# Storage organization:

UNIT- IV

Run time Environment

* From compiler writer perspective, executing program runs in its own logical address space in which each program value has location
* Management and organization of logical address space is shared between compiler, operating system and target machine. Os maps logical address to physical address.
* Run time representation of object program in logical address space consist of data and program area.

|  |
| --- |
| Code |
| Static |
| heap |
| Free memory |
| Stack |

* Run time storage comes in block of memory (continuous bytes).byte is 8 bits and 4 bytes form a machine word multi byte objects are stored in consecutive bytes.
* Storage layout for data objects is strongly influenced by addressing constraints of target machine.
* Size of generated target code is fixed at compile time. So compiler can place target code in statically determined area code usually in low end of memory.
* Global constraints and information supported to garbage collection may be known at compile time, and these are placed in statically determined area static.
* Addresses of code and static area objects are compiled in to target code.
* To maximum utilization of space at runtime, other 2 areas are stack and heap. these are dynamic, their size can change as program executes.

## Static versus dynamic storage allocation:

* Layout and allocation as data to memory location in runtime environment are key issues in storage management.
* Storage allocation decision is static, if it can make by compiler looking only at text of program.
* Storage allocation decision is dynamic, if it can decide only while program is running.
* Many compilers use some combination of following strategies for dynamic storage allocation.

## Stack storage:

Names local to procedure are allocated space on stack it supports normal call/return policy for procedure.

## Heap storage:

Data that may out live (not local) call to procedure are allocated space on heap of reusable storage.

* To support heap, garbage collection enables run time system to detect useless data elements and reuse their storage.

# Static allocation of space:

* Almost all compilers that use procedures functions or methods manage runtime environments as stack.
* Each time procedure called, space for local variable is pushed on stack, and when procedure terminates that space is popped off stack.

## Activation trees:

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

**Example:** program that reads 9 integers into array ‘a’ and sort them using recursive quick sort algorithm.

int a[11];

void readArray()

{

/\*read 9 integers into a[1] a[9]\*/

Int i;

\_ \_ \_ \_ \_ \_

}

int partition(int m, int n)

{

/\*pick separator value v1 and partitions a[m\_ \_ \_ \_ \_n] to a[m p-1] are less than v

and a[p+1 n] are greater than v \*/

\_ \_ \_ \_ \_ \_

}

Void Quick sort(int m, int n)

{

Int i;

if(n>m)

{

i = partition(m, n); Quick sort(m, i-1); Quick sort(i+1, n);

}

}

main(){

read Array(); a[0] = -9999;

a[10] = 9999;

Quick sort(1, 9);

}

* Main function has three tasks. It calls readArray and then calls Quick sort on array and call to partition will split array base on separator.
* In the above, procedure activations are nested in time. If activation of procedure p call procedure q then activation of q ends before activation of p end. There are 3 common cases.

1. Activation of q terminates normally then control resumes after point of p at which call to q mode.
2. Activation of q either directly or indirectly aborts in that case p ends simultaneously with q.
3. Activation of q terminates with exception that q can’t handle but p may handle it. In such situation q has terminated while p will continues. If p not handles it then p also terminated.

* Activation of procedure during running of program can be viewed by tree are called activation tree. Each node corresponds to one activation.
* Possible activation of above program is enter main()

enter readarray() leave readarray() enter Quick sort(i,9)

enter partition(1,9) leave partition(1,9) enter Quick sort(1,3)

\_ \_ \_ \_

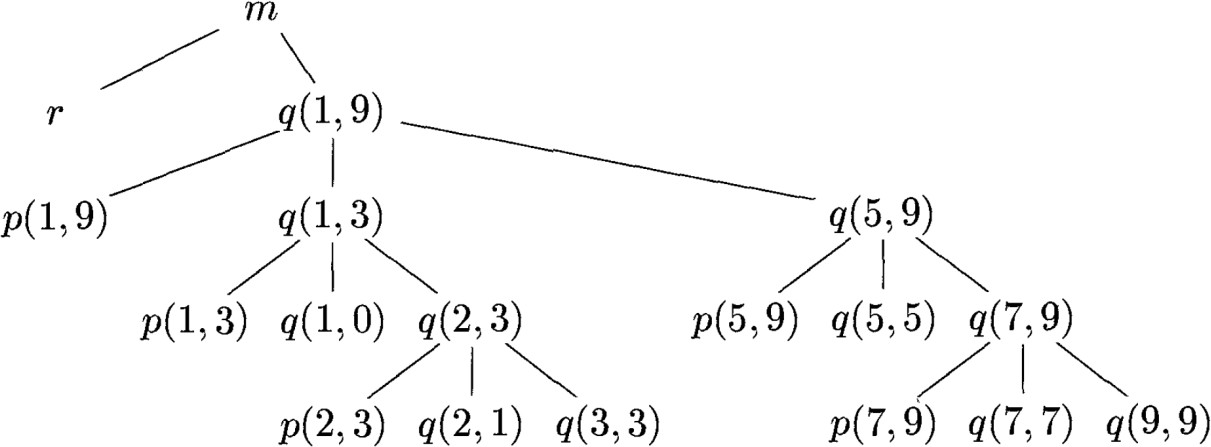
leave Quick sort(1,3) enter Quick sort(5,9)

\_ \_ \_ \_

leave Quick sort(5,9) leave Quick sort(i,9)

leave main()

* Activation tree is



* Stack will enabled several relationships between tree and behavior of program.

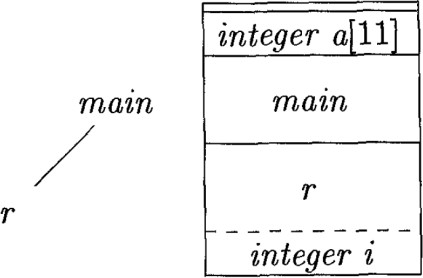
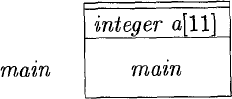
1. Sequence of procedure calls corresponds to pre order traversal of tree.
2. Sequence of returns corresponds to post order traversal of tree.
3. Order in which activations called is order they appear along path to N and returns should be reverse order.

## Activation records:

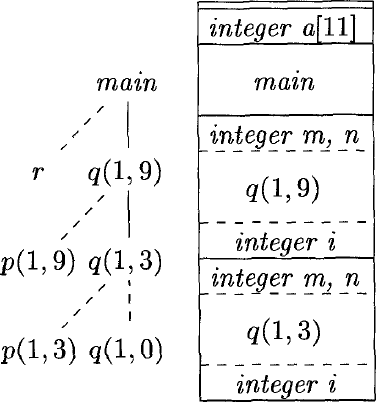
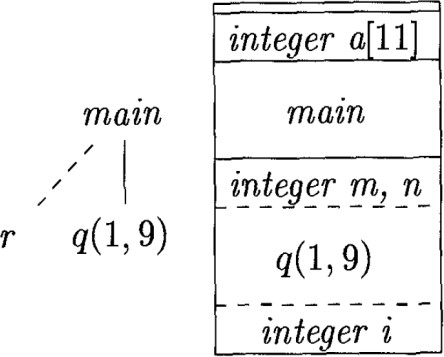
* Procedure calls and returns are usually managed by runtime stack called control stack.
* Each live activation has activation record on control stack with root of activation tree at bottom.
* We draw control stacks with bottom of stack than top. Elements appear at lowest are closest to top.
* Here is a list of kinds of data that might appear in an activation record. These are vary with language implemented it.

1. Temporary values, such as those rising from evaluation of expression.
2. Local data belongs to procedure which activation record this is.
3. Saved machine states, information about state of machine before call procedure. Information includes return address and registers used by calling procedure.
4. Access link may be needed to locate data needed by called procedure in other activation record.
5. Control link, pointing to activation record of caller.
6. Space for return value of called function if any again called procedure return a value.
7. Actual parameters used by calling procedure. Commonly these are not place in record. Some cases it is possible.

|  |
| --- |
| Actual Parameters |
| Returned Values |
| Control Link |
| Access Link |
| Saved Machine Status |
| Local Data |
| Temporaries |

**Example:** Below picture shows run time, stacks as control slow through the activation tree here a is global .so, space is allocated for it before execution starts.

* 1. Frame for main (b) r is activated

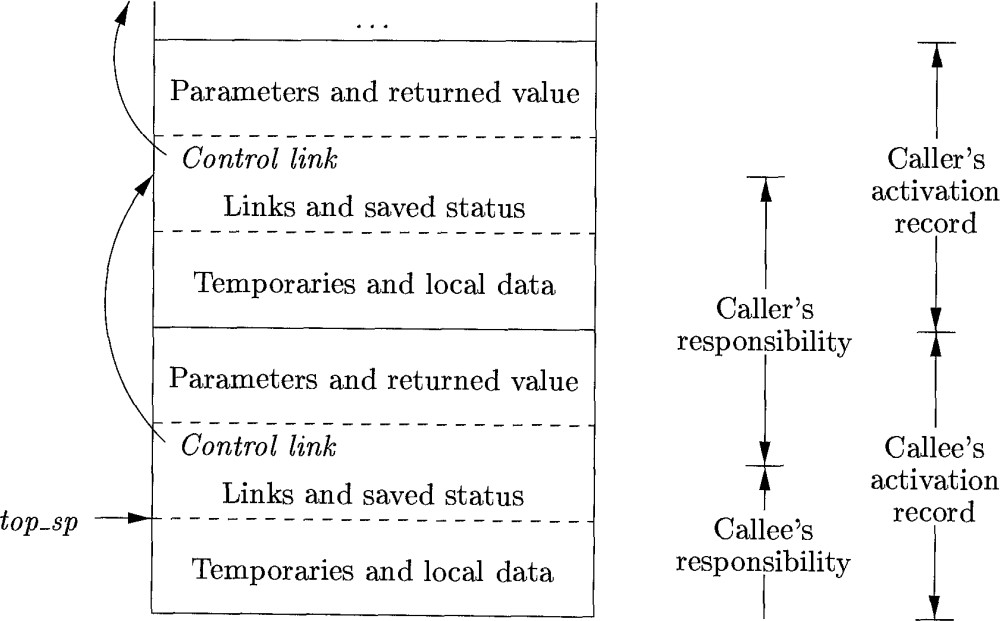


(a) r is popped and q(1,9) is pushed (b) control returns to q(1,3)

* Control reaches first call in body main, procedure is activated and its activation record pushed onto control returns from this activation, its record is popped.
* Control then reaches call to q and activation record for this call is placed on top of stack. When q(1,q) returns ,stack again has only activation record for main.
* Same procedure will be done for remaining nodes of activation tree. When a procedure is recursive it is normal to several of its activation records on stack at same time.

## Calling sequences:

* Procedure calls are implemented by calling sequences, which consists of code that allocates activation record on stack.
* Return sequences is similar code to restore state of machine.
* The code in calling sequence is divide between calling procedure (caller) and procedure it calls (caller) and there is no extract division of runtime tasks between them.
* When designing calling sequences and layout of activation records, the principles are helpful.

1. Values communicated between caller and callee are generally placed at beginning of callee’s activation record. So they are close to caller’s activation record.
2. Fixed length items are generally placed in middle. Those items include control link, access link and machine status fields.
3. Items whose size may not be known early are placed at the end of activation record. Those items include local variables and temporary variables.
4. We must locate top of stack pointer common approach is locate at end of fixed length fields in activation record.

* How caller and callee might cooperate in managing stack should be shown in the above.
* The calling sequence and it’s division between caller and callee are as follows:

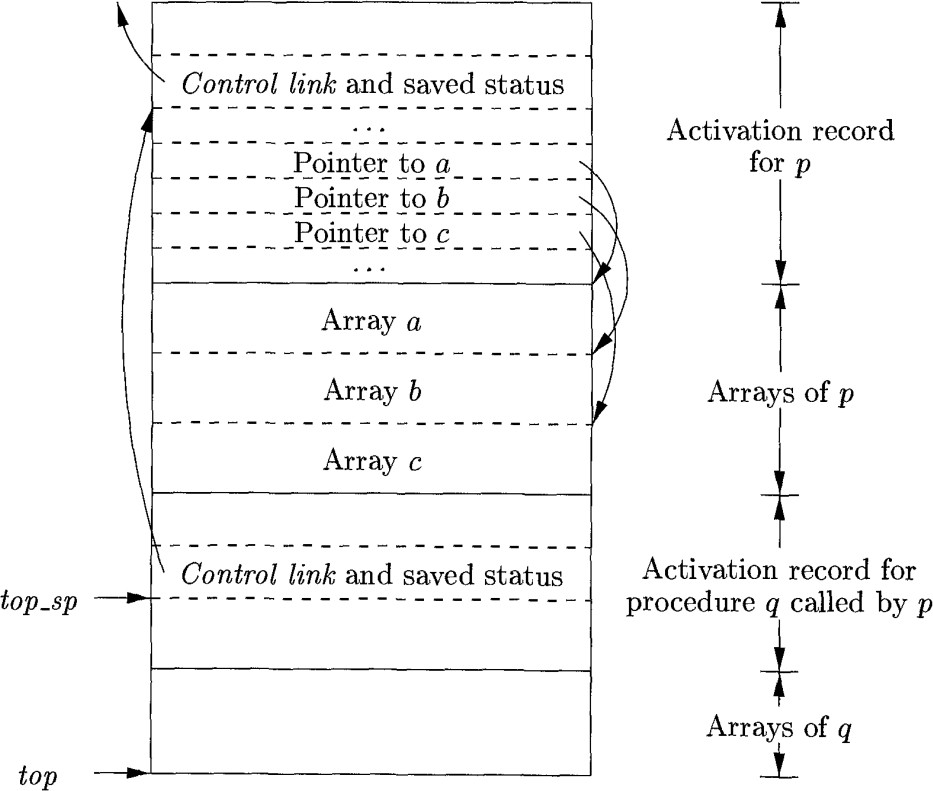
1. Caller evaluates actual parameters.
2. Caller stores return address and old value of top\_sp into activation record of callee.
3. Callee saves register value and other status information.
4. Callee initializes it’s local data and begin execution.

* Suitable, corresponding return sequence is

1. Callee place return value next to parameters.
2. Callee restores top-sp and other registers and branches to return address, those place in caller status field.
3. top\_sp has decremented, caller knows where return value is relative to current value of top\_sp. then caller use this value.

* Above calling and return sequences allow number of arguments of called procedure to vary from call to call.

## Variable length data on stack:

* Allocation of space for objects whose size not known at compile time are allocated on stack.
* Reason for prefer placing objects on stack is to avoid expenses of garbage collecting.
* Common strategy for allocating variable length arrays is shown below. Same schema works for objects of any type.
* In the above diagram, procedure p has 3 local arrays whose sizes can’t be determined at compile time.
* Storage allocation of these arrays is not part of activation record for p, although it does appear on stack.
* Only a pointer to beginning of each array appears in activation record itself. When p executing array elements accessed through these pointers.
* Activation record of q, called by p begins after array of p. Any variable length array of q is after activation record of q.
* Access to data on stack is through two pointers top and top\_sp.

Top marks actual top of stack. it points at which next activation record begins. Top\_sp is used to find local, fixed length fields of top activation record

* Code to reposition top and top-sp can generated at compile time,in terms of size that is known at runtime
* When q returns, top\_sp can restore from save control link in activation record for q.new value of top is top\_sp minus the length of fields in q’s activation record.

# Access to Nonlocal data on stack:

* Here we consider how procedures access their data that not belongs to procedure.
* Access becomes more complicated in languages where procedures can be declared inside other procedures.
* ML language permits both nested function declarations and functions as class objects;i.e function takes function as arguments and return functions as values.

## Data accessing without nested procedures

* In c family of languages, all variables defined either with in single function or outside any function.
* It is impossible to declare one procedure whose scope is entirely within another procedure.
* For language that do not allow nested procedure declarations, allocation of storage for variables and access to those are simple:

1. Global variables are allocated static store locations of those variables are fixed and known at compile time.
2. Any other name must be local to activation at top of the stack .We may access these variables through top-sp pointer of the stack.

## Issues with nested procedure:

* Access become far more complicated when languages allows procedure declaration to be nested.
* Reason is that knowing at compile time that the declaration of p is immediately nested with in q doesn't tell us relative positions of their activation records at runtime.
* Finding declaration that applies to nonlocal name x in nested procedure p is static decision. It can be done by static scope rule for blocks.
* X is declared in enclosing procedure q finding relevant activation of q from activation of p is dynamic. It requires additional runtime information about activation.
* One possible solution to this problem is to use access links.

## Language with Nested Procedure Declaration:

* C family of languages and many other familiar languages do not support nested procedure.
* History of nested Procedure in language is long, one such language in ML(Meta Language).

1. ML is functional language, means variable, once declared and initialized are not changed.
2. Variables are designed and they have their unchangeable values initialized by statement of form.

val (name) =<expression>

1. Functions are defined using syntax:

fun (name) (<arguments>) =<body>

1. For function bodies we use let-statements of form. let <list of definitions > in <statements> end

* Scope of each such definition consists of all following definitions up to in and all statements up to end.
* Function definitions can be nested. Example body of p contains let statement include definition of other function q.

## Nesting depth:

* Give nesting depth 1to procedures that are not nesting with in any other Procedure.
* All C functions are at nesting depth 1. If Procedure p is defined with in Procedure at nesting i then nesting depth of p is i+1

## Example:

fun sort(inputFile, outputFile) = let

val a = array( 11,0 );

fun readArray(inputFile) = ;

------- a ;

fun exchange( i, j ) =

------- a ;

fun quicksort( m, n ) let

val v = ;

fun partition( y, z ) =

------- a -------- v -------- exchange --------

in

end

in

------- a -------- v -------- partition quicksort

------- a -------- readArray quicksort

End ;

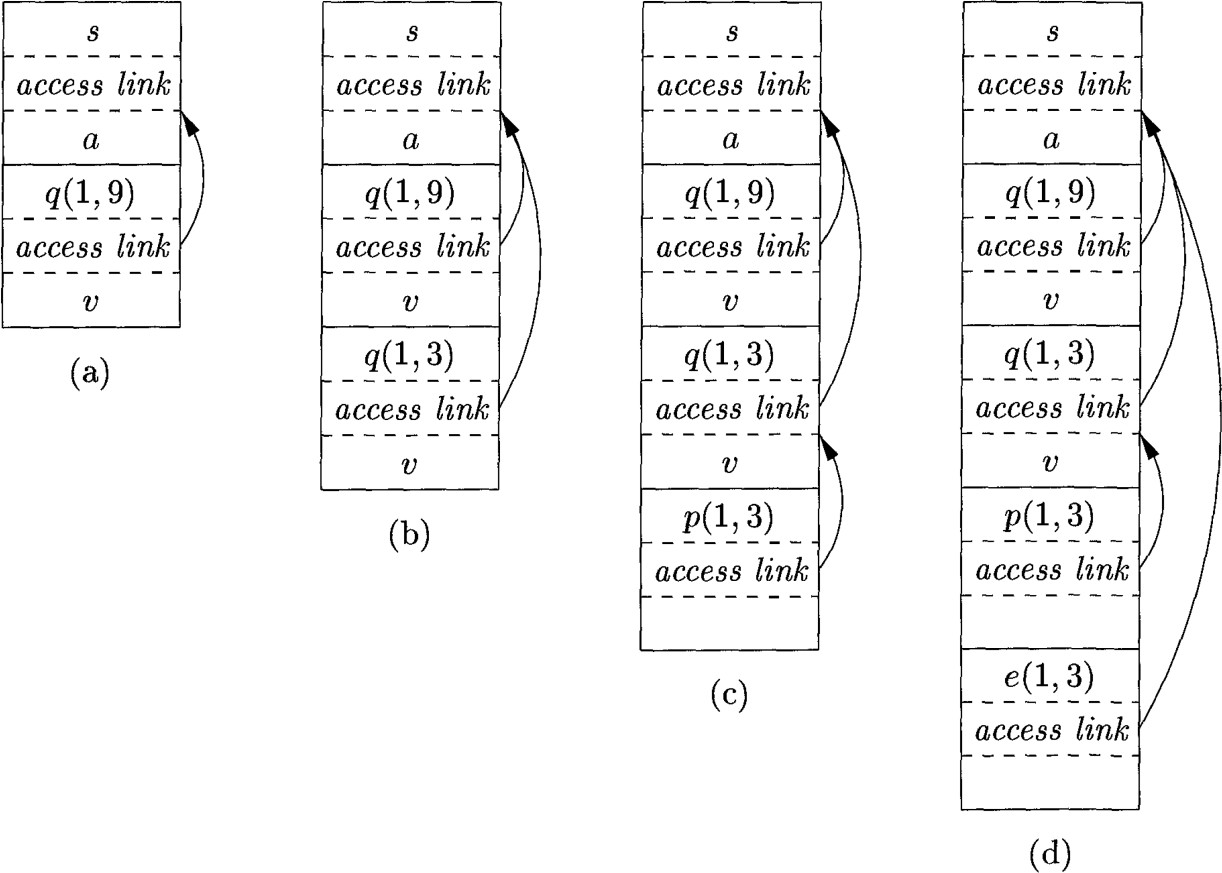
* In above ML program, only function at nesting depth 1 is outermost function, sort which reads array as a 9 integers and sort them using quicksort algorithm.
* In an array a, first argument says array 11 elements and second arguments says array a holds value 0.
* Sort consist several functions: readArray, exchange and quick sort. readArray and exchange each access array a.
* Each of the above function immediately within the function is at nesting depth 1, their nesting depths are all 2.
* In quick sort, local value v, the pivot for the partition is declared. These partitions uses pivot value v. Partition is immediately with in function at nesting depth 2,It is depth 3.
* Quick sort accesses variable a and v, the function partition and itself recursively. Outer function accesses a and call the two Procedure readArray and quicksort.

## Access links:

* Direct implementation of normal static scope rule for nested functions is obtained by access links of each activation record.
* If Procedure p is nested immediately within the Procedure q then access links in any activation of p points to most recent activation of q.
* Access links form a chain from activation record at top of the stack to sequence of activation at lower nesting depth.

**Example:** Sequence of stacks will show result of function sort. Here function name represent first letters, show some data and access links for each activation.

We see situation after sort called read Array to load input in to array a then called Quicksort(1,9).The access link from Quicksort(1,9) points to activation record sort.l

In successive steps we see recursive call to Quicksort(1,3) followed by partition which calls exchange. Quicksort(1,3) points to sort for same array of Quicksort (1,9).

Exchange access link by passes activation records for Quicksort and partition , since exchange is nested immediately with in sort.

## Manipulating Access links:

* Simple case occurs when Procedure calls to particular Procedure whose name is given explicitly in Procedure call.
* Harder case is when call is to Procedure parameter in that ,Procedure call is not known until runtime and nested depth is differ for different executions of call.
* What should happen when Procedure q calls Procedure p explicitly? There are 2 cases:

1. Procedure p is at higher nesting depth than q. p must be devised immediately within q then access link from p must lead to q.
2. Procedure p is at lower or equal nesting depth than q in p=q case links for p and q are same .if p is lesser than access link from q leads to p.

## Access links for procedure parameters:

* When procedure p is passed to another procedure q as parameter and q then calls its parameter .It is possible that q does not know context in which p appears in program.
* It is impossible for q to know how to set access link for p solution to this problem is when procedures are used as parameters then caller needs to pass access link with parameter.

**Example:** Below function a has functions b and c nested within It. Function b has function valued parameter f. Function c defines with init a function d and c then calls b with actual parameter d.

fun a(x) = let

fun b(f) =

----- f ---;

fun c(y) = let

fun d(z) = ----

in

--- b(d) ---

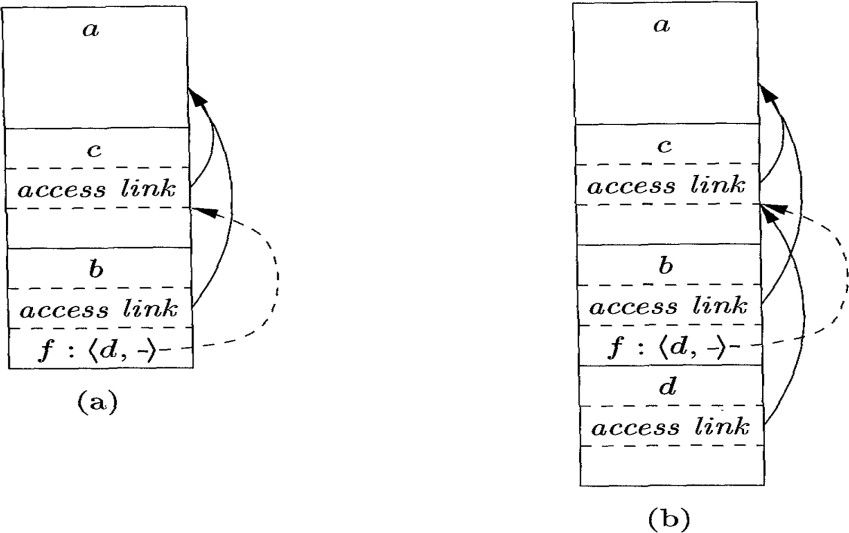
end

in

--- c(1) ---

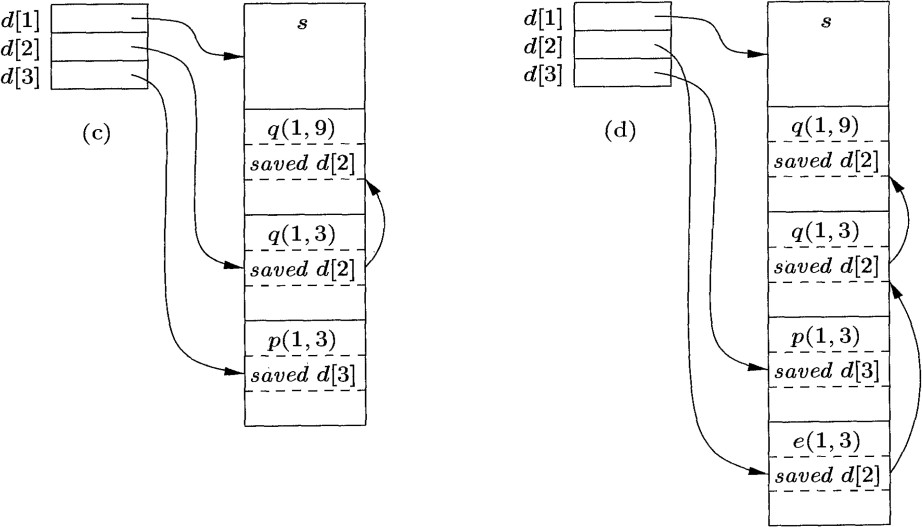
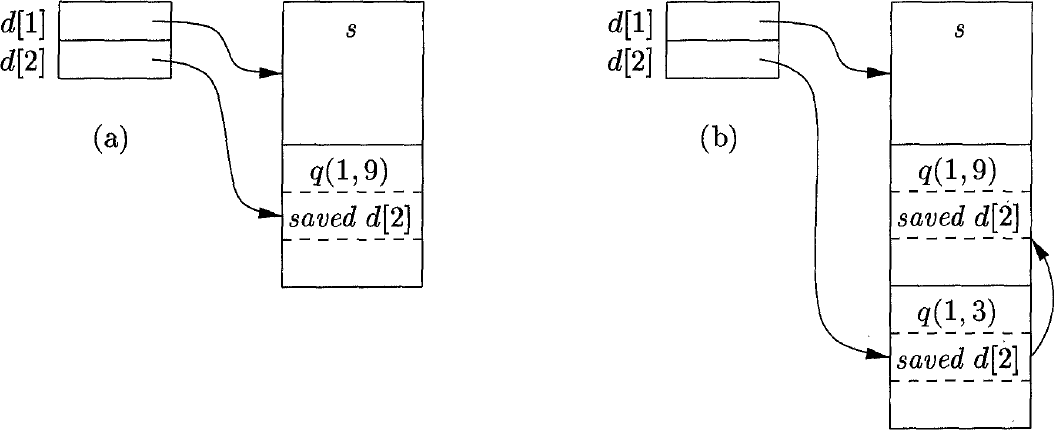
end ;

* First, a calls c. We place activation record for c above a on stack .Access link for c points to record for c points record for a in c calls b(d).
* Within this activation record actual parameter d and It is access link together form. Formal parameter f in activation record for b.
* In effect of calling d, activation record of d appears on the stack. Access link to place in activation record is found in value for parameter in f ; link is to activation record for c, c immediately surrounds d



## Displays:

* One problem with the access link approach to non local data is that if the nesting depth gets large, we may have to follow long chain of links to reach data.
* Efficient implementation uses auxiliary array d called display, which consists of one pointer for each nesting depth.
* d[i] is pointer to highest activation record on stack for any procedure at nesting depth i.



* Advantage of display is if procedure P is executing and it needs to access element x belongs to procedure q, we need to look only in d[i] where i is nested depth of q.
* In order to maintain display correctly we need to save previous values of display entries in new activation records.