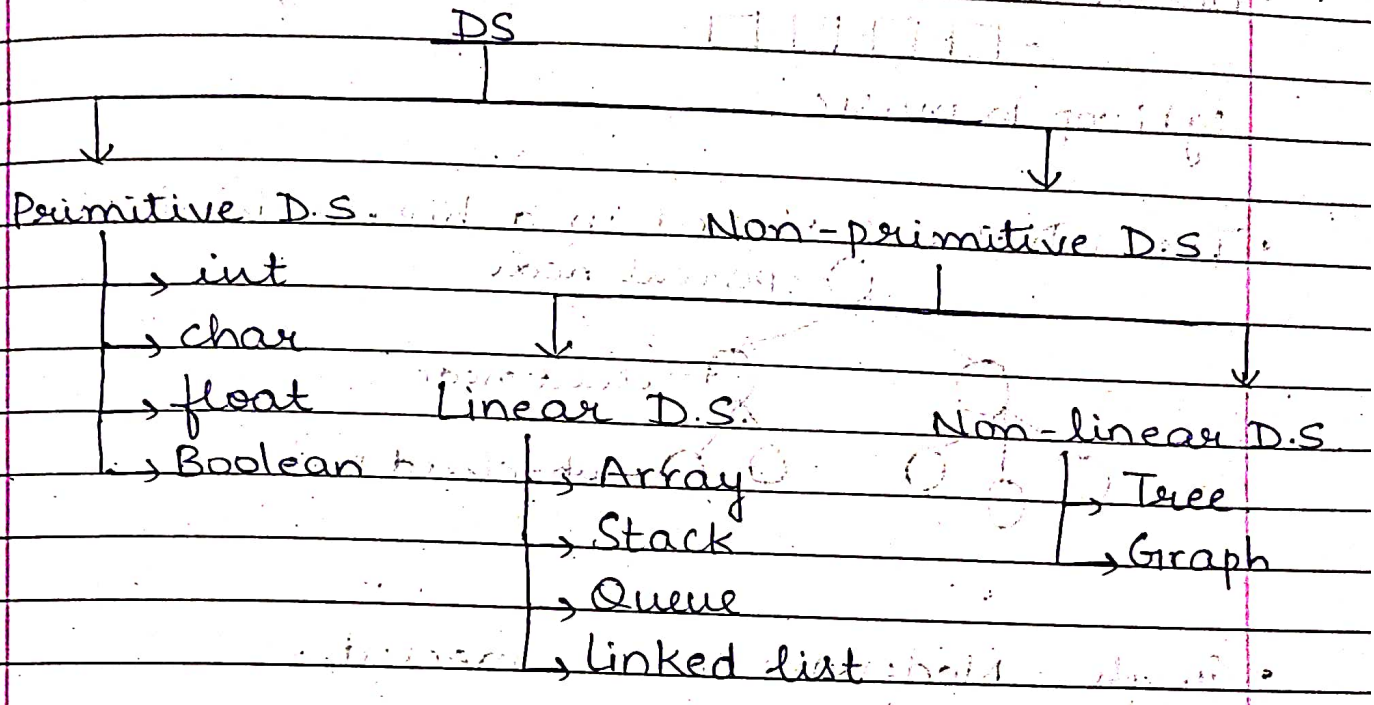
28/04/'22. DATA STRUCTURES

Page No. 01 DOLLY
Date:

- Data is the collection of raw facts and figures.
- Information is processed meaningful data.
- We organize data to reduce processing time and increase speed.
- Data structures are a set of algorithms written in a programming language which allow us to organize our data in an efficient manner in the main memory or primary memory.



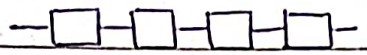
- Linear D.S - A way of structuring or organizing the data in a linear form.
- Non-linear D.S - No particular order is followed while organizing the data.
- Operations on D.S. -

1. Insertion
2. Deletion
3. Update
4. Merge
5. Sort

- Stack : Stack follows LIFO
  Last In First Out
  eg) Stack of books, undo, back button on websites.

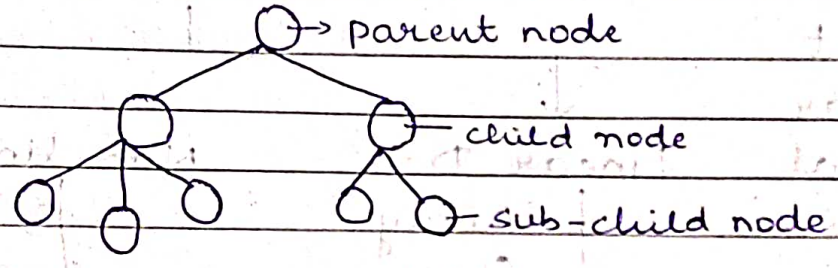
- Queue : Queue follows FIFO
  First In First Out
  eg) Any queue, playlist

- Linked list : Data is stored in the form of nodes.



  eg) Loop in music.

- Tree : Data is stored in a hierarchial structure.



- Graph :- Nodes are interconnected.
  eg) google maps.

- A program and an algorithm are step by step instructions to solve a problem. Program uses syntax, while algorithm is written in english.

ARRAYS

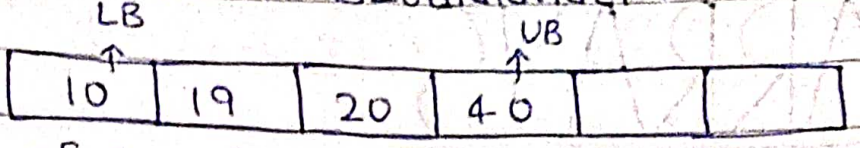
- An array is a linear data structure.
- It is a collection of similar types of data types stored in contiguous memory location.
- Index from 0 to n-1
- Operations performed on an array are:
  1. Traversing
  2. Insertion
     - → At beginning
     - → At end
     - → At specific location.
  3. Deletion
     - → At beginning
     - → At end
     - → At specific location.
  4. Sorting
  5. Searching
     - → linear search
     - → Binary search
  6. Merging.

① INSERTION : When an element is inserted in an array, it is called insertion.

- When we want to insert an element in an array that is full, it is in overflow condition.
- If overflow condition is false then we enter the element.

29/04/'22.

* INSERTION AT BEGINNING :



- LB : Lower bound (Index 0)
  UB : Upper bound (Index n-1)
  where, n = no. of elements.
- when UB = m-1, the array is full and in overflow condition. Here, m is max. size of array.
- For insertion at beginning, the element will be entered at LB.

```
i = UB
while (i >= LB)
{ a[i+1] = a[i]          } This clears the first
  i--                    } position.
}
a[LB] = element
   n = n+1
```

- Algorithm :-

Step 1 : Start
Step 2 : Let "a" be an array of max. size m containing 'n' elements.
Step 3 : If UB = m-1, then print overflow and exit
Step 4 : Read element
Step 5 : i = UB (UB = n-1)
Step 6 : Repeat step 7 until condition while (i >= LB) is true.
Step 7 : a[i+1] = a[i]

i--
Step 8: a[LB] = element
Step 9: n = n+1
Step 10: Stop

* INSERTION AT END :-

• Algorithm -

Step 1 : Start
Step 2 : let "a" be an array of max. size 'm', containing 'n' elements.
Step 3 : If UB = m-1, then print overflow and exit
Step 4 : Read element.
Step 5 : UB = UB + 1 (where, UB = n-1)
Step 6 : a[UB] = element.
Step 7 : n = n+1
Step 8 : Stop.

* INSERTION AT SPECIFIC POSITION

• Algorithm :

Step 1 : Start
Step 2 : Let "a" be array of 'm' size and 'n' element
Step 3 : If UB = m-1, then print overflow and exit
Step 4 : Read element
Step 5 : Read position
Step 6 : i = UB
Step 7 : Repeat step 8 until condition, while (i >= position) is true

Step 8: a[i+1] = a[i]
i--
Step 9: a[position] = element
Step 10: n = n+1
Step 11: Stop

② TRAVERSING :-

• Algorithm -

Step 1 : Start
Step 2 : Let "a" be an array of max. size 'm' h
'n' elements.
Step 3 : i = 0
Step 4 : Repeat step 5 until the condition
while (i ≤ n) becomes false.
Step 5 : Process the element
i++
Step 6 : Stop.

③ DELETION :-

* DELETION AT END :

• Algorithm -

Step 1 : Start
Step 2 : Let "a" be an array of max. size '
having 'n' elements.
Step 3 : If UB = 0/NULL, print underflow and

02/05/'22.

Step 4 : a[UB] = NULL (UB = n-1)
Step 5 : UB = UB - 1
Step 6 : Stop

* DELETION AT BEGINNING :

• Algorithm -

Step 1 : Start
Step 2 : Let "a" be an array of max. size 'm' having 'n' elements
Step 3 : If UB = 0/NULL, print underflow and exit
Step 4 : i = LB
Step 5 : Repeat step 6 until condition while (i<UB) is satisfied.
Step 6 : a[i] = a[i+1]
i = i+1
Step 7 : a[UB] = NULL (where, UB = n-1)
Step 8 : UB = UB - 1
Step 9 : Stop

* DELETION AT SPECIFIC POSITION :

• Algorithm -

Step 1 : Start
Step 2 : let "a" be an array of max. size 'm' having 'n' elements
Step 3 : If UB = 0/NULL, print underflow and exit
Step 4 : Read position
Step 5 : i = LB

Step 6 : Repeat step 7 until condition while (i != position) becomes false.
Step 7 : i++
Step 8 : Repeat step 9 until condition while (i<UB) becomes false
Step 9 : a[i] = a[i+1]
i = i+1
Step 10 : a[UB] = NULL
Step 11 : UB = UB - 1
Step 12 : Stop

④ SEARCHING

* LINEAR SEARCH :

• Algorithm -

Step 1 : Start
Step 2 : Let "a" be an array of size 'n'
Step 3 : Read element to be searched.
Step 4 : Repeat i = 0
Step 5 : Repeat step 6 while (i<n)
Step 6 : if (x == a[i]
return success
else
return failure.
Step 7 : Stop.

05/05/22.

DOLLY
Page No. 09
Date:

* BINARY SEARCH :

- Algorithm -

Step 1 : Start
Step 2 : Let "a" be an array of size 'n'. (sorted)
Step 3: Read element to be searched (x).
Step 4 : L = 0, H = n-1
Step 5: Repeat step 6 while (L < H)
Step 6 : $mid = \frac{L+H}{2}$

```
if x = a[mid]
        return success
else
        if x > a[mid]
                L = mid + 1
        else
                H = mid - 1
```

Step 7: If element is not found, return failure.
Step 8: Stop.

* INSERTION ALGORITHM FOR END, SP & BEGINNING:

Step 1 : Insert (value, pos)
Step 2 : Set I = UB + 1
Step 3 : Repeat step 4 while (I > pos)
Step 4 : A[I] = A[I+1]
I = I - 1
[End of loop]
Step 5 : A[pos] = value
Step 6 : UB = UB + 1
Step 7 : Stop.

★ Calculating the Address of Array Elements :-

Address of data element, A[K] = BA(A) + w(K - lower-bound)

where,
K : index of element
BA : base address of array A
w : size of one element in memory.

Q. Given an array,
int marks = {99, 67, 78, 56, 88, 90, 34, 85},
calculate the address of marks[4] if the base address = 1000.

→ Address of data element,
A[K] = BA(A) + w(K - lower-bound)
marks[4] = 1000 + 2(4 - 0)
= 1000 + 8
= 1008.

* PROGRAM FOR OPERATIONS ON ARRAY :

```
#include<stdio.h>
#include<stdlib.h>
int insertion(int DATA[10], int n, int ITEM, int LOC);
int deletion(int DATA[10], int n, int ITEM, int LOC);
void selection(int DATA[10], int n, int ITEM, int LOC);
void display(int DATA[10], int n, int ITEM, int LOC);
void searching(int DATA[10], int n, int ITEM, int LOC);
int main()
{
    int DATA[10], n, ITEM, LOC, j;
    printf("Enter number of elements :");
    scanf("%d", &n);
    printf("Enter elements of array :");
    for(j=0, j<n; j++)
    {
        scanf("%d", &DATA[j]);
    }
    selection(DATA, n, ITEM, LOC);
    return 0;
}
void selection(int DATA[10], int n, int ITEM, int LOC)
{
    int CHOICE, p;
    printf("\n Press 1 for insertion at beginning. \n
    Press 2 for insertion at end. \n Press 3 for
    insertion at specific position. \n Press 4 to
    display. \n Press 5 for deletion at beginning. \n
    Press 6 for deletion at end. \n Press 7 for
    deletion at specific position. \n Press 8 for
```

```
searching element\n Press 0 to exit.");
scanf("%d", &CHOICE);
switch(CHOICE)
{
    case 1 :
        printf(" Enter element to be inserted
        scanf("%d", &ITEM);
        LOC = 0;
        n = insertion(DATA, n, ITEM, LOC);
        display(DATA, n, ITEM, LOC);
        selection(DATA, n, ITEM, LOC);
        break;
    case 2:
        printf("Enter element to be inserted:
        scanf("%d", &ITEM);
        LOC = n;
        n = insertion(DATA, n, ITEM, LOC);
        display(DATA, n, ITEM, LOC);
        selection(DATA, n, ITEM, LOC);
        break;
    case 3:
        printf("Enter element to be inserted:");
        scanf("%d", &ITEM);
        printf("Enter location :");
        scanf("%d", &p);
        LOC = p-1;
        n = insertion(DATA, n, ITEM, LOC);
        display(DATA, n, ITEM, LOC);
        selection(DATA, n, ITEM, LOC);
        break;
    case 4:
```

```
            display (DATA, n, ITEM, LOC);
            selection (DATA, n, ITEM, LOC);
            break;
        case 5:
            LOC = 0;
        n= deletion(DATA, n, ITEM, LOC);
            display (DATA, n, ITEM, LOC);
            selection (DATA, n, ITEM, LOC);
            break;
        case 6:
            LOC = n-1;
        n= deletion (DATA, n, ITEM, LOC);
            display (DATA, n, ITEM, LOC);
            Selection (DATA, n, ITEM, LOC);
            break;
        case 7:
            printf ("Enter position of element to be
                deleted: ");
            scanf ("%d", &p);
            LOC = p-1;
        n= deletion (DATA, n, ITEM, LOC);
            display (DATA, n, ITEM, LOC);
            selection (DATA, n, ITEM, LOC);
            break;
        case 8:
            searching (DATA, n, ITEM, LOC);
            Selection (DATA, n, ITEM, LOC);
            break;
        case 0:
            exit (0);
}
```

```
}
int insertion(int DATA[10], int n, int ITEM, int LOC
{
    int i = n-1;
    while (i >= LOC)
    {
        DATA[i+1] = DATA[i];
        i--;
    }
    DATA[LOC] = ITEM;
    n = n+1;    selection(DATA, n, ITEM, LOC);
    return n;
}
int deletion(int DATA[10], int n; int ITEM, int LOC)
{
    int i = LOC;
    while (i<n-1)
    {
        DATA[i] = DATA[i+1]
        i++;
    }
    DATA[n-1] = NULL;
    n = n-1;    selection(DATA, n, ITEM, LOC);
    return n;
}
void display(int DATA[10], int n, int ITEM, int L
{
    int j;
    for(j=0; j<n; j++)
    {
        printf("%d    ", DATA[j]);
```

```
    }
    selection(DATA, n, ITEM, LOC);
}
void searching(int DATA[10], int n; int ITEM, int LOC
{
  int i, K=0;
  printf("Enter element to be found:");
  scanf("%d", &ITEM);
  for(i=0; i<n; i++)
  {
    if (DATA[i] == ITEM)
    {
      K=1;
    }
  }
  if (K==1)
  printf("Element found.");
  else
  printf("Element not found.");
  selection(DATA, n, ITEM, LOC);
}
```

05/05/22 UNIT 2 :-

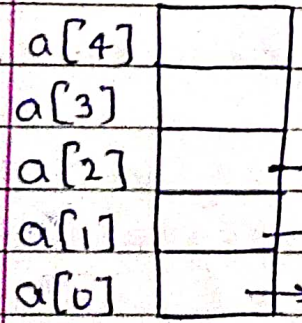
STACKS

- A stack is a linear data structure.
- It is called a LIFO (Last In First Out) data structure, as the element that was inserted last is the first one to come out.
- Stacks can be implemented using either arrays or linked lists.

* OPERATIONS ON A STACK :
  1) Push (Insertion at top)
  2) Pop (Deletion at top)
  3) Peep (View element at top)

- Push :- To insert we check overflow
  If top = n-1, the stack is in overflow.
  If overflow condition is false,
  we increment top and then enter element.

- Pop :- To delete we check underflow
  If top = -1, the stack is underflow.
  If underflow condition is false,
  we delete element at top and then decrement



- Algorithm for Push (Insertion):

Step 1: Start
Step 2: Let "a" be a stack of 'n' elements
Step 3: If TOP = n-1, print overflow and exit
Step 4: Read element to be inserted
Step 5: TOP = TOP + 1
Step 6: Insert element at top.
stack [TOP] = element
Step 7: Stop

- Algorithm for Pop (Deletion):

Step 1: Start
Step 2: Let "a" be a stack of 'n' elements.
Step 3: If TOP = -1, print underflow and exit
Step 4: Delete element at TOP
stack [TOP] = NULL
Step 5: TOP = TOP - 1
Step 6: Stop.

- Algorithm for Peep:

Step 1: Start
Step 2: Let "a" be a stack of 'n' elements
Step 3: If TOP = -1, print underflow and exit
Step 4: Print element with index TOP
Step 5: Stop

09/05/'22.

* EVALUATION OF ARITHMETIC EXPRESSIONS

1. Infix : a+b : Operator is between operands
2. Prefix : +ab : Operator is before operands.
3. Postfix : ab+ : Operator is after operands.

⇒ Conversion of infix to postfix :

1. (a+b) * c
   ab+ * c
   ab+c *

2. (A+B) / (C+D) - (D * E)
   AB+ / (C+D) - (D * E)
   AB+ / CD+ - (D * E)
   AB+ / CD+ - DE *
   AB+CD+ / - DE *
   AB+CD+/DE * -

3. A + B - C * D * E ^ F ^ G
   A + B - C * D * E ^ FG^
   A + B - C * D * EFG^^
   A + B - CD* * EFG^^
   A + B - CD*EFG^^*
   AB+ - CD*EFG^^*
   AB+CD*EFG^^*-

4. a + b * (c^d - e) ^ (f + g * h) - i
   a + b * (cd^ - e) ^ (f + g * h) - i
   a + b * (cd^e -) ^ (f + g * h) - i
   a + b * cd^e - ^ (f + gh*) - i

a+b* cd^e- ^ fgh*+ -i
a+b* cd^e-fgh*+^ -i
a+ bcd^e-fgh*+^* -i
abcd^e-fgh*+^*+ -i
abcd^e-fgh*+^*+i-

5. a+b+(c^d)*e/f/g*h+i
a+b+ cd^ *e/f/g*h+i
a+b+ cd^ * ef//g*h+i
a+b+cd^ * ef/g/*h+i
a+b+ cd^ef/g/* *h+i
a+b+ cd^ef/g/*h*+i
ab+ + cd^ef/g/*h*+i
abcd^ef/g/*h*+ +i
abcd^ef/g/*h*+i+

6. A-(B/C+(D%(E*F)/G)*H)
A-(B/C+(D% EF*/G)*H)
A-(B/C+(DEF*%/G)*H)
A-(B/C+(DEF*%G/)*H)
A-(BC/+DEF*%G/*H)
A-(BC/+DEF*%G/H*)
A-BC/DEF*%G/H*+
ABC/DEF*%G/H*+-

⇒ Conversion from infix to prefix:

1. (A+B) * C
   +AB * C
   * +ABC

2. (A+B) / (C+D) - (D*E)
   +AB / (C+D) - (D*E)
   +AB / +CD - (D*E)
   +AB / +CD - *DE
   /+AB+CD - *DE
   -/ +AB+CD*DE

3. A+B-C*D*E^F^G
   A+B-C*D*E^F^G
   A+B-C*D* ^E^FG
   A+B- *CD * ^E^FG
   A+B- **CD^E^FG
   +AB - **CD^E^FG
   -+AB**CD^E^FG

4. a+b*(c^d-e)^(f+g*h)-i
   a+b* (^cd-e)^(f+g*h)-i
   a+b* (-^cde)^(f+g*h)-i
   a+b* -^cde ^ (f+ *gh)-i
   a+b* -^cde ^ +f*gh -i
   a+b* ^-^cde+f*gh -i
   a+ * b^-^cde+f*gh -i
   +a*b^-^cde+f*gh -i
   -+a*b^-^cde+f*ghi

5. a+b+(c^d)*e/f/g*h+i
a+b+(^cd)*e/f/g*h+i
a+b+^cd*/ef/g*h+i
a+b+^cd*//efg*h+i
a+b+*^cd//efg*h+i
a+b+**^cd//efgh+i
+ab+**^cd//efgh+i
++ab**^cd//efgh+i
+++ab**^cd//efghi

6. A-(B/C+(D%(E*F)/G)*H)
A-(B/C+(D%(*EF)/G)*H)
A-(B/C+(%D*EF/G)*H)
A-(B/C+/%D*EFG*H)
A-(/BC+/%D*EFG*H)
A-(/BC+*/%D*EFGH)
A-+/BC*/%D*EFGH
-A+/BC*/%D*EFGH

⇒ While doing this conversion or conversion using stack, always draw a priority chart with associativity.

11/05/22.

* INFIX TO POSTFIX USING STACK

⊙ Algorithm :-

- Step 1 : If stack is empty or it contains the left parenthesis on top, push the incoming operator on stack.
- Step 2 : If the incoming symbol is opening parenthesis C, push it into the stack.
- Step 3 : If the incoming symbol is closing parenthesis pop the stack and print the operator until the opening parenthesis C is found.
- Step 4 : If the incoming symbol has higher precedence than top of the stack then push it in stack.
- Step 5 : If the incoming symbol has lower precedence than top of the stack then print the top.
- Step 6 : If the incoming symbol has equal precedence with top of the stack, use the associativity rule.
- Step 7 : If you come to the end of the expression, pop and print all the operators of the stack.

Q. a+b*(c^d-e)^(f+g*h)-i

→

Scanned Element	Stack	Postfix Expression
a		a
+	+	a
b	+	ab
*	+*	ab
(	+*(	ab
c	+*(	abc
^	+*(^	abc
d	+*(^	abcd
-	+*(-	abcd^
e	+*(-	abcd^e
)	+*	abcd^e-
^	+*^	abcd^e-
(	+*^(	abcd^e-
f	+*^(	abcd^e-f
+	+*^(+	abcd^e-f
g	+*^(+	abcd^e-fg
*	+*^(+*	abcd^e-fg
h	+*^(+*	abcd^e-fgh
)	+*^	abcd^e-fgh*+
-	-	abcd^e-fgh*+^*+
i	-	abcd^e-fgh*+^*+i

Ans: abcd^e-fgh*+^*+i-

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

Scanned Element	Stack	Postfix Expression
A		A
+	+	A
(	+ (	A
B	+ (	AB
*	+ (*	AB
C	+ (*	ABC
-	+ (-	ABC *
(	+ (- (	ABC * D
D	+ (- (	ABC * D
/	+ (- (/	ABC * D
E	+ (- (/	ABC * DE
^	+ (- (/ ^	ABC * DE
F	+ (- (/ ^	ABC * DEF
)	+ (-	ABC * DEF ^ /
+	+ (+	ABC * DEF ^ / -
G	+ (+	ABC * DEF ^ / - G
)	+	ABC * DEF ^ / - G +
*	+ *	ABC * DEF ^ / - G +
H	+ *	ABC * DEF ^ / - G + H
		ABC * DEF ^ / - G + H * +

Ans: ABC * DEF ^ / - G + H * +

Q. a+b+(c^d)*e/f/g*h+i

Scanned element	Stack	Postfix Expression
a		a
+	+	a
b	+	ab
+	+	ab+
(	+(	ab+
c	+(	ab+c
^	+(^	ab+c
d	+(^	ab+cd
)	+	ab+cd^
*	+*	ab+cd^
e	+*	ab+cd^e
/	+*/	ab+cd^e
f	+*/	ab+cd^ef
/	+*/	ab+cd^ef/
g	+*/	ab+cd^ef/g
*	+*	ab+cd^ef/g/*
h	+*	ab+cd^ef/g/*h
+	+	ab+cd^ef/g/*h*+
i	+	ab+cd^ef/g/*h*+i
		ab+cd^ef/g/*h*+i+

Ans : ab+cd^ef/g/*h*+i+

Q. A + B - C * D * E ^ F ^ G

Scanned Element	Stack	Postfix Expression
A		A
+	+	A
B	+	AB
-	-	AB+
C	-	AB+C
*	-*	AB+C
D	-*	AB+CD
*	-*	AB+CD*
E	-*	AB+CD*E
^	-*^	AB+CD*E
F	-*^	AB+CD*EF
^	-*^	AB+CD*EF^
G	-*^	AB+CD*EF^G
		AB+CD*EF^G^*-

Ans : AB+CD*EF^G^*-

Q. a+b*(c^d-e)^(f+g*h)-i

→

Scanned element	Stack	Postfix Expression
a		a
+	+	a
b	+	ab
*	+*	ab
(	+*(	ab
c	+*(	abc
^	+*(^	abc
d	+*(^	abcd
-	+*(-	abcd^
e	+*(-	abcd^e
)	+*	abcd^e-
^	+*^	abcd^e-^
(	+*^(	abcd^e-^
f	+*^(	abcd^e-^f
+	+*^(+	abcd^e-^f
g	+*^(+	abcd^e-^fg
*	+*^(+*	abcd^e-^fg
h	+*^(+*	abcd^e-^fgh
)	+*^	abcd^e-^fgh*+
-	-	abcd^e-^fgh*+^*+
i	-	abcd^e-^fgh*+^*+i
		abcd^e-^fgh*+^*+i-

Ans: abcd^e-^fgh*+^*+i-

Q. A - (B / C + (D % (E * F) / G) * H)

→

Scanned element	Stack	Postfix Expression
A		A
-	-	A
(	-(	A
B	-(	AB
/	-(/	AB
C	-(/	ABC
+	-(+	ABC/
(	-(+(	ABC/+
D	-(+(	ABC/+D
%	-(+(%	ABC/+D
(	-(+(%(	ABC/+D
E	-(+(%(	ABC/+DE
*	-(+(%(*	ABC/+DE
F	-(+(%(*	ABC/+DEF
)	-(+(%	ABC/+DEF*
/	-(+/	ABC/+DEF*%
G	-(+/	ABC/+DEF*%G
)	-(+	ABC/+DEF*%G/
*	-*	ABC/+DEF*%G/
H	-*	ABC/+DEF*%G/H
)	-*	ABC/+DEF*%G/H
		ABC/+DEF*%G/H*-

Ans : ABC/+DEF*%G/H*-

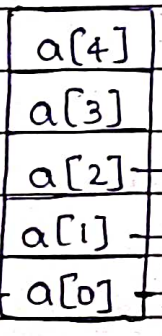
12/05/22 UNIT 3:

QUEUE

Q. What is a queue?

→ A queue is a linear data structure.

- It is called a FIFO (First In First Out) data structure as the element that is inserted first is the first one to be removed.
- There are 4 types of queues:
  1) Linear queue
  2) Circular queue
  3) Priority queue
  4) Double Ended queue
- The operations that can be performed on a queue are :- i) Enequ (Insertion) and ii) Dequ (Deletion). For this, variable 'front' and 'rear' are used.
- Queues can be implemented using either arrays or linked lists.



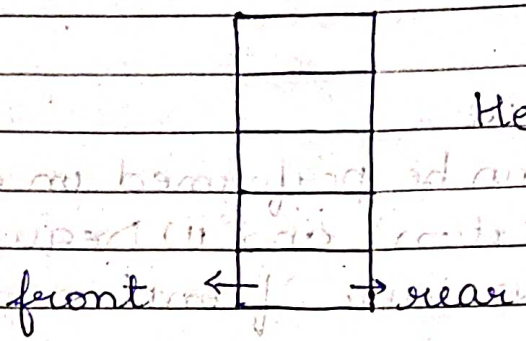
a[2] → Rear when queue has 2 elements
a[1] → Rear when queue has 1 elemen
Front when queue is empty ← a[0] → Rear when queue is empty

- When we insert an element, rear is incremented and when we delete an element, front is incremented.
- When front = rear, the queue has only one element

- Conditions for underflow:
  1. front = -1
  2. front > rear
- Condition for overflow:
  1. rear = max - 1

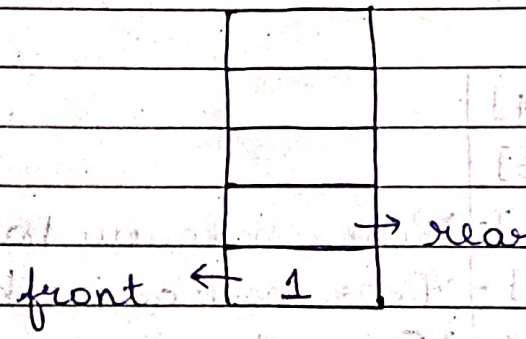
⇒ Let us consider a queue with maximum elem five.

- When the queue is empty,



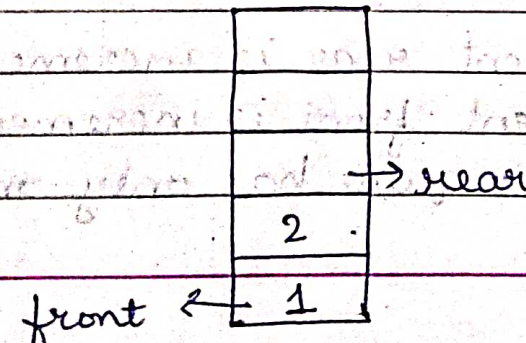
Here,
rear = 0
front = 0

- When we insert element '1', rear is incremen



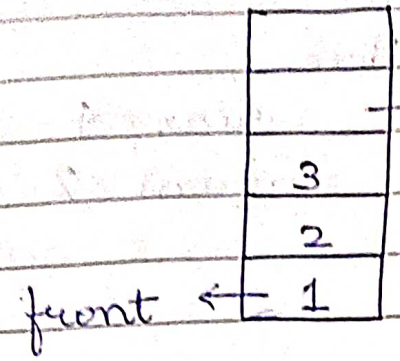
Here,
rear = 1
front = 0

- When we insert element '2', rear is incremented



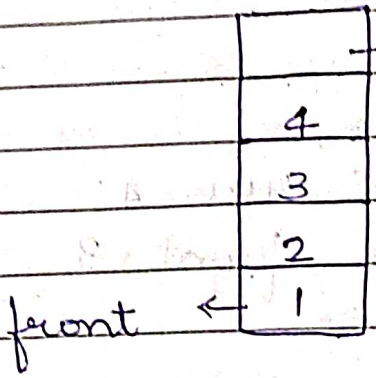
Here,
rear = 2
front = 0

- When we insert element '3', rear is incremented



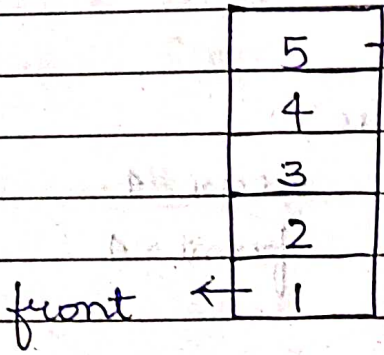
Here,
rear = 3
front = 0

- When we insert element '4', rear is incremented



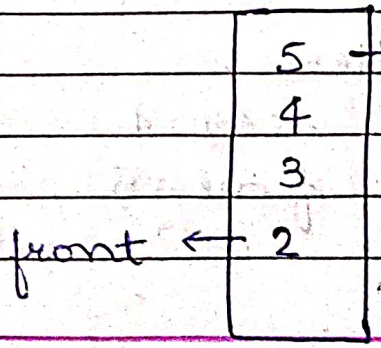
Here,
rear = 4
front = 0

- When element '5' is inserted, the queue is full



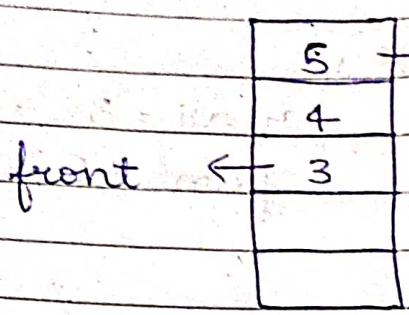
Here,
rear = 4
front = 0

- When we delete an element, front is incremented



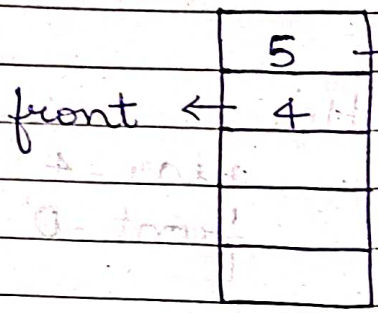
Here,
rear = 4
front = 1

- When another element is deleted,



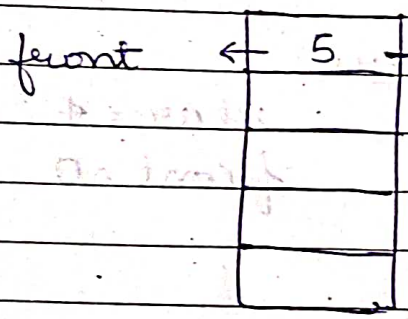
Here,
rear = 4
front = 2

- When another element is deleted,



Here,
rear = 4
front = 3

- When another element is deleted



Here,
rear = 4
front = 4

- When the last element is deleted,

Here,
rear = 4
front = 5

- Here, rear = max - 1, so queue is in overflow condition

- Disadvantage :- A simple queue gives us an overflow condition even when we have empty spaces.

⊙ Algorithm for insertion (Enequeue) :-

Step 1 : Start
Step 2 : Let "a" be a queue of elements 'm'
Step 3 : If rear = max - 1, print overflow and exit.
Step 4 : Read element to be inserted
Step 5 : If front = -1 and rear = -1, then set
front = rear = 0
else,
rear = rear + 1
Step 6 : queue [rear] = element
Step 7 : Stop.

- Algorithm for deletion [Deque] :-

Step 1 : Start
Step 2 : let "a" be a queue of max. size 'm'
Step 3 : If front = -1
or front > rear,
print underflow and exit
Step 4 : queue [front] = NULL
Step 5 : front = front + 1
Step 6 : Stop.

14/05/22

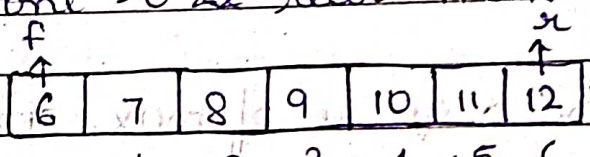
* CIRCULAR QUEUE

⊙ Algorithm for Enqueue (Insertion) in circular q

Step 1: Start
Step 2: Let "a" be a circular queue with 'max' elements.
Step 3: If front = 0 && rear = max-1
or ~~F=0 &&~~ F = (R+1) % max
print overflow and exit.
Step 4: Read element to be inserted.
Step 5: if front = -1 && rear = -1
set front = rear = 0
else if (front != 0 && rear = max-1)
set rear = 0
else
rear = rear + 1
Step 6: queue[rear] = element
Step 7: Stop.

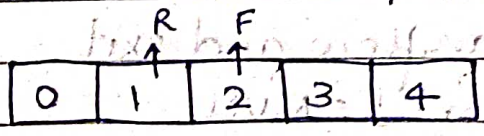
⇒ Overflow conditions:

① front = 0 && rear = max-1



; max = 7

② front = (rear + 1) % max



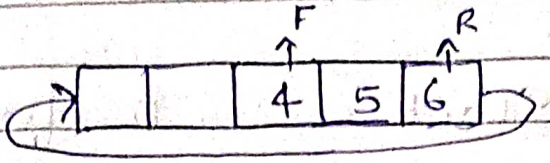
F = (R+1) % max
2 = (1+1) % 5 = 2 % 5

∴ 2 = 2 ⇒ Overflow.

⇒ Step 5:

front != 0 and rear = max-1

set rear = 0

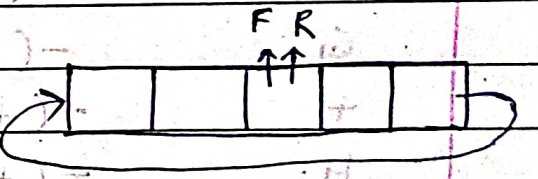


⊙ Alogrithm for Deque (deletion) in circular queue:

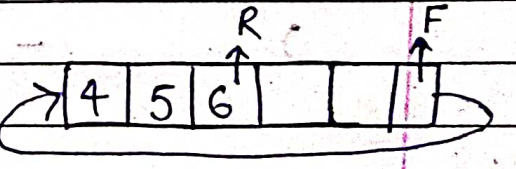
Step 1 : Start
Step 2 : Let "a" be a circular queue with 'max' elements
Step 3 : If front = -1,
print underflow and exit
Step 4 : queue[front] = NULL
Step 5 : if front = rear,
set front = rear = -1
else if front = max-1
set front = 0
else
front = front + 1
Step 6 : Stop

⇒ Step 5 :

① if front = rear
set front = rear = -1



② else if front = max-1
front = 0



* Convert from Infix to Postfix using Stack.

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

Scanned Element	Stack	Postfix Expression
A		A
^	^	A
B	^	AB
/	/	AB^
C	/	AB^C
+	+	AB^C/
(	+(	AB^C/
D	+(	AB^C/D
*	+(*	AB^C/D
E	+(*	AB^C/DE
)	+	AB^C/DE*
^	+^	AB^C/DE*
F	+^	AB^C/DE*F
-	-	AB^C/DE*F^+
(	-(	AB^C/DE*F^+
G	-(	AB^C/DE*F^+G
+	-(+	AB^C/DE*F^+G
H	-(+	AB^C/DE*F^+GH
)	-	AB^C/DE*F^+GH+
		AB^C/DE*F^+GH+-

Ans:- AB^C/DE*F^+GH+-

16/05/'22

- Postfix notation is also called reverse polish notation. It is used to evaluate an expression without parenthesis or priority.

* Expression Evaluation using Stack:

Q. ABC *D+E/- , where, A=9, B=3, C=4, D=8 and E=4.

→

Scanned Element	Stack	Operation
9	9	push 9
3	9 3	push 3
4	9 3 4	push 4
*	9 12	evaluate 3*4
8	9 12 8	push 8
+	9 20	evaluate 12+8
4	9 20 4	push 4
/	9 5	evaluate 20/4
-	4	evaluate 9-5

18/05/'22 UNIT 3 :-

LINKED LIST

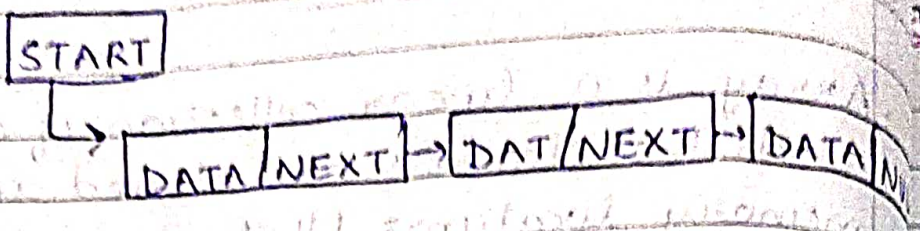
- Array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array which will restrict the number of elements that the array can store.
- To make efficient use of memory, the elements must be stored randomly at any location. So there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition.
- Linked list is a data structure that removes this restriction.

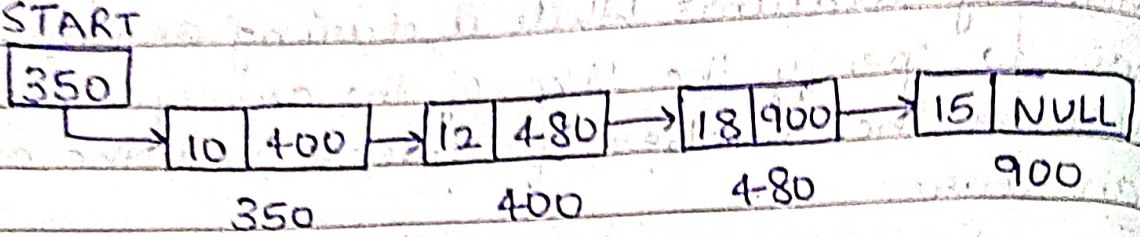
⊙ limitation of linked list :- A linked list does not allow random access of data elements. The elements of linked list can only be accessed in a sequential manner.

- DEFINITION :- A linked list is a linear collection of data elements. These data elements are called nodes. Every node in a linked list is having two parts - data and pointer to the next node connected to it. The last node will have no next node connected to it so it will store a special value called as NULL.

NODE :- | DATA | NEXT/LINK |

Linked List:





DATA NEXT

START : 350

350	10	400
400	12	480
480	18	900
900	15	NULL (-1)

⇒
```
struct node
{
   int data;
   struct node * next;
};
```

⇒ | NAME | MARKS | ID | NULL |

```
struct node
{
   char name[10];
   float marks;
   int id;
   struct node * next;
};
```

19/05/22

Page No. 9-1 DOLLY
Date:

- Declare a pointer START = NULL to indicate that the linked list is initially empty.

For traversal,

```
p = start
while (p != NULL)
    p = p → next
```

⊙ Algorithm for traversing of linked list

Step 1 : Start
Step 2 : if (start = null)
print linked list is empty and exit.
Step 3 : p = start
Step 4 : while (p → next != null)
display data part of node
p = p → next
Step 5 : Stop

⊙ Algorithm for counting number of elements in linked list

Step 1 : Start
Step 2 : if (start = null)
print no elements in list and exit
Step 3 : p = start ; count = 0
Step 4 : while (p → next != null)
{
count ++
p = p → next ;
}
Step 5 : print count as number of elements

Step 6 : Stop.

* INSERTION OF NODES AT BEGINNING

① Create a node
② If start = null; node created will be the only node.
③ Node will be inserted at front.

⊙ Algorithm for insertion of node at beginning.

Step 1 : Start
Step 2: newnode = (struct node *)malloc (sizeof (struct node))
Step 3: Get data from user
Step 4: newnode → link = null
Step 5: if (start = null)
{
newnode → link = null
start = newnode
}
else
{
newnode → link = start
newnode = Start
}
Step 6 : Stop.

* INSERTION OF NODE AT END

⊙ Algorithm for insertion of node at end:

Step 1: Start
Step 2: newnode = (struct node *) malloc (sizeof (struct node))
Step 3: Get data part from user.
Step 4: newnode → link = null
Step 5: if (start = null)
{
newnode → link = null
start = newnode
}
else
{
p = start
while (p → link != null)
{
p = p → link
}
p → link = newnode.
}
Step 6: Stop.