Data Structures

Part VII: Programming with Recursion

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

Recursive Function Call

- 재귀 호출은 호출된 함수가 호출하는 함수와 동일한 함수 호출
  A recursive call is a function call in which the called function is the same as the one making the call.

- 즉, 함수가 자신을 호출하면 재귀 발생
  In other words, recursion occurs when a function calls itself!

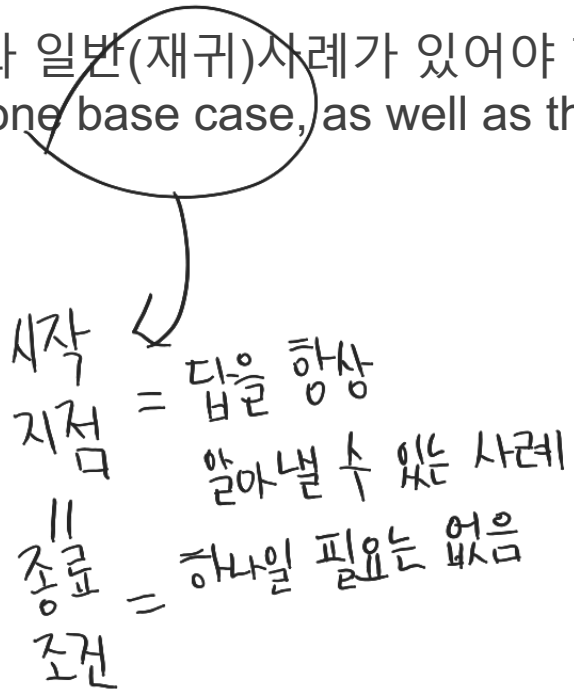
- 무한 함수 호출 (무한 재귀)는 피해야 함
  We must avoid making an infinite sequence of function calls (infinite recursion).

Finding a Recursive Solution

- 각 연속적 재귀 호출은 응답이 알려진 상황에 더 가까이 다가가야 함
  Each successive recursive call should bring you closer to a situation in which the answer is known.

- 답이 알려진 (재귀 없이 표현될 수 있는) 경우를 기본사례라고 함
  A case for which the answer is known (and can be expressed without recursion) is called a base case.

- 각 재귀 알고리즘에는 하나 이상의 기본사례와 일반(재귀)사례가 있어야 함
  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
```

Writing a recursive function to find n factorial

- DISCUSSION

  - ✓ Factorial(4) 함수 호출의 값은 4*3*2*1이므로 24여야 함
    The function call Factorial(4) should have value 24, because that is 4 * 3 * 2 * 1 .

  - ✓ 답이 알려진 상황의 경우: 0!의 값은 1
    For a situation in which the answer is known, the value of 0! is 1.

  - ✓ 따라서 기본 사례는
    So our base case could be along the lines of

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

Writing a recursive function to find Factorial(n)

- 일반적인 경우에서..
  Now for the general case . . .

  ✓ Factorial(n)의 값은 n*(n-1)에서 1까지의 곱으로 쓸 수 있으며, 아래와 같이 표현할 수 있음
  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, $n * Factorial(n - 1)$ → 언젠가는 Base Case에 도달

  ✓ 재귀 호출 Factorial(n-1)은 Factorial(0)의 기본사례에 "가까이" 도달함
  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

- 기본사례 질문: 함수에서 비 재귀적인 방법이 있나?
  Base-Case Question: Is there a non-recursive way out of the function?

재귀의 장점: 단점:

- 작은 경우 질문: 각 재귀 함수 호출에는 기본사례로 이어지는 원래 문제의 작은 사례가 포함되나?
  Smaller-Caller Question: Does each recursive function call involve a smaller case of the original problem leading to the base case?

- 일반 사례 질문: 각 재귀 호출이 올바르게 동작한다고 가정하면 전체 기능이 올바르게 동작하나?
  General-Case Question: Assuming each recursive call works correctly, does the whole function work correctly?

함수를 매번 호출 : 비용이 많이 듦
⇒ 근데 무조건 재귀가 비효율적인 건 아님

Another example where recursion comes naturally

- 수학에서 우리는..
  From mathematics, we know that

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

- 일반적으로..
  In general,

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

for integer x, and integer n > 0.

- 우리는 x^{n-1}의 관점에서 x^n을 재귀적으로 정의할 수 있음
  Here we are defining x^n recursively, in terms of x^{n-1}

Another example where recursion comes naturally

```
// 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.

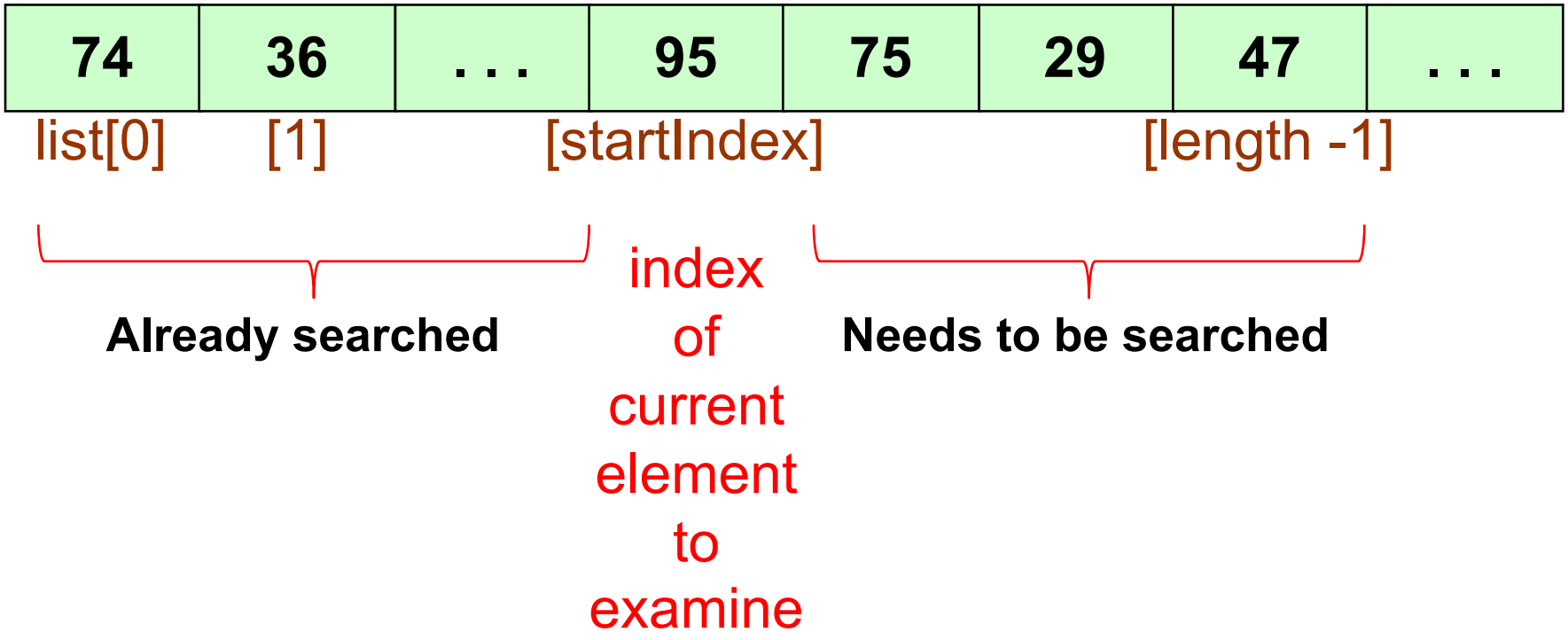
struct ListType

```
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) ;



Recursive function to determine if value is in list

```
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?

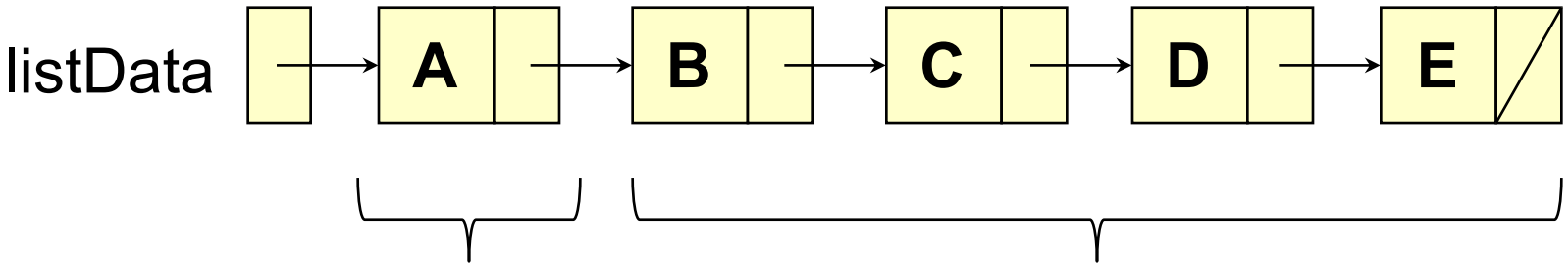
- 이러한 예제는 반복을 사용하여 재귀 없이 작성될 수 있음
  Those examples could have been written without recursion, using iteration i nstead.
  - ✓ 반복 솔루션은 루프를 사용하고, 재귀 솔루션은 if문을 사용
    The iterative solution uses a loop, and the recursive solution uses an if statement

- 하지만, 특정 문제의 경우 재귀 솔루션이 가장 자연스러운 방법임
  However, for certain problems the recursive solution is the most natural solution.
  - ✓ 포인터 변수가 사용될 때 종종 발생
    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);



THEN, print this element

FIRST, print out this section of list, backwards

Base Case and General Case

- 기본 사례는 “작은”목록의 관점에서 해결책이 될 수 있음
  A base case may be a solution in terms of a “smaller” list.
  - ✓ 확실히 요소가 0인 목록의 경우 더 이상 처리할 작업이 없음
    Certainly for a list with 0 elements, there is no more processing to do.

- 일반사례는 기본사례 상황에 더 가까이 다가가야 함.
  Our general case needs to bring us closer to the base case situation.
  - ✓ 즉, 처리될 리스트 요소의 수는 각 재귀 호출마다 1씩 감소
    That is, the number of list elements to be processed decreases by 1 with each recursive call.
  - ✓ 일반사례에서 하나의 요소를 출력하고, 더 작은 나머지 목록을 처리하면 결국 0개의 목록 요소가 처리될 상황에 도달
    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.

- 일반상황에서 나머지 작은 목록의 요소를 역순으로 출력한 다음 현재 지정된 요소를 출력
  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 element
    }
    // Base case : if the list is empty, do nothing
}
```

Function BinarySearch( )

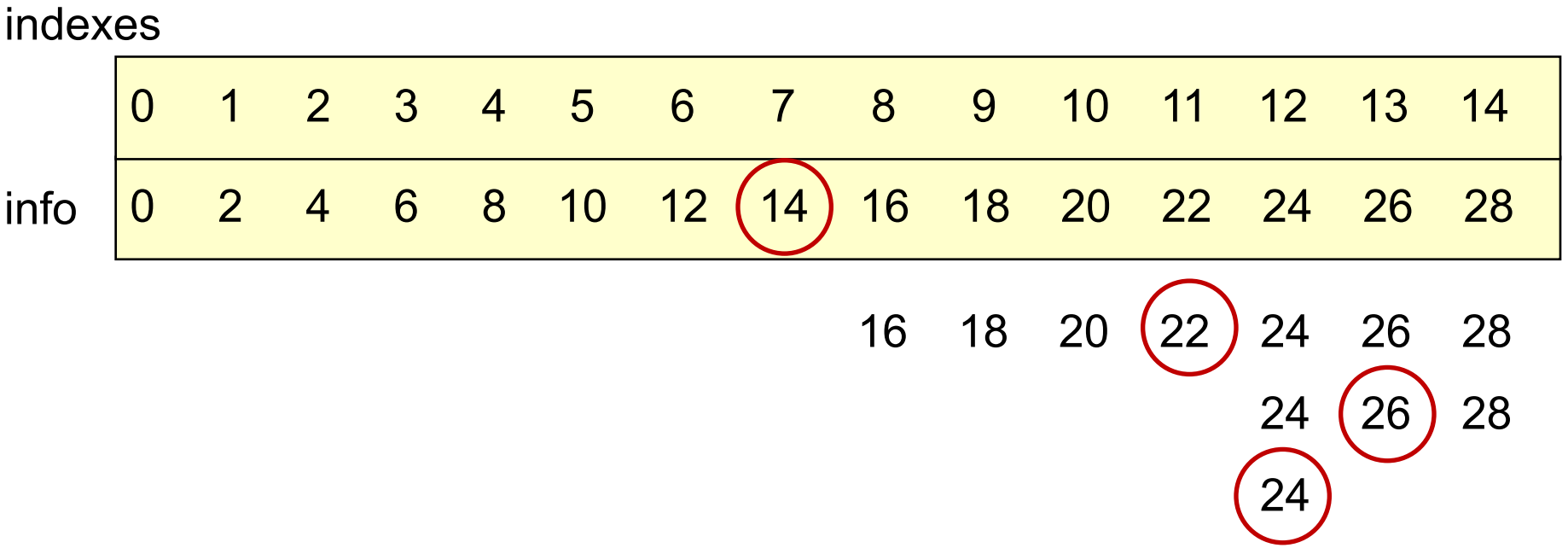
- 이진탐색은 정렬된 배열 정보와 두 개의 첨자 fromLoc, toLoc을 인수로 취함
  BinarySearch takes sorted array **info**, and two subscripts, **fromLoc** and **toLoc**, and item as arguments.
  - ✓ Info[fromLoc…toLoc] 요소에 항목이 없으면 false를 반환
    It returns false if item is not found in the elements **info[fromLoc…toLoc]**.
  - ✓ 그렇지 않으면 true 반환
    Otherwise, it returns true.

- 이진탐색은 반복 및 재귀를 사용하여 작성 가능
  BinarySearch can be written using iteration, or using recursion

Function BinarySearch( )

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

item fromLoc toLoc



NOTE: ◯ denotes element examined

Function BinarySearch( )

```
// 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...

- 호출 블록에서 함수 코드로 제어 전환이 발생
  A transfer of control occurs from the calling block to the code of the function.
  - ✓ 함수 코드가 실행된 후, 호출블록의 올바른 위치로 되돌아가야함
    It is necessary that there be a return to the correct place in the calling block after the function code is executed.
  - ✓ 이 올바른 장소를 반환주소라 함
    This correct place is called the return address.

- 함수가 호출되면 런타임 스택이 사용됨
  When any function is called, the run-time stack is used.
  - ✓ 이 스택에는 함수 호출을 위한 활성화 레코드(스택프레임)가 배치됨
    On this stack is placed an activation record(stack frame) for the function call.

스택을 쓴다는 게 중요함

Stack Activation Frames

- 활성화 레코드는 이 함수 호출에 대한 리턴 주소와 매개 변수, 로컬 변수 및 void가 아닌 경우 함수의 리턴 값을 저장
  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.

- 함수 코드의 최종 닫는 중괄호에 도달하거나 함수 코드에서 return 문에 도달하면 특정 함수 호출에 대한 활성화 레코드가 런타임 스택에서 pop됩니다.
  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.

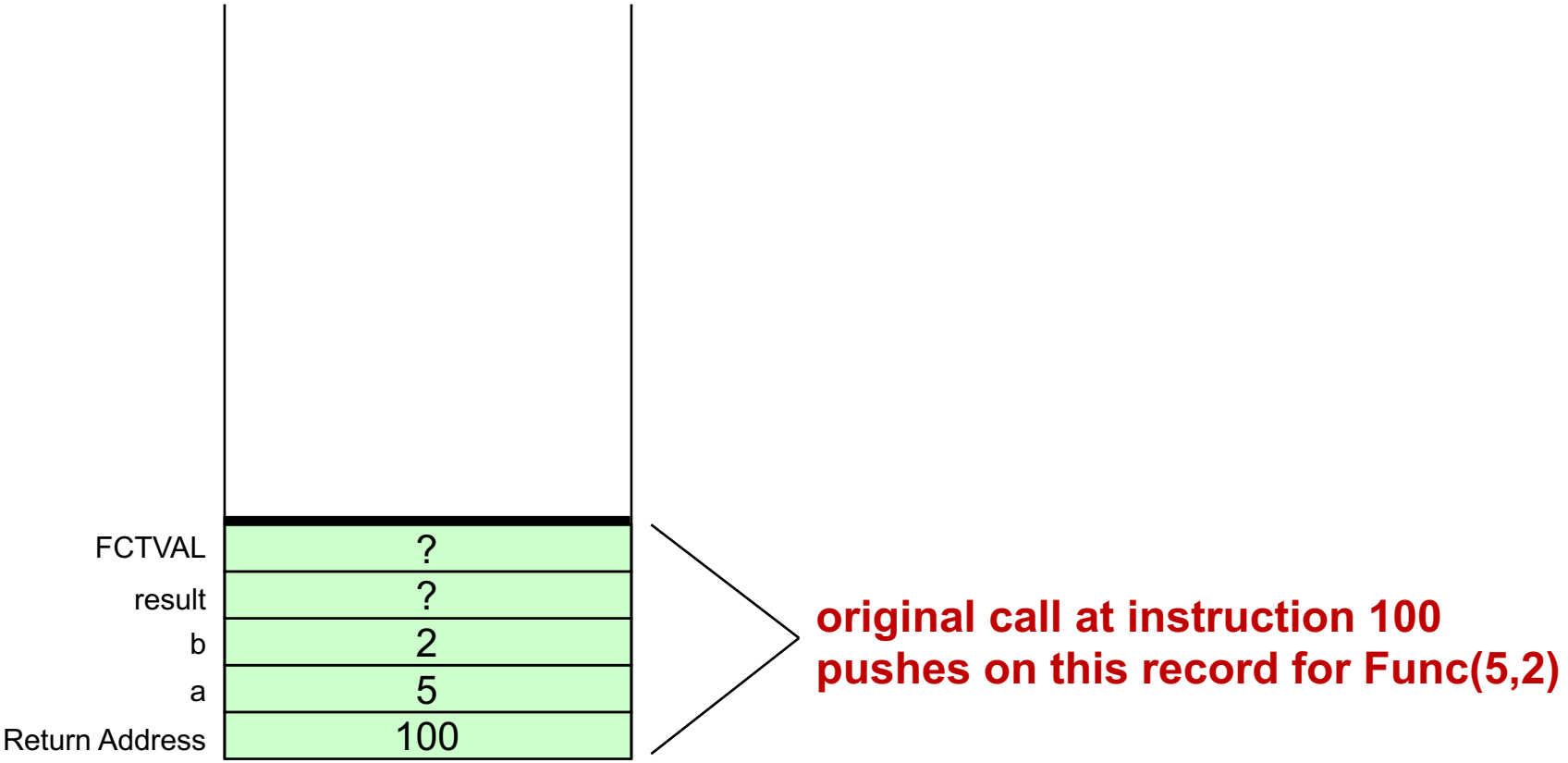
- 이때, void가 아닌 경우 함수의 반환 값은 호출 블록 반환 주소로 다시 가져와서 사용할 수 있습니다.
  At this time the function's return value, if non-void, is brought back to the calling block return address for use there.

Example

```
// 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;
}
```

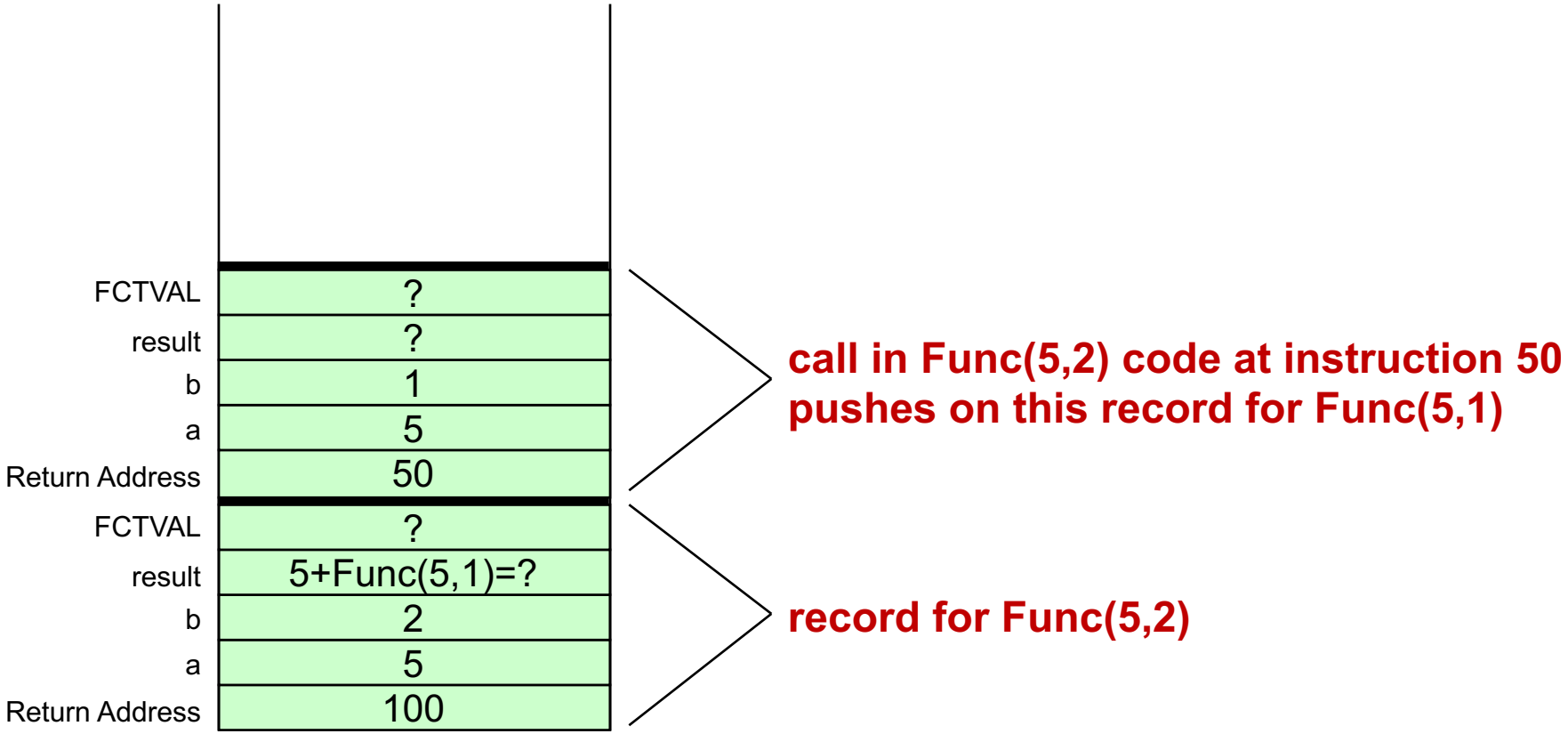
Run-Time Stack Activation Records

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



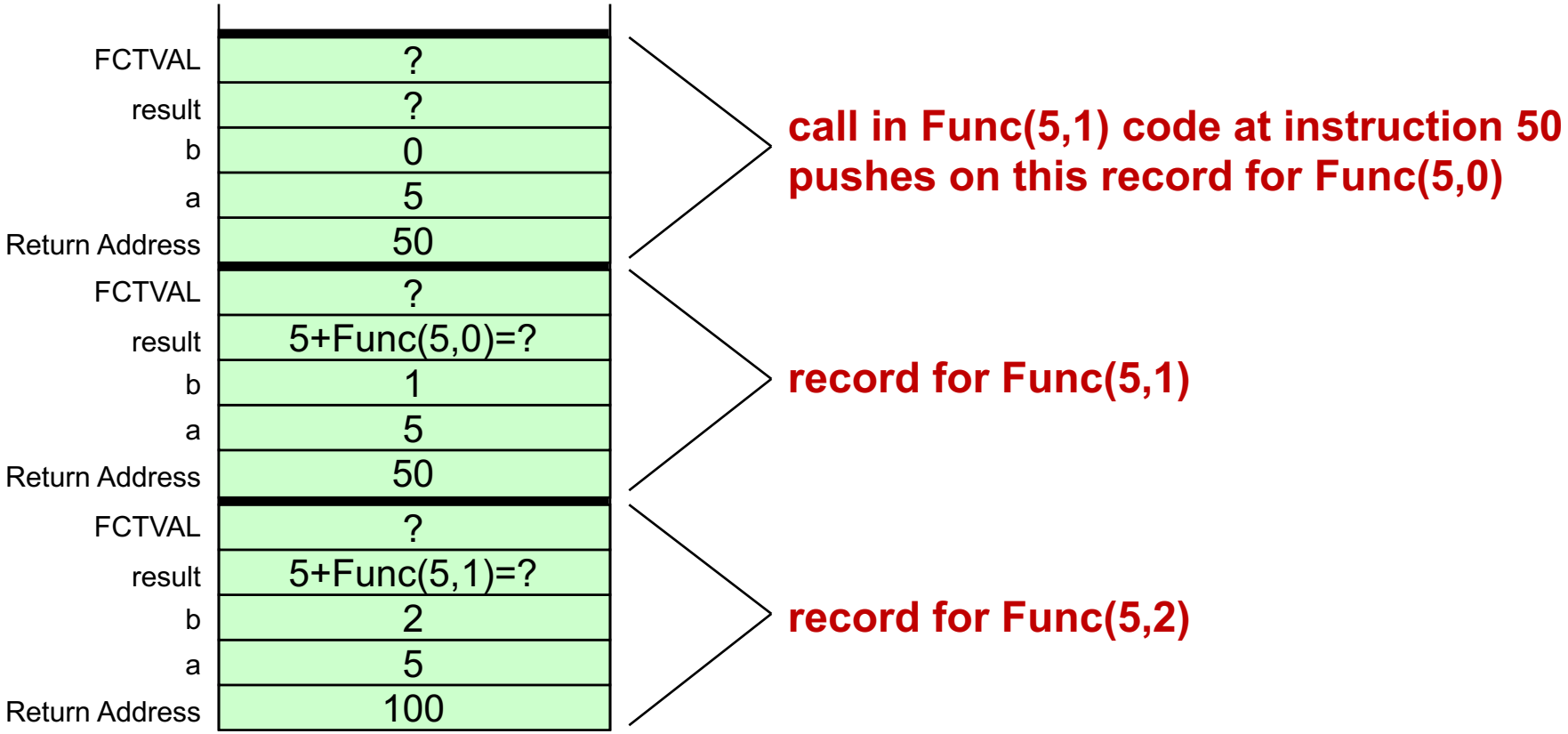
Run-Time Stack Activation Records

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



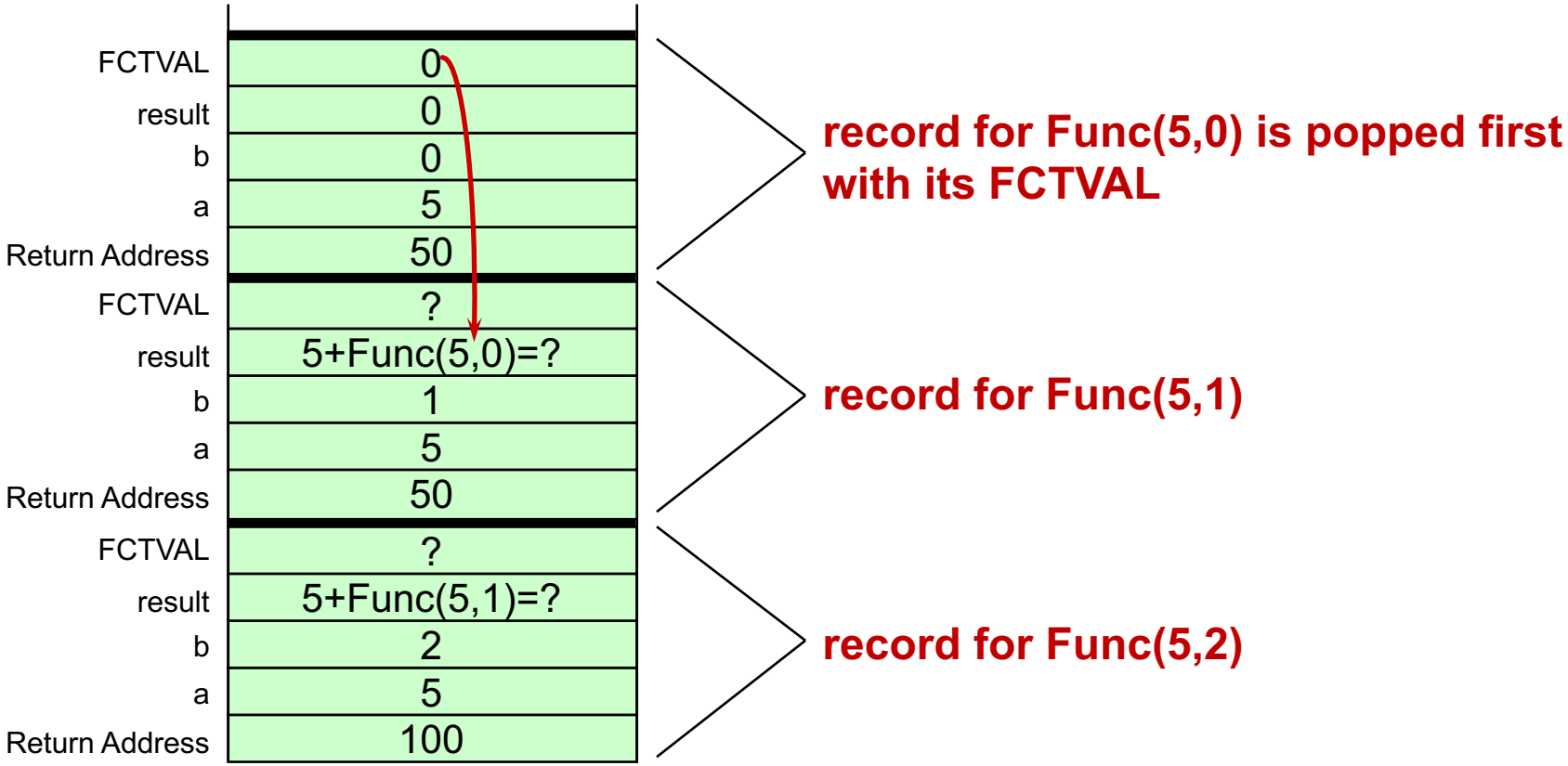
Run-Time Stack Activation Records

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



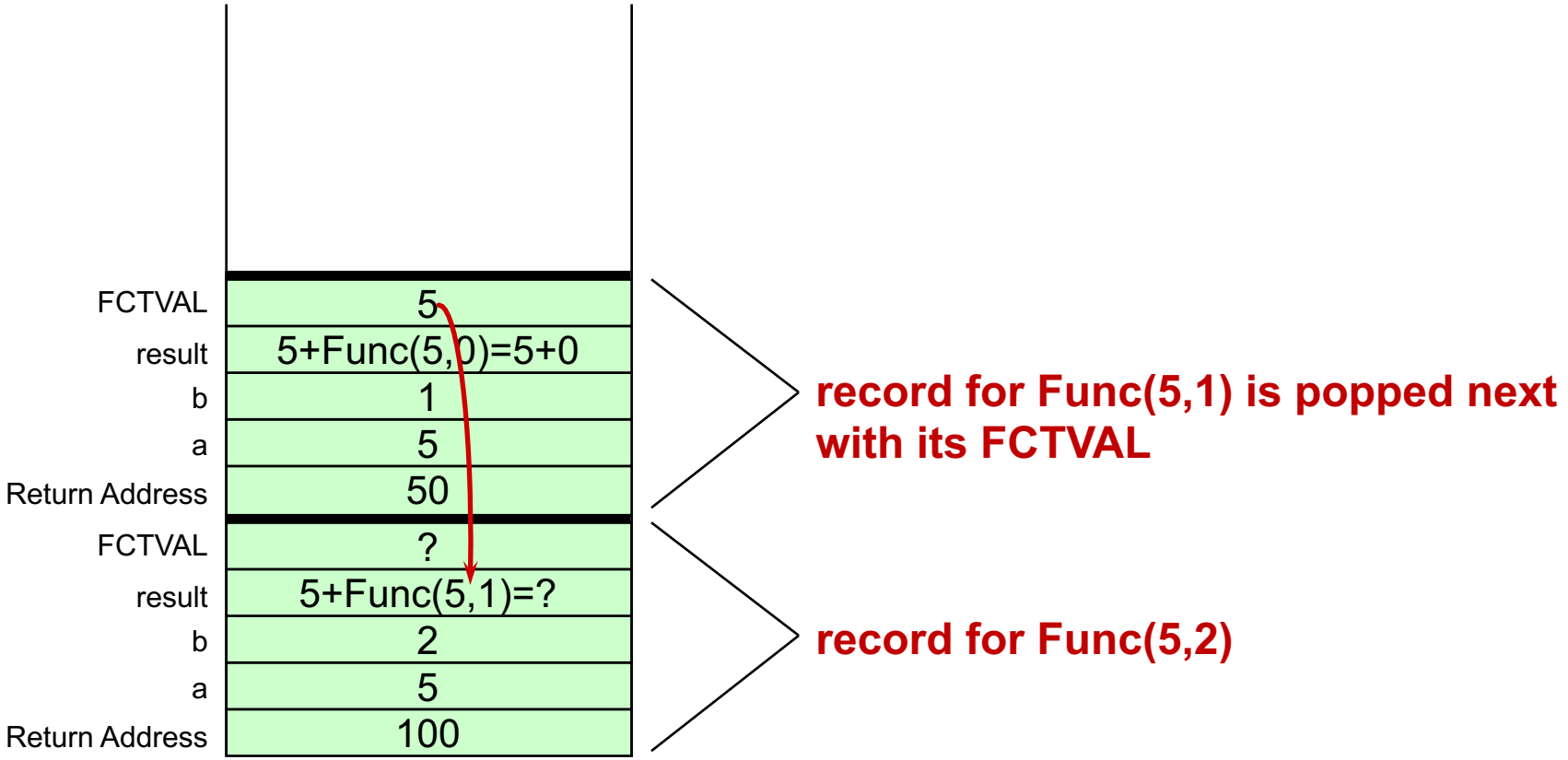
Run-Time Stack Activation Records

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



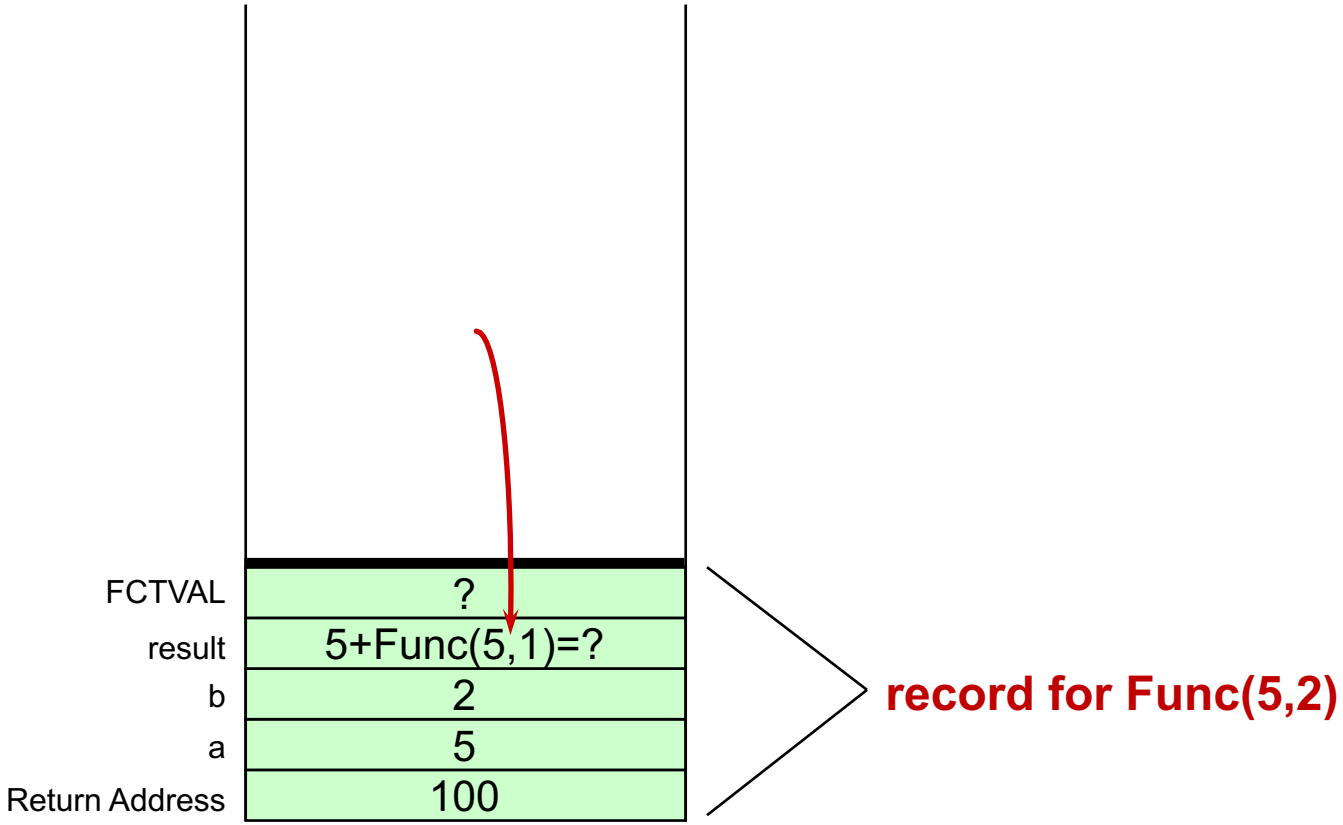
Run-Time Stack Activation Records

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



Run-Time Stack Activation Records

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



Show Activation Records for these calls

x = Func(- 5, - 3);

x = Func(5, - 3);

- Func(a, b)를 시뮬레이션 하면 어떻게 동작하나?
  What operation does Func(a, b) simulate?

Tail Recursion

- 함수가 단일 재귀 호출만 포함하고 함수에서 마지막으로 실행되는 명령문 인 경우를 의미
  The case in which a function contains only a single recursive call and it is the last statement to be executed in the function.

- 다음 예에서와 같이 테일 재귀를 반복으로 대체하여 솔루션에서 재귀를 제거할 수 있음
  Tail recursion can be replaced by iteration to remove recursion from the solution as in the next example.

Tail Recursion

```
// 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 ) ;
}
```

Tail Recursion

```
// 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 ;
}
```

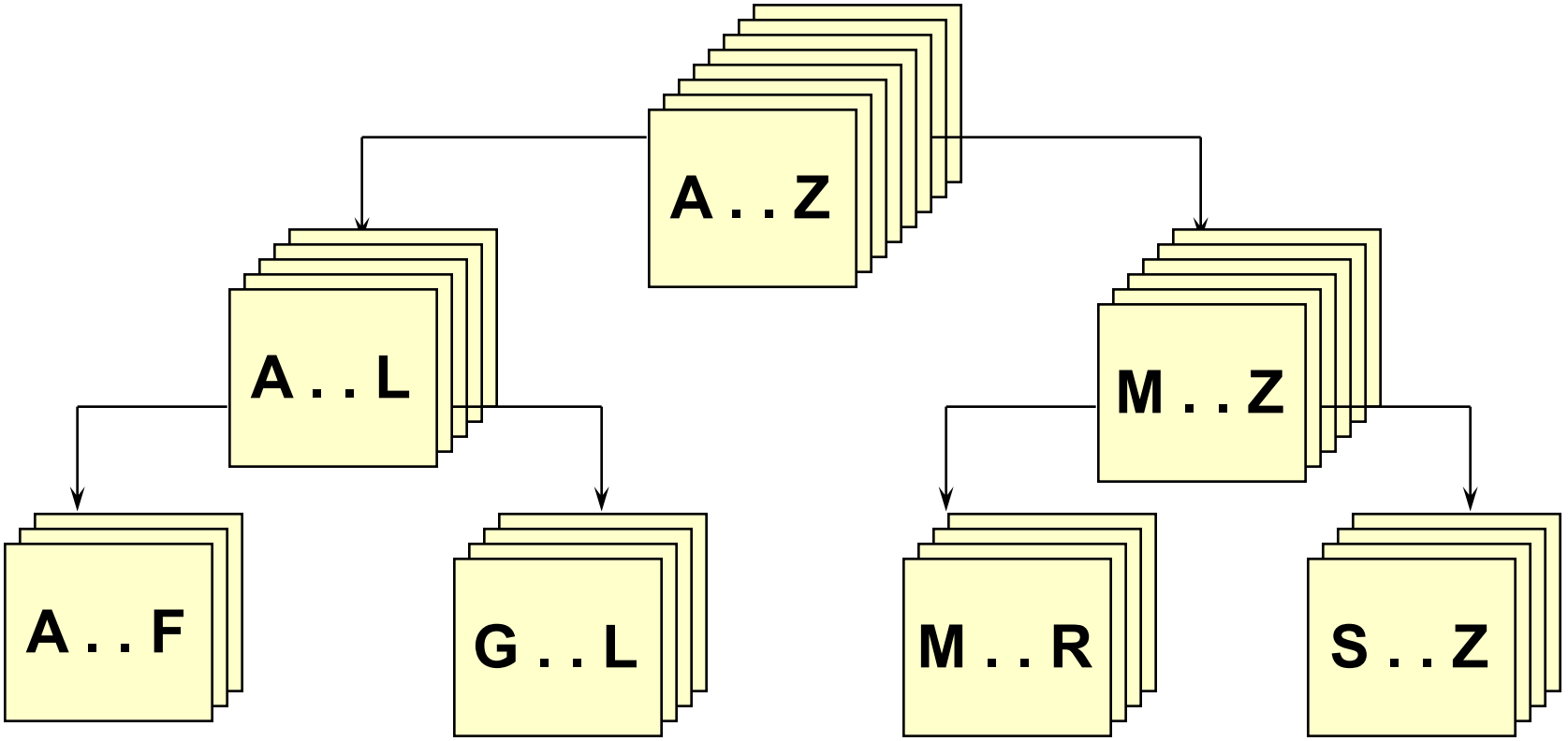
Use a recursive solution when:

- 재귀 호출의 깊이는 문제의 크기에 비해 상대적으로 얕을 때
  The depth of recursive calls is relatively “shallow” compared to the size of the problem.

- 재귀 버전은 비 재귀 버전과 거의 같은 양의 작업을 수행할 때
  The recursive version does about the same amount of work as the non-recursive version.

- 재귀 버전은 비 재귀 솔루션보다 짧고 간결할 때
  The recursive version is shorter and simpler than the non-recursive solution.

Using quick sort algorithm



Before call to function Split

splitVal = 9

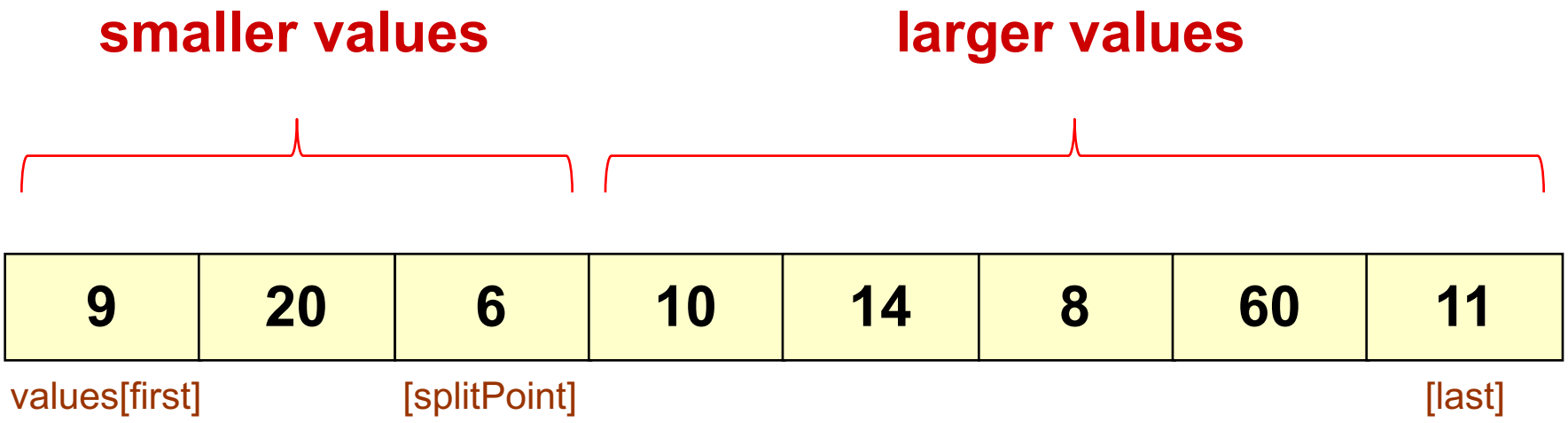
GOAL: splitVal을 왼쪽에 splitVal 이하의 모든 값과 오른쪽에 더 큰 모든 값으로 올바른 위치에 배치
place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	20	6	10	14	8	60	11

values[first] [last]

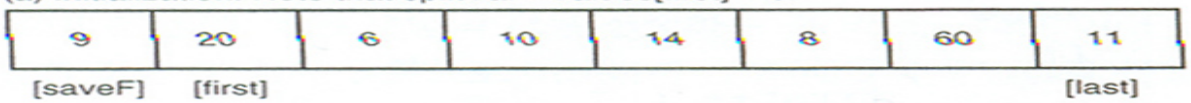
After call to function Split

splitVal = 9



Split function

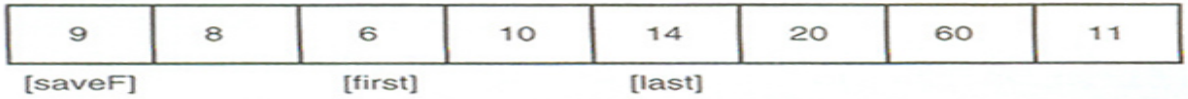
(a) Initialization. Note that splitVal = values[first] = 9.



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

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

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



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

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

Split function

```
// 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 );
     }
}
```

Thank You!
Q&A