

Chapter

7

*Programming with
Recursion*



What Is Recursion?

- **Recursive call** A method call in which the method being called is the same as the one making the call
- **Direct recursion** Recursion in which a method directly calls itself
- **Indirect recursion** Recursion in which a chain of two or more method calls returns to the method that originated the chain



Recursion

- You must be careful when using recursion.
- Recursive solutions can be less efficient than iterative solutions.
- Still, many problems lend themselves to simple, elegant, recursive solutions.



Some Definitions

- **Base case** The case for which the solution can be stated nonrecursively
- **General (recursive) case** The case for which the solution is expressed in terms of a smaller version of itself
- **Recursive algorithm** A solution that is expressed in terms of (a) smaller instances of itself and (b) a base case



Recursive Function Call

- A **recursive call** is a function call in which the called function is the same as the one making the call.
- In other words, *recursion occurs when a function calls itself!*
- We must avoid making an infinite sequence of function calls (infinite recursion).



Finding a Recursive Solution

- Each successive recursive call should bring you closer to a situation in which the answer is known.
- A case for which the answer is known (and can be expressed without recursion) is called a **base case**.
- Each recursive algorithm must have at least one base case, as well as the **general (recursive) case**

General format for many recursive functions

if (some condition for which answer is known)

// base case

solution statement

else

// general case

recursive function call

SOME EXAMPLES . . .

Writing a recursive function to find n factorial

DISCUSSION

The function call Factorial(4) should have value 24, because that is $4 * 3 * 2 * 1$.

For a situation in which the answer is known, the value of $0!$ is 1.

So our **base case** could be along the lines of

```
if ( number == 0 )  
    return 1;
```

Writing a recursive function to find Factorial(n)

Now for the **general case** . . .

The value of **Factorial(n)** can be written as
n * the product of the numbers from (n - 1) to 1,
that is,

$$n * (n - 1) * \dots * 1$$

or, **n * Factorial(n - 1)**

And notice that the recursive call **Factorial(n - 1)**
gets us “closer” to the base case of **Factorial(0)**.

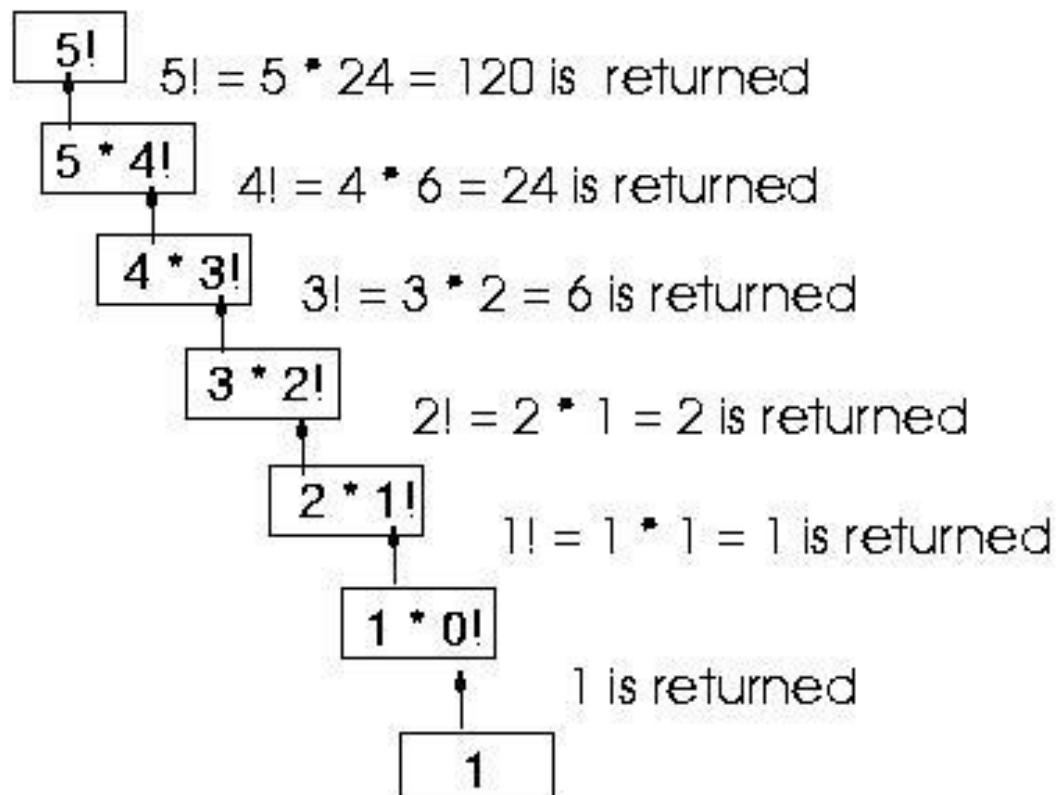
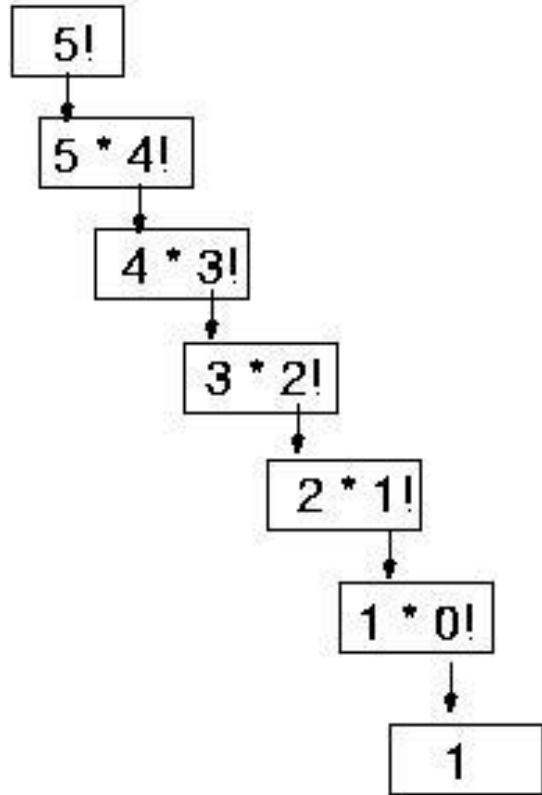


Recursive Solution

```
int Factorial ( int number )
// Pre: number is assigned and number >= 0.
{
    if ( number == 0)                                // base case
        return 1 ;
    else                                              // general case
        return number + Factorial ( number - 1 ) ;
}
```



Final value = 120





Another example: n choose k (combinations)

- Given n things, how many different sets of size k can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \quad (\text{recursive solution})$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \quad (\text{closed-form solution})$$

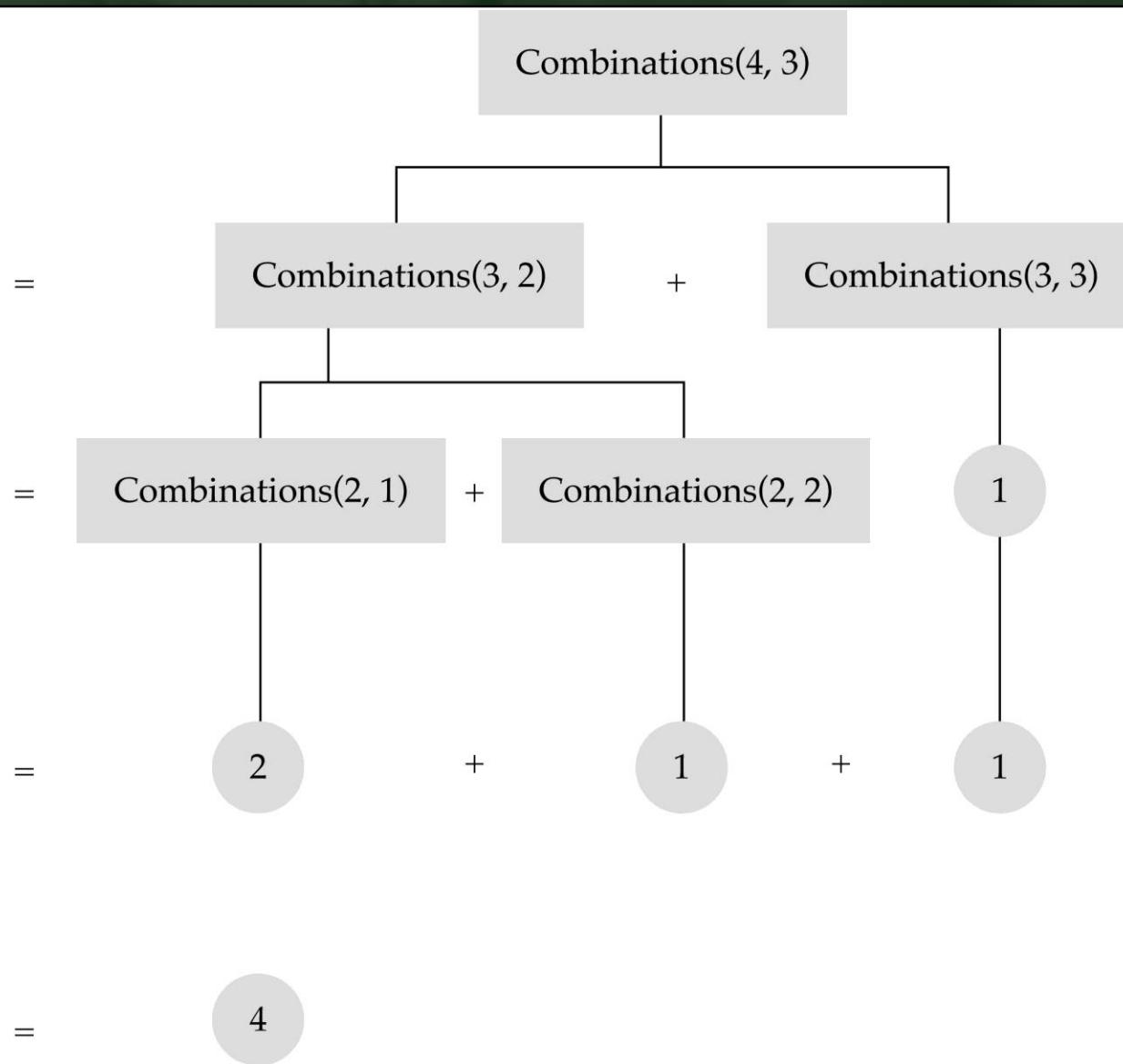
with base cases:

$$\binom{n}{1} = n \quad (k = 1), \quad \binom{n}{n} = 1 \quad (k = n)$$



n choose k (combinations)

```
int Combinations(int n, int k)
{
    if(k == 1) // base case 1
        return n;
    else if (n == k) // base case 2
        return 1;
    else
        return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```





Three-Question Method of verifying recursive functions

- **Base-Case Question:** Is there a nonrecursive way out of the function?
- **Smaller-Caller Question:** Does each recursive function call involve a smaller case of the original problem leading to the base case?
- **General-Case Question:** Assuming each recursive call works correctly, does the whole function work correctly?



Another example where recursion comes naturally

- From mathematics, we know that

$$2^0 = 1 \quad \text{and} \quad 2^5 = 2 * 2^4$$

- In general,

$$x^0 = 1 \quad \text{and} \quad x^n = x * x^{n-1}$$

for integer x , and integer $n > 0$.

- Here we are defining x^n recursively, in terms of x^{n-1}



```
// Recursive definition of power function

int Power ( int x, int n )

    // Pre:      n >= 0.  x, n are not both zero
    // Post:     Function value = x raised to the power n.

{
    if ( n == 0 )
        return 1;                      // base case
    else                         // general case
        return ( x * Power ( x , n-1 ) ) ;
}
```

Of course, an alternative would have been to use looping instead of a recursive call in the function body.



struct ListType

```
struct ListType
{
    int length ;      // number of elements in the list

    int info[ MAX_ITEMS ] ;

} ;

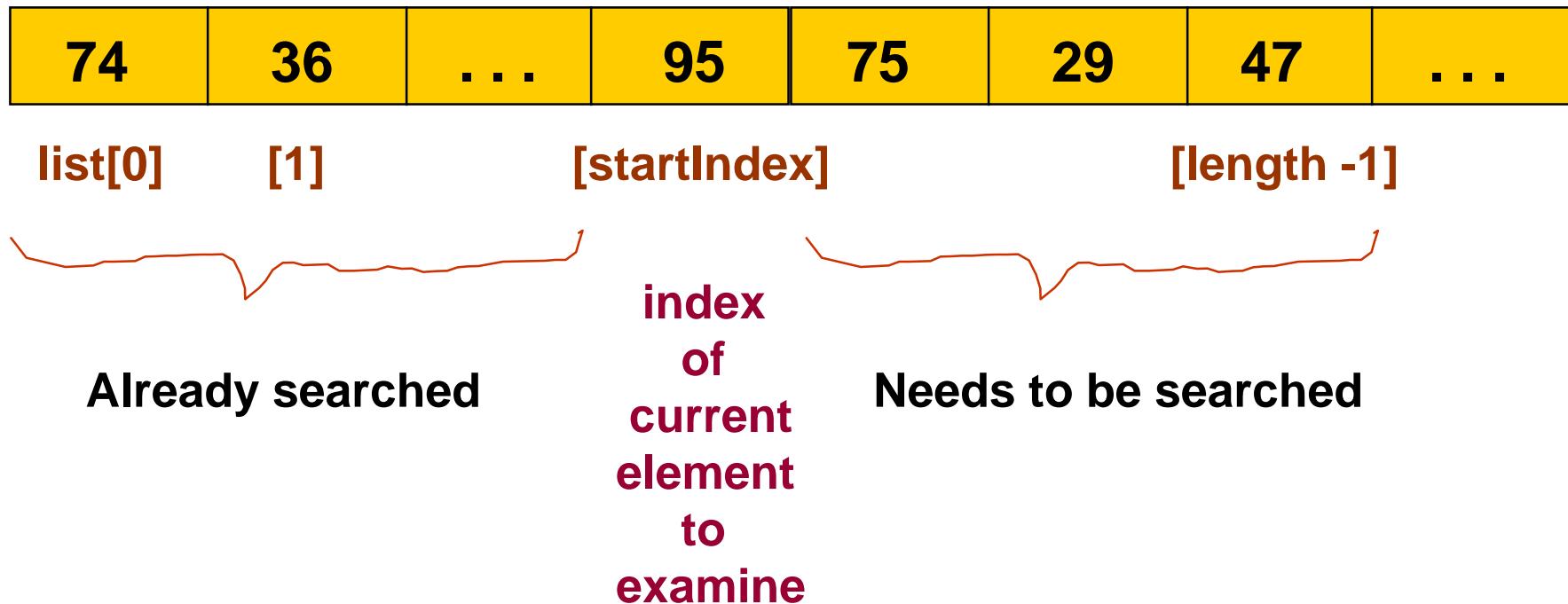
ListType list ;
```



Recursive function to determine if value is in list

PROTOTYPE

```
bool ValueInList( ListType list , int value , int startIndex ) ;
```





```
bool ValueInList ( ListType  list , int  value, int startIndex )  
  
// Searches list for value between positions startIndex  
// and list.length-1  
// Pre: list.info[ startIndex ] . . list.info[ list.length - 1 ]  
//      contain values to be searched  
// Post: Function value =  
//      ( value exists in list.info[ startIndex ] . .  
//      list.info[ list.length - 1 ] )  
{  
    if  ( list.info[startIndex] == value )      // one base case  
        return  true ;  
    else  if  (startIndex == list.length -1 ) // another base case  
        return  false ;  
    else                                // general case  
        return ValueInList( list, value, startIndex + 1 ) ;  
}
```



“Why use recursion?”

Those examples could have been written without recursion, using iteration instead. The iterative solution uses a loop, and the recursive solution uses an if statement.

However, for certain problems the recursive solution is the most natural solution. This often occurs when pointer variables are used.



struct ListType

```
struct NodeType
{
    int info ;
    NodeType* next ;
}

class SortedType
{
public :

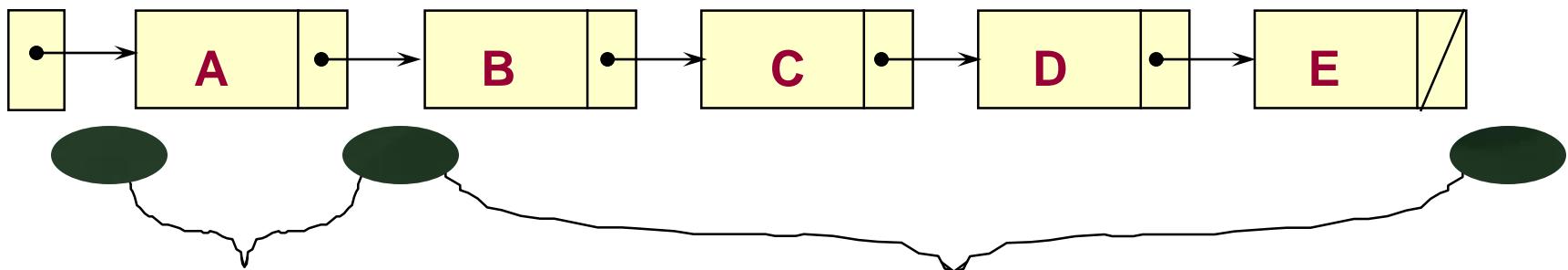
    . . . . . // member function prototypes

private :
    NodeType* listData ;
} ;
```



RevPrint(listData) ;

listData



FIRST, print out this section of list, backwards

**THEN, print
this element**



Base Case and General Case

A base case may be a solution in terms of a “smaller” list. Certainly for a list with 0 elements, there is no more processing to do.

Our general case needs to bring us closer to the base case situation. That is, the number of list elements to be processed decreases by 1 with each recursive call. By printing one element in the general case, and also processing the smaller remaining list, we will eventually reach the situation where 0 list elements are left to be processed.

In the general case, we will print the elements of the smaller remaining list in reverse order, and then print the current pointed to element.



Using recursion with a linked list

```
void    RevPrint ( NodeType*  listPtr )  
  
// Pre: listPtr points to an element of a list.  
// Post: all elements of list pointed to by listPtr  
// have been printed out in reverse order.  
{  
    if  ( listPtr != NULL )           // general case  
    {  
        RevPrint ( listPtr->next ) ; //process the rest  
        std::cout << listPtr->info << std::endl ;  
                           // print this element  
    }  
    // Base case : if the list is empty, do nothing  
}
```



Function BinarySearch()

- **BinarySearch takes sorted array info, and two subscripts, fromLoc and toLoc, and item as arguments.** It returns false if item is not found in the elements `info[fromLoc...toLoc]`. Otherwise, it returns true.
- **BinarySearch can be written using iteration, or using recursion.**



```
found = BinarySearch(info, 25, 0, 14);
```

item fromLoc toLoc

indexes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----

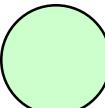
info

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

16 18 20 22 24 26 28

24 26 28

24

NOTE:  denotes element examined



Non-recursive implementation

```
template<class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
    int midPoint;
    int first = 0;
    int last = length - 1;

    found = false;
    while( (first <= last) && !found) {
        midPoint = (first + last) / 2;
        if (item < info[midPoint])
            last = midPoint - 1;
        else if(item > info[midPoint])
            first = midPoint + 1;
        else {
            found = true;
            item = info[midPoint];
        }
    }
}
```



Recursive binary search

- What is the *size factor*?
The number of elements in (*info[first] ... info[last]*)
- What is the *base case(s)*?
 - (1) If *first > last*, return *false*
 - (2) If *item==info[midPoint]*, return *true*
- What is the *general case*?
if item < info[midPoint] search the first half
if item > info[midPoint], search the second half



```
template<class ItemType>
bool BinarySearch ( ItemType info[ ] , ItemType item ,
                    int fromLoc , int toLoc )
// Pre: info [ fromLoc . . toLoc ] sorted in ascending order
// Post: Function value = ( item in info [ fromLoc .. toLoc] )

{ int mid ;
  if ( fromLoc > toLoc )      // base case -- not found
      return false ;
  else {
    mid = ( fromLoc + toLoc ) / 2 ;
    if ( info [ mid ] == item ) //base case-- found at mi
      return true ;
    else if ( item < info [ mid ] ) // search lower half
      return BinarySearch ( info, item, fromLoc, mid-1 ) ;
            else                  // search upper half
      return BinarySearch( info, item, mid + 1, toLoc ) ;
  }
}
```



When a function is called...

- A **transfer of control** occurs from the calling block to the code of the function. It is necessary that there be a return to the correct place in the calling block after the function code is executed. This correct place is called the **return address**.
- When any function is called, the **run-time stack** is used. On this stack is placed an **activation record (stack frame)** for the function call.



Stack Activation Frames

- The **activation record** stores the return address for this function call, and also the parameters, local variables, and the function's return value, if non-void.
- The activation record for a particular function call is **popped off the run-time stack** when the final closing brace in the function code is reached, or when a return statement is reached in the function code.
- At this time the function's return value, if non-void, is brought back to the calling block return address for use there.



```
// Another recursive function
int Func ( int a, int b )

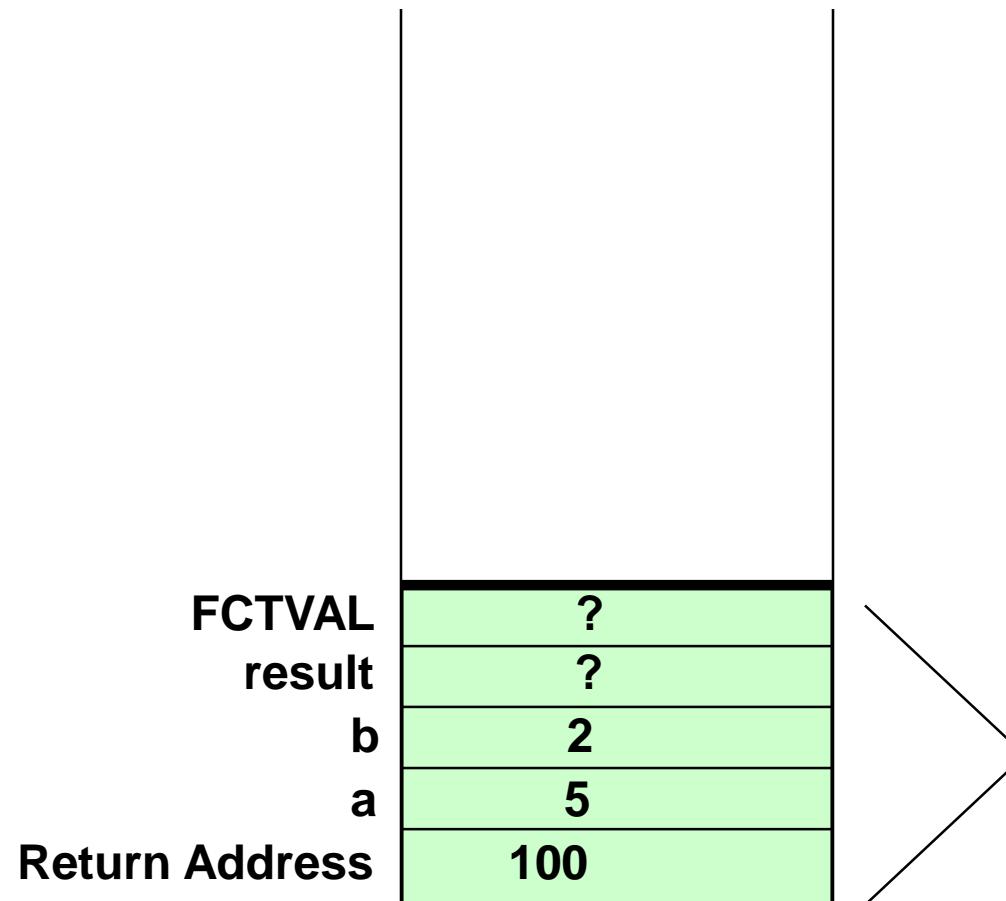
    // Pre:    a and b have been assigned values
    // Post:   Function value = ??


{
    int result;
    if ( b == 0 )                                // base case
        result = 0;
    else if ( b > 0 )                            // first general case
        result = a + Func ( a , b - 1 ) ) ;      // instruction 50
    else                                         // second general case
        result = Func ( - a , - b ) ;            // instruction 70
    return result;
}
```



Run-Time Stack Activation Records

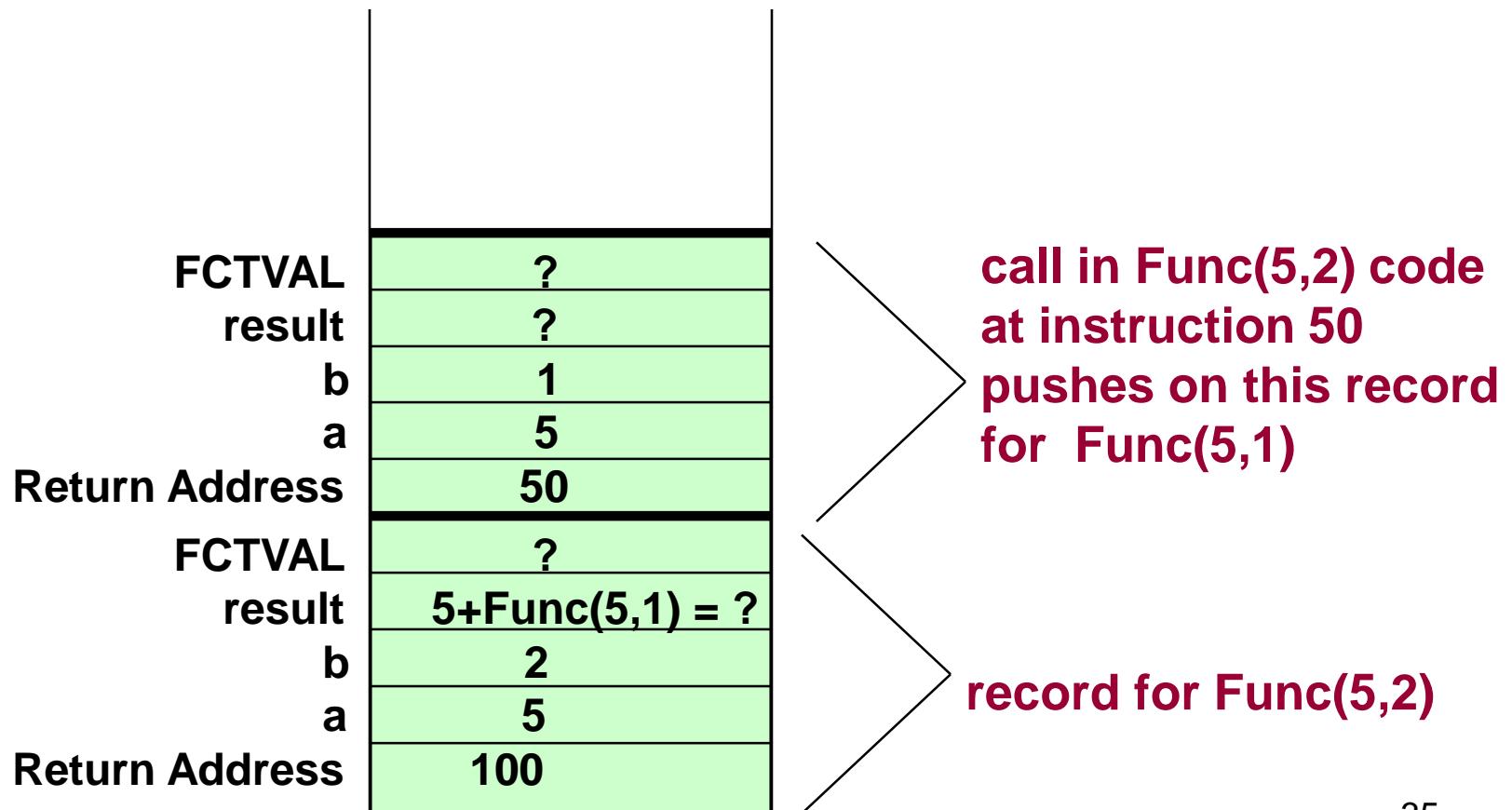
```
x = Func(5, 2); // original call is instruction 100
```



original call
at instruction 100
pushes on this record
for Func(5,2)

Run-Time Stack Activation Records

`x = Func(5, 2); // original call at instruction 100`





Run-Time Stack Activation Records

```
x = Func(5, 2); // original call at instruction 100
```

	FCTVAL	?
	result	?
	b	0
	a	5
Return Address		50
	FCTVAL	?
	result	$5+Func(5,0) = ?$
	b	1
	a	5
Return Address		50
	FCTVAL	?
	result	$5+Func(5,1) = ?$
	b	2
	a	5
Return Address		100

call in Func(5,1) code
at instruction 50
pushes on this record
for Func(5,0)

record for Func(5,1)

record for Func(5,2)

Run-Time Stack Activation Records

```
x = Func(5, 2); // original call at instruction 100
```

FCTVAL	0
result	0
b	0
a	5
Return Address	50
FCTVAL	?
result	$5+Func(5,0) = ?$
b	1
a	5
Return Address	50
FCTVAL	?
result	$5+Func(5,1) = ?$
b	2
a	5
Return Address	100

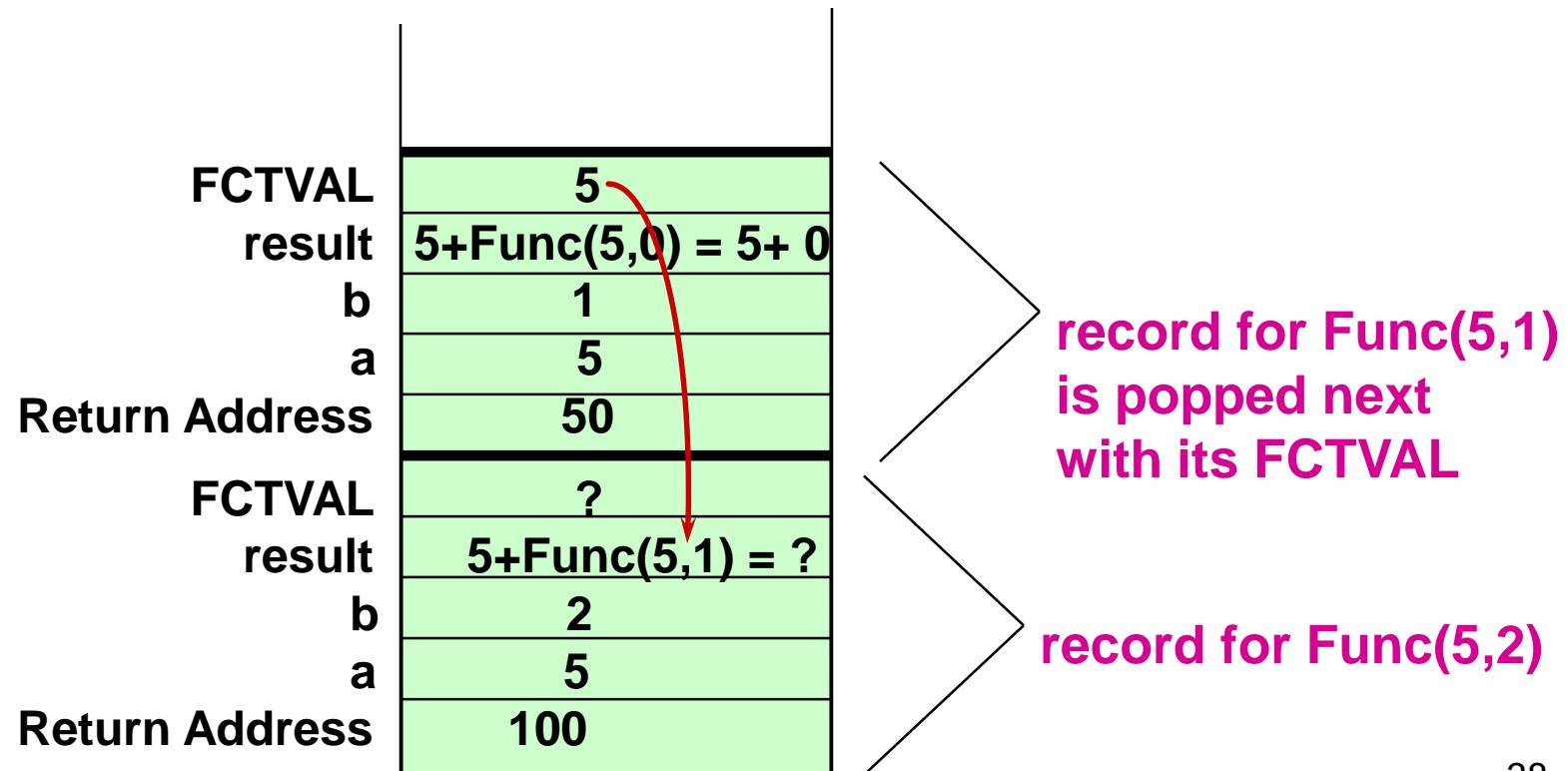
record for Func(5,0)
is popped first
with its FCTVAL

record for Func(5,1)

record for Func(5,2)

Run-Time Stack Activation Records

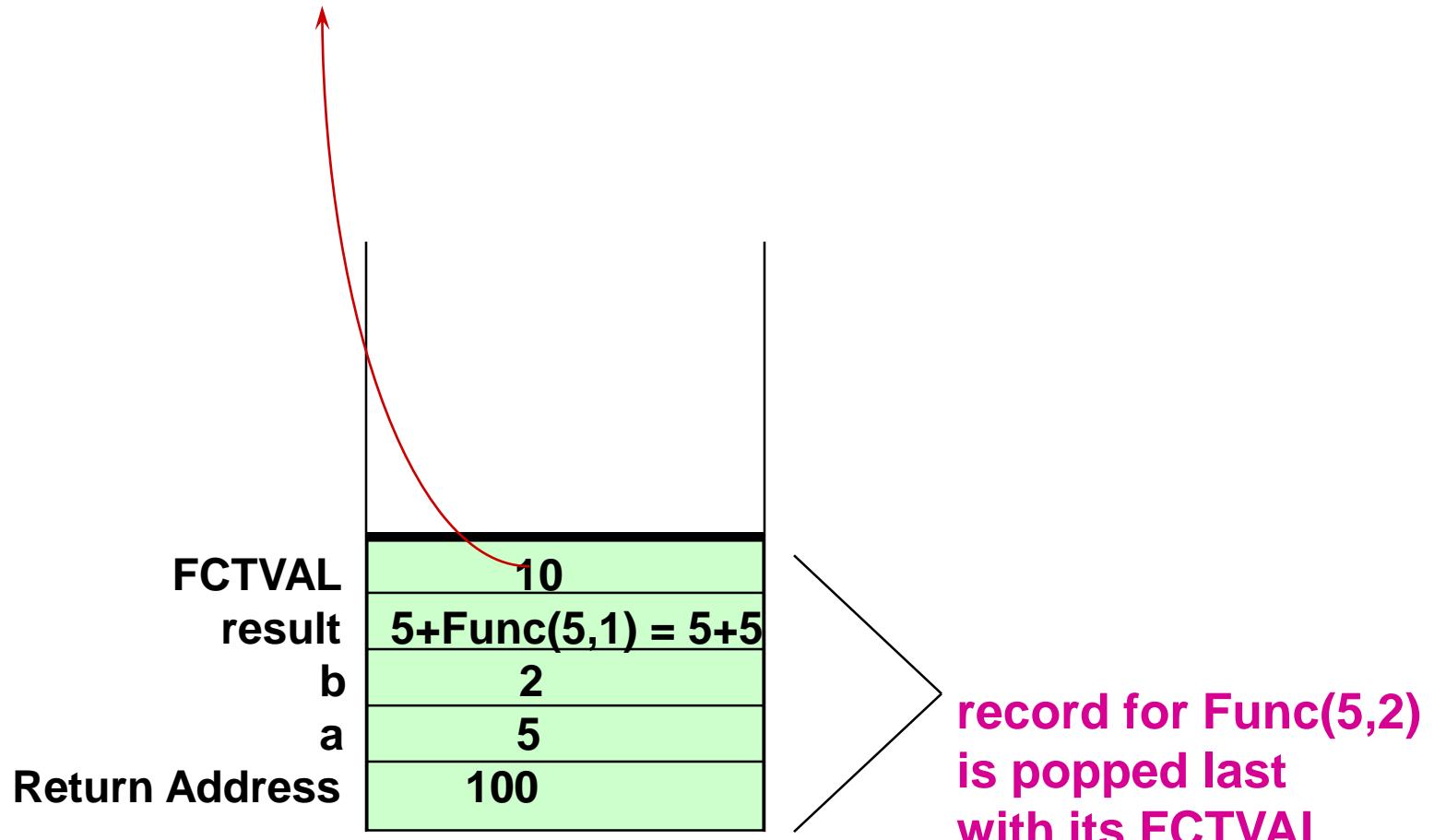
```
x = Func(5, 2);           // original call at instruction 100
```





Run-Time Stack Activation Records

```
x = Func(5, 2);           // original call at line 100
```



Show Activation Records for these calls

x = Func(- 5, - 3);

x = Func(5, - 3);

What operation does Func(a, b) simulate?



Recursive InsertItem (sorted list)

- What is the *size factor*?

The number of elements in the current list

What is the *base case(s)*?

- 1) If the list is empty, insert item into the empty list
- 2) If $item < location->info$, insert item as the first node in the current list

- What is the *general case*?

$Insert(location->next, item)$

Recursive InsertItem (sorted list)

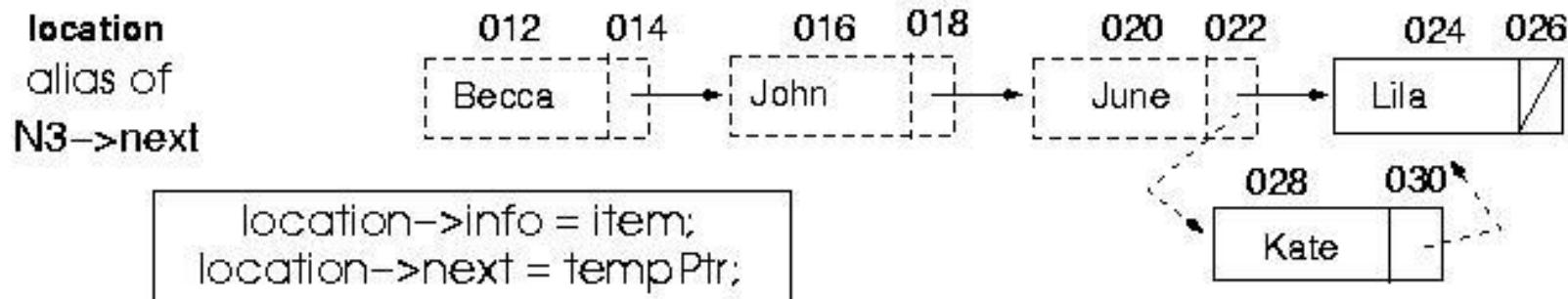
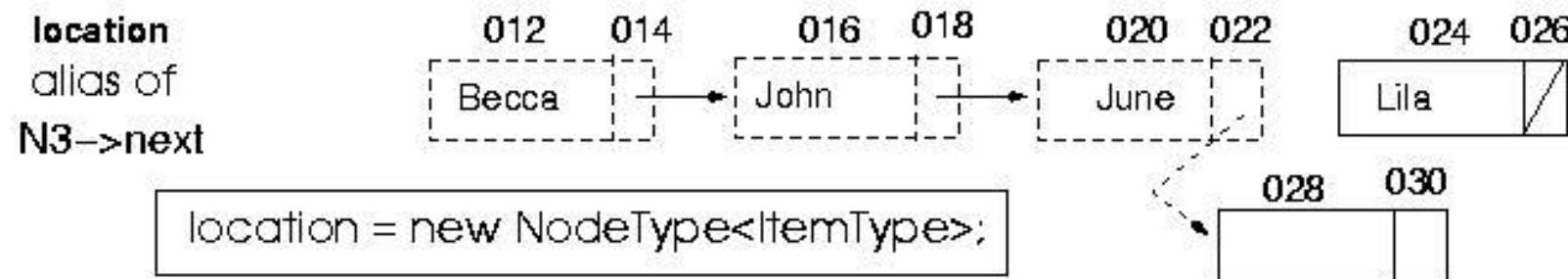
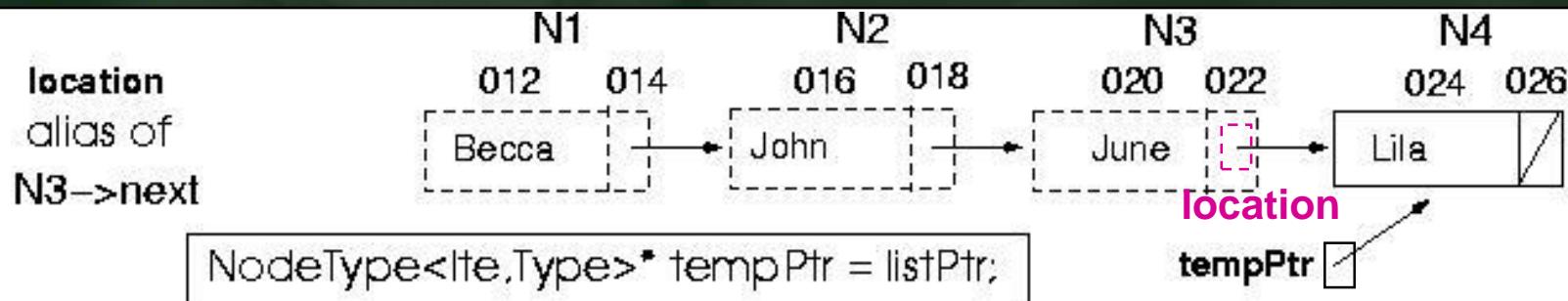
```
template <class ItemType>
void Insert(NodeType<ItemType>*&location, ItemType item)
{
    if(location == NULL) || (item < location->info)) { // base cases

        NodeType<ItemType>* tempPtr = location;
        location = new NodeType<ItemType>;
        location->info = item;
        location->next = tempPtr;
    }
    else
        Insert(location->next, newItem); // general case
}
```

```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType newItem)
{
    Insert(listData, newItem);
}
```



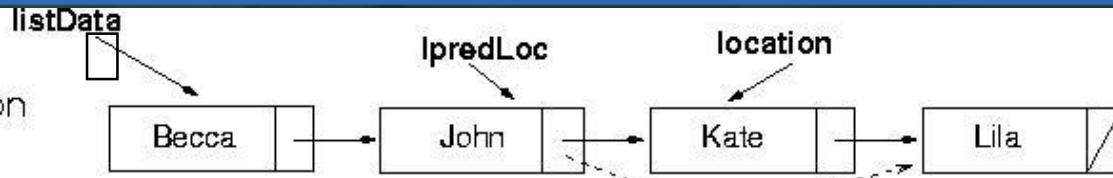
- No "predLoc" pointer is needed for insertion



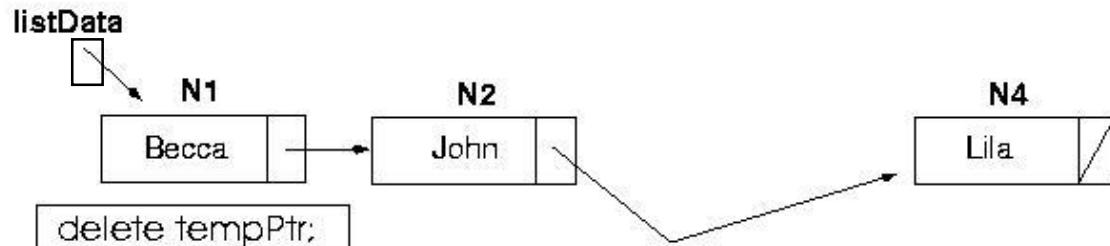
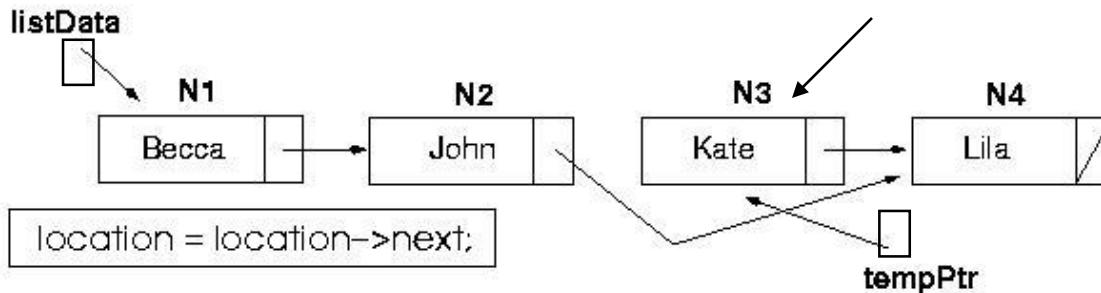
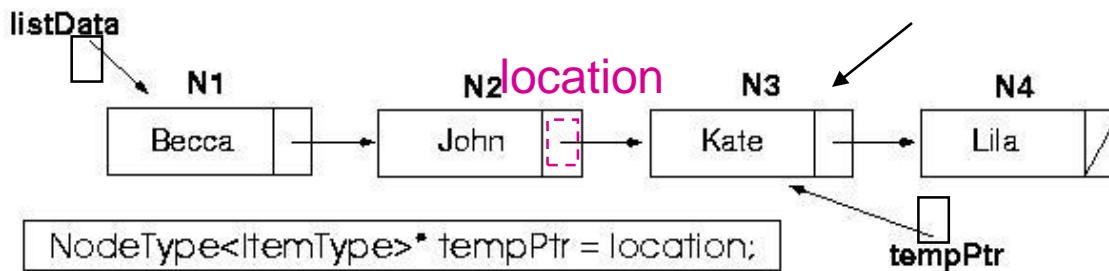


Recursive DeleteItem (sorted list)

Previous
Implementation



location
alias of
 $N2 \rightarrow \text{next}$





Recursive DeleteItem (sorted list)

- What is the *size factor*?
The number of elements in the list
- What is the *base case(s)*?
If *item == location->info*, delete node pointed by *location*
- What is the *general case*?
Delete(location->next, item)



Recursive DeleteItem (sorted list)

```
template <class ItemType>
void Delete(NodeType<ItemType>*& location, ItemType item)
{
    if(item == location->info) {

        NodeType<ItemType>* tempPtr = location;
        location = location->next;
        delete tempPtr;
    }
    else
        Delete(location->next, item);
}
```

```
template <class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item)
{
    Delete(listData, item);
}
```



Tail Recursion

- The case in which a function contains only a single recursive call and it is the last statement to be executed in the function.
- Tail recursion can be replaced by iteration to remove recursion from the solution as in the next example.

// USES TAIL RECURSION

```
bool ValueInList ( ListType  list , int  value , int  startIndex  )

// Searches list for value between positions startIndex
// and list.length-1
// Pre:  list.info[ startIndex ] . . . list.info[ list.length - 1 ]
// contain values to be searched
// Post: Function value =
// ( value exists in list.info[ startIndex ] . .
// list.info[ list.length - 1 ] )
{
    if  ( list.info[startIndex] == value )          // one base case
        return  true ;
    else  if  (startIndex == list.length -1 )      // another base case
        return  false ;
    else
        return ValueInList( list, value, startIndex + 1 ) ;
}
```



remove recursion

- The recursive call causes an activation record to put on the run-time stack to hold the function's parameters and local variables
- Because the recursive call is the last statement in the function, the function terminates without using these values
- So we need to change the “smaller-caller” variable(s) on the recursive call’s parameter list and then “jump” back to the beginning of the function. In other words, we need a loop.

 // ITERATIVE SOLUTION

```

bool ValueInList ( ListType list , int value , int startIndex )

// Searches list for value between positions startIndex
// and list.length-1
// Pre: list.info[ startIndex ] . . list.info[ list.length - 1 ]
//       contain values to be searched
// Post: Function value =
//       ( value exists in list.info[ startIndex ] . .
//       list.info[ list.length - 1 ] )
/* in the iterative solution:
the base cases become the terminating conditions of the loop
in the general case each subsequent execution of the loop body processes a
smaller version of the problem; the unsearched part of the list shrinks
with each execution of the loop body because startIndex is incremented */
{   bool found = false ;
    while ( !found && startIndex < list.length ) //it includes both base cases
    {   if ( value == list.info[ startIndex ] )
            found = true ;
        else     startIndex++ ;                                // related to the general case
    }
    return found ;
}

```



Recursive Solution

```
void RevPrint ( NodeType* listPtr )
//The size is the number of elements in the list pointed to by list listPtr.
// Pre: listPtr points to an element of a list.
// Post: all elements of list pointed to by listPtr have been printed
//        out in reverse order.
{
    if ( listPtr != NULL )                                // general case
    {
        RevPrint ( listPtr-> next ) ; // (a)           // process the rest
        std::cout << listPtr->info << std::endl ; // (b) // print this element
    }
    // (c) Base case : if the list is empty, do nothing
}
/* We must keep track of the pointer to each node, until we reach
   the end of the list. Then print the info data member of the last
   node.
```

Next, we back up and print again, back up and print again, and so on until we have printed the 1st list element . The run-time stack keep track of the pointers */

Stacking Technique: When it is the not last statement to be executed in a recursive function

// Non recursive version – stacks: RevPrint()

```
// We must replace the stacking that was done by the system with stacking that is done by the
programmer
// We must keep track of the pointer to each node, until we reach the end of the list. Then print the
info data member of the last node. Next, we back up and print again, back up and print again, and
so on until we have printed the 1st list element
// The stack allows to store pointers and retrieves them in reverse order
#include "Stack3.h"
void ListType::RevPrint() //now it can be a member function, because on longer has parameter
{
    StackType<NodeType*> stack;
    NodeType* listPtr;
    listPtr = listData;

    while (listPtr != NULL) // Put pointers onto the stack.
    {
        stack.Push(listPtr);
        listPtr = listPtr->next;
    }
                                // Retrieve pointers in reverse order and print elements.
    while (!stack.IsEmpty())
    {
        listPtr = stack.Top();
        stack.Pop();
        cout << listPtr->info;
    }
}
```

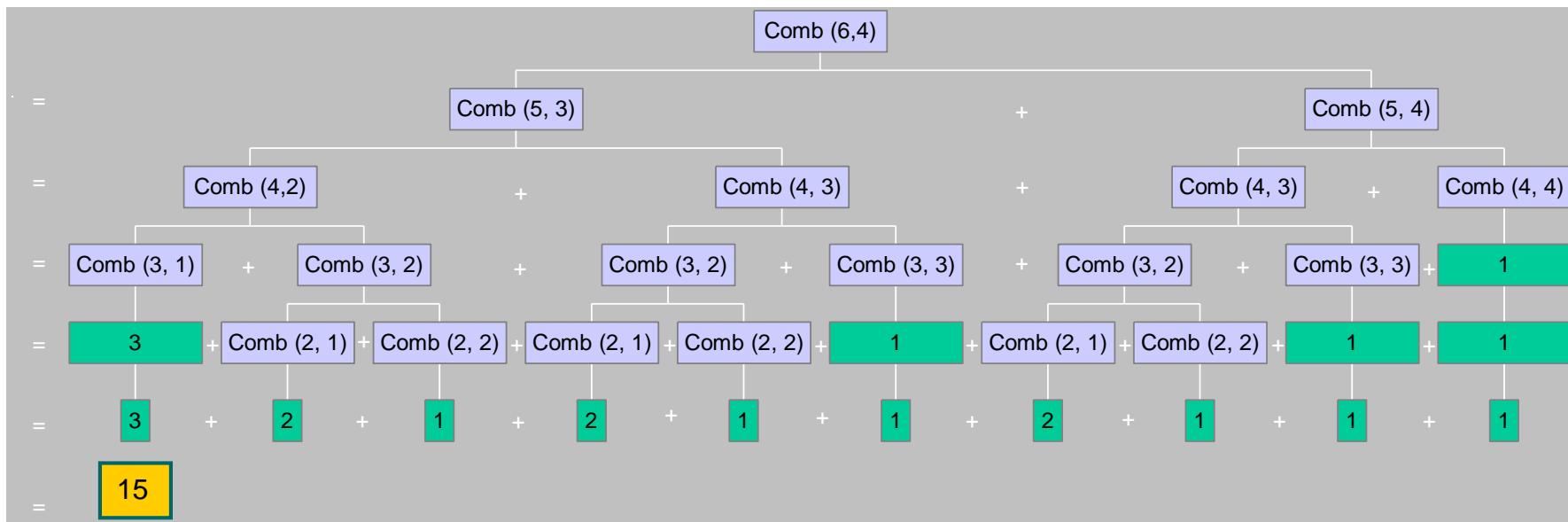


Recursion vs. iteration

- Iteration can be used in place of recursion
 - An iterative algorithm uses a *looping construct*
 - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in *shorter*, more easily understood source code



Recursion can be very inefficient in some cases





Use a recursive solution when:

- The depth of recursive calls is relatively “shallow” compared to the size of the problem.
- The recursive version does about the same amount of work as the nonrecursive version.
- The recursive version is shorter and simpler than the nonrecursive solution.

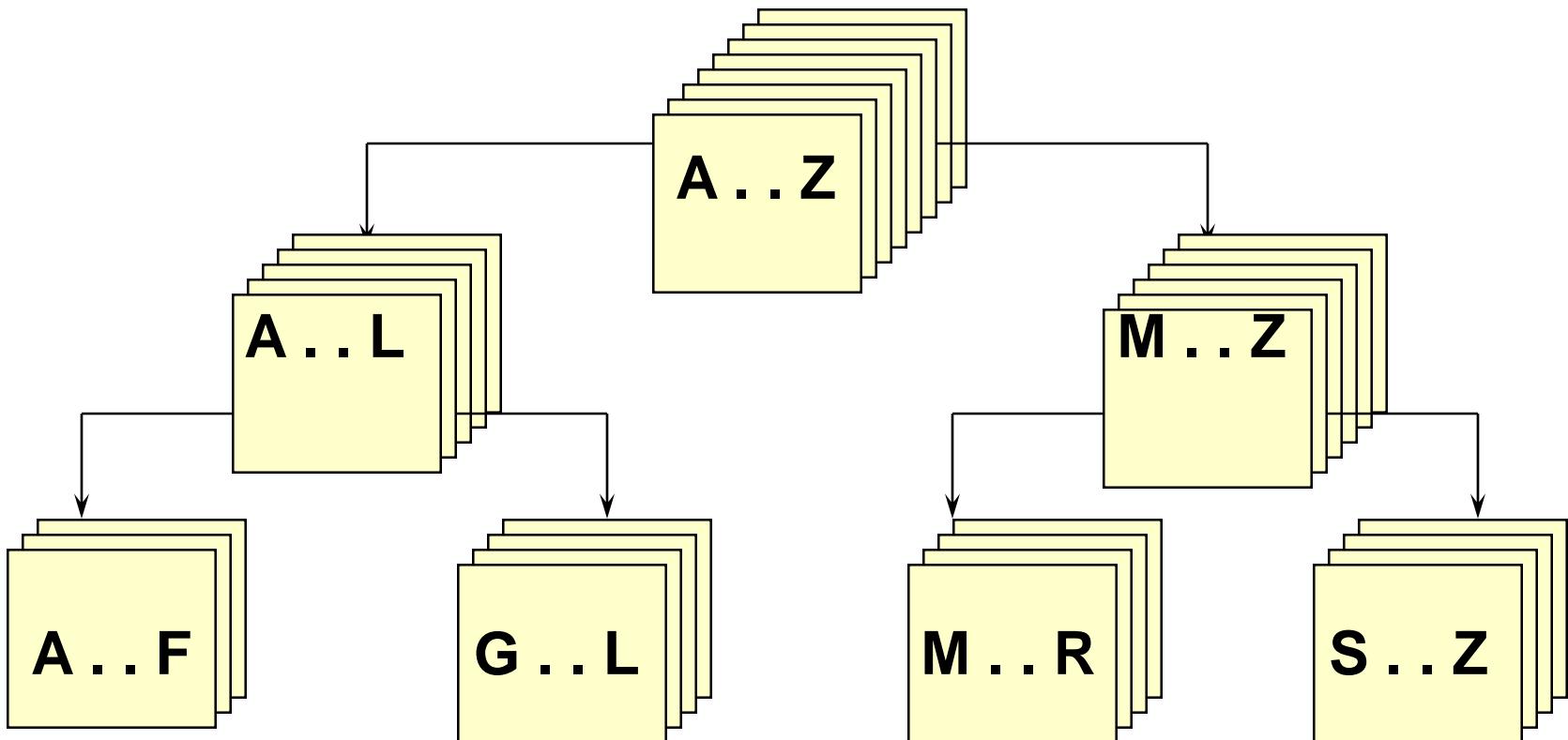
SHALLOW DEPTH

EFFICIENCY

CLARITY



Using quick sort algorithm





Before call to function Split

splitVal = 9

GOAL: place `splitVal` in its proper position with
all values less than or equal to `splitVal` on its left
and all larger values on its right

9	20	6	10	14	3	60	11
---	----	---	----	----	---	----	----

`values[first]`

`[last]`



After call to function Split

splitVal = 9

smaller values

larger values



6	3	9	10	14	20	60	11
---	---	---	----	----	----	----	----

values[first]

[splitPoint]

[last]

```

// Recursive quick sort algorithm

template <class ItemType>
void QuickSort ( ItemType values[ ] , int first, int last )

// Pre: first <= last
// Post: Sorts array values[ first. .last ] into
// ascending order
{
    if ( first < last )                                // general case

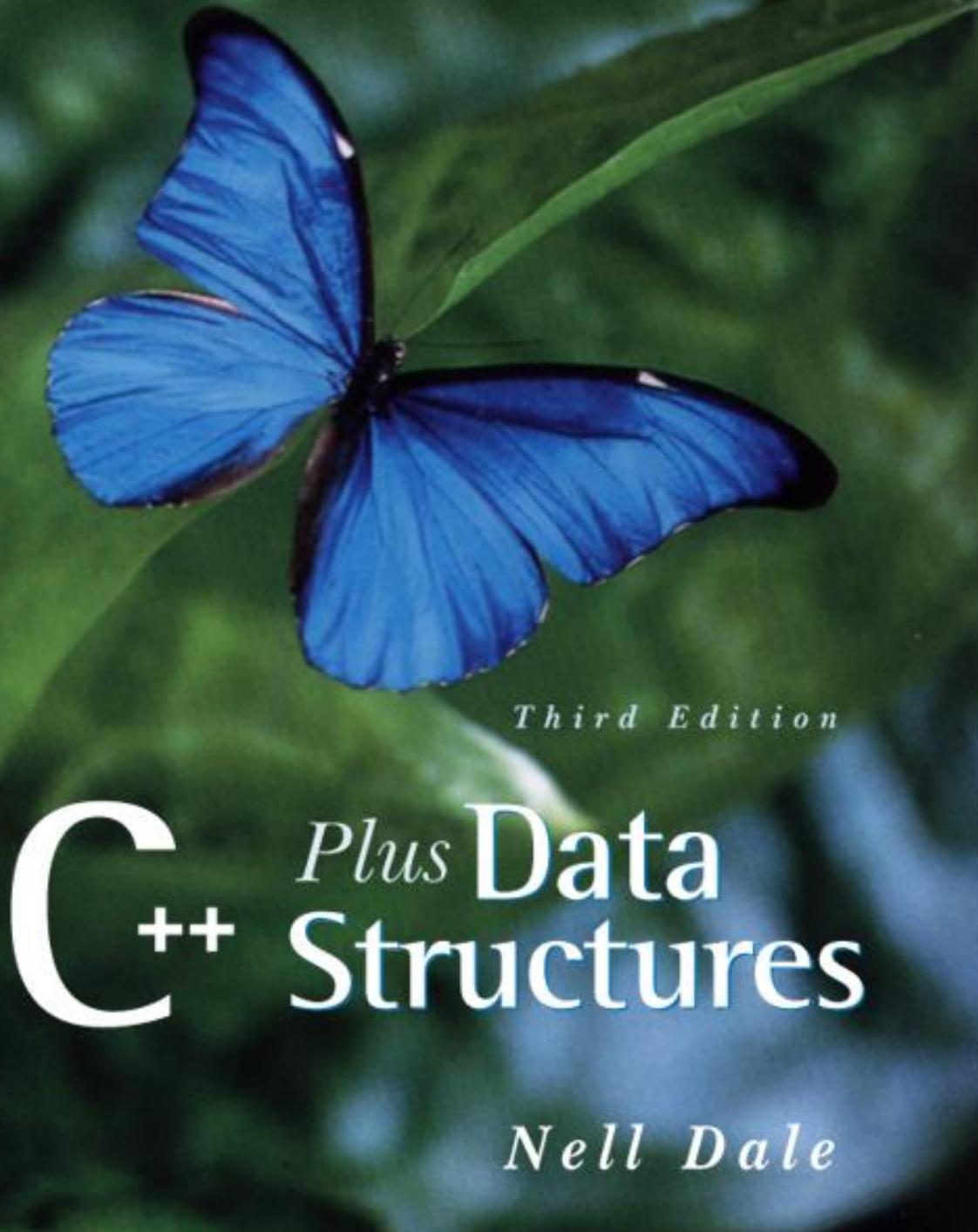
    { int splitPoint ;

        Split ( values, first, last, splitPoint ) ;

        // values [ first ] . . values[splitPoint - 1 ] <= splitVal
        // values [ splitPoint ] = splitVal
        // values [ splitPoint + 1 ] . . values[ last ] > splitVal

        QuickSort( values, first, splitPoint - 1 ) ;
        QuickSort( values, splitPoint + 1, last ) ;
    }
}

```

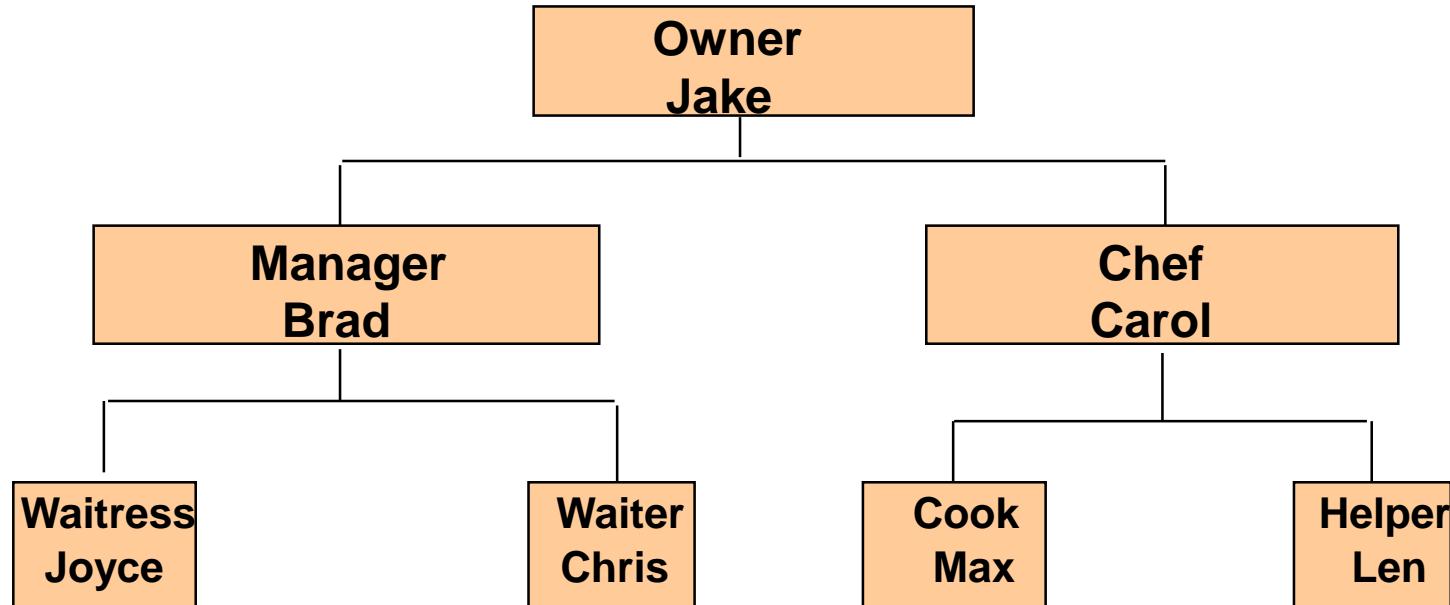


Chapter
8-1

Binary
Search Trees



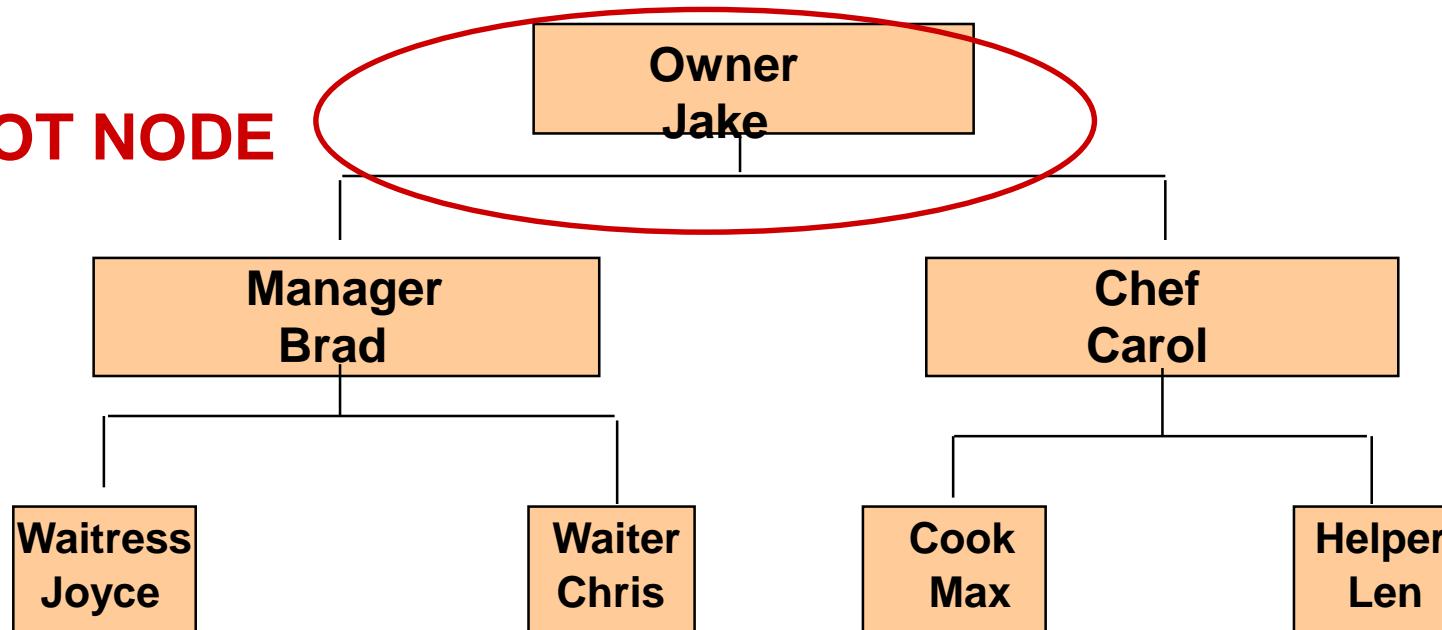
Jake's Pizza Shop





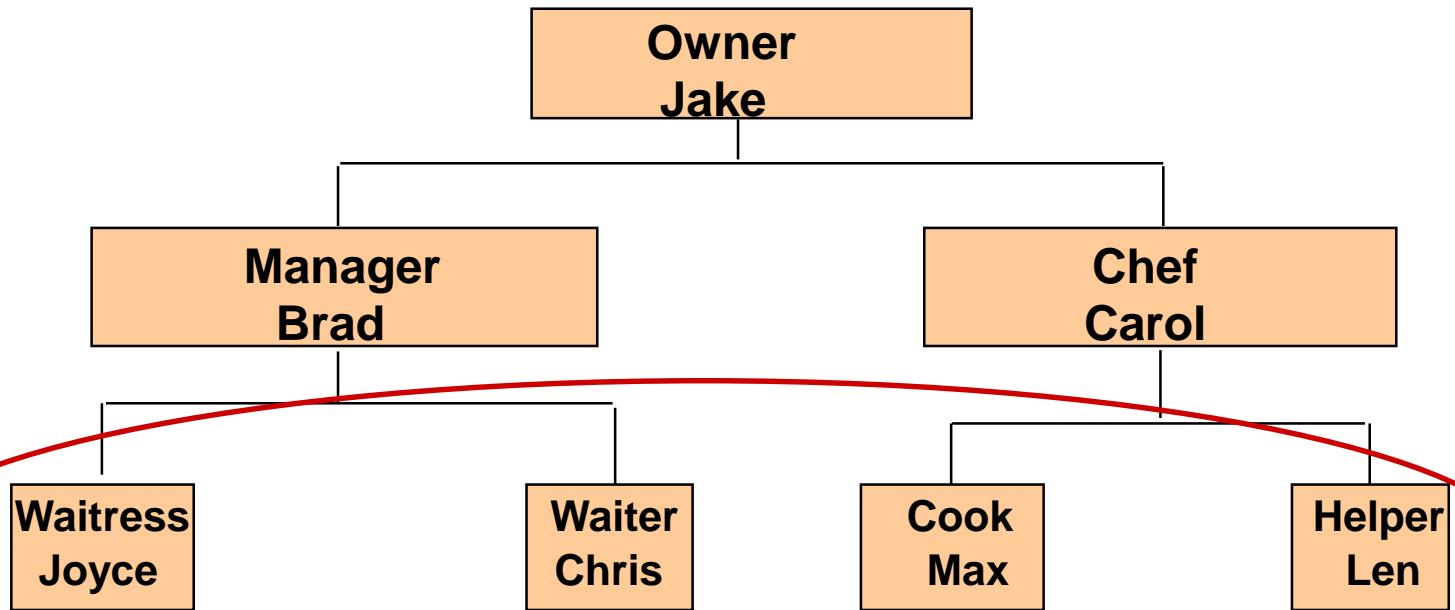
A Tree Has a Root Node

ROOT NODE





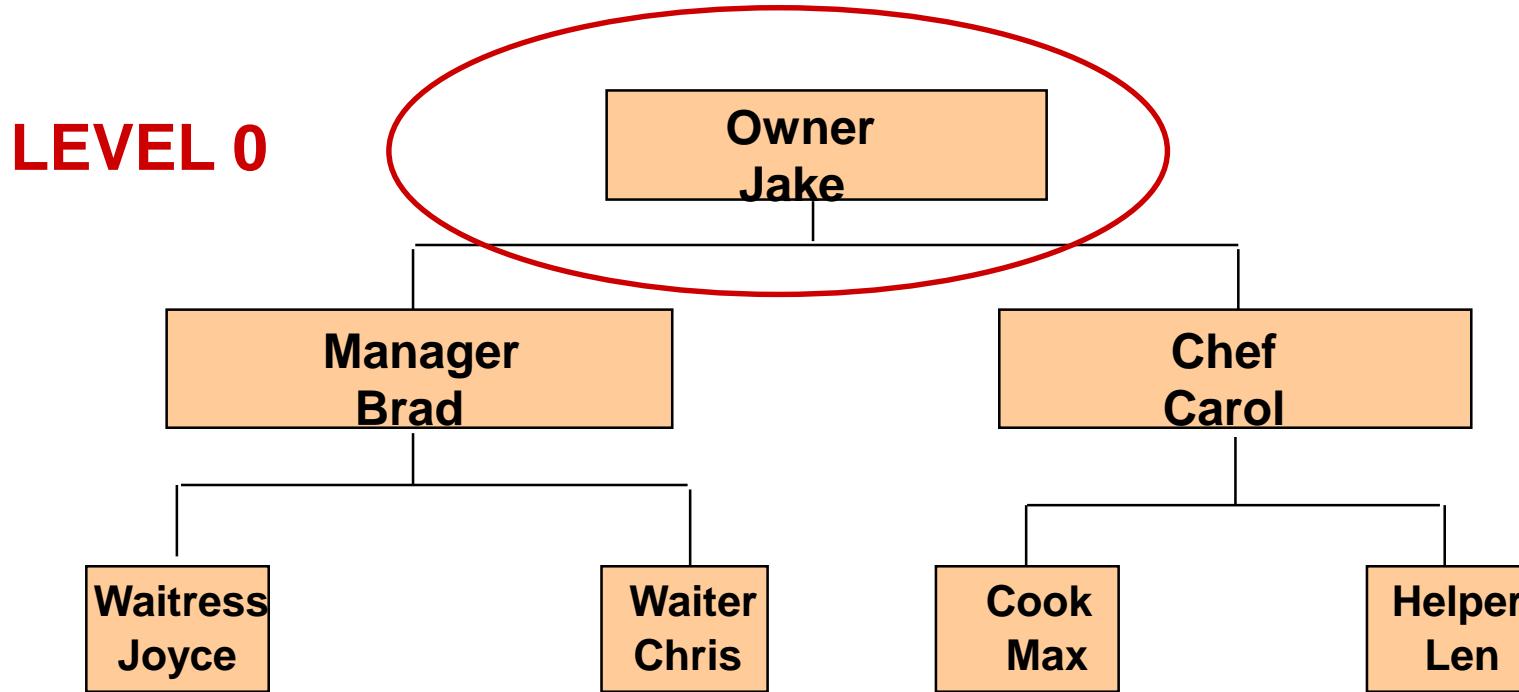
Leaf Nodes have No Children



LEAF NODES



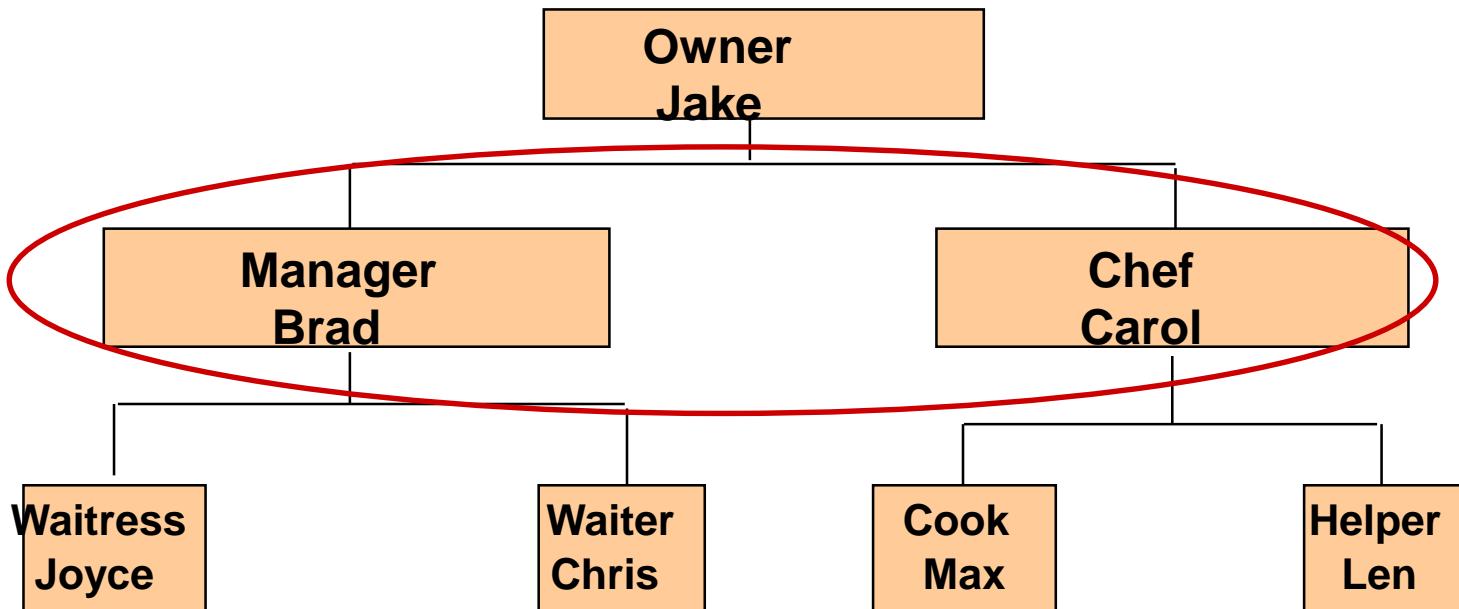
A Tree Has Leaves





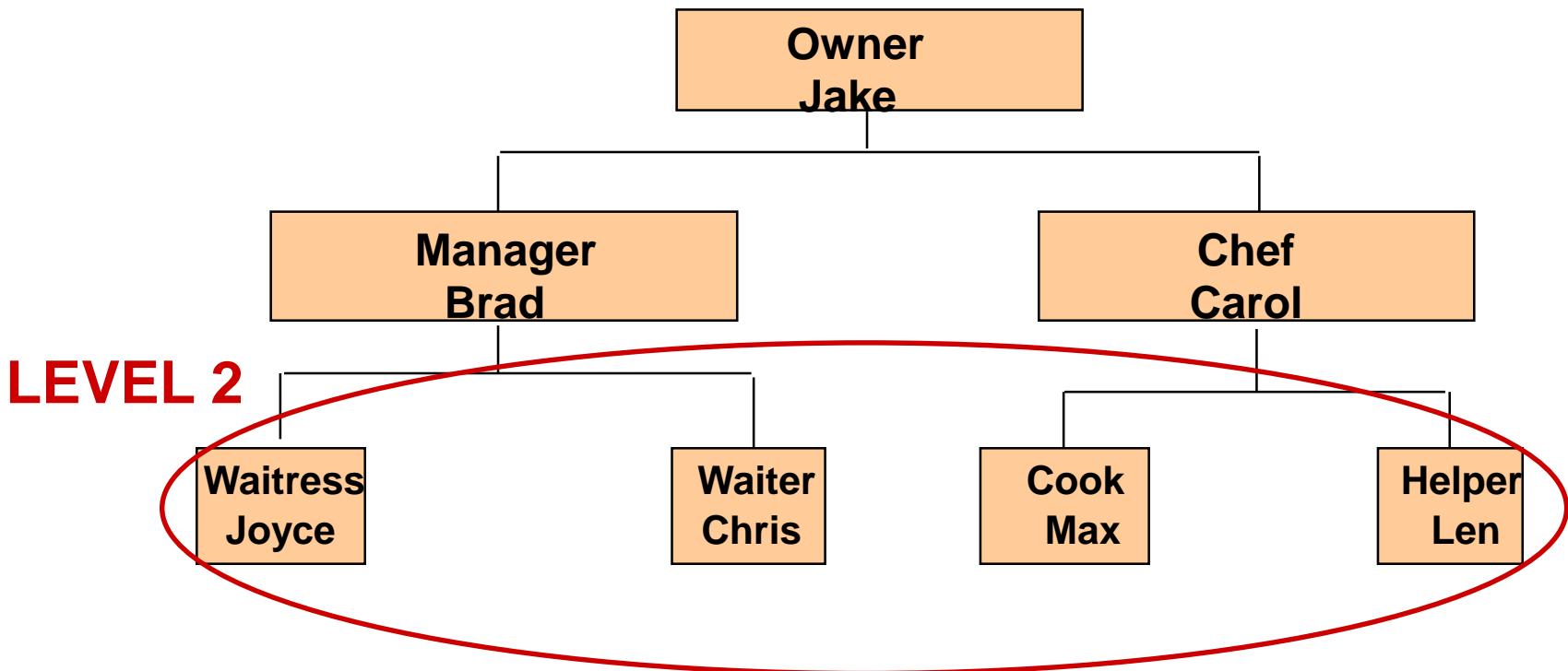
Level One

LEVEL 1



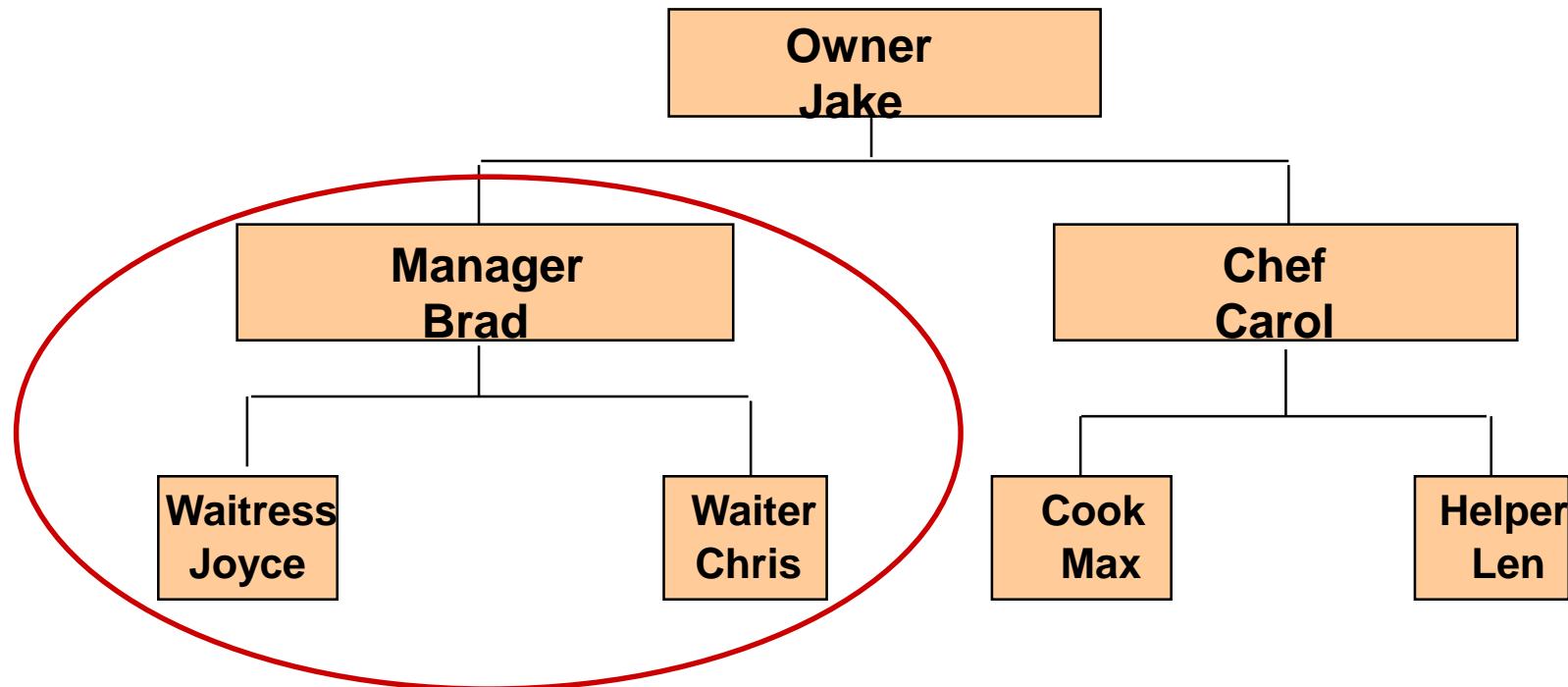


Level Two



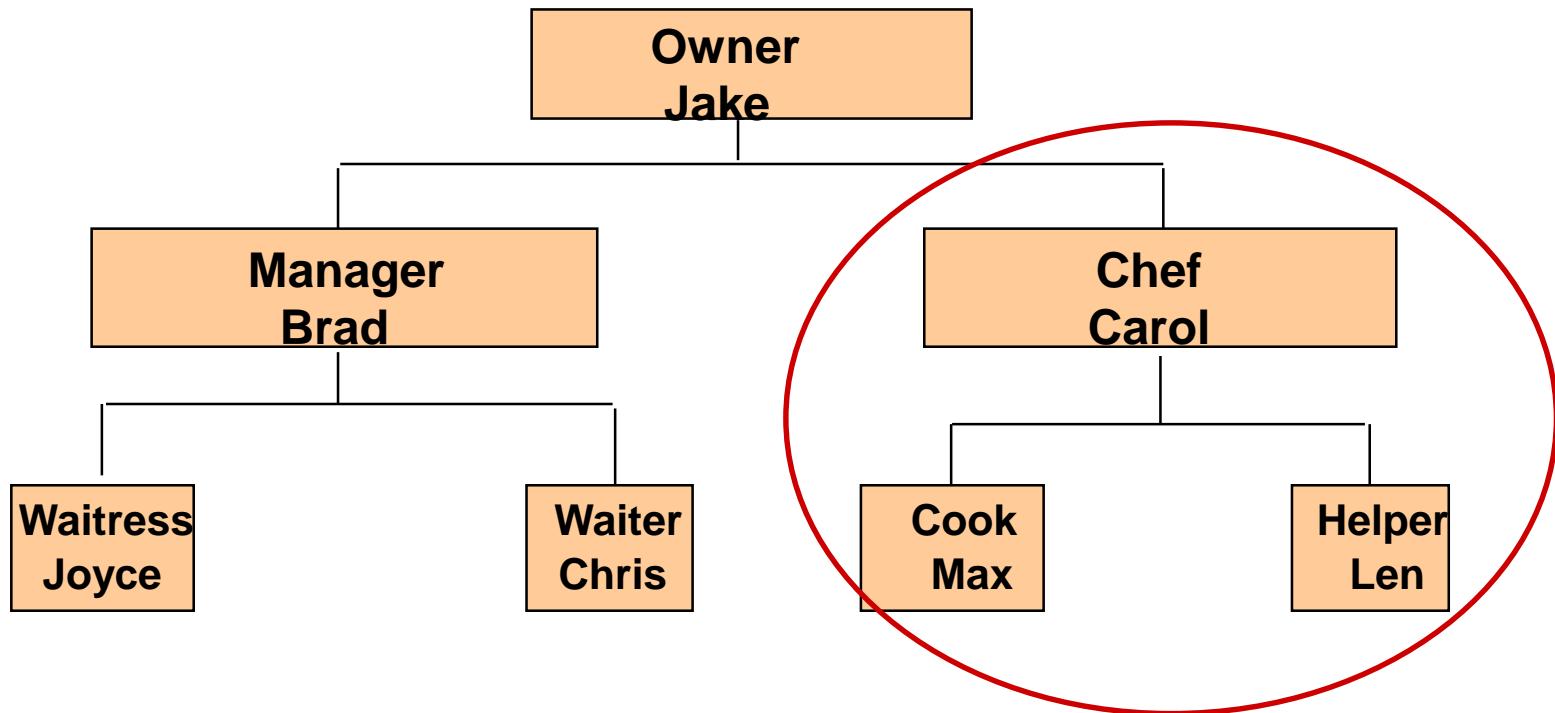


A Subtree



LEFT SUBTREE OF ROOT NODE

Another Subtree

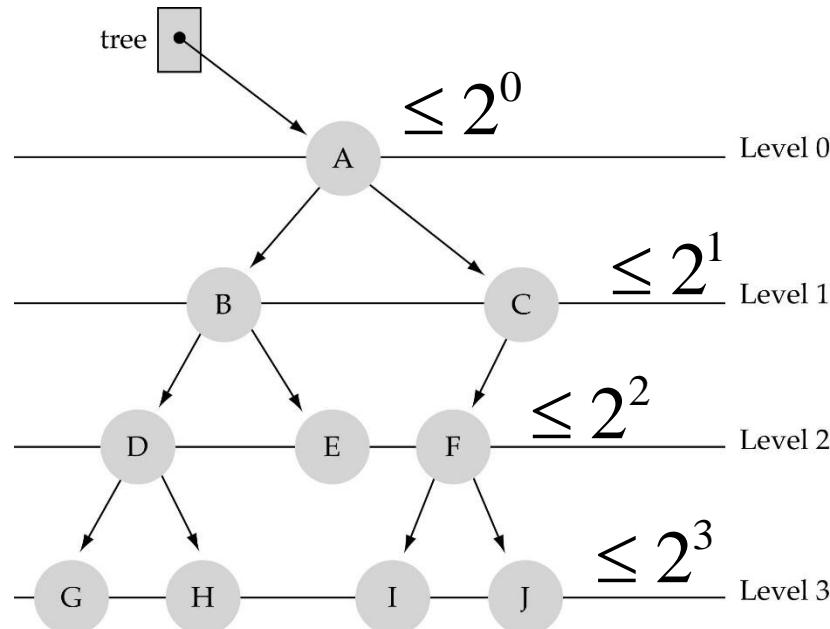


**RIGHT SUBTREE
OF ROOT NODE**



What is a binary tree?

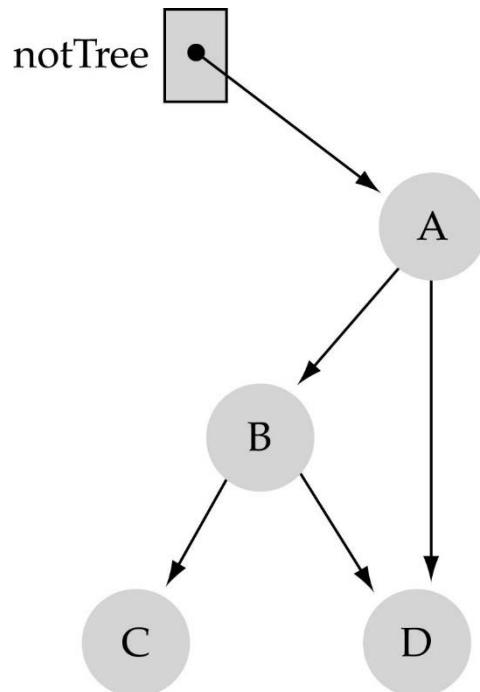
- *Property 1*: each node can have up to two successor nodes (*children*)
 - The predecessor node of a node is called its *parent*
 - The "beginning" node is called the *root* (no parent)
 - A node without *children* is called a *leaf*





What is a binary tree? (cont.)

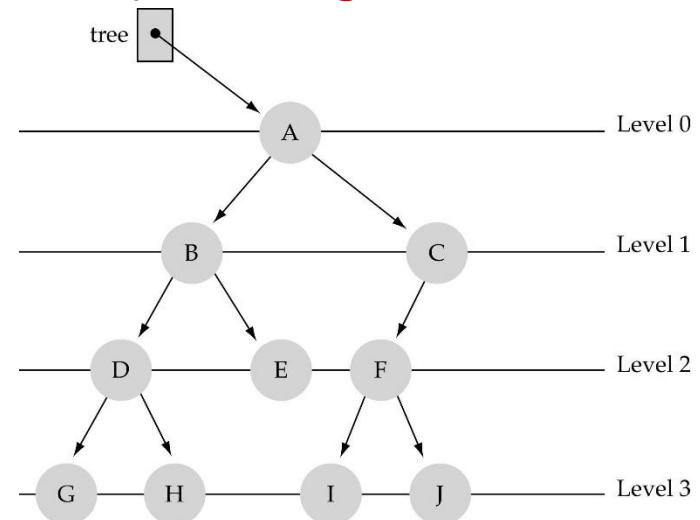
- *Property2:* a unique path exists from the root to every other node





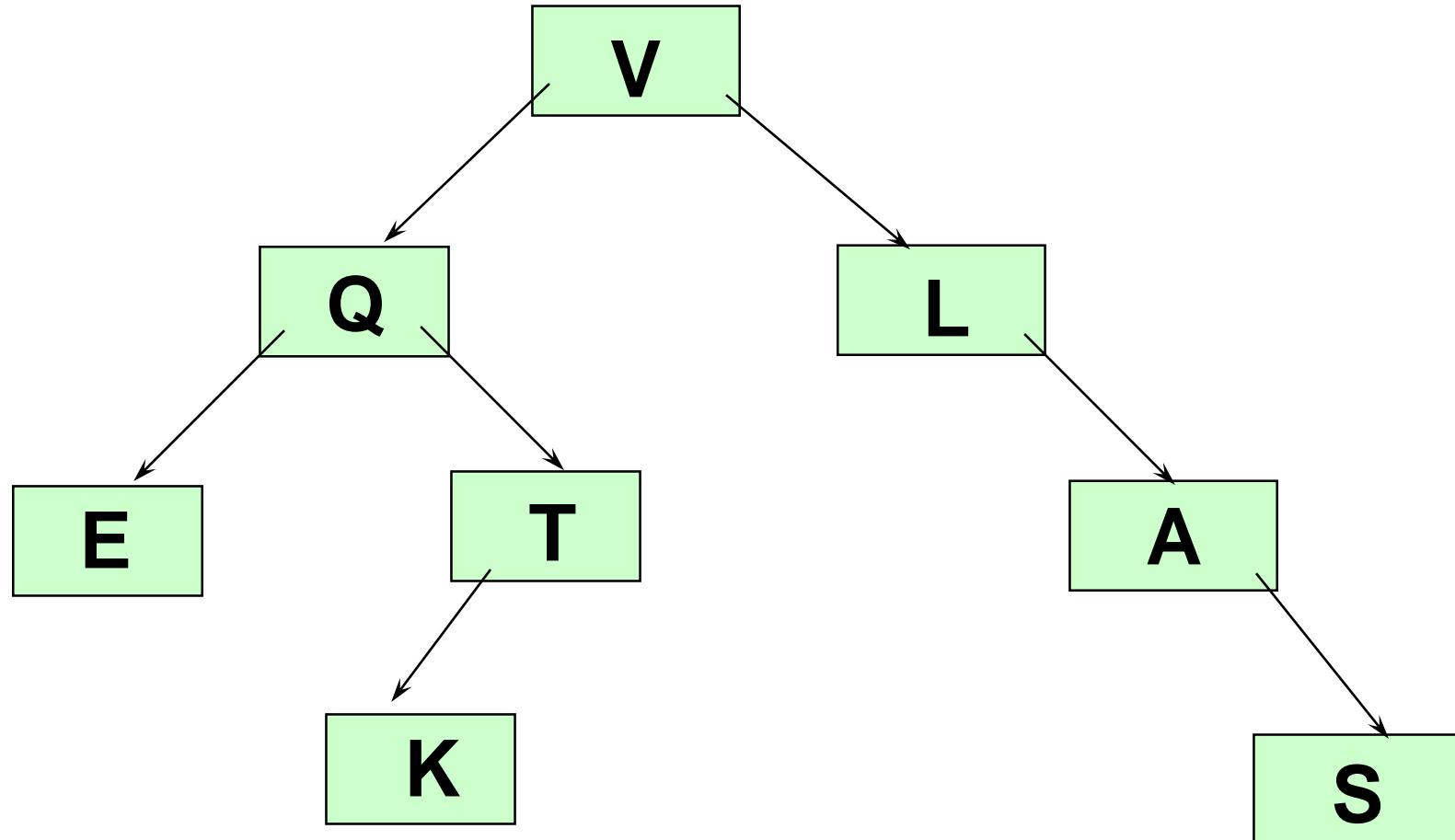
Some terminology

- Ancestor of a node: any node on the path from the root to that node (parent, grandparent, ...)
- Descendant of a node: any node on a path from the node to the last node in the path (child, grandchild, ...)
- Level (depth) of a node: number of edges in the path from the root to that node
- Height of a tree: number of levels (**warning**: some books define it as #levels - 1)



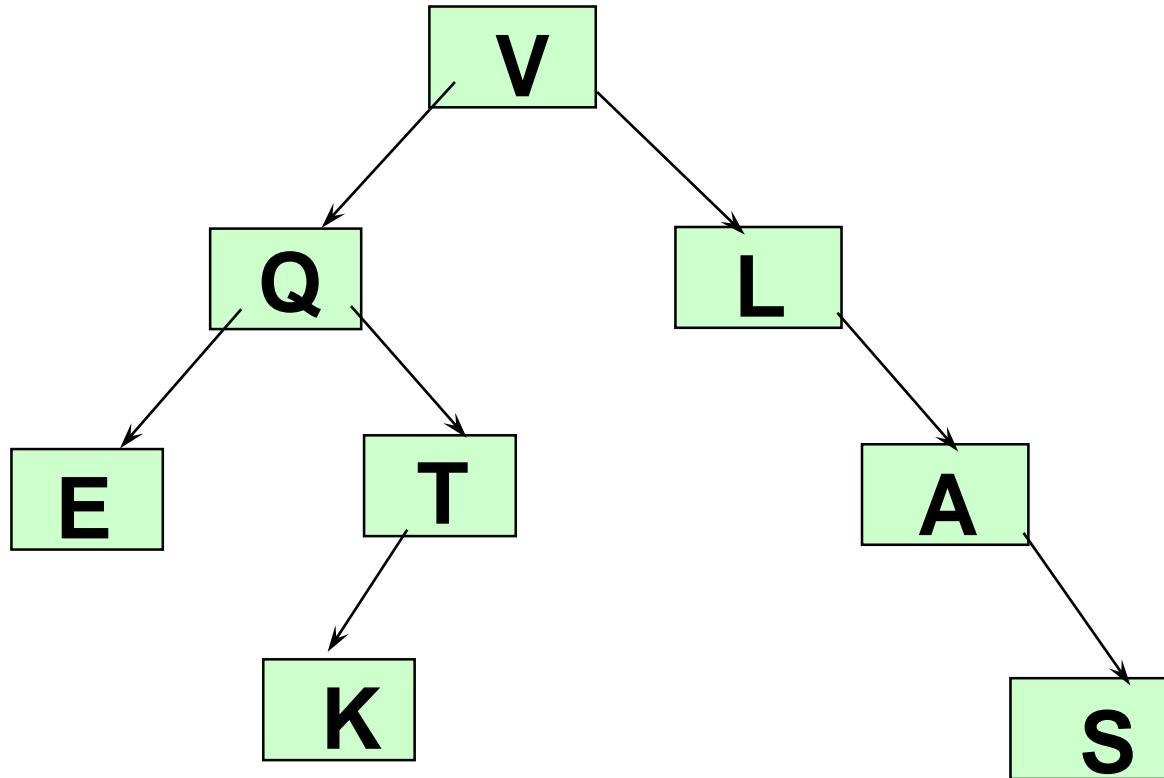


A Binary Tree



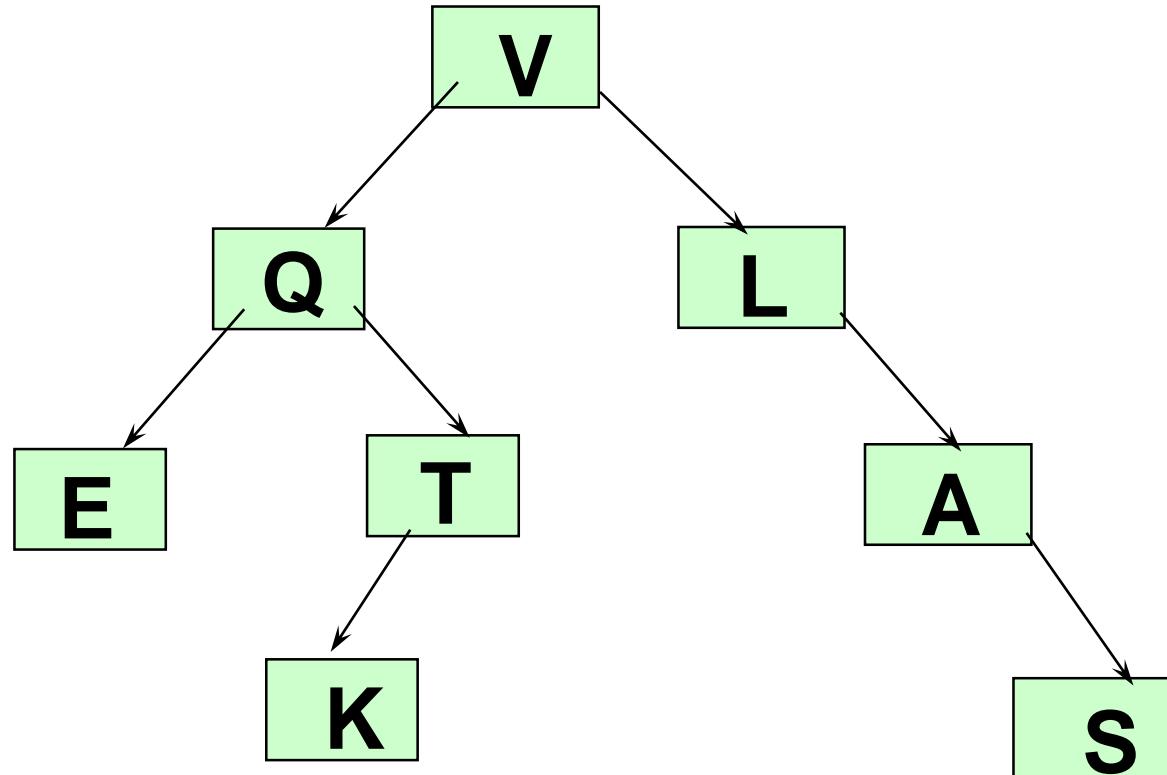


How many leaf nodes?



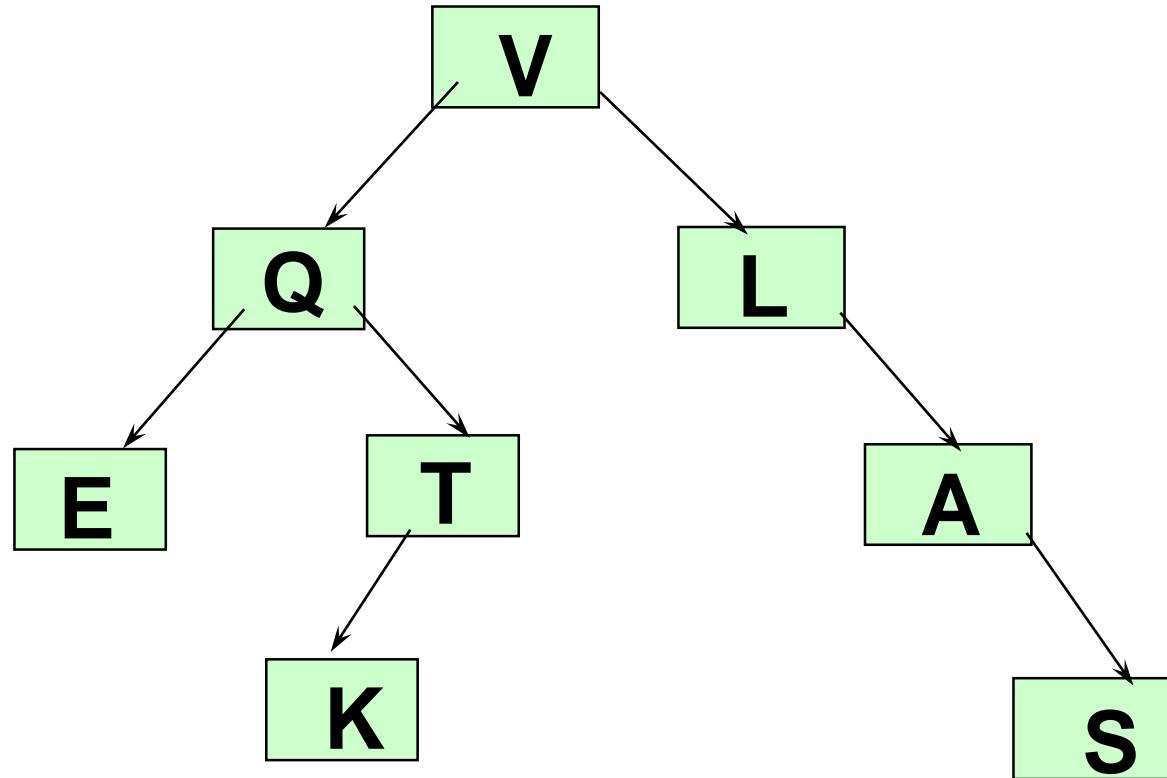


How many descendants of Q?





How many ancestors of K?





What is the # of nodes N of a full tree with height h?

full tree: a tree in which all of the leaves are on the same level and every nonleaf node has two children

The max #nodes at level l is 2^l

$$N = \sum_{l=0}^{h-1} 2^l = 2^h - 1$$

using the geometric series:

$$x^0 + x^1 + \dots + x^{n-1} = \sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$



What is the height h of a full tree with N nodes?

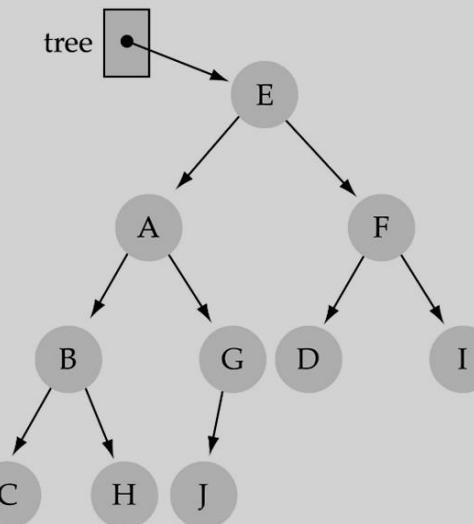
$$2^h - 1 = N$$

$$\Rightarrow 2^h = N + 1$$

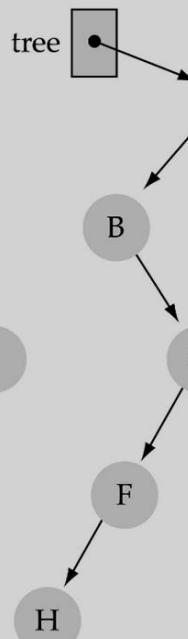
$$\Rightarrow h = \log(N + 1) \rightarrow O(\log N)$$

- The max height of a tree with N nodes is N (same as a linked list)
- The min height of a tree with N nodes is $\log(N+1)$

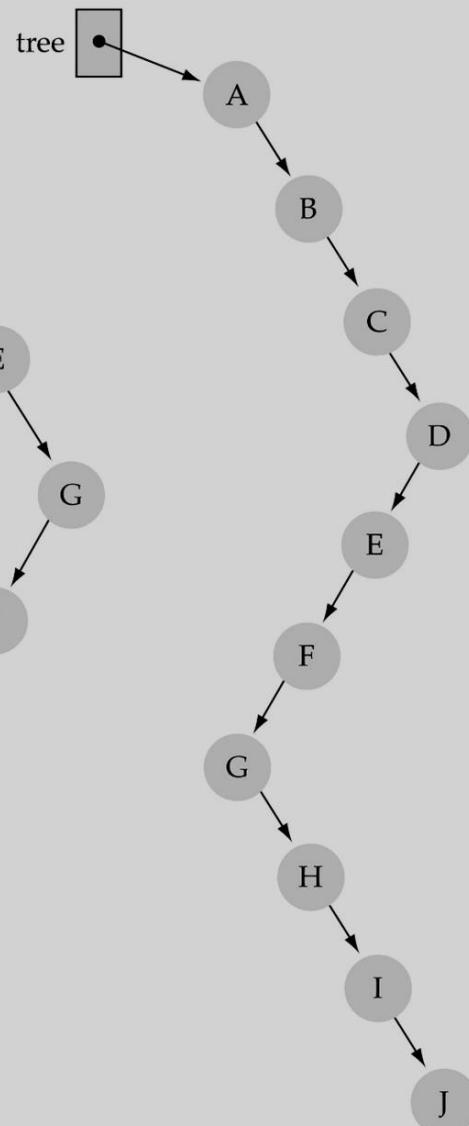
(a) A 4-level tree



(b) A 5-level tree



(c) A 10-level tree





Searching a binary tree

- (1) Start at the root
- (2) Search the tree level by level, until you find the element you are searching for
($O(N)$ time in worst case)

Is this better than searching a linked list?

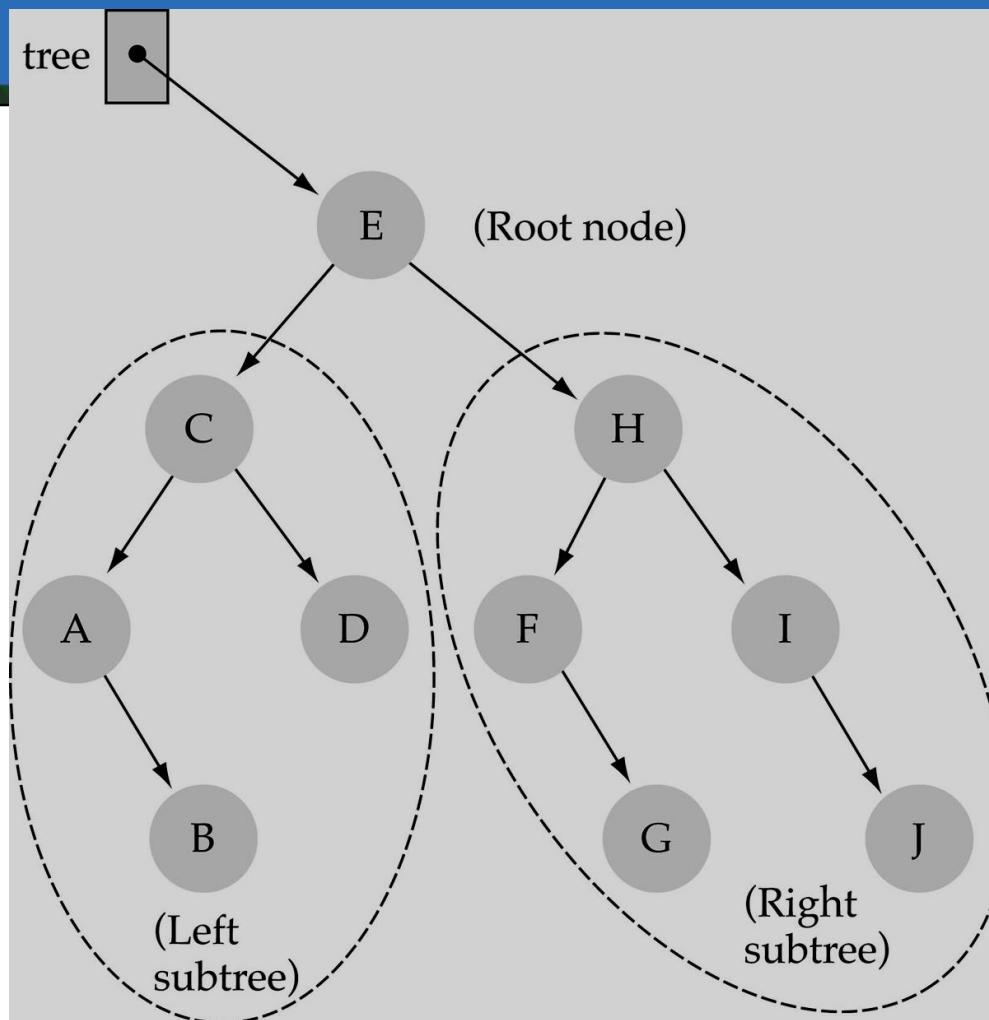
No ---> $O(N)$



A Binary Search Tree (BST) is . . .

A special kind of binary tree in which:

1. Each node contains a distinct data value,
2. The key values in the tree can be compared using “greater than” and “less than”, and
3. The key value of each node in the tree is **less than every key value in its right subtree, and greater than every key value in its left subtree.**



All values in the left subtree are less than the value in the root node.

All values in the right subtree are greater than the value in the root node.



Shape of a binary search tree . . .

Depends on its key values and their order of insertion.

Insert the elements ‘J’ ‘E’ ‘F’ ‘T’ ‘A’ in that order.

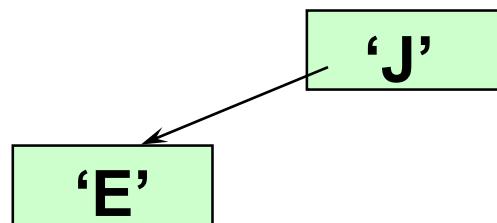
The first value to be inserted is put into the root node.

‘J’



Inserting 'E' into the BST

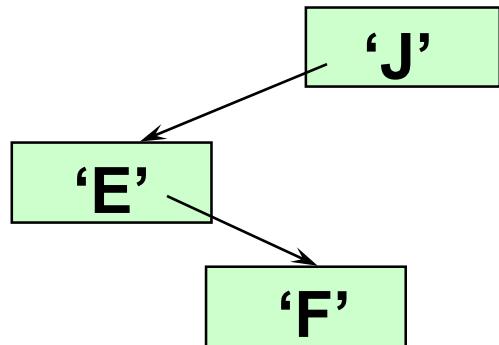
Thereafter, each value to be inserted begins by comparing itself to the value in the root node, moving left if it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.





Inserting ‘F’ into the BST

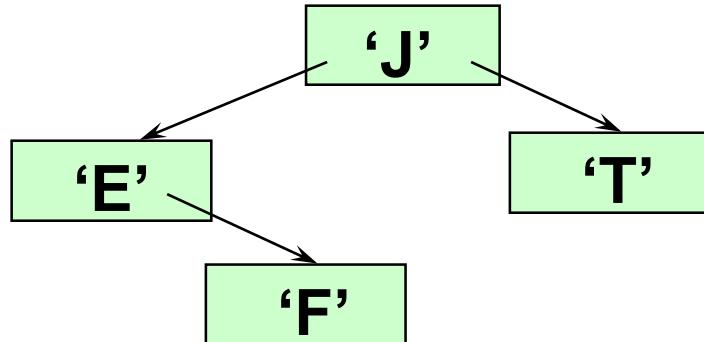
Begin by comparing ‘F’ to the value in the root node, moving left if it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.





Inserting ‘T’ into the BST

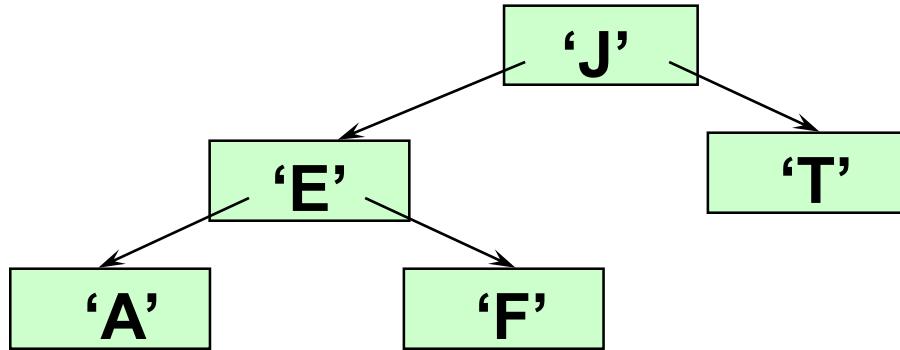
Begin by comparing ‘T’ to the value in the root node, moving left if it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.





Inserting ‘A’ into the BST

Begin by comparing ‘A’ to the value in the root node, moving left if it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.





What binary search tree . . .

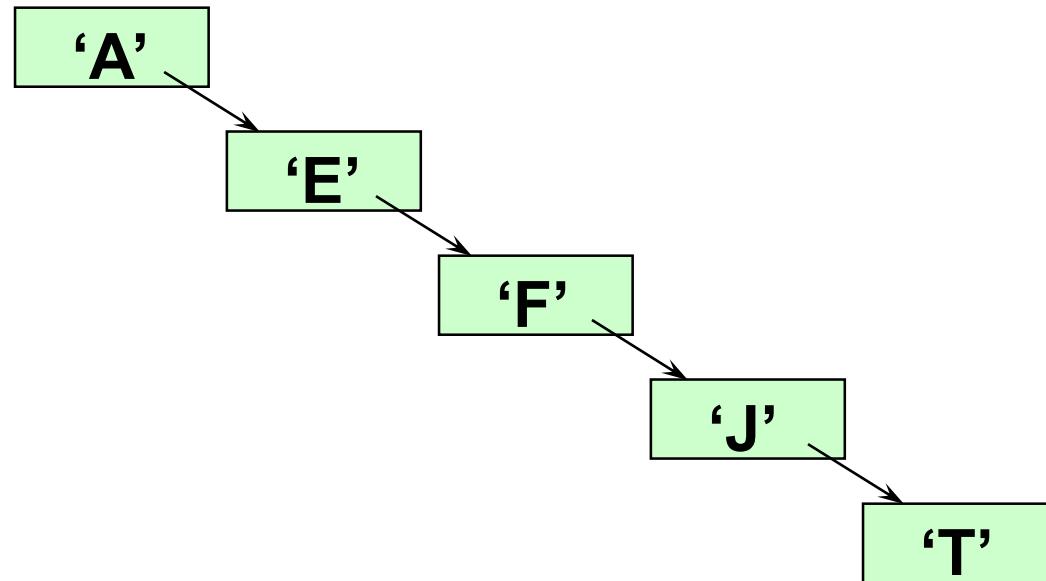
**is obtained by inserting
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?**

‘A’



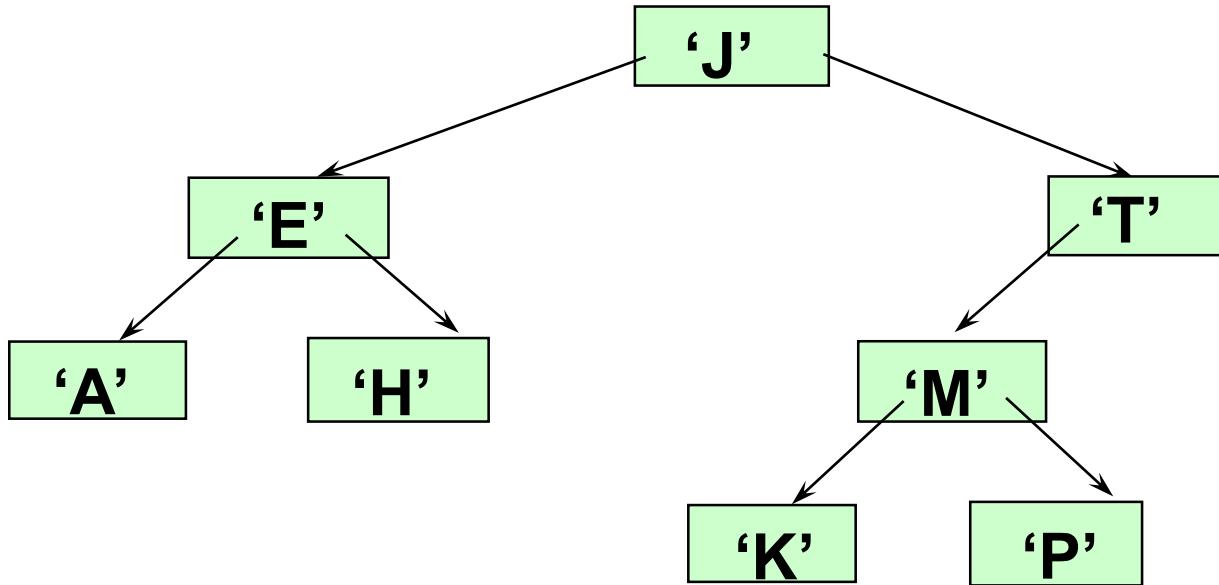
Binary search tree . . .

obtained by inserting
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order.





Another binary search tree

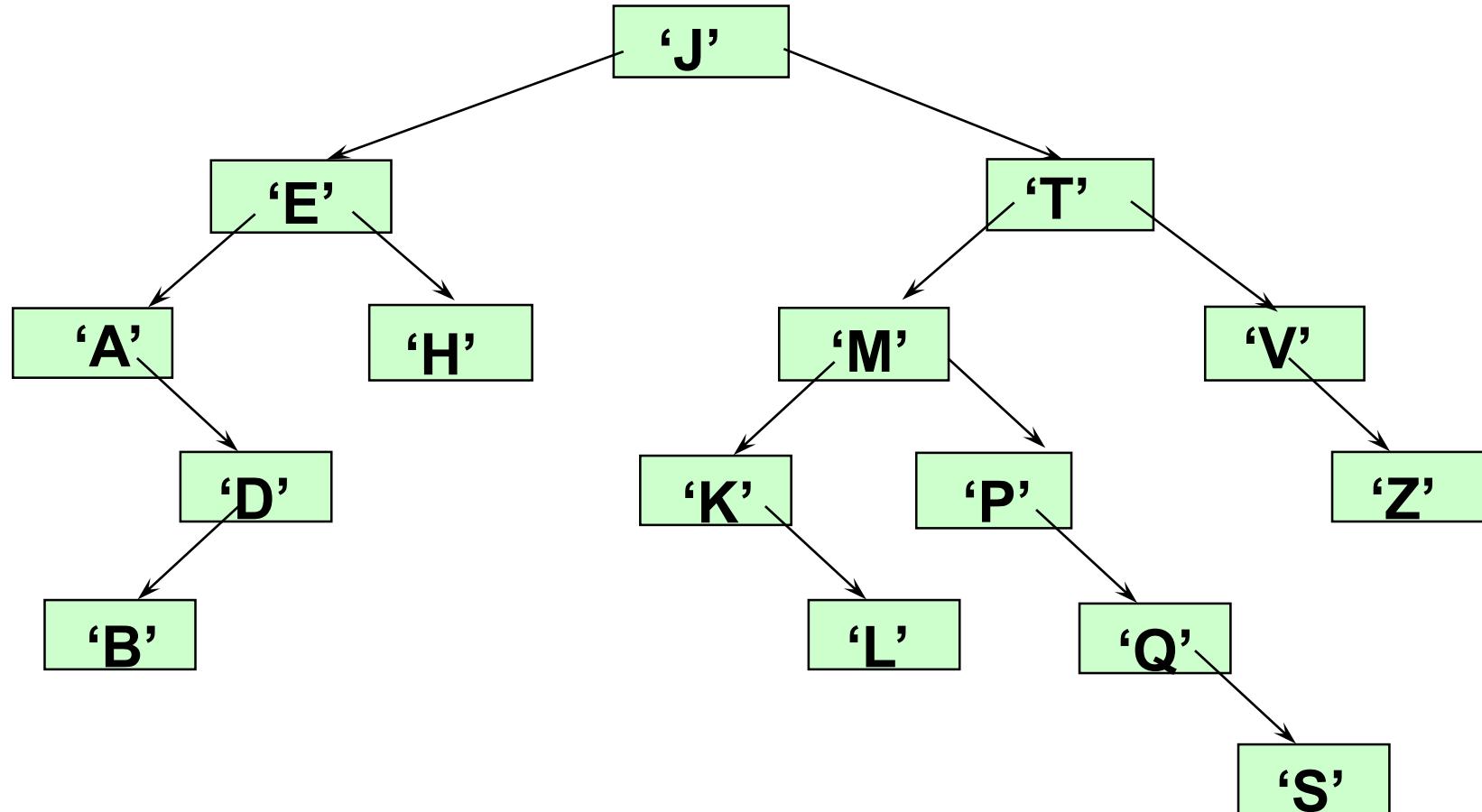


Add nodes containing these values in this order:

'D' 'B' 'L' 'Q' 'S' 'V' 'Z'



Is 'F' in the binary search tree?





Searching a binary search tree

- (1) Start at the root
- (2) Compare the value of the item you are searching for with the value stored at the root
- (3) If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*



Searching a binary search tree (cont.)

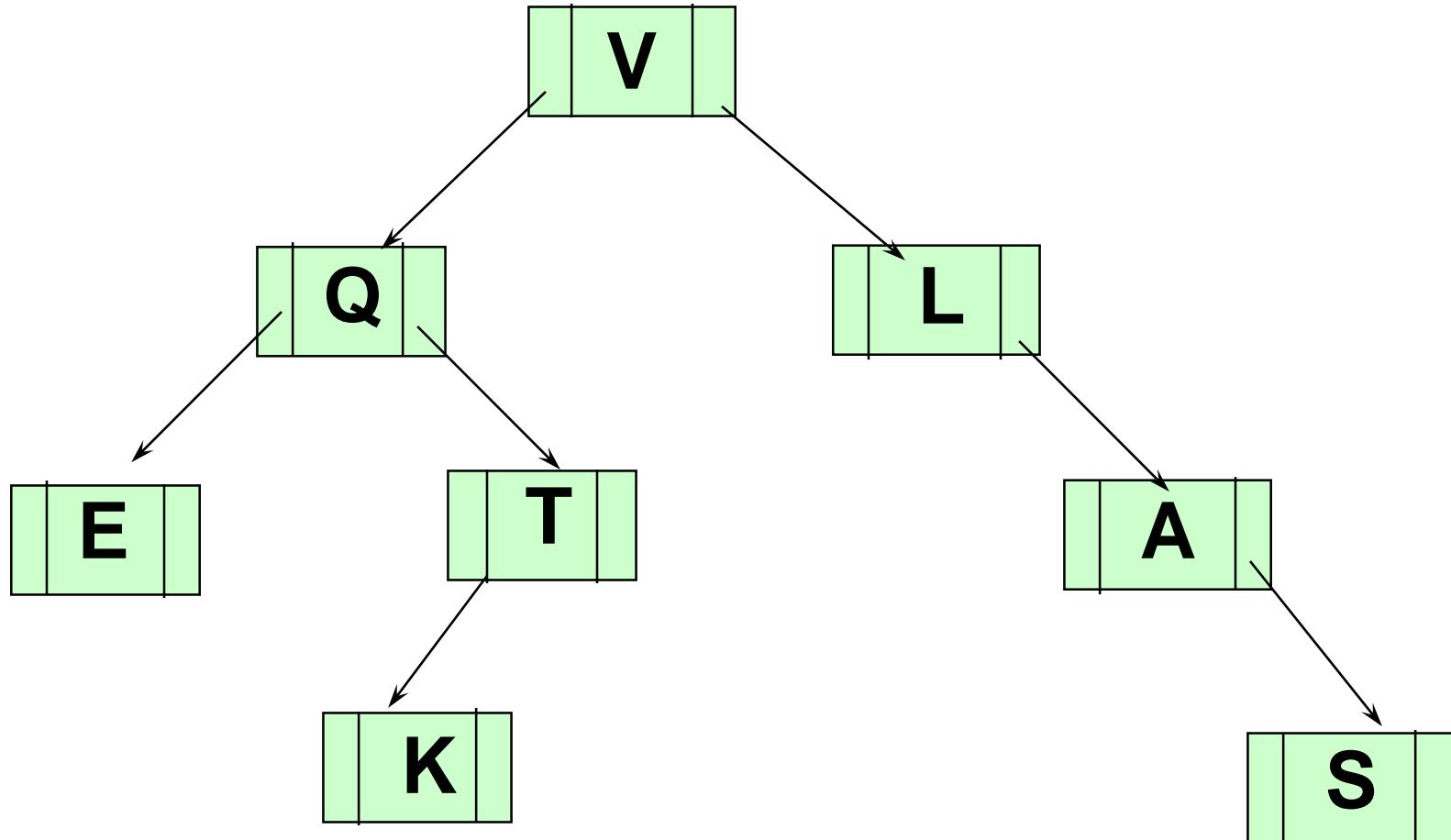
- (4) If it is less than the value stored at the root, then search the left subtree
- (5) If it is greater than the value stored at the root, then search the right subtree
- (6) Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

Is this better than searching a linked list?

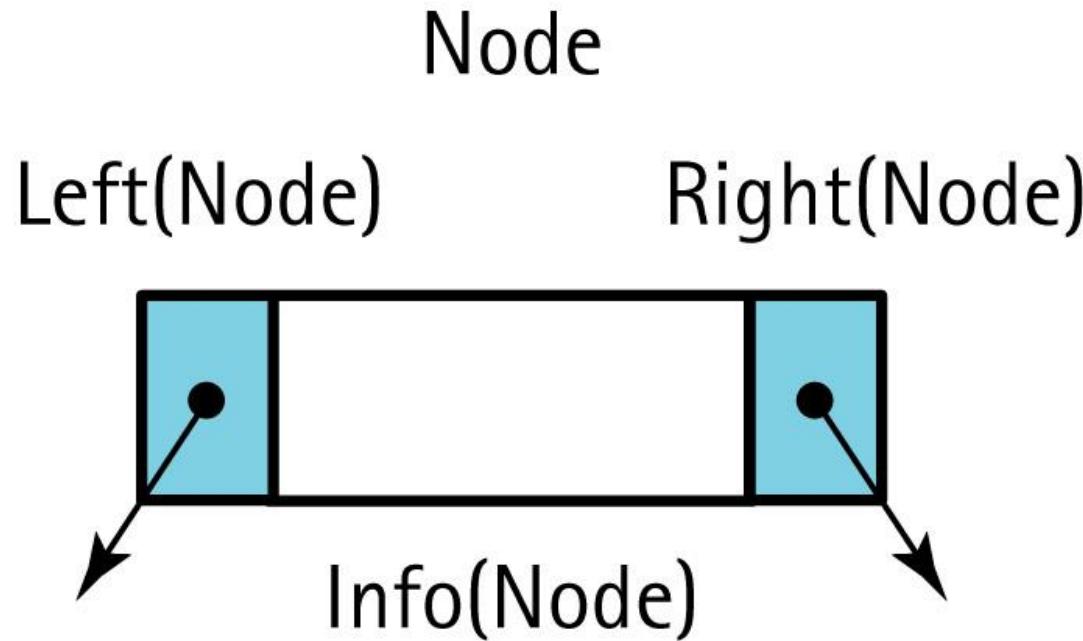
Yes !! ---> $O(\log N)$



Implementing a Binary Tree with Pointers and Dynamic Data



Node Terminology for a Tree Node



```
template<class ItemType>
struct TreeNode {
    ItemType info;
    TreeNode* left;
    TreeNode* right; };
```



Binary Search Tree Specification

```
#include <fstream.h>

template<class ItemType>
struct TreeNode;

enum OrderType { PRE_ORDER, IN_ORDER, POST_ORDER };

template<class ItemType>
class TreeType {
public:
    TreeType();
    ~TreeType();
    TreeType(const TreeType<ItemType>&);
    void operator=(const TreeType<ItemType>&);
```

(continues)



Binary Search Tree Specification

(cont.)

```
void MakeEmpty();  
bool IsEmpty() const;  
bool IsFull() const;  
int LengthIs() const;  
void RetrieveItem(ItemType&, bool& found);  
void InsertItem(ItemType);  
void DeleteItem(ItemType);  
void ResetTree(OrderType);  
void GetNextItem(ItemType&, OrderType, bool&);  
void PrintTree(ofstream&) const;  
private:  
TreeNode<ItemType>*> root;  
};
```



Functions IsFull() & IsEmpty()

```
bool TreeType::IsFull() const
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
        return false;
    }
    catch(std::bad_alloc exception)
    {
        return true;
    }
}

bool TreeType::IsEmpty() const
{
    return root == NULL;
}
```



Function CountNodes

- Recursive implementation

#nodes in a tree =

#nodes in left subtree + #nodes in right subtree + 1

- What is the size factor?

Number of nodes in the tree we are examining

- What is the base case?

The tree is empty

- What is the general case?

CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1



CountNodes Version 1

```
if (Left(tree) is NULL) AND (Right(tree) is NULL)  
    return 1  
else  
    return CountNodes(Left(tree)) +  
        CountNodes(Right(tree)) + 1
```

What happens when Left(tree) is NULL?



CountNodes Version 2

```
if (Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else if Left(tree) is NULL
    return CountNodes(Right(tree)) + 1
else if Right(tree) is NULL
    return CountNodes(Left(tree)) + 1
else return CountNodes(Left(tree)) +
    CountNodes(Right(tree)) + 1
```

What happens when the initial tree is NULL?



CountNodes Version 3

```
if tree is NULL
    return 0
else if (Left(tree) is NULL) AND (Right(tree) is NULL)
    return 1
else if Left(tree) is NULL
    return CountNodes(Right(tree)) + 1
else if Right(tree) is NULL
    return CountNodes(Left(tree)) + 1
else return CountNodes(Left(tree)) +
    CountNodes(Right(tree)) + 1
```

Can we simplify this algorithm?



CountNodes Version 4

if tree is NULL

 return 0

else

 return CountNodes(Left(tree)) +

 CountNodes(Right(tree)) + 1

Is that all there is?



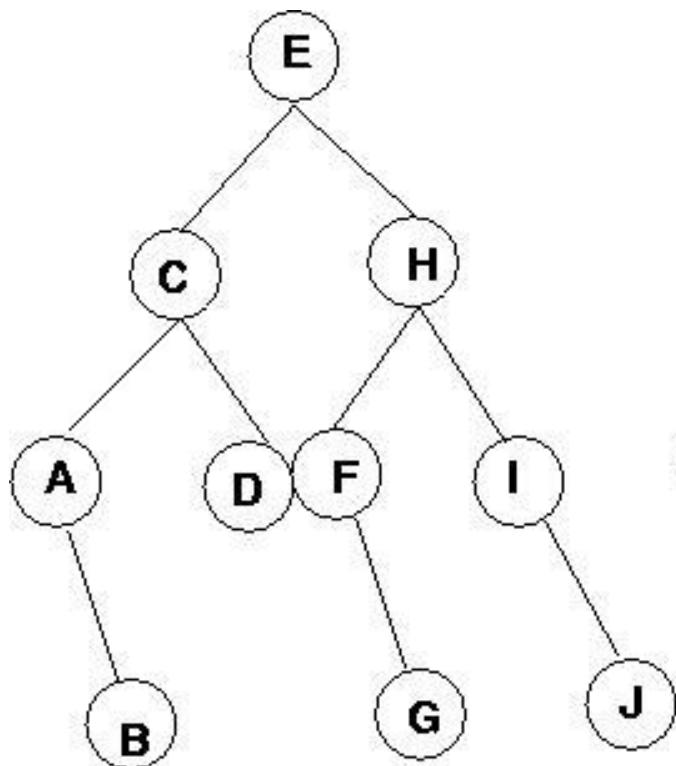
Function LengthIs()

```
// Implementation of Final Version
int CountNodes(TreeNode* tree); // Prototype
int TreeType::LengthIs() const
// Class member function
{
    return CountNodes(root);
}

int CountNodes(TreeNode* tree)
// Recursive function that counts the nodes
{
    if (tree == NULL)
        return 0;
    else
        return CountNodes(tree->left) +
            CountNodes(tree->right) + 1;
}
```



Let's consider the first few steps:



Count(left E) + Count(right E) + 1
ret 2

Count(left C) + Count(right C) + 1
ret 0 ret 1

Count(left A) + Count(right A) + 1
ret 0 ret 0

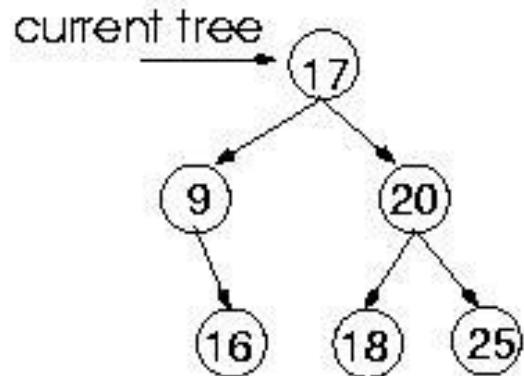
Count(left B) + Count(right B) + 1



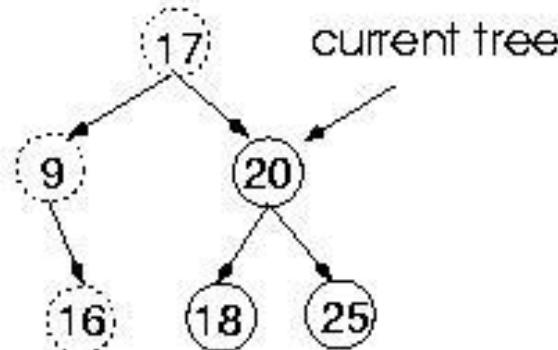
Function RetrieveItem

Retrieve: 18

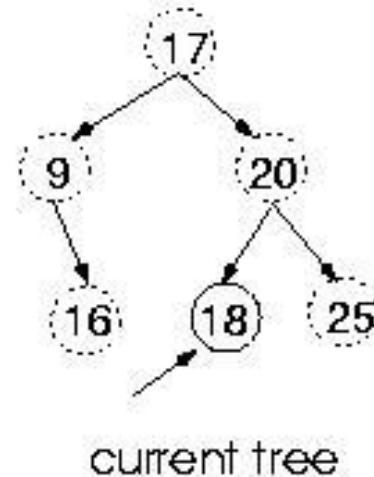
Compare 18 with 17:
Choose right subtree



Compare 18 with 20:
Choose left subtree



Compare 18 with 18:
Found !!





Function RetrieveItem

- What is the size of the problem?
Number of nodes in the tree we are examining
- What is the base case(s)?
 - 1) When the key is found
 - 2) The tree is empty (key was not found)
- What is the general case?
Search in the left or right subtrees



Retrieval Operation

```
void TreeType::RetrieveItem(ItemType& item, bool& found)
{
    Retrieve(root, item, found);
}

void Retrieve(TreeNode* tree,
              ItemType& item, bool& found)
{
    if (tree == NULL) // base case 2
        found = false;
    else if (item < tree->info)
        Retrieve(tree->left, item, found);
```



Retrieval Operation, cont.

```
else if (item > tree->info)
    Retrieve(tree->right, item, found);
else // base case 1
{
    item = tree->info;
    found = true;
}
}
```



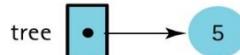
The Insert Operation

- A new node is always inserted into its appropriate position in the tree as a leaf.

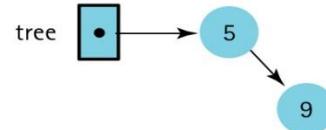
Insertions into a Binary Search Tree



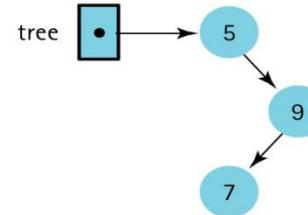
(b) Insert 5



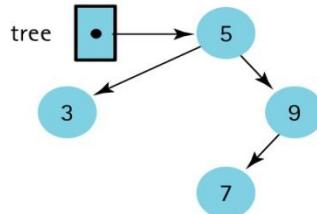
(c) Insert 9



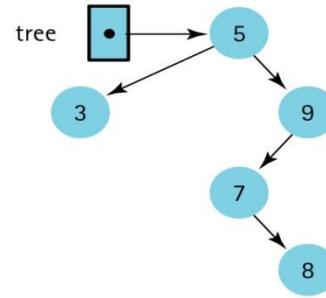
(d) Insert 7



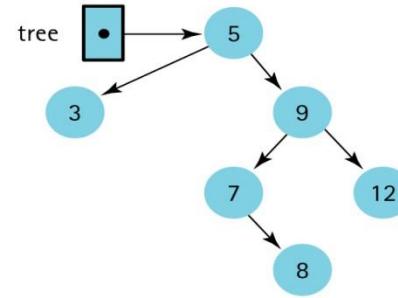
(e) Insert 3



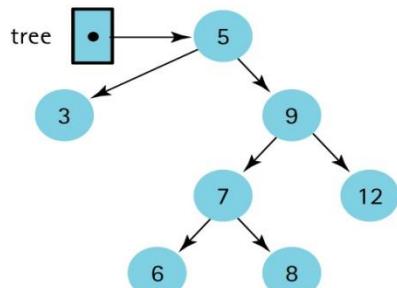
(f) Insert 8



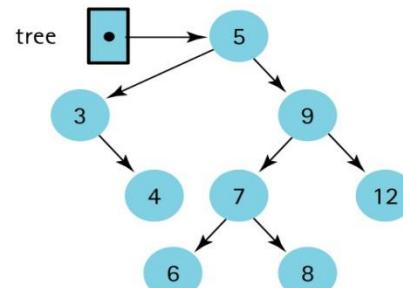
(g) Insert 12



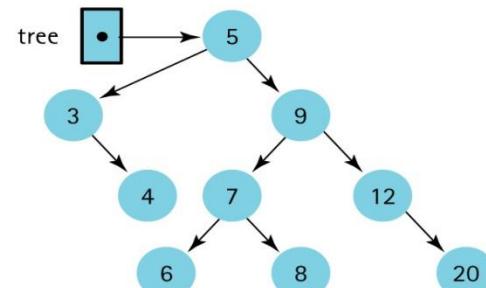
(h) Insert 6



(i) Insert 4



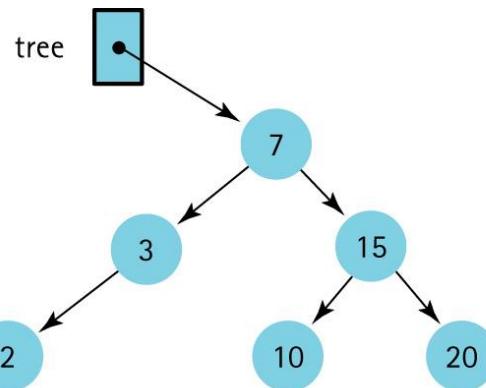
(j) Insert 20



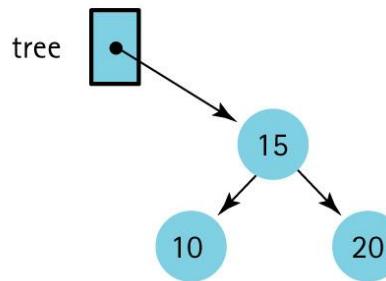
The recursive InsertItem operation

insert 11

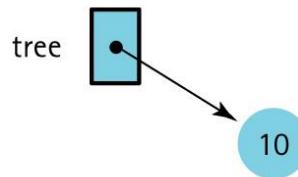
(a) The initial call



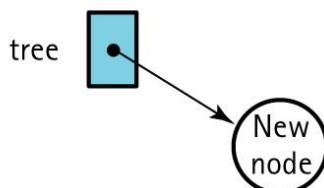
(b) The first recursive call



(c) The second recursive call



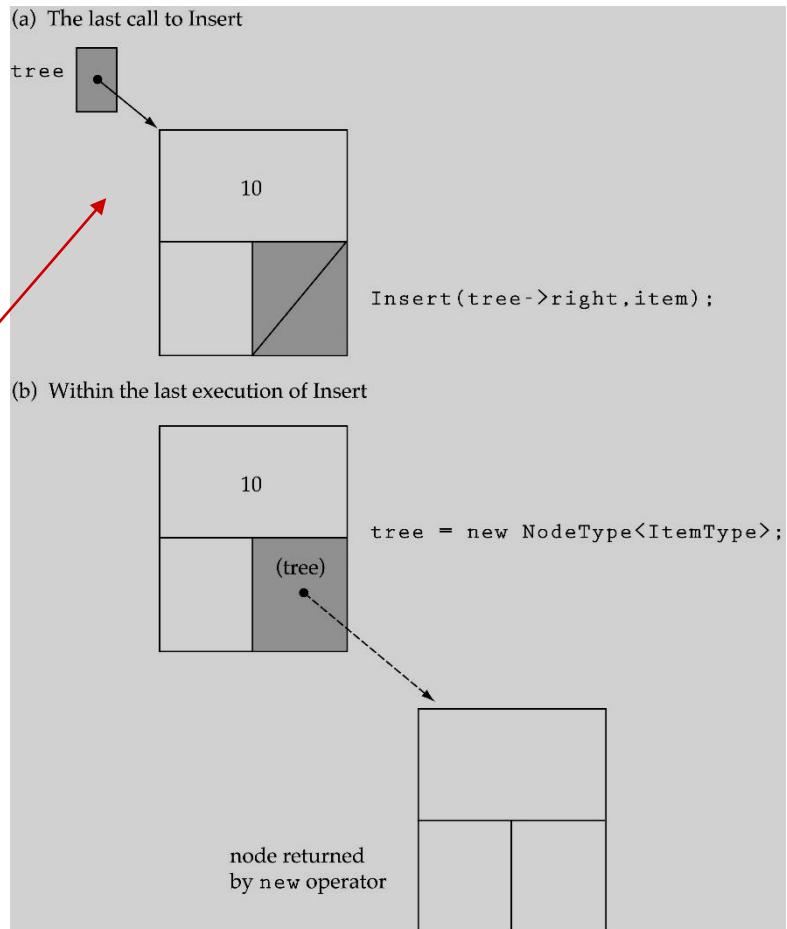
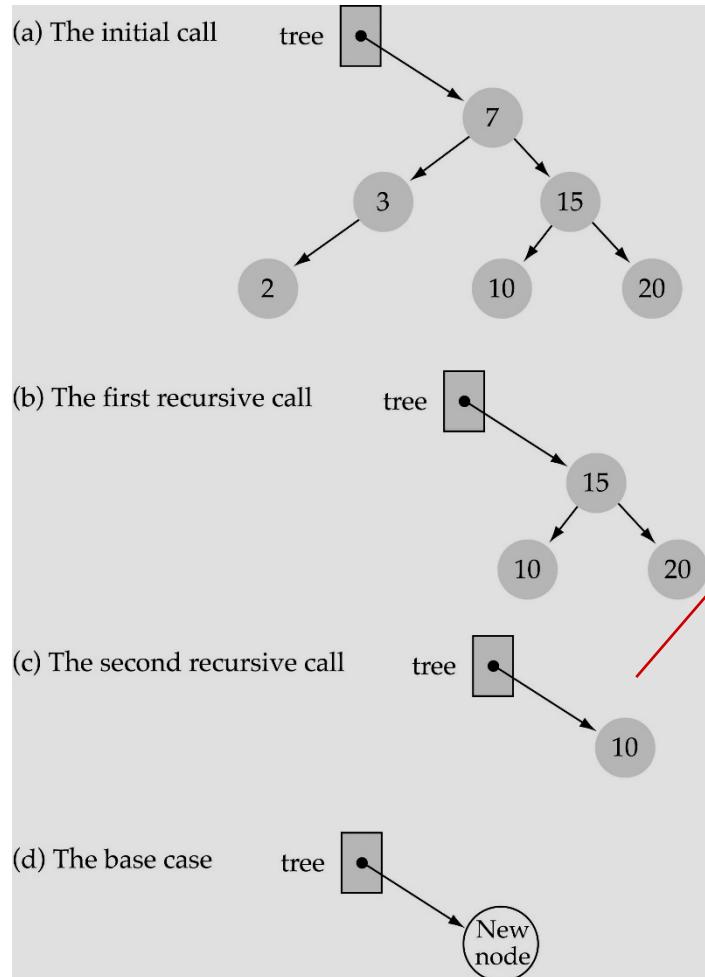
(d) The base case





The tree parameter is a pointer within the tree

Insert 11





Recursive Insert

```
void Insert(TreeNode*& tree, ItemType item)
{
    if (tree == NULL)
        // Insertion place found.
        tree = new TreeNode;
        tree->right = NULL;
        tree->left = NULL;
        tree->info = item;
    }
    else if (item < tree->info)
        Insert(tree->left, item);
    else
        Insert(tree->right, item);
}
```

referenc
e type



Does the order of inserting elements into a tree matter?

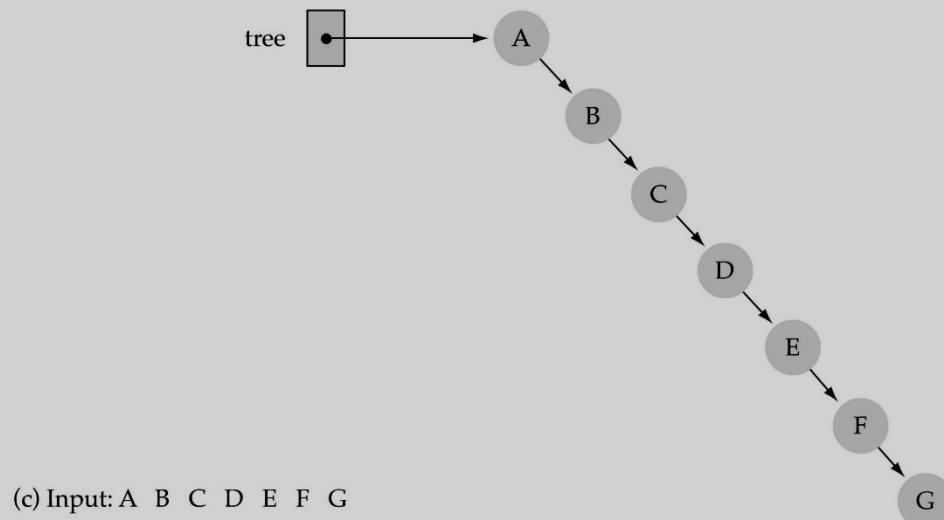
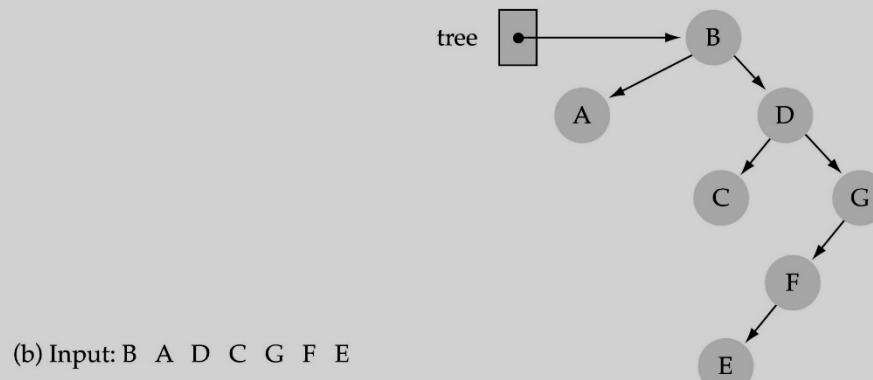
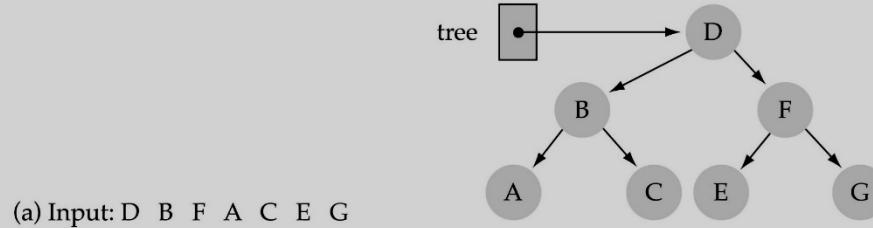
- Yes, certain orders produce very unbalanced trees!!
- Unbalanced trees are not desirable because search time increases!!
- There are advanced tree structures (e.g., "red-black trees") which guarantee balanced trees



Does the order of inserting elements into a tree matter?

Yes!!!

A random mix of the elements produces a shorter, “bushy” tree



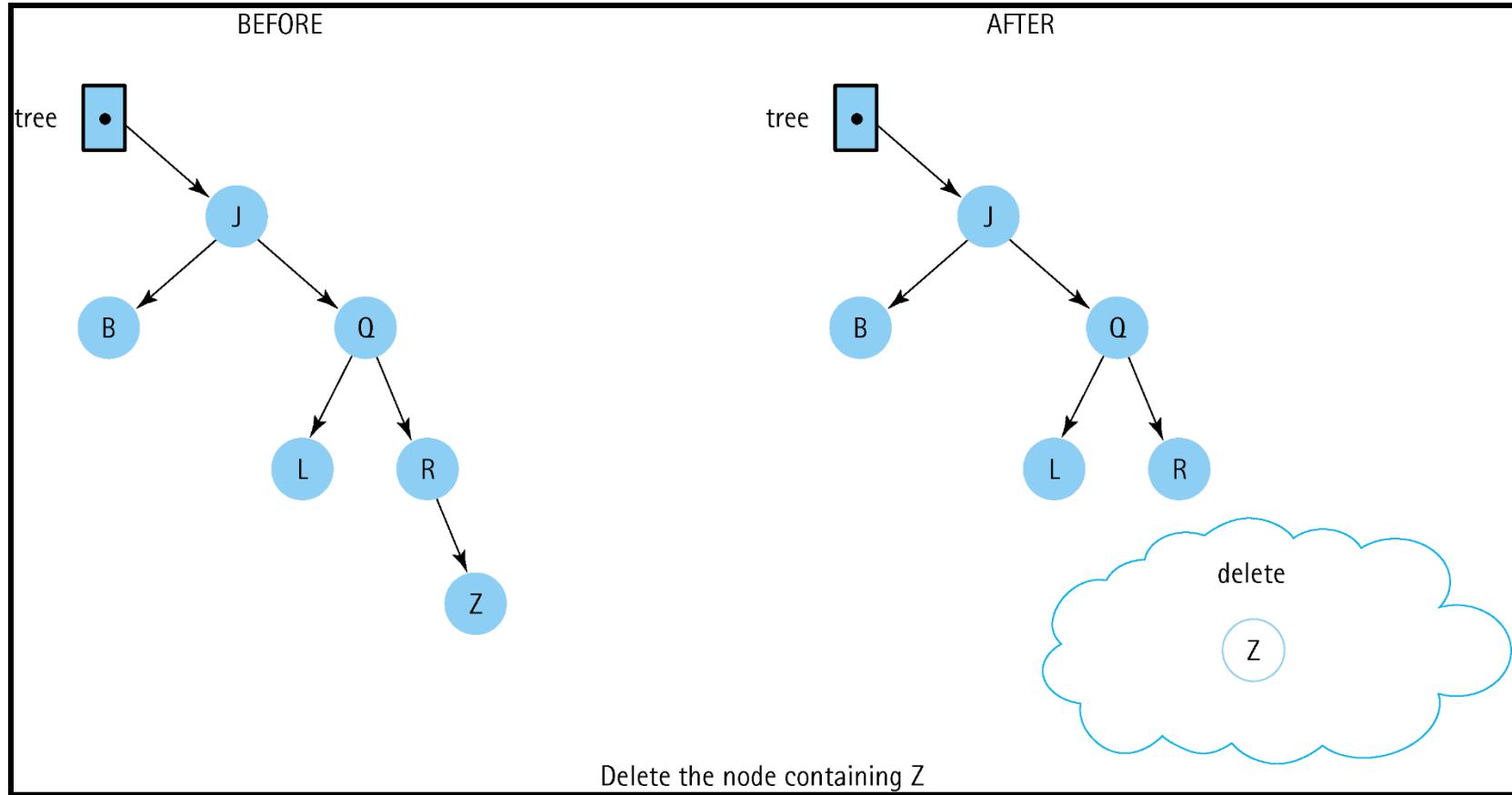


Function DeleteItem

- First, find the item; then, delete it
- Important: binary search tree property must be preserved!!
- We need to consider three different cases:
 - (1) Deleting a leaf
 - (2) Deleting a node with only one child
 - (3) Deleting a node with two children

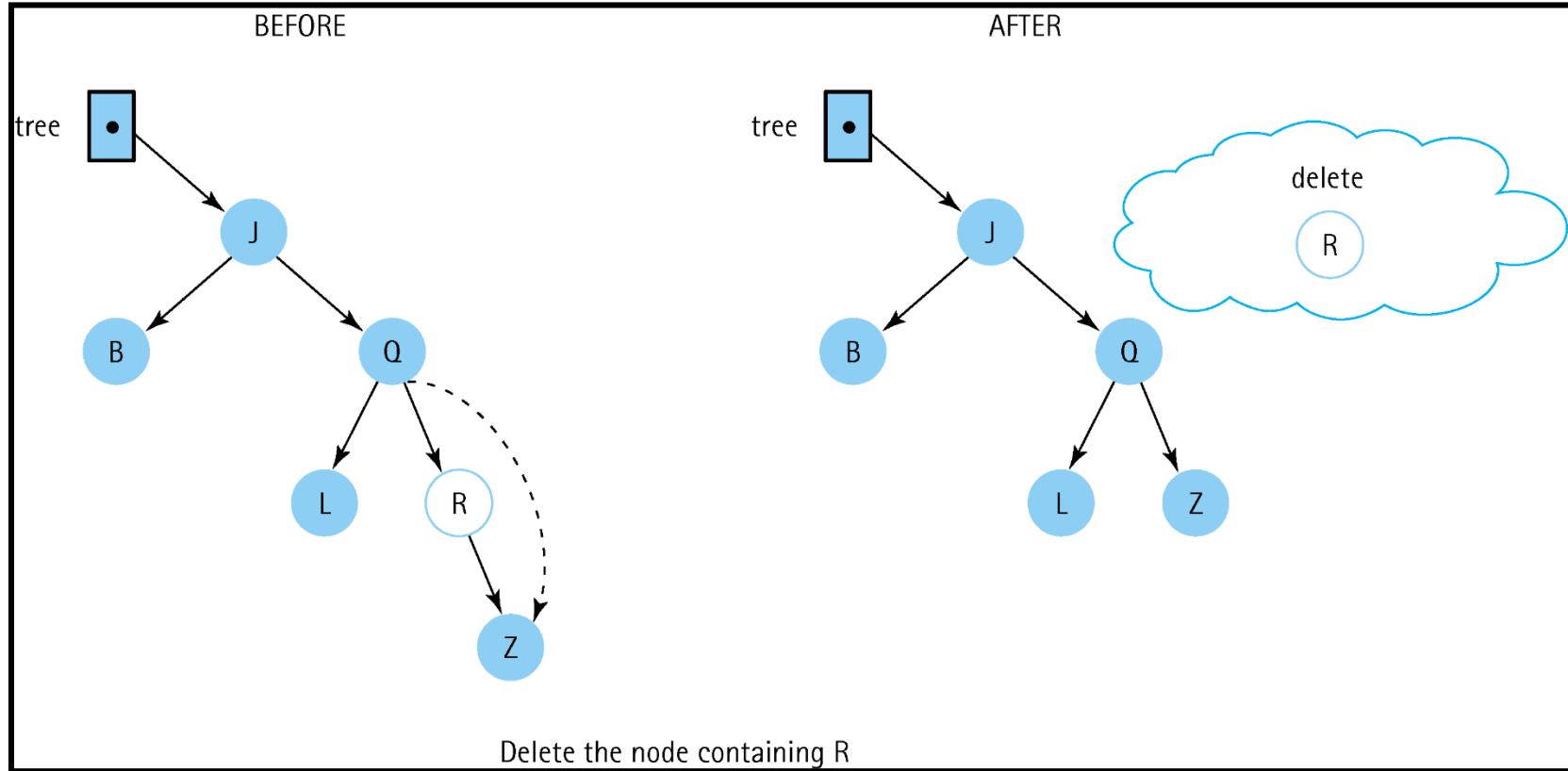


Deleting a Leaf Node

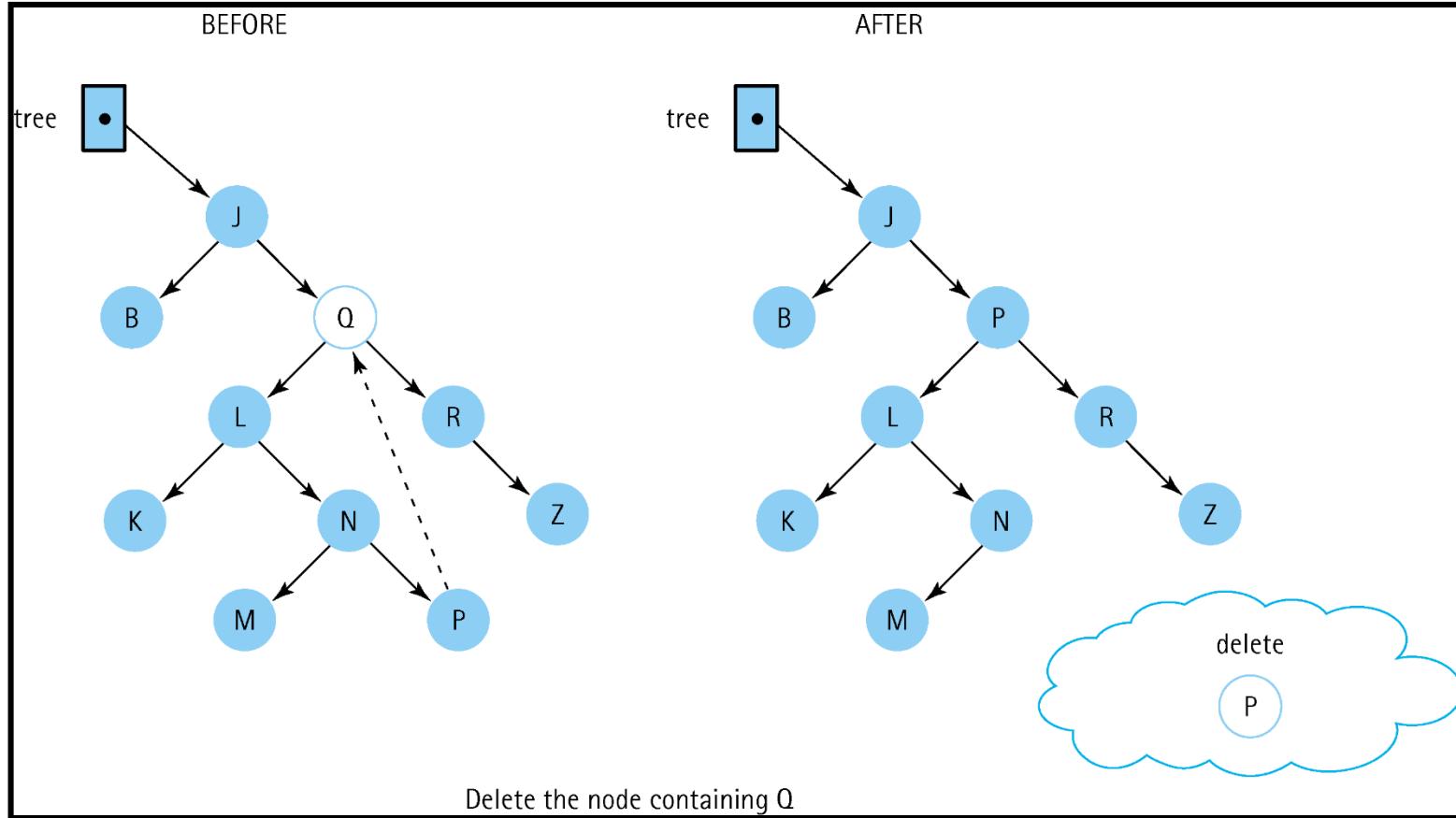




Deleting a Node with One Child



Deleting a Node with Two Children





Deleting a Node with Two Children (cont'd)

- Find predecessor (it is the rightmost node in the left subtree)
- Replace the data of the node to be deleted with predecessor's data
- Delete predecessor node

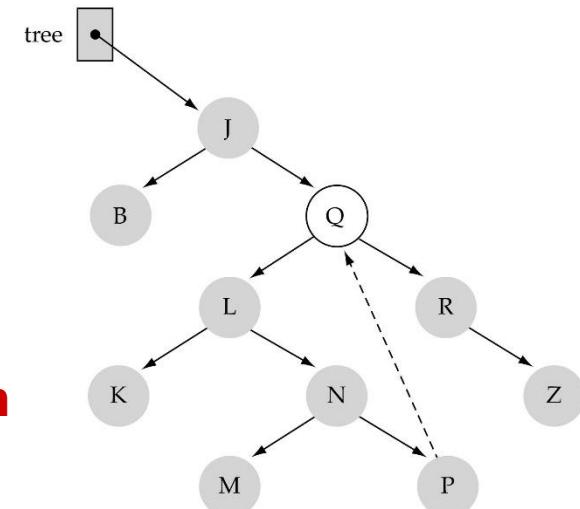
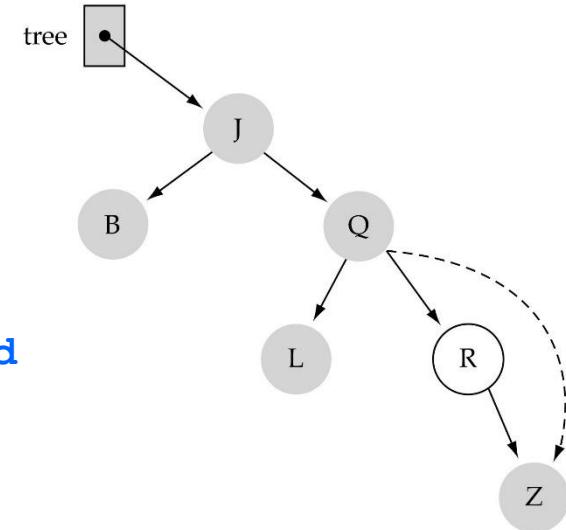


DeleteNode Algorithm

```
if (Left(tree) is NULL) AND (Right(tree) is NULL)
    Set tree to NULL
else if Left(tree) is NULL
    Set tree to Right(tree)
else if Right(tree) is NULL
    Set tree to Left(tree)
else
    Find predecessor
    Set Info(tree) to Info(predecessor)
    Delete predecessor
```

Code for DeleteNode

```
void DeleteNode(TreeNode*& tree)
{
    ItemType data;
    TreeNode* tempPtr;
    tempPtr = tree;
    if (tree->left == NULL) { //right child
        tree = tree->right;
        delete tempPtr; }      0 or 1 child
    else if (tree->right == NULL) { //left child
        tree = tree->left;
        delete tempPtr; }      0 or 1 child
    else{
        GetPredecessor(tree->left, data);
        tree->info = data;
        Delete(tree->left, data); } 2 children
}
```





Definition of Recursive Delete

Definition: Removes item from tree

Size: The number of nodes in the path from the root to the node to be deleted.

Base Case: If item's key matches key in Info(tree),
delete node pointed to by tree.

General Case: If item < Info(tree),
Delete(Left(tree), item);
else
Delete(Right(tree), item).



Code for Recursive Delete

```
void Delete(TreeNode*& tree, ItemType item)
{
    if (item < tree->info)
        Delete(tree->left, item) ;
    else if (item > tree->info)
        Delete(tree->right, item) ;
    else
        DeleteNode(tree); // Node found
}
```



Code for GetPredecessor

```
void GetPredecessor(TreeNode* tree,
    ItemType& data)
{
    while (tree->right != NULL)
        tree = tree->right;
    data = tree->info;
}
```

Why is the code not recursive?



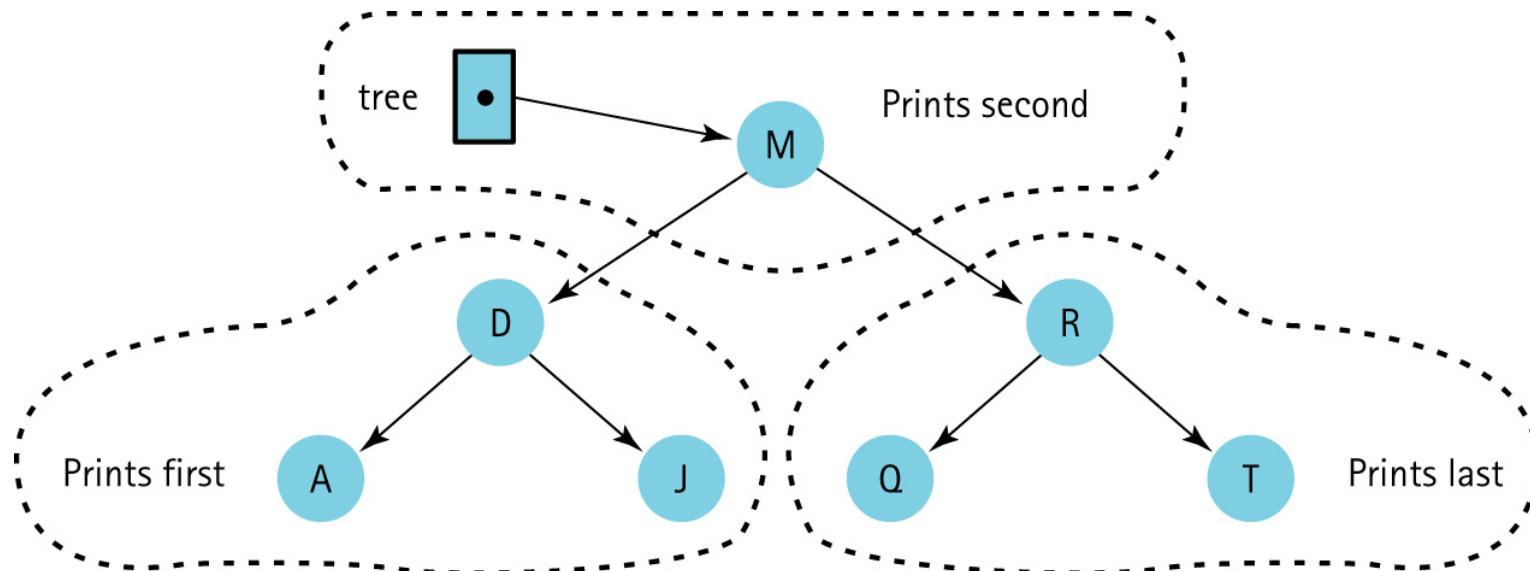
Third Edition

C++ Plus Data Structures

Nell Dale

Chapter
8-2
Binary
Search Trees

Printing all the Nodes in Order





Function Print

Function Print

Definition: Prints the items in the binary search tree in order from smallest to largest.

Size: The number of nodes in the tree whose root is tree

Base Case: If tree = NULL, do nothing.

General Case: Traverse the left subtree in order.
Then print Info(tree).
Then traverse the right subtree in order.



Code for Recursive InOrder Print

```
void PrintTree(TreeNode* tree,  
    std::ofstream& outFile)  
{  
    if (tree != NULL)  
    {  
        PrintTree(tree->left, outFile);  
        outFile << tree->info;  
        PrintTree(tree->right, outFile);  
    }  
}
```

Is that all there is?

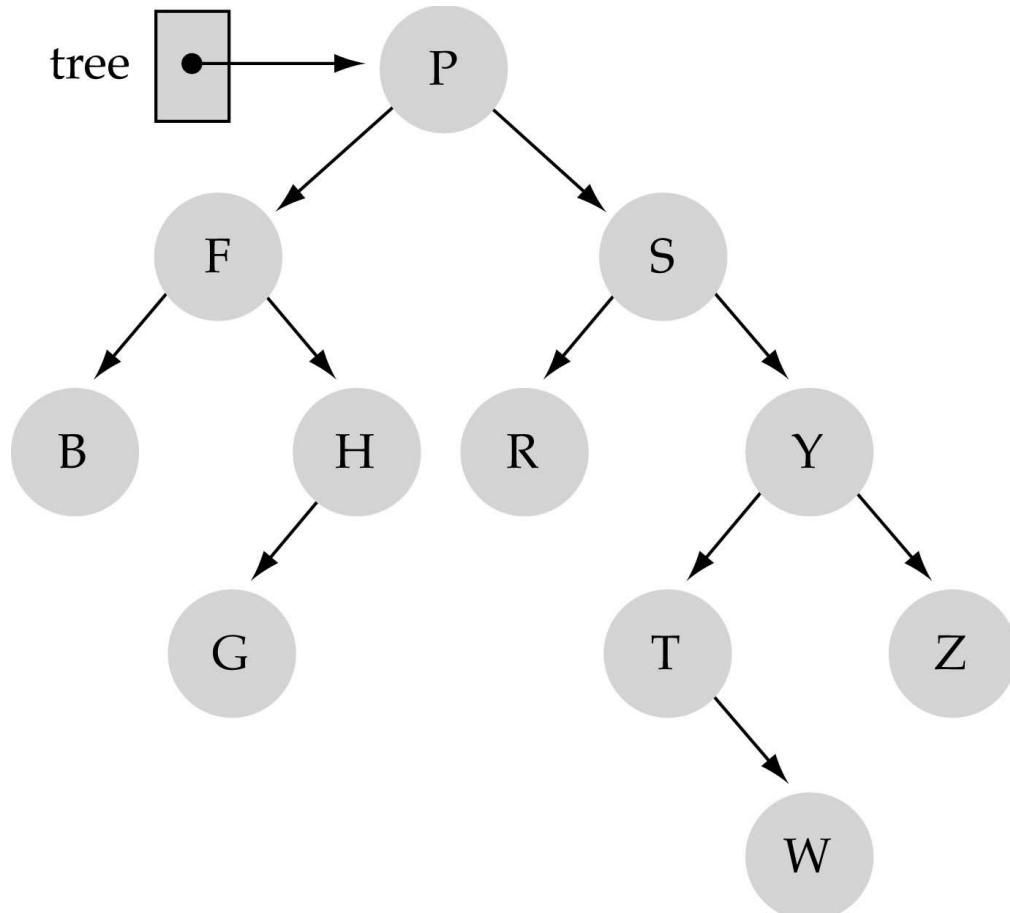


Class Constructor

```
template<class ItemType>
TreeType<ItemType>::TreeType ()
{
    root = NULL;
}
```



Class Destructor



How should we delete the nodes of a tree?

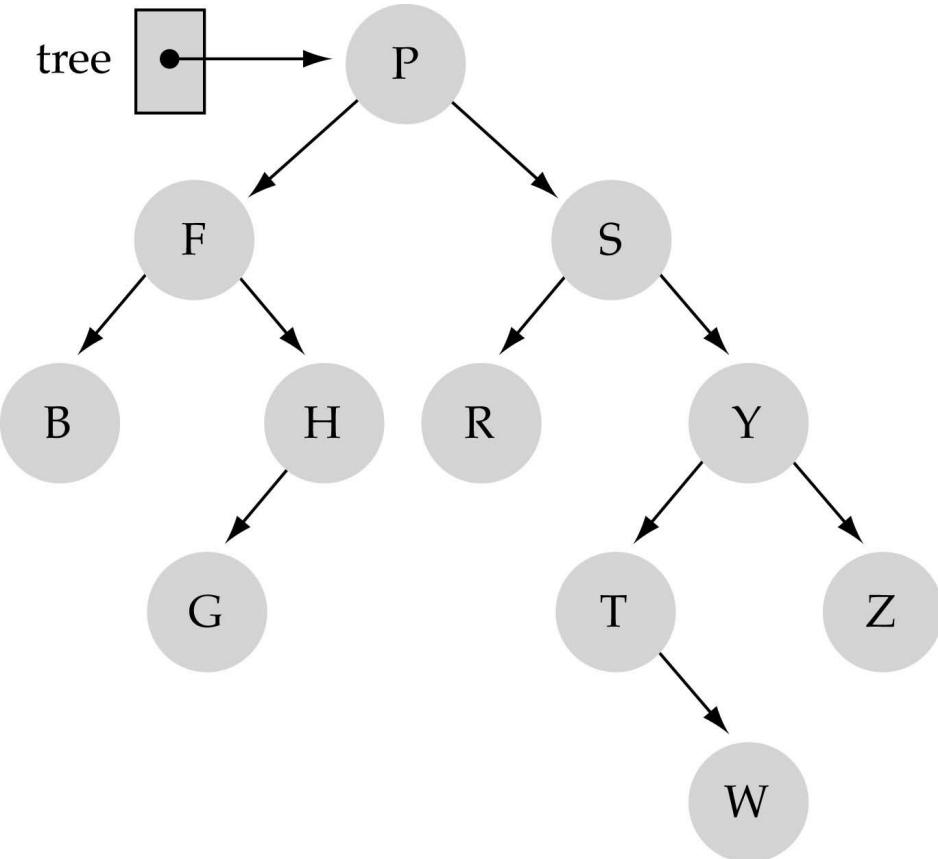


Destructor

```
void Destroy(TreeNode*& tree) ;  
TreeType::~TreeType()  
{  
    Destroy(root) ;  
}  
  
void Destroy(TreeNode*& tree)  
{  
    if (tree != NULL)  
    {  
        Destroy(tree->left) ;  
        Destroy(tree->right) ;  
        delete tree ;  
    }  
}
```



Copy Constructor



How should we
create a copy of
a tree?



Algorithm for Copying a Tree

```
if (originalTree is NULL)
    Set copy to NULL
else
    Set Info(copy) to Info(originalTree)
    Set Left(copy) to Left(originalTree)
    Set Right(copy) to Right(originalTree)
```

What traversal order do we use?



Code for CopyTree

```
TreeType::TreeType(const TreeType& originalTree)
{
    CopyTree(root, originalTree.root);
}

void CopyTree(TreeNode*& copy,
              const TreeNode* originalTree)
{
    if (originalTree == NULL)
        copy = NULL;
    else
    {
        copy = new TreeNode;
        copy->info = originalTree->info;
        CopyTree(copy->left, originalTree->left);
        CopyTree(copy->right, originalTree->right);
    }
}
```



Tree Traversal

- A tree traversal means visiting all the nodes in the tree
- “visit” means that the algorithm does something with the values in the node, e.g., print the value



Tree Traversal Methods

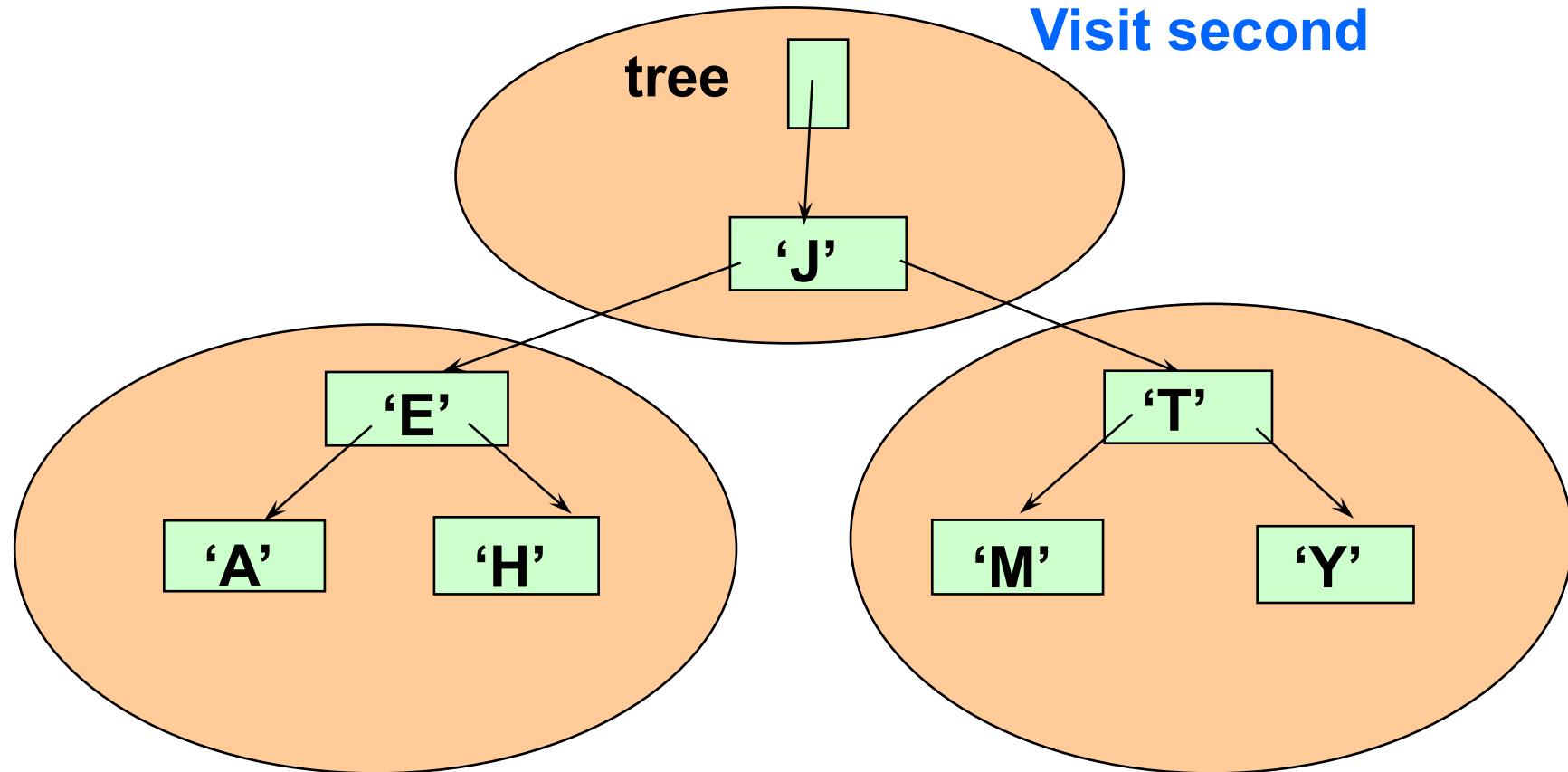
There are mainly three ways to traverse a tree:

- 1) Inorder Traversal
- 2) Postorder Traversal
- 3) Preorder Traversal



Inorder Traversal

Inorder Traversal: A E H J M T Y



Visit left subtree first

Visit right subtree last



Inorder(tree)

if tree is not NULL

Inorder(Left(tree))

Visit Info(tree)

Inorder(Right(tree))

To print in alphabetical order



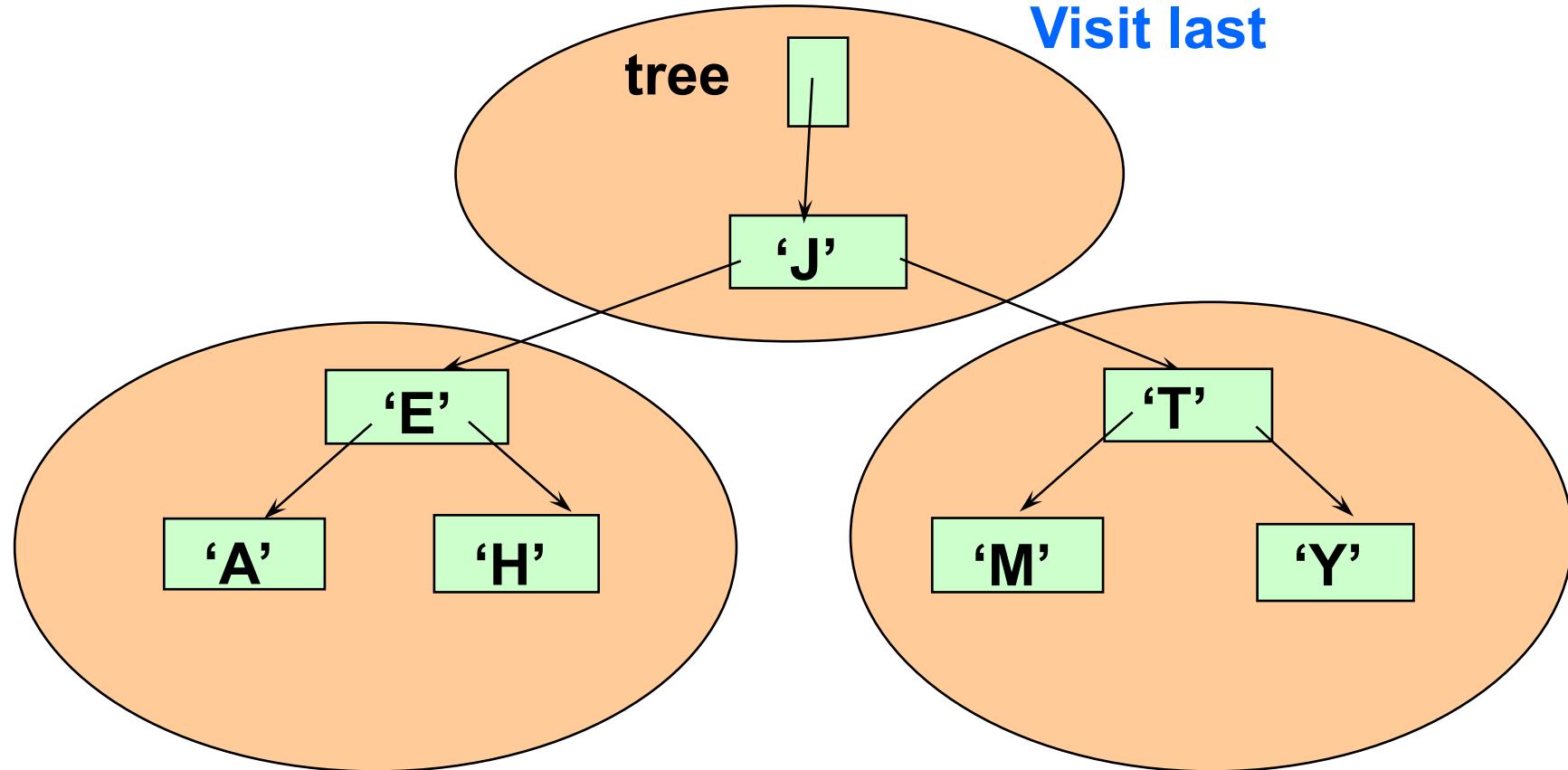
InOrder Implementation

```
void InOrder(TreeNode* tree,
QueType& inQue)
{
    if(tree != NULL) {
        InOrder(tree->left, inQue);
        inQue.Enqueue(tree->info);
        InOrder(tree->right, inQue);
    }
}
```



Postorder Traversal

Postorder Traversal: A H E M Y T J



Visit left subtree first

Visit right subtree second



Postorder(tree)

if tree is not NULL

 Postorder(Left(tree))

 Postorder(Right(tree))

 Visit Info(tree)

*Visits leaves first
(good for deletion)*



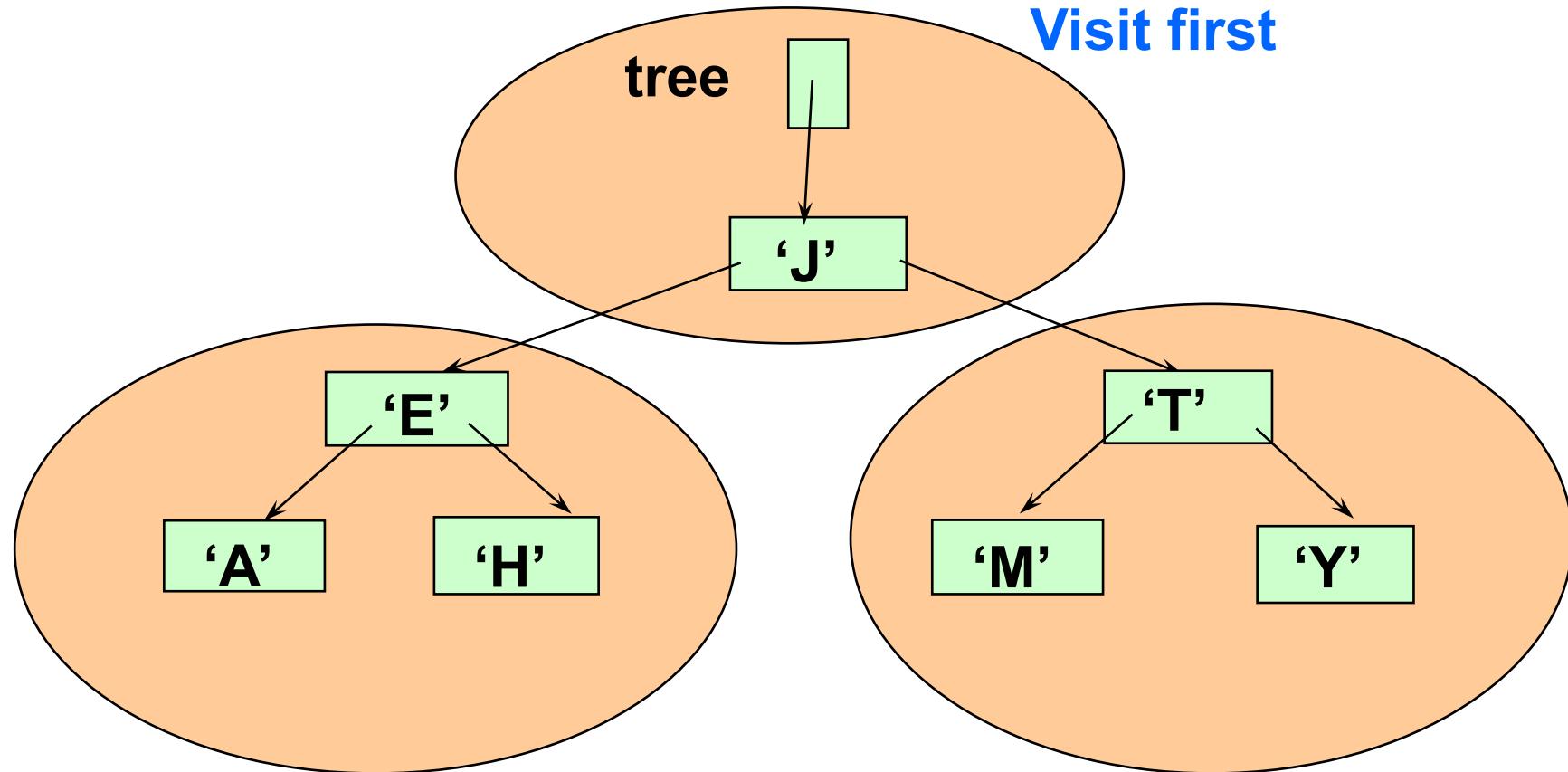
PostOrder Implementation

```
void PostOrder(TreeNode *tree,
    QueType& postQue)
{
    if(tree != NULL) {
        PostOrder(tree->left, postQue);
        PostOrder(tree->right, postQue);
        postQue.Enqueue(tree->info);
    }
}
```



Preorder Traversal

Preorder Traversal: J E A H T M Y



Visit left subtree second

Visit right subtree last



Preorder(tree)

if tree is not NULL

 Visit Info(tree)

 Preorder(Left(tree))

 Preorder(Right(tree))

*Useful with binary trees
(not binary search trees)*

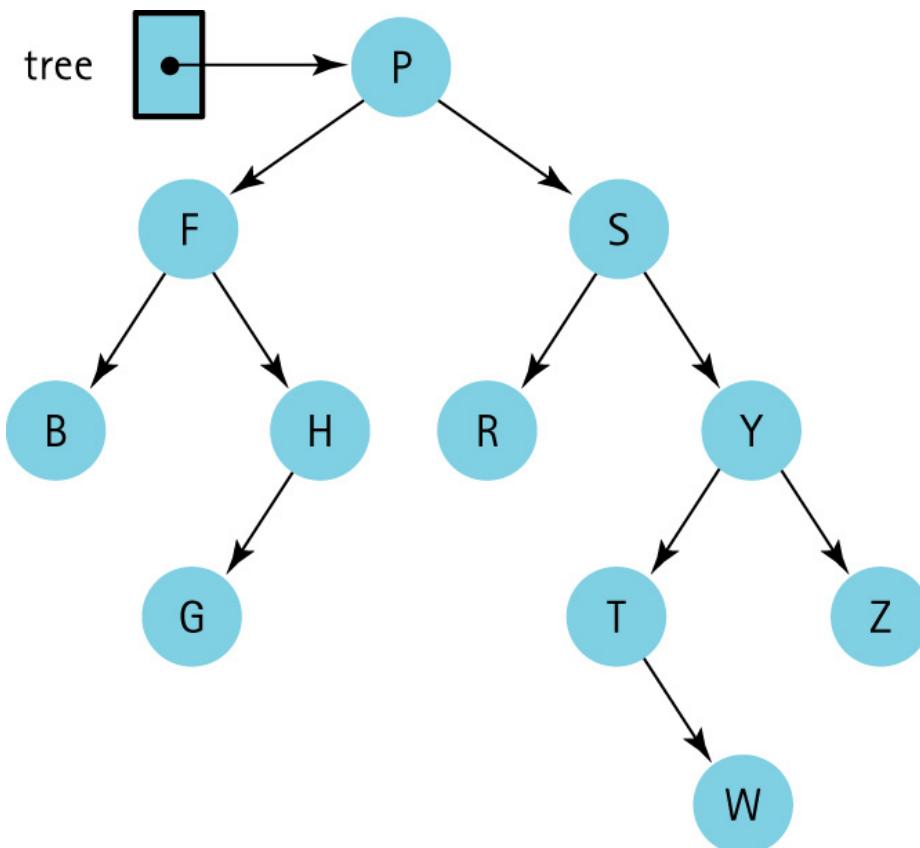


PreOrder Implementation

```
void PreOrder(TreeNode* tree,
    QueType& preQue)
{
    if(tree != NULL) {
        preQue.Enqueue(tree->info);
        PreOrder(tree->left, preQue);
        PreOrder(tree->right, preQue);
    }
}
```



Three Tree Traversals



Inorder: B F G H P R S T W Y Z

Preorder: P F B H G S R Y T W Z

Postorder: B G H F R W T Z Y S P



Our Iteration Approach

- The client program passes the ResetTree and GetNextItem functions a parameter indicating which of the three traversals to use
- For efficiency, ResetTree generates a queue of node contents in the indicated order
- GetNextItem processes the node contents from the appropriate queue: inQue, preQue, postQue.



Modification of Class TreeType

```
enum OrderType { PRE_ORDER, IN_ORDER,
    POST_ORDER } ;

class TreeType {
public:
    // same as before
private:
    TreeNode* root;
    QueType preQue;           new private data
    QueType inQue;
    QueType postQue;
};
```



Code for ResetTree

```
void TreeType::ResetTree(OrderType order)
// Calls function to create a queue of the tree
// elements in the desired order.
{
    switch (order)
    {
        case PRE_ORDER : PreOrder(root, preQue);
                          break;
        case IN_ORDER  : InOrder(root, inQue);
                          break;
        case POST_ORDER: PostOrder(root, postQue);
                          break;
    }
}
```



Code for GetNextItem

```
void TreeType::GetNextItem(ItemType& item,
    OrderType order, bool& finished)
{
    finished = false;
    switch (order)
    {
        case PRE_ORDER : preQue.Dequeue(item);
                          if (preQue.IsEmpty())
                              finished = true;
                          break;
        case IN_ORDER  : inQue.Dequeue(item);
                          if (inQue.IsEmpty())
                              finished = true;
                          break;
        case POST_ORDER: postQue.Dequeue(item);
                          if (postQue.IsEmpty())
                              finished = true;
                          break;
    }
}
```



Prototypes of Traversal Functions

```
void PreOrder(TreeNode* ,  
QueType&);
```

```
void InOrder(TreeNode* ,  
QueType&);
```

```
void PostOrder(TreeNode* ,  
QueType&);
```



Iterative Versions

FindNode

Set nodePtr to tree

Set parentPtr to NULL

Set found to false

while more elements to search AND NOT found

if item < Info(nodePtr)

 Set parentPtr to nodePtr

 Set nodePtr to Left(nodePtr)

else if item > Info(nodePtr)

 Set parentPtr to nodePtr

 Set nodePtr to Right(nodePtr)

else

 Set found to true



```
void FindNode(TreeNode* tree, ItemType item,
    TreeNode*& nodePtr, TreeNode*& parentPtr)
{
    nodePtr = tree;
    parentPtr = NULL;
    bool found = false;
    while (nodePtr != NULL && !found)
    { if (item < nodePtr->info)
        {
            parentPtr = nodePtr;
            nodePtr = nodePtr->left;
        }
        else if (item > nodePtr->info)
        {
            parentPtr = nodePtr;
            nodePtr = nodePtr->right;
        }
        else found = true;
    }
}
```

Code for FindNode



InsertItem

Create a node to contain the new item.

Find the insertion place.

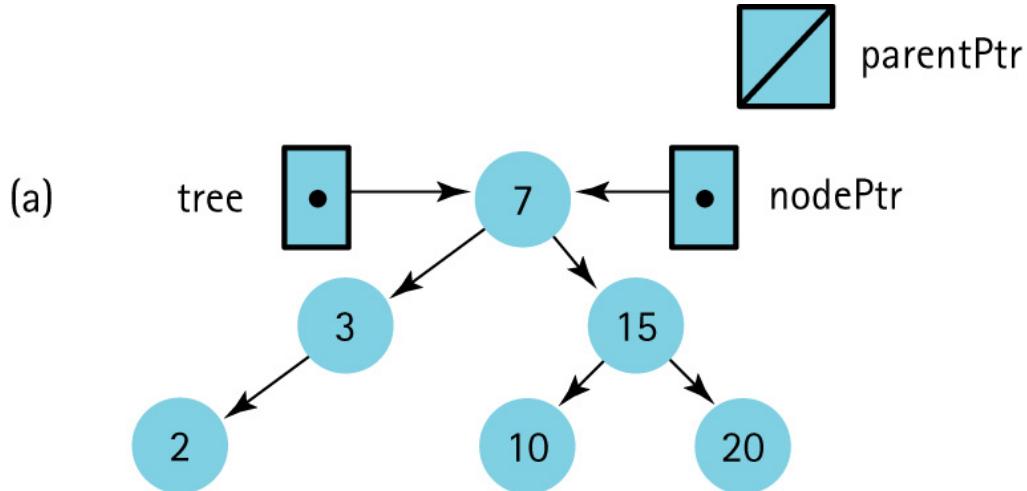
Attach new node.

Find the insertion place

FindNode(tree, item, nodePtr, parentPtr);

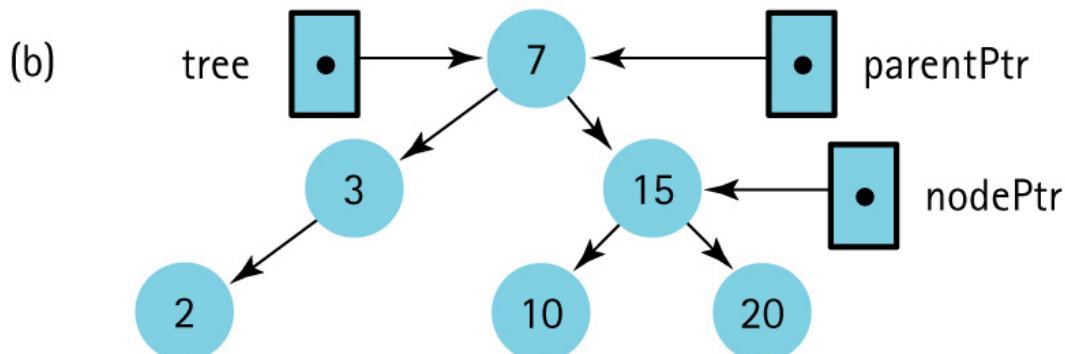


Using function FindNode to find the insertion point (Insert 13)



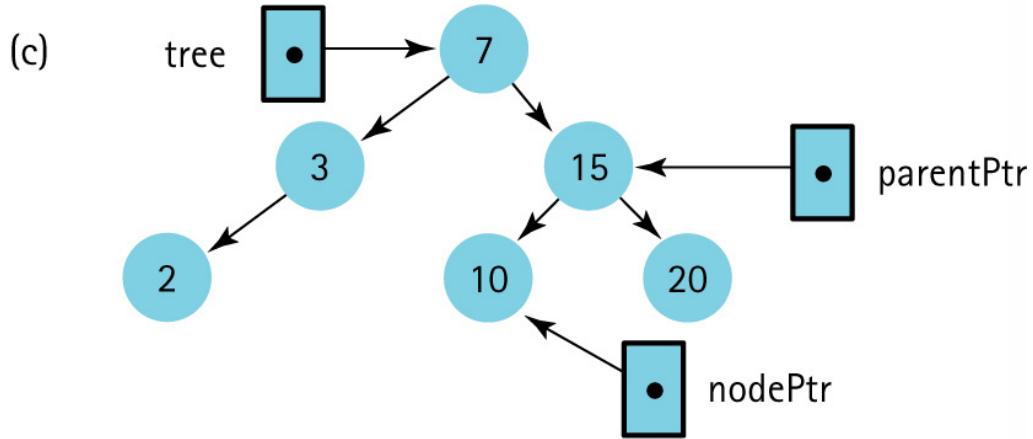


Using function FindNode to find the insertion point (Insert 13)



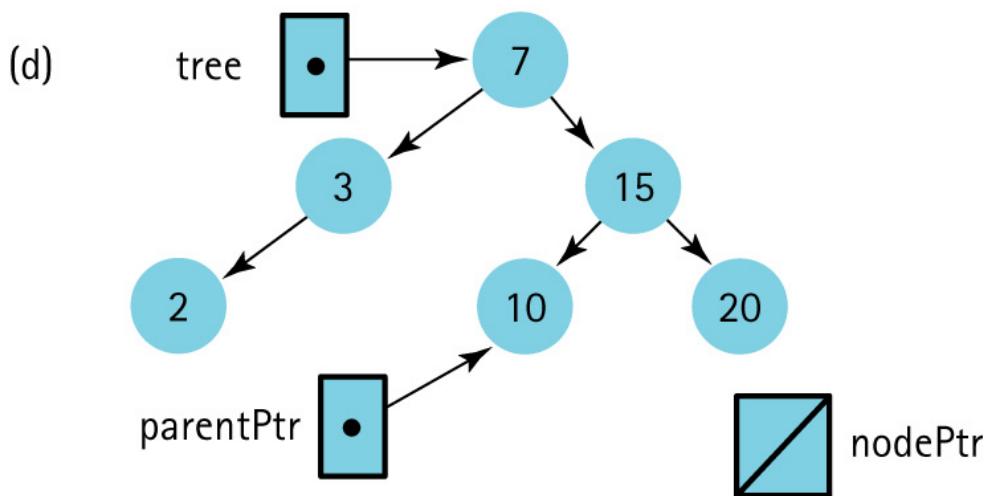


Using function FindNode to find the insertion point (Insert 13)



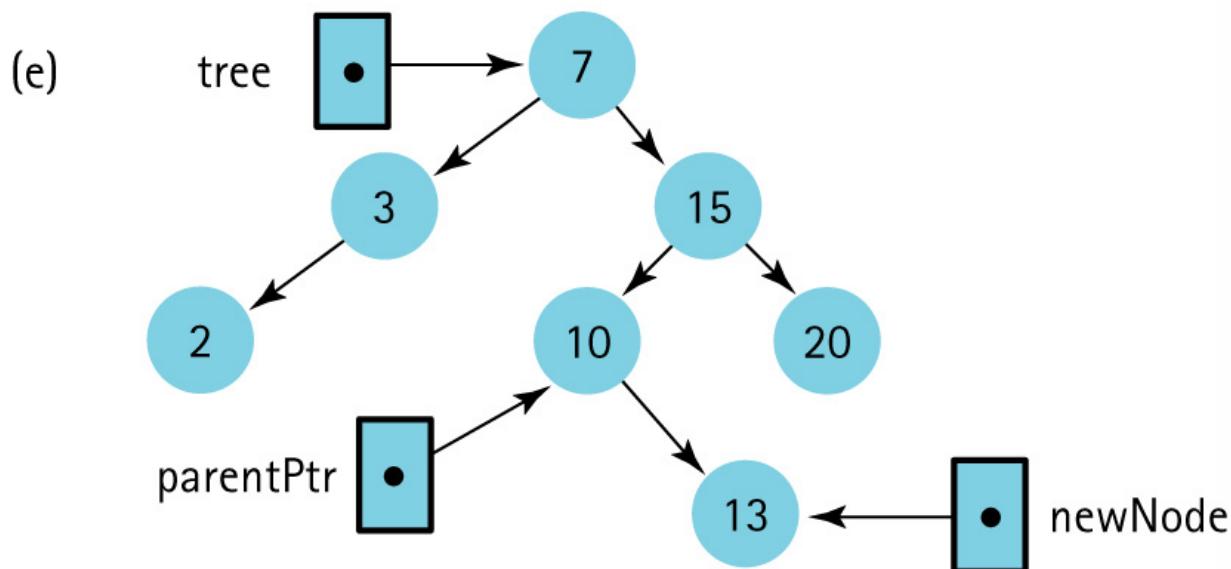


Using function FindNode to find the insertion point (Insert 13)





Using function FindNode to find the insertion point (Insert 13)





AttachNewNode

```
if item < Info(parentPtr)
    Set Left(parentPtr) to newNode
else
    Set Right(parentPtr) to newNode
```

What's wrong?



AttachNewNode(revised)

```
if parentPtr equals NULL  
    Set tree to newNode  
else if item < Info(parentPtr)  
    Set Left(parentPtr) to newNode  
else  
    Set Right(parentPtr) to newNode
```



Code for InsertItem

```
void TreeType::InsertItem(ItemType item)
{
    TreeNode* newNode;
    TreeNode* nodePtr;
    TreeNode* parentPtr;
    newNode = new TreeNode;
    newNode->info = item;
    newNode->left = NULL;
    newNode->right = NULL;
    FindNode(root, item, nodePtr, parentPtr);
    if (parentPtr == NULL)
        root = newNode;
    else if (item < parentPtr->info)
        parentPtr->left = newNode;
    else parentPtr->right = newNode;
}
```

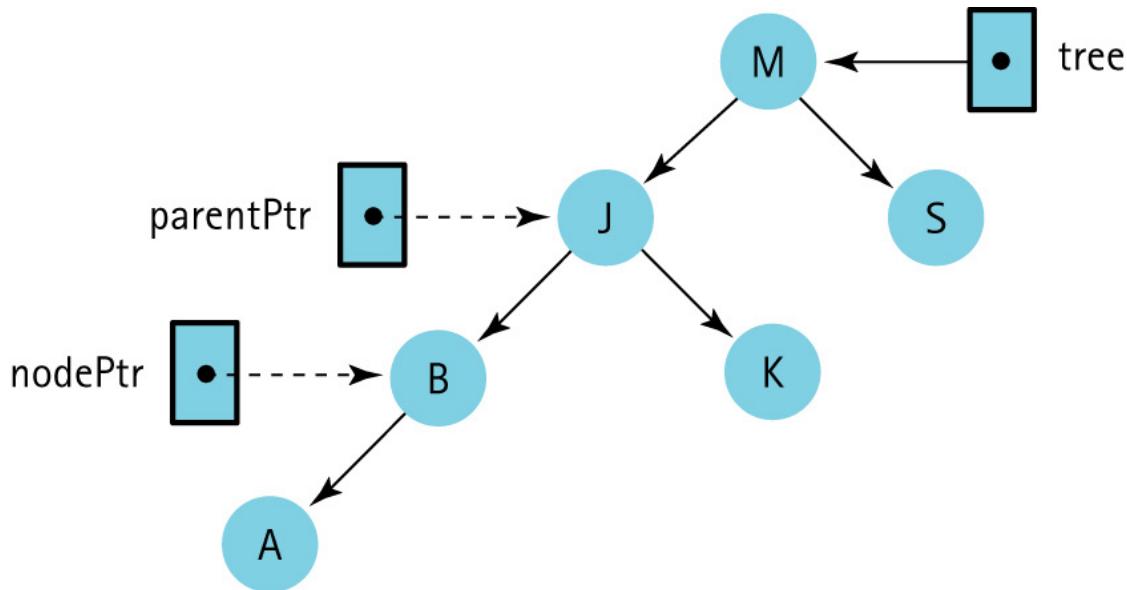


Code for DeleteItem

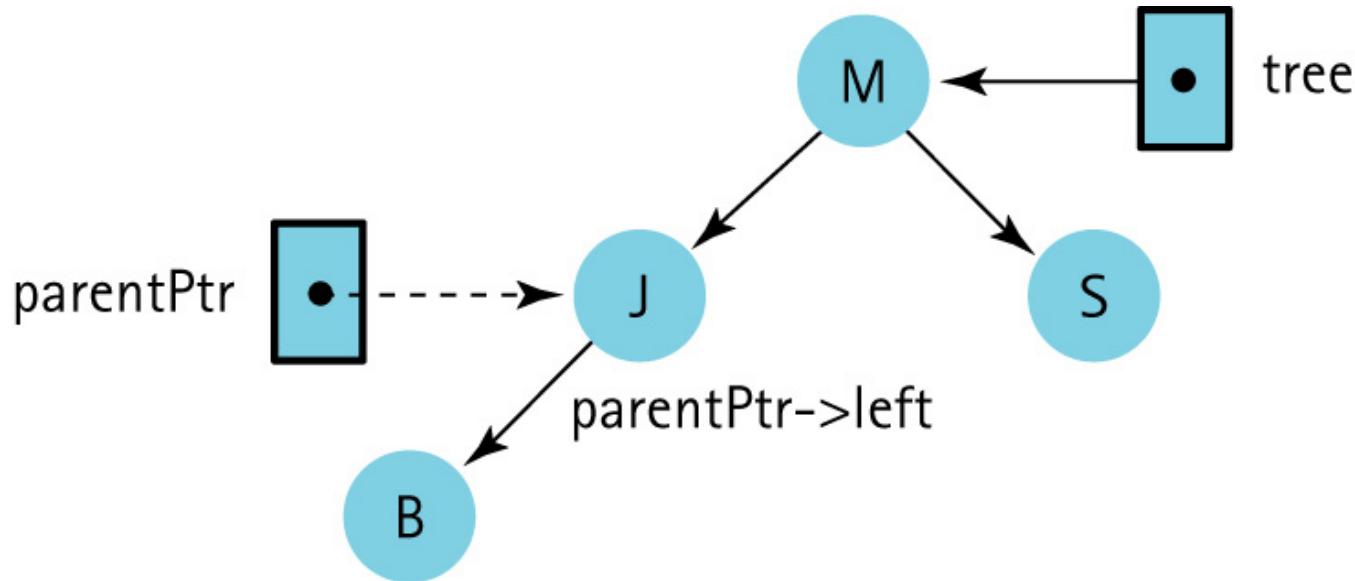
```
void TreeType::DeleteItem(ItemType item)
{
    TreeNode* nodePtr;
    TreeNode* parentPtr;
    FindNode(root, item, nodePtr, parentPtr);
    if (nodePtr == root)
        DeleteNode(root);
    else
        if (parentPtr->left == nodePtr)
            DeleteNode(parentPtr->left);
        else DeleteNode(parentPtr->right);
}
```



Pointers `nodePtr` and `parentPtr` Are External to the Tree



Pointer parentPtr is External to the Tree, but parentPtr-> left is an Actual Pointer in the Tree





Comparing Binary Search Trees to Linear Lists

Big-O Comparison

Operation	Binary Search Tree	Array-based List	Linked List
Constructor	$O(1)$	$O(1)$	$O(1)$
Destructor	$O(N)$	$O(1)$	$O(N)$
IsFull	$O(1)$	$O(1)$	$O(1)$
IsEmpty	$O(1)$	$O(1)$	$O(1)$
RetrieveItem	$O(\log N)$	$O(\log N)$	$O(N)$
InsertItem	$O(\log N)$	$O(N)$	$O(N)$
DeleteItem	$O(\log N)$	$O(N)$	$O(N)$

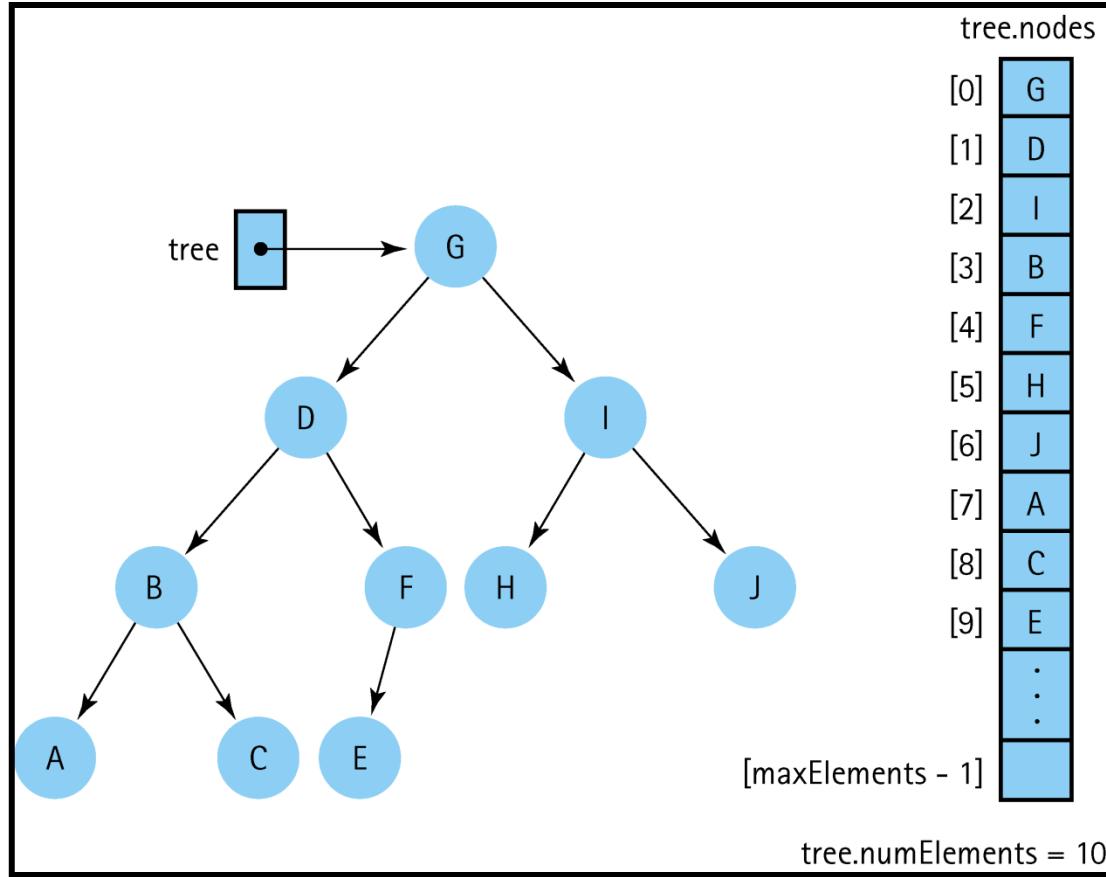


With Array Representation

- For any node `tree.nodes[index]`
 - its left child is in `tree.nodes[index*2 + 1]`
 - right child is in `tree.nodes[index*2 + 2]`
 - its parent is in `tree.nodes[(index - 1)/2]`.



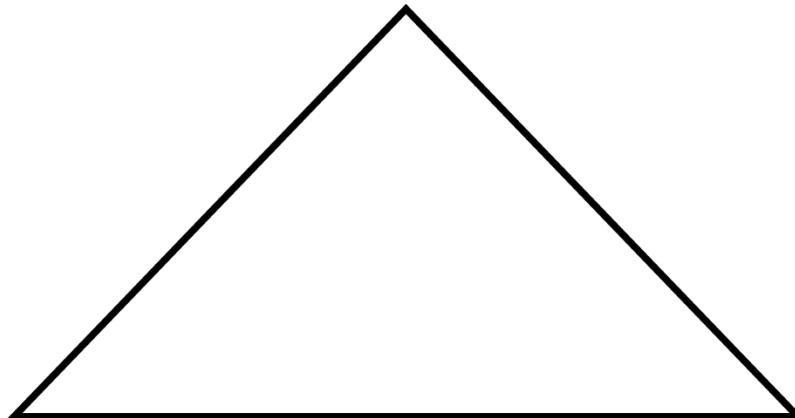
A Binary Tree and Its Array Representation





Definitions

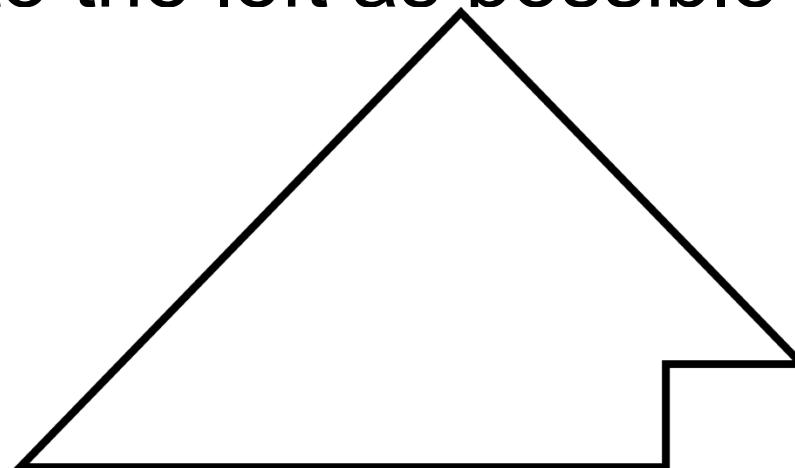
- **Full Binary Tree:** A binary tree in which all of the leaves are on the same level and every nonleaf node has two children





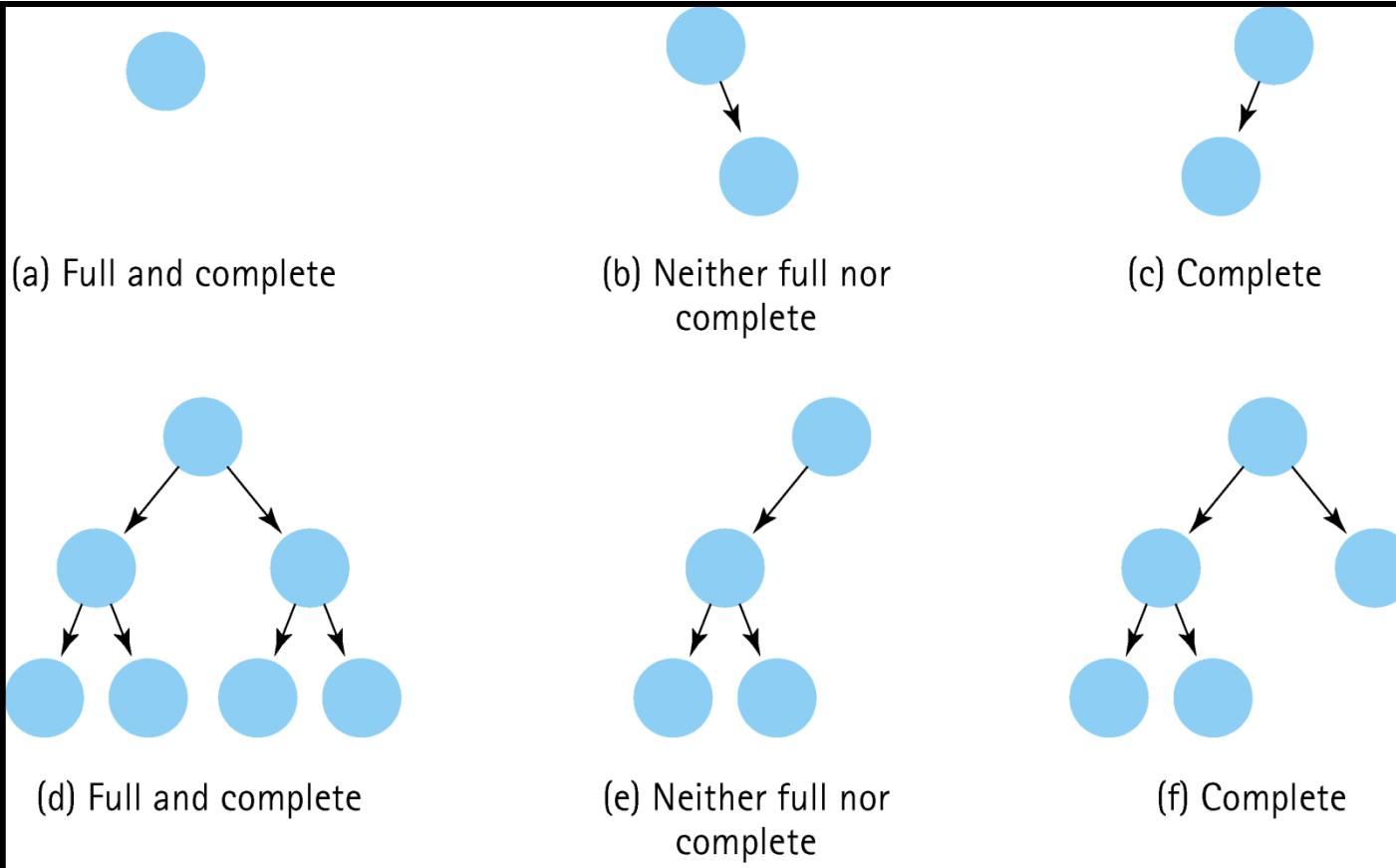
Definitions (cont.)

- **Complete Binary Tree:** A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible



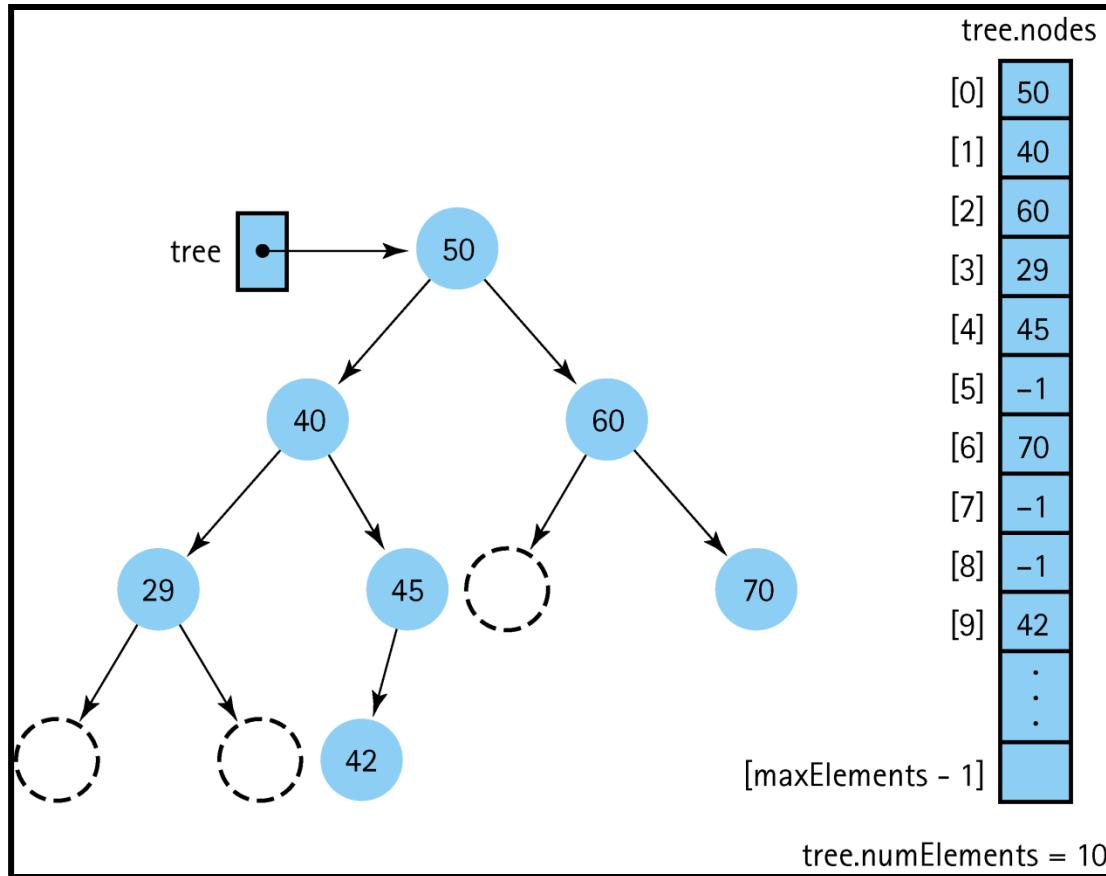


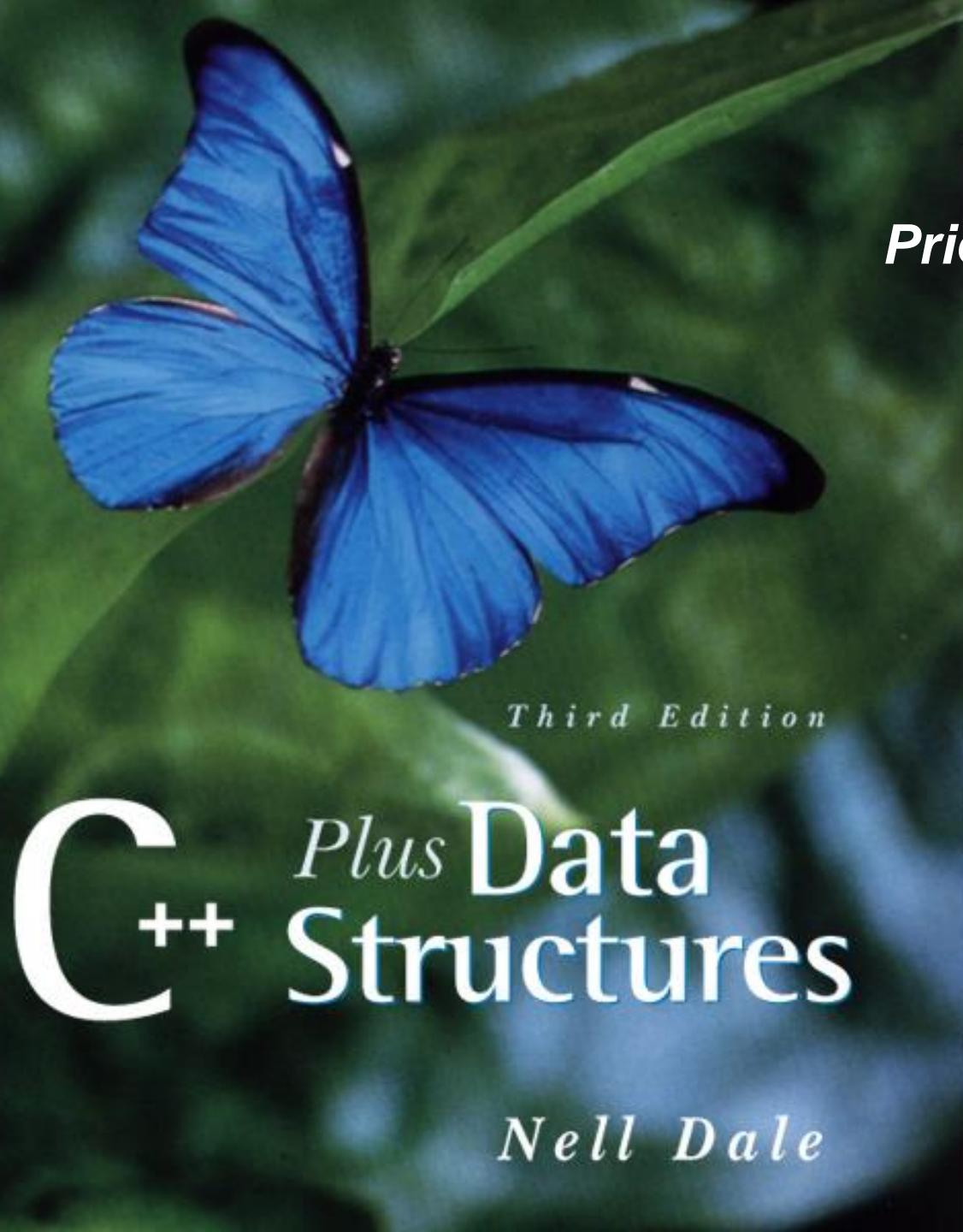
Examples of Different Types of Binary Trees





A Binary Search Tree Stored in an Array with Dummy Values





Chapter

9

*Priority Queues, Heaps,
and Graphs*



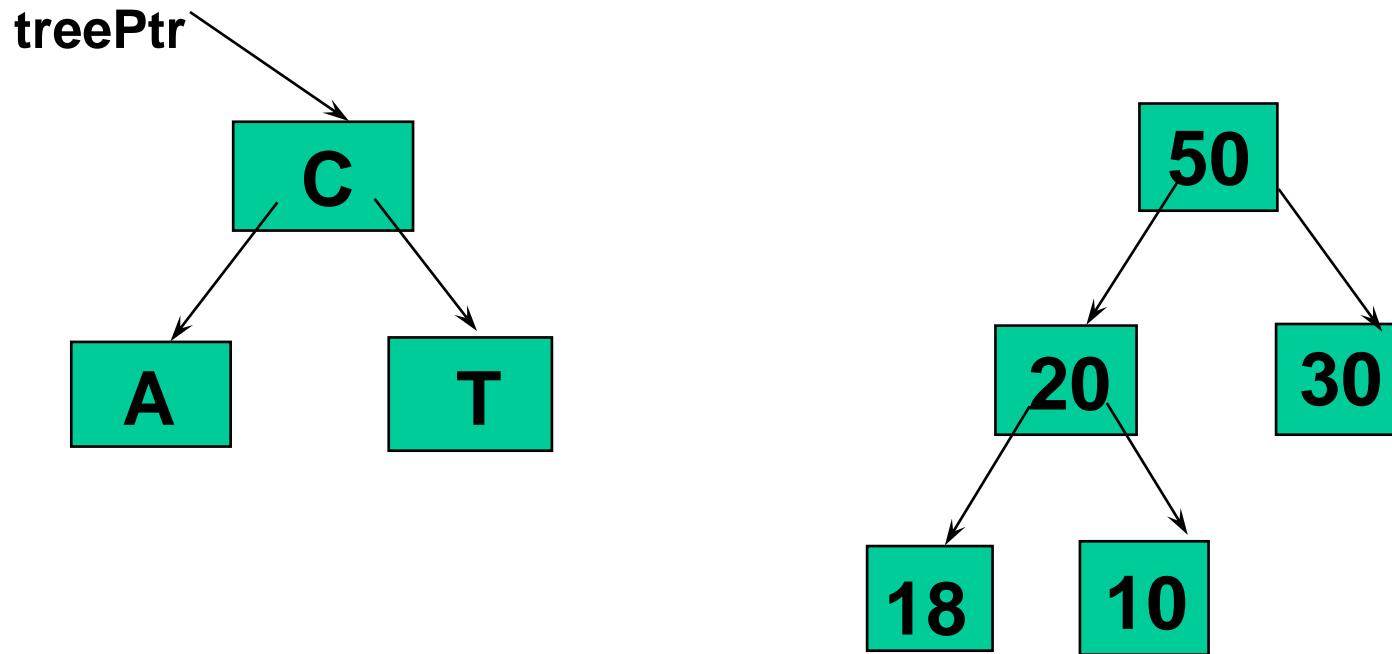
What is a Heap?

A heap is a binary tree that satisfies these special **SHAPE** and **ORDER** properties:

- **SHAPE** property: Its shape must be a complete binary tree.
- **ORDER** property: For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

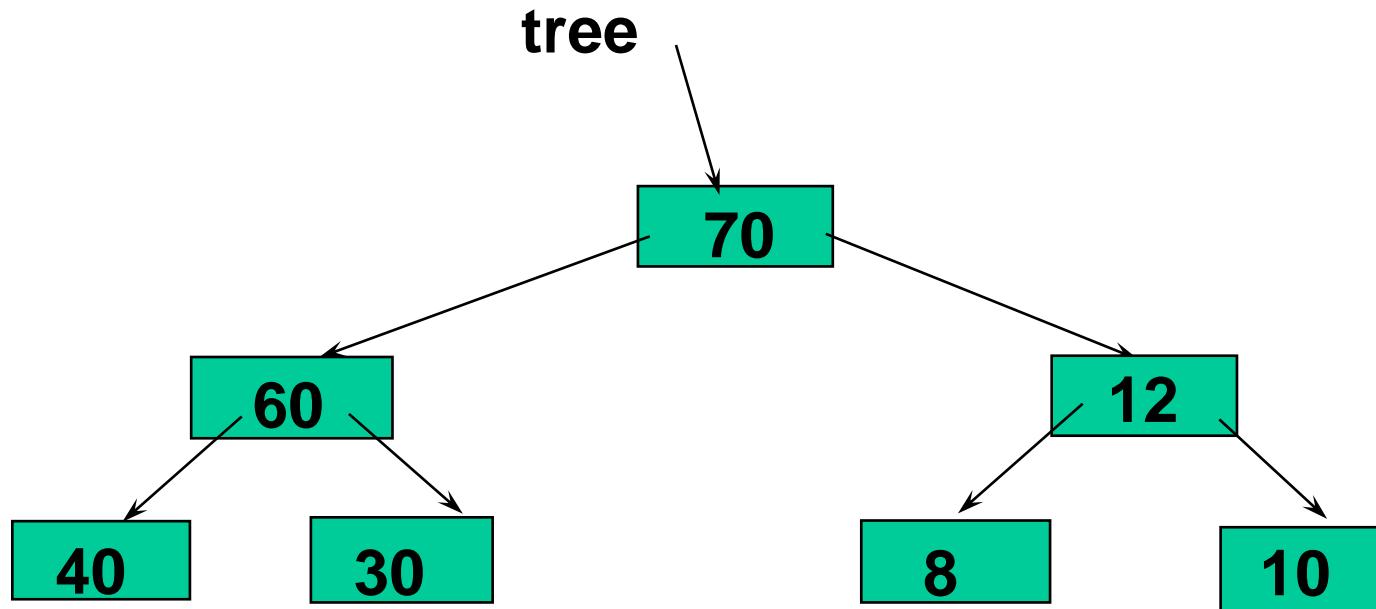


Are these Both Heaps?



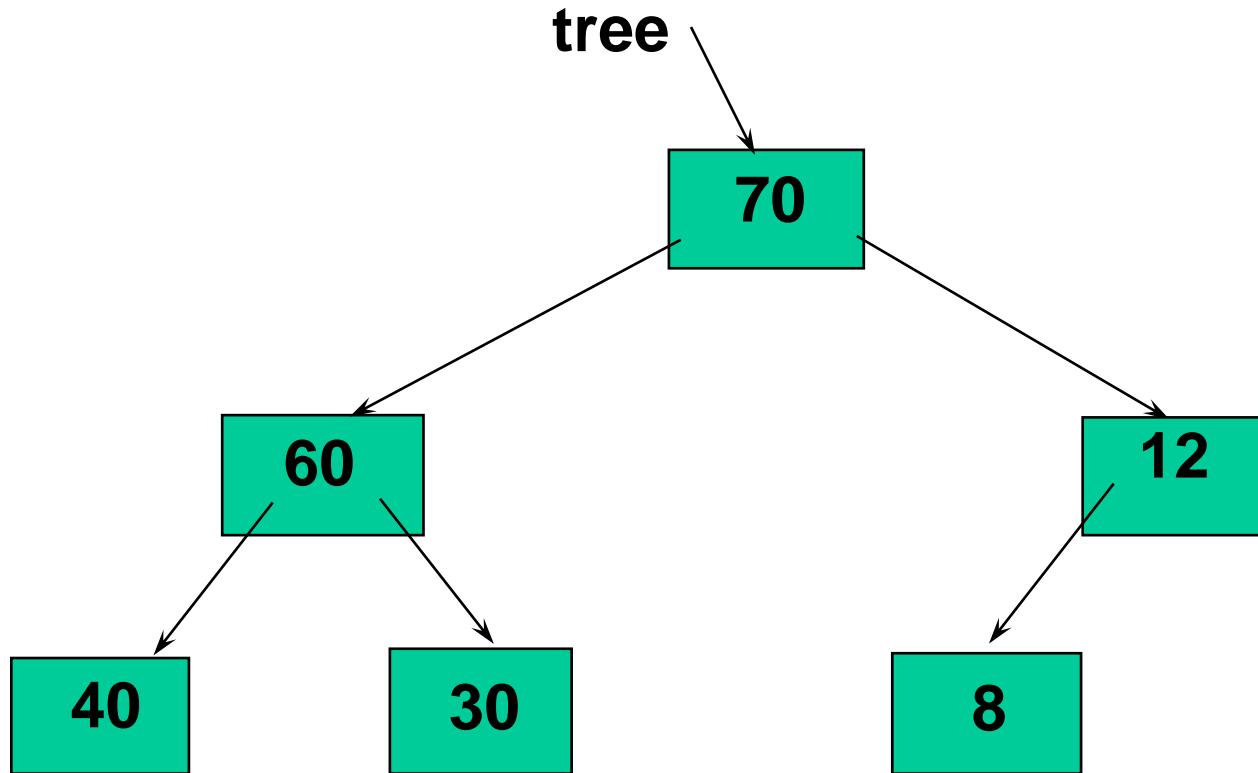


Is this a Heap?





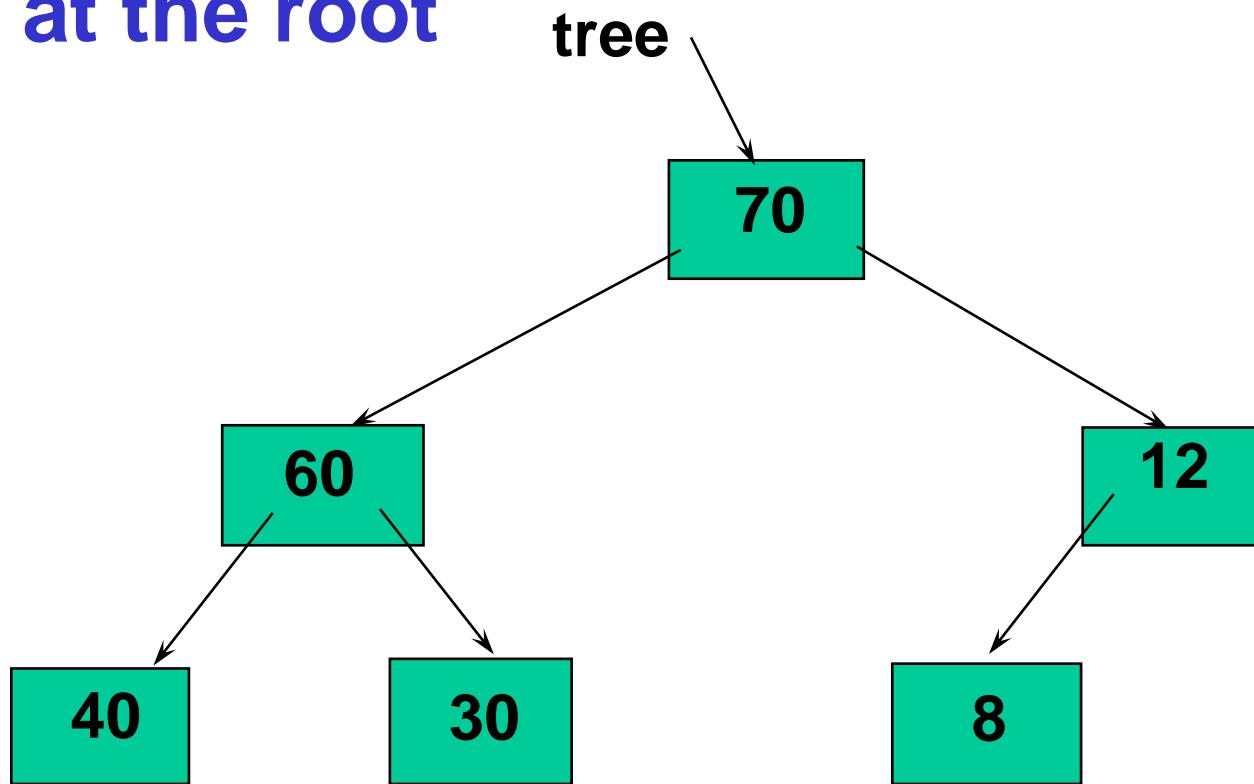
Where is the Largest Element in a Heap Always Found?





Where is the Largest Element in a Heap Always Found?

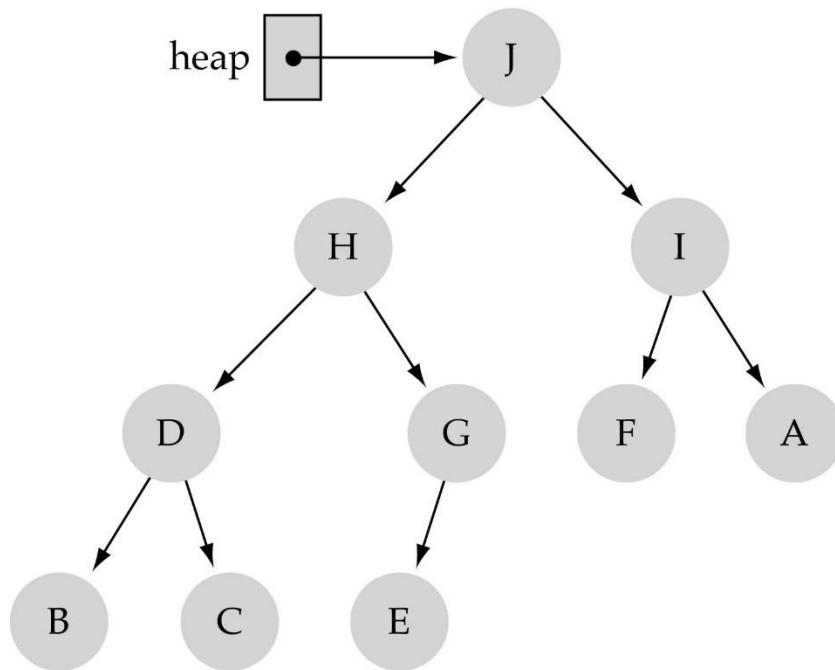
From *ORDER property*, the largest value of the heap is always stored at the root





Heap implementation using array representation

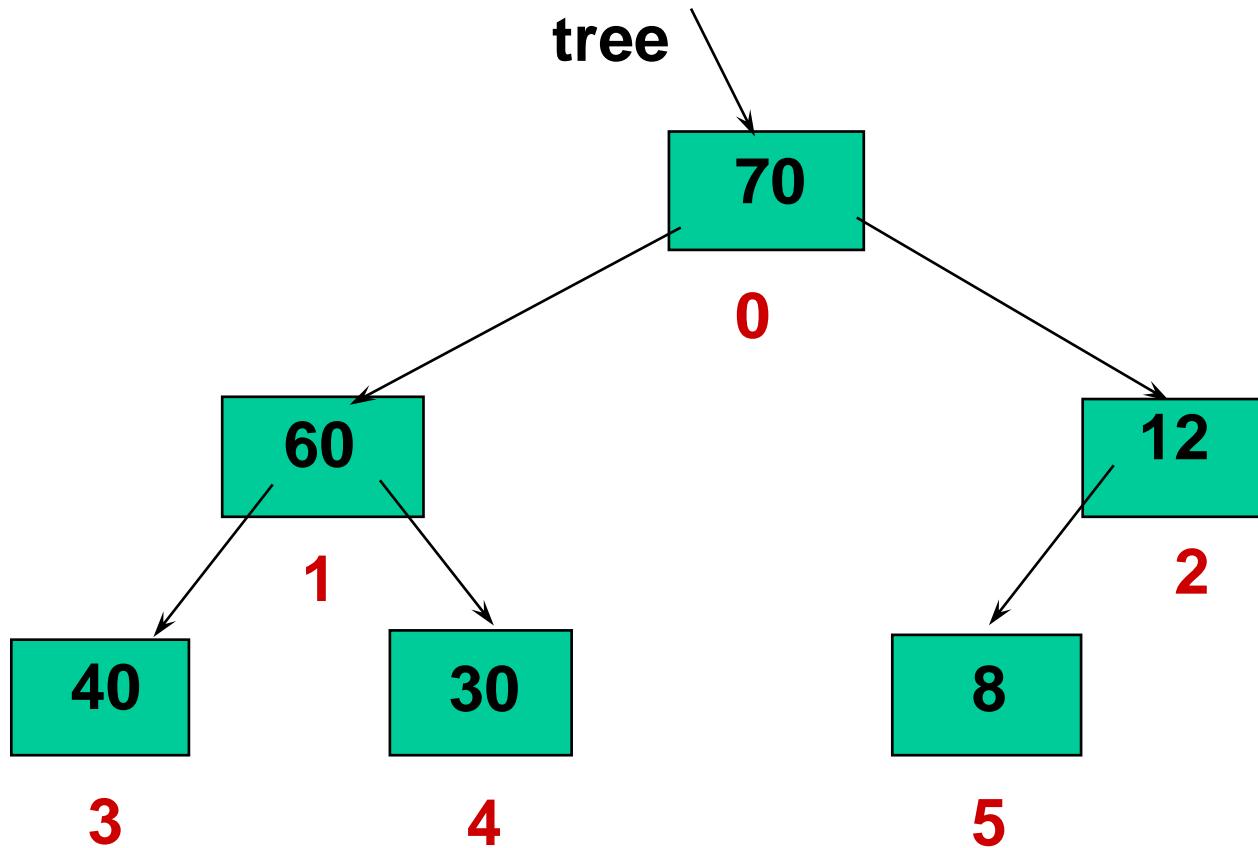
- A heap is a complete binary tree, so it is easy to be implemented using an array representation



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



We Can Number the Nodes Left to Right by Level This Way

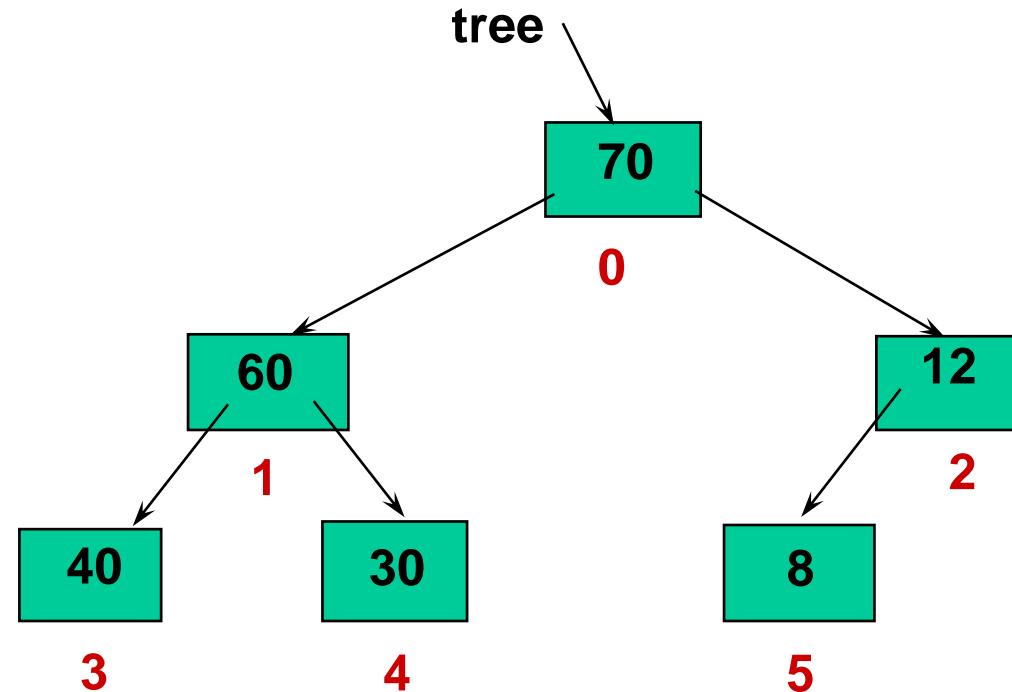




And use the Numbers as Array Indexes to Store the Trees

`tree.nodes`

[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	





```
// HEAP SPECIFICATION

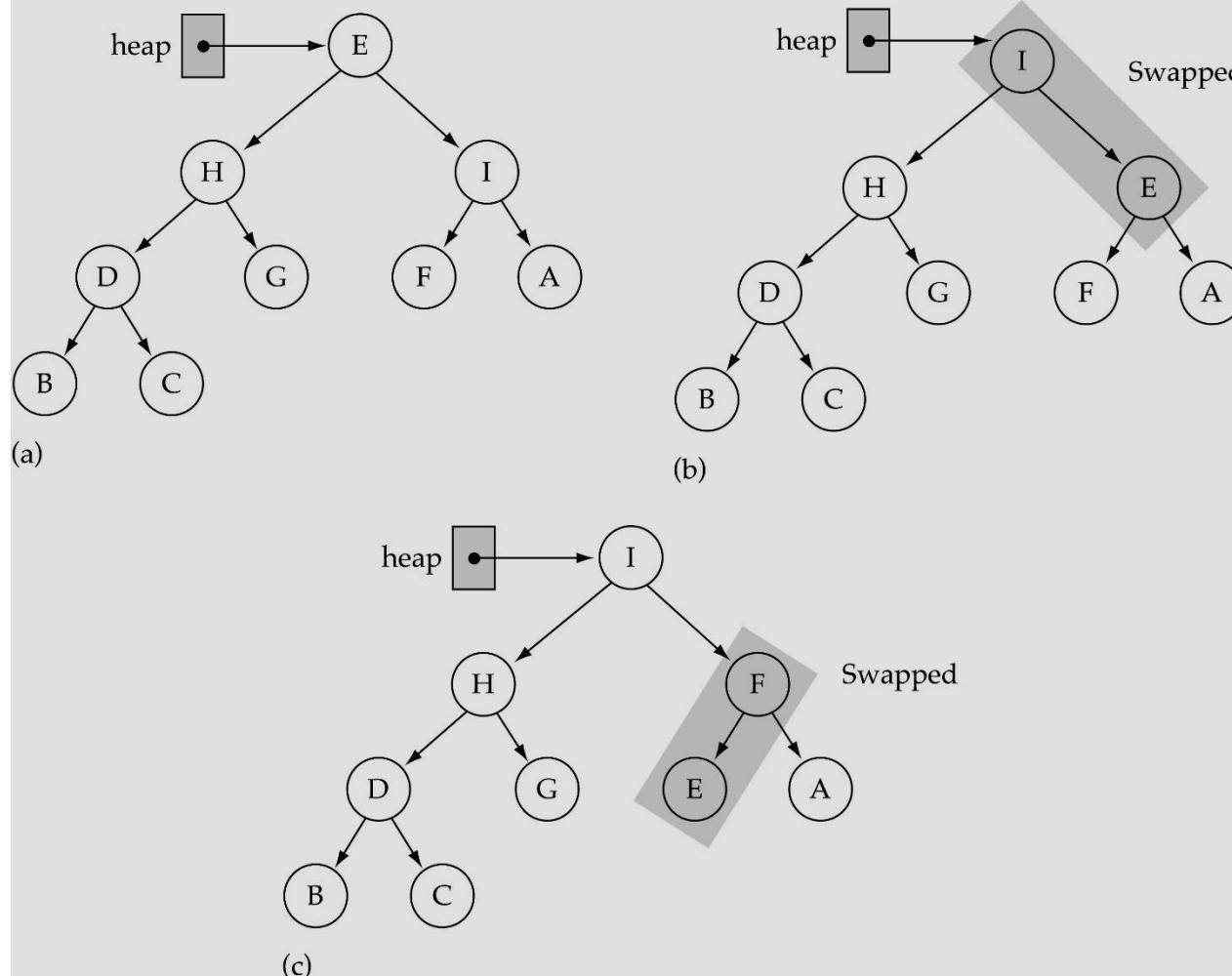
// Assumes ItemType is either a built-in simple data
// type or a class with overloaded relational operators.

template< class ItemType >
struct HeapType
{
    void ReheapDown ( int root , int bottom ) ;
    void ReheapUp ( int root, int bottom ) ;

    ItemType* elements; //ARRAY to be allocated dynamically
    int numElements ;
};
```



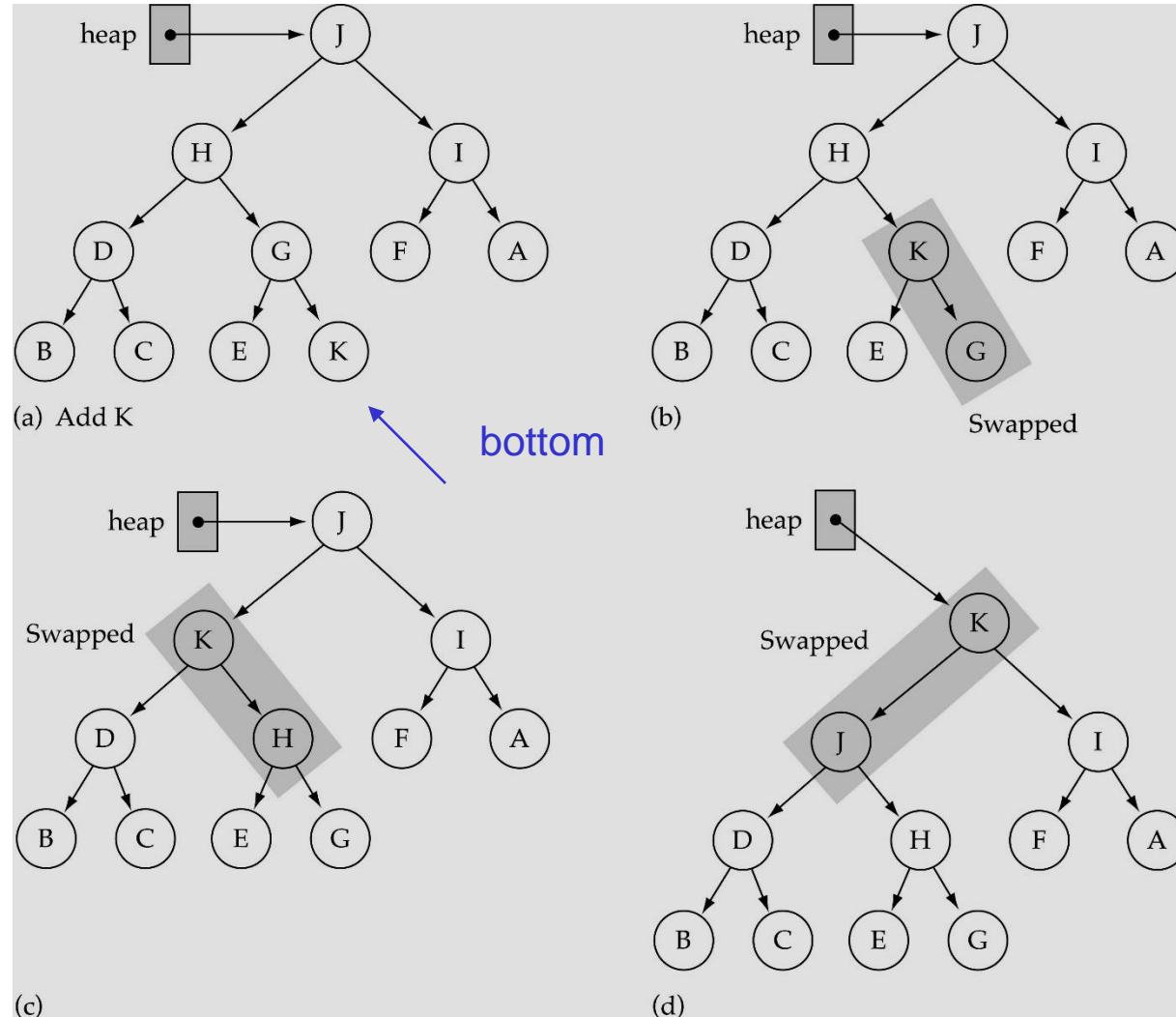
The ReheapDown function (used by deleteItem)



Assumption:
heap property is
violated at the
root of the tree



The ReheapUp function (used by insertItem)



Assumption:
heap property is
violated at the
rightmost node
at the last level
of the tree



ReheapDown

```
// IMPLEMENTATION OF RECURSIVE HEAP MEMBER FUNCTIONS rightmost node  
// in the last level  
  
template< class ItemType >  
void HeapType<ItemType>::ReheapDown ( int root, int bottom )  
  
// Pre: root is the index of the node that may violate the  
// heap order property  
// Post: Heap order property is restored between root and bottom  
  
{  
    int maxChild ;  
    int rightChild ;  
    int leftChild ;  
  
    leftChild = root * 2 + 1 ;  
    rightChild = root * 2 + 2 ;
```





ReheapDown (cont)

```
if  ( leftChild  <=  bottom ) // Is there leftChild?  
{  
    if ( leftChild  == bottom ) // only one child  
        maxChild  =  leftChld ;  
    else // two children  
    {  
        if (elements [ leftChild ] <= elements [ rightChild ] )  
            maxChild  =  rightChild ;  
        else  
            maxChild  =  leftChild ;  
    }  
    if  ( elements [ root ] < elements [ maxChild ] )  
    {  
        Swap ( elements [ root ] , elements [ maxChild ] ) ;  
        ReheapDown ( maxChild, bottom ) ;  
    }  
}  
}
```



ReheapUp

// IMPLEMENTATION

continued

rightmost node
in the last level

```
template< class ItemType >
void HeapType<ItemType>::ReheapUp ( int root, int bottom )  
  
// Pre: bottom is the index of the node that may violate the heap
// order property. The order property is satisfied from root to
// next-to-last node.
// Post: Heap order property is restored between root and bottom  
  
{  
    int parent ;  
  
    if ( bottom > root ) // tree is not empty
    {
        parent = ( bottom - 1 ) / 2;
        if ( elements [ parent ] < elements [ bottom ] )
        {
            Swap ( elements [ parent ], elements [ bottom ] ) ;
            ReheapUp ( root, parent ) ;
        }
    }
}
```



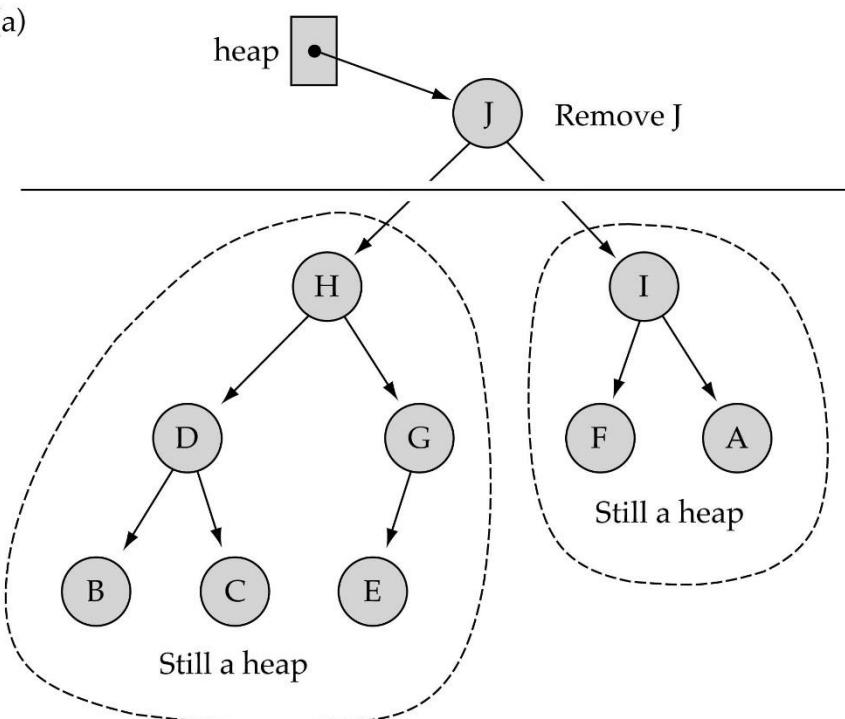
Removing the largest element from the heap

- (1) Copy the bottom rightmost element to the root
- (2) Delete the bottom rightmost node
- (3) Fix the heap property by calling
ReheapDown

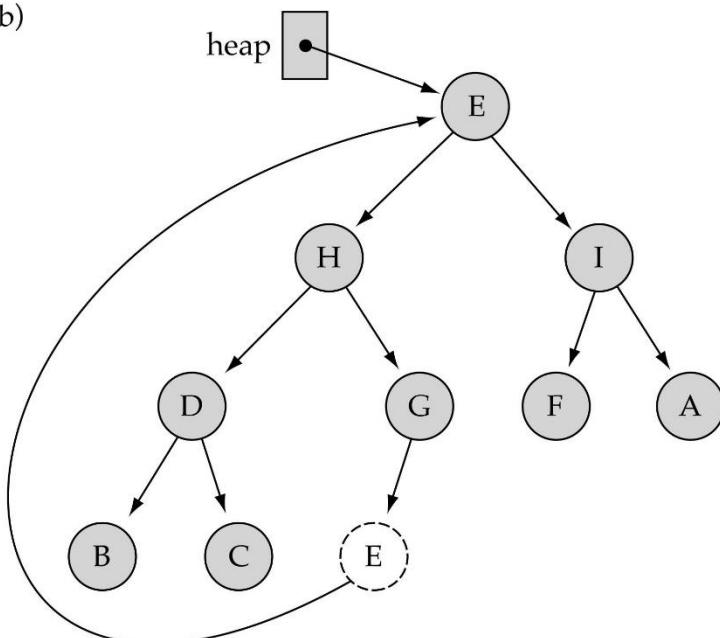


Removing the largest element from the heap (cont.)

(a)



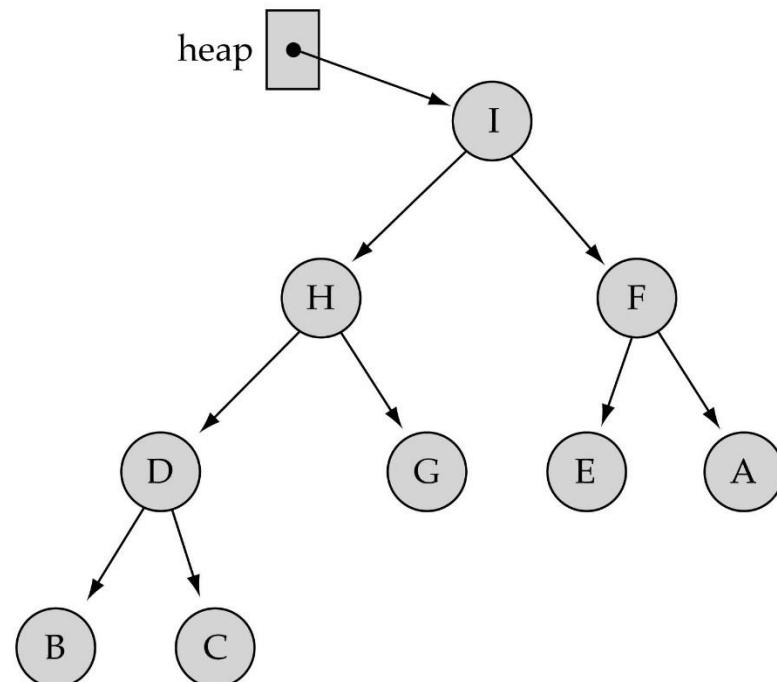
(b)





Removing the largest element from the heap (cont.)

(c)



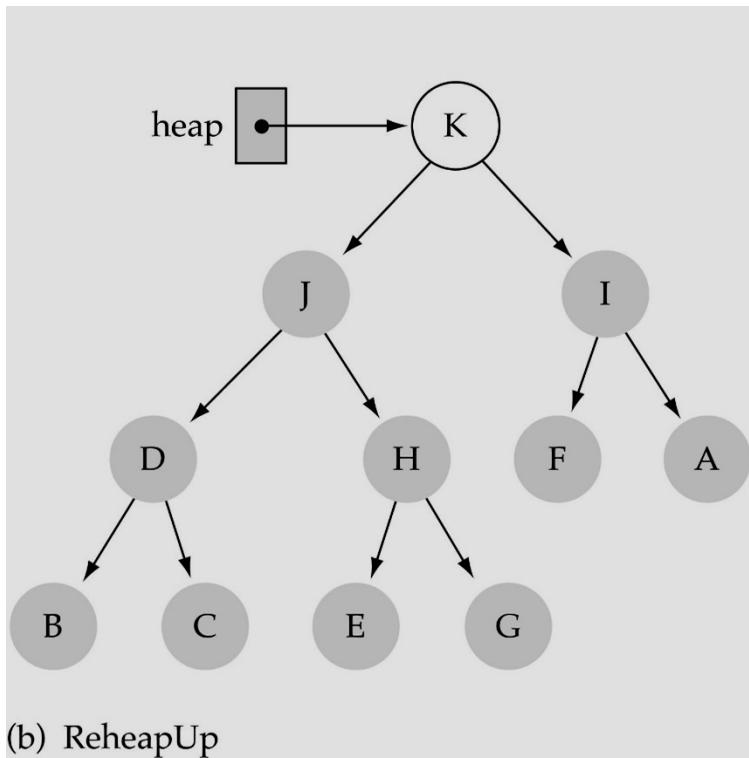
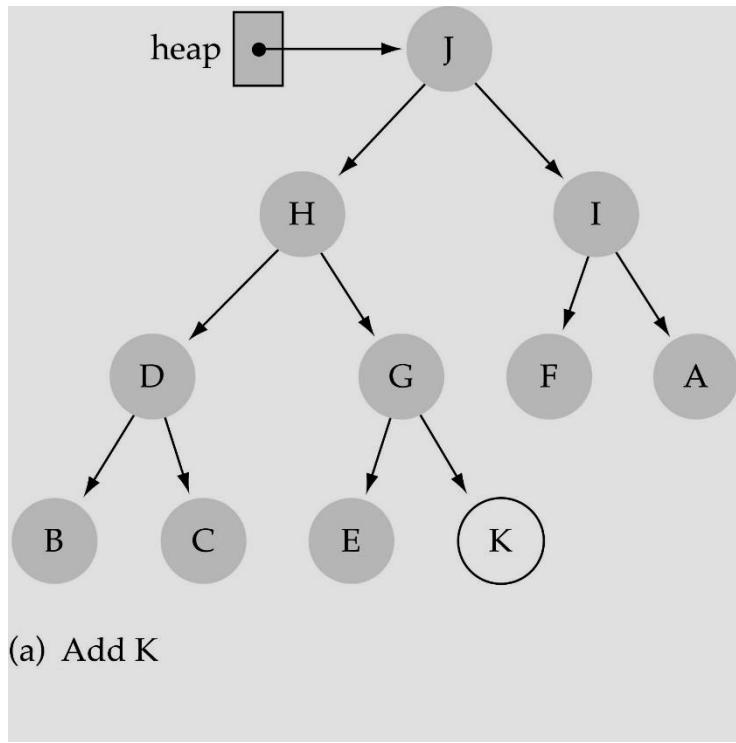


Inserting a new element into the heap

- (1) Insert the new element in the next bottom **leftmost** place
- (2) Fix the heap property by calling *ReheapUp*



Inserting a new element into the heap (cont.)





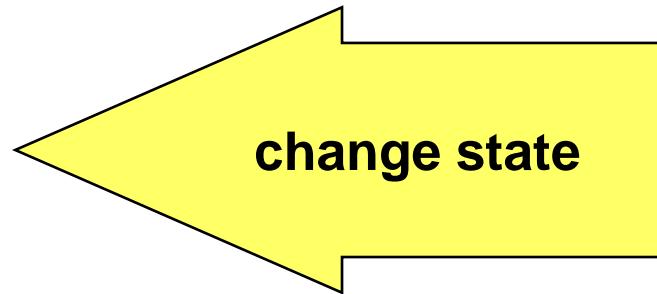
Priority Queue

A priority queue is an ADT with the property that **only the highest-priority element can be accessed at any time.**

ADT Priority Queue Operations

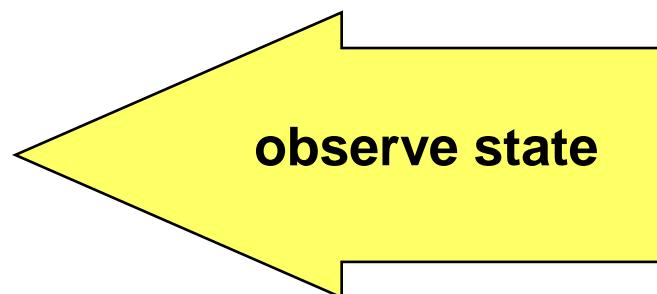
Transformers

- **MakeEmpty**
- **Enqueue**
- **Dequeue**



Observers

- **IsEmpty**
- **IsFull**



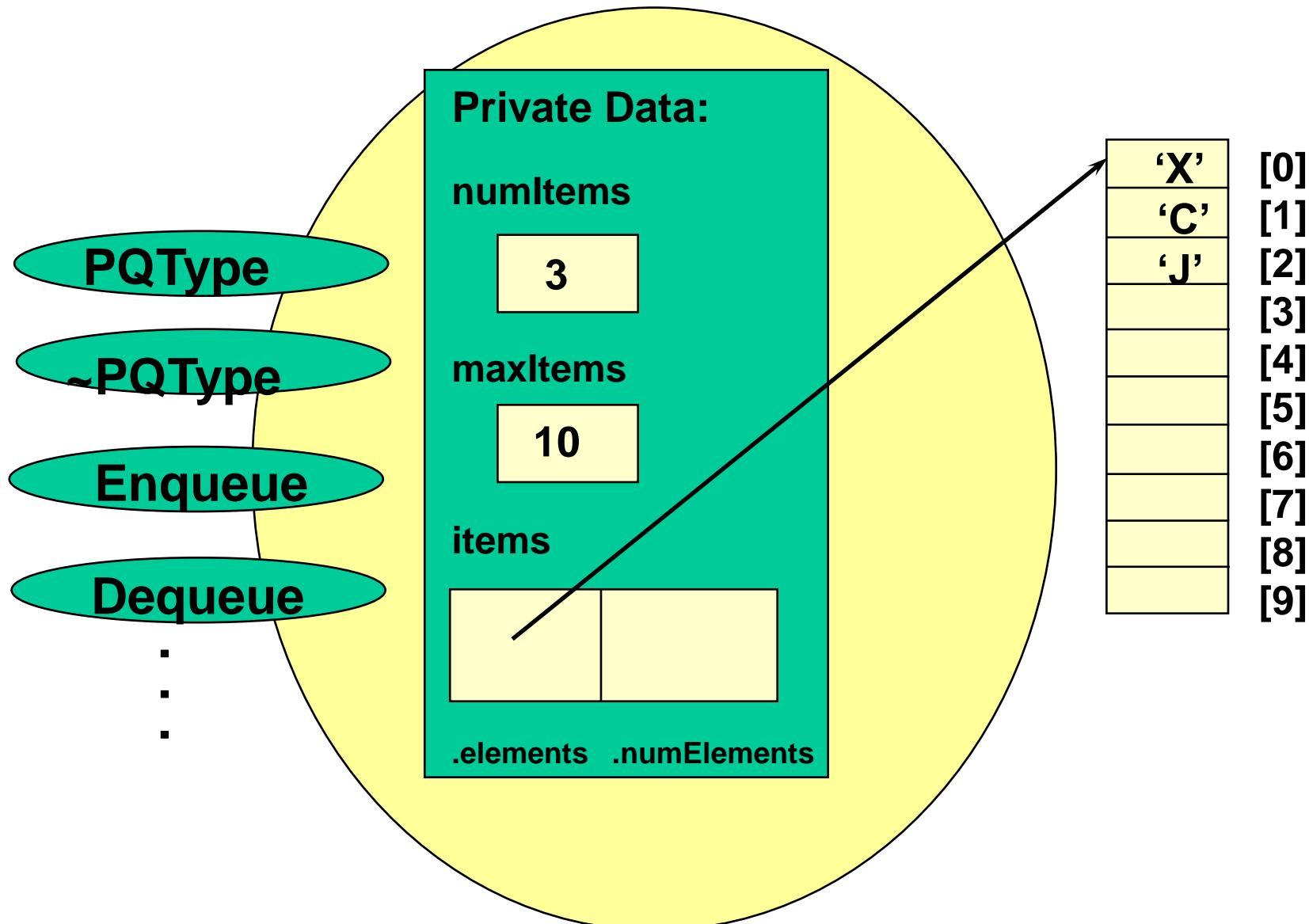


Implementation Level

- There are many ways to implement a priority queue
 - **An unsorted List-** dequeuing would require searching through the entire list
 - **An Array-Based Sorted List-** Enqueuing is expensive
 - **A Linked Sorted List-** Enqueuing again is $O(N)$
 - **A Binary Search Tree-** On average, $O(\log_2 N)$ steps for both enqueue and dequeue
 - **A Heap-** guarantees $O(\log_2 N)$ steps, even in the worst case



class PQType<char>





Class PQType Declaration

```
class FullPQ(){};  
class EmptyPQ(){};  
template<class ItemType>  
class PQType  
{  
public:  
    PQType(int);  
    ~PQType();  
    void MakeEmpty();  
    bool IsEmpty() const;  
    bool IsFull() const;  
    void Enqueue(ItemType newItem);  
    void Dequeue(ItemType& item);  
private:  
    int length;  
    HeapType<ItemType> items;  
    int maxItems;  
};
```



Class PQType Function Definitions

```
template<class ItemType>
PQType<ItemType>::PQType(int max)
{
    maxItems = max;
    items.elements = new ItemType[max];
    length = 0;
}

template<class ItemType>
void PQType<ItemType>::MakeEmpty()
{
    length = 0;
}

template<class ItemType>
PQType<ItemType>::~PQType()
{
    delete [] items.elements;
}
```



Class PQType Function Definitions

Dequeue

Set item to root element from queue

Move last leaf element into root position

Decrement length

items.ReheapDown(0, length-1)

Enqueue

Increment length

Put newItem in next available position

items.ReheapUp(0, length-1)



Code for Dequeue

```
template<class ItemType>
void PQType<ItemType>::Dequeue(ItemType& item)
{
    if (length == 0)
        throw EmptyPQ();
    else
    {
        item = items.elements[0];
        items.elements[0] = items.elements[length-1];
        length--;
        items.ReheapDown(0, length-1);
    }
}
```



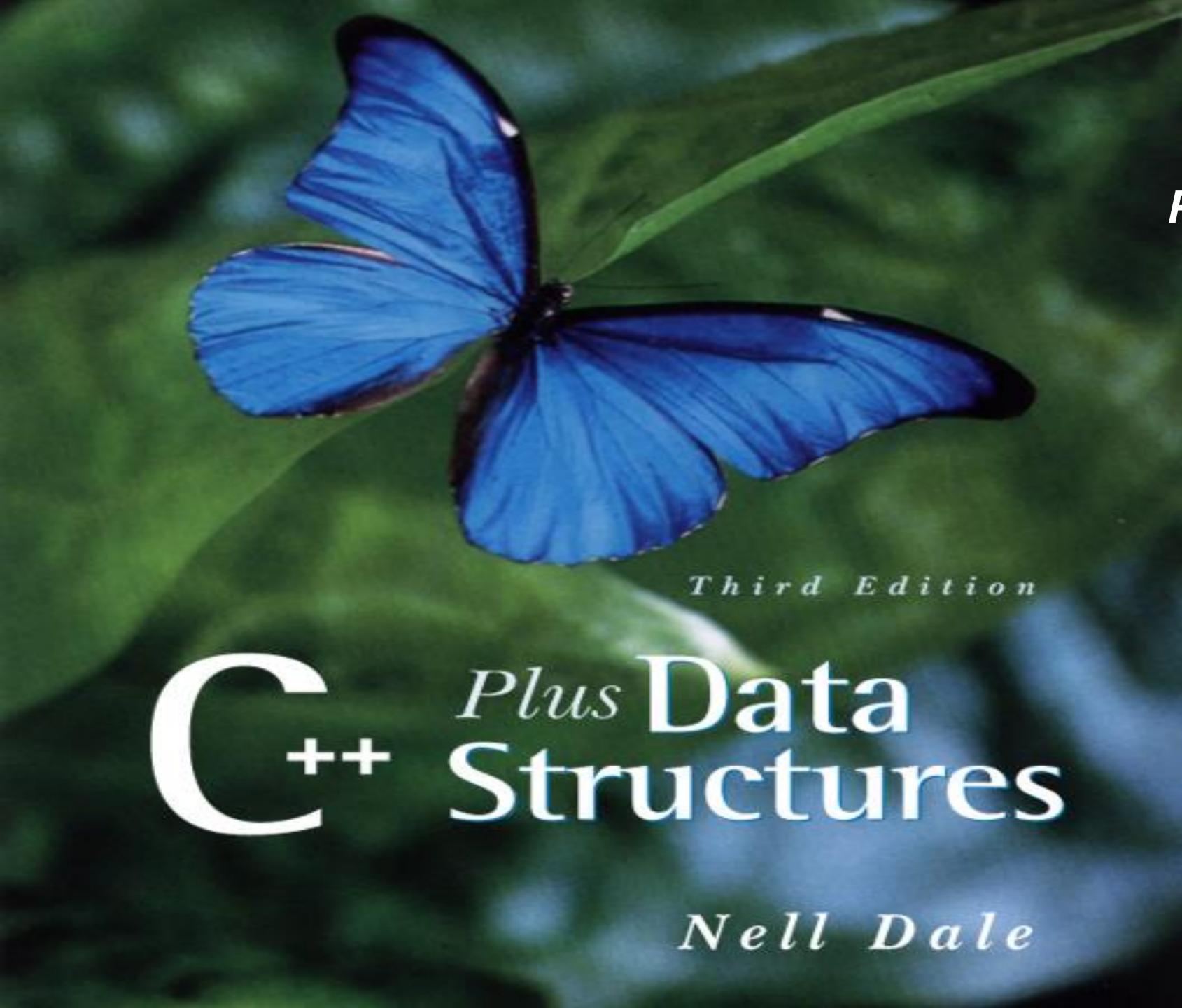
Code for Enqueue

```
template<class ItemType>
void PQType<ItemType>::Enqueue(ItemType newItem)
{
    if (length == maxItems)
        throw FullPQ();
    else
    {
        length++;
        items.elements[length-1] = newItem;
        items.ReheapUp(0, length-1);
    }
}
```



Comparison of Priority Queue Implementations

	<i>Enqueue</i>	<i>Dequeue</i>
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(N)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$



Chapter

9

Priority Queues, Heaps, and Graphs

Third Edition

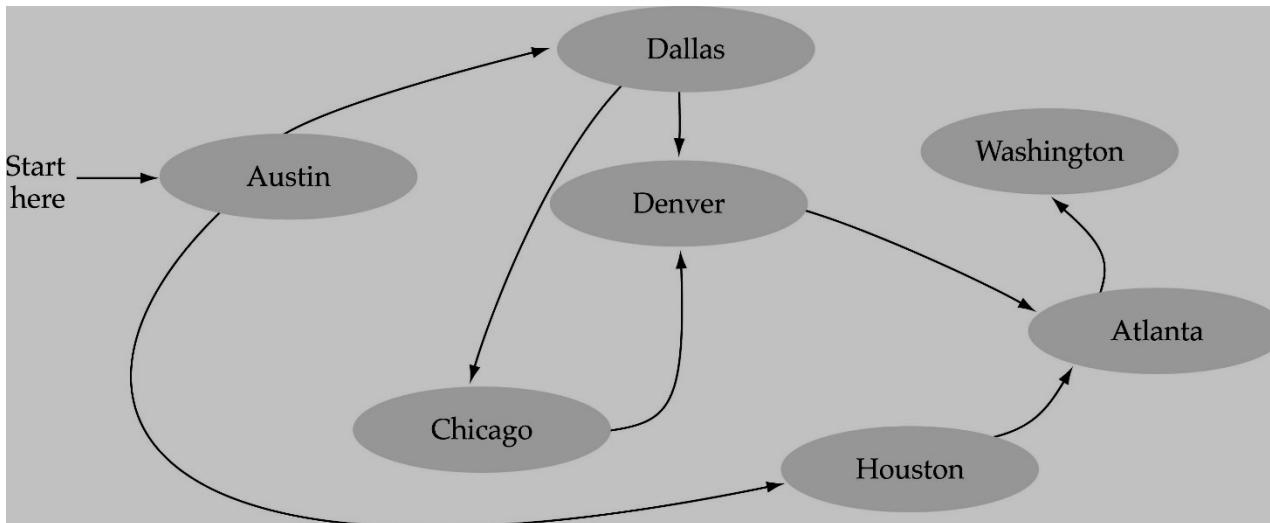
C++ Plus Data Structures

Nell Dale



What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices





Formal definition of graphs

- A graph G is defined as follows:

$$G=(V,E)$$

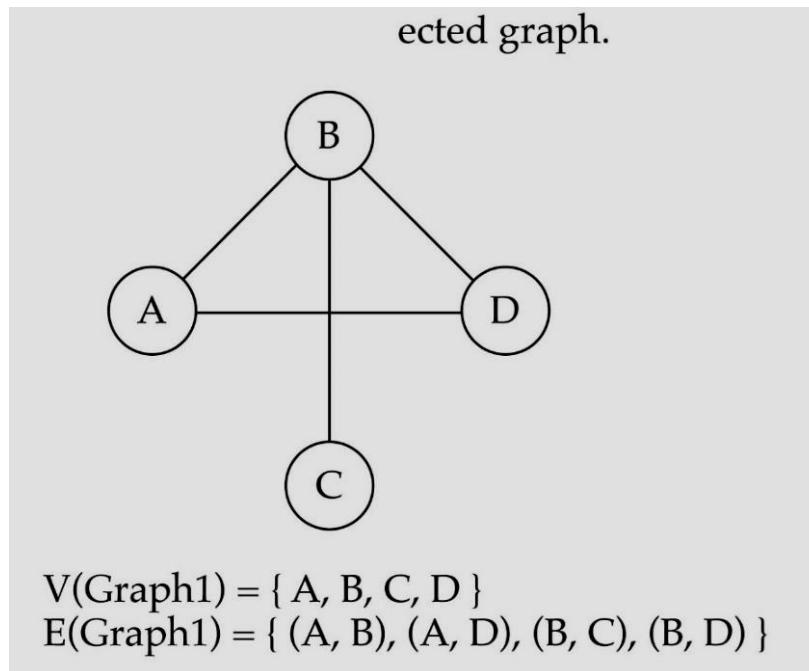
$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)



Directed vs. undirected graphs

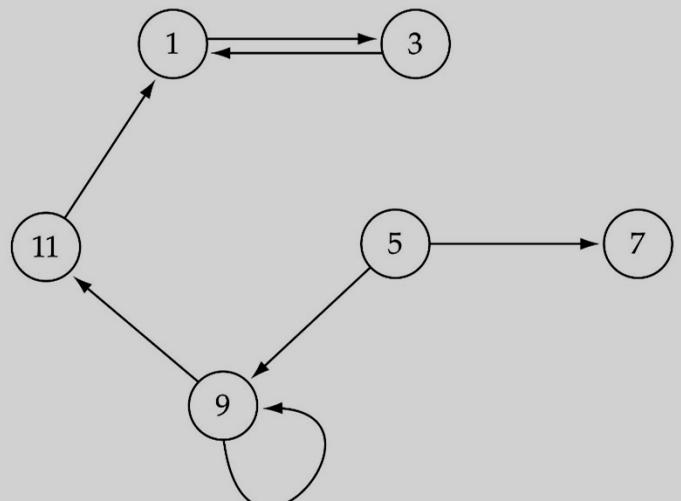
- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



$$V(\text{Graph2}) = \{ 1, 3, 5, 7, 9, 11 \}$$

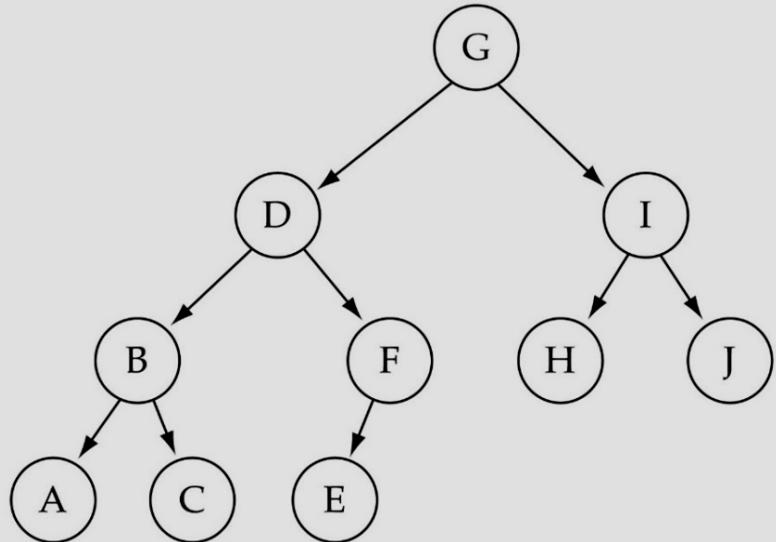
$$E(\text{Graph2}) = \{(1,3), (3,1), (5,9), (9,11), (5,7), (9,9), (11,1)\}$$

Warning: if the graph is directed, the order of the vertices in each edge is important !!

Trees vs. graphs

- Trees are special cases of graphs!!

(c) Graph3 is a directed graph.



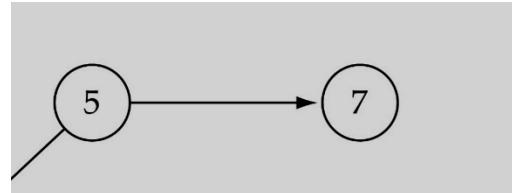
$$V(\text{Graph3}) = \{ A, B, C, D, E, F, G, H, I, J \}$$

$$E(\text{Graph3}) = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$$



Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent **to** 7
7 is adjacent **from** 5

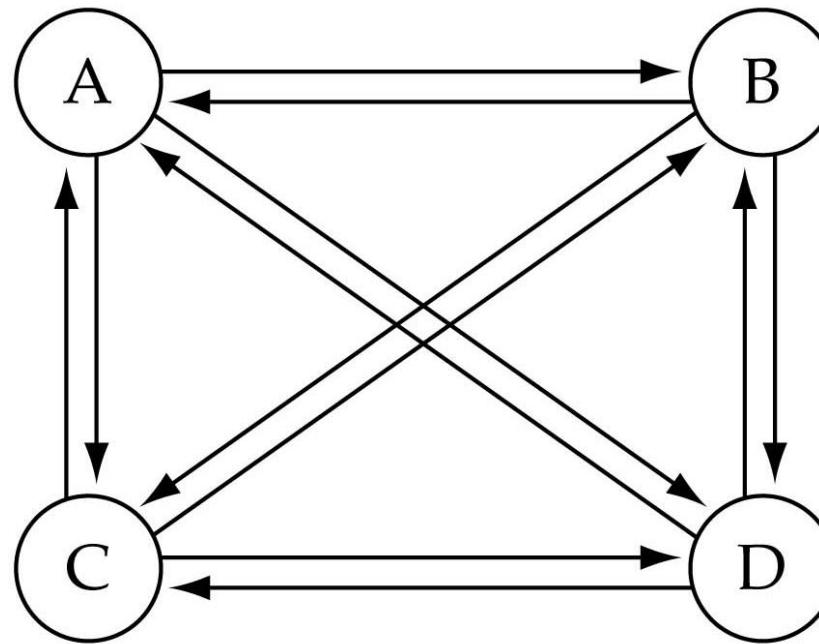
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

$$N * (N-1)$$

$O(N^2)$



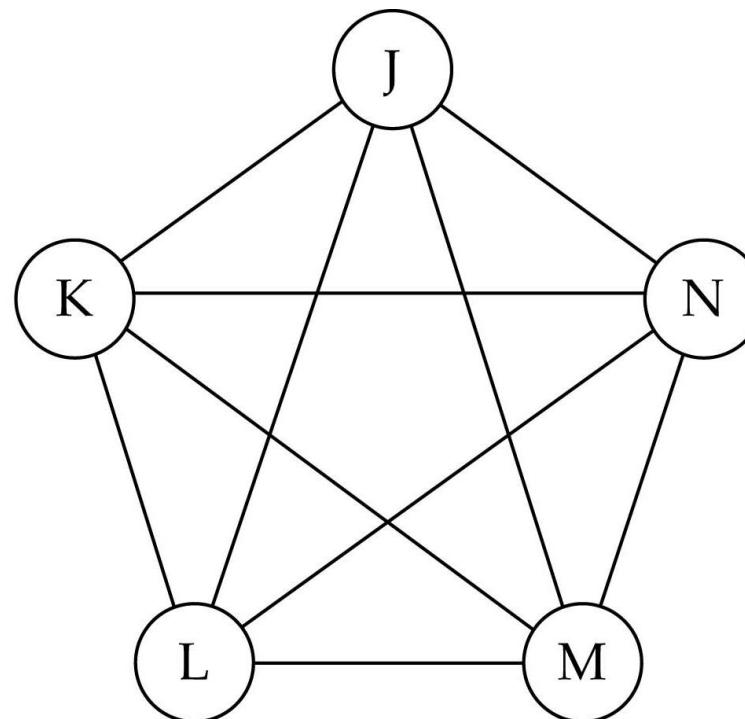
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

$$N * (N-1) / 2$$

$$O(N^2)$$

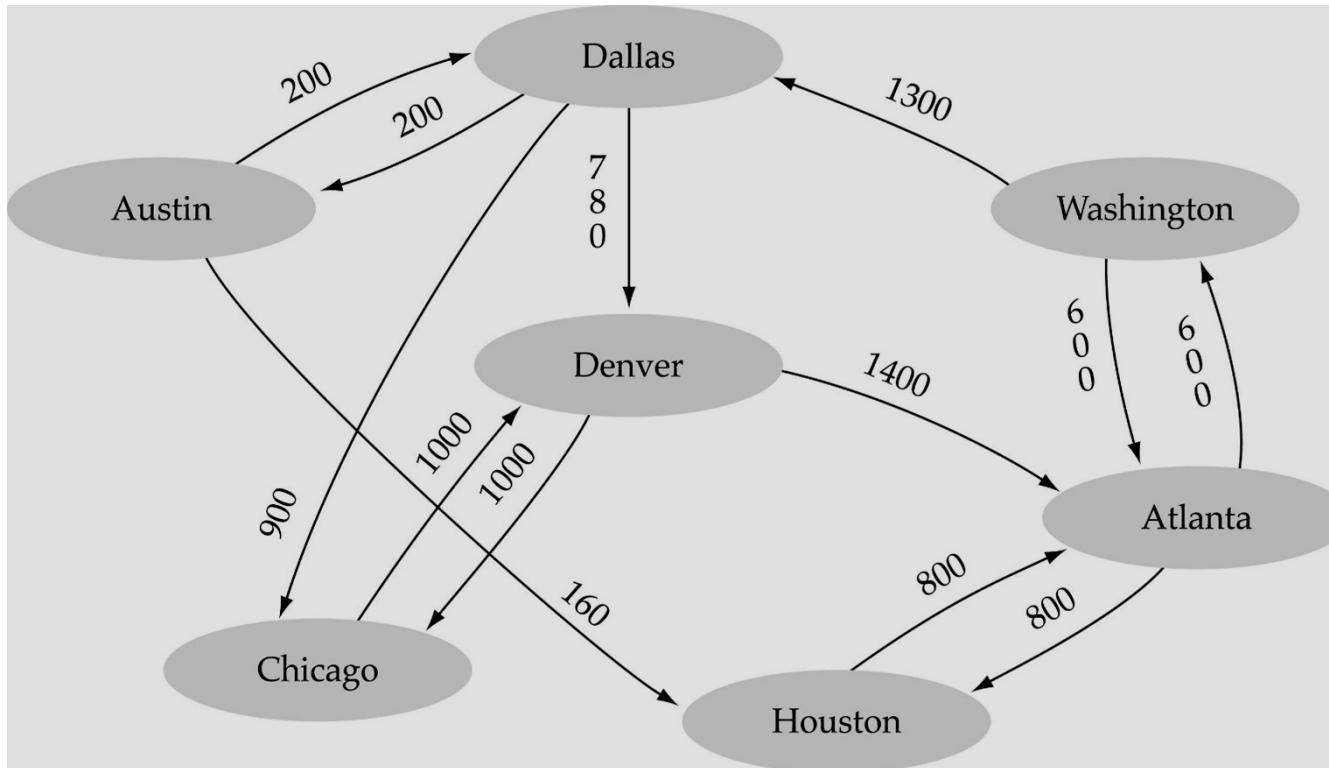


(b) Complete undirected graph.



Graph terminology (cont.)

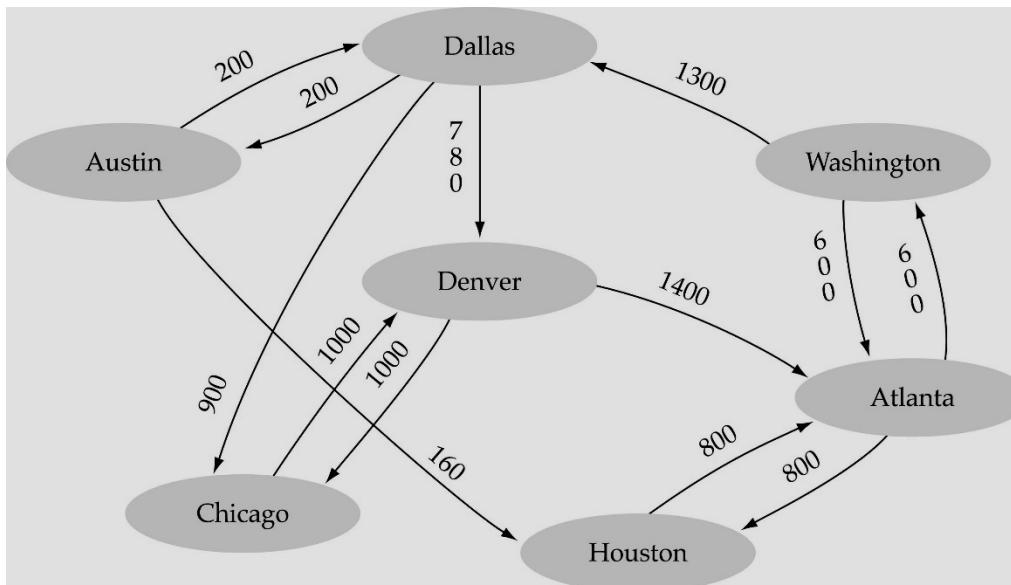
- Weighted graph: a graph in which each edge carries a value





Graph implementation

- Adjacency Matrix: Array-based implementation
 - A 1D array is used to represent the vertices
 - A 2D array (adjacency matrix) is used to represent the edges



Array-based implementation

graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

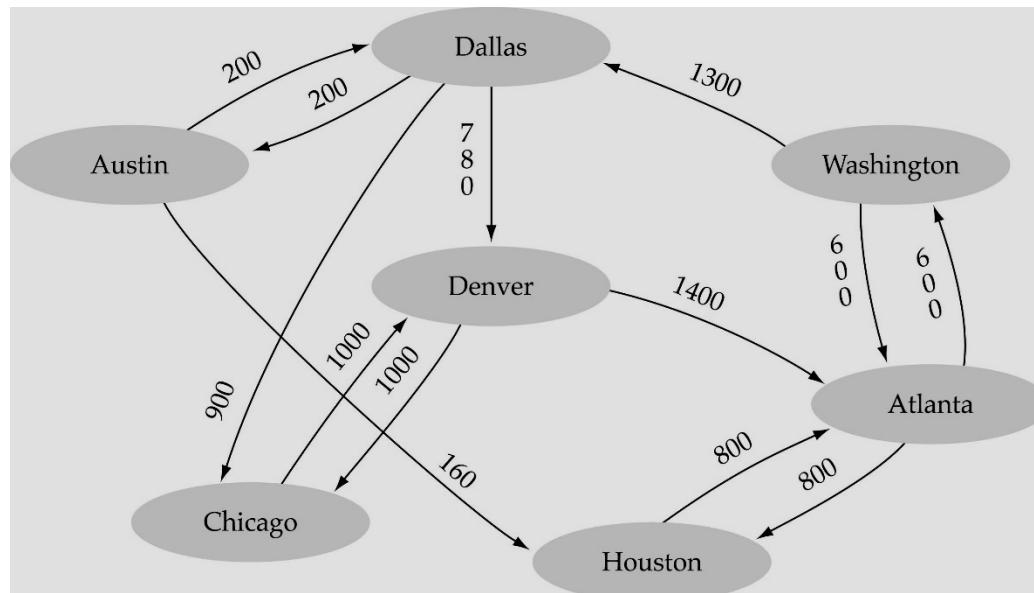
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

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

(Array positions marked '•' are undefined)

Graph implementation (cont.)

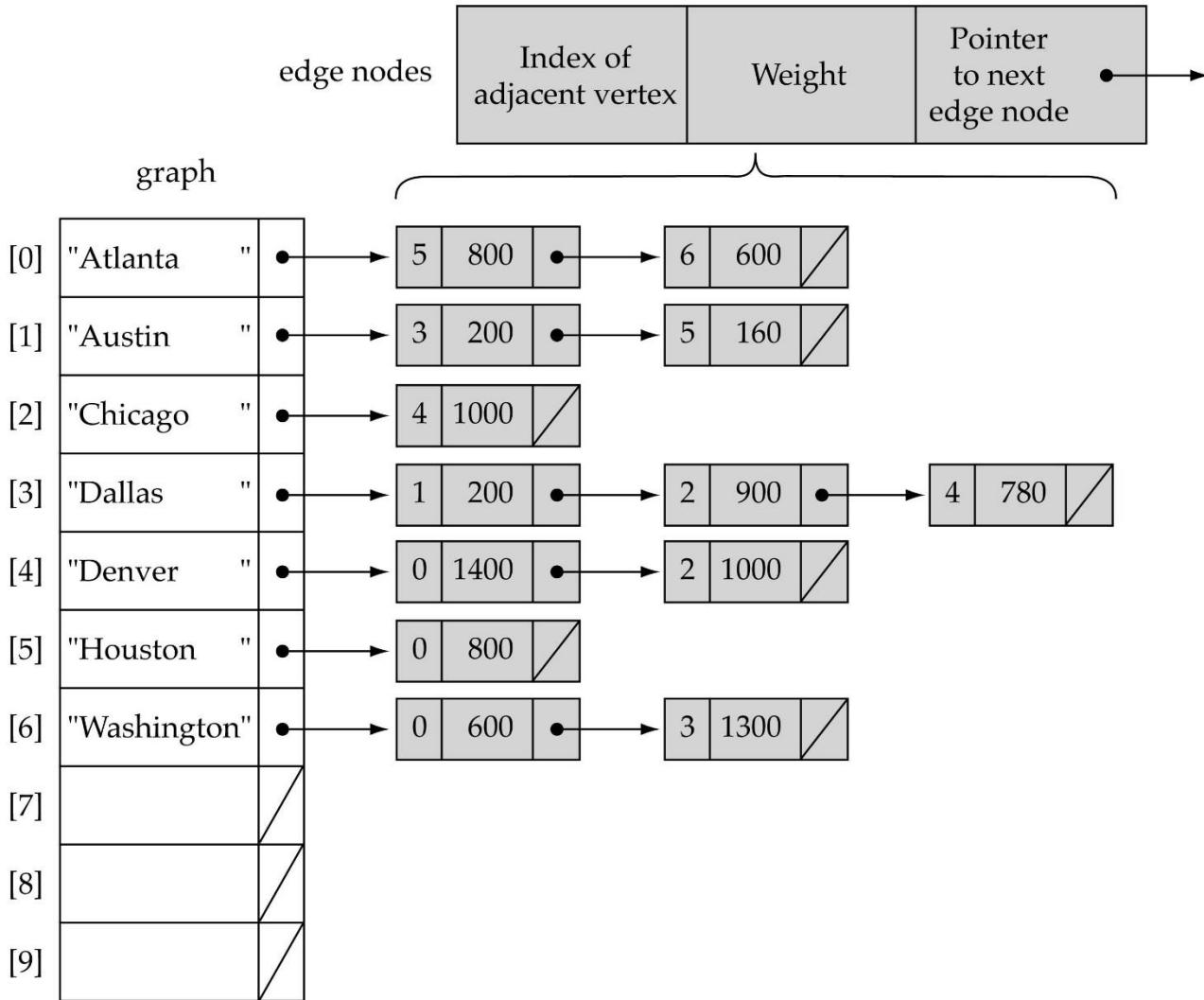
- Adjacency List: Linked-list implementation
 - A 1D array is used to represent the vertices
 - A list is used for each vertex v which contains the vertices which are adjacent from v





Linked-list implementation

(a)





Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
 - Good for dense graphs -- $|E| \sim O(|V|^2)$
 - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
 - Connectivity between two vertices can be tested quickly
- **Adjacency list**
 - Good for sparse graphs -- $|E| \sim O(|V|)$
 - Memory requirements: $O(|V| + |E|) = O(|V|)$
 - Vertices adjacent to another vertex can be found quickly



Graph specification based on adjacency matrix representation

```
const int NULL_EDGE = 0;

template<class VertexType>
class GraphType {
public:
    GraphType(int);
    ~GraphType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void AddVertex(VertexType);
    void AddEdge(VertexType, VertexType, int);
    int WeightIs(VertexType, VertexType);
    void GetToVertices(VertexType,
QueType<VertexType>&);
    void ClearMarks();
    void MarkVertex(VertexType);
    bool IsMarked(VertexType) const;
private:
    int numVertices;
    int maxVertices;
    VertexType* vertices;
    int **edges;
    bool* marks;
};
```

(continues)



```
template<class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
    numVertices = 0;
    maxVertices = maxV;
    vertices = new VertexType[maxV];
    edges = new int[maxV];
    for(int i = 0; i < maxV; i++)
        edges[i] = new int[maxV];
    marks = new bool[maxV];
}

template<class VertexType>
GraphType<VertexType>::~GraphType()
{
    delete [] vertices;
    for(int i = 0; i < maxVertices; i++)
        delete [] edges[i];
    delete [] edges;
    delete [] marks;
}
```

(continues)



```
void GraphType<VertexType>::AddVertex(VertexType vertex)
{
    vertices[numVertices] = vertex;

    for(int index = 0; index < numVertices; index++) {
        edges[numVertices][index] = NULL_EDGE;
        edges[index][numVertices] = NULL_EDGE;
    }

    numVertices++;
}

template<class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex,
                                      VertexType toVertex, int weight)
{
    int row;
    int column;

    row = IndexIs(vertices, fromVertex);
    col = IndexIs(vertices, toVertex);
    edges[row][col] = weight;
}
```

(continues)



```
template<class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex,
                                         VertexType toVertex)
{
    int row;
    int column;

    row = IndexIs(vertices, fromVertex);
    col = IndexIs(vertices, toVertex);
    return edges[row][col];
}
```



Graph searching

- Problem: find a path between two nodes of the graph (e.g., Austin and Washington)
- Methods: Depth-First-Search (DFS) or Breadth-First-Search (BFS)



Depth-First-Search (DFS)

- What is the idea behind DFS?
 - Visit all nodes in a branch to its deepest point before moving up
 - Travel as far as you can down a path
- DFS can be implemented efficiently using a *stack*

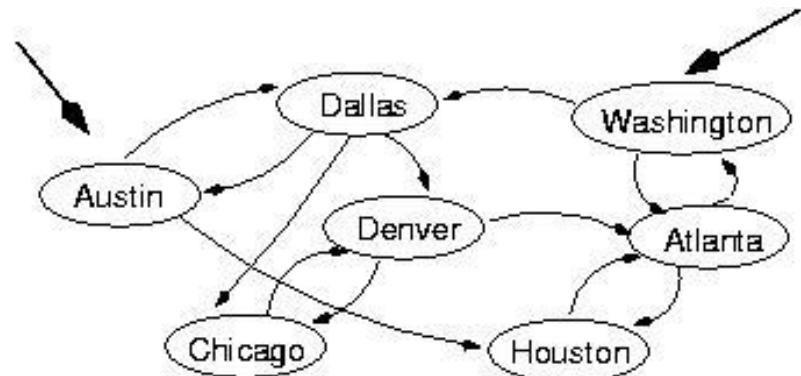


Depth-First-Search (DFS) (cont.)

```
Set found to false  
stack.Push(startVertex)  
DO  
    stack.Pop(vertex)  
    IF vertex == endVertex  
        Set found to true  
    ELSE  
        Push all adjacent vertices onto stack  
    WHILE !stack.IsEmpty() AND !found  
  
    IF(!found)  
        Write "Path does not exist"
```

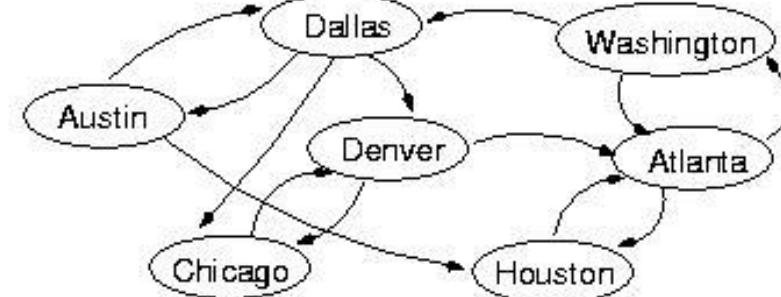


start

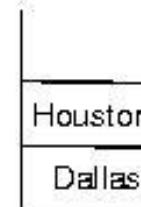


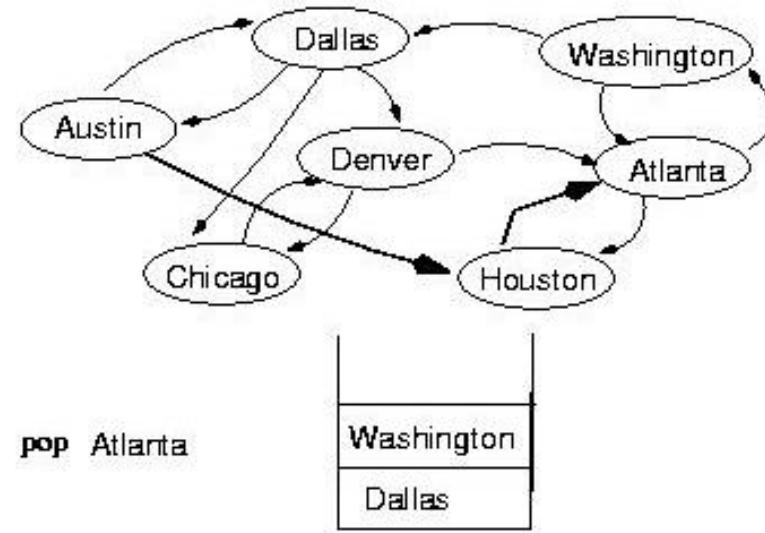
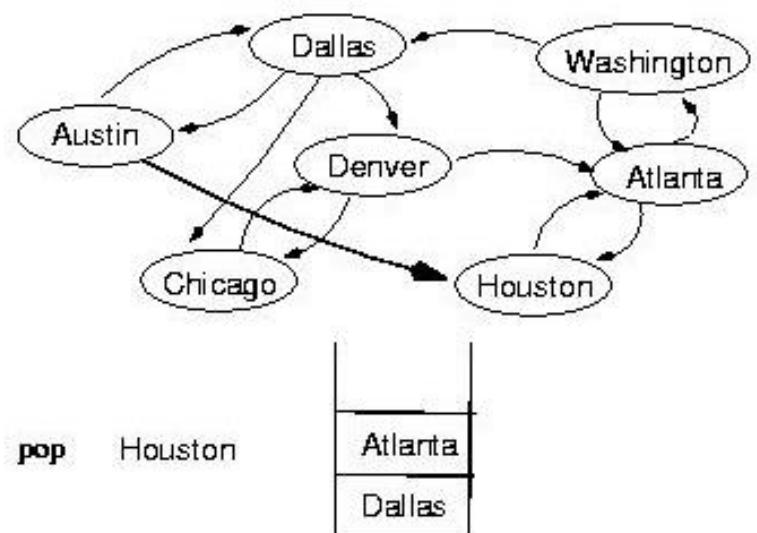
(initialization)

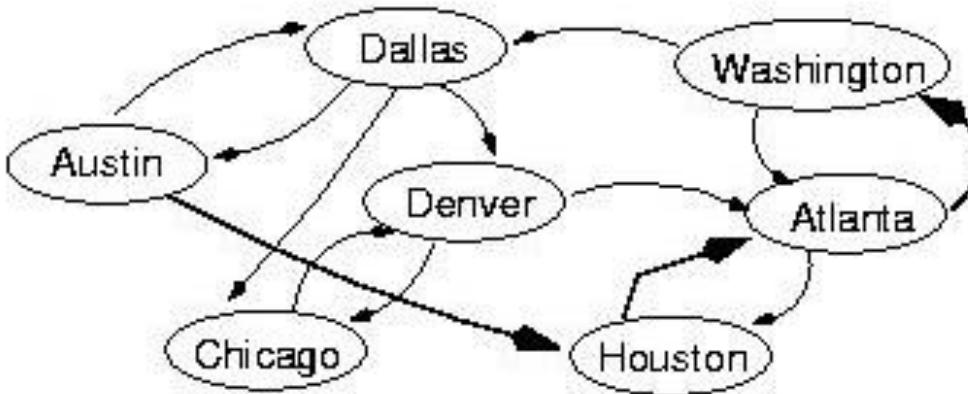
end



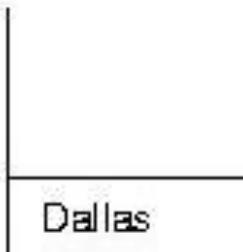
pop Austin







pop Washington





```
template <class ItemType>
void DepthFirstSearch(GraphType<VertexType> graph,
                      VertexType startVertex, VertexType endVertex)
{
    StackType<VertexType> stack;
    QueType<VertexType> vertexQ;

    bool found = false;
    VertexType vertex;
    VertexType item;

    graph.ClearMarks();
    stack.Push(startVertex);
    do {
        stack.Pop(vertex);
        if(vertex == endVertex)
            found = true;
```

(continues)



```
else {
    if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
            vertexQ.Dequeue(item);
            if(!graph.IsMarked(item))
                stack.Push(item);
        }
    }
} while(!stack.IsEmpty() && !found);

if(!found)
    cout << "Path not found" << endl;
}
```

(continues)

```
template<class VertexType>
void
    GraphType<VertexType>::GetToVertices (VertexType
vertex,
                                         QueTye<VertexType>& adjvertexQ)
{
    int fromIndex;
    int toIndex;

    fromIndex = IndexIs(vertices, vertex);
    for(toIndex = 0; toIndex < numVertices;
        toIndex++)
        if(edges[fromIndex] [toIndex] != NULL_EDGE)
            adjvertexQ.Enqueue(vertices [toIndex]);
}
```



Breadth-First-Searching (BFS)

- What is the idea behind BFS?
 - Visit all the nodes on one level before going to the next level
 - Look at all possible paths at the same depth before you go at a deeper level



Breadth-First-Searching (BFS) (cont.)

- BFS can be implemented efficiently using a *queue*

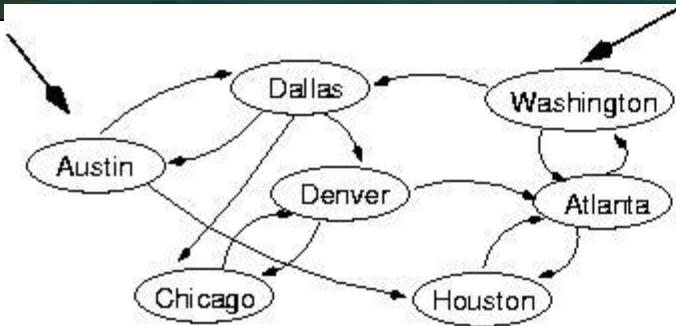
```
Set found to false
queue.Enqueue(startVertex)
DO
    queue.Dequeue(vertex)
    IF vertex == endVertex
        Set found to true
    ELSE
        Enqueue all adjacent vertices onto queue
    WHILE !queue.IsEmpty() AND !found
    IF(!found)
        Write "Path does not exist"
```

- Should we mark a vertex when it is enqueued or when it is dequeued ?

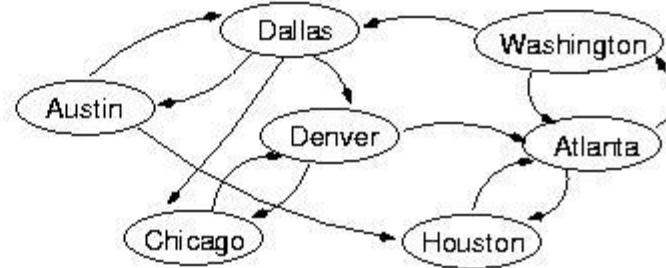
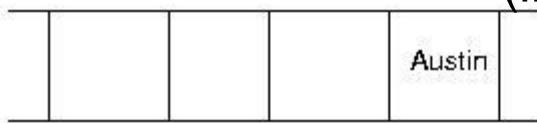


start

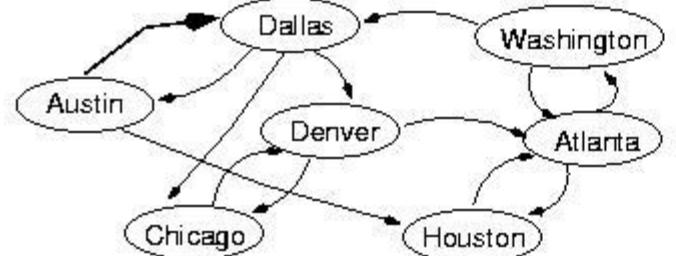
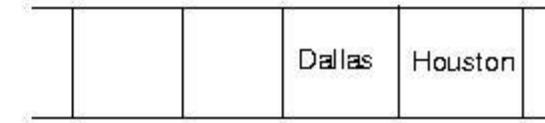
end



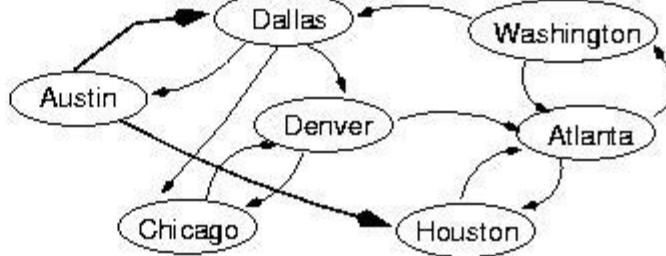
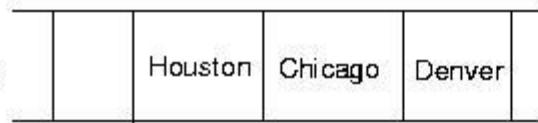
(initialization)



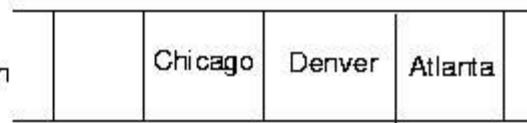
dequeue Austin

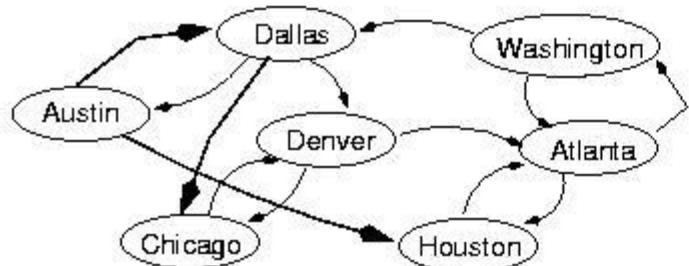


dequeue Dallas

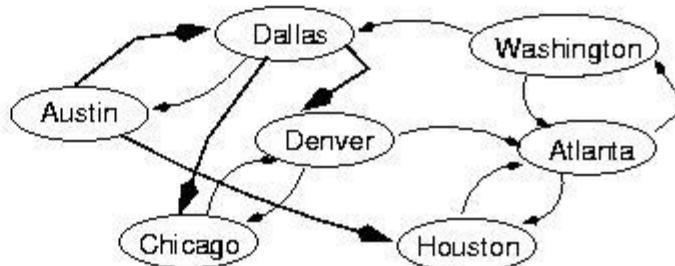
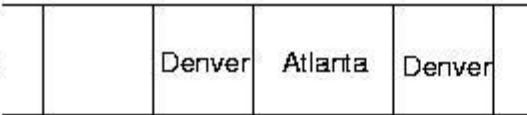


dequeue Houston

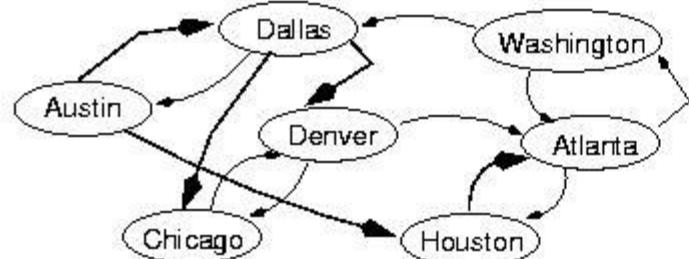




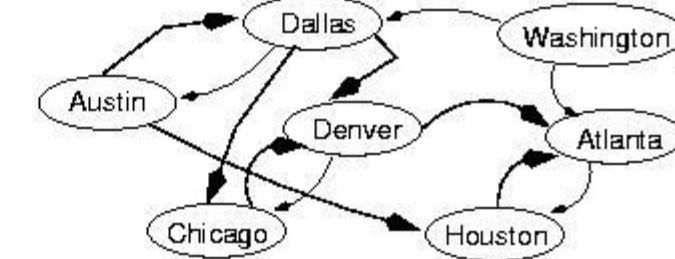
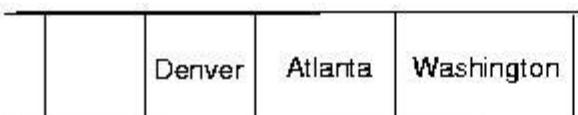
dequeue Chicago



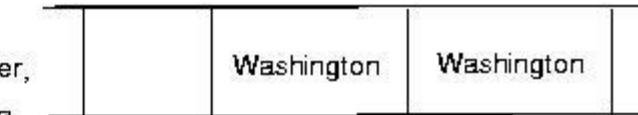
dequeue Denver

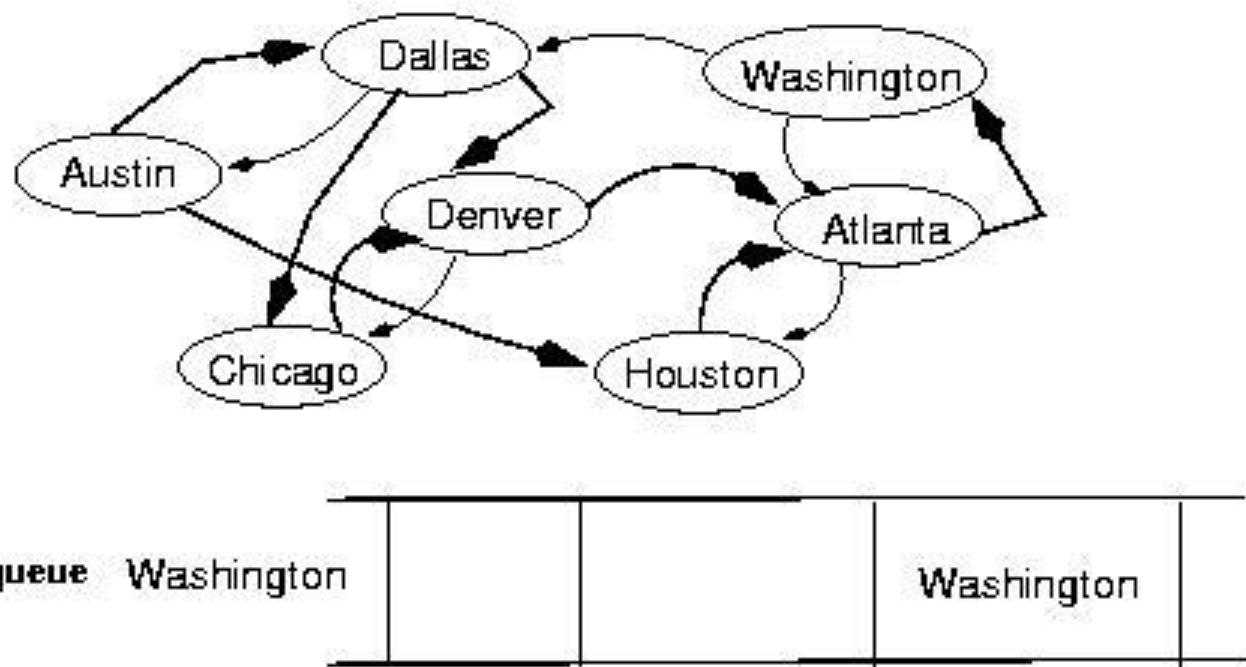


dequeue Atlanta



dequeue Denver,
next: Atlanta







```
template<class VertexType>
void BreadthFirstSearch(GraphType<VertexType>
    graph, VertexType startVertex, VertexType
    endVertex);
{
    QueType<VertexType> queue;
    QueType<VertexType> vertexQ;//

    bool found = false;
    VertexType vertex;
    VertexType item;

    graph.ClearMarks();
    queue.Enqueue(startVertex);
    do {
        queue.Dequeue(vertex);
        if(vertex == endVertex)
            found = true;
    }
}
```

(continues)



```
else {
    if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
            vertexQ.Dequeue(item);
            if(!graph.IsMarked(item))
                queue.Enqueue(item);
        }
    }
} while (!queue.IsEmpty() && !found);

if(!found)
    cout << "Path not found" << endl;
}
```



Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- Shortest path: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
 - Austin->Houston->Atlanta->Washington: 1560 miles
 - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles



Single-source shortest-path problem (cont.)

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same
(mark vertices before Enqueue)



ADT Set Definitions

Base type: The type of the items in the set

Cardinality: The number of items in a set

Cardinality of the base type: The number of items in the base type

Union of two sets: A set made up of all the items in either sets

Intersection of two sets: A set made up of all the items in both sets

Difference of two sets: A set made up of all the items in the first set that are not in the second set



Beware: At the Logical Level

- Sets can not contain duplicates. Storing an item that is already in the set does not change the set.
- If an item is not in a set, deleting that item from the set does not change the set.
- Sets are not ordered.



Implementing Sets

Explicit implementation (Bit vector)

Each item in the base type has a representation in each instance of a set. The representation is either true (item is in the set) or false (item is not in the set).

Space is proportional to the cardinality of the base type.

Algorithms use Boolean operations.



Implementing Sets (cont.)

Implicit implementation (List)

The items in an instance of a set are on a list that represents the set. Those items that are not on the list are not in the set.

Space is proportional to the cardinality of the set instance.

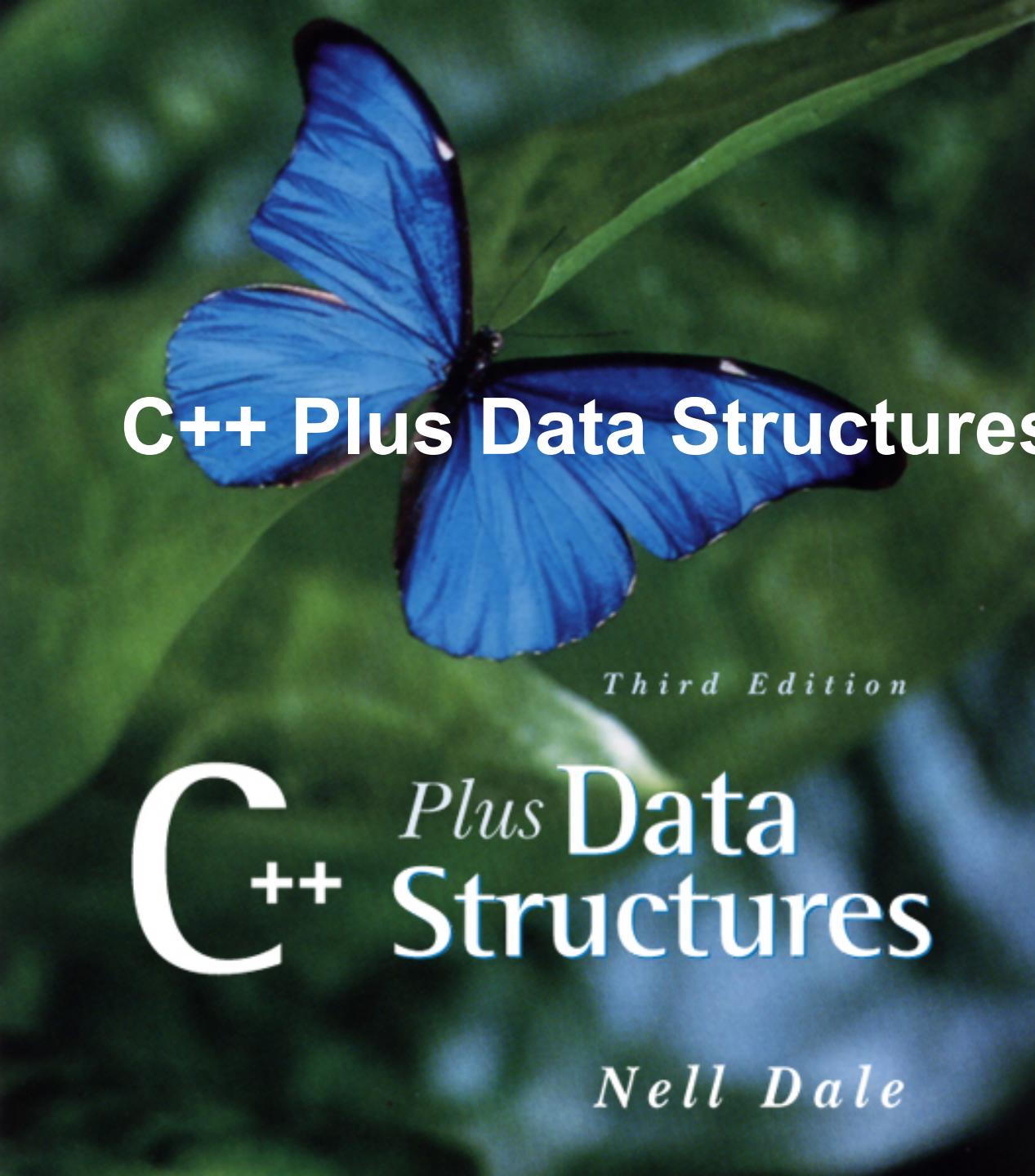
Algorithms use ADT List operations.



Explain:

Although sets are not ordered, why is the SortedList ADT a better choice as the implementation structure for the implicit representation?

Chapter
10
*Sorting and
Searching
Algorithms*



C++ Plus Data Structures

Third Edition

C++ *Plus* Data
Structures

Nell Dale



Sorting means . . .

- The values stored in an array have keys of a type for which the relational operators are defined. (We also assume unique keys.)
- Sorting rearranges the elements into either ascending or descending order within the array. (We'll use ascending order.)



Straight Selection Sort

values	[0]	[1]	[2]	[3]	[4]
	36	24	10	6	12

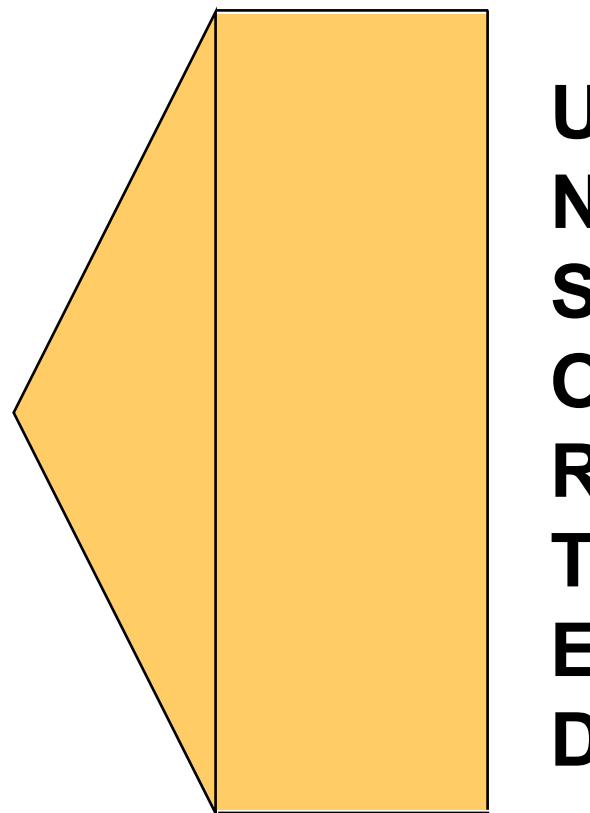
Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.



Selection Sort: Pass One

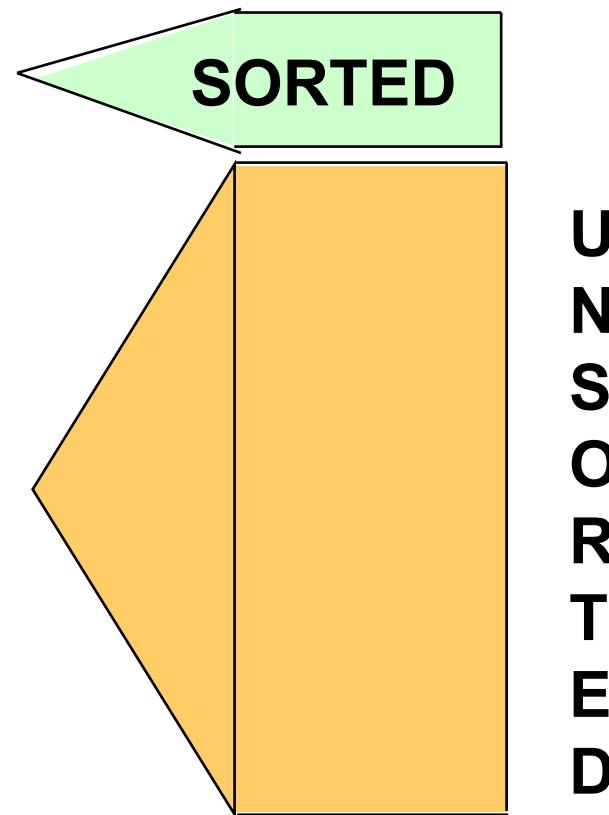
values [0] 36
[1] 24
[2] 10
[3] 6
[4] 12





Selection Sort: End Pass One

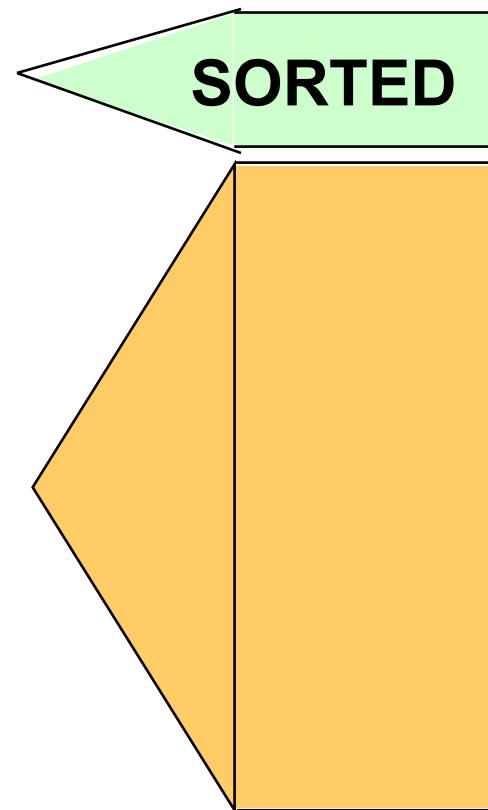
values [0] 6
[1] 24
[2] 10
[3] 36
[4] 12



Selection Sort: Pass Two

values [0] [1] [2] [3] [4]

6
24
10
36
12

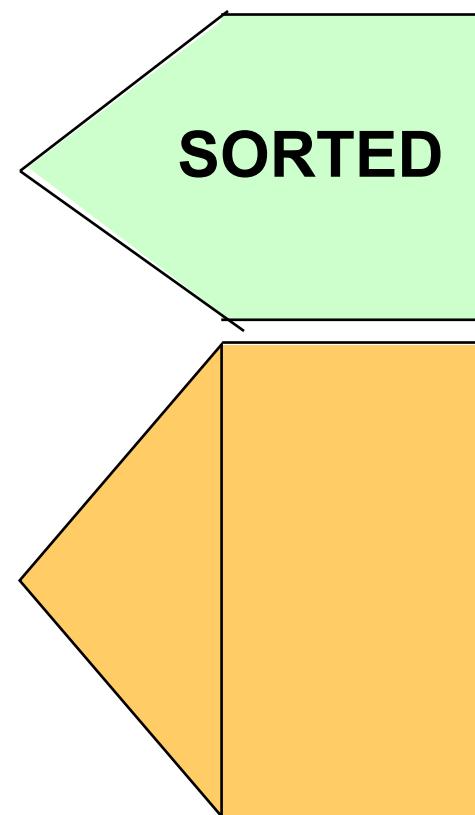




Selection Sort: End Pass Two

values [0] [1] [2] [3] [4]

6
10
24
36
12



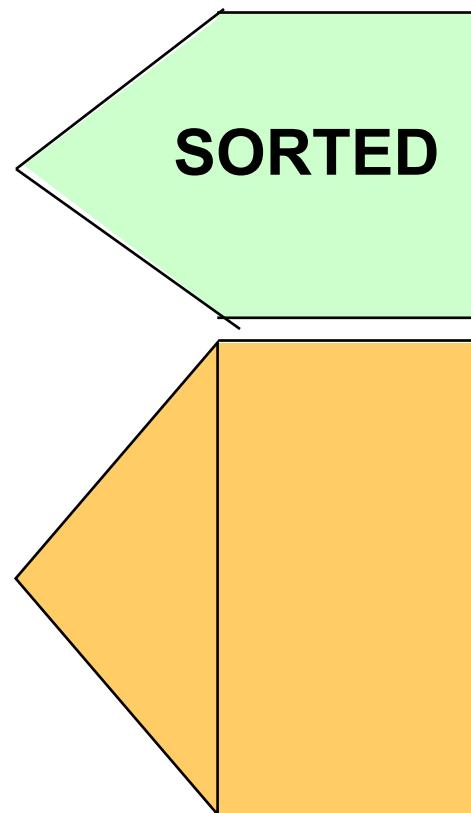
U
N
S
O
R
T
E
D



Selection Sort: Pass Three

values [0] [1] [2] [3] [4]

6
10
24
36
12



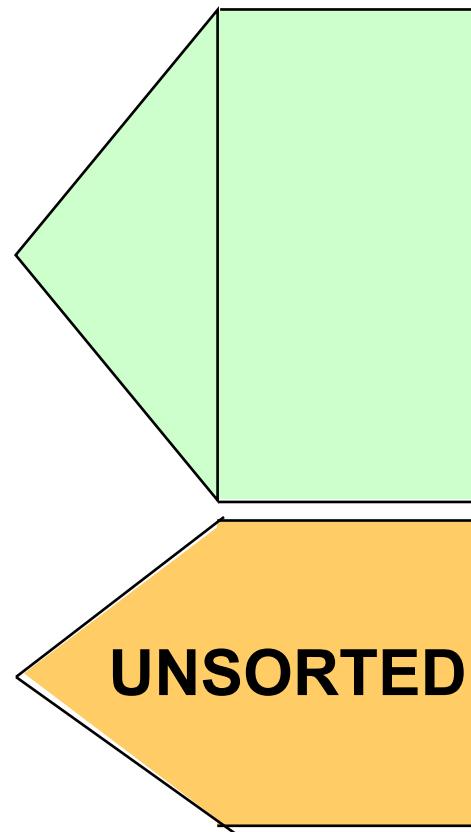
U
N
S
O
R
T
E
D



Selection Sort: End Pass Three

values [0] [1] [2] [3] [4]

6
10
12
36
24

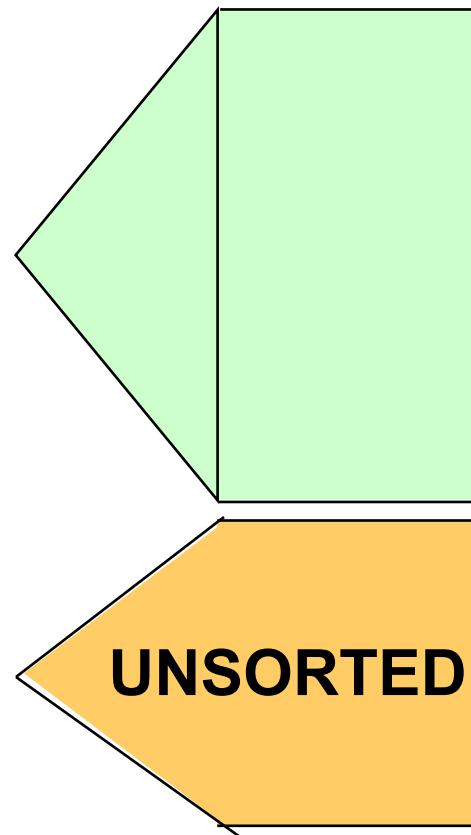




Selection Sort: Pass Four

values [0] [1] [2] [3] [4]

6
10
12
36
24



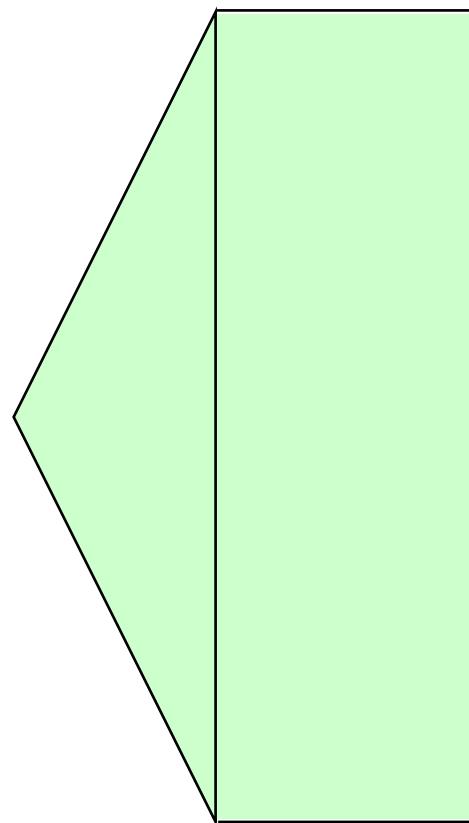
S O R T E D



Selection Sort: End Pass Four

values [0] [1] [2] [3] [4]

6
10
12
24
36



S
O
R
T
E
D



Selection Sort: How many comparisons?

values	[0]	6
	[1]	10
	[2]	12
	[3]	24
	[4]	36

4 compares for values[0]

3 compares for values[1]

2 compares for values[2]

1 compare for values[3]

$$= 4 + 3 + 2 + 1$$



For selection sort in general

- The number of comparisons when the array contains N elements is

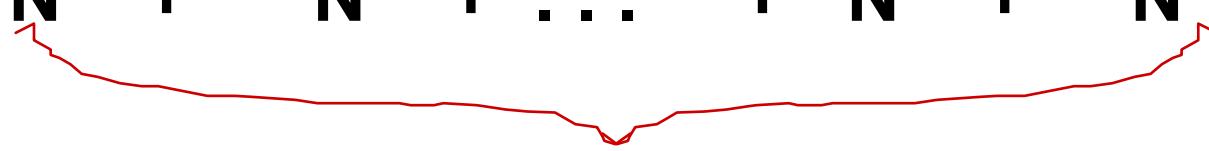
$$\text{Sum} = (N-1) + (N-2) + \dots + 2 + 1$$



Notice that . . .

$$\text{Sum} = (N-1) + (N-2) + \dots + 2 + 1$$

$$+ \text{ Sum} = 1 + 2 + \dots + (N-2) + (N-1)$$

$$2 * \text{Sum} = N + N + \dots + N + N$$


$$2 * \text{Sum} = N * (N-1)$$

$$\text{Sum} = \frac{N * (N-1)}{2}$$



For selection sort in general

- The number of comparisons when the array contains N elements is

$$\text{Sum} = (N-1) + (N-2) + \dots + 2 + 1$$

$$\text{Sum} = N * (N-1) / 2$$

$$\text{Sum} = .5 N^2 - .5 N$$

$$\text{Sum} = O(N^2)$$



```
template <class ItemType>
int MinIndex(ItemType values [ ], int start, int end)
// Post: Function value = index of the smallest value
// in values [start] . . . values [end].
{
    int indexOfMin = start;

    for(int index = start + 1 ; index <= end ; index++)
        if (values[ index ] < values [indexOfMin])
            indexOfMin = index ;

    return    indexOfMin;
}
```



```
template <class ItemType >
void SelectionSort (ItemType values[ ] ,
    int numValues )

// Post: Sorts array values[0 . . numValues-1 ]
// into ascending order by key
{
    int endIndex = numValues - 1 ;
    for (int current = 0 ; current < endIndex;
        current++)
        Swap (values[current] ,
            values[MinIndex(values, current, endIndex)]) ;
}
```



Bubble Sort

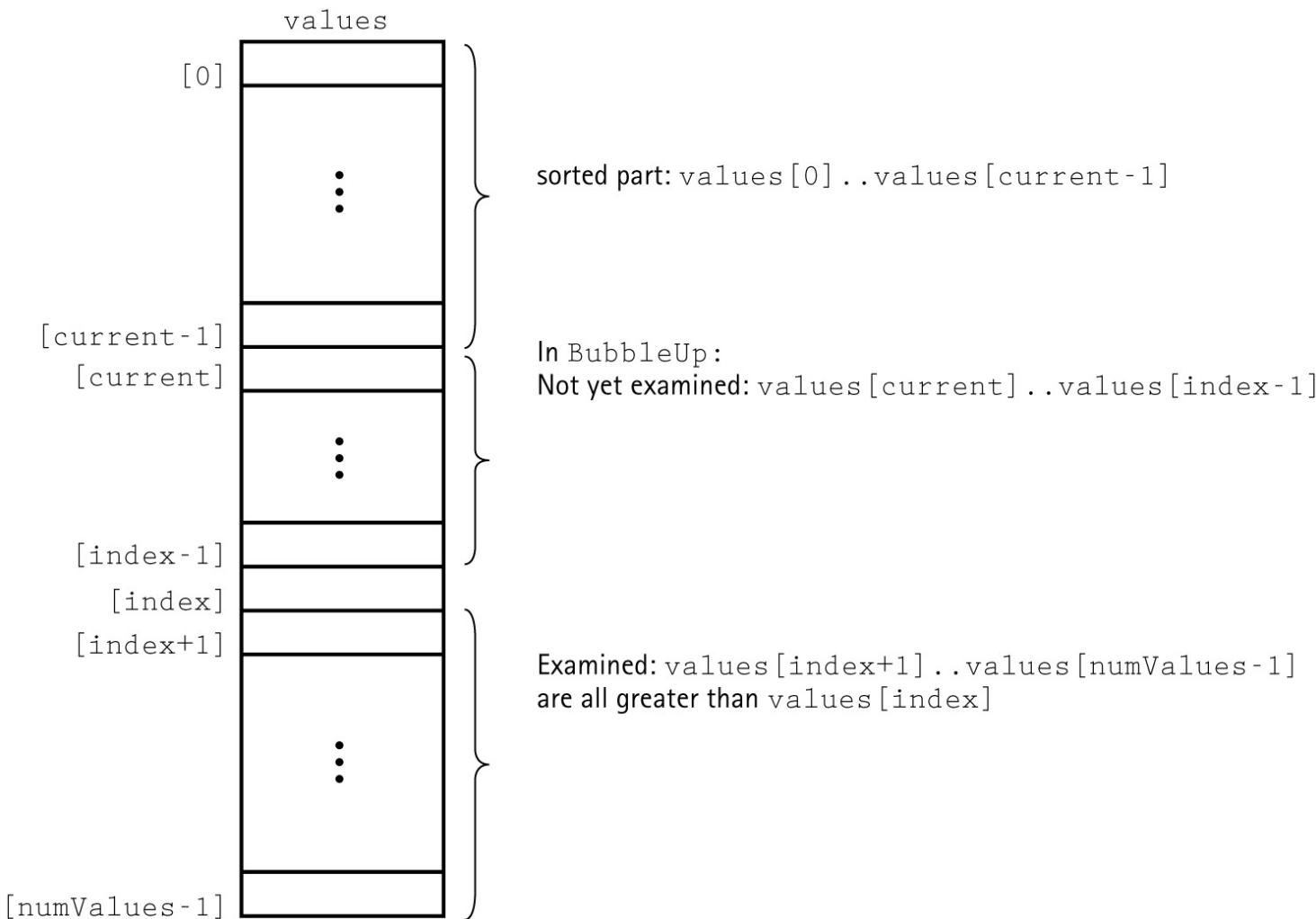
values	[0]
	36
[1]	24
[2]	10
[3]	6
[4]	12

Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to “bubble up” to its correct place in the array.



Snapshot of BubbleSort





Code for BubbleSort

```
template<class ItemType>
void BubbleSort(ItemType values[ ] ,
    int numValues)
{
    int current = 0;
    while (current < numValues - 1)
    {
        BubbleUp(values, current, numValues-1);
        current++;
    }
}
```



Code for BubbleUp

```
template<class ItemType>
void BubbleUp(ItemType values[],
    int startIndex, int endIndex)
// Post: Adjacent pairs that are out of
//       order have been switched between
//       values[startIndex] .. values[endIndex]
//       beginning at values[endIndex] .

{
    for (int index = endIndex;
        index > startIndex; index--)
        if (values[index] < values[index-1])
            Swap(values[index], values[index-1]);
}
```



Observations on BubbleSort

This algorithm is *always* $O(N^2)$.

There can be a large number of intermediate swaps.

Can this algorithm be improved?



Insertion Sort

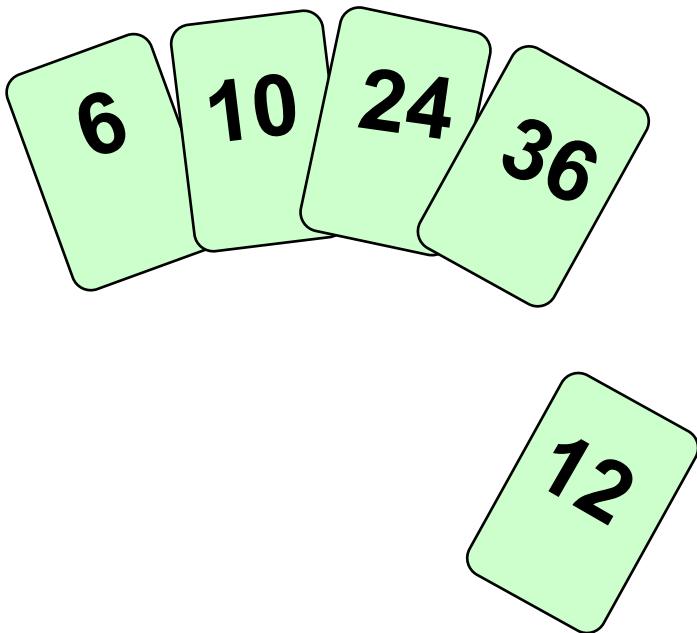
values [0] 36
 [1] 24
 [2] 10
 [3] 6
 [4] 12

One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.

On each pass, this causes the number of already sorted elements to increase by one.



Insertion Sort

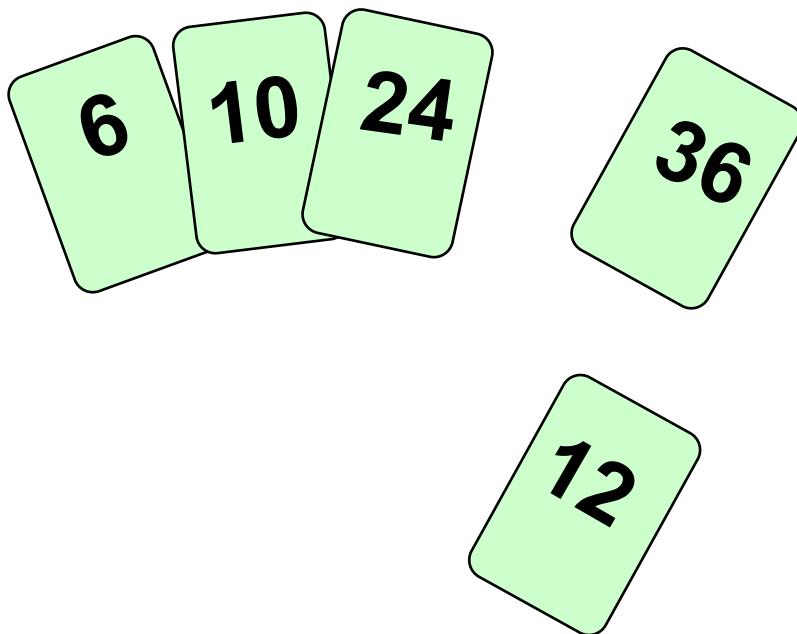


Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.



Insertion Sort

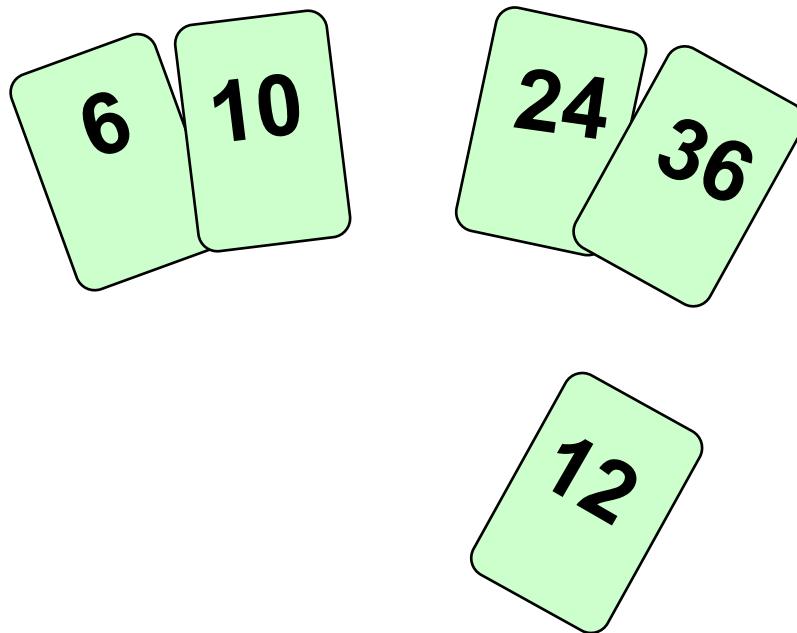


Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.



Insertion Sort

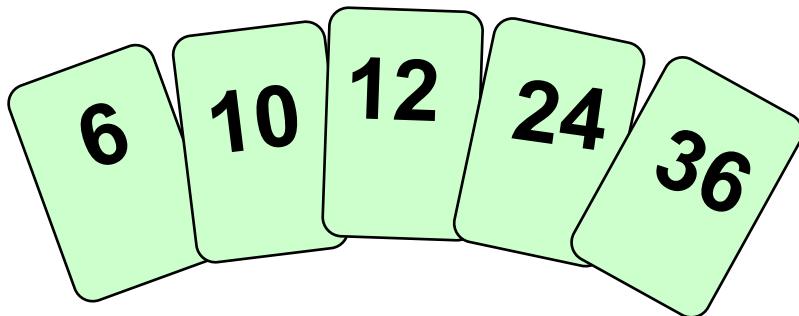


Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.



Insertion Sort

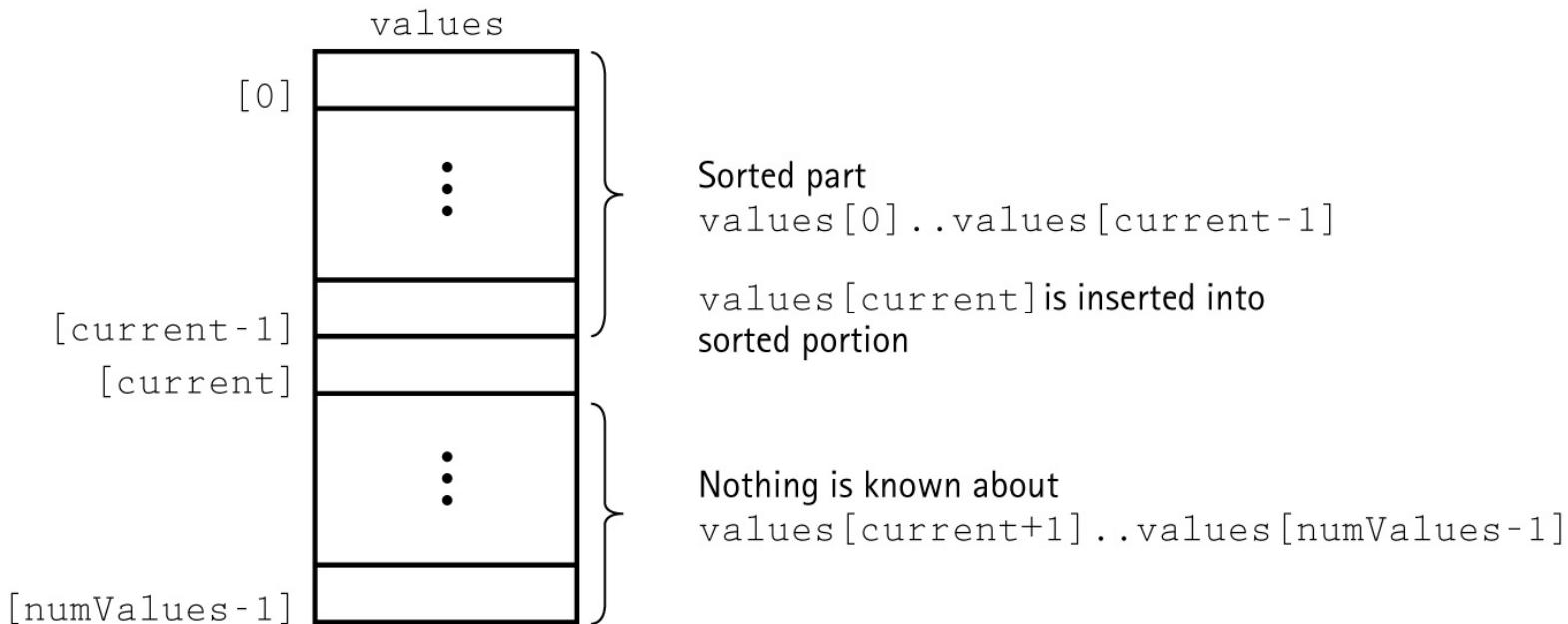


Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.



A Snapshot of the Insertion Sort Algorithm





```
template <class ItemType>
void InsertItem ( ItemType values [ ] , int start ,
int end )
// Post: Elements between values[start] and values
// [end] have been sorted into ascending order by key.
{
    bool finished = false ;
    int current = end ;
    bool moreToSearch = (current != start) ;

    while (moreToSearch && !finished )
    {
        if (values[current] < values[current - 1])
        {
            Swap(values[current] , values[current - 1]);
            current--;
            moreToSearch = ( current != start );
        }
        else
            finished = true ;
    }
}
```



```
template <class ItemType>
void InsertionSort ( ItemType values [ ] ,
int numValues )

// Post: Sorts array values[0 . . numValues-1] into
// ascending order by key
{
    for (int count = 0 ; count < numValues; count++)

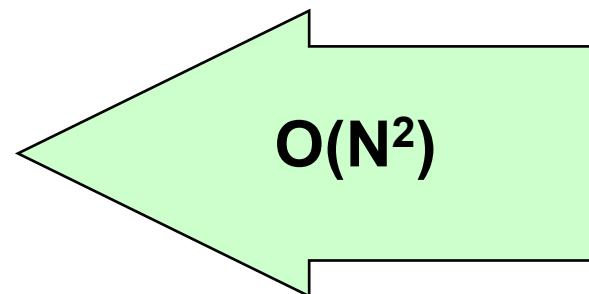
        InsertItem ( values , 0 , count ) ;
}
```



Sorting Algorithms and Average Case Number of Comparisons

Simple Sorts

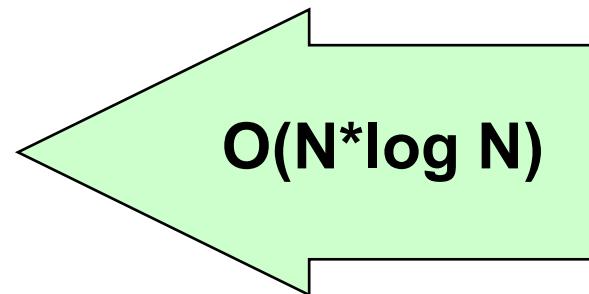
- Straight Selection Sort
- Bubble Sort
- Insertion Sort



$O(N^2)$

More Complex Sorts

- Quick Sort
- Merge Sort
- Heap Sort



$O(N * \log N)$



Recall that . . .

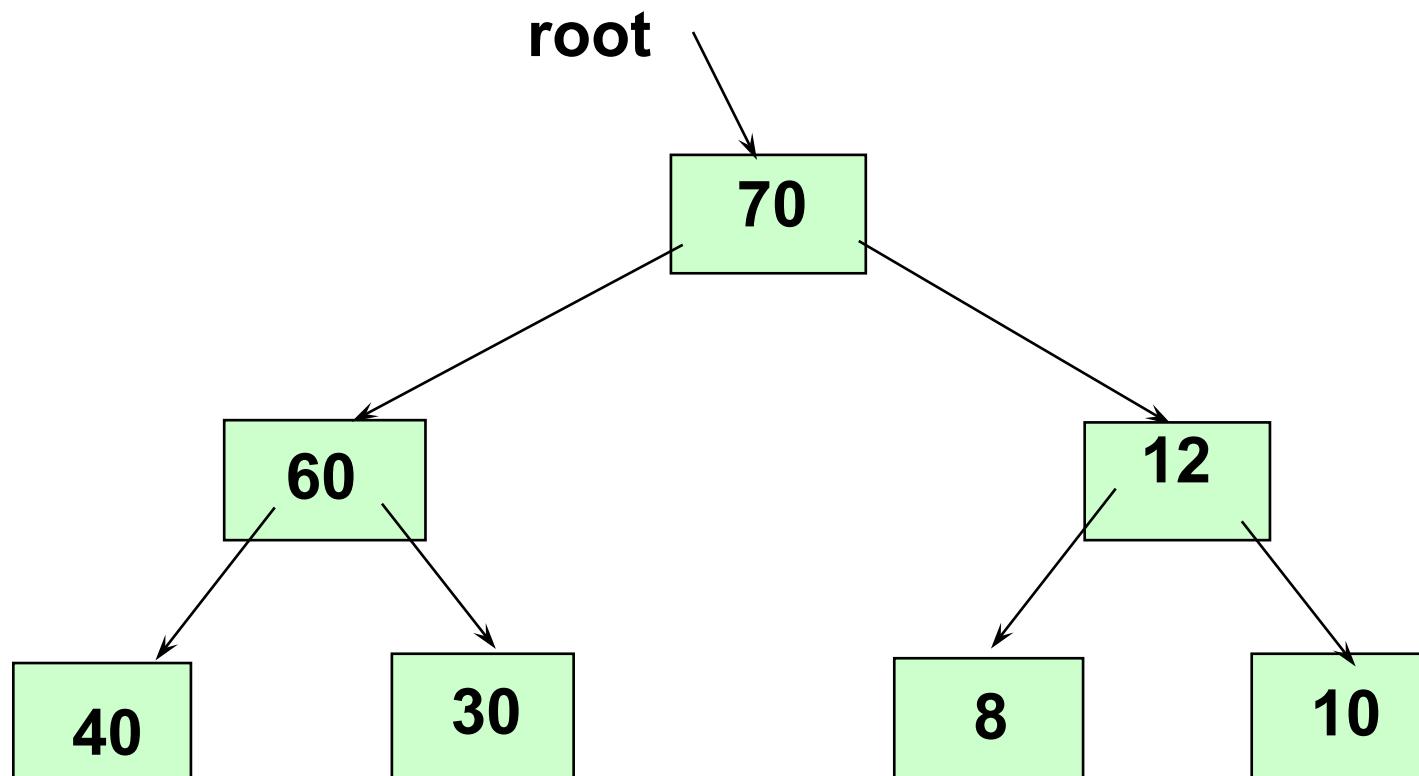
A heap is a binary tree that satisfies these special **SHAPE** and **ORDER** properties:

- **Its shape must be a complete binary tree.**
- **For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**



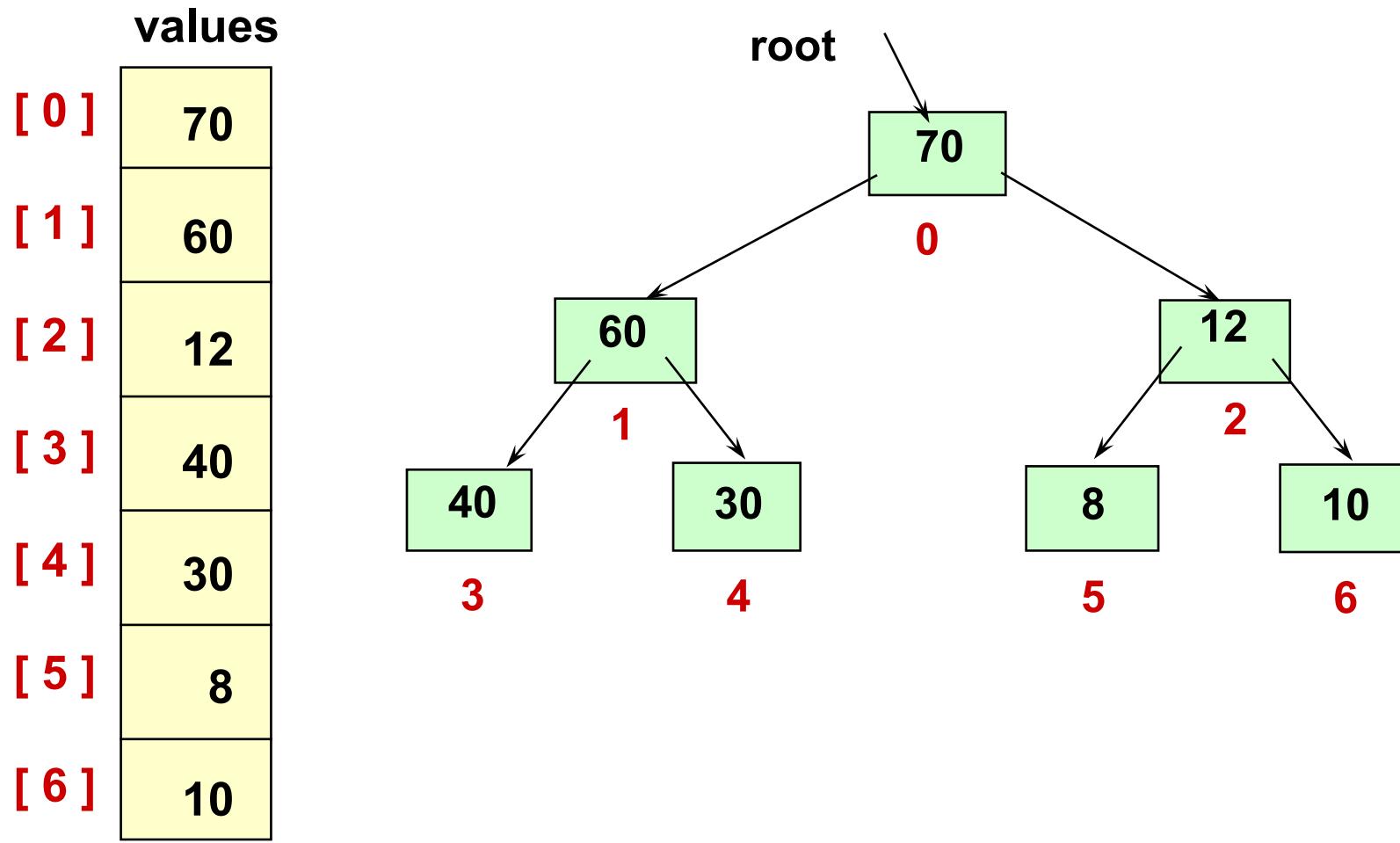
The largest element in a heap

is always found in the root node





The heap can be stored in an array





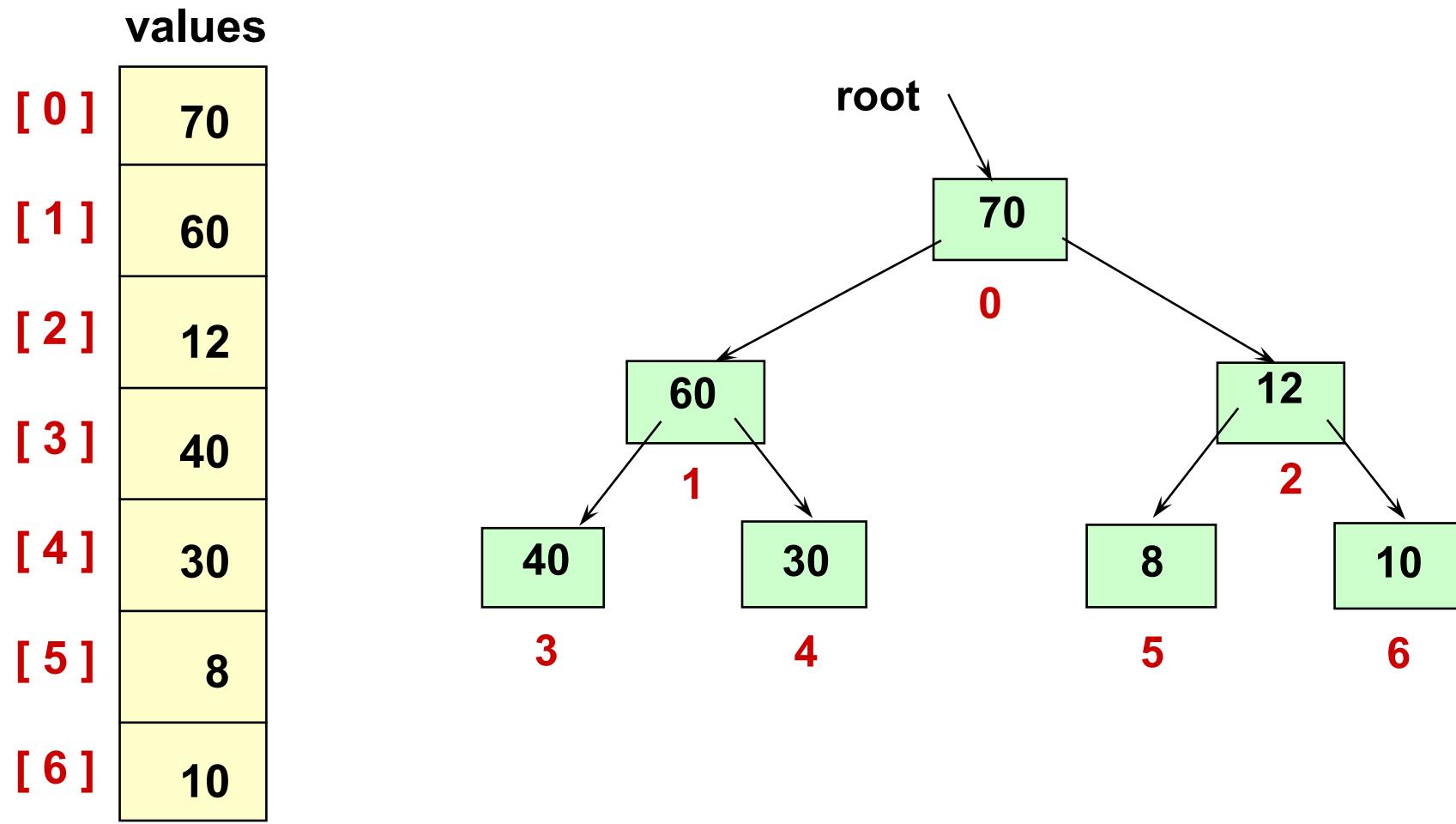
Heap Sort Approach

First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

- **Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.**
- **Reheap the remaining unsorted elements.**
(This puts the next-largest element into the root position).

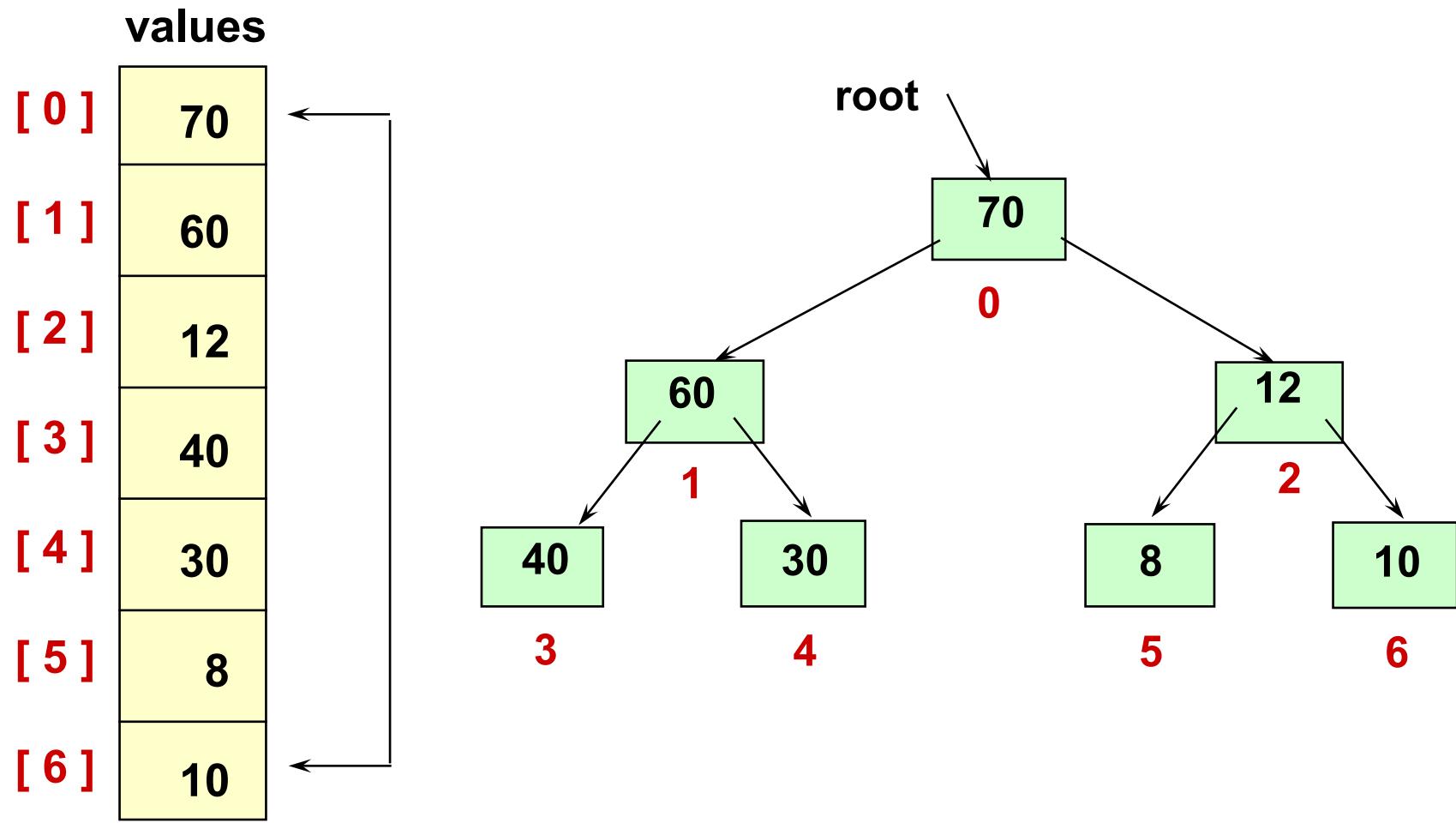


After creating the original heap

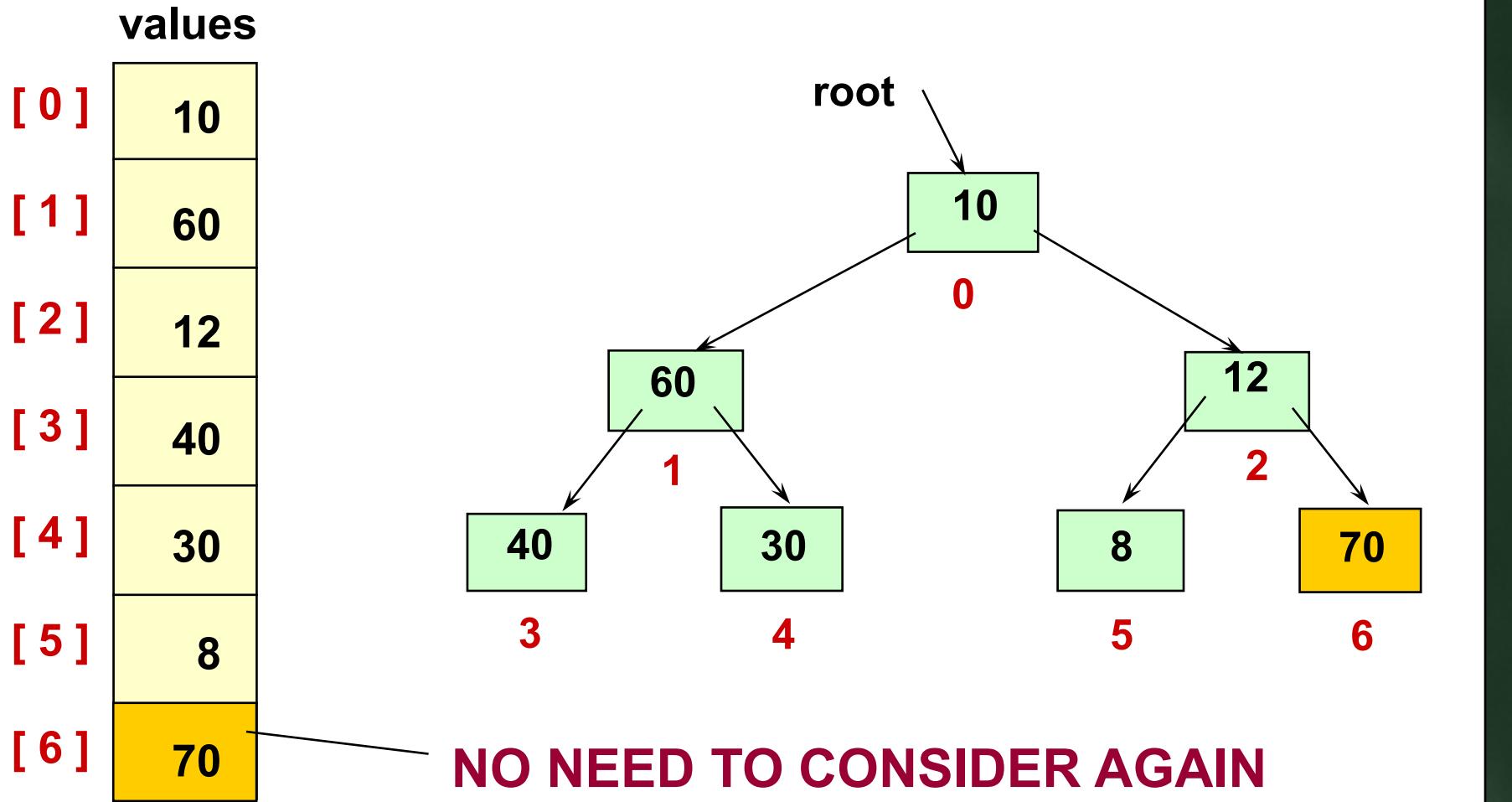




Swap root element into last place in unsorted array



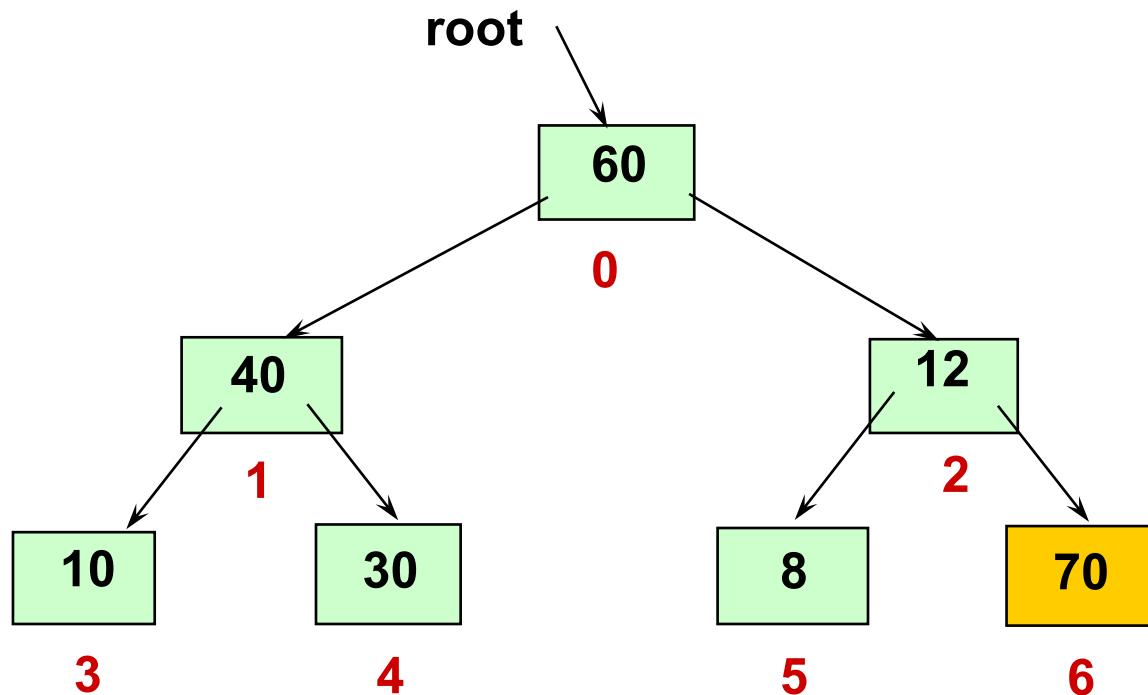
After swapping root element into its place





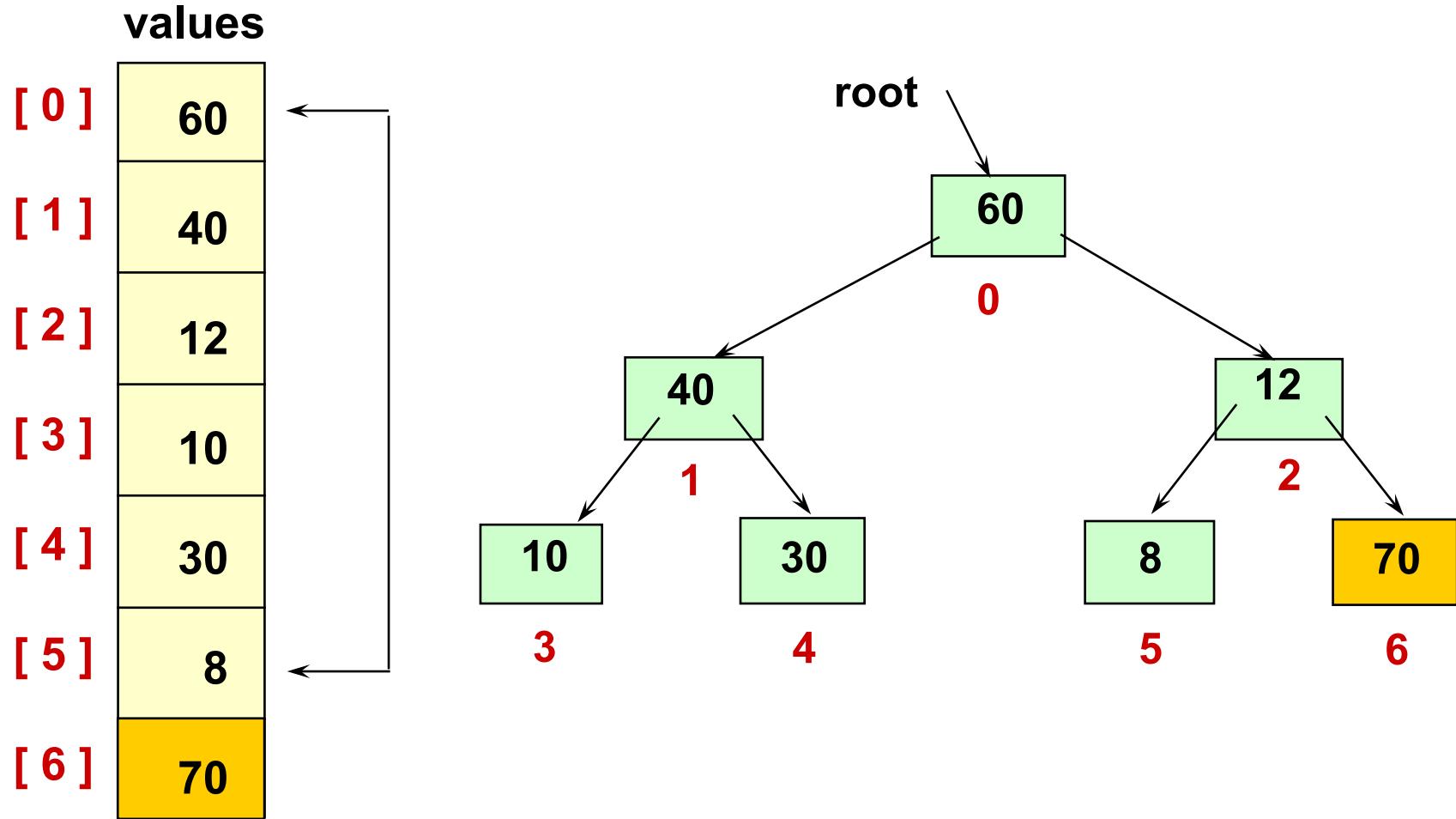
After reheaping remaining unsorted elements

values
[0]
60
[1]
40
[2]
12
[3]
10
[4]
30
[5]
8
[6]
70



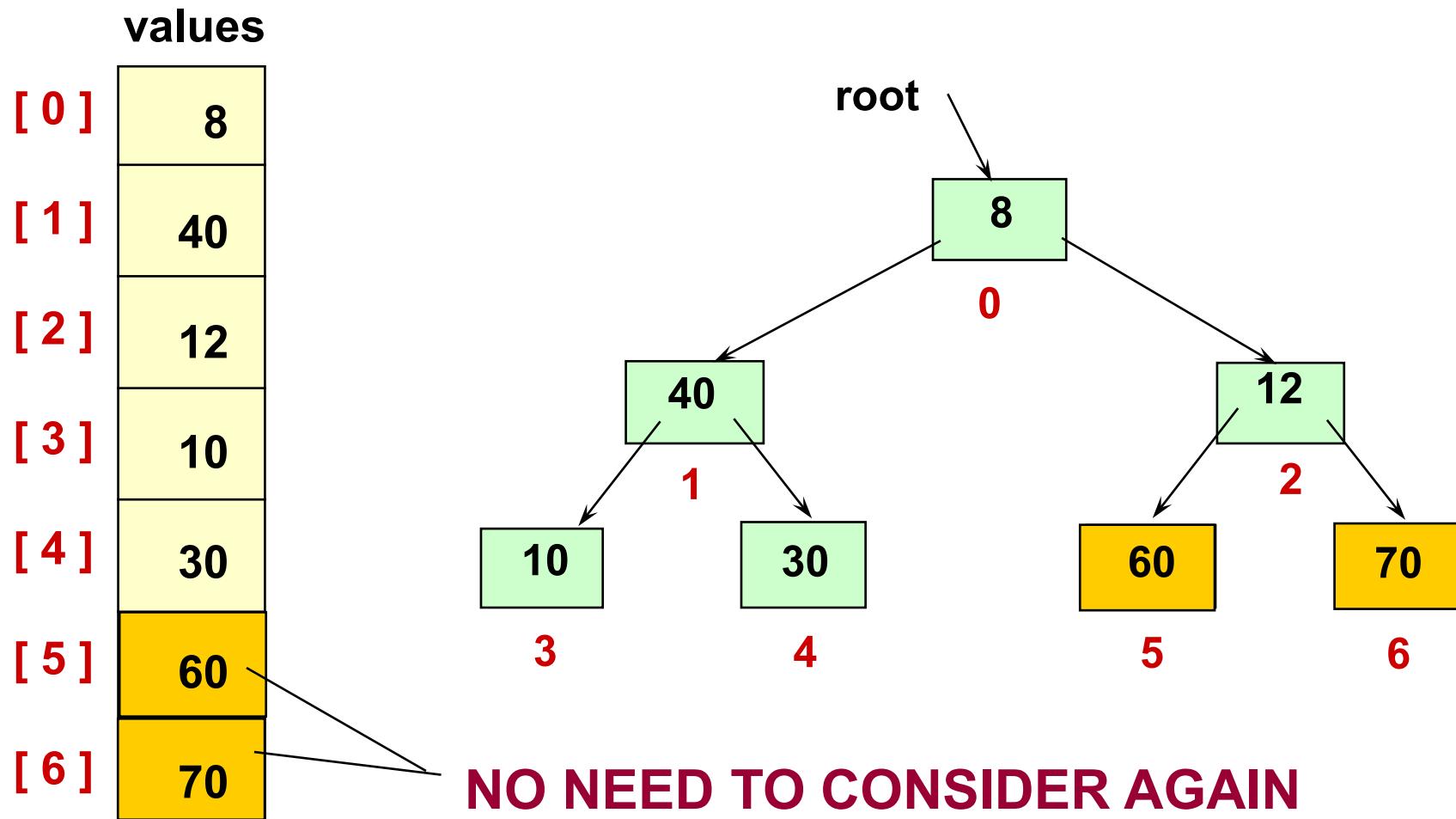


Swap root element into last place in unsorted array





After swapping root element into its place

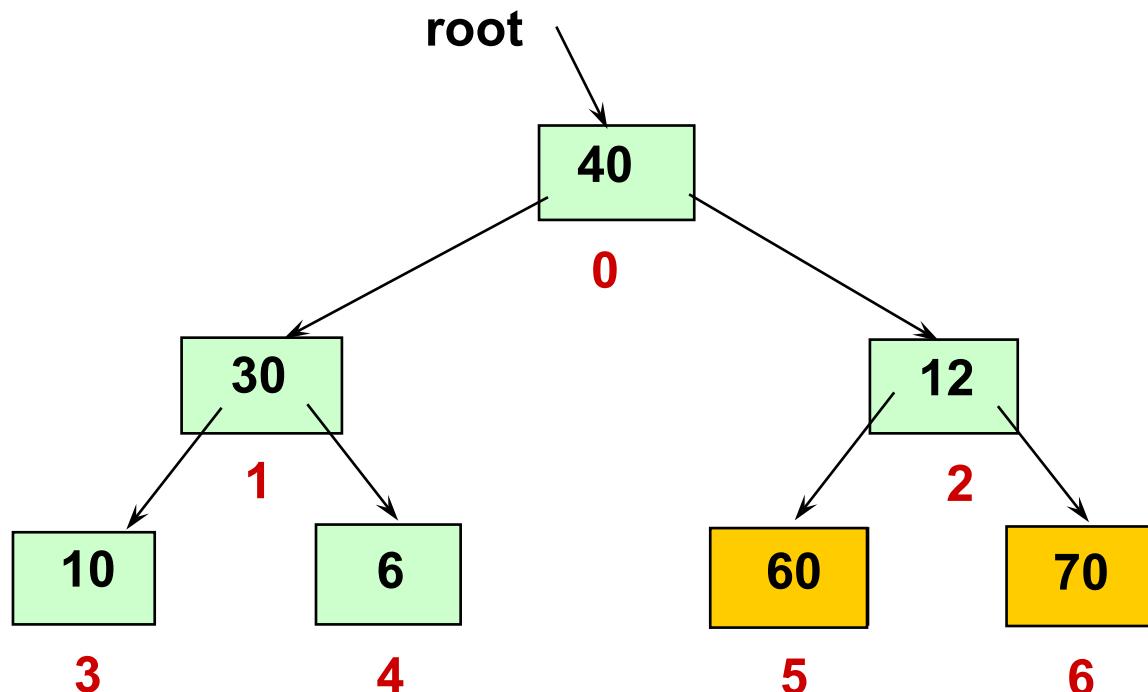




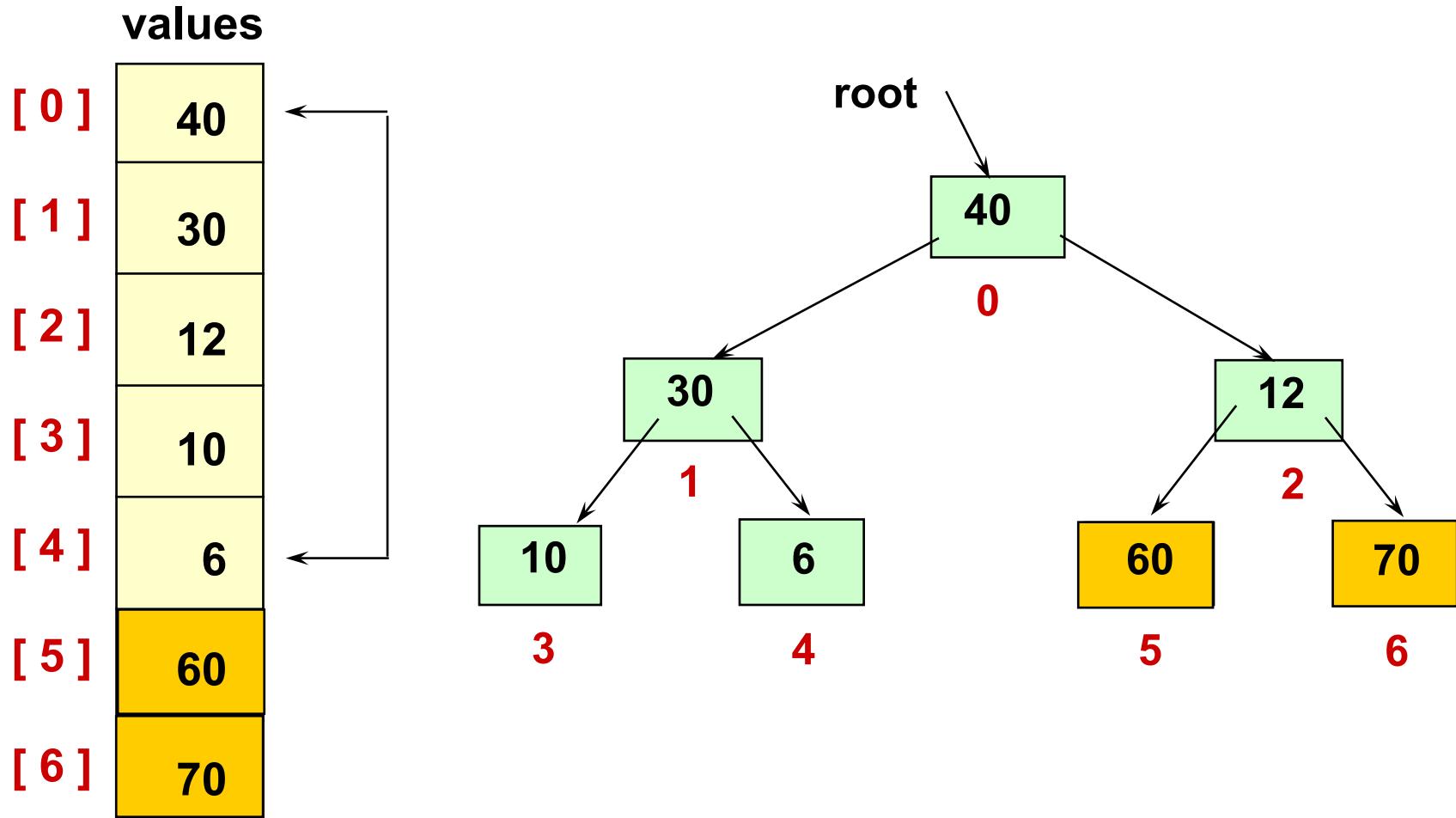
After reheaping remaining unsorted elements

values

[0]	40
[1]	30
[2]	12
[3]	10
[4]	6
[5]	60
[6]	70

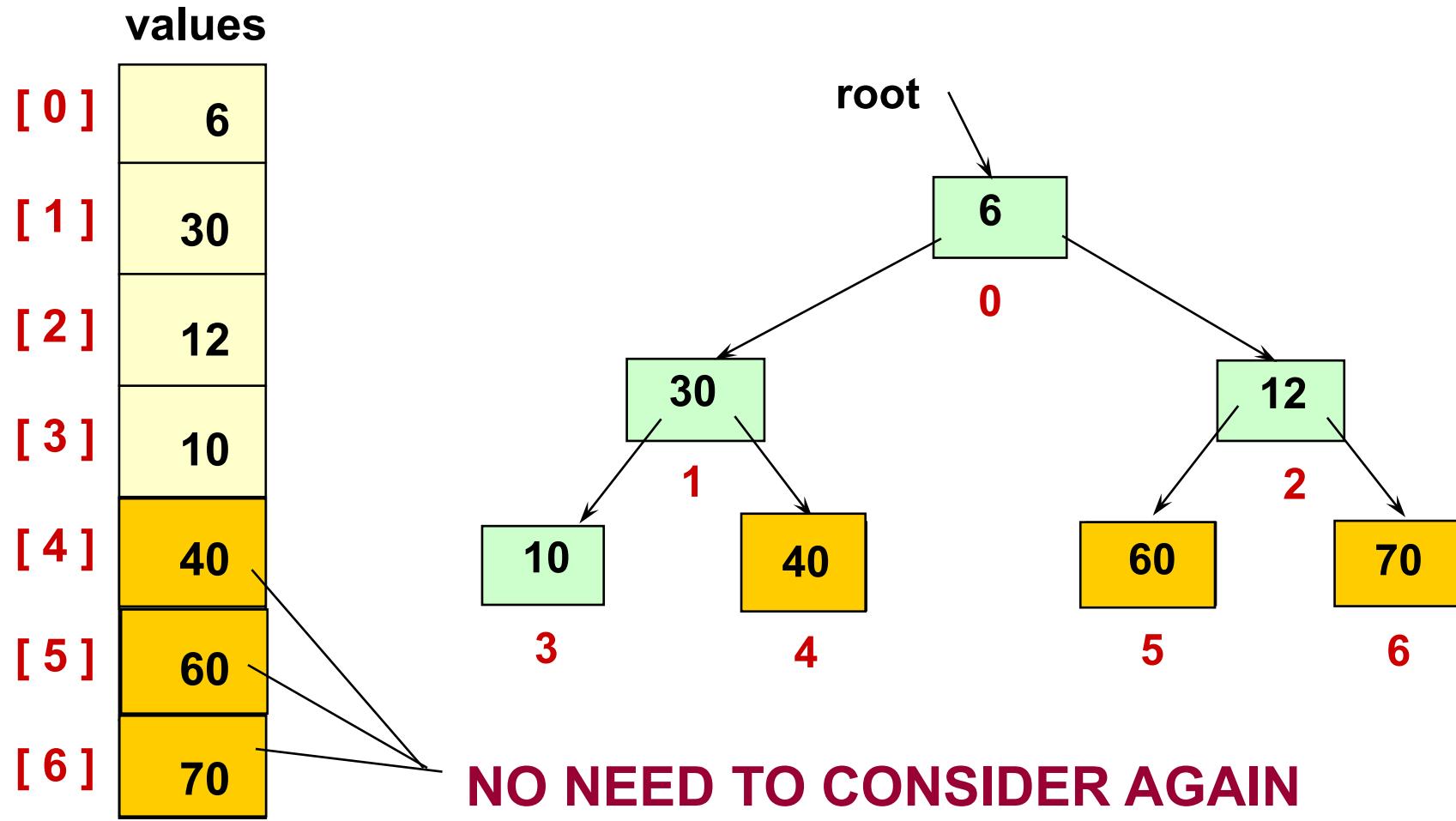


Swap root element into last place in unsorted array



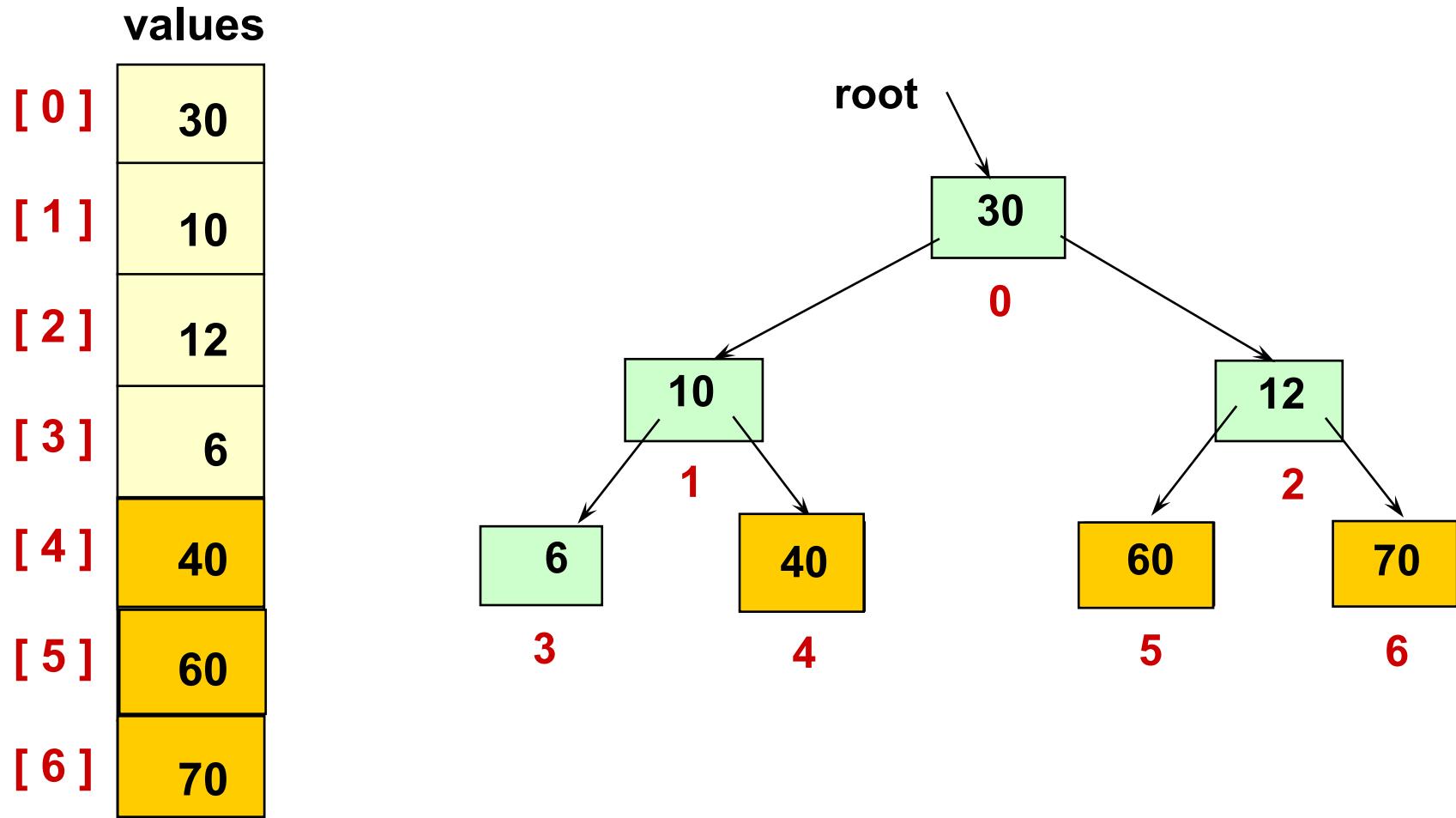


After swapping root element into its place

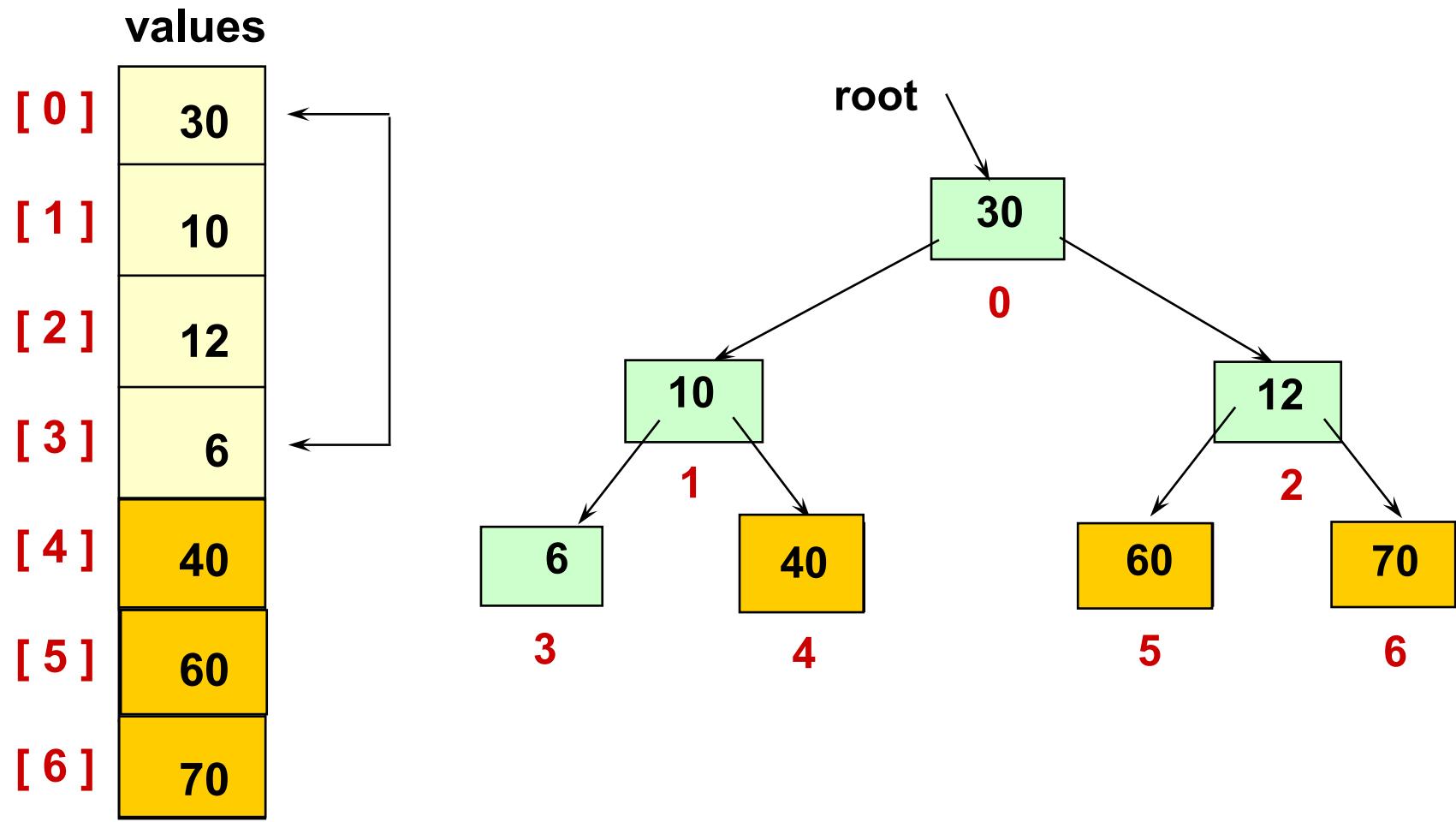




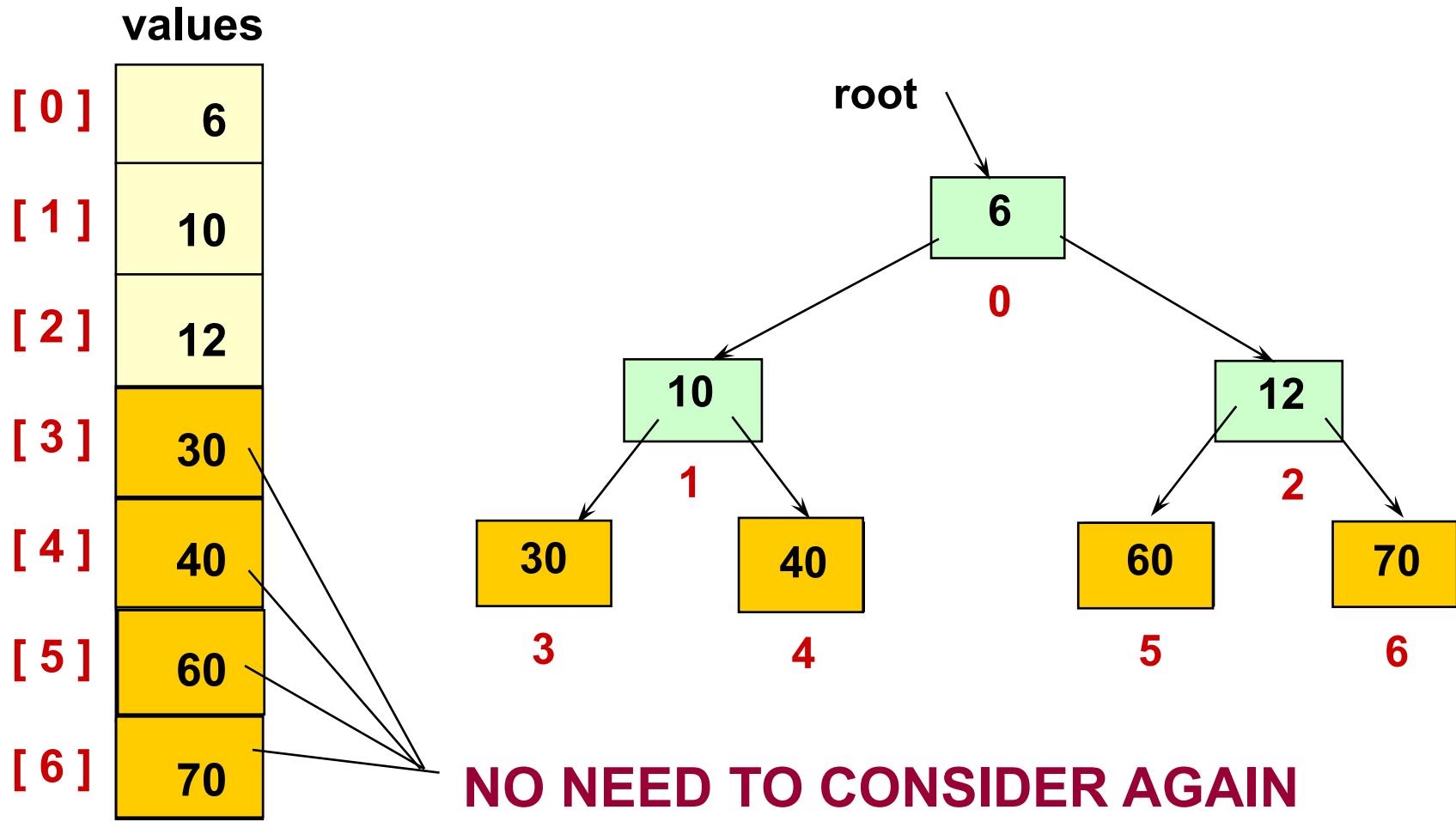
After reheapng remaining unsorted elements



Swap root element into last place in unsorted array

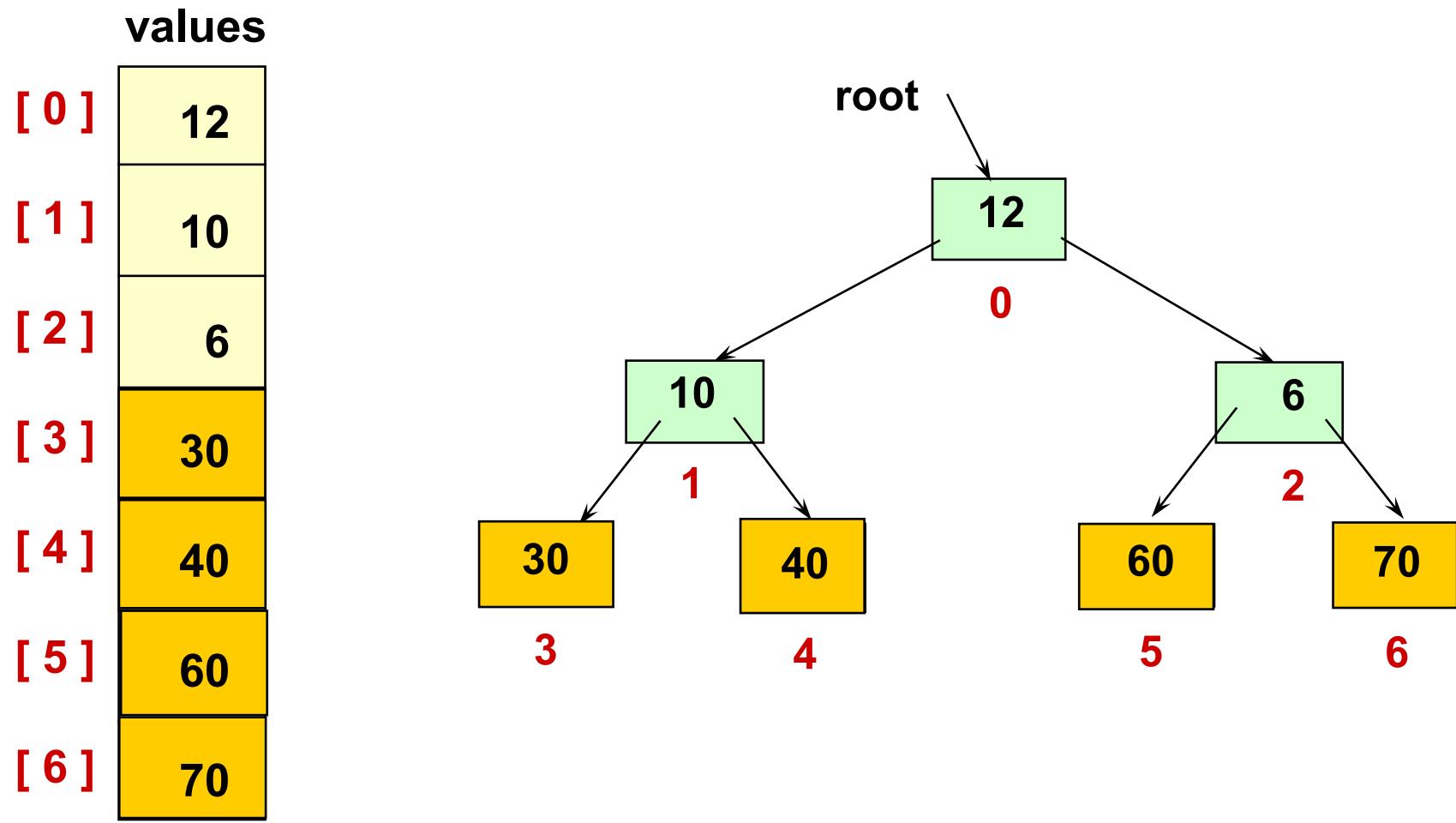


After swapping root element into its place



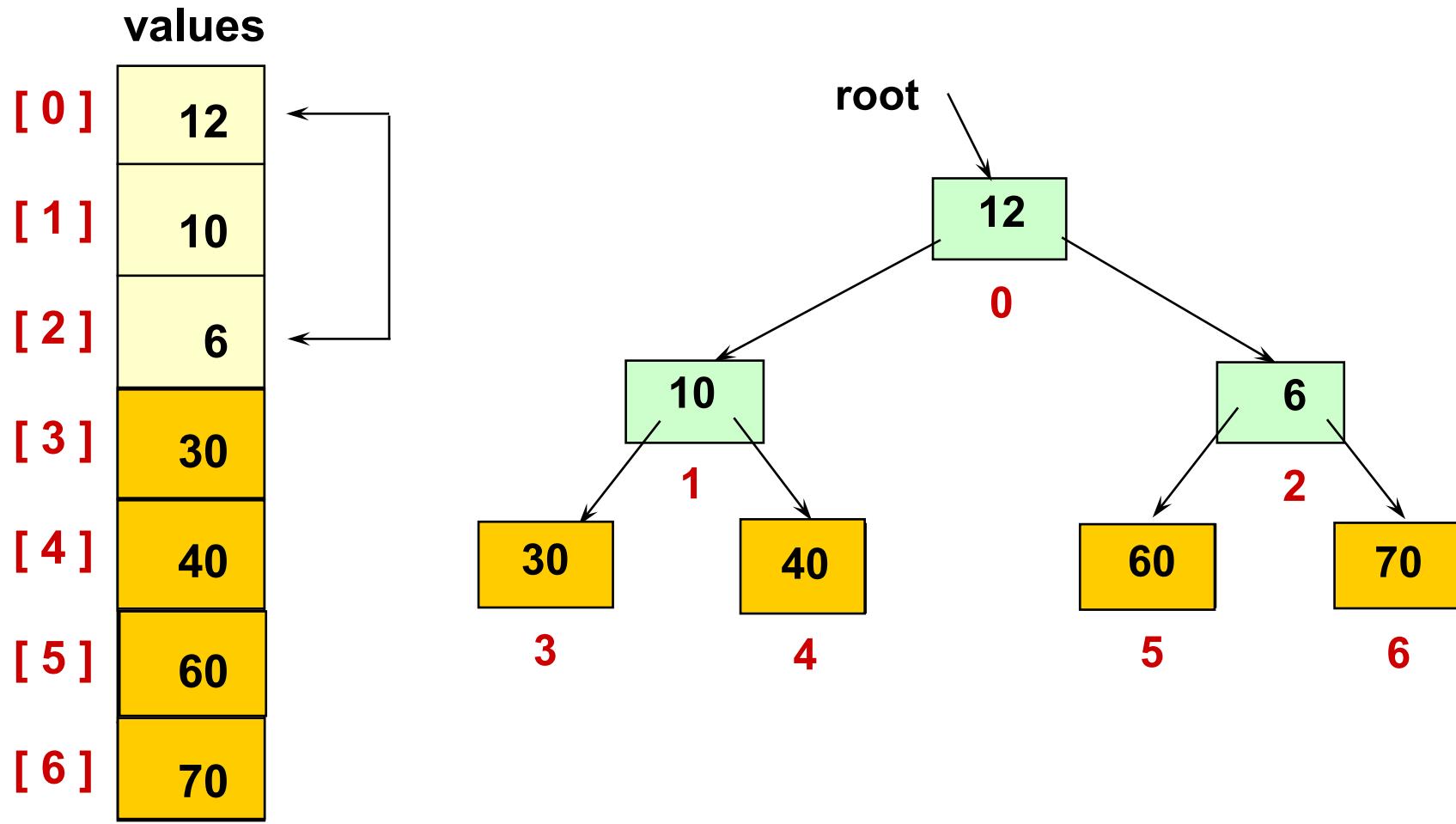


After reheaping remaining unsorted elements

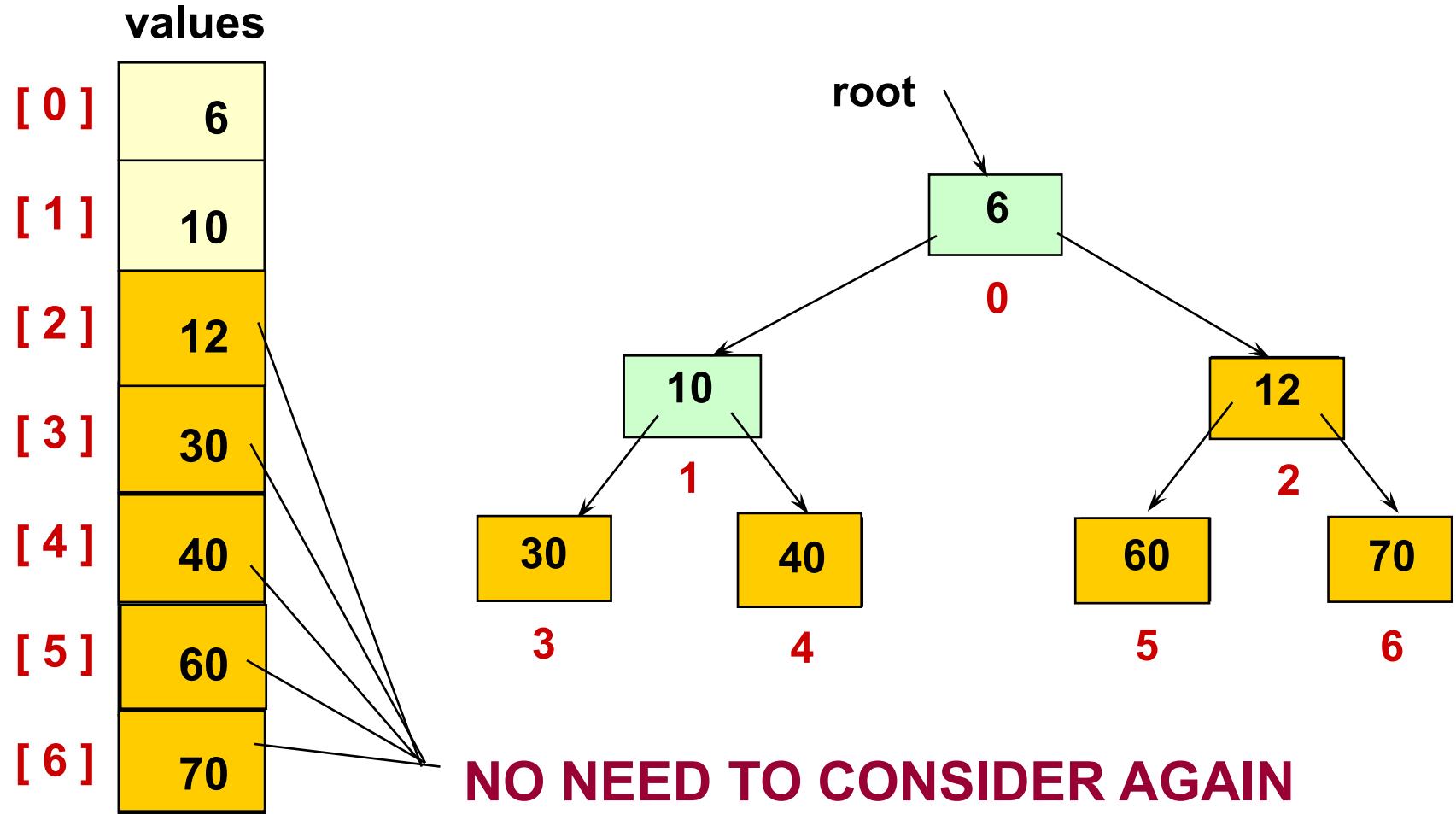




Swap root element into last place in unsorted array

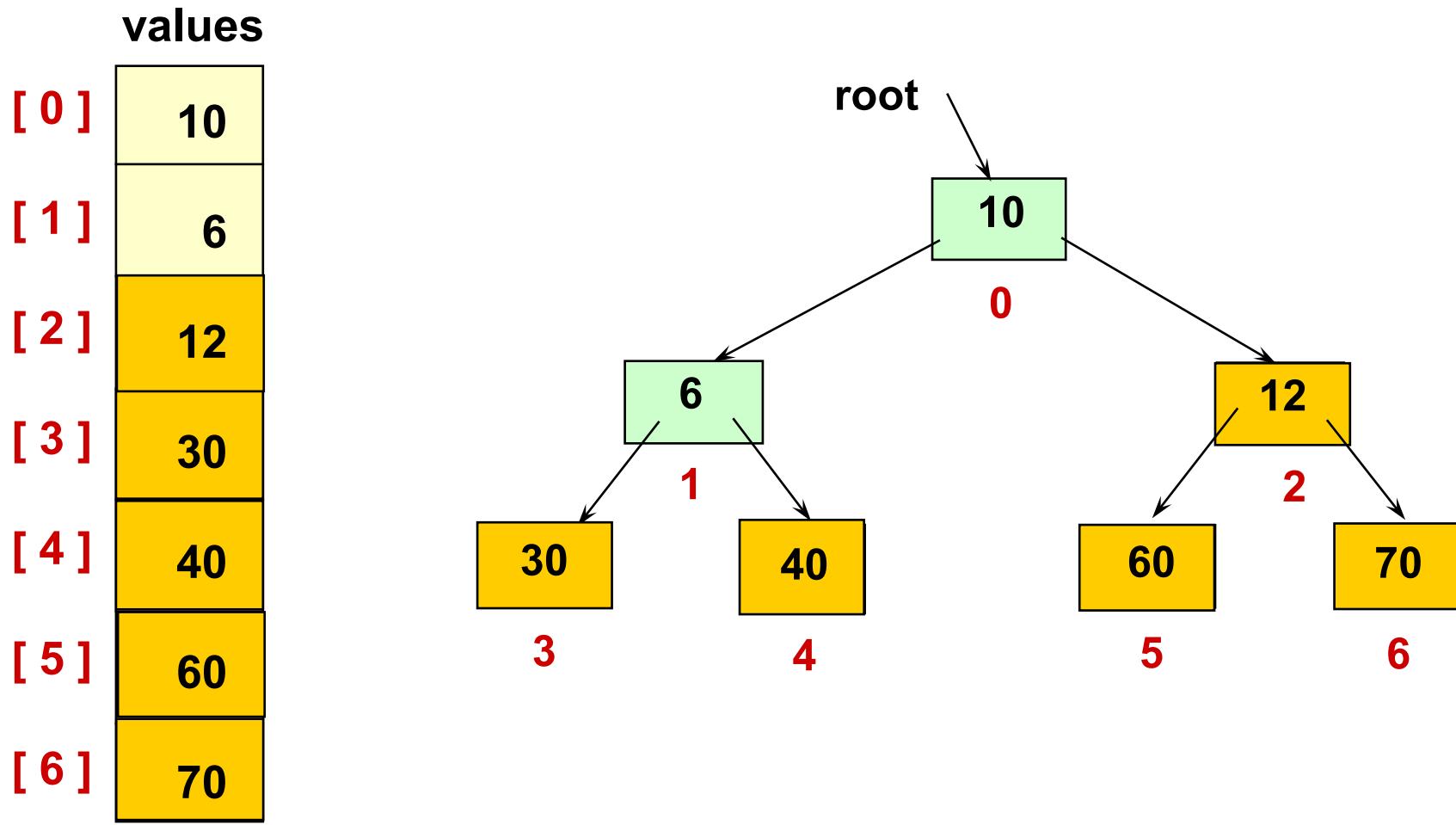


After swapping root element into its place

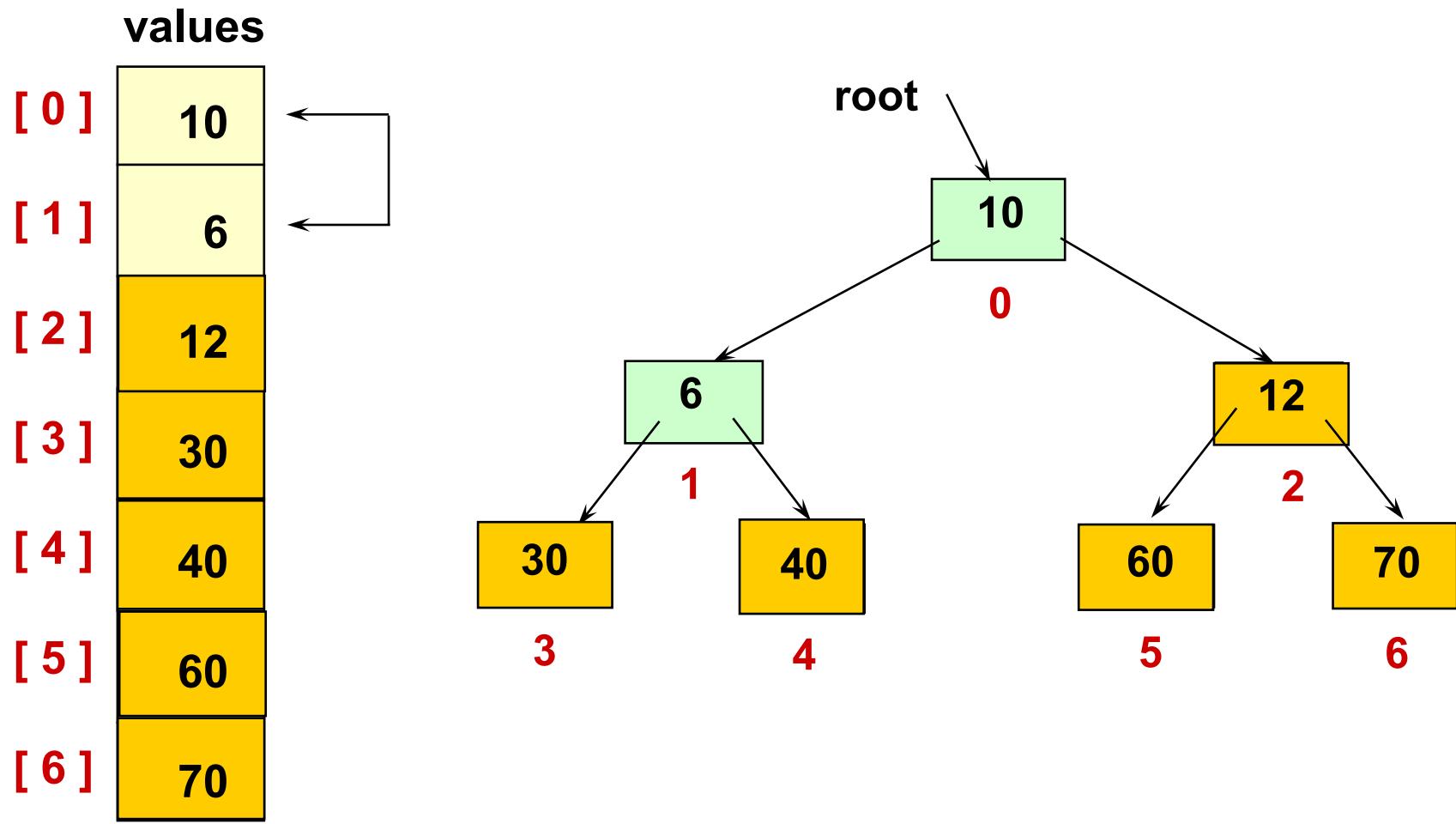




After reheaping remaining unsorted elements



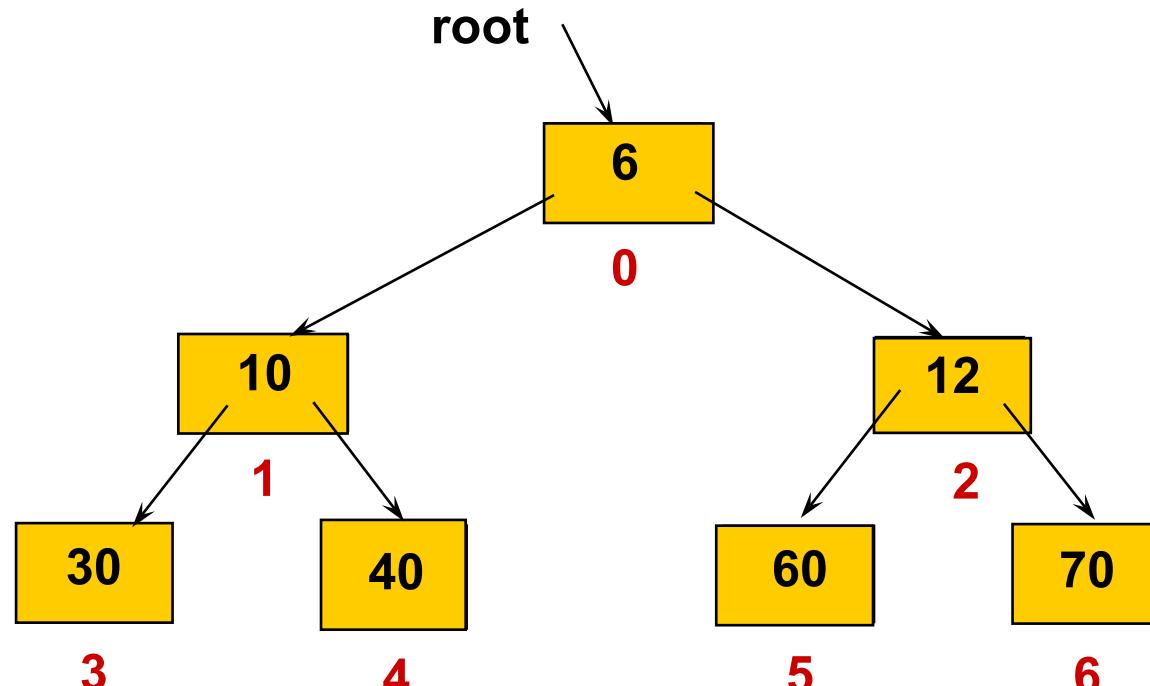
Swap root element into last place in unsorted array





After swapping root element into its place

values	
[0]	6
[1]	10
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70



ALL ELEMENTS ARE SORTED



```
template <class ItemType>
void HeapSort ( ItemType values [ ] , int
    numValues )
// Post: Sorts array values[ 0 . . numValues-1 ] into
// ascending order by key
{
    int index ;

    // Convert array values[0..numValues-1] into a heap
    for (index = numValues/2 - 1; index >= 0; index--)
        ReheapDown ( values , index , numValues - 1 ) ;

    // Sort the array.
    for (index = numValues - 1; index >= 1; index--)
    {
        Swap (values [0] , values [index]);
        ReheapDown (values , 0 , index - 1);
    }
}
```



ReheapDown

```
template< class ItemType >
void ReheapDown ( ItemType values [ ], int root,
                  int bottom )

// Pre: root is the index of a node that may violate the
//       heap order property
// Post: Heap order property is restored between root and
//       bottom

{
    int maxChild ;
    int rightChild ;
    int leftChild ;

    leftChild = root * 2 + 1 ;
    rightChild = root * 2 + 2 ;
```



```
if (leftChild <= bottom)      // ReheapDown continued
{
    if (leftChild == bottom)
        maxChild = leftChild;
    else
    {
        if (values[leftChild] <= values [rightChild])
            maxChild = rightChild ;
        else
            maxChild = leftChild ;
    }
    if (values[ root ] < values[maxChild])
    {
        Swap (values[root] , values[maxChild]);
        ReheapDown ( maxChild, bottom  ;
    }
}
```



Building Heap From Unsorted Array

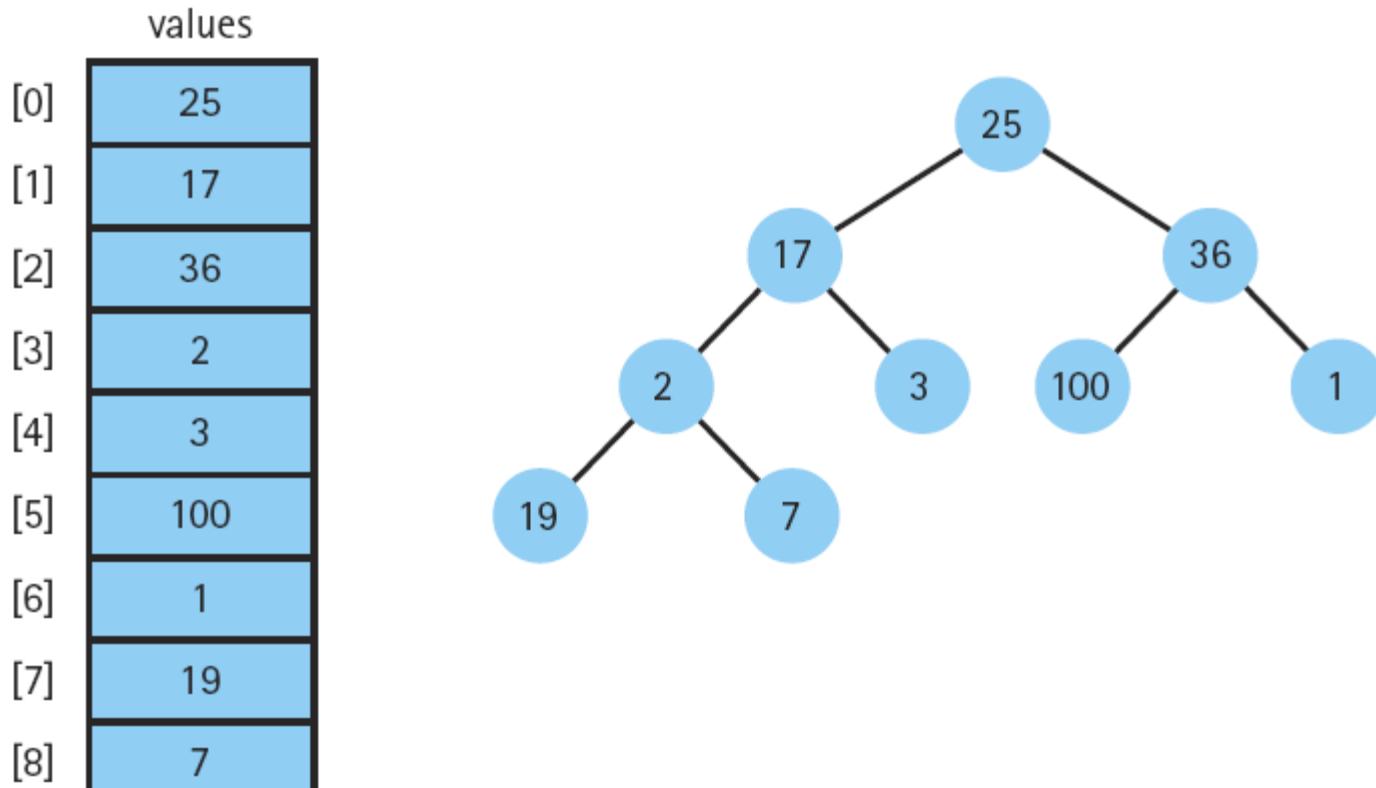
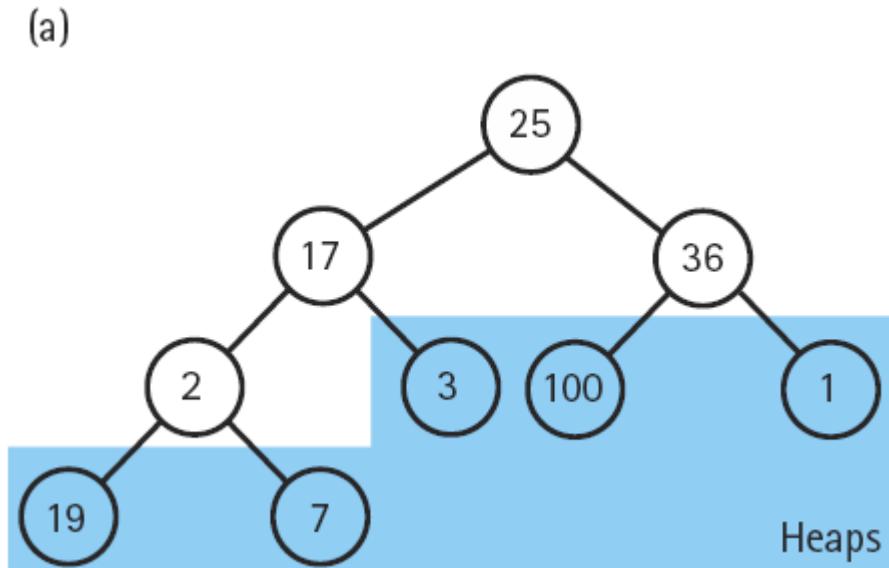


Figure 10.12 An unsorted array and its tree

Building Heap From Unsorted Array (cont'd)

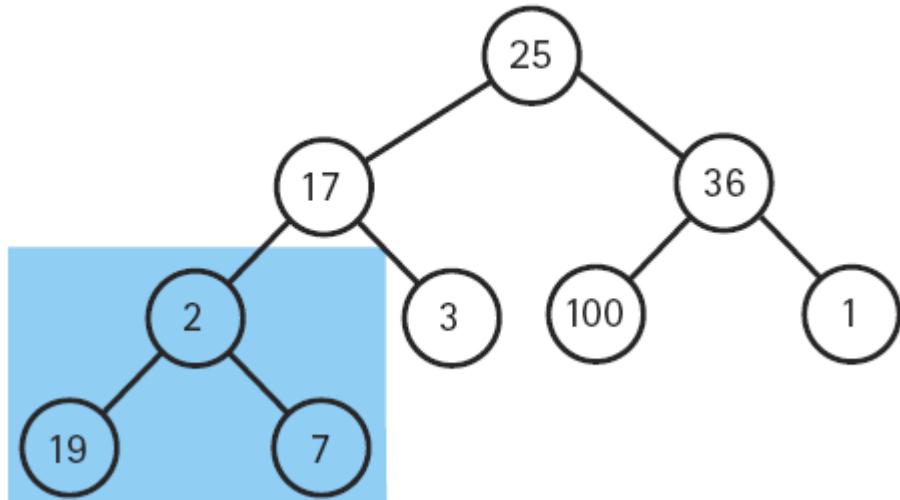
- Leaf nodes are already heaps



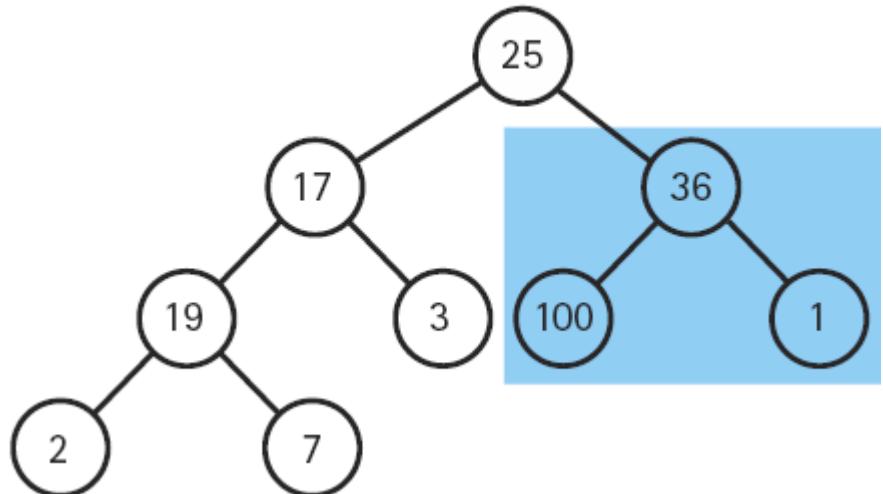
Building Heap From Unsorted Array (cont'd)

- The subtrees rooted at first nonleaf nodes are almost heaps

(b)



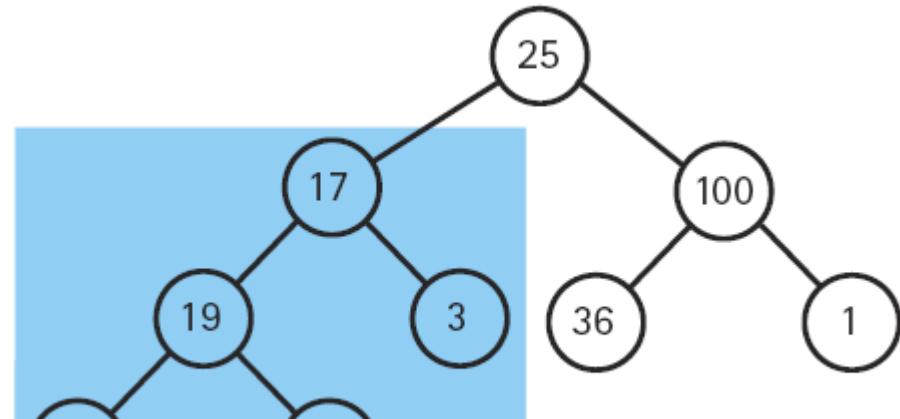
(c)



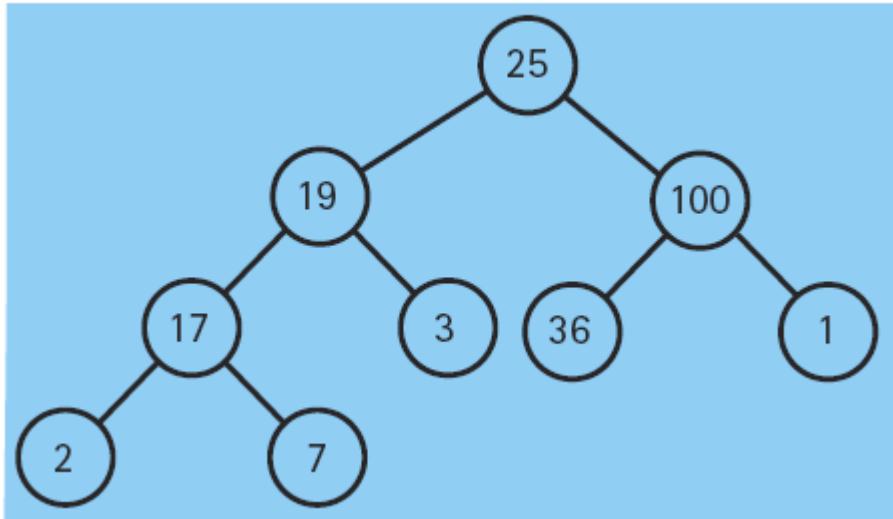
Building Heap From Unsorted Array (cont'd)

- Move up a level in the tree and continue reheapaging until we reach the root node

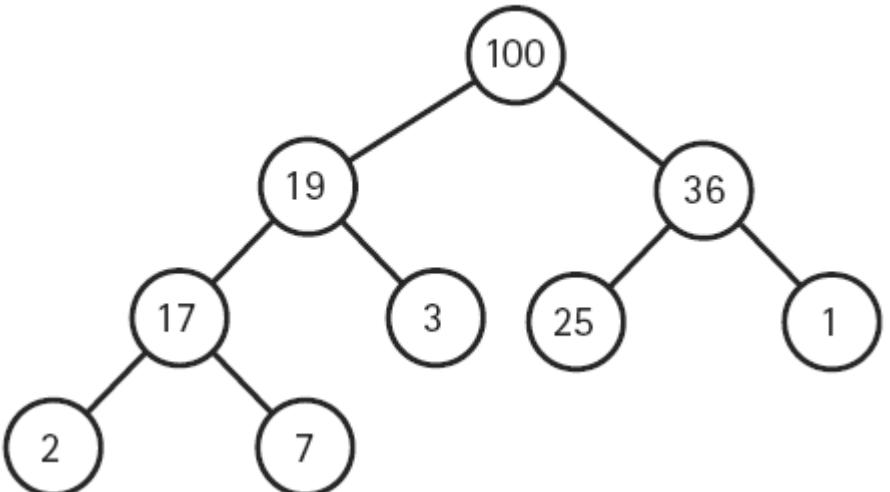
(d)



(e)

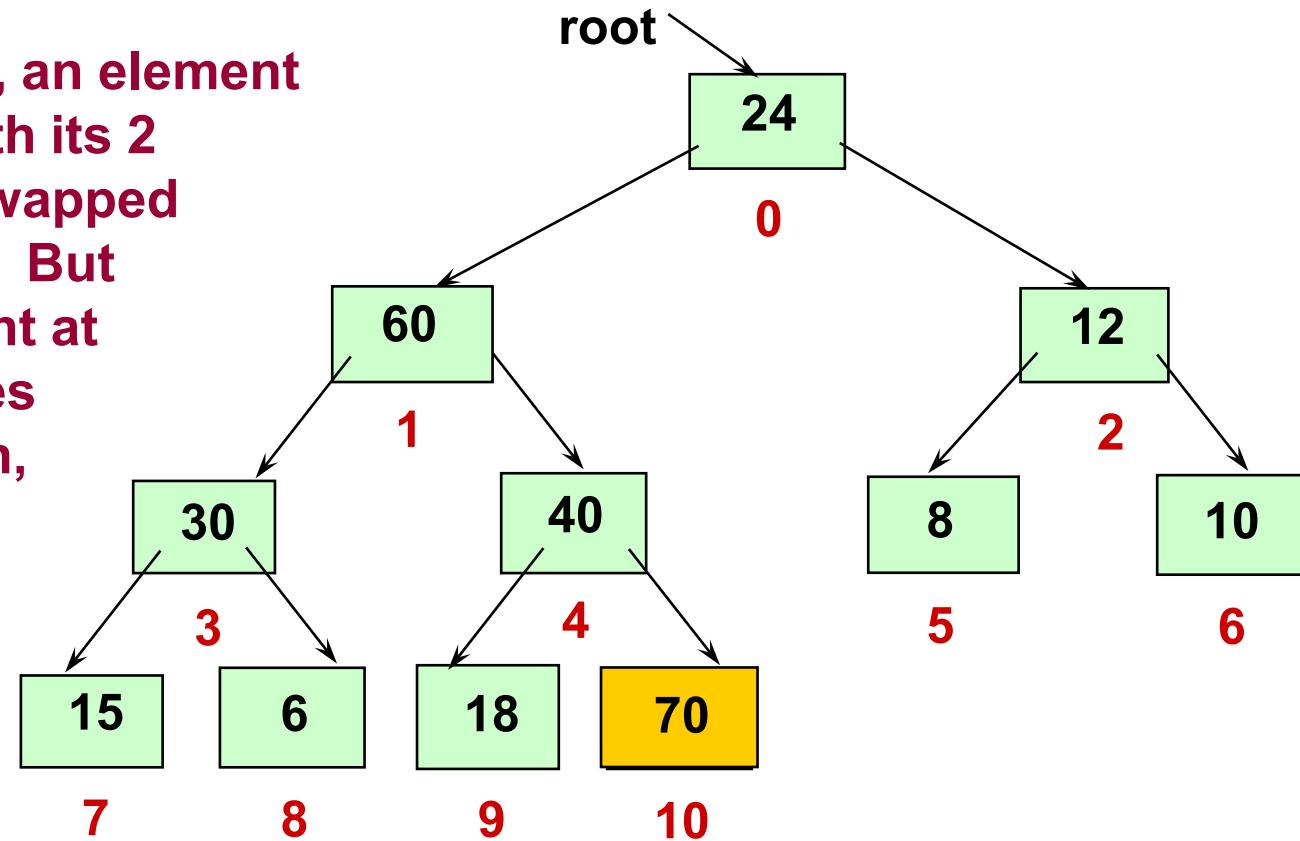


(f) Tree now represents a heap



Heap Sort: How many comparisons?

In reheap down, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only $O(\log_2 N)$ levels.





Heap Sort of N elements: How many comparisons?

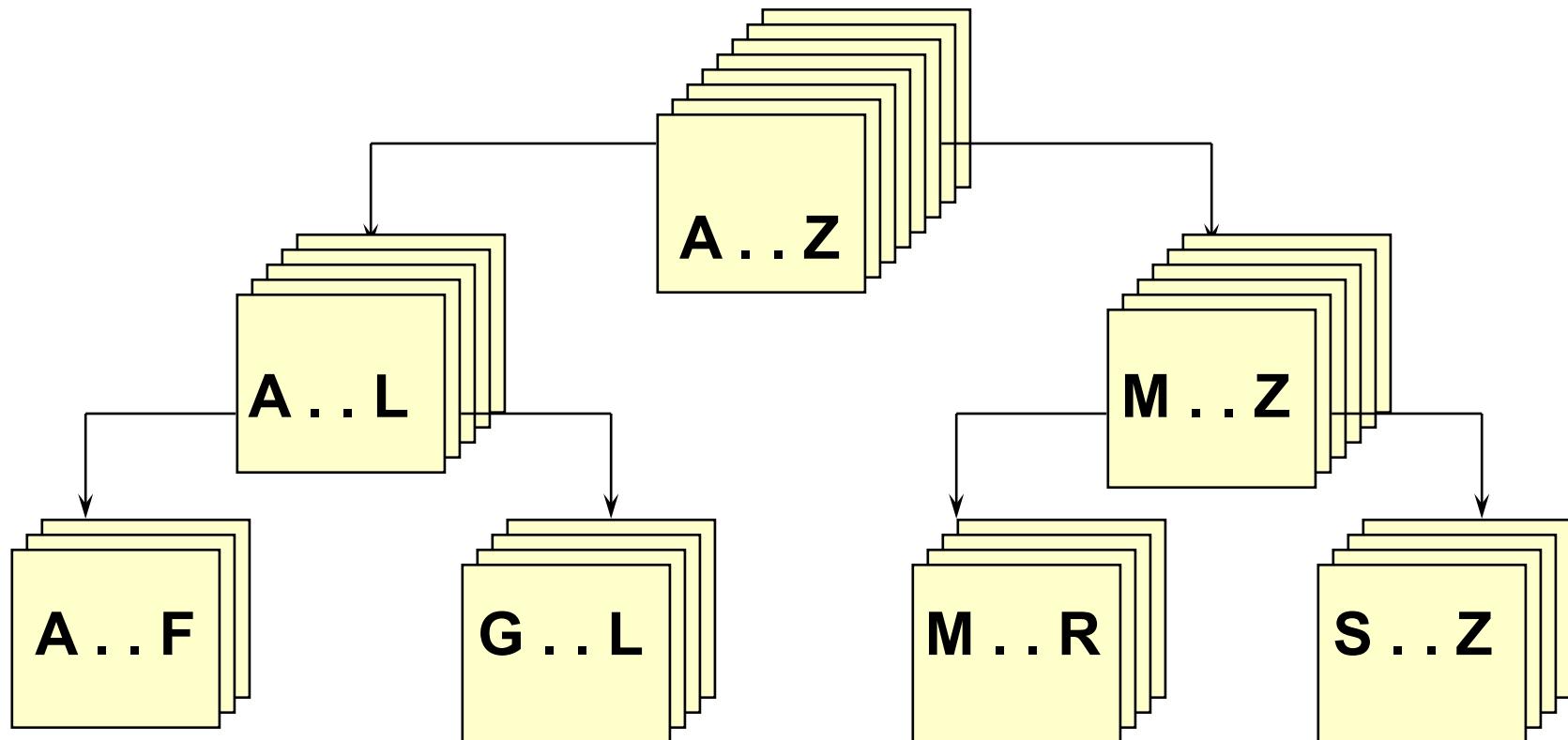
$(N/2) * O(\log N)$ compares to create original heap

$(N-1) * O(\log N)$ compares for the sorting loop

= $O(N * \log N)$ compares total



Using quick sort algorithm





```
// Recursive quick sort algorithm

template <class ItemType>
void QuickSort ( ItemType values[ ] , int first ,
                 int last )

// Pre: first <= last
// Post: Sorts array values[ first . . . last ] into
       ascending order
{
    if ( first < last )                                // general case
    {
        int splitPoint ;
        Split ( values, first, last, splitPoint ) ;
        // values [first]..values[splitPoint - 1] <= splitVal
        // values [splitPoint] = splitVal
        // values [splitPoint + 1]..values[last] > splitVal
        QuickSort(values, first, splitPoint - 1);
        QuickSort(values, splitPoint + 1, last);
    }
} ;
```



Before call to function Split

splitVal = 9

GOAL: place `splitVal` in its proper position with
all values less than or equal to `splitVal` on its left
and all larger values on its right

9	20	6	18	14	3	60	11
---	----	---	----	----	---	----	----

`values[first]`

`[last]`

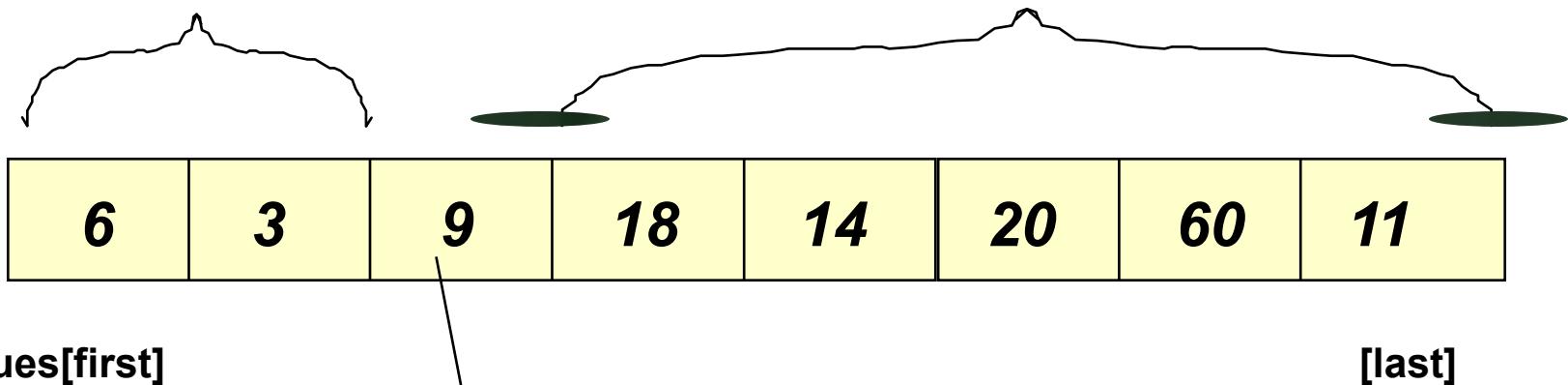


After call to function Split

splitVal = 9

**smaller values
in left part**

**larger values
in right part**



splitVal in correct position



Quick Sort of N elements: How many comparisons?

N For first call, when each of N elements
 is compared to the split value

$2 * N/2$ For the next pair of calls, when $N/2$
 elements in each “half” of the original
 array are compared to their own split values.

$4 * N/4$ For the four calls when $N/4$ elements in each
 “quarter” of original array are compared to
 their own split values.

.

.

.

HOW MANY SPLITS CAN OCCUR?



Quick Sort of N elements: How many splits can occur?

It depends on the order of the original array elements!

If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N * \log_2 N)$.

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as $N-1$ splits, and QuickSort is $O(N^2)$.



Before call to function Split

splitVal = 9

GOAL: place `splitVal` in its proper position with
all values less than or equal to `splitVal` on its left
and all larger values on its right

9	20	26	18	14	53	60	11
---	----	----	----	----	----	----	----

values[first]

[last]

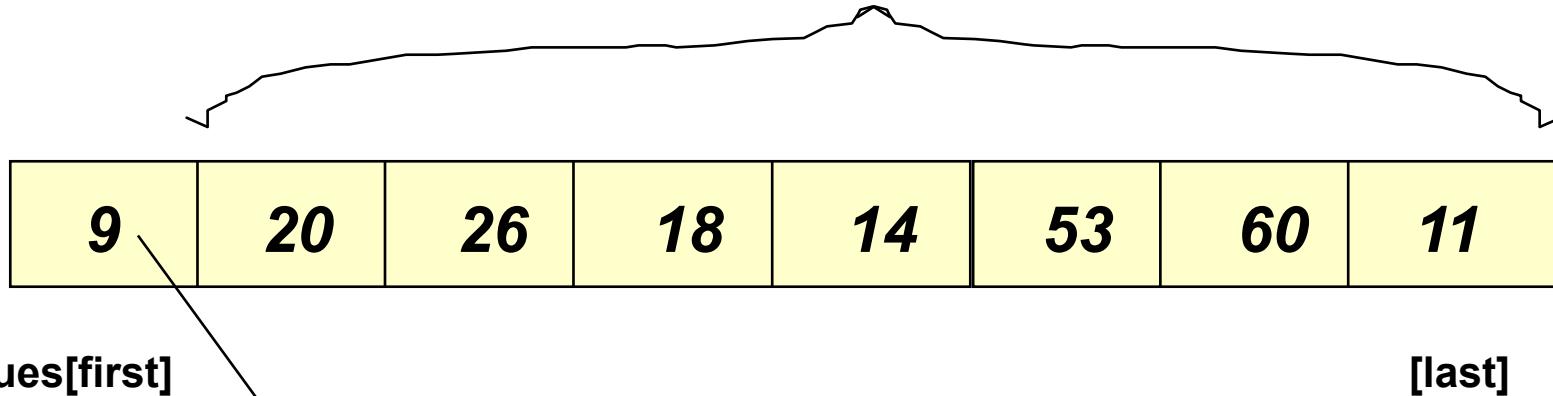


After call to function Split

splitVal = 9

**no smaller values
empty left part**

**larger values
in right part with N-1 elements**



splitVal in correct position



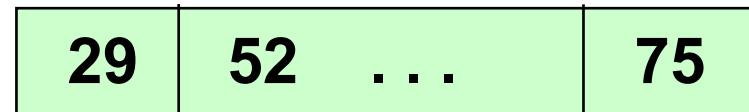
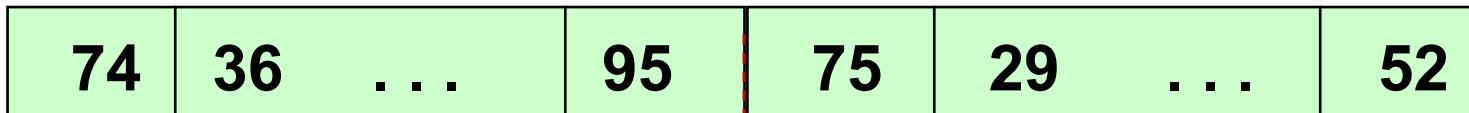
Merge Sort Algorithm

Cut the array in half.

Sort the left half.

Sort the right half.

Merge the two sorted halves into one sorted array.





```
// Recursive merge sort algorithm

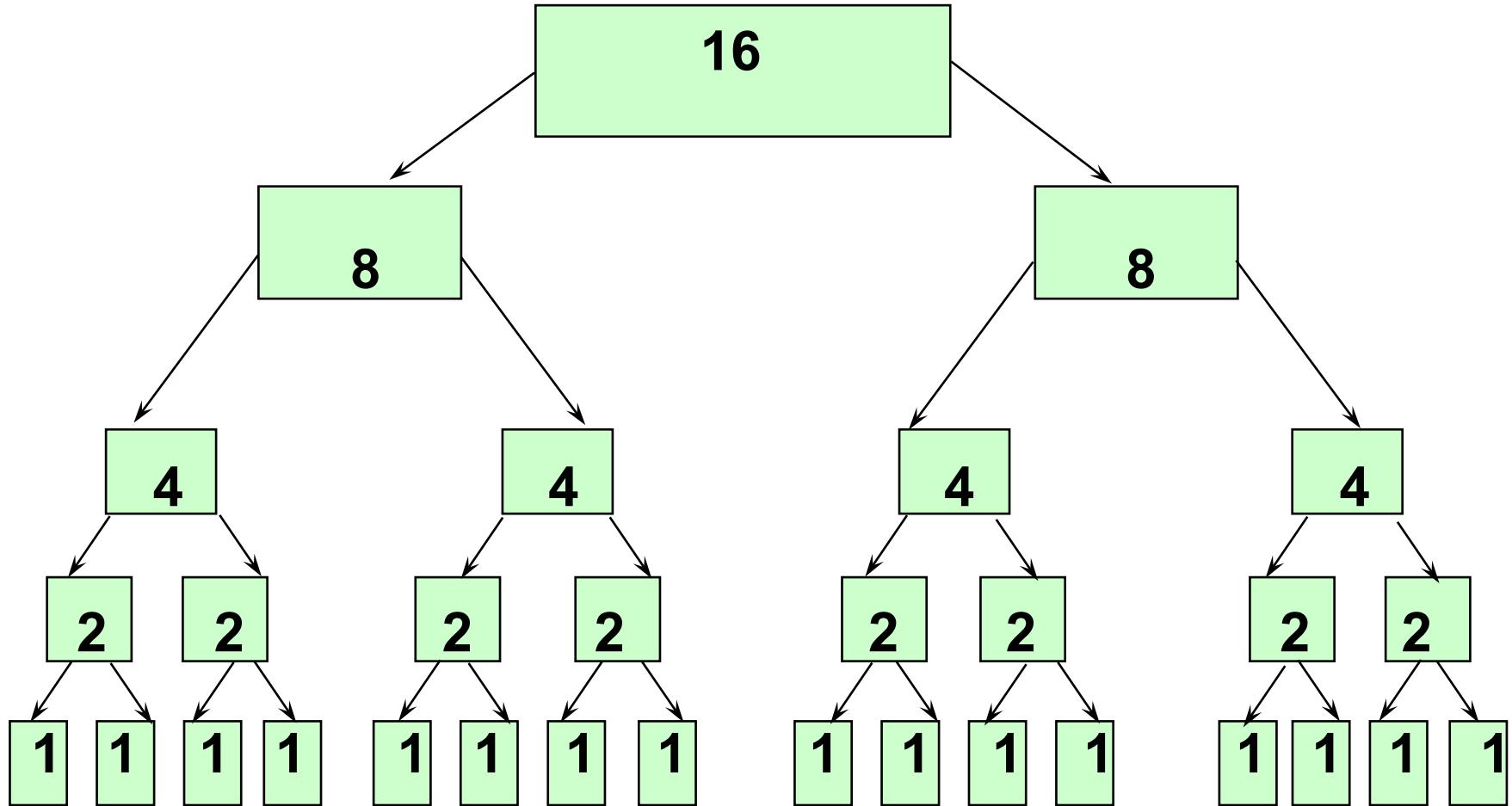
template <class ItemType>
void MergeSort ( ItemType values[ ] , int first ,
    int last )
// Pre: first <= last
// Post: Array values[first..last] sorted into
// ascending order.
{
    if ( first < last )                                // general case
    {
        int middle = ( first + last ) / 2 ;
        MergeSort ( values, first, middle ) ;
        MergeSort( values, middle + 1, last ) ;

        // now merge two subarrays
        // values [ first . . . middle ] with
        // values [ middle + 1, . . . last ].

        Merge(values, first, middle, middle + 1, last) ;
    }
}
```



Using Merge Sort Algorithm with N = 16





Merge Sort of N elements: How many comparisons?

The entire array can be subdivided into halves only $\log_2 N$ times.

Each time it is subdivided, function Merge is called to re-combine the halves. Function Merge uses a temporary array to store the merged elements. Merging is $O(N)$ because it compares each element in the subarrays.

Copying elements back from the temporary array to the values array is also $O(N)$.

MERGE SORT IS $O(N * \log_2 N)$.



Comparison of Sorting Algorithms

Sort	Order of Magnitude		
	Best Case	Average Case	Worst Case
selectionSort	$O(N^2)$	$O(N^2)$	$O(N^2)$
bubbleSort	$O(N^2)$	$O(N^2)$	$O(N^2)$
shortBubble	$O(N)$ (*)	$O(N^2)$	$O(N^2)$
insertionSort	$O(N)$ (*)	$O(N^2)$	$O(N^2)$
mergeSort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$
quickSort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$ (depends on split)
heapSort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$

*Data almost sorted.



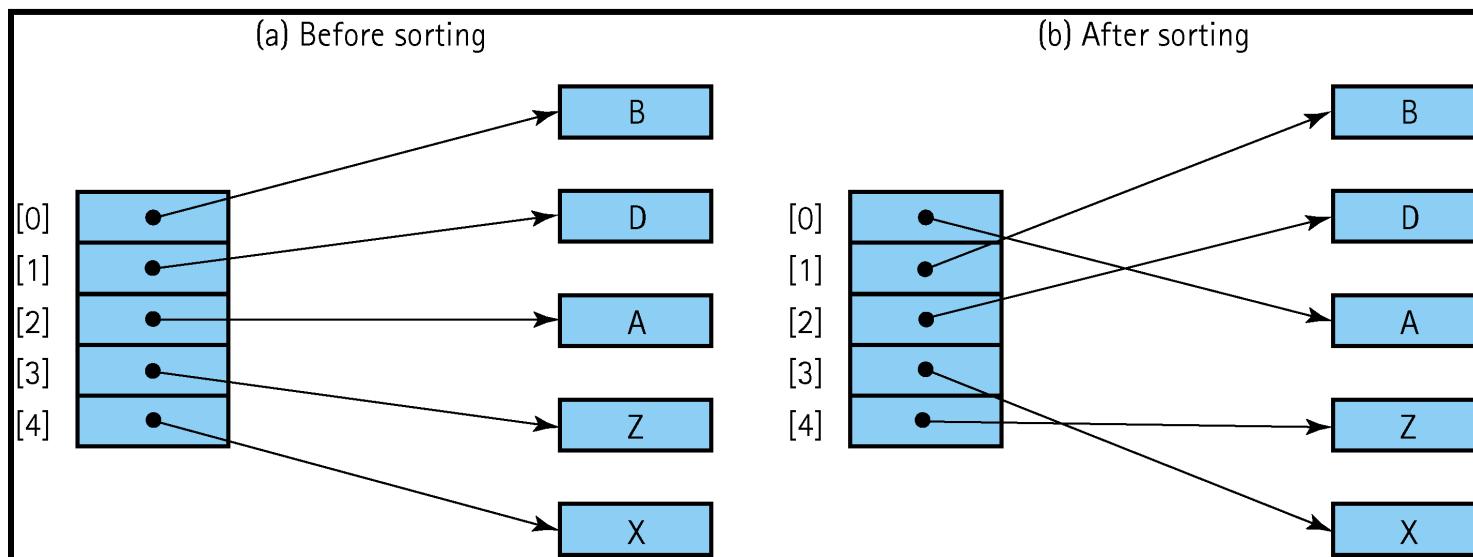
Testing

- To thoroughly test our sorting methods we should vary the size of the array they are sorting
- Vary the original order of the array-test
 - Reverse order
 - Almost sorted
 - All identical elements



Sorting Objects

- When sorting an array of objects we are manipulating references to the object, and not the objects themselves





Stability

- Stable Sort: A sorting algorithm that preserves the order of duplicates
- Of the sorts that we have discussed in this book, only `heapSort` and `quickSort` are inherently unstable



Searching

- Linear (or Sequential) Searching
 - Beginning with the first element in the list, we search for the desired element by examining each subsequent item's key
- High-Probability Ordering
 - Put the most-often-desired elements at the beginning of the list
 - *Self-organizing* or *self-adjusting* lists
- Key Ordering
 - Stop searching before the list is exhausted if the element does not exist



Function BinarySearch()

- **BinarySearch takes sorted array info, and two subscripts, fromLoc and toLoc, and item as arguments. It returns false if item is not found in the elements info[fromLoc...toLoc]. Otherwise, it returns true.**
- **BinarySearch is O(log₂N).**



```
found = BinarySearch(info, 25, 0, 14);
```

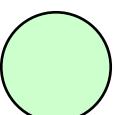
item fromLoc toLoc

indexes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----

info

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28
								16	18	20	22	24	26	28
											22	24	26	28
												24	26	28

NOTE:  denotes element examined



```
template<class ItemType>
bool BinarySearch(ItemType info[ ], ItemType item,
                  int fromLoc , int toLoc )
    // Pre: info [ fromLoc . . toLoc ] sorted in ascending order
    // Post: Function value = ( item in info[fromLoc .. toLoc])
{
    int mid ;
    if ( fromLoc > toLoc ) // base case -- not found
        return false ;
    else
    {
        mid = ( fromLoc + toLoc ) / 2 ;
        if ( info[mid] == item )           // base case-- found at mid
            return true ;
        else
            if ( item < info[mid])      // search lower half
                return BinarySearch( info, item, fromLoc, mid-1 ) ;
            else                         // search upper half
                return BinarySearch( info, item, mid + 1, toLoc ) ;
    }
}
```



Hashing

- is a means used to order and access elements in a list quickly -- the goal is $O(1)$ time -- by using a function of the key value to identify its location in the list.
- The function of the key value is called a hash function.

FOR EXAMPLE . . .



Using a hash function

values
[0]
Empty
[1]
4501
[2]
Empty
[3]
7803
[4]
Empty
:
:
:
[97]
Empty
[98]
2298
[99]
3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100



Placing Elements in the Array

values
[0]
Empty
[1]
4501
[2]
Empty
[3]
7803
[4]
Empty
.
.
.
[97]
Empty
[98]
2298
[99]
3699

Use the hash function

Hash(key) = partNum % 100

**to place the element with
part number 5502 in the
array.**



Placing Elements in the Array

values	
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Next place part number
6702 in the array.

Hash(key) = partNum % 100

$$6702 \% 100 = 2$$

But values[2] is already
occupied.

COLLISION OCCURS

the condition resulting when two or more
keys produce the same hash location



How to Resolve the Collision?

values
[0]
Empty
[1]
4501
[2]
5502
[3]
7803
[4]
Empty
.
.
.
.
[97]
Empty
[98]
2298
[99]
3699

One way is by linear probing.
This uses the rehash function

$$(\text{HashCode} + 1) \% 100$$

repeatedly until an empty location
is found for part number 6702.



Resolving the Collision

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

$$(\text{HashCode} + 1) \% 100$$



Collision Resolved

values
[0]
Empty
[1]
4501
[2]
5502
[3]
7803
[4]
Empty
.
.
.
.
[97]
Empty
[98]
2298
[99]
3699

Part 6702 can be placed at the location with index 4.



Collision Resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	
[4]	7803
.	
.	
.	
[97]	
[98]	Empty
[99]	2298
	3699

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?



Deletion with Linear Probing

Order of Insertion:	[00]	Empty
14001	[01]	Element with key = 14001
00104	[02]	Empty
50003	[03]	Element with key = 50003
77003	[04]	Element with key = 00104
42504	[05]	Element with key = 77003
33099	[06]	Element with key = 42504
:	[07]	Empty
	[08]	Empty
	:	:
	[99]	Element with key = 33099

What happens if we perform

- first, delete the element with 77003**
- then, search for the element with 42504**



Deletion with Linear Probing

Order of Insertion:

14001

00104

50003

77003

42504

33099

:

[00]	Empty
[01]	Element with key = 14001
[02]	Empty
[03]	Element with key = 50003
[04]	Element with key = 00104
[05]	Element with key = 77003
[06]	Element with key = 42504
[07]	Empty
[08]	Empty
	⋮
[99]	Element with key = 33099

set this slot to
Deleted rather than
Empty

We cannot find the element with 42504 if we set the deleted slot to *Empty*



Resolving Collisions: Rehashing

- Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key
- Linear probing
 - $(\text{HashCode} + 1) \% \text{array-size}$
 - $(\text{HashCode} + \text{constant}) \% \text{array-size}$
- quadratic probing
 - $(\text{HashCode} \pm l^2) \% \text{array-size}$
- random probing
 - $(\text{HashCode} + \text{random-number}) \% \text{array-size}$

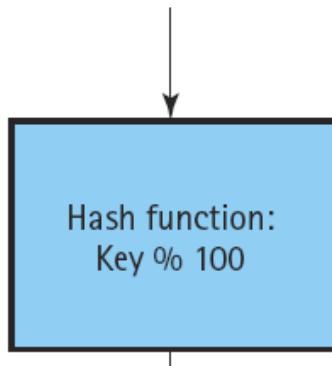


Resolving Collisions: Buckets and Chaining

- The main idea is to allow multiple element keys to hash to the same location
- ***Bucket*** A collection of elements associated with a particular hash location
- ***Chain*** A linked list of elements that share the same hash location

Resolving Collisions: Buckets

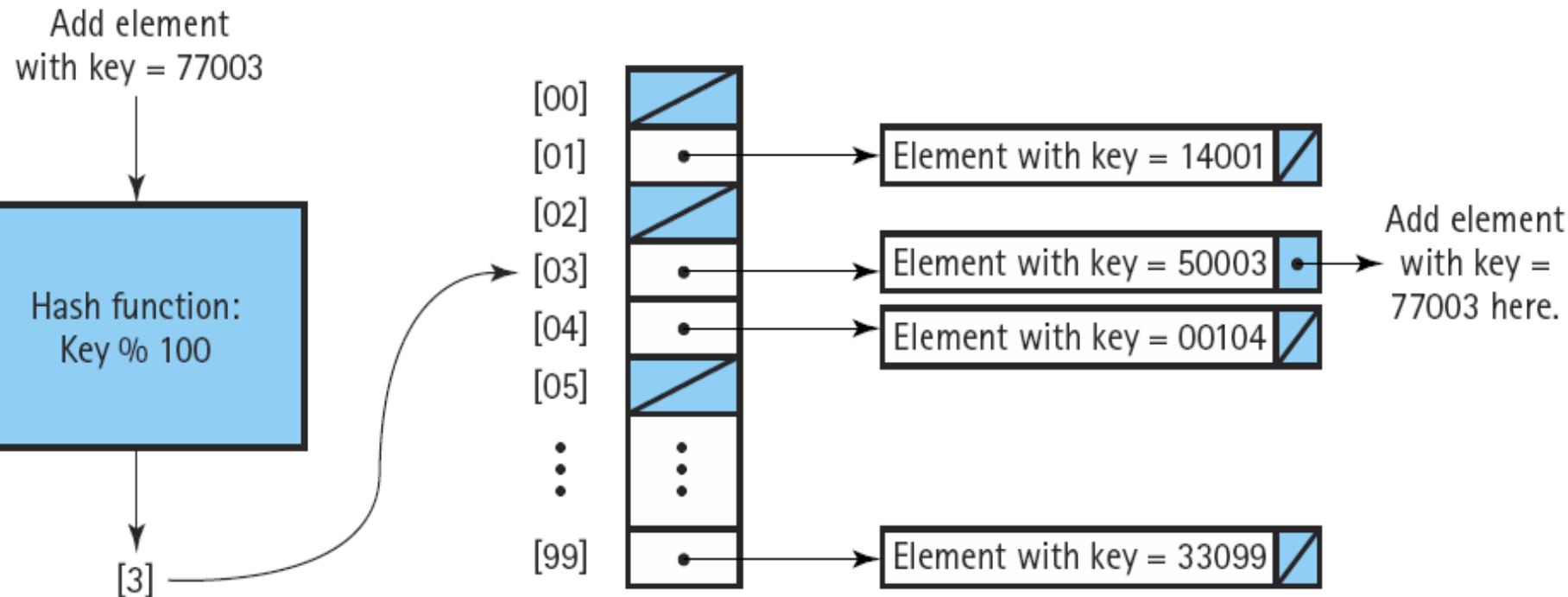
Add element
with key = 77003



[00]	Empty	Empty	Empty
[01]	Element with key = 14001	Element with key = 72101	Empty
[02]	Empty	Empty	Empty
[03]	Element with key = 50003	Add new element here	Empty
[04]	Element with key = 00104	Element with key = 30504	Element with key = 56004
[05]	Empty	Empty	Empty
:	:	:	:
[99]	Element with key = 56399	Element with key = 32199	Empty



Resolving Collisions: Chain





Choosing a Good Hash Functions

- Two ways to minimize collisions are
 - Increase the range of the hash function
 - Distribute elements as uniformly as possible throughout the hash table
- How to choose a good hash function
 - Utilize knowledge about statistical distribution of keys
 - Select appropriate hash functions
 - division method
 - sum of characters
 - folding
 - ...



Radix Sort

Radix sort

Is *not* a comparison sort

Uses a radix-length array of queues of records

Makes use of the values in digit positions in the keys to select the queue into which a record must be enqueued



Original Array

762
124
432
761
800
402
976
100
001
999



Queues After First Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800	761	762		124		976			999
100	001	432							
		402							



Array After First Pass

800
100
761
001
762
432
402
124
976
999



Queues After Second Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800		124	432			761	976		999
100						762			
001									
402									



Array After Second Pass

800
100
001
402
124
432
761
762
976
999



Queues After Third Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
001	100			402			761	800	976
	124			432			762		999



Array After Third Pass

001
100
124
402
432
761
762
800
976
999