

P.G. DEPARTMENT OF COMPUTER SCIENCE

[SNDT Women's University](https://sndt.ac.in/)

Master Of Computer Application

SUBJECT : ADVANCED DATA STRUCTURE.

PRACTICAL JOURNAL 2022

FYMCA SEM – 1

CHUMKI KARTICK PANJA

MC2140

**INDEX**

|  |  |  |
| --- | --- | --- |
| **SrNo.** | **Practicals** | **sign** |
| 1 | Code for insertion sort. |  |
| 2 | Code for bubble sort. |  |
| 3 | Code for selection sort. |  |
| 4 | Code for shell sort. |  |
| 5 | Code for linear search. |  |
| 6 | Code for binary search. |  |
| 7 | Code for implementation of array with insertion , deletion of element and traversing through the array. |  |
| 8 | Code for implementation of stack using array . |  |
| 9 | Code for implementation of stack using linked list |  |
| 10 | Code for implementation of single linked list with insertion , deletion of element at the start,at given position,at the end of linked list and traversing . |  |
| 11 | Code for implementation of doubly linked list with insertion , deletion of element at the start,at given position,at the end of linked list and traversing . |  |
| 12 | Code for implementation of circular linked list with insertion , deletion of element at the start,at given position,at the end of linked list and traversing . |  |
| 13 | Code for implementation of queue in an array . |  |
| 14 | Code for implementation of double queue in an array . |  |
| 15 | Code for implementation of circular queue . |  |
| 16 | Code for implementation of priority queue in an array . |  |
| 17 | Code for implementation of binary search tree, inorder ,postorder and preorder traversal using linked list. |  |
| 18 | Code for implementation of graph. |  |
| 19 | Code for implementation of BFS |  |
| 20 | Code for implementation of DFS |  |

Program 1: Code for insertion sort.

//insertionsort.c

#include<stdio.h>

#include<stdlib.h>

int main()

{

   int counter1,counter2,chk,temp\_val,val[100];

   printf("Please enter the total count of the elements that you want to sort: \n");

   scanf("%d",&chk);

   printf("Please input the elements that has to be sorted:\n");

   for(counter1=0;counter1<chk;counter1++)

    {

      scanf("%d",&val[counter1]);

    }

    for(counter1=1;counter1<=chk-1;counter1++)

    {

        temp\_val=val[counter1];

        counter2=counter1-1;

        while((temp\_val<val[counter2])&&(counter2>=0))

        {

           val[counter2+1]=val[counter2];

           counter2=counter2-1;

        }

       val[counter2+1]=temp\_val;

    }

    printf("\nOutput generated after using insertion sort: \n");

    for(counter1=0;counter1<chk;counter1++)

    {

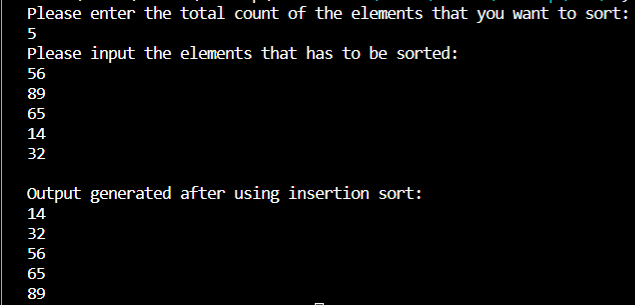
       printf("%d\n",val[counter1]);

    }

    return 0;

}

OUTPUT:



Program 2: Code for bubble sort.

//bubblesort.c

#include<stdlib.h>

#include <stdio.h>

int main()

{

  int total\_count, counter, counter1, swap\_var;

  int array[20];

  printf("How many number you want to input?\n");

  scanf("%d", &total\_count);

  printf("Please enter %d integers that has to be sorted\n", total\_count);

  for (counter = 0; counter < total\_count; counter++)

  scanf("%d", &array[counter]);

  for (counter = 0 ; counter < total\_count - 1; counter++)

    {

      for (counter1 = 0 ; counter1 < total\_count - counter - 1; counter1++)

        {

            if  (array[counter1] > array[counter1+1]) /\* For decreasing order use < \*/

            {

                swap\_var        = array[counter1];

                array[counter1]   = array[counter1+1];

                array[counter1+1] = swap\_var;

            }

        }

    }

  printf("Below is the list of elements sorted in ascending order:\n");

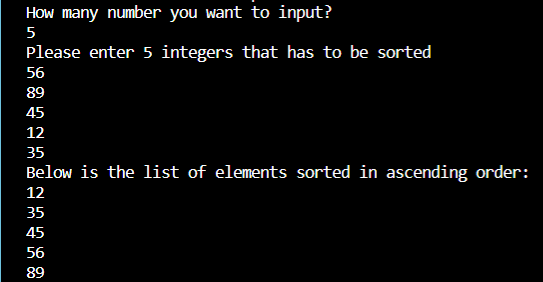
  for (counter = 0; counter < total\_count; counter++)

  printf("%d\n", array[counter]);

  return 0;

}

OUTPUT:



Program 3: Code for selection sort.

//selectionsort.c

#include<stdio.h>

#include<conio.h>

void main()

{

   int total\_count,counter1,counter2,minimum,temp\_value;

   int a[20];

   printf("Enter the Number of Elements:\n");

   scanf("%d",&total\_count);

   printf("Enter %d Elements:\n",total\_count);

   for(counter1=0;counter1<total\_count;counter1++)

    {

     scanf("%d",&a[counter1]);

    }

   for(counter1=0;counter1<total\_count-1;counter1++)

    {

      minimum=counter1;

      for(counter2=counter1+1;counter2<total\_count;counter2++)

        {

          if(a[minimum]>a[counter2])

          minimum=counter2;

        }

      if(minimum!=counter1)

        {

          temp\_value=a[counter1];

          a[counter1]=a[minimum];

          a[minimum]=temp\_value;

        }

    }

   printf("The Sorted array in ascending order:\n");

   for(counter1=0;counter1<total\_count;counter1++)

    {

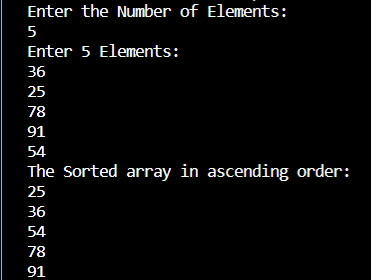
       printf("%d\n",a[counter1]);

    }

   getch();

}

OUTPUT:



Program 4: Code for shell sort.

//shellsort.c

#include <stdio.h>

void shellSort(int array[], int n) {

  // Rearrange elements at each n/2, n/4, n/8, ... intervals

  for (int interval = n / 2; interval > 0; interval /= 2) {

    for (int i = interval; i < n; i += 1) {

      int temp = array[i];

      int j;

      for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {

        array[j] = array[j - interval];

      }

      array[j] = temp;

    }

  }

}

void printArray(int array[], int size) {

  for (int i = 0; i < size; ++i) {

    printf("%d  ", array[i]);

  }

  printf("\n");

}

int main() {

  int data[] = {9, 8, 3, 7, 5, 6, 4, 1};

  int size = sizeof(data) / sizeof(data[0]);

  shellSort(data, size);

  printf("Sorted array: \n");

  printArray(data, size);

}

OUTPUT:



Program 5: Code for linear search.

//linearsearch.c

#include<stdio.h>

#include<conio.h>

#define MAX\_SIZE 5

int main()

{

  int arr\_search[MAX\_SIZE], i, element;

  printf("\nEnter %d Elements for Searching : \n", MAX\_SIZE);

  scanf("%d", &arr\_search[i]);

  for (i = 0; i < MAX\_SIZE; i++)

  {

    scanf("%d", &arr\_search[i]);

  }

  printf("Enter Element to Search : ");

  scanf("%d", &element);

  /\* for : Check elements one by one - Linear \*/

  for (i = 0; i < MAX\_SIZE; i++)

  {

    /\* If for Check element found or not \*/

    if (arr\_search[i] == element)

    {

      printf("Linear Search : %d is Found at array : %d.\n", element, i + 1);

      break;

    }

  }

 if (i == MAX\_SIZE)

  {

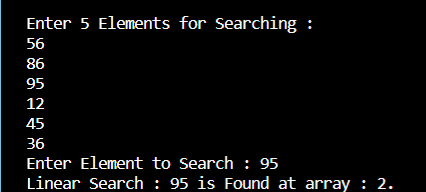
   printf("\nSearch Element : %d  : Not Found \n", element);

  }

  getch();

}

OUTPUT:



Program 6: Code for binary search.

//binarysearch.c

#include<stdio.h>

#define MAX\_SIZE 5

int main()

{

    int arr\_search[MAX\_SIZE], i,element, f = 0, r =  MAX\_SIZE, mid;

    printf("Enter %d Elements for Searching : \n", MAX\_SIZE);

    scanf("%d", &arr\_search[i]);

    for (i = 0; i < MAX\_SIZE; i++)

    {

      scanf("%d", &arr\_search[i]);

    }

    printf("Enter Element to Search : ");

    scanf("%d", &element);

    while (f <= r)

    {

      mid = (f+r)/2;

      if (arr\_search[mid] == element)

        {

         printf("\nSearch Element  : %d  : Found :  Position : %d.\n", element, mid+1);

         break;

        }

      else if (arr\_search[mid] < element)

        {

          f = mid + 1;

        }

      else

        {

         r = mid - 1;

        }

    }

   if (f > r)

    {

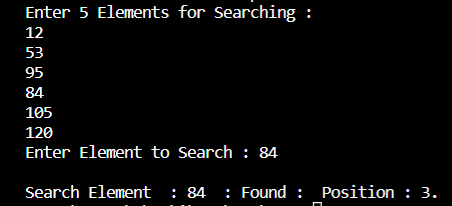
      printf("\nSearch Element : %d  : Not Found \n", element);

    }

    return 0 ;

}

OUTPUT:



Program 7: Code for implementation of array with insertion , deletion of element and traversing through the array.

// implemention of array

#include <stdio.h>

int main()

{

  int values[5];

  printf("Enter 5 integers:\n");

  // taking input and storing it in an array

  for(int i = 0; i < 5; ++i)

  {

    scanf("%d", &values[i]);

  }

  printf("Displaying integers:\n");

  // printing elements of an array

  for(int i = 0; i < 5; ++i)

  {

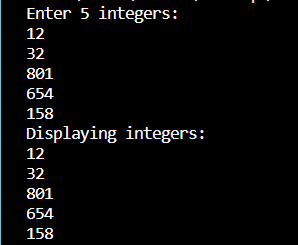
    printf("%d\n", values[i]);

  }

  return 0;

}

OUTPUT:



Program 8: Code for implementation of stack using array .

//implemention of stack using array

#include <stdio.h>

#include<stdlib.h>

#define MAX 4  //#define preprocessor directive to declare this constant

int stack\_arr[MAX];  //global declaration of stack\_arr[];

int top = -1;

void push(int data)

{

    if(top == MAX-1)

    {

        printf("Stack overflow\n");

        return;

    }

    top = top+1;

    stack\_arr[top] = data;

}

int pop()

{

    int value;

    if(top == -1)

    {

        printf("stack underflow\n");

        exit(1); //means abnormal termination of the program

    }

    value = stack\_arr[top];

    top = top-1;

    return value;

}

void print()

{

    int i;

    if(top == -1)

    {

        printf("stack underflow\n");

        return;

    }

    for(i=top ; i>=0 ; i--)

    printf("%d\n",stack\_arr[i]);

    printf("\n");

}

int main()

{

    int data;

    printf("Inserting the data:\n");

    push(1);

    push(2);

    push(3);

    push(4);

    print();

    printf("Inserting data 5\n");

    push(5);

    printf("\n");

    printf("Deleting the data:\n");

    data = pop();

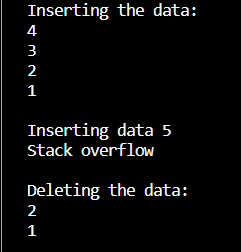
    data = pop();

    print();

    return 0;

}

OUTPUT:



Program 9: Code for implementation of stack using linked list .

//Stack using Linked List

#include <stdio.h>

#include <stdlib.h>

struct node {

    int data;

    struct node \*next;

}\*top;

/\*

Initialize an empty stack

\*/

void initialize() {

    top = NULL;

}

/\*

Checks if Stack is empty or not

\*/

int isEmpty() {

    if (top == NULL)

        return 1;

    else

        return 0;

}

/\*

Returns the top element of Stack

\*/

int peek()

{

    return top->data;

}

/\* Count stack elements \*/

int getStackSize(struct node \*head){

    /\* Input Validation \*/

    if (head == NULL) {

       printf("Error : Invalid stack pointer !!!\n");

       return;

    }

    int length = 0;

    while(head != NULL){

        head = head->next;

        length++;

    }

    return length;

}

/\*

Push an Element in Stack

\*/

void push(int num) {

    struct node \*temp;

    temp =(struct node \*)malloc(1\*sizeof(struct node));

    temp->data = num;

    if (top == NULL) {

        top = temp;

        top->next = NULL;

    } else {

        temp->next = top;

        top = temp;

    }

}

/\*

Pop Operation: Removes Top Element of the Stack

\*/

void pop() {

    struct node \*temp;

    if (isEmpty(top)) {

        printf("\nStack is Empty\n");

        return;

    } else {

        temp = top;

        top = top->next;

        printf("Removed  Element : %d\n", temp->data);

        free(temp);

    }

}

/\*

 Prints the linked list representation of a stack

\*/

void printStack(struct node \*nodePtr) {

  while (nodePtr != NULL) {

     printf("%d", nodePtr->data);

     nodePtr = nodePtr->next;

     if(nodePtr != NULL)

         printf("-->");

  }

  printf("\n");

}

void main() {

   /\* Initialize Stack \*/

   initialize();

   /\* Push Elements in stack \*/

   push(1);

   push(2);

   push(3);

   push(4);

   /\* Prints Size of Stack \*/

   printf("Stack Size : %d\n", getStackSize(top));

   /\* Printing top element of Stack \*/

   printf("\nTop Element : %d\n", peek());

   /\* Printing Stack \*/

   printf("Stack as linked List\n");

   printStack(top);

   /\* Removing elements from stack \*/

   pop();

   pop();

   pop();

   pop();

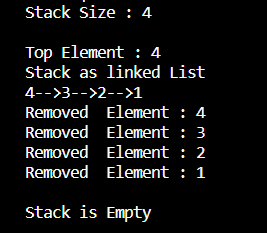
   pop();

   printStack(top);

   return;

}

OUTPUT:



Program 10: Code for implementation of single linked list with insertion , deletion

of element at the start,at given position,at the end of linked list and

traversing .

#include <stdio.h>

#include <stdlib.h>

// Create a node

struct Node {

  int data;

  struct Node\* next;

};

// Insert at the beginning

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

  // Allocate memory to a node

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

  // insert the data

  new\_node->data = new\_data;

  new\_node->next = (\*head\_ref);

  // Move head to new node

  (\*head\_ref) = new\_node;

}

// Insert a node after a node

void insertAfter(struct Node\* prev\_node, int new\_data) {

  if (prev\_node == NULL) {

  printf("the given previous node cannot be NULL");

  return;

  }

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

  new\_node->data = new\_data;

  new\_node->next = prev\_node->next;

  prev\_node->next = new\_node;

}

// Insert the the end

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

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

  struct Node\* last = \*head\_ref; /\* used in step 5\*/

  new\_node->data = new\_data;

  new\_node->next = NULL;

  if (\*head\_ref == NULL) {

  \*head\_ref = new\_node;

  return;

  }

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

  last->next = new\_node;

  return;

}

// Delete a node

void deleteNode(struct Node\*\* head\_ref, int key) {

  struct Node \*temp = \*head\_ref, \*prev;

  if (temp != NULL && temp->data == key) {

  \*head\_ref = temp->next;

  free(temp);

  return;

  }

  // Find the key to be deleted

  while (temp != NULL && temp->data != key) {

  prev = temp;

  temp = temp->next;

  }

  // If the key is not present

  if (temp == NULL) return;

  // Remove the node

  prev->next = temp->next;

  free(temp);

}

// Search a node

int searchNode(struct Node\*\* head\_ref, int key) {

  struct Node\* current = \*head\_ref;

  while (current != NULL) {

  if (current->data == key) return 1;

  current = current->next;

  }

  return 0;

}

// Print the linked list

void printList(struct Node\* node) {

  while (node != NULL) {

  printf(" %d ", node->data);

  node = node->next;

  }

}

int main() {

  struct Node\* head = NULL;

  insertAtEnd(&head, 1);

  insertAtBeginning(&head, 2);

  insertAtBeginning(&head, 3);

  insertAtEnd(&head, 4);

  insertAfter(head->next, 5);

  printf("Linked list: ");

  printList(head);

  printf("\nAfter deleting an element: ");

  deleteNode(&head, 3);

  printList(head);

  int item\_to\_find = 3;

  if (searchNode(&head, item\_to\_find)) {

  printf("\n%d is found", item\_to\_find);

  } else {

  printf("\n%d is not found", item\_to\_find);

  }

}

OUTPUT:



Program 11: Code for implementation of doubly linked list with insertion ,

deletion of element at the start,at given position,at the end of linked

list and traversing.

#include <stdio.h>

#include <stdlib.h>

// node creation

struct Node {

  int data;

  struct Node\* next;

  struct Node\* prev;

};

// insert node at the front

void insertFront(struct Node\*\* head, int data) {

  // allocate memory for newNode

  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

  // assign data to newNode

  newNode->data = data;

  // make newNode as a head

  newNode->next = (\*head);

  // assign null to prev

  newNode->prev = NULL;

  // previous of head (now head is the second node) is newNode

  if ((\*head) != NULL)

    (\*head)->prev = newNode;

  // head points to newNode

  (\*head) = newNode;

}

// insert a node after a specific node

void insertAfter(struct Node\* prev\_node, int data) {

  // check if previous node is null

  if (prev\_node == NULL) {

    printf("previous node cannot be null");

    return;

  }

  // allocate memory for newNode

  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

  // assign data to newNode

  newNode->data = data;

  // set next of newNode to next of prev node

  newNode->next = prev\_node->next;

  // set next of prev node to newNode

  prev\_node->next = newNode;

  // set prev of newNode to the previous node

  newNode->prev = prev\_node;

  // set prev of newNode's next to newNode

  if (newNode->next != NULL)

    newNode->next->prev = newNode;

}

// insert a newNode at the end of the list

void insertEnd(struct Node\*\* head, int data) {

  // allocate memory for node

  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

  // assign data to newNode

  newNode->data = data;

  // assign null to next of newNode

  newNode->next = NULL;

  // store the head node temporarily (for later use)

  struct Node\* temp = \*head;

  // if the linked list is empty, make the newNode as head node

  if (\*head == NULL) {

    newNode->prev = NULL;

    \*head = newNode;

    return;

  }

  // if the linked list is not empty, traverse to the end of the linked list

  while (temp->next != NULL)

    temp = temp->next;

  // now, the last node of the linked list is temp

  // assign next of the last node (temp) to newNode

  temp->next = newNode;

  // assign prev of newNode to temp

  newNode->prev = temp;

}

// delete a node from the doubly linked list

void deleteNode(struct Node\*\* head, struct Node\* del\_node) {

  // if head or del is null, deletion is not possible

  if (\*head == NULL || del\_node == NULL)

    return;

  // if del\_node is the head node, point the head pointer to the next of del\_node

  if (\*head == del\_node)

    \*head = del\_node->next;

  // if del\_node is not at the last node, point the prev of node next to del\_node to the previous of del\_node

  if (del\_node->next != NULL)

    del\_node->next->prev = del\_node->prev;

  // if del\_node is not the first node, point the next of the previous node to the next node of del\_node

  if (del\_node->prev != NULL)

    del\_node->prev->next = del\_node->next;

  // free the memory of del\_node

  free(del\_node);

}

// print the doubly linked list

void displayList(struct Node\* node) {

  struct Node\* last;

  while (node != NULL) {

    printf("%d->", node->data);

    last = node;

    node = node->next;

  }

  if (node == NULL)

    printf("NULL\n");

}

int main() {

  // initialize an empty node

  struct Node\* head = NULL;

  insertEnd(&head, 5);

  insertFront(&head, 1);

  insertFront(&head, 6);

  insertEnd(&head, 9);

  // insert 11 after head

  insertAfter(head, 11);

  // insert 15 after the seond node

  insertAfter(head->next, 15);

  displayList(head);

  // delete the last node

  deleteNode(&head, head->next->next->next->next->next);

  displayList(head);

}

OUTPUT:



Program 12: Code for implementation of circular linked list with insertion ,

deletion of element at the start,at given position,at the end of linked

list and traversing .

#include <stdio.h>

#include <stdlib.h>

struct Node {

  int data;

  struct Node\* next;

};

struct Node\* addToEmpty(struct Node\* last, int data) {

  if (last != NULL) return last;

  // allocate memory to the new node

  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

  // assign data to the new node

  newNode->data = data;

  // assign last to newNode

  last = newNode;

  // create link to iteself

  last->next = last;

  return last;

}

// add node to the front

struct Node\* addFront(struct Node\* last, int data) {

  // check if the list is empty

  if (last == NULL) return addToEmpty(last, data);

  // allocate memory to the new node

  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

  // add data to the node

  newNode->data = data;

  // store the address of the current first node in the newNode

  newNode->next = last->next;

  // make newNode as head

  last->next = newNode;

  return last;

}

// add node to the end

struct Node\* addEnd(struct Node\* last, int data) {

  // check if the node is empty

  if (last == NULL) return addToEmpty(last, data);

  // allocate memory to the new node

  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

  // add data to the node

  newNode->data = data;

  // store the address of the head node to next of newNode

  newNode->next = last->next;

  // point the current last node to the newNode

  last->next = newNode;

  // make newNode as the last node

  last = newNode;

  return last;

}

// insert node after a specific node

struct Node\* addAfter(struct Node\* last, int data, int item) {

  // check if the list is empty

  if (last == NULL) return NULL;

  struct Node \*newNode, \*p;

  p = last->next;

  do {

  // if the item is found, place newNode after it

  if (p->data == item) {

    // allocate memory to the new node

    newNode = (struct Node\*)malloc(sizeof(struct Node));

    // add data to the node

    newNode->data = data;

    // make the next of the current node as the next of newNode

    newNode->next = p->next;

    // put newNode to the next of p

    p->next = newNode;

    // if p is the last node, make newNode as the last node

    if (p == last) last = newNode;

    return last;

  }

  p = p->next;

  } while (p != last->next);

  printf("\nThe given node is not present in the list");

  return last;

}

// delete a node

void deleteNode(struct Node\*\* last, int key) {

  // if linked list is empty

  if (\*last == NULL) return;

  // if the list contains only a single node

  if ((\*last)->data == key && (\*last)->next == \*last) {

  free(\*last);

  \*last = NULL;

  return;

  }

  struct Node \*temp = \*last, \*d;

  // if last is to be deleted

  if ((\*last)->data == key) {

  // find the node before the last node

  while (temp->next != \*last) temp = temp->next;

  // point temp node to the next of last i.e. first node

  temp->next = (\*last)->next;

  free(\*last);

  \*last = temp->next;

  }

  // travel to the node to be deleted

  while (temp->next != \*last && temp->next->data != key) {

  temp = temp->next;

  }

  // if node to be deleted was found

  if (temp->next->data == key) {

  d = temp->next;

  temp->next = d->next;

  free(d);

  }

}

void traverse(struct Node\* last) {

  struct Node\* p;

  if (last == NULL) {

  printf("The list is empty");

  return;

  }

  p = last->next;

  do {

  printf("%d ", p->data);

  p = p->next;

  } while (p != last->next);

}

int main() {

  struct Node\* last = NULL;

  last = addToEmpty(last, 6);

  last = addEnd(last, 8);

  last = addFront(last, 2);

  last = addAfter(last, 10, 2);

  traverse(last);

  deleteNode(&last, 8);

  printf("\n");

  traverse(last);

  return 0;

}

OUTPUT:



Program 13: Code for implementation of queue in an array .

//implementation of queue in array

#include <stdio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int items[SIZE], front = -1, rear = -1;

int main()

{

    //deQueue is not possible on empty queue

    deQueue();

    //enQueue 5 elements

    enQueue(56);

    enQueue(105);

    enQueue(25);

    enQueue(4);

    enQueue(96);

    // 6th element can't be added to because the queue is full

    enQueue(6);

    display();

    //deQueue removes element entered first i.e. 56

    deQueue();

    //Now we have just 4 elements

    display();

    //deQueue removes element entered first i.e. 105

    deQueue();

    //Now we have just 3 elements

    display();

    //deQueue removes element entered first i.e. 25

    deQueue();

    //Now we have just 2 elements

    display();

    return 0;

}

void enQueue(int value)

{

  if (rear == SIZE - 1)

    printf("\nQueue is Full!!");

  else

 {

    if (front == -1)

    front = 0;

    rear++;

    items[rear] = value;

    printf("\nInserted -> %d", value);

  }

}

void deQueue()

{

  if (front == -1)

    printf("\nQueue is Empty!!");

  else

  {

    printf("\nDeleted : %d", items[front]);

    front++;

    if (front > rear)

    front = rear = -1;

  }

}

// Function to print the queue

void display()

{

  if (rear == -1)

    printf("\nQueue is Empty!!!");

  else

  {

    int i;

    printf("\nQueue elements are:\n");

    for (i = front; i <= rear; i++)

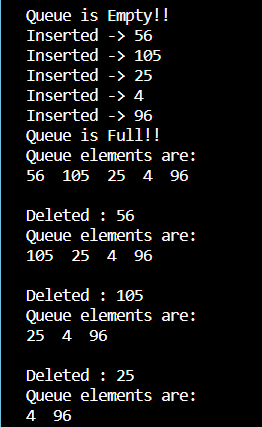
    printf("%d  ", items[i]);

  }

  printf("\n");

}

OUTPUT:



Program 14: Code for implementation of deque in an array .

#include <stdio.h>

#define MAX 10

void addFront(int \*, int, int \*, int \*);

void addRear(int \*, int, int \*, int \*);

int delFront(int \*, int \*, int \*);

int delRear(int \*, int \*, int \*);

void display(int \*);

int count(int \*);

int main() {

  int arr[MAX];

  int front, rear, i, n;

  front = rear = -1;

  for (i = 0; i < MAX; i++)

    arr[i] = 0;

  addRear(arr, 5, &front, &rear);

  addFront(arr, 12, &front, &rear);

  addRear(arr, 11, &front, &rear);

  addFront(arr, 5, &front, &rear);

  addRear(arr, 6, &front, &rear);

  addFront(arr, 8, &front, &rear);

  printf("\nElements in a deque: ");

  display(arr);

  i = delFront(arr, &front, &rear);

  printf("\nremoved item: %d", i);

  printf("\nElements in a deque after deletion: ");

  display(arr);

  addRear(arr, 16, &front, &rear);

  addRear(arr, 7, &front, &rear);

  printf("\nElements in a deque after addition: ");

  display(arr);

  i = delRear(arr, &front, &rear);

  printf("\nremoved item: %d", i);

  printf("\nElements in a deque after deletion: ");

  display(arr);

  n = count(arr);

  printf("\nTotal number of elements in deque: %d", n);

}

void addFront(int \*arr, int item, int \*pfront, int \*prear) {

  int i, k, c;

  if (\*pfront == 0 && \*prear == MAX - 1) {

    printf("\nDeque is full.\n");

    return;

  }

  if (\*pfront == -1) {

    \*pfront = \*prear = 0;

    arr[\*pfront] = item;

    return;

  }

  if (\*prear != MAX - 1) {

    c = count(arr);

    k = \*prear + 1;

    for (i = 1; i <= c; i++) {

      arr[k] = arr[k - 1];

      k--;

    }

    arr[k] = item;

    \*pfront = k;

    (\*prear)++;

  } else {

    (\*pfront)--;

    arr[\*pfront] = item;

  }

}

void addRear(int \*arr, int item, int \*pfront, int \*prear) {

  int i, k;

  if (\*pfront == 0 && \*prear == MAX - 1) {

    printf("\nDeque is full.\n");

    return;

  }

  if (\*pfront == -1) {

    \*prear = \*pfront = 0;

    arr[\*prear] = item;

    return;

  }

  if (\*prear == MAX - 1) {

    k = \*pfront - 1;

    for (i = \*pfront - 1; i < \*prear; i++) {

      k = i;

      if (k == MAX - 1)

        arr[k] = 0;

      else

        arr[k] = arr[i + 1];

    }

    (\*prear)--;

    (\*pfront)--;

  }

  (\*prear)++;

  arr[\*prear] = item;

}

int delFront(int \*arr, int \*pfront, int \*prear) {

  int item;

  if (\*pfront == -1) {

    printf("\nDeque is empty.\n");

    return 0;

  }

  item = arr[\*pfront];

  arr[\*pfront] = 0;

  if (\*pfront == \*prear)

    \*pfront = \*prear = -1;

  else

    (\*pfront)++;

  return item;

}

int delRear(int \*arr, int \*pfront, int \*prear) {

  int item;

  if (\*pfront == -1) {

    printf("\nDeque is empty.\n");

    return 0;

  }

  item = arr[\*prear];

  arr[\*prear] = 0;

  (\*prear)--;

  if (\*prear == -1)

    \*pfront = -1;

  return item;

}

void display(int \*arr) {

  int i;

  printf("\n front:  ");

  for (i = 0; i < MAX; i++)

    printf("  %d", arr[i]);

  printf("  :rear");

}

int count(int \*arr) {

  int c = 0, i;

  for (i = 0; i < MAX; i++) {

    if (arr[i] != 0)

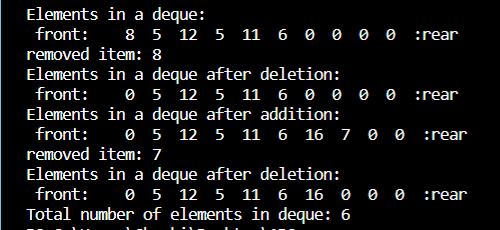
      c++;

  }

  return c;

}

OUTPUT:



Program 15: Code for implementation of circular queue in an array .

// Circular Queue implementation in C

#include <stdio.h>

#define SIZE 5

int items[SIZE];

int front = -1, rear = -1;

// Check if the queue is full

int isFull() {

  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;

  return 0;

}

// Check if the queue is empty

int isEmpty() {

  if (front == -1) return 1;

  return 0;

}

// Adding an element

void enQueue(int element) {

  if (isFull())

    printf("\n Queue is full!! \n");

  else {

    if (front == -1) front = 0;

    rear = (rear + 1) % SIZE;

    items[rear] = element;

    printf("\n Inserted -> %d", element);

  }

}

// Removing an element

int deQueue() {

  int element;

  if (isEmpty()) {

    printf("\n Queue is empty !! \n");

    return (-1);

  } else {

    element = items[front];

    if (front == rear) {

      front = -1;

      rear = -1;

    }

    // Q has only one element, so we reset the

    // queue after dequeing it. ?

    else {

      front = (front + 1) % SIZE;

    }

    printf("\n Deleted element -> %d \n", element);

    return (element);

  }

}

// Display the queue

void display() {

  int i;

  if (isEmpty())

    printf(" \n Empty Queue\n");

  else {

    printf("\n Front -> %d ", front);

    printf("\n Items -> ");

    for (i = front; i != rear; i = (i + 1) % SIZE) {

      printf("%d ", items[i]);

    }

    printf("%d ", items[i]);

    printf("\n Rear -> %d \n", rear);

  }

}

int main() {

  // Fails because front = -1

  deQueue();

  enQueue(25);

  enQueue(29);

  enQueue(35);

  enQueue(48);

  enQueue(50);

  // Fails to enqueue because front == 0 && rear == SIZE - 1

  enQueue(53);

  display();

  deQueue();

  display();

  enQueue(89);

  display();

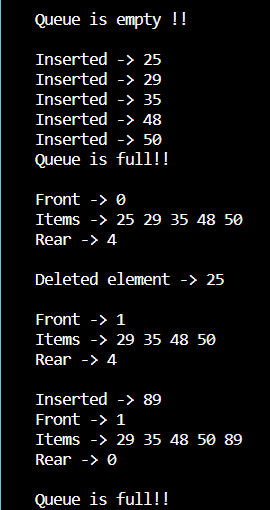
  // Fails to enqueue because front == rear + 1

  enQueue(98);

  return 0;

}

OUTPUT:



Program 16: Code for implementation of priority queue in an array .

#include <stdio.h>

int size = 0;

void swap(int \*a, int \*b) {

  int temp = \*b;

  \*b = \*a;

  \*a = temp;

}

// Function to heapify the tree

void heapify(int array[], int size, int i) {

  if (size == 1) {

    printf("Single element in the heap");

  } else {

    // Find the largest among root, left child and right child

    int largest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

    if (l < size && array[l] > array[largest])

      largest = l;

    if (r < size && array[r] > array[largest])

      largest = r;

    // Swap and continue heapifying if root is not largest

    if (largest != i) {

      swap(&array[i], &array[largest]);

      heapify(array, size, largest);

    }

  }

}

// Function to insert an element into the tree

void insert(int array[], int newNum) {

  if (size == 0) {

    array[0] = newNum;

    size += 1;

  } else {

    array[size] = newNum;

    size += 1;

    for (int i = size / 2 - 1; i >= 0; i--) {

      heapify(array, size, i);

    }

  }

}

// Function to delete an element from the tree

void deleteRoot(int array[], int num) {

  int i;

  for (i = 0; i < size; i++) {

    if (num == array[i])

      break;

  }

  swap(&array[i], &array[size - 1]);

  size -= 1;

  for (int i = size / 2 - 1; i >= 0; i--) {

    heapify(array, size, i);

  }

}

// Print the array

void printArray(int array[], int size) {

  for (int i = 0; i < size; ++i)

    printf("%d ", array[i]);

  printf("\n");

}

// Driver code

int main() {

  int array[10];

  insert(array, 3);

  insert(array, 4);

  insert(array, 9);

  insert(array, 5);

  insert(array, 2);

  printf("Max-Heap array: ");

  printArray(array, size);

  deleteRoot(array, 4);

  printf("After deleting an element: ");

  printArray(array, size);

}

OUTPUT:



Program 17: Code for implementation of binary search tree, inorder ,postorder

and preorder traversal using linked list.

#include <stdio.h>

#include <stdlib.h>

struct node

{

    int data;

    struct node\* left;

    struct node\* right;

};

struct node\* createnode(int x)

{

    struct node\* temp = malloc(sizeof(struct node));

    temp->data = x;

    temp->left = NULL;

    temp->right = NULL;

    return temp;

}

void PreOrder(struct node \*root)

{

    if(root == NULL)

    return;

    printf("%d\t",root->data);

    PreOrder(root->left);

    PreOrder(root->right);

     return;

}

void InOrder(struct node \*root)

{

    if (root==NULL)

    return;

    InOrder(root->left);

    printf("%d\t",root->data);

    InOrder(root->right);

    return;

}

void PostOrder(struct node \*root)

{

    if (root == NULL)

    return;

    PostOrder(root->left);

    PostOrder(root->right);

    printf("%d\t",root->data);

    return;

}

void main()

{

    struct node\*root = NULL;

    root = createnode(2);//root

    root->left = createnode(7);

    root->left->left= createnode(2);

    root->left->right = createnode(6);

    root->left->right->left = createnode(5);

    root->left->right->right = createnode(11);

    root->right = createnode(5);

    root->right->right = createnode(9);

    root->right->right->left = createnode(4);

    printf("Pre Order Traversal:\t");

    PreOrder(root);

    printf("\n");

    printf("In Order Traversal:\t");

    InOrder(root);

    printf("\n");

    printf("Post Order Traversal:\t");

    PostOrder(root);

    printf("\n");

    return ;

}

OUTPUT:



Program 18: Code for implementation of graph.

#include <stdio.h>

#include <stdlib.h>

// A structure to represent an adjacency list node

struct AdjListNode {

    int dest;

    struct AdjListNode\* next;

};

// A structure to represent an adjacency list

struct AdjList {

    struct AdjListNode\* head;

};

// A structure to represent a graph. A graph

// is an array of adjacency lists.

// Size of array will be V (number of vertices

// in graph)

struct Graph {

    int V;

    struct AdjList\* array;

};

// A utility function to create a new adjacency list node

struct AdjListNode\* newAdjListNode(int dest)

{

    struct AdjListNode\* newNode

        = (struct AdjListNode\*)malloc(

            sizeof(struct AdjListNode));

    newNode->dest = dest;

    newNode->next = NULL;

    return newNode;

}

// A utility function that creates a graph of V vertices

struct Graph\* createGraph(int V)

{

    struct Graph\* graph

        = (struct Graph\*)malloc(sizeof(struct Graph));

    graph->V = V;

    // Create an array of adjacency lists.  Size of

    // array will be V

    graph->array = (struct AdjList\*)malloc(

        V \* sizeof(struct AdjList));

    // Initialize each adjacency list as empty by

    // making head as NULL

    int i;

    for (i = 0; i < V; ++i)

        graph->array[i].head = NULL;

    return graph;

}

// Adds an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest)

{

    // Add an edge from src to dest.  A new node is

    // added to the adjacency list of src.  The node

    // is added at the beginning

    struct AdjListNode\* check = NULL;

    struct AdjListNode\* newNode = newAdjListNode(dest);

    if (graph->array[src].head == NULL) {

        newNode->next = graph->array[src].head;

        graph->array[src].head = newNode;

    }

    else {

        check = graph->array[src].head;

        while (check->next != NULL) {

            check = check->next;

        }

        // graph->array[src].head = newNode;

        check->next = newNode;

    }

    // Since graph is undirected, add an edge from

    // dest to src also

    newNode = newAdjListNode(src);

    if (graph->array[dest].head == NULL) {

        newNode->next = graph->array[dest].head;

        graph->array[dest].head = newNode;

    }

    else {

        check = graph->array[dest].head;

        while (check->next != NULL) {

            check = check->next;

        }

        check->next = newNode;

    }

    // newNode = newAdjListNode(src);

    // newNode->next = graph->array[dest].head;

    // graph->array[dest].head = newNode;

}

// A utility function to print the adjacency list

// representation of graph

void printGraph(struct Graph\* graph)

{

    int v;

    for (v = 0; v < graph->V; ++v) {

        struct AdjListNode\* pCrawl = graph->array[v].head;

        printf("\n Adjacency list of vertex %d\n head ", v);

        while (pCrawl) {

            printf("-> %d", pCrawl->dest);

            pCrawl = pCrawl->next;

        }

        printf("\n");

    }

}

// Driver program to test above functions

int main()

{

    // create the graph given in above fugure

    int V = 5;

    struct Graph\* graph = createGraph(V);

    addEdge(graph, 0, 1);

    addEdge(graph, 0, 4);

    addEdge(graph, 1, 2);

    addEdge(graph, 1, 3);

    addEdge(graph, 1, 4);

    addEdge(graph, 2, 3);

    addEdge(graph, 3, 4);

    // print the adjacency list representation of the above

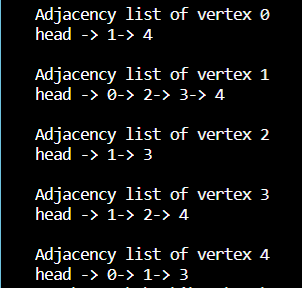
    // graph

    printGraph(graph);

    return 0;

}

OUTPUT:



Program 19: Code for implementation of BFS.

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

  int items[SIZE];

  int front;

  int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

  int vertex;

  struct node\* next;

};

struct node\* createNode(int);

struct Graph {

  int numVertices;

  struct node\*\* adjLists;

  int\* visited;

};

// BFS algorithm

void bfs(struct Graph\* graph, int startVertex) {

  struct queue\* q = createQueue();

  graph->visited[startVertex] = 1;

  enqueue(q, startVertex);

  while (!isEmpty(q)) {

    printQueue(q);

    int currentVertex = dequeue(q);

    printf("Visited %d\n", currentVertex);

    struct node\* temp = graph->adjLists[currentVertex];

    while (temp) {

      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {

        graph->visited[adjVertex] = 1;

        enqueue(q, adjVertex);

      }

      temp = temp->next;

    }

  }

}

// Creating a node

struct node\* createNode(int v) {

  struct node\* newNode = malloc(sizeof(struct node));

  newNode->vertex = v;

  newNode->next = NULL;

  return newNode;

}

// Creating a graph

struct Graph\* createGraph(int vertices) {

  struct Graph\* graph = malloc(sizeof(struct Graph));

  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices \* sizeof(struct node\*));

  graph->visited = malloc(vertices \* sizeof(int));

  int i;

  for (i = 0; i < vertices; i++) {

    graph->adjLists[i] = NULL;

    graph->visited[i] = 0;

  }

  return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

  // Add edge from src to dest

  struct node\* newNode = createNode(dest);

  newNode->next = graph->adjLists[src];

  graph->adjLists[src] = newNode;

  // Add edge from dest to src

  newNode = createNode(src);

  newNode->next = graph->adjLists[dest];

  graph->adjLists[dest] = newNode;

}

// Create a queue

struct queue\* createQueue() {

  struct queue\* q = malloc(sizeof(struct queue));

  q->front = -1;

  q->rear = -1;

  return q;

}

// Check if the queue is empty

int isEmpty(struct queue\* q) {

  if (q->rear == -1)

    return 1;

  else

    return 0;

}

// Adding elements into queue

void enqueue(struct queue\* q, int value) {

  if (q->rear == SIZE - 1)

    printf("\nQueue is Full!!");

  else {

    if (q->front == -1)

      q->front = 0;

    q->rear++;

    q->items[q->rear] = value;

  }

}

// Removing elements from queue

int dequeue(struct queue\* q) {

  int item;

  if (isEmpty(q)) {

    printf("Queue is empty");

    item = -1;

  } else {

    item = q->items[q->front];

    q->front++;

    if (q->front > q->rear) {

      printf("Resetting queue ");

      q->front = q->rear = -1;

    }

  }

  return item;

}

// Print the queue

void printQueue(struct queue\* q) {

  int i = q->front;

  if (isEmpty(q)) {

    printf("Queue is empty");

  } else {

    printf("\nQueue contains \n");

    for (i = q->front; i < q->rear + 1; i++) {

      printf("%d ", q->items[i]);

    }

  }

}

int main() {

  struct Graph\* graph = createGraph(6);

  addEdge(graph, 0, 1);

  addEdge(graph, 0, 2);

  addEdge(graph, 1, 2);

  addEdge(graph, 1, 4);

  addEdge(graph, 1, 3);

  addEdge(graph, 2, 4);

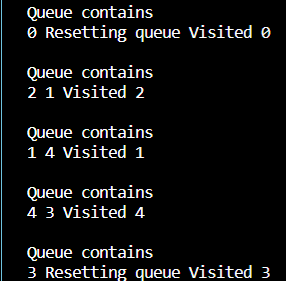
  addEdge(graph, 3, 4);

  bfs(graph, 0);

  return 0;

}

OUTPUT:



Program 20: Code for implementation of DFS.

#include <stdio.h>

#include <stdlib.h>

struct node {

  int vertex;

  struct node\* next;

};

struct node\* createNode(int v);

struct Graph {

  int numVertices;

  int\* visited;

  // We need int\*\* to store a two dimensional array.

  // Similary, we need struct node\*\* to store an array of Linked lists

  struct node\*\* adjLists;

};

// DFS algo

void DFS(struct Graph\* graph, int vertex) {

  struct node\* adjList = graph->adjLists[vertex];

  struct node\* temp = adjList;

  graph->visited[vertex] = 1;

  printf("Visited %d \n", vertex);

  while (temp != NULL) {

    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {

      DFS(graph, connectedVertex);

    }

    temp = temp->next;

  }

}

// Create a node

struct node\* createNode(int v) {

  struct node\* newNode = malloc(sizeof(struct node));

  newNode->vertex = v;

  newNode->next = NULL;

  return newNode;

}

// Create graph

struct Graph\* createGraph(int vertices) {

  struct Graph\* graph = malloc(sizeof(struct Graph));

  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices \* sizeof(struct node\*));

  graph->visited = malloc(vertices \* sizeof(int));

  int i;

  for (i = 0; i < vertices; i++) {

    graph->adjLists[i] = NULL;

    graph->visited[i] = 0;

  }

  return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

  // Add edge from src to dest

  struct node\* newNode = createNode(dest);

  newNode->next = graph->adjLists[src];

  graph->adjLists[src] = newNode;

  // Add edge from dest to src

  newNode = createNode(src);

  newNode->next = graph->adjLists[dest];

  graph->adjLists[dest] = newNode;

}

// Print the graph

void printGraph(struct Graph\* graph) {

  int v;

  for (v = 0; v < graph->numVertices; v++) {

    struct node\* temp = graph->adjLists[v];

    printf("\n Adjacency list of vertex %d\n ", v);

    while (temp) {

      printf("%d -> ", temp->vertex);

      temp = temp->next;

    }

    printf("\n");

  }

}

int main() {

  struct Graph\* graph = createGraph(4);

  addEdge(graph, 0, 1);

  addEdge(graph, 0, 2);

  addEdge(graph, 1, 2);

  addEdge(graph, 2, 3);

  printGraph(graph);

  DFS(graph, 2);

  return 0;

}

OUTPUT:

