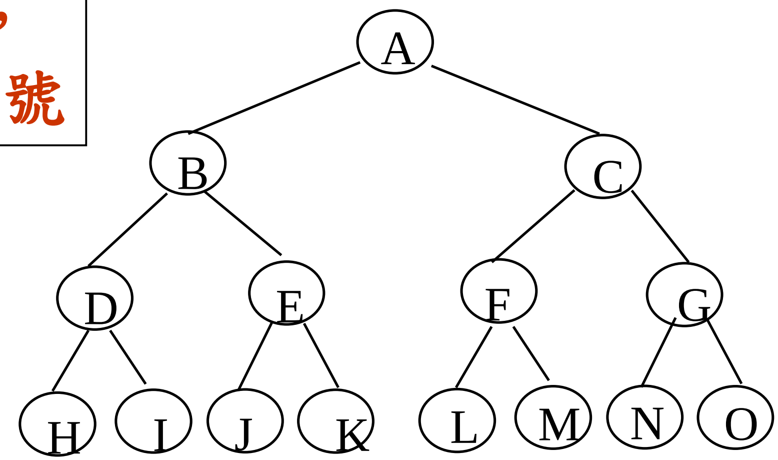
BT VS BT

- 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.

由上至下，
由左至右編號



Complete binary tree

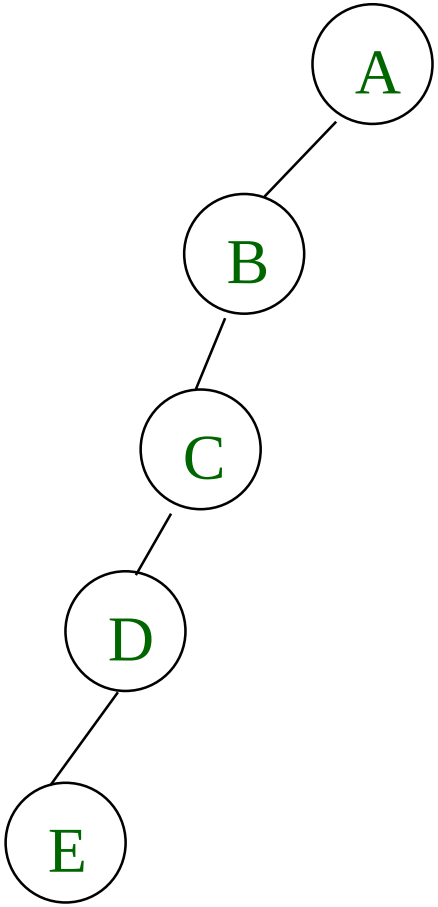


Full binary tree of depth 4

Binary Tree Representations

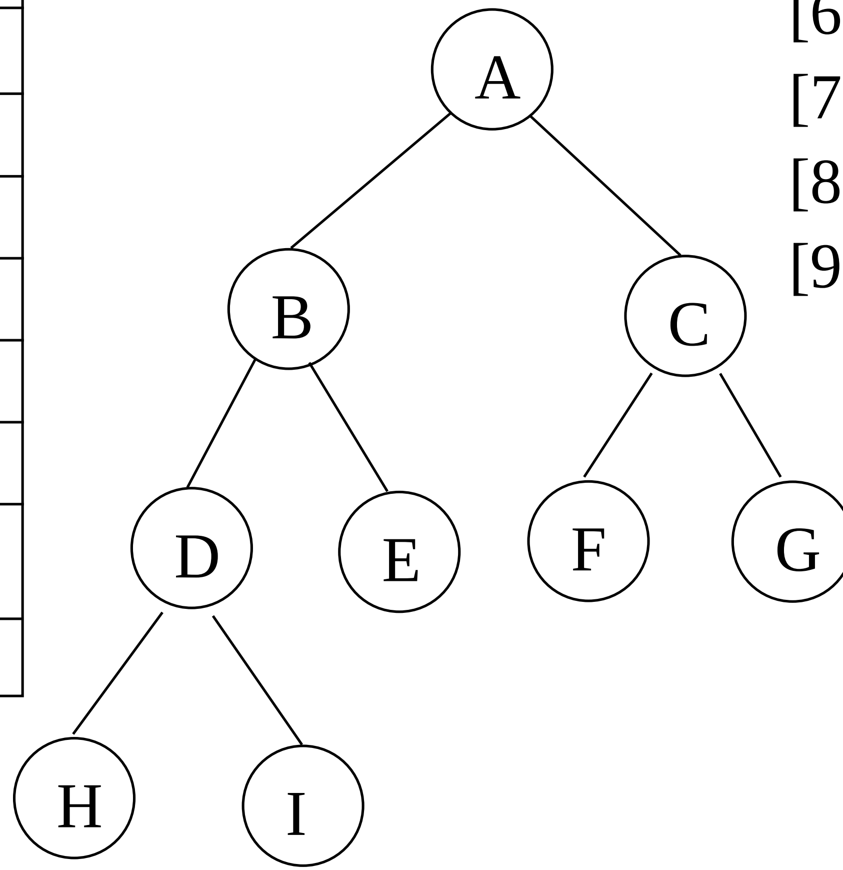
- If a complete binary tree with n nodes (depth = log n + 1) is represented sequentially, then for any node with index i, $1 <= i <= n$, we have:
  - *parent*(i) is at $\lfloor i/2 \rfloor$ if i!=1. If i=1, i is at the root and has no parent.
  - *left_child*(i) ia at $2i$ if $2i <= n$. If $2i$>n, then 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.

Sequential Representation



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

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

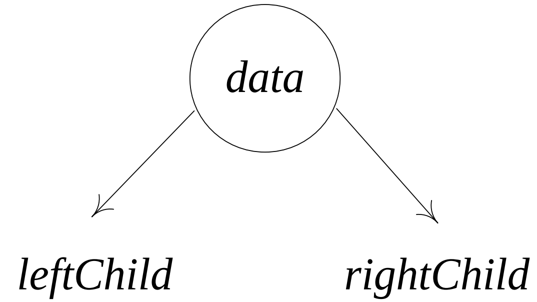


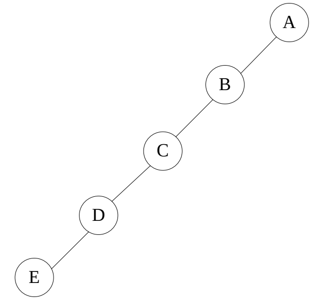
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

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

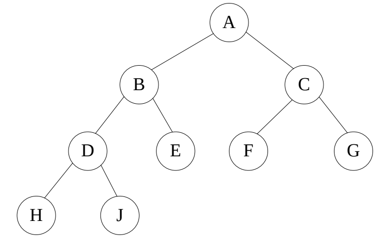
leftChild	*data*	*rightChild*





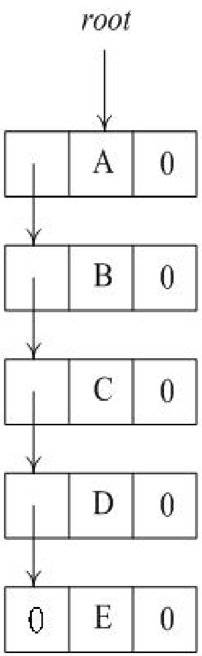
A
B
C
D
E

(a)



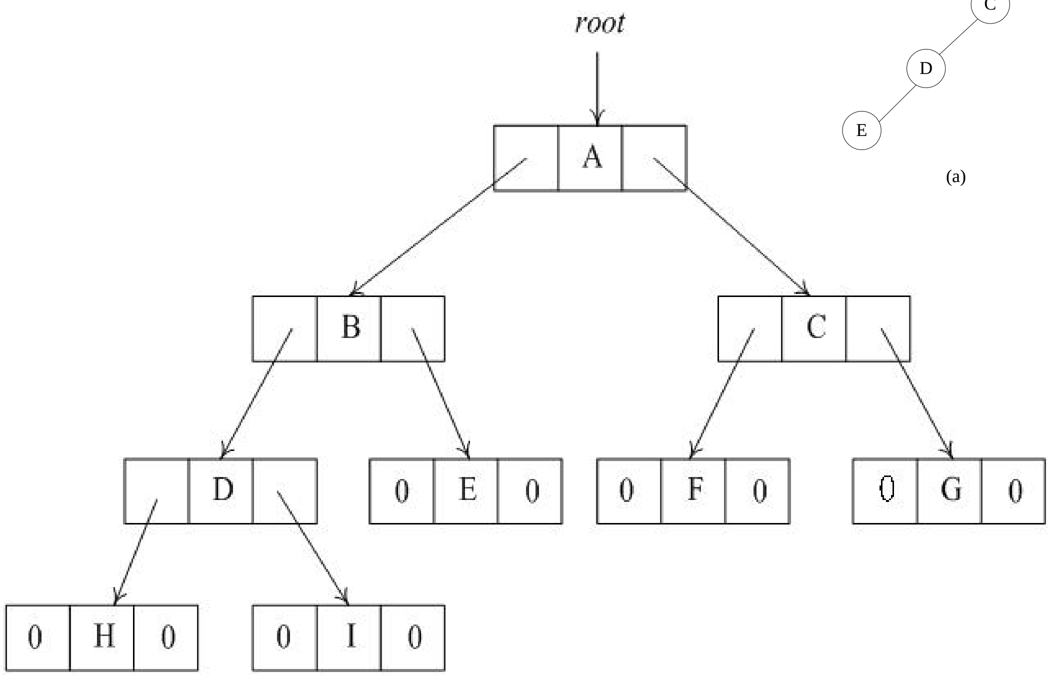
A
B
C
D
E
F
G
H
J

(b)



root
A 0
B 0
C 0
D 0
0 E 0

(a)



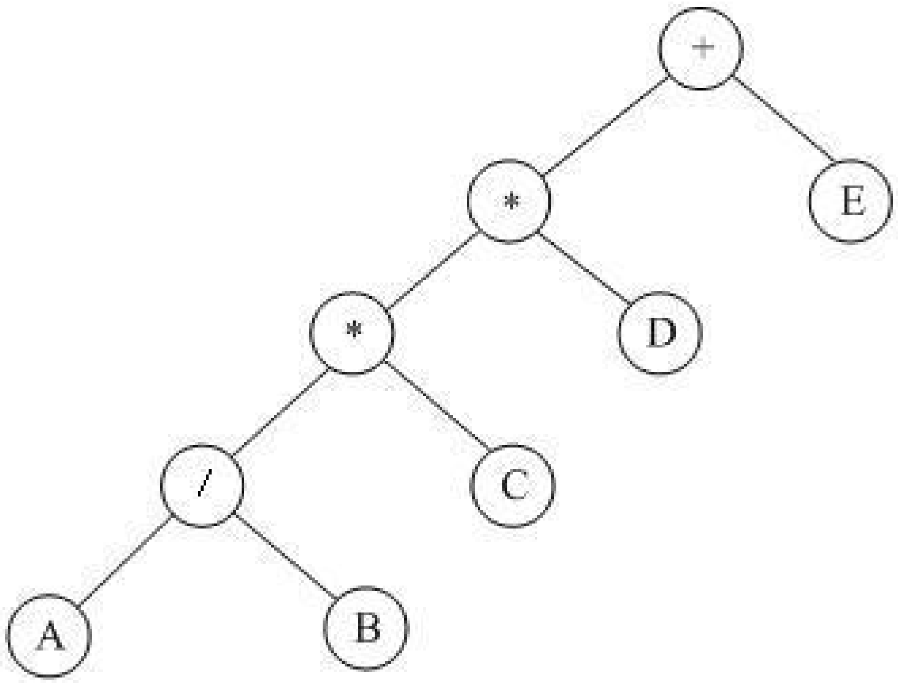
root
A
B
C
D
0 E 0
0 F 0
0 G 0
0 H 0
0 I 0

(b)

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

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);
    }
}
```

A / B * C * D + E

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);
    }
}
```

+ * * / A B C D E

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);
    }
}
```

A B / C * D * E +

Iterative Inorder Traversal

(using stack)

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

O(n)

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	

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*)

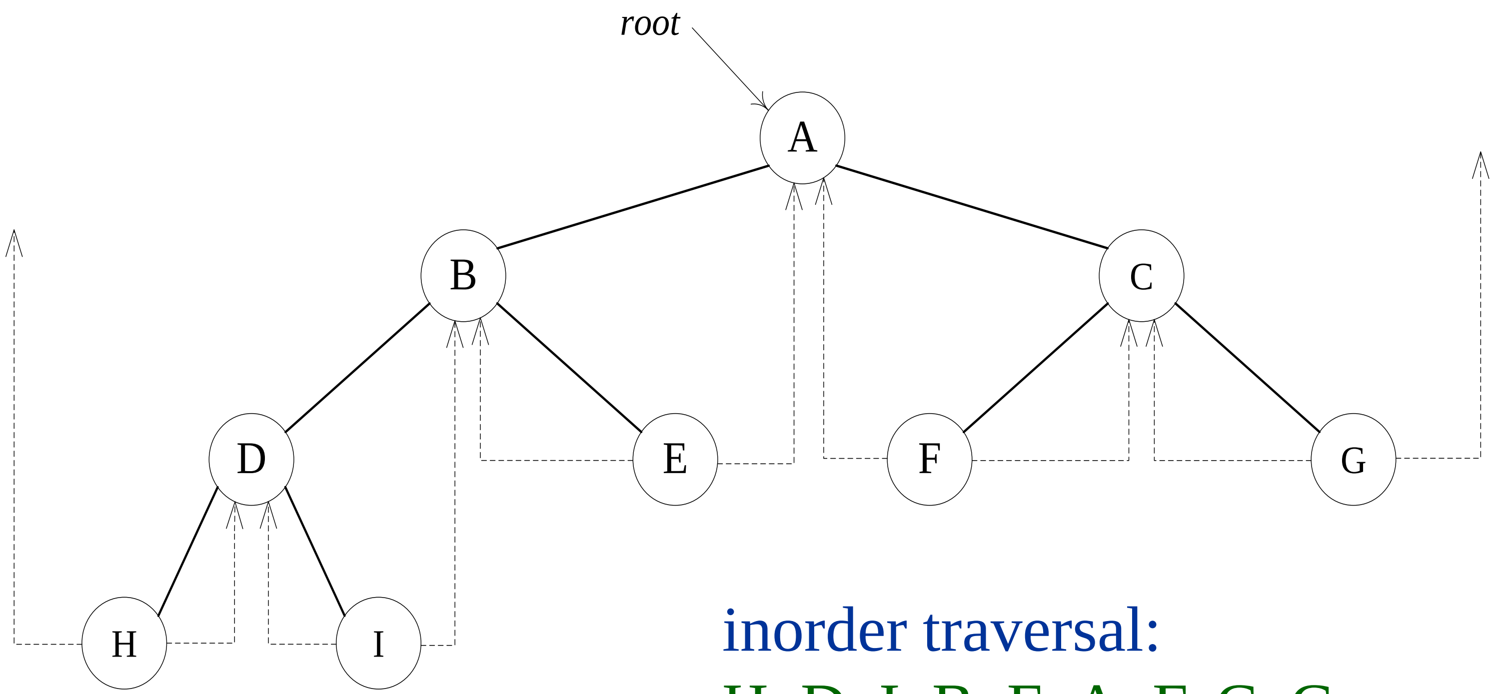
If `ptr->left_child` is null,

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*

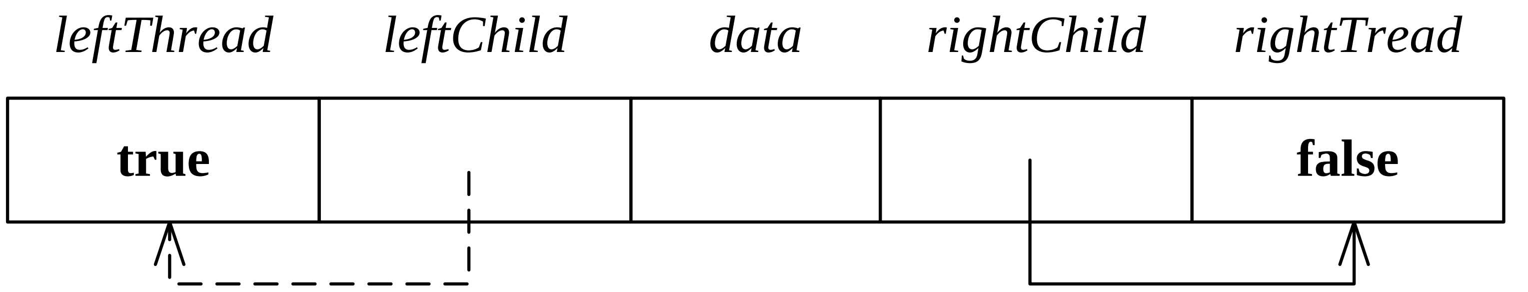
A Threaded Binary Tree



inorder traversal:

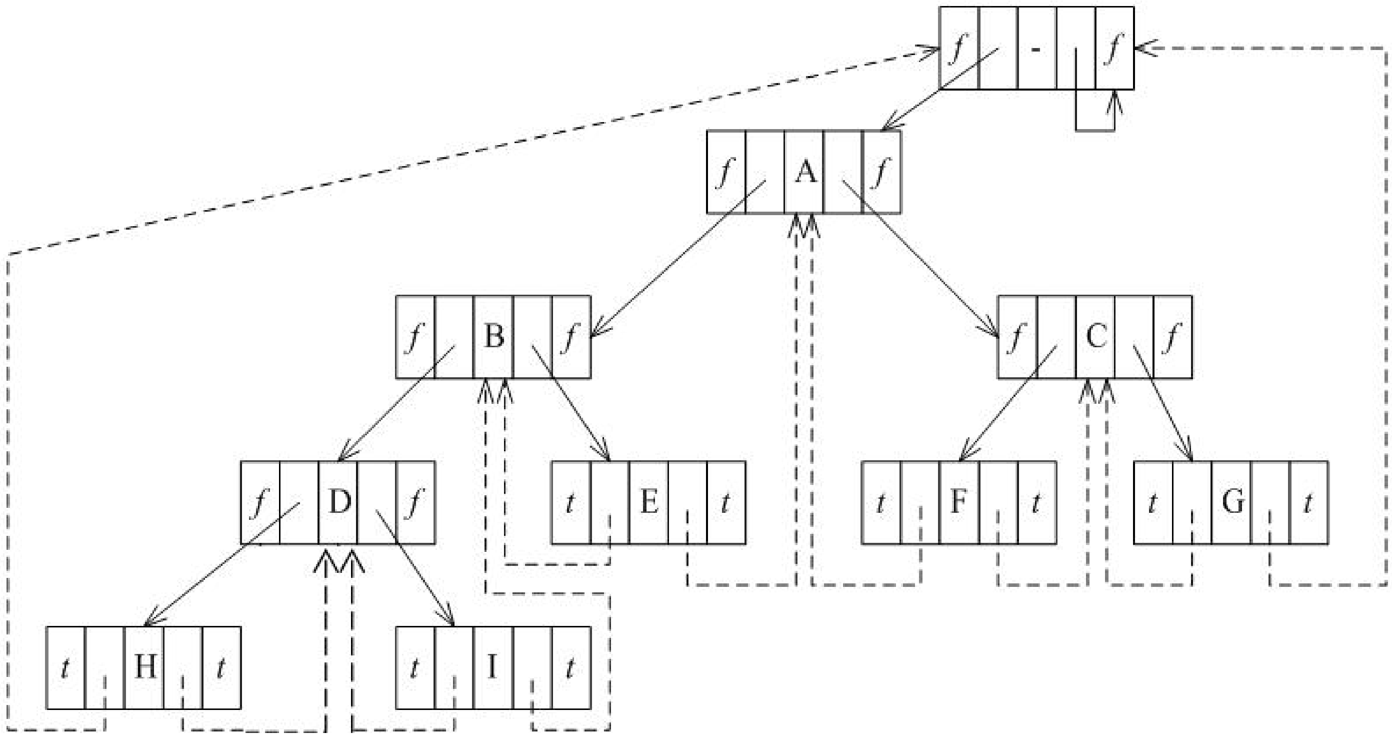
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



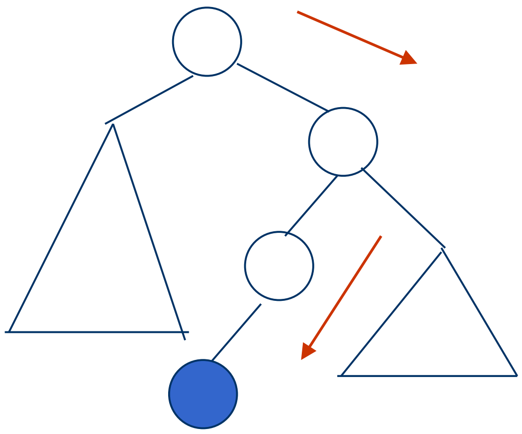

```
typedef struct threaded_tree
  *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;  };
```

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);
    }
}
```

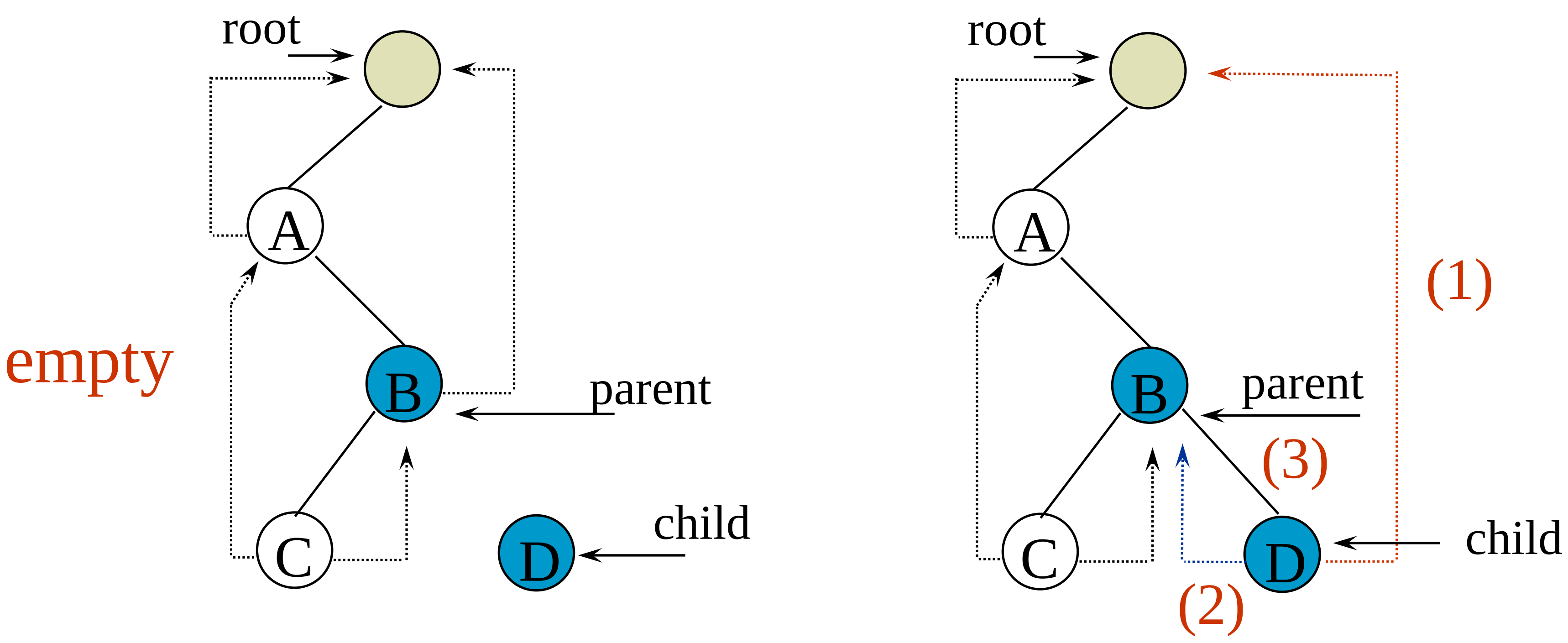
O(n)

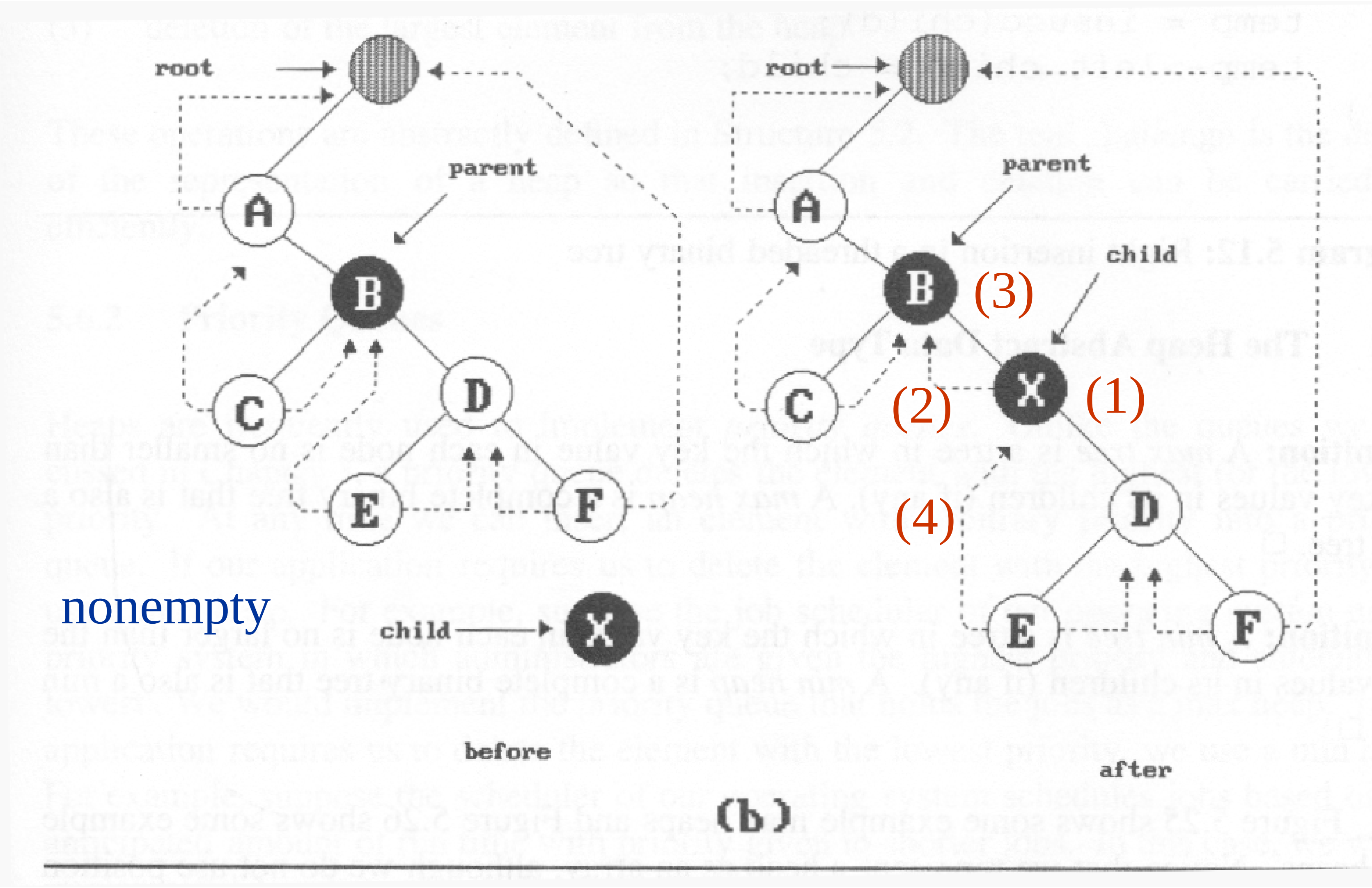
Inserting Nodes into Threaded BTs

- 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->left_child` to point to `parent`
  - set `child->right_child` to `parent->right_child`
  - change `parent->right_child` to point to `child`

Examples

Insert a node D as a right child of B.





root
parent
child
(3)
(1)
(2)
(4)
nonempty
before
after
(b)

Right Insertion in Threaded BTs

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

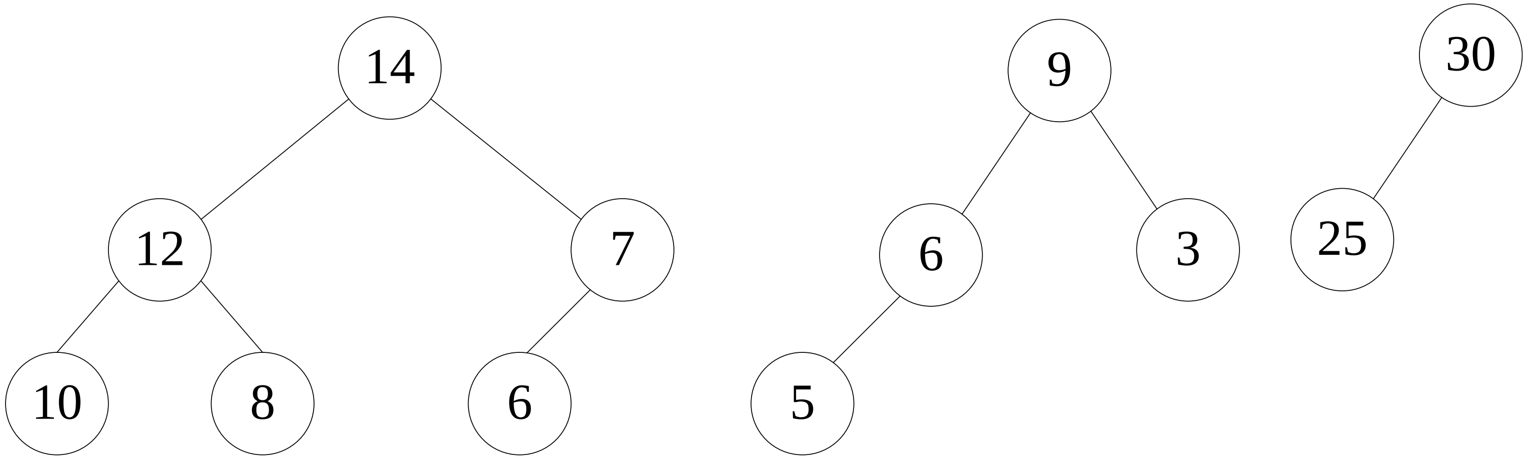
Heap(1/4)

- A is a tree in which the key value in each node is no smaller than the key values in its children. A is a complete binary tree that is also a max tree.
- A is a tree in which the key value in each node is no larger than the key values in its children. A is a complete binary tree that is also a min tree.

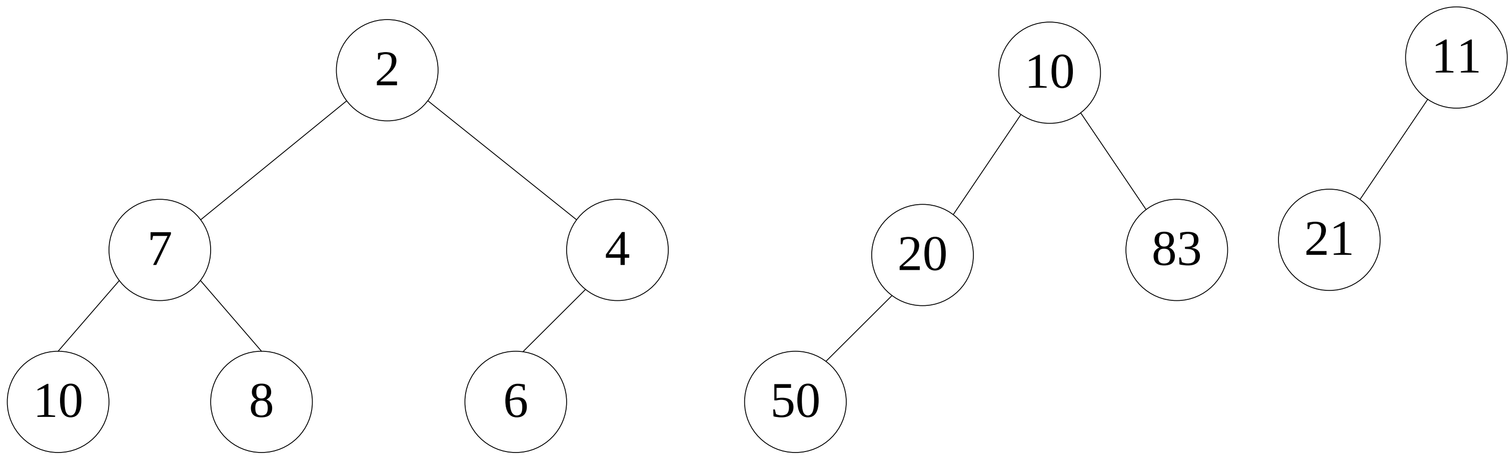
Heap(2/4)

- Property:
  - The root of max heap (min heap) contains the largest (smallest) element.
- Operations on heaps
  - creation of an empty heap
  - insertion of a new element into the heap
  - deletion of the largest (smallest) element from the max (min) heap

Heap(3/4)-max heap



Heap(4/4)-min heap



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

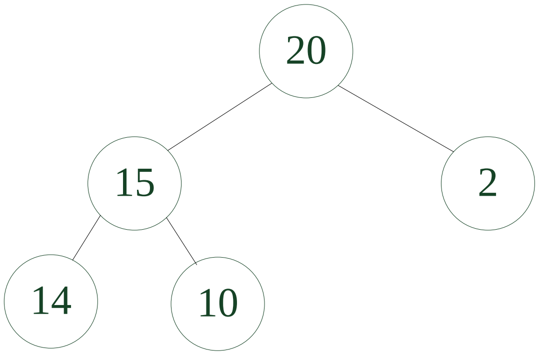
- machine service
  - amount of time (min heap)
  - amount of payment (max heap)
- factory
  - time tag

Data Structures

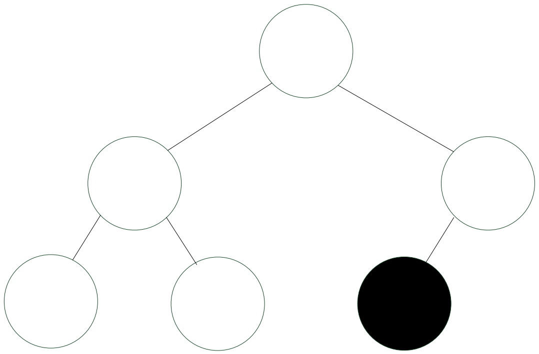
- unordered linked list
- unordered array
- sorted linked list
- sorted array
- heap

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

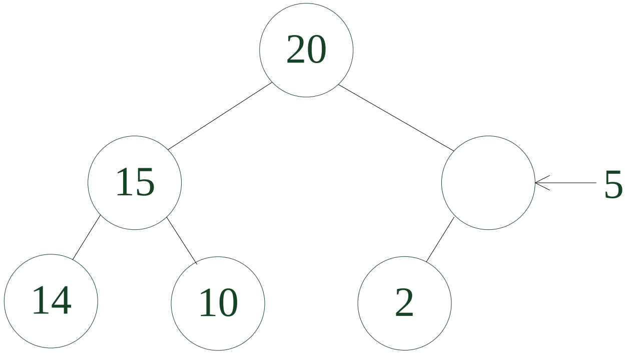
Example of Insertion to Max Heap



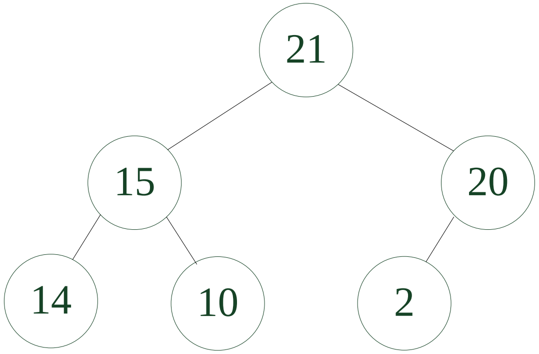
(a)



(b)



(c)



(d)

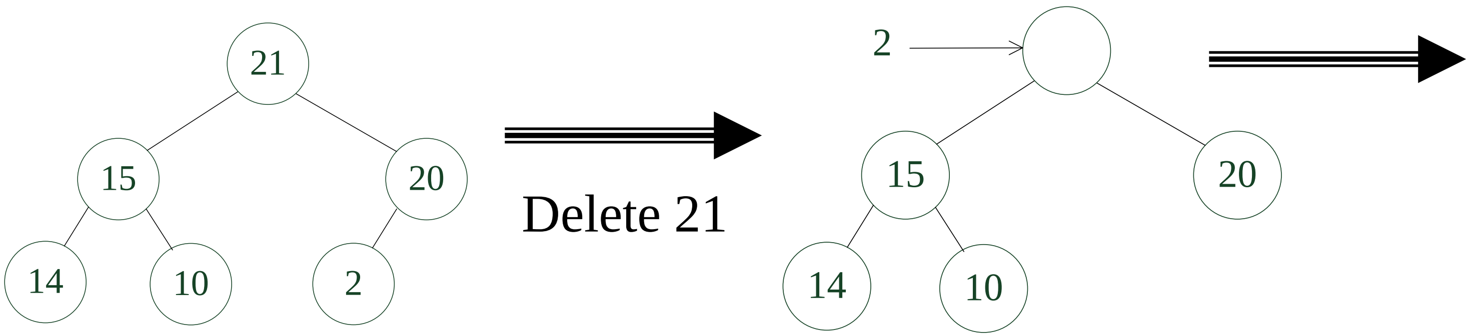
Insertion into a Max Heap

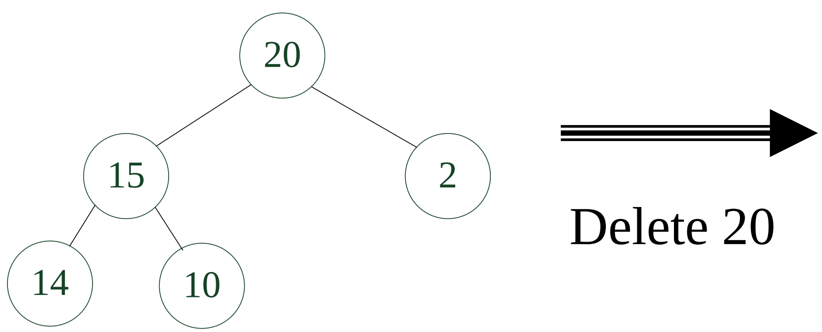
```
void insert_max_heap(element item, int *n)
{
  int i;
  if (HEAP_FULL(*n)) {
    fprintf(stderr, "the heap is full.\n");
    exit(1);
  }
  i = ++(*n);
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

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

$O(\log_2 n)$

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;
```

```
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;
}
```

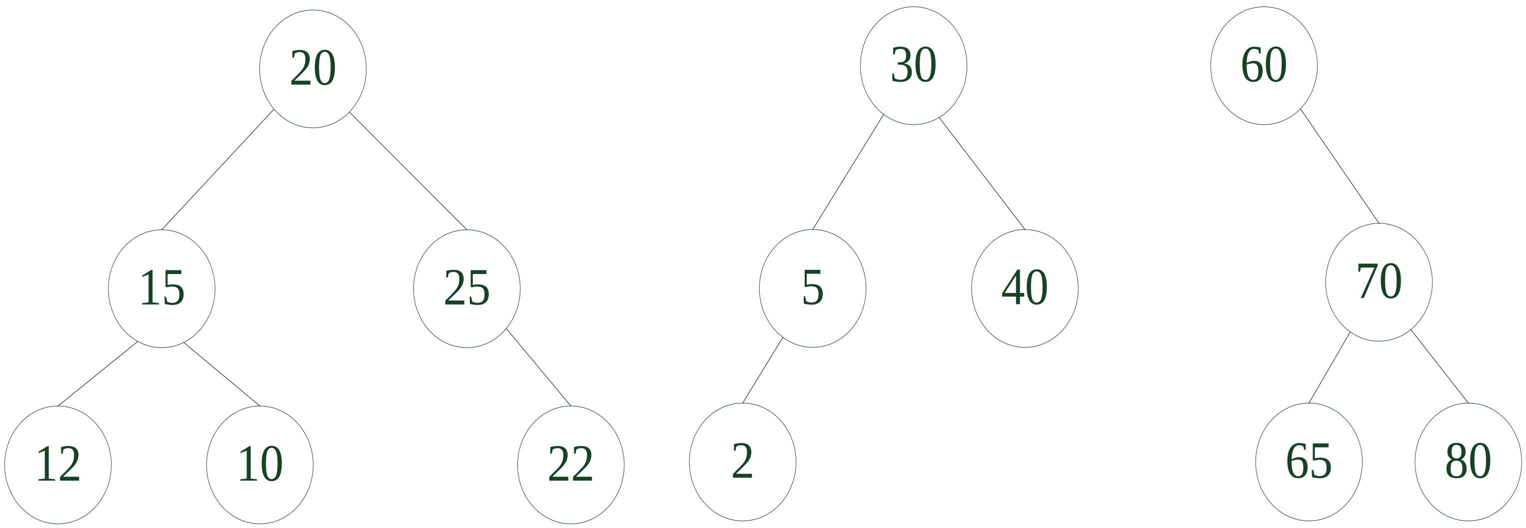
Binary Search Tree(1/2)

- Heap
  - a min (max) element is deleted.
  - deletion of an arbitrary element
  - search for an arbitrary element

Binary Search Tree(2/2)

- Binary search tree
  - Each node has exactly one key and the keys in the three are distinct
  - The keys in a the left subtree ( ) are smaller ( ) than the key in the root
  - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



(a) (b) (c)

Searching a Binary Search Tree

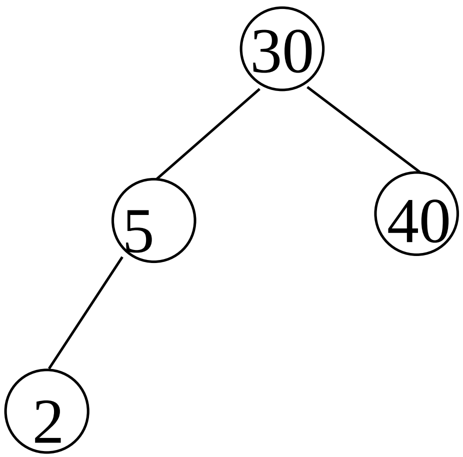
```
tree_pointer search(tree_pointer root,
                    int key)
{
/* return a pointer to the node that
  contains key. If there is no such
  node, return NULL */

   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);
}
```

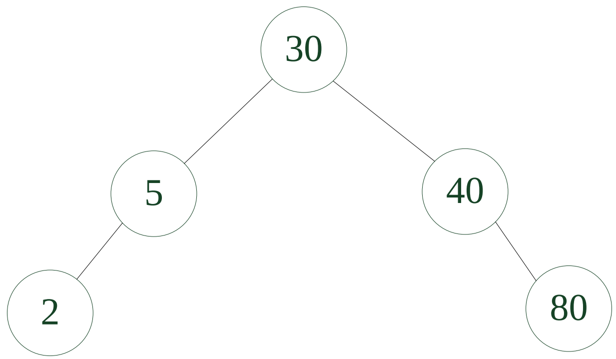
Another Searching Algorithm

```
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;
}
```

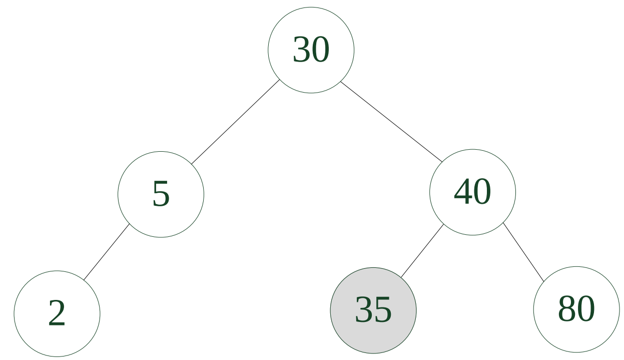
O(h)

Insert Node in Binary Search Tree





(a)



(b)

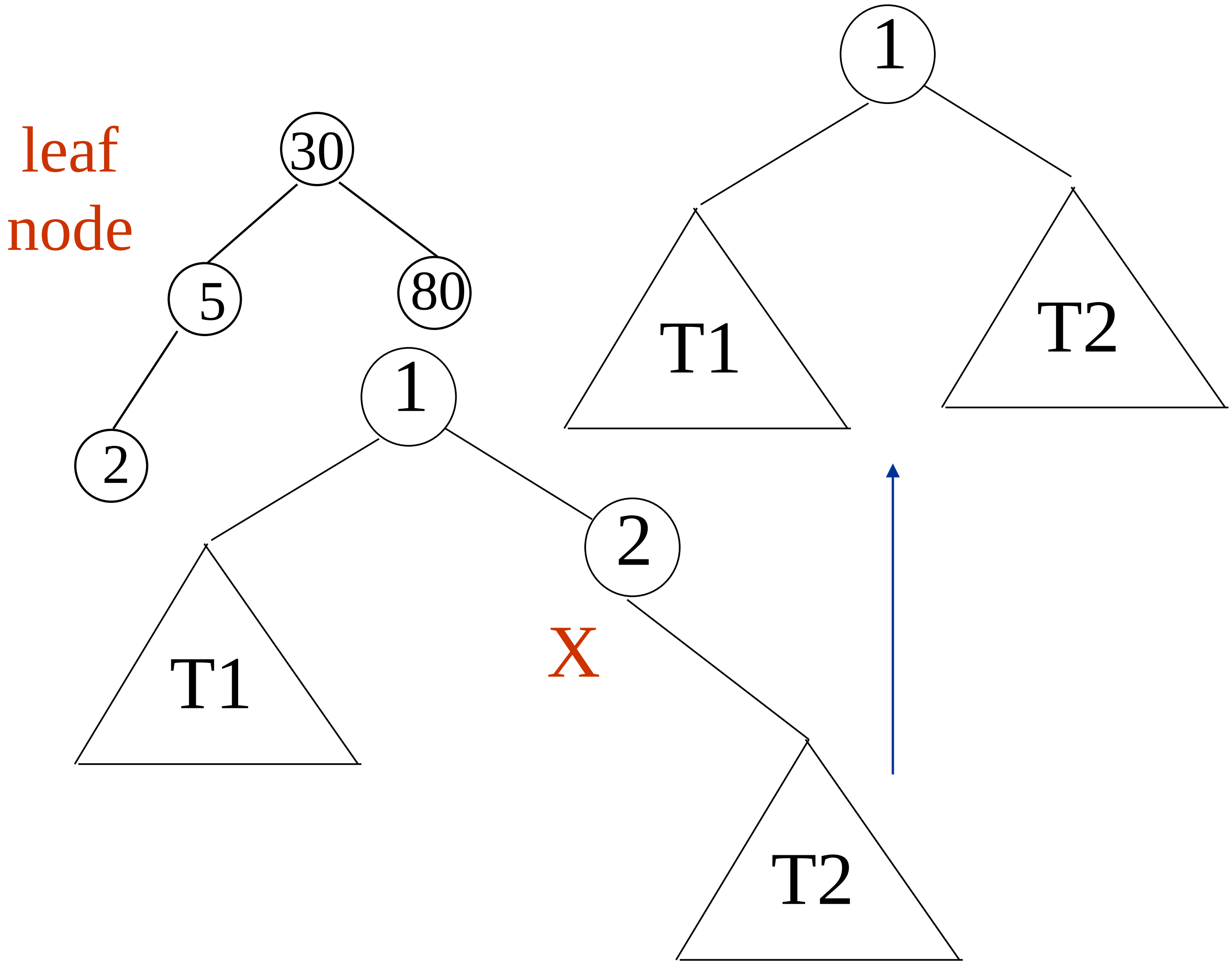
Insert 80

Insert 35

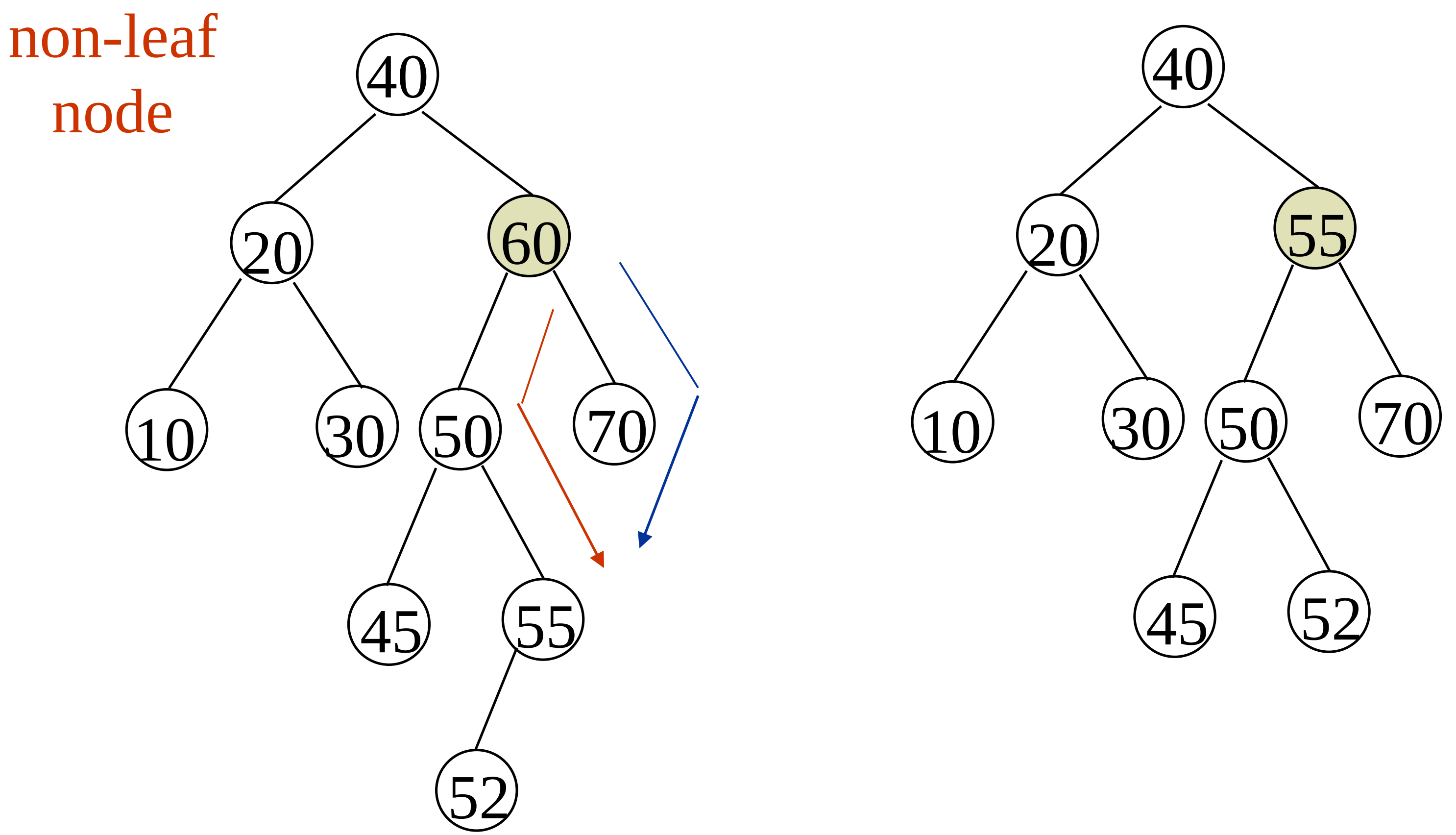
Insertion into A Binary Search Tree

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

Deletion for A Binary Search Tree



Deletion for A Binary Search Tree



Before deleting 60 After deleting 60

Selection Trees

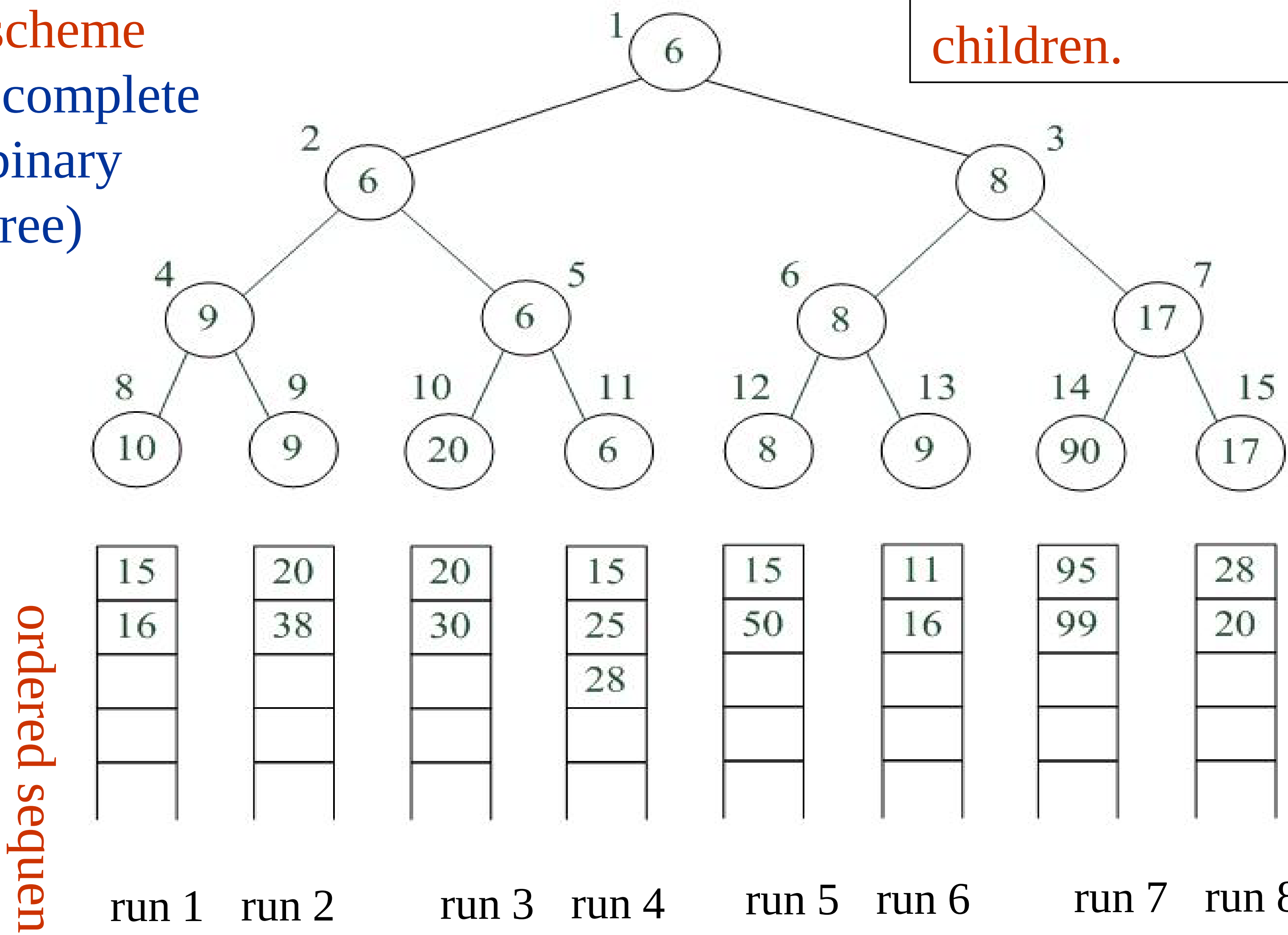
(1) tree

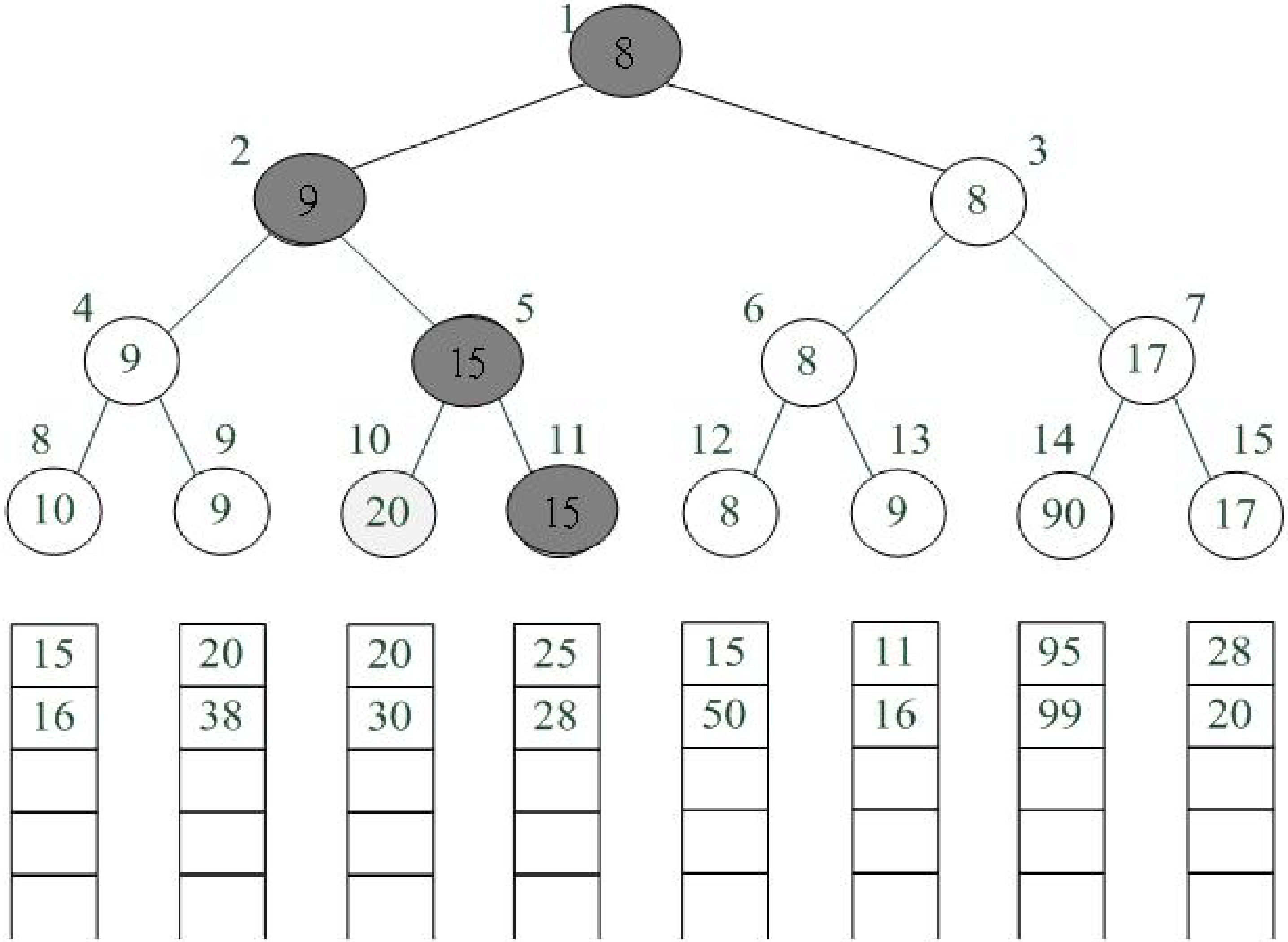
(2) tree

winner tree

sequential allocation scheme (complete binary tree)

Each node represents the smaller of its two children.



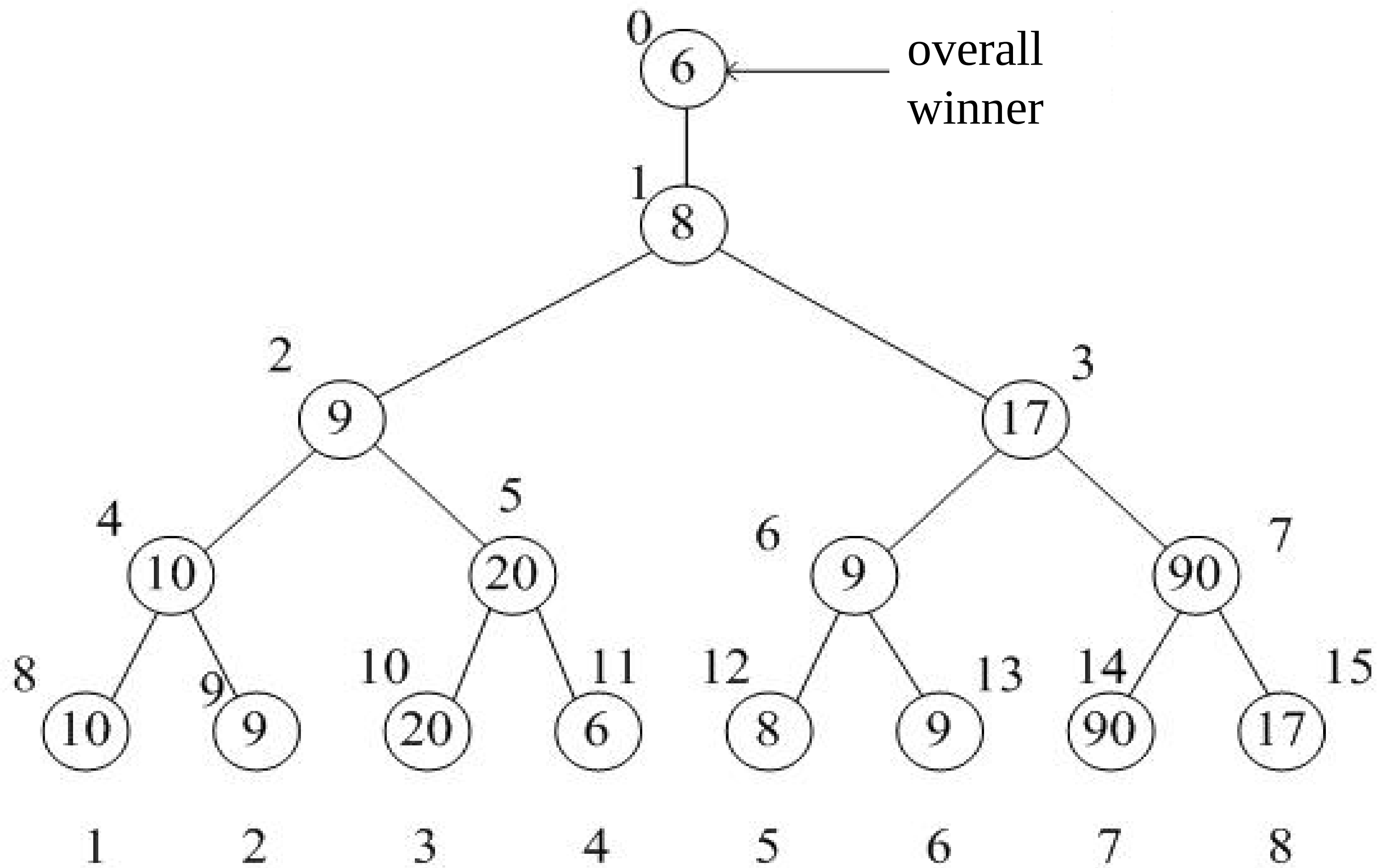


1
8
2
9
3
8
4
9
5
15
6
8
7
17
8
10
9
9
10
20
11
15
12
8
13
9
14
90
15
17
15
16
20
38
20
30
25
28
15
50
11
16
95
99
28
20

Analysis

- K: # of runs
- n: # of records
- setup time: O(K) (K-1)
- restructure time: $O(\log_2 K)$ $\lceil \log_2(K+1) \rceil$
- merge time: $O(n\log_2 K)$
- slight modification: loser tree
  - consider the parent node only (vs. sibling nodes)

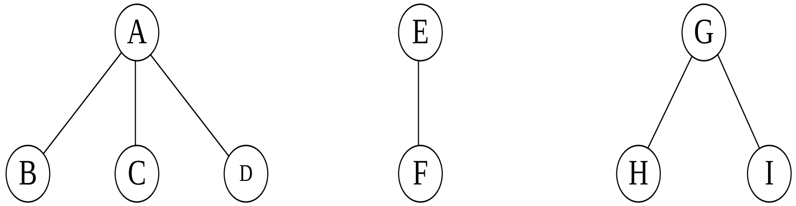
Loser tree

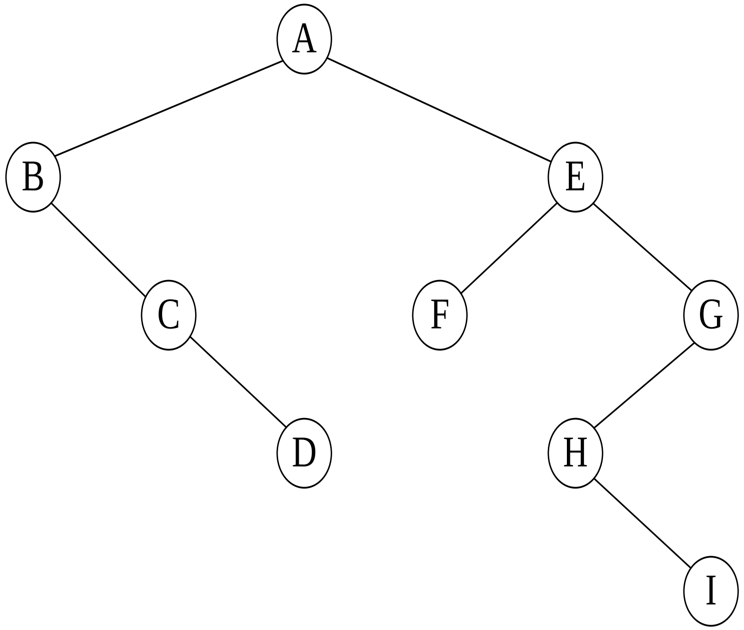


0
6
overall
winner
1
8
2
9
3
17
4
10
5
20
6
9
7
90
8
10
9
9
10
20
11
6
12
8
13
9
14
90
15
17
1
2
3
4
5
6
7
8

Forest

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





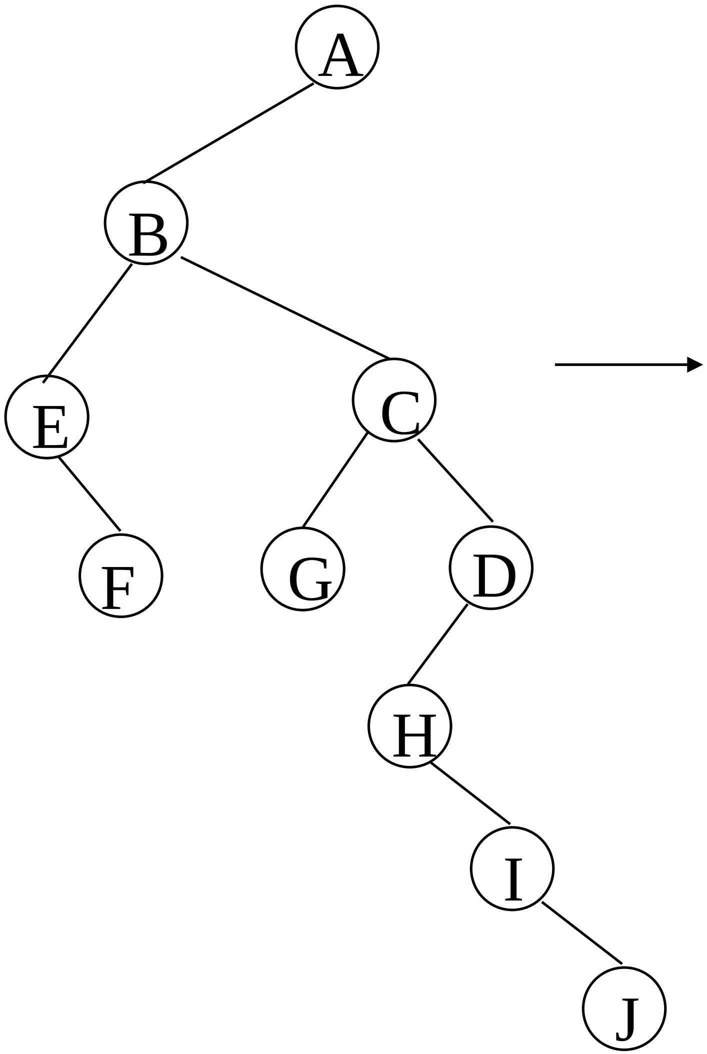
Transform a forest into a binary tree

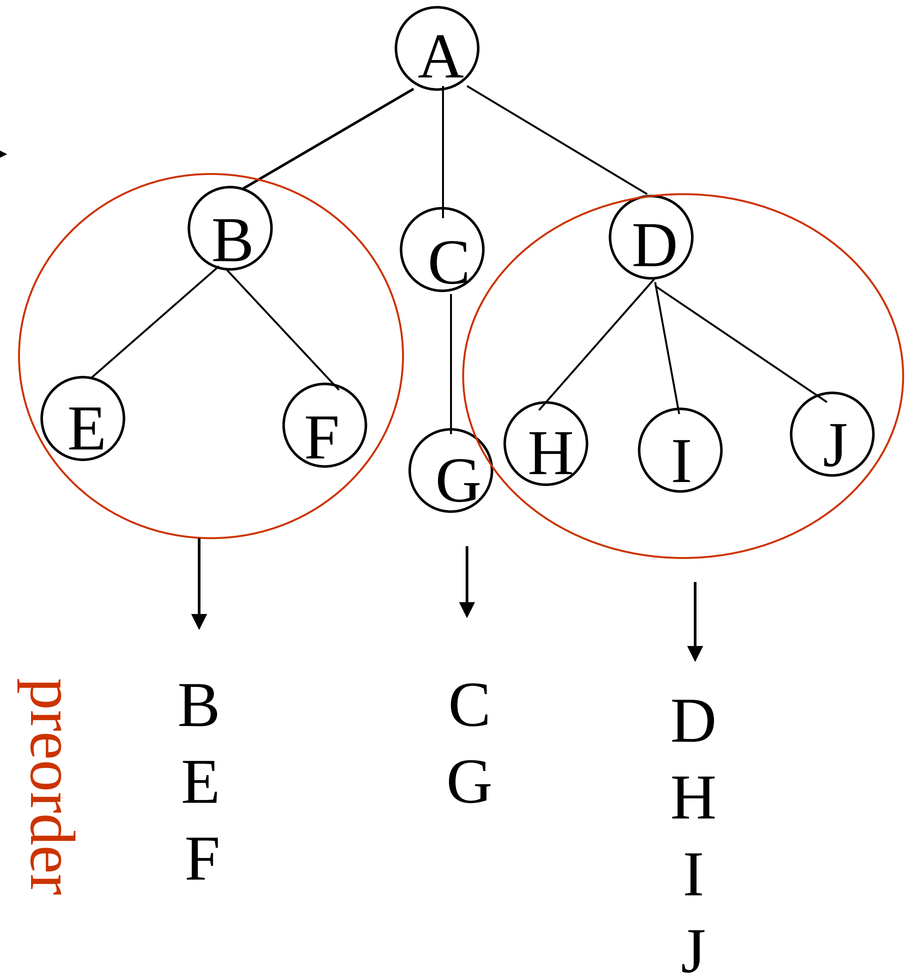
- T1, T2, …, T*n*: a forest of trees
  B(T1, T2, …, Tn): a binary tree corresponding to this forest
- algorithm
  (1) empty, if n = 0
  (2) has root equal to root(T1)
  has left subtree equal to B(T11,T12,…,T1*m*)
  has right subtree equal to B(T2,T3,…,Tn)

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

inorder: EFBGCHIJDA
preorder: ABEFCGDHIJ

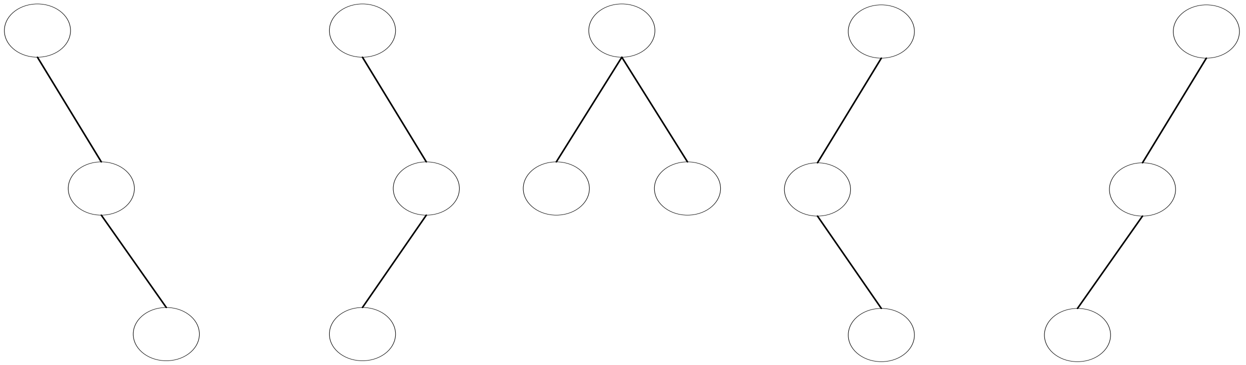




Counting Binary Trees

N=2

N=3



preorder: A B C D E F G H I

inorder: B C A E D G H F I