DATA STRUCTURES

(CSE-203 E)

Unit - 3

By:

Gurpreet Singh

Dean Academics & Astt. Professor (CSE)

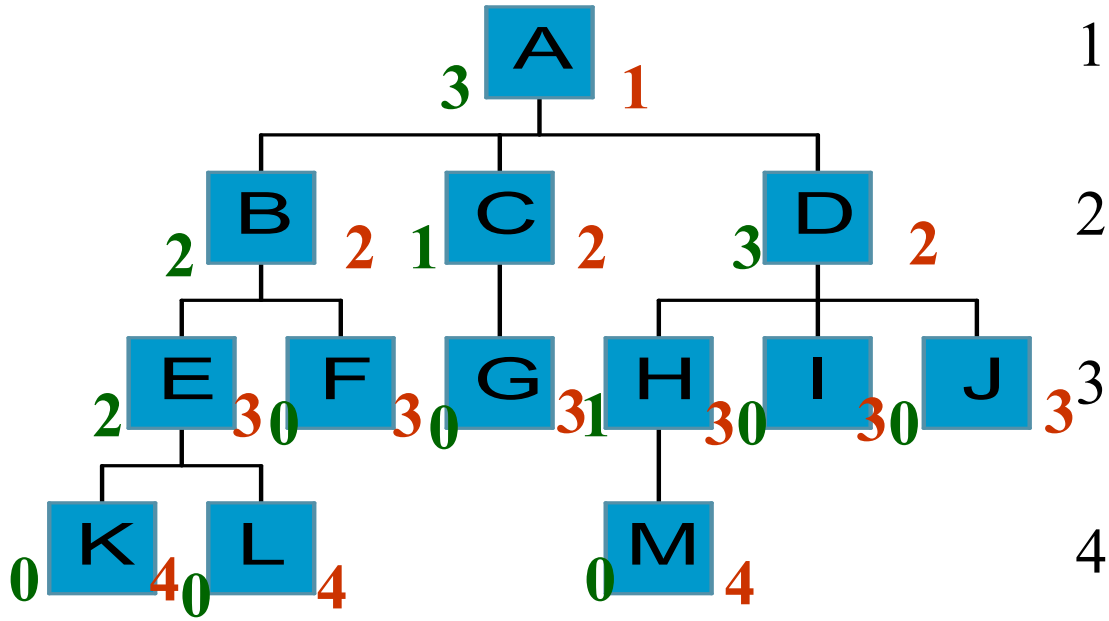
Yamuna Institute of Engineering & Technology, Gadholi

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, ..., T_n, where each of these sets is a tree.
  - We call T_1, ..., 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)

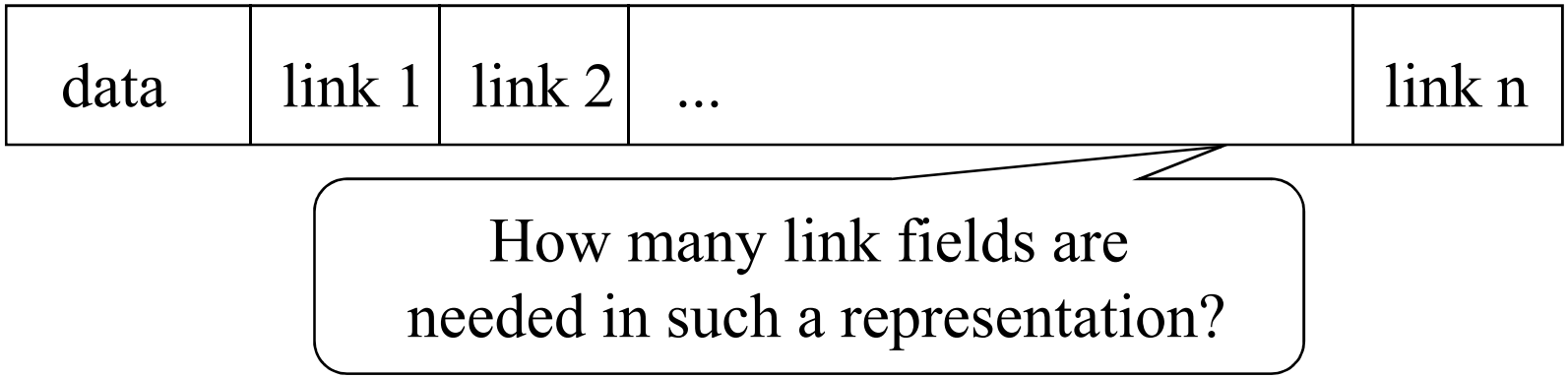


Terminology

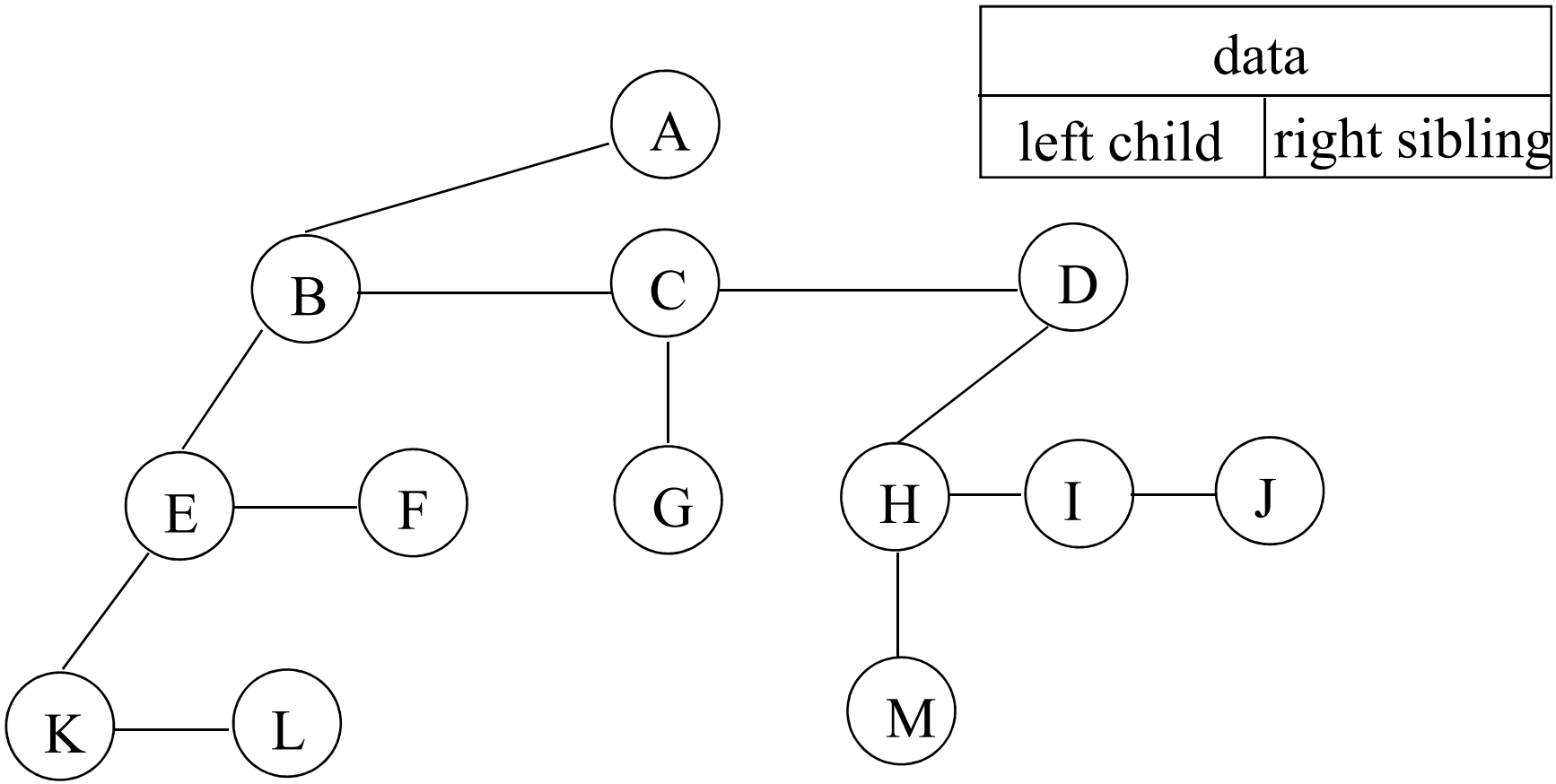
- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children 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



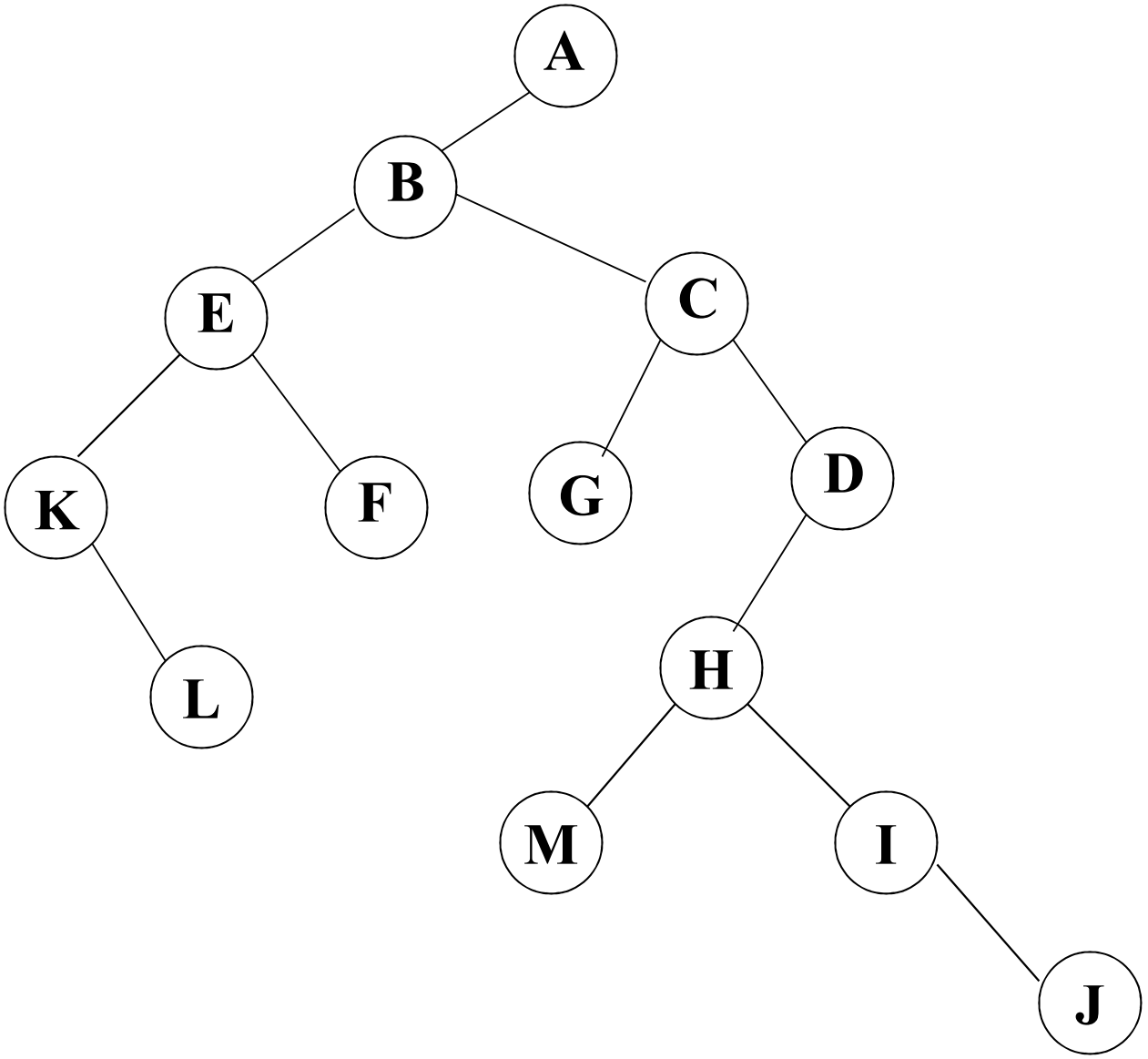
Left Child - Right Sibling



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.

Figure : Left child-right child tree representation of a tree



Abstract Data Type Binary_Tree

structure *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*

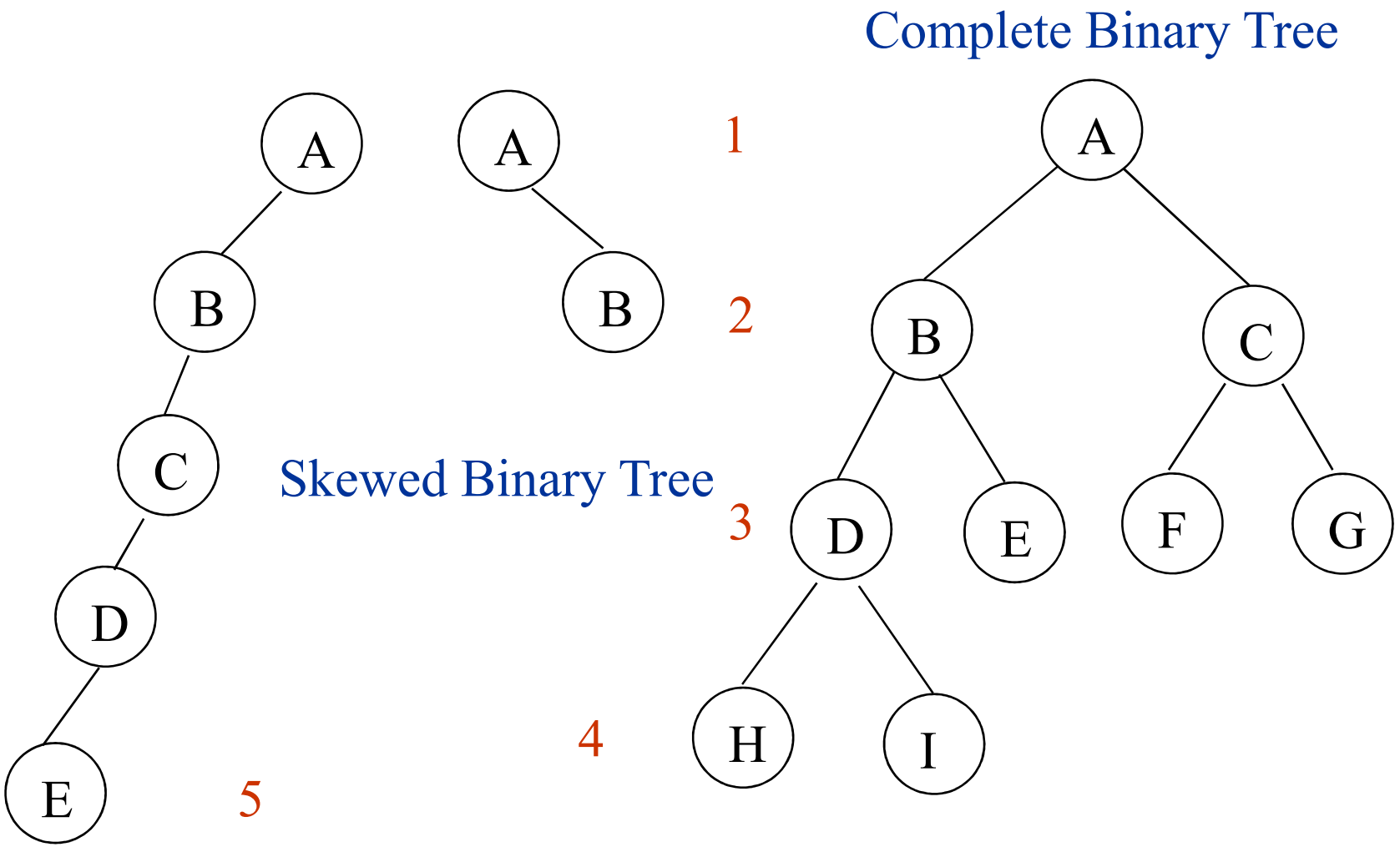
BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree
whose left subtree is *bt1*, whose right subtree is *bt2*,
and whose root node contains the data *item*

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



Example : A binary tree with only left sub trees is called as left skewed binary tree

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 n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0=n2+1

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.

$n = n0+n1+n2$, $B+1=n$, $B=n1+2n2$

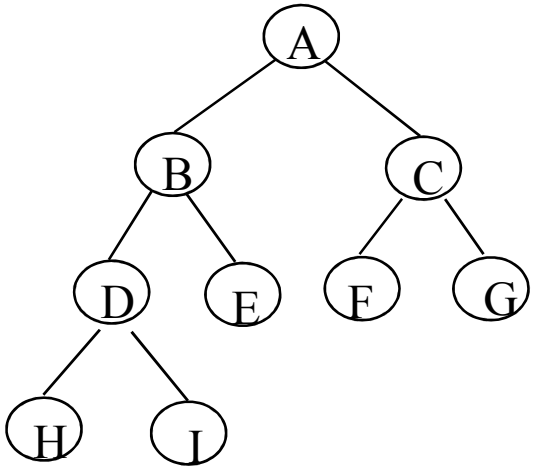
==> $n1+2n2+1= n$,

==> $n1+2n2+1= n0+n1+n2$

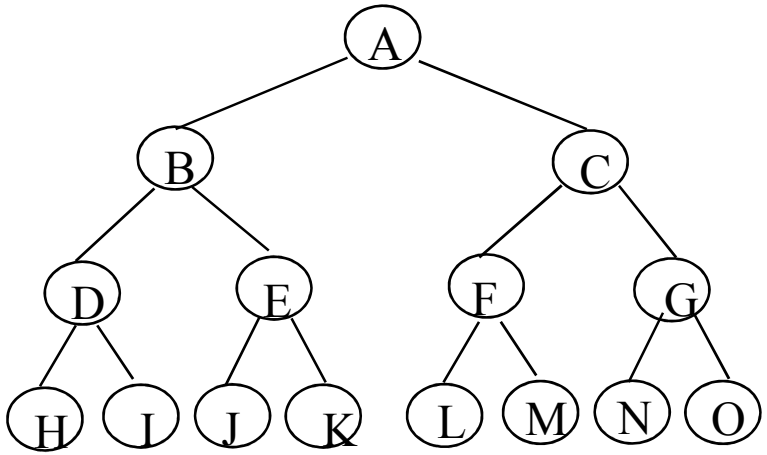
==> $n0=n2+1$

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

Binary Tree Representations

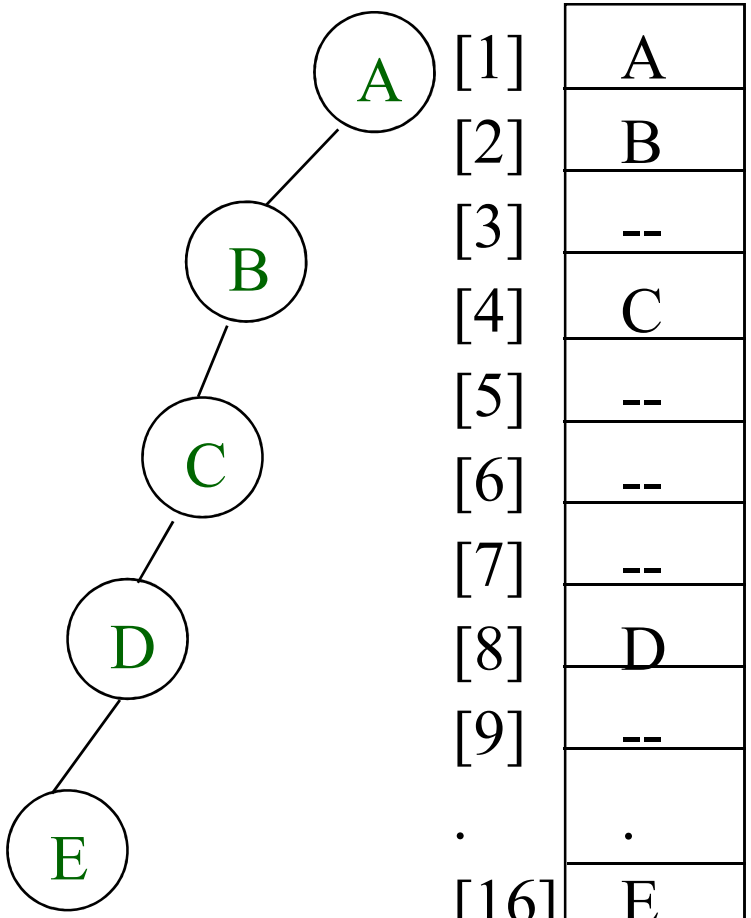
- If a complete binary tree with *n* nodes (depth = log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have:

  – *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.

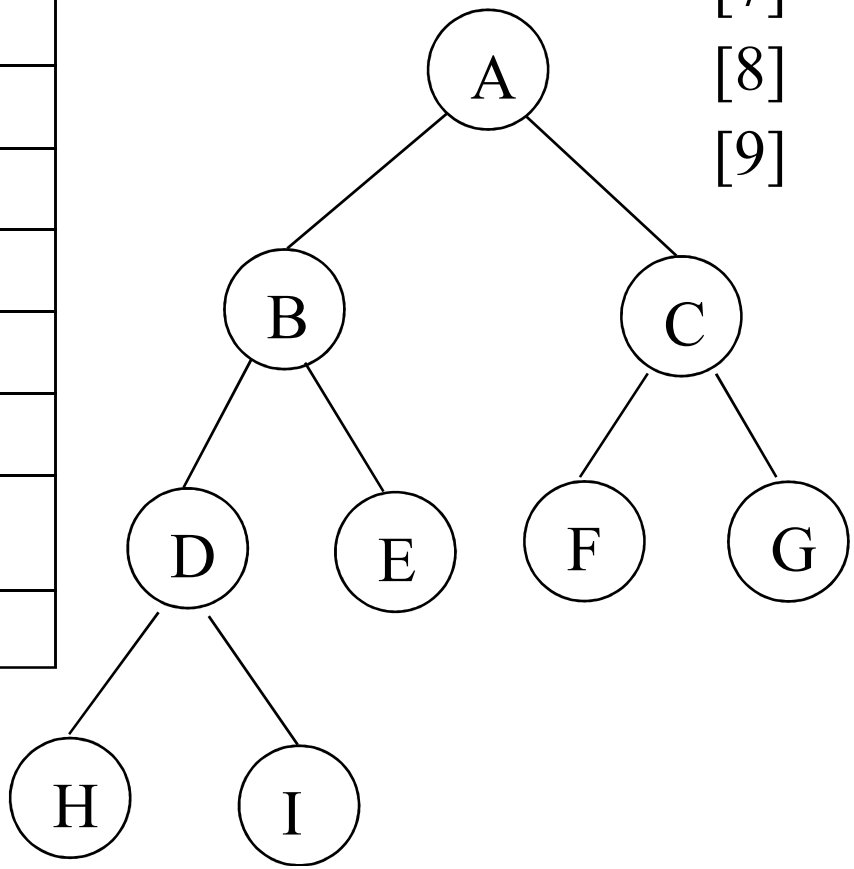
  – *left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child.

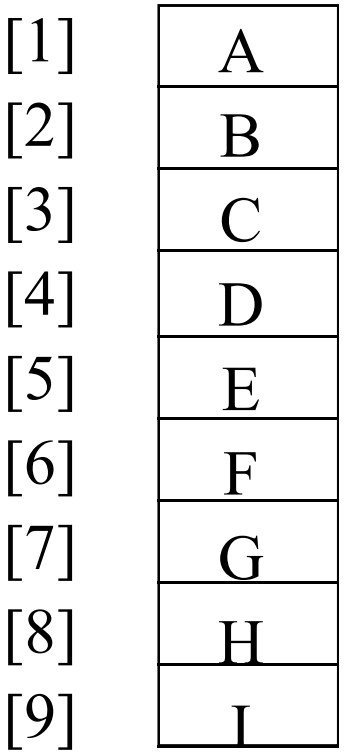
  – *right_child*(*i*) ia at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.

Sequential Representation



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





Example:

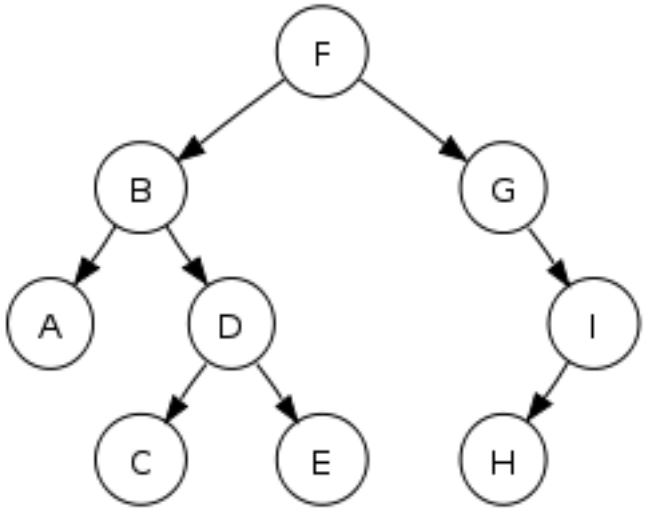
Refer to the binary tree to the right. In this binary tree,

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

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

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

Generally in-order traversal is used to represent algebraic equations.



Sequential representation

In sequential representation only one linear array is used.

The root R is stored at TREE [1].

If the node occupies TREE[K], then its left child is stored at in TREE[K*2] and

right child is stored in TREE[K*2 + 1].

TREE

1	F
2	B
3	G
4	A
5	D
6	NULL
7	I
8	NULL
9	NULL
10	C
11	E
12	NULL
13	NULL
14	H
15	NULL

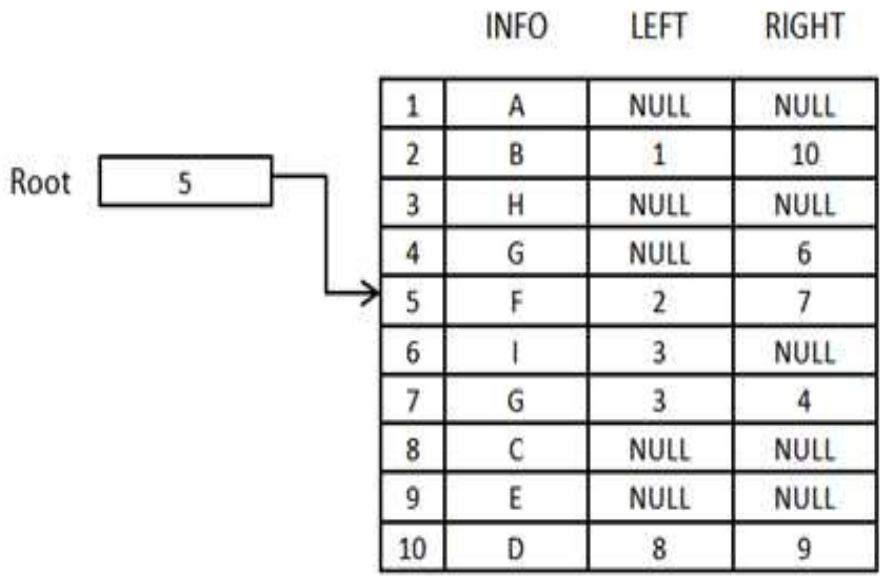
Representing binary trees in memory

Binary trees can be represented in memory either by linked representation or in a single array using sequential representation.

Linked representation

Linked representation uses three parallel arrays, INFO, LEFT and RIGHT and a pointer variable ROOT. Each node N of T will correspond to a location K such that –

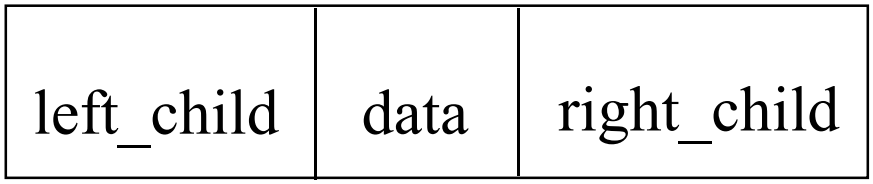
- INFO[K] contains the data at node N
- LEFT[K] contains the location of left child node N
- RIGHT[K] contains the location of right child node N
- ROOT will contain the location of root R of T

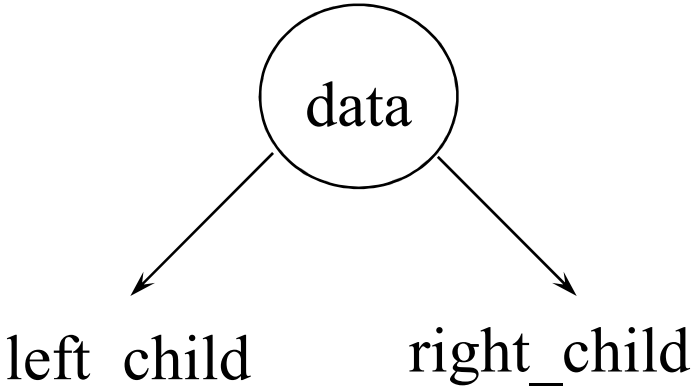


	INFO	LEFT	RIGHT
1	A	NULL	NULL
2	B	1	10
3	H	NULL	NULL
4	G	NULL	6
5	F	2	7
6	I	3	NULL
7	G	3	4
8	C	NULL	NULL
9	E	NULL	NULL
10	D	8	9

Linked Representation

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





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

Nonrecursive Preorder Traversal of the Binary Tree

PREORD(INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory. The algorithm does a preorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

1. [Initially push NULL onto STACK, and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Apply PROCESS to INFO[PTR].
4. [Right child?]
   If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
   Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
   [End of If structure.]
5. [Left child?]
   If LEFT[PTR] ≠ NULL, then:
   Set PTR := LEFT[PTR].
   Else: [Pop from STACK.]
   Set PTR := STACK[TOP] and TOP := TOP − 1.
   [End of If structure.]
   [End of Step 2 loop.]
6. Exit.

Nonrecursive Inorder Traversal of the Binary Tree

INORD(INFO, LEFT, RIGHT, ROOT)
A binary tree is in memory. This algorithm does an inorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK.]
   (*a*) Set TOP := TOP + 1 and STACK[TOP] := PTR. [Saves node.]
   (*b*) Set PTR := LEFT[PTR]. [Updates PTR.]
   [End of loop.]
3. Set PTR := STACK[TOP] and TOP := TOP − 1. [Pops node from STACK.]
4. Repeat Steps 5 to 7 while PTR ≠ NULL: [Backtracking.]
5. Apply PROCESS to INFO[PTR].
6. [Right child?] If RIGHT[PTR] ≠ NULL, then:
   (*a*) Set PTR := RIGHT[PTR].
   (*b*) Go to Step 3.
   [End of If structure.]
7. Set PTR := STACK[TOP] and TOP := TOP − 1. [Pops node.]
   [End of Step 4 loop.]
8. Exit.

Nonrecursive Postorder Traversal of the Binary Tree

POSTORD(INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. [Push left-most path onto STACK.]
   Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Set TOP := TOP + 1 and STACK[TOP] := PTR.
   [Pushes PTR on STACK.]
4. If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
   Set TOP := TOP + 1 and STACK[TOP] := −RIGHT[PTR].
   [End of If structure.]
5. Set PTR := LEFT[PTR]. [Updates pointer PTR.]
   [End of Step 2 loop.]
6. Set PTR := STACK[TOP] and TOP := TOP − 1.
   [Pops node from STACK.]
7. Repeat while PTR > 0:
   (*a*) Apply PROCESS to INFO[PTR].
   (*b*) Set PTR := STACK[TOP] and TOP := TOP − 1.
   [Pops node from STACK.]
   [End of loop.]
8. If PTR < 0, then:
   (*a*) Set PTR := −PTR.
   (*b*) Go to Step 2.
   [End of If structure.]
9. Exit.

Nonrecursive Inorder Traversal: General Algorithm

```
1.  current = root;  //start traversing the binary tree at
                     // the root node
2.  while(current is not NULL and stack is nonempty)
    if(current is not NULL)
    {
       push current onto stack;
       current = current->llink;
    }
    else
    {
       pop stack into current;
       visit current;    // visit and Print the node
       current = current->rlink;     //move to the
                                     //right child
    }
```

Nonrecursive Preorder Traversal: General Algorithm

```
1. current = root;  //start the traversal at the root node
2. while(current is not NULL and stack is nonempty)
    if(current is not NULL)
    {
       visit current;     // visit and Print the node
       push current onto stack;
       current = current->llink;
    }
    else
    {
       pop stack into current;
       current = current->rlink;    //prepare to visit
                                    //the right subtree
    }
```

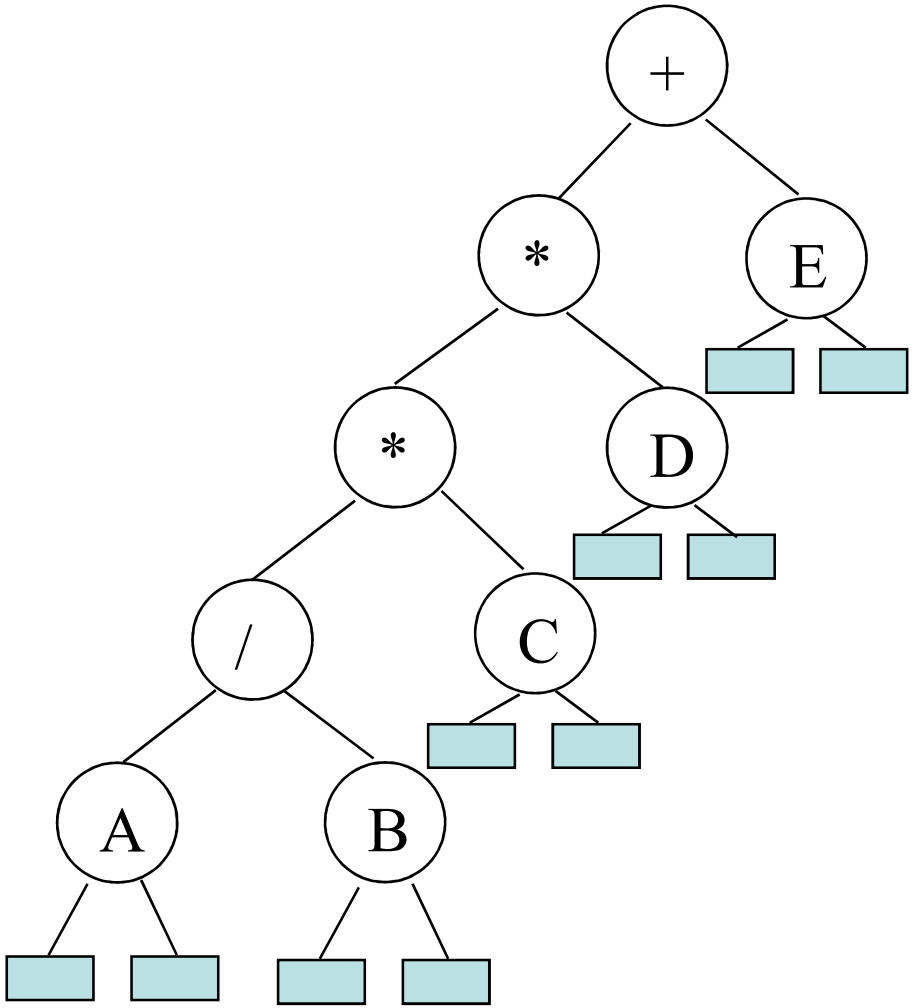
Nonrecursive Postorder Traversal

```
1.  current = root;  //start traversal at root node
2.  v = 0;
3.  if(current is NULL)
       the binary tree is empty
4.  if(current is not NULL)
   a.  push current into stack;
   b.  push 1 onto stack;
   c.  current = current->llink;
   d.  while(stack is not empty)
          if(current is not NULL and v is 0)
          {
               push current and 1 onto stack;
               current = current->llink;
          }
```

Nonrecursive Postorder Traversal (Continued)

```
else
{
        pop stack into current and v;
        if(v == 1)
        {
                push current and 2 onto stack;
                current = current->rlink;
                v = 0;
        }
        else
                visit current;
 }
```

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)

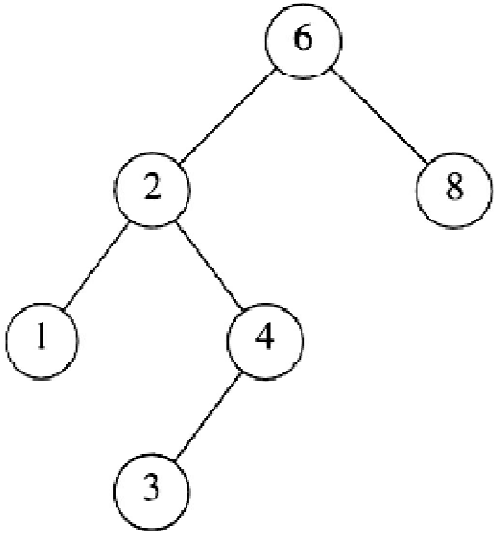
```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
     if (ptr) {
          postorder(ptr->left_child);
          postdorder(ptr->right_child);
          printf("%d", ptr->data);
     }
}
```

A B / C * D * E +

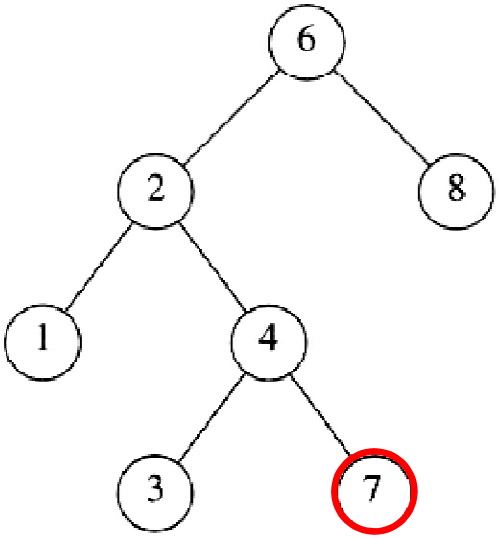
Binary Search Trees

- Data in each node
  - Larger than the data in its left child
  - Smaller than the data in its right child
- A **binary search tree**,*t*, is either empty or:
  - *T* has a special node called the **root** node
  - *T* has two sets of nodes, *LT* and *RT*, called the left subtree and right subtree of *T*, respectively
  - Key in root node larger than every key in left subtree and smaller than every key in right subtree
  - *LT* and *RT* are binary search trees

Binary Search Trees



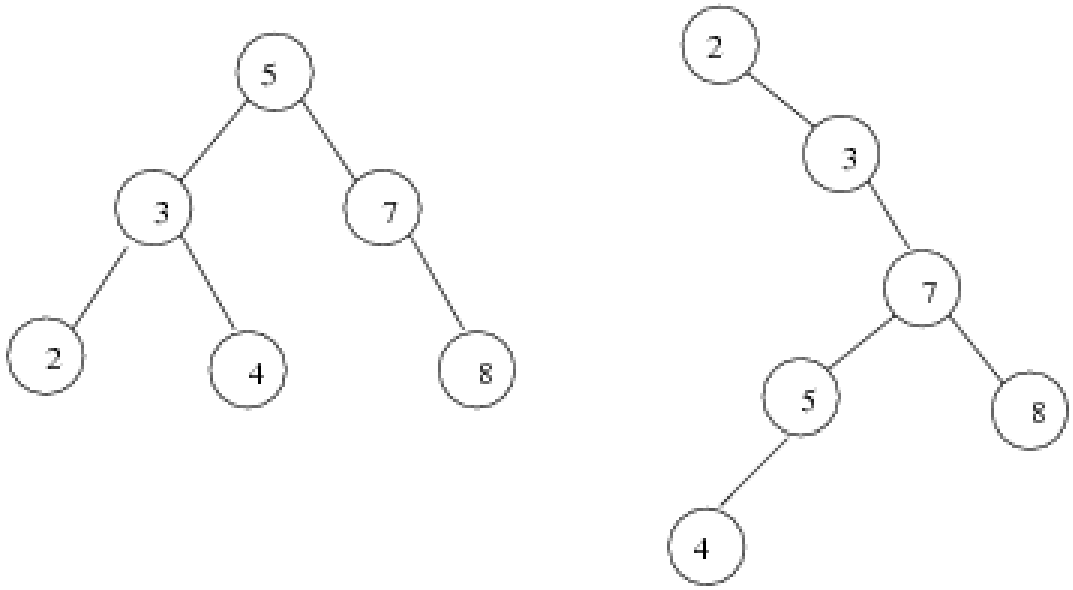
A binary search tree



Not a binary search tree

Binary search trees

Two binary search trees representing the same set:



- Average depth of a node is O(log N); maximum depth of a node is O(N)

Find an element in Binary Search Tree

FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
A binary search tree T is in memory and an ITEM of information is given. This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM. There are three special cases:

(i) LOC = NULL and PAR = NULL will indicate that the tree is empty.
(ii) LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root of T.
(iii) LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

1. [Tree empty?]
   If ROOT = NULL, then: Set LOC := NULL and PAR := NULL, and Return.
2. [ITEM at root?]
   If ITEM = INFO[ROOT], then: Set LOC := ROOT and PAR := NULL, and Return.
3. [Initialize pointers PTR and SAVE.]
   If ITEM < INFO[ROOT], then:
      Set PTR := LEFT[ROOT] and SAVE := ROOT.
   Else:
      Set PTR := RIGHT[ROOT] and SAVE := ROOT.
   [End of If structure.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5.   [ITEM found?]
      If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE, and Return.
6.   If ITEM < INFO[PTR], then:
         Set SAVE := PTR and PTR := LEFT[PTR].
      Else:
         Set SAVE := PTR and PTR := RIGHT[PTR].
      [End of If structure.]
   [End of Step 4 loop.]
7. [Search unsuccessful.] Set LOC := NULL and PAR := SAVE.
8. Exit.

Insert an element in Binary Search Tree

INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)
A binary search tree T is in memory and an ITEM of information is given. This algorithm finds the location LOC of ITEM in T or adds ITEM as a new node in T at location LOC.

1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
   [Procedure 7.4.]
2. If LOC ≠ NULL, then Exit.
3. [Copy ITEM into new node in AVAIL list.]
   (*a*) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
   (*b*) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and
   INFO[NEW] := ITEM.
   (*c*) Set LOC := NEW, LEFT[NEW] := NULL and
   RIGHT[NEW] := NULL.
4. [Add ITEM to tree.]
   If PAR = NULL, then:
   Set ROOT := NEW.
   Else if ITEM < INFO[PAR], then:
   Set LEFT[PAR] := NEW.
   Else:
   Set RIGHT[PAR] := NEW.
   [End of If structure.]
5. Exit.

Delete an element in Binary Search Tree

DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)
A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

1. [Find the locations of ITEM and its parent, using Procedure 7.4.]
   Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
2. [ITEM in tree?]
   If LOC = NULL, then: Write: ITEM not in tree, and Exit.
3. [Delete node containing ITEM.]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL, then:
      Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   Else:
      Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LEFT[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

Delete an element in Binary Search Tree (CASE – A)

```
CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure deletes the node N at location LOC, where N does not have two
children. The pointer PAR gives the location of the parent of N, or else
PAR = NULL indicates that N is the root node. The pointer CHILD gives the
location of the only child of N, or else CHILD = NULL indicates N has no
children.

1.  [Initializes CHILD.]
    If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then:
        Set CHILD := NULL.
    Else if LEFT[LOC] ≠ NULL, then:
        Set CHILD := LEFT[LOC].
    Else
        Set CHILD := RIGHT[LOC].
    [End of If structure.]
2.  If PAR ≠ NULL, then:
        If LOC = LEFT[PAR], then:
            Set LEFT[PAR] := CHILD.
        Else:
            Set RIGHT[PAR] := CHILD.
        [End of If structure.]
    Else:
        Set ROOT := CHILD.
    [End of If structure.]
3.  Return.
```

Delete an element in Binary Search Tree (CASE – B)

CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)
This procedure will delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

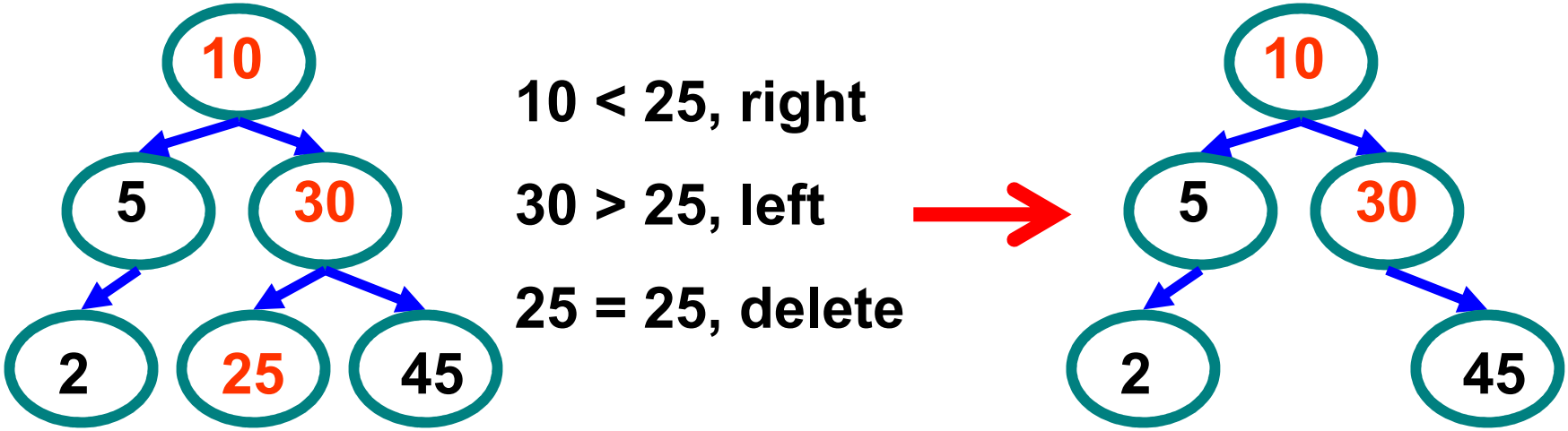
1. [Find SUC and PARSUC.]
   (*a*) Set PTR := RIGHT[LOC] and SAVE := LOC.
   (*b*) Repeat while LEFT[PTR] ≠ NULL:
         Set SAVE := PTR and PTR := LEFT[PTR].
         [End of loop.]
   (*c*) Set SUC := PTR and PARSUC := SAVE.
2. [Delete inorder successor, using Procedure 7.6.]
   Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).
3. [Replace node N by its inorder successor.]
   (*a*) If PAR ≠ NULL, then:
         If LOC = LEFT[PAR], then:
            Set LEFT[PAR] := SUC.
         Else:
            Set RIGHT[PAR] := SUC.
         [End of If structure.]
      Else:
         Set ROOT := SUC.
      [End of If structure.]
   (*b*) Set LEFT[SUC] := LEFT[LOC] and
      RIGHT[SUC] := RIGHT[LOC].
4. Return.

Binary Search Tree – Deletion

- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node
     a) Replace with largest value Y on left subtree
        OR smallest value Z on right subtree
     b) Delete replacement value (Y or Z) from subtree
- Observation
  - O(log(n)) operation for balanced tree
  - Deletions may unbalance tree

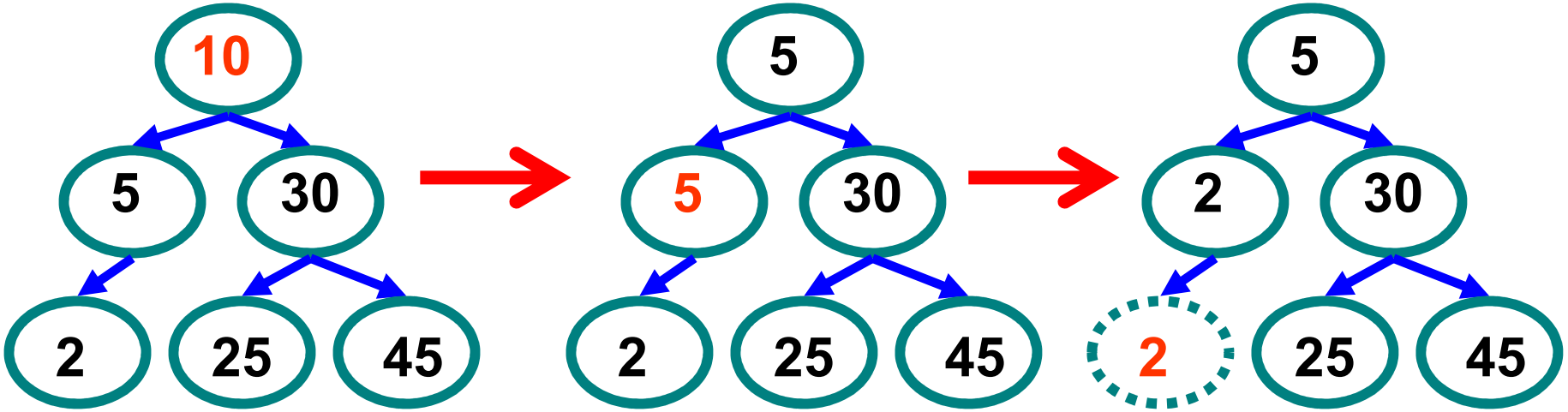
Example Deletion (Leaf)

- Delete (25)



Example Deletion (Internal Node)

- Delete (10)



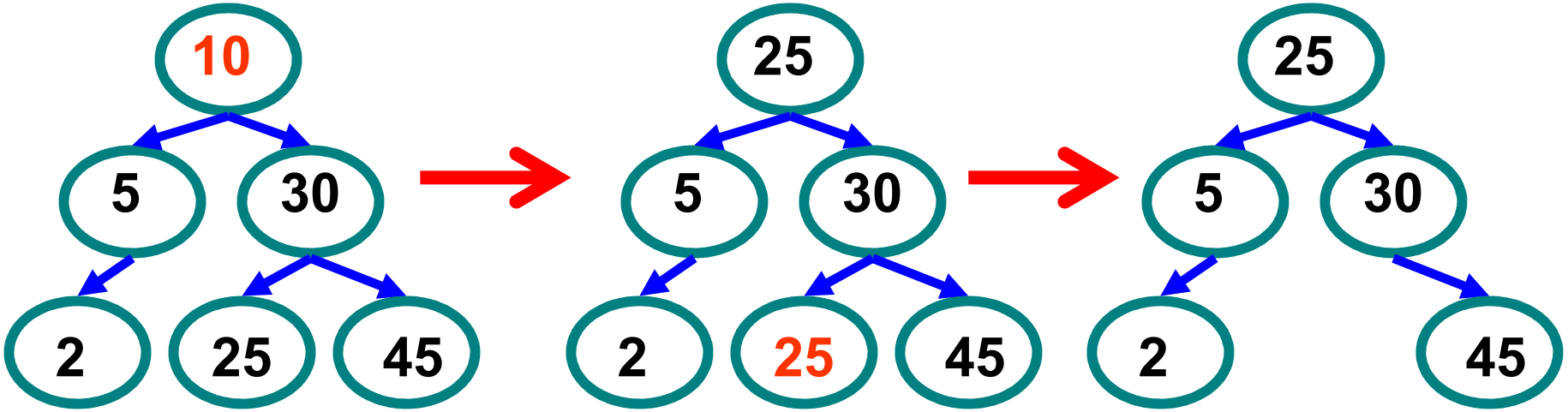
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

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*)

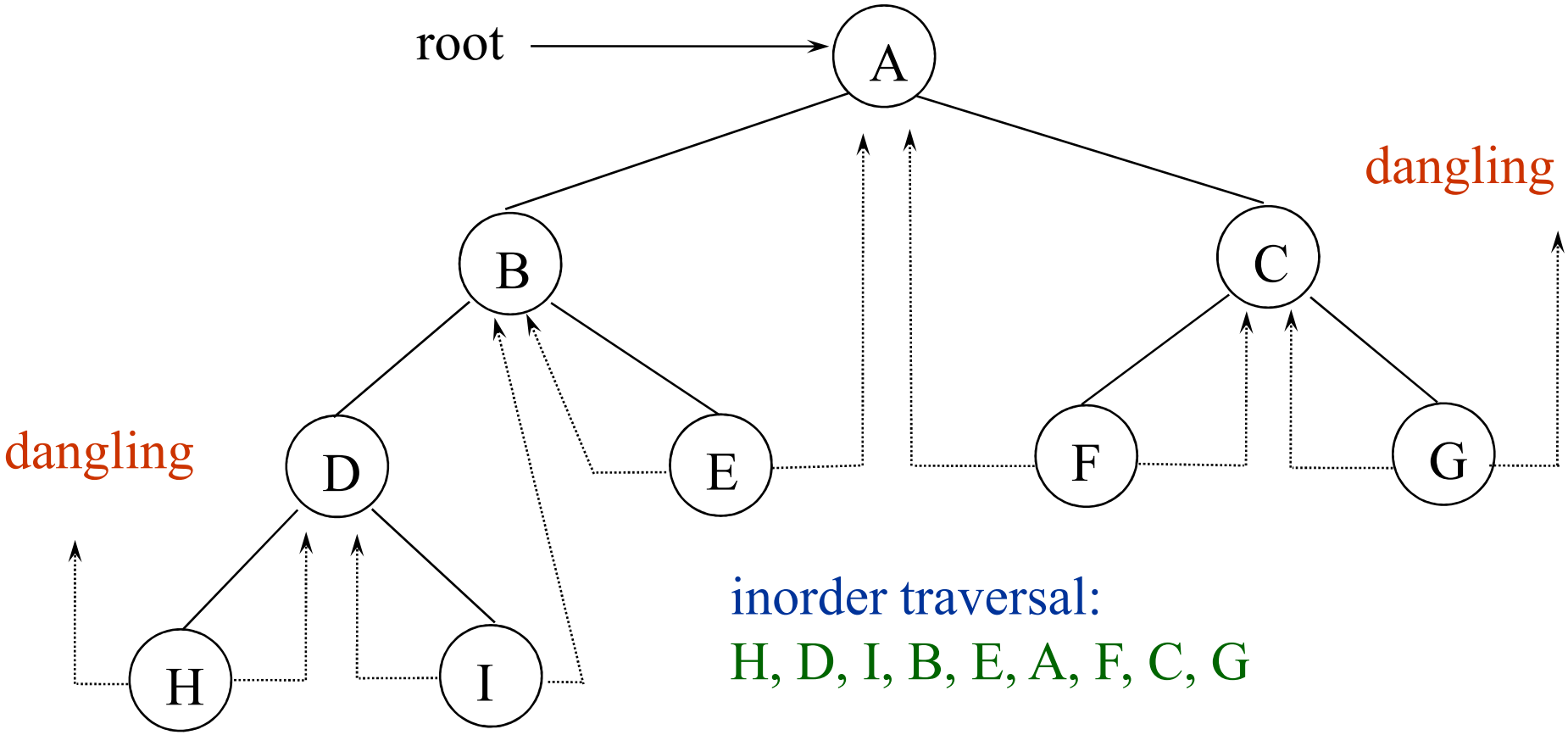
If `ptr->left_child` is null,

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

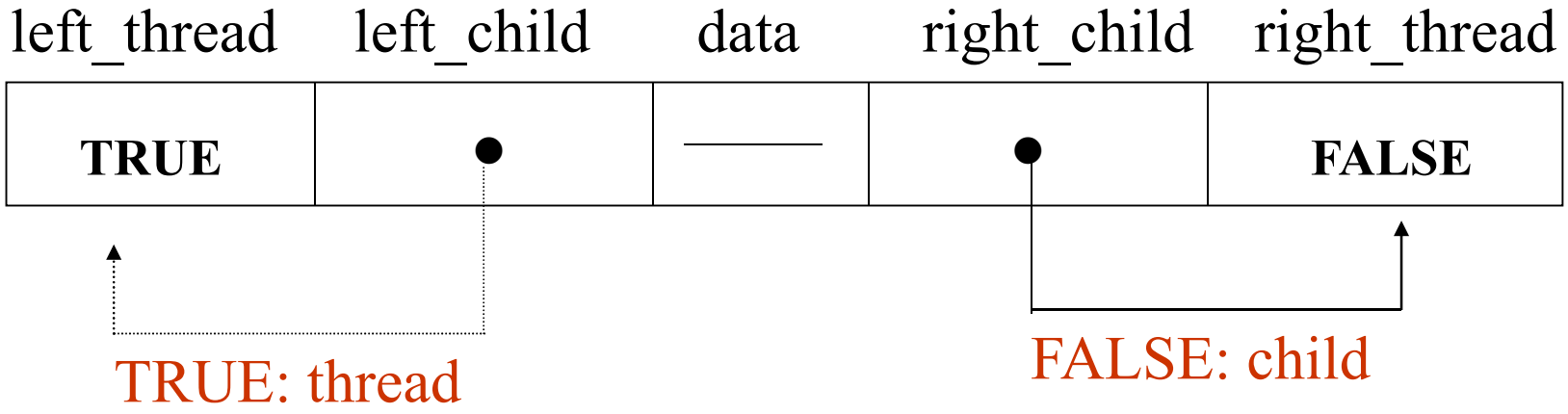
If `ptr->right_child` is null,

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

A Threaded Binary Tree

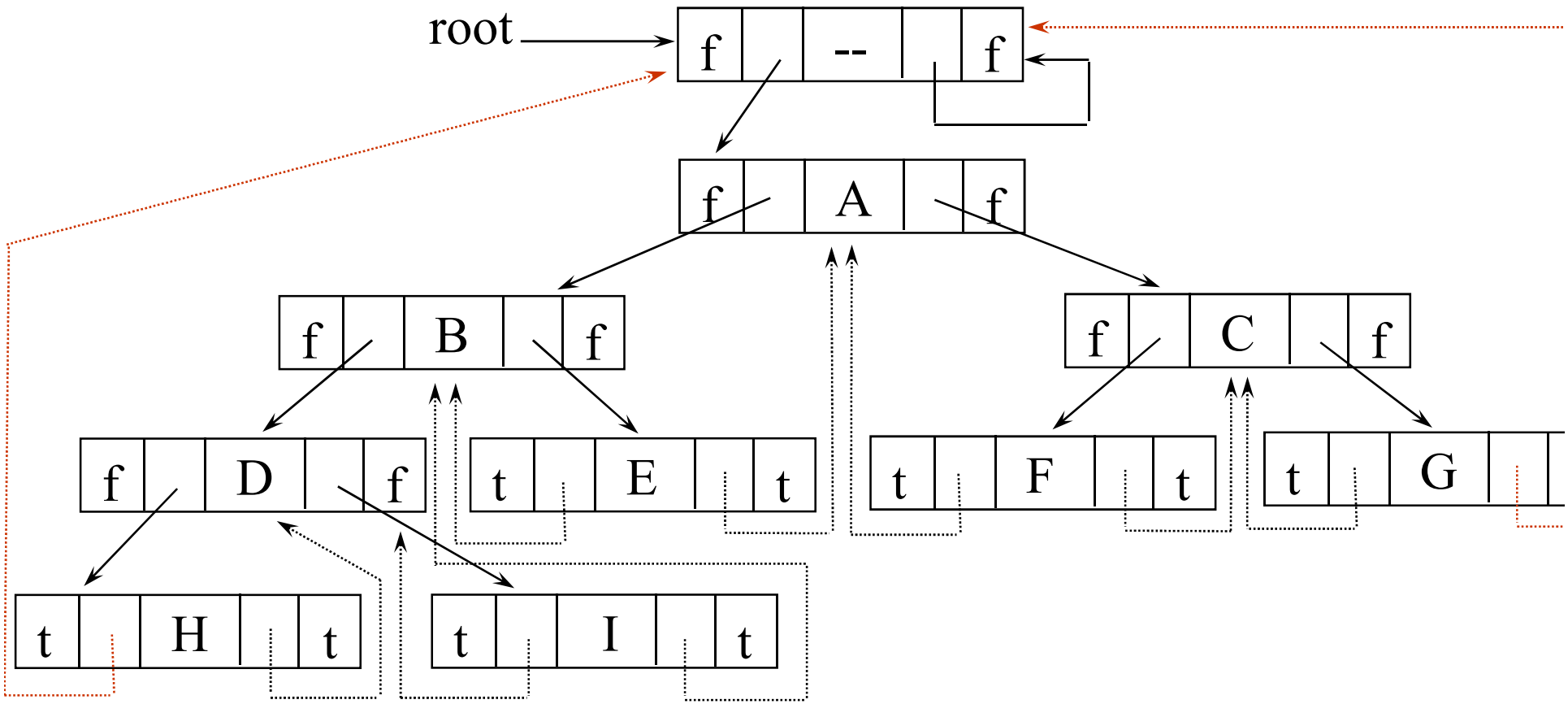


Data Structures for Threaded BT

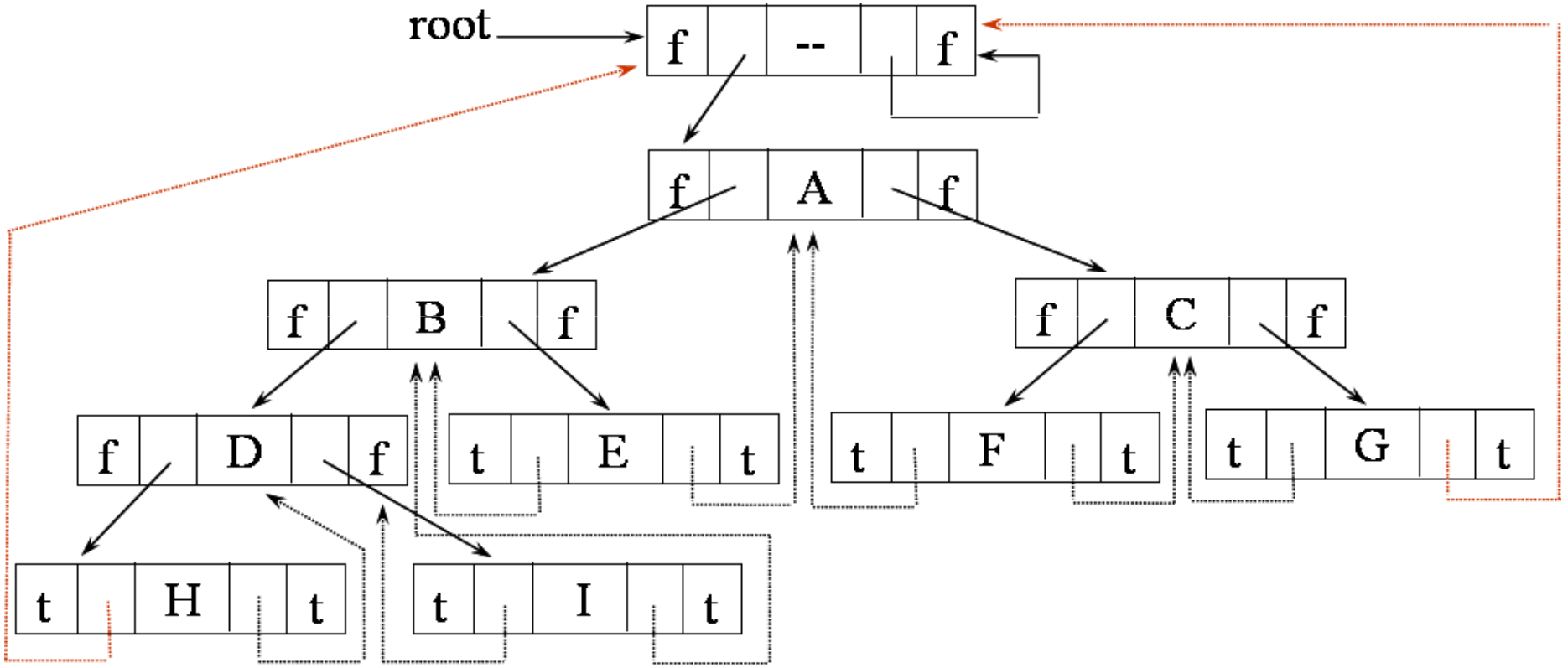



```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;  };
```

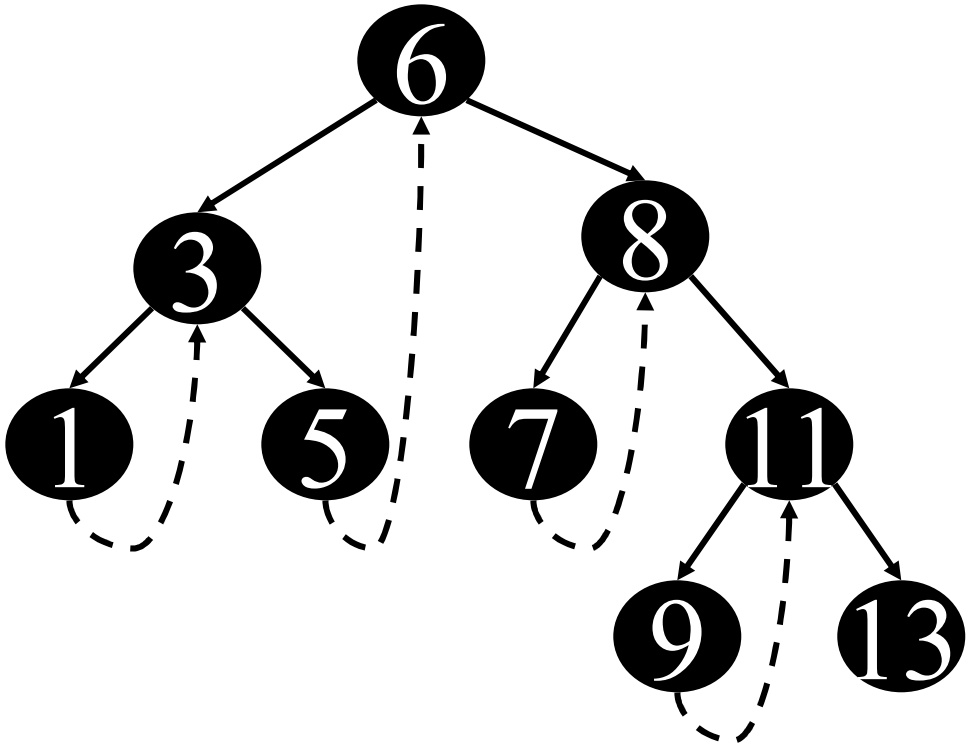
Memory Representation of A Threaded BT



Memory Representation of A Threaded BT



Threaded Tree Example

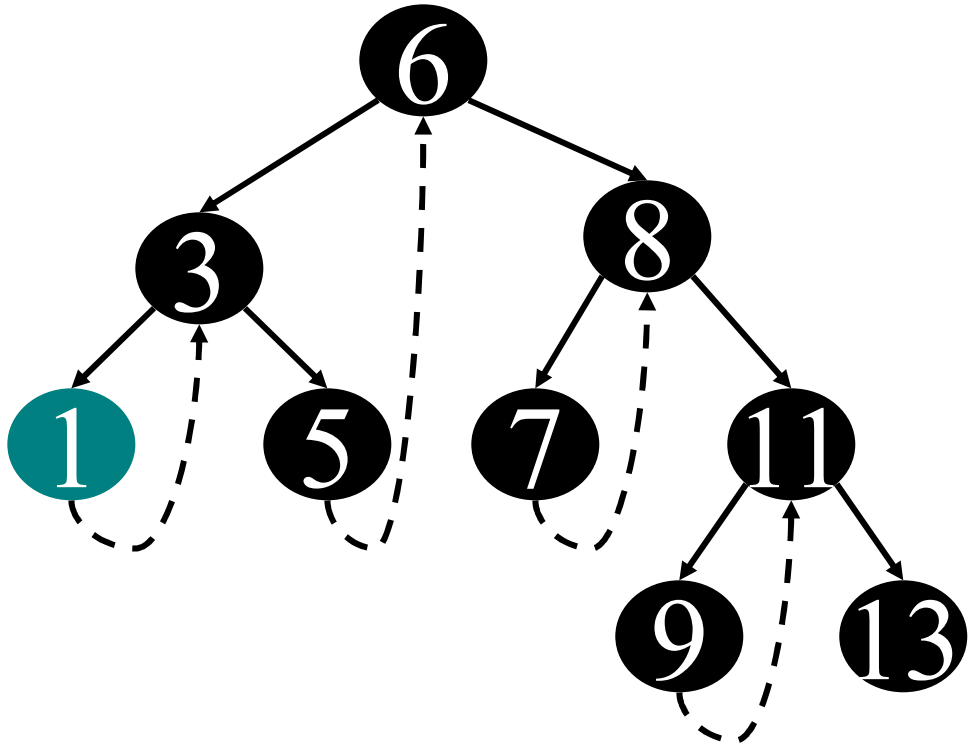


Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue

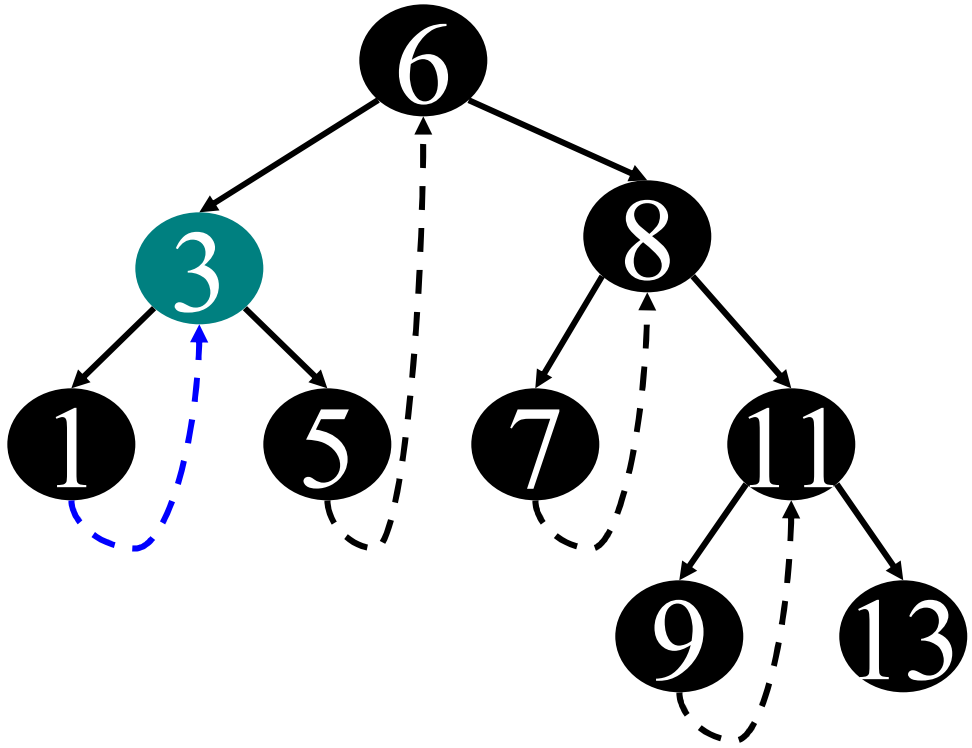
Threaded Tree Traversal

Output
1



Start at leftmost node, print it

Threaded Tree Traversal



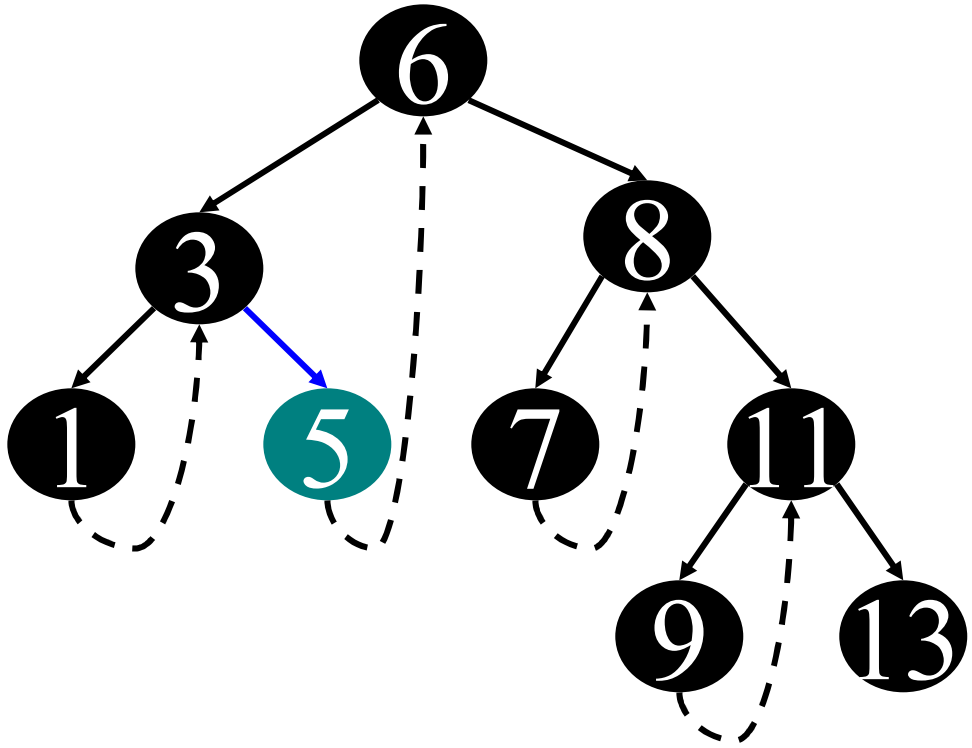
Output

1

3

Follow thread to right, print node

Threaded Tree Traversal



Output

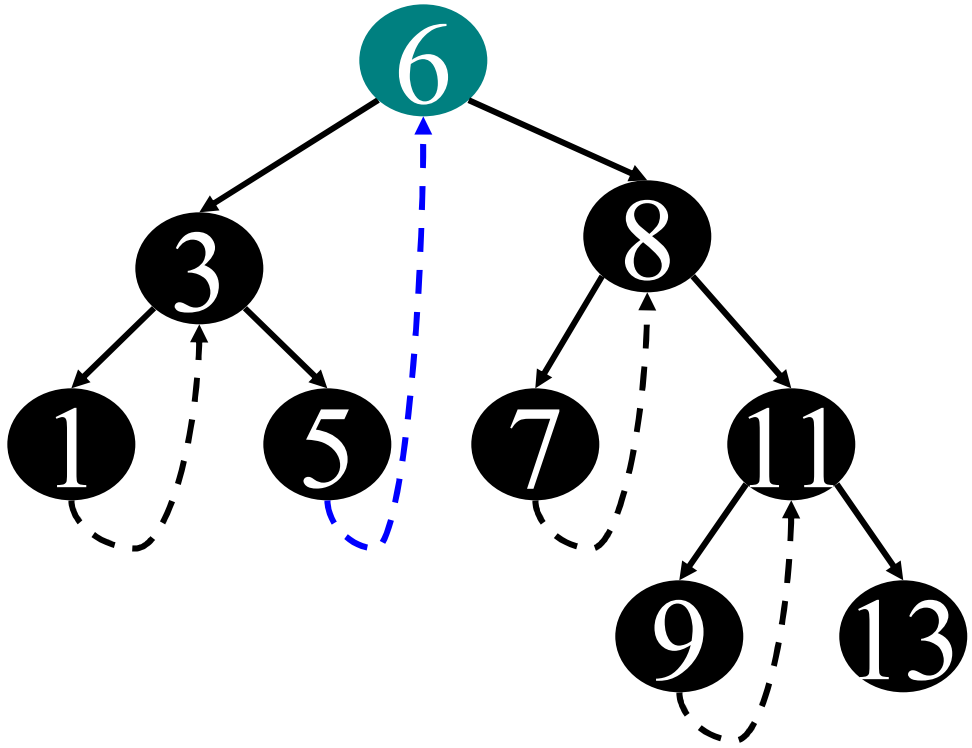
1

3

5

Follow link to right, go to leftmost node and print

Threaded Tree Traversal



Output

1

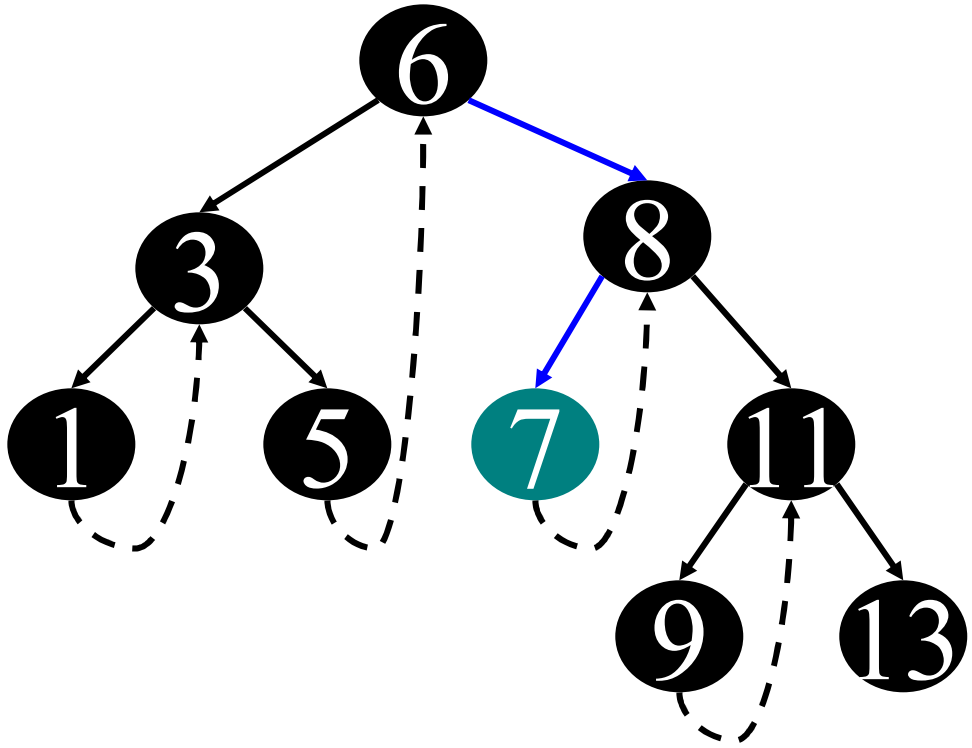
3

5

6

Follow thread to right, print node

Threaded Tree Traversal

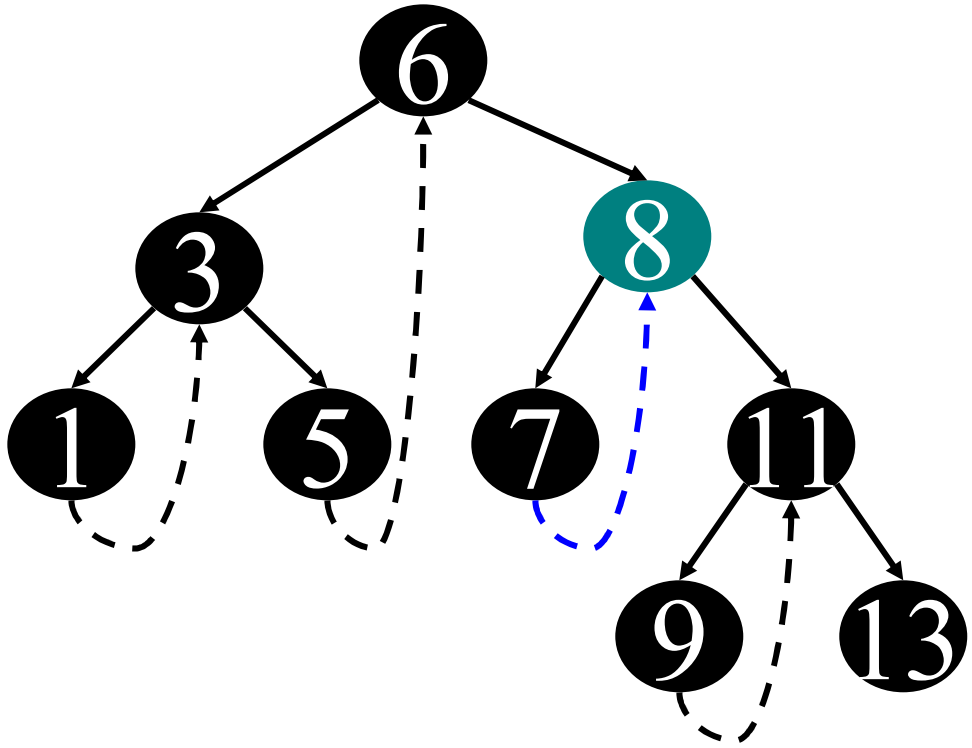


Output

1
3
5
6
7

Follow link to right, go to leftmost node and print

Threaded Tree Traversal

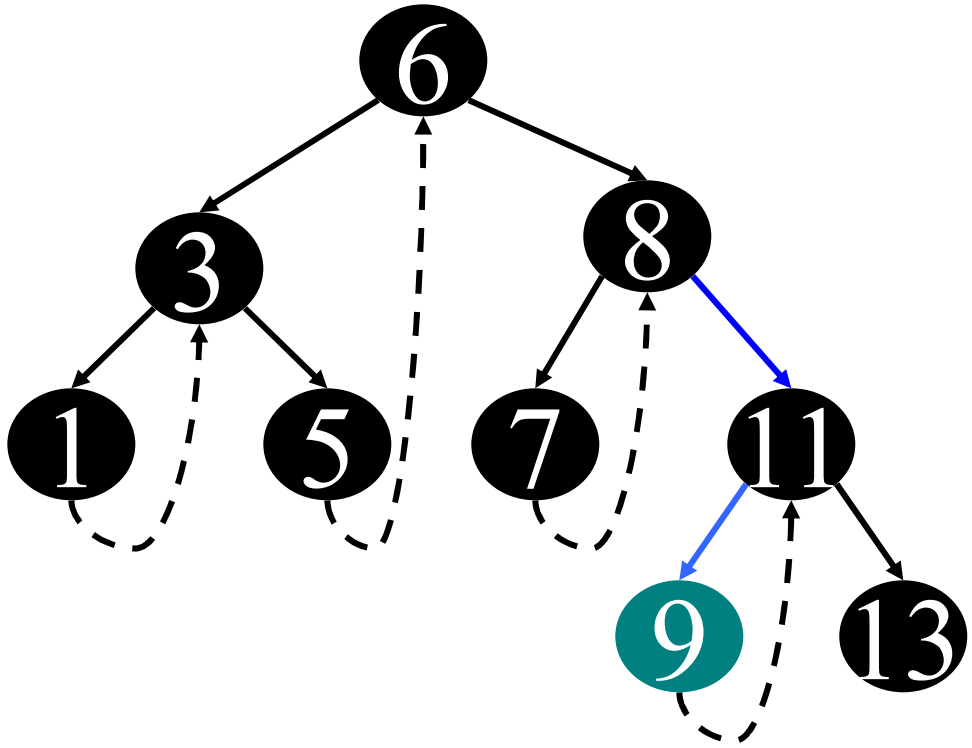


Output

1
3
5
6
7
8

Follow thread to right, print node

Threaded Tree Traversal

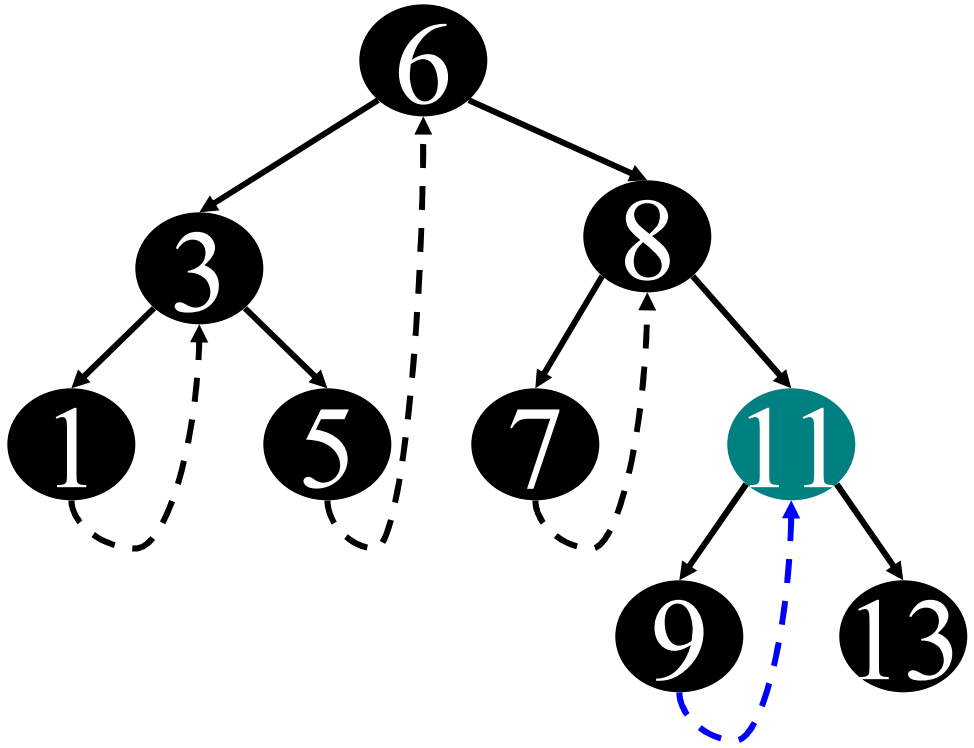


Output

1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

Threaded Tree Traversal

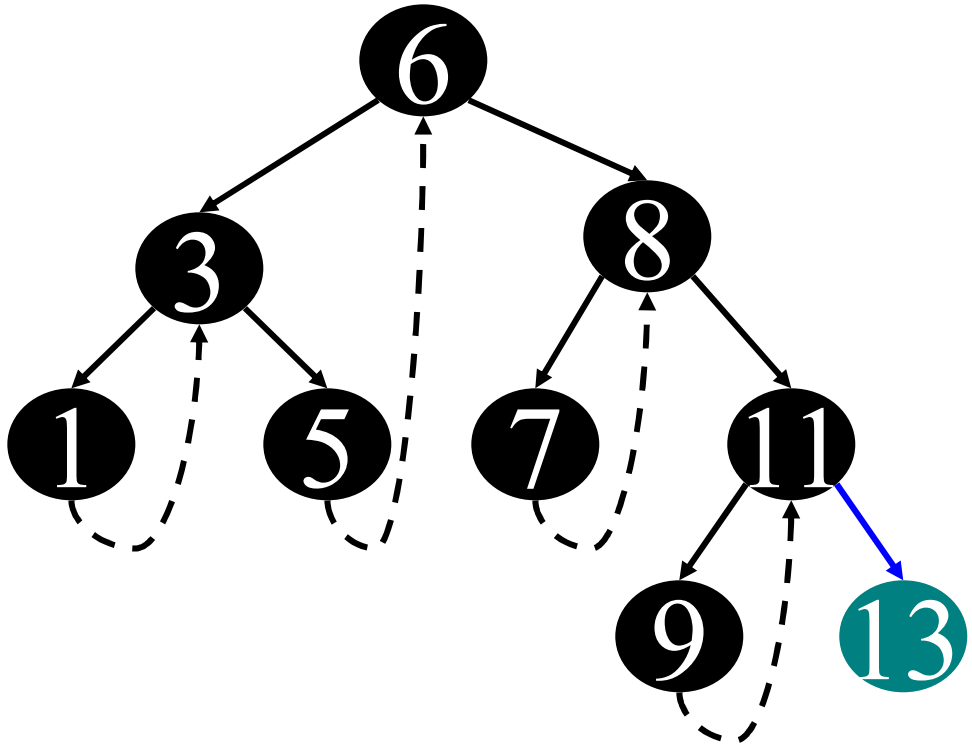


Output

1
3
5
6
7
8
9
11

Follow thread to right, print node

Threaded Tree Traversal



Output

1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

Threaded Tree Traversal Code

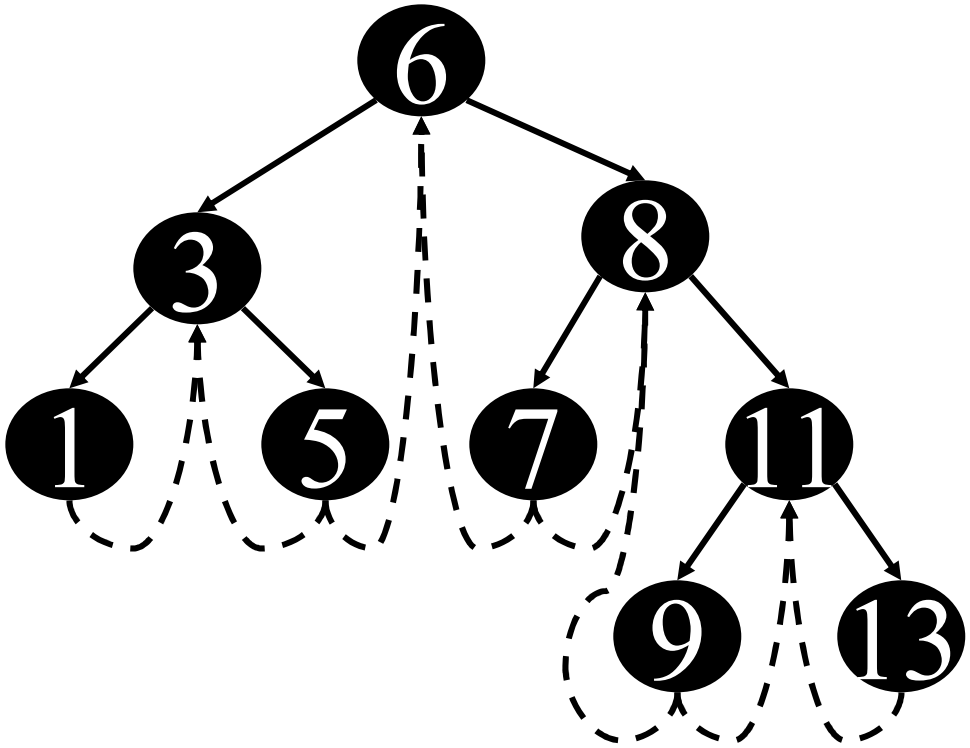
```
Node leftMost(Node n)
{
    Node ans = n;
    if (ans == null)
   {
        return null;
    }
    while (ans.left != null)
    {
        ans = ans.left;
    }
    return ans;
}
```

```
void inOrder(Node n)
{
    Node cur = leftmost(n);
    while (cur != null)
  {
        print(cur);
        if (cur.rightThread)
       {
            cur = cur.right;
        } else
        {
            cur = leftmost(cur.right);
        }
    }
}
```

Threaded Tree Modification

- We’re still wasting pointers, since half of our leafs’ pointers are still null
- We can add threads to the previous node in an inorder traversal as well, which we can use to traverse the tree backwards or even to do postorder traversals

Threaded Tree Modification



Creating a Heap Tree

HEAPSORT(A, N)
An array A with N elements is given. This algorithm sorts the elements of A.

1. [Build a heap H, using Procedure 7.9.]
   Repeat for J = 1 to N − 1:
      Call INSHEAP(A, J, A[J + 1]).
   [End of loop.]
2. [Sort A by repeatedly deleting the root of H, using Procedure 7.10.]
   Repeat while N > 1:
      (*a*) Call DELHEAP(A, N, ITEM).
      (*b*) Set A[N + 1] := ITEM.
   [End of Loop.]
3. Exit.

Insert into Heap Tree

INSHEAP(TREE, N, ITEM)
A heap H with N elements is stored in the array TREE, and an ITEM of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

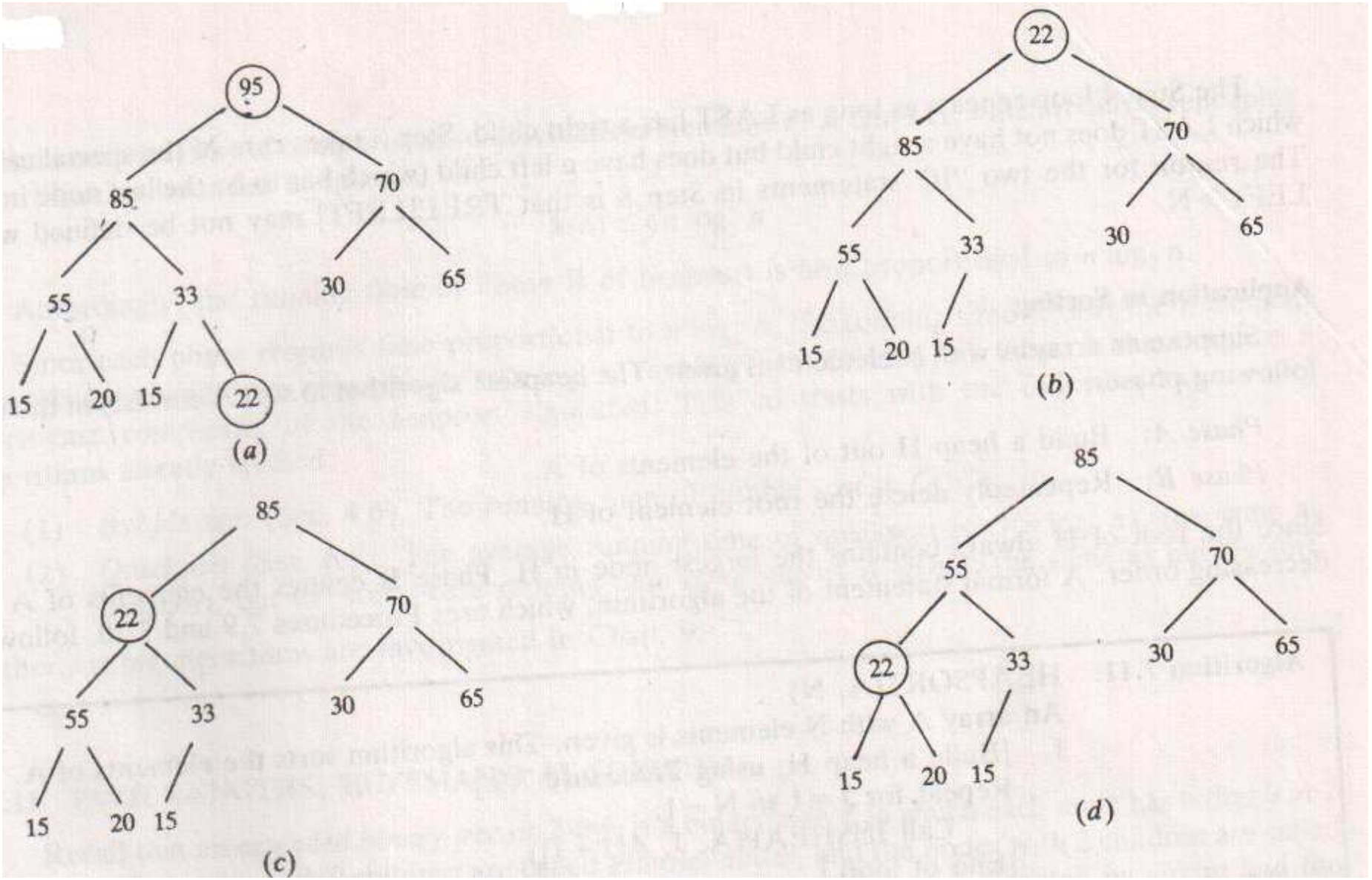
1. [Add new node to H and initialize PTR.]
   Set N := N + 1 and PTR := N.
2. [Find location to insert ITEM.]
   Repeat Steps 3 to 6 while PTR < 1.
3. Set PAR := [PTR/2]. [Location of parent node.]
4. If ITEM ≤ TREE[PAR], then:
   Set TREE[PTR] := ITEM, and Return.
   [End of If structure.]
5. Set TREE[PTR] := TREE[PAR]. [Moves node down.]
6. Set PTR := PAR. [Updates PTR.]
   [End of Step 2 loop.]
7. [Assign ITEM as the root of H.]
   Set TREE[1] := ITEM.
8. Return.

Deletion From Heap Tree

DELHEAP(TREE, N, ITEM)
A heap H with N elements is stored in the array TREE. This procedure assigns the root TREE[1] of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

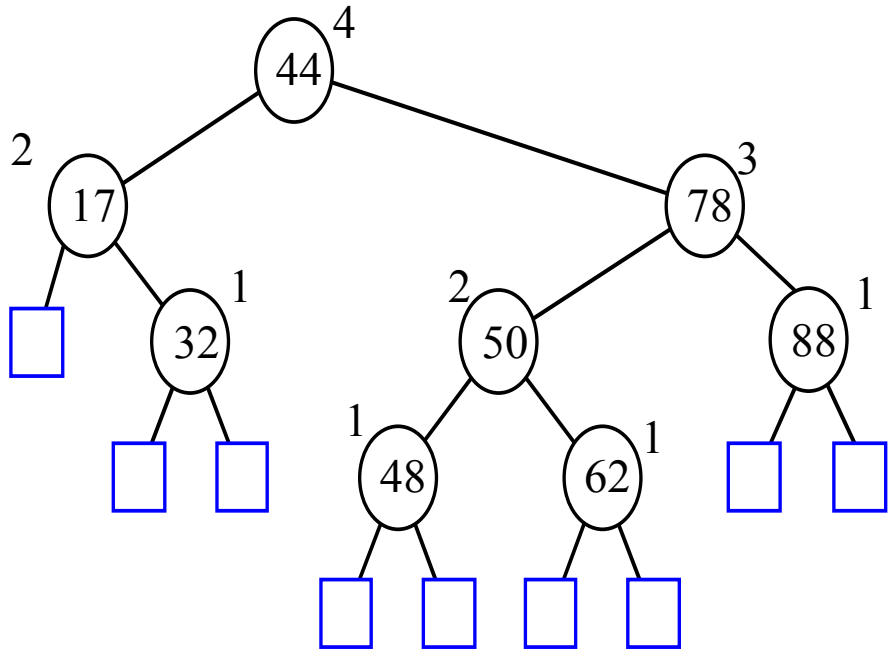
1. Set ITEM := TREE[1]. [Removes root of H.]
2. Set LAST := TREE[N] and N := N − 1. [Removes last node of H.]
3. Set PTR := 1, LEFT := 2 and RIGHT := 3. [Initializes pointers.]
4. Repeat Steps 5 to 7 while RIGHT ≤ N:
5. If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT], then:
   Set TREE[PTR] := LAST and Return.
   [End of If structure.]
6. IF TREE[RIGHT] ≤ TREE[LEFT], then:
   Set TREE[PTR] := TREE[LEFT] and PTR := LEFT.
   Else:
   Set TREE[PTR] := TREE[RIGHT] and PTR := RIGHT.
   [End of If structure.]
7. Set LEFT := 2 * PTR and RIGHT := LEFT + 1.
   [End of Step 4 loop.]
8. If LEFT = N and if LAST < TREE[LEFT], then: Set PTR := LEFT.
9. Set TREE[PTR] := LAST.
10. Return.

Deletion From Heap Tree



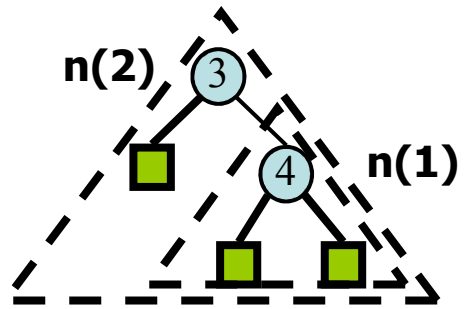
AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1*.



An example of an AVL tree where the heights are shown next to the nodes:

Theorem : Height of an AVL Tree



- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
  - We easily see that n(1) = 1 and n(2) = 2
  - For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
  - That is, n(h) = 1 + n(h-1) + n(h-2)
  - Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
    n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),
    $n(h) > 2^i n(h-2i)$
  - Solving the base case we get: $n(h) > 2^{h/2-1}$
  - Taking logarithms: h < 2log n(h) +2
  - Thus the height of an AVL tree is O(log n)

AVL (Height-balanced Trees)

- A **perfectly balanced** binary tree is a binary tree such that:
  - The height of the left and right subtrees of the root are equal
  - The left and right subtrees of the root are perfectly balanced binary trees

Perfectly Balanced Binary Tree

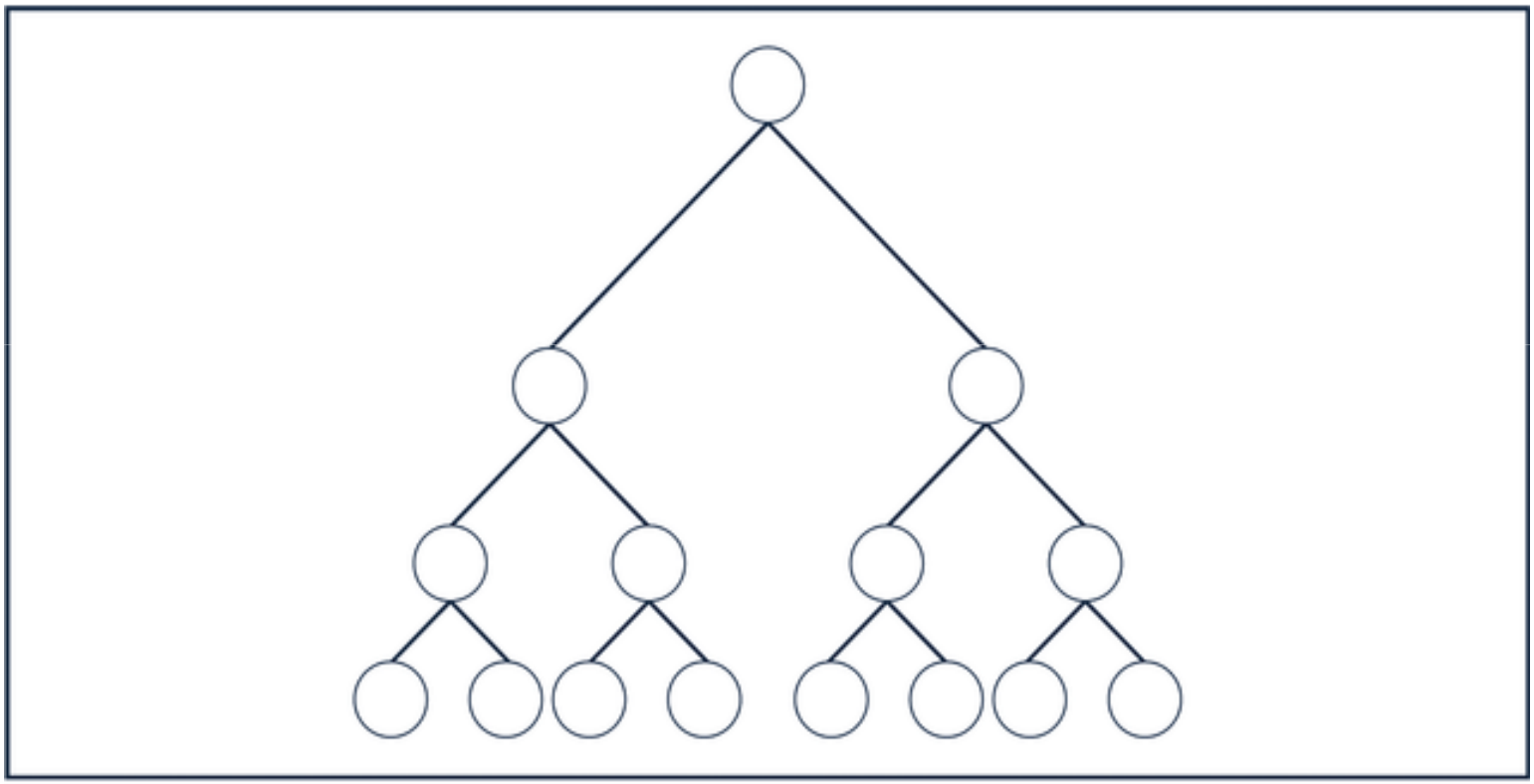


Figure 11-18 Perfectly balanced binary tree

AVL (Height-balanced Trees)

- An **AVL tree** (or **height-balanced tree**) is a binary search tree such that:
  - The height of the left and right subtrees of the root differ by at most 1
  - The left and right subtrees of the root are AVL trees

AVL Trees

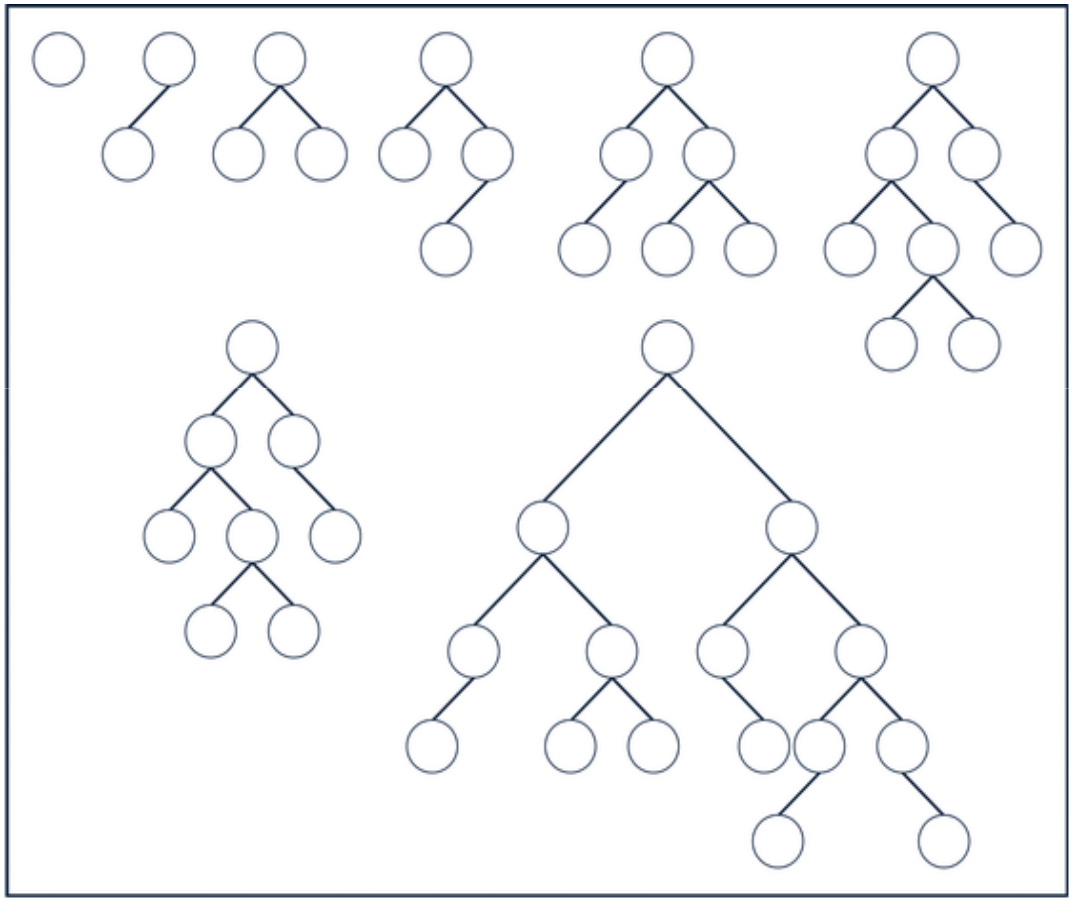


Figure 11-19 AVL trees

Non-AVL Trees

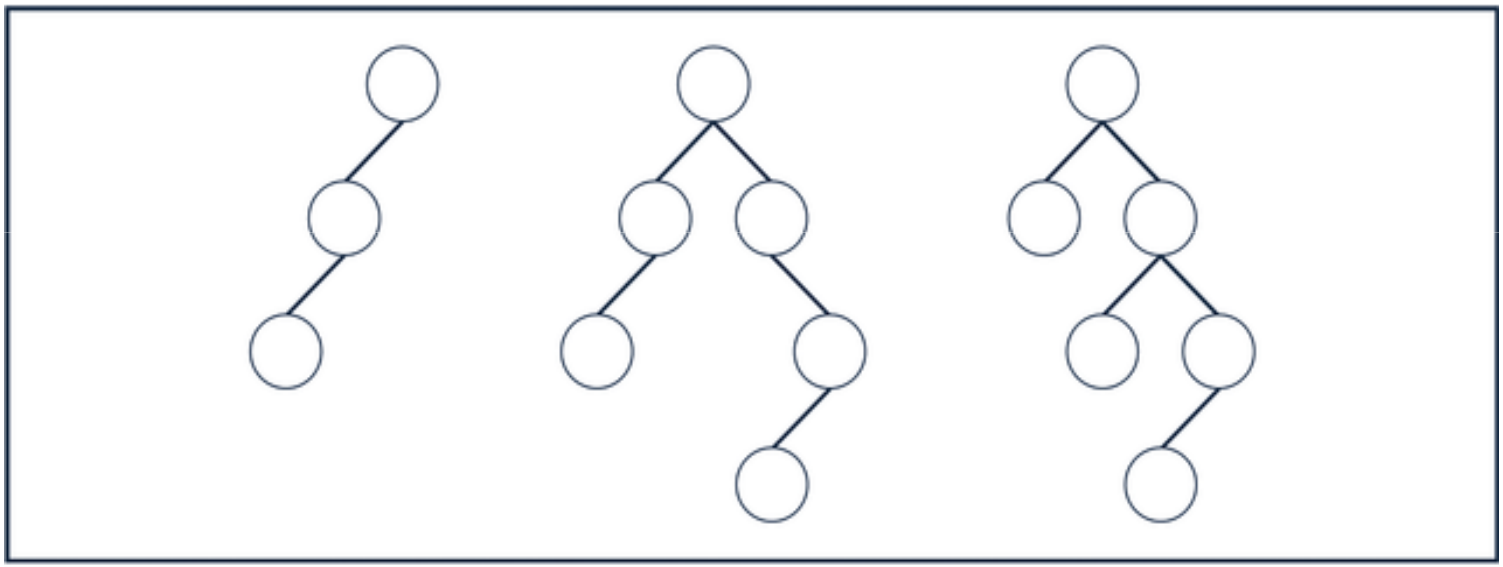


Figure 11-20 Non-AVL trees

Insertion Into AVL Tree

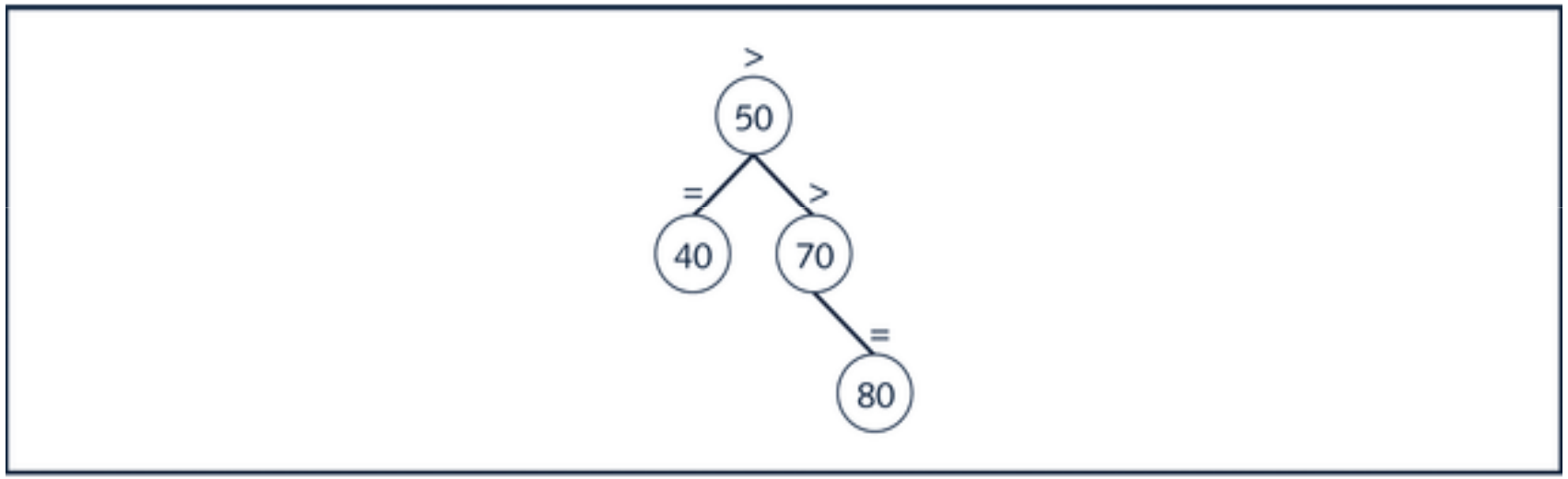


Figure 11-21 AVL tree before inserting 90

Insertion Into AVL Trees

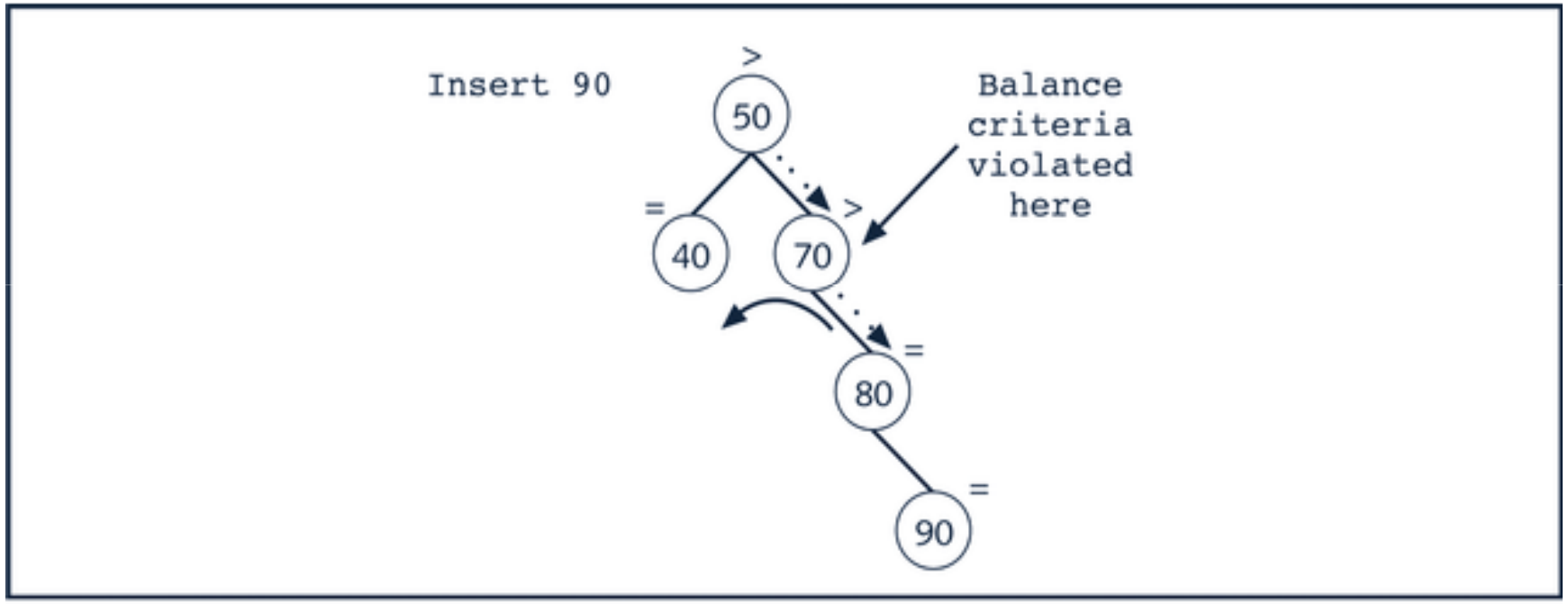


Figure 11-22 Binary tree of Figure 11-21 after inserting 90; nodes other than 90 show their balance factors before insertion

Insertion Into AVL Trees

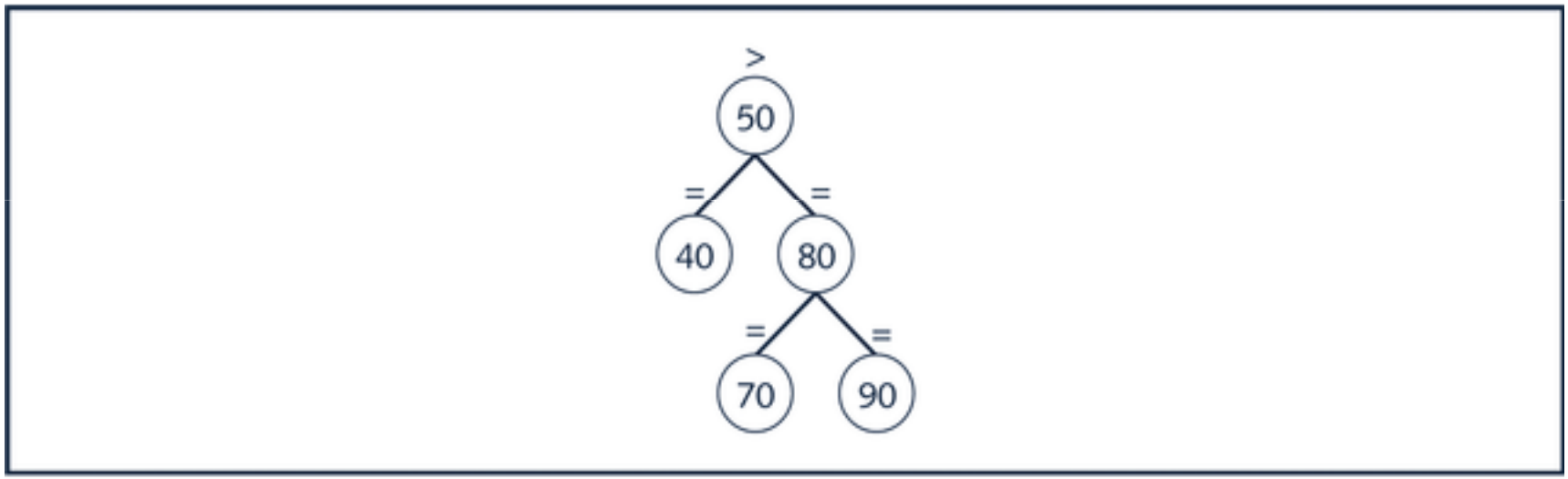


Figure 11-23 AVL tree of Figure 11-21 after inserting 90 and adjusting the balance factors

Insertion Into AVL Trees

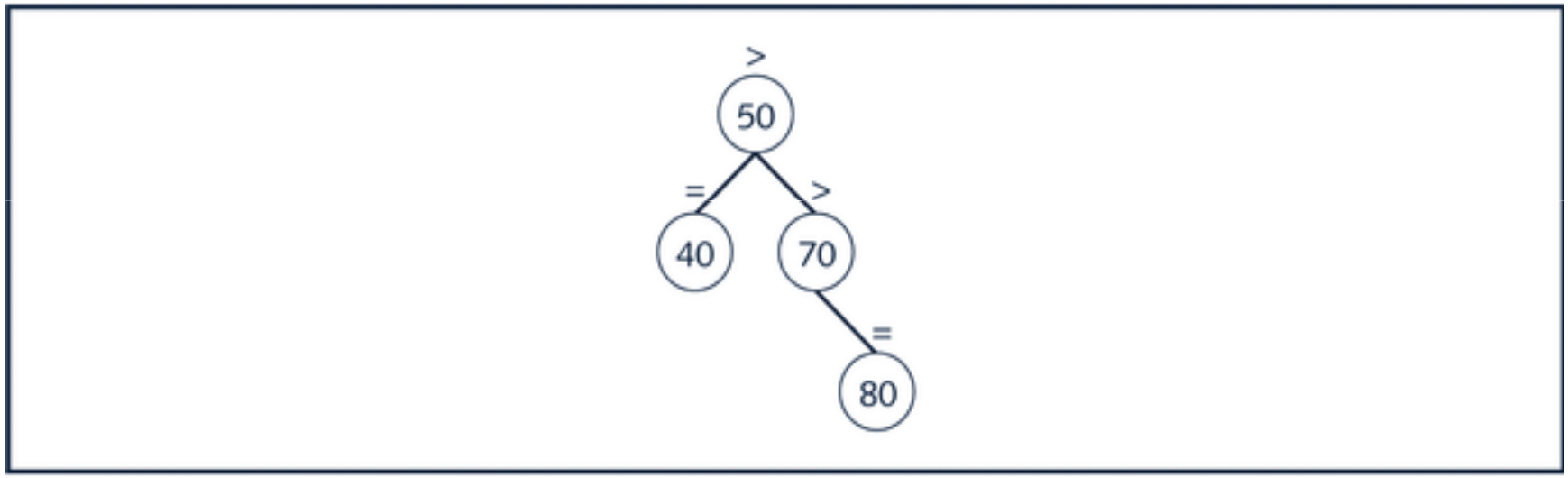


Figure 11-24 AVL tree before inserting 75

Insertion Into AVL Trees

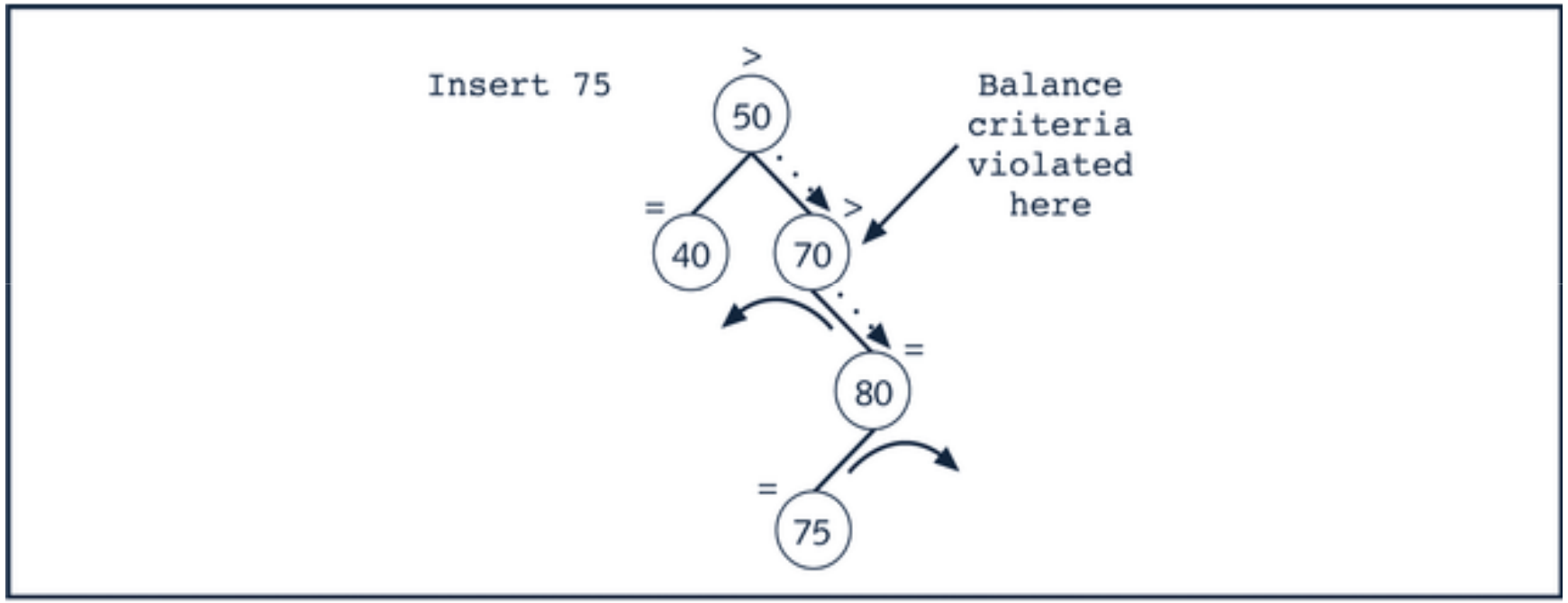


Figure 11-25 Binary tree of Figure 11-24 after inserting 75; nodes other than 75 show their balance factors before insertion

AVL Tree Rotations

- Reconstruction procedure: **rotating** tree
- **left rotation** and **right rotation**
- Suppose that the rotation occurs at node *x*
- Left rotation: certain nodes from the right subtree of *x* move to its left subtree; the root of the right subtree of *x* becomes the new root of the reconstructed subtree
- Right rotation at *x*: certain nodes from the left subtree of *x* move to its right subtree; the root of the left subtree of *x* becomes the new root of the reconstructed subtree

AVL Tree Rotations

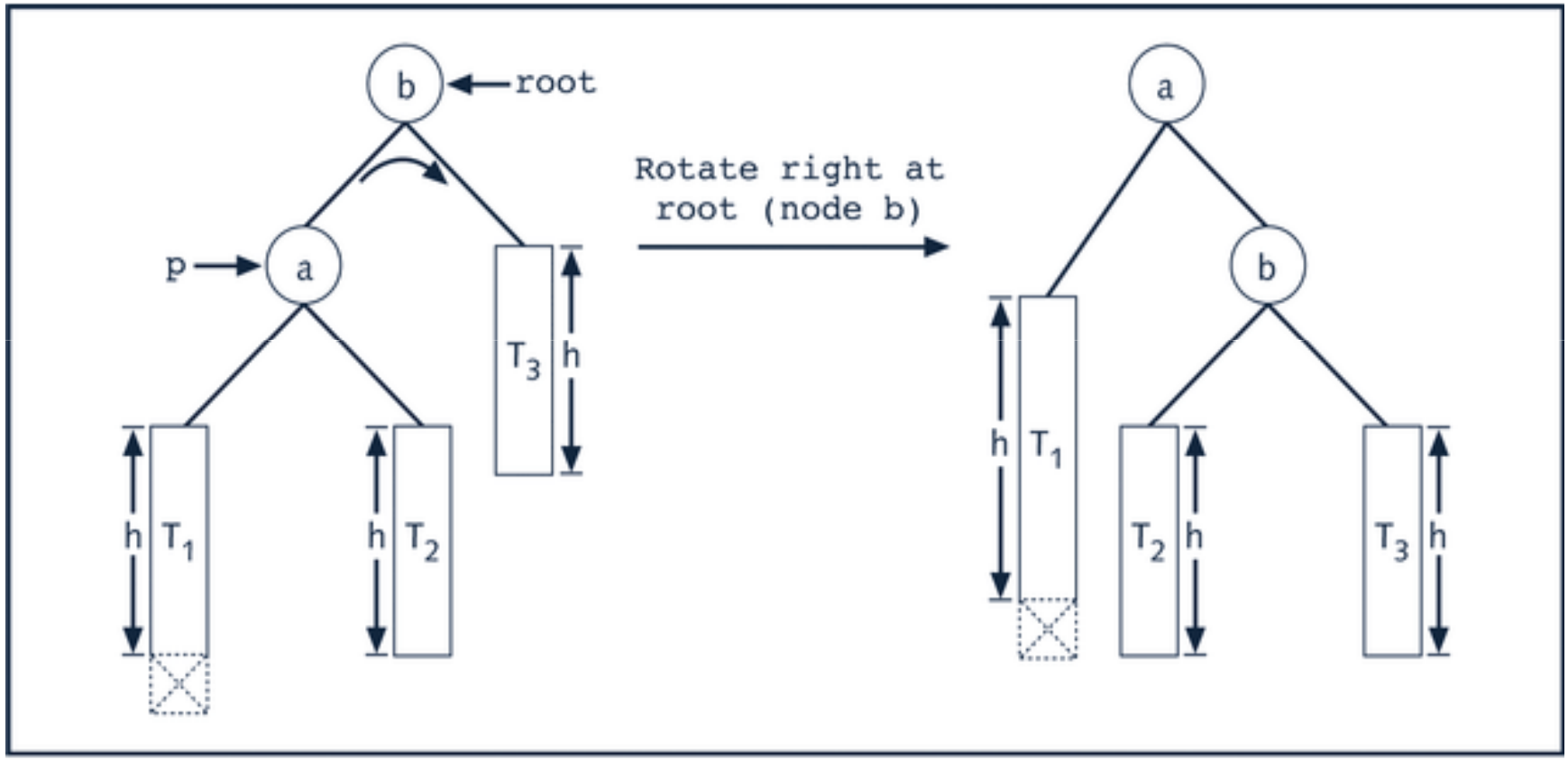


Figure 11-33 Right rotation at *b*

AVL Tree Rotations

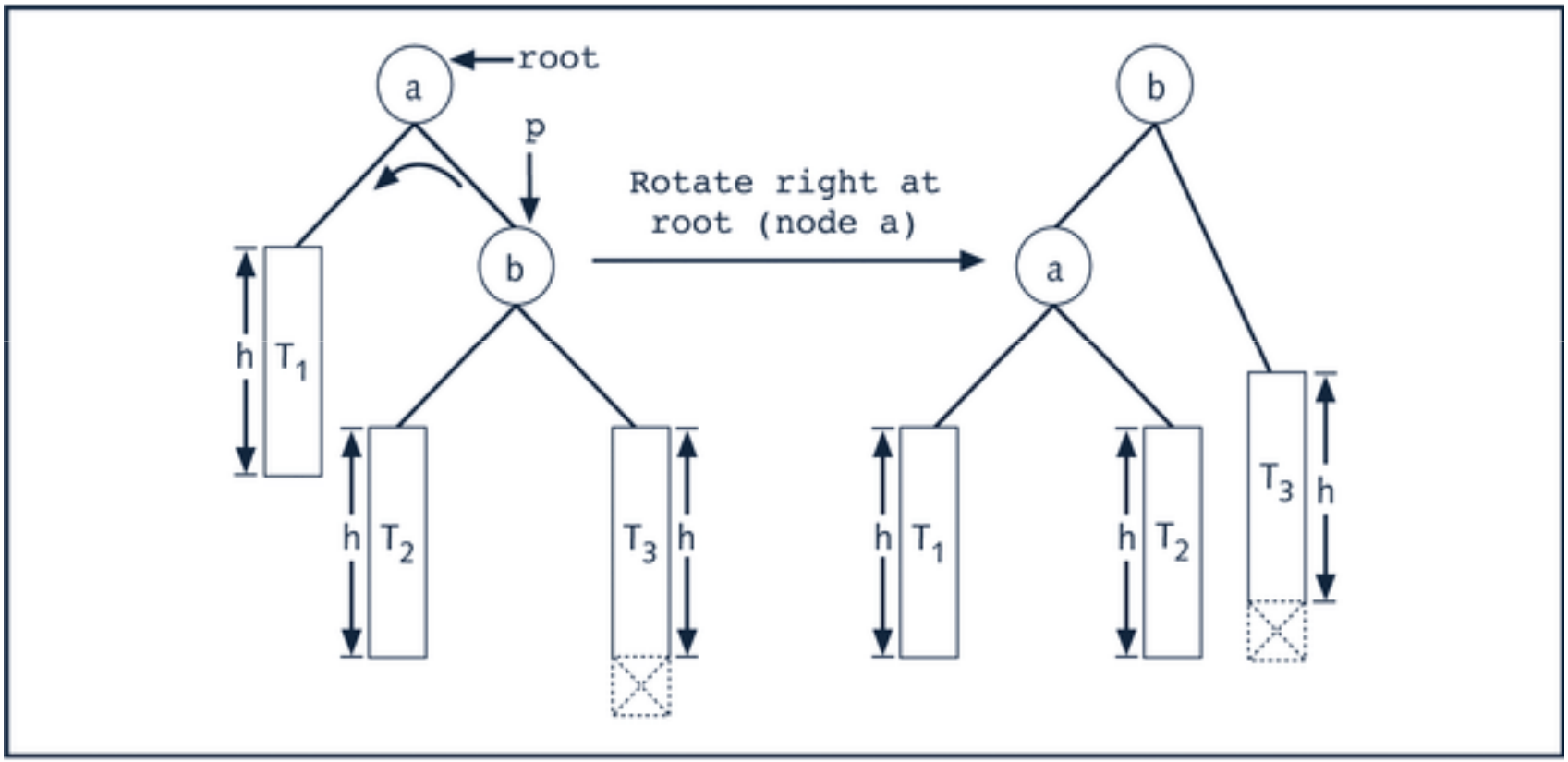


Figure 11-34 Left rotation at *a*

AVL Tree Rotations

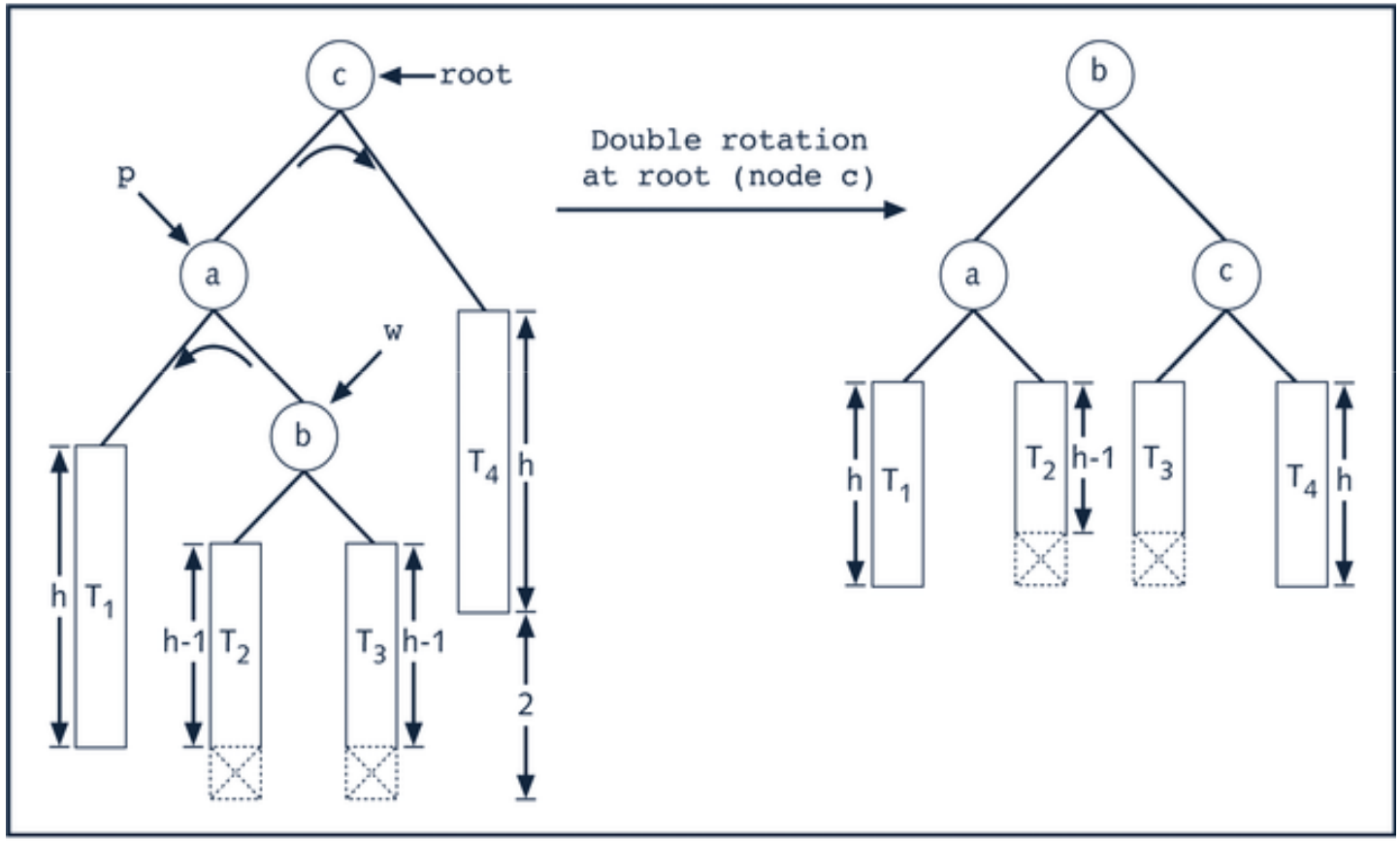


Figure 11-35 Double rotation: first rotate left at *a*, then rotate right at *c*

AVL Tree Rotations

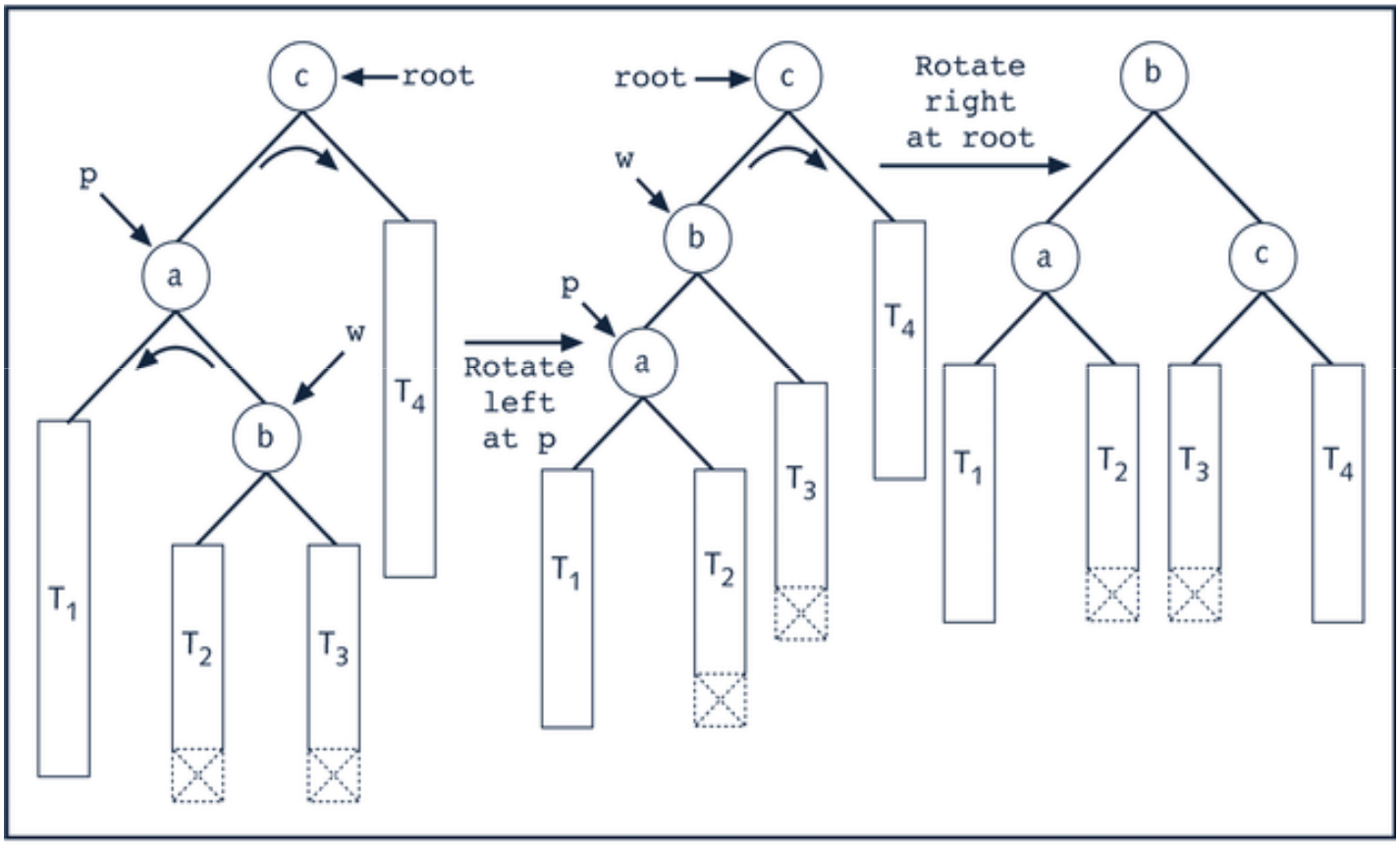


Figure 11-36 Left rotation at *a* followed by a right rotation at *c*

AVL Tree Rotations

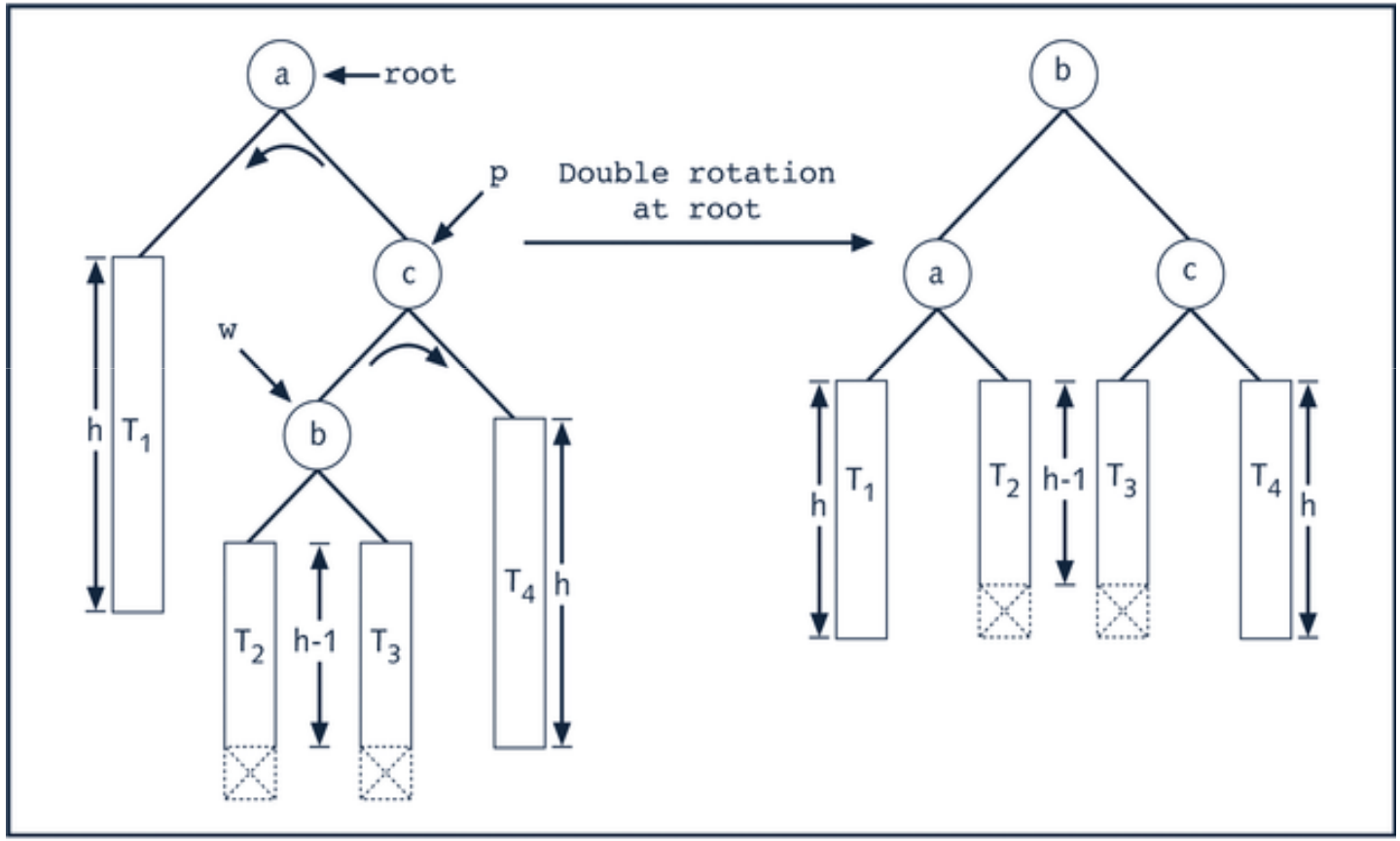


Figure 11-37 Double rotation: first rotate right at *c*, then rotate left at *a*

AVL Tree Rotations

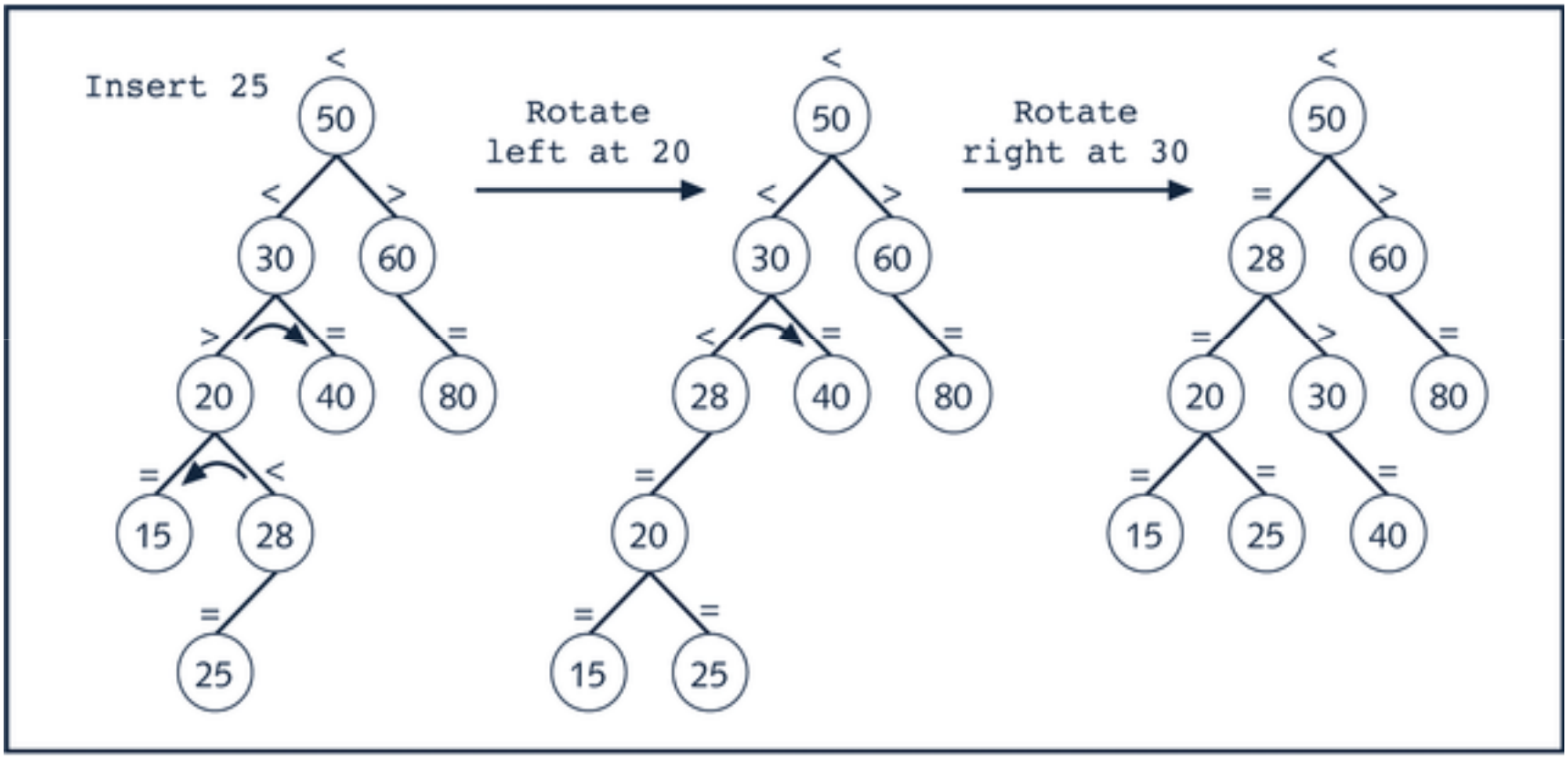


Figure 11-46 AVL tree after inserting 25

Deletion From AVL Trees

- **Case 1:** the node to be deleted is a leaf
- **Case 2:** the node to be deleted has no right child, that is, its right subtree is empty
- **Case 3:** the node to be deleted has no left child, that is, its left subtree is empty
- **Case 4:** the node to be deleted has a left child and a right child

Analysis: AVL Trees

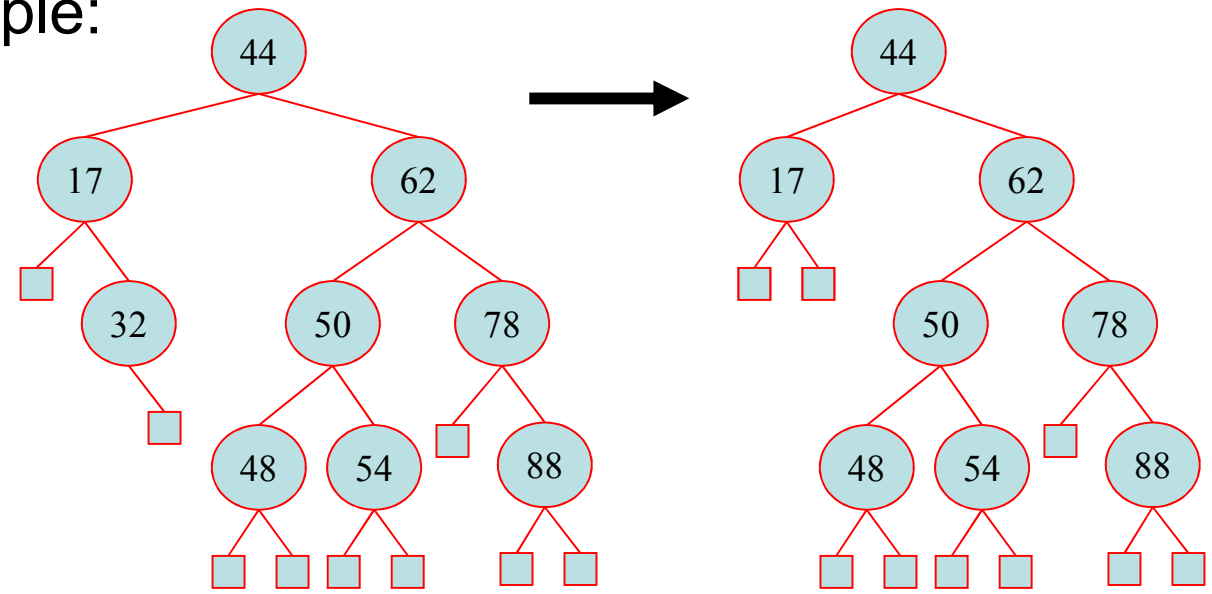
Consider all the possible AVL trees of height h. Let T_h be an AVL tree of height h such that T_h has the fewest number of nodes. Let T_{hl} denote the left subtree of T_h and T_{hr} denote the right subtree of T_h. Then:

$$|T_h| = |T_{hl}| + |T_{hr}| + 1$$

where $|T_h|$ denotes the number of nodes in T_h.

Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



before deletion of 32

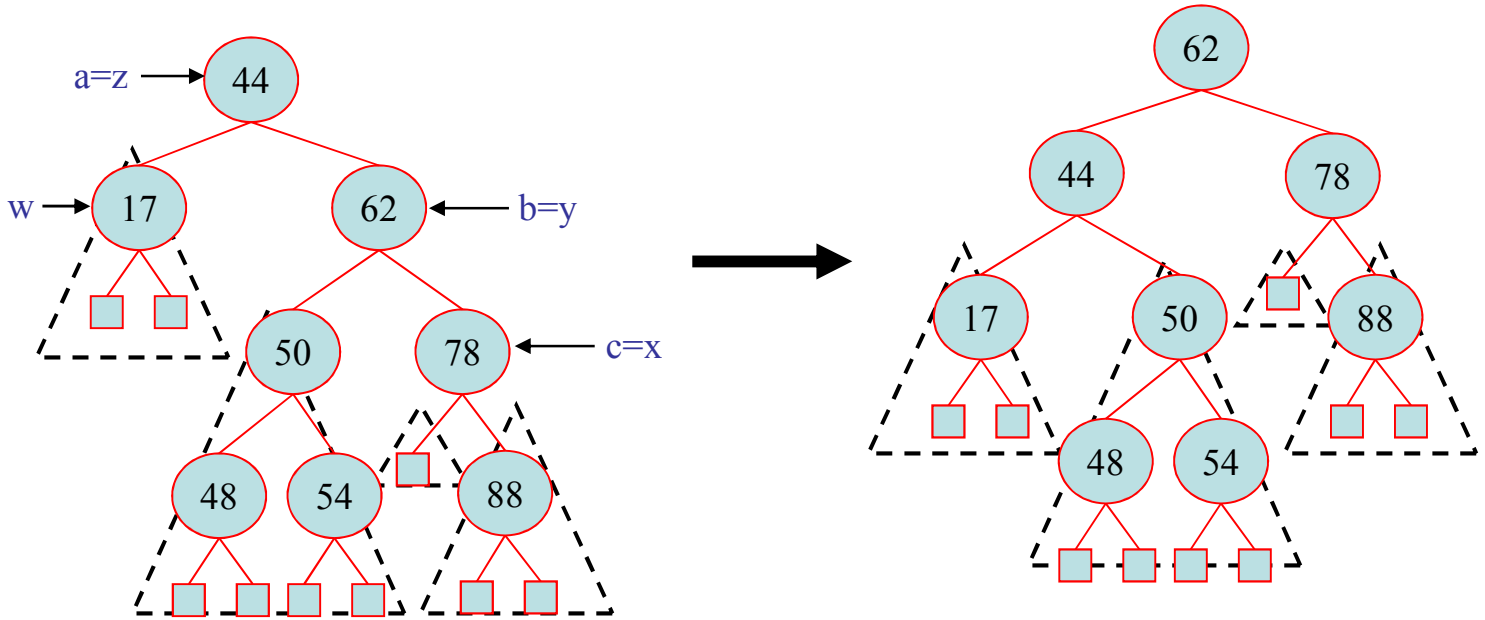
after deletion

Rebalancing after a Removal

- Let *z* be the first unbalanced node encountered while travelling up the tree from w. Also,
- let *y* be the child of z with the larger height,
- let *x* be the child of y defined as follows;
  - If one of the children of y is taller than the other, choose x as the taller child of y.
  - If both children of y have the same height, select x be the child of y on the same side as y (i.e., if y is the left child of z, then x is the left child of y; and if y is the right child of z then x is the right child of y.)

Rebalancing after a Removal

- We perform restructure(x) to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



Running Times for AVL Trees

- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

B-tree

A B-tree of order *m* (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most *m* children.
- Every node (except root) has at least *m*⁄2 children.
- The root has at least two children if it is not a leaf node.
- All leaves appear in the same level, and carry information.
- A non-leaf node with *k* children contains *k*−1 keys.

Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has three child nodes (or subtrees) then it must have two separation values or elements *a*1 and *a*2. All values in the leftmost subtree will be less than *a*1 , all values in the middle subtree will be between *a*1 and *a*2, and all values in the rightmost subtree will be greater than *a*2.

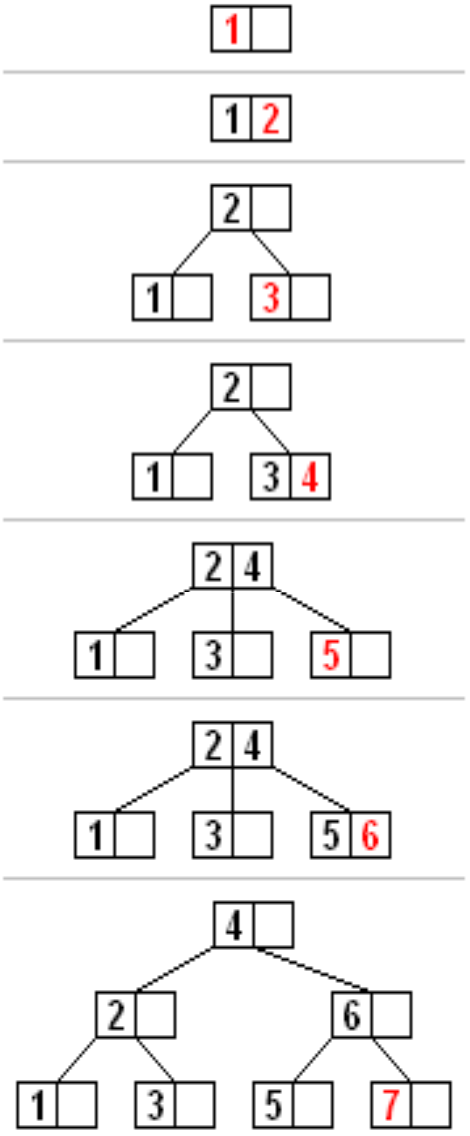
Internal nodes in a B-tree – nodes which are not leaf nodes – are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of *U* children and – other than the root – a **minimum** of *L* children. For all internal nodes other than the root, the number of elements is one less than the number of child pointers; the number of elements is between *L*−1 and *U*−1. The number *U* must be either 2*L* or 2*L*−1; thus each internal node is at least half full. This relationship between *U* and *L* implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there is room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

Best case and worst case heights

- The best case height of a B-Tree is:
  - $\log_m n$.
- The worst case height of a B-Tree is:
  - $\log_{m/2} n$
- where *m* is the maximum number of children a node can have.

Insertion in B-tree

- All insertions start at a leaf node. To insert a new element
- Search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:
- If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
- Otherwise the node is full, so evenly split it into two nodes.
  - A single median is chosen from among the leaf's elements and the new element.
  - Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
  - Insert the separation value in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).



Deletion from B-Tree

- There are two popular strategies for deletion from a B-Tree.

  or
- The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

- the element in an internal node may be a separator for its child nodes
- deleting an element may put its node under the minimum number of elements and children.
- Each of these cases will be dealt with in order.

Deletion from a leaf node

- Search for the value to delete.
- If the value is in a leaf node, it can simply be deleted from the node,
- If underflow happens, check siblings to either transfer a key or fuse the siblings together.
- if deletion happened from right child retrieve the max value of left child if there is no underflow in left child
- in vice-versa situation retrieve the min element from right

Deletion from B-Tree

- **Deletion from an internal node**
- Each element in an internal node acts as a separation value for two subtrees, and when such an element is deleted, two cases arise. In the first case, both of the two child nodes to the left and right of the deleted element have the minimum number of elements, namely $L-1$. They can then be joined into a single node with $2L-2$ elements, a number which does not exceed $U-1$ and so is a legal node. Unless it is known that this particular B-tree does not contain duplicate data, we must then also (recursively) delete the element in question from the new node.

- In the second case, one of the two child nodes contains more than the minimum number of elements. Then a new separator for those subtrees must be found. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is the smallest element which is still greater than the separator. Both of those elements are in leaf nodes, and either can be the new separator for the two subtrees.

  - If the value is in an internal node, choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
  - This has deleted an element from a leaf node, and so is now equivalent to the previous case

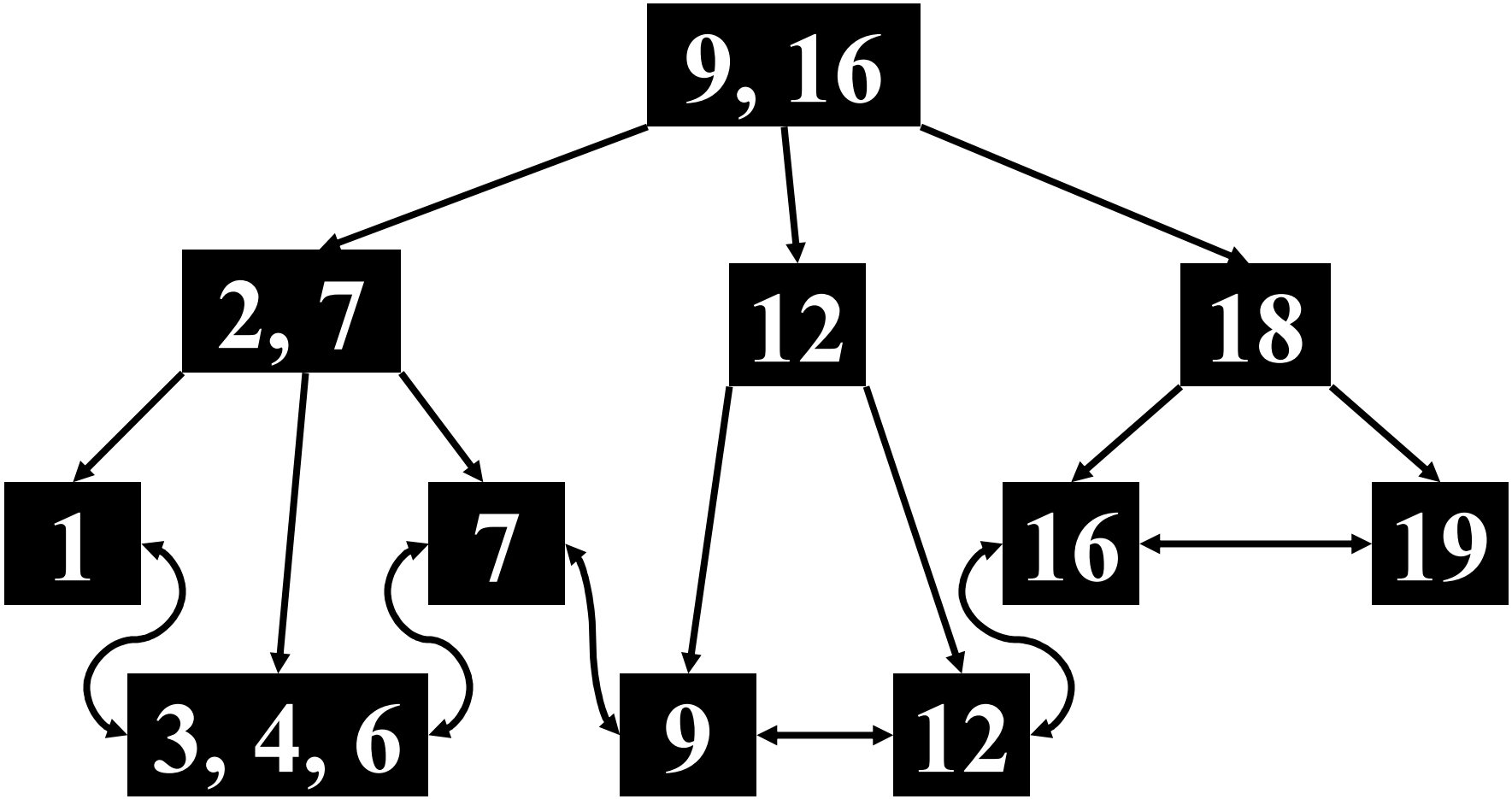
Difference between B-tree and B+ tree

- In a B- tree you can store both keys and data in the internal/leaf nodes. But in a B+ tree you have to store the data in the leaf nodes only.

B+ Trees

- Similar to B trees, with a few slight differences
- All data is stored at the leaf nodes (*leaf pages*); all other nodes (*index pages*) only store keys
- Leaf pages are linked to each other
- Keys may be duplicated; every key to the right of a particular key is >= to that key

B+ Tree Example

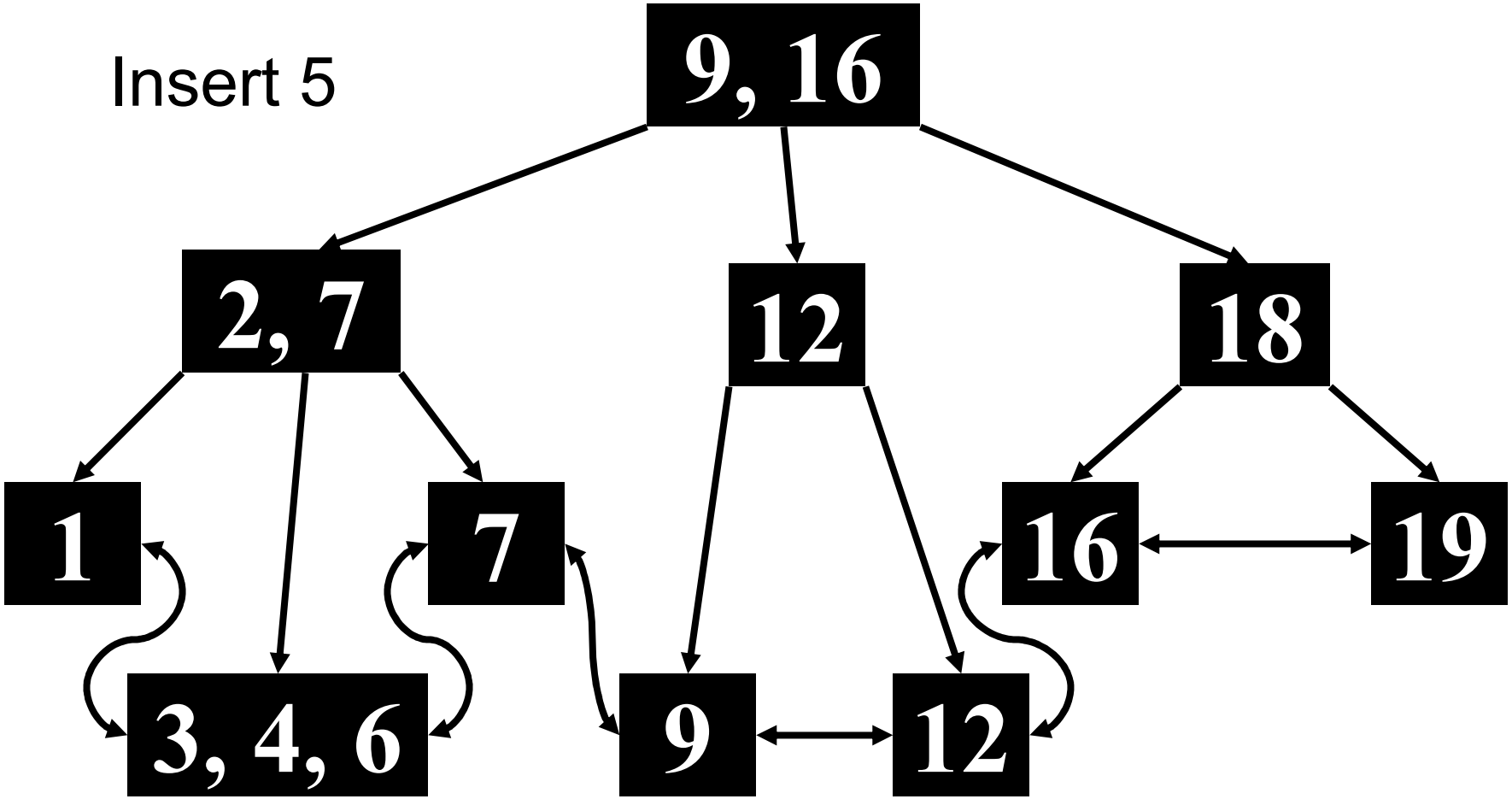


B+ Tree Insertion

- Insert at bottom level
- If leaf page overflows, split page and copy middle element to next index page
- If index page overflows, split page and move middle element to next index page

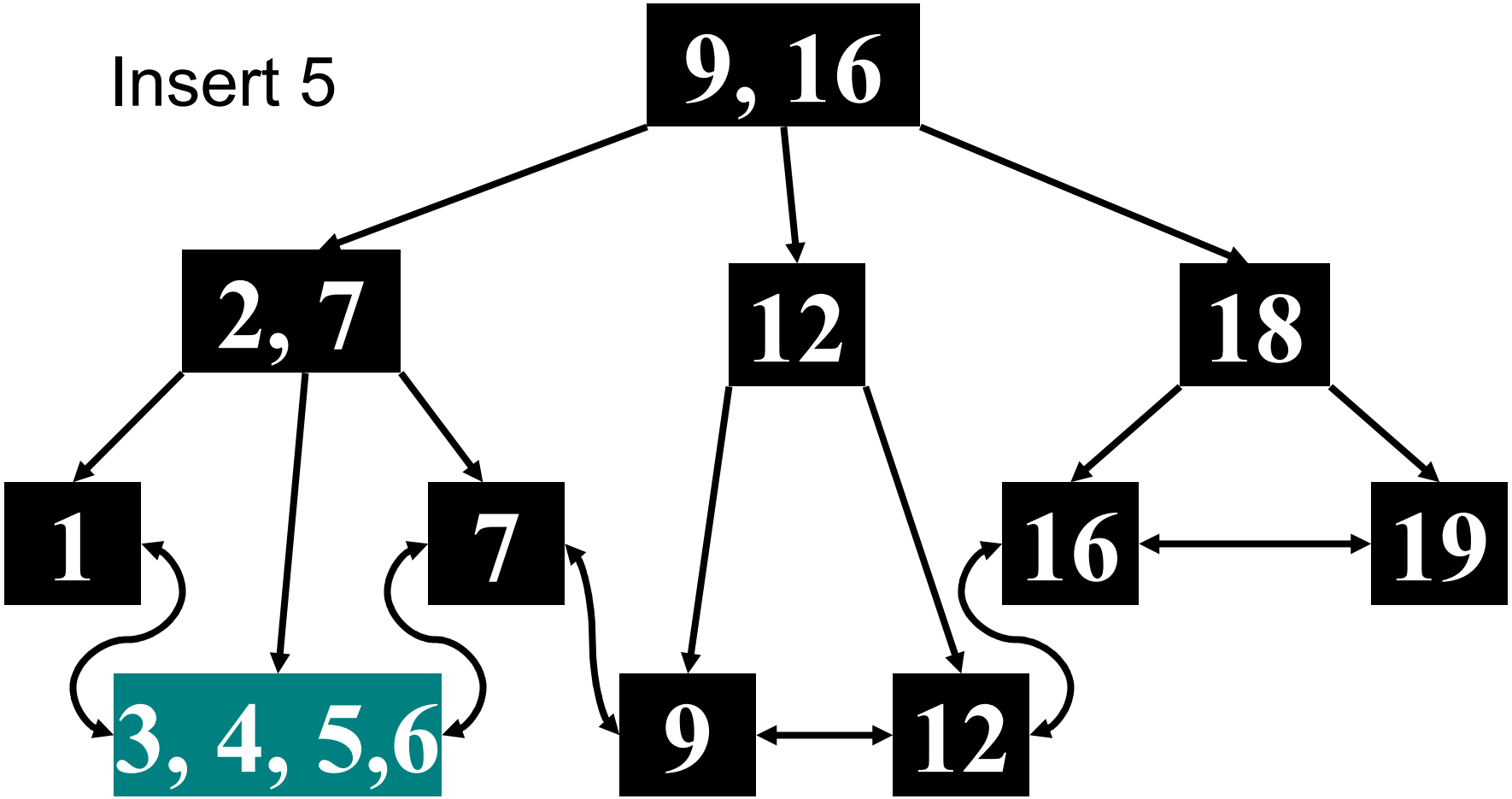
B+ Tree Insertion Example

Insert 5

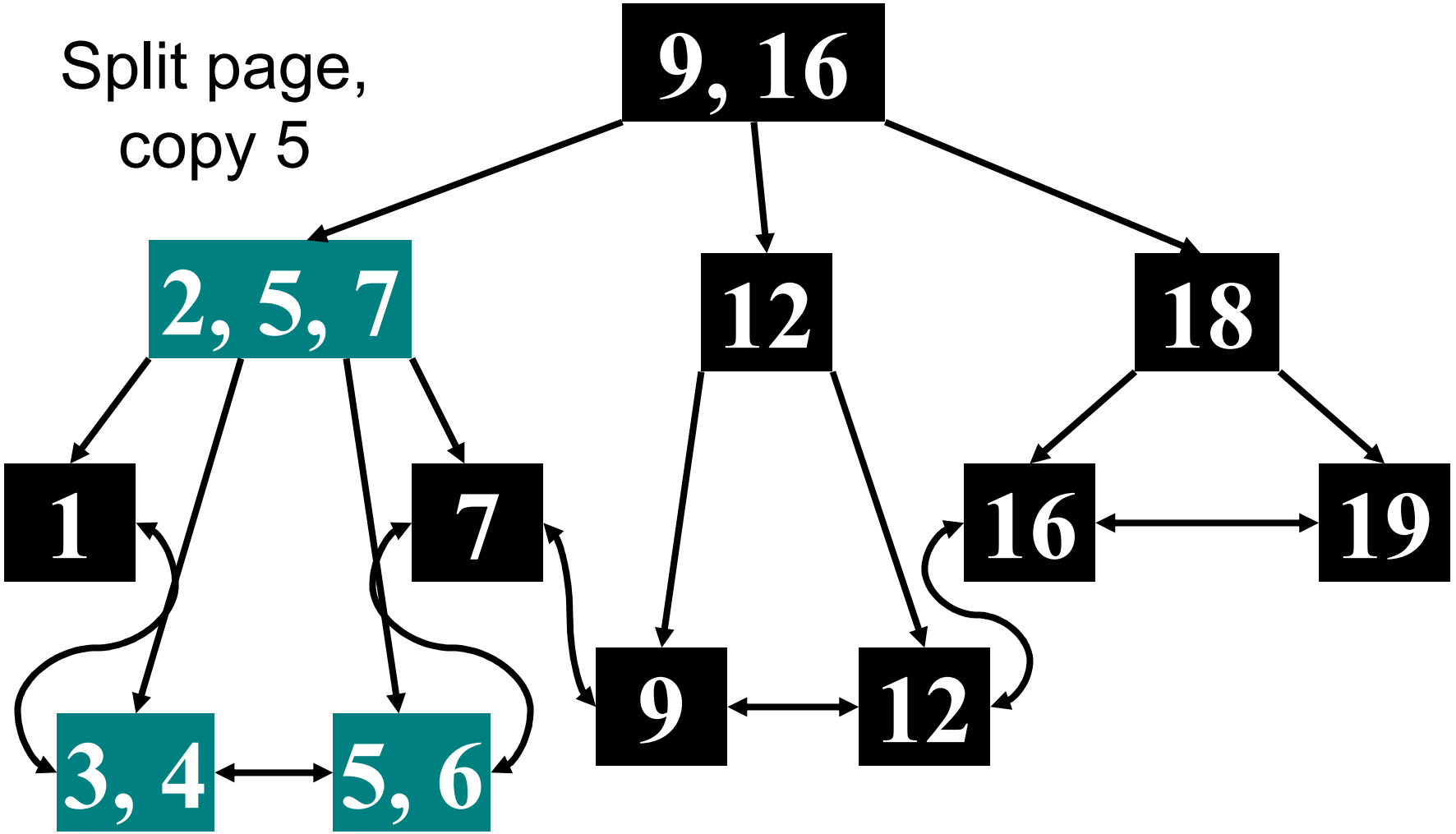


B+ Tree Insertion Example

Insert 5

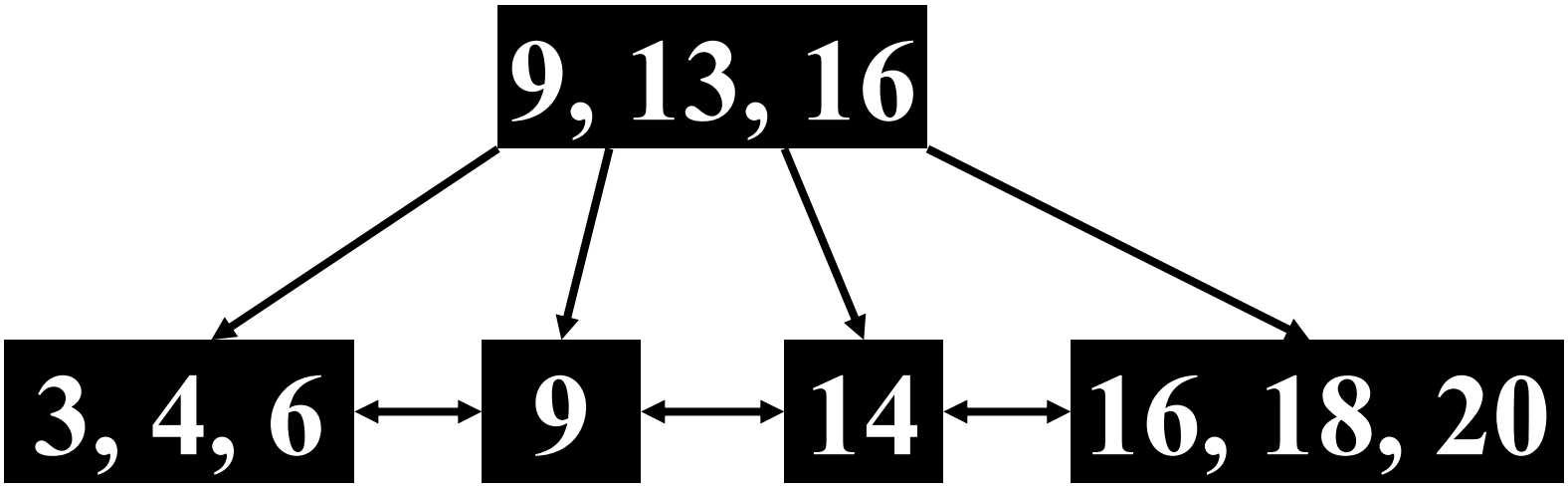


B+ Tree Insertion Example



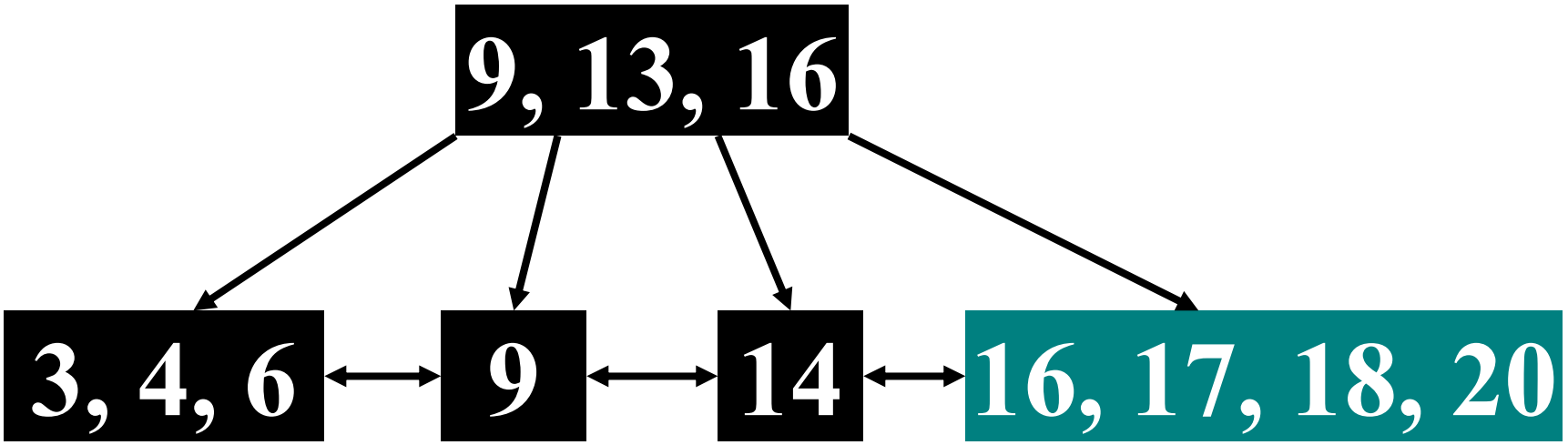
B+ Tree Insertion Example 2

Insert 17



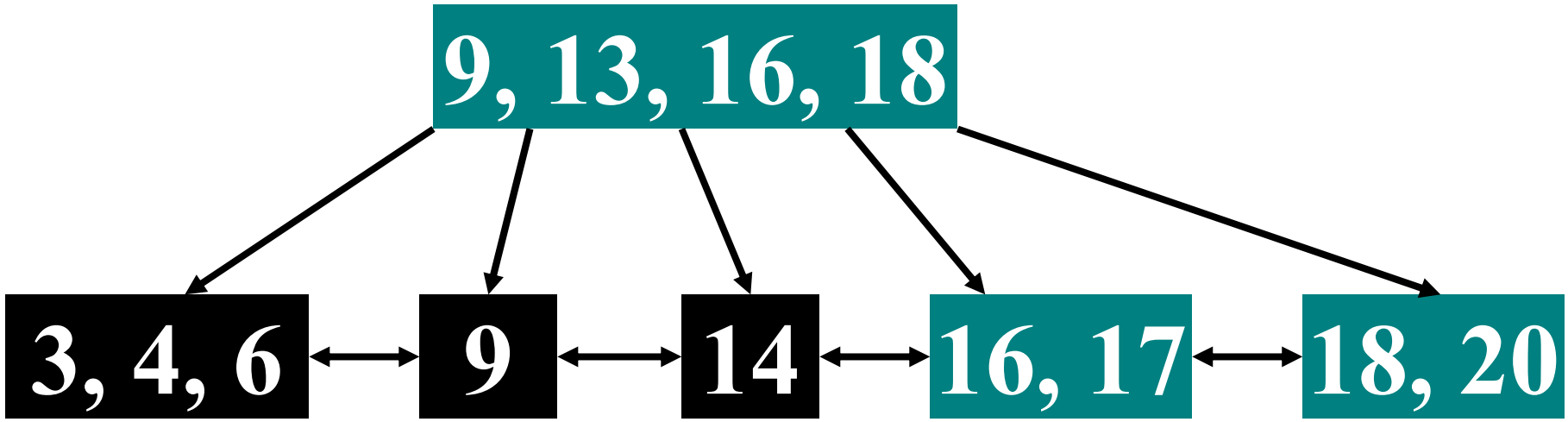
B+ Tree Insertion Example 2

Insert 17

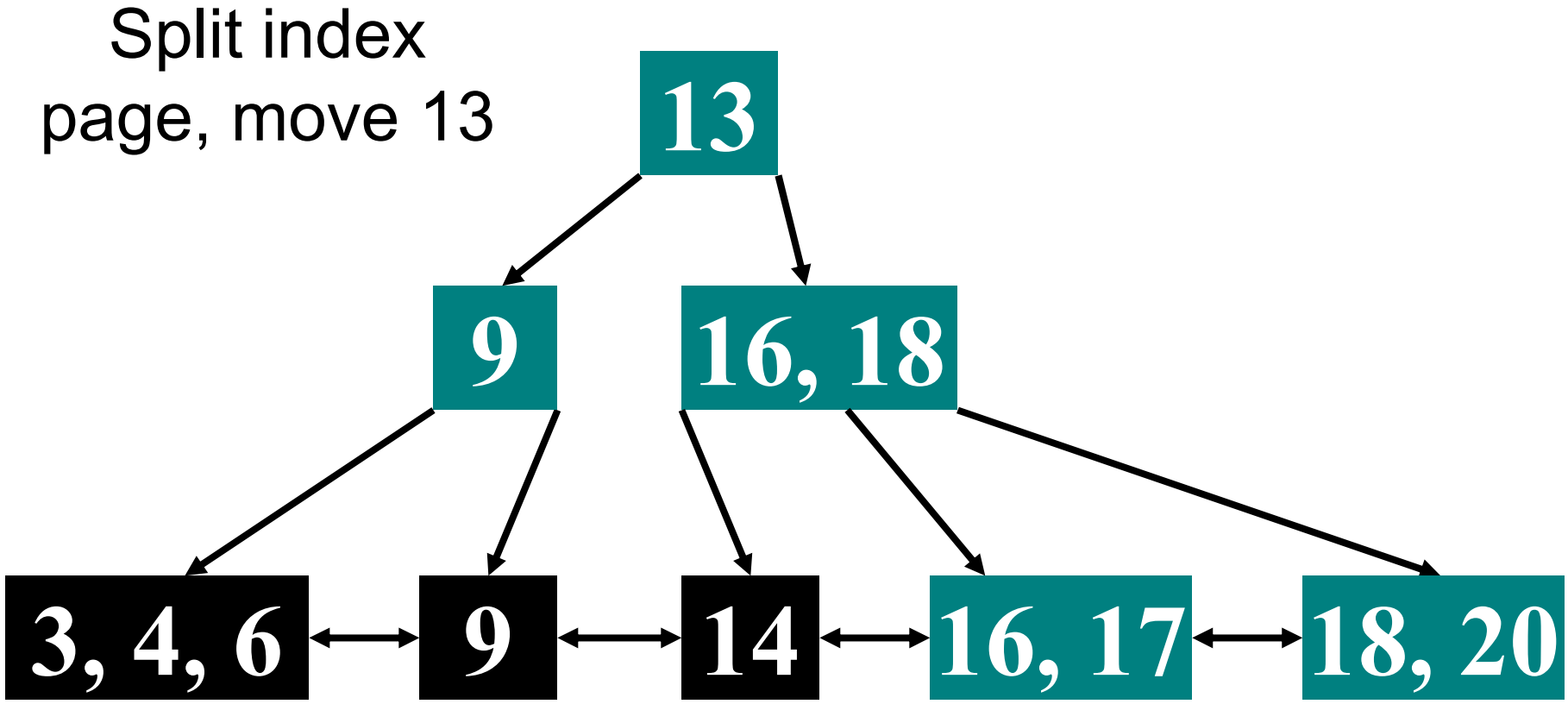


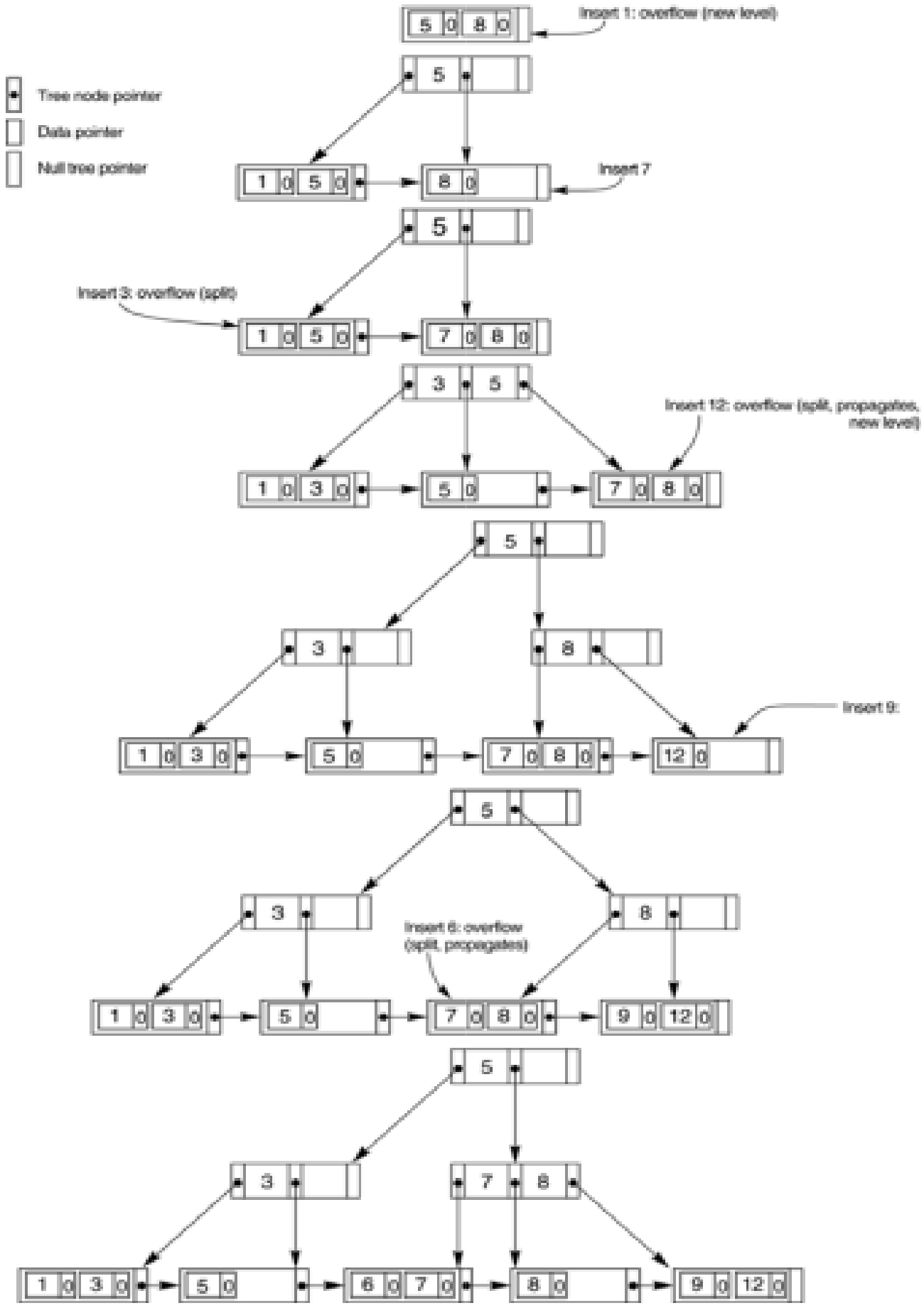
B+ Tree Insertion Example 2

Split leaf page, copy 18

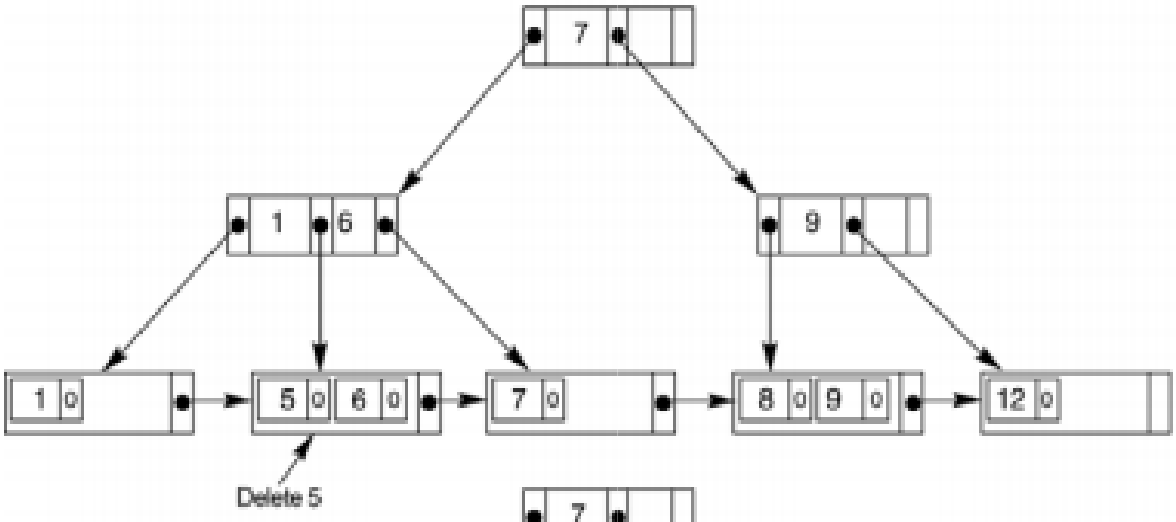


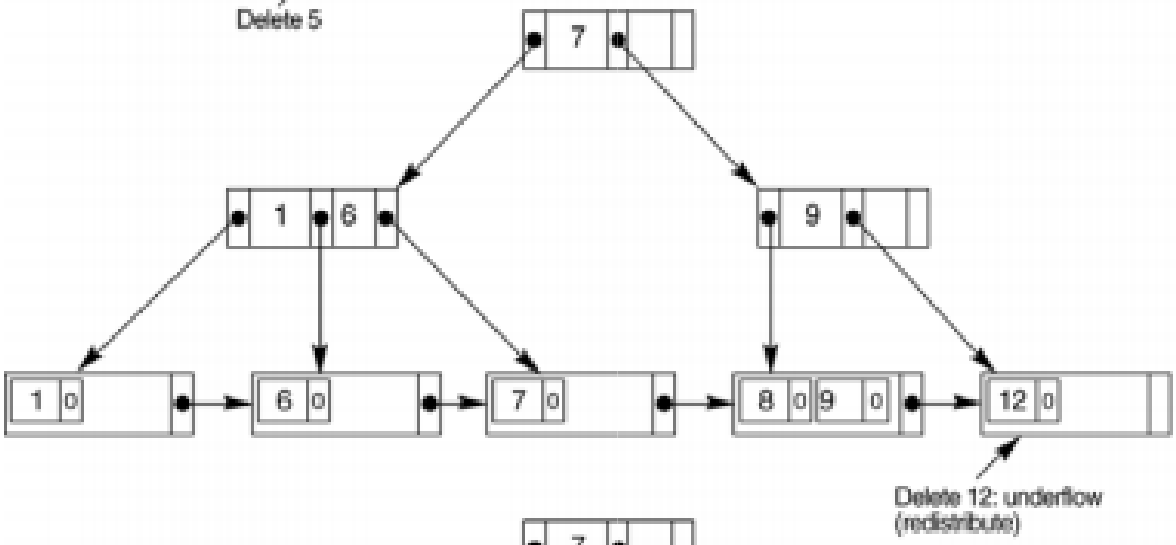
B+ Tree Insertion Example 2

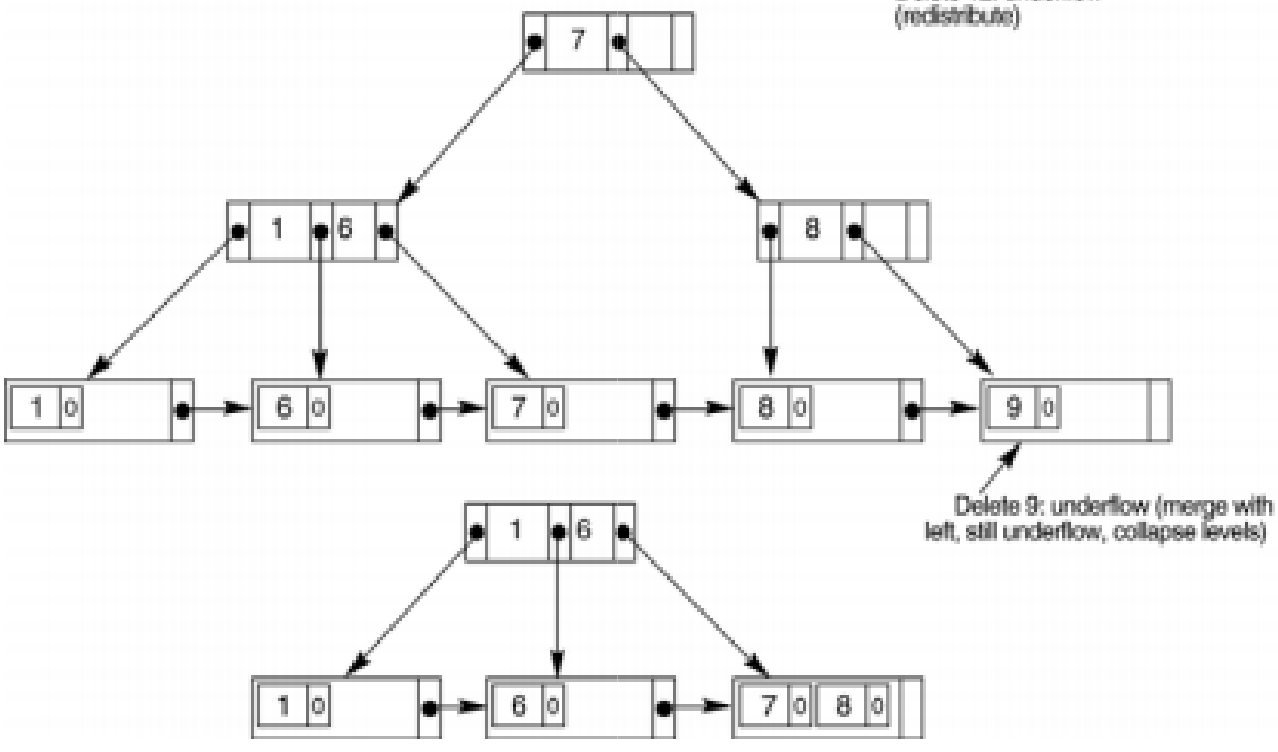



INSERTION SEQUENCE: 8, 5, 1, 7, 3, 12, 9, 6
Insert 1: overflow (new level)
Tree node pointer
Data pointer
Null tree pointer
Insert 7
Insert 3: overflow (split)
Insert 12: overflow (split, propagates, new level)
Insert 9:
Insert 6: overflow (split, propagates)

DELETION SEQUENCE: 5, 12, 9





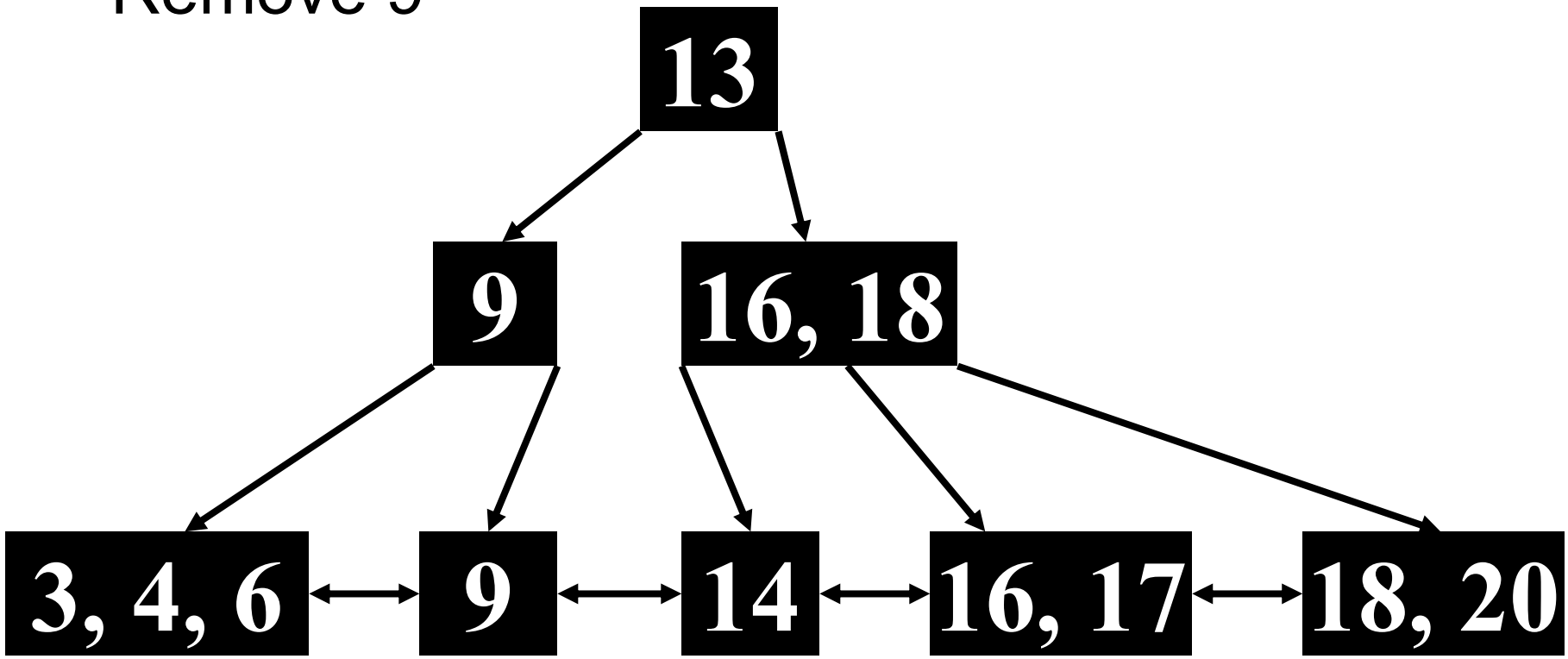


B+ Tree Deletion

- Delete key and data from leaf page
- If leaf page underflows, merge with sibling and delete key in between them
- If index page underflows, merge with sibling and move down key in between them

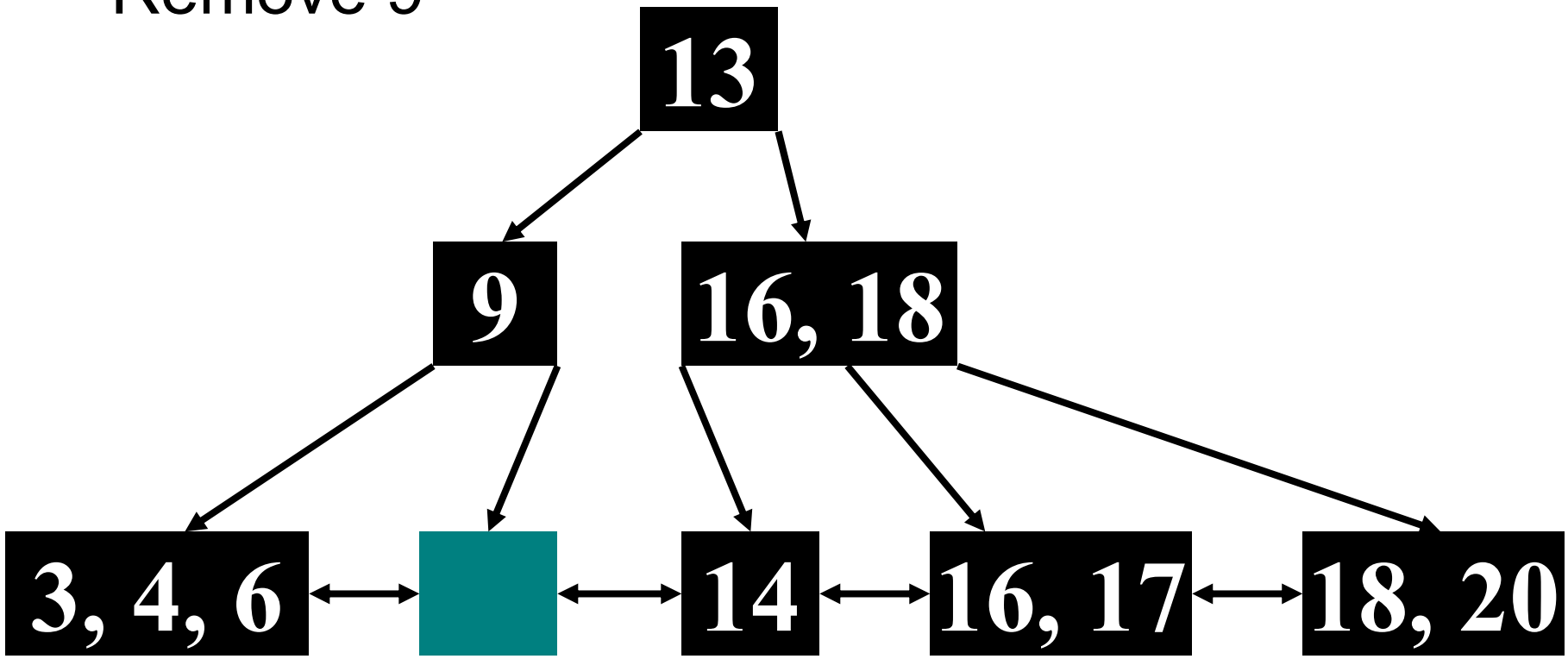
B+ Tree Deletion Example

Remove 9



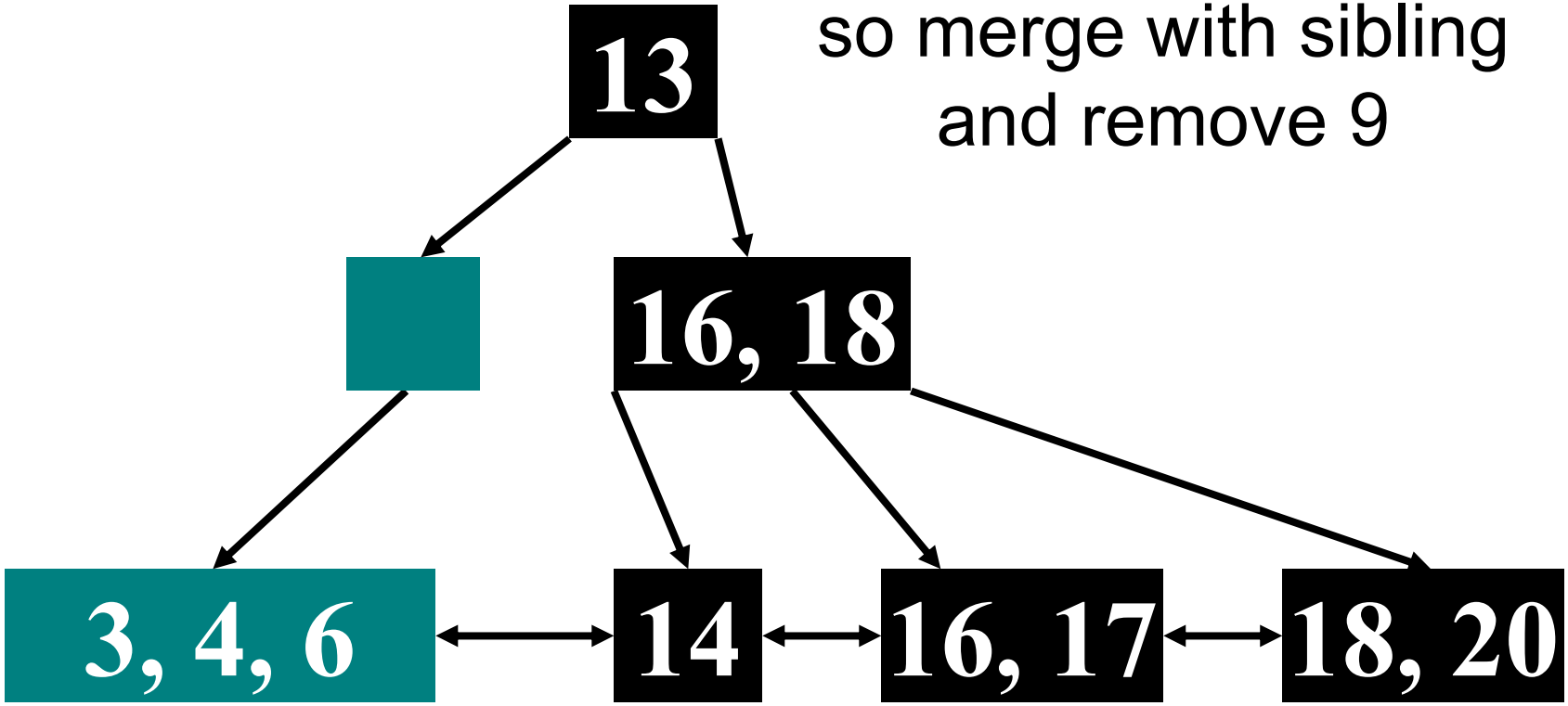
B+ Tree Deletion Example

Remove 9



B+ Tree Deletion Example

Leaf page underflow, so merge with sibling and remove 9



B+ Tree Deletion Example

Index page underflow,
so merge with sibling
and demote 13