UNIT-IV

Run-time Environments: Storage organization, Stack allocation of space, Access to Nonlocal Data on the stack, Heap Management, Introduction to Garbage collection, Introduction to Trace-based collection.

Machine-Independent Optimizations: The principal Source of optimization, Introduction to Data-Flow Analysis, Foundations of Data-Flow Analysis, constant propagation partial Redundancy elimination, Loops in Flow Graphs.

Run-time Environment

Introduction:

When a program is executed then the compiler converts the source code into machine code.

→ The machine code should loaded into the RAM.

→ Along with the machine code there are some supporting terms in RAM such as

* Library files
* Environment variables
* Memory space.

} These terms

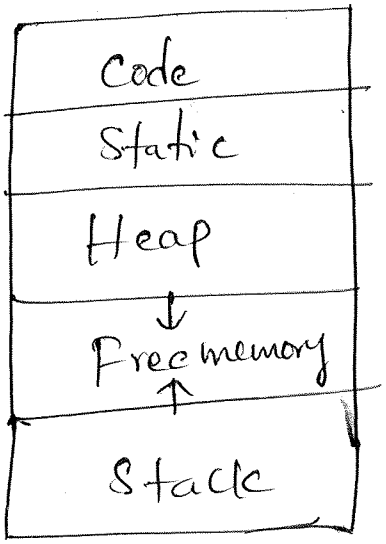
can combinely referred as Run time environment.

→ An Environment in which a program or application is executed is called Run-time environment.

I Storage Organization:

The target program runs in its own logical address space. This logical address space is shared between compiler, operating system and target m/c

The sub division of run-time memory into code and data areas are



Code: This area is used to place the executable target code, as the size of the generated code is fixed at compile time.

Static: The size of some program data objects may al be known at compile time. These objects are placed in the static area.

Stack and Heap: These areas are placed at the two ends o the unused space, and grows towards each other.

→ Stack area is used to store Activation Records

→ Heap area is used to allocate and deallocate arbitrary, dynamic chunks of storage.

(i) Static Versus Dynamic Storage Allocation

Static	dynamic
→ Is also known as compile-time allocation	→ Is also known as run-ti allocation.
→ The decision can be made by compiler looking at text of the program.	→ The decision can be mad by only when program is running.
→ Eg:- code and static area	→ Eg:- Stack and Heap areas.

Compilers allocate dynamic space using a combination of both the stack and Heap areas.

→ Garbage allocation enables the reuse of space allocate to useless data elements.

II Stack allocation of space:

A compiler uses procedures, functions or methods.

→ Each time a procedure is called a space to its local variables is pushed onto a stack.

→ when procedure terminates that space is popped off the stack automatically.

Activation: The execution flow of a procedure is called activation.

Activation Record: It contains all necessary informa n require to call a procedure.

(i) Activation Trees:

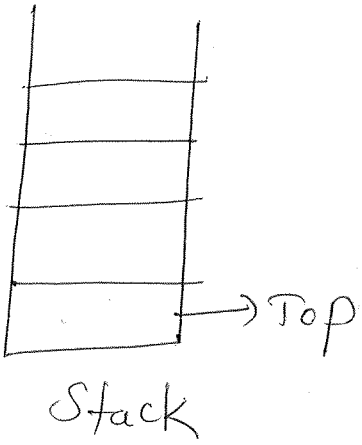
Whenever a procedure is executed its activation record is stored on the stack which is also known as control stack.

→ The activations of procedures during the running of an entire program by a tree called activation tree.

→ Each node corresponds to the activation and the root is the activation of "main" procedure that initiates execution of a program.

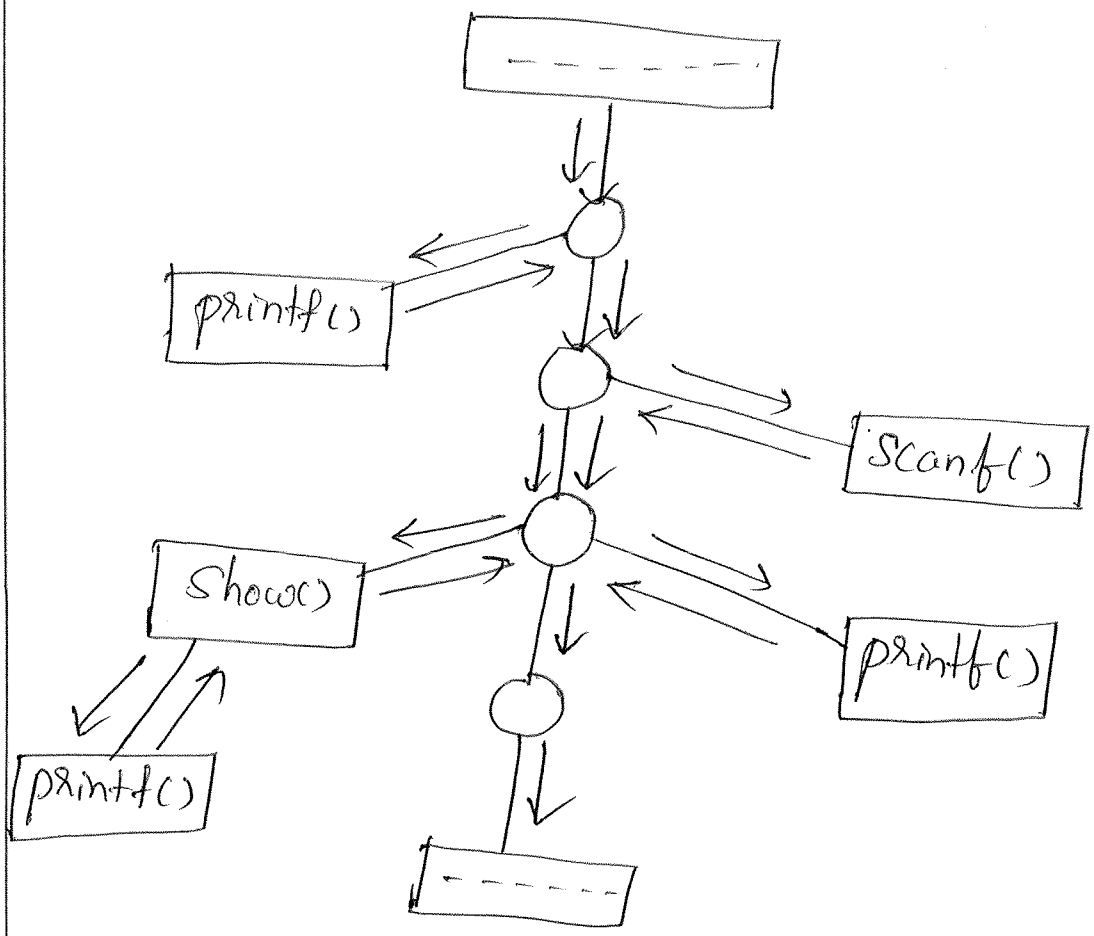
Eg:-

```
main()
{
 ----
fnct call();
}
void fnct()
{
 ----
}
```



The series of activations in the form of a tree i.e Activation tree can be be represented as

```
printf ("Enter name");
Scanf ("%s", username);
show (username); printf ("Enter any key");
 ----
void show (char *user)
{
  printf ("your name : %s", user);
}
```



The sequence of procedure calls corresponds to pre order traversal of the tree.

(iii) Activation Records:

Procedure calls and returns are usually managed by run-time stack called control stack.

→ Each live activation has an activation record.

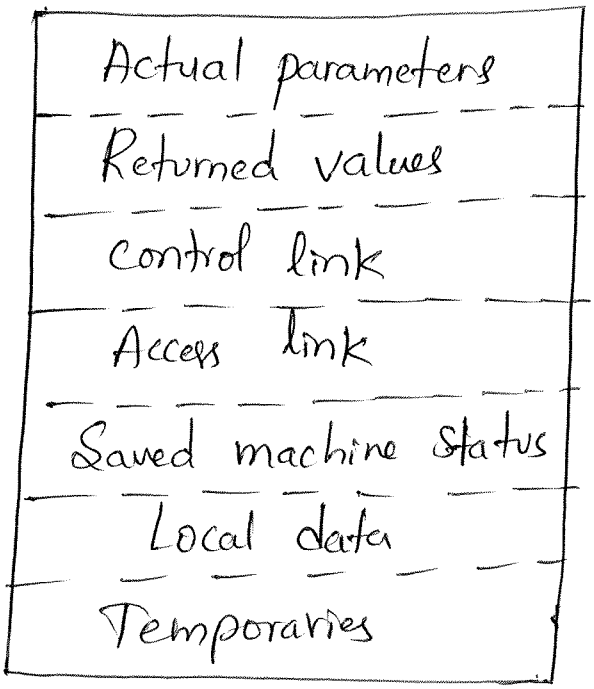


fig:- General Activation Record.

1. Temporaries — It holds/stores temporary values
2. Local data — It represents local data of a procedure.
3. Saved machine status — It gives information about state of machine. It includes return address and contents of registers.
4. Access link — It gives the information of data stored outside the local scope.
5. Control link — It points the activation record of caller.
6. Returned values — It is a space to hold any values returned by called procedure.
7. Actual parameters :- It is a space to store actual parameters of a procedure (calling

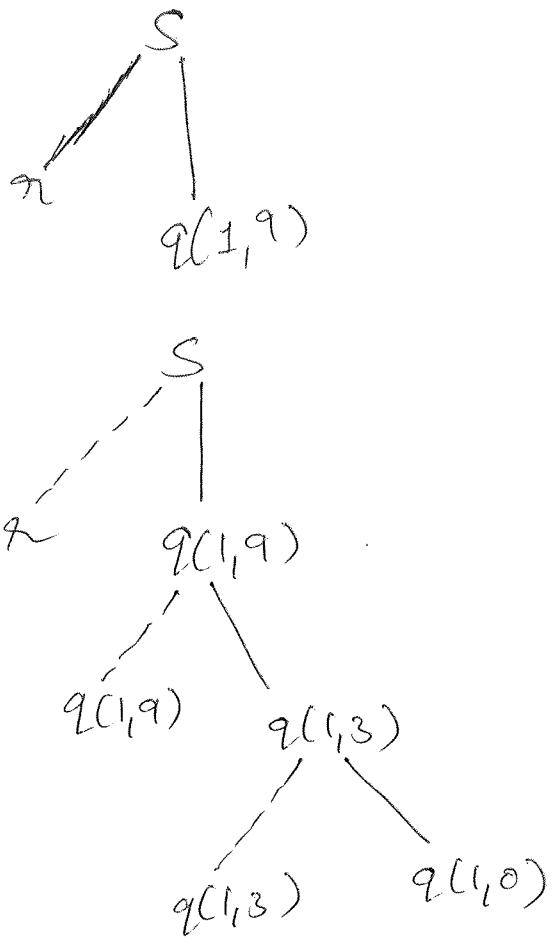
(iii) Calling Sequences:

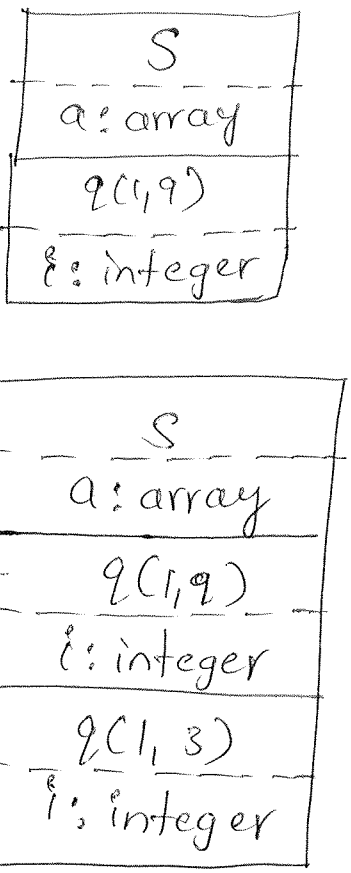
Here procedure calls are implemented by generat
calling sequences.

→ Calling Sequence → It is a code that allocates an activation record on stack.

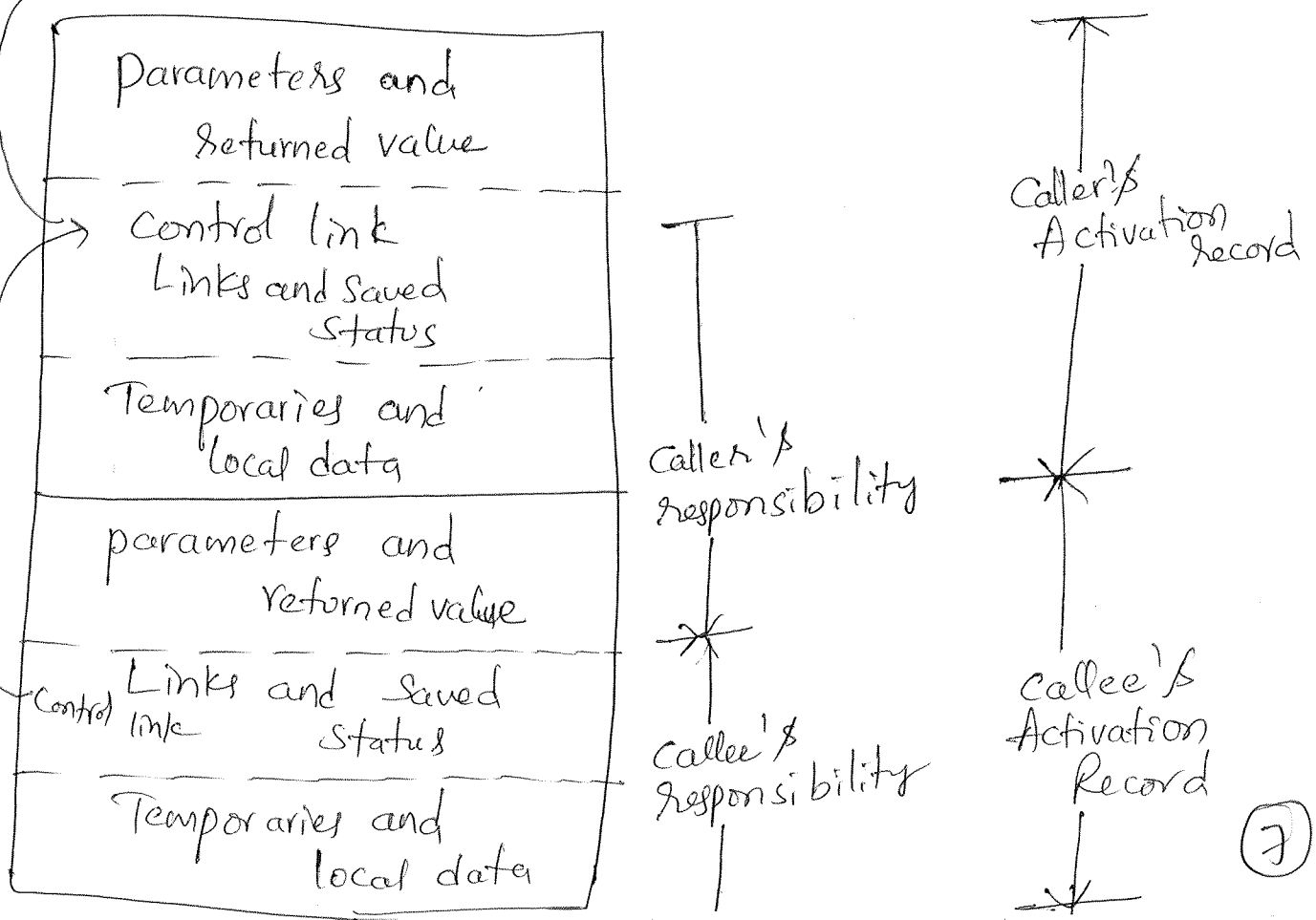
→ Return Sequence → It is a code that restores the state of the machine.

Position in Activation tree	AR on the stack
S S r	S a : array S a : array r i : integer





→ The code in a calling sequence is devided between the calling procedure (caller) and the called procedure ("callee")



(iv) Variable length Data:

Stacks are usually used to allocate space to fixed size variable, but can also be used for variables whose size is not known at compile time.

→ This space allocation can be done using arrays.

→ The storage for arrays is not part of Activation Record.

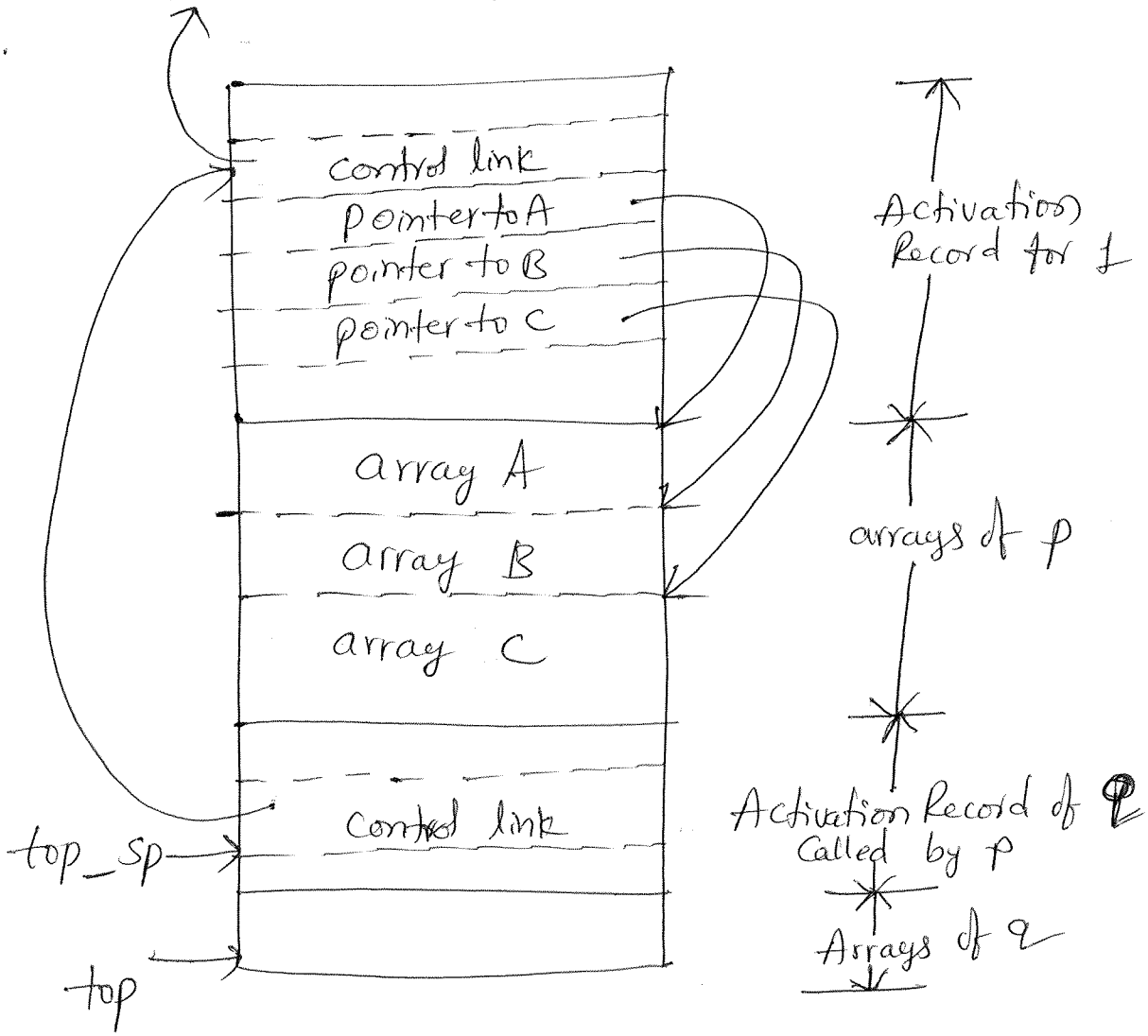


fig:- Access to dynamically allocated Arrays

(iii) Access to Nonlocal data on the Stack:

Here, we consider how procedures access their data. It is a mechanism for finding data used within procedure "p" but does not belongs to "p".

Here we will discuss the following

(i) Data Access without Nested procedures
(ii) Issues with Nested procedures
(iii) A language with Nested procedure Declarations
(iv) Nesting Depth
(v) Access Links
(vi) Manipulating Access Links
(vii) Access Links for procedure parameters
(viii) Displays.

(i) Data Access without Nested procedures:

In the C family of languages, all variables are defined either within a single function or outside any functio (globally)

→ It is impossible to declare one procedure whose scope is entirely within another procedure.

→ A global variable has a scope consisting of all functi

In non nested procedures, variables are accessed as follows

1. Global variables are declared statically.
2. Other variables (local) are declared locally at the top of the stack, using the top_sp pointer.

(ii) Issues with nested procedures:

There are some issues present with nested procedu they are

(a) Access becomes more complicated when procedure declarations are nested.

(b) The nested declarations does not tell the relative position of Activation Record at run time.

(iii) A language with Nested procedure Declarations-ML

The C family of languages, and other languages do not support nested procedures so, another language is implemented i.e., ML.

Properties of ML:

→ ML - is a functional language - means the variables once declared and initialized, are not changed

→ variable declaration

Syntax:- val <name> = <expression>

→ Syntax for function definition is

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

→ For function bodies we shall use let-statements as

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

(iv) Nesting depth:

Nesting depth defines the level from the start of procedure at which a particular nested procedure is defined.

→ A procedure without nested, defined immediately within the procedure then the nesting depth is 1.

→ The nesting depth for nested procedure is "i+1"

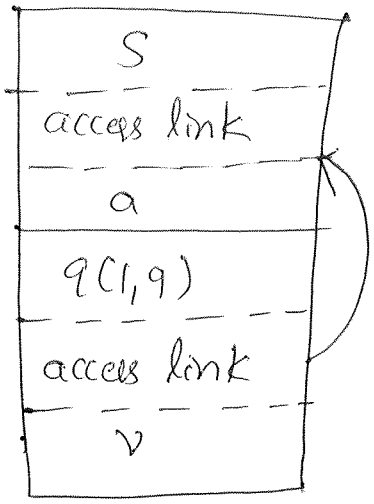
```
fun sort(inputFile, outputFile)=
  let
     val a= array(11,0);
    fun readArray(inputFile) =---;
          ---- a---;
    fun exchange(i, j) =
          ---a ---;
    fun quicksort(m,n) =
      let
         val v=---;
         fun partition(y,z)=
            ---a---v--- exchange ---
      in
         ---a---v--- partition ... quicksort
      end
  in
    --- a--- readArray --- quicksort---
  end;
```

Fig:- A version of quicksort in ML style, using nested functions.

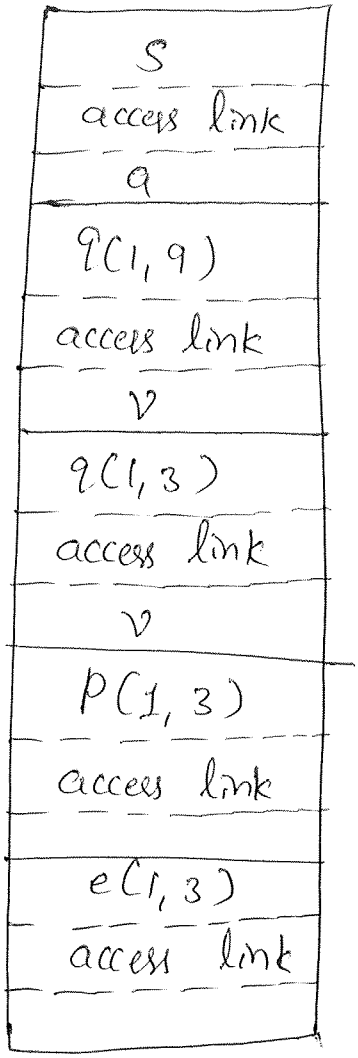
(V) Access Link:

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

The access link for finding nonlocal data.



(a)



(b)

(vi) Manipulating Access links:

Let us consider what should happen when a procedu. q calls procedure p, explicitly.

(i) p is at higher nesting depth than q. Then p must be defined immediately within q.

(ii) The nesting depth of n_p of p is less than or equal to th. nesting depth of n_q of q.

(vii) Access Links for procedure parameters:

When a procedure p is passed to another procedure q as a parameter and q calls its parameter, it is possible that q does not know the context in which

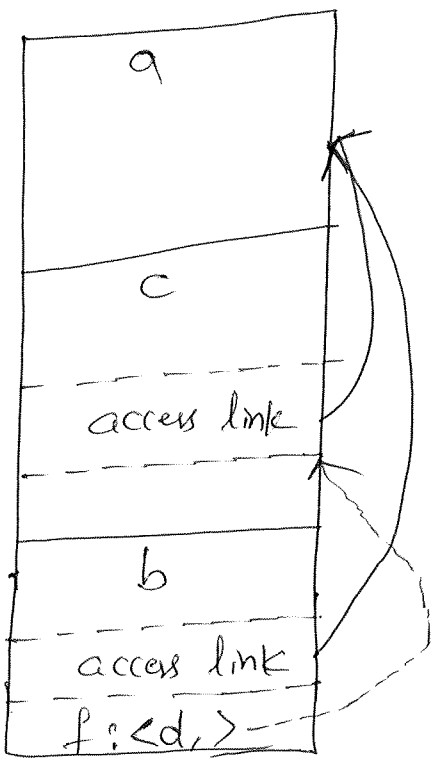
p appears in the program

→ The solution to this problem is when procedures are used as parameters, the caller needs to pass, along with the name of procedure-parameter, the proper access link for that parameter.

Eg:-

```
fun a(x) =
    let
        fun b(f) =
        ---- f---;
        fun c(y) =
    let
        fun d(3) = ----
    in
        --- b(d) ---.
    end
in
    ---- c(1) ----
end
```

fig:- Sketch of ML program that uses function parameters.



(viii) Displays:

If the nesting depth gets large, then we have to follow long chains of links to reach the data.

→ An auxiliary array d, called the display, which consists of one pointer for each nesting depth is implemented.

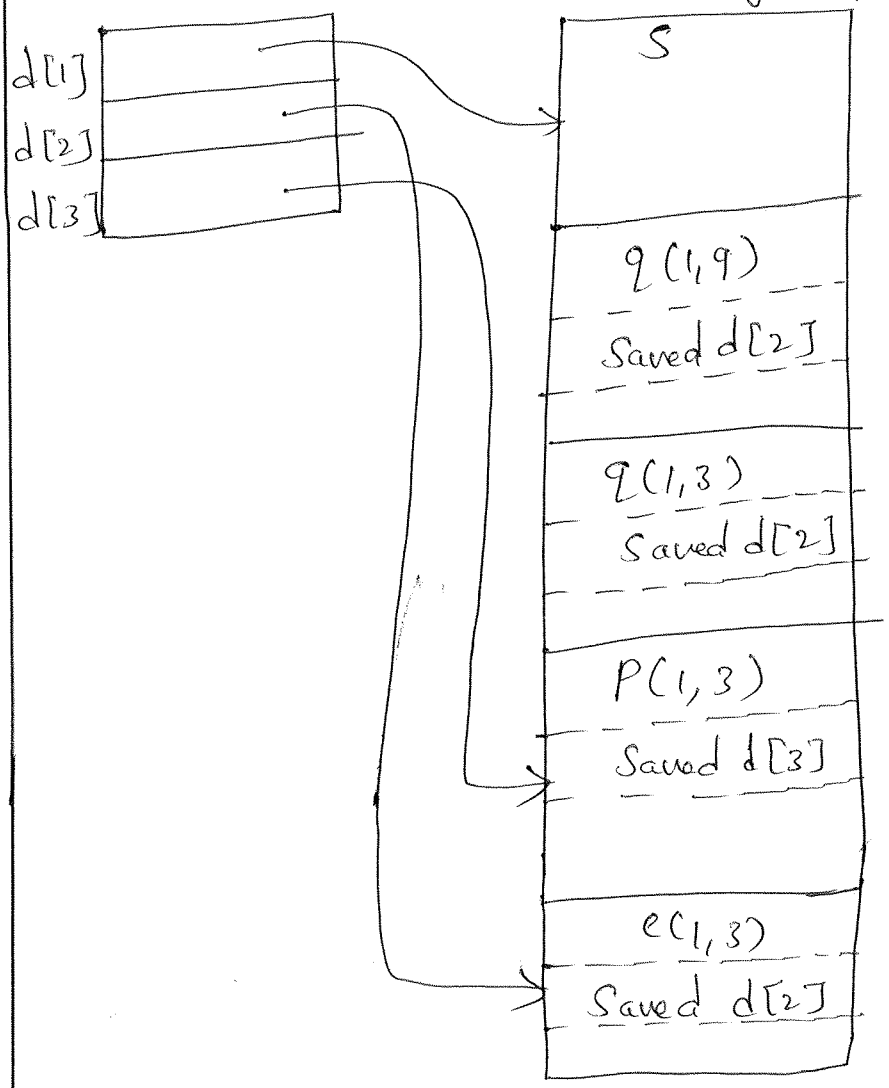


Figure: Maintaining the display.

(iv) Heap Management:

Heap is unused memory space available for allocation dynamically.

→ It is used for data that lives indefinitely, and is freed only explicitly.

→ The existence of such data is independent of the procedure that created it.

(i) The memory manager:

Memory manager is used to keep account of the free space available in the heap area.

Its functions include,

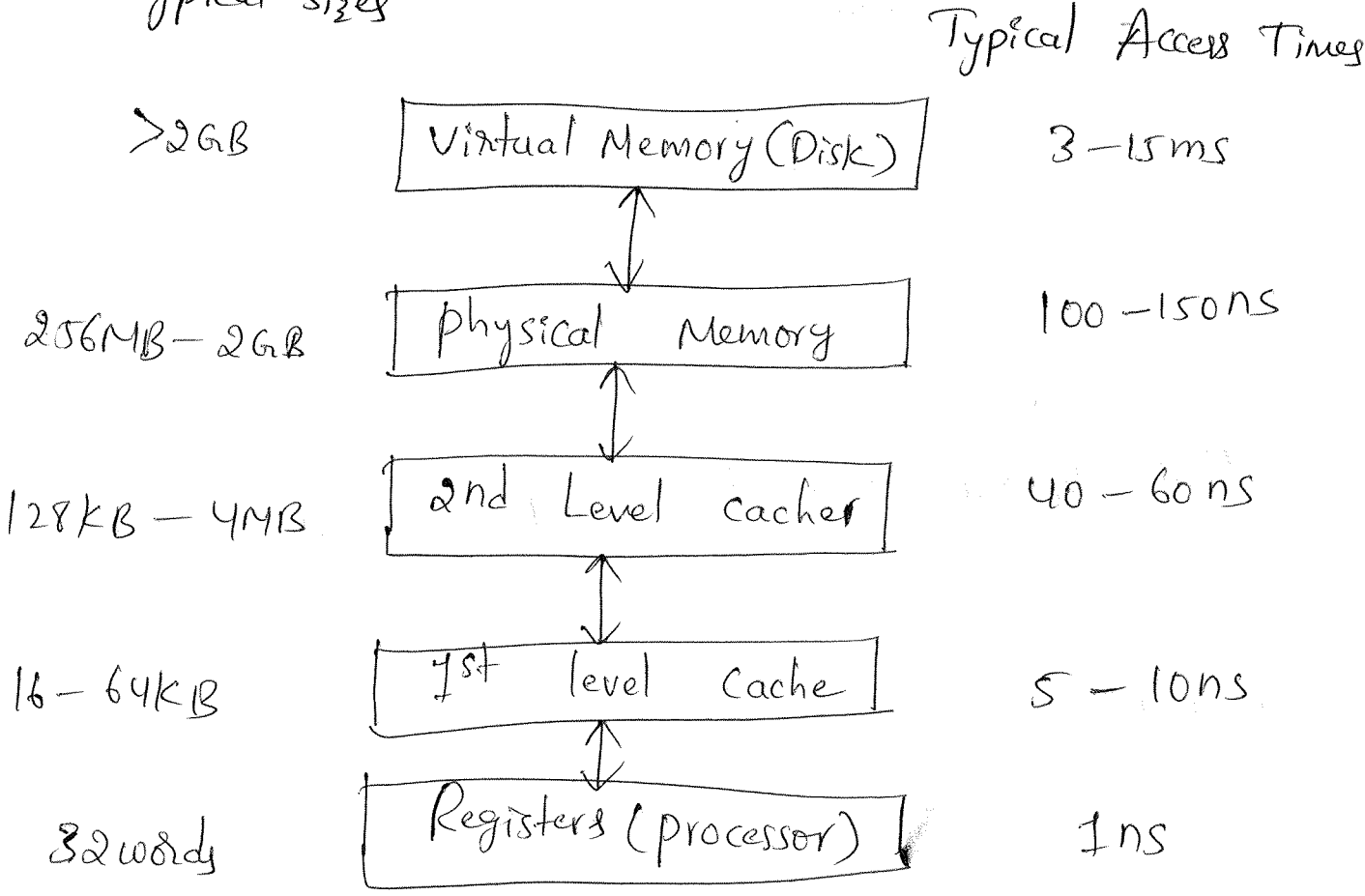
* allocation
* deallocation

The properties of an efficient memory manager include

* Space efficiency
* Program efficiency
* low overhear

(ii) The memory hierarchy of a computer

* Typical Sizes



A memory hierarchy consists of a series of storage elements, with the smaller faster ones "closer" to the processor, and the larger slower ones further away.

(iii) Locality in programs:

Locality refers to the amount of data requirement for a particular program, and the time required to access or locate the data.

locality is of two types namely,

(a) Temporal locality: This is present if the memory location accessed by it are likely to be accessed again soon.

(b) spatial locality: this is the condition when locations close to the location accessed are likely to be accessed within a short period of time.

(iv) Reducing Fragmentation

At the beginning of program execution, the heap is a contiguous unit of free space

→ As the program allocates and de-allocates memory, this space is broken up into free and used chunks of memory as holes.

Best-fit and Next-fit object placement

we can reduce fragmentation by controlling how the memory manager places objects in the heap.

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

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

Garbage Collection:

Garbage refers to the data that cannot be referenced. Such garbage can be collected either manually or automatically.

→ Garbage collector collects garbage and reduces heap space.

(i) 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 may introduce & drop references to existing objects. Objects become garbage when the mutator program cannot reach them.

Type Safety

Not all languages are good condidates for automatic garbage collection.

→ It 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 chunk of allocated memory space.

→ For some languages in which type cannot be determined at compile time, But can be determined at run time. This is called dynamically typed languages.

(ii) Performance metrics:

Some of the performance metrics that must be considered when designing a garbage collector.

(i) Overall execution Time:
(ii) Space usage
(iii) pause time
(iv) Program Locality

(iii) 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.

VI Trace-Based Collection:

run periodically to find unreachable objects and reclaim their space.

(i) Mark-and-Sweep collector:

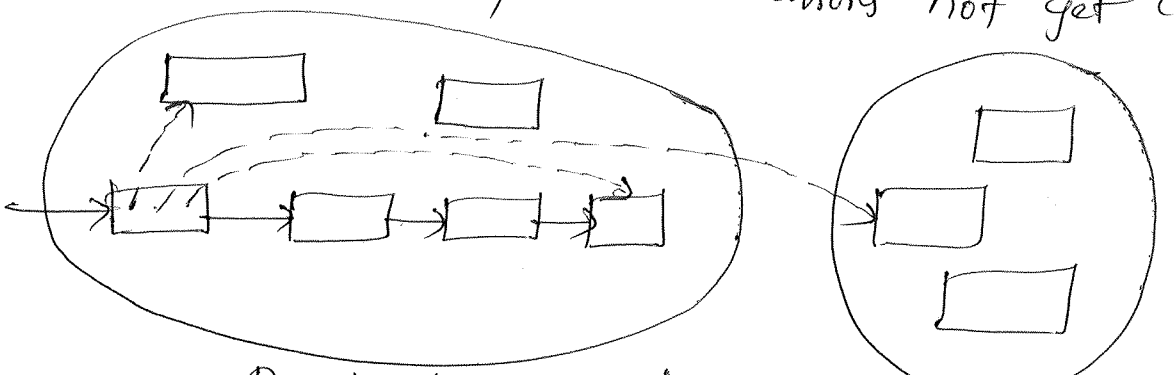
It is straight forward, finds all unreachable objects and put them on the list of free space.

Algorithm:

```
1. add each object referenced by the root set to list unscanned and set its reached-bit to 1;
2. while (unscanned ≠ ϕ) {
3. remove some object o from unscanned;
4. for (each object o' referenced in o) {
5.     if (o' is unreached) {
6.         Set the reached-bit of o' to 1;
7.         put o' in unscanned;
        }
    }
}
```

8. Free = ϕ;
9. for (each chunk of memory o in the heap) {
10. if (o is unreached, i.e., its reached_bit is 0)
add o to Free
11. else set the reached_bit of o to 0;
}

Free - holds objects known to be free
unscanned - reached, but successors not yet considered.



Previously scanned objects reached bit = 1

Free and unreached objects reached bit = 0.

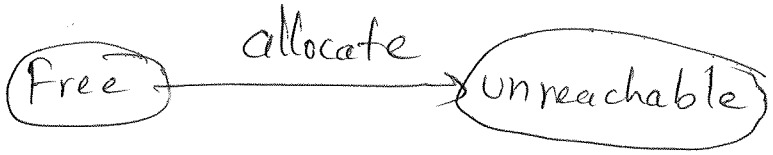
(ii) Basic Abstraction:

(a) the program runs and makes allocation request
(b) the Garbage collector discovers reachability by tracing
(c) the Garbage collector reclaims storage for unreachable objects

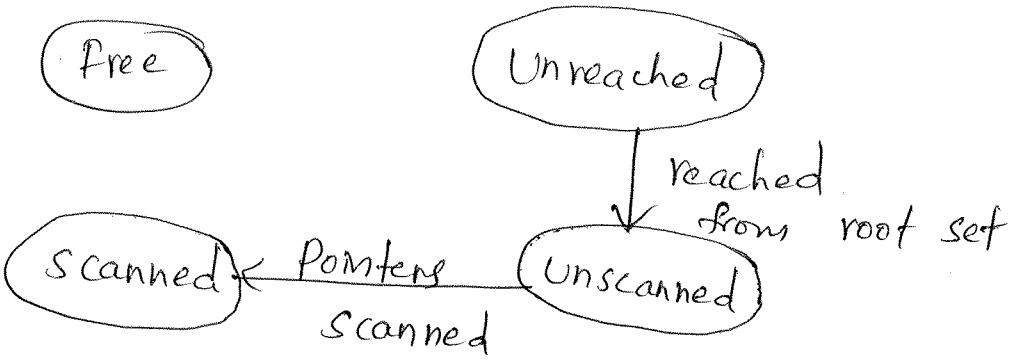
Four states for chunks of memory

Free - It is ready to be allocated
unreached - If the reachability has not been established
unscanned - It is reachable, but its pointers not yet been scanned.

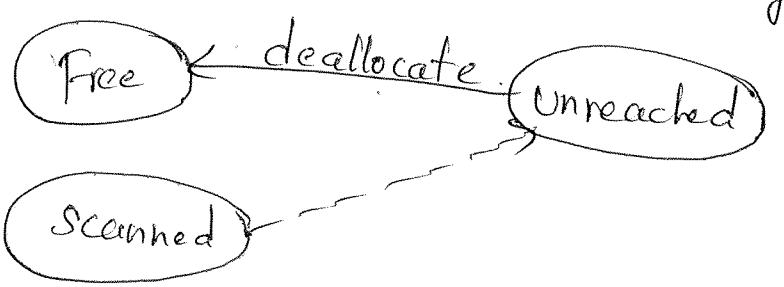
Scanned – Unscanned object will be scanned



(a) Before tracing.



(b) Discovering reachability.



(iii) Mark- and compact Garbage collectors:

~~There~~ are used to relocating collectors. i.e., move reachable objects in the heap to eliminate memory fragmentation.

It can be done in three phases

1. Marking phase
2. Algorithm Scans allocated section of the heap and computes new address for each of reachable objects
3. Copies objects to new locations.

(iv) Coping Collectors:

Garbage collection is performed by copying live objects from one semispace to the other semispace, which becomes the new heap this process is called copying collectors.