Write a Program to Create a structure called "Student" with member's name, age, and total marks. Also find the input data for two students, display their information, and find the average of total marks

Source code:

```
#include <stdio.h>

struct Student {

   char name[50];

   int age;

   float total_marks;

};

int main() {

   struct Student student1, student2;

   float average_marks;

   printf("Enter the name, age, and total marks of the first student: ");

   scanf("%s %d %f", student1.name, &student1.age, &student1.total_marks);

   printf("Enter the name, age, and total marks of the second student: ");

   scanf("%s %d %f", student2.name, &student2.age, &student2.total_marks);

   average_marks = (student1.total_marks + student2.total_marks) / 2;

   printf("\n--- Student Information ---\n");

   printf("Student 1: %s, Age: %d, Total Marks: %.2f\n", student1.name, student1.age,
student1.total_marks);

   printf("Student 2: %s, Age: %d, Total Marks: %.2f\n", student2.name, student2.age,
student2.total_marks);

   printf("Average Total Marks: %.2f\n", average_marks);

   return 0;

}
```

Output:

Enter the name, age, and total marks of the first student: John 20 85.5

Enter the name, age, and total marks of the second student: Alice 19 90.0

--- Student Information ---

Student 1: John, Age: 20, Total Marks: 85.50

Student 2: Alice, Age: 19, Total Marks: 90.00

Average Total Marks: 87.75

Create a structure named "Employee" to store employee details such as employee ID, name, and salary. Write a program to input data for three employees, find the highest salary employee, and display their information.

Source code:

```
#include <stdio.h>

struct Employee {

    int emp_id;

    char name[50];

    float salary;

};

int main() {

    struct Employee emp[3];

    int i, max_index = 0;

    for (i = 0; i < 3; i++) {

        printf("Enter Employee ID, Name, and Salary for Employee %d: ", i + 1);

        scanf("%d %s %f", &emp[i].emp_id, emp[i].name, &emp[i].salary);

        if (emp[i].salary > emp[max_index].salary)

            max_index = i;

    }

    printf("\nHighest Salary Employee:\n");

    printf("ID: %d, Name: %s, Salary: %.2f\n", emp[max_index].emp_id, emp[max_index].name,
emp[max_index].salary);

    return 0;

}
```

Output:

Enter Employee ID, Name, and Salary for Employee 1: 101 John 50000

Enter Employee ID, Name, and Salary for Employee 2: 102 Alice 60000

Enter Employee ID, Name, and Salary for Employee 3: 103 Bob 55000

Highest Salary Employee:

ID: 102, Name: Alice, Salary: 60000.00

Write a program in C to find the largest element using Dynamic Memory Allocation.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int n, i;
   float *arr, max;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   arr = (float *)malloc(n * sizeof(float));
   if (arr == NULL) {
      printf("Memory allocation failed!");
      return 1;
   }
   printf("Enter the elements:\n");
   for (i = 0; i < n; i++) {
      scanf("%f", &arr[i]);
   }
   max = arr[0];
   for (i = 1; i < n; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
   }
   printf("Largest element: %.2f\n", max);
   free(arr);
   return 0;
}
```

Output:

Enter the number of elements: 5

Enter the elements:

12.5 7.8 25.4 3.6 18.9

Largest element: 25.40

Write a program in C to multiply 2 Matrices using Dynamic Memory Allocation

Source code:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int r1, c1, r2, c2, i, j, k;
    printf("Enter rows and columns for first matrix: ");
    scanf("%d%d", &r1, &c1);
    printf("Enter rows and columns for second matrix: ");
    scanf("%d%d", &r2, &c2);
    if (c1 != r2) {
        printf("Matrix multiplication not possible.\n");
        return 1;
    }
    int **mat1 = (int **)malloc(r1 * sizeof(int *));
    int **mat2 = (int **)malloc(r2 * sizeof(int *));
    int **result = (int **)malloc(r1 * sizeof(int *));
    for (i = 0; i < r1; i++) mat1[i] = (int *)malloc(c1 * sizeof(int));
    for (i = 0; i < r2; i++) mat2[i] = (int *)malloc(c2 * sizeof(int));
    for (i = 0; i < r1; i++) result[i] = (int *)malloc(c2 * sizeof(int));
    printf("Enter elements of first matrix:\n");
    for (i = 0; i < r1; i++)
        for (j = 0; j < c1; j++)
            scanf("%d", &mat1[i][j]);
    printf("Enter elements of second matrix:\n");
    for (i = 0; i < r2; i++)
        for (j = 0; j < c2; j++)
            scanf("%d", &mat2[i][j]);
    for (i = 0; i < r1; i++)
        for (j = 0; j < c2; j++) {
```

```
            result[i][j] = 0;
            for (k = 0; k < c1; k++)
                result[i][j] += mat1[i][k] * mat2[k][j];
        }
    printf("Resultant matrix:\n");
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c2; j++)
            printf("%d ", result[i][j]);
        printf("\n");
    }
    for (i = 0; i < r1; i++) free(mat1[i]);
    for (i = 0; i < r2; i++) free(mat2[i]);
    for (i = 0; i < r1; i++) free(result[i]);
    free(mat1);
    free(mat2);
    free(result);
    return 0;
}
```

Output:

Enter rows and columns for first matrix: 2 3

Enter rows and columns for second matrix: 3 2

Enter elements of first matrix:

1 2 3

4 5 6

Enter elements of second matrix:

7 8

9 10

11 12

Resultant matrix:

58 64

139 154

Write a C Program to Create a Singly Linked List and perform the following Operations. a. Insert at Beginning b. Insert at Last

Source code:

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node *next;

};


void insertAtBeginning(struct Node **head, int value) {

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

   newNode->data = value;

   newNode->next = *head;

   *head = newNode;

}


void insertAtEnd(struct Node **head, int value) {

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

   newNode->data = value;

   newNode->next = NULL;

   if (*head == NULL) {

      *head = newNode;

      return;

   }

   struct Node *temp = *head;

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

   temp->next = newNode;

}
```

```
void displayList(struct Node *head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node *head = NULL;
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);
    printf("Linked List: ");
    displayList(head);
    return 0;
}
```

Output:

Linked List: 20 -> 10 -> 30 -> 40 -> NULL

Write a C Program to Create a Singly Linked List and perform the following Operations. A) Insert at any random location b) search for an element

Source code:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next;
};
```

```
void insertAtPosition(struct Node **head, int value, int position) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        return;
    }
    struct Node *temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++)
        temp = temp->next;
    if (temp == NULL) {
        printf("Position out of range.\n");
        free(newNode);
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

int search(struct Node *head, int key) {
    int position = 1;
    while (head != NULL) {
        if (head->data == key) return position;
        head = head->next;
        position++;
    }
    return -1;
}

void displayList(struct Node *head) {
```

```
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node *head = NULL;
    insertAtPosition(&head, 10, 1);
    insertAtPosition(&head, 20, 2);
    insertAtPosition(&head, 30, 3);
    insertAtPosition(&head, 15, 2); // Insert 15 at position 2
    printf("Linked List: ");
    displayList(head);

    int key = 15;
    int pos = search(head, key);
    if (pos != -1) printf("Element %d found at position %d\n", key, pos);
    else printf("Element %d not found in the list.\n", key);

    return 0;
}
```

Output:

Linked List: 10 -> 15 -> 20 -> 30 -> NULL

Element 15 found at position 2

Write a C Program to Create a Singly Linked List and perform the following Operations. a. Delete node after specified location . b. Display the List

Source code:

```
#include <stdio.h>
#include <stdlib.h>
```

```
struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void insert(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

void deleteAfter(int position) {
    if (head == NULL || position < 0) return;
    struct Node* current = head;
    for (int i = 0; current != NULL && i < position; i++) {
        current = current->next;
    }
    if (current == NULL || current->next == NULL) return;
    struct Node* temp = current->next;
    current->next = temp->next;
    free(temp);
}

void display() {
    struct Node* temp = head;
    while (temp) {
        printf("%d -> ", temp->data);
```

```
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    insert(10);
    insert(20);
    insert(30);
    printf("Original List: ");
    display();
    deleteAfter(1);
    printf("After Deletion after Position 1: ");
    display();
    return 0;
}
```

Output

Original List: 30 -> 20 -> 10 -> NULL

After Deletion after Position 1: 30 -> 10 -> NULL

Write a C Program to Create a Doubly Linked List and perform the following Operations. a. Insert at Beginning b. Insert at Last

Source code:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

```c
struct Node* head = NULL;

void insertAtBeginning(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = head;
    newNode->prev = NULL;
    if (head != NULL) head->prev = newNode;
    head = newNode;
}

void insertAtLast(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void display() {
    struct Node* temp = head;
    while (temp) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
```

```c
    }
    printf("NULL\n");
}

int main() {
    insertAtBeginning(10);
    insertAtBeginning(20);
    insertAtLast(30);
    insertAtLast(40);
    printf("Doubly Linked List: ");
    display();
    return 0;
}
```

Output

Doubly Linked List: 20 <-> 10 <-> 30 <-> 40 <-> NULL

Write a C Program to Create a Doubly Linked List and perform the following Operations. A) Insert at any random location b) search for an element

Source code

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

struct Node* head = NULL;

void insertAtPosition(int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;

    if (position == 0) {
        newNode->next = head;
        if (head != NULL) head->prev = newNode;
        head = newNode;
        return;
    }

    struct Node* temp = head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds\n");
        free(newNode);
        return;
    }

    newNode->next = temp->next;
    if (temp->next != NULL) temp->next->prev = newNode;
    temp->next = newNode;
    newNode->prev = temp;
}

void search(int key) {
    struct Node* temp = head;
    int position = 0;
```

```
    while (temp != NULL) {
        if (temp->data == key) {
            printf("Element %d found at position %d\n", key, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Element %d not found in the list\n", key);
}

void display() {
    struct Node* temp = head;
    while (temp) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    insertAtPosition(10, 0);
    insertAtPosition(20, 1);
    insertAtPosition(30, 1);
    insertAtPosition(40, 3);
    printf("Doubly Linked List: ");
    display();

    search(20);
    search(50);
```

```
    return 0;
}
```

Output

Doubly Linked List: 10 <-> 30 <-> 20 <-> 40 <-> NULL

Element 20 found at position 2

Element 50 not found in the list

Write a C Program to Create a Doubly Linked List and perform the following Operations. A. Delete from Beginning b. Delete from Last

Source code

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

struct Node* head = NULL;

void insertAtEnd(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;

    if (head == NULL) {
        head = newNode;
        return;
    }
```

```
    struct Node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void deleteFromBeginning() {
    if (head == NULL) return;

    struct Node* temp = head;
    head = head->next;

    if (head != NULL) head->prev = NULL;
    free(temp);
}

void deleteFromEnd() {
    if (head == NULL) return;

    struct Node* temp = head;
    while (temp->next != NULL) temp = temp->next;

    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    } else {
        head = NULL;
    }
    free(temp);
}

void display() {
```

```
    struct Node* temp = head;
    while (temp) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtEnd(30);
    printf("Original List: ");
    display();

    deleteFromBeginning();
    printf("After Deletion from Beginning: ");
    display();

    deleteFromEnd();
    printf("After Deletion from End: ");
    display();

    return 0;
}
```

Output:

```
Original List: 10 <-> 20 <-> 30 <-> NULL
After Deletion from Beginning: 20 <-> 30 <-> NULL
After Deletion from End: 20 <-> NULL
```

Write a C Program to Create a Doubly Linked List and perform the following Operations. a. Delete node after specified location . b. Display the List

Source code

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a doubly linked list node
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};

// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   newNode->prev = NULL;
   return newNode;
}

// Function to insert a node at the end of the list
void insertEnd(struct Node** head_ref, int data) {
   struct Node* newNode = createNode(data);
   if (*head_ref == NULL) {
      *head_ref = newNode;
      return;
   }
   struct Node* temp = *head_ref;
   while (temp->next != NULL) {
      temp = temp->next;
   }
```

```
    temp->next = newNode;
    newNode->prev = temp;
}

// Function to delete a node after a specified location
void deleteAfter(struct Node** head_ref, int position) {
    if (*head_ref == NULL || position < 0) {
        printf("List is empty or invalid position.\n");
        return;
    }
    struct Node* temp = *head_ref;
    int count = 0;

    // Traverse to the specified position
    while (temp != NULL && count < position) {
        temp = temp->next;
        count++;
    }

    // If the position is valid and has a next node to delete
    if (temp != NULL && temp->next != NULL) {
        struct Node* nodeToDelete = temp->next;
        temp->next = nodeToDelete->next;
        if (nodeToDelete->next != NULL) {
            nodeToDelete->next->prev = temp;
        }
        free(nodeToDelete);
        printf("Node deleted after position %d.\n", position);
    } else {
        printf("No node to delete after position %d.\n", position);
    }
```

```
}

// Function to display the list
void displayList(struct Node* node) {
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    // Insert some nodes
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    insertEnd(&head, 40);
    insertEnd(&head, 50);

    printf("Original list: ");
    displayList(head);

    // Delete node after specified position
    int position = 2; // Delete node after position 2 (which is value 30)
    deleteAfter(&head, position);

    printf("List after deletion: ");
    displayList(head);
```

```
    return 0;
}
```

Output

Original list: 10 20 30 40 50

Node deleted after position 2.

List after deletion: 10 20 40 50

Write a C Program to Implement a stack using Array and perform the following operations a. Push an Element on to Stack b. Pop an Element from Stack c. Display the Stack

Source code

```
#include <stdio.h>
#define MAX 100
int stack[MAX], top = -1;

void push(int value) {
    if (top >= MAX - 1) printf("Stack Overflow\n");
    else stack[++top] = value;
}

int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}

void display() {
    if (top < 0) printf("Stack is empty\n");
    else {
        for (int i = top; i >= 0; i--) printf("%d ", stack[i]);
```

```
        printf("\n");
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    printf("Stack after pushes: ");
    display();
    printf("Popped element: %d\n", pop());
    printf("Stack after pop: ");
    display();
    return 0;
}
```

Output

Stack after pushes: 30 20 10

Popped element: 30

Stack after pop: 20 10

Write a C Program to Implement a stack using Linked List and perform the following operations a. Push an Element on to Stack b. Pop an Element from Stack c. Display the Stack

Source code

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Stack {
```

```
    struct Node* top;
};

void initStack(struct Stack* stack) {
    stack->top = NULL;
}

int isEmpty(struct Stack* stack) {
    return stack->top == NULL;
}

void push(struct Stack* stack, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = stack->top;
    stack->top = newNode;
}

int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }
    struct Node* temp = stack->top;
    int poppedValue = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return poppedValue;
}

void display(struct Stack* stack) {
```

```
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return;
    }
    struct Node* temp = stack->top;
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Stack stack;
    initStack(&stack);
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    printf("Stack after pushes: ");
    display(&stack);
    printf("Popped element: %d\n", pop(&stack));
    printf("Stack after pop: ");
    display(&stack);
    return 0;
}
```

Output:

Stack after pushes: 30 20 10

Popped element: 30

Stack after pop: 20 10

Write a C Program to implement a Binary Search Tree (BST) and perform following operations a. Traverse the BST in Inorder b. Traverse the BST in Preorder c. Traverse the BST in and Post Order

Source code

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct Node* insert(struct Node* root, int data) {
   if (root == NULL) return createNode(data);
   if (data < root->data)
      root->left = insert(root->left, data);
   else
      root->right = insert(root->right, data);
   return root;
}

void inorder(struct Node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
```

```
    }
}

void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal: ");
    inorder(root);
```

```
    printf("\n");

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Output

Inorder traversal: 20 30 40 50 60 70 80

Preorder traversal: 50 30 20 40 70 60 80

Postorder traversal: 20 40 30 60 80 70 50

Implement binary tree

Source code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 3 // Maximum degree (number of children)

struct BTreeNode {
    int keys[MAX - 1];
    struct BTreeNode* children[MAX];
    int numKeys;
    int isLeaf;
};

struct BTreeNode* createNode(int isLeaf) {
```

```c
    struct BTreeNode* newNode = (struct BTreeNode*)malloc(sizeof(struct BTreeNode));
    newNode->isLeaf = isLeaf;
    newNode->numKeys = 0;
    for (int i = 0; i < MAX; i++) newNode->children[i] = NULL;
    return newNode;
}

void traverse(struct BTreeNode* node) {
    if (node != NULL) {
        for (int i = 0; i < node->numKeys; i++) {
            if (!node->isLeaf) traverse(node->children[i]);
            printf("%d ", node->keys[i]);
        }
        if (!node->isLeaf) traverse(node->children[node->numKeys]);
    }
}

void splitChild(struct BTreeNode* parent, int index, struct BTreeNode* child) {
    struct BTreeNode* newChild = createNode(child->isLeaf);
    newChild->numKeys = MAX / 2 - 1;
    for (int i = 0; i < newChild->numKeys; i++) newChild->keys[i] = child->keys[i + MAX / 2];
    if (!child->isLeaf) {
        for (int i = 0; i <= newChild->numKeys; i++) newChild->children[i] = child->children[i + MAX / 2];
    }
    child->numKeys = MAX / 2 - 1;
    for (int i = parent->numKeys; i >= index + 1; i--) parent->children[i + 1] = parent->children[i];
    parent->children[index + 1] = newChild;
    for (int i = parent->numKeys - 1; i >= index; i--) parent->keys[i + 1] = parent->keys[i];
    parent->keys[index] = child->keys[MAX / 2 - 1];
    parent->numKeys++;
}
```

```
void insertNonFull(struct BTreeNode* node, int key) {
   int i = node->numKeys - 1;
   if (node->isLeaf) {
      while (i >= 0 && key < node->keys[i]) {
         node->keys[i + 1] = node->keys[i];
         i--;
      }
      node->keys[i + 1] = key;
      node->numKeys++;
   } else {
      while (i >= 0 && key < node->keys[i]) i--;
      i++;
      if (node->children[i]->numKeys == MAX - 1) {
         splitChild(node, i, node->children[i]);
         if (key > node->keys[i]) i++;
      }
      insertNonFull(node->children[i], key);
   }
}

struct BTreeNode* insert(struct BTreeNode* root, int key) {
   if (root == NULL) {
      struct BTreeNode* newNode = createNode(1);
      newNode->keys[0] = key;
      newNode->numKeys = 1;
      return newNode;
   }
   if (root->numKeys == MAX - 1) {
      struct BTreeNode* newRoot = createNode(0);
      newRoot->children[0] = root;
```

```
        splitChild(newRoot, 0, root);
        insertNonFull(newRoot, key);
        return newRoot;
    } else {
        insertNonFull(root, key);
        return root;
    }
}

int main() {
    struct BTreeNode* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 12);
    root = insert(root, 30);
    root = insert(root, 40);

    printf("Traversal of the B-Tree: ");
    traverse(root);
    printf("\n");

    return 0;
}
```

Output:

Traversal of the B-Tree: 5 6 10 12 20 30 40

Write a C Program for Creating a Graph and Print all the nodes reachable from a given starting node using BFS method

Source code

```
#include <stdio.h>
```

```
#include <stdlib.h>

#define MAX 100

struct Node {
   int vertex;
   struct Node* next;
};

struct Graph {
   struct Node* adjLists[MAX];
   int visited[MAX];
   int numVertices;
};

struct Node* createNode(int v) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

void initGraph(struct Graph* g, int vertices) {
   g->numVertices = vertices;
   for (int i = 0; i < vertices; i++) {
      g->adjLists[i] = NULL;
      g->visited[i] = 0;
   }
}

void addEdge(struct Graph* g, int src, int dest) {
```

```
    struct Node* newNode = createNode(dest);
    newNode->next = g->adjLists[src];
    g->adjLists[src] = newNode;

    newNode = createNode(src); // For undirected graph
    newNode->next = g->adjLists[dest];
    g->adjLists[dest] = newNode;
}

void bfs(struct Graph* g, int startVertex) {
    int queue[MAX], front = -1, rear = -1;
    g->visited[startVertex] = 1;
    queue[++rear] = startVertex;

    printf("BFS Traversal starting from vertex %d: ", startVertex);
    while (front < rear) {
        front++;
        int currentVertex = queue[front];
        printf("%d ", currentVertex);

        struct Node* temp = g->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!g->visited[adjVertex]) {
                g->visited[adjVertex] = 1;
                queue[++rear] = adjVertex;
            }
            temp = temp->next;
        }
    }
    printf("\n");
```

```
}

int main() {
   struct Graph g;
   initGraph(&g, 6); // Create a graph with 6 vertices

   addEdge(&g, 0, 1);
   addEdge(&g, 0, 2);
   addEdge(&g, 1, 3);
   addEdge(&g, 1, 4);
   addEdge(&g, 2, 5);

   bfs(&g, 0); // Start BFS from vertex 0

   return 0;
}
```

Output

BFS Traversal starting from vertex 0: 0 2 1 5 3 4

Write a C Program for Creating a Graph and Print all the nodes reachable from a given starting node using DFS method

Source code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

struct Node {
   int vertex;
   struct Node* next;
};
```

```
struct Graph {
    struct Node* adjLists[MAX];
    int visited[MAX];
    int numVertices;
};

struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

void initGraph(struct Graph* g, int vertices) {
    g->numVertices = vertices;
    for (int i = 0; i < vertices; i++) {
        g->adjLists[i] = NULL;
        g->visited[i] = 0;
    }
}

void addEdge(struct Graph* g, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = g->adjLists[src];
    g->adjLists[src] = newNode;

    newNode = createNode(src); // For undirected graph
    newNode->next = g->adjLists[dest];
    g->adjLists[dest] = newNode;
}
```

```
void dfs(struct Graph* g, int vertex) {
   g->visited[vertex] = 1;
   printf("%d ", vertex);

   struct Node* temp = g->adjLists[vertex];
   while (temp) {
      int adjVertex = temp->vertex;
      if (!g->visited[adjVertex]) {
         dfs(g, adjVertex);
      }
      temp = temp->next;
   }
}

int main() {
   struct Graph g;
   initGraph(&g, 6); // Create a graph with 6 vertices

   addEdge(&g, 0, 1);
   addEdge(&g, 0, 2);
   addEdge(&g, 1, 3);
   addEdge(&g, 1, 4);
   addEdge(&g, 2, 5);

   printf("DFS Traversal starting from vertex 0: ");
   dfs(&g, 0); // Start DFS from vertex 0
   printf("\n");

   return 0;
}
```

Output

DFS Traversal starting from vertex 0: 0 1 3 4 2 5