

SEM 3\Exp1\SLL_implementation.c

```
1 // program to implement singly linked list
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 // Core structure of Node that forms Linked List
7 struct node
8 {
9     int data;
10    struct node *next;
11 };
12
13 // End of the linked list should not point to anything(NULL)
14 struct node *head = NULL;
15
16 // Function to insert a node at the beginning of the list
17 void insertFirst(int data)
18 {
19     struct node *new_node = (struct node *)malloc(sizeof(struct node));
20
21     new_node->data = data;
22     new_node->next = head;
23
24     head = new_node;
25 }
26
27 // Function to insert a node at the end of the list
28 void insertEnd(int data)
29 {
30     struct node *new_node = (struct node *)malloc(sizeof(struct node));
31
32     new_node->data = data;
33     new_node->next = NULL;
34
35     if (head == NULL)
36     {
37         head = new_node;
38         return;
39     }
40
41     struct node *temp = head;
42
43     while (temp->next != NULL)
44     {
45         temp = temp->next;
46     }
47
48     temp->next = new_node;
49 }
50
51 void insertPos(int data, int pos)
```

```
52 {
53     struct node *new_node = (struct node *)malloc(sizeof(struct node));
54
55     new_node->data = data;
56
57     int curr_pos = 0;
58     struct node *temp = head;
59
60     while (temp->next != NULL && curr_pos < pos - 1)
61     {
62         temp = temp->next;
63         curr_pos++;
64     }
65
66     new_node->next = temp->next;
67     temp->next = new_node;
68 }
69
70 void deleteFirst()
71 {
72     if (head == NULL)
73     {
74         printf("List is empty");
75         return;
76     }
77
78     struct node *temp = head;
79     head = head->next;
80     free(temp);
81 }
82
83 void deleteEnd()
84 {
85     if (head == NULL)
86     {
87         printf("List is empty");
88         return;
89     }
90
91     struct node *temp = head;
92     struct node *prev = NULL;
93
94     while (temp->next != NULL)
95     {
96         prev = temp;
97         temp = temp->next;
98     }
99
100     prev->next = NULL;
101     free(temp);
102 }
103
104 void deletePos(int pos)
105 {
```

```
106     if (head == NULL)
107     {
108         printf("List is empty");
109         return;
110     }
111
112     struct node *temp = head;
113     struct node *prev = NULL;
114     int curr_pos = 0;
115
116     while (temp->next != NULL && curr_pos < pos - 1)
117     {
118         prev = temp;
119         temp = temp->next;
120         curr_pos++;
121     }
122
123     prev->next = temp->next;
124     free(temp);
125 }
126
127 void display()
128 {
129     struct node *temp = head;
130
131     while (temp != NULL)
132     {
133         printf("%d -> ", temp->data);
134         temp = temp->next;
135     }
136     printf("NULL\n");
137 }
138
139 int main()
140 {
141     printf("Linked List creation and Manipulation\n");
142     printf("Enter from the following options:\n");
143     printf("1. Insert at the beginning of the list\n");
144     printf("2. Insert at the end of the list\n");
145     printf("3. Insert at a specific position in the list\n");
146     printf("4. Delete from the beginning of the list\n");
147     printf("5. Delete from the end of the list\n");
148     printf("6. Delete from a specific position in the list\n");
149     printf("7. Display the list\n");
150     printf("8. Exit\n");
151
152     int choice;
153     int data;
154     int pos;
155
156     while (1)
157     {
158         printf("Enter your choice: ");
159         scanf("%d", &choice);
```

```
160
161     switch (choice)
162     {
163     case 1:
164         printf("Enter the data to be inserted: ");
165         scanf("%d", &data);
166         insertFirst(data);
167         break;
168     case 2:
169         printf("Enter the data to be inserted: ");
170         scanf("%d", &data);
171         insertEnd(data);
172         break;
173     case 3:
174         printf("Enter the data to be inserted: ");
175         scanf("%d", &data);
176         printf("Enter the position to insert the data: ");
177         scanf("%d", &pos);
178         insertPos(data, pos);
179         break;
180     case 4:
181         deleteFirst();
182         break;
183     case 5:
184         deleteEnd();
185         break;
186     case 6:
187         printf("Enter the position to delete the data: ");
188         scanf("%d", &pos);
189         deletePos(pos);
190         break;
191     case 7:
192         display();
193         break;
194     case 8:
195         exit(0);
196     default:
197         printf("Invalid choice");
198         break;
199     }
200 }
201
202 return 0;
203 }
```

```
● @NyanAditya → /workspaces/Chamber_of_C (main) $ cd  
Linked List creation and Manipulation  
Enter from the following options:  
1. Insert at the beginning of the list  
2. Insert at the end of the list  
3. Insert at a specific position in the list  
4. Delete from the beginning of the list  
5. Delete from the end of the list  
6. Delete from a specific position in the list  
7. Display the list  
8. Exit  
Enter your choice: 1  
Enter the data to be inserted: 15  
Enter your choice: 1  
Enter the data to be inserted: 42  
Enter your choice: 1  
Enter the data to be inserted: 96  
Enter your choice: 7  
96 -> 42 -> 15 -> NULL  
Enter your choice: 2  
Enter the data to be inserted: 4  
Enter your choice: 2  
Enter the data to be inserted: 7  
Enter your choice: 2  
Enter the data to be inserted: 1  
Enter your choice: 7  
96 -> 42 -> 15 -> 4 -> 7 -> 1 -> NULL  
Enter your choice: 4  
Enter your choice: 7  
42 -> 15 -> 4 -> 7 -> 1 -> NULL  
Enter your choice: 5  
Enter your choice: 7  
42 -> 15 -> 4 -> 7 -> NULL  
Enter your choice: 6  
Enter the position to delete the data: 2  
Enter your choice: 7  
42 -> 4 -> 7 -> NULL  
Enter your choice: 3  
Enter the data to be inserted: 69  
Enter the position to insert the data: 3  
Enter your choice: 7  
42 -> 4 -> 7 -> 69 -> NULL  
Enter your choice: 8
```

```
○ @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp1
```

SEM 3\Exp2\DLL_implementation.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Node structure for the doubly linked list
5  struct Node
6  {
7      int data;
8      struct Node *prev;
9      struct Node *next;
10 };
11
12 // Insert at the end of the doubly linked list
13 void insert(struct Node **head_ref, int new_data)
14 {
15     struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
16     struct Node *last = *head_ref;
17     new_node->data = new_data;
18     new_node->next = NULL;
19
20     if (*head_ref == NULL)
21     {
22         new_node->prev = NULL;
23         *head_ref = new_node;
24         return;
25     }
26
27     while (last->next != NULL)
28         last = last->next;
29
30     last->next = new_node;
31     new_node->prev = last;
32 }
33
34 // Display the doubly linked list
35 void display(struct Node *node)
36 {
37     struct Node *last;
38     printf("Traversal in forward direction:\n");
39     while (node != NULL)
40     {
41         printf("%d ", node->data);
42         last = node;
43         node = node->next;
44     }
45
46     printf("\nTraversal in reverse direction:\n");
47     while (last != NULL)
48     {
49         printf("%d ", last->data);
50         last = last->prev;
51     }
```

```
52     printf("\n");
53 }
54
55 // Delete a node from the doubly linked list
56 void deleteNode(struct Node **head_ref, int key)
57 {
58     struct Node *temp = *head_ref;
59
60     if (*head_ref == NULL)
61         return;
62
63     while (temp != NULL && temp->data != key)
64         temp = temp->next;
65
66     if (temp == NULL)
67         return;
68
69     if (*head_ref == temp)
70         *head_ref = temp->next;
71
72     if (temp->next != NULL)
73         temp->next->prev = temp->prev;
74
75     if (temp->prev != NULL)
76         temp->prev->next = temp->next;
77
78     free(temp);
79 }
80
81 // Search for a key in the doubly linked list
82 void search(struct Node *head, int key)
83 {
84     struct Node *temp = head;
85     int pos = 0;
86     while (temp != NULL)
87     {
88         if (temp->data == key)
89         {
90             printf("Element %d found at position %d\n", key, pos);
91             return;
92         }
93         temp = temp->next;
94         pos++;
95     }
96     printf("Element %d not found in the list\n", key);
97 }
98
99 // Count the number of nodes in the doubly linked list
100 int count(struct Node *head)
101 {
102     int count = 0;
103     struct Node *temp = head;
104     while (temp != NULL)
105     {
```

```
106         count++;
107         temp = temp->next;
108     }
109     return count;
110 }
111
112 int main()
113 {
114     struct Node *head = NULL;
115     int choice, value, key;
116
117     printf("\nDoubly Linked List Operations:\n");
118     printf("1. Insert\n");
119     printf("2. Display\n");
120     printf("3. Delete\n");
121     printf("4. Search\n");
122     printf("5. Count\n");
123     printf("6. Exit\n");
124
125     while (1)
126     {
127         printf("Enter your choice: ");
128         scanf("%d", &choice);
129
130         switch (choice)
131         {
132             case 1:
133                 printf("Enter the value to insert: ");
134                 scanf("%d", &value);
135                 insert(&head, value);
136                 printf("\n");
137                 break;
138             case 2:
139                 display(head);
140                 printf("\n");
141                 break;
142             case 3:
143                 printf("Enter the value to delete: ");
144                 scanf("%d", &key);
145                 deleteNode(&head, key);
146                 printf("\n");
147                 break;
148             case 4:
149                 printf("Enter the value to search: ");
150                 scanf("%d", &key);
151                 search(head, key);
152                 printf("\n");
153                 break;
154             case 5:
155                 printf("The number of nodes in the list: %d\n", count(head));
156                 printf("\n");
157                 break;
158             case 6:
159                 exit(0);
```



```
160         default:
161             printf("Invalid choice!\n");
162         }
163     }
164
165     return 0;
166 }
167
```

Doubly Linked List Operations:

1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit

Enter your choice: 1

Enter the value to insert: 84

Enter your choice: 1

Enter the value to insert: 41

Enter your choice: 1

Enter the value to insert: 25

Enter your choice: 1

Enter the value to insert: 2

Enter your choice: 2

Traversal in forward direction:

84 41 25 2

Traversal in reverse direction:

2 25 41 84

Enter your choice: 3

Enter the value to delete: 41

Enter your choice: 5

The number of nodes in the list: 3

Enter your choice: 4

Enter the value to search: 84

Element 84 found at position 0

Enter your choice: 6

SEM 3\Exp3\CSLL.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Node structure for the circular linked list
5  struct Node
6  {
7      int data;
8      struct Node *next;
9  };
10
11 // Insert a new node at the end of the circular linked list
12 void insert(struct Node **head_ref, int new_data)
13 {
14     struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
15     struct Node *temp = *head_ref;
16     new_node->data = new_data;
17     new_node->next = *head_ref;
18
19     if (*head_ref == NULL)
20     {
21         new_node->next = new_node;
22         *head_ref = new_node;
23         return;
24     }
25
26     while (temp->next != *head_ref)
27         temp = temp->next;
28
29     temp->next = new_node;
30 }
31
32 // Display the circular linked list
33 void display(struct Node *head)
34 {
35     struct Node *temp = head;
36     if (head != NULL)
37     {
38         do
39         {
40             printf("%d ", temp->data);
41             temp = temp->next;
42         } while (temp != head);
43         printf("\n");
44     }
45     else
46     {
47         printf("List is empty.\n");
48     }
49 }
50
51 // Delete a node with a specific value from the circular linked list
```

```
52 void deleteNode(struct Node **head_ref, int key)
53 {
54     if (*head_ref == NULL)
55         return;
56
57     struct Node *temp = *head_ref, *prev;
58
59     // If the node to be deleted is the head
60     if (temp->data == key && temp->next == *head_ref)
61     {
62         *head_ref = NULL;
63         free(temp);
64         return;
65     }
66
67     // If the node to be deleted is the head and the list has more than one node
68     if (temp->data == key)
69     {
70         while (temp->next != *head_ref)
71             temp = temp->next;
72         temp->next = (*head_ref)->next;
73         free(*head_ref);
74         *head_ref = temp->next;
75         return;
76     }
77
78     // If the node to be deleted is not the head
79     prev = temp;
80     while (temp->next != *head_ref && temp->data != key)
81     {
82         prev = temp;
83         temp = temp->next;
84     }
85
86     if (temp->data == key)
87     {
88         prev->next = temp->next;
89         free(temp);
90     }
91 }
92
93 void search(struct Node *head, int key)
94 {
95     struct Node *temp = head;
96     int pos = 0;
97
98     if (head == NULL)
99     {
100         printf("List is empty.\n");
101         return;
102     }
103
104     do
105     {
```

```
106         if (temp->data == key)
107         {
108             printf("Element %d found at position %d\n", key, pos);
109             return;
110         }
111         temp = temp->next;
112         pos++;
113     } while (temp != head);
114
115     printf("Element %d not found in the list\n", key);
116 }
117
118 int count(struct Node *head)
119 {
120     int count = 0;
121     struct Node *temp = head;
122
123     if (head == NULL)
124         return 0;
125
126     do
127     {
128         count++;
129         temp = temp->next;
130     } while (temp != head);
131
132     return count;
133 }
134
135 int main()
136 {
137     struct Node *head = NULL;
138     int choice, value, key;
139
140     printf("\nCircular Linked List Operations:\n");
141     printf("1. Insert\n");
142     printf("2. Display\n");
143     printf("3. Delete\n");
144     printf("4. Search\n");
145     printf("5. Count\n");
146     printf("6. Exit\n");
147
148     while (1)
149     {
150
151         printf("Enter your choice: ");
152         scanf("%d", &choice);
153
154         switch (choice)
155         {
156             case 1:
157                 printf("Enter the value to insert: ");
158                 scanf("%d", &value);
159                 insert(&head, value);
```

```
160         printf("\n");
161         break;
162     case 2:
163         display(head);
164         printf("\n");
165         break;
166     case 3:
167         printf("Enter the value to delete: ");
168         scanf("%d", &key);
169         deleteNode(&head, key);
170         printf("\n");
171         break;
172     case 4:
173         printf("Enter the value to search: ");
174         scanf("%d", &key);
175         search(head, key);
176         printf("\n");
177         break;
178     case 5:
179         printf("The number of nodes in the list: %d\n", count(head));
180         printf("\n");
181         break;
182     case 6:
183         exit(0);
184     default:
185         printf("Invalid choice!\n");
186     }
187 }
188
189 return 0;
190 }
191
```

Circular Linked List Operations:

1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit

Enter your choice: 1

Enter the value to insert: 25

Enter your choice: 1

Enter the value to insert: 50

Enter your choice: 1

Enter the value to insert: 75

Enter your choice: 1

Enter the value to insert: 100

Enter your choice: 2

25 50 75 100

Enter your choice: 5

The number of nodes in the list: 4

Enter your choice: 4

Enter the value to search: 75

Element 75 found at position 2

Enter your choice: 4

Enter the value to search: 25

Element 25 found at position 0

Enter your choice: 3

Enter the value to delete: 50

Enter your choice: 2

25 75 100

Enter your choice: 6

SEM 3\Exp4\CDLL.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Node structure for the circular doubly linked list
5  struct Node
6  {
7      int data;
8      struct Node *next;
9      struct Node *prev;
10 };
11
12 // Insert a node at the end of the circular doubly linked list
13 void insert(struct Node **head_ref, int new_data)
14 {
15     struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
16     new_node->data = new_data;
17
18     if (*head_ref == NULL)
19     {
20         new_node->next = new_node;
21         new_node->prev = new_node;
22         *head_ref = new_node;
23         return;
24     }
25
26     struct Node *last = (*head_ref)->prev;
27
28     new_node->next = *head_ref;
29     (*head_ref)->prev = new_node;
30     new_node->prev = last;
31     last->next = new_node;
32 }
33
34 // Display the circular doubly linked list
35 void display(struct Node *head)
36 {
37     if (head == NULL)
38     {
39         printf("List is empty.\n");
40         return;
41     }
42
43     struct Node *temp = head;
44     printf("Traversal in forward direction:\n");
45     do
46     {
47         printf("%d ", temp->data);
48         temp = temp->next;
49     } while (temp != head);
50     printf("\n");
51 }
```



```
52     printf("Traversal in reverse direction:\n");
53     temp = head->prev;
54     do
55     {
56         printf("%d ", temp->data);
57         temp = temp->prev;
58     } while (temp->next != head);
59     printf("\n");
60 }
61
62 // Delete a node from the circular doubly linked list
63 void deleteNode(struct Node **head_ref, int key)
64 {
65     if (*head_ref == NULL)
66         return;
67
68     struct Node *current = *head_ref;
69
70     while (current->data != key)
71     {
72         current = current->next;
73         if (current == *head_ref)
74         {
75             printf("Element %d not found in the list.\n", key);
76             return;
77         }
78     }
79
80     if (current->next == *head_ref && current->prev == *head_ref)
81     {
82         *head_ref = NULL;
83         free(current);
84         return;
85     }
86
87     if (current == *head_ref)
88     {
89         struct Node *last = (*head_ref)->prev;
90         *head_ref = current->next;
91         last->next = *head_ref;
92         (*head_ref)->prev = last;
93         free(current);
94         return;
95     }
96
97     current->prev->next = current->next;
98     current->next->prev = current->prev;
99
100     free(current);
101 }
102
103 void search(struct Node *head, int key)
104 {
105     if (head == NULL)
```

```
106     {
107         printf("List is empty.\n");
108         return;
109     }
110
111     struct Node *temp = head;
112     int pos = 0;
113
114     do
115     {
116         if (temp->data == key)
117         {
118             printf("Element %d found at position %d\n", key, pos);
119             return;
120         }
121         temp = temp->next;
122         pos++;
123     } while (temp != head);
124
125     printf("Element %d not found in the list\n", key);
126 }
127
128 int count(struct Node *head)
129 {
130     if (head == NULL)
131         return 0;
132
133     struct Node *temp = head;
134     int count = 0;
135
136     do
137     {
138         count++;
139         temp = temp->next;
140     } while (temp != head);
141
142     return count;
143 }
144
145 int main()
146 {
147     struct Node *head = NULL;
148     int choice, value, key;
149
150     printf("\nCircular Doubly Linked List Operations:\n");
151     printf("1. Insert\n");
152     printf("2. Display\n");
153     printf("3. Delete\n");
154     printf("4. Search\n");
155     printf("5. Count\n");
156     printf("6. Exit\n");
157
158     while (1)
159     {
```

```
160
161     printf("Enter your choice: ");
162     scanf("%d", &choice);
163
164     switch (choice)
165     {
166     case 1:
167         printf("Enter the value to insert: ");
168         scanf("%d", &value);
169         insert(&head, value);
170         printf("\n");
171         break;
172     case 2:
173         display(head);
174         printf("\n");
175         break;
176     case 3:
177         printf("Enter the value to delete: ");
178         scanf("%d", &key);
179         deleteNode(&head, key);
180         printf("\n");
181         break;
182     case 4:
183         printf("Enter the value to search: ");
184         scanf("%d", &key);
185         search(head, key);
186         printf("\n");
187         break;
188     case 5:
189         printf("The number of nodes in the list: %d\n", count(head));
190         printf("\n");
191         break;
192     case 6:
193         exit(0);
194     default:
195         printf("Invalid choice!\n");
196     }
197 }
198
199 return 0;
200 }
201
```

Circular Doubly Linked List Operations:

1. Insert
2. Display
3. Delete
4. Search
5. Count
6. Exit

Enter your choice: 1

Enter the value to insert: 45

Enter your choice: 1

Enter the value to insert: 78

Enter your choice: 1

Enter the value to insert: 94

Enter your choice: 1

Enter the value to insert: 32

Enter your choice: 2

Traversal in forward direction:

45 78 94 32

Traversal in reverse direction:

32 94 78 45

Enter your choice: 5

The number of nodes in the list: 4

Enter your choice: 4

Enter the value to search: 78

Element 78 found at position 1

Enter your choice: 3

Enter the value to delete: 78

Enter your choice: 2

Traversal in forward direction:

45 94 32

Traversal in reverse direction:

32 94 45

Enter your choice: 6

SEM 3\Exp5\Stack_ARR.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX 100 // Maximum size of the stack
5
6  // Stack structure using arrays
7  struct StackArray
8  {
9      int top;
10     int arr[MAX];
11 };
12
13 // Function to create a stack
14 struct StackArray *createStack()
15 {
16     struct StackArray *stack = (struct StackArray *)malloc(sizeof(struct
StackArray));
17     stack->top = -1; // Initialize the top index
18     return stack;
19 }
20
21 // Check if the stack is full
22 int isFull(struct StackArray *stack)
23 {
24     return stack->top == MAX - 1;
25 }
26
27 // Check if the stack is empty
28 int isEmpty(struct StackArray *stack)
29 {
30     return stack->top == -1;
31 }
32
33 // Push an element onto the stack
34 void push(struct StackArray *stack, int value)
35 {
36     if (isFull(stack))
37     {
38         printf("Stack overflow!\n");
39         return;
40     }
41     stack->arr[++stack->top] = value;
42     printf("%d pushed onto stack\n", value);
43 }
44
45 // Pop an element from the stack
46 int pop(struct StackArray *stack)
47 {
48     if (isEmpty(stack))
49     {
50         printf("Stack underflow!\n");
51         return -1;
```

```
52     }
53     return stack->arr[stack->top--];
54 }
55
56 // Peek at the top element of the stack
57 int peek(struct StackArray *stack)
58 {
59     if (isEmpty(stack))
60     {
61         printf("Stack is empty!\n");
62         return -1;
63     }
64     return stack->arr[stack->top];
65 }
66
67 // Display the stack
68 void display(struct StackArray *stack)
69 {
70     if (isEmpty(stack))
71     {
72         printf("Stack is empty!\n");
73         return;
74     }
75     printf("Stack elements: ");
76     for (int i = stack->top; i >= 0; i--)
77     {
78         printf("%d ", stack->arr[i]);
79     }
80     printf("\n");
81 }
82
83 int main()
84 {
85     struct StackArray *stack = createStack();
86     int choice, value;
87
88     printf("\nStack Operations (Array Implementation):\n");
89     printf("1. Push\n");
90     printf("2. Pop\n");
91     printf("3. Peek\n");
92     printf("4. Display\n");
93     printf("5. Exit\n");
94
95     while (1)
96     {
97
98         printf("Enter your choice: ");
99         scanf("%d", &choice);
100
101         switch (choice)
102         {
103             case 1:
104                 printf("Enter the value to push: ");
105                 scanf("%d", &value);
```

```
106         push(stack, value);
107         printf("\n");
108         break;
109     case 2:
110         value = pop(stack);
111         if (value != -1)
112             printf("Popped value: %d\n", value);
113
114         printf("\n");
115         break;
116     case 3:
117         value = peek(stack);
118         if (value != -1)
119             printf("Top value: %d\n", value);
120
121         printf("\n");
122         break;
123     case 4:
124         display(stack);
125         printf("\n");
126         break;
127     case 5:
128         free(stack);
129         exit(0);
130     default:
131         printf("Invalid choice!\n");
132     }
133 }
134
135 return 0;
136 }
137
```

Stack Operations (Array Implementation):

1. Push
2. Pop
3. Peek
4. Display
5. Exit

Enter your choice: 4

Stack is empty!

Enter your choice: 1

Enter the value to push: 71

71 pushed onto stack

Enter your choice: 1

Enter the value to push: 73

73 pushed onto stack

Enter your choice: 1

Enter the value to push: 2

2 pushed onto stack

Enter your choice: 1

Enter the value to push: 63

63 pushed onto stack

Enter your choice: 4

Stack elements: 63 2 73 71

Enter your choice: 3

Top value: 63

Enter your choice: 2

Popped value: 63

Enter your choice: 2

Popped value: 2

Enter your choice: 4

Stack elements: 73 71

Enter your choice: 3

Top value: 73

Enter your choice: 5

SEM 3\Exp5\Stack_LL.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Node structure for the linked list
5  struct Node
6  {
7      int data;
8      struct Node *next;
9  };
10
11 // Stack structure using linked lists
12 struct StackLinkedList
13 {
14     struct Node *top;
15 };
16
17 // Function to create a stack
18 struct StackLinkedList *createStack()
19 {
20     struct StackLinkedList *stack = (struct StackLinkedList *)malloc(sizeof(struct
StackLinkedList));
21     stack->top = NULL; // Initialize the top pointer
22     return stack;
23 }
24
25 // Check if the stack is empty
26 int isEmpty(struct StackLinkedList *stack)
27 {
28     return stack->top == NULL;
29 }
30
31 // Push an element onto the stack
32 void push(struct StackLinkedList *stack, int value)
33 {
34     struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
35     new_node->data = value;
36     new_node->next = stack->top;
37     stack->top = new_node;
38     printf("%d pushed onto stack\n", value);
39 }
40
41 // Pop an element from the stack
42 int pop(struct StackLinkedList *stack)
43 {
44     if (isEmpty(stack))
45     {
46         printf("Stack underflow!\n");
47         return -1;
48     }
49     struct Node *temp = stack->top;
50     int popped_value = temp->data;
51     stack->top = stack->top->next;
```

```
52     free(temp);
53     return popped_value;
54 }
55
56 // Peek at the top element of the stack
57 int peek(struct StackLinkedList *stack)
58 {
59     if (isEmpty(stack))
60     {
61         printf("Stack is empty!\n");
62         return -1;
63     }
64     return stack->top->data;
65 }
66
67 // Display the stack
68 void display(struct StackLinkedList *stack)
69 {
70     if (isEmpty(stack))
71     {
72         printf("Stack is empty!\n");
73         return;
74     }
75     struct Node *temp = stack->top;
76     printf("Stack elements: ");
77     while (temp != NULL)
78     {
79         printf("%d ", temp->data);
80         temp = temp->next;
81     }
82     printf("\n");
83 }
84
85 int main()
86 {
87     struct StackLinkedList *stack = createStack();
88     int choice, value;
89
90     printf("\nStack Operations (Linked List Implementation):\n");
91     printf("1. Push\n");
92     printf("2. Pop\n");
93     printf("3. Peek\n");
94     printf("4. Display\n");
95     printf("5. Exit\n");
96
97     while (1)
98     {
99
100         printf("Enter your choice: ");
101         scanf("%d", &choice);
102
103         switch (choice)
104         {
105             case 1:
```

```
106     printf("Enter the value to push: ");
107     scanf("%d", &value);
108     push(stack, value);
109     printf("\n");
110     break;
111 case 2:
112     value = pop(stack);
113     if (value != -1)
114         printf("Popped value: %d\n", value);
115
116     printf("\n");
117     break;
118 case 3:
119     value = peek(stack);
120     if (value != -1)
121         printf("Top value: %d\n", value);
122
123     printf("\n");
124     break;
125 case 4:
126     display(stack);
127     printf("\n");
128     break;
129 case 5:
130     // Free linked list nodes (cleanup)
131     while (!isEmpty(stack))
132     {
133         pop(stack);
134     }
135     free(stack);
136     exit(0);
137 default:
138     printf("Invalid choice!\n");
139 }
140 }
141
142 return 0;
143 }
144
```

● @NyanAditya → /workspaces/Chamber_of_C (main) \$ cd

Stack Operations (Linked List Implementation):

1. Push
2. Pop
3. Peek
4. Display
5. Exit

Enter your choice: 4

Stack is empty!

Enter your choice: 1

Enter the value to push: 5

5 pushed onto stack

Enter your choice: 1

Enter the value to push: 94

94 pushed onto stack

Enter your choice: 1

Enter the value to push: 98

98 pushed onto stack

Enter your choice: 1

Enter the value to push: 42

42 pushed onto stack

Enter your choice: 3

Top value: 42

Enter your choice: 4

Stack elements: 42 98 94 5

Enter your choice: 2

Popped value: 42

Enter your choice: 2

Popped value: 98

Enter your choice: 4

Stack elements: 94 5

Enter your choice: 5

○ @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp5

SEM 3\Exp6\Queue_ARR.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX 100 // Maximum size of the queue
5
6  // Queue structure using arrays
7  struct QueueArray
8  {
9      int front, rear;
10     int arr[MAX];
11 };
12
13 // Function to create a queue
14 struct QueueArray *createQueue()
15 {
16     struct QueueArray *queue = (struct QueueArray *)malloc(sizeof(struct
QueueArray));
17     queue->front = -1;
18     queue->rear = -1;
19     return queue;
20 }
21
22 // Check if the queue is full
23 int isFull(struct QueueArray *queue)
24 {
25     return queue->rear == MAX - 1;
26 }
27
28 // Check if the queue is empty
29 int isEmpty(struct QueueArray *queue)
30 {
31     return queue->front == -1 || queue->front > queue->rear;
32 }
33
34 // Enqueue an element into the queue
35 void enqueue(struct QueueArray *queue, int value)
36 {
37     if (isFull(queue))
38     {
39         printf("Queue overflow!\n");
40         return;
41     }
42     if (isEmpty(queue))
43     {
44         queue->front = 0; // Initialize front if queue was empty
45     }
46     queue->arr[++queue->rear] = value;
47     printf("%d enqueued to queue\n", value);
48 }
49
50 // Dequeue an element from the queue
51 int dequeue(struct QueueArray *queue)
```

```
52 {
53     if (isEmpty(queue))
54     {
55         printf("Queue underflow!\n");
56         return -1;
57     }
58     return queue->arr[queue->front++];
59 }
60
61 // Peek at the front element of the queue
62 int peek(struct QueueArray *queue)
63 {
64     if (isEmpty(queue))
65     {
66         printf("Queue is empty!\n");
67         return -1;
68     }
69     return queue->arr[queue->front];
70 }
71
72 // Display the queue
73 void display(struct QueueArray *queue)
74 {
75     if (isEmpty(queue))
76     {
77         printf("Queue is empty!\n");
78         return;
79     }
80     printf("Queue elements: ");
81     for (int i = queue->front; i <= queue->rear; i++)
82     {
83         printf("%d ", queue->arr[i]);
84     }
85     printf("\n");
86 }
87
88 int main()
89 {
90     struct QueueArray *queue = createQueue();
91     int choice, value;
92
93     printf("\nQueue Operations (Array Implementation):\n");
94     printf("1. Enqueue\n");
95     printf("2. Dequeue\n");
96     printf("3. Peek\n");
97     printf("4. Display\n");
98     printf("5. Exit\n");
99
100     while (1)
101     {
102
103         printf("Enter your choice: ");
104         scanf("%d", &choice);
105
```

```
106     switch (choice)
107     {
108     case 1:
109         printf("Enter the value to enqueue: ");
110         scanf("%d", &value);
111         enqueue(queue, value);
112         printf("\n");
113         break;
114     case 2:
115         value = dequeue(queue);
116         if (value != -1)
117             printf("Dequeued value: %d\n", value);
118
119         printf("\n");
120         break;
121     case 3:
122         value = peek(queue);
123         if (value != -1)
124             printf("Front value: %d\n", value);
125
126         printf("\n");
127         break;
128     case 4:
129         display(queue);
130         printf("\n");
131         break;
132     case 5:
133         free(queue);
134         exit(0);
135     default:
136         printf("Invalid choice!\n");
137     }
138 }
139
140 return 0;
141 }
142
```

● @NyanAditya → /workspaces/Chamber_of_C (main) \$ cd

Queue Operations (Array Implementation):

1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit

Enter your choice: 4

Queue is empty!

Enter your choice: 1

Enter the value to enqueue: 82

82 enqueued to queue

Enter your choice: 1

Enter the value to enqueue: 96

96 enqueued to queue

Enter your choice: 1

Enter the value to enqueue: 23

23 enqueued to queue

Enter your choice: 1

Enter the value to enqueue: 5

5 enqueued to queue

Enter your choice: 3

Front value: 82

Enter your choice: 4

Queue elements: 82 96 23 5

Enter your choice: 2

Dequeued value: 82

Enter your choice: 2

Dequeued value: 96

Enter your choice: 4

Queue elements: 23 5

Enter your choice: 5

○ @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp6

SEM 3\Exp6\Queue_LL.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Node structure for the linked list
5  struct Node
6  {
7      int data;
8      struct Node *next;
9  };
10
11 // Queue structure using linked lists
12 struct QueueLinkedList
13 {
14     struct Node *front;
15     struct Node *rear;
16 };
17
18 // Function to create a queue
19 struct QueueLinkedList *createQueue()
20 {
21     struct QueueLinkedList *queue = (struct QueueLinkedList *)malloc(sizeof(struct
QueueLinkedList));
22     queue->front = queue->rear = NULL; // Initialize front and rear
23     return queue;
24 }
25
26 // Check if the queue is empty
27 int isEmpty(struct QueueLinkedList *queue)
28 {
29     return queue->front == NULL;
30 }
31
32 // Enqueue an element into the queue
33 void enqueue(struct QueueLinkedList *queue, int value)
34 {
35     struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
36     new_node->data = value;
37     new_node->next = NULL;
38
39     if (isEmpty(queue))
40     {
41         queue->front = queue->rear = new_node; // First node
42         printf("%d enqueued to queue\n", value);
43         return;
44     }
45
46     queue->rear->next = new_node; // Add new node at the end
47     queue->rear = new_node;      // Update the rear pointer
48     printf("%d enqueued to queue\n", value);
49 }
50
51 // Dequeue an element from the queue
```

```
52 int dequeue(struct QueueLinkedList *queue)
53 {
54     if (isEmpty(queue))
55     {
56         printf("Queue underflow!\n");
57         return -1;
58     }
59     struct Node *temp = queue->front;
60     int dequeued_value = temp->data;
61     queue->front = queue->front->next;
62
63     // If the front becomes NULL, set rear to NULL as well
64     if (queue->front == NULL)
65     {
66         queue->rear = NULL;
67     }
68
69     free(temp);
70     return dequeued_value;
71 }
72
73 // Peek at the front element of the queue
74 int peek(struct QueueLinkedList *queue)
75 {
76     if (isEmpty(queue))
77     {
78         printf("Queue is empty!\n");
79         return -1;
80     }
81     return queue->front->data;
82 }
83
84 // Display the queue
85 void display(struct QueueLinkedList *queue)
86 {
87     if (isEmpty(queue))
88     {
89         printf("Queue is empty!\n");
90         return;
91     }
92     struct Node *temp = queue->front;
93     printf("Queue elements: ");
94     while (temp != NULL)
95     {
96         printf("%d ", temp->data);
97         temp = temp->next;
98     }
99     printf("\n");
100 }
101
102 int main()
103 {
104     struct QueueLinkedList *queue = createQueue();
105     int choice, value;
```

```
106
107     printf("\nQueue Operations (Linked List Implementation):\n");
108     printf("1. Enqueue\n");
109     printf("2. Dequeue\n");
110     printf("3. Peek\n");
111     printf("4. Display\n");
112     printf("5. Exit\n");
113
114     while (1)
115     {
116
117         printf("Enter your choice: ");
118         scanf("%d", &choice);
119
120         switch (choice)
121         {
122             case 1:
123                 printf("Enter the value to enqueue: ");
124                 scanf("%d", &value);
125                 enqueue(queue, value);
126                 printf("\n");
127                 break;
128             case 2:
129                 value = dequeue(queue);
130                 if (value != -1)
131                     printf("Dequeued value: %d\n", value);
132
133                 printf("\n");
134                 break;
135             case 3:
136                 value = peek(queue);
137                 if (value != -1)
138                     printf("Front value: %d\n", value);
139
140                 printf("\n");
141                 break;
142             case 4:
143                 display(queue);
144                 printf("\n");
145                 break;
146             case 5:
147                 // Free linked list nodes (cleanup)
148                 while (!isEmpty(queue))
149                 {
150                     dequeue(queue);
151                 }
152                 free(queue);
153                 exit(0);
154             default:
155                 printf("Invalid choice!\n");
156         }
157     }
158
159     return 0;
```

```
160 | }  
161 |
```

● @NyanAditya → /workspaces/Chamber_of_C (main) \$ cd

Queue Operations (Linked List Implementation):

1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit

Enter your choice: 4

Queue is empty!

Enter your choice: 1

Enter the value to enqueue: 65

65 enqueued to queue

Enter your choice: 1

Enter the value to enqueue: 87

87 enqueued to queue

Enter your choice: 1

Enter the value to enqueue: 55

55 enqueued to queue

Enter your choice: 1

Enter the value to enqueue: 42

42 enqueued to queue

Enter your choice: 3

Front value: 65

Enter your choice: 4

Queue elements: 65 87 55 42

Enter your choice: 2

Dequeued value: 65

Enter your choice: 2

Dequeued value: 87

Enter your choice: 4

Queue elements: 55 42

Enter your choice: 5

○ @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp6

SEM 3\Exp7\Bin_Search.c

```
1  #include <stdio.h>
2
3  // Function to perform binary search
4  int binarySearch(int arr[], int size, int key)
5  {
6      int left = 0;
7      int right = size - 1;
8
9      while (left <= right)
10     {
11         int mid = left + (right - left) / 2;
12
13         if (arr[mid] == key)
14         {
15             return mid; // Return the index of the found element
16         }
17         if (arr[mid] < key)
18         {
19             left = mid + 1; // Search in the right half
20         }
21         else
22         {
23             right = mid - 1; // Search in the left half
24         }
25     }
26     return -1; // Return -1 if the element is not found
27 }
28
29 int main()
30 {
31     int arr[] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19}; // Note: Array must be sorted
32     for binary search
33     int size = sizeof(arr) / sizeof(arr[0]);
34     int key;
35
36     printf("Enter the element to search for (Binary Search): ");
37     scanf("%d", &key);
38
39     int index = binarySearch(arr, size, key);
40     if (index != -1)
41     {
42         printf("Element %d found at index %d.\n", key, index);
43     }
44     else
45     {
46         printf("Element %d not found in the array.\n", key);
47     }
48     return 0;
49 }
50
```

- @NyanAditya → /workspaces/Chamber_of_C (main) \$ cd '
Enter the element to search for (Binary Search): 11
Element 11 found at index 5.

SEM 3\Exp7\Linear_Search.c

```
1  #include <stdio.h>
2
3  // Function to perform linear search
4  int linearSearch(int arr[], int size, int key)
5  {
6      for (int i = 0; i < size; i++)
7      {
8          if (arr[i] == key)
9          {
10             return i; // Return the index of the found element
11          }
12      }
13      return -1; // Return -1 if the element is not found
14 }
15
16 int main()
17 {
18     int arr[] = {4, 84, 23, 45, 67, 12, 98, 34, 56, 78};
19     int size = sizeof(arr) / sizeof(arr[0]);
20     int key;
21
22     printf("Enter the element to search for (Linear Search): ");
23     scanf("%d", &key);
24
25     int index = linearSearch(arr, size, key);
26     if (index != -1)
27     {
28         printf("Element %d found at index %d.\n", key, index);
29     }
30     else
31     {
32         printf("Element %d not found in the array.\n", key);
33     }
34
35     return 0;
36 }
37
```


- @NyanAditya → /workspaces/Chamber_of_C (main) \$ cd "/workspa
Enter the element to search for (Linear Search): 12
Element 12 found at index 5.
- @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp7 (main) \$

SEM 3\Exp8\bubble_Sort.c

```
1  #include <stdio.h>
2
3  // Function to perform bubble sort
4  void bubbleSort(int arr[], int size)
5  {
6      for (int i = 0; i < size - 1; i++)
7      {
8          for (int j = 0; j < size - i - 1; j++)
9          {
10             if (arr[j] > arr[j + 1])
11             {
12                 // Swap arr[j] and arr[j + 1]
13                 int temp = arr[j];
14                 arr[j] = arr[j + 1];
15                 arr[j + 1] = temp;
16             }
17         }
18     }
19 }
20
```

SEM 3\Exp8\insertion_Sort.c

```
1 // Function to perform insertion sort
2 void insertionSort(int arr[], int size)
3 {
4     for (int i = 1; i < size; i++)
5     {
6         int key = arr[i];
7         int j = i - 1;
8
9         // Move elements greater than key to one position ahead
10        while (j >= 0 && arr[j] > key)
11        {
12            arr[j + 1] = arr[j];
13            j--;
14        }
15        arr[j + 1] = key;
16    }
17 }
18
```

SEM 3\Exp8\quick_sort.c

```
1 // Function to perform quick sort
2 int partition(int arr[], int low, int high)
3 {
4     int pivot = arr[high]; // Choosing the rightmost element as pivot
5     int i = (low - 1);      // Index of smaller element
6
7     for (int j = low; j < high; j++)
8     {
9         // If the current element is smaller than or equal to pivot
10        if (arr[j] <= pivot)
11        {
12            i++; // Increment index of smaller element
13            int temp = arr[i];
14            arr[i] = arr[j];
15            arr[j] = temp;
16        }
17    }
18    // Swap the pivot element with the element at i + 1
19    int temp = arr[i + 1];
20    arr[i + 1] = arr[high];
21    arr[high] = temp;
22    return i + 1; // Return the partitioning index
23 }
24
25 void quickSort(int arr[], int low, int high)
26 {
27     if (low < high)
28     {
29         int pi = partition(arr, low, high); // Partitioning index
30         quickSort(arr, low, pi - 1);        // Recursively sort elements before
partition
31         quickSort(arr, pi + 1, high);        // Recursively sort elements after
partition
32     }
33 }
34
```

SEM 3\Exp8\Selection_sort.c

```
1 // Function to perform selection sort
2 void selectionSort(int arr[], int size)
3 {
4     for (int i = 0; i < size - 1; i++)
5     {
6         int minIndex = i;
7         for (int j = i + 1; j < size; j++)
8         {
9             if (arr[j] < arr[minIndex])
10            {
11                minIndex = j; // Find the index of the minimum element
12            }
13        }
14        // Swap the found minimum element with the first element
15        int temp = arr[minIndex];
16        arr[minIndex] = arr[i];
17        arr[i] = temp;
18    }
19 }
20
```

SEM 3\Exp8\shell_Sort.c

```
1 // Function to perform shell sort
2 void shellSort(int arr[], int size)
3 {
4     for (int gap = size / 2; gap > 0; gap /= 2)
5     {
6         for (int i = gap; i < size; i++)
7         {
8             int temp = arr[i];
9             int j;
10
11             // Shift earlier gap-sorted elements up until the correct location for
arr[i] is found
12             for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
13             {
14                 arr[j] = arr[j - gap];
15             }
16             arr[j] = temp;
17         }
18     }
19 }
20
```

SEM 3\Exp8\main.c

```
1  #include <stdio.h>
2
3  // Function declarations for sorting algorithms
4  void bubbleSort(int arr[], int size);
5  void insertionSort(int arr[], int size);
6  void selectionSort(int arr[], int size);
7  void quickSort(int arr[], int low, int high);
8  void shellSort(int arr[], int size);
9
10 // Function to display the array
11 void display(int arr[], int size)
12 {
13     for (int i = 0; i < size; i++)
14         printf("%d ", arr[i]);
15     printf("\n");
16 }
17
18 int main()
19 {
20     int arr1[] = {64, 34, 25, 12, 22, 11, 90};
21     int size1 = sizeof(arr1) / sizeof(arr1[0]);
22
23     printf("Original array for Bubble Sort: ");
24     display(arr1, size1);
25     bubbleSort(arr1, size1);
26     printf("Sorted array using Bubble Sort: ");
27     display(arr1, size1);
28
29     // Reset the array for next sorting
30     int arr2[] = {64, 34, 25, 12, 22, 11, 90};
31     int size2 = sizeof(arr2) / sizeof(arr2[0]);
32
33     printf("\nOriginal array for Insertion Sort: ");
34     display(arr2, size2);
35     insertionSort(arr2, size2);
36     printf("Sorted array using Insertion Sort: ");
37     display(arr2, size2);
38
39     // Reset the array for next sorting
40     int arr3[] = {64, 34, 25, 12, 22, 11, 90};
41     int size3 = sizeof(arr3) / sizeof(arr3[0]);
42
43     printf("\nOriginal array for Selection Sort: ");
44     display(arr3, size3);
45     selectionSort(arr3, size3);
46     printf("Sorted array using Selection Sort: ");
47     display(arr3, size3);
48
49     // Reset the array for next sorting
50     int arr4[] = {64, 34, 25, 12, 22, 11, 90};
51     int size4 = sizeof(arr4) / sizeof(arr4[0]);
```

```
52
53     printf("\nOriginal array for Quick Sort: ");
54     display(arr4, size4);
55     quickSort(arr4, 0, size4 - 1);
56     printf("Sorted array using Quick Sort: ");
57     display(arr4, size4);
58
59     // Reset the array for next sorting
60     int arr5[] = {64, 34, 25, 12, 22, 11, 90};
61     int size5 = sizeof(arr5) / sizeof(arr5[0]);
62
63     printf("\nOriginal array for Shell Sort: ");
64     display(arr5, size5);
65     shellSort(arr5, size5);
66     printf("Sorted array using Shell Sort: ");
67     display(arr5, size5);
68
69     return 0;
70 }
71
```


SEM 3\Exp8\Makefile

```
1  # Compiler to use
2  CC = gcc
3
4  # Compiler flags
5  CFLAGS = -Wall -Wextra -g
6
7  # Object files to compile
8  OBJS = main.o bubble_Sort.o insertion_Sort.o Selection_sort.o quick_sort.o
   shell_Sort.o
9
10 # The final executable name
11 TARGET = Exp8_sorting_program
12
13 # Default target to build the executable
14 all: $(TARGET)
15
16 # Rule to link object files into the final executable
17 $(TARGET): $(OBJS)
18     $(CC) -o $(TARGET) $(OBJS)
19
20 # Rule to compile each .c file into a .o file
21 %.o: %.c
22     $(CC) $(CFLAGS) -c $<
23
24 # Clean target to remove object files and the executable
25 clean:
26     rm -f $(OBJS) $(TARGET)
27
```

```
● @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp8 (main) $ make
gcc -Wall -Wextra -g -c main.c
gcc -Wall -Wextra -g -c bubble_Sort.c
gcc -Wall -Wextra -g -c insertion_Sort.c
gcc -Wall -Wextra -g -c Selection_sort.c
gcc -Wall -Wextra -g -c quick_sort.c
gcc -Wall -Wextra -g -c shell_Sort.c
gcc -o Exp8_sorting_program main.o bubble_Sort.o insertion_Sort.o Selection_sort.o quick_sort.o shell_Sort.o
● @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp8 (main) $ ./Exp8_sorting_program
Original array for Bubble Sort: 64 34 25 12 22 11 90
Sorted array using Bubble Sort: 11 12 22 25 34 64 90

Original array for Insertion Sort: 64 34 25 12 22 11 90
Sorted array using Insertion Sort: 11 12 22 25 34 64 90

Original array for Selection Sort: 64 34 25 12 22 11 90
Sorted array using Selection Sort: 11 12 22 25 34 64 90

Original array for Quick Sort: 64 34 25 12 22 11 90
Sorted array using Quick Sort: 11 12 22 25 34 64 90

Original array for Shell Sort: 64 34 25 12 22 11 90
Sorted array using Shell Sort: 11 12 22 25 34 64 90
○ @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp8 (main) $ |
```

SEM 3\Exp9\Bucket_Sort.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define BUCKET_SIZE 10
5
6  typedef struct Bucket
7  {
8      int count;
9      int *values;
10 } Bucket;
11
12 void bucketSort(int array[], int n);
13 void insertionSort(int array[], int n);
14
15 void bucketSort(int array[], int n)
16 {
17     Bucket buckets[BUCKET_SIZE];
18     for (int i = 0; i < BUCKET_SIZE; i++)
19     {
20         buckets[i].count = 0;
21         buckets[i].values = (int *)malloc(n * sizeof(int));
22     }
23
24     for (int i = 0; i < n; i++)
25     {
26         int bucketIndex = array[i] / BUCKET_SIZE;
27         buckets[bucketIndex].values[buckets[bucketIndex].count++] = array[i];
28     }
29
30     for (int i = 0; i < BUCKET_SIZE; i++)
31     {
32         if (buckets[i].count > 0)
33         {
34             insertionSort(buckets[i].values, buckets[i].count);
35         }
36     }
37
38     int index = 0;
39     for (int i = 0; i < BUCKET_SIZE; i++)
40     {
41         for (int j = 0; j < buckets[i].count; j++)
42         {
43             array[index++] = buckets[i].values[j];
44         }
45         free(buckets[i].values);
46     }
47 }
48
49 void insertionSort(int array[], int n)
50 {
51     for (int i = 1; i < n; i++)
```

```
52     {  
53         int key = array[i];  
54         int j = i - 1;  
55         while (j >= 0 && array[j] > key)  
56         {  
57             array[j + 1] = array[j];  
58             j--;  
59         }  
60         array[j + 1] = key;  
61     }  
62 }  
63
```

SEM 3\Exp9\counting_sort.c

```
1 // Function to perform counting sort
2 void countingSort(int arr[], int size)
3 {
4     int output[size];
5     int count[100] = {0}; // Assuming the range of input numbers is known (0-99)
6
7     // Store the count of occurrences
8     for (int i = 0; i < size; i++)
9         count[arr[i]]++;
10
11     // Build the output array
12     for (int i = 0, j = 0; i < 100; i++)
13     {
14         while (count[i] > 0)
15         {
16             output[j++] = i;
17             count[i]--;
18         }
19     }
20
21     // Copy the output array to arr[]
22     for (int i = 0; i < size; i++)
23         arr[i] = output[i];
24 }
25
```

SEM 3\Exp9\Heap_Sort.c

```
1 // Function to heapify a subtree rooted at index i
2 void heapify(int arr[], int size, int i)
3 {
4     int largest = i;    // Initialize largest as root
5     int left = 2 * i + 1; // left = 2*i + 1
6     int right = 2 * i + 2; // right = 2*i + 2
7
8     // If left child is larger than root
9     if (left < size && arr[left] > arr[largest])
10         largest = left;
11
12     // If right child is larger than largest so far
13     if (right < size && arr[right] > arr[largest])
14         largest = right;
15
16     // If largest is not root
17     if (largest != i)
18     {
19         int temp = arr[i];
20         arr[i] = arr[largest];
21         arr[largest] = temp;
22
23         // Recursively heapify the affected subtree
24         heapify(arr, size, largest);
25     }
26 }
27
28 // Function to perform heap sort
29 void heapSort(int arr[], int size)
30 {
31     // Build heap (rearrange array)
32     for (int i = size / 2 - 1; i >= 0; i--)
33         heapify(arr, size, i);
34
35     // One by one extract an element from heap
36     for (int i = size - 1; i > 0; i--)
37     {
38         // Move current root to end
39         int temp = arr[0];
40         arr[0] = arr[i];
41         arr[i] = temp;
42
43         // Call heapify on the reduced heap
44         heapify(arr, i, 0);
45     }
46 }
47
```

SEM 3\Exp9\Merge_Sort.c

```
1  #include <stdio.h>
2
3  // Function to merge two subarrays
4  void merge(int arr[], int left, int mid, int right)
5  {
6      int i, j, k;
7      int n1 = mid - left + 1;
8      int n2 = right - mid;
9
10     // Create temporary arrays
11     int L[n1], R[n2];
12
13     // Copy data to temporary arrays
14     for (i = 0; i < n1; i++)
15         L[i] = arr[left + i];
16     for (j = 0; j < n2; j++)
17         R[j] = arr[mid + 1 + j];
18
19     // Merge the temporary arrays
20     i = 0;    // Initial index of first subarray
21     j = 0;    // Initial index of second subarray
22     k = left; // Initial index of merged subarray
23     while (i < n1 && j < n2)
24     {
25         if (L[i] <= R[j])
26         {
27             arr[k] = L[i];
28             i++;
29         }
30         else
31         {
32             arr[k] = R[j];
33             j++;
34         }
35         k++;
36     }
37
38     // Copy remaining elements of L[], if any
39     while (i < n1)
40     {
41         arr[k] = L[i];
42         i++;
43         k++;
44     }
45
46     // Copy remaining elements of R[], if any
47     while (j < n2)
48     {
49         arr[k] = R[j];
50         j++;
51         k++;
```

```
52     }
53 }
54
55 // Function to perform merge sort
56 void mergeSort(int arr[], int left, int right)
57 {
58     if (left < right)
59     {
60         int mid = left + (right - left) / 2;
61
62         // Sort first and second halves
63         mergeSort(arr, left, mid);
64         mergeSort(arr, mid + 1, right);
65         merge(arr, left, mid, right);
66     }
67 }
68
```


SEM 3\Exp9\Radix_sort.c

```
1 // Function to get the maximum value in an array
2 int getMax(int arr[], int size)
3 {
4     int max = arr[0];
5     for (int i = 1; i < size; i++)
6         if (arr[i] > max)
7             max = arr[i];
8     return max;
9 }
10
11 // Function to perform counting sort based on a specific digit
12 extern void countingSort(int arr[], int size, int exp);
13
14 void radixSort(int arr[], int size)
15 {
16     // Get the maximum number to know the number of digits
17     int max = getMax(arr, size);
18
19     // Apply counting sort to sort elements based on each digit
20     for (int exp = 1; max / exp > 0; exp *= 10)
21         countingSort(arr, size, exp);
22 }
23
```

SEM 3\Exp9\main.c

```
1  #include <stdio.h>
2
3  // Function declarations (you can also include headers for better organization)
4  void mergeSort(int arr[], int left, int right);
5  void radixSort(int arr[], int size);
6  void countingSort(int arr[], int size);
7  void bucketSort(int arr[], int size);
8  void heapSort(int arr[], int size);
9
10 // Function to display the array (add this in main.c)
11 void display(int arr[], int size)
12 {
13     for (int i = 0; i < size; i++)
14         printf("%d ", arr[i]);
15     printf("\n");
16 }
17
18 int main()
19 {
20     // Array for testing sorting algorithms
21     int arr1[] = {74, 34, 25, 12, 22, 11, 90, 65, 32, 1};
22     int size1 = sizeof(arr1) / sizeof(arr1[0]);
23
24     // Merge Sort
25     printf("Original array for Merge Sort: ");
26     display(arr1, size1);
27     mergeSort(arr1, 0, size1 - 1);
28     printf("Sorted array using Merge Sort: ");
29     display(arr1, size1);
30
31     // Radix Sort
32     int arr2[] = {74, 34, 25, 12, 22, 11, 90, 65, 32, 1};
33     printf("\nOriginal array for Radix Sort: ");
34     display(arr2, size1);
35     radixSort(arr2, size1);
36     printf("Sorted array using Radix Sort: ");
37     display(arr2, size1);
38
39     // Counting Sort
40     int arr3[] = {74, 34, 25, 12, 22, 11, 90, 65, 32, 1};
41     printf("\nOriginal array for Counting Sort: ");
42     display(arr3, size1);
43     countingSort(arr3, size1);
44     printf("Sorted array using Counting Sort: ");
45     display(arr3, size1);
46
47     // Bucket Sort
48     int arr4[] = {74, 34, 25, 12, 22, 11, 90, 65, 32, 1};
49     int size4 = sizeof(arr4) / sizeof(arr4[0]);
50     printf("\nOriginal array for Bucket Sort: ");
51     display(arr4, size4);
```

```
52     bucketSort(arr4, size4);
53     printf("Sorted array using Bucket Sort: ");
54     display(arr4, size4);
55
56     // Heap Sort
57     int arr5[] = {74, 34, 25, 12, 22, 11, 90, 65, 32, 1};
58     int size5 = sizeof(arr5) / sizeof(arr5[0]);
59     printf("\nOriginal array for Heap Sort: ");
60     display(arr5, size5);
61     heapSort(arr5, size5);
62     printf("Sorted array using Heap Sort: ");
63     display(arr5, size5);
64
65     return 0;
66 }
67
```

SEM 3\Exp9\Makefile

```
1  # Compiler to use
2  CC = gcc
3
4  # Compiler flags
5  CFLAGS = -Wall -Wextra -g
6
7  # Object files to compile
8  OBJS = main.o Bucket_Sort.o counting_sort.o Heap_Sort.o Merge_Sort.o Radix_sort.o
9
10 # The final executable name
11 TARGET = Exp9_sorting_program
12
13 # Default target to build the executable
14 all: $(TARGET)
15
16 # Rule to link object files into the final executable
17 $(TARGET): $(OBJS)
18     $(CC) -o $(TARGET) $(OBJS)
19
20 # Rule to compile each .c file into a .o file
21 %.o: %.c
22     $(CC) $(CFLAGS) -c $<
23
24 # Clean target to remove object files and the executable
25 clean:
26     rm -f $(OBJS) $(TARGET)
27
```

- @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp9 (main) \$ make
gcc -Wall -Wextra -g -c main.c
gcc -Wall -Wextra -g -c Bucket_Sort.c
gcc -Wall -Wextra -g -c counting_sort.c
gcc -Wall -Wextra -g -c Heap_Sort.c
gcc -Wall -Wextra -g -c Merge_Sort.c
gcc -Wall -Wextra -g -c Radix_sort.c
gcc -o Exp9_sorting_program main.o Bucket_Sort.o counting_sort.o Heap_Sort.o Merge_Sort.o Radix_sort.o
- @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp9 (main) \$./Exp9_sorting_program
Original array for Merge Sort: 74 34 25 12 22 11 90 65 32 1
Sorted array using Merge Sort: 1 11 12 22 25 32 34 65 74 90

Original array for Radix Sort: 74 34 25 12 22 11 90 65 32 1
Sorted array using Radix Sort: 1 11 12 22 25 32 34 65 74 90

Original array for Counting Sort: 74 34 25 12 22 11 90 65 32 1
Sorted array using Counting Sort: 1 11 12 22 25 32 34 65 74 90

Original array for Bucket Sort: 74 34 25 12 22 11 90 65 32 1
Sorted array using Bucket Sort: 1 11 12 22 25 32 34 65 74 90

Original array for Heap Sort: 74 34 25 12 22 11 90 65 32 1
Sorted array using Heap Sort: 1 11 12 22 25 32 34 65 74 90
- @NyanAditya → /workspaces/Chamber_of_C/SEM 3/Exp9 (main) \$