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

#define MAX_VERTICES 100

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

struct Graph {
    int vertices;
    struct Node*
adjacencyList[MAX_VERTICES];
};

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

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct
Graph*)malloc(sizeof(struct Graph));
    graph->vertices = vertices;
    for (int i = 0; i < vertices; ++i) {
        graph->adjacencyList[i] = NULL;
    }
    return graph;
}

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

void printAdjacencyList(struct Graph*
graph) {
    printf("Adjacency List:\n");
    for (int i =1; i <=graph->vertices; ++i) {
        struct Node* current =
graph->adjacencyList[i];
        printf("Vertex %d: ", i);
        while (current != NULL) {
            printf("%d -> ", current->data);
            current = current->next;
        }
        printf("NULL\n");
    }
}

void printAdjacencyMatrix(struct Graph*
graph) {
    printf("Adjacency Matrix:\n");
    for (int i =1; i <=graph->vertices; ++i) {
        for (int j =1; j <=graph->vertices; ++j)
{
            int isConnected = 0;
            struct Node* current =
graph->adjacencyList[i];
            while (current != NULL) {
                if (current->data == j) {
                    isConnected = 1;
                    break;
                }
                current = current->next;
            }
            printf("%d ", isConnected);
        }
        printf("\n");
    }
}

void DFSUtil(struct Graph* graph, int
vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

    struct Node* current =
graph->adjacencyList[vertex];
    while (current != NULL) {
        if (!visited[current->data]) {
            DFSUtil(graph, current->data,
visited);
        }
        current = current->next;
    }
}
```

```
void DFS(struct Graph* graph, int
startVertex) {
    printf("Depth First Search (DFS):\n");
    int visited[MAX_VERTICES] = {0};
    DFSUtil(graph, startVertex, visited);
    printf("\n");
}

void BFS(struct Graph* graph, int
startVertex) {
    printf("Breadth First Search (BFS):\n");
    int visited[MAX_VERTICES] = {0};
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;

    visited[startVertex] = 1;
    queue[rear++] = startVertex;

    while (front < rear) {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);

        struct Node* current =
graph->adjacencyList[currentVertex];
        while (current != NULL) {
            if (!visited[current->data]) {
                visited[current->data] = 1;
                queue[rear++] = current->data;
            }
            current = current->next;
        }
    }

    printf("\n");
}

int main() {
    int vertices, edges, src, dest;

    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    struct Graph* graph =
createGraph(vertices);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    for (int i=1; i<=edges;++i) {
        printf("Enter edge %d (source
destination): ", i);
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }

    printAdjacencyList(graph);
    printAdjacencyMatrix(graph);

    int startVertex;
    printf("Enter the starting vertex for DFS
and BFS: ");
    scanf("%d", &startVertex);

    DFS(graph, startVertex);
    BFS(graph, startVertex);

    free(graph);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
struct Node {
    int data;
    struct Node* next;
};

struct HashTable {
    struct Node* table[SIZE];
};

void initializeHashTable(struct HashTable*
ht) {
    for (int i = 0; i < SIZE; ++i) {
        ht->table[i] = NULL;
    }
}

int hashFunction(int key) {
    return key % SIZE;
}

void insert(struct HashTable* ht, int key) {
    int index = hashFunction(key);
    struct Node* newNode = (struct
Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        return;
    }
    newNode->data = key;
    newNode->next = NULL;
    if (ht->table[index] == NULL) {
        ht->table[index] = newNode;
    } else {
        newNode->next = ht->table[index];
        ht->table[index] = newNode;
    }
}

void displayHashTable(struct HashTable*
ht) {
    for (int i = 0; i < SIZE; ++i) {
        printf("Index %d:", i);
        struct Node* current = ht->table[i];
        while (current != NULL) {
            printf(" %d", current->data);
            current = current->next;
        }

        printf("\n");
    }
}

int main() {
    struct HashTable hashTable;
    initializeHashTable(&hashTable);
    insert(&hashTable, 5);
    insert(&hashTable, 15);
    insert(&hashTable, 25);
    insert(&hashTable, 7);
    insert(&hashTable, 17);
    displayHashTable(&hashTable);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
struct HashTable {
   int table[SIZE];
   int isOccupied[SIZE];
};
void initializeHashTable(struct HashTable*
ht) {
   for (int i = 0; i < SIZE; ++i) {
      ht->table[i] = -1;
      ht->isOccupied[i] = 0;
   }
}

int hashFunction(int key) {
   return key % SIZE;
}
int linearProbe(int index, int attempt) {
   return (index + attempt) % SIZE;
}
void insert(struct HashTable* ht, int key) {
   int index = hashFunction(key);
   int attempt = 0;
   while (ht->isOccupied[index] &&
ht->table[index] != key) {
      attempt++;
      index =
linearProbe(hashFunction(key), attempt);
   }

   ht->table[index] = key;
   ht->isOccupied[index] = 1;
}
void displayHashTable(struct HashTable*
ht) {
   for (int i = 0; i < SIZE; ++i) {
      printf("Index %d:", i);
      if (ht->isOccupied[i]) {
         printf(" %d", ht->table[i]);
      }
      printf("\n");
   }
}

int main() {
   struct HashTable hashTable;
   initializeHashTable(&hashTable);
   insert(&hashTable, 5);
   insert(&hashTable, 15);
   insert(&hashTable, 25);
   insert(&hashTable, 7);
   insert(&hashTable, 17);
   displayHashTable(&hashTable);
   return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
struct MemoryPool {
    struct Node* head;
    struct Node* tail;
};
void initializeMemoryPool(struct MemoryPool* mp) {
    mp->head = NULL;
    mp->tail = NULL;
}
struct Node* allocateMemory(struct MemoryPool* mp, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        return NULL;
    }

    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    if (mp->head == NULL) {
        mp->head = newNode;
        mp->tail = newNode;
    } else {
        newNode->prev = mp->tail;
        mp->tail->next = newNode;
        mp->tail = newNode;
    }
    return newNode;
}
void deallocateMemory(struct MemoryPool* mp, struct Node* node) {
    if (node == NULL) {
        return;
    }
    if (node == mp->head) {
        mp->head = node->next;
        if (mp->head != NULL) {
            mp->head->prev = NULL;
        }
    } else {
        if (node->prev != NULL) {
            node->prev->next = node->next;
        }
        if (node->next != NULL) {
            node->next->prev = node->prev;
        }
    }
    free(node);
}
void displayMemoryPool(struct MemoryPool* mp) {
    printf("Memory Pool Contents:");

    struct Node* current = mp->head;
    while (current != NULL) {
        printf(" %d", current->data);
        current = current->next;
    }

    printf("\n");
}

int main() {
    struct MemoryPool memoryPool;
    initializeMemoryPool(&memoryPool);
    struct Node* block1 = allocateMemory(&memoryPool, 10);
    struct Node* block2 = allocateMemory(&memoryPool, 20);
    struct Node* block3 = allocateMemory(&memoryPool, 30);
    displayMemoryPool(&memoryPool);
    deallocateMemory(&memoryPool, block2);
    displayMemoryPool(&memoryPool);

    return 0;
}
```

```
#include <stdio.h>
void insertionSort(int arr[],int n) {
int i,key,j;
for (i=1;i<n;i++) {
key=arr[i];
j=i-1;
while (j>=0 && arr[j]>key) {
arr[j+1]=arr[j];
j=j-1;
}
arr[j+1]=key;
}
}
void heapProcess(int arr[],int n,int i) {
int largest=i;
int left=2*i+1;
int right=2*i+2;
if(left<n && arr[left]>arr[largest]) {
largest=left;
}
if (right<n && arr[right]>arr[largest]) {
largest=right;
}
if (largest!=i) {
int temp=arr[i];
arr[i]=arr[largest];
arr[largest]=temp;
heapProcess(arr,n,largest);
}
}
void heapSort(int arr[],int n) {
for (int i=n/2-1;i>= 0;i--) {
heapProcess(arr,n,i);
}
for (int i=n-1;i>0;i--) {
int temp=arr[0];
arr[0]=arr[i];
arr[i]=temp;
heapProcess(arr,i,0);
}
}
void selectionSort(int arr[], int n) {
int i,j,minidx;
for (i = 0;i<n-1;i++) {
minidx=i;
for (j=i+1;j<n;j++) {
if (arr[j]<arr[minidx]) {
minidx=j;
```

```
}
}
int temp=arr[i];
arr[i]=arr[minidx];
arr[minidx]=temp;
}
}
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);
}
```

```
}
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++;
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}
int temp=arr[i+1];
arr[i+1]=arr[high];
arr[high]=temp;
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);
}
}
int main() {
FILE *file;
int data[1000],n=0;
int choice;
file = fopen("input.txt","r");
if (file == NULL) {
fprintf(stderr,"Error opening file.\n");
return 1;
}
while (fscanf(file,"%d",&data[n])==1) {
n++;
}
fclose(file);
printf("Choose a sorting method:\n");
printf("1. Insertion Sort\n");
printf("2. Heap Sort\n");
printf("3. Selection Sort\n");
printf("4. Merge Sort\n");
printf("5. Quick Sort\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
insertionSort(data,n);
break;
case 2:
heapSort(data,n);

break;
case 3:
selectionSort(data,n);
break;
case 4:
mergeSort(data,0,n-1);
break;
case 5:
quickSort(data,0,n-1);
break;
default:
printf("Invalid choice\n");
return 1;
}
printf("Sorted elements:\n");
for (int i = 0; i < n; i++) {
printf("%d\n", data[i]);
}
return 0;
```

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

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

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

struct Node* insertNode(struct Node* root, int value) {
    if (root == NULL)
        return createNode(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else if (value > root->data)
        root->right = insertNode(root->right, value);

    return root;
}

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

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

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

struct Node* findMin(struct Node* node) {
    while (node->left != NULL)
        node = node->left;
    return node;
}

struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    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;
}

void displayMenu() {
    printf("\nMenu:\n");
    printf("a. Insert a new node\n");
    printf("b. Inorder traversal\n");
    printf("c. Preorder traversal\n");
    printf("d. Postorder traversal\n");
    printf("e. Delete a node\n");
    printf("f. Exit\n");
    printf("Enter your choice: ");
}

int main() {
    struct Node* root = NULL;
    char choice;
    int value;

    do {
        displayMenu();
        scanf(" %c", &choice);

        switch (choice) {
            case 'a':
                printf("Enter the value to insert: ");

                scanf("%d", &value);
                root = insertNode(root, value);
                break;

            case 'b':
                printf("Inorder Traversal: ");
                inorderTraversal(root);
                printf("\n");
                break;

            case 'c':
                printf("Preorder Traversal: ");
                preorderTraversal(root);
                printf("\n");
                break;

            case 'd':
                printf("Postorder Traversal: ");
                postorderTraversal(root);
                printf("\n");
                break;

            case 'e':
                printf("Enter the value to delete: ");
                scanf("%d", &value);
                root = deleteNode(root, value);
                break;

            case 'f':
                printf("Exiting the program.\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 'f');

    return 0;
}
```