Linked Lists

Singly Linked List

- push_front- Make the new node the head pointer, and make it point to the previous head node
- push_back- Make the tail node point to the new back node, and the back node point to null
- pop_front- Make the next node, following the head node, the new head node
- pop_back- Make the node before the tail node point to null, rather than the tail node

> *Efficiently = O(1) time complexity*

Doubly Linked List

- Contains a reference to the next element, as well as a reference to the previous element

```
// Insertion
auto I = Cities.begin();

for (; I != Cities.end(); ++I) {

        if (“Miami” == *I) {
        break;
        }
}

//Insert the new string

Cities.insert(I, “Orlando”);

// “Jacksonville”, “Tallahassee”, “Gainesville”, “Orlando”, “Miami”

// Remove Orlando
List<string>::iterator I = Cities.begin();

// auto I = Cities.begin();  // c++11
while( I != Cities.end()) {
if (“Orlando” == *I) {
        I = Cities.erase(I);
} else {
        I++;
}}
```

Node -

- Data Value
- Pointers to the previous and next element
- Defined within the List class, with limited scope

Creating A List

```
template <typename Object>

class List

{
  private:
    struct Node
    {
        Object  data;
        Node   *prev; // Points to previous Node
        Node   *next; // Points to next Node

        Node( const Object & d = Object{ }, Node * p = nullptr, Node * n =
nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };
```

Insertion within List

```
iterator insert( iterator itr, const Object & x )

    {

        Node *p = itr.current;

        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ x, p->prev, p
} );

    }



  iterator insert( iterator itr, Object && x )

    {

        Node *p = itr.current;

        ++theSize;

        return iterator( p->prev = p->prev->next = new Node{ std::move( x
), p->prev, p } );

    }
```

Empty List

```
  private:
    int   theSize;
    Node *head;
    Node *tail;
    void init( )
        // Doubly Linked List Init
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail; // Head -> &Tail
        tail->prev = head; // Tail (previos) -> &Head
    }

};
```

Erase Node

```
iterator erase( iterator itr )
```

```
   {

       Node *p = itr.current;

       iterator retVal( p->next );

       p->prev->next = p->next;

       p->next->prev = p->prev;

       delete p;

       --theSize;

       return retVal;
   }

iterator erase( iterator from, iterator to )

   {
       for( iterator itr = from; itr != to; )
           itr = erase( itr );
       return to;
   }
```