Chapter 5
Trees

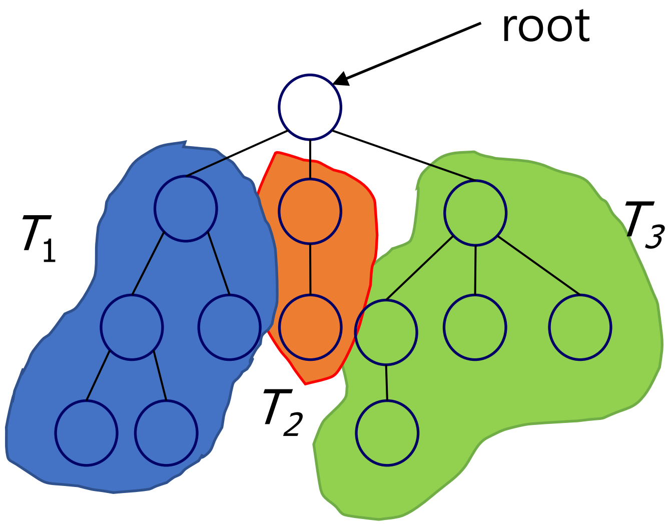
2024 Spring

Ri Yu
Ajou University

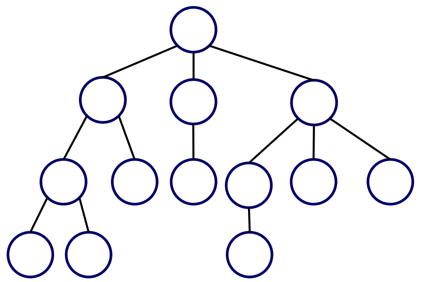
Contents

Trees

❖ Definition: A *tree* is a finite set of one or more nodes such that:
- There is a specially designated node called *root*
- The remaining nodes are partitioned into $n \geqq 0$ disjoint set $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called the *subtrees* of the root



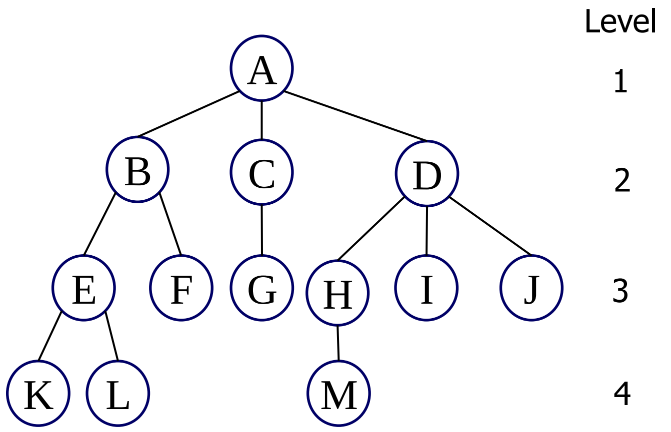
Terminology



- Node: the item of information plus the branches to other nodes
- Degree of a node: the number of subtrees of a node
- Degree of a tree: the maximum of the degree of the nodes in the tree
- Terminal node (or leaf): node with degree zero
- A node that has subtrees is the parent of the roots of the subtrees, and the roots of the subtrees are the children of the node
- Siblings: children of the same parent
- Ancestors of a node: all the nodes along the path from the root to that node
- Descendants of a node: all the nodes in its subtrees
- Level of a node: the level of the node's parent plus one (the root is at level 1)
- Height (or depth): the maximum level of any node in the tree

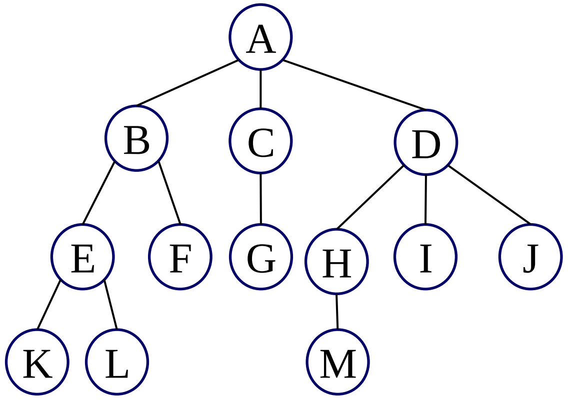
Example

- Root node of the tree: A
- Degree of A: 3
- Leaf or terminal node: {K, L, F, G, M, I, J}
- Parent of D: A
- Children of D: H, I, J
- Siblings of H: I, J
- The depth(height) of the tree is 4
- The degree of the tree is 3
- The ancestors of node M: A, D, H
- The descendants of node D: H, I, J, M

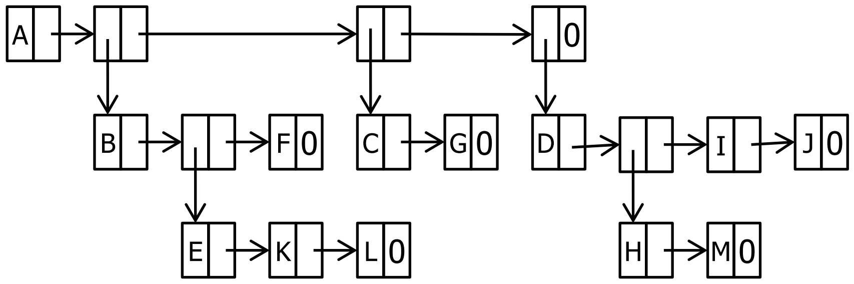


Representation of Trees

- List representation
  - Write the tree as a list in which each of the subtrees is also a list
  - The information in the root node comes first, followed by a list of the subtrees of that node



(A (B (E (K, L), F), C (G), D (H (M), I, J)))



List representation of the tree
(*tag* fields not shown)

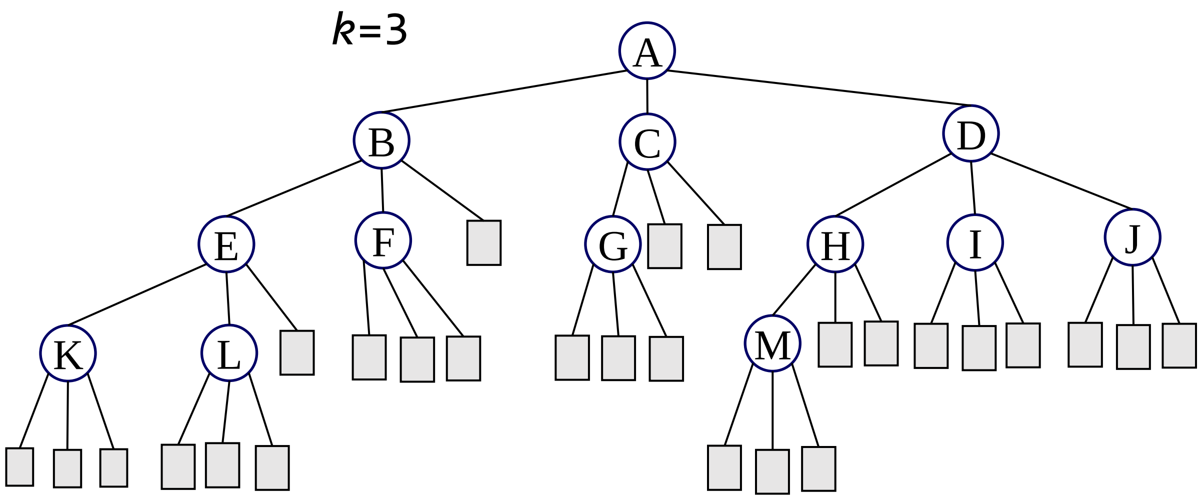
Representation of Trees

❖ K-ary Tree Representation

DATA	CHILD1	...	CHILD k

Possible node structure for a tree of degree k

Lemma 5.1 If T is k-ary tree with n nodes, each having a fixed size, then $n \cdot (k-1) + 1$ of the $n \cdot k$ child fields are 0, $n \geq 1$



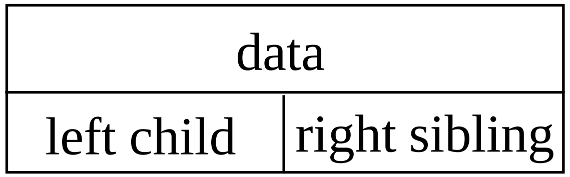
# of non-empty child field = n -1

# of child fields = $n \cdot k$

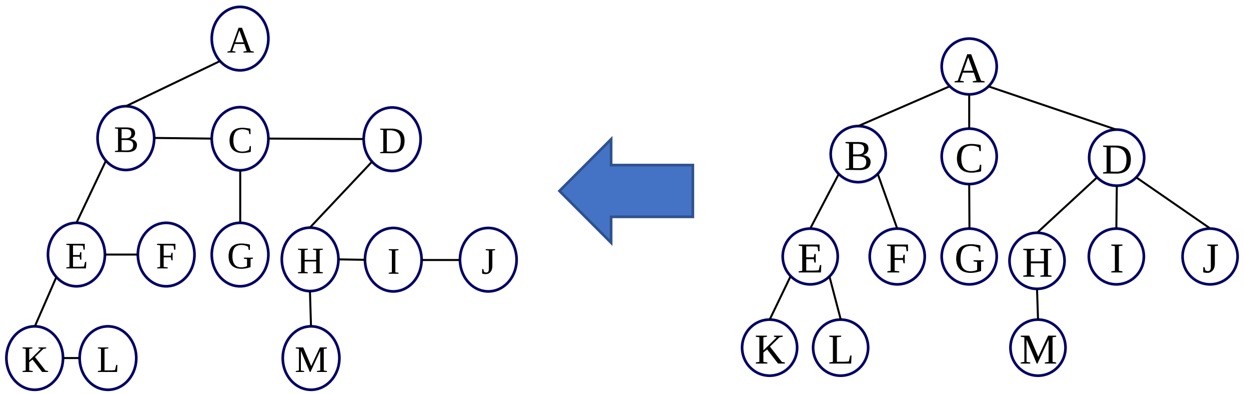
# of empty child fields
= $n \cdot k - (n - 1)$
= $n \cdot (k-1) + 1$

Representation of Trees

- Left child-right sibling representation
  - It is easier to work with nodes of a fixed size
  - Every node has at most one leftmost child and one closest right sibling
    - The leftmost child of A: B
    - The closest right sibling of B: C



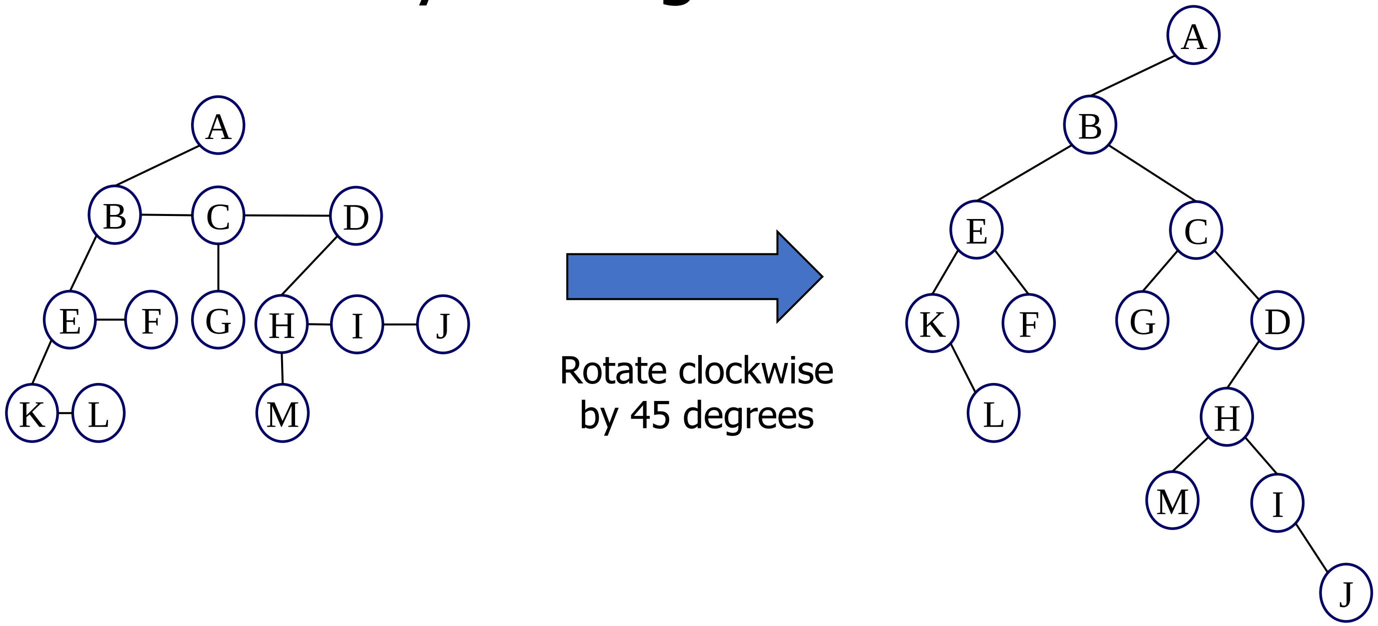
Left child-right sibling node structure



- Since the order of children in a tree is not important, any of the children of a node could be the leftmost child, and any of its siblings could be the closest right sibling

Representation as a Degree-Two Tree

❖ Simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees



Left child-right sibling tree

Left child-right child tree = **binary tree**

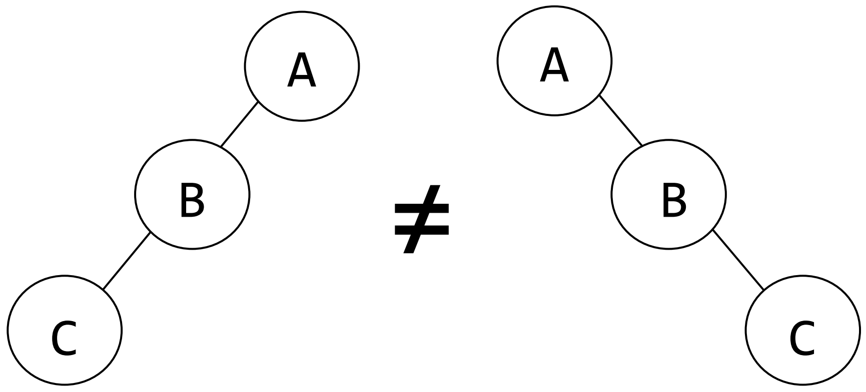
Binary Trees

- ❖ Definition (recursive)
  - ➢ 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
- ❖ The chief characteristic of a binary tree is the stipulation that the degree of any given node must not exceed two
- ❖ The left subtree and the right subtree are distinguished
- ❖ Any tree can be transformed into binary tree by left child-right sibling representation

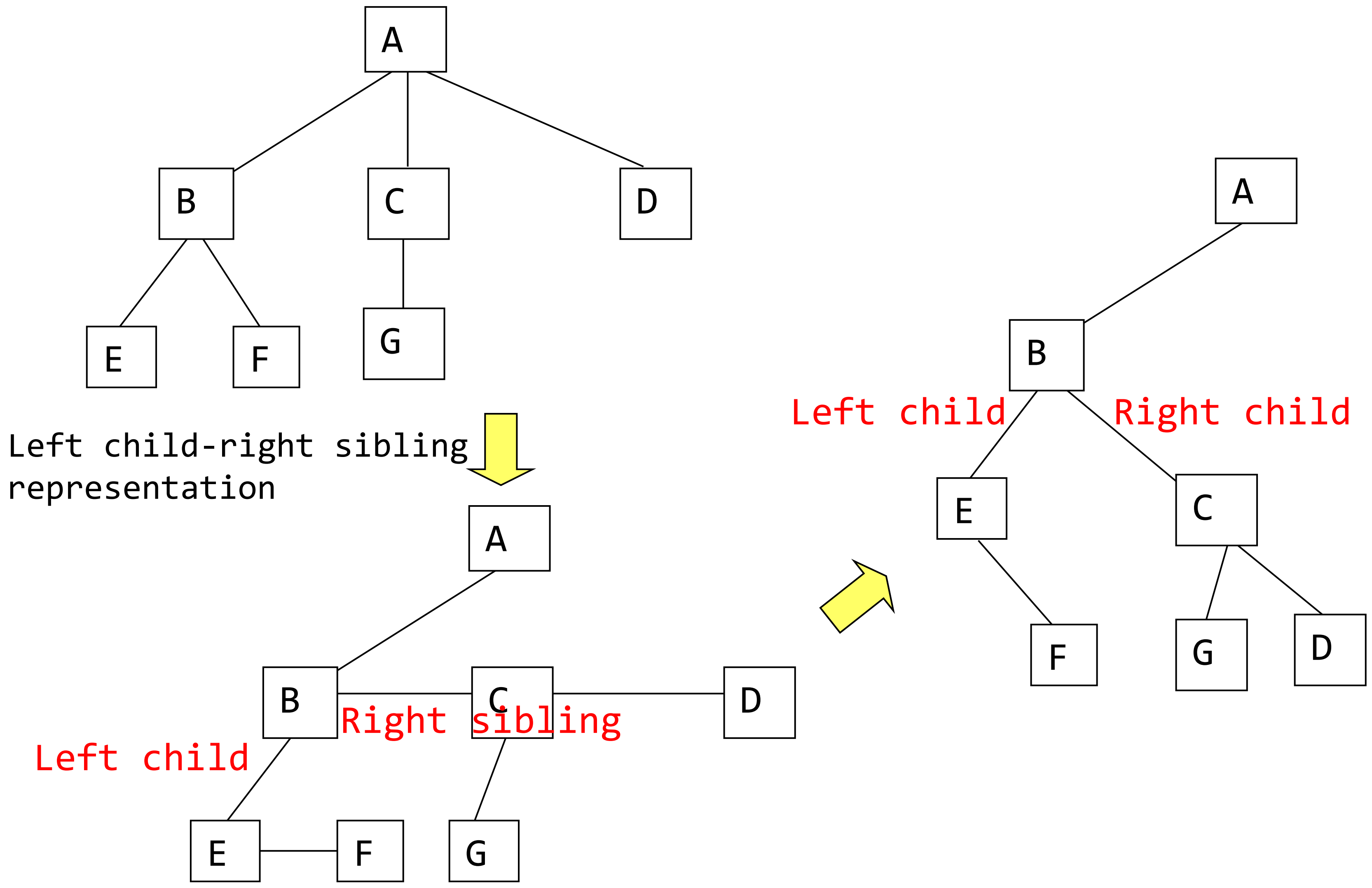
Binary Trees

❖ Differences from Tree

- Minimum number of nodes : 1 vs. 0
- Significance of the order of the children : No vs. Yes



Binary Trees



Abstract Data Type Binary_Tree

ADT Binary_Tree(abbreviated BinTree) is

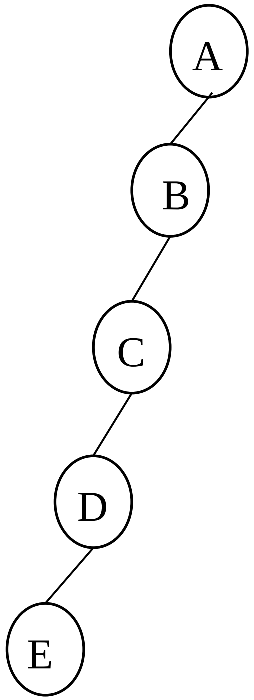
objects: A finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

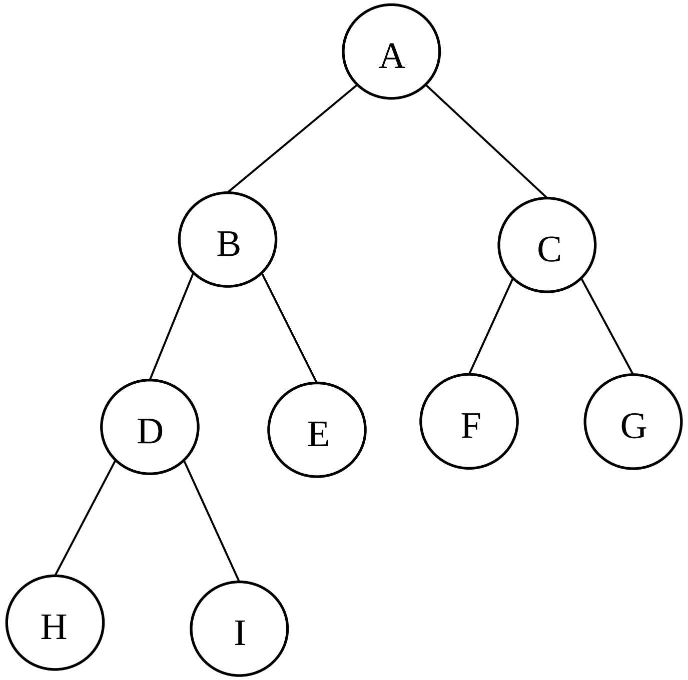
for all *bt*, *bt*1, *bt*2 $\in$ *BinTree*, *item* $\in$ *element*

BinTree Create()	::= creates an empty binary tree
Boolean IsEmpty(*bt*)	::= **if** (*bt*==empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::= **return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, 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*

Two Special Kinds of Binary Trees



Skewed Binary Trees



Complete Binary Tree

Properties of Binary Trees

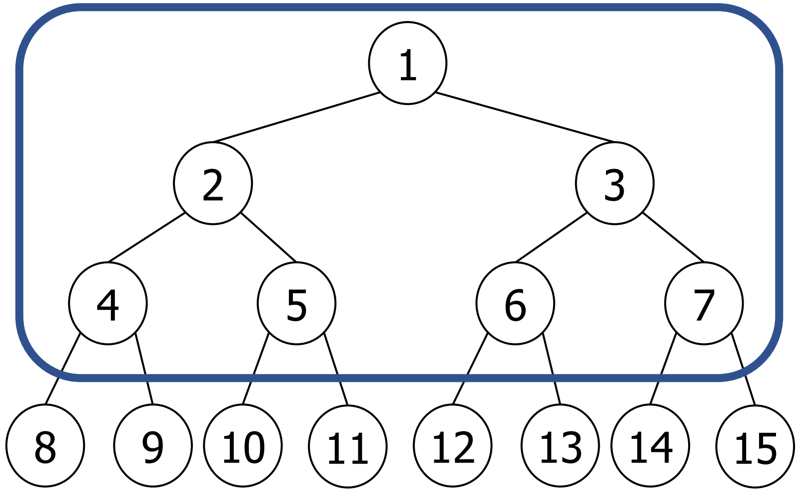
- Lemma 5.2 [Maximum number of nodes in a binary tree]
  - The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1
  - The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1
  - Proof (귀납법)
    - Induction base (귀납 기초)
      - The root is the only node on level i=1. Hence, the maximum number of nodes on level i=1 is $2^{i-1} = 2^0 = 1$
    - Induction hypothesis (귀납 가설)
      - Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i-1 is 2^{i-2}
    - Induction step (귀납 과정)
      - The maximum number of nodes on level i-1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}

$$\sum_{i=1}^{k}(\text{maximum number of nodes on level } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

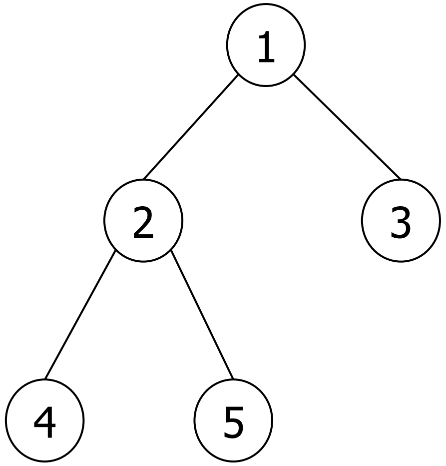
Full & Complete Binary Tree

❖ Definition: A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geqq 0$

❖ Definition: 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



Full binary tree of depth 4 with sequential node numbers



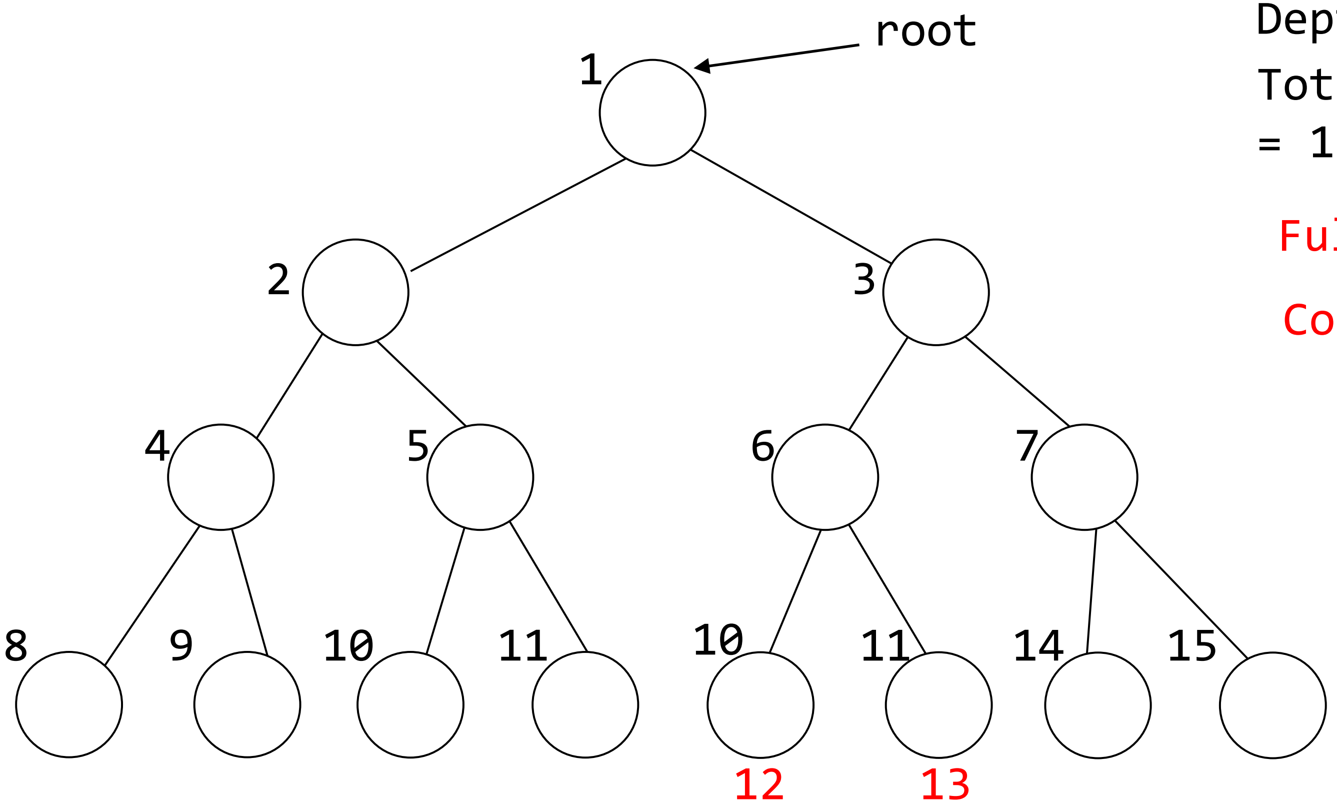
Complete binary tree

The height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$ where $\lceil x \rceil$ is the smallest integer $\geq x$

By Lemma 5.2,

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

Example



Depth = 4

Total number of nodes

= 15 = 2^4 -1

Full binary tree ?

Complete binary tree ?

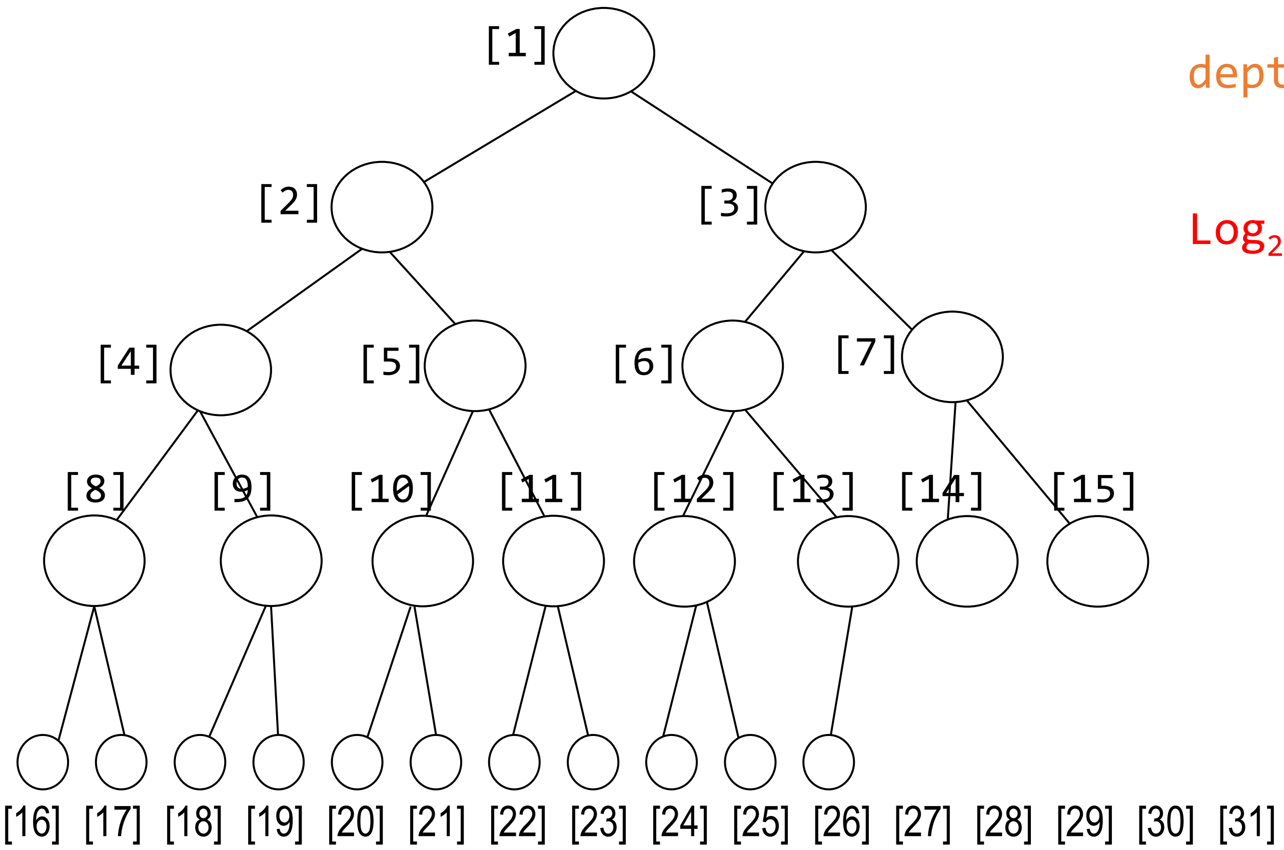
Array Representation

❖ If a complete tree with n nodes is represented sequentially, then for any node with index i, $1 \leqq i \leqq n$, we have:

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

Example

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



n = 26

depth = 5

$= \lceil log_2(26+1) \rceil$

$\mathrm{Log}_2 2^{5-1} < \log_2 27 < \log_2 2^5$

Parent of 5 ?

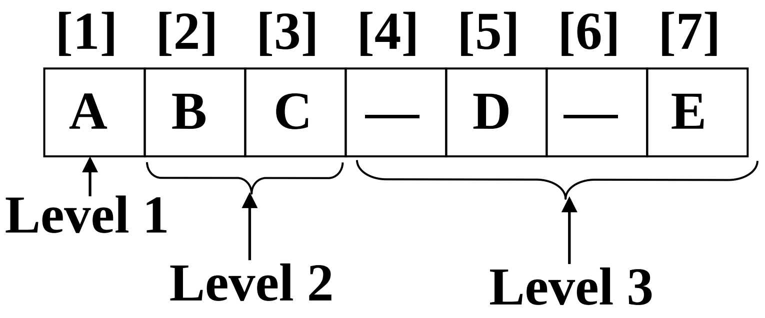
Left child of 5 ?

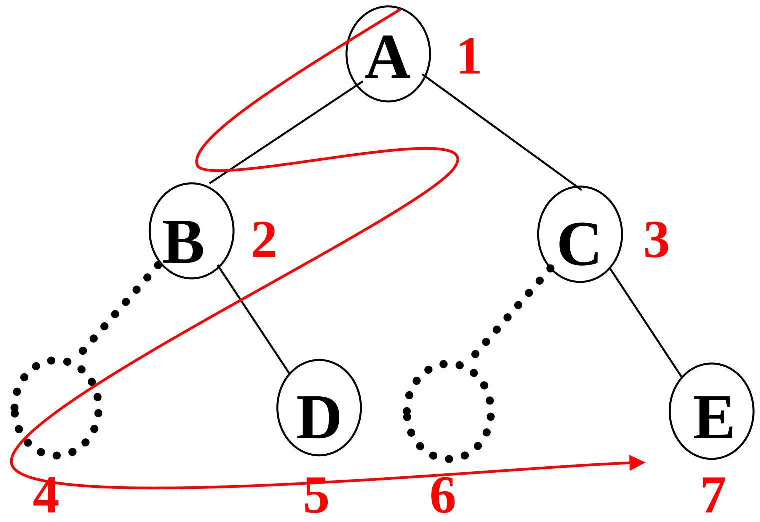
Right child of 5 ?

Left child of 13 ?

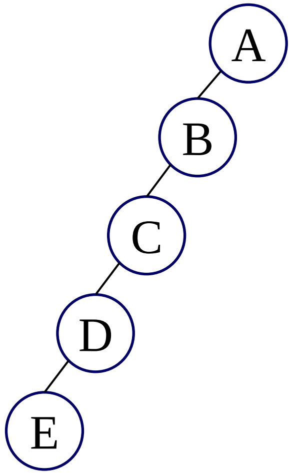
Right child of 13 ?

Array Representation

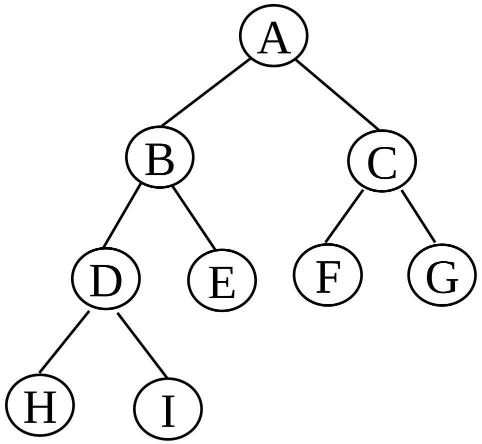




Array Representation



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



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

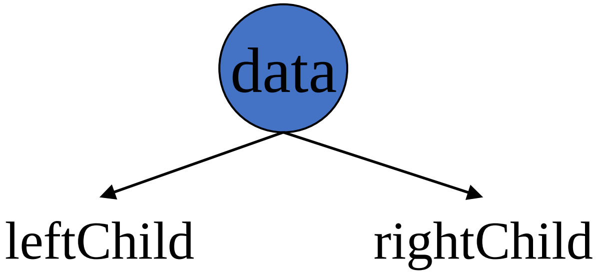
- ❖ Space wastes: In the worst case, a skewed tree of depth *k* will require 2^k-1 spaces. Only *k* spaces will be occupied

- ❖ Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes

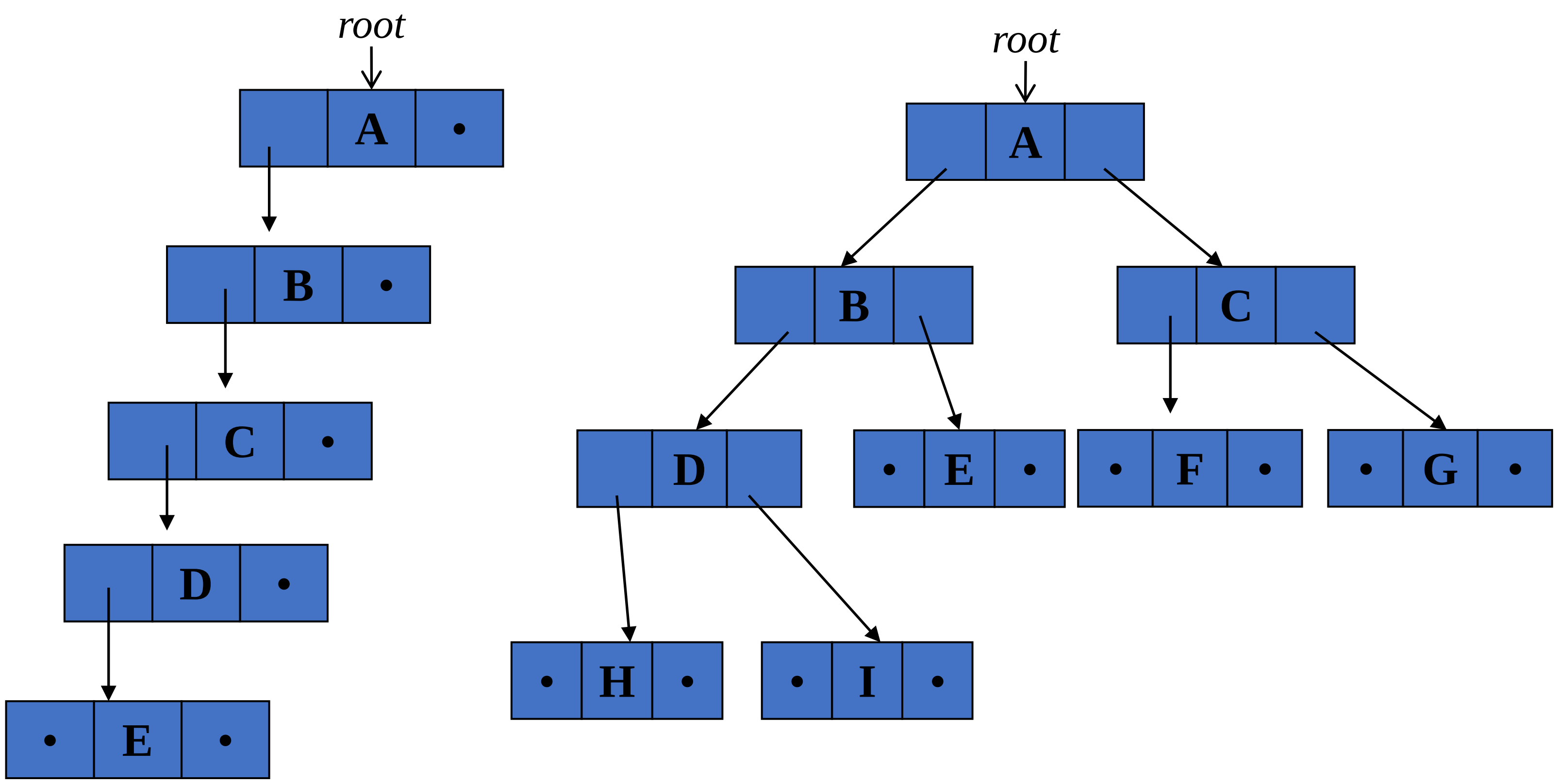
Linked Representation

- Although the array representation is good for complete binary trees, it is wasteful for many other binary trees
  - Can be overcome through the use of a linked representation
- Each node has three fields, leftChild, data, and rightChild

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



Examples of Linked Representation



Binary Tree Traversals

❖ How to traverse a tree or visit each node in the tree exactly once?

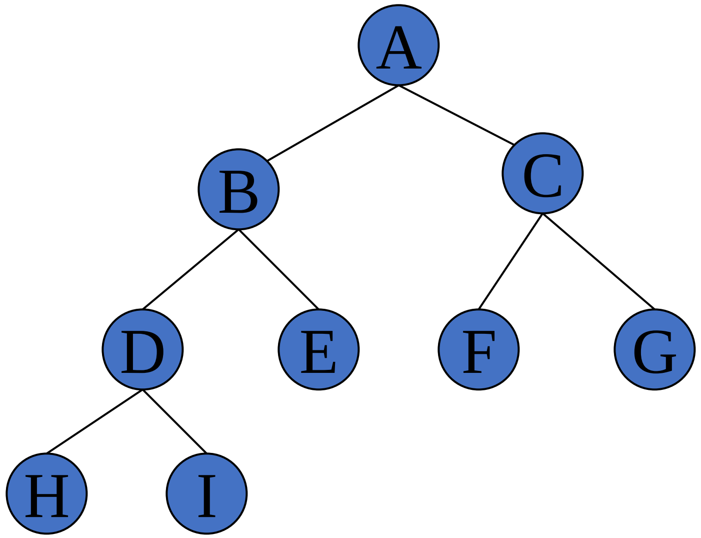
- Let *L*, *V*, and *R* stand for moving left, visiting the node, and moving right
- Six possible combinations of traversal

  LVR, LRV, VLR, VRL, RVL, RLV

- Adopt the convention that we traverse left before right, then only 3 traversals remain

  LVR (*inorder*), *LRV* (*postorder*), *VLR* (*preorder*)

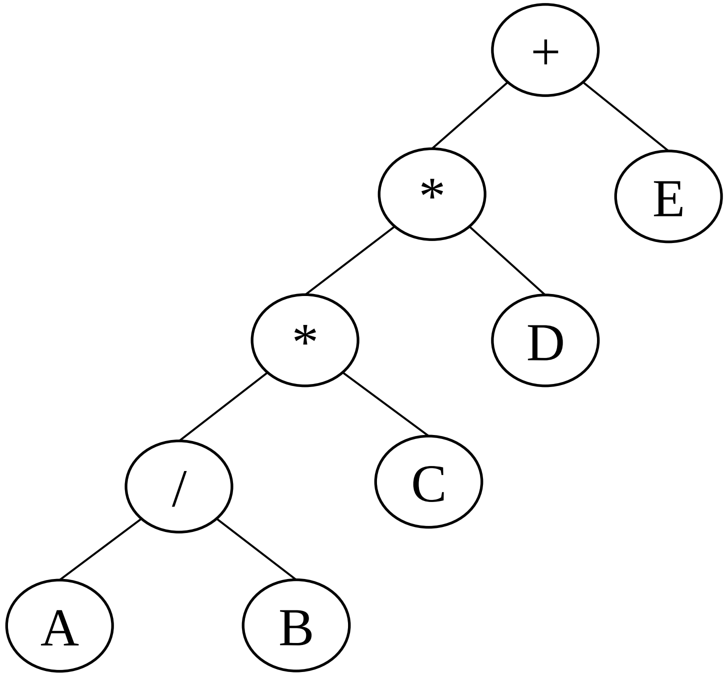
Binary Tree Traversals



Inorder : H – D – I – B – E – A – F – C – G
Preorder : A – B – D – H – I – E – C – F – G
postorder : H – I – D – E – B – F – G – C – A

Arithmetic Expression Using Binary Tree

❖ There is a natural correspondence between these traversals and producing the infix, postfix, and prefix forms of an expression



❖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)

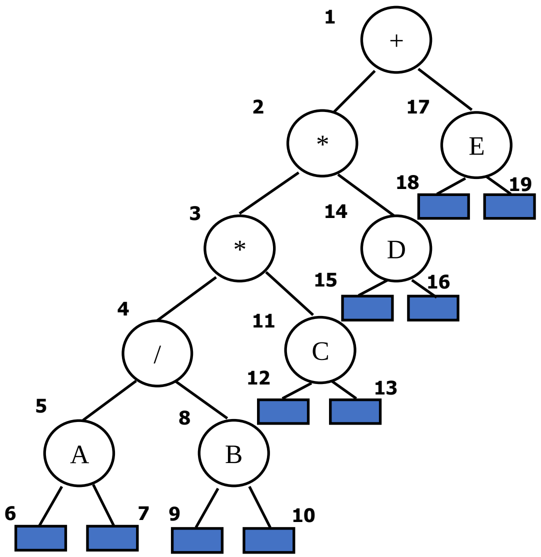
Inorder Traversal

❖ Informally, *inorder traversal* calls for

- moving down the tree toward the left until you can go no farther
- Then "visit" the node, move one node to the right and continue
- If you cannot move to the right, go back one more node.

❖ By recursion

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



output:A / B * C * D + E

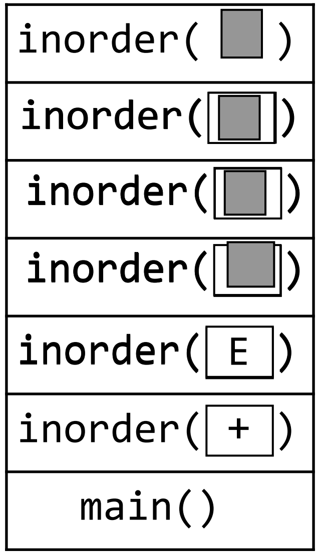
Call of *inorder*	Value in *root*	Action	*inorder*	in *root*	Value Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf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

Figure 5.17

Example

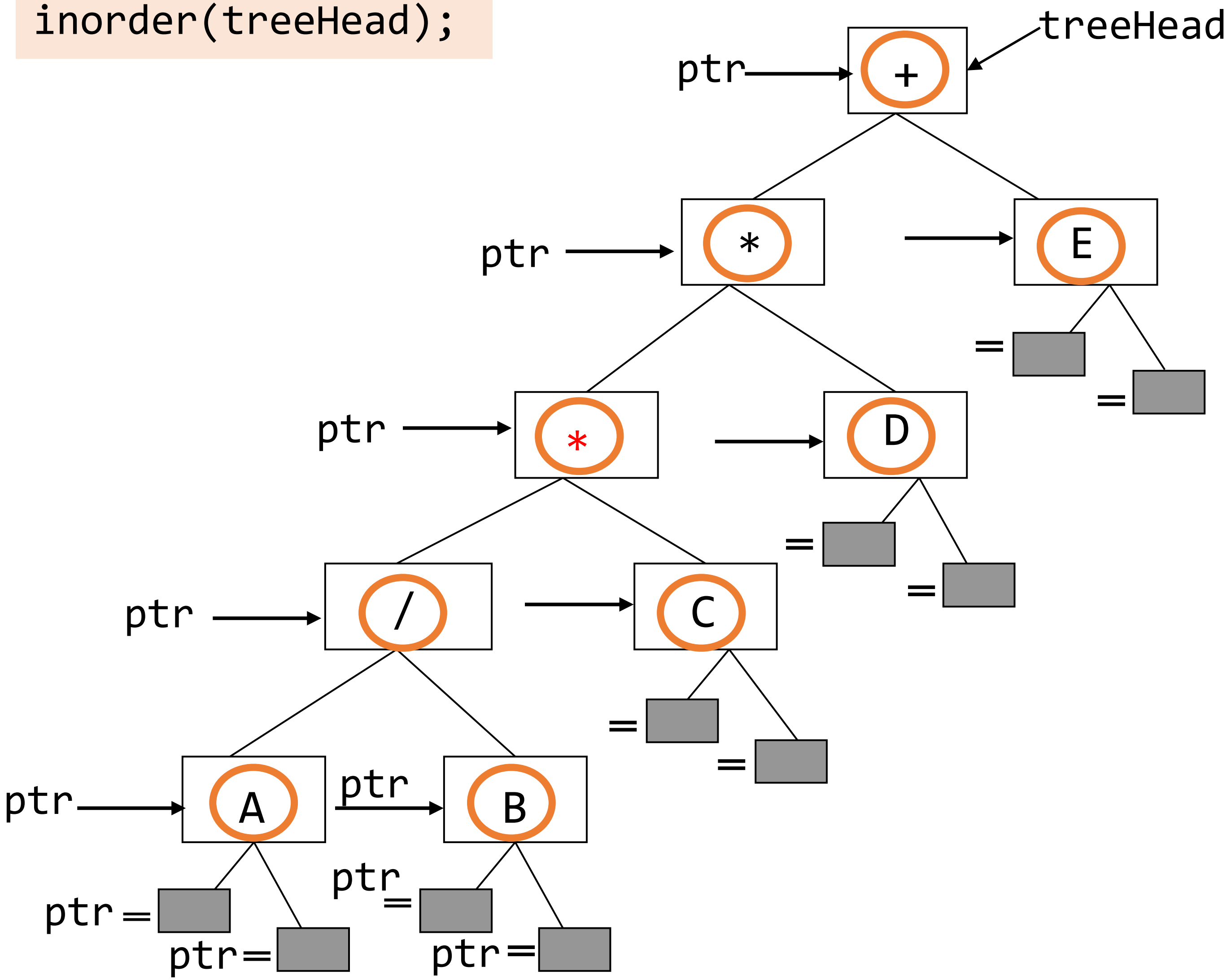
```
void inorder(treePointer ptr)
{
  if (ptr) {
    inorder(ptr->leftChild);
    printf("%c", ptr->data);
    inorder(ptr->rightChild);
  }
  return;
}
```

A / B * C * D + E



Function call stack

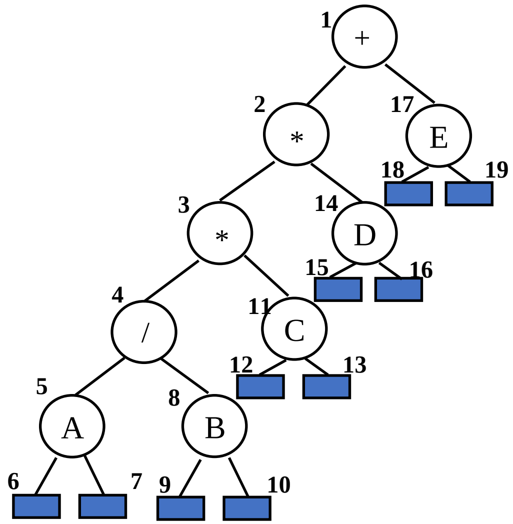
inorder(treeHead);



Preorder Traversal

- ❖ Visit a node, traverse left, and continue
- ❖ When you cannot continue, move right and begin again
- ❖ Or move back until you can move right and resume

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



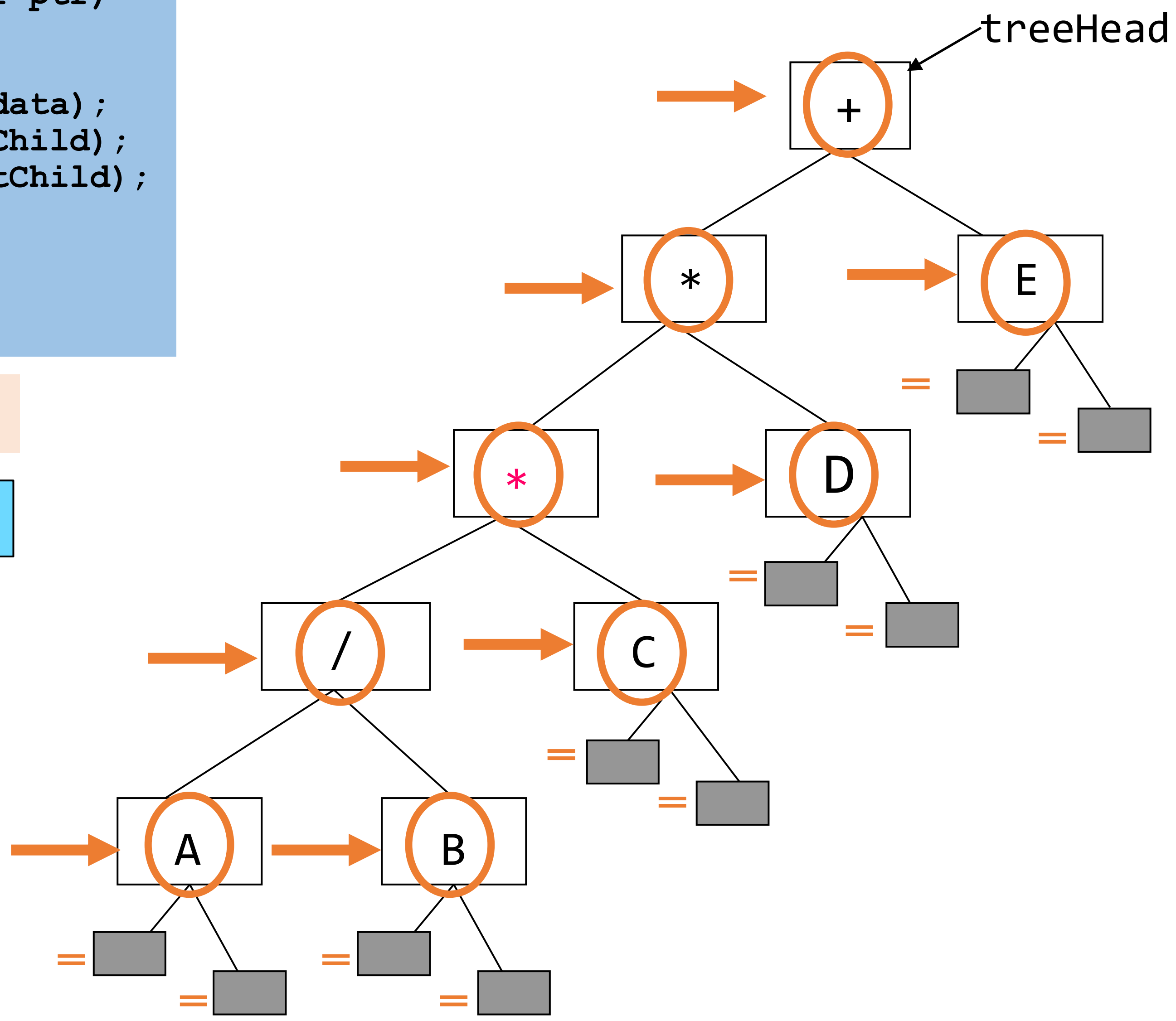
output: + * * / A B C D E

Example

```
void preorder(treePointer ptr)
{
    if (ptr) {
        printf("%c", ptr->data);
        preorder(ptr->leftChild);
        preorder(ptr->rightChild);
    }
    return;
}
```

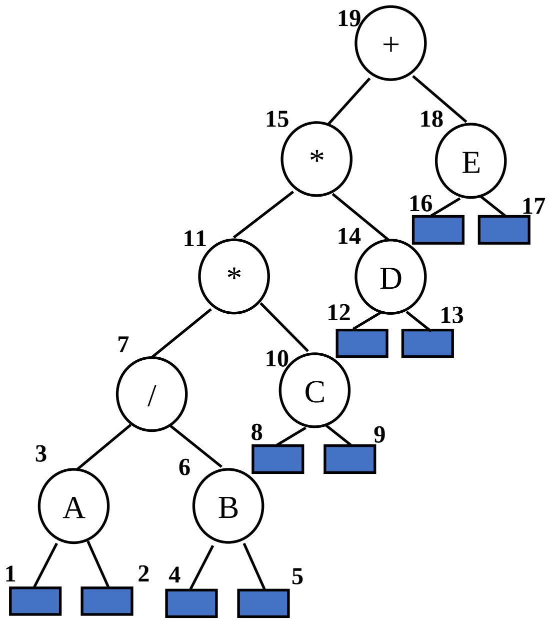
```
preorder(treeHead);
```

+ * * / A B C D E



Postorder Traversal

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



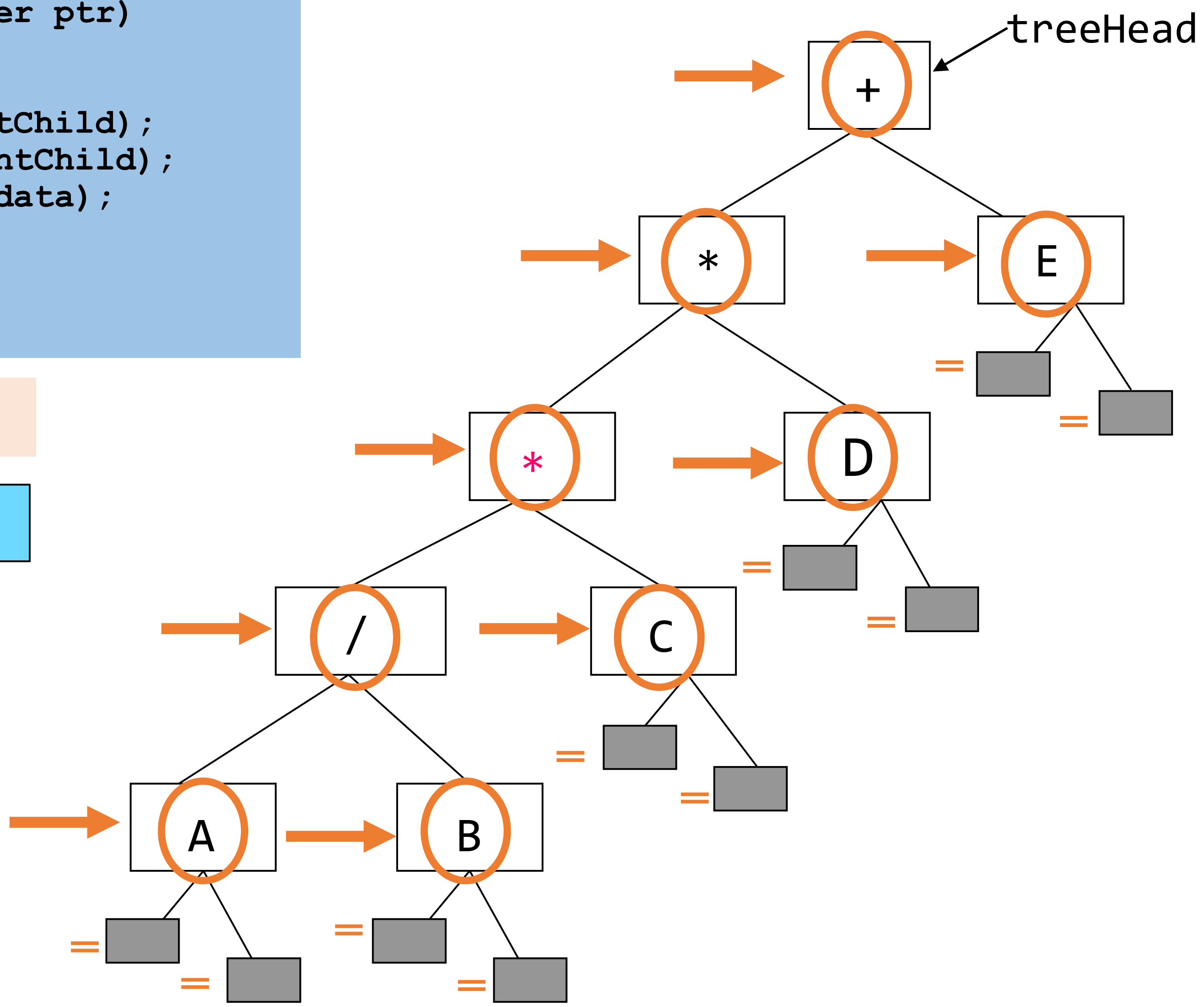
output: A B / C * D * E +

Example

```
void postorder(treePointer ptr)
{
    if (ptr) {
        postorder(ptr->leftChild);
        postorder(ptr->rightChild);
        printf("%c", ptr->data);
    }
        return;
}
```

postorder(treeHead);

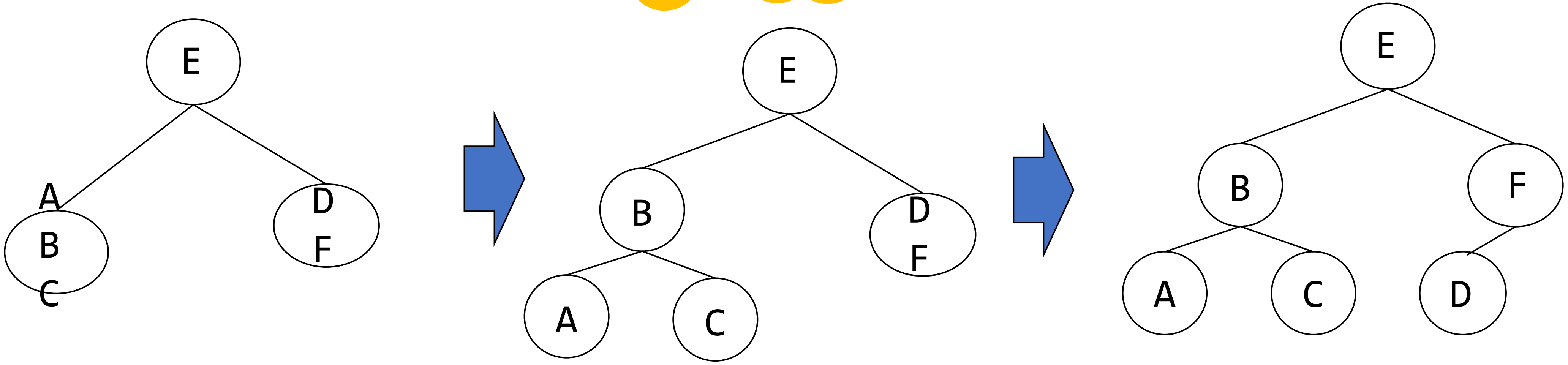
A B / C * D * E +



How to rebuild a tree from traversal results

Inorder traversal A B C E D F

Postorder traversal A C B D F E



Iterative Inorder Traversal

❖ We can develop equivalent iterative functions for the inorder, preorder, and postorder traversal functions using a stack

- ➢ Figure 5.17 implicitly shows the stacking and unstacking
  - A node that has no action indicates that the node is added to the stack
  - A *printf* action indicates that the node is removed from the stack

```
void iterInorder(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;
     }
}
```

Call of *inorder*	Value in *root*	Action	*inorder*	in *root*	Value 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	

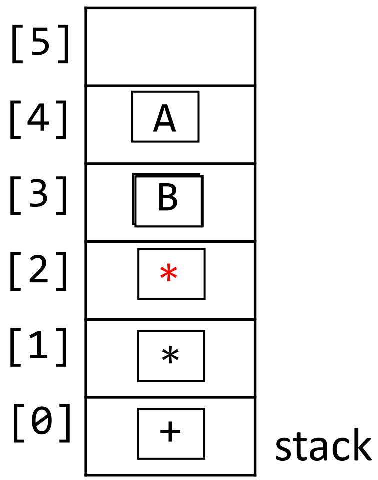
Figure 5.17

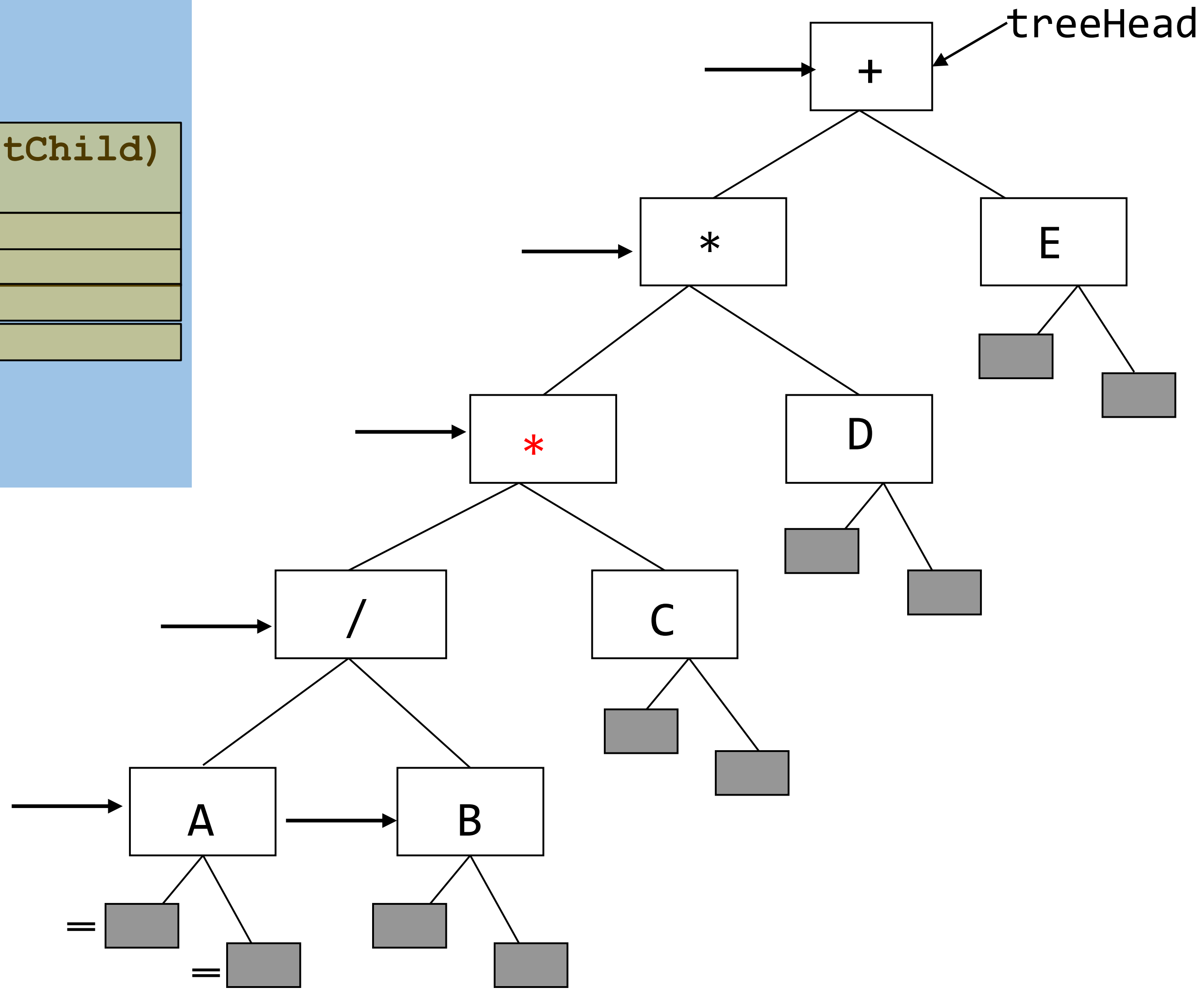
Example

```
iterInorder(treeHead);
```

```
int top = -1;
treePointer stack[MAX_STACK_SIZE];
void iterInorder(treePointer node)
{
    for (;;) {
        for (; node; node = node->leftChild)
            push(node);
        node = pop();
        if (!node) break;
        printf("%c", node->data);
        node = node->rightChild;
    }
    return;
}
```

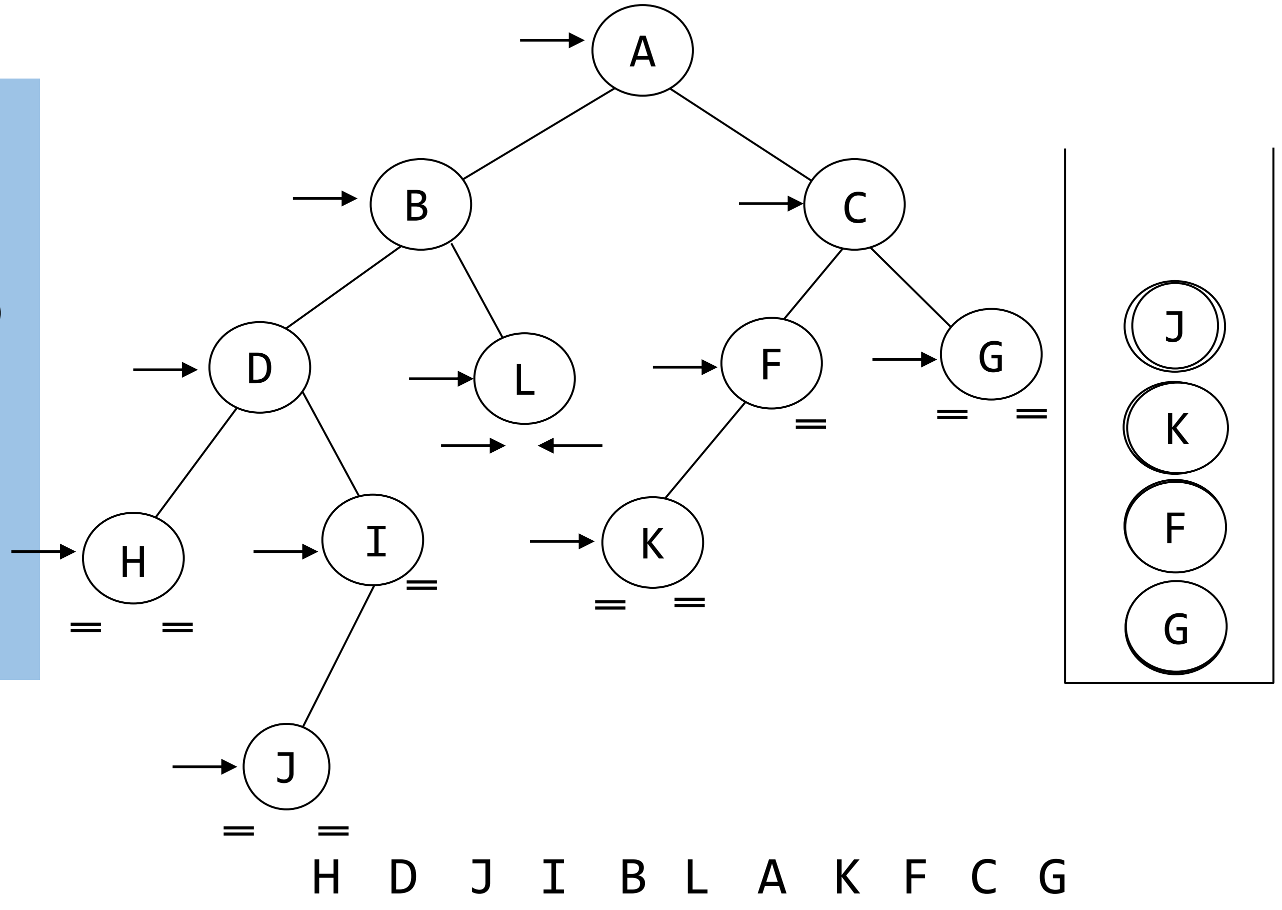
A /





Example

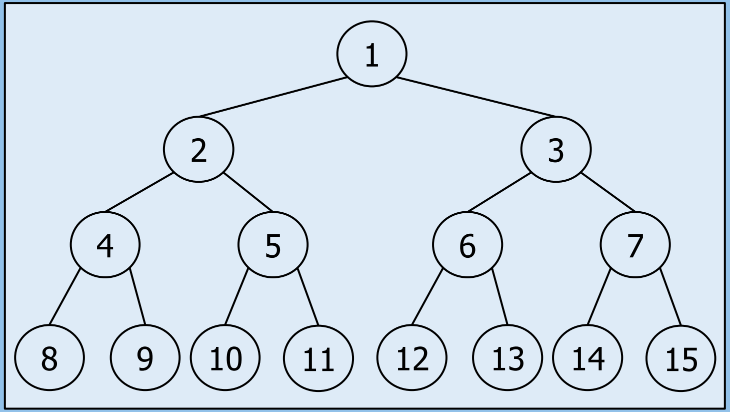
```
int top = -1;
treePointer stack[MAX_STACK_SIZE];
void iterInorder(treePointer node)
{
    for (;;) {
        for (; node; node = node->leftChild)
            push(node);
        node = pop();
        if (!node) break;
        printf("%c", node->data);
        node = node->rightChild;
    }
    return;
}
```

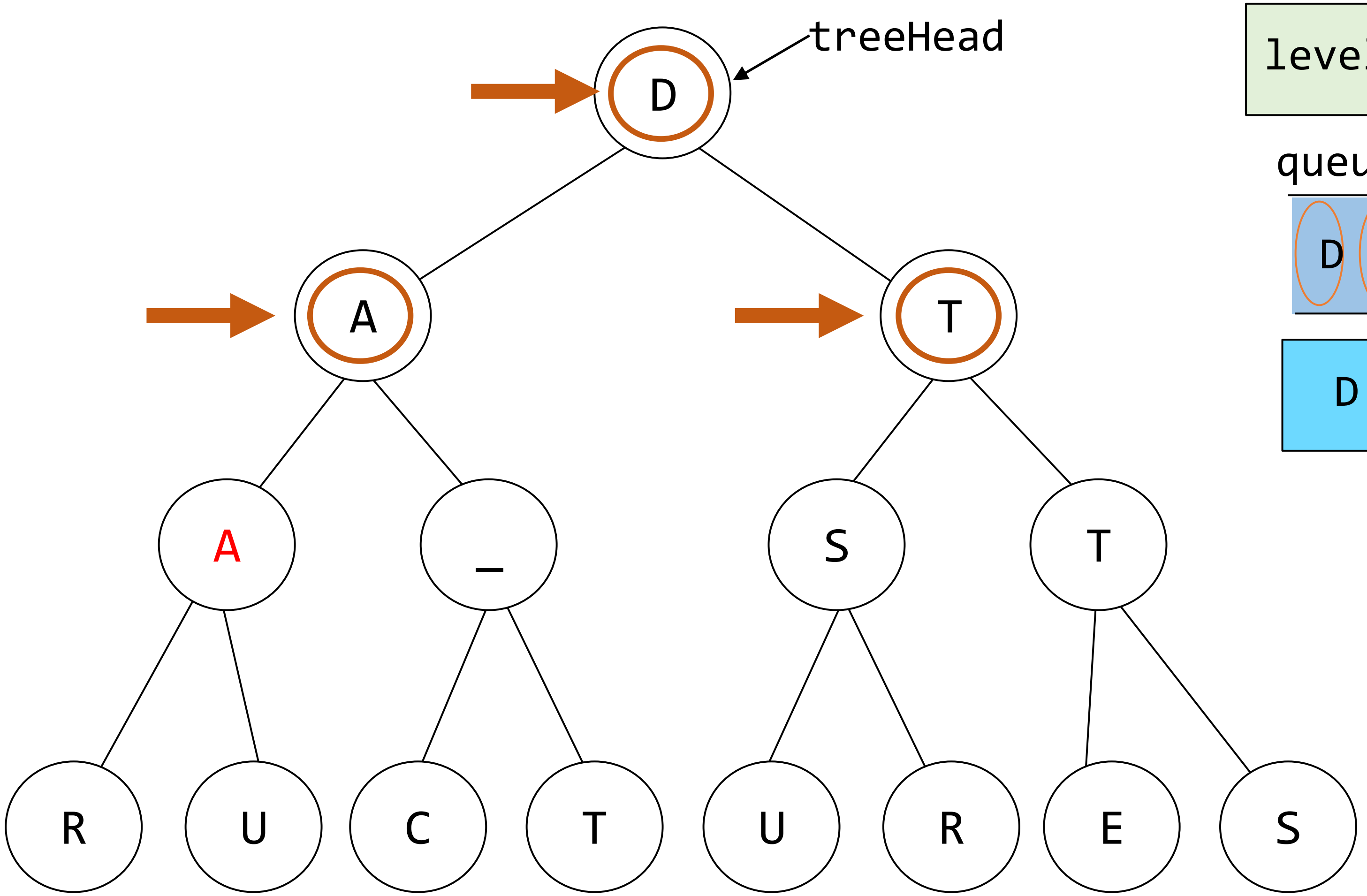


Level Order Traversal (=Breadth-first traversal)

- A traversal requiring a queue
  - Visit the root first, then the root's left child, followed by the root's right child

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





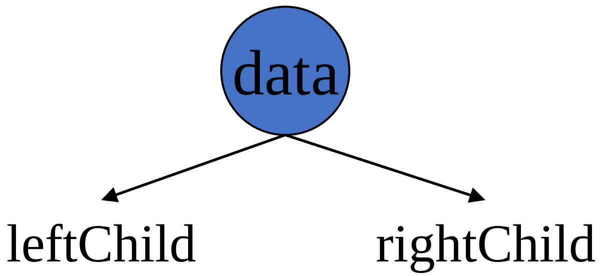
`levelOrder(treeHead);`

queue

D A T A _ S T

D A T

Representation of Trees



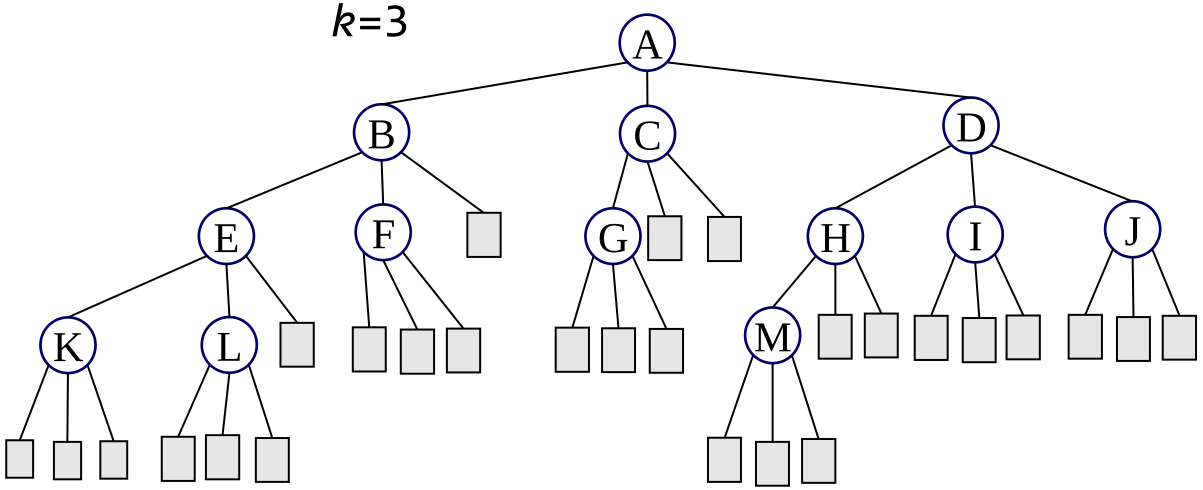
❖ K-ary Tree Representation

DATA	CHILD1	...	CHILD k

Possible node structure for a tree of degree k

Lemma 5.1 If T is k-ary tree with n nodes, each having a fixed size, then $n \cdot (k-1) + 1$ of the $n \cdot k$ child fields are 0, $n \geq 1$

k=3



# of non-empty child field = n -1

# of child fields = $n \cdot k$

# of empty child fields
= $n \cdot k - (n - 1)$
= $n \cdot (k-1) + 1$

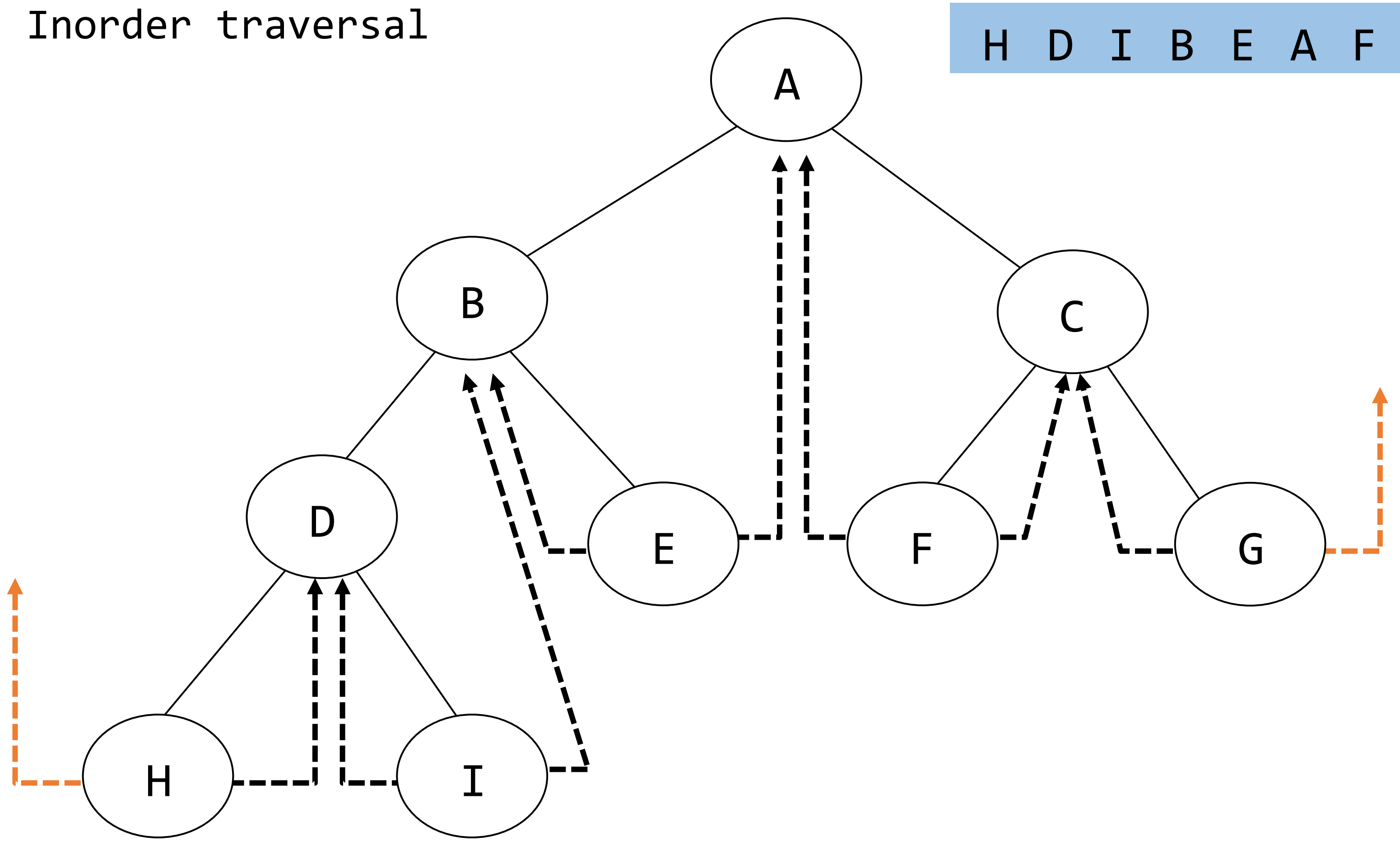
Threaded Binary Trees

- There are more null links than actual pointers at the linked representation of any binary tree
  - n: number of nodes
  - number of non-null links: n-1
  - total links: 2n
  - null links: 2n-(n-1) = n+1
  - Replace the null links by pointers, called *threads*, to other nodes
- Rules for constructing the threads
  - If ptr->leftChild is null
    - Replace it with a pointer to the node visited *before ptr* in an *inorder traversal*
    - :*inorder predecessor* of *ptr (중위 선행자)*
  - If ptr->right_child is null
    - Replace it with a pointer to the node visited *after ptr* in an *inorder traversal*
    - *inorder successor* of *ptr (중위 후속자)*

An Example of Threaded Binary Tree

Inorder traversal

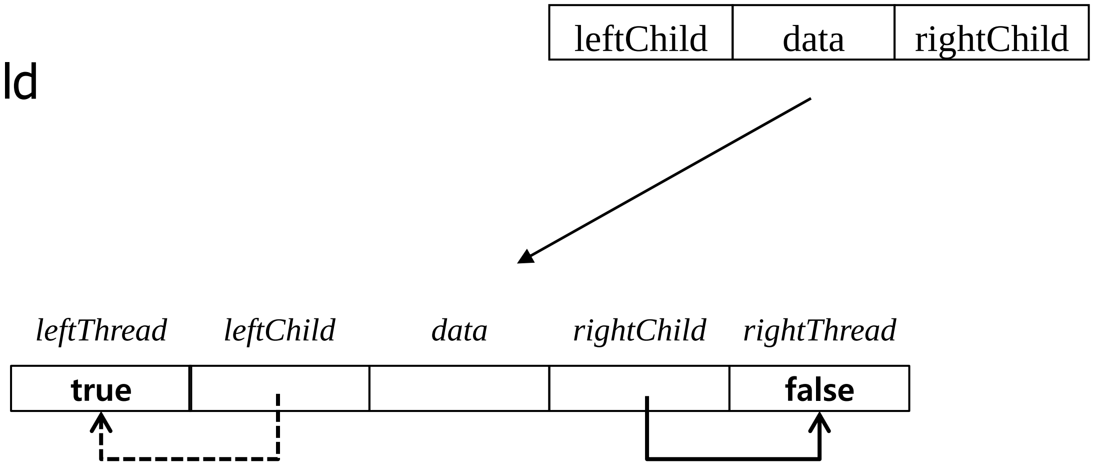
H D I B E A F C G



Threaded Binary Trees

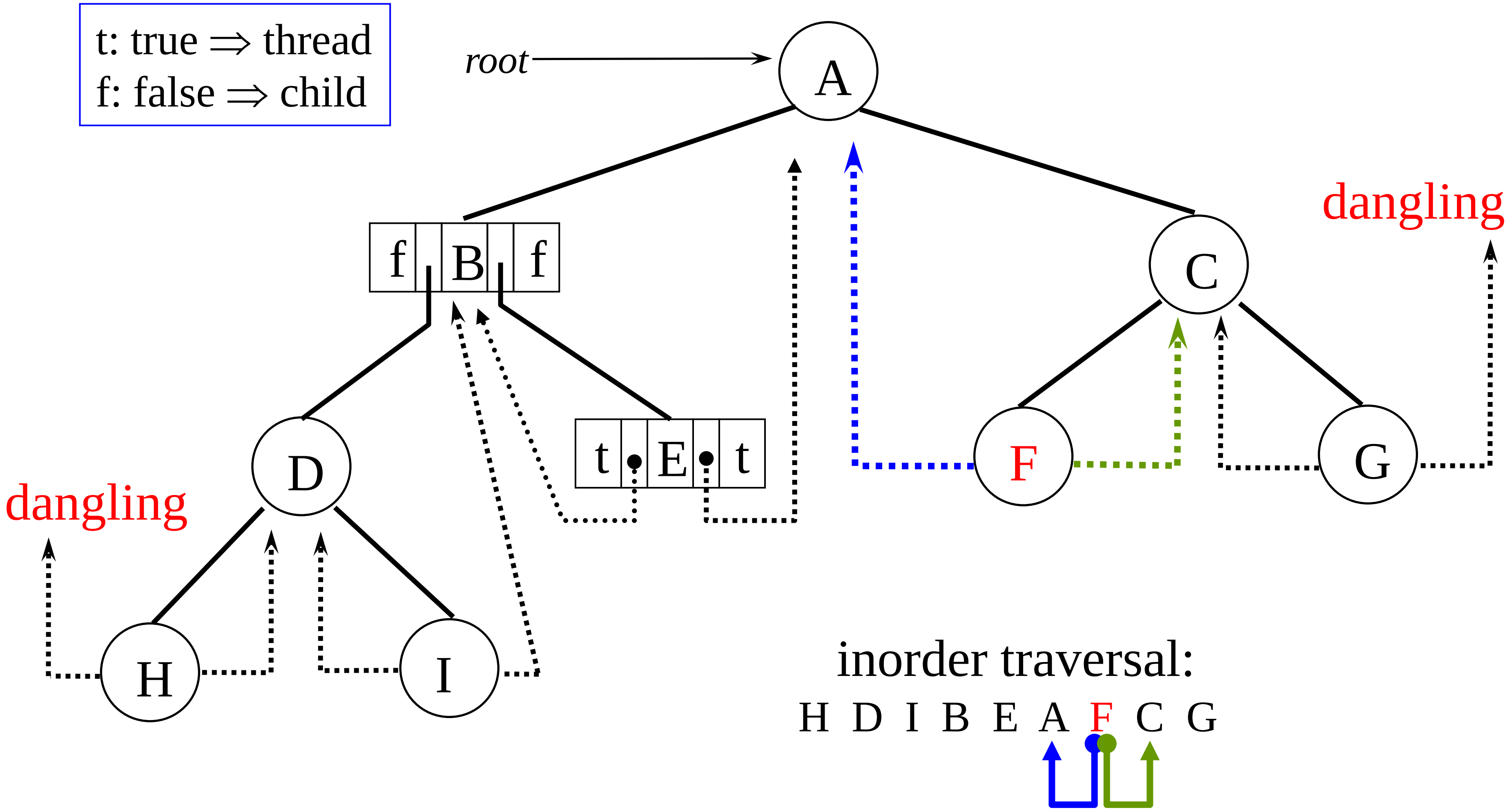
- Two additional fields of the node structure
  - *leftThread* and *rightThread*
  - If *ptr->leftThread*=TRUE
    - *ptr->leftChild* contains a thread
    - Otherwise it contains a pointer to the left child
  - Similarly for the *ptr->rightThread*

```
typedef struct threadedTree *threadedPointer;
typedef struct threadedTree {
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
};
```



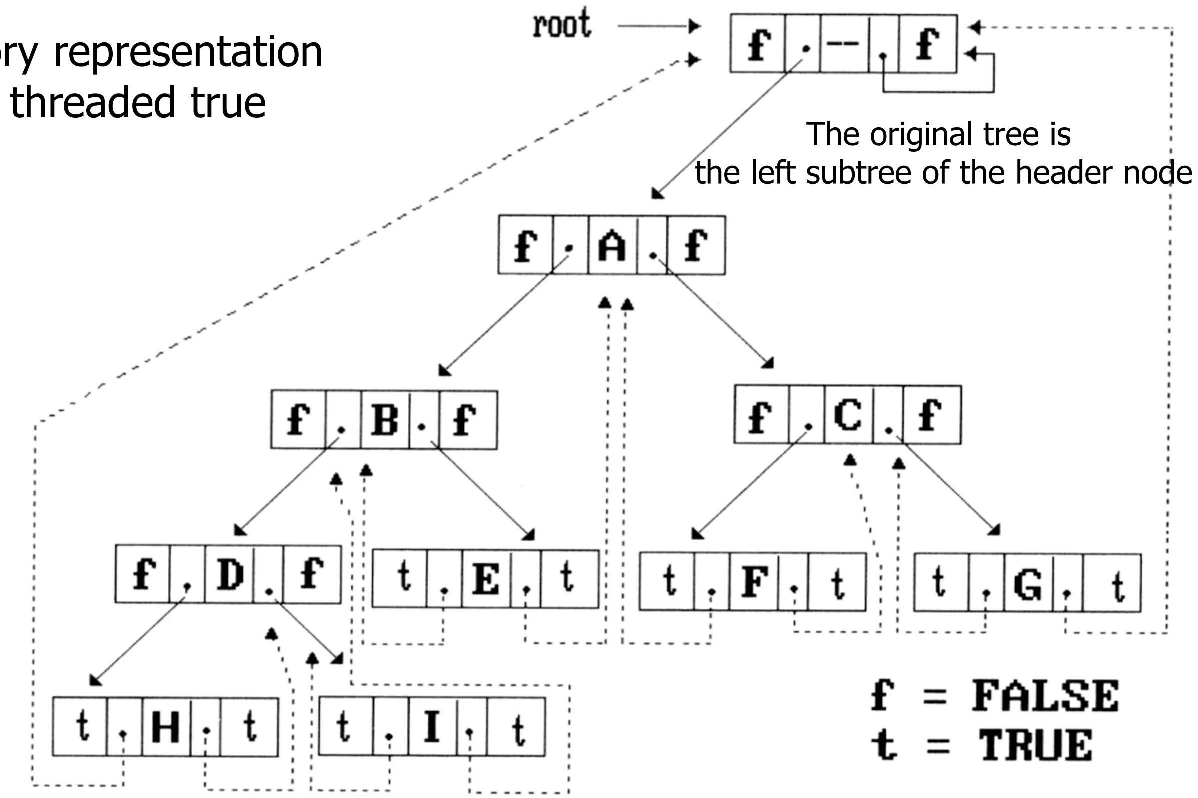
An empty threaded binary tree => a header node

Threaded Binary Trees



Threaded Binary Trees

Memory representation
of threaded true



- ❖ Avoid being dangling pointers
  - The left pointer of H and the right pointer of G
  - Create a root node and make these pointers point the root node

Threaded Binary Trees

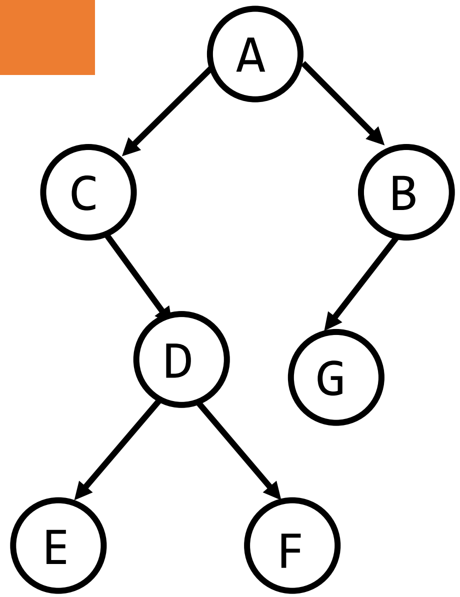
- Inorder traversal of a threaded binary tree
  - We can perform an inorder traversal without making use of a stack

1. If *ptr->rightThread* = *TRUE* , the inorder successor of *ptr* is *ptr->rightChild*
2. Otherwise, follow a path of left-child links from the right-child of *ptr* until we reach a node with *leftThread* = *TRUE*

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

```
threadedPointer insucc(threadedPointer tree)
{         /*find the inorder successor of tree in a threaded binary tree */
          threadedPointer temp;
          temp = tree->rightChild;
          if( !tree->rightThread )
                    while( !temp->leftThread )
                              temp = temp->leftChild;
          return temp;
}
```

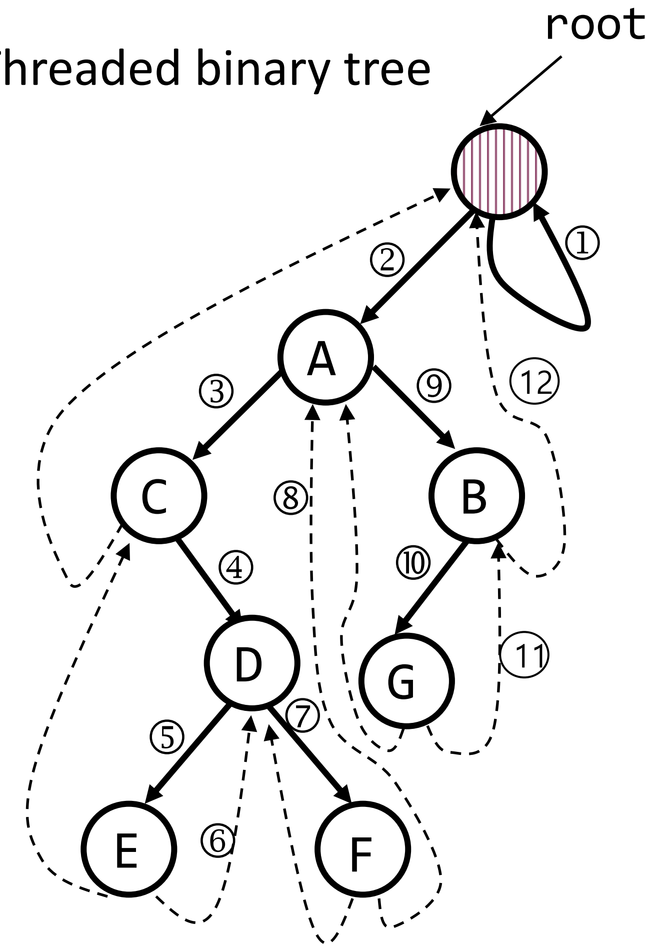
Example



Inorder traversal result

C E D F A G B

Threaded binary tree




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

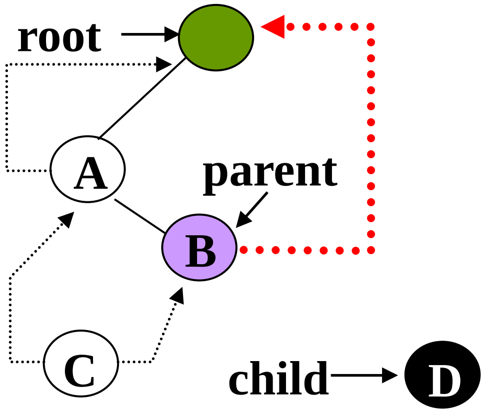
```
threadedPointer insucc(threadedPointer tree)
{       /*find the inorder successor of tree in a threaded binary tree */
        threadedPointer temp;
        temp = tree->rightChild;
        if( !tree->rightThread )
                while( !temp->leftThread )
                        temp = temp->leftChild;
        return temp;
}
```

Threaded Binary Trees

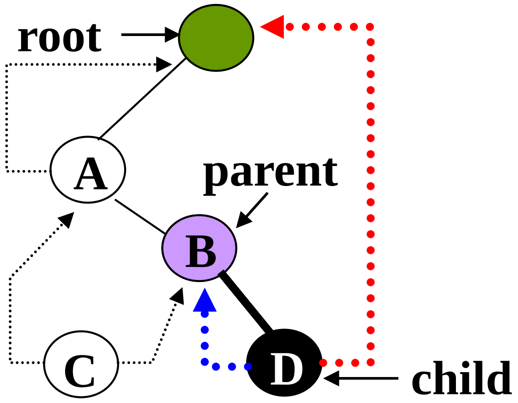
❖ Inserting a node into a threaded binary tree

➢ Inserting *D* as the right child of a node *B*

- If *B* has an empty right subtree, then the insertion is simple



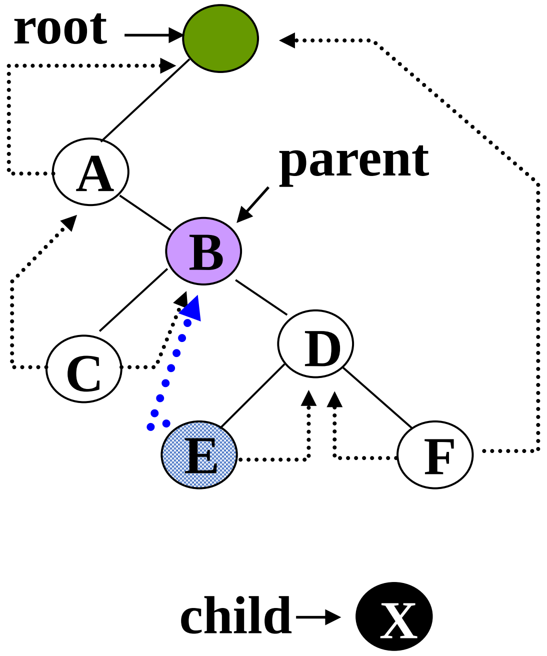
Before



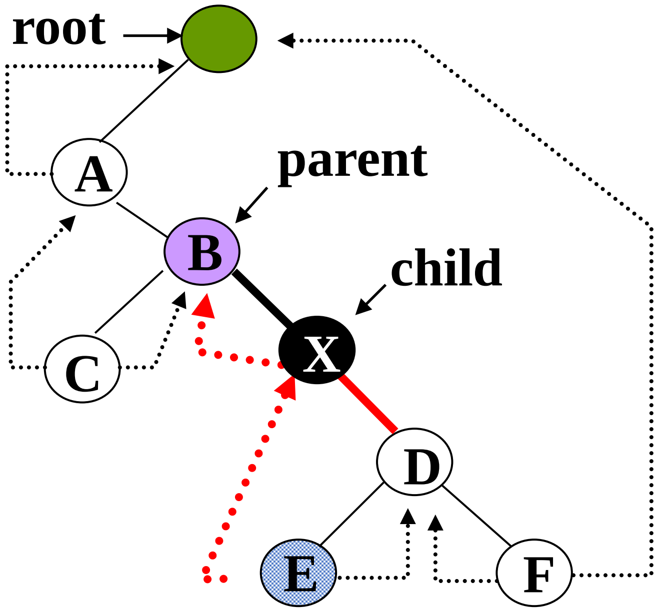
After

Threaded Binary Trees

❖ Inserting a right child of a parent in a threaded binary tree



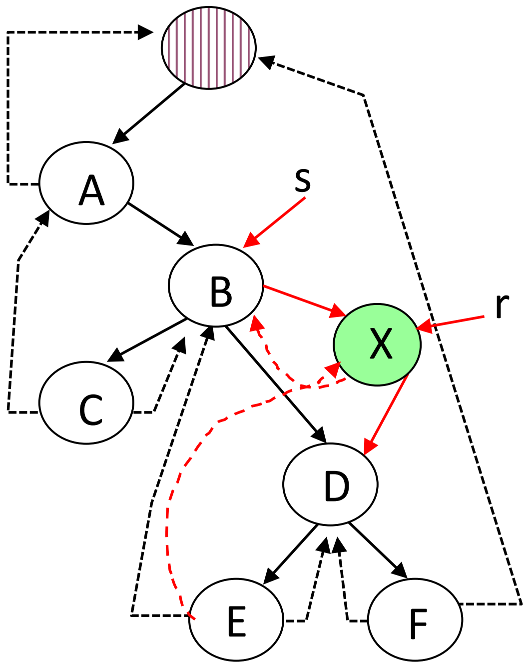
Before



After

Inserting a Node as a Right Child

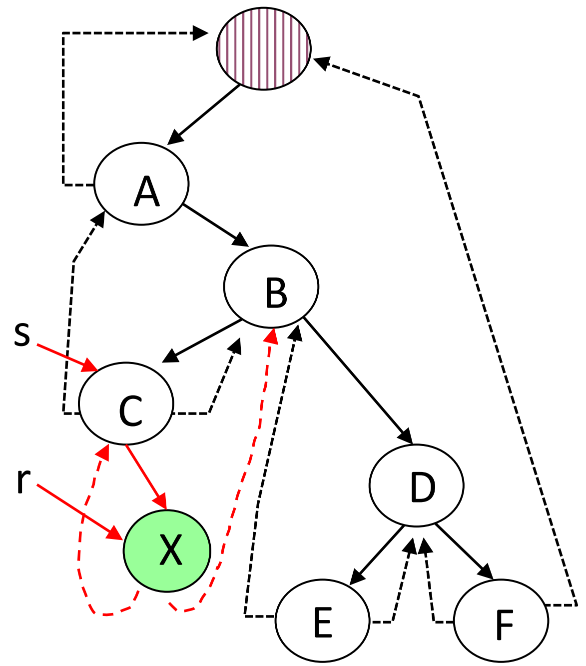
```
void insertRight(threadedPointer s, threadedPointer r)
/* insert r as the right child of s */
{
   threadedPointer temp;
   r->rightChild = s->rightChild;
   r->rightThread = s->rightThread;
   r->leftChild = s;
   r->leftThread = TRUE;
   s->rightChild = r;
   s->rightThread = FALSE;
   if (!r->rightThread){
      temp = insucc(r);
      temp->leftChild = r;
   }
}
```



Inorder traversal
Before insertion: A C B E D F
After insertion: A C B X E D F

Inserting a Node as a Right Child

```
void insertRight(threadedPointer s, threadedPointer r)
/* insert r as the right child of s */
{
   threadedPointer temp;
   r->rightChild = s->rightChild;
   r->rightThread = s->rightThread;
   r->leftChild = s;
   r->leftThread = TRUE;
   s->rightChild = r;
   s->rightThread = FALSE;
   if (!r->rightThread){
      temp = insucc(r);
      temp->leftChild = r;
   }
}
```



Heaps

❖ Priority queues (우선순위 큐)

- Heaps are frequently used to implement *priority queues*
- The element to be deleted is the one with highest (or lowest) priority

ADT *MaxPriorityQueue* is
objects: a collection of n>0 elements, each element has a key
functions:
for all $q \in MaxPriorityQueue$, $item \in Element$, $n \in integer$

MaxPriorityQueue create(*max_size*)	::= create an empty priority queue
Boolean isEmpty(q, n)	::= **if**(n>0) **return** FALSE **else return** TRUE
Element top(q, n)	::= **if**(!isEmpty(q, n)) **return** an instance of the largest element in q **else return** error
Element pop(q, n)	::= **if**(!isEmpty(q, n)) **return** an instance of the largest element in q and remove it from the heap **else return** error
MaxPriorityQueue push(q, *item*, n)	::= insert *item* into q and return the resulting priority queue

Heaps

- Example: selling the services of a machine
  - Each user pays a fixed amount per use
  - The time needed by each user is different
  - How to maximize the returns from this machine under the assumption that the machine is not to be kept idle unless no user is available
  - This can be done by maintaining a priority queue of all persons waiting to use the machine
    - The user with the smallest time requirement is selected
    - A min priority queue is required

Max Heap

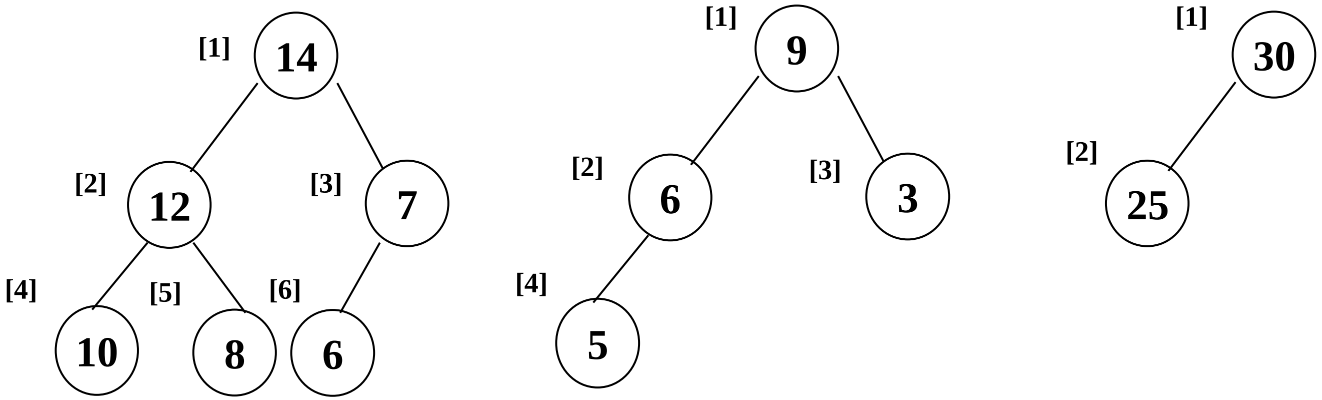
- A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children (if any)
- A *max heap* is a complete binary tree that is also a *max tree*

- A *min tree* is a tree in which the key value in each node is no larger than the key values in its children (if any)
- A *min heap* is a complete binary tree that is also a *min tree*

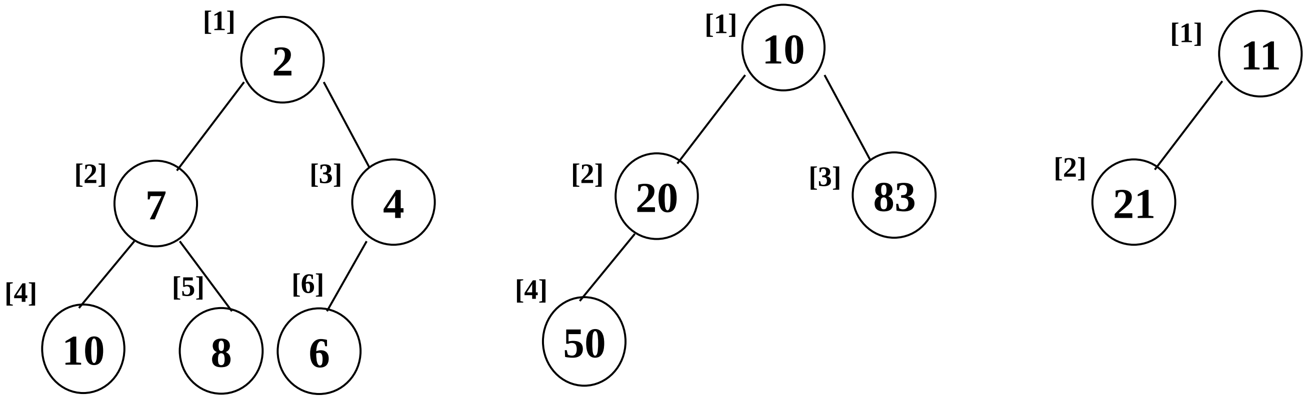
- The basic operations are the same as those for a max priority queue
- Since a max heap is a complete binary tree, we represent it using an array heap

Heaps

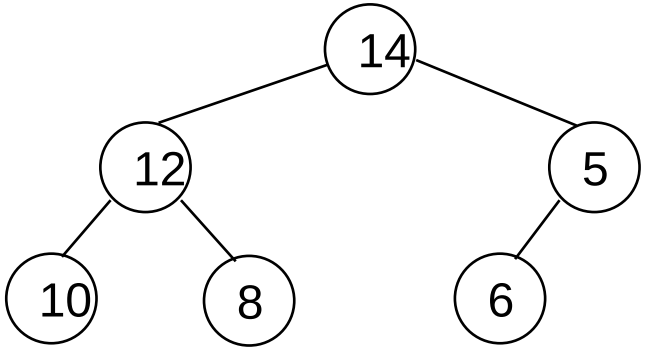
❖ Examples of max heap

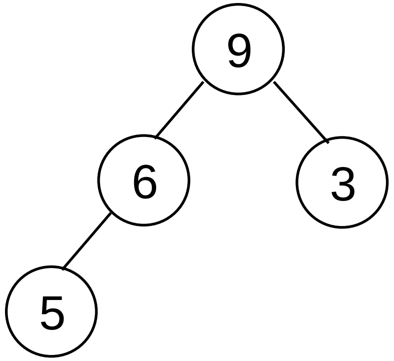


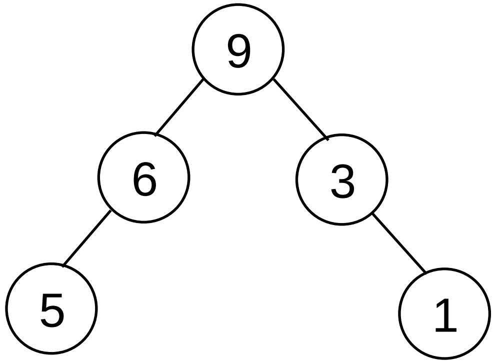
❖ Examples of min heap

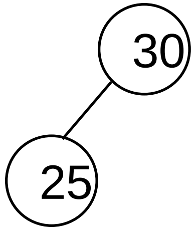


Are these trees max heaps?

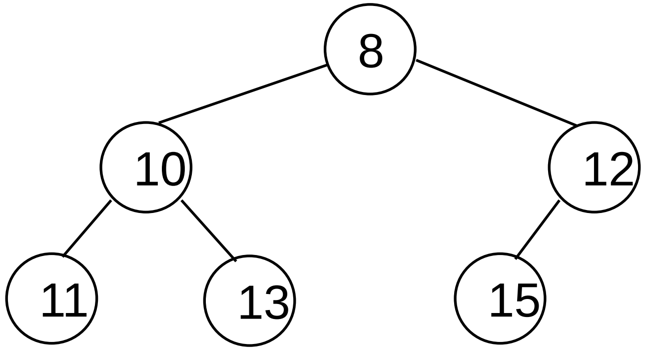


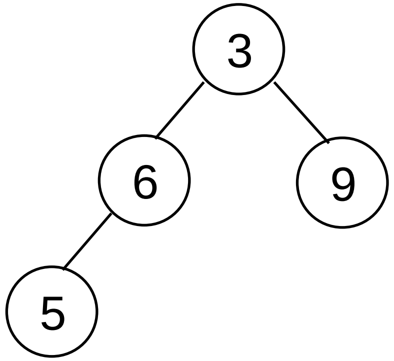


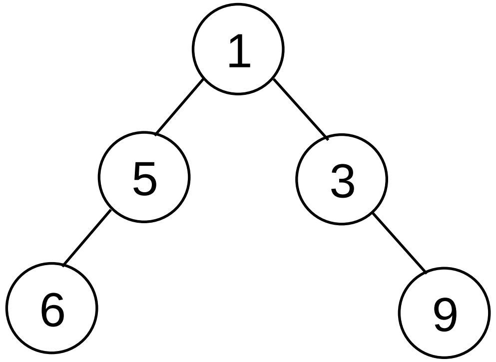


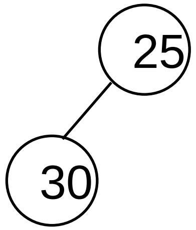


Are these trees min heaps?









Priority Queues

- Priority queue
  - The items added to a queue have a priority associated with them (payment, importance, …)
  - A queue in which the items are sorted so that the **highest priority item is always the next one to be extracted**

- We could use a tree structure
  - It generally provides O(log *n*) performance for both insertion and deletion (*n* is the number of node in a tree)
  - Unfortunately, if the tree becomes unbalanced, performance will degrade to O(*n*) in pathological cases

    → This will probably not be acceptable when dealing with time critical cases.

- Heap will provide guaranteed O(log *n*) performance for both insertion and deletion

Representations of Priority Queues

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	$O(\log_2 n)$	$O(\log_2 n)$

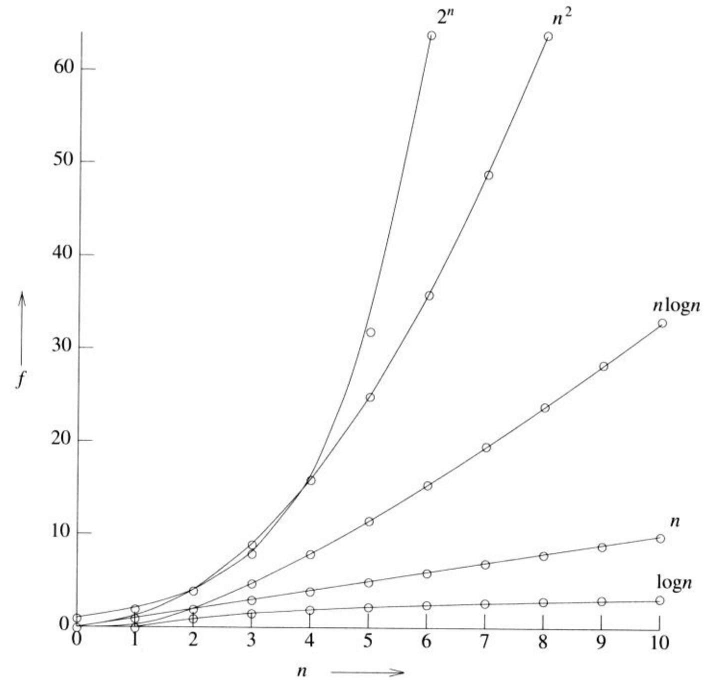
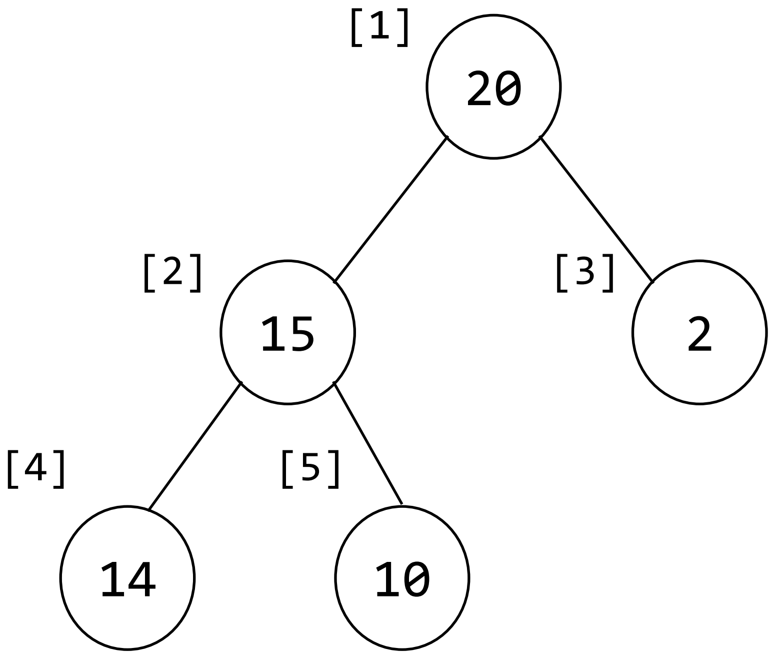


Figure 1.8 Plot of function values

Implementation of Max Heap

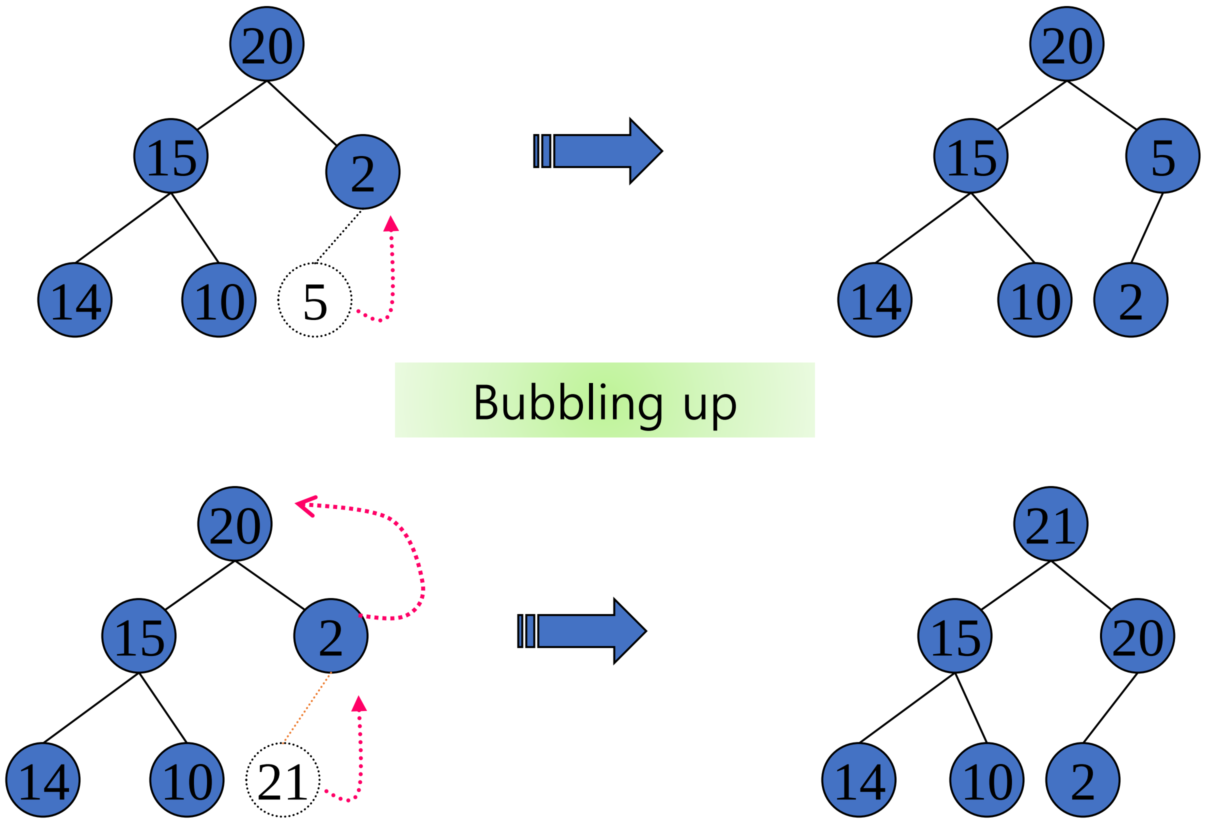
```
#define MAX_ELEMENTS 200 /* maximum size of heap+1 */
#define HEAP_FULL(n)  (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n)  (!n)
typedef struct {
        int key;
        /* other fields */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	20	15	2	14	10		



Insert 21?

Insertion into a Max Heap



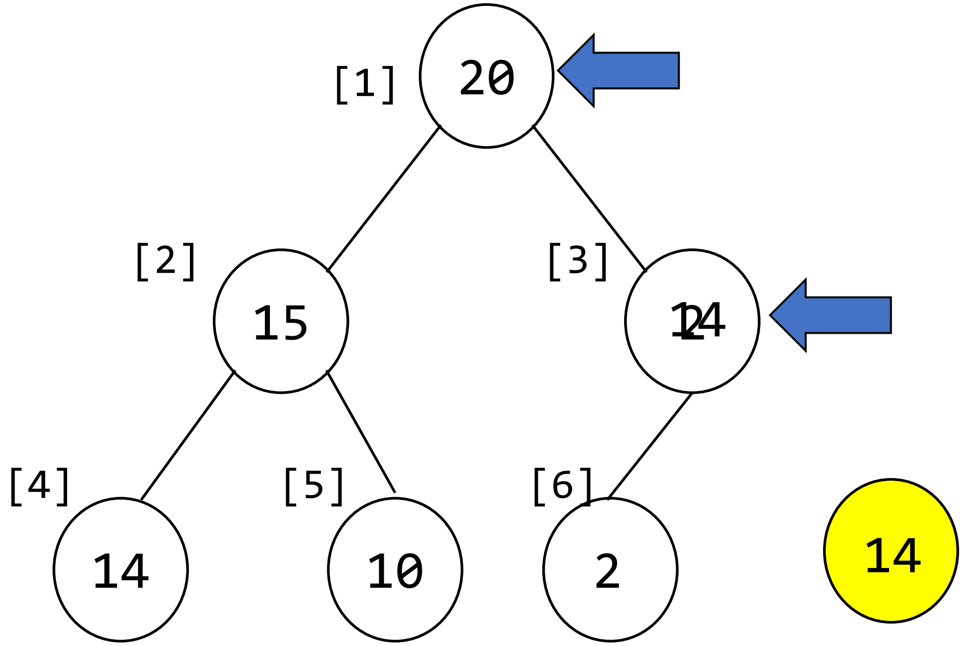
Insertion into a Max Heap

```
#define MAX_ELEMENTS 200    /* maximum heap size + 1 */
#define HEAP_FULL(n) (n==MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct{
         int key;
         /* other fields */
}element;
element heap[MAX_ELEMENTS];
int n=0;

void push(element item, int *n)
{        /* insert item into a max heap of current size * n */
         int i;
         if (HEAP_FULL(*n)) {
             fprintf(stderr, "the heap is full.\n");
             exit( EXIT_FAILURE );
         }
         i = ++(*n);
         while ( (i!=1) && (item.key>heap[i/2].key) )  {
                  heap[i] = heap[i/2];
                  i /= 2;
         }
         heap[i]= item;
}
```

Example

```
element e;
e.key=14;
no=5;
push(e, &no);
```

```
void push(element item, int *n)
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full. ");
        exit(EXIT_FAILURE);
    }
    i = ++(*n);   no=6  i=6
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;  i=3
    }
    heap[i] = item;
}
```



Analysis of *push*

The height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$ where $\lceil x \rceil$ is the smallest integer $\geq x$

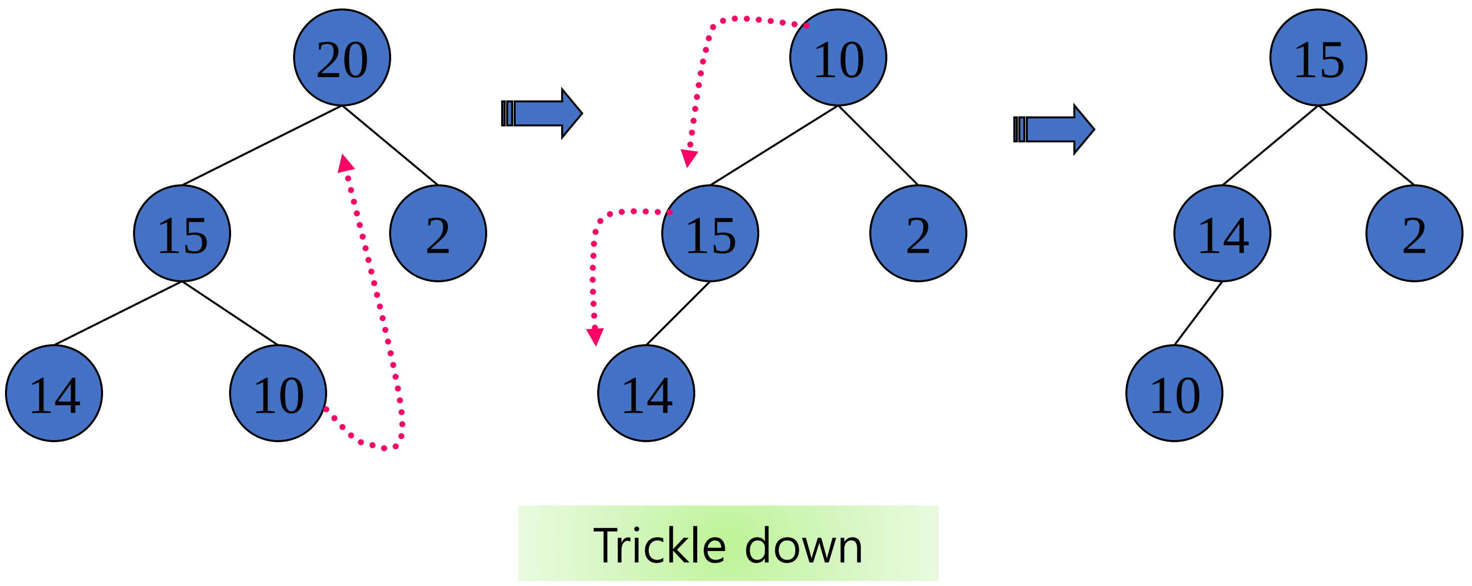
By Lemma 5.2,

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

- Complete Binary tree with n nodes
  $\Rightarrow$ its height is $\lceil \log_2(n+1) \rceil$
- The while loop is iterated $(\log_2 n)$ times.
  $\Rightarrow$ The complexity of the insertion functions is $O(\log_2 n)$

Deletion from a Max Heap

- ❖ When an element is to be deleted from a max heap, it is taken from the root of the heap
- ❖ Remove '20'

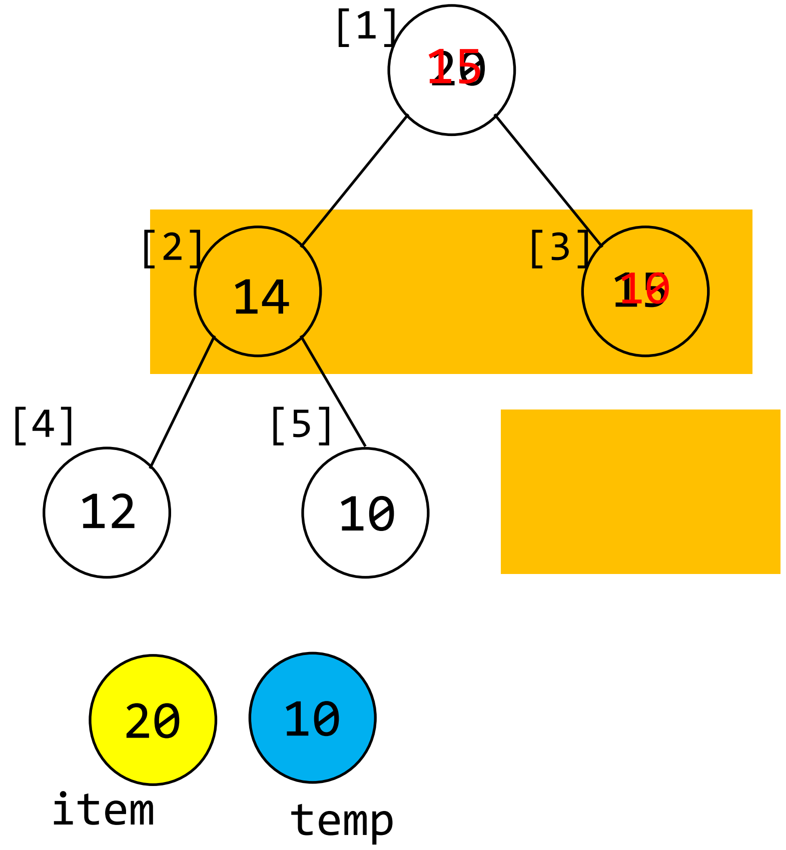


Deletion from a Max Heap

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

Example

```
int no=5;
element e=pop(&no);
```

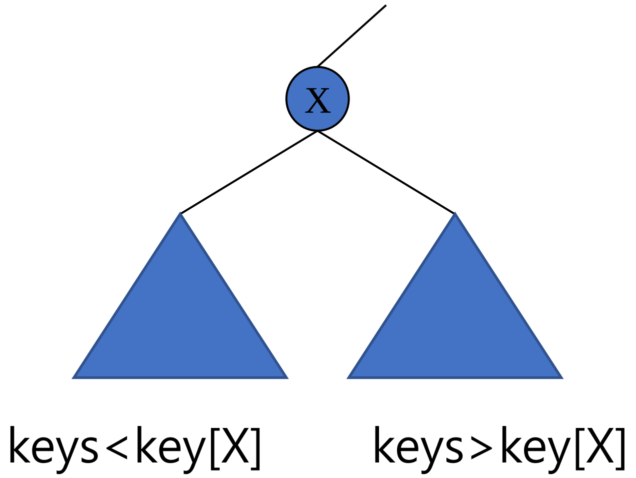



```
element pop(int *n)
{
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty");
        exit(EXIT_FAILURE);
    }
    item = heap[1];
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
    while (child <= *n) {
      if((child<*n)&&
          (heap[child].key<heap[child+1].key))
          child++;  child=3
      if (temp.key >= heap[child].key) break;
      heap[parent] = heap[child];
      parent = child;    parent=3
      child *= 2;        child=6
   }
   heap[parent] = temp;
   return item;
}
```

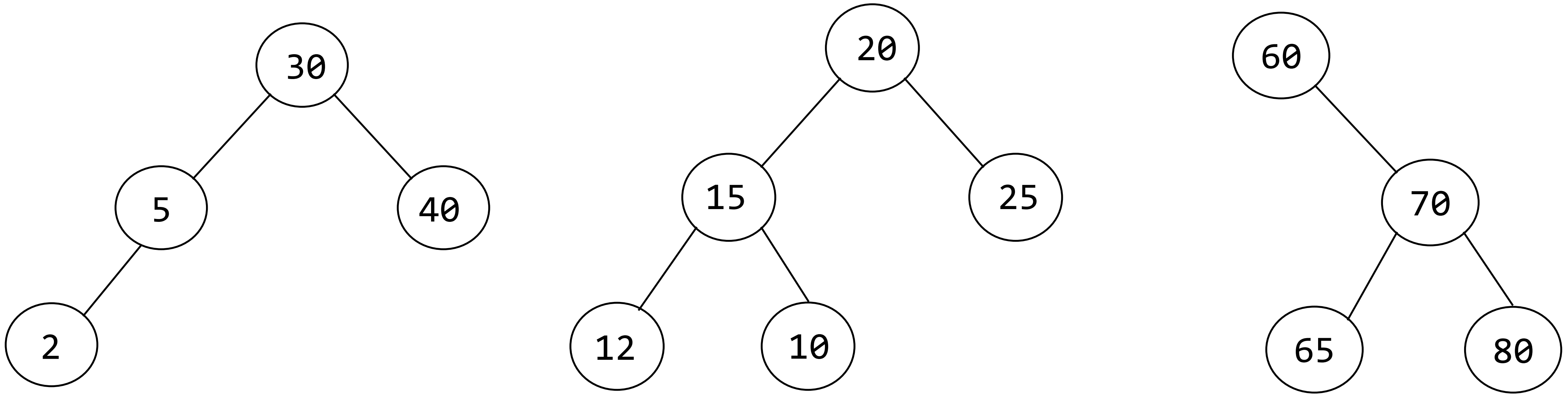
Binary Search Tree

❖ A *binary search tree* is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1) Each node has exactly one key and the keys in the tree are distinct
2) The keys (if any) in the left subtree are smaller than the key in the root
3) The keys (if any) in the right subtree are larger than the key in the root
4) The left and right subtrees are also binary search trees



Examples of Binary Search Tree



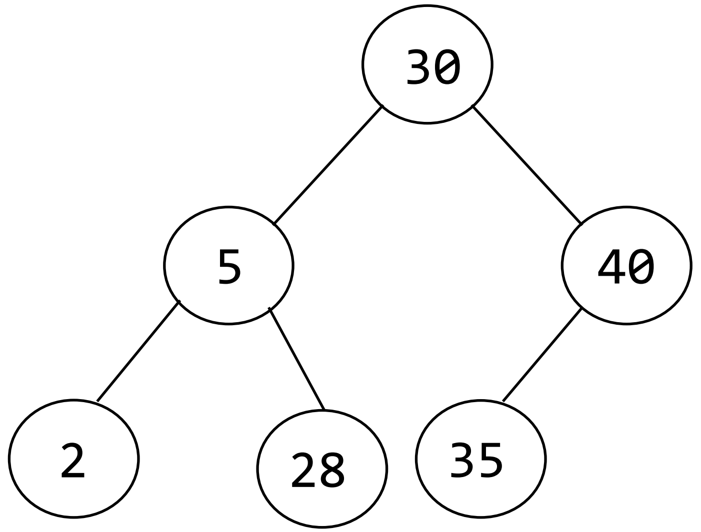
Features of BST

- Searching, insertion, deletion is bounded by O(*h*) where *h* is the height of the BST
- These operations can be performed both
  - by key value

    e.g.) delete the element with key x
  - by rank

    e.g.) delete the fifth smallest element
- Inorder traversal of BST generates a sorted list

```
Inorder traversal

    2   5   28   30   35   40
```



Searching a Binary Search Tree

```
element* search(treePointer root, int k)
{         /* return a pointer to the element whose key is k,
             if there is no such element, return NULL */

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

Recursive search

```
typedef struct element {
   int key;
};

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

```
element* iterSearch(treePointer tree, int k)
{         /* return a pointer to the element whose key is k,
             if there is no such element, return NULL */
          while( tree ) {
                   if ( k == tree->data.key ) return &(tree->data);
                   if ( k < tree->data.key )
                            tree = tree->leftChild;
                   else
                            tree = tree->rightChild;
          }
          return NULL;
}
```

Iterative search

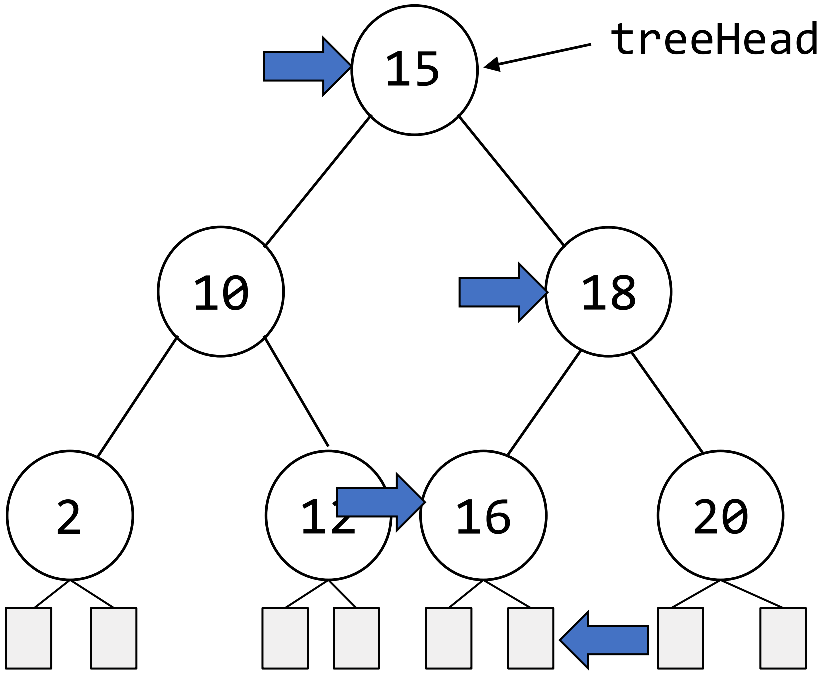
Example

```
treePointer tr = search(treeHead, 17);
```

There is not 17 in this tree

search(, 17)	NULL
search(16, 17)	NULL
search(18, 17)	NULL
search(15, 17)	NULL
main()	NULL

Function call stack

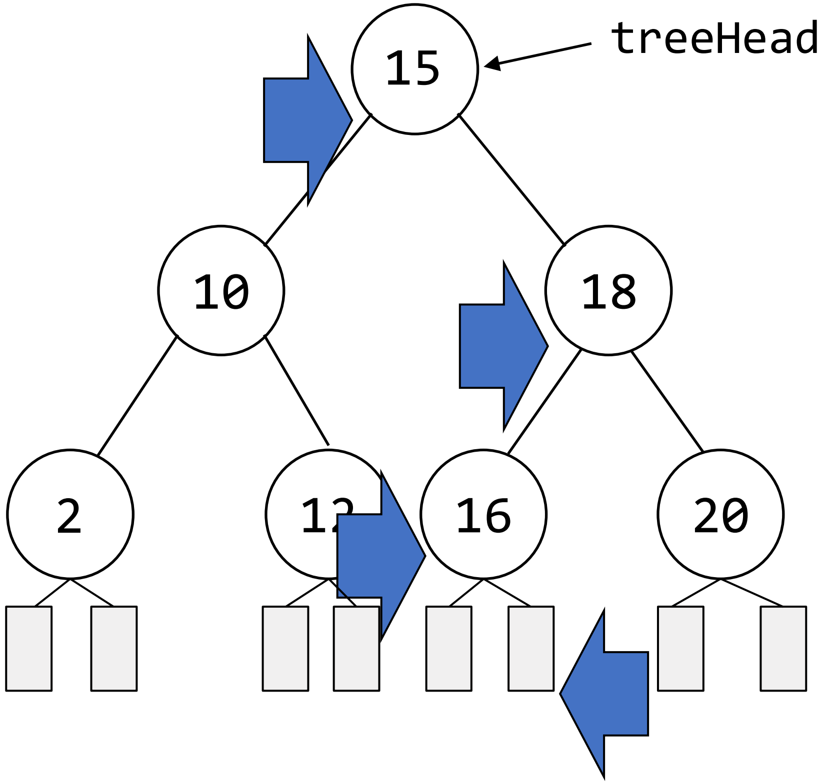



```
element* search(treePointer root, int k)
{          /* return a pointer to the element whose key is k,
              if there is no such element, return NULL */

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

Example

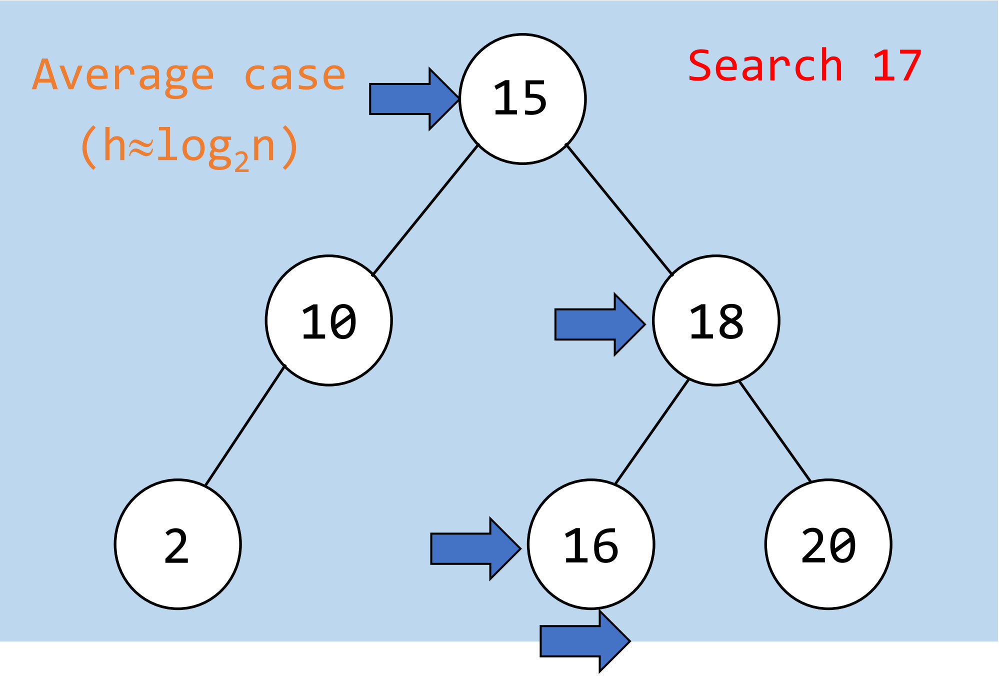
```
treePointer tr = iterSearch(treeHead, 17);
```

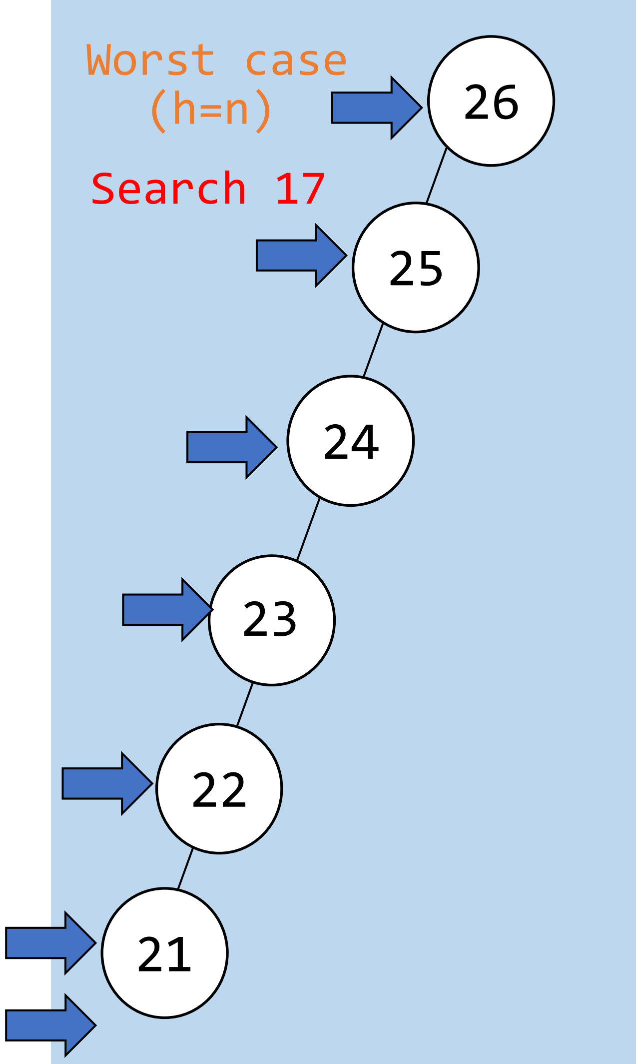
```
element* iterSearch (treePointer tree, int k)
{
   while (tree) {
     if (k == tree->data.key) return &(tree->data);
     if (k < tree->data.key)
        tree = tree->leftChild;
     else
        tree = tree->rightChild;
   } /* while */
   return NULL;
}
```



Time Complexity of Searching BST

- Average case
  O(*h*), where *h* is the height of BST
- Worst case
  O(*n*), where *n* is the number of nodes



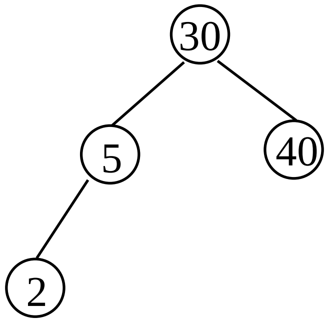


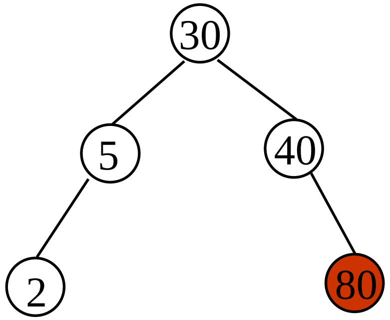
Inserting into A Binary Search Tree

❖ Insert a key 80 into the tree

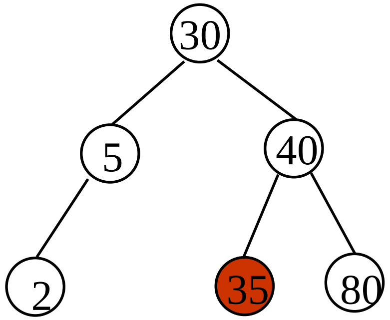
- First search the tree for 80
- The last node examined has a key 40
- Insert the key 80 as the right child of the node

❖ Insert a key 35 into the resulting tree





Insert 80



Insert 35

Insertion into A Binary Search Tree

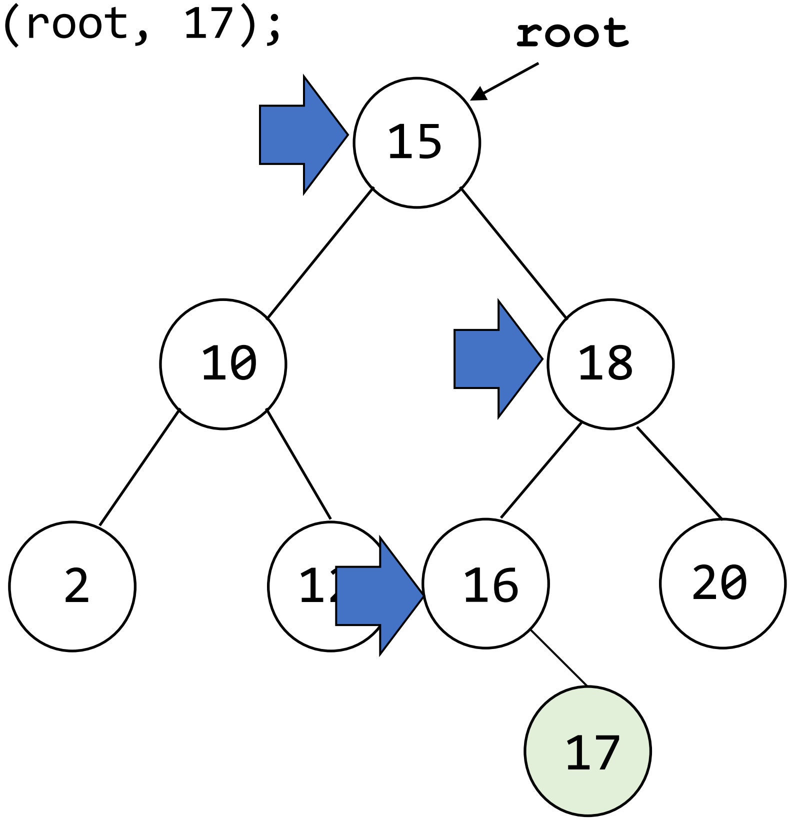
```
void insert (treePointer *node, int k)
{       /* if k is in the tree pointed at by node do nothing;
            otherwise add a new node with data =(k) */
        treePointer ptr;
         /* searches the binary search tree *node for the key k */
        treePointer temp = modifiedSearch( *node, k );
        if ( temp || !(*node) ) {
                /* k is not in the tree */
                MALLOC( ptr, sizeof(*ptr) );
                ptr->data.key = k;
                ptr->leftChild = prt->rightChild = NULL;

                if ( *node ) /* insert as child of temp */
                        if (k<temp->data.key)
                                temp->leftChild = ptr;
                        else
                                temp->rightChild = ptr;
                else
                        *node = ptr;
        }
}
```

modifiedSearch():
if (a tree is empty || k is in the tree)
        return *NULL*
else
        return the pointer of the last examined node during the search.

Insertion into A Binary Search Tree

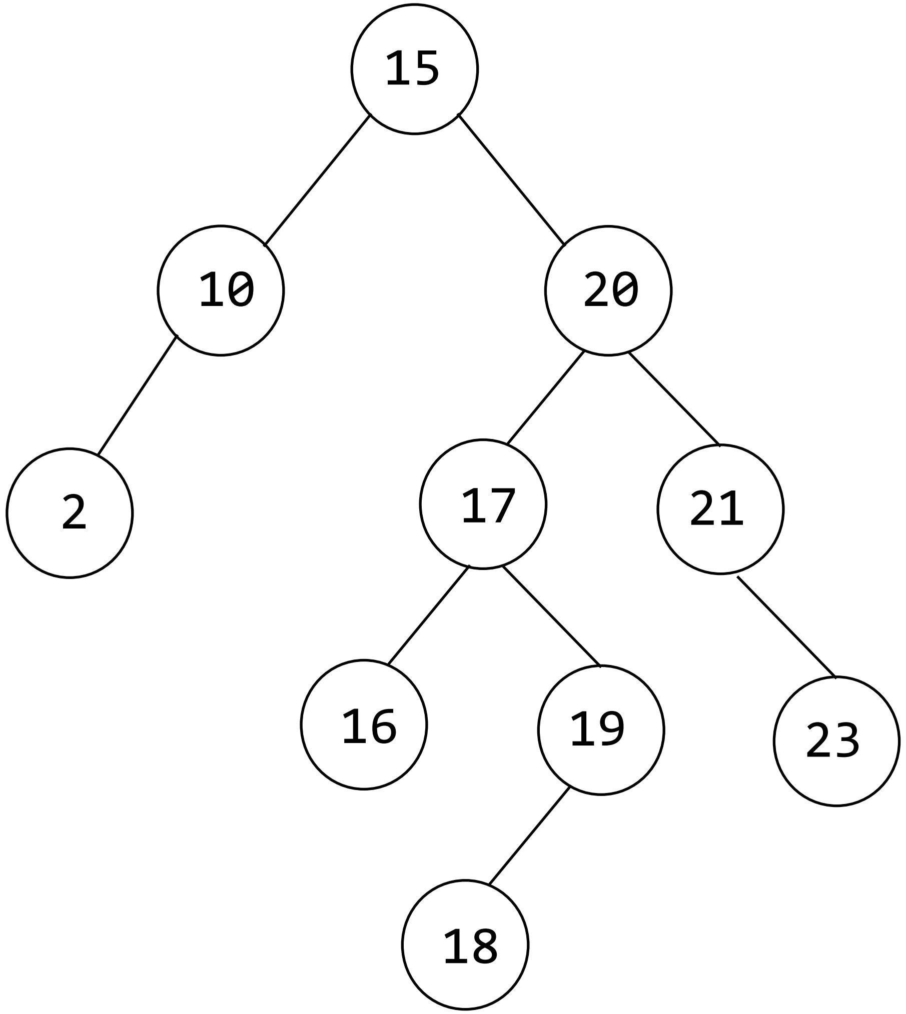
```
void insert(treePointer *node, int k) {
   treePointer ptr, temp = modifiedSearch(*node, k);
   if (temp || !(*node)) {
      MALLOC(ptr, sizeof (*ptr));
      ptr->data.key = k;
     ptr->leftChild = ptr->rightChild = NULL;
      if (*node) {
          if (k < temp->data.key) {
             temp->leftChild = ptr;
          } else {
             temp->rightChild = ptr; }
      } else { *node = ptr; }
   }
}
```

`insert(root, 17);`

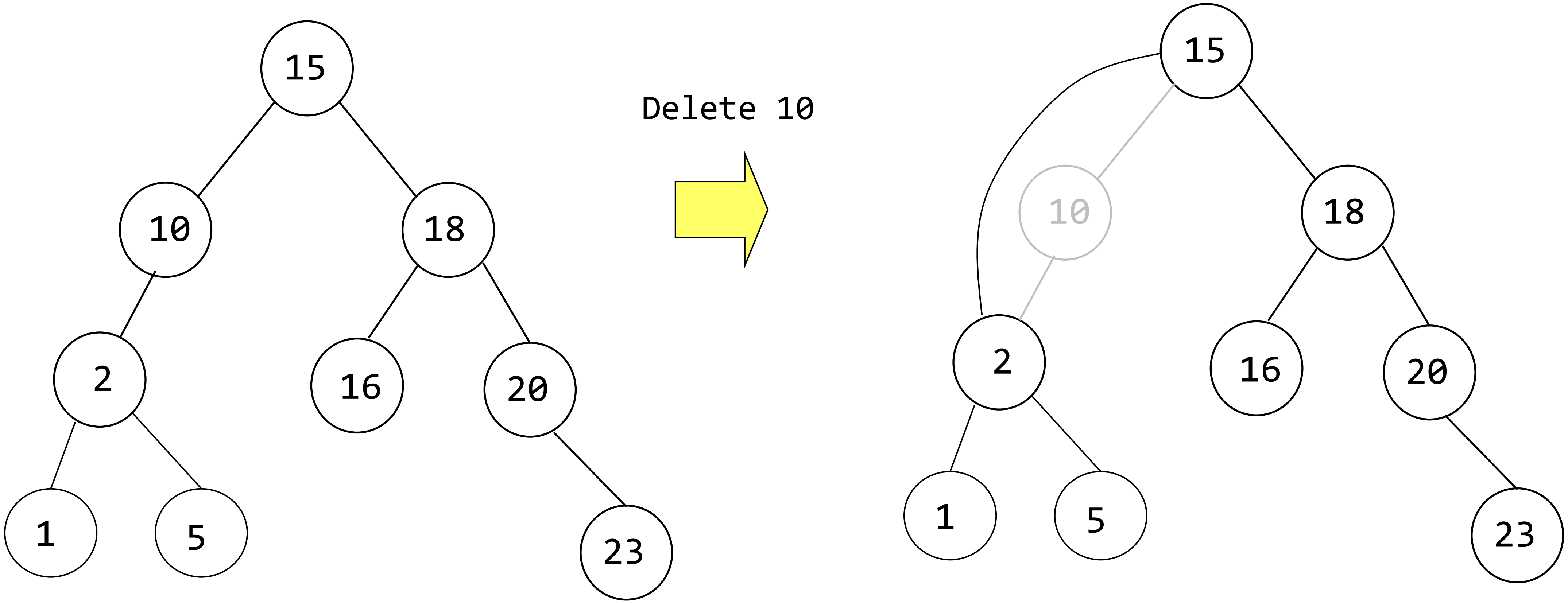


Deletion from A Binary Search Tree

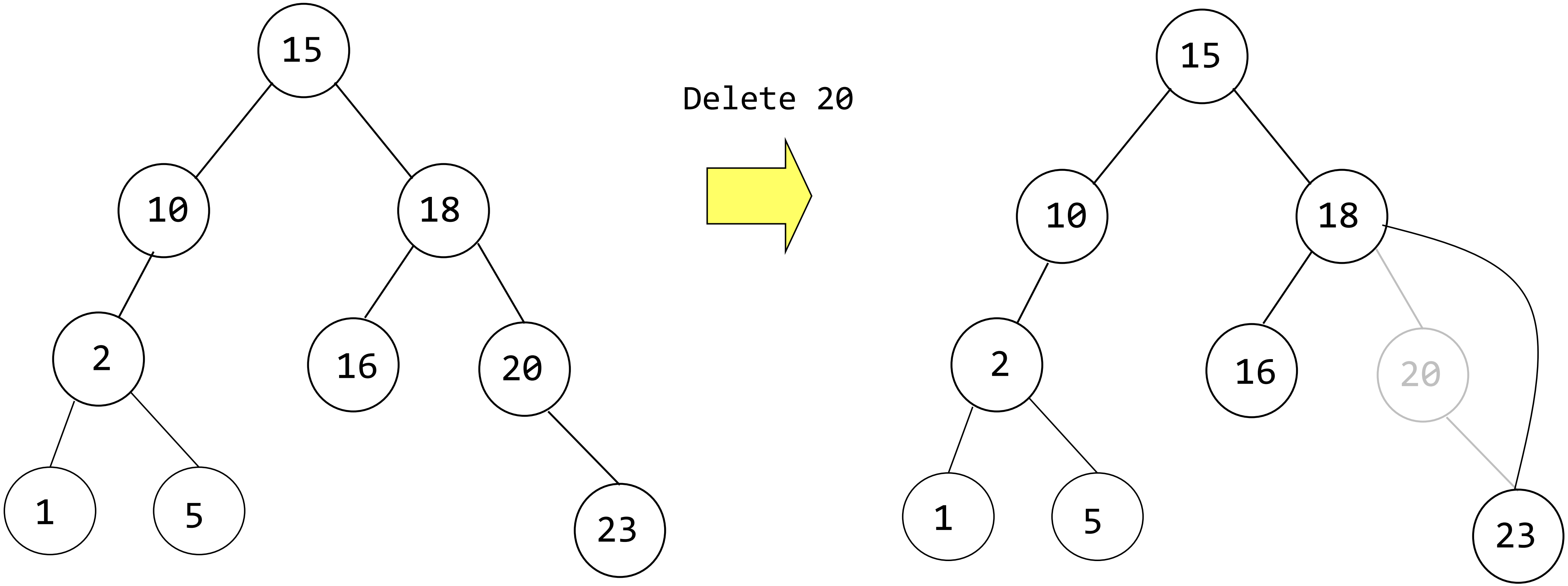
- deletion of a leaf node
- deletion of a node with 1 child
- deletion of a node with 2 children



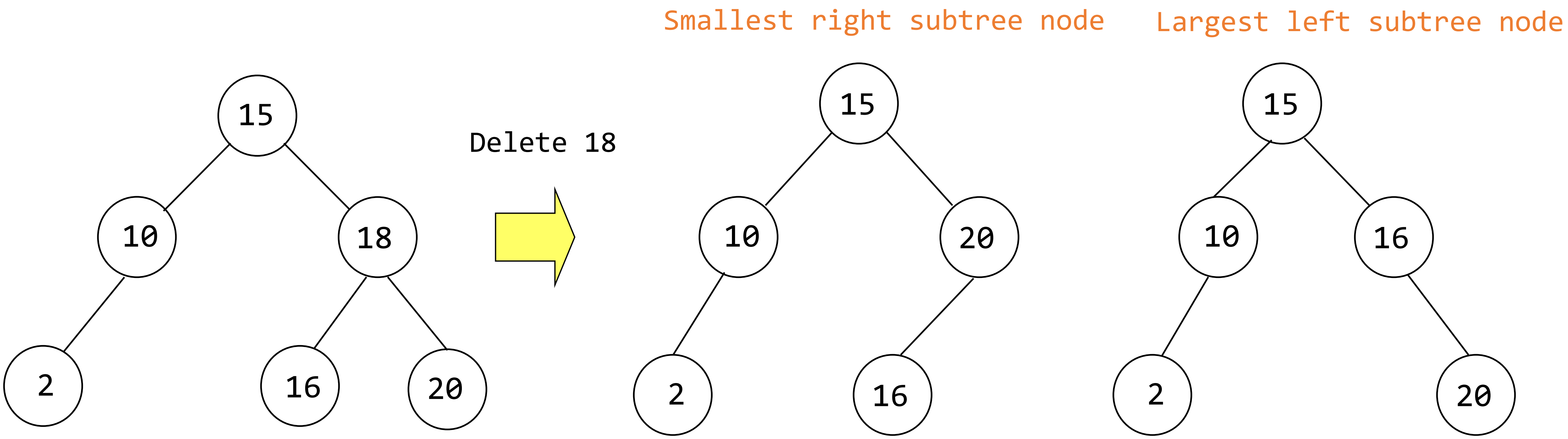
Deleting a Node with 1 Child Node



Deleting a Node with 1 Child Node

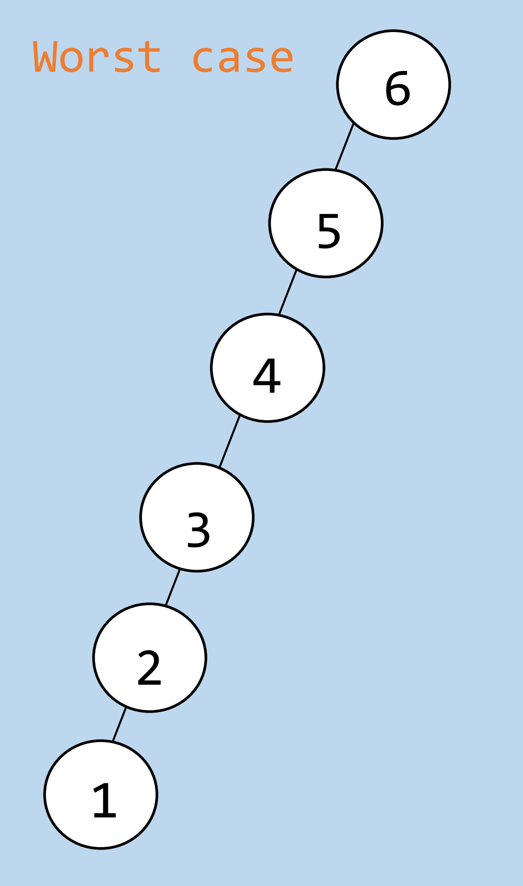


Deleting a Node with 2 Child Nodes



→ Time complexity : O(*h*), where *h* is the height of BST

Height of BST



❖ The height of a BST with n elements

- average case: $O(\log_2 n)$
- worst case: $O(n)$

e.g.) Use insert() to insert the keys 1,2,3,…, n into an initially empty BST

❖ Balanced Search Trees

- Worst case height : $O(\log_2 n)$
- Searching, insertion, deletion are bounded by $O(h)$, where h is the height of a binary tree

e.g.) AVL tree, 2-3 tree, red-black tree

Next Topic

Graphs