**RAJALAKSHMI ENGINEERING COLLEGE**

**An Autonomous Institution, Affiliated to Anna University Rajalakshmi Nagar, Thandalam – 602 105**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

|  |
| --- |
| **CS23231 – DATA STRUCTURES**  **(Regulation 2023)** |
| **LAB MANUAL** |

Name: darshini.r

Rig no: 2116231801026

YEAR/BRANCH/SECTION: B. Tech Artificial intelligence and Data science

SEMESTER: II

ACADEMIC YEAR: I

**LESSON PLAN**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Course Code** | **Course Title**  **(Laboratory Integrated Theory Course)** | **L** | **T** | **P** | **C** |
| **CS23231** | **Data Structures** | **3** | **0** | **4** | **5** |

|  |  |
| --- | --- |
| **LIST OF EXPERIMENTS** | |
| **Sl. No** | **Name of the experiment** |
| Week 1 | Implementation of Single Linked List (Insertion, Deletion and Display) |
| Week 2 | Implementation of Doubly Linked List (Insertion, Deletion and Display) |
| Week 3 | Applications of Singly Linked List (Polynomial Manipulation) |
| Week 4 | Implementation of Stack using Array and Linked List implementation |
| Week 5 | Applications of Stack (Infix to Postfix) |
| Week 6 | Applications of Stack (Evaluating Arithmetic Expression) |
| Week 7 | Implementation of Queue using Array and Linked List implementation |
| Week 8 | Implementation of Binary Search Tree |
| Week 9 | Performing Tree Traversal Techniques |
| Week 10 | Implementation of AVL Tree |
| Week 11 | Performing Topological Sorting |
| Week 12 | Implementation of BFS, DFS |
| Week 13 | Implementation of Prim’s Algorithm |
| Week 14 | Implementation of Dijkstra’s Algorithm |
| Week 15 | Program to perform Sorting |
| Week 16 | Implementation of Open Addressing (Linear Probing and Quadratic Probing) |
| Week 17 | Implementation of Rehashing |

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Name of the Experiment** | **Expt. Date** |
| 1 | Implementation of Single Linked List (Insertion, Deletion and Display) | 28/02/2024 |
| 2 | Implementation of Doubly Linked List (Insertion, Deletion and Display) | 06/03/2024 |
| 3 | Applications of Singly Linked List (Polynomial Manipulation) | 13/03/2024 |
| 4 | Implementation of Stack using Array and Linked List implementation | 20/03/2024 |
| 5 | Applications of Stack (Infix to Postfix) | 27/03/2024 |
| 6 | Applications of Stack (Evaluating Arithmetic Expression) | 08/04/2024 |
| 7 | Implementation of Queue using Array and Linked List implementation | 10/04/2024 |
| 8 | Performing Tree Traversal Techniques | 17/04/2024 |
| 9 | Implementation of Binary Search Tree | 08/05/2024 |
| 10 | Implementation of AVL Tree | 08/05/2024 |
| 11 | Implementation of BFS, DFS | 15/05/2024 |
| 12 | Performing Topological Sorting | 15/05/2024 |
| 13 | Implementation of Prim’s Algorithm | 22/05/2024 |
| 14 | Implementation of Dijkstra’s Algorithm | 22/05/2024 |
| 15 | Program to perform Sorting | 29/05/2024 |
| 16 | Implementation of Collision Resolution Techniques | 29/05/2024 |

|  |  |  |
| --- | --- | --- |
| **Ex. No.:1** | **Implementation of Single Linked List** | **Date:28/02/2024** |

**Write a C program to implement the following operations on Singly Linked List.**

1. **Insert a node in the beginning of a list.**
2. **Insert a node after P**
3. **Insert a node at the end of a list**
4. **Find an element in a list**
5. **FindNext**
6. **FindPrevious**
7. **isLast**
8. **isEmpty**
9. **Delete a node in the beginning of a list.**
10. **Delete a node after P**
11. **Delete a node at the end of a list**
12. **Delete the List**

**Algorithm:**

**STEP 1: start**

**STEP 1**Define a structure to represent a node in the linked list. Each node should contain a data field to hold the value and a pointer field to point to the next node.

**STEP 1**Write a function to create a new node with a given data value. This function should allocate memory for the node, set its data field, and initialize its next pointer to NULL.

**STEP 1**Write a function to insert a new node at the beginning of the linked list. This function should take the address of the head pointer and the data value to be inserted. It should create a new node, set its next pointer to the current head, and update the head pointer to point to the new node.

**STEP 1**Write a function to insert a new node at the end of the linked list. This function should traverse the list until it reaches the last node, then create a new node with the given data value and add it after the last node.

**STEP 1**: Write a function to delete a node with a given data value from the linked list. This function should handle cases where the node to be deleted is the first node, an intermediate node, or the last node in the list.

**STEP 1**: Write a function to search for a given data value in the linked list. This function should traverse the list, comparing each node's data value with the target value until it finds a match or reaches the end of the list.

**STEP 1**Write a function to traverse the entire linked list and print its elements. This function should start from the head node and iterate through each node, printing its data value until it reaches the end of the list.

**STEP 1:stop**

Program:

#include <stdio.h>

#include<malloc.h>

void createfnode(int ele);

void insertfront(int ele);

void insertend(int ele);

void display();

//type declaration of a node

struct node

{

    int data;

    struct node\* next;

};

struct node\* head = NULL;

struct node \*newnode;

void insertfront(int ele)

 {

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

 if(newnode!=NULL)

 { newnode->data=ele;

     if(head!=NULL)

     {

        newnode->next=head;

        head=newnode;

     }

     else

     {

         newnode->next=NULL;

         head=newnode;

     }

  }

  }

   void insertend(int ele)

  {

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

    if(newnode!=NULL)

    {

        newnode->data=ele;

        newnode->next=NULL;

     if(head!=NULL)

     {

         struct node \*t;

         t=head;

         while(t->next!=NULL)

         {

             t=t->next;

         }

        newnode->next=NULL;

        t->next=newnode;

     }

     else

     {

         head=newnode;

     }

  }

  }

  int listsize()

  {

      int c=0;

      struct node \*t;

      t=head;

      while(t!=NULL)

      {

          c=c+1;

          t=t->next;

      }

      printf("\n The size of the list is %d:\n",c);

      return c;

  }

  void insertpos(int ele,int pos)

  {

     int ls=0;

     ls=listsize();

     if(head == NULL && (pos <= 0 || pos > 1))

    {

        printf("\nInvalid position to insert a node\n");

        return;

    }

    if(head != NULL && (pos <= 0 || pos > ls))

    {

        printf("\nInvalid position to insert a node\n");

        return;

    }

     struct node\* newnode = NULL;

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

    if(newnode != NULL)

    {

        newnode->data=ele;

        struct node\* temp = head;

        int count = 1;

        while(count < pos-1)

        {

            temp = temp -> next;

            count += 1;

        }

        if(pos == 1)

        {

            newnode->next = head;

            head = newnode;

        }

        else

        {

            newnode->next = temp->next;

            temp->next = newnode;

        }

    }

}

void findnext(int s)

{

    struct node \*temp;

    temp=head;

    if(temp==NULL&&temp->next==NULL)

    {

        printf("No next element ");

    }

    else

    {

        while(temp->data!=s)

        {

            temp=temp->next;

        }

                printf("\nNext Element of %d is %d\n",s,temp->next->data);

    }

}

void findprev(int s)

{

    struct node \*temp;

    temp=head;

    if(temp==NULL)

    {

        printf("List is empty ");

    }

    else

    {

        while(temp->next->data!=s)

        {

            temp=temp->next;

          }

          printf("\n The previous ele of %d is %d\n",s,temp->data);

    }

}

void find(int s)

{

    struct node \*temp;

    temp=head;

    if(head==NULL)

    {

        printf("\n List is empty");

    }

    else

    {

            while(temp->data!=s && temp->next!=NULL)

            {

                temp=temp->next;

            }

            if(temp!=NULL && temp->data==s)

            {

      printf("\n Searching ele %d is present in the addr of %p",temp->data,temp);

    }

    else

    {

        printf("\n Searching elem %d is not present",s);

    }

}

}

void isempty()

{

    if(head==NULL)

    {

        printf("\nList is empty\n");

    }

    else

    {

        printf("\nList is not empty\n");

    }

}

void deleteAtBeginning()

{

    struct node \*t;

      t=head;

      head=t->next;

}

void deleteAtEnd()

{

    struct node \*temp;

    temp=head;

    if(head==NULL)

    {

        printf("\n List is empty");

    }

    else

    {

            while(temp->next->next!=NULL)

            {

                temp=temp->next;

            }

           temp->next=NULL;

    }

}

  void display()

  {

      struct node \*t;

      t=head;

      while(t!=NULL)

      {

          printf("%d\t",t->data);

          t=t->next;

      }

  }

  void delete(int ele)

{

    struct node \*t;

    t=head;

    if(t->data==ele)

    {

        head=t->next;

    }

    else

    {

    while(t->next->data!=ele)

    {

        t=t->next;

    }

    t->next=t->next->next;

 }

}

int main()

{

    do

    {

    int ch,a,pos;

    printf("\n Choose any one operation that you would like to perform\n");

    printf("\n 1.Insert the element at the beginning");

    printf("\n 2.Insert the element at the end");

    printf("\n 3. To insert at the specified position");

    printf("\n 4. To view list");

    printf("\n 5.To view list size");

    printf("\n 6.To delete first element");

    printf("\n 7.To delete last element");

    printf("\n 8.To find next element");

    printf("\n 9. To find previous element");

    printf("\n 10. To find search for an element");

    printf("\n 11. To quit");

    printf("\n Enter your choice\n");

    scanf("%d",&ch);

        switch(ch)

        {

        case 1:

        printf("\n Insert an element to be inserted at the beginning\n");

        scanf("%d",&a);

        insertfront(a);

        break;

        case 2:

         printf("\n Insert an element to be inserted at the End\n");

        scanf("%d",&a);

        insertend(a);

        break;

        case 3:

         printf("\n Insert an element and the position to insert in the list\n");

        scanf("%d%d",&a,&pos);

        insertpos(a,pos);

        break;

        case 4:

        display();

        break;

        case 5:

        listsize();

        break;

        case 6:

        printf("\n Delete an element to be in the beginning\n");

        deleteAtBeginning();

        break;

        case 7:

        printf("\n Delete an element to be at the end\n");

        deleteAtEnd();

        break;

        case  8:

        printf("\n enter the element to which you need to find next ele in the list\n");;

        scanf("%d",&a);

        findnext(a);

        break;

        case 9:

        printf("\n enter the element to which you need to find prev ele in the list\n");;

        scanf("%d",&a);

        findprev(a);

        break;

        case 10:

        printf("\n enter the element to find the address of it\n");;

        scanf("%d",&a);

        find(a);

        break;

        case 11:

        printf("Ended");

           exit(0);

        default:

        printf("Invalid option is chosen so the process is quit");

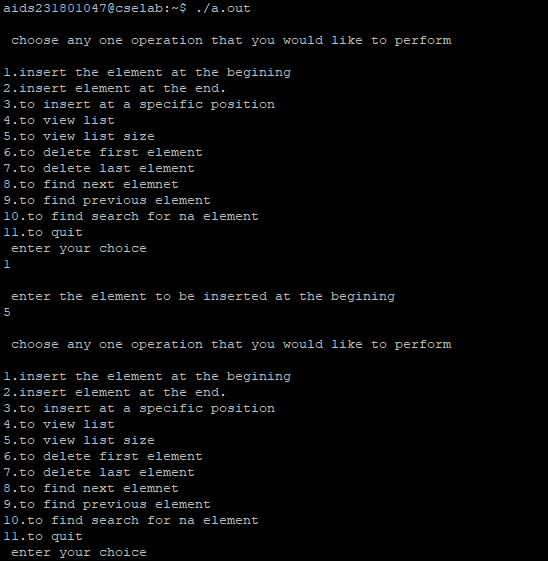
        }

    }while(1);

 return 0;

}

OUTPUT



|  |  |  |
| --- | --- | --- |
| **Ex. No.:2** | **Implementation of Doubly Linked List** | **Date:06/03/2024** |

**Write a C program to implement the following operations on Doubly Linked List.**

1. **Insertion**
2. **Deletion**
3. **Search**
4. **Display**

**Algorithm:**

**Step1:start**

**Step 2:**Specify the structure of a list node.

Step 3 :Declare a variable to maintain track of the list's head node.

Step 4:Create a new node by implementing a function.

Step5:Create a function to add a new node to the list.

Step6:Create a function to remove a node from the list.

Step7:stop

Program:

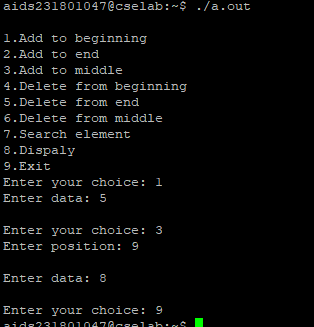
#include<stdio.h>  
#include<stdlib.h>  
struct node  
{  
        struct node \*prev;  
        int data;  
        struct node \*next;  
};  
struct node \*createnode(int data)  
{  
        struct node \*newnode=malloc(sizeof(struct node));  
        newnode->prev=NULL;  
        newnode->data=data;  
        newnode->next=NULL;  
        return newnode;  
}  
struct node \*addToBeginning(struct node \*head,int data)  
{  
        struct node \*newnode=createnode(data);  
        if(head!=NULL)  
        {  
                head->prev=head;  
        }  
        newnode->next=head;  
        return newnode;  
}  
struct node \*addToEnd(struct node \*head,int data)  
{  
        struct node \*newnode=createnode(data);  
        if(head==NULL)  
        {  
                return newnode;  
        }  
        struct node \*temp=head;  
        while(temp->next!=NULL)  
        {  
                temp=temp->next;  
        }  
        temp->next=newnode;  
        newnode->prev=temp;  
        return head;  
}  
struct node \*addToMiddle(struct node \*head,int pos,int data)  
{  
        if(head==NULL||pos<=0)  
        {  
                printf("Invalid position\n");  
                return head;  
        }  
        struct node \*newnode=createnode(data);  
        struct node \*temp=head;  
        while(pos>1 && temp->next!=NULL)  
        {  
                temp=temp->next;  
                pos--;  
        }  
        newnode->next=temp->next;  
        newnode->prev=temp;  
        if(temp->next!=NULL)  
        {  
                temp->next->prev=newnode;  
        }

 temp->next=newnode;  
        return head;  
}  
struct node \*deleteFromBeginning(struct node \*head)  
{  
        if(head==NULL)  
        {  
                printf("List is empty\n");  
                return NULL;  
        }  
        struct node \*temp=head;  
        head=head->next;  
        if(head!=NULL)  
        {  
                head->prev=NULL;  
        }  
        free(temp);  
        return head;  
}  
struct node \*deleteFromEnd(struct node \*head)  
{  
        if(head==NULL)  
        {  
                printf("List is empty\n");  
                return NULL;  
        }  
        struct node \*temp=head;  
        while(temp->next!=NULL)  
        {  
                temp=temp->next;  
        }  
        if(temp->prev!=NULL)  
        {  
                temp->prev->next=NULL;  
        }  
        free(temp);  
        return head;  
}  
struct node \*deleteFromMiddle(struct node \*head,int pos)  
{  
        if(head==NULL)  
        {  
                printf("List is empty\n");  
                return NULL;  
        }  
        struct node \*temp=head;  
        while(pos>1 && temp->next!=NULL)  
        {  
                temp=temp->next;  
                pos--;  
        }  
        if(temp==head)  
        {  
                head=deleteFromBeginning(head);  
        }  
        else if(temp->next==NULL)  
        {  
                head=deleteFromEnd(head);  
        }  
        else

 {  
                temp->prev->next=temp->next;  
                temp->next->prev=temp->prev;  
                free(temp);  
        }  
        return head;  
}  
void printList(struct node \*head)  
{  
        struct node \*temp=head;  
        while(temp!=NULL)  
        {  
                printf("%d",temp->data);  
                temp=temp->next;  
        }  
        printf("NULL\n");  
}  
struct node \*findElement(struct node \*head,int key)  
{  
        struct node \*current=head;  
        while(current!=NULL)  
        {  
                if(current!=NULL)  
                {  
                        printf("Element %d found in the list\n",key);  
                        return current;  
                }  
                current=current->next;  
        }  
        printf("Element not found");  
        return NULL;  
}  
int main()  
{  
        struct node \*head=NULL;  
        int choice,data,pos;  
        printf("\n1.Add to beginning");  
        printf("\n2.Add to end");  
        printf("\n3.Add to middle");  
        printf("\n4.Delete from beginning");  
        printf("\n5.Delete from end");  
        printf("\n6.Delete from middle");  
        printf("\n7.Search element");  
        printf("\n8.Dispaly");  
        printf("\n9.Exit");  
        while(1)  
        {  
                printf("\nEnter your choice: ");  
                scanf("%d",&choice);  
                switch(choice)  
                {  
                        case 1:  
                                {  
                                printf("Enter data: ");  
                                scanf("%d",&data);  
                                head=addToBeginning(head,data);  
                                break;  
                                }  
                        case 2:  
                                {

 {  
                                printf("Enter data: ");  
                                scanf("%d",&data);  
                                head=addToEnd(head,data);  
                                break;  
                                }  
                        case 3:  
                                {  
                                printf("Enter position: ");  
                                scanf("%d",&pos);  
                                printf("\nEnter data: ");  
                                scanf("%d",&data);  
                                head=addToMiddle(head,pos,data);  
                                break;  
                                }  
                        case 4:  
                                {  
                                head=deleteFromBeginning(head);  
                                break;  
                                }  
                        case 5:  
                                {  
                                head=deleteFromEnd(head);  
                                break;  
                                }  
                        case 6:  
                                {  
                                printf("Enter position: ");  
                                scanf("%d",&pos);  
                                head=deleteFromMiddle(head,pos);  
                                break;  
                                }  
                        case 7:  
                                {  
                                printf("Enter element: ");  
                                scanf("%d",&data);  
                                head=findElement(head,data);  
                                break;  
                                }  
                        case 8:  
                                {  
                                printf("List:");  
                                printList(head);  
                                break;  
                                }  
                        case 9:  
                                {  
                                exit(0);  
                                }  
                        default:  
                                {  
                                printf("Invalid choice\n");  
                                }  
                }  
                return 0;  
        }  
}

Output



|  |  |  |
| --- | --- | --- |
| **Ex. No.:3** | **Polynomial Manipulation** | **Date:13/03/2024** |

**Write a C program to implement the following operations on Singly Linked List.**

1. **Polynomial Addition**
2. **Polynomial Subtraction**
3. **Polynomial Multiplication**

ALGORITHM:

Step1:start

Step 2:Create a Node structure with:int coeff (coefficient)int pow (power/exponent)struct Node\* next (pointer to the next node)

step 3:Create a Polynomial structure with: Node\* head (pointer to the first node)

step 4:Define a function Node\* createNode(int coeff, int pow):Allocate memory for a new node using malloc.Set the coeff and pow fields.Initialize the next pointer to NULL.return the new node.

Step 5:Define a function void insertNode(Polynomial\* poly, int coeff, int pow):Create a new node using createNode.If the polynomial is empty or the new node's power is greater than the head's power:Set the new node's next pointer to the current head.Update the head to the new node.Otherwise, traverse the list to find the correct position:nsert the new node maintaining the order of powers.

Step 6:Define a function Polynomial addPolynomials(Polynomial\* poly1, Polynomial\* poly2):Initialize a result polynomial.Use two pointers to traverse both input polynomials.

Step7:While traversing both lists:If powers are equal, add coefficients and insert the sum into the result polynomial.If one power is greater, insert the corresponding node into the result polynomial and move the pointer.Append remaining nodes from the non-exhausted polynomial.

Step8:Define a function Polynomial multiplyPolynomials(Polynomial\* poly1, Polynomial\* poly2):Initialize a result polynomial.For each term in the first polynomial, multiply by each term in the second polynomial.Insert the product term into the result polynomial:combine like terms (terms with the same power).

Step9:Define a function void printPolynomial(Polynomial\* poly):Initialize a pointer to the head of the polynomial.Traverse the list:rint each term in the format coeff\*x^pow.Move to the next node.

Step10:stop

PROGRAM:

**Polynomial Addition**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int coeff;

int pow;

struct node \*Next;

};

 struct node \*Poly1,\*Poly2,\*Result;

void Create(struct node \*List);

void Display(struct node \*List);

void Addition(struct node \*Poly1,struct node \*Poly2,struct node \*Result);

int main()

{

Poly1=(struct node\*)malloc(sizeof(struct node));

Poly2=(struct node\*)malloc(sizeof(struct node));

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

Poly1->Next = NULL;

Poly2->Next = NULL;

printf("Enter the values for first polynomial :\n");

Create(Poly1);

printf("The polynomial equation is : ");

Display(Poly1);

printf("\nEnter the values for second polynomial :\n");

Create(Poly2);

printf("The polynomial equation is : ");

Display(Poly2);

Addition(Poly1, Poly2, Result);

printf("\nThe polynomial equation addition result is : ");

Display(Result);

return 0;

}

void Create(struct node \*List)

{

int choice;

struct node \*Position, \*NewNode;

Position = List;

do

{

NewNode = malloc(sizeof(struct node));

printf("Enter the coefficient : ");

scanf("%d", &NewNode->coeff);

printf("Enter the power : ");

scanf("%d", &NewNode->pow);

NewNode->Next = NULL;

Position->Next = NewNode;

Position = NewNode;

printf("Enter 1 to continue : ");

scanf("%d", &choice);

} while(choice == 1);

}

void Display(struct node \*List)

{

struct node \*Position;

Position = List->Next;

while(Position != NULL)

{

printf("%dx^%d", Position->coeff, Position->pow);

Position = Position->Next;

if(Position != NULL && Position->coeff > 0)

{

printf("+");

}

}

}

void Addition(struct node \*Poly1, struct node \*Poly2, struct node \*Result)

{

struct node \*Position;

struct node \*NewNode;

Poly1 = Poly1->Next;

Poly2 = Poly2->Next;

Result->Next = NULL;

Position = Result;

while(Poly1 != NULL && Poly2 != NULL)

{

NewNode = malloc(sizeof(struct node));

if(Poly1->pow == Poly2->pow)

{

NewNode->coeff = Poly1->coeff + Poly2->coeff;

NewNode->pow = Poly1->pow;

Poly1 = Poly1->Next;

Poly2 = Poly2->Next;

}

else if(Poly1->pow > Poly2->pow)

{

NewNode->coeff = Poly1->coeff;

NewNode->pow = Poly1->pow;

Poly1 = Poly1->Next;

}

else if(Poly1->pow < Poly2->pow)

{

NewNode->coeff = Poly2->coeff;

NewNode->pow = Poly2->pow;

Poly2 = Poly2->Next;

}

NewNode->Next = NULL;

Position->Next = NewNode;

Position = NewNode;

}

while(Poly1 != NULL || Poly2 != NULL)

{

NewNode = malloc(sizeof(struct node));

if(Poly1 != NULL)

{

NewNode->coeff = Poly1->coeff;

NewNode->pow = Poly1->pow;

Poly1 = Poly1->Next;

}

if(Poly2 != NULL)

{

NewNode->coeff = Poly2->coeff;

NewNode->pow = Poly2->pow;

Poly2 = Poly2->Next;

}

NewNode->Next = NULL;

Position->Next = NewNode;

Position = NewNode;

}

}

**Program 2:**

#include<stdio.h>

#include<stdlib.h>

struct node

{

    int coeff;

    int expo;

    struct node \*next;

};

struct node\* insert(struct node \*head,int co,int exp)

{

    struct node \*temp;

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

    newnode->coeff=co;

    newnode->expo=exp;

    newnode->next=NULL;

    if(head==NULL || exp>head->expo)

    {

        newnode->next=head;

        head=newnode;

    }

    else

    {

        temp=head;

        while(temp->next!=NULL &&temp->next->expo>=exp)

            temp=temp->next;

        newnode->next=temp->next;

        temp->next=newnode;

    }

    return head;

}

struct node\* create(struct node \*head)

{

    int n,i;

    int coeff;

    int expo;

    printf("Enter the no of terms:");

    scanf("%d",&n);

    for(i=0;i<n;i++)

    {

        printf("Enter the coeefficient for term %d:",i+1);

        scanf("%d",&coeff);

        printf("Enter the exponent for term %d:",i+1);

        scanf("%d",&expo);

        head=insert(head,coeff,expo);

    }

    return head;

}

    void print(struct node\* head)

    {

        if(head==NULL)

            printf("No Polynomial");

        else

        {

            struct node \*temp=head;

            while(temp!=NULL)

            {

                printf("%dx^%d",temp->coeff,temp->expo);

                temp=temp->next;

                if(temp!=NULL)

                    printf("+");

                else

                    printf("\n");

            }

        }

    }

    void polyAdd(struct node \*head1, struct node \*head2)

    {

        struct node \*ptr1=head1;

        struct node \*ptr2=head2;

        struct node \*head3=NULL;

        while(ptr1!=NULL && ptr2!=NULL)

        {

            if(ptr1->expo == ptr2->expo)

            {

            head3=insert(head3,ptr1->coeff+ptr2->coeff,ptr1->expo);

                ptr1=ptr1->next;

                ptr2=ptr2->next;

            }

            else if(ptr1->expo > ptr2->expo)

            {

                head3=insert(head3,ptr1->coeff,ptr1->expo);

                ptr1=ptr1->next;

            }

            else if(ptr1->expo < ptr2->expo)

            {

                head3=insert(head3,ptr2->coeff,ptr2->expo);

                ptr2=ptr2->next;

            }

        }

        while(ptr1!=NULL)

        {

            head3=insert(head3,ptr1->coeff,ptr1->expo);

            ptr1=ptr1->next;

        }

        while(ptr2!=NULL)

        {

            head3=insert(head3,ptr2->coeff,ptr2->expo);

            ptr2=ptr2->next;

        }

       printf("Added Polynomial is: ") ;

       print(head3);

    }

    int main()

    {

        struct node \*head1=NULL;

        struct node \*head2=NULL;

        printf("Enter first polynomial\n");

        head1=create(head1);

        printf("Enter second polynomial\n");

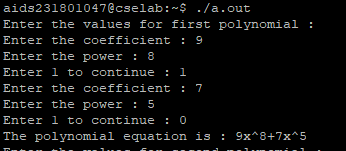
        head2=create(head2);

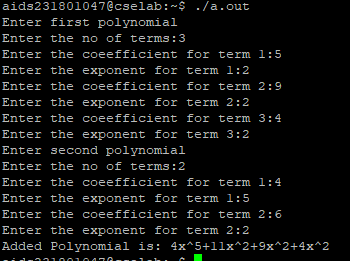
        polyAdd(head1,head2);

        return 0;

    }

OUTPUT





|  |  |  |
| --- | --- | --- |
| **Ex. No.:4** | **Implementation of Stack using Array and Linked List Implementation** | **Date:20/03/2024** |

**Write a C program to implement a stack using Array and linked List implementation and execute the following operation on stack.**

1. **Push an element into a stack**
2. **Pop an element from a stack**
3. **Return the Top most element from  a stack**
4. **Display the elements in a stack**

**Algorithm:**

Step 1: Start the program.

Step 2: For Push operation, check for stack overflow

Step 3: If Top>=N then print stack overflow else increment Top and insert the

            element.

Step 4: For Pop operation, check for underflow of the stack.

Step 5: If Top=0 then print stack underflow else decrement Top and delete the

           element

Step 6: Stop the program.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int Data;

struct Node \*next;

}\*top;

void popStack()

{

struct Node \*temp, \*var=top;

if(var==top)

{

top = top->next;

free(var);

}

else

printf("\nStack Empty");

}

void push(int value)

{

struct Node \*temp;

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

temp->Data=value;

if (top == NULL)

{

top=temp;

top->next=NULL;

}

else

{

temp->next=top;

top=temp;

}

}

void display()

{

struct Node \*var=top;

if(var!=NULL)

{

printf("\nElements are as:\n");

while(var!=NULL)

{

printf("\t%d\n",var->Data);

var=var->next;

}

printf("\n");

}

else

printf("\nStack is Empty");

}

int main()

{

int i=0;

top=NULL;

clrscr();

printf(" \n1. Push to stack");

printf(" \n2. Pop from Stack");

printf(" \n3. Display data of Stack");

printf(" \n4. Exit\n");

while(1)

{

printf(" \nChoose Option: ");

scanf("%d",&i);

switch(i)

{

case 1:

{

int value;

printf("\nEnter a value to push into Stack: ");

scanf("%d",&value);

push(value);

break;

}

case 2:

{

popStack();

printf("\n The last element is popped");

break;

}

case 3:

{

display();

break;

}

case 4:

{

struct Node \*temp;

while(top!=NULL)

{

temp = top->next;

free(top);

top=temp;

}

exit(0);

}

default:

{

printf("\nwrong choice for operation");

}}}}

Implementation of Stack using Array

#include<stdio.h>

int stack[100],choice,n,top,x,i;

void push(void);

void pop(void);

void display(void);

int main()

{

    top=-1;

    printf("\n Enter the size of STACK[MAX=100]:");

    scanf("%d",&n);

    printf("\n\t STACK OPERATIONS USING ARRAY");

    printf("\n\t--------------------------------");

    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

    do

    {

        printf("\n Enter the Choice:");

        scanf("%d",&choice);

        switch(choice)

        {

            case 1:

            {

                push();

                break;

            }

            case 2:

            {

                pop();

                break;

            }

            case 3:

            {

                display();

                break;

            }

            case 4:

            {

                printf("\n\t EXIT POINT ");

                break;

            }

            default:

            {

                printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");

            }

        }

    }

    while(choice!=4);

    return 0;

}

void push()

{

    if(top>=n-1)

    {

        printf("\n\tSTACK is over flow");

    }

    else

    {

        printf(" Enter a value to be pushed:");

        scanf("%d",&x);

        top++;

        stack[top]=x;

    }

}

void pop()

{

    if(top<=-1)

    {

        printf("\n\t Stack is under flow");

    }

    else

    {

        printf("\n\t The popped elements is %d",stack[top]);

        top--;

    }

}

void display()

{

    if(top>=0)

    {

        printf("\n The elements in STACK \n");

        for(i=top; i>=0; i--)

            printf("\n%d",stack[i]);

        printf("\n Press Next Choice");

    }

    else

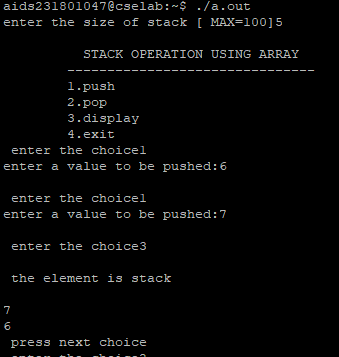
    {

        printf("\n The STACK is empty");

    }

}

OUTPUT



|  |  |  |
| --- | --- | --- |
| Ex. No.:5 | **Infix to Postfix Conversion** | Date:27/03/2024 |

**Write a C program to perform infix to postfix conversion using stack.**

**Algorithm:**

Step 1: Start the program.

Step 2: Get the infix expression as input.

Step 3: Read the input from left to right.

 Step 4: If the input is operand then place it in the postfix expression.

Step 5: Else if the input symbol is an operator then check for the operator type and

also the precedence, pop entries from the stack and place them in the

postfix expression until the lowest priority operator is encountered.

Step 6: ‘(‘symbol will be popped from stack only when we get a ‘)’ symbol.

Step 7: When the input is completely read then pop the elements in stack until it

becomes empty.

 Step 8: Display the postfix expression.

Step 9: Stop the program.

PROGRAM:

#include<stdio.h>

#include<conio.h>

#include<alloc.h>

int top=0,st[20];

char inf[40],post[40];

void postfix();

void push(int);

char pop();

void main()

{

clrscr();

printf("Enter the infix expression:");

scanf("%s",inf);

postfix();

getch();

}

void postfix()

{int i,j=0;

for(i=0;inf[i]!=0;i++)

{switch(inf[i])

{

case '+':while(st[top]>=1)

post[j++]=pop();

push(1);

break;

case '-':while(st[top]>=1)

post[j++]=pop();

push(2);

break;

case '\*':while(st[top]>=3)

post[j++]=pop();

push(3);

break;

case '/':while(st[top]>=4)

post[j++]=pop();

push(4);

break;

case '^':

post[j++]=pop();

push(5);

break;

case '(':push(0);

break;

case ')':while(st[top]!=0)

post[j++]=pop();

top--;

break;

default:

post[j++]=inf[i];

}}

while(top>0)

post[j++]=pop();

printf("\nPostfix expression is =>\n\t\t%s",post);

}void push(int ele)

{

top++;

st[top]=ele;

}char pop()

{int el;

char e;

el=st[top];

top--;

switch(el)

{case 1:

e='+';

break;

case 2:

e='-';

break;

case 3:

e='\*';

break;

case 4:

e='/';

break;

case 5:

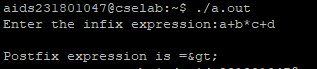
e='^';

break;

}return(e);

}

OUTPUT



|  |  |  |
| --- | --- | --- |
| Ex. No.:6 | **Evaluating Arithmetic Expression** | Date:03/04/2024 |

**Write a C program to evaluate Arithmetic expression using stack.**

**Algorithm:**

Step 1: Start the program.

Step 2: Read the postfix expression from left to right

Step 3: If the symbol read is an operand then push it onto the stack

Step 4: If the operator is read POP two operands and perform arithmetic

operations if operator is

+ then result=operand 1 + operand 2

- then result=operand 1 - operand 2

\* then result=operand 1 \* operand 2

/ then result=operand 1 / operand 2

Step 5: Push the result onto the stack

Step 6: Repeat steps 2-5 till the postfix expression is not over

Step 7: Stop the program.

PROGRAM:

#include <stdio.h>

#include <string.h>

int top = -1;

int stack[100];

void push (int data) {

stack[++top] = data;

}

int pop () {

int data;

if (top == -1)

return -1;

data = stack[top];

stack[top] = 0;

top--;

return (data);

}

int main()

 {

char str[100];

int i, data = -1, operand1, operand2, result;

printf("Enter ur postfix expression:");

fgets(str, 100, stdin);

for (i = 0; i < strlen(str); i++)

 {

if (isdigit(str[i]))

{

data = (data == -1) ? 0 : data;

data = (data \* 10) + (str[i] - 48);

continue;

}

if (data != -1)

 {

push(data);

}

if (str[i] == '+' || str[i] == '-'|| str[i] == '\*' || str[i] == '/')

{

operand2 = pop();

operand1 = pop();

if (operand1 == -1 || operand2 == -1)

break;

switch (str[i])

{

case '+':

result = operand1 + operand2;

push(result);

break;

case '-':

result = operand1 - operand2;

push(result);

break;

case '\*':

result = operand1 \* operand2;

push(result);

break;

case '/':

result = operand1 / operand2;

push(result);

break;

}

}

data = -1;

}

if (top == 0)

printf("The answer is:%d\n", stack[top]);

else

printf("u have given wrong postfix expression\n");

return 0;

}

OUTPUT



|  |  |  |
| --- | --- | --- |
| **Ex. No.:7** | **Implementation of Queue using Array and Linked List Implementation** | **Date:10/04/2024** |

**Write a C program to implement a Queue using Array and linked List implementation and execute the following operation on stack.**

1. **Enqueue**
2. **Dequeue**
3. **Display the elements in a Queue**

ALGORITHM:

Step 1: Start the program.

Step 2: For queue insertion operation, check for queue overflow

Step 3: If R>=N then print queue overflow else increment rear pointer and insert

            the element.

Step 4: For queue deletion operation, check for underflow of the queue.

Step 5: If F=0 then print queue underflow else delete the element and increment

            the front pointer

Step 6: Stop the program

PROGRAM:

#include<stdio.h >

#include<conio.h >

#include<alloc.h >

struct queue

{

int data;

struct queue \*next;

};

struct queue \*addq(struct queue \*front);

struct queue \*delq(struct queue \*front);

void main()

{

struct queue \*front;

int reply,option,data;

clrscr();

front=NULL;

do

{

printf("\n1.addq");

printf("\n2.delq");

printf("\n3.exit");

printf("\nSelect the option");

scanf("%d",&option);

switch(option)

{

case 1 : //addq

front=addq(front);

printf("\n The element is added into the queue");

break;

case 2 : //delq

front=delq(front);

break;

case 3 : exit(0);

}

}while(1);

}

struct queue \*addq(struct queue \*front)

{

struct queue \*c,\*r;

//create new node

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

if(c==NULL)

{

printf("Insufficient memory");

return(front);

}

//read an insert value from console

printf("\nEnter data");

scanf("%d",&c->data);

c->next=NULL;

if(front==NULL)

{

front=c;

}

else

{

//insert new node after last node

r=front;

while(r->next!=NULL)

{

r=r->next;

}}

return(front);

}

struct queue \*delq(struct queue \*front)

{

struct queue \*c;

if(front==NULL)

{

printf("Queue is empty");

return(front);

}

//print the content of first node

printf("Deleted data:%d",front->data);

//delete first node

c=front;

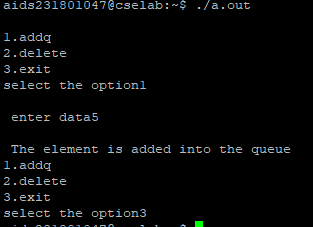
front=front->next;

free(c);

return(front);

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| **Ex. No.:8** | **Tree Traversal** | **Date:17/04/2024** |

**Write a C program to implement a Binary tree and perform the following tree traversal operation.**

1. **Inorder Traversal**
2. **Preorder Traversal**
3. **Postorder Traversal**

**Algorithm:**

**Step 1:start**

Step2: Return the root node value. Traverse the left subtree by recursively calling the pre-order function. Traverse the right subtree by recursively calling the pre-order function.

Step 3: Traverse the left subtree by recursively calling the in-order function. Return the root node value. Traverse the right subtree by recursively calling the in-order function.

Step4: Traverse the left subtree by recursively calling the post-order function. Traverse the right subtree by recursively calling the post-order function. Return the root node value.

Step 5: stop

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

struct node {

    int element;

    struct node\* left;

    struct node\* right;

};

struct node\* createNode(int val)

{

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

    Node->element = val;

    Node->left = NULL;

    Node->right = NULL;

    return (Node);

}

void traversePreorder(struct node\* root)

{

    if (root == NULL)

        return;

    printf(" %d ", root->element);

    traversePreorder(root->left);

    traversePreorder(root->right);

}

void traverseInorder(struct node\* root)

{

    if (root == NULL)

        return;

    traverseInorder(root->left);

    printf(" %d ", root->element);

    traverseInorder(root->right);

}

 void traversePostorder(struct node\* root)

{

    if (root == NULL)

        return;

    traversePostorder(root->left);

    traversePostorder(root->right);

    printf(" %d ", root->element);

}

int main()

{

    struct node\* root = createNode(36);

    root->left = createNode(26);

    root->right = createNode(46);

    root->left->left = createNode(21);

    root->left->right = createNode(31);

    root->left->left->left = createNode(11);

    root->left->left->right = createNode(24);

    root->right->left = createNode(41);

    root->right->right = createNode(56);

    root->right->right->left = createNode(51);

    root->right->right->right = createNode(66);

    printf("\n The Preorder traversal of given binary tree is -\n");

    traversePreorder(root);

    printf("\n The Inorder traversal of given binary tree is -\n");

    traverseInorder(root);

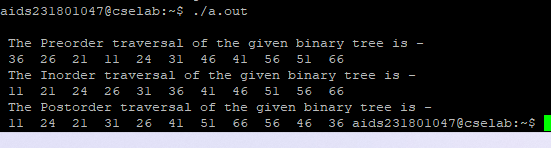
    printf("\n The Postorder traversal of given binary tree is -\n");

    traversePostorder(root);

    return 0;

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| **Ex. No.:9** | **Implementation of Binary Search tree** | **Date:08/05/2024** |

**Write a C program to implement a Binary Search Tree and perform the following operations.**

1. **Insert**
2. **Delete**
3. **Search**
4. **Display**

ALGORITHM:

Step1:start

Step 2:Create a Node structure with:int data ,Node\* left ,Node\* right

Step 3:Define a BST structure with: Node\* root

Step 4:Define a function Node\* createNode(int data):,Allocate memory for a new node using malloc.,Set the data field,Initialize left and right pointers to NULL.,Return the new node.

Step 5:Define a function Node\* insertNode(Node\* root, int data):If root is NULL, create and return a new node,If data is less than root->data, recursively insert in the left subtree,If data is greater than root->data, recursively insert in the right subtree.,Return the root

Step 6:Define a function Node\* searchNode(Node\* root, int data):If root is NULL or root->data is data, return root.,If data is less than root->data, recursively search in the left subtree.,If data is greater than root->data, recursively search in the right subtree.,Return the result.

Step 7:Define a function Node\* findMin(Node\* root):Traverse the left subtree until left is NULL.,Return the node.

Step 8:Define a function Node\* deleteNode(Node\* root, int data):If root is NULL, return root.,If data is less than root->data, recursively delete in the left subtree.,If data is greater than root->data, recursively delete in the right subtree.,If data is equal to root->data:,If the node has no children, return NULL.,If the node has one child, return the non-NULL child.,If the node has two children:,Find the minimum node in the right subtree.,Replace root->data with the minimum node's data.,Recursively delete the minimum node in the right subtree.,Return the root.

Step9:stop

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

struct BinaryTreeNode {

    int key;

    struct BinaryTreeNode \*left, \*right;

};

struct BinaryTreeNode\* newNodeCreate(int value)

{

    struct BinaryTreeNode\* temp

        = (struct BinaryTreeNode\*)malloc(

            sizeof(struct BinaryTreeNode));

    temp->key = value;

    temp->left = temp->right = NULL;

    return temp;

}

struct BinaryTreeNode\*

searchNode(struct BinaryTreeNode\* root, int target)

{

    if (root == NULL || root->key == target) {

        return root;

    }

    if (root->key < target) {

        return searchNode(root->right, target);

    }

    return searchNode(root->left, target);

}

struct BinaryTreeNode\*

insertNode(struct BinaryTreeNode\* node, int value)

{

    if (node == NULL) {

        return newNodeCreate(value);

    }

    if (value < node->key) {

        node->left = insertNode(node->left, value);

    }

    else if (value > node->key) {

        node->right = insertNode(node->right, value);

    }

    return node;

}

void postOrder(struct BinaryTreeNode\* root)

{

    if (root != NULL) {

        postOrder(root->left);

        postOrder(root->right);

        printf(" %d ", root->key);

    }

}

void inOrder(struct BinaryTreeNode\* root)

{

    if (root != NULL) {

        inOrder(root->left);

        printf(" %d ", root->key);

        inOrder(root->right);

    }

}

void preOrder(struct BinaryTreeNode\* root)

{

    if (root != NULL) {

        printf(" %d ", root->key);

        preOrder(root->left);

        preOrder(root->right);

    }

}

struct BinaryTreeNode\* findMin(struct BinaryTreeNode\* root)

{

    if (root == NULL) {

        return NULL;

    }

    else if (root->left != NULL) {

        return findMin(root->left);

    }

    return root;

}

struct BinaryTreeNode\* delete (struct BinaryTreeNode\* root,

                               int x)

{

    if (root == NULL)

        return NULL;

    if (x > root->key) {

        root->right = delete (root->right, x);

    }

    else if (x < root->key) {

        root->left = delete (root->left, x);

    }

    else {

        if (root->left == NULL && root->right == NULL) {

            free(root);

            return NULL;

        }

        else if (root->left == NULL

                 || root->right == NULL) {

            struct BinaryTreeNode\* temp;

            if (root->left == NULL) {

                temp = root->right;

            }

            else {

                temp = root->left;

            }

            free(root);

            return temp;

        }

        else {

            struct BinaryTreeNode\* temp

                = findMin(root->right);

            root->key = temp->key;

            root->right = delete (root->right, temp->key);

        }

    }

    return root;

}

int main()

{

    struct BinaryTreeNode\* root = NULL;

    root = insertNode(root, 50);

    insertNode(root, 30);

    insertNode(root, 20);

    insertNode(root, 40);

    insertNode(root, 70);

    insertNode(root, 60);

    insertNode(root, 80);

    if (searchNode(root, 60) != NULL) {

        printf("60 found");

    }

    else {

        printf("60 not found");

    }

    printf("\n");

    postOrder(root);

    printf("\n");

    preOrder(root);

    printf("\n");

    inOrder(root);

    printf("\n");

    struct BinaryTreeNode\* temp = delete (root, 70);

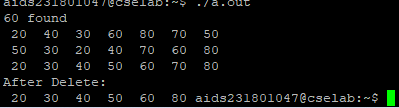
    printf("After Delete: \n");

    inOrder(root);

    return 0;

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| **Ex. No.:10** | **Implementation of AVL Tree** | **Date:08/05/2024** |

**Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.**

**Algorithm:**

**Step 1 :Define node structure (data, left, right, balance\_factor).**

**Step 2 :Create recursive insert function (root, data).**

**Step 3:Perform standard BST insertion within insert.**

**Step 4 :Update balance factors on way back up in insert.**

**Step 5 :Check for imbalance (balance factor outside -1 to 1).**

**Step 6 :Select rotation type based on imbalance.**

**Step 7 :Perform rotation to restore balance.**

**Step 8 :Return updated root after rotation.**

**Step 9 :Return new root from top-level insert call.**

**Step 10 :Create wrapper function avl\_insert.**

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

**struct** node

{

**int** data;

**struct** node\* left;

**struct** node\* right;

**int** ht;

};

**struct** node\* root = NULL;

**struct** node\* create(**int**);

**struct** node\* insert(**struct** node\*, **int**);

**struct** node\* **delete**(**struct** node\*, **int**);

**struct** node\* search(**struct** node\*, **int**);

**struct** node\* rotate\_left(**struct** node\*);

**struct** node\* rotate\_right(**struct** node\*);

**int** balance\_factor(**struct** node\*);

**int** height(**struct** node\*);

**void** inorder(**struct** node\*);

**void** preorder(**struct** node\*);

**void** postorder(**struct** node\*);

**int** main()

{

**int** user\_choice, data;

**char** user\_continue = 'y';

**struct** node\* result = NULL;

**while** (user\_continue == 'y' || user\_continue == 'Y')

    {

        printf("\n\n------- AVL TREE --------\n");

        printf("\n1. Insert");

        printf("\n2. Delete");

        printf("\n3. Search");

        printf("\n4. Inorder");

        printf("\n5. Preorder");

        printf("\n6. Postorder");

        printf("\n7. EXIT");

        printf("\n\nEnter Your Choice: ");

        scanf("%d", &user\_choice);

**switch**(user\_choice)

        {

**case** 1:

                printf("\nEnter data: ");

                scanf("%d", &data);

                root = insert(root, data);

**break**;

**case** 2:

                printf("\nEnter data: ");

                scanf("%d", &data);

                root = **delete**(root, data);

**break**;

**case** 3:

                printf("\nEnter data: ");

                scanf("%d", &data);

                result = search(root, data);

**if** (result == NULL)

                {

                    printf("\nNode not found!");

                }

**else**

                {

                    printf("\n Node found");

                }

**break**;

**case** 4:

                inorder(root);

**break**;

**case** 5:

                preorder(root);

**break**;

**case** 6:

                postorder(root);

**break**;

**case** 7:

                printf("\n\tProgram Terminated\n");

**return** 1;

**default**:

                printf("\n\tInvalid Choice\n");

        }

        printf("\n\nDo you want to continue? ");

        scanf(" %c", &user\_continue);

    }

**return** 0;

}

**struct** node\* create(**int** data)

{

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

**if** (new\_node == NULL)

    {

        printf("\nMemory can't be allocated\n");

**return** NULL;

    }

    new\_node->data = data;

    new\_node->left = NULL;

    new\_node->right = NULL;

**return** new\_node;

}

**struct** node\* rotate\_left(**struct** node\* root)

{

**struct** node\* right\_child = root->right;

    root->right = right\_child->left;

    right\_child->left = root;

    root->ht = height(root);

    right\_child->ht = height(right\_child);

**return** right\_child;

}

**struct** node\* rotate\_right(**struct** node\* root)

{

**struct** node\* left\_child = root->left;

    root->left = left\_child->right;

    left\_child->right = root;

    root->ht = height(root);

    left\_child->ht = height(left\_child);

**return** left\_child;

}

**int** balance\_factor(**struct** node\* root)

{

**int** lh, rh;

**if** (root == NULL)

**return** 0;

**if** (root->left == NULL)

        lh = 0;

**else**

        lh = 1 + root->left->ht;

**if** (root->right == NULL)

        rh = 0;

**else**

        rh = 1 + root->right->ht;

**return** lh - rh;

}

**int** height(**struct** node\* root)

{

**int** lh, rh;

**if** (root == NULL)

    {

**return** 0;

    }

**if** (root->left == NULL)

        lh = 0;

**else**

        lh = 1 + root->left->ht;

**if** (root->right == NULL)

        rh = 0;

**else**

        rh = 1 + root->right->ht;

**if** (lh > rh)

**return** (lh);

**return** (rh);

}

**struct** node\* insert(**struct** node\* root, **int** data)

{

**if** (root == NULL)

    {

**struct** node\* new\_node = create(data);

**if** (new\_node == NULL)

        {

**return** NULL;

        }

        root = new\_node;

    }

**else** **if** (data > root->data)

    {

        root->right = insert(root->right, data);

**if** (balance\_factor(root) == -2)

        {

**if** (data > root->right->data)

            {

                root = rotate\_left(root);

            }

**else**

            {

                root->right = rotate\_right(root->right);

                root = rotate\_left(root);

            }

        }

    }

**else**

    {

        root->left = insert(root->left, data);

**if** (balance\_factor(root) == 2)

        {

**if** (data < root->left->data)

            {

                root = rotate\_right(root);

            }

**else**

            {

                root->left = rotate\_left(root->left);

                root = rotate\_right(root);

            }

        }

    }

    root->ht = height(root);

**return** root;

}

**struct** node \* **delete**(**struct** node \*root, **int** x)

{

**struct** node \* temp = NULL;

**if** (root == NULL)

    {

**return** NULL;

    }

**if** (x > root->data)

    {

        root->right = **delete**(root->right, x);

**if** (balance\_factor(root) == 2)

        {

**if** (balance\_factor(root->left) >= 0)

            {

                root = rotate\_right(root);

            }

**else**

            {

                root->left = rotate\_left(root->left);

                root = rotate\_right(root);

            }

        }

    }

**else** **if** (x < root->data)

    {

        root->left = **delete**(root->left, x);

**if** (balance\_factor(root) == -2)

        {

**if** (balance\_factor(root->right) <= 0)

            {

                root = rotate\_left(root);

            }

**else**

            {

                root->right = rotate\_right(root->right);

                root = rotate\_left(root);

            }

        }

    }

**else**

    {

**if** (root->right != NULL)

        {

            temp = root->right;

**while** (temp->left != NULL)

                temp = temp->left;

            root->data = temp->data;

            root->right = **delete**(root->right, temp->data);

**if** (balance\_factor(root) == 2)

            {

**if** (balance\_factor(root->left) >= 0)

                {

                    root = rotate\_right(root);

                }

**else**

                {

                    root->left = rotate\_left(root->left);

                    root = rotate\_right(root);

                }

            }

        }

**else**

        {

**return** (root->left);

        }

    }

    root->ht = height(root);

**return** (root);

}

**struct** node\* search(**struct** node\* root, **int** key)

{

**if** (root == NULL)

    {

**return** NULL;

    }

**if**(root->data == key)

    {

**return** root;

    }

**if**(key > root->data)

    {

        search(root->right, key);

    }

**else**

    {

        search(root->left, key);

    }

}

**void** inorder(**struct** node\* root)

{

**if** (root == NULL)

    {

**return**;

    }

    inorder(root->left);

    printf("%d ", root->data);

    inorder(root->right);

}

/ **void** preorder(**struct** node\* root)

{

**if** (root == NULL)

    {

**return**;

    }

    printf("%d ", root->data);

    preorder(root->left);

    preorder(root->right);

}

**void** postorder(**struct** node\* root)

{

**if** (root == NULL)

    {

**return**;

    }

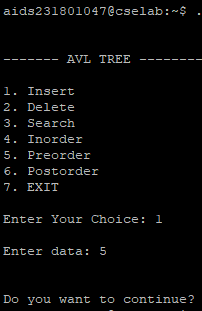
    postorder(root->left);

    postorder(root->right);

    printf("%d ", root->data);

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| Ex. No.:11 | **Graph Traversal** | **Date:15/05/2024** |

**Write a C program to create a graph and perform a Breadth First Search and Depth First Search.**

**Algorithm:**

**DFS:**

Step 1: Choose any node in the graph. Designate it as the search node and mark it as visited.

Step 2: Using the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.

Step 3: Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.

Step 4: When a return to the previous search node in (3) is impossible, the search from the originally chosen search node is complete.

Step 5: If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

BFS:

Step 1: Choose any node in the graph, designate it as the search node and mark it as visited.

Step 2: Using the adjacency matrix of the graph, find all the unvisited adjacent nodes to the search node and enqueue them into the queue Q.

Step 3: Then the node is dequeued from the queue. Mark that node as visited and designate it as the new search node.

Step 4: Repeat step 2 and 3 using the new search node.

Step 5: This process continues until the queue Q which keeps track of the adjacent nodes is empty.

PROGRAM:

**DFS Program**

#include <stdio.h>

#include <stdlib.h>

int vis[100];

struct Graph {

    int V;

    int E;

    int\*\* Adj;

};

struct Graph\* adjMatrix()

{

    struct Graph\* G = (struct Graph\*)

        malloc(sizeof(struct Graph));

    if (!G) {

        printf("Memory Error\n");

        return NULL;

    }

    G->V = 7;

    G->E = 7;

    G->Adj = (int\*\*)malloc((G->V) \* sizeof(int\*));

    for (int k = 0; k < G->V; k++) {

        G->Adj[k] = (int\*)malloc((G->V) \* sizeof(int));

    }

    for (int u = 0; u < G->V; u++) {

        for (int v = 0; v < G->V; v++) {

            G->Adj[u][v] = 0;

        }

    }

    G->Adj[0][1] = G->Adj[1][0] = 1;

    G->Adj[0][2] = G->Adj[2][0] = 1;

    G->Adj[1][3] = G->Adj[3][1] = 1;

    G->Adj[1][4] = G->Adj[4][1] = 1;

    G->Adj[1][5] = G->Adj[5][1] = 1;

    G->Adj[1][6] = G->Adj[6][1] = 1;

    G->Adj[6][2] = G->Adj[2][6] = 1;

    return G;

}

void DFS(struct Graph\* G, int u)

{

    vis[u] = 1;

    printf("%d ", u);

    for (int v = 0; v < G->V; v++) {

        if (!vis[v] && G->Adj[u][v]) {

            DFS(G, v);

        }

    }

}

void DFStraversal(struct Graph\* G)

{

    for (int i = 0; i < 100; i++) {

        vis[i] = 0;

    }

    for (int i = 0; i < G->V; i++) {

        if (!vis[i]) {

            DFS(G, i);

        }

    }

}

void main()

{

    struct Graph\* G;

    G = adjMatrix();

    DFStraversal(G);

}

**BFS PROGRAM**

#include <stdio.h>

#include <stdlib.h>

struct node {

    int vertex;

    struct node\* next;

};

struct adj\_list {

    struct node\* head;

};

struct graph {

    int num\_vertices;

    struct adj\_list\* adj\_lists;

    int\* visited;

};

struct node\* new\_node(int vertex) {

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

new\_node->vertex = vertex;

new\_node->next = NULL;

    return new\_node;

}

struct graph\* create\_graph(int n) {

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

    graph->num\_vertices = n;

    graph->adj\_lists = (struct adj\_list\*)malloc(n \* sizeof(struct adj\_list));

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

    int i;

    for (i = 0; i< n; i++) {

        graph->adj\_lists[i].head = NULL;

        graph->visited[i] = 0;

    }

    return graph;

}

void add\_edge(struct graph\* graph, int src, int dest) {

    struct node\* new\_node1 = new\_node(dest);

    new\_node1->next = graph->adj\_lists[src].head;

    graph->adj\_lists[src].head = new\_node1;

    struct node\* new\_node2 = new\_node(src);

    new\_node2->next = graph->adj\_lists[dest].head;

    graph->adj\_lists[dest].head = new\_node2;

}

void bfs(struct graph\* graph, int v) {

    int queue[1000];

    int front = -1;

    int rear = -1;

    graph->visited[v] = 1;

    queue[++rear] = v;

    while (front != rear) {

        int current\_vertex = queue[++front];

printf("%d ", current\_vertex);

        struct node\* temp = graph->adj\_lists[current\_vertex].head;

        while (temp != NULL) {

            int adj\_vertex = temp->vertex;

            if (graph->visited[adj\_vertex] == 0) {

                graph->visited[adj\_vertex] = 1;

                queue[++rear] = adj\_vertex;

            }

            temp = temp->next;

        }

    }

}

int main() {

    struct graph\* graph = create\_graph(6);

add\_edge(graph, 0, 1);

add\_edge(graph, 0, 2);

add\_edge(graph, 1, 3);

add\_edge(graph, 1, 4);

add\_edge(graph, 2, 4);

add\_edge(graph, 3, 4);

add\_edge(graph, 3, 5);

add\_edge(graph, 4,5);

printf("BFS traversal starting from vertex 0: ");

bfs(graph, 0);

    return 0;

}

OUTPUT:





|  |  |  |
| --- | --- | --- |
| **Ex. No.:12** | **Topological Sorting** | **Date:15/05/2024** |

**Write a C program to create a graph and display the ordering of vertices.**

**Algorithm:**

Step 1: Find the indegree for every vertex.

Step 2: Place the vertices whose indegree is 0 on the empty queue.

Step 3: Dequeue the vertex v and decrement the indegree of all its adjacent vertices.

Step 4: Enqueue the vertex on the queue if its indegree falls to zero.

Step 5: Repeat from step 3 until the queue becomes empty.

Step 6: The topological ordering is the order in which the vertices dequeue.

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

int s[100], j, res[100];

void AdjacencyMatrix(int a[][100], int n) {

    int i, j;

    for (i = 0; i < n; i++) {

        for (j = 0; j <= n; j++) {

            a[i][j] = 0;

        }

    }

    for (i = 1; i < n; i++) {

        for (j = 0; j < i; j++) {

            a[i][j] = rand() % 2;

            a[j][i] = 0;

        }

    }

}

void dfs(int u, int n, int a[][100]) {

    int v;

    s[u] = 1;

    for (v = 0; v < n - 1; v++) {

        if (a[u][v] == 1 && s[v] == 0) {

            dfs(v, n, a);

        }

    }

    j += 1;

    res[j] = u;

}

void topological\_order(int n, int a[][100]) {

    int i, u;

    for (i = 0; i < n; i++) {

        s[i] = 0;

    }

    j = 0;

    for (u = 0; u < n; u++) {

        if (s[u] == 0) {

            dfs(u, n, a);

        }

    }

    return;

}

int main() {

    int a[100][100], n, i, j;

    printf("Enter number of vertices\n");

    scanf("%d", &n);

    AdjacencyMatrix(a, n); /\*GENERATE ADJACENCY MATRIX \*/

    printf("\t\tAdjacency Matrix of the graph\n");

    for (i = 0; i < n; i++) {

        for (j = 0; j < n; j++) {

            printf("\t%d", a[i][j]);

        }

        printf("\n");

    }

    printf("\nTopological order:\n");

    topological\_order(n, a);

    for (i = n; i >= 1; i--) {

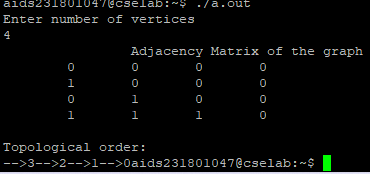
        printf("-->%d", res[i]);

    }

    return 0;

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| **Ex. No.:13** | **Graph Traversal** | **Date:22/05/2024** |

**Write a C program to create a graph and find a minimum spanning tree using prims algorithm.**

**Algorithm:**

**Step 1:** Determine an arbitrary vertex as the starting vertex of the MST.  
**Step 2:** Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).  
**Step 3:** Find edges connecting any tree vertex with the fringe vertices.  
**Step 4:** Find the minimum among these edges.  
**Step 5:** Add the chosen edge to the MST if it does not form any cycle.  
**Step 6:** Return the MST and exit

PROGRAM:

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minKey(int key[], int mstSet[], int vertices) {

    int min = INT\_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {

        if (!mstSet[v] && key[v] < min) {

            min = key[v];

            minIndex = v;

        }

    }

    return minIndex;

}

void printMST(int parent[], int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

    printf("Edge \tWeight\n");

    for (int i = 1; i < vertices; i++) {

        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);

    }

}

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

    int parent[MAX\_VERTICES];

    int key[MAX\_VERTICES];

    int mstSet[MAX\_VERTICES]; in MST

    for (int i = 0; i < vertices; i++) {

        key[i] = INT\_MAX;

        mstSet[i] = 0;

    }

    key[0] = 0;

    parent[0] = -1;

    for (int count = 0; count < vertices - 1; count++) {

        int u = minKey(key, mstSet, vertices);

        mstSet[u] = 1;

        for (int v = 0; v < vertices; v++) {

            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

                parent[v] = u;

                key[v] = graph[u][v];

            }

        }

    }

    printMST(parent, graph, vertices);

}

int main() {

    int vertices;

    printf("Input the number of vertices: ");

    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX\_VERTICES) {

        printf("Invalid number of vertices. Exiting...\n");

        return 1;

    }

    int graph[MAX\_VERTICES][MAX\_VERTICES];

    printf("Input the adjacency matrix for the graph:\n");

    for (int i = 0; i < vertices; i++) {

        for (int j = 0; j < vertices; j++) {

            scanf("%d", &graph[i][j]);

        }

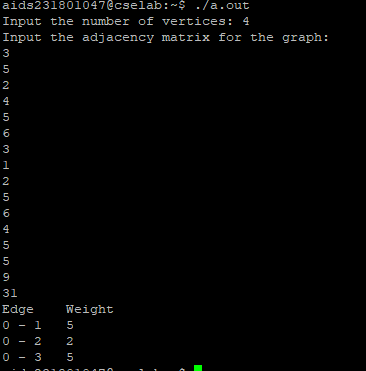
    }

    primMST(graph, vertices);

    return 0;

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| **Ex. No.:14** | **Graph Traversal** | **Date:22/05/2024** |

**Write a C program to create a graph and find the shortest path using Dijikstra’s Algorithm.**

**Algorithm:**

STEP1:Mark the source node with a current distance of 0 and the rest with infinity.

STEP 2:Set the non-visited node with the smallest current distance as the current node.

STEP 3:For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.

STEP 4:Mark the current node 1 as visited.

STEP 5:Go to step 2 if there are any nodes are unvisited.

PROGRAM:

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minDistance(int dist[], int sptSet[], int vertices) {

    int min = INT\_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {

        if (!sptSet[v] && dist[v] < min) {

            min = dist[v];

            minIndex = v;

        }

    }

    return minIndex;

}

void printSolution(int dist[], int vertices) {

    printf("Vertex \tDistance from Source\n");

    for (int i = 0; i < vertices; i++) {

        printf("%d \t%d\n", i, dist[i]);

    }

}

void dijkstra(int graph[MAX\_VERTICES][MAX\_VERTICES], int src, int vertices) {

    int dist[MAX\_VERTICES];

    int sptSet[MAX\_VERTICES

    for (int i = 0; i < vertices; i++) {

        dist[i] = INT\_MAX;

        sptSet[i] = 0;

    }

    dist[src] = 0;

    for (int count = 0; count < vertices - 1; count++) {

        int u = minDistance(dist, sptSet, vertices);

        sptSet[u] = 1;

        for (int v = 0; v < vertices; v++) {

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

                dist[v] = dist[u] + graph[u][v];

            }

        }

    }

    printSolution(dist, vertices);

}

int main() {

    int vertices;

    printf("Input the number of vertices: ");

    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX\_VERTICES) {

        printf("Invalid number of vertices. Exiting...\n");

        return 1;

    }

    int graph[MAX\_VERTICES][MAX\_VERTICES];

    printf("Input the adjacency matrix for the graph (use INT\_MAX for infinity):\n");

    for (int i = 0; i < vertices; i++) {

        for (int j = 0; j < vertices; j++) {

            scanf("%d", &graph[i][j]);

        }

    }

    int source;

printf("Input the source vertex: ");

    scanf("%d", &source);

    if (source < 0 || source >= vertices) {

        printf("Invalid source vertex. Exiting...\n");

        return 1;

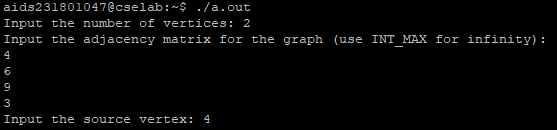
    }

    dijkstra(graph, source, vertices);

    return 0;

}

OUTPUT:



|  |  |  |
| --- | --- | --- |
| **Ex. No.:15** | **Sorting** | **Date:29/05/2024** |

**Write a C program to take n numbers and sort the numbers in ascending order. Try to implement the same using following sorting techniques.**

1. **Quick Sort**
2. **Merge Sort**

**Algorithm:**

**QUICK SORT:**

Step 1 : Start.

Step 2 : If LEFT < RIGHT then Goto

Step 3 else Goto Step 23. Step 3 : Set PIVOT = LEFT.

Step 4 : Set I = LEFT + 1.

Step 5 : Set J = RIGHT.

Step 6 : Repeat While I < J.

Step 7 : Repeat While A[I] < A[PIVOT].

Step 8 : Increment I by 1.

Step 9 : [End of Step 7 While loop].

Step 10 : Repeat While A[J] > A[PIVOT].

Step 11 : Decrement J by 1.

Step 12 : [End of Step 10 While loop].

Step 13 : If I < J then Goto Step 14 else Goto Step 17.

Step 14 : Set TEMP = A[I].

Step 15 : Set A[I] = A[J].

Step 16 : Set A[J] = TEMP.

Step 17 : [End of Step 6 While loop].

Step 18 : Set TEMP = A[PIVOT].

Step 19 : Set A[PIVOT] = A[J].

Step 20 : Set A[J] = TEMP.

Step 21 : QUICKSORT(LEFT, J – 1),

Step 22 : QUICKSORT(J + 1, RIGHT).

Step 23 : Stop.

**MERGE SORT:**

Step 1 : Start.

Step 2 : Set N1 = CENTER - LEFT + 1.

Step 3 : Set N2 = RIGHT - CENTER.

Step 4 : Repeat For I = 0 to N1 - 1.

Step 5 : Set A[I] = ARR[LEFT + I].

Step 6 : Increment I by 1.

Step 7 : [End of Step 4 For loop].

Step 8 : Repeat For J = 0 to N2 - 1.

Step 9 : Set B[J] = ARR[CENTER + 1 + J].

Step 10 : Increment J by 1.

Step 11 : [End of Step 8 For loop].

Step 12 : Repeat While APTR < N1 AND BPTR < N2.

Step 13 : If A[APTR]<= B[BPTR] then Goto Step 14 else Goto Step 18.

Step 14 : Set ARR[CPTR] = A[APTR].

Step 15 : Increment APTR by 1 and Goto Step 19.

Step 16 : Set ARR[CPTR] = B[BPTR].

Step 17 : Increment BPTR by 1.

Step 18 : Increment CPTR by 1.

Step 19 : [End of Step 12 While loop].

Step 20 : Repeat While APTR < N1.

Step 21 : Set ARR[CPTR] = A[APTR].

Step 22 : Increment APTR by 1.

Step 23 : Increment CPTR by 1.

Step 24 : [End of Step 20 While loop].

Step 25 : Repeat While BPTR < N2.

Step 26 : Set ARR[CPTR] = B[BPTR].

Step 27 : Increment BPTR by 1.

Step 28 : Increment CPTR by 1.

Step 29 : [End of Step 25 While loop].

Step 30 : Stop.

**PROGRAM:**

**QUICK SORT:**

#include <stdio.h>

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

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high)

{

int pivot = arr[low];

int i = low;

int j = high;

while (i < j) {

)

while (arr[i] <= pivot && i <= high - 1) {

i++;

}

while (arr[j] > pivot && j >= low + 1) {

j--;

}

if (i < j) {

swap(&arr[i], &arr[j]);

}

}

swap(&arr[low], &arr[j]);

return j;

}

void quickSort(int arr[], int low, int high)

{

if (low < high) {

int partitionIndex = partition(arr, low, high);

quickSort(arr, low, partitionIndex - 1);

quickSort(arr, partitionIndex + 1, high);

}

}

int main()

{

int arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

quickSort(arr, 0, n - 1);

printf("\nSorted array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

return 0;

}

**MERGE SORT:**

#include <stdio.h>

#include <stdlib.h>

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;

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 l, int r)

{

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

void printArray(int A[], int size)

{

int i;

for (i = 0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

}

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

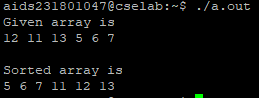
printArray(arr, arr\_size);

return 0;

}

OUTPUT:





|  |  |  |
| --- | --- | --- |
| **Ex. No.:16** | **Hashing** | **Date:29/05/2024** |

**Write a C program to create a hash table and perform collision resolution using the following techniques.**

1. **Open addressing**
2. **Closed Addressing**
3. **Rehashing**

**Algorithm:**

**Step 1:**We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.

**Step 2:**So, let’s assign

“a” = 1,

“b”=2, .. etc, to all alphabetical characters.

**Step 3:**Therefore, the numerical value by summation of all characters of the string:

“ab” = 1 + 2 = 3,

“cd” = 3 + 4 = 7 ,

“efg” = 5 + 6 + 7 = 18

**Step 4:**Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.

**Step 5:**So we will then store

“ab” in 3 mod 7 = 3,

“cd” in 7 mod 7 = 0, and

“efg” in 18 mod 7 = 4.

**PROGRAM:**

**OPEN ADDRESSING:**

#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };

int b[10];

void merging(int low, int mid, int high) {

   int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {

      if(a[l1] <= a[l2])

         b[i] = a[l1++];

      else

         b[i] = a[l2++];

   }

   while(l1 <= mid)

      b[i++] = a[l1++];

   while(l2 <= high)

      b[i++] = a[l2++];

   for(i = low; i <= high; i++)

      a[i] = b[i];

}

void sort(int low, int high) {

   int mid;

   if(low < high) {

      mid = (low + high) / 2;

      sort(low, mid);

      sort(mid+1, high);

      merging(low, mid, high);

   } else {

      return;

   }

}

int main() {

   int i;

   printf("List before sorting\n");

   for(i = 0; i <= max; i++)

      printf("%d ", a[i]);

   sort(0, max);

   printf("\nList after sorting\n");

   for(i = 0; i <= max; i++)

      printf("%d ", a[i]);

}

CLOSE ADDRESSING:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Node {

    int key;

    int value;

    struct Node\* next;

} Node;

typedef struct HashTable {

    int size;

    Node\*\* table;

} HashTable;

Node\* createNode(int key, int value) {

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

    newNode->key = key;

    newNode->value = value;

    newNode->next = NULL;

    return newNode;

}

HashTable\* createTable(int size) {

    HashTable\* newTable = (HashTable\*)malloc(sizeof(HashTable));

    newTable->size = size;

    newTable->table = (Node\*\*)malloc(sizeof(Node\*) \* size);

    for (int i = 0; i < size; i++) {

        newTable->table[i] = NULL;

    }

    return newTable;

}

int hashFunction(int key, int size) {

    return key % size;

}

void insert(HashTable\* hashTable, int key, int value) {

    int hashIndex = hashFunction(key, hashTable->size);

    Node\* newNode = createNode(key, value);

    newNode->next = hashTable->table[hashIndex];

    hashTable->table[hashIndex] = newNode;

}

int search(HashTable\* hashTable, int key) {

    int hashIndex = hashFunction(key, hashTable->size);

    Node\* current = hashTable->table[hashIndex];

    while (current != NULL) {

        if (current->key == key) {

            return current->value;

        }

        current = current->next;

    }

    return -1

}

void delete(HashTable\* hashTable, int key) {

    int hashIndex = hashFunction(key, hashTable->size);

    Node\* current = hashTable->table[hashIndex];

    Node\* prev = NULL;

    while (current != NULL && current->key != key) {

        prev = current;

        current = current->next;

    }

    if (current == NULL) {

        // Key not found

        return;

    }

    if (prev == NULL) {

        // Node to delete is the first node in the list

        hashTable->table[hashIndex] = current->next;

    } else {

        prev->next = current->next;

    }

    free(current);

}

void freeTable(HashTable\* hashTable) {

    for (int i = 0; i < hashTable->size; i++) {

        Node\* current = hashTable->table[i];

        while (current != NULL) {

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(hashTable->table);

    free(hashTable);

}

int main() {

    HashTable\* hashTable = createTable(10);

    insert(hashTable, 1, 10);

    insert(hashTable, 2, 20);

    insert(hashTable, 12, 30);

    printf("Value for key 1: %d\n", search(hashTable, 1));

    printf("Value for key 2: %d\n", search(hashTable, 2));

    printf("Value for key 12: %d\n", search(hashTable, 12));

    printf("Value for key 3: %d\n", search(hashTable, 3)); // Key not present

    delete(hashTable, 2);

    printf("Value for key 2 after deletion: %d\n", search(hashTable, 2));

    freeTable(hashTable);

    return 0;

}

REHASHING:

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

    int key;

    int value;

    struct Node\* next;

} Node;

typedef struct HashTable {

    int size;

    int count

    Node\*\* table;

} HashTable;

Node\* createNode(int key, int value) {

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

    newNode->key = key;

    newNode->value = value;

    newNode->next = NULL;

    return newNode;

}

HashTable\* createTable(int size) {

    HashTable\* newTable = (HashTable\*)malloc(sizeof(HashTable));

    newTable->size = size;

    newTable->count = 0;

    newTable->table = (Node\*\*)malloc(sizeof(Node\*) \* size);

    for (int i = 0; i < size; i++) {

        newTable->table[i] = NULL;

    }

    return newTable;

}

int hashFunction(int key, int size) {

    return key % size;

}

void insert(HashTable\* hashTable, int key, int value);

void rehash(HashTable\* hashTable) {

    int oldSize = hashTable->size;

    Node\*\* oldTable = hashTable->table;

    int newSize = oldSize \* 2;

    hashTable->table = (Node\*\*)malloc(sizeof(Node\*) \* newSize);

    hashTable->size = newSize;

    hashTable->count = 0;

    for (int i = 0; i < newSize; i++) {

        hashTable->table[i] = NULL;

    }

    for (int i = 0; i < oldSize; i++) {

        Node\* current = oldTable[i];

        while (current != NULL) {

            insert(hashTable, current->key, current->value);

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(oldTable);

}

void insert(HashTable\* hashTable, int key, int value) {

    if ((float)hashTable->count / hashTable->size >= 0.75) {

        rehash(hashTable);

    }

    int hashIndex = hashFunction(key, hashTable->size);

    Node\* newNode = createNode(key, value);

    newNode->next = hashTable->table[hashIndex];

    hashTable->table[hashIndex] = newNode;

    hashTable->count++;

}

int search(HashTable\* hashTable, int key) {

    int hashIndex = hashFunction(key, hashTable->size);

    Node\* current = hashTable->table[hashIndex];

    while (current != NULL) {

        if (current->key == key) {

            return current->value;

        }

        current = current->next;

    }

    return -1;

}

void delete(HashTable\* hashTable, int key) {

    int hashIndex = hashFunction(key, hashTable->size);

    Node\* current = hashTable->table[hashIndex];

    Node\* prev = NULL;

    while (current != NULL && current->key != key) {

        prev = current;

        current = current->next;

    }

    if (current == NULL) {

        return;

    }

    if (prev == NULL) {

        hashTable->table[hashIndex] = current->next;

    } else {

        prev->next = current->next;

    }

    free(current);

    hashTable->count--;

}

void freeTable(HashTable\* hashTable) {

    for (int i = 0; i < hashTable->size; i++) {

        Node\* current = hashTable->table[i];

        while (current != NULL) {

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(hashTable->table);

    free(hashTable);

}

int main() {

    HashTable\* hashTable = createTable(5);

    insert(hashTable, 1, 10);

    insert(hashTable, 2, 20);

    insert(hashTable, 3, 30);

    insert(hashTable, 4, 40);

    insert(hashTable, 5, 50);

    insert(hashTable, 6, 60);

    printf("Value for key 1: %d\n", search(hashTable, 1));

    printf("Value for key 2: %d\n", search(hashTable, 2));

    printf("Value for key 3: %d\n", search(hashTable, 3));

    printf("Value for key 4: %d\n", search(hashTable, 4));

    printf("Value for key 5: %d\n", search(hashTable, 5));

    printf("Value for key 6: %d\n", search(hashTable, 6));

    delete(hashTable, 3);

    printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));

    freeTable(hashTable);

    return 0;

}

OUTPUT:

