Arrangement of data in a organized way.

Types of data structures

Stack
Queue
tree
Graph

The requirement of the user decides the implementation of data structure.

20.06.2018
Wednesday

DSA

PERIOD: 5

```
for (i←0 to n-1 ; i++)
```

$i \leftarrow 0 \rightarrow 1$

$i > n-1 \Rightarrow 2(n+1)$

$i \leftarrow i+1 \Rightarrow 2(n)$

$1 + 2(n+1) + 2n = 4n+3$

POSSIBLE PRIMITIVE OPERATIONS

1. Assign a value to a variable
2. Method invocation (calling a function or subroutine)
3. Perfosming a simple arithmetic operation.
4. Indexing into an array
5. Following an object reference.
6. Returning foom a method.

Algorithm innerproduct

INPUT : Non-negative integer n and two integer arrays A and B of size n.

OUTPUT : The innerproduct of two arrays

```
Prod ← 0
for i = 0 to i < n-1 do
        Prod ← Prod + A[i] * B[i]
return Prod
```

$= 1+1+2(n+1)+5n+1+2n$

$= 3+7n+2+2n$

$= 5+7n+2n$

$= 9n+5$

22.06.18
Friday

DSA

5th hour.

Q. Algorithm FindMax (S, n)

Input : An array S storing n numbers, $n>=1$

output : Max Element in S.

```
curMax ← S[0]
i ← 0
while i < n-1 do
  if (curMax < A[i]) then
    curMax ← A[i];
  i ← i+1;
return curMax
```

2
1
2n
2(n-1)
2(n-1)
2(n-1)
1

$8n+3-2-2-2+1$

$8n+4-6$

$8n-2$

Best case:-

The Minimum number of operations done by the algorithm.

Average case:-

23.06.18 DSA 5th hour.

Saturday

1. Sum = 0; — ①
   for(i=0; i<n; i++) — (n+1) + 2n + 1
   Sum++; — 2(n)

   1 + 3n + 2 + 2n = 5n + 3

2. Sum = 0;
   for(i=0; i<n; i++) — 1 + 2n + (n+1) + 2n
   for(j=0; j<n; j++) — 1 + (n+1) + 2n
   Sum++; — 2n

   = 3n + 4 + n + 1 + 2n

   = 6n + 5

• 1 + n + (n+1)(n+1)

• 1 + (n+1) + 2n

= 1 + 1 + (n+1) + n + n(n+1) + n(2n) + n(2n) + 2n

= 2 + n + 1 + n + n² + n + 2n² + 2n² + 2n

= 5n² + 5n + 3

3. `for (i=0; i<n; i*=2)`

i=0 → i+∞

i=0 → 1

`while (i<n)` → ∞

{

`i = i*2;`

}

i = i*2

= 0

i=1 → 1

`while (i<n)` → n

{

`i*=2;` → 2(n-1)

}

i = i*2

2(n-1)

i = i*2

i = 2*2

i = 4

i = 4*2 = 8

i = 8*2 = 16

i = 16*2 = 32

i = 32*2 = 64

1 + n + 2(n-1) = 3n - 1

i = 1 → 1

`while (i<n)` → n

{

`i = i*2;` → $2n^2$

}

~~$2n^2+n+1$~~

1

2

4 } N

8 }

16 } N

32 }

N = 4

N, $\frac{N}{2}$, $\frac{N}{4}$, $\frac{N}{8}$

[2, 4) [4, 8) [8, 16)

2, 3 4, 5, 6, 7 . . .

Q. Sum = 0

2^4 17

1

2

3

16

$(\log_2 4)+2$

1 2 4 8 16 32

$\log_2(17) = 4 + 1 = 5$

Q. Sum → 0; →1

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
      sum++;
```

① $j < n^2$

② $j < i$

③ Sum = Sum + cal(i) ④ instead of i++, i=i*2

where, cal(n)

```
{
   s=0;
  for (i=0; i<n; i++)
     s++;
     return s;
}
```

Q.

```
Sum →0;

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
      sum++;
```

```
Sum = 0    — 1
i=0        — 1
while (i<n)  -(n+1)
{
  j=0          → n
  while (j<n²) → n(n²+1)
   {
    sum = sum+1;  n(2n²)
    j = j+1;      n(2n²)
   }
  i = i+1;   2n
}
```

$= 1+1+n+1+n+n^3+n+2n^3 + 2n^3+2n$

$= 3+2n+n^3+n+2n^3+2n^3+2n$

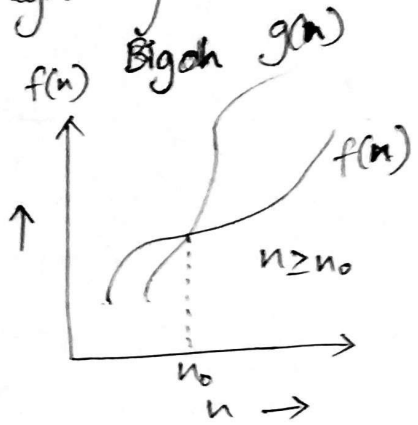
$= 5n^3+5n+3.$

```
sum=0;
for(i=0; i<n; i++)
{ for(j=0; j<n; j++)
  {
    sum++;
  }
}
```

```
Sum=0          — 1
i=0            — 1
while (i<n)    — (n+1)
{
  j=0          — (n)
  while (j<i)
  {
    sum = sum+1;
    j=j+1;
  }
  i=i+1;
}
```

Big O
Theta
Omega.
} These are the three notations.

Bigoh

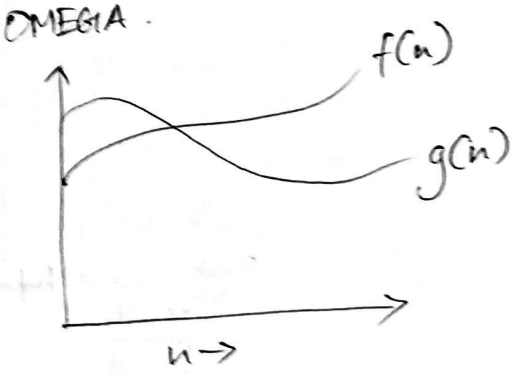


$f(n) \leq c\, g(n)$

$f(n) = O(g(n))$ → bigOh notation

$f(n) \in \Theta(g(n))$

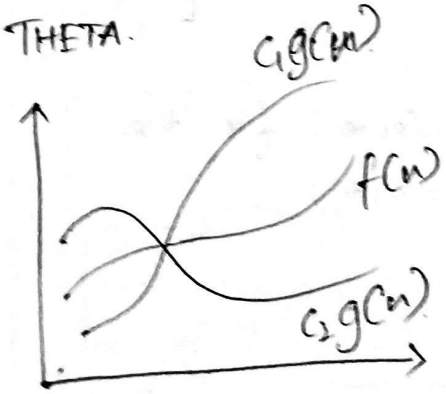
OMEGA.



$f(n) \geq g(n)$

$f(n)$ is $\Omega\, g(n)$

THETA.



$c_2 g(n) \leq f(n) \leq c_1 g(n)$

Q. $8n+4 < 9n$.

~~$8(n)+4 < 9$~~

when $n = n_0$

$8n_0+4 < 9n_0$

~~$n_0 \to 4$~~ $n_0 = 4$

$c = 9$

Q. $5n^4+3n^3+2n^2+4n+1$

$5n^4+3n^4+2n^4+4n^4+n^4$

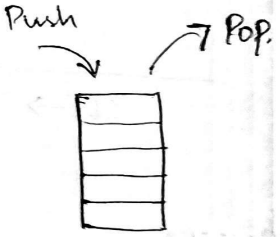
$15n^4$

$c = 15 \quad 0$

ABSTRACT DATA TYPE:-

STACK:

Last In First Out (LIFO)



Push(e):

Adds element e to the top of the stack

Pop:

Removes and returns the top element from the stack (or null if the stack is empty)

Accessor methods

top():

Returns the top element of the stack, without removing it (or null if the stack is empty)

size():

Returns the number of elements in the stack.

isEmpty( ):
Returns a boolean indicating whether the stack is empty.

Method	Return value	Stack contents
		(5)
Push(5)	–	(5,3)
Push(3)	–	(5,3)
size( )	2	(5)
pop( )	3	(5)
isEmpty( )	False	( )
pop( )	5	( )
isEmpty( )	True	( )
pop( )	10	(7)
Push(7)	–	(7)
top( )	7	(7,4)
Push(4)	–	(7,4)
size( )	2	(7)
pop( )	4	(7,6)
Push(6)	–	(7,6,8)
Push(8)	–	(7,6)
pop( )	8	

```
Public interface Stack<E>
{
   int size();
   boolean isEmpty();
   E top();
   Void Push(E element);
   E pop();
}
```

Stack ADT operations.

1. Implementing stack using array.

i) Push (e)

~~Push(e)~~ Initializing Stack variable ~~S~~ Init

~~top ← top+1~~

~~S[top] ← e~~

- Push (e)

{

top ← top+1

S[top] ← e

}

- Push (e)

{

if size() < N then

{

top → top+1

S[top] → e

}

}

→ O(1)

Because, $f(n) \leq cg(n)$

$5 \leq 5*1$

ii) Pop().

e → S[top]

top → top-1

return e

- Algorithm pop()

if not (IsEmpty()) then → O(1)

e → S[top]

top → top-1

return e

else

throw StackEmptyException

- Algorithm size()
  return top+1 → O(1)
- Algorithm isEmpty() → O(1)

```
if (top<0) then
  return true
else
  return false.
```

- Algorithm top() → O(1)

```
if isEmpty() then
   throw StackEmptyException
else
  return S[top]
```

- Time complexities of the above functions are different for implementation using array and linked lists.
- Space complexity O(N) => Independent of stack size N

  N - size of the array.

Demerits of stack

→ Difficulty in specifying size of the stack.
→ Smaller size may cause earlier stack full exception throw.
→ Larger size may cause memory wastage

Object is the top class for everything.

for Generic data types for specifying the memory for arrays

```
CAPACITY = getId.getCapacity();
data = (E[]) (new Object[CAPACITY]);
```

- Static keyword is used in main method and also at few methods. The use of the static keyword is that all the objects are ~~common~~ common.

A 04.07.18 DSA 5th hour

Algorithm for Palindrome checking.

```
Scanner obj = new Scanner(System.in);
char a[], b[];
    a = new char [10];
    a = obj.nextLine();
    b = new char [10];
```

~~void Push(char e)~~ ~~int n = sto a.length;~~ int t = -1;

~~{~~

~~void POP()~~ ~~for (int i=0; i<n; i++)~~

~~{~~ ~~{~~

~~temp int temp =~~ ~~POP(i);~~

~~char temp = a[~~

~~void pop(int i).~~

~~{~~

~~temp = a[i];~~

~~a[i] = null;~~

```
push (char e)
{
    if( size() < CAPACITY)
      {
          t=t+1
          s[t]= e;
      }
}
```

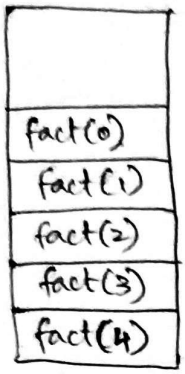
```
POP( )
{
  if(t<0)
  { throw Empty stack exception
  }
else {   s[t]= null;
         t=t-1;
     return s[t];
}
```

```
for (i=0; i<n; i++)
  {
      obj. push(a[i]);
  }
for (i=0; i<n ; i++)
    {
       pop(); b[i]= pop();
    }
for(i=0; i<n; i++)
   {  if(a[i] != b[i])
      {
        Print not a palindrome
      }
   }
```

RECURSION

Eg. Factorial.

$$n! = \begin{cases} n & \text{if } n=0 \quad \text{Base case} \\ n-(n-1)! & \text{if } n \geq 1 \quad \text{Recursive case} \end{cases}$$



Stact for temporary storage

Factorial(4)
↳ factorial(3)
↳ factorial(2)
↳ factorial(1)
↳ factorial(0)

Types of recursion.

1. Linear Recursion
2. Binary Recursion
3. Multiple Recursion

1.

Recursive method is designed so that each invocation of the body makes new recursive calls.

2.

Each invocation of the body makes two new recursive calls.

```
public static int binarySum(int[] data, int n)
{
    if (low > high)
        return 0;
    else if (low == high)
        return data[low];
    else
```

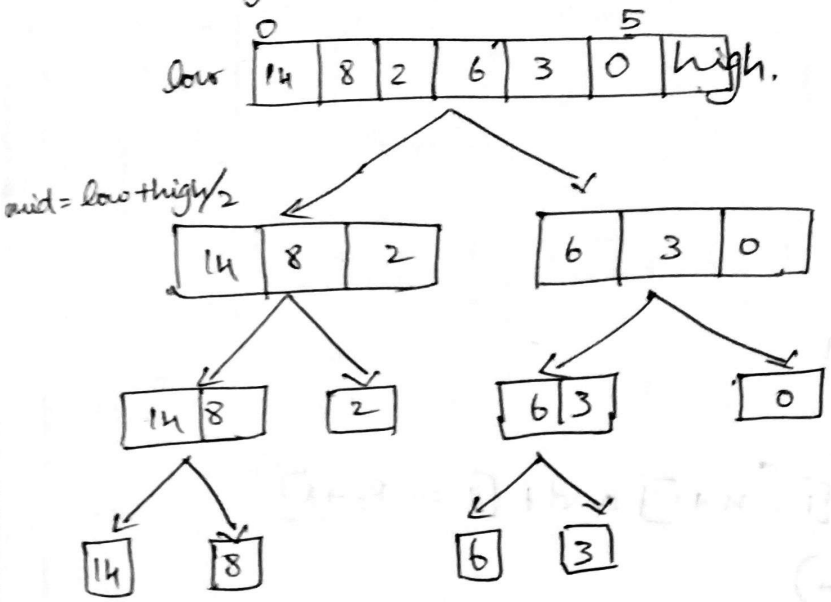
```
int mid = (low + high)/2;
return binarysum(data, low, mid) +
                binarysum(data, mid+1, high);
}
```

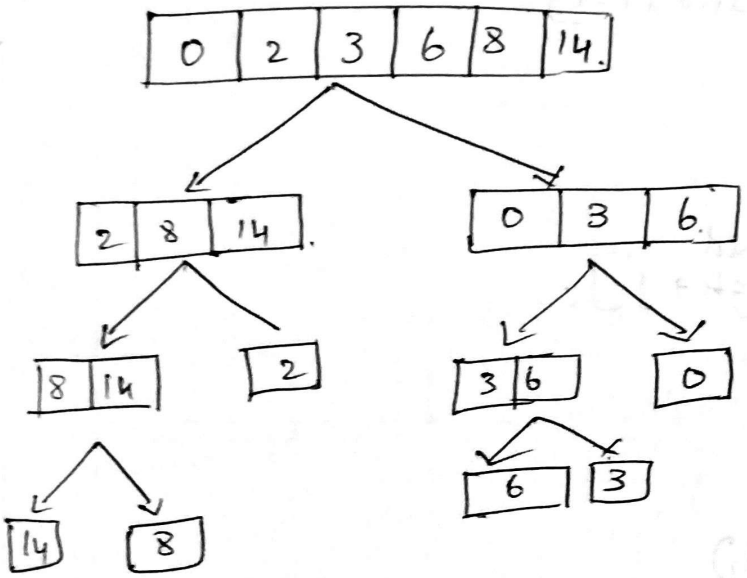
06.07.18.
Friday.

DSA

5th hour

Merge sort (Divide and conquer)





$\frac{n}{2^0}$

$\frac{n}{2^1}$

$\frac{n}{2^4}$

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log n = k$

```
int l1, l2, h1, h2, i, j.
merge sort (low, high)
{
  if (low == high)
  return
  else
     mid = low + high/2
     merge sort (low, mid);
     merge sort (mid++, high);
     merge (l, m, h);
}
```

```
merge (l, m, h)
n1 = high - mid + 1
n2 -> low - mid.
create arrays L [1 ... n1 + 1] and R [1 ... R2 + 1]
for (i = 0 ; i < n1 ; i++)
   {
      L[i] ← A [low + i - 1];
   }
for (j = 1 ; j < n2 ; j++)
   {
      R[j] ← A[mid + j];
   }
L [n1 + 1] ← ∞
R [n2 + 1] ← ∞
for (k ← low to high)
   do if L[i] ≤ R[j]
      {
         A[k] ← L[i]
         i ← i + 1;
      }
```

else A[k] ← B[j]

j ← j+1

09.04.17.

Growing Stack

Stack's size can be increased during the creation of the stack.

data[] = (E[]) new Object (E[]),

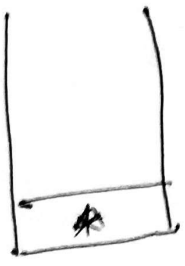
type []

data = type.

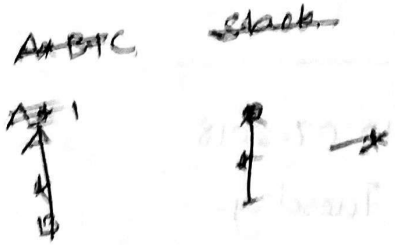
∴ The stack [data] can refer to the stack type.

INFIX To POSTFIX.

① A * B + C



A

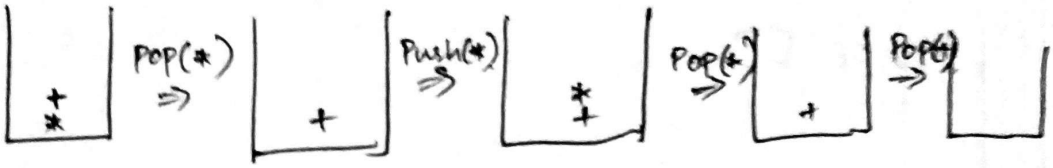


The lower precedence are taken to the maintained in the ~~state~~ stack.

① A + B * C.

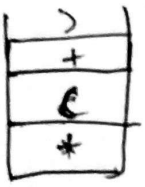
ABC* +

② A * B + C * d.



(AB* cd* +)

③ A * (B + C)

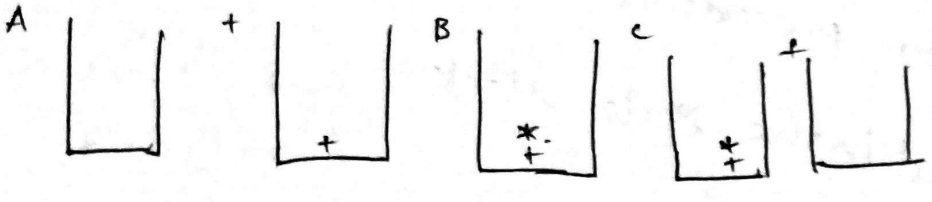


Postfix expression will not have any parenthesis

ABC+ *

10.07.2018.
Tuesday. DSA 3rd hour.

① A + B * C + D.



A ABC* + D +

② Application of binary recursion.

By invoking ~~two~~ the function two times the process will be reduced by 2^k times, where

k is any integer.

→ Merge sort, Fibbonacci series.

③ What is Activation record?

If a function is called, then the record is stored in the stack. It maintains the record of recursive functions in order.

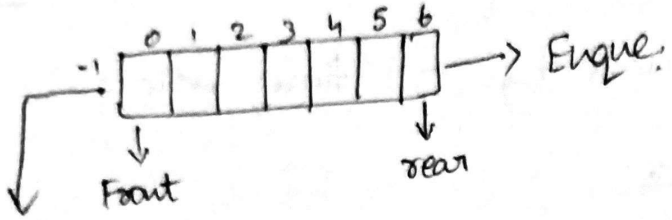
④ Stack size = 15.

Find the middle element.

⑤ Is it possible to maintain 2 stacks in a same array.

Yes, By using two pointers for a the same array.

QUEUE :-



Initially, Front and rear will be at -1

→ While inserting 1st element, both Front and rear are incremented.

→ For inserting 2nd element, only rear is incremented.

① Both front and rear = -1.

enque

```
if(size()>N)
{
   System.out.println("Queue is full");
}
else
{ if(front==-1)
   {
      front++;
      front=e;
   }
   else{
   rear++;
   rear=e;
   front=e;}
}
```

enque.

```
if(size()<N)
  rear = rear+1
  s[rear]=e
```

deque

```
if( !IsEmpty())
{
    temp = S[front];
    front = front + 1;
    S[front] = null;
    return temp;
}
```

```
IsEmpty()
{ if(size() == 0) return true; }
```

QUEUE.

```
front = 0
rear = N-1
size = 0

enque(e)
{
  if (size() < N)
  {
    rear = (rear+1) mod N;
    data[rear] = e;
    size++;
  }
  else {
    S.O.P ("Queue full ex");
  }
}
```

```
deque()
{
  if (size > 0)
  {
    temp = data[front];
    data[front] = null;
    front = (front+1) mod N;
    size--;
    return temp.;
  }
  else {
    S.O.P ("Queue Empty Ex");
    return null;
  }
}
```

18/07/18. DSA

1. Give a pseudocode for an array based implementation of double ended Queue ADT.

What is the running time for each operation?

4 functions in double ended Queue. they are

i) Insert first
ii) Insert last
iii) Remove first
iv) Remove last

2. Reverse the order ~~the~~ of elements in a stack using one additional Queue.

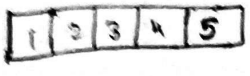
3. Implement stack ADT using two queues. What is the running time of push() and pop() in this case.

4. ~~Ret~~ Sorting a queue without extra space

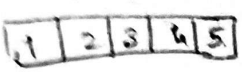
5. Reverse a queue using recursion.

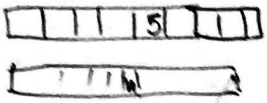
DOUBLE ENDED QUEUE.

1. IsEmpty, Size() - access functions
2. Insert First(e), Insert last(e), remove first(), remove last(). - update functions

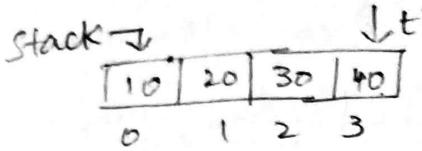


dequeue(1)

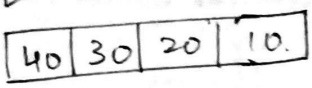




2. Reverse stack using queue



Required output

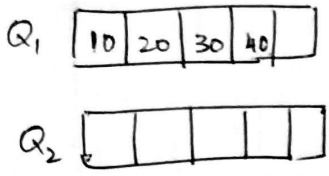


Popping the elements from the stack and enqueing them into a queue.

```
{
    e = S.pop().
    Q.enque(e)
}
```

for n elements, the time complexity will be O(n)

3. stack using two queues



Q_1.dequeue for n-1 times and enqueing in Q_2.

∴ nth element in Q_1 can be pointed out or can be accessed easily.

Queue

New(): ADT - Creates an emptyqueue

enqueue(): ~~ADT~~ ADT - Inserts element e at the rear of the queue

Dequeue(): ADT - Removes an element at the front of the queue, an error occurs if the queue is empty.

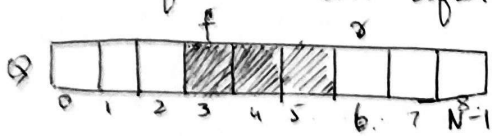
Front(): element - Returns, but does not remove the front element an error occurs if queue is empty

Size(S: ADT): integer

IsEmpty(S: ADT): boolean

QUEUE - ARRAY IMPLEMENTATION.

- Create a queue using an array in a circular fashion
- Max size N is specified
- The queue consists of an N-element array Q and two integer variables.
  - ↳ Index of front element (head for dequeue)
  - ↳ Index of the element after the rear one (tail for enqueue)



3-(4+1) 8-4-1

9

what does f=r mean?

It can either be full or empty.

```
Algorithm enque(e)
if size()=N-1 then
return QueuefullException
Q[r] ← e
r ← (r+1) mod N
```

```
Algorithm sizec)
return (N-f+r) mod N

Algorithm isEmpty()
return(f=r)
```

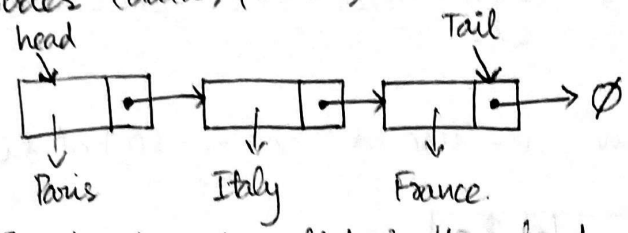
```
Algorithm front()
if isEmpty() then
return Queueemptyexception
return Q[f]
```

```
Algorithm dequeue()
if isEmpty() then
return Queue emptyexception
Q[f] — null
f ← (f+1) mod N
```

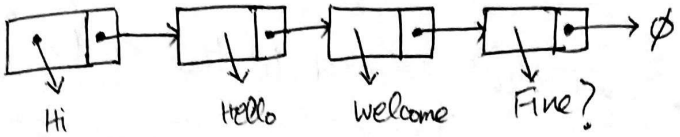
Disadvantage of array implementation is we can use with a fixed size.

Implementing Queue with linked lists

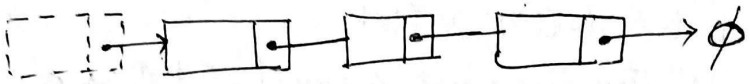
- Nodes (data, pointer) connected in a chain by links



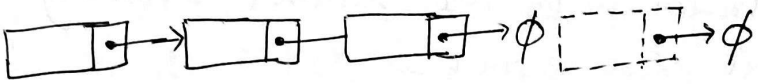
- The head of the list is the front of the queue, the tail of the list is the rear of the queue.



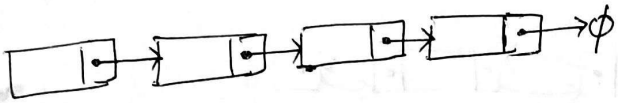
- Dequeue - advance head reference.



- Enqueue - Create a new node at the tail



Chain it and move the tail reference.



DOUBLE ENDED QUEUE :-

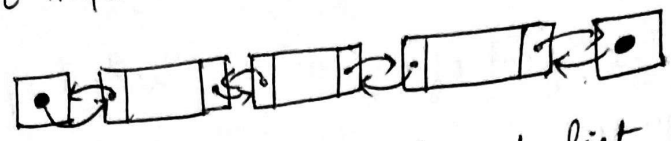
~~Deque~~ It supports insertion and deletion from the front and back.

It supports six fundamental methods

- InsertFirst (S:ADT, O:element): ADT - Inserts e at the ~~beggi~~ beginning of deque.
- InsertLast (S:ADT, O:element): ADT - Inserts e at the end of deque.
- RemoveFirst (S:ADT):ADT - Removes the first element
- RemoveLast (S:ADT):ADT - Removes the last element.

DOUBLY LINKED LISTS

- Deletions at the tail of a singly linked list cannot be done in constant time.
- To implement a DEqueue, we use a doubly linked list.



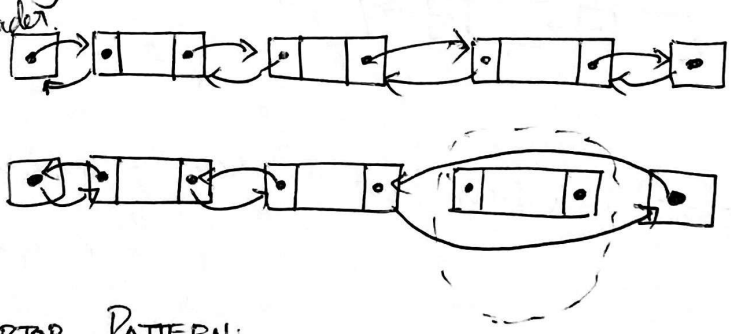
- A node of a doubly linked list has a next and a previous link.
- Two special nodes were added to the end of the lists.

→ Header node goes before the first list element. It has a valid next link but a null prev link

→ Trailer node goes after the last element. It has a valid prev reference but a null nesct reference.

Header and trailer nodes are sentinel or dummy. Bcoz they don't store elements.

Deleting the last element.



ADAPTOR PATTERN:

Using a double ended queue to implement a stack or queue is an example of the adaptor pattern. Adaptor pattern means implementing a class by using methods of other class.

SEQUENCES

* Vectors
* Positions
* Lists
* General sequences.

VECTOR ADT

A sequence S with n elements the supports the following methods...

→ There will be a rank for every element of sequence

elemAtRank(r): Return the element of S with rank r; an error occurs if $r<0$ or $r>n-1$

replaceAtRank(r,e): Replace the element at rank r with e and return the old element; an error condition occurs if $r<0$ or $r>n-1$

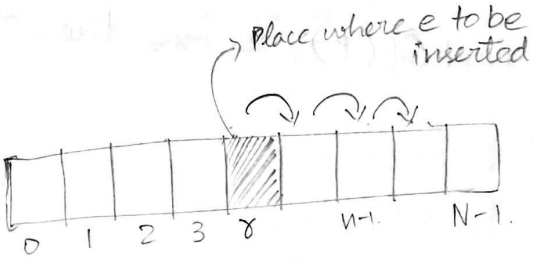
insertAtRank(r,e): Insert a new element into S which will have rank r; an error occurs if $r<0$ or $r>n$.

removeAtRank(r): Remove from S the element at rank r; an error occurs if $r<0$ or $r>n-1$

Array Based Implementation.

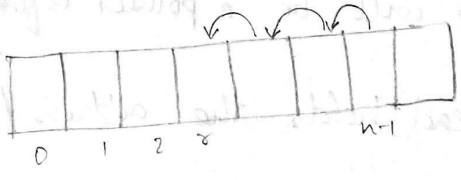
1. Algorithm insertAtRank(r,e):

```
for (i=n-1 ; i>=r; i--)
 {
    S[i+1] = S[i];
 }
  S[r] = e;
  n = n+1;
```



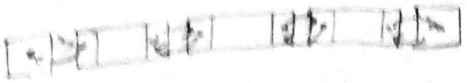
2. Algorithm removeAtRank(r);

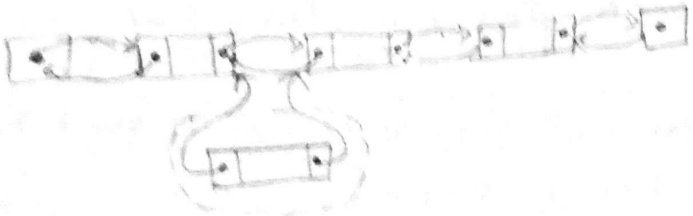
```
  e ← S[r]
  for (i=r ; i<n-1; i++)
    {
       S[i] = S[i+1];
    }
    n= n-1;
```



Implementing with a doubly Linked list

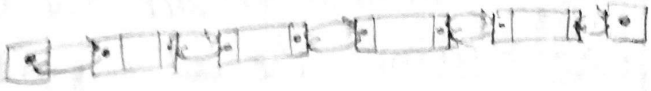
The list before insertion





creating a new node to insert

List after inserting new node

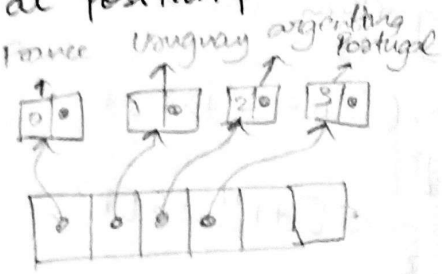


SEQUENCE ADT

- combines the vector and List ADT (multiple inheritance)
- Adds methods that bridge b/w ranks and positions

atRank(r): returns the position at rank r

RankOf(P): returns the rank at position P



LINKED LIST

There will be a pointer before the 1st node called as head.

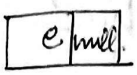
- Head holds the address / reference of the 1st node

The tail ~~node~~ pointer will be present at the last node

```
temp= head
while ( temp! = null)
{

    temp = temp. next ;
}
```

Q. There is an existing list of elements 10, 20, 30, 40.
Write the algorithm to insert 5 at the beginning

① creating a new node.




```
Insert First(e)
new = Node (e) // new is a variable
new . next = head
head = new
```

Q. Algorithm for inserting at last

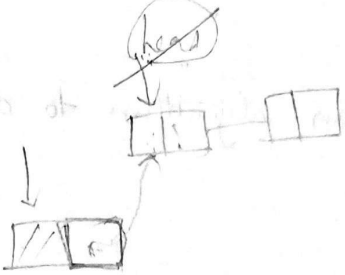
① creating a node.

```
Insert Last (e)
new = Node (e)
~~new . next = tail~~
Tail . next = new
tail = new
```

```
temp = head
while (temp.next != null)
{
    temp . next = new
}
tail = temp.
```

Q. Algorithm for remove First

```
Remove First
temp = head // data
head = head . next // copying to next
temp.next = null. // deleting the head pointer.
```



newnode.next = head

head = newnode

Complexity analysis and Stack. -TUTORIAL- I

Java code for linked list implementation.

temp = head.

Doubly Linked list

InsertFirst ()

```
new = Node (e)
new =
new.next = head
head.prev = new;
new.prev = null;
head = new
```

if head == null
{ head = new
tail = new
new.next = null
new.prev = null.
}

Insert Last ()

```
new = Node (e)
tail.next = new
new.next =
new.prev = tail
new.next = null
tail = new
```

Q write an algorithm to display the common elements in two lists

QUICK SORT

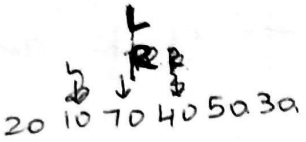
Partitioning and Sorting

low → 40 20 6 8 70 30 ← high

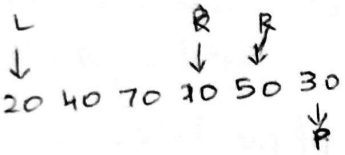
30 ↳ Pivot element

6 8 20 < 30 < 40 70

```
QuickSort (A , low, high)
{
    P = Partition (A , low, high)
    Quicksort (A, low, P-1)
    Quicksort (A, P+1, high)
}
```




```
Partition (A, low, high)
{
    l = low
    r = high-1
    P = A[high]
    while(l <r)
    {
        while (A[l] <P)
            l++ ;
        while (A[r] >P)
            r--;
        Swap (A[l], A[r])
    }
```



L	R
~~20~~	~~50~~
~~40~~	~~10~~
10	~~40~~
	70

Periodicals Portion

Introduction to data S

- Def, org of data, efficiency.

Complexity analysis

- Basic
- asymptotic
- Big-Oh

- ABStract Data type
- Stack.
  - Array Based Imp
  - Implementing recursion using stacks
- Linear, Binary, multiple recursions
- Merge sort, Quick sort
- Queue, Double ended queue
- circular queue, applications, array based implementation
- Singly, Doubly, circular linked lists
- Stack and queue implementation using linked lists

Stack - Functions and applications
Queue - Functions and applications
Double ended queue - Functions and applications

Implementation of Stack using arrays
Implementation of Stack using linked lists
Implementation of Queue using arrays
Implementation of Queue using linked lists

01/08/2018. TREES 3rd hour.

Trees - Hierarchial relationship between elements

Decision tree

Parse tree

Basic terms

* root
* leaf (external node)
* internal node.
* Parent
* children
* siblings
* ancestors
* height
* depth

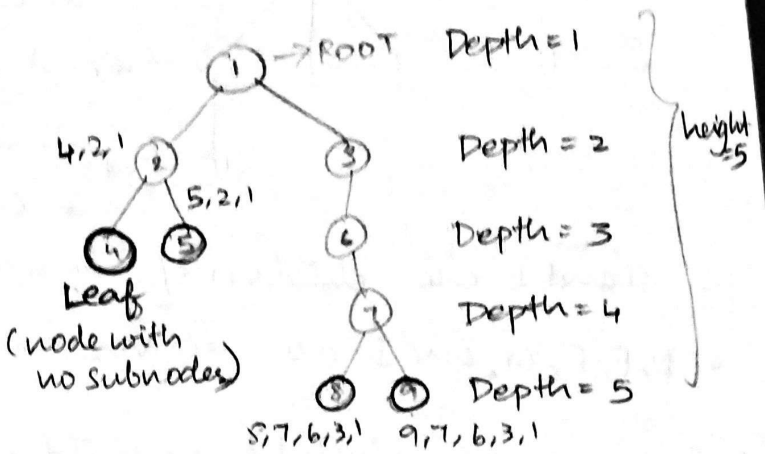


Fig T.1

A tree can have a single node or. It can also be empty.

Internal node:

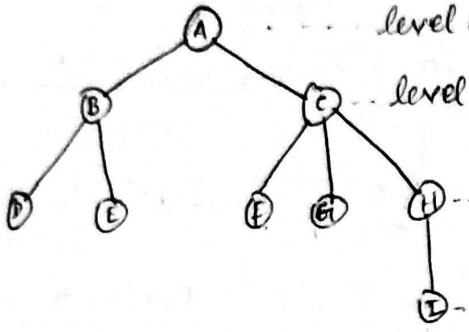
They have both parent and child.

Fig. T.1 2, 3, 6, 7 are called as internal nodes.

Trees:-

A set of nodes storing elements such that the nodes have a parent child relationship that satisfies the following conditions.

i) If T is nonempty, it has a special node, called the root of T, that has no parent.



- A is the root node
- B is the parent of D and E
- C is the parent of F, G and H
- A is ancestor of D and E
- D and E are descendants of A
- C is the sibling of B

- D and E are children of B.
- D, E, F, G, and I are leaves

A leaf is a node which has no children.

- A, B, C and H are called as internal nodes.
- Maximum height of the tree = Maximum level of the tree

Degree of a node:

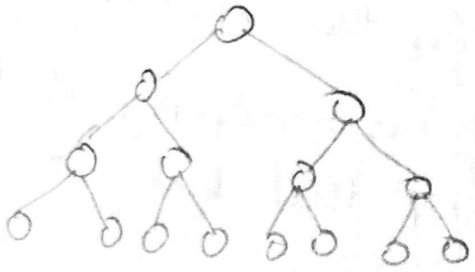
The no. of children the node has

Ordered tree: It is the one which the children of each node are ordered.

Binary tree: An ordered tree in which each node it has at most 2 children

Decision tree: Each node in the tree which corresponds to a decision.
It is an example of Binary tree.

A complete Binary tree



Level i has 2^i nodes.
In a tree of height h.

- Leaves are at height h
- NO of leaves is 2^h
- NO of internal nodes $= 1+2+2^2+\dots+2^{h-1} = 2^h - 1$
- " " " " = NO of leaves $- 1$
- Total no of nodes $= 2^{h+1} - 1 = n$

- In a tree of n nodes
  No. of leaves $= (n+1)/2$
  Height $= \log_2(\text{no. of leaves})$

Minimum height of a binary tree

- A binary tree of height h has
  - At most 2^i nodes at i^{th} level
  - At most $1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$ nodes
- If the tree has n nodes then
  - $n <= 2^{h+1} - 1$
  - Hence $h >= \log_2((n+1)/2)$

Max. height of a binary tree

- A binary tree on n nodes has height at most $n-1$
- This is obtained when every node has exactly one child.

Tree should be balanced to reduce the complexity.

Edge: An edge of tree T is a pair of nodes (u, v) such that u is parent of v, or vice versa.

Path: Path of T is a sequence of nodes such that any two nodes connected with.

- A tree is ordered if there is a meaningful linear order among the children of each node.
- Purposefully identify the children of a node as being the 1st, 2nd and 3rd & so on.
- order is usually visualized by arranging siblings left to right according to their order.

unordered trees

Examples: organizational chart for a company

Tree to describe an inheritance hierarchy

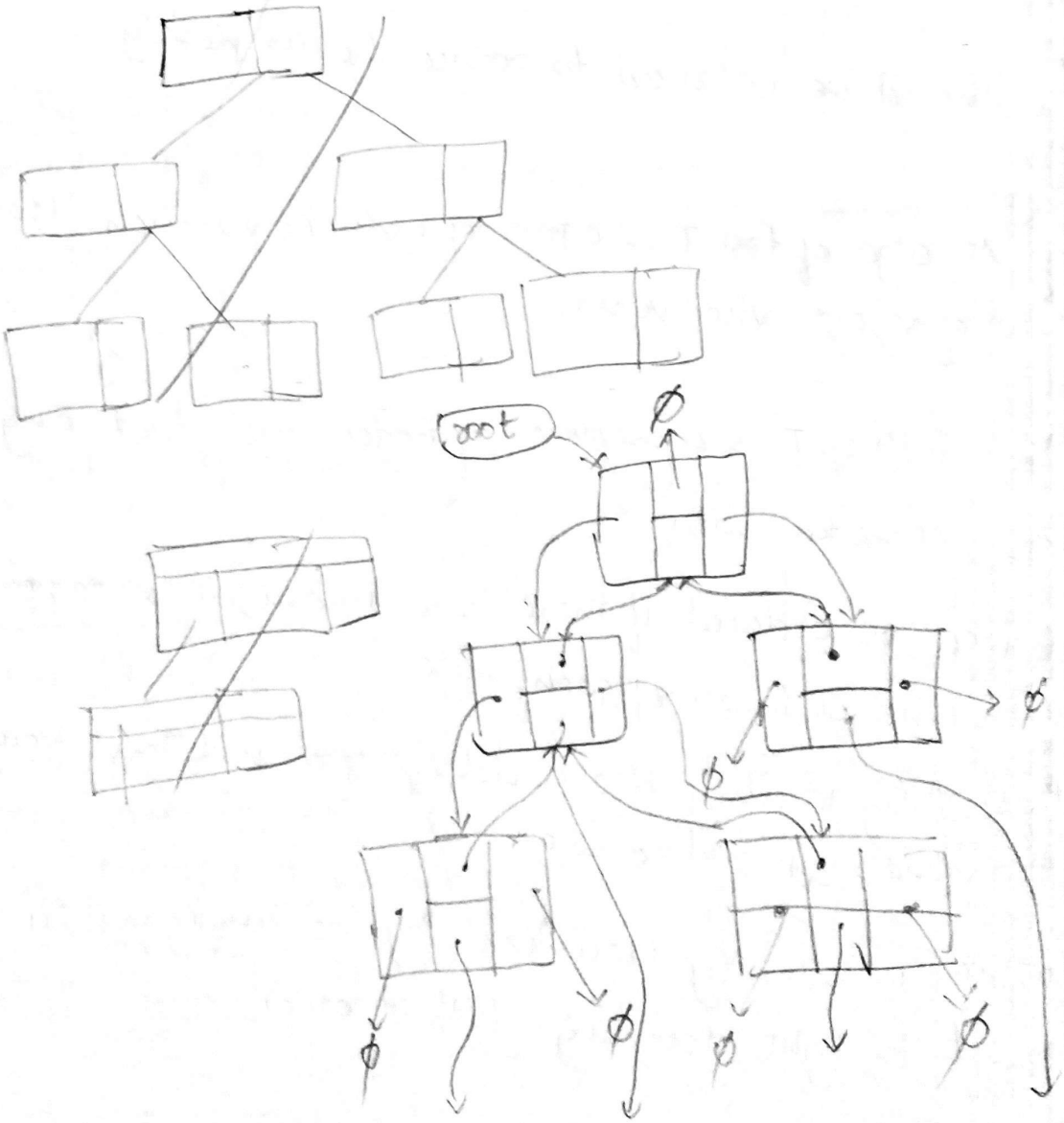
Tree in modelling a computer's file system

operations of tree:

getElement()
root()
parent(p)
children(p)
numChildren(p)

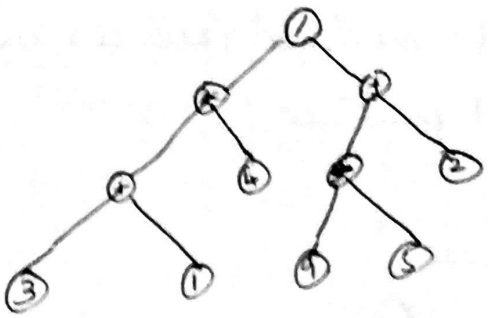
isInternal(p)
isExternal(p)
isRoot(p)
size()
isEmpty()
Iterator()
positions()

Binary tree can be implemented using array and linked list



Implementation of trees using arrays.

	Index
root	0.
Left child of root	2(0)+1
Right child of root	2(0)+2



For every position P of T, let f(P) be the integer defined as follows:

- If P is the root of T, then f(P) = 0
- If P is the left child of position q, then f(P) = 2f(q)+1
- If P is the right child of position q, then f(P) = 2f(q)+2

The numbering function f is known as a level numbering of the positions in a binary tree T.

- Space usage is O(N) where N = $2^n - 1$ in worstcase
- Running time of update method is O(n) in worst case.

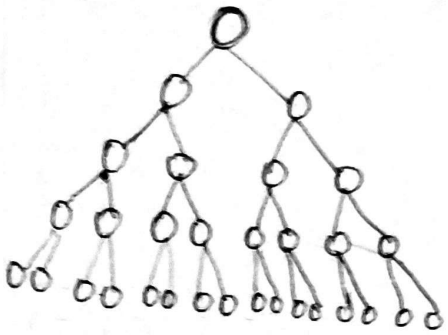
```
Class Node <E>
{
    Private E element;
    private Node<E> parent;
    Private Node <E> leftchild;
    Private Node <E> Rightchild;

    Public Node ()
    { element=null;
      Parent = null;
      leftchild = null;
      Rightchild = null;
    }
```

void constructTree (E[] eltArray)

void insertChild (node<E> curnode, E elt, char type);

void printTree ();

Depth = 4



$2^{depth+1} - 1 = 31$

Height = h

no. of nodes = $2^{h+1} - 1$

In a binary tree,

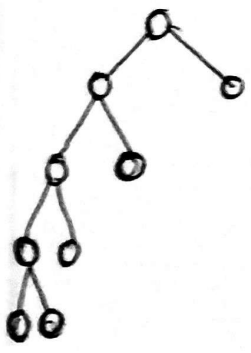
$h+1 \leq n \leq 2^{h+1} - 1$

$h \leq n_I \leq 2^h - 1$

$1 \leq n_E \leq 2^h$

$\log(n+1) - 1 \leq h \leq n-1$

Proper tree - A proper node should have 0 or 2 children

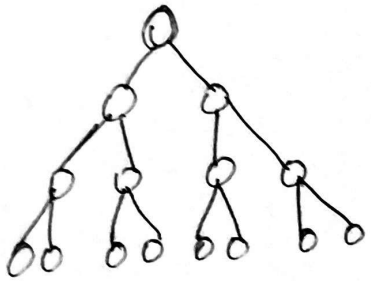


→ Proper Binary tree with minimum number of nodes

$n = 2h+1$

$n_E = h+1$

$n_I = h$



→ Proper Binary tree with max no. of nodes

For Proper tree. (or) full tree.

$$2h+1 \leq n \leq 2^{h+1}-1$$

$$h \leq n_I \leq 2^h - 1$$

$$h+1 \leq n_E \leq 2^h$$

$$\log(n+1) - 1 \leq h \leq (n-1)/2$$

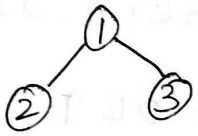
NODE:-

Binary tree → 0, 1, 2 children

Proper tree → 0 (or) 2

complete Binary tree → Except last level all the levels are having max. no. of nodes. Last level can have 1 node or maximum no. of nodes.

Perfect tree - All the levels are having maximum no. of nodes.



TREE TRAVERSAL

1. Inorder Traversal

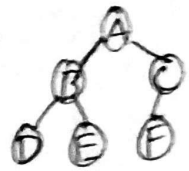
The order of traversal is 2 1 3

2. Preorder Traversal

order of traversal 1 2 3

3. Postorder Traversal

order of Traversal 2 3 1



In order traversal of A.

↳ In order traversal of left child B

↳ In order traversal of left child D

↳ In order traversal of right child E

↳ In order traversal of right child C

↳ In order traversal of left child F

D B E A F C

Pre order traversal of A.

A B D E C F

Post order traversal of A.

D E B F C A

LEVEL ORDER TRAVERSAL:

Level order traversal of A

A B C D E F

Traversing the nodes level by level.

Process - visit