Application of Stack

Date : / /

① Operator → +, −, *, /, ^

② Operand → A-Z, 1, -9

(+, −) L→R (X, /) L→R (↑) R→L

Computer understands Prefix/Polish

((4*2)+3) → (*42+3) → (+*423)

Infix Prefix

Prefix starts from left side when it founds operator it calculate two Numb

((4*2)+3) → (42*+3) (423*+)

Infinix Postfix

(7+(4×5)) − (2+0) → Prefix conversion

Step 1: Reverse the string →

)0+2(−)1 5×4+ 7(

Step 2:

→ If (0-9) Print the value

→ If ')' Push to stack

→ If '(' POP() from stack until ')' found

→ [+, −] 1, [×, /] 2, [^] 3, [...] −1

① Less presidence হলে বসে যাবে বা ~~same n হলেও বের করে stack থে~~

② higher presidence হলে অপারেটর না same বা less n না পাই ততক্ষণ POP() করতে থাকবে

③ Same presidence প্রবেশ করতে পারবে না।

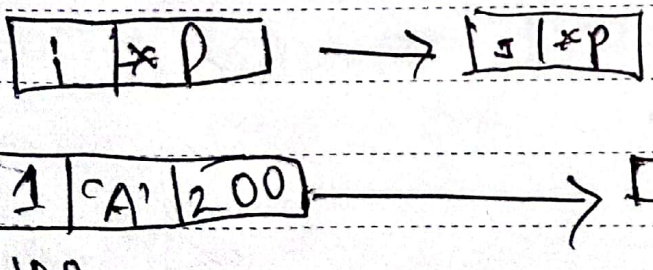
Self Referential Structures

Self Referential structure are those stacture that have ~~more than~~ one or more pointers which point to same type of structure as their Member.

Date : / /

In other words structure pointing to the same type of structure are self-referential in nature.

```
struct node2
    int data1;
    char data2;
    struct node* link;
2
int main()2
    struct node obj;
    return 0;
2
```

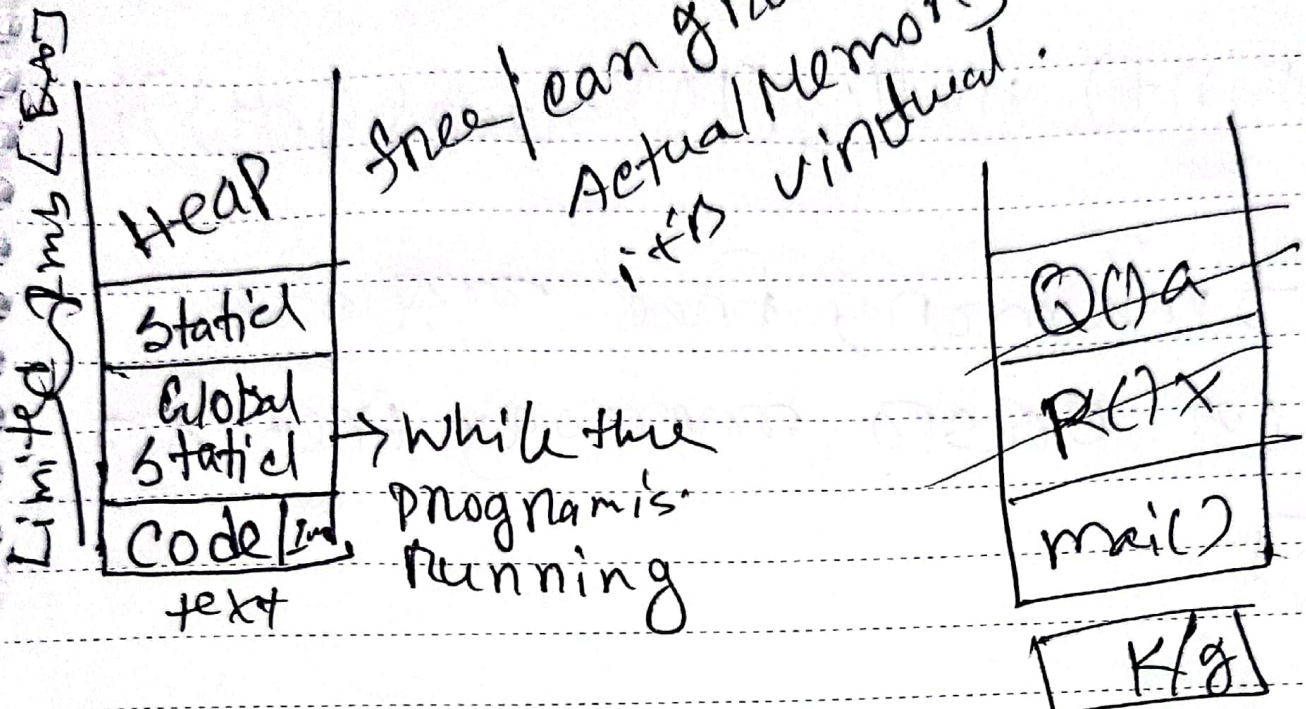


① Pointer and structure variable name same 260 260

Dynamic Memory Allocation

```
#include<stdio.h>
int g; /* global var. */
Q() {
    int a; /* local var */
}
P() {
    int x; /* local var */
    Q();
}
int main() {
    int static k;
    P();
    printf();
}
```

Date : / /



Run time ও Memory দুইভাবে হতে পারে এবং Dynamic memory Allocation প্রক্রিয়ার জন্য Heap এর malloc() calloc() free() realloc()

Malloc() <stdlib.h>

malloc ⇒ Memory allocation

syntax ⇒ void * malloc(sizeof());

size of ⇒ different type of memory size জানার জন্য ব্যবহার, int 2 byte বা 4 byte হতে পারে তার জন্য এর

int *p ⇒ *p = (int) malloc(sizeof(int));

P → [box] -2000 [memory adress

p = (*int) malloc(10* sizeof(int));

free → memory free করে দেয়া হলো যে অংশের memory তৈরি হয়েছে রাখতে।

calloc() → contigious

① calloc need two arguments

void * calloc(size_t n, size_t size);

int *ptr = (int*) calloc(10, sizeof(int));

② memory allocated by calloc is initialized by zero & by malloc with some garbage value.

Date : ../../..

Malloc & calloc returns null when sufficient memory in not available in heap.

realloc()

* Changes the size of the memory block without losing old data.

```
int *Ptr = (int*) malloc (sizeof (int));   // 4 bytes
Ptr = (int*) realloc (Ptr, 2*sizeof(int));  // 8 bytes
```

* This will allocate memory space 2 of size
* Also this function moves the contens of the old block to a new block and the data of the old block is not lost
* We may loose the data when the new size is smaller than the old size.

Free()

* Used to release the dynamically allocated memory.

void free(PTR)

The memory allocated to heap will not be released automatically after using memory. The space remains there and cant be used.

int *P = (int*) malloc(4 * sizeof(int));

Typedef

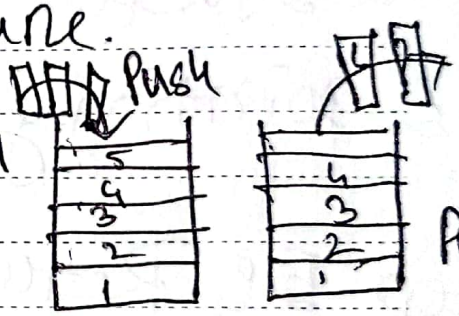
It is used to give a new name to a type.

Date : / /

typedef struct node point;
old new

Stack

① Abstract data type. ② Linear data structure.
works on LIFO structure.
Insertion is called push
Removal is called POP



Peek() - get the top data element of stack
isFull() → check if the stack is full
isEmpty() → checks if stack is empty

Push operation

Step 1: Check if the stack is full.
Step 2: If the stack is full produce error.

Step3: If the stack is not full, increament top to point to next empty space.

Step 4: Adds data element to the stack location, where top is pointing.

Step 5: Return success.

pop operation

In an array the data element is no actually removed instead top is decremented to a lower position in stack in pop() operation. But in linked list

Date : / /

POP() actually removed data element and deallocate memory space.

Step1: Checks if the stack is empty().

Step2: If the stack is empty, produce an error and exit.

Step3: If the stack is not empty access the data element at which top is pointing

Step4: Decrease the value of top by 1

Operators	Presidence
('↑ or ^)	Highest (3)
*, /	Next highest / Mid (2)
to +, -	lowest (1)

~~Prefix to Infix~~

Postfix to Infix

Or

(1) Read the operands push it to stack

(2) If it is operator pop two value from stack Add it between them and push it.

Evaluate ABC*DEF^/G*-H*+

→

Reading of Postfix	Stack to
A	A
B	A | B
C	A | B | C
*	A | B*C
D	A | B*C | D
E	A | B*C | D | E
F	A | B*C | D | E | F

Operator ↑

2nd | 1st top

↑	A | B ↑ C | D | E ↑ F
/	A | B ↑ C | (D / (E ↑ F))
G	A | B ↑ C | (D / (E ↑ F)) | G
*	A | B ↑ C | (D / (E ↑ F)) * G)
-	A | ((B ↑ C) - ((D / (E ↑ F)) * G))
H	A | ((B ↑ C) - ((D / (E ↑ F)) * G)) | H
*	A | (((B ↑ C) - ((D / (E ↑ F)) * G)) * H)
+	(A + (((B ↑ C) - ((D / (E ↑ F)) * G)) * H))

Ans: (A+(((B↑C)-((D/(E↑F))*G))*H))

Infix to Postfix

① Print 1-9 / A-Z

② If '(' Push to stack

③ If ')' → Pop from u until '(' find it.

④ [+ -]₁, [* /]₂, ↑₃, ...₋₁

Given: 2+4/5+(5-3)^5^4

Symbol	Stack	Expression
2		2
+	+	2
4	+	24
/	+/	24
5	+/	245
*	+*	245/
(	+*(	245/
5	+*(	245/5
-	+*(-	245/5
3	+*(-	245/53
)	+*	245/53-
^	+*^	245/53-
5	+*^	245/53-5

Date : ../../..

Symbol

Stack	Stack	Expression
^	+ * ^	245 ^ 53 - 5 ^
4	+ * ^	245 ^ 53 - 5 ^ 4
	+ *	245 ^ 53 - 5 ^ 4 ^
	+	245 ^ 53 - 5 ^ 4 ^ *
		245 ^ 53 - 5 ^ 4 ^ * +

Infix to Prefix

① Highest priority operator will not stay in the stack when lowest operator will be inserted.

② Same level can stay together.

A * B ^ C - D + E / F / (G + H)

(H + G) / F / E + D - C ^ B * A

③ Reverse the equation
Do Operation
Reverse it again

Symbol	Stack	Expression
(	(	
H	(	H
+	(+	H
G	(+	HG
)		HG+
/	/.	HG+
F	/	HG+F
/	//	HG+F
E	//	HG+FE
+	+	HG+FE//
D	+	HG+FE//D
-	+-	HG+FE//D
C	+-	HG+FE//DC
^	+-^	HG+FE//DC
B	+-^	HG+FE//DCB

Date : / /

~~*~~	+ - *	H G + F E // D C B ^
A	+ - *	H G + F E // D C B ^ A
	+ -	H G + F E // D C B ^ A *
		H G + F E // D C B ^ A * - +

(+ - * A ^ B C D // E F + G H)

Prefix to Infix

(1) Scan the prefix expression from High to left / reverse it.

(2) If the scanned Symbol is an operand then push onto the stack

(3) If the scanned symbol is an operator pop two operands from the stack

creat it an string by placing the operators in between two Operands and push it to stack.

(iv) Repeat step (3 & 2)

top value
x = E
y = F
x operator y

⇒ ~~H G F E A D /~~

⇒ + A ↑ − ↑ B ↑ ↑ / D ∧ E F G H

Symbol	Stack
H	H
G	H | G
F	H | G | F
E	H | G | F | E → E∧F
∧	H | G | E∧F
D	H | G | E∧F | D

Date : / /

/ | H | G | (D / (E ^ F))

* | H | (D / (E ^ F) * G)

C | H | (D / (E ^ F) * G) | C

B | H | (D / (E ^ F) * G) | C | B

* | H | (D / (E ^ F) * G) | B * C

~~H | (D / (E ^ F) *~~

\ | H | ((B * C) - (D / (E ^ F) * G))

* | ((B * C) - (D / (E ^ F) * G)) * H

A | ((B * C) - ((D / (E ^ F) * G)) * H | A

+ (A + ((B * C) - ((D / (E ^ F) * G)) * H)

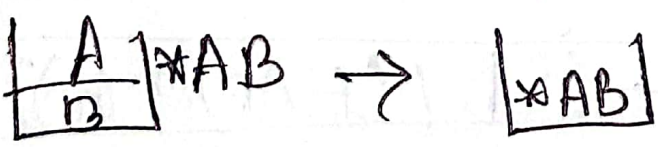
Postfix to Prefix

① Scan the postfix from left to right

② If the scan symbol is an operand

Push it onto the stack.

③ If the scanned symbol is an operator pop two operand from stack and creat it as a string by placing the operand infront of the operands and push it onto the stack

Ex: AB* → 

④ Repeat Step ② & ③.

~~If~~ If Operator

Given Expression:

ABC*DEF^/G*-H*+

Date : / /

Symbol	Stack
A	A
B	AB
C	ABC → *BC
*	A*BC
D	A*BCD
E	A*BCDE
F	A*BCDEF → ^EF
^	A*BCD^EF → /D^EF
/	A*BC/D^EF
G	A*BC/D^EFG → *C/D^EFG
*	A-BC*/D^EFG
-	A-*BC*/D^EFG
H	A-*BC*/D^EFGH
*	A-*BC*/D^EFGH
+	+A*-*BC*/D^EFGH

Prefix to POSTfix

1. Scan from ~~Right~~ to left R→L
2. If scanned symbol is operand push it to stack

(3) If the scanned symbol is an operator pop two operator from the stack and create it as a string by placing the operator after the operands and push it to stack

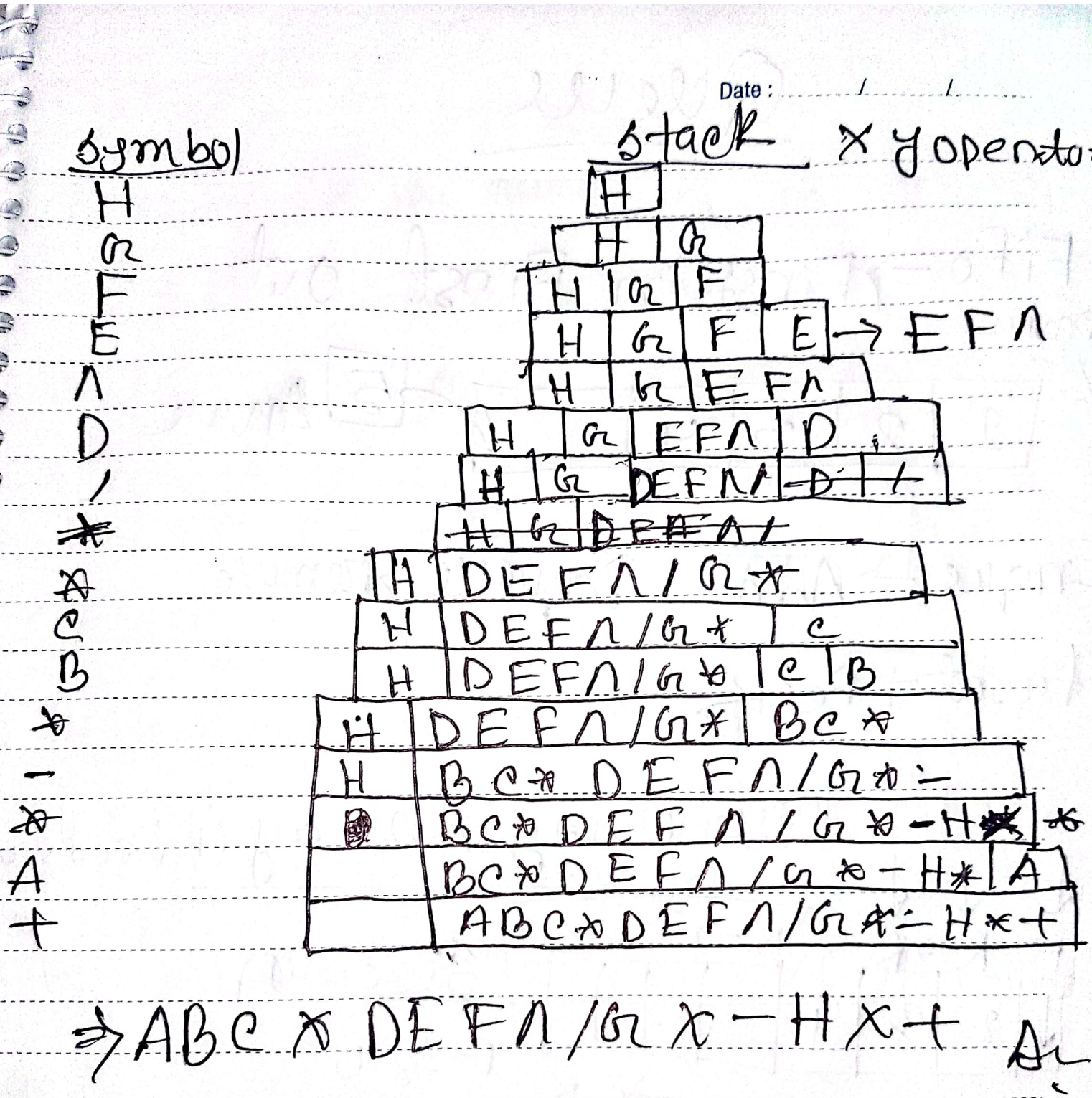
AB [B A] AB

(iv) Repeat step (3) & (2) -

Evaluate : +A* - *B@*/D^EFG#H

X = E Y = F X^Y operator

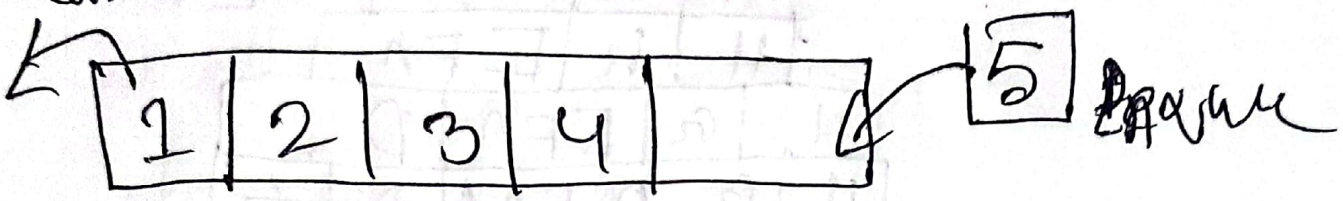
Date : / /



Queue

FIFO → First in First out

Dequeue

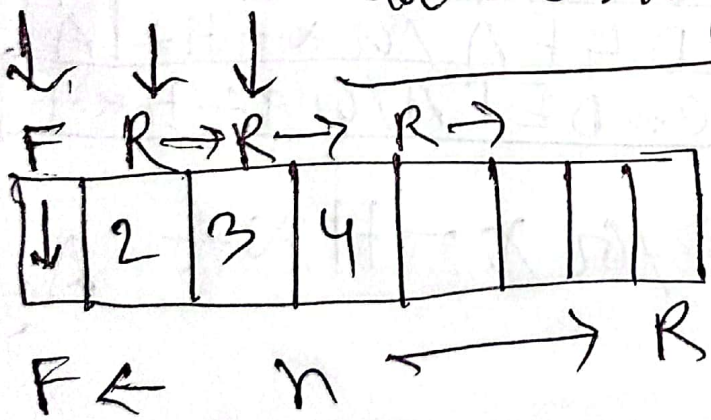


Enque

Enque → Add Dequeue → remove

front → Peek

Queue using Array (Fixed size)



size = 8

F → 0

R → idx of last

R = −1 if empty

ADD

1. Check is Full
2. If queue is Full produce overflow error
3. If not creat a temp node and assigning

Date : / /

Dequeue:

① check is the queue is not empty

② If empty then produce overflow

③ Is not the access the data where front is pointing

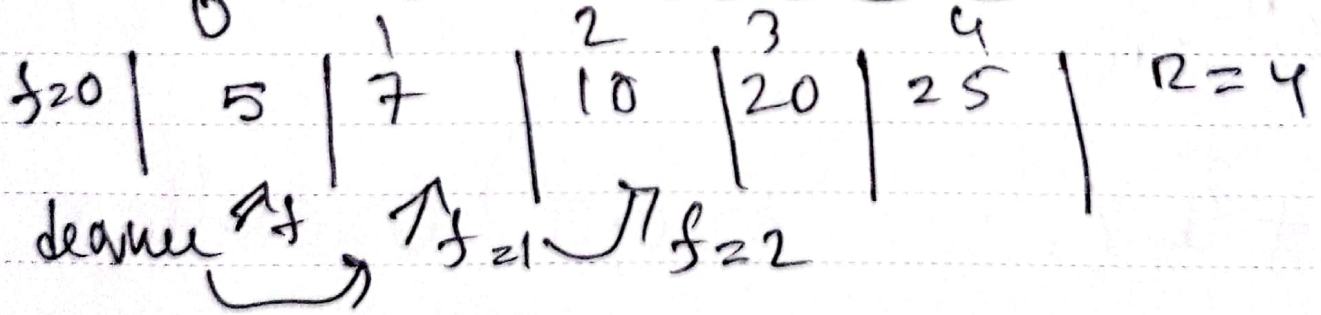
④ Increment the front portion to point to the next available data element.

Overflow: When the Queue is full (Enqueue)

Underflow: Whe the Queue is empty (Dequeue)

① Draw backs

~~Circular Queue~~

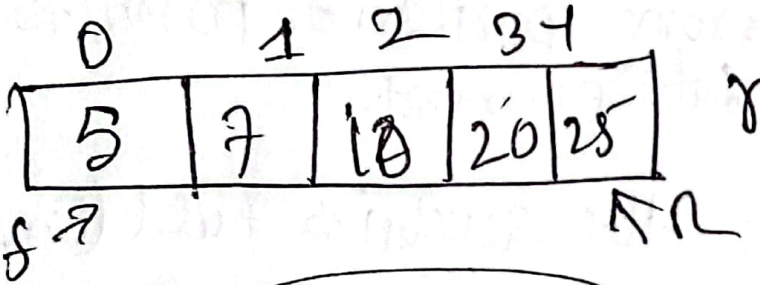


Q 0, 1 index are free but the front is moved to two so we are not using the empty 0 indexes.

Circular Queue

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

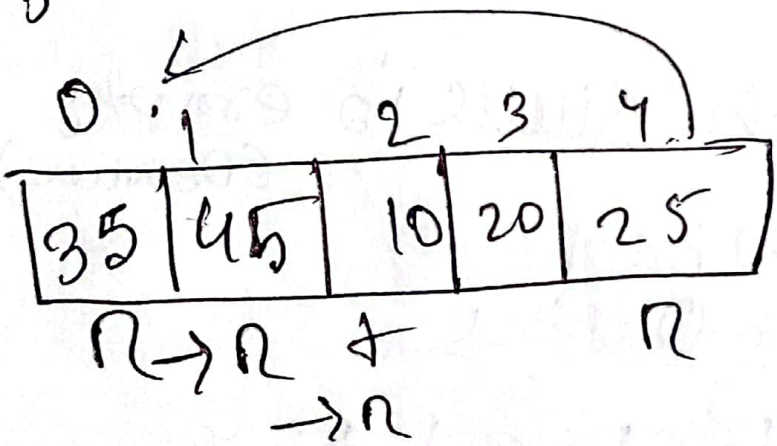
Initially $f = r = -1$



$n = 5$

$r = r - 1$

$4 = 4$



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

$= 5 \% 5$

$= 0$

((r+1) % n == front) overflow

Pseudo code

```
void enque(int x)
if ((rear+1) % N) == front)
    print("overflow")
```

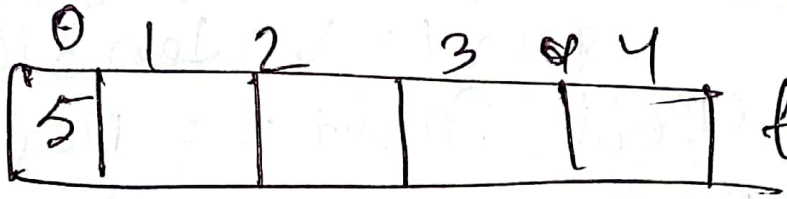
```
else if (front==-1 && rear==-1)
Date : / /
  2 front = rear = 0;
    arr[rear]=x; 2
else2
    rear = (rear+1)%N;
    queue[rear] = x; 2

void dequeue();2
    if (front == -1 && rear == -1)2
        prin("Underflow"); 2
    else if (front == rear)
    2 printf(arr[front]);
      front = rear = -1
    else2 print (queue[front]);
          front = (front-1)%N; 23

void display()2
    int i= front
  if (front==-1 && rear==-1)
      Print("empty");
```

```
else {
    while (i != rear) {
        printf("%d ", queue[i]);
        i = (i + 1) % N;
    }
    printf("%d", queue[rear]);
```

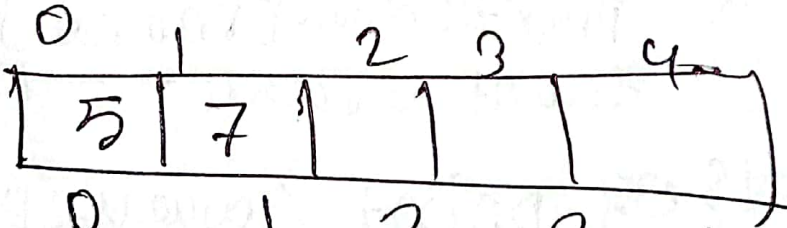
---

f = r - 1

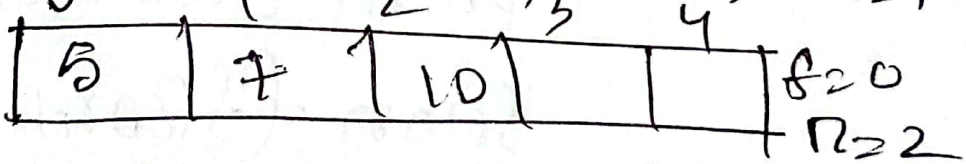
enqueue(5)



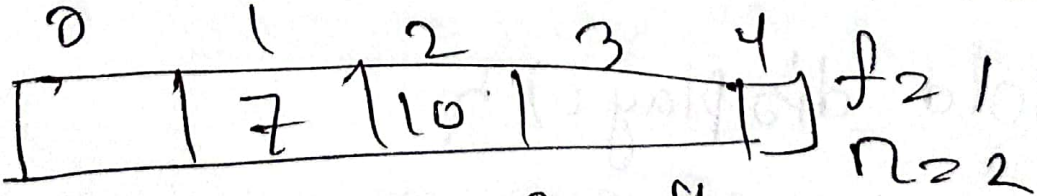
enqueue(7)



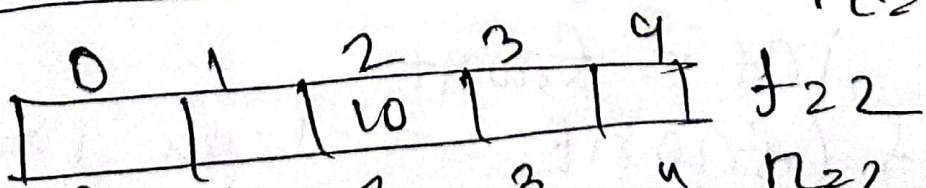
enqueue(10)



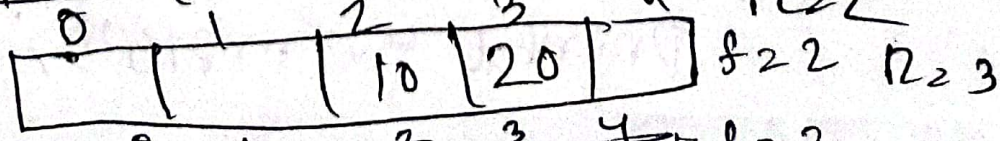
dequeue()



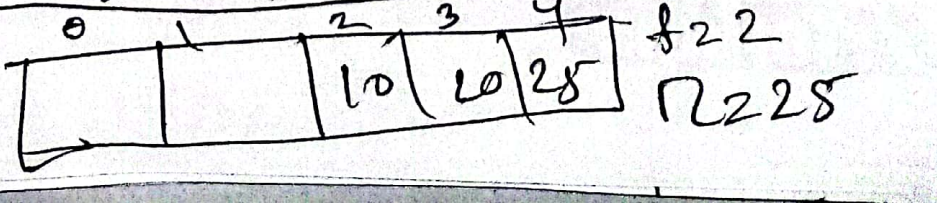
dequeue()



enqueue(20)



enqueue(25)



for 2 $q=3$

$(4+1)\%5) = 0$ Date : / /

	0	1	2	3	4
enqueue (30)	50		10	20	25
enqueue (40)	50	40	10	20	25

Linked List

With Array:

```
struct Node {
    int data;
    struct Node *Next;
};
struct Node *creatLinkedList(int arr[], int size)
{
    struct Node *head = NULL, *temp = NULL, *curr = NULL;
    int i;
    for (i=0; i<size; i++) {
        temp = (struct Node*) malloc (sizeof (struct Node));
        temp->data = arr[i];
        temp->next = NULL;
        if (head == NULL) {
            head = temp;
            current = temp;
        }
        else {
            current->next = temp;
```

curr = current → next ;

return head; }

First data : 10 head = 10

curr = NULL

Second data : 20 curr → Next = temp;

curr = current → next

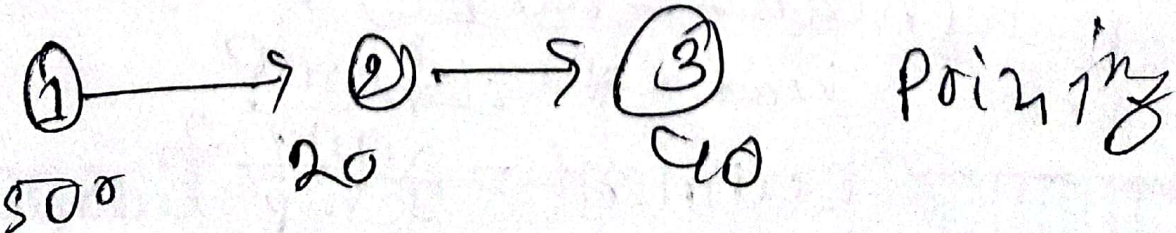
Use of linked list

① Stack & Queue ② Image Gallery

③ Music player ④ Browser

⑤ Hash map & Hash set.

Representation of linked list



Pointing

struct Node {
int val;
Node * next;
};

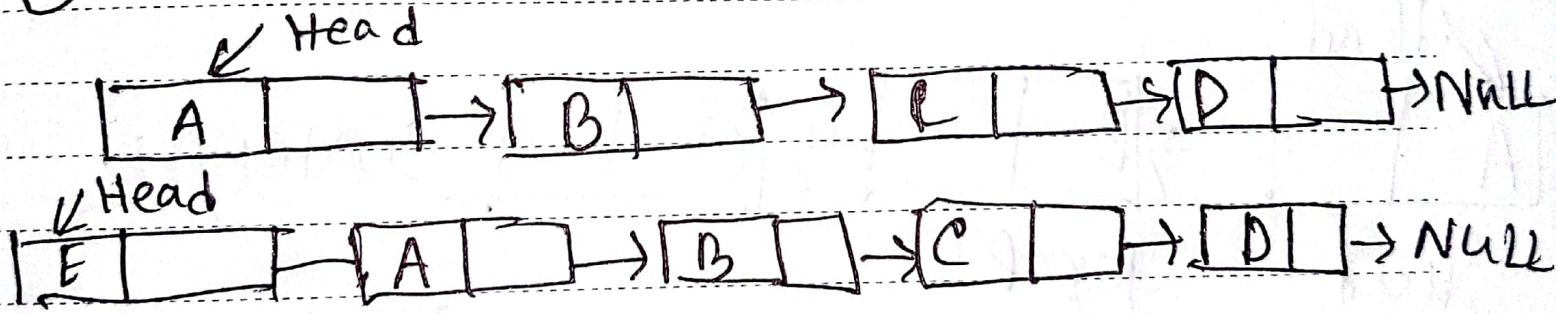
Date : / /

Insertion at head O(1)

① Make the first node of linked list to the new node.

② Remove head from first node

③ Make the new node as the head of l.list.



Insent a Node after a given Node O(1)

① check If the given node exists or not

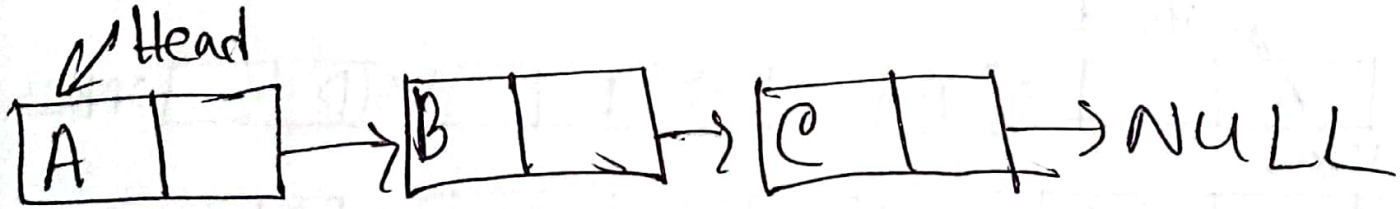
② If it donot exists → terminate the proc

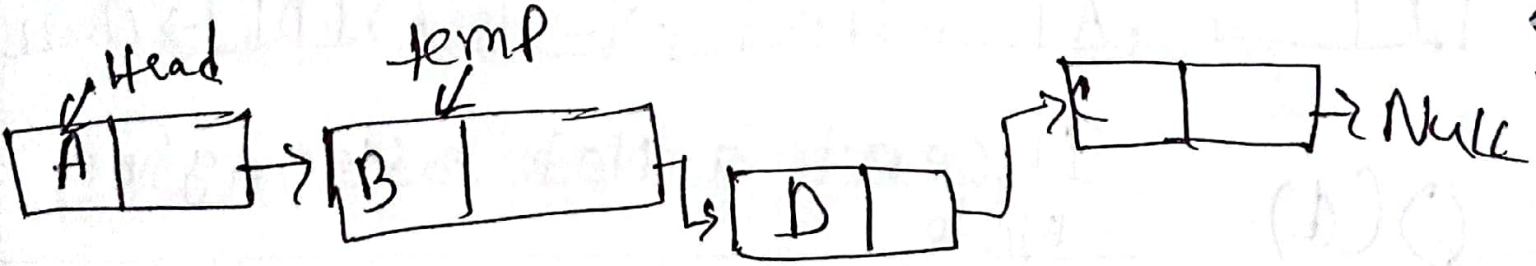
③ If given node exists :

- Make the element to be inserted as a new node

* change the next pointer of given node to the new node

* Now shift the orginal next pointer of given node to the next pointer of new node.





Insertion of Node at the END of Linked List

1. Traverse to end of the linked list
2. change the next pointer of last node to the new node.

Date : / /

③ Make the next pointer node as NULL

Delete at Begining

① Point head to the next node i.e 2nd node

```
temp = head
head = head -> next
```

② Free the temp node

Delete at end

① Traverse to last element of Node

② change the prev node to NULL.

```
While (end -> next) {
    prev = end;
    end = end -> next }
prev -> next = NULL;
free (end); // delete
```

Delete from Middle

Keeps the track of pointer before node to delete and pointer to node to delete.

```
temp = head;
prev = head;
for( i=0  i<position; i++)
    if(i==0 && pos==1)
    head = head->next;
     free(temp); }
    else { if(i==pos-1 && temp)
     prev->ptr = temp->next;
      free(temp); }
    else else { prev=temp;
      if(prev==NULL)
```

Date : / /

// position was greater than number of nodes in the list).
break;
temp = temp → next;

222

Searching

① Set Ptr = head
Set I = 0

② If PTR = NULL
throw error "empty list";
Goto step 7
END OF IF

③ Repeat step 5 & 7 until PTR != NULL

④ IF Ptr → data == item
write i+1
end of if

⑤ i = i+1

⑥ Ptr = PTR → Next

⑦ EXIT