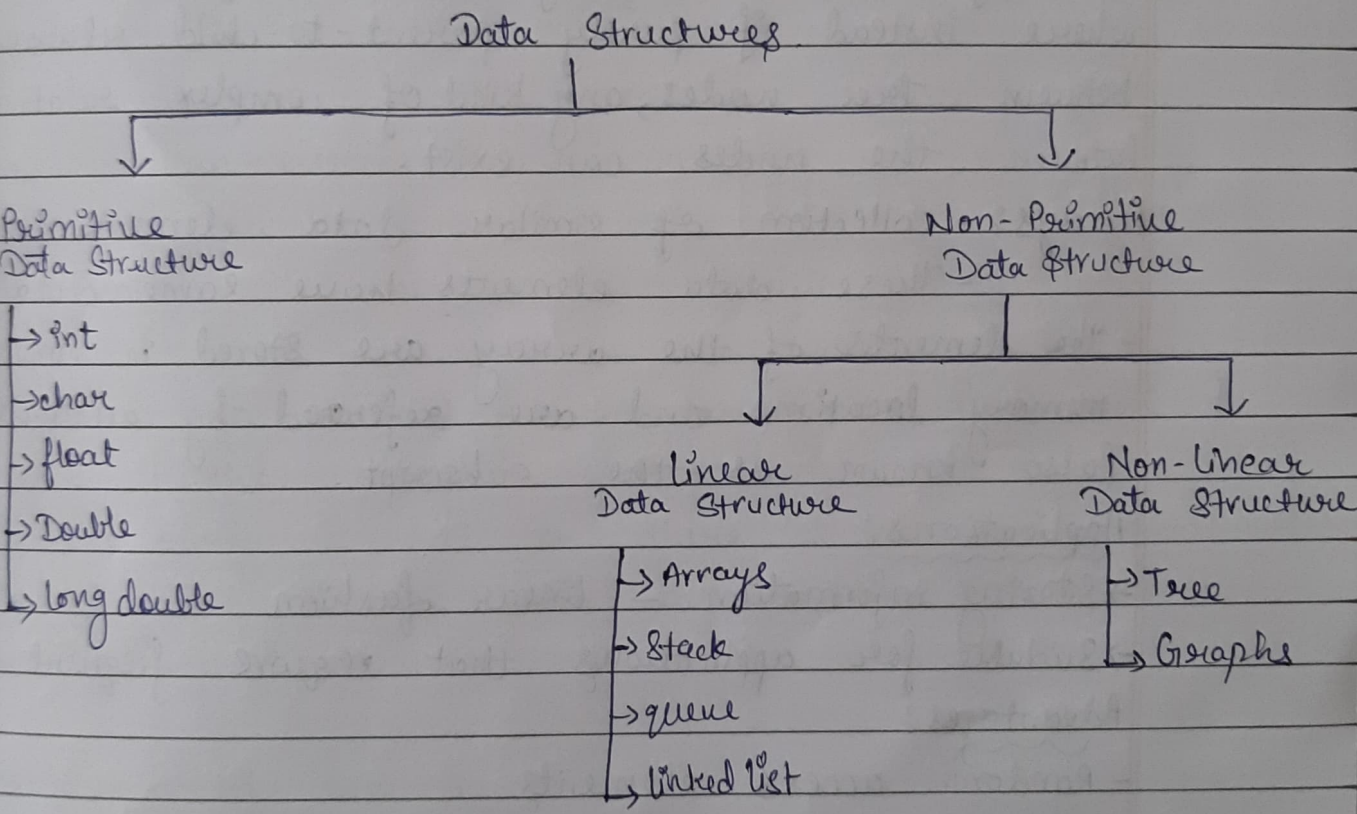
28/04/2022

Data Structures And Algorithms.

Data :- collection of raw facts and figures.
information :- process data
important data

→ Data structure allow us to organise data in an efficient manner in your main memory or in primary memory.
→ A data structure is a crucial part of data management.
→ A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organising data in a computer so that it can be used efficiently.
→ It is not a programming language. It is set of algorithms which allow.



Linear Data Structure : If the elements are stored in a linear or sequential order, then it is a linear data structure.

Non-linear Data Structure: If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.

→ Operation in Data Structure:
- Insertion
- Deletion
- Update
- Merge
- Sort

Stack: Follows LIFO (Last in First Out)
Queue: Follows FIFO (First in First Out)
Linked List: data is stored in form of nodes
Tree: Stored in a hierarchical order.
Graphs: It is a collection of vertices and edges. It is often viewed as a generalization of the tree structure, where instead of purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.
Arrays: - collection of similar data elements.
- These data elements have same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript.

Applications:
→ Storing information in linear fashion
→ Suitable for applications that require frequent searching

Advantages:
- Random access elements
- Easy sorting and Iteration
- Replacement of multiple variables

Disadvantages:
- Size is fixed
- Difficult to insert and delete.
- If capacity is more and occupancy less, most of the array gets wasted.
- Needs contiguous memory

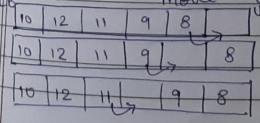
Operations On Arrays
- Traversing an Array
- Inserting an element in an array
- Deleting an element from an array
- Sorting an array in ascending or descending order
- Searching an element in an array
- Merging two arrays.

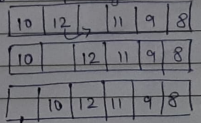
Traversing an Array: Traversing an array means accessing each and every element of the array for a specific purpose.

Inserting an element in an array:
Insertion can be done at
- beginning
- end
- specific position.

→ To insert an element in an array, we first need to check if the array is full or not.
If the array is full or there is no space to insert an element then this condition is known as 'overflow' condition.

Insertion at Beginning
If we want to insert an element in beginning, we need to move each element to right.





LB → lower bound
UB → upper bound

condition for overflow : UB = n-1

LB = lower bound = 0

Algorithm :

Step 1 : Start. let 'a' be the array of size n.

Step 2 : If UB = n-1 then print overflow and exit.

Step 3 : Read element

Step 4 : i = UB

Step 5 : Repeat step-6 until while (i ≥ LB)

Step 6 : a[i+1] = a[i]

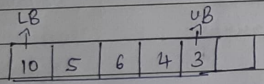
i = i-1

Step 7 : a[LB] = element

Step 8 : n = n+1

step 9 : stop

Insertion at End



Algorithm :

Step 1 : Start

let 'a' be the array of size n.

Step 2 : If UB = n-1 then print overflow and exit

Step 3 : Read element

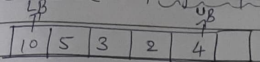
Step 4 : UB = UB + 1

Step 5 : a[UB] = element

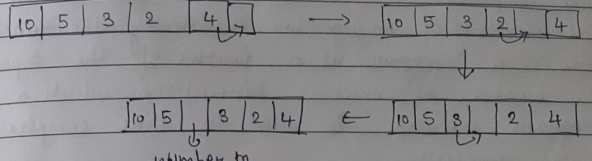
Step 6 : n = n+1

Step 7 : Stop

Insertion at specific position



if at 3rd position i.e. at index 2.



Algorithm :

Step 1 : Start

let 'a' be the array of size n.

Step 2 : if UB = n-1 then print overflow and exit

Step 3 : Read element

i = UB

Step 4 : Enter position where to insert an element

Step 5 : i = UB

Repeat step-6 while i ≥ pos

Step 6 : a[i+1] = a[i]

i = i-1

Step 7 : a[pos] = element

Step 8 : Stop

2/05/2022

Traversing an Array :

Algorithm :

Step 1 : Start

Step 2 : Declare array

Step 3 : i = 0

Step 4 : Repeat step 5 until while (i ≤ n)

Step 5: Processing array element.
i = i+1

Step 6: Stop

Complexity

i) Time Complexity:

→ Time complexity of an algorithm is basically the running time of a program as a function of the input size.

⇒ The number of machine instructions which a program executes is called its time complexity.

ii) Space complexity:

The space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

Cases for Time Complexity:-

Best case: The 'best-case performance' is used to analyse an algorithm under optimal conditions.

Eg:- To insert an element at the end.

Average case: The average-case running time of an algorithm is an estimate of the running time for an 'average' input. Average case running time assumes that all inputs of a given size are equally likely.

Eg:- To insert an element at a specific position

Worst case: This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input.

Eg:- To insert an element at end the beginning

Deleting an element from an array:

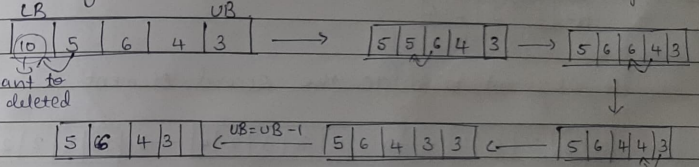
Deletion can be done at
- beginning
- end
- specific position

→ To delete an element from an array, we first need to check if the array is empty or not.

If the array is empty or there is no element present in the array to be deleted then this condition is known as 'underflow' condition.

Deletion at Beginning

If we want to delete an element from an array in beginning, we need to move each element to left.

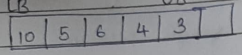


Here, deletion means we are not deleting an element instead we are copying it or say overwriting it

Algorithm:

Step 1: Start
Step 2: Let A be the array of size max.
Step 3: if UB = 0 then print underflow and exit
Step 4: i = LB
Step 5: Repeat step-6 until
while (i < UB)
Step 6: a[i] = a[i+1]
i = i+1
Step 7: a[UB] = NULL
Step 8: UB = UB-1
Step 9: Stop

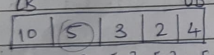
Deletion at End



Algorithm:

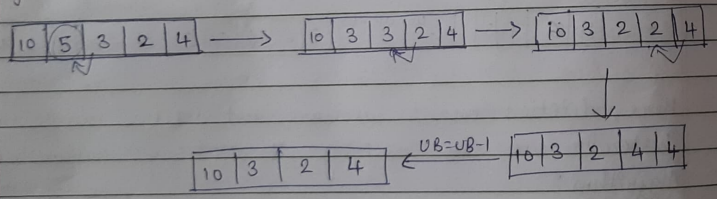
Step 1: Start
Step 2: let A be the array of size max.
Step 3: if UB=0 then print underflow and exit
Step 4: a[UB] = NULL
Step 5: UB = UB-1
Step 6: Stop

Deletion at specific position



a[0] a[1] a[2] a[3] a[4]

If we want to delete the second element.,



Algorithm:

Step 1: Start
Step 2: let A be the array of size max.
Step 3: if UB=0 then print underflow and exit.
Step 4: Read element.
Step 5: i=LB.
Step 6: Repeat step-7
while (i != pos)
Step 7: i = i+1
step 8: Repeat step-9 until
while (i<UB)
Step 9: a[i] = a[i+1]
i = i+1
Step 10: a[UB] = NULL
Step 11: UB = UB-1
Step 12: Stop.

Insertion at beginning, end and specific position

Algorithm:

Step 1: Insert (value, pos)
Step 2: Set i = UB+1
Step 3: Repeat step-4 to while (i>pos)
Step 4: A[i] = A[i-1]
Step 5: Set i = i-1
[End of loop]
Step 6: A[pos] = value.
Step 7: So, UB = UB+1
Step 8: Exit.

Searching an element in an array:
There are two types of searching —
- linear search
- Binary search

- Linear search :-

Algorithm:

Step 1: Start
Step 2: Declare array
Step 3: Read what to search (Let it be x)
Step 4: i==0
Step 5: Repeat step-6 while (i≤n)
Step 6: if x = A[i] i++
return success

else
return failure

Step 7: Stop

5/05/2022

- Binary Search :-

index no:	0	1	2	3	4	5	6	7
	10	15	20	30	35	40	56	70
	Low		mid-1	mid	mid+1			High

element to be searched, x = 35

L = 0

H = n-1

$mid = \frac{(L+H)}{2} = \frac{0+7}{2} = 3$

if x == a[mid]
Success
else if x > a[mid]
search on right side of array.
else
search on left side of array.

Algorithm:

Step 1: Start
Step 2: let A be the array
Step 3: Read element to be searched, say x
Step 4: L = 0, H = n-1
Step 5: Repeat step-6 until while (L < H)
mid = (L+H)/2
if x == a[mid]
return success;
else if x > a[mid]
L = mid + 1
else
H = mid - 1

Step 7: if not found return failure.
Step 8: Stop.

5/05/2022

Stack

- Stack is an important data structure which stores its elements in an ordered manner.
- A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.
- Hence, a stack is called a LIFO (Last-in-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Operations On A Stack

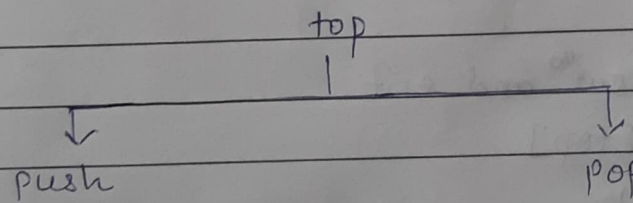
A stack support three basic operations:

- push - pop - peek

→ The push operation adds an element to the top of the stack and the pop operation removes the element from the top of the stack.

→ The peek operation returns the value of the topmost element of the stack.

Stack can be implemented using arrays and linked lists.



[will be done on top of stack]

PUSH

→ The push operation is used to insert an element into the stack.

→ The new element is added at the topmost position of the stack.

→ However, before inserting the value, we must first check if TOP=MAX-1, because if that is the case, then the stack is full and

no more insertions can be done.

Algorithm:

Step 1: Start
Step 2: Let A be the stack of size max.
Step 3: Read element to insert.
Step 4: If Top = Max - 1
Print overflow and exit.
Step 5: Set top = top + 1
Step 6: Set stack[top] = element.
Step 7: Stop

Pop

→ The pop operation is used to delete the topmost element from the stack.

→ However, before deleting the value, we must first check if Top = Null because if that is the case, then it means the stack is empty and no more deletions can be done.

Algorithm:

Step 1: Start
Step 2: Let A be the stack of size max.
Step 3: if top = null.
print "underflow" and exit.
Step 4: Set val = stack[top]
Step 5: Set top = top - 1
Step 6: Stop

Peek

→ Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.

→ However, the Peek operation first checks if the stack is empty, i.e., if top = null, then an appropriate message is printed, else the value is returned.

Algorithm:

Step 1: Start
Step 2: Let A be the stack of size max.
Step 3: ~~Read element to insert.~~
Step 3: if top = null
print "stack is empty" and exit.
Step 4: return stack[top]
Step 5: Stop

9/05/2022

→ Types of Expression

Infix, postfix and prefix notations are three different but equivalent notations of writing algebraic expressions.

infix → a+b
postfix → ab+
prefix → +ab

Conversion of an Infix Expression into a Postfix Expression

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-^fgh*+-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^*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/g/*h*+i
ab++cd^e/f/g/*h*+i
ab+cd^e/f/g/*h*++i
ab+cd^e/f/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*%GH*)
A-BC/DEF*%GH**+
ABC/DEF*%GH**+-

Conversion of an infix expression into a ~~postfix~~ prefix expression

1. a+b-*cd
+ab-*cd
-+ab*cd

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*^E^FG
A+B-C**D^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*gh-i

5. 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*//efg*h+i
a+b++^cd//efg*h+i
a+b+**^cd//efgh+i
+ab+**^cd//efgh+i
++ab**^cd//efgh+i
+++ab**^cd//efgh+i

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*EFGH)
A-+/BC*/%D*EFGH
+A+/BC*/%D*EFGH

→ Priority: This represents the evaluation of expression starts from "what operator"

→ Associativity: It represents which operator should be evaluated first if an expression is containing more than one operator with same priority.

OPERATORS	ASSOCIATIVITY
() [] . ++ (postfix) -- (postfix)	L to R
++ (prefix) -- (prefix) !~ sizeof (type) + (unary) - (unary) & (address) * (indirection)	R to L
* / %	L to R
+ -	L to R
<< >>	L to R
< <= > >=	L to R
== !=	L to R
&	L to R
^	R to L
|	L to R
&&	L to R
||	L to R
?:	R to L
= += -= *= /= %= >>= <<= &= ^= |=	R to L
, (comma operator)	L to R

1/05/2022

* Conversion of an infix express to postfix expression using stack

Algorithm:-

i) If stack is empty or it contains left parenthesis on top, push the incoming operator on to the stack.

ii) If the incoming symbol is opening bracket, push into the stack.

iii) If incoming symbol is closing bracket pop the stack and print the operators until left paranthesis is found.

iv) If the incoming symbol has higher precedence than top of the stack, push it into the stack.

v) If the incoming symbol has a lower precedence than top of stack then print the top.

vi) If the incoming symbol has equal precedence with top of the stack use the associativity.

vii) 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
		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
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*+

14/05/2022

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

Convert infix to postfix using stack.

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+-

12/05/2022

Queue

- A queue is a linear data structure.
- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the rear and removed from the other end called the front.
- Like stacks, queue can be implemented by using either arrays or linked lists.

Operations On Queues

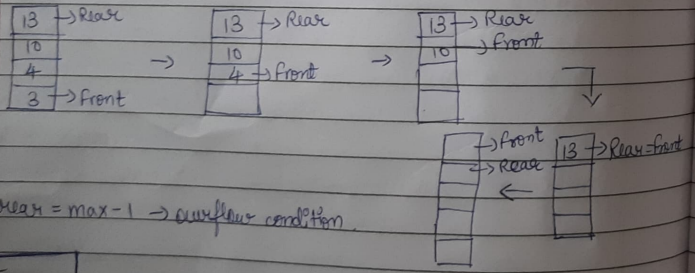
A queue support two basic operations:-

- Enqueue - Deque.

→ Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

If front = -1 } then queue is empty
rear = -1 } i.e. underflow condition.

front > rear → underflow condition.



rear = max-1 → overflow condition.

Enqueue

Algorithm:

Step 1:- Start
Let A be the queue of size max
Step 2:- If rear = max-1 print "overflow" and exit.
Step 3:- Read element (x)
Step 4:- If front = -1 and rear = -1
set front = rear = 0
else
rear = rear + 1
Step 5:- queue [rear] = x
Step 6:- Stop.

Deque

Algorithm:

Step 1:- Start
Step 2:- Let A be the queue of size n.
Step 3:- If front = -1 or front > rear
print underflow and exit.
Step 4:- value = queue [front]
Step 5:- front = front + 1
Step 6:- Stop.

→ 4 Types of queue:-

- Linear queue
- circular queue
- Priority queue
- Multiple queue.

14/05/2022

Circular Queue

→ In linear queues, insertions can be done only at one end called the rear and deletions are always done from the other end called the front.

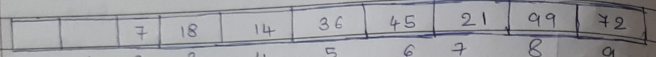
54	9	7	18	14	36	45	21	89	72
0	1	2	3	4	5	6	7	8	9

Now, if we want to insert another value, it will not be

possible because the queue is completely full. There is no empty space where the value can be inserted.

→ Consider a scenario in which two successive deletions are made.

The queue will be —



Here, front = 2 and rear = 9

Suppose we want to insert a new element.

→ Even though there is space available, the overflow condition still exists because the condition rear = max-1 still holds true. This is a major drawback of a linear queue.

Enque :-

For insertion, we now have to check for the following three conditions :

- If front = 0 and rear = max-1, the circular queue is full.
- If rear != max-1, then rear will be incremented and the value will be inserted.
- If front != 0 and rear = max-1, then it means that the queue is not full. So, set rear = 0 and insert the new element.

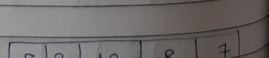
front = 0
rear = max-1
} → the queue is overflow.

Algorithm :

Step 1 : Start
Let A be the queue of size max.

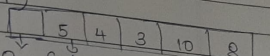
Step 2 : If front = 0 && rear = max-1
print "overflow" and exit

Step 3 : Read element.



Step 4 : If front = -1 && rear = -1
set front = rear = 0

Step 5 : else if (front != 0 && Rear = max-1)
set rear = 0



Step 6 : else, rear = rear + 1

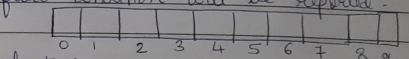
Step 7 : Q[Rear] = element.

Step 8 : Stop

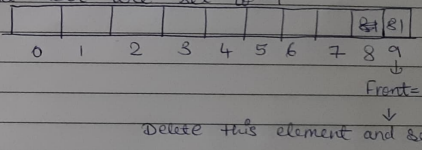
Deque :-

To delete an element, again we check for three conditions :

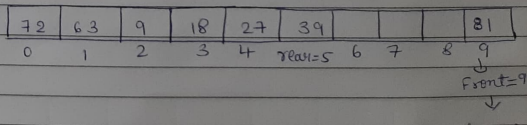
- If front = -1, then there are no elements in the queue. So, an underflow condition will be reported.



- If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to -1.



- If the queue is not empty, and front = max-1, then after deleting the element at the front, front is set to 0.



Deque

Algorithm:-

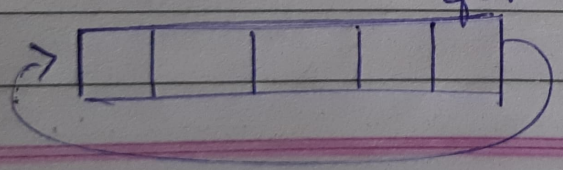
Step 1:- start.

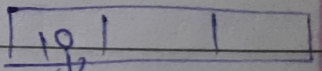
Step 2:- let A be the queue of size max.

Step 3:- if front = -1

print "underflow" and exit.

~~Step 4:~~ element = q [front]





- Queue is empty
- initial

// queue is having a single element, switch queue then queue to initial state = -1

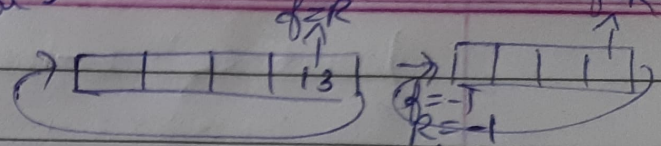
if front = rear

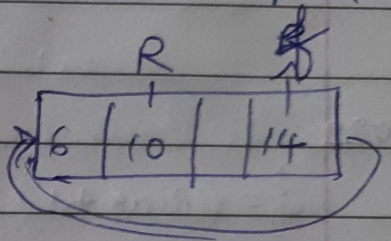
front = rear = -1

else if front = max - 1

Set front = 0

else front = front + 1



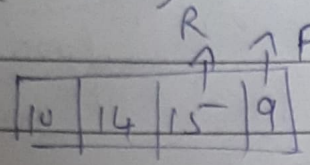


front = max - 1

front = 0.

16/05/2022

f = (Rear + 1) mod max size → overflow



if front = (Rear + 1) mod max size

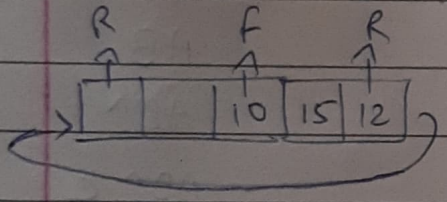
or

front = 0 and Rear = max - 1

print "overflow" and exit.

if front = Rear = -1

set front = Rear = 0.



If front != 0 && Rear = max - 1

set rear = 0.

else if front != 0 && Rear = max - 1

set rear = 0.

else

rear = rear + 1.

* Stack — Expression Evaluation using Stack :-

Postfix → Reverse polished notation.

The postfix expression is used to evaluate expression.

Q. 934 * 8 + 4 / -

Scanning	stack	Operation
9	9	
3	93	
4	934	
*	9 12	
8	9 12 8	
+	9 20	
4	9 20 4	
/	9 5	
-	4	

18/05/2022

DS

Q. AB+cd * EFG ^ * -

A=2, B=3, C=1, d=2, E=2, F=2, G=2

2 3 + 1 2 * 2 2 2 ^ ^ * -

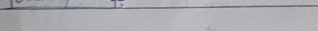
Scanning	Stack.
2	2
3	2 3
+	5
1	5 1
2	5 1 2
*	5 2
2	5 2 2
2	5 2 2 2
^	5 2 4
^	5 16
*	80
-	27

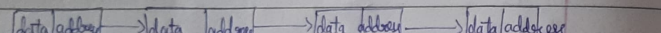
18/05/2022

Linked list

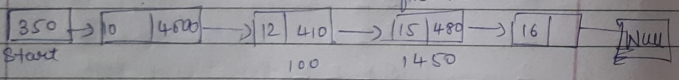
Limitations of Array:
- Sequential access
- Size specify
- Size

Linked list is the collection of nodes.
-> Every node in linked list has two elements: data and address.





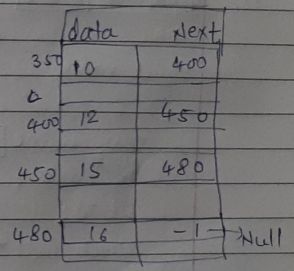
Start node will store the base address i.e. address of first node



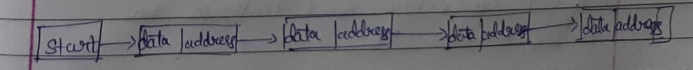
- The array is the linear collection of data elements in which the elements are stored in consecutive memory locations.
- While declaring arrays, we have to specify the size of the arrays which will restrict the number of elements that the array can store.
- To make efficient use of memory the elements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

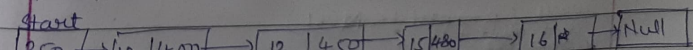
So, linked list is a data structure that remove this restriction.

* Limitation of linked list:
A linked list doesnot allow random access of data elements.
The elements in this linked list can be only be accessed in sequential manner.



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





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

*link

EX:- | name | marks | id | next |

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

→ Declare a pointer, start = Null to indicate the linked list is initially empty

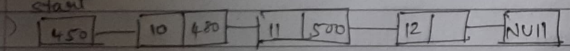
```
start = Null.
link list is empty can't traverse
count=0
p= start
while (p->next != NULL)
count= count + 1
p=p->next.
```

19/05/2022 Data Structures

Linked List

* Traversing of linked list :-

Step1 : start
Step2 : if (start = NULL)
print "link is empty" and exit.
Step3 : p = start
Step4 : while (p -> link != NULL) or while (p != Null)
Display the data part.
p = p -> next link.
Step5 : Stop

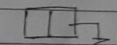


* Counting number of elements in linked list.
Step1 : Start
Step2 : if (start = NULL)
print "link is empty" and exit
Step3 : count = 0
p = start
Step4 : while (p -> link != Null)
Count the no. of elements in linked list.
count = count + 1.
p = p -> link
Display count.
Step5 : Stop

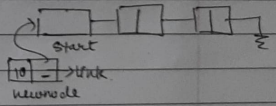
* Insertion in linked list :
→ Insert node at beginning OR create a node
→ if start = Null.
node => only node in linked list.
→ Node will be inserted at first position when we create node.

. Create node.
newnode = (struct node *) malloc (size of (struct node))
Get Data from user
newnode -> link = Null;
if (start = Null)
{
newnode -> link = NULL
start = newnode
}

node will be the only element
start = Null



else
{
newnode -> link = start
newnode = start
}



* Insertion at end
- Create new node.
newnode = (struct node *) malloc (size of (struct node))
Get Data from user.
newnode -> link = Null;
if (start = Null)
{
newnode -> link = Null
start = newnode
}

```
else
{
    p= start ;
    while (p -> link != Null)
    {
        p= p -> link
    }
    p -> link = newnode .
}
```

~~pos = 0.~~

* Insertion at specific position.