C++ Plus Data Structures

Nell Dale

David Teague

Chapter 7

Programming with Recursion

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,

$$n \;*\; \underbrace{(n - 1) \;*\; . \;.\; . \;*\; 1}$$

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 )  ;
}
```

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]	

Already searched

index of current element to examine

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?”

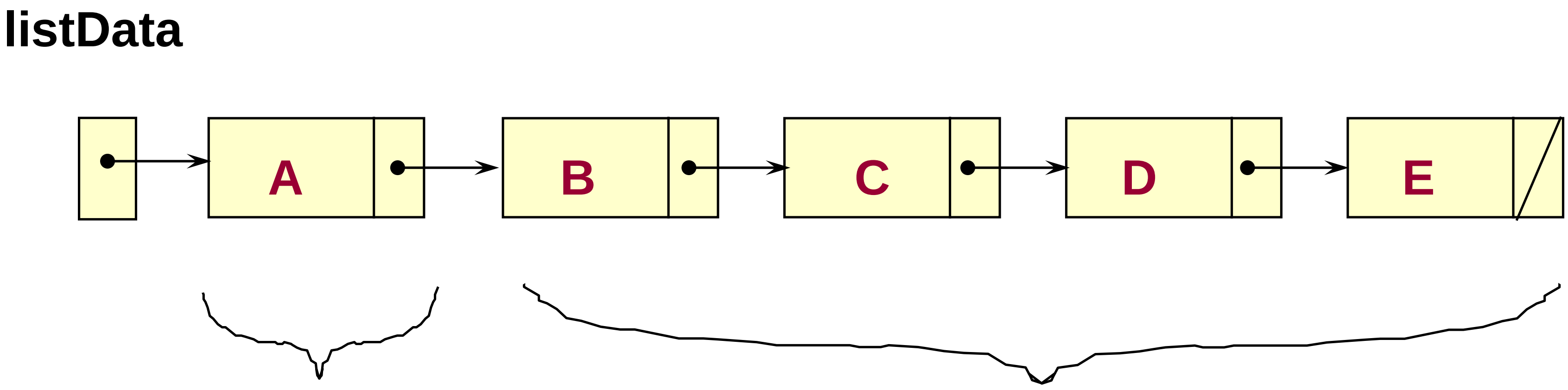
Those examples could have been written without recursion, using iteration instead. The iterative solution uses a loop, and the recursive solution uses an if statement.

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);



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   <<  endl ;     // then print  this elemen
    }
    // 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

```
// Recursive definition

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 ) ;
            }
}
```

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.

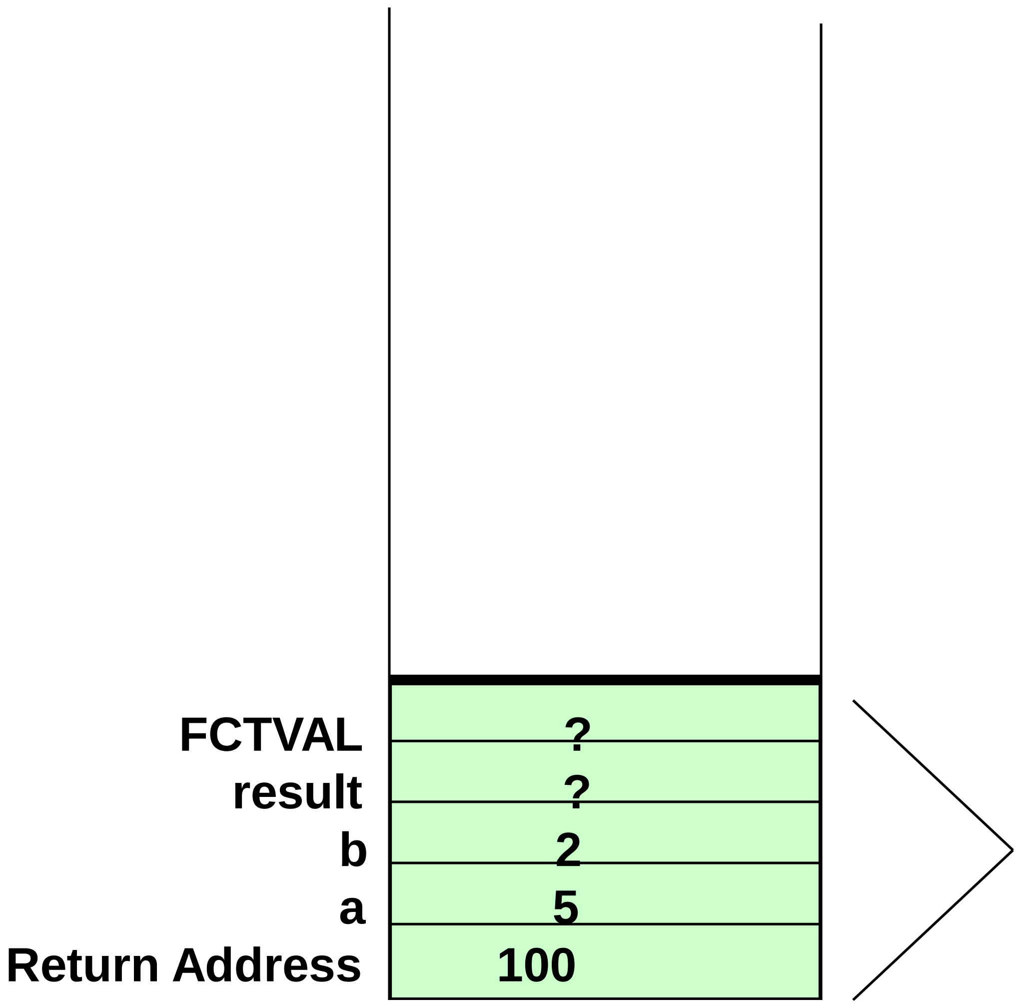
Stack Activation Frames

- 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, 1);` // 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)

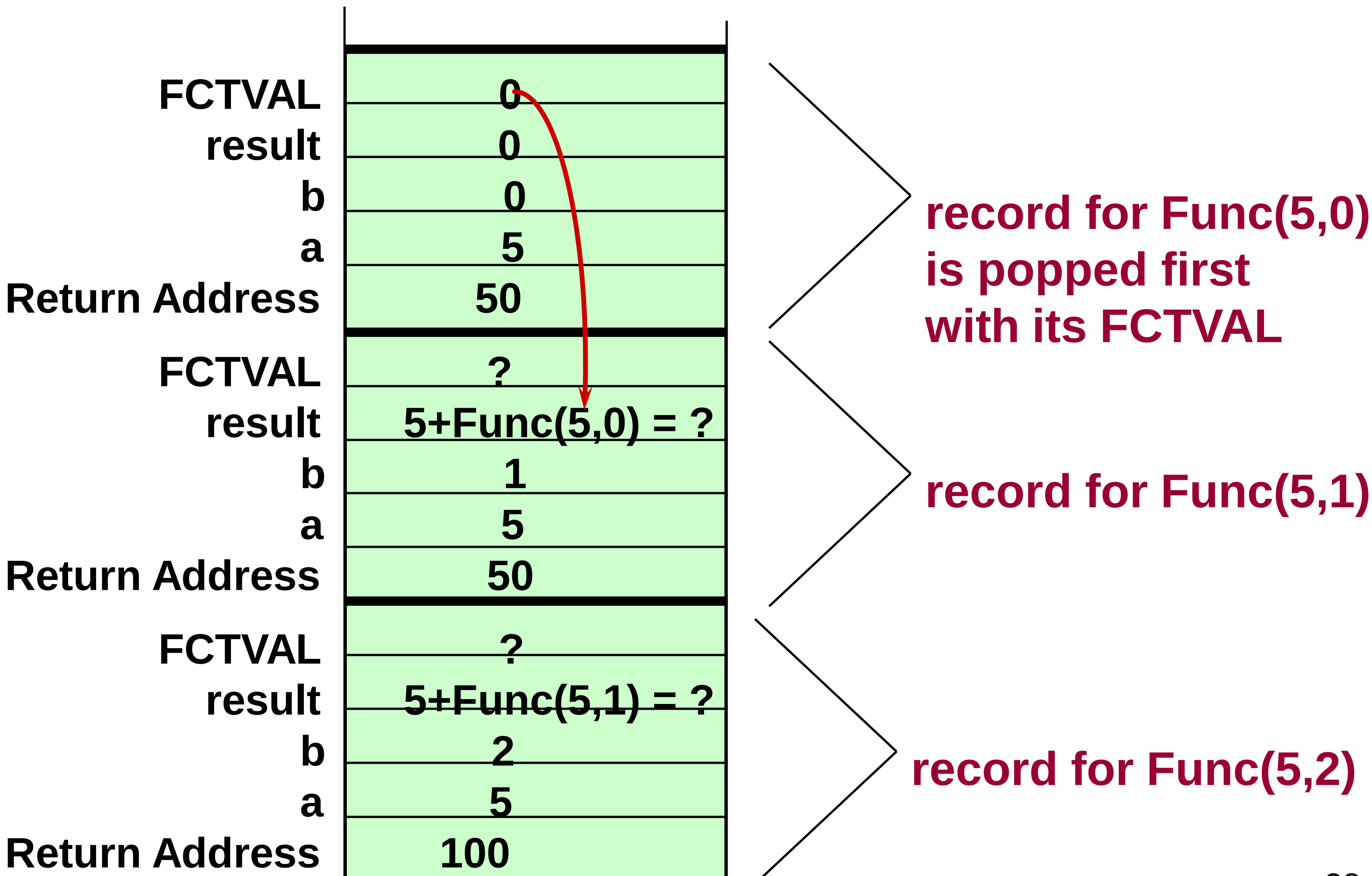
Run-Time Stack Activation Records

`x = Func(5, 0);` **// original call at instruction 100**

FCTVAL	**?**	**call in Func(5,1) code at instruction 50 pushes on this record for Func(5,0)**
result	**?**	
b	**0**	
a	**5**	
Return Address	**50**	
FCTVAL	**?**	**record for Func(5,1)**
result	**5+Func(5,0) = ?**	
b	**1**	
a	**5**	
Return Address	**50**	
FCTVAL	**?**	**record for Func(5,2)**
result	**5+Func(5,1) = ?**	
b	**2**	
a	**5**	
Return Address	**100**	

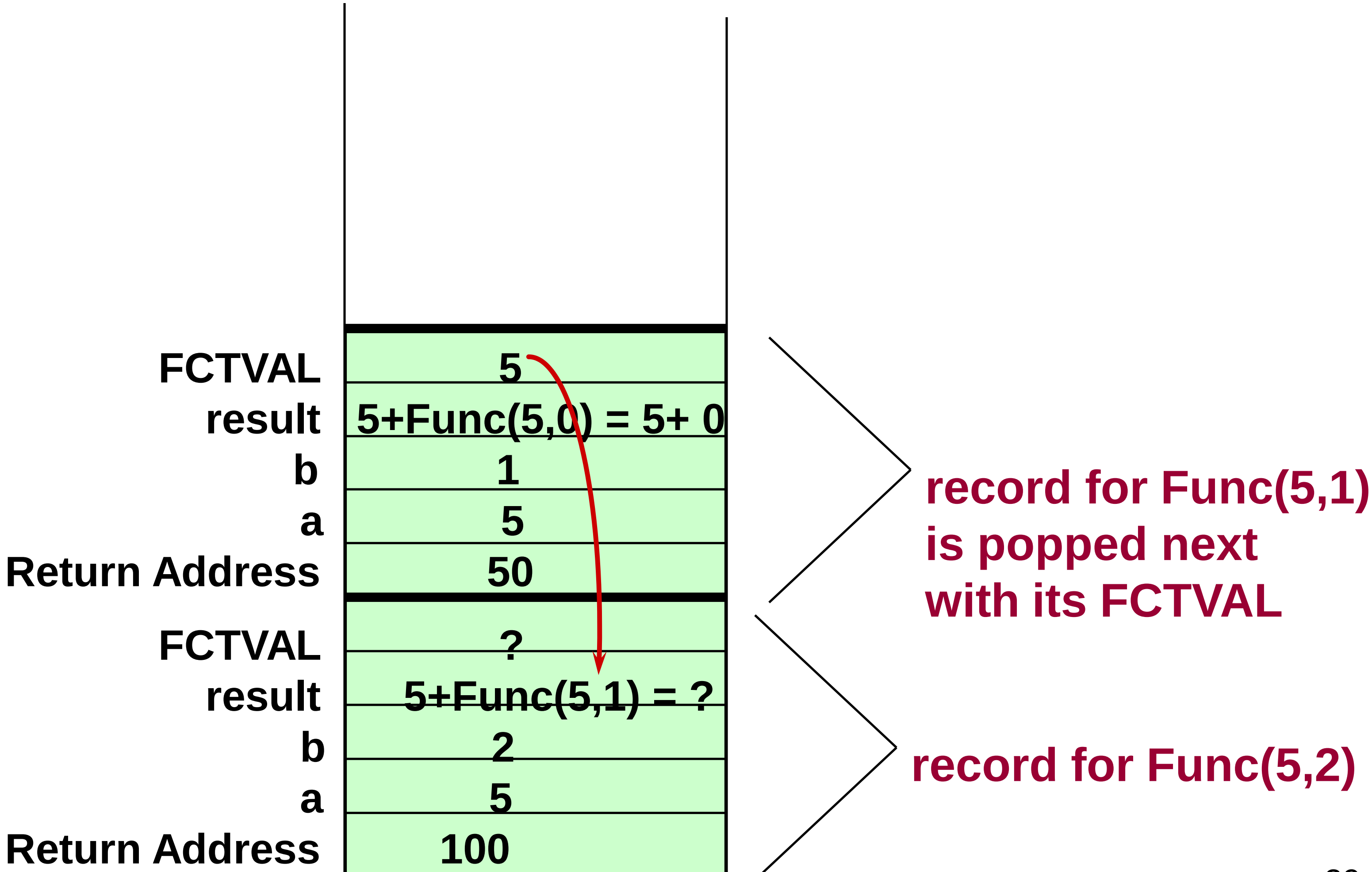
Run-Time Stack Activation Records

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



Run-Time Stack Activation Records

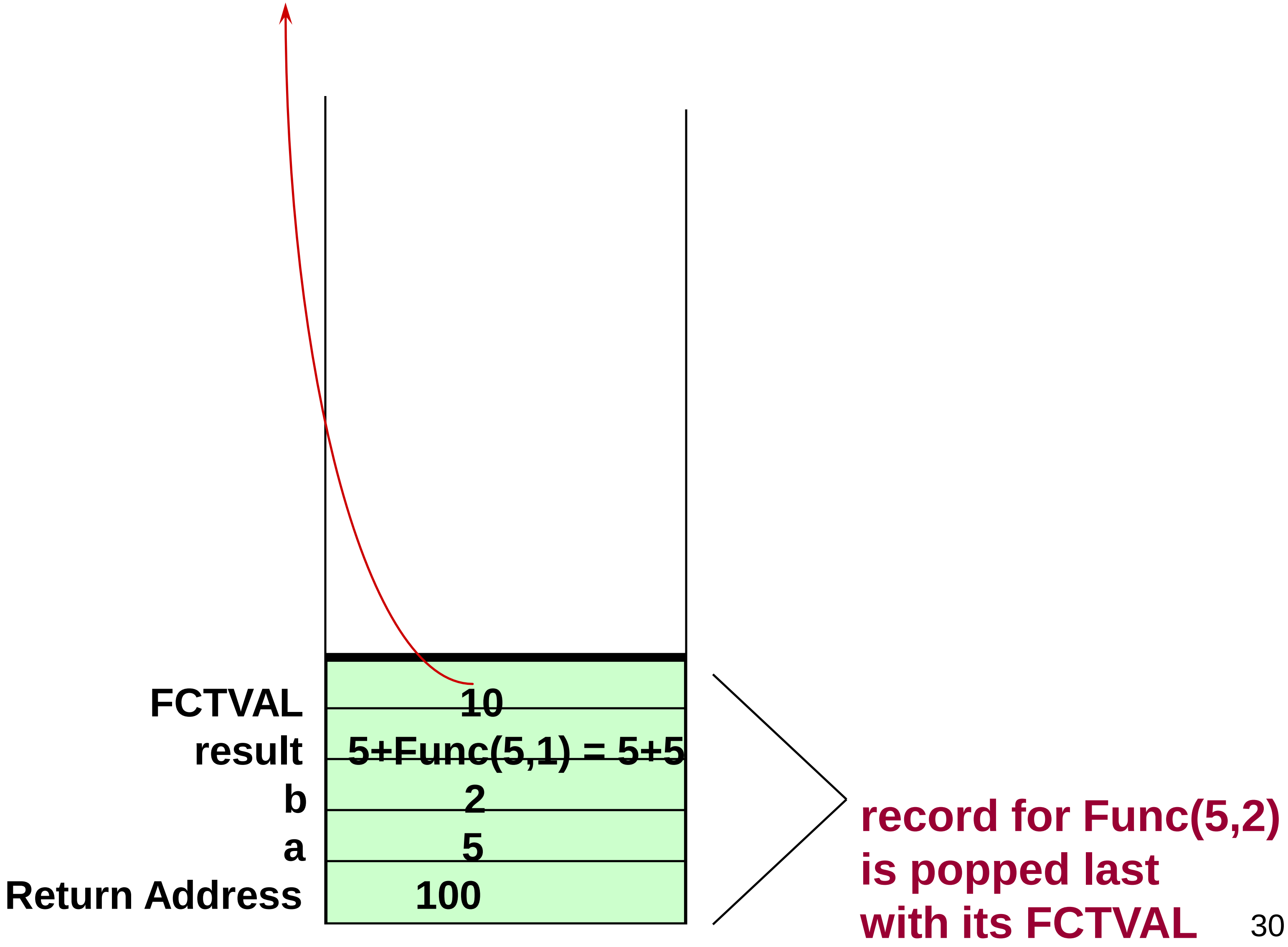
`x = Func(5, 2);` **// original call at instruction 100**



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?

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 ) ;
}
```

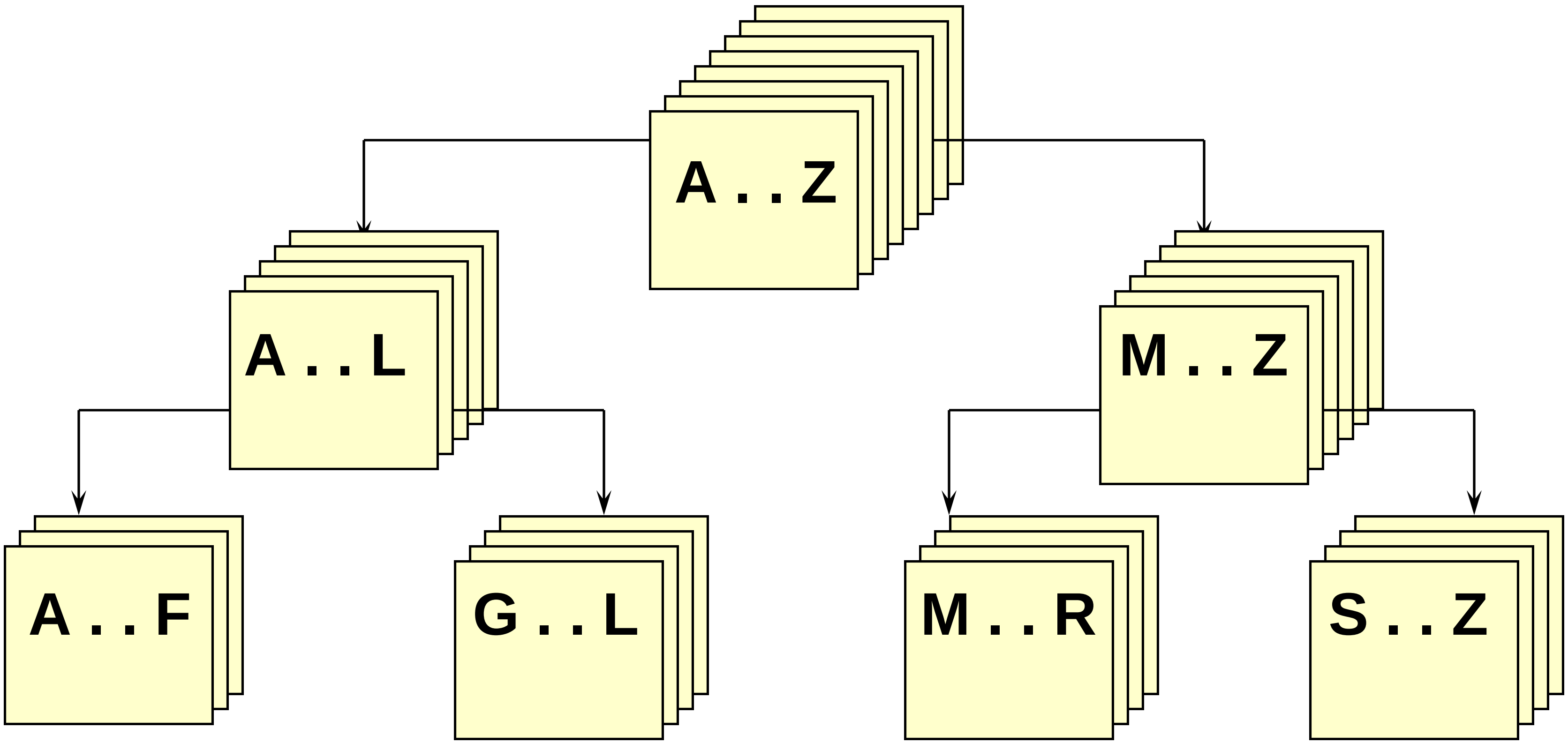
```
// ITERATIVE SOLUTION

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 ] )
{   bool   found  =  false ;
    while  ( !found  &&  startIndex < list.length )
    {      if ( value == list.info[ startIndex ] )
                   found = true ;
           else    startIndex++ ;
    }
    return  found ;
}
```

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]**

After call to function Split

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

Split Algorithm

1. Select the first element as a split value
2. Set first=1 and last=length-1
3. Move first forward until value[first]>splitVal
4. Move last backward until value[last]<splitVal
5. Swap value[first] and value[last]
6. first++, last—
7. If (first>last)

    Swap split value(value[saveF] and value[last]

    break
8. Otherwise repeat from Step 3

Split function

(a) Initialization. Note that splitVal = values[first] = 9.

9	20	6	10	14	8	60	11
[saveF]	[first]						[last]

(b) Increment first until values[first]>splitVal

9	20	6	10	14	8	60	11
[saveF]	[first]						[last]

(c) Decrement last until values[last]<= splitVal

9	20	6	10	14	8	60	11
[saveF]	[first]				[last]		

(d) Swap values [first] and values[last]; move first and last toward each other

9	8	6	10	14	20	60	11
[saveF]		[first]		[last]			

(e) Increment first until values[first]>splitVal or first>last.
Decrement last until values[last]<= splitVal or first>last

9	8	6	10	14	20	60	11
[saveF]		[last]	[first]				

(f) first>last so no swap occurs within the loop.
swap values[saveF] and values[last]

6	8	9	10	14	20	60	11
[saveF]		[last] (splitPoint)					

```
// 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 );
    }

}
```