Linked List
List Definitions

• **Linear relationship**  Each element except the first has a unique predecessor, and each element except the last has a unique successor.

• **Length**  The number of items in a list; the length can vary over time.
Abstract Data Type (ADT)

- A data type whose properties (domain and operations) are specified independently of any particular implementation.
- Logical (or ADT) level: abstract view of the domain and operations.
- Implementation level: specific representation of the structure to hold the data items, and the coding for operations.
4 Basic Kinds of ADT Operations

- **Constructor/Destructor** -- creates/deletes a new instance (object) of an ADT.

- **Transformer** -- changes the state of one or more of the data values of an instance.

- **Observer** -- allows us to observe the state of one or more of the data values of an instance without changing them.

- **Iterator** -- allows us to process all the components in a data structure sequentially.
ADT List Operations

• Transformers
  – MakeEmpty()
  – InsertItem()
  – DeleteItem()

• Observers
  – LengthIs()
  – RetrieveItem()

• Iterators
  – ResetList()
  – GetNextItem()
class List
{
    public:
        void UnsortedType ( ) ;
        int LengthIs ( ) const ;
        void RetrieveItem (ItemType& item,
                               bool& found ) ;
        void InsertItem ( ItemType item ) ;
        void DeleteItem ( ItemType item ) ;
        void ResetList ( ) ;
        void GetNextItem ( ItemType& item ) ;
    private:
        int length ;
        ItemType info[MAX_ITEMS] ;
        int currentPos ;
};
Singly Linked List Diagram

- Pointer to front node
- Data carried in node
- Pointer to next node

Nodes:
- A
- B
- C (NULL)
class Node {
public:
    Node(Item i);

private:
    Item item;
    Node *link;
};
void ShoppingList::insert(const Item & anItem, int pos) {
    Node *prev, *cur;

    Inserting A at position 0 in empty list
assert(pos >= 0 && pos <= size);
size++;
if (pos == 0) { // Inserting at the front
    cur = front;
}
front = new Node(anItem);
Empty Insert Example (5 of 6)

front->link = cur;

```
size  prev  cur  anItem  pos
0  1  ?  A  0

item  link
 A
```

front->link = cur;

```
size  prev  cur  anItem  pos
1  0  A  A  0

item  link
 A
```
return;
}

```c
size
1
front

item
A
link
```
void ShoppingList::insert(const Item & anItem, int pos) {
    Node *prev,
    *cur;

    Inserting D at position 0

    size 3
    front

    prev cur
    ?  ?

    anItem pos
    D  0

    item link
    A

    item link
    B

    item link
    C
assert(pos >= 0 && pos <= size);
size++;
if (pos == 0) { // Inserting at the front
    cur = front;

Front Insert Example (4 of 6)

```java
front = new Node(anItem);
```
front->link = cur;

```cpp
front->link = cur;
```
Front Insert Example (6 of 6)

```c
return;
}
```
void ShoppingList::insert(const Item & anItem, int pos) {
    Node *prev,
    *cur;

    Inserting D at position 2

    size | front
    3

    prev | cur
    ? | ?

    anItem | pos
    D | 2

    item | link
    A

    item | link
    B

    item | link
    C
assert(pos >= 0 && pos <= size);
size++;
prev = NULL;
cur = front;

![Diagram of middle insert example with nodes A, B, and C, and an item D at position 2.]
while (pos > 0) {
    prev = cur;
    cur = cur->link;
    pos--;
}
while (pos > 0) {
    prev = cur;
    cur = cur->link;
    pos--;
}

Second Iteration
prev->link = new Node(anItem);
prev->link->link = cur;
void ShoppingList::insert(const Item & anItem, int pos) {
    Node *prev, *cur;

    Inserting D at position 3
assert(pos >= 0 && pos <= size);
size++;
prev = 0;
cur = front;
while (pos > 0) {
    prev = cur;
    cur = cur->link;
    pos--;
}

First Iteration
while (pos > 0) {
    prev = cur;
    cur = cur->link;
    pos--;  
}

Second Iteration
while (pos > 0) {
    prev = cur;
    cur = cur->link;
    pos--;
}

Third Iteration

```
size  front
  4

prev  cur  anItem  pos

item   link  item   link  item   link
  A      B      C

```
prev->link = new Node(anItem);
prev->link->link = cur;
void ShoppingList::remove(int pos) {
    Node *cur,
    *prev;

Removing at position 1

Removing at position 1
assert(pos >= 0 && pos < size);
size--;
prev = NULL;
cur = front;
while (pos > 0) {
    prev = cur;
    cur = cur->link;
    pos--;
}

First iteration
Remove Example (5 of 7)

\[ \text{prev->link} = \text{cur->link}; \]
Remove Example (6 of 7)

delete cur;

```
2
prev
A
item
link
B
item
link
pos
0
C
```
Remove Example (7 of 7)
void ShoppingList::deleteAll() {
    Node *kil;
}
while (front != NULL) {
    kil = front;
    front = front->link;
}

*First iteration*
kil->link = NULL;
delete kil;

First iteration
while (front != NULL) {
    kil = front; 
    front = front->link;
}

**Second iteration**
kil->link = NULL;
delete kil;
}

Second iteration
while (front != NULL) {
    kil = front;
    front = front->link;
}

Third iteration
kil->link = 0;
delete kil;

Third iteration
size = 0;
}

size  front
0     0
Copy Constructors and Overloaded = Operators

- Copy constructors used to make copies when passing by value and returning by value in functions
- Operator = used for assignment
- Both have default behaviors if not user-defined, which is to do a bitwise copy of all member data
- The default behavior can be a problem when there are pointers in member data (shallow copying)
- Users should define their own copy constructor and overloaded operator = when they are creating a class with pointer data members
Deep Copying (1 of 3)

- Consider the following two ShoppingList objects
- What would the result of \texttt{slist2} = \texttt{slist1}; be if the default copy behavior took place?

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diagram.png}
\end{figure}
Deep Copying (2 of 3)

- `slist2` becomes a “shallow copy” of `slist1`
- `slist2` doesn’t get a copy of `slist1` data
- All references to `slist2` data are lost (mem. leak)
Deep Copying (3 of 3)

- Here is what `slist2` would look like as a "deep copy" of `slist1`
- Note that `slist2` has its own set of data

```plaintext
slist1

<table>
<thead>
<tr>
<th>size</th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>

slist2

<table>
<thead>
<tr>
<th>size</th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>
```
void ShoppingList::clone(const ShoppingList &slist) {
    Node *cur1,
    *cur2;
}
this->deleteAll();
this->size = slist.size;
this->front = new Node(*(slist.front));
cur1 = this->front;
cur2 = slist.front;
while (cur2->link) {
    cur2 = cur2->link;
    cur1->link = new Node(*cur2);
    cur1 = cur1->link;
}

First iteration
while (cur2->link) {
    cur2 = cur2->link;
    cur1->link = new Node(*cur2);
    cur1 = cur1->link;
}

Second iteration
cur1 = cur2 = 0;
Doubly Linked List

- Each node has 2 pointers, one to the node before and one to the node after
- No longer limited to left-to-right pointer movement
- Typically, a rear pointer is employed to simplify reverse traversals
Circular Linked List

- “Last” node’s link points at the “front” node
- Concept of “front” optional (alternative: “current”)