

# Linked List

Textbook Ch 10.2

# Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- Application

# List ADT

An Abstract List (or List ADT) is linearly ordered data

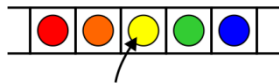
$$( A_1 A_2 \dots A_{n-1} A_n )$$

- The same value may occur more than once

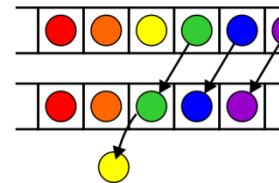
# Operations

Operations at the  $k^{\text{th}}$  entry of the list include:

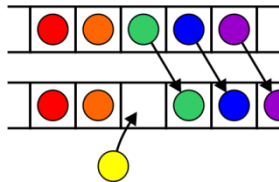
Access to the object



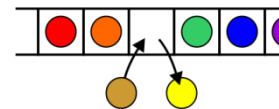
Erasing an object



Insertion of a new object

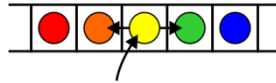


Replacement of the object



# Operations

Given access to the  $k^{\text{th}}$  object, gain access to either the previous or next object



Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

# Abstract Strings

A specialization of an Abstract List is an Abstract String:

- The entries are restricted to *characters* from a finite *alphabet*
- This includes regular strings, e.g., “Hello world!”

The restriction using an alphabet emphasizes specific operations that would seldom be used otherwise

- Substrings, matching substrings, string concatenations

It also allows more efficient implementations

- String searching/matching algorithms
- Regular expressions

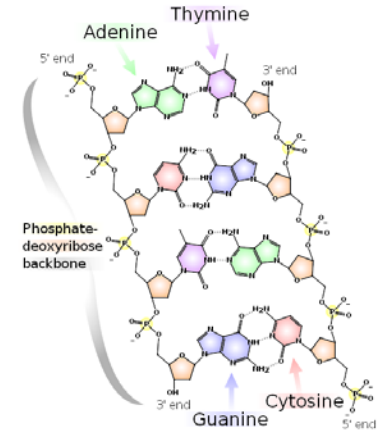
# Abstract Strings

Strings also include DNA

- The alphabet has 4 *characters*: A, C, G, and T
- These are the nucleobases:  
adenine, cytosine, guanine, and thymine

Bioinformatics today uses many of the algorithms traditionally restricted to computer science:

- Dan Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge, 1997  
<http://books.google.ca/books?id=STGlisyqtjYMC>
- References:  
<http://en.wikipedia.org/wiki/DNA>  
<http://en.wikipedia.org/wiki/Bioinformatics>



# Arrays



	Accessing the $k^{\text{th}}$ entry	Front	Insert or erase at the $k^{\text{th}}$ entry	Back
Arrays	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$



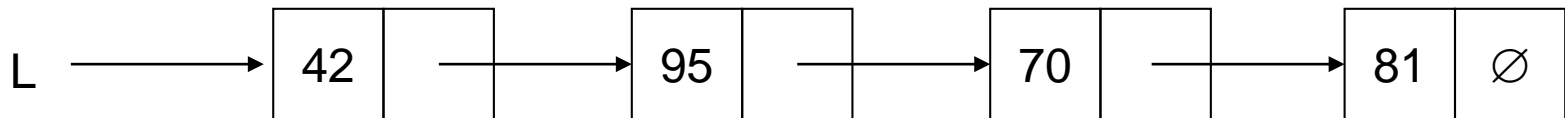
# Outline

- List ADT
- **Linked list**
- Doubly linked list
- Node-based storage with arrays
- Application

# Definition

A linked list is a data structure where each object is stored in a *node*

As well as storing data, the node must also contains a reference/pointer to the node containing the next item of data



# Node Class

The node must store **data** and a **pointer**:

```
class Node {  
    private:  
        int element;  
        Node *next_node;  
    public:  
        Node( int = 0, Node * = nullptr );  
  
        int retrieve() const;  
        Node *next() const;  
};
```

# Node Constructor

The constructor assigns the two member variables based on the arguments

```
Node::Node( int e, Node *n ):  
    element( e ),  
    next_node( n ) {  
        // empty constructor  
    }
```

The default values are given in the class definition:

```
Node( int = 0, Node * = nullptr );
```

# Accessors

The two member functions are accessors which simply return the **element** and the **next\_node** member variables, respectively

```
int Node::retrieve() const {  
    return element;  
}
```

```
Node *Node::next() const {  
    return next_node;  
}
```

# Linked List Class

Because each node in a linked lists refers to the next, the linked list class need only link to the first node in the list

The linked list class requires member variable: a pointer to a node

```
class List {  
    private:  
        Node *list_head;  
        // ...  
};
```

# Structure

Let us look at the internal representation of a linked list

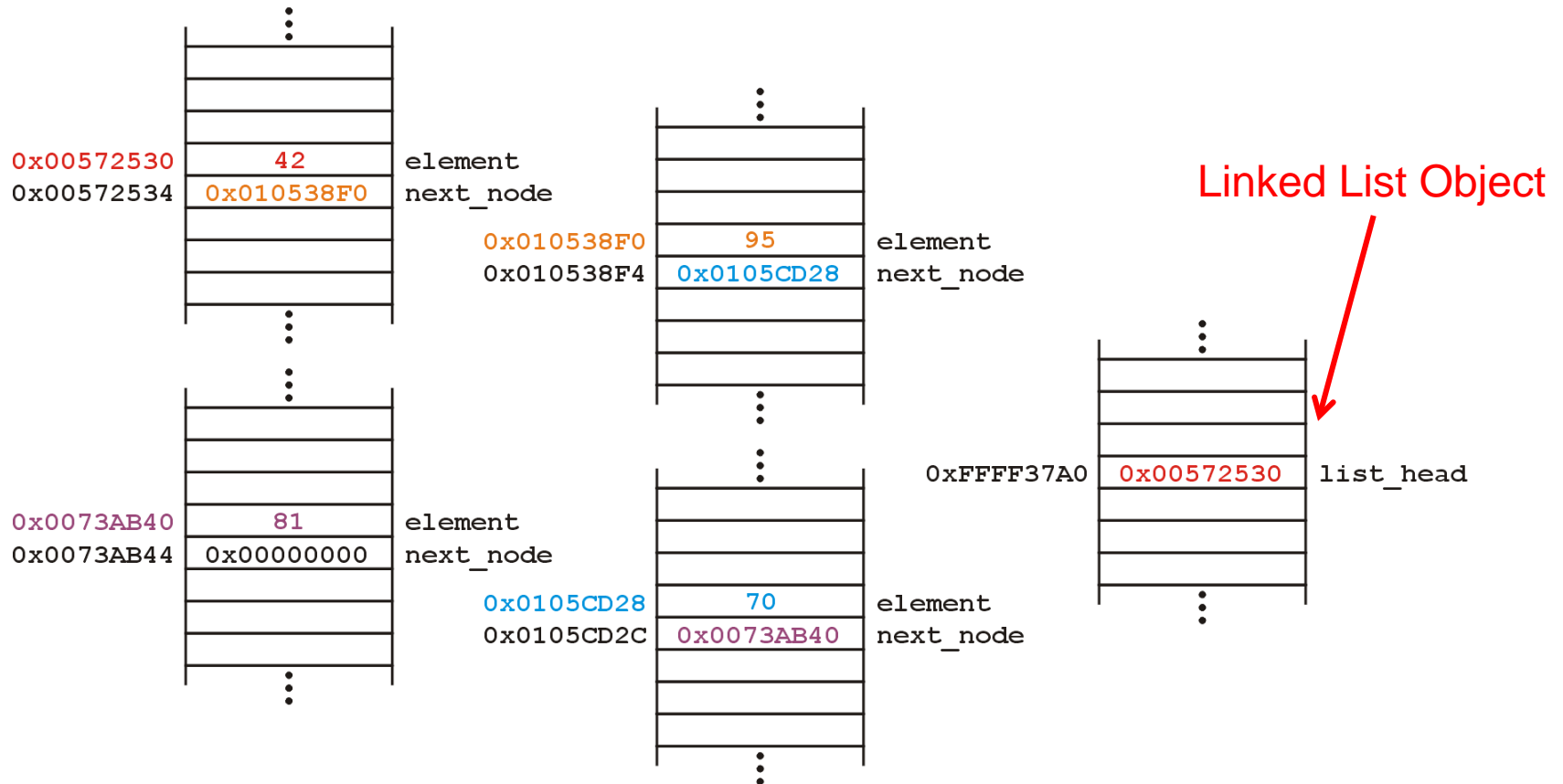
Suppose we want a linked list to store the values

42 95 70 81

in this order

# Structure

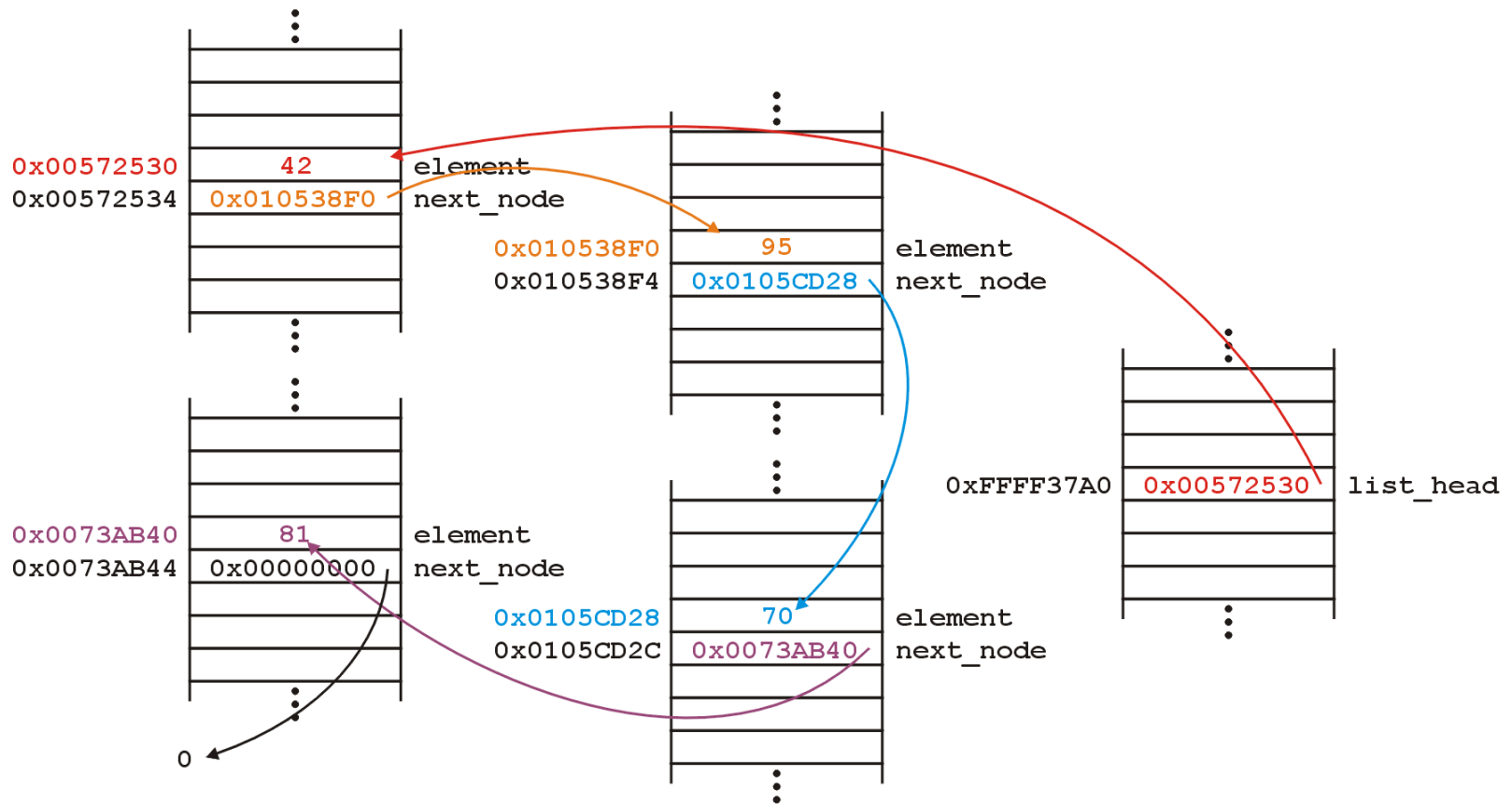
A linked list uses linked allocation, and therefore each node may appear anywhere in memory:





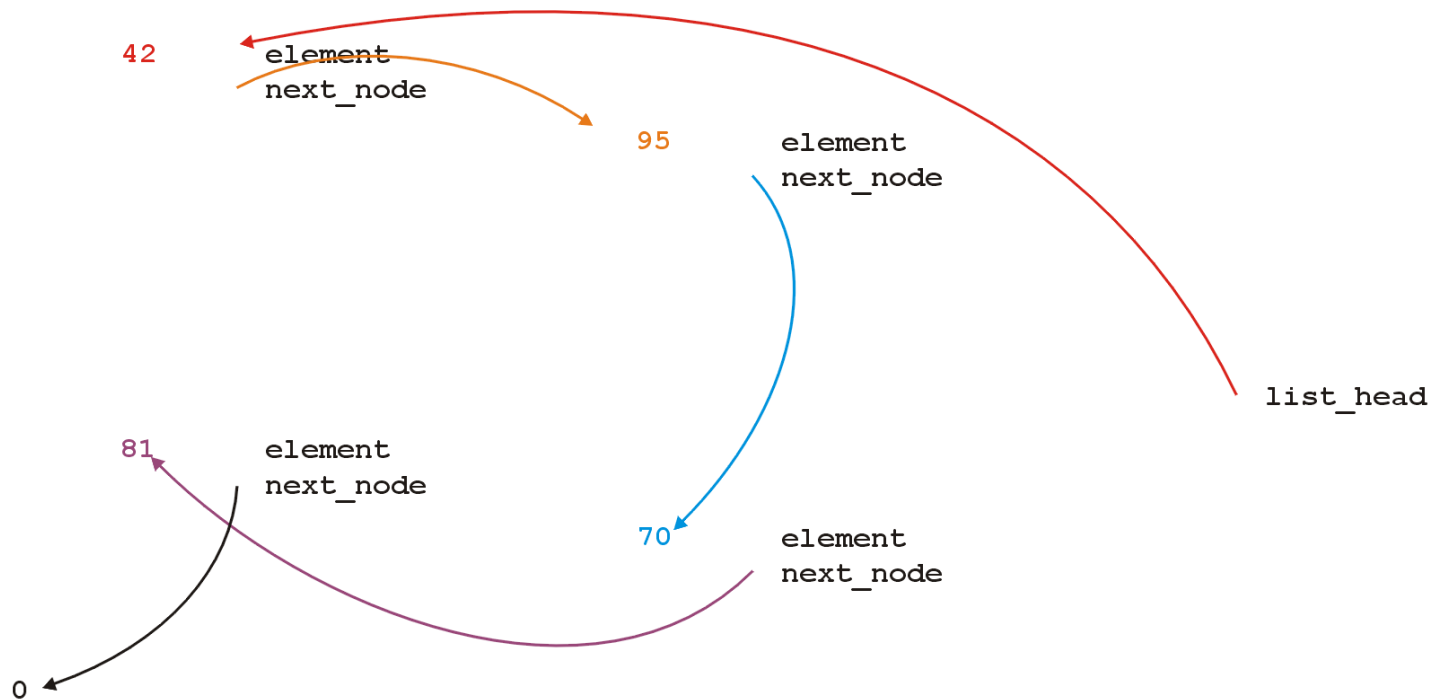
# Structure

The **next\_node** pointers store the addresses of the next node in the list



# Structure

Because the addresses are arbitrary, we can remove that information:



# Structure

We will clean up the representation as follows:



We do not specify the addresses because they are arbitrary and:

- The contents of the circle is the element
- The `next_node` pointer is represented by an arrow

# Operations

First, we want to create a linked list

We also want to be able to:

- insert into,
- access, and
- erase from

the elements stored in the linked list

# Operations

We can do them with the following operations:

- Adding, retrieving, or removing the value at the front of the linked list

```
void push_front( int );
```

```
int front() const;
```

```
void pop_front();
```

- We may also want to access the head of the linked list

```
Node *head() const;
```

# Operations

All these operations relate to the first node of the linked list

We may want to perform operations on an arbitrary node of the linked list, for example:

- Find the number of instances of an integer in the list:

```
int count( int ) const;
```

- Remove all instances of an integer from the list:

```
int erase( int );
```

# Linked Lists

Additionally, we may wish to check the state:

- Is the linked list empty?

```
bool empty() const;
```

- How many objects are in the list?

```
int size() const;
```

The list is empty when the `list_head` pointer is set to `nullptr`

# The Constructor

In the constructor, we assign `list_head` the value `nullptr`

```
List::List():list_head( nullptr ) {  
    // empty constructor  
}
```

We will always ensure that when a linked list is empty, the list head is assigned `nullptr`



# bool empty() const

Starting with the easier member functions:

```
bool List::empty() const {  
    if ( list_head == nullptr ) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

Better yet:

```
bool List::empty() const {  
    return ( list_head == nullptr );  
}
```

## Node \*head() const

The member function `Node *head() const` is easy enough to implement:

```
Node *List::head() const {  
    return list_head;  
}
```

This will always work: if the list is empty, it will return `nullptr`

```
int front() const
```

To get the first element in the linked list, we must access the node to which the `list_head` is pointing

Because we have a pointer, we must use the `->` operator to call the member function:

```
int List::front() const {  
    return head()->retrieve();  
}
```

```
int front() const
```

What if the list is empty?

If we tried to access a member function of a pointer set to `nullptr`, we would access restricted memory and the OS would terminate the running program

`int front() const`

Thus, the full function is

```
int List::front() const {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    return head()->retrieve();  
}
```

# int front() const

Why is `empty()` better than

```
int List::front() const {  
    if ( list_head == nullptr ) {  
        throw underflow();  
    }  
  
    return list_head->element;  
}
```

Two benefits:

- More readable
- If the implementation changes we do nothing

```
void push_front( int )
```

Next, let us add an element to the list

If it is empty, we start with:

`list_head`  $\longrightarrow$  0

and, if we try to add 81, we should end up with:

`list_head`  $\longrightarrow$  (81)  $\longrightarrow$  0

```
void push_front( int )
```

We must:

- create a new node which:
  - stores the value **81**, and
  - is pointing to **0**
- assign its address to `list_head`

We can do this as follows:

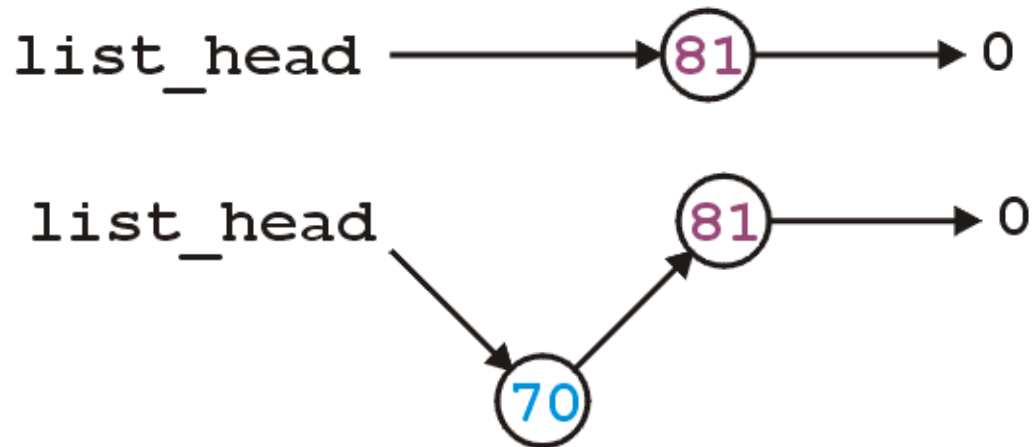
```
list_head = new Node( 81, nullptr );
```



```
void push_front( int )
```

Suppose however, we already have a non-empty list

Adding **70**, we want:



```
void push_front( int )
```

To achieve this, we must we must create a new node which:

- stores the value `70`, and
  - is pointing to the current list head
- we must then assign its address to `list_head`

We can do this as follows:

```
list_head = new Node( 70, list_head );
```

```
void push_front( int )
```

Thus, our implementation could be:

```
void List::push_front( int n ) {  
    if ( empty() ) {  
        list_head = new Node( n, nullptr );  
    } else {  
        list_head = new Node( n, head() );  
    }  
}
```

```
void push_front( int )
```


We could, however, note that when the list is empty, `list_head == 0`, thus we could shorten this to:

```
void List::push_front( int n ) {  
    list_head = new Node( n, list_head );  
}
```

# void push\_front( int )

Are we allowed to do this?

```
void List::push_front( int n ) {  
    list_head = new Node( n, head() );  
}
```



Yes: the right-hand side of an assignment is evaluated first

- The original value of `list_head` is accessed first before the function call is made

# int pop\_front()

Erasing from the front of a linked list is even easier:

- We assign the list head to the next pointer of the first node

Graphically, given:



we want:



# int pop\_front()

Easy enough:

```
int List::pop_front() {  
    int e = front();  
    list_head = head()->next();  
    return e;  
}
```

Unfortunately, we have some problems:

- The list may be empty
- We still have the memory allocated for the node containing 70

# int pop\_front()

Does this work?

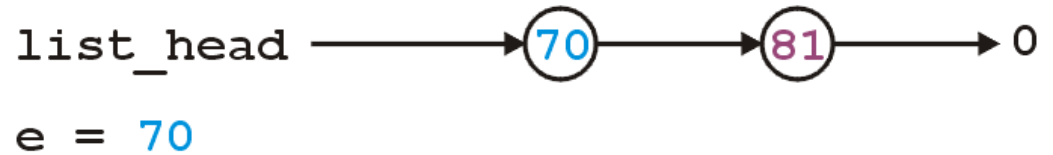
```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    delete head();  
    list_head = head()->next();  
    return e;  
}
```



# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }
```

```
    int e = front();
```



```
    delete head();
```

```
    list_head = head()->next();
```

```
    return e;
```

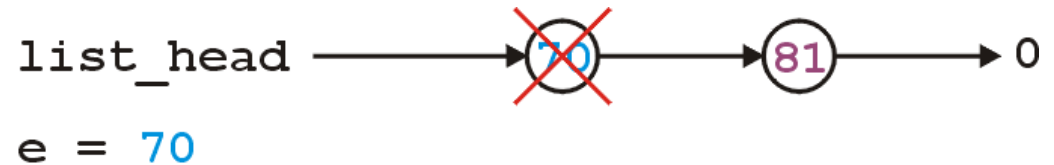
```
}
```

# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }
```

```
    int e = front();
```

```
    delete head();
```



```
    list_head = head()->next();
```

```
    return e;
```

```
}
```

# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }
```

```
    int e = front();
```

```
    delete head();
```

```
    list_head = head()->next();
```

```
    return e;
```

```
}
```

list\_head

e = 70



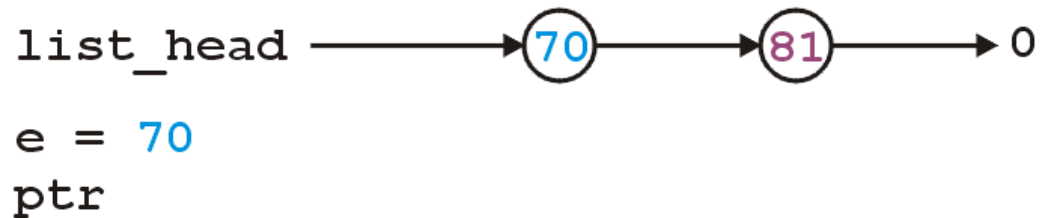
# int pop\_front()

The correct implementation assigns a temporary pointer to point to the node being deleted:

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
    Node *ptr = list_head;  
    list_head = list_head->next();  
    delete ptr;  
    return e;  
}
```

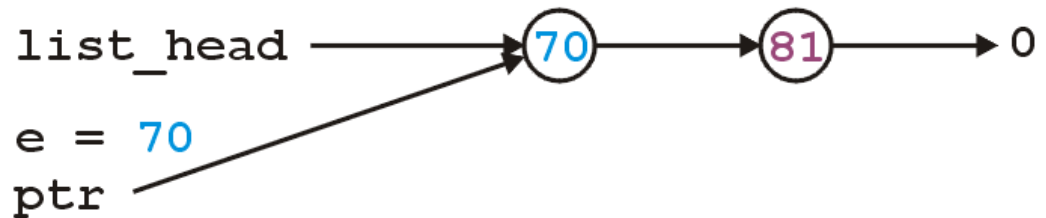
# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
  
    Node *ptr = head();  
  
    list_head = head()->next();  
  
    delete ptr;  
  
    return e;  
}
```



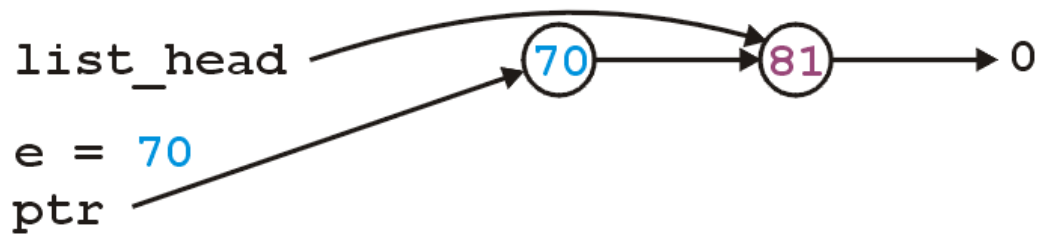
# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
  
    Node *ptr = head();  
  
    list_head = head()->next();  
  
    delete ptr;  
  
    return e;  
}
```



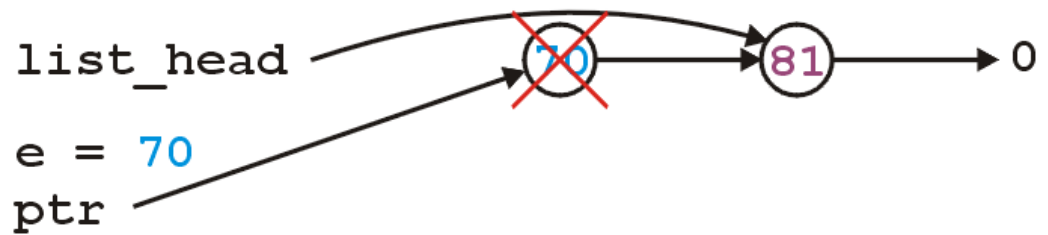
# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
  
    Node *ptr = head();  
  
    list_head = head()->next();  
  
    delete ptr;  
  
    return e;  
}
```



# int pop\_front()

```
int List::pop_front() {  
    if ( empty() ) {  
        throw underflow();  
    }  
  
    int e = front();  
  
    Node *ptr = head();  
  
    list_head = head()->next();  
  
    delete ptr;  
  
    return e;  
}
```





# Stepping through a Linked List

The next step is to look at member functions which potentially require us to step through the entire list:

```
int size() const;  
int count( int ) const;  
int erase( int );
```

The second counts the number of instances of an integer, and the last removes the nodes containing that integer

# Stepping through a Linked List

The process of stepping through a linked list can be thought of as being analogous to a for-loop:

- We initialize a temporary pointer with the list head
- We continue iterating until the pointer equals `nullptr`
- With each step, we set the pointer to point to the next object

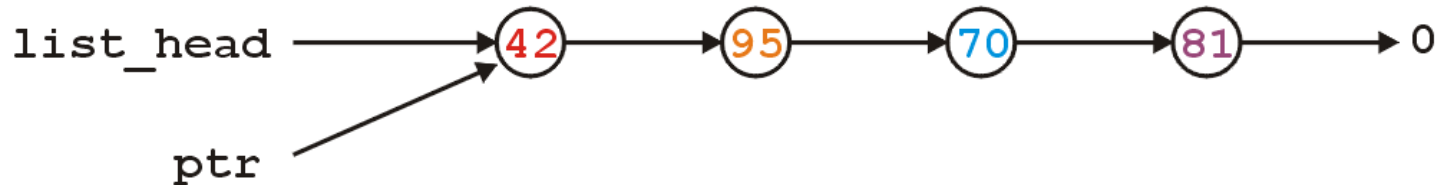
# Stepping through a Linked List

Thus, we have:

```
for ( Node *ptr = head(); ptr != nullptr; ptr = ptr->next() ) {  
    // do something  
    // use ptr->fn() to call member functions  
    // use ptr->var to assign/access member variables  
}
```

# Stepping through a Linked List

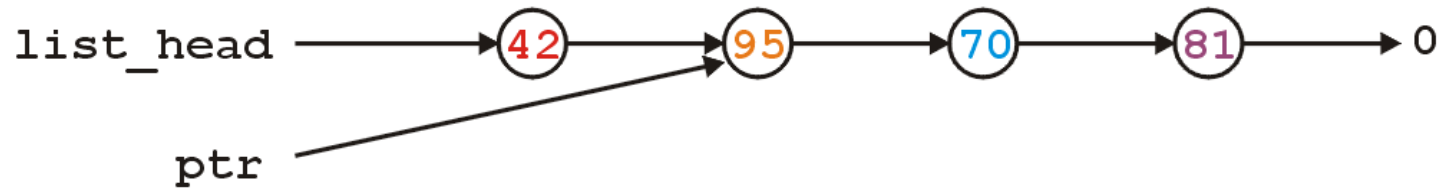
With the initialization and first iteration of the loop, we have:



`ptr != nullptr` and thus we evaluate the body of the loop and then set `ptr` to the next pointer of the node it is pointing to

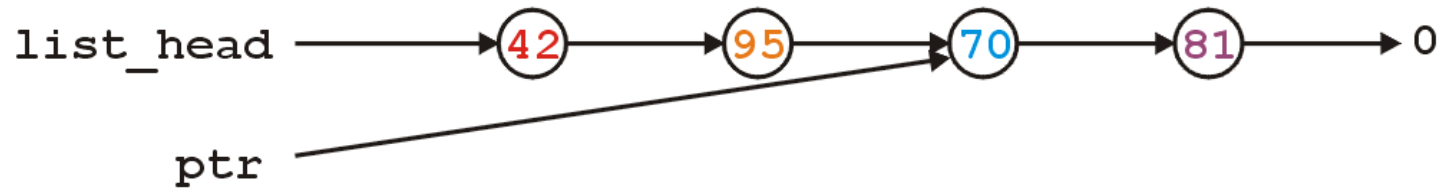
# Stepping through a Linked List

`ptr != nullptr` and thus we evaluate the loop and increment the pointer



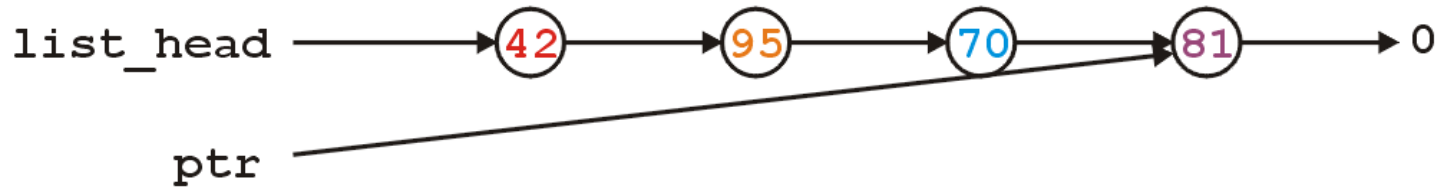
# Stepping through a Linked List

`ptr != nullptr` and thus we evaluate the loop and increment the pointer



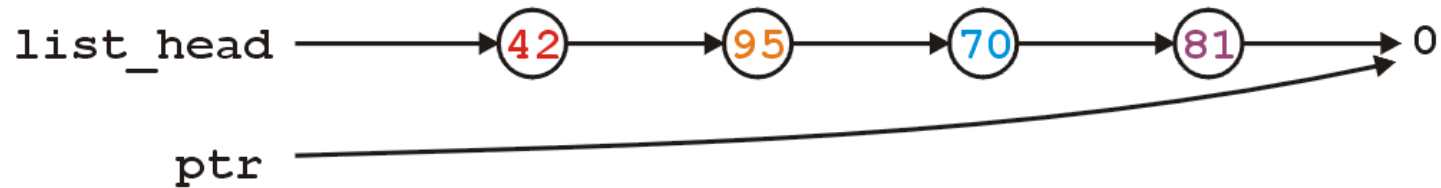
# Stepping through a Linked List

`ptr != nullptr` and thus we evaluate the loop and increment the pointer



# Stepping through a Linked List

Here, we check and find `ptr != nullptr` is false, and thus we exit the loop





```
int count( int ) const
```

To implement `int count(int) const`, we simply check if the argument matches the element with each step

- Each time we find a match, we increment the count
- When the loop is finished, we return the count
- The size function is simplification of count

# int count( int ) const

The implementation:

```
int List::count( int n ) const {  
    int node_count = 0;  
  
    for ( Node *ptr = list(); ptr != nullptr; ptr = ptr->next() ) {  
        if ( ptr->retrieve() == n ) {  
            ++node_count;  
        }  
    }  
  
    return node_count;  
}
```

# int erase( int )

To remove an arbitrary element, *i.e.*, to implement  
`int erase( int )`, we must update the previous node

For example, given



if we delete 70, we want to end up with



# Accessing Private Member Variables

Notice that the `erase` function must modify the member variables of the node prior to the node being removed

Thus, it must have access to the member variable `next_node`

We could supply the member function

```
void set_next( Node * );
```

however, this would be globally accessible

Possible solutions:

- Friends
- Nested classes
- Inner classes (Java/C#)

# Destructor

We dynamically allocated memory each time we added a new `int` into this list

Suppose we delete a list before we remove everything from it

- This would leave the memory allocated with no reference to it



# Destructor

The destructor has to delete any memory which had been allocated but has not yet been deallocated

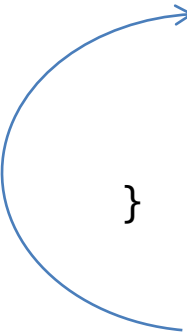
This is straight-forward enough:

```
while ( !empty() ) {  
    pop_front();  
}
```

# Modifying Arguments

You want to pass a copy of a linked list to a function—where the function may modify the copy, but the original list shall be unchanged.

```
void func( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```



With the *default copy constructor*, all the member variables are simply copied over into the new instance of the class

# Modifying Arguments

```
void send_copy( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```

```
int main() {  
    List prim;  
  
    for ( int i = 2; i <= 4; ++i ) {  
        prim.push_front( i*i );  
    }  
  
    send_copy( prim );  
  
    std::cout << prim.empty() << std::endl;  
  
    return 0;  
}
```

First, the list `prim` is created and three elements are pushed onto it





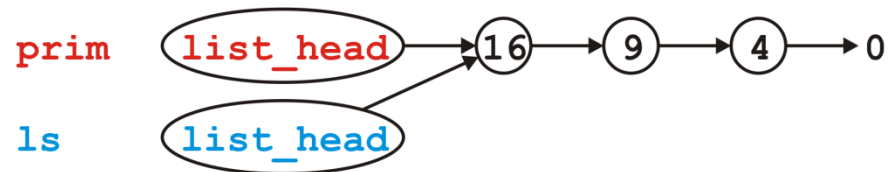
# Modifying Arguments

```
void send_copy( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```

```
int main() {  
    List prim;  
  
    for ( int i = 2; i <= 4; ++i ) {  
        prim.push_front( i*i );  
    }  
  
    send_copy( prim );  
  
    std::cout << prim.empty() << std::endl;  
  
    return 0;  
}
```

Next, we call `send_copy` and assigns a copy of `prim` to `ls`. The default is to copy member variables:

`ls.list_head = prim.list_head`

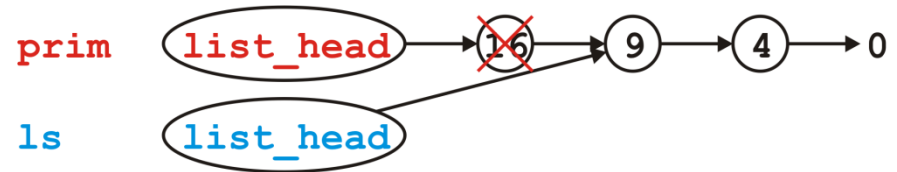


# Modifying Arguments

```
void send_copy( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```

When send\_copy returns, the destructor is called on ls

```
int main() {  
    List prim;  
  
    for ( int i = 2; i <= 4; ++i ) {  
        prim.push_front( i*i );  
    }  
  
    send_copy( prim );  
  
    std::cout << prim.empty() << std::endl;  
  
    return 0;  
}
```

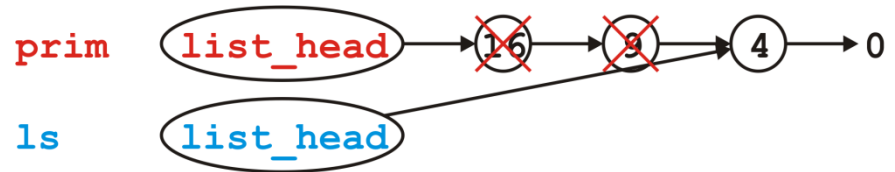


# Modifying Arguments

```
void send_copy( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```

When `send_copy` returns, the destructor is called on `ls`

```
int main() {  
    List prim;  
  
    for ( int i = 2; i <= 4; ++i ) {  
        prim.push_front( i*i );  
    }  
  
    send_copy( prim );  
  
    std::cout << prim.empty() << std::endl;  
  
    return 0;  
}
```

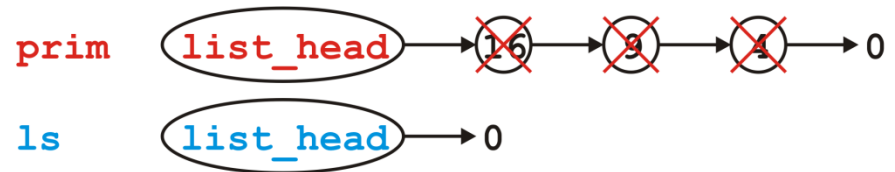


# Modifying Arguments

```
void send_copy( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```

When `send_copy` returns, the destructor is called on `ls`

```
int main() {  
    List prim;  
  
    for ( int i = 2; i <= 4; ++i ) {  
        prim.push_front( i*i );  
    }  
  
    send_copy( prim );  
  
    std::cout << prim.empty() << std::endl;  
  
    return 0;  
}
```



# Modifying Arguments

```
void send_copy( List ls ) {  
    // The compiler creates a new instance and copies the values  
    // The function does something with 'ls'  
    // The compiler ensures the destructor is called on 'ls'  
}
```

```
int main() {  
    List prim;  
  
    for ( int i = 2; i <= 4; ++i ) {  
        prim.push_front( i*i );  
    }
```

```
    send_copy( prim );
```

```
    std::cout << prim.empty() << std::endl;
```

```
    return 0;
```

```
}
```

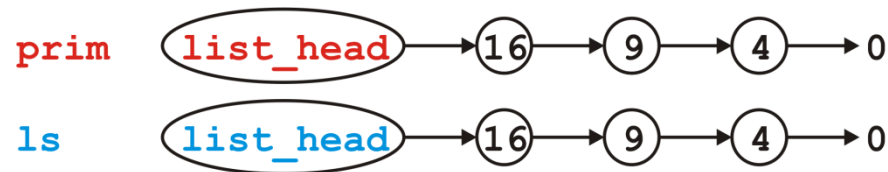
Back in `main()`, `prim.list_head` still stores the address of the Node containing 16, memory that has since been returned to the OS



# Modifying Arguments

What do we really want?

- We really want a copy of the linked list
- If this copy is modified, it leaves the original unchanged

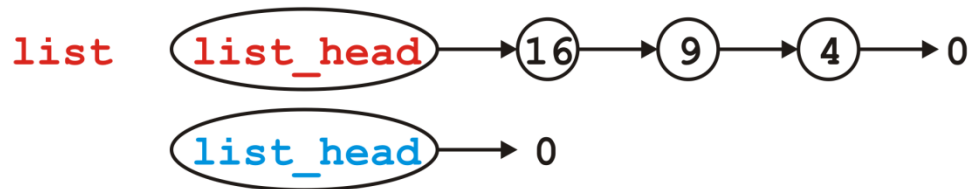


# Copy Constructor

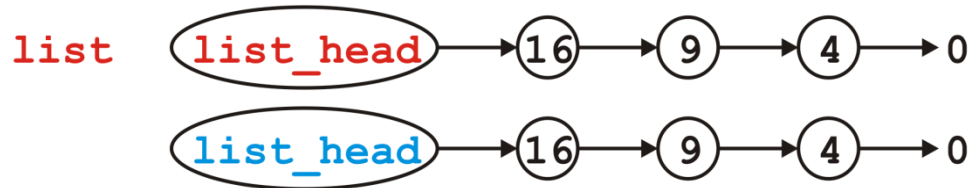
You can modify how copies are made by defining a *copy constructor*

```
List::List( List const &list ):list_head( nullptr ) {  
    // Make a copy of list  
}
```

We now want to go from



to



# Copy Constructor

First, make life simple: if `list` is empty, we are finished, so return

```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
}
```

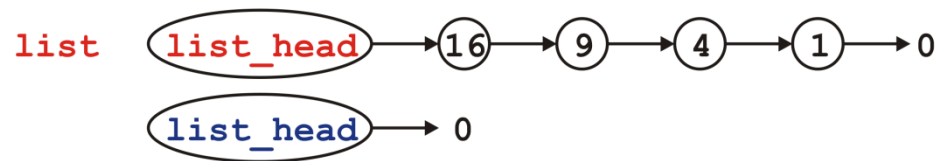
```
}
```



# Copy Constructor

Otherwise, the list being copied is not empty...

```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
}
```

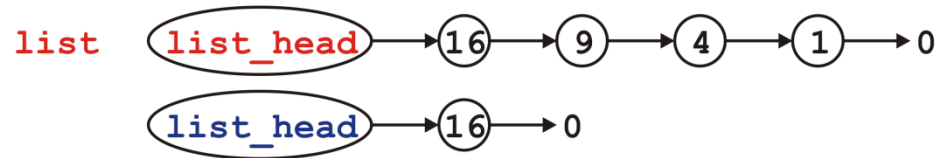


}

# Copy Constructor

Copy the first node—we no longer modifying `list_head`

```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
  
    push_front( list.front() );  
  
}
```

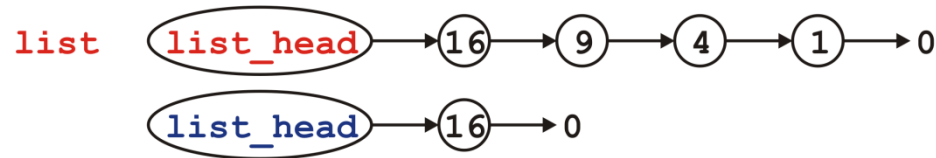


}

# Copy Constructor

We will need to loop through the list... How about a for loop?

```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
  
    push_front( list.front() );
```

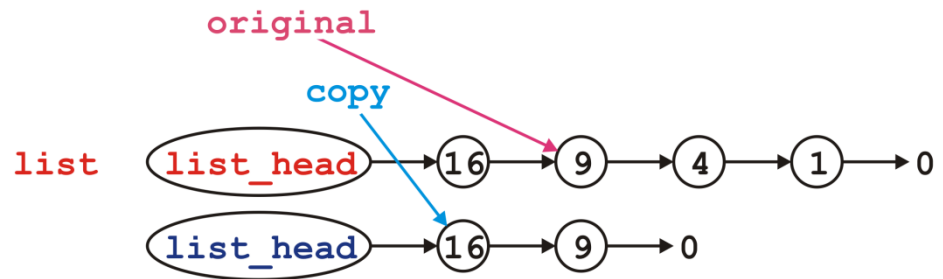


```
}
```

# Copy Constructor

We modify the next pointer of the node pointed to by **copy**

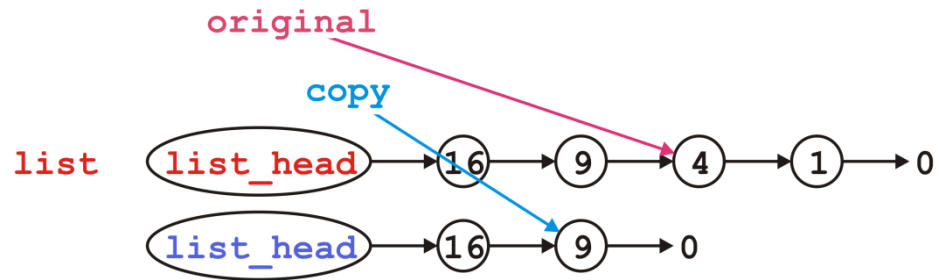
```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
  
    push_front( list.front() );  
  
    for (  
        Node *original = list.head()->next(), *copy = head();  
        original != nullptr;  
        original = original->next(), copy = copy->next()  
    ) {  
        copy->next_node = new Node( original->retrieve(), nullptr );  
    }  
}
```



# Copy Constructor

Then we move each pointer forward:

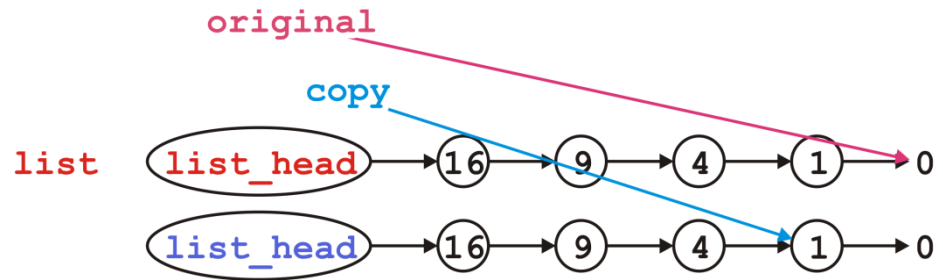
```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
  
    push_front( list.front() );  
  
    for (  
        Node *original = list.head()->next(), *copy = head();  
        original != nullptr;  
        original = original->next(), copy = copy->next()  
    ) {  
        copy->next_node = new Node( original->retrieve(), nullptr );  
    }  
}
```



# Copy Constructor

We'd continue copying until we reach the end

```
List::List( List const &list ):list_head( nullptr ) {  
    if ( list.empty() ) {  
        return;  
    }  
  
    push_front( list.front() );  
  
    for (  
        Node *original = list.head()->next(), *copy = head();  
        original != nullptr;  
        original = original->next(), copy = copy->next()  
    ) {  
        copy->next_node = new Node( original->retrieve(), nullptr );  
    }  
}
```



# Assignment

What about assignment?

- Suppose you have linked lists:

```
List lst1, lst2;
```

```
lst1.push_front( 35 );
```

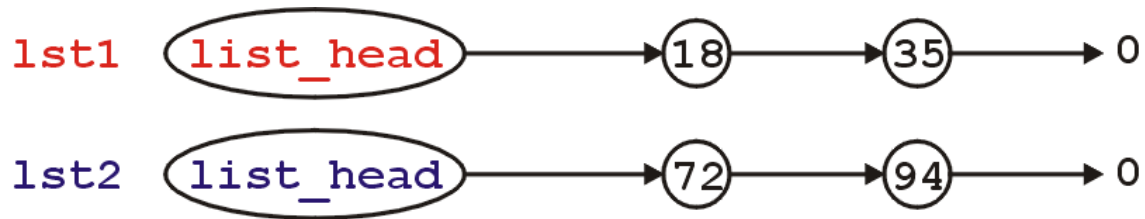
```
lst1.push_front( 18 );
```

```
lst2.push_front( 94 );
```

```
lst2.push_front( 72 );
```

# Assignment

This is the current state:



Consider an assignment:

```
lst2 = lst1;
```

What do we want? What do we actually do?



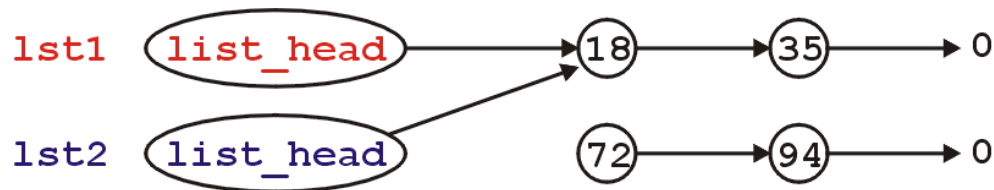
# Assignment

The default behavior: the member variables of this class are copied over

It is equivalent to writing:

```
lst2.list_head = lst1.list_head;
```

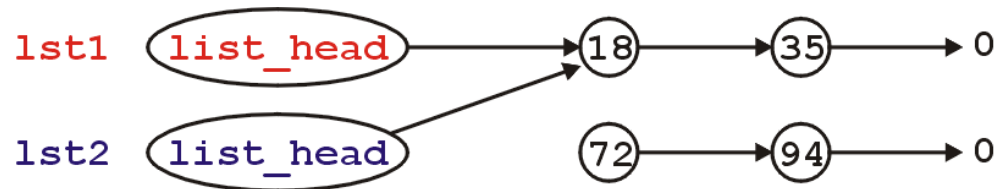
Graphically:



# Assignment

What's wrong with this picture?

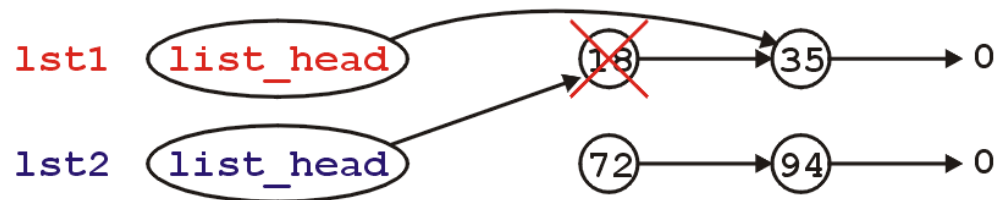
- We no longer have links to either of the nodes storing 72 or 94 (memory leak)



- Also, suppose we call the member function

`lst1.pop_front();`

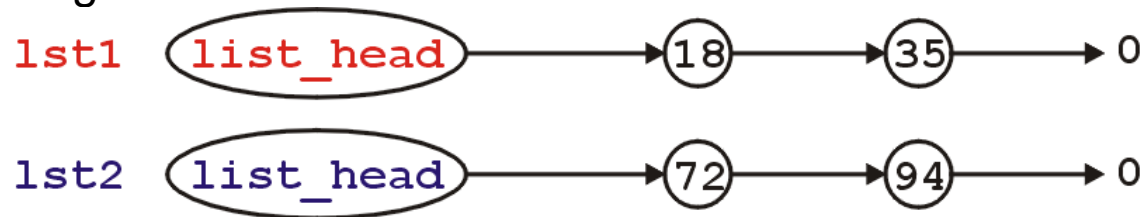
- `lst2` is now invalid



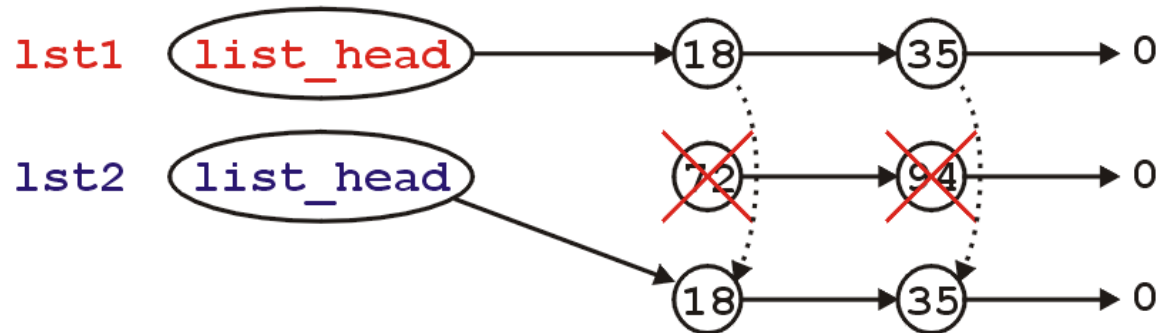
# Assignment

Like making copies, we must have a reasonable means of assigning

- Starting with

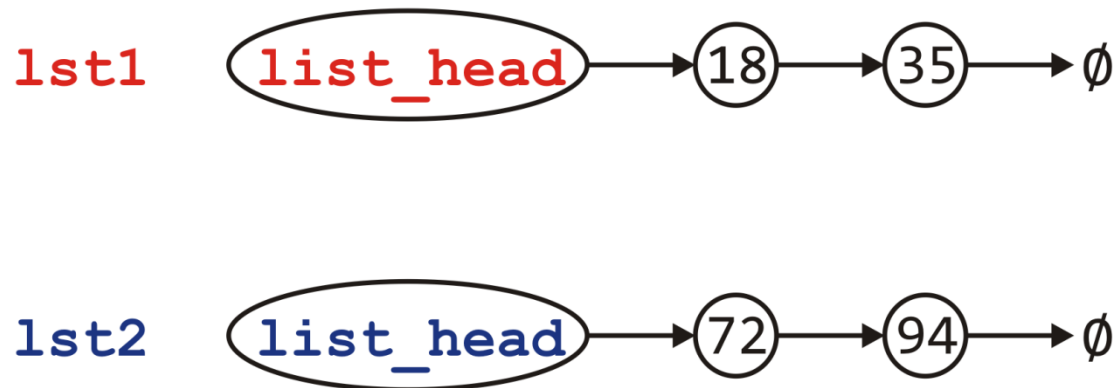


- We need to erase the content of `lst2` and copy over the nodes in `lst1`



# Assignment

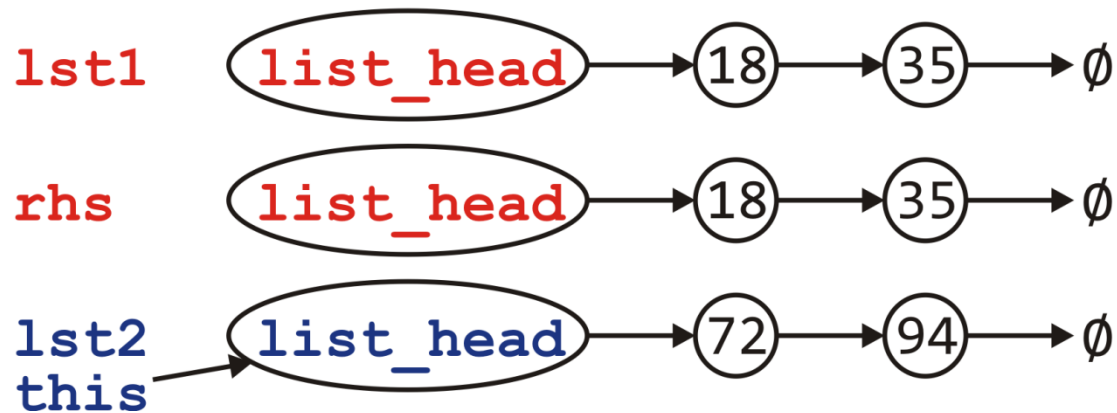
Visually, we are doing the following:



# Assignment

Visually, we are doing the following:

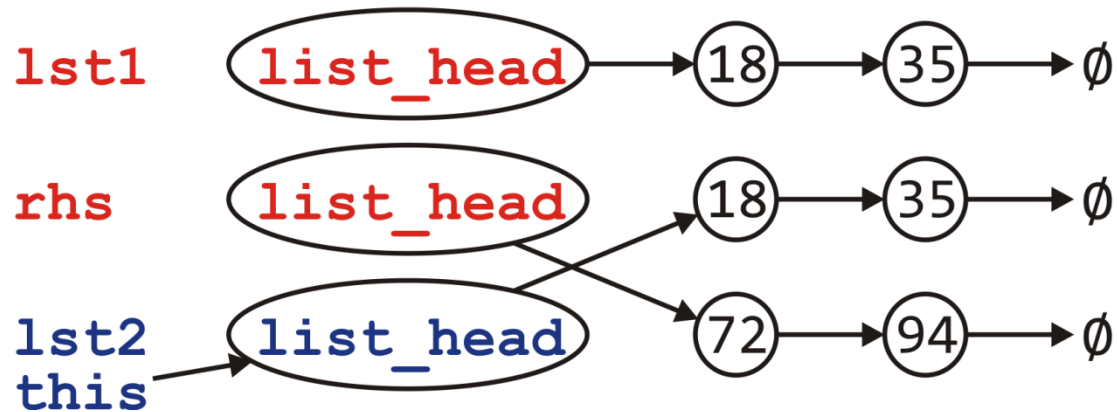
- Call the copy constructor to create rhs



# Assignment

Visually, we are doing the following:

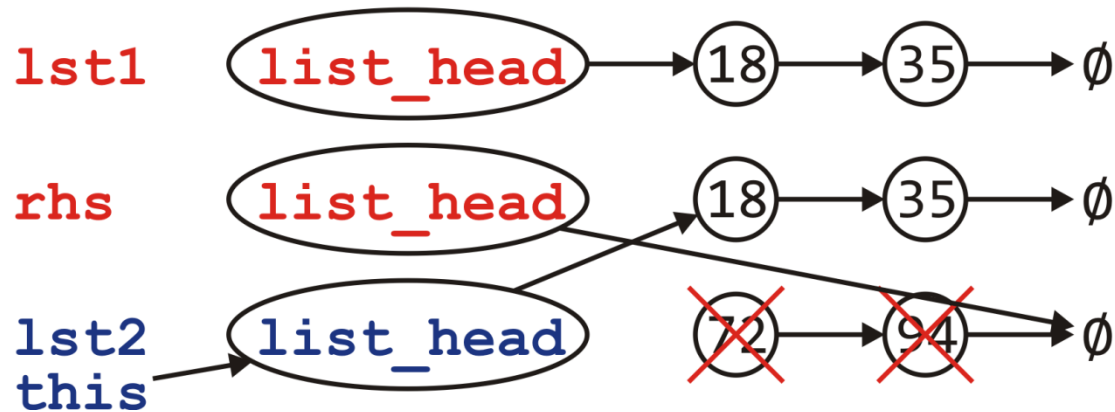
- Call the copy constructor to create rhs
- Swapping the member variables of \*this and rhs



# Assignment

Visually, we are doing the following:

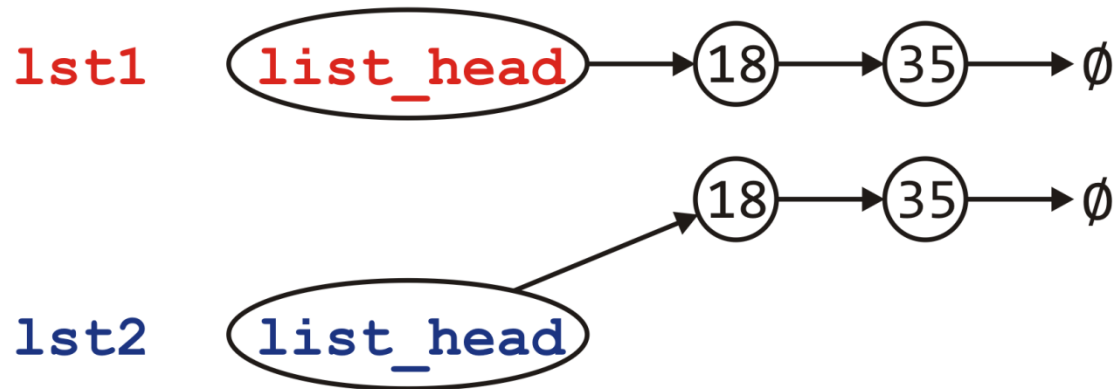
- Call the copy constructor to create rhs
- Swapping the member variables of \*this and rhs
- The destructor is called on rhs



# Assignment

Visually, we are doing the following:

- Call the copy constructor to create rhs
- Swapping the member variables of \*this and rhs
- The destructor is called on rhs



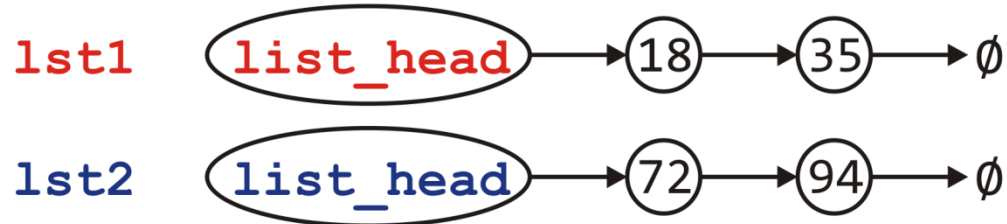


# Assignment

Can we do better?

Consider the calls to new and delete

- Each of these is very expensive...
- Would it not be better to reuse the nodes if possible?

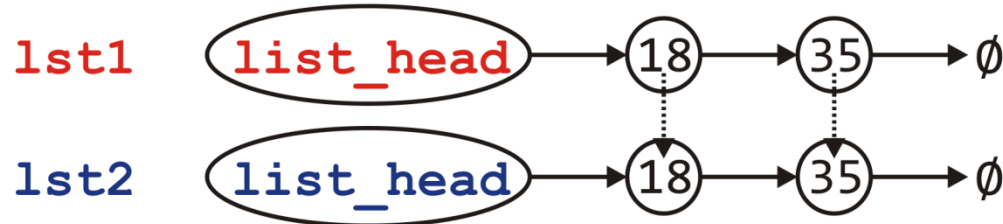


# Assignment

Can we do better?

Consider the calls to new and delete

- Each of these is very expensive...
- Would it not be better to reuse the nodes if possible?



- No calls to new or delete

# Assignment

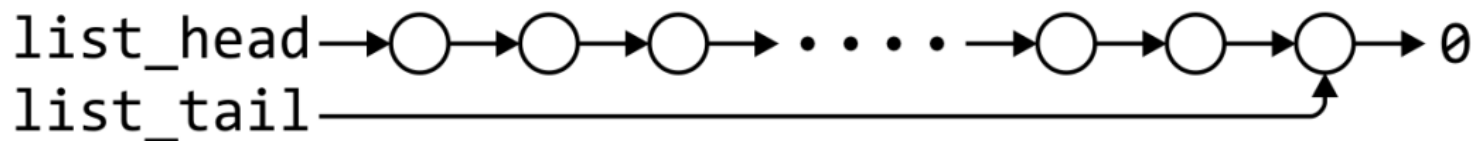
What is the plan?

- If the right-hand side is empty, it's straight-forward:
  - Just empty this list
- Otherwise, step through the right-hand side list and for each node there
  - If there is a corresponding node in this, copy over the value, else
  - There is no corresponding node; create a new node and append it
- If there are any nodes remaining in this, delete them

# Linked list

	Front/1 <sup>st</sup> node	$k^{\text{th}}$ node	Back/ $n^{\text{th}}$ node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

\* These assume we have already accessed the  $k^{\text{th}}$  entry—an  $O(n)$  operation

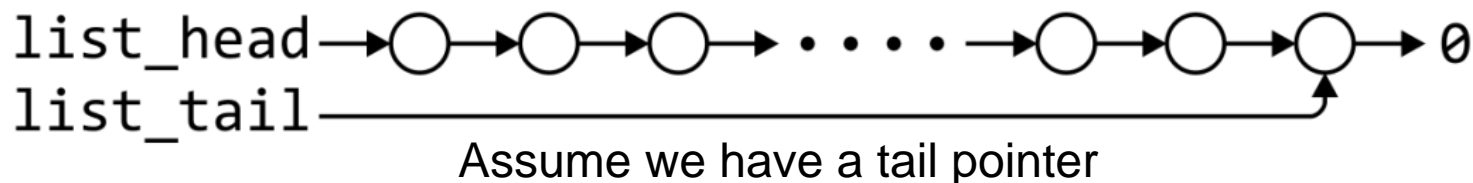


Assume we have a tail pointer

# Linked list

	Front/1 <sup>st</sup> node	$k^{\text{th}}$ node	Back/ $n^{\text{th}}$ node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

By replacing the value in the node in question, we can speed things up



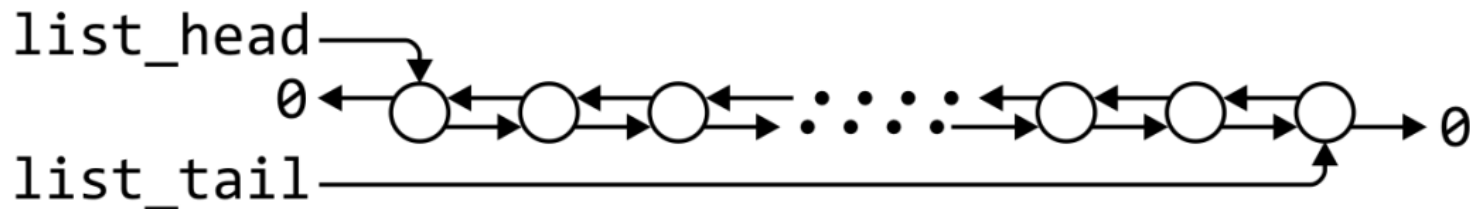
# Outline

- List ADT
- Linked list
- **Doubly linked list**
- Node-based storage with arrays
- Application

# Doubly linked lists

	Front/1 <sup>st</sup> node	$k^{\text{th}}$ node	Back/ $n^{\text{th}}$ node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

\* These assume we have already accessed the  $k^{\text{th}}$  entry—an  $O(n)$  operation



# Memory usage versus run times

Using a doubly linked list requires  $\Theta(n)$  additional memory, but it speeds up many operations

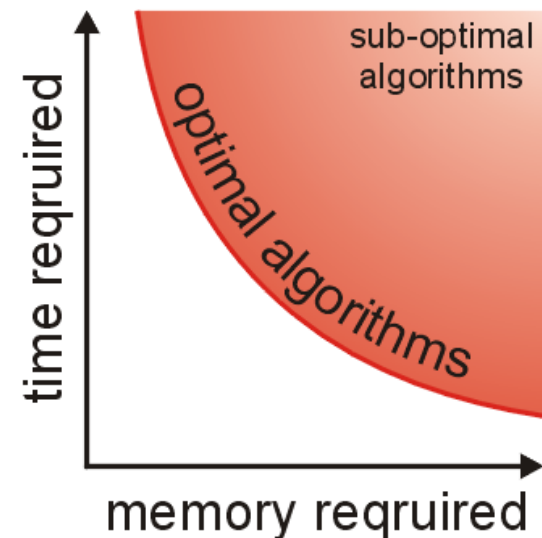


# Memory usage versus run times

In general, there is an interesting relationship between memory and time efficiency

For a data structure/algorithm:

- Improving the run time usually requires more memory
- Reducing the required memory usually requires more run time



# Memory usage versus run times

Warning: programmers often mistake this to suggest that given any solution to a problem, any solution which may be faster must require more memory

This guideline not true in general: there may be different data structures and/or algorithms which are both faster and require less memory

- This requires thought and research

# Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- Application

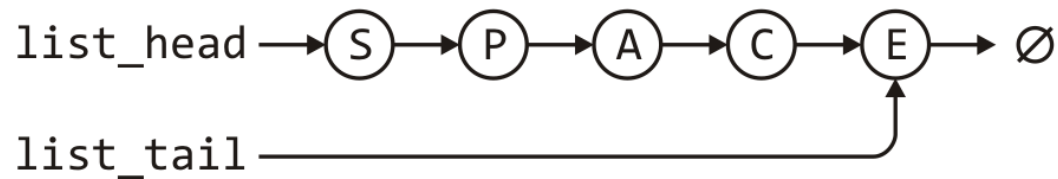
# The issue

A significant issue with linked lists: node-based data structures require  $\Theta(n)$  calls to `new`

- Each `new` operation requires a call to the operating system requesting a memory allocation

# Using an array?

Suppose we store this linked list in an array?



```
list_head = 5;  
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		-1	0		3	2	

# Using an array?

Rather than using, -1, use a constant assigned that value

- This makes reading your code easier

```
list_head = 5;  
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		NULLPTR	0		3	2	

# A solution

To achieve this, we must create an array of objects that:

- Store the value
- Store the array index where the next entry is stored

```
template <typename Type>
class Single_node {
    private:
        Type element;
        int next_node;
    public:
        Type retrieve() const;
        int next() const;
};
```

# A solution

Now, memory allocation is done once in the constructor:

```
template <typename Type>
class Single_list {
    private:
        int list_capacity;
        int list_head;
        int list_tail;
        int list_size;
        Single_node<Type> *node_pool;

        static const int NULL;
    public:
        Single_list( int = 16 );
        // member functions
};
```

```
const int Single_list::NULLPTR = -1;
```

```
template <typename Type>
Single_list<Type>::Single_list( int n ):
    list_capacity( n ),
    list_head( NULLPTR ),
    list_tail( NULLPTR ),
    list_size( 0 ),
    node_pool( new Single_node<Type>[n] ) {
    // Empty constructor
}
```



# A solution

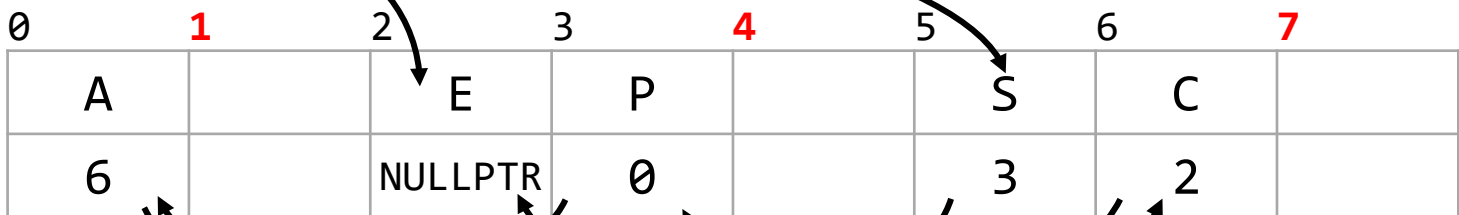
Problem: when inserting a new element...

how do you know which cell to use?

- Solution: keep a container (a stack) of the indices of unused nodes

`list_head = 5;`

`list_tail = 2;`



`stack_size = 3;`



# A solution

The stack would be initialized with all the entries

```
list_head = NULLPTR;  
list_tail = NULLPTR;
```

0	1	2	3	4	5	6	7

```
stack_size = 8;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

When pushing onto the list, the entry at the top of the stack is used

```
list_head = NULLPTR;  
list_tail = NULLPTR;
```

0	1	2	3	4	5	6	7

```
stack_size = 8;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

Now, `push_front( '0' )` would result in

```
list_head = 7;  
list_tail = 7;
```

0	1	2	3	4	5	6	7
							0
							NULLPTR

```
stack_size = 7;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

Suppose we call `push_front( 'N' )`

```
list_head = 7;  
list_tail = 7;
```

0	1	2	3	4	5	6	7
							0
							NULLPTR

```
stack_size = 7;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

Suppose we call `push_front( 'N' )`

- The next node is at index 6

```
list_head = 6;
```

```
list_tail = 7;
```

0	1	2	3	4	5	6	7
						N	0
						7	NULLPTR

```
stack_size = 6;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

Suppose we call `push_back( 'R' )`

```
list_head = 6;  
list_tail = 7;
```

0	1	2	3	4	5	6	7
						N	0
						7	NULLPTR

```
stack_size = 6;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

Suppose we call `push_back( 'R' )`

- The next node is at index 5

```
list_head = 6;
```

```
list_tail = 5;
```

0	1	2	3	4	5	6	7
					R	N	0
					NULLPTR	7	5

```
stack_size = 5;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7



# A solution

Finally, suppose we call `pop_front()`

```
list_head = 6;  
list_tail = 5;
```

0	1	2	3	4	5	6	7
					R	N	0
					NULLPTR	7	5

```
stack_size = 5;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

# A solution

Finally, suppose we call `pop_front()`

- The popped node is placed back into the stack

```
list_head = 7;
```

```
list_tail = 5;
```

0	1	2	3	4	5	6	7
					R	N	0
					NULLPTR	7	5

```
stack_size = 6;
```

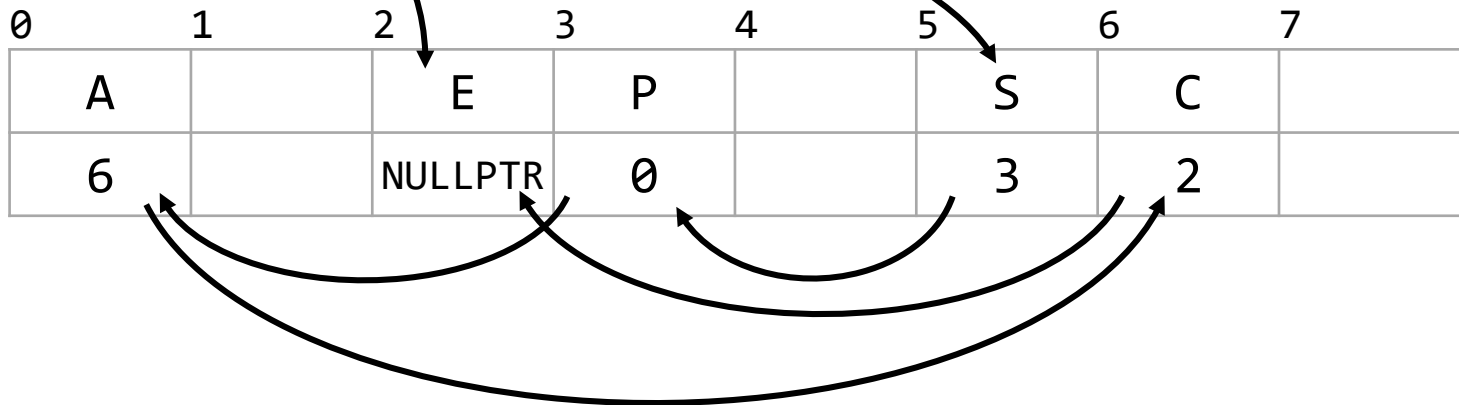
0	1	2	3	4	5	6	7
0	1	2	3	4	6	6	7

# A better solution

## Problem:

- Our solution requires  $\Theta(N)$  additional memory
- In our initial example, the unused nodes are 1, 4 and 7
- How about using these to define a second stack-as-linked-list?

```
list_head = 5;  
list_tail = 2;
```

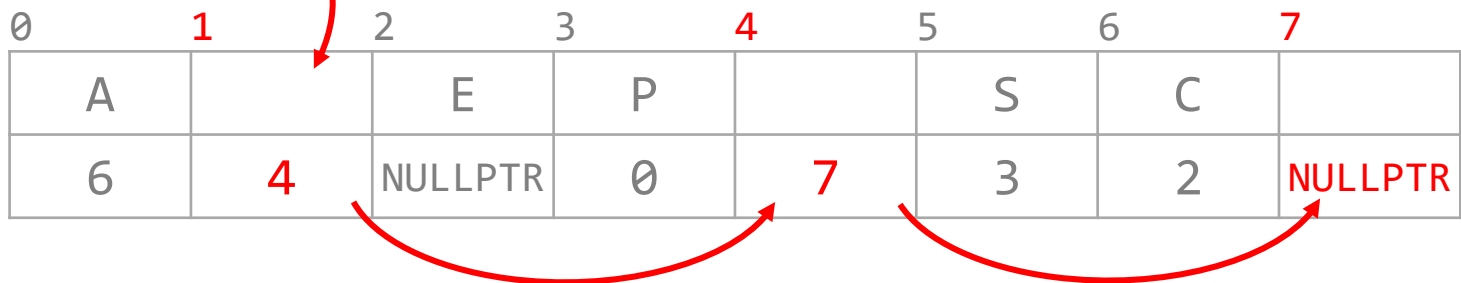


# A better solution

## Problem:

- Our solution requires  $\Theta(N)$  additional memory
- In our initial example, the unused nodes are 1, 4 and 7
- How about using these to define a second stack-as-linked-list?

```
list_head = 5;  
list_tail = 2;  
stack_top = 1;
```



- We only need a head pointer for the stack-as-linked-list

# A better solution

Suppose we call `pop_front()`

```
list_head = 5;  
list_tail = 2;  
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	3	2	NULLPTR

# A better solution

Suppose we call `pop_front()`

- The extra node is placed onto the stack

```
list_head = 3;  
list_tail = 2;  
stack_top = 5;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	1	2	NULLPTR

# A better solution

Suppose we now call `push_back( 'D' )`

```
list_head = 3;  
list_tail = 2;  
stack_top = 5;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	1	2	NULLPTR

# A better solution

Suppose we now call `push_back( 'D' )`

- We pop the node off of the top of the stack

```
list_head = 3;  
list_tail = 5;  
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		D	C	
6	4	5	0	7	NULLPTR	2	NULLPTR



# A better solution

Suppose we finally call `pop_front()` again

```
list_head = 3;  
list_tail = 5;  
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		D	C	
6	4	5	0	7	NULLPTR	2	NULLPTR

# A better solution

Suppose we finally call `pop_front()` again

- The node containing 'P' is pushed back onto the stack

```
list_head = 0;  
list_tail = 5;  
stack_top = 3;
```

0	1	2	3	4	5	6	7
A		E	P		D	C	
6	4	5	1	7	NULLPTR	2	NULLPTR

# A better solution

In this case, our data structure would be initialized to:

```
list_head = NULLPTR;  
list_tail = NULLPTR;  
stack_top = 0;
```

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	NULLPTR

# A solution

Our class would look something like:

```
template <typename Type>
class Single_list {
    private:
        int list_head;
        int list_tail;
        int list_size;
        int list_capacity;
        Single_node<Type> *node_pool;
        int stack_top;

        static const int NULL;
    public:
        Single_list( int = 16 );
        // member functions
};

const int Single_list::NULLPTR = -1;
```

```
template <typename Type>
Single_list<Type>::Single_list( int n ):
    list_head( NULLPTR ),
    list_tail( NULLPTR ),
    list_size( 0 ),
    list_capacity( n ),
    node_pool( new Single_node<Type>[n] ),
    stack_top( 0 ) {
    for ( int i = 1; i < n; ++i ) {
        node_pool[i - 1].next = i;
    }

    node_pool[n - 1] = NULLPTR;
}
```

# Analysis

This solution:

- Requires only three more member variable than our linked list class
- It still requires  $O(N)$  additional memory over an array
- All the run-times are identical to that of a linked list
- Only one call to new, as opposed to  $\Theta(n)$
- There is a potential for up to  $O(N)$  wasted memory

Question: What happens if we run out of memory?

# Reallocation of memory

Suppose we start with a capacity  $N$  but after a while, all the entries have been allocated

- We can double the size of the array and copy the entries over

```
list_head = 6;  
list_tail = 4;  
list_size = 8;  
list_capacity = 8;  
stack_top = NULLPTR;
```

0	1	2	3	4	5	6	7
C	R	U	T	R	U	S	T
7	2	0	1	NULLPTR	4	3	5

# Reallocation of memory

Suppose we start with a capacity  $N$  but after a while, all the entries have been allocated

- We can double the size of the array and copy the entries over
- Only the stack needs to be updated and the old array deleted

```
list_head = 6;  
list_tail = 4;  
list_size = 8;  
list_capacity = 16;  
stack_top = 8;
```

0	1	2	3	4	5	6	7
C	R	U	T	R	U	S	T
7	2	0	1	NULLPTR	4	3	5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T								
7	2	0	1	NULLPTR	4	3	5	9	10	11	12	13	14	15	NULLPTR

# Reallocation of memory

Now push\_back( 'E' ) would use the next location

```
list_head = 6;  
list_tail = 4;  
list_size = 8;  
list_capacity = 16;  
stack_top = 8;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T								
7	2	0	1	NULLPTR	4	3	5	9	10	11	12	13	14	15	NULLPTR



# Reallocation of memory

Now push\_back( 'E' ) would use the next location

```
list_head = 6;  
list_tail = 8;  
list_size = 9;  
list_capacity = 16;  
stack_top = 9;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T	E							
7	2	0	1	8	4	3	5	NULLPTR	10	11	12	13	14	15	NULLPTR

# Reallocation of memory

If at some point, we decide it is desirable to reduce the memory allocated, it might be easier to just insert the entries into a newer and smaller table

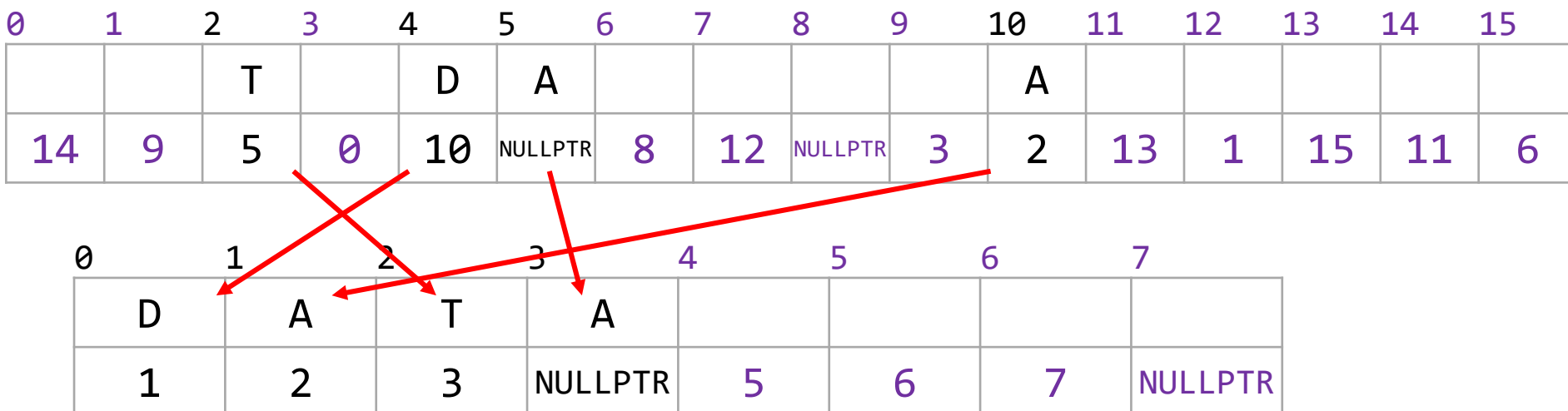
```
list_head = 4;  
list_tail = 5;  
list_size = 4;  
list_capacity = 16;  
stack_top = 7;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		T		D	A					A					
14	9	5	0	10	NULLPTR	8	12	NULLPTR	3	2	13	1	15	11	6

# Reallocation of memory

If at some point, we decide it is desirable to reduce the memory allocated, it might be easier to just insert the entries into a newer and smaller table

```
list_head = 4;  
list_tail = 5;  
list_size = 4;  
list_capacity = 16;  
stack_top = 7;
```



# Reallocation of memory

If at some point, we decide it is desirable to reduce the memory allocated, it might be easier to just insert the entries into a newer, and smaller table

- Now, delete the old array and update the member variables

```
list_head = 0;  
list_tail = 3;  
list_size = 4;  
list_capacity = 8;  
stack_top = 4;
```

0	1	2	3	4	5	6	7
D	A	T	A				
1	2	3	NULLPTR	5	6	7	NULLPTR

# Outline

- List ADT
- Linked list
- Doubly linked list
- Node-based storage with arrays
- **Application**

# Polynomial

- Possible linked list implementation
  - $A_i$  is the coefficient of the  $x^{i-1}$  term

$$5 + 2x + 3x^2$$

( 5 2 3 )

$$7 + 8x$$

( 7 8 )

$$3 + x^2$$

( 3 0 2 )

Problem?

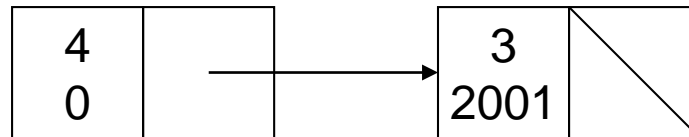
$$4 + 3x^{2001}$$

[illegible]

# Sparse Vector Data Structure:

$$4 + 3x^{2001}$$

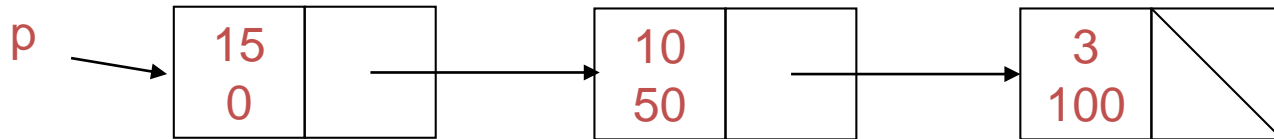
(~~<4 0>~~    <2001 3>)



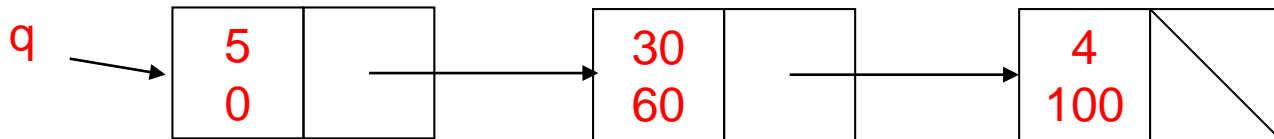


# Addition of Two Polynomials?

$$15+10x^{50}+3x^{100}$$

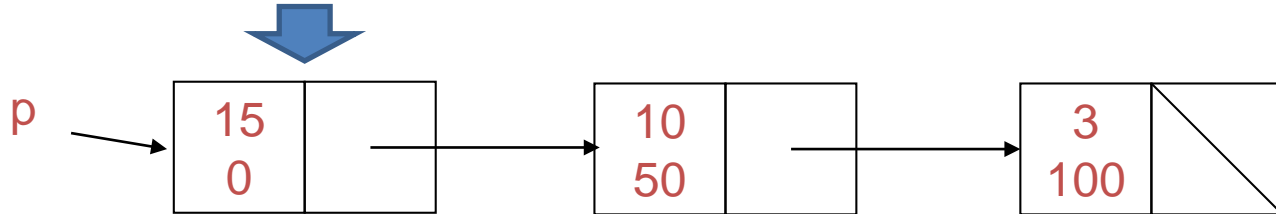


$$5+30x^{60}+4x^{100}$$

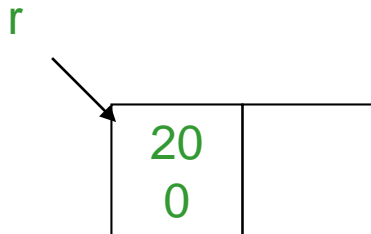
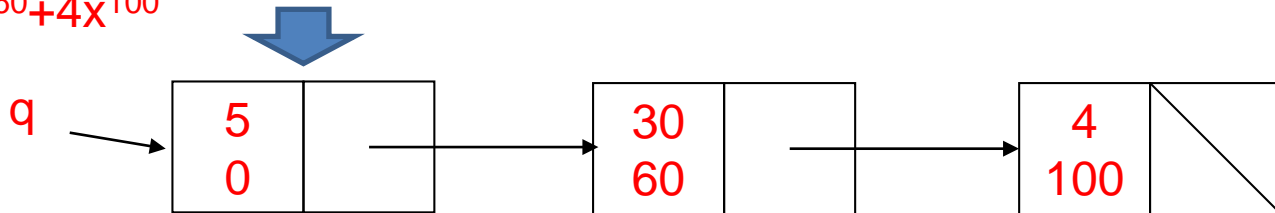


# Addition of Two Polynomials

$$15 + 10x^{50} + 3x^{100}$$

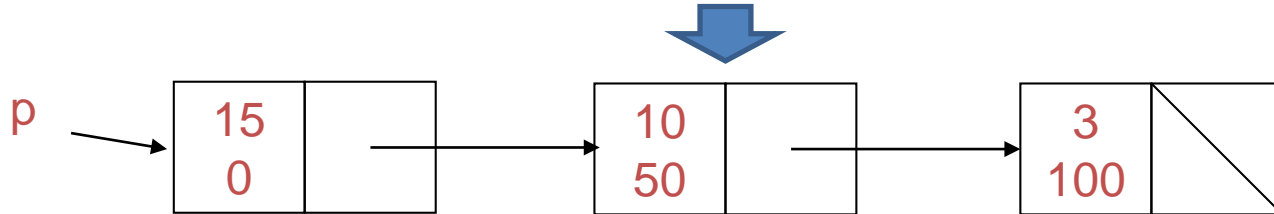


$$5 + 30x^{60} + 4x^{100}$$

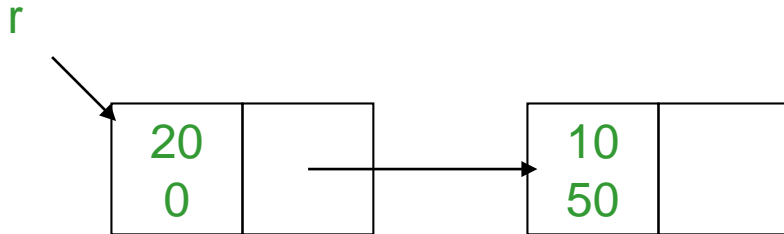
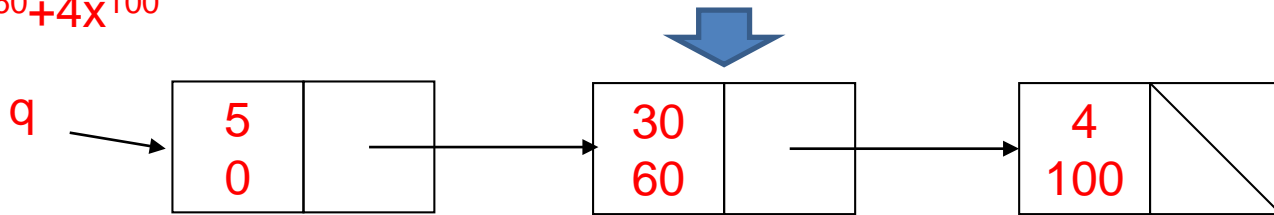


# Addition of Two Polynomials

$$15+10x^{50}+3x^{100}$$

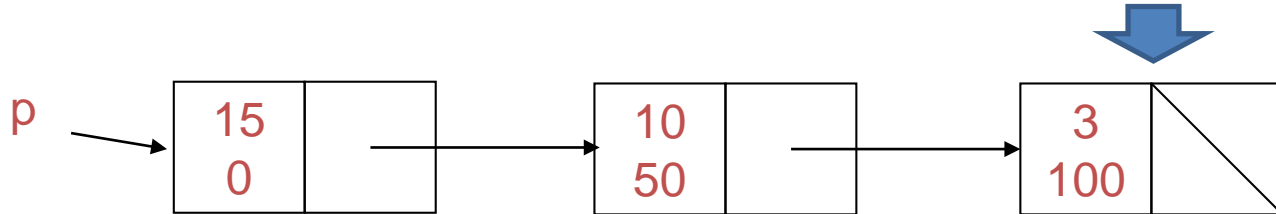


$$5+30x^{60}+4x^{100}$$

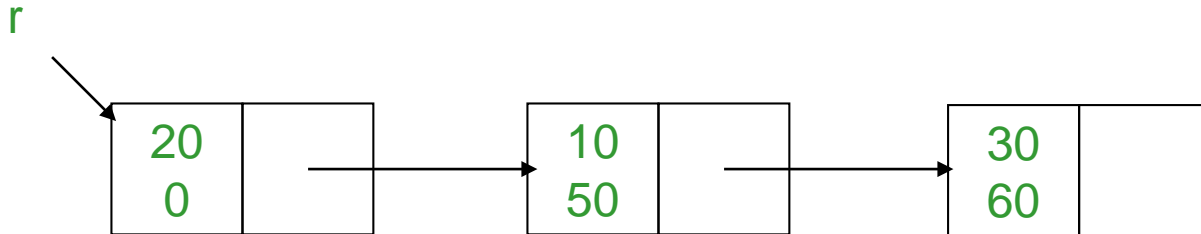
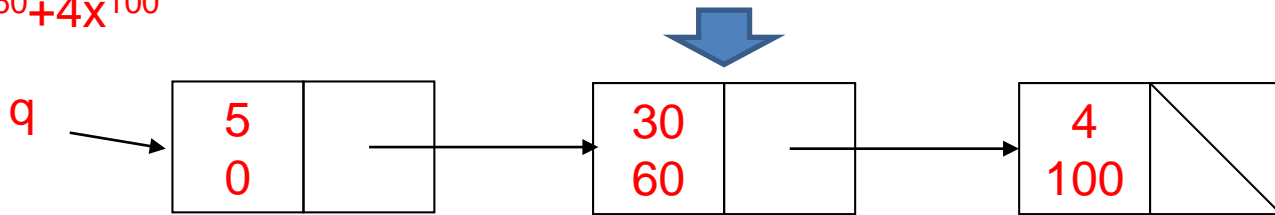


# Addition of Two Polynomials

$$15+10x^{50}+3x^{100}$$



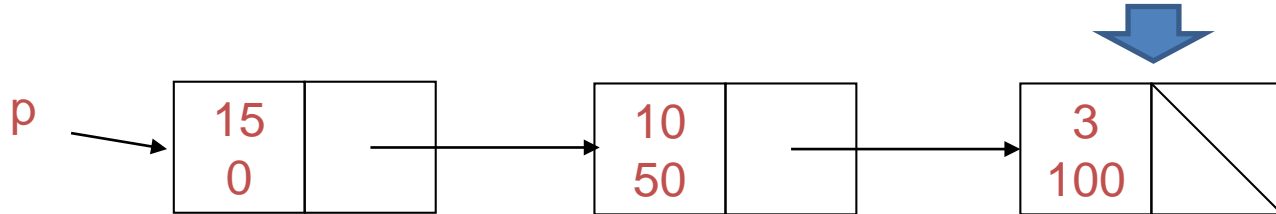
$$5+30x^{60}+4x^{100}$$



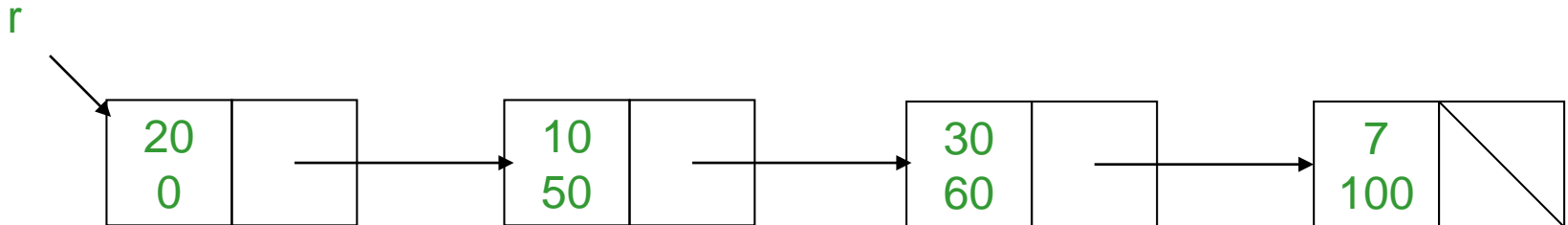
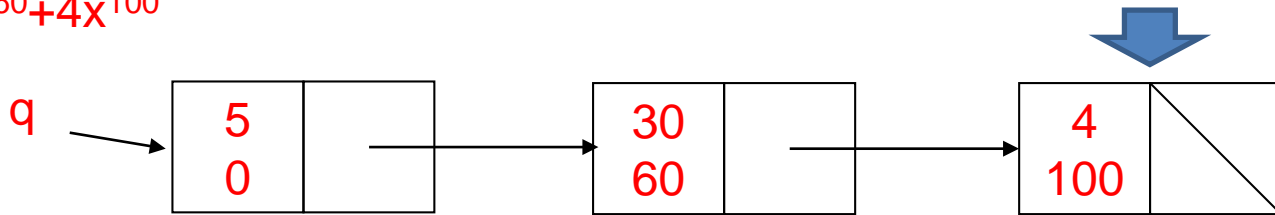
# Addition of Two Polynomials

- One pass down each list:  $\Theta(n + m)$

$$15 + 10x^{50} + 3x^{100}$$

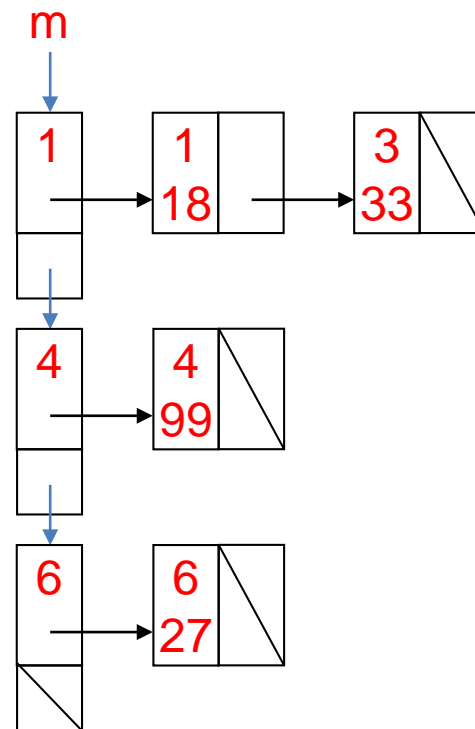


$$5 + 30x^{60} + 4x^{100}$$



# Sparse Matrices

18	0	33	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	99	0	0
0	0	0	0	0	0
0	0	0	0	0	27



# Summary

- List ADT
  - A sequence of elements (special case: string)
  - Array
- Linked list
  - Accessors and mutators
  - Stepping through a linked list
  - Copy and assignment operator
- Doubly linked list
  - Memory usage versus run times
- Node-based storage with arrays
  - No longer need to call `new` for each new node
- Application
  - Polynomial, sparse matrix