**DS- Final\_Exam (Theory) Answer Script**

| Question No. 01` |
| --- |
| Traverse the following binary tree using the inorder, preorder, postorder, and level order techniques. Level each of the nodes of the tree. Also, find the height of the tree. (8)https://lh4.googleusercontent.com/qoQf9lF7e-cZtb4i52FViM3l9c-AI67Q8VaH3ZItIQPMFbR60PXqoSs8EMfed8qh4cebe3iiEWRn66Ce2IMyXBcNq4hc6IAq2C7PKvupXhj1I02n5Tzj5PMq-7UWYiGhbchBLW5Nf2gl-B9vjVAcP6ZY7SMbrnwiaWHeTni9kGcOI5fIWyh1jphR8g |
| Answer:    **Height:** height is the number of edges in the longest path from the root of a tree to a leaf node.  In this case, height is 3 |

| Question No. 02 |
| --- |
| Draw a binary tree using the given preorder and in order sequences:  Pre order: ABDEFCG;  In order: DBFEACG (5) |
| Answer: |

| Question No. 03 |
| --- |
| Draw a binary tree using the given in order and post order sequences (5)  In order: DBEFAGC  Post order: DFEBGCA |
| Answer: |

| Question No. 04 |
| --- |
| Draw a max-heap and min-heap trees from the following data where X=last digits of your birth month + 1.  10 40 20 8 99 X 15 17 (5) |
| Answer:  Let birth Month is 05, so X=5+1=6 Max and Min heap is shown below,   |  | | --- | |  | |  | |

| Question No. 05 |
| --- |
| Use HeapSort to Sort the Data in Descending Order. Show the status of the array and heap in each iteration. Data: 20 50 40 5 30 15 (7) |
| Answer:    After sorting(Descending Order) Final answer is: 50 40 30 20 15 5 |

| Question No. 06 |
| --- |
| Draw a binary search tree for the following data 10 40 20 8 99 16 15 17 11 14 1.  Can We insert duplicate values in BST? State your opinion with a logical explanation. (7) |
| Answer:    **Duplicate in BST:**  Characteristics of a BST are as follows,  1. Left node must be smaller than the parent node,  2. Right node must be greater than the parent node  3. No duplication is allowed.  Duplicate values violate all these conditions, so duplicate values are not allowed in a BST.  With these conditions in place, duplicate values will be omitted in the output. |

| Question No. 07 |
| --- |
| Perform the Following Operations on the BST given in the Figure below. (2+2+2)   * Delete 12 * Insert 11 * Delete 6 |
| Answer: |
| Question No. 08 |
| Given Infix Notation: (5\*((6^2)+(7-(2/6))))-((7\*(8+1))+(5\*4)) (4+2+4+2)   1. Convert it into Prefix Notation using Stack and Show the status of Stack and Console in all the steps 2. Evaluate the Prefix Notation derived in (A) using Stack and Find the result of the statement. Show the status of the Stack in each step. 3. Convert it into Postfix Notation using Stack and Show the status of Stack and Console in all the steps 4. Evaluate the Postfix Notation derived in (C) using Stack and Find the result of the statement. Show the status of the Stack in each step. |
| Answer: **A. Infix to prefix:**  .    **B. prefix Evaluation:**    C and D: Not answered |

| Question No. 09 |
| --- |
| Write down all the steps of **Counting Sort** on the Following Array.   | Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Value | 3 | 3 | 1 | 7 | 7 | 4 | 4 | 5 | |
| Answer:  Frequency array:  Max value=7 , Freq[8]   | Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Value | 0 | 1 | 0 | 2 | 2 | 1 | 0 | 2 | 0 |   Cumulative array C[8]:   | Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Value | 0 | 1 | 1 | 3 | 5 | 6 | 6 | 8 | 8 |   Final array[8]:   | Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Value | 1 | 3 | 3 | 4 | 4 | 5 | 7 | 7 |   Steps:   | Steps | i | a[i] | k | k=k-1 | Final[k]=a[i] | | --- | --- | --- | --- | --- | --- | | 1 | 7 | a[7]=5 | C[5]=6 | 5 | Final[5]=a[7]=5 | | 2 | 6 | a[6]=4 | C[4]=5 | 4 | Final[4]=a[6]=4 | | 3 | 5 | a[5]=4 | C[4]=4 | 3 | Final[3]=a[5]=4 | | 4 | 4 | a[4]=7 | C[7]=8 | 7 | Final[7]=a[4]=7 | | 5 | 3 | a[3]=7 | C[7]=7 | 6 | Final[6]=a[3]=7 | | 6 | 2 | a[2]=1 | C[1]=1 | 0 | Final[0]=a[2]=1 | | 7 | 1 | a[1]=3 | C[3]=3 | 2 | Final[2]=a[1]=3 | | 8 | 0 | a[0]=3 | C[3]=2 | 1 | Final[1]=a[0]=3 | |

| Question No. 10 |
| --- |
| Comparing the time and space complexity, give your opinion on the following statement ***HeapSort is more efficient than Counting Sort*** (2) |
| Answer:  Heap sort time complexity is O(nlogn), and space complexity is O(1). For counting sort space complexity is O(n). Although the space complexity of counting sort is less than heap sort, it is not practical for large values.  If maximum value in an array is very large than counting sort can create a problem because it requires (n+1) space as memory to count the frequency of the elements. |

| Question No. 11 |
| --- |
| Find the location of A[15] [20] for the following data int A[50][100]. Assume, loc(A[0][0])= (AE92F6)H and Assume column-wise memory allocation (An Integer is a word addressable (4 bytes) datatype) (5) |
| Answer:  For Column-major ordering:  A[i][j]=Base +w[m(j-base)+(i-base)]……eq. 1  Here,  i=15, j= 20, Base=(AE92F6)H, w=word size=4, m=elements per column=50, base=0  so,  A[i][j]=Base +w[m(j-base)+(i-base)]  => A[15][20]=(AE92F6)H+(4[50(20-0)+(15-0)])D  =(AE92F6)H+(4060)D  =(AE92F6)H+(FDC)H (converting to Hexadecimal, using calculator)  =(AEA2D2)H |

| Question No. 12 |
| --- |
| Answer the following questions for the doubly linked list as shown below, where p = 12 , q = p+4, r = p+q, s = r-3, t = r+s. (5)  a)      head −> next −>next-> value = ?  b)      last −> prev −> next->value = ?  c)      temp −> prev −> prev −> prev= ?  d)      temp −> next−> prev −>prev->value = ?  e)      last −> prev −> prev −>next-> value = ?  ` |
| Answer:  Here, p=12, q=16, r=28, s=25, t=53   1. 28 2. 53 3. NULL 4. 16 5. 25 |

| Question No. 13 |
| --- |
| Write an algorithm to display the data stored in a doubly linked list in reverse order. Assume only the head pointer is given for the linked list.  What are the merits and demerits of a doubly linked list over a linear linked list? (4) |
| Answer:  **Algorithm to reverse a Doubly Linked List:**   1. At first, consider the head pointer as temp pointer 2. Initially check, if the temp or head pointer is NULL or not. If NULL, print   ’-1’ and return.   1. If temp or head is not NULL initially, then traverses from the head to the last node following the loop condition temp->next is not equal to NULL   (temp->next !=NULL), and temp=temp->next;   1. Then traverse from the last node to the head node following the loop condition as temp is not equal to NULL and using temp=temp->prev.   **Advantages of Doubly Linked List over singly Linked List:**   1. Unlike singly LL, doubly LL has an extra pointer named “previous”, which points to the previous element. This makes it easier to reverse the double linked list compared to singly LL. 2. Deletion is easier because we can go back and forth very easily with the help of the previous pointer.   **Disadvantages Of Doubly Linked List Over Singly Linked List:**  It requires extra memory space because it has an extra pointer. |

| Question No. 14 |
| --- |
| Show the status of a QUEUE and PRIORITY QUEUE for the following operations, where the QUEUE is implemented by an array of sizes, m=3. Here, Enqueue and Dequeue mean insert and delete respectively, and  x=9, y=x+3, z=x+y, and p=y+z.  Enqueue(z), Enqueue(p), Dequeue(), Enqueue(y), Enqueue(z) |
| Answer:  Here, x=9, y=9+3=12, z=y+x=9+12=21, p=y+z=12+21=33  **Steps for QUEUE:**  (we know the queue follows FIFO and Here F=Front pointer, R=Rear pointer)   | Enqueue(z)=Enqueue(21) | F=0, R=0 | array[0]=21 | | --- | --- | --- | | Enqueue(p)=Enqueue(33) | F=0, R=1 | array[0]=21  array[1]=33 | | Dequeue() | F=1, R=1 | array[1]=33 | | Enqueue(y)=Enqueue(12) | F=1, R=2 | array[1]=33  array[2]=12 | | Enqueue(z)=Enqueue(21) | F=1, R=0 | array[0]=21  array[1]=33  array[2]=12 |   **Steps for Priority QUEUE( Min- Priority Queue):**  (we know the queue follows FIFO and Here F=Front pointer, R=Rear pointer)   | Enqueue(z)=Enqueue(21) | F=0, R=0 | array[0]=21 | | --- | --- | --- | | Enqueue(p)=Enqueue(33) | F=0, R=1 | array[0]=21  array[1]=33 | | Dequeue() | F=1, R=1 | array[1]=33 | | Enqueue(y)=Enqueue(12) | F=0, R=1 | array[0]=12  array[1]=33 | | Enqueue(z)=Enqueue(21) | F=0, R=2 | array[0]=12  array[1]=21  array[2]=33 | |

| Question No. 15 |
| --- |
| What are the merits of implementing a QUEUE using Array in a circular fashion?  How do you check the underflow and overflow in the QUEUE implemented circularly? (2+2) |
| Answer:  **Advantages of circular queue implementation using an array:**  As an array has a limited memory, when the rear pointer reaches the last index, no more elements can be inserted. Although because of dequeue operations there may be vacant locations before the front pointer but they cannot be used if the queue is not circular.  In the case of a circular queue, elements can be inserted in vacant memory locations. this helps in efficient memory uses.  **check the underflow and overflow:**  Let’s consider, F=Front pointer, R=Rear pointer, array size=n  Steps to detect overflow.   1. At first F=R=-1. 2. If (F=R=-1), after the first enqueue, F and R are incremented to 0 and the value is inserted into the array index 0.**(F=0, R=0)** 3. For the next enqueue operations, a new variable s is considered, where s= ((r+1) mod n) and then a new condition s! =F is checked. 4. This follows until s=n-1 and the value is inserted up to index n-1. 5. Next when r is n-1, s becomes equal to 0. 6. If there is no dequeue operation happens until now then still **F=0**. 7. The conditions**! =F is** False, which means s==F, so the value is going to be inserted at index 0, the same index where F is now located, this situation is called overflow.   In summary, it’s always checked whether s becomes equal to F or not, to solve the overflow problem.  Steps to detect underflow:   1. At first F=R=-1. 2. Let’s consider the queue is full (unto index n-1) and F=0 and R=n-1. 3. For every dequeue operation, two conditions array [F] ==0 and F==R is checked. 4. For every dequeue operation, F increments by 1, until F=n-1. 5. In this case F becomes equal to R (F==R), then array [F] is set to zero   (array [F] = 0).   1. Now, the queue is totally empty. 2. For the next dequeue operation, (array [F] ==0 && F==R) is true and underflow occurs.   In summary, when F and R become equal and array[F] is zero simultaneously, underflow detects. |

| Question No. 16 |
| --- |
| Given the root of a binary tree, Write down a function to invert the tree and return its root.Node\* invert\_tree(Node\* root)  (6) |
| Answer:  Code snippet:  // Invert binary Tree:  Node \*invert\_tree(Node \*root)  {      if (root == NULL)      {          return root;      }      // swap left-right      Node \*temp = root->left;      root->left = root->right;      root->right = temp;      // recursive call      invert\_tree(root->left);      invert\_tree(root->right);      return root;  } |

| Question No. 17 |
| --- |
| Write down the Pseudocode of the following traversals in the Binary Tree. (4+4)   1. Boundary Traversal: 15 11 8 6 9 14 20 35 30 26 2. ZigZag Level Wise Traversal: 15 26 11 8 12 20 30 35 14 9 6   https://lh4.googleusercontent.com/qoQf9lF7e-cZtb4i52FViM3l9c-AI67Q8VaH3ZItIQPMFbR60PXqoSs8EMfed8qh4cebe3iiEWRn66Ce2IMyXBcNq4hc6IAq2C7PKvupXhj1I02n5Tzj5PMq-7UWYiGhbchBLW5Nf2gl-B9vjVAcP6ZY7SMbrnwiaWHeTni9kGcOI5fIWyh1jphR8g |
| Answer:   1. **Pseudocode for Boundary Traversal:**   For left non leaves(11, 8):   * 1. Check if root==NULL, if NULL, return   2. If Root!=NULL,   3. Check if the root has left child or not, * If it has left child, print the left child data * Recursive call to root left child.   1. Check if it has right child or not * If it has right child then print the right child data * Recursive call to root right child   For right non leaves(26, 30):   * 1. Check if root==NULL, if NULL, return   2. If Root!=NULL,   3. Check if the root has right child or not, * If it has right child then print the right child data * Recursive call to root right child   1. Check if it has left child or not * If it has left child then print the left child data * Recursive call to root left child.   For leaves(left and right) (6, 9, 14, 20, 35)   * 1. Check if root==NULL, if NULL, return   2. If root left child and right child both are NULL then that’s a leaf node and print the data.  1. Recursively go the left child or right child until root is NULL. |
| 1. **Zig-Zag Level Wise Traversal:**   Following BFS method,   1. Initialize a queue named q. 2. Initialize an integer type variable count=0 to count levels of the tree. 3. Push root and NULL in q. 4. While q is not empty 5. Save the front data from q in a new node named **present node**. 6. If the **present node** is not NULL 7. Pop and print the present node. 8. If(**count is odd-push left child first**)   - if it has left child, push in q  - if it has right child, push in q   1. If(**count is even-push right child first**)   - if it has right child, push in q  - if it has left child, push in q   1. if the **present node** is NULL 2. if q is not empty(it has other elements-children from previous nodes, this means tree level is changing) 3. push NULL in q 4. Increment count by 1. |
| **Code snippet for boundary traversal:**  void printLeftNonLeaves(treeNode \*root)  {  if (root == NULL)  return;  if (root->leftChild != NULL)  {  cout << root->data << " ";  printLeftNonLeaves(root->leftChild);  }  else if (root->rightChild != NULL)  {  cout << root->data << " ";  printLeftNonLeaves(root->rightChild);  }  }  void printLeaves(treeNode \*root)  {  if (root == NULL)  return;  if (root->leftChild == NULL && root->rightChild == NULL)  {  cout << root->data << " ";  }  printLeaves(root->leftChild);  printLeaves(root->rightChild);  }  void printRightNonLeaves(treeNode \*root)  {  if (root == NULL)  return;  if (root->rightChild != NULL)  {  cout << root->data << " ";  printRightNonLeaves(root->rightChild); // check  }  else if (root->leftChild != NULL)  {  cout << root->data << " ";  printRightNonLeaves(root->leftChild); // check  }  } |
| **Code snippet for zig-zag traversal:**  void zigzag\_order(treeNode \*root)  {  int count = 0;  if (root == NULL)  {  return;  }  queue<treeNode \*> q;  q.push(root);  q.push(NULL);  //queue<int>z;  while (!q.empty())  {  treeNode \*presentNode = q.front();  q.pop();  if (presentNode != NULL)  {  cout << presentNode->val << " ";  //z.push(presentNode->val);  // count odd  if (count % 2 != 0)  {  if (presentNode->left != NULL) //left first  {  q.push(presentNode->left);  }  if (presentNode->right != NULL)  {  q.push(presentNode->right);  }  }  // count even  if(count % 2 == 0)  {  if (presentNode->right != NULL)//right first  {  q.push(presentNode->right);  }  if (presentNode->left != NULL)  {  q.push(presentNode->left);  }    }  }  else  {  if (!q.empty())  {  q.push(NULL);  count++;  }  }  } |

Submitted by: Subrata Saha

Email:[Subratabaec@gmail.com](mailto:Subratabaec@gmail.com)

Date: 18.09.2022

THE END