Assignment :- 1.

(1) What is data organization, and why is it important in computer science ?

→ Data organization refers to the way data is structured, arranged, and stored in a system so that it can be easily accessed, managed, and processed.

→ In computer science, this includes how data is organi-zed in files, databases, memory, or even in more abstract structures like arrays, trees, and graphs.

- Importance of Data Organization in Computer Science:

(1) Efficient Data Retrieval :- Organized data can be accessed faster.

→ for eg: a well structured database allows quick querying and reduces the time taken to retrieve relevant information.

(2) Data Management : Proper data organization helps with better management of large amounts of data

→ It supports efficient storage, modification, and deletion of data.

(3) Memory Optimization :- Organizing data optimally allows for more efficient use of memory.

→ Data structures like arrays, linked lists, or hash tables help in using memory resources effiectively.

Scanned with OKEN Scanner

(4) Scalability:- As data grows, a good organizational strategy ensures that the system can handle larger datasets without a drop in performance.

(5) Improved Processing:- Algorithms that operate on well-organized data can perform better.
→ Sorting, searching, and other operations become more efficient with organized data structures.

(6) Maintainability:- Organized data is easier to maintain, update, and expand over time, which is crucial for long-term software projects.

(2) Explain the concept of arrays. How are arrays used in data organization?

→ An array is a data structure used to store multiple elements of the same data type in a contiguous block of memory.

→ Each element in the array can be accessed directly using an index.

→ Arrays are fundamental in computer science for organizing and storing data in a linear sequence, allowing for efficient access and manipulation of the data.

Scanned with OKEN Scanner

- characteristics of Arrays :-

→ Fixed size :- Arrays have a fixed size, meaning the number of elements is defined when the array is created.

→ Same Datatype :- All elements within an array must be of the same datatype, such as integers, floats, ~~and~~ or strings.

→ Indexing :- Arrays use zero-based indexing in most programming languages, meaning the first element is accessed with index 0, the second with index 1, and so on

→ Contiguous Memory Allocation :- Elements are stored in adjacent memory locations, which allows fast access to individual elements via indexing.

- Arrays are used in Data Organization :-

→ Data storage :- Arrays allow the storage of a large number of elements in a structured and indexed manner, which is essential for organizing data such as lists of values, records, or collections.

→ Efficient Access :- Since arrays allow direct access to elements using an index, retrieving or modifying any element is very efficient.

Scanned with OKEN Scanner

→ Iteration and Traversal :- Arrays allow easy iteration through elements using loops, which is essential in various algorithms like searching sorting, or filtering data.

→ Multidimensional Data storage :- Arrays can be extended into multiple dimensions to organize complex data structures like matrices, tables, or grids.

→ This is useful in fields like image processing or game development.

- Example applications of Arrays :-

→ Sorting algorithms :- Arrays are used to implement sorting algorithms like quicksort, mergesort, and bubblesort.

→ Data storage structures: Arrays are used to implement higher-level data structures like stacks, queues, and heaps.

(3) What are the common operations performed on data structures ?

- The common operations performed on Data structures are :-

(1) Create :- It results in reserving memory for program elements.

(2) Destroy :- It destroys memory space allocated for specified data structure.

(3) Selection :- It deals with accessing a particular data within a data structure.

(4) Updation :- It updates or modifies the data in the data structure.

(5) Searching :- It finds the presence of desired data item in the list of data items.

(6) Sorting :- It is a process of arranging all data items in a data structure in a particular order.

(7) Merging :- It is a process of combining the data items of two different sorted list into a single sorted list.

(8) Splitting :- It is a process of partitioning single list to multiple list.

(9) Traversal :- It is a process of visiting each and every node of a list in systematic manner.

(4) Define and explain primitive and non-primitive data types.

- Primitive Data types :-

→

Page No:
Date:

→ Primitive data types are the basic, predefined data types provided by the programming language.

→ These types are the building blocks for data manipulation.

→ They are simple, have a fixed size, and represent fundamental data like numbers, characters, and booleans.

→ eg :- Integer, floating type, character, Boolean, Void etc.

- Characteristics of Primitive Data types :-

→ They are defined by the language and usually have a fixed size.

→ They directly store the value assigned to them.

→ Operations on primitive types are generally fast and performed at the machine level.

→ They are not derived from any other types.

eg :-
```
int num = 10;
float price = 99.97;
char letter = 'Z';
```

Page No:
Date:

- Non- Primitive data-types:-

→ Non- primitive data types, also known as reference types or derived types, are more complex data types created using primitive types.

→ They are defined by the programmer or language libraries and are used to store collections of values or complex data.

→ example :- Arrays, structures, Unions, pointers, strings, Enumerations (enum), etc;

- Characteristics of Non- Primitive Data Types :-

→ They can store multiple values or complex data.

→ The size of non-primitive types can vary based on the data ~~to~~ they hold.

→ They typically reference memory locations.

→ They can be defined by the programmer, allowing more flexibility.

→ They are often used to create data structures like linked lists, stacks, queues, trees, etc.

Scanned with OKEN Scanner

Page No:
Date:

(5) Explain significance of data structures.

→ Organization and management :- Data structures organize, manage, and store data efficiently.

→ They ensure that data is relevant to the task and easy to use.

→ For example, lists and arrays are well-known data structures.

→ Efficient Algorithms: Data structures are essential for creating fast and powerful algorithms.

→ They help manage and organize data, reduce complexity, and increase efficiency.

→ Memory Optimization: Properly chosen data structures optimize memory usage.

→ For instance, using arrays and linked lists minimizes memory wastages, especially with large datasets.

→ logical Representation :- Data structures provide a logical or mathematical representation of data.

→ They define how data is stored and manipulated within a program.

Scanned with OKEN Scanner

(6) Define Data structure. Explain linear and non-linear data structure with example.

→ A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently.

→ It encompasses both the logical or mathematical representation of data and its implementation in a computer program.

• Linear Data structure:-

→ In a linear Data structure, data elements are arranged sequentially or linearly.

→ Each element is attached to its previous and next adjacent elements.

→ Examples :- Array, stack, queue, linked list.

• Non linear Data structure :-

→ In non-linear data structures, data elements are not arranged sequentially.

→ They often have hierarchical relationships.

→ example :- Tree, Graph.

(7) Define time complexity ? Explain worst case and best case complexity with examples.

→ Time complexity refers to the number of operations needed to run an algorithm on large amounts of data.

• Worst case :-

→ In worst case, we calculate the upper bound on the running time of an algorithm.

→ This means we determine the maximum amount of time the algorithm requires to process all input values.

→ For example :- Consider the linear search algorithm.

→ The worst case occurs when the element to be searched is not present in the array.

→ In this scenario, the search function compares the target element with all elements in the array, resulting in an worst-case time complexity.

• Best Case :-

→ In the best-case, we calculate the lower bound on the running time of an algorithm.

→ This represents the minimum time required for the algorithm to complete when given the optimal input.

→ For linear search, the best case occurs when the target element is present at the first location.

→ In this case, the number of operations is constant, resulting in a best-case time complexity.

(8) Explain the concept of user-defined data types with an example in a programming language.

→ User defined data types allow programmers to create custom data types based on existing ones.

- Structure :- A structure is used to group different data types into a single type.

→ It provides a way to encapsulate related variables.

example :-
```
# include <stdio.h>
struct person {
    char company[50];
    int lifespan;
};
int main () {
Struct person person1;
strcpy (person1.company, "Geek");
person1.lifespan = 30;
printf ("Name: %s\n", person1.company);
printf (" Age : %d\n", person1.lifespan);
return 0;
}
```

output :

Name : Geeks

Age : 30.

- **Union** :- A union stores all its members in the same memory location, allowing only one number to contain data at a time.

example:-

```
#include <stdio.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main() {
    union Data data;
    data.i = 10;
    printf("Data.i : %d\n", data.i);
    data.f = 3.14;
    printf("Data.f : %f\n", data.f);
    strcpy(data.str, "Hello", c");
    printf("Data.str : %s\n", data.str);
    return 0;
}
```

Output :-

Data.i : 10

Data.f : 3.140000

Data.str : Hello , c

Page No:
Date:

- Enumeration (enum)

→ Enums allow creating custom data types with named integer constants. They enhance program redability.

example:-
```
#include <stdio.h>
enum cafes {
    Zeel_Art_cafe,
    Tea_Villa_Cafe,
    // ....
};
int main () {
    enum cafes mycafe = Tea_villa_cafe;
printf("My favorite cafe : %d \n", mycafe);
return 0;
}
```

Output : My favorite cafe : 1

(9) What is a linked list, and how does it differ from an array ?

- Linked list :-

→ A linked list is a linear data structure where elements (nodes) are chained together using pointers or references.

→ Nodes in a linked list may not be allocated sequentially in memory.

Scanned with OKEN Scanner

Page No:
Date:

- Each node contains two parts :
→ Data : The actual value or payload.
→ Next Pointer : A reference to the next node in the list.

→ Types of linked lists include singly linked lists, doubly linked lists, and circular linked lists.

- advantages :-

→ Efficient insertion and deletion at any point.

→ Space - efficient when the number of elements is dynamic or unknown.

→ Useful for implementing CPU scheduling algorithms.

- Arrays :-
→ An array is a data structure that stores elements of the same data type in contiguous memory locations.

→ Elements in an array have easily calculable addresses, allowing quick access by index.

- advantages :-

→ Random access to elements.

Scanned with OKEN Scanner

→ cache-friendly due to contiguous storage

(10) What is a stack, and how does it differ from an array?

• Stack:-

→ A stack is a linear data structure that operates on the Last In, first out (LIFO) principle.

→ Elements are inserted and removed only from one end of the list, called the top.

→ Common stack operations include push (insertion) and pop (deletion).

→ example :- Think of a stack of plates where you add or remove plates from the top.

→ Can be implemented using array.

• Arrays:-

→ An array is a collection of elements stored at contiguous memory locations.

→ Elements are accessed by their index.

→ Example :- Think of an array of numbered boxes where each box holds an item.

→ Arrays have a predetermined size.

→ Can be implemented using static memory allocation.

Assignment : 2.

(1) What is an array ? and how is it represented in memory ?

→ An array is a fundamental data structure that stores a collection of elements of the same type.

→ It allows efficient access to individual elements based on their position (index).

• Memory Allocation :-

→ When you declare an array, memory is allocated for its elements sequentially.

→ The elements are stored in contiguous memory locations.

→ for eg, if you declare an integer array int arr [5]; , memory is allocated for 5 integer elements in a row.

• Static Memory Allocation :-

→ In languages like c++, the array size is fixed at compile time.

→ This means that the memory for array elements is determined during compilation.

→ Static allocation ensures efficient access but may lead to wasted memory if the array size is too large or insufficient for the data

Page No:
Date:

- Dynamic memory allocation:-

→ In some cases, we don't know the array size beforehand.

→ Dynamic memory allocation allows flexibility.

→ It's preferred when the array size varies at run time.

(2) What is sparse matrix? write efficient vector representation of the following sparse matrix.

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3. \end{matrix}$$

→ Sparse matrix is a matrix which contains very few non-zero elements.

→ A sparse matrix can be represented by using two representation, those are :-

(1) Triplet representation (array representation)
(2) Linked Representation.

(1) Triplet Representation of the given sparse matrix.

	0	1	2
0	1	0	0
1	0	2	0
2	0	0	3

→

Rows	Columns	Values
3	3	3
0	0	1
1	1	2
2	2	3

$$\boxed{4 + 2xy + 3x^2y^2}$$

(2) Linked Representation of the given matrix.

Row number	column number	Value	Pointer to Next Node.

0	0	1	→	1	1	2	→	2	2	5		NULL.

(3) Represent the following polynomial equation using an array.

(a) $2x^2 + 5xy + y^2$

	0	1	2
0	0	0	1
1	0	5	0
2	2	0	0

→

Rows	Columns	Values.
0	2	1
1	1	5
2	0	2

(b) $x^2 + 3xy + y^2 + y - x$

	0	1	2
0	0	1	1
1	-1	3	0
2	1	0	0

→

Rows	Columns	Values
0	1	1
0	2	1
1	0	-1
1	1	3
2	0	1

(4) Write recursive solution for tower of Hanoi. How many moves require for transfering three discs?

→ Tower of Hanoi is a mathematical puzzle which consist of 3 poles and number of discs of different sizes.

→ Initially all the discs will be places in the single pole with the largest disc at the bottom and smaller on the top.

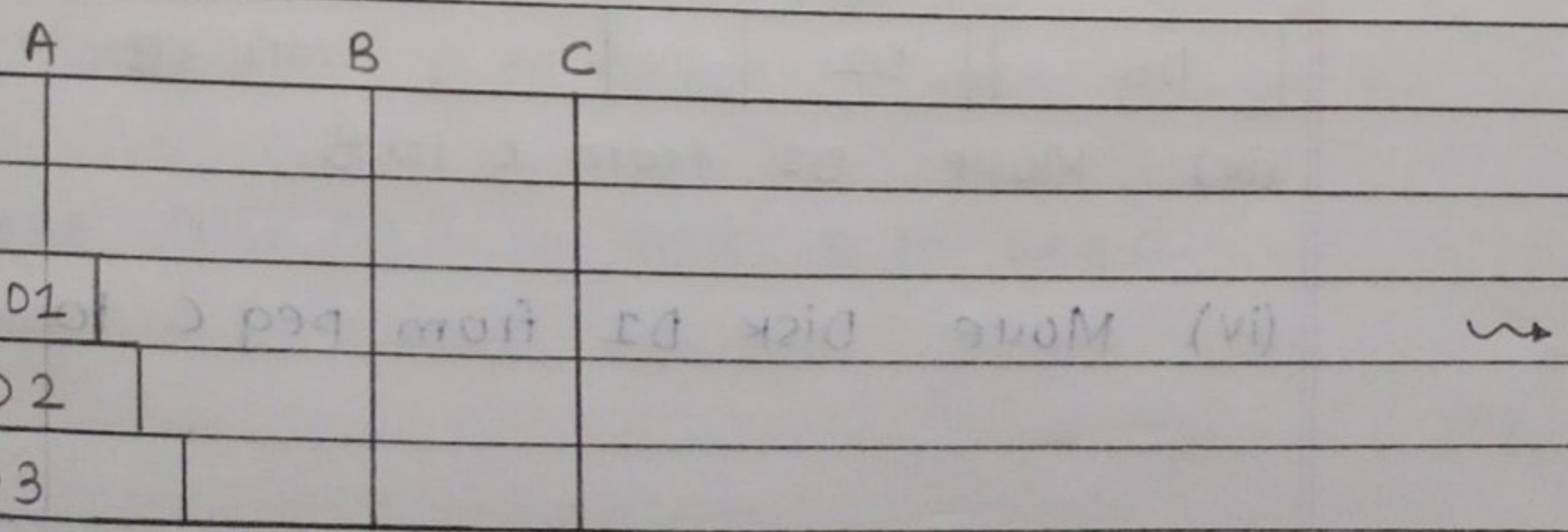
→ We need to move all the disc from the first pole to the third pole with the smallest disc at the top and the largest at the bottom.

→ Only one disc can be moved at a time.
→ larger disc cannot be placed on a smaller disc.

→ Only top disk on any peg may be moved to any other peg.

* → If there are three disks, then there will be seven movements of disks in transferring from peg A to peg C.

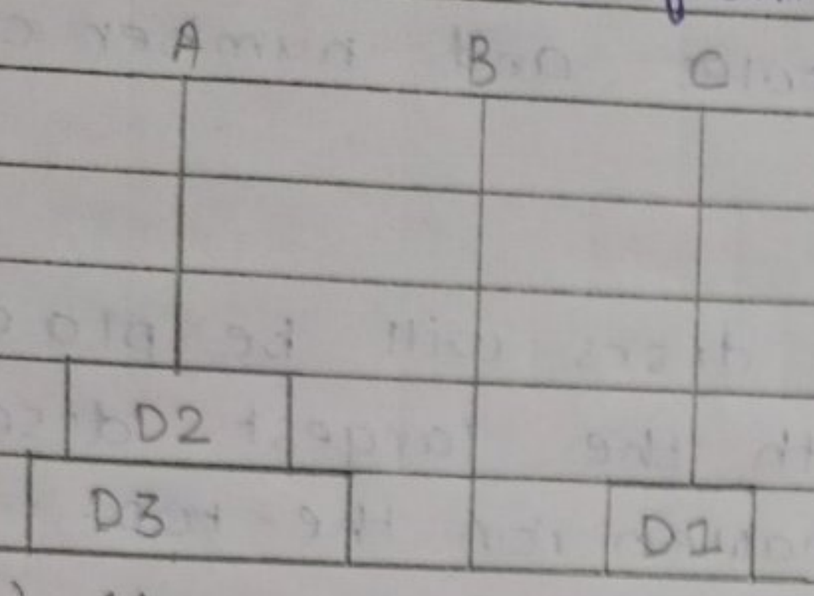
→ ~~(i) Move disk~~.



(i) Initial state.

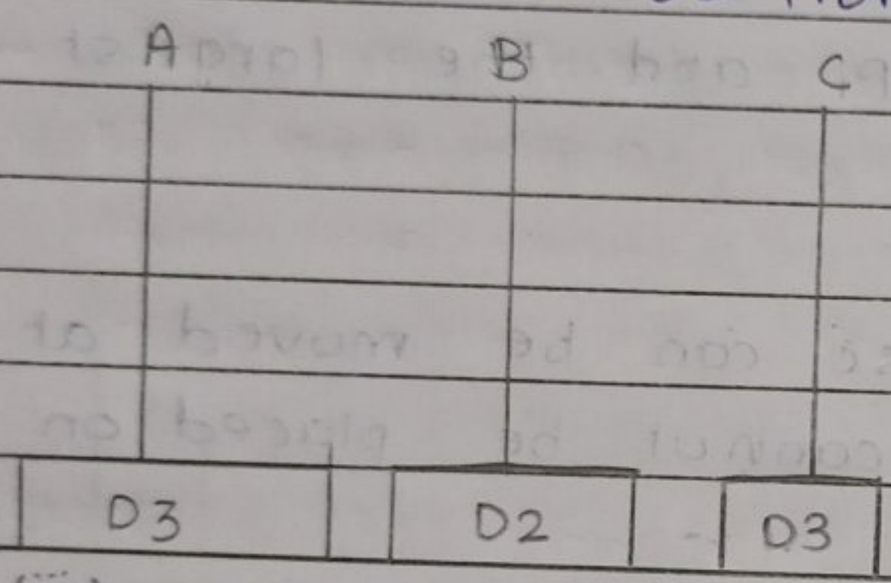
Disk = 3.

(i) Move disk 1 from peg A to peg C.



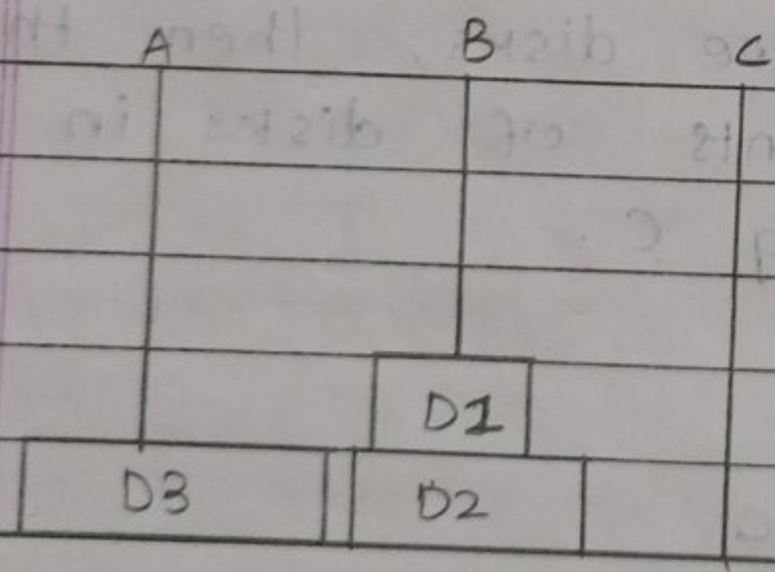
(ii) Move D1 from A to C.

(ii) Move Disk D1 from peg A to peg C



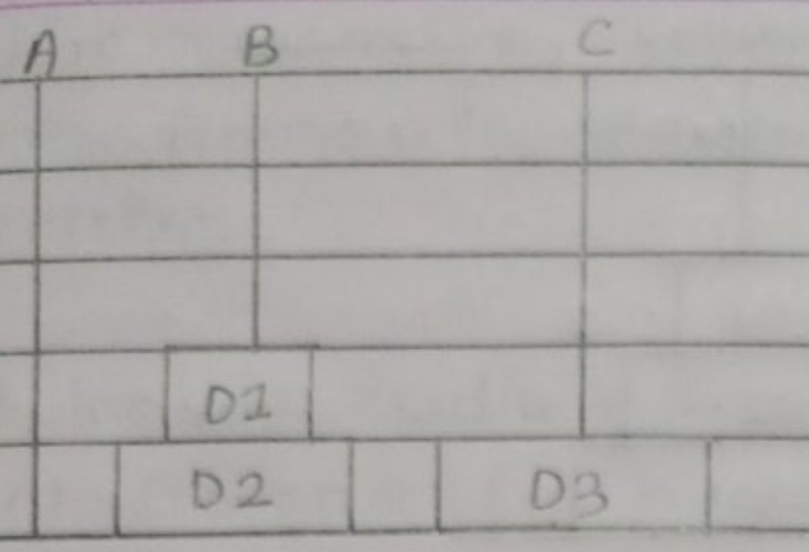
(iii) Move D2 from A to B

(iii) Move Disk D2 from pegA to pegB.



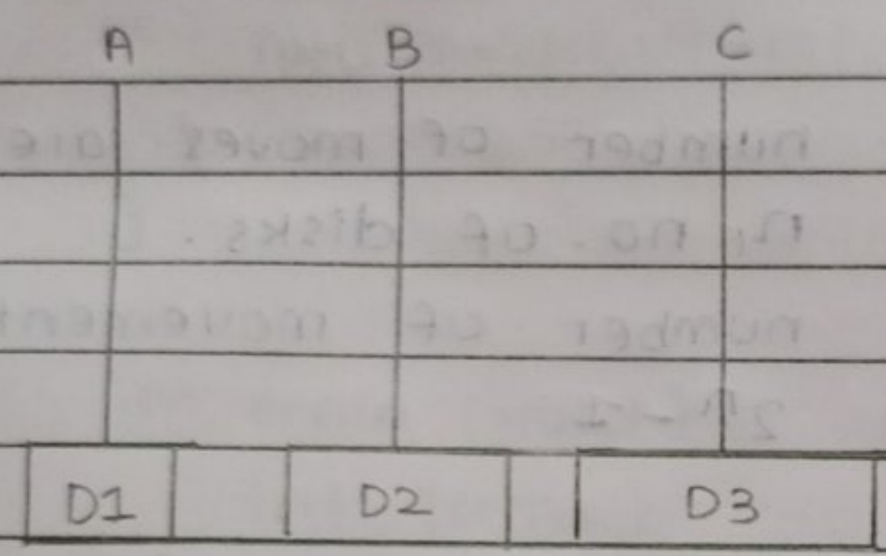
(iv) Move D1 from C to B

(iv) Move Disk D1 from peg C to peg B.



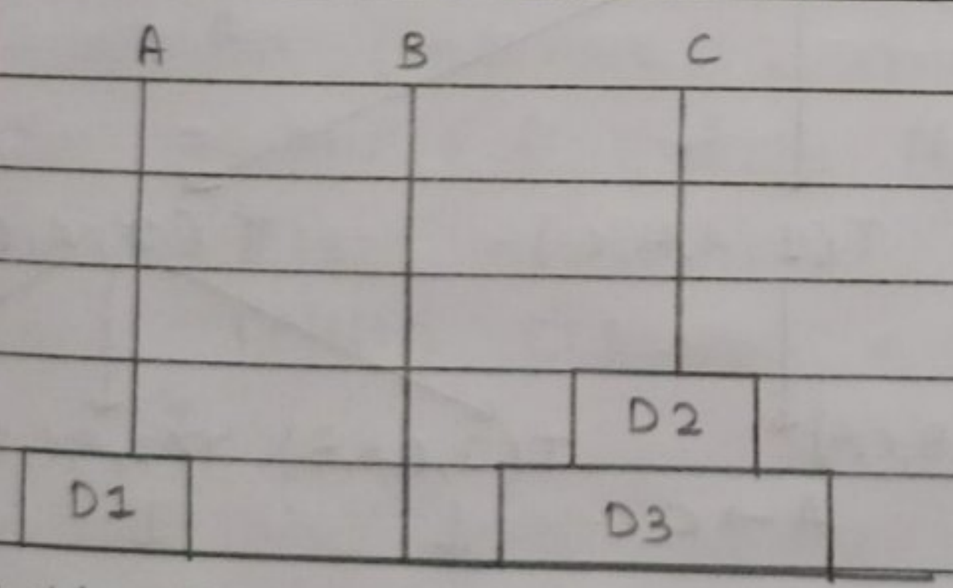
(V) Move D3 from A to C

(vi) Move Disk ~~3~~ D3 from peg A to peg C



(vi) Move D1 from B to A

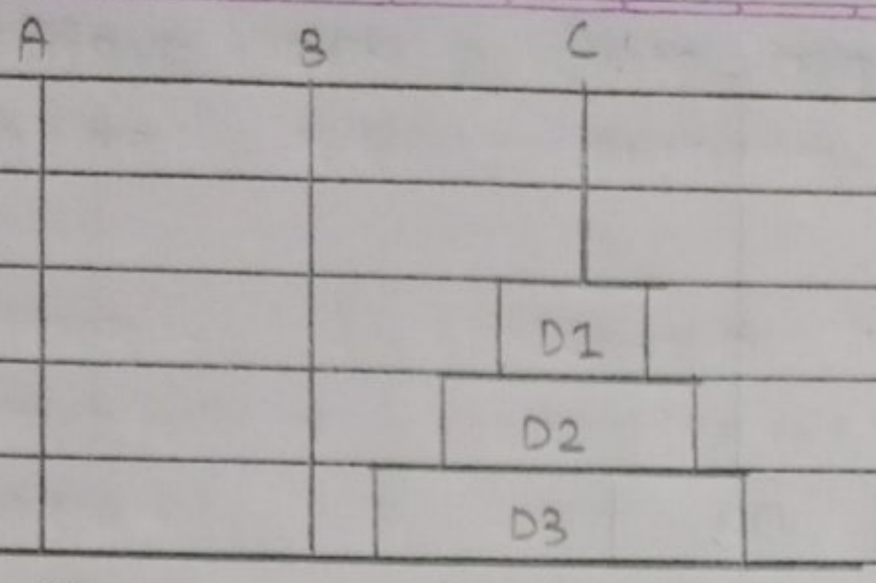
(vi) Move Disk D1 from peg B to peg A.



(vii) Move D2 from B to C

(vii) Move Disk D2 from peg B to peg C.

Scanned with OKEN Scanner



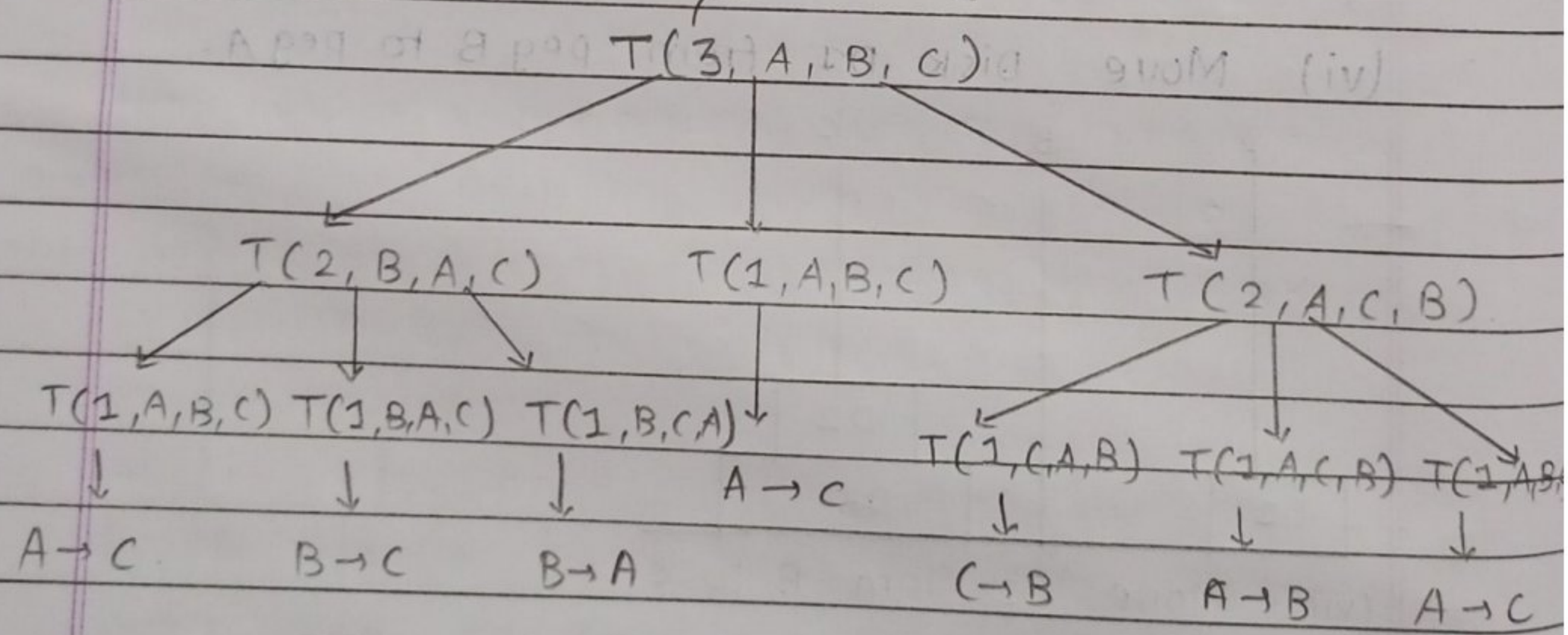
(viii) Move D1 from A to C.

(viii) Move Disk D1 from peg A to peg C.

→ So the Total number of moves are 7.
If there are n no. of disks.
then the total number of movement of disks will be $2^n - 1$.

Here Disks = 3 then $2^n - 1 = 2^3 - 1 = 8 - 1 = \boxed{7}$

[No. of Disks].



(5) Write ~~recursive~~ ~~solution~~ ~~for~~ C program to find the fibonacci sequence of n terms using recursion.

```
# include <stdio.h>
int fibonacci (int num) {
    if (num == 0 || num == 1) {
    return num;
  } else {
    return fibonacci (num - 1) + fibonacci (num - 2);
  }
}

int main (void) {
    int terms;
    printf (" Enter the no. of terms: ");
    scanf (" %d", &terms);

  for (int n = 0; n < terms; n++) {
    printf (" %d ", fibonacci (n));
  }
  return 0;
}.
```

Output :-

Enter the no. of terms : 10

0 1 1 2 3 5 8 13 21 34.

(6) Define stack. Write algorithms for PUSH, POP and PEEP operations of stack.

→ Stack :- A stack is a linear data structure that follows the Last-In-First-out (LIFO) principle.

→ Imagine a stack of plates: The last plate you put on top is the first one you'll remove.

→ Similarly, in a stack, the last element inserted is the first to be popped out.

Algorithms :-

(1) PUSH

→ Used to insert element into stack.

→ In push operation, we can add elements to the top of the stack, so before push operation, user must check the stack, it should not be a full.

→ If the stack is already full and when we try to add the elements then error occurs.

→ It is called " stack overflow" error

PUSH (S, N, TOP, VAL)

* stack is represented by vector S which contain N elements.

→ TOP is a pointer which points to the top element of the stack.

→ VAL is the element to be inserted.

[1] [check for stack overflow]
If TOP == N - 1 then
print "stack overflow"
exit.

[2] [increment TOP pointer]
TOP = TOP + 1

[3] [insert new element at TOP]
S[TOP] = VAL

[4] [FINISH]
EXIT.

(2) POP

→ Used to Delete element from stack.

→ In POP operation, we can delete or remove an elements from top of the stack, so before POP operation, user must check the stack, stack should not be a empty.

→ If the stack is empty, and we try to pop an element, then error occur. It is called "stack Underflow" error.

POP (S, TOP).

→ Stack is represented by vector S which contains N elements.

→ TOP is a pointer which points to the top element of the stack.

[1] [check for underflow]
If TOP < 0 then
print "stack underflow"
exit

[2] [copy the element from the top of stack]
VAL = S[TOP]

[3] [Decrement the TOP pointer]
TOP = TOP - 1

[4] [Return the deleted value]
Return VAL.

[5] [FINISH]
exit

(3) PEEP

→ Peep is an operation that returns the value of the ith element from the top of the stack without deleting it from the stack.

PEEP (S, TOP, I)

→ Stack is represented by vector S which contains N elements.

→ TOP is a pointer which points to the top element of the stack.

→ I is the position of the element want to access.

[1] [Check for underflow]

If TOP - i + 1 < 0 then
print " stack underflow"
exit.

[2] [Return ith element from the top of the stack]

return S[TOP - I + 1]

[3] [FINISH]

exit.

(7) Enlist and explain any 3 applications of stack in computer science with example.

→ Applications of stack :-

(1) Recursion.
(2) Evaluate Polish Notation.
(3) Stack Machine.
(4) Tower of Hanoi
(5) Reversing a list.

→ Applications of stack in computer science.

(1) Function calls.
(2) Expression Evaluation.
(3) Memory Management.

(1) Function Calls:-

→ Stack are used to keep track of return addresses during function calls.

→ When a function is called, its return address (the location in the program where execution should continue after the function completes) is pushed onto the stack.

example:-

→ Consider a recursive function like calculating the factorial of a number.

→ Each recursive call pushes its return address onto the stack, allowing the program to return to the correct location after completing each recursive cell.

(2) Expression evaluation:-

→ Stacks are essential for evaluating expressions in postfix notation (also known as Reverse polish notation).

→ In postfix notation, ~~operation~~ operators follow their operands, making it easier to compute expressions.

→ Example:-

→ To evaluate the expression "34+5," we use a stack.

→ We push operands (3 and 4) onto the stack, encounter the "+" operator, pop the top two elements (4 and 3), perform the addition, and push the result (7) back onto the stack.

→ Then, we encounter the "*" operator, pop 7 and 5, multiply them, and push the final result (35) onto the stack.

(3) Memory Management :-

→ Some operating systems and programming langua-ges use stacks for memory allocation and management.

→ Example :-

→ In a call stack, local variables and function parameters are allocated memory.

→ When a function exists, its stack frame is deallocated, freeing up memory.

→ This stack - based memory management ensures efficient use of resources.

(8) Write an algorithm for infix to postfix conversion.

→ Algorithm : Infix to Postfix conversion.

(1) Intialize :-

→ Create an empty stack is for operators.

→ Create an empty list postfix for the output expression.

(2) Read input expression:

→ Traverse the infix expression from left to right.

(3) Process each symbol in the expression:

→ If the symbol is an operand append it to the postfix list.

→ If the symbol is an operator:

→ While there is an operator at the top of the stack with greater precedence than the current operator or the top operator has equal precedence and the current operator is left associative, pop the top operator from the stack and append it to postfix.

→ If the symbol is an opening parenthesis '('
Push it onto the stack.

→ If the symbol is closing parenthesis ')'.
Pop operators from the stack and append them to postfix until an opening parenthesis '(' is found.

→ Discard the opening parenthesis.

(4) Pop remaining operators:

→ After traversing the entire infix expression, pop all remaining operators from the stack and append them to postfix.

(5) End :

→ The postfix list now contains the postfix expression.

example :-

A + B * C - D

→ Read A, append to postfix.
→ Read +, push onto stack
→ Read B, append to postfix
→ Read *, push onto stack
→ Read C, append to postfix.
→ Read -, pop * as it is higher precedence. and append to postfix, pop + and append to postfix, then push - onto stack.
→ Read, D append to postfix.
→ Pop remaining operators from stack and append to postfix

Postfix	Stack
[A]	[+]
[B]	[*]
[C]	[+, *] ↳ pop
[*, +]	
push	[-]
[D]	
final Postfix expression :-	
A B C * + D -	

(4) Convert the following infix expression into postfix and prefix expression using stack. show stack contents after each symbol.

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

(10) Evaluate the postfix expression in tabular form:

(a) 8 2 / 6 7 * +

Element	Stack	Action.
8	8.	Push 8
2	8 2	Push 2
/	4	Pop 2 and 8, evaluate 8/2=4, Push 4
6	4 6	Push 6
7	4 6 7	Push 7
*	4 2	pop 7 and 6, evaluate 6 * 7 = 42 push 42
+	-	pop 42 and 4, evaluate 4 + 42 = 46, push 46.

→ So the final result of the given expression is $\boxed{46}$.

(b) 5 3 + 6 2 / * 3 5 * +

element	Stack	Operation.
5	5	Push 5
3	5 3	Push 3
+	8	Pop 3 and 5, evaluate 3+5=8, push 8
6	8 6	Push 6.
2	8 6 2	Push 2
/	3	pop 2 and 6, evaluate 6/2=3, push 3
*	24	pop 3 and 8, evaluate 8 * 3 = 24, push 24
3	24 3	Push 3
5	24 3 5	Push 5
*	15	Pop 5 and 3, evaluate 3 * 5 = 15, push 15
+	-	Pop 15 and 24, evaluate 24+15=39 push 39.

(11) Answer the following questions.

(a) let S is an instance of the stack. Consider the following sequence of operations performed on S, which initially contains element 55 as top most elements. What is the status of top of stack after execution of each step ?

```
S.push(33);
S. push (20);
S. pop ();
S. pop ();
S. push (10);
S. push (40);
S. pop (~~40~~);
```

Initial stack :

TOP → 55

(1) S.push (33);

TOP → 33

55

(2) S.push (20);

TOP → 20

33

55

(3) S.pop();

TOP → 33 20 will be removed

55

(4) S.pop();

TOP → 55 33 will be removed.

(5) S.push(10);

Top → 10
55

(6) S.push(40);

Top → 40
10
55

(7) S.pop().

Top → 10 40 will be removed.
55

∴ Final status of stack = | Top → 10 |
| 55 |

(b) Let Q be an instance of a Queue. Q.enqueue(x) is insertion and Q.dequeue() deletion functions. Consider the following code and determine what will be the value of the variable count, when completes its execution? And show trace for every step.

```
Q.enqueue(1);
int count = 1;
do
{
count = count + 1;
x = Q.dequeue();
Q.enqueue(2* x);
Q.enqueue(4* x);
} while (x != 32);
```

→ q is initially empty.
→ Enqueue 1 into q: q = [1]
→ set count to 1.

→ loop begins:
count becomes 2.
Dequeue 1 from q, so x = 1.
enqueue 2*x = 2 and 4*x = 4 into q: q = [2,4].

→ loop continues:
count becomes 3.
Dequeue 2 from q, so x = 2.
Enqueue 2*x = 4 and 4*x = 8 into q: q = [4,8].

→ loop continues:
count becomes 4.
Dequeue 4 from q, so x = 4.
Enqueue 2*x = 8 and 4*x = 16 into q: q = [8,16]

→ loop continues:
count becomes 5
Dequeue 8 from q, so, x = 8
Enque 2*x = 16 and 4*x = 32 into q: q = [16, 32]

→ loop continues
count becomes 6
Dequeue 16 from q, so x = 16.
Enqueue 2*x = 32 and 4*x = 64 into q: q = [32, 64].

→ loop stops because x is now 32.
→ The final value of count is 6.

Scanned with OKEN Scanner

(12) Define queue. Write algorithms for INSERT, DELETE and DISPLAY operations of queue.

→ A queue is a linear data structure that follows the FIFO (First In First Out) principle.

→ In other words, the first element inserted is the first to be removed.

=[2,4].

→ Imagine it like a line of people waiting to purchase tickets, where the person at the front is ~~seter~~ served first.

[4,8].

(1) ENQUEUE (Insert).

Enqueue (Q, F, R, N, VAL)

→ Queue is represented by vector Q which contains N elements.

2,16]

→ F is a pointer which points to the front end. R is a pointer which points to the rear end.

→ VAL is the element to be inserted.

32]

[1] [Check for overflow]

```
If R>=N then
    Print "Queue Querflow"
    Exit.
```

4].

[2] [Increment rear pointer and check front pointer]

```
If F==0 then
    F=1
else
```

R = R+1

[3] [Insert new element at rear]
Q[R] = VAL

[4] [FINISH]
Exit.

(2) Dequeue [Delete].
Dequeue (Q, F, R)

→ Queue is represented by vector Q which contains N elements.

→ F is a pointer which points to the front end.
R is a pointer which points to the rear end.

[1] [Check for underflow]
If F == 0 and R == 0 then
Print "Queue Underflow"
Exit.

[2] [Copy the front element]
y = Q[F]

[3] [Increment the front pointer]
If F == R then
F = R = 0
else
F = F+1

[4] [Return the deleted value]
Return y

[5] [FINISH]
Exit.

(13) State disadvantages of simple queue. How to overcome it ? Explain with example.

- Limited flexibility:-

→ Normal Queue Data structure, we can insert elements until queue becomes full.

→ But once the queue becomes full, we can not insert the next element untill all the elements are deleted from the queue.

10	20	30	40	50	60	70

} Queue is full.

↑ Front ↑ Rear

→ Queue is Full and we cannot insert the new element because 'rear' is still at last position.

Queue is full (even three elements are deleted)

1			40	50	60	70

↑ front ↑ Rear

- Inefficient use of memory:-

→ ~~Elements can only be~~ A simple queue can lead to memory waste if elements are removed frequently, leaving empty spaces.

- No prioritization:-
→ All elements are treated equally, with no consideration for priority or urgency.

→ • To overcome these disadvantages, you can use variations of queues, such as:

(1) Circular Queue:-
→ A circular queue uses a fixed-size array to store elements, with the last element connected to the first element, making it more memory-efficient.

(2) Priority Queue:-
→ A priority queue allows elements to be assigned a priority, ensuring that the most important elements are processed first.

(3) Dynamic Queue:-
→ A dynamic queue, like a linked list-based queue, can grow or shrink as elements are added or removed, making it more memory-efficient.

example:-
→ Suppose you are managing a print queue with a simple queue.

→ Documents are added to the end, and the printer processes them in the order they were received.

→ However, if a high-priority document needs to be printed urgently, it would have to wait until all previous documents are printed.

→ By using a priority queue, you can assign high priority to the urgent document, ensuring it's printed next, even if it was added later than other documents.

(14) Write algorithm to (i) Insert (ii) Delete, and (iii) Display elements in circular queue.

→ (1) INSERT :-

→ CEnqueue (Cqueue, F, R, N, VAL).

→ Queue is represented by vector Cqueue which contains N elements.

→ F and R are pointers, incremented in circular way.

→ VAL is the element to be inserted.

```
(1) [Check for Overflow]
  If (F==1 and R == N) or (R==F-1) Then
        write (" Queue Overflow")
        Return
    END IF

(2) [Increment Rear in Circular way]
  IF (R==N) Then
        R = 1
  else
        R = R+1
  end if
```

(3) [Reset Queue if first time Enqueue].

```
IF (F==0) Then
        F = 1
end if
```

(4) [Insert the element]

```
Queue [R] = VAL
```

(5) [FINISH]

exit

(2) DELETE :-

→ CDequeue (Queue, F, R)

→ Queue is represented by vector Queue which contains N elements.

→ F and R are pointers, incremented in circular way.

(1) [check Underflow]

```
If (F==0) Then
        Write ("Queue Underflow")
        Return
end if
```

(2) [Copy the front element]

```
Temp = Queue[F]
```

(3) [Reset Queue if last element]

```
If (F==R) Then
        F = R = 0
end if
```

(4) [Increment the front in circular way]

```
If F == N Then
        F = 1
else
        F = F+1
```

(5) [Return the copied element]

```
Return (Temp)
```

(6) [FINISH]

```
exit.
```

(3) DISPLAY :-

→ Cqueue_Display (Cqueue, F, R): This procedure traverses the circular queue and displays all its elements Queue is represented by vector Cqueue which contains N elements. F and R are pointers.

[1] [Check Queue Underflow]

```
If F == 0 Then
        write ("Queue Underflow")
end if
```

(2) [Traverse the Queue and display the elements]

```
If F < R then
    for count = F to R
Do
        print CQueue [count];
End do
Else
```

```
for count = F To N (size)
        DO
        print (queue [count]
        End do.

For  count = 1 to R
        DO
        print (queue [count]
        End do.
[Finish]
  exit.
```

(15) Design an algorithm to insert and delete an element in doubly linked list.

- INSERT.

DOU_INS (L, R, M, X).

→ The insertion is to be performed to the left of a specific node with its address given by the pointer variable M.

→ Typical node of doubly linked list contains following fields. LPTR, RPTR and INFO.

→ LPTR is pointer variable pointing to predecessor of a node.

→ RPTR is pointer variable pointing to successor of a node.

→ L and R are pointer variables pointing for Leftmost and Rightmost node of linked list.

→ New

→ X is

DOU_I

(1) [Cre
Ne

(2) [c
Inf

(3) [I
IF
Th

(4) [
If
T

(5) [I
LPT
RP
LP
RP
Re

→ New is the address of New node.

→ X is value to be Inserted.

DOU_INS (L, R, M, X).

(1) [Create New Empty Node]
New = Node.

(2) [copy information field]
Info (New) = X

(3) [Insert into an empty list]
IF R = Null
Then LPTR(New) Null
RPTR(New) Null
L R New
Return

(4) [Is left most Insertion ?]
If M = L
Then LPTR(New) NULL
RPTR (New) M
LPTR (M) New
L New
Return.

(5) [Insert in middle]
LPTR (New) LPTR (M)
RPTR (New) M
LPTR (M) New
RPTR (LPTR (New)) New
Return.

• Delete.
DOU_DEL (L, R, OLD)

→ Typical node of doubly linked list contains following fields LPTR, RPTR and INFO.

→ LPTR is pointer variable pointing to Predecessor of a node.

→ RPTR is pointer variable pointing to Successor of a node.

→ L & R are pointer variables Pointing for Leftmost and Rightmost node of Linked list.

• DOU_DEL (L, R, OLD)

(1) [Is underflow ?]
If R = Null
Then write ("Underflow")
Return

(2) [Delete Node]
If L = R (single node in list)
Then L R Null
else if OLD = L (left most)
Then L RPTR(L).
LPTR (L) = Null
else if OLD = R (Right most)
Then R LPTR (R)
RPTR (R) Null
else RPTR (LPTR (old)) RPTR(old)
LPTR (RPTR (old)) LPTR (old)

(3) [free delet
Free (old

(16) What is
is anyho
answer

→ Priority
→ A priorit
structure
associated

→ Unlike
the FIF
dequeue

→ Elements
before

→ Common
managin
algorithm

→ Impleme
Fibonac
structure

(2) Simple
→ Follow

→ Element
were a

(3) [free deleted node ?]
Free (old)

(16) What is priority queue ? Is simple queue is anyhow priority queue ? Explain your answer

→ Priority Queue :-

→ A priority Queue is a specialized data structure where each element has an associated priority.

→ Unlike a regular queue (which follows the FIFO principle), a priority queue dequeues elements based on their priority.

→ Elements with higher priority are dequeued before those with lower priority.

→ Common use cases include scheduling tasks, managing events, and optimizing algorithms. ~~like~~

→ Implementation options include binary heaps, Fibonacci heaps, and other efficient structures.

(2) Simple Queue (FIFO):

→ Follows the first-in-first-out (FIFO) order

→ Elements are processed in the order they were added.

→ No concept of priority.

→ Example:- Regular line at a grocery store.

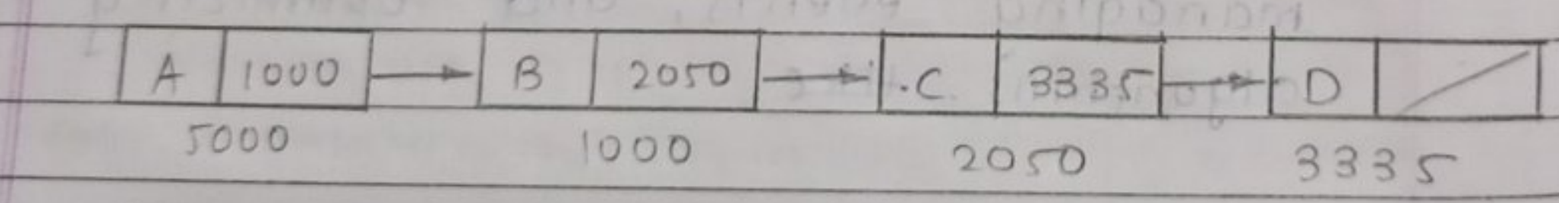
→ While a simple queue doesn't consider priorities, a priority queue optimizes element processing based on their importance or urgency.

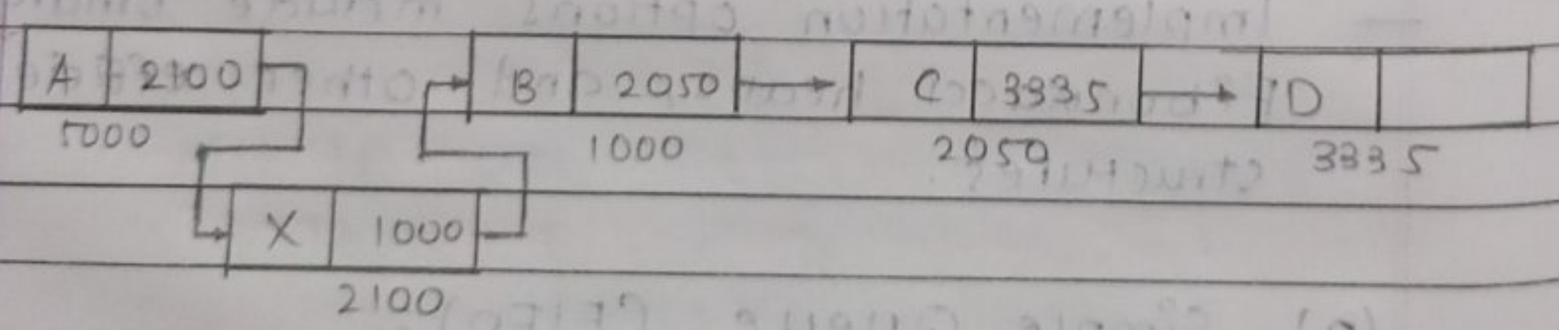
(17) Explain pros & cons of linked allocation

- Insertion Operation

→ We have an n elements in list and it is required to insert a new element between the first and second element, what to do with sequential allocation & linked allocation?

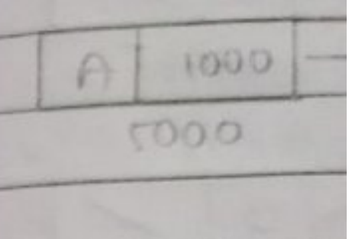
→ Insertion operation is more efficient in linked allocation

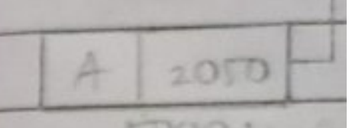




- Deletio

→ Deletion allocatio





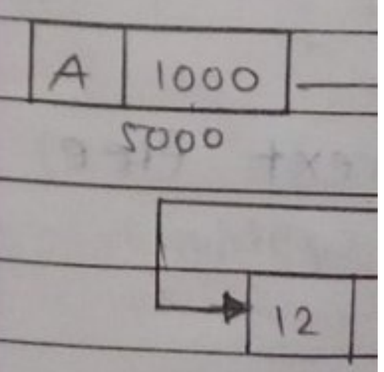
- Search

→ If po it is re the first node is time co a sequ

→ Search in li

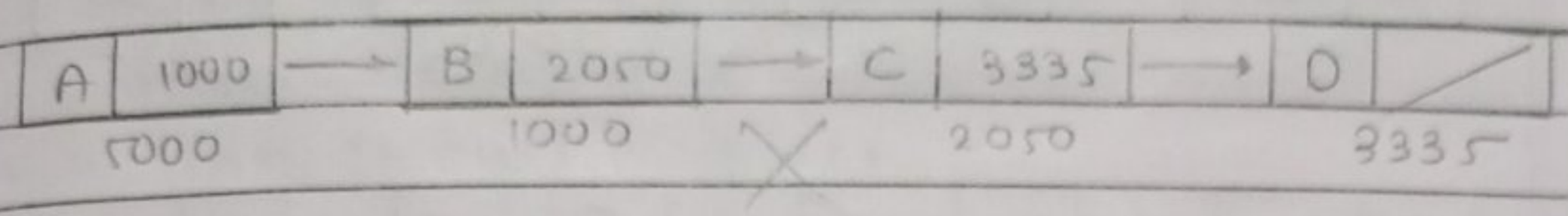
- Join o

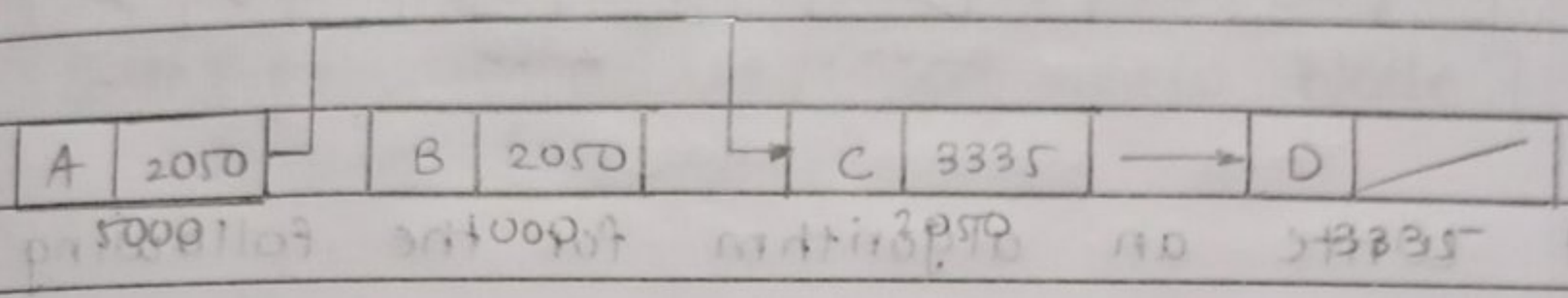
→ Join allocatio



- Deletion Operation.

→ Deletion operation is more efficient in linked allocation.





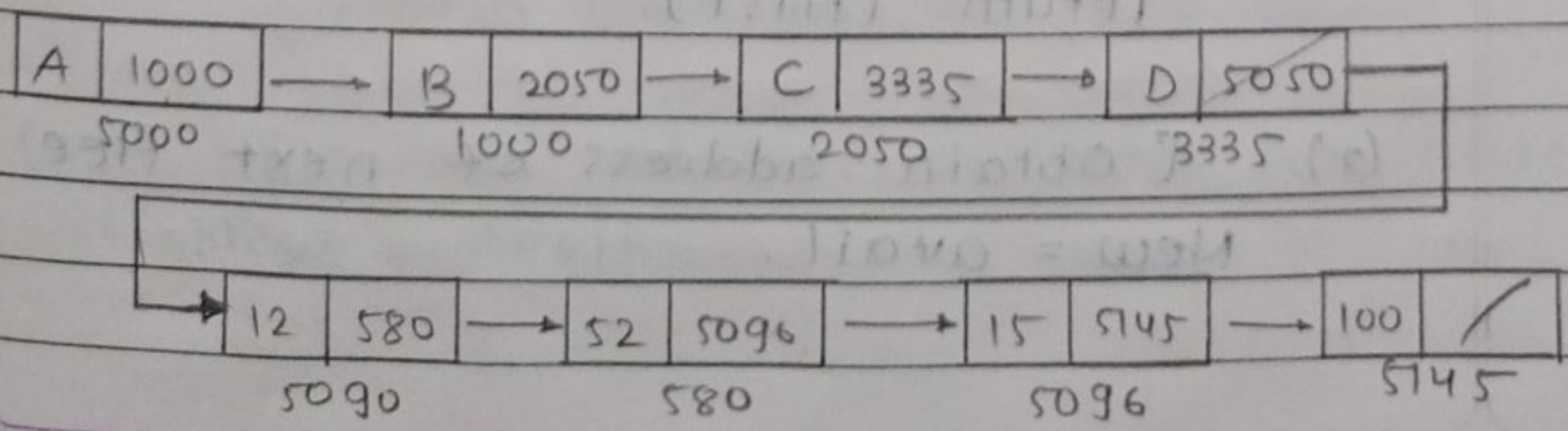
- Search Operation.

→ If particular node in the list is required, it is ~~required~~ necessary to follow links from the first node onwards until the desired node is found; in this situation it is more time consuming to go through linked list than a sequential list.

→ Search operation is more time consuming in linked allocation.

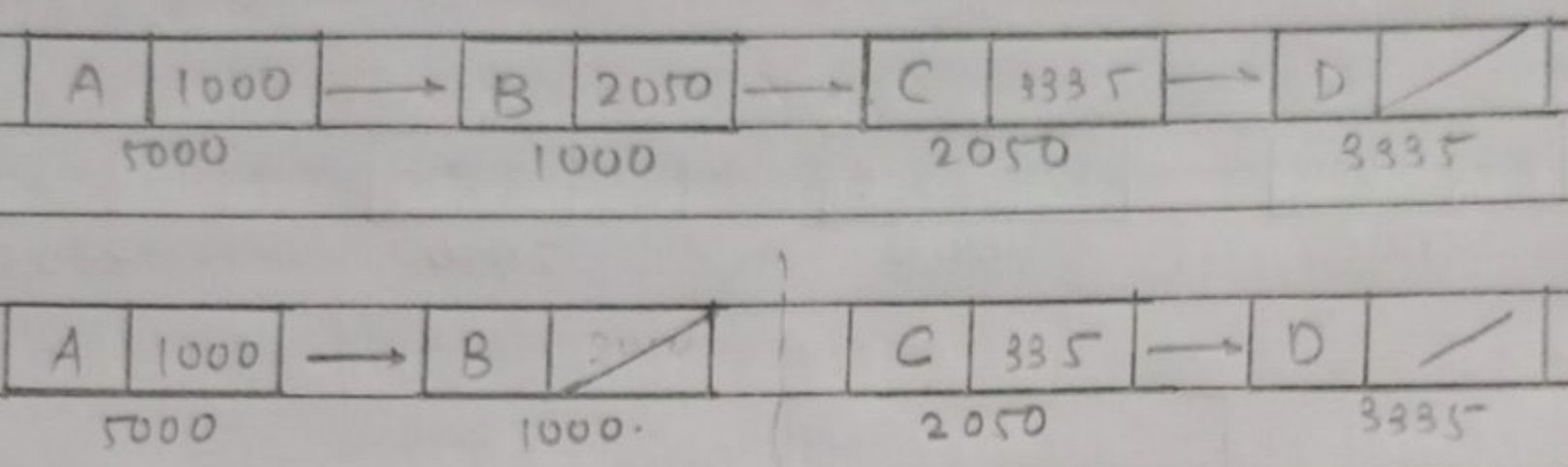
- Join operation:-

→ Join operation is more efficient in linked allocation.



- Split operation.

→ Split operation is more efficient in linked allocation.



(18) Write an algorithm for the following in singly linked list.

(a) Insert at first position.

INSERT (X, First).

→ This algorithm returns, address of first node

→ X is a new element to be inserted.

→ First is a pointer to the first element of a singly linked linear list.

→ Typical node contains INFO and LINK fields.

→ AVAIL is a pointer to the top element of the availability stack.

→ New is a temporary pointer variable.

[1] [Underflow ?]
If AVAIL = NULL
Then write ("Availability Stack Underflow")
return (first)

(2) [Obtain address of next free Node]
New = avail.

(3) [Remove free node from availability stack]
avail = link(avail)

(4) [Initialize fields of new node and its link to the list]
Info (New) = X
link (New) = First.

(5) [Return address of new node]
Return (New)

(b) Last Position :-

→ This algorithm inserts a new node at the last position of linked list.
INSEND (X, First)

→ X is a new element to be inserted.

→ First is a pointer to the first element of a singly linked linear list.

→ Typical node contains INFO and LINK fields

→ Avail is a pointer to the top element of the availability stack.

→ New is a temporary pointer variable.

(1) [Underflow ?]
If AVAIL = NULL
Then write ("availability stack Underflow")
return (FIRST)

(2) [Obtain address of next free node]
New = Avail

(3) [Remove free node from availability Stack]
Avail = link(avail).

(4) [Initialize fields of new node]
INFO(New) = X
Link (New) = Null

(5) [Is the list empty ?]
If First = NULL
Then Return (New).

(6) [Initialize search for a last node]
Save = first

(7) [Search for end of list]
Repeat while link (save) ≠ Null
Save = link (Save).

(8) [Set link field of last node to NEW]
Link (Save) = New.

(9) [Return first node pointer]
Return (First).

(c) Insert in ordered linked list:

→ This algorithm inserts a new node such that linked list preserves the ordering of the terms in increasing order of their INFO field.

→ X is a new element to be inserted.

→ First
a Sin

→ Typica
→ New
INSO
(1) [und
If
Then
Re

(2) [obt
New

(3) [Rem
avail

(4) [Is
If
then
Ret

(5) [Does
node i
If IN
then link (
ret

(6) [Initi
Save

→ First is a pointer to the first element of a singly linked linear list.

→ Typical node contains INFO and LINK fields.

→ New is a temporary pointer variable

INSORD (X, First)

(1) [underflow ?]
If AVAIL = Null
Then write ("Availability stack underflow")
Return (First).

(2) [obtain address of next free node]
New ← avail.

(3) [Remove free node from availability stack]
avail ← link(avail)

(4) [Is the list is empty]
If FIRST = NULL
then link (new) ← NULL
Return (New).

(5) [Does the new node precede all other node in the list ?]
If INFO (New) ≤ Info(First)
then link (new) ← first.
return (New)

(6) [Initialize temporary pointer].
Save ← First.

(7) [search for predecessor of new node]
Repeat while link (save) ≠ null and
info(new) ≥ info(link(save))
save ← link(save).

(8) [set link field of new node and its predecessor].
link(new) ← link(save)
link(save) ← New

(9) [Return first node pointer]
Return (First).

(d) Delete element :-

→ This algorithm delete a node whose address is given by variable X.

→ First is a pointer to the first element.
→ Typical node contains INFO and LINK fields.
→ save and pred are temp. pointer variable.

Delete (X, First)
(1) [Is empty list?]
If First = Null
then write ("Underflow")
return

(2) [Initialize search for X]
save ← first

(3) [Find x]
→ Repeat thru step-5 while save ≠ x and link(save) ≠ Null.

(4) [update predecessor marker]
pred ← save

(5) [Move to next node]
save ← link(save)

(6) [End of the list]
If save ≠ x
then write ("Node not found")
return.

(7) [Delete x].
If x = first (if x is first node)?
then first ← link(first)
else link(pred) ← link(x)

(8) [Free deleted Node]
Free [x].

(e) copy linked list
→ This algorithm copy a link list and creates a new linked list

(1) [Is empty List ?]
If First = NULL
then return (Null)

(2) [copy first node]
New ← Node
New ← Avail
avail ← link(avail)
field (New) ← Info(first)
Begin ← New.

(3) [Initialize traversal]
save ← First

(4) [Move the next node if not at the end if list]
Repeat thru step 6 while (save) ≠ Null.

(5) [update predecessor and save pointer]
pred ← New
save ← link (save)

(6) [copy node]
If avail = Null
then write ("Availability stack Underflow")
Return (0)
else New ← Avail
avail ← link(avail)
field (New) ← Info(save)
ptr (pred) ← New

(7) [set link of last node and return]
ptr (New) ← Null
Return (Begin)

Page No:
Date:

(19) Write an algorithm for the following in circular linked list

(a) Insert at first Position.

→ CIRCULAR_LINK_INSERT_FIRST (X, First, Last)

(1). [create New Empty Node]
New ← Node

(2) [Initialize fields of new node and its link to the list]
Info (New) ← X
If First = Null
then Link(New) ← New
First ← Last ← New
else Link (New) ← First
Link (Last) ← New
First ← New
Return

(b) Insert at Last Position.
CIR_LINK_INSERT_END (X, First, Last).

(1) [create New Empty Node]
New ← Node

(2) [Initialize fields of new node and its link to the list]
If First = Null
then link (New) ← New
First ← last ← New

Scanned with OKEN Scanner

else Link(New) ← First
Link (Last) ← New
Last ← New
Return.

(c) Insert in Ordered linked list :-

CIR_LINK_INSERT_ORDER(X, First, Last)

[1]. [create New Empty Node]
New ← Node.

[2] [copy Information content into new node]
Info (New) ← X

[3] [Is linked list is empty ?]
If First = NULL
then Link(New) ← New
First & ← Last ← New
Return.

[4] [Does new node precedes all other nodes in list?]
If INFO(New) ≤ Info (First)
then link (New) ← First
Link (Last) ← New
First ← New
Return.

[5] [Initialize Temporary Pointer]
Save ← First

[6] [Search for predecessor of new node]
Repeat while save ≠ last and Info (New) ≥
Info (link (save))
save ← link(save).

[7] [set link field of New node and its predecess-or]
LINK (New) ← link(save)
Link(save) ← New
If save = Last
then last ← New

[8] [FINISH]
Return.

(d) Delete Element:-
CIR_LINK_Delete (X, First, Last):

(1) [Is empty list ?]
If First = NULL
then write ('linked list is Empty')
Return

(2) [Initialize Search for X]
save ← first

(3) [Find X]
Repeat thru step 5 while save ≠ X and
save ≠ last.

(4) [Update predecessor marker]
Pred ← save

(5) [Move to next node]
save ← link(save)

(6) [End of linked list]
If save ≠ x
then write ('Node Not found')
Return

(7) [Delete x]
If x = first
then first ← link(first)
link(last) ← first
else Link(Pred) ← link(x)
If x = last
then Last ← Pred.

(8) [Free DeletedNode]
Free (x).

(20) Explain the following operations of doubly linked list with example.
(a) Insert first:-
DOUB_INS (L, R, M, x).

(1) [Create New Empty Node]
New ← Node

(2) [Copy information field]
Info (New) ← x

(3) [Insert into an empty list]

If R = Null
then LPTR (New) ← RPTR (Null) ← Null
L ← R ← New
Return.

(4) [Is left most insertion ?]
If M = L
then LPTR (New) ← Null
RPTR (New) ← M
LPTR (M) ← New
L ← New
Return.

(5) [Insert in Middle].
LPTR (New) ← LPTR (M)
RPTR (New) ← M
LPTR (M) ← New
RPTR (LPTR (New)) ← New
Return

(b) Insert in Ordered linked list:-
DOUB_INS_ORD (L, R, M, X).

(1) [Create New Empty Node]
New ← Node

(2) [Copy Information field]
INFO (New) ← X.

(3) [Insert into an empty list]

If R = NULL
then LPTR (New) ← RPTR (NULL) ← Null
L ← R ← New
Return

(4) [Does the new node precedes all other nodes in list ?]
If INFO(NEW) ≤ INFO(L)
then RPTR (New) ← L
LPTR (New) ← Null
LPTR (L) ← New
L ← New
Return

(5) [Initialize temporary Pointer]
Save ← L

(6) [Search for predecessor of New node]
Repeat while RPTR (save) ≠ Null and INFO(New) ≥ Info(RPTR (save))
Save ← RPTR (save)

(7) [Set link field of new node and its predecessor]
RPTR (New) ← RPTR(save)
LPTR (RPTR(save)) ← New
RPTR (save) ← New
LPTR (New) ← save

If save = R
then RPTR(save) ← New.

(c) Delete:-

DOUB_DEL (L, R, OLD)

[1] [Is underflow?]
If R = NULL
then write ('underflow')
return.

(2) [Delete Node]
If L = R (single node in list)
then L ← R ← Null
else If OLD = L.
then L ← RPTR (L)
LPTR (L) ← Null.
else If OLD = R
then R ← LPTR (R)
RPTR (R) ← Null
else RPTR (LPTR(OLD)) ← RPTR (OLD)
LPTR (RPTR (OLD)) ← LPTR (OLD)

[3] [Free deleted Node]
Free (OLD).

(21) Explain Enqueue and Dequeue operations of simple queue using a singly linked list.

→ (1) Enqueue:-
→ when you enqueue an element, you add it to the end (rear) of the queue.

→ Create a new node with the given data.

→ If the queue is empty (the head is none), set the head to the new node.

→ Otherwise, traverse the list to find the last node.

→ Set the last node's next pointer to the next node.

→ (2) Dequeue.

→ When you dequeue an element, you remove it from the front (head) of the queue.

→ If the queue is empty, there's nothing to dequeue.

→ Otherwise, update the head to point to the next node.

(22) Explain push and pop operations of stack using singly linked list.

→ (1) PUSH:-

→ The push operation adds a new element to the top (beginning) of the stack.

→ Create a new node with the given data.

→ Set the new node's next pointer to the current top (if any).

→ Update the top pointer to point to the new node.

→ (2) POP:-

→ The pop operation removes the top element from the stack.

→ If the stack is empty, there's nothing to pop.

→ ~~If the stack~~ Otherwise, remove the top node by updating the top pointer to point to the next node.

→ Return Data of the removed node.

Assignment : 3

(1) How can a general tree be converted into a binary tree? Provide a step-by-step explanation

→ (1) Root Node :-

→ The root of the binary tree is the same as the root of the general tree.

→ (2) Children Mapping :-

→ For each node in the general tree :

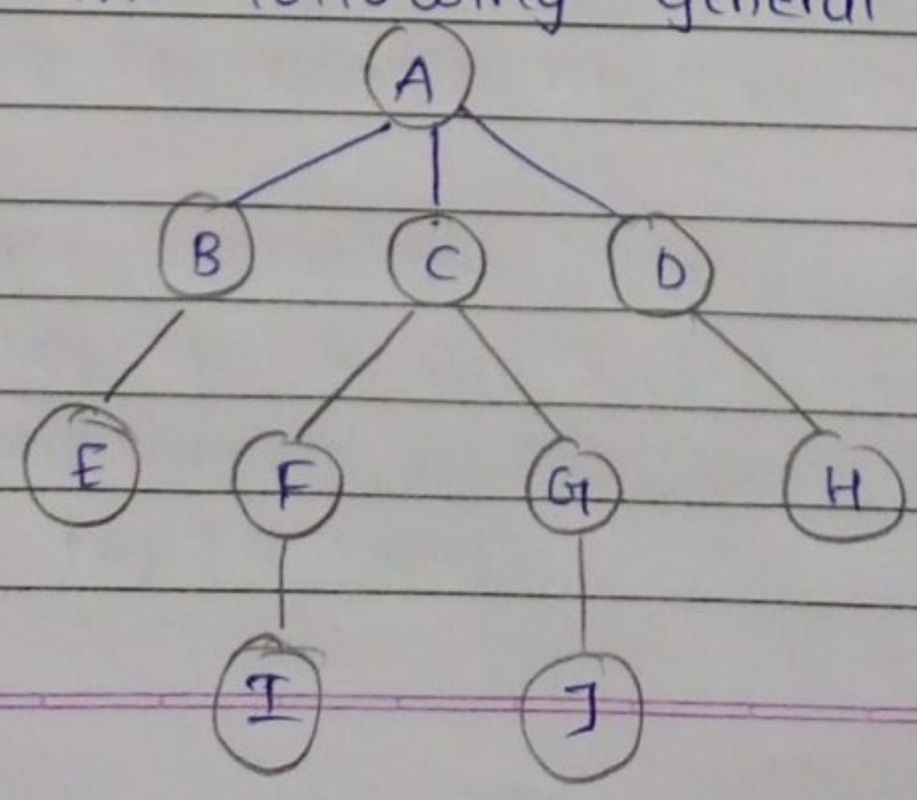
→ The left child of that node in the ~~gre~~ general tree becomes the left child of the corresponding node in the binary tree.

→ The right sibling of that node in the general tree becomes the right child of the corresponding node in the binary tree.

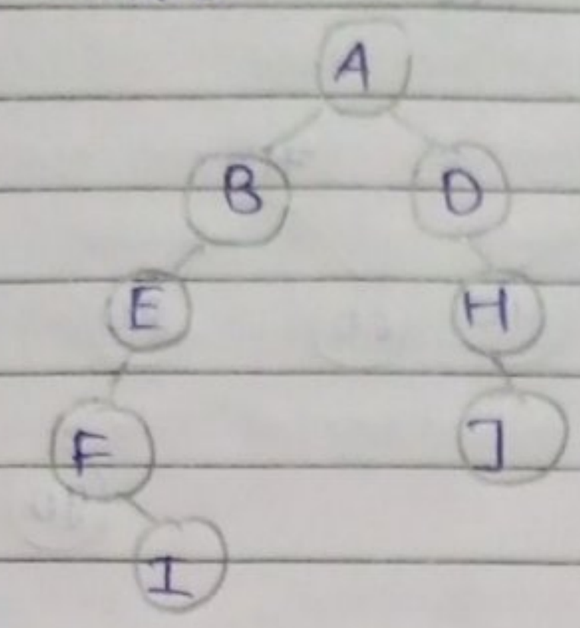
• → If a parent node in the general tree has only a right child, that right child becomes the rightmost child node of the last node following the parent node in the binary tree.

→ For example :-

cosider the following general tree :



→ Binary tree:-



② Explain binary search tree with suitable example.

→ A binary search tree is a binary tree in which each node possessed a key that satisfy the following conditions:-

→ All key (if any) in the left sub tree of the root precedes the key in the root.

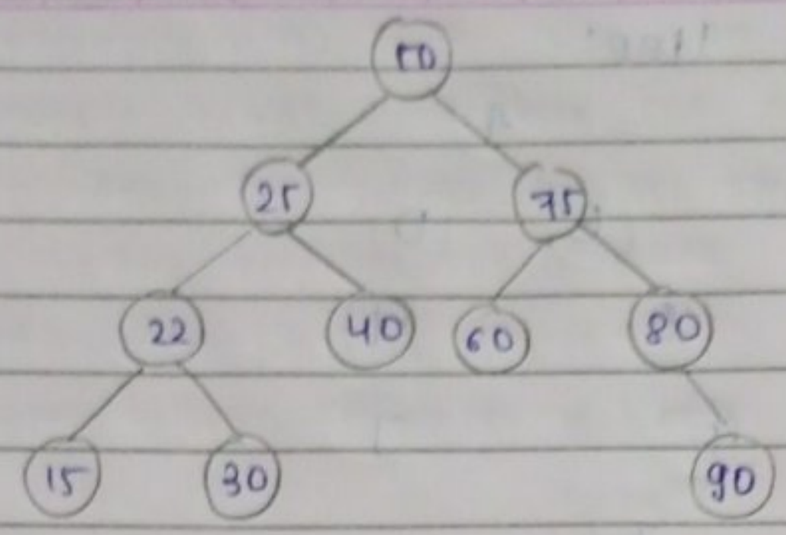
→ The key in the root precedes all key (if any) in the right subtree

→ The left and right sub tree of the root are again search trees.

- example :-

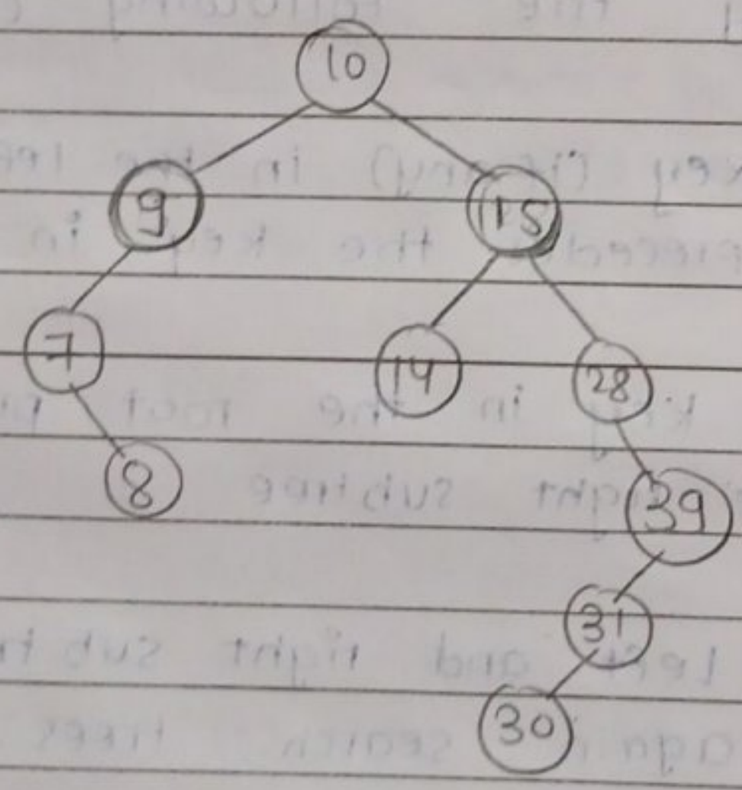
Create a binary search tree for the following data:-

50, 25, 75, 22, 40, 60, 80, 90, 15, 30



(3) Create Binary Search tree for following Data and write pre-order traversal, in-order traversal and post-order traversal of the constructed tree.

(a) 10, 15, 28, 09, 39, 31, 30, 14, 07, 08.



→ In-order traversal :-

7, 8, 9, 10, 15, 14, 28, 31, 30, 39

→ Pre-order traversal :-

10, 9, 7, 8, ~~15~~, 14, 15, 28, 30, 31, 39

→ Post-order traversal :-

8, 7, 9, 14, 30, 31 39, 28, 15, 10

(4) What is AVL tree? Explain various rotations with suitable ~~tree~~ example.

→ A tree is called AVL tree (Height Balanced tree), if each node possessed one of the following properties:-

→ A node is called left heavy, if the longest path in its ~~su~~ left sub tree is one longer than the longest path of its right sub tree.

→ A node is called right heavy, if the longest path in its right subtree is one longer than the longest path of its left sub tree

→ A node is called balanced, if the longest path in both the right and left-subtrees are equal.

- Types of rotations:-

(1) Insertion into left sub-tree of nodes left child - Single Right Rotation.

(2) Insertion into Right sub-tree of node's left child - left Right Rotation.

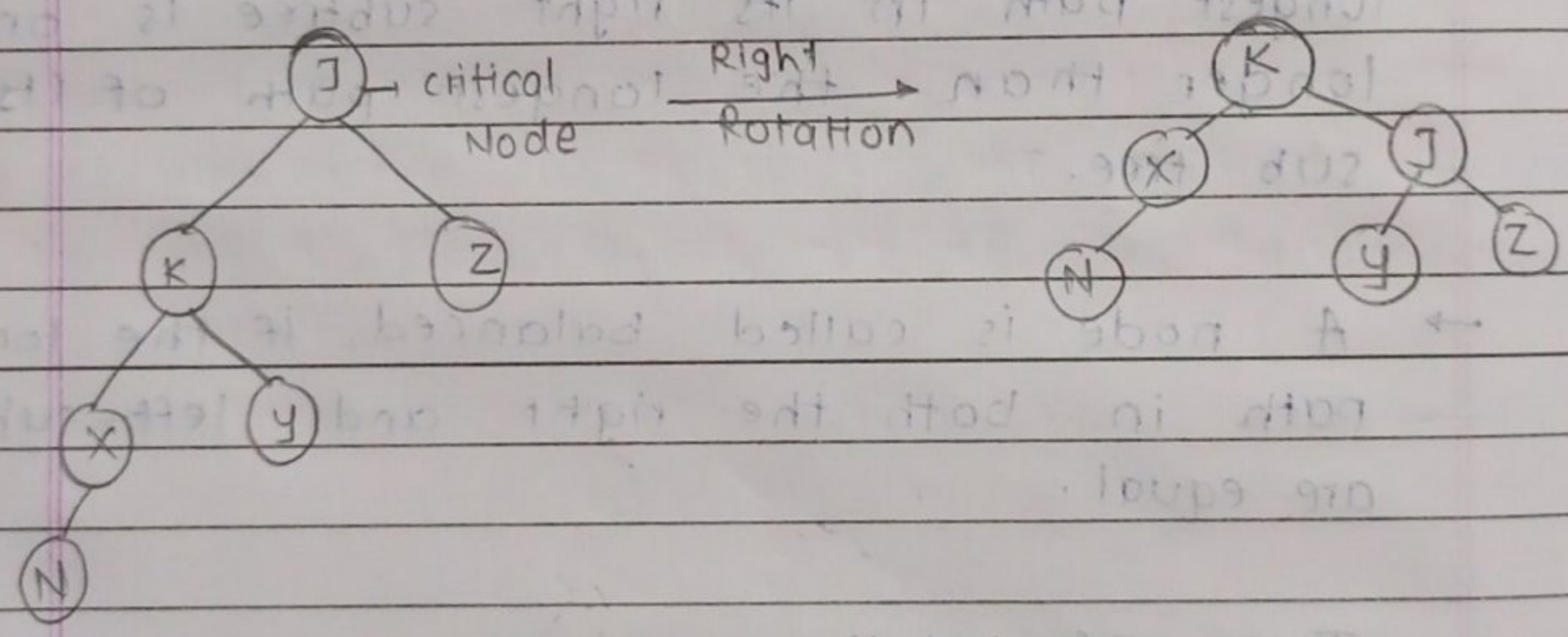
(3) Insertion into left sub-tree of node's Right child Right left Rotation.

(4) Insertion into Right sub-tree of node's Right child - single left Rotation.

(1) Insertion into left sub-tree of nodes left child.
Single Right Rotation.

→ If node becomes unbalanced after insertion of new node at ~~left~~ left sub-tree of nodes left child, then we need to perform Single Right Rotation for unbalanced node.

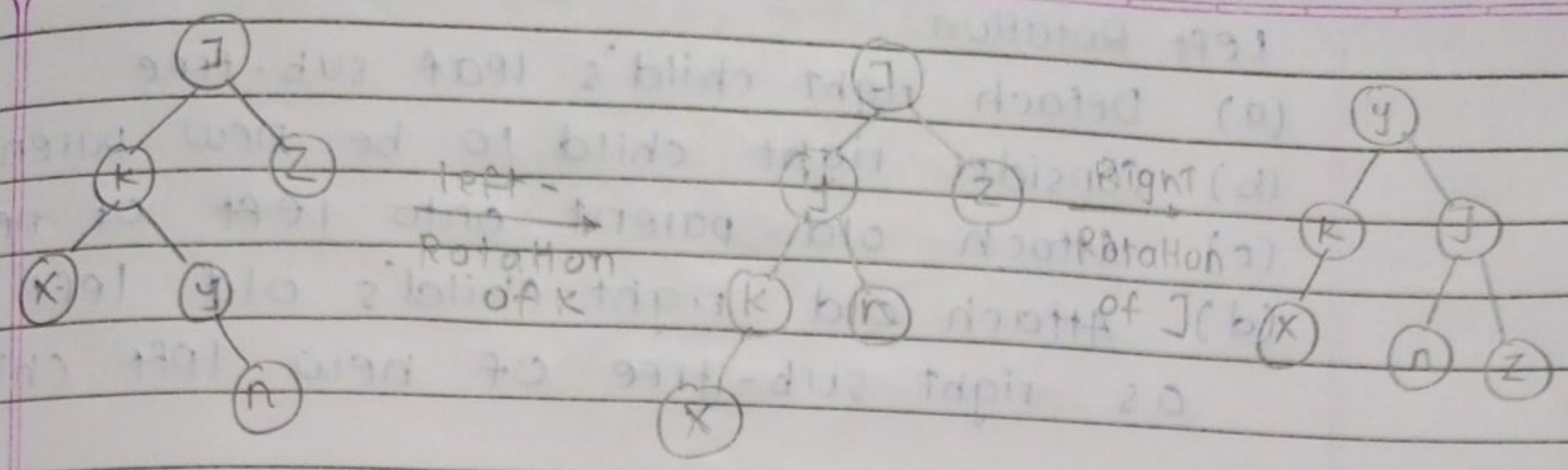
• Right Rotation :-
(a) Detach leaf child's right subtree.
(b) Consider leaf child to be the new parent.
(c) Attach old parent onto right of new parent.
(d) Attach old leaf child's old right sub-tree as leaf sub-tree of new right child.



(2) Insertion into Right sub-tree of node's left child - left Right Rotation.

→ If node becomes unbalanced after insertion of new node at Right sub-tree of node's left child, then we need to perform Left Right Rotation for unbalanced node.

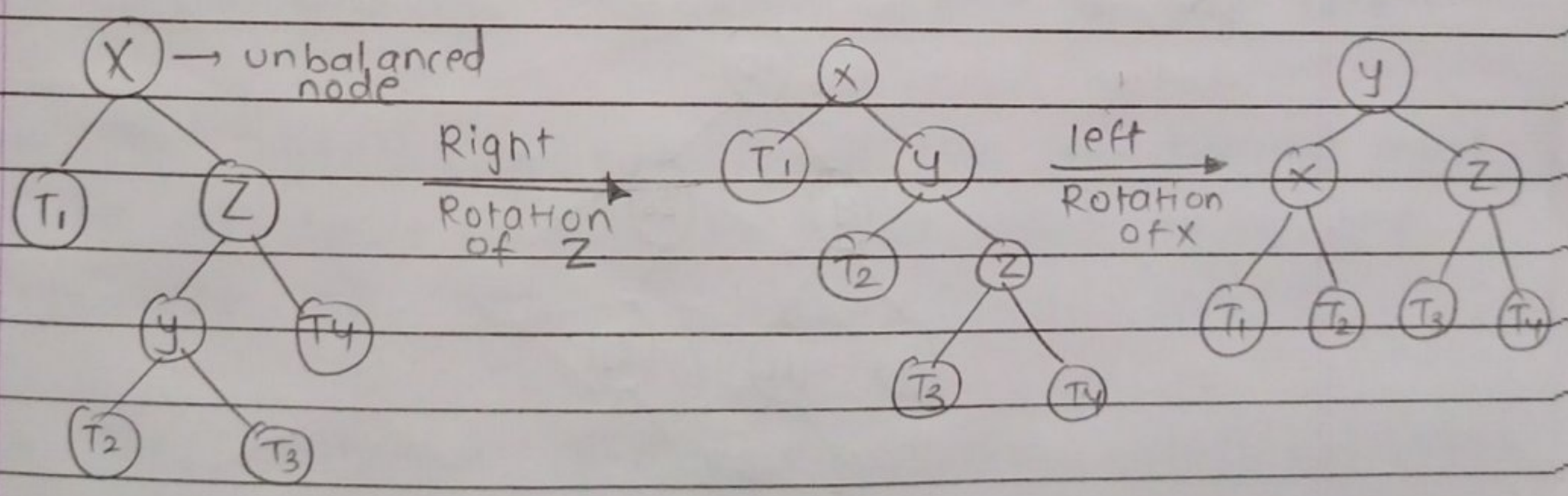
→ Leaf rotation of leaft child followed by right rotation of parent.



(3) Insertion into left sub tree of node's Right child - Right left Rotation.

→ If node becomes unbalanced after insertion of new node at left sub-tree of node's Right child, then we need to perform Right left Rotation for unbalanced node.

→ Single right rotation of right child followed by left rotation of Parent.

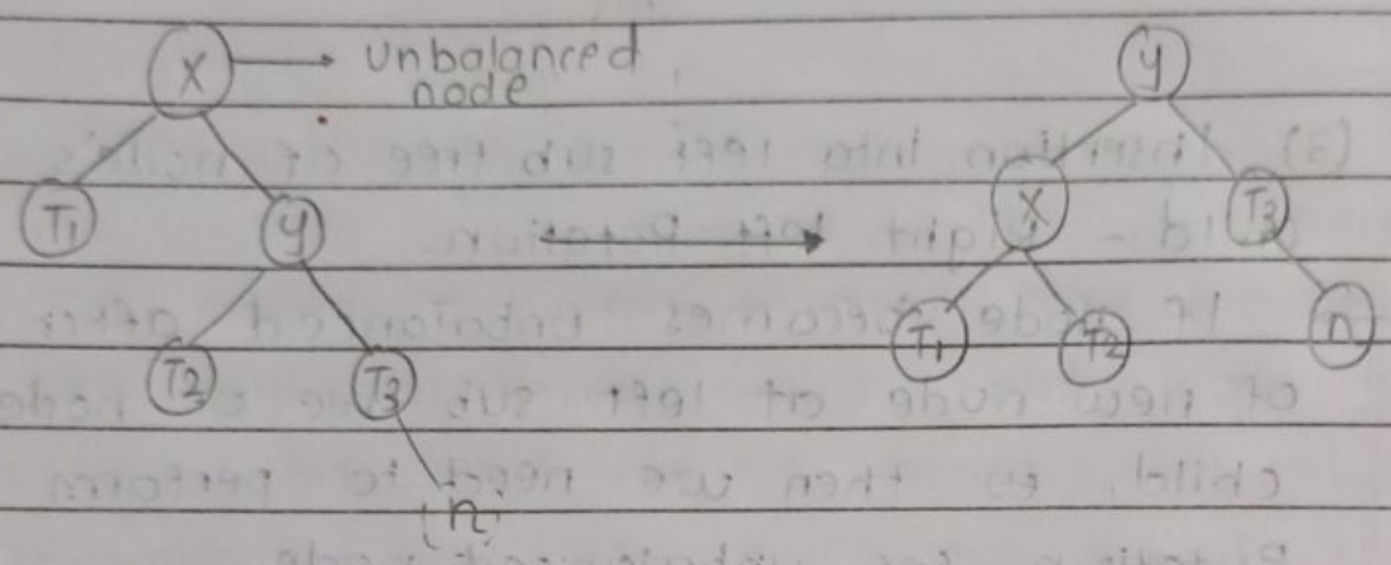


(4) Insertion into Right sub-tree of node's Right child - Single left Rotation.

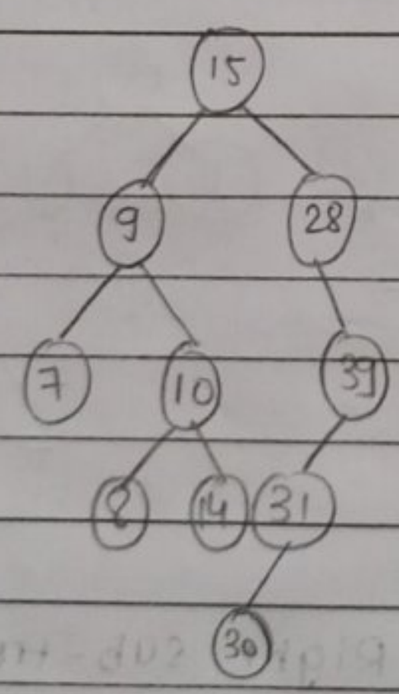
→ If node becomes unbalanced after insertion of new node at Right sub-tree of nodes Right child, then we need to perform Single left Rotation for unbalanced node.

Left Rotation

(a) Detach right child's leaf sub-tree
(b) Consider right child to be new parent
(c) Attach old parent onto left of new parent
(d) Attach old right child's old left sub-tree as right sub-tree of new left child.

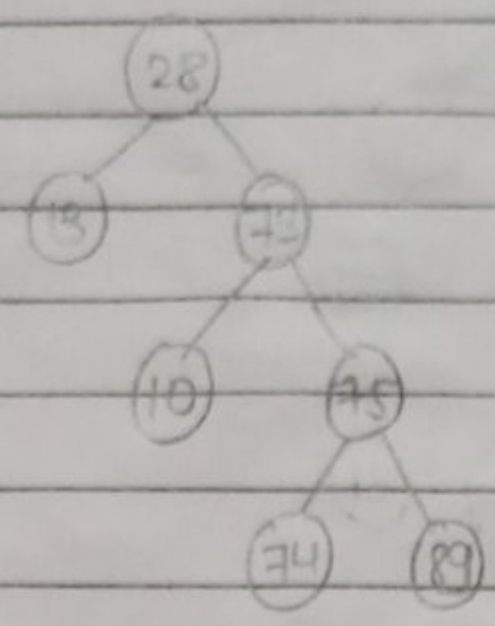


(5) Create AVL tree for following data and write pre-order traversal of the constructed tree.
(a) 10, 15, 28, 09, 39, 31, 30, 14, 07, 08.



Pre-order Traversal:-
→ 15, 9, 7, 10, 8, 14, 28, 39, 31, 30.

(b) 28, 73, 89, 75, 74, 13, 10.



Pre-order Traversal:-

→ 28, 13, 10, 73, 75, 74, 89

(6) Explain weight balanced tree with proper eg.

→ A weight-balanced tree is a type of binary search tree (BST) where the balanced is determined by the number of nodes in each subtree.

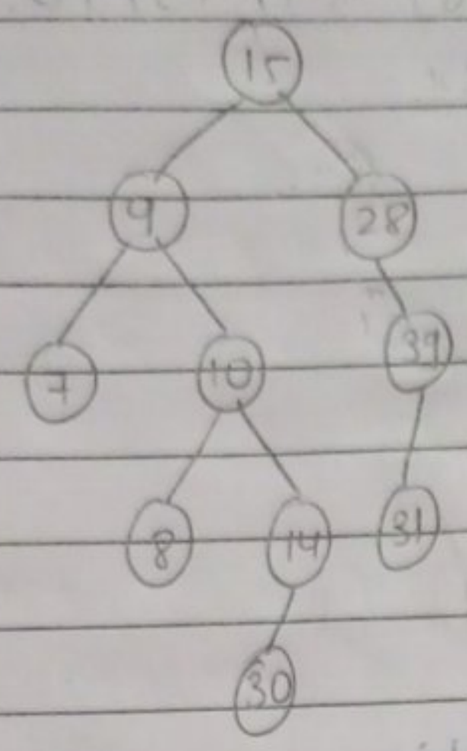
→ Specifically, for each node in the tree:

→ The number of nodes in the left subtree must be at least half and at most twice the number of nodes in the right subtree.

→ This balance is based on the weights rather than the height of the subtrees.

→ example :-

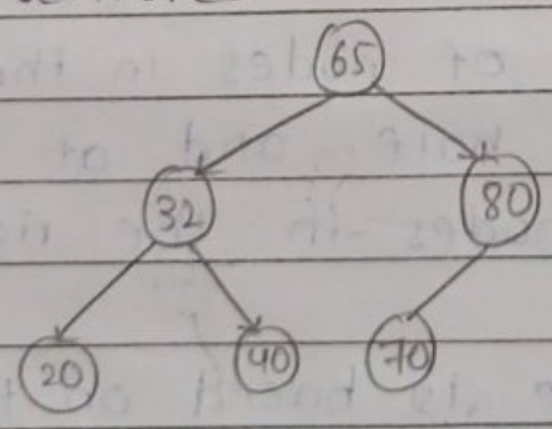
→ Consider the following weight-balanced tree :-



→ For node 15 : Left subtree has 3 nodes (9, 7, 10) and right subtree has 4 nodes (28, 14, 39, 31).

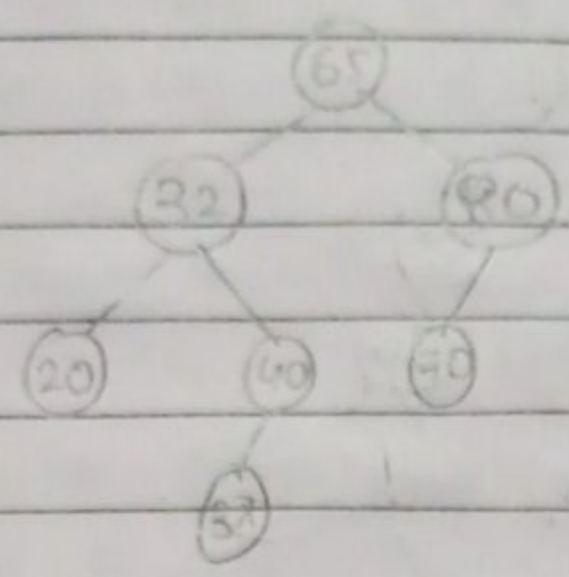
→ For node 28 : Left subtree has 1 node (14), and right subtree has 3 nodes (39, 31, 30).

(7) Insert 38, 75, 85, and 15 in the AVL tree shown below. Draw rebalancing of tree after every insertion.



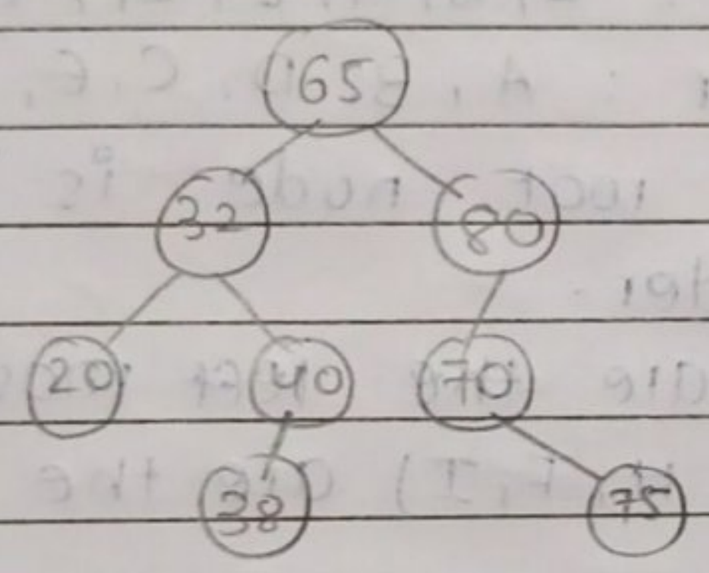
→ Insert 38.

→ Here 38 is greater than 32 but less than 40, so it goes to the left of 40.



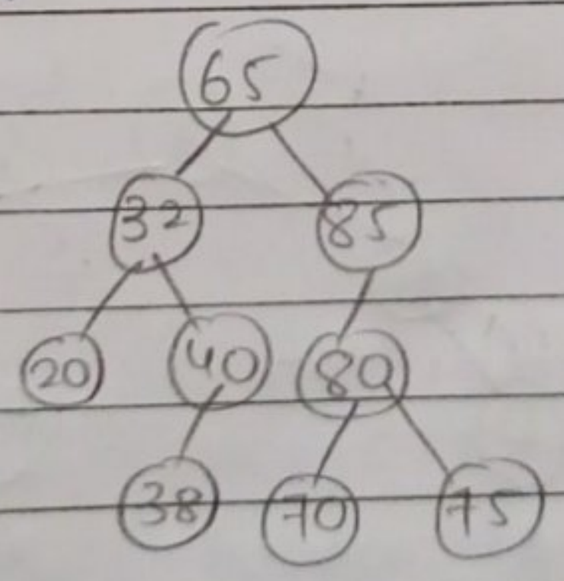
→ Insert 75.

→ 75 is less than 80 but greater than 70, so it goes to the right of 70.



→ Insert 85

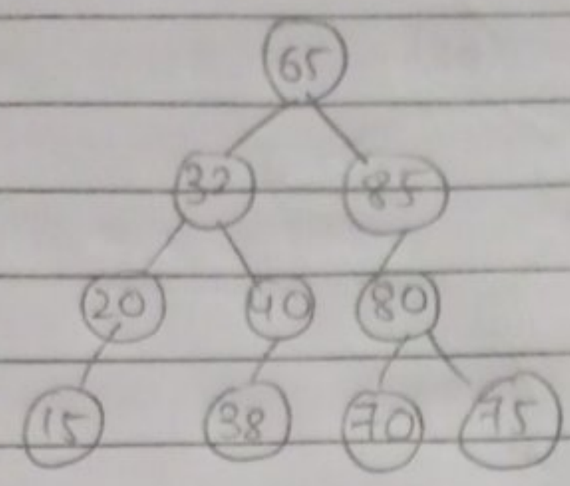
→ 85 is greater than 80, so, it goes to the right of 80.



→ Insert 15.

→ 15 is less than 32 ~~b~~ and less than 20, so it goes to the left of 20.

Scanned with OKEN Scanner



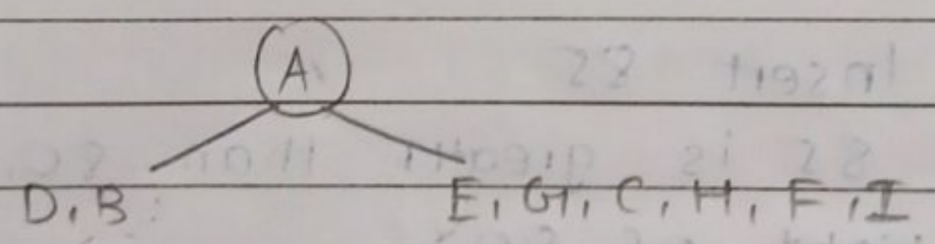
(b)

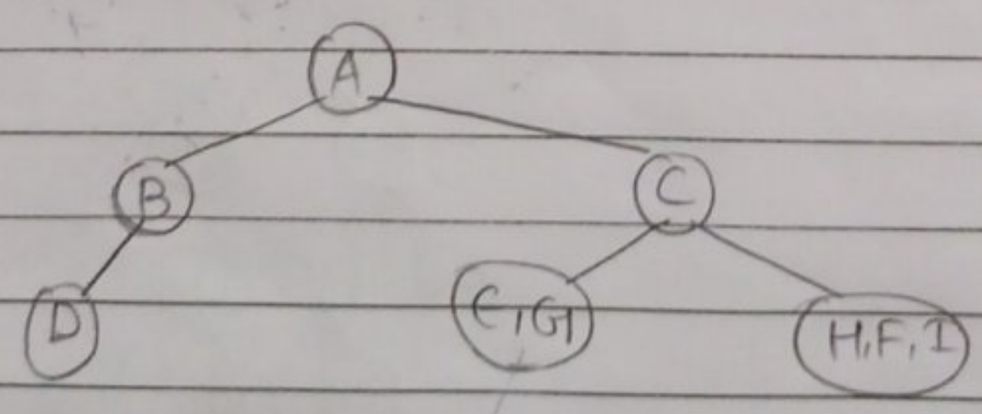
(8) Construct a binary tree from the traversals given below:

(a) In-order : D, B, A, E, G, C, H, F, I
Pre-order : A, B, D, C, E, G, F, H, I

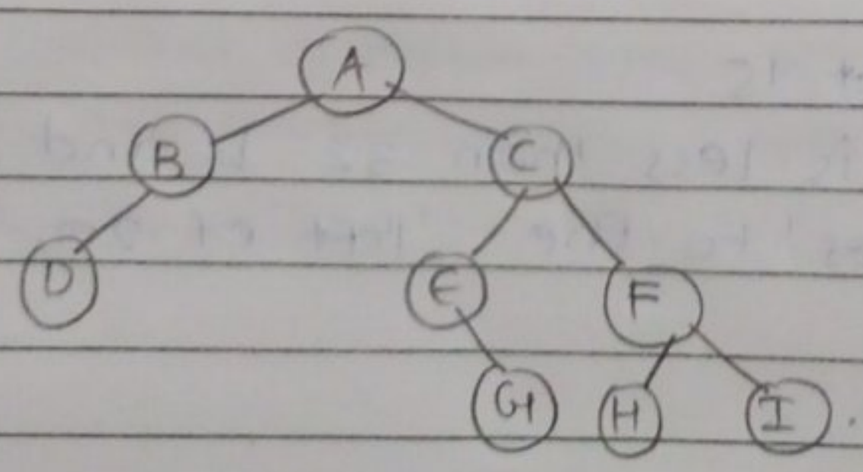
Here, the root node is 'A' as it is pre-order.

→ (D, B) are the left most child of A and (E, G, C, H, F, I) are the Right most child of A.





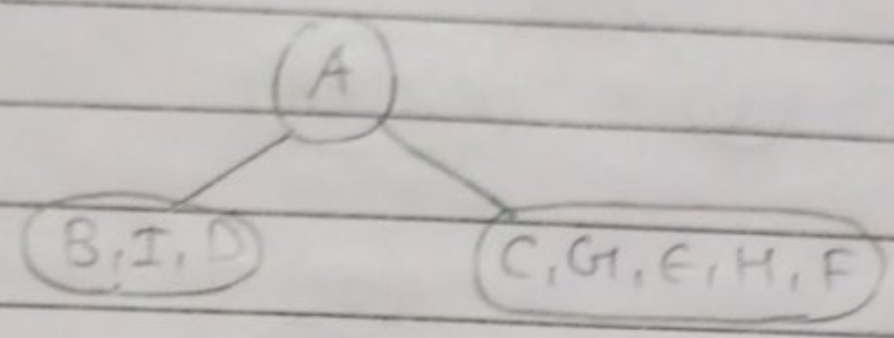
(c)

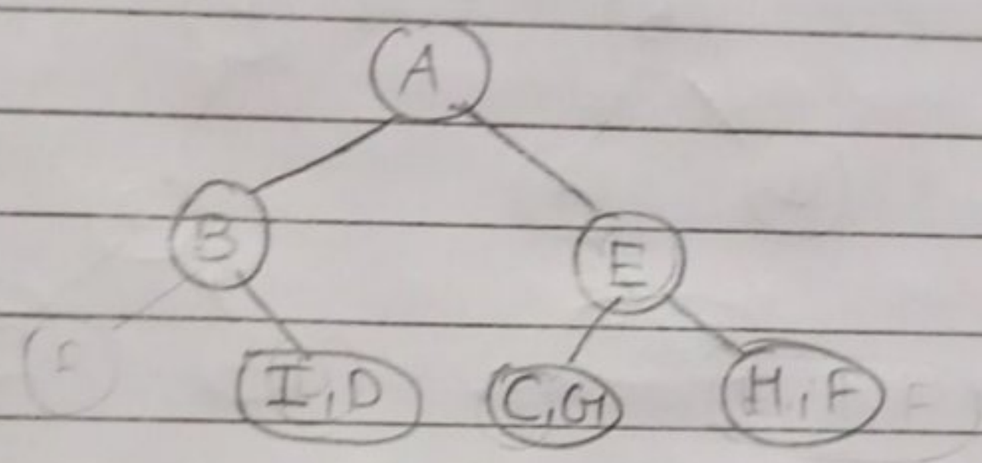


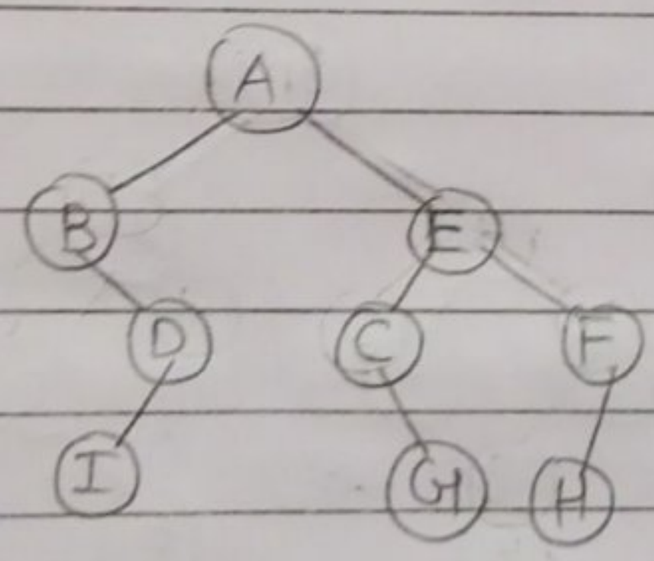
(b) In-order : B, I, D, A, C, G, E, H, F
Post-order : I, D, B, G, C, H, F, E, A

→ Here, the root node is 'A' as it is post-order.

→ B, I, D are the leftmost child and C, G, E, H, F are the rightmost child of A.



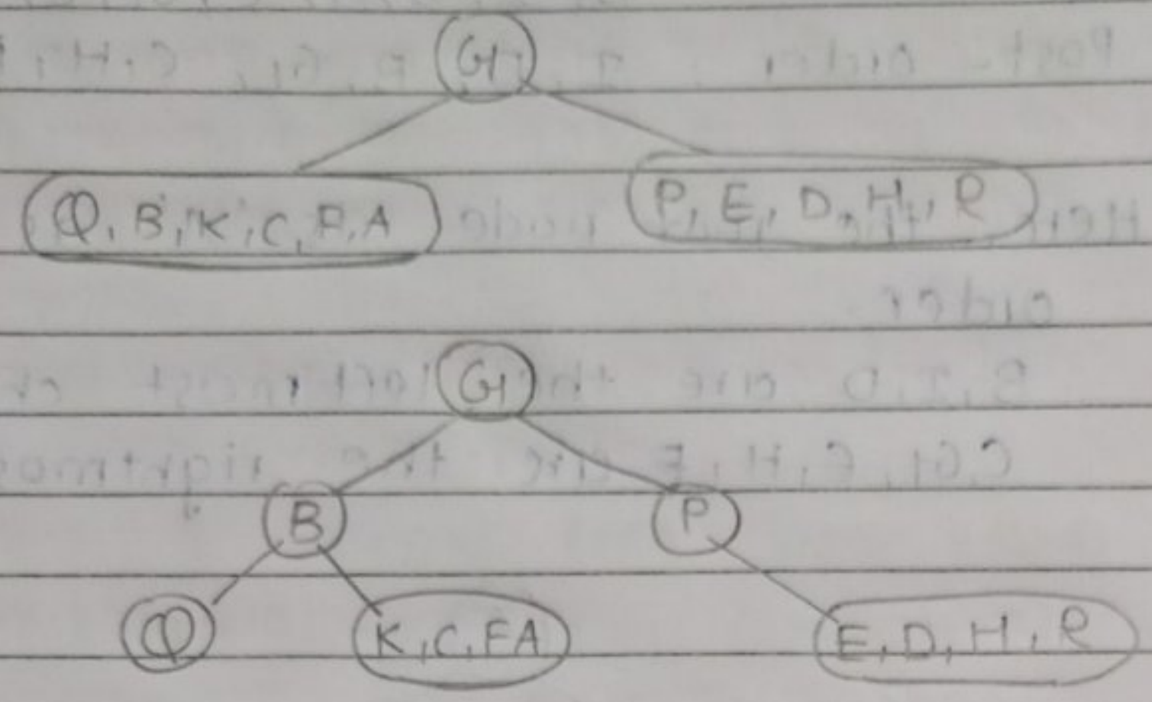


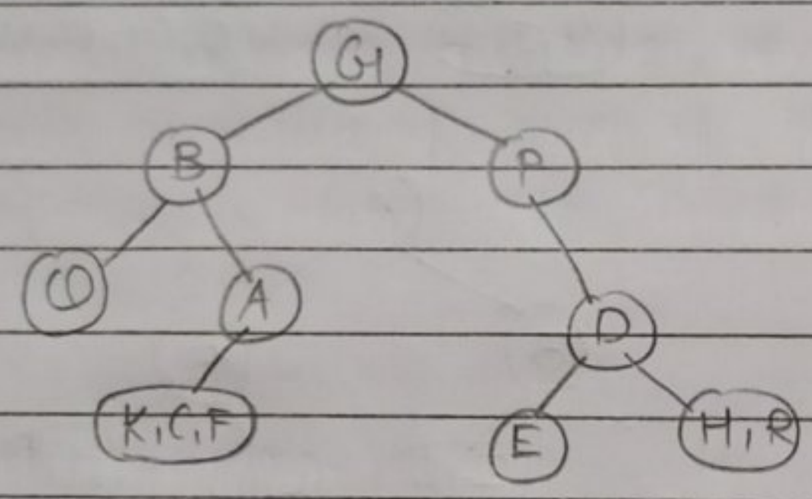


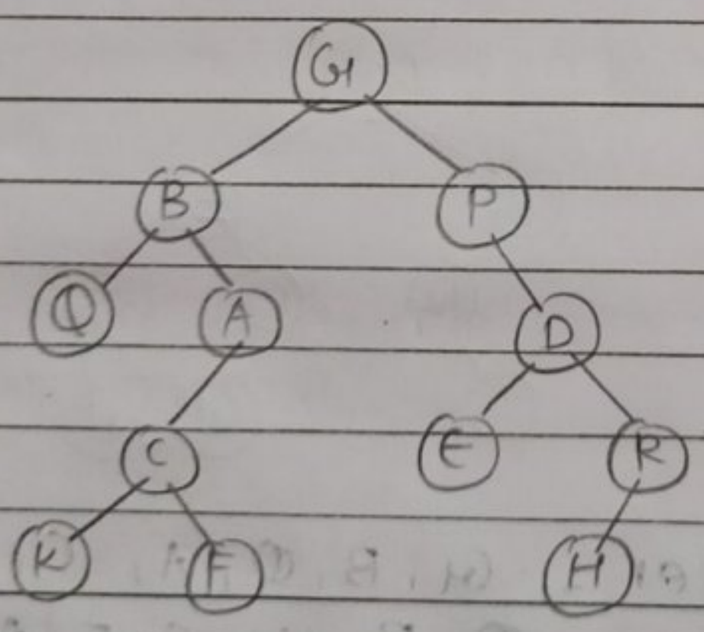
(c) Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H
In-order : Q, B, K, C, F, A, G, P, E, D, H, R.

→ Here, the root node is 'G' as it is Post order

→ Q, B, K, C, F, A are the leftmost child and P, E, D, H, R are the rightmost child of G.







(9) Search the number 50 from the given data using binary search technique. Illustrate the searching process.
10, 14, 20, 39, 41, 45, 49, 50, 60.

- Binary Search :-

→ Find the middle element which '41'.

→ Compare 41 with target value (50).

→ Since 50 is greater than 41, we focus on the right half of the data 45, 49, 50, 60

→ Repeat the process with the right half.

→ The middle element of the right half is 50, which matches target value.

→ Number 50 is found at index 7.

10	14	20	39	41	45	49	50	60
0	1	2	3	4	5	6	7	8

(10) Define :-

(1) Acyclic graph :-

→ An acyclic graph is a graph that has no cycles (no closed loops).

→ Acyclic graphs are also known as acyclic diagraphs or directed acyclic graphs when considering directed edges.

→ A tree data structure is an acyclic graph.

(2) leaf node:-

→ In a tree, a leaf node (also called terminal node) is a node that has no children. (it is the end of a branch).

→ In a family tree, the individuals at the tips are leaf nodes.

(3) Complete Binary tree:-

→ A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes, which are filled from left to right.

(4) connected graph:-

→ A connected graph is a graph in which there is a path connecting any two vertices in the graph.

→ A graph that is not connected is called a disconnected graph.

(5) Threaded Tree:-

→ A threaded binary tree is a binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor.

→ Threads allow traversal of the tree without using recursion or a stack.

(6) Degree of node :-

→ The degree of a node in a graph is the number of edges incident to that node.

→ In a tree, the degree of a node is the number of its children.

(7) M-ary tree :-

→ An M-ary tree is a tree in which each internal node has exactly M children.

→ (8) Null Graph :-

→ A null graph (or empty graph) is a graph with no vertices (and therefore no edges).

(9) Weighted Graph :-

→ A weighted graph (or network) is a graph in which a weight is assigned to each edge

→ These weights can represent distances, costs, capacities, etc.

(10) Adjacent Nodes :-

→ In a graph, two nodes (vertices) are adjacent if there is an edge connecting them.

→ Adjacent nodes are also called neighbors.

(11) Mixed Graphs :-

→ A mixed graph is a graph that can have both directed & edges and undirected edges.

→ It allows loops and multiple edges between the same pair of vertices.

(12) Loop (sling) :-

→ A loop or sling is an edge that connects a vertex to itself.

→ It forms a cycle of length 1.

(13) Forest :-

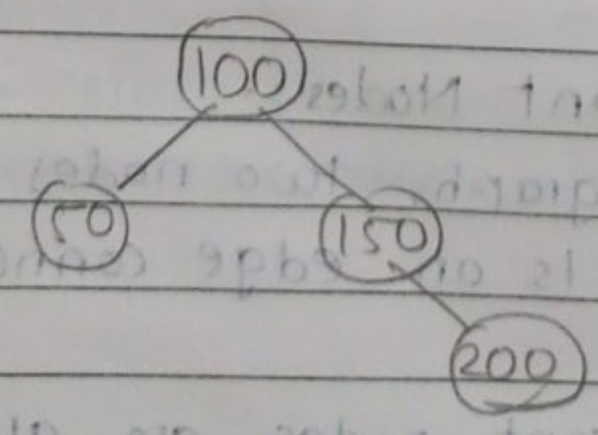
→ A Forest is a collection of disjoint trees. (set of trees with no common vertices)

→ Eg. A collection of family trees (each family tree is a separate tree) forms a forest.

(11) Create 2-3 Tree for the following sequence; 50, 100, 150, 200.

→ Start with an empty 2-3 tree.

→ Insert each value from the sequence one by one.



→ The root node contains the value 100.

→ The left child of the root contains the value 50.

→ The right chid of the root contains the value 150.

→ The right child of 150 contains the value 200.

(12) Discuss Threaded Binary tree with proper example.

→ A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor.

→ Threads allow traversal of the tree without using recursion or a stack.

→ Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal.

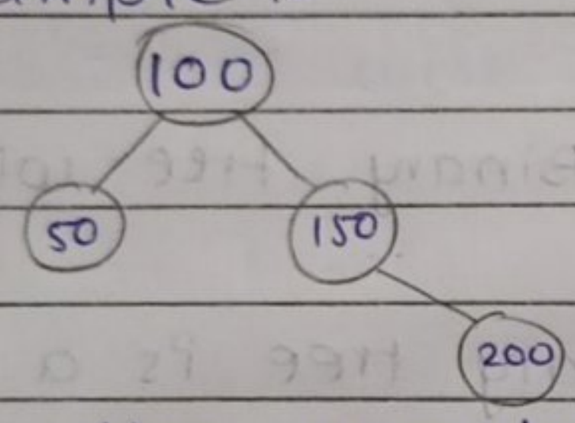
Scanned with OKEN Scanner

- Types of threaded Binary trees:-

→ Single - Threaded :- In this type, the null right pointers are made to point to the in-order successor

→ Double- Threaded - In this type, both left and right Null pointers are made to point to the in- order predecessor and successor, respectively.

- Example :-



→ In the threaded tree, the null right pointer of 150 points to its in-order successor, which is 200.

(13) Construct 5 order (5 way) three from the following data

1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25.

(1) Insert 1

1

(2) Insert 7

1 7

(3) Insert 6

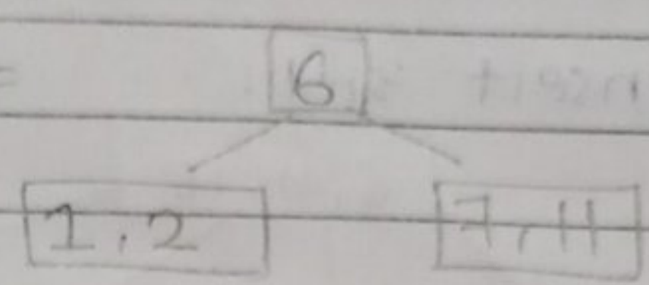
1, 6, 7

(4) Insert 2

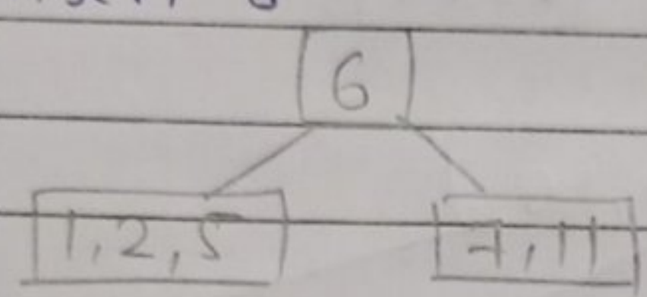
1, 2, 6, 7

(5) Insert 11

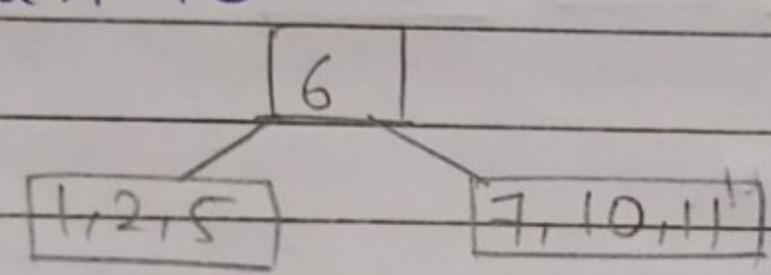
1, 2, 6, 7, 11 =



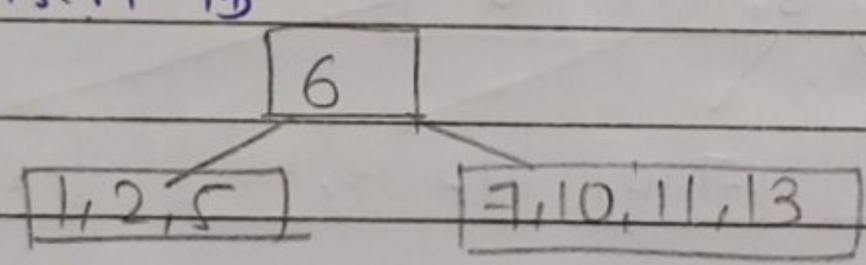
(6) Insert 5



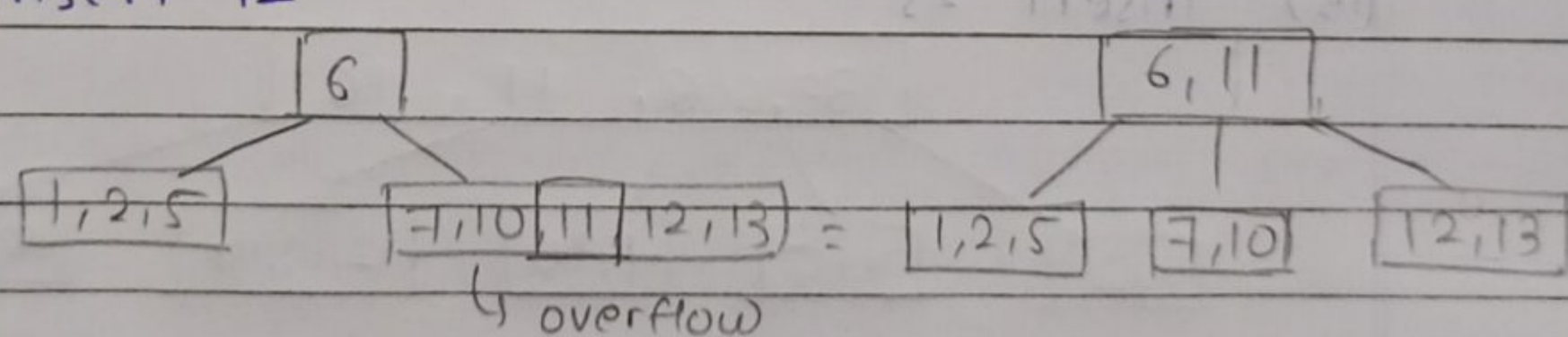
(7) Insert 10



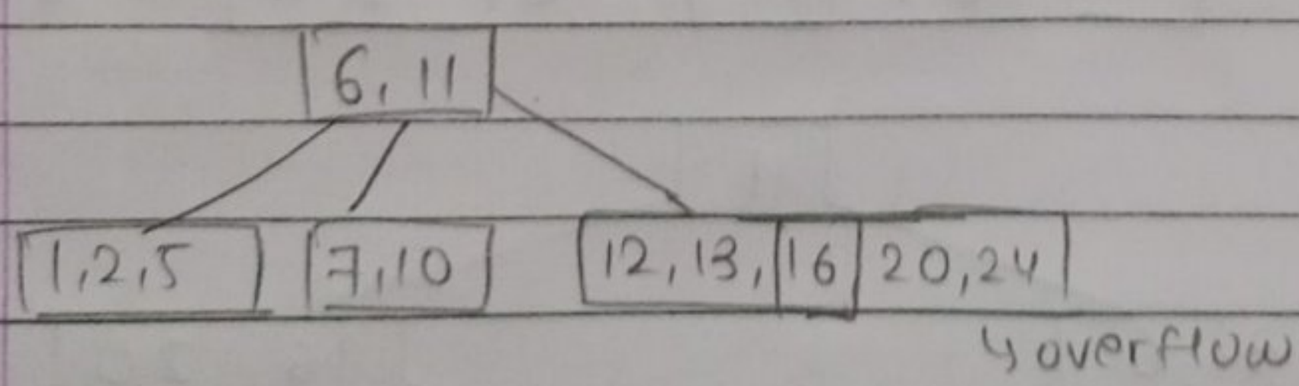
(8) Insert 13

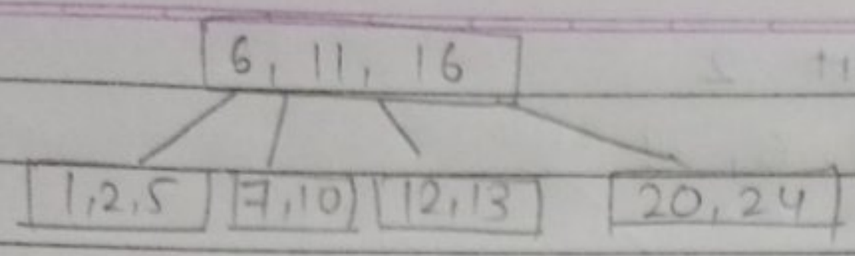


(9) Insert 12

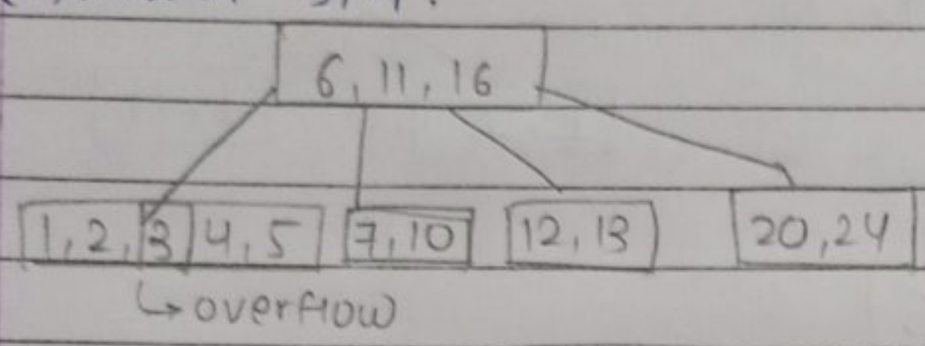


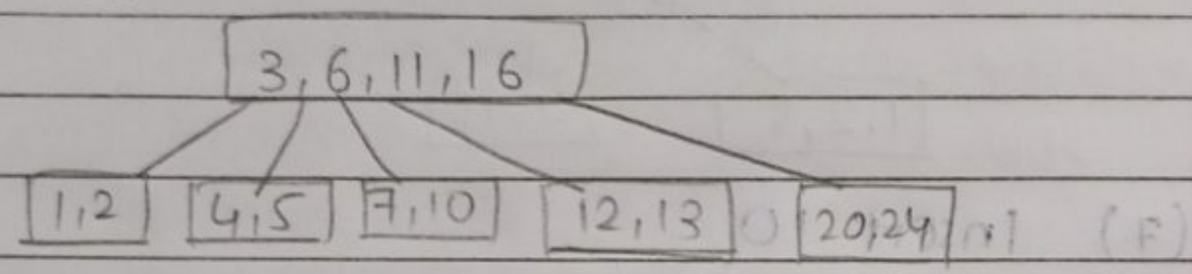
(10) Insert 20, 16, 24.



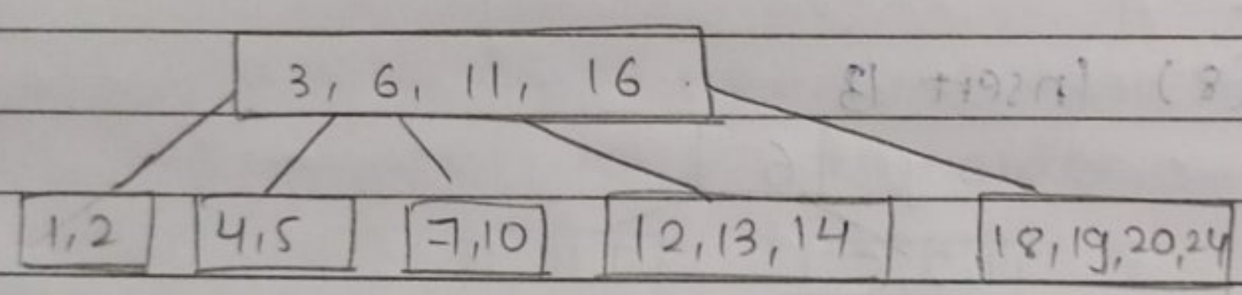


(11) Insert 3,4.

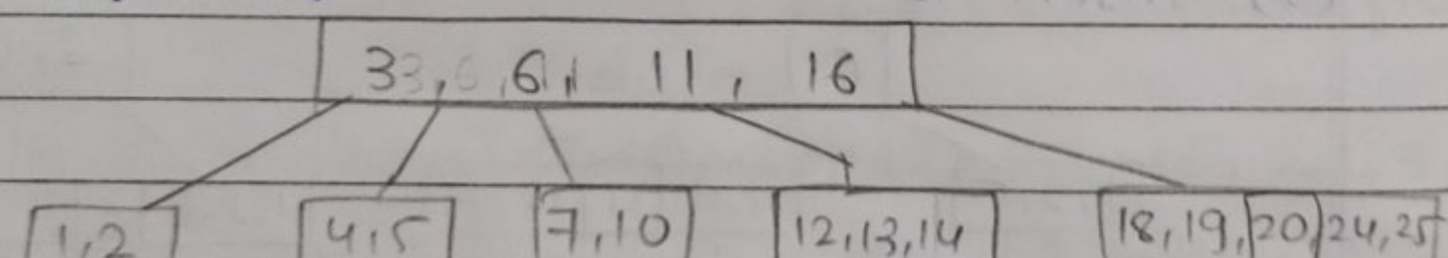


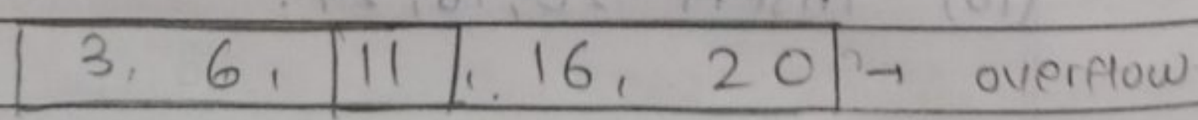


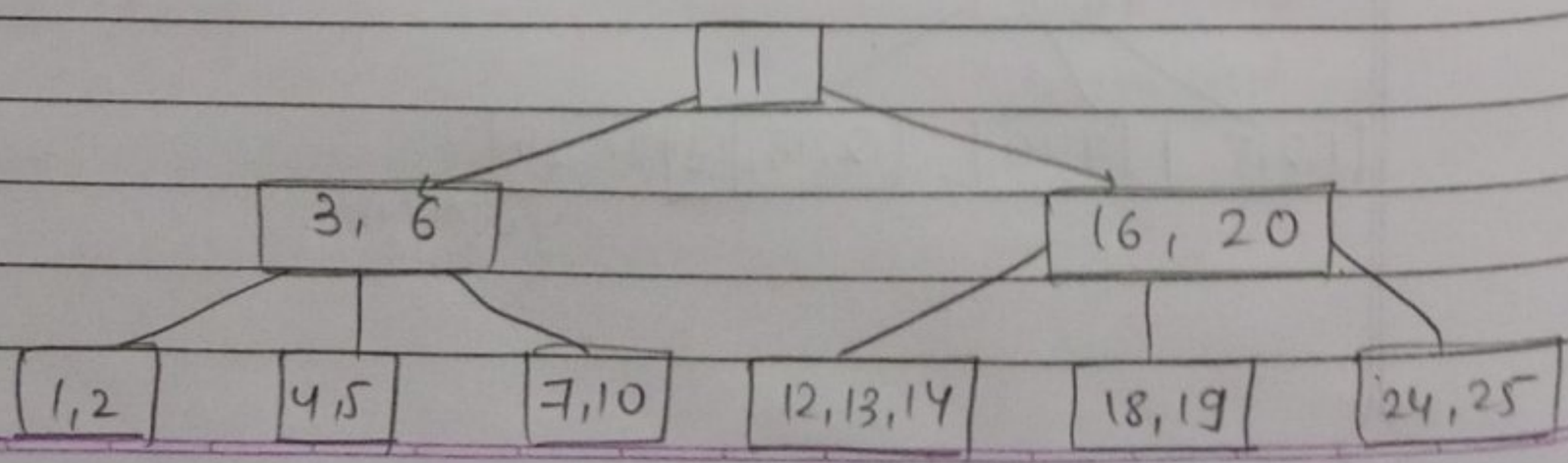
(12) Insert 18, 19, 24.



(13) Insert 25.







(14) Expla
exan

→ A
edge
dest

(mea

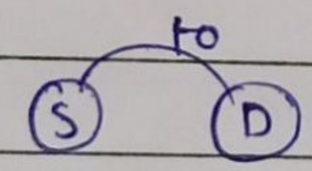
→ Exam

(15) which
the
both

→ Th
a gr
and
stack

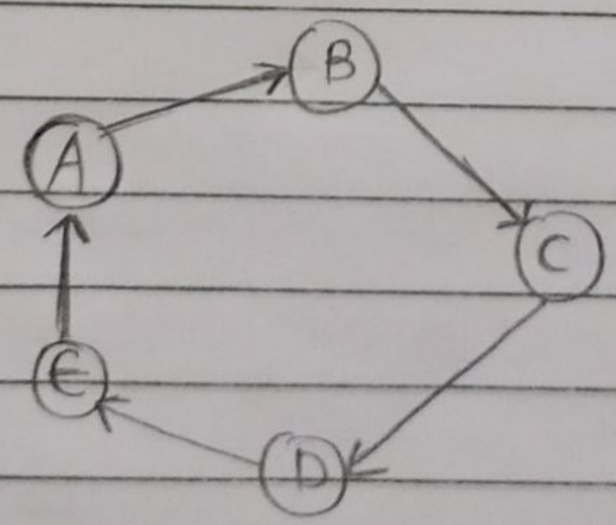
(14) Explain path of the graph with suitable example.

→ A path is a sequence of consecutive edge from the source node ~~or~~ to destination node.

(means 

take edges arrows cycle)

→ Example:-



Path of G is

A → B → C → D → E