dongguk UNIVERSITY

CSE 2017 Data Structures and Lab

Lecture #7: Recursion

Eun Man Choi

What is recursion?

- A technique that solves problem by solving smaller versions of the same problem!

- When you turn this into a program, you end up with functions that call themselves (i.e., recursive functions)

- Why use recursion?
  - Recursive algorithms can simplify the solution of a problem, often resulting in shorter, more easily understood source code.
  - But …they often less efficient, both in terms of *time* and *space*, than non-recursive (e.g., iterative) solutions

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 )  ;
}
```

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.

Writing Recursive Functions

- Get an exact definition of the problem to be solved.
- Determine the *size* of the problem on this call to the function.
- Identify and solve the *base case(s).*
- Identify and solve the *general case(s)* in terms of a smaller case of the same problem – a recursive call.

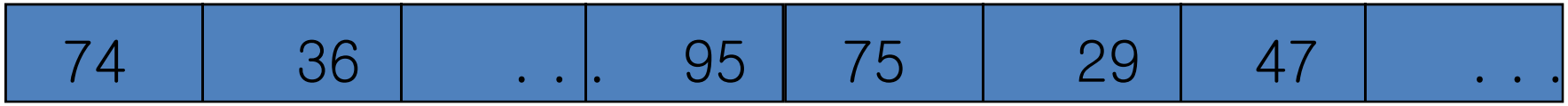
Another example: a boolean function

```
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 ) ;
```



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

Already searched

Needs to be searched

index of current element to examine

```
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.

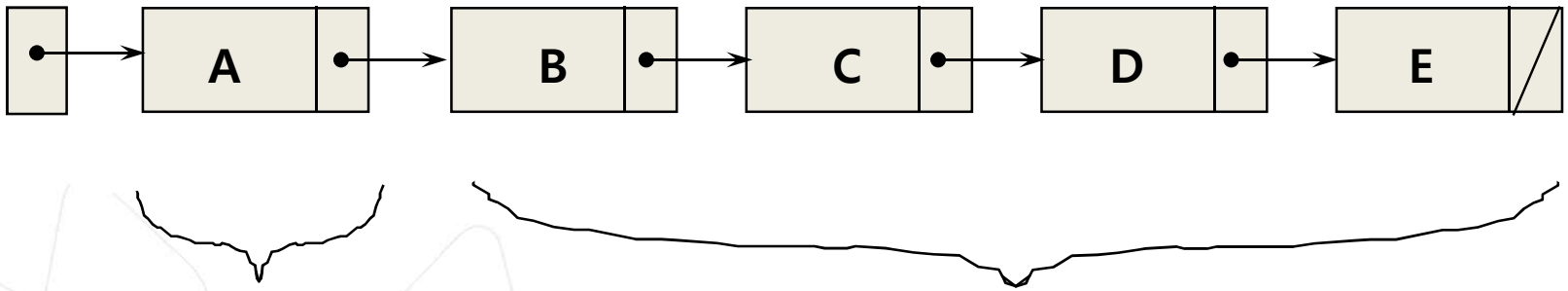
Recursive Linked List Processing

- 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

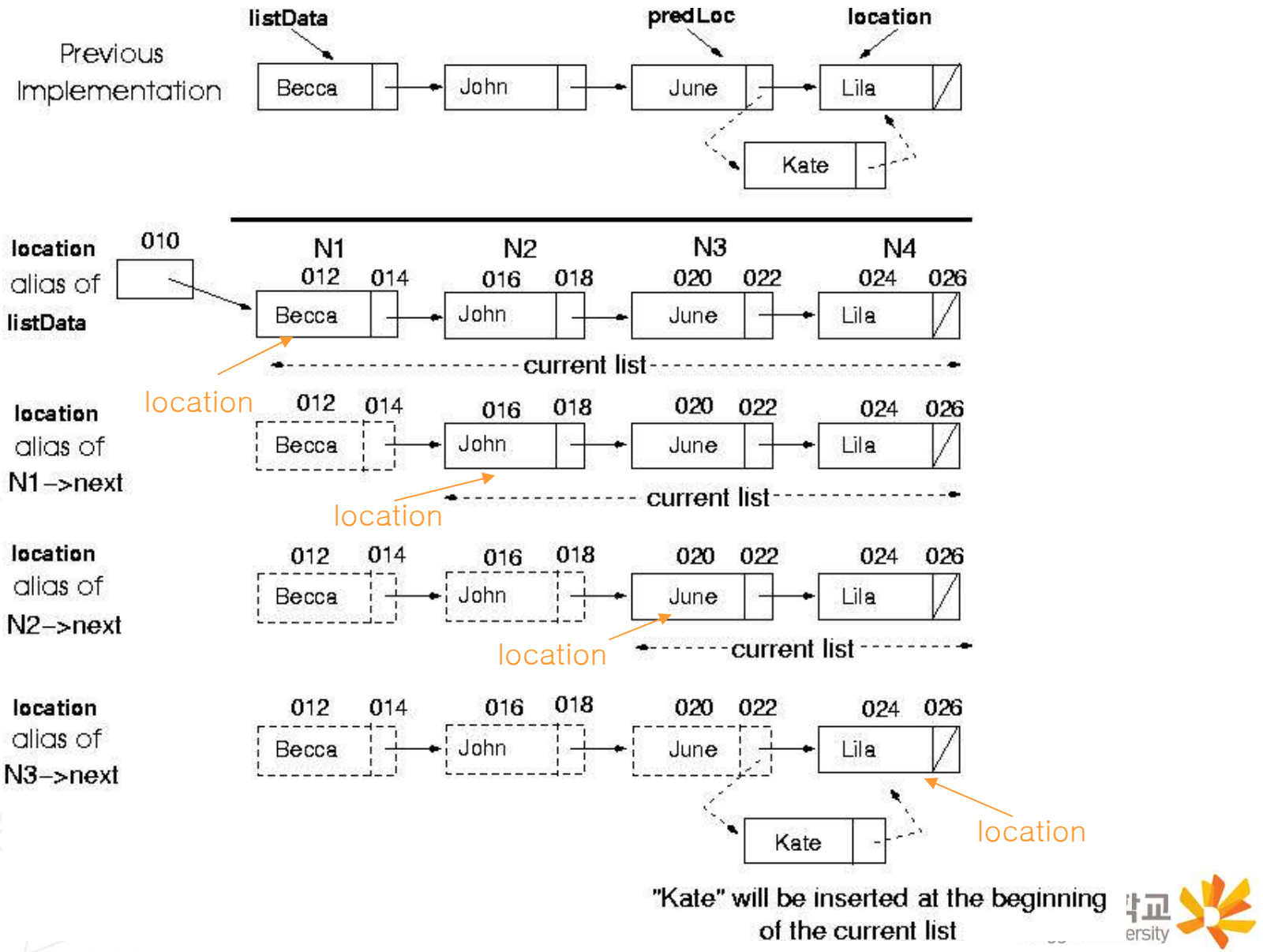
Base Case and General Case

- 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
          cout   <<   listPtr->info    <<   endl ;     // then print   this element
   }
                    // Base case : if the list is empty, do nothing
}
```

Recursive InsertItem (sorted list)



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 at the front in the current list

- What is the *general case*?

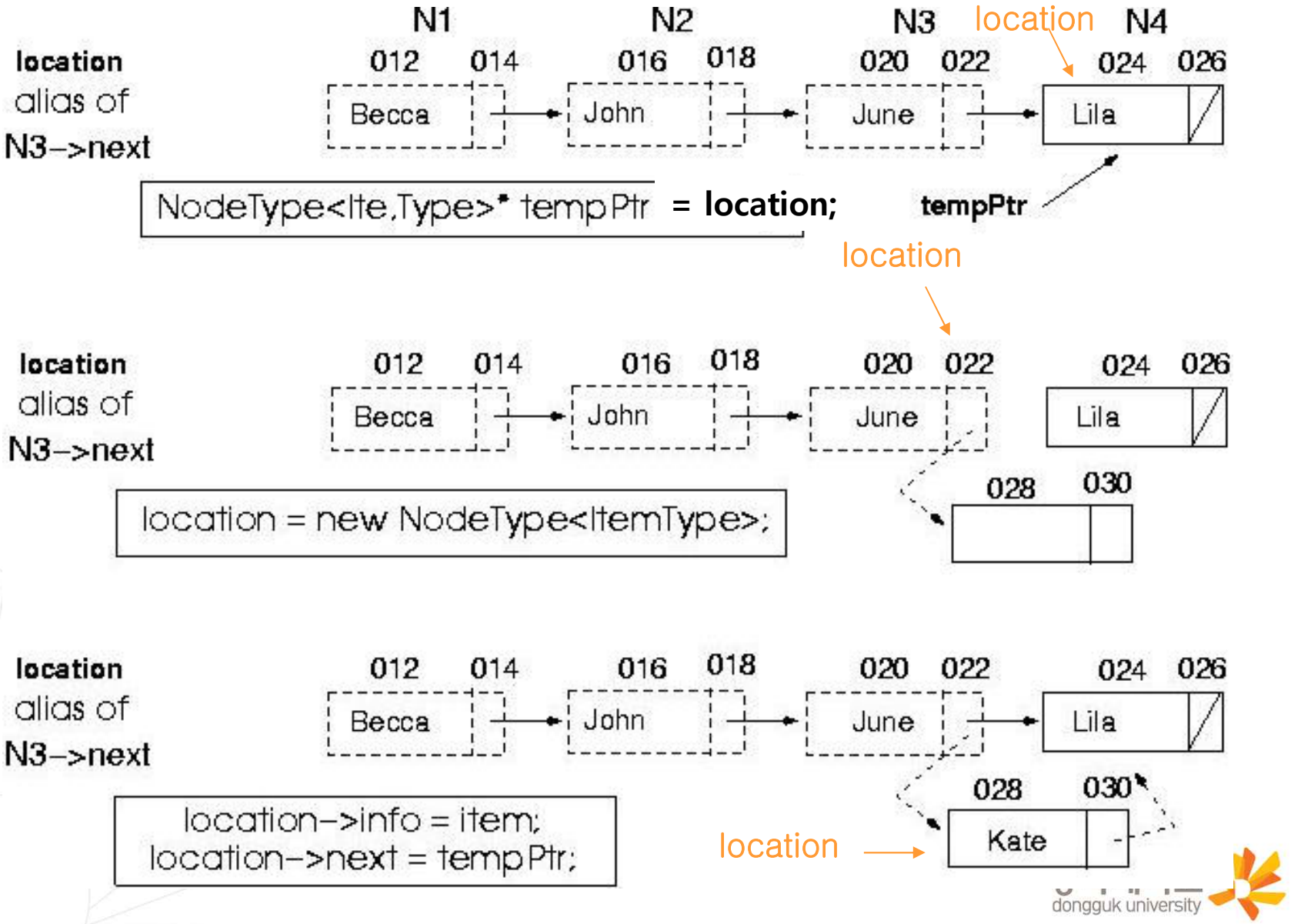
  Insert(location->next, item)

Recursive InsertItem (sorted list)

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

```
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
}
```

Note: no "predLoc" pointer is needed for insertion!



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*?

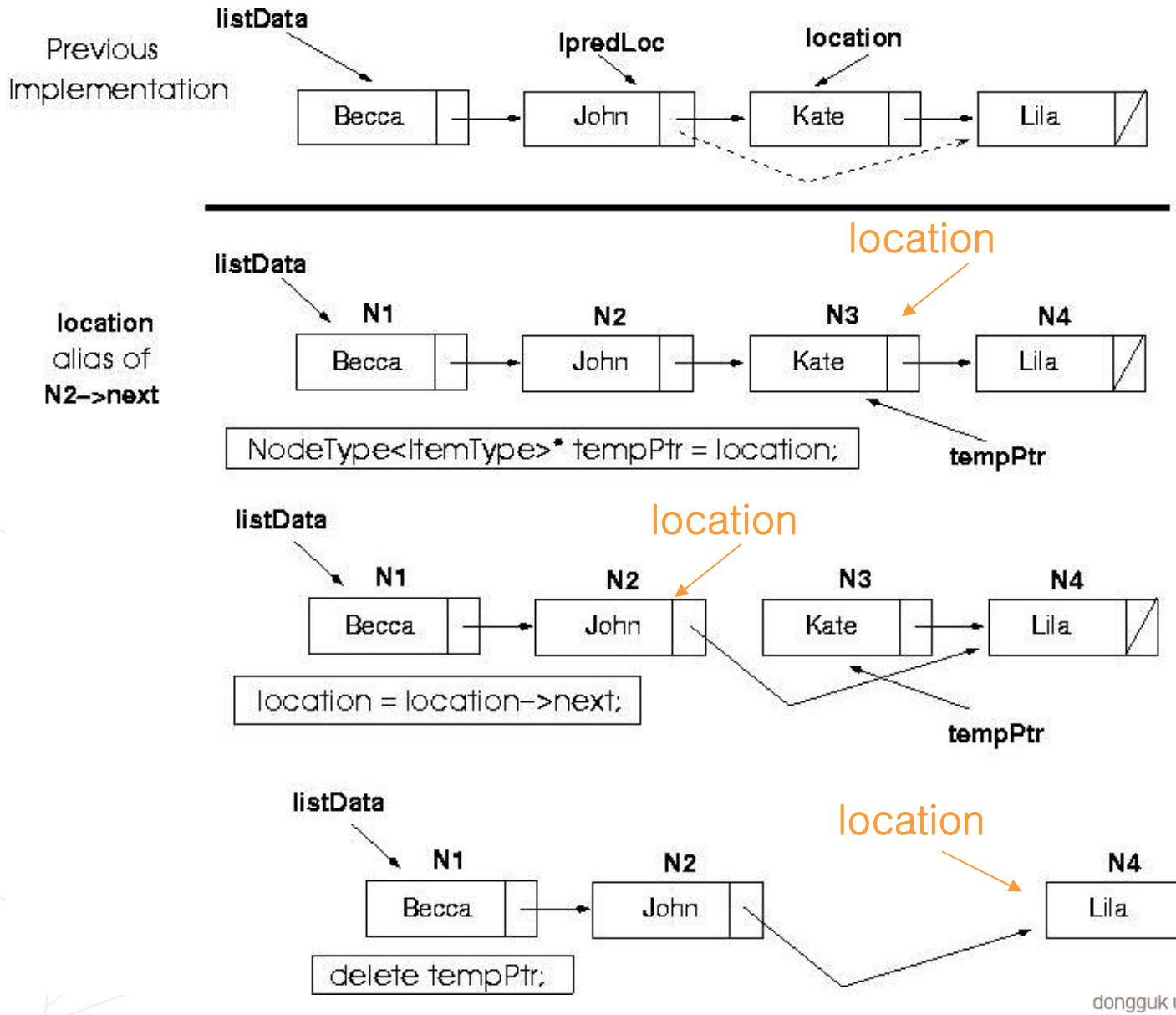
  Delete(location->next, item)

Recursive DeleteItem (sorted list) (cont.)

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

```
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);
}
```

Recursive DeleteItem (sorted list)



Deciding whether to use a recursive solution ...

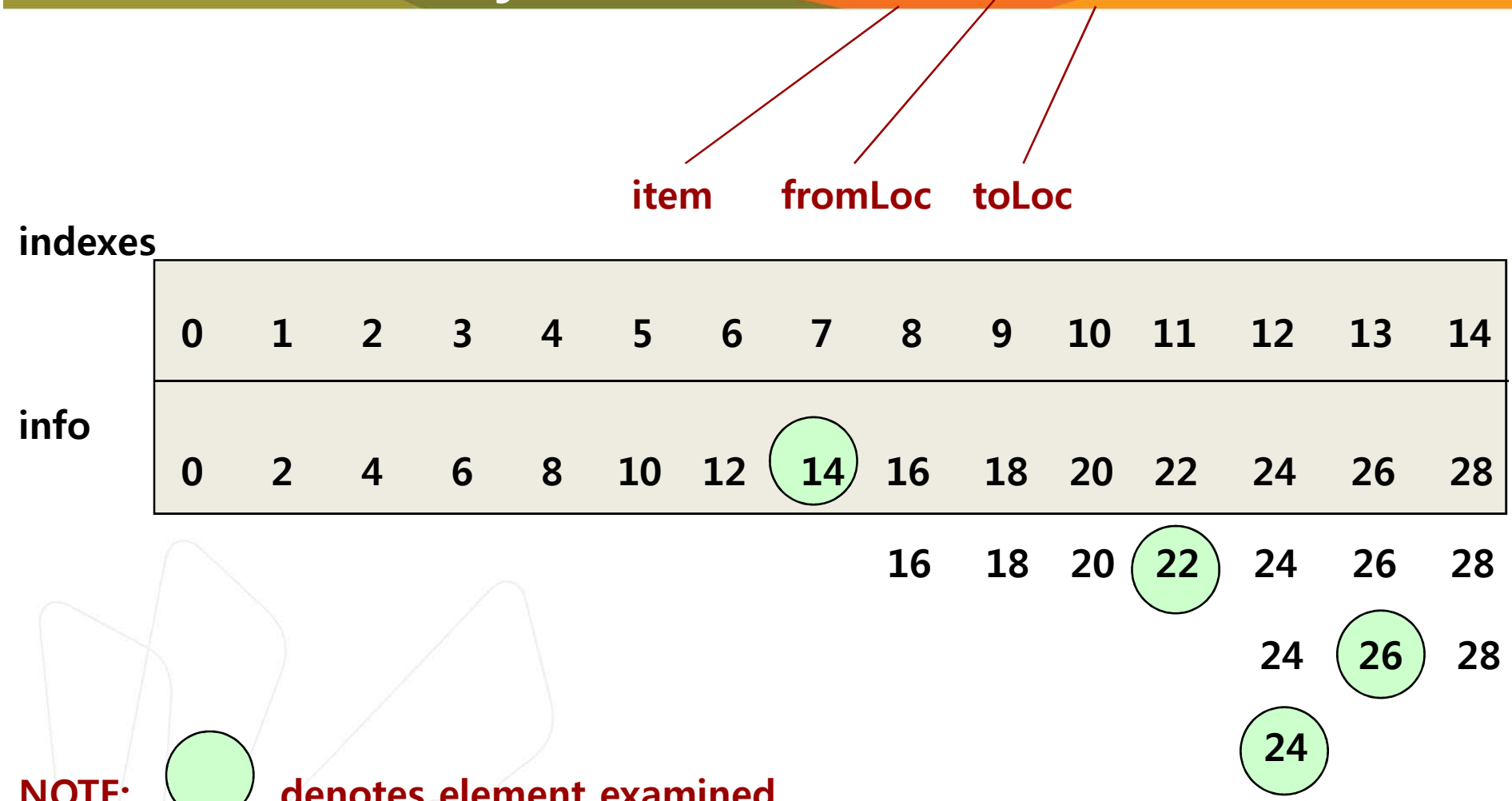
- The recursive version is shorter and simpler than the non-recursive solution.
- The depth of recursive calls is relatively "shallow“.
- The recursive version does about the same amount of work as the non-recursive version.

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);



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 ) ;
                }
}
```

How Recursion Works

- ***Static storage allocation*** **associates variable names with memory locations at** ***compile time.***

- ***Dynamic storage allocation*** **associates variable names with memory locations at** ***execution time.***

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 is 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.

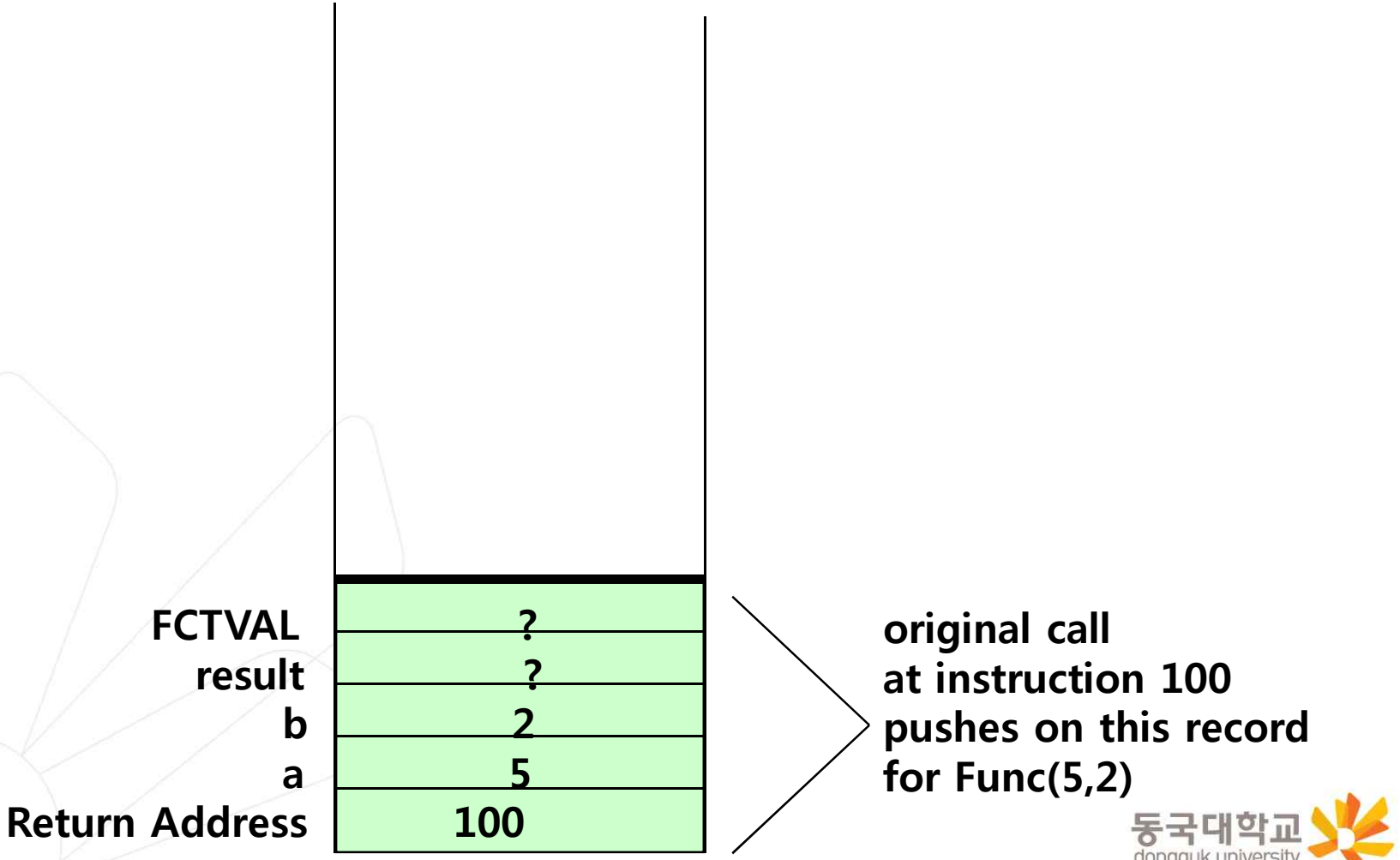
Stack Activation Frames

- 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;
}
```

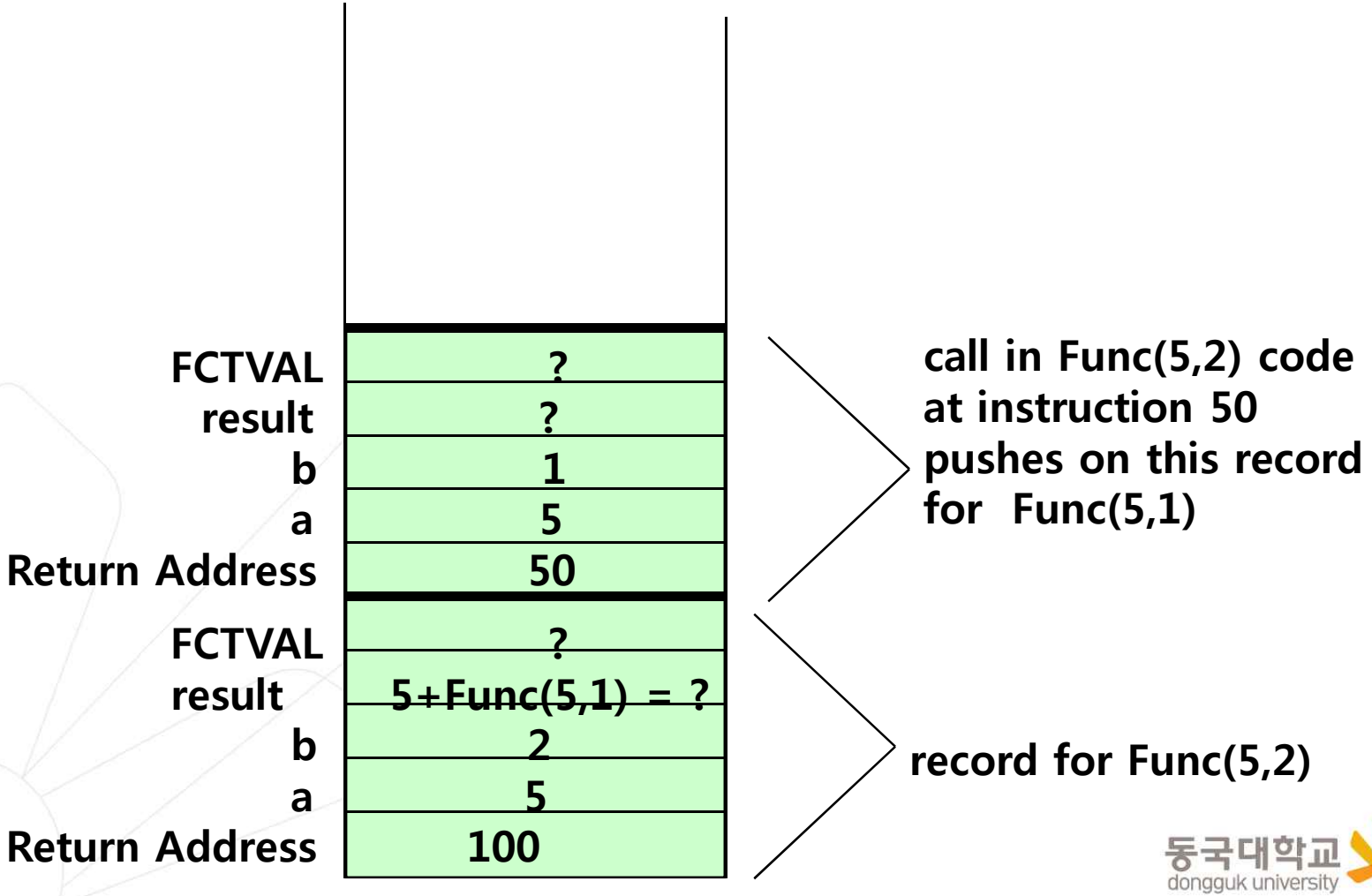
Run-Time Stack Activation Records

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



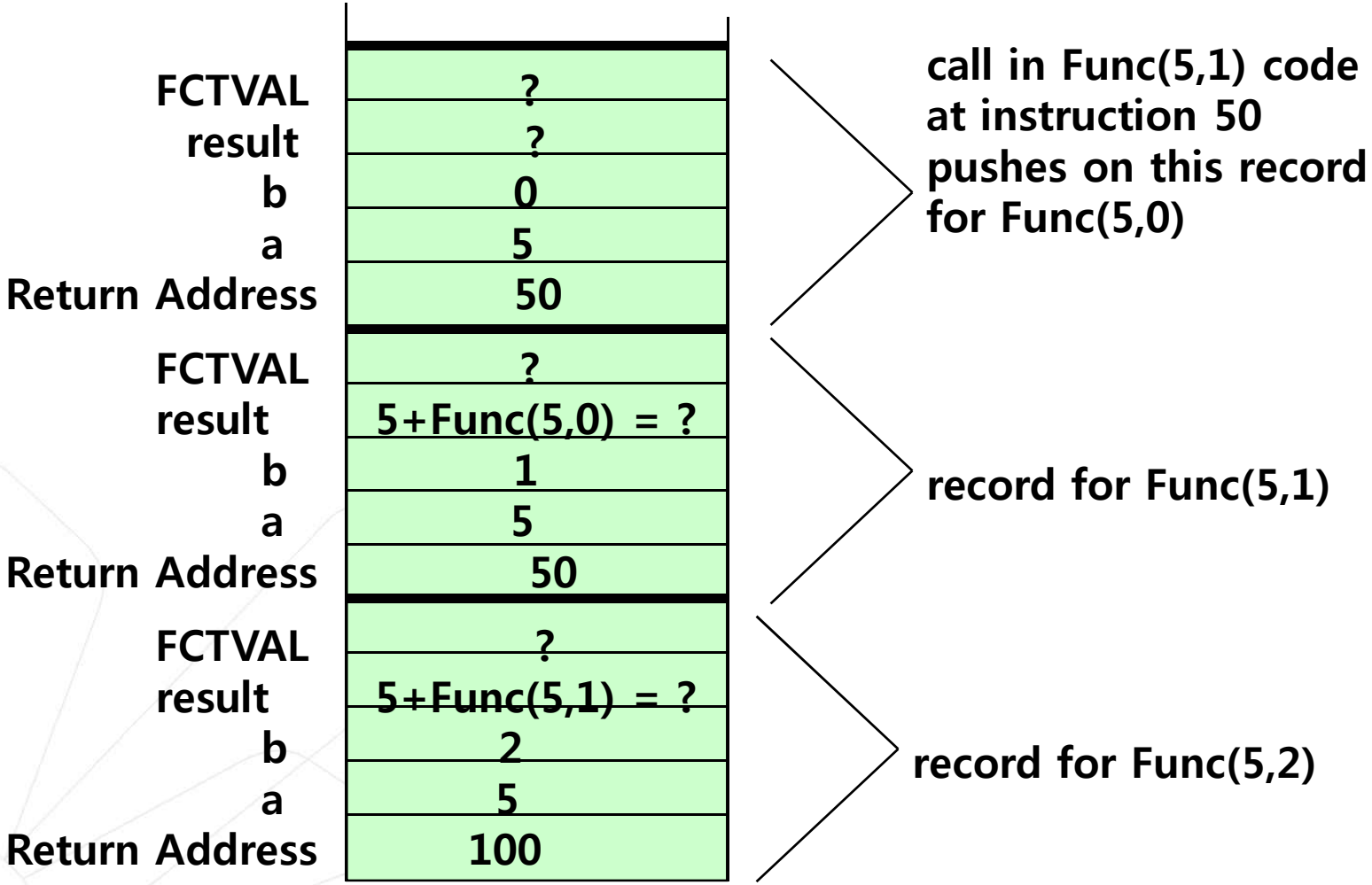
Run-Time Stack Activation Records

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



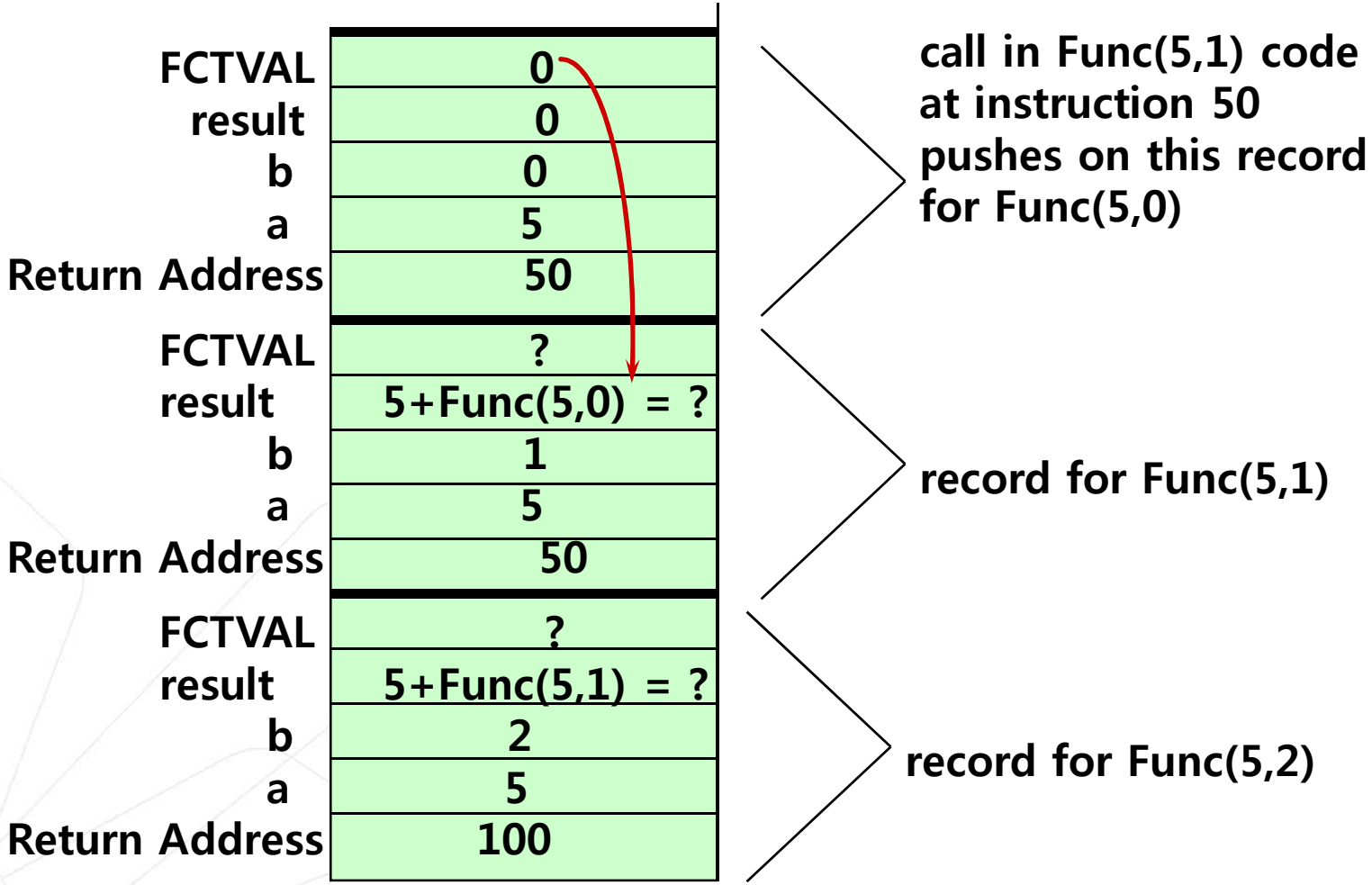
Run-Time Stack Activation Records

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



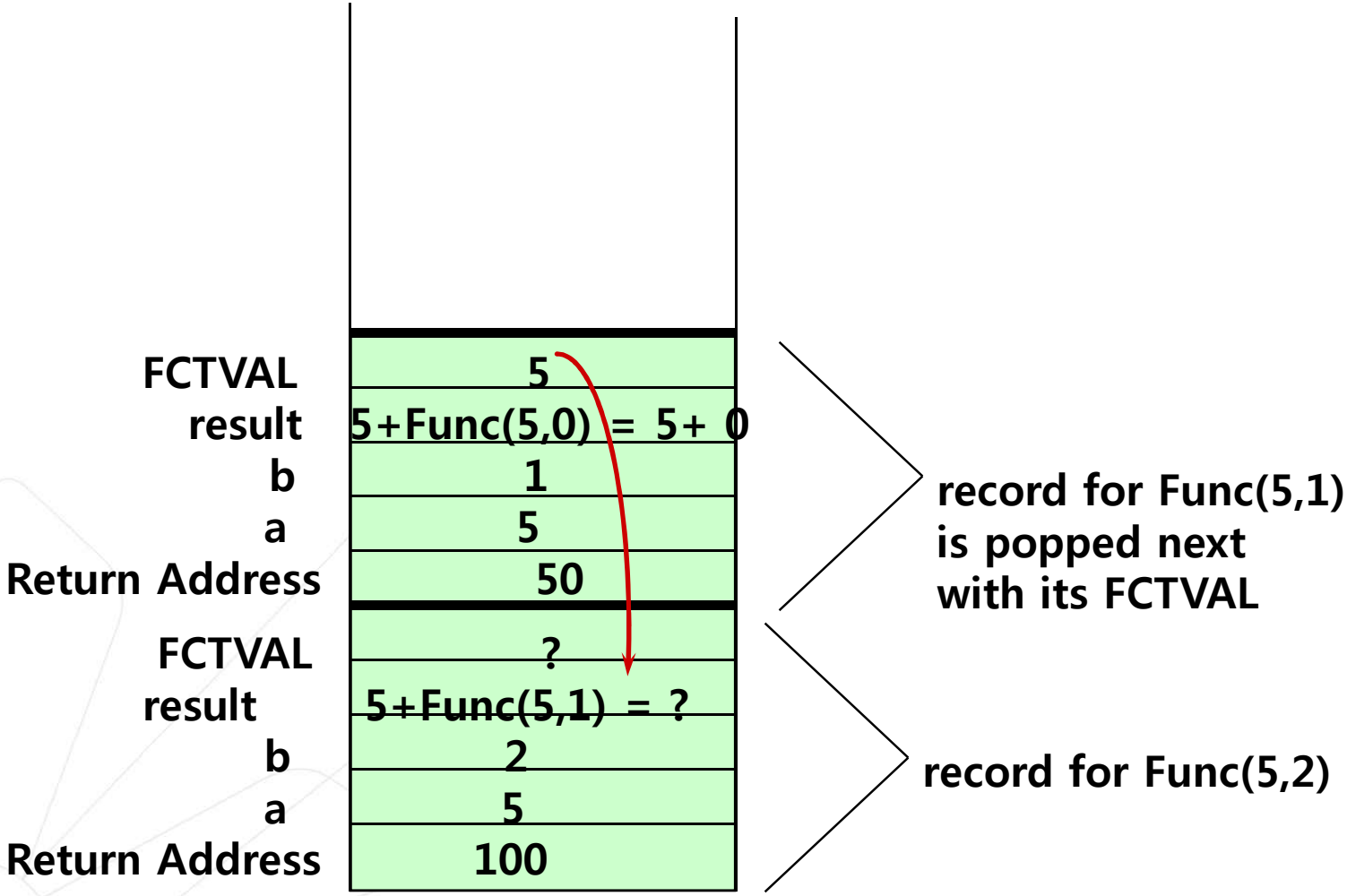
Run-Time Stack Activation Records

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



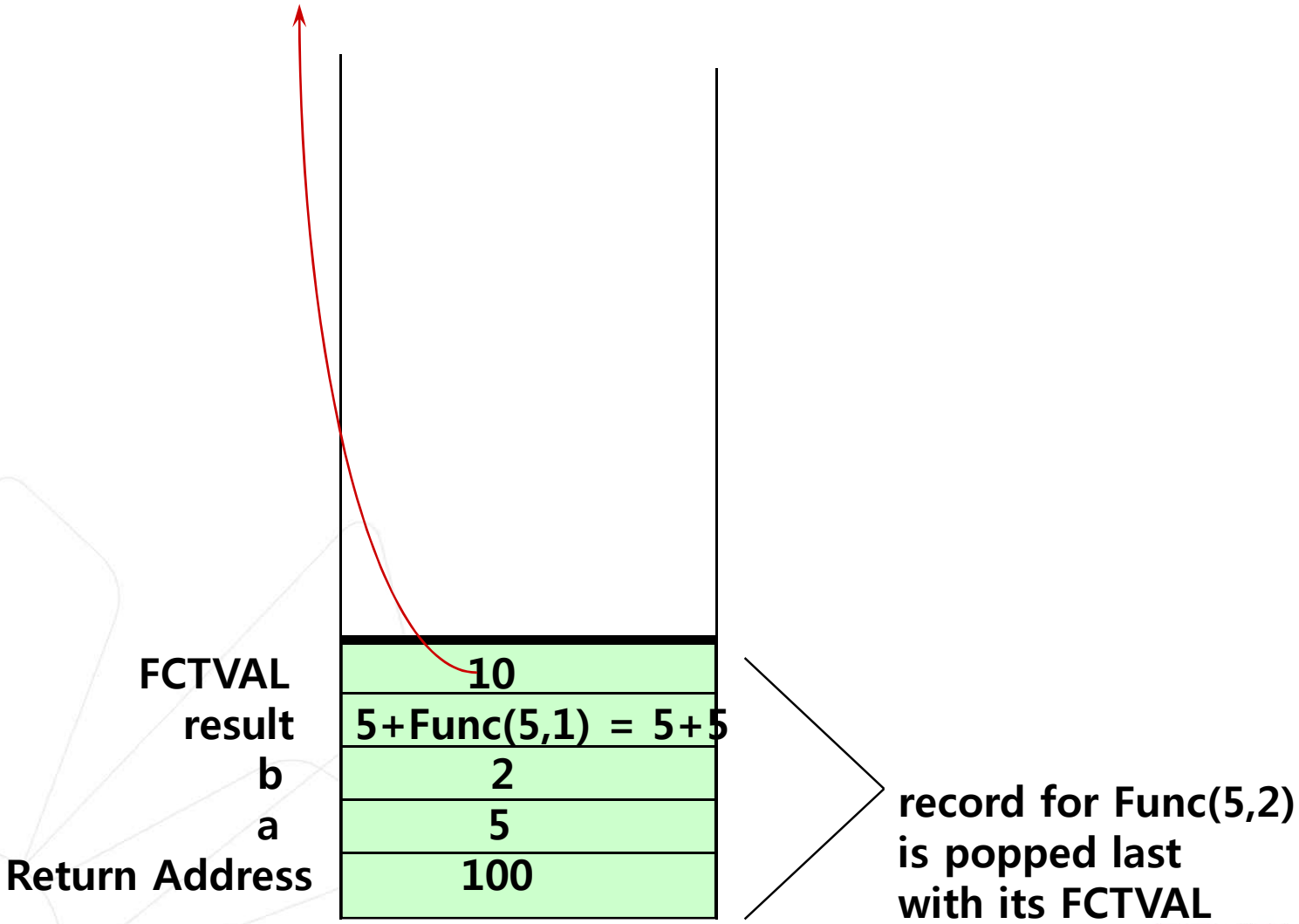
Run-Time Stack Activation Records

```
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?

Debugging Recursive Routines

- Using the Three-Question Method.
- Using a branching statement (if/switch).
- Put debug output statement during testing.
- …

Removing Recursion

When the language doesn't support recursion, or recursive solution is too costly (space or time), or ...

- **Iteration**
- **Stacking**

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 ;
}
```

Stacking

Replace the stack that was done by the system with stacking that is done by the programmer.

An example…

Recursive implementation:

```
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
      cout   <<   listPtr->info    <<   endl ;       // then print   this element

   }
                  // Base case : if the list is empty, do nothing
}
```

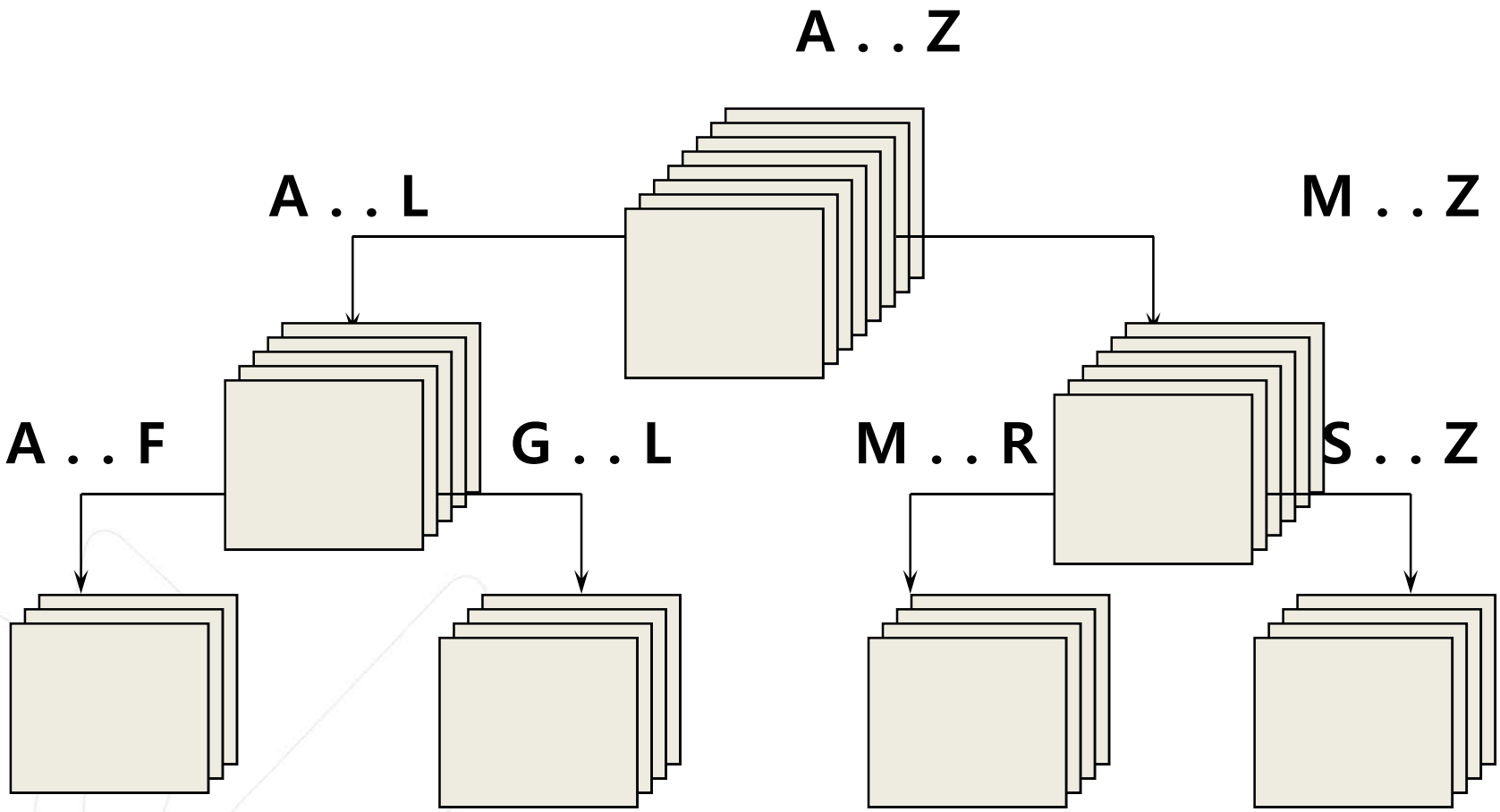
Nonrecursive implementation:

```
#include “StackType.h”
void   ListType :: RevPrint ( )
{
   StackType<NodeType*> stack;
   NodeType* listPtr;

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

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.

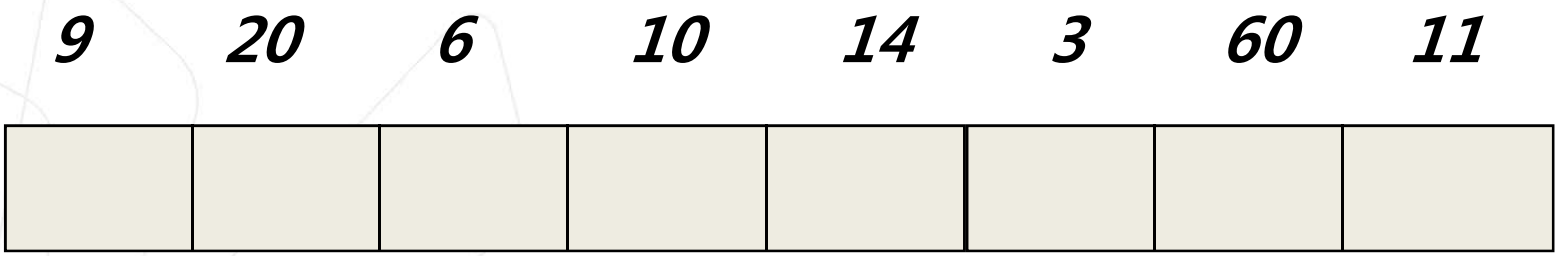
Using quick sort algorithm



Before call to function Split

splitVal = 9 (=values[first])

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

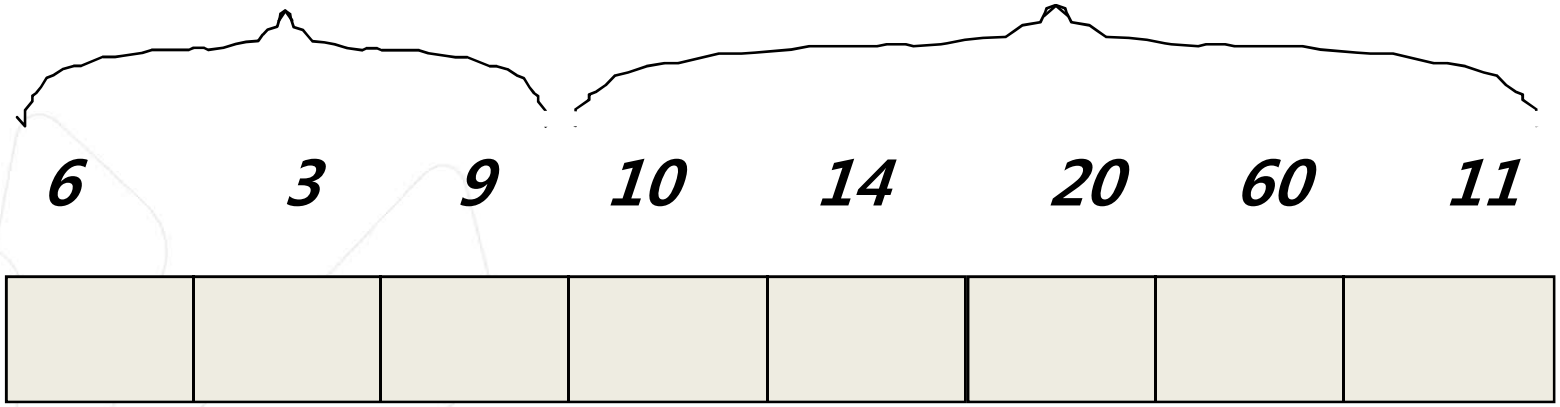


values[first] [last]

After call to function Split

splitVal = 9

smaller values **larger values**



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 ) ;
          // split the array so that
          // 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 );
  }

}
```