CMPSC 24: Lecture 6
Abstract Data Type: Lists

Divyakant Agrawal
Department of Computer Science
UC Santa Barbara
LIST ADT
Sorted and Unsorted List ADTs

**UNSORTED LIST**

Elements are placed into the list in no particular order.

**SORTED LIST**

List elements are in a sorted order---either numerically or alphabetically by the elements themselves, or by a component of the element (called a **KEY** member).

Name some possible keys
ADT Unsorted List

- **Transformers**
  - MakeEmpty
  - InsertItem
  - DeleteItem

- **Observers**
  - IsFull
  - GetLength
  - RetrieveItem

- **Iterators**
  - ResetList
  - GetNextItem
ADT Unsorted List

Common vocabulary

- location accesses a particular element
- Node(location) is all data of element
- Info(location) is the user's data at location
- Info(last) is the user's data at the last location
- Next(location) is the node following Node(location)

Two implementations
class UnsortedType // declares a class data type
{
    public: // 8 public member functions
        UnsortedType();
        void MakeEmpty( );
        bool IsFull( ) const;
        int GetLength( ) const; // returns length of list
        void RetrieveItem( ItemType& item, bool& found );
        void InsertItem( ItemType item );
        void DeleteItem( ItemType item );
        void ResetList( );
        void GetNextItem( ItemType& item );
};

Public declarations are the same for both implementations; only private data changes
Generic Data Type

• A type for which the operations are defined but the types of the items being manipulated are not defined
• *How can we make the items on the list generic?*
• List items are of class `ItemType`, which has a `ComparedTo` function that returns (LESS, GREATER, EQUAL)
Array-Based Implementation

Private data members for array-based implementation

```java
private
    int length;
    ItemType info[MAX_ITEMS];
    int currentPos;
};
```

Where does MAX_ITEMS come from?
Array-Based Implementation

Notice the difference between the array and the list stored in the array

Array-based implementation
A special member function of a class that is implicitly invoked when a class object is defined

*What should the constructor do?*

```cpp
UnsortedType::UnsortedType()
{
    length = 0;
}
```
Checking for full and empty lists

• What is a full list? An empty list?

```cpp
bool UnsortedType::IsFull()
{
    return (length == MAX_ITEMS);
}
```

```cpp
bool UnsortedType::IsEmpty()
{
    return (length == 0);
}
```
If the list is unsorted, where should the next element go?

```
insert("Doe");
```
Insert

<table>
<thead>
<tr>
<th>info</th>
<th>[0]</th>
<th>Maxwell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[1]</td>
<td>Bradley</td>
</tr>
<tr>
<td></td>
<td>[2]</td>
<td>Asad</td>
</tr>
<tr>
<td></td>
<td>[3]</td>
<td>Doe</td>
</tr>
<tr>
<td></td>
<td>[MAX_ITEMS-1]</td>
<td></td>
</tr>
</tbody>
</table>

length 4

insert("Doe");

That was easy! Can you code it?
void UnsortedType::InsertItem(ItemType item)
// Post: item is in the list.
{
    info[length] = item;
    length++;
}
RetrieveItem

*How would you go about finding an item in the list?*

Cycle through the list looking for the item

*What are the two ending cases?*

The item is found

The item is not in the list

*How do we compare items?*

We use function ComparedTo in class ItemType
Pseudocode for RetrieveItem

Initialize location to position of first item
Set found to false
Set moreToSearch to (have not examined Info(last))
while moreToSearch AND NOT found
    if (item.ComparedTo(Info(location))) == EQUAL)
        { Set found to true
            Set item to Info(location)
        }
    else
        { Set location to Next(location)
            Set moreToSearch to (have not examined Info(last))
        }

Replace bold general statements with array-based code
void UnsortedType::RetrieveItem(ItemType& item, bool& found)
// Pre:  Key member(s) of item is initialized.
// Post: If found, item's key matches an element's key in the
//       list and a copy of that element has been stored in item;
//       otherwise, item is unchanged.
{
    bool moreToSearch;
    int location = 0;
    found = false;
    moreToSearch = (location < length);
    while (moreToSearch && !found){
        if (item.ComparedTo(info[location]) == EQUAL){
            found = true;
            item = info[location];
        } else {
            location++;
            moreToSearch = (location < length);
        }
    }
}
C++ code for RetrieveItem

```cpp
void UnsortedType::RetrieveItem(ItemType& item, bool& found)
// Pre: Key member(s) of item is initialized.
// Post: If found, item's key matches an element's key in the
//       list and a copy of that element has been stored in item;
//       otherwise, item is unchanged.
{
    bool moreToSearch;
    int location = 0;
    found = false;
    moreToSearch = (location < length);
    while (moreToSearch && !found)
    {
        if (item.ComparedTo(info[location]) == EQUAL)
        {
            found = true;
            item = info[location];
        }
        else
        {
            location++;
            moreToSearch = (location < length);
        }
    }
}
```

Loop invariant:

(location <= length) and
moreToSearch == (location < length) and
(!found → (ItemType.key not in Info [0..location-1])) and
found → ItemType.key == Info[location].key
Delete

How do you delete an item?
First you find the item

Yes, but how do you delete it?
Move those below it up one slot, or
Replace it with another item

What other item?
How about the item at info[length-1]?
C++ code for delete

```cpp
void UnsortedType::DeleteItem ( ItemType  item )
// Pre: item’s key has been initialized.
// An element in the list has a key that matches item’s.
// Post: No element in the list has a key that matches item’s.
{
    int  location  =  0 ;
    while (item.ComparedTo (info[location])  !=  EQUAL)
        location++;
    // move last element into position where item was located
    info [location] = info [length - 1 ] ;
    length-- ;
}
```

Why don't we have to check for end of list?
void UnsortedType::DeleteItem ( ItemType  item )
// Pre: item’s key has been initialized.
// An element in the list has a key that matches item’s.
// Post: No element in the list has a key that matches item’s.
{
    int  location  =  0 ;
    while (item.ComparedTo (info[location])  !=  EQUAL)
        location++;
    // move last element into position where item was located
    info [location] = info [length - 1 ] ;
    length-- ;
}

Why don't we have to check for end of list?

Loop invariant:
(location < length) and
(item.key in Info[location..length-1])
void PrintList(ofstream& dataFile, UnsortedType list)
// Pre: list has been initialized.
//      dataFile is open for writing.
// Post: Each component in list has been written.
//       dataFile is still open.
{
    int length;
    ItemType item;

    list.ResetList();
    length = list.GetLength();
    for (int counter = 1; counter <= length; counter++)
    {
        list.GetNextItem(item);
        item.Print(dataFile);
    }
}
ResetList and GetNextItem

```cpp
void UnsortedType::ResetList ( )
// Pre:  List has been initialized.
// Post: Current position is prior to first element.
{
    currentPos = -1 ;
}

void UnsortedType::GetNextItem ( ItemType& item )
// Pre:  List has been initialized. Current position is defined.
//       Element at current position is not last in list.
// Post: Current position is updated to next position.
//       item is a copy of element at current position.
{
    currentPos++  ;
    item = info [currentPos] ;
}
```
Class **ItemType**

```cpp
// SPECIFICATION FILE itemtype.h

const int MAX_ITEM = 5;
enum RelationType { LESS, EQUAL, GREATER };

class ItemType // declares class data type
{
    public: // 3 public member functions
        RelationType ComparedTo( ItemType ) const;
        void Print( ) const;
        void Initialize( int number );

    private: // one private data member
        int value; // could be any type
};
```
Class ItemType

// IMPLEMENTATION FILE ( itemtype.cpp )
// Implementation depends on the data type of value.

#include "itemtype.h"
#include <iostream>

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

void Print( ) const
{
    using namespace std;
    cout << value << endl;
}

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

How would this class change if the items on the list were strings rather than integers?
UML Diagram

**ItemType**
- `MAX_ITEMS`: 5
- `RelationType`: {LESS, GREATER, EQUAL}
  - `value`: int
- `ItemType()`
- `CompareTo(otherItem: ItemType)`: RelationType
- `Print(ofstream): void`
- `Initialize(number: int): void`

**UnsortedType**
- `length`: int
- `currentPos`: int
- `listData`: array
- `UnsortedType()`
- `GetLength(): int`
- `IsFull(): bool`
- `MakeEmpty(): void`
- `RetrieveItem(item: ItemType&, found: bool&): void`
- `InsertItem(item: ItemType): void`
- `DeleteItem(item: ItemType): void`
- `ResetList(): void`
- `GetNextItem(item: ItemType&): void`
Class **ItemType**

// SPECIFICATION FILE itemtype.h

const int MAX_ITEM = 5;
enum RelationType { LESS, EQUAL, GREATER };

class ItemType // declares class data type
{
    public: // 3 public member functions
        RelationType ComparedTo( ItemType ) const;
        void Print( ) const;
        void Initialize( int number );

    private: // one private data member
        int value; // could be any type
} ;
Class ItemType

// IMPLEMENTATION FILE ( itemtype.cpp )
// Implementation depends on the data type of value.

#include "itemtype.h"
#include <iostream>

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

void Print( ) const
{
    using namespace std;
    cout << value << endl;
}

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

How would this class change if the items on the list were strings rather than integers?
Linked List Implementation of Unsorted List
Linked List Implementation

Be sure you understand the differences among location, *location, and location->info.
// SPECIFICATION FILE    ( unsortedType.h )
#include "ItemType.h"
struct NodeType;

class UnsortedType    // declares a class data type
{
    public:            // 8 public member functions
        UnsortedType();
        void MakeEmpty( );
        bool IsFull( ) const;
        int GetLength( ) const;  // returns length of list
        void RetrieveItem( ItemType&  item, bool&  found );
        void InsertItem( ItemType  item );
        void DeleteItem( ItemType  item );
        void ResetList( );
        void GetNextItem( ItemType&  item );
};
Linked List Implementation

*Remember the design notation?*

Node(location)       *location

Info(location)       location->info

Next(location)       location->next

How do you set location to Next (location)?

How do you set Info(location) to value?
Linked List Implementation (private part)

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

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

List with two items
Linked List Implementation

*How do you know that a linked list is empty?*

- listData is NULL

*What should the constructor do?*

- Set length to 0
- Set listData to NULL

*What about currentPos?*

- We let ResetList take care of initializing currentPos
Linked List Implementation

What about the observers IsFull and GetLength?
GetLength just returns length

Can a linked list ever be full?
Yes, if you run out of memory
Ask for a new node within a try/catch
bool Unsortedtype::IsFull() const
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
        return false;
    }
    catch (std::bad_alloc exception)
    {
        return true;
    }
}

What about MakeEmpty?
Exceptions

An unusual, generally unpredictable event, detectable by software or hardware, that requires special processing; the event may or may not be erroneous.

Three parts to an exception mechanism:
- **Define** the exception
- **Generate** (raising) the exception
- **Handling** the exception

Try, throw, and catch statements in C++
Exceptions

• An exception is an unusual situation that occurs when the program is running.

• Exception Management
  – Define the error condition
  – Enclose code containing possible error (try).
  – Alert the system if error occurs (throw).
  – Handle error if it is thrown (catch).
**try, catch, and throw**

```
Try
{
    // code that contains a possible error
    ... throw string("An error has occurred in function ...");
}
Catch (string message)
{
    std::cout << message << std::endl;
    return 1;
}
```
void Unsortedtype::MakeEmpty()
{
    NodeType* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}
Linked List Implementation of RetrieveItem

Initialize location to position of first item
Set found to false
Set moreToSearch to (have not examined Info(last))
while moreToSearch AND NOT found
  if item.ComparedTo(Info(location)) == EQUAL
    { Set found to true;
    Set item to Info(location)
  }
else
  {
    Set location to Next(location)
    Set moreToSearch to (have not examined Info(last))
  }

Replace bold general statements with linked list code
void UnsortedType::RetrieveItem( ItemType& item, bool& found )
// Pre: Key member of item is initialized.
// Post: If found, item’s key matches an element’s key in the list
// and a copy of that element has been stored in item; otherwise,
// item is unchanged.
{
    bool moreToSearch;
    NodeType* location;
    location = listData;
    found = false;
    moreToSearch = ( location != NULL )
while ( moreToSearch && !found )
{
    if ( item.ComparedTo( location->info ) == EQUAL )
    {
        found = true;
        item = location->info;
    }
else
    { location = location->next;
        moreToSearch = ( location != NULL );
    }
}

Loop invariant:
(location == NULL or points to a node on the list) and
moreToSearch == (location != NULL) and
(found → (ItemType.key not in list prior to location) and
found → ItemType.key == Info[location].key)
InsertItem

*How do we add a node to our list?*

Get a node using `new`

```cpp
location = new NodeType
```

Put value into info portion

```cpp
location->info = Item
```

Put node into list ...

*Where? Where should the node go?*

*Does it matter?*
InsertItem

(a) new(location)

(b) info(location) <- newElement

Now we must put the two parts together--carefully!
These steps must be done in this order! Why?
void UnsortedType::InsertItem ( ItemType item )
// Pre: list is not full and item is not in list.
// Post: item is in the list; length has been incremented.
{
    NodeType* location;
    // obtain and fill a node
    location = new NodeType<ItemType>;
    location->info = item;
    location->next = listData;
    listData = location;
    length++;
}
Deleteltem

*How do you delete an item?*

Find the item

Remove the item

(a) Delete Lila

(b) Delete Kate
void UnsortedType::DeleteItem(ItemType item)
// Pre: item’s key has been initialized and an element in the list has a key that matches item’s
// Post: No element in the list has a key that matches item’s
{
    NodeType* location = listData;
    NodeType* tempLocation;
    // Find the item
    if (item.ComparedTo(listData->info) == EQUAL)
    {
        // item in first location
        tempLocation = location;
        listData = listData->next;
    }
    else
    {
        while (item.ComparedTo((location->next)->info) != EQUAL)
        {
            location = location->next;
            tempLocation = location->next;
            location->next = (location ->next)->next;
        }
    }
    delete tempLocation;
    length--;
Linked Implementation

ResetList and GetNextItem

What was `currentPos` in the array-based implementation?

What would be the equivalent in a linked implementation?
void UnsortedType::ResetList()
// Pre: List has been initialized.
// Post: Current position is prior to first element.
{
    currentPos = NULL;
}
void UnsortedType::GetNextItem(ItemType& item)
// Pre: List has been initialized. Current position is defined.
    // Element at current position is not last in list.
// Post: Current position is updated to next position.
    // item is a copy of element at current position.
{
    if (currentPos == NULL)
        currentPos = listData;
    else
        currentPos = currentPos->next;
    item = currentPos->info;
}
UML Diagram

Note the differences between the UML Diagrams for the two implementations.
C++ concepts to come

Exceptions
Deep vs. shallow copying
Constructor, destructor, copy constructor