UNIT - 4

RUNTIME ENVIRONMENT

- Runtime Environment / Runtime storage management deals with variety of issues, such as -

i) Managing the processors stack

ii) Allocation of memory for variables used in the source program

iii) Allocating and deallocating of memory with the help of O.S

iv) Mechanism to pass variable

v) Mechanism used by target program to access variables.

Eg:- If we consider any programming Language -

1. The source program directly go and store in the secondary memory.
2. Compilation also takes place in secondary memory
3. After compilation, compiler demands a free block of memory from operating system in order to store the program in main memory for execution.
4. Operating System allocates free block of memory to the corresponding program.

- All these points collectively known as Runtime Environment.

Storage Organization -

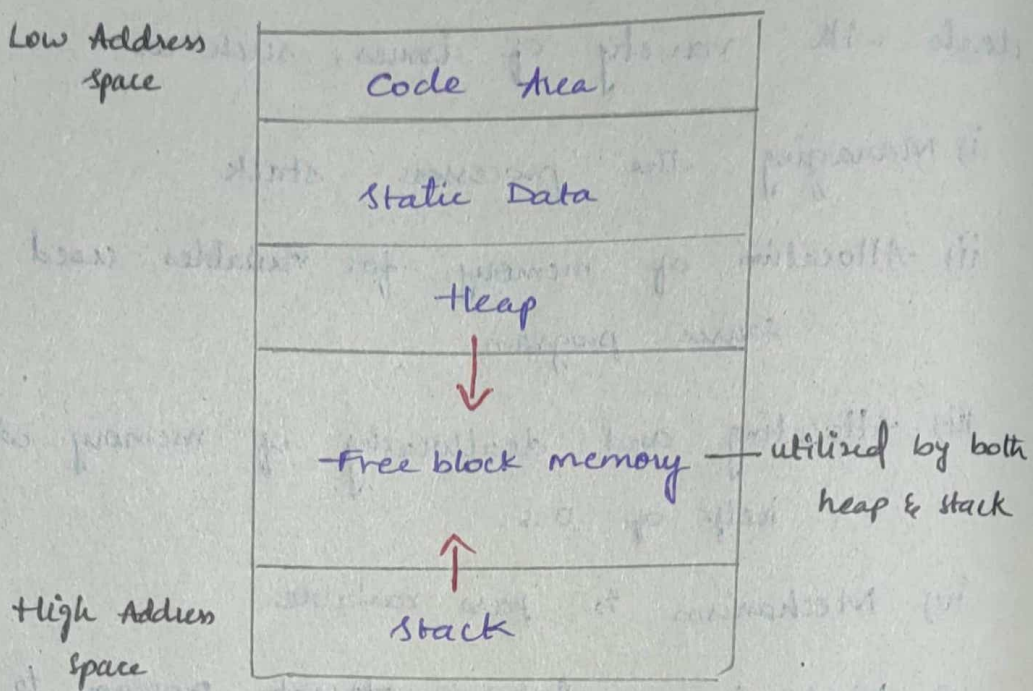


Fig: Subdivision of Runtime Memory management.

1. **Code Area** - It is used to store executable code. (Loader and Linker)

2. **Static Data Area** - It is used to store static variables and global variables.

3. **Heap** - Heap is used to allocate memory at Runtime.

Eg: malloc, calloc, realloc, free } In 'C'

new, delete } In 'Java'

Stack - Stack works on the principle called LIFO (Last - In - First - out).

- Stack is used to store activation record information.

STACK ALLOCATION OF SPACE :

- Almost all compilers uses procedures / functions / methods as a unit of user defined action.

- Activation Tree -

i) Activation - The execution flow of procedure / function is called as activation

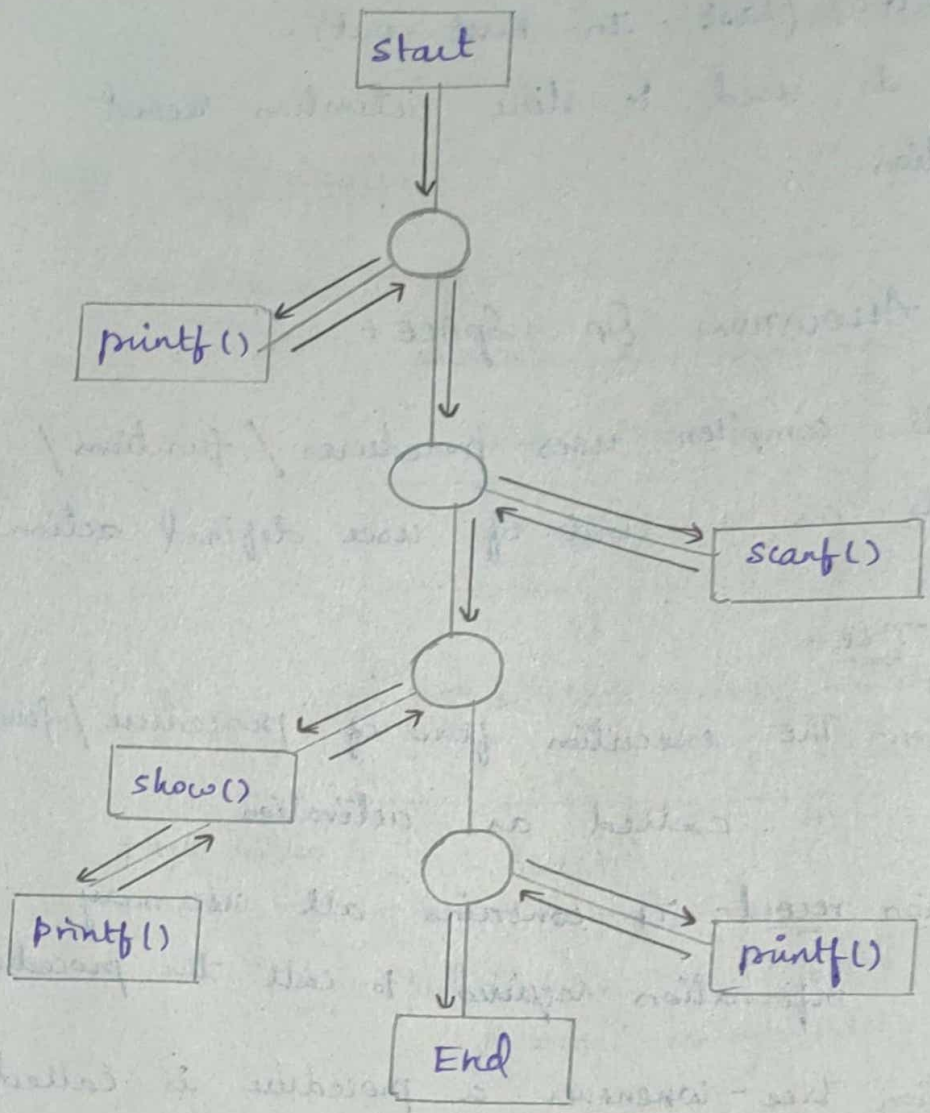
ii) Activation record - It contains all necessary information required to call the procedures

iii) Activation tree - Whenever a procedure is called, an activation record gets created and stored on the top of the stack, is known as control stack

Eg: to represent Activation tree

```
printf ("Enter your name");
scanf ("%s", usname);
show (usname);
printf ("Enter any key");
void show (*user)
{
  printf ("%s", user);
}
```

Activation Tree -



Activation Record structure -

{ Resides in stack }

- Activation Record Contains 7 fields.

- Actual parameters
- Returned values
- control link
- Access / static link
- Saved machine states
- Local variables
- Temporary variables.

a) Actual parameters - It holds the actual parameters of calling function.

b) Returned values - To store the result of function call

c) Control Link - to store the address of calling function.

d) Access Link - To store the local data of called function

e) Saved machine status - To store the address of next instruction.

Eg: Registers
program counter (PC).

f) Local variable - These variables are local to a function

g) Temporary variables - are needed during Expression evaluation.

20/12/22

Examples of Activation Records:

i) C Sample procedure code without Recursion-

Code:

```
void main()
{ int p;
  - - -
  B(p);
  - - -
}
void B (int x)
{
  int s, t;
```

```
    A(s);
    ---
}
void A(int x)
{
    int y;
    ---
    C(y);
    ---
}
Void C(int q)
{
    ---
    ---
}
```

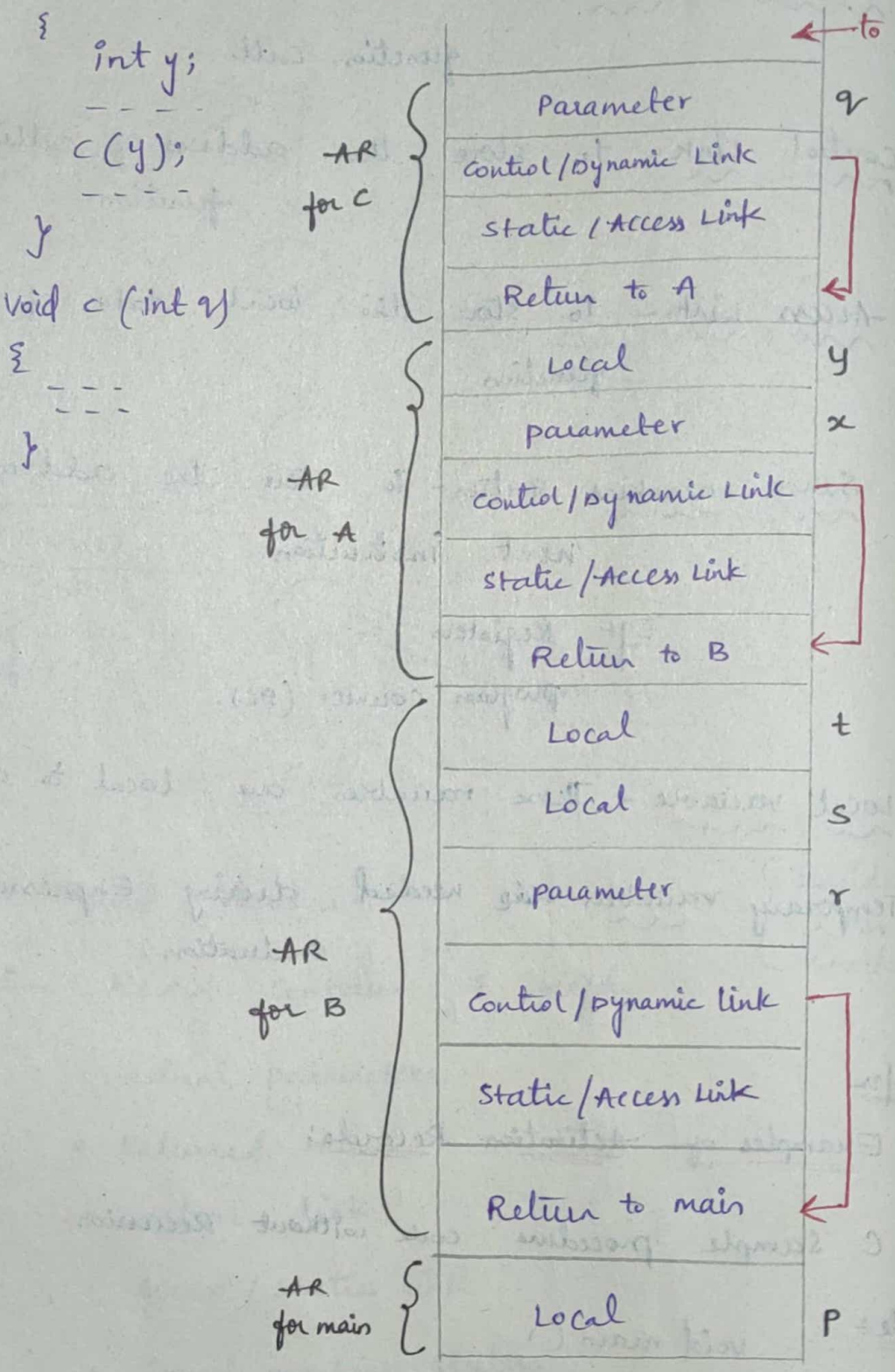


Fig: Stack of Activation Record

Eg:- sample code for Recursion.

code -

```
void main()
{
  int x;
  x = fact(4);
  print(x)
}
int fact(n)
{
 if (n==0) || (n==1) return 1;
 int x=n;
 int y = fact(n-1)
 return x*y;
}
```

22/12/22

Access To Non-Local Data On The Stack

Is a mechanism for finding data used within a procedure, but it doesn't belongs to procedure 'p'.

- Different methods to Access Non Local data on the stack
  1. Data Access without nested procedures.
  2. Issues with nested procedures.
  3. A language with nested procedures.
  4. Nesting Depth
  5. Access Link
  6. Display

i) Data Access without nested procedures-

- In 'c' language it doesn't support nested procedures.
- In 'c', all variables either are declared either in single function (or) outside function (global).
  (local)
- Local variables are declared within the functions.
- Global variables can access all functions.

access to non-local data is done by access link

Ex:

```
int a[10]; // non-local data
int randarray()
{
 ----a----
}
int partition(int x, int y)
{
 ----a----
}
quicksort (int m, int n)
{
 ----------
}
main()
{
 ---a---
}
```

ii) Issues with nested procedures -

- Accessing becomes more complicated when procedures declaration are in nested form.
- Reason - The nested declaration doesn't tell the relative position of activation record at run time.
- possible solution is the usage of access link.

Nested procedure -

```
p()
{
   q()

}
```

(iii) A language with nested procedures-

- Languages that support nested procedures are

  - pascal
  - ALGOL – 60
  - Meta Language

Ex: pascal program with nested procedures-

```
sort (input file , output file);

    var a : array [0....10] of Integer;
    var x : integer
    Procedure readarray ( );
        var i : integer
        begin
            a
        end;
    procedure Exchange (i,j : Integer);
        begin
            i = a[i]
            a[i] = a[j]
            a[j] = i
        end;
    Procedure Quicksort (m,n :integer);
        var v : integer
        var k : integer
        Function partition (y,z : integer) : integer;
            begin
                - - a - - // defined in sort()
                - - v - - // defined in quicksort()
                Exchange (y,z); // defined in sort
                - - - -
            end
    begin ____ end;

begin ____ end;
```

Return value

iv) Nesting Depth -

- If Main procedure nesting depth is 'i'.
- Add i+1 to the next nesting depth.

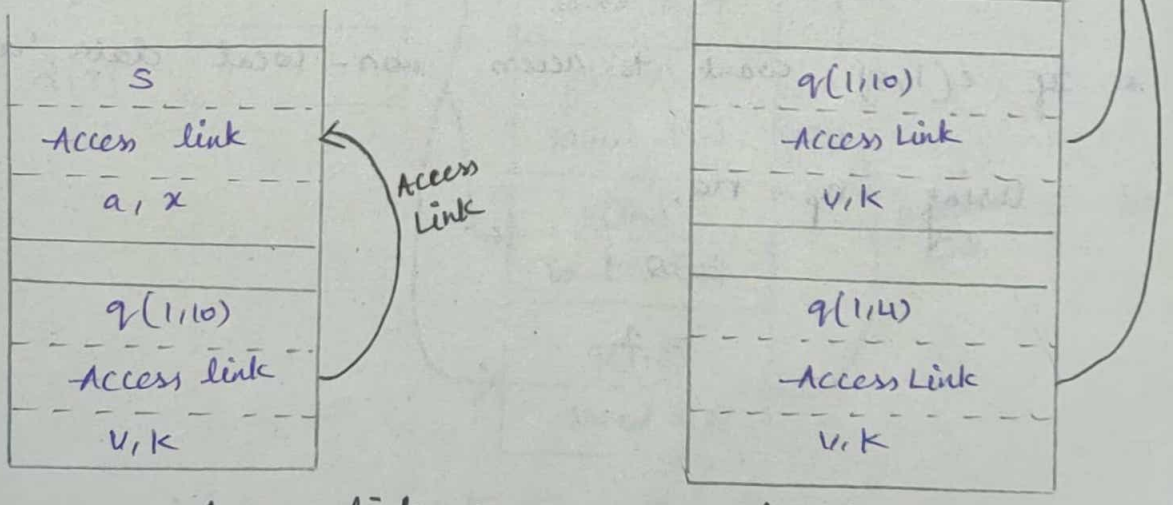
Eg:-

Procedure	Nesting Depth
Sort	1
Readarray	2
Exchange	2
Quicksort	2
partition	3

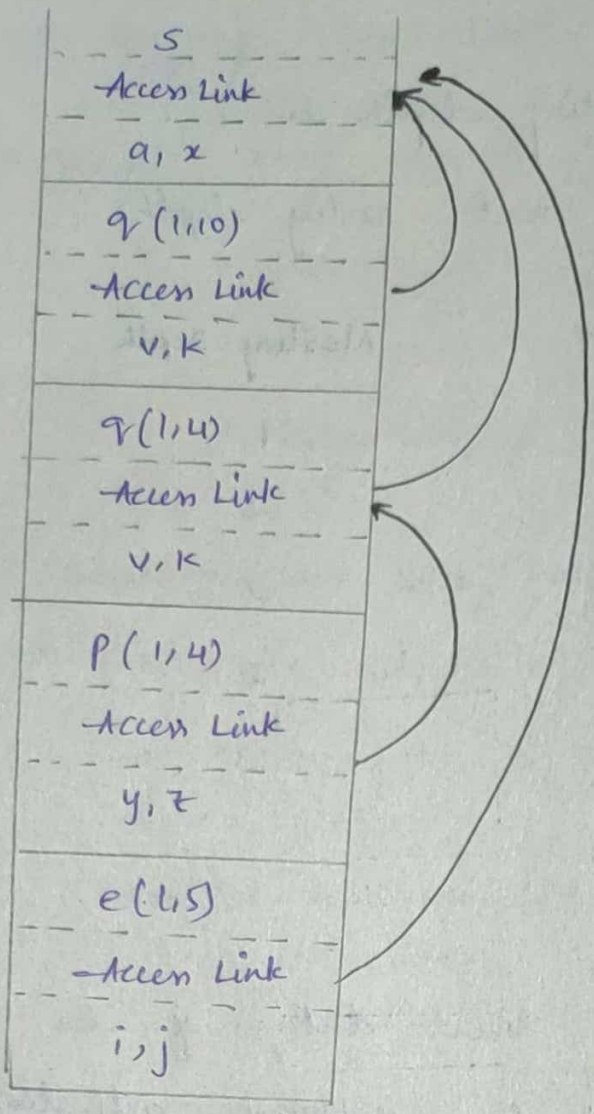
v) Access Link -

- If refers to the local data of the called function, but found in different activation Record.
- formula $n_p - n_a$ helps to reach Activation record of non-local data
- Use Nesting Depth.

Eg:-



Access link for Non-Local Data



Access Link for Non-local data

* If P(1,4) want to access non-local data 'a'?

Sol Using $n_p - n_a$,

$= 3 - 1$

$= 2$

* If e(1,5) want to access non-local data 'a'?

Sol Using $n_p - n_a$,

$= 2 - 1$

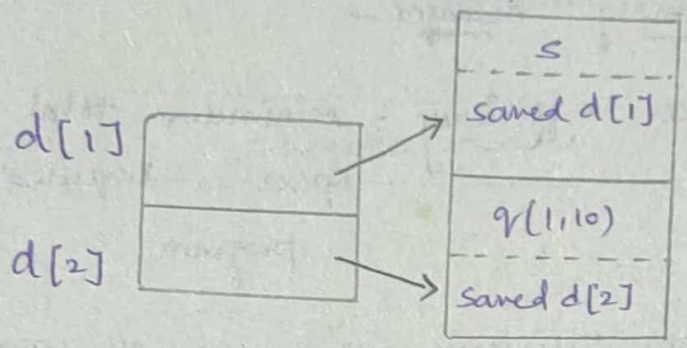
$= 1$

iii) Display -

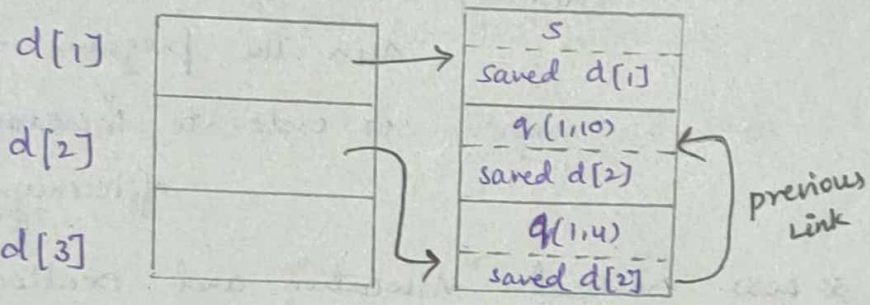
- If nesting depth gets larger then, we have to follow long chain of links to reach non-local data
- An auxilary array 'd' is used.
- Array of pointers is used for each activation record / Nesting depth.

Eg:

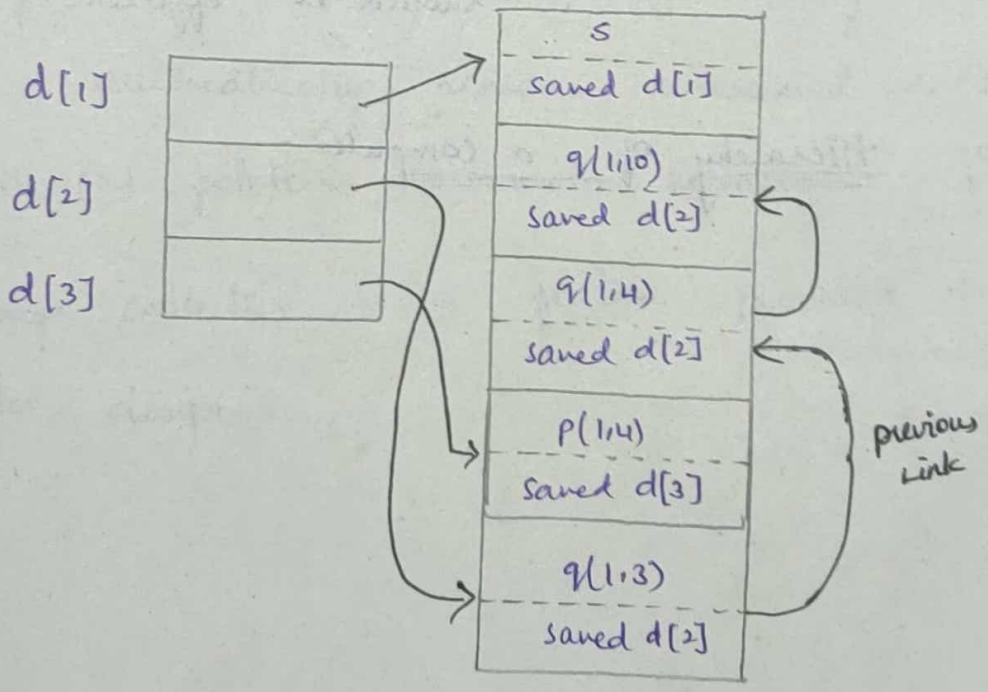
i)



ii)



iii)



Display of Activation Record

Heap Management -

- Heap is a portion of memory that holds data during life time of a program.

- Memory management manager - It is a software called memory manager that manages allocation and deallocation of memory within the heap.

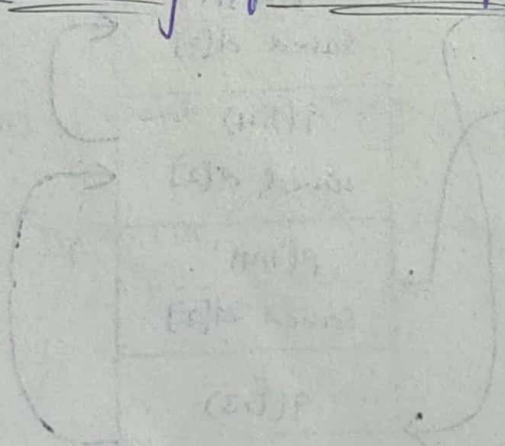
- Properties of memory manager -

  * Space Efficiency - minimum total heap space is required by a program.

  * Program Efficiency - Better use of space to run the program faster in order to increase the efficiency.

  * Low overhead - Allocation and Deallocation should be efficient.

- Memory Hierarchy of a computer -

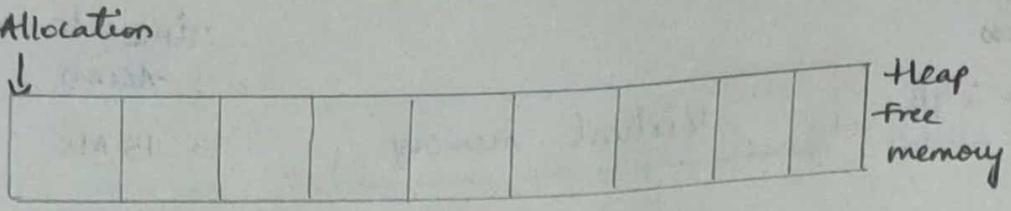


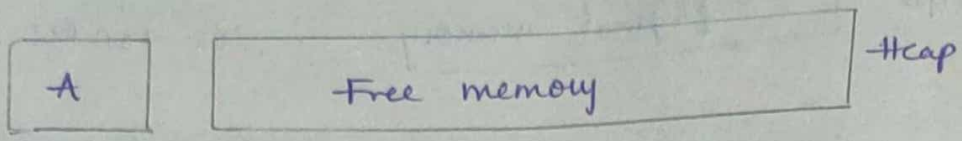
Space to Access		Time to Access
> 2GB	Virtual memory	3-15 MS
128 KB - 2GB	physical memory	100-180 MS
128 KB - 4MB	2nd level cache memory	40-60 MS
16 - 64KB	1st level cache memory	5-10 MS
32 words	Registers	1MS

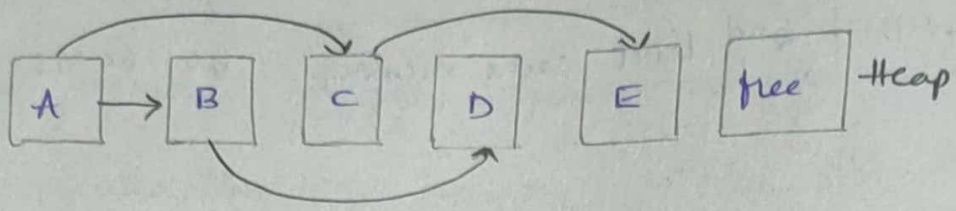
Introduction to Garbage collection -

- Data that cannot be referenced is known as garbage
- Garbage collection is a type of memory management that automatically removes unwanted data objects (or) pointers in main memory.
- Garbage collection is a feature provided by compiler design.

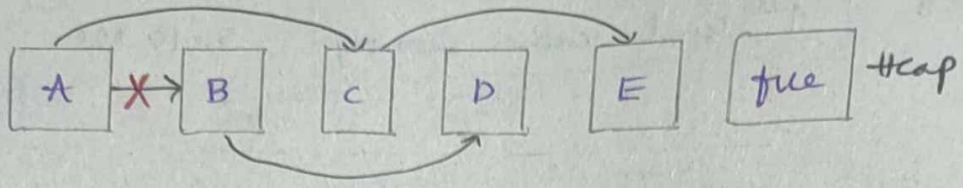
Eg :-







Assume:

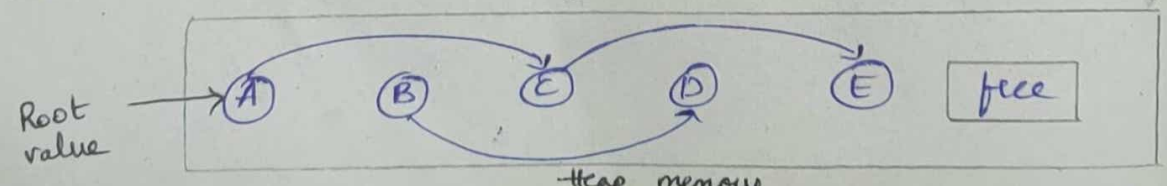


- from Root A, B are not linked. So here, B and D are garbage values.
- Where as A, C, E are alive i.e exist in Heap memory.
- Hence garbage collector works to remove B, D objects from Heap memory.

23/12/22

Introduction to Trace-Based Collection

- Garbage values are identified by considering the Root value and its links. This method is known as 'TRACING'.



- Garbage collection works on three algorithms-

i) Mark sweep garbage collector

ii) Mark and compact garbage collector

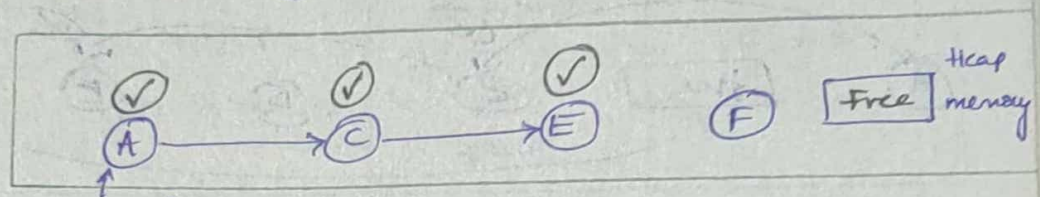
iii) copying garbage collector.

i) Mark Sweep Garbage collector -

Mark - Tracing the live objects

Identify the live object and mark with mark bit i.e "✓".

Eg:-



Eg: mark phase

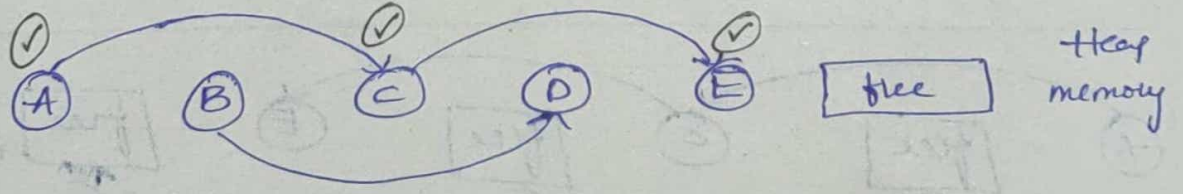
Sweep - Reclaim the garbage ~~collection~~ memory

- Firstly, check whether the objects are marked or not

i) If objects are Marked then Reset it for future use

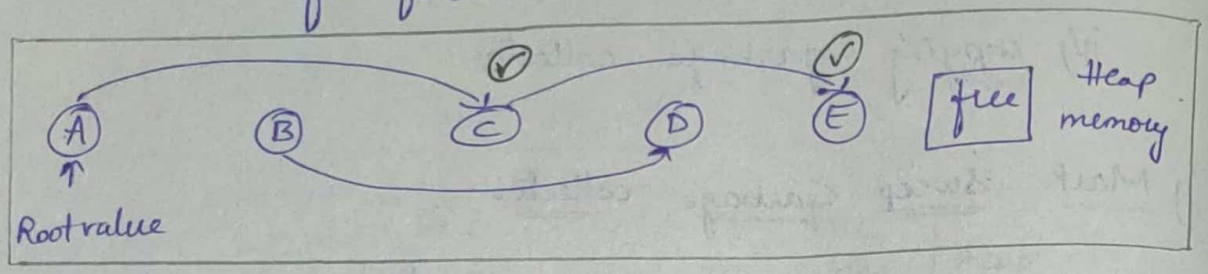
ii) If objects are unmarked then make it as free for future use

Eg:-

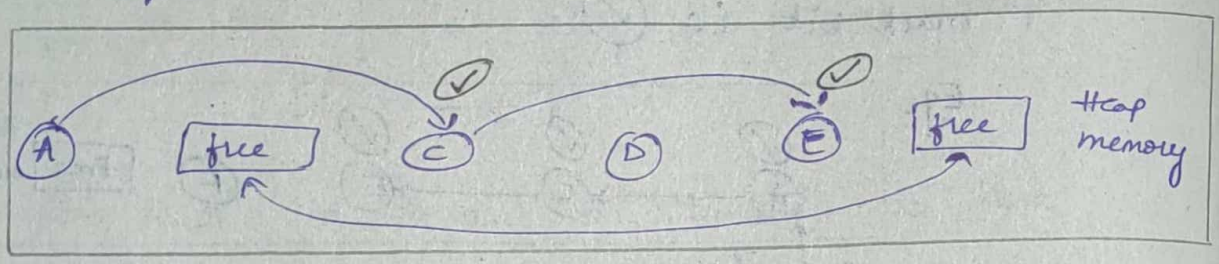


Marking :

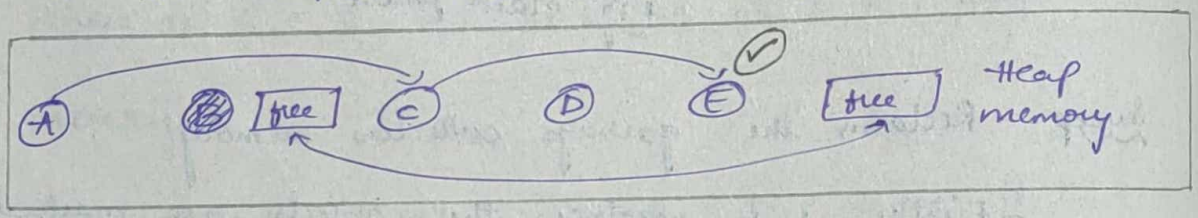
1. 'A' is Root value and it is marked then reset it for future use



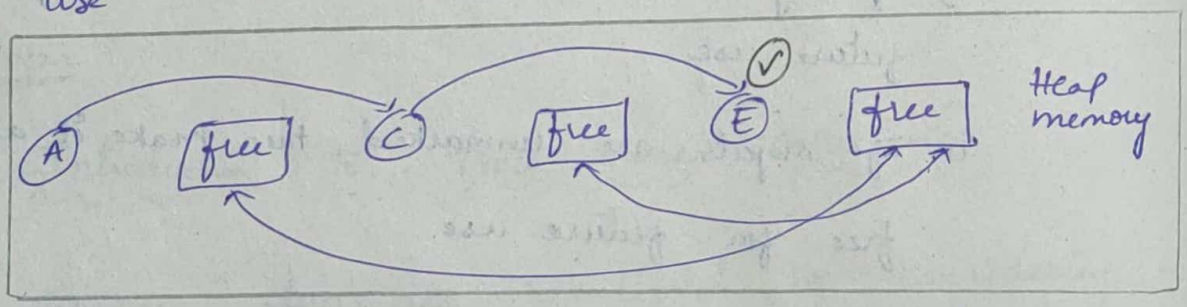
2. 'B' is unmarked and make B as free for future use.



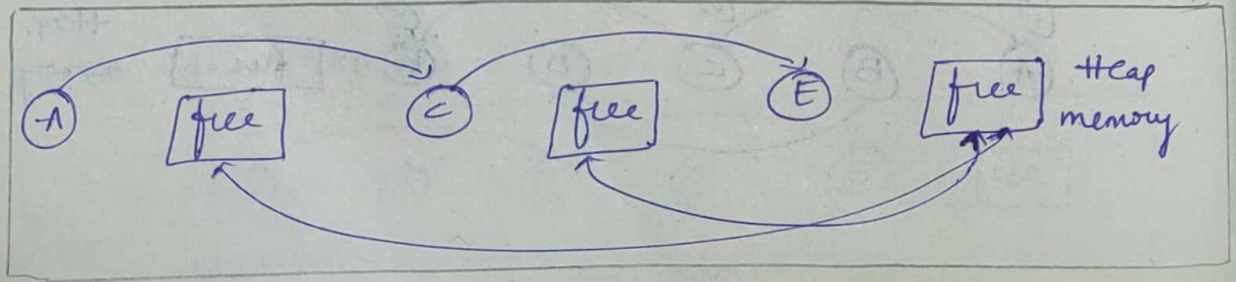
3. 'C' is marked then Reset it for future use.



4. 'D' is unmarked and make 'D' as free for future use



5. 'E' is marked then reset it for future use

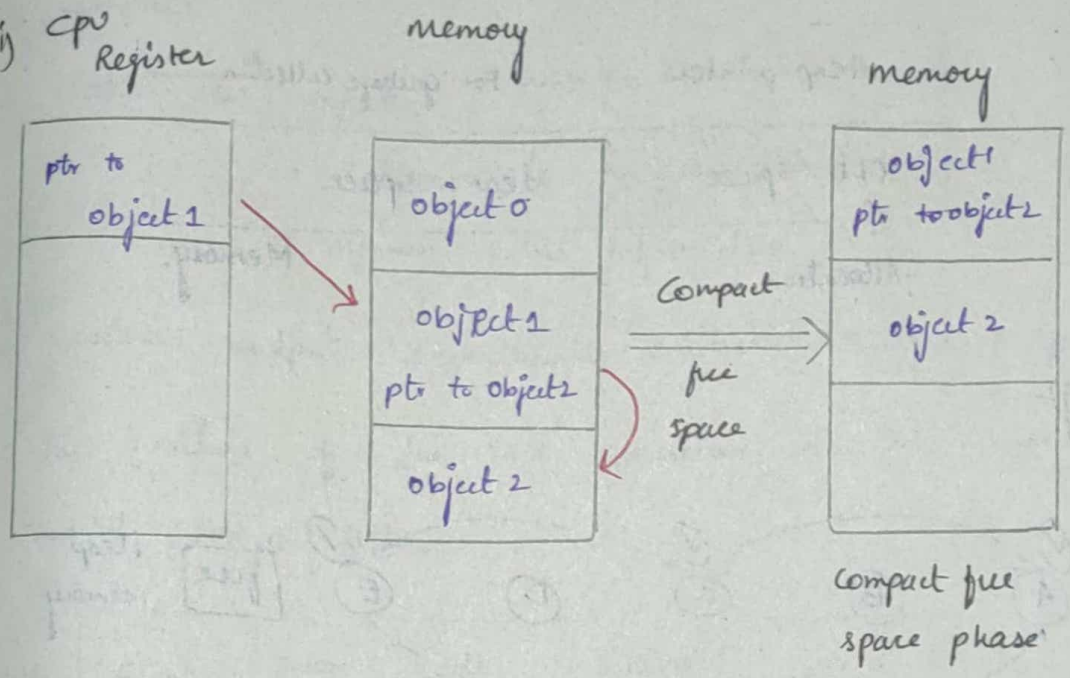


ii) Mark and Compact Garbage collection-

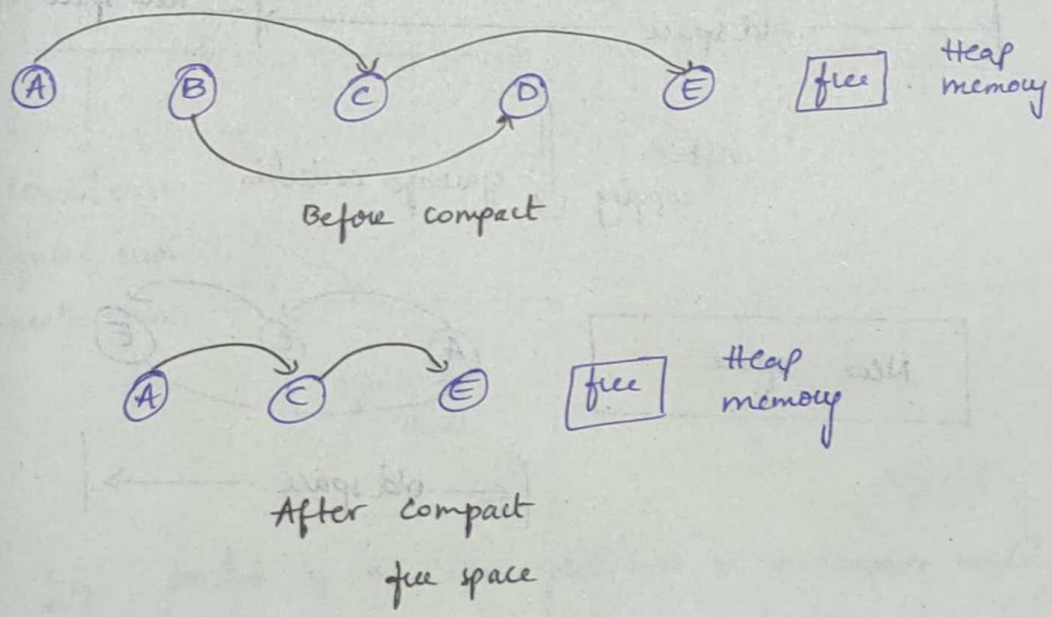
- It is same like - Mark and sweep method.
- However, we compact the free space

Eg:-

i)



ii)

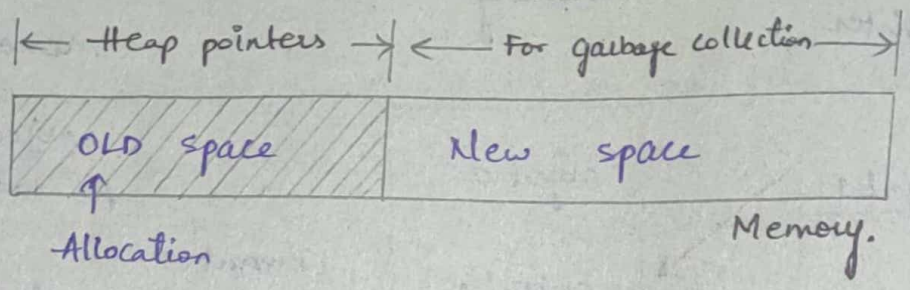


Copying Garbage collection-

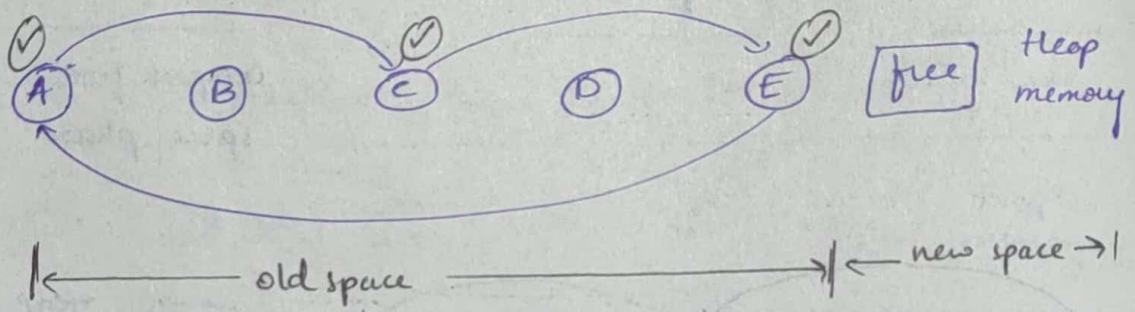
- Here memory is organized in two areas:

i) Old space - Used for Allocation

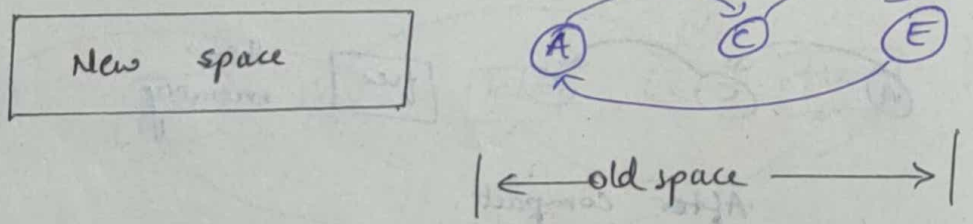
ii) New space - used for garbage collection.



Eg:-



After copying garbage collection



- We need to find all reachable objects as mark and sweep.
- As we find a reachable objects and copy it into newspace

and we have to fix all pointers pointing to it.

2 7/12/22

Introduction to Code Generation -

- The code generation is the final phase of a compiler.
- Takes as input:
  - Intermediate Representation (IR) code
  - symbol table information.
- Produces output : Semantically equivalent Target program.
- The position of the code generation in the compiler model is as follows:

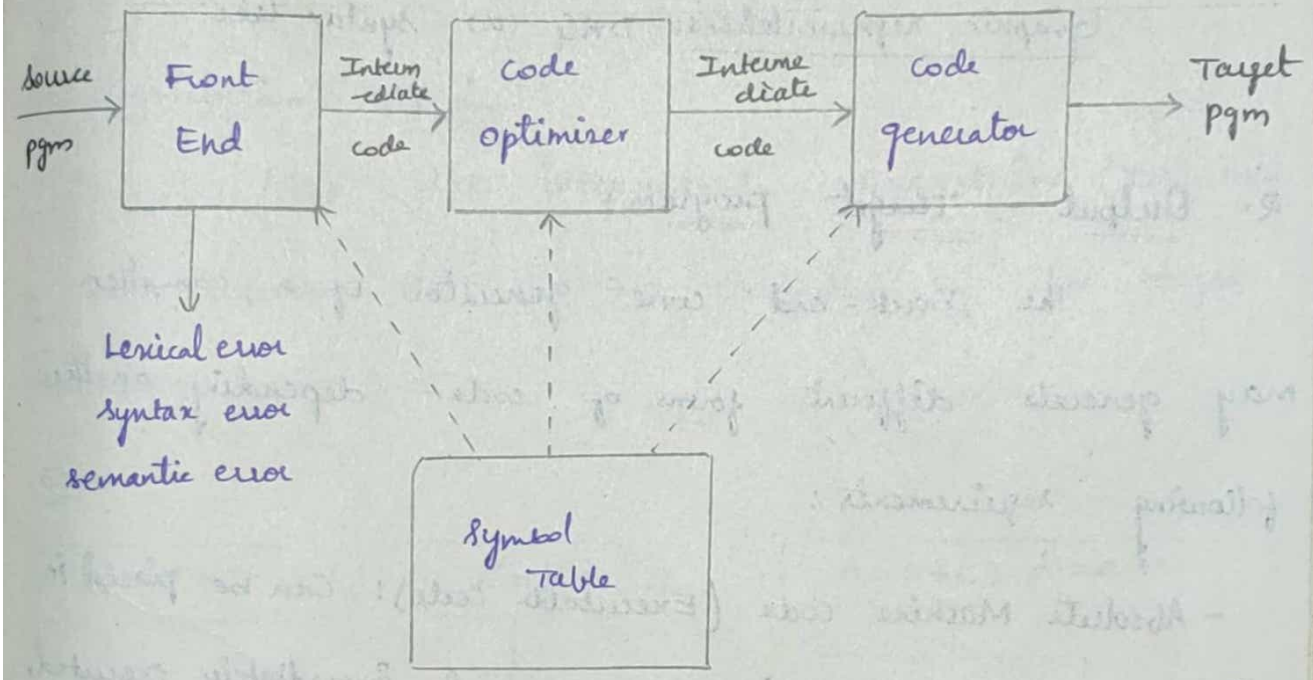


Fig: position of a code generation in a compiler model

Code Generation - Main issues

1. Input to the code generation
2. Output - Target program
3. Memory management

4. Instruction selection
5. Register allocation
6. Choice of Evaluation order.

1. Input to the code generation :

The front-end code generator of a compiler may generate different forms of code- depending on the following Intermediate Representations (IR):

- Three Address code : quadruples, triples and Indirect triples
- Virtual machine : bytecode, stack-machine code
- Linear representations: postfix notation
- Graphic representations: DAG (or) syntax tree

2. Output - Target program :

The back-end code generator of a compiler may generate different forms of code- depending on the following requirements :

- Absolute Machine code (Executable code) : Can be placed in a fixed location in memory and immediately executed

- Relocatable Machine code (object files for linker) : Sub programs can be compiled seperately, then linked together and loaded for execution by linking loader.

- Assembly Language (Facilitates debugging):
  - It makes the proces of code generation easier.
  - Here symbolic instructions are generated and macro facilities of assembler are used to generate code.
- Byte Code form: For interpreters

  Eg: JVM

3. Memory Management -

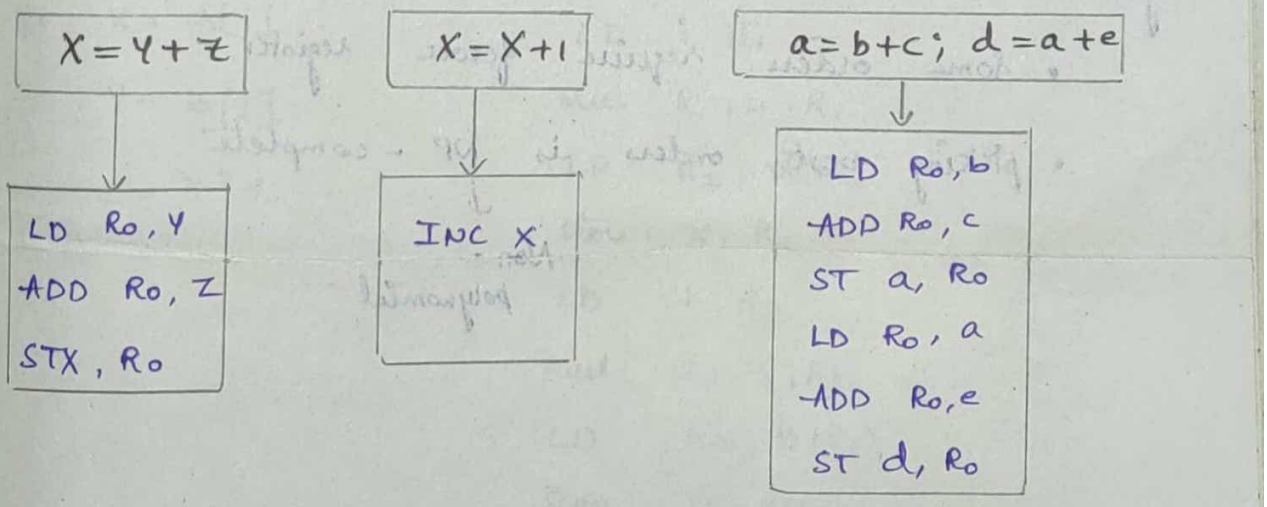
Names in the source program are mapped to addres of data objects in runtime memory (symbol table).

Ex: Symbol table management

4. Instruction selection -

Map the intermediate representation (IR) into a code sequence that can be executed by the target machine.

Ex:



5. Register Allocation -

There are two types of Register allocation. They are

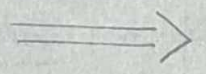
i) Register allocation - select the set of variables that will reside in registers.

ii) Register assignment - pick the specific Register that a variable will resides in.

Ex :-

```
t = a + b
t = t * c
t = t / d
```

Three Address code



```
LD   R1, a
ADD  R1, b
Mul  R1, c
DIV  R1, d
STR  R1, t
```

Target pgm

6. Choice of Evaluation Order :-

The order can affect the efficiency of the target code :

- Some orders require fewer register
- picking best orders is NP - complete
  ↓
  Non - polynomial

Samples of Generated codes:

1. B = A[i]

```
LD   i, R1        // R1 = i
Mul  R1, 4, R1    // R1 = R1 * 4
LD   R2, A(R1)    // R2 = (A=R1)
Store B, R2       // B = R2
```

Each element of array 'A' is of 4 bytes long.

2. X[j] = y

```
LD   R0, y        // R0 = y
LD   R1, j        // R1 = j
MUL  R2, 4, R1    // R2 = 4 * R1
ST   X(R2), R0    // x[R2] = R0
```

3. x = *p

```
LD   R1, p        // R1 = p
LD   R2, 0(R1)    // R2 = *p
ST   x, R2        // x = R2
```

4. x = a[i]
   y = b[j]
   z = x * y

```
LD   i, R1
Mul  R1, 4, R1
LD   R2, a(R1)
store x, R2
LD   i, R3
Mul  R3, 4, R3
LD   R4, b(R3)
Store 4, R4
Mul  X, 4
STR  Z, X
```

Target Machine Language

The target machine and its instruction set is a prerequisite for designing a good code generator.

Here, we shall use as a target language assembly code for a simple computer that is representative of many register machines.

Target machine model is a three Address machine with Load and Store operation,
Computation operation,
jump operators.

Computer is byte-addressable with 'n' general purpose registers like $R_0, R_1, \ldots R_n$.

There are different kinds of op codes and operations.

Op codes are -

1. MOV (Move content to destination from source).
2. ADD (add content of source to destination)
3. SUB (subtract content of source from destination)

etc

Operations are -

1. Load operation - Assignment : dst = addr

LD dst, addr

Ex:- LD R, X // R=X

2. Store operation- Assignment $X = R$

ST dst, src

Ex:- ST X, R_1 // $X = R_1$

3. Computation Operation -

op, dst, src1, src2

Eg:- SUB R_1, R_2, R_3 // $R_1 = R_2 - R_3$

4. Jump operations -

1. Unconditional jump - BR L

2. Conditional jump - BLTZ R_1, L

(Branch on Less than zero)

Eg:-

$x = y - z$

LD R_1, y
LD R_2, z
SUB R_1, R_1, R_2
ST x, R_1

if $x < y$ goto L

LD R_1, x
LD R_2, y
SUB R_3, R_1, R_2
BLTZ R_1, L

Addressing Modes-

Different ways in which location of an operand can be specified in the instruction.

Mode	Form	Address	Added cost
Absolute	M	M	1
Register	R	R	0
Indexed	c(R)	c + contents (R)	1
Indirect register	*R	contents (R)	0
Indirect indexed	*c(R)	Contents (c + contents (R))	1
Literal	#c	NA	1

Eg:- Three Address instruction $b = a[i]$.

```
LD   R1, i          // R1 = i
MUL  R1, R1, 8      // R1 = R1 * 8
LD   R2, a(R1)      // R2 = contents (a + contents (R1))
ST   b, R2          // b = R2
```

Eg:- $a[j] = c$

```
LD   R1, c          // R1 = c
LD   R2, j          // R2 = j
MUL  R2, R2, 8      // R2 = R2 * 8
ST   a(R2), R1      // contents (a + contents (R2)) = R1
```

Eg: $x = *p$

```
LD R1, p        // R1 = p
LD R2, 0(R1)    // R2 = contents(0+contents(R1))
ST x, R2        // x = R2
```

Eg: $*p = y$

```
LD R1, p        // R1 = p
LD R2, y        // R2 = y
ST 0(R1) R2     // Contents(0+contents(R1)) = R2
```

Eg: ~~if x<y~~

29/12/22

BASIC BLOCK-

- The Basic block is a sequence of consecutive statements which are always executed in sequence without halt or branching.
- The basic block doesnot have any jump statements among them.

Eg:

* $a = b+c+d$

Three Address code

$t_1 = b+c$ ✓

$t_2 = t_1+d$

$A = t_2$

* if A < B then 1 else 0

1. If (A<B) goto (4)
2. T1 = 0
3. goto (5)
4. $T_1 = 1$

✗

Rules for partitioning into Basic Blocks :

After an intermediate code is generated for the given code, we can use the following rules to partition into basic blocks -

Rule 1 - Determine the leaders :

a) The first statement is a Leader.

b) Any target statement of conditional (or) unconditional goto is a Leader.

c) Any statement that immediately follow a goto is Leader.

Rule 2 - The basic block is formed starting at the Leader statement and ending just before the next leader statement appearing.

Flow Graph :

A flow graph is a directed graph in which the flow control information is added to the basic blocks -

Rules - a) The basic blocks are the nodes to the flow graph.

b) the block whose leader is the first statement is called initial block.

c) There is a directed edge from block B1 to block B2 if B2 immediately follows B1 in the given sequence,

can say that B1 is a predecessor of B2.

Examples of Basic Block and *flow graph -

***

1)* Consider the following code -

```
Prod = 0;
i = 1;
do
{
   prod = prod + a[i] * b[i];
   i = i + 1;
} while (i<=10);
```

a) Compute Three Address code

b) Compute the basic block and draw flow graph.

Sol:

a) Three Address code -

1. prod = 0
2. i = 1
3. $T_1 = 4 * i$
4. $T_2 = a[T_1]$
5. $T_3 = 4 * i$
6. $T_4 = b[T_3]$
7. $T_5 = T_2 * T_4$
8. $T_6 = T_5 + prod$
9. prod = T_6
10. $T_7 = i + 1$
11. $i = T_7$
12. if (i<=10) goto 3.

Basic Block -

- First statement is a Leader.

  So, prod = 0 is a Leader.

- Also, target statement of a conditional (or) unconditional goto is a Leader.

  So, $T_1 = 4 * i$ is also a Leader.

- Thus, the above generated Three Address code can be partioned into basic block as -

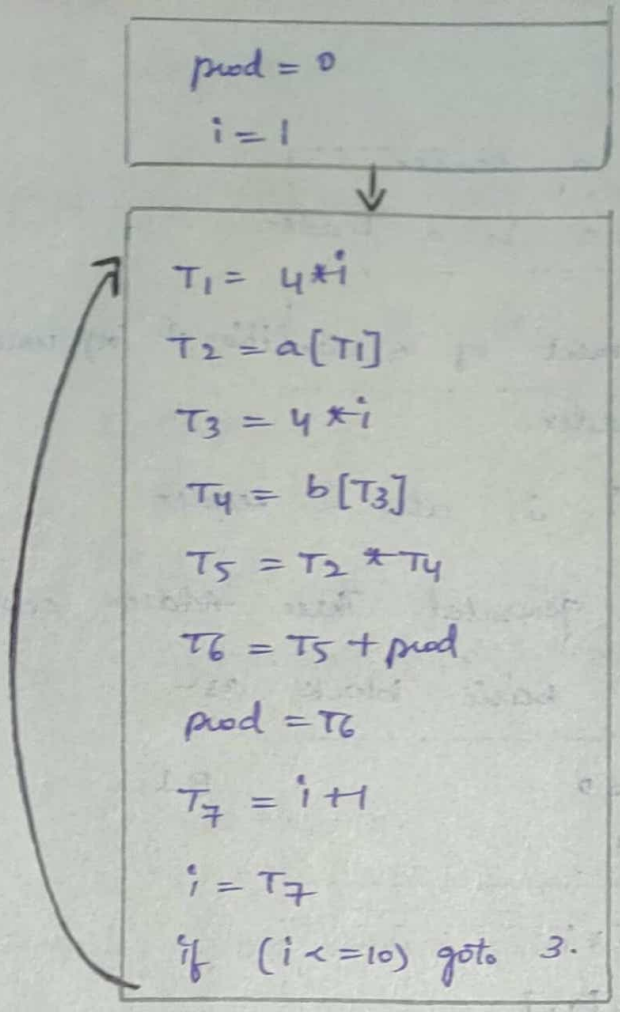
B1

```
prod = 0
i = 1
```

B2

```
T1 = 4 * i
T2 = a[T1]
T3 = 4 * i
T4 = b[T3]
T5 = T2 * T4
T6 = T5 + prod
prod = T6
T7 = i + 1
i = T7
if (i <= 10) goto 3
```

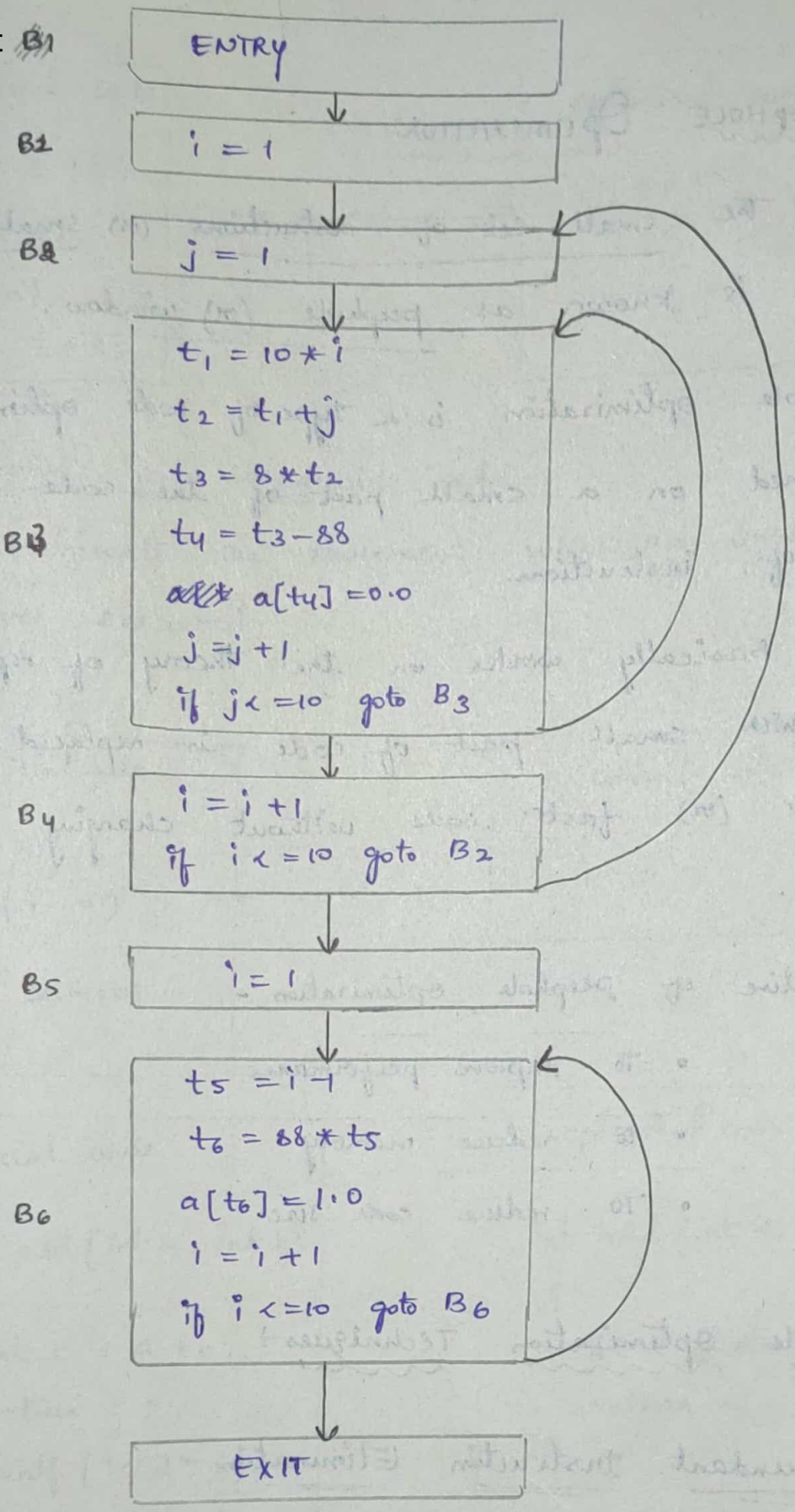
Flow Graph -



2) Consider the following TAC–

1.	i = 1
2.	j = 1
3.	t1 = 10 * i
4.	t2 = t1 + j
5.	t3 = 8 * t2
6.	t4 = t3 - 88
7.	a[t4] = 0.0
8.	j = j + 1
9.	if j <= 10 goto 3
10.	i = i + 1
11.	if i <= 10 goto 2
12.	i = 1
13.	t5 = i - 1
14.	t6 = 88 * t5
15	a[t6] = 1.0
16.	i = i + 1
17.	if i <= 10 goto 13

Consider basic block and Compute Flow graph



30/12/22

PEEPHOLE OPTIMIZATION

The small set of instructions (or) small part of code is known as peephole (or) window.

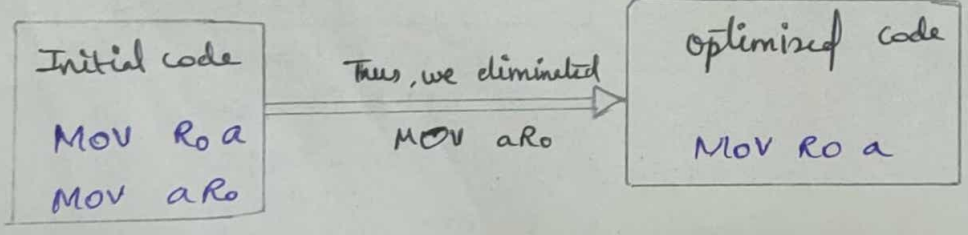
- peephole optimization is a type of code optimization performed on a small part of the code (or) small set of instructions.
- It basically works on the theory of replacement in which small part of code is replaced by shorter (or) faster code without changing the output.
- Objective of peephole optimization -
  - to improve performance
  - to reduce memory
  - To reduce code size.

peephole optimization Techniques :

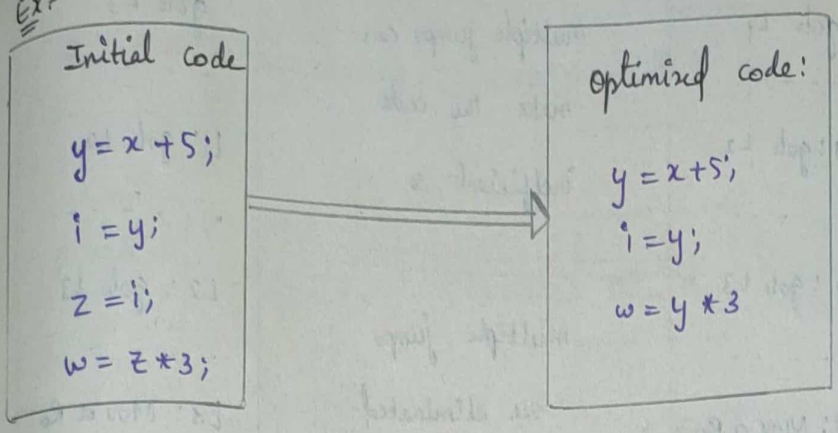
1. Redundant Instruction Elimination -

In this technique, the redundancy is eliminated

Ex :-



Ex:-

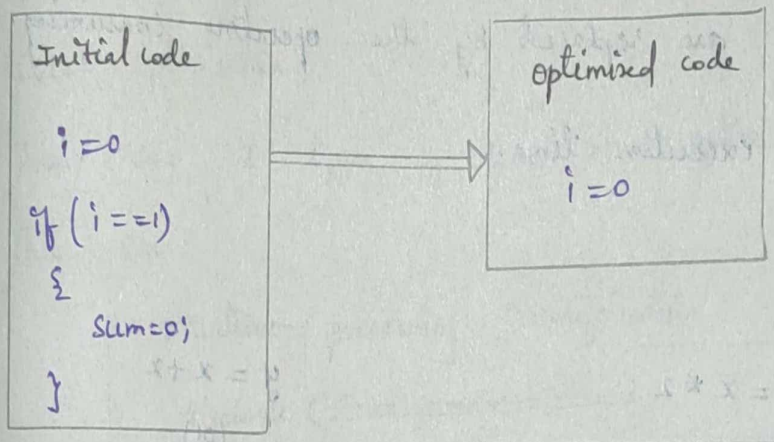


2. Removal of Unreachable code -

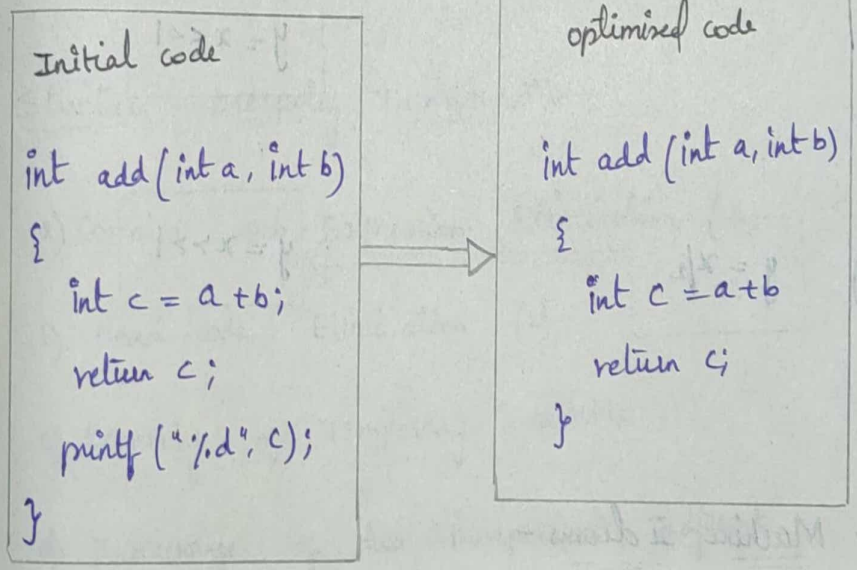
Eliminate the statements which are unreachable i.e Never executed.

Eg:-

i)

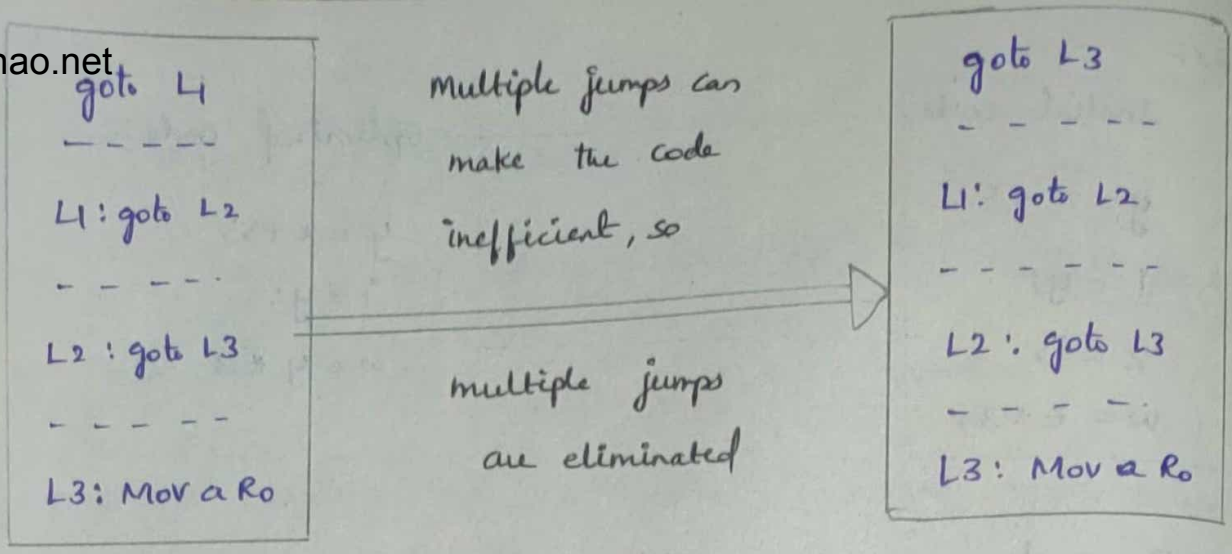


ii)



3. Flow of control optimization -

Using peephole optimization unnecessary jumps can be eliminated.



4. Algebraic Simplifications / Strength Reduction

The operators that consume higher execution time are replaced by the operators consuming less execution time.

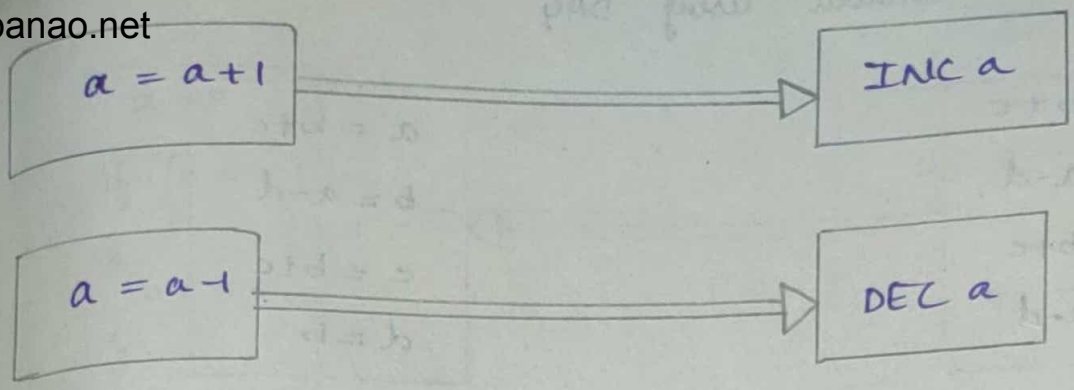
Eg :-

i) $y = x * 2$ ⟹ $y = x + x$ (or) $y = x << 1$

ii) $y = x/2$ ⟹ $y = x >> 1$

5. Machine Idioms -

It is the process of using powerful features of CPU Instructions.



Optimization of Basic Blocks : (Local optimization)

- used to improve the running time of code within each Basic Block.
- Optimization applied on a Basic block is called Local optimization.
- There are 2 types of Basic block optimization. They are
  1. structure - preserving Transformation
  2. Algebraic Transformation

1. Structure - presewing Transformation -

a) Common sub - Expression Elimination (1)

b) Dead code Elimination (2)

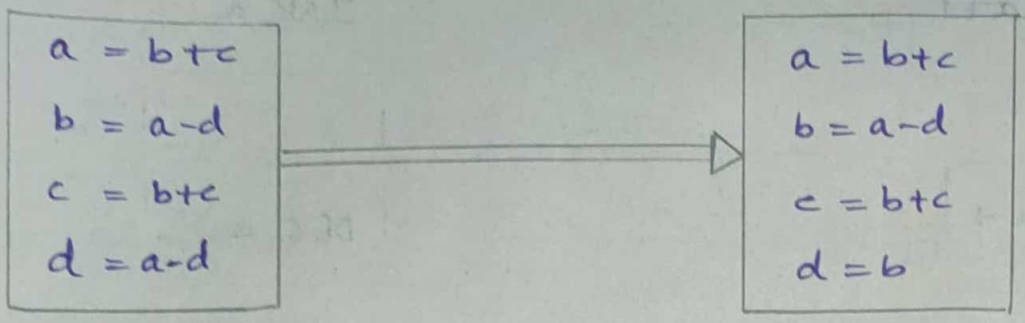
c) Renaming of Temporary variable

d) Interchange of two independent adjacent statements.

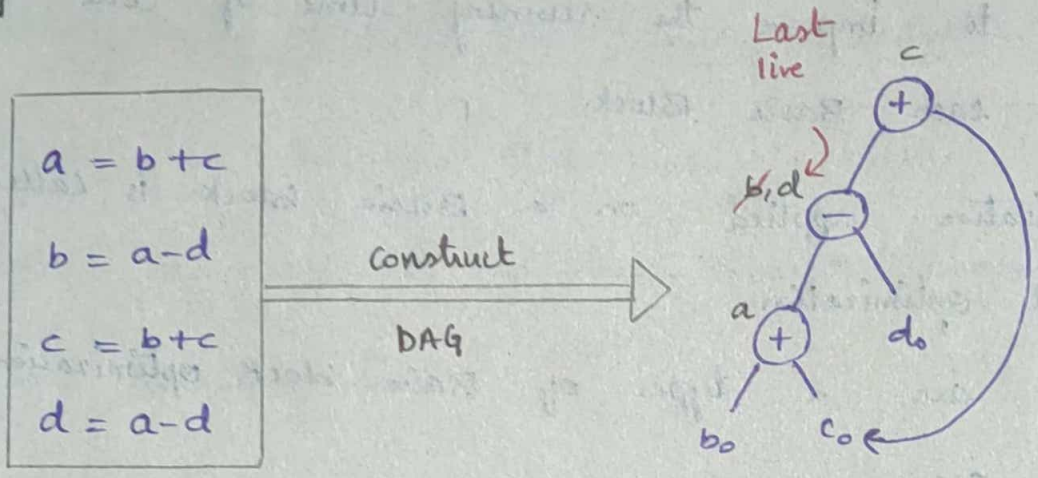
a) Common sub Expression Elimination -

Can be done in 2 ways i) without DAG
ii) with DAG

Eg:- without using DAG



Eg:- using DAG



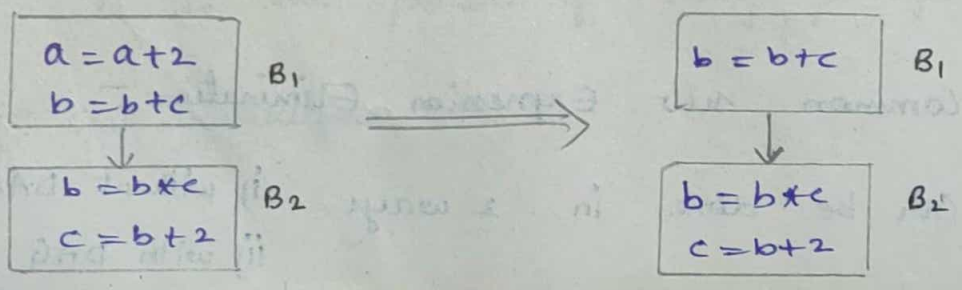
∴ optimized block is,

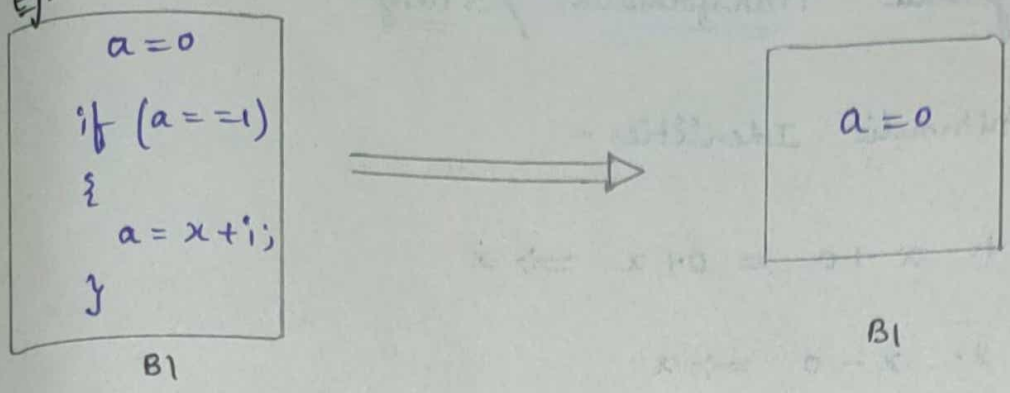
```
a = b+c
d = a-d
c = d+c
```

b) Dead-code Elimination -

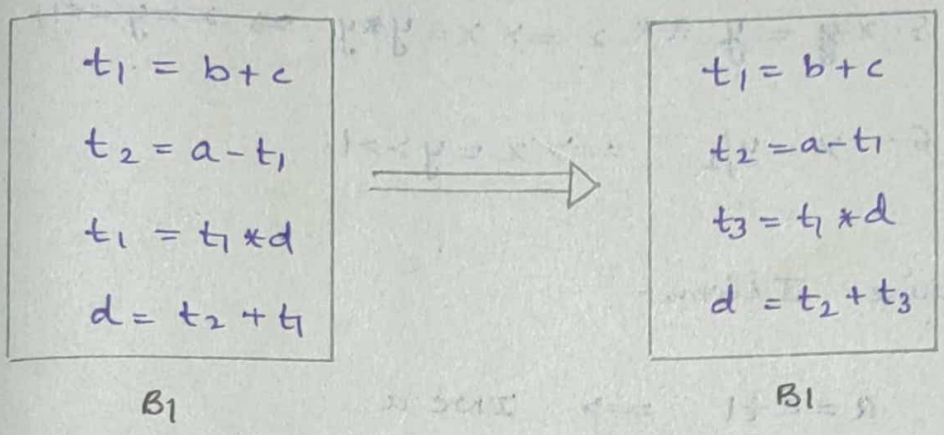
It eliminates the statement which are never executed.

Eg:-

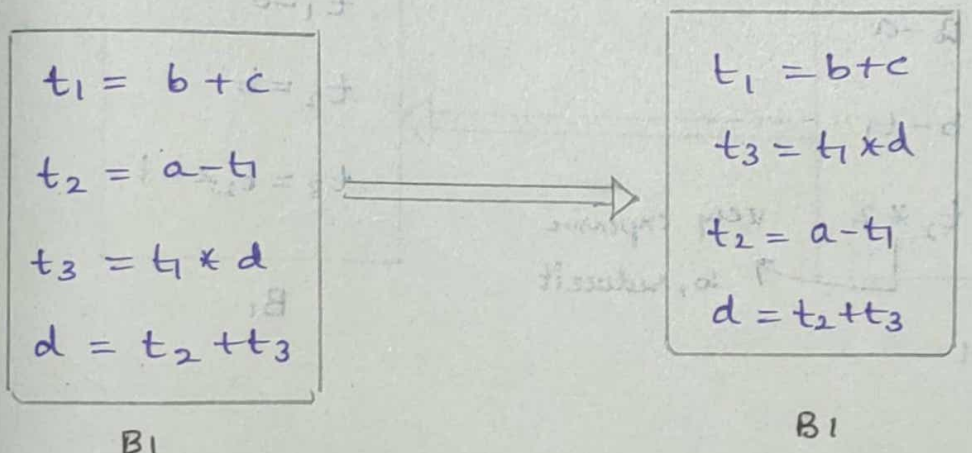




c) Renaming temporary variables -



d) Interchange of two - independent adjacent statements -



2. Algebraic Transformation / Strength Reduction

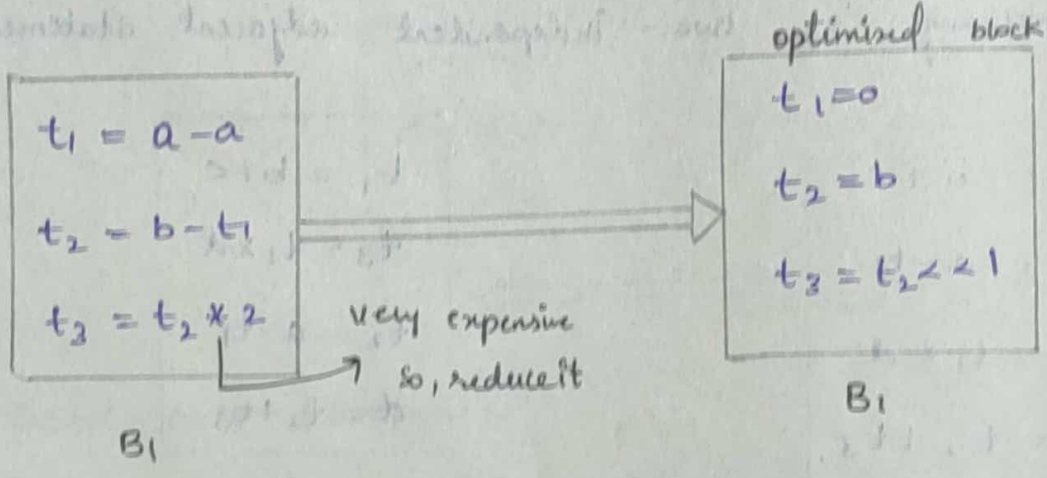
a) Arithmetic Identities -

1. $x + 0 = 0 + x \Rightarrow x$
2. $x - 0 \Rightarrow x$
3. $x * 1 = 1 * x \Rightarrow x$
4. $x / 1 \Rightarrow x$
5. $x * y = y * x \Rightarrow x = y * y$ (as $x == y < < 1$)
6. $x = y / 2 \Rightarrow x = y >> 1$.

b) Machine Idioms -

$a = a + 1 \Rightarrow$ INC a

$a = a - 1 \Rightarrow$ DEC a



UNIT - 4

Dynamic programming Code - Generation

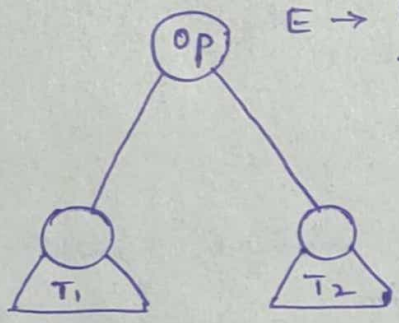
It generates code for any machine with 'r' interchangeable registers $R_0, R_1, R_2, \ldots R_{r-1}$ and load, store and operation instruction.

1. Contiguous Evaluation -

Partitions the optimal code for an expression into optimal code for sub-expressions

i.e $E \rightarrow E_1 + E_2$

- It evaluates an expression contiguosly



$E \rightarrow E_1$ op E_2
T_1 T_2

2. The Dynamic programming Algorithm-

It contains three phases.

i) compute bottom-up for each node 'n' of the expression tree 'T'.

ii) Traverse T, using the cost vector. (For subtree cost)

(iii) Traverse each tree using cost vector and associated instructions (for target code).

```
LD   Ri , Mj
op   Ri , Ri , Rj
op   Ri , Ri , Mj
LD   Ri , Rj
ST   Mi , Rj
```

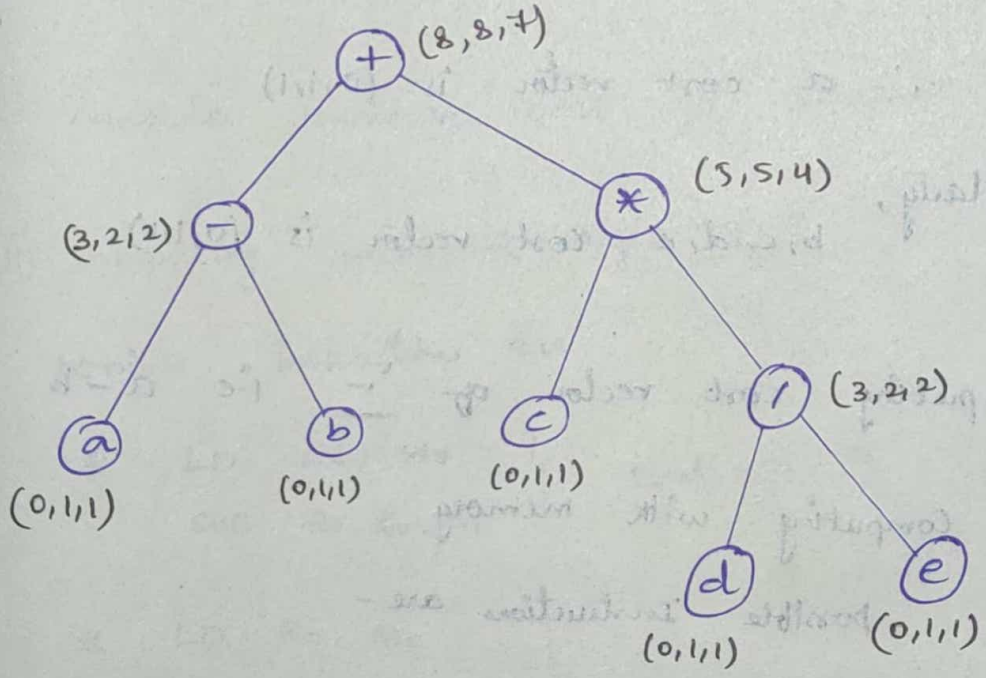
Assume Every instruction cost is 1

Machine instructions

Eg:- Generate optimal code for the following expression (a-b) + c * (d/e) using dynamic programming algorithm.

Ans Given Expression is, (a-b) + c * (d/e)

Step1- Draw syntax tree



to compute cost vector?

1. Computing cost vector for Leaf Node -

i) Moving variable into memory

ii) Moving variable into 1 register.

iii) Moving variable into 2 registers.

From Example,

compute Leaf node 'a' cost vector

i) Moving 'a' into memory - 0 [∵ a is already in the memory]

ii) Moving 'a' into 1 Register - 1

```
LD R0 , a
```

iii) Moving 'a' with 2 registers - 1

```
LD R0, a
```

∴ a cost vector is (0,1,1)

Similarly, b, c, d, e cost vector is (0,1,1).

2. computing cost vector of '-' i.e a - b

i) Computing with memory

possible Instructions are -

```
* LD   R0, M0   (∵a)
  SUB  R0, R0, M1  (∵b)
  ST   M , R0
```

} cost = 3 ✓

* LD R0, M0 (∵a)
  LD R1, M1 (∵b)
  SUB R0, R0, R1
  ST M, R0

} cost = 4

NOTE - Consider minimum cost for memory

ii) computing with 1 register -

possible instructions are -

* LD R0, M0(a)
  SUB R0, R0, M1(b)

} cost = 2 ✓

* LD R0, M0
  LD R1, M1
  SUB R0, R0, R1
  ST M1, R0

} cost = 4 (∵ R0 is free)

NOTE - Consider minimum cost for 1 Register.

iii) computing with 2 registers -

possible instructions are

* LD R0, M0
  SUB R0, R0, M1

} cost = 2 ✓

* LD R0, M0
  LD R1, M1
  SUB R0, R0, R1

} cost = 3

NOTE - Consider minimum cost for 2 Registers.

∴ (-) cost reacts is

∴ '-' cost vector is (3,2,2)

Similarly, '/' is also (3,2,2).

3. Compute the cost vector for Root-

i) Determine the minimum cost for 1 Register

ii) Determine the minimum cost for 2 Registers.

iii) Determine the cost of with memory

Ex - ADD R0, R0, M → Matches the Root

i) Minimum cost of Evaluating root with 1 register

- Minimum cost of right subtree into memory
- Minimum cost of left subtree into 1 Register.
- Add 1 for ADD R0, R0, R1

i.e cost = 5 + 2 + 1

= 8

ii) Minimum cost of Evaluating root with 2 registers

- minimum cost of right subtree with 2 register
- minimum cost of left subtree with 1 register
- Add 1 for ADD R0, R0, R1

i.e cost = 4 + 2 + 1

= 7

iii) Minimum cost of evaluating root with memory.

- minimum cost of rightsubtree into memory
- minimum cost of left subtree with 2 registers
- Add 1 for ADD R0, R0, M

i.e cost = 5 + 2 + 1

= 8

∴ '+' cost vector is (8,8,7)

4. Compute the cost vector for '*'

i) memory

- minimum cost of right subtree into memory
- minimum cost of left subtree with 2 registers
- Add 1

i.e cost = 3 + 1 + 1

= 5

ii) 2 Registers

- minimum cost of right subtree with 2 registers
- minimum cost of left subtree with 1 register
- Add 1

i.e cost = 2 + 1 + 1

= 4

iii) 1 Register

- minimum cost of rightsubtree into memory
- minimum cost of left subtree into 1 Register
- Add 1

i.e cost = 3 + 1 + 1

= 5 ∴ (5,5,4)

Optimal code for the given expression

$(a-b) + c * (d/e)$ is generated as follows during dynamic programming.

```
LD   R0, c
LD   R1, d
DIV  R1, R1, e
MUL  R0, R0, R1
LD   R1, a
SUB  R1, R1, b
ADD  R1, R1, R0
```

6/1/23 UNIT 5 –

LOOPS IN FLOW GRAPH –

Loop is a collection of nodes in flow graph such that

i) all such nodes are strongly connected.

ii) collection of nodes has unique entity. entry.

iii) The loop that contains no other loop is called inner loop.

- Some common technologies being used for loop in flow graph –

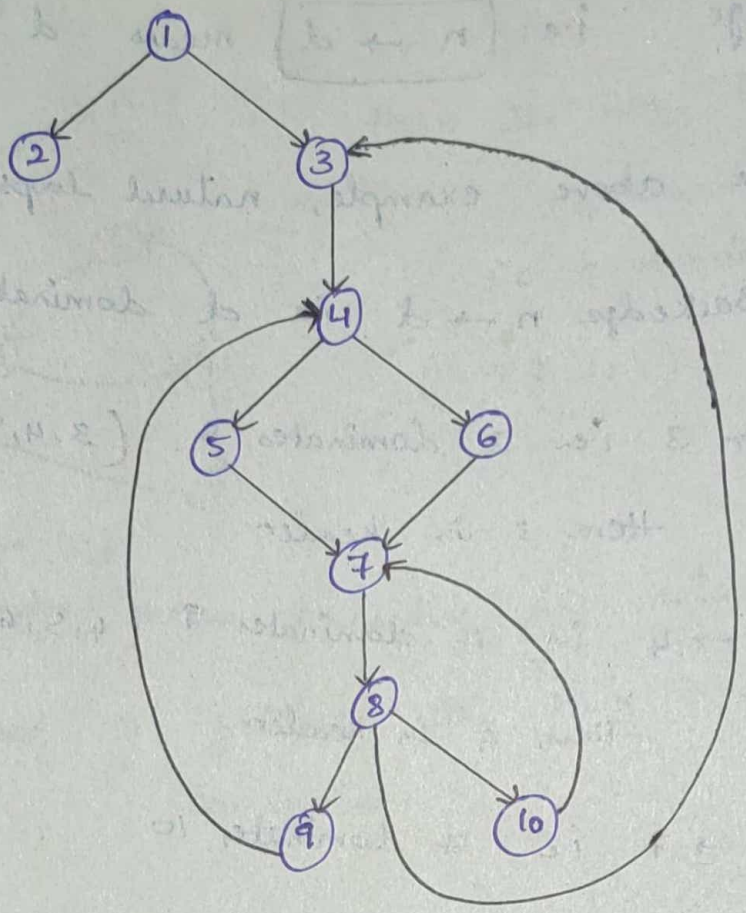
1. Dominators –

Every initial node dominates all the

remaining nodes in the flow graph.

- Similarly, every node dominates itself.

Ex:-



Nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Node 1 - Dominates all the nodes, in the flow graph (itself also)

Node 2 - Dominates itself

Node 3 - Dominates all nodes except 1 and 2.

Node 4 - Dominates all nodes except 1, 2 and 3

Node 5 and 6 - Dominates themselves.

Node 7 - Dominates 7, 8, 9, 10

Node 8 - Dominates 8, 9, 10

Node 9 and 10 - Dominates themselves.

Natural Loops-

Natural loops can be defined by 'Back Edge' i.e $n \to d$ means d dominates n.

- In the above example, natural loops are,

Backedge $n \to d$ ie d dominates n.

i) $8 \to 3$ i.e 3 dominates 8 (3, 4, 5, 6, 7, 8)

Here, 3 is header

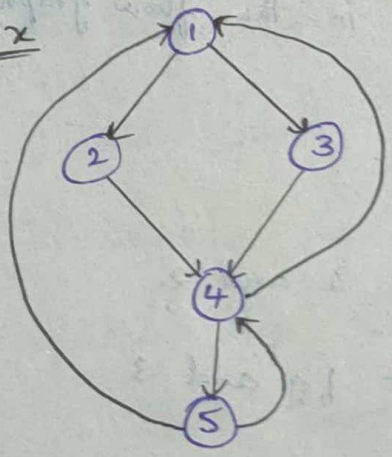
ii) $9 \to 4$ i.e 4 dominates 9 (4, 5, 6, 7, 8, 9)

Here, 4 is header

iii) $10 \to 7$ i.e 7 dominates 10

Here, header is 7

Ex



Natural loops are,

Backedge :-

i) $4 \to 1$

1 dominates 4 (1, 2, 3, 4)

ii) $5 \to 4$

4 dominates 5 (4, 5)

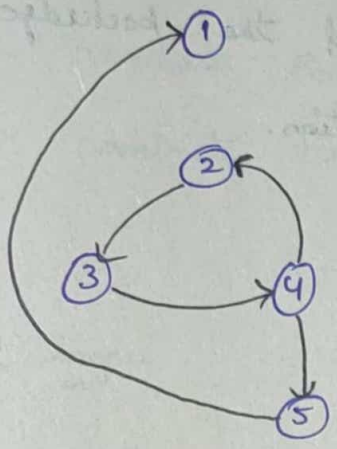
iii) $5 \to 1$

1 dominates 5 (5 → 1, 2, 3, 4, 5)

3. Inner loop -

Inner loop is a loop that contains no other loop.

Ex:-



Here the inner loop is,

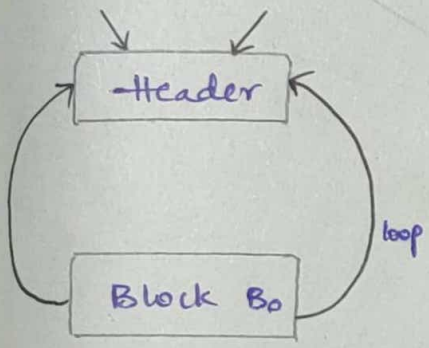
$4 \to 2$

i.e 2 dominates 4

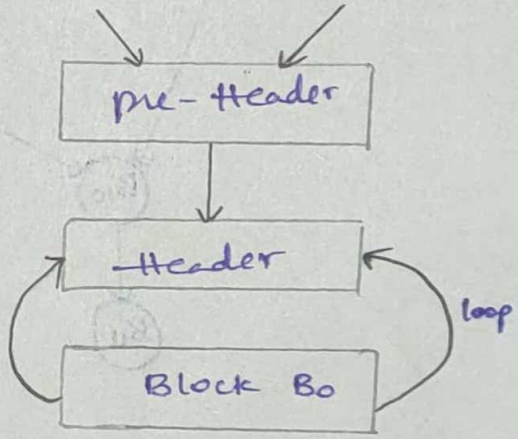
$(4 \to 2, 3, 4)$

4. Pre-Header - means preceeding the Actual Header.

- It is used to facilitate the loop transformation computation.



Before pre-Header

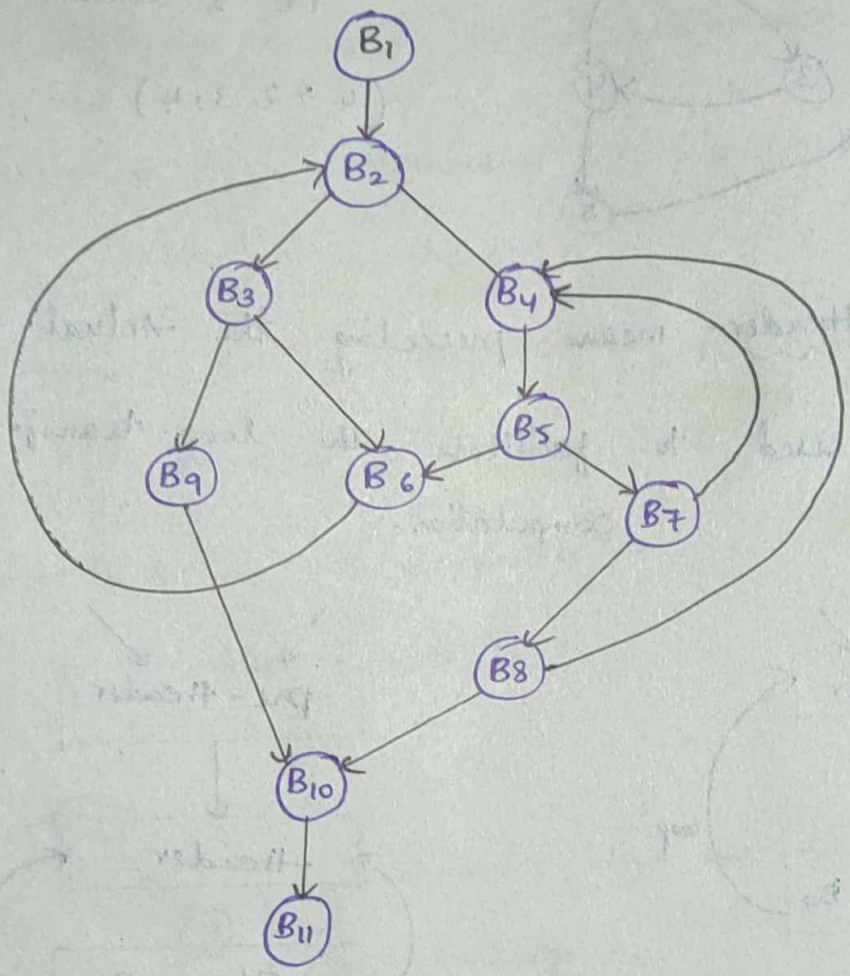


After pre-Header

Example - Consider the following flow graph and find a) Dominators for each basic block

b) Detect all the loops in the flow graph

c) For each loop, find the backedge and header-block information.



NOTE- Every node dominates itself

a) Dominators -

Node B_1 - Dominates all the nodes

Node B_2 - Dominates all nodes except B_1.

Node B_3 - Dominates B_9.

Node B_4 - Dominates B_5, B_7, B_8

Node B_5 - Dominates B_7, B_8

Node B_6 - Dominates itself.

Node B_7 - Dominates B_8

Node B_8 - Dominates itself

Node B_9 - Dominates itself.

Node B_{10} - Dominates B_{11}.

Node B_{11} - Dominates itself

b) - c)

Natural loops -

i) $B_6 \rightarrow B_2$ $\therefore B_2$ is header

ii) $B_7 \rightarrow B_4$ $\therefore B_4$ is header

iii) $B_8 \rightarrow B_4$ $\therefore B_4$ is header