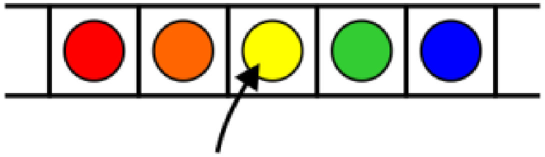
Lists

Abstract List

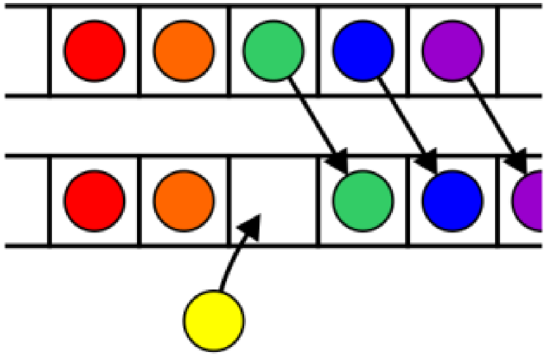
- An **Abstract List (or List ADT)** is linearly ordered data where the programmer explicitly defines the ordering
    - We will look at the most common operations that are usually
    - The most obvious implementation is to use either an array or linked list
    - These are, however, not always the most optimal

Operations

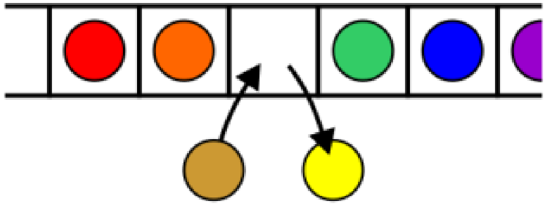
- Operations at the kth entry of the list include:



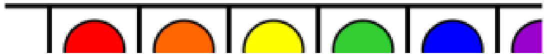
  - Access to the object

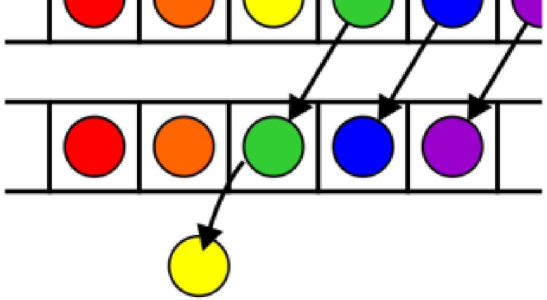


  - Insertion of a new object



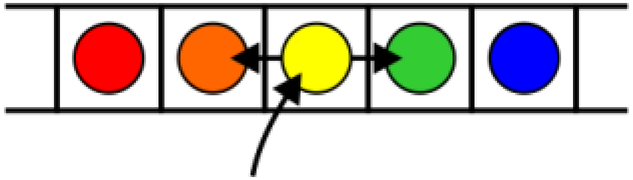
  - Replacement of the object





- Erasing an object

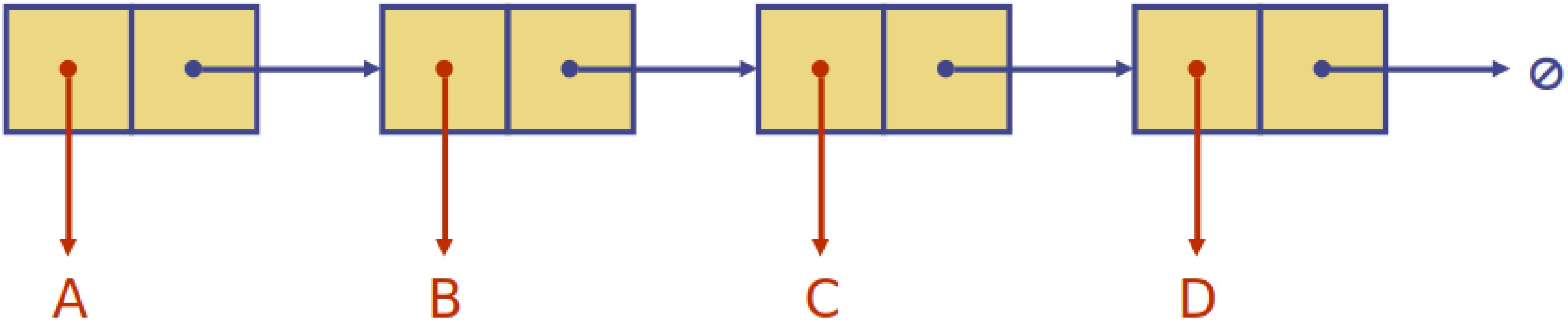
- Given access to the kth object, gain access to either the previous or next object



- Given two abstract lists, we may want to
    - Concatenate the two lists
    - Determine if one is a sub-list of the other

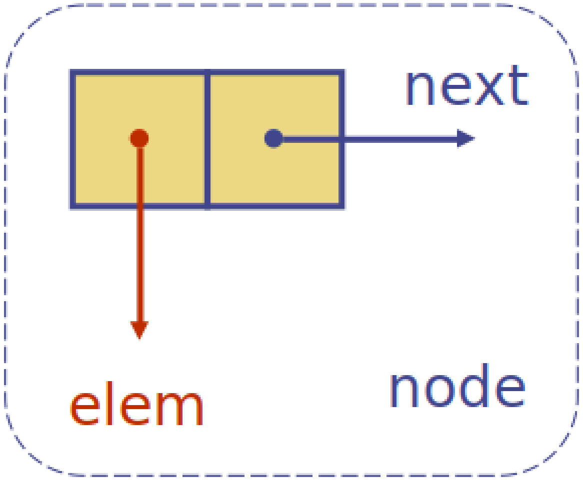
Singly Linked List

- A **linked list** is a data structure consisting of a sequence of object 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 List ADT

- The **Node List ADT** models a sequence of positions storing arbitrary objects
    - It establishes a before/after relation between positions
- The node are dynamically created in a linked list
- A `Node` class must store the **data** and a **reference** to the next node (also a pointer)



In [3]:

```
class Node {
public:
    Node( int = 0, Node* = nullptr );

    int value() const;
    Node* next() const;

private:
    int node_value;
    Node *next_node;
};
```

Accessors

- The two member functions are accessors which simply return the `node_value` and the `next_node` member variables, respectively
    - Member functions that do not change the object acted upon are variously called *accessors*, *readonly functions*, *inspectors*, and, when it involves simply returning a member variable, *getters*

Accessors

- The two member functions are accessors which simply return the `node_value` and the `next_node` member variables, respectively
    - Member functions that do not change the object acted upon are variously called *accessors*, *readonly functions*, *inspectors*, and, when it involves simply returning a member variable, *getters*

In [4]:

```
int Node::value() const {
    return node_value;
}
```

Accessors

- The two member functions are accessors which simply return the `node_value` and the `next_node` member variables, respectively
    - Member functions that do not change the object acted upon are variously called *accessors*, *readonly functions*, *inspectors*, and, when it involves simply returning a member variable, *getters*

In [4]:

```
int Node::value() const {
    return node_value;
}
```

In [5]:

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

Constructor

- The constructor assigns the two member variables based on the arguments

Constructor

- The constructor assigns the two member variables based on the arguments

In [6]:

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

Constructor

- The constructor assigns the two member variables based on the arguments

In [6]:

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

In [7]:

```
{
    Node n1;
    cout << n1.value() << " " << n1.next() << endl;
    Node n2{12};
    cout << n2.value() << " " << n2.next() << endl;
    Node n3{12, &n2};
    cout << n3.value() << " " << n3.next() << endl;
}
```

```
0 0
12 0
12 0x7ffe169e60a8
```

Accessors (cont.)

- In C++, a member function cannot have the same name as a member variable

	Member Variables	**Member Functions**
Vary capitalization	next_node	Next_node() or NextNode()
Prefix with "get"	next_node	get_next_node() / getNextNode()
Use an underscore	next*node*	next_node()
Different names	next_node	next()

- Always use the naming convention and coding styles used by your employer - even if you disagree with them
    - Consistency aids in maintenance

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 {
    public:
        class Node {...};

    private:
        Node *list_head;
    // ...
};
```

Structure

- To begin, let us look at the internal representation of a linked list
- Suppose we want a linked list to store the values in this order

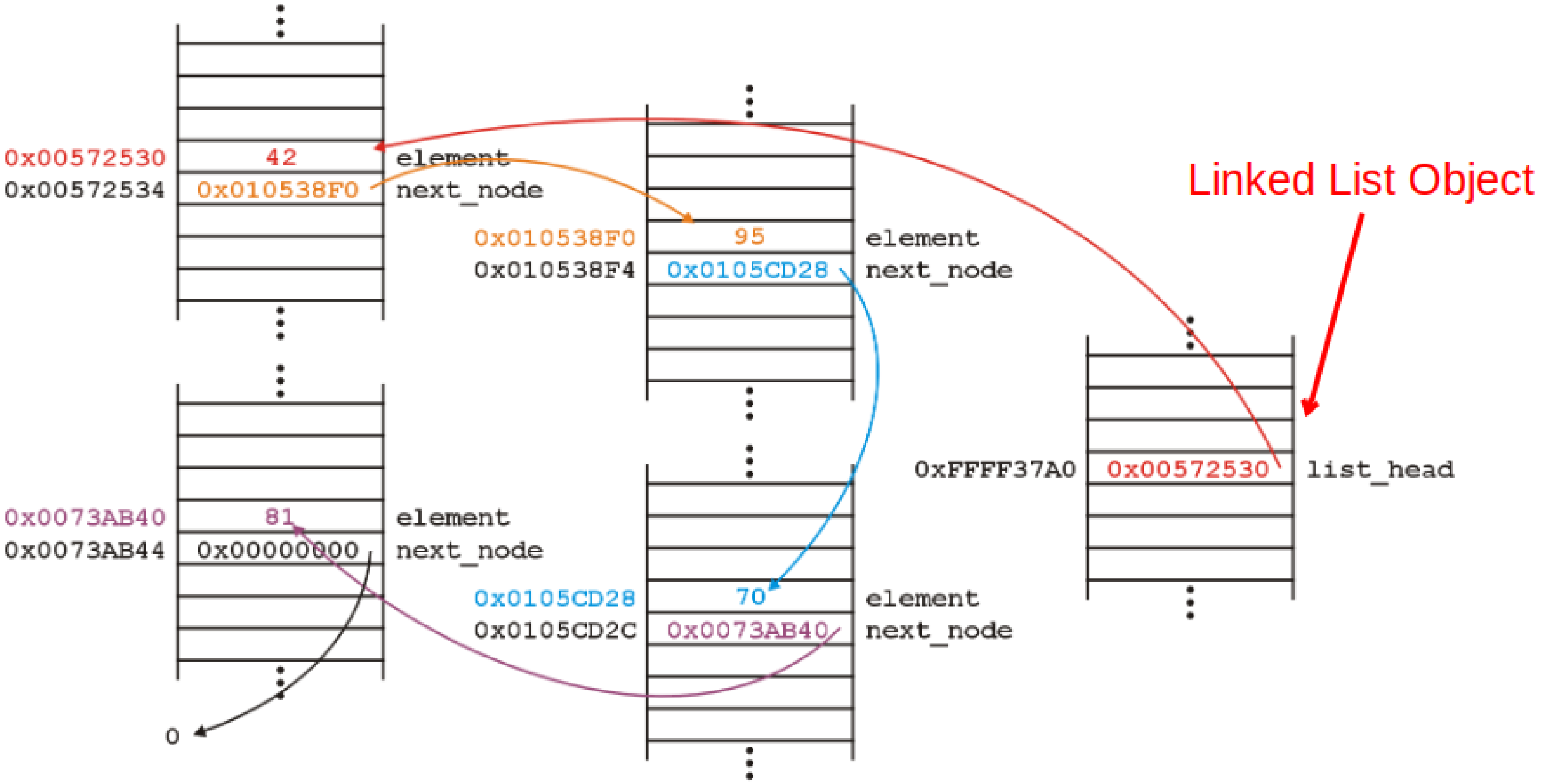
  42 95 70 81

Structure (cont.)

- A linked list uses *linked allocation*, and therefore each node may appear anywhere in memory
- Also the memory required for each node equals the memory required by the member variables
    - 4 bytes for the linked list (a pointer)
    - 8 bytes for each node (an int and a pointer)
        - We are assuming a 32-bit machine

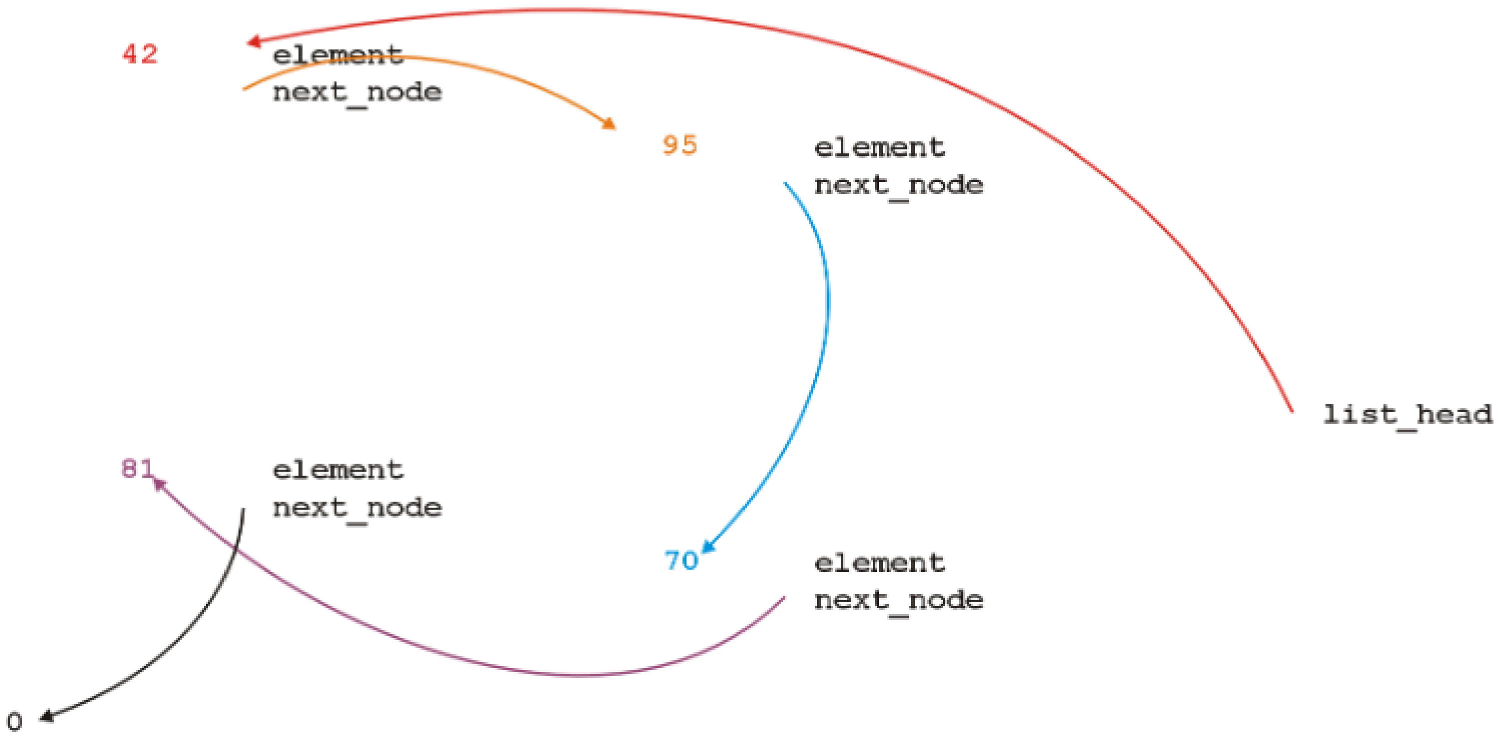
Structure (cont.)

- Such a list could occupy memory as follows:
  - The `next_node` pointers store the addresses of the next node in the list



Structure (cont.)

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



Structure (cont.)

- 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 value
    - 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 manage the stored values in the linked list
    - insert into,
    - access, and
    - erase from

Operations (cont.)

- 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 *begin() const;
    ```

- Member functions that may change the object acted upon are variously called *mutators*, *modifiers*, and, when it involves changing a single member variable, *setters*

Operations (cont.)

- 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 );
```

Capacity

- 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`

Consider this simple (but **incomplete**) linked list class:

Consider this simple (but **incomplete**) linked list class:

In [8]:

```
class List {
public:
    // we defined it outside of the List class scope
    //class Node {...};
    List();
    ~List(){};

    // Accessors
    bool empty() const;
    int size() const;
    int front() const;
    Node* begin() const;
    Node* end() const;

    // Mutators
    void push_front( int );
    int pop_front();

    // Misc
    int count( int ) const;
    int erase( int );

private:
    Node *list_head; // head pointer of the list
};
```

Constructor

- The constructor initializes the linked list
    - We do not count how may objects are in this list, thus:
        - we must rely on the last pointer in the linked list to point to a special value
        - in C++, that standard value is `nullptr`
- Thus, in the constructor, we assign `list_head` the value `nullptr`
- We will always ensure that when a linked list is empty, the list head is assigned `nullptr`

Constructor

- The constructor initializes the linked list
    - We do not count how may objects are in this list, thus:
        - we must rely on the last pointer in the linked list to point to a special value
        - in C++, that standard value is `nullptr`
- Thus, in the constructor, we assign `list_head` the value `nullptr`
- We will always ensure that when a linked list is empty, the list head is assigned `nullptr`

In [9]:

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

Allocation

The constructor is called whenever an object is created, either:

- Statically
    - The following statement defines `ls` to be a linked list and the compiler deals with memory allocation

      ```
      List ls;
      ```
- Dynamically
    - The following statement requests sufficient memory from the OS to store an instance of the class

      ```
      List *pls = new List();
      ```
- In both cases, the memory is allocated and then the constructor is called

Static Allocation

Static Allocation

In [10]:

```
int f() {
    List ls;   // ls is declared as a local variable on the stack

    ls.push_front( 3 );
    cout << ls.front() << endl;

    // The return value is evaluated
    // The compiler then calls the destructor for local variables
    // The memory allocated for 'ls' is deallocated

    return 0;
}
```

Dynamic Allocation

Dynamic Allocation

In [11]:

```
List* f( int n ) {
    List *pls = new List();  // pls is allocated memory by the OS

    pls->push_front( n );
    cout << pls->front() << endl;

    // The address of the linked list is the return value
    // After this, the 4 bytes for the pointer 'pls' is deallocated
    // The memory allocated for the linked list is still there

    return pls;
}
```

`empty()`

- Starting with the easier member functions:

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

- Better yet:

`empty()`

- Starting with the easier member functions:

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

- Better yet:

In [12]:

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

`empty()`

- Starting with the easier member functions:

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

- Better yet:

In [12]:

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

In [13]:

```
{
    List ls;
    cout << ls.empty() << endl;
}
```

```
true
```

`begin()`

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

`begin()`

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

In [14]:

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

`begin()`

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

In [14]:

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

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

`begin()`

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

In [14]:

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

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

In [15]:

```
{
    List ls;
    cout << ls.empty() << endl;
    cout << ls.begin() << endl;
}
```

```
true
0
```

`end()`

- The member function `Node* end() const` equals whatever the last node in the linked list points to

`end()`

- The member function `Node* end() const` equals whatever the last node in the linked list points to

In [16]:

```
// In this case, nullptr front
Node* List::end() const {
    return nullptr;
}
```

`front()`

- To get the first value 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 begin()->value();
}
```

- The member function `int front() const` requires some additional consideration, however:
    - 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
    - The operating system would terminate the running program
    - Instead, we can use an exception handling mechanism where we thrown an exception

- The member function `int front() const` requires some additional consideration, however:
    - 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
    - The operating system would terminate the running program
    - Instead, we can use an exception handling mechanism where we thrown an exception

In [17]:

```
int List::front() const {
    if ( empty() ) {
        throw underflow_error("List is empty");
    }
    return begin()->value();
}
```

Software Engening Tip

- Why is `empty()` better than

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

    return list_head->node_value;
}
```

- Two benefits:
    - More readable
    - If the implementation changes we do nothing

Inserting at the Head

- Step required for insering a new element at the beginning of the list
    - Allocate a new node
    - Insert new element value
    - Have new node point to old head
    - Update head to point to new node
- Corresponding mutator function is `void push_front(int)`

`push_front`

Let us add a value in front of the list

- If it is empty, we start with:

`push_front`

Let us add a value in front of the list

- If it is empty, we start with:
- and, if we try to add 81, we should end up with:

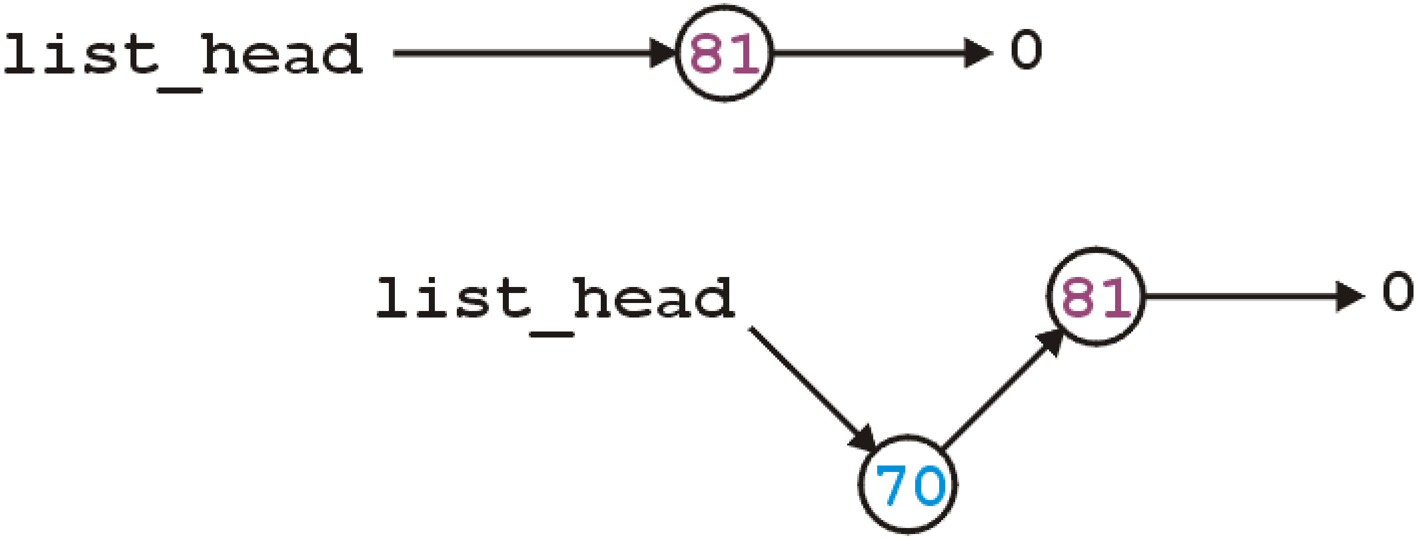
- To visualize what we must do:
  - We must create a new node which:
    - stores the value 81, and
    - is pointing to 0
  - We must then assign its address to list_head
- We can do this as follows:

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

- We could also use the default value...

- Suppose however, we already have a non-empty list

- Suppose however, we already have a non-empty list



- Adding 70, we want:

- 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 );
```

In [18]:

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

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

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

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

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

- Are we allowed to do this?

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

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

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

- Are we allowed to do this?

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

- **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

Question

- Does this work?

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

Question

- Does this work?

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

- Why or why not? What happens to `new_node`?

Question

- Does this work?

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

- Why or why not? What happens to `new_node`?
- How does this differ from

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

Insertion

- We can generalize `push_front`, in order to insert a node at any position of the list:

```
void List::insert( Node* p, int n ) {
    p->next_node = new Node( n, p->next() );
}
```

Insertion

- We can generalize `push_front`, in order to insert a node at any position of the list:

```
void List::insert( Node* p, int n ) {
    p->next_node = new Node( n, p->next() );
}
```

- General `insert` method can be used to rewrite `push` methods

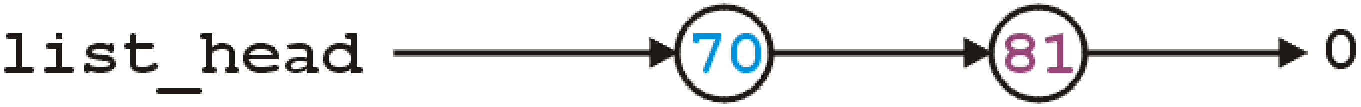
```
void List::push_front( int n ) {
    insert( begin(), n );
}

void List::push_back( int n ) {
    insert( end(), n ) // if we have tail pointer
}
```

Removing at the Head

- Erasing the element from the front of the list requires:
    1. Update head to point to next node in the list
    2. Free memory of the former first node

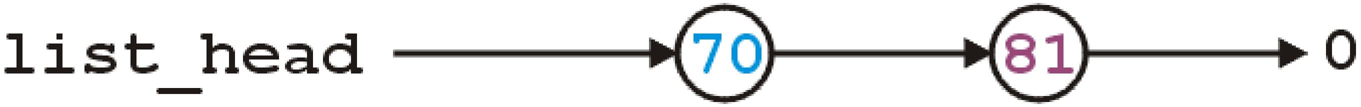
`pop_front`

- 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:

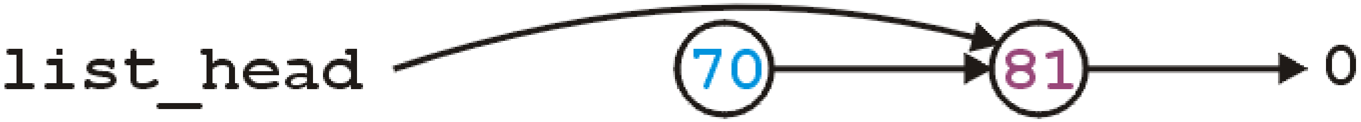


`pop_front`

- 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



- Easy enough:

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

- Easy enough:

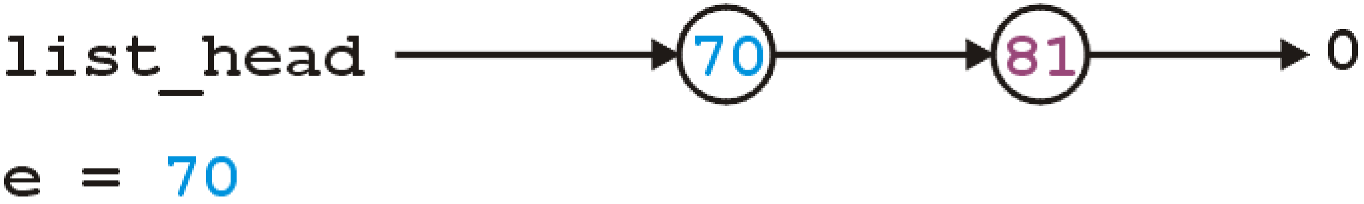
```
int List::pop_front() {
  int e = front();
  list_head = begin()->next();
  return e;
}
```

- Unfortunately, we have some problems:
    - The list may be empty
    - We still have the memory allocated for the node containing 70

- Does this work?

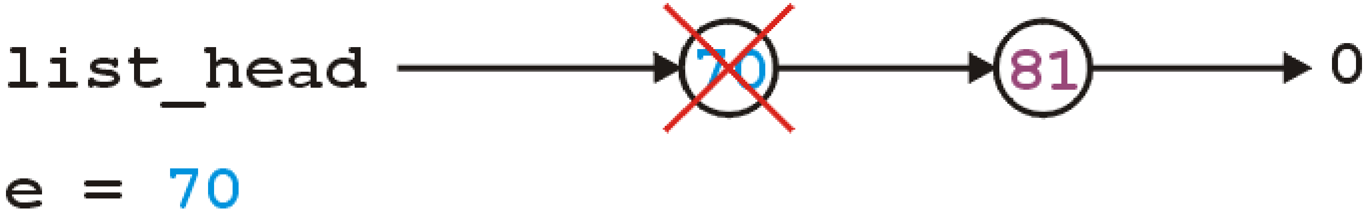
```
int List::pop_front() {
    if ( empty() ) {
        throw underflow_error("List is empty");
    }

    int e = front();
    delete begin();                 /// ????
    list_head = begin()->next();  /// ????
    return e;
}
```

```
int List::pop_front() {
    if ( empty() ) { throw underflow_error(); }
    int e = front();
```



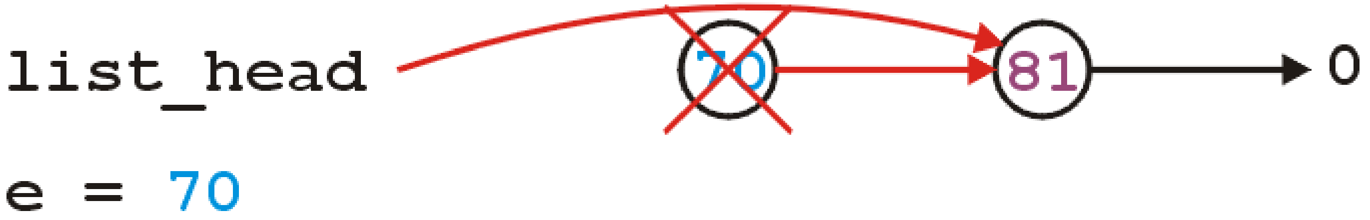
```
    delete begin();
    list_head = begin()->next();
    return e; </div> }
```

```
int List::pop_front() {
    if ( empty() ) { throw underflow_error(); }
    int e = front();
    delete begin();
```



```
    list_head = begin()->next();
    return e; </div> }
```

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



```
    return e;
    </div> }
```

Problem

- The problem is, we are accessing a node which we have just deleted
- Unfortunately, this will work more than 99% of the time:
    - The running program (process) may still own the memory
- Once in a while it will fail ...
    - ... and it will be almost impossible to debug

```
prev->next = toDelete->next;
delete toDelete;

// if only forgetting were
// this easy for me.
```

```
assert "It's going to be okay.";
```

Solution

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

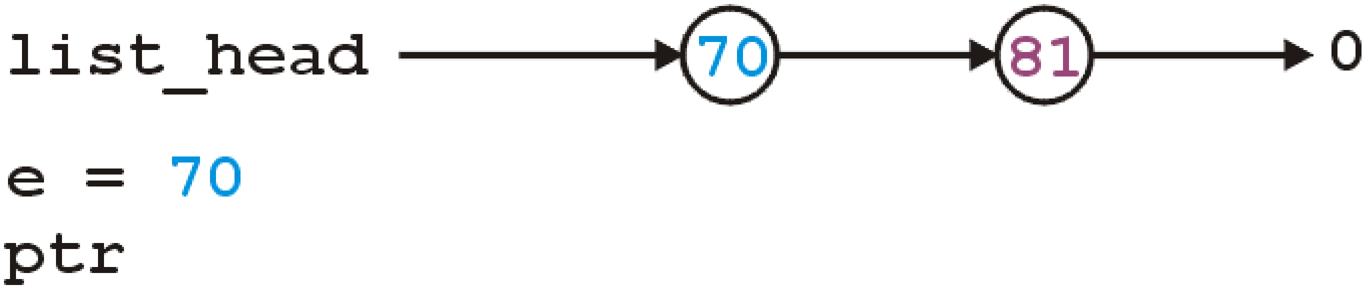
Solution

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

In [19]:

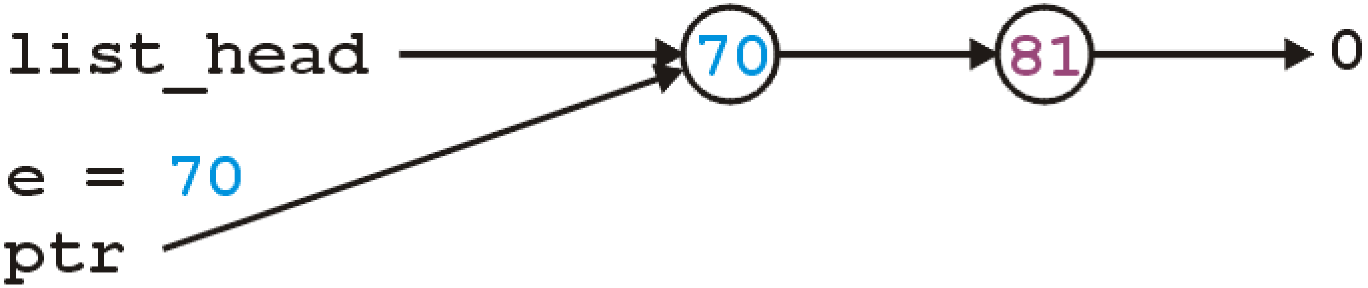
```
int List::pop_front() {
     if ( empty() ) {
          throw underflow_error("List is empty");
     }

    int e = front();
    Node *ptr = list_head;
    list_head = list_head->next();
    delete ptr;
    return e;
}
```

```
int List::pop_front() {
    if ( empty() ) { throw underflow_error(); }
    int e = front();
```



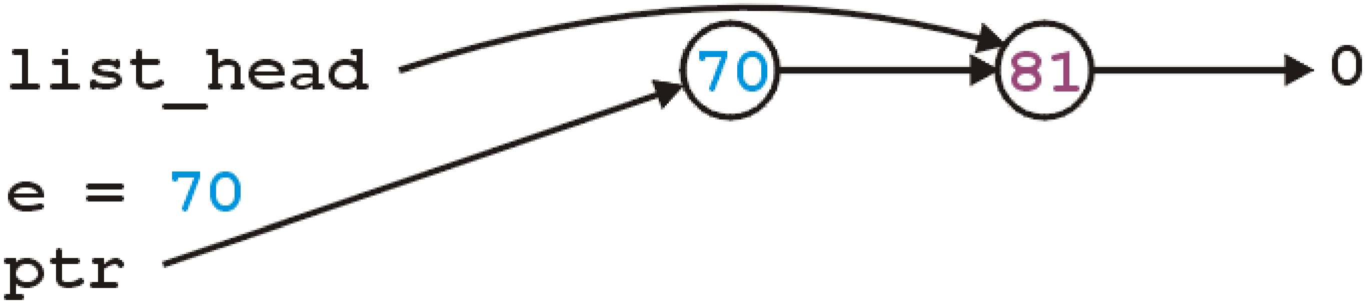
```
    Node *ptr = begin();

    list_head = begin()->next();

    delete ptr;

    return e;

    </div> }
```

```
int List::pop_front() {

    if ( empty() ) { throw underflow_error(); } int e = front();
    Node *ptr = begin();
```



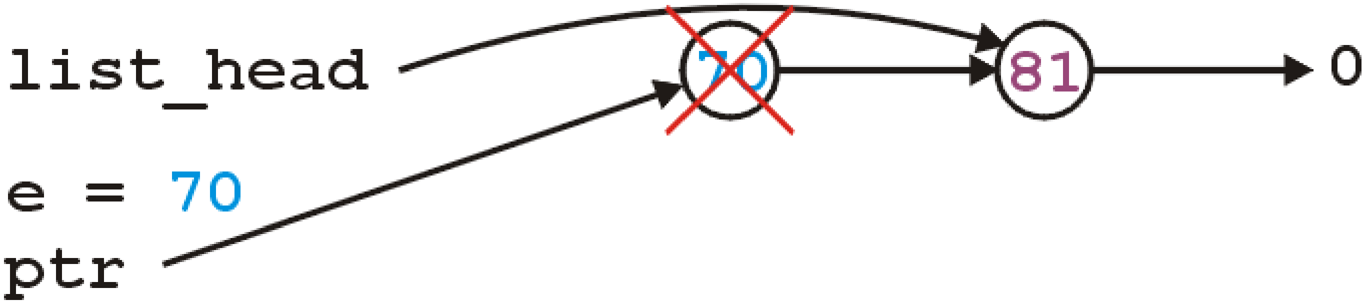
```
    list_head = begin()->next();

    delete ptr;

    return e;

    </div> }
```

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



```
    delete ptr;
    return e;
    </div> }
```

```
int List::pop_front() {

    if ( empty() ) { throw underflow_error(); } int e = front(); Node *ptr = begin(); list_head =
    begin()->next();
```



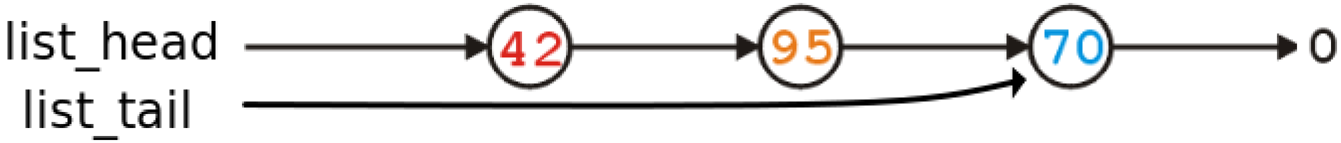
```
    delete ptr;
    return e;

    </div> }
```

Inserting at the Tail

- Inserting or removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node
    - Unless the list ADT maintains the `tail` pointer

`push_back`

- For a given the linked list with `tail` pointer



- Allocate a new node & insert new element into it
- Have new node point to null
- Have old last node point to new node
- Update `tail` pointer to point to new node

`push_back`

- For a given the linked list with `tail` pointer

- Allocate a new node & insert new element into it
- Have new node point to null
- Have old last node point to new node
- Update `tail` pointer to point to new node

`push_back`

- For a given the linked list with `tail` pointer
- Allocate a new node & insert new element into it
- Have new node point to null

- Have old last node point to new node
- Update `tail` pointer to point to new node

`push_back`

- For a given the linked list with `tail` pointer
- Allocate a new node & insert new element into it
- Have new node point to null
- Have old last node point to new node

- Update `tail` pointer to point to new node

`push_back`

- For a given the linked list with `tail` pointer
- Allocate a new node & insert new element into it
- Have new node point to null
- Have old last node point to new node
- Update `tail` pointer to point to new node

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

- 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 `end()` (e.g. `nullptr`)
- With each step, we set the pointer to point to the next object

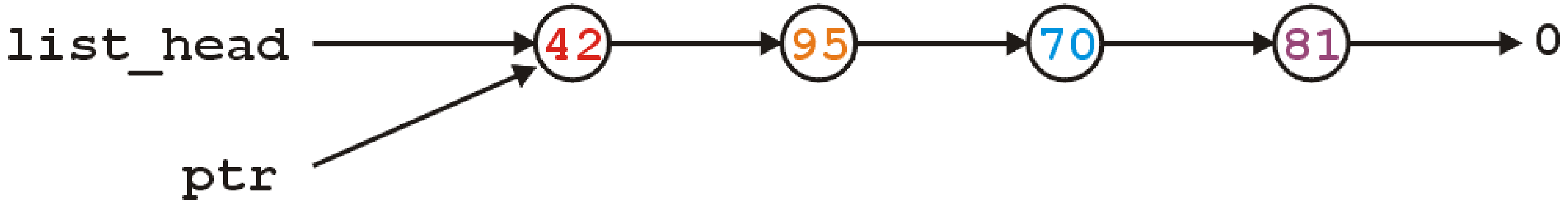
- The process of stepping through a linked list can be thought of as being analogous to a for-loop:
- We initialize a temporary pointer with the list `head`
- We continue iterating until the pointer equals `end()` (e.g. `nullptr`)
- With each step, we set the pointer to point to the next object

Thus, we have:

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

Initialization

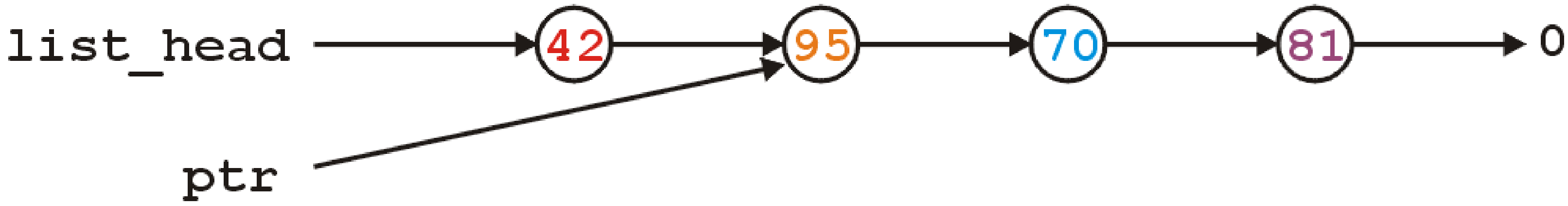
- With the initialization and first iteration of the loop, we have:



- `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

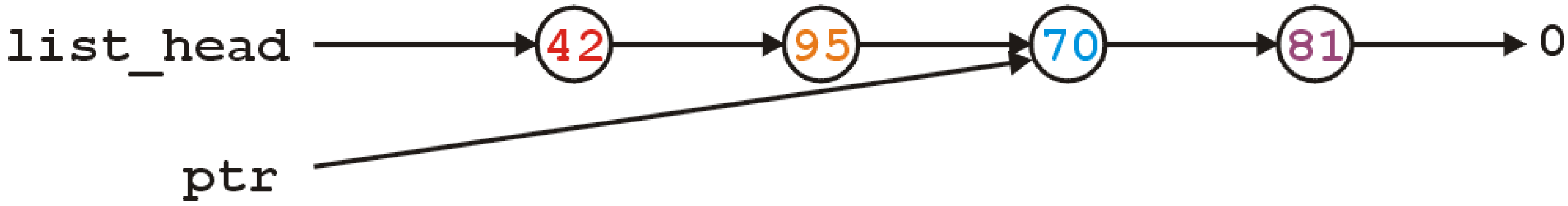
- `ptr != nullptr` and thus we evaluate the loop and increment the pointer



- In the loop, we can access the value being pointed to by using `ptr->value()`

Stepping

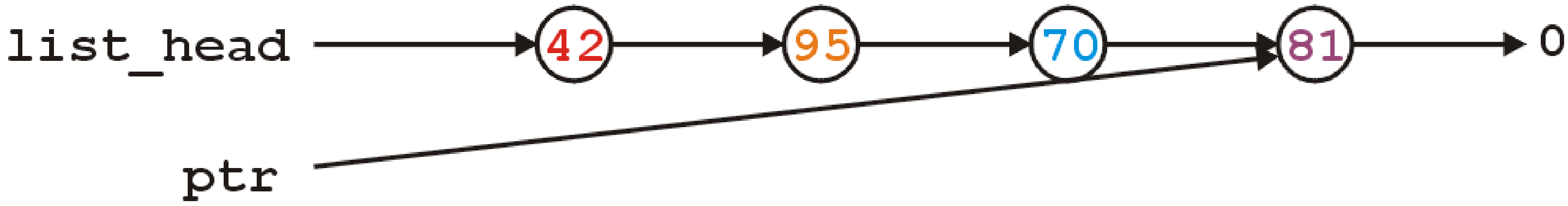
- `ptr != nullptr` and thus we evaluate the loop and increment the



- Also, in the loop, we can access the next node in the list by using `ptr->next()`

Stepping

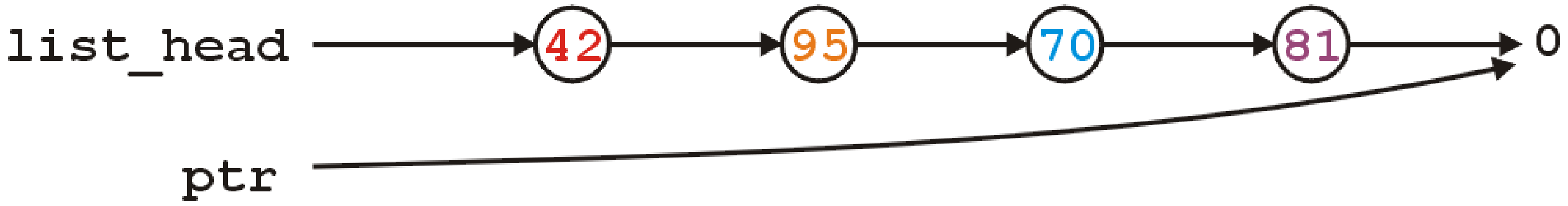
- `ptr != nullptr` and thus we evaluate the loop and increment the



- This last increment causes `ptr == nullptr`

Reached the End

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



- Because the variable `ptr` was declared inside the loop, we can no longer access it

`count`

- To implement `int count(int) const`, we simply check if the argument matches the value 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

`count`

- To implement `int count(int) const`, we simply check if the argument matches the value 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

In [20]:

```
int List::count( int n ) const {
    int node_count = 0;

    for ( Node *ptr = begin(); ptr != end(); ptr = ptr->next() ) {
        if ( ptr->value() == n ) {
            ++node_count;
        }
    }

    return node_count;
}
```

In [21]:

```cpp
#include "../src/BasicLinkedList.h"
{
    BasicLinkedList<int> ls;
    ls.push_front(7);
    ls.push_front(6);
    ls.push_front(5);
    ls.push_front(7);
    std::cout << ls << std::endl;
    std::cout << "List size = "<< ls.size() << std::endl;
    std::cout << "# of 7s in the list = "<< ls.count(7) << std::endl;
}
```

```
]->(7)[0x5563577b6410]->(5)[0x556356f097e0]->(6)[0x556356cc4180]->(7)[0x55
635777bcb0]->0
List size = 4
# of 7s in the list = 2
```

`erase`

- To remove an arbitrary value, 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

In [22]:

```
#include "../src/BasicLinkedList.h"
{
    BasicLinkedList<int> ls;
    ls.push_front(6);
    ls.push_front(7);
    ls.push_front(3);
    ls.push_front(7);
    std::cout << ls << std::endl;
    ls.erase(3);
    std::cout << ls << std::endl;
}
```

```
]->(7)[0x55635722b9c0]->(3)[0x556357925870]->(7)[0x556356575130]->(6)[0x55
635778a250]->0
]->(7)[0x55635722b9c0]->(7)[0x556356575130]->(6)[0x55635778a250]->0
```

Software Engening Tip

- 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

Friends

- In C++, you could explicitly break encapsulation by declaring the class `List` to be a **friend** of the class `Node`:

```
class A {
    private:
        int class_size;
    // ... declaration ...
    friend class B;
};
```

- Now, any member function of class `B` has access to all private member variables of class `A`

- For example, if the `Node` class was one class, and the `List` class was a **friend** of the `Node` class, `List::erase` could modify nodes:

```
int List::erase( int n ) {
    int node_count = 0;
    // ...
    for ( Node *ptr = begin(); ptr != end(); ptr = ptr->next() ) {
        // ...
        if ( some condition ) {
            // access private `next_node` of the Node class
            ptr->next_node = ptr->next()->next();
            // ...
            ++node_count;
        }
    }
    return node_count;
}
```

Nested Classes

- In C++, you can nest one class inside another, which is what we do:

```
class Outer {
    private:
        class Nested {
            private:
                int node_value;
            public:
                int get() const;
                void set( int );
        };
        Nested stored;
    public:
        int get() const;
        void set( int );
};
```

The function definitions are as one would expect:

```
int Outer::Nested::get() const {
    return node_value;
}

void Outer::Nested::set( int n ) {
    node_value = n;
}

int Outer::get() const {
    return stored.get();
}

void Outer::set( int n ) {
    // Not allowed, as node_value is private
    // stored.node_value = n;
    stored.set( n );
}
```

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

- 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();
}
```

- 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();
}
```

- Is this efficient?
    - It runs in $O(n)$ time, where n is the number of objects in the linked list
    - Given that *delete* is approximately 100× slower than most other instructions (it does call the OS), the extra overhead is negligible...

Making Copies

- Is the above sufficient for a linked list class?
- Initially, it may appear yes, but we now have to look at how C++ copies objects during:
  - Passing by value (making a copy), and
  - Assignment

Modifying Arguments

- **Pass by reference** could be used to modify a list

Modifying Arguments

- **Pass by reference** could be used to modify a list

In [23]:

```
void reverse( BasicLinkedList<int> &list ) {
    BasicLinkedList<int> tmp;
    // pop from the front and push into other list
    while ( !list.empty() ) {
        tmp.push_front( list.pop_front() );
    }
    // All the member variables of 'list' and 'tmp' are swapped
    std::swap( list, tmp );
    // The memory for 'tmp' will be cleaned up
}
```

Modifying Arguments

- **Pass by reference** could be used to modify a list

In [23]:

```
void reverse( BasicLinkedList<int> &list ) {
    BasicLinkedList<int> tmp;
    // pop from the front and push into other list
    while ( !list.empty() ) {
        tmp.push_front( list.pop_front() );
    }
    // All the member variables of 'list' and 'tmp' are swapped
    std::swap( list, tmp );
    // The memory for 'tmp' will be cleaned up
}
```

In [24]:

```
{
    BasicLinkedList<int> ls;
    ls.push_front(5); ls.push_front(2); ls.push_front(3);
    std::cout << ls << std::endl;
    reverse(ls);
    std::cout << ls << std::endl;
}
```

```
]->(3)[0x556356d67ee0]->(2)[0x5563579860a0]->(5)[0x556357783d40]->0
]->(5)[0x556357783d40]->(2)[0x5563579860a0]->(3)[0x556356d67ee0]->0
```

- If you wanted to prevent the argument from being modified, you could declare it `const`

- If you wanted to prevent the argument from being modified, you could declare it `const`

In [32]:

```
double average( const BasicLinkedList<double> &ls ) {
    double sum = 0, count = 0;
    for ( SinglyLinkedNode<double> *ptr = ls.begin(); ptr != ls.end(); ptr = ptr->next() ) {
        sum += ptr->value();
        ++count;
    }
    return sum/count;
}
```

- If you wanted to prevent the argument from being modified, you could declare it `const`

In [32]:

```
double average( const BasicLinkedList<double> &ls ) {
    double sum = 0, count = 0;
    for ( SinglyLinkedNode<double> *ptr = ls.begin(); ptr != ls.end(); ptr = ptr->next() )
        sum += ptr->value();
        ++count;
    }
    return sum/count;
}
```

In [25]:

```
{
    BasicLinkedList<double> ls;
    ls.push_front(10.0); ls.push_front(25.0); ls.push_front(35.0);
    std::cout << "Average: " << average(ls) << std::endl;
}
```

```
Average: 23.3333
```

- What if you want to pass a copy of a linked list to a function - where the function can modify the passed argument, but the original is unchanged?
    - By default, all the member variables are simply copied over into the new instance of the class
    - This is the default **copy constructor** behavior
    - Because a copy is made, the destructor must also be called on it

Copy Constructor

- You can modify how copies are made by defining a copy constructor
    - The default copy constructor simply copies the member variables
    - In this case, this is not what we want
- The signature for the copy constructor is

```
Class_name( const Class_name & );
```

- For the linked list, we would define the member function

```
List( const List & );
```

- If such a function is defined, every time an instance is passed by value, the copy constructor is called to make that copy
- Additionally, you can use the copy constructor as follows:

- If such a function is defined, every time an instance is passed by value, the copy constructor is called to make that copy

- Additionally, you can use the copy constructor as follows:

In [26]:

```
#include "../src/BasicLinkedList.h"
{
    BasicLinkedList<int> ls1;
    ls1.push_front( 4 );
    ls1.push_front( 2 );
    std::cout << ls1 << std::endl;

    BasicLinkedList<int> ls2( ls1 );  // make a copy of ls1
    std::cout << ls2 << std::endl;
}
```

```
]->(2)[0x565505585030]->(4)[0x5655059c7ff0]->0
]->(2)[0x565503eda680]->(4)[0x565505f2dbe0]->0
```

- If such a function is defined, every time an instance is passed by value, the copy constructor is called to make that copy

- Additionally, you can use the copy constructor as follows:

In [26]:

```
#include "../src/BasicLinkedList.h"
{
    BasicLinkedList<int> ls1;
    ls1.push_front( 4 );
    ls1.push_front( 2 );
    std::cout << ls1 << std::endl;

    BasicLinkedList<int> ls2( ls1 );  // make a copy of ls1
    std::cout << ls2 << std::endl;
}
```

```
]->(2)[0x565505585030]->(4)[0x5655059c7ff0]->0
]->(2)[0x565503eda680]->(4)[0x565505f2dbe0]->0
```

- When an object is **passed/returned by value**, again, the copy constructor is called to make a copy of the passed/returned value

- Thus, we must define a copy constructor:
    - The copy constructor allows us to initialize the member variables
    - Naïvely, we step through `list` and call `push_front( int )`:

```
List::List( List const &list ):list_head( nullptr ) {
  for ( Node *ptr = list.begin();
        ptr != list.end(); ptr = ptr->next() ) {
      push_front( ptr->value() );
  }
}
```

- Thus, we must define a copy constructor:
  - The copy constructor allows us to initialize the member variables
  - Naïvely, we step through `list` and call `push_front( int )`:

```
List::List( List const &list ):list_head( nullptr ) {
    for ( Node *ptr = list.begin();
            ptr != list.end(); ptr = ptr->next() ) {
        push_front( ptr->value() );
    }
}
```

- Does this work?
  - How could we make this work?
  - We need a `push_back( int )` member function

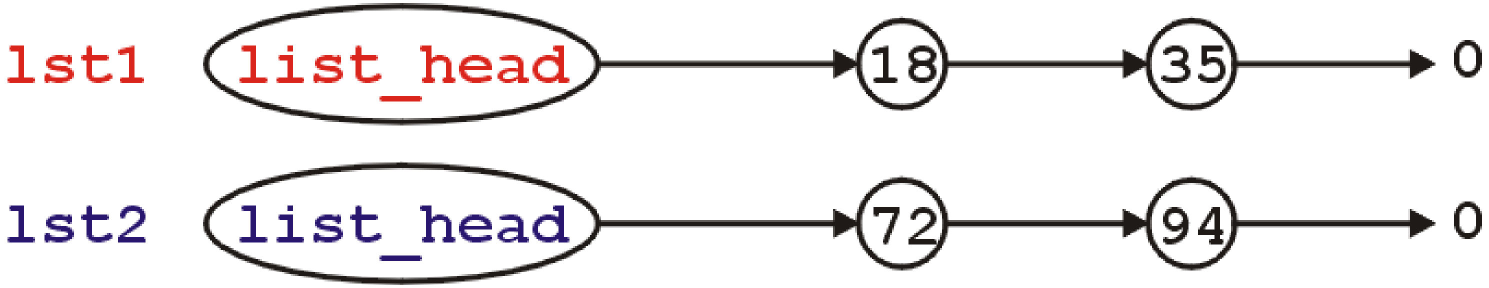
- Unfortunately, to make `push_back( int )` more efficient, we need a pointer to the last node in the linked list
- We require a `list_tail` member variable
- Otherwise, `push_back( int )` becomes a $\Theta(n)$ function
    - This would make the copy constructor $\Theta(n^2)$
- In Assignment 3, you will define and use the member variable `list_tail`

In [ ]:

```
List::List( List const &list ):list_head( nullptr ) {
    // if list is empty, we are finished
    if ( list.empty() ) {
        return;
    }
    // copy the first node
    push_front(list.front());
    // modify the next pointer of the node pointed to by copy
    for (
        Node *original = list.begin()->next(), *copy = begin();
        original != list.end();
        original = original->next(), copy = copy->next()
    ) {
        copy->next_node = new Node( original->value(), nullptr );
    }
}
```

Assignment

- Let's have two lists



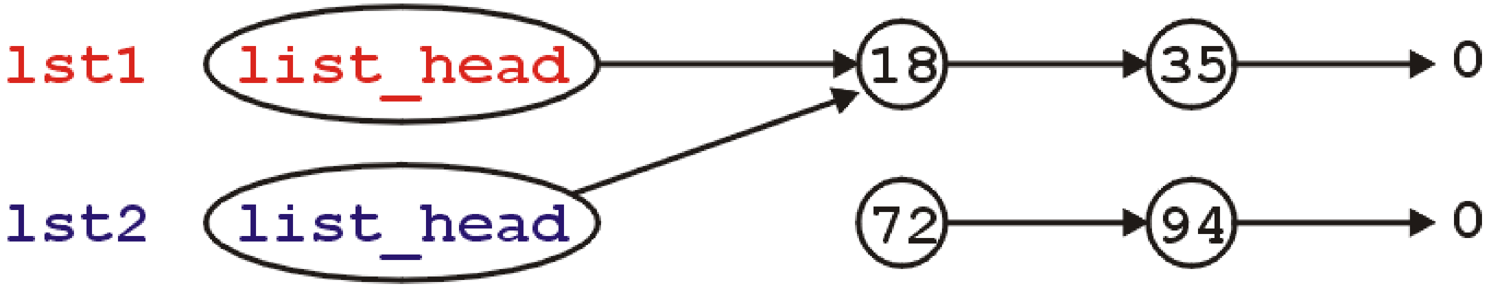
- Consider an assignment:

```
lst2 = lst1;
```

- What do we want? What should this do?
    - The default is to copy the member variables from `lst1` to `lst2`

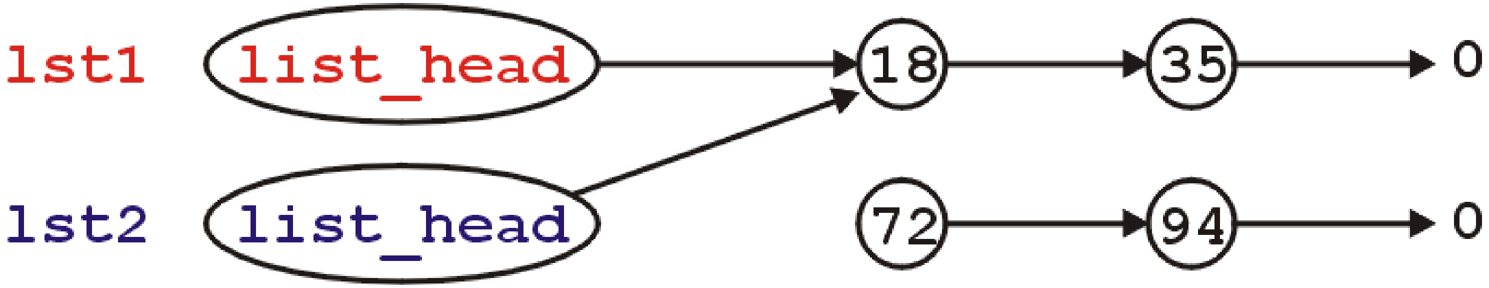
- Because the only member variable of this class is `list_head`, the value it is storing (the address of the first node) is copied over
- It is equivalent to writing:

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



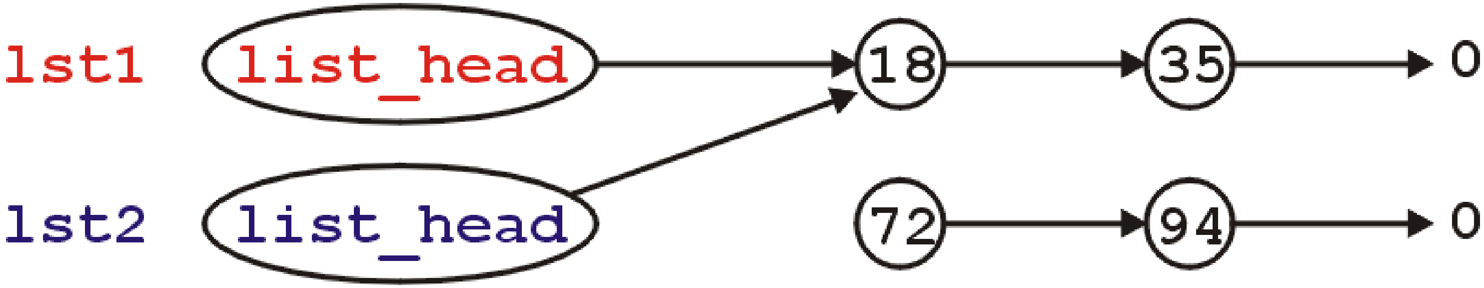
Problem

- What's wrong with this picture?



Problem

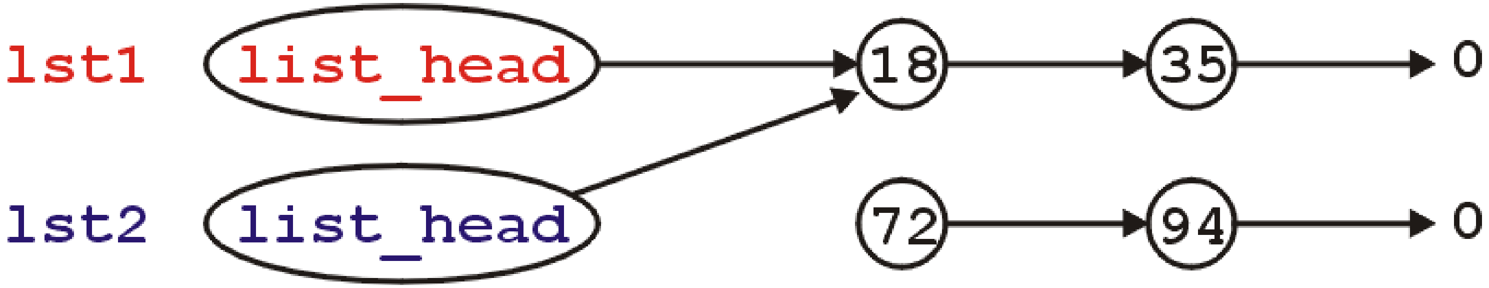
- What's wrong with this picture?



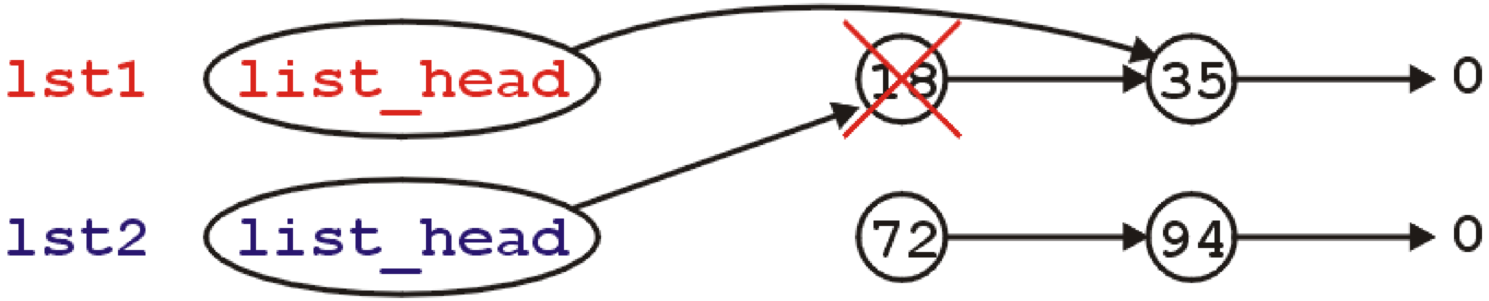
- Also, suppose we call the member function: `lst1.pop_front();`

Problem

- What's wrong with this picture?



- Also, suppose we call the member function: `lst1.pop_front();`
- This only affects the member variable from the object `lst1`

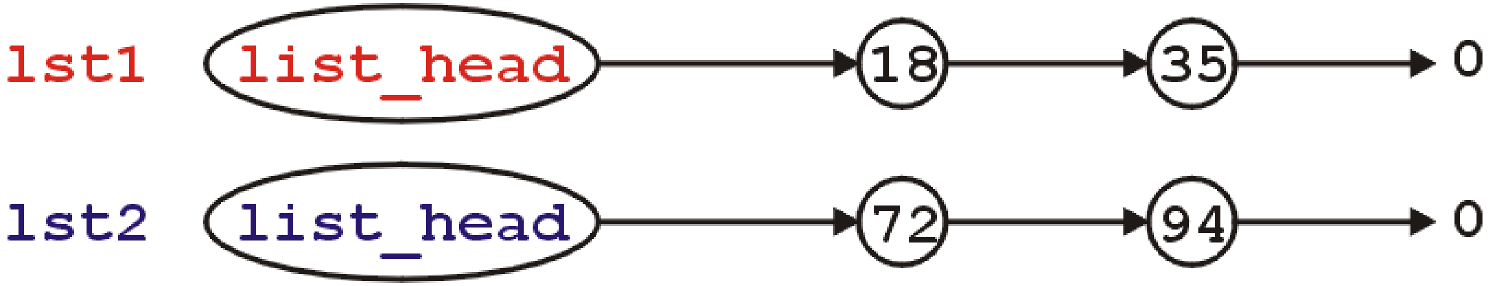


- Now, the second list `lst2` is pointing to memory which has been deallocated...
- What is the behavior if we make this call?

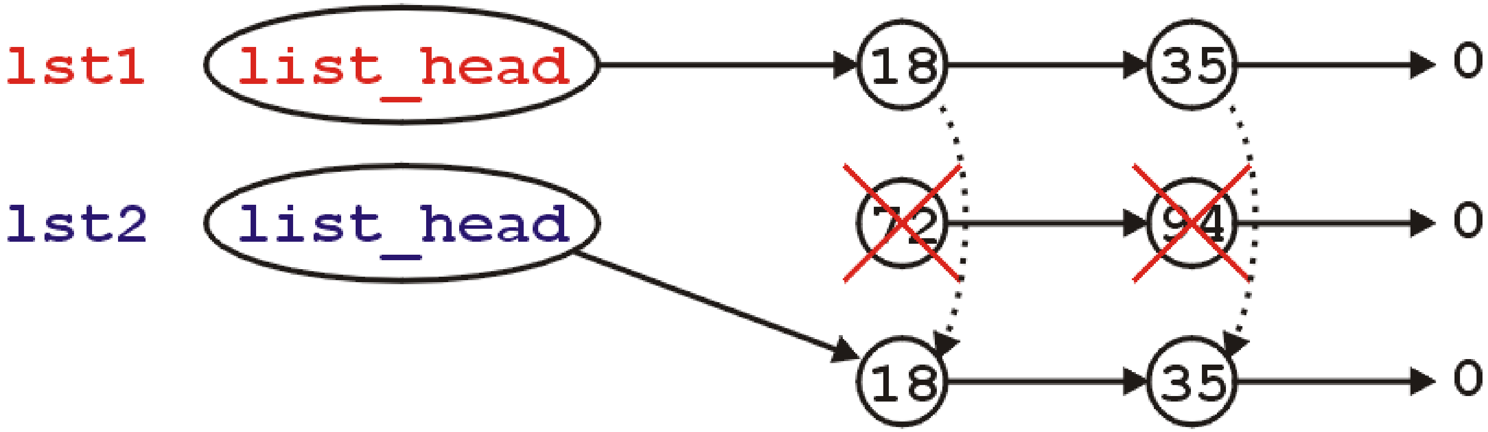
```
lst2.pop_front();
```

- The behavior is undefined, however, soon this will probably lead to an access violation

- 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 Operator

- First, to overload the assignment operator, we must overload the function named `operator=`
    - This is a how you indicate to the compiler that you are overloading the assignment (=) operator
- The signature is:

```
List& operator= ( List );
```

    - The right-hand side `rhs` is passed by *value* (a copy is made)
    - The return value is passed by *reference*

- We will swap all the values of the member variables between the left- and right-hand sides
    - `rhs` is already a copy, so we swap all member variables of it and `*this`

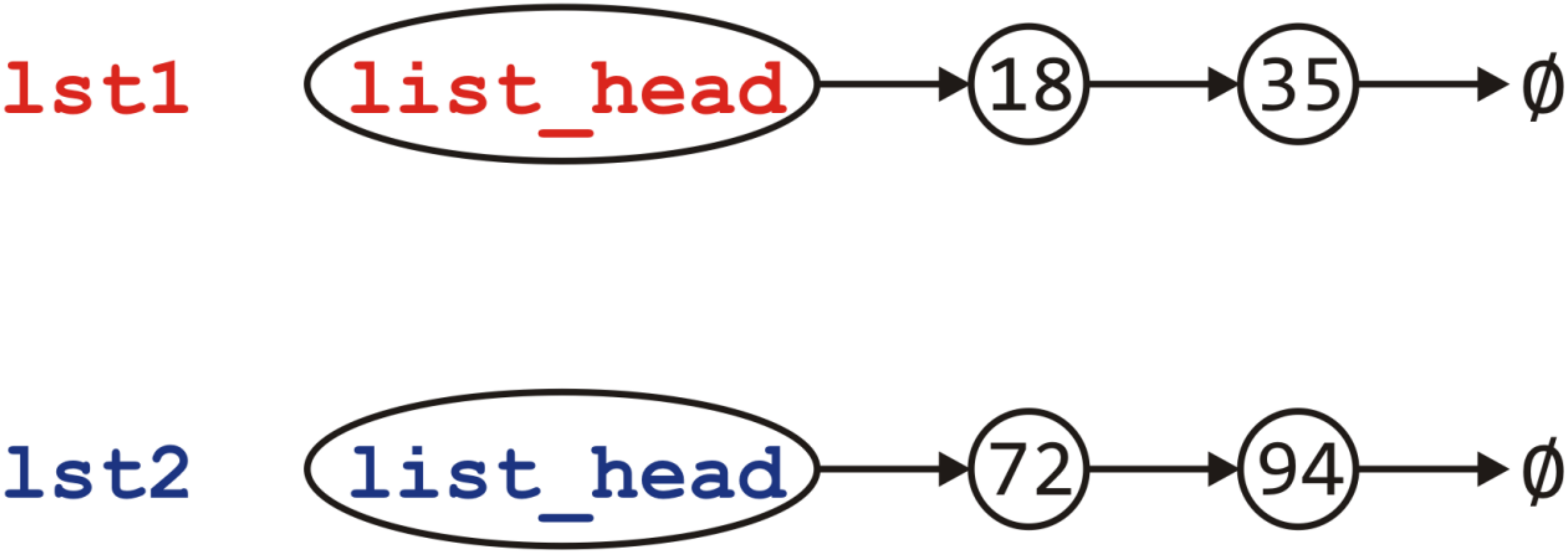
```
List& operator = ( List rhs ) {
    // 'rhs' is passed by value
    // it is a copy of the right-hand side of the assignment
    // copy/move constructor is called to construct `rhs`

    // Swap all the entries of the copy with this

    return *this;
}
```

In [ ]:

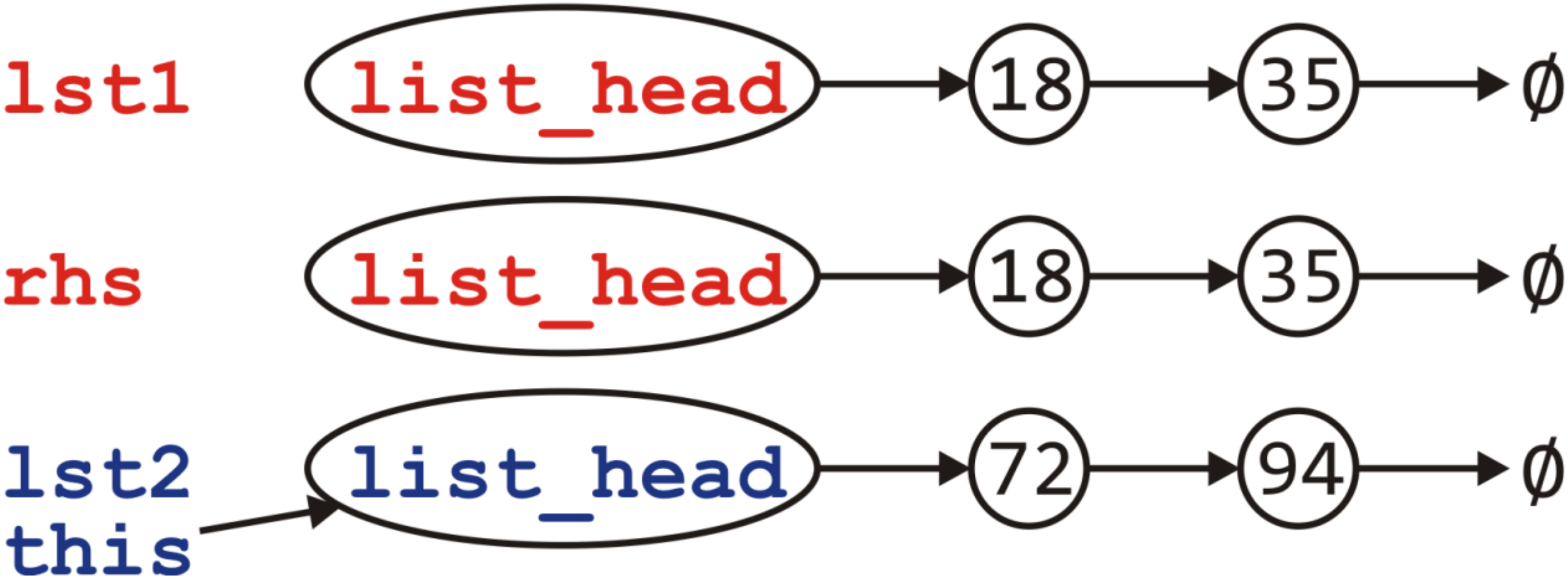
```
List& List::operator= ( List rhs ) {
    std::swap( *this, rhs );
    // Memory for rhs was allocated on the stack
    // and the destructor will delete it
    return *this;
}
```

Copy Assignment

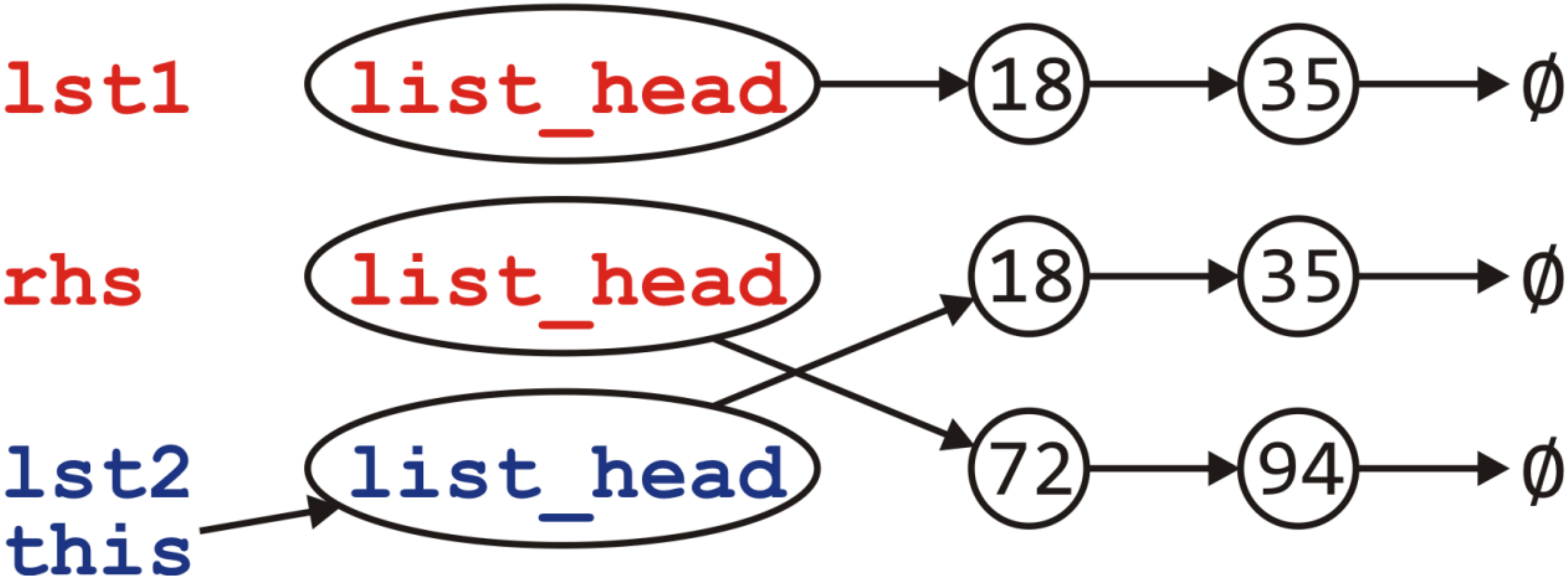
- Visually, we are doing the following:



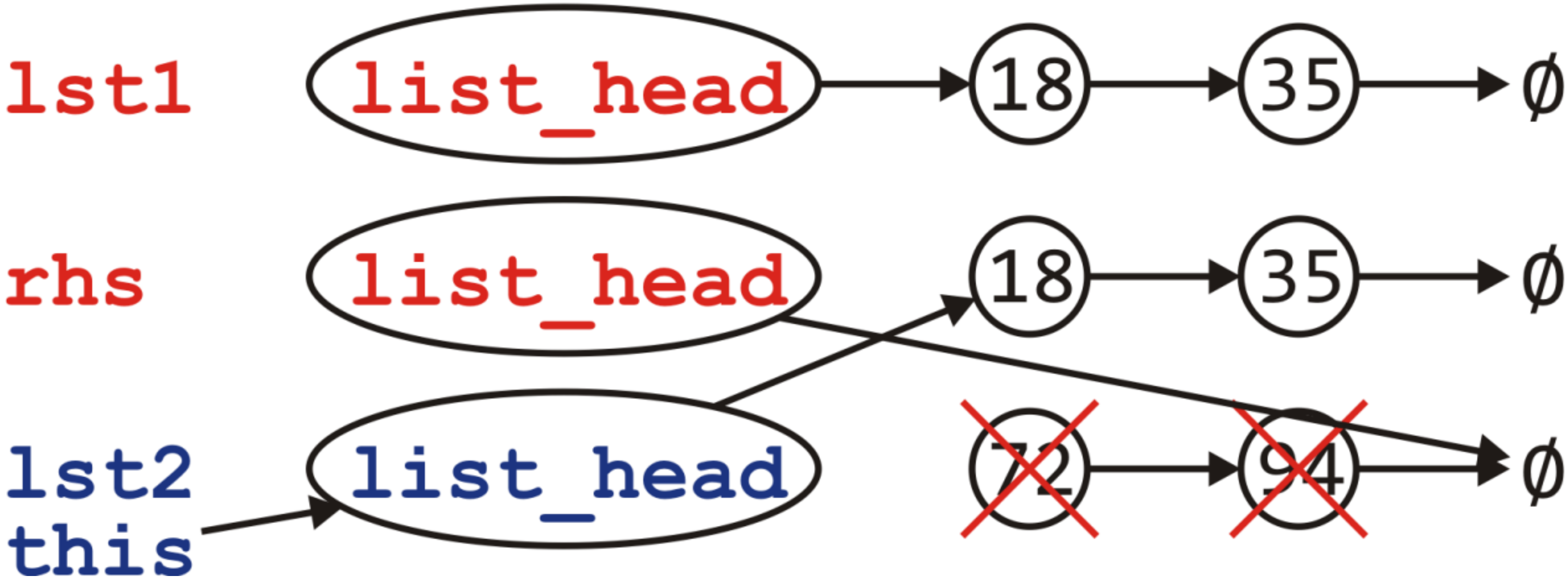
- Visually, we are doing the following:
    - Passed by value, the copy constructor is called to create `rhs`



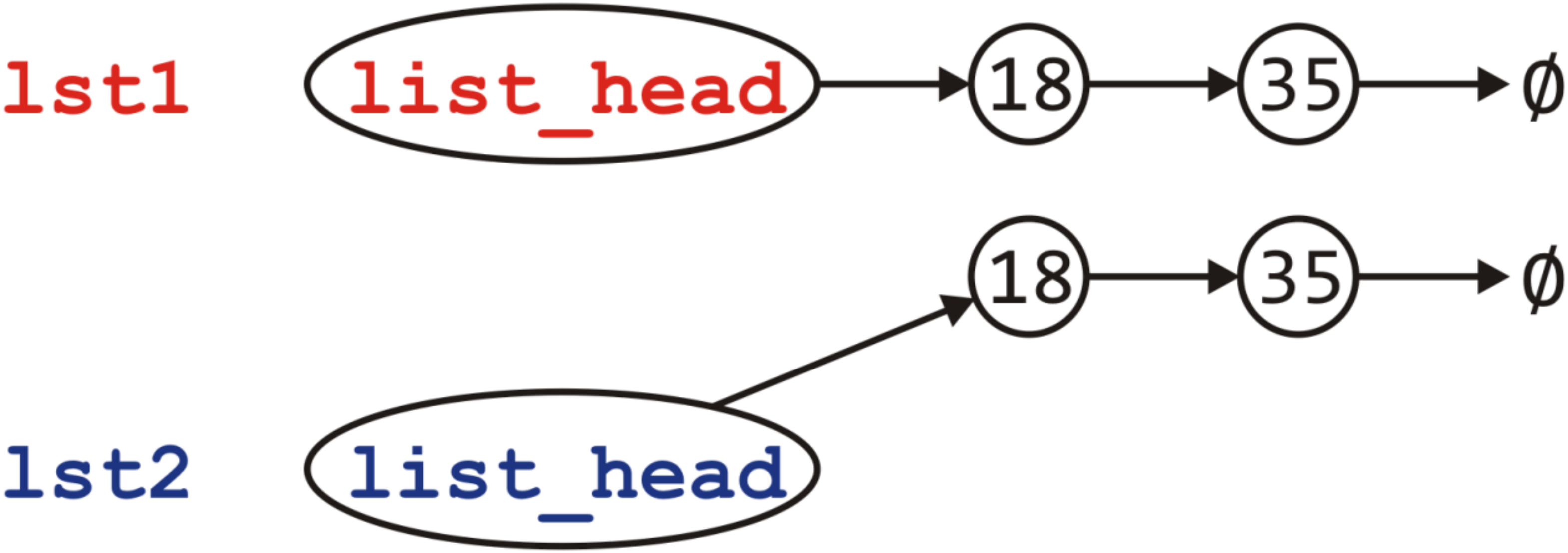
- Visually, we are doing the following:
    - Passed by value, the copy constructor is called to create `rhs`
    - Swapping the member variables of `*this` and `rhs`



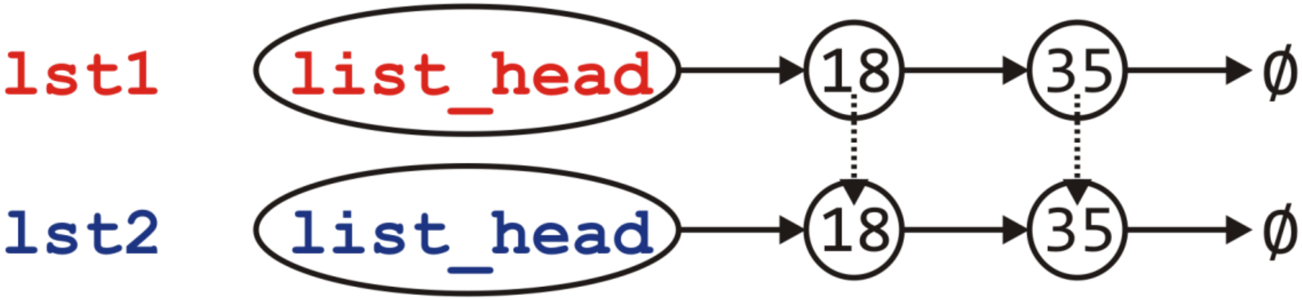
- Visually, we are doing the following:
    - Passed by value, the copy constructor is called to create `rhs`
    - Swapping the member variables of `*this` and `rhs`
    - We return and the destructor is called on `rhs`



- Visually, we are doing the following:
    - Passed by value, the copy constructor is called to create `rhs`
    - Swapping the member variables of `*this` and `rhs`
    - We return and the destructor is called on `rhs`
    - Back in the calling function, the two lists contain the same values



- Can we do better?
    - This idea of *copy and swap* is highly visible in the literature
    - If the copy constructor is written correctly, it will be fast
    - Is it always the most efficient?
- Consider the calls to new and delete
    - Each of these is very expensive
    - Would it not be better to reuse the nodes if possible?



Move Assignment

- Move assignment operators typically **"steal"** the resources held by the argument, rather than make *copies* of them, and leave the argument in some valid but otherwise indeterminate state.

Move Assignment

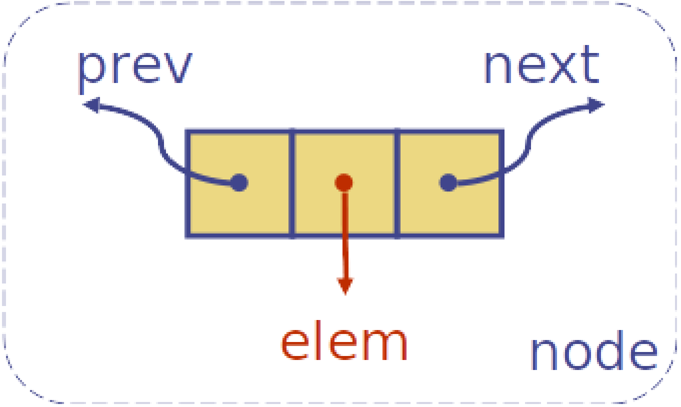
- Move assignment operators typically **"steal"** the resources held by the argument, rather than make *copies* of them, and leave the argument in some valid but otherwise indeterminate state.

In [ ]:

```
List& List::operator= ( List &&rhs ) {
    while ( !empty() ) {
        pop_front();
    }

    list_head = rhs.begin();
    rhs.list_head = nullptr;

    return *this;
}
```

Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored



- Nodes implement **Position ADT** (element at position) and store:
  - element
  - link to the previous node
  - link to the next node

In [31]:

```
class DoubleNode {
public:
    DoubleNode( int e = 0, DoubleNode* p = nullptr, DoubleNode* n = nullptr );

    int value() const;
    DoubleNode* next() const;
    DoubleNode* previous() const;

private:
    int node_value;
    DoubleNode *previous_node;
    DoubleNode *next_node;
};
```

Doubly Linked List

- A doubly linked list provides a natural implementation of the **Node List ADT**
  - We have every node maintain a link to its previous node in the list
  - Also, special trailer and header sentinel nodes can be added

Consider this simple (but **incomplete**) doubly linked list class:

Consider this simple (but **incomplete**) doubly linked list class:

In [32]:

```
class DoublyList {
public:
    // we defined it outside of the List class scope
    //class DoubleNode {...};
    DoublyList();
    ~DoublyList();

    // Accessors
    bool empty() const;
    int size() const;
    int front() const;
    int back() const;
    Node* begin() const;
    Node* end() const;

    // Mutators
    void push_front( int );
    void push_back( int );
    int pop_front();
    int pop_back();

    // Misc
    int count( int ) const;
    int erase( int );

private:
    DoubleNode *list_head;
    DoubleNode *list_tail;
};
```

Insertion

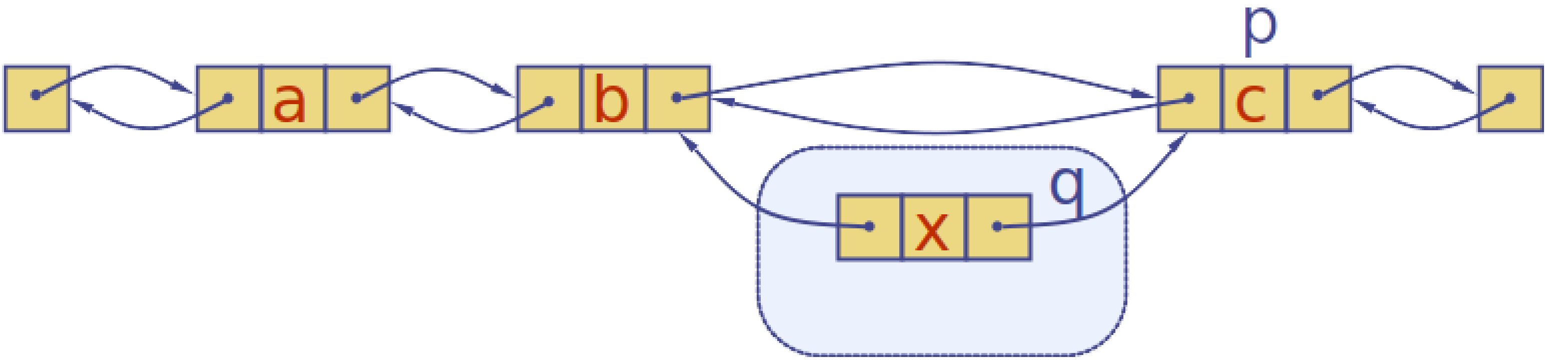
- Because of its double link structure, it is possible to insert a node at **any** position within a doubly linked list.

Insertion

- Because of its double link structure, it is possible to insert a node at **any** position within a doubly linked list.

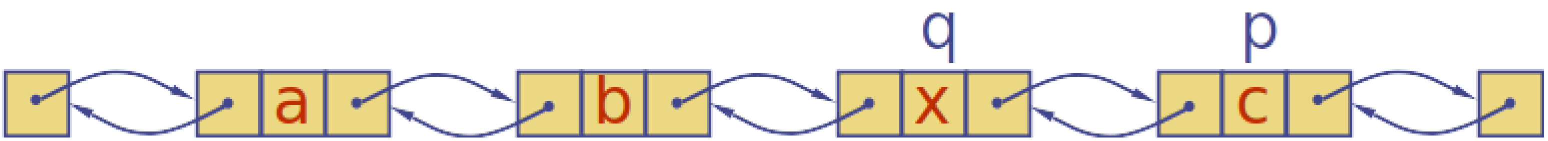
void DoubleList::insert(DoubleNode &p, const int &x) {

DoubleNode *q = new DoubleNode{ x, p->prev, p };



p->prev = p->prev->next = q;
</div> }

```
void DoubleList::insert( DoubleNode &p, const int &x ) {

    DoubleNode *q = new DoubleNode{ x, p->prev, p };
    p->prev = p->prev->next = q;
```



```
    </div> }
```

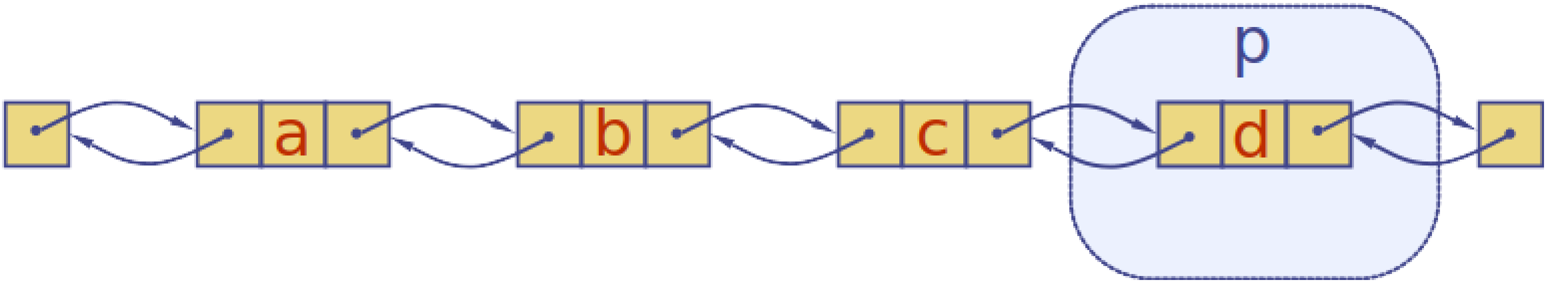
Deletion

- If `p` points to the node being removed, only two pointers change before the node can be reclaimed:

Deletion

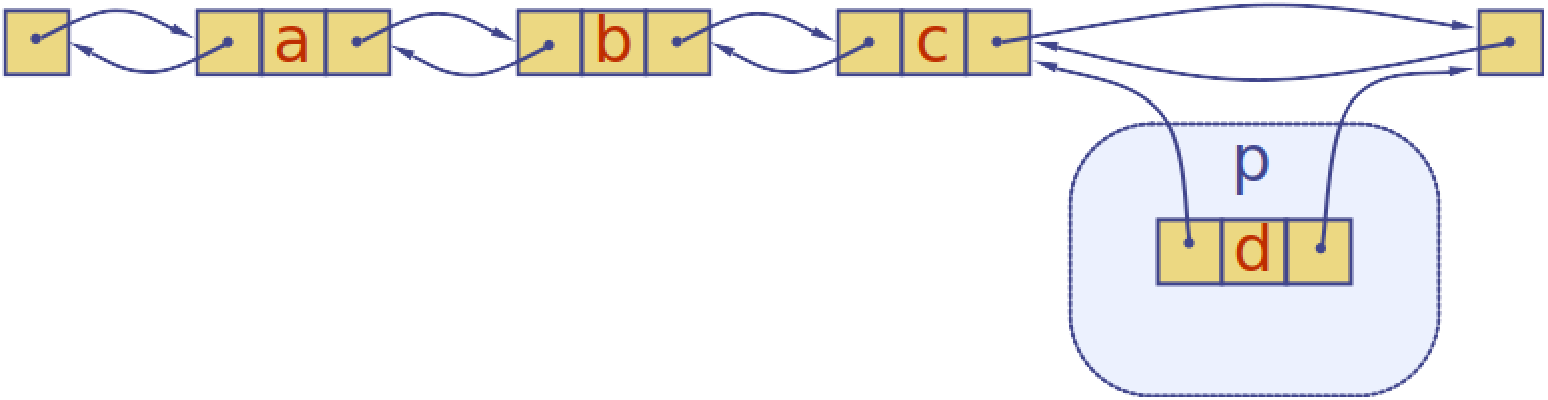
- If `p` points to the node being removed, only two pointers change before the node can be reclaimed:

```
void DoubleList::remove( DoubleNode &p ) {

   p->prev->next = p->next; // linking out of p.
```



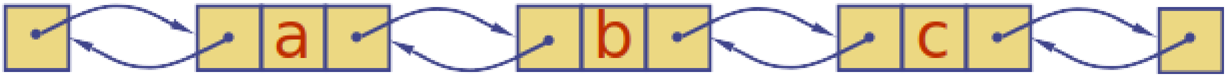
```
   p->next->prev = p->prev;

   delete p; </div> }
```

```
void DoubleList::remove( DoubleNode &p ) {

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



```
   delete p; </div> }
```

```
void DoubleList::remove( DoubleNode &p ) {

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



```
    delete p;
    </div> }
```

Sentinels

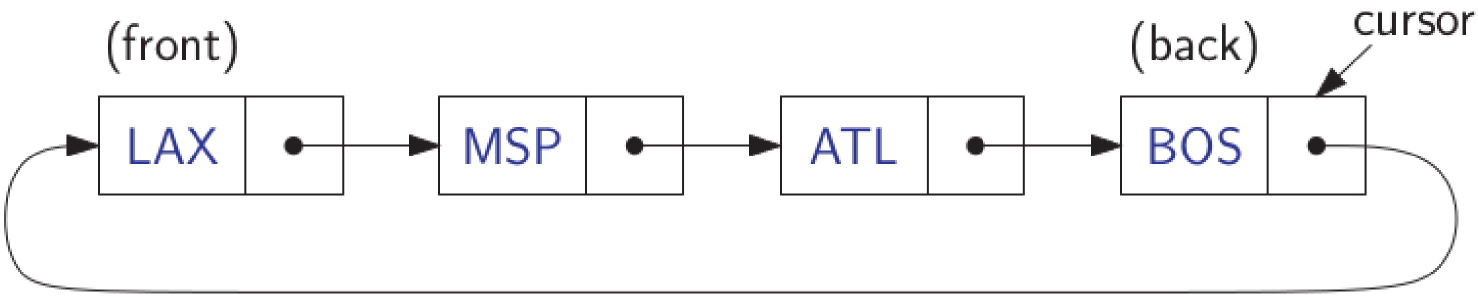
- It is convenient to add special nodes at both ends of a doubly linked list:
  - a header node just before the head of the list, and
  - a trailer node just after the tail of the list.

- These "dummy" or sentinel nodes do not store any elements.

- They provide quick access to the first and last nodes of the list.
  - the header's next pointer points to the first node of the list, and
  - the trail pointer of the trailer node points to the last node of the list.

Circular Linked List

- A **circularly linked list** has the same kind of nodes as a singly linked list.
- Each node in a circularly linked list has a next pointer and an element value, but **no** `head` or `tail`.



- A special node is marked as the **cursor**.
    - The cursor node allows us to have a place to starting point in the list.
    - The element that is referenced by the cursor, which is called the **back**, and
    - The element *immediately following* it in the circular order, which is called the **front**.

In [49]:

```
class CircularList {
public:
    CircularList() : cursor{nullptr} {}
    ~CircularList() { while (!empty()) pop(); }

    // Accessors
    bool empty() const { return cursor == nullptr; }
    int front() const { return cursor->next()->value(); }
    int back() const { return cursor->value(); }

    // Mutators
    void push( int );
    void pop();
private:
    Node *cursor; // head pointer of the list
};
```

In [ ]:

```
void CircularList::push(int e) {
    Node* tmp = new Node(e, nullptr);
    if ( empty() )
        // link node to itself
        cursor = tmp->next_node = tmp;
    else {
        // point new node to the next from the cursor
        tmp->next_node = cursor->next();
        // point cursor to new node
        cursor->next_node = tmp;
    }
}
```

In [ ]:

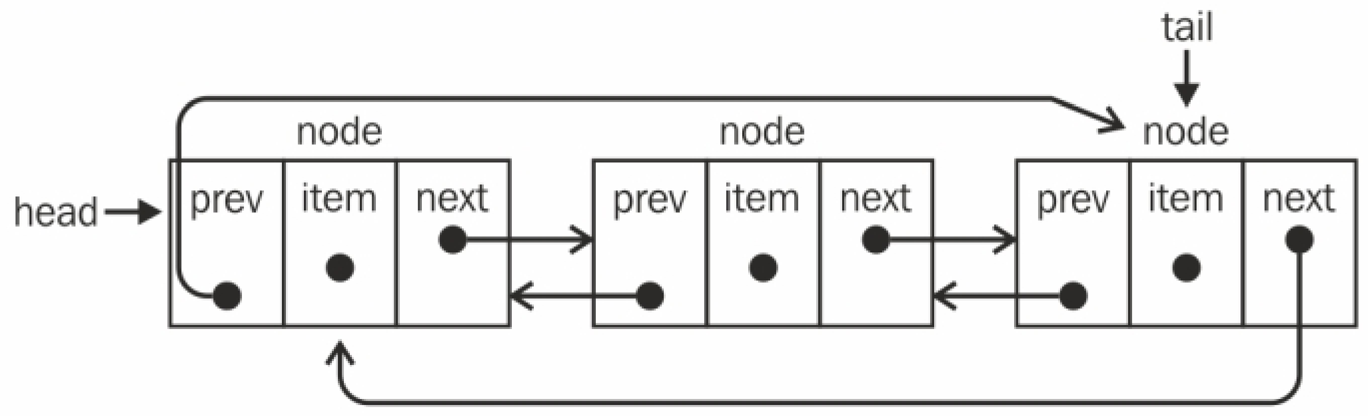
```
void CircularList::push(int e) {
    Node* tmp = new Node(e, nullptr);
    if ( empty() )
        // link node to itself
        cursor = tmp->next_node = tmp;
    else {
        // point new node to the next from the cursor
        tmp->next_node = cursor->next();
        // point cursor to new node
        cursor->next_node = tmp;
    }
}
```

In [ ]:

```
void CircularList::pop() {
    Node* old = cursor->next();
    if (old == cursor)
        // remove only element from the list, it points to itself
        cursor = nullptr;
    else
        // remove next element from cursor
        cursor->next_node = old->next();
    delete old;
}
```

Doubly Circular Linked List

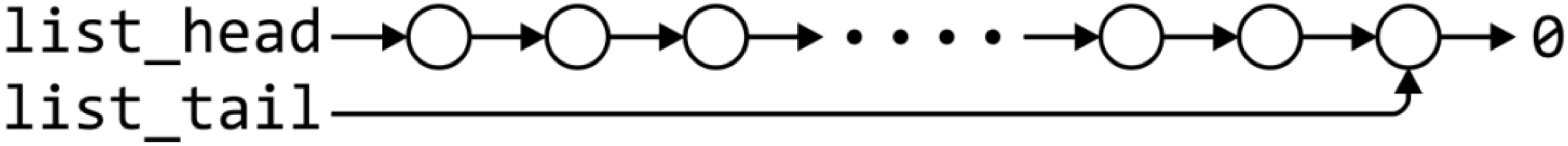
- In a doubly circular linked list has `tail->next` pointing to the `head` element and `head->prev` pointing to the `tail` element



Locations and run times

- The most obvious data structures for implementing an abstract list are **arrays** and **linked lists**
    - We will review the run time operations on these structures
- We will consider the amount of time required to perform actions such as finding, inserting new entries before or after, or erasing entries at
    - the first location (the front)
    - an arbitrary (kth) location
    - the last location (the back or nth)
- The run times will be $\Theta(1)$, $O(n)$ or $\Theta(n)$

Singly Linked List

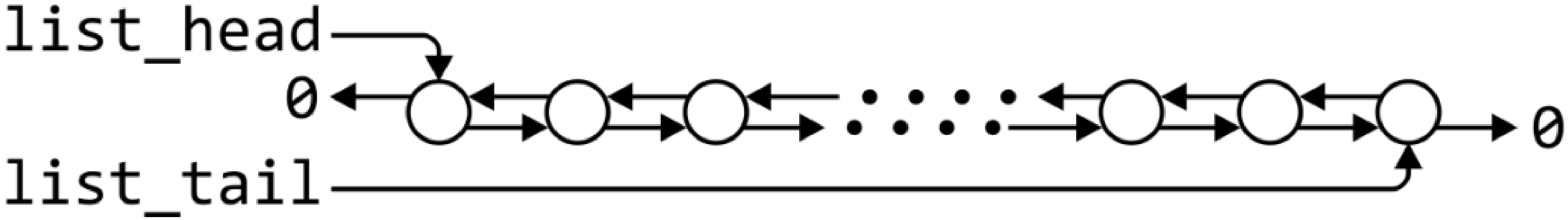


- With asymptotic analysis of linked lists, we can now make the following statements:

	front / 1st node	**arbitrary / kth node**	**back / nth node**
find	$\Theta(1)$	$O(n)$	$\Theta(1)^1$
insert before	$\Theta(1)$	$O(n)$	$\Theta(n)$
insert after	$\Theta(1)$	$\Theta(1)^2$	$\Theta(1)^1$
replace	$\Theta(1)$	$\Theta(1)^2$	$\Theta(1)^1$
erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
next	$\Theta(1)$	$\Theta(1)^2$	n/a
previous	n/a	$O(n)$	$\Theta(n)$

- ^{1} These become $\Theta(n)$ if we don't have a tail pointer
- ^{2} These assume we have already accessed the kth entry - an $O(n)$ operation

Doubly Linked List



- The asymptotic analysis of doubly linked lists shows:

	front / 1st node	**arbitrary / kth node**	**back / nth node**
find	$\Theta(1)$	$O(n)$	$\Theta(1)$
insert before	$\Theta(1)$	$\Theta(1)^1$	$\Theta(1)$
insert after	$\Theta(1)$	$\Theta(1)^1$	$\Theta(1)$
replace	$\Theta(1)$	$\Theta(1)^1$	$\Theta(1)$
erase	$\Theta(1)$	$\Theta(1)^1$	$\Theta(1)$
next	$\Theta(1)$	$\Theta(1)^1$	n/a
previous	n/a	$\Theta(1)^1$	$\Theta(1)$

- ^{1} These assume we have already accessed the kth entry - an $O(n)$ operation

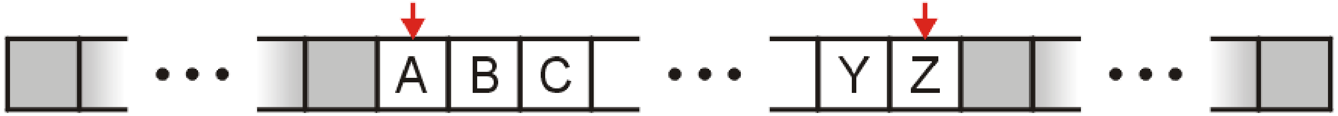
Other operations on linked lists

- Allocation and deallocating the memory requires $\Theta(n)$ time
- Concatenating two linked lists can be done in $\Theta(1)$
    - This requires a tail pointer

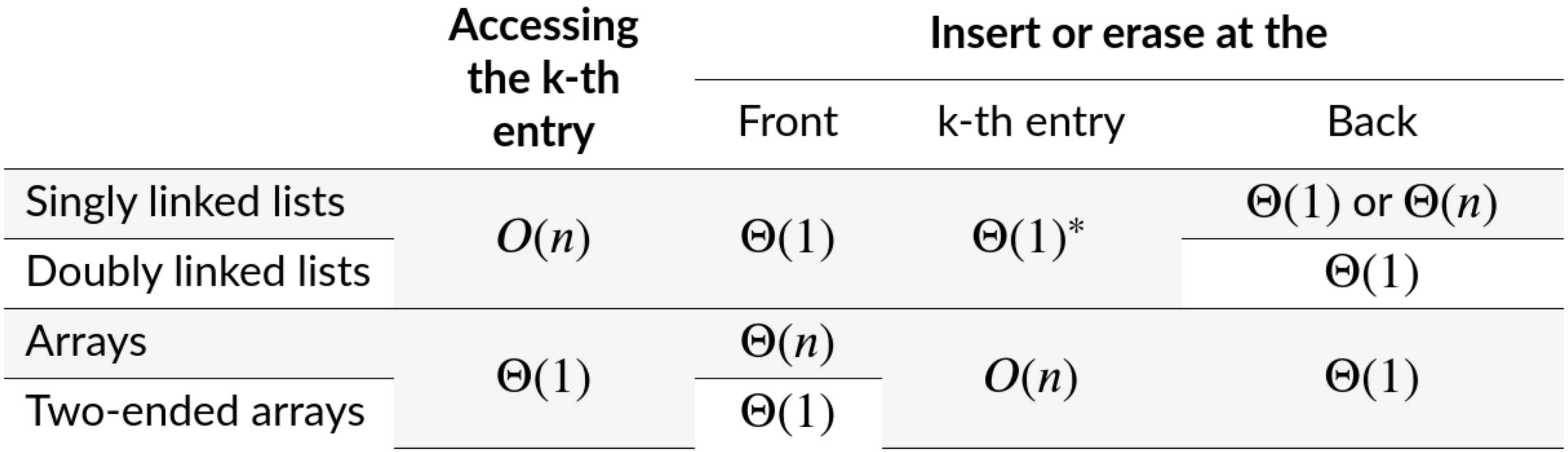
Arrays

- Consider these operations for arrays, including
  - Standard or one-ended arrays

  - Two-ended arrays



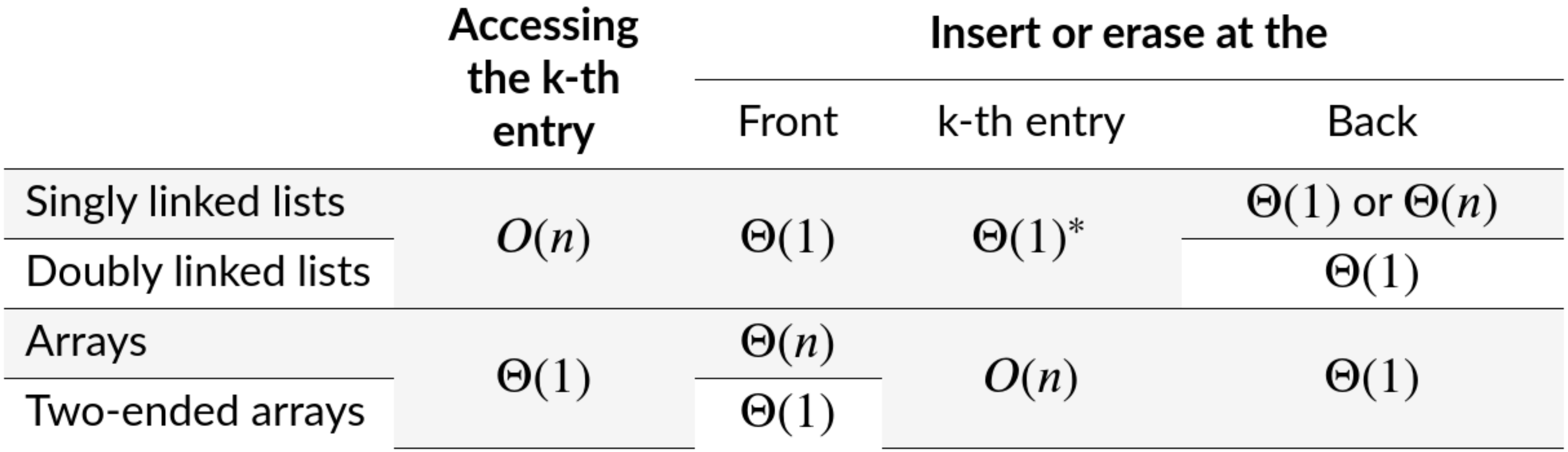
Run times



	Accessing the k-th entry	**Insert or erase at the**		
		Front	k-th entry	Back
Singly linked lists	$O(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ or $\Theta(n)$
Doubly linked lists				$\Theta(1)$
Arrays	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Two-ended arrays		$\Theta(1)$		

- * Assume we have a pointer to this node

Run times



	Accessing the k-th entry	Insert or erase at the		
		Front	k-th entry	Back
Singly linked lists	$O(n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$ or $\Theta(n)$
Doubly linked lists				$\Theta(1)$
Arrays	$\Theta(1)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Two-ended arrays		$\Theta(1)$		

- * Assume we have a pointer to this node
- In general, we will only use these basic data structures if we can restrict ourselves to operations that execute in $\Theta(1)$ time, as the only alternative is $O(n)$ or $\Theta(n)$

Memory usage versus run times

- All of list data structures require $\Theta(n)$ memory
- Using a two-ended array requires one more member variable,$\Theta(1)$, in order to significantly speed up certain operations
- Using a doubly linked list, however, required $\Theta(n)$ additional memory to speed up other operations