Data Structures

MODULE : 1

- Ordered list (Linear relationship)
- Array : collection of elements of same type stored in consecutive memory locations (static memory allocation)

Stack & Queue

- Stack : last in first out eg: nested fernc calls
- Queue : first in first out

Stack using Array

- A stack is an ordered list where insertions and deletions are done at one end called top of the stack
  - 1. Push ≡ Insert
    2. Pop ≡ Delete
    3. Display
  - Two conditions: (beco3 array has static memory)

1. Overflow : the stack will be full
2. Underflow : no elements in the stack

eg: int top;
int stack[5];

- top = n-1 (here 4) : full
- top = -1 : no elements
- top = 0 : one element

top : index of topmost element.

- During each push , top increment by 1
- During each pop , top decrement by 1

Algorithm for Push(item)

1. Start
2. Check for overflow
   if (top = n-1)
   printf("Stack full");
   exit;
3. Increment top by 1
   top += 1
4. Place the item at position given by top
   Stack[top] = item
5. Stop

Algorithm for pop(~~item~~)

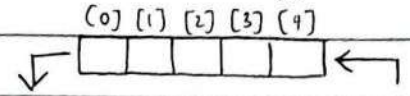
1. Start
2. Check for underflow
   if (top = -1)
   print ("Stack empty")
   exit;
3. item = stack[top]
   printf("Deleted item = " item)
4. top = top-1
5. Stop

- Cannot delete an element from an array. It can only be over written.

Algorithm for display()

1. Start
2. Check for underflow
3. for (i = top, i ≥ 0; i--)
   print (stack[i])
4. Stop

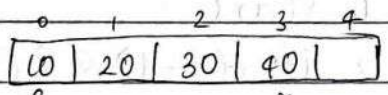
Queue using arrays



front end (f) rear end (r)

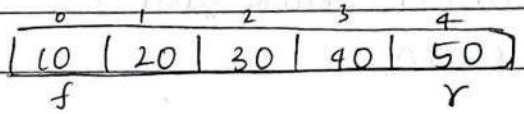
- Operations:
1. Enqueue : Enter 3. Display
2. Dequeue : Delete

- full : rear = n-1
  empty : front = rear = -1
- Enqueue : rear = rear+1



f = 0
r = 3

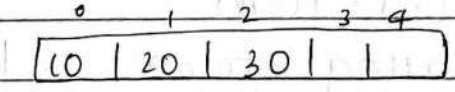
enqueue(50) :



f = 0
r = 4

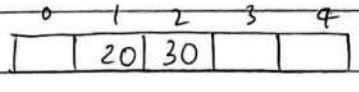
enqueue(60) : no change

- Dequeue :



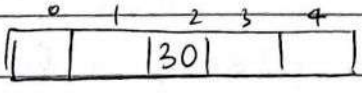
f = 0
r = 2

dequeue() :



f = 1
r = 2

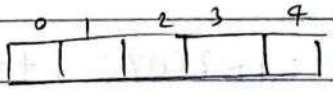
dequeue() :



f = 2
r = 2

Here f = r = 2 ∴ one element

dequeue()



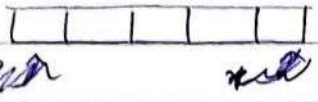
f = -1
r = -1

Here f = r = -1 ie, initialised to -1, no elem.

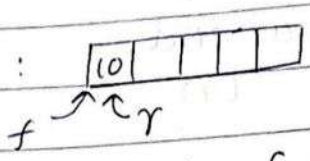
$r = f-1 \% n$

$f = (r+1) \% n$

⇒ Enqueue when $f = r = -1$



after enqueue(10) :



- during first 'enqueue' $f += 1$
- every enqueue step : $r += 1$

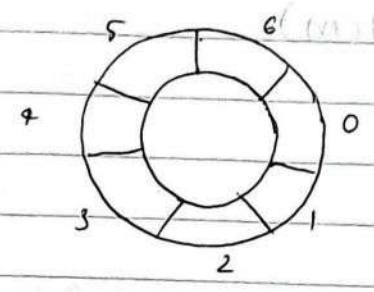
Linear Queue

1. Enqueue (item)
   1. Start
   2. if ($r = n-1$)
      printf("~~Stack~~ Queue full")
      return;
   3. else
      $r = r+1$
   4. queue[r] = item
   5. When inserting into an empty queue,
      if ($f = -1$)
      $f = f+1$
   6. Stop

2. Dequeue()
   1. Start
   2. if ($f = -1$) && ($r = -1$)
      print("Queue empty")
      exit
   3. item = queue[f]
   4. print("deleted item" = item)
   5. If ($f == r$)
      $f = r = -1$
      else
      $f = f+1$
   6. Stop

3. Display
   1. Start
   2. for (i = front; i ≤ rear; i++)
      printf(Queue[i])
   3. Stop

Circular Queue (CQ)



Queue [7]

(n = 1)

r	(r+1) % n
0	1
1	2
2	3
3	4
4	5
5	6
6	0

1. Enqueue_CQ(item)
   1. Start
   2. if (front = (rear+1)%n)
      print("Queue full")
      exit
   3. rear=(rear+1)%n
   4. queue [rear] = item
   5. //insert to ~~single element~~ empty queue
      if (front = -1)
      front += 1
   6. Stop

2. Dequeue_CQ()
   1. Start
   2. if (rear = (front-1)%n)
      printf("Queue empty")
      exit
   3. item = queue [front]
   4.

2. Dequeue_CQ()
   1. Start
   2. if (front = -1) && (rear = -1)
      print ("Empty queue")
      exit

3. item = queue[front]
4. printf("Deleted item=" item)
4. if (front = rear)
front = rear = -1
else
front = (front+1) % n
5 Stop

(126)

f=2
f=5

3. Display
1. start
2. for (i = front; i <= rear; i++)
printf (Queue[i])
3. Stop

(OR)

i = front
while (i != r)
printf (queue[i])
i = (i+1) % n
print (queue[r])

(OR)

for (i=f; i==r+1; c = (i+1) % n)
print (queue[i])

Disadvantage of linear queue
- As we delete elements in a queue, the front gets incremented

To overcome:
- Shifting the elements to the front after each deletion. But this involves significant overhead becoz especially when the size of

the queue is large

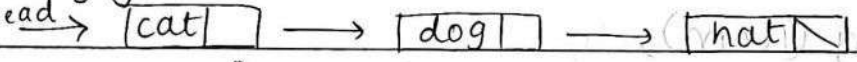
- Treating the array as a circular one, which is known as circular queue.

Linked List

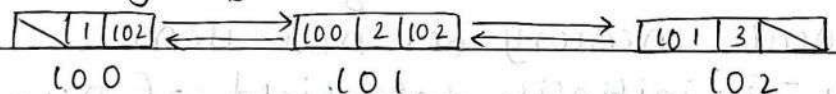
- A linked list is an ~~homogenous~~ ordered collection of homogenous elements called nodes where the linear order is maintained by means of links (also called as pointers).
- Sequential access

8/7/25 Types of linked list

1. Singly linked list



2. Doubly linked list



Operations of linked list

1. Insert : 3 cases → @ front ⓑ between ⓒ end
2. Delete : 3 cases
3. Create
4. Display

⇒

```
struct node {
    char data[3]
    struct node *link;
};
now = (struct node*)
```

malloc (size of (struct node))

```
struct node {
    char data[3]
    struct node *link;
};
struct node *head *new;
typedef struct node Node;
new = (struct node*) malloc (sizeof (struct node))
```

* Pointer : Dynamic memory allocation

```
int *p
p = (int *) malloc (size of (int))
```

Insert @ front

```
Insert bag (item);
new = (struct node*) malloc (size of (struct node))
        ↳ allocate memory for new item
new → data = 5; initialise data field of new node
new → link = head; attach the link field of new
                node to 1st of already existing s
head = new; change the head.
```

11/7/25

Insert @ Back end

```
Insert_end (item)
1. new = (struct node*) malloc (size of (struct node))
2. new → data = item
3. new → link = NULL
4. ptr = head
5 while (ptr → link != NULL)
```

ptr = ptr → link // go to next node

6 ptr → link = new

Q.1 Count the number of nodes in a linked list.

→ 1. new = (struct node *) malloc (size of (struct node))

2. int count = 1

3. ptr → head

4. while (ptr → link != null) {

ptr = ptr → link

count ++ }

5. print count

Print

Q.2. Print the elements in a SLL

→ 1. ptr = head

2. while (ptr != null)

printf(" %d", ptr → data)

ptr = ptr → link

Insert in between

Insert_in b/w (item)

1. new =

2. new → data = item

3. ptr = head // ptr is a temp. pointer

4 while (ptr → data != key) && (ptr → link != null)

ptr = ptr → link

5. /outside loop

If (ptr → link != null)

new → link = ptr → link

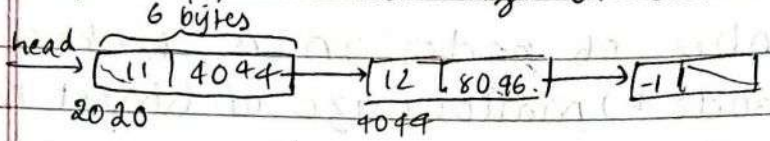
ptr → link = new

& else print ("key not found")

14/7

- Scale factor: expand according to the size of the element

ptr+1 ← adds size of element



ptr = 2020 so, ptr = ptr → link

ptr+1 : 2026 ∴ ptr = 4044

Delete @ beginning()

1. Start
2. if (head == NULL)
   print ("Link is empty")
   exit
3. ptr = head
4. print ("Deleted %d", ptr → data)
5. head = head → link
6. free (ptr)
7. Stop

Delete_@_end()

1. Start
2. Check for empty
3. prev = NULL, ptr = head
4. while (ptr → link != NULL)
   prev = ptr
   ptr = ptr → link
5. prev → link = NULL
6. free (ptr)
7. Stop

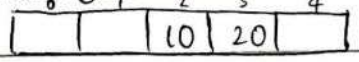
Delete a node

1. Start
2. Check for empty
3. ptr = head, prev = NULL, next = NULL
4. Read key
5. while (ptr → data != NULL) && (ptr != NULL)
   prev = ptr
   ptr = ptr → link
6. if (ptr == NULL)
   print "key not found"
   else
   del = ptr
   prev → link = ptr → link
7. print deleted element
8. free (del)
9. stop

Double ended queue

Insertions and deletions can happen at both the ends but not in between.

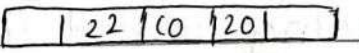
deque [5]



f = 2
r = 3

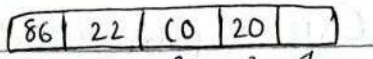
• insert through front : f ↓ by 1

insert @ front end (22)



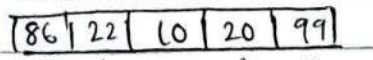
f = 1
r = 3

insert @ front end (86)



f = 0
r = 3

insert @ front end (99)



f = 4
r = 3

r==0 3 4%5 = 4

delete_rear

1. Start
2. Check (f==-1 && r==-1)
   print (Queue empty)
   exit
3. item = Queue[r]
   printf("Deleted item")
4. if (f==r) { f=r=-1 } if (r==0) { r=n-1 }
   else
   r--
5. Stop

insert_front(item)

1. Start
2. if (f=(r+1)%n)
   print (Queue full)
3. if (f=0)
   f=n-1
   else
   f=f-1
4. queue[f] = item
5. Stop

Q. Write algorithm to perform push and ~~push~~ pop on a stack using linked list.

→ push_SLL(item)

1. Start
2. ~~check~~ new=(struct node*) malloc(sizeof(struct node))
3. new→data=item
4. if (new==NULL)
   print(no enough space)
   exit(0)

Queue :

4 new → link = top
5 top = new

pop - SLL
1. Start
2 If (top = NULL)
print stack is empty
exit
3. else
ptr = top
top = top → link
free(ptr)
4. Stop

display_SLL
1. Start
2. if (top == NULL)
print no elements
3. ptr = top
4. ~~for~~ while (ptr != NULL)
print ("%d", ptr → data)
ptr = ptr → link
5 Stop

Queue using LL

Enqueue_SLL(item)
1. Start
2. new = (struct node*) malloc (size of (struct node))
3. new → data = item
4. if (front = NULL)
front = rear = new

Print front → rear

Date
Page

else

rear → link = new

5. new → link = NULL
6. rear = new
7. stop.

Dequeue - ALL

1. Start
2. if (front = rear = NULL)
   print ("Queue empty")
   exit
3. ptr = front
4. front = front → link
5. free (ptr)
6. stop

4. if (front = rear)
   front = rear = NULL
   else

Display()

1. Start
2. if (front = rear = NULL)
   print Queue empty
   exit
3. ptr = front
4. while (ptr → link != NULL)
   print (ptr → data)
   ptr = ptr → link

Representing Single Variable Polynomials

I Array

1. Representation I

$A(x) = 5x^5 + 3x^3 + 2x^2 + 10x + 5$ (size of array : 6)

	0	1	2	3	4	5
coeb	5	10	2	3	0	5

- Array of size "n+1", n: implicit exponent
- Sparse polynomial (very few) : $10x^{1000} + 64x^{88} - 16x^{42} - 6x^3$

0	...	16	...	16	...	64	...	10
0	...	3	...	42	...	88	...	1000

(size = 1001)

- it has very very few information in 1000, ie only 4 elements and the rest is zero.
- wastage of memory
- Advantage : easy access
  simpler code for operations
  $C(x) = A(x) + B(x)$

17/7

Abstract data types

```
stack
 {
  push (item)
  pop()
  top()
  Isempty()
}
```

2. Representation II

Array of terms - 2 fields → coefficient field
↘ exponent field

A(x) =

	0	1	2	3	4
coeff	5	3	2	10	5
exp	5	3	2	1	0

B(x) =

	0	1	2	3
coeff	6	4	2	10
exp	6	4	3	0

- Struct term {
  float coeff;
  int exp;
  }

  A[0].exp = 5

  struct term A[5] B[5]
- Advantage: sparse polynomial can be stored without wasting memory
- Disadvantage: code is complex for operations.

18/7

Algorithm

~~Algo~~ to add two single variable polynomials using array representation II

1. Start
2. i=0, j=0, k=0

Step 3. while (i<np) & (j<nq)

case 1: if p[i].exp < q[j].exp
r[k].coeff = q[j].coeff
r[k].exp = q[j].exp
k++, j++;

case 2: if p[i].exp > q[j].exp
r[k].coeff = p[i].coef
r[k].exp = p[i].exp
k++, i++;

case 3: if p[i].exp = q[j].exp
r[k].coeff = p[i].coef + q[j].exp
r[k].exp = p[i].exp
k++, i++, j++

// end while.

eg: $P(x) = 10x^7 + 5x^5 + 3x^3 + 2x$

$q(x) = 8x^6 + 10x^5 + 4x^4 + 2x^2 + 10x + 15$

p:

i →	0	1	2	3
	10	5	3	2
	7	5	3	1

q:

j →	0	1	2	3	4	5
	8	10	4	2	10	15
	6	5	4	2	1	0

∴ r:

k →	0	1	2	3	4	5	6	7
	10	8	15	4	3	2	12	15
	7	6	5	4	3	2	1	0

Step 4. // copy the remaining terms from poly 'p'

```
while (i<np){
    r[k].coef = p[i].coeb
    r[k].exp = p[i].exp
    k++, i++;
}
// copy the remaining terms from poly 'q'
while (j<nq){
    r[k].coeb= q[j].coeb
    r[k].exp = q[j].exp
    k++; j++;
}
```

5. Stop

Assignment : write the ~~polynomial~~ algorithm to multiply two ~~algorithms~~ polynomials.

→ 1. Start

2. Initialise i = 0, j = 0, k = 0

3. For i = 0 to n_1-1 // n_1 : no. of terms in P_1

   For j = 0 to n_2-1 // n_2 : " " " P_2

   r[k].coeff = p[i]coef * q[j].coeb

   r[k].exp = p[i].exp + q[j].exp

```
        k = k+1
4. for  i=0  to  k-1
       for  j = i+1  to  k-1
            if  r[i].exp == r[j].exp
                r[i].coeff = r[i].coeff + r[j].coeff
                r[j].coeff = 0
                r[j].exp = -1
5. //Remove zero coefficient terms
   Create result array r2[]
   set  m = 0
   for  i = 0  to  k-1
      if  r[i].coeff ≠ 0
          r2[m].coeff = r[i].coeff
          r2[m].exp = r[i].exp
          m = m+1
6. Stop
```

23/7

- Self referential structures : Structures that points to other structures of the same type.

Polynomial addition using SLL

```
1. Start
2. i=p, j=q, k=r
3. while (i!=NULL) && (j!=NULL)
   case 1 : if i→exp > j→exp
                new = getnode()
                new→coeff = i→coeff
                new→exp = i→exp
                new→link = NULL
                i = i→link
   case 2 : if i→exp < j→exp
                new = getnode()
```

```
        new→coeff = j→coeff
        new→exp = j→exp
        new→link = NULL
        j = j→link
case 3 : if  i→exp = j→exp
        new = getnode()
        new→coeff = i→coeff + j→coeff
        new→exp = i→exp
        new→link = NULL
        i = i→link , j = j→link

if (r = NULL)
        r = k = new
else
        k→link = new , k = new
//end of while..
```

4.

//code to add the remaining terms

Sparse matrix Representation / Tuple/ Triple Representation

$$M = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 2 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 11 \\ 0 & 0 & 5 & 6 & 0 \end{bmatrix} \end{array}_{3\times5}$$

tuple/triple : <row index, col index, value>

⇒

0	3	5	6
1	0	1	2
2	0	3	1
3	1	0	6
4	1	4	11
5	2	2	5
6	2	3	6

} book keeping row

(n: no. of non zero elements)

$(n+1) \times 3$

⇒
```
struct tuple {
    int row, col;
    float value;
}
struct tuple a[n+1];
```

Q. Matrix 10x10 have 5 non zero values. How much space can be saved?

→ $100 \times 2 = 200$

here, $((5+1)\times 3)\times 2 = 36$

∴ it saves $200 - 36 = 164$ bytes //

Algorithm to find transpose of a matrix.

1. Start
2. // Initialise 0th row of the transpose matrix b
   b[0].row = a[0].col
   b[0].col = a[0].row
   b[0].value = a[0].value.
3. // initialise the position in 'b'
   current b = 1
4. for i=0 to a[0].col do
   for j=1 to a[0].value do
   if a[j].col=i then
   b[current b].row = a[j].col
   b[current b].col = a[j].row
   b[current b].value = a[j].value
   current b++
5. Stop

Applications of Stack

1. Infix : operators comes in b/w the operands
   eg: 5 + 6/2 - 4 * 3
2. Postfix : operators comes after corresponding operand
   eg: 5 6 2 / + 4 3 * -
3. Prefix : operators comes before corresponding operand
   eg: - + 5 / 6 2 * 4 3

Infix ⟶ Postfix
Step 1: Parenthesis according to precedence
Step 2: move operator to right of corresponding ')' paranthesis.

Eg: ((A + (B / C)) - (D * E)) ⟶ A B C / + D E * -

Algorithm for multiplication of Polynomials : SLL

1. Start
2. Define node structure c, p, l
3. Initialise head pointer as NULL
4. Input no. of terms
5. Repeat n terms
   a) Input coeff and power
   b) Create new nodes with this value
   c) Insert node at end of list
6. Initialise an empty linked list result
7. For each term t_1 in P_1
   For each term t_2 in P_2
   a) Multiply coeff : $c = t_1.c * t_2.c$
   b) Add power : $p = t_1.p + t_2.p$

c) Insert or add this term to result list
d) If a term with same power already exists add its coeff
e) otherwise insert new node

8. Traverse result linked list
9. Print each term in the form coeff x^ power
10. stop.

30/7/25 Sparse Matrix Addition

a:

i			
	10	10	8
↓	1	2	2
	2	1	3
	2	3	1
	3	5	6
	4	5	7
	5	0	2
	7	7	9
	9	8	9

b:

j			
	10	10	6
↓	1	2	3
	2	2	4
	2	4	3
	2	5	-1
	3	5	16
	9	8	7

result:

k			=k
↓	10	10	(?)
	1	2	5
	2	1	3

1. Start
2. if a[0].row != b[0].row || a[0].col != b[0].col
   printf("Addition not possible")
3. c[0].row = a[0].row
   c[0].col = a[0].col
4. i = 1, j = 1, k = 1
5. while (i <= t1) && (j <= t2)
   case 1 : if a[i].row = b[j].row && a[i].col = b[j].col
   c[k].row = a[i].row
   c[k].col = a[i].col

c[k].val = a[i].value + b[j].value
i++, j++, k++

case 2: if a[i].row = b[j].row && a[i].col < b[j].col
c[k].row = a[i].row
c[k].col = a[i].col
c[k].val = a[i].val
i++, k++

case 3: if a[i].row = b[j].row && a[i].col > b[j].col
c[k].row = b[j].row
c[k].col = b[j].col
c[k].val = b[j].val
j++, k++

case 4: if a[i].col = b[j].col && a[i].row < b[j].row
c[k].row = a[i].row
c[k].col = a[i].col
c[k].val = a[i].val
i++, k++

case 5: if a[i].col = b[j].col && a[i].row > b[j].row
c[k].row = b[j].row
c[k].col = b[j].col
c[k].val = b[j].val
j++, k++

//end while

6. while (i <= t1)
c[k].row = a[i].row
c[k].col = a[i].row
c[k].val = a[i].val
i++, k++

7 while (j <= t2)
c[k].row = b[j].row
c[k].col = b[j].col
c[k].val = b[j].val

j++, k++

8. c[0].value = k
9. Stop

Postfix Advantage over Infix
→ easy evaluation
→ parcing easy for compiler.

31/7 Convert the following to prefix and postfix

1. (a+b)/c*d
((a+b)/c)*d)
Postfix : ab+c/d*
Prefix : */+abcd

2. a+b*c/d+e^f/g ⇒ ((a+((b*c)/d))+((e^f)/g))
Postfix : abc*d/+ef^g/+
Prefix : ++a/*bcd/^efg

3. A+B-C ⇒ ((A+B)-C)
Postfix : AB+C-
Prefix : -+ABC

4. a+b/(c*d^e) ⇒ (a+(b/(c*(d^e))))
Postfix : abcde^*/+
Prefix : +a/b*c^de

5. A+(B*C-D/E)-F = ((A+((CB*C)-(D/E)))-F)
Postfix : ABC*DE/-+F-
Prefix : -+A-*BC/DEF

6. $1 + (2 * 5 - (6 / 2 \wedge 3) * 8) * 4$

$\Rightarrow (1 + (((2 * 5) - (((6 / (2 \wedge 3)) * 8)) * 4))$

Post : 1 2 5 * 6 2 3 ^ / 8 * - 4 * +

Pre : + 1 * - * 2 5 * / 6 ^ 2 3 8 4

7. $((2+3) * 2 - (4-3)) \wedge (1+2)$

Post : 2 3 + 2 * 4 3 - - 1 2 + ^

Pre : ^ - * + 2 3 2 - 4 3 + 1 2

1/8/25 Algorithm to convert Infix to Postfix using Stack

1. Start
2. Push '(' onto stack and add ')' at end of Infix expression Q
3. Read Q from left to right and repeat step 4 to 6 for each symbol of Q ~~using~~ until end of expression.
4. If an operand ⊗ occurs, add it to P.
5. If an operator ⊗ occurs then,
   A. If repeatedly pop from stack and to P each operator which has same or higher precedence than ⊗.
   B. Push ⊗
6. If a ')' occurs then
   A. Repeatedly pop from the stack and add to P each operator until '('
   B. Pop '('
7. Stop.

4) A. If '(' occurs Push ('(')

Eg:

	Infix 'Q'	stack	Postfix 'P'	Action
①	a + b / c * d)	(		Initial
	+ b / c * d)	(	a	Add a to P
	b / c * d)	(+	a	Push (+)
	/ c * d)	(+	a b	Add b to P
	c * d)	(+ /	a b	Push (/)
	* d)	(+ /	a b c	Add c to P
		(+	a b c /	~~Push~~ Pop ()
	d)	(+ *	a b c /	Push (*)
	)	(+ *	a b c / d	Add d to P
		(+ *)	a b c / d	end of exp
				pop (), pop ()
			a b c / d * +	pop ()

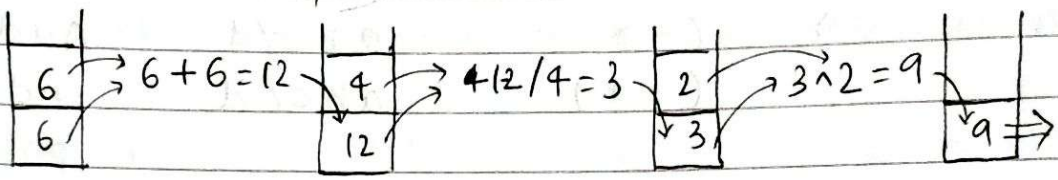
② a + b * (c / d) + e ^ (f / g)

Infix	Stack	P.E	Action
a+b*(c/d)+e^(f/g))	(		Initial
+b*(c/d)+e^(f/g))	(	a	Add a to P
b*(c/d)+e^(f/g))	(+	a	Push (+)
*(c/d)+e^(f/g))	(+	a b	Add b to P
(c/d)+e^(f/g))	(+ *	a b	Push (*)
c/d)+e^(f/g))	(+ * (	a b	Push (()
/d)+e^(f/g))	(+ * (	a b c	Add c to P
d)+e^(f/g))	(+ * (/	a b c	Push (/)
)+e^(f/g))	(+ * (/	a b c d	Add d to P
+e^(f/g))	(+ *	a b c d /	Pop (), Pop ()
+e^(f/g))	(+	a b c d / *	Pop (), ~~Pop ()~~
+e^(f/g))	(	a b c d / * +	Pop ()
e^(f/g))	(+	a b c d / * +	Push (+)
^(f/g))	(+	a b c d / * + e	Add e to P
(f/g))	(+ ^	a b c d / * + e	Push (^)
f/g))	(+ ^ (	a b c d / * + e	Push (()

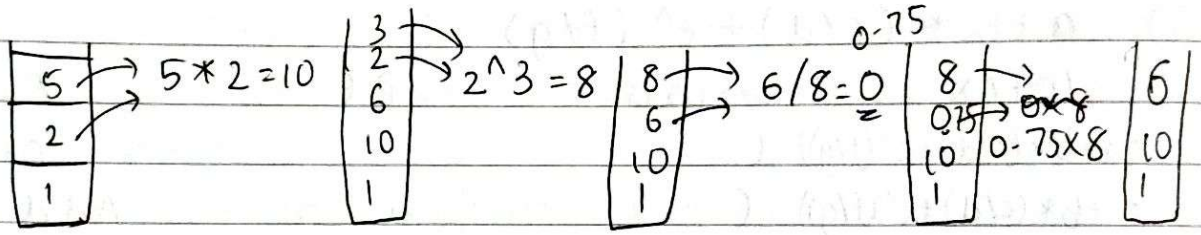
/g))	(+^(	abcd/*+ef	Add f to P
g))	(+^(/	abcd/*+ef	Push (/)
))	(+^(/	abcd/*+efg	Add g to P
)	(+^	abcd/*+efg/	Pop(), Pop()
	(+	abcd/*+efg/^	Pop()
	empty	abcd/*+efg/^+	Pop()

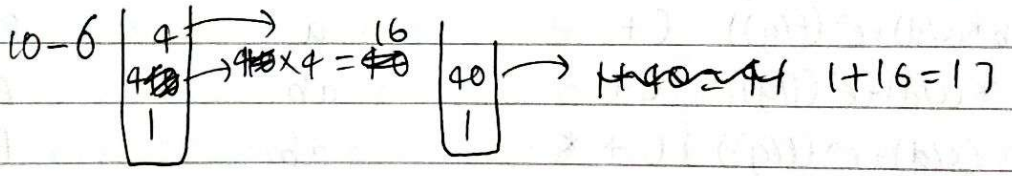
Evaluate P.E using stack

Q.1 6 6 + 4 / 2 ^ $



2. 1 2 5 * 6 2 3 ^ / 8 * - 4 * + $





6/8/25

Algorithm to evaluate a Postfix expression

1. Start
2. Scan the postfix expression from L → R and repeat steps 3 → 5
3. If an operand occurs, push it to stack
4. If an operator ⊗ occurs then
   A. Pop top two element off stack where B is the top element and A is the next top element
   B. Evaluate A ⊗ B

c. Push the result of 4B into the stack
5. Set the value of P as top of stack.

Eg:

Input	Stack	Action
125*623^/8*-4*+	~~:~~ —	Initial
25*623^/8*-4*+	1	Push (1)
5*623^/8*-4*+	1 2	Push (2)
623^/8-4*+	1 2 5	Push (5)
623^/8*-4*+	1 10	Pop 2 elem. ~~Evalu~~ Push 2*5
23^/8*-4*+	1 10 6	Push (6)
3^/8*-4*+	1 10 6 2	Push (2)
^/8*-4*+	1 10 6 2 3	Push (3)
/8*-4*+	1 10 6 8	Pop 2 elem Push (2^3)
8*-4*+	1 10 ~~0.75~~ 0	Pop 2 elem Push (6/8)
-4+	1 10 0~~75~~ 8	Push (8)
-4*+	1 10 ~~6~~ 0	Pop 2 elem Push (0 ~~75~~ x8)
4*+	1 10	Pop 2 elem Push (10-0)
*+	1 10 4	Push (4)
+	1 40	Pop 2 elem Push (10x4)
—	41	Pop 2 elem Push (1+40)

Hence answer 41

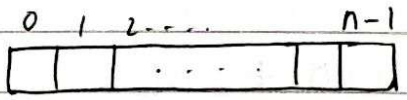
2)

Input	Stack	Action
23+2*43--12+^	—	Initial
3+2*43--12+^	2	Push(2)
+2*43--12+^	23	Push(3)
2*43--12+^	5	Pop 2 elem, Push(2+3)
*43--12+^	52	Push(2)
43--12+^	10	Pop 2 elem, Push(5*2)
3--12+^	10 4	Push(4)
--12+^	10 4 3	Push(3)
-12+^	10 1	Pop 2 elem, Push(4-3)
12+^	9	Pop 2 elem, Push(10-1)
2+^	9 1	Push(1)
		~~Pop 2 elem, Push(9^2)~~
+^	9 1 2	Push(2)
^	9 3	Pop 2 elem, Push(1^2)
—	~~243~~ 729	" ", Push(9^3)

Hence, 729

Multiple Stack

- two stacks in one array (stacks growing in opposite direction)



Stack A → ← Stack B

Top A ⟶ ← Top B

- Initialise : Top A = -1 and Top B = n
- Condition for full array : Top A = Top B - 1

Algorithm for operations on multiple stack.

1. Start
2. Initialise ~~Top A = -1 and Top B = n~~ stack [n]

3. / Push A

3.1. Check overflow

if ~~TopB~~ (top A = top B - 1)

print (" Array full")

exit

3.2. else

top A = top A + 1

Stack (top A) = item

4. / Push B

if (top A = top B - 1)

print (" Array full")

exit

else

top B--

Stack [top B] = item

5. / Pop A

if (top A = -1)

print (" Stack A empty")

exit

else

top A --

6. / Pop B

if (top B = n)

print (" Stack B empty")

exit

else

top B ++

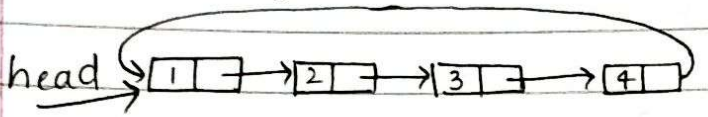
7. Stop

Circular linked list

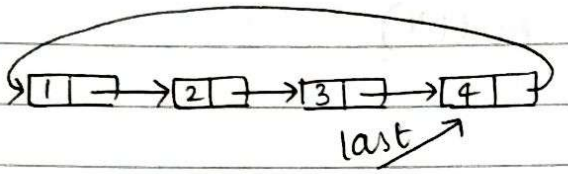
- Advantage: any element is accessible from each element of list

Three ways

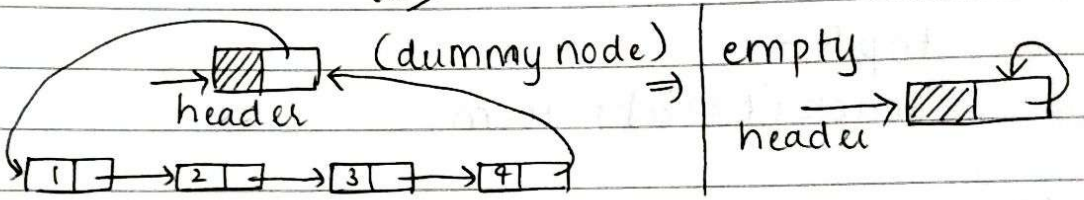
1.



2.



3.



Insert @ beg_CLL (item)

1. Start
2. new = getnode()
3. new → data = item
4. new → link = header → link
5. header ~~new~~ → link = new
6. Stop

Insert @ end_CLL (item)

1. Start
2. new = getnode()
3. new → data = item
4. ~~write~~ ptr = header → link
5. while (ptr → link != header)

ptr = ptr → link

6. new → link = header
7. ptr → link = new
8. Stop

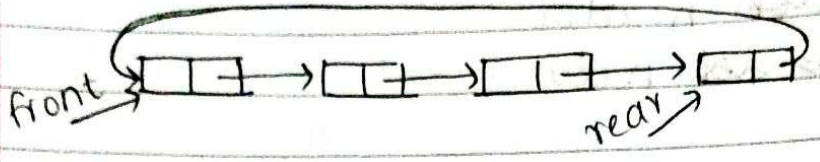
Delete @ front _ CLL

1. Start
2. If header → link == header
   List empty
3. temp = header → link
4. if temp → link == temp:
   header → link = header
   Free temp
5. else
   while (ptr → link != header)
   ptr = ptr → link
   ptr = header → link
   while (ptr → link != header → link)
   ptr = ptr → next
   header → link = temp → link
   ptr → link = header → link
   Free temp.
6. Stop

Delete @ end _ CLL

1. temp = header → link
2. if temp → link == temp
   header → link = header
   Free temp
3. Else
   prev = NULL
   while (temp → link != header → link)
   prev = temp
   temp = temp → link
   prev → link = header → link
   Free temp

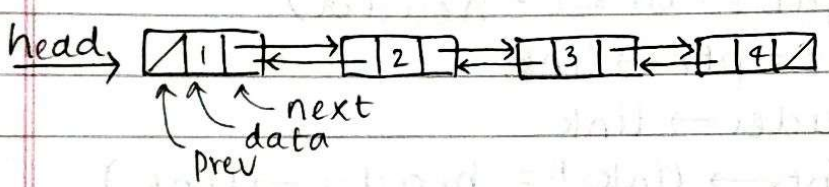
55 Circular queue using CLL (Lab)



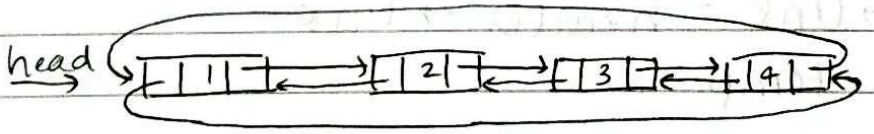
Doubly Linked List

- Each node have a data value and the address of previous and next node.

```
struct node{
    int data;
    struct node *prev, *next;
}
```



Circular doubly linked list



Insert_@ beg (item) [Doubly linked list]

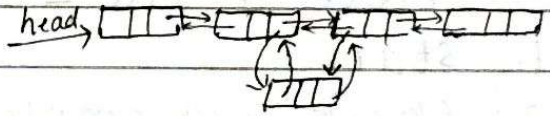
1. Start
2. new = getnode()
3. new → data = item
4. new → next = head
   head → prev = new → prev
   new → prev = NULL

4. new → prev = NULL
5. new → next = head
6. head → prev = new
7. head = new
8. Stop

Insert@end_DLL(item)

1. Start
2. new = getnode()
3. new → data = item
4. new → next = NULL
5. ptr = head
6. while (ptr → next != NULL)
   ptr = ptr → next
7. ptr → next = new
8. new → prev = ptr
9. Stop

Insert in b/w_DLL(item)

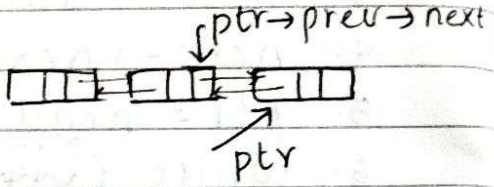


1. Start
2. Read key
3. new = getnode()
4. new → data = item
5. temp = head
6. while (temp → data != key) && (temp → next != NULL)
   temp = temp → next
7. if (temp → next != NULL)
   new → prev = temp
   new → next = temp → next
   temp → next → prev = new
   temp → next = new
   else print ("key not found");
8. Stop

Delete at front - DLL

1. Start
2. ~~ptr = head~~ Check for empty
3. ptr = head
4. head = head → next
5. head → prev = NULL
6. Free ptr
7. Stop.

Delete at end - DLL



1. Start
2. Check for empty
3. ptr = head
4. while (ptr → next != NULL)
   ptr = ptr → next
5. ptr → prev → next = NULL
6. free (ptr)
7. Stop

Delete in blw - DLL

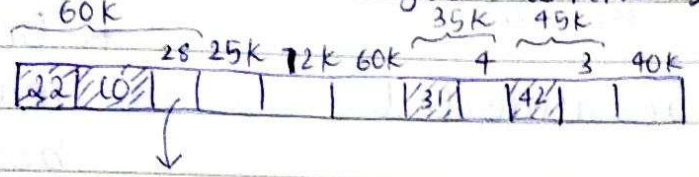
1. Start
2. Check for empty
3. ~~ptr~~ ptr = head
4. while (ptr → data != key) && (ptr → next != NULL)
   ptr = ptr → next
5. If (ptr → next != NULL)
   (ptr → prev) → next = ptr → next
   (ptr → next) → prev = ptr → prev
   free (ptr)
6. else print key not found
7. Stop.

Double Ended Queue using DLL (lab)

8/8/25 Operating System

Memory allocation schemes

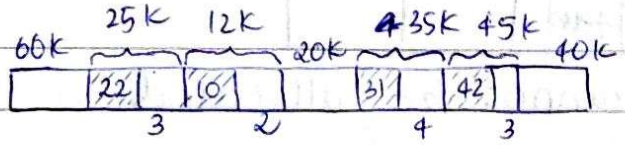
1. First fit
   - largest available free hole which can accomodate the job while scanning from L → R



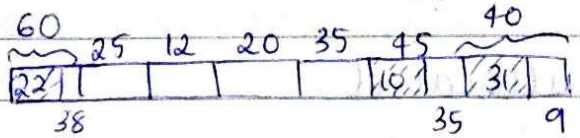
Job Request
J_1 : 22k
J_2 : 10k
J_3 : 43k
J_4 : 31k

   - when lot of requests is there, it become slower ~~scan the entire list~~

2. Best fit
   - finds the best hole.
   - need to scan entire list. • leads to internal fragmentation



3. Worst fit
   - opposite of first fit
   - allocates into the largest hole present in list



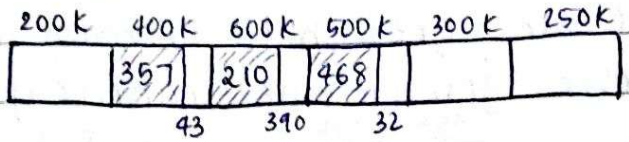
here 43k cannot be allocated.

Q.1 Consider 6 memory partitions of size 200k, 400k, 600k, 500k, 300k, 250k. These partitions need to be allocated to 4 processes of size 357k, 210k, 468k, and 491k. Perform the allocation of

processes using first bit, best bit and worst bit.

2. Memory position bit : 100K, 500K, 200K, 300K, 600K
Job request : 212, 417, 112, 426

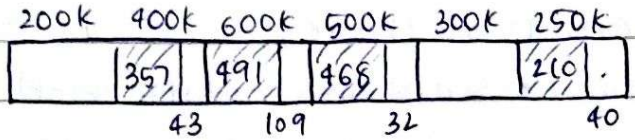
ans: 1. First fit



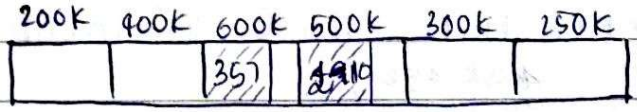
491 cannot be allocated

* Best bit makes the best allocation of memory

Best fit

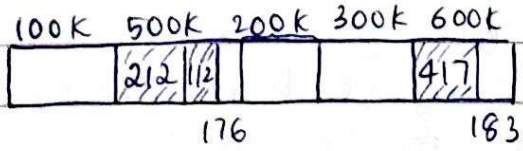


Worst fit



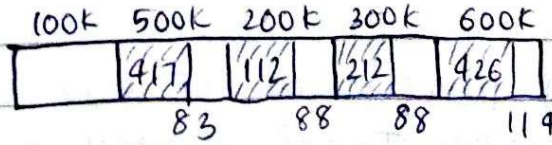
468 and 491 cannot be allocated.

2. First fit

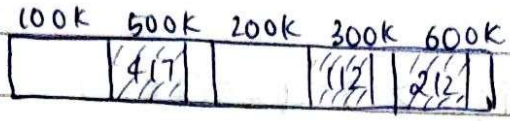


426 cannot be allocated.

Best fit



Worst fit



426 cannot be allocated.

Hence Best fit makes the best allocation of memory.

Algorithm for first fit

1. Start from the head of free list
2. For each block
   a. Check if it's size ≥ requested size
   b. if yes
      → Allocate memory from this block
      → Update the block
      → Return
   c. else
      → traverse to next block
3. If no suitable block is found, request cannot be granted.

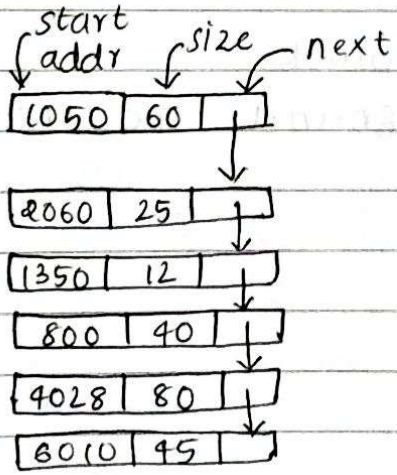
11/8

Algm_FF (req_size)

1. Start
2. temp = head , prev = NULL
3. while (temp != NULL){
4. if (temp→size = req_size)
   // perfect fit, delete block
   if prev = NULL
   head = head→next
   else prev = temp→next

4. free (temp)
8. return()
5. else if temp→size > req-size
// large size, split the block
temp→start_addr += req-size
temp→size -= req_size
return()
6. else // examine the next block
prev = temp
temp = temp→next
// end of while.
7. If (temp = NULL) : printf("memory not availiable")

⇒
```
struct block{
    int start_addr;
    int size;
    int * struct block *next;
}
```



ⓐ req_size = 70
here, temp: 4028 | 80 |
4098 | 10 | (changes)
i.e, 4028 + 70 = 4098
80 - 70 = 10

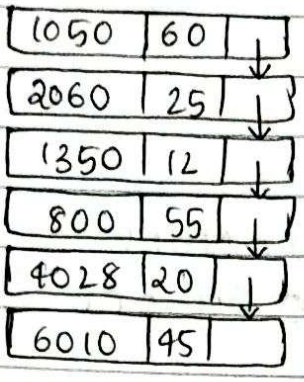
ⓑ req_size = 80
temp: 4028 | 80 |
the block is deleted.

Algorithm for Best Fit

```
1. Start
2. temp = head, prev = NULL
3. best_blk = NULL , best_prev = NULL
4. // examine the free list one-by-one
   while (temp != NULL){
5.       if (temp->size ≥ req_size){

6        if (best_blk = NULL || best_blk->size > temp->size)
7        {     best_prev = prev
8              best_blk = temp
   } // end of outer if
9  prev = temp
10 temp = temp->next
   } end of while.
11. if (best_blk == NULL){
        print ("Req cannot be allocated")
        exit(0)
    }
12. if (best_blk->size = req_size)
        // perfect fit, delete blk
13. if (best_prev = NULL)
14.     head = head->next
15. else  best_prev->next = best_blk->next
16. free (best_blk)
17. return()
18. else if (best_blk->size > req_size){
        // large block , split the block
19. best_blk->start_addr += req_size
20  best_blk->size -= req_size
21. return()
    }
```

req_size = 42



Step 1 : best_blk : 1050 | 60 |
best_pre = NULL

Step 2 : On traversing, ...
~~temp~~ = best_blk 800 | 55 |
best_prev = 1350 | 12 |

Step 3 : best_blk = 6010 | 45 |
best_prev = 4082 | 20 |

- ~~After step 3~~, In each step, best_blk ~~is~~ and best_prev is updated when another best block is found
  ie, 60 > 55 ✓ (55 can occupy 42)
- After step 3, ie end of list, the best_blk and best_prev is found.

Algorithm for worst fit

1. Start
2. temp = head, prev = NULL
3. worst_blk = NULL, worst_prev = NULL
4. while (temp != NULL) {
   if (temp → size >= request_size) {
   if (worst_blk = NULL || temp → size > worst_blk → size) {
   worst_blk = temp;
   worst_prev = prev;
   }}
   prev = temp
   temp = temp → next
   }

```
if (worst_block = NULL){
    print ("No sufficient memory block avail")
    exit;
}
print ("Memory allocated at %d, worst_blk →
                                start_addr);

if(worst_blk→size = req_size){
        // perfect fit
        if (worst_prev == NULL)
                head = worst_blk→next;
        else
                worst_prev→next = worst_blk→next
        free (worst_blk);
}
else{
        // split the block
        worst_blk→start_addr+= req_size
        worst_blk→size-= req_size.
    }
}
```

13/8

Post office Customer Simulation
- Priority Queue

- Browser navigation simulation.

element	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
priority	1	5	2	4	2	1	5	2	1	3

← deleted first (for processing)

5 → highest priority
1 → least priority

1. Simple array

⇒ · Insert @ rear
· delete based on priority
· if same priority elements occur : first come first serve

⇒ can insert based on priority & delete as usual.

P_2	P_3	P_1		
5	2	1		...

↖ P_4 : 4

next_priority = 4

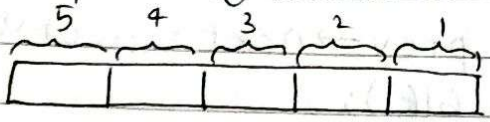
4 > 1 , 4 > 2 , 4 < 5

hence insert b/w P_2 & P_3

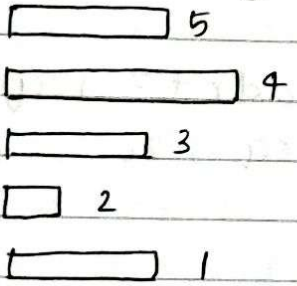
(shifting the rest elements)

2. Multiple Queues

⇒

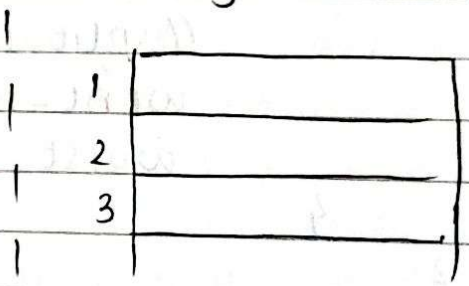


⇒ using many arrays



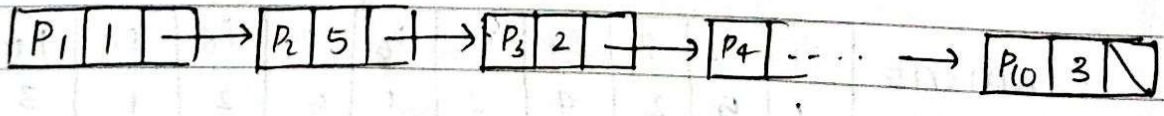
· based on no. of elements in each priority

⇒ using 2D matrix



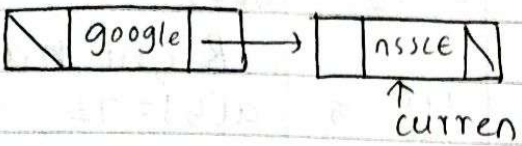
· each row with each priority.

3. Linked List



4. Heap

Browser Navigation simulation (DLL)



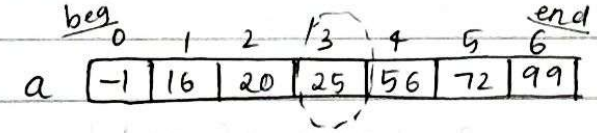
Operations : VisitPage()
GoBack()
GoForward()
DisplayCurrentPage()
Exit

- There is no delete option.

Searching

Linear Search : one by one

Binary Search



1. Find middle element, $mid = \frac{beg+end}{2} = \frac{0+6}{2} = 3$
   hence element of index 3 is middle.
2. Compare key with a[mid]
   a) if key < a[mid] : search left half
      here beg = beg , end = mid − 1
   b) if key > a[mid] : Search right half
      here beg = mid + 1 , end = end
   c) if key = a[mid] : key found.

- The array must be sorted for binary search.

(key=99)

Iteration	beg	end	mid	a[mid]
1	0	6	$\frac{0+6}{2}=3$	a[3]=25, 99>25 Right half
2	4	6	$\frac{4+6}{2}=5$	a[5]=72, 99>72 Right half
3	6	6	$\frac{6+6}{2}=6$	a[6]=99, 99=99 stop

(key=2)

Iteration	beg	end	mid	a[mid]
1	0	6	3	a[3]=25, 2<25 left half
2	0	3-1=2	1	a[1]=16, 2<16 left half
3	0	1-1=0	0	a[0]=-1, 2>-1 right half
4	1	0	// stop element not found	

- To stop search
  1. beg > end 2. key = a[mid]
- No. of iterations for Binary < Lineal search.
- For each iteration, size of array becomes half
  $n \rightarrow n/2 \rightarrow n/4 \rightarrow \cdots$

eg: 2)

0	1	2	3	4	5
-1	16	20	25	56	72

key = -1

Iteration	beg	end	mid	a[mid]
1	0	5	$\frac{0+5}{2}=2$	a[2]=20, -1<20 left half
2	0	1	$\frac{0+1}{2}=0$	a[0]=-1, -1=-1 // stop key found.

Key = 16

Iteration	beg	end	mid	a[mid]
1	0	5	$\frac{0+5}{2} = 2$	a[2] = 20, 16 < 20 left half
2	0	2-1=1	$\frac{0+1}{2} = 0$	a[0] = -1, 16 > -1 right half
3	0+1=1	1	$\frac{1+1}{2} = 1$	a[1] = 16, 16 = 16

// stop key found.

Key = 99

Iteration	beg	end	mid	a[mid
1	0	5	$\frac{0+5}{2} = 2$	a[2] = 20, 99 > 20 Right half
2	2 0+1=3	5	$\frac{3+5}{2} = 4$	a[4] = 56, 99 > 56 Right half
3	5	5	$\frac{5+5}{2} = 5$	a[5] = 72, 99 > 72 Right half
4	6	5	// stop beg > end.	

Performance Analysis

- Analysis is of the algorithm to see if it is efficient
- Based on time and space complexity.
- Theoretical (beco3 not possible to analyse using algorithm).

- Sum of 2 nos

```
int sum(a,b){
    a=2
    b=3
    sum=(a+b)
    print(sum)
}
```

Frequency count = 4

- Sum of 'n' nos

```
1. Start
1  2. Read n
1  3. sum=0
n+1  4. Repeat for i=1 to n {
n        read a[i]
n        sum+=a[i]
*    }
1  5. print(sum)
```

⇒ Freq count = 3n + 4.

- Add 2 matrices

```
1. start
1  2. Read row, col size
1  3. sum=0
n+1  4. for i=1 to row size
n(n+1)      for j=1 to col size
n×n         Read a[i,j], b[i,j]
n×n         c[i,j] = a[i,j] + b[i,j]
    5. Stop.
```

∴ Frear count = $3n^2+2n+3$

Asymptotic Notation

- limiting behaviour

(to make freq count to a common notation)

- The three asymptotic notations are,

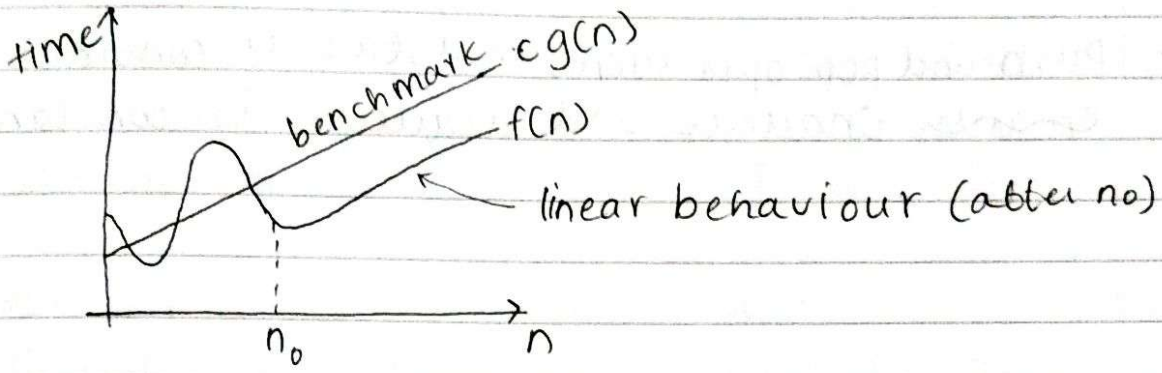
1. Big Oh notation (O)
2. " omega " (Ω)
3. " theta " (θ)

Big Oh Notation

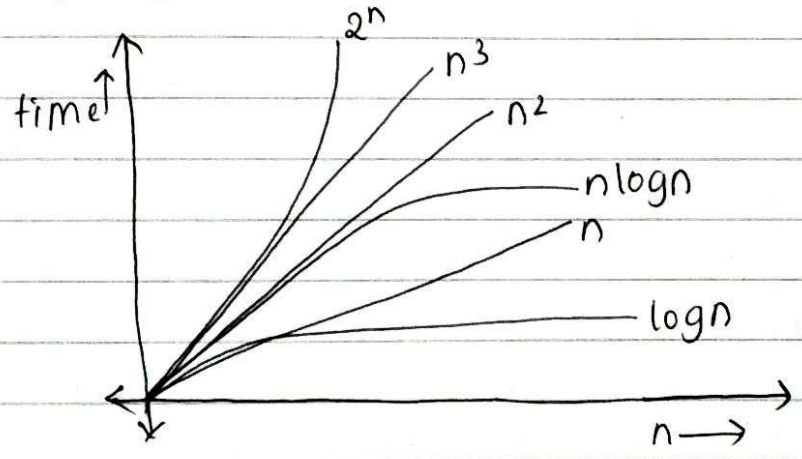
Let f(n) and g(n) be two functions, we say that f(n) = O(g(n)) if and only if there

exists two +ve constants c and n_0 such that $f(n) <= c\,g(n)$ for all $n \geq n_0$.

- ~~Exponentiation : unsolvable~~



Rate of Growth



constant
logarithm
linear
quadratic
polynomial
exponential

- logarithm is best
- exponentiation : unsolvable

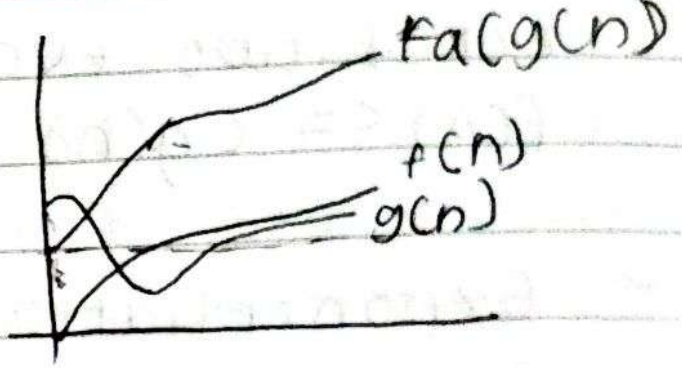
Q. Show that $3n + 4 = O(n)$

→ 1. $3n + 4 \leq 3n + n$
$\leq 4n$
$n_0 = 4$ and $c = 4$

2. $3n + 4 \leq 3n + 4n$
$\leq 7n$
$n_0 = 1$, $c = 7$

n	3n+4	3n+n
0	4	0
1	7	4

2	10	8
3	13	12
4	16	16
5	19	20



- Push and pop operations in a stack is constant.
- ~~Enqueue~~, Enqueue and dequeue is constant.