```
<html><head><style></style><script type="text/javascript" charset="utf-8"
src="https://browser.gwdang.com/js/gwdang-notifier.js"></script><script
src="https://s11.cnzz.com/z_stat.php?id=1256793290&amp;web_id=1256793290" async="" defer="">
</script></head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">
```



```
#include &lt;stdio.h&gt;

/* START: fig1_2.txt */
        int
        F( int X )
        {
/* 1*/      if( X == 0 )
/* 2*/          return 0;
            else
/* 3*/          return 2 * F( X - 1 ) + X * X;
        }
/* END */

main( )
{
    printf( "F(5) = %d\n", F( 5 ) );
    return 0;
}
```

```
#include &lt;stdio.h&gt;

/* START: fig1_3.txt */
        int
        Bad( unsigned int N )
        {
/* 1*/      if( N == 0 )
/* 2*/          return 0;
            else
/* 3*/          return Bad( N / 3 + 1 ) + N - 1;
        }
/* END */

main( )
{
    printf( "Bad is infinite recursion\n" );
    return 0;
}
```



```
#include &lt;stdio.h&gt;

#define PrintDigit( Ch )      ( putchar( ( Ch ) + '0' ) )

/* START: fig1_4.txt */
        void
        PrintOut( unsigned int N )  /* Print nonnegative N */
        {
            if( N &gt;= 10 )
                PrintOut( N / 10 );
```

```
                PrintOut( N / 10 );
            PrintDigit( N % 10 );
        }
/* END */

main( )
{
    PrintOut( 1369 );
    putchar( '\n' );
    return 0;
}
```



```
#include &lt;stdio.h&gt;

/* Define one of CubicAlgorithm, QuadraticAlgorithm, NlogNAlgorithm,
 * or LinearAlgorithm to get one algorithm compiled */

#define NlogNAlgorithm

#ifdef CubicAlgorithm

/* START: fig2_5.txt */
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j, k;

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i &lt; N; i++ )
/* 3*/          for( j = i; j &lt; N; j++ )
                {
```

```
                {
/* 4*/              ThisSum = 0;
/* 5*/              for( k = i; k &lt;= j; k++ )
/* 6*/                  ThisSum += A[ k ];

/* 7*/              if( ThisSum &gt; MaxSum )
/* 8*/                  MaxSum = ThisSum;
                }
/* 9*/      return MaxSum;
        }
/* END */

#endif


#ifdef QuadraticAlgorithm

/* START: fig2_6.txt */
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j;

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i &lt; N; i++ )
            {
/* 3*/          ThisSum = 0;
/* 4*/          for( j = i; j &lt; N; j++ )
                {
/* 5*/              ThisSum += A[ j ];

/* 6*/              if( ThisSum &gt; MaxSum )
/* 7*/                  MaxSum = ThisSum;
                }
            }
/* 8*/      return MaxSum;
        }
/* END */
```



```
#endif


#ifdef NlogNAlgorithm

        static int
        Max3( int A, int B, int C )
        {
            return A &gt; B ? A &gt; C ? A : C : B &gt; C ? B : C;
        }

/* START: fig2_7.txt */
        static int
        MaxSubSum( const int A[ ], int Left, int Right )
        {
            int MaxLeftSum, MaxRightSum;
            int MaxLeftBorderSum, MaxRightBorderSum;
            int LeftBorderSum, RightBorderSum;
            int Center, i;

/* 1*/      if( Left == Right )  /* Base case */
/* 2*/          if( A[ Left ] &gt; 0 )
/* 3*/              return A[ Left ];
                else
```

```
/* 4*/                return 0;

/* 5*/        Center = ( Left + Right ) / 2;
/* 6*/        MaxLeftSum = MaxSubSum( A, Left, Center );
/* 7*/        MaxRightSum = MaxSubSum( A, Center + 1, Right );

/* 8*/        MaxLeftBorderSum = 0; LeftBorderSum = 0;
/* 9*/        for( i = Center; i &gt;= Left; i-- )
              {
/*10*/            LeftBorderSum += A[ i ];
/*11*/            if( LeftBorderSum &gt; MaxLeftBorderSum )
/*12*/                MaxLeftBorderSum = LeftBorderSum;
              }

/*13*/        MaxRightBorderSum = 0; RightBorderSum = 0;
/*14*/        for( i = Center + 1; i &lt;= Right; i++ )
              {
/*15*/            RightBorderSum += A[ i ];
/*16*/            if( RightBorderSum &gt; MaxRightBorderSum )
/*17*/                MaxRightBorderSum = RightBorderSum;
              }

/*18*/        return Max3( MaxLeftSum, MaxRightSum,
/*19*/                            MaxLeftBorderSum + MaxRightBorderSum );
          }

          int
          MaxSubsequenceSum( const int A[ ], int N )
```



```
          {
              return MaxSubSum( A, 0, N - 1 );
          }
/* END */


#endif


#ifdef LinearAlgorithm
/* START: fig2_8.txt */
          int
          MaxSubsequenceSum( const int A[ ], int N )
          {
              int ThisSum, MaxSum, j;

/* 1*/        ThisSum = MaxSum = 0;
/* 2*/        for( j = 0; j &lt; N; j++ )
              {
/* 3*/            ThisSum += A[ j ];

/* 4*/            if( ThisSum &gt; MaxSum )
/* 5*/                MaxSum = ThisSum;
/* 6*/            else if( ThisSum &lt; 0 )
/* 7*/                ThisSum = 0;
              }
/* 8*/        return MaxSum;
          }
/* END */


#endif

main( )
{
    static int A[ ] = { 4, -3, 5, -2, -1, 2, 6, -2 };
```

```
    printf( "Maxsum = %d\n",
            MaxSubsequenceSum( A, sizeof( A ) / sizeof( A[ 0 ] ) ) );
    return 0;
}
```



```
#include &lt;stdio.h&gt;

typedef int ElementType;
#define NotFound (-1)

/* START: fig2_9.txt */
        int
        BinarySearch( const ElementType A[ ], ElementType X, int N )
        {
            int Low, Mid, High;

/* 1*/      Low = 0; High = N - 1;
/* 2*/      while( Low &lt;= High )
            {
/* 3*/          Mid = ( Low + High ) / 2;
/* 4*/          if( A[ Mid ] &lt; X )
/* 5*/              Low = Mid + 1;
                else
/* 6*/          if( A[ Mid ] &gt; X )
/* 7*/              High = Mid - 1;
                else
/* 8*/              return Mid;  /* Found */
            }
/* 9*/      return NotFound;     /* NotFound is defined as -1 */
        }
/* END */

main( )
{
    static int A[ ] = { 1, 3, 5, 7, 9, 13, 15 };
    const int SizeofA = sizeof( A ) / sizeof( A[ 0 ] );
    int i;

    for( i = 0; i &lt; 20; i++ )
        printf( "BinarySearch of %d returns %d\n",
                 i, BinarySearch( A, i, SizeofA ) );
    return 0;
}
```

```
#include &lt;stdio.h&gt;

/* START: fig2_10.txt */
        unsigned int
        Gcd( unsigned int M, unsigned int N )
        {
            unsigned int Rem;

/* 1*/      while( N &gt; 0 )
            {
/* 2*/          Rem = M % N;
/* 3*/          M = N;
/* 4*/          N = Rem;
            }
/* 5*/      return M;
        }
/* END */

main( )
{
    printf( "Gcd( 45, 35 ) = %d\n", Gcd( 45, 35 ) );
    printf( "Gcd( 1989, 1590 ) = %d\n", Gcd( 1989, 1590 ) );
    return 0;
}
```

```
#include &lt;stdio.h&gt;

#define IsEven( N ) ( ( N ) % 2 == 0 )

/* START: fig2_11.txt */
        long int
        Pow( long int X, unsigned int N )
        {
/* 1*/      if( N == 0 )
/* 2*/          return 1;
/* 3*/      if( N == 1 )
/* 4*/          return X;
/* 5*/      if( IsEven( N ) )
/* 6*/          return Pow( X * X, N / 2 );
            else
/* 7*/          return Pow( X * X, N / 2 ) * X;
        }
/* END */

main( )
{
    printf( "2^21 = %ld\n", Pow( 2, 21 ) );
    return 0;
}
```

```c
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;

#define Error( Str )        FatalError( Str )
#define FatalError( Str )   fprintf( stderr, "%s\n", Str ), exit( 1 )
```



```c
        #include "list.h"
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        /* Place in the interface file */
        struct Node
        {
            ElementType Element;
```

```
            Position    Next;
        };

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

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

        int
        IsEmpty( List L )
        {
            return L-&gt;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-&gt;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;

/* 1*/      P = L-&gt;Next;
/* 2*/      while( P != NULL &amp;&amp; P-&gt;Element != X )
/* 3*/          P = P-&gt;Next;

/* 4*/      return P;
```



```
        }
/* END */

/* START: fig3_11.txt */
        /* Delete from a list */
        /* Cell pointed to by P-&gt;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 */
```

```
            {                   /* X is found; delete it */
                TmpCell = P-&gt;Next;
                P-&gt;Next = TmpCell-&gt;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;

/* 1*/      P = L;
/* 2*/      while( P-&gt;Next != NULL &amp;&amp; P-&gt;Next-&gt;Element != X )
/* 3*/          P = P-&gt;Next;

/* 4*/      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;

/* 1*/      TmpCell = malloc( sizeof( struct Node ) );
/* 2*/      if( TmpCell == NULL )
/* 3*/          FatalError( "Out of space!!!" );
```



```
/* 4*/      TmpCell-&gt;Element = X;
/* 5*/      TmpCell-&gt;Next = P-&gt;Next;
/* 6*/      P-&gt;Next = TmpCell;
        }
/* END */

#if 0
/* START: fig3_14.txt */
        /* Incorrect DeleteList algorithm */

        void
        DeleteList( List L )
        {
            Position P;

/* 1*/      P = L-&gt;Next;  /* Header assumed */
/* 2*/      L-&gt;Next = NULL;
/* 3*/      while( P != NULL )
            {
/* 4*/          free( P );
/* 5*/          P = P-&gt;Next;
            }
        }
/* END */
#endif

/* START: fig3_15.txt */
```

```
        /* Correct DeleteList algorithm */

        void
        DeleteList( List L )
        {
            Position P, Tmp;

/* 1*/      P = L-&gt;Next;  /* Header assumed */
/* 2*/      L-&gt;Next = NULL;
/* 3*/      while( P != NULL )
            {
/* 4*/          Tmp = P-&gt;Next;
/* 5*/          free( P );
/* 6*/          P = Tmp;
            }
        }
/* END */

        Position
        Header( List L )
        {
            return L;
        }

        Position
        First( List L )
        {
            return L-&gt;Next;
        }
```



```
        Position
        Advance( Position P )
        {
            return P-&gt;Next;
        }

        ElementType
        Retrieve( Position P )
        {
            return P-&gt;Element;
        }
```

```
        typedef int ElementType;

/* START: fig3_6.txt */
        #ifndef _List_H
        #define _List_H

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

        List MakeEmpty( List L );
        int IsEmpty( List L );
        int IsLast( Position P, List L );
        Position Find( ElementType X, List L );
        void Delete( ElementType X, List L );
        Position FindPrevious( ElementType X, List L );
        void Insert( ElementType X, List L, Position P );
        void DeleteList( List L );
        Position Header( List L );
        Position First( List L );
        Position Advance( Position P );
        ElementType Retrieve( Position P );

        #endif    /* _List_H */
/* END */
```

```
#include &lt;stdio.h&gt;
#include "list.h"

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

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

    printf( "Finished deletions\n" );

    PrintList( L );

    DeleteList( L );

    return 0;
}
```

```
/* This code doesn't really do much */
/* Thus I haven't bothered testing it */

#include "fatal.h"

#define MaxDegree 100

static int
Max( int A, int B )
{
    return A &gt; B ? A : B;
}

/* START: fig3_18.txt */
        typedef struct
        {
            int CoeffArray[ MaxDegree + 1 ];
            int HighPower;
        } *Polynomial;
/* END */

/* START: fig3_19.txt */
        void
        ZeroPolynomial( Polynomial Poly )
        {
            int i;

            for( i = 0; i &lt;= MaxDegree; i++ )
                Poly-&gt;CoeffArray[ i ] = 0;
            Poly-&gt;HighPower = 0;
        }
/* END */

/* START: fig3_20.txt */
        void
        AddPolynomial( const Polynomial Poly1, const Polynomial Poly2,
                       Polynomial PolySum )
        {
            int i;

            ZeroPolynomial( PolySum );
            PolySum-&gt;HighPower = Max( Poly1-&gt;HighPower,
                                      Poly2-&gt;HighPower );

            for( i = PolySum-&gt;HighPower; i &gt;= 0; i-- )
                PolySum-&gt;CoeffArray[ i ] = Poly1-&gt;CoeffArray[ i ]
                                       + Poly2-&gt;CoeffArray[ i ];
        }
/* END */

/* START: fig3_21.txt */
        void
        MultPolynomial( const Polynomial Poly1,
                        const Polynomial Poly2, Polynomial PolyProd )
        {
            int i, j;
```

```
            ZeroPolynomial( PolyProd );
            PolyProd-&gt;HighPower = Poly1-&gt;HighPower + Poly2-&gt;HighPower;

            if( PolyProd-&gt;HighPower &gt; MaxDegree )
                Error( "Exceeded array size" );
            else
                for( i = 0; i &lt;= Poly1-&gt;HighPower; i++ )
                    for( j = 0; j &lt;= Poly2-&gt;HighPower; j++ )
                        PolyProd-&gt;CoeffArray[ i + j ] +=
                                Poly1-&gt;CoeffArray[ i ] *
                                Poly2-&gt;CoeffArray[ j ];
        }
/* END */

#if 0
/* START: fig3_23.txt */
        typedef struct Node *PtrToNode;

        struct Node
        {
            int Coefficient;
            int Exponent;
            PtrToNode Next;
        };

        typedef PtrToNode Polynomial;  /* Nodes sorted by exponent */
/* END */
#endif

void
PrintPoly( const Polynomial Q )
{
    int i;

    for( i = Q-&gt;HighPower; i &gt; 0; i-- )
        printf( "%dx^%d + ", Q-&gt;CoeffArray[ i ], i );
    printf( "%d\n", Q-&gt;CoeffArray[ 0 ] );
}

main( )
{
    Polynomial P, Q;

    P = malloc( sizeof( *P ) );
    Q = malloc( sizeof( *Q ) );

    P-&gt;HighPower = 1; P-&gt;CoeffArray[ 0 ] = 1; P-&gt;CoeffArray[ 1 ] = 1;
    MultPolynomial( P, P, Q );
    MultPolynomial( Q, Q, P );
    AddPolynomial( P, P, Q );
    PrintPoly( Q );

    return 0;
}
```

Aug 12 15:27 1996 cursor.c Page 1

```
        #include "cursor.h"
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        /* Place in the interface file */
        struct Node
        {
```

```
            ElementType Element;
            Position    Next;
        };

        struct Node CursorSpace[ SpaceSize ];

/* START: fig3_31.txt */
        static Position
        CursorAlloc( void )
        {
            Position P;

            P = CursorSpace[ 0 ].Next;
            CursorSpace[ 0 ].Next = CursorSpace[ P ].Next;

            return P;
        }

        static void
        CursorFree( Position P )
        {
            CursorSpace[ P ].Next = CursorSpace[ 0 ].Next;
            CursorSpace[ 0 ].Next = P;
        }
/* END */

        void
        InitializeCursorSpace( void )
        {
            int i;

            for( i = 0; i &lt; SpaceSize; i++ )
                CursorSpace[ i ].Next = i + 1;
            CursorSpace[ SpaceSize - 1 ].Next = 0;
        }

        List
        MakeEmpty( List L )
        {
            if( L != NULL )
                DeleteList( L );
            L = CursorAlloc( );
            if( L == 0 )
                FatalError( "Out of memory!" );
            CursorSpace[ L ].Next = 0;
            return L;
        }

/* START: fig3_32.txt */
```



```
        /* Return true if L is empty */

        int
        IsEmpty( List L )
        {
            return CursorSpace[ L ].Next == 0;
        }
/* END */

/* START: fig3_33.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 CursorSpace[ P ].Next == 0;
        }
```

```
        }
/* END */

/* START: fig3_34.txt */
        /* Return Position of X in L; 0 if not found */
        /* Uses a header node */

        Position
        Find( ElementType X, List L )
        {
            Position P;

/* 1*/      P = CursorSpace[ L ].Next;
/* 2*/      while( P &amp;&amp; CursorSpace[ P ].Element != X )
/* 3*/          P = CursorSpace[ P ].Next;

/* 4*/      return P;
        }
/* END */

/* START: fig3_35.txt */
        /* Delete from a list */
        /* 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 = CursorSpace[ P ].Next;
                CursorSpace[ P ].Next = CursorSpace[ TmpCell ].Next;
                CursorFree( TmpCell );
            }
        }
/* END */
```



```
        /* If X is not found, then Next field of returned value is 0 */
        /* Assumes a header */

        Position
        FindPrevious( ElementType X, List L )
        {
            Position P;

/* 1*/      P = L;
/* 2*/      while( CursorSpace[ P ].Next &amp;&amp;
                     CursorSpace[ CursorSpace[ P ].Next ].Element != X )
/* 3*/          P = CursorSpace[ P ].Next;

/* 4*/      return P;
        }

/* START: fig3_36.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;
```

```
            Position TmpCell;

/* 1*/      TmpCell = CursorAlloc( );
/* 2*/      if( TmpCell == 0 )
/* 3*/          FatalError( "Out of space!!!" );

/* 4*/      CursorSpace[ TmpCell ].Element = X;
/* 5*/      CursorSpace[ TmpCell ].Next = CursorSpace[ P ].Next;
/* 6*/      CursorSpace[ P ].Next = TmpCell;
        }
/* END */


        /* Correct DeleteList algorithm */

        void
        DeleteList( List L )
        {
            Position P, Tmp;

/* 1*/      P = CursorSpace[ L ].Next;  /* Header assumed */
/* 2*/      CursorSpace[ L ].Next = 0;
/* 3*/      while( P != 0 )
            {
/* 4*/          Tmp = CursorSpace[ P ].Next;
/* 5*/          CursorFree( P );
/* 6*/          P = Tmp;
            }
        }

        Position
```



```
        Header( List L )
        {
            return L;
        }

        Position
        First( List L )
        {
            return CursorSpace[ L ].Next;
        }

        Position
        Advance( Position P )
        {
            return CursorSpace[ P ].Next;
        }

        ElementType
        Retrieve( Position P )
        {
            return CursorSpace[ P ].Element;
        }
```

```
        typedef int ElementType;
        #define SpaceSize 100

/* START: fig3_28.txt */
        #ifndef _Cursor_H
        #define _Cursor_H

        typedef int PtrToNode;
        typedef PtrToNode List;
        typedef PtrToNode Position;

        void InitializeCursorSpace( void );

        List MakeEmpty( List L );
        int IsEmpty( const List L );
        int IsLast( const Position P, const List L );
        Position Find( ElementType X, const List L );
        void Delete( ElementType X, List L );
        Position FindPrevious( ElementType X, const List L );
        void Insert( ElementType X, List L, Position P );
        void DeleteList( List L );
        Position Header( const List L );
        Position First( const List L );
        Position Advance( const Position P );
        ElementType Retrieve( const Position P );

        #endif    /* _Cursor_H */

/* END */
```

```
#include &lt;stdio.h&gt;
#include "cursor.h"

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;

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

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

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

    printf( "Finished deletions\n" );

    PrintList( L );

    DeleteList( L );

    return 0;
}
```

```
        typedef int ElementType;
/* START: fig3_45.txt */
        #ifndef _Stack_h
        #define _Stack_h

        struct StackRecord;
        typedef struct StackRecord *Stack;

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

        #endif  /* _Stack_h */

/* END */
```

```
        #include "stackar.h"
        #include "fatal.h"
        #include &lt;stdlib.h&gt;

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

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

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

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

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

/* 1*/      if( MaxElements &lt; MinStackSize )
/* 2*/          Error( "Stack size is too small" );

/* 3*/      S = malloc( sizeof( struct StackRecord ) );
/* 4*/      if( S == NULL )
/* 5*/          FatalError( "Out of space!!!" );

/* 6*/      S-&gt;Array = malloc( sizeof( ElementType ) * MaxElements );
/* 7*/      if( S-&gt;Array == NULL )
/* 8*/          FatalError( "Out of space!!!" );
/* 9*/      S-&gt;Capacity = MaxElements;
/*10*/      MakeEmpty( S );

/*11*/      return S;
        }
/* END */

/* START: fig3_49.txt */
        void
        MakeEmpty( Stack S )
        {
            S-&gt;TopOfStack = EmptyTOS;
```



```
        }
/* END */

/* START: fig3_47.txt */
        void
        DisposeStack( Stack S )
```

```
        {
            if( S != NULL )
            {
                free( S-&gt;Array );
                free( S );
            }
        }
/* END */

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


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

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

/* START: fig3_53.txt */
        ElementType
        TopAndPop( Stack S )
        {
            if( !IsEmpty( S ) )
                return S-&gt;Array[ S-&gt;TopOfStack-- ];
            Error( "Empty stack" );
```



```
            return 0;  /* Return value used to avoid warning */
        }
/* END */
```

```
#include &lt;stdio.h&gt;
#include "stackar.h"

main( )
{
    Stack S;
    int i;

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

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

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

```
        typedef int ElementType;
/* START: fig3_39.txt */
        #ifndef _Stack_h
        #define _Stack_h

        struct Node;
        typedef struct Node *PtrToNode;
        typedef PtrToNode Stack;

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

        #endif  /* _Stack_h */

/* END */
```

```
        #include "stackli.h"
        #include "fatal.h"
        #include &lt;stdlib.h&gt;

        struct Node
        {
            ElementType Element;
            PtrToNode   Next;
        };

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

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

            S = malloc( sizeof( struct Node ) );
            if( S == NULL )
                FatalError( "Out of space!!!" );
            MakeEmpty( S );
            return S;
        }

        void
        MakeEmpty( Stack S )
        {
            if( S== NULL )
                Error( "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 )
                FatalError( "Out of space!!!" );
            else
            {
                TmpCell-&gt;Element = X;
                TmpCell-&gt;Next = S-&gt;Next;
                S-&gt;Next = TmpCell;
            }
        }
/* END */

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

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

            if( IsEmpty( S ) )
                Error( "Empty stack" );
            else
            {
                FirstCell = S-&gt;Next;
                S-&gt;Next = S-&gt;Next-&gt;Next;
                free( FirstCell );
            }
        }
/* END */
```

```
#include &lt;stdio.h&gt;
#include "stackli.h"

main( )
{
    Stack S;
    int i;

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

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

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

```
        typedef int ElementType;
/* START: fig3_57.txt */
        #ifndef _Queue_h
        #define _Queue_h

        struct QueueRecord;
        typedef struct QueueRecord *Queue;

        int IsEmpty( Queue Q );
        int IsFull( Queue Q );
        Queue CreateQueue( int MaxElements );
        void DisposeQueue( Queue Q );
        void MakeEmpty( Queue Q );
        void Enqueue( ElementType X, Queue Q );
        ElementType Front( Queue Q );
        void Dequeue( Queue Q );
        ElementType FrontAndDequeue( Queue Q );

        #endif  /* _Queue_h */
/* END */
```



```
        #include "queue.h"
        #include "fatal.h"
        #include &lt;stdlib.h&gt;

        #define MinQueueSize ( 5 )
```

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

/* START: fig3_58.txt */
        int
        IsEmpty( Queue Q )
        {
            return Q-&gt;Size == 0;
        }
/* END */

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

        Queue
        CreateQueue( int MaxElements )
        {
            Queue Q;

/* 1*/      if( MaxElements &lt; MinQueueSize )
/* 2*/          Error( "Queue size is too small" );

/* 3*/      Q = malloc( sizeof( struct QueueRecord ) );
/* 4*/      if( Q == NULL )
/* 5*/          FatalError( "Out of space!!!" );

/* 6*/      Q-&gt;Array = malloc( sizeof( ElementType ) * MaxElements );
/* 7*/      if( Q-&gt;Array == NULL )
/* 8*/          FatalError( "Out of space!!!" );
/* 9*/      Q-&gt;Capacity = MaxElements;
/*10*/      MakeEmpty( Q );

/*11*/      return Q;
        }

/* START: fig3_59.txt */
        void
        MakeEmpty( Queue Q )
        {
            Q-&gt;Size = 0;
            Q-&gt;Front = 1;
```



```
            Q-&gt;Rear = 0;
        }
/* END */

        void
        DisposeQueue( Queue Q )
        {
            if( Q != NULL )
            {
                free( Q-&gt;Array );
                free( Q );
            }
        }
```

```
/* START: fig3_60.txt */

        static int
        Succ( int Value, Queue Q )
        {
            if( ++Value == Q-&gt;Capacity )
                Value = 0;
            return Value;
        }

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



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

        void
        Dequeue( Queue Q )
        {
            if( IsEmpty( Q ) )
                Error( "Empty queue" );
            else
            {
```



```
                Q-&gt;Size--;
                Q-&gt;Front = Succ( Q-&gt;Front, Q );
            }
        }

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

            if( IsEmpty( Q ) )
                Error( "Empty queue" );
            else
            {
                Q-&gt;Size--;
                X = Q-&gt;Array[ Q-&gt;Front ];
                Q-&gt;Front = Succ( Q-&gt;Front, Q );
            }
            return X;
        }
```

```
#include &lt;stdio.h&gt;
#include "queue.h"

main( )
{
    Queue Q;
    int i;

    Q = CreateQueue( 12 );

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

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

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

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

```
        typedef int ElementType;

/* START: fig4_16.txt */
        #ifndef _Tree_H
        #define _Tree_H

        struct TreeNode;
        typedef struct TreeNode *Position;
        typedef struct TreeNode *SearchTree;

        SearchTree MakeEmpty( SearchTree T );
        Position Find( ElementType X, SearchTree T );
        Position FindMin( SearchTree T );
        Position FindMax( SearchTree T );
        SearchTree Insert( ElementType X, SearchTree T );
        SearchTree Delete( ElementType X, SearchTree T );
        ElementType Retrieve( Position P );

        #endif  /* _Tree_H */

/* END */
```

```
        #include "tree.h"
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

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

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

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

/* START: fig4_19.txt */
        Position
        FindMin( SearchTree T )
        {
            if( T == NULL )
                return NULL;
            else
            if( T-&gt;Left == NULL )
                return T;
            else
```

```
                return FindMin( T-&gt;Left );
        }
/* END */

/* START: fig4_20.txt */
```



```
        Position
        FindMax( SearchTree T )
        {
            if( T != NULL )
                while( T-&gt;Right != NULL )
                    T = T-&gt;Right;

            return T;
        }
/* END */

/* START: fig4_22.txt */
        SearchTree
        Insert( ElementType X, SearchTree T )
        {
/* 1*/      if( T == NULL )
            {
                /* Create and return a one-node tree */
/* 2*/          T = malloc( sizeof( struct TreeNode ) );
/* 3*/          if( T == NULL )
/* 4*/              FatalError( "Out of space!!!" );
                else
                {
/* 5*/              T-&gt;Element = X;
/* 6*/              T-&gt;Left = T-&gt;Right = NULL;
                }
            }
            else
/* 7*/      if( X &lt; T-&gt;Element )
/* 8*/          T-&gt;Left = Insert( X, T-&gt;Left );
            else
/* 9*/      if( X &gt; T-&gt;Element )
/*10*/          T-&gt;Right = Insert( X, T-&gt;Right );
            /* Else X is in the tree already; we'll do nothing */

/*11*/      return T;  /* Do not forget this line!! */
        }
/* END */

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

            if( T == NULL )
                Error( "Element not found" );
            else
            if( X &lt; T-&gt;Element )  /* Go left */
                T-&gt;Left = Delete( X, T-&gt;Left );
            else
            if( X &gt; T-&gt;Element )  /* Go right */
                T-&gt;Right = Delete( X, T-&gt;Right );
            else  /* Found element to be deleted */
            if( T-&gt;Left &amp;&amp; T-&gt;Right )  /* Two children */
            {
```

```
                /* Replace with smallest in right subtree */
                TmpCell = FindMin( T-&gt;Right );
                T-&gt;Element = TmpCell-&gt;Element;
                T-&gt;Right = Delete( T-&gt;Element, T-&gt;Right );
            }
            else  /* One or zero children */
            {
                TmpCell = T;
                if( T-&gt;Left == NULL ) /* Also handles 0 children */
                    T = T-&gt;Right;
                else if( T-&gt;Right == NULL )
                    T = T-&gt;Left;
                free( TmpCell );
            }

            return T;
        }
/* END */

        ElementType
        Retrieve( Position P )
        {
            return P-&gt;Element;
        }
```



```
#include "tree.h"
#include &lt;stdio.h&gt;

main( )
```

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

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

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

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



```
        typedef int ElementType;

/* START: fig4_35.txt */
        #ifndef _AvlTree_H
        #define _AvlTree_H

        struct AvlNode;
        typedef struct AvlNode *Position;
        typedef struct AvlNode *AvlTree;

        AvlTree MakeEmpty( AvlTree T );
        Position Find( ElementType X, AvlTree T );
        Position FindMin( AvlTree T );
```

```
        Position FindMax( AvlTree T );
        AvlTree Insert( ElementType X, AvlTree T );
        AvlTree Delete( ElementType X, AvlTree T );
        ElementType Retrieve( Position P );

        #endif  /* _AvlTree_H */
/* END */
```

Aug 12 15:27 1996 avltree.c Page 1

```
        #include "avltree.h"
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

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

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

```
        }

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

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

        Position
        FindMax( AvlTree T )
        {
            if( T != NULL )
                while( T-&gt;Right != NULL )
                    T = T-&gt;Right;
```



```
            return T;
        }

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

        static int
        Max( int Lhs, int Rhs )
        {
            return Lhs &gt; 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-&gt;Left;
```

```
            K1 = K2-&gt;Left;
            K2-&gt;Left = K1-&gt;Right;
            K1-&gt;Right = K2;

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

            K1-&gt;Height = Max( Height( K1-&gt;Left ), Height( K1-&gt;Right ) ) + 1;
```



```
            K2-&gt;Height = Max( Height( K2-&gt;Right ), K1-&gt;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-&gt;Left = SingleRotateWithRight( K3-&gt;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-&gt;Right = SingleRotateWithLeft( K1-&gt;Right );

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


/* START: fig4_37.txt */
        AvlTree
        Insert( ElementType X, AvlTree T )
```

```
        Insert( ElementType X, AvlTree T )
        {
            if( T == NULL )
            {
                /* Create and return a one-node tree */
                T = malloc( sizeof( struct AvlNode ) );
                if( T == NULL )
                    FatalError( "Out of space!!!" );
                else
                {
                    T-&gt;Element = X; T-&gt;Height = 0;
                    T-&gt;Left = T-&gt;Right = NULL;
                }
            }
            else
            if( X &lt; T-&gt;Element )
```



```
            {
                T-&gt;Left = Insert( X, T-&gt;Left );
                if( Height( T-&gt;Left ) - Height( T-&gt;Right ) == 2 )
                    if( X &lt; T-&gt;Left-&gt;Element )
                        T = SingleRotateWithLeft( T );
                    else
                        T = DoubleRotateWithLeft( T );
            }
            else
            if( X &gt; T-&gt;Element )
            {
                T-&gt;Right = Insert( X, T-&gt;Right );
                if( Height( T-&gt;Right ) - Height( T-&gt;Left ) == 2 )
                    if( X &gt; T-&gt;Right-&gt;Element )
                        T = SingleRotateWithRight( T );
                    else
                        T = DoubleRotateWithRight( T );
            }
            /* Else X is in the tree already; we'll do nothing */

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

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

        ElementType
        Retrieve( Position P )
        {
            return P-&gt;Element;
        }
```

```
#include "avltree.h"
#include &lt;stdio.h&gt;

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

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

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

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

```
/* Here are some of the hash functions */
/* for strings that are in the text */

typedef unsigned int Index;

/* START: fig5_3.txt */
        Index
        Hash1( const char *Key, int TableSize )
        {
            unsigned int HashVal = 0;

/* 1*/      while( *Key != '\0' )
/* 2*/          HashVal += *Key++;

/* 3*/      return HashVal % TableSize;
        }
/* END */

/* START: fig5_4.txt */
        Index
        Hash2( const char *Key, int TableSize )
        {
            return ( Key[ 0 ] + 27 * Key[ 1 ] + 729 * Key[ 2 ] )
                        % TableSize;
        }
/* END */

/* START: fig5_5.txt */
        Index
        Hash3( const char *Key, int TableSize )
        {
            unsigned int HashVal = 0;

/* 1*/      while( *Key != '\0' )
/* 2*/          HashVal = ( HashVal &lt;&lt; 5 ) + *Key++;

/* 3*/      return HashVal % TableSize;
        }
/* END */
```



```
/* Interface for separate chaining hash table */
typedef int ElementType;
```

```
/* START: fig5_2.txt */
        typedef unsigned int Index;
/* END */

/* START: fig5_7.txt */
        #ifndef _HashSep_H
        #define _HashSep_H

        struct ListNode;
        typedef struct ListNode *Position;
        struct HashTbl;
        typedef struct HashTbl *HashTable;

        HashTable InitializeTable( int TableSize );
        void DestroyTable( HashTable H );
        Position Find( ElementType Key, HashTable H );
        void Insert( ElementType Key, HashTable H );
        ElementType Retrieve( Position P );
        /* Routines such as Delete are MakeEmpty are omitted */

        #endif  /* _HashSep_H */
/* END */
```

Aug 12 15:27 1996 hashsep.c Page 1

```
       #include "fatal.h"
       #include "hashsep.h"
       #include &lt;stdlib.h&gt;

       #define MinTableSize (10)

        struct ListNode
        {
            ElementType Element;
            Position    Next;
        };
```

```
        typedef Position List;

        /* List *TheList will be an array of lists, allocated later */
        /* The lists use headers (for simplicity), */
        /* though this wastes space */
        struct HashTbl
        {
            int TableSize;
            List *TheLists;
        };

        /* Return next prime; assume N &gt;= 10 */
        static int
        NextPrime( int N )
        {
            int i;

            if( N % 2 == 0 )
                N++;
            for( ; ; N += 2 )
            {
                for( i = 3; i * i &lt;= N; i += 2 )
                    if( N % i == 0 )
                        goto ContOuter;  /* Sorry about this! */
                return N;
              ContOuter: ;
            }
        }

        /* Hash function for ints */
        Index
        Hash( ElementType Key, int TableSize )
        {
            return Key % TableSize;
        }

/* START: fig5_8.txt */
        HashTable
        InitializeTable( int TableSize )
        {
            HashTable H;
            int i;

/* 1*/      if( TableSize &lt; MinTableSize )
```



```
            {
/* 2*/          Error( "Table size too small" );
/* 3*/          return NULL;
            }

            /* Allocate table */
/* 4*/      H = malloc( sizeof( struct HashTbl ) );
/* 5*/      if( H == NULL )
/* 6*/          FatalError( "Out of space!!!" );

/* 7*/      H-&gt;TableSize = NextPrime( TableSize );

            /* Allocate array of lists */
/* 8*/      H-&gt;TheLists = malloc( sizeof( List ) * H-&gt;TableSize );
/* 9*/      if( H-&gt;TheLists == NULL )
/*10*/          FatalError( "Out of space!!!" );

            /* Allocate list headers */
/*11*/      for( i = 0; i &lt; H-&gt;TableSize; i++ )
            {
/*12*/          H-&gt;TheLists[ i ] = malloc( sizeof( struct ListNode ) );
```

```
/*13*/            if( H-&gt;TheLists[ i ] == NULL )
/*14*/                FatalError( "Out of space!!!" );
                  else
/*15*/                H-&gt;TheLists[ i ]-&gt;Next = NULL;
              }

/*16*/        return H;
          }
/* END */

/* START: fig5_9.txt */
          Position
          Find( ElementType Key, HashTable H )
          {
              Position P;
              List L;

/* 1*/        L = H-&gt;TheLists[ Hash( Key, H-&gt;TableSize ) ];
/* 2*/        P = L-&gt;Next;
/* 3*/        while( P != NULL &amp;&amp; P-&gt;Element != Key )
                                  /* Probably need strcmp!! */
/* 4*/            P = P-&gt;Next;
/* 5*/        return P;
          }
/* END */

/* START: fig5_10.txt */
          void
          Insert( ElementType Key, HashTable H )
          {
              Position Pos, NewCell;
              List L;

/* 1*/        Pos = Find( Key, H );
/* 2*/        if( Pos == NULL )   /* Key is not found */
```



```
              {
/* 3*/            NewCell = malloc( sizeof( struct ListNode ) );
/* 4*/            if( NewCell == NULL )
/* 5*/                FatalError( "Out of space!!!" );
                  else
                  {
/* 6*/                L = H-&gt;TheLists[ Hash( Key, H-&gt;TableSize ) ];
/* 7*/                NewCell-&gt;Next = L-&gt;Next;
/* 8*/                NewCell-&gt;Element = Key;  /* Probably need strcpy! */
/* 9*/                L-&gt;Next = NewCell;
                  }
              }
          }
/* END */

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

          void
          DestroyTable( HashTable H )
          {
              int i;

              for( i = 0; i &lt; H-&gt;TableSize; i++ )
              {
                  Position P = H-&gt;TheLists[ i ];
                  Position Tmp;
```

```
                while( P != NULL )
                {
                    Tmp = P-&gt;Next;
                    free( P );
                    P = Tmp;
                }
            }

            free( H-&gt;TheLists );
            free( H );
        }
```

Nov 4 20:35 1997 hashquad.h Page 1

```
/* Interface for quadratic probing hash table */
typedef int ElementType;

/* START: fig5_14.txt */
        #ifndef _HashQuad_H
        #define _HashQuad_H

        typedef unsigned int Index;
        typedef Index Position;

        struct HashTbl;
        typedef struct HashTbl *HashTable;

        HashTable InitializeTable( int TableSize );
        void DestroyTable( HashTable H );
        Position Find( ElementType Key, HashTable H );
        void Insert( ElementType Key, HashTable H );
        ElementType Retrieve( Position P, HashTable H );
        HashTable Rehash( HashTable H );
        /* Routines such as Delete are MakeEmpty are omitted */

        #endif  /* _HashQuad_H */

/* END */
```

```
#include "fatal.h"
#include "hashquad.h"
#include &lt;stdlib.h&gt;

#define MinTableSize (10)

enum KindOfEntry { Legitimate, Empty, Deleted };

struct HashEntry
{
    ElementType      Element;
    enum KindOfEntry Info;
};

typedef struct HashEntry Cell;

/* Cell *TheCells will be an array of */
/* HashEntry cells, allocated later */
struct HashTbl
{
    int TableSize;
    Cell *TheCells;
};

/* Return next prime; assume N &gt;= 10 */

static int
NextPrime( int N )
{
    int i;

    if( N % 2 == 0 )
        N++;
    for( ; ; N += 2 )
    {
        for( i = 3; i * i &lt;= N; i += 2 )
            if( N % i == 0 )
                goto ContOuter;  /* Sorry about this! */
        return N;
      ContOuter: ;
    }
}

/* Hash function for ints */
Index
Hash( ElementType Key, int TableSize )
{
    return Key % TableSize;
}
```

```
/* START: fig5_15.txt */
        HashTable
        InitializeTable( int TableSize )
        {
            HashTable H;
            int i;
```



```
/* 1*/      if( TableSize &lt; MinTableSize )
            {
/* 2*/          Error( "Table size too small" );
/* 3*/          return NULL;
            }

            /* Allocate table */
/* 4*/      H = malloc( sizeof( struct HashTbl ) );
/* 5*/      if( H == NULL )
/* 6*/          FatalError( "Out of space!!!" );

/* 7*/      H-&gt;TableSize = NextPrime( TableSize );

            /* Allocate array of Cells */
/* 8*/      H-&gt;TheCells = malloc( sizeof( Cell ) * H-&gt;TableSize );
/* 9*/      if( H-&gt;TheCells == NULL )
/*10*/          FatalError( "Out of space!!!" );

/*11*/      for( i = 0; i &lt; H-&gt;TableSize; i++ )
/*12*/          H-&gt;TheCells[ i ].Info = Empty;

/*13*/      return H;
        }
/* END */

/* START: fig5_16.txt */
        Position
        Find( ElementType Key, HashTable H )
        {
            Position CurrentPos;
            int CollisionNum;

/* 1*/      CollisionNum = 0;
/* 2*/      CurrentPos = Hash( Key, H-&gt;TableSize );
/* 3*/      while( H-&gt;TheCells[ CurrentPos ].Info != Empty &amp;&amp;
                   H-&gt;TheCells[ CurrentPos ].Element != Key )
                            /* Probably need strcmp!! */
            {
/* 4*/          CurrentPos += 2 * ++CollisionNum - 1;
/* 5*/          if( CurrentPos &gt;= H-&gt;TableSize )
/* 6*/              CurrentPos -= H-&gt;TableSize;
            }
/* 7*/      return CurrentPos;
        }
/* END */

/* START: fig5_17.txt */
        void
        Insert( ElementType Key, HashTable H )
        {
            Position Pos;

            Pos = Find( Key, H );
            if( H-&gt;TheCells[ Pos ].Info != Legitimate )
            {
```

```
                /* OK to insert here */
                H-&gt;TheCells[ Pos ].Info = Legitimate;
                H-&gt;TheCells[ Pos ].Element = Key;
                         /* Probably need strcpy! */
            }
        }
/* END */

/* START: fig5_22.txt */
        HashTable
        Rehash( HashTable H )
        {
            int i, OldSize;
            Cell *OldCells;

/* 1*/      OldCells = H-&gt;TheCells;
/* 2*/      OldSize  = H-&gt;TableSize;

            /* Get a new, empty table */
/* 3*/      H = InitializeTable( 2 * OldSize );

            /* Scan through old table, reinserting into new */
/* 4*/      for( i = 0; i &lt; OldSize; i++ )
/* 5*/          if( OldCells[ i ].Info == Legitimate )
/* 6*/              Insert( OldCells[ i ].Element, H );

/* 7*/      free( OldCells );

/* 8*/      return H;
        }
/* END */



        ElementType
        Retrieve( Position P, HashTable H )
        {
            return H-&gt;TheCells[ P ].Element;
        }

        void
        DestroyTable( HashTable H )
        {
            free( H-&gt;TheCells );
            free( H );
        }
```



```
#define SepChain    /* Define the appropriate hash algorithm */
```

```
#ifdef SepChain
    #include "hashsep.h"
#endif

#ifdef QuadProb
    #include "hashquad.h"
#endif


#include &lt;stdio.h&gt;

#define NumItems 400

main( )
{
    HashTable H;
    Position P;
    int i;
    int j = 0;
    int CurrentSize;

    H = InitializeTable( CurrentSize = 13 );

    for( i = 0; i &lt; NumItems; i++, j += 71 )
    {
    #ifdef QuadProb
        if( i &gt; CurrentSize / 2 )
        {
            H = Rehash( H );
            printf( "Rehashing...\n" );
            CurrentSize *= 2;
        }
    #endif
        Insert( j, H );
    }

    for( i = 0, j = 0; i &lt; NumItems; i++, j += 71 )
    #ifdef SepChain
        if( ( P = Find( j, H ) ) == NULL || Retrieve( P ) != j )
    #endif
    #ifdef QuadProb
        if( Retrieve( ( P = Find( j, H ) ), H ) != j )
    #endif
            printf( "Error at %d\n", j );

    printf( "End of program.\n" );
    return 0;
}
```



```
        typedef int ElementType;

/* START: fig6_4.txt */
        #ifndef _BinHeap_H
        #define _BinHeap_H

        struct HeapStruct;
        typedef struct HeapStruct *PriorityQueue;

        PriorityQueue Initialize( int MaxElements );
```

```
        PriorityQueue Initialize( int MaxElements );
        void Destroy( PriorityQueue H );
        void MakeEmpty( PriorityQueue H );
        void Insert( ElementType X, PriorityQueue H );
        ElementType DeleteMin( PriorityQueue H );
        ElementType FindMin( PriorityQueue H );
        int IsEmpty( PriorityQueue H );
        int IsFull( PriorityQueue H );

        #endif

/* END */
```



```
        #include "binheap.h"
        #include "fatal.h"
        #include &lt;stdlib.h&gt;

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

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

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

```
/* 1*/      if( MaxElements &lt; MinPQSize )
/* 2*/          Error( "Priority queue size is too small" );

/* 3*/      H = malloc( sizeof( struct HeapStruct ) );
/* 4*/      if( H ==NULL )
/* 5*/          FatalError( "Out of space!!!" );

            /* Allocate the array plus one extra for sentinel */
/* 6*/      H-&gt;Elements = malloc( ( MaxElements + 1 )
                                    * sizeof( ElementType ) );
/* 7*/      if( H-&gt;Elements == NULL )
/* 8*/          FatalError( "Out of space!!!" );

/* 9*/      H-&gt;Capacity = MaxElements;
/*10*/      H-&gt;Size = 0;
/*11*/      H-&gt;Elements[ 0 ] = MinData;

/*12*/      return H;
        }
/* END */

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

/* START: fig6_8.txt */
        /* H-&gt;Element[ 0 ] is a sentinel */

        void
        Insert( ElementType X, PriorityQueue H )
        {
            int i;

            if( IsFull( H ) )
```



```
            {
                Error( "Priority queue is full" );
                return;
            }

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

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

/* 1*/      if( IsEmpty( H ) )
            {
/* 2*/          Error( "Priority queue is empty" );
/* 3*/          return H-&gt;Elements[ 0 ];
            }
/* 4*/      MinElement = H-&gt;Elements[ 1 ];
/* 5*/      LastElement = H-&gt;Elements[ H-&gt;Size-- ];

/* 6*/      for( i = 1; i * 2 &lt;= H-&gt;Size; i = Child )
            {
                /* Find smaller child */
```

```
/* 7*/          Child = i * 2;
/* 8*/          if( Child != H-&gt;Size &amp;&amp; H-&gt;Elements[ Child + 1 ]
/* 9*/                                &lt; H-&gt;Elements[ Child ] )
/*10*/              Child++;

                /* Percolate one level */
/*11*/          if( LastElement &gt; H-&gt;Elements[ Child ] )
/*12*/              H-&gt;Elements[ i ] = H-&gt;Elements[ Child ];
                else
/*13*/              break;
            }
/*14*/      H-&gt;Elements[ i ] = LastElement;
/*15*/      return MinElement;
        }
/* END */

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

        int
        IsEmpty( PriorityQueue H )
```



```
        {
            return H-&gt;Size == 0;
        }

        int
        IsFull( PriorityQueue H )
        {
            return H-&gt;Size == H-&gt;Capacity;
        }

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

        #if 0
/* START: fig6_14.txt */
        for( i = N / 2; i &gt; 0; i-- )
            PercolateDown( i );
/* END */
        #endif
```

```
#include "binheap.h"
#include &lt;stdio.h&gt;

#define MaxSize (1000)

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

    H = Initialize( MaxSize );
    for( i=0, j=MaxSize/2; i&lt;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;
}
```

```
        typedef int ElementType;

/* START: fig6_25.txt */
        #ifndef _LeftHeap_H
        #define _LeftHeap_H

        struct TreeNode;
        typedef struct TreeNode *PriorityQueue;

        /* Minimal set of priority queue operations */
        /* Note that nodes will be shared among several */
        /* leftist heaps after a merge; the user must */
        /* make sure to not use the old leftist heaps */

        PriorityQueue Initialize( void );
        ElementType FindMin( PriorityQueue H );
        int IsEmpty( PriorityQueue H );
        PriorityQueue Merge( PriorityQueue H1, PriorityQueue H2 );

        #define Insert( X, H ) ( H = Insert1( ( X ), H ) )
        /* DeleteMin macro is left as an exercise */

        PriorityQueue Insert1( ElementType X, PriorityQueue H );
        PriorityQueue DeleteMin1( PriorityQueue H );

        #endif

/* END */
```

```
        #include "leftheap.h"
        #include "fatal.h"
        #include &lt;stdlib.h&gt;


        struct TreeNode
        {
            ElementType   Element;
            PriorityQueue Left;
            PriorityQueue Right;
            int           Npl;
        };

        PriorityQueue
        Initialize( void )
        {
            return NULL;
        }
        static PriorityQueue Merge1( PriorityQueue H1, PriorityQueue H2 );

/* START: fig6_26.txt */
        PriorityQueue
        Merge( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1 == NULL )
/* 2*/          return H2;
/* 3*/      if( H2 == NULL )
/* 4*/          return H1;
/* 5*/      if( H1-&gt;Element &lt; H2-&gt;Element )
/* 6*/          return Merge1( H1, H2 );
            else
/* 7*/          return Merge1( H2, H1 );
        }
/* END */

        void
        SwapChildren( PriorityQueue H )
        {
            PriorityQueue Tmp;

            Tmp = H-&gt;Left;
            H-&gt;Left = H-&gt;Right;
            H-&gt;Right = Tmp;
        }

/* START: fig6_27.txt */
        static PriorityQueue
        Merge1( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1-&gt;Left == NULL )  /* Single node */
/* 2*/          H1-&gt;Left = H2;      /* H1-&gt;Right is already NULL,
                                       H1-&gt;Npl is already 0 */
            else
            {
/* 3*/          H1-&gt;Right = Merge( H1-&gt;Right, H2 );
/* 4*/          if( H1-&gt;Left-&gt;Npl &lt; H1-&gt;Right-&gt;Npl )
```

```
/* 5*/                SwapChildren( H1 );

/* 6*/            H1-&gt;Npl = H1-&gt;Right-&gt;Npl + 1;
            }
/* 7*/        return H1;
        }
/* END */

/* START: fig6_29.txt */

        PriorityQueue
        Insert1( ElementType X, PriorityQueue H )
        {
            PriorityQueue SingleNode;

/* 1*/        SingleNode = malloc( sizeof( struct TreeNode ) );
/* 2*/        if( SingleNode == NULL )
/* 3*/            FatalError( "Out of space!!!" );
            else
            {
/* 4*/            SingleNode-&gt;Element = X; SingleNode-&gt;Npl = 0;
/* 5*/            SingleNode-&gt;Left = SingleNode-&gt;Right = NULL;
/* 6*/            H = Merge( SingleNode, H );
            }
/* 7*/        return H;
        }
/* END */


/* START: fig6_30.txt */
        /* DeleteMin1 returns the new tree; */
        /* To get the minimum, use FindMin */
        /* This is for convenience */

        PriorityQueue
        DeleteMin1( PriorityQueue H )
        {
            PriorityQueue LeftHeap, RightHeap;

/* 1*/        if( IsEmpty( H ) )
            {
/* 2*/            Error( "Priority queue is empty" );
/* 3*/            return H;
            }

/* 4*/        LeftHeap = H-&gt;Left;
/* 5*/        RightHeap = H-&gt;Right;
/* 6*/        free( H );
/* 7*/        return Merge( LeftHeap, RightHeap );
        }
/* END */

        ElementType
        FindMin( PriorityQueue H )
        {
            if( !IsEmpty( H ) )
```



```
                return H-&gt;Element;
            Error( "Priority Queue is Empty" );
            return  0;
        }

        int
        IsEmpty( PriorityQueue H )
        {
            return H == NULL;
        }
```

```
        }
```



```
#include "leftheap.h"
#include &lt;stdio.h&gt;

#define MaxSize 5000

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

    H = Initialize( );
    for( i=0, j=MaxSize/2; i&lt;MaxSize; i++, j=( j+17)%MaxSize )
        Insert( j, H );

    j = 0;
    while( !IsEmpty( H ) )
        if( FindMin( H ) != j++ )
            printf( "Error in DeleteMin, %d\n", j );
        else
```

```
            H = DeleteMin1( H );
    printf( "Done...\n" );
    return 0;
}
```

Nov 4 20:34 1997 binomial.h Page 1

```
        typedef long ElementType;
        #define Infinity (30000L)

        #ifndef _BinHeap_H
        #define _BinHeap_H

        #define MaxTrees (14)   /* Stores 2^14 -1 items */
        #define Capacity (16383)

        struct BinNode;
        typedef struct BinNode *BinTree;
        struct Collection;
        typedef struct Collection *BinQueue;

        BinQueue Initialize( void );
        void Destroy( BinQueue H );
        BinQueue MakeEmpty( BinQueue H );
        BinQueue Insert( ElementType Item, BinQueue H );
        ElementType DeleteMin( BinQueue H );
        BinQueue Merge( BinQueue H1, BinQueue H2 );
        ElementType FindMin( BinQueue H );
        int IsEmpty( BinQueue H );
        int IsFull( BinQueue H );
        #endif
/* END */
```

```
        #include "binomial.h"
        #include "fatal.h"


/* START: fig6_52.txt */
        typedef struct BinNode *Position;

        struct BinNode
        {
            ElementType Element;
            Position    LeftChild;
            Position    NextSibling;
        };

        struct Collection
        {
            int CurrentSize;
            BinTree TheTrees[ MaxTrees ];
        };

        BinQueue
        Initialize( void )
        {
            BinQueue H;
            int i;

            H = malloc( sizeof( struct Collection ) );
            if( H == NULL )
                FatalError( "Out of space!!!" );
            H-&gt;CurrentSize = 0;
            for( i = 0; i &lt; MaxTrees; i++ )
                H-&gt;TheTrees[ i ] = NULL;
            return H;
        }

        static void
        DestroyTree( BinTree T )
        {
```

```
        {
            if( T != NULL )
            {
                DestroyTree( T-&gt;LeftChild );
                DestroyTree( T-&gt;NextSibling );
                free( T );
            }
        }

        void
        Destroy( BinQueue H )
        {
            int i;

            for( i = 0; i &lt; MaxTrees; i++ )
                DestroyTree( H-&gt;TheTrees[ i ] );
        }

        BinQueue
```



```
        MakeEmpty( BinQueue H )
        {
            int i;

            Destroy( H );
            for( i = 0; i &lt; MaxTrees; i++ )
                H-&gt;TheTrees[ i ] = NULL;
            H-&gt;CurrentSize = 0;

            return H;
        }

        /* Not optimized for O(1) amortized performance */
        BinQueue
        Insert( ElementType Item, BinQueue H )
        {
            BinTree NewNode;
            BinQueue OneItem;

            NewNode = malloc( sizeof( struct BinNode ) );
            if( NewNode == NULL )
                FatalError( "Out of space!!!" );
            NewNode-&gt;LeftChild = NewNode-&gt;NextSibling = NULL;
            NewNode-&gt;Element = Item;

            OneItem = Initialize( );
            OneItem-&gt;CurrentSize = 1;
            OneItem-&gt;TheTrees[ 0 ] = NewNode;

            return Merge( H, OneItem );
        }

/* START: fig6_56.txt */
        ElementType
        DeleteMin( BinQueue H )
        {
            int i, j;
            int MinTree;   /* The tree with the minimum item */
            BinQueue DeletedQueue;
            Position DeletedTree, OldRoot;
            ElementType MinItem;

            if( IsEmpty( H ) )
            {
                Error( "Empty binomial queue" );
                return -Infinity;
            }
```

```
            }

            MinItem = Infinity;
            for( i = 0; i &lt; MaxTrees; i++ )
            {
                if( H-&gt;TheTrees[ i ] &amp;&amp;
                    H-&gt;TheTrees[ i ]-&gt;Element &lt; MinItem )
                {
                    /* Update minimum */
                    MinItem = H-&gt;TheTrees[ i ]-&gt;Element;
```



```
                    MinTree = i;
                }
            }

            DeletedTree = H-&gt;TheTrees[ MinTree ];
            OldRoot = DeletedTree;
            DeletedTree = DeletedTree-&gt;LeftChild;
            free( OldRoot );

            DeletedQueue = Initialize( );
            DeletedQueue-&gt;CurrentSize = ( 1 &lt;&lt; MinTree ) - 1;
            for( j = MinTree - 1; j &gt;= 0; j-- )
            {
                DeletedQueue-&gt;TheTrees[ j ] = DeletedTree;
                DeletedTree = DeletedTree-&gt;NextSibling;
                DeletedQueue-&gt;TheTrees[ j ]-&gt;NextSibling = NULL;
            }

            H-&gt;TheTrees[ MinTree ] = NULL;
            H-&gt;CurrentSize -= DeletedQueue-&gt;CurrentSize + 1;

            Merge( H, DeletedQueue );
            return MinItem;
        }
/* END */

        ElementType
        FindMin( BinQueue H )
        {
            int i;
            ElementType MinItem;

            if( IsEmpty( H ) )
            {
                Error( "Empty binomial queue" );
                return 0;
            }

            MinItem = Infinity;
            for( i = 0; i &lt; MaxTrees; i++ )
            {
                if( H-&gt;TheTrees[ i ] &amp;&amp;
                            H-&gt;TheTrees[ i ]-&gt;Element &lt; MinItem )
                    MinItem = H-&gt;TheTrees[ i ]-&gt;Element;
            }

            return MinItem;
        }

        int
        IsEmpty( BinQueue H )
        {
            return H-&gt;CurrentSize == 0;
        }

        int IsFull( BinQueue H )
```

```
        {
            return H-&gt;CurrentSize == Capacity;
        }

/* START: fig6_54.txt */
        /* Return the result of merging equal-sized T1 and T2 */
        BinTree
        CombineTrees( BinTree T1, BinTree T2 )
        {
            if( T1-&gt;Element &gt; T2-&gt;Element )
                return CombineTrees( T2, T1 );
            T2-&gt;NextSibling = T1-&gt;LeftChild;
            T1-&gt;LeftChild = T2;
            return T1;
        }
/* END */

/* START: fig6_55.txt */
        /* Merge two binomial queues */
        /* Not optimized for early termination */
        /* H1 contains merged result */

        BinQueue
        Merge( BinQueue H1, BinQueue H2 )
        {
            BinTree T1, T2, Carry = NULL;
            int i, j;

            if( H1-&gt;CurrentSize + H2-&gt;CurrentSize &gt; Capacity )
                Error( "Merge would exceed capacity" );

            H1-&gt;CurrentSize += H2-&gt;CurrentSize;
            for( i = 0, j = 1; j &lt;= H1-&gt;CurrentSize; i++, j *= 2 )
            {
                T1 = H1-&gt;TheTrees[ i ]; T2 = H2-&gt;TheTrees[ i ];

                switch( !!T1 + 2 * !!T2 + 4 * !!Carry )
                {
                    case 0: /* No trees */
                    case 1: /* Only H1 */
                        break;
                    case 2: /* Only H2 */
                        H1-&gt;TheTrees[ i ] = T2;
                        H2-&gt;TheTrees[ i ] = NULL;
                        break;
                    case 4: /* Only Carry */
                        H1-&gt;TheTrees[ i ] = Carry;
                        Carry = NULL;
                        break;
                    case 3: /* H1 and H2 */
                        Carry = CombineTrees( T1, T2 );
                        H1-&gt;TheTrees[ i ] = H2-&gt;TheTrees[ i ] = NULL;
                        break;
                    case 5: /* H1 and Carry */
                        Carry = CombineTrees( T1, Carry );
                        H1-&gt;TheTrees[ i ] = NULL;
```

```
                        break;
                    case 6: /* H2 and Carry */
                        Carry = CombineTrees( T2, Carry );
                        H2-&gt;TheTrees[ i ] = NULL;
                        break;
                    case 7: /* All three */
                        H1-&gt;TheTrees[ i ] = Carry;
                        Carry = CombineTrees( T1, T2 );
                        H2-&gt;TheTrees[ i ] = NULL;
                        break;
                }
            }
            return H1;
        }
/* END */
```



```
#include "binomial.h"
#include &lt;stdio.h&gt;

#define MaxSize (12000)

main( )
{
    BinQueue H;
    int i, j;
```

```
    int i, j;
    ElementType AnItem;

    H = Initialize( );
    for( i=0, j=MaxSize/2; i&lt;MaxSize; i++, j=( j+71)%MaxSize )
    {
 /*     printf( "Inserting %d\n", j );
 */     H = Insert( j, H );
    }
 #if 1
    j = 0;
    while( !IsEmpty( H ) )
    {
 /*     printf( "DeleteMin\n" );
        H = DeleteMin( &amp;AnItem, H );
  */    if( DeleteMin( H ) != j++ )
            printf( "Error in DeleteMin, %d\n", j );
    }
    if( j != MaxSize )
        printf( "Error in counting\n" );
    #endif
    printf( "Done...\n" );
    return 0;
}
```



```
        /* This file contains a collection of sorting routines */

        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        typedef int ElementType;

        void
        Swap( ElementType *Lhs, ElementType *Rhs )
        {
            ElementType Tmp = *Lhs;
            *Lhs = *Rhs;
            *Rhs = Tmp;
        }

/* START: fig7_2.txt */
        void
        InsertionSort( ElementType A[ ], int N )
```

```
        InsertionSort( ElementType A[ ], int N )
        {
            int j, P;
            ElementType Tmp;

/* 1*/      for( P = 1; P &lt; N; P++ )
            {
/* 2*/          Tmp = A[ P ];
/* 3*/          for( j = P; j &gt; 0 &amp;&amp; A[ j - 1 ] &gt; Tmp; j-- )
/* 4*/              A[ j ] = A[ j - 1 ];
/* 5*/          A[ j ] = Tmp;
            }
        }
/* END */

/* START: fig7_4.txt */
        void
        Shellsort( ElementType A[ ], int N )
        {
            int i, j, Increment;
            ElementType Tmp;

/* 1*/      for( Increment = N / 2; Increment &gt; 0; Increment /= 2 )
/* 2*/          for( i = Increment; i &lt; N; i++ )
                {
/* 3*/              Tmp = A[ i ];
/* 4*/              for( j = i; j &gt;= Increment; j -= Increment )
/* 5*/                  if( Tmp &lt; A[ j - Increment ] )
/* 6*/                      A[ j ] = A[ j - Increment ];
                        else
/* 7*/                      break;
/* 8*/              A[ j ] = Tmp;
                }
        }
/* END */

/* START: fig7_8.txt */

        #define LeftChild( i )  ( 2 * ( i ) + 1 )
```



```
        void
        PercDown( ElementType A[ ], int i, int N )
        {
            int Child;
            ElementType Tmp;

/* 1*/      for( Tmp = A[ i ]; LeftChild( i ) &lt; N; i = Child )
            {
/* 2*/          Child = LeftChild( i );
/* 3*/          if( Child != N - 1 &amp;&amp; A[ Child + 1 ] &gt; A[ Child ] )
/* 4*/              Child++;
/* 5*/          if( Tmp &lt; A[ Child ] )
/* 6*/              A[ i ] = A[ Child ];
                else
/* 7*/              break;
            }
/* 8*/      A[ i ] =Tmp;
        }

        void
        Heapsort( ElementType A[ ], int N )
        {
            int i;

/* 1*/      for( i = N / 2; i &gt;= 0; i-- )  /* BuildHeap */
/* 2*/          PercDown( A, i, N );
```

```
/* 3*/      for( i = N - 1; i &gt; 0; i-- )
            {
/* 4*/          Swap( &amp;A[ 0 ], &amp;A[ i ] );  /* DeleteMax */
/* 5*/          PercDown( A, 0, i );
            }
        }
/* END */

/* START: fig7_10.txt */
        /* Lpos = start of left half, Rpos = start of right half */

        void
        Merge( ElementType A[ ], ElementType TmpArray[ ],
                        int Lpos, int Rpos, int RightEnd )
        {
            int i, LeftEnd, NumElements, TmpPos;

            LeftEnd = Rpos - 1;
            TmpPos = Lpos;
            NumElements = RightEnd - Lpos + 1;

            /* main loop */
            while( Lpos &lt;= LeftEnd &amp;&amp; Rpos &lt;= RightEnd )
                if( A[ Lpos ] &lt;= A[ Rpos ] )
                    TmpArray[ TmpPos++ ] = A[ Lpos++ ];
                else
                    TmpArray[ TmpPos++ ] = A[ Rpos++ ];

            while( Lpos &lt;= LeftEnd )  /* Copy rest of first half */
```



```
                TmpArray[ TmpPos++ ] = A[ Lpos++ ];
            while( Rpos &lt;= RightEnd ) /* Copy rest of second half */
                TmpArray[ TmpPos++ ] = A[ Rpos++ ];

            /* Copy TmpArray back */
            for( i = 0; i &lt; NumElements; i++, RightEnd-- )
                A[ RightEnd ] = TmpArray[ RightEnd ];
        }
/* END */

/* START: fig7_9.txt */
        void
        MSort( ElementType A[ ], ElementType TmpArray[ ],
                        int Left, int Right )
        {
            int Center;

            if( Left &lt; Right )
            {
                Center = ( Left + Right ) / 2;
                MSort( A, TmpArray, Left, Center );
                MSort( A, TmpArray, Center + 1, Right );
                Merge( A, TmpArray, Left, Center + 1, Right );
            }
        }

        void
        Mergesort( ElementType A[ ], int N )
        {
            ElementType *TmpArray;

            TmpArray = malloc( N * sizeof( ElementType ) );
            if( TmpArray != NULL )
            {
                MSort( A, TmpArray, 0, N - 1 );
                free( TmpArray );
```

```
            }
            else
                FatalError( "No space for tmp array!!!" );
        }
/* END */

/* START: fig7_13.txt */
        /* Return median of Left, Center, and Right */
        /* Order these and hide the pivot */

        ElementType
        Median3( ElementType A[ ], int Left, int Right )
        {
            int Center = ( Left + Right ) / 2;

            if( A[ Left ] &gt; A[ Center ] )
                Swap( &amp;A[ Left ], &amp;A[ Center ] );
            if( A[ Left ] &gt; A[ Right ] )
                Swap( &amp;A[ Left ], &amp;A[ Right ] );
            if( A[ Center ] &gt; A[ Right ] )
```



```
                Swap( &amp;A[ Center ], &amp;A[ Right ] );

            /* Invariant: A[ Left ] &lt;= A[ Center ] &lt;= A[ Right ] */

            Swap( &amp;A[ Center ], &amp;A[ Right - 1 ] );  /* Hide pivot */
            return A[ Right - 1 ];                 /* Return pivot */
        }
/* END */

/* START: fig7_14.txt */
        #define Cutoff ( 3 )

        void
        Qsort( ElementType A[ ], int Left, int Right )
        {
            int i, j;
            ElementType Pivot;

/* 1*/      if( Left + Cutoff &lt;= Right )
            {
/* 2*/          Pivot = Median3( A, Left, Right );
/* 3*/          i = Left; j = Right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( A[ ++i ] &lt; Pivot ){ }
/* 6*/              while( A[ --j ] &gt; Pivot ){ }
/* 7*/              if( i &lt; j )
/* 8*/                  Swap( &amp;A[ i ], &amp;A[ j ] );
                    else
/* 9*/                  break;
                }
/*10*/          Swap( &amp;A[ i ], &amp;A[ Right - 1 ] );  /* Restore pivot */

/*11*/          Qsort( A, Left, i - 1 );
/*12*/          Qsort( A, i + 1, Right );
            }
            else  /* Do an insertion sort on the subarray */
/*13*/          InsertionSort( A + Left, Right - Left + 1 );
        }
/* END */

        /* This code doesn't work; it's Figure 7.15. */
#if 0
/* START: fig7_15.txt */
/* 3*/          i = Left + 1; j = Right - 2;
```

```
/* 4*/          for( ; ; )
                {
/* 5*/              while( A[ i ] &lt; Pivot ) i++;
/* 6*/              while( A[ j ] &gt; Pivot ) j--;
/* 7*/              if( i &lt; j )
/* 8*/                  Swap( &amp;A[ i ], &amp;A[ j ] );
                    else
/* 9*/                  break;
                }
/* END */
#endif
```



```
/* START: fig7_12.txt */
        void
        Quicksort( ElementType A[ ], int N )
        {
            Qsort( A, 0, N - 1 );
        }
/* END */

/* START: fig7_16.txt */
        /* Places the kth smallest element in the kth position */
        /* Because arrays start at 0, this will be index k-1 */
        void
        Qselect( ElementType A[ ], int k, int Left, int Right )
        {
            int i, j;
            ElementType Pivot;

/* 1*/      if( Left + Cutoff &lt;= Right )
            {
/* 2*/          Pivot = Median3( A, Left, Right );
/* 3*/          i = Left; j = Right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( A[ ++i ] &lt; Pivot ){ }
/* 6*/              while( A[ --j ] &gt; Pivot ){ }
/* 7*/              if( i &lt; j )
/* 8*/                  Swap( &amp;A[ i ], &amp;A[ j ] );
                    else
/* 9*/                  break;
                }
/*10*/          Swap( &amp;A[ i ], &amp;A[ Right - 1 ] );  /* Restore pivot */

/*11*/          if( k &lt;= i )
/*12*/              Qselect( A, k, Left, i - 1 );
/*13*/          else if( k &gt; i + 1 )
/*14*/              Qselect( A, k, i + 1, Right );
            }
            else  /* Do an insertion sort on the subarray */
/*15*/          InsertionSort( A + Left, Right - Left + 1 );
        }
/* END */

        /* ROUTINES TO TEST THE SORTS */

        void
        Permute( ElementType A[ ], int N )
        {
            int i;

            for( i = 0; i &lt; N; i++ )
                A[ i ] = i;
            for( i = 1; i &lt; N; i++ )
                Swap( &amp;A[ i ], &amp;A[ rand( ) % ( i + 1 ) ] );
        }
```

```
}
```



```
void
Checksort( ElementType A[ ], int N )
{
    int i;
    for( i = 0; i &lt; N; i++ )
        if( A[ i ] != i )
            printf( "Sort fails: %d %d\n", i, A[ i ] );
    printf( "Check completed\n" );
}

void
Copy( ElementType Lhs[ ], const ElementType Rhs[ ], int N )
{
    int i;
    for( i = 0; i &lt; N; i++ )
        Lhs[ i ] = Rhs[ i ];
}

#define MaxSize 7000
int Arr1[ MaxSize ];
int Arr2[ MaxSize ];

main( )
{
    int i;

    for( i = 0; i &lt; 10; i++ )
    {
        Permute( Arr2, MaxSize );
        Copy( Arr1, Arr2, MaxSize );
        InsertionSort( Arr1, MaxSize );
        Checksort( Arr1, MaxSize );

        Copy( Arr1, Arr2, MaxSize );
        Shellsort( Arr1, MaxSize );
        Checksort( Arr1, MaxSize );

        Copy( Arr1, Arr2, MaxSize );
        Heapsort( Arr1, MaxSize );
        Checksort( Arr1, MaxSize );

        Copy( Arr1, Arr2, MaxSize );
        Mergesort( Arr1, MaxSize );
        Checksort( Arr1, MaxSize );

        Copy( Arr1, Arr2, MaxSize );
        Quicksort( Arr1, MaxSize );
        Checksort( Arr1, MaxSize );

        Copy( Arr1, Arr2, MaxSize );
        Qselect( Arr1, MaxSize / 2 + 1 + i, 0, MaxSize - 1 );
        if( Arr1[ MaxSize / 2 + i ] != MaxSize / 2 + i )
            printf( "Select error: %d %d\n", MaxSize / 2 + i ,
                                    Arr1[ MaxSize / 2 + i ] );
        else
            printf( "Select works\n" );
```

```
            }

            return 0;
        }
```

Aug 12 15:27 1996 disjsets.c Page 1

```
        /* Disjoint set data structure */
        /* All in one file because it's so short */

#define FastAlg

#define NumSets 128
```

```
/* START: fig8_6.txt */
        #ifndef _DisjSet_H

        typedef int DisjSet[ NumSets + 1 ];
        typedef int SetType;
        typedef int ElementType;

        void Initialize( DisjSet S );
        void SetUnion( DisjSet S, SetType Root1, SetType Root2 );
        SetType Find( ElementType X, DisjSet S );

        #endif  /* _DisjSet_H */
/* END */

/* START: fig8_7.txt */
        void
        Initialize( DisjSet S )
        {
            int i;

            for( i = NumSets; i &gt; 0; i-- )
                S[ i ] = 0;
        }
/* END */

#ifdef SlowAlg
/* START: fig8_8.txt */
        /* Assumes Root1 and Root2 are roots */
        /* union is a C keyword, so this routine */
        /* is named SetUnion */

        void
        SetUnion( DisjSet S, SetType Root1, SetType Root2 )
        {
            S[ Root2 ] = Root1;
        }
/* END */

/* START: fig8_9.txt */
        SetType
        Find( ElementType X, DisjSet S )
        {
            if( S[ X ] &lt;= 0 )
                return X;
            else
                return Find( S[ X ], S );
        }
/* END */
#else
```



```
/* START: fig8_13.txt */
        /* Assume Root1 and Root2 are roots */
        /* union is a C keyword, so this routine */
        /* is named SetUnion */

        void
        SetUnion( DisjSet S, SetType Root1, SetType Root2 )
        {
            if( S[ Root2 ] &lt; S[ Root1 ] ) /* Root2 is deeper set */
                S[ Root1 ] = Root2;  /* Make Root2 new root */
            else
            {
                if( S[ Root1 ] == S[ Root2 ] )  /* Same height, */
                    S[ Root1 ]--;               /* so update */
                S[ Root2 ] = Root1;
```

```
            }
        }
/* END */

/* START: fig8_15.txt */
        SetType
        Find( ElementType X, DisjSet S )
        {
            if( S[ X ] &lt;= 0 )
                return X;
            else
                return S[ X ] = Find( S[ X ], S );
        }
/* END */
#endif

main( )
{
    DisjSet S;
    int i, j, k, Set1, Set2;

    Initialize( S );
    j = k = 1;
    while( k &lt;= 8 )
    {
        j = 1;
        while( j &lt; NumSets )
        {
            Set1 = Find( j, S );
            Set2 = Find( j + k, S );
            SetUnion( S, Set1, Set2 );
            j += 2 * k;
        }
        k *= 2;
    }
    i = 1;
    for( i = 1; i &lt;= NumSets; i++ )
    {
        Set1 = Find( i, S );
        printf( "%d**", Set1 );
```



```
    }
    printf( "\n" );
    return 0;
}
```

```
        #include &lt;stdio.h&gt;

typedef double Matrix[ 2 ][ 2 ];

/* START: fig10_38.txt */
        /* Standard matrix multiplication */
        /* Arrays start at 0 */

        void
        MatrixMultiply( Matrix A, Matrix B, Matrix C, int N )
        {
            int i, j, k;

            for( i = 0; i &lt; N; i++ )  /* Initialization */
                for( j = 0; j &lt; N; j++ )
                    C[ i ][ j ] = 0.0;

            for( i = 0; i &lt; N; i++ )
                for( j = 0; j &lt; N; j++ )
                    for( k = 0; k &lt; N; k++ )
                        C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
         }
/* END */

main( )
{
    Matrix A = { { 1, 2 }, { 3, 4 } };
    Matrix C;

    MatrixMultiply( A, A, C, 2 );
    printf( "%6.2f %6.2f\n%6.2f %6.2f\n", C[ 0 ][ 0 ], C[ 0 ][ 1 ],
          C[ 1 ][ 0 ], C[ 1 ][ 1 ] );
    return 0;
}
```

```
#include &lt;stdio.h&gt;

/* START: fig10_40.txt */
        /* Compute Fibonacci numbers as described in Chapter 1 */

        int
        Fib( int N )
        {
            if( N &lt;= 1 )
                return 1;
            else
                return Fib( N - 1 ) + Fib( N - 2 );
        }
/* END */

/* START: fig10_41.txt */
        int
        Fibonacci( int N )
        {
            int i, Last, NextToLast, Answer;

            if( N &lt;= 1 )
                return 1;

            Last = NextToLast = 1;
            for( i = 2; i &lt;= N; i++ )
            {
                Answer = Last + NextToLast;
                NextToLast = Last;
                Last = Answer;
            }

            return Answer;
        }
/* END */

main( )
{
    printf( "%d\n%d\n", Fib( 7 ), Fibonacci( 7 ) );
    return 0;
}
```

```
#include &lt;stdio.h&gt;

/* START: fig10_43.txt */
        double
        Eval( int N )
        {
            int i;
            double Sum;

            if( N == 0 )
                return 1.0;
            else
            {
                Sum = 0.0;
                for( i = 0; i &lt; N; i++ )
                    Sum += Eval( i );
                return 2.0 * Sum / N + N;
            }
        }
/* END */

main( )
{
    printf( "%f\n", Eval( 10 ) );
    return 0;
}
```

```
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;
#include "fatal.h"


/* START: fig10_45.txt */
        double
        Eval( int N )
        {
            int i, j;
            double Sum, Answer;
            double *C;

            C = malloc( sizeof( double ) * ( N + 1 ) );
            if( C == NULL )
                FatalError( "Out of space!!!" );

            C[ 0 ] = 1.0;
            for( i = 1; i &lt;= N; i++ )
            {
                Sum = 0.0;
                for( j = 0; j &lt; i; j++ )
                    Sum += C[ j ];
                C[ i ] = 2.0 * Sum / i + i;
            }

            Answer = C[ N ];
            free( C );

            return Answer;
        }
/* END */

main( )
{
    printf( "%f\n", Eval( 10 ) );
    return 0;
}
```

```
        #include &lt;stdio.h&gt;
        #include &lt;limits.h&gt;

        typedef long int TwoDimArray[ 5 ][ 5 ];
        #define Infinity INT_MAX

/* START: fig10_46.txt */
        /* Compute optimal ordering of matrix multiplication */
        /* C contains number of columns for each of the N matrices */
        /* C[ 0 ] is the number of rows in matrix 1 */
        /* Minimum number of multiplications is left in M[ 1 ][ N ] */
        /* Actual ordering is computed via */
        /* another procedure using LastChange */
        /* M and LastChange are indexed starting at 1, instead of 0 */
        /* Note: Entries below main diagonals of M and LastChange */
        /* are meaningless and uninitialized */

        void
        OptMatrix( const long C[ ], int N,
                   TwoDimArray M, TwoDimArray LastChange )
        {
            int i, k, Left, Right;
            long ThisM;

            for( Left = 1; Left &lt;= N; Left++ )
                M[ Left ][ Left ] = 0;
            for( k = 1; k &lt; N; k++ )   /* k is Right - Left */
                for( Left = 1; Left &lt;= N - k; Left++ )
                {
                    /* For each position */
                    Right = Left + k;
                    M[ Left ][ Right ] = Infinity;
                    for( i = Left; i &lt; Right; i++ )
                    {
                        ThisM = M[ Left ][ i ] + M[ i + 1 ][ Right ]
                                + C[ Left - 1 ] * C[ i ] * C[ Right ];
                        if( ThisM &lt; M[ Left ][ Right ] ) /* Update min */
                        {
                            M[ Left ][ Right ] = ThisM;
                            LastChange[ Left ][ Right ] = i;
                        }
                    }
                }
        }
/* END */

main( )
{
    long C[ ] = { 50, 10, 40, 30, 5 };
    long M[ 5 ][ 5 ], LastChange[ 5 ][ 5 ];
    int i, j;

    OptMatrix( C, 4, M, LastChange );
    for( i = 1; i &lt;= 4; i++ )
    {
        for( j = 1; j &lt;= 4; j++ )
```



```
            printf( "%14d", M[ i ][ j ] );
        printf( "\n" );
    }
    for( i = 1; i &lt;= 4; i++ )
    {
        for( j = 1; j &lt;= 4; j++ )
```

```
            printf( "%14d", LastChange[ i ][ j ] );
        printf( "\n" );
    }
    return 0;
}
```

Aug 12 15:27 1996 fig10_53.c Page 1

```
        #include &lt;stdio.h&gt;

        #define NotAVertex (-1)
        typedef int TwoDimArray[ 4 ][ 4 ];

/* START: fig10_53.txt */
        /* Compute All-Shortest Paths */
        /* A[ ] contains the adjacency matrix */
        /* with A[ i ][ i ] presumed to be zero */
        /* D[ ] contains the values of the shortest path */
        /* N is the number of vertices */
        /* A negative cycle exists iff */
        /* D[ i ][ i ] is set to a negative value */
        /* Actual path can be computed using Path[ ] */
        /* All arrays are indexed starting at 0 */
```

```
        /* NotAVertex is -1 */

        void
        AllPairs( TwoDimArray A, TwoDimArray D,
                        TwoDimArray Path, int N )
        {
            int i, j, k;

            /* Initialize D and Path */
/* 1*/      for( i = 0; i &lt; N; i++ )
/* 2*/          for( j = 0; j &lt; N; j++ )
                {
/* 3*/              D[ i ][ j ] = A[ i ][ j ];
/* 4*/              Path[ i ][ j ] = NotAVertex;
                }

/* 5*/      for( k = 0; k &lt; N; k++ )
                /* Conider each vertex as an intermediate */
/* 6*/          for( i = 0; i &lt; N; i++ )
/* 7*/              for( j = 0; j &lt; N; j++ )
/* 8*/                  if( D[ i ][ k ] + D[ k ][ j ] &lt; D[ i ][ j ] )
                        {
                            /* Update shortest path */
/* 9*/                      D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
/*10*/                      Path[ i ][ j ] = k;
                        }
        }
/* END */

main( )
{
    TwoDimArray A = { { 0, 2, -2, 2 }, { 1000, 0, -3, 1000 },
                      { 4, 1000, 0, 1000 }, { 1000, -2, 3, 0 } };
    TwoDimArray D, Path;
    int i, j;

    AllPairs( A, D, Path, 4 );
    for( i = 0; i &lt; 4; i++ )
    {
        for( j = 0; j &lt; 4; j++ )
            printf( "%6d ", D[ i ][ j ] );
```



```
        printf( "\n" );
    }

    for( i = 0; i &lt; 4; i++ )
    {
        for( j = 0; j &lt; 4; j++ )
            printf( "%6d ", Path[ i ][ j ] );
        printf( "\n" );
    }


    return 0;
}
```

```
        /* Bad random number generator */

        #include &lt;stdio.h&gt;

/* START: fig10_55.txt */
        static unsigned long Seed = 1;

        #define A 48271L
        #define M 2147483647L
        #define Q ( M / A )
        #define R ( M % A )

        double
        Random( void )
        {
            long TmpSeed;

            TmpSeed = A * ( Seed % Q ) - R * ( Seed / Q );
            if( TmpSeed &gt;= 0 )
                Seed = TmpSeed;
            else
                Seed = TmpSeed + M;

            return ( double ) Seed / M;
        }

        void
        Initialize( unsigned long InitVal )
        {
            Seed = InitVal;
        }
/* END */

main( )
```

```
main( )
{
    int i;

    for( i = 0; i &lt; 20; i++ )
        printf( "%6f\n", Random( ) );
    return 0;
}
```



```
        #include &lt;stdio.h&gt;
        #include &lt;stdlib.h&gt;

        typedef long HugeInt;

        HugeInt
        RandInt( HugeInt Low, HugeInt High )
        {
            return rand( ) % ( High - Low + 1 ) + Low;
        }


/* START: fig10_62.txt */
        /* If Witness does not return 1, N is definitely composite */
        /* Do this by computing ( A ^ i ) mod N and looking for */
        /* non-trivial square roots of 1 along the way */
        /* We are assuming very large numbers, so this is pseudocode */

        HugeInt
        Witness( HugeInt A, HugeInt i, HugeInt N )
        {
            HugeInt X, Y;

            if( i == 0 )
                return 1;

            X = Witness( A, i / 2, N );
            if( X == 0 )  /* If N is recursively composite, stop */
                return 0;

            /* N is not prime if we find a non-trivial root of 1 */
            Y = ( X * X ) % N;
            if( Y == 1 &amp;&amp; X != 1 &amp;&amp; X != N - 1 )
                return 0;

            if( i % 2 != 0 )
                Y = ( A * Y ) % N;

            return Y;
        }

        /* IsPrime: Test if N &gt;= 3 is prime using one value of A */
        /* Repeat this procedure as many times as needed for */
```

```
        /* desired error rate */

        int
        IsPrime( HugeInt N )
        {
            return Witness( RandInt( 2, N - 2 ), N - 1, N ) == 1;
        }
/* END */

main( )
{   int i;

    for( i = 101; i &lt; 200; i += 2 )
```



```
        if( IsPrime( i ) )
            printf( "%d is prime\n", i );

    return 0;
}
```

```
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        typedef int ElementType;
        #define Infinity 30000
        #define NegInfinity (-30000)

/* START: fig12_5.txt */
        #ifndef _Splay_H
        #define _Splay_H

        struct SplayNode;
        typedef struct SplayNode *SplayTree;

        SplayTree MakeEmpty( SplayTree T );
        SplayTree Find( ElementType X, SplayTree T );
        SplayTree FindMin( SplayTree T );
        SplayTree FindMax( SplayTree T );
        SplayTree Initialize( void );
        SplayTree Insert( ElementType X, SplayTree T );
        SplayTree Remove( ElementType X, SplayTree T );
        ElementType Retrieve( SplayTree T );  /* Gets root item */

        #endif  /* _Splay_H */
/* END */
```

```
#include "splay.h"
#include &lt;stdlib.h&gt;
#include "fatal.h"

struct SplayNode
{
    ElementType Element;
    SplayTree      Left;
    SplayTree      Right;
};

typedef struct SplayNode *Position;
static Position NullNode = NULL;  /* Needs initialization */

SplayTree
Initialize( void )
{
    if( NullNode == NULL )
    {
        NullNode = malloc( sizeof( struct SplayNode ) );
        if( NullNode == NULL )
            FatalError( "Out of space!!!" );
        NullNode-&gt;Left = NullNode-&gt;Right = NullNode;
    }
    return NullNode;
}

static SplayTree Splay( ElementType Item, Position X );

SplayTree
MakeEmpty( SplayTree T )
{
    if( T != NullNode )
    {
        MakeEmpty( T-&gt;Left );
        MakeEmpty( T-&gt;Right );
        free( T );
    }
    return NullNode;
}

void
PrintTree( SplayTree T )
{
    if( T != NullNode )
    {
        PrintTree( T-&gt;Left );
        printf( "%d ", T-&gt;Element );
        PrintTree( T-&gt;Right );
    }
}

SplayTree
Find( ElementType X, SplayTree T )
{
    return Splay( X, T );
```



```
}

SplayTree
FindMin( SplayTree T )
{
```

```
            return Splay( NegInfinity, T );
        }

        SplayTree
        FindMax( SplayTree T )
        {
            return Splay( Infinity, T );
        }

        /* 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-&gt;Left;
            K2-&gt;Left = K1-&gt;Right;
            K1-&gt;Right = K2;

            return K1;  /* New root */
        }

        /* 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-&gt;Right;
            K1-&gt;Right = K2-&gt;Left;
            K2-&gt;Left = K1;

            return K2;  /* New root */
        }

/* START: fig12_6.txt */
        /* Top-down splay procedure, */
        /* not requiring Item to be in tree */

        SplayTree
        Splay( ElementType Item, Position X )
        {
            static struct SplayNode Header;
            Position LeftTreeMax, RightTreeMin;
```



```
            Header.Left = Header.Right = NullNode;
            LeftTreeMax = RightTreeMin = &amp;Header;
            NullNode-&gt;Element = Item;

            while( Item != X-&gt;Element )
            {
                if( Item &lt; X-&gt;Element )
                {
                    if( Item &lt; X-&gt;Left-&gt;Element )
                        X = SingleRotateWithLeft( X );
                    if( X-&gt;Left == NullNode )
                        break;
                    /* Link right */
                    RightTreeMin-&gt;Left = X;
```

```
                    RightTreeMin = X;
                    X = X-&gt;Left;
                }
                else
                {
                    if( Item &gt; X-&gt;Right-&gt;Element )
                        X = SingleRotateWithRight( X );
                    if( X-&gt;Right == NullNode )
                        break;
                    /* Link left */
                    LeftTreeMax-&gt;Right = X;
                    LeftTreeMax = X;
                    X = X-&gt;Right;
                }
            }  /* while Item != X-&gt;Element */

            /* Reassemble */
            LeftTreeMax-&gt;Right = X-&gt;Left;
            RightTreeMin-&gt;Left = X-&gt;Right;
            X-&gt;Left = Header.Right;
            X-&gt;Right = Header.Left;

            return X;
        }
/* END */




/* START: fig12_7.txt */
        SplayTree
        Insert( ElementType Item, SplayTree T )
        {
            static Position NewNode = NULL;

            if( NewNode == NULL )
            {
                NewNode = malloc( sizeof( struct SplayNode ) );
                if( NewNode == NULL )
                    FatalError( "Out of space!!!" );
            }
            NewNode-&gt;Element = Item;
```



```
            if( T == NullNode )
            {
                NewNode-&gt;Left = NewNode-&gt;Right = NullNode;
                T = NewNode;
            }
            else
            {
                T = Splay( Item, T );
                if( Item &lt; T-&gt;Element )
                {
                    NewNode-&gt;Left = T-&gt;Left;
                    NewNode-&gt;Right = T;
                    T-&gt;Left = NullNode;
                    T = NewNode;
                }
                else
                if( T-&gt;Element &lt; Item )
                {
                    NewNode-&gt;Right = T-&gt;Right;
                    NewNode-&gt;Left = T;
                    T-&gt;Right = NullNode;
                    T = NewNode;
                }
```

```
                }
                else
                    return T;  /* Already in the tree */
            }

            NewNode = NULL;   /* So next insert will call malloc */
            return T;
        }
/* END */


/* START: fig12_8.txt */
        SplayTree
        Remove( ElementType Item, SplayTree T )
        {
            Position NewTree;

            if( T != NullNode )
            {
                T = Splay( Item, T );
                if( Item == T-&gt;Element )
                {
                    /* Found it! */
                    if( T-&gt;Left == NullNode )
                        NewTree = T-&gt;Right;
                    else
                    {
                        NewTree = T-&gt;Left;
                        NewTree = Splay( Item, NewTree );
                        NewTree-&gt;Right = T-&gt;Right;
                    }
                    free( T );
                    T = NewTree;
```



```
                }
            }

            return T;
        }

/* END */

        ElementType
        Retrieve( SplayTree T )
        {
            return T-&gt;Element;
        }
```

```
#include "splay.h"
#include &lt;stdio.h&gt;
#define NumItems 500

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

    T = Initialize( );
    T = MakeEmpty( T );
    for( i = 0; i &lt; NumItems; i++, j = ( j + 7 ) % NumItems )
        T = Insert( j, T );

    for( j = 0; j &lt; 2; j++ )
        for( i = 0; i &lt; NumItems; i++ )
        {
            T = Find( i, T );
            if( Retrieve( T ) != i )
                printf( "Error1 at %d\n", i );
        }

    printf( "Entering remove\n" );

    for( i = 0; i &lt; NumItems; i += 2 )
        T = Remove( i, T );

    for( i = 1; i &lt; NumItems; i += 2 )
    {
        T = Find( i, T );
        if( Retrieve( T ) != i )
            printf( "Error2 at %d\n", i );
    }

    for( i = 0; i &lt; NumItems; i += 2 )
    {
        T = Find( i, T );
        if( Retrieve( T ) == i )
            printf( "Error3 at %d\n", i );
    }
```

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

    return 0;
}
```

Nov 4 20:35 1997 dsl.h Page 1

```
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        typedef int ElementType;
        #define Infinity (10000)

        #ifndef _SkipList_H
        #define _SkipList_H

        struct SkipNode;
        typedef struct SkipNode *Position;
        typedef struct SkipNode *SkipList;

        SkipList MakeEmpty( SkipList L );
        Position Find( ElementType X, SkipList L );
        Position FindMin( SkipList L );
        Position FindMax( SkipList L );
        SkipList Initialize( void );
        SkipList Insert( ElementType X, SkipList L );
        SkipList Remove( ElementType X, SkipList L );
        ElementType Retrieve( Position P );

        #endif  /* _SkipList_H */

/* END */
```

```
#include "dsl.h"
#include &lt;stdlib.h&gt;
#include "fatal.h"


/* START: fig12_23.txt */
        struct SkipNode
        {
            ElementType Element;
            SkipList    Right;
            SkipList    Down;
        };

        static Position Bottom = NULL;  /* Needs initialization */
        static Position Tail   = NULL;  /* Needs initialization */

        /* Initialization procedure */

        SkipList
        Initialize( void )
        {
            SkipList L;

            if( Bottom == NULL )
            {
                Bottom = malloc( sizeof( struct SkipNode ) );
                if( Bottom == NULL )
                    FatalError( "Out of space!!!" );
                Bottom-&gt;Right = Bottom-&gt;Down = Bottom;

                Tail = malloc( sizeof( struct SkipNode ) );
                if( Tail == NULL )
                    FatalError( "Out of space!!!" );
                Tail-&gt;Element = Infinity;
                Tail-&gt;Right = Tail;
            }

            /* Create the header node */
            L = malloc( sizeof( struct SkipNode ) );
            if( L == NULL )
                FatalError( "Out of space!!!" );
            L-&gt;Element = Infinity;
            L-&gt;Right = Tail;
            L-&gt;Down = Bottom;

            return L;
        }
/* END */

        void
        Output( ElementType Element )
        {
            printf( "%d\n", Element );
        }

        /* Memory reclamation is left as an exercise */
```

```
        /* Hint: Delete from top level to bottom level */
        SkipList
        MakeEmpty( SkipList L )
        {
            L-&gt;Right = Tail;
            L-&gt;Down = Bottom;
            return L;
        }

/* START: fig12_24.txt */
        /* Return Position of node containing Item, */
        /* or Bottom if not found */

        Position
        Find( ElementType Item, SkipList L )
        {
            Position Current = L;

            Bottom-&gt;Element = Item;
            while( Item != Current-&gt;Element )
                if( Item &lt; Current-&gt;Element )
                    Current = Current-&gt;Down;
                else
                    Current = Current-&gt;Right;

            return Current;
        }
/* END */

        Position
        FindMin( SkipList L )
        {
            Position Current = L;

            while( Current-&gt;Down != Bottom )
                Current = Current-&gt;Down;

            return Current;
        }

        Position
        FindMax( SkipList L )
        {
            Position Current = L;

            while( Current-&gt;Right-&gt;Right != Tail ||
                   Current-&gt;Down != Bottom )
            {
                if( Current-&gt;Right-&gt;Right != Tail )
                    Current = Current-&gt;Right;
                else
                    Current = Current-&gt;Down;
            }

            return Current;
        }
```



```
/* START: fig12_25.txt */
        SkipList
        Insert( ElementType Item, SkipList L )
```

```
        Insert( ElementType Item, SkipList L )
        {
            Position Current = L;
            Position NewNode;

            Bottom-&gt;Element = Item;
            while( Current != Bottom )
            {
                while( Item &gt; Current-&gt;Element )
                    Current = Current-&gt;Right;

                /* If gap size is 3 or at bottom level */
                /* and must insert, then promote middle element */
                if( Current-&gt;Element &gt;
                          Current-&gt;Down-&gt;Right-&gt;Right-&gt;Element )
                {
                    NewNode = malloc( sizeof( struct SkipNode ) );
                    if( NewNode == NULL )
                        FatalError( "Out of space!!!" );
                    NewNode-&gt;Right = Current-&gt;Right;
                    NewNode-&gt;Down = Current-&gt;Down-&gt;Right-&gt;Right;
                    Current-&gt;Right = NewNode;
                    NewNode-&gt;Element = Current-&gt;Element;
                    Current-&gt;Element = Current-&gt;Down-&gt;Right-&gt;Element;
                }
                else
                    Current = Current-&gt;Down;
            }

            /* Raise height of DSL if necessary */
            if( L-&gt;Right != Tail )
            {
                NewNode = malloc( sizeof( struct SkipNode ) );
                if( NewNode == NULL )
                    FatalError( "Out of space!!!" );
                NewNode-&gt;Down = L;
                NewNode-&gt;Right = Tail;
                NewNode-&gt;Element = Infinity;
                L = NewNode;
            }

            return L;
        }
/* END */

        SkipList
        Remove( ElementType Item, SkipList L )
        {
            printf( "Remove is unimplemented\n" );
            if( Item )
                return L;
            return L;
        }
```



```
        ElementType
        Retrieve( Position P )
        {
            return P-&gt;Element;
        }
```

```
#include "dsl.h"
#include &lt;stdio.h&gt;

#define N 800

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

    T = Initialize( );
    T = MakeEmpty( T );

    for( i = 0; i &lt; N; i++, j = ( j + 7 ) % N )
        T = Insert( j, T );
    printf( "Inserts are complete\n" );


    for( i = 0; i &lt; N; i++ )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
```

```
        printf( "Error at %d\n", i );

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

    return 0;
}
```



```
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        typedef int ElementType;

        #ifndef _AATree_H
        #define _AATree_H

        struct AANode;
        typedef struct AANode *Position;
        typedef struct AANode *AATree;

        AATree MakeEmpty( AATree T );
        Position Find( ElementType X, AATree T );
        Position FindMin( AATree T );
        Position FindMax( AATree T );
        AATree Initialize( void );
        AATree Insert( ElementType X, AATree T );
        AATree Remove( ElementType X, AATree T );
        ElementType Retrieve( Position P );

        extern Position NullNode;

        #endif  /* _AATree_H */

/* END */
```

```
#include "aatree.h"
#include &lt;stdlib.h&gt;
#include "fatal.h"

/* START: fig12_27.txt */
            /* Returned for failures */
        Position NullNode = NULL;  /* Needs more initialization */

        struct AANode
        {
            ElementType Element;
            AATree      Left;
            AATree      Right;
            int         Level;
        };

        AATree
        Initialize( void )
        {
            if( NullNode == NULL )
            {
                NullNode = malloc( sizeof( struct AANode ) );
                if( NullNode == NULL )
                    FatalError( "Out of space!!!" );
                NullNode-&gt;Left = NullNode-&gt;Right = NullNode;
                NullNode-&gt;Level = 0;
            }
            return NullNode;
        }
/* END */

        AATree
        MakeEmpty( AATree T )
        {
            if( T != NullNode )
            {
                MakeEmpty( T-&gt;Left );
                MakeEmpty( T-&gt;Right );
                free( T );
            }
            return NullNode;
```

```
            return NullNode;
        }

        Position
        Find( ElementType X, AATree T )
        {
            if( T == NullNode )
                return NullNode;
            if( X &lt; T-&gt;Element )
                return Find( X, T-&gt;Left );
            else
            if( X &gt; T-&gt;Element )
                return Find( X, T-&gt;Right );
            else
                return T;
        }
```



```
        Position
        FindMin( AATree T )
        {
            if( T == NullNode )
                return NullNode;
            else
            if( T-&gt;Left == NullNode )
                return T;
            else
                return FindMin( T-&gt;Left );
        }

        Position
        FindMax( AATree T )
        {
            if( T != NullNode )
                while( T-&gt;Right != NullNode )
                    T = T-&gt;Right;

            return T;
        }

        /* 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-&gt;Left;
            K2-&gt;Left = K1-&gt;Right;
            K1-&gt;Right = K2;

            return K1;  /* New root */
        }

        /* 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-&gt;Right;
            K1-&gt;Right = K2-&gt;Left;
```

```
            K2-&gt;Left = K1;

            return K2;  /* New root */
        }

/* START: fig12_29.txt */
```



```
        /* If T's left child is on the same level as T, */
        /* perform a rotation */

        AATree
        Skew( AATree T )
        {
            if( T-&gt;Left-&gt;Level == T-&gt;Level )
                T = SingleRotateWithLeft( T );
            return T;
        }

        /* If T's rightmost grandchild is on the same level, */
        /* rotate right child up */

        AATree
        Split( AATree T )
        {
            if( T-&gt;Right-&gt;Right-&gt;Level == T-&gt;Level )
            {
                T = SingleRotateWithRight( T );
                T-&gt;Level++;
            }
            return T;
        }
/* END */



/* START: fig12_36.txt */
        AATree
        Insert( ElementType Item, AATree T )
        {
            if( T == NullNode )
            {
                /* Create and return a one-node tree */
                T = malloc( sizeof( struct AANode ) );
                if( T == NULL )
                    FatalError( "Out of space!!!" );
                else
                {
                    T-&gt;Element = Item; T-&gt;Level = 1;
                    T-&gt;Left = T-&gt;Right = NullNode;
                }
            }
            else
            if( Item &lt; T-&gt;Element )
                T-&gt;Left = Insert( Item, T-&gt;Left );
            else
            if( Item &gt; T-&gt;Element )
                T-&gt;Right = Insert( Item, T-&gt;Right );

            /* Otherwise it's a duplicate; do nothing */

            T = Skew( T );
            T = Split( T );
```

```
            return T;
        }
/* END */

/* START: fig12_38.txt */
        AATree
        Remove( ElementType Item, AATree T )
        {
            static Position DeletePtr, LastPtr;

            if( T != NullNode )
            {
                /* Step 1: Search down tree */
                /*         set LastPtr and DeletePtr */
                LastPtr = T;
                if( Item &lt; T-&gt;Element )
                    T-&gt;Left = Remove( Item, T-&gt;Left );
                else
                {
                    DeletePtr = T;
                    T-&gt;Right = Remove( Item, T-&gt;Right );
                }

                /* Step 2: If at the bottom of the tree and */
                /*         item is present, we remove it */
                if( T == LastPtr )
                {
                    if( DeletePtr != NullNode &amp;&amp;
                            Item == DeletePtr-&gt;Element )
                    {
                        DeletePtr-&gt;Element = T-&gt;Element;
                        DeletePtr = NullNode;
                        T = T-&gt;Right;
                        free( LastPtr );
                    }
                }

                /* Step 3: Otherwise, we are not at the bottom; */
                /*         rebalance */
                else
                    if( T-&gt;Left-&gt;Level &lt; T-&gt;Level - 1 ||
                        T-&gt;Right-&gt;Level &lt; T-&gt;Level - 1 )
                    {
                        if( T-&gt;Right-&gt;Level &gt; --T-&gt;Level )
                            T-&gt;Right-&gt;Level = T-&gt;Level;
                        T = Skew( T );
                        T-&gt;Right = Skew( T-&gt;Right );
                        T-&gt;Right-&gt;Right = Skew( T-&gt;Right-&gt;Right );
                        T = Split( T );
                        T-&gt;Right = Split( T-&gt;Right );
                    }
            }
            return T;
        }
/* END */
```



```
        ElementType
        Retrieve( Position P )
        {
```

```
        {
            return P-&gt;Element;
        }
```



```
#include "aatree.h"
#include &lt;stdio.h&gt;

#define NumItems 20

main( )
{

    AATree T;
    Position P;
    int i;
    int j = 0;
```

```c
    int j = 0;

    T = Initialize( );
    T = MakeEmpty( NullNode );
    for( i = 0; i &lt; NumItems; i++, j = ( j + 7 ) % NumItems )
        T = Insert( j, T );
    for( i = 0; i &lt; NumItems; i++ )
        if( ( P = Find( i, T ) ) == NullNode || Retrieve( P ) != i )
            printf( "Error at %d\n", i );

    for( i = 0; i &lt; NumItems; i += 2 )
        T = Remove( i, T );

    for( i = 1; i &lt; NumItems; i += 2 )
        if( ( P = Find( i, T ) ) == NullNode || Retrieve( P ) != i )
            printf( "Error at %d\n", i );

    for( i = 0; i &lt; NumItems; i += 2 )
        if( ( P = Find( i, T ) ) != NullNode )
            printf( "Error at %d\n", i );

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

    return 0;
}
```



```c
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        typedef int ElementType;
        #define Infinity 32767

        #ifndef _Treap_H
        #define _Treap_H

        struct TreapNode;
        typedef struct TreapNode *Position;
        typedef struct TreapNode *Treap;

        Treap MakeEmpty( Treap T );
        Position Find( ElementType X, Treap T );
        Position FindMin( Treap T );
        Position FindMax( Treap T );
        Treap Initialize( void );
        Treap Insert( ElementType X, Treap T );
        Treap Remove( ElementType X, Treap T );
        ElementType Retrieve( Position P );
```

```
        extern Position NullNode;

        #endif  /* _Treap_H */

/* END */
```



```
#include "treap.h"
#include &lt;stdlib.h&gt;
#include "fatal.h"


        struct TreapNode
        {
            ElementType Element;
            Treap       Left;
            Treap       Right;
            int         Priority;
        };

        Position NullNode = NULL;  /* Needs initialization */

/* START: fig12_39.txt */
        Treap
        Initialize( void )
        {
            if( NullNode == NULL )
            {
                NullNode = malloc( sizeof( struct TreapNode ) );
                if( NullNode == NULL )
                    FatalError( "Out of space!!!" );
                NullNode-&gt;Left = NullNode-&gt;Right = NullNode;
                NullNode-&gt;Priority = Infinity;
            }
            return NullNode;
        }
/* END */
```

```
/* Use ANSI C random number generator for simplicity */

int
Random( void )
{
    return rand( ) -  1;
}

Treap
MakeEmpty( Treap T )
{
    if( T != NullNode )
    {
        MakeEmpty( T-&gt;Left );
        MakeEmpty( T-&gt;Right );
        free( T );
    }
    return NullNode;
}

void
PrintTree( Treap T )
{
    if( T != NullNode )
    {
```



```
            PrintTree( T-&gt;Left );
            printf( "%d ", T-&gt;Element );
            PrintTree( T-&gt;Right );
        }
    }

    Position
    Find( ElementType X, Treap T )
    {
        if( T == NullNode )
            return NullNode;
        if( X &lt; T-&gt;Element )
            return Find( X, T-&gt;Left );
        else
        if( X &gt; T-&gt;Element )
            return Find( X, T-&gt;Right );
        else
            return T;
    }

    Position
    FindMin( Treap T )
    {
        if( T == NullNode )
            return NullNode;
        else
        if( T-&gt;Left == NullNode )
            return T;
        else
            return FindMin( T-&gt;Left );
    }

    Position
    FindMax( Treap T )
    {
        if( T != NullNode )
            while( T-&gt;Right != NullNode )
                T = T-&gt;Right;

        return T;
```

```
            return T;
        }

        /* 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-&gt;Left;
            K2-&gt;Left = K1-&gt;Right;
            K1-&gt;Right = K2;

            return K1;  /* New root */
```



```
        }

        /* 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-&gt;Right;
            K1-&gt;Right = K2-&gt;Left;
            K2-&gt;Left = K1;

            return K2;  /* New root */
        }

/* START: fig12_40.txt */
        Treap
        Insert( ElementType Item, Treap T )
        {
            if( T == NullNode )
            {
                /* Create and return a one-node tree */
                T = malloc( sizeof( struct TreapNode ) );
                if( T == NULL )
                    FatalError( "Out of space!!!" );
                else
                {
                    T-&gt;Element = Item; T-&gt;Priority = Random( );
                    T-&gt;Left = T-&gt;Right = NullNode;
                }
            }
            else
            if( Item &lt; T-&gt;Element )
            {
                T-&gt;Left = Insert( Item, T-&gt;Left );
                if( T-&gt;Left-&gt;Priority &lt; T-&gt;Priority )
                    T = SingleRotateWithLeft( T );
            }
            else
            if( Item &gt; T-&gt;Element )
            {
                T-&gt;Right = Insert( Item, T-&gt;Right );
                if( T-&gt;Right-&gt;Priority &lt; T-&gt;Priority )
                    T = SingleRotateWithRight( T );
            }
```

```
            /* Otherwise it's a duplicate; do nothing */

            return T;
        }
/* END */

/* START: fig12_41.txt */
```



```
        Treap
        Remove( ElementType Item, Treap T )
        {
            if( T != NullNode )
            {
                if( Item &lt; T-&gt;Element )
                    T-&gt;Left = Remove( Item, T-&gt;Left );
                else
                if( Item &gt; T-&gt;Element )
                    T-&gt;Right = Remove( Item, T-&gt;Right );
                else
                {
                    /* Match found */
                    if( T-&gt;Left-&gt;Priority &lt; T-&gt;Right-&gt;Priority )
                        T = SingleRotateWithLeft( T );
                    else
                        T = SingleRotateWithRight( T );

                    if( T != NullNode )    /* Continue on down */
                        T = Remove( Item, T );
                    else
                    {
                        /* At a leaf */
                        free( T-&gt;Left );
                        T-&gt;Left = NullNode;
                    }
                }
            }
            return T;
        }
/* END */

        ElementType
        Retrieve( Position P )
        {
            return P-&gt;Element;
        }
```

```
#include "treap.h"
#include &lt;stdio.h&gt;

#define NumItems 12000

main( )
{

    Treap T;
    Position P;
    int i;
    int j = 0;

    T = Initialize( );
    T = MakeEmpty( NullNode );
    for( i = 0; i &lt; NumItems; i++, j = ( j + 7 ) % NumItems )
        T = Insert( j, T );
    for( i = 0; i &lt; NumItems; i++ )
        if( ( P = Find( i, T ) ) == NullNode || Retrieve( P ) != i )
            printf( "Error1 at %d\n", i );

    for( i = 0; i &lt; NumItems; i += 2 )
        T = Remove( i, T );

    for( i = 1; i &lt; NumItems; i += 2 )
        if( ( P = Find( i, T ) ) == NullNode || Retrieve( P ) != i )
           printf( "Error2 at %d\n", i );

    for( i = 0; i &lt; NumItems; i += 2 )
        if( ( P = Find( i, T ) ) != NullNode )
            printf( "Error3 at %d\n", i );
    printf( "Min is %d, Max is %d\n", Retrieve( FindMin( T ) ),
               Retrieve( FindMax( T ) ) );

    return 0;
}
```



```
        #include &lt;stdlib.h&gt;
```

```
        #include "fatal.h"

        typedef int ElementType;
        #define NegInfinity (-10000)

        #ifndef _RedBlack_H
        #define _RedBlack_H

        struct RedBlackNode;
        typedef struct RedBlackNode *Position;
        typedef struct RedBlackNode *RedBlackTree;

        RedBlackTree MakeEmpty( RedBlackTree T );
        Position Find( ElementType X, RedBlackTree T );
        Position FindMin( RedBlackTree T );
        Position FindMax( RedBlackTree T );
        RedBlackTree Initialize( void );
        RedBlackTree Insert( ElementType X, RedBlackTree T );
        RedBlackTree Remove( ElementType X, RedBlackTree T );
        ElementType Retrieve( Position P );
        void PrintTree( RedBlackTree T );

        #endif  /* _RedBlack_H */

/* END */
```



```
#include "redblack.h"
#include &lt;stdlib.h&gt;
#include "fatal.h"


/* START: fig12_14.txt */
        typedef enum ColorType { Red, Black } ColorType;

        struct RedBlackNode
        {
            ElementType  Element;
```

```
            ElementType  Element;
            RedBlackTree Left;
            RedBlackTree Right;
            ColorType    Color;
        };

        static Position NullNode = NULL;  /* Needs initialization */

        /* Initialization procedure */
        RedBlackTree
        Initialize( void )
        {
            RedBlackTree T;

            if( NullNode == NULL )
            {
                NullNode = malloc( sizeof( struct RedBlackNode ) );
                if( NullNode == NULL )
                    FatalError( "Out of space!!!" );
                NullNode-&gt;Left = NullNode-&gt;Right = NullNode;
                NullNode-&gt;Color = Black;
                NullNode-&gt;Element = 12345;
            }

            /* Create the header node */
            T = malloc( sizeof( struct RedBlackNode ) );
            if( T == NULL )
                FatalError( "Out of space!!!" );
            T-&gt;Element = NegInfinity;
            T-&gt;Left = T-&gt;Right = NullNode;
            T-&gt;Color = Black;

            return T;
        }
/* END */

        void
        Output( ElementType Element )
        {
            printf( "%d\n", Element );
        }

/* START: fig12_13.txt */
        /* Print the tree, watch out for NullNode, */
        /* and skip header */
```



```
        static void
        DoPrint( RedBlackTree T )
        {
            if( T != NullNode )
            {
                DoPrint( T-&gt;Left );
                Output( T-&gt;Element );
                DoPrint( T-&gt;Right );
            }
        }

        void
        PrintTree( RedBlackTree T )
        {
            DoPrint( T-&gt;Right );
        }
/* END */

        static RedBlackTree
        MakeEmptyRec( RedBlackTree T )
```

```
{
    if( T != NullNode )
    {
        MakeEmptyRec( T-&gt;Left );
        MakeEmptyRec( T-&gt;Right );
        free( T );
    }
    return NullNode;
}

RedBlackTree
MakeEmpty( RedBlackTree T )
{
    T-&gt;Right = MakeEmptyRec( T-&gt;Right );
    return T;
}

Position
Find( ElementType X, RedBlackTree T )
{
    if( T == NullNode )
        return NullNode;
    if( X &lt; T-&gt;Element )
        return Find( X, T-&gt;Left );
    else
    if( X &gt; T-&gt;Element )
        return Find( X, T-&gt;Right );
    else
        return T;
}

Position
FindMin( RedBlackTree T )
{
    T = T-&gt;Right;
    while( T-&gt;Left != NullNode )
```



```
        T = T-&gt;Left;

    return T;
}

Position
FindMax( RedBlackTree T )
{
    while( T-&gt;Right != NullNode )
        T = T-&gt;Right;

    return T;
}

/* 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-&gt;Left;
    K2-&gt;Left = K1-&gt;Right;
    K1-&gt;Right = K2;

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

```
        /* 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-&gt;Right;
            K1-&gt;Right = K2-&gt;Left;
            K2-&gt;Left = K1;

            return K2;  /* New root */
        }

/* START: fig12_15.txt */
        /* Perform a rotation at node X */
        /* (whose parent is passed as a parameter) */
        /* The child is deduced by examining Item */

        static Position
        Rotate( ElementType Item, Position Parent )
        {

            if( Item &lt; Parent-&gt;Element )
```



```
                return Parent-&gt;Left = Item &lt; Parent-&gt;Left-&gt;Element ?
                    SingleRotateWithLeft( Parent-&gt;Left ) :
                    SingleRotateWithRight( Parent-&gt;Left );
            else
                return Parent-&gt;Right = Item &lt; Parent-&gt;Right-&gt;Element ?
                    SingleRotateWithLeft( Parent-&gt;Right ) :
                    SingleRotateWithRight( Parent-&gt;Right );
        }
/* END */

/* START: fig12_16.txt */
        static Position X, P, GP, GGP;

        static
        void HandleReorient( ElementType Item, RedBlackTree T )
        {
            X-&gt;Color = Red;        /* Do the color flip */
            X-&gt;Left-&gt;Color = Black;
            X-&gt;Right-&gt;Color = Black;

            if( P-&gt;Color == Red )  /* Have to rotate */
            {
                GP-&gt;Color = Red;
                if( (Item &lt; GP-&gt;Element) != (Item &lt; P-&gt;Element) )
                    P = Rotate( Item, GP );  /* Start double rotate */
                X = Rotate( Item, GGP );
                X-&gt;Color = Black;
            }
            T-&gt;Right-&gt;Color = Black;  /* Make root black */
        }

        RedBlackTree
        Insert( ElementType Item, RedBlackTree T )
        {
            X = P = GP = T;
            NullNode-&gt;Element = Item;
            while( X-&gt;Element != Item )  /* Descend down the tree */
            {
```

```
            GGP = GP; GP = P; P = X;
            if( Item &lt; X-&gt;Element )
                X = X-&gt;Left;
            else
                X = X-&gt;Right;
            if( X-&gt;Left-&gt;Color == Red &amp;&amp; X-&gt;Right-&gt;Color == Red )
                HandleReorient( Item, T );
        }

        if( X != NullNode )
            return NullNode;  /* Duplicate */

        X = malloc( sizeof( struct RedBlackNode ) );
        if( X == NULL )
            FatalError( "Out of space!!!" );
        X-&gt;Element = Item;
        X-&gt;Left = X-&gt;Right = NullNode;
```



```
        if( Item &lt; P-&gt;Element )  /* Attach to its parent */
            P-&gt;Left = X;
        else
            P-&gt;Right = X;
        HandleReorient( Item, T ); /* Color it red; maybe rotate */

        return T;
    }
/* END */

    RedBlackTree
    Remove( ElementType Item, RedBlackTree T )
    {
        printf( "Remove is unimplemented\n" );
        if( Item )
            return T;
        return T;
    }

    ElementType
    Retrieve( Position P )
    {
        return P-&gt;Element;
    }
```

```
#include "redblack.h"
#include &lt;stdio.h&gt;

#define N 800

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

    T = Initialize( );
    T = MakeEmpty( T );

    for( i = 0; i &lt; N; i++, j = ( j + 7 ) % N )
        T = Insert( j, T );
    printf( "Inserts are complete\n" );



    for( i = 0; i &lt; N; i++ )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );


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

    return 0;
}
```

```
        #include &lt;stdio.h&gt;
        #include &lt;stdlib.h&gt;
        #include "fatal.h"

        typedef int ElementType;
        typedef ElementType ItemType[ 2 ];

        struct KdNode;
        typedef struct KdNode *Position;
        typedef struct KdNode *KdTree;

        struct KdNode
        {
            ItemType Data;
            KdTree   Left;
            KdTree   Right;
        };

/* START: fig12_43.txt */
        static KdTree
        RecursiveInsert( ItemType Item, KdTree T, int Level )
        {
            if( T == NULL )
            {
                T = malloc( sizeof( struct KdNode ) );
                if( T == NULL )
                    FatalError( "Out of space!!!" );
                T-&gt;Left = T-&gt;Right = NULL;
                T-&gt;Data[ 0 ] = Item[ 0 ];
                T-&gt;Data[ 1 ] = Item[ 1 ];
            }
            else
            if( Item[ Level ] &lt; T-&gt;Data[ Level ] )
                T-&gt;Left = RecursiveInsert( Item, T-&gt;Left, !Level );
            else
                T-&gt;Right = RecursiveInsert( Item, T-&gt;Right, !Level );
            return T;
        }

        KdTree
        Insert( ItemType Item, KdTree T )
        {
            return RecursiveInsert( Item, T, 0 );
        }
/* END */

/* START: fig12_44.txt */
        /* Print items satisfying */
        /* Low[ 0 ] &lt;= Item[ 0 ] &lt;= High[ 0 ] and */
        /* Low[ 1 ] &lt;= Item[ 1 ] &lt;= High[ 1 ] */

        static void
        RecPrintRange( ItemType Low, ItemType High,
                       KdTree T, int Level )
        {
            if( T != NULL )
```

```
            {
                if( Low[ 0 ] &lt;= T-&gt;Data[ 0 ] &amp;&amp;
                                T-&gt;Data[ 0 ] &lt;= High[ 0 ] &amp;&amp;
                                Low[ 1 ] &lt;= T-&gt;Data[ 1 ] &amp;&amp;
                                T-&gt;Data[ 1 ] &lt;= High[ 1 ] )
                    printf( "(%d,%d)\n",
                                T-&gt;Data[ 0 ], T-&gt;Data[ 1 ] );

                if( Low[ Level ] &lt;= T-&gt;Data[ Level ] )
                    RecPrintRange( Low, High, T-&gt;Left, !Level );
                if( High[ Level ] &gt;= T-&gt;Data[ Level ] )
                    RecPrintRange( Low, High, T-&gt;Right, !Level );
            }
        }

        void
        PrintRange( ItemType Low, ItemType High, KdTree T )
        {
            RecPrintRange( Low, High, T, 0 );
        }
/* END */

main( )
{
    KdTree T;
    ItemType It, L, H;
    int i;

    printf( "Starting program\n" );

    T = NULL;
    for( i = 300; i &lt; 370; i++ )
    {
        It[ 0 ] = i; It[ 1 ] = 2500 - i;
        T = Insert( It, T );
    }

    printf( "Insertions complete\n" );

    i = 1;
    L[ 0 ] = 70; L[ 1 ] = 2186; H[ 0 ] = 1200; H[ 1 ] = 2200;
    PrintRange( L, H, T );

    printf( "Done...\n" ) ;
    return 0;
}
```



```
        typedef int ElementType;

        #ifndef _PairHeap_H
        #define _PairHeap_H

        struct PairNode;
        typedef struct PairNode *PairHeap;
        typedef struct PairNode *Position;
```

```
PairHeap Initialize( void );
void Destroy( PairHeap H );
PairHeap MakeEmpty( PairHeap H );
PairHeap Insert( ElementType Item, PairHeap H, Position *Loc );
PairHeap DeleteMin( ElementType *MinItem, PairHeap H );
ElementType FindMin( PairHeap H );
PairHeap DecreaseKey( Position P,
                      ElementType NewVal, PairHeap H );
int IsEmpty( PairHeap H );
int IsFull( PairHeap H );
#endif
```



```
#include "pairheap.h"
#include "fatal.h"
#include &lt;stdlib.h&gt;

struct PairNode
{
    ElementType Element;
    Position    LeftChild;
    Position    NextSibling;
    Position    Prev;
};

#define MaxSiblings 1000

Position CompareAndLink( Position First, Position Second );
PairHeap CombineSiblings( Position FirstSibling );


PairHeap
```

```
        PairHeap
        Initialize( void )
        {
            return NULL;
        }

        PairHeap
        MakeEmpty( PairHeap H )
        {
            if( H != NULL )
            {
                MakeEmpty( H-&gt;LeftChild );
                MakeEmpty( H-&gt;NextSibling );
                free( H );
            }
            return NULL;
        }

/* START: fig12_54.txt */
        /* Insert Item into pairing heap H */
        /* Return resulting pairing heap */
        /* A pointer to the newly allocated node */
        /* is passed back by reference and accessed as *Loc */

        PairHeap
        Insert( ElementType Item, PairHeap H, Position *Loc )
        {
            Position NewNode;

            NewNode = malloc( sizeof( struct PairNode ) );
            if( NewNode == NULL )
                FatalError( "Out of space!!!" );
            NewNode-&gt;Element = Item;
            NewNode-&gt;LeftChild = NewNode-&gt;NextSibling = NULL;
            NewNode-&gt;Prev = NULL;

            *Loc = NewNode;
            if( H == NULL )
```



```
                return NewNode;
            else
                return CompareAndLink( H, NewNode );
        }


        /* Lower item in Position P by Delta */

        PairHeap
        DecreaseKey( Position P, ElementType Delta, PairHeap H )
        {
            if( Delta &lt; 0 )
                Error( "DecreaseKey called with negative Delta" );

            P-&gt;Element -= Delta;
            if( P == H )
                return H;

            if( P-&gt;NextSibling != NULL )
                P-&gt;NextSibling-&gt;Prev = P-&gt;Prev;
            if( P-&gt;Prev-&gt;LeftChild == P )
                P-&gt;Prev-&gt;LeftChild = P-&gt;NextSibling;
            else
                P-&gt;Prev-&gt;NextSibling = P-&gt;NextSibling;

            P-&gt;NextSibling = NULL;
            return CompareAndLink( H, P );
        }
```

```
/* END */


/* START: fig12_55.txt */
        PairHeap
        DeleteMin( ElementType *MinItem, PairHeap H )
        {
            Position NewRoot = NULL;

            if( IsEmpty( H ) )
                Error( "Pairing heap is empty!" );
            else
            {
                *MinItem = H-&gt;Element;
                if( H-&gt;LeftChild != NULL )
                    NewRoot = CombineSiblings( H-&gt;LeftChild );
                free( H );
            }
            return NewRoot;
        }
/* END */

/* START: fig12_53.txt */
        /* This is the basic operation to maintain order */
        /* Links First and Second together to satisfy heap order */
        /* Returns the resulting tree */
        /* First is assumed NOT NULL */
        /* First-&gt;NextSibling  MUST be NULL on entry */
```



```
        Position
        CompareAndLink( Position First, Position Second )
        {
            if( Second == NULL )
                return First;
            else
            if( First-&gt;Element &lt;= Second-&gt;Element )
            {
                /* Attach Second as the leftmost child of First */
                Second-&gt;Prev = First;
                First-&gt;NextSibling = Second-&gt;NextSibling;
                if( First-&gt;NextSibling != NULL )
                    First-&gt;NextSibling-&gt;Prev = First;
                Second-&gt;NextSibling = First-&gt;LeftChild;
                if( Second-&gt;NextSibling != NULL )
                    Second-&gt;NextSibling-&gt;Prev = Second;
                First-&gt;LeftChild = Second;
                return First;
            }
            else
            {
                /* Attach First as the leftmost child of Second */
                Second-&gt;Prev = First-&gt;Prev;
                First-&gt;Prev = Second;
                First-&gt;NextSibling = Second-&gt;LeftChild;
                if( First-&gt;NextSibling != NULL )
                    First-&gt;NextSibling-&gt;Prev = First;
                Second-&gt;LeftChild = First;
                return Second;
            }
        }
/* END */

/* START: fig12_56.txt */
        /* Assumes FirstSibling is NOT NULL */
```

```
        PairHeap
        CombineSiblings( Position FirstSibling )
        {
            static Position TreeArray[ MaxSiblings ];
            int i, j, NumSiblings;

            /* If only one tree, return it */
            if( FirstSibling-&gt;NextSibling == NULL )
                return FirstSibling;

            /* Place each subtree in TreeArray */
            for( NumSiblings = 0; FirstSibling != NULL; NumSiblings++ )
            {
                TreeArray[ NumSiblings ] = FirstSibling;
                FirstSibling-&gt;Prev-&gt;NextSibling = NULL; /* Break links */
                FirstSibling = FirstSibling-&gt;NextSibling;
            }
            TreeArray[ NumSiblings ] = NULL;
```



```
            /* Combine the subtrees two at a time, */
            /* going left to right */
            for( i = 0; i + 1 &lt; NumSiblings; i += 2 )
                TreeArray[ i ] = CompareAndLink(
                        TreeArray[ i ], TreeArray[ i + 1 ] );

            /* j has the result of the last CompareAndLink */
            /* If an odd number of trees, get the last one */
            j = i - 2;
            if( j == NumSiblings - 3 )
                TreeArray[ j ] = CompareAndLink(
                        TreeArray[ j ], TreeArray[ j + 2 ] );

            /* Now go right to left, merging last tree with */
            /* next to last. The result becomes the new last */
            for( ; j &gt;= 2; j -= 2 )
                TreeArray[ j - 2 ] = CompareAndLink(
                        TreeArray[ j - 2 ],  TreeArray[ j ] );

            return TreeArray[ 0 ];
        }
/* END */

        ElementType
        FindMin( PairHeap H )
        {
            if( !IsEmpty( H ) )
                return H-&gt;Element;
            Error( "Priority Queue is Empty" );
            return 0;
        }

        int
        IsEmpty( PairHeap H )
        {
            return H == NULL;
        }

        int
        IsFull( PairHeap H )
        {
            return 0;   /* Never full */
        }

        void
        Destroy( PairHeap H )
        {
```

```
        {
            MakeEmpty( H );
        }
```



```
#include "pairheap.h"
#include &lt;stdio.h&gt;

void
sleep( int x )
{
    int i, j, k, m;

    for( i = 0; i &lt; 10000; i++ )
        for( j = 0; j &lt; 1000; j++ )
            for( k = 0; k &lt; x; k++ )
                m++;
    printf( "Done sleeping!! %d", m );
}

#define MaxSize 500

main( )
{
    PairHeap H;
    Position P[ MaxSize ];
    int i, j;
    int AnItem;

    H = Initialize( );
    for( i=0, j=MaxSize/2; i&lt;MaxSize; i++, j=( j+71)%MaxSize )
        H = Insert( j + MaxSize, H, &amp;P[ j ] );

    printf( "Done inserting\n" );

    for( i = 0, j = MaxSize / 2; i &lt;MaxSize; i++, j=(j+51)%MaxSize )
        H = DecreaseKey( P[ j ], MaxSize, H );

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

    return 0;
}
```

</pre><embed id="xunlei_com_thunder_helper_plugin_d462f475-c18e-46be-bd10-327458d045bd"
type="application/thunder_download_plugin" height="0" width="0"></body></html>