

0 - Include Library

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <ctype.h>
```

1 - Lined List

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef int ElementType;
5 typedef struct Node *PtrToNode;
6 typedef PtrToNode List;
7 typedef PtrToNode Position;
8
9 struct Node
10 {
11     ElementType Element;
12     Position Next;
13 };
14
15 List MakeEmpty( List L )
16 {
17     if( L != NULL )
18         DeleteList( L );
19     L = malloc( sizeof( struct Node ) );
20     if( L == NULL )
21         printf( "Out of memory!" );
22     L->Next = NULL;
23     return L;
24 }
25
26 /* START: fig3_8.txt */
27 /* Return true if L is empty */
28
29 int IsEmpty( List L )
30 {
31     return L->Next == NULL;
32 }
33 /* END */
34
35 /* START: fig3_9.txt */
36 /* Return true if P is the last position in list L */
37 /* Parameter L is unused in this implementation */
38 int IsLast( Position P, List L )
39 {
40     return P->Next == NULL;
41 }
42 /* END */
43
44 /* START: fig3_10.txt */
45 /* Return Position of X in L; NULL if not found */
46 Position Find( ElementType X, List L )
47 {
48     Position P;
49
```

```
50     P = L->Next;
51     while( P != NULL && P->Element != X )
52         P = P->Next;
53
54     return P;
55 }
56 /* END */
57
58 /* START: fig3_11.txt */
59 /* Delete from a list */
60 /* Cell pointed to by P->Next is wiped out */
61 /* Assume that the position is legal */
62 /* Assume use of a header node */
63 void Delete( ElementType X, List L )
64 {
65     Position P, TmpCell;
66     P = FindPrevious( X, L );
67
68     if( !IsLast( P, L ) ) /* Assumption of header use */
69     {                       /* X is found; delete it */
70         TmpCell = P->Next;
71         P->Next = TmpCell->Next; /* Bypass deleted cell */
72         free( TmpCell );
73     }
74 }
75 /* END */
76
77 /* START: fig3_12.txt */
78 /* If X is not found, then Next field of returned value is NULL */
79 /* Assumes a header */
80 Position FindPrevious( ElementType X, List L )
81 {
82     Position P;
83     P = L;
84
85     while( P->Next != NULL && P->Next->Element != X )
86         P = P->Next;
87
88     return P;
89 }
90 /* END */
91
92 /* START: fig3_13.txt */
93 /* Insert (after legal position P) */
94 /* Header implementation assumed */
95 /* Parameter L is unused in this implementation */
96
97 void Insert( ElementType X, List L, Position P )
98 {
99     Position TmpCell;
100
101     TmpCell = malloc( sizeof( struct Node ) );
102     if( TmpCell == NULL )
103         printf( "Out of space!!!" );
104
105     TmpCell->Element = X;
106     TmpCell->Next = P->Next;
107     P->Next = TmpCell;
```

```
108 }
109 /* END */
110
111 /* START: fig3_15.txt */
112 /* Correct DeleteList algorithm */
113 void DeleteList( List L )
114 {
115     Position P, Tmp;
116
117     P = L->Next; /* Header assumed */
118     L->Next = NULL;
119     while( P != NULL )
120     {
121         Tmp = P->Next;
122         free( P );
123         P = Tmp;
124     }
125 }
126 /* END */
127
128 Position Header( List L )
129 {
130     return L;
131 }
132
133 Position First( List L )
134 {
135     return L->Next;
136 }
137
138 Position Advance( Position P )
139 {
140     return P->Next;
141 }
142
143 ElementType Retrieve( Position P )
144 {
145     return P->Element;
146 }
147
148 void PrintList( const List L )
149 {
150     Position P = Header( L );
151
152     if( IsEmpty( L ) )
153         printf( "Empty list\n" );
154     else
155     {
156         do
157         {
158             P = Advance( P );
159             printf( "%d ", Retrieve( P ) );
160         } while( !IsLast( P, L ) );
161         printf( "\n" );
162     }
163 }
164
165 main( )
```

```
166 {
167     List L;
168     Position P;
169     int i;
170
171     L = MakeEmpty( NULL );
172     P = Header( L );
173     PrintList( L );
174
175     for( i = 0; i < 10; i++ )
176     {
177         Insert( i, L, P );
178         PrintList( L );
179         P = Advance( P );
180     }
181     for( i = 0; i < 10; i+= 2 )
182         Delete( i, L );
183
184     for( i = 0; i < 10; i++ )
185         if( ( i % 2 == 0 ) == ( Find( i, L ) != NULL ) )
186             printf( "Find fails\n" );
187
188     printf( "Finished deletions\n" );
189
190     PrintList( L );
191
192     DeleteList( L );
193
194     return 0;
195 }
```

2 - Stack (Array)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define EmptyTOS ( -1 )
5 #define MinStackSize ( 5 )
6
7 typedef int ElementType;
8
9 struct StackRecord
10 {
11     int Capacity;
12     int TopOfStack;
13     ElementType *Array;
14 };
15 typedef struct StackRecord *Stack;
16
17 int IsEmpty(Stack S);
18 int IsFull(Stack S);
19 void MakeEmpty(Stack S);
20 Stack CreateStack(int MaxElements);
21 void DisposeStack( Stack S );
22 void Push( ElementType X, Stack S );
23 ElementType Top( Stack S );
24 void Pop( Stack S );
25 ElementType TopAndPop( Stack S );
26
27 /* START: fig3_48.txt */
28 int IsEmpty( Stack S )
29 {
30     return S->TopOfStack == EmptyTOS;
31 }
32 /* END */
33
34 int IsFull( Stack S )
35 {
36     return S->TopOfStack == S->Capacity - 1;
37 }
38
39 /* START: fig3_49.txt */
40 void MakeEmpty( Stack S )
41 {
42     S->TopOfStack = EmptyTOS;
43 }
44 /* END */
45
46 /* START: fig3_46.txt */
47 Stack CreateStack( int MaxElements )
48 {
49     Stack S;
50
51     if( MaxElements < MinStackSize )
52         printf( "Stack size is too small" );
53
54     S = malloc( sizeof( struct StackRecord ) );
55     if( S == NULL )
56         printf( "Out of space!!!" );
```

```
57
58     S->Array = malloc( sizeof( ElementType ) * MaxElements );
59     if( S->Array == NULL )
60         printf( "Out of space!!!" );
61     S->Capacity = MaxElements;
62     MakeEmpty( S );
63
64     return S;
65 }
66 /* END */
67
68
69
70 /* START: fig3_47.txt */
71 void DisposeStack( Stack S )
72 {
73     if( S != NULL )
74     {
75         free( S->Array );
76         free( S );
77     }
78 }
79 /* END */
80
81 /* START: fig3_50.txt */
82 void Push( ElementType X, Stack S )
83 {
84     if( IsFull( S ) )
85         printf( "Full stack" );
86     else
87         S->Array[ ++S->TopOfStack ] = X;
88 }
89 /* END */
90
91
92 /* START: fig3_51.txt */
93 ElementType Top( Stack S )
94 {
95     if( !IsEmpty( S ) )
96         return S->Array[ S->TopOfStack ];
97     printf( "Empty stack" );
98     return 0; /* Return value used to avoid warning */
99 }
100 /* END */
101
102 /* START: fig3_52.txt */
103 void Pop( Stack S )
104 {
105     if( IsEmpty( S ) )
106         printf( "Empty stack" );
107     else
108         S->TopOfStack--;
109 }
110 /* END */
111
112 /* START: fig3_53.txt */
113 ElementType TopAndPop( Stack S )
114 {
```

```
115     if( !IsEmpty( S ) )
116         return S->Array[ S->TopOfStack-- ];
117     printf( "Empty stack" );
118     return 0; /* Return value used to avoid warning */
119 }
120 /* END */
121
122 main( )
123 {
124     Stack S;
125     int i;
126
127     S = CreateStack( 12 );
128     for( i = 0; i < 10; i++ )
129         Push( i, S );
130
131     while( !IsEmpty( S ) )
132     {
133         printf( "%d\n", Top( S ) );
134         Pop( S );
135     }
136
137     DisposeStack( S );
138     return 0;
139 }
```

2 - Stack (Linked list)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef int ElementType;
5 typedef struct Node *PtrToNode;
6 typedef PtrToNode Stack;
7
8 struct Node
9 {
10     ElementType Element;
11     PtrToNode Next;
12 };
13
14 /* START: fig3_40.txt */
15 int IsEmpty( Stack S )
16 {
17     return S->Next == NULL;
18 }
19 /* END */
20
21 /* START: fig3_41.txt */
22 Stack CreateStack( void )
23 {
24     Stack S;
25
26     S = malloc( sizeof( struct Node ) );
27     if( S == NULL )
28         printf( "Out of space!!!" );
29     S->Next = NULL;
30     MakeEmpty( S );
31     return S;
32 }
33
34 void MakeEmpty( Stack S )
35 {
36     if( S == NULL )
37         printf( "Must use CreateStack first" );
38     else
39         while( !IsEmpty( S ) )
40             Pop( S );
41 }
42 /* END */
43
44 void DisposeStack( Stack S )
45 {
46     MakeEmpty( S );
47     free( S );
48 }
49
50 /* START: fig3_42.txt */
51 void Push( ElementType X, Stack S )
52 {
53     PtrToNode TmpCell;
54
55     TmpCell = malloc( sizeof( struct Node ) );
56     if( TmpCell == NULL )
```



```
57     printf( "Out of space!!!" );
58     else
59     {
60         TmpCell->Element = X;
61         TmpCell->Next = S->Next;
62         S->Next = TmpCell;
63     }
64 }
65 /* END */
66
67 /* START: fig3_43.txt */
68 ElementType Top( Stack S )
69 {
70     if( !IsEmpty( S ) )
71         return S->Next->Element;
72     printf( "Empty stack" );
73     return 0; /* Return value used to avoid warning */
74 }
75 /* END */
76
77 /* START: fig3_44.txt */
78 void Pop( Stack S )
79 {
80     PtrToNode FirstCell;
81
82     if( IsEmpty( S ) )
83         printf( "Empty stack" );
84     else
85     {
86         FirstCell = S->Next;
87         S->Next = S->Next->Next;
88         free( FirstCell );
89     }
90 }
91 /* END */
92
93 main( )
94 {
95     Stack S;
96     int i;
97
98     S = CreateStack( );
99     for( i = 0; i < 10; i++ )
100         Push( i, S );
101
102     while( !IsEmpty( S ) )
103     {
104         printf( "%d\n", Top( S ) );
105         Pop( S );
106     }
107
108     DisposeStack( S );
109     return 0;
110 }
```

3 - Queue

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MinQueueSize ( 5 )
5
6 typedef int ElementType;
7 struct QueueRecord
8 {
9     int Capacity;
10    int Front;
11    int Rear;
12    int Size;
13    ElementType *Array;
14 };
15 typedef struct QueueRecord *Queue;
16
17 int IsEmpty( Queue Q )
18 {
19     return Q->Size == 0;
20 }
21
22 int IsFull( Queue Q )
23 {
24     return Q->Size == Q->Capacity;
25 }
26
27 Queue CreateQueue( int MaxElements )
28 {
29     Queue Q;
30
31     if( MaxElements < MinQueueSize )
32         printf( "Queue size is too small" );
33
34     Q = malloc( sizeof( struct QueueRecord ) );
35     if( Q == NULL )
36         printf( "Out of space!!!" );
37
38     Q->Array = malloc( sizeof( ElementType ) * MaxElements );
39     if( Q->Array == NULL )
40         printf( "Out of space!!!" );
41     Q->Capacity = MaxElements;
42     MakeEmpty( Q );
43
44     return Q;
45 }
46
47 /* START: fig3_59.txt */
48 void MakeEmpty( Queue Q )
49 {
50     Q->Size = 0;
51     Q->Front = 1;
52     Q->Rear = 0;
53 }
54 /* END */
55
56 void DisposeQueue( Queue Q )
```

```
57 {
58     if( Q != NULL )
59     {
60         free( Q->Array );
61         free( Q );
62     }
63 }
64
65 /* START: fig3_60.txt */
66 static int Succ( int Value, Queue Q )
67 {
68     if( ++Value == Q->Capacity )
69         Value = 0;
70     return Value;
71 }
72
73 void Enqueue( ElementType X, Queue Q )
74 {
75     if( IsFull( Q ) )
76         printf( "Full queue" );
77     else
78     {
79         Q->Size++;
80         Q->Rear = Succ( Q->Rear, Q );
81         Q->Array[ Q->Rear ] = X;
82     }
83 }
84 /* END */
85
86
87
88 ElementType Front( Queue Q )
89 {
90     if( !IsEmpty( Q ) )
91         return Q->Array[ Q->Front ];
92     printf( "Empty queue" );
93     return 0; /* Return value used to avoid warning */
94 }
95
96 void Dequeue( Queue Q )
97 {
98     if( IsEmpty( Q ) )
99         printf( "Empty queue" );
100     else
101     {
102         Q->Size--;
103         Q->Front = Succ( Q->Front, Q );
104     }
105 }
106
107 ElementType FrontAndDequeue( Queue Q )
108 {
109     ElementType X = 0;
110
111     if( IsEmpty( Q ) )
112         printf( "Empty queue" );
113     else
114     {
```

```
115     Q->Size--;
116     X = Q->Array[ Q->Front ];
117     Q->Front = Succ( Q->Front, Q );
118 }
119 return X;
120 }
121
122 main( )
123 {
124     Queue Q;
125     int i;
126
127     Q = CreateQueue( 12 );
128
129     for( i = 0; i < 10; i++ )
130         Enqueue( i, Q );
131
132     while( !IsEmpty( Q ) )
133     {
134         printf( "%d\n", Front( Q ) );
135         Dequeue( Q );
136     }
137     for( i = 0; i < 10; i++ )
138         Enqueue( i, Q );
139
140     while( !IsEmpty( Q ) )
141     {
142         printf( "%d\n", Front( Q ) );
143         Dequeue( Q );
144     }
145
146     DisposeQueue( Q );
147     return 0;
148 }
```

4 - BST Traverse

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct node{
5     int val;
6     struct node* left;
7     struct node* right;
8 } Node;
9
10 // Depth first search
11 void dfs(Node* root){
12     if(root == NULL) return;
13
14     Node** stack = malloc(sizeof(Node*)*100);
15     int top = 0;
16
17     stack[top++] = root;
18
19     while(top > 0){
20         top--;
21         Node* current = stack[top];
22
23         printf("%d ", current->val);
24
25         if(current->right != NULL){
26             stack[top++] = current->right;
27         }
28
29         if(current->left != NULL){
30             stack[top++] = current->left;
31         }
32     }
33
34     free(stack);
35 }
36
37 // Breadth first search
38 void bfs(Node* root){
39
40     if(root == NULL) return;
41
42     Node** queue = malloc(sizeof(Node*) * 100);
43     int front = 0;
44     int back = 0;
45
46     queue[back++] = root;
47
48     while(front < back){
49
50         Node* current = queue[front++];
51
52         printf("%d ", current->val);
53
54         if(current->left != NULL){
55             queue[back++] = current->left;
56         }
57     }
58 }
```

```
57
58     if(current->right != NULL){
59         queue[back++] = current->right;
60     }
61
62 }
63
64 free(queue);
65
66 }
67 int main(){
68     // Create tree
69     //      1
70     //    /  \
71     //   2    3
72     //  / \  / \
73     // 4  5 6  7
74
75     //DFS traverse order: 1 2 4 5 3 6 7
76     //BFS traverse order: 1 2 3 4 5 6 7
77
78     Node root = {1, NULL, NULL};
79     Node node2 = {2, NULL, NULL};
80     Node node3 = {3, NULL, NULL};
81     Node node4 = {4, NULL, NULL};
82     Node node5 = {5, NULL, NULL};
83     Node node6 = {6, NULL, NULL};
84     Node node7 = {7, NULL, NULL};
85
86     root.left = &node2;
87     root.right = &node3;
88     node2.left = &node4;
89     node2.right = &node5;
90     node3.left = &node6;
91     node3.right = &node7;
92
93     // Traverse tree
94     dfs(&root);
95     printf("\n");
96
97     bfs(&root);
98
99     return 0;
100 }
```

4 - BST

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef int ElementType;
5 typedef struct TreeNode *Position;
6 typedef struct TreeNode *SearchTree;
7
8 struct TreeNode
9 {
10     ElementType Element;
11     SearchTree Left;
12     SearchTree Right;
13 };
14
15 /* START: fig4_17.txt */
16 SearchTree MakeEmpty( SearchTree T )
17 {
18     if( T != NULL )
19     {
20         MakeEmpty( T->Left );
21         MakeEmpty( T->Right );
22         free( T );
23     }
24     return NULL;
25 }
26 /* END */
27
28 /* START: fig4_18.txt */
29 Position Find( ElementType X, SearchTree T )
30 {
31     if( T == NULL )
32         return NULL;
33     if( X < T->Element )
34         return Find( X, T->Left );
35     else if( X > T->Element )
36         return Find( X, T->Right );
37     else
38         return T;
39 }
40 /* END */
41
42 /* START: fig4_19.txt */
43 Position FindMin( SearchTree T )
44 {
45     if( T == NULL )
46         return NULL;
47     else if( T->Left == NULL )
48         return T;
49     else
50         return FindMin( T->Left );
51 }
52 /* END */
53
54 /* START: fig4_20.txt */
55 Position FindMax( SearchTree T )
56 {
```

```
57     if( T != NULL )
58     while( T->Right != NULL )
59         T = T->Right;
60
61     return T;
62 }
63 /* END */
64
65 /* START: fig4_22.txt */
66 SearchTree Insert( ElementType X, SearchTree T )
67 {
68     if( T == NULL )
69     {
70         /* Create and return a one-node tree */
71         T = malloc( sizeof( struct TreeNode ) );
72         if( T == NULL )
73             printf( "Out of space!!!" );
74         else
75         {
76             T->Element = X;
77             T->Left = T->Right = NULL;
78         }
79     }
80     else if( X < T->Element )
81         T->Left = Insert( X, T->Left );
82     else if( X > T->Element )
83         T->Right = Insert( X, T->Right );
84     /* Else X is in the tree already; we'll do nothing */
85
86     return T; /* Do not forget this line!! */
87 }
88 /* END */
89
90 /* START: fig4_25.txt */
91 SearchTree Delete( ElementType X, SearchTree T )
92 {
93     Position TmpCell;
94
95     if( T == NULL )
96         printf( "Element not found" );
97     else if( X < T->Element ) /* Go left */
98         T->Left = Delete( X, T->Left );
99     else if( X > T->Element ) /* Go right */
100         T->Right = Delete( X, T->Right );
101     else if( T->Left && T->Right ) /* Two children */
102     {
103         /* Replace with smallest in right subtree */
104         TmpCell = FindMin( T->Right );
105         T->Element = TmpCell->Element;
106         T->Right = Delete( T->Element, T->Right );
107     }
108     else /* One or zero children */
109     {
110         TmpCell = T;
111         if( T->Left == NULL ) /* Also handles 0 children */
112             T = T->Right;
113         else if( T->Right == NULL )
114             T = T->Left;
```



```
115     free( TmpCell );
116 }
117
118     return T;
119 }
120 /* END */
121
122 ElementType Retrieve( Position P )
123 {
124     return P->Element;
125 }
126
127 main( )
128 {
129     SearchTree T;
130     Position P;
131     int i;
132     int j = 0;
133
134     T = MakeEmpty( NULL );
135     for( i = 0; i < 50; i++, j = ( j + 7 ) % 50 )
136         T = Insert( j, T );
137     for( i = 0; i < 50; i++ )
138         if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
139             printf( "Error at %d\n", i );
140
141     for( i = 0; i < 50; i += 2 )
142         T = Delete( i, T );
143
144     for( i = 1; i < 50; i += 2 )
145         if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
146             printf( "Error at %d\n", i );
147     for( i = 0; i < 50; i += 2 )
148         if( ( P = Find( i, T ) ) != NULL )
149             printf( "Error at %d\n", i );
150
151     printf( "Min is %d, Max is %d\n", Retrieve( FindMin( T ) ),
152           Retrieve( FindMax( T ) ) );
153
154     return 0;
155 }
```

5 - AVL Tree

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef int ElementType;
5 typedef struct AvlNode *Position;
6 typedef struct AvlNode *AvlTree;
7
8 struct AvlNode
9 {
10     ElementType Element;
11     AvlTree Left;
12     AvlTree Right;
13     int Height;
14 };
15
16 AvlTree MakeEmpty( AvlTree T )
17 {
18     if( T != NULL )
19     {
20         MakeEmpty( T->Left );
21         MakeEmpty( T->Right );
22         free( T );
23     }
24     return NULL;
25 }
26
27 Position Find( ElementType X, AvlTree T )
28 {
29     if( T == NULL ) return NULL;
30     if( X < T->Element ) return Find( X, T->Left );
31     else if( X > T->Element ) return Find( X, T->Right );
32     else return T;
33 }
34
35 Position FindMin( AvlTree T )
36 {
37     if( T == NULL ) return NULL;
38     else if( T->Left == NULL ) return T;
39     else return FindMin( T->Left );
40 }
41
42 Position FindMax( AvlTree T )
43 {
44     if( T != NULL )
45         while( T->Right != NULL )
46             T = T->Right;
47
48     return T;
49 }
50
51 /* START: fig4_36.txt */
52 static int Height( Position P )
53 {
54     if( P == NULL ) return -1;
55     else return P->Height;
56 }
```

```
57 /* END */
58
59 static int Max( int Lhs, int Rhs )
60 {
61     return Lhs > Rhs ? Lhs : Rhs;
62 }
63
64 /* START: fig4_39.txt */
65 /* This function can be called only if K2 has a left child */
66 /* Perform a rotate between a node (K2) and its left child */
67 /* Update heights, then return new root */
68
69 static Position SingleRotateWithLeft( Position K2 )
70 {
71     Position K1;
72
73     K1 = K2->Left;
74     K2->Left = K1->Right;
75     K1->Right = K2;
76
77     K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
78     K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;
79
80     return K1; /* New root */
81 }
82 /* END */
83
84 /* This function can be called only if K1 has a right child */
85 /* Perform a rotate between a node (K1) and its right child */
86 /* Update heights, then return new root */
87
88 static Position SingleRotateWithRight( Position K1 )
89 {
90     Position K2;
91
92     K2 = K1->Right;
93     K1->Right = K2->Left;
94     K2->Left = K1;
95
96     K1->Height = Max( Height( K1->Left ), Height( K1->Right ) ) + 1;
97     K2->Height = Max( Height( K2->Right ), K1->Height ) + 1;
98
99     return K2; /* New root */
100 }
101
102 /* START: fig4_41.txt */
103     /* This function can be called only if K3 has a left */
104     /* child and K3's left child has a right child */
105     /* Do the left-right double rotation */
106     /* Update heights, then return new root */
107
108 static Position DoubleRotateWithLeft( Position K3 )
109 {
110     /* Rotate between K1 and K2 */
111     K3->Left = SingleRotateWithRight( K3->Left );
112
113     /* Rotate between K3 and K2 */
114     return SingleRotateWithLeft( K3 );
```

```
115 }
116 /* END */
117
118 /* This function can be called only if K1 has a right */
119 /* child and K1's right child has a left child */
120 /* Do the right-left double rotation */
121 /* Update heights, then return new root */
122
123 static Position DoubleRotateWithRight( Position K1 )
124 {
125     /* Rotate between K3 and K2 */
126     K1->Right = SingleRotateWithLeft( K1->Right );
127
128     /* Rotate between K1 and K2 */
129     return SingleRotateWithRight( K1 );
130 }
131
132
133 /* START: fig4_37.txt */
134 AvlTree Insert( ElementType X, AvlTree T )
135 {
136     if( T == NULL )
137     {
138         /* Create and return a one-node tree */
139         T = malloc( sizeof( struct AvlNode ) );
140         if( T == NULL )
141             printf( "Out of space!!!" );
142         else
143         {
144             T->Element = X; T->Height = 0;
145             T->Left = T->Right = NULL;
146         }
147     }
148     else if( X < T->Element )
149     {
150         T->Left = Insert( X, T->Left );
151         if( Height( T->Left ) - Height( T->Right ) == 2 )
152             if( X < T->Left->Element )
153                 T = SingleRotateWithLeft( T );
154             else
155                 T = DoubleRotateWithLeft( T );
156     }
157     else if( X > T->Element )
158     {
159         T->Right = Insert( X, T->Right );
160         if( Height( T->Right ) - Height( T->Left ) == 2 )
161             if( X > T->Right->Element )
162                 T = SingleRotateWithRight( T );
163             else
164                 T = DoubleRotateWithRight( T );
165     }
166     /* Else X is in the tree already; we'll do nothing */
167
168     T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
169     return T;
170 }
171 /* END */
172
```

```
173 AvlTree Delete( ElementType X, AvlTree T )
174 {
175     printf( "Sorry; Delete is unimplemented; %d remains\n", X );
176     return T;
177 }
178
179 ElementType Retrieve( Position P )
180 {
181     return P->Element;
182 }
183
184 main( )
185 {
186     AvlTree T;
187     Position P;
188     int i;
189     int j = 0;
190
191     T = MakeEmpty( NULL );
192     for( i = 0; i < 50; i++, j = ( j + 7 ) % 50 )
193         T = Insert( j, T );
194     for( i = 0; i < 50; i++ )
195         if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
196             printf( "Error at %d\n", i );
197
198     /* for( i = 0; i < 50; i += 2 )
199         T = Delete( i, T );
200
201     for( i = 1; i < 50; i += 2 )
202         if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
203             printf( "Error at %d\n", i );
204     for( i = 0; i < 50; i += 2 )
205         if( ( P = Find( i, T ) ) != NULL )
206             printf( "Error at %d\n", i );
207 */
208     printf( "Min is %d, Max is %d\n", Retrieve( FindMin( T ) ),
209            Retrieve( FindMax( T ) ) );
210
211     return 0;
212 }
```

6 - Heap

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MaxSize (1000)
5 #define MinPQSize (10)
6 #define MinData (-32767)
7
8 typedef int ElementType;
9 struct HeapStruct
10 {
11     int Capacity;
12     int Size;
13     ElementType *Elements;
14 };
15
16 typedef struct HeapStruct *PriorityQueue;
17
18 /* START: fig6_0.txt */
19 PriorityQueue Initialize( int MaxElements )
20 {
21     PriorityQueue H;
22
23     if( MaxElements < MinPQSize )
24         printf( "Priority queue size is too small" );
25
26     H = malloc( sizeof( struct HeapStruct ) );
27     if( H ==NULL )
28         printf( "Out of space!!!" );
29
30     /* Allocate the array plus one extra for sentinel */
31     H->Elements = malloc( ( MaxElements + 1 ) * sizeof( ElementType ) )
32     ;
33     if( H->Elements == NULL )
34         printf( "Out of space!!!" );
35
36     H->Capacity = MaxElements;
37     H->Size = 0;
38     H->Elements[ 0 ] = MinData;
39
40     return H;
41 }
42 /* END */
43
44 void MakeEmpty( PriorityQueue H )
45 {
46     H->Size = 0;
47 }
48
49 /* START: fig6_8.txt */
50 /* H->Element[ 0 ] is a sentinel */
51 void Insert( ElementType X, PriorityQueue H )
52 {
53     int i;
54
55     if( IsFull( H ) )
56     {
```

```
56     printf( "Priority queue is full" );
57     return;
58 }
59
60 for( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
61     H->Elements[ i ] = H->Elements[ i / 2 ];
62 H->Elements[ i ] = X;
63 }
64 /* END */
65
66 /* START: fig6_12.txt */
67 ElementType DeleteMin( PriorityQueue H )
68 {
69     int i, Child;
70     ElementType MinElement, LastElement;
71
72     if( IsEmpty( H ) )
73     {
74         printf( "Priority queue is empty" );
75         return H->Elements[ 0 ];
76     }
77     MinElement = H->Elements[ 1 ];
78     LastElement = H->Elements[ H->Size-- ];
79
80     for( i = 1; i * 2 <= H->Size; i = Child )
81     {
82         /* Find smaller child */
83         Child = i * 2;
84         if( Child != H->Size && H->Elements[ Child + 1 ] < H->Elements[
85             Child ] )
86             Child++;
87
88         /* Percolate one level */
89         if( LastElement > H->Elements[ Child ] )
90             H->Elements[ i ] = H->Elements[ Child ];
91         else
92             break;
93     }
94     H->Elements[ i ] = LastElement;
95     return MinElement;
96 }
97 /* END */
98
99 ElementType FindMin( PriorityQueue H )
100 {
101     if( !IsEmpty( H ) )
102         return H->Elements[ 1 ];
103     printf( "Priority Queue is Empty" );
104     return H->Elements[ 0 ];
105 }
106
107 int IsEmpty( PriorityQueue H )
108 {
109     return H->Size == 0;
110 }
111
112 int IsFull( PriorityQueue H )
113 {
```

```
113     return H->Size == H->Capacity;
114 }
115
116 void Destroy( PriorityQueue H )
117 {
118     free( H->Elements );
119     free( H );
120 }
121
122 main( )
123 {
124     PriorityQueue H;
125     int i, j;
126
127     H = Initialize( MaxSize );
128     for( i=0, j=MaxSize/2; i<MaxSize; i++, j=( j+71)%MaxSize )
129         Insert( j, H );
130
131     j = 0;
132     while( !IsEmpty( H ) )
133         if( DeleteMin( H ) != j++ )
134             printf( "Error in DeleteMin, %d\n", j );
135     printf( "Done...\n" );
136     return 0;
137 }
```