# CS1020E: DATA STRUCTURES AND ALGORITHMS I

## Tutorial 5 – Linked List

(Week 7, starting 26 September 2016)

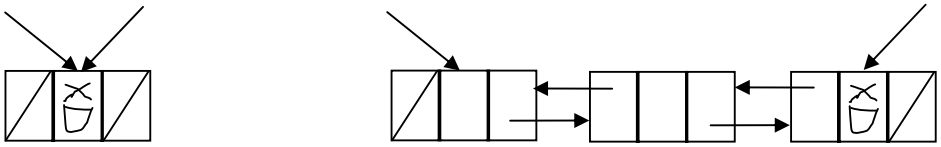
### 1. *Linked List Operations*

Examine the file `T5_mystery.cpp`, which contains a simplified implementation of a doubly linked list with a sentinel (dummy) node at the back. We are trying to simulate the STL `list<T>` data type.

- What is the **purpose of the class** `GuessWhatIsThis`?
- What does **each operation** in `mysteryA..L()` do?
- What is each method's **STL equivalent**, and how is that used?

This question is designed for you to **trace through** each linked list operation, based on the **implementation of each operation**. Only look at the behavior of each operation in main() to check your answers!

Remember, since tutorial 1, we have been asking you to draw diagrams of what happens in memory to trace through what each operation does. For reference-based data structures, it is often important to be able to visualize what each statement does to your data in memory. Use the diagrams below to help you.



**Answer**

**(Class)** This class is the equivalent of the STL iterator. You should be able to deduce this from the attributes and methods of the iterator, and from the way it is used.

Conceptually, the iterator 'points' to an element, but is able to move from one node to another. STL:

```
list<char>::iterator itr = list.begin();
```

**(a)** `empty()` **(b)** `size()` **(c)** `front()` **(d)** `back()`

front() and back() do not return a node, but returns a reference to the element instead, so that any changes allow the element to be modified directly. STL:

```
list.back() = 'X';
```

back() here does not return the element in the node pointed to by `tail`, because that node is a sentinel.

**(e)** `push_front(T&)` **(f)** `push_back(T&)`

Adds an element before the first existing element, or after the last existing element. The sentinel node does not count. You should be able to deduce this from the fact that the number of nodes is increased, and that a new node is created.

**(g)** `pop_front()` **(h)** `pop_back()`

Removes first / last element. You should be able to deduce this from the fact that the number of nodes is decreased, and that the first / second last node is destroyed. Don't forget that `next` and `prev` pointers are not the only ones that need to be adjusted - the `head` pointer has to too.

**(i)** `begin()` **(j)** `end()` **(k)** `insert(iterator, T&)` **(l)** `erase(iterator)`

Each of these 4 methods deal with iterators. Don't forget that `end()` returns one 'pointing' to past the last element, i.e. in this implementation, pointing to the sentinel node. `insert()` inserts the element before the 'current' one, while `erase()` invalidates any iterators pointing to the removed node. STL:

```
for (list<char>::iterator itr = list.begin(); itr != list.end(); itr++)
     if (*itr == 'Y') list.insert(itr, 'X'); // replace all Y with XY
     if (*itr == 'Z') itr = list.erase(itr); // remove Z, update itr
```

`list<T>` iterator is not a RandomAccessIterator, so (`itr + x`) and (`itr1 < itr2`) are not allowed.

## *2. More STL List<T> Operations - Online Discussion*

How is each of the following list method used, and what does each do to the nodes in a linked list?

```
void merge(list<T>& other);
void resize(int newSize); // C++ 11 overloaded version
void reverse();
void splice(...); // The overload with the shortest signature
```

You can try writing code to implement those functions if you like, and explain them on the Facebook discussion group **after** the tutorials on Friday...!

**Answer**

`merge()` requires `*this` and `other` to already be sorted. One element from each list is compared with one from the other, and the smaller ends up in front. Eventually, `*this` contains all elements while `other` has 0 elements, and the list ends up with elements in order. List nodes themselves are not created/destroyed, so all iterators are still valid, but iterate on `*this`.

`splice(iterator, list& other)` efficiently appends the elements in one list to the other, without creating/destroying list nodes. To be efficient, we should not look at every element in the other list.

The other 2 methods are self-explanatory. `resize()` can either shrink or expand the list, destroying/creating nodes.

- Now halfway through! ☻ -

Think of various cases
Design full algo before coding