

01 - Arrays and Abstract Data Type\V06-aadt.c

```
20 //Implementing Array as an Abstract Data Type
21 #include<stdio.h>
22 #include<stdlib.h>
23 // Structure to represent a dynamic array
24 struct myArray {
25     int total_size; // Total size of the array
26     int used_size; // Number of elements used in the array
27     int *ptr;       // Pointer to the dynamically allocated array
28 };
29 // Function to create an array with specified total size and used size
30 void createArray(struct myArray *a, int tsize, int usize){
31     (*a).total_size = tsize;
32     (*a).used_size = usize;
33     (*a).ptr = (int*)malloc(tsize * sizeof(int)); // Allocate memory for the array
34 /*
35 //same thing diff syntax.....
36 a->total_size = tsize;
37 a->used_size = usize;
38 a->ptr = (int*)malloc(tsize * sizeof(int));
39 */
40 }
41 // Function to display the elements of the array
42 void show(struct myArray *a){
43     for (int i = 0; i < (*a).used_size; i++) {
44         printf("%d,", (*a).ptr[i]);
45     }
46 }
47 // Function to set values for each element of the array
48 void setVal(struct myArray *a){
49     int n;
50     for (int i = 0; i < (*a).used_size; i++) {
51         printf("Enter element %d: ", i + 1);
52         scanf("%d", &n);
53         (*a).ptr[i] = n;
54     }
55 }
56 int main() {
57     struct myArray marks;
58     createArray(&marks, 10, 5);
59     printf("we are runnin setval now\n");
60     setVal(&marks);
61     printf("we re running show now\n");
62     show(&marks);
63     free(marks.ptr); // Free the dynamically allocated memory before exiting
64     return 0;
65 }
```

02 - Operations on Arrays in Data Structures\V10-ins.c

```
40 //Insertion Operation in Array
41 #include <stdio.h>
42 #include <stdlib.h>
43 // Function to display the elements of an array
44 void display(int arr[], int size) {
45     for (int i = 0; i < size; i++) {
46         printf("%d,", arr[i]);
47     }
48     printf("\n");
49 }
50 // Function to insert an element at a specified index in the array
51 int index_insertion(int arr[], int size, int element_to_ins, int at_index, int capacity) {
52     // Check if the array is already at full capacity
53     if (size >= capacity) {
54         return -1;
55     }
56     // Shift elements to the right to create space for the new element
57     for (int i = size - 1; i >= at_index; i--) {
58         arr[i + 1] = arr[i];
59     }
60     // Insert the new element at the specified index
61     arr[at_index] = element_to_ins;
62     return 1;
63 }
64 int main() {
65     // Initialize an array with some values
66     int arr[100] = {1, 2, 3, 4, 5, 6, 8, 9, 10};
67     // Initial size and capacity of the array
68     int size = 9, capacity = 100;
69     // Element to insert and the index at which to insert
70     int element_to_ins = 7, at_index = 6;
71     printf("Array before insertion: ");
72     display(arr, size);
73     index_insertion(arr, size, element_to_ins, at_index, capacity);
74     size++;
75     printf("Array after insertion: ");
76     display(arr, size);
77     return 0;
78 }
```

02 - Operations on Arrays in Data Structures\V11-del.c

```
3 //-----
-----  
4 //Deletion Operation in Array  
5  
6 #include <stdio.h>  
7 #include <stdlib.h>  
8  
9 // Function to display the elements of an array  
10 void display(int arr[], int size) {  
11     for (int i = 0; i < size; i++) {  
12         printf("%d,", arr[i]);  
13     }  
14     printf("\n");  
15 }  
16  
17 int index_deletion(int arr[], int size, int index_to_del) {  
18     for (int i = index_to_del; i < size -1 ; i++)  
19     {  
20         arr[i] = arr[i+1];  
21     }  
22  
23     return 1;  
24 }  
25  
26  
27 int main() {  
28     // Initialize an array with some values  
29     int arr[100] = {1, 2, 3, 4, 5};  
30     // Initial size and capacity of the array  
31     int size = 5, capacity = 100;  
32     // Element to insert and the index at which to insert  
33     int index_to_del = 2;  
34  
35     printf("Array before deletion: ");  
36     display(arr, size);  
37  
38     index_deletion(arr, size, index_to_del);  
39     size--;  
40  
41     printf("Array after deletion: ");  
42     display(arr, size);  
43  
44     return 0;  
45 }  
46  
47  
48 //-----
```

03 - LinearSearch & BinarySearch\V12-BS.c

```
1 //-----  
2 //Binary search  
3  
4 #include<stdio.h>  
5  
6 int binarySearch(int arr[],int low,int high, int key){  
7     if(low>=high){  
8         int mid = low + (high-low)/2;  
9         if (arr[mid]== key)  
10            {  
11                return mid;  
12            }  
13            if (arr[mid]>key)  
14            {  
15                return binarySearch(arr,low,mid -1,key);  
16            }  
17            if (arr[mid]<key)  
18            {  
19                return binarySearch(arr,mid+1,high,key);  
20            }  
21        }  
22        return -1;  
23    }  
24  
25 int main() {  
26     int arr[] = {2,4,6,8,12,34,56,78,90,99};  
27     int size = sizeof(arr)/sizeof(arr[0]);  
28     // printf("%d",size);  
29     int key = 8;  
30     int indx = binarySearch(arr,0,size-1,key);  
31     if (indx== -1)  
32     {  
33         printf("not found");  
34     }  
35     else{  
36         printf("indx = %d",indx);  
37     }  
38     return 0;  
39 }
```

03 - LinearSearch & BinarySearch\V12-LS.c

```
15 //-----
16 //Linear search
17
18 #include<stdio.h>
19 #include<stdlib.h>
20
21 int LinearSearch(int arr[],int key, int size){
22
23     for (int i = 0; i < size; i++)
24     {
25         if (arr[i]==key)
26         {
27             return i;
28         }
29     }
30     return -1;
31 }
32
33 int main() {
34     int arr[] = {5,6,7,8,23,52,536,35635,6356,546,4565,4654,6,45654,654,654,645,6546,45,645,
35,23,343,6,99,45,87,674,63442,11,25,345,463,436};
35     int size = sizeof(arr)/sizeof(arr[0]);
36     int key = 99;
37     int indx = LinearSearch(arr,key,size);
38     printf("%d is present in the index %d",key,indx);
39     return 0;
40 }
```

04 - Linked List\V14-LL.c

```
7 // -----
8 // linked list
9
10 #include <stdio.h>
11 #include <stdlib.h>
12
13 struct node
14 {
15     int data;
16     struct node *next;
17 };
18
19 void llTraversal(struct node *temp)
20 {
21     while (temp != NULL)
22     {
23         printf("elems: %d\n", temp->data);
24         temp = temp->next;
25     }
26 }
27
28 int main()
29 {
30
31     struct node *head;
32     struct node *second;
33     struct node *third;
34
35     // allocate memory for nodes in heap
36     head = (struct node *)malloc(sizeof(struct node));
37     second = (struct node *)malloc(sizeof(struct node));
38     third = (struct node *)malloc(sizeof(struct node));
39
40     head->data = 10;
41     head->next = second;
42
43     second->data = 100;
44     second->next = third;
45
46     third->data = 1000;
47     third->next = NULL;
48
49     llTraversal(head);
50
51     return 0;
52 }
```

04 - Linked List\V16-I.c

```

108 // -----
109 // insertion in linked list
110 /*
111 case 1 - insert at beginning O(1)
112 case 2 - insert at between O(n)
113 case 3 - insert at the end O(n)
114 case 4 - insert after a given node O(1)
115 */
116 #include <stdio.h>
117 #include <stdlib.h>
118 // creating structure node
119 struct node
120 {
121     int data;
122     struct node *next;
123 };
124 // calling function
125 void linkedListTraversal(struct node *ptr);
126 struct node *insertAtFirst(struct node *head, int data);
127 struct node *insertAtIndex(struct node *head, int data, int index);
128 struct node *insertAtEnd(struct node *head, int data);
129 struct node *insertAfterNode(struct node *head, struct node *prevNode, int data);
130 int main()
131 {
132     /* //one way
133         struct node *head;
134         struct node *second;
135         struct node *third;
136         struct node *fourth;
137
138         //allocate memory for nodes in the linked list in heap
139         head = (struct node *)malloc(sizeof(struct node));
140         second = (struct node *)malloc(sizeof(struct node));
141         third = (struct node *)malloc(sizeof(struct node));
142         fourth = (struct node *)malloc(sizeof(struct node));
143     */
144     // declare and allocate memory for nodes in the linked list in heap
145     struct node *head = (struct node *)malloc(sizeof(struct node));
146     struct node *second = (struct node *)malloc(sizeof(struct node));
147     struct node *third = (struct node *)malloc(sizeof(struct node));
148     struct node *fourth = (struct node *)malloc(sizeof(struct node));
149     // creating linked list chain
150     // link first and second nodes
151     head->data = 10;
152     head->next = second;
153     // link second and third nodes
154     second->data = 20;
155     second->next = third;
156     // link third and fourth nodes
157     third->data = 30;
158     third->next = fourth;
159     // terminating the list with null in 4th
160     fourth->data = 40;
161     fourth->next = NULL;
162     printf("\nlist before insertion-----\n");
163     linkedListTraversal(head);
164     // head = insertAtFirst(head, 99);

```

```
165     // head = insertAtIndex(head, 99, 2);
166     // head = insertAtEnd(head, 99);
167     // head = insertAfterNode(head, third,99);
168     printf("\nlist after insertion-----\n");
169     linkedListTraversal(head);
170     return 0;
171 }
172 void linkedListTraversal(struct node *ptr)
173 {
174     while (ptr != NULL)
175     {
176         printf("elements: %d\n", ptr->data);
177         ptr = ptr->next;
178     }
179 }
180 struct node *insertAtFirst(struct node *head, int data)
181 {
182     struct node *ptr = (struct node *)malloc(sizeof(struct node));
183     ptr->data = data;
184     ptr->next = head;
185     return ptr;
186 }
187 struct node *insertAtIndex(struct node *head, int data, int index)
188 {
189     struct node *ptr = (struct node *)malloc(sizeof(struct node));
190     struct node *p = head;
191     int i = 0;
192     while (i != index - 1)
193     {
194         p = p->next;
195         i++;
196     }
197     ptr->data = data;
198     ptr->next = p->next;
199     p->next = ptr;
200     return head;
201 }
202 struct node *insertAtEnd(struct node *head, int data)
203 {
204     struct node *ptr = (struct node *)malloc(sizeof(struct node));
205     ptr->data = data;
206     struct node *p = head;
207     while (p->next != NULL)
208     {
209         p = p->next;
210     }
211     p->next = ptr;
212     ptr->next = NULL;
213     return head;
214 }
215 struct node * insertAfterNode(struct node* head,struct node * prevNode,int data){
216     struct node * ptr = (struct node *)malloc(sizeof(struct node));
217     ptr->data = data;
218     ptr->next = prevNode->next;
219     prevNode->next = ptr;
220     return head;
221 }
```

04 - Linked List\V17-D.c

```
5 //-----
6 //deletion of linked list
7 #include <stdio.h>
8 #include <stdlib.h>
9 struct Node{
10     int data;
11     struct Node *next;
12 };
13 void linkedListTraversal(struct Node *ptr){
14     while (ptr != NULL){
15         printf("Element: %d\n", ptr->data);
16         ptr = ptr->next;
17     }
18 }
19 // Case 1: Deleting the first element from the linked list
20 struct Node * deleteFirst(struct Node * head){
21     struct Node * ptr = head;
22     head = head->next;
23     free(ptr);
24     return head;
25 }
26 // Case 2: Deleting the element at a given index from the linked list
27 struct Node * deleteAtIndex(struct Node * head, int index){
28     struct Node *p = head;
29     struct Node *q = head->next;
30     for (int i = 0; i < index-1; i++){
31         p = p->next;
32         q = q->next;
33     }
34
35     p->next = q->next;
36     free(q);
37     return head;
38 }
39 // Case 3: Deleting the last element
40 struct Node * deleteAtLast(struct Node * head){
41     struct Node *p = head;
42     struct Node *q = head->next;
43     while(q->next !=NULL) {
44         p = p->next;
45         q = q->next;
46     }
47     p->next = NULL;
48     free(q);
49     return head;
50 }
51 // Case 4: Deleting the element with a given value from the linked list
52 struct Node * deleteAtIndex(struct Node * head, int value){
53     struct Node *p = head;
54     struct Node *q = head->next;
55     while(q->data!=value && q->next!= NULL){
56         p = p->next;
57         q = q->next;
58     }
59     if(q->data == value){
60         p->next = q->next;
61         free(q);
```

```
62     }
63     return head;
64 }
65 int main(){
66     struct Node *head;
67     struct Node *second;
68     struct Node *third;
69     struct Node *fourth;
70     // Allocate memory for nodes in the linked list in Heap
71     head = (struct Node *)malloc(sizeof(struct Node));
72     second = (struct Node *)malloc(sizeof(struct Node));
73     third = (struct Node *)malloc(sizeof(struct Node));
74     fourth = (struct Node *)malloc(sizeof(struct Node));
75     // Link first and second nodes
76     head->data = 4;
77     head->next = second;
78     // Link second and third nodes
79     second->data = 3;
80     second->next = third;
81     // Link third and fourth nodes
82     third->data = 8;
83     third->next = fourth;
84     // Terminate the list at the third node
85     fourth->data = 1;
86     fourth->next = NULL;
87     printf("Linked list before deletion\n");
88     linkedListTraversal(head);
89     // head = deleteFirst(head); // For deleting first element of the linked list
90     // head = deleteAtIndex(head, 2);
91     head = deleteAtLast(head);
92     printf("Linked list after deletion\n");
93     linkedListTraversal(head);
94     return 0;
95 }
```

04 - Linked List\V20-CLL.c

```
9 // -----
10 // circular linked list
11 #include <stdio.h>
12 #include <stdlib.h>
13 struct node{
14     int data;
15     struct node *next;
16 };
17 void linkedListTraversal(struct node *head){
18     struct node *ptr = head;
19     do{
20         printf("element: %d\n", ptr->data);
21         ptr = ptr->next;
22     } while (ptr != head);
23 }
24 struct node * insertAtFirst(struct node * head, int data){
25     struct node * ptr = (struct node *)malloc(sizeof(struct node));
26     ptr->data = data;
27     struct node * p = head->next;
28     while(p->next != head){
29         p = p->next;
30     } //no its point to a before head
31     p->next = ptr;
32     ptr->next = head;
33     head = ptr;
34     return head;
35 }
36 int main(){
37     struct node *head = (struct node *)malloc(sizeof(struct node));
38     struct node *second = (struct node *)malloc(sizeof(struct node));
39     struct node *third = (struct node *)malloc(sizeof(struct node));
40     struct node *fourth = (struct node *)malloc(sizeof(struct node));
41     head->data = 4;
42     head->next = second;
43     second->data = 3;
44     second->next = third;
45     third->data = 6;
46     third->next = fourth;
47     fourth->data = 1;
48     fourth->next = head;
49     printf("circular linked list traversal before insertion\n");
50     linkedListTraversal(head);
51     head = insertAtFirst(head,80);
52     printf("circular linked list traversal after insertion\n");
53     linkedListTraversal(head);
54     return 0;
55 }
```

04 - Linked List\V21-DLL.c

```
39 // -----
40 // doubly linked list
41 #include <stdio.h>
42 #include <stdlib.h>
43 struct node{
44     struct node *prev;
45     int data;
46     struct node *next;
47 };
48 void linkedListTraversal(struct node *head){
49     struct node *ptr = head;
50     while (ptr != NULL){
51         printf("elemnts: %d\n", ptr->data);
52         ptr = ptr->next;
53     }
54     printf("----\n");
55     do{ //error here
56         printf("element: %d\n", ptr->data);
57         ptr = ptr->prev;
58     } while (ptr != NULL);
59 }
60 int main(){
61     struct node *head = (struct node *)malloc(sizeof(struct node));
62     struct node *n2 = (struct node *)malloc(sizeof(struct node));
63     struct node *n3 = (struct node *)malloc(sizeof(struct node));
64     struct node *n4 = (struct node *)malloc(sizeof(struct node));
65     head->next = n2;
66     head->prev = NULL;
67     head->data = 10;
68     n2->next = n3;
69     n2->prev = head;
70     n2->data = 20;
71     n3->next = n4;
72     n3->prev = n2;
73     n3->data = 30;
74     n4->next = NULL;
75     n4->prev = n3;
76     n4->data = 40;
77     linkedListTraversal(head);
78     return 0;
79 }
```

05- Stack\V24-stack.c

```
19 // -----
20 //stack
21 #include<stdio.h>
22 #include<stdlib.h>
23 struct stack{
24     int size;
25     int top;
26     int *arr;
27 };
28 int isEmpty(struct stack * ptr){
29     if(ptr->top == -1){
30         return 1;
31     }
32     else{
33         return 0;
34     }
35 }
36 int isFull(struct stack * ptr){
37     if (ptr->top == ptr->size-1){
38         return 1;
39     }
40     else{
41         return 0;
42     }
43 }
44 void push(struct stack * ptr,int val){
45     if(isFull(ptr)){
46         printf("overflow");
47     }
48     else{
49         ptr->top++;
50         ptr->arr[ptr->top] = val;
51     }
52 }
53 int pop(struct stack * ptr){
54     if (isEmpty(ptr)){
55         printf("underflow");
56         return -1;
57     }
58     else{
59         int val = ptr->arr[ptr->top];
60         ptr->top--;
61         return val;
62     }
63 }
64 int peek(struct stack * ptr, int i){
65     if (ptr->top-i+1<0){
66         printf("invalid positon");
67         return -1;
68     }
69     else{
70         return ptr->arr[ptr->top-i+1];
71     }
72 }
73 void display(struct stack * ptr){
74     if (isEmpty(ptr)){
75         printf("empty");
```

```
76     }
77     else{
78         for (int i = ptr->size-1; i >= 0; i--){
79             printf("%d,",ptr->arr[i]);
80         }
81     }
82 }
83 int stackTop(struct stack * ptr){
84     return ptr->arr[ptr->top];
85 }
86 int stackBottom(struct stack * ptr){
87     return ptr->arr[0];
88 }
89 int main() {
90     struct stack * S;
91     S->size = 5;
92     S->top = -1;
93     S->arr = (int*)malloc(S->size * sizeof(int));
94     printf("%d\n", isEmpty(S));
95     printf("%d\n", isFull(S));
96     push(S,50);
97     push(S,50);
98     push(S,50);
99     push(S,50);
100    push(S,50);
101    printf("%d\n", isEmpty(S));
102    printf("%d\n", isFull(S));
103    printf("%d\n", peek(S,2));
104    printf("%d",S->arr[S->top]);
105    printf("\n----\n");
106    display(S);
107    return 0;
108 }
```

05- Stack\V29-stack_LinkedList.c

```
15 //-----
16 //stack using linked list
17 #include<stdio.h>
18 #include<stdlib.h>
19 struct stack{
20     int size;
21     int top;
22     int * arr[];
23 };
24 struct node{
25     int data;
26     struct node * next;
27 };
28 void linkedListTraversal(struct node * top){
29     while (top != NULL){
30         printf("element: %d\n",top->data);
31         top = top->next;
32     }
33 }
34 int isEmpty(struct node * top){
35     if (top==NULL){
36         return 1;
37     }
38     else{
39         return 0;
40     }
41 }
42 int isFull(struct node * top){
43     struct node * ptr = (struct node *)malloc(sizeof(struct node));
44     if (ptr == NULL){
45         return 1;
46     }
47     else{
48         return 0;
49     }
50 }
51 struct node * push(struct node * top, int x){
52     if (isFull(top)){
53         printf("overflow");
54     }
55     else{
56         struct node * n = (struct node *)malloc(sizeof(struct node));
57         n->data = x;
58         n->next = top;
59         top = n;
60         return top;
61     }
62 }
63 }
64 int pop(struct node ** top){
65     if (isEmpty(*top)){
66         printf("underflow");
67     }
68     else{
69         struct node * n = *top;
70         *top = (*top)->next;
71         int x = n->data;
```

```
72     free(n);
73     return x;
74 }
75 }
76 int peek(struct node *top, int pos) {
77     struct node *ptr = top;
78     for (int i = 0; (i < pos - 1 && ptr != NULL); i++) {
79         ptr = ptr->next;
80     }
81     if (ptr != NULL) {
82         return ptr->data;
83     } else {
84         return -1;
85     }
86 }
87 int main() {
88     struct node * top = NULL;
89     top = push(top,10);
90     top = push(top,11);
91     top = push(top,12);
92     top = push(top,13 );
93     printf("\nbefore:-----\n");
94     linkedListTraversal(top);
95     int el_p = pop(&top);
96     printf("\nafter:-----\n");
97     linkedListTraversal(top);
98     printf("poped elem is%d",el_p);
99     return 0;
100 }
```

05- Stack\V33-paranthesisMatching.c

```
76 //-----  
77 // application of stack -> paranthesis matching in an expression  
78 #include <stdio.h>  
79 #include <stdlib.h>  
80 struct stack  
81 {  
82     int size;  
83     int top;  
84     char *arr;  
85 };  
86 int isEmpty(struct stack *ptr){  
87     if (ptr->top == -1){  
88         return 1;  
89     }  
90     else{  
91         return 0;  
92     }  
93 }  
94 int isFull(struct stack *ptr){  
95     if (ptr->top == ptr->size - 1){  
96         return 1;  
97     }  
98     else{  
99         return 0;  
100    }  
101 }  
102 void push(struct stack *ptr, char val){  
103     if (isFull(ptr)){  
104         printf("overflow");  
105     }  
106     else{  
107         ptr->top++;  
108         ptr->arr[ptr->top] = val;  
109     }  
110 }  
111 char pop(struct stack *ptr){  
112     if (isEmpty(ptr)){  
113         printf("underflow");  
114     }  
115     else{  
116         char val = ptr->arr[ptr->top];  
117         ptr->top--;  
118         return val;  
119     }  
120 }  
121 int parenthesisMatch(char *exp){  
122     struct stack * sp;  
123     sp->size = 100;  
124     sp->top = -1;  
125     sp->arr = (char *)malloc(sp->size * sizeof(char));  
126     for (int i = 0; exp[i]!='\0'; i++){  
127         if (exp[i]=='('){  
128             push(sp, '(');  
129         }  
130         else if(exp[i]==')'){  
131             if (isEmpty(sp)){  
132                 return 0;
```

```
133         }
134         pop(sp);
135     }
136 }
137     if (isEmpty(sp)){
138         return 1;
139     }
140     else{
141         return 0;
142     }
143 }
144 int main(){
145     char * exp = "((45343ty536v2t4(2vtc32r)))";
146     if (parenthesisMatch(exp)){
147         printf("paranthesis is matching");
148     }
149     else{
150         printf("paranthesis is not matching");
151     }
152     return 0;
153 }
```

05- Stack\V34-multiParanthesis.c

```
97 //-----
98 // multiple paranthesis matching
99 #include <stdio.h>
100 #include <stdlib.h>
101 struct stack{
102     int size;
103     int top;
104     char *arr;
105 };
106 int isEmpty(struct stack *ptr){
107     if (ptr->top == -1){
108         return 1;
109     }
110     else{
111         return 0;
112     }
113 }
114 int isFull(struct stack *ptr){
115     if (ptr->top == ptr->size - 1){
116         return 1;
117     }
118     else{
119         return 0;
120     }
121 }
122 void push(struct stack *ptr, char val){
123     if (isFull(ptr)){
124         printf("overflowed");
125     }
126     else{
127         ptr->top++;
128         ptr->arr[ptr->top] = val;
129     }
130 }
131 char pop(struct stack *ptr){
132     if (isEmpty(ptr)){
133         printf("underflow");
134         return -1;
135     }
136     else{
137         char val = ptr->arr[ptr->top];
138         ptr->top--;
139         return val;
140     }
141 }
142 char stackTOP(struct stack *ptr){
143     return ptr->arr[ptr->top];
144 }
145
146 int match(char a, char b){
147     if (a == '(' && b == ')'){
148         return 1;
149     }
150     if (a == '{' && b == '}'){
151         return 1;
152     }
153     if (a == '[' && b == ']'){


```

```
154     return 1;
155 }
156 return 0;
157 }
158 int parenthesisMatch(char *exp){
159     char popped_ch;
160     struct stack *mmp = (struct stack *)malloc(sizeof(struct stack));
161     mmp->size = 100;
162     mmp->top = -1;
163     mmp->arr = (char *)malloc(mmp->size * sizeof(char));
164     for (int i = 0; exp[i] != '\0'; i++){
165         if (exp[i] == '(' || exp[i] == '{' || exp[i] == '['){
166             push(mmp, exp[i]);
167         }
168         else if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']'){
169             if (isEmpty(mmp)){
170                 return 0;
171             }
172             popped_ch = pop(mmp);
173             if (!match(popped_ch, exp[i])){
174                 return 0;
175             }
176         }
177     }
178     if (isEmpty(mmp)){
179         return 1;
180     }
181     else{
182         return 0;
183     }
184 }
185 int main(){
186     char *exp = "{3424[343]}";
187     if (parenthesisMatch(exp)){
188         printf("the paranthesis is balanced");
189     }
190     else{
191         printf("the parenthesis is not balanced");
192     }
193     return 0;
194 }
```

05- Stack\V37-infixPrefixPostfix.c

```

94 #include <stdio.h>
95 #include <stdlib.h>
96 #include <string.h>
97 struct stack{
98     int size;
99     int top;
100    char *arr;
101 };
102 int stackTop(struct stack* sp){
103     return sp->arr[sp->top];
104 }
105 int isEmpty(struct stack *ptr){
106     if (ptr->top == -1){
107         return 1;
108     }
109     else{
110         return 0;
111     }
112 }
113 int isFull(struct stack *ptr){
114     if (ptr->top == ptr->size - 1){
115         return 1;
116     }
117     else{
118         return 0;
119     }
120 }
121 void push(struct stack* ptr, char val){
122     if(isFull(ptr)){
123         printf("Stack Overflow! Cannot push %d to the stack\n", val);
124     }
125     else{
126         ptr->top++;
127         ptr->arr[ptr->top] = val;
128     }
129 }
130 char pop(struct stack* ptr){
131     if(isEmpty(ptr)){
132         printf("Stack Underflow! Cannot pop from the stack\n");
133         return -1;
134     }
135     else{
136         char val = ptr->arr[ptr->top];
137         ptr->top--;
138         return val;
139     }
140 }
141 int precedence(char ch){
142     if(ch == '*' || ch=='/')
143         return 3;
144     else if(ch == '+' || ch=='-')
145         return 2;
146     else
147         return 0;
148 }
149 int isOperator(char ch){
150     if(ch=='+' || ch=='-' || ch=='*' || ch=='/')

```

```
151     return 1;
152 else
153     return 0;
154 }
155 char* infixToPostfix(char* infix){
156     struct stack * sp = (struct stack *) malloc(sizeof(struct stack));
157     sp->size = 10;
158     sp->top = -1;
159     sp->arr = (char *) malloc(sp->size * sizeof(char));
160     char * postfix = (char *) malloc(strlen(infix)+1) * sizeof(char));
161     int i=0; // Track infix traversal
162     int j = 0; // Track postfix addition
163     while (infix[i]!='\0'){
164         if(!isOperator(infix[i])){
165             postfix[j] = infix[i];
166             j++;
167             i++;
168         }
169         else{
170             if(precedence(infix[i])> precedence(stackTop(sp))){
171                 push(sp, infix[i]);
172                 i++;
173             }
174             else{
175                 postfix[j] = pop(sp);
176                 j++;
177             }
178         }
179     }
180     while (!isEmpty(sp))
181     {
182         postfix[j] = pop(sp);
183         j++;
184     }
185     postfix[j] = '\0';
186     return postfix;
187 }
188 int main(){
189     char * infix = "x-y/z-k*d";
190     printf("postfix is %s", infixToPostfix(infix));
191 }
```

06 - Queue\V41-Queue.c

```
4 #include<stdio.h>
5 #include<stdlib.h>
6 struct queue{
7     int size,f,r;
8     int* arr;
9 };
10 int isEmpty(struct queue *q){
11     if(q->r==q->f){
12         return 1;
13     }
14     return 0;
15 }
16 int isFull(struct queue *q){
17     if(q->r==q->size-1){
18         return 1;
19     }
20     return 0;
21 }
22 void enqueue(struct queue *q, int val){
23     if(isFull(q)){
24         printf("This Queue is full\n");
25     }
26     else{
27         q->r++;
28         q->arr[q->r] = val;
29         printf("Enqueued element: %d\n", val);
30     }
31 }
32 int dequeue(struct queue *q){
33     int a = -1;
34     if(isEmpty(q)){
35         printf("This Queue is empty\n");
36     }
37     else{
38         q->f++;
39         a = q->arr[q->f];
40     }
41     return a;
42 }
43 int main(){
44     struct queue q;
45     q.size = 4;
46     q.f = q.r = 0;
47     q.arr = (int*) malloc(q.size*sizeof(int));
48     enqueue(&q, 12); // Enqueue few elements
49     printf("Dequeueing element %d\n", dequeue(&q));
50     enqueue(&q, 45);
51     if(isEmpty(&q)){
52         printf("Queue is empty\n");
53     }
54     if(isFull(&q)){
55         printf("Queue is full\n");
56     }
57     return 0;
58 }
```

06 - Queue\V44-CQ.c

```

50 //Circular Queue
51 #include<stdio.h>
52 #include<stdlib.h>
53 struct circularQueue{
54     int size;
55     int f;
56     int r;
57     int* arr;
58 };
59 int isEmpty(struct circularQueue *q){
60     if(q->r==q->f){
61         return 1;
62     }
63     return 0;
64 }
65 int isFull(struct circularQueue *q){
66     if((q->r+1)%q->size == q->f){
67         return 1;
68     }
69     return 0;
70 }
71 void enqueue(struct circularQueue *q, int val){
72     if(isFull(q)){
73         printf("This Queue is full");
74     }
75     else{
76         q->r = (q->r +1)%q->size;
77         q->arr[q->r] = val;
78         printf("Enqueued element: %d\n", val);
79     }
80 }
81 int dequeue(struct circularQueue *q){
82     int a = -1;
83     if(isEmpty(q)){
84         printf("This Queue is empty");
85     }
86     else{
87         q->f = (q->f +1)%q->size;
88         a = q->arr[q->f];
89     }
90     return a;
91 }
92 int main(){
93     struct circularQueue q;
94     q.size = 4;
95     q.f = q.r = 0;
96     q.arr = (int*) malloc(q.size*sizeof(int));
97     enqueue(&q, 12);    // Enqueue few elements
98     printf("Dequeueing element %d\n", dequeue(&q));
99     enqueue(&q, 45);
100    return 0;
101 }
```

06 - Queue\V46-LLQ.c

```
51 //Queue with linked list
52 #include <stdio.h>
53 #include <stdlib.h>
54 struct Node *f = NULL;
55 struct Node *r = NULL;
56 struct Node{
57     int data;
58     struct Node *next;
59 };
60 void linkedListTraversal(struct Node *ptr){
61     printf("Printing the elements of this linked list\n");
62     while (ptr != NULL){
63         printf("Element: %d\n", ptr->data);
64         ptr = ptr->next;
65     }
66 }
67 void enqueue(int val){
68     struct Node *n = (struct Node *) malloc(sizeof(struct Node));
69     if(n==NULL){
70         printf("Queue is Full");
71     }
72     else{
73         n->data = val;
74         n->next = NULL;
75         if(f==NULL){
76             f=r=n;
77         }
78         else{
79             r->next = n;
80             r=n;
81         }
82     }
83 }
84 int dequeue(){
85     int val = -1;
86     struct Node *ptr = f;
87     if(f==NULL){
88         printf("Queue is Empty\n");
89     }
90     else{
91         f = f->next;
92         val = ptr->data;
93         free(ptr);
94     }
95     return val;
96 }
97 int main(){
98     linkedListTraversal(f);
99     printf("Dequeueing element %d\n", dequeue());
100    enqueue(34);
101    printf("Dequeueing element %d\n", dequeue());
102    linkedListTraversal(f);
103    return 0;
104 }
```

06 - Sortings\V51-BBS.c

```
32 //Bubble sort
33 #include<stdio.h>
34 void printArray(int* A, int n){
35     for (int i = 0; i < n; i++)
36     {
37         printf("%d ", A[i]);
38     }
39     printf("\n");
40 }
41 void bubbleSort(int *A, int n){
42     int temp;
43     int isSorted = 0;
44     for (int i = 0; i < n-1; i++) // For number of pass
45     {
46         printf("Working on pass number %d\n", i+1);
47         for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
48         {
49             if(A[j]>A[j+1]){
50                 temp = A[j];
51                 A[j] = A[j+1];
52                 A[j+1] = temp;
53             }
54         }
55     }
56 }
57 void bubbleSortAdaptive(int *A, int n){
58     int temp;
59     int isSorted = 0;
60     for (int i = 0; i < n-1; i++){ // For number of pass
61         printf("Working on pass number %d\n", i+1);
62         isSorted = 1;
63         for (int j = 0; j <n-1-i ; j++){ // For comparison in each pass
64             if(A[j]>A[j+1]){
65                 temp = A[j];
66                 A[j] = A[j+1];
67                 A[j+1] = temp;
68                 isSorted = 0;
69             }
70         }
71         if(isSorted){
72             return;
73         }
74     }
75 }
76 int main(){
77     // int A[] = {12, 54, 65, 7, 23, 9};
78     int A[] = {1, 2, 5, 6, 12, 54, 625, 7, 23, 9, 987};
79     // int A[] = {1, 2, 3, 4, 5, 6};
80     int n = 11;
81     printArray(A, n); // Printing the array before sorting
82     bubbleSort(A, n); // Function to sort the array
83     printArray(A, n); // Printing the array before sorting
84     return 0;
85 }
```

06 - Sortings\V53-IS.c

```

37 //insersion sort
38 #include<stdio.h>
39 void printArray(int* A, int n){
40     for (int i = 0; i < n; i++){
41         printf("%d ", A[i]);
42     }
43     printf("\n");
44 }
45 void insertionSort(int *A, int n){
46     int key, j;
47     for (int i = 1; i <= n-1; i++){ // Loop for passes
48         key = A[i];
49         j = i-1;
50         while(j>=0 && A[j] > key){ // Loop for each pass
51             A[j+1] = A[j];
52             j--;
53         }
54         A[j+1] = key;
55     }
56 }
57 int main(){
58     // -1 0 1 2 3 4 5
59     // 12,| 54, 65, 07, 23, 09 --> i=1, key=54, j=0
60     // 12,| 54, 65, 07, 23, 09 --> 1st pass done (i=1)!
61     // 12, 54,| 65, 07, 23, 09 --> i=2, key=65, j=1
62     // 12, 54,| 65, 07, 23, 09 --> 2nd pass done (i=2)!
63     // 12, 54, 65,| 07, 23, 09 --> i=3, key=7, j=2
64     // 12, 54, 65,| 65, 23, 09 --> i=3, key=7, j=1
65     // 12, 54, 54,| 65, 23, 09 --> i=3, key=7, j=0
66     // 12, 12, 54,| 65, 23, 09 --> i=3, key=7, j=-1
67     // 07, 12, 54,| 65, 23, 09 --> i=3, key=7, j=-1--> 3rd pass done (i=3)!
68     // Fast forwarding and 4th and 5th pass will give:
69     // 07, 12, 54, 65,| 23, 09 --> i=4, key=23, j=3
70     // 07, 12, 23, 54,| 65, 09 --> After the 4th pass
71     // 07, 12, 23, 54, 65,| 09 --> i=5, key=09, j=4
72     // 07, 09, 12, 23, 54, 65| --> After the 5th pass
73     int A[] = {12, 54, 65, 7, 23, 9};
74     int n = 6;
75     printArray(A, n);
76     insertionSort(A, n);
77     printArray(A, n);
78     return 0;
79 }
```

06 - Sortings\V55-SS.c

```
14 //selection sort
15 #include<stdio.h>
16 void printArray(int* A, int n){
17     for (int i = 0; i < n; i++){
18         printf("%d ", A[i]);
19     }
20     printf("\n");
21 }
22 void selectionSort(int *A, int n){
23     int indexOfMin, temp;
24     printf("Running Selection sort...\\n");
25     for (int i = 0; i < n-1; i++){
26         indexOfMin = i;
27         for (int j = i+1; j < n; j++){
28             if(A[j] < A[indexOfMin]){
29                 indexOfMin = j;
30             }
31         }
32         // Swap A[i] and A[indexOfMin]
33         temp = A[i];
34         A[i] = A[indexOfMin];
35         A[indexOfMin] = temp;
36     }
37 }
38 int main(){
39     // Input Array (There will be total n-1 passes. 5-1 = 4 in this case!)
40     // 00 01 02 03 04
41     // |03, 05, 02, 13, 12
42     // After first pass
43     // 00 01 02 03 04
44     // 02,|05, 03, 13, 12
45     // After second pass
46     // 00 01 02 03 04
47     // 02, 03,|05, 13, 12
48     // After third pass
49     // 00 01 02 03 04
50     // 02, 03, 05,|13, 12
51     // After fourth pass
52     // 00 01 02 03 04
53     // 02, 03, 05, 12,|13
54     int A[] = {3, 5, 2, 13, 12};
55     int n = 5;
56     printArray(A, n);
57     selectionSort(A, n);
58     printArray(A, n);
59     return 0;
60 }
```

06 - Sortings\V56-QS.c

```
4 //Quick sort
5 #include <stdio.h>
6 void printArray(int *A, int n){
7     for (int i = 0; i < n; i++){
8         printf("%d ", A[i]);
9     }
10    printf("\n");
11 }
12 int partition(int A[], int low, int high){
13     int pivot = A[low];
14     int i = low + 1;
15     int j = high;
16     int temp;
17     do{
18         while (A[i] <= pivot){
19             i++;
20         }
21         while (A[j] > pivot){
22             j--;
23         }
24         if (i < j){
25             temp = A[i];
26             A[i] = A[j];
27             A[j] = temp;
28         }
29     } while (i < j);
30     temp = A[low];// Swap A[low] and A[j]
31     A[low] = A[j];
32     A[j] = temp;
33     return j;
34 }
35 void quickSort(int A[], int low, int high){
36     int partitionIndex; // Index of pivot after partition
37     if (low < high){
38         partitionIndex = partition(A, low, high);
39         quickSort(A, low, partitionIndex - 1); // sort left subarray
40         quickSort(A, partitionIndex + 1, high); // sort right subarray
41     }
42 }
43 int main(){
44     //int A[] = {3, 5, 2, 13, 12, 3, 2, 13, 45};
45     int A[] = {9, 4, 4, 8, 7, 5, 6};
46     // 3, 5, 2, 13, 12, 3, 2, 13, 45
47     // 3, 2, 2, 13i, 12, 3j, 5, 13, 45
48     // 3, 2, 2, 3j, 12i, 13, 5, 13, 45 --> first call to partition returns 3
49     int n = 9;
50     n = 7;
51     printArray(A, n);
52     quickSort(A, 0, n - 1);
53     printArray(A, n);
54     return 0;
55 }
```

06 - Sortings\V59-MS.c

```
11 //merge sort
12 #include <stdio.h>
13 void printArray(int *A, int n){
14     for (int i = 0; i < n; i++){
15         printf("%d ", A[i]);
16     }
17     printf("\n");
18 }
19 void merge(int A[], int mid, int low, int high){
20     int i, j, k, B[100];
21     i = low, j = mid + 1, k = low;
22     while (i <= mid && j <= high){
23         if (A[i] < A[j]){
24             B[k] = A[i];
25             i++;
26             k++;
27         }
28         else{
29             B[k] = A[j];
30             j++;
31             k++;
32         }
33     }
34     while (i <= mid){
35         B[k] = A[i];
36         k++;
37         i++;
38     }
39     while (j <= high){
40         B[k] = A[j];
41         k++;
42         j++;
43     }
44     for (int i = low; i <= high; i++){
45         A[i] = B[i];
46     }
47 }
48 void mergeSort(int A[], int low, int high){
49     int mid;
50     if(low<high){
51         mid = (low + high) /2;
52         mergeSort(A, low, mid);
53         mergeSort(A, mid+1, high);
54         merge(A, mid, low, high);
55     }
56 }
57 int main(){
58     int A[] = {9, 1, 4, 14, 4, 15, 6};
59     int n = 7;
60     printArray(A, n);
61     mergeSort(A, 0, 6);
62     printArray(A, n);
63     return 0;
64 }
```

06 - Sortings\V60-CS.c

```
4 //count sort
5 #include<stdio.h>
6 #include<limits.h>
7 #include<stdlib.h>
8 void printArray(int *A, int n){
9     for (int i = 0; i < n; i++){
10         printf("%d ", A[i]);
11     }
12     printf("\n");
13 }
14 int maximum(int A[], int n){
15     int max = INT_MIN;
16     for (int i = 0; i < n; i++){
17         if (max < A[i]){
18             max = A[i];
19         }
20     }
21     return max;
22 }
23 void countSort(int * A, int n){
24     int i, j;
25     // Find the maximum element in A
26     int max = maximum(A, n);
27     // Create the count array
28     int* count = (int *) malloc((max+1)*sizeof(int));
29     // Initialize the array elements to 0
30     for (i = 0; i < max+1; i++){
31         count[i] = 0;
32     }
33     // Increment the corresponding index in the count array
34     for (i = 0; i < n; i++){
35         count[A[i]] = count[A[i]] + 1;
36     }
37     i = 0; // counter for count array
38     j = 0; // counter for given array A
39     while(i <= max){
40         if(count[i] > 0){
41             A[j] = i;
42             count[i] = count[i] - 1;
43             j++;
44         }
45         else{
46             i++;
47         }
48     }
49 }
50 int main(){
51     int A[] = {9, 1, 4, 14, 4, 15, 6};
52     int n = 7;
53     printArray(A, n);
54     countSort(A, n);
55     printArray(A, n);
56     return 0;
57 }
```

07 - Binary Tree\V65-BT.c

```
44 //Linked list representation of binary tree
45 #include<stdio.h>
46 #include<malloc.h>
47 struct node{
48     int data;
49     struct node* left;
50     struct node* right;
51 };
52 struct node* createNode(int data){
53     struct node *n; // creating a node pointer
54     n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
55     n->data = data; // Setting the data
56     n->left = NULL; // Setting the left and right children to NULL
57     n->right = NULL; // Setting the left and right children to NULL
58     return n; // Finally returning the created node
59 }
60 int main(){
61     /*
62     // Constructing the root node
63     struct node *p;
64     p = (struct node *) malloc(sizeof(struct node));
65     p->data = 2;
66     p->left = NULL;
67     p->right = NULL;
68
69     // Constructing the second node
70     struct node *p1;
71     p1 = (struct node *) malloc(sizeof(struct node));
72     p1->data = 1;
73     p1->left = NULL;
74     p1->right = NULL;
75
76     // Constructing the third node
77     struct node *p2;
78     p2 = (struct node *) malloc(sizeof(struct node));
79     p2->data = 4;
80     p2->left = NULL;
81     p2->right = NULL;
82     */
83     // Constructing the root node - Using Function (Recommended)
84     struct node *p = createNode(2);
85     struct node *p1 = createNode(1);
86     struct node *p2 = createNode(4);
87     // Linking the root node with left and right children
88     p->left = p1;
89     p->right = p2;
90     return 0;
91 }
```

07 - Binary Tree\V70-OT.c

```

51 //PreOrder, PostOrder and InOrder tranversal in Tree
52 #include<stdio.h>
53 #include<malloc.h>
54 struct node{
55     int data;
56     struct node* left;
57     struct node* right;
58 };
59 struct node* createNode(int data){
60     struct node *n; // creating a node pointer
61     n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
62     n->data = data; // Setting the data
63     n->left = NULL; // Setting the left and right children to NULL
64     n->right = NULL; // Setting the left and right children to NULL
65     return n; // Finally returning the created node
66 }
67 void preOrder(struct node* root){
68     if(root!=NULL){
69         printf("%d ", root->data);
70         preOrder(root->left);
71         preOrder(root->right);
72     }
73 }
74 void postOrder(struct node* root){
75     if(root!=NULL){
76         postOrder(root->left);
77         postOrder(root->right);
78         printf("%d ", root->data);
79     }
80 }
81 void inOrder(struct node* root){
82     if(root!=NULL){
83         inOrder(root->left);
84         printf("%d ", root->data);
85         inOrder(root->right);
86     }
87 }
88 int main(){
89     // Constructing the root node - Using Function (Recommended) // Finally The tree looks
90     // like this:
91     struct node *p = createNode(4); //      4
92     struct node *p1 = createNode(1); //      /   |
93     struct node *p2 = createNode(6); //      1   6
94     struct node *p3 = createNode(5); //      /   |
95     struct node *p4 = createNode(2); //      5   2
96     // Linking the root node with left and right children
97     p->left = p1;
98     p->right = p2;
99     p1->left = p3;
100    p1->right = p4;
101    // preOrder(p); printf("\n"); postOrder(p);
102    inOrder(p);
103    return 0;
104 }
```

07 - Binary Tree\V72-!BST.c

```

52 //Checking if a binary tree is a binary search tree or not
53 #include<stdio.h>
54 #include<malloc.h>
55 struct node{
56     int data;struct node* left;struct node* right;
57 };
58 struct node* createNode(int data){
59     struct node *n; // creating a node pointer
60     n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
61     n->data = data; // Setting the data
62     n->left = NULL; // Setting the left and right children to NULL
63     n->right = NULL; // Setting the left and right children to NULL
64     return n; // Finally returning the created node
65 }
66 void preOrder(struct node* root);      //Functions
67 void postOrder(struct node* root);     //Functions
68 void inOrder(struct node* root);       //Functions
69 int isBST(struct node* root){
70     static struct node *prev = NULL;
71     if(root!=NULL){
72         if(!isBST(root->left)){
73             return 0;
74         }
75         if(prev!=NULL && root->data <= prev->data){
76             return 0;
77         }
78         prev = root;
79         return isBST(root->right);
80     }
81     else{
82         return 1;
83     }
84 }
85 int main(){
86     struct node *p = createNode(5);           // Constructing
the root node - Using Function (Recommended)
87     struct node *p1 = createNode(3);          // Finally The
tree looks like this:
88     struct node *p2 = createNode(6);          //      5
89     struct node *p3 = createNode(1);          //      / |
90     struct node *p4 = createNode(4);          //      3   6
91     p->left = p1;                          //      / |
92     p->right = p2; // Linking the root node with left and right children // 1   4
93     p1->left = p3;
94     p1->right = p4;
95     inOrder(p);    // preOrder(p); printf("\n"); postOrder(p);
96     printf("\n");  // printf("%d", isBST(p));
97     if(isBST(p)){
98         printf("This is a bst" );
99     }
100    else{
101        printf("This is not a bst");
102    }
103    return 0;
}

```

07 - Binary Tree\V74-BST.c

```

24 // Searching in a BST
25 #include<stdio.h>
26 #include<malloc.h>
27 struct node{
28     int data;
29     struct node* left;
30     struct node* right;
31 };
32 struct node* createNode(int data){
33     struct node *n; // creating a node pointer
34     n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
35     n->data = data; // Setting the data
36     n->left = NULL; // Setting the left and right children to NULL
37     n->right = NULL; // Setting the left and right children to NULL
38     return n; // Finally returning the created node
39 }
40 int isBST(struct node* root);           //FUNCTION
41 struct node * search(struct node* root, int key){
42     if(root==NULL){
43         return NULL;
44     }
45     if(key==root->data){
46         return root;
47     }
48     else if(key<root->data){
49         return search(root->left, key);
50     }
51     else{
52         return search(root->right, key);
53     }
54 }
55 int main(){
56     struct node *p = createNode(5);          // Constructing the root node - Using Function
57     (Recommended)                           // Finally The tree looks like this:
58     struct node *p1 = createNode(3);         //      5
59     struct node *p2 = createNode(6);         //      /   |
60     struct node *p3 = createNode(1);         //      3   6
61     struct node *p4 = createNode(4);         //      /   |
62     p->left = p1;                         //      1   4
63     p->right = p2;                        // Linking the root node with left and right
64     p1->left = p3;                        // children
65     p1->right = p4;
66     struct node* n = search(p, 10);
67     if(n!=NULL){
68         printf("Found: %d", n->data);
69     }
70     else{
71         printf("Element not found");
72     }
73     return 0;
74 }
```

07 - Binary Tree\V75-ISBST.c

```

28 //Iterative Search in a Binary Search Tree
29 #include<stdio.h>
30 #include<malloc.h>
31 struct node{
32     int data;
33     struct node* left;
34     struct node* right;
35 };
36 struct node* createNode(int data){
37     struct node *n; // creating a node pointer
38     n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
39     n->data = data; // Setting the data
40     n->left = NULL; // Setting the left and right children to NULL
41     n->right = NULL; // Setting the left and right children to NULL
42     return n; // Finally returning the created node
43 }
44 int isBST(struct node* root); //FUNCTION
45 struct node * searchIter(struct node* root, int key){
46     while(root!=NULL){
47         if(key == root->data){
48             return root;
49         }
50         else if(key<root->data){
51             root = root->left;
52         }
53         else{
54             root = root->right;
55         }
56     }
57     return NULL;
58 }
59 int main(){
60     struct node *p = createNode(5); //      5           // Finally The tree looks like this:
61     struct node *p1 = createNode(3); //      / |           // Constructing the root node -
Using Function (Recommended)
62     struct node *p2 = createNode(6); //      3   6
63     struct node *p3 = createNode(1); //      /   |
64     struct node *p4 = createNode(4); //      1   4
65     p->left = p1;// Linking the root node with left and right children
66     p->right = p2;
67     p1->left = p3;
68     p1->right = p4;
69     struct node* n = searchIter(p, 6);
70     if(n!=NULL){
71         printf("Found: %d", n->data);
72     }
73     else{
74         printf("Element not found");
75     }
76     return 0;
77 }
```

07 - Binary Tree\V76-insBST.c

```

45 //insertion in BST
46 #include<stdio.h>
47 #include<malloc.h>
48 struct node{
49     int data;struct node* left;struct node* right;
50 };
51 struct node* createNode(int data); //FUNCTION
52 int isBST(struct node* root); //FUNCTION
53 struct node * searchIter(struct node* root, int key){
54     while(root!=NULL){
55         if(key == root->data){
56             return root;
57         }
58         else if(key<root->data){
59             root = root->left;
60         }
61         else{
62             root = root->right;
63         }
64     }
65     return NULL;
66 }
67 void insert(struct node *root, int key){
68     struct node *prev = NULL;
69     while(root!=NULL){
70         prev = root;
71         if(key==root->data){
72             printf("Cannot insert %d, already in BST", key);
73             return;
74         }
75         else if(key<root->data){
76             root = root->left;
77         }
78         else{
79             root = root->right;
80         }
81     }
82     struct node* new = createNode(key);
83     if(key<prev->data){
84         prev->left = new;
85     }
86     else{
87         prev->right = new;
88     }
89 }
90 int main(){
91     struct node *p = createNode(5);
92     struct node *p1 = createNode(3);struct node *p2 = createNode(6);
93     struct node *p3 = createNode(1);struct node *p4 = createNode(4);
94     p->left = p1;p->right = p2;p1->left = p3;p1->right = p4;
95     insert(p, 16);
96     printf("%d", p->right->right->data);
97     return 0;
98 }
```

07 - Binary Tree\V78-delBST.c

```

25 //deletion in BST
26 #include<stdio.h>
27 #include<malloc.h>
28 struct node{
29     int data;struct node* left;struct node* right;
30 };
31 struct node* createNode(int data);           //FUNCTION
32 int isBST(struct node* root);               //FUNCTION
33 struct node * searchIter(struct node* root, int key); //FUNCTION
34 struct node *inOrderPredecessor(struct node* root){
35     root = root->left;
36     while (root->right!=NULL){
37         root = root->right;
38     }
39     return root;
40 }
41 struct node *deleteNode(struct node *root, int value){
42     struct node* iPre;
43     if (root == NULL){
44         return NULL;
45     }
46     if (root->left==NULL&&root->right==NULL){
47         free(root);
48         return NULL;
49     }
50     if (value < root->data){           //searching for the node to be deleted
51         root-> left = deleteNode(root->left,value);
52     }
53     else if (value > root->data){
54         root-> right = deleteNode(root->right,value);
55     }
56     else{                           //deletion strategy when the node is found
57         iPre = inOrderPredecessor(root);
58         root->data = iPre->data;
59         root->left = deleteNode(root->left, iPre->data);
60     }
61     return root;
62 }
63 int main(){
64     struct node *p = createNode(5);struct node *p1 = createNode(3);
65     struct node *p2 = createNode(6);struct node *p3 = createNode(1);
66     struct node *p4 = createNode(4);
67     p->left = p1;p->right = p2;p1->left = p3;p1->right = p4;
68     inOrder(p);
69     printf("\n");
70     deleteNode(p, 3);
71     inOrder(p);
72     return 0;
73 }
```

08 - AVL tree\V82-IRAVL.c

```

50 //AVL Tree Insertion & Rotation
51 #include <stdio.h>
52 #include <stdlib.h>
53 struct Node{
54     int key;
55     struct Node *left;
56     struct Node *right;
57     int height;
58 };
59 int getHeight(struct Node *n){
60     if(n==NULL)
61         return 0;
62     return n->height;
63 }
64 struct Node *createNode(int key){
65     struct Node* node = (struct Node *) malloc(sizeof(struct Node));
66     node->key = key;
67     node->left = NULL;
68     node->right = NULL;
69     node->height = 1;
70     return node;
71 }
72 int max (int a, int b){
73     return (a>b)?a:b;
74 }
75 int getBalanceFactor(struct Node * n){
76     if(n==NULL){
77         return 0;
78     }
79     return getHeight(n->left) - getHeight(n->right);
80 }
81 struct Node* rightRotate(struct Node* y){
82     struct Node* x = y->left;
83     struct Node* T2 = x->right;
84     x->right = y;
85     y->left = T2;
86     x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
87     y->height = max(getHeight(y->right), getHeight(y->left)) + 1;
88     return x;
89 }
90 struct Node* leftRotate(struct Node* x){
91     struct Node* y = x->right;
92     struct Node* T2 = y->left;
93     y->left = x;
94     x->right = T2;
95     x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
96     y->height = max(getHeight(y->right), getHeight(y->left)) + 1;
97     return y;
98 }
99 struct Node *insert(struct Node* node, int key){
100     if (node == NULL)
101         return createNode(key);
102     if (key < node->key)
103         node->left = insert(node->left, key);

```

```
104     else if (key > node->key)
105         node->right = insert(node->right, key);
106         node->height = 1 + max(getHeight(node->left), getHeight(node->right));
107     int bf = getBalanceFactor(node);
108     if(bf>1 && key < node->left->key){           // Left Left Case
109         return rightRotate(node);
110     }
111     if(bf<-1 && key > node->right->key){        // Right Right Case
112         return leftRotate(node);
113     }
114     if(bf>1 && key > node->left->key){           // Left Right Case
115         node->left = leftRotate(node->left);
116         return rightRotate(node);
117     }
118     if(bf<-1 && key < node->right->key){        // Right Left Case
119         node->right = rightRotate(node->right);
120         return leftRotate(node);
121     }
122     return node;
123 }
124 void preOrder(struct Node *root){
125     if(root != NULL)
126     {
127         printf("%d ", root->key);
128         preOrder(root->left);
129         preOrder(root->right);
130     }
131 }
132 int main(){
133     struct Node * root = NULL;
134     root = insert(root, 1);
135     root = insert(root, 2);
136     root = insert(root, 4);
137     root = insert(root, 5);
138     root = insert(root, 6);
139     root = insert(root, 3);
140     preOrder(root);
141     return 0;
142 }
```

09 - Graphs\V87-BFS.c

```
80 // BFS Implementation
81
82 #include<stdio.h>
83 #include<stdlib.h>
84
85 struct queue
86 {
87     int size;
88     int f;
89     int r;
90     int* arr;
91 };
92
93
94 int isEmpty(struct queue *q){
95     if(q->r==q->f){
96         return 1;
97     }
98     return 0;
99 }
100
101 int isFull(struct queue *q){
102     if(q->r==q->size-1){
103         return 1;
104     }
105     return 0;
106 }
107
108 void enqueue(struct queue *q, int val){
109     if(isFull(q)){
110         printf("This Queue is full\n");
111     }
112     else{
113         q->r++;
114         q->arr[q->r] = val;
115         // printf("Enqueued element: %d\n", val);
116     }
117 }
118
119 int dequeue(struct queue *q){
120     int a = -1;
121     if(isEmpty(q)){
122         printf("This Queue is empty\n");
123     }
124     else{
125         q->f++;
126         a = q->arr[q->f];
127     }
128     return a;
129 }
130
131 int main(){
132     // Initializing Queue (Array Implementation)
133     struct queue q;
```

```
134     q.size = 400;
135     q.f = q.r = 0;
136     q.arr = (int*) malloc(q.size*sizeof(int));
137
138 // BFS Implementation
139 int node;
140 int i = 1;
141 int visited[7] = {0,0,0,0,0,0,0};
142 int a [7][7] = {
143     {0,1,1,1,0,0,0},
144     {1,0,1,0,0,0,0},
145     {1,1,0,1,1,0,0},
146     {1,0,1,0,1,0,0},
147     {0,0,1,1,0,1,1},
148     {0,0,0,0,1,0,0},
149     {0,0,0,0,1,0,0}
150 };
151 printf("%d", i);
152 visited[i] = 1;
153 enqueue(&q, i); // Enqueue i for exploration
154 while (!isEmpty(&q))
155 {
156     int node = dequeue(&q);
157     for (int j = 0; j < 7; j++)
158     {
159         if(a[node][j] == 1 && visited[j] == 0){
160             printf("%d", j);
161             visited[j] = 1;
162             enqueue(&q, j);
163         }
164     }
165 }
166 return 0;
167 }
```

09 - Graphs\V89-DFS.c

```
33 // DFS Implementation
34
35 #include<stdio.h>
36 #include<stdlib.h>
37
38 int visited[7] = {0,0,0,0,0,0,0};
39     int A [7][7] = {
40         {0,1,1,1,0,0,0},
41         {1,0,1,0,0,0,0},
42         {1,1,0,1,1,0,0},
43         {1,0,1,0,1,0,0},
44         {0,0,1,1,0,1,1},
45         {0,0,0,0,1,0,0},
46         {0,0,0,0,1,0,0}
47     };
48
49 void DFS(int i){
50     printf("%d ", i);
51     visited[i] = 1;
52     for (int j = 0; j < 7; j++)
53     {
54         if(A[i][j]==1 && !visited[j]){
55             DFS(j);
56         }
57     }
58 }
59
60 int main(){
61     // DFS Implementation
62     DFS(0);
63     return 0;
64 }
```