Linear Search (C)

```c
#include <stdio.h>

void linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key) {
         printf("Element found at index %d\n", i);
         return;
      }
   }
   printf("Element not found\n");
}

int main() {
   int n, key;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];

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

   printf("Enter the element to search for: ");
   scanf("%d", &key);

   linearSearch(arr, n, key);
   return 0;
}
```

Linear Search (Python)

```python
def linear_search(arr, key):
   for i in range(len(arr)):
      if arr[i] == key:
         print(f"Element found at index {i}")
         return
   print("Element not found")

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
   arr.append(int(input()))

key = int(input("Enter the element to search for: "))
linear_search(arr, key)
```

Binary Search (C)

```c
#include <stdio.h>

void binarySearch(int arr[], int n, int key) {
   int low = 0, high = n - 1;
   while (low <= high) {
      int mid = (low + high) / 2;
      if (arr[mid] == key) {
         printf("Element found at index %d\n", mid);
         return;
      } else if (arr[mid] < key) {
         low = mid + 1;
      } else {
         high = mid - 1;
      }
   }
   printf("Element not found\n");
}

int main() {
   int n, key;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];

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

   printf("Enter the element to search for: ");
   scanf("%d", &key);

   binarySearch(arr, n, key);
   return 0;
}
```

Binary Search (Python)

```python
def binary_search(arr, key):
   low, high = 0, len(arr) - 1
   while low <= high:
      mid = (low + high) // 2
      if arr[mid] == key:
         print(f"Element found at index {mid}")
         return
      elif arr[mid] < key:
         low = mid + 1
      else:
```

```
            high = mid - 1
    print("Element not found")

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements in sorted order:")
for _ in range(n):
    arr.append(int(input()))

key = int(input("Enter the element to search for: "))
binary_search(arr, key)
```

Selection Sort (C)

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIdx];
        arr[minIdx] = temp;
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

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

    selectionSort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Selection Sort (Python)

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
    arr.append(int(input()))

selection_sort(arr)
print("Sorted array:", arr)
```

Insertion Sort (C)

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

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

    insertionSort(arr, n);
```

```
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Insertion Sort(Python)

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
    arr.append(int(input()))

insertion_sort(arr)
print("Sorted array:", arr)
```

Quick Sort(C)

```
#include <stdio.h>

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

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

   quickSort(arr, 0, n - 1);

   printf("Sorted array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
   return 0;
}
```

Quick Sort(Python)

```
def quick_sort(arr):
   if len(arr) <= 1:
      return arr
   else:
      pivot = arr[len(arr) // 2]
      left = [x for x in arr if x < pivot]
      middle = [x for x in arr if x == pivot]
      right = [x for x in arr if x > pivot]
      return quick_sort(left) + middle + quick_sort(right)

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
   arr.append(int(input()))

arr = quick_sort(arr)
print("Sorted array:", arr)
```

Merge Sort(C)

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
  int n1 = mid - left + 1;
  int n2 = right - mid;

  int L[n1], R[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[left + i];
  for (int j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j];

  int i = 0, j = 0, k = left;

  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 left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;

    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
  }
}
```

```
int main() {
   int n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];

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

   mergeSort(arr, 0, n - 1);

   printf("Sorted array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
   return 0;
}
```

Merge Sort(Python)

```
def merge_sort(arr):
   if len(arr) > 1:
      mid = len(arr) // 2
      left_half = arr[:mid]
      right_half = arr[mid:]

      merge_sort(left_half)
      merge_sort(right_half)

      i = j = k = 0

      while i < len(left_half) and j < len(right_half):
         if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
         else:
            arr[k] = right_half[j]
            j += 1
         k += 1

      while i < len(left_half):
         arr[k] = left_half[i]
         i += 1
         k += 1
```

```
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
    arr.append(int(input()))

merge_sort(arr)
print("Sorted array:", arr)
```

Radix Sort(C)

```
#include <stdio.h>

int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

void countingSort(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
```

```
}

void radixSort(int arr[], int n) {
    int max = getMax(arr, n);

    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

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

    radixSort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Radix Sort(Python)

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
```

```
        count[index % 10] -= 1
        i -= 1

    for i in range(len(arr)):
        arr[i] = output[i]

def radix_sort(arr):
    max_element = max(arr)
    exp = 1
    while max_element // exp > 0:
        counting_sort(arr, exp)
        exp *= 10

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
    arr.append(int(input()))

radix_sort(arr)
print("Sorted array:", arr)
```

Singly Linked List(C)

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

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

struct Node* head = NULL;

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

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

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

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

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

void insertAtIndex(int data, int index) {
   if (index == 0) {
      insertAtBeginning(data);
      return;
   }

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;

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

   if (temp == NULL) {
      printf("Index out of range.\n");
      free(newNode);
      return;
   }

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

void deleteAtBeginning() {
   if (head == NULL) {
      printf("List is empty.\n");
      return;
   }
   struct Node* temp = head;
   head = head->next;
   free(temp);
}
```

```
void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;

    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

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

void deleteAtIndex(int index) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    if (index == 0) {
        deleteAtBeginning();
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;

    for (int i = 0; temp != NULL && i < index; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Index out of range.\n");
        return;
    }

    prev->next = temp->next;
    free(temp);
}
```

```
void traverse() {
   struct Node* temp = head;
   if (temp == NULL) {
      printf("List is empty.\n");
      return;
   }
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}

int main() {
   int choice, data, index;

   while (1) {
      printf("\nMenu:\n");
      printf("1. Create node\n");
      printf("2. Insert at beginning\n");
      printf("3. Insert at end\n");
      printf("4. Insert at index\n");
      printf("5. Delete at beginning\n");
      printf("6. Delete at end\n");
      printf("7. Delete at index\n");
      printf("8. Traverse\n");
      printf("9. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter data for new node: ");
            scanf("%d", &data);
            createNode(data);
            break;
         case 2:
            printf("Enter data to insert at beginning: ");
            scanf("%d", &data);
            insertAtBeginning(data);
            break;
         case 3:
            printf("Enter data to insert at end: ");
            scanf("%d", &data);
            insertAtEnd(data);
            break;
         case 4:
            printf("Enter data to insert at index: ");
            scanf("%d", &data);
```

```
            printf("Enter index: ");
            scanf("%d", &index);
            insertAtIndex(data, index);
            break;
        case 5:
            deleteAtBeginning();
            break;
        case 6:
            deleteAtEnd();
            break;
        case 7:
            printf("Enter index to delete: ");
            scanf("%d", &index);
            deleteAtIndex(index);
            break;
        case 8:
            traverse();
            break;
        case 9:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }
  return 0;
}
```

Singly Linked List(Python)

```
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None

class SinglyLinkedList:
  def __init__(self):
    self.head = None

  def create_node(self, data):
    new_node = Node(data)
    self.head = new_node

  def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

  def insert_at_end(self, data):
    new_node = Node(data)
```

```
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def insert_at_index(self, data, index):
        if index == 0:
            self.insert_at_beginning(data)
            return

        new_node = Node(data)
        temp = self.head
        for i in range(index - 1):
            if temp is None:
                print("Index out of range.")
                return
            temp = temp.next

        if temp is None:
            print("Index out of range.")
            return

        new_node.next = temp.next
        temp.next = new_node

    def delete_at_beginning(self):
        if self.head is None:
            print("List is empty.")
            return
        self.head = self.head.next

    def delete_at_end(self):
        if self.head is None:
            print("List is empty.")
            return
        temp = self.head
        prev = None
        while temp.next:
            prev = temp
            temp = temp.next
        if prev:
            prev.next = None
        else:
            self.head = None

    def delete_at_index(self, index):
        if self.head is None:
```

```
            print("List is empty.")
            return
        if index == 0:
            self.delete_at_beginning()
            return

        temp = self.head
        prev = None
        for i in range(index):
            if temp is None:
                print("Index out of range.")
                return
            prev = temp
            temp = temp.next

        if temp is None:
            print("Index out of range.")
            return

        prev.next = temp.next

    def traverse(self):
        temp = self.head
        if temp is None:
            print("List is empty.")
            return
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("NULL")

def main():
    sll = SinglyLinkedList()
    while True:
        print("\nMenu:")
        print("1. Create node")
        print("2. Insert at beginning")
        print("3. Insert at end")
        print("4. Insert at index")
        print("5. Delete at beginning")
        print("6. Delete at end")
        print("7. Delete at index")
        print("8. Traverse")
        print("9. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter data for new node: "))
            sll.create_node(data)
        elif choice == 2:
```

```
            data = int(input("Enter data to insert at beginning: "))
            sll.insert_at_beginning(data)
        elif choice == 3:
            data = int(input("Enter data to insert at end: "))
            sll.insert_at_end(data)
        elif choice == 4:
            data = int(input("Enter data to insert at index: "))
            index = int(input("Enter index: "))
            sll.insert_at_index(data, index)
        elif choice == 5:
            sll.delete_at_beginning()
        elif choice == 6:
            sll.delete_at_end()
        elif choice == 7:
            index = int(input("Enter index to delete: "))
            sll.delete_at_index(index)
        elif choice == 8:
            sll.traverse()
        elif choice == 9:
            break
        else:
            print("Invalid choice! Please try again.")

if __name__ == "__main__":
    main()
```

Doubly Linked List(C)

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

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

struct Node* head = NULL;

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

void insertAtBeginning(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

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

    if (head != NULL) {
        head->prev = newNode;
    }
    head = newNode;
}

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

    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        return;
    }

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

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

void insertAtIndex(int data, int index) {
    if (index == 0) {
        insertAtBeginning(data);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

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

    if (temp == NULL) {
        printf("Index out of range.\n");
        free(newNode);
        return;
    }
```

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

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

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;

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

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

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

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

void deleteAtIndex(int index) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    if (index == 0) {
        deleteAtBeginning();
```

```
        return;
    }

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

    if (temp == NULL) {
        printf("Index out of range.\n");
        return;
    }

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

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

void traverse() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    int choice, data, index;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Create node\n");
        printf("2. Insert at beginning\n");
        printf("3. Insert at end\n");
        printf("4. Insert at index\n");
        printf("5. Delete at beginning\n");
        printf("6. Delete at end\n");
        printf("7. Delete at index\n");
        printf("8. Traverse\n");
        printf("9. Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data for new node: ");
                scanf("%d", &data);
                createNode(data);
                break;
            case 2:
                printf("Enter data to insert at beginning: ");
                scanf("%d", &data);
                insertAtBeginning(data);
                break;
            case 3:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 4:
                printf("Enter data to insert at index: ");
                scanf("%d", &data);
                printf("Enter index: ");
                scanf("%d", &index);
                insertAtIndex(data, index);
                break;
            case 5:
                deleteAtBeginning();
                break;
            case 6:
                deleteAtEnd();
                break;
            case 7:
                printf("Enter index to delete: ");
                scanf("%d", &index);
                deleteAtIndex(index);
                break;
            case 8:
                traverse();
                break;
            case 9:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Doubly Linked List(Python)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def create_node(self, data):
        new_node = Node(data)
        self.head = new_node

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        new_node.prev = None
        if self.head is not None:
            self.head.prev = new_node
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            new_node.prev = None
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last

    def insert_at_index(self, data, index):
        if index == 0:
            self.insert_at_beginning(data)
            return

        new_node = Node(data)
        temp = self.head
        for i in range(index - 1):
            if temp is None:
                print("Index out of range.")
                return
            temp = temp.next

        if temp is None:
```

```
            print("Index out of range.")
            return

        new_node.next = temp.next
        new_node.prev = temp
        if temp.next is not None:
            temp.next.prev = new_node
        temp.next = new_node

    def delete_at_beginning(self):
        if self.head is None:
            print("List is empty.")
            return
        temp = self.head
        self.head = self.head.next
        if self.head is not None:
            self.head.prev = None
        temp = None

    def delete_at_end(self):
        if self.head is None:
            print("List is empty.")
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        if temp.prev is not None:
            temp.prev.next = None
        else:
            self.head = None
        temp = None

    def delete_at_index(self, index):
        if self.head is None:
            print("List is empty.")
            return
        if index == 0:
            self.delete_at_beginning()
            return

        temp = self.head
        for i in range(index):
            if temp is None:
                print("Index out of range.")
                return
            temp = temp.next

        if temp is None:
            print("Index out of range.")
            return
```

```
        if temp.next is not None:
            temp.next.prev = temp.prev
        if temp.prev is not None:
            temp.prev.next = temp.next
        temp = None

    def traverse(self):
        temp = self.head
        if temp is None:
            print("List is empty.")
            return
        while temp:
            print(temp.data, end=" <-> ")
            temp = temp.next
        print("NULL")

def main():
    dll = DoublyLinkedList()
    while True:
        print("\nMenu:")
        print("1. Create node")
        print("2. Insert at beginning")
        print("3. Insert at end")
        print("4. Insert at index")
        print("5. Delete at beginning")
        print("6. Delete at end")
        print("7. Delete at index")
        print("8. Traverse")
        print("9. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter data for new node: "))
            dll.create_node(data)
        elif choice == 2:
            data = int(input("Enter data to insert at beginning: "))
            dll.insert_at_beginning(data)
        elif choice == 3:
            data = int(input("Enter data to insert at end: "))
            dll.insert_at_end(data)
        elif choice == 4:
            data = int(input("Enter data to insert at index: "))
            index = int(input("Enter index: "))
            dll.insert_at_index(data, index)
        elif choice == 5:
            dll.delete_at_beginning()
        elif choice == 6:
            dll.delete_at_end()
        elif choice == 7:
```

```
            index = int(input("Enter index to delete: "))
            dll.delete_at_index(index)
        elif choice == 8:
            dll.traverse()
        elif choice == 9:
            break
        else:
            print("Invalid choice! Please try again.")

if __name__ == "__main__":
    main()
```

Circular Linked List(C)

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

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

struct Node* head = NULL;

void createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    if (head == NULL) {
        head = newNode;
        newNode->next = head; // Point to itself
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head; // Complete the circular link
    }
}

void insertAtBeginning(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (head == NULL) {
        head = newNode;
        newNode->next = head; // Point to itself
    } else {
        struct Node* temp = head;
```

```
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head; // Complete the circular link
        head = newNode; // Update head to new node
    }
}

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

    if (head == NULL) {
        head = newNode;
        newNode->next = head; // Point to itself
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head; // Complete the circular link
    }
}

void insertAtIndex(int data, int index) {
    if (index == 0) {
        insertAtBeginning(data);
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    struct Node* temp = head;
    for (int i = 0; temp != NULL && i < index - 1; i++) {
        temp = temp->next;
        if (temp == head) {
            printf("Index out of range.\n");
            free(newNode);
            return;
        }
    }

    if (temp == NULL) {
        printf("Index out of range.\n");
        free(newNode);
        return;
    }
```

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

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    if (temp->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* last = head;
        while (last->next != head) {
            last = last->next;
        }
        head = head->next;
        last->next = head; // Update last node's next to new head
        free(temp);
    }
}

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;

    while (temp->next != head) {
        prev = temp;
        temp = temp->next;
    }

    if (prev == NULL) {
        free(head);
        head = NULL; // List becomes empty
    } else {
        prev->next = head; // Update previous node's next
        free(temp);
    }
}

void deleteAtIndex(int index) {
```

```
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    if (index == 0) {
        deleteAtBeginning();
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;
    for (int i = 0; i < index; i++) {
        prev = temp;
        temp = temp->next;
        if (temp == head) {
            printf("Index out of range.\n");
            return;
        }
    }

    if (temp == head) {
        printf("Index out of range.\n");
        return;
    }

    prev->next = temp->next; // Bypass the node to delete
    free(temp);
}

void traverse() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(back to head)\n");
}

int main() {
    int choice, data, index;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Create node\n");
```

```
printf("2. Insert at beginning\n");
printf("3. Insert at end\n");
printf("4. Insert at index\n");
printf("5. Delete at beginning\n");
printf("6. Delete at end\n");
printf("7. Delete at index\n");
printf("8. Traverse\n");
printf("9. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter data for new node: ");
      scanf("%d", &data);
      createNode(data);
      break;
   case 2:
      printf("Enter data to insert at beginning: ");
      scanf("%d", &data);
      insertAtBeginning(data);
      break;
   case 3:
      printf("Enter data to insert at end: ");
      scanf("%d", &data);
      insertAtEnd(data);
      break;
   case 4:
      printf("Enter data to insert at index: ");
      scanf("%d", &data);
      printf("Enter index: ");
      scanf("%d", &index);
      insertAtIndex(data, index);
      break;
   case 5:
      deleteAtBeginning();
      break;
   case 6:
      deleteAtEnd();
      break;
   case 7:
      printf("Enter index to delete: ");
      scanf("%d", &index);
      deleteAtIndex(index);
      break;
   case 8:
      traverse();
      break;
   case 9:
      exit(0);
```

```
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Circular Linked List (Python)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def create_node(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head  # Point to itself
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head  # Complete the circular link

    def insert_at_beginning(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            new_node.next = self.head  # Point to itself
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head  # Complete the circular link
            self.head = new_node  # Update head to new node

    def insert_at_end(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
```

```
            new_node.next = self.head  # Point to itself
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head  # Complete the circular link

    def insert_at_index(self, data, index):
        if index == 0:
            self.insert_at_beginning(data)
            return

        new_node = Node(data)
        temp = self.head
        for i in range(index - 1):
            if temp is None:
                print("Index out of range.")
                return
            temp = temp.next
            if temp == self.head:
                print("Index out of range.")
                return

        new_node.next = temp.next
        temp.next = new_node

    def delete_at_beginning(self):
        if self.head is None:
            print("List is empty.")
            return

        temp = self.head
        if temp.next == self.head:
            self.head = None  # List becomes empty
        else:
            last = self.head
            while last.next != self.head:
                last = last.next
            self.head = self.head.next
            last.next = self.head  # Update last node's next

    def delete_at_end(self):
        if self.head is None:
            print("List is empty.")
            return

        temp = self.head
        prev = None
        while temp.next != self.head:
```

```
            prev = temp
            temp = temp.next

        if prev is None:
            self.head = None  # List becomes empty
        else:
            prev.next = self.head  # Update previous node's next

    def delete_at_index(self, index):
        if self.head is None:
            print("List is empty.")
            return

        if index == 0:
            self.delete_at_beginning()
            return

        temp = self.head
        prev = None
        for i in range(index):
            prev = temp
            temp = temp.next
            if temp == self.head:
                print("Index out of range.")
                return

        if temp == self.head:
            print("Index out of range.")
            return

        prev.next = temp.next  # Bypass the node to delete

    def traverse(self):
        if self.head is None:
            print("List is empty.")
            return

        temp = self.head
        while True:
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
                break
        print("(back to head)")

def main():
    cll = CircularLinkedList()
    while True:
        print("\nMenu:")
        print("1. Create node")
```

```
        print("2. Insert at beginning")
        print("3. Insert at end")
        print("4. Insert at index")
        print("5. Delete at beginning")
        print("6. Delete at end")
        print("7. Delete at index")
        print("8. Traverse")
        print("9. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter data for new node: "))
            cll.create_node(data)
        elif choice == 2:
            data = int(input("Enter data to insert at beginning: "))
            cll.insert_at_beginning(data)
        elif choice == 3:
            data = int(input("Enter data to insert at end: "))
            cll.insert_at_end(data)
        elif choice == 4:
            data = int(input("Enter data to insert at index: "))
            index = int(input("Enter index: "))
            cll.insert_at_index(data, index)
        elif choice == 5:
            cll.delete_at_beginning()
        elif choice == 6:
            cll.delete_at_end()
        elif choice == 7:
            index = int(input("Enter index to delete: "))
            cll.delete_at_index(index)
        elif choice == 8:
            cll.traverse()
        elif choice == 9:
            break
        else:
            print("Invalid choice! Please try again.")

if __name__ == "__main__":
    main()
```

Stacks_Array|

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

struct Stack {
    int arr[MAX];
    int top;
```

```
};

void initStack(struct Stack* s) {
   s->top = -1;
}

int isFull(struct Stack* s) {
   return s->top == MAX - 1;
}

int isEmpty(struct Stack* s) {
   return s->top == -1;
}

void push(struct Stack* s, int data) {
   if (isFull(s)) {
      printf("Stack Overflow!\n");
   } else {
      s->arr[++(s->top)] = data;
      printf("%d pushed to stack.\n", data);
   }
}

int pop(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack Underflow!\n");
      return -1;
   } else {
      return s->arr[(s->top)--];
   }
}

void peek(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack is empty.\n");
   } else {
      printf("Top element is: %d\n", s->arr[s->top]);
   }
}

void display(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack is empty.\n");
   } else {
      printf("Stack elements are:\n");
      for (int i = s->top; i >= 0; i--) {
         printf("%d\n", s->arr[i]);
      }
   }
}
```

```
int main() {
    struct Stack stack;
    initStack(&stack);
    int choice, data;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to push: ");
                scanf("%d", &data);
                push(&stack, data);
                break;
            case 2:
                data = pop(&stack);
                if (data != -1) {
                    printf("Popped element: %d\n", data);
                }
                break;
            case 3:
                peek(&stack);
                break;
            case 4:
                display(&stack);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Stacks_Linked List

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

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

struct Stack {
   struct Node* top;
};

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

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

void push(struct Stack* s, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = s->top;
   s->top = newNode;
   printf("%d pushed to stack.\n", data);
}

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

void peek(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack is empty.\n");
   } else {
      printf("Top element is: %d\n", s->top->data);
   }
}
```

```
void display(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack is empty.\n");
   } else {
      struct Node* temp = s->top;
      printf("Stack elements are:\n");
      while (temp != NULL) {
         printf("%d\n", temp->data);
         temp = temp->next;
      }
   }
}

int main() {
   struct Stack stack;
   initStack(&stack);
   int choice, data;

   while (1) {
      printf("\nMenu:\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Peek\n");
      printf("4. Display\n");
      printf("5. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter data to push: ");
            scanf("%d", &data);
            push(&stack, data);
            break;
         case 2:
            data = pop(&stack);
            if (data != -1) {
               printf("Popped element: %d\n", data);
            }
            break;
         case 3:
            peek(&stack);
            break;
         case 4:
            display(&stack);
            break;
         case 5:
            exit(0);
         default:
```

```
            printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Queue_Array

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

struct Queue {
    int arr[MAX];
    int front, rear;
};

void initQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
}

int isFull(struct Queue* q) {
    return (q->rear + 1) % MAX == q->front;
}

int isEmpty(struct Queue* q) {
    return q->front == -1;
}

void enqueue(struct Queue* q, int data) {
    if (isFull(q)) {
        printf("Queue Overflow!\n");
    } else {
        if (isEmpty(q)) {
            q->front = q->rear = 0;
        } else {
            q->rear = (q->rear + 1) % MAX;
        }
        q->arr[q->rear] = data;
        printf("%d enqueued to queue.\n", data);
    }
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow!\n");
        return -1;
    } else {
```

```
        int data = q->arr[q->front];
        if (q->front == q->rear) {
            q->front = q->rear = -1; // Queue is now empty
        } else {
            q->front = (q->front + 1) % MAX;
        }
        return data;
    }
}

void display(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements are:\n");
        int i = q->front;
        while (1) {
            printf("%d ", q->arr[i]);
            if (i == q->rear) break;
            i = (i + 1) % MAX;
        }
        printf("\n");
    }
}

int main() {
    struct Queue queue;
    initQueue(&queue);
    int choice, data;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(&queue, data);
                break;
            case 2:
                data = dequeue(&queue);
                if (data != -1) {
                    printf("Dequeued element: %d\n", data);
                }
```

```
            break;
        case 3:
            display(&queue);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }
  return 0;
}
```

Queue_LL

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

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

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

void initQueue(struct Queue* q) {
  q->front = NULL;
  q->rear = NULL;
}

int isEmpty(struct Queue* q) {
  return q->front == NULL;
}

void enqueue(struct Queue* q, int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->next = NULL;

  if (isEmpty(q)) {
    q->front = q->rear = newNode;
  } else {
    q->rear->next = newNode;
    q->rear = newNode;
  }
  printf("%d enqueued to queue.\n", data);
```

```
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow!\n");
        return -1;
    } else {
        struct Node* temp = q->front;
        int dequeuedData = temp->data;
        q->front = q->front->next;
        if (q->front == NULL) {
            q->rear = NULL; // Queue is now empty
        }
        free(temp);
        return dequeuedData;
    }
}

void display(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
    } else {
        struct Node* temp = q->front;
        printf("Queue elements are:\n");
        while (temp) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    struct Queue queue;
    initQueue(&queue);
    int choice, data;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
```

```
            enqueue(&queue, data);
            break;
        case 2:
            data = dequeue(&queue);
            if (data != -1) {
                printf("Dequeued element: %d\n", data);
            }
            break;
        case 3:
            display(&queue);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }
  return 0;
}
```

Tree Traversals

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

// Structure for a BST node
struct Node {
  int data;
  struct Node* left;
  struct Node* right;
};

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

// Function to insert a node in the BST
struct Node* insert(struct Node* root, int data) {
  if (root == NULL) {
    return createNode(data);
  }
  if (data < root->data) {
    root->left = insert(root->left, data);
  } else {
    root->right = insert(root->right, data);
```

```
    }
    return root;
}

// In-order traversal (Left, Root, Right)
void inOrder(struct Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

// Pre-order traversal (Root, Left, Right)
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Post-order traversal (Left, Right, Root)
void postOrder(struct Node* root) {
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}

// Main function to demonstrate the BST and traversals
int main() {
    struct Node* root = NULL;
    int choice, data;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. In-order Traversal\n");
        printf("3. Pre-order Traversal\n");
        printf("4. Post-order Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
```

```
            root = insert(root, data);
            break;
        case 2:
            printf("In-order Traversal: ");
            inOrder(root);
            printf("\n");
            break;
        case 3:
            printf("Pre-order Traversal: ");
            preOrder(root);
            printf("\n");
            break;
        case 4:
            printf("Post-order Traversal: ");
            postOrder(root);
            printf("\n");
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
      }
    }

    return 0;
}
```