UNIT-5

Symbol table:

Symbol table is a datastructure which is used by compiler to keep track of scope and binding information about names i.e, a compiler needs to collect and use information about the names appearing in the source program. this information is entered into a datastructure called a symbol table.

Symbol table Format:

The symbol table is ~~but~~ used by various phases as follows

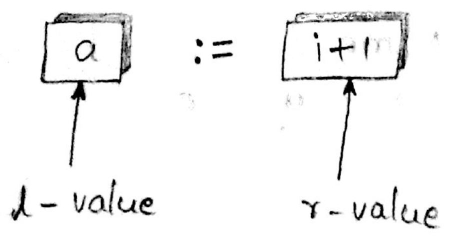
1. Lexical analyzer stores the information of the symbols in symbol table.
2. parser while checking the syntax of the source, makes use of symbol table to verify the information about the tokens being generated.
3. semantic analysis phase refers symbol table for type conflict issue.
4. code generation refers symbol table knowing how much run time space allocated? what type of run time space is allocated?

some important terms appearing with regards to symbol table as

l-value and r-value:

The l and r prefixes come from left and right side assignment.

For example,



Entries in symbol table:/ use of symbol tables:

→ To achieve compile time efficiency compiler makes use of symbol table.

→ It associates lexical names with their attributes.

Symbol table

Names	Attributes

symbols get stored with associated Information

→ The item to be stored in symbol table are,

1. variable names
2. constants
3. Procedure names
4. Function names
5. Literal constants and strings
6. Compiler generated temporaries
7. Labels in source languages.

Name Representations in symbol table:

There are two types of name representation. They are

1. **Fixed-Length Name:** A Fixed space for each name is allocated in symbol table. In this type of storage if name is ~~too~~ small then ther is wastage of space.

For example,

Fixed-length symbol table

Name										Attribute
c	a	l	c	u	l	a	t	e		
s	u	m								
a										
b										

The name can be referred by pointer to symbol table entry.

Variable-length Names

The amount of space required by string is used to store the names.

The name can be stored with the help of starting index and length of each name.

for example,

Variable-length symbol table

Name		Attribute
starting Index	length	
0	10	
10	4	
14	2	
16	2	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	L	c	u	L	a	t	e	$	s	u	m	$	a	$	b	$

Need for symbol tables

The symbol tables are required for.

i, for quick insertion of identifier and related informatio

ii, for quick searching of identifier

Data structure to symbol table

There are 3 types of data structures used for organization of i, list ii, hashing iii, Trees.

i, list : The easiest way to implement a symbol Table is a list data structure

list can be of 2 types

i, linear list

ii, self organising list.

i, linear list : linear list is a simplest kind of mechanism to implement a symbol Table

- Arrays are nothing but the linear list.
- linear list uses a single array or no of arrays to implement a symbol Table.
- The linear list is open at one end so that symbols can be added only from that end point (i.e., bottom end)

symbol	properties
Name 1	Info 1
Name 2	Info 2
⋮	
Name n	Info n

Available
$\longrightarrow$
(start of empty slot)

→ In this method an array is used to store names and their associated information

→ New names can be added in the order as They arrive.

→ The ptr 'available' is maintained at the end of all stored records

→ To retrieve the information about some name we start from begining of array & goon searching upto available pointer. If we reach at pointer available without finding a name we get an error 'use of undeclared name'.

→ while inserting a new name we should ensure that it should not already be there.

→ If it is already present then the error is 'Multiple defined name'.

ii, self - organising list:

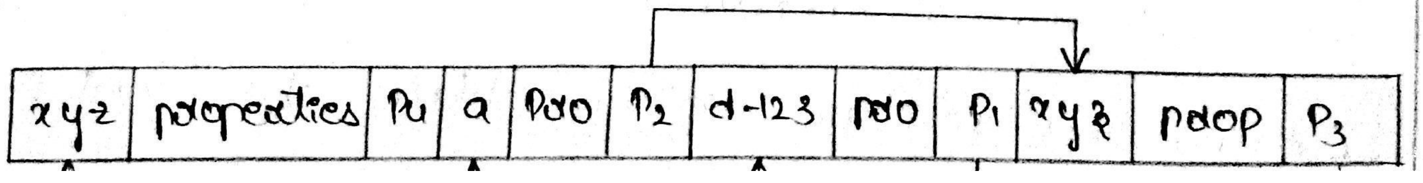
A self - organising list names makes uses of linked list

symbol	properties	pointer

pointer is used to point the next record of the list

eg: for the order of symbols given by

d - 123, a, m, xyz



first →

Name 1	Info 1
N_2	I_2
N_3	I_3
N_4	I_4

2. Hashing: In hashing we maintain two tables

i, hash table
ii, symbol Table

i, hash tables:-

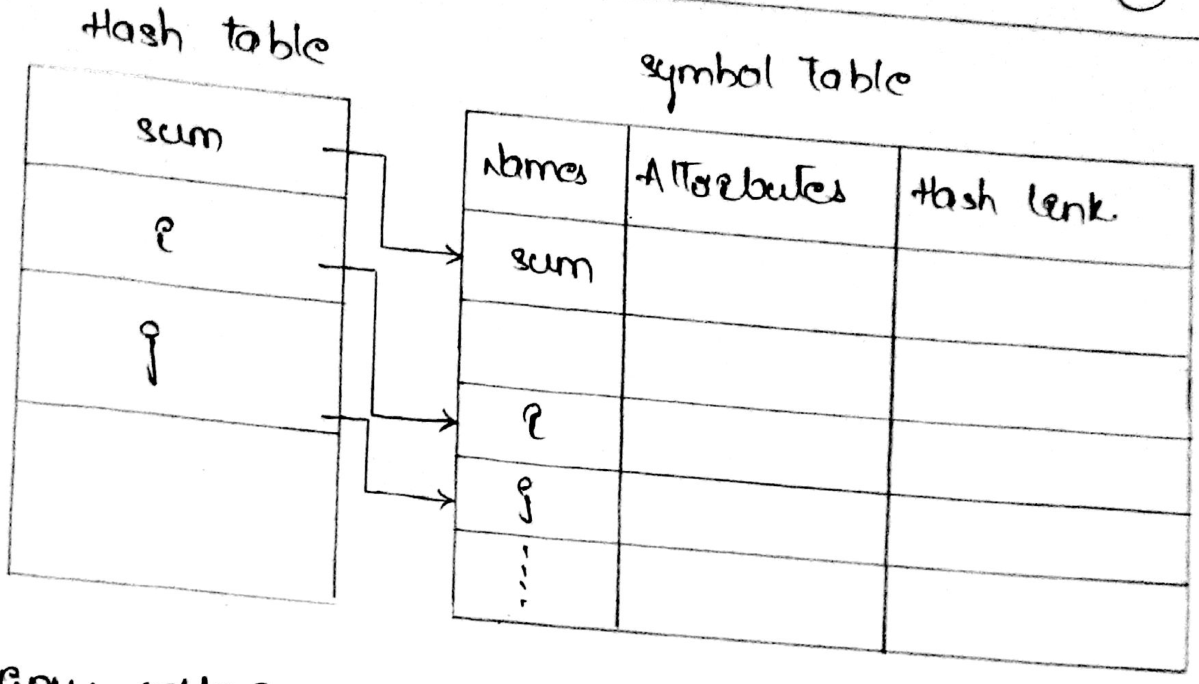
Hash table consists of 'k' no of entries i.e., 0, 1, 2, 3, 4, 5 ---- k-1 and those entries are nothing but pointers to the symbol Table which inturn pointing to names of symbol table.

→ we use a hash function 'h' such that h(name) will result any integer between '0' to 'k-1'.

→ the hash function is

position = h(name).

→ using this position we can obtain the exact locations of names in the symbol table.



→ Various collision resolution techniques are

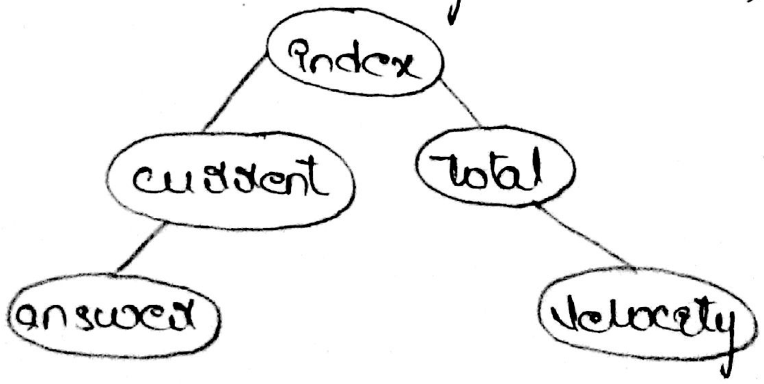
1. open addressing
2. chaining
3. Rehashing

3. Trees

A Binary Tree is built using ordering of tokens and the datastructure is represented as.

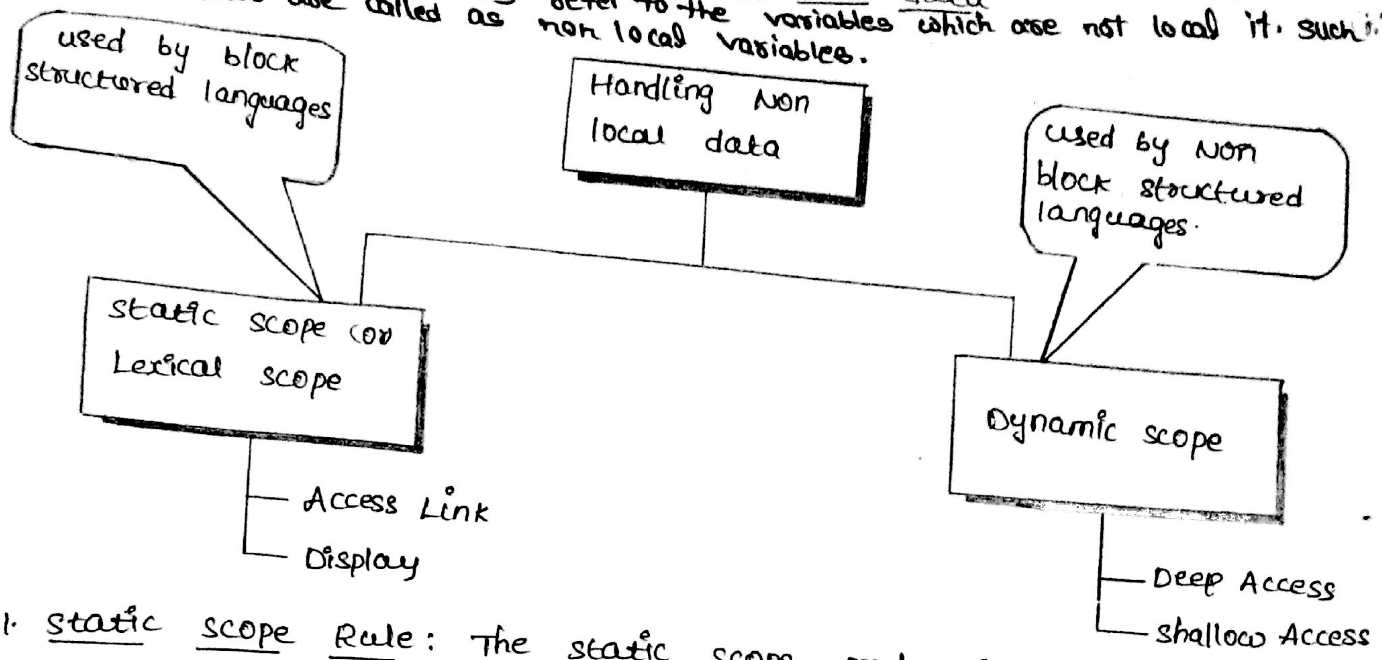
left child	Name of the symbol	Mode info	Right child

eg: int index, total, velocity, current, answer;



Access to Non local data

A Procedure may sometimes refer to the variables which are not local it. such variables are called as non local variables.


used by block structured languages
Handling Non local data
used by Non block structured languages.
static scope (or Lexical scope
Dynamic scope
Access Link
Display
Deep Access
shallow Access

1. <u>static</u> <u>scope</u> <u>Rule</u>: The static scope rule is called as lexical scope. In this type the scope is determined by examining the program text. PASCAL, C and ADA are the languages that use the static scope rule. These languages are also called as block structured languages.

2. <u>Dynamic</u> <u>scope</u> <u>Rule</u>: For non block structured languages this dynamic scope allocation rules are used.

The dynamic scope rule determines the scope of declaration of the names at runtime by considering the current activation.

LISP and SNOBOL are the languages which use dynamic scope rule.

<u>Static</u> <u>Scope</u> (or) <u>Lexical</u> <u>scope</u>:

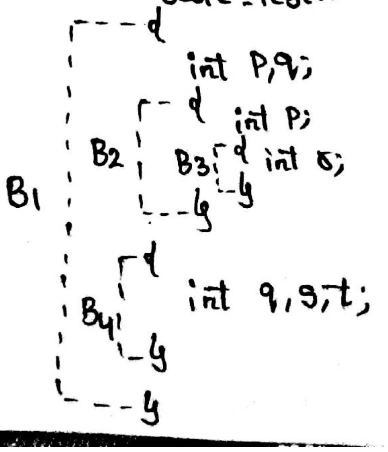
<u>Block</u>: The block is a sequence of statements containing the ocal data declaration and enclosed within the delimiters.

or example,

Declaration statements

- - - -

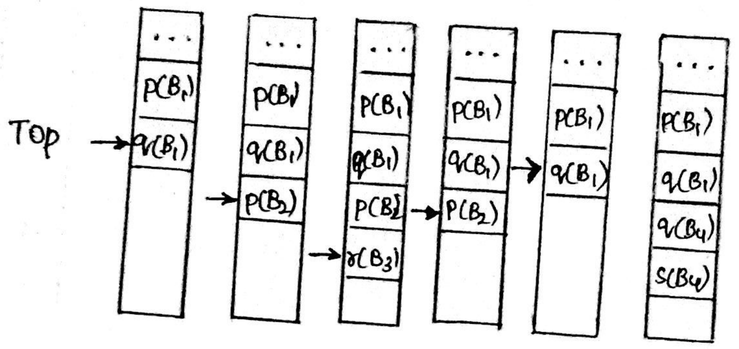
<u>EX</u> :-



```
scope_test()
{
  int p,q;
  {
    int p;
    {
      int r;
    }
  }
  {
    int q,s,t;
  }
}
```

The declarations are visible at a program point are,
1. The declarations that are made locally in the procedure.
2. The names of all enclosing procedures.
3. The declarations of names made immediately within such procedures.

Block structure storage Allocation by stack



Lexical scope for Nested Procedure:

→ Nested procedure is a procedure that can be declared within another procedure.

→ A procedure pi, can call any procedure that is its direct ancestor or older siblings of its direct ancestor.

→ The nested procedures can be as shown below,

procedure main

procedure p1

procedure p2

procedure p3

procedure p4

Nesting Depth:

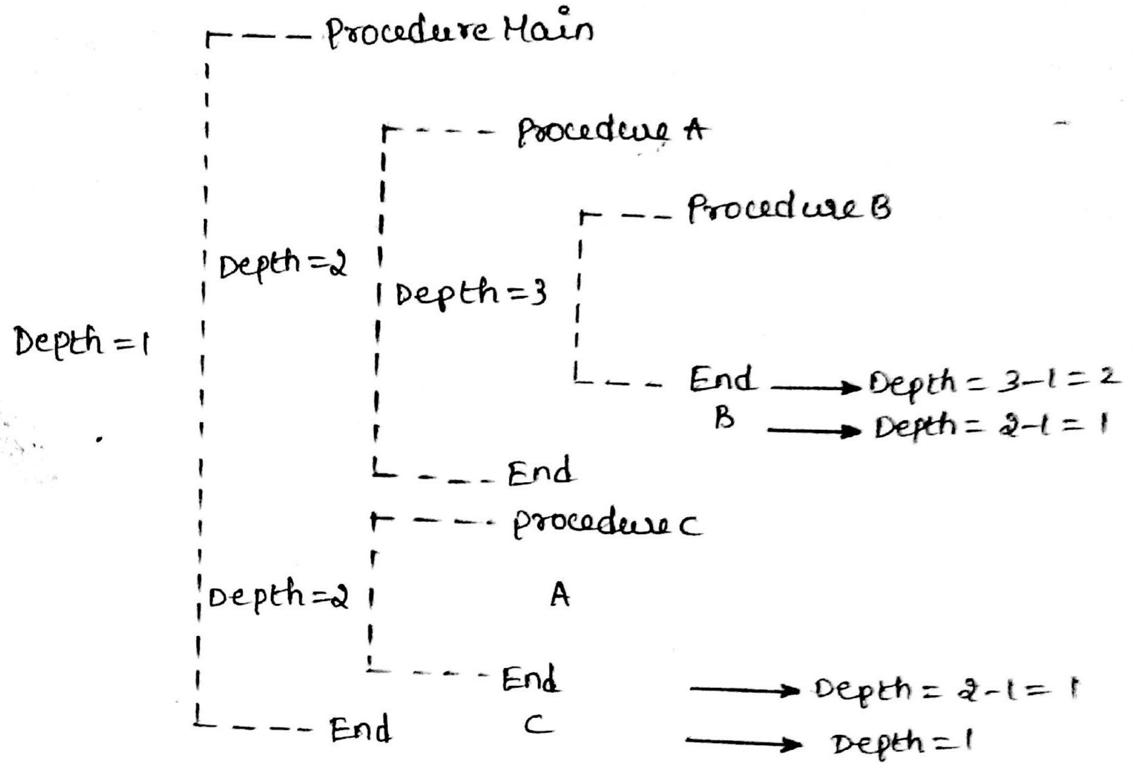
Nesting depth of a procedure is used to implement lexical Scope. The nesting depth can be calculated as follows.

1. The nesting depth of main program is 1
2. Add 1 to depth each time when a new procedure begins
3. subtract 1 from depth each time when you exit from a nested procedure.

4. The variable declared in specific procedure is associated with nesting depth.

For example: consider the following piece of code

Representing the depths of Nested procedures



The lexical scope can be implemented using access link and displays.

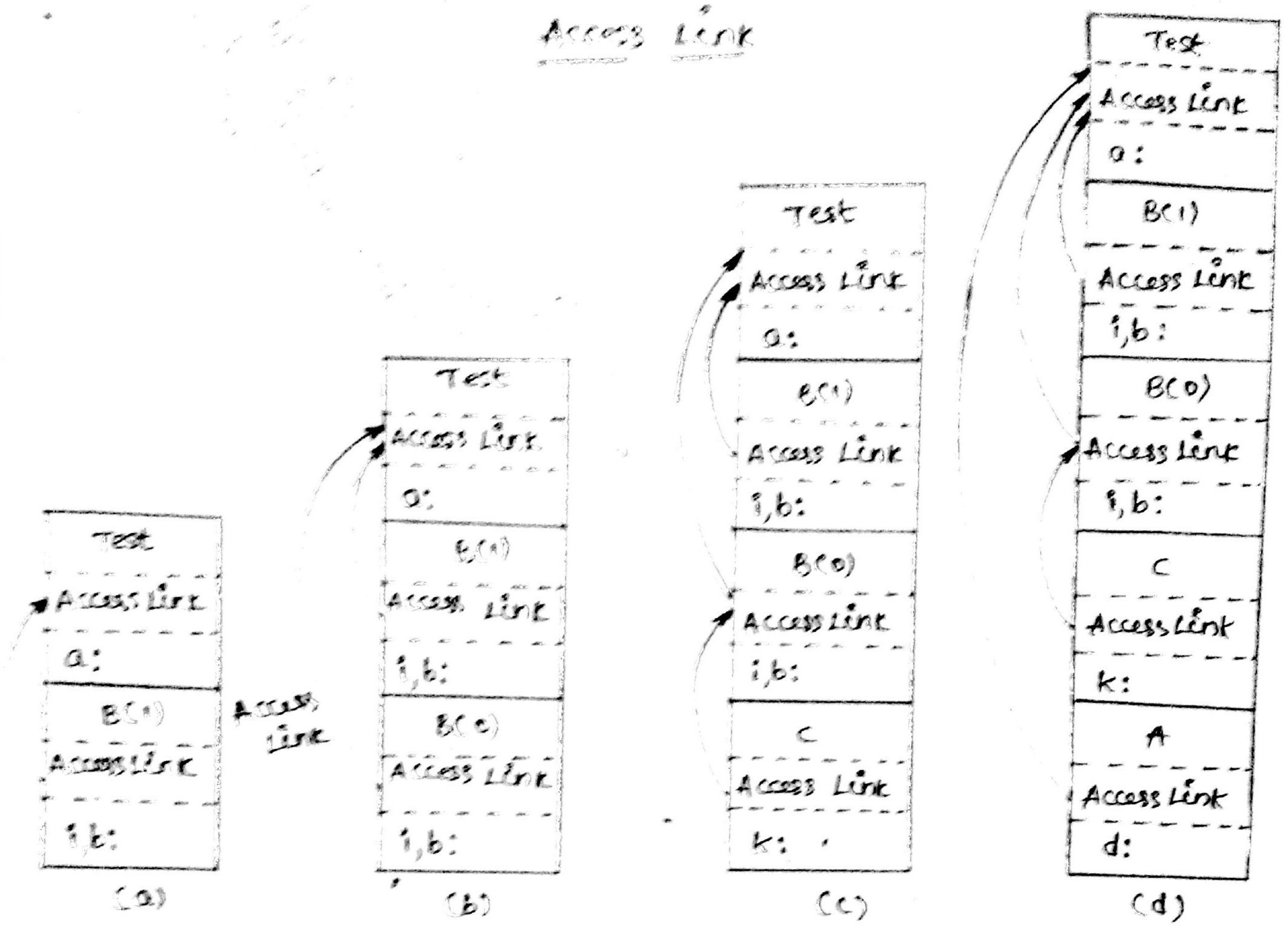
1. Access Link:

The implementation of lexical scope can be obtained by using pointer to each activation record. These pointers are called access links.

For example:

```
Program test;
var a: int;
  Procedure A;
   vard: int ;
    begin a=1, end;
 Procedure B (i: int));
  var b : int;
    Procedure C;
     var k: int ;
      begin A ; end;
        begin
        if (i<>0) then B(i-1)
            else C; end;
        begin B(1); end;
```

Access Link



To set up the access links at compile time

i) If procedure A at depth n_A calls procedure B at depth n_B then

case 1: If $n_A < n_B$, then B is enclosed in A and $n_A = n_B - 1$

case 2: If $n_A \geq n_B$, then it is either a recursive call or calling a previously declared procedure.

ii) the access link of activation record of procedure A points to activation record of procedure B where the procedure B has procedure A nested within it.

iii) The activation record for B must be active at the time of pointing.

iv) If there are several activation records for B then the most recent activation record will be pointed.

Thus by traversing the access links, non locals can be correctly accessed.

2. <u>Displays</u>: It is expensive to traverse down access link every time when a particular non Local variable is accessed. speed up the access to non locals can be achieved by maintaining an array of pointers called display.

In display

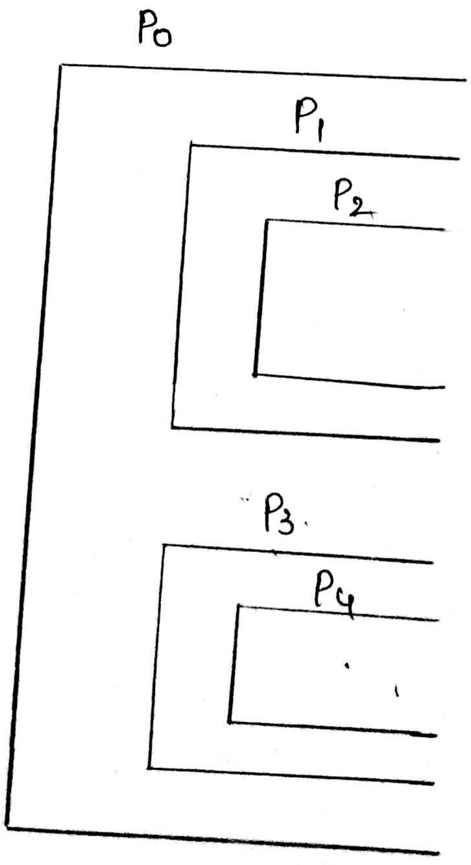
i) An array of pointers to activation record is maintained.

ii) Array is indexed by nesting level.

iii) The pointers point to only accessible activation record.

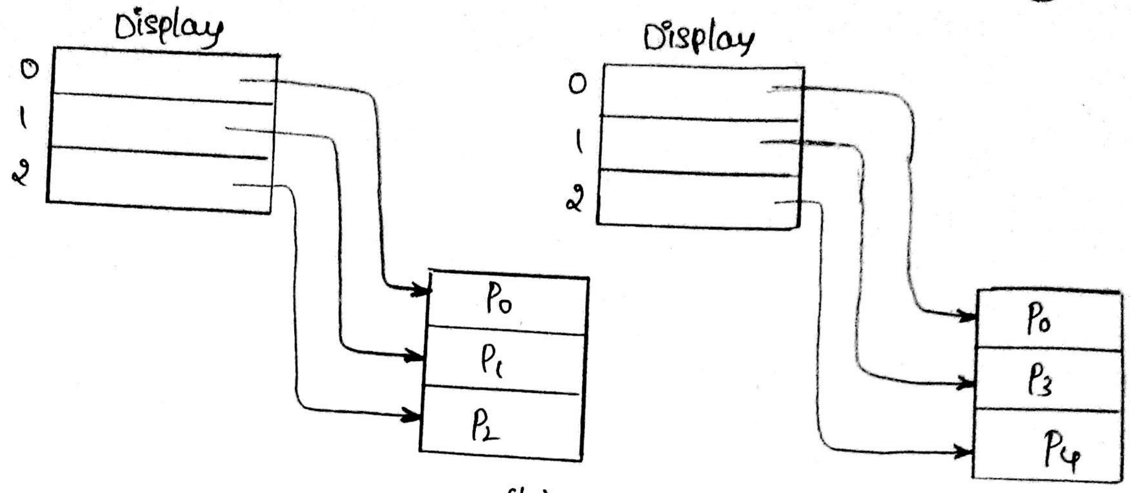
iv) The display changes when a new activation occurs and it must be reset when control returns from the new activation.

<u>For example</u>: Consider the scope of procedures as shown below



procedure

(a)



(b)

Maintaining the Display

Dynamic scoping: (storage allocation for Non block structured Languages)

In dynamic scoping a use of non local variable refers to the non local data declared in most recently called and still active procedure. therefore each time new bindings are set up for local names Called procedure. In dynamic scoping symbol tables can be required at run time.

To understand the concept of dynamic scoping consider the following example

```
procedure main( )
      value : Int ;
 procedure first( )
 begin
      value : Int ;
      value : = 1 ;
      print(value);
end
begin
    value :=2 ;
    print (value);
    first ( );
```

end ;

In PASCAL like languages static or lexical scope is used. Hence for a static scope the output will be,

2 2

However, LISP like languages use dynamic scope and under dynamic scope output will be,

2 1

There are two ways to implement non local accessing under dynamic scoping.

1. Deep Access: The idea is keep a stack of active variables, use control links instead of access links and when you want to find a variable then search the stack from top to bottom looking for most recent activation record that contains the space for desired variables. This method of accessing non local variables is called deep access.

As search is made "deep" in the stack hence the method is called deep access.

In this method a symbol table need to be used at run time.

2. Shallow Access: The idea is to keep a central storage with one slot for every variable name. If the names are not created at runtime ~~t~~when that storage layout can be fixed at compile time otherwise when new activation of procedure occurs, then that procedure changes the storage entries for its locals at entry and exit i.e, while entering in the procedure and while leaving the procedure.

* → Storage organization:-

Two types of structural languages

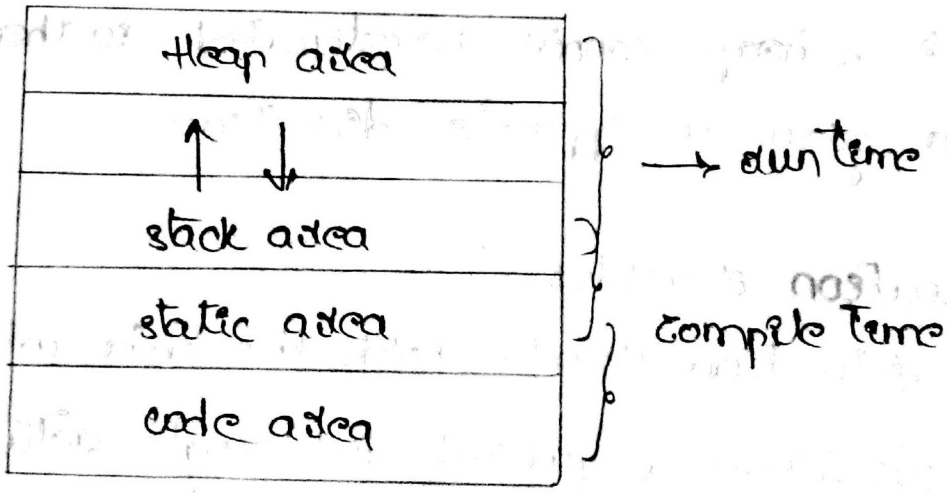
1. Block structured
2. Non-Block structured

Storage organization is used in structured language like pascal, fortran, C

Mainly it deals with 3 different issues.

→ Division of runtime memory into different areas
→ Management of the activation records
→ layout of the local data.

1. Initial division of run-time memory:-



code area :- this contains the generated Target code

→ stack area:

The stack stores the status of machine before the calling a procedure and the status can be restored

when the called procedure returns
The local variables and parameters of all 'c' procedures except for main program would reside in this area.

Static area:
In this static area address can be determined at compile time (in front)
In 'c' language global variables are stored in this area.
In fortran the addresses of all the variables can be determined statistically and kept in static area.

Heap area:
It contains the data created at runtime sizes of stack & heap cannot be estimated so they are arranged to grow in opposite direction

2. Activation records:
Activation record holds the data which contains all information required for single activation of a procedure.
→ Activation record is pushed on to the stack when a procedure is called & popped when procedure is returned
→ In fortran activation records are held in static area.

→ In pascal , activation records are held in stack area

formal of activation records:

return value	
Actual parameter	
[optional] control link	→ dynamic
[optional] Access link	→ static
saved machine status	
local data	
temporaries	

Various field of activation record are as follows.

1. local variables:

the data that is local to the execution of the procedure is stored in this field of activation record.

2. Temporaries (or) temporary values:

the temporary values are needed during evaluation of expression

such type of variables are stored in this field of activation record.

. saved machine status:

this field carries the information regarding the status of machine just before a calling a procedure.

→ And this field contains the machine register &

program counter.

4. control link:

this is optional. It points to the activation record of the calling procedure. This link is also called Dynamic link.

5. Access link:

this is also optional. It refers to the non local data in other activation record. This field is also called static link.

6. Actual parameters:

It holds the information about the actual parameters. This actual parameters are passed through called procedures.

7. Return value:

This field is used to store the result of the function call.

Example for Activation Record:

Note: To refer any variable 'x' in the procedure

= base pointer pointing to start of the procedure + offset of variable 'x' from base pointer

saved Reg
Parameters
local va

8	8
16	16
32	32
32	32
64	32
32	32
r = 8	32

storage as needed at runtime from data area is known as Heap.

a) static allocation:-

* static data area stores those data item's whose names are binded and associated to memory locations at compile time. such binding is called 'static binding'.

* In static binding the size of data item is available at compile time.

* once a data item is statistically allocated its memory space cannot be changed.

Limitations for static storage allocation:-

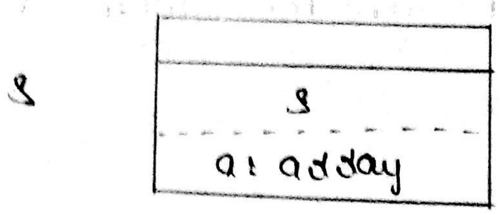
1. the size required must be known at compile time
2. Recursive procedures cannot be implemented if all local variables are statistically allocated
3. No data structure can be created dynamically if all data are static.

b) stack allocation:-

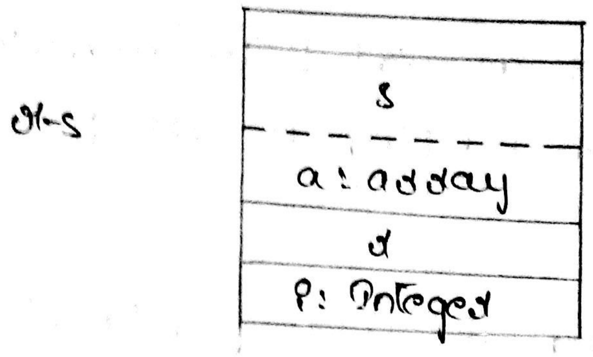
position in activation on activation tree

the stack

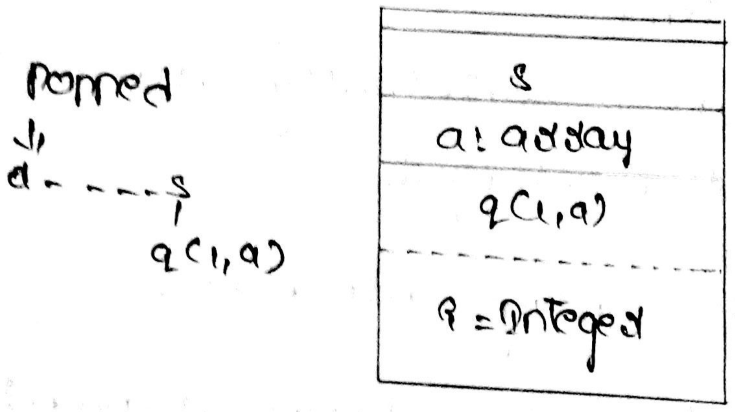
Remarks



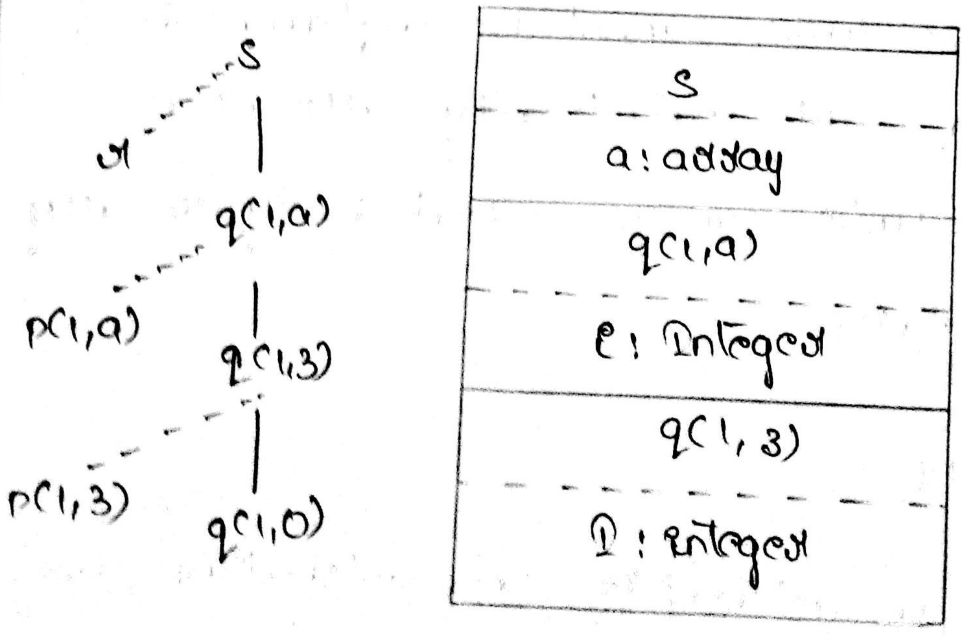
frame for 's'



'r' is Activated

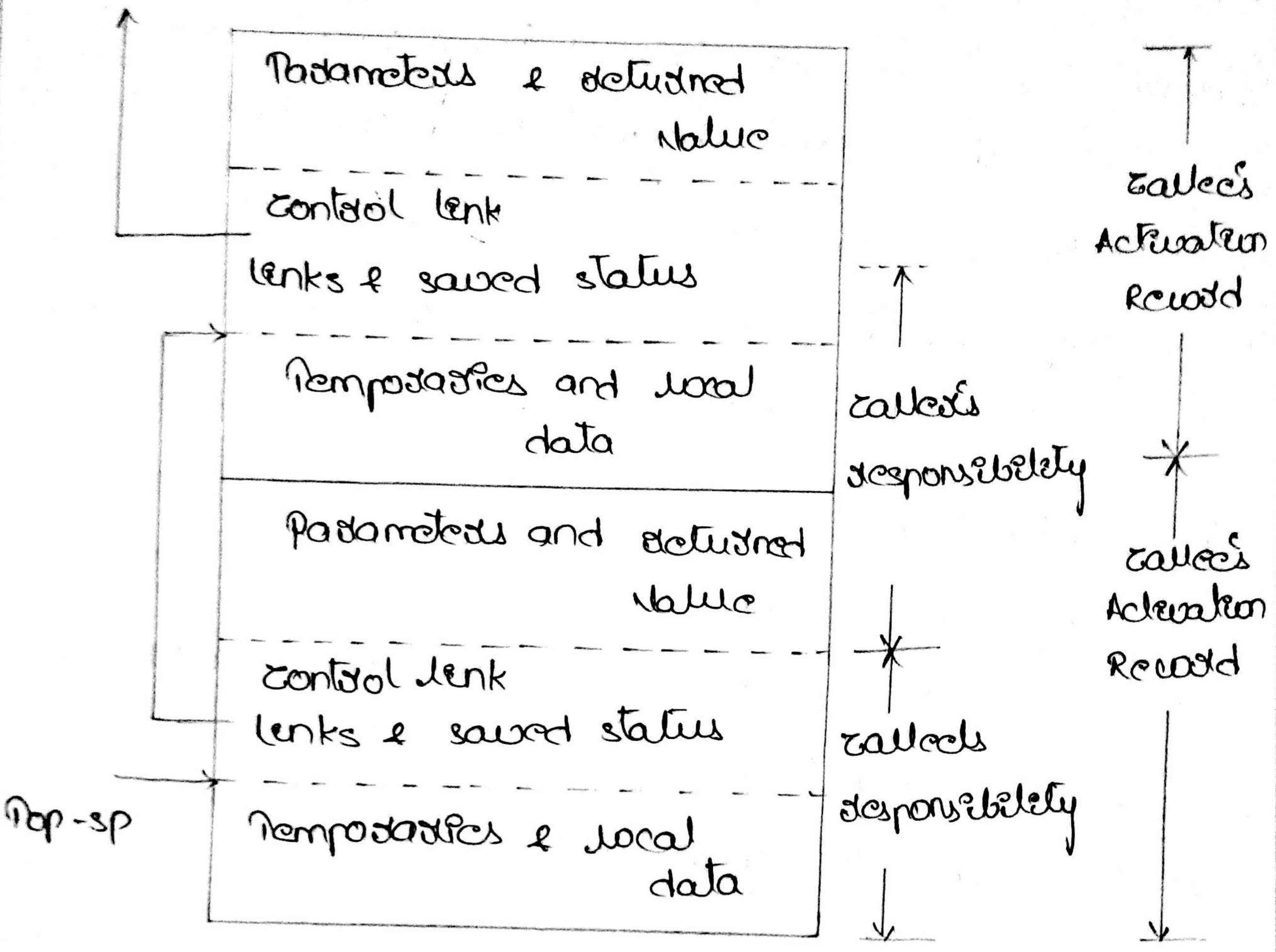


frame for 'r' has been popped and q(1,9) pushed.



control has just returns to q(1,3)

Division of tasks b/w caller & callee:-



Calling sequence:

* procedure calls are Implemented by calling sequence
* A call sequence allocates an activation record & enters the information into its fields.

Return sequence

* Return sequence restores the state of the machine so that the calling procedure can continue execution
* The code in a calling sequence is divided between the calling procedure (the caller) and the procedure it calls (the callee)

During a procedure call the callers activities are :

1. Makes space on the stack for a return value
2. It puts the actual parameters on the stack
3. It sets the static link.
4. It jumps to the called procedure, return address is saved on the stack.

callee's activities

1. sets the dynamic link
2. sets the base of the new activation record.
3. saves the registers
4. Makes space for local variables on the stack.

During a return sequence the caller's responsibility are

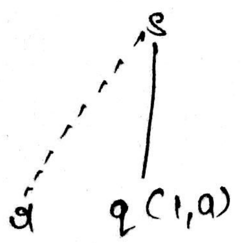
Restores stack pointer to the location containing the returned value.

callees activities

1. Restores the registers
2. sets the base pointer to the activation record of the calling procedure.
3. Returns to the caller.

→ Heap allocation :

comparision of AR on a stack Vs AR in a heap.

Position in Activation Tree	AR on the stack	AR in a Heap	Remark's
	s a: array q(1,9) i: Integer	s a: array r(1,9) i: integer q(1,9) i: Integer	frame for 'r' is popped & q(1,9) is pushed Since 'r' is in heap it is still maintained

* stack allocation cannot be used in following situation

1 situation: The value of local names must be retained when an activation ends.

2nd :- A called Activation out lives its caller

* Differences between heap and stack allocation:

→ Heap allocation retain the activation record even befor after the activation is completed

→ where as in stack allocation it does not retain the activation record after the execution is completed

→ In heap when it is deallocated the left free space is filled up by the heap manager for sum other objects.

* Paramerer passing:

There are two types of parameters. they are.

1. formal parameters

2. Actual parameters

And based on these parameters there are various parameter passing methods, the most common methods are,

i, call by value:

This is the simplest method of parameter passing

→ The actual parameters are evaluated and their r-values are passed to called procedure.

→ The operations on formal parameters do not change the values of actual parameter.

ii, call by reference:

This method is also called as call by address (or) call by location.

→ The L-value, the address of actual parameters is passed to the called routenus activation record.

→ the values of actual parameters can be changed

→ The actual parameters should have an L-value.

iii, copy-restore:

This method is a hybrid between call by value and call by reference. this method is also known as copy-in-copy-out or values result.

→ The calling procedure calculates the value of the actual parameters and it is then copied to activation record for the called procedure.

→ During execution of called procedure, the actual parameters value is not affected.

→ If the actual parameters has L-value then at return the value of formal parameter is copied to actual parameters.

9.), Call by Value & Name:

This is less popular method of parameter passing

→ procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formals.

→ The actual parameters can be surrounded by parenthesis to preserve their integrity

→ The local names of called procedure and names of calling procedure are distinct.

HEAP MANAGEMENT:

There are two techniques used in dynamic memory allocation and those are

1. Explicit allocation
2. Implicit allocation

→ Explicit memory allocation and deallocation is done with the help of some functions such as new and dispose.

→ Implicit memory allocation is done by compiler using run time support packages.

EXPLICIT ALLOCATION:

The explicit allocation can be done for fixed size and variable sized blocks.

Explicit allocation for fixed size blocks:

→ In this technique a free list is used. Free list is a set of free blocks.

→ This list is observed when we want to allocate memory.

→ If some memory is de-allocated then the free list gets appended.

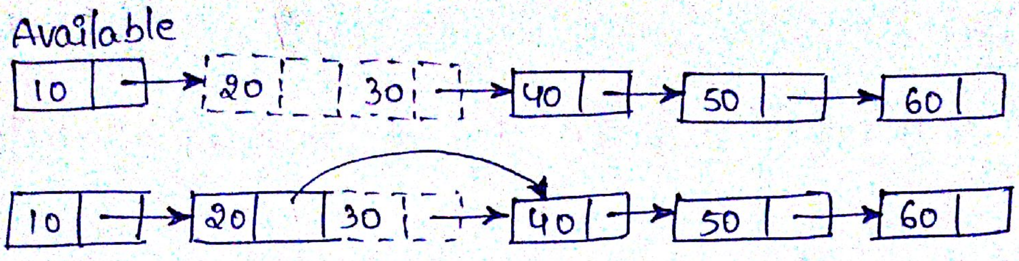
→ The block are linked to each other in a list structure.

→ The memory allocation can be done by pointing previous node to the newly allocated block.

→ Similarly memory deallocation can be done by de referencing the previous link.

→ The pointer which points to first block of memory is called "Available".

→ This memory allocation and deallocation is done using heap memory.



Deallocate block '30'

→ The explicit allocation consists of taking a block off the list and deallocation consist of putting the block back on the list.

→ The advantage of this technique is that there is no space overhead.

Explicit allocation of variable sized blocks:

→ Due to frequent memory allocation and deallocation the heap memory becomes fragmented.

→ That means heap may consist of some blocks that are free and some that are allocated.

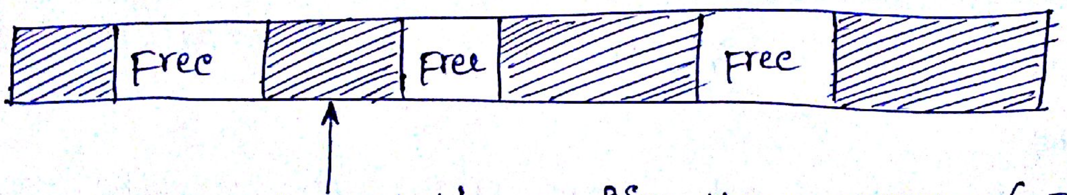


fig: Heap memory (Fragmented)

→ suppose a list of 7 blocks gets allocated and second, forth and sixth block is deallocated then fragmentation occurs.

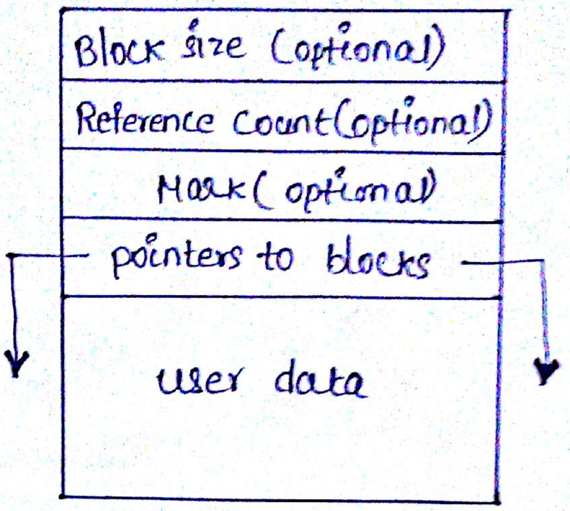
→ Then we get variable sized blocks that are available free.

→ sometimes all the free blocks are collected together to form a large free block. This ultimately avoids the problem of fragmentation.

IMPLICIT ALLOCATION:

→ The implicit allocation is performed using user program and runtime packages.

→ The run time package is required to know when the storage block is not in use.



Block format

→ There are two problems that are faced in implicit memory allocation.

→ First problem is recognizing block boundaries.

→ If the block size is fixed then user data can be easily accessible

→ second problem is to determine which block is in use.

→ There are two approaches used for implicit allocation.

1. Reference count: Reference count is a special counter used during implicit memory allocation.
   → If any block is referred by some another block then its reference count incremented by one.

2. Marketing techniques: this is an alternative approach to determine whether the block is in use or not.
   → In this method, the user program is suspended temporarily and frozen pointers are used to mark the blocks that are in use.

→ sometime bitmaps are used.

→ Bitmaps are used to mark the blocks which are in use.

→ A bit-table stores all the bits.

→ Bit table indicates the blocks which are in use currently.