List Implementation Code

CSE 225 - Data Structures and Algorithms

Md. Mahfuzur Rahman
ECE Department
North South University

1 Unsorted List Implementation

1.1 Summary

```
ItemType
  ItemType.h     Specification for items on the list
  ItemType.cpp     Implementation file for items on the list

Array Based Implementation
  unsorted.h     Specification file for UnsortedType class
  unsorted.cpp     Implementation file for UnsortedType class

Linked-List Based Implementation
  unsorted.h     Specification file for UnsortedType class
  unsorted.cpp     Implementation file for UnsortedType class
```

1.2 ItemType

```

// The following declarations and definitions go into file
// ItemType.h.

#include <fstream>
const int MAX_ITEMS = 5;
enum RelationType   {LESS, GREATER, EQUAL};

class ItemType
{
public:
  ItemType();
  RelationType ComparedTo(ItemType) const;
  void Print(std::ostream&) const;
  void Initialize(int number);
private:
  int value;
};
```

```
// The following definitions go into file ItemType.cpp.
#include <fstream>
#include <iostream>
#include "ItemType.h"

ItemType::ItemType()
{
  value = 0;
}
```

```
10
11 RelationType ItemType::ComparedTo(ItemType otherItem) const
12 {
13   if (value < otherItem.value)
14     return LESS;
15   else if (value > otherItem.value)
16     return GREATER;
17   else return EQUAL;
18 }
19
20 void ItemType::Initialize(int number)
21 {
22   value = number;
23 }
24
25 void ItemType::Print(std::ostream& out) const
26 // pre:  out has been opened.
27 // post: value has been sent to the stream out.
28 {
29   out << value;
30 }
```

1.3 Array Based Implementation

```
1 #include "ItemType.h"
2 // File ItemType.h must be provided by the user of this class.
3 //  ItemType.h must contain the following definitions:
4 //  MAX_ITEMS:     the maximum number of items on the list
5 //  ItemType:      the definition of the objects on the list
6 //  RelationType:  {LESS, GREATER, EQUAL}
7 //  Member function ComparedTo(ItemType item) which returns
8 //       LESS, if self "comes before" item
9 //       GREATER, if self "comes after" item
10 //      EQUAL, if self and item are the same
11
12 class UnsortedType
13 {
14 public:
15   UnsortedType();
16   // Constructor
17
18   void MakeEmpty();
19   // Function: Returns the list to the empty state.
20   // Post:  List is empty.
21
22   bool IsFull() const;
23   // Function:  Determines whether list is full.
24   // Pre:  List has been initialized.
25   // Post: Function value = (list is full)
26
27   int GetLength() const;
28   // Function: Determines the number of elements in list.
29   // Pre:  List has been initialized.
```

```
30    // Post: Function value = number of elements in list
31
32    ItemType GetItem(ItemType, bool&);
33    // Function: Retrieves list element whose key matches item's key (if
34    //           present).
35    // Pre:  List has been initialized.
36    //       Key member of item is initialized.
37    // Post: If there is an element someItem whose key matches
38    //       item's key, then found = true and someItem is returned.
39    //   otherwise found = false and item is returned.
40    //       List is unchanged.
41
42    void PutItem(ItemType item);
43    // Function: Adds item to list.
44    // Pre:  List has been initialized.
45    //       List is not full.
46    //       item is not in list.
47    // Post: item is in list.
48
49    void DeleteItem(ItemType item);
50    // Function: Deletes the element whose key matches item's key.
51    // Pre:  List has been initialized.
52    //       Key member of item is initialized.
53    //       One and only one element in list has a key matching item's key.
54    // Post: No element in list has a key matching item's key.
55
56    void ResetList();
57    // Function: Initializes current position for an iteration through the
        list.
58    // Pre:  List has been initialized.
59    // Post: Current position is prior to list.
60
61    ItemType GetNextItem();
62    // Function: Gets the next element in list.
63    // Pre:  List has been initialized and has not been changed since last
        call.
64    //       Current position is defined.
65    //       Element at current position is not last in list.
66    //
67    // Post: Current position is updated to next position.
68    //       item is a copy of element at current position.
69
70  private:
71    int length;
72    ItemType info[MAX_ITEMS];
73    int currentPos;
74  };
```

```
1   // Implementation file for Unsorted.h
2
3   #include "unsorted.h"
4
5   UnsortedType::UnsortedType()
6   {
```

```
7   length = 0;
8 }
9 bool UnsortedType::IsFull() const
10 {
11   return (length == MAX_ITEMS);
12 }
13 int UnsortedType::GetLength() const
14 {
15   return length;
16 }
17
18 ItemType UnsortedType::GetItem(ItemType item, bool& found)
19 // Pre:  Key member(s) of item is initialized.
20 // Post: If found, item's key matches an element's key in the
21 //       list and a copy of that element has been returned;
22 //       otherwise, item is returned.
23 {
24   bool moreToSearch;
25   int location = 0;
26   found = false;
27
28   moreToSearch = (location < length);
29
30   while (moreToSearch && !found)
31   {
32     switch (item.ComparedTo(info[location]))
33     {
34       case LESS    :
35       case GREATER : location++;
36                      moreToSearch = (location < length);
37                      break;
38       case EQUAL   : found = true;
39                      item = info[location];
40                      break;
41     }
42   }
43   return item;
44 }
45 void UnsortedType::MakeEmpty()
46 // Post: list is empty.
47 {
48   length = 0;
49 }
50 void UnsortedType::PutItem(ItemType item)
51 // Post: item is in the list.
52 {
53   info[length] = item;
54   length++;
55 }
56 void UnsortedType::DeleteItem(ItemType item)
57 // Pre:  item's key has been initialized.
58 //       An element in the list has a key that matches item's.
59 // Post: No element in the list has a key that matches item's.
60 {
```

```
61    int location = 0;
62
63    while (item.ComparedTo(info[location]) != EQUAL)
64      location++;
65
66    info[location] = info[length - 1];
67    length--;
68  }
69  void UnsortedType::ResetList()
70  // Post: currentPos has been initialized.
71  {
72    currentPos = -1;
73  }
74
75  ItemType UnsortedType::GetNextItem()
76  // Pre:  ResetList was called to initialized iteration.
77  //       No transformer has been executed since last call.
78  //       currentPos is defined.
79  // Post: item is current item.
80  //       Current position has been updated.
81  {
82    currentPos++;
83    return info[currentPos];
84  }
```

1.4 Linked-List Based Implementation

```
1
2  #include "ItemType.h"
3  // File ItemType.h must be provided by the user of this class.
4  //  ItemType.h must contain the following definitions:
5  //  MAX_ITEMS:     the maximum number of items on the list
6  //  ItemType:      the definition of the objects on the list
7  //  RelationType:  {LESS, GREATER, EQUAL}
8  //  Member function ComparedTo(ItemType item) which returns
9  //       LESS, if self "comes before" item
10 //       GREATER, if self "comes after" item
11 //       EQUAL, if self and item are the same
12 struct NodeType;
13
14 class UnsortedType
15 {
16 public:
17   UnsortedType(); // Constructor
18   ~UnsortedType(); // Destructor
19   void MakeEmpty();
20   // Function: Returns the list to the empty state.
21   // Post:  List is empty.
22   bool IsFull() const;
23   // Function:  Determines whether list is full.
24   // Pre:  List has been initialized.
25   // Post: Function value = (list is full)
26
```

```
27    int GetLength() const;
28    // Function: Determines the number of elements in list.
29    // Pre:  List has been initialized.
30    // Post: Function value = number of elements in list
31
32    ItemType GetItem(ItemType& item, bool& found);
33    // Function: Retrieves list element whose key matches item's key (if
34    //           present).
35    // Pre:  List has been initialized.
36    //       Key member of item is initialized.
37    // Post: If there is an element someItem whose key matches
38    //       item's key, then found = true and someItem is returned;
39    //   otherwise found = false and item is returned.
40    //       List is unchanged.
41
42    void PutItem(ItemType item);
43    // Function: Adds item to list.
44    // Pre:  List has been initialized.
45    //       List is not full.
46    //       item is not in list.
47    // Post: item is in list.
48
49    void DeleteItem(ItemType item);
50    // Function: Deletes the element whose key matches item's key.
51    // Pre:  List has been initialized.
52    //       Key member of item is initialized.
53    //       One and only one element in list has a key matching item's key.
54    // Post: No element in list has a key matching item's key.
55
56    void ResetList();
57    // Function: Initializes current position for an iteration through the
        list.
58    // Pre:  List has been initialized.
59    // Post: Current position is prior to list.
60
61    ItemType GetNextItem();
62    // Function: Gets the next element in list.
63    // Pre:  List has been initialized and has not been changed since last
        call.
64    //       Current position is defined.
65    //       Element at current position is not last in list.
66    //
67    // Post: Current position is updated to next position.
68    //       item is a copy of element at current position.
69
70  private:
71    NodeType* listData;
72    int length;
73    NodeType* currentPos;
74  };
```

```
1   // This file contains the linked implementation of class
2   // UnsortedType.
3
```

```
4  #include "unsorted.h"
5  struct NodeType
6  {
7      ItemType info;
8      NodeType* next;
9  };
10
11 UnsortedType::UnsortedType()  // Class constructor
12 {
13   length = 0;
14   listData = NULL;
15 }
16 bool UnsortedType::IsFull() const
17 // Returns true if there is no room for another ItemType
18 //  on the free store; false otherwise.
19 {
20   NodeType* location;
21   try
22   {
23     location = new NodeType;
24     delete location;
25     return false;
26   }
27   catch(std::bad_alloc exception)
28   {
29     return true;
30   }
31 }
32
33 int UnsortedType::GetLength() const
34 // Post: Number of items in the list is returned.
35 {
36   return length;
37 }
38
39 void UnsortedType::MakeEmpty()
40 // Post: List is empty; all items have been deallocated.
41 {
42   NodeType* tempPtr;
43
44     while (listData != NULL)
45     {
46       tempPtr = listData;
47       listData = listData->next;
48       delete tempPtr;
49   }
50   length = 0;
51 }
52 void UnsortedType::PutItem(ItemType item)
53 // item is in the list; length has been incremented.
54 {
55   NodeType* location;      // Declare a pointer to a node
56
57   location = new NodeType;     // Get a new node
```

```
58    location->info = item;     // Store the item in the node
59    location->next = listData;  // Store address of first node
60                //   in next field of new node
61    listData = location;     // Store address of new node into
62                //   external pointer
63    length++;        // Increment length of the list
64  }
65
66  ItemType UnsortedType::GetItem(ItemType& item, bool& found)
67  // Pre:  Key member(s) of item is initialized.
68  // Post: If found, item's key matches an element's key in the
69  //       list and a copy of that element has been stored in item;
70  //       otherwise, item is unchanged.
71  {
72    bool moreToSearch;
73    NodeType* location;
74
75    location = listData;
76    found = false;
77    moreToSearch = (location != NULL);
78
79    while (moreToSearch && !found)
80    {
81      switch (item.ComparedTo(location->info))
82      {
83        case LESS    :
84        case GREATER : location = location->next;
85                       moreToSearch = (location != NULL);
86                       break;
87        case EQUAL   : found = true;
88                       item = location->info;
89                       break;
90      }
91    }
92    return item;
93  }
94
95   void UnsortedType::DeleteItem(ItemType item)
96  // Pre:  item's key has been initialized.
97  //       An element in the list has a key that matches item's.
98  // Post: No element in the list has a key that matches item's.
99  {
100   NodeType* location = listData;
101   NodeType* tempLocation;
102
103   // Locate node to be deleted.
104   if (item.ComparedTo(listData->info) == EQUAL)
105   {
106     tempLocation = location;
107     listData = listData->next;     // Delete first node.
108   }
109   else
110   {
111     while (item.ComparedTo((location->next)->info) != EQUAL)
```

```
112       location = location->next;
113
114     // Delete node at location->next
115     tempLocation = location->next;
116     location->next = (location->next)->next;
117   }
118   delete tempLocation;
119   length--;
120 }
121
122 void UnsortedType::ResetList()
123 // Post: Current position has been initialized.
124 {
125   currentPos = NULL;
126 }
127
128 ItemType UnsortedType::GetNextItem()
129 // Post:  A copy of the next item in the list is returned.
130 //        When the end of the list is reached, currentPos
131 //        is reset to begin again.
132 {
133   ItemType item;
134   if (currentPos == NULL)
135     currentPos = listData;
136   else
137     currentPos = currentPos->next;
138   item = currentPos->info;
139   return item;
140 }
141
142 UnsortedType::~UnsortedType()
143 // Post: List is empty; all items have been deallocated.
144 {
145   NodeType* tempPtr;
146
147   while (listData != NULL)
148   {
149     tempPtr = listData;
150     listData = listData->next;
151     delete tempPtr;
152   }
153 }
```

2 Sorted List Implementation

2.1 Summary

```
ItemType
  ItemType.h     Specification for items on the list
  ItemType.cpp     Implementation file for items on the list

Array Based Implementation
  sorted.h     Specification file for SortedType class
  sorted.cpp     Implementation file for SortedType class

Linked-List Based Implementation
  sortedType.h     Specification file for SortedType class
  sortedType.cpp     Implementation file for SortedType class
```

2.2 ItemType

```

// The following declarations and definitions go into file
// ItemType.h.

#include <fstream>
const int MAX_ITEMS = 5;
enum RelationType  {LESS, GREATER, EQUAL};

class ItemType
{
public:
  ItemType();
  RelationType ComparedTo(ItemType) const;
  void Print(std::ostream&) const;
  void Initialize(int number);
private:
  int value;
};


// The following definitions go into file ItemType.cpp.
#include <fstream>
#include <iostream>
#include "ItemType.h"

ItemType::ItemType()
{
  value = 0;
```

```
}

RelationType ItemType::ComparedTo(ItemType otherItem) const
{
  if (value < otherItem.value)
    return LESS;
  else if (value > otherItem.value)
    return GREATER;
  else return EQUAL;
}

void ItemType::Initialize(int number)
{
  value = number;
}

void ItemType::Print(std::ostream& out) const
// pre:  out has been opened.
// post: value has been sent to the stream out.
{
  out << value;
}
```

2.3 Array Based Implementation

```
#ifndef SORTED
#define SORTED

#include "ItemType.h"
// File ItemType.h must be provided by the user of this class.
//  ItemType.h must contain the following definitions:
//  MAX_ITEMS:     the maximum number of items on the list
//  ItemType:      the definition of the objects on the list
//  RelationType:  {LESS, GREATER, EQUAL}
//  Member function ComparedTo(ItemType item) which returns
//       LESS, if self "comes before" item
//       GREATER, if self "comes after" item
//       EQUAL, if self and item are the same

class SortedType
{
public:
  SortedType();

  void MakeEmtpy();
  // Function: Returns list to the empty state
  // Post:  List is empty.

  bool IsFull() const;
  // Function:  Determines whether list is full.
  // Pre:  List has been initialized.
  // Post: Function value = (list is full)

```

```
29    int GetLength() const;
30    // Function: Determines the number of elements in list.
31    // Pre:  List has been initialized.
32    // Post: Function value = number of elements in list
33
34    ItemType GetItem(ItemType item, bool& found);
35    // Function: Retrieves list element whose key matches item's key (if
36    //           present).
37    // Pre:  List has been initialized.
38    //       Key member of item is initialized.
39    // Post: If there is an element someItem whose key matches
40    //       item's key, then found = true and item is returned;
41    //       someItem; otherwise found = false and item is returned.
42    //       List is unchanged.
43
44    void PutItem(ItemType item);
45    // Function: Adds item to list.
46    // Pre:  List has been initialized.
47    //       List is not full.
48    //       item is not in list.
49    //       List is sorted.
50    // Post: item is in list.
51    //       List is sorted
52
53    void DeleteItem(ItemType item);
54    // Function: Deletes the element whose key matches item's key.
55    // Pre:  List has been initialized.
56    //       Key member of item is initialized.
57    //       One and only one element in list has a key matching item's key.
58    // Post: No element in list has a key matching item's key.
59    //       List is sorted.
60
61    void ResetList();
62    // Function: Initializes current position for an iteration through the
        list.
63    // Pre:  List has been initialized.
64    // Post: Current position is prior to list.
65
66    ItemType GetNextItem();
67    // Function: Gets the next element in list.
68    // Pre:  List has been initialized and has not been changed since last
        call.
69    //       Current position is defined.
70    //       Element at current position is not last in list.
71    //
72    // Post: Current position is updated to next position.
73    //       Returns a copy of element at current position.
74
75    void MakeEmpty();
76    // Function: Make the list empty
77    // Pre:  List has been initialized.
78    // Post: The list is empty
79
80  private:
```

```
81    int length;
82    ItemType info[MAX_ITEMS];
83    int currentPos;
84  };
85  #endif
```

```
1   // Implementation file for sorted.h
2
3   #include "sorted.h"
4   SortedType::SortedType()
5   {
6     length = 0;
7   }
8
9   void SortedType::MakeEmpty()
10  {
11    length = 0;
12  }
13
14
15  bool SortedType::IsFull() const
16  {
17    return (length == MAX_ITEMS);
18  }
19
20  int SortedType::GetLength() const
21  {
22    return length;
23  }
24
25  ItemType SortedType::GetItem(ItemType item, bool& found)
26  {
27    int midPoint;
28    int first = 0;
29    int last = length - 1;
30
31    bool moreToSearch = first <= last;
32    found = false;
33    while (moreToSearch && !found)
34    {
35      midPoint = ( first + last) / 2;
36      switch (item.ComparedTo(info[midPoint]))
37      {
38        case LESS    : last = midPoint - 1;
39                       moreToSearch = first <= last;
40                       break;
41        case GREATER : first = midPoint + 1;
42                       moreToSearch = first <= last;
43                       break;
44        case EQUAL   : found = true;
45                       item = info[midPoint];
46                       break;
47      }
48    }
```

```
49    return item;
50  }
51
52  void SortedType::DeleteItem(ItemType item)
53  {
54    int location = 0;
55
56    while (item.ComparedTo(info[location]) != EQUAL)
57      location++;
58    for (int index = location + 1; index < length; index++)
59      info[index - 1] = info[index];
60    length--;
61  }
62
63  void SortedType::PutItem(ItemType item)
64  {
65    bool moreToSearch;
66    int location = 0;
67
68    moreToSearch = (location < length);
69    while (moreToSearch)
70    {
71      switch (item.ComparedTo(info[location]))
72      {
73        case LESS    : moreToSearch = false;
74                       break;
75        case GREATER : location++;
76                       moreToSearch = (location < length);
77                       break;
78      }
79    }
80    for (int index = length; index > location; index--)
81      info[index] = info[index - 1];
82    info[location] = item;
83    length++;
84  }
85
86  void SortedType::ResetList()
87  // Post: currentPos has been initialized.
88  {
89    currentPos = -1;
90  }
91
92  ItemType SortedType::GetNextItem()
93  // Post: item is current item.
94  //       Current position has been updated.
95  {
96    currentPos++;
97    return info[currentPos];
98  }
```

2.4 Linked-List Based Implementation

```
#include "ItemType.h"
// Header file for Sorted List ADT.
struct NodeType;

class SortedType
{
public:
  SortedType();       // Class constructor
  ~SortedType();      // Class destructor

  bool IsFull() const;
  int  GetLength() const;
  void MakeEmpty();
  ItemType GetItem(ItemType& item, bool& found);
  void PutItem(ItemType item);
  void DeleteItem(ItemType item);
  void ResetList();
  ItemType GetNextItem();

private:
  NodeType* listData;
  int length;
  NodeType* currentPos;
};
```

```
#include "sortedType.h"

struct NodeType
{
  ItemType info;
  NodeType* next;
};

SortedType::SortedType()  // Class constructor
{
  length = 0;
  listData = NULL;
}

bool SortedType::IsFull() const
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
```

```
28 }
29
30 int SortedType::GetLength() const
31 {
32   return length;
33 }
34
35 void SortedType::MakeEmpty()
36 {
37   NodeType* tempPtr;
38
39   while (listData != NULL)
40   {
41     tempPtr = listData;
42     listData = listData->next;
43     delete tempPtr;
44   }
45   length = 0;
46 }
47
48 ItemType SortedType::GetItem(ItemType& item, bool& found)
49 {
50   bool moreToSearch;
51   NodeType* location;
52
53   location = listData;
54   found = false;
55   moreToSearch = (location != NULL);
56
57   while (moreToSearch && !found)
58   {
59     switch(item.ComparedTo(location->info))
60     {
61       case GREATER: location = location->next;
62                     moreToSearch = (location != NULL);
63                     break;
64       case EQUAL:   found = true;
65                     item = location->info;
66                     break;
67       case LESS:    moreToSearch = false;
68                     break;
69     }
70   }
71   return item;
72 }
73
74 void SortedType::PutItem(ItemType item)
75 {
76   NodeType* newNode;     // pointer to node being inserted
77   NodeType* predLoc;     // trailing pointer
78   NodeType* location;    // traveling pointer
79   bool moreToSearch;
80
81   location = listData;
```

```
82    predLoc = NULL;
83    moreToSearch = (location != NULL);
84
85    // Find insertion point.
86    while (moreToSearch)
87    {
88      switch(item.ComparedTo(location->info))
89      {
90        case GREATER: predLoc = location;
91                      location = location->next;
92                      moreToSearch = (location != NULL);
93                      break;
94        case LESS:    moreToSearch = false;
95                      break;
96      }
97
98    }
99
100   // Prepare node for insertion
101   newNode = new NodeType;
102   newNode->info = item;
103   // Insert node into list.
104   if (predLoc == NULL)          // Insert as first
105   {
106     newNode->next = listData;
107     listData = newNode;
108   }
109   else
110   {
111     newNode->next = location;
112     predLoc->next = newNode;
113   }
114   length++;
115 }
116 void SortedType::DeleteItem(ItemType item)
117 {
118   NodeType* location = listData;
119   NodeType* tempLocation;
120
121   // Locate node to be deleted.
122   if (item.ComparedTo(listData->info) == EQUAL)
123   {
124     tempLocation = location;
125     listData = listData->next;  // Delete first node.
126   }
127   else
128   {
129     while (item.ComparedTo((location->next)->info) != EQUAL)
130       location = location->next;
131
132     // Delete node at location->next
133     tempLocation = location->next;
134     location->next = (location->next)->next;
135   }
```

```
136   delete tempLocation;
137   length--;
138 }
139
140 void SortedType::ResetList()
141 {
142   currentPos = NULL;
143 }
144
145 ItemType SortedType::GetNextItem()
146 {
147   ItemType item;
148   if (currentPos == NULL)
149     currentPos = listData;
150   item = currentPos->info;
151   currentPos = currentPos->next;
152   return item;
153
154 }
155
156 SortedType::~SortedType()
157 {
158   NodeType* tempPtr;
159
160   while (listData != NULL)
161   {
162     tempPtr = listData;
163     listData = listData->next;
164     delete tempPtr;
165   }
166 }
```