

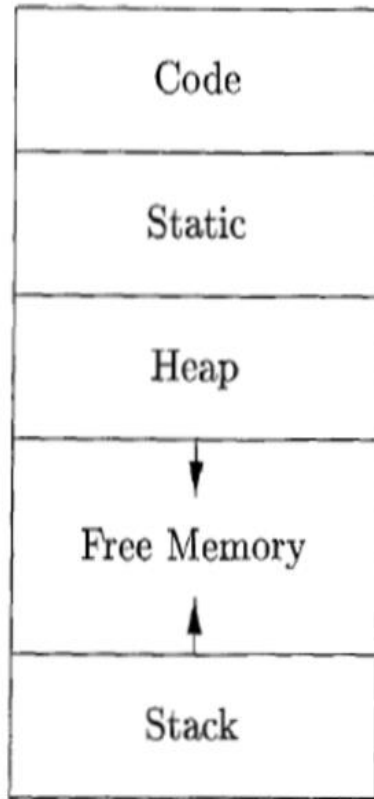
Chapter-07

Run-Time Environments

Storage Organization

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location
- The management and organization of this logical address space is shared between the compiler, operating system and target machine
- The operating system maps the logical addresses into physical addresses which are usually spread throughout memory

Storage Organization(Cont...)



- The run-time representation of an object program in the logical address space consists of data and program areas as shown in figure
- A compiler for a language like C++ on an operating system like Linux might subdivide memory in this way

Typical subdivision of run-time memory into code and data areas

Padding

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine
- On many machines instructions to add integers may expect integers to be aligned that is placed at an address divisible by 4
- Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment, leaving 2 bytes unused
- Space left unused due to alignment considerations is referred to as **padding**

Padding(Cont...)

- When space is at a premium, a compiler may pack data so that no padding is left
- Additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned

Storage Organization(Cont...)

- The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area **Code** usually in the low end of memory
- Similarly, the size of some program data objects such as global constants and data generated by the compiler such as information to support garbage collection may be known at compile time and these data objects can be placed in another statically determined area called **Static**

Storage Organization(Cont...)

- To maximize the utilization of space at run time the other two areas **Stack** and **Heap** are at the opposite ends of the remainder of the address space
- These areas are dynamic
- Their size can change as the program executes
- These areas grow towards each other as needed
- The stack is used to store data structures called activation records that get generated during procedure calls

Storage Organization(Cont...)

- In practice, the stack grows towards lower addresses and the heap towards higher
- However, in this book the authors assume that the stack grows towards higher addresses so that they can use positive offsets for examples

Storage Organization(Cont...)

- An activation record is used to store information about the status of the machine such as the value of the program counter and machine registers when a procedure call occurs
- When control returns from the call the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call
- Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation

Storage Organization(Cont...)

- Many programming languages allow the programmer to allocate and deallocate data under program control
- For example, C has the functions malloc and free that can be used to obtain and give back arbitrary chunks of storage
- The heap is used to manage this kind of long-lived data

Static Versus Dynamic Storage Allocation

- The layout and allocation of data to memory locations in the run-time environment are key issues in storage management
- The two adjectives static and dynamic distinguish between compile time and run time respectively
- We say that a storage-allocation decision is static if it can be made by the compiler looking only at the text of the program not at what the program does when it executes
- Conversely, a decision is dynamic if it can be decided only while the program is running

Static Versus Dynamic Storage Allocation(Cont...)

- Many compilers use some combination of the following two strategies for dynamic storage allocation:
 - **Stack storage:** Names local to a procedure are allocated space on a stack
 - **Heap storage:** Data that may outlive the call to the procedure that created it is usually allocated on a “heap” of reusable storage

Stack Allocation of Space

- Almost all compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack
- Each time a procedure is called space for its local variables is pushed onto a stack and when the procedure terminates that space is popped off the stack
- This arrangement allows space to be shared by procedure calls whose durations do not overlap in time
- It allows to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same regardless of the sequence of procedure calls

Activation Trees

- Stack allocation would not be feasible if procedure calls, or activations of procedures did not nest in time

Example 7.1

- Figure contains a sketch of a program that reads nine integers into an array `a` and sorts them using the recursive quicksort algorithm
- The main function has three tasks
- It calls `readArray`, sets the sentinels, and then calls `quicksort` on the entire data array

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value  $v$ , and partitions  $a[m..n]$  so that
        $a[m..p-1]$  are less than  $v$ ,  $a[p] = v$ , and  $a[p+1..n]$  are
       equal to or greater than  $v$ . Returns  $p$ . */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Sketch of a quicksort program

Example 7.1(Cont...)

- Figure suggests a sequence of calls that might result from an execution of the program
- In this execution, the call to `partition(l,9)` returns 4, so `a[l]` through `a[3]` hold elements less than its chosen separator value `v`, while the larger elements are in `a[5]` through `a[9]`

```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
    ...
    leave quicksort(1,3)
    enter quicksort(5,9)
    ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

Activation Trees(Cont...)

- In this example, as is true in general procedure activations are nested in time
- If an activation of procedure p calls procedure q then that activation of q must end before the activation of p can end
- There are three common cases:
 - The activation of q terminates normally, then in essentially any language control resumes just after the point of p at which the call to q was made

Activation Trees(Cont...)

- There are three common cases:
 - The activation of q or some procedure q called either directly or indirectly aborts; i.e. it becomes impossible for execution to continue, in that case, p ends simultaneously with q
 - The activation of q terminates because of an exception that q cannot handle; Procedure p may handle the exception in which case the activation of q has terminated while the activation of p continues although not necessarily from the point at which the call to q was made; if p cannot handle the exception, then this activation of p terminates at the same time as the activation of q and presumably the exception will be handled by some other open activation of a procedure

Activation Trees(Cont...)

- We therefore can represent the activations of procedures during the running of an entire program by a tree, called an activation tree
- Each node corresponds to one activation and the root is the activation of the “main” procedure that initiates execution of the program
- At a node for an activation of procedure p the children correspond to activations of the procedures called by this activation of p
- We show these activations in the order that they are called from left to right
- Notice that one child must finish before the activation to its right can begin

Example 7.2

- One possible activation tree that completes the sequence of calls and returns suggested is shown in figure
- Functions are represented by the first letters of their names
- Remember that this tree is only one possibility since the arguments of subsequent calls and also the number of calls along any branch is influenced by the values returned by partition

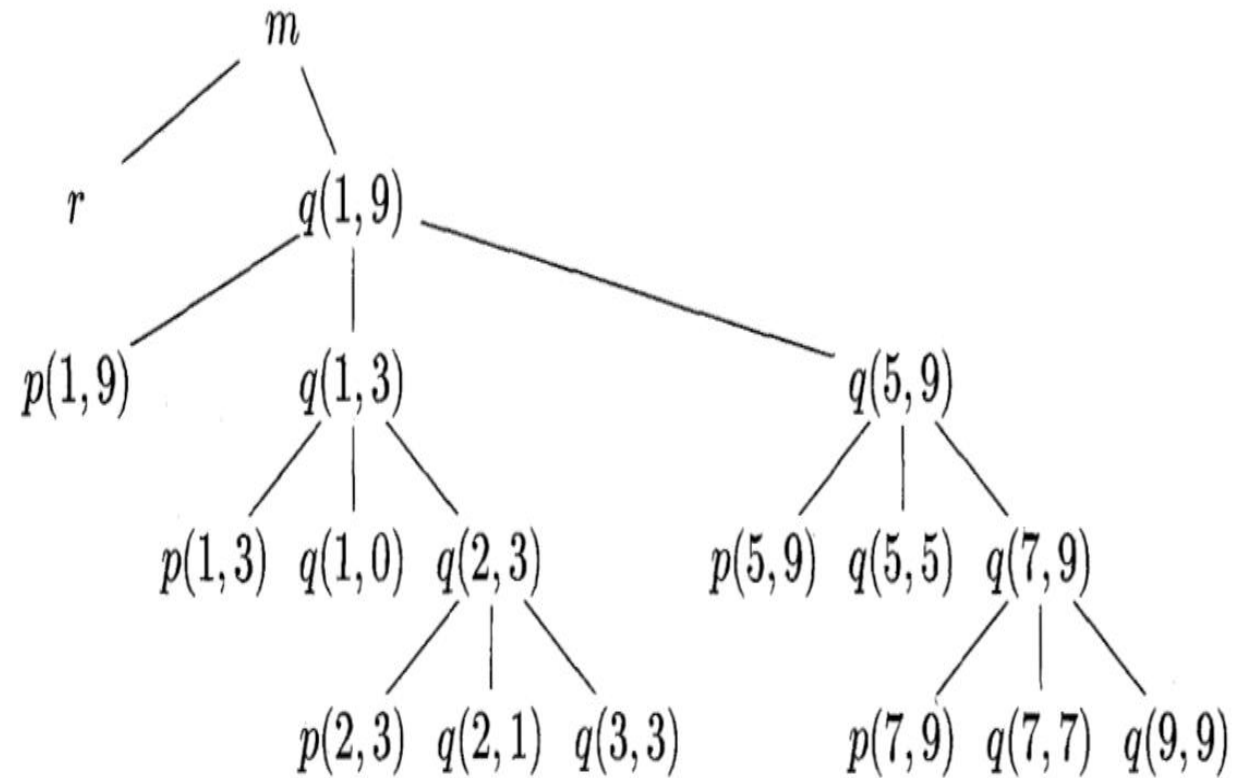


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Trees(Cont...)

- The sequence of procedure calls corresponds to a preorder traversal of the activation tree
- The sequence of returns corresponds to a postorder traversal of the activation tree

Activation Trees(Cont...)

- Suppose that control lies within a particular activation of some procedure, corresponding to a node N of the activation tree; then the activations that are currently open (live) are those that correspond to node N and its ancestors; the order in which these activations were called is the order in which they appear along the path to N starting at the root and they will return in the reverse of that order

Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the control stack
- Each live activation has an activation record(sometimes called a frame) on the control stack with the root of the activation tree at the bottom and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides
- The latter activation has its record at the top of the stack

Example 7.3

- If control is currently in the activation $q(2,3)$ of the tree, then the activation record for $q(2,3)$ is at the top of the control stack
- Just below is the activation record for $q(1,3)$, the parent of $q(2,3)$ in the tree
- Below that is the activation record $q(1,9)$, and at the bottom is the activation record for m , the main function and root of the activation tree

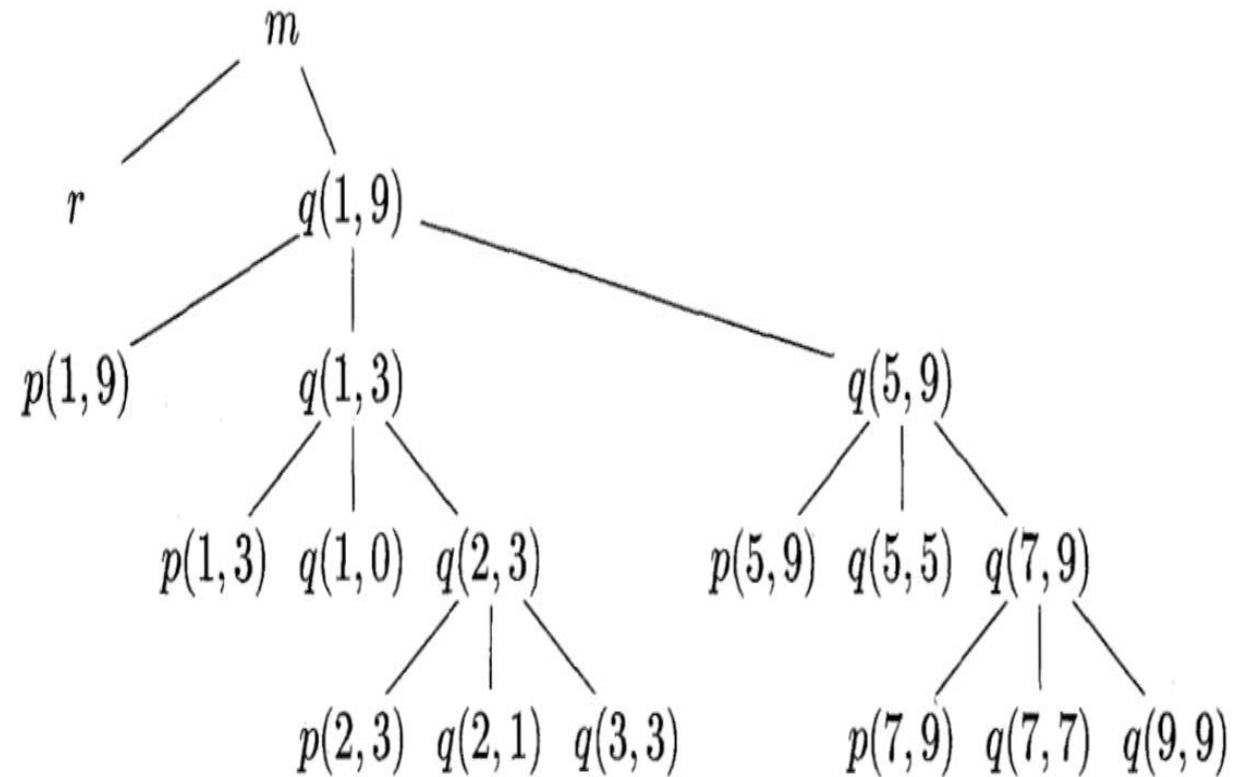


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Contents of an Activation Record

- The contents of activation records vary with the language being implemented
- Here is a list of the kinds of data that might appear in an activation record
- See figure for a summary and possible order for these elements.

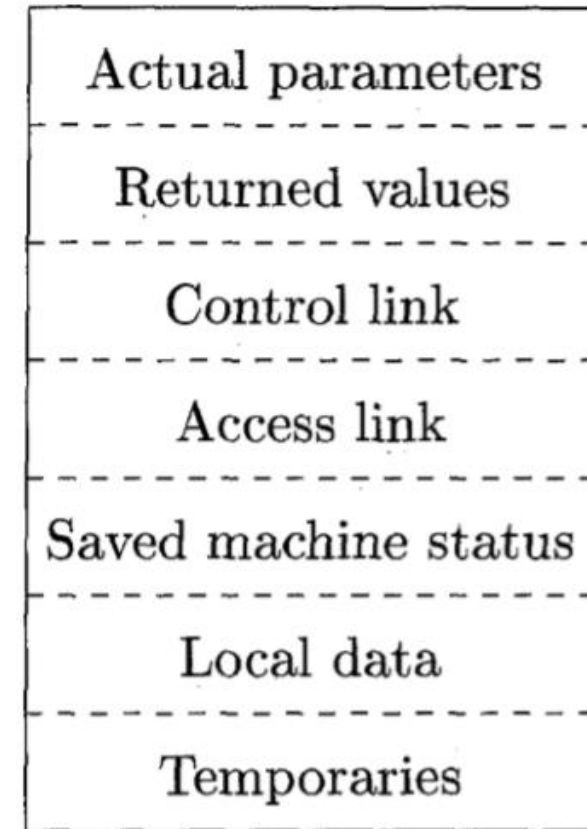


Figure 7.5: A general activation record

Contents of an Activation Record(Cont...)

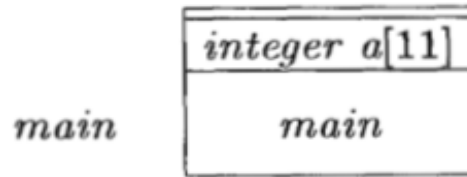
- **Temporary values** such as those arising from the evaluation of expressions in cases where those temporaries cannot be held in registers
- **Local data** belonging to the procedure whose activation record this is
- **A saved machine status** with information about the state of the machine just before the call to the procedure; this information typically includes the return address (value of the program counter to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs

Contents of an Activation Record(Cont...)

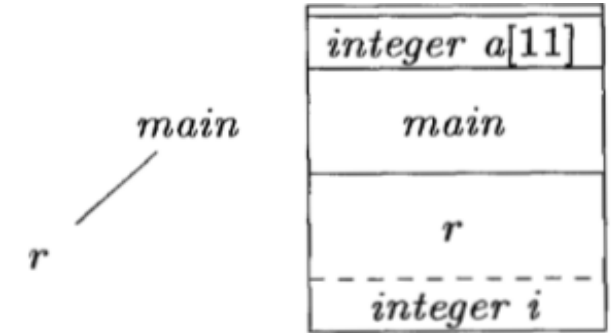
- An **access link** may be needed to locate data needed by the called procedure but found elsewhere e.g. in another activation record
- A **control link** pointing to the activation record of the caller
- Space for the **return value** of the called function, if any; again not all called procedures return a value and if one does we may prefer to place that value in a register for efficiency
- The **actual parameters** used by the calling procedure; commonly, these values are not placed in the activation record but rather in registers when possible for greater efficiency; however, we show a space for them to be completely general

Example 7.4

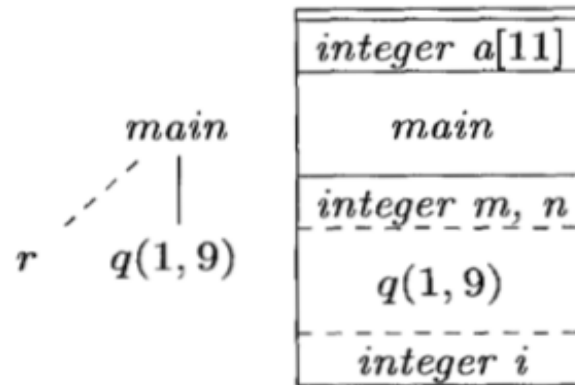
- Figure shows snapshots of the run-time stack as control flows through the activation tree
- Dashed lines in the partial trees go to activations that have ended



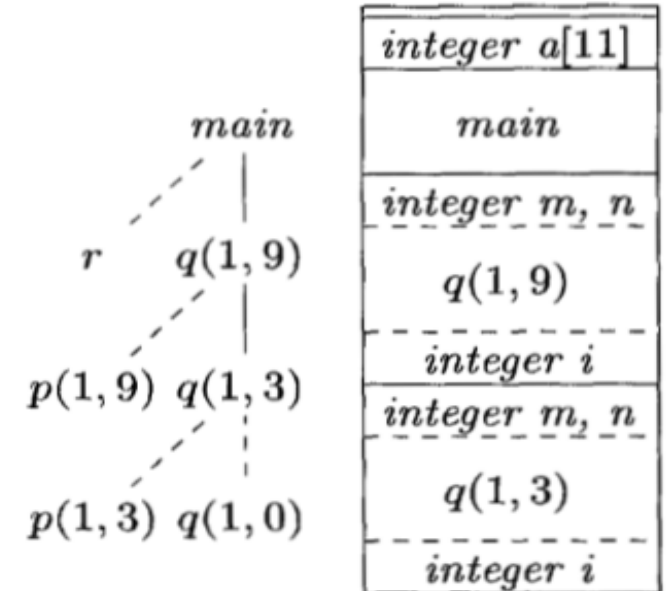
(a) Frame for *main*



(b) *r* is activated



(c) *r* has been popped and *q(1,9)* pushed

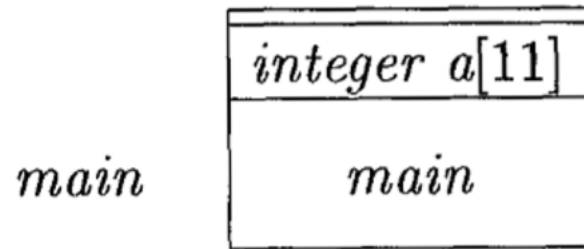


(d) Control returns to *q(1,3)*

Figure 7.6: Downward-growing stack of activation records

Example 7.4(Cont...)

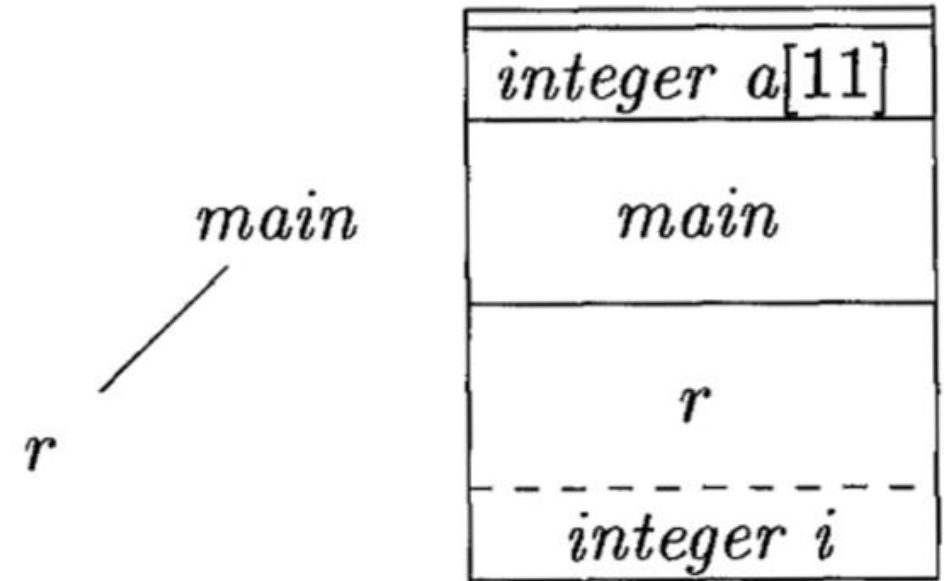
- Since array *a* is global, space is allocated for it before execution begins with an activation of procedure *main* as shown in (a)



(a) Frame for *main*

Example 7.4(Cont...)

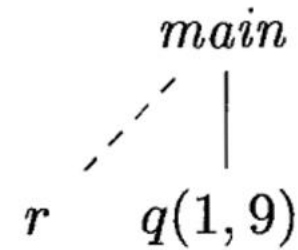
- When control reaches the first call in the body of main procedure *r* is activated and its activation record is pushed onto the stack (b)
- The activation record for *r* contains space for local variable *i*
- The top of stack is at the bottom of diagrams.



(b) *r* is activated

Example 7.4(Cont...)

- When control returns from this activation its record is popped leaving just the record for main on the stack
- Control then reaches the call to *q* (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack as in (c)
- The activation record for *q* contains space for the parameters *m* and *n* and the local variable *i* following the general layout

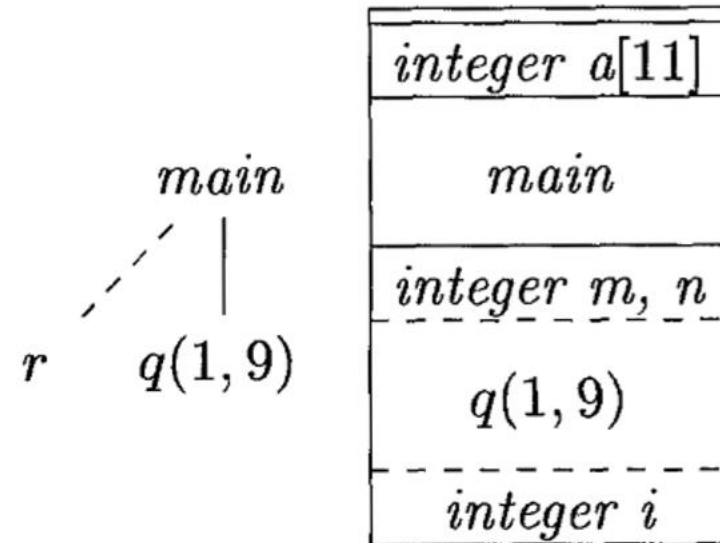


<i>integer a[11]</i>
<i>main</i>
<i>integer m, n</i>
<i>q(1, 9)</i>
<i>integer i</i>

(c) *r* has been popped and *q(1, 9)* pushed

Example 7.4(Cont...)

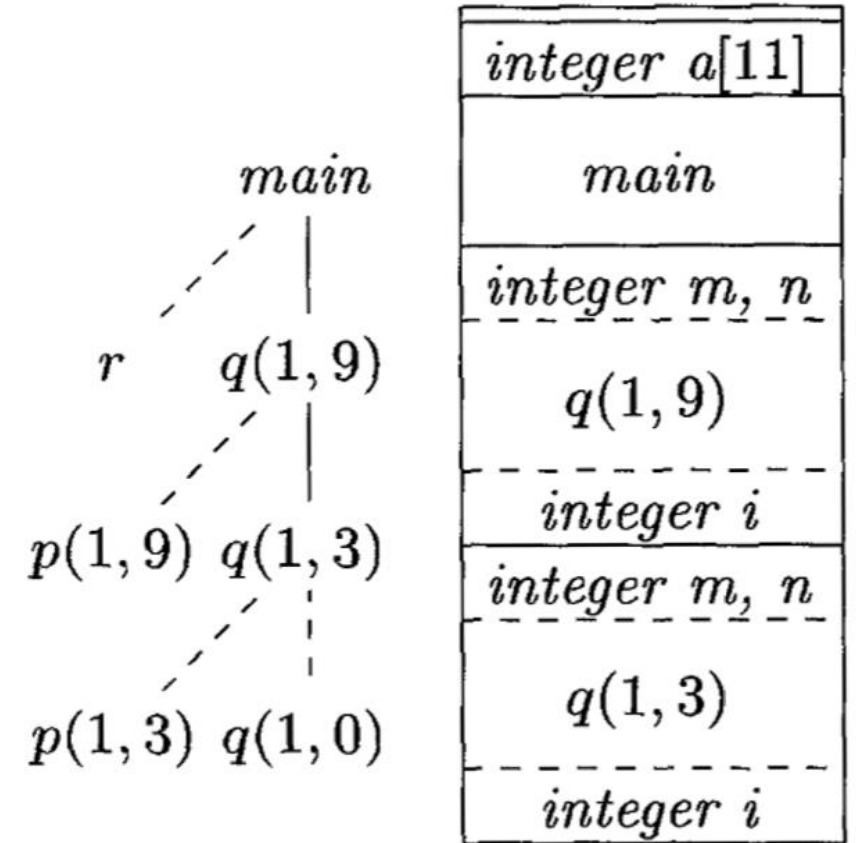
- Notice that space once used by the call of r is reused on the stack
- No trace of data local to r will be available to $q(1,9)$
- When $q(1,9)$ returns the stack again has only the activation record for main



(c) r has been popped and $q(1,9)$ pushed

Example 7.4(Cont...)

- Several activations occur between the last two snapshots
- A recursive call to $q(1,3)$ was made
- Activations $p(1,3)$ and $q(1,0)$ have begun and ended during the lifetime of $q(1,3)$ leaving the activation record for $q(1,3)$ on top of (d)
- Notice that when a procedure is recursive it is normal to have several of its activation records on the stack at the same time



(d) Control returns to $q(1,3)$

*Thank
you*

