# 0 - Include Library

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
```

# 1 - Lined List

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

typedef int ElementType;
typedef struct Node *PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

struct Node
{
    ElementType Element;
    Position Next;
};

List MakeEmpty( List L )
{
    if( L != NULL )
        DeleteList( L );
    L = malloc( sizeof( struct Node ) );
    if( L == NULL )
        printf( "Out of memory!" );
    L->Next = NULL;
    return L;
}

/* START: fig3_8.txt */
/* Return true if L is empty */

int IsEmpty( List L )
{
    return L->Next == NULL;
}
/* END */

/* START: fig3_9.txt */
/* Return true if P is the last position in list L */
/* Parameter L is unused in this implementation */
int IsLast( Position P, List L )
{
    return P->Next == NULL;
}
/* END */

/* START: fig3_10.txt */
/* Return Position of X in L; NULL if not found */
Position Find( ElementType X, List L )
{
    Position P;

```

```
    P = L->Next;
    while( P != NULL && P->Element != X )
        P = P->Next;

    return P;
}
/* END */

/* START: fig3_11.txt */
/* Delete from a list */
/* Cell pointed to by P->Next is wiped out */
/* Assume that the position is legal */
/* Assume use of a header node */
void Delete( ElementType X, List L )
{
    Position P, TmpCell;
    P = FindPrevious( X, L );

    if( !IsLast( P, L ) )  /* Assumption of header use */
    {                      /* X is found; delete it */
        TmpCell = P->Next;
        P->Next = TmpCell->Next;  /* Bypass deleted cell */
        free( TmpCell );
    }
}
/* END */

/* START: fig3_12.txt */
/* If X is not found, then Next field of returned value is NULL */
/* Assumes a header */
Position FindPrevious( ElementType X, List L )
{
    Position P;
    P = L;

    while( P->Next != NULL && P->Next->Element != X )
        P = P->Next;

    return P;
}
/* END */

/* START: fig3_13.txt */
/* Insert (after legal position P) */
/* Header implementation assumed */
/* Parameter L is unused in this implementation */

void Insert( ElementType X, List L, Position P )
{
    Position TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if( TmpCell == NULL )
        printf( "Out of space!!!" );

    TmpCell->Element = X;
    TmpCell->Next = P->Next;
    P->Next = TmpCell;
```

```
}
/* END */

/* START: fig3_15.txt */
/* Correct DeleteList algorithm */
void DeleteList( List L )
{
    Position P, Tmp;

    P = L->Next;  /* Header assumed */
    L->Next = NULL;
    while( P != NULL )
    {
        Tmp = P->Next;
        free( P );
        P = Tmp;
    }
}
/* END */

Position Header( List L )
{
    return L;
}

Position First( List L )
{
    return L->Next;
}

Position Advance( Position P )
{
    return P->Next;
}

ElementType Retrieve( Position P )
{
    return P->Element;
}

void PrintList( const List L )
{
    Position P = Header( L );

    if( IsEmpty( L ) )
        printf( "Empty list\n" );
    else
    {
        do
        {
            P = Advance( P );
            printf( "%d ", Retrieve( P ) );
        } while( !IsLast( P, L ) );
        printf( "\n" );
    }
}

main( )
```

```
{
    List L;
    Position P;
    int i;

    L = MakeEmpty( NULL );
    P = Header( L );
    PrintList( L );

    for( i = 0; i < 10; i++ )
    {
        Insert( i, L, P );
        PrintList( L );
        P = Advance( P );
    }
    for( i = 0; i < 10; i+= 2 )
        Delete( i, L );

    for( i = 0; i < 10; i++ )
        if( ( i % 2 == 0 ) == ( Find( i, L ) != NULL ) )
            printf( "Find fails\n" );

    printf( "Finished deletions\n" );

    PrintList( L );

    DeleteList( L );

    return 0;
}
```

# 2 - Stack (Array)

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

#define EmptyTOS ( -1 )
#define MinStackSize ( 5 )

typedef int ElementType;

struct StackRecord
{
    int Capacity;
    int TopOfStack;
    ElementType *Array;
};
typedef struct StackRecord *Stack;

int IsEmpty(Stack S);
int IsFull(Stack S);
void MakeEmpty(Stack S);
Stack CreateStack(int MaxElements);
void DisposeStack( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );
ElementType TopAndPop( Stack S );

/* START: fig3_48.txt */
int IsEmpty( Stack S )
{
    return S->TopOfStack == EmptyTOS;
}
/* END */

int IsFull( Stack S )
{
    return S->TopOfStack == S->Capacity - 1;
}

/* START: fig3_49.txt */
void MakeEmpty( Stack S )
{
    S->TopOfStack = EmptyTOS;
}
/* END */

/* START: fig3_46.txt */
Stack CreateStack( int MaxElements )
{
    Stack S;

    if( MaxElements < MinStackSize )
        printf( "Stack size is too small" );

    S = malloc( sizeof( struct StackRecord ) );
    if( S == NULL )
        printf( "Out of space!!!" );
```

```c

    S->Array = malloc( sizeof( ElementType ) * MaxElements );
    if( S->Array == NULL )
        printf( "Out of space!!!" );
    S->Capacity = MaxElements;
    MakeEmpty( S );

    return S;
}
/* END */



/* START: fig3_47.txt */
void DisposeStack( Stack S )
{
    if( S != NULL )
    {
        free( S->Array );
        free( S );
    }
}
/* END */

/* START: fig3_50.txt */
void Push( ElementType X, Stack S )
{
    if( IsFull( S ) )
        printf( "Full stack" );
    else
        S->Array[ ++S->TopOfStack ] = X;
}
/* END */


/* START: fig3_51.txt */
ElementType Top( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack ];
    printf( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
/* END */

/* START: fig3_52.txt */
void Pop( Stack S )
{
    if( IsEmpty( S ) )
        printf( "Empty stack" );
    else
        S->TopOfStack--;
}
/* END */

/* START: fig3_53.txt */
ElementType TopAndPop( Stack S )
{
```

```c
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack-- ];
    printf( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
/* END */

main( )
{
    Stack S;
    int i;

    S = CreateStack( 12 );
    for( i = 0; i < 10; i++ )
        Push( i, S );

    while( !IsEmpty( S ) )
    {
        printf( "%d\n", Top( S ) );
        Pop( S );
    }

    DisposeStack( S );
    return 0;
}
```

# 2 - Stack (Linked list)

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

typedef int ElementType;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

struct Node
{
    ElementType Element;
    PtrToNode   Next;
};

/* START: fig3_40.txt */
int IsEmpty( Stack S )
{
    return S->Next == NULL;
}
/* END */

/* START: fig3_41.txt */
Stack CreateStack( void )
{
    Stack S;

    S = malloc( sizeof( struct Node ) );
    if( S == NULL )
        printf( "Out of space!!!" );
    S->Next = NULL;
    MakeEmpty( S );
    return S;
}

void MakeEmpty( Stack S )
{
    if( S == NULL )
        printf( "Must use CreateStack first" );
    else
        while( !IsEmpty( S ) )
            Pop( S );
}
/* END */

void DisposeStack( Stack S )
{
    MakeEmpty( S );
    free( S );
}

/* START: fig3_42.txt */
void Push( ElementType X, Stack S )
{
    PtrToNode TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if( TmpCell == NULL )
```

```
        printf( "Out of space!!!" );
    else
    {
        TmpCell->Element = X;
        TmpCell->Next = S->Next;
        S->Next = TmpCell;
    }
}
/* END */

/* START: fig3_43.txt */
ElementType Top( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Next->Element;
    printf( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
/* END */

/* START: fig3_44.txt */
void Pop( Stack S )
{
    PtrToNode FirstCell;

    if( IsEmpty( S ) )
        printf( "Empty stack" );
    else
    {
        FirstCell = S->Next;
        S->Next = S->Next->Next;
        free( FirstCell );
    }
}
/* END */

main( )
{
    Stack S;
    int i;

    S = CreateStack( );
    for( i = 0; i < 10; i++ )
        Push( i, S );

    while( !IsEmpty( S ) )
    {
        printf( "%d\n", Top( S ) );
        Pop( S );
    }

    DisposeStack( S );
    return 0;
}
```

# 3 - Queue

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

#define MinQueueSize ( 5 )

typedef int ElementType;
struct QueueRecord
{
    int Capacity;
    int Front;
    int Rear;
    int Size;
    ElementType *Array;
};
typedef struct QueueRecord *Queue;

int IsEmpty( Queue Q )
{
    return Q->Size == 0;
}

int IsFull( Queue Q )
{
    return Q->Size == Q->Capacity;
}

Queue CreateQueue( int MaxElements )
{
    Queue Q;

    if( MaxElements < MinQueueSize )
        printf( "Queue size is too small" );

    Q = malloc( sizeof( struct QueueRecord ) );
    if( Q == NULL )
        printf( "Out of space!!!" );

    Q->Array = malloc( sizeof( ElementType ) * MaxElements );
    if( Q->Array == NULL )
        printf( "Out of space!!!" );
    Q->Capacity = MaxElements;
    MakeEmpty( Q );

    return Q;
}

/* START: fig3_59.txt */
void MakeEmpty( Queue Q )
{
    Q->Size = 0;
    Q->Front = 1;
    Q->Rear = 0;
}
/* END */

void DisposeQueue( Queue Q )
```

```
{
    if( Q != NULL )
    {
        free( Q->Array );
        free( Q );
    }
}

/* START: fig3_60.txt */
static int Succ( int Value, Queue Q )
{
    if( ++Value == Q->Capacity )
        Value = 0;
    return Value;
}

void Enqueue( ElementType X, Queue Q )
{
    if( IsFull( Q ) )
        printf( "Full queue" );
    else
    {
        Q->Size++;
        Q->Rear = Succ( Q->Rear, Q );
        Q->Array[ Q->Rear ] = X;
    }
}
/* END */



ElementType Front( Queue Q )
{
    if( !IsEmpty( Q ) )
        return Q->Array[ Q->Front ];
    printf( "Empty queue" );
    return 0;  /* Return value used to avoid warning */
}

void Dequeue( Queue Q )
{
    if( IsEmpty( Q ) )
        printf( "Empty queue" );
    else
    {
        Q->Size--;
        Q->Front = Succ( Q->Front, Q );
    }
}

ElementType FrontAndDequeue( Queue Q )
{
    ElementType X = 0;

    if( IsEmpty( Q ) )
        printf( "Empty queue" );
    else
    {
```

```
        Q->Size--;
        X = Q->Array[ Q->Front ];
        Q->Front = Succ( Q->Front, Q );
    }
    return X;
}

main( )
{
    Queue Q;
    int i;

    Q = CreateQueue( 12 );

    for( i = 0; i < 10; i++ )
        Enqueue( i, Q );

    while( !IsEmpty( Q ) )
    {
        printf( "%d\n", Front( Q ) );
        Dequeue( Q );
    }
    for( i = 0; i < 10; i++ )
        Enqueue( i, Q );

    while( !IsEmpty( Q ) )
    {
        printf( "%d\n", Front( Q ) );
        Dequeue( Q );
    }

    DisposeQueue( Q );
    return 0;
}
```

# 4 - BST Traverse

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

typedef struct node{
    int val;
    struct node* left;
    struct node* right;
} Node;

// Depth first search
void dfs(Node* root){
    if(root == NULL) return;

    Node** stack = malloc(sizeof(Node*)*100);
    int top = 0;

    stack[top++] = root;

    while(top > 0){
        top--;
        Node* current = stack[top];

        printf("%d ", current->val);

        if(current->right != NULL){
            stack[top++] = current->right;
        }

        if(current->left != NULL){
            stack[top++] = current->left;
        }
    }

    free(stack);
}

// Breadth first search
void bfs(Node* root){

    if(root == NULL) return;

    Node** queue = malloc(sizeof(Node*) * 100);
    int front = 0;
    int back = 0;

    queue[back++] = root;

    while(front < back){

        Node* current = queue[front++];

        printf("%d ", current->val);

        if(current->left != NULL){
            queue[back++] = current->left;
        }
```

```c

        if(current->right != NULL){
            queue[back++] = current->right;
        }

    }

    free(queue);

}
int main(){
    // Create tree
    //       1
    //     /   \
    //    2     3
    //   / \   / \
    //  4   5 6   7

    //DFS traverse order: 1 2 4 5 3 6 7
    //BFS traverse order: 1 2 3 4 5 6 7

    Node root = {1, NULL, NULL};
    Node node2 = {2, NULL, NULL};
    Node node3 = {3, NULL, NULL};
    Node node4 = {4, NULL, NULL};
    Node node5 = {5, NULL, NULL};
    Node node6 = {6, NULL, NULL};
    Node node7 = {7, NULL, NULL};

    root.left = &node2;
    root.right = &node3;
    node2.left = &node4;
    node2.right = &node5;
    node3.left = &node6;
    node3.right = &node7;

    // Traverse tree
    dfs(&root);
    printf("\n");

    bfs(&root);

    return 0;
}
```

# 4 - BST

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

typedef int ElementType;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;

struct TreeNode
{
    ElementType Element;
    SearchTree  Left;
    SearchTree  Right;
};

/* START: fig4_17.txt */
SearchTree MakeEmpty( SearchTree T )
{
    if( T != NULL )
    {
        MakeEmpty( T->Left );
        MakeEmpty( T->Right );
        free( T );
    }
    return NULL;
}
/* END */

/* START: fig4_18.txt */
Position Find( ElementType X, SearchTree T )
{
    if( T == NULL )
        return NULL;
    if( X < T->Element )
        return Find( X, T->Left );
    else if( X > T->Element )
        return Find( X, T->Right );
    else
        return T;
}
/* END */

/* START: fig4_19.txt */
Position FindMin( SearchTree T )
{
    if( T == NULL )
        return NULL;
    else if( T->Left == NULL )
        return T;
    else
        return FindMin( T->Left );
}
/* END */

/* START: fig4_20.txt */
Position FindMax( SearchTree T )
{
```

```c
    if( T != NULL )
    while( T->Right != NULL )
        T = T->Right;

    return T;
}
/* END */

/* START: fig4_22.txt */
SearchTree Insert( ElementType X, SearchTree T )
{
    if( T == NULL )
    {
        /* Create and return a one-node tree */
        T = malloc( sizeof( struct TreeNode ) );
        if( T == NULL )
            printf( "Out of space!!!" );
        else
        {
            T->Element = X;
            T->Left = T->Right = NULL;
        }
    }
    else if( X < T->Element )
        T->Left = Insert( X, T->Left );
    else if( X > T->Element )
        T->Right = Insert( X, T->Right );
    /* Else X is in the tree already; we'll do nothing */

    return T;  /* Do not forget this line!! */
}
/* END */

/* START: fig4_25.txt */
SearchTree Delete( ElementType X, SearchTree T )
{
    Position TmpCell;

    if( T == NULL )
        printf( "Element not found" );
    else if( X < T->Element )  /* Go left */
        T->Left = Delete( X, T->Left );
    else if( X > T->Element )  /* Go right */
        T->Right = Delete( X, T->Right );
    else if( T->Left && T->Right )  /* Two children */
    {
        /* Replace with smallest in right subtree */
        TmpCell = FindMin( T->Right );
        T->Element = TmpCell->Element;
        T->Right = Delete( T->Element, T->Right );
    }
    else  /* One or zero children */
    {
        TmpCell = T;
        if( T->Left == NULL ) /* Also handles 0 children */
            T = T->Right;
        else if( T->Right == NULL )
            T = T->Left;
```

```
        free( TmpCell );
    }

    return T;
}
/* END */

ElementType Retrieve( Position P )
{
    return P->Element;
}

main( )
{
    SearchTree T;
    Position P;
    int i;
    int j = 0;

    T = MakeEmpty( NULL );
    for( i = 0; i < 50; i++, j = ( j + 7 ) % 50 )
        T = Insert( j, T );
    for( i = 0; i < 50; i++ )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );

    for( i = 0; i < 50; i += 2 )
        T = Delete( i, T );

    for( i = 1; i < 50; i += 2 )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );
    for( i = 0; i < 50; i += 2 )
        if( ( P = Find( i, T ) ) != NULL )
            printf( "Error at %d\n", i );

    printf( "Min is %d, Max is %d\n", Retrieve( FindMin( T ) ),
               Retrieve( FindMax( T ) ) );

    return 0;
}
```

# 5 - AVL Tree

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

typedef int ElementType;
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;

struct AvlNode
{
    ElementType Element;
    AvlTree Left;
    AvlTree Right;
    int Height;
};

AvlTree MakeEmpty( AvlTree T )
{
    if( T != NULL )
    {
        MakeEmpty( T->Left );
        MakeEmpty( T->Right );
        free( T );
    }
    return NULL;
}

Position Find( ElementType X, AvlTree T )
{
    if( T == NULL ) return NULL;
    if( X < T->Element ) return Find( X, T->Left );
    else if( X > T->Element ) return Find( X, T->Right );
    else return T;
}

Position FindMin( AvlTree T )
{
    if( T == NULL ) return NULL;
    else if( T->Left == NULL ) return T;
    else return FindMin( T->Left );
}

Position FindMax( AvlTree T )
{
    if( T != NULL )
        while( T->Right != NULL )
            T = T->Right;

    return T;
}

/* START: fig4_36.txt */
static int Height( Position P )
{
    if( P == NULL ) return -1;
    else return P->Height;
}
```

```
/* END */

static int Max( int Lhs, int Rhs )
{
    return Lhs > Rhs ? Lhs : Rhs;
}

/* START: fig4_39.txt */
/* This function can be called only if K2 has a left child */
/* Perform a rotate between a node (K2) and its left child */
/* Update heights, then return new root */

static Position SingleRotateWithLeft( Position K2 )
{
    Position K1;

    K1 = K2->Left;
    K2->Left = K1->Right;
    K1->Right = K2;

    K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
    K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;

    return K1;  /* New root */
}
/* END */

/* This function can be called only if K1 has a right child */
/* Perform a rotate between a node (K1) and its right child */
/* Update heights, then return new root */

static Position SingleRotateWithRight( Position K1 )
{
    Position K2;

    K2 = K1->Right;
    K1->Right = K2->Left;
    K2->Left = K1;

    K1->Height = Max( Height( K1->Left ), Height( K1->Right ) ) + 1;
    K2->Height = Max( Height( K2->Right ), K1->Height ) + 1;

    return K2;  /* New root */
}

/* START: fig4_41.txt */
        /* This function can be called only if K3 has a left */
        /* child and K3's left child has a right child */
        /* Do the left-right double rotation */
        /* Update heights, then return new root */

static Position DoubleRotateWithLeft( Position K3 )
{
    /* Rotate between K1 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left );

    /* Rotate between K3 and K2 */
    return SingleRotateWithLeft( K3 );
```

```
}
/* END */

/* This function can be called only if K1 has a right */
/* child and K1's right child has a left child */
/* Do the right-left double rotation */
/* Update heights, then return new root */

static Position DoubleRotateWithRight( Position K1 )
{
    /* Rotate between K3 and K2 */
    K1->Right = SingleRotateWithLeft( K1->Right );

    /* Rotate between K1 and K2 */
    return SingleRotateWithRight( K1 );
}


/* START: fig4_37.txt */
AvlTree Insert( ElementType X, AvlTree T )
{
    if( T == NULL )
    {
        /* Create and return a one-node tree */
        T = malloc( sizeof( struct AvlNode ) );
        if( T == NULL )
            printf( "Out of space!!!" );
        else
        {
            T->Element = X; T->Height = 0;
            T->Left = T->Right = NULL;
        }
    }
    else if( X < T->Element )
    {
        T->Left = Insert( X, T->Left );
        if( Height( T->Left ) - Height( T->Right ) == 2 )
            if( X < T->Left->Element )
                T = SingleRotateWithLeft( T );
            else
                T = DoubleRotateWithLeft( T );
    }
    else if( X > T->Element )
    {
        T->Right = Insert( X, T->Right );
        if( Height( T->Right ) - Height( T->Left ) == 2 )
            if( X > T->Right->Element )
                T = SingleRotateWithRight( T );
            else
                T = DoubleRotateWithRight( T );
    }
    /* Else X is in the tree already; we'll do nothing */

    T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
    return T;
}
/* END */
```

```c
AvlTree Delete( ElementType X, AvlTree T )
{
    printf( "Sorry; Delete is unimplemented; %d remains\n", X );
    return T;
}

ElementType Retrieve( Position P )
{
    return P->Element;
}

main( )
{
    AvlTree T;
    Position P;
    int i;
    int j = 0;

    T = MakeEmpty( NULL );
    for( i = 0; i < 50; i++, j = ( j + 7 ) % 50 )
        T = Insert( j, T );
    for( i = 0; i < 50; i++ )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );

 /* for( i = 0; i < 50; i += 2 )
        T = Delete( i, T );

    for( i = 1; i < 50; i += 2 )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );
    for( i = 0; i < 50; i += 2 )
        if( ( P = Find( i, T ) ) != NULL )
            printf( "Error at %d\n", i );
*/
    printf( "Min is %d, Max is %d\n", Retrieve( FindMin( T ) ),
                Retrieve( FindMax( T ) ) );

    return 0;
}
```

# 6 - Heap

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

#define MaxSize (1000)
#define MinPQSize (10)
#define MinData (-32767)

typedef int ElementType;
struct HeapStruct
{
    int Capacity;
    int Size;
    ElementType *Elements;
};

typedef struct HeapStruct *PriorityQueue;

/* START: fig6_0.txt */
PriorityQueue Initialize( int MaxElements )
{
    PriorityQueue H;

    if( MaxElements < MinPQSize )
        printf( "Priority queue size is too small" );

    H = malloc( sizeof( struct HeapStruct ) );
    if( H ==NULL )
        printf( "Out of space!!!" );

    /* Allocate the array plus one extra for sentinel */
    H->Elements = malloc( ( MaxElements + 1 ) * sizeof( ElementType ) );
    if( H->Elements == NULL )
        printf( "Out of space!!!" );

    H->Capacity = MaxElements;
    H->Size = 0;
    H->Elements[ 0 ] = MinData;

    return H;
}
/* END */

void MakeEmpty( PriorityQueue H )
{
    H->Size = 0;
}

/* START: fig6_8.txt */
/* H->Element[ 0 ] is a sentinel */
void Insert( ElementType X, PriorityQueue H )
{
    int i;

    if( IsFull( H ) )
    {
```

```
        printf( "Priority queue is full" );
        return;
    }

    for( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];
    H->Elements[ i ] = X;
}
/* END */

/* START: fig6_12.txt */
ElementType DeleteMin( PriorityQueue H )
{
    int i, Child;
    ElementType MinElement, LastElement;

    if( IsEmpty( H ) )
    {
        printf( "Priority queue is empty" );
        return H->Elements[ 0 ];
    }
    MinElement = H->Elements[ 1 ];
    LastElement = H->Elements[ H->Size-- ];

    for( i = 1; i * 2 <= H->Size; i = Child )
    {
        /* Find smaller child */
        Child = i * 2;
        if( Child != H->Size && H->Elements[ Child + 1 ] < H->Elements[ Child ] )
            Child++;

        /* Percolate one level */
        if( LastElement > H->Elements[ Child ] )
            H->Elements[ i ] = H->Elements[ Child ];
        else
            break;
    }
    H->Elements[ i ] = LastElement;
    return MinElement;
}
/* END */

ElementType FindMin( PriorityQueue H )
{
    if( !IsEmpty( H ) )
        return H->Elements[ 1 ];
    printf( "Priority Queue is Empty" );
    return H->Elements[ 0 ];
    }

int IsEmpty( PriorityQueue H )
{
    return H->Size == 0;
}

int IsFull( PriorityQueue H )
{
```

```c
    return H->Size == H->Capacity;
}

void Destroy( PriorityQueue H )
{
    free( H->Elements );
    free( H );
}

main( )
{
    PriorityQueue H;
    int i, j;

    H = Initialize( MaxSize );
    for( i=0, j=MaxSize/2; i<MaxSize; i++, j=( j+71)%MaxSize )
        Insert( j, H );

    j = 0;
    while( !IsEmpty( H ) )
        if( DeleteMin( H ) != j++ )
            printf( "Error in DeleteMin, %d\n", j );
    printf( "Done...\n" );
    return 0;
}
```