1. An ordered list is a sequence of items in some order. The list presented in this discussion contains a sequence of integers in ascending order.
2. A sentinel is employed to make the insertion and removal of a value simpler. The sentinel is the minimum value (*MsMin*) and is placed at the beginning of an ascending sequence.
3. An element in the list can be found using a binary search. A key is given to the function performing the search and the index of the matching key is returned. If the list does not contain an element that matches the key, an invalid index is returned, usually zero (0).
4. A value is inserted in the list by shifting elements larger than the input value toward the end of the list.
5. A value is removed from the list by removing the element and shifting elements larger than the element removed one position toward the front of the list.
6. Support for iterating through the list is implemented by adding member *cursor* that records the current position in the anchor for the list.
7. Member data for **class** *List* are shown in Figure 1.
   - 7.1. Member *L* points to a dynamically allocated array used to store the integers in the list and the sentinel.
   - 7.2. Member *size* records the number of elements available to store integers on the list. Member *size* includes the storage allocated for the sentinel.
   - 7.3. Member *count* records the number of integers currently stored in the list. Member *count* does not include the sentinel.
   - 7.4. Member *cursor* records the current position.
   - 7.5. Member *MsMin* is the value assigned to the sentinel. Member *MsMin* is initialized to the smallest integer *INT_MIN*. Value *INT_MIN* is found in **#include** <*limits.h*>.
   - 7.6. Function *Index* returns the index of parameter *key* or zero if the key is not in the list.
   - 7.7. Constructor *List***(int)** initializes **class** *List*, allocates storage for the list, and assign the sentinel to element ***L*[0]**.
   - 7.8. Destructor ***~List*()** reclaims storage allocated by the constructor.
   - 7.9. The default constructor for class *ListFullException* is called when no more space is available to insert another key.
   - 7.10. The default constructor for class ListRangeException is called when the cursor is outside the valid range $1 \le cursor \le count$.

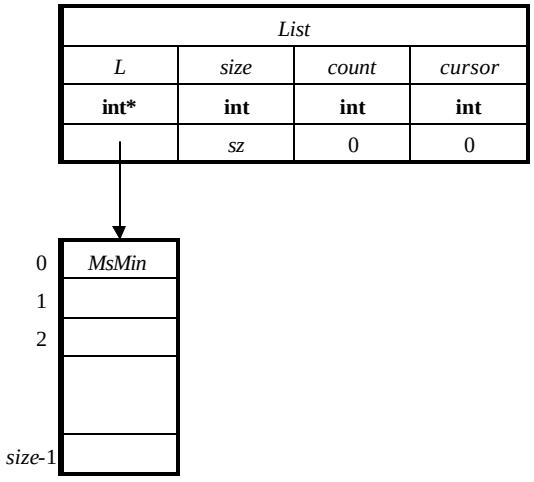

Figure 1. Member data for **class** *List*.

```
class List {
      int size;
      int* L;
      int count;
      int cursor;
      const int MsMin;
      int Index(int key);
public:
      List(int sz=100);
      ~List();
      class ListFullException{};
      class ListRangeException{};
      bool IsFull(void);
      void Insert(int key);
      void Remove(int key);
      void First(void);
      bool IsEol(void);
      void Next(void);
      int ElementValue(void);
      bool IsMember(int key);
};
```

Figure 2. class *List*.

   - 7.11. Function *IsFull* determines if there is space for another key.
   - 7.12. Function Insert inserts a unique *key* into the list.
   - 7.13. Function Remove deletes a *key* from the list. If the *key* is not in the list no action is taken.

7.14. Function *First* puts the cursor on the smallest element in the list.
7.15. Function *IsEol* determines if the cursor is past the last element in the list.
7.16. Function *Next* puts the cursor on the next largest element on the list.
7.17. Function *ElementValue* returns the value of the current element as determined by the cursor.
7.18. Function *IsMember* determines if parameter *key* is a member of the list.

```
List::List(int sz)
      :size(sz),count(0),cursor(0)
      ,MsMin(INT_MIN)
{     L=new int[size]; }
```

Figure 3. Constructor *List*::*List*(**int** *sz*)

```
List::~List(){ if (L) delete[] L; }
```

Figure 4. Destructor *List*::~*List*()

void *List*::*Insert*(**int** *key*)
Discussion: Keys are shifted to make room for the new key as shown in Figure 5.

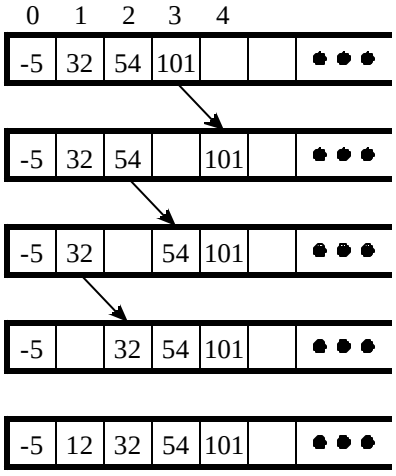

Figure 5. Insert key 12.

Code to insert the second and subsequent keys is given in Figure 6.

```
int i=count++;
while (key < L[i-1]) {
      L[i]=L[i-1];
      i--;
}
L[i]=key;
```

Figure 6. Insert 2nd and subsequent elements.

Each row of Figure 7 represents one iteration of the *while-statement* in Figure 6.

i	*i*-1	*key*	*L*[*i*-1]	*key* < *L*[*i*-1]	*i*--
4	3	12	101	yes	3
3	2	12	54	yes	2
2	1	12	32	yes	1
1	0	12	-5	no	

Figure 7.

To insert the first key requires that the while-test be modified to include the guard shown in Figure 8.

```
int i=count++;
while (i>0 && key < L[i-1]) {
      L[i]=L[i-1];
      i--;
}
L[i]=key;
```

Figure 8. Insert with a guard

The guard can be removed by placing a sentinel at the beginning of the list. A value smaller than any key is placed in element zero of array *L*. The shaded rectangle in element zero represents the sentinel containing a smaller value than any key in the list as shown in Figure 9.

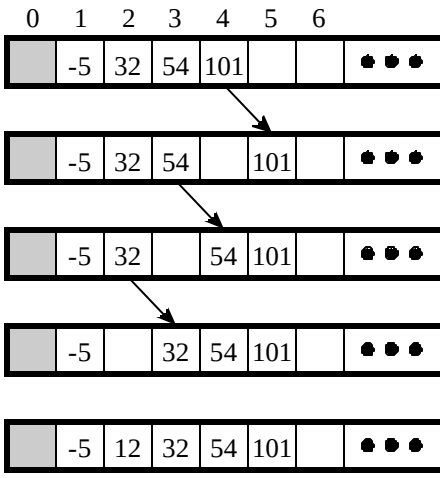

Figure 9. Insert key 12 in the list with a sentinel

Code to insert a key in a list containing a sentinel is given in Figure 10. The only difference between the code in Figure 10 and the code in Figure 6 is that member count is pre-incremented rather than post-incremented.

```
void List::Insert(int key)
{    int i=++count;
     while (key < L[i-1]) {
          L[i]=L[i-1];
          i--;
     }
     L[i]=key;
}
```

Figure 10. Insert with a sentinel

Now that we have the basic algorithm, we must guard against certain boundary conditions: These include:

1. Duplicate keys
2. Overfilling the array

Member function *IsMember* can be used to determine if a key exists. The code in Figure 11 contains the guard against inserting a duplicate key.

```
void List::Insert(int key)
{    if (IsMember(key)) return;
     int i=++count;
     while (key < L[i-1]) {
          L[i]=L[i-1];
          i--;
     }
     L[i]=key;
}
```

Figure 11. Insert with guard against duplicate keys

Member function *IsFull* is used to prevent overfilling the array containing keys as shown in Figure 12.

```
void List::Insert(int key)
{    if (IsMember(key)) return;
     if (IsFull()) throw ListFullException();
     int i=++count;
     while (key < L[i-1]) {
          L[i]=L[i-1];
          i--;
     }
     L[i]=key;
}
```

Figure 12. Insert with guard against overfilling array *L*.