Exercise 1: Array Manipulation

Write a program to reverse an array.

```
#include <stdio.h>

void reverseArray(int arr[], int start, int end) {
while (start < end) {
int temp = arr[start];
arr[start] = arr[end];
arr[end] = temp;
start++;
end--;
}
}

int main() {
int n;
int arr[100];
// Ask the user for the number of elements
printf("Enter the number of elements in the array: ");
scanf("%d", &n);

// Input the elements of the array
printf("Enter the elements of the array:\n");
for(int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}

// Reverse the array
reverseArray(arr, 0, n-1);

// Print the reversed array
printf("Reversed array:\n");
for(int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n");
```

```
return 0;
    }
```

C Programs to implement the Searching Techniques – Linear & Binary Search

```
#include <stdio.h>

int linearSearch(int arr[], int size, int value) {
for (int i = 0; i < size; i++) {
if (arr[i] == value) {
return i; // Return the index of the found value
}
}
return -1; // Value not found
}

int main() {
int n, value, position;

// Prompt the user to enter the number of elements in the array
printf("Enter the number of elements in the array: ");
scanf("%d", &n);
int arr[n]; // Declare an array of size n
// Read the elements of the array
printf("Enter the elements of the array:\n");
for (int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}

// Prompt the user to enter the value to search for
printf("Enter the value to search for: ");
scanf("%d", &value);

// Perform the linear search
position = linearSearch(arr, n, value);

if (position == -1) {
```

```
 printf("Value not found in the array.\n");
 } else {
 printf("Value found at position: %d\n", position + 1); // Position is index+1 for user-friendly
     output
 }
 return 0;

}
```

Binary Search

```
 #include <stdio.h>
 // Function to perform binary search on a sorted array
 int binarySearch(int arr[], int l, int r, int x) {
 while (l <= r) {
 int m = l + (r - l) / 2;

 // Check if x is present at mid
 if (arr[m] == x)
 return m;

 // If x greater, ignore left half
 if (arr[m] < x)
 l = m + 1;
 // If x is smaller, ignore right half
 else
 r = m - 1;
 }
 // If the element is not present in array
 return -1;
 }

 int main() {
 int arr[50], n, x, result;
 // User input for array size
 printf("Enter number of elements in the array: ");
 scanf("%d", &n);
```

```
// Check for valid array size
if (n <= 0 || n > 50) {
printf("Invalid array size. Please enter a value between 1 and 50.\n");
return 1; // Exit with error
}

printf("Enter %d integers in ascending order: ", n);
for (int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}

// User input for the search value
printf("Enter the value to search: ");
scanf("%d", &x);

// Function call to search x in arr[]
result = binarySearch(arr, 0, n - 1, x);

// Output result
if (result == -1)
printf("Element is not present in the array.");
else
printf("Element is present at index %d.", result);

return 0;
    }
```

C Programs to implement Sorting Techniques – Bubble, Selection, and Insertion Sort

Bubble Sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
int i, j, temp;
for (i = 0; i < n-1; i++)
// Last i elements are already in place
for (j = 0; j < n-i-1; j++)
```

```
 if (arr[j] > arr[j+1]) {
 // swap temp and arr[i]
 temp = arr[j];
 arr[j] = arr[j+1];
 arr[j+1] = temp;
 }
 }

 int main() {
 int array[100], n, i;
 printf("Enter number of elements: ");
 scanf("%d", &n);
 printf("Enter %d integers: ", n);
 for(i = 0; i < n; i++)
 scanf("%d", &array[i]);
 bubbleSort(array, n);
 printf("Sorted array: \n");
 for(i = 0; i < n; i++)
 printf("%d ", array[i]);
 printf("\n");
 return 0;
     }
```

Selection Sort

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
int i, j, min_idx, temp;

// One by one move the boundary of the unsorted subarray
for (i = 0; i < n-1; i++) {
// Find the minimum element in the unsorted array
min_idx = i;
for (j = i+1; j < n; j++) {
if (arr[j] < arr[min_idx])
min_idx = j;
}
```

```
// Swap the found minimum element with the first element
temp = arr[min_idx];
arr[min_idx] = arr[i];
arr[i] = temp;
}
}

int main() {
int arr[100], n, i;

// Input number of elements
printf("Enter number of elements in the array: ");
scanf("%d", &n);

// Input array elements
printf("Enter %d integers:\n", n);
for (i = 0; i < n; i++)
scanf("%d", &arr[i]);

// Call the selectionSort function
selectionSort(arr, n);

// Print the sorted array
printf("Sorted array in ascending order:\n");
for (i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");

return 0;
}
```

Insertion Sort

```
#include <stdio.h>

void insertionSort(int array[], int size) {
```

```
int i, key, j;
for (i = 1; i < size; i++) {
key = array[i];
j = i - 1;

// Move elements of array[0..i-1], that are greater than key,
// to one position ahead of their current position
while (j >= 0 && array[j] > key) {
array[j + 1] = array[j];
j = j - 1;
}
array[j + 1] = key;
}
}

int main() {
int n, i;

// User input for the number of elements
printf("Enter number of elements: ");
scanf("%d", &n);
int arr[n];

// User input for array elements
printf("Enter %d integers: ", n);
for (i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}

// Sorting the array
insertionSort(arr, n);

// Printing the sorted array
printf("Sorted array: \n");
for (i = 0; i < n; i++) {
printf("%d ", arr[i]);
```

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

return 0;
}
```

Exercise 2: Linked List Implementation
Implement a singly linked list and perform insertion and deletion operations.

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

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

// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(struct Node** head_ref, int new_data) {
// Allocate memory for new node
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
// Assign data to new node
new_node->data = new_data;
// Link the new node to the head of the list
new_node->next = *head_ref;
// Move the head to point to the new node
*head_ref = new_node;
printf("Inserted %d at the beginning\n", new_data);
}

// Function to delete a node with given key from the linked list
void deleteNode(struct Node** head_ref, int key) {
// Store head node
struct Node* temp = *head_ref, *prev;
```

```
// If head node itself holds the key to be deleted
if (temp != NULL && temp->data == key) {
*head_ref = temp->next; // Changed head
free(temp); // Free old head
printf("Deleted %d\n", key);
return;
}

// Search for the key to be deleted, keep track of the previous node as we need to change 'prev->next'
while (temp != NULL && temp->data != key) {
prev = temp;
temp = temp->next;
}

// If key was not present in linked list
if (temp == NULL) {
printf("Key %d not found\n", key);
return;
}

// Unlink the node from linked list
prev->next = temp->next;
free(temp); // Free memory
printf("Deleted %d\n", key);
}

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

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

// Insert some nodes
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 7);
insertAtBeginning(&head, 9);

// Print the linked list
printf("Linked list: ");
printList(head);

// Delete some nodes
deleteNode(&head, 7);
deleteNode(&head, 5); // Key not found

// Print the linked list after deletion
printf("Linked list after deletion: ");
printList(head);

return 0;
}
```

Develop a program to reverse a linked list iteratively and recursively.

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

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

// Function to create a new node
```

```
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
if (newNode == NULL) {
printf("Memory allocation failed\n");
exit(1);
}
newNode->data = data;
newNode->next = NULL;
return newNode;
}

// Function to insert a node at the beginning of the linked list
void insertNode(struct Node** headRef, int data) {
struct Node* newNode = createNode(data);
newNode->next = *headRef;
*headRef = newNode;
}

// Function to reverse the linked list iteratively
void reverseList(struct Node** headRef) {
struct Node* prev = NULL;
struct Node* current = *headRef;
struct Node* next = NULL;

while (current != NULL) {
// Store next node
next = current->next;
// Reverse current node's pointer
current->next = prev;
// Move pointers one position ahead
prev = current;
current = next;
}
// Update head pointer to point to the new first node (previously the last node)
*headRef = prev;
}
```

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

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

// Insert elements into the linked list
insertNode(&head, 1);
insertNode(&head, 2);
insertNode(&head, 3);
insertNode(&head, 4);
insertNode(&head, 5);

printf("Original linked list: ");
printList(head);

// Reverse the linked list
reverseList(&head);

printf("Reversed linked list: ");
printList(head);

return 0;
}
```

Develop a program to reverse a linked list recursively.

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

```
#include <stdlib.h>

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

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

// Function to insert a node at the beginning of the linked list
void insert(struct Node** head_ref, int data) {
struct Node* new_node = newNode(data);
new_node->next = *head_ref;
*head_ref = new_node;
}

// Function to reverse a linked list recursively
struct Node* reverse(struct Node* head) {
// Base case: if the list is empty or has only one node
if (head == NULL || head->next == NULL) {
return head;
}

// Recursive case: reverse the rest of the list
struct Node* rest = reverse(head->next);

// Change the next of the second node
// Head becomes the last node
head->next->next = head;
```

```
head->next = NULL;

// Return the new head of the reversed list
return rest;
}

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

// Main function
int main() {
struct Node* head = NULL;
insert(&head, 5);
insert(&head, 4);
insert(&head, 3);
insert(&head, 2);
insert(&head, 1);

printf("Original linked list: \n");
printList(head);

head = reverse(head);

printf("Reversed linked list: \n");
printList(head);

return 0;
}
```

Solve problems involving linked list traversal and manipulation.

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

// Structure for a node in the linked list representing a digit
struct Node {
int digit;
struct Node* next;
};

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

// Function to insert a node at the beginning of the linked list
void insert(struct Node** head_ref, int digit) {
struct Node* new_node = newNode(digit);
new_node->next = *head_ref;
*head_ref = new_node;
}

// Function to print the linked list representing a long integer
void printList(struct Node* node) {
while (node != NULL) {
printf("%d", node->digit);
node = node->next;
}
}

// Function to add two long integers represented by linked lists
struct Node* add(struct Node* num1, struct Node* num2) {
```

```
struct Node* result = NULL;
int carry = 0;

while (num1 != NULL || num2 != NULL || carry != 0) {
int sum = carry;

if (num1 != NULL) {
sum += num1->digit;
num1 = num1->next;
}
if (num2 != NULL) {
sum += num2->digit;
num2 = num2->next;
}
carry = sum / 10;
sum = sum % 10;

insert(&result, sum);
}
return result;
}

// Main function
int main() {
struct Node* num1 = NULL;
struct Node* num2 = NULL;

// Insert digits of the first number
insert(&num1, 9);
insert(&num1, 9);
insert(&num1, 9);

// Insert digits of the second number
insert(&num2, 9);
insert(&num2, 9);
insert(&num2, 9);
```

```
printf("First number: ");
printList(num1);
printf("\n");

printf("Second number: ");
printList(num2);
printf("\n");

// Add the two numbers
struct Node* sum = add(num1, num2);

printf("Sum: ");
printList(sum);
printf("\n");

return 0;
}
```

Exercise 3: Linked List Applications

Create a program to detect and remove duplicates from a linked list.

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

// Structure for a node in the linked list
struct Node {
int data;
struct Node* next;
};

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

```
return temp;
}

// Function to insert a node at the beginning of the linked list
void insert(struct Node** head_ref, int data) {
struct Node* new_node = newNode(data);
new_node->next = *head_ref;
*head_ref = new_node;
}

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

// Function to remove duplicates from a sorted linked list
void removeDuplicates(struct Node* head) {
struct Node* current = head;

// Traverse the linked list
while (current != NULL && current->next != NULL) {
// Check if the next node has the same data as the current node
if (current->data == current->next->data) {
// If duplicate, remove the next node
struct Node* next_next = current->next->next;
free(current->next);
current->next = next_next;
} else {
// If not duplicate, move to the next node
current = current->next;
}
}
```

```
}

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

// Insert elements into the linked list
insert(&head, 5);
insert(&head, 4);
insert(&head, 4);
insert(&head, 2);
insert(&head, 2);
insert(&head, 1);

printf("Original linked list: ");
printList(head);

// Remove duplicates
removeDuplicates(head);

printf("Linked list after removing duplicates: ");
printList(head);

return 0;
}
```

Implement a linked list to represent polynomials and perform addition.

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

// Structure for a node in the linked list representing a term in a polynomial
struct Term {
int coefficient;
int exponent;
struct Term* next;
```

```
};

// Function to create a new term node
struct Term* newTerm(int coefficient, int exponent) {
struct Term* temp = (struct Term*)malloc(sizeof(struct Term));
temp->coefficient = coefficient;
temp->exponent = exponent;
temp->next = NULL;
return temp;
}

// Function to insert a term node at the beginning of the linked list
void insertTerm(struct Term** head_ref, int coefficient, int exponent) {
struct Term* new_node = newTerm(coefficient, exponent);
new_node->next = *head_ref;
*head_ref = new_node;
}

// Function to print the polynomial represented by the linked list
void printPolynomial(struct Term* head) {
struct Term* temp = head;
while (temp != NULL) {
printf("%dx^%d ", temp->coefficient, temp->exponent);
if (temp->next != NULL && temp->next->coefficient >= 0) {
printf("+ ");
}
temp = temp->next;
}
printf("\n");
}

// Function to add two polynomials represented by linked lists
struct Term* addPolynomials(struct Term* poly1, struct Term* poly2) {
struct Term* result = NULL;
struct Term* temp = NULL;
```

```
// Traverse both polynomials until one of them ends
while (poly1 != NULL && poly2 != NULL) {
// If the exponents of the current terms are equal, add their coefficients
if (poly1->exponent == poly2->exponent) {
int sum = poly1->coefficient + poly2->coefficient;
insertTerm(&result, sum, poly1->exponent);
poly1 = poly1->next;
poly2 = poly2->next;
}
// If the exponent of the current term in the first polynomial is greater, add it to the result
else if (poly1->exponent > poly2->exponent) {
insertTerm(&result, poly1->coefficient, poly1->exponent);
poly1 = poly1->next;
}
// If the exponent of the current term in the second polynomial is greater, add it to the result
else {
insertTerm(&result, poly2->coefficient, poly2->exponent);
poly2 = poly2->next;
}
}

// If there are any remaining terms in the first polynomial, add them to the result
while (poly1 != NULL) {
insertTerm(&result, poly1->coefficient, poly1->exponent);
poly1 = poly1->next;
}

// If there are any remaining terms in the second polynomial, add them to the result
while (poly2 != NULL) {
insertTerm(&result, poly2->coefficient, poly2->exponent);
poly2 = poly2->next;
}

return result;
}
```

```
// Main function
int main() {
struct Term* poly1 = NULL;
struct Term* poly2 = NULL;

// Insert terms into the first polynomial
insertTerm(&poly1, 5, 2);
insertTerm(&poly1, 4, 1);
insertTerm(&poly1, 3, 0);

// Insert terms into the second polynomial
insertTerm(&poly2, 3, 2);
insertTerm(&poly2, 2, 1);
insertTerm(&poly2, 1, 0);

printf("First polynomial: ");
printPolynomial(poly1);

printf("Second polynomial: ");
printPolynomial(poly2);

// Add the two polynomials
struct Term* sum = addPolynomials(poly1, poly2);

printf("Sum of polynomials: ");
printPolynomial(sum);

return 0;
}
```

Implement a double-ended queue (deque) with essential operations.

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

```
// Structure for a node in the deque
struct Node {
int data;
struct Node* prev;
struct Node* next;
};

// Structure for the deque
struct Deque {
struct Node* front;
struct Node* rear;
};

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

// Function to create an empty deque
struct Deque* createDeque() {
struct Deque* deque = (struct Deque*)malloc(sizeof(struct Deque));
deque->front = NULL;
deque->rear = NULL;
return deque;
}

// Function to check if the deque is empty
int isEmpty(struct Deque* deque) {
return deque->front == NULL;
}

// Function to insert an element at the front of the deque
```

```
void insertFront(struct Deque* deque, int data) {
struct Node* new_node = newNode(data);
if (isEmpty(deque)) {
deque->front = new_node;
deque->rear = new_node;
} else {
new_node->next = deque->front;
deque->front->prev = new_node;
deque->front = new_node;
}
}

// Function to insert an element at the rear of the deque
void insertRear(struct Deque* deque, int data) {
struct Node* new_node = newNode(data);
if (isEmpty(deque)) {
deque->front = new_node;
deque->rear = new_node;
} else {
new_node->prev = deque->rear;
deque->rear->next = new_node;
deque->rear = new_node;
}
}

// Function to delete an element from the front of the deque
void deleteFront(struct Deque* deque) {
if (isEmpty(deque)) {
printf("Deque is empty.\n");
} else {
struct Node* temp = deque->front;
deque->front = deque->front->next;
free(temp);
if (deque->front == NULL) {
deque->rear = NULL;
} else {
```

```
deque->front->prev = NULL;
}
}
}

// Function to delete an element from the rear of the deque
void deleteRear(struct Deque* deque) {
if (isEmpty(deque)) {
printf("Deque is empty.\n");
} else {
struct Node* temp = deque->rear;
deque->rear = deque->rear->prev;
free(temp);
if (deque->rear == NULL) {
deque->front = NULL;
} else {
deque->rear->next = NULL;
}
}
}

// Function to get the front element of the deque
int getFront(struct Deque* deque) {
if (isEmpty(deque)) {
printf("Deque is empty.\n");
return -1;
} else {
return deque->front->data;
}
}

// Function to get the rear element of the deque
int getRear(struct Deque* deque) {
if (isEmpty(deque)) {
printf("Deque is empty.\n");
return -1;
```

```
} else {
return deque->rear->data;
}
}

// Function to print the elements of the deque
void printDeque(struct Deque* deque) {
struct Node* temp = deque->front;
printf("Deque: ");
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
printf("\n");
}

// Main function
int main() {
struct Deque* deque = createDeque();

insertFront(deque, 1);
insertFront(deque, 2);
insertRear(deque, 3);
insertRear(deque, 4);

printDeque(deque);

printf("Front element: %d\n", getFront(deque));
printf("Rear element: %d\n", getRear(deque));

deleteFront(deque);
deleteRear(deque);

printDeque(deque);

return 0;
```

```
}
```

Exercise 4: Double Linked List Implementation

Implement a doubly linked list and perform various operations to understand its properties and applications.

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

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

void insertAtFront(struct Node** head, int data) {
// Create a new struct node and allocate memory.
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
// Initialize data in the newly created node.
newNode->data = data;
// Make the next pointer of the new node point to the head.
// In this way, this new node is added at front of the linked list.
newNode->next = (*head);
newNode->prev = NULL;
// If the head is not NULL, then we update the prev pointer in the current head.
// The prev pointer of the current head will point to the new node.
if((*head) != NULL) {
(*head)->prev = newNode;
}
// Update the head pointer to point to the new node.
(*head) = newNode;
}

void insertAtEnd(struct Node** head, int data) {
// Create a new struct node and allocate memory.
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
// Initialize data in the newly created node.
newNode->data = data;
newNode->next = NULL;
if((*head) == NULL) {
// If the head is NULL, then the new node is the new head.
newNode->prev = NULL;
// Assign head as the new node.
(*head) = newNode;
return ;
}
// If the list is not empty then traverse to the end of the list.
struct Node* ptr = (*head);
while(ptr->next != NULL) {
ptr = ptr->next;
}
// From the last node, add a link to the new node.
ptr->next = newNode;
// Update the prev pointer in the new node to point to the previous last node.
newNode->prev = ptr;
}

void insertAfterNode(struct Node* node, int data) {
if(node == NULL) {
// If the node is NULL, then we cannot insert.
printf("The given node cannot be NULL.");
return;
}
// Create a new struct node and allocate memory.
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
// Initialize data in the newly created node.
newNode->data = data;
// Add a link from new node to next_node.
newNode->next = node->next;
// Add a link from the given node to the new node.
node->next = newNode;
// Update the prev pointer of the new node to the given node.
```

```
newNode->prev = node;
// If the next_node is not NULL, then update the prev of the next_node.
if(newNode->next != NULL) {
newNode->next->prev = newNode;
}
}

void insertBeforeNode(struct Node** head, struct Node* node, int data) {
if(node == NULL) {
// If the given node is NULL, then we cannot insert a node before a NULL node.
printf("The given node cannot be NULL.");
return ;
}
// Create a new struct node and allocate memory.
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
// Initialize data in the newly created node.
newNode->data = data;
// Add a link from the new node to the given node.
newNode->next = node;
if(node->prev != NULL) {
// If the given node is not the head node, then we update the next pointer of previous node
node->prev->next = newNode;
// Update the prev pointer of the new node to point to the previous node.
newNode->prev = node->prev;
}
else {
// If the previous value is NULL, then this means the given node is head.
// So we are inserting it at the front of head.
// Hence, we update the value of the head pointer to the new node.
*head = newNode;
}
// Update the previous pointer of the given node.
node->prev = newNode;
}

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

```
// If the head pointer is NULL, then we cannot delete it.
if((*head) == NULL) return ;
struct Node* curr = *head;
// Update the head pointer to the next node.
*head = (*head)->next;
// Make the previous head pointer NULL and free the memory.
curr->next = NULL;
free(curr);
}

void deleteAtLast(struct Node** head) {
// If the head pointer is NULL, then we cannot delete it.
if((*head) == NULL) return ;
// Traverse to the last node.
struct Node* curr = *head;
while(curr->next != NULL) {
curr = curr->next;
}
// Second last node will point to NULL.
struct Node* prev = curr->prev;
prev->next = NULL;
// Make the last node NULL and free the memory.
curr->prev = NULL;
free(curr);
}

void deleteAtPosition(struct Node** head, int position) {
struct Node* curr = *head;
if(position == 1) {
// If the position is 1, then delete the head and return.
deleteAtFront(head);
return ;
}
// Go the node to be deleted.
while(position > 1 && curr) {
// While the position is greater than 1 and the current node is not NULL,
```

```
// we iterate forward and decrease position by 1.
curr = curr->next;
position --;
}
// If the current node is NULL, then we cannot delete it.
// The node at the specified position does not exist.
if(curr == NULL) {
printf("Node at position is not present.\n");
return ;
}
// Create pointers to the previous node and the next node of the current node.
struct Node* prevNode = curr->prev;
struct Node* nextNode = curr->next;
// Update the next pointer of the previous node and the previous pointer of the next node.
// In this way, the middle node is removed from the list.
curr->prev->next = nextNode;
if(nextNode != NULL) {
nextNode->prev = prevNode;
}
// Remove the link to the list and free the memory.
curr->next = curr->prev = NULL;
free(curr);
}

void traverse(struct Node* head) {
for(struct Node* i = head; i != NULL; i = i->next) {
printf("%d", i->data);
if(i->next) printf(" <--> ");
}
printf(" -> NULL \n");
}



int main() {
struct Node* head = NULL;
printf("Insert 1 at the front\n");
```

```
insertAtFront(&head, 1);
traverse(head);
printf("Insert 2 at the front\n");
insertAtFront(&head, 2);
traverse(head);
printf("Insert 4 at the end\n");
insertAtEnd(&head, 4);
traverse(head);
printf("Insert 5 at the front\n");
insertAtEnd(&head, 5);
traverse(head);
printf("Insert 3 after head->next\n");
insertAfterNode(head->next, 3);
traverse(head);
printf("Insert -1 before the head\n");
insertBeforeNode(&head, head, -1);
traverse(head);
printf("Insert 100 before head->next->next\n");
insertBeforeNode(&head, head->next->next, 100);
traverse(head);
printf("Delete the front node.\n");
deleteAtFront(&head);
traverse(head);
printf("Delete the last node\n");
deleteAtLast(&head);
traverse(head);
printf("Delete the second node from the front.\n");
deleteAtPosition(&head, 2);
traverse(head);
return 0;
}
```

Implement a circular linked list and perform insertion, deletion, and traversal.

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

```
#include <stdlib.h>

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

// Function to insert a node at the beginning of a circular linked list
void insertAtBeginning(struct Node **head_ref, int data) {
struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
struct Node *last = *head_ref;

new_node->data = data;
new_node->next = *head_ref;

// If the list is empty, make the new node as head
if (*head_ref == NULL) {
new_node->next = new_node;
} else {
// Otherwise, traverse to the last node and update its next pointer
while (last->next != *head_ref) {
last = last->next;
}
last->next = new_node;
}

*head_ref = new_node; // Update head pointer
}

// Function to delete a node from a circular linked list
void deleteNode(struct Node **head_ref, int key) {
if (*head_ref == NULL)
return;

struct Node *temp = *head_ref, *prev = NULL;
```

```
// If the head node itself holds the key to be deleted
if (temp->data == key && temp->next == *head_ref) {
*head_ref = NULL;
free(temp);
return;
}

// Search for the key to be deleted, keep track of the previous node
do {
if (temp->data == key)
break;
prev = temp;
temp = temp->next;
} while (temp != *head_ref);

// If key was not present in the linked list
if (temp == *head_ref)
return;

// Unlink the node from the list
if (temp == *head_ref) // If the node to be deleted is the head node
*head_ref = temp->next;
prev->next = temp->next;
free(temp);
}

// Function to display a circular linked list
void display(struct Node *head) {
struct Node *temp = head;
if (head != NULL) {
do {
printf("%d ", temp->data);
temp = temp->next;
} while (temp != head);
printf("\n");
```

```
}
}

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

// Insert some nodes
insertAtBeginning(&head, 1);
insertAtBeginning(&head, 2);
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 4);

printf("Circular Linked List: ");
display(head);

// Delete a node
deleteNode(&head, 3);
printf("Circular Linked List after deleting node with value 3: ");
display(head);

return 0;
}
```

Exercise 5: Stack Operations
Implement a stack using arrays.

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

#define MAX_SIZE 100

// Structure for stack
struct Stack {
int items[MAX_SIZE];
int top;
};
```

```
// Function to initialize the stack
void initializeStack(struct Stack *stack) {
stack->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *stack) {
return (stack->top == -1);
}

// Function to check if the stack is full
int isFull(struct Stack *stack) {
return (stack->top == (MAX_SIZE - 1));
}

// Function to push an element onto the stack
void push(struct Stack *stack, int value) {
if (isFull(stack)) {
printf("Stack Overflow\n");
return;
}
stack->items[++stack->top] = value;
}

// Function to pop an element from the stack
int pop(struct Stack *stack) {
if (isEmpty(stack)) {
printf("Stack Underflow\n");
exit(EXIT_FAILURE);
}
return stack->items[stack->top--];
}

// Function to return the top element of the stack without removing it
int peek(struct Stack *stack) {
```

```
if (isEmpty(stack)) {
printf("Stack is empty\n");
exit(EXIT_FAILURE);
}
return stack->items[stack->top];
}
int main() {
struct Stack stack;
initializeStack(&stack);
// Pushing elements onto the stack
push(&stack, 1);
push(&stack, 2);
push(&stack, 3);
push(&stack, 4);
push(&stack, 5);
printf("Top element of the stack: %d\n", peek(&stack));
// Popping elements from the stack
printf("Popping elements from the stack: ");
while (!isEmpty(&stack)) {
printf("%d ", pop(&stack));
}
printf("\n");
return 0;
}
```

Implement a stack using linked lists.

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

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

// Function to create a new node
```

```
struct Node* newNode(int data) {
struct Node* node = (struct Node*)malloc(sizeof(struct Node));
if (node == NULL) {
printf("Memory allocation failed\n");
exit(EXIT_FAILURE);
}
node->data = data;
node->next = NULL;
return node;
}

// Function to check if the stack is empty
int isEmpty(struct Node* root) {
return !root;
}

// Function to push an element onto the stack
void push(struct Node** root, int data) {
struct Node* node = newNode(data);
node->next = *root;
*root = node;
printf("%d pushed to stack\n", data);
}

// Function to pop an element from the stack
int pop(struct Node** root) {
if (isEmpty(*root)) {
printf("Stack underflow\n");
return -1;
}
struct Node* temp = *root;
*root = (*root)->next;
int popped = temp->data;
free(temp);
return popped;
}
```

```
// Function to peek at the top element of the stack
int peek(struct Node* root) {
if (isEmpty(root)) {
printf("Stack is empty\n");
return -1;
}
return root->data;
}

int main() {
struct Node* root = NULL;
push(&root, 10);
push(&root, 20);
push(&root, 30);
printf("%d popped from stack\n", pop(&root));
printf("Top element is %d\n", peek(root));
return 0;
}
```

Write a program to evaluate a postfix expression using a stack.

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

#define MAX_STACK_SIZE 100

// Stack structure
struct Stack {
int top;
int items[MAX_STACK_SIZE];
};

// Function to initialize a stack
void initializeStack(struct Stack *s) {
s->top = -1;
```

```
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
return s->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack *s) {
return s->top == MAX_STACK_SIZE - 1;
}

// Function to push an element onto the stack
void push(struct Stack *s, int value) {
if (isFull(s)) {
printf("Stack overflow\n");
exit(EXIT_FAILURE);
}
s->items[++(s->top)] = value;
}

// Function to pop an element from the stack
int pop(struct Stack *s) {
if (isEmpty(s)) {
printf("Stack underflow\n");
exit(EXIT_FAILURE);
}
return s->items[(s->top)--];
}

// Function to evaluate a postfix expression
int evaluatePostfix(char *expression) {
struct Stack stack;
initializeStack(&stack);

// Traverse the expression
```

```
for (int i = 0; expression[i] != '\0'; ++i) {
// If the current character is a digit, push it onto the stack
if (isdigit(expression[i])) {
push(&stack, expression[i] - '0');
} else {
// If the current character is an operator, pop two operands from the stack,
// perform the operation, and push the result back onto the stack
int operand2 = pop(&stack);
int operand1 = pop(&stack);
switch (expression[i]) {
case '+':
push(&stack, operand1 + operand2);
break;
case '-':
push(&stack, operand1 - operand2);
break;
case '*':
push(&stack, operand1 * operand2);
break;
case '/':
push(&stack, operand1 / operand2);
break;
}
}
}

// The final result will be at the top of the stack
return pop(&stack);
}

int main() {
char expression[100];
printf("Enter a postfix expression: ");
scanf("%s", expression);
int result = evaluatePostfix(expression);
printf("Result of evaluation: %d\n", result);
```

```
return 0;
}
```

Implement a program to check for balanced parentheses using a stack.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 100
// Stack structure
struct Stack {
int top;
char items[MAX_STACK_SIZE];
};

// Function to initialize a stack
void initializeStack(struct Stack *s) {
s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
return s->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack *s) {
return s->top == MAX_STACK_SIZE - 1;
}

// Function to push an element onto the stack
void push(struct Stack *s, char value) {
if (isFull(s)) {
printf("Stack overflow\n");
exit(EXIT_FAILURE);
}
s->items[++(s->top)] = value;
}
```

```
// Function to pop an element from the stack
char pop(struct Stack *s) {
if (isEmpty(s)) {
printf("Stack underflow\n");
exit(EXIT_FAILURE);
}
return s->items[(s->top)--];
}
// Function to check if parentheses are balanced
int areParenthesesBalanced(char *expression) {
struct Stack stack;
initializeStack(&stack);

// Traverse the expression
for (int i = 0; expression[i] != '\0'; ++i) {
// If the current character is an opening parenthesis, push it onto the stack
if (expression[i] == '(' || expression[i] == '[' || expression[i] == '{') {
push(&stack, expression[i]);
}
// If the current character is a closing parenthesis, pop from the stack
// and check if the popped parenthesis matches the current parenthesis
else if (expression[i] == ')' || expression[i] == ']' || expression[i] == '}') {
if (isEmpty(&stack)) {
return 0; // Unbalanced if closing parenthesis with no opening parenthesis
}
char popped = pop(&stack);
if ((expression[i] == ')' && popped != '(') ||
(expression[i] == ']' && popped != '[') ||
(expression[i] == '}' && popped != '{')) {
return 0; // Mismatched parenthesis
}
}
}

// If the stack is empty at the end, parentheses are balanced
```

```
return isEmpty(&stack);
}

int main() {
char expression[100];
printf("Enter an expression: ");
scanf("%s", expression);
if (areParenthesesBalanced(expression)) {
printf("Parentheses are balanced\n");
} else {
printf("Parentheses are not balanced\n");
}
return 0;
}
```

Exercise 6: Queue Operations

Implement a queue using arrays.

```
#include <stdio.h>
#define MAX 50
void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
int choice;
while (1)
{
printf("1.Insert element to queue \n");
printf("2.Delete element from queue \n");
printf("3.Display all elements of queue \n");
printf("4.Quit \n");
printf("Enter your choice : ");
scanf("%d", &choice);
```

```
switch (choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(1);
default:
printf("Wrong choice \n");
} /* End of switch */
} /* End of while */
} /* End of main() */

void insert()
{
int add_item;
if (rear == MAX - 1)
printf("Queue Overflow \n");
else
{
if (front == - 1)
/*If queue is initially empty */
front = 0;
printf("Inset the element in queue : ");
scanf("%d", &add_item);
rear = rear + 1;
queue_array[rear] = add_item;
}
} /* End of insert() */
```

```
void delete()
{
if (front == - 1 || front > rear)
{
printf("Queue Underflow \n");
return ;
}
else
{
printf("Element deleted from queue is : %d\n", queue_array[front]);
front = front + 1;
}
} /* End of delete() */

void display()
{
int i;
if (front == - 1)
printf("Queue is empty \n");
else
{
printf("Queue is : \n");
for (i = front; i <= rear; i++)
printf("%d ", queue_array[i]);
printf("\n");
}
}
```

Implement a queue using arrays and linked lists.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
```

```
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
int choice;
while(choice != 4)
{
printf("\n************************Main Menu*****************************\n");
printf("\n=====================================================================\n");
printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",& choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
}
}
}
void insert()
{
```

```
struct node *ptr;
int item;

ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
printf("\nEnter value?\n");
scanf("%d",&item);
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
}
}
}
void delete ()
{
struct node *ptr;
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
```

```
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}
void display()
{
struct node *ptr;
ptr = front;
if(front == NULL)
{
printf("\nEmpty queue\n");
}
else
{   printf("\nprinting values .....\n");
while(ptr != NULL)
{
printf("\n%d\n",ptr -> data);
ptr = ptr -> next;
}
}
}
```

Develop a program to simulate a simple printer queue system.

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

#define MAX_QUEUE_SIZE 100

// Structure to represent a print job
struct PrintJob {
int jobId;
int priority;
```

```
};

// Queue structure
struct Queue {
int front, rear, size;
unsigned capacity;
struct PrintJob* array;
};

// Function to create a queue
struct Queue* createQueue(unsigned capacity) {
struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
if (queue == NULL) {
printf("Memory allocation failed\n");
exit(EXIT_FAILURE);
}
queue->capacity = capacity;
queue->front = queue->size = 0;
queue->rear = capacity - 1; // Important to initialize rear as -1 for enQueue
queue->array = (struct PrintJob*)malloc(queue->capacity * sizeof(struct PrintJob));
if (queue->array == NULL) {
printf("Memory allocation failed\n");
exit(EXIT_FAILURE);
}
return queue;
}

// Function to check if the queue is full
int isFull(struct Queue* queue) {
return (queue->size == queue->capacity);
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
return (queue->size == 0);
}
```

```
// Function to add a print job to the queue
void enQueue(struct Queue* queue, struct PrintJob job) {
if (isFull(queue)) {
printf("Queue overflow\n");
return;
}
queue->rear = (queue->rear + 1) % queue->capacity;
queue->array[queue->rear] = job;
queue->size += 1;
printf("Print job with ID %d enqueued to queue\n", job.jobId);
}

// Function to remove a print job from the queue
struct PrintJob deQueue(struct Queue* queue) {
if (isEmpty(queue)) {
printf("Queue underflow\n");
exit(EXIT_FAILURE);
}
struct PrintJob job = queue->array[queue->front];
queue->front = (queue->front + 1) % queue->capacity;
queue->size -= 1;
return job;
}

int main() {
struct Queue* printQueue = createQueue(MAX_QUEUE_SIZE);

// Enqueue print jobs with different priorities
struct PrintJob job1 = {1, 3};
struct PrintJob job2 = {2, 1};
struct PrintJob job3 = {3, 2};
struct PrintJob job4 = {4, 2};

enQueue(printQueue, job1);
enQueue(printQueue, job2);
```

```
enQueue(printQueue, job3);
enQueue(printQueue, job4);

// Dequeue and process print jobs based on priority
while (!isEmpty(printQueue)) {
struct PrintJob job = deQueue(printQueue);
printf("Processing print job with ID %d and priority %d\n", job.jobId, job.priority);
}
return 0;
}
```

Solve problems involving circular queues.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_QUEUE_SIZE 5
// Circular Queue structure
struct CircularQueue {
int front, rear, size;
unsigned capacity;
int* array;
};

// Function to create a circular queue
struct CircularQueue* createCircularQueue(unsigned capacity) {
struct CircularQueue* queue = (struct CircularQueue*)malloc(sizeof(struct CircularQueue));
if (queue == NULL) {
printf("Memory allocation failed\n");
exit(EXIT_FAILURE);
}
queue->capacity = capacity;
queue->front = queue->size = 0;
queue->rear = capacity - 1; // Important to initialize rear as -1 for enQueue
queue->array = (int*)malloc(queue->capacity * sizeof(int));
if (queue->array == NULL) {
printf("Memory allocation failed\n");
exit(EXIT_FAILURE);
```

```
}
return queue;
}

// Function to check if the circular queue is full
int isFull(struct CircularQueue* queue) {
return (queue->size == queue->capacity);
}

// Function to check if the circular queue is empty
int isEmpty(struct CircularQueue* queue) {
return (queue->size == 0);
}

// Function to add an element to the circular queue
void enQueue(struct CircularQueue* queue, int item) {
if (isFull(queue)) {
printf("Queue overflow\n");
return;
}
queue->rear = (queue->rear + 1) % queue->capacity;
queue->array[queue->rear] = item;
queue->size += 1;
printf("Passenger %d got ticket\n", item);
}

// Function to remove an element from the circular queue
int deQueue(struct CircularQueue* queue) {
if (isEmpty(queue)) {
printf("Queue underflow\n");
exit(EXIT_FAILURE);
}
int item = queue->array[queue->front];
queue->front = (queue->front + 1) % queue->capacity;
queue->size -= 1;
return item;
```

```
}

int main() {
struct CircularQueue* ticketCounterQueue = createCircularQueue(MAX_QUEUE_SIZE);

// Let's simulate passengers arriving at the ticket counter
for (int i = 1; i <= 7; ++i) {
enQueue(ticketCounterQueue, i);
}

// Ticket counter serving passengers
printf("Passengers getting served at the ticket counter:\n");
while (!isEmpty(ticketCounterQueue)) {
int passenger = deQueue(ticketCounterQueue);
printf("Passenger %d got ticket served\n", passenger);
}
return 0;
}
```

Exercise 7: Stack and Queue Applications

Use a stack to evaluate an infix expression and convert it to postfix.

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

char stack[100];
int top = -1;

void push(char x)
{
stack[++top] = x;
}

char pop()
{
if(top == -1)
```

```
return -1;
else
return stack[top--];
}

int priority(char x)
{
if(x == '(')
return 0;
if(x == '+' || x == '-')
return 1;
if(x == '*' || x == '/')
return 2;
return 0;
}

int main()
{
char exp[100];
char *e, x;
printf("Enter the expression : ");
scanf("%s",exp);
printf("\n");
e = exp;

while(*e != '\0')
{
if(isalnum(*e))
printf("%c ",*e);
else if(*e == '(')
push(*e);
else if(*e == ')')
{
while((x = pop()) != '(')
printf("%c ", x);
}
```

```
else
{
while(priority(stack[top]) >= priority(*e))
printf("%c ",pop());
push(*e);
}
e++;
}

while(top != -1)
{
printf("%c ",pop());
}return 0;
}
```

Create a program to determine whether a given string is a palindrome or not.

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

char* stack;
int top = -1;

// push function
void push(char ele)
{
stack[++top] = ele;
}

// pop function
char pop()
{
return stack[top--];
}
```

```
// Function that returns 1
// if str is a palindrome
int isPalindrome(char str[])
{
int length = strlen(str);

// Allocating the memory for the stack
stack = (char*)malloc(length * sizeof(char));

// Finding the mid
int i, mid = length / 2;

for (i = 0; i < mid; i++) {
push(str[i]);
}

// Checking if the length of the string
// is odd, if odd then neglect the
// middle character
if (length % 2 != 0) {
i++;
}

// While not the end of the string
while (str[i] != '\0') {
char ele = pop();

// If the characters differ then the
// given string is not a palindrome
if (ele != str[i])
return 0;
i++;
}
return 1;
}
```

```
// Driver code
int main()
{
char str[] = "madam";

if (isPalindrome(str)) {
printf("Yes");
}
else {
printf("No");
}

return 0;
}
```

Implement a stack or queue to perform comparison and check for symmetry

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 50

int top = -1, front = 0;
int stack[MAX];
void push(char);
void pop();
void main()
{
int i, choice;
char s[MAX], b;
while (1)
{
printf("1-enter string\n2-exit\n");
printf("enter your choice\n");
scanf("%d", &choice);
switch (choice)
{
```

```
case 1:
printf("Enter the String\n");
scanf("%s", s);
for (i = 0;s[i] != '\0';i++)
{
b = s[i];
push(b);
}
for (i = 0;i < (strlen(s) / 2);i++)
{
if (stack[top] == stack[front])
{
pop();
front++;
}
else
{
printf("%s is not a palindrome\n", s);
break;
}
}
if ((strlen(s) / 2)  =  =  front)
printf("%s is palindrome\n",  s);
front  =  0;
top  =  -1;
break;
case 2:
exit(0);
default:
printf("enter correct choice\n");
}
}
}
/* to push a character into stack */
void push(char a)
{
```

```
top++;
stack[top]  =  a;
}
/* to delete an element in stack */
void pop()
{
top--;
}
```

Exercise 8: Binary Search Tree

Implementing a BST using Linked List.

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

struct node {
int data; //node will store some data
struct node *right_child; // right child
struct node *left_child; // left child
};

//function to create a node
struct node* new_node(int x) {
struct node *temp;
temp = malloc(sizeof(struct node));
temp -> data = x;
temp -> left_child = NULL;
temp -> right_child = NULL;

return temp;
}

// searching operation
struct node* search(struct node * root, int x) {
if (root == NULL || root -> data == x) //if root->data is x then the element is found
return root;
else if (x > root -> data) // x is greater, so we will search the right subtree
```

```
return search(root -> right_child, x);
else //x is smaller than the data, so we will search the left subtree
return search(root -> left_child, x);
}

// insertion
struct node* insert(struct node * root, int x) {
//searching for the place to insert
if (root == NULL)
return new_node(x);
else if (x > root -> data) // x is greater. Should be inserted to the right
root -> right_child = insert(root -> right_child, x);
else // x is smaller and should be inserted to left
root -> left_child = insert(root -> left_child, x);
return root;
}

//function to find the minimum value in a node
struct node* find_minimum(struct node * root) {
if (root == NULL)
return NULL;
else if (root -> left_child != NULL) // node with minimum value will have no left child
return find_minimum(root -> left_child); // left most element will be minimum
return root;
}

// deletion
struct node* delete(struct node * root, int x) {
//searching for the item to be deleted
if (root == NULL)
return NULL;
if (x > root -> data)
root -> right_child = delete(root -> right_child, x);
else if (x < root -> data)
root -> left_child = delete(root -> left_child, x);
else {
```

```
//No Child node
if (root -> left_child == NULL && root -> right_child == NULL) {
free(root);
return NULL;
}

//One Child node
else if (root -> left_child == NULL || root -> right_child == NULL) {
struct node *temp;
if (root -> left_child == NULL)
temp = root -> right_child;
else
temp = root -> left_child;
free(root);
return temp;
}

//Two Children
else {
struct node *temp = find_minimum(root -> right_child);
root -> data = temp -> data;
root -> right_child = delete(root -> right_child, temp -> data);
}
}
return root;
}

// Inorder Traversal
void inorder(struct node *root) {
if (root != NULL) // checking if the root is not null
{
inorder(root -> left_child); // traversing left child
printf(" %d ", root -> data); // printing data at root
inorder(root -> right_child); // traversing right child
}
}
```

```
int main() {
struct node *root;
root = new_node(20);
insert(root, 5);
insert(root, 1);
insert(root, 15);
insert(root, 9);
insert(root, 7);
insert(root, 12);
insert(root, 30);
insert(root, 25);
insert(root, 40);
insert(root, 45);
insert(root, 42);

inorder(root);
printf("\n");
root = delete(root, 1);
root = delete(root, 40);
root = delete(root, 45);
root = delete(root, 9);
inorder(root);
printf("\n");
return 0;
}
```

Traversing of BST.

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

struct node {
int item;
struct node* left;
struct node* right;
};
```

```
// Inorder traversal
void inorderTraversal(struct node* root) {
if (root == NULL) return;
inorderTraversal(root->left);
printf("%d ->", root->item);
inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
if (root == NULL) return;
printf("%d ->", root->item);
preorderTraversal(root->left);
preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
if (root == NULL) return;
postorderTraversal(root->left);
postorderTraversal(root->right);
printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(value) {
struct node* newNode = malloc(sizeof(struct node));
newNode->item = value;
newNode->left = NULL;
newNode->right = NULL;

return newNode;
}

// Insert on the left of the node
```

```
struct node* insertLeft(struct node* root, int value) {
root->left = createNode(value);
return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
root->right = createNode(value);
return root->right;
}

int main() {
struct node* root = createNode(1);
insertLeft(root, 12);
insertRight(root, 9);

insertLeft(root->left, 5);
insertRight(root->left, 6);

printf("Inorder traversal \n");
inorderTraversal(root);

printf("\nPreorder traversal \n");
preorderTraversal(root);

printf("\nPostorder traversal \n");
postorderTraversal(root);
}
```

Exercise 9: Hashing

Implement a hash table with collision resolution techniques.

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

#define size 7

struct node
```

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

struct node *chain[size];

void init()
{
int i;
for(i = 0; i < size; i++)
chain[i] = NULL;
}

void insert(int value)
{
//create a newnode with value
struct node *newNode = malloc(sizeof(struct node));
newNode->data = value;
newNode->next = NULL;

//calculate hash key
int key = value % size;

//check if chain[key] is empty
if(chain[key] == NULL)
chain[key] = newNode;
//collision
else
{
//add the node at the end of chain[key].
struct node *temp = chain[key];
while(temp->next)
{
temp = temp->next;
}
```

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

void print()
{
int i;

for(i = 0; i < size; i++)
{
struct node *temp = chain[i];
printf("chain[%d]-->",i);
while(temp)
{
printf("%d -->",temp->data);
temp = temp->next;
}
printf("NULL\n");
}
}

int main()
{
//init array of list to NULL
init();

insert(7);
insert(0);
insert(3);
insert(10);
insert(4);
insert(5);

print();
```

```
return 0;
}
```

Write a program to implement a simple cache using hashing(Using Queue and DLL).

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


typedef struct QNode {
struct QNode *prev, *next;
unsigned
pageNumber; // the page number stored in this QNode
} QNode;

// A Queue (A FIFO collection of Queue Nodes)
typedef struct Queue {
unsigned count; // Number of filled frames
unsigned numberOfFrames; // total number of frames
QNode *front, *rear;
} Queue;

// A hash (Collection of pointers to Queue Nodes)
typedef struct Hash {
int capacity; // how many pages can be there
QNode** array; // an array of queue nodes
} Hash;

// A utility function to create a new Queue Node. The queue
// Node will store the given 'pageNumber'
QNode* newQNode(unsigned pageNumber)
{
// Allocate memory and assign 'pageNumber'
QNode* temp = (QNode*)malloc(sizeof(QNode));
temp->pageNumber = pageNumber;

// Initialize prev and next as NULL
temp->prev = temp->next = NULL;
```

```
return temp;
}

// A utility function to create an empty Queue.
// The queue can have at most 'numberOfFrames' nodes
Queue* createQueue(int numberOfFrames)
{
Queue* queue = (Queue*)malloc(sizeof(Queue));

// The queue is empty
queue->count = 0;
queue->front = queue->rear = NULL;

// Number of frames that can be stored in memory
queue->numberOfFrames = numberOfFrames;

return queue;
}

// A utility function to create an empty Hash of given
// capacity
Hash* createHash(int capacity)
{
// Allocate memory for hash
Hash* hash = (Hash*)malloc(sizeof(Hash));
hash->capacity = capacity;

// Create an array of pointers for referring queue nodes
hash->array
= (QNode**)malloc(hash->capacity * sizeof(QNode*));

// Initialize all hash entries as empty
int i;
for (i = 0; i < hash->capacity; ++i)
hash->array[i] = NULL;
```

```
return hash;
}

// A function to check if there is slot available in memory
int AreAllFramesFull(Queue* queue)
{
return queue->count == queue->numberOfFrames;
}

// A utility function to check if queue is empty
int isQueueEmpty(Queue* queue)
{
return queue->rear == NULL;
}

// A utility function to delete a frame from queue
void deQueue(Queue* queue)
{
if (isQueueEmpty(queue))
return;

// If this is the only node in list, then change front
if (queue->front == queue->rear)
queue->front = NULL;

// Change rear and remove the previous rear
QNode* temp = queue->rear;
queue->rear = queue->rear->prev;

if (queue->rear)
queue->rear->next = NULL;

free(temp);

// decrement the number of full frames by 1
```

```
queue->count--;
}

// A function to add a page with given 'pageNumber' to both
// queue and hash
void Enqueue(Queue* queue, Hash* hash, unsigned pageNumber)
{
// If all frames are full, remove the page at the rear
if (AreAllFramesFull(queue)) {
// remove page from hash
hash->array[queue->rear->pageNumber] = NULL;
deQueue(queue);
}

// Create a new node with given page number,
// And add the new node to the front of queue
QNode* temp = newQNode(pageNumber);
temp->next = queue->front;

// If queue is empty, change both front and rear
// pointers
if (isQueueEmpty(queue))
queue->rear = queue->front = temp;
else // Else change the front
{
queue->front->prev = temp;
queue->front = temp;
}

// Add page entry to hash also
hash->array[pageNumber] = temp;

// increment number of full frames
queue->count++;
}
```

```
// This function is called when a page with given
// 'pageNumber' is referenced from cache (or memory). There
// are two cases:
// 1. Frame is not there in memory, we bring it in memory
// and add to the front of queue
// 2. Frame is there in memory, we move the frame to front
// of queue
void ReferencePage(Queue* queue, Hash* hash,
unsigned pageNumber)
{
QNode* reqPage = hash->array[pageNumber];

// the page is not in cache, bring it
if (reqPage == NULL)
Enqueue(queue, hash, pageNumber);

// page is there and not at front, change pointer
else if (reqPage != queue->front) {
// Unlink rquested page from its current location
// in queue.
reqPage->prev->next = reqPage->next;
if (reqPage->next)
reqPage->next->prev = reqPage->prev;

// If the requested page is rear, then change rear
// as this node will be moved to front
if (reqPage == queue->rear) {
queue->rear = reqPage->prev;
queue->rear->next = NULL;
}

// Put the requested page before current front
reqPage->next = queue->front;
reqPage->prev = NULL;

// Change prev of current front
```

```
reqPage->next->prev = reqPage;

// Change front to the requested page
queue->front = reqPage;
}
}

// Driver code
int main()
{
// Let cache can hold 4 pages
Queue* q = createQueue(4);

// Let 10 different pages can be requested (pages to be
// referenced are numbered from 0 to 9
Hash* hash = createHash(10);

// Let us refer pages 1, 2, 3, 1, 4, 5
ReferencePage(q, hash, 1);
ReferencePage(q, hash, 2);
ReferencePage(q, hash, 3);
ReferencePage(q, hash, 1);
ReferencePage(q, hash, 4);
ReferencePage(q, hash, 5);

// Let us print cache frames after the above referenced
// pages
printf("%d ", q->front->pageNumber);
printf("%d ", q->front->next->pageNumber);
printf("%d ", q->front->next->next->pageNumber);
printf("%d ", q->front->next->next->next->pageNumber);

return 0;
}
```

Exercise 10: Graphs

i) Write a program to implement a graph using BFS

```
#include <stdio.h>

int n, i, j, visited[10], queue[10], front = -1, rear = -1;
int adj[10][10];

void bfs(int v)
{
for (i = 1; i <= n; i++)
if (adj[v][i] && !visited[i])
queue[++rear] = i;
if (front <= rear)
{
visited[queue[front]] = 1;
bfs(queue[front++]);
}
}

void main()
{
int v;
printf("Enter the number of vertices: ");
scanf("%d", &n);
for (i = 1; i <= n; i++)
{
queue[i] = 0;
visited[i] = 0;
}
printf("Enter graph data in matrix form:    \n");
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
scanf("%d", &adj[i][j]);
printf("Enter the starting vertex: ");
scanf("%d", &v);
bfs(v);
```

```
printf("The node which are reachable are:     \n");
for (i = 1; i <= n; i++)
if (visited[i])
printf("%d\t", i);
else
printf("BFS is not possible. Not all nodes are reachable");
return 0;
}
```

Write a program to implement a graph using DFS

```
#include<stdio.h>
#include<conio.h>
int a[20][20], reach[20], n;
void dfs(int v) {
int i;
reach[v] = 1;
for (i = 1; i <= n; i++)
if (a[v][i] && !reach[i]) {
printf("\n %d->%d", v, i);
dfs(i);
}
}
int main(int argc, char **argv) {
int i, j, count = 0;
printf("\n Enter number of vertices:");
scanf("%d", &n);
for (i = 1; i <= n; i++) {
reach[i] = 0;
for (j = 1; j <= n; j++)
a[i][j] = 0;
}
printf("\n Enter the adjacency matrix:\n");
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
scanf("%d", &a[i][j]);
dfs(1);
printf("\n");
```

```
for (i = 1; i <= n; i++) {
if (reach[i])
count++;
}
if (count == n)
printf("\n Graph is connected");
else
printf("\n Graph is not connected");
return 0;
}
```