|  |  |
| --- | --- |
| **Practical No: 01** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Implement following searching techniques and find the time  complexity: i. linear ii. Binary |
| **Theory:** | Linear Search:  To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.  Let the elements of array are -  Linear Search Algorithm  Let the element to be searched is K = 41  Now, start from the first element and compare K with each element of the array.  Linear Search Algorithm  The value of K, i.e., 41, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.  Linear Search Algorithm  Now, the element to be searched is found. So algorithm will return the index of the element matched.To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.  Let the elements of array are -  Linear Search Algorithm  Let the element to be searched is K = 41  Now, start from the first element and compare K with each element of the array.  Linear Search Algorithm  The value of K, i.e., 41, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.  Linear Search Algorithm  Now, the element to be searched is found. So algorithm will return the index of the element matched.  **Time Complexity**    **Binary Search:**  Now, let's see the working of the Binary Search Algorithm.  To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.  There are two methods to implement the binary search algorithm -   * Iterative method * Recursive method   The recursive method of binary search follows the divide and conquer approach.  Let the elements of array are –  Binary Search Algorithm  Let the element to search is, **K = 56**  We have to use the below formula to calculate the **mid** of the array -   1. mid = (beg + end)/2   So, in the given array -  **beg** = 0  **end** = 8  **mid** = (0 + 8)/2 = 4. So, 4 is the mid of the array.  Binary Search Algorithm Binary Search Algorithm Binary Search Algorithm  Now, the element to search is found. So algorithm will return the index of the element matched.  **Time Complexity**    **Algorithm- Binary Search:** |
| **Source Code/Algorithm/Flow Chart:** | **Linear Search:**  #include<stdio.h>  #include<stdbool.h>  int main()  {  int arr[5]={2,10,5,6,37};  int key,i,count=0;  //bool x=false;  printf("Enter the key element: ");  scanf("%d",&key);  for (i=0;i<5;i++)  {  if(arr[i]==key)  {  //x=true;  count++;  }    }  if(count>0)  {  printf("Element found.");  }  else  {  printf("Element not found.");  }  return 0;  }  **Binary Search:**  **#include<stdio.h>**  **int BSI(int arr[], int l, int h, int key)**  **{**  **while(l<=h)**  **{**  **int mid=(l+h)/2;**  **if (key==arr[mid])**  **{**  **return mid;**  **}**  **if(key<arr[mid])**  **{**  **h=mid-1;**  **}**  **else**  **{**  **l=mid+1;**  **}**  **}**  **return -1;**  **}**  **int main()**  **{**  **int arr[8]={20,30,40,50,60,70,80,90};**  **int key;**  **printf("Enetr the key element: ");**  **scanf("%d",&key);**  **int element=BSI(arr,0,7,key);**  **if(element == -1)**  **{**  **printf("Element not found.");**  **}**  **else**  **{**  **printf("Element found at : %d", element);**  **}**  **return 0;**  **}** |
| **Output Screenshots (if applicable)** | 1)    2)    3) |
| **Conclusion** | Thus we have studied linear and binary searching techniques with their time complexities. |
| **Post Lab Questions:** | 1. Compare linear search and binary search 2. Why complexity of Binary search is O(log n)? |

|  |  |
| --- | --- |
| **Practical No: 02** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Implement following sorting techniques and find the time  complexity: i. Bubble ii. Selection iii. Insertion |
| **Theory:** | **Bubble Sort:**  To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n2).**  Let the elements of array are -  Bubble sort Algorithm **First Pass** Sorting will start from the initial two elements. Let compare them to check which is greater.  Bubble sort Algorithm  Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.  Bubble sort Algorithm  Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -  Bubble sort Algorithm  Now, compare 32 and 35.  Bubble sort Algorithm  Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.  Now, the comparison will be in between 35 and 10.  Bubble sort Algorithm  Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -  Bubble sort Algorithm  Now, move to the second iteration. **Second Pass** The same process will be followed for second iteration.  Bubble sort Algorithm  Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -  Bubble sort Algorithm  Now, move to the third iteration. **Third Pass** The same process will be followed for third iteration.  Bubble sort Algorithm  Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -  Bubble sort Algorithm  Now, move to the fourth iteration. **Fourth pass** Similarly, after the fourth iteration, the array will be -  Bubble sort Algorithm  Hence, there is no swapping required, so the array is completely sorted.  **Time Complexity:**    **Selection Sort:**  To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.  Let the elements of array are -  selection Sort Algorithm  Now, for the first position in the sorted array, the entire array is to be scanned sequentially.  At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.  selection Sort Algorithm  So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.  selection Sort Algorithm  For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.  selection Sort Algorithm  Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.  selection Sort Algorithm  The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.  selection Sort Algorithm  Now, the array is completely sorted.  **Time Complexity:**   |  |  | | --- | --- | | **Case** | **Time Complexity** | | **Best Case** | O(n2) | | **Average Case** | O(n2) | | **Worst Case** | O(n2) |   **Insertion Sort:**  To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.  Let the elements of array are -  Insertion Sort Algorithm  Initially, the first two elements are compared in insertion sort.  Insertion Sort Algorithm  Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.  Insertion Sort Algorithm  Now, move to the next two elements and compare them.  Insertion Sort Algorithm  Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.  For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.  Insertion Sort Algorithm  Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.  Insertion Sort Algorithm  Both 31 and 8 are not sorted. So, swap them.  Insertion Sort Algorithm  After swapping, elements 25 and 8 are unsorted.  Insertion Sort Algorithm  So, swap them.  Insertion Sort Algorithm  Now, elements 12 and 8 are unsorted.  Insertion Sort Algorithm  So, swap them too.  Insertion Sort Algorithm  Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.  Insertion Sort Algorithm  Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.  Insertion Sort Algorithm  Move to the next elements that are 32 and 17.  Insertion Sort Algorithm  17 is smaller than 32. So, swap them.  Insertion Sort Algorithm  Swapping makes 31 and 17 unsorted. So, swap them too.  Insertion Sort Algorithm  Now, swapping makes 25 and 17 unsorted. So, perform swapping again.  Insertion Sort Algorithm  Now, the array is completely sorted.  **Time Complexity:**   |  |  | | --- | --- | | **Case** | **Time Complexity** | | **Best Case** | O(n) | | **Average Case** | O(n2) | | **Worst Case** | O(n2) |   **Bubble sort Algorithm:**    **Selection sort Algorithm:**    **Insertion Sort Algorithm:** |
| **Source Code/Algorithm/Flow Chart:** | 1. **Bubble Sort:**   // Bubble Sort  #include<stdio.h>  #include<stdbool.h>  void swap(int \*a, int \*b)  {  int temp;  temp=\*a;  \*a=\*b;  \*b=temp;  }  int BS(int arr[],int n)  {  int i,j;  bool flag;  for (i=0; i<n-1; i++)  {  flag=false;  for(j=0; j<n-i-1; j++)  {  if(arr[j]>arr[j+1])  {  swap(&arr[j], &arr[j+1]);  flag=true;  }  }  if(flag==false)  {  printf("The Array is already Sorted\n");  break;  }  }  return 0;  }  int main()  {  int arr[5];  int n=5;  int i,j;  printf("Enter array element: ");  for(i=0; i<n; i++)  {  scanf("%d",&arr[i]);  }  BS(arr,n);  printf("Sorted Array is: ");  for(i=0;i<n;i++)  {  printf("\n %d",arr[i]);  }  return 0;  }   1. **Insertion Sort:**   #include<stdio.h>  void InsertionSort(int arr[], int size)  {  int i,j;  for (i=1; i<size;i++)  {  int key=arr[i];  int j=i-1;  while(j>=0 && arr[j]>key)  {  arr[j+1]=arr[j];  j--;  }  arr[j+1]=key;  }  }  int main()  {  int arr[]={64,34,55,90,100,55};  int i;  int size=sizeof(arr)/sizeof(arr[0]);  InsertionSort(arr, size);    printf("Sorted Array: ");  for (i=0;i<size-1;i++)  {  printf("%d",arr[i]);  printf("\n");  }  return 0;  }   1. **Selection Sort:**   #include<stdio.h>  void SS(int arr[],int n)  {  int i,j;  for(i=0; i<n-1; i++)  {  int min\_index=1;  for(j=i+1; j<n; j++)  {  if(arr[j]<arr[min\_index])  {  min\_index=j;  }  }  int temp=arr[i];  arr[i]=arr[min\_index];  arr[min\_index]=temp;  }  }  int main()  {  int i;  int arr[]={64,25,12,22,11};  int n=sizeof(arr)/sizeof(arr[0]);  printf("Original Array: \n");  for(i=0; i<n; i++)  {  printf("%d\n",arr[i]);  }  SS(arr,n);  printf("\nSorted Array: ");  for(i=0; i<n; i++)  {  printf("\n%d",arr[i]);  }  return 0;  } |
| **Output Screenshots (if applicable)** | 1)    2)    3) |
| **Conclusion** | Thus we have studied bubble, selection and insertion sorting techniques with their time complexities. |
| **Post Lab Questions:** | 1. What is the worst-case complexity of selection sort? Why? 2. What is the advantage of selection sort over other sorting techniques? 3. What is the best-case time complexity of bubble sort? Explain in detail. |

|  |  |
| --- | --- |
| **Practical No: 03** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Implement following sorting techniques find the time complexity: Merge sort |
| **Theory:** | To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.  Let the elements of array are -  Merge sort  According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.  As there are eight elements in the given array, so it is divided into two arrays of size 4.  Merge sort  Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.  Merge sort  Now, again divide these arrays to get the atomic value that cannot be further divided.  Merge sort  Now, combine them in the same manner they were broken.  In combining, first compare the element of each array and then combine them into another array in sorted order.  So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.  Merge sort  In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.  Merge sort  Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -  Merge sort  Now, the array is completely sorted.  **Algorithm**:   1. MERGE\_SORT(arr, beg, end) 3. **if** beg < end 4. set mid = (beg + end)/2 5. MERGE\_SORT(arr, beg, mid) 6. MERGE\_SORT(arr, mid + 1, end) 7. MERGE (arr, beg, mid, end) 8. end of **if** 9. END MERGE\_SORT |
| **Source Code/Algorithm/Flow Chart:** | **Merge Sort:**  **#include <stdio.h>**  **void merge(int arr[], int left, int mid, int right) {**  **int n1 = mid - left + 1;**  **int n2 = right - mid;**  **int L[n1], R[n2];**  **for (int i = 0; i < n1; i++)**  **L[i] = arr[left + i];**  **for (int j = 0; j < n2; j++)**  **R[j] = arr[mid + 1 + j];**  **int i = 0, j = 0, k = left;**  **while (i < n1 && j < n2) {**  **if (L[i] <= R[j]) {**  **arr[k] = L[i];**  **i++;**  **} else {**  **arr[k] = R[j];**  **j++;**  **}**  **k++;**  **}**  **while (i < n1) {**  **arr[k] = L[i];**  **i++;**  **k++;**  **}**  **while (j < n2) {**  **arr[k] = R[j];**  **j++;**  **k++;**  **}**  **}**  **void mergeSort(int arr[], int left, int right) {**  **if (left < right) {**  **int mid = left + (right - left) / 2;**  **mergeSort(arr, left, mid);**  **mergeSort(arr, mid + 1, right);**  **merge(arr, left, mid, right);**  **}**  **}**  **int main() {**  **int n;**  **printf("Enter the number of elements: ");**  **scanf("%d", &n);**  **int arr[n];**  **printf("Enter the elements:\n");**  **for (int i = 0; i < n; ++i) {**  **scanf("%d", &arr[i]);**  **}**  **mergeSort(arr, 0, n - 1);**  **printf("Sorted array:\n");**  **for (int i = 0; i < n; ++i) {**  **printf("%d ", arr[i]);**  **}**  **return 0;**  **}** |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied merge sorting techniques with its time complexity. |
| **Post Lab Questions:** | 1. How can we make Merge sort more efficient? 2. How does the Divide and Conquer Strategy work with Merge Sort? 3. Is merge sort adaptive or not? 4. When does the worst case occur in Merge Sort? |

|  |  |
| --- | --- |
| **Practical No: 04** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a menu driven program that implements singly linked list for the following operations: Create, insert, delete, reverse, concatenate |
| **Theory:** | **Linked List:**   * Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory. * A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory. * The last node of the list contains pointer to the null.   DS Linked List  **Uses of Linked List**   * The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space. * list size is limited to the memory size and doesn't need to be declared in advance. * Empty node can not be present in the linked list. * We can store values of primitive types or objects in the singly linked list.   **Why use linked list over array?**  Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.  Array contains following limitations:   1. The size of array must be known in advance before using it in the program. 2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time. 3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.   Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,   1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers. 2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.   **Singly linked list or One way chain**  Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.  One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.  Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.  DS Singly Linked List  In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.  In a single linked list, the insertion operation can be performed in three ways. They are as follows...  Inserting At Beginning of the list  Inserting At End of the list  Inserting At Specific location in the list  **Inserting At Beginning of the list**  We can use the following steps to insert a new node at beginning of the single linked list...  Step 1 - Create a newNode with given value.  Step 2 - Check whether list is Empty (head == NULL)  Step 3 - If it is Empty then, set newNode→next = NULL and head = newNode.  Step 4 - If it is Not Empty then, set newNode→next = head and head = newNode.  **Inserting At End of the list**  We can use the following steps to insert a new node at end of the single linked list...  Step 1 - Create a newNode with given value and newNode → next as NULL.  Step 2 - Check whether list is Empty (head == NULL).  Step 3 - If it is Empty then, set head = newNode.  Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.  Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).  Step 6 - Set temp → next = newNode.  **Inserting At Specific location in the list (After a Node)**  We can use the following steps to insert a new node after a node in the single linked list...  Step 1 - Create a newNode with given value.  Step 2 - Check whether list is Empty (head == NULL)  Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.  Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.  Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).  Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.  Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'  **Deletion**  In a single linked list, the deletion operation can be performed in three ways. They are as follows...  Deleting from Beginning of the list  Deleting from End of the list  Deleting a Specific Node  **Deleting from Beginning of the list**  We can use the following steps to delete a node from beginning of the single linked list...  Step 1 - Check whether list is Empty (head == NULL)  Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.  Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.  Step 4 - Check whether list is having only one node (temp → next == NULL)  Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)  Step 6 - If it is FALSE then set head = temp → next, and delete temp.  **Deleting from End of the list**  We can use the following steps to delete a node from end of the single linked list...  Step 1 - Check whether list is Empty (head == NULL)  Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.  Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.  Step 4 - Check whether list has only one Node (temp1 → next == NULL)  Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)  Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)  Step 7 - Finally, Set temp2 → next = NULL and delete temp1.  **Deleting a Specific Node from the list**  We can use the following steps to delete a specific node from the single linked list...  Step 1 - Check whether list is Empty (head == NULL)  Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.  Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.  Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.  Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.  Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not  Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).  Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).  Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.  Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).  Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).  Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).  **Displaying a Single Linked List**  We can use the following steps to display the elements of a single linked list...  Step 1 - Check whether list is Empty (head == NULL)  Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.  Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.  Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node  Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL). |
| **Source Code/Algorithm/Flow Chart:** | **Single Linked List:**  #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Function to swap the data of two node  void swap(struct Node\* a, struct Node\* b) {  int temp = a->data;  a->data = b->data;  b->data = temp;  }  // Function to create a new node  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = NULL;  return newNode;  }  // Function to add a node at the beginning of the linked list  void addToBeginning(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  newNode->next = \*head;  \*head = newNode;  }  // Function to add a node at the end of the linked list  void addToEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  return;  }  struct Node\* current = \*head;  while (current->next != NULL) {  current = current->next;  }  current->next = newNode;  }  // Function to add a new node after the nth node in the linked list  void addAfterNthNode(struct Node\*\* head, int n, int data) {  if (\*head == NULL) {  printf("List is empty.\n");  return;  }    if (n <= 0) {  printf("Invalid index.\n");  return;  }    struct Node\* current = \*head;  for (int i = 1; current != NULL && i < n; ++i) {  current = current->next;  }    if (current == NULL) {  printf("Index out of range.\n");  return;  }    struct Node\* newNode = createNode(data);  newNode->next = current->next;  current->next = newNode;  }  // Function to delete the first node  void deleteFirstNode(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is already empty.\n");  return;  }  struct Node\* temp = \*head;  \*head = (\*head)->next;  free(temp);  }  // Function to delete the last node  void deleteLastNode(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is already empty.\n");  return;  }  if ((\*head)->next == NULL) {  free(\*head);  \*head = NULL;  return;  }  struct Node\* current = \*head;  while (current->next->next != NULL) {  current = current->next;  }  free(current->next);  current->next = NULL;  }  // Function to delete the nth node from the start of the linked list  void deleteNthNode(struct Node\*\* head, int n) {  if (\*head == NULL) {  printf("List is empty.\n");  return;  }    if (n <= 0) {  printf("Invalid index.\n");  return;  }    if (n == 1) {  struct Node\* temp = \*head;  \*head = (\*head)->next;  free(temp);  return;  }    struct Node\* current = \*head;  for (int i = 1; current != NULL && i < n - 1; ++i) {  current = current->next;  }    if (current == NULL || current->next == NULL) {  printf("Index out of range.\n");  return;  }    struct Node\* temp = current->next;  current->next = temp->next;  free(temp);  }  // Function to merge two linked lists  struct Node\* mergeLists(struct Node\* list1, struct Node\* list2) {  if (list1 == NULL) {  return list2;  }  if (list2 == NULL) {  return list1;  }  struct Node\* mergedList = NULL;  if (list1->data < list2->data) {  mergedList = list1;  mergedList->next = mergeLists(list1->next, list2);  } else {  mergedList = list2;  mergedList->next = mergeLists(list1, list2->next);  }  return mergedList;  }  // Function to print the linked list  void printList(struct Node\* head) {  struct Node\* current = head;  while (current != NULL) {  printf("%d -> ", current->data);  current = current->next;  }  printf("NULL\n");  }  // Function to sort the linked list using bubble sort  void bubbleSort(struct Node\*\* head) {  if (\*head == NULL) {  return;  }  int swapped;  struct Node\* current;  struct Node\* last = NULL;  swapped = 1; // Initialize swapped to any non-zero value to enter the loop  while (swapped) {  swapped = 0;  current = \*head;  while (current->next != last) {  if (current->data > current->next->data) {  swap(current, current->next);  swapped = 1;  }  current = current->next;  }  last = current;  }  }  // Function to reverse a linked list  void reverseList(struct Node\*\* head) {  struct Node\* prevNode = NULL;  struct Node\* currentNode = \*head;  struct Node\* nextNode = NULL;  while (currentNode != NULL) {  nextNode = currentNode->next; // Store the next node  currentNode->next = prevNode; // Change next of current node  prevNode = currentNode; // Move prev to the current node  currentNode = nextNode; // Move current to the next node  }  \*head = prevNode; // Update the head to the last node (new start of the list)  }  // Prints the operations menu  void printMenu() {  printf("\nDoubly Linked List Operations:\n");  printf("1. Add Node to List 1 (Beginning)\n");  printf("2. Add Node to List 1 (End)\n");  printf("3. Add Node after nth Node in List 1\n");  printf("4. Delete First Node from List 1\n");  printf("5. Delete Last Node from List 1\n");  printf("6. Delete nth Node from List 1\n");  printf("7. Reverse List 1\n");  printf("8. Sort List 1\n");  printf("9. Print List 1\n");  printf("10. Add Node to List 2 (Beginning)\n");  printf("11. Add Node to List 2 (End)\n");  printf("12. Add Node after nth Node in List 2\n");  printf("13. Delete First Node from List 2\n");  printf("14. Delete Last Node from List 2\n");  printf("15. Delete nth Node from List 2\n");  printf("16. Reverse List 2\n");  printf("17. Sort List 2\n");  printf("18. Print List 2\n");  printf("19. Merge Lists\n");  printf("20. Add Node to Merged List (Beginning)\n");  printf("21. Add Node to Merged List (End)\n");  printf("22. Add Node after nth Node in Merged List\n");  printf("23. Delete First Node from Merged List\n");  printf("24. Delete Last Node from Merged List\n");  printf("25. Delete nth Node from Merged List\n");  printf("26. Reverse Merged List\n");  printf("27. Sort Merged List\n");  printf("28. Print Merged List\n");  printf("29. Exit\n");  }  int main() {  struct Node\* list1 = NULL;  struct Node\* list2 = NULL;  struct Node\* mergedList = NULL;  int choice, data, n;  while (1) {  printMenu();  printf("Enter your choice: ");  scanf("%d", &choice);  switch (choice) {  case 1:  printf("Enter data: ");  scanf("%d", &data);  addToBeginning(&list1, data);  break;  case 2:  printf("Enter data: ");  scanf("%d", &data);  addToEnd(&list1, data);  break;  case 3:  printf("Enter data: ");  scanf("%d", &data);  printf("Enter n: ");  scanf("%d", &n);  addAfterNthNode(&list1, n, data);  break;  case 4:  deleteFirstNode(&list1);  printf("First node deleted from List 1.\n");  break;  case 5:  deleteLastNode(&list1);  printf("Last node deleted from List 1.\n");  break;  case 6:  printf("Enter n: ");  scanf("%d", &n);  deleteNthNode(&list1, n);  printf("Node at position %d deleted from List 1.\n", n);  break;  case 7:  reverseList(&list1);  printf("List 1 reversed.\n");  break;  case 8:  bubbleSort(&list1);  printf("List 1 sorted.\n");  break;  case 9:  printf("List 1: ");  printList(list1);  break;  case 10:  printf("Enter data: ");  scanf("%d", &data);  addToBeginning(&list2, data);  break;  case 11:  printf("Enter data: ");  scanf("%d", &data);  addToEnd(&list2, data);  break;  case 12:  printf("Enter data: ");  scanf("%d", &data);  printf("Enter n: ");  scanf("%d", &n);  addAfterNthNode(&list2, n, data);  break;  case 13:  deleteFirstNode(&list2);  printf("First node deleted from List 2.\n");  break;  case 14:  deleteLastNode(&list2);  printf("Last node deleted from List 2.\n");  break;  case 15:  printf("Enter n: ");  scanf("%d", &n);  deleteNthNode(&list2, n);  printf("Node at position %d deleted from List 2.\n", n);  break;  case 16:  reverseList(&list2);  printf("List 2 reversed.\n");  break;  case 17:  bubbleSort(&list2);  printf("List 2 sorted.\n");  break;  case 18:  printf("List 2: ");  printList(list2);  break;  case 19:  mergedList = mergeLists(list1, list2);  printf("Lists merged.\n");  break;  case 20:  printf("Enter data: ");  scanf("%d", &data);  addToBeginning(&mergedList, data);  break;  case 21:  printf("Enter data: ");  scanf("%d", &data);  addToEnd(&mergedList, data);  break;  case 22:  printf("Enter data: ");  scanf("%d", &data);  printf("Enter n: ");  scanf("%d", &n);  addAfterNthNode(&mergedList, n, data);  break;  case 23:  deleteFirstNode(&mergedList);  printf("First node deleted from Merged List.\n");  break;  case 24:  deleteLastNode(&mergedList);  printf("Last node deleted from Merged List.\n");  break;  case 25:  printf("Enter n: ");  scanf("%d", &n);  deleteNthNode(&mergedList, n);  printf("Node at position %d deleted from Merged List.\n", n);  break;  case 26:  reverseList(&mergedList);  printf("Merged List reversed.\n");  break;  case 27:  bubbleSort(&mergedList);  printf("Merged List sorted.\n");  break;  case 28:  printf("Merged List: ");  printList(mergedList);  break;  case 29:  printf("Exiting...\n");  exit(0);  default:  printf("Invalid choice. Please try again.\n");  }  }    return 0;  } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Single Linked List and performed various operations on it. |
| **Post Lab Questions:** | 1. State and explain advantages and disadvantages of linked list 2. Why is merge sort a better option than quicksort for linked lists? 3. State the applications Of Linked List 4. How to detect loop in the singly linked list? |

|  |  |
| --- | --- |
| **Practical No: 05** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a menu driven program that implements circular linked list for the following operations: Create, insert, delete, reverse, concatenate. |
| **Theory:** | **Circular Linked List:**  In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.  We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.  The following image shows a circular singly linked list.  Circular Singly Linked List  Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.  **Memory Representation of circular linked list:**  In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.  However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.  Circular Singly Linked List  We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.  **Sample operations on circular linked list:**   1. Insertion      1. Deletion and Traversing     **Insertion in the circular linked list:**  A node can be added in three ways:   1. Insertion at the beginning of the list 2. Insertion at the end of the list 3. Insertion in between the nodes   1) **Insertion at the beginning of the list:**To insert a node at the beginning of the list, follow these steps:   * Create a node, say T. * Make T -> next = last -> next. * last -> next = T.   https://media.geeksforgeeks.org/wp-content/uploads/20220818092431/CircularSinglyLinkedlist4.png  Circular linked list before insertion  And then,  https://media.geeksforgeeks.org/wp-content/uploads/20220818092530/CircularSinglLinkedList5-660x168.png  Circular linked list after insertion  2) **Insertion at the end of the list:**  To insert a node at the end of the list, follow these steps:   * Create a node, say T. * Make T -> next = last -> next; * last -> next = T. * last = T.   Before insertion,  https://media.geeksforgeeks.org/wp-content/uploads/20220818093034/CircularSinglyLinkedlist6.png  Circular linked list before insertion of node at the end  After insertion,  https://media.geeksforgeeks.org/wp-content/uploads/20220818093259/CircularSinglyLinkedlist7-660x234.png  Circular linked list after insertion of node at the end  3) **Insertion in between the nodes:**To insert a node in between the two nodes, follow these steps:   * Create a node, say T. * Search for the node after which T needs to be inserted, say that node is P. * Make T -> next = P -> next; * P -> next = T.   Suppose 12 needs to be inserted after the node has the value 10,  https://media.geeksforgeeks.org/wp-content/uploads/20220818093642/circularll1-660x203.png  Circular linked list before insertion  After searching and insertion,  https://media.geeksforgeeks.org/wp-content/uploads/20220818093800/CircularSinglyLinkedList9-660x212.png  Circular linked list after insertion  **2. Deletion in a circular linked list:**  **1) Delete the node only if it is the only node in the circular linked list:**   * Free the node’s memory * The last value should be NULL. A node always points to another node, so NULL assignment is not necessary. Any node can be set as the starting point. Nodes are traversed quickly from the first to the last.   **2) Deletion of the last node:**   * Locate the node before the last node (let it be temp) * Keep the address of the node next to the last node in temp * Delete the last memory * Put temp at the end   **3) Delete any node from the circular linked list:**We will be given a node and our task is to delete that node from the circular linked list.  **Algorithm:** **Case 1**: List is empty.   1. If the list is empty, we will simply return.   **Case 2**: List is not empty   * If the list is not empty, then we define two pointers **curr**and **prev** and initialize the pointer **curr** with the **head**node. * Traverse the list using **curr** to find the node to be deleted and before moving to curr to the next node, every time set prev = curr. * If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr). * If the list has more than one node, check if it is the first node of the list. Condition to check this(curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr. * If curr is not the first node, we check if it is the last node in the list. Condition to check this is (curr -> next == head). * If curr is the last node. Set prev -> next = head and delete the node curr by free(curr). * If the node to be deleted is neither the first node nor the last node, then set prev -> next = curr -> next and delete curr. * If the node is not present in the list return head and don’t do anything. |
| **Source Code/Algorithm/Flow Chart:** | #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  struct Node\* head = NULL;  void create() {  int n, data;  struct Node \*newNode, \*temp;  printf("Enter the number of nodes: ");  scanf("%d", &n);  if (n <= 0) {  printf("Invalid number of nodes.\n");  return;  }  printf("Enter data for node 1: ");  scanf("%d", &data);  head = (struct Node\*)malloc(sizeof(struct Node));  head->data = data;  head->next = head;  temp = head;  int i;  for ( i = 2; i <= n; i++) {  printf("Enter data for node %d: ", i);  scanf("%d", &data);  newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = head;  temp->next = newNode;  temp = newNode;  }  printf("Circular linked list created.\n");  }  void insert() {  int data, position;  struct Node \*newNode, \*temp;  if (head == NULL) {  printf("List is empty!\n");  return;  }  printf("Enter data to be inserted: ");  scanf("%d", &data);  printf("Enter the position to insert: ");  scanf("%d", &position);  if (position < 1) {  printf("Invalid position.\n");  return;  }  newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  if (position == 1) {  newNode->next = head;  head = newNode;  struct Node\* last = head;  while (last->next != head) {  last = last->next;  }    last->next = newNode;  printf("Node inserted at the beginning.\n");  return;  }  temp = head;  int i;  for ( i = 1; i < position - 1; i++) {  temp = temp->next;  if (temp == head) {  printf("Invalid position.\n");  return;  }  }  newNode->next = temp->next;  temp->next = newNode;  printf("Node inserted at position %d.\n", position);  }  void delete() {  int position;  struct Node \*temp, \*prev;  if (head == NULL) {  printf("List is empty!\n");  return;  }  printf("Enter node position to delete: ");  scanf("%d", &position);  if (position < 1) {  printf("Invalid position.\n");  return;  }  if (position == 1) {  temp = head;  struct Node\* last = head;  while (last->next != head) {  last = last->next;  }    head = head->next;  last->next = head;  free(temp);  printf("Beginning node deleted.\n");    return;  }  temp = head;  prev = NULL;    int i;  for(i = 1; i < position; i++) {  prev = temp;  temp = temp->next;    if (temp == head) {  printf("Invalid position.\n");  return;  }  }  prev->next = temp->next;  free(temp);  printf("Node at position %d deleted.\n", position);  }  void reverse() {  struct Node \*current, \*prev, \*next;    if (head == NULL) {  printf("The list is empty. Please create the list first.\n");  return;  }    last->next = newNode;  printf("Node inserted at the beginning.\n");  return;  }  temp = head;  int i;  for ( i = 1; i < position - 1; i++) {  temp = temp->next;  if (temp == head) {  printf("Invalid position.\n");  return;  }  }  newNode->next = temp->next;  temp->next = newNode;  printf("Node inserted at position %d.\n", position);  }  void delete() {  int position;  struct Node \*temp, \*prev;  if (head == NULL) {  printf("List is empty!\n");  return;  }  printf("Enter node position to delete: ");  scanf("%d", &position);  if (position < 1) {  }  current = head;  prev = NULL;    while (current->next != head) {  next = current->next;  current->next = prev;  prev = current;  current = next;  }    current->next = prev;  head = current;  printf("List reversed.\n");  }  void concatenate() {  struct Node \*temp1, \*temp2;  if (head == NULL) {  printf("The list is empty. Please create the list first.\n");  return;  }  int n, data;  printf("Enter the number of nodes to concatenate: ");  scanf("%d", &n);  if (n <= 0) {  printf("Invalid number of nodes.\n");  return;  }  temp1 = head;  while (temp1->next != head) {  temp1 = temp1->next;  }    int i;  for (i = 1; i <= n; i++) {  printf("Enter data for node %d: ", i);  scanf("%d", &data);    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  newNode->data = data;  newNode->next = head;  temp1->next = newNode;  temp1 = newNode;  }  printf("List concatenated.\n");  }  void display() {  struct Node\* temp;  if (head == NULL) {  printf("List is empty.\n");  return;  }  temp = head;  printf("CLL: ");  do {  printf("%d ", temp->data);  temp = temp->next;  } while (temp != head);  printf("\n");  }  int main() {  int choice;  while (1) {  printf("\nCircular Linked List Operations\n");  printf("1. Create\n");  printf("2. Insert\n");  printf("3. Delete\n");  printf("4. Reverse\n");  printf("5. Concatenate\n");  printf("6. Display\n");  printf("7. Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);  switch (choice) {  case 1:  create();  break;  case 2:  insert();  break;  case 3:  delete();  break;  case 4:  reverse();  break;  case 5:  concatenate();  break;  case 6:  display();  break;  case 7:  exit(0);  break;  default:  printf("Invalid choice!\n");  }  }  return 0;  } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Circular Linked List and performed various operations on it. |
| **Post Lab Questions:** | * 1. What is the time complexity of searching for an element in a circular linked list?   2. State various applications of a circular linked list? |

|  |  |
| --- | --- |
| **Practical No: 06** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a menu driven program that implements doubly linked list for the following operations: Create, insert, delete, reverse, concatenate. |
| **Theory:** | **Double Linked List:**  Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.  Doubly linked list A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.  Doubly linked list In C, structure of a node in doubly linked list can be given as :    The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.  In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.  Memory Representation of a doubly linked list  Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).  In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.  We can traverse the list in this way until we find any node containing null or -1 in its next part.  Doubly linked list  Operations on doubly linked list  Following are the basic operations supported by a list.    **Insertion at the Beginning**  In this operation, we create a new node with three compartments, one containing the data, the others containing the address of its previous and next nodes in the list. This new node is inserted at the beginning of the list.  **Algorithm**  1. START  2. Create a new node with three variables: prev, data, next.  3. Store the new data in the data variable  4. If the list is empty, make the new node as head.  5. Otherwise, link the address of the existing first node to the next variable of the new node, and assign null to the prev variable.  6. Point the head to the new node.  7. END  **Deletion at the Beginning**  This deletion operation deletes the existing first nodes in the doubly linked list. The head is shifted to the next node and the link is removed.  **Algorithm**  1. START  2. Check the status of the doubly linked list  3. If the list is empty, deletion is not possible  4. If the list is not empty, the head pointer is shifted to the next node.  5. END  **Insertion at the End**  In this insertion operation, the new input node is added at the end of the doubly linked list; if the list is not empty. The head will be pointed to the new node, if the list is empty.  **Algorithm:**  1. START  2. If the list is empty, add the node to the list and point the head to it.  3. If the list is not empty, find the last node of the list.  4. Create a link between the last node in the list and the new node.  5. The new node will point to NULL as it is the new last node.  6. END |
| **Source Code/Algorithm/Flow Chart:** | Double Linked List:  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* prev;  struct Node\* next;  };  struct Node\* createNode(int data) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Underflow");  }  newNode->data = data;  newNode->prev = NULL;  newNode->next = NULL;  return newNode;  }    void insertAtBeginning(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  newNode->next = \*head;  if (\*head != NULL) {  (\*head)->prev = newNode;  }  \*head = newNode;  }  void insertAtEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  struct Node\* current = \*head;  if (\*head == NULL) {  \*head = newNode;  return;  }  while (current->next != NULL) {  current = current->next;  }  current->next = newNode;  newNode->prev = current;  }  //insert node at any position  void insertAtPosition(struct Node\*\* head, int data, int position) {  if (position < 0) {  printf("Position not found\n");  return;  }  if (position == 0 || \*head == NULL) {  insertAtBeginning(head, data);  return;  }  struct Node\* newNode = createNode(data);  struct Node\* current = \*head;  int currentPosition = 0;  while (currentPosition < position - 1 && current->next != NULL) {  current = current->next;  currentPosition++;  }  newNode->next = current->next;  if (current->next != NULL) {  current->next->prev = newNode;  }  current->next = newNode;  newNode->prev = current;  }  void deleteFirstNode(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Nothing to delete.\n");  return;  }  struct Node\* temp = \*head;  \*head = (\*head)->next;  if (\*head != NULL) {  (\*head)->prev = NULL;  }  free(temp);  }  void deleteLastNode(struct Node\*\* head)  {  if(\*head==NULL)  {  printf("List is empty.Nothing to delete.\n");  return;  }  struct Node\* current = \*head;  while (current->next != NULL) {  current = current->next;  }  if (current->prev != NULL) {  current->prev->next = NULL;  }  else {    \*head = NULL;  }  free(current);  }  void deleteAtPosition(struct Node\*\* head, int position) {  if (position < 0) {  printf("Invalid position\n");  return;  }  if (position == 0 || \*head == NULL) {  printf("Invalid position or empty list\n");  return;  }  struct Node\* current = \*head;  int currentPosition = 0;  while (currentPosition < position && current != NULL) {  current = current->next;  currentPosition++;  }  if (current == NULL) {  printf("Position out of range\n");  return;  }  if (current->prev != NULL) {  current->prev->next = current->next;  }  if (current->next != NULL) {  current->next->prev = current->prev;  }  free(current);  }  void concat(struct Node\*\* head1, struct Node\*\* head2) {  if (\*head1 == NULL) {  \*head1 = \*head2;  } else if (\*head2 != NULL) {  struct Node\* current = \*head1;  while (current->next != NULL) {  current = current->next;  }  current->next = \*head2;  (\*head2)->prev = current;  }  }  void reverse(struct Node \*\*head) {  struct Node \*current = \*head;  struct Node \*temp = NULL;    while (current != NULL) {    temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  if (temp != NULL) {  \*head = temp->prev;  }  }  void display(struct Node\* head) {  struct Node\* current = head;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  }  void bubbleSort(struct Node\*\* head) {  int swapped;  struct Node\* current;  struct Node\* last = NULL;    if (\*head == NULL)  return;  do {  swapped = 0;  current = \*head;  while (current->next != last) {  if (current->data > current->next->data) {    int temp = current->data;  current->data = current->next->data;  current->next->data = temp;  swapped = 1;  }  current = current->next;  }  last = current;  } while (swapped);  }  void search(struct Node\* head, int value) {  struct Node\* current = head;  int pos=1;  while (current != NULL) {  if (current->data == value){  printf("%d found at %d position\n",value,pos);  break;  }  current = current->next;  pos++;  }  }  int ispresent(struct Node \*\*head, int target)  {  struct Node \*temp=\*head;    while (temp != NULL)  {  if (temp->data == target)  {  return 1;  }  temp = temp->next;  }  return 0;  }  struct Node \*intersection(struct Node \*\*head1, struct Node \*\*head2)  {  struct Node \*result = NULL;  struct Node \*t1 = \*head1;  while (t1 != NULL) {  if (ispresent(head2, t1->data)==1){  insertAtBeginning(&result, t1->data);  }  t1 = t1->next;  }  return result;  }  int main(){  struct Node \*head1 = NULL;  struct Node \*head2 = NULL;  struct Node \*inter = NULL;  int temp;  insertAtEnd(&head1, 10);  insertAtEnd(&head1, 20);  insertAtEnd(&head1, 30);  insertAtEnd(&head1, 40);  insertAtEnd(&head1, 50);    insertAtEnd(&head2, 10);  insertAtEnd(&head2, 200);  insertAtEnd(&head2, 30);    display(head1);  printf("\n");  search(head1,30);  concat(&head1,&head2);  display(head1);  reverse(&head1);  printf("\n");  inter=intersection(&head1,&head2);    return 0;  } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Double Linked List and performed various operations on it. |
| **Post Lab Questions:** | * Explain what a doubly linked list is and how it differs from a singly linked list? * Are there any disadvantages to using a doubly linked list over a singly linked list? |

|  |  |
| --- | --- |
| **Practical No: 07** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a menu-driven program to  i. Create a binary search tree  ii. Traverse the tree in inorder, preorder and postorder  iii. Search  iv. delete a node from the tree. |
| **Theory:** | **What is a tree?** A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially. **What is a Binary Search tree?** A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.Let's understand the concept of Binary search tree with an example.    In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.  Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.  Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.    In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree. **Advantages of Binary search tree**  * Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element. * As compared to array and linked lists, insertion and deletion operations are faster in BST.  **Example of creating a binary search tree** Now, let's see the creation of binary search tree using an example.  Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**   * First, we have to insert **45** into the tree as the root of the tree. * Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element. * Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.   Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -  **Step 1 - Insert 45.**  Binary Search tree  **Step 2 - Insert 15.**  As 15 is smaller than 45, so insert it as the root node of the left subtree.    **Step 3 - Insert 79.**  As 79 is greater than 45, so insert it as the root node of the right subtree.    **Step 4 - Insert 90.**  90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.    **Step 5 - Insert 10.**  10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.    **Step 6 - Insert 55.**  55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.    **Step 7 - Insert 12.**  12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.    **Step 8 - Insert 20.**  20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.    **Step 9 - Insert 50.**  50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.    Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.  We can perform insert, delete and search operations on the binary search tree.  Let's understand how a search is performed on a binary search tree. **Searching in Binary search tree** Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -   1. First, compare the element to be searched with the root element of the tree. 2. If root is matched with the target element, then return the node's location. 3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree. 4. If it is larger than the root element, then move to the right subtree. 5. Repeat the above procedure recursively until the match is found. 6. If the element is not found or not present in the tree, then return NULL.   Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.  **Step1:**  Binary Search tree  **Step2:**  Binary Search tree  **Step3:**  Binary Search tree  Now, let's see the algorithm to search an element in the Binary search tree.   **Deletion in Binary Search tree** In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -   * The node to be deleted is the leaf node, or, * The node to be deleted has only one child, and, * The node to be deleted has two children   We will understand the situations listed above in detail. **When the node to be deleted is the leaf node** It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.  We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.  Binary Search tree **When the node to be deleted has only one child** In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.  We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.  So, the replaced node 79 will now be a leaf node that can be easily deleted.  Binary Search tree **When the node to be deleted has two children** This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -   * First, find the inorder successor of the node to be deleted. * After that, replace that node with the inorder successor until the target node is placed at the leaf of tree. * And at last, replace the node with NULL and free up the allocated space.   The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.  We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.  Binary Search tree  Now let's understand how insertion is performed on a binary search tree. **Insertion in Binary Search tree** A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.  Now, let's see the process of inserting a node into BST using an example.  Binary Search tree  Binary Search tree **The complexity of the Binary Search tree** Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.   1. Time Complexity      1. Space complexity:     **In order traversal:**  **Steps**  1.Traverse the left sub-tree in in-order  2.Visit the root  3.Traverse the right sub-tree in in-order  **Algorithm**  Step 1: Repeat Steps 2 to 4 while TREE != NULL  Step 2: INORDER(TREE -> LEFT)  Step 3: Write TREE -> DATA  Step 4: INORDER(TREE -> RIGHT)  [END OF LOOP]  Step 5: END  **Pre-order traversal:**  **Steps**  1.Visit the root node  2.traverse the left sub-tree in pre-order  3.traverse the right sub-tree in pre-order  **Algorithm**  Step 1: Repeat Steps 2 to 4 while TREE != NULL  Step 2: Write TREE -> DATA  Step 3: PREORDER(TREE -> LEFT)  Step 4: PREORDER(TREE -> RIGHT)  [END OF LOOP]  Step 5: END  **Post-order traversal:**  **Steps:**  1. Traverse the left subtree by calling the postorder function recursively.  2. Traverse the right subtree by calling the postorder function recursively.  3. Access the data part of the current node.  **Algorithm**:  Step 1: Repeat Steps 2 to 4 while TREE != NULL  Step 2: POSTORDER(TREE -> LEFT)  Step 3: POSTORDER(TREE -> RIGHT)  Step 4: Write TREE -> DATA  [END OF LOOP]  Step 5: END |
| **Source Code/Algorithm/Flow Chart:** | Binary Search Tree:  #include<stdio.h>  #include<stdlib.h>  //creating a binary search tree  struct node{  struct node \*left, \*right;  int data;  };  //performing basic operations  struct node\* newnode(int x)  {  struct node \*temp = malloc(sizeof(struct node));  temp->data = x;  temp->left = NULL;  temp->right = NULL;  return temp;  }  struct node\* insert(struct node \*root, int x)  {  if(root == NULL)  {  return newnode(x);  }  else if(x > root->data)  {  root->right = insert(root->right, x);  }  else if(x <= root->data)  {  root->left = insert(root->left, x);  }  return root;  }  struct node\* del(struct node \*root, int x)  {  if(root == NULL)  {  return NULL;  }  else if(x > root->data)  {  root->right = del(root->right, x);  }  else if(x < root->data)  {  root->left = del(root->left, x);  }  else  {  // Case 1: Node with only one child or no child  if (root->left == NULL)  {  struct node \*temp = root->right;  free(root);  return temp;  }  else if (root->right == NULL)  {  struct node \*temp = root->left;  free(root);  return temp;  }  // Case 2: Node with two children    struct node\* temp = root->right;  while (temp->left != NULL)  {  temp = temp->left;  }  root->data = temp->data;  root->right = del(root->right, temp->data);  }  return root;  }  int height(struct node \*root)  {  if(root==NULL)  {  return 0;  }  else{  int lh=height(root->left);  int rh=height(root->right);    if(rh>lh)  {  return (rh+1);  }    else{  return (lh+1);  }  }    }  void pre(struct node \*root)  {  if(root!=NULL){  printf("%d ",root->data);  pre(root->left);  pre(root->right);  }  }  void in(struct node \*root)  {  if(root!=NULL){  in(root->left);  printf("%d ",root->data);  in(root->right);  }  }  void post(struct node \*root)  {  if(root!=NULL){  post(root->left);  post(root->right);  printf("%d ",root->data);  }  }  int main()  {  int choice=0;  struct node \*root=NULL;  while(choice!=7)  {  printf("\n\nMenu\nEnter 1 to create BST\nEnter 2 to insert child\nEnter 3 to delete child\nEnter 4 Preorder Traversal\nEnter 5 Inorder traversal\nEnter 6 PostOrder Traversal\nEnter 7 to Exit\n");  printf("\nEnter your choice : ");  scanf("%d",&choice);  switch(choice)  {  case 1:{    printf("\nEnter value of root node : ");  int x;  scanf("%d",&x);  root=newnode(x);  break;  }    case 2:{  printf("How many nodes do you wish to enter : ");  int n;  scanf("%d",&n);  int i,a[n];  printf("Enter values of nodes : ");  for(i=0;i<n;i++)  {  scanf("%d",&a[i]);  root=insert(root,a[i]);  }  break;  }    case 3:{  int z;  printf("Enter value of element to be deleted : ");  scanf("%d",&z);  root=del(root,z);  break;  }    case 4:  printf("\nPreorder Traversal : ");  pre(root);  break;    case 5:  printf("\nInorder Traversal : ");  in(root);  break;    case 6:  printf("\nPostorder Traversal : ");  post(root);  break;    case 7:  break;    default:  printf("\nNo such option.");  }  }    return 0;    } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Binary Search Tree and performed given operations on it. |
| **Post Lab Questions:** | * What is a binary search tree? * What is a self-balanced tree? * What is the Red-Black tree data structure? |

|  |  |
| --- | --- |
| **Practical No: 08** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a program to insert and delete nodes in the graph using and adjacency list. |
| **Theory:** | A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:   * **Sequential representation** (or, Adjacency matrix representation) * **Linked list representation** (or, Adjacency list representation)   In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph. **Sequential representation** In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.  If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.  An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -  aij = 1 {if there is a path exists from Vi to Vj}  aij = 0 {Otherwise}  It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.  If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.  Now, let's see the adjacency matrix representation of an undirected graph.  Graph Representation  In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.  There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj. **Adjacency matrix for a directed graph** In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.  Consider the below-directed graph and try to construct the adjacency matrix of it.    In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.  **Adjacency matrix for a weighted directed graph**  It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.    In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.  Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.  Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space. Linked list representation An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.  Let's see the adjacency list representation of an undirected graph.  Graph Representation  In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.  An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.  The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.  Now, consider the directed graph, and let's see the adjacency list representation of that graph.  Graph Representation  For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.  Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.    In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.  In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space. |
| **Source Code/Algorithm/Flow Chart:** | #include<stdio.h>  #define V 4  // Function to create a matrix of size V x V  void create(int arr[V][V]) {  int i, j;  for(i = 0; i < V; i++) {  for(j = 0; j < V; j++) {  arr[i][j] = 0;  }  }  }  // Function to add an edge with weight to the graph  void addEdge(int arr[V][V]) {  int i, j, weight;  printf("Enter both vertices of the edge: ");  scanf("%d %d", &i, &j);  printf("Enter weight of the edge: ");  scanf("%d", &weight);  arr[i][j] = weight;  arr[j][i] = weight;  }  // Function to print the Adjency Matrix of the graph  void printMatrix(int arr[V][V]) {  int i, j;  for(i = 0; i < V; i++) {  printf("\n");  for(j = 0; j < V; j++) {  printf("%d ", arr[i][j]);  }  }  }  int main() {  int matrix[V][V];    // Creating the matrix  create(matrix);    int choice = -1;  while(choice!=0) {  printf("\nEnter 1 to add an Edge\nEnter 2 to print the Adjency Matrix\nEnter 0 to quit\nEnter choice: ");  scanf("%d", &choice);  switch(choice) {  case 0:  printf("Exiting...");  break;  case 1:  addEdge(matrix);  break;  case 2:  printf("Adjacency Matrix: ");  printMatrix(matrix);  break;  default:  printf("Invalid input!");  }  }    return 0;  } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Graph with adjacency list. |
| **Post Lab Questions:** | Given the adjacency list and number of vertices and edges of a graph, the task is to represent the adjacency list for a directed graph.  Input: V = 3, edges[][]= {{0, 1}, {1, 2} {2, 0}} |

|  |  |
| --- | --- |
| **Practical No: 09** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a program in C to implement Depth First Search. |
| **Theory:** | Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.  DFS algorithm  Now, let's start examining the graph starting from Node H.  **Step 1** - First, push H onto the stack.   1. STACK: H   **Step 2** - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.   1. Print: H]STACK: A   **Step 3** - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.   1. Print: A 2. STACK: B, D   **Step 4** - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.   1. Print: D 2. STACK: B, F   **Step 5** - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.   1. Print: F 2. STACK: B   **Step 6** - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.   1. Print: B 2. STACK: C   **Step 7** - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.   1. Print: C 2. STACK: E, G   **Step 8** - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.   1. Print: G 2. STACK: E   **Step 9** - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.   1. Print: E 2. STACK:   Now, all the graph nodes have been traversed, and the stack is empty.  The step-by-step process to implement the DFS traversal is given as follows -   1. First, create a stack with the total number of vertices in the graph. 2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack. 3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack. 4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top. 5. If no vertex is left, go back and pop a vertex from the stack. 6. Repeat steps 2, 3, and 4 until the stack is empty.   **Applications of DFS algorithm**  The applications of using the DFS algorithm are given as follows -   * DFS algorithm can be used to implement the topological sorting. * It can be used to find the paths between two vertices. * It can also be used to detect cycles in the graph. * DFS algorithm is also used for one solution puzzles. * DFS is used to determine if a graph is bipartite or not.   **Algorithm**  **Step 1:** SET STATUS = 1 (ready state) for each node in G  **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)  **Step 3:** Repeat Steps 4 and 5 until STACK is empty  **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)  **Step 5:** Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)  [END OF LOOP]  **Step 6:** EXIT |
| **Source Code/Algorithm/Flow Chart:** | #include <stdio.h>  #include <stdbool.h>  #define MAX 100  int graph[MAX][MAX], stack[MAX];  int vertices, top = -1;  bool visited[MAX];  // Function to push a vertex to the stack  void push(int vertex) {  stack[++top] = vertex;  }  // Function to pop a vertex from the stack  int pop() {  return stack[top--];  }  // Function to check if the stack is empty  bool isEmpty() {  return top == -1;  }  // Function to initialize the stack  void init() {  top = -1;  int i;  for (i = 0; i < MAX; i++) {  visited[i] = false;  }  }  // Function to implement DFS algorithm  void depthFirstSearch(int start) {  push(start);  visited[start] = true;    while (!isEmpty()) {  int current = pop();  printf("%d ", current);    int i;  for (i = 0; i < vertices; i++) {  if (graph[current][i] && !visited[i]) {  push(i);  visited[i] = true;  }  }  }  }  // Main function  int main() {  printf("Enter the number of vertices: ");  scanf("%d", &vertices);    int i, j;  printf("Enter the adjacency matrix:\n");  for (i = 0; i < vertices; i++) {  for (j = 0; j < vertices; j++) {  scanf("%d", &graph[i][j]);  }  }  int start;  printf("Enter start vertex: ");  scanf("%d", &start);  printf("DFS: ");  init();  depthFirstSearch(start);  return 0;  } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Depth First Search. |
| **Post Lab Questions:** | * State the differences between DFS and BFS. * Find the DFS and BFS of following tree |

|  |  |
| --- | --- |
| **Practical No: 10** | |
|  | |
| **Explanation/ Stepwise-Procedure/ Algorithm** | Write a program in C to implement prims algorithm for a given directed Graph. |
| **Theory:** | **Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.  Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.  How does the prim's algorithm work?  Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -   * First, we have to initialize an MST with the randomly chosen vertex. * Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree. * Repeat step 2 until the minimum spanning tree is formed.   The applications of prim's algorithm are -   * Prim's algorithm can be used in network designing. * It can be used to make network cycles. * It can also be used to lay down electrical wiring cables.   Example of prim's algorithm  Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.  Suppose, a weighted graph is -    **Step 1 -** First, we have to choose a vertex from the above graph. Let's choose B.    **Step 2 -** Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.    **Step 3 -** Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.    **Step 4 -** Now, select the edge CD, and add it to the MST.    **Step 5 -** Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.    So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -  Cost of MST = 4 + 2 + 1 + 3 = 10 units.  **Algorithm**    **Complexity of Prim's algorithm**  Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -    Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires O(|V|2) running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.  The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices. |
| **Source Code/Algorithm/Flow Chart:** | #include <stdio.h>  #include <stdbool.h>  #include <limits.h>  #define V 5  int minKey(int key[], bool mstSet[]) {  int min = INT\_MAX, min\_index;  for (int v = 0; v < V; v++) {  if (mstSet[v] == false && key[v] < min) {  min = key[v];  min\_index = v;  }  }  return min\_index;  }  void printMST(int parent[], int graph[V][V]) {  printf("Edge \tWeight\n");  for (int i = 1; i < V; i++) {  printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);  }  }  void primMST(int graph[V][V]) {  int parent[V];  int key[V];  bool mstSet[V];  for (int i = 0; i < V; i++) {  key[i] = INT\_MAX;  mstSet[i] = false;  }  key[0] = 0;  parent[0] = -1;  for (int count = 0; count < V - 1; count++) {  int u = minKey(key, mstSet);  mstSet[u] = true;  for (int v = 0; v < V; v++) {  if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {  parent[v] = u;  key[v] = graph[u][v];  }  }  }  printMST(parent, graph);  }  int main() {  int graph[V][V] = {  {0, 2, 0, 6, 0},  {2, 0, 3, 8, 5},  {0, 3, 0, 0, 7},  {6, 8, 0, 0, 9},  {0, 5, 7, 9, 0}  };  primMST(graph);  return 0;  } |
| **Output Screenshots (if applicable)** |  |
| **Conclusion** | Thus, we have studied and implemented Prims algorithm for a given directed graph. |
| **Post Lab Questions:** | Use prims algorithm to find the minimum spanning tree for the following graph: |