Name- Mohammed Raihan

Course - MCA 2C

Subject - DFS

DFS Theory Assignment 1

Solved Question Paper 2019

Questions	Fully Attempted	Partially Attempted	Not Attempted	Total Questions
Q-1 a	✓			
Q-1 b	✓			
Q-1 c	✓			
Q-1 d	✓			
Q-1 e	✓			
Q-1 f	✓			
Q-1 g			✓	
Q-2 a	✓			
Q-2 b	✓			
Q-3	✓			
Q-4 a	✓			
Q-4 b	✓			
Q-5	✓			

(Please write your Exam Roll No.) Exam Roll No. 10711604418

END TERM EXAMINATION

SECOND SEMESTER [MCA] MAY-JUNE-2019

Paper Code: MCA-102 **Subject: Data and File Structure**

Time : 3 Hours **Maximum Marks :75**

Note: Attempt five questions in all including Q. No.1 which is compulsory. Select one question from each unit.

Q1 (a) Write any five applications of stacks. (2.5)
(b) Define queue and what are the operations that can be performed on queues? (2.5)
(c) Differentiate between full binary tree & complete binary tree? (2.5)
(d) List the properties of B-tree? (2.5)
(e) Write short note on bubble sort algorithm. (2.5)
(f) What are the merits and demerits of binary search? (2.5)
(g) Explain critical path. (2.5)
(h) What is meant by activity network? (2.5)
(i) What are the various disadvantages of sequential file organization? (2.5)
(j) Explain polyphose merge. (2.5)

UNIT-I

Q2 (a) Write various steps to transform the following postfix expression to infix expression: ABCDE - + $ * EF * -. (8)
(b) Differentiate between multistack and multiqueue. (4.5)

Q3 Define doubly linked list. Write the procedure to implement following operations on a doubly linked list. (12.5)
(i) Insert an element at beginning.
(ii) Insert an element at a particular location.
(iii) Delete an element from beginning.
(iv) Delete an element form a particular location.

UNIT-II

Q4 (a) Construct a binary tree whose order of needs in is given as blow.
Inorder: D B H E A I F J C G
Preorder: A B D E H C F I J G (8)
(b) What is a AVL tree? Explain the advantages of AVL trees. (4.5)

Q5 Explain in detail the various operations that can be performed on priority queue. (12.5)

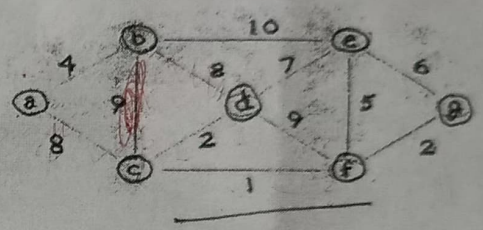
UNIT-III

Q6 (a) Write an algorithm to sort the elements using mergesort. (10)
(b) Write a short note on Hash Table. (2.5)

MCA-102 P1/2

P.T.O.

Q7 Find the minimum spanning tree of the following graph using Kruskal's Algorithm. (12.5)

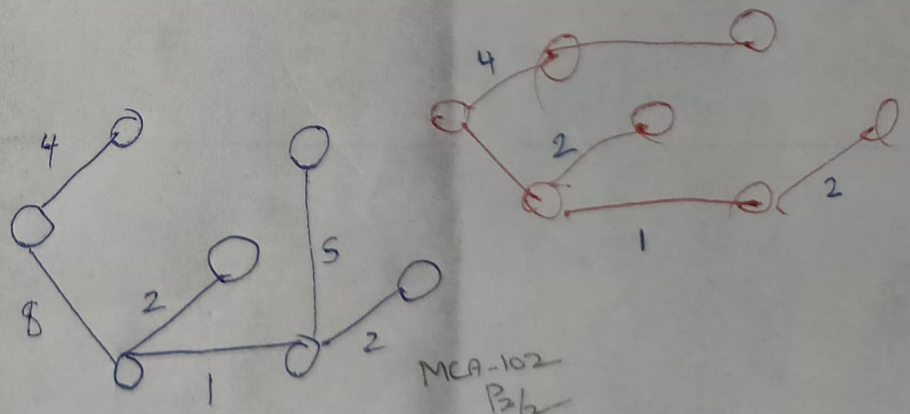


UNIT-IV

Q8 (a) Differentiate between double buffering and block buffering. (4)
(b) Write a C program to read name and marks of n number of students from user and store them in a file. (8.5)

Q9 Explain any six functions that are used for handling sequential files in C language. (12.5)

.....



MCA-102 P3/2

2-1a) Write any 5 applications of stack

Fonction call management is programming language

Expression evaluation & conversion - Infix to Postfix and prefix & vise-versa

Undo Fonctionality - Stacks are commonly used to implement undo functionality in applications. Each action performed by the user is pushed onto the stack, & when the user wants to undo an action, it is popped from the stack.

Syntax Parsing / Parenthesis matching - Stack plays a crucial role in syntax parsing especially in compilers & interpreters.

Backtracking - Stacks are used in algorithms that require backtracking, such as DFS in graph traversal.

Memory Management

Browser History - Stacks are used to implement browser history fonctionality. Each visited URL is pushed onto the stack, allowing user to navigate backword through their browsing history.

2-1b) Define Queue & what are the various operations that can be performed on queues

A queue is a fondamental data structure that represents a callection of elements where elements are inserted from one end (rear) & removed from the other end (front). It follows the First In First Out (FIFO) principle, meaning the element that is inserted first will be the one to be removed first.

Operations

1) Enqueue - Adding an element to the rear of the queue
2) Dequeue - Removing an element from the front of the queue
3) Peek / Front - Viewing the element at the front of the queue without removing it.
4) Empty - Checking if queue is empty (Underflow)
5) Full - Checking if queue is full (Overflow)
6) Size - Determining the number of elements in a queue

Q-1 c) Full Binary tree V/s Complete Binary tree

Characteristics	Full Binary Tree	Complete Binary Tree
Definition	A binary tree in which every node other than the leaves has exactly two children.	A binary tree in which every level except possibly the last, is completely filled, & all nodes are as far as ~~left~~ as possible
Node count	The no. of nodes can be calculated using the formula $2^{h+1} - 1$	The no. of nodes varies but can range from 2^h to $2^{h+1} - 1$.
Shape	Full Binary tree exhibit a balanced & Symmetric shape	Complete binary trees may not exibit perfect symmetry but are balanced in terms of level filling
Leaf Node	All leaf nodes are at same level	Leaf nodes are concentrated towards the bottom level but may not be at the same level.

Q-1d) List the properties of B tree

) B tree maintain data in sorted order. Each node contains key that are arranged in ascending order

) B tree are self balancing, ensuring that the height of the tree remains logrithmic with respect to the number of keys in the tree. This balance ensures efficient search, insertion & deletion.

) Each node can have a variable no. of children within a certain range. commonly denoted by a paramter t (the minimum degree of the tree). Each node in B tree can contain multiple keys & pointers to child nodes. The no of keys in a node determines the no. of child pointer it has. A B tree must have at least t-1 keys & at most 2t-1 keys.

) Root Property - The root of the B tree can have a minimum of 1 key if its not a leaf node. It may have 0 keys if its a leaf node.

) Internal nodes - Internal nodes (Non leaf nodes) except for the root have a minimum of t children.

Search operation - Searching is similar to BST but is more complex due to multiple keys per node.

Insertion Operation - Insertion involves finding the appropriate leaf node inserting the key while monitoring the order of keys within nodes & the balance property of the tree. If the insertion causes a node to exceed the maximum no. of keys, it is split.

Q-1c) Write short note on bubble sort algorithm

Bubble sort gets its name from the fact that lighter data (bubbles) flo at the top of the dataset. Bubble sort is alternatively called "sinki sort" for the opposite reason, which is that some elements of data sink to the bottom of the dataset.

Ex In an array of {1,2,3,4,5}

1 is lighter thus comes first or top

5 is heavier thus comes last or bottom

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average & worst case time is quit high.

Bubble sort algorithm

1) Traverse from left & compare the adjacent element & the higher one placed at right side.
2) In this way, the largest element is moved to the rightmost end first.
3) This process is continued to find the second largest & place it & so until the data is sorted

Time complexity - $O(n*n)$

Bubble is a stable algorithm

Bubble is not Adaptive by default, but we convert it to an adaptive algorithm

2-1 f) Merits & Demerits of binary search

Merits

1) Efficiency - Binary search is highly efficient, especially for large databases, as it has a time complexity of O(log n), where n is the number of elements in the array. This is efficiency makes it suitable for applications where performance is crucial.
2) Simplicity - The algorithm is relatively simple to implement & understand, requiring only a sorted array & a few comparisons to find the desired element.
3) Space Efficiency - Binary search operates directly on the given array or list without requiring additional data structures, making it space efficient compared to other search algorithms like hash tables.
4) Versatility - Binary search can be applied to various types of data, as long as data is sorted. It is not limited to specific data types or structure.

Demerits

1) Requirement of Sorted Data
2) Limited Applicability - Binary search is applicable only to static data structure like arrays. Dynamic data structures, such as linked lists, require additional steps for binary search to be efficient. Such as converting them to arrays etc.
3) Not suitable for Unsorted arrays or data.
4) Limited to single dimensional data.

Name - Mohammed Raihan

Course - MCA 2C

Subject - DFS

Solved Question Paper 2019

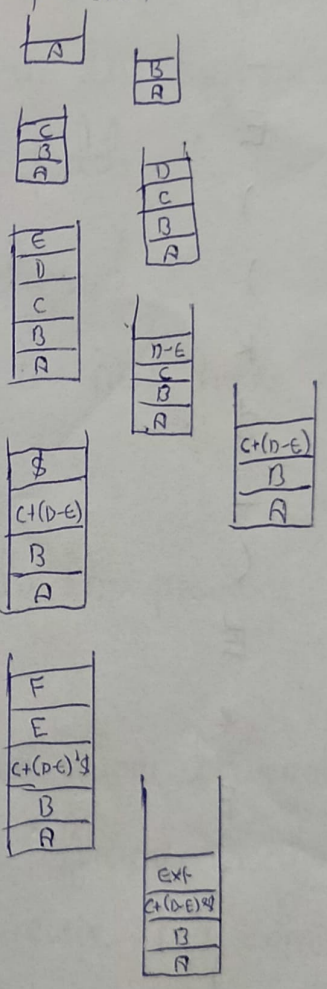
UNIT - 1

Q-2 a) Write various steps to transform the following postfix expression to infix expression ABCDE−+$*EF*−

Postfix to Infix : ABCDE−+$*EF*−

Reading of Postfix	stack top
A	A
B	B
C	C
D	D
E	E
−	(D-E)
+	C+(D-E)
$	C+(D-E)
*	C+(D-E)*$
E	E
F	F
*	(E*F)
−	(C+(D-E))*$ − (E*F)
	B(C+(D-E))*$ − (E*F)
	A(B(C+(D-E))*$ − (E*F))

Expression



Postfix Expression

ABCDE−+$*EF*−

Infix Expression

A(B(C+(D-E))*$−(E*F))

In order to check if the obtained infix expression is correct convert it to postfix and compare Infix A(B(C+(D-E))*$-(E*F))

Character	stack	Expression
A		A
(	(	A
B	(	AB
(	((	AB
C	((	ABC
+	((+	ABC
(	((+(	ABC
D	((+(	ABCD
-	((+(-	ABCD
E	((+(-	ABCDE
)	((+	ABCDE-
)	(	ABCDE-+
*	(*	ABCDE-+
$	(*	ABCDE-+$
-	(-	ABCDE-+$*
(	(-(	ABCDE-+$*
E	(-(	ABCDE-+$*E
*	(-(*	ABCDE-+$*E
F	(-(*	ABCDE-+$*EF
)	(-	ABCDE-+$*EF*
)		ABCDE-+$*EF*-

Q-2 b) Differentiate b/w multistack & multiqueue

Q-3) Define Doubly linked list. Write the procedure to implement the following

i) Insert an element at beginning
ii) Insert an element at a particular location
iii) Delete an element from beginning
iv) Delete an element from a particular location

A doubly linked list in a linear data structure similar to a singly linked list but with an additional pointer in each node, pointing to the previous node as well as the next node in the sequence. In doubly linked list, each node contains three fields

1) Data - This field stores the actual value or data element
2) Pointer to the next node - This pointer points to the next node in the sequence
3) Pointer to the previous node - This pointer points to the previous node in the sequence.

Because of these additional pointers, doubly linked list allow traversal in both forward & backward direction, enabling more efficient insertion & deletion operation at both ends of the list. However, they consume slightly more memory due to the extra pointer.

(i) Insert an element at beginning

1) Allocate memory for a new node
2) Set the data of the new node to the value you want to insert
3) Set the next pointer of the new node to point to the current hea of the list
4) Set the previous pointer of the current head (if exists) to point to the new node
5) Update the head pointer to point to the new node

(ii) Insert element at a particular position

1) Traverse the list to find the node at the desired location
2) Allocate memory for a new node
3) Set the data of the new node to the value you want to insert
4) Adjust the pointers of the adjacent nodes to include the new n
5) Update the pointers of the new node to point to the adjacent
6) Connect the new node into the list by updating the pointers of the adjacent nodes

- Traverse the list to find the node at position 'pos-1'
- Allocated memory for the new node
- Set the data of the new node
- Adjust the pointers of the new node previous node 'pos-1' & the current node ('pos') to include the new node
- Update the pointers of the new node to point to the previous nod and the next node
- If needed, update the head of the list, if the insertion is at the beginning

Delete an element from beginning

Check if the list is empty, If it is empty, there is nothing to delete

If the list is not empty, update the head pointer to point to the second node (if it exists)

Free the memory allocated for the node to be deleted

If the list has more than one node, update the previous pointer of the new noode to point to NULL.

Return the deleted node's data (optional)

```
Procedure deleteAtBeginning(head_ref):
    // check if linked list is empty
    If head_ref is NULL:
        Print "List is Empty. Nothing to delete"
        Return NULL

    // Store the reference to the node to be deleted
    deleted_node = head_ref

    // Update the head pointer to point to the next node
    head_ref = head_ref → next

    // If there is another node, update its previous pointer to NULL
    If head_ref is not NULL:
        head_ref → prev = NULL

    // Free the memory allocated for the deleted node
    Free deleted_node

    // optionally, return the data of the deleted node
    Return deleted_node → data
```

(IV) Delete an element from particular location

1) Traverse the linked list to find the node at the desired location
2) If the position is out of bounds or the list is empty, return an error or do nothing
3) Otherwise, adjust the pointers of the adjacent nodes to bypass the node to be deleted
4) Free the memory allocated for the node to be deleted.

Q-2 b) Differentiate b/w multistack v/s multiqueue

Multi Stack - Is a data structure that supports multiple stacks within a single data structure

- One common approach is to allocate a fixed amount of memory & divide it into multiple sections, each of which behaves like an indepen stack.
- Usage - Multistacks are usefull in scenarios where you need to manage multiple stacks simultaneously.

Multi Queue - Is a data structure that supports multiple queues within a single data structure.

- One common opprach is to allocate a fixed amount of memory & divide it into multiple sections each of which behaves like an independent queue.
- Multi Queues are bst beneficial when you have multiple streams of dat or tasks that need to be processed independently. For instance, in o you might have seperate queues for handling different types of processes or I/O requests.

UNIT-II

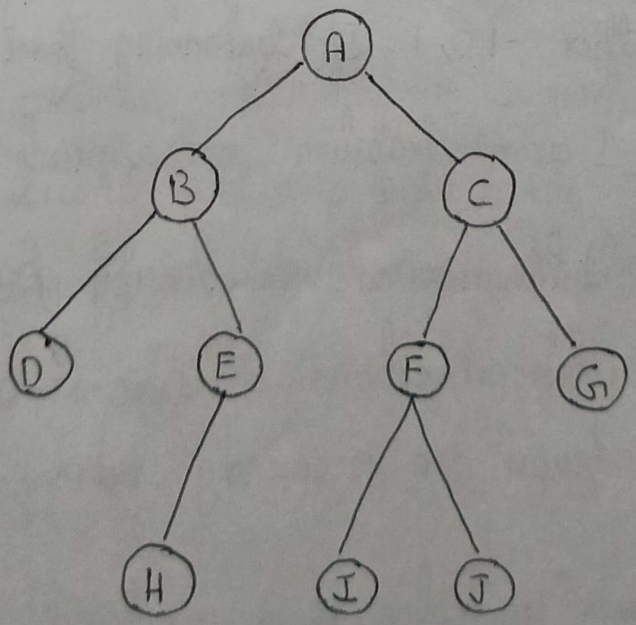
Q-4a) Construct a binary tree whose

Inorder : D B H E A I F J C G

Preorder : A B D E H C F I J G

You can use the preorder traversal to identify the root node & then partition the inorder traversal accordingly to construct the left & right subtree recursively.

1) The first element in the preorder traversal A is the root node
2) Find the position of A in the inorder traversal. A divides the inorder traversal into two parts "DBHE" & "IFJCG" (right subtree)
3) Recursively construct the left subtree using the elements "DBHE" & preorder traversal is "BDEH".
4) Recursively construct the right subtree using the elements "IFJCG" & the preorder traversal "CFIJG"



Q-4b) What is a AVL tree? Explain the advantages of AVL tree

An Avl tree, named after its inventors Addson-Velsky & Landis, is a self balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. This property helps in maintaining the balance of the tree, ensuring that the height of the tree remains logarithmic in the number of nodes.

Insertion or deletion of a node may cause the tree to become unbalanced. In such cases, one or more rotations are performed to restore the balance of the tree while maintaining the AVL property.

The rotations in AVL trees include single rotations (left & right rotations) and double rotations (left-right & right left rotations). These rotations help in maintainning the balance of the tree while preserving the order preserving the ordering property of binary search tree.

The balancing factor (difference blw left & right subtree) of each node in an AVL tree is either -1, 0, 1. If balancing factor of any node becomes greater than 1 or -1, rotations are applied to restore balan

- Most in memory sets & dictionaries are stored using AVL Tree
- Database applications, where insertions & deletions are less common but frequent data lookups are necessary. Also frequently employ AVL Trees.
- In addition to databases application, it is employed in other applications that call for better searching
- softwares that needs optimized search
- It is applied in corporate areas & storyline games

2-5) Explain in detail about various operations that can be performed in a priority Queue

A priority queue is an abstract data structure that stores a collection of elements with priorities. Priority queues are normally implemented using data structures such as heaps.

Operations

Insertion (enqueue) - This operation adds a new element with its associated priority to the priority queue. The new element is typically added at an appropriate position according to its priority.

Deletion (dequeue) - This operation removes & returns the element with the highest (or lowest) priority from the priority queue. After removal, the priority queue adjusts itself to maintain its properties.

Peek/Top - This operation retrieves the element with the highest or lowest priority from the priority queue without removing it. It allows you to observe the top element without altering the queue.

- Change Priority - Some implementations support changing the priority of an element already in the priority queue. This operation is used when the priority of an element needs to be updated dynamically.
- Size - This operations returns the number of elements currently stored in the priority queue.

1) Insertion (enque)

- When inserting an element into a priority queue, the element is typically placed in a position that maintains the priority queue's ordering property.
- In most implementation, this means inserting the new element at the end of the queue & then adjusting its position upwards (for max heap) or downwards (for min heap) until the heap property is satisfied.
- The time complexity for insertion in a binary heap is $O(\log n)$, where n is the number of elements in the heap.