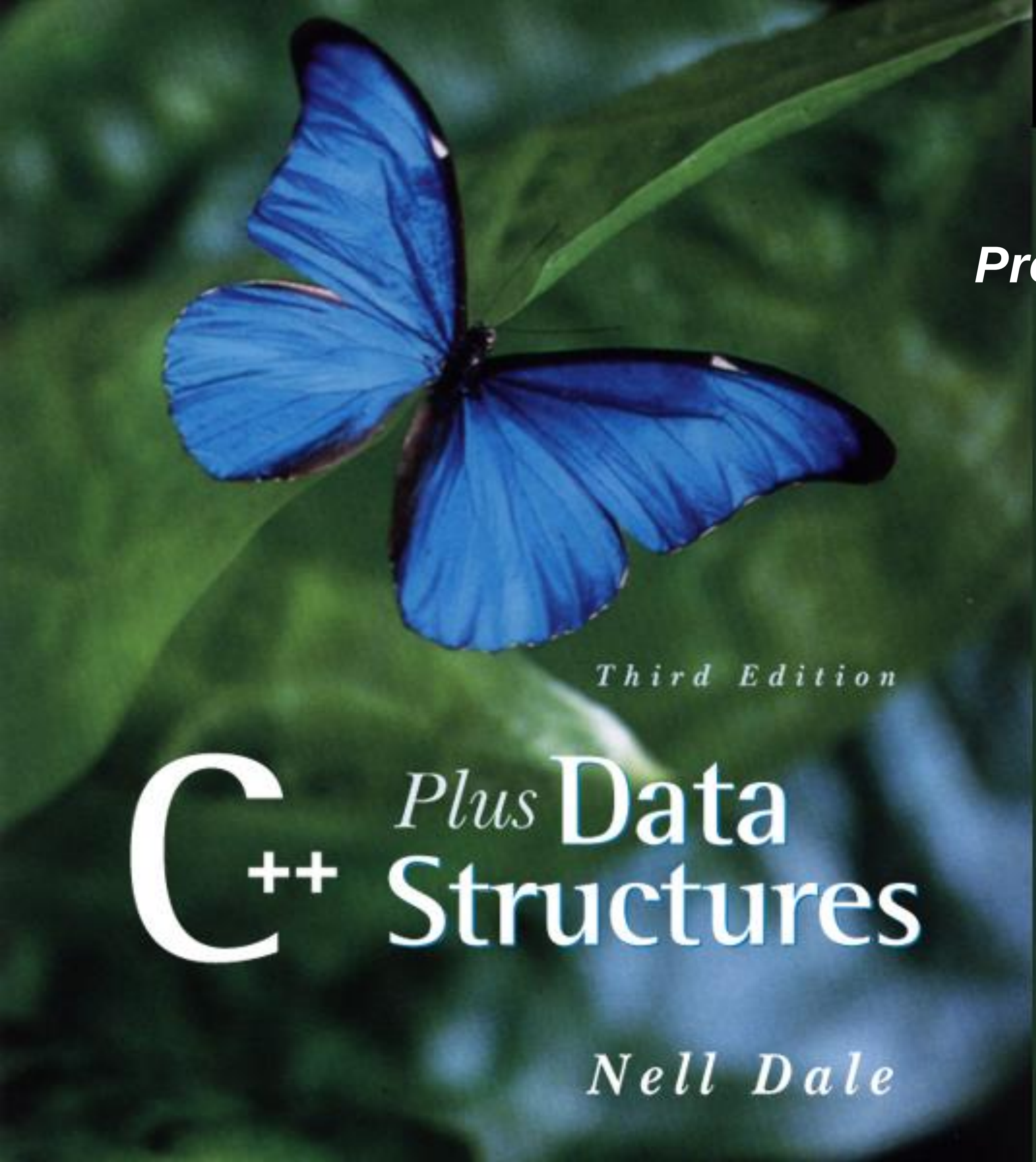


Chapter

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,

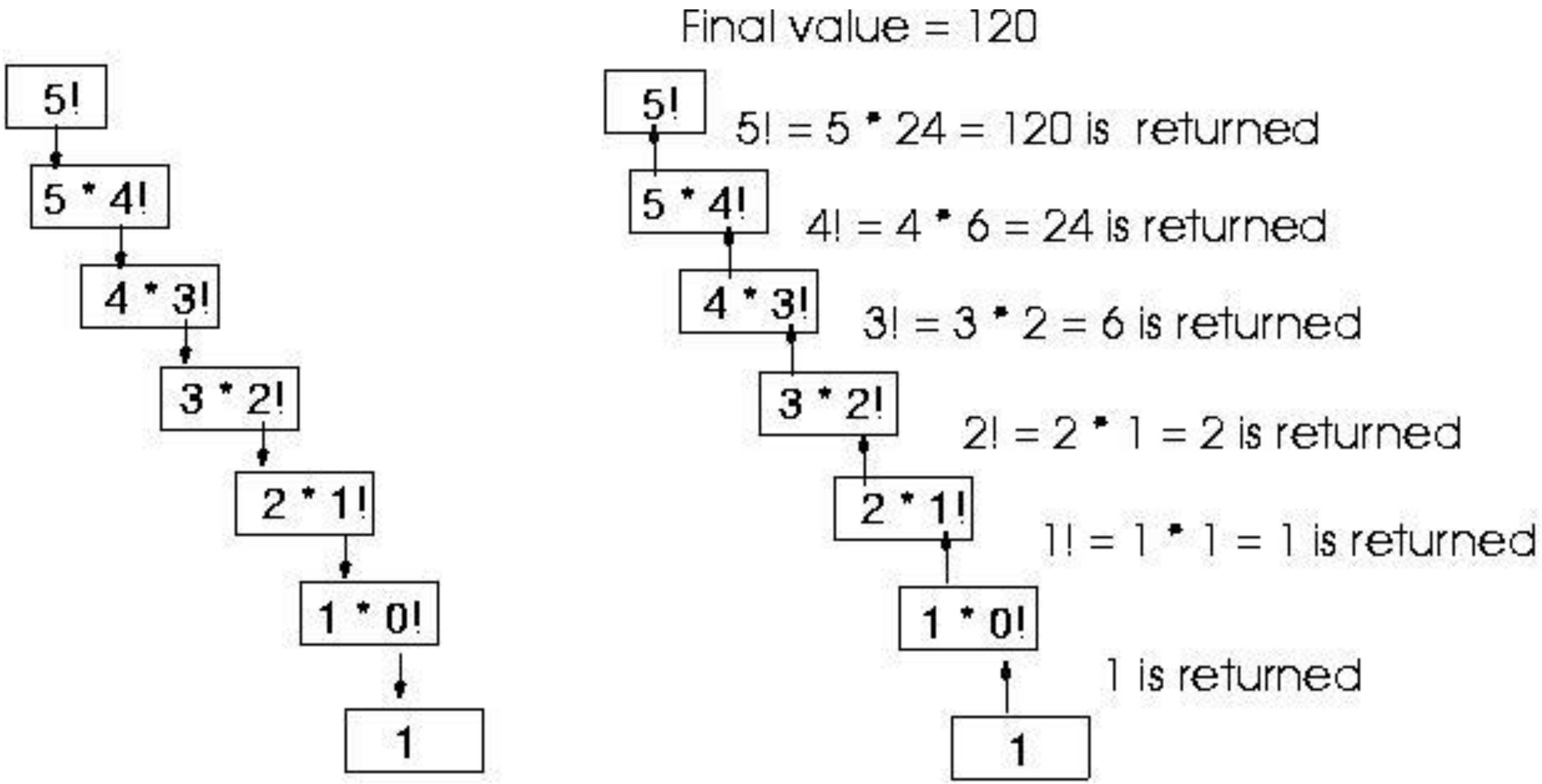
n * (n - 1) * . . . * 1

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
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5! = 5 * 24 = 120 is returned
4! = 4 * 6 = 24 is returned
3! = 3 * 2 = 6 is returned
2! = 2 * 1 = 2 is returned
1! = 1 * 1 = 1 is returned
1 is returned

Another example:
n choose *k* (combinations)

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

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

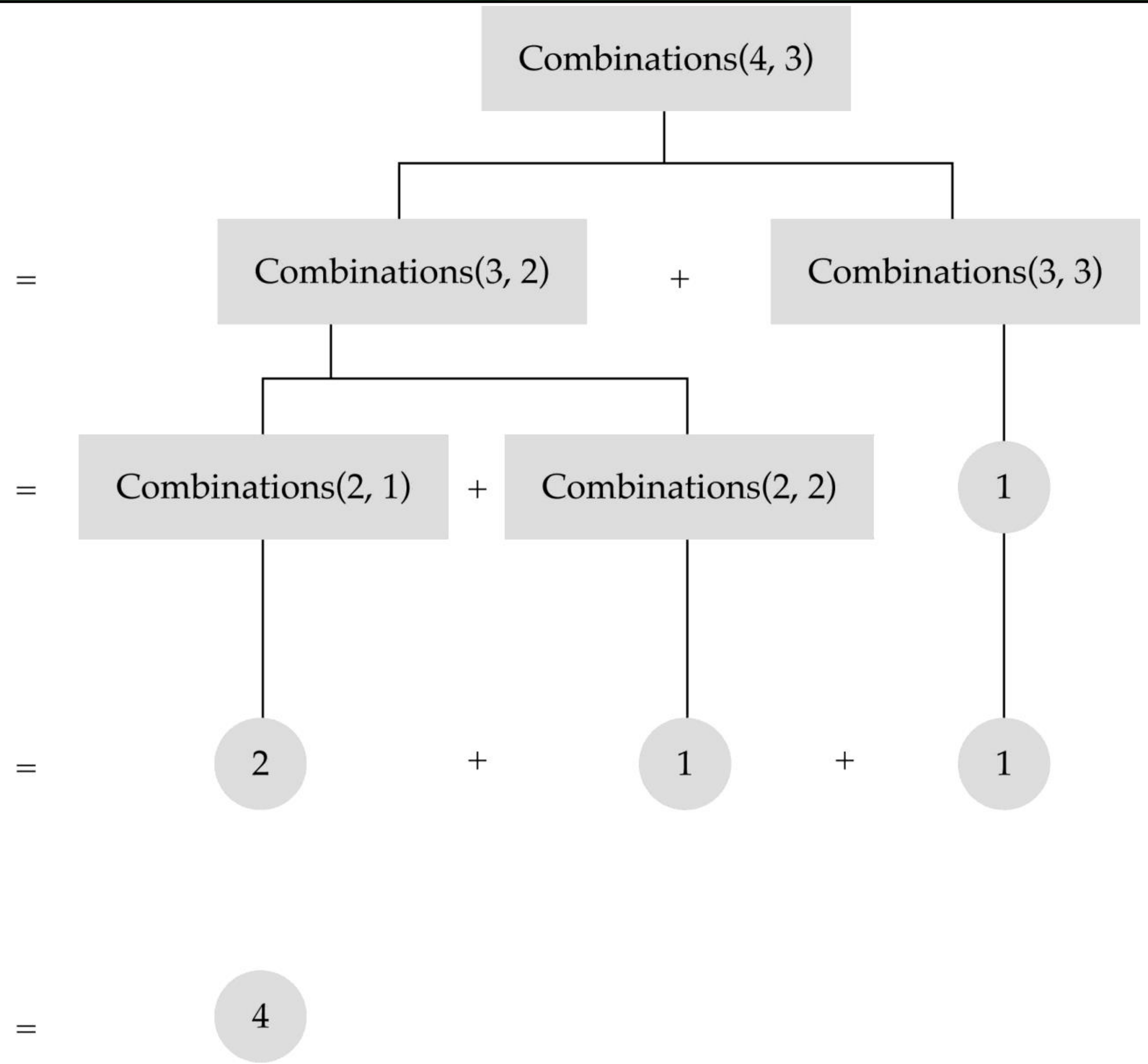
$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad , \ 1 < k < n \quad \textit{(closed-form solution)}$$

with base cases:

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

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));
}
```


Combinations(4, 3)
=
Combinations(3, 2)
+
Combinations(3, 3)
=
Combinations(2, 1)
+
Combinations(2, 2)
1
=
2
+
1
+
1
=
4

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

  $2^0 = 1$ and $2^5 = 2 * 2^4$

- In general,

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

  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) ;

74	36	. . .	95	75	29	47	. . .

list[0] [1] [startIndex] [length -1]

$\underbrace{\hspace{6em}}_{\text{Already searched}}$ index of current element to examine $\underbrace{\hspace{6em}}_{\text{Needs to be searched}}$

```
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?”

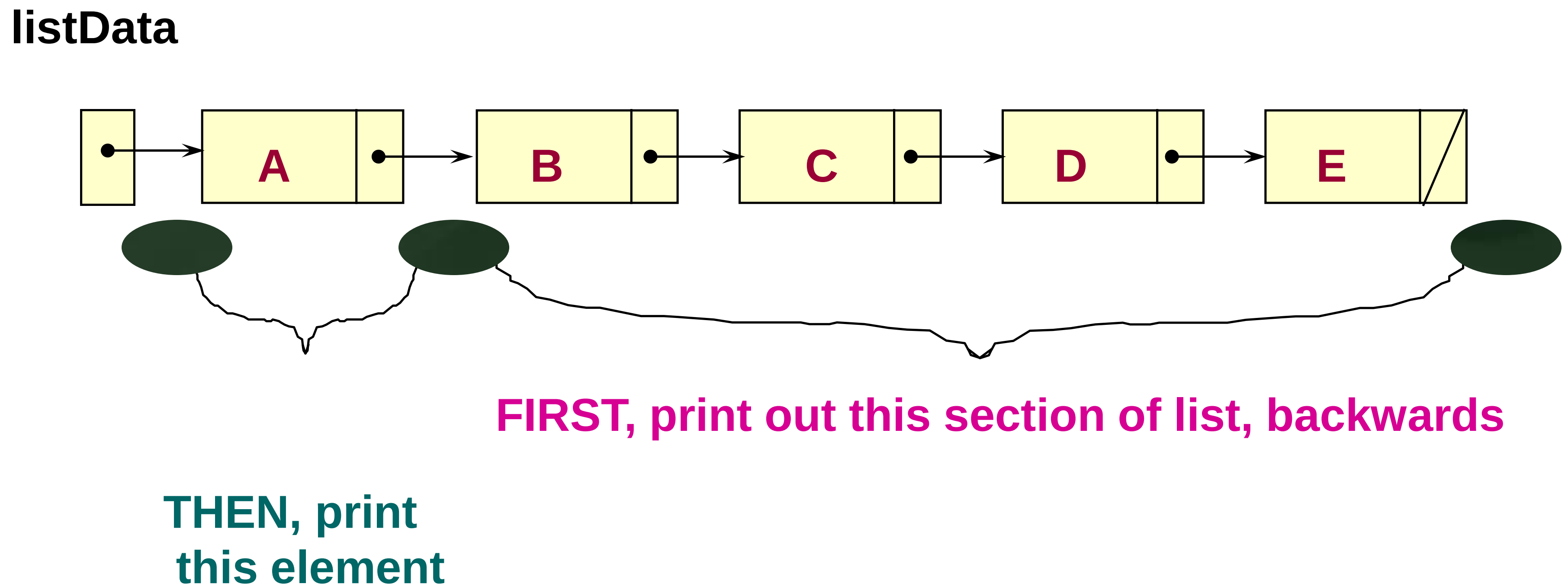
Those examples could have been written without recursion, using iteration instead. The iterative solution uses a loop, and the recursive solution uses an if statement.

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);



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]* <u>search the first half</u>

  if *item* > *info[midPoint]*, <u>search the second half</u>

```
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.

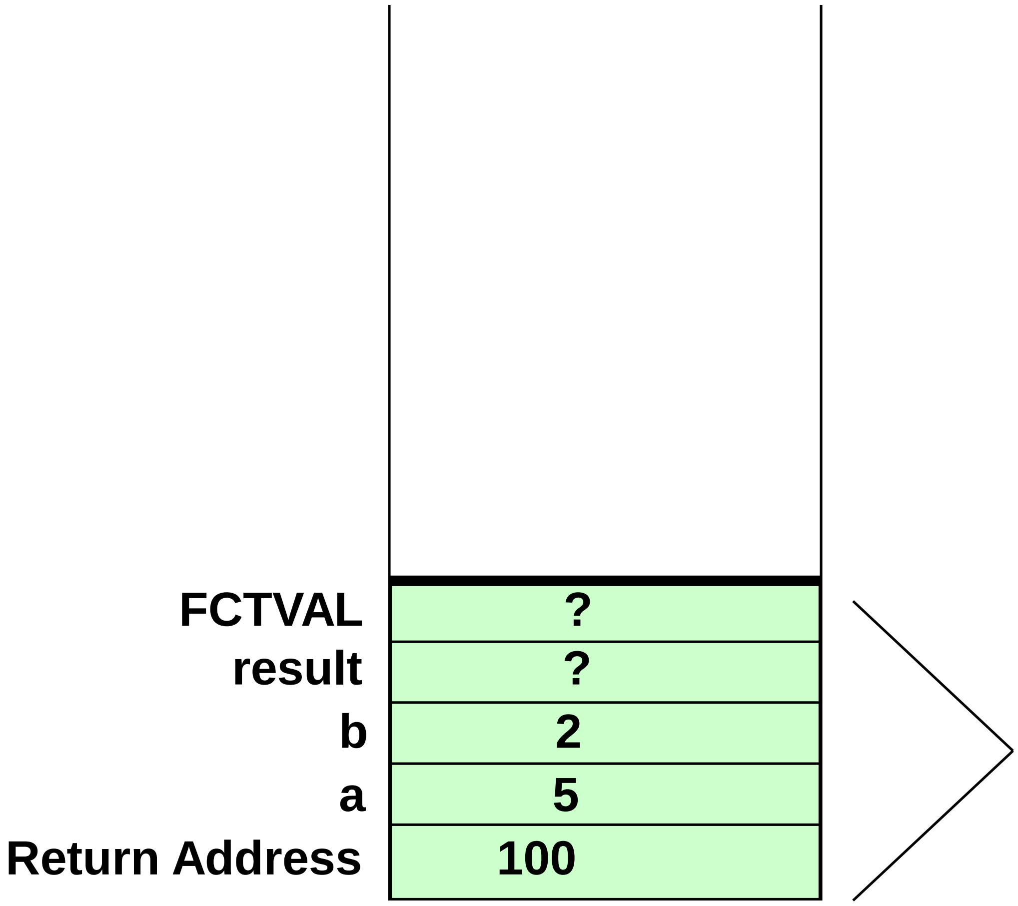
Stack Activation Frames

- 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**

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

call in Func(5,2) code at instruction 50 pushes on this 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	?
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

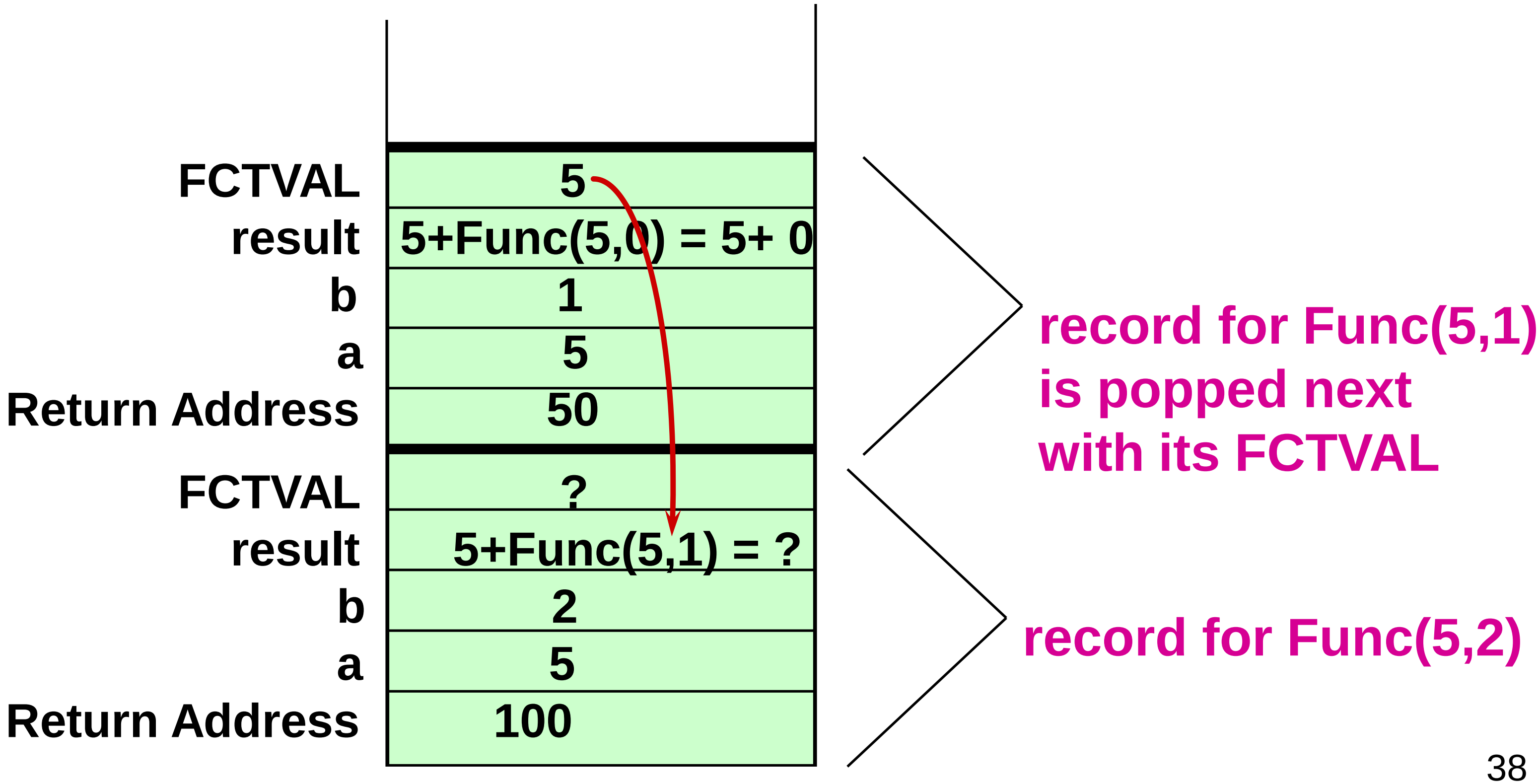
record for Func(5,0) is popped first with its FCTVAL

record for Func(5,1)

record for Func(5,2)

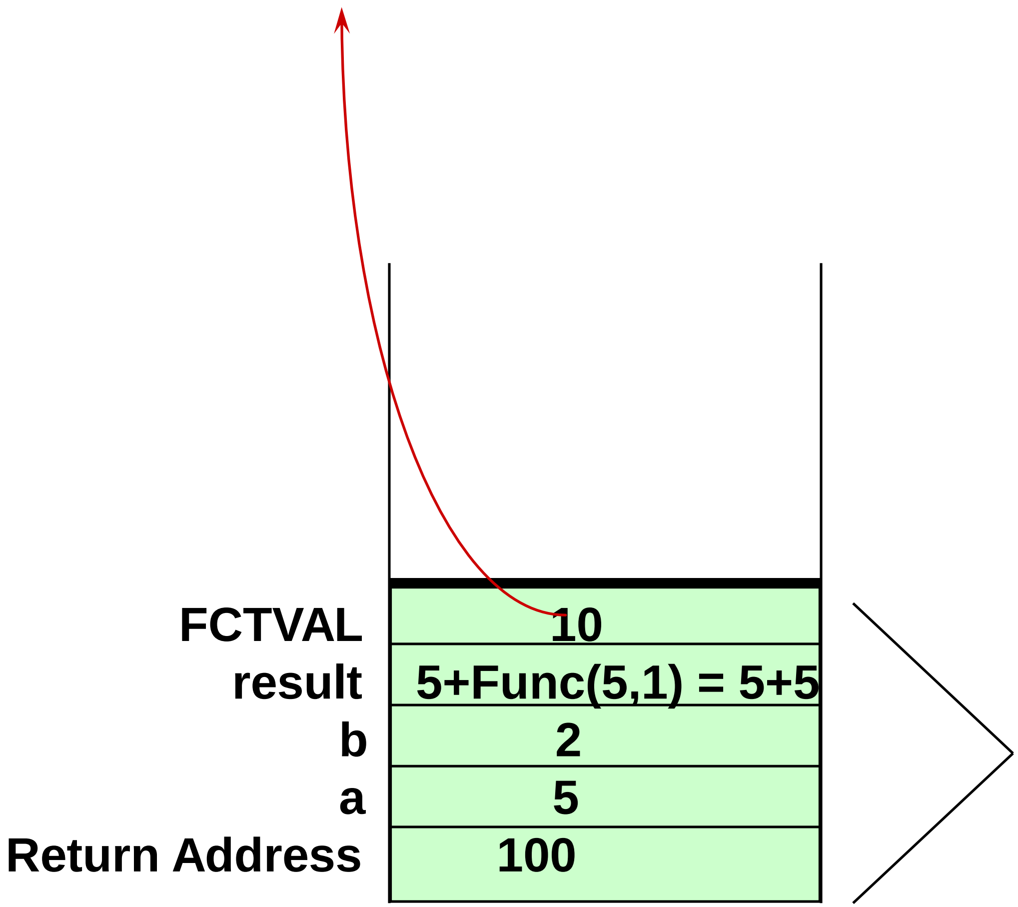
Run-Time Stack Activation Records

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



Run-Time Stack Activation Records

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



record for Func(5,2) is popped last with its FCTVAL

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*?

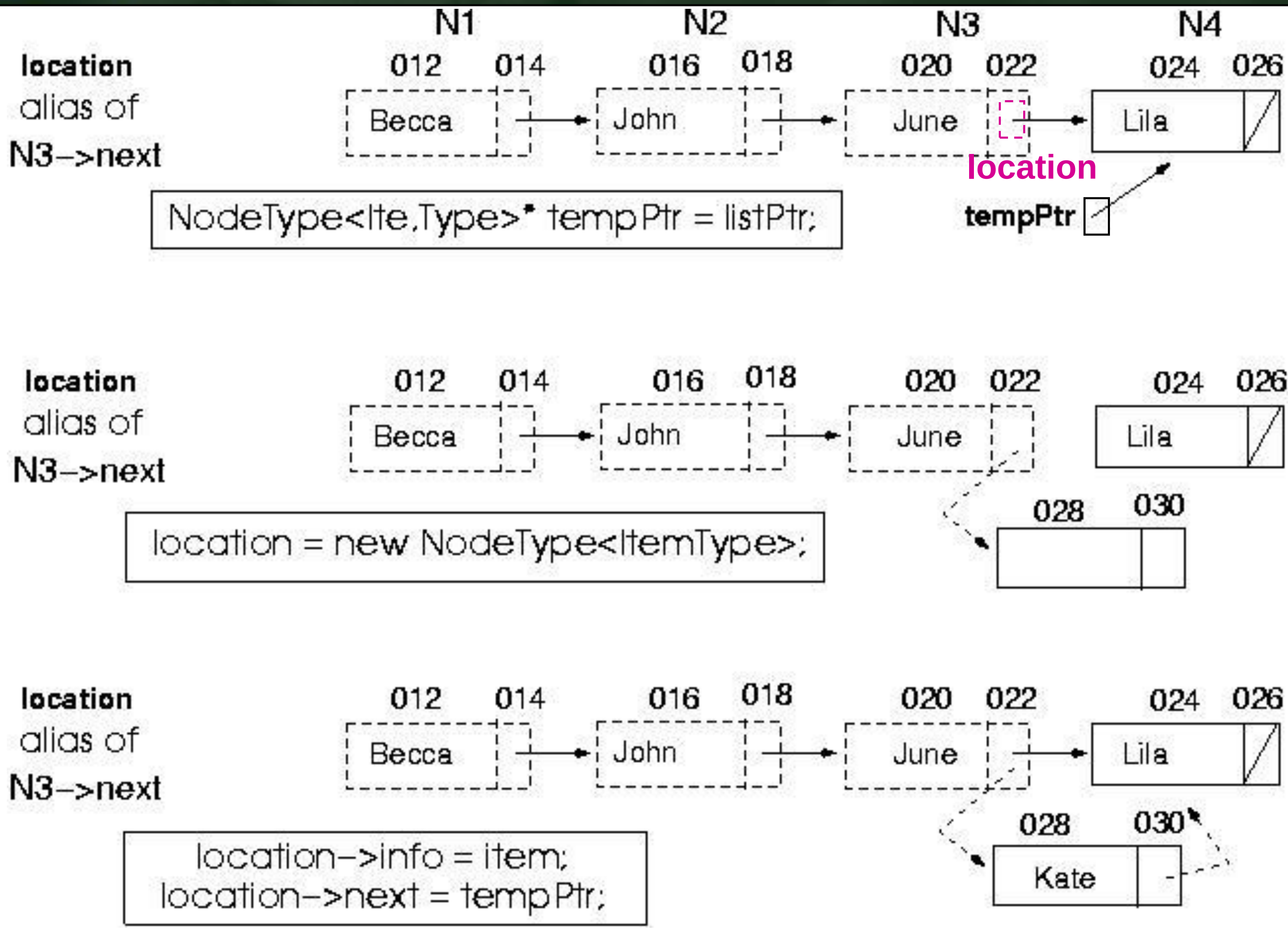
  Insert(location->next, item)

Recursive InsertItem (sorted list)

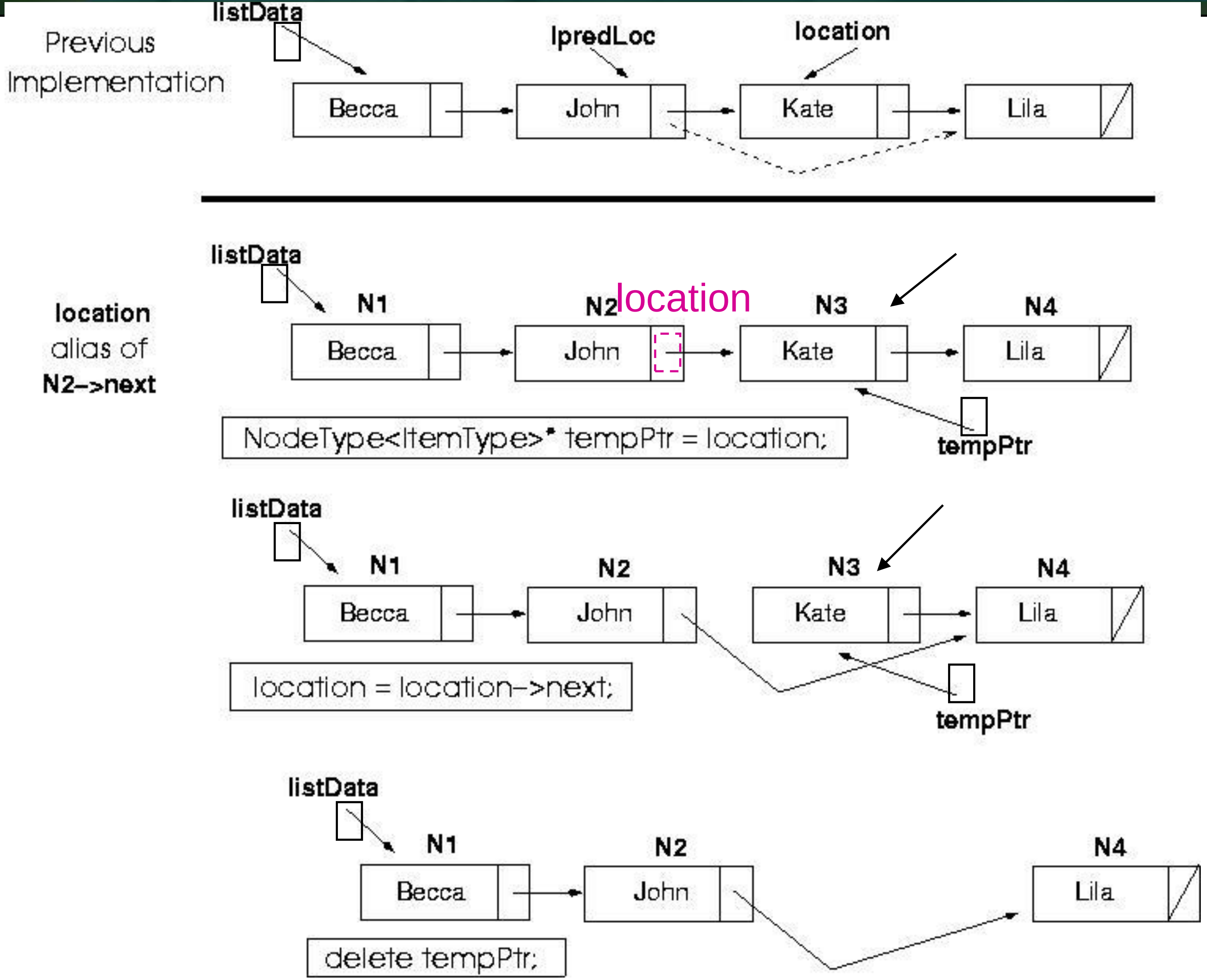
```
template <class ItemType>
void Insert(NodeType<ItemType>* &location, ItemType item)
{
 if(location == NULL) || (item < location->info)) {  // base cases

   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)



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                                           // general case
         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 */

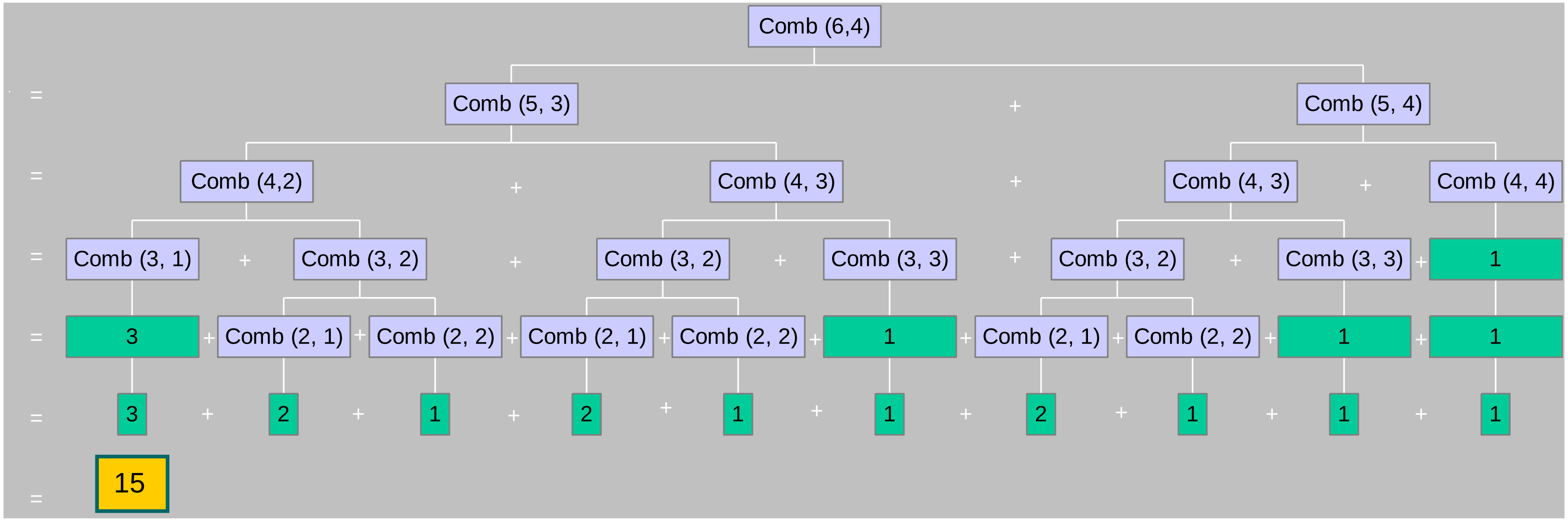
Stacking Technique: When it is the <u>not</u> last statement to be executed in a recursive function

```
// 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 is some cases



Use a recursive solution when:

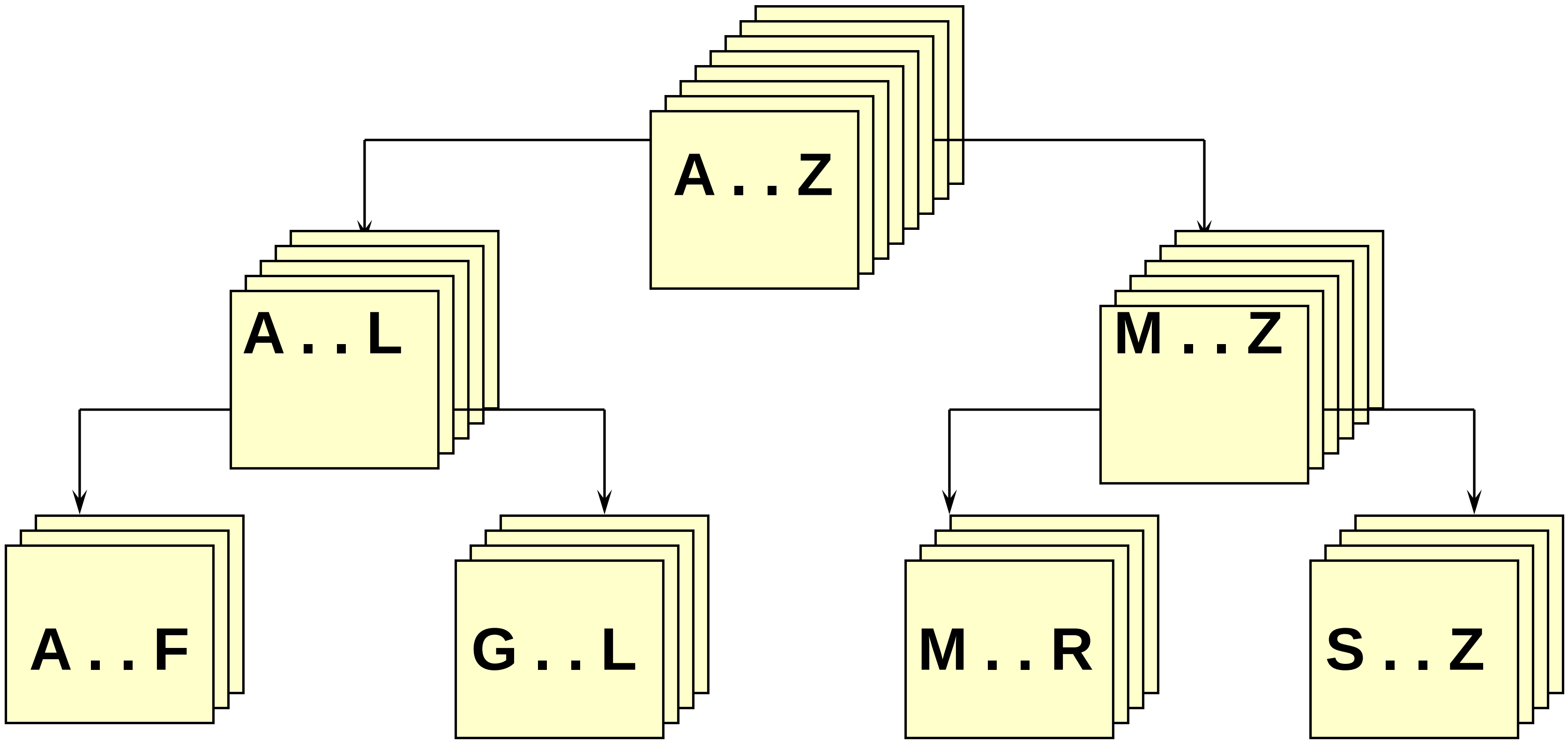
- The depth of recursive calls is relatively "shallow" compared to the size of the problem.
- The recursive version does about the same amount of work as the nonrecursive version.
- The recursive version is shorter and simpler than the nonrecursive solution.

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 );
   }

}
```