|  |  |  |
| --- | --- | --- |
| Ex. No.:1 | **Implementation of Single Linked List** | Date:29.3.24 |

**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:**

**Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct node**

**{**

**int Element;**

**struct node \*Next;**

**};**

**typedef struct node Node;**

**int IsEmpty(Node \*List);**

**int IsLast(Node \*Position, Node \*List);**

**Node \*Find(Node \*List, int x);**

**Node \*FindPrevious(Node \*List, int x);**

**Node \*FindNext(Node \*List, int x);**

**void InsertBeg(Node \*List, int e);**

**void InsertLast(Node \*List, int e);**

**void InsertMid(Node \*List, int p, int e);**

**void DeleteBeg(Node \*List);**

**void DeleteEnd(Node \*List);**

**void DeleteMid(Node \*List, int e);**

**void Traverse(Node \*List);**

**int main()**

**{**

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

**List->Next = List;**

**Node \*Position;**

**int ch, e, p;**

**printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");**

**printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");**

**printf("\n7.Find \n8.Traverse \n9.Exit\n");**

**do**

**{**

**printf("Enter your choice : ");**

**scanf("%d", &ch);**

**switch(ch)**

**{**

**case 1:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**InsertBeg(List, e);**

**break;**

**case 2:**

**printf("Enter the position element : ");**

**scanf("%d", &p);**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**InsertMid(List, p, e);**

**break;**

**case 3:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**InsertLast(List, e);**

**break;**

**case 4:**

**DeleteBeg(List);**

**break;**

**case 5:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**DeleteMid(List, e);**

**break;**

**case 6:**

**DeleteEnd(List);**

**break;**

**case 7:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**Position = Find(List, e);**

**if(Position != List)**

**printf("Element found...!\n");**

**else**

**printf("Element not found...!\n");**

**break;**

**case 8:**

**Traverse(List);**

**break;**

**}**

**} while(ch <= 8);**

**return 0;**

**}**

**int IsEmpty(Node \*List)**

**{**

**if(List->Next == List)**

**return 1;**

**else**

**}**

**return 0;**

**int IsLast(Node \*Position, Node \*List)**

**{**

**if(Position->Next == List)**

**return 1;**

**else**

**}**

**return 0;**

**Node \*Find(Node \*List, int x)**

**{**

**Node \*Position;**

**Position = List->Next;**

**while(Position != List && Position->Element != x)**

**Position = Position->Next;**

**return Position;**

**}**

**Node \*FindPrevious(Node \*List, int x)**

**{**

**Node \*Position;**

**Position = List;**

**while(Position->Next != List && Position->Next->Element != x)**

**Position = Position->Next;**

**return Position;**

**}**

**Node \*FindNext(Node \*List, int x)**

**{**

**Node \*Position;**

**Position = Find(List, x);**

**return Position->Next;**

**}**

**void InsertBeg(Node \*List, int e)**

**{**

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

**NewNode->Element = e;**

**NewNode->Next = List->Next;**

**List->Next = NewNode;**

**}**

**void InsertLast(Node \*List, int e)**

**{**

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

**Node \*Position;**

**NewNode->Element = e;**

**if(IsEmpty(List))**

**{**

**}**

**else**

**{**

**}**

**}**

**NewNode->Next = List;**

**List->Next = NewNode;**

**Position = List;**

**while(Position->Next != List)**

**Position = Position->Next;**

**Position->Next = NewNode;**

**NewNode->Next = List;**

**void InsertMid(Node \*List, int p, int e)**

**{**

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

**Node \*Position;**

**Position = Find(List, p);**

**NewNode->Element = e;**

**NewNode->Next = Position->Next;**

**Position->Next = NewNode;**

**}**

**void DeleteBeg(Node \*List)**

**{**

**if(!IsEmpty(List))**

**{**

**Node \*TempNode;**

**TempNode = List->Next;**

**List->Next = TempNode->Next;**

**printf("The deleted item is %d\n", TempNode->Element);**

**free(TempNode);**

**}**

**else**

**printf("List is empty...!\n");**

**}**

**void DeleteEnd(Node \*List)**

**{**

**if(!IsEmpty(List))**

**{**

**}**

**else**

**}**

**Node \*Position;**

**Node \*TempNode;**

**Position = List;**

**while(Position->Next->Next != List)**

**Position = Position->Next;**

**TempNode = Position->Next;**

**Position->Next = List;**

**printf("The deleted item is %d\n", TempNode->Element);**

**free(TempNode);**

**printf("List is empty...!\n");**

**void DeleteMid(Node \*List, int e)**

**{**

**if(!IsEmpty(List))**

**{**

**Node \*Position;**

**Node \*TempNode;**

**Position = FindPrevious(List, e);**

**if(!IsLast(Position, List))**

**{**

**}**

**}**

**else**

**TempNode = Position->Next;**

**Position->Next = TempNode->Next;**

**printf("The deleted item is %d\n", TempNode->Element);**

**free(TempNode);**

**printf("List is empty...!\n");**

**}**

**void Traverse(Node \*List)**

**{**

**if(!IsEmpty(List))**

**{**

**Node \*Position;**

**Position = List;**

**while(Position->Next != List)**

**{**

**Position = Position->Next;**

**printf("%d\t", Position->Element);**

**}**

**}**

**else**

**}**

**printf("\n");**

**printf("List is empty...!\n");**

**}**

**OUTPUT**

**1.Insert Beg**

**2.Insert Middle**

**3.Insert End**

**4.Delete Beg**

**5.Delete Middle**

**6.Delete End**

**7.Find**

**8.Traverse**

**9.Exit Enter your choice : 1**

**Enter the element : 40**

**Enter your choice : 1**

**Enter the element : 30**

**Enter your choice : 1**

**Enter the element : 20**

**Enter your choice : 1**

**Enter the element : 10**

**Enter your choice : 8**

**10 20 30 40**

**Enter your choice : 7**

**Enter the element : 30**

**Element found...!**

**Enter your choice : 1**

**Enter the element : 5**

**Enter your choice : 8**

**5 10 20 30 40**

**Enter your choice : 3**

**Enter the element : 45**

**Enter your choice : 8 5 10 20 30 40 45**

**Enter your choice : 2**

**Enter the position element : 20**

**Enter the element : 25**

**Enter your choice : 8**

**5 10 20 25 30 40 45**

**Enter your choice : 4**

**The deleted item is 5**

**Enter your choice : 8**

**10 20 25 30 40 45**

**Enter your choice : 6**

**The deleted item is 45**

**Enter your choice : 8**

**10 20 25 30 40**

**Enter your choice : 5**

**Enter the element : 30**

**The deleted item is 30**

**Enter your choice : 8**

**10 20 25 40**

**Enter your choice:9**

|  |  |  |
| --- | --- | --- |
| Ex. No.:2 | **Implementation of Doubly Linked List** | Date:2.3.24 |

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

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

**Algorithm:**

**Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct node**

**{**

**struct node \*Prev;**

**int Element;**

**struct node \*Next;**

**};**

**typedef struct node Node;**

**int IsEmpty(Node \*List);**

**int IsLast(Node \*Position);**

**Node \*Find(Node \*List, int x);**

**void InsertBeg(Node \*List, int e);**

**void InsertLast(Node \*List, int e);**

**void InsertMid(Node \*List, int p, int e);**

**void DeleteBeg(Node \*List);**

**void DeleteEnd(Node \*List);**

**void DeleteMid(Node \*List, int e);**

**void Traverse(Node \*List);**

**int main()**

**{**

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

**List->Prev = NULL;**

**List->Next = NULL;**

**Node \*Position;**

**int ch, e, p;**

**printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");**

**printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");**

**printf("\n7.Find \n8.Traverse \n9.Exit\n");**

**do**

**{**

**printf("Enter your choice : ");**

**scanf("%d", &ch);**

**switch(ch)**

**{**

**case 1:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**InsertBeg(List, e);**

**break;**

**case 2:**

**printf("Enter the position element : ");**

**scanf("%d", &p);**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**InsertMid(List, p, e);**

**break;**

**case 3:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**InsertLast(List, e);**

**break;**

**case 4:**

**DeleteBeg(List);**

**break;**

**case 5:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**DeleteMid(List, e);**

**break;**

**case 6:**

**DeleteEnd(List);**

**break;**

**case 7:**

**printf("Enter the element : ");**

**scanf("%d", &e);**

**Position = Find(List, e);**

**if(Position != NULL)**

**printf("Element found...!\n");**

**else**

**printf("Element not found...!\n");**

**break;**

**case 8:**

**Traverse(List);**

**break;**

**}**

**} while(ch <= 8);**

**return 0;**

**}**

**int IsEmpty(Node \*List)**

**{**

**if(List->Next == NULL)**

**return 1;**

**else**

**}**

**return 0;**

**int IsLast(Node \*Position)**

**{**

**if(Position->Next == NULL)**

**return 1;**

**else**

**}**

**return 0;**

**Node \*Find(Node \*List, int x)**

**{**

**Node \*Position;**

**Position = List->Next;**

**while(Position != NULL && Position->Element != x)**

**Position = Position->Next;**

**return Position;**

**}**

**void InsertBeg(Node \*List, int e)**

**{**

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

**NewNode->Element = e;**

**if(IsEmpty(List))**

**NewNode->Next = NULL;**

**else**

**{**

**}**

**NewNode->Next = List->Next;**

**NewNode->Next->Prev = NewNode;**

**NewNode->Prev = List;**

**List->Next = NewNode;**

**}**

**void InsertLast(Node \*List, int e)**

**{**

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

**Node \*Position;**

**NewNode->Element = e;**

**NewNode->Next = NULL;**

**if(IsEmpty(List))**

**{**

**}**

**else**

**{**

**}**

**}**

**NewNode->Prev = List;**

**List->Next = NewNode;**

**Position = List;**

**while(Position->Next != NULL)**

**Position = Position->Next;**

**Position->Next = NewNode;**

**NewNode->Prev = Position;**

**void InsertMid(Node \*List, int p, int e)**

**{**

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

**Node \*Position;**

**Position = Find(List, p);**

**NewNode->Element = e;**

**NewNode->Next = Position->Next;**

**Position->Next->Prev = NewNode;**

**Position->Next = NewNode;**

**NewNode->Prev = Position;**

**}**

**void DeleteBeg(Node \*List)**

**{**

**if(!IsEmpty(List))**

**{**

**Node \*TempNode;**

**TempNode = List->Next;**

**List->Next = TempNode->Next;**

**if(List->Next != NULL)**

**TempNode->Next->Prev = List;**

**printf("The deleted item is %d\n", TempNode->Element);**

**free(TempNode);**

**printf("List is empty...!\n");**

**void DeleteEnd(Node \*List)**

**{**

**if(!IsEmpty(List))**

**{**

**}**

**else**

**}**

**Node \*Position;**

**Node \*TempNode;**

**Position = List;**

**while(Position->Next != NULL)**

**Position = Position->Next;**

**TempNode = Position;**

**Position->Prev->Next = NULL;**

**printf("The deleted item is %d\n", TempNode->Element);**

**free(TempNode);**

**printf("List is empty...!\n");**

**void DeleteMid(Node \*List, int e)**

**{**

**if(!IsEmpty(List))**

**{**

**Node \*Position;**

**Node \*TempNode;**

**Position = Find(List, e);**

**if(!IsLast(Position))**

**{**

**}**

**}**

**else**

**TempNode = Position;**

**Position->Prev->Next = Position->Next;**

**Position->Next->Prev = Position->Prev;**

**printf("The deleted item is %d\n", TempNode->Element);**

**free(TempNode);**

**printf("List is empty...!\n");**

**}**

**void Traverse(Node \*List)**

**{**

**if(!IsEmpty(List))**

**{**

**Node \*Position;**

**Position = List;**

**while(Position->Next != NULL)**

**{**

**Position = Position->Next;**

**printf("%d\t", Position->Element);**

**}**

**printf("\n");**

**}**

**else**

**printf("List is empty...!\n");**

**}**

**Output**

**1.Insert Beg**

**2.Insert Middle**

**3.Insert End**

**4.Delete Beg**

**5.Delete Middle**

**6.Delete End**

**7.Find**

**8.Traverse**

**9.Exit**

**Enter your choice : 1**

**Enter the element : 40**

**Enter your choice : 1**

**Enter the element : 30**

**Enter your choice : 1**

**Enter the element : 20**

**Enter your choice : 1**

**Enter the element : 10**

**Enter your choice : 8**

**1.Insert Beg**

**2.Insert Middle**

**3.Insert End**

**4.Delete Beg**

**5.Delete Middle**

**6.Delete End**

**7.Find**

**8.Traverse**

**10 20 30 40**

**Enter your choice : 7**

**Enter the element : 30**

**Element found...!**

**Enter your choice : 1**

**Enter the element : 5**

**Enter your choice : 8**

**5 10 20 30 40**

**Enter your choice : 3**

**Enter the element : 45**

**Enter your choice : 8**

**5 10 20 30 40 45**

**Enter your choice : 2**

**Enter the position element : 20**

**Enter the element : 25**

**Enter your choice : 8**

**5 10 20 25 30 40 45**

**Enter your choice : 4**

**The deleted item is 5**

**Enter your choice : 8**

**77 10 20 25 30 40 45**

**Enter your choice : 6**

**The deleted item is 45**

**Enter your choice : 8**

**10 20 25 30 40**

**Enter your choice : 5**

**Enter the element : 30**

**The deleted item is 30**

**Enter your choice : 8**

**10 20 25 40**

**Enter your choice:9**

|  |  |  |
| --- | --- | --- |
| Ex. No.:3 | **Polynomial Manipulation** | Date:16.3.24 |

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

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

**Algorithm:**

**Code:**

**Polynomial addition:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct poly**

**{**

**int coeff;**

**int pow;**

**struct poly \*Next;**

**};**

**typedef struct poly Poly;**

**void Create(Poly \*List);**

**void Display(Poly \*List);**

**void Addition(Poly \*Poly1, Poly \*Poly2, Poly \*Result);**

**int main()**

**{**

**Poly \*Poly1 = malloc(sizeof(Poly));**

**Poly \*Poly2 = malloc(sizeof(Poly));**

**Poly \*Result = malloc(sizeof(Poly));**

**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(Poly \*List)**

**{**

**int choice;**

**Poly \*Position, \*NewNode;**

**Position = List;**

**do**

**{**

**NewNode = malloc(sizeof(Poly));**

**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(Poly \*List)**

**{**

**Poly \*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(Poly \*Poly1, Poly \*Poly2, Poly \*Result)**

**{**

**Poly \*Position;**

**Poly \*NewNode;**

**Poly1 = Poly1->Next;**

**Poly2 = Poly2->Next;**

**Result->Next = NULL;**

**Position = Result;**

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

**{**

**NewNode = malloc(sizeof(Poly));**

**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(Poly));**

**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;**

**}**

**}**

**Polynomial Subtraction:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct poly**

**{**

**int coeff;**

**int pow;**

**struct poly \*Next;**

**};**

**typedef struct poly Poly;**

**void Create(Poly \*List);**

**void Display(Poly \*List);**

**void Subtraction(Poly \*Poly1, Poly \*Poly2, Poly \*Result);**

**int main()**

**{**

**Poly \*Poly1 = malloc(sizeof(Poly));**

**Poly \*Poly2 = malloc(sizeof(Poly));**

**Poly \*Result = malloc(sizeof(Poly));**

**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);**

**Subtraction(Poly1, Poly2, Result);**

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

**Display(Result);**

**return 0;**

**}**

**void Create(Poly \*List)**

**{**

**int choice;**

**Poly \*Position, \*NewNode;**

**Position = List;**

**do**

**{**

**NewNode = malloc(sizeof(Poly));**

**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(Poly \*List)**

**{**

**Poly \*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 Subtraction(Poly \*Poly1, Poly \*Poly2, Poly \*Result)**

**{**

**Poly \*Position;**

**Poly \*NewNode;**

**Poly1 = Poly1->Next;**

**Poly2 = Poly2->Next;**

**Result->Next = NULL;**

**Position = Result;**

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

**{**

**NewNode = malloc(sizeof(Poly));**

**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(Poly));**

**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;**

**}**

**}**

**Polynomial Multiplication:**

**#include <stdio.h>**

**#include <stdlib.h>**

**typedef struct Node {**

**int data;**

**int power;**

**struct Node \* next;**

**} Node;**

**Node \* getNode(int data, int power) {**

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

**if (ref == NULL) {**

**return NULL;**

**}**

**ref->data = data;**

**ref->power = power;**

**ref->next = NULL;**

**return ref;**

**}**

**void updateRecord(Node \* ref, int data, int power) {**

**ref->data = data;**

**ref->power = power;**

**}**

**typedef struct MultiplyPolynomial {**

**struct Node \* head;**

**} MultiplyPolynomial;**

**MultiplyPolynomial \* getMultiplyPolynomial() {**

**MultiplyPolynomial \* ref = (MultiplyPolynomial \* ) malloc(sizeof(MultiplyPolynomial));**

**if (ref == NULL) {**

**return NULL;**

**}**

**ref->head = NULL;**

**return ref;**

**}**

**void insert(MultiplyPolynomial \* ref, int data, int power) {**

**if (ref->head == NULL) {**

**ref->head = getNode(data, power);**

**} else {**

**Node \* node = NULL;**

**Node \* temp = ref->head;**

**Node \* location = NULL;**

**while (temp != NULL && temp->power >= power) {**

**location = temp;**

**temp = temp->next;**

**}**

**if (location != NULL && location->power == power) {**

**location->data = location->data + data;**

**} else {**

**node = getNode(data, power);**

**if (location == NULL) {**

**node->next = ref->head;**

**ref->head = node;**

**} else {**

**node->next = location->next;**

**location->next = node;**

**}**

**}**

**}**

**}**

**MultiplyPolynomial \* multiplyPolynomials(MultiplyPolynomial \* ref, MultiplyPolynomial \* other) {**

**MultiplyPolynomial \* result = getMultiplyPolynomial();**

**Node \* poly1 = ref->head;**

**Node \* temp = other->head;**

**int power\_value = 0;**

**int coefficient = 0;**

**while (poly1 != NULL) {**

**temp = other->head;**

**while (temp != NULL) {**

**power\_value = poly1->power + temp->power;**

**coefficient = poly1->data \* temp->data;**

**insert(result, coefficient, power\_value);**

**temp = temp->next;**

**}**

**poly1 = poly1->next;**

**}**

**return result;**

**}**

**void display(MultiplyPolynomial \* ref) {**

**if (ref->head == NULL) {**

**printf("Empty Polynomial ");**

**}**

**printf(" ");**

**Node \* temp = ref->head;**

**while (temp != NULL) {**

**if (temp != ref->head) {**

**printf(" + %d", temp->data);**

**} else {**

**printf("%d", temp->data);**

**}**

**if (temp->power != 0) {**

**printf("x^%d", temp->power);**

**}**

**temp = temp->next;**

**}**

**printf("\n");**

**}**

**int main() {**

**MultiplyPolynomial \* a = getMultiplyPolynomial();**

**MultiplyPolynomial \* b = getMultiplyPolynomial();**

**insert(a, 9, 3);**

**insert(a, 4, 2);**

**insert(a, 3, 0);**

**insert(a, 7, 1);**

**insert(a, 3, 4);**

**insert(b, 7, 3);**

**insert(b, 4, 0);**

**insert(b, 6, 1);**

**insert(b, 1, 2);**

**printf("\n Polynomial A\n");**

**display(a);**

**printf(" Polynomial B\n");**

**display(b);**

**MultiplyPolynomial \* result = multiplyPolynomials(a, b);**

**printf(" Result\n");**

**display(result);**

**}**

**OUTPUT**

**Polynomial A**

**3x^4 + 9x^3 + 4x^2 + 7x^1 + 3**

**Polynomial B**

**7x^3 + 1x^2 + 6x^1 + 4**

**Result**

**21x^7 + 39x^6 + 67x^5 + 91x^4 + 55x^3 + 52x^2 + 46x^1 + 12**

|  |  |  |
| --- | --- | --- |
| Ex No:4 | **Implementation of Stack using Array and Linked List Implementation** | Date:6.4.24 |

**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:**

**Code:**

**Array Implementation:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX\_SIZE 10**

**struct ArrayStack {**

**int top;**

**int array[MAX\_SIZE];**

**};**

**struct ArrayStack\* createArrayStack() {**

**struct ArrayStack\* stack = (struct ArrayStack\*)malloc(sizeof(struct ArrayStack));**

**stack->top = -1;**

**return stack;**

**}**

**int isArrayStackEmpty(struct ArrayStack\* stack) {**

**return (stack->top == -1);**

**}**

**void pushArray(struct ArrayStack\* stack, int data) {**

**if (stack->top == MAX\_SIZE - 1) {**

**printf("Stack Overflow\n");**

**return;**

**}**

**stack->array[++stack->top] = data;**

**}**

**int popArray(struct ArrayStack\* stack) {**

**if (isArrayStackEmpty(stack)) {**

**printf("Stack Underflow\n");**

**return -1;**

**}**

**return stack->array[stack->top--];**

**}**

**int main() {**

**struct ArrayStack\* arrayStack = createArrayStack();**

**// Pushing elements onto the stack**

**pushArray(arrayStack, 10);**

**pushArray(arrayStack, 20);**

**pushArray(arrayStack, 30);**

**// Popping elements from the stack**

**printf("Popped from arrayStack: %d\n", popArray(arrayStack));**

**printf("Popped from arrayStack: %d\n", popArray(arrayStack));**

**printf("Popped from arrayStack: %d\n", popArray(arrayStack));**

**return 0;**

**}**

**Linked List Implementation:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct LinkedListStack {

struct Node\* top;

};

struct LinkedListStack\* createLinkedListStack() {

struct LinkedListStack\* stack = (struct LinkedListStack\*)malloc(sizeof(struct LinkedListStack));

stack->top = NULL;

return stack;

}

int isLinkedListStackEmpty(struct LinkedListStack\* stack) {

return (stack->top == NULL);

}

void pushLinkedList(struct LinkedListStack\* stack, int data) {

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

newNode->data = data;

newNode->next = stack->top;

stack->top = newNode;

}

int popLinkedList(struct LinkedListStack\* stack) {

if (isLinkedListStackEmpty(stack)) {

printf("Stack Underflow\n");

return -1;

}

struct Node\* temp = stack->top;

int data = temp->data;

stack->top = temp->next;

free(temp);

return data;

}

int main() {

struct LinkedListStack\* linkedListStack = createLinkedListStack();

// Pushing elements onto the stack

pushLinkedList(linkedListStack, 40);

pushLinkedList(linkedListStack, 50);

pushLinkedList(linkedListStack, 60);

// Popping elements from the stack

printf("Popped from linkedListStack: %d\n", popLinkedList(linkedListStack));

printf("Popped from linkedListStack: %d\n", popLinkedList(linkedListStack));

printf("Popped from linkedListStack: %d\n", popLinkedList(linkedListStack));

return 0;

}

**OUTPUT :**

**Stack Menu**

**1. Push**

**2. Pop**

**3. Display**

**4. Exit**

**Enter your choice: 1**

**Enter data to push: 10**

**10 pushed to stack**

**Enter your choice: 1**

**Enter data to push: 20**

**20 pushed to stack**

**Enter your choice: 1**

**Enter data to push: 30**

**30 pushed to stack**

**Enter your choice: 3**

**Elements in stack: 30 20 10**

**Enter your choice: 2**

**Popped element: 30**

**Enter your choice: 3**

**Elements in stack: 20 10**

**Enter your choice: 2**

**Popped element: 20**

**Enter your choice: 2**

**Popped element: 10**

**Enter your choice: 2**

**Stack Underflow**

**Enter your choice: 3**

**Stack is empty**

**Enter your choice: 4**

**Exiting program**

|  |  |  |
| --- | --- | --- |
| Ex. No.:5 | **Infix to Postfix Conversion** | Date:8.4.24 |

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

**Algorithm:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIZE 100

int isOperator(char ch) {

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/');

}

int precedence(char ch) {

if (ch == '+' || ch == '-')

return 1;

else if (ch == '\*' || ch == '/')

return 2;

else

return 0;

}

void infixToPostfix(char \*infix, char \*postfix) {

int i = 0, j = 0;

char stack[MAX\_SIZE];

int top = -1;

while (infix[i] != '\0') {

if (isalnum(infix[i]))

postfix[j++] = infix[i++];

else if (infix[i] == '(')

stack[++top] = infix[i++];

else if (infix[i] == ')') {

while (stack[top] != '(')

postfix[j++] = stack[top--];

top--;

i++;

} else if (isOperator(infix[i])) {

while (top != -1 && precedence(stack[top]) >= precedence(infix[i]))

postfix[j++] = stack[top--];

stack[++top] = infix[i++];

}

}

while (top != -1)

postfix[j++] = stack[top--];

postfix[j] = '\0';

}

int main() {

char infix[MAX\_SIZE], postfix[MAX\_SIZE];

printf("Enter infix expression: ");

scanf("%s", infix);

infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

**OUTPUT:**

**Enter infix expression: a+b\*c**

**Postfix expression: abc\*+**

|  |  |  |
| --- | --- | --- |
| Ex. No.:6 | **Evaluating Arithmetic Expression** | Date:13.4.24 |

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

**Algorithm:**

Code:

#include <ctype.h>

#define MAX\_SIZE 100

char stack[MAX\_SIZE];

int top = -1;

void push(char value) {

stack[++top] = value;

}

char pop() {

return stack[top--];

}

int precedence(char op) {

switch(op) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

default:

return 0;

}

}

void infixToPostfix(char \*infix, char \*postfix) {

int i, j;

for (i = 0, j = 0; infix[i] != '\0'; i++) {

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

} else if (infix[i] == '(') {

push(infix[i]);

} else if (infix[i] == ')') {

while (stack[top] != '(') {

postfix[j++] = pop();

}

pop(); // Discard '('

} else {

while (top != -1 && precedence(stack[top]) >= precedence(infix[i])) {

postfix[j++] = pop();

}

push(infix[i]);

}

}

while (top != -1) {

postfix[j++] = pop();

}

postfix[j] = '\0';

}

**INPUT:**

**5 3 + 4 \* 6 2 / -**

**OUTPUT:**

**Result : 32**

|  |  |  |
| --- | --- | --- |
| Ex. No.:7 | **Implementation of Queue using Array and Linked List Implementation** | Date:13.4.24 |

**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:**

CODE:-

//Queue linked list

#include <stdio.h>

#include <stdlib.h>

struct Queue

{ int ele;

struct Queue \*next;};

typedef struct Queue q;

q \*f=NULL;

q \*r=NULL;

void enqueue(int x)

{ q \*newnode=malloc(sizeof(q));

newnode->ele=x;

if(f==NULL && r==NULL)

{ f=r=newnode;

newnode->next=NULL;

return;

}

r->next=newnode;

r=newnode;

newnode->next=NULL;}

//f=newnode;}

void dequeue()

{ if(f==NULL && r==NULL)

{ printf("UNDERFLOW\n");

return;}

if(f==r)

{ printf("THE DELETED ELE IS %d\n",f->ele);

f=r=NULL;

return;}

q \*temp=f;

printf("DELETED ELEMENT IS %d\n",temp->ele);

f=f->next;

free(temp);

}

void display()

{ q \*temp=f;

while(temp!=NULL)

{ printf("%d ",temp->ele);

temp=temp->next;

}

printf("\n");

}

int main()

{

int ch;

printf("1 TO ENQUEUE\n2 TO DEQUEUE\n3 TO DISPLAY\n");

do

{ printf("ENTER YOUR CHOICE ");

scanf("%d",&ch);

switch(ch)

{ case 1:

int x;

printf("ELEMENT TO BE ADDED");

scanf("%d",&x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

default:

break;

} } while(ch<=3);

printf("THANK YOU");

}

//QUEUE USING ARRAY IMPLEMENTATION

//queue array

#include <stdio.h>

#include <stdlib.h>

#define SIZE 100

int q[SIZE];

int f=-1,r=-1;

void enqueue(int x)

{ if(f==-1 && r==-1)

{ f++;

r++;

q[f]=x;

return;

}

if(r==SIZE-1)

{ printf("OVERFLOW\n");

return;}

r++;

q[r]=x;

}

void dequeue()

{ if(f==-1 && r==-1)

{ printf("UNDERFLOW\n");

return;}

if(f==r)

{ printf("THE DELETED ELE %d\n",q[f]);

f=r=-1;

return;}

printf("The deleted element is %d\n",q[f]);

f++;

}

void display()

{ for(int i=f;i<=r;i++)

{ printf("%d ",q[i]);

}

printf("\n");}

int main()

{

int ch;

printf("1 TO ENQUEUE\n2 TO DEQUEUE\n3 TO DISPLAY\n");

do

{ printf("ENTER YOUR CHOICE ");

scanf("%d",&ch);

switch(ch)

{ case 1:

int x;

printf("ELEMENT TO BE ADDED");

scanf("%d",&x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

default:

break;

} } while(ch<=3);

printf("THANK YOU");

}

OUTPUT:-

1 TO ENQUEUE

2 TO DEQUEUE

3 TO DISPLAY

ENTER YOUR CHOICE 1

ELEMENT TO BE ADDED20

ENTER YOUR CHOICE 1

ELEMENT TO BE ADDED30

ENTER YOUR CHOICE 1

ELEMENT TO BE ADDED40

ENTER YOUR CHOICE 3

20 30 40

ENTER YOUR CHOICE 2

DELETED ELEMENT IS 20

ENTER YOUR CHOICE 3

30 40

ENTER YOUR CHOICE 4

THANK YOU

|  |  |  |
| --- | --- | --- |
| Ex. No.:8 | **Tree Traversal** | Date:4.5.24 |

**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:**

CODE:-

#include <stdio.h>

#include <stdlib.h>

struct Tree

{ int ele;

struct Tree \*left;

struct Tree \*right;};

typedef struct Tree tree;

//tree \*root=NULL;

tree \*create(tree \*root,int x)

{ if(root==NULL)

{ tree \*newnode=malloc(sizeof(tree));

newnode->ele=x;

newnode->left=NULL;

newnode->right=NULL;

root=newnode;}

else if(x<root->ele)

{ root->left=create(root->left,x);

}

else if(x>root->ele)

{ root->right=create(root->right,x);

}

return root;

}

void inorder(tree \*root)

{ if(root!=NULL)

{ inorder(root->left);

printf("%d ",root->ele);

inorder(root->right);

}

}

void preorder(tree \*root)

{ if(root!=NULL)

{

printf("%d ",root->ele);

preorder(root->left);

preorder(root->right);

}

}

void postorder(tree \*root)

{ if(root!=NULL)

{

postorder(root->left);

postorder(root->right);

printf("%d ",root->ele);

}

}

int main()

{ tree \*root=NULL;

int n,x;

printf("ENTER NO OF ELEMENTS");

scanf("%d",&n);

printf("ENTER THE ELEMENTS ");

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

{ scanf("%d",&x);

root=create(root,x);

}

printf("INORDER TRAVERSAL IS ");

inorder(root);

printf("\nPOSTORDER TRAVERSAL IS ");

postorder(root);

printf("\nPREORDER TRAVERSAL IS ");

preorder(root);

return 0;

}

OUTPUT:-

ENTER NO OF ELEMENTS7

ENTER THE ELEMENTS 100 90 110 80 95 105 111

INORDER TRAVERSAL IS 80 90 95 100 105 110 111

POSTORDER TRAVERSAL IS 80 95 90 105 111 110 100

PREORDER TRAVERSAL IS 100 90 80 95 110 105 111

|  |  |  |
| --- | --- | --- |
| Ex. No.:9 | **Implementation of Binary Search tree** | Date:4.5.24 |

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

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

**Algorithm:**

CODE:-

#include <stdio.h>

#include <stdlib.h>

// Define a structure for the BST node

struct Node {

int data;

struct Node \*left, \*right;

};

// Function to create a new BST node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to insert a node into the BST

struct Node\* insertNode(struct Node\* root, int data) {

if (root == NULL) return createNode(data);

if (data < root->data)

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

else if (data > root->data)

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

return root;

}

// Function to find the minimum value node in a tree

struct Node\* findMin(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

// Function to delete a node from the BST

struct Node\* deleteNode(struct Node\* root, int data) {

if (root == NULL) return root;

if (data < root->data)

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

else if (data > root->data)

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

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// Function to search a node in the BST

struct Node\* searchNode(struct Node\* root, int data) {

if (root == NULL || root->data == data)

return root;

if (root->data < data)

return searchNode(root->right, data);

return searchNode(root->left, data);

}

// Function to perform in-order traversal and display the tree

void inOrder(struct Node\* root) {

if (root != NULL) {

inOrder(root->left);

printf("%d ", root->data);

inOrder(root->right);

}

}

int main() {

struct Node\* root = NULL;

int choice, data,n;

printf("Enter the no of elements to be inserted");

scanf("%d",&n);

printf("Enter elements");

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

{ scanf("%d",&data);

root=insertNode(root, data);}

while (1) {

printf("\nBinary Search Tree Operations Menu\n");

printf("1. Insert\n");

printf("2. Delete\n");

printf("3. Search\n");

printf("4. Display\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

root = insertNode(root, data);

printf("%d inserted.\n", data);

break;

case 2:

printf("Enter data to delete: ");

scanf("%d", &data);

root = deleteNode(root, data);

printf("%d deleted.\n", data);

break;

case 3:

printf("Enter data to search: ");

scanf("%d", &data);

struct Node\* foundNode = searchNode(root, data);

if (foundNode != NULL)

printf("%d found in the tree.\n", data);

else

printf("%d not found in the tree.\n", data);

break;

case 4:

printf("In-order display of the BST: ");

inOrder(root);

printf("\n");

break;

case 5:

exit(0);

break;

default:

printf("Invalid choice! Please try again.\n");

}

}

return 0;

}

OUTPUT:-

Enter the no of elements to be inserted6

Enter elements100 90 110 80 95 105

Binary Search Tree Operations Menu

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 4

In-order display of the BST: 80 90 95 100 105 110

Binary Search Tree Operations Menu

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 2

Enter data to delete: 90

90 deleted.

Binary Search Tree Operations Menu

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 4

In-order display of the BST: 80 95 100 105 110

Binary Search Tree Operations Menu

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 3

Enter data to search: 80

80 found in the tree.

Binary Search Tree Operations Menu

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 5

|  |  |  |
| --- | --- | --- |
| Ex. No.:10 | **Implementation of AVL Tree** | Date:1.6.24 |

**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:**

CODE:-

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int key;

struct Node \*left;

struct Node \*right;

int height;

};

typedef struct Node node;

int height(node \*n)

{

if (n==NULL)

return 0;

return n->height;

}

node \*findmin(node \*tree)

{ if(tree==NULL)

return NULL;

else if(tree->left==NULL)

return tree;

else

return findmin(tree->left);

}

int max(int a,int b)

{ return (a>b)?a:b;

}

node \*rightrotate(node \*y)

{ node \*x=y->left;

node \*t2=x->right;

x->right=y;

y->left=t2;

y->height=1+max(height(y->left),height(y->right));

x->height=1+max(height(x->left),height(x->right));

return x;

}

node \*leftrotate(node \*x)

{ node \*y=x->right;

node \*t2=y->left;

y->left=x;

x->right=t2;

x->height=1+max(height(x->left),height(x->right));

y->height=1+max(height(y->left),height(y->right));

return y;

}

int getbalance(struct Node \*n)

{

if (n == NULL)

return 0;

return height(n->left) - height(n->right);

}

node \*insert(node \*tree,int k)

{ if(tree==NULL)

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

newnode->key=k;

newnode->left=NULL;

newnode->right=NULL;

newnode->height=1;

tree=newnode;

}

else if(k<tree->key)

tree->left=insert(tree->left,k);

else if(k>tree->key)

tree->right=insert(tree->right,k);

//else

//return tree;

tree->height=1+max(height(tree->left),height(tree->right));

int bal=getbalance(tree);

if(bal>1 && k<tree->left->key)

return rightrotate(tree);

if(bal<-1 && k>tree->right->key)

return leftrotate(tree);

if(bal>1 && k>tree->left->key)

{ tree->left=leftrotate(tree->left);

return rightrotate(tree); }

if(bal<-1 && k<tree->right->key)

{ tree->right=rightrotate(tree->right);

return leftrotate(tree); }

return tree;

}

node \*delete(node \*tree,int e)

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

if(e<tree->key)

tree->left=delete(tree->left,e);

else if(e>tree->key)

tree->right=delete(tree->right,e);

else if(tree->left && tree->right)

{ temp=findmin(tree->right);

tree->key=temp->key;

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

}

else

{ temp=tree;

if(tree->left==NULL)

tree=tree->right;

else if(tree->right==NULL)

tree=tree->left;

free(temp);

}

if (tree == NULL)

return tree;

tree->height = 1 + max(height(tree->left),

height(tree->right));

int balance = getbalance(tree);

// If this node becomes unbalanced,

// then there are 4 cases

// Left Left Case

if (balance > 1 &&

getbalance(tree->left) >= 0)

return rightrotate(tree);

// Left Right Case

if (balance > 1 &&

getbalance(tree->left) < 0)

{

tree->left = leftrotate(tree->left);

return rightrotate(tree);

}

// Right Right Case

if (balance < -1 &&

getbalance(tree->right) <= 0)

return leftrotate(tree);

// Right Left Case

if (balance < -1 &&

getbalance(tree->right) > 0)

{

tree->right = rightrotate(tree->right);

return leftrotate(tree);

}

return tree;

}

void inorder(node \*tree)

{ if(tree!=NULL)

{ //printf("%d ",tree->key);

inorder(tree->left);

printf("%d ",tree->key);

inorder(tree->right);}

}

int main()

{

node \*tree=NULL;

int n;

printf("ENTER TOT NO OF ELEMENTS");

scanf("%d",&n);

int e;

printf("ENETR ELEMENTS");

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

{ scanf("%d",&e);

tree=insert(tree,e);}

//inorder(tree);

printf("ENETR ELE TO BE DELETED");

scanf("%d",&e);

tree = delete(tree,e);

inorder(tree);

return 0;

}

OUTPUT:-

ENTER TOT NO OF ELEMENTS9

ENETR ELEMENTS9 5 10 0 6 11 -1 1 2

ENETR ELE TO BE DELETED10

-1 0 1 2 5 6 9 11

|  |  |  |
| --- | --- | --- |
| Ex. No.:11 | **Graph Traversal** | Date:1.6.24 |

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

**Algorithm:**

CODE:-

//BFS

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node \*next;

};

typedef struct Node node;

node \*create(int data) {

node \*N = malloc(sizeof(node));

N->data = data;

N->next = NULL;

return N;

}

struct Queue {

int ele;

struct Queue \*next;

};

typedef struct Queue q;

q \*f = NULL;

q \*r = NULL;

void enqueue(int ele) {

q \*newnode = malloc(sizeof(q));

newnode->ele = ele;

newnode->next = NULL;

if (f == NULL && r == NULL) {

f = r = newnode;

return;

}

r->next = newnode;

r = newnode;

}

int dequeue() {

if (f == NULL) {

return -1; // Return -1 if the queue is empty

}

q \*temp = f;

f = f->next;

int s = temp->ele;

free(temp);

if (f == NULL) {

r = NULL; // Update rear pointer if the queue becomes empty

}

return s;

}

void addedge(node \*adj[], int u, int v) {

node \*newnode = create(v);

newnode->next = adj[u];

adj[u] = newnode;

}

void bfs(node \*adj[], int si, int v) {

int visited[v];

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

visited[i] = 0;

}

enqueue(si);

visited[si] = 1;

while (f != NULL) {

int u = dequeue();

printf("%d ", u);

node \*temp = adj[u];

while (temp != NULL) {

int d = temp->data;

if (!visited[d]) {

visited[d] = 1;

enqueue(d);

}

temp = temp->next;

}

}

printf("\n");

}

int main() {

int vertices = 5;

// Adjacency list representation of the graph

node \*adjList[vertices];

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

adjList[i] = NULL;

// Add edges to the graph

addedge(adjList, 0, 1);

addedge(adjList, 0, 2);

addedge(adjList, 1, 3);

addedge(adjList, 1, 4);

addedge(adjList, 2, 4);

// Perform BFS traversal starting from vertex 0

printf("Breadth First Traversal starting from vertex 0: ");

bfs(adjList, 0, vertices);

return 0;}

OUTPUT FOR BFS:-

Breadth First Traversal starting from vertex 0: 0 2 1 4 3

CODE:-

//DFS

#include <stdio.h>

#include <stdlib.h>

// Node structure for adjacency list

struct Node {

int data;

struct Node \*next;

};

typedef struct Node node;

// Create a new node

node \*create(int data) {

node \*N = malloc(sizeof(node));

N->data = data;

N->next = NULL;

return N;

}

// Add edge to the graph

void addedge(node \*adj[], int u, int v) {

node \*newnode = create(v);

newnode->next = adj[u];

adj[u] = newnode;

}

// Recursive DFS function

void dfsUtil(node \*adj[], int v, int visited[]) {

visited[v] = 1;

printf("%d ", v);

node \*temp = adj[v];

while (temp != NULL) {

int d = temp->data;

if (!visited[d]) {

dfsUtil(adj, d, visited);

}

temp = temp->next;

}

}

// DFS traversal

void dfs(node \*adj[], int si, int vertices) {

int visited[vertices];

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

visited[i] = 0;

}

dfsUtil(adj, si, visited);

}

int main() {

int vertices = 5;

// Adjacency list representation of the graph

node \*adjList[vertices];

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

adjList[i] = NULL;

// Add edges to the graph

addedge(adjList, 0, 1);

addedge(adjList, 0, 2);

addedge(adjList, 1, 3);

addedge(adjList, 1, 4);

addedge(adjList, 2, 4);

// Perform DFS traversal starting from vertex 0

printf("Depth First Traversal starting from vertex 0: ");

dfs(adjList, 0, vertices);

return 0;

}

OUTPUT FOR DFS:-

Depth First Traversal starting from vertex 0: 0 2 4 1 3

|  |  |  |
| --- | --- | --- |
| Ex. No.:12 | **Topological Sorting** | Date:8.6.24 |

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

**Algorithm:**

CODE:-

// C Program to implement Topological Sorting

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

struct Stack {

int data;

struct Stack\* next;

};

struct Graph {

int V; // No. of vertices

// Pointer to an array containing adjacency lists

struct List\* adj;

};

// Structure to represent a list (adjacency list)

struct List {

int data;

struct List\* next;

};

// Create a new node for the stack

struct Stack\* createStackNode(int data)

{

struct Stack\* newNode

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Create a new node for the adjacency list

struct List\* createListNode(int data)

{

struct List\* newNode

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to initialize a graph with V vertices

struct Graph\* createGraph(int V)

{

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

graph->V = V;

graph->adj

= (struct List\*)malloc(V \* sizeof(struct List));

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

graph->adj[i].next = NULL;

}

return graph;

}

// Function to add an edge to the graph

void addEdge(struct Graph\* graph, int v, int w)

{

struct List\* newNode = createListNode(w);

newNode->next = graph->adj[v].next;

graph->adj[v].next = newNode;

}

// A recursive function used by topologicalSort

void topologicalSortUtil(struct Graph\* graph, int v,

bool visited[],

struct Stack\*\* stack)

{

visited[v] = true;

struct List\* current = graph->adj[v].next;

while (current != NULL) {

int adjacentVertex = current->data;

if (!visited[adjacentVertex]) {

topologicalSortUtil(graph, adjacentVertex,

visited, stack);

}

current = current->next;

}

// Push the current vertex to stack which stores the

// result

struct Stack\* newNode = createStackNode(v);

newNode->next = \*stack;

\*stack = newNode;

}

// The function to do Topological Sort. It uses recursive

// topologicalSortUtil

void topologicalSort(struct Graph\* graph)

{

struct Stack\* stack = NULL;

// Mark all the vertices as not visited

bool\* visited = (bool\*)malloc(graph->V \* sizeof(bool));

for (int i = 0; i < graph->V; ++i) {

visited[i] = false;

}

// Call the recursive helper function to store

// Topological Sort starting from all vertices one by

// one

for (int i = 0; i < graph->V; ++i) {

if (!visited[i]) { topologicalSortUtil(graph, i, visited, &stack); }

}

// Print contents of stack

while (stack != NULL) {

printf("%d ", stack->data);

struct Stack\* temp = stack;

stack = stack->next;

free(temp);

}

// Free allocated memory

free(visited);

free(graph->adj);

free(graph);

}

// Driver program to test above functions

int main()

{

// Create a graph given in the above diagram

struct Graph\* g = createGraph(6);

addEdge(g, 5, 2);

addEdge(g, 5, 0);

addEdge(g, 4, 0);

addEdge(g, 4, 1);

addEdge(g, 2, 3);

addEdge(g, 3, 1);

printf("Topological Sorting Order: ");

topologicalSort(g);

return 0;

}

OUTPUT:-

Topological Sorting Order:5 4 2 3 1 0

|  |  |  |
| --- | --- | --- |
| Ex. No.:13 | **Graph Traversal** | Date:8.6.24 |

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

**Algorithm:**

**CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <limits.h>**

**#include <stdbool.h>**

**#define V 5**

**struct Edge {**

**int src, dest, weight;**

**};**

**struct Graph {**

**int V, E;**

**struct Edge\* edge;**

**};**

**struct Graph\* createGraph(int V, int E) {**

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

**graph->V = V;**

**graph->E = E;**

**graph->edge = (struct Edge\*) malloc(graph->E \* sizeof(struct Edge));**

**return graph;**

**}**

**int minKey(int key[], bool mstSet[]) {**

**int min = INT\_MAX, min\_index;**

**for (int v = 0; v < V; v++)**

**if (mstSet[v] == false && key[v] < min)**

**min = key[v], min\_index = v;**

**return min\_index;**

**}**

**void printMST(int parent[], struct Graph\* graph) {**

**printf("Edge \tWeight\n");**

**for (int i = 1; i < graph->V; i++)**

**printf("%d - %d \t%d \n", parent[i], i, graph->edge[i].weight);**

**}**

**void primMST(struct Graph\* graph) {**

**int parent[V];**

**int key[V];**

**bool mstSet[V];**

**for (int i = 0; i < V; i++)**

**key[i] = INT\_MAX, mstSet[i] = false;**

**key[0] = 0;**

**parent[0] = -1;**

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

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

**mstSet[u] = true;**

**for (int v = 0; v < V; v++)**

**if (graph->edge[u].weight && mstSet[v] == false && graph->edge[u].weight < key[v])**

**parent[v] = u, key[v] = graph->edge[u].weight;**

**}**

**printMST(parent, graph);**

**}**

**int main() {**

**int V = 5;**

**int E = 7;**

**struct Graph\* graph = createGraph(V, E);**

**graph->edge[0].src = 0; graph->edge[0].dest = 1; graph->edge[0].weight = 2;**

**graph->edge[1].src = 0; graph->edge[1].dest = 3; graph->edge[1].weight = 6;**

**graph->edge[2].src = 1; graph->edge[2].dest = 2; graph->edge[2].weight = 3;**

**graph->edge[3].src = 1; graph->edge[3].dest = 3; graph->edge[3].weight = 8;**

**graph->edge[4].src = 1; graph->edge[4].dest = 4; graph->edge[4].weight = 5;**

**graph->edge[5].src = 2; graph->edge[5].dest = 4; graph->edge[5].weight = 7;**

**graph->edge[6].src = 3; graph->edge[6].dest = 4; graph->edge[6].weight = 9;**

**primMST(graph);**

**return 0;**

**}**

|  |  |  |
| --- | --- | --- |
| Ex. No.:14 | **Graph Traversal** | Date:8.6.24 |

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

**Algorithm:**

**CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <limits.h>**

**#include <stdbool.h>**

**#define V 5**

**int minDistance(int dist[], bool sptSet[]) {**

**int min = INT\_MAX, min\_index;**

**for (int v = 0; v < V; v++)**

**if (sptSet[v] == false && dist[v] <= min)**

**min = dist[v], min\_index = v;**

**return min\_index;**

**}**

**void printSolution(int dist[]) {**

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

**for (int i = 0; i < V; i++)**

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

**}**

**void dijkstra(int graph[V][V], int src) {**

**int dist[V];**

**bool sptSet[V];**

**for (int i = 0; i < V; i++)**

**dist[i] = INT\_MAX, sptSet[i] = false;**

**dist[src] = 0;**

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

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

**sptSet[u] = true;**

**for (int v = 0; v < V; 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);**

**}**

**int main() {**

**int graph[V][V] = {**

**{0, 4, 0, 0, 0},**

**{4, 0, 8, 0, 0},**

**{0, 8, 0, 7, 0},**

**{0, 0, 7, 0, 9},**

**{0, 0, 0, 9, 0}**

**};**

**dijkstra(graph, 0);**

**return 0;**

**}**

|  |  |  |
| --- | --- | --- |
| Ex. No.:15 | **Sorting** | Date:8.6.24 |

**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:**

**CODE:**

**1. QUICK SORT:**

**#include <stdio.h>**

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

**int t = \*a;**

**\*a = \*b;**

**\*b = t;**

**}**

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

**int pivot = arr[high];**

**int i = (low - 1);**

**for (int j = low; j <= high - 1; j++) {**

**if (arr[j] < pivot) {**

**i++;**

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

**}**

**}**

**swap(&arr[i + 1], &arr[high]);**

**return (i + 1);**

**}**

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

**if (low < high) {**

**int pi = partition(arr, low, high);**

**quickSort(arr, low, pi - 1);**

**quickSort(arr, pi + 1, high);**

**}**

**}**

**void printArray(int arr[], int size) {**

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

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

**printf("\n");**

**}**

**int main() {**

**int n;**

**printf("Enter the number of elements: ");**

**scanf("%d", &n);**

**int arr[n];**

**printf("Enter %d numbers:\n", n);**

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

**scanf("%d", &arr[i]);**

**}**

**quickSort(arr, 0, n - 1);**

**printf("Sorted array: \n");**

**printArray(arr, n);**

**return 0;**

**}**

**2. MERGE SORT**

#include <stdio.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 n;

printf("Enter the number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d numbers:\n", n);

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

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

}

mergeSort(arr, 0, n - 1);

printf("Sorted array: \n");

printArray(arr, n);

return 0;}

|  |  |  |
| --- | --- | --- |
| Ex. No.:16 | **Hashing** | Date:8.6.24 |

**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:**

**CODE:**

**i)Open Addressing:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**#define SIZE 10**

**struct HashNode {**

**int key;**

**int value;**

**bool occupied;**

**};**

**struct HashNode\* createHashNode(int key, int value) {**

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

**newNode->key = key;**

**newNode->value = value;**

**newNode->occupied = true;**

**return newNode;**

**}**

**struct HashNode\*\* createHashTable() {**

**struct HashNode\*\* table = (struct HashNode\*\*)malloc(SIZE \* sizeof(struct HashNode\*));**

**for (int i = 0; i < SIZE; i++) {**

**table[i] = NULL;**

**}**

**return table;**

**}**

**int hashCode(int key) {**

**return key % SIZE;**

**}**

**void insertOpenAddressing(struct HashNode\*\* table, int key, int value) {**

**int index = hashCode(key);**

**while (table[index] != NULL && table[index]->occupied) {**

**index = (index + 1) % SIZE;**

**}**

**table[index] = createHashNode(key, value);**

**}**

**void displayHashTable(struct HashNode\*\* table) {**

**printf("Hash Table:\n");**

**for (int i = 0; i < SIZE; i++) {**

**if (table[i] != NULL && table[i]->occupied) {**

**printf("Index %d: Key = %d, Value = %d\n", i, table[i]->key, table[i]->value);**

**} else {**

**printf("Index %d: Empty\n", i);**

**}**

**}**

**}**

**int main() {**

**struct HashNode\*\* hashTable = createHashTable();**

**insertOpenAddressing(hashTable, 5, 10);**

**insertOpenAddressing(hashTable, 15, 20);**

**insertOpenAddressing(hashTable, 25, 30);**

**insertOpenAddressing(hashTable, 35, 40);**

**insertOpenAddressing(hashTable, 45, 50);**

**insertOpenAddressing(hashTable, 55, 60);**

**insertOpenAddressing(hashTable, 65, 70);**

**insertOpenAddressing(hashTable, 75, 80);**

**insertOpenAddressing(hashTable, 85, 90);**

**insertOpenAddressing(hashTable, 95, 100);**

**displayHashTable(hashTable);**

**return 0;**

**}**

**2. Closed Addressing:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define SIZE 10**

**struct HashNode {**

**int key;**

**int value;**

**struct HashNode\* next;**

**};**

**struct HashNode\* createHashNode(int key, int value) {**

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

**newNode->key = key;**

**newNode->value = value;**

**newNode->next = NULL;**

**return newNode;**

**}**

**struct HashNode\*\* createHashTable() {**

**struct HashNode\*\* table = (struct HashNode\*\*)malloc(SIZE \* sizeof(struct HashNode\*));**

**for (int i = 0; i < SIZE; i++) {**

**table[i] = NULL;**

**}**

**return table;**

**}**

**int hashCode(int key) {**

**return key % SIZE;**

**}**

**void insertClosedAddressing(struct HashNode\*\* table, int key, int value) {**

**int index = hashCode(key);**

**if (table[index] == NULL) {**

**table[index] = createHashNode(key, value);**

**} else {**

**struct HashNode\* newNode = createHashNode(key, value);**

**newNode->next = table[index];**

**table[index] = newNode;**

**}**

**}**

**void displayHashTable(struct HashNode\*\* table) {**

**printf("Hash Table:\n");**

**for (int i = 0; i < SIZE; i++) {**

**struct HashNode\* current = table[i];**

**printf("Index %d: ", i);**

**while (current != NULL) {**

**printf("Key = %d, Value = %d -> ", current->key, current->value);**

**current = current->next;**

**}**

**printf("NULL\n");**

**}**

**}**

**int main() {**

**struct HashNode\*\* hashTable = createHashTable();**

**insertClosedAddressing(hashTable, 5, 10);**

**insertClosedAddressing(hashTable, 15, 20);**

**insertClosedAddressing(hashTable, 25, 30);**

**insertClosedAddressing(hashTable, 35, 40);**

**insertClosedAddressing(hashTable, 45, 50);**

**insertClosedAddressing(hashTable, 55, 60);**

**insertClosedAddressing(hashTable, 65, 70);**

**insertClosedAddressing(hashTable, 75, 80);**

**insertClosedAddressing(hashTable, 85, 90);**

**insertClosedAddressing(hashTable, 95, 100);**

**displayHashTable(hashTable);**

**return 0;**

**}**

**3. Rehashing:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**#define SIZE 10**

**#define LOAD\_FACTOR\_THRESHOLD 0.7**

**struct HashNode {**

**int key;**

**int value;**

**bool occupied;**

**};**

**struct HashNode\* createHashNode(int key, int value) {**

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

**newNode->key = key;**

**newNode->value = value;**

**newNode->occupied = true;**

**return newNode;**

**}**

**struct HashNode\*\* createHashTable(int size) {**

**struct HashNode\*\* table = (struct HashNode\*\*)malloc(size \* sizeof(struct HashNode\*));**

**for (int i = 0; i < size; i++) {**

**table[i] = NULL;**

**}**

**return table;**

**}**

**int hashCode(int key, int size) {**

**return key % size;**

**}**

**void insertOpenAddressing(struct HashNode\*\* table, int key**

**, int value, int size) {**

**int index = hashCode(key, size);**

**while (table[index] != NULL && table[index]->occupied) {**

**index = (index + 1) % size;**

**}**

**table[index] = createHashNode(key, value);**

**}**

**void displayHashTable(struct HashNode\*\* table, int size) {**

**printf("Hash Table:\n");**

**for (int i = 0; i < size; i++) {**

**if (table[i] != NULL && table[i]->occupied) {**

**printf("Index %d: Key = %d, Value = %d\n", i, table[i]->key, table[i]->value);**

**} else {**

**printf("Index %d: Empty\n", i);**

**}**

**}**

**}**

**double getLoadFactor(int count, int size) {**

**return (double)count / size;**

**}**

**void rehash(struct HashNode\*\* oldTable, int oldSize, struct HashNode\*\* newTable, int newSize) {**

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

**if (oldTable[i] != NULL && oldTable[i]->occupied) {**

**int index = oldTable[i]->key % newSize;**

**while (newTable[index] != NULL && newTable[index]->occupied) {**

**index = (index + 1) % newSize;**

**}**

**newTable[index] = oldTable[i];**

**}**

**}**

**free(oldTable);**

**}**

**int main() {**

**int initialSize = SIZE;**

**struct HashNode\*\* hashTable = createHashTable(initialSize);**

**int count = 0;**

**insertOpenAddressing(hashTable, 5, 10, initialSize);**

**insertOpenAddressing(hashTable, 15, 20, initialSize);**

**insertOpenAddressing(hashTable, 25, 30, initialSize);**

**insertOpenAddressing(hashTable, 35, 40, initialSize);**

**insertOpenAddressing(hashTable, 45, 50, initialSize);**

**insertOpenAddressing(hashTable, 55, 60, initialSize);**

**insertOpenAddressing(hashTable, 65, 70, initialSize);**

**insertOpenAddressing(hashTable, 75, 80, initialSize);**

**insertOpenAddressing(hashTable, 85, 90, initialSize);**

**insertOpenAddressing(hashTable, 95, 100, initialSize);**

**count = 10;**

**double loadFactor = getLoadFactor(count, initialSize);**

**if (loadFactor > LOAD\_FACTOR\_THRESHOLD) {**

**int newSize = initialSize \* 2; // Double the size**

**struct HashNode\*\* newTable = createHashTable(newSize);**

**rehash(hashTable, initialSize, newTable, newSize);**

**hashTable = newTable;**

**initialSize = newSize;**

**}**

**displayHashTable(hashTable, initialSize);**

**return 0;**

**}**