DS / Data Structure

02/04/24

Lab + Theory

L 1 (0-1)

Topic - 1

Searching (Imp Pgm final)

Linear Search (@ O(n))

Given 'value' and array a[], find index i such that a[i] = value, or report that no such index exists

↑ this case, a[] is search space = place to search in In search space

* Best Case Condition : value in first index of search O(1) O(≈ n)
* Average " : value in any index thats not in first or last
* Worst " : value in last index of search (@) O(n)

Time Complexity: Big Oh (O)
" Theta (Θ)
" Omega (Ω)

} Asymptotic Notation

Can be expressed in a number line in between 1 and n!

Random Search

Given . . . , randomly pick index i such that . . .

* Exploitation : Reduction of search space

No replace

Best Case : 1

Average " : $\approx n$

Worst " : n

* Linear search is a variation of Random Search

Replace

Best Case : 1

Average " : n

Worst " : ∞

Binary Search

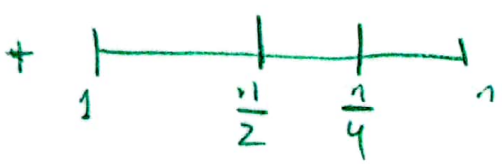
Given 'value' and Sorted array . . .

Invariant : Maintains $a[low] \leq value \leq a[high]$

Best case : 1

Avg " :

Worst " : $\log_2(n)$



until you find it

Binary Search Algo

Bin_search (A, k) (A → Array, k → value)

```
low = 0
high = size(A) - 1
while low <= high
    mid = (low + high) / 2
    if A[mid] = k :
        return found
    else if A[mid] > k :
        high = mid - 1
    else
        low = mid + 1
return not found
```

; [replace {{ with (

* Algorithm is code without syntax

+ Binary search in an unsorted array is the same as replacing random array

+ For best case, worst case of B.S, we trace

* use for any sorted ~~arr~~ thing

2/4/24

C-1 (W-1)

Pseudocode, Algorithm, Flowchart

* Pseudocode :

Artificial and informal language that helps to develop algorithms. Very similar to plain/everyday English (Avoid syntaxes and mathematical sym operation symbols)

* Algorithm :

Series of instructions to accomplish a task; very close to programming language; more like steps involving notations; ordered sequence of steps

(Avoid syntaxes

* Flowchart :

Graphical representation of the sequence of operations (i.e. Algorithms

*Example:

write an algorithm to determine a student's final grade and indicate whether the student is passing/failing. The final grade is calculated as the average of four marks.

Pseudocode:

1. input a set of 4 marks
2. calculate the average by summing and dividing by 4
3. If average is below 40, print fail
4. else print pass

Algorithm:

1. input M1, M2, M3, M4
2. grade ← $\frac{M1+M2+M3+M4}{4}$ (used for assigning)
3. if grade < 40 : (in then) print fail
4. else print pass

code:

```
1. cin >> M1 >> M2 >> M3 >> M4;
2. double grade = (M1+M2+M3+M4) / 4.0;
3. if (grade <40.0){
       cout<< "Fail" <<endl;
   }
4. else{
       cout << "Pass" <<endl;
   }
```

* "Class theke kisu asbe na" ~ Sir "15 marks common from moments"

From an array find minimum value:

pseudocode:

1. take min as first element of array
2. run a loop for each element in array
3. if min is greater than element, change the value of min to that element
4. else continue running the loop
5. print min

algorithm:

1. min ← arr[0]
2. For i = 0 to size of arr:
   - if min > arr[i]:
     - min ← arr[i]
   - end if
   - else
   - end for
3. print min

code:

```
1. int min = arr[0];
2. for (int i = 0; i < sizeof(arr); i++) {
       if (min > arr[i]) {
           min = arr[i];
       }
3. cout << min << endl;
```

C-2 (w-2)

15/04/24

Selection Sort

1) Swap two numbers ⟶ Swap (A, B)
Built in c++ function

2) Find the minimum value from a vector of integers

Algorithm: 0. Find_Min (A):

```
1.      n = size(A)        → finde size of vector
        min = A[0]         → considering first term to be lowest
2.      pos = 0
3.      For i = 1 to n-1:
4.          if A[i] < min :
5.              min = A[i]
                → pos = i
6.          end if
7.      end for
8.      print min
        → print pos
```

③ Find index of min

→ compare and swap find min

Finding index

20 29 72 98 13 87 66 52 51 36

0. 13 | 72 98 29 87 66 52 51 36

assumption of next array

1. 13 29 | 98 72 87 66 52 51 36

2. 13 29 36 | 72 87 66 52 51 98

3. 13 29 36 51 | 87 66 52 72 98

4. 13 29 36 51 52 66 87 72 98

5. 13 29 36 51 52 66 87 72 98

6. 13 29 36 51 52 66 72 87 98

7. 13 29 36 51 52 66 72 87 98

Algorithm Passing

* For n terms, sort has to be done n-1 times

Passing ⟶ ~~Iteration~~ Iteration / Tracing of

[4] Selection Sort Algorithm:

```
0. Selection_Sort(A):
1.     n = size(A)
2.     For j=0 to n-2:
3.         min = A[j]
4.         pos = j
5.         For i = j+1 to n-1;
6.             if A[i] < min:
7.                 min = A[i]
8.                 pos = i
9.             end if
10.        end For
11.        swap A[pos], A[j]
12.    end For
```

<= ⟶ do NOT use min

Complex. (min)

Best : n

Avg : n

Worst : n

Complexity of Selection Algo sorting:

Best : $n(n-2) = n^2 - 2n$

Avg : $n(n-2) =$ "

Worst : $n(n-2) =$ "

C-3(W-2) 16/04/24

Insertion Sort

* Insert an element into the sorted array

A = | 10 | 15 | 20 | 25 | 30 |

Says we want to insert 18,

Two different ways to achieve this ⟶

(1) ~~She~~ Search, then shift → Complexity (n): [number of search + no. shift]

(2) search + Shift

*Shift from last element

~~She~~ Search then ~~and~~ Shift:

A = | 10 | 15 | ~~18~~ | 20 | ~~25~~ | 30 |

Search and Shift:

Use j-- to navigate

if insert ~~0~~ ~~0~~ >=index, A[index]

~~A[index] =~~

A[index+1] = ~~A[index]~~ insert

else

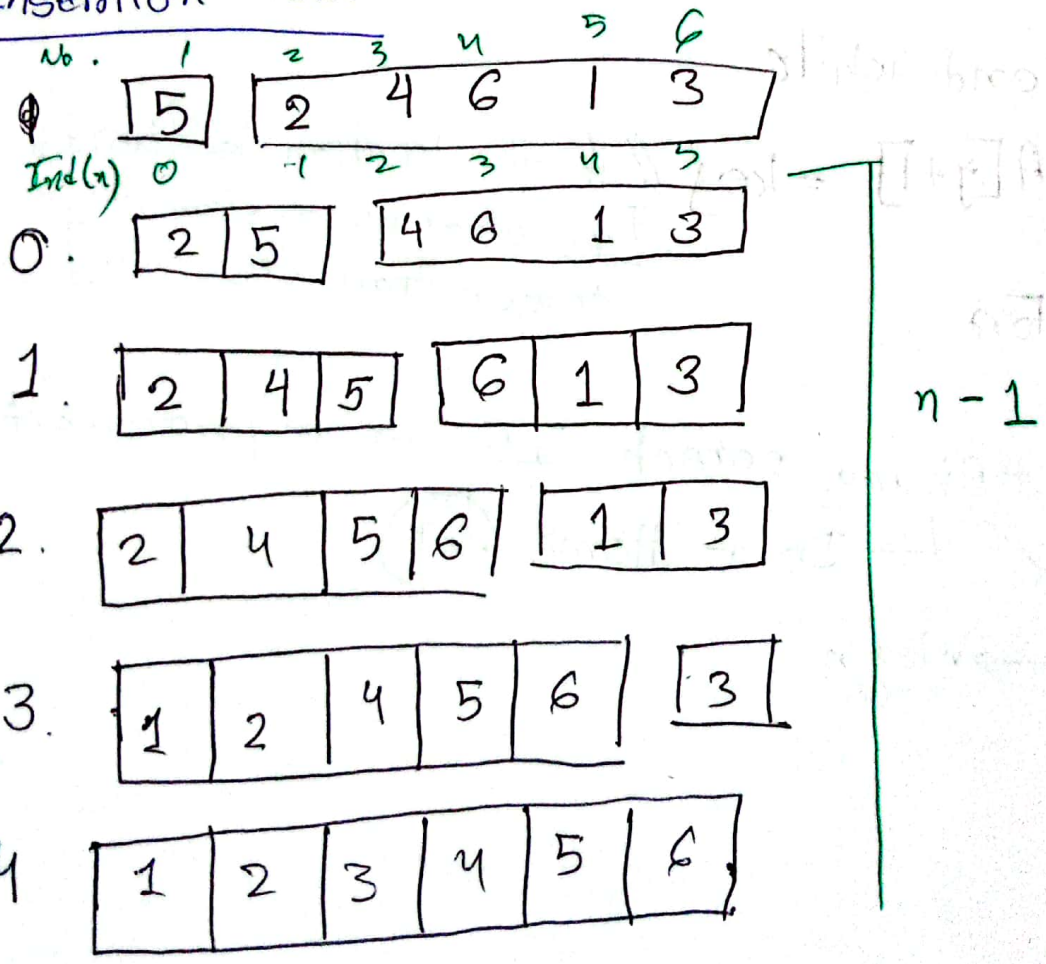
A[index+1] = A[index]

Algorithm (Insert):

→ insert element

0. Insert(A, key):
1. n = size(A)
2. j = n - 1
3. while j ≥ 0 and A[j] > key
4. | A[j+1] = A[j]
5. | j = j - 1
6. end while
7. A[j+1] = key

Insertion Sort:



Algorithm (Insertion Sort):

```
0. Insertion_Sort(A):
1.      n = size(A)
2.      For i = 1 to n-1: //starts from 1 NOT 0
3.          key = A[i] //consider the i-th term to be inserted
4.          j = i-1 // to work on the prev indexes
5.          while j ≥ 0 and A[j] > key:
6.              A[j+1] = A[j] // shift
7.              j = j - 1 // for the loop
8.          end while
9.          A[j+1] = key // if the location is found; [the insert element is greater than the index]
10.     end For
```

Complxty:

Best : n

Avg : n^2 → $n \log_2 n$

Worst : n^2

* Binary search IS an improvement
  ↳ Do at Home (for Quit)

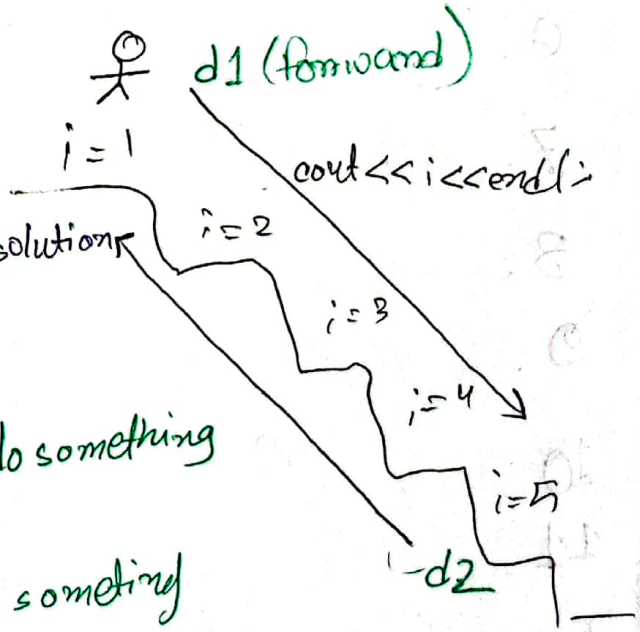
C-4 (W-3) 22/04/24

Recursion:

① Definition of Recursion = see the definition of Recursion

② n = 5 ; 1 2 3 4 5
5 — 4 — 3 — 2 — 1 — 0

for (int i = 0; i <= n; i--) → iterative solution
{
    cout << i << " "; // do something
}



③ void FOR (int i, int n) → recursive solution
{
    if (i <= n)
    {
        d1 cout << i << " "; // do something
        FOR (i+1, n);
        d2 cout << i << " "; // do something
    }
}

* Useful for [dynamic programming] → saves every state of the program
  ↳ params become "states"

int main ()
{
    FOR (1, 5);
}

Recursion Steps:

(12)

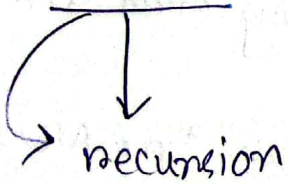
1. call FOR_1 with $i = 1$
2. call FOR_2 with $i = 2$
3. call FOR_3 with $i = 3$
4. call FOR_4 with $i = 4$
5. call FOR_5 with $i = 5$
6. call FOR_6 with $i = 6$
7. break; no more calls
8. return to FOR_5
9. print 5
10. return to FOR_4
11. print 4
12. return to FOR_3
13. print 3
14. return to FOR_2
15. print 2
16. return to FOR_1
17. print 1
18. return to main

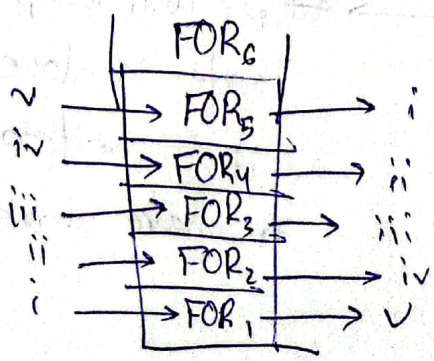
* Reversing an array :

```
void FOR (int i, int n) // (int i, n)
{
    if (i >= 0) // (i < n)
    {
        cout << i << " "; // nothing
        FOR(i - 1); // (i+1, n)
    }
}

int main()
{
    FOR(n - 1); // FOR(0, n-1)
}
```

* Memory where recursion is kept is called STACK MEMORY

Stack





L-3 (W-3)

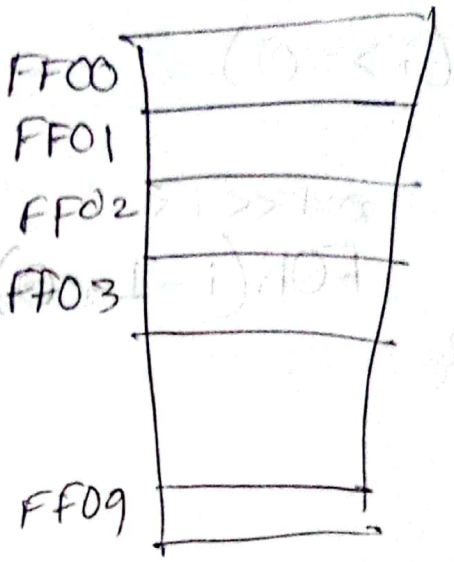
Pointers

↳ important for linked list

23/04/24

```
int x = 5
```

variable : x
value : 5
Address : FF00




```
int x = 5
int *p
p = &x
```
} int *p

*After declaring p = &x

```
*p = 25
≡ *(&x) = 25
≡ x = 25
```

+ int *p = NULL
  p = &x
→ p will keep pointing even after code is closed
∴ it is better to use *p = NULL to make sure address is not assigned

* int *p = arr; ≡ int *p = &arr[0]

arr = {10, 12, 13, 14, 15}

arr[0] ⟶ 10

&arr[0] ⟶ address of 10

*(&arr[0]) ⟶ 10

*(&arr[0]) + 1 ⟶ 11 // 10 + 1

↳ *p + 1

*(&arr[0] + 1) ⟶ 12 // addr. of 10 + 1

↳ ↳

* Good practice to free the pointers

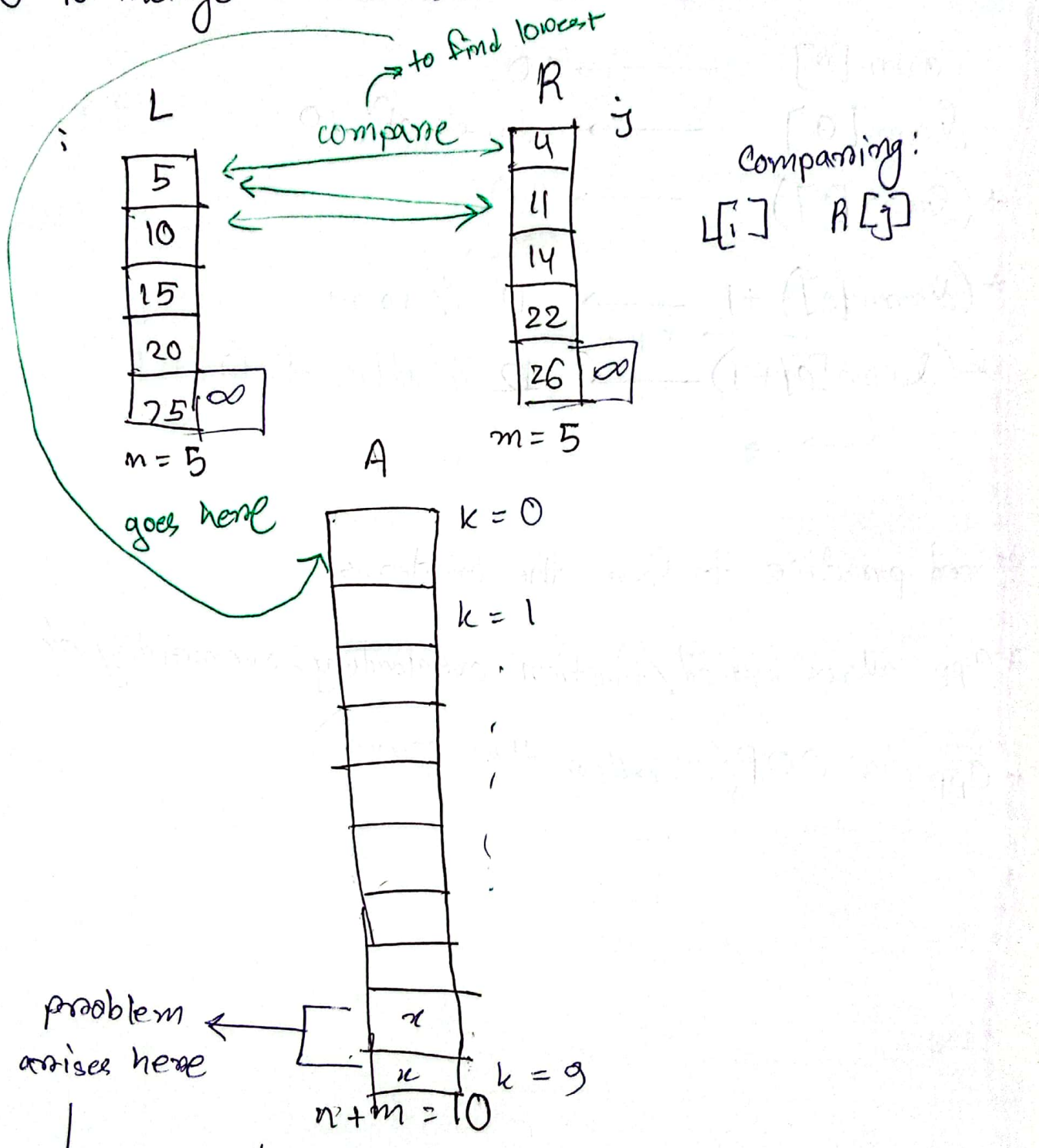
* Cpp allows method/function overloading, overriding etc

* Cpp is OOP (:. better than java)

C-5(w-3) Merge Sort: Merge 23/04/24

↳ Conquer

* How to merge two sorted lists into one sorted list



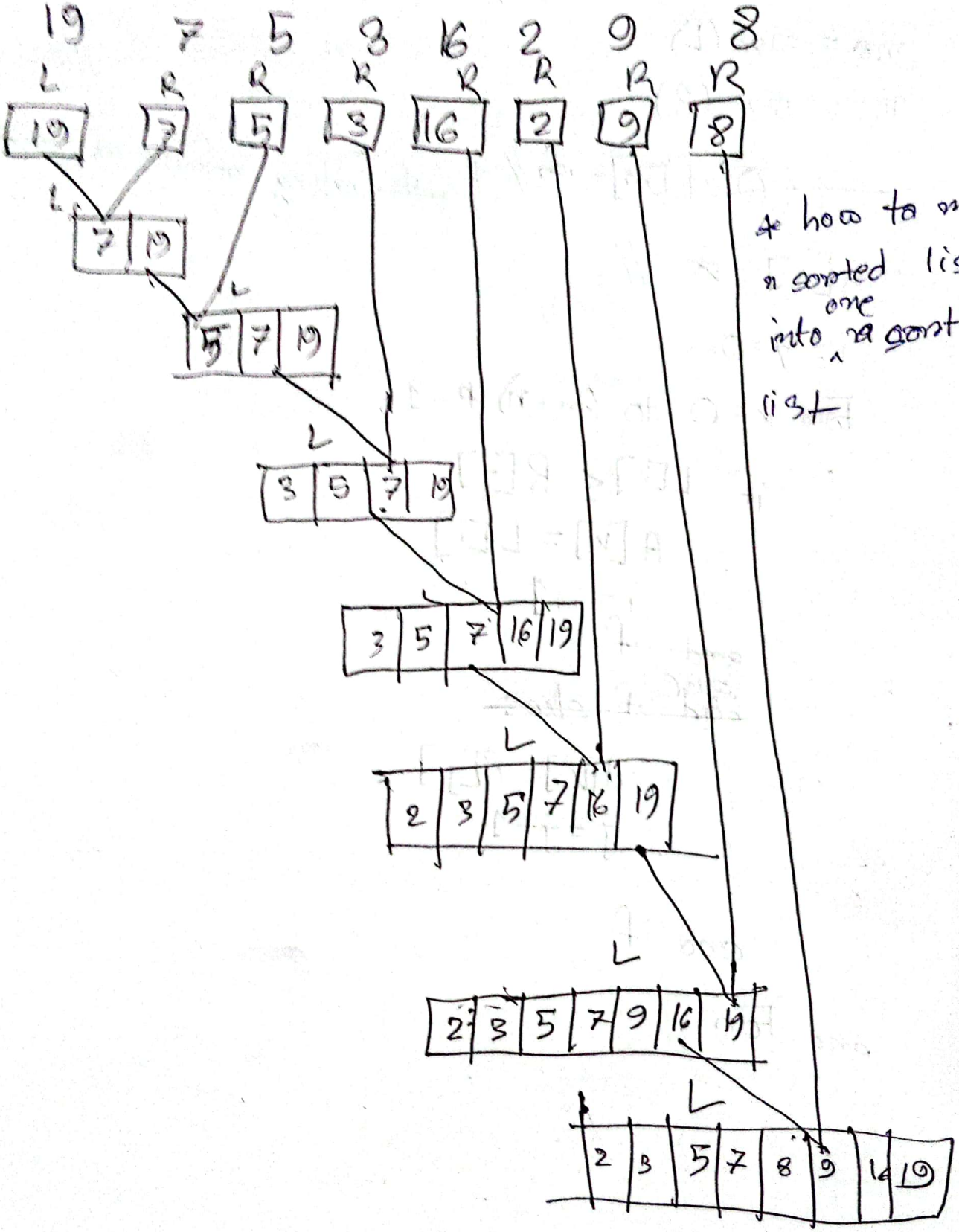
↳ same value will ~~happ~~ be placed here twice

↳ sol^n → add index + put infinity there

*Algorithm for merging:

```
0. MERGE(L, R)
1.      m = size(L)
2.      n = size(R)
3.      L[m] = ∞ //   } adding infinity at the end
4.      R[n] = ∞ //
5.      i = j = 0
6.      For k = 0 to (m+n) - 1 :
7.          if L[i] ≤ R[j]:
8.              A[k] = L[i]
9.              i = i+1
10.         else:
11.             A[k] = R[j]
12.             j = j+1
13.         end if
14.     end For
```

* MERGE SORT

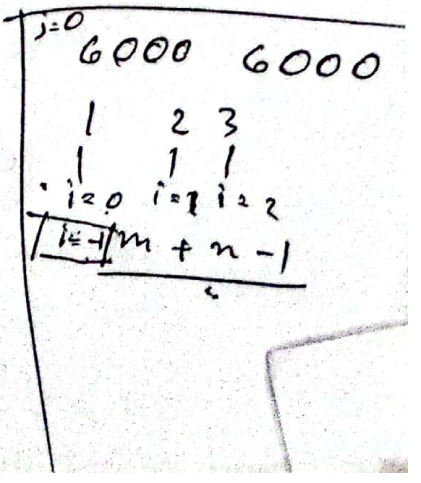


* how to merge n sorted lists into one sorted list

* Algorithm (one ascending, one descending)

L → ascending, R → descending

```
MERGE(L, R)
    m = size(L)
    n = size(R)
    L[m] = ∞
    R[n] = -∞ ⟶ do not use
    i = 0
    j = n-1
    For k = 0 to (m+n)-1:
        if L[i] ≤ R[j]:
            A[k] = L[i]
            i = i+1
        else:
            A[k] = R[j]
            j = j-1
        end if
    end For
```



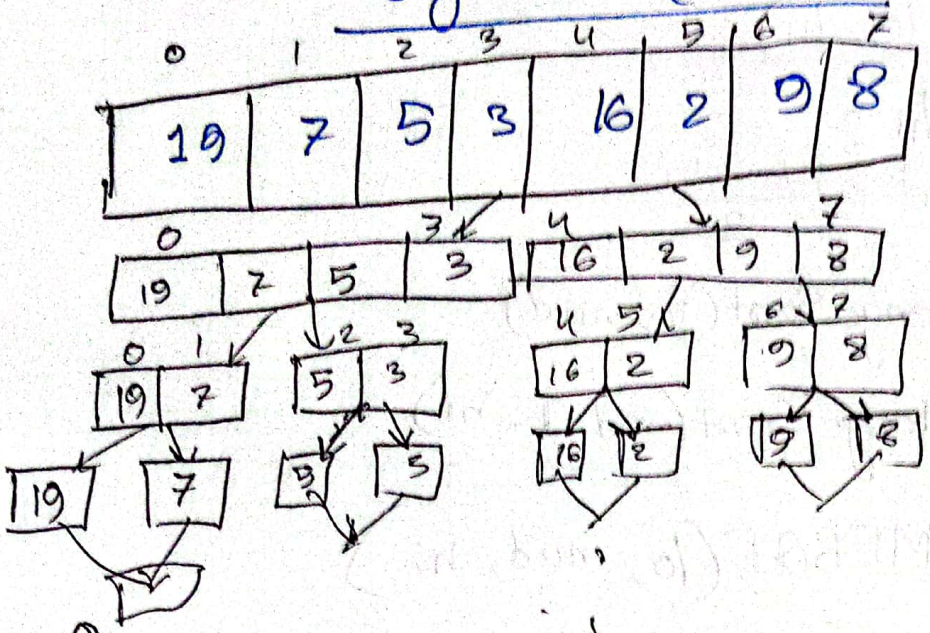
Alt:

```
0. MERGE (L, R):
1.      m = size(L)
2.      n = size(R)
        k = 0
3.      i = 0
        j = n - 1
        while i ≤ m and j ≥ 0:
            if L[i] ≤ R[j]:
                A[k] = L[i]
                i = i + 1
                k = k + 1
            else:
                A[k] = R[j]
                j = j - 1
                k = k + 1
            end if
        end while
        while i < m:
            A[k] = L[i]
            k = k + 1
            i = i + 1
        end while
        while j ≥ 0:
            A[k] = R[j]
            k = k + 1
            j = j + 1
```

C-6(w-4)

29/04/24

Merge Sort (Divide)



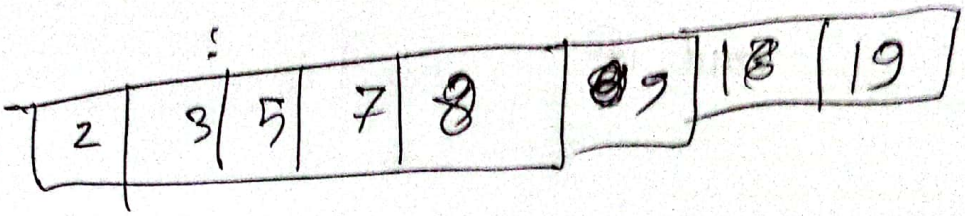
low = 0

hi = 7

$$mid = \frac{low + hi}{2}$$

$$= \frac{0 + 7}{2}$$

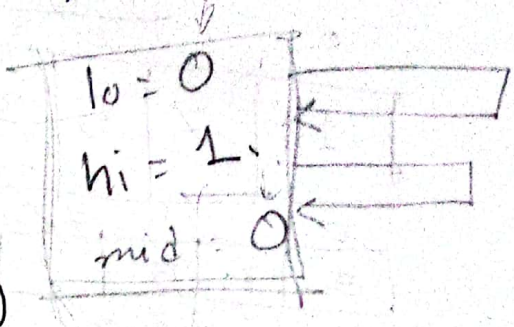
$$= 3$$



* Algorithm:

Complex: $n \log_2 n$

0. ~~ME~~ MergeSort(lo, hi)
1. if lo < hi
2. mid = $\frac{lo + hi}{2}$
3. MergeSort(lo, mid)
4. MergeSort(mid+1, hi)
5. ~~end if~~ MERGE(lo, mid, hi)
6. end if



```
0.  MERGE(lo, mid, hi):
1.      m = mid - lo + 1
2.      n = hi - mid
3.      Let, L[0..m-1] and R[0...n-1] be new arrays
4.      For i = 0 to m-1:
5.          L[i] = A[lo + i]
6.      end For
7.      For j = 0 to n-1:
8.          R[j] = A[mid + 1 + j]
9.      end For
10.     L[m] = ∞, R[n] = ∞
11.     i = j = 0
12.     For k = 0 to hi:
13.         if L[i] ≤ R[j]:
14.             A[k] = L[i]
15.             i = i + 1
16.         else:
17.             A[k] = R[j]
18.             j = j - 1
19.         end if
20.     end For
```

0 → m + n - 1
lo → hi

~~L-5~~ L-4

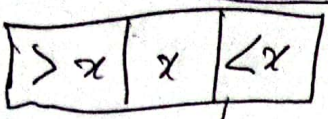
Quick Sort

30/4/24

[Online]

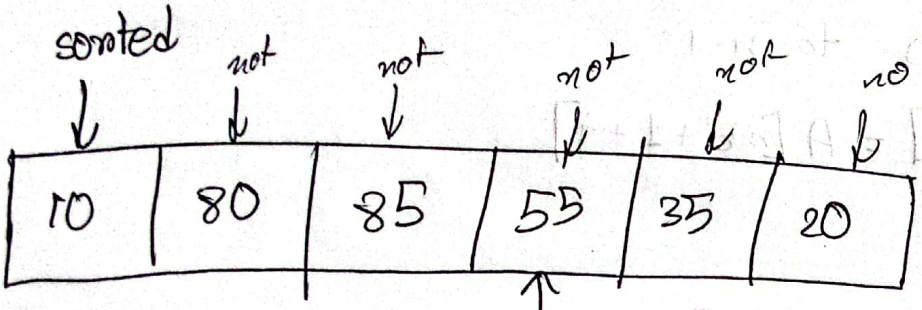
A number is sorted when,

or,



→ elements in >x and <x being sorted by themselves is not a requirement

So,

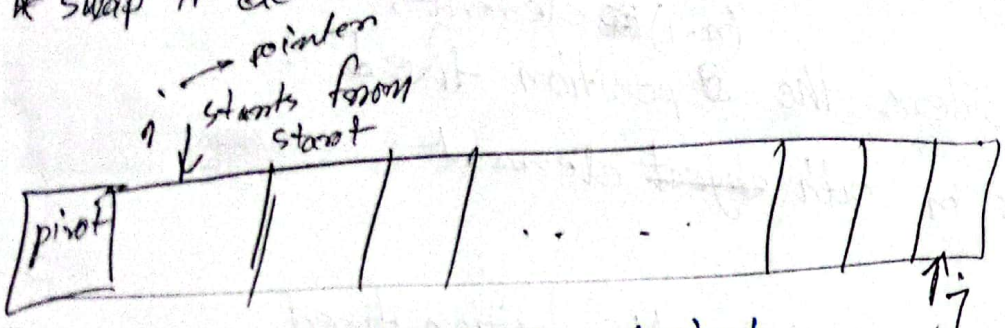


might be in its position but its not correct according to the definition

* To find the sorted position

Pivot → element in its sorted position
- → can be
  - → start
  - → end
  - → mid
  - → random

* Swap if element is not in position



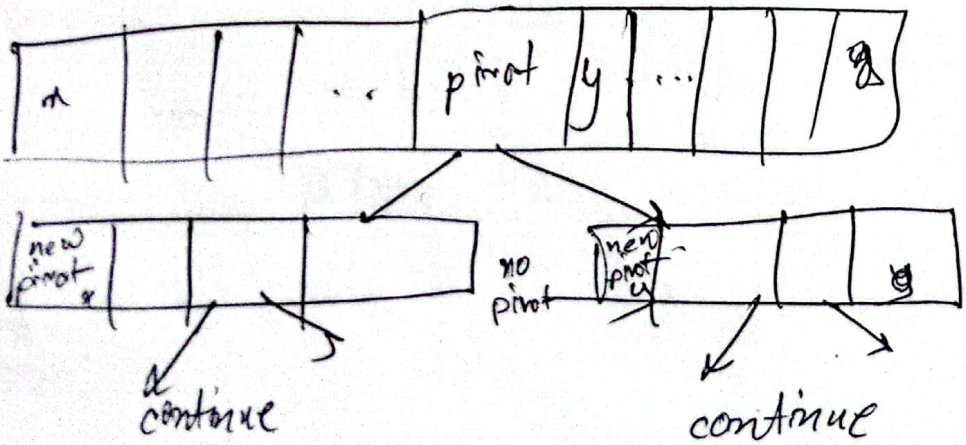
i++ → happens when ≤ pivot
→ until ~~≥~~ ~~pivot~~ ≥ pivot

j-- → happens until ≤ pivot
→ until ≥ pivot

swap (i, j) → swapping between small and beeg

when j < i,

swap (pivot, j) → partitioning
→ swapping between ~~bee~~ pivot and j



* Diff bet" merge and quick

1. based on index
1. based on pivot

Sel Sort ⟶ Considers the (n-1) elements position fixed
Puts in nth object element

Quick Sort ⟶ Considers the nth position fixed
Puts in (n-1) elements

Algorithm

```
0. Partition (lo, hi):
1.     pivot = A(lo)
2.     i = lo // starts from lowest index
3.     j = hi //  "      "  highest index
4.     while i<j : // will run until they dont cross
5.         while A[i] ≤ pivot:
6.             i = i+1 // to find big number
7.         end while
8.         while A[j] ≥ pivot:
9.             j = ~~j++~~ j-1 // to find small number
10.        end while
11.        if i<j:
12.            swap A[i], A[j]
13.        end if
14.    end while
15.    swap A[lo], A[j]
16.    return j
```

0. QuickSort(lo, hi):

1. if lo < hi:

   p = Partition(lo, hi)

2. Quicksort(lo, p-1) // left side

3. QuickSort(p+1, hi) // ~~hi~~ right side

4.

Complexity:

Partition : $O(n)$

Recursive Breaking Down : $O(\log_2 n)$

∴ QuickSort = ~~$n \log_2 n$~~ Best : $n \log_2 n$

Avg : $n \log_2 n$

Worst : n^2

L-5 (W-5) Linked List 07/05/24

Linked list can be defined as a collection of data items that can be stored in scattered memory locations. Here, data items must be linked with each other. Data items are known as ~~node~~ nodes. the

* A linear linked list (singly) is a list where there is only one way link between nodes.
* A doubly linked list is a list where there are links in both directions between nodes.
* A circular linked list is a list where the last node has a pointer which points to the first node

Each type of linked list requires an external pointer to point the first node of the list (root)

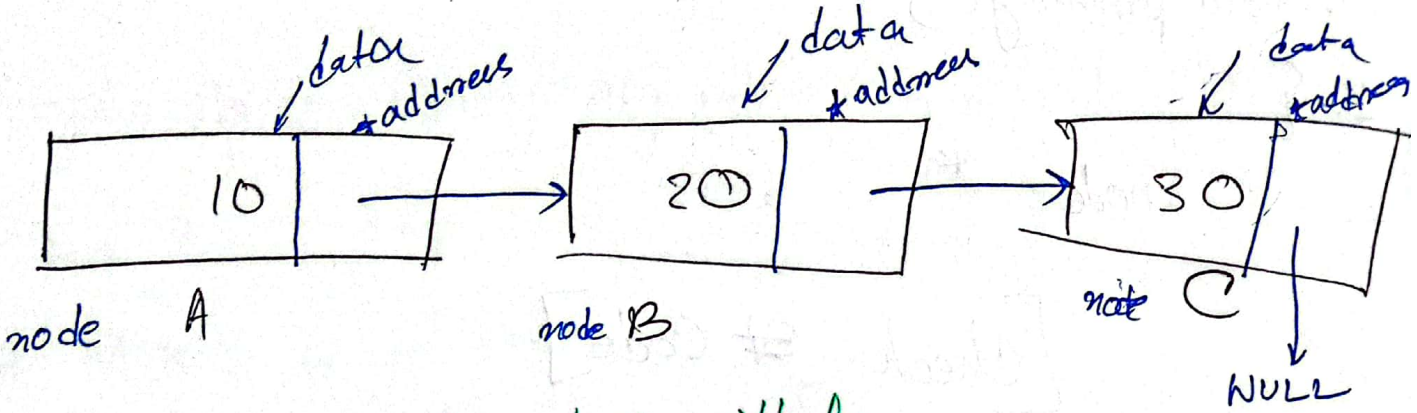
- insert
  - first
  - last
  - anywhere
- delete
  - first
  - last
  - anywhere
- traverse
  - print
  - search
  - others
- create
  - list/structure declaration
  - pointer declaration

```
#include
using

struct node
{
    int data;
    node *address;

};
node *root = NULL;
int main()
{
    node *A, *B, *C;
    A = new node();
    B = new node();
    C = new node();
    A.data = 10;   A->data = 10;
    B.data = 20;   B->data = 20;
    C.data = 30;   C->data = 30;
    A->address = B
    B->address = C
    C->address = NULL
    cout << A.data << " " << B.data << " " << C.data;
    root = A;
    cout << A->data << " " << A->address->data << " "
         << A->address->address->data;
```

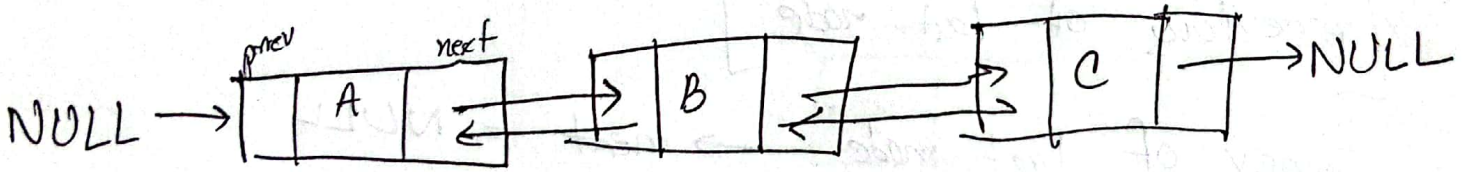
root

root

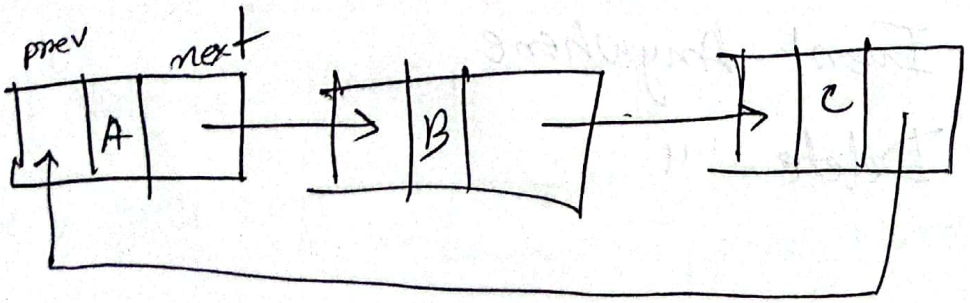
→ show error



* We want to call B and C with A
* Not possi! Does not have ~~same~~ sequential address
* A will have address of B
* data type will be struct for struct pointer
* Cpp uses -> for pointer struct.



~~Doubly~~ Doubly



0. void printing()
1. {
2. node *

[Check ~~St~~ Code]

* Always assume list is ready
* If inserted at first, the list is ~~atom~~ automatically reversed.

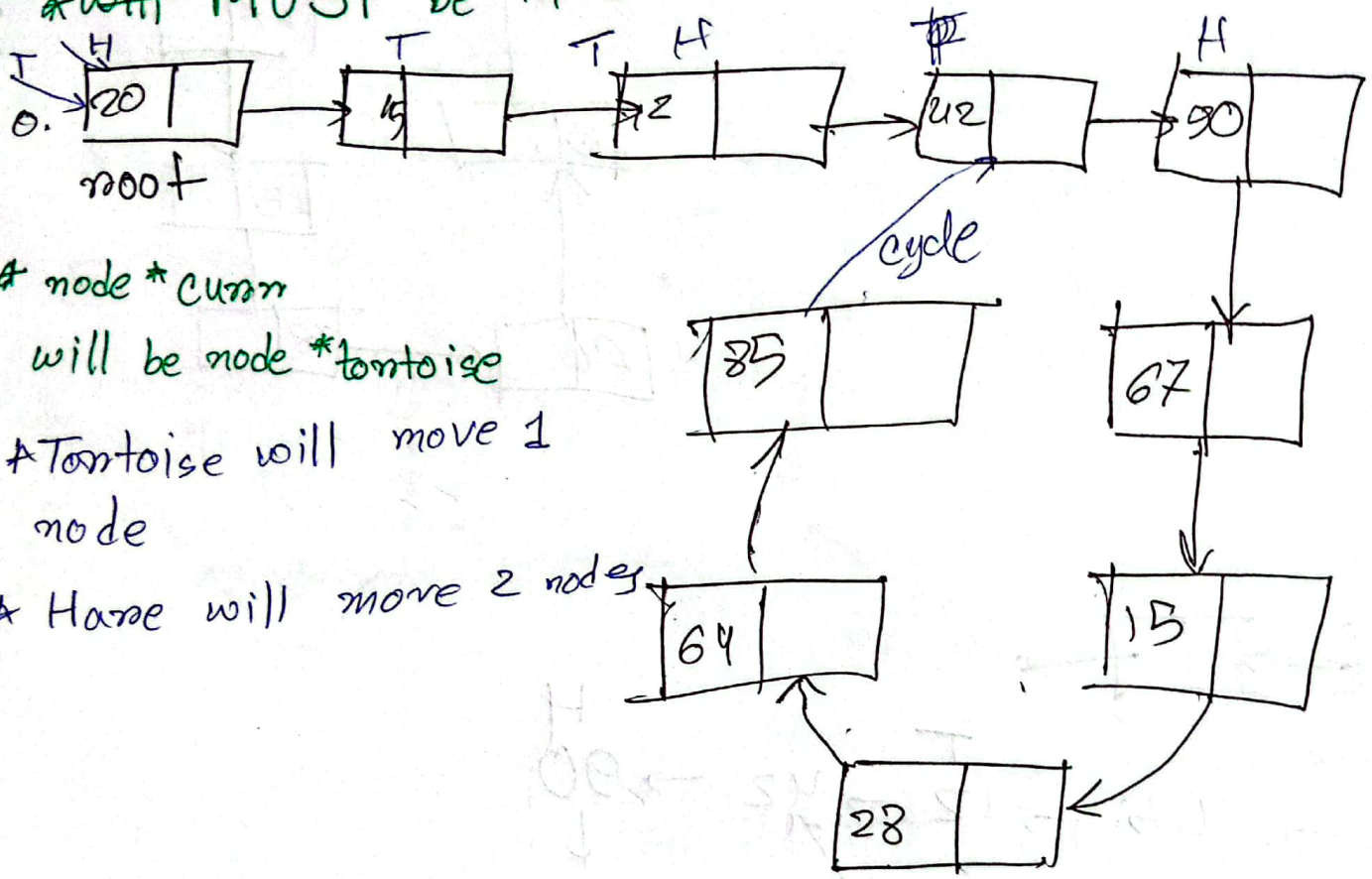
void delete last ()
{
[previous of last node]
prev of last node -> next = NULL
}

* Assignment: Insert Anywhere
Delete "

C-7 (W-6) 13/05/24

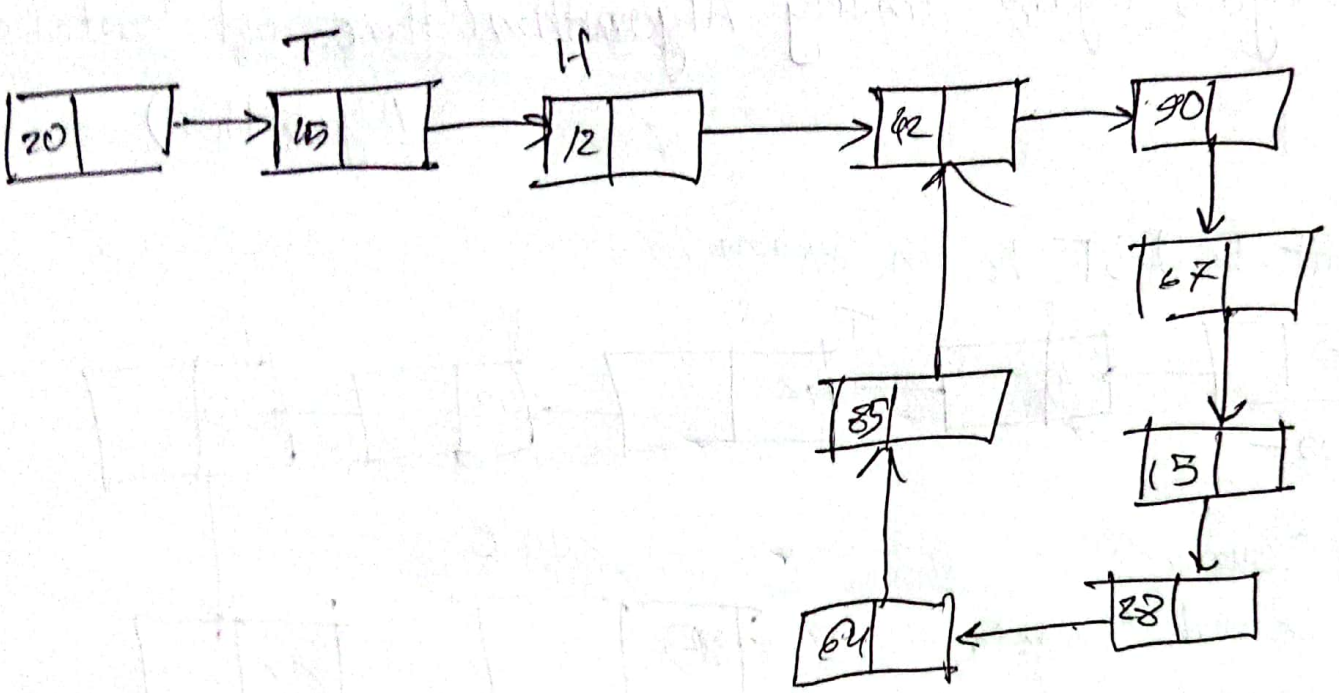
Floyd's Cycle Finding Algorithm ('Hare and Tortoise Algorithm')

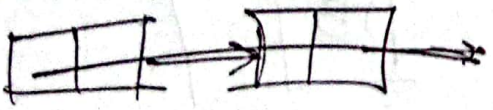
* ~~Will~~ MUST be in exam



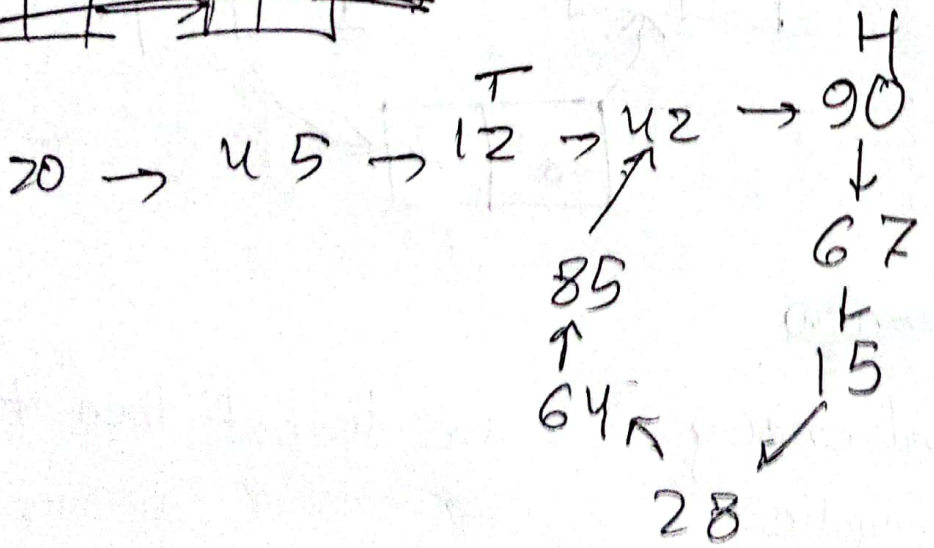
* node *curr
will be node *tortoise
* Tortoise will move 1 node
* Hare will move 2 nodes
* Imp for job interview
* If starting node and ending nodes are distinct then it ~~is a~~ will not be cyclic
* To check, you can → ~~the~~ Map nodes
→ Compare address

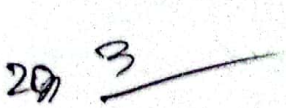
1.

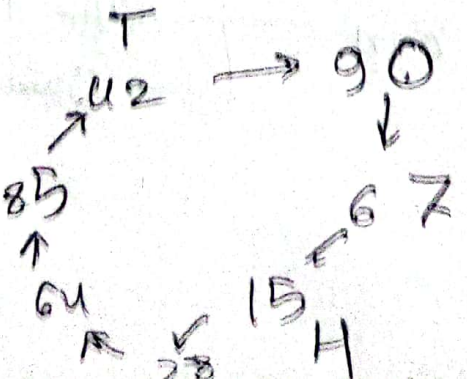

T
H
20
45
12
42
90
67
85
13
64
28

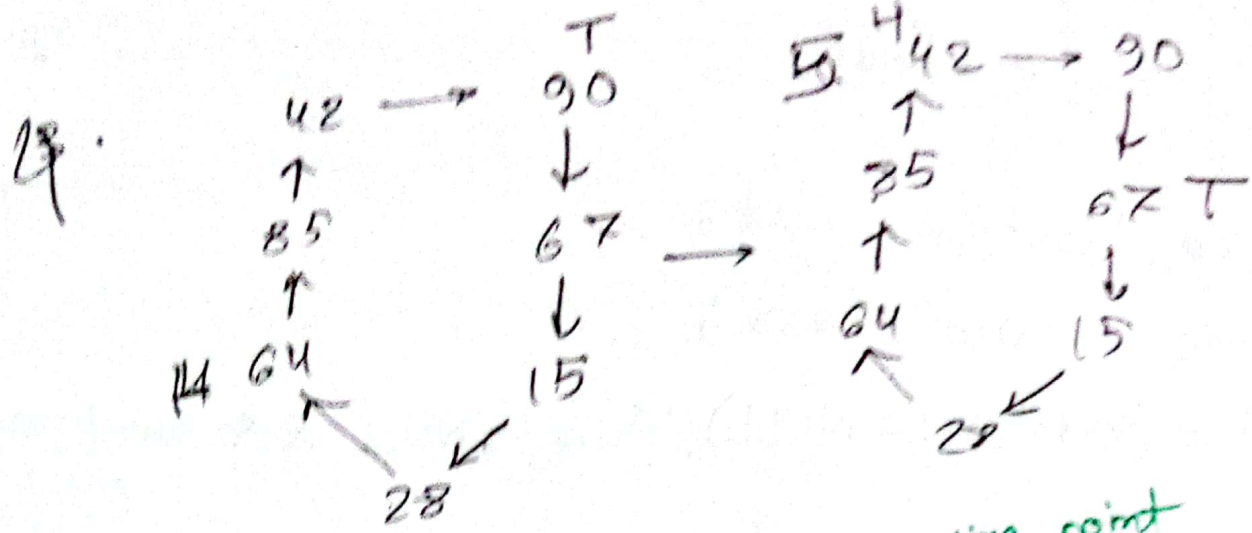


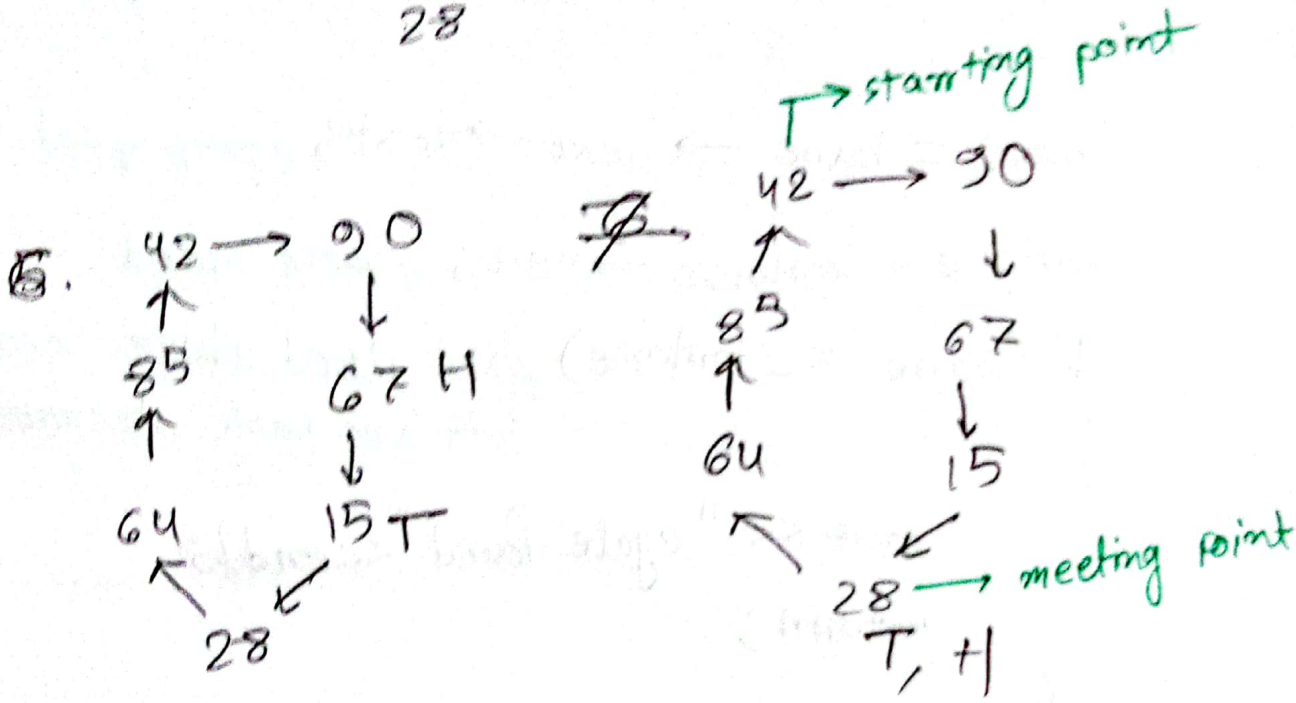
2.


T
H
20
45
12
42
90
67
85
15
64
28




T
42
90
85
67
64
15
H
28



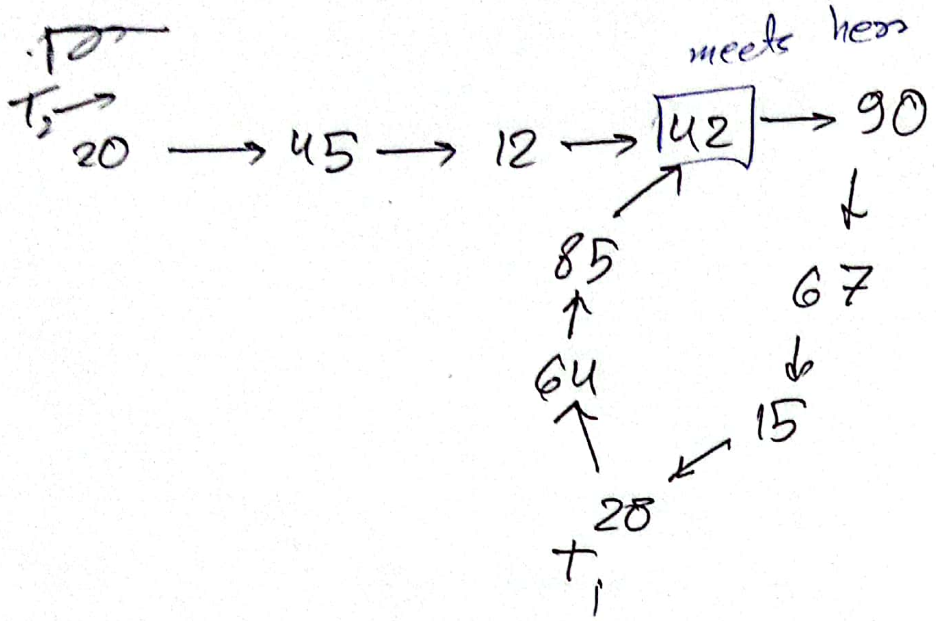


∴ T and H will meet if and only if one runs at the other's multiple speed

```
void cycle_void find()
{
    node *tortoise = root;
    node *hare = root;
    while (tortoise != NULL)// (hare!=NULL && hare->next!=NULL)
    {
        hare = hare -> next -> next; //jump speed
        tortoise = tortoise -> next; // move speed
        if (hare == tortoise) // Not placed above because both are roots at start
        {
            cout << " cycle found " << endl;
            return;
        }
    }
    cout << "cycle not found" << endl;
}
```

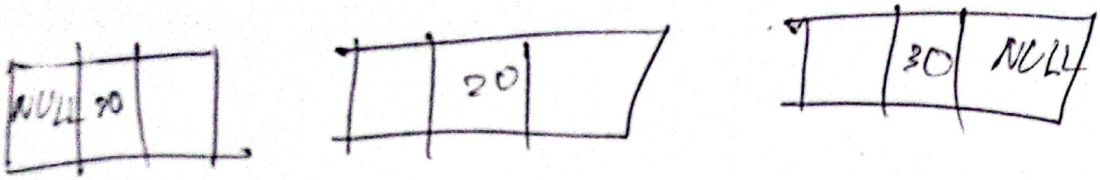
* Complexity : O(m * n)
+ Shafayet's planet
* Multiple of each other

* To find meeting point




```
void starting_node(){
{
    node *tortoise = root;
    .
    .
    while( . - .
    .
    .
            if(hare = = tortoise)
            {
                cout << "cycle found" <<endl;
            }
    }
    node *tortoise 2 = root;
    while(tortoise 2 != tortoise)
    {
            tortoise = tortoise -> next
            tortoise2 = tortoise2 -> next
    } cout << tortoise -> data << endl;
```

Doubly Linked List



Insert anywhere!

curr_node → next = temp

temp → next = ponen_node

+ do prev as well

~~etc~~

Delete anywhere:

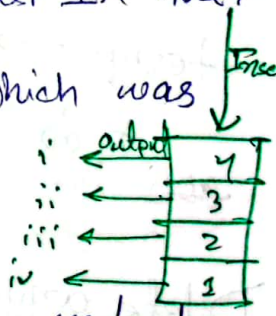
agen_node → next = ponen_node

ponen_node → prev = agen_node

C-7(W-7) 20/5/24

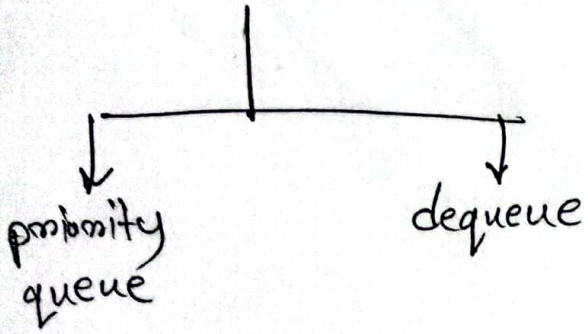
Stack and Queue

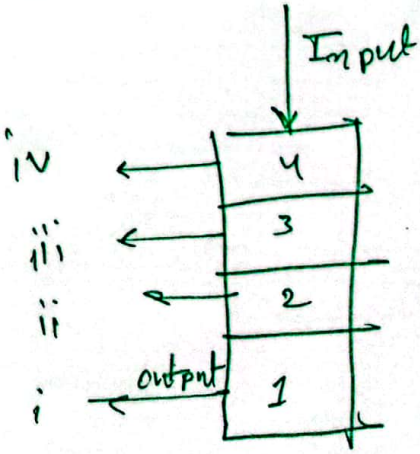
- Stack is a linear list where any element is added at the top of the list and any element is deleted/accessed from the top of the list. Add operation for a stack is called 'push' operation and deletion operation is called 'pop' operation. Stack is LIFO (Last In First Out) data structure. That means the element which was added will be deleted/accessed first.



Example: Scrolling/Browsing/Undo+Redo

- Queue is a linear list where all additions are made at one end, called 'rear' and all ~~data~~ deletions/accesses are made from another end called 'front' of the list. Que is a FIFO (First In First Out) data structure. That means the element that is added first will be deleted/accessed first.





Standard template library

```
#include
using namespace

int main()
{
    stack<int> s; // container
    s.push(10);
    s.push(20);
    s.push(30);
    s.push(40);
    cout<<s.size()<<endl;
    while (true) // while (!s.empty())
    {
        //if(s.size()>0)
        //{
            int val = s.top();
            cout<<val<<endl;
            s.pop();

        //}

    //}

}
```

```
40
30
20
10
```

Stack — using Linked List.

push() ⟶ insertFirst()

top() ⟶ root → data

pop() ⟶ deleteFirst()

size() ⟶ size() // count++

empty() ⟶ size() // count == 0

* if ~~deleted~~ inserted last, deleted last

Queue.

push() ⟶ insertLast()

front() ⟶ root → data

pop() ⟶ deleteFirst()

size() ⟶ size() // count++

empty ⟶ size() // count == 0

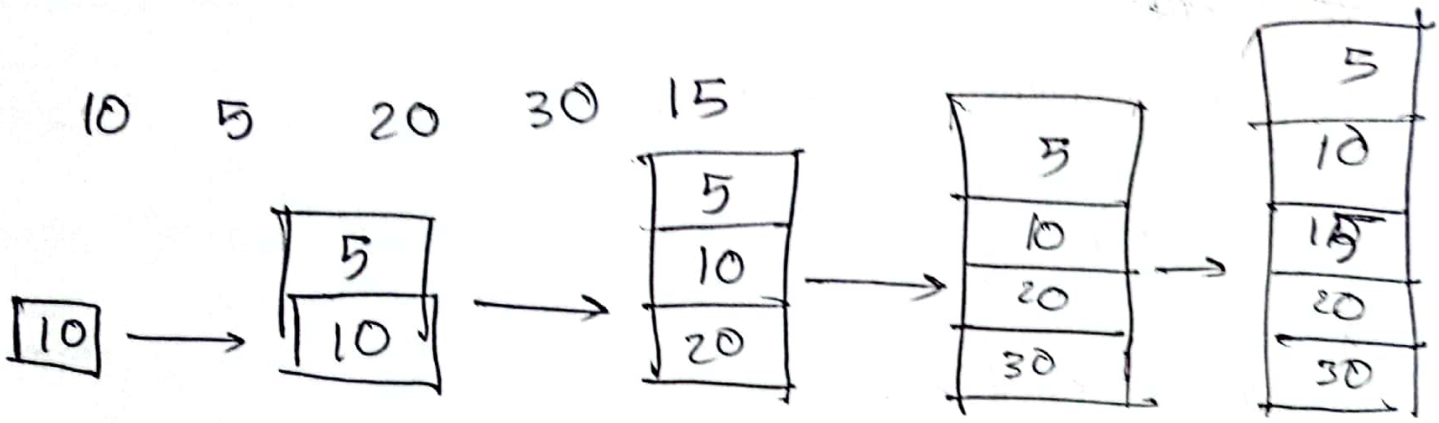
STL

Queue

```
#include <
using namespace

int main()
{
    queue <int> s;
    s.push(10);
    s.push(20);
      .
      .
    while(!s.empty())
    {
        int val = s.front();
          .
          .
    }
}
```

*Priority Queue

* setting condition for data insertion



priority – queue <int > s;

for descending

s.push(-data)

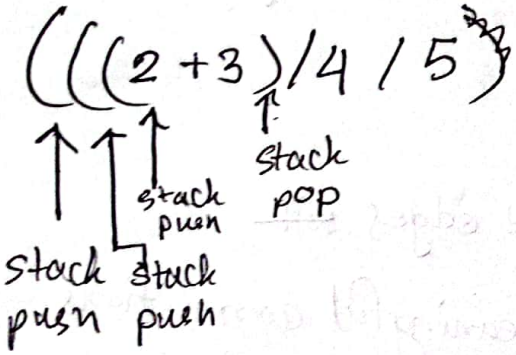
cout << -val <<endl;

for ascending

* Dequeue

* Not necessary
* Double ended removal + added.

* Balanced Parantheses Problem (Stack)



If stack size != 0, not balanced parantheses

* Light Og

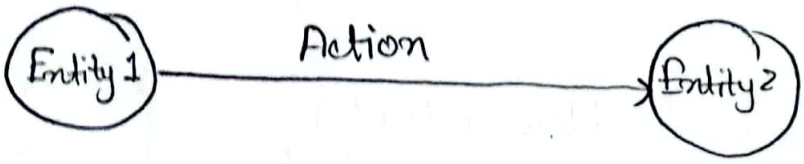
↳ Discovering the web

stack <char> temp

if '(' '{' '[' '{'

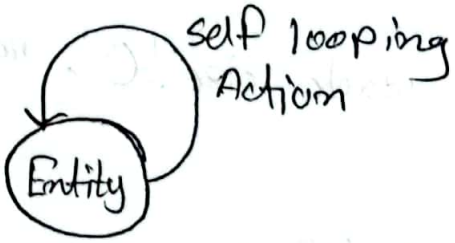
L-7(W-7) 21/05/24

Graph → 30/100 in lab
→ 30/100 in theory



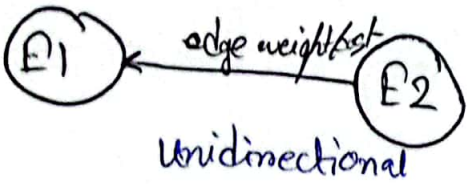
~~Whatever~~ Entities must have meaningful connections.

Graph: * A group of nodes and edges ~~will~~
* Nodes will have meaningful connections
* Connections are edges.
* Nodes > Edges

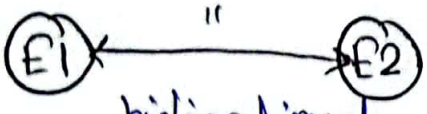


→ At least 1 node is mandatory

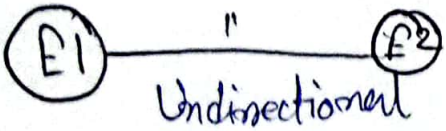
→ 0 Edges are not mandatory



Unidirectional



bidirectional

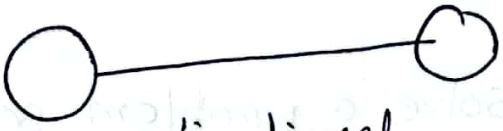


Undirectional

Difference:
bidirectional can have 2 different weight
undirectional will ALWAYS have same weight

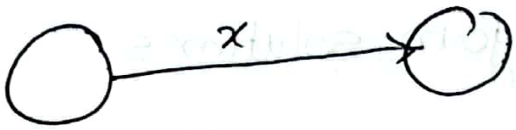
* all bidirectional are undirectional but not all undirectional are bidirectional

unidirectional unweighted graph



undirectional unweighted graph

bidirectional weighted graph



unidirectional weighted graph

* If any graph as at least one detached node, it is called disconnected graph

* Max edges : $\frac{n(n-1)}{2}$. [Undirectional]

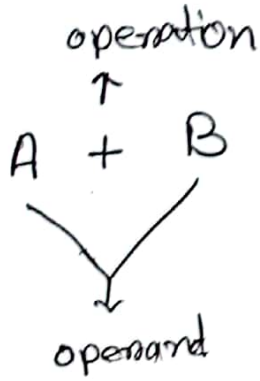
* Max edges in unidirectional, undirectional and bidirectional?

C-8 (W-7) 21/05/24

Expression

(Part of stack)

An arithmetic expression can be represented in various forms such as prefix, infix or postfix. 'Pre', 'In', 'Post' refer to the relative position of the operator with respect to its operands. For example:



infix : <operand> <operator> <operand> A + B; Easy for human, Hard for computer
prefix : <operator> <operand> <operand> + AB
postfix : <operand> <operand> <operator> AB +

6 + 2 * 7

Precedence | Associativity

(i) (), {}, []

(ii) ^ . . . R - L

(iii) *, / . . . L - R

(iv) +, − . . . L - R

Prefix :

* A * B + C / D

=> *AB + C/D

=> x + C/D

=> x + /CD

=> x + y

=> +xy

=> + *AB/CD (Ans)

1. a + b * c / d

2. 5 * (6 + 2) - (12 / 4)

1. a+b*c/d

=> a+ *bc / d

=> a + x / d

=> a + / xd

=> a + y

=> + ay

=> + a / xd

= + a / *bcd ✓

2. 5 * (6 + 2) - (12 / 4)

=> 5 * (+ 62) - (/ 124)

=> 5 * a - b

=> * 5 a - b

=> c - b

=> - b c b

=> - * 5 a b

=> - * 5 + 62 / 124

=> - - * 5 + 62 / 124

Postfix

* a+b*c/d

=> a+ bc*/d

=> a+x/d

=> a+ yxd/

=> a+y

=> ay+

=> axd/+

=> abc*d/+ ✓

* A*B+C/D

=> AB*+C/D

=> x+C/D

=> x+Y

=> xy+

=> AB*CD/+

* 5+(6+2)-(12/4)

= 5*62+-124/

= 5*x-y

= 5x*-y

= z-y

= zy-

= 5x*124/-

= 562+*124/-

* K + L - M * N / P ^ Q

=> K + L - M * N / QP ^
                    a

=> K + L - M * N / a

=> K + L - MN* / a
           b

=> K + L - b / a

=> K + L - ba/
           c

=> K + L - c

=> KL+ - c
   d

=> d - c

=> dc -

=> KL+ ba/ -

=> KL+ MN*QP^/ -.

C-9(W-8)

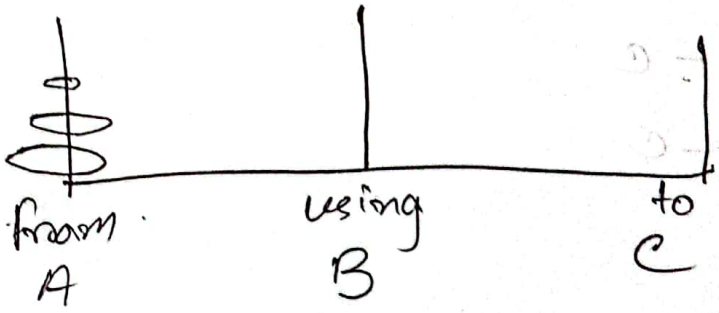
28/05/24

Tower of Hanoi

- Three towers fixed
- all discs from source to destination
- at any time, no large disc on top of smaller ones.

Answer steps:

1. A to C
2. A to B
3. C to B
4. A to C
5. B to A
6. B to C
7. A to C

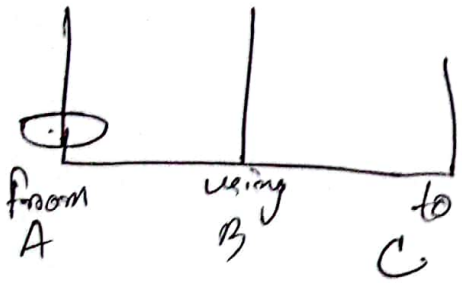


$n = 3$

$2^n - 1 = 2^3 - 1$

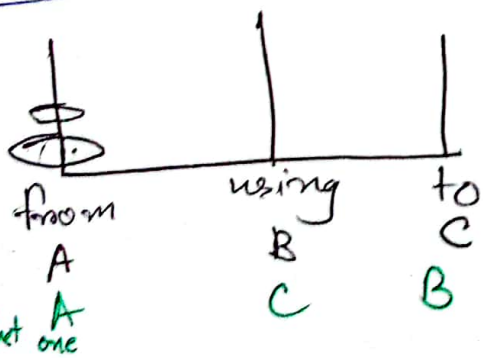
$= 7$

Solution for 1 disc (n = 1):



- Move a disc from A to C

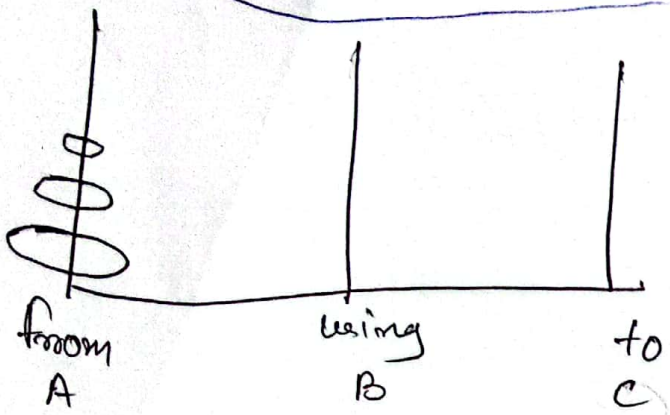
solution for 2 disc (n = 2):



for first one A

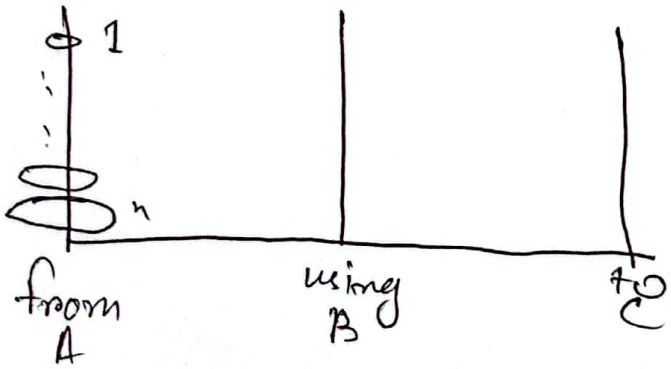
- Move 1 disc from A to B using C
- Move disc from A to C
- Move 1 dic from B to C using A

Solution for 3 discs (n = 3):



- Move 2 discs from A to B using C
- Move a disc from A to C
- Move 2 discs from B to C using A

Solution of n discs:



- Move (n-1) disc from A to B using C
- Move a disc from A to C
- Move (n-1) discs from B to C using A

Code:

```
void TOH (n, A, B, C) // no. of discs, from, using, to)
{
    TOH
    if (n > 0)
    {
        TOH (n-1, A, C, B)
        print A to C // A to C will be changed depending on the recursion
        TOH (n-1, B, A, C)
    }
}
```

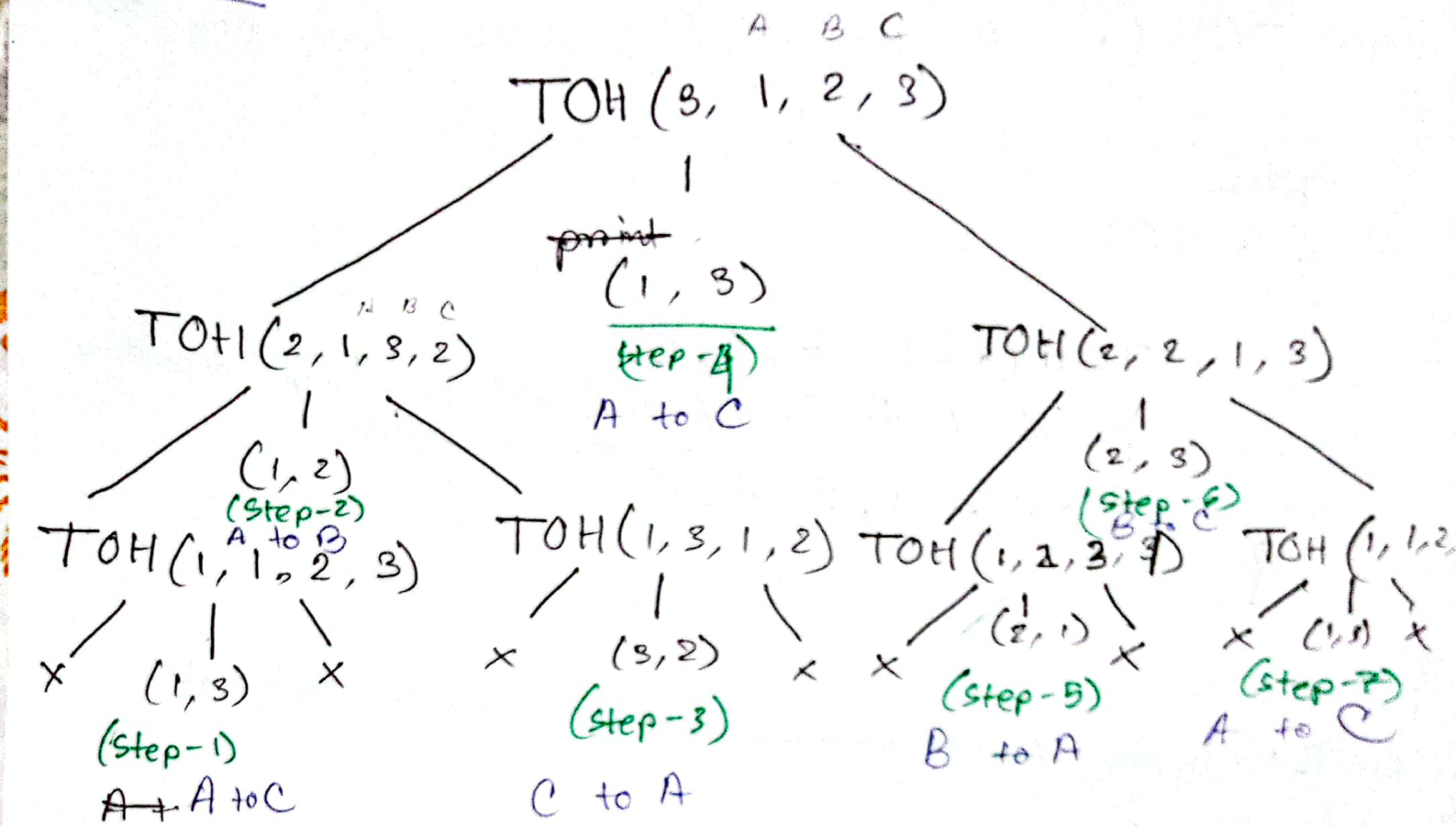
// for DM: Dynamic Programming, use an array for each state

Complexity: 2^n

* zobayen.blogspot.com — I, Me and Myself
  — Attacking Recursion

*Do TOH for 4 towers

Execution:


A B C
TOH (3, 1, 2, 3)
print
(1, 3)
(Step-4)
A to C
TOH(2, 1, 3, 2)
(1, 2)
(Step-2)
A to B
TOH(2, 2, 1, 3)
(2, 3)
(Step-6)
TOH(1, 1, 2, 3)
(1, 3)
(Step-1)
A to C
TOH(1, 3, 1, 2)
(3, 2)
(Step-3)
C to A
TOH(1, 2, 3, 3)
(1, 1)
(Step-5)
B to A
(Step-7)

L-8(W-9) 04/06/24

~~Graph~~

BFS → 0/1 BFS
→ 2D BFS → use chess

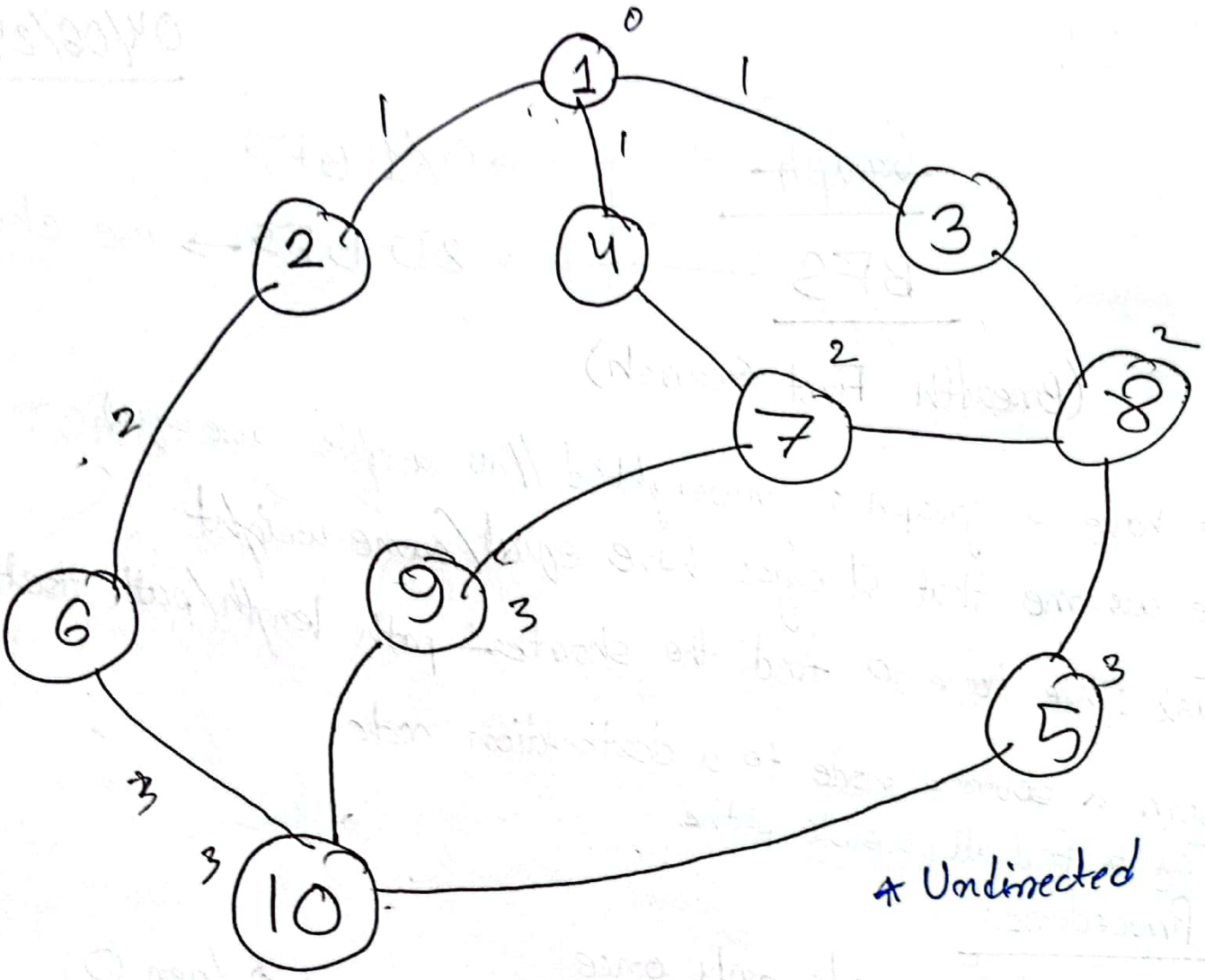
(Breadth First Search)

- We have a graph : unweighted // all weights are equal
- we assume that all edges have equal/same weight
- Task : We have to find the shortest path length/path itself from a source node to a destination node.
  // Try to find all possible paths

BFS Procedure:

- we ~~ha~~ visit a node only once
- we assume that the source node is in level 0/gen 0
- All the nodes reachable from source nodes are in level 1; all the nodes reachable from level 1 are in level 2 and so on. .

* Can find the length with path
* Path : All nodes passed through to reach a destination
* Considers everything others than source as destination
- Considers each path as unique

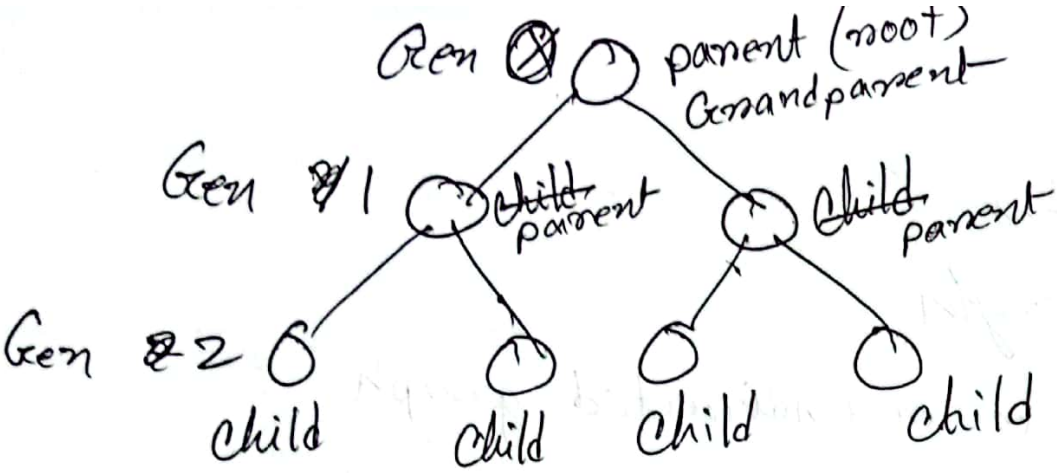


Input

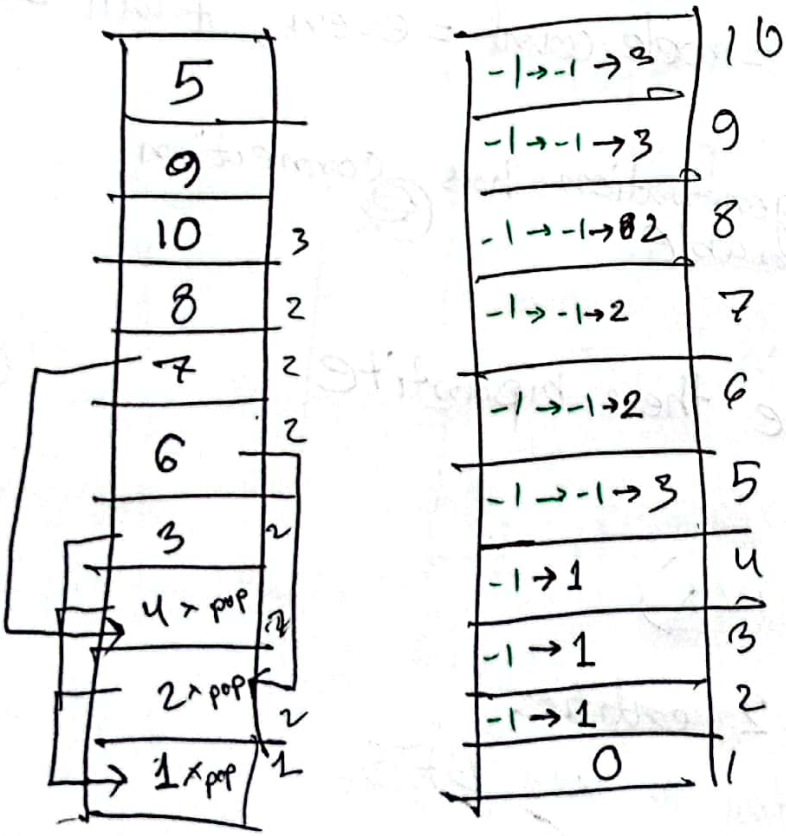
10	13
1	2
1	4
1	3
2	6
4	7
3	8
3	7
6	10
9	10
9	7
7	8
8	5
10	5

Adjacency list

1 → 2, 4, 3
2 → ~~8~~ 1, 6
3 → ~~8~~ ~~7~~ 1 8 7
4 → ~~7~~ 1 7
5 → ~~3~~ 10
6 → ~~10~~ 2 10
7 → ~~8~~ 4 3 9 8
8 → ~~5~~ 3 7 5
9 → 10 7
10 → ~~5~~ 6 9 5



- Find Generation
- Same generation will not have multiple parents



Queue

*BFS solves :-

- Finds shortest path
- Finds distance length
- Cycle detection in an undirected graph
  - ↳ if edge count reduces, there is a cycle (Learn)
- Bi-colorable
  - ↳ Cycle ~~→ edge count = node count in cycle~~
    - ↳ if edge count = node count = even, it will be bicolorable
    - ↳ if node in same generation has connection then not bicolorable
- Bi-partition
  - ↳ if bicolorable then bipartite

- Commandos (Light OJ, UVA)
- Any problem involving 2 entities
  - ↳ try to use BFS

* MEGA MEAT MAN
* Farthest node in a tree — (1) and (11)
* Going Together

Algorithm:

```
0. BFS(source)
1.     Declare a queue
2.     Enque @ source
3.     level[source] = 0
4.     while queue is not empty
5.     |       u = deque
6.     |       remove u from queue
7.     |       For all adjacent nodes of u: // for adjacency
8.     |       |     v = adjecent node of u // v is child, u is parent
9.     |       |     if level[v] = -1
10.    |       |     |     level[v] = level[u] + 1
11.    |       |     |     Enque v
12.    |       |     end if
13.    |       end for
14.    end while
```

Complexity: O(node + edge) = O(n + m)

Code:

```
0. void
   int level[12];

0. void BFS(int source){
1.     queue <int> Q;
2.     Q.push(source);
3.     level[source] = 0;
4.     while(!Q.empty()){
5.         int u = Q.front;
6.         Q.pop();
7.         for(int j=0; j<Node_Vec[u].size(); j++){
8.             int v = Node_Vec[u][j];
9.             if(level[v] == -1){
               level[v] = level[u]+1;
10             Q.push(v); }
11.        }
       }
   }
```

C-10 (W-9) 04/03/24

Infix to Postfix using Stack

A * (B + C) - (D / E)

Pre: ~~A * x - y~~ A* (+BC) - (/DE)

= A * x - y

= * Ax - y

= z - y

= - zy

= - * Axy

= - * A + BC / DE

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

= A* (BC +) - (DE /)

= A* x - y

= Ax * - y

= z - y

= zy -

= Ax * y -

= ABC + * DE / -

A + B + C

+ AB + C

= + + ABC

* Infix to Postfix :

$A*(B+C)-(D/E)$

Symbol scanned	Stack	Postfix
A		A
*	*	A
(	*, (↓Top	A
B	*, (	AB
+	*, (, +	AB
C	*, (, + ↓Top	ABC
)	*	ABC+ → lower precedence
-	-	ABC+* ABC+**
(	-, (	ABC+*
D	-, (,	ABC+*D
/	-, (, /	ABC+*DE
E	-, (, /	ABC+*DE/
)	-	ABC+*DE/-

Symbol Table

b | For negative numbers: convert $-x$ to $(-1 * x)$

options:

- operators
- operands
- brackets → opening / closed

Precedence

parantheses
^
*, /
+, -

Conversion: → write for prefix

1) If symbol = operand, add to postfix
2) If symbol = '(', push to stack
3) If symbol = operator, check top of stack
   i) (i) if precedence (symbol) ≥ precedence (stack top), push symbol to stack.
   (ii) Otherwise, pop one by one from stack (operators) and add to post fix until '(', is ~~it~~ add to 'postfix'
4) If symbol = ')', pop one by one,

(~~Otherwise~~ (Other solutions are available)

$(A + B / C) * (D + E) - F) = A\ BC/DE+*F-$

Symbol Scan	Stack	Postfix expression
(	(	
A	(	A ✓
+	(, +	AB ✓
B	(, +	AB ✓
/	(, +, /	AB ✓
C	(, +, /	ABC ✓
*	(, +, *,	ABC/
(	(, +, *, (	ABC/
D	(, +, *, (	ABC/D
+	(, +, *, (, +	ABC/D
E	(, +, *, (, +	ABC/DE
)	(, +, *	ABC/DE+
−	(, +, −	ABC/DE+*
F	(, +, −	ABC/DE+*F
)	—	ABC/DE+*F−

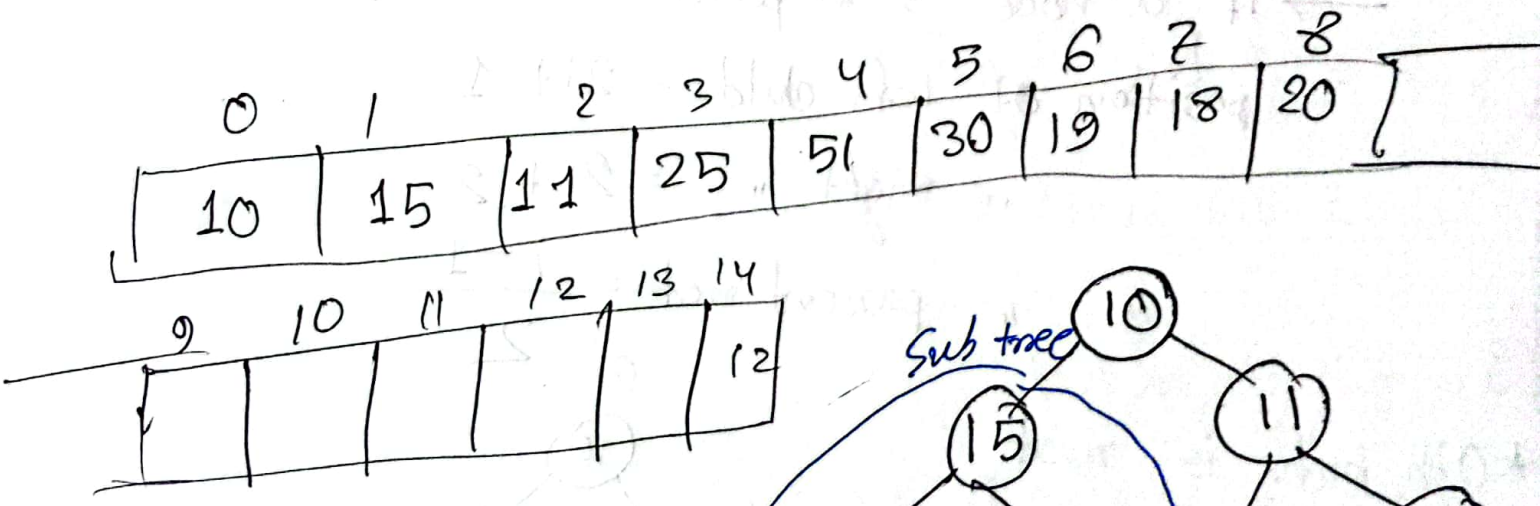
C-11 (W-10)

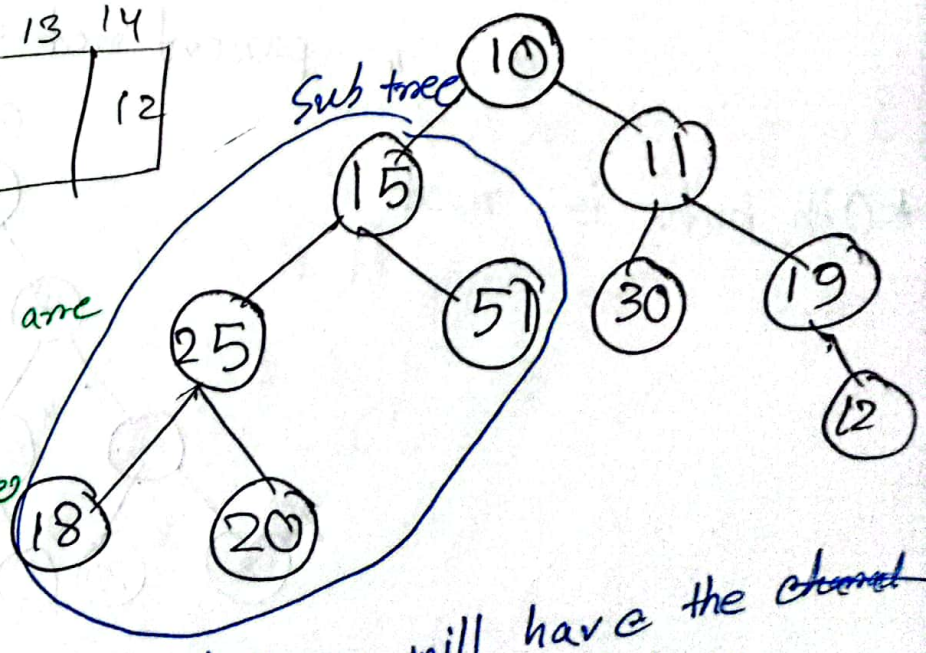
Tree

(Quiz-4 Syllabus Start)

24/06/24

* A tree is a finite collection of nodes that reflects one to many relationship among the nodes. An ordered tree has a specially designated node called root node. The nodes of a level are connected to the nodes of the upper level. The node that has no child is called leaf node. A node with a child node is called parent node.





* All nodes non-leaf nodes are parent nodes
* Basic tree = m-ary tree
* All sub trees will have the characteristics of parent tree

*Binary Tree: A finite set of nodes with a root node and every node has at most 2 child.

In a binary tree, // For Array

→ if no. of levels = k then,

maximum number of nodes in tree, $n = 2^k - 1$

→ if maximum no. nodes = n then,

no. of levels in tree, $k = \lceil \log_2(n+1) \rceil$

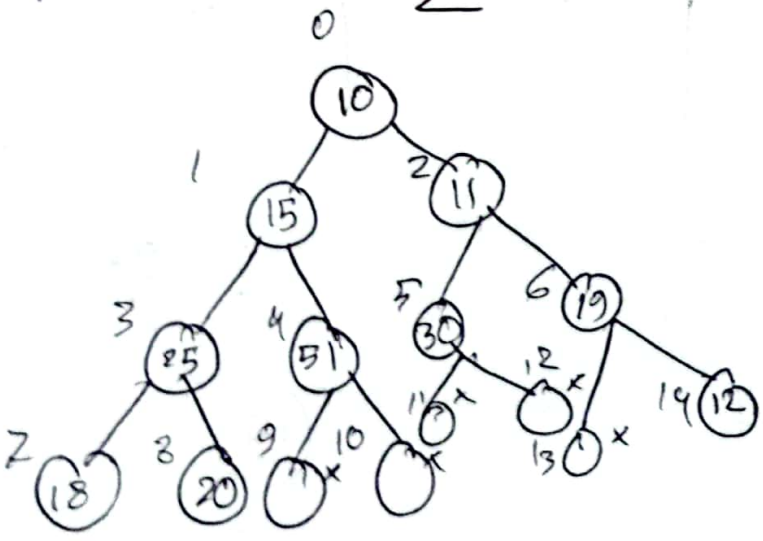
→ if a node is in position = i then,

position of left child = $2i + 1$

" " right " = $2i + 2$

" " parent node = $\frac{i-1}{2}$

*0th index is root



*Doubly linked list can also make binary tree

* Tree Traversal Technique:

(a) Pre - Order

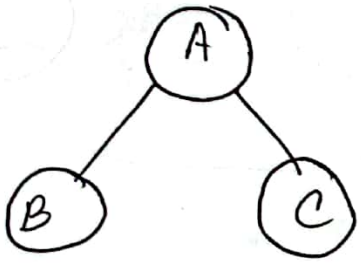
root
left
right

(b) In - Order

left
root
right

(c) Post - Order

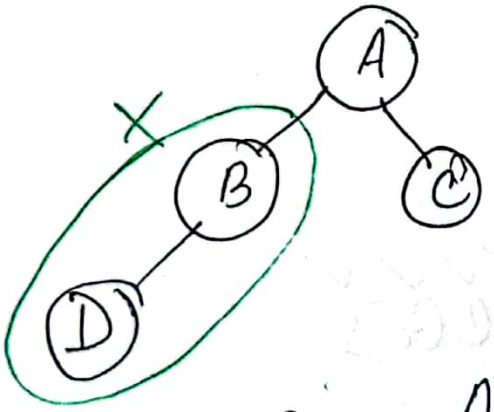
left
right
root



Pre : A B C

In : B A C

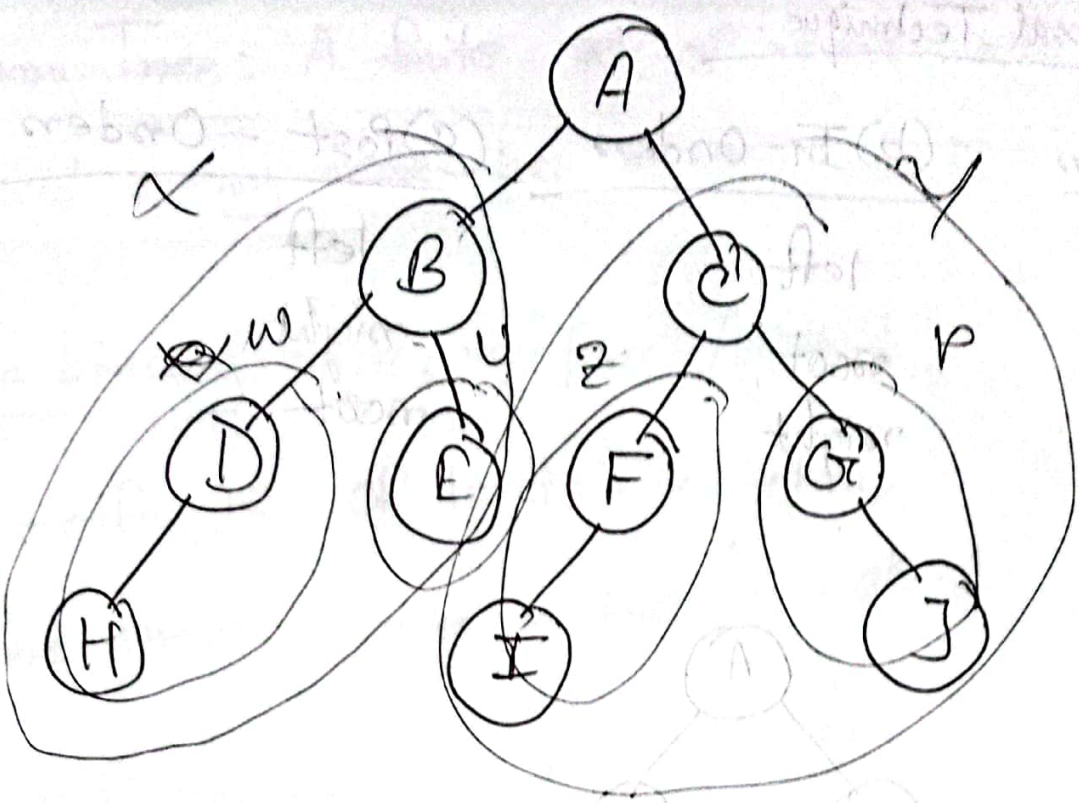
Post : B C A



Pre-order : A X C = A B D C

In - order :

Post - order :



Pre : A X Y

= A B W U C Z P

= A B D H E C F I G J

~~Post~~ In : X A Y

= W B U A Z C P

= H D B E A I F C G J

Post : X Y A = W U B Z P C A

= H D B E B I F J G C A

* Algorithm:

```
0. Pre-order ( temp):
1.      if temp ≠ NULL :
2.          print temp → data //
3.          preorder (temp → left) // print in in-order
4.          pre-order(temp → right) // print in post order
5.      end if
```

* For array implementation we send index
* Root is printed directed

Complexity: $O(n) = \log_2(node)$

* Categories of Binary Tree:

~~(a) Rooted Binary~~

(a) Rooted Binary Tree:

Every binary tree is a rooted binary tree.

(b) Full Binary Tree:

A binary tree in which every node has either

① or 2 nodes.

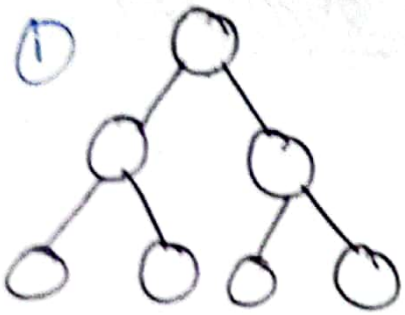
(c) Perfect Binary Tree:

A binary tree in which all interior nodes have 2 children all leaves are in the same level.

(d) Complete Binary Tree:

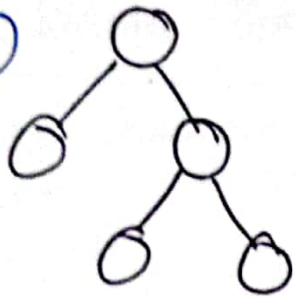
A binary tree in which every level except possibly the last is completely filled and nodes in the last level are as far LEFT as possible

(1)



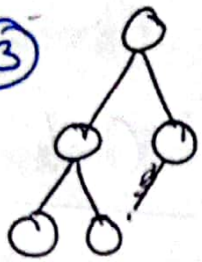
- Rooted
- Perfect
- Full
- Complete

(2)



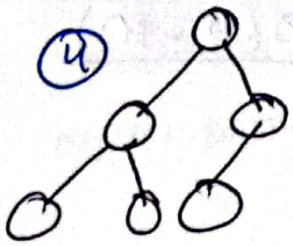
- Rooted
- Full

(3)



- Rooted
- Full
- Complete

(4)



- Rooted
- Complete

L-9(W-10)

25/06/24

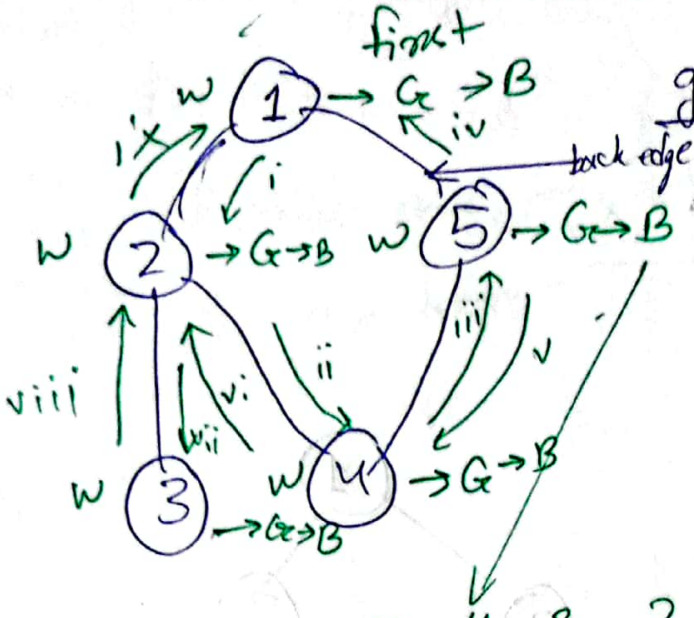
Online

See question

```
for (i = 1; i <= node_no; i++)
{
    if (level[i] == -1)
    {
        BFS(i);
        cout << i << endl;
        k++;
    }
}
```

HEAVILY Depends on user input ← DFS (Depth First Search)

↳ Use recursion to reach the deepest point first



graph input

5	5
1	2
2	4
4	5
5	1

* Going back prev node depends on stack memory. NOT EDGE.

5 4 3 2 1

*DFS DOES NOT give shortest path

~~* if back edge :~~

Three Colors:

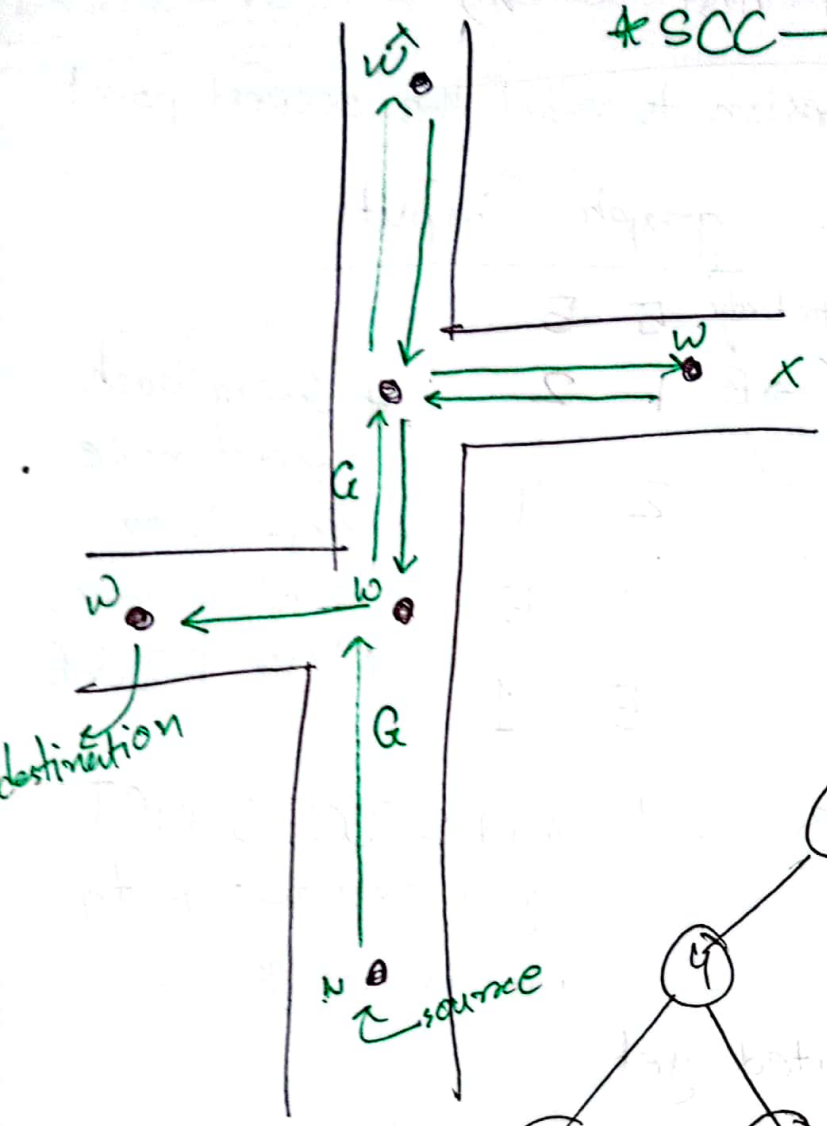
white = node not visited yet

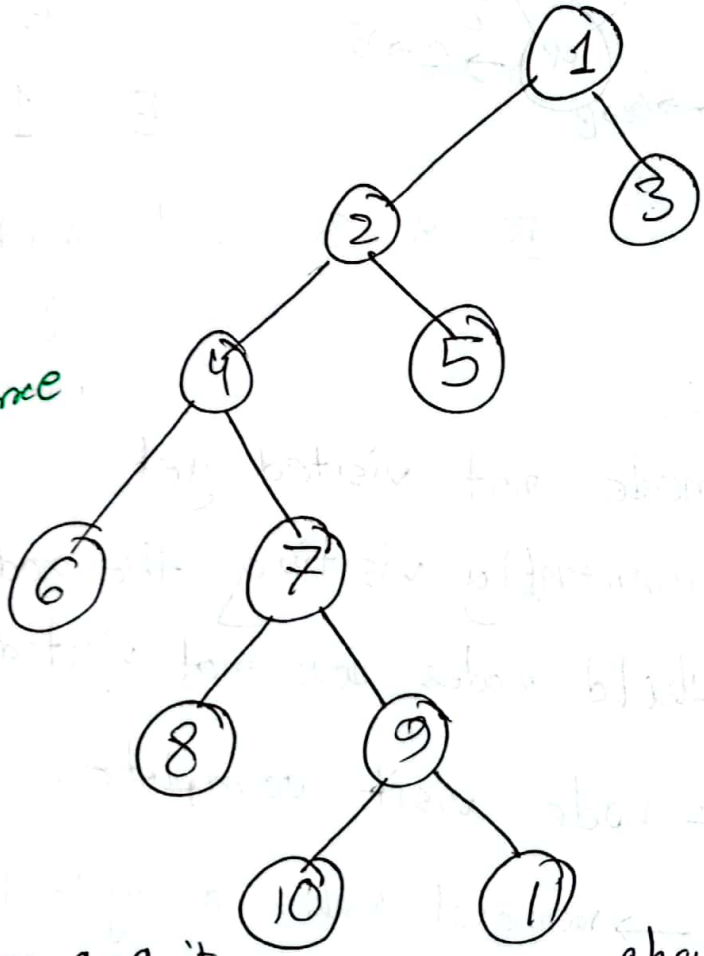
gray = currently visiting the node but all the child nodes are not visited yet

black = node visit complete.

* back edge → removal makes a cycle-less ~~tree~~. graph/tree
* tree edge.

* SCC → Used to find number of cycles





* graph traversal
* cycle finding → when g-g it has cycle
* connected/disconnected graph finding → check after DFS, check if any has white
* count number of components in a graph → Number of times comp BFS/DFS is run
* Topological sort

* Topological Sort:

In a directed dependent graph, it will give a sorted step.

→ Multiple solutions are possible

→ Run DFS with the least in order.

DFS Algorithm:

```
0. DFS(u) // initial data
1.  color[u] = gray // set it to gray
2.  For all adjacent edges of u: /* same loop as BFS
3.      v = adjacent node of u */
4.      if color[v] = white :
5.          DFS(v) // recursion
6.      end if
7.  end For
8.  Color[u] = black // when done make it black
```

Complexity : O(node + edge)

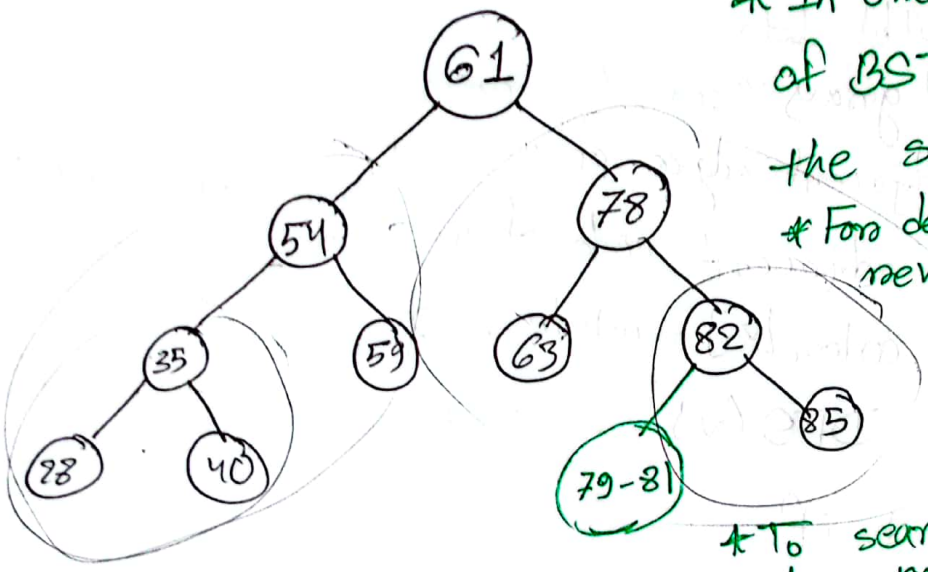
C-12 (w-11)

01/07/24

Binary Search Tree (BST)

A binary search tree is a binary tree where all the node values of the left sub tree are smaller than the node value of the root and all the nodes values of the right sub tree are greater than the node value of the node

* In order traversal of BST sorts the set
* For descending order, reverse the tree



* To search, we precede algorithm then use if switch to check

Inorder: X 61 Y

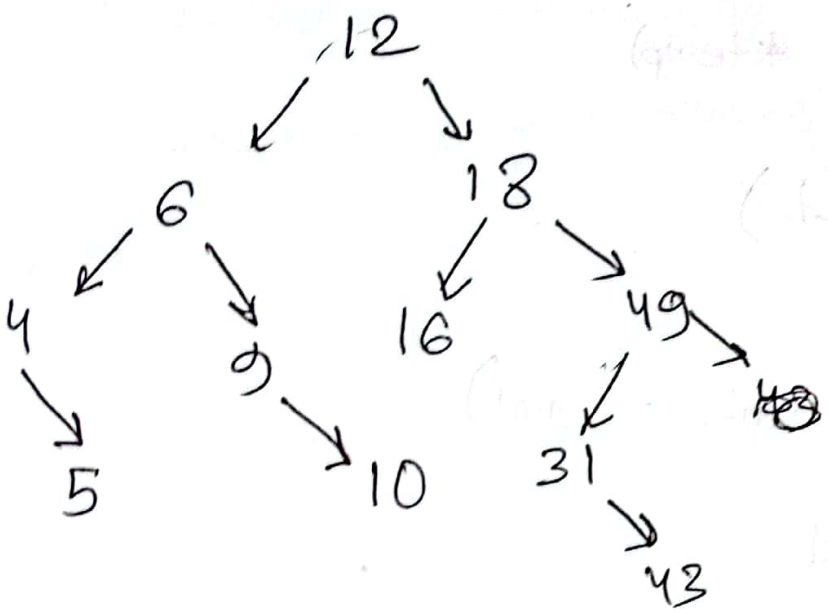
= W 54 59 61 63 78 V

= 28 35 40 54 59 61 63 78 82 85

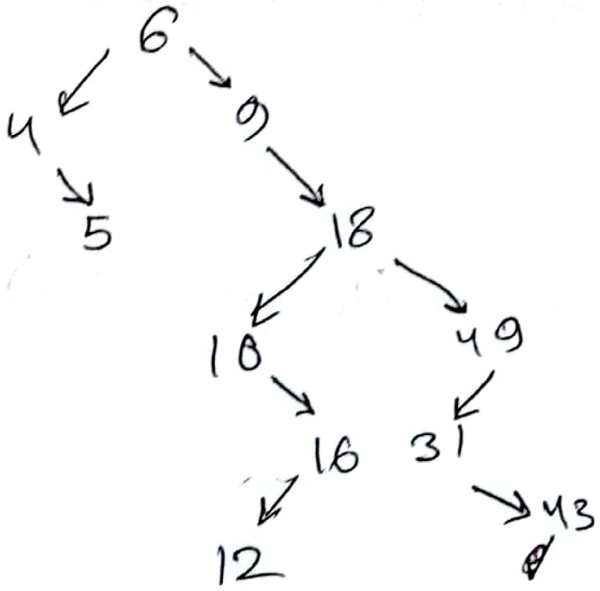
Search Code :

```
int val, flag = 0;

void pre_order(node *temp)
{
    if (temp != NULL)
    {
        if (temp -> data == val)
        {
            flag = 1;
        }
        else if (temp -> data > val)
        {
            pre_order (temp -> left) // For greatest ignore this
        }
        else
        {
            pre_order (temp -> right) // For smallest ignore this
        }
    }
}
```

12, 6, 9, 18, 4, 10, 5, 16, 49, 31, 43



Now 12 at last,



Binary Search Tree Code : → Complexity : $O(n) = n\log_2 n$

```
struct node
{
    int data;
    node * left;
    node *right;
};
node * root = NULL

void BST insert (int val)
{
    node *temp;
    temp = new node();
    temp -> dat = val;
    temp -> left = NULL;
    temp -> right = NULL;
    if(root == NULL){root = temp;}
    else
    {   node *curr_node = root;
        node * prev_node = root;
        while (curr_node != NULL)
        {
            prev_node = curr_node;
            if(curr_node -> data > val){curr_node -> left;}
            else{ curr_node = curr_node -> right; }
        }
        if (prev_node -> data > val){prev_node -> left = temp;}
        else{prev_node -> right = temp;}
    }
```

* predecessor
* succesor

C-13 (w-12) *Ostrich Algorithm 09/07/24

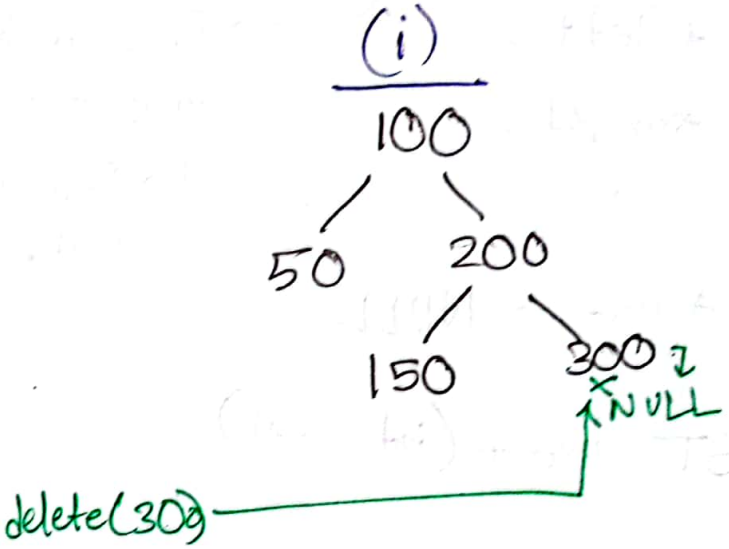
Delete From a Binary Search Tree

Tree cases:

(i) leaf node

(ii) node with one child

(iii) node with two child



```
void delete_node(int val)
{
    node *curr_node = root;
    node * prev = root;
    while (curr_node != NULL)
    {
        if (curr_node -> data == val){
            break;
        }
        prev = curr_node;
        if (curr_node->data < val) {
            curr_node = curr_node -> right;
        }
        else{
            curr_node = curr_node -> left;
        }
    }
    if (curr_node == NULL) {
        return;
    }else {
        if ((curr_node -> right == NULL) && (curr_node -> left
            == NULL))
        {
            if(prev -> right == curr_node)
            {
                prev -> right = NULL;
            }else {
                prev -> left = NULL;
            }
```

Does not activate until value is found

used to navigate

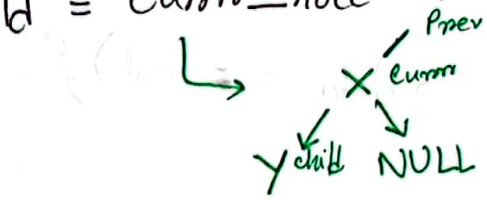
find value

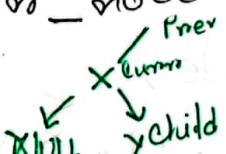
If not found

case (i) On Leaf

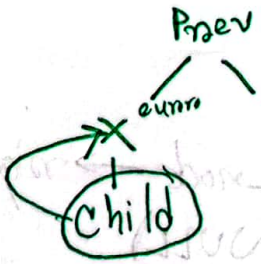
```
else if (curr_node -> right == NULL || curr_node -> left == NULL)
{
    node *child; // temp node
    if (curr_node -> right == NULL)
    {
        child = curr_node -> left;
    }
    else
    {
        child = curr_node -> right;
    }
    if (prev -> right == curr_node) // Find curr wrt prev
    {
        prev -> right = child;
    }
    else
    {
        prev -> left = child;
    }
}
```

Set child





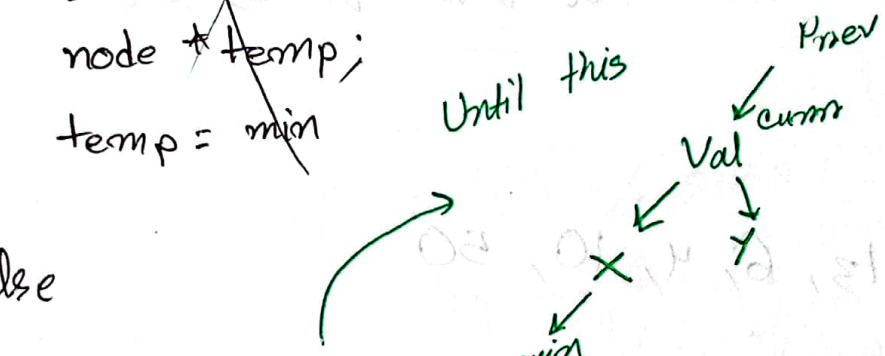
case (i) → one child.



```
else
{
    node *min = curr_node;
    prev = curr_node;
    while (min->left != NULL)
    {
        min = min->left;
    }
    node *temp;
    temp = min
```

* Find Do the same with ~~right~~ max value.

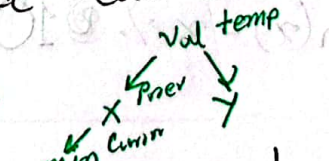
Until this

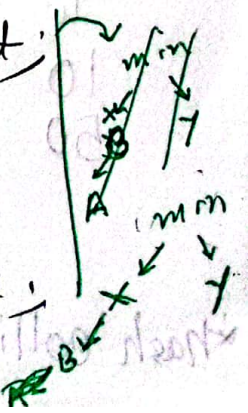



```
else
{
    node *temp = curr_node;
    prev = curr_node;
    curr_node = curr_node->right;
    while (curr_node->left != NULL)
    {
        prev = curr_node;
        curr_node = curr_node->left;
    }
    temp->data = curr_node->data; // swaps Val with the min data
    if (prev->left == curr_node)
    {
        prev->left = curr_node->right;
    }
    else
    {
        prev->right = cur_node->right;
    }
```

Finding minimum of

case (iii) → two children





C-14 (W-13) 15/07/24

Hashing (10 from here)

~~TOO~~

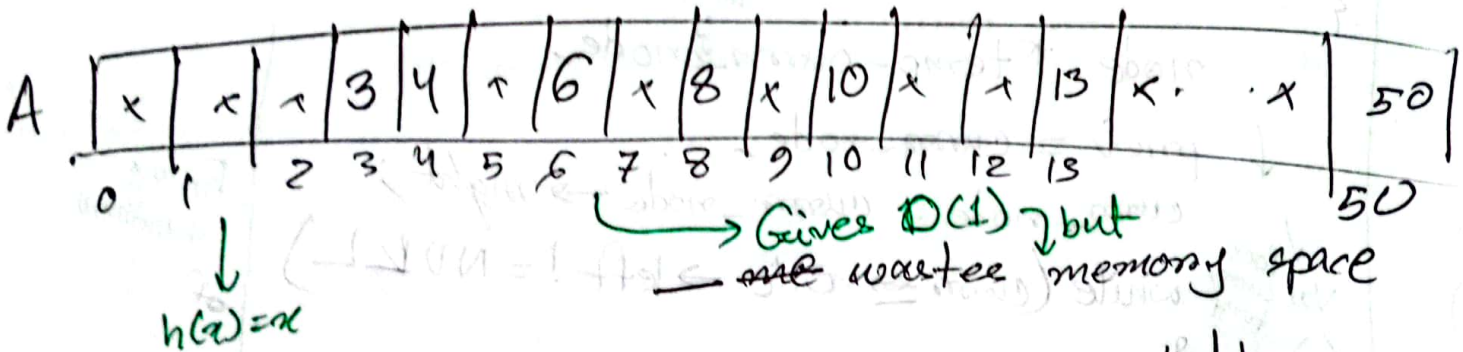
- Different from cryptography. Hashing is more involved with searching.
- Does not give O(1) but does work better than O(n) and O(log₂n)

Linear search — O(n)

Binary Search — O(log₂n)

We want a search method that requires O(1) time

keys = 8, 3, 13, 6, 4, 10, 50



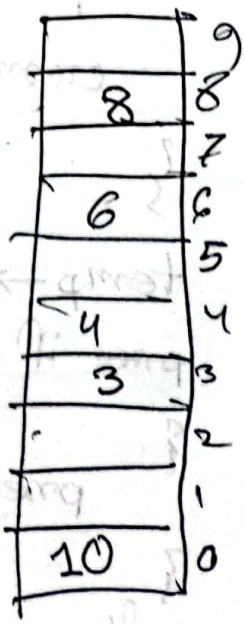
key space

8
3
13
6
4
10
50

hashing function

$h(x) = x \% 10$

hash table



*hash collision : conflict for space.

hash collision

- linear probing
- quadratic probing
- separate chaining

Linear Probing

$h(x) = x \,\%\, size$

$h'(x) = [h(x) + f(i)] \,\%\, size$

$f(i) = i \;;\; i = 0, 1, 2 \ldots$

* search = $A[x] \rightarrow A[x+1] \rightarrow \ldots$

$h(13) = 13 \,\%\, 10 = 3 \rightarrow$ collision

$\therefore h'(13) = (13 + 1) \,\%\, 10 = 4 \rightarrow$ collision

$\therefore h'(14) = (13 + 2) \,\%\, 10 = 5 \rightarrow$ free

Quadratic Probing

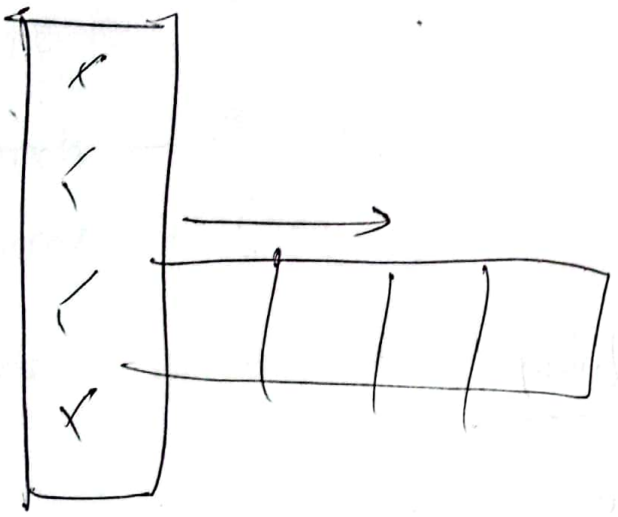
$h(x) = x \,\%\, size$

$h'(x) = [h(x) + f(i)] \,\%\, size$

$f(i) = i^2 \;;\; i = 0, 1, 2 \ldots$

~~Sepa~~

Separate chaining

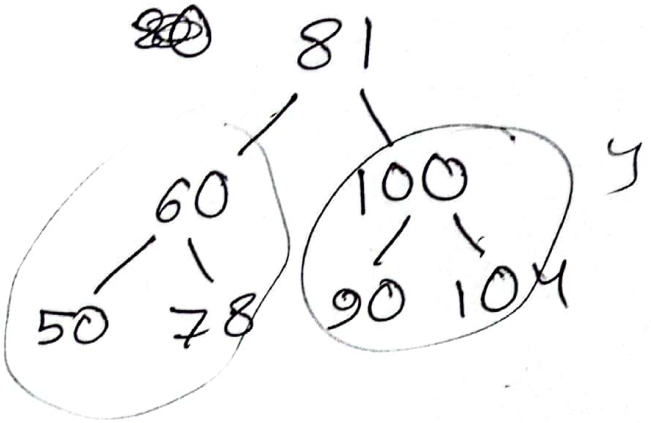


* 2D vector

#Coalescent Probing

17/09/24

L-10(ω-13)



Pre: ~~x 81 y~~ ~~81 x y~~ ~~x 81~~

= ~~81 60 ω v 100 p q~~

= ~~81 60 50~~ 81 60 50 ~~75~~ 100 90 104

In: x 81 y

= ω 60 v 81 p 100 q

= 50 60 ~~75~~ 81 90 100 104

Post: ~~ω~~ xy 81

= ω v 60 p q 100 81

= 50 ~~75~~ 60 90 104 100 81