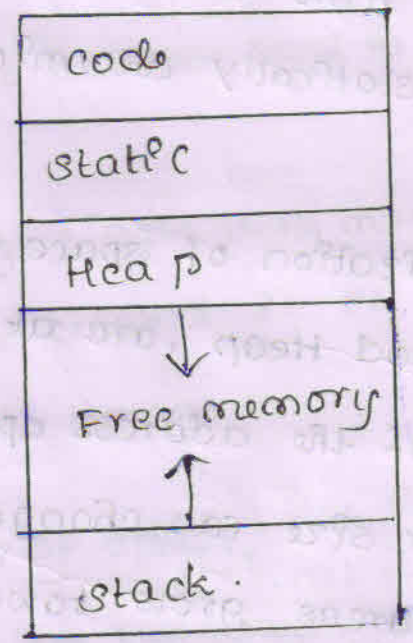
UNIT-7

RUN-TIME ENVIRONMENTS

1. Storage organization

* The executing target program runs in its own logical address space in which each program value has a location. The management & organization of this logical address space is shared between the compiler, operating system & target machine.

* The typical subdivision of run time memory into code and data areas is,



* Assume the run time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. A byte is 8 bits & 4 byte form a

word.

* The amount of storage needed for a name is determined by its type. An elementary data types such as char, int or float can be stored in an integral number of bytes. Storage for an aggregate type, such as an array or structure must be large enough to hold all its components.

* The size of the generated 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 constraints & 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.

* To maximize the utilization of space at run time, the other 2 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 structure called activation records that get generated during procedure calls.

Static Versus Dynamic Storage Allocation

* The layout and allocation of data to memory locations in the runtime environment are key issues in storage management.

* The 2 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.

* A decision is dynamic if it can be decided only while the program is running. Many compiler uses some combination of the following 2 strategies for dynamic storage allocation.

(i) Stack storage : Names local to a procedure are allocated space on stack.

(ii) Heap storage : Data that may outlive the call to the procedure that created it is usually allocated on a heap of reusable storage.

7.2 Stack Allocation of space

* The compilers for languages uses procedures, functions or methods as unit of user defined actions manage at least part of their runtime memory as a stack. Each time a procedure is called, a space for its local variables is pushed onto a stack, & when the procedure terminates

that space is popped off the stack.

Activation Trees

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

Ex: Consider quick sort program

```
int a[11]

void readArray ()
{
  int i;
...
}
int partition (int m, int n) {
/* picks a separator value v & partition 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);
}
```

* On the above ex, as is true in general, procedure activations are nested in time. If an activation of procedures p calls procedure q, then that activation of q must end before the activation of p can end. There are 3 common cases

(i) the activation of q terminates normally. Then in essentially any languages, control resumes just after the point of p at which the call to q was made

(ii) 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

(iii) The activation of q terminates because of an exception that q can't handle.

* We 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 procedures that initiates the execution of the program.

* The possible activations for the quicksort program is,

enter main()

enter readArray()

leave readArray()

enter quicksort(1,9)

enter partition(1,9)

leave partition(1,9)

enter quicksort(1,3)

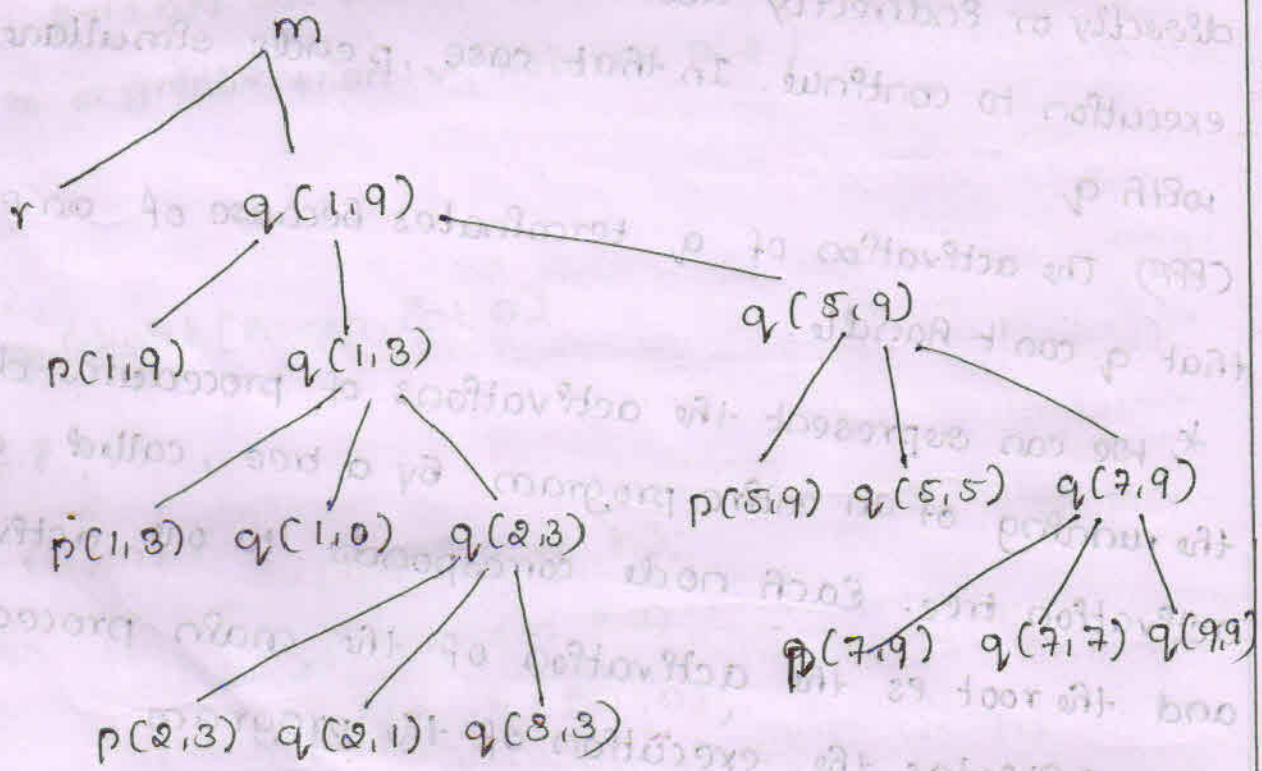
leave quicksort(1,3)

enter quicksort(5,9)

leave quicksort(1,9)

leave main()

* The activation tree is,



* The use of a run time stack is enabled by several useful relationships b/w the activation tree & the behaviour

of the program.

(b) the sequence of procedure calls corresponds to a preorder traversal of the activation tree.

Ex: the sequence of procedure call for quicksort is,

m, r, q(1,9), p(1,9) q(1,3) p(1,3) q(1,0) q(2,3)
p(2,3) q(2,1) q(3,3) q(5,9) p(5,9) q(5,5) q(7,9)
p(7,9) q(7,7) q(9,9)

(c) the sequence of returns corresponds to a postorder traversal of the activation tree.

Ex: the sequence of return for quicksort is

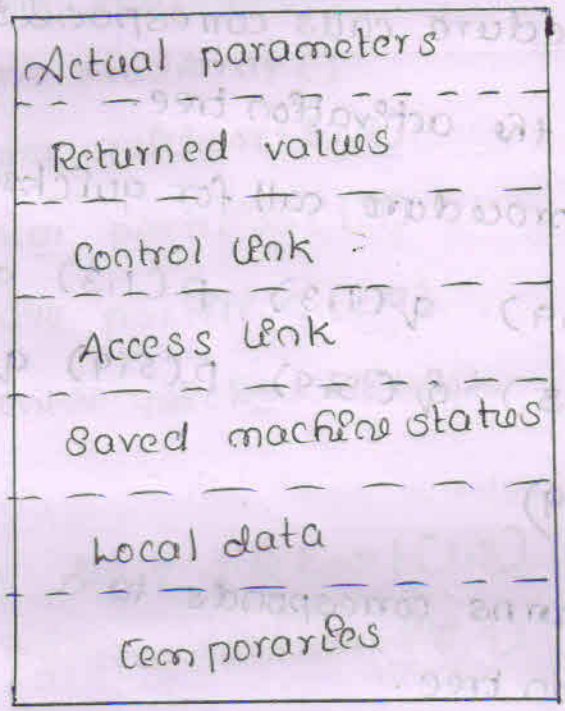
r p(1,9) p(1,3) q(1,0) p(2,3) q(2,1) q(3,3)
q(2,3) q(1,3) p(5,9) q(5,5) p(7,9) q(7,7)
q(9,9) q(7,9) q(5,9) q(1,9) m

(d) 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 are those that correspond to node N & its ancestors.

Activation records

* Activation record is a sequence of the procedure calls & the return values which is managed by the run time stack.

* The components of activation records are,



(i) Temporaries: temporary values, such as those arising from the evaluation of expressions, in case where those temporaries cannot be held in registers

(ii) Local data: It belonging to the procedure whose activation record this is.

(iii) Saved machine status: It has the information about the state of the machine just before the call to the procedure. This information typically includes the return address & the contents of registers that were used by the calling procedure & that must be restored when the return occurs.

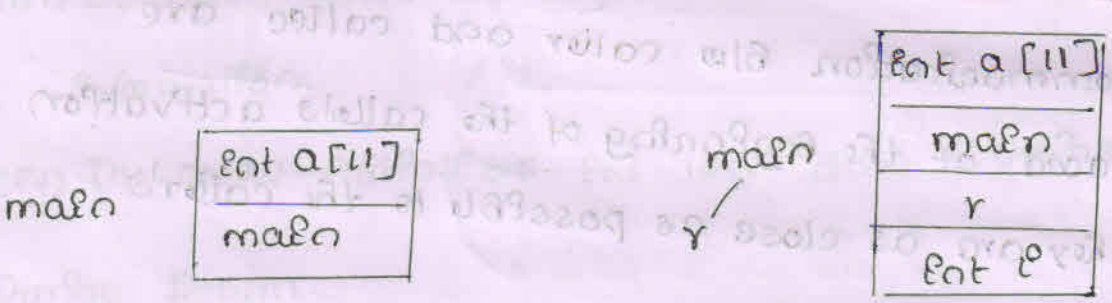
(iv) An "access link" may needed to local data needed by the called procedure but found else where eg, in another activation record

(v) Control links: It points to the activation record of the caller

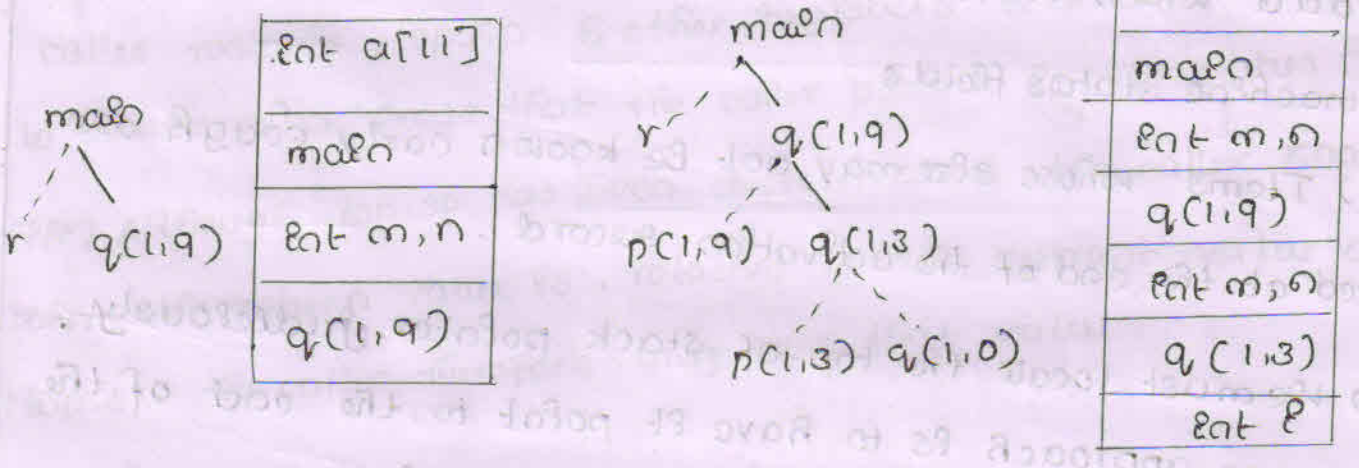
(vii) Returned values: Space for the return value of the called function, if any. Again, not all called procedures return a value & if one does, we may refer to place that value in a register for efficiency.

(viii) Actual parameter: It is used by the calling procedure.

The activation record for quick sort program is,



(a) Frame for main (b) r is activated



(c) r has been popped & q(1,9) pushed (d) Control returns to q(1,3)

Calling Sequence

* Procedures calls are implemented by calling sequences, which consists of code that allocates an activation record on the stack & enters information into its fields.

* The code in a calling sequence is often divided between

the calling procedure and the procedure it calls. There is no exact division of run-time tasks b/w caller & callee, the source language, the target machine, and the operating system impose requirements that may favor one solution over another.

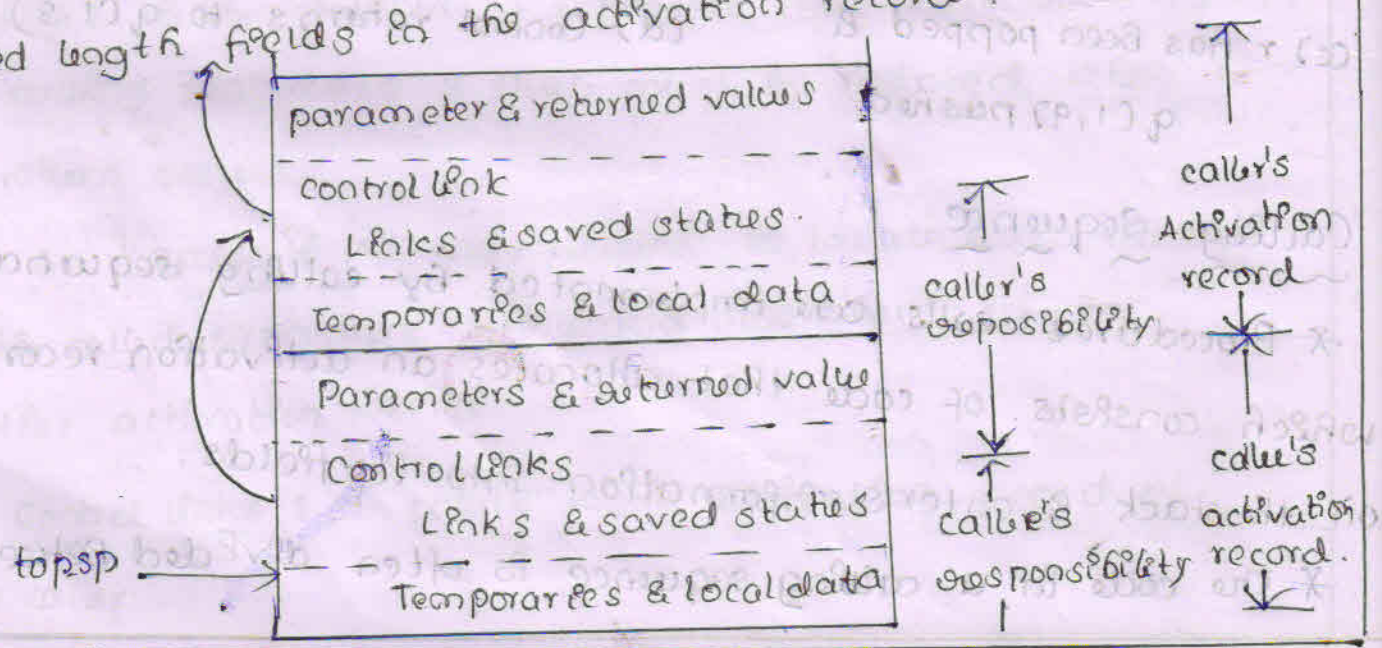
* the principles of calling sequence are,

(i) values communication b/w caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

(ii) Fixed length items are generally placed in the middle which includes the control link, the access link & the machine states fields.

(iii) Items whose size may not be known early enough are placed at the end of the activation record.

(iv) We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of the fixed length fields in the activation record.



*Calling sequence is,

During calling

(i) Caller will evaluates the actual parameters

(ii) The caller stores a return address and the old value of top-sp into the callee's activation record.

(iii) The callee saves the register values and other states information.

(iv) The callee initializes its local data & begins execution

During Return.

(i) The callee places the return value next to the parameter

(ii) Using information in the machine states field, the callee restores, top-sp & other registers & then branches to the return address that the caller placed in the states field.

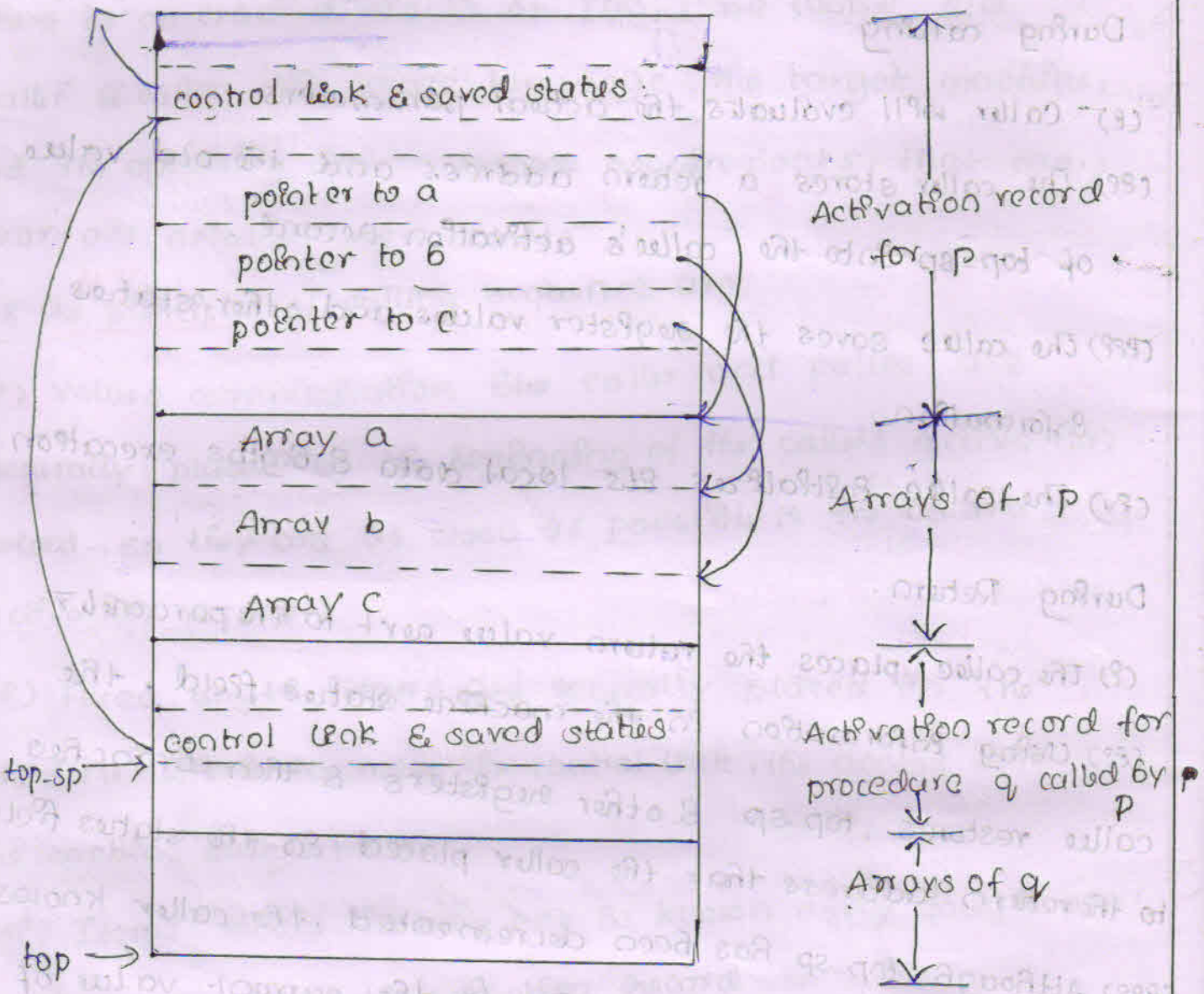
(iii) Although top-sp has been decremented, the caller knows where the return value is, relative to the current value of top-sp the caller therefore may use that value.

Variable length data on the stack.

* The run time memory management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure & thus may be allocated on the stack. But it is placed in heap in modern languages.

* However, it is also possible to allocate objects, arrays, or other structures of unknown size on the stack.

* A common strategy for allocating variable length array is,



* Access to the data on the stack is through 2 pointers, top & top-sp. Here top marks the actual top of stack, it points to the position at which the next activation record will begin. The second top-sp is used to find local, fixed length fields of the top activation record.

* The code to reposition top & top-sp can be generated at compile time, in terms of sizes that will become known at run time. When q returns top-sp can be restored from the saved control link in the activation record for q.

The new value of top is top-sp minus the length of the machine-status, control & access link, return value & parameter fields in q's activation record. This length is known at compile time to the callee, although it may depend on the caller if the no. of parameters can vary across calls to q.

7.3 Access to Nonlocal data on the stack.

Here we consider how procedures access their data. Especially important is the mechanism for finding data used within a procedure p but that does not belong to p. Access becomes more complicated in language where procedure can be declared inside other procedure.

Data Access without Nested Procedures.

* It is impossible to declare one procedure whose scope is entirely within another procedure. Rather, a global variable v has a scope consisting of all the functions that follow the declaration of v, except where there is a local definition of the identifier v.

* For languages that do not allow nested procedure declarations, allocation of storage for variables & access to those variables is simple.

(a) Global variables are allocated static storage. The location of these variable remain fixed & are known at compile time. So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.

(iii) Any other name must be local to the activation at the top of the stack. We may access these variables through the top_sp pointer of the stack.

* An important benefit of static allocation for globals is that declared procedures may be passed as parameters or returned as results, with no substantial change in the data access strategy.

Issues with nested procedures.

* Access becomes far more complicated when a language allows procedure declarations to be nested and also uses the normal static scoping rule, that is a procedure can access variables of the procedures ~~for a~~ whose declarations surround its own declaration. Here the relative position of the activation record is not known at run time.

* Finding the declaration that applies to a nonlocal name x in a nested procedure p is a static decision, it can be done by an extension of the static scope rule for blocks. Suppose x is declared in the enclosing procedure q. Finding the relevant activation of q from an activation of p is a dynamic decision, it requires additional run time information about activations. One possible solution to this problem is to use "access links".

A language with nested procedure declarations

C & other many languages do not support nested procedure.

Languages with nested procedure, one of most influential is ML, and it in this a language whose syntax and semantics we shall borrow.

(i) ML is functional language, meaning that variables, once declared & initialized, are not changed. There are only a few exceptions, such as the array, whose elements can be changed by special function calls.

(ii) Variables are defined, & have their unchangeable values initialized by a statement of the form

val <name> = <expression>

(iii) Functions are defined using the syntax

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

(iv) For function bodies we shall use let-statement of the form

let <list of definitions> in <statements> end.

Nesting Depth.

Let us give nesting depth 1 to procedure that are not nested within any other procedure

Ex: All C functions are at nesting depth 1. However, if a procedure p is defined immediately within a procedure at nesting depth i, then give p the nesting depth i+1

Consider quick sort in ML

(1) fun sort (inputFile, outputFile) =

```
let ...
(2) val a = array(11,0);
(3) fun readArray(inputFile) = ...;
(4)     ...a...;
(5) fun exchange(i,j) =
(6)     ...a...;
(7) fun quicksort(m,n) =
    let
(8)     val v = ...;
(9)     fun partition(y,z) =
(10)        ...a...v...exchange...
        in
(11)        ...a...v...partition...quicksort
        end
    in
(12)    ...a...readArray...quicksort...
    end;
```

* The only function at nesting depth 1 is the outermost fn, sort.

* At line (2) array is declared, the first argument specifies the no. of element and second argument specifies the initial value of the elements. This choice of initial value lets ML compiler deduce the array type, so we never have to declare the type

* Within a sort function readArray, exchange & quicksort are used at line (3) (5) (7) respectively. They all will be in depth 2.

* The partition function in line (9) will be in depth 3 as it is inside the f^n which is at nesting depth 2. 9

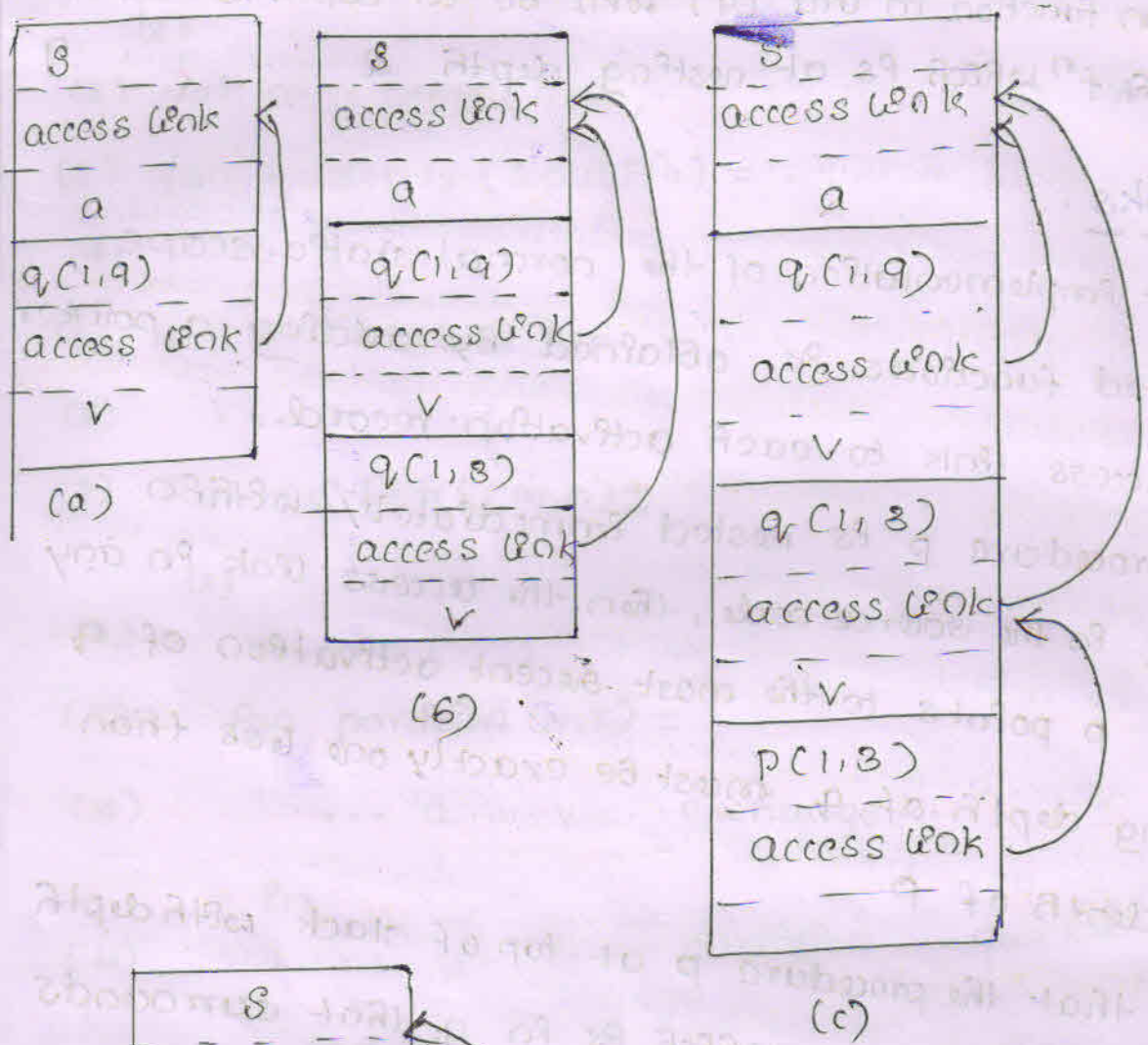
Access Links.

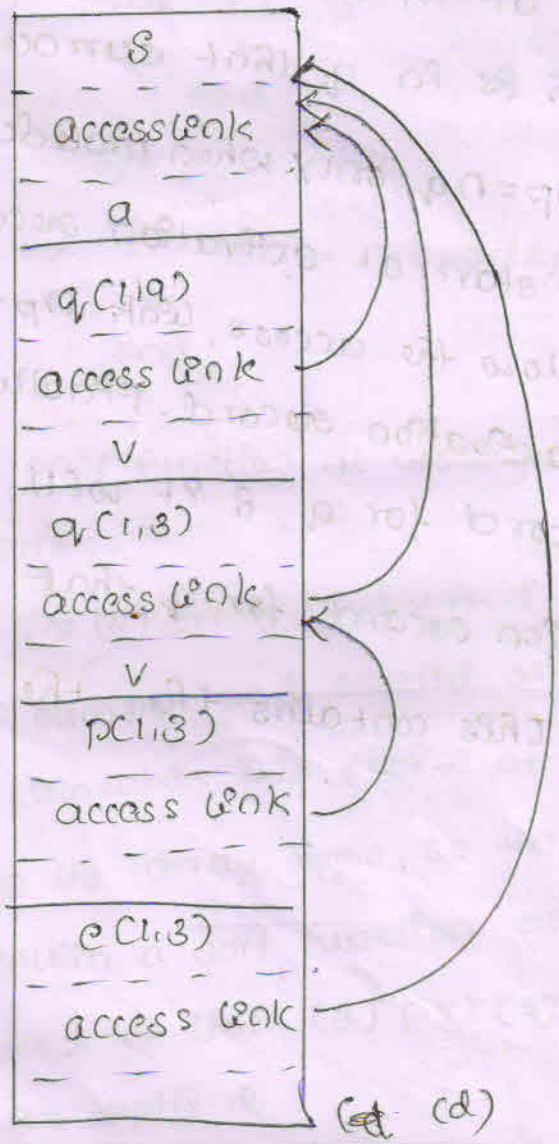
* A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the access link to each activation record.

* If the procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.

* The nesting depth of q must be exactly one less than the nesting depth of p.

* Suppose that the procedure p at top of stack with depth n_p & p needs to access x which is in q that surrounds p & has depth n_q. ($n_p \geq n_q$, $n_p = n_q$ only when procedure p & q are same). To find x, we start at activation record for p at the top of the stack & follow the access link $n_p - n_q$ times from activation record to activation record. Finally, we wind up at an activation record for q & it will always be the most recent activation record for q that currently appears on the stack. This contains the the (closest x


S
access lock
a
q(1,9)
access lock
V
(a)
S
access lock
a
q(1,9)
access lock
V
q(1,3)
access lock
V
(b)
S
access lock
a
q(1,9)
access lock
V
q(1,3)
access lock
V
p(1,3)
access lock
(c)


S
access lock
a
q(1,9)
access lock
V
q(1,3)
access lock
V
p(1,3)
access lock
c(1,3)
access lock
(d)

* In the above example quicksort (1,9) points to the activation record for sort not because sort called it but because sort is the most closely nested fn for it.

* In (d) the access link for exchange bypasses the activation records for quicksort & partition, since exchange is nested immediately within sort.

Access links for procedure parameters

* When a procedure p is passed to another procedure q as a parameter & q then calls it parameter, sometimes it may not know the context in which p appears in the program. Then it is impossible for q to know how to set the access link for p.

* This problem can be solved as, when procedures are used as parameters, the caller needs to pass the proper access link for the parameter.

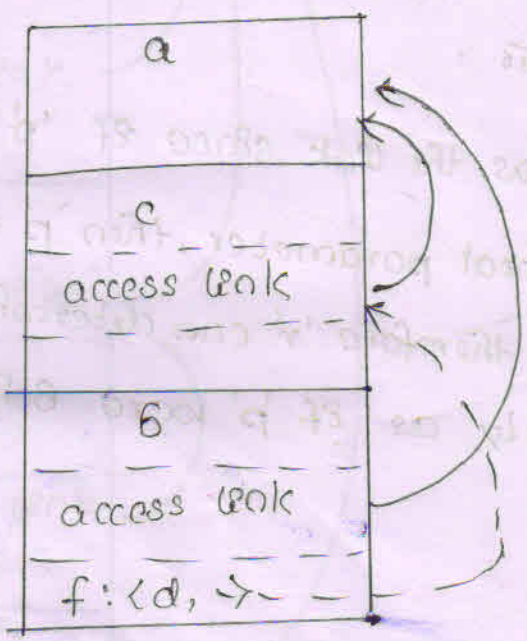
* The caller always knows the link, since if 'p' is passed by procedure 'r' as an actual parameter, then 'p' must be a name accessible to 'r', & therefore 'r' can determine the access link for 'p' exactly as if 'p' were being called by 'r' directly.

Consider the ex

```
fun a(x) =
    let
```

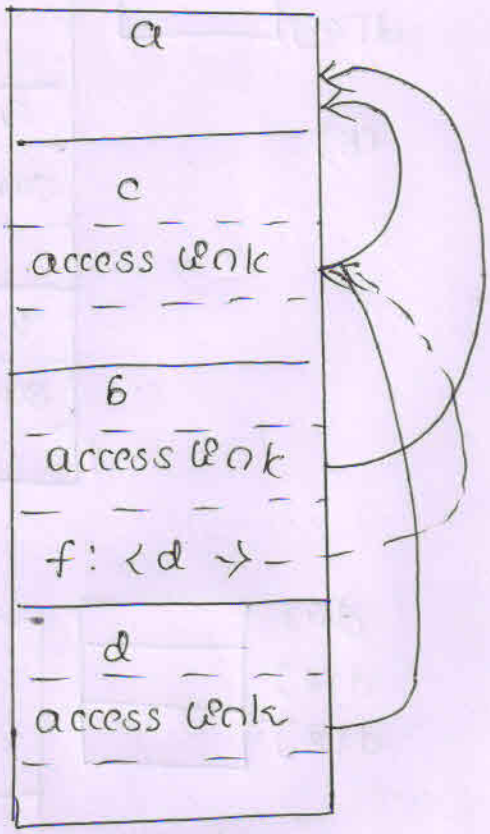
```
fun b(f) =
  ... f ... ;

fun c(y) =
  let
    fun d(z) = ...
    in
      ... b(d) ...
    end
in
  ... c(1) ...
end;
```

* First 'a' calls 'c' so we place an activation record for c above that for a on the stack. The access link for c points to the record for a, since c is defined immediately within a. Then c calls b(d). The calling sequence is,



* In b at some point, it uses its parameter f, which

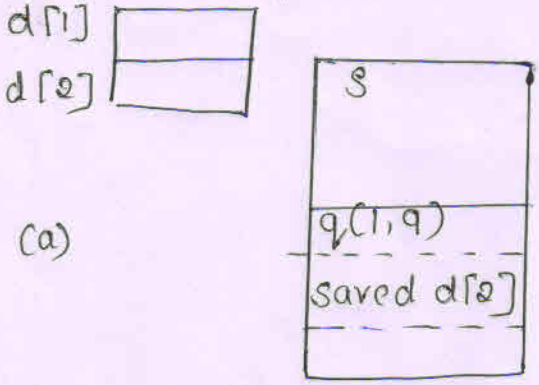
has the effect of calling d. An activation record for d appears on the stack as,



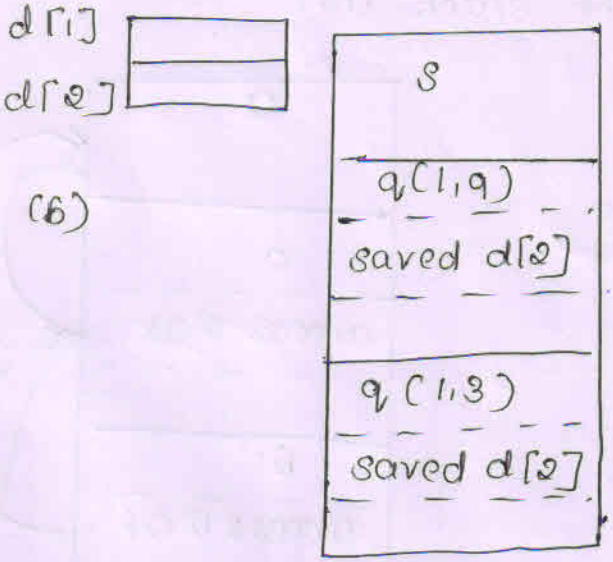
* The proper access link to place in this activation record is found in the value for parameter f, the link is to the activation record for c, since c immediately surrounds the definition of d.

Displays

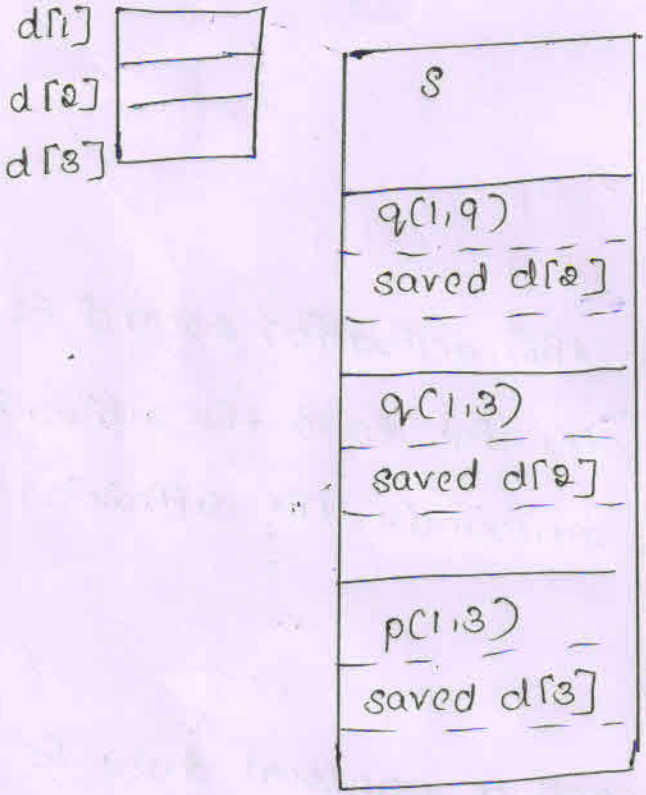
* The problem with access link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need.
* A more efficient auxiliary array d, called the display, which consists of one pointer for each nesting depth.
* Here, at all times, d[i] is a pointer to the highest activation record on the stack for any procedure at nesting depth i.

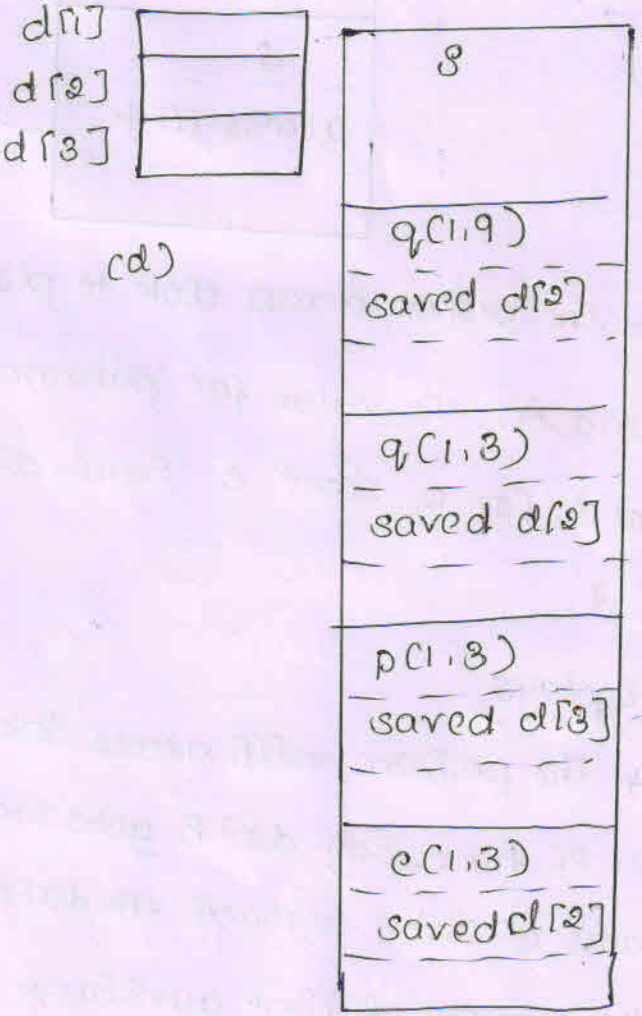


(a)



(b)





(d)

* the display d, with d[1] holding a pointer to the activation record for sort, the highest activation record for a fn at nesting depth 1. Also d[2] holds a pointer to the activation

record for exchange, the highest record at depth 2 & d[3] points to partition, the highest record at depth 3.

Advantage: If procedure p is executing & it needs to access element x belonging to some procedure q, we need to look only to d[e], where e is the nesting depth of q; we follow the pointer d[e] to the activation record for q, where x is found at a known offset. The compiler knows what-e is, so it can generate code to access x using d[e] & the offset x from the top of the activation record for q. Thus, the code never needs to follow a long chain of access links.

* Inorder to maintain the display correctly, we need to save previous values of display entries in new activation records.

7.4 Heap Management.

The heap is the portion of the store that is used for data that lives indefinitely or until the program explicitly deletes it. While local variables typically becomes inaccessible when their procedures end.

The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. It performs 2 basic fn

(i) Allocation: when a program requests memory for a variable or object the memory manager produces a chunk of contiguous heap memory of the requested size

If possible, it satisfies an allocation request using free space in the heap; if no chunks of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the OS. If space is exhausted, the memory manager passes that information back to the appln pgm.

(ii) Deallocation: The memory manager return deallocated space to the pool of freespace, so it can reuse the space to satisfy other allocation requests.

Memory management would be easier if

(a) all allocation requests were for chunks of the same size

(b) storage were released predictably, say, first allocated, first deallocated.

There are some languages such as Lisp. for which condition (a) holds: pure Lisp uses only one data element - a 2 pointer cell - from which all the data structure are built. Condition (b) also holds in some situation. Inmost lang. neither (a) nor (b) Rold in general.

* The properties of memory managers are

(i) Space Efficiency: A memory manager should minimize the total heap space needed by a program. Doing so allows larger programs to run in a fixed virtual address space. Space efficiency is achieved by minimizing fragmentation

(ii) Program efficiency: A memory manager should make good

use of the memory subsystem to allow programs to run faster. By attention to the placement of objects in memory, the memory manager can make better use of space and hopefully, make the program run faster.

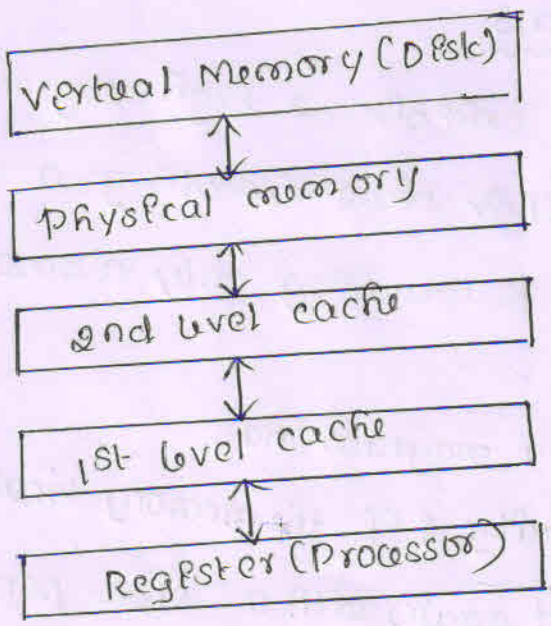
(iii) Low overhead : Allocation & deallocation are frequent operations & it is imp that these operations be as efficient as possible.

The memory Hierarchy of a Computer

* The efficiency of a program is determined not just by no. of instructions executed, but also by how long it takes to execute each of these instructions.

* The time taken to execute an instruction varies since the time taken to access different parts of memory can vary from nanoseconds to milliseconds.

* The large variance in memory access times is due to the fundamental limitation in hardware technology; we can build small & fast storage or large & slow storage but not storage that is both large & fast. Therefore practically all modern comp uters arrange their storage as a memory hierarchy



* A processor has a small no. of registers, whose contents are under s/w control.

* It has one or more levels of cache, usually made out of static RAM.

* the next level of hierarchy is the physical memory, made out of hundreds of megabytes or GB of dynamic RAM.

* the virtual memory is implemented by gigabytes of disks.

* Upon a memory access, the machine first looks for the data in the closest storage and, if the data is not there, looks for the next level & so on.

* Caches are managed exclusively in h/w in order to keep up with the relatively fast RAM access times. Because disks are relatively slow, the virtual memory is managed by the OS, with assistance of a h/w structure known as the translation lookaside buffer.

Locality in Programs

* Most programs exhibit a high degree of locality; that is, they spend most of their time executing a relatively small fraction of the code & touching only a small fraction of the data.

* We can say that a program has

↳ temporal locality: if the memory locations it accesses are likely to be accessed again with a short period of time

↳ spatial locality : If memory locations close to the location accessed are likely also to be accessed within a short period of time.

* Programs spend 90% of their time executing 10% of the code because.

↳ Programs often ~~obtai~~ contain many instructions that are never executed.

↳ Only a small fraction of the code that could be invoked is actually executed in a typical run of the program

↳ The typical program spends most of its time executing innermost loops & tight recursive cycles in a program.

* Locality allows us to take advantage of the memory hierarchy of a modern computer by placing the most common instruct^n & data in the fast-but-small storage, while leaving the rest in the slow but-large storage.

Optimization using the Memory Hierarchy.

* The policy of keeping the most recently used instructions in the cache tends to work well.

* When a new instruction is executed, there is a high probability that the next instruction also will be executed. This is an example of spatial locality.

* One technique to improve the spatial locality of instructions is to have the compiler place basic blocks that are likely

to follow each other contiguously.

* We can also improve the temporal & spatial locality of data accessed in a program by changing the data layout or the order of the computation.

Reducing Fragmentation.

* At the beginning of program execution, the heap is one contiguous unit of free space.

* As the program allocates & deallocates memory, this space is broken up into free & used chunks of memory & the free chunks need not reside in a contiguous area of the heap. We refer to the free chunks of memory as holes

* With each allocation request, the memory manager must place the requested chunk of memory into a large enough hole.

* Unless a hole of exactly the right size is found, we need to split some hole, creating a yet smaller hole.

* With each deallocation request, the freed chunks of memory are added back to the pool of free space.

* If we are not careful, the free memory may end up getting fragmented, consisting of large no. of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request even though there may be sufficient aggregate free space.

Best-fit & Next-fit object placement.

* We can reduce fragmentation by controlling how the memory manager places new objects on the heap.

* Best-fit: It is a good strategy for minimizing fragmentation for real life pgms is to allocate the requested memory in the smallest available hole that is large ~~or~~ enough.

* First fit: where an object is placed in first hole in which it fits, takes less time to place objects.

* To implement best fit placement more efficiently, we can separate free space chunks into bins, according to their sizes.

* Binning makes it easy to find the best fit chunk

↳ If, as for small sizes requested from the Lea memory manager, there is a bin for chunks of that size only, we may take any chunk from that bin.

↳ For sizes that do not have a privat bin, we find the one bin that is allowed to include chunks of the desired size.

↳ However, it may be that the target bin is empty, or all chunks in that bin are too small to satisfy the request for space. In that case, we simply repeat the search, using the bin for the next larger size.

* While best fit placement tends to improve space utilization, it may not be the best in terms of spatial locality.

Managing & coalescing Free space.

* When an object deallocates the memory manager must make its chunk free, so it can be allocated again.
* It is also possible to combine that chunk with adjacent chunks of the heap to form a larger chunk.
* The advantage of doing so, is, we can use larger chunk to do the work of small chunks of equal total size.
* If we keep a bin for chunks of one fixed size, then we may prefer not to coalesce adjacent blocks of ~~one~~ that size ~~as m~~ into a chuck of double the size.
* A simpler allocation / deallocation scheme is to keep a bit-map, with one bit for each chunk in the bin. A 1 indicates the chunk is occupied; 0 indicates it is free.
* There are 2 data structures that are useful to support coalescing of adjacent free blocks.

(i) Boundary Tags: At both the low & high ends of each chunk, whether free or allocated, we keep vital information. At both ends, we keep a free/used bit that tells whether or not the block is currently allocated or available. Adjacent to each free/used bit is a count of the total no. of bytes in the chunk.

(ii) A Doubly linked, Embedded free List: The free chunks are also linked in a doubly linked list. The pointers for this list are within the blocks themselves, say adjacent to the

boundary tag at either end. Thus, no additional space is needed for the free list, although its existence does plan a lower bound on how small chunks can get, they must accommodate 2 boundary tags & 2 pointers, even if the object is a single byte.

Manual Deallocation Requests.

* Here the programmer must explicitly arrange for the deallocation of data.
* Any storage that may be referenced must not be deleted whereas any storage that will no longer be accessed should be deleted.

Problem with Manual Deallocation.

* Manual memory management is error prone. The common mistakes takes 2 forms
  (1) Failing ever to delete data that cannot be referenced is called a memory leak error.
  (2) Referencing deleted data is a dangling pointer dereference error.
* Memory leaks may slow down the execution of a program due to increased memory usage, they do not affect program correctness, as long as the machine doesn't run out of memory.
* For long-running programs, & especially if the leakage is slow nonstop programs like OS or server code, it is

critical that they not have leaks.

* Automatic garbage collection get rid of memory leaks by deallocating all the garbage.

* To dulit some storage & then try to refer to the data in the dellocated storage. Pointers to storage that has been deallocated are known as dangling pointers.

* We refer to any operation such as read, write or deallocate that follows a pointer & tries to use the object it points to as dereferencing the pointer.

* Reading through a dangling pointer may return an arbitrary value. Writing through a dangling pointer arbitary changes the value of the new variable.

* Unlike memory leaks, dereferencing a dangling pointer after the freed storage is reallocated almost always creates a program error that is hard to debug.

* A related form of programming error is to access an illegal address.

Ex: dereferencing null pointers & accessing an out of bounds array element.

Programming Conventions & Tools

* Some of the most popular conventions & tools are

(i) Object ownership: It is useful when an object's lifetime can be statically reasoned out. The idea is to associate an owner with each object at all times. The owner is a

pointer to that object, presumably belonging to some fn invocation.

(ii) Reference counting : It is useful when an object's lifetime needs to be determined dynamically. This idea is to associate a count with each dynamically allocated object. Whenever a reference to the object is created, we increment the reference count; whenever a reference is removed, we decrement the reference count.

(iii) Region-based allocation : It is useful for collection of objects whose lifetimes are tied to specific phases in a computation. When objects are created to be used only within some step of a computation, we can allocate all such objects in the same region. We then delete the entire region once that computation step completes.

7.5. Introduction to Garbage Collection.

Data that cannot be referenced is generally known as garbage. Many high-level programming language remove the burden of manual memory management from the programmer by offering automatic garbage collection, which deallocates unreachable data.

Design Goals for Garbage Collectors

* Garbage collection is the reclamation of chunks of storage

holding objects that can no longer be accessed by a program.

* A user program, which we shall refer to as the mutator, modifies the collection of objects in the heap. The mutator creates objects by acquiring space from the memory manger, & the mutator may introduce & drop references to existing objects. Objects become garbage when the mutator program cannot reach them.

* A Basic requirement: type safety.

* Not all languages are good candidates for automatic garbage collection. For a garbage collection to work, it must be able to tell whether any given data element or component of a data element is, or could be used as, a pointer to a chunk of allocated memory space.

* type-safe: A language in which the type of any data component can be determined. Ex: ML, Java.

* For some languages, type cannot be determined at compile time but can be determined at run time. This is called dynamically typed languages

* Unsafe: If a language is neither statically nor dynamically type safe, then is said to be unsafe.

Ex: C, C++.

* Performance metrics.

* Some of the performance metrics that must be considered

(i) Overall Execution Time: Garbage collection can be very slow. It is imp that it not significantly increase the total run time of an appln.

(ii) Space usage: It is important that garbage collection avoid fragmentation & make the best use of the available memory

(iii) Pause time: Simple garbage collectors are notorious for causing programs - the mutators - to pause suddenly for an extremely long time, as garbage collection kicks in without warning, thus, besides minimizing the overall execution time, it is desirable that max. pause time be minimized.

(iv) Program Locality: We cannot evaluate the speed of a garbage collector solely by its running time. The garbage collector controls the placement of data & thus influences the data locality of the mutator program. It can improve a mutator's temporal locality by freeing up space and reusing it, it can improve the mutator's special locality by relocating data used together in the same cache or pages

Reachability

* We refer to all the data that can be accessed directly by a program without having to dereference any pointer, as the root set.

* Reachability becomes a bit more complex when the program has been optimized by the compiler.

* A compiler may keep reference variables in registers. These references must also be considered part of the root set.

* Even though in a type safe language programmers do not get to manipulate memory address directly, a compiler often does so for the sake of speeding up the code.

* Here are some things an optimizing compiler can do to enable the garbage collector to find the correct root set

(i) The compiler can restrict the invocation of garbage collection to only certain code points in the program, when no "hidden" references exist

(ii) The compiler can write out information that the garbage collector can use to recover all the references, such as specifying which registers contain references, or how to compute the base address of an object that is given an internal address

(iii) The compiler can assure that there is a reference to the base address of all reachable objects wherever the garbage collector may be invoked.

* There are 4 basic operations that a mutator performs to change the set of reachable objects

(i) Object allocations: These are performed by the memory manager, which returns a reference to each newly ~~created~~ allocated chunks of memory. This operation adds members to the set of reachable objects

(ii) Parameter Passing & Return Values: References to objects are passed from the actual input parameter to the corresponding formal parameter, & from the returned result back to the caller. Objects pointed to by those references remain reachable.

(iii) Reference Assignments: Assignments of the form $u = v$, where u & v are references, have 2 effects

(a) u is now a reference to the object referred by v. As long as u is reachable, the object it refers to is surely reachable.

(b) The original reference to u is lost. If this reference is the last to some reachable object, then that object become unreachable. Any time an object becomes unreachable, all objects that are reachable only through references contained in that object also become unreachable.

(iv) Procedure Returns: As a procedure exits, the frame holding its local variables is popped off the stack. If the frame holds the only reachable reference to any object, that object becomes unreachable.

Reference Counting Garbage Collectors.

* We will consider a garbage collector based on reference counting, which identifies garbage as an object changes from reachable to unreachable, the object can be deleted when its count drops to zero. With a reference counting garbage collector, every object must have a field for the # reference count & it should be maintained as follows

(ii) Object Allocation : The reference count of the new object is set to 1.

(iii) Parameter Passing : The reference count of each object passed into a procedure is incremented.

(iv) Reference Assignments : For statement u = v, where u & v are references, the reference count of the object referred to by v goes up by one & the count for the old object referred to by u goes down by one.

(v) Procedure Returns : As a procedure exits, objects referred to by the local variables in its activation record have their counts decremented. If several local variables hold references to the same object, that object's count must be decremented once for each such reference

(vi) Transitive Loss of Reachability : Whenever the reference count of an object becomes zero, we must also decrement the count of each object pointed to by a reference within the object.

* Reference counting has a main disadvantages :

(i) It cannot collect unreachable, cyclic data structures and it is expensive.

(ii) Cyclic data structure are quite plausible ; data structure often point back to their parent nodes or point to each other as cross references.