Linked List

Textbook Ch 10.2

Outline

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

List ADT

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

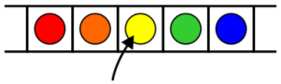
$$(\ A_1 \ A_2 \ \ldots \ A_{n-1} \ A_n \ )$$

– The same value may occur more than once

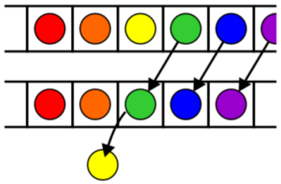
Operations

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

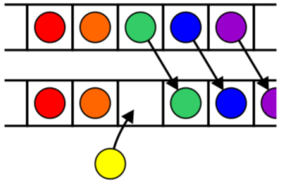
Access to the object



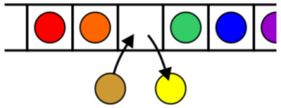
Erasing an object



Insertion of a new object

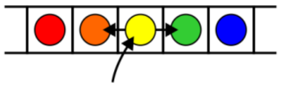


Replacement of the object



Operations

Given access to the k^{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!”

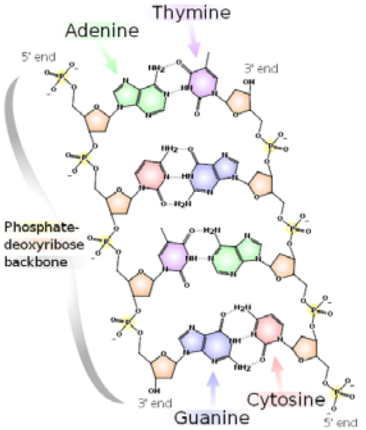
The restriction using an alphabet emphasizes specific operations that would seldom be used otherwise

- 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=STGlsyqtjYMC
- References:

  http://en.wikipedia.org/wiki/DNA

  http://en.wikipedia.org/wiki/Bioinformatics

Arrays

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

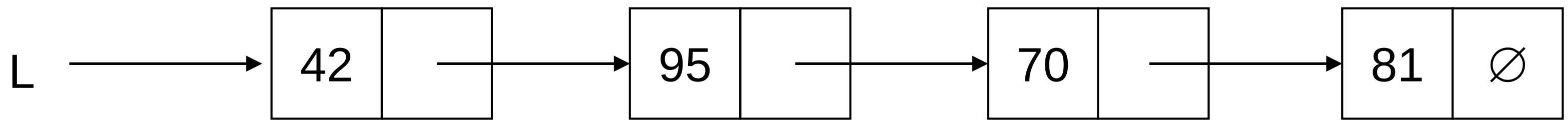
Outline

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

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

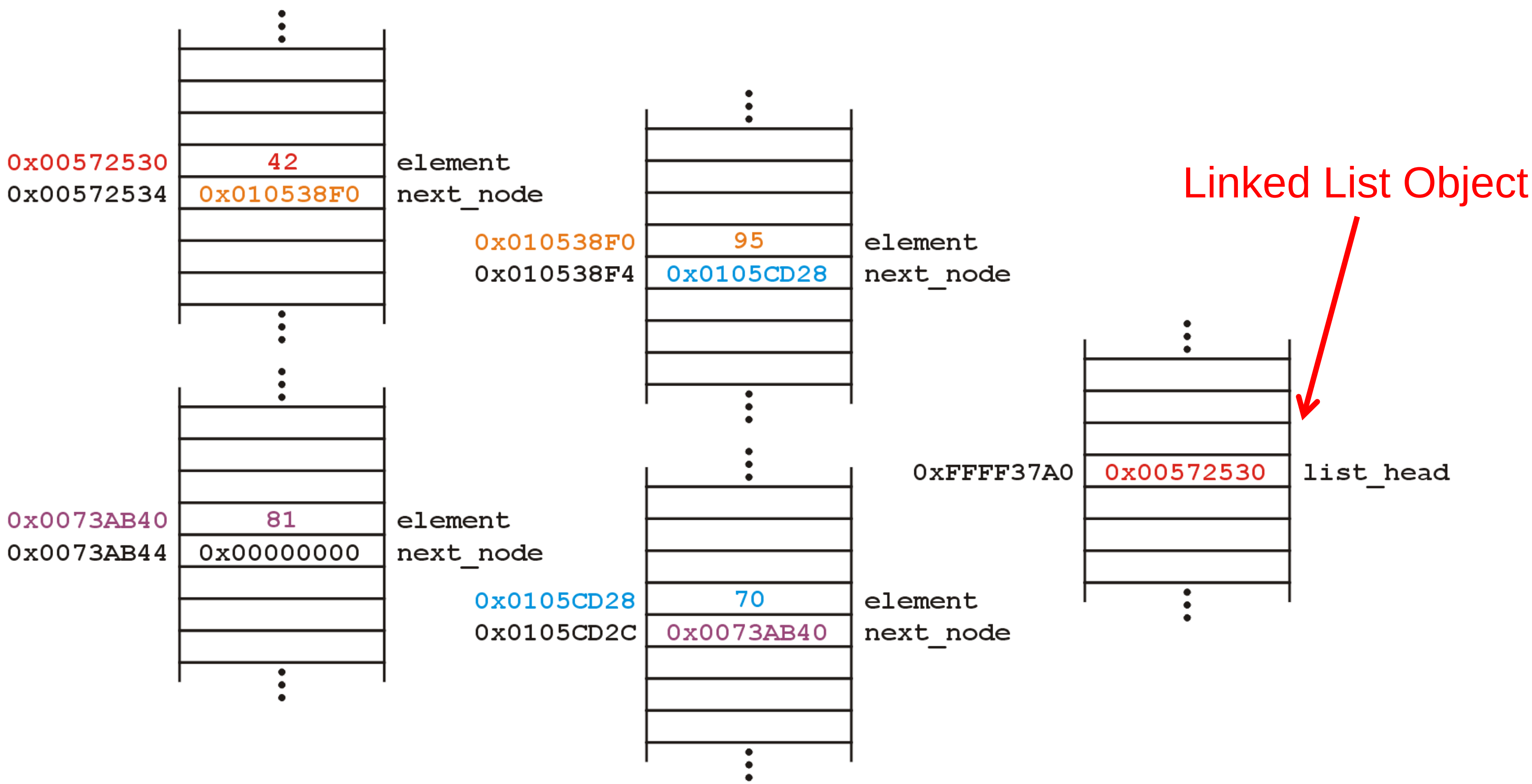
Suppose we want a linked list to store the values

42 95 70 81

in this order

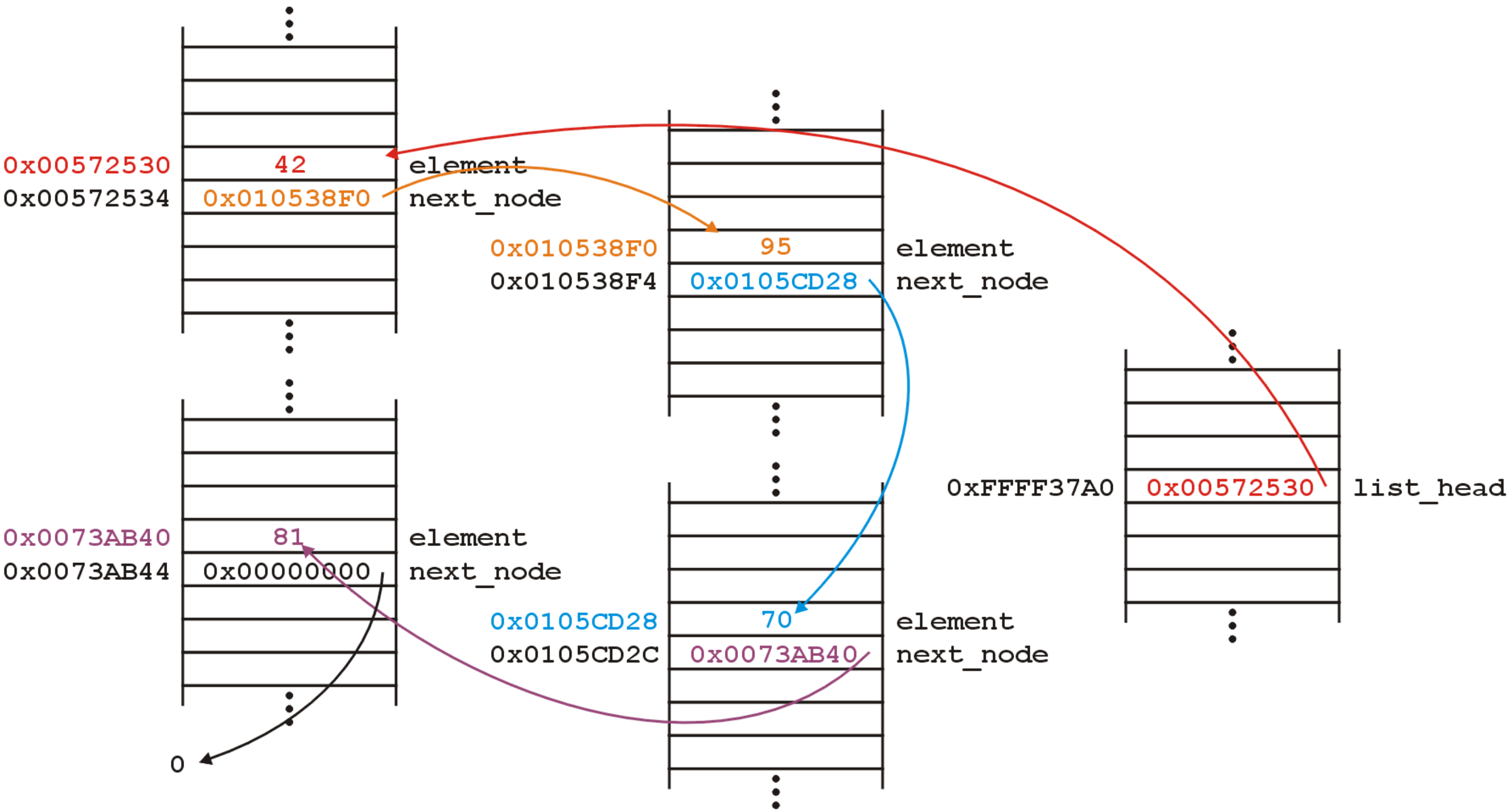
Structure

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



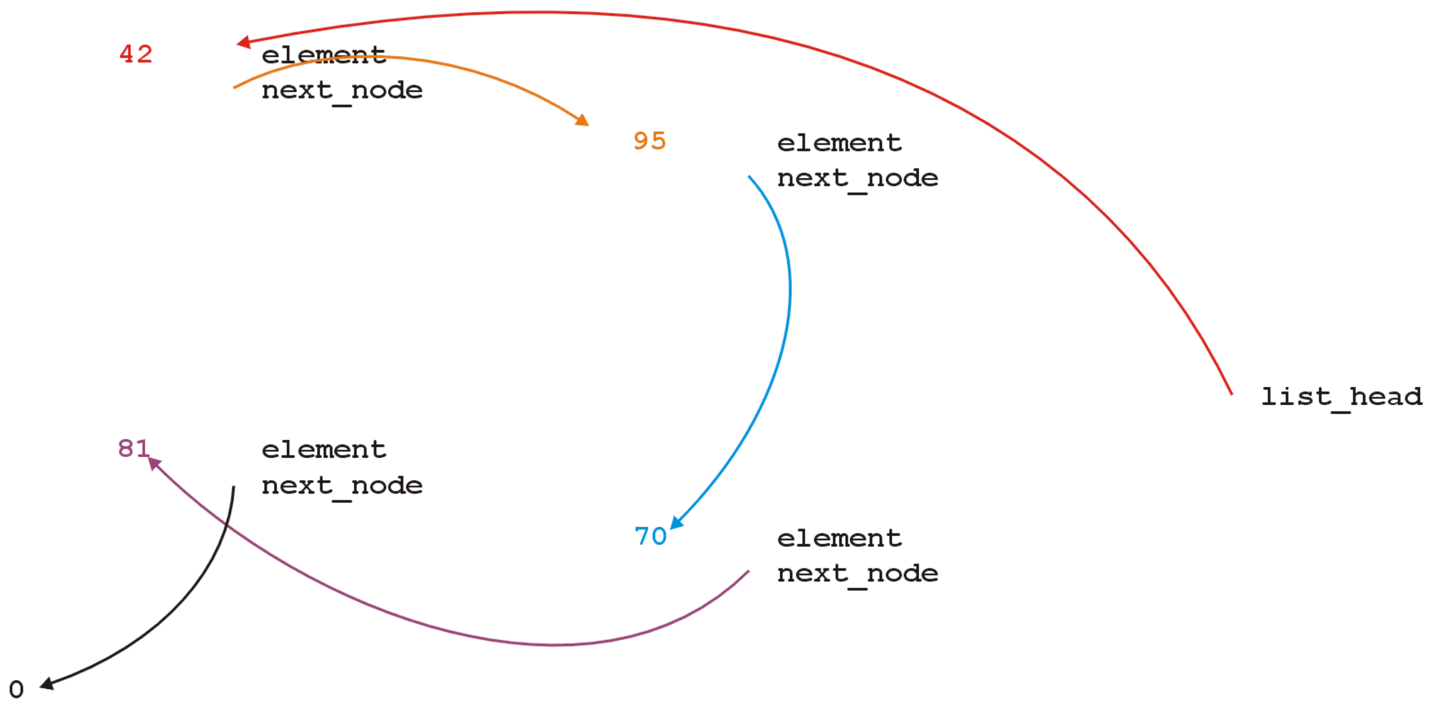
Structure

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 emtpy() 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:

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

`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`

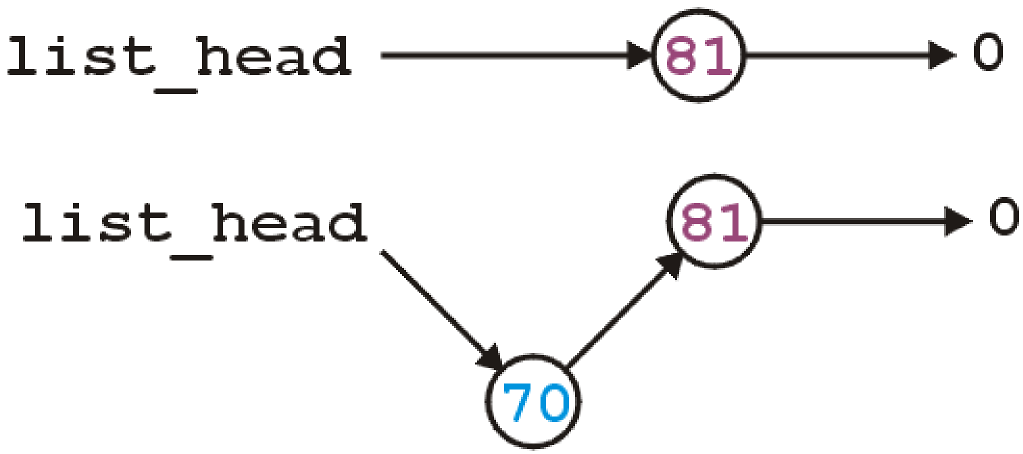
We can do this as follows:

```
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

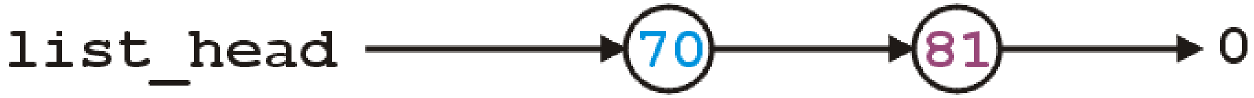
- The original value of `list_head` is accessed first before the function call is made

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**

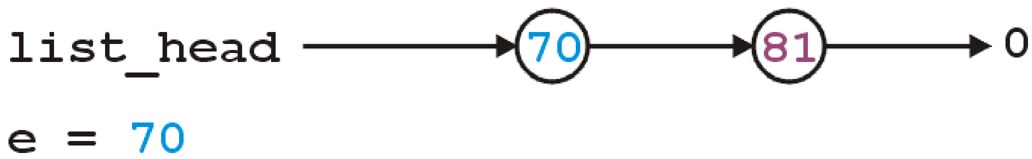
int pop_front()

Does this work?

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

    int e = front();
    delete head();
    list_head = head()->next();
    return e;
}
```

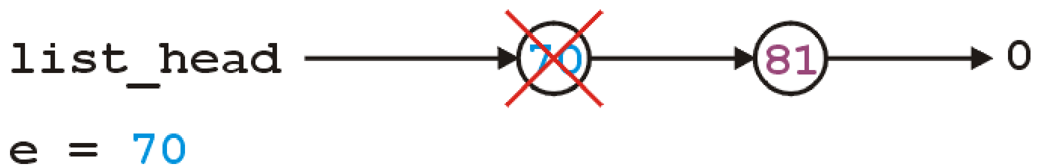
int pop_front()

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

    int e = front();

    delete head();

    list_head = head()->next();

    return e;
}
```



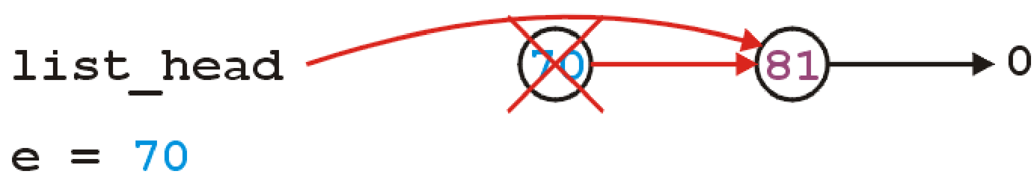
int pop_front()

```
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;
}
```



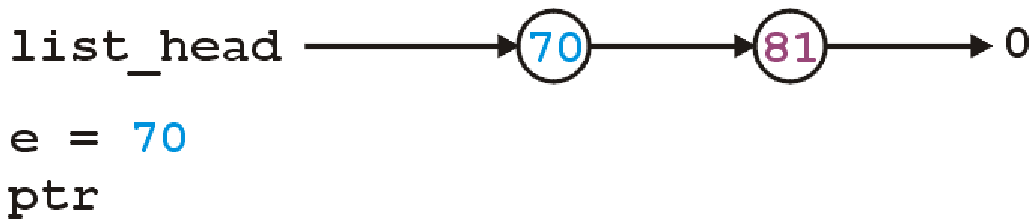
int pop_front()

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;
}
```

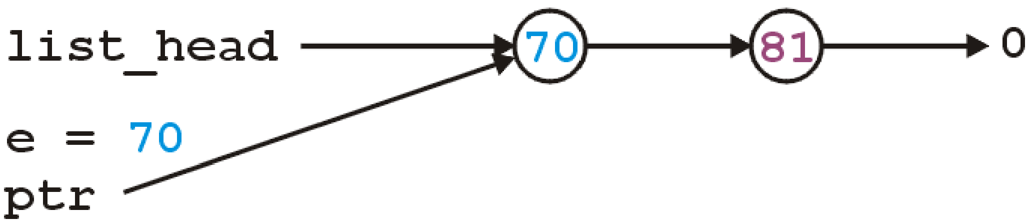
int pop_front()

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

    int e = front();

    Node *ptr = head();

    list_head = head()->next();

    delete ptr;

    return e;
}
```



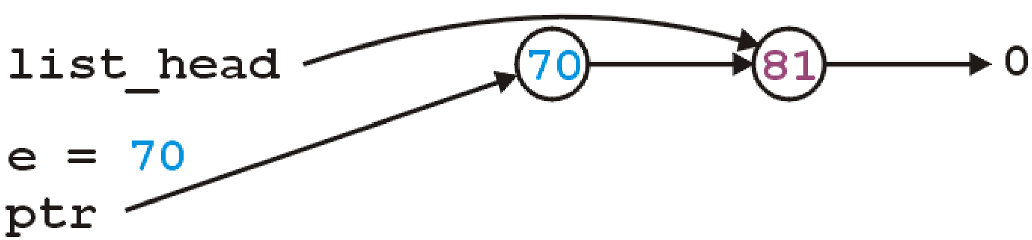
int pop_front()

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

    int e = front();

    Node *ptr = head();

    list_head = head()->next();

    delete ptr;

    return e;
}
```



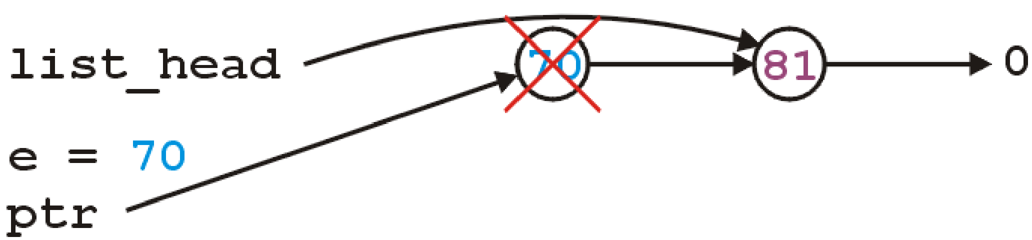
int pop_front()

```
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

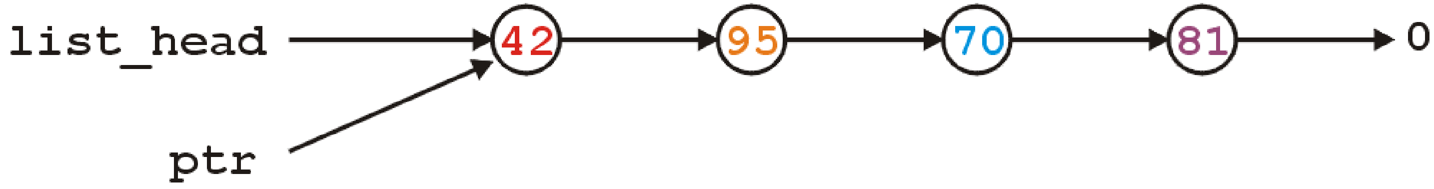
Stepping through a Linked List

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

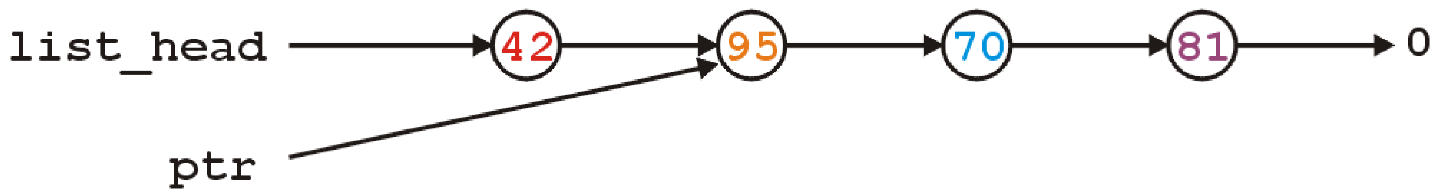
With the initialization and first iteration of the loop, we have:



`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

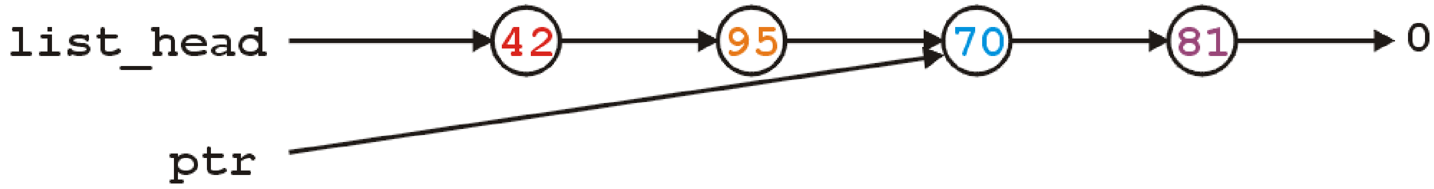
Stepping through a Linked List

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



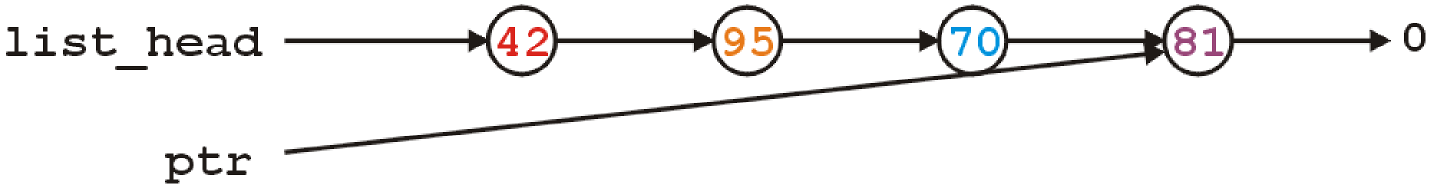
Stepping through a Linked List

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



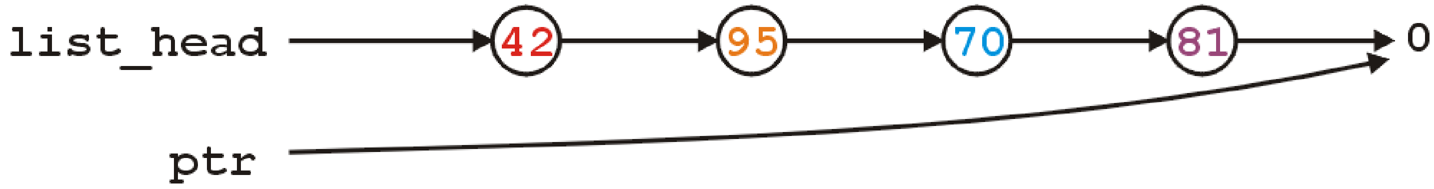
Stepping through a Linked List

`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:

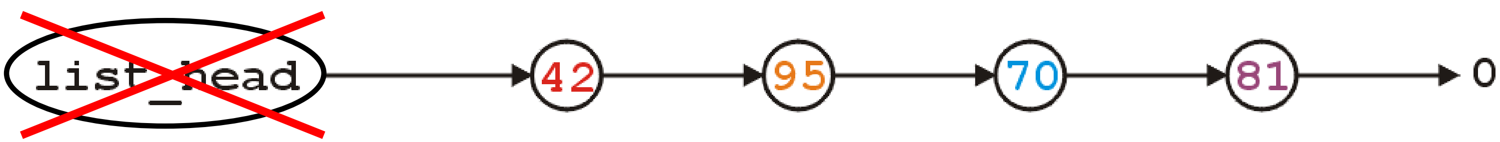
- Friends
- Nested classes
- Inner classes (Java/C#)

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();
}
```