6. LL Mechanics

**Intro Data Structures & SE**

**Table of Contents**
- Linear Lists
- List Node Design
- Self Referencing Structures
- List Formation Example
- List Traversal
- Linked List Operations
- Data Element Dependency
- Some Basic Operations
- List Insertion Cases 1 & 2
- List Insertion Case 3
- List Insertion Case 4
- Ordered Insertion
- List Remove Case 1
- List Remove Case 2
- List Remove Case 3
- Ordered Remove
- Ordered Modify
- Linear Linked-List Variations
- Linked-List Variation Declarations
- Circular Double Linked Insertion
- Circular Double Linked Deletion
- Ordered List: Array of Records
- Array of Records: Manipulations
- Array of Records: Insert

**Sequential Lists**

Array
- Logical order matches physical order
  - i.e.: Class List = (1704, 2704, 2604)

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1704</td>
<td>2704</td>
<td>2604</td>
</tr>
<tr>
<td>address</td>
<td>004A08</td>
<td>004A0C</td>
<td>004A10</td>
</tr>
</tbody>
</table>

**Linked Lists**

- Logical group of ordered elements whose physical order is independent of location in physical storage

**Linear Lists**

- List is a sequence of nodes. Each node contains data and a pointer to the next node in the list.
- Here, Head and Tail pointers are used to keep track of the ends of the list.

Inserting & deleting elements in linked lists only require changes to the links.

Inserting & deleting elements in a sequential list requires copying & shifting of elements.
List Node Design

Each node of a linked list is a structured variable that contains both data and (at least) a pointer to the next node in the list.

There should be a declaration of a node type. That type may be either a struct or a class. We will consider both approaches in these notes.

Whichever approach is taken, the Data field “should” (must) encapsulate all logical data stored in the node and separate that data from the Link field which points to the next node in the linked list.

So, the Data field should also be a structured variable (struct or class) and the node type will then be hierarchical.

Ideally, the linked list is implemented as a “container” for data; that is, the list will accept data elements of essentially any type.

We’ll achieve that goal by treating the data element as a type defined “capsule”.

We will revisit the issue of the proper engineering of a linked list later; for now, we will focus on the mechanics of creating and manipulating a linked list.

Self Referencing Structures

Linked List Node Declaration Issues

A linked list node must contain a pointer to a linked list node. That poses a delicate scope issue in C++.

```
typedef int Item;
typedef struct {
    Item   Element;
    Node*  Next;
} Node; //error
```

This code won’t compile. The declaration of the pointer field uses the type name Node before Node has been declared.

```
typedef int Item;
struct Node {     // forward declaration
    Item   Element;
    Node*  Next;
};
```

This code will compile, but it’s better practice to typedef the pointers. And that brings us back to the situation above. You can’t typedef Node* before Node has been declared.

Forward Declaration

Fortunately, C++ provides a solution. We can simply make a forward declaration of Node, prior to defining the Node type:

```
typedef int Item;
struct Node; // forward declaration
typedef Node* NodePtr;
```

```
struct Node {     // forward declaration
    Item   Element;
    NodePtr Next;
};
```

```
NodePtr Head = NULL; // head pointer for list
```
List Formation Example

Assume the declarations from the previous slide.

```cpp
NodePtr p = NULL;
q = NULL;
p = new Node; // #1
p->Element = 120;  // #2
p->Next = NULL;    // #3
Head = p;          // #4
p = NULL;
```

```
p = new Node;
p->Element = 6; p->Next = Head; Head = p;
p = new Node;      // #1
p->Element = 28;   // #2
p->Next = Head;    // #3
Head = p;          // #4
p = NULL;
```

// What would the following stmt do?
`delete p;`
Linked List Operations

Basic Sorted/Ordered List Operations: Design

The following operations are necessary for a robust, well-defined minimal interface to an ordered linked-list ADT:

- `makeList()`: creates & initializes List to be empty
- `emptyList()`: tests if the List is empty
- `fullList()`: tests if the List is full
- `insert()`: inserts a given item in L in the correct ordered position
- `remove()`: searches List for an item with some given "key" value & deletes the element if located
- `retrieveElem()`: searches List for an item with some given "key" value and returns the element if located
- `modify()`: searches List for an item with some given "key" value & replaces the item with a given, (updated), item if located
- `destroyList()`: erases all items from the List

The ordered list ADT operations are responsible for maintaining the sorted list item ordering, and the integrity of the list structure NOT the user/client of the ADT. Error checking responsibilities are shared between the ordered list ADT & the user/client of the ADT.

An unordered list ADT embodies the concept of a list position and entails a different set of operations.

Data Element Dependency

Type Defined Data Element Declaration

```cpp
typedef int Item;
struct Node; //forward declaration
typedef Node* NodePtr;
struct Node {
  Item Element; //type defined data field
  NodePtr Next;
};
```

Considerations

In the implementation of the list operations, one needs to be able to compare variables of type `Item` to each other to maintain the ordering; but an `Item` variable could be anything from a simple type (such as an `int`) to a highly complex structured type. The client/user should decide how the list will be ordered, (i.e. upon what struct field or class data member(s), termed the key/primary field/member).

Thus, the implementation must depend on the internal details of `Item` and that dependency should be localized as much as possible. If the `Item` type is a simple base type then the C++ language comparison operations provide this service. This can be easily accomplished for a class by overloading the relational operators, which will be covered later.

```cpp
bool lessThan(Item elem1, Item elem2);
bool equalTo(Item elem1, Item elem2);
bool greaterThan(Item elem1, Item elem2);
```
Some Basic Operations

List Initialization

Head pointer is NULL when list is empty.

Empty List Test

Head pointer contains address of first node if list is NOT empty.

Full List Test

Dependent upon underlying implementation

Destroy List

Head pointer is NULL when list is empty.

Head pointer contains address of first node if list is NOT empty.

List Insertion Case 1 and 2

#1: Insert into empty list

// create a new Node
p = new(nothrow) Node; // #1
// initialize new Node
p->Element = 6; // #2
p->Next = NULL; // #3
// insert new Node
Head = p; // #4

#2: Insert at end (tail) of list

Here we assume we have a pointer Tmp to the last node in the list.

Declarations

NodePtr Head = NULL;
NodePtr p = NULL;

#1: Insert into empty list

p = new(nothrow) Node; // #1
p->Element = 6; // #2
p->Next = NULL; // #3
Head = p; // #4

#2: Insert at end (tail) of list

Here we assume we have a pointer Tmp to the last node in the list.
List Insertion Case 3

#3: Insert into middle of list

Here we assume we have a pointer Tmp to the Node that will precede the Node that’s to be inserted.

```
// create and init new Node
p = new(nothrow) Node; // #1
p->Element = 55;       // #2
p->Next    = NULL;     // #3

// insert new Node
p->Next = Tmp->Next; // #4
Tmp->Next = p;       // #5
```

Note that #4 and #5 must be performed in the correct order.

What happens if the statements are reversed?

List Insertion Case 4

#4: Insert at the front (head) of the list

```
// create and initialize new Node
p = new(nothrow) Node; // #1
p->Element = 1;        // #2
p->Next = NULL;        // #3

// insert new Node
p->Next = Head;        // #4
Head = p;              // #5
```

What happens if statements #4 and #5 are reversed here?
6. LL Mechanics

Ordered Insertion

Ordered (ascending) List Insertion code:

```c++
// Pre: List is initialized
// Post: List contains the inserted elem in order
// insert()
{
    NodePtr prevPtr, currPtr, newPtr;
    newPtr = new(nothrow) Node;
    if (newPtr == NULL)
        return false;          //no memory available
    newPtr->Element = elem; //elem is item to
    newPtr->Next = NULL; //be inserted
    prevPtr = NULL;
    currPtr = Head;
    while ((currPtr != NULL) &&
        (elem > currPtr->Element) {
        prevPtr = currPtr;
        currPtr = currPtr->Next;
    }
    if (prevPtr == NULL) {    //insert at head, or
        newPtr->Next = Head;   //        empty list
        Head = newPtr;
    } else {
        prevPtr->Next = newPtr;  //insert in middle
        newPtr->Next = currPtr; //      or at tail
    }
    return true ;          // successful insertion
}
```

prevPtr is used as a 'trailer' pointer

depends on Boolean short-circuiting

Assumes relational item comparison

List Remove Case 1

Consider the list:

```plaintext
Head -> 6 -> 28 -> 120 -> ...
```

Case #1: remove the head of the list

```c++
// get a ptr to the target Node
p = Head;              // #1
// reset Head "around" target Node
Head = Head->Next;     // #2
p->Next = NULL;        // #3
// delete the target Node
delete p;              // #4
```

Pointer reset in deleted node (#3) is logically unnecessary since Node will be deleted in the next statement.
6. LL Mechanics

List Remove Case 2
Case #2: remove from the middle of the list
Assume the initial list below. Now we will need a pointer p to the Node that PRECEDES the targeted Node (containing 71 in this case).

```
Head → 6 → 43 → 71 → 120 •
p
```

// get ptr to targeted Node
NodePtr Tmp = p->Next; // #1
// reset ptr in preceding Node "around" target Node
p->Next = Tmp->Next; // #2
Tmp->Next = NULL; // #3
// delete targeted Node
delete Tmp; // #4

List Remove Case 3
Case #3: remove the last Node (tail) of the list
Assume the initial list below. Again, we will need a pointer p to the Node that PRECEDES the targeted Node (containing 120 in this case).

```
Head → 6 → 43 → 71 → 120 •
p
```

// get ptr to targeted (tail) Node
NodePtr Tmp = p->Next; // #1
// reset ptr in preceding Node
p->Next = Tmp->Next; // #2
Tmp->Next = NULL; // #3
// delete targeted Node
delete Tmp; // #4

#2 could also be:
p->Next = Tmp->Next;
So, this case is really just the same as deleting in the middle of the list.
Ordered Remove

Ordered (ascending) List Removal Function

```c
// Pre: List is initialized
// Post: delElem Item has been removed from the List
// remove()
// Uses a one-node lookahead technique. Trailer pointer method is also applicable
{   NodePtr ptr, delPtr;
    ptr = Head;
    if (emptyList(Head))
        return false;            // removal failure
    if (delElem == Head->Element) {
        Head = Head->Next;      // delete head node
        ptr->Next = NULL;
        delete ptr;
        return true;            // successful removal
    }
    // check for 1-element list
    if (ptr->Next == NULL)
        return false;               // removal failure
    // list has > 1-element
    // perform 1-element look-ahead search while( (ptr->Next->Next != NULL) &&
    // if (delElem == ptr->Next->Element){
        delPtr = ptr->Next;
        ptr->Next = delPtr->Next;
        delPtr->Next = NULL;
        delete delPtr;
        return true;             // successful removal
    } //end of list && delElem not found
    return false;            // removal failure
}
```

Ordered Modify

```c
// Pre: List is initialized
// Post: modElem Item has been replaced in the List
// modify()
{   NodePtr p = Head;
    bool foundElement; false;
    bool endList = (Head == NULL);
    if (!endList)
        foundElement = (p->Element == modElem);
    while (!endList && !foundElement) {
        p = p->Next;
        endList = (p == NULL);
        if (!endList)
            foundElement = (p->Element == modElem);
    } //while
    if (foundElement) {  //replace list element
        p->Element = modElem;return true;      //successful modification
    }
    return false;        //unsuccessful modification
}
```

modify does NOT rely upon Boolean short-circuiting.
6. LL Mechanics

Linear Linked-List Variations

Circular List

Double Linked-List (non-circular)

Circular Double Linked-List

Linked-List Variation Declarations

Double Linked-List Declaration (non-circular)

```c
typedef int Item;
struct Node;
typedef Node* NodePtr;
struct Node {
    Item Element;
    NodePtr Prev, Next;
};
struct dblList{
    NodePtr First, Last;
};
dblList dlist;
```

Double Linked-List Declaration (circular)

```c
typedef int Item;
struct Node;
typedef Node* dblList;
struct Node {
    Item Element;
    dblList Prev, Next;
};
dblList dlist;
```
6. LL Mechanics

### Circular Double Linked-List Insertion

Insert 83 into the ordered list:

```
struct Node;
typedef struct Node* dblList;
struct Node {
    infoType Element;
    dblList Prev, Next;
};

dblList dlist;
p->Prev = q;         // #1
p->Next = q->next;   // #2
q->Next = p;         // #3
p->Next->Prev = p;   // #4
```

Order is important!

### Circular Double Linked-List Deletion

Delete 75 from the list:

```
p->Prev->Next = p->Next;   // #1
p->Next->Prev = p->Prev;   // #2
p->Next = p->Prev = NULL;  // #3
```

What if list has one element?

---

**Linked List Mechanics**
## Ordered List: Array of Records

**Representation**
- Array of Records (structures)

**Implementation Instance (nodes structure array)**

```
<table>
<thead>
<tr>
<th>List</th>
<th>Item</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>K</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>R</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>?</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Initialization
```
Avail: (0) ⇒ (1) ⇒ (2) ⇒ (3) ⇒ (4) ⇒ (5) ⇒ (6) ⇒ (7) ⇒ (8) ⇒ (9) ⇒ (-1) null
```

**Available Pool**
- Linked List of Free Nodes
- Deletion performed by list ADT function `AllocateNewNode()`
- Insertion performed by list ADT function `FreeNode()`
- No ordering of available pool list
- Operations must be efficient
  - Head of List — Inserts/Deletes
  - Link field contains index to next available node.
  - Item fields contain ‘leftover’ values which are ignored

---

## Array of Records: Manipulations

### Results of insert(List, “A”);

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>K</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>
```

### Results of remove(List, “X”);

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>K</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>
```

---

**Results of insert(List, “A”):**

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>K</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>G</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>
```

**Results of remove(List, “X”):**

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>K</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>G</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
</tr>
</tbody>
</table>
```
Linked List Mechanics

Array of Records: Insert

6. LL Mechanics 25

Sorted (ascending) List Insertion Function

// Pre: List is initialized
// Post: List contains the inserted elem in order
// insert()
{
    nodePtr prevPtr, currPtr, newPtr;
    newPtr = AllocateNewNode(); // make new node
    if (newPtr == null) return false; // heap is empty
    nodes[newPtr].Element = elem;
    nodes[newPtr].link = null;
    prevPtr = null;
    currPtr = List;
    while ((currPtr != null) && (elem > nodes[currPtr].Element) { // make new node
        prevPtr = currPtr;
        currPtr = nodes[currPtr].link;
    }
    if (prevPtr == null) { //insert at head or
        prevPtr = currPtr;
        currPtr = nodes[currPtr].link;
    } else { //insert at middle
        nodes[prevPtr].link = newPtr;
        nodes[newPtr].link = currPtr;
    }
    return true; // successful insertion
}

Logic is identical to insert() for pointer representation. Syntax for access to underlying structure has changed.