Lab-5

To implement queue operations for linear, circular and priority queue:

Linear queue:

Algorithm:

1. Initialize the queue data structure. This includes the queue array, the front and rear pointers, and the maximum capacity of the queue.
2. Repeat the following steps until the user chooses to exit:
   1. Get the choice from the user.
   2. If the choice is to enqueue a data item, then do the following:
      - Check if the queue is full. If it is, then print an error message.
      - Otherwise, add the data item to the queue at the index of the rear pointer. Increment the rear pointer.
   3. If the choice is to dequeue a data item, then do the following:
      - Check if the queue is empty. If it is, then print an error message.
      - Otherwise, remove the data item at the index of the front pointer. Increment the front pointer.
   4. If the choice is to display the queue, then do the following:
      - Print the contents of the queue.
   5. If the choice is to make the queue empty, then do the following:
      - Set the front and rear pointers to -1.

Source code:
```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

int isFull() {
  return rear == MAX - 1;
}
int isEmpty() {
  return front == -1 || front > rear;
}

void makeEmpty() {
  front = -1;
  rear = -1;
}

void enqueue(int data) {
  if (isFull()) {
    printf("Queue is full\n");
  } else {
    if (front == -1) {
      front = 0;
    }
    rear++;
    queue[rear] = data;
    printf("%d added to the queue\n", data);
```

```
    }
  }
void dequeue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    } else {
        printf("%d removed from the queue\n", queue[front]);
        front++;
    }
  }

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements are:\n");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, data;
    while (1) {
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4.
                    Make Empty\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                 break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                makeEmpty();
                printf("Queue is made empty\n");
                break;
            case 5:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

Output:

```
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 1
Enter data to enqueue: 5
5 added to the queue
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 1
Enter data to enqueue: 10
10 added to the queue
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 3
Queue elements are:
5 10
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 5
```

Circular queue:

```
#include <stdio.h>
#define MAX 5
int cqueue[MAX];
int front, rear;

int isFull() {
   return (front == 0 && rear == MAX - 1) || (front == rear + 1);
}

int isEmpty() {
   return front == -1;
}
void makeEmpty() {
   front = -1;
   rear = -1;
}
void enqueue(int data) {
   if (isFull()) {
      printf("Queue is full\n");
   } else {
      if (front == -1) {
         front = 0;
      }
      rear = (rear + 1) % MAX;
      cqueue[rear] = data;
      printf("%d added to the queue\n", data);
   }
}
void dequeue() {
   if (isEmpty()) {
      printf("Queue is empty\n");
   } else {
      printf("%d removed from the queue\n", cqueue[front]);
      if (front == rear) {
         front = -1;
         rear = -1;
      } else
         front = (front + 1) % MAX;
   }
}
void display() {
   if (isEmpty()) {
      printf("Queue is empty\n");
   } else {
      printf("Queue elements are:\n");
      int i = front;
      do {
         printf("%d ", cqueue[i]);
         i = (i + 1) % MAX;
      } while (i != rear);
   }
}
```

```
int main() {
   int choice, data;
   while (1) {
      printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Make Empty\n5. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            printf("Enter data to enqueue: ");
            scanf("%d", &data);
            enqueue(data);
            break;
         case 2:
            dequeue();
            break;
         case 3:
            display();
            break;
         case 4:
            makeEmpty();
            printf("Queue is made empty\n");
            break;
         case 5:
            return 0;
         default:
            printf("Invalid choice\n");
      }
   }
}
```

Output:

```
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 2
Queue is empty
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 5
```

Priority queue:

```
#include <stdio.h>
#define MAX 5
int pqueue[MAX];
int rear;

int isFull() {
   return rear == MAX - 1; }
int isEmpty() {
   return rear == -1;    }
void makeEmpty() {
   rear = -1;
}
void enqueue(int data) {
   if (isFull()) {
      printf("Queue is full\n");
   } else {
      int i;
      for (i = rear; i >= 0; i--) {
         if (data > pqueue[i]) {
            pqueue[i + 1] = pqueue[i];
```

```
            } else {
                break;
            }
        }
        pqueue[i + 1] = data;
        rear++;
        printf("%d added to the queue\n", data);
    }
}
void dequeue() {
    if (isEmpty())
        printf("Queue is empty\n");
else
        printf("%d removed from the queue\n", pqueue[rear]);
        rear--;
    }
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements are:\n");
        for (int i = rear; i >= 0; i--) {
            printf("%d ", pqueue[i]);
            }
        printf("\n");
    }
}
int main() {
    int choice, data;
    while (1) {
        printf("1. Enqueue\n2. Dequeue\n3. Display
                \n4. Make Empty\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;
            case 2:
                dequeue();      break;
            case 3:
                display();         break;
            case 4:
                makeEmpty();
                printf("Queue is made empty\n");
                break;
            case 5:
                return 0;
            default:
                printf("Invalid choice\n");
        } }}
```

__Output:__

```
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 1
Enter data to enqueue: 5
5 added to the queue
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 2
5 removed from the queue
1. Enqueue
2. Dequeue
3. Display
4. Make Empty
5. Exit
Enter your choice: 5
```

Lab-6

To implement list using array and linked list:

Algorithms for implementing a list using an array and a linked list in C:

1. Implementing a list using an array:
- Define a struct ArrayList with an integer array arr and an integer size.
- Define a function insert that takes a pointer to an ArrayList and an integer value as arguments.
    - If the size of the list is less than the maximum size, insert the value at the end of the array and increment the size.
    - Otherwise, print an error message indicating that the list is full.
- Define a function display that takes an ArrayList as an argument.
    - Iterate over the elements of the array and print each element.
- In the main function, create an ArrayList, insert some values, and display the contents of the list.

2. Implementing a list using a linked list:
- Define a struct Node with an integer data and a pointer to the next Node.
- Define a struct LinkedList with a pointer to the head Node.
- Define a function insert that takes a pointer to a LinkedList and an integer value as arguments.
    - Create a new Node with the given value and a null next pointer.
    - If the head of the list is null, set the head to the new Node.
    - Otherwise, iterate over the Nodes until the last Node is reached and set its next pointer to the new Node.
- Define a function display that takes a LinkedList as an argument.
    - Iterate over the Nodes of the list and print the data of each Node.
- In the main function, create a LinkedList, insert some values, and display the contents of the list.

Source code:

Implementing a list using an array:

```
#include <stdio.h>
#define MAX_SIZE 100

struct ArrayList {
    int arr[MAX_SIZE];
    int size;
};
void insert(struct ArrayList *list, int value) {
    if (list->size < MAX_SIZE)
        list->arr[list->size++] = value;
     else
        printf("List is full.\n");
}
void display(struct ArrayList list) {
    for (int i = 0; i < list.size; i++) {
        printf("%d\n", list.arr[i]);
    }
}
int main() {
    struct ArrayList list = { .size = 0 };
    insert(&list, 10);
```

```
   insert(&list, 20);
   insert(&list, 30);
   display(list);
return 0;
}
```

Output:

```
10

20

30
```

Implementing a list using a linked list:

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

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

   if (list->head == NULL) {
      list->head = newNode;
   } else {
      struct Node* temp = list->head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}
void display(struct LinkedList list) {
   struct Node* temp = list.head;
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}

int main() {
   struct LinkedList list = { .head = NULL };
   insert(&list, 10);
   insert(&list, 20);
   insert(&list, 30);
   display(list);
   return 0;
}
```

Output:

```
10 20 30
```

Lab-7

To implement stack and queue using linked list:

Algorithm:
1. Create two data structures, a stack and a queue.
2. Initialize the stack and queue.
3. Push the following elements onto the stack: 10, 20, and 30.
4. Pop two elements from the stack and print them.
5. Enqueue the following elements into the queue: 10, 20, and 30.
6. Dequeue two elements from the queue and print them.

Source code:
```
#include <stdio.h>
#include <stdlib.h>

// Node structure for a linked list
struct Node {
  int data;
  struct Node* next;
};

// Stack structure
struct Stack {
  struct Node* top;
};

// Queue structure
struct Queue {
  struct Node* front;
  struct Node* rear;
};

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

// Function to initialize a stack
void initStack(struct Stack* stack) {
  stack->top = NULL;
}

// Function to push an element onto a stack
void push(struct Stack* stack, int data) {
  struct Node* node = createNode(data);
  node->next = stack->top;
```

```
  stack->top = node;
}

// Function to pop an element from a stack
int pop(struct Stack* stack) {
  if (stack->top == NULL) {
    return -1;
  }

  int data = stack->top->data;
  struct Node* temp = stack->top;
  stack->top = stack->top->next;
  free(temp);
  return data;
}

// Function to initialize a queue
void initQueue(struct Queue* queue) {
  queue->front = NULL;
  queue->rear = NULL;
}

// Function to enqueue an element into a queue
void enqueue(struct Queue* queue, int data) {
  struct Node* node = createNode(data);

  if (queue->front == NULL) {
    queue->front = node;
    queue->rear = node;
  } else {
    queue->rear->next = node;
    queue->rear = node;
  }
}

// Function to dequeue an element from a queue
int dequeue(struct Queue* queue) {
  if (queue->front == NULL) {
    return -1;
  }

  int data = queue->front->data;
  struct Node* temp = queue->front;
  queue->front = queue->front->next;
  free(temp);

  if (queue->front == NULL) {
    queue->rear = NULL;
  }

  return data;
}
```

```c
// Driver code
int main() {
  struct Stack stack;
  initStack(&stack);

  push(&stack, 10);
  push(&stack, 20);
  push(&stack, 30);

  printf("Popped element from stack: %d\n", pop(&stack));
  printf("Popped element from stack: %d\n", pop(&stack));

  struct Queue queue;
  initQueue(&queue);

  enqueue(&queue, 10);
  enqueue(&queue, 20);
  enqueue(&queue, 30);

  printf("Dequeued element from queue: %d\n", dequeue(&queue));
  printf("Dequeued element from queue: %d\n", dequeue(&queue));

  return 0;
}
```

Output:

```
Popped element from stack: 30

Popped element from stack: 20

Dequeued element from queue: 10

Dequeued element from queue: 20
```

Lab-8

To implement sorting, searching and hashing algorithms:

Merge sort :

Algorithm:

1. If the array has only one element, then the array is already sorted.
2. Otherwise, divide the array into two halves.
3. Sort each half of the array recursively.
4. Merge the two sorted halves back together.

Source code:

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }
  while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }
while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
   }
}
```

```
void mergeSort(int arr[], int l, int r) {
   if (l < r) {
      int m = l + (r - l) / 2;
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

      merge(arr, l, m, r);
   }
}
void printArray(int arr[], int size) {
   int i;
   for (i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

int main() {
   int arr[] = { 38, 27, 43, 3, 9, 82, 10 };
   int arr_size = sizeof(arr) / sizeof(arr[0]);

   printf("Given array is \n");
   printArray(arr, arr_size);

   mergeSort(arr, 0, arr_size - 1);

   printf("\nSorted array is \n");
   printArray(arr, arr_size);
   return 0;
}
```

Output:

```
Given array is

38 27 43 3 9 82 10

Sorted array is

3 9 10 27 38 43 82
```

Hashing:

1. Define a function called hash() that takes an integer as input and returns the hash value of the integer.
2. The hash value is calculated by taking the integer and dividing it by a constant.
3. The main function of the program takes an integer as input and calls the hash() function to get the hash value of the integer.
4. The hash value is then printed to the console.

Source code:

```
#include <stdio.h>
int hash(int key) {
  return key % 7;
}
int main() {
  int value = 123;
  int hash_value = hash(value);
  printf("The hash value of %d is %d\n", value, hash_value);
  return 0;
}
```

Output:

```
The hash value of 123 is 4
```

Lab-9

To implement binary search tree and AVL tree:

Algorithm:

1. Binary Search Tree:
- Create a struct Node with data, left, and right pointers.
- Create a function createNode to allocate memory for a new node and initialize its values.
- Create a function insertBST to insert a new node into the binary search tree.
- In insertBST, if the root is NULL, create a new node and return it.
- If the value is less than the root's data, insert it into the left subtree.
- If the value is greater than the root's data, insert it into the right subtree.
- Create a function inorder to traverse the binary search tree in inorder and print its contents.
- In inorder, recursively traverse the left subtree, print the root's data, and recursively traverse the right subtree.
- Create the binary search tree by inserting nodes into it using insertBST.
- Print the contents of the binary search tree using inorder.
2. AVL Tree:
- Create a struct Node with data, left, right, and height pointers.
- Create a function createNode to allocate memory for a new node and initialize its values.
- Create a function insertAVL to insert a new node into the AVL tree.
- In insertAVL, if the root is NULL, create a new node and return it.
- If the value is less than the root's data, insert it into the left subtree.
- If the value is greater than the root's data, insert it into the right subtree.
- Update the height of the node.
- Calculate the balance factor of the node.
- If the balance factor is greater than 1 and the value is less than the root's left child's data, perform a right rotation.
- If the balance factor is less than -1 and the value is greater than the root's right child's data, perform a left rotation.
- If the balance factor is greater than 1 and the value is greater than the root's left child's data, perform a left rotation on the left child and then a right rotation on the root.
- If the balance factor is less than -1 and the value is less than the root's right child's data, perform a right rotation on the right child and then a left rotation on the root.
- Create a function inorder to traverse the AVL tree in inorder and print its contents.
- In inorder, recursively traverse the left subtree, print the root's data, and recursively traverse the right subtree.
- Create the AVL tree by inserting nodes into it using insertAVL.
- Print the contents of the AVL tree using inorder.

Source code:

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

struct Node {
   int data;
   struct Node* left;
```

```
    struct Node* right;
    int height;
};

int max(int a, int b) {
    return (a > b) ? a : b;
}

int height(struct Node* node) {
    if (node == NULL) {
        return 0;
    }
    return node->height;
}

int getBalance(struct Node* node) {
    if (node == NULL) {
        return 0;
    }
    return height(node->left) - height(node->right);
}

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->height = 1;
    return newNode;
}

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

struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

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

```
 return x;
}

struct Node* leftRotate(struct Node* x) {
   struct Node* y = x->right;
   struct Node* T2 = y->left;

   y->left = x;
   x->right = T2;

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

   return y;
}

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

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

   int balance = getBalance(node);

   if (balance > 1 && value < node->left->data) {
      return rightRotate(node);
   }
   if (balance < -1 && value > node->right->data) {
      return leftRotate(node);
   }
   if (balance > 1 && value > node->left->data) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
   }
   if (balance < -1 && value < node->right->data) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
   }

   return node;
}

void inorder(struct Node* root) {
   if (root != NULL) {
      inorder(root->left);
```

```
        printf("%d ", root->data);
        inorder(root->right);
    }
}


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

    printf("Binary Search Tree: ");
    inorder(rootBST);
    printf("\n");

    struct Node* rootAVL = NULL;
    rootAVL = insertAVL(rootAVL, 10);
    rootAVL = insertAVL(rootAVL, 20);
    rootAVL = insertAVL(rootAVL, 30);
    rootAVL = insertAVL(rootAVL, 40);
    rootAVL = insertAVL(rootAVL, 50);
    rootAVL = insertAVL(rootAVL, 25);

    printf("AVL Tree: ");
    inorder(rootAVL);
    printf("\n");

    return 0;
}
```

Output:

```
Binary Search Tree: 20 30 40 50 60 70 80

AVL Tree: 10 20 25 30 40 50
```

Lab-10

To implement searching, sorting and shortest path:

Algorithm:

1. Initialize the distance of all vertices to infinity and the distance of the source vertex to 0.
2. Create a set of visited vertices and mark all vertices as unvisited.
3. While there are unvisited vertices:
   a. Choose the vertex with the minimum distance from the source vertex.
   b. Mark the vertex as visited.
   c. For each unvisited neighbor of the vertex, calculate the distance from the source vertex through the current vertex.
   d. If the calculated distance is less than the current distance of the neighbor, update the distance of the neighbor.
4. Print the shortest distance and the path from the source vertex to each vertex.

Source code:

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

#define V 9

int minDistance(int dist[], bool visitedSet[]) {
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++) {
    if (visitedSet[v] == false && dist[v] <= min) {
      min = dist[v];
      min_index = v;
    }
  }

  return min_index;
}

void printPath(int parent[], int j) {
  if (parent[j] == -1) {
    return;
  }

  printPath(parent, parent[j]);
  printf("%d ", j);
}

void printSolution(int dist[], int parent[], int src, int dest) {
  printf("Vertex\tDistance\tPath\n");
  for (int i = 0; i < V; i++) {
```

```
        printf("%d\t%d\t\t%d ", i, dist[i], src);
        printPath(parent, i);
        printf("\n");
    }
    printf("Shortest path from %d to %d: ", src, dest);
    printPath(parent, dest);
    printf("\n");
}

void dijkstra(int graph[V][V], int src, int dest) {
    int dist[V];
    bool visitedSet[V];
    int parent[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        visitedSet[i] = false;
    }

    dist[src] = 0;
    parent[src] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, visitedSet);
        visitedSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (!visitedSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
                parent[v] = u;
            }
        }
    }

    printSolution(dist, parent, src, dest);
}

int main() {
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

 dijkstra(graph, 0, 4);
  return 0;
}
```

<u>Output:</u>

```
Vertex  Distance        Path
0       0               0
1       4               0 1
2       12              0 1 2
3       19              0 1 2 3
4       21              0 7 6 5 4
5       11              0 7 6 5
6       9               0 7 6
7       8               0 7
8       14              0 1 2 8
Shortest path from 0 to 4: 7 6 5 4
```