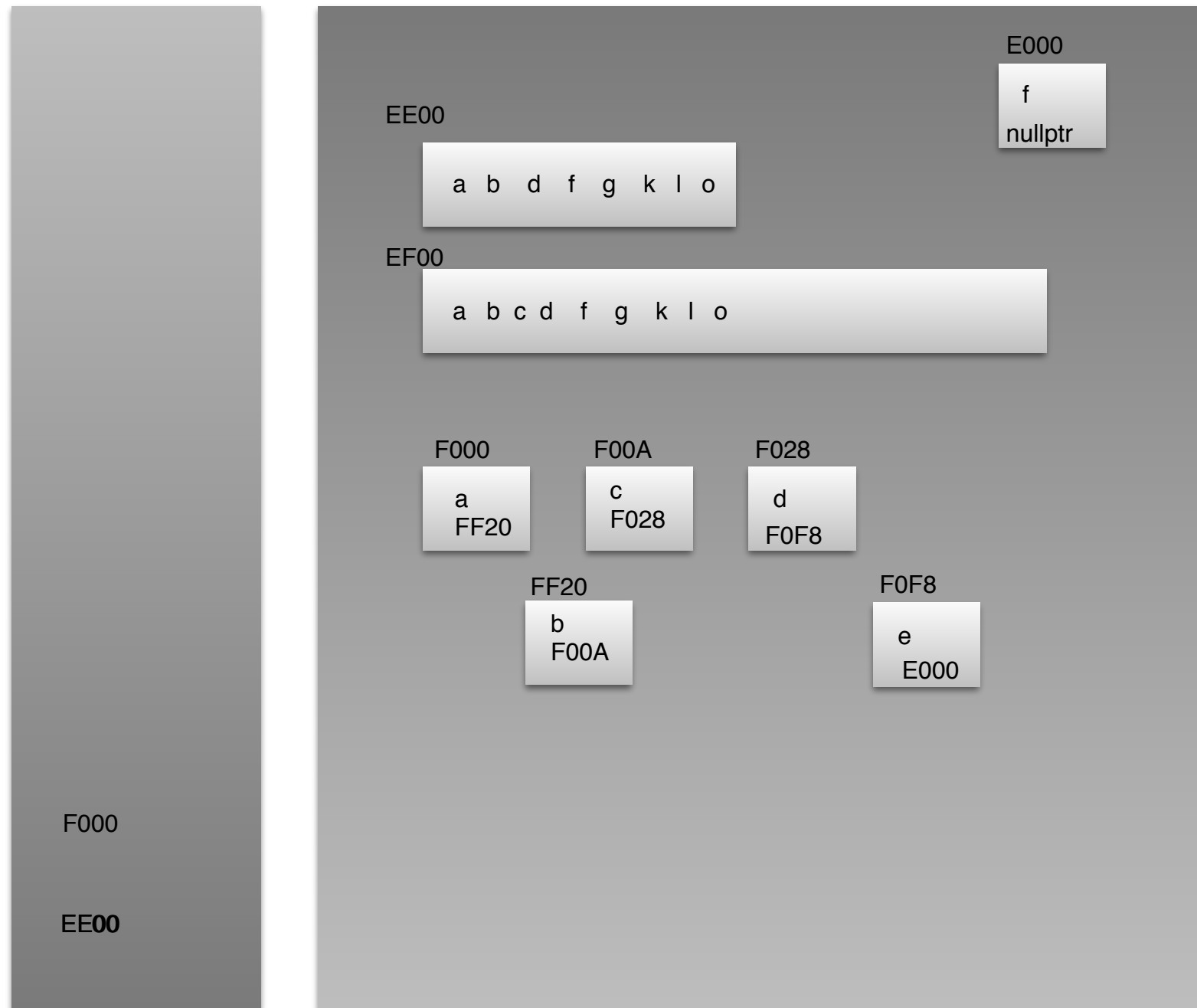


Lecture 13

Linked Lists

Thinking about ADT list



Vectors

- allow fast, convenient access to elements using []
- inserting new elements in the front or middle is expensive

List ADT

$A_1, A_2, A_3, \dots, A_n,$

This is not a complete list

STL methods

vector & list methods

```
int size() const
void clear()
bool empty() const
iter begin()
iter end()

void push_back( const Object & x)
void pop_back()

const Object & back() const
const Object & front() const
iter insert( iter pos, const Object & x )
iter insert( iter pos, Object && x )
iter erase( iter pos )
iter erase( iter start, iter end)
__ operator=( const __ & rhs)
```

List methods

```
void push_front( const Object & x)
void pop_front()
void sort( )
template<class Pred>
void sort( Pred pred )
```

Vector methods

```
Object & operator[](int idx)
int capacity() const
void reserve(int newCapacity)
```

- Forward_list
- fast
 - many methods not included in the forward_list for performance reasons

List ADT

$A_1, A_2, A_3, \dots, A_n,$

This is not a complete list

STL methods

forward_list & list methods

```
void clear()
bool empty() const
iter begin()
iter end()
void push_front( const Object & x)
void pop_front()
const Object & front() const
void remove( const Object & x)
___ operator=( const ___ & rhs)
void sort( )
template<class Pred>
void sort( Pred pred )
```

List methods

```
iter erase( iter pos )
iter erase( iter start, iter end)
int size() const
iter insert( iter pos, const Object & x )
iter insert( iter pos, Object && x )
void push_back( const Object & x)
void pop_back()
const Object & back() const
```

forward_list methods

```
itr before_begin() const
itr erase_after( itr start)
itr insert_after( itr start)
```

Linked Lists

- Arranged in a **linear** order
 - Order determined by a pointer to next object
 - Data stored in a node
- Advantages:
 - Flexible, add/delete easily
 - Dynamic, no wasted memory
 - No slowdowns as in array when the memory is doubled
 - Changes made by using only a constant number of data movements
- Disadvantages:
 - Linear time to get to i^{th} element - sequential access
 - Store address of next item

List Basics

A node is a self referential object; it contains data and a pointer to another node

singly linked list

```
template<class Object>
```

```
struct Node
```

```
{
```

```
    Object data;
```

```
    Node * next;
```

```
}
```

```
Node * frontOfList = new Node<char>;
```

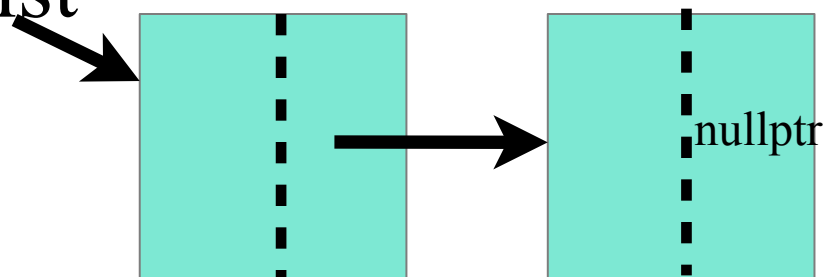
```
frontOfList->next = nullptr;
```

```
frontOfList->next = new Node<char>;
```

```
frontOfList->next->next = nullptr;
```

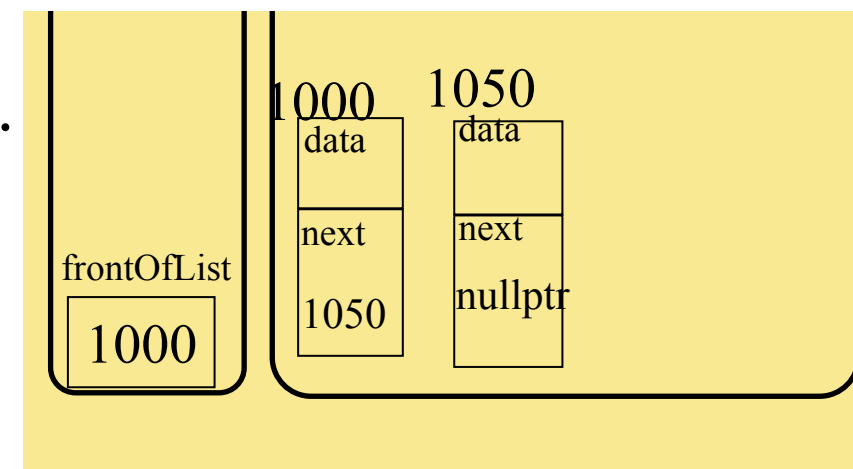
Conceptual Representation

frontOfList



CS2134

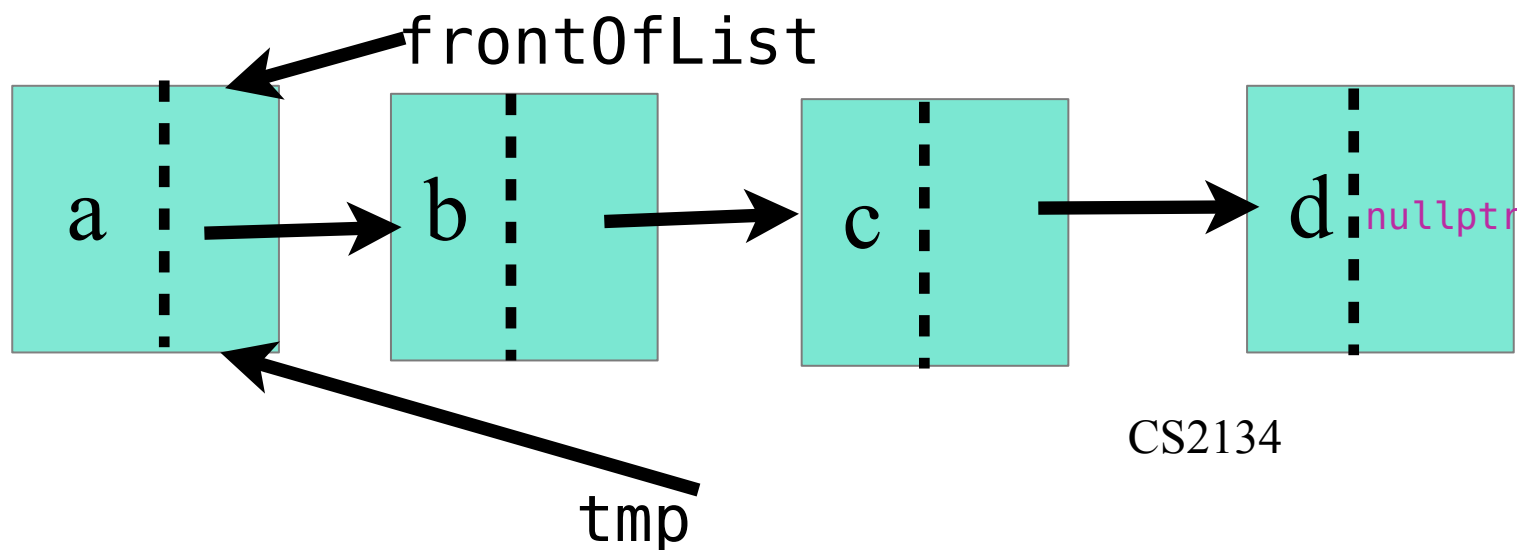
Memory Level



```
template<class Object>
struct Node
{
    Object data;
    Node* next;
};
```

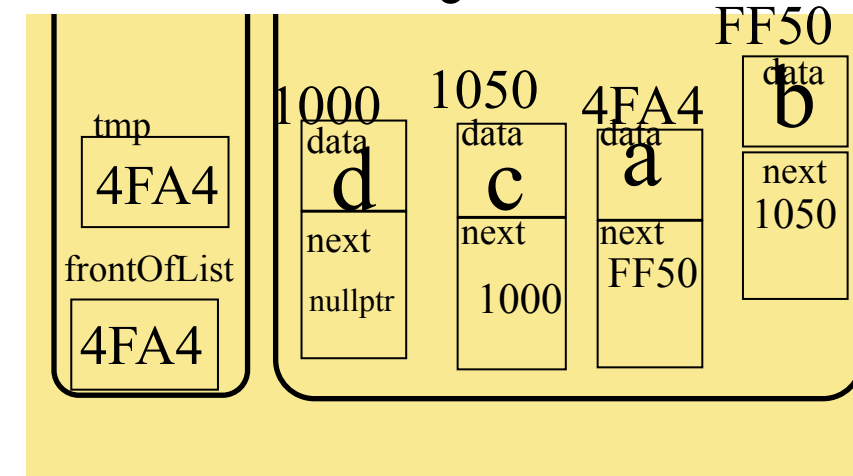
```
int main ()
{
    Node<char>* frontOfList = nullptr;
    for (char ch = 'd'; ch >='a'; --ch)
    {
        Node<char>* tmp = new Node<char>;
        tmp->data = ch;
        tmp->next = frontOfList;
        frontOfList = tmp;
    }
```

Conceptual Representation



ADT a b c d

Memory Level



```

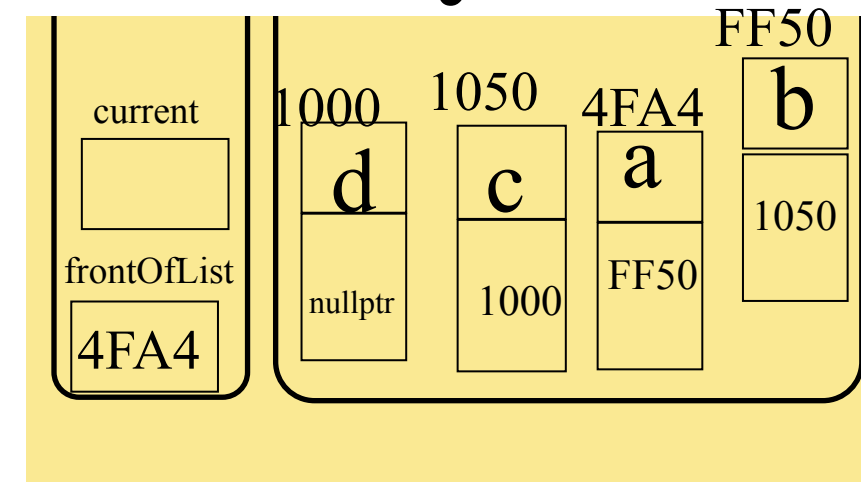
template<class Object>
struct Node
{
    Object data;
    Node* next;
};

int main ()
{
    // code to enter 'd', 'c', 'b', and 'a'
    // where frontOfList points to 'a'.
    Node<char>* current = frontOfList;
    for (; current != nullptr; current = current->next)
        cout << current->data << ", ";
}

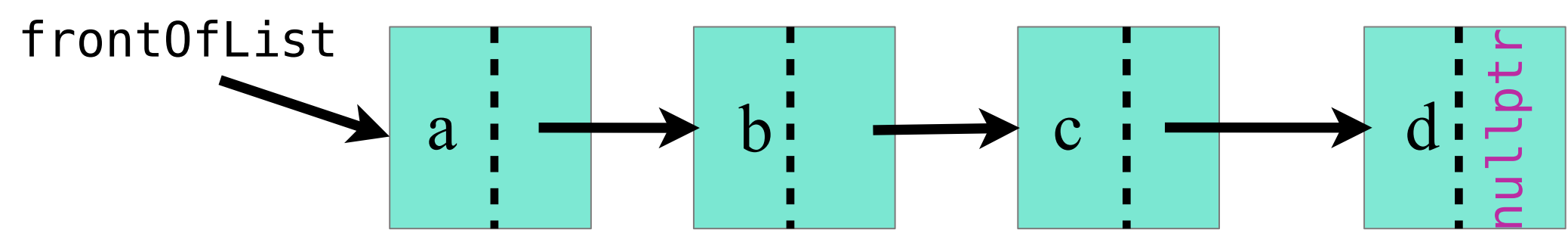
```

ADT a b c d

Memory Level



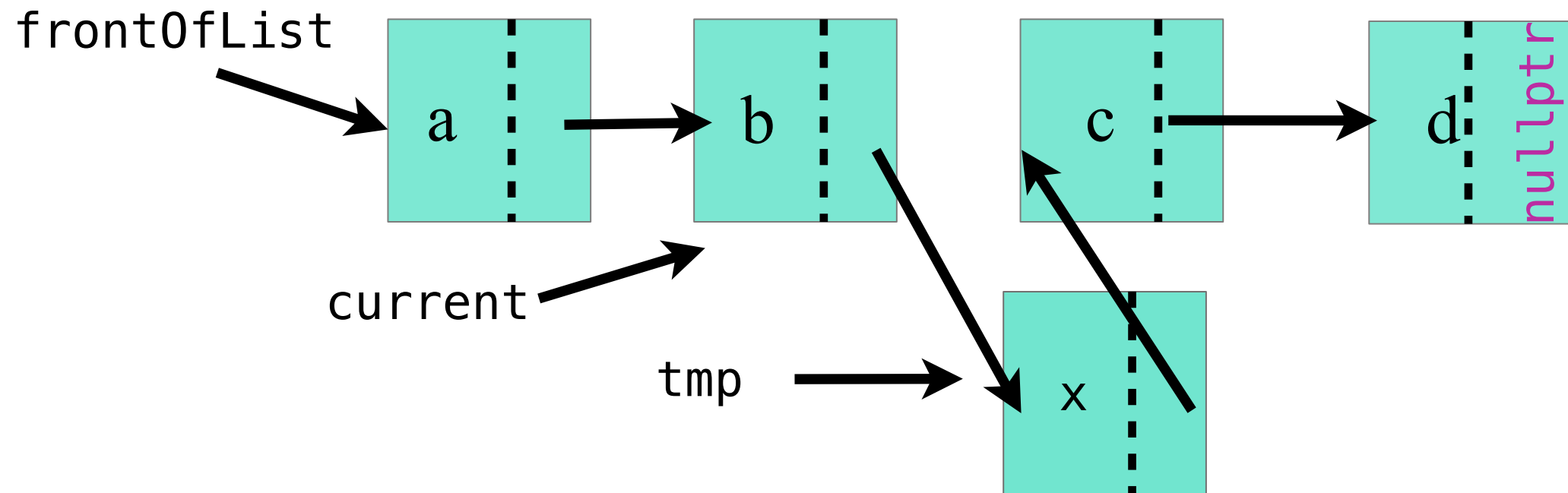
Conceptual Representation



Linked Lists

- Review of basic forward list ops:
 - insert
 - `insert_after(itr, x)`
 - `push_front(x)`
 - erase
 - `erase_after(itr)`
 - `pop_front()`
 - `remove(x)`
 - traversing a list
 - ?
- Design issues
 - with or without header nodes
 - iterator awareness

Insertion of 'x' after a node in the list



```
Node<char>* current = frontOfList->next;
```

```
Node<char>* tmp;
```

```
tmp = new Node<char>;
```

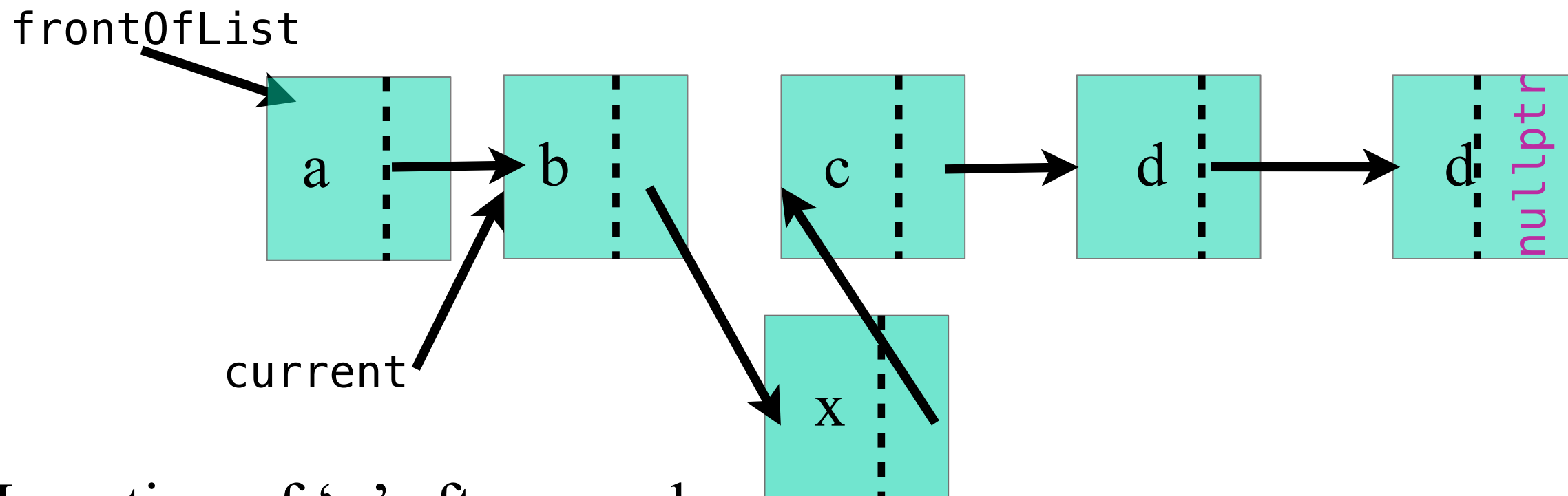
```
tmp->data = 'x';
```

```
tmp->next = current->next;
```

```
current->next = tmp;
```

Simplifying the code after adding a **constructor** that initializes the data

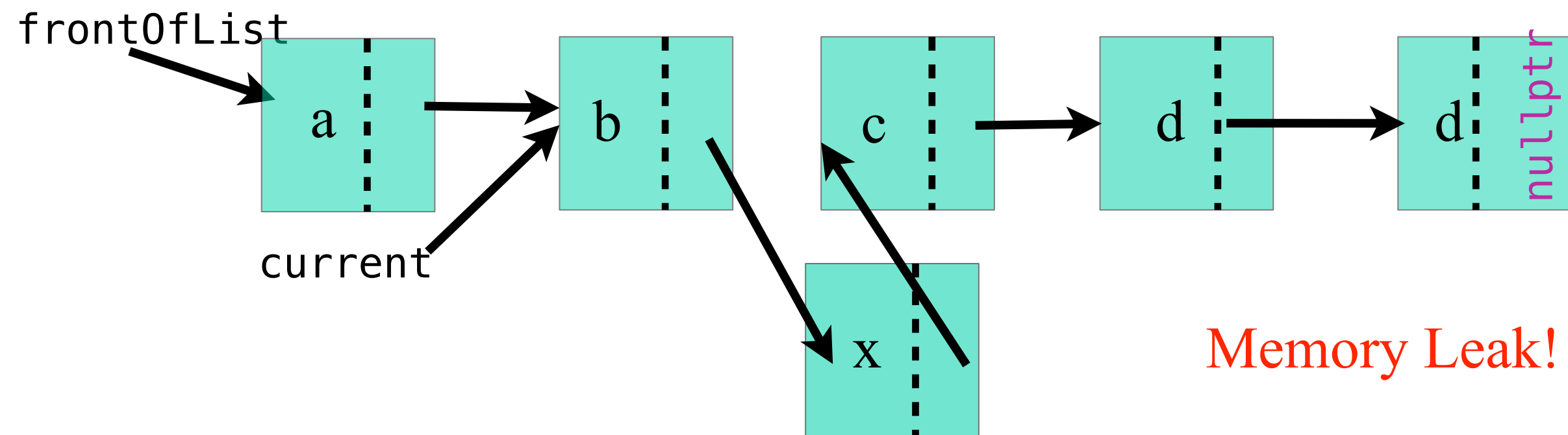
```
template<class Object>
struct Node
{
    Object data;
    Node* next;
    Node(const Object & in = Object(), Node* ptr = nullptr):data(in),next(ptr){}
    Node( Object && d, Node * n = nullptr ): data{ std::move( d ) }, next{ n } { }
};
```



Insertion of 'x' after a node:

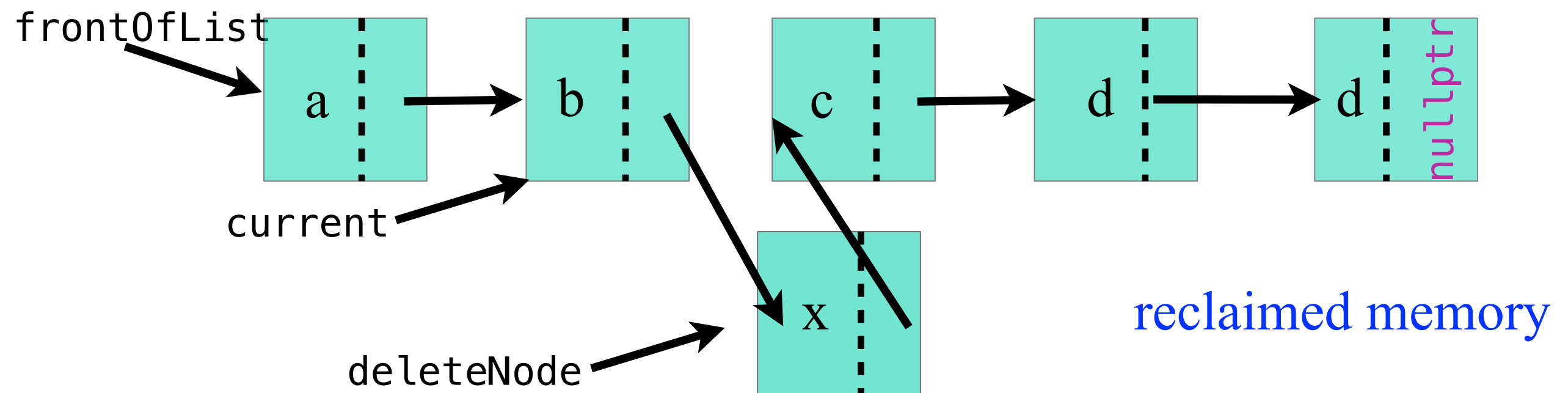
```
current->next = new Node<char>(char('x'), current->next);
```

Deletion of a node in the list, **after** a pointer to the previous node



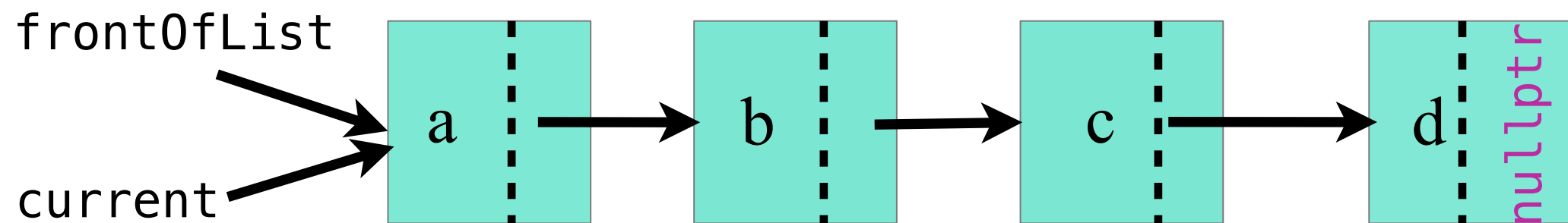
```
current->next = current->next->next;
```

Deletion of a node in the list, **after** a pointer to the previous node



```
Node<char>* deleteNode = current->next;  
current->next = current->next->next;  
delete deleteNode;  
deleteNode = nullptr;
```

Insertion and Deletion at the Front of the List

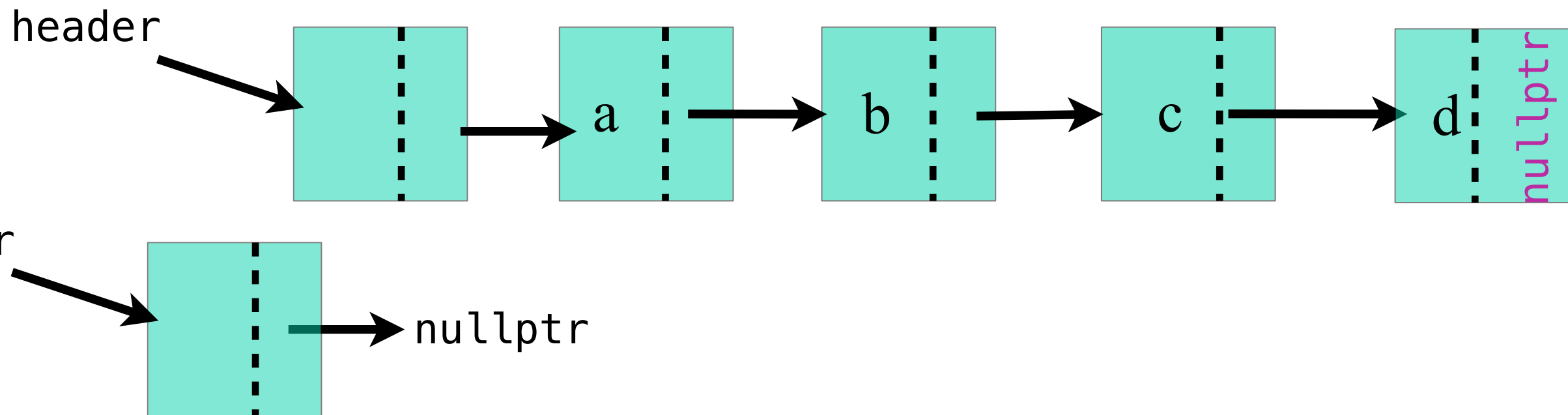


Special Case

Header Node

Header Node

Extra node that holds no data. Created so that every node containing an item has a previous node.

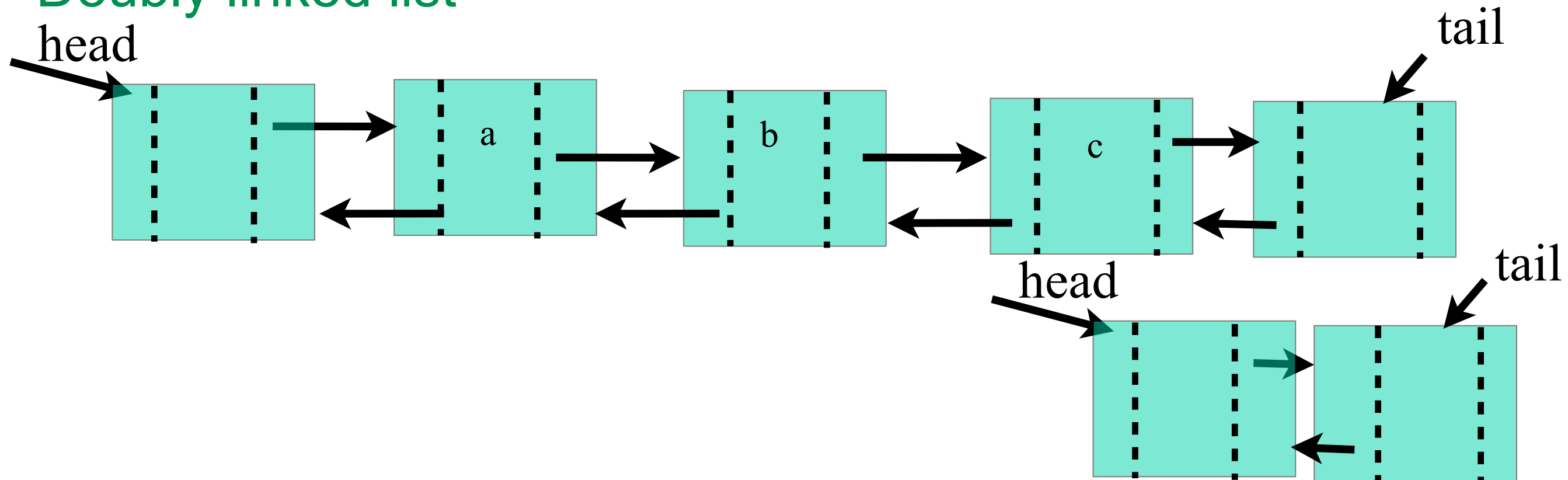


No Special Case

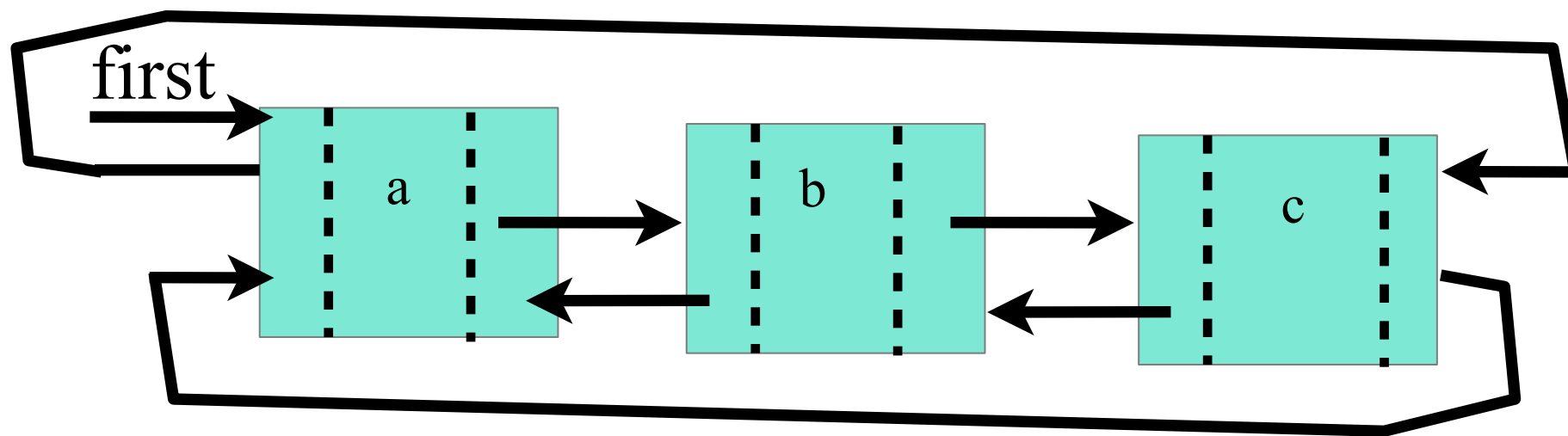
Other Choices

- singly vs doubly linked list
- header and tail node
- circular lists

Doubly linked list



Circular doubly linked list



C++ Implementation of a Singly Linked List

(not the same as the STL)

- Node class
- List class
- iterator class

C++ Implementation of a Singly Linked List

(not the same as the STL)

- Node class
- List class
- iterator class (++itr, itr++, *itr, ==, !=)

The Node Class (Inside the List Class)

```
struct Node
```

```
{
```

```
    Object data;
```

```
    Node * next;
```

```
    Node( const Object & d = Object{ }, Node * n = nullptr )
```

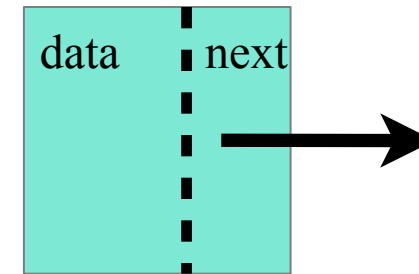
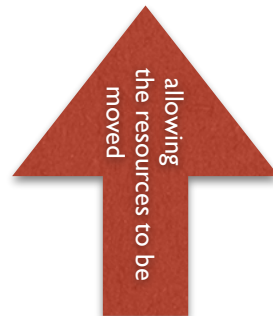
```
    : data{ d }, next{ n }{ }
```

d is an lvalue whose type is an
rvalue reference

```
    Node( Object && d, Node * n = nullptr )
```

```
    : data{ std::move( d ) }, next{ n }{ }
```

```
};
```



```

template <class Object>
class List
{

private:
    struct Node
    { // on previous slide
    };

public:
    class iterator
    { // later in the slides
    };

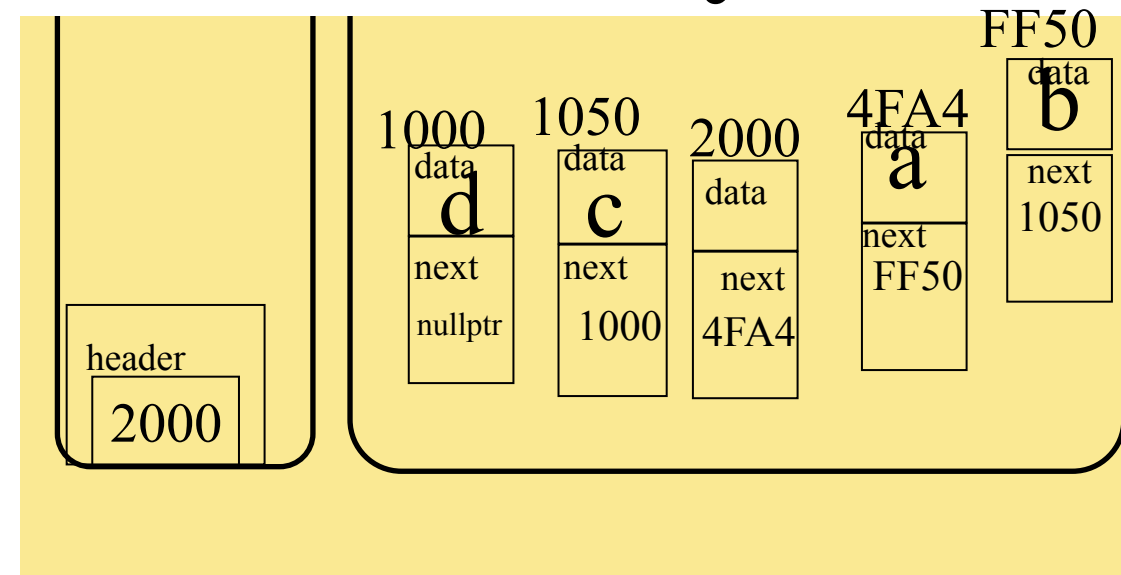
public:
    // The big five

    // other methods ...

private:
    LListNode<Object> *header;
};

```

Memory Level



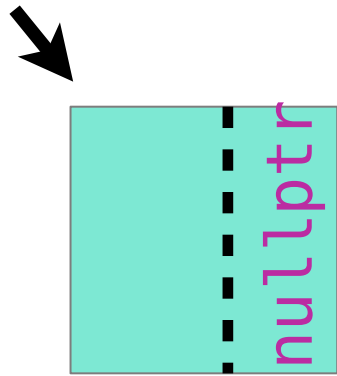
```

List<char> L;
for (char ch = 'c'; ch >='a'; --ch)
    L.push_front( ch );

```

constructor: `LList()`

header



// Construct the list.

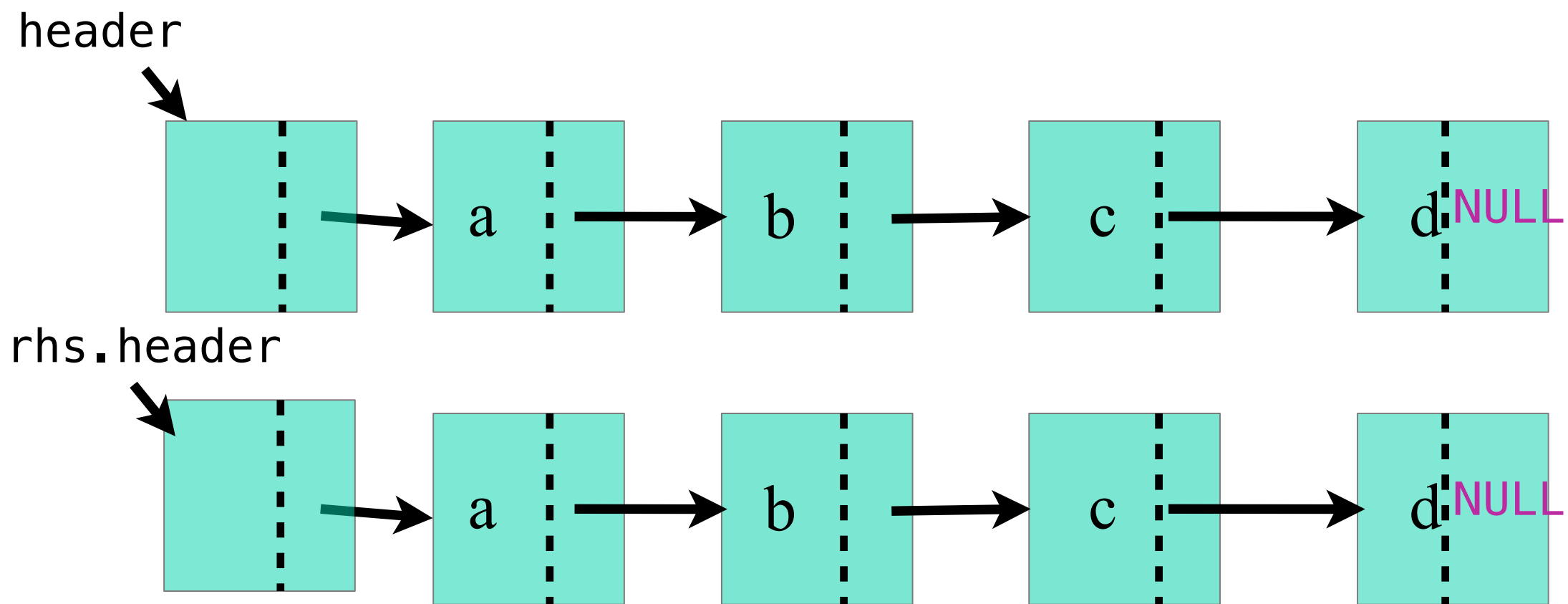
`LList()`

```
{  
    header = new Node;  
}
```

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

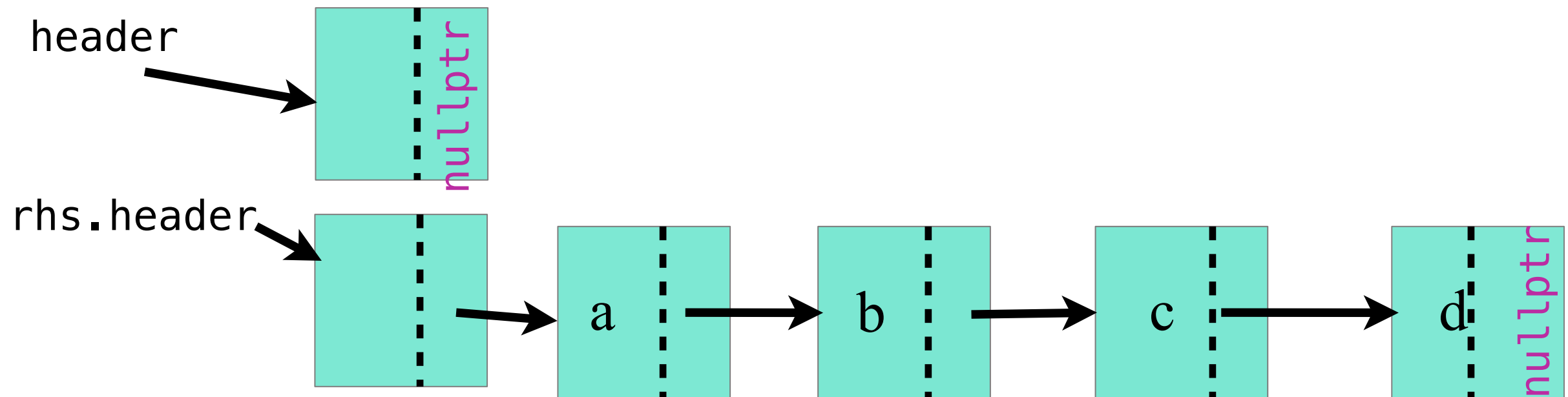
The big five!

copy constructor



```
// Copy constructor.  
List( const List & rhs )  
{  
    Homework!  
}
```

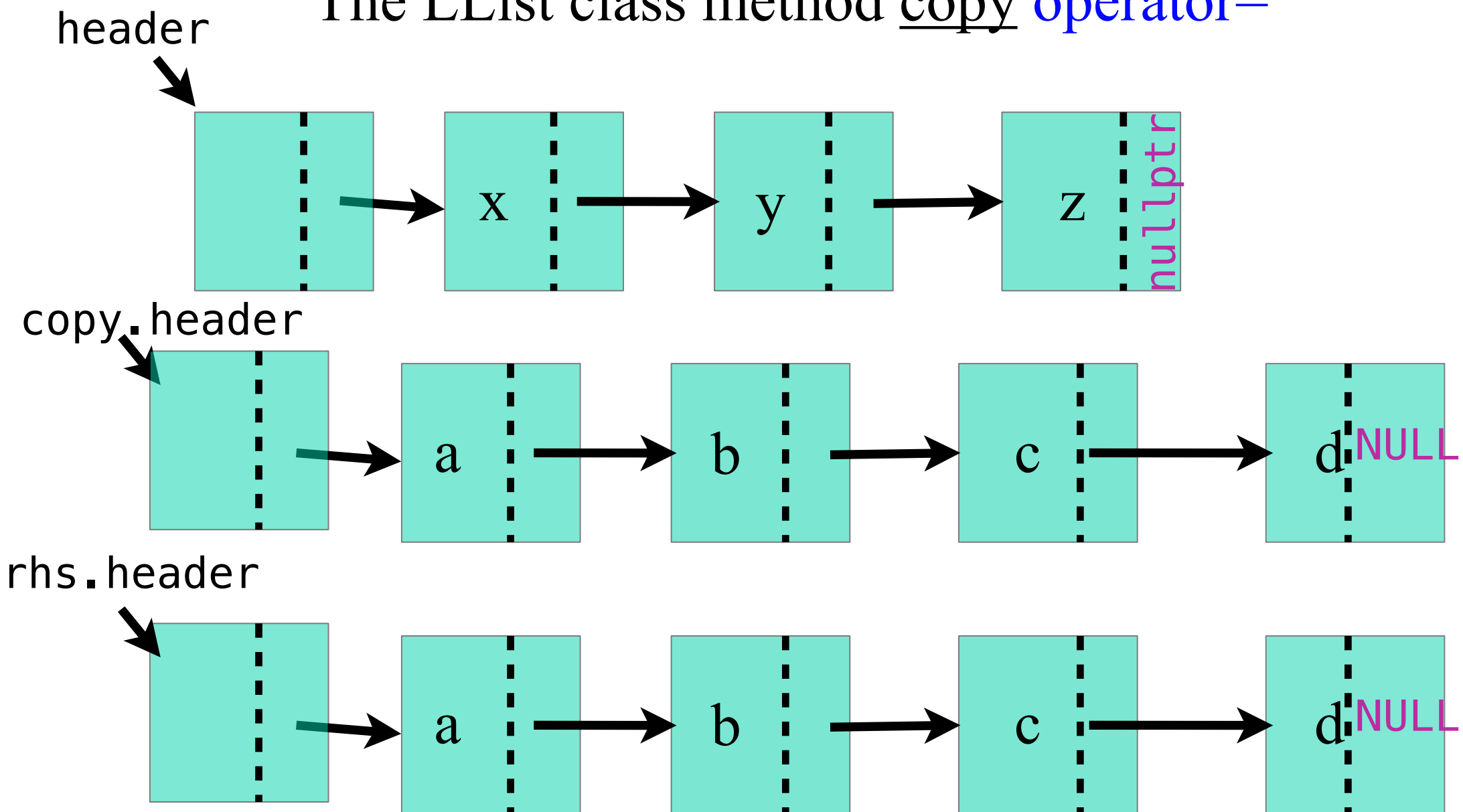

move constructor



// Move constructor.

```
List( List<Object> && rhs ):header(new Node)
{
    header->next = rhs.header->next;
    rhs.header->next = nullptr;
}
```

The LList class method copy operator=



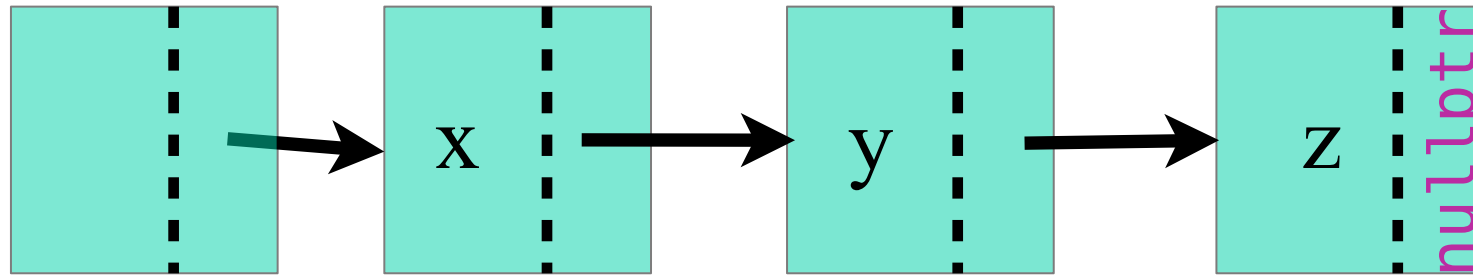
```
List & operator=(const List & rhs)
{
```

```
    List copy(rhs);
    std::swap( *this, copy );
    return *this;
```

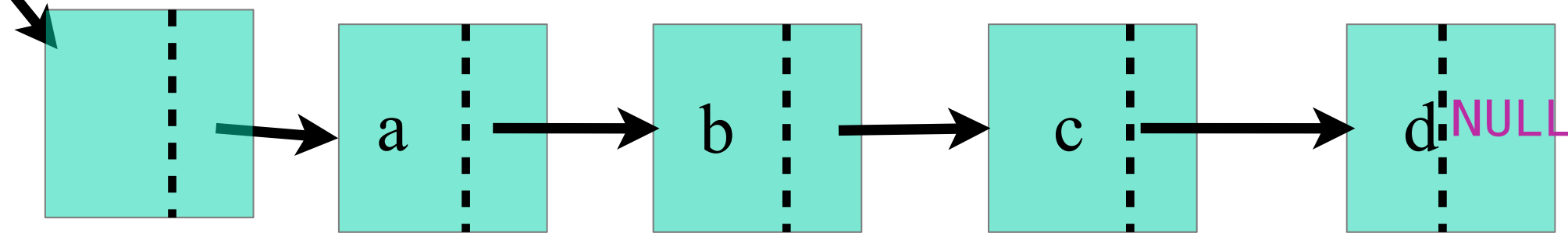
```
}
```

The LList class method move operator=

header



rhs.header



```
List & operator=( List && rhs)
```

```
{
```

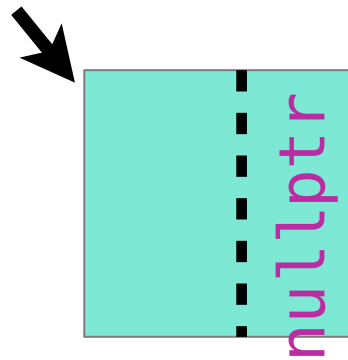
```
    std::swap( header, rhs.header);
```

```
    return *this;
```

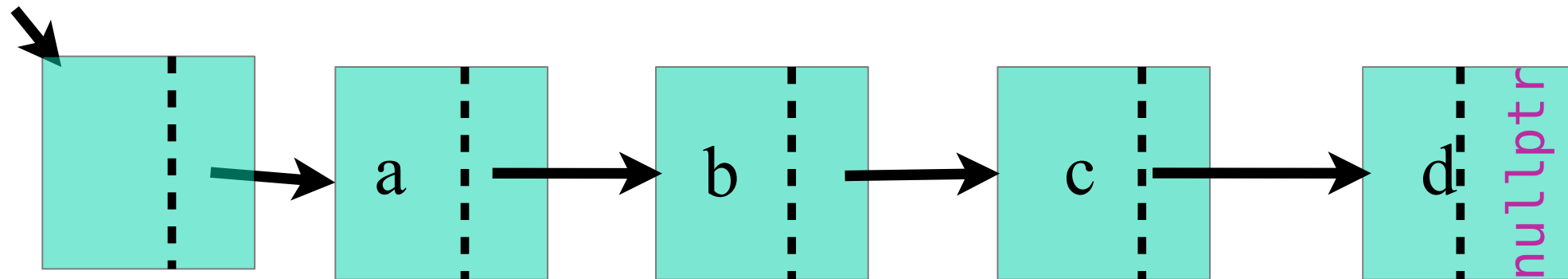
```
}
```

destructor: `~LinkedList()`

header



header



header



How much access should be allowed to the list class?

- Primitive strategy is to allow the user to have access to the list by a pointer.

What could go wrong?

- Another strategy is to have a member variable be a pointer to a node and control how it is used.

What could go wrong?

- Another strategy is to define a separate iterator class.

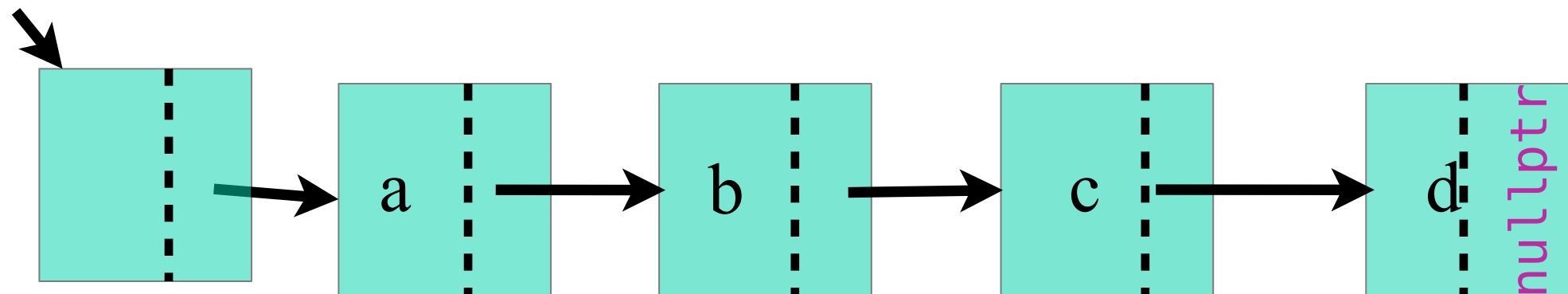
Advantages?

Writing a non member function that returns the number of items in a linked list.

```
template<class Object>
int listSize( const LList<Object> & theList)
{
    typename List<Object>::iterator itr;

    int size = 0;
    for ( itr = theList.begin(); itr != theList.end(); ++itr )
        size++;
    return size;
}
```

header



```

template <typename Object>
class List
{
    // Node class
public:
    class iterator
    {
    public:
        iterator( ) : current( nullptr ) { }
        Object & operator* ( ) { return current->data; }
        const Object & operator* ( ) const { return current->data; }

        iterator & operator++ ( )
        {
            current = current->next;
            return *this;
        }
        iterator operator++ ( int )
        {
            iterator old = *this;
            ++( *this );
            return old;
        }

        bool operator== ( const iterator & rhs ) const { return current == rhs.current; }
        bool operator!= ( const iterator & rhs ) const { return !( *this == rhs ); }
    private:
        Node * current;

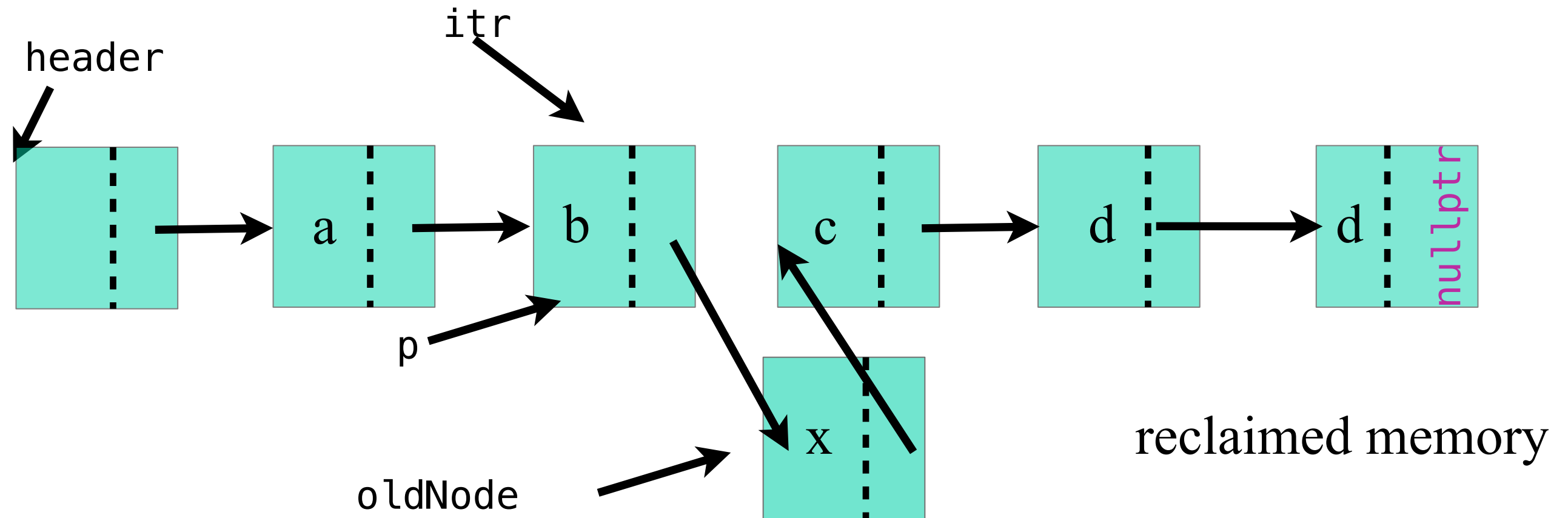
        iterator( Node *p ) : current{ p } { }
        friend class List<Object>;
    };
    // Other methods
private:
    LListNode<Object> *header;
};

```

zero parameter
constructor so
we can create a
vector of
iterators

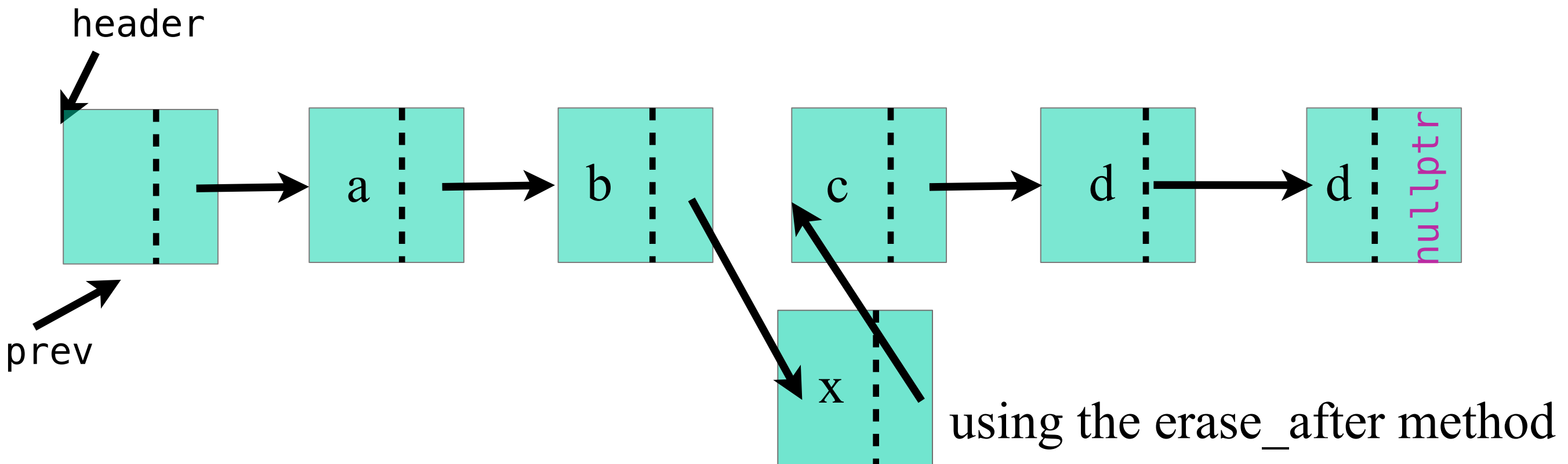
what we use in
our code!

iterator erase_after(iterator itr) method



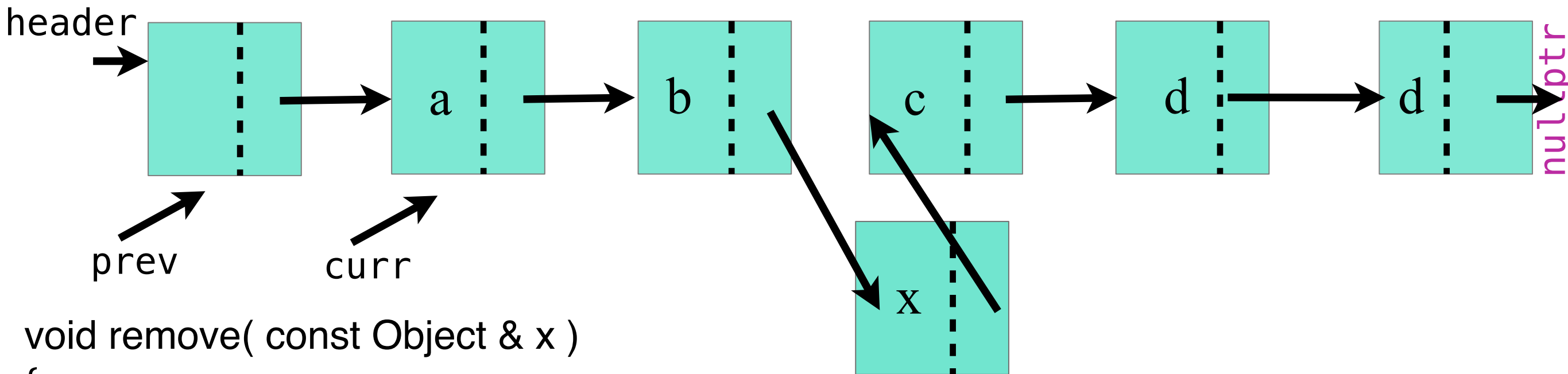
```
// Erase item after itr.  
iterator erase_after( iterator itr )  
{  
    Node *p = itr.current;  
    Node *oldNode = p->next;  
  
    p->next = oldNode->next;  
    delete oldNode;  
    oldNode = nullptr;  
  
    return iterator( p->next );  
}
```


void remove(const Object & x) method



```
void remove( const Object & x )  
{  
    Node * prev = header;  
  
    while ( prev->next != nullptr )  
    {  
        if ( prev->next->data == x )  
            erase_after( iterator(prev) );  
        else  
            prev = prev->next;  
    }  
}
```

another way to write the **remove** method



```
void remove( const Object & x )
{
```

```
    Node * prev = header;
```

```
    Node * curr = header->next;
```

```
    while ( curr != nullptr )
```

```
    {
```

```
        if ( curr->data == x )
```

```
        {
```

ah that is better

```
            curr = erase_after( iterator( prev ) ).current;
```

```
        }
```

```
    else
```

```
    {
```

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

```
        curr = curr->next;
```

```
    }
```

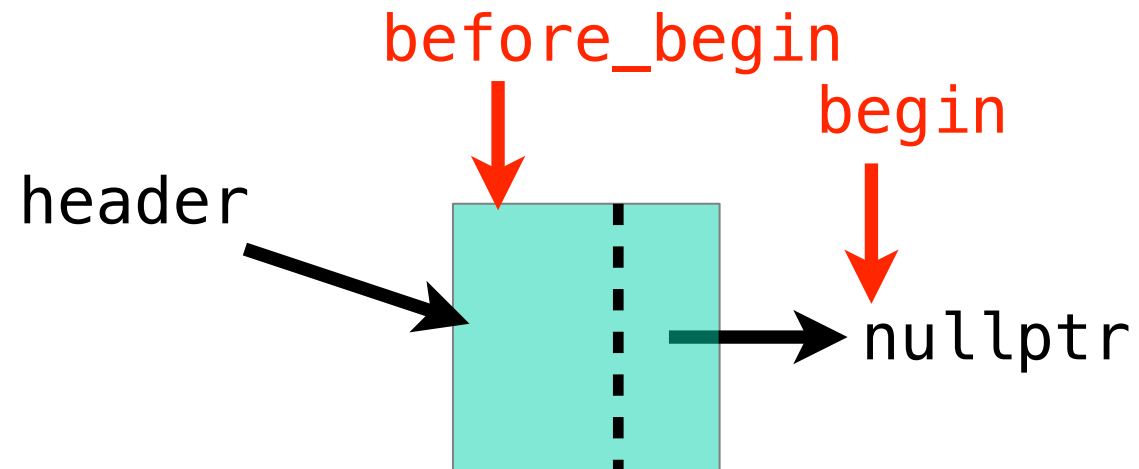
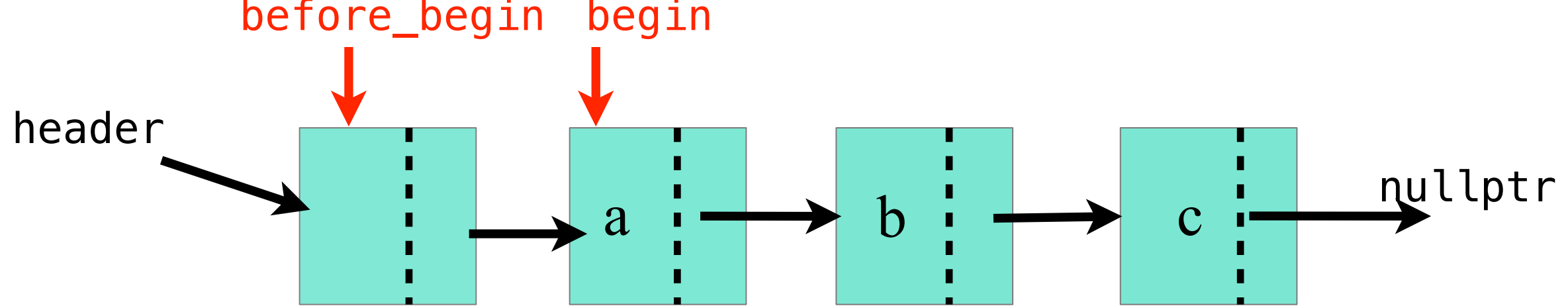
```
} }
```

using the `erase_after` method

//or

```
// erase_after( iterator(prev) );
```

```
// curr = prev->next;
```



// Return an iterator representing the header node.

iterator **before_begin()** const

```
{
```

```
    return iterator( header );
```

```
}
```

// Return an iterator representing the first node in the list.

iterator **begin()** const

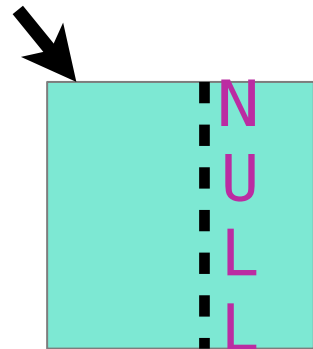
```
{
```

```
    return iterator( header->next );
```

```
}
```

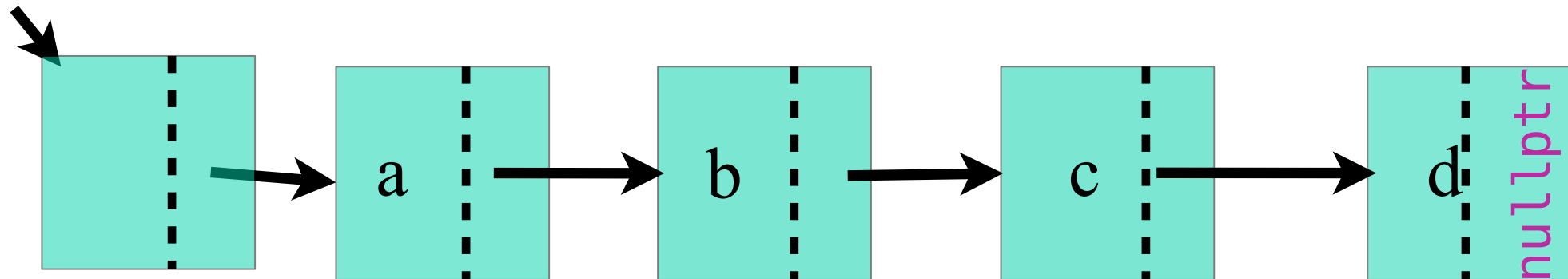
method `empty()`

header



true!

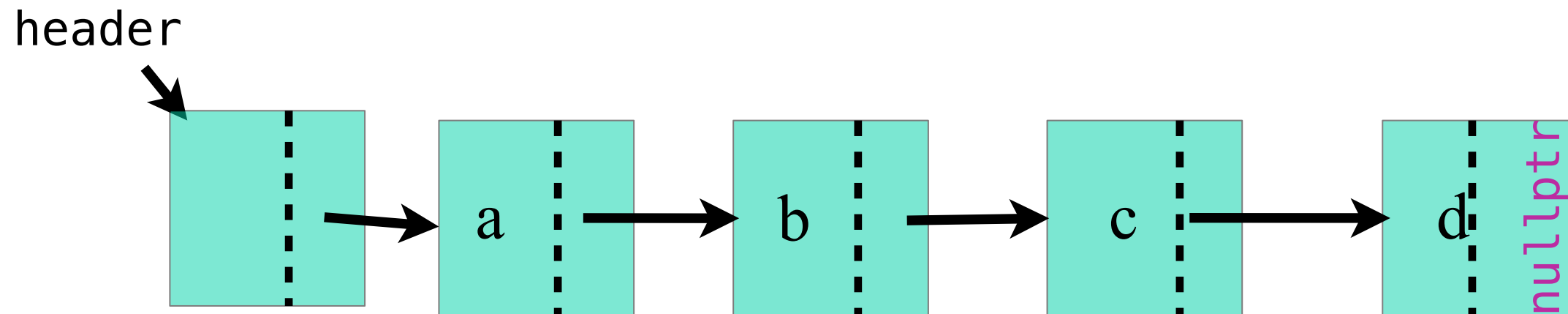
header



false

```
// Test if the list is logically empty.  
bool empty( ) const  
{  
    return header->next == nullptr;  
}
```

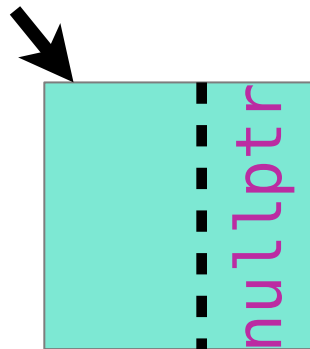
method `pop_front()`



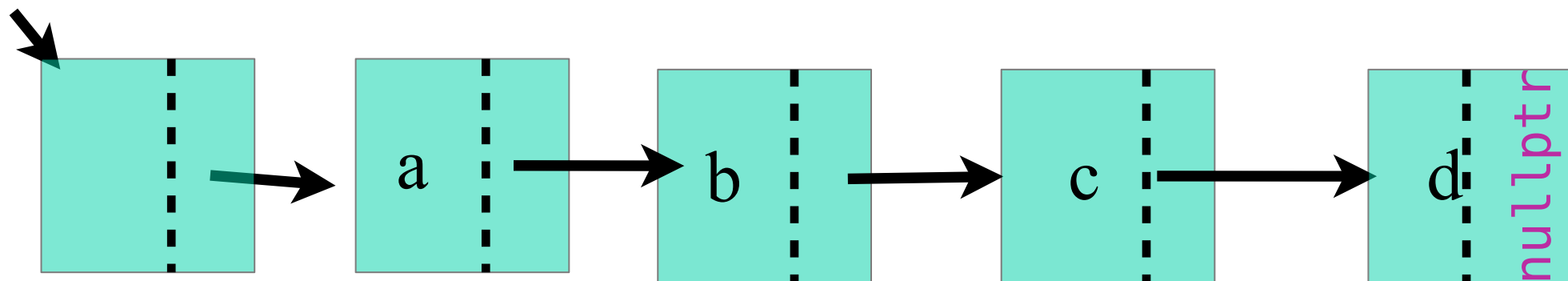
```
void pop_front( )  
{  
    erase_after( before_begin( ) );  
}
```

method `clear()`, making the list logically empty

header



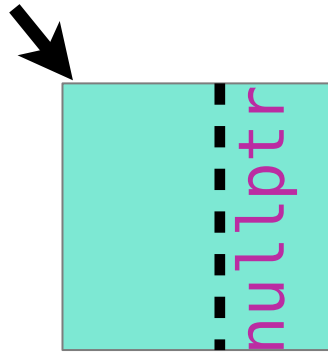
header



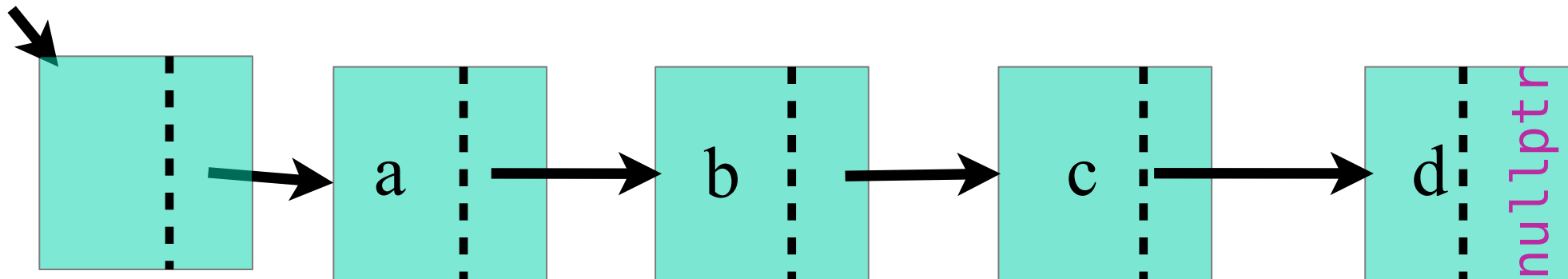
```
void clear( )  
{  
    while( !empty( ) )  
        pop_front( );  
}
```

destructor: `~LList()`

header



header



// Destructor.

`~LList()`

{

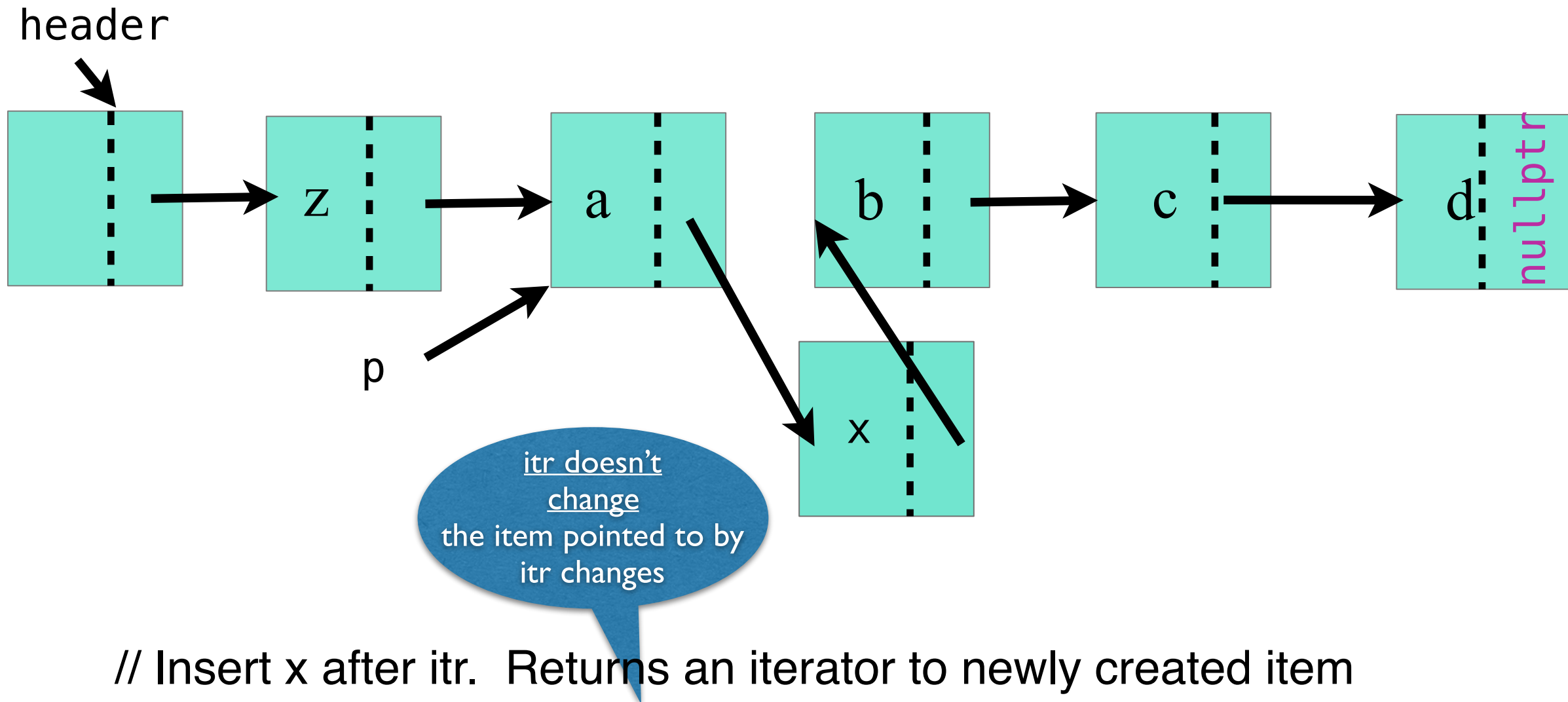
clear();

delete header;

}

Insertion

iterator insert_after(iterator itr, const Object & x);



// Insert x after itr. Returns an iterator to newly created item
iterator insert_after(iterator itr, const Object & x)

{

Node *p = itr.current;
p->next = new Node{ x, p->next };

return iterator(p->next);

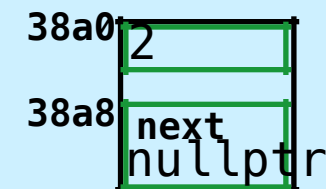
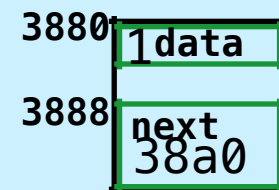
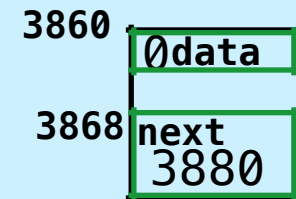
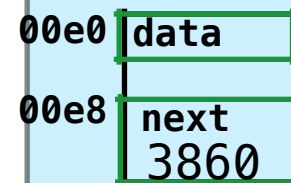
}

Using the class

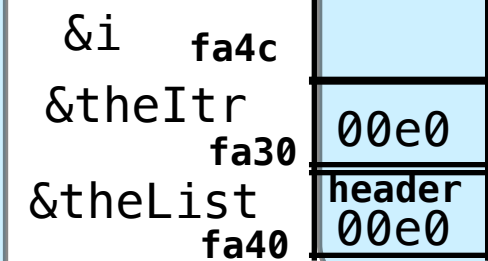
Using the LList class

```
int main( ) {  
    List<int> theList;  
    List<int>::iterator theItr = theList.begin();  
  
    int i;  
  
    for( i = 0; i < 3; i++ )  
    {  
        theList.insert_after( theItr, i );  
        ++theItr;  
    }  
  
    for( i = 0; i < 3; i ++ )  
        theList.remove( i );  
  
    return 0;  
}
```

heap



stack



Our List - Forward Iterator

- `sl.empty()` $O(1)$
- `s.clear()` $O(n)$
- `sl.remove(x)` $O(n)$
- `sl.insert_after(itr, x)` $O(1)$
- `sl.begin()` $O(1)$
- `sl.before_begin()` $O(1)$
- `sl.end()` $O(1)$
- `sl.erase_after(itr)` $O(1)$
- `sl_lhs = sl_rhs` $O(n)$

Note: all these times do not include constructor/destructor times which many vary according to the type

List Traversals

- Lots of applications require iteration through list, accessing each element (until end or until some condition is met)
- Sometimes useful to keep additional iterators, such as “prev” (sometimes called “trailer”) that stays one step behind “current” iterator or pointer.

Example

finding the first negative element in the list

```
List<int> L; // ... assume L has some elements
```

```
List<int>::iterator curr = L.begin();
```

```
while ( curr != L.end() && ( *curr >= 0 ) )  
    ++curr;
```

```
if ( curr != L.end() )  
    cout << "First negative element is " << *curr;  
else  
    cout << "All elements are positive."
```

Example 2

Removing the first negative element from the list

```
LList<int> L;
```

```
...
```

```
List<int>::iterator curr = L.begin();
```

```
List<int>::iterator prev = L.before_begin();
```

```
while ( curr != L.end() && ( *curr >= 0) )
```

```
{
```

```
    prev = curr;
```

```
    ++curr;
```

```
}
```

```
if ( curr != L.end() )
```

```
    L.erase_after(prev);
```

STL Lists

- <http://www.sgi.com/tech/stl/List.html>
- Doubly linked list with
 - Forward & reverse iterators
 - $O(1)$ insertion and deletion at front, back and arbitrary points
 - Iterators not invalidated on insertion, splicing or removal (except removed node)
 - ...
- STL also provides singly linked list: `forward_list`

list - Bidirectional Iterators



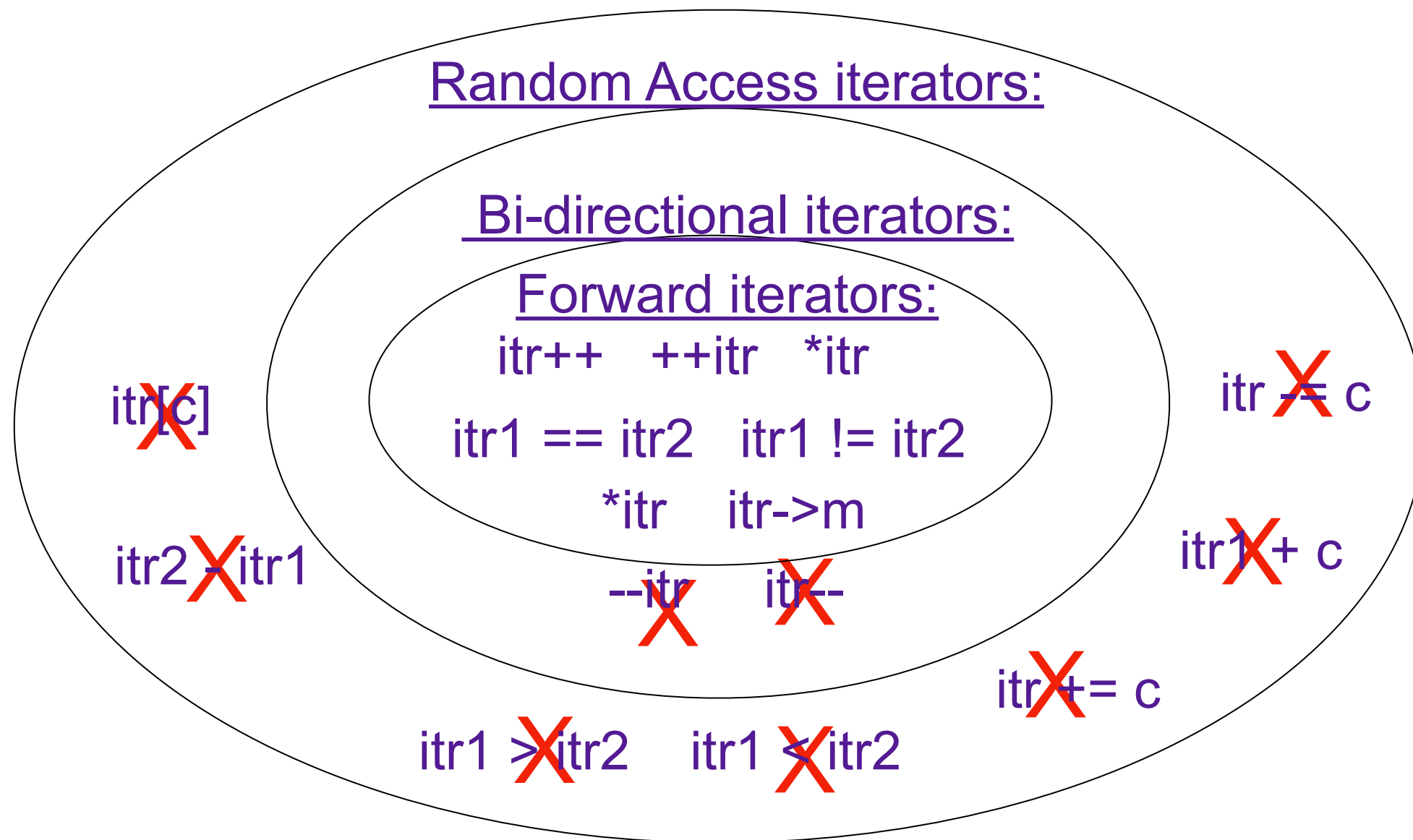
```
#include<list>
```

- `l.push_back(value)` $O(1)$
- `l.push_front(value)` $O(1)$
- `l.erase(v.begin(),v.end())` $O(n)$
- `l.erase(iterator)` $O(1)$
- `l.clear()` $O(n)$
- `l.size()` $O(1)$
- `l.insert(iterator,value)` //inserts before iterator $O(1)$
- `l.begin()` $O(1)$
- `l.end()` $O(1)$
- `l.sort()` & `l.sort(comparator)` $O(n\log(n))$

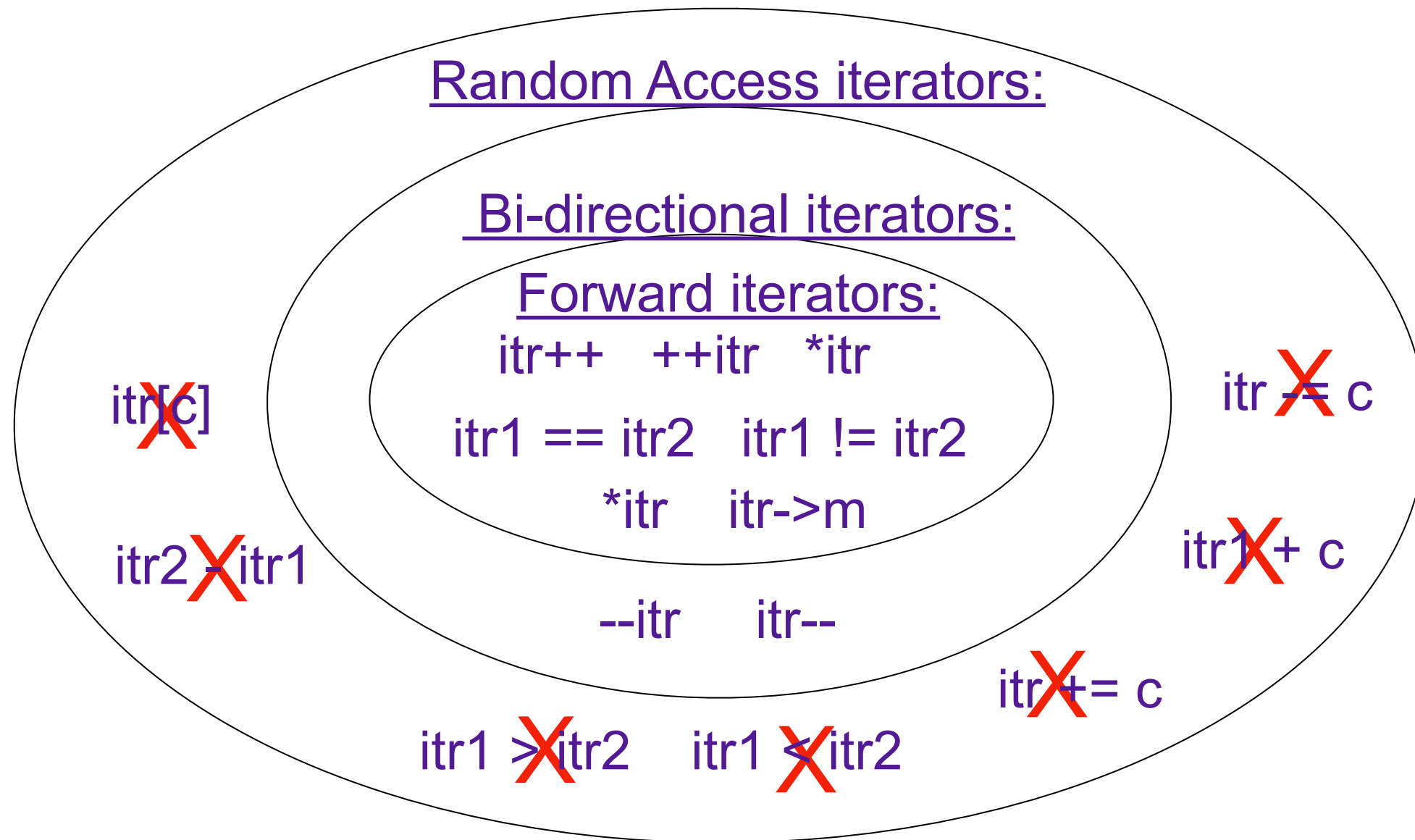
Note: all these times do not include constructor/destructor times which many vary according to the type

Iterators

What type of iterator operations for a singly linked list?



What type of iterator operations for a doubly linked list?



Instantiating an Iterator

Random Access Iterators

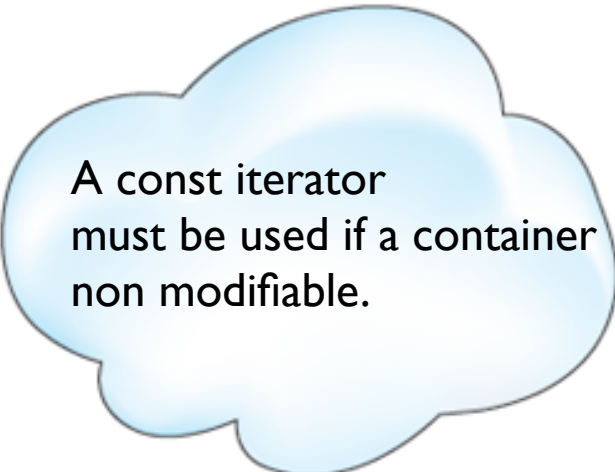
`vector<T>::iterator vecItr;` `vector<T>::const_iterator constVecItr;`

Bidirectional Iterators

`list<T>::iterator listItr;` `list<T>::const_iterator const_listItr;`
`map<K, V>::iterator mapItr;` `map<K, V>::const_iterator const_mapItr;`
`set<K, V>::iterator setItr;` `set<K, V>::const_iterator const_setItr;`

Forward Iterators

`forward_list<T>::iterator slistItr;` `forward_list<T>::const_iterator const_sListItr;`



A const iterator
must be used if a container
non modifiable.