

DS ASSIGNMENT 01.

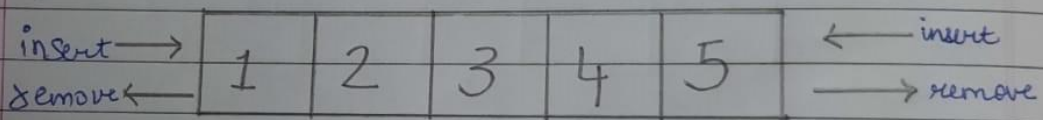
Assignment 1

Gaurav Amarnani D7A/67

Q.1 Explain Dequeue and Priority queue with proper examples.

→ Dequeue :

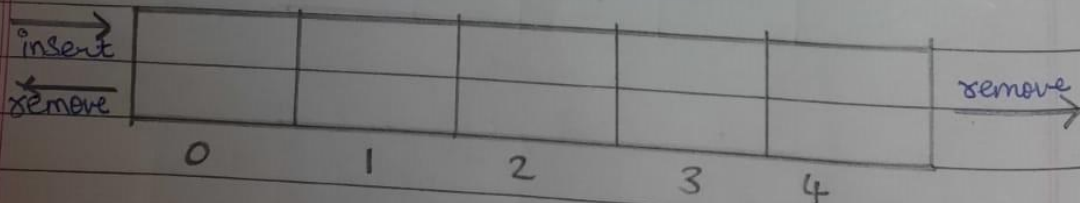
The dequeue stands for double ended queue. It is a linear data structure in which the insertion and deletion operation are performed from both ends. Thus, it does not follow FIFO rule (First In First Out). We can say that dequeue is generalized version of the queue.



There are two types of dequeue

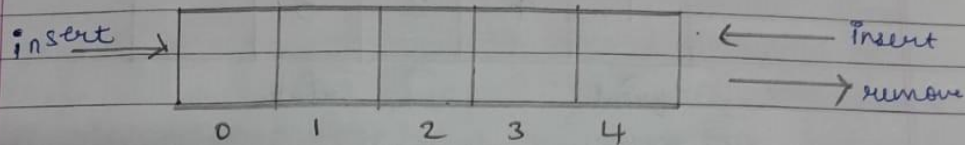
i Input-restricted queue :

The input-restricted queue means that some restrictions are applied to the insertion. In this queue, the input (insertion) is applied to one end while the deletion is applied from both ends.



ii Output restricted queue:

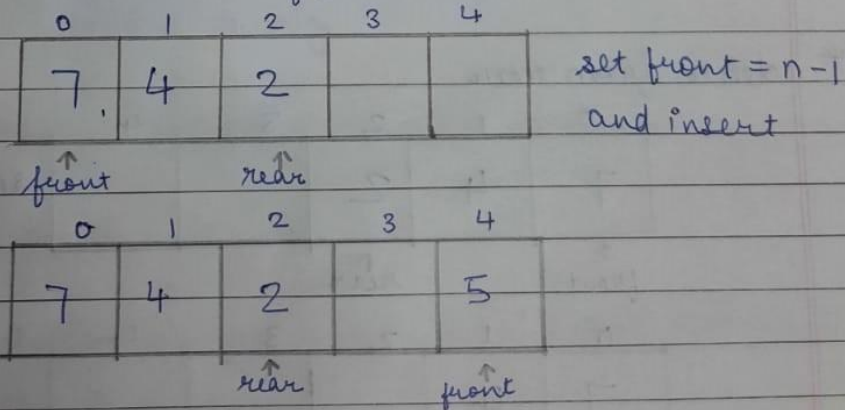
The output restricted queue means that same restriction are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



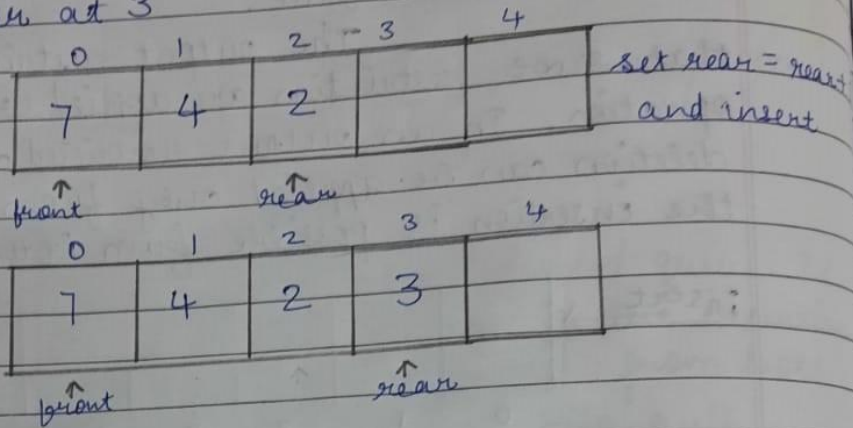
Operations on Deque

- Insert at Front
- Insert at rear
- Delete from Front
- Delete from rear

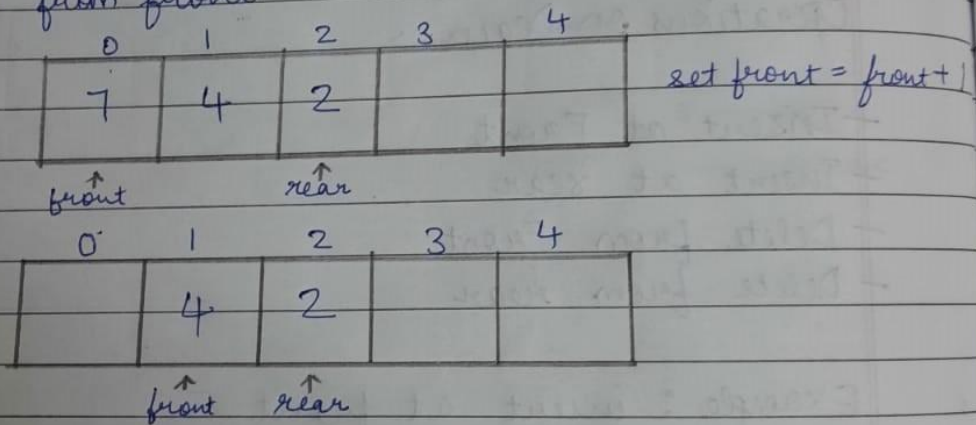
Example : insert at front 5



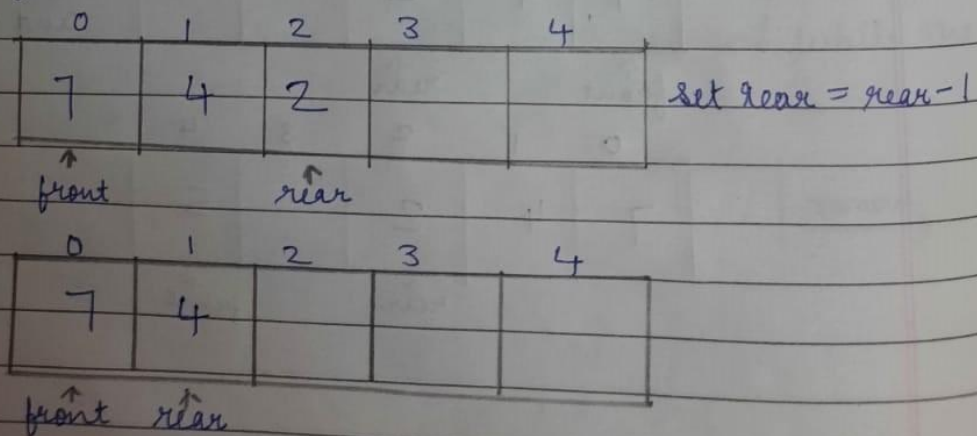
* insert rear at 3



* delete from front



* delete from rear



part

 $+1$

Diagram illustrating a priority queue (min-heap) structure:

10	6	4	2	1
----	---	---	---	---

Arrows indicate the flow of elements:

- Dequeue (left arrow pointing to 10)
- Enqueue (right arrow pointing to 1)

Annotation: Element with highest priority (pointing to 1)

Operation on priority queue

0	1	2	3	4
130	40	10	5	

↑ rear

front ↑

20 will be inserted after 40 i.e at index 2

0	1	2	3	4
130	40	20	10	5
\uparrow front				\uparrow rear

ii Delete

0	1	2	3	4
130	40	10	5	

set front = front + 1

front rear

0	1	2	3	4
	40	10	5	
	\uparrow front		\uparrow rear	

Q2 Convert the following infix expression to postfix using stack and then evaluate the same using stack.

a. $25 * 15 / 5^2 + 10 - 2 * 3$

Scanned	Stack	Postfix
C	C	
25	C	25
*	C*	25
15	C*	25 15
/	C/	25 15 *
5	C/	25 15 * 5
^	C/^	25 15 * 5
2	C/^	25 15 * 5 2
+	C+	25 15 * 5 2 ^ /
10	C+	25 15 * 5 2 ^ / 10 +
-	C-	25 15 * 5 2 ^ / 10 + 2
2	C-	25 15 * 5 2 ^ / 10 + 2
*	C-*	25 15 * 5 2 ^ / 10 + 2 3
3	C-*	25 15 * 5 2 ^ / 10 + 2 3 *
)		25 15 * 5 2 ^ / 10 + 2 3 *

Evaluate above postfix expression

Scanned	Stack	Operation
25	25	
15	25 15	
*	375	$25 * 15$
5	375 5	

2	375 5 2	5^2
^	375 25	$375/25$
/	15	
10	15 10	$15+10$
+	25	
2	25 2	
3	25 2 3	$2*3$
*	25 6	$25-6$
-	19	

\therefore Postfix expression : 25 15* 5 2^/10 + 2 3 -
 Evaluation : 19

b. $-10+9/3*2+13-4$

scanned	stack	Postfix
((
-	(-	
10	(-	10
+	(+	10-
9	(+	10-9
/	(+ /	10-9
3	(+ /	10-93
*	(+ *	10-93 /
2	(+ *	10-93 /
+	(+ *	10-93 / 2
13	(+ *	10-93 / 2 * +
-	(-	10-93 / 2 * + 13
4	(-	10-93 / 2 * + 13 +
)		10-93 / 2 * + 13 + 4

Evaluate above postfix expression

Scanned	stack	Operation
10	10	
-	-10	$0-10$
9	-10 9	
3	-10 9 3	
/	-10 3	$9/3$
2	-10 3 2	
*	-10 6	$3*2$
+	-4	$-10+6$
13	-4 13	
+	9	$-4+13$
4	9 4	
-	5	$9-4$

∴ Postfix expression: $10-9\ 3/2\ *+13+4-$

Evaluation:- 5

Q.3 Convert the following infix expression to postfix expression.

i $A - (B + C - (D + E^F) / G) * H$

Scanned	Stack	Postfix
		A
C	C	A
A	C	A
-	C -	A B
C	C - C	A B
B	C - C	ABC
+	C - C +	ABC +
C	C - C +	ABC +
-	C - (-	ABC + D
C	C - C - C	ABC + D
D	C - C - C	ABC + DE
+	C - C - C +	ABC + DE
E	C - C - C +	ABC + DEF
^	C - C - C +	ABC + DEF^
F	C - C - C + ^	ABC + DEF^ +
)	C - (-	ABC + DEF^ +
/	C - C - /	ABC + DEF^ + G /
G	C - C - /	ABC + DEF^ + G / -
)	C -	ABC + DEF^ + G / -
*	C - *	ABC + DEF^ + G / -
H	C - *	ABC + DEF^ + G / - H
)		ABC + DEF^ + G / - H * -

∴ Postfix expression: $ABC + DEF^ + G / - H * -$

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.

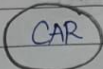
fix

Assignment 2

Gaurav Amarnani D7A/67

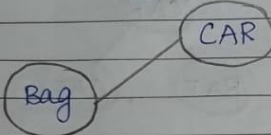
Q.1 Create AVL Tree for the following data and show all the steps with rotation: CAR, BAG, ANT, BAT, CAT, ART, APT (use alphabetic order)

→ step 1: insert CAR

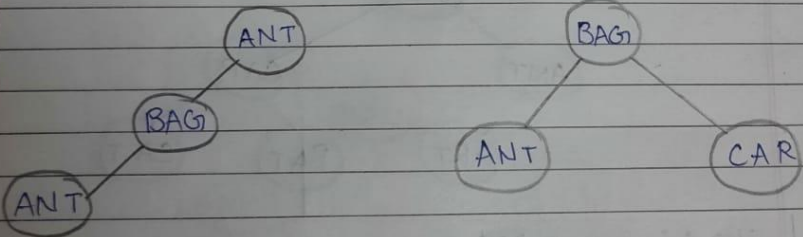


Balance factor: 0

Step 2: insert Bag



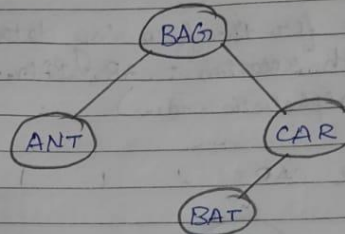
Step 3: insert ANT



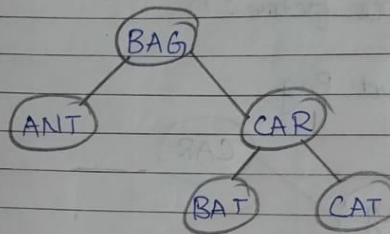
As balance factor of Ant is 2

∴ perform Right rotation

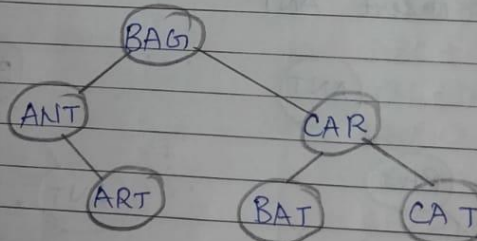
Step 4: insert Bat



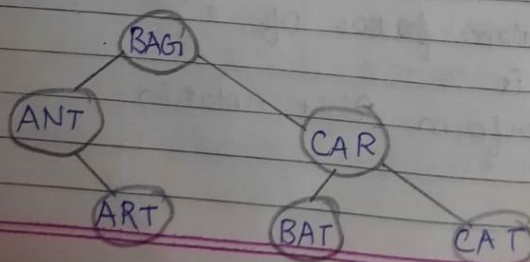
Step 5: insert cat



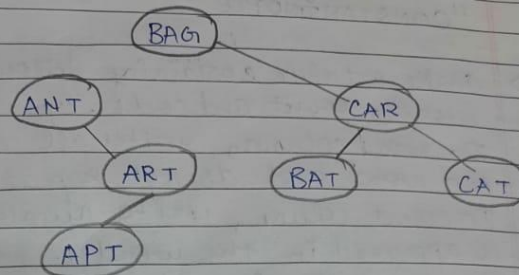
Step 6: insert Art



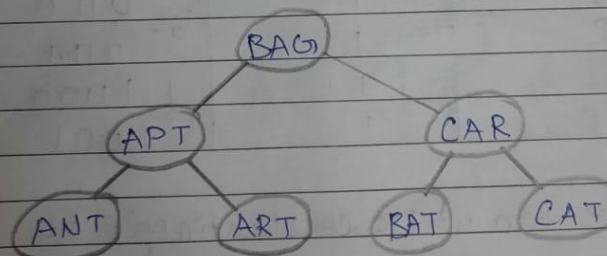
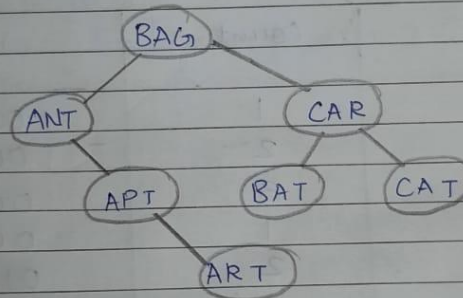
Step 6: insert Art



Step 7: Insert Apt



As the balance factor of ANT is -2
 \therefore perform Right Left Rotation



\therefore Total rotation required are
 Left : 1
 Right : 2

Q.2 Explain Huffman Encoding for the word:
"CONSTANTINOPLE"

- Make a table containing following columns -
word, count and code.
In word column, write all alphabet from the word and do not repeat single alphabet.
In count column, write number of times alphabet is appeared in the word.
In constantinople, there are 10 characters (without repeating). So, we will need 10 unique code in binary form.

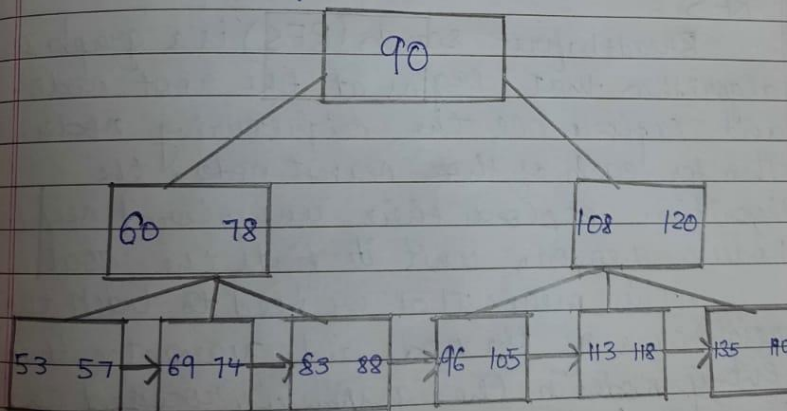
word	count	Code
C	1	0000
O	2	0001
N	3	0010
S	1	0011
T	2	0100
A	1	0101
I	1	0110
P	1	0111
L	1	1000
E	1	1001

∴ we can write constantinople as
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

Q3 Explain B+ Tree with example.

→ B+ Tree is an extension of BTree which allows efficient insertion, deletion and search operations. The leaf nodes of B+ Tree are linked together in the form of singly linked lists to make the search queries more efficient.

B+ Tree are used to store large amount of data which cannot be stored in main memory. The internal nodes of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory. Internal nodes of B+ Tree are often called index nodes. B+ Tree of order 3 is shown in the following figure



Advantages of B+ Tree

- Records can be fetched in equal number of disk access
- Height of Tree remain balanced and less as compared to B Tree
- We can access the data stored in B+ Tree sequentially as well as directly.
- Keys are used for indexing
- Faster search queries as the data is stored only on the leaf nodes.

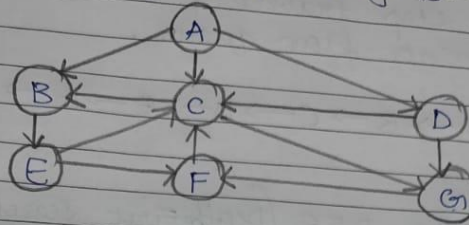
Q4 Explain BFS and DFS Traversal method with example.

→ BFS:

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes and so on, until it finds the goal.

This means that we need to track the neighbours of the nodes and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue and a variable to represent current state of the node.

Traverse following graph using BFS.



- Add A to queue and Null to origin

Queue : A

Origin : ∅

- Now add all the neighbours of A

Queue : A B C D E

Origin : ∅ A A A B

- Now add C as origin, add all neighbours of C

Queue : A B C D E G

Origin : ∅ A A A B C

- Moving to D, its neighbours is C and G, but both elements are already present in queue

- Now add E as origin and add all neighbours of E

Queue : A B C D E G F

Origin : ∅ A A A B C E

— As all the nodes of graph are visited, we stop traversing and dequeue all elements from queue.

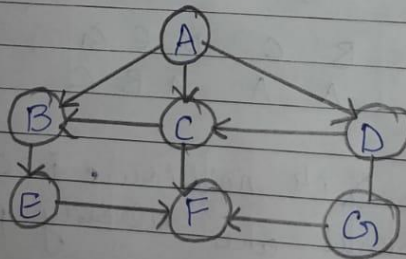
$\therefore A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow G \rightarrow F$

DFS:

The DFS (Depth First Search) algorithm progresses by expanding the starting node of graph and then going deeper and deeper until the goal node is found, or until a node that has not been completely explored.

That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. Then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

Traverse following graph using DFS



— Add A to the stack

stack : A

Printed elements Empty

- Remove A and add neighbour of D (C is already added)

stack : B C G
printed elements : A

- Remove D and add neighbour of D (C is already added)

stack : B C G
printed elements : A D

- Remove G and neighbour of G

stack : B C F
printed elements : A D G

- Now neighbouring elements of F and C are visited, so remove C, F and B and then add neighbours of B.

stack : E
printed elements : A D G F C B

- Now all neighbour of E is visited. so, remove E.

stack : Empty
printed elements : A D G F C B E

∴ $A \rightarrow D \rightarrow G \rightarrow F \rightarrow C \rightarrow B \rightarrow E$

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 $O(\log n)$ 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.