Lecture 6

List ADT

It is very pervasive
Lecture Overview

- List ADT
  - Specification

- Implementation for List ADT
  - Array Based
    - Pros and Cons
  - Linked List Based
    - Pros and Cons
List ADT

- A sequence of items where positional order matter \(<a_1, a_2, \ldots, a_{n-1}, a_n>\)
- Lists are very pervasive in computing
  - e.g. student list, list of events, list of appointments etc

<table>
<thead>
<tr>
<th>idx</th>
<th>Position, integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>Data stored in list, can be any data type</td>
</tr>
</tbody>
</table>

The list ADT

- List()
- isEmpty()
- add(idx, item)
- remove(idx)
- retrieve(idx, item)
- getLength()
// includes are not shown

class ListBase {
public:
    virtual bool isEmpty() = 0; // Operations to check on the state of list.
    virtual int getLength() = 0;

    virtual bool insert(int index, const int& newItem) = 0;
    virtual bool remove(int index) = 0;
    virtual bool retrieve(int index, int& dataItem) = 0;

    virtual string toString() = 0; // Operation to ease printing & debugging.
};
Design Decisions

- This is a simplified design:
  - to reduce the "syntax burden"
  - to concentrate on the internal logic

- You are encouraged to enhance the class:
  - After you have understood the internal logic

- Possible enhancements:
  - Use **Template Class**:
    - So that list can contain item of any data type
  - Use **Inheritance + Polymorphism**:
    - Similar to the Complex Number ADT
Two Major Implementations

1. Array implementation
2. Linked list implementation (discussed soon)

- General steps:
  1. Choose an internal data structure
     - e.g. Array or linked list
  2. Figure out the algorithm needed for each of the major operations in List ADT:
     - insert, remove, and retrieve
  3. Implement the algorithm from step (2)
List ADT – Version A

Array Implementation
Implement List ADT: Using Array

- Array is a prime candidate for implementing the ADT
  - Simple construct to handle a collection of items

**Advantage:**
- Very fast retrieval

```
<table>
<thead>
<tr>
<th>size</th>
<th>array with [0..m] of locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a_1</td>
</tr>
<tr>
<td>1</td>
<td>a_2</td>
</tr>
<tr>
<td>2</td>
<td>a_3</td>
</tr>
<tr>
<td>......</td>
<td>..........</td>
</tr>
<tr>
<td>n-1</td>
<td>a_n</td>
</tr>
<tr>
<td>m-1</td>
<td>......</td>
</tr>
</tbody>
</table>
```

Internal of the list ADT, Array Version
Insertion : Using Array

- **Simplest Case:** Insert to the end of array
- Other Insertions:
  - Some items in the list needs to be shifted
  - **Worst case:** Inserting at the head of array

Example: Insert item “it” into the 3\textsuperscript{rd} position

```
size
8
```

```
items
a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8 \ \ldots
```

- **Step 1:** Shift right
- **Step 2:** Write into gap
- **Step 3:** Update Size
Deletion: Using Array

- **Simplest Case:** Delete item from the end of array
- Other deletions:
  - Items needs to be shifted
  - **Worst Case:** Deleting at the head of array

**Example:** remove the item at 5\textsuperscript{th} position

```
size
8
```

```
| a_1 | a_2 | a_3 | a_4 | a_5 | a_6 | a_7 | a_8 | ....... |
```

**Step 1: Close Gap**

```
size
7
```

```
| a_1 | a_2 | a_3 | a_4 | a_6 | a_7 | a_8 | ....... |
```

**Step 2: Update Size**
#include "ListBase.h"

const int MAX_LIST = 50;

class ListArray : public ListBase {

private:
    int _size;
    int _items[MAX_LIST];

public:
    ListArray();
    virtual bool isEmpty();
    virtual int getLength();

    virtual bool insert(int index, const int& newItem);
    virtual bool remove(int index);
    virtual bool retrieve(int index, int& dataItem);

    virtual string toString();
};
#include <sstream>
#include "ListArray.h"

ListArray::ListArray() {
    _size = 0;
}

bool ListArray::isEmpty() {
    return _size == 0;
}

int ListArray::getLength() {
    return _size;
}

- `isEmpty()` and `getLength()` methods are easy to code:
  - will be omitted in later implementations
bool ListArray::insert(int userIdx, const int& newItem) {
    int index = userIdx-1;

    if (_size >= MAX_LIST)
        return false;  // Maximum capacity reached

    if ((index < 0) || (index >= _size+1))
        return false;  // List index out of range

    for (int pos = _size-1; pos >= index; pos--)
        _items[pos+1] = _items[pos];  // Step 1. Shift items

    _items[index] = newItem;  // Step 2. Write into gap
    _size++ ;  // Step 3. Update Size

    return true;
}
bool ListArray::remove(int userIdx) {
    int index = userIdx - 1;

    if ((index < 0) || (index >= _size))
        return false;

    for (int pos = index; pos < _size - 1; pos++)
        _items[pos] = _items[pos + 1];

    _size--;  
    return true;
}
### List Array: Implementation (4/4)

```cpp
bool ListArray::retrieve(int userIdx, int& dataItem) {
    int index = userIdx - 1;

    if ((index < 0) || (index >= _size))
        return false;

    dataItem = _items[index];
    return true;
}
```

Retrieval is simple, as array item can be accessed directly.

The result is passed back through the reference parameter.

```cpp
string ListArray::toString() {
    ostringstream os;

    os << "[ ";
    for (int i = 0; i < _size; i++)
        os << _items[i] << " ";
    os << "]";

    return os.str();
}
```

A useful method to print all items into a string with the format

```
[ item1 item2 ... itemN ]
```
Using the List ADT: User Program

- Instead of an actual List ADT application, we show a program used to test the implementation of various List ADT operations.

Pay attention to how we test the operations:

- For each operations:
  - Test different scenarios, basically to exercise different "decision path" in the implementation.
  - For example, to test the `insert` operation:
    - Insert into an empty list
    - Insert at the first, middle and last position of the list
    - Insert with incorrect index
#include <iostream>
#include "ListArray.h"
using namespace std;

int main() {
    ListArray intList;
    int rItem;

    if (intList.insert(1, 333))
        cout << "Insertion successful!\n";
    else
        cout << "Insertion failed!\n";

    intList.insert(1, 111);
    intList.insert(3, 777);
    intList.insert(3, 555);

    This is one way to use the operations: Check the return result for the status of the operation.

    Using the array implementation of list

    If the insertion is implemented properly, the list should contain [ 111 333 555 777 ] at this point
}

List ADT : Sample User Program 1/2
cout << intList.toString() << endl;

intList.retrieve(1, rItem);
cout << "First item is " << rItem << endl;
intList.retrieve(intList.getLength(), rItem);
cout << "Last item is " << rItem << endl;

cout << "Remove test" << endl;
intList.remove(1);
intList.remove(2);
intList.remove(intList.getLength());

intList.retrieve(1, rItem);
cout << "First item is " << rItem << endl;
intList.retrieve(intList.getLength(), rItem);
cout << "Last item is " << rItem << endl;

return 0;
}
Array Implementation: Efficiency (time)

- **Retrieval:**
  - **Fast:** one access

- **Insertion:**
  - **Best case:** No shifting of elements
  - **Worst case:** Shifting of all $N$ elements.

- **Deletion:**
  - **Best case:** No shifting of elements
  - **Worst case:** Shifting of all $N$ elements
Array Implementation: Efficiency (space)

- Size of array is restricted to MAX_LIST

Problem:

- Maximum size is not known in advance
  - MAX_LIST is too big == unused space is wasted
  - MAX_LIST is too small == run out of space easily

Solution:

- Make MAX_LIST a variable
- When array is full:
  1. Create a larger array
  2. Move the elements from the old array to the new array

- No more limits on size, but space wastage and copying overhead is still a problem
Array Implementation: Observations

- For fixed-size collections
  - Arrays are great

- For variable-size collections, where dynamic operations such as insert/delete are common
  - Array is a poor choice of data structure

- For such applications, there is a better way......
List ADT – Version B

Linked List Implementation
Implement List ADT using Linked List

- **Pointer Based Linked List:**
  - Allow elements to be **non-contiguous** in memory
  - Order the elements by associating each with its **neighbour(s)** through pointers

```plaintext
This is one node in the list

... and this one comes after it.
```
A single node in the Linked List

```c
struct ListNode {
    int item;
    ListNode *next;
};
```

C++ Allows structure name to be used **without** the keyword `struct`
List ADT: Using Linked List

- List of four items \( <a_1, a_2, a_3, a_4> \)

We need:
- \textbf{head} pointer to indicate the first node
  - Other nodes are accessed by "hopping" through the next pointer
- \textbf{size} for the number of items in the linked list
Linked List Implementation: Design

- Linked list implementation is more complicated:
  - Need to handle a number of scenarios separately

- Let us walkthrough the insertion algorithm in detail:
  - Highlight the importance of design before coding
  - Highlight the design considerations:
    - How to consider different cases?
    - How to modularize and reuse common code?
Linked List Insertion: General

- List ADT provides the `insert()` method to add an item:
  - The new item itself is given
  - The index \([1\ldots size+1]\) of the new item is given

- Due to the nature of linked list, there are several possible scenarios:
  1. Item is added to an **empty** linked list
  2. Item is added to the **head (first item)** of the linked list
  3. Item is added to the **last position** of the linked list
  4. Item is added to the **other positions** of the linked list
Linked List Insertion: Preliminary

- The **List** object stores:
  - Head pointer and the current size of linked list

```cpp
class List {
private:
    int _size;
    ListNode* _head;
    ...
};
```

- For all valid cases, we need to construct a new linked list node to store the new item

```cpp
ListNode* newPtr;
newPtr = new ListNode;
newPtr->item = a_new;
newPtr->next = NULL;
```
Insertion Case 1: Empty Linked List

Question: is `newPtr` needed after this operation?

```c
_size++;  
_head = newPtr;
```
Insertion Case 2: Head of Linked List

```c
_size++;
newPtr->next = _head;
_head = newPtr;
```

**Question 1:** Can we reorder the last two lines of code above?

**Question 2:** Very similar to previous case, can we combine?
Insertion Case 1 and 2: Common Code

- Insert into head of linked list (possibly empty)

Can use the same code:

```cpp
_size++; 
newPtr->next = _head; 
_head = newPtr;
```
Linked List Insertion: Traversal

- Since we only keep the head pointer, **list traversal** is needed to reach other positions
  - Needed for insertion case 3 and 4

![Diagram of linked list traversal]

```
_move_to_posth_node_in_list
ListNode *ptr;

ptr = _head;
for (i = 1; i < Pos; i++) {
    ptr = ptr->next;
}
```
Insertion Case 3: End of Linked List

ListNode* prev;
prev = Traverse to $N^{th}$ node
_size++;
prev->next = newPtr;

Use the list traversal code discussed to move $\text{prev}$ pointer
Insertion Case 4: \(K^{th}\) Position (Middle)

We only need to change at most two pointers for ANY insertion.
Insertion Case 3 and 4: Common Code

- The code for case 4 happens to be a more general form of case 3:
  - Case 3 can be handled with the same code in case 4

```c
ListNode* prev;
K = N+1
prev = Traverse to K-1\textsuperscript{th} node
_size++;
newPtr->next = prev->next;
prev->next = newPtr;
```
Insertion Code: Summary

- To insert **ItemNew** into **Index**\(^{th}\) position
  - Assume **Index** is in range [1, size+1]

```c
ListNode *newPtr, *prev;

1. newPtr = new ListNode // Create a new node
   i. item = ItemNew
   ii. next pointer = NULL
2. _size increases by 1
3. If Index is 1    // Case 1 + 2
   i. newPtr->next = _head
   ii. _head = newPtr
4. Else              // Case 3 + 4
   i. prev = Traverse to **Index**-1\(^{th}\) node
   ii. newPtr->next = prev->next
   iii. prev->next = newPtr
```
For Linked List deletion, the cases can be simplified similar to:
1. Deletion of head node (1^{st} Node in list)
2. Deletion of other node (including middle or end of list)

Try to deduce the code logic using similar approach
Deletion Case 1: Head of Linked List

```c
ListNode* cur;
_size--;  
cur = _head;
_head = _head->next;
delete cur;
```

**Question:** What if there is only 1 node? Will the code work?
Deletion Case 2: K\textsuperscript{th} Position (Middle)

```c
ListNode *prev, *cur;
prev = Traverse to K-1\textsuperscript{th} node
_size--;  // Subtracting 1
cur = prev->next;
prev->next = cur->next;
delete cur;
```

**Question:** What if the K\textsuperscript{th} node is the last node?
Deletion Code: Summary

To delete item at $Index^{th}$ position

- Assume $Index$ is in range [1….size]

```c
ListNode *prev, *cur;

1. _size decreases by 1
2. If Index is 1 // Case 1
   i. cur = _head
   ii. _head = _head->next;
3. Else // Case 2
   i. prev = Traverse to $Index-1^{th}$ node
   ii. cur = prev->next
   iii. prev->next = cur->next
4. Free memory pointed by cur pointer
```

The traversal code can be shared between insertion and deletion. Let's make it into another method.
List (Linked List): Specification

```cpp
#include "ListBase.h"
class ListLL : public ListBase {
private:
    struct ListNode {
        int item;
        ListNode *next;
    };
    int _size;
    ListNode* _head;
    ListNode* traverseTo(int index);
public:
    ListLL();
    ~ListLL();

    virtual bool isEmpty();
    virtual int getLength();

    virtual bool insert(int index, const int& newItem);
    virtual bool remove(int index);
    virtual bool retrieve(int index, int& dataItem);

    virtual string toString();
};
```

The ListNode structure declaration is hidden from outsider.

The traversal method is only used internally.
ListLL: Implementation (1/5)

```cpp
ListLL::ListLL() {
    _head = NULL;
    _size = 0;
}

ListLL::~ListLL() {
    while (!isEmpty())
        remove(1);
}

string ListLL::toString() {
    ostringstream os;
    ListNode *cur;

    os << "[ ";
    for (cur = _head; cur != NULL; cur = cur->next) {
        os << cur->item << " ";
    }
    os << "]";
    
    return os.str();
}
```

We need a destructor to free each node as they are dynamically allocated.

Go through all nodes to print all items.
ListLL: Implementation (2/5)

This is the return type of `find()` method. `ListLL::` is needed because `ListNode` is a private declaration in class `ListLL`.

```cpp
ListLL::ListNode* ListLL::traverseTo(int index) {
    if ((index < 1) || (index > getLength()))
        return NULL;
    else {
        ListNode *cur = _head;

        for (int skip = 1; skip < index; skip++)
            cur = cur->next;

        return cur;
    }
}
```

Traversal code as discussed.

This method is used in both insertion and deletion.

ListLL.cpp (part 2)
bool ListLL::insert(int userIdx, const int& newItem) {
    int newLength = getLength()+1;

    if ((userIdx < 1) || (userIdx > newLength))
        return false;
    else {
        ListNode *newPtr = new ListNode;
        newPtr->item = newItem;
        newPtr->next = NULL;
        _size = newLength;

        if (userIdx == 1) {
            newPtr->next = _head;
            _head = newPtr;
        } else {
            ListNode *prev = traverseTo(userIdx-1);
            newPtr->next = prev->next;
            prev->next = newPtr;
        }
    }
    return true;
}
bool ListLL::remove(int userIdx) {
    ListNode *cur;

    if ((userIdx < 1) || (userIdx > getLength()))
        return false;
    else {
        --_size;
        if (userIdx == 1) {
            cur = _head;
            _head = _head->next;
        }
        else {
            ListNode *prev = traverseTo(userIdx-1);
            cur = prev->next;
            prev->next = cur->next;
        }
        delete cur;
        cur = NULL;
    }
    return true;
}
With the complete linked list implementation of List ADT:
- We can use the same user program to try it out
- What do you think is the main difference between array and linked list implementation of List ADT?
List ADT: Sample User Program (Again!)

```cpp
#include <iostream>
#include "ListLL.h"
using namespace std;

int main() {
    ListLL intList;
    int rItem;

    // All other usage of List ADT remain
    // unchanged............

    // You can see how easy to change between
    // the two implementations of List ADT
```

Question:
What do you think is the main difference between `ListArray` and `ListLL` implementation of List ADT?
References

- [Carrano]
  - Chapter 3
    - List ADT and array based implementation
  - Chapter 4
    - Linked List and STL list

- [Koffman & Wolfgang]
  - Chapter 4.5 to 4.8
Summary

User Programs

Uses

List ADT

- insert()
- remove()
- retrieve()

Manipulation is possible for all positions

Implements

Array

Implements

Linked List

Applications

List

Implementations
Summary

- List ADT
  - Usage
  - Specification

- Implementation of List ADT
  - Array Based
    - Pros and Cons
  - Linked List Based
    - Pros and Cons