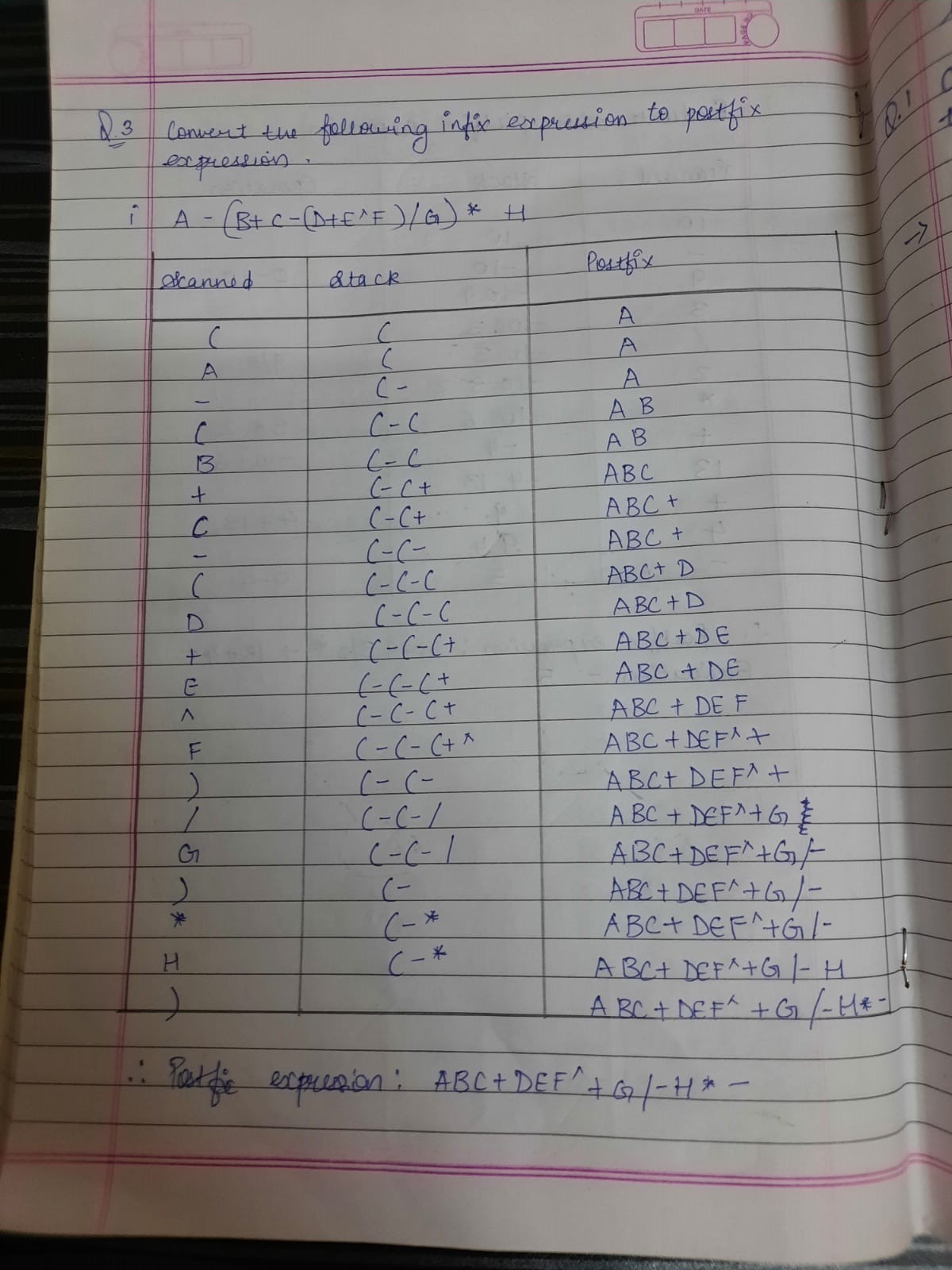
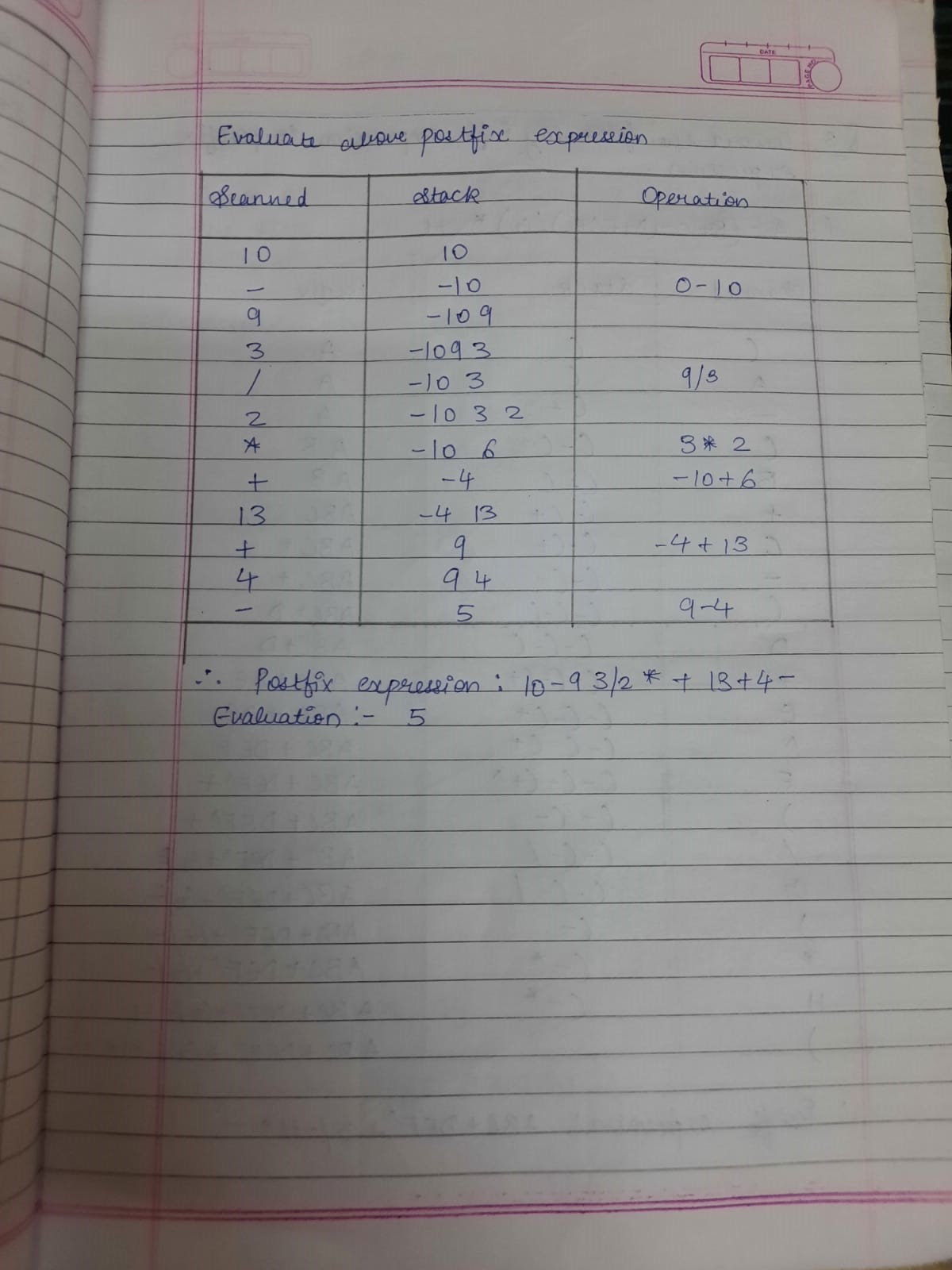
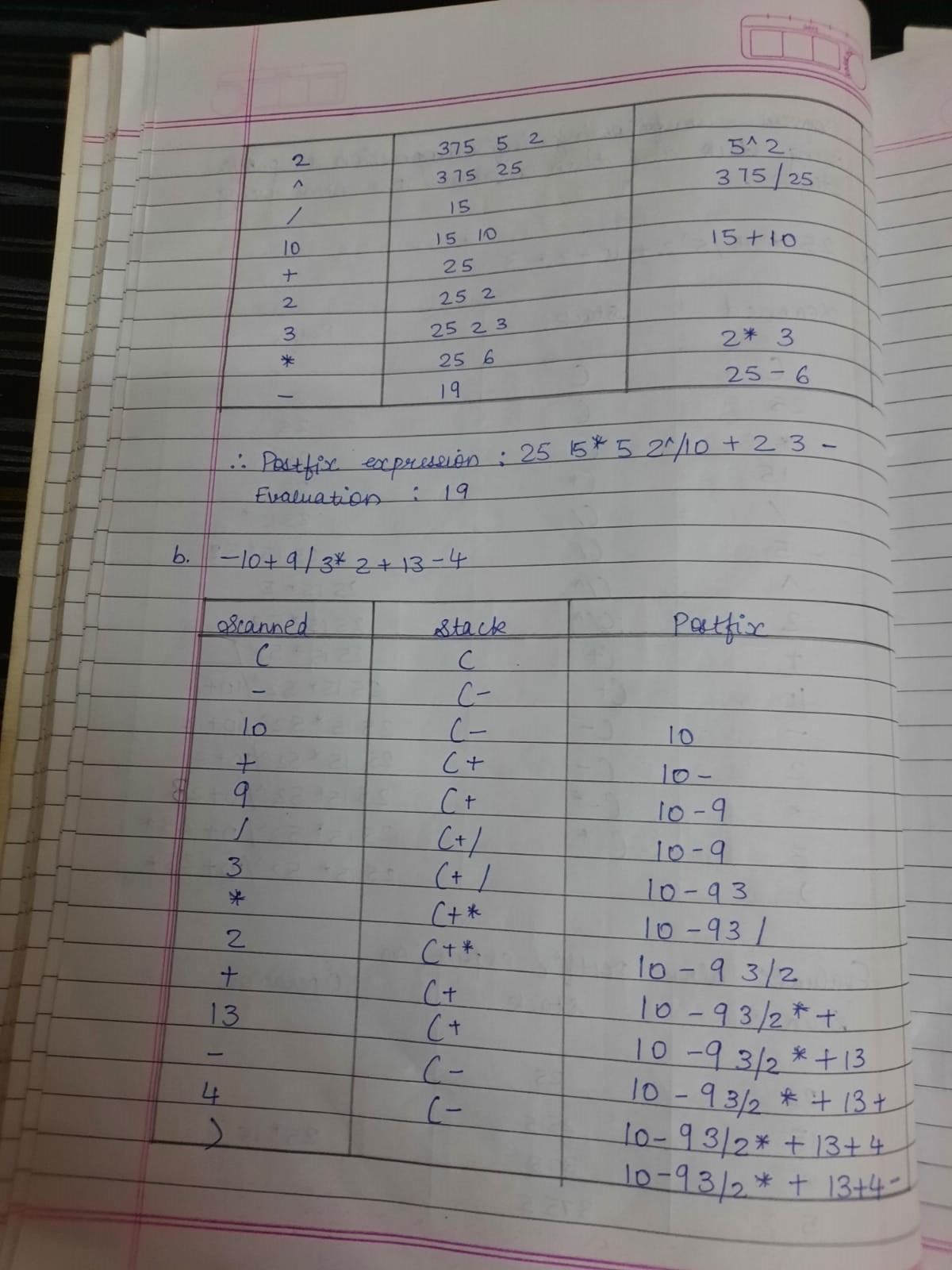
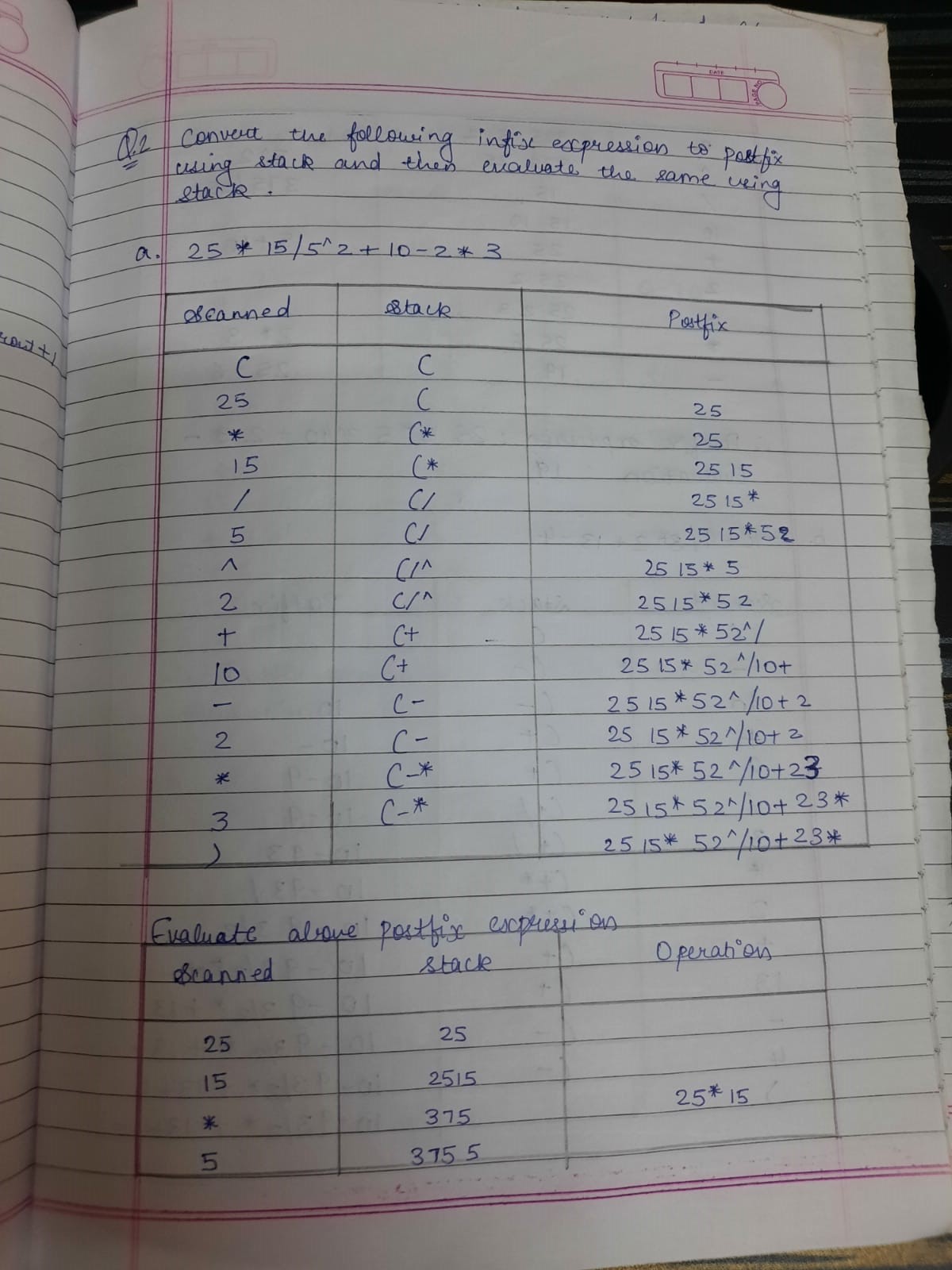
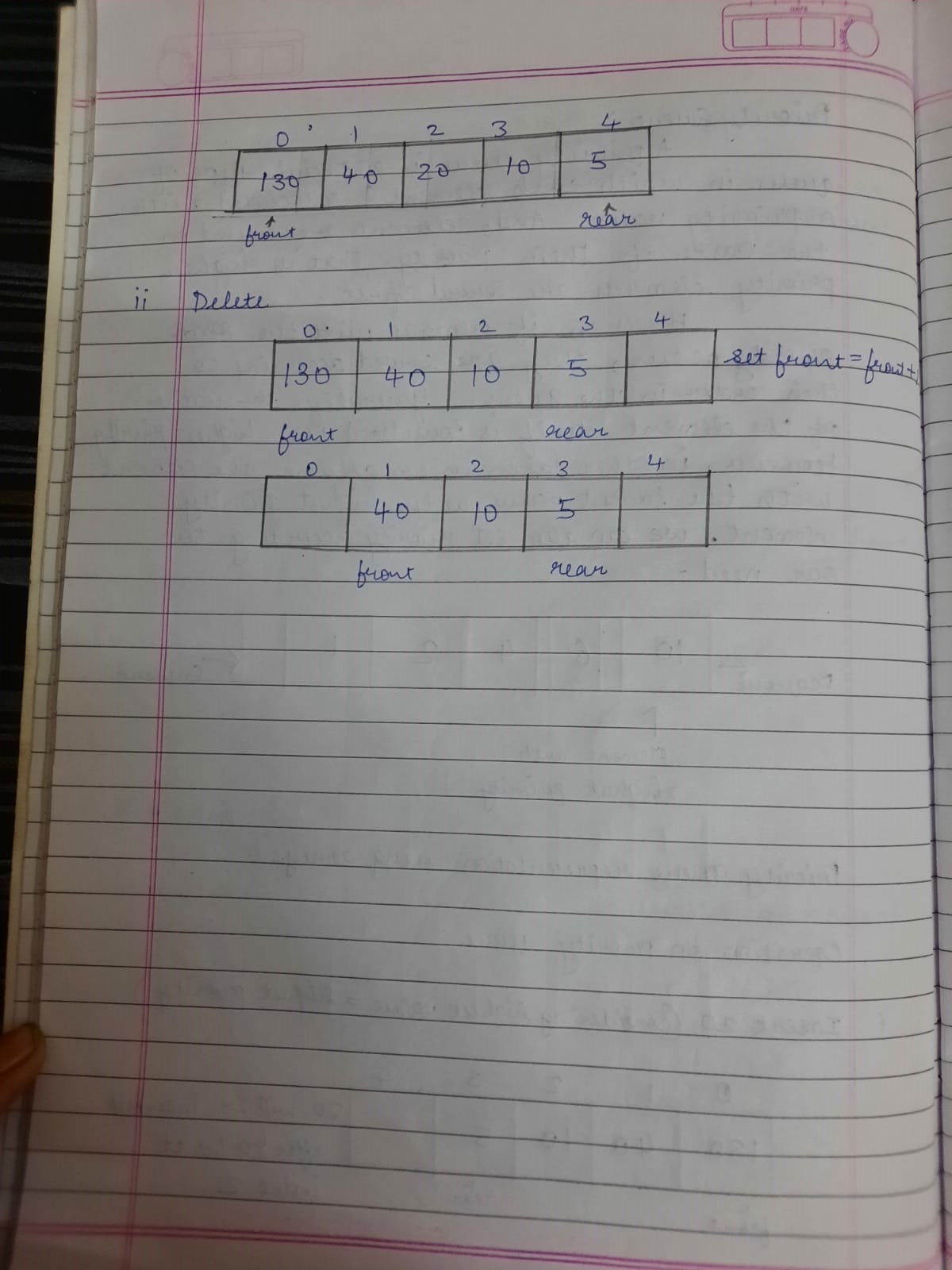
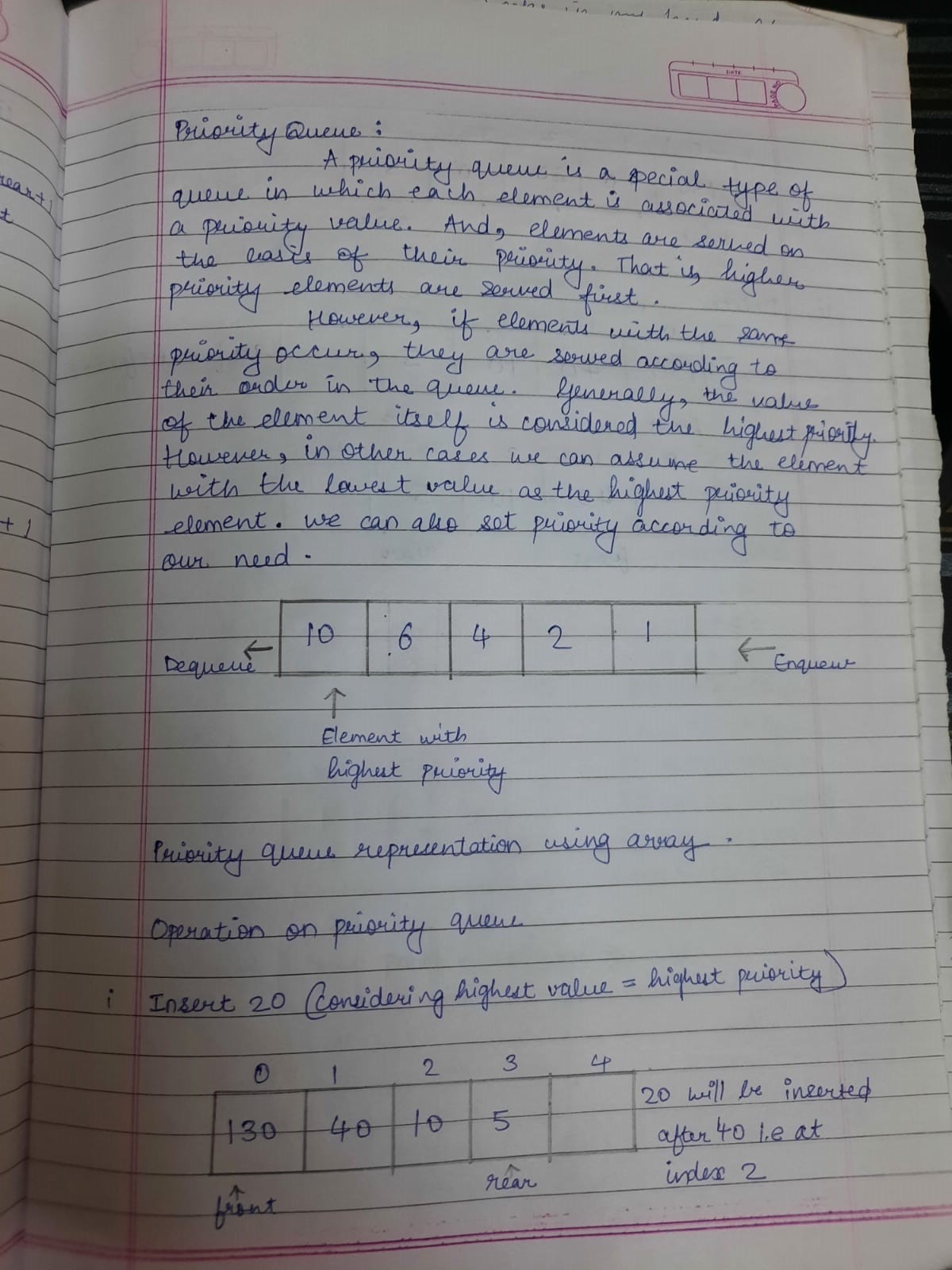
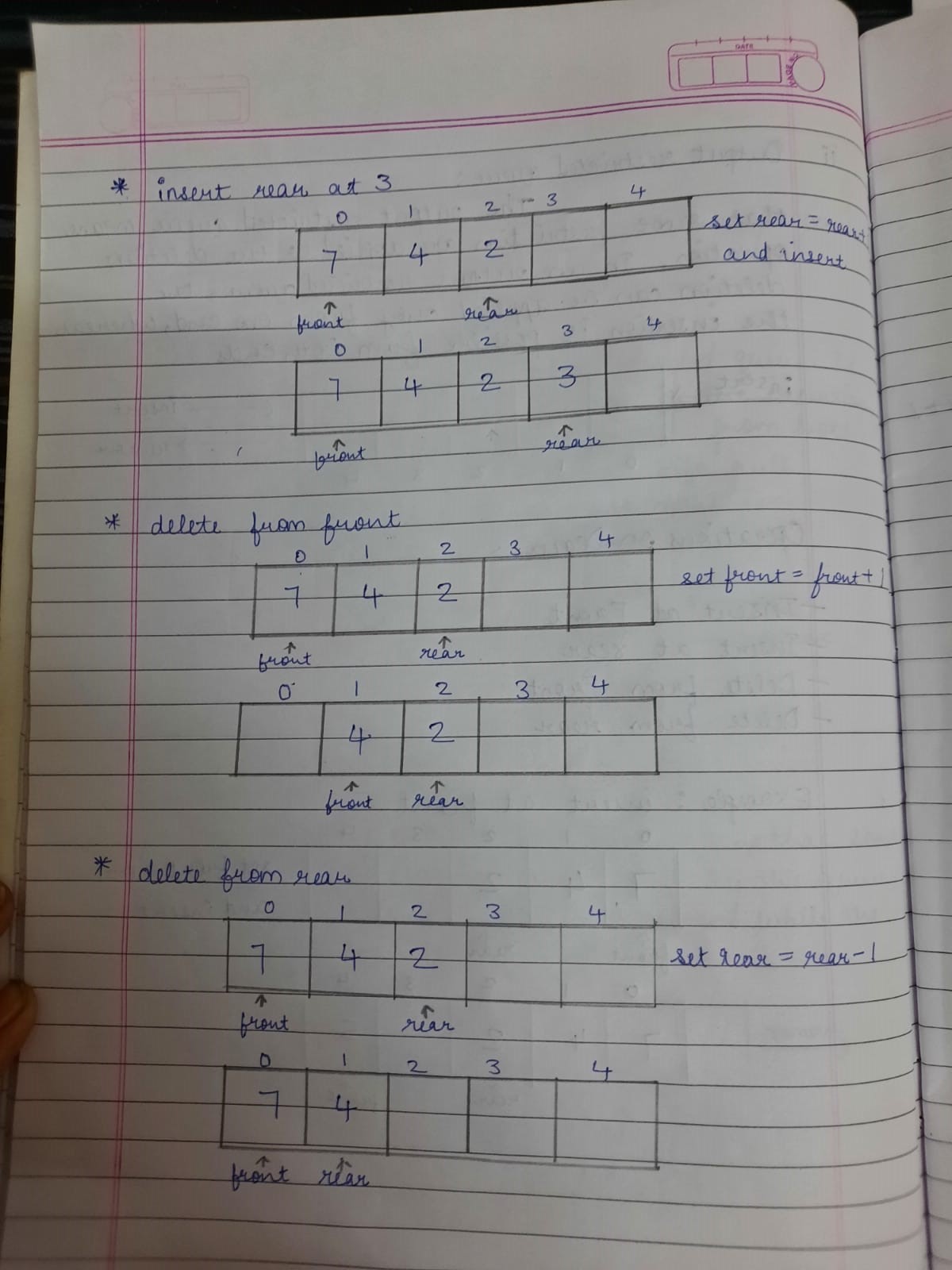
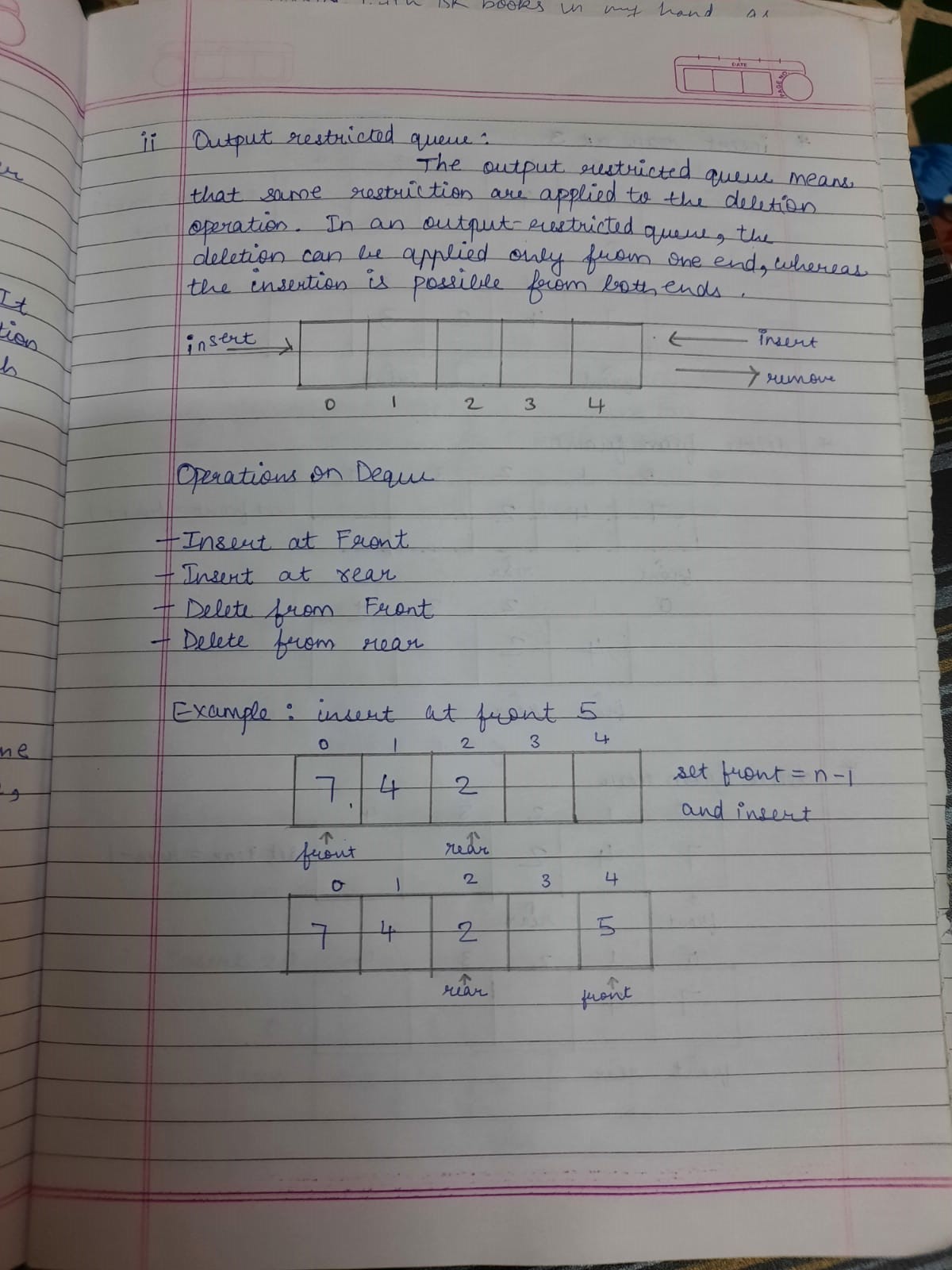
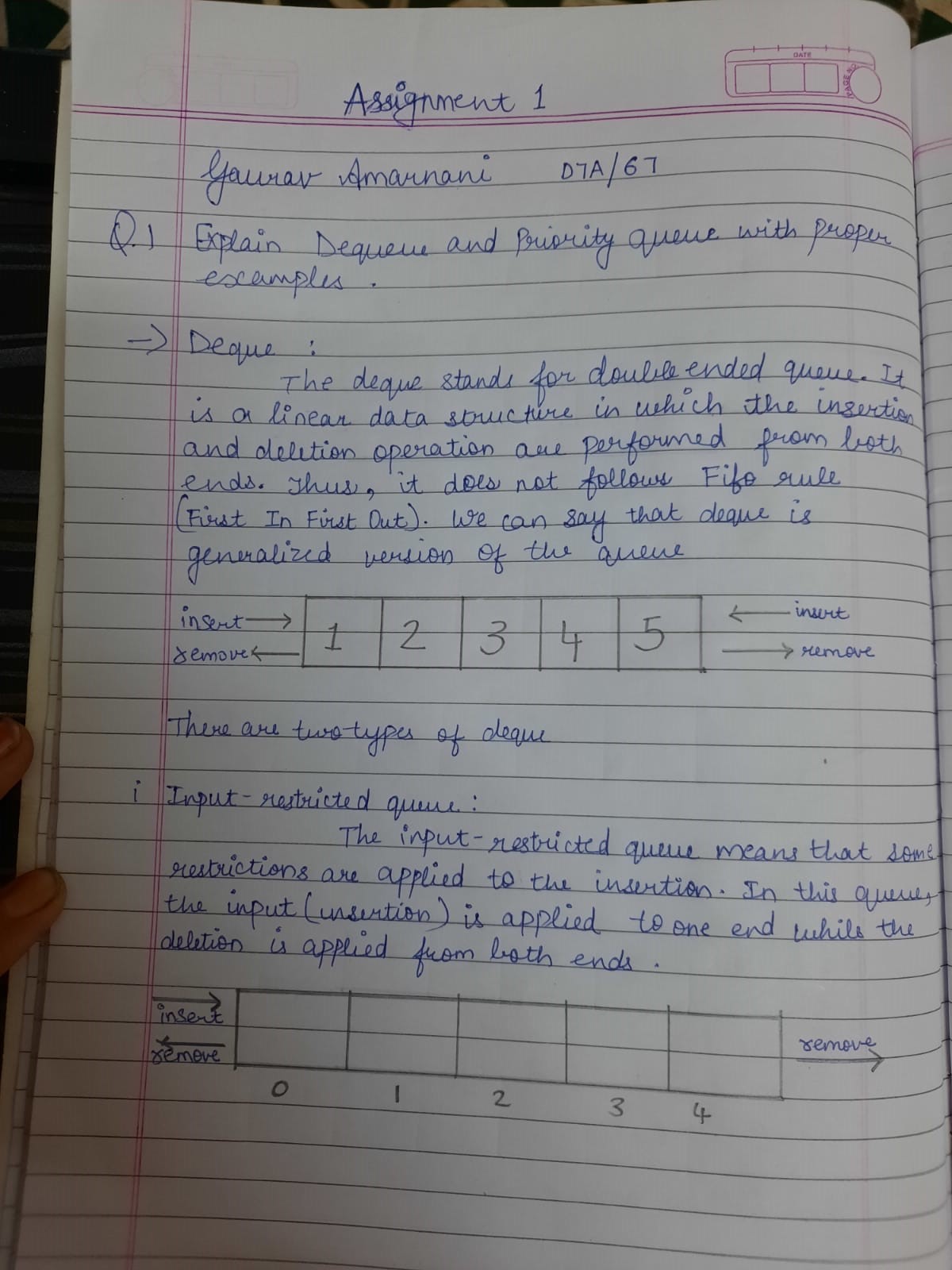
**DS ASSIGNMENT 01.**



# Q4. Implement a Singly linked list. Program:

#include <stdio.h> #include <stdlib.h> #include <conio.h> #include <malloc.h> struct node{

int data;

struct node \*next;

};

struct node \*start = NULL;

struct node \*create\_ll(struct node \*); struct node \*display(struct node \*); struct node \*insert\_beg(struct node \*); struct node \*insert\_end(struct node \*);

struct node \*insert\_before(struct node \*); struct node \*insert\_after(struct node \*); struct node \*delete\_beg(struct node \*); struct node \*delete\_end(struct node \*); struct node \*delete\_node(struct node \*); struct node \*delete\_after(struct node \*); struct node \*delete\_list(struct node \*); struct node \*sort\_list(struct node \*);

void main() {

int option; do{

printf(“\n\n \*\*\*\*\*MAIN MENU \*\*\*\*\*”); printf(“\n 1: Create a list”);

printf(“\n 2: Display the list”);

printf(“\n 3: Add a node at the beginning”); printf(“\n 4: Add a node at the end”);

printf(“\n 5: Add a node before a given node”); printf(“\n 6: Add a node after a given node”); printf(“\n 7: Delete a node from the beginning”); printf(“\n 8: Delete a node from the end”); printf(“\n 9: Delete a given node”);

printf(“\n 10: Delete a node after a given node”); printf(“\n 11: Delete the entire list”);

printf(“\n 12: Sort the list”); printf(“\n 13: EXIT”);

printf(“\n\n Enter your option : “); scanf(“%d”, &option); switch(option){

case 1: start = create\_ll(start);

printf(“\n LINKED LIST CREATED”);

break;

case 2: start = display(start); break;

case 3: start = insert\_beg(start); break;

case 4: start = insert\_end(start); break;

case 5: start = insert\_before(start); break;

case 6: start = insert\_after(start); break;

case 7: start = delete\_beg(start); break;

case 8: start = delete\_end(start); break;

case 9: start = delete\_node(start); break;

case 10: start = delete\_after(start); break;

case 11: start = delete\_list(start);

printf(“\n LINKED LIST DELETED”);

break;

case 12: start = sort\_list(start); break;

}

}while(option !=13); getch();

}

struct node \*create\_ll(struct node \*start){ struct node \*new\_node, \*ptr;

int num;

printf(“\n Enter -1 to end”); printf(“\n Enter the data : “); scanf(“%d”, &num);

while(num!=-1){

new\_node = (struct node\*)malloc(sizeof(struct node)); new\_node -> data=num;

if(start==NULL){

new\_node -> next = NULL; start = new\_node;

}

else{

}

ptr=start;

while(ptr->next!=NULL) ptr=ptr->next;

ptr->next = new\_node; new\_node->next=NULL;

printf(“\n Enter the data : “); scanf(“%d”, &num);

}

return start;

}

struct node \*display(struct node \*start){ struct node \*ptr;

ptr = start; while(ptr != NULL){

printf(“\t %d”, ptr -> data); ptr = ptr -> next;

}

return start;

}

struct node \*insert\_beg(struct node \*start){ struct node \*new\_node;

int num;

printf(“\n Enter the data : “); scanf(“%d”, &num);

new\_node = (struct node \*)malloc(sizeof(struct node)); new\_node -> data = num;

new\_node -> next = start; start = new\_node;

return start;

}

struct node \*insert\_end(struct node \*start){ struct node \*ptr, \*new\_node;

int num;

printf(“\n Enter the data : “); scanf(“%d”, &num);

new\_node = (struct node \*)malloc(sizeof(struct node)); new\_node -> data = num;

new\_node -> next = NULL; ptr = start;

while(ptr -> next != NULL) ptr = ptr -> next;

ptr -> next = new\_node; return start;

}

struct node \*insert\_before(struct node \*start){ struct node \*new\_node, \*ptr, \*preptr; int num, val;

printf(“\n Enter the data : “); scanf(“%d”, &num);

printf(“\n Enter the value before which the data has to be inserted : “); scanf(“%d”, &val);

new\_node = (struct node \*)malloc(sizeof(struct node)); new\_node -> data = num;

ptr = start;

while(ptr -> data != val){ preptr = ptr;

ptr = ptr -> next;

}

preptr -> next = new\_node; new\_node -> next = ptr; return start;

}

struct node \*insert\_after(struct node \*start){ struct node \*new\_node, \*ptr, \*preptr; int num, val;

printf(“\n Enter the data : “); scanf(“%d”, &num);

printf(“\n Enter the value after which the data has to be inserted : “); scanf(“%d”, &val);

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node -> data = num; ptr = start;

preptr = ptr;

while(preptr -> data != val){ preptr = ptr;

ptr = ptr -> next;

}

preptr -> next=new\_node; new\_node -> next = ptr; return start;

}

struct node \*delete\_beg(struct node \*start){ struct node \*ptr;

ptr = start;

start = start -> next; free(ptr);

return start;

}

struct node \*delete\_end(struct node \*start){ struct node \*ptr, \*preptr;

ptr = start;

while(ptr -> next != NULL){ preptr = ptr;

ptr = ptr -> next;

}

preptr -> next = NULL; free(ptr);

return start;

}

struct node \*delete\_node(struct node \*start){ struct node \*ptr, \*preptr;

int val;

printf(“\n Enter the value of the node which has to be deleted : “); scanf(“%d”, &val);

ptr = start;

if(ptr -> data == val){

start = delete\_beg(start); return start;

}

else{

while(ptr -> data != val){ preptr = ptr;

ptr = ptr -> next;

}

preptr -> next = ptr -> next; free(ptr);

return start;

}

}

struct node \*delete\_after(struct node \*start){ struct node \*ptr, \*preptr;

int val;

printf(“\n Enter the value after which the node has to deleted : “); scanf(“%d”, &val);

ptr = start; preptr = ptr;

while(preptr -> data != val){ preptr = ptr;

ptr = ptr -> next;

}

preptr -> next=ptr -> next; free(ptr);

return start;

}

struct node \*delete\_list(struct node \*start){ struct node \*ptr;

if(start!=NULL){

ptr=start;

while(ptr != NULL){

printf(“\n %d is to be deleted next”, ptr -> data); start = delete\_beg(ptr);

ptr = start;

}

}

return start;

}

struct node \*sort\_list(struct node \*start){ struct node \*ptr1, \*ptr2;

int temp; ptr1 = start;

while(ptr1 -> next != NULL){ ptr2 = ptr1 -> next; while(ptr2 != NULL){

if(ptr1 -> data > ptr2 -> data){ temp = ptr1 -> data;

ptr1 -> data = ptr2 -> data; ptr2 -> data = temp;

}

ptr2 = ptr2 -> next;

}

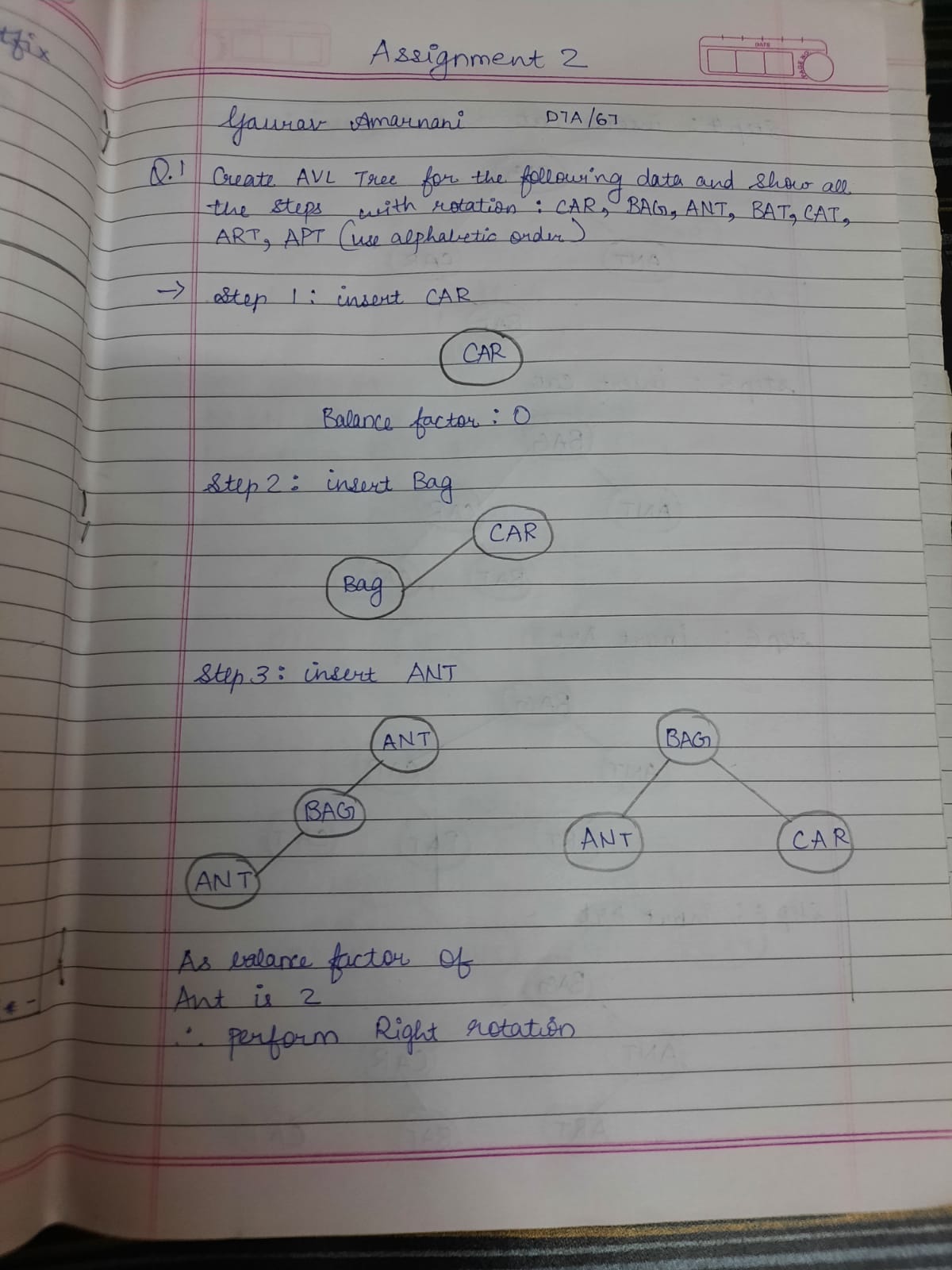
ptr1 = ptr1 -> next;

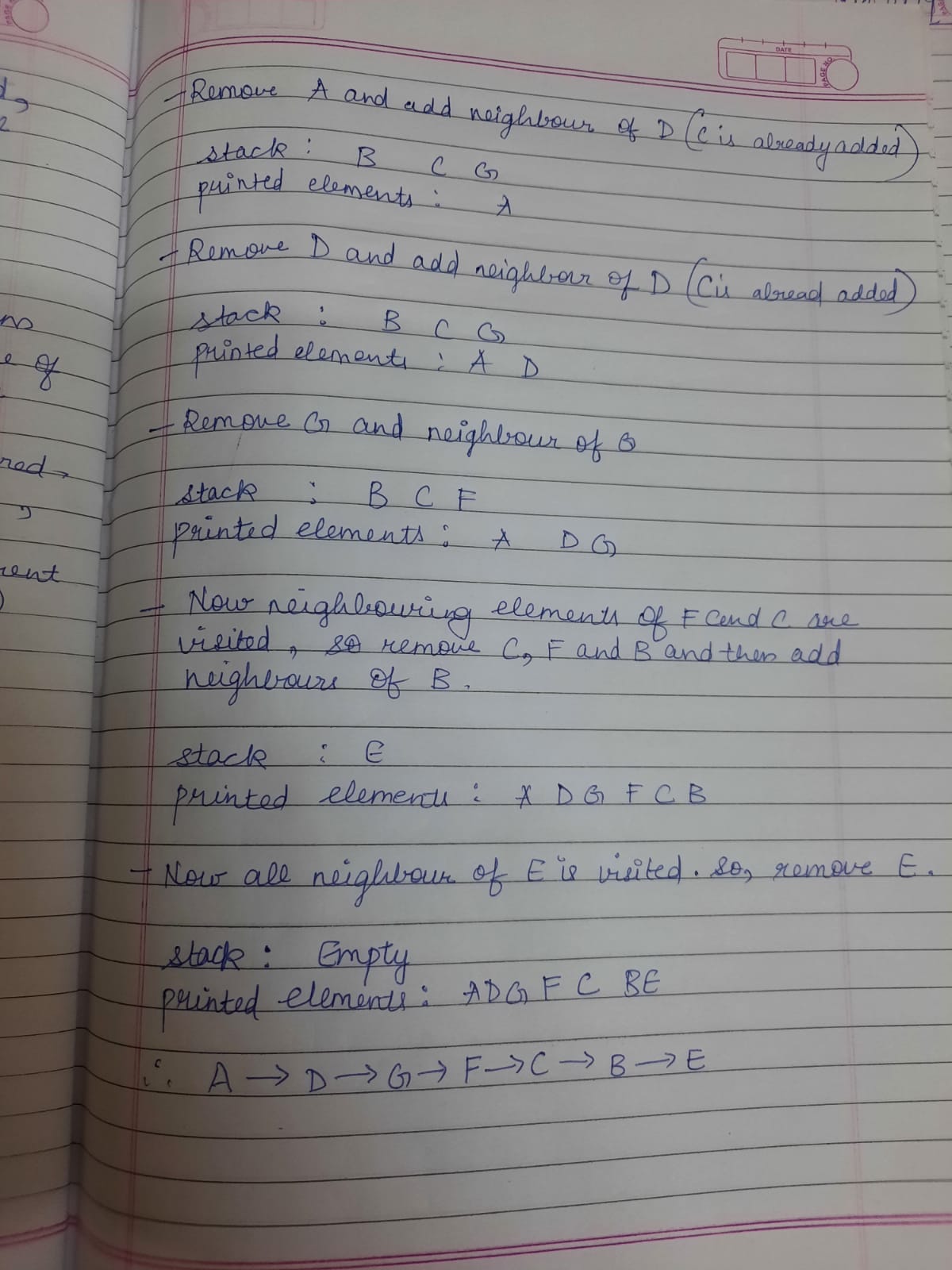
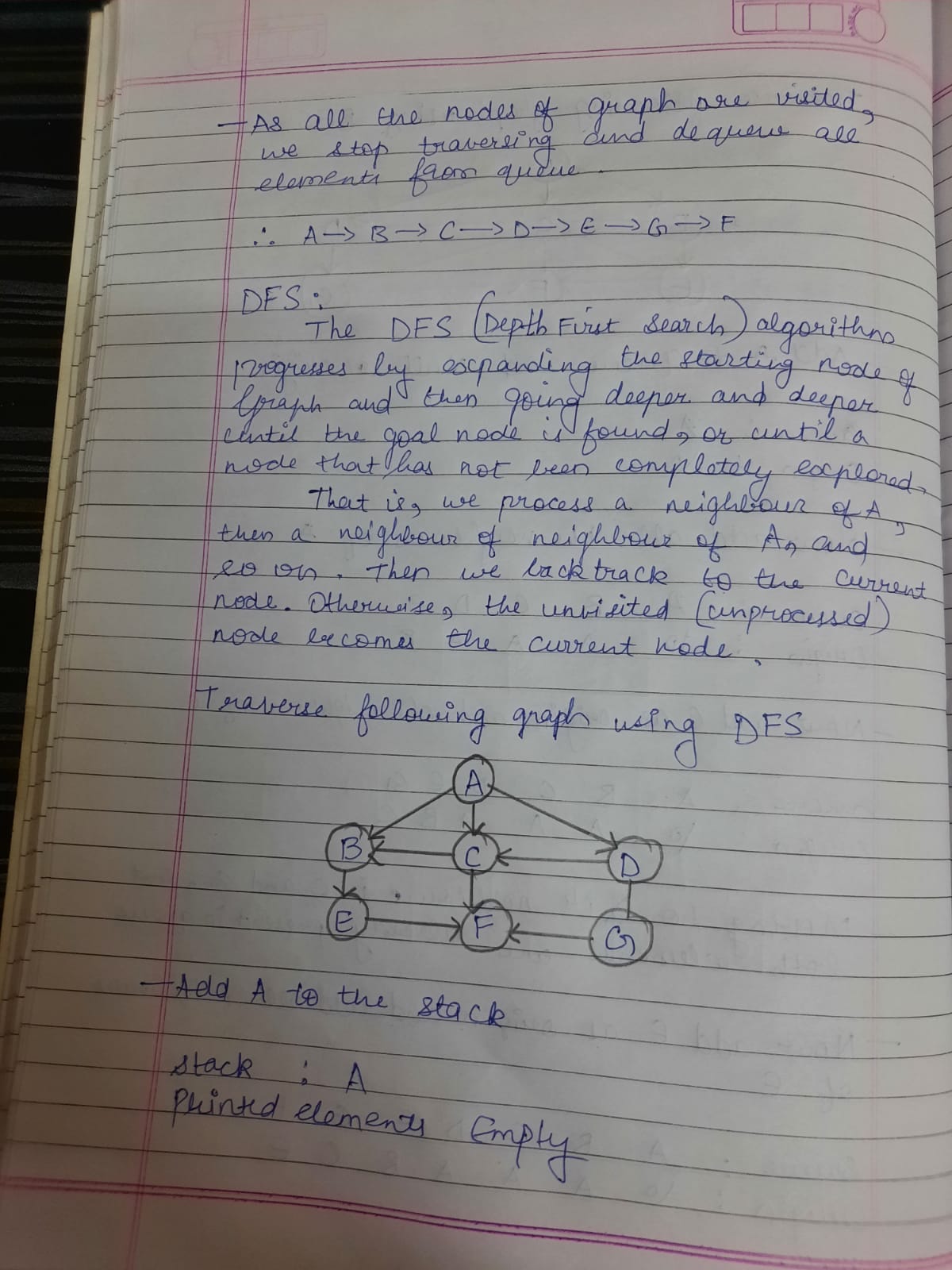
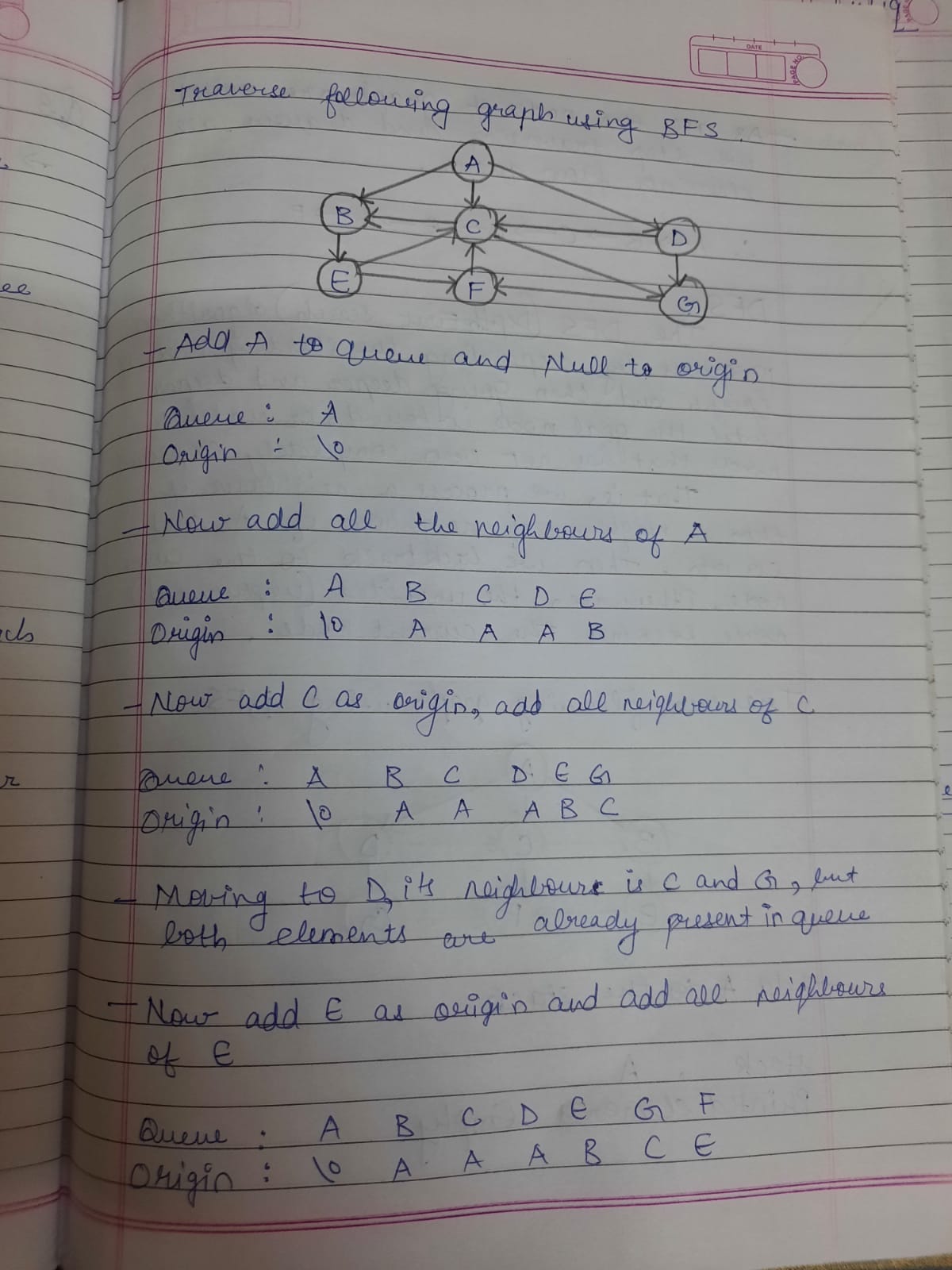
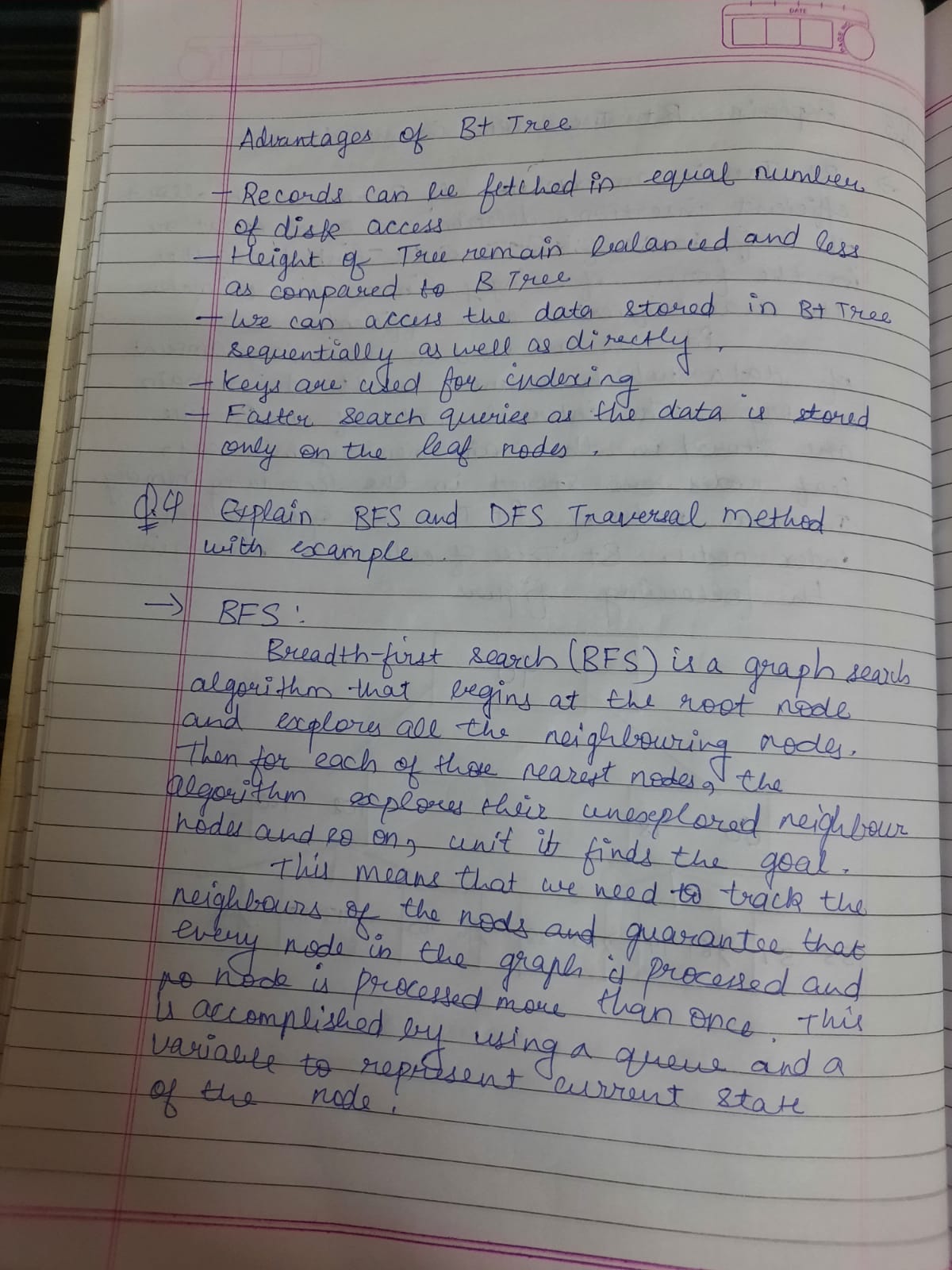
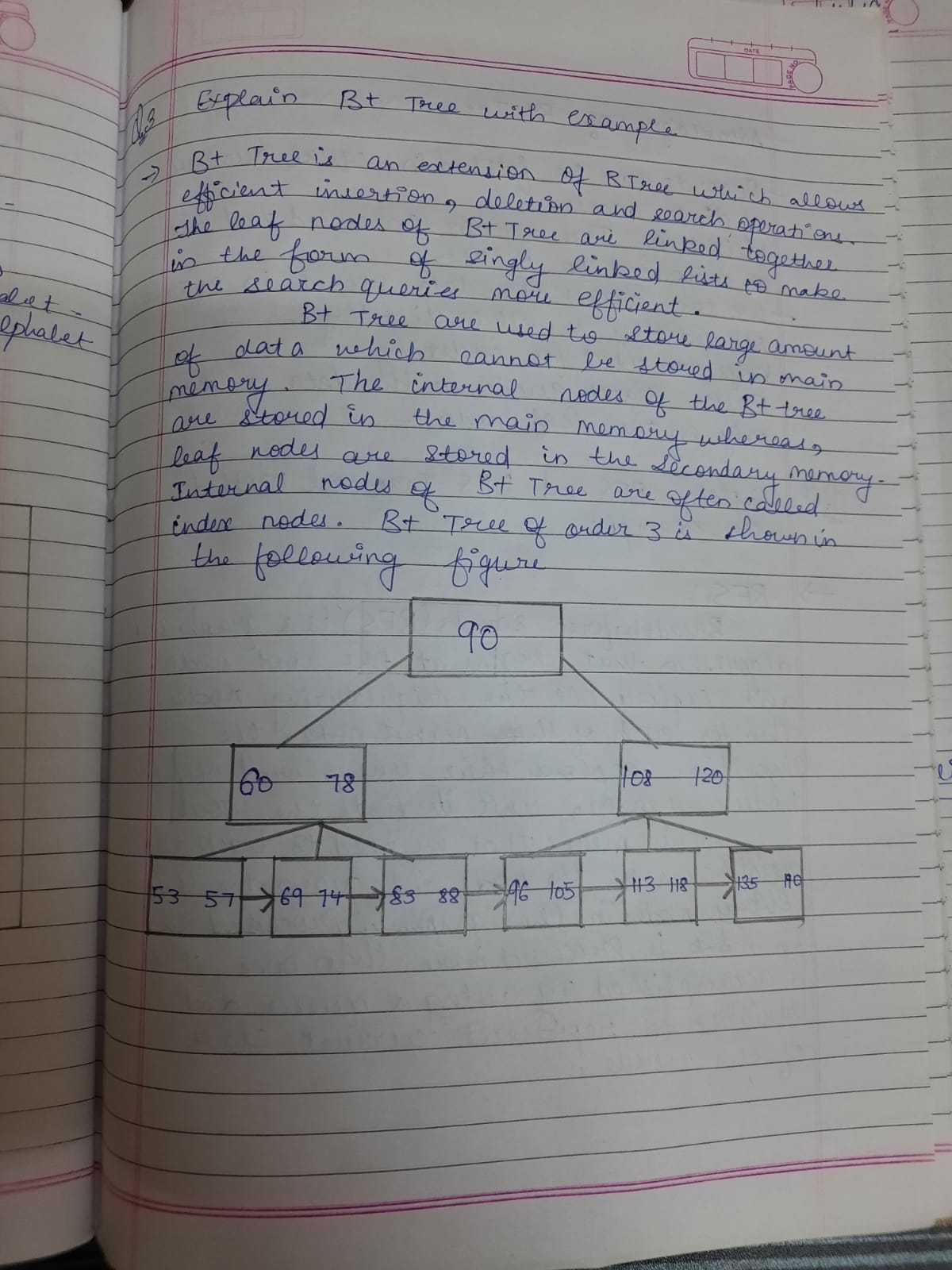
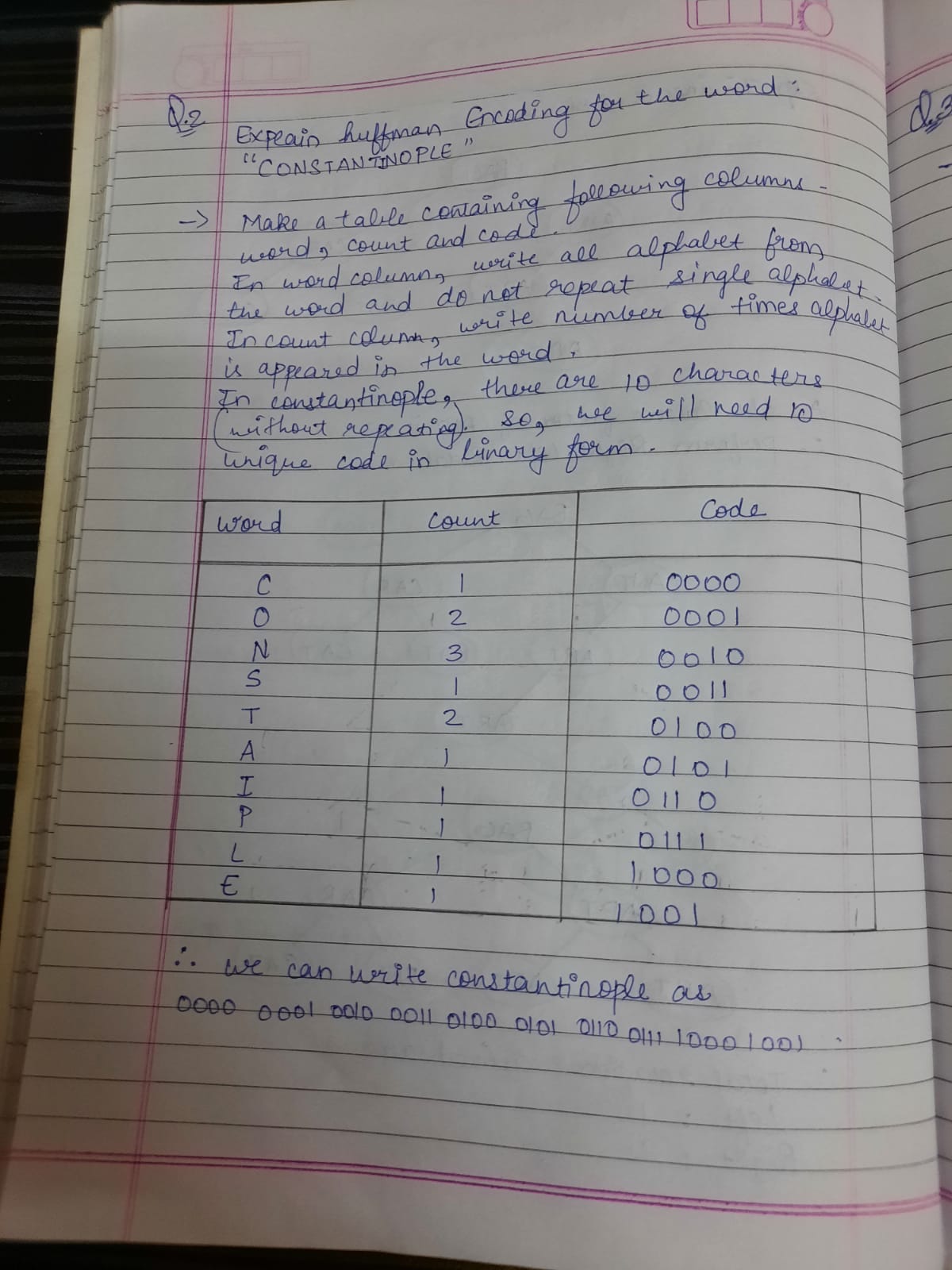
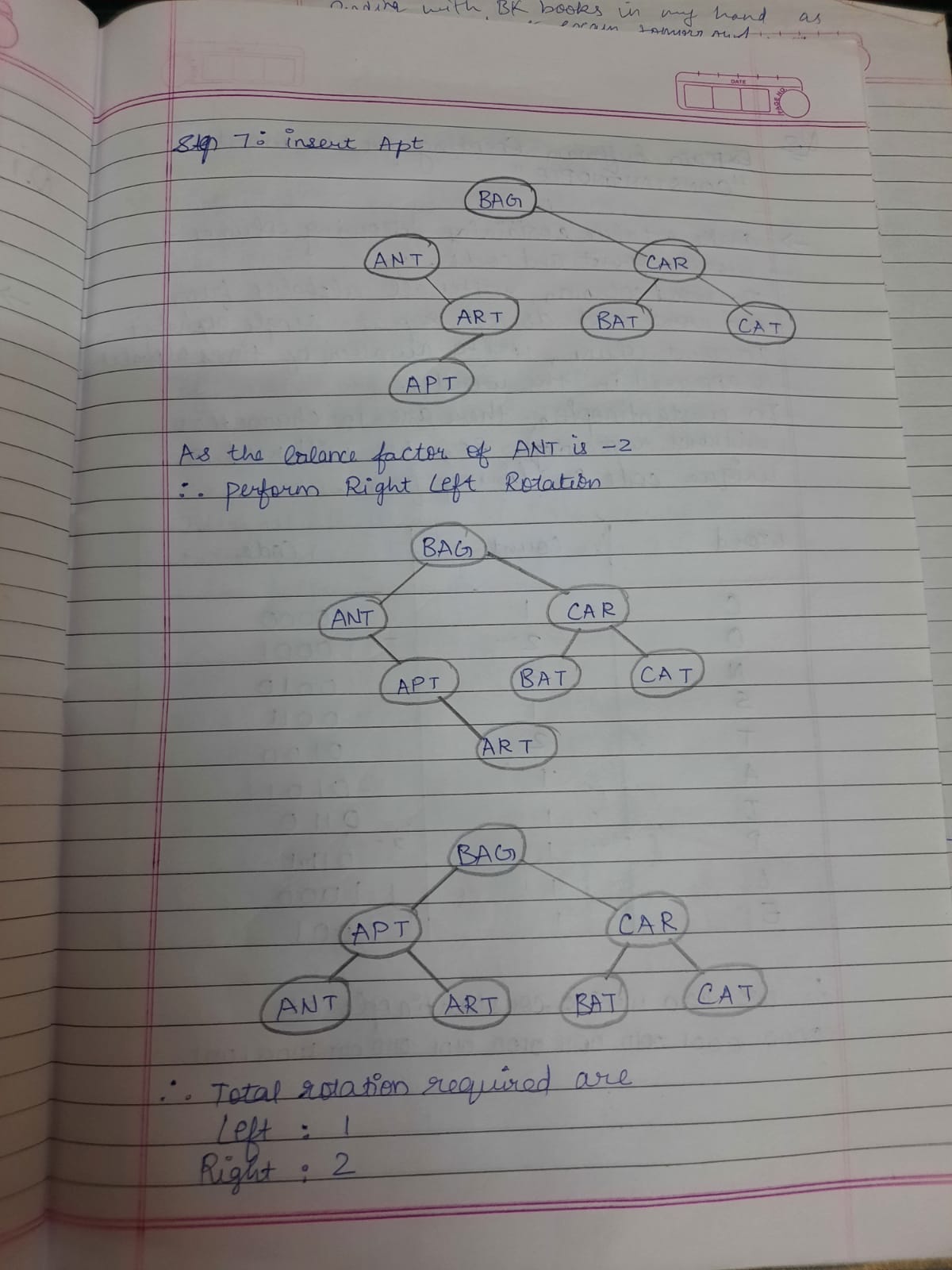
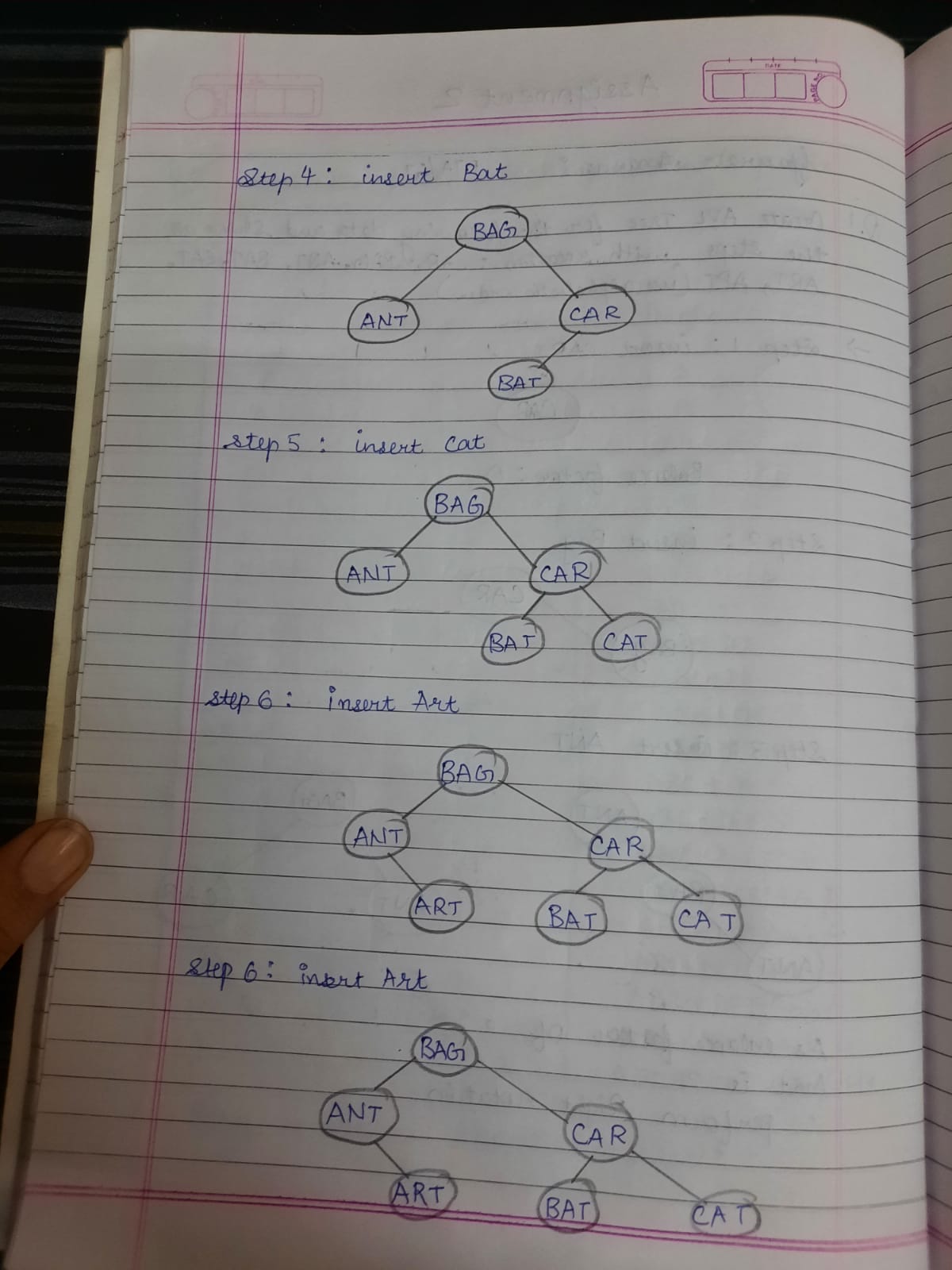
}

return start;

}

**DS ASSIGNMENT 02.**





**DS ASSIGNMENT 03.**

**Applications of:**

## Array:

* Maintains multiple variable names using a single name. Arrays help to maintain large data under a single variable name. This avoids the confusion of using multiple variables.
* Arrays can be used for sorting data elements. Different sorting techniques like Bubble sort, Insertion sort, Selection sort etc use arrays to store and sort elements easily.
* Arrays can be used for performing matrix operations. Many databases, small and large, consist of one-dimensional and two-dimensional arrays whose elements are records.
* Arrays can be used for CPU scheduling.
* Lastly, arrays are also used to implement other data structures like Stacks, Queues, Heaps, Hash tables etc.

## Linked List

**Singly Linked List:**

* It is used to implement stacks and queues which are like fundamental needs throughout computer science.
* To prevent the collision between the data in the hash map, we use a singly linked list.
* If we ever noticed the functioning of a casual notepad, it also uses a singly linked list to perform undo or redo or deleting functions.

## Doubly Linked List:

* a doubly linked list is used in navigation systems, as it needs front and back navigation.
* It is easily possible to implement other data structures like a binary tree, hash tables, stack, etc. using doubly linked lists.
* It is used in music playing systems where you can easily play the previous one or next one song as many times as one person wants to. Basically it provides full flexibility to perform functions and make the system user-friendly.

## Circular Linked List:

* Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can easily be found in this.
* It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism
* It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.

## Stack:

* Function-call abstraction. Most programs use stacks implicitly because they support a natural way to implement function calls, as follows: at any point during the execution of a function, define its state to be the values of all of its variables and a pointer to the next instruction to be executed. The natural way to implement the function-call abstraction is to use a stack. To call a function, push the state on a stack. To return from a function call, pop the state from the stack to restore all variables to their values before the function call and resume execution at the next instruction to be executed.
* Arithmetic expression evaluation. An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation.

## Queue:

**Linear Queue:**

* Linear Queues are used for managing requests on a single shared resource such as CPU scheduling and disk scheduling.
* Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
* Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

## Circular queue

* Memory Management: The unused memory locations in the case of ordinary queues can be utilized in circular queues.
* Traffic system: In computer controlled traffic systems, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
* CPU Scheduling: Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

## Priority Queue

* When the graph is stored in the form of an adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra’s algorithm.
* It is used to implement Prim’s Algorithm to store keys of nodes and extract minimum key nodes at every step.
* The A\* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

## Tree

**General tree:**

* Store hierarchical data, like folder structure, organization structure, XML/HTML data.
* If we organize keys in the form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.

## Binary tree:

* Binary trees can also be used for classification purposes. A decision tree is a supervised machine learning algorithm. The binary tree data structure is used here to emulate the decision-making process.
* Another useful application of binary trees is in expression evaluation. In mathematics, expressions are statements with operators and operands that evaluate a value. The leaves of the binary tree are the operands while the internal nodes are the operators.

## AVL tree:

* AVL trees are mostly used for in-memory sorts of sets and dictionaries.
* AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
* It is used in applications that require improved searching apart from the database applications.

## B tree:

* B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.
* Searching an un-indexed and unsorted database containing n key values needs O(n) running time in the worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in the worst case.

## B+ tree:

* Searching of data in larger unsorted data sets takes a lot of time but this can be improved significantly with indexing using B tree.
* B trees are used to index the data especially in large databases as access to data stored in large databases on disks is very time-consuming.
* Application of B+tree is almost same as B tree

## Graph:

* In Computer science graphs are used to represent the flow of computation.
* Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
* In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graphs. It was the basic idea behind Google Page Ranking Algorithm.