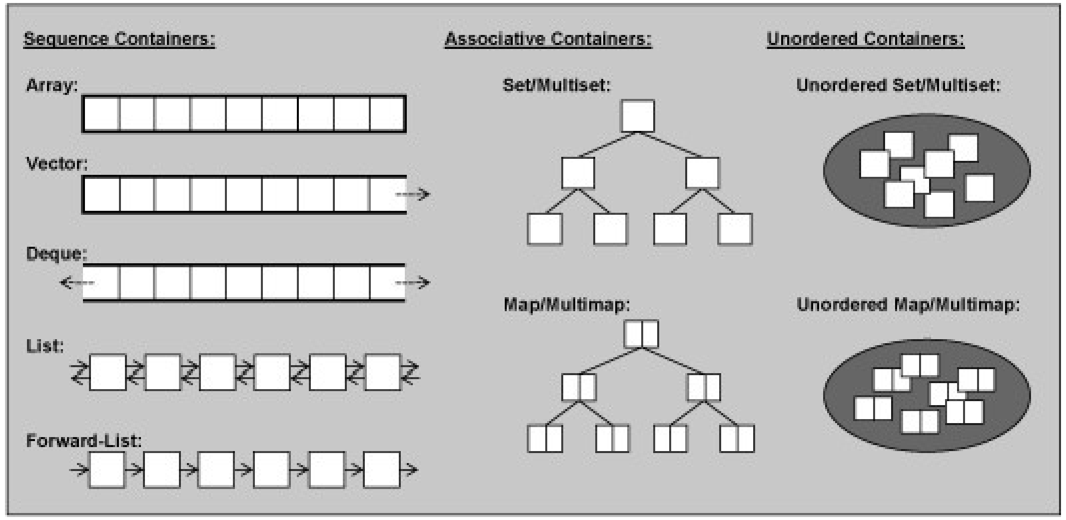
CPSC 131 – Data Structures

Singly and Doubly Linked List Abstract Data Types



Professor T. L. Bettens
Fall 2020

Concepts & Interface

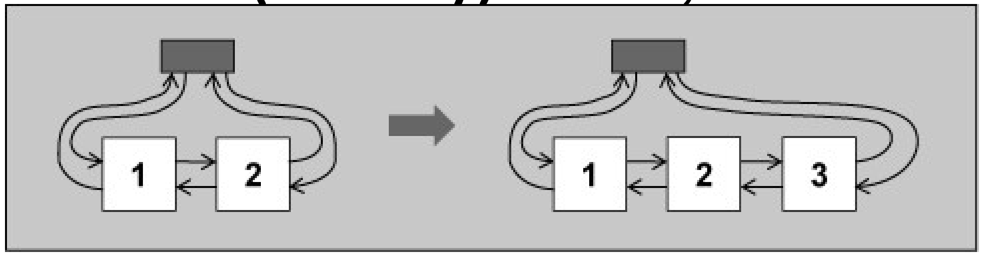
- Jusuttis,
  The C++ Standard Library
  - 6.2. Containers
  - 7.1. Common Container Abilities and Operations
  - 7.6. Forward (Singly Linked) Lists
    - 7.6.4. Examples of Using Forward Lists
  - 7.5. (Doubly Linked) Lists
    - 7.5.4. Examples of Using Lists

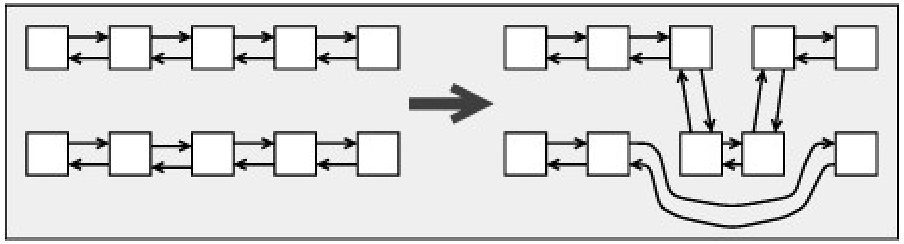
- CPPReference.com
  - Containers library
  - std::forward_list
  - std::list

- zyBook
  - tbd

List Abstract Data Type
Common Implementation choices

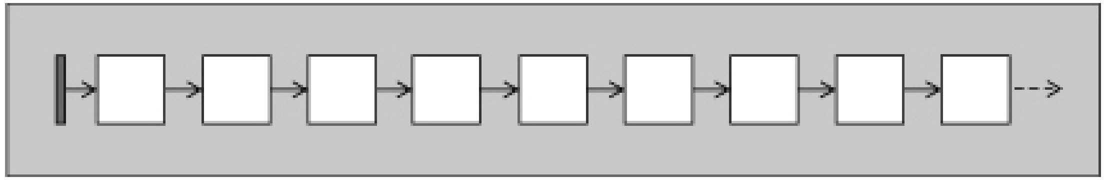
Sentinel (Dummy) Nodes, or not





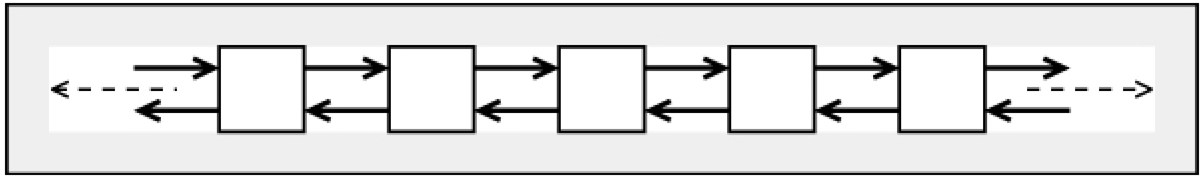
Singly Linked List

- Can move only forward
- Can insert and delete the “next” element, never the “current” element

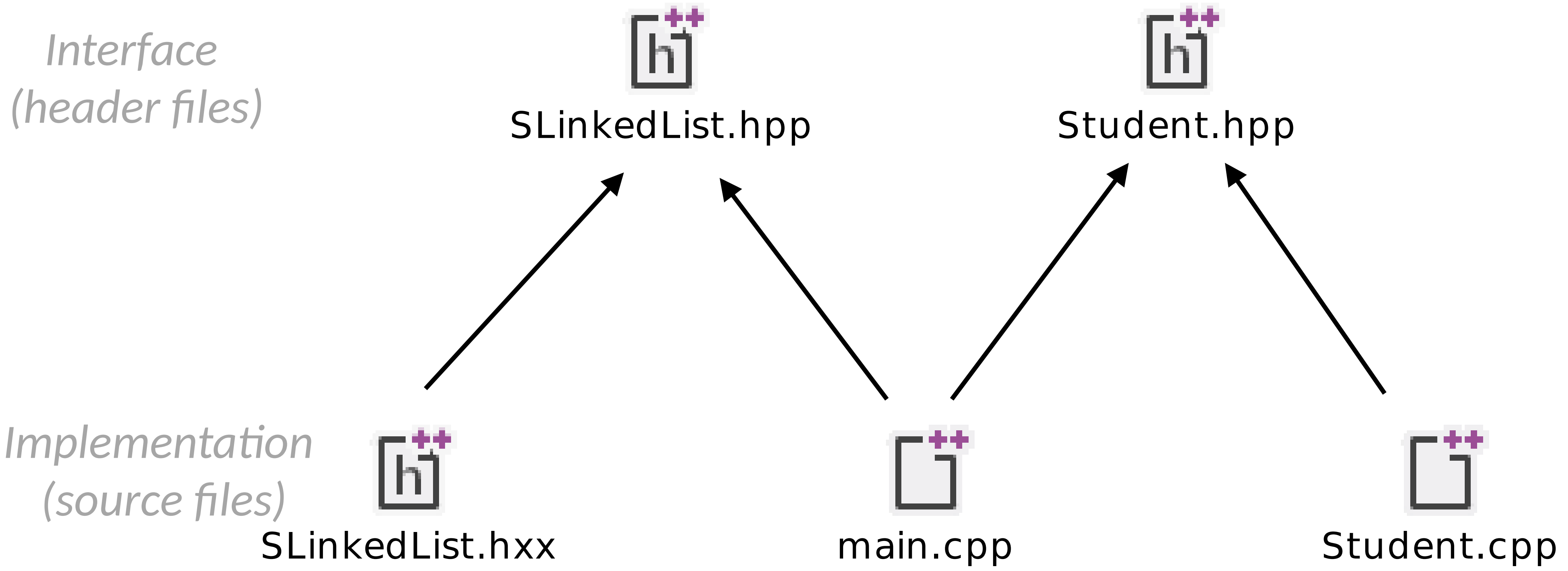


Doubly Linked List

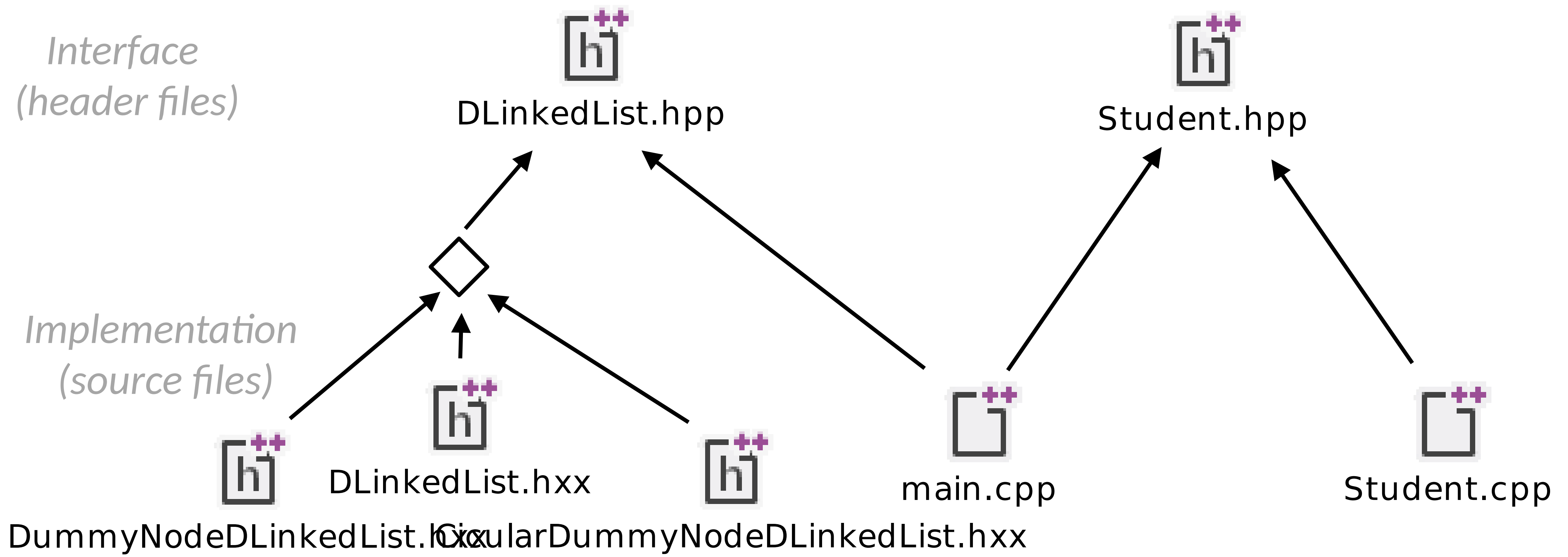
- Can move in both directions
- Can delete “current” element
- Can insert before or after “current”



Singly Linked List Implementation Example



Doubly Linked List Implementation Example



Analysis of the Vector Abstract Data Type
Complexity Analysis (1)

Function	Analysis – std::forward_list<T> (Singly Linked List)	Analysis – std::list<T> (Doubly Linked List)
at()	Not available	Not available
size()	O(n) std::forward_list<T> calculates size on demand	same
empty()	O(1)	same
clear()	O(n) All elements are destroyed and size set to zero	same

Analysis of the Vector Abstract Data Type
Complexity Analysis (2)

Function	Analysis – std::forward_list<T> (Singly Linked List)	Analysis – std::list<T> (Doubly Linked List)
push_back()	Not available std::forward_list<T> has no tail pointer	O(1) has tail pointer
erase()	O(1) assumes you have erasure point Can erase only after erasure point	O(1) assumes you have erasure point Can erase only before erasure point
splice	Not available std::forward_list<T> has no tail pointer	O(1)

Analysis of the Vector Abstract Data Type
Complexity Analysis (3)

Function	Analysis – std::forward_list<T> (Singly Linked List)	Analysis – std::list<T> (Doubly Linked List)
insert()	O(1) assumes you have insertion point Can insert only <u>after</u> insertion point	O(1) assumes you have insertion point Can insert only <u>before</u> insertion point
default construction	O(1) creates an empty container	same
Equality $C_1 == C_2$	O(n)	same

Analysis of the Vector Abstract Data Type
Complexity Analysis (4)

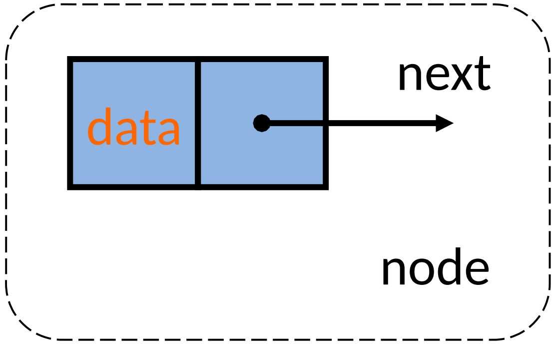
Function	**Analysis – std::forward_list<T>** **(Singly Linked List)**	**Analysis – std::list<T>** **(Doubly Linked List)**
push_front	O(1)	same
resize	O(n)	same
find	O(n) linear search from begin() to end() *(i.e. head to tail)*	same

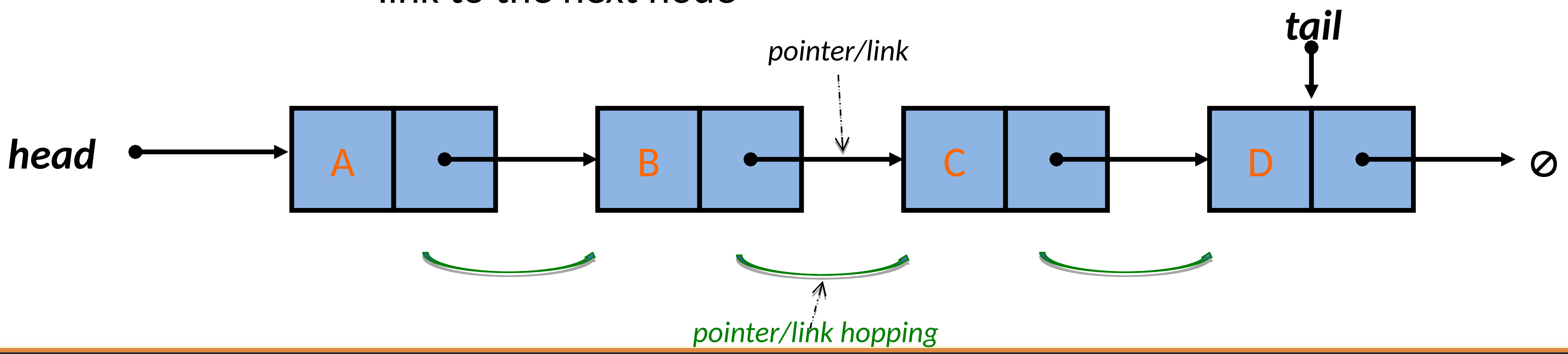
Analysis of the Vector Abstract Data Type
Complexity Analysis (5)

Function	Analysis – std::forward_list<T> (Singly Linked List)	Analysis – std::list<T> (Doubly Linked List)
Visit every element **e.g. print()**	O(n) Visiting every node from begin() to end()	same
Visit in reverse **e.g. print_reverse()**	not possible	O(n) Visiting every node from rbegin() to rend() Direction doesn't matter

Singly Linked Lists

- A singly linked list is a concrete **data structure** consisting of a sequence of nodes
- Each node stores
  - Data element
  - link to the next node



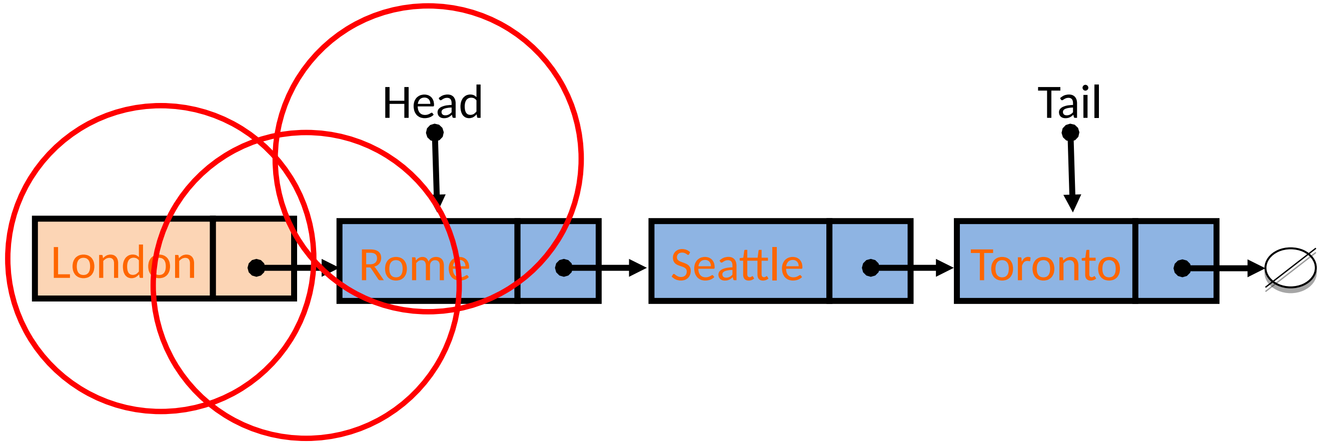


Inserting at the Head

1. Allocate and populate a new node

2. Have new node point to old head

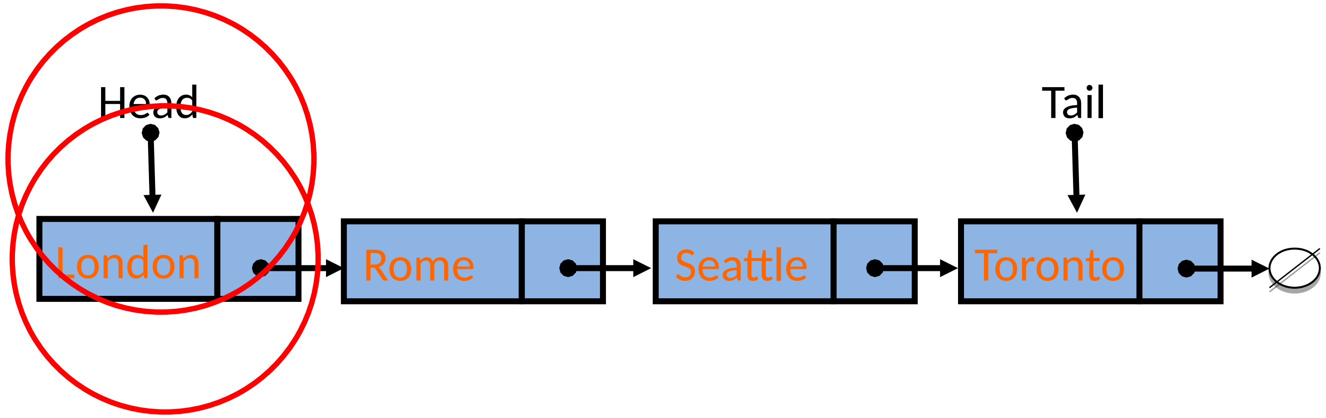
3. Update head to point to new node



Deleting at the Head

1. Update head to point to the next node in the list

2. Delete the former first node



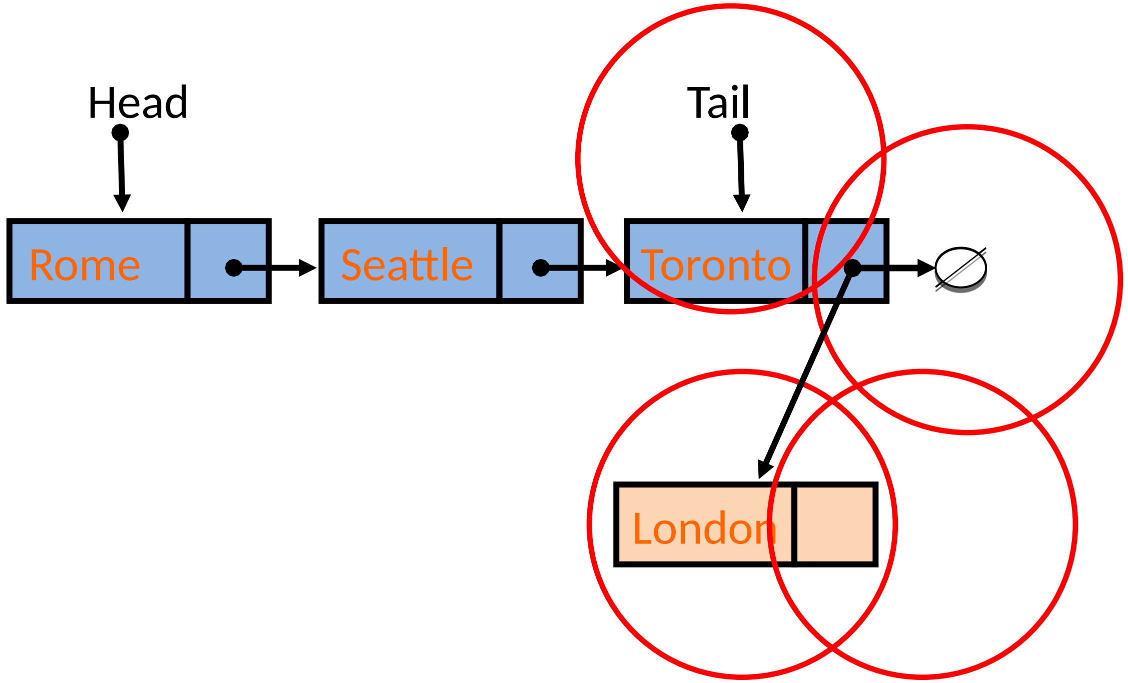
Inserting at the Tail

1. Allocate and populate a new node

2. Have new node point to whatever the tail node pointed to, namely nullptr

3. Point the old Tail node to the new node

4. Update tail to point to new node

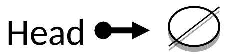


Draw data structure for this code

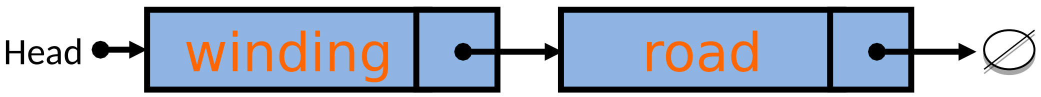
prepend == push_front

removeFront == pop_front

```
SLinkedList<string> ds;
cout << ds.size();
```




```
ds.prepend("road");
ds.prepend("winding");
cout << ds.front();
```

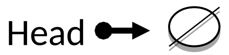



```
ds.prepend("and");
ds.removeFront();
ds.prepend("long");
cout << ds.front();
```

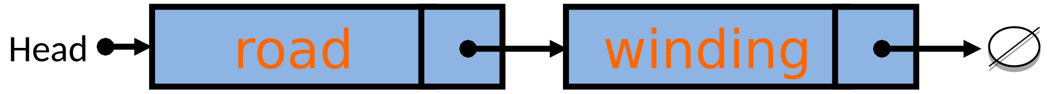
Draw data structure for this code

append == push_back
removeFront == pop_front

```
SLinkedList<string> ds;
cout << ds.size();
```




```
ds.append("road");
ds.append("winding");
cout << ds.front();
```




```
ds.append("and");
ds.removeFront();
ds.append("long");
cout << ds.front();
```

Nodes

To create a linked list using dynamic storage, we need a class which has two data members:

- one to hold information
- one to point to another object of the *same* class

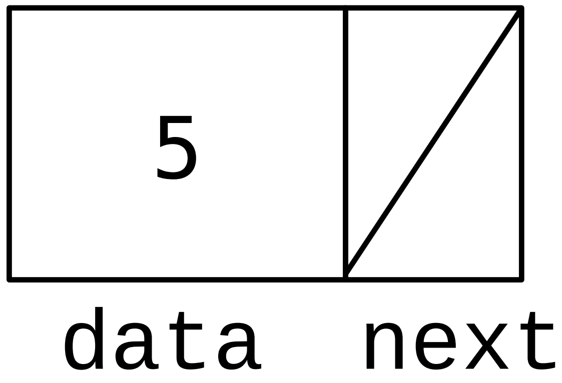
Singly Linked List Node

```
template <typename Data_t>
struct SLinkedList<Data_t>::Node // singly linked list node
{
  Node() = delete;                  // Intentionally no default constructor
  Node( const Data_t & element ) : data( element ) {}

  Data_t data;                      // linked list element value
  Node * next = nullptr;            // next item in the list
};
```

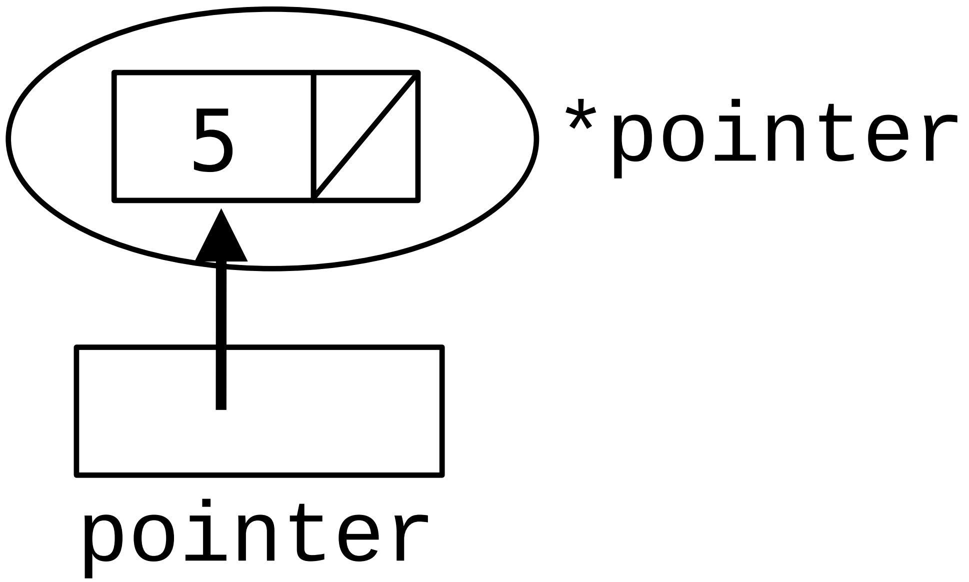
Picture of a Node

```
SLinkedList<int>::Node node(5);
node.next = nullptr;
```



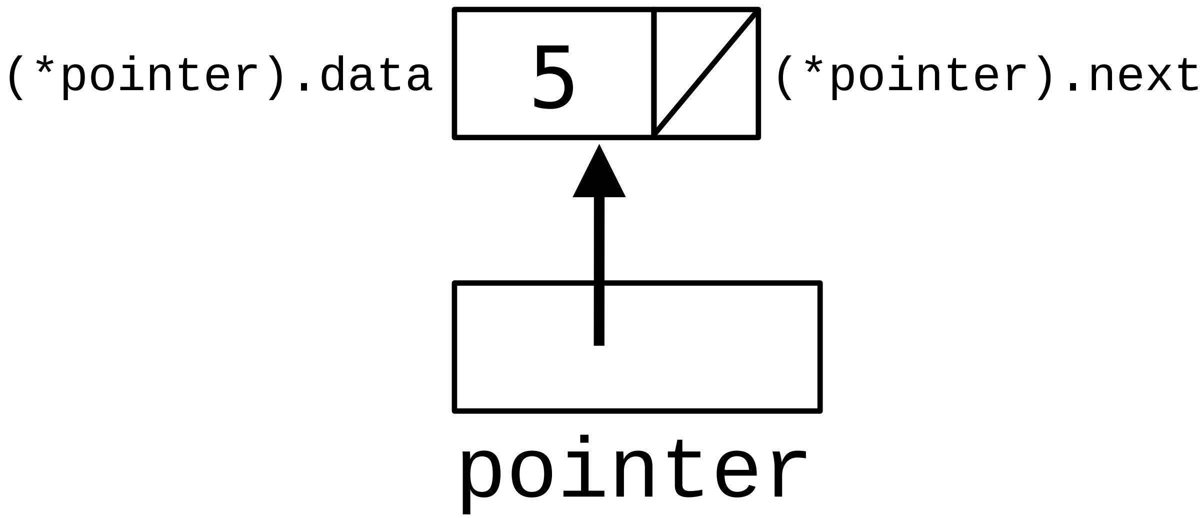
Creating a node as a dynamic variable

```
SLinkedList<int>::Node * pointer;
pointer = new SLinkedList<int>::Node(5)
```



Accessing the fields of the node

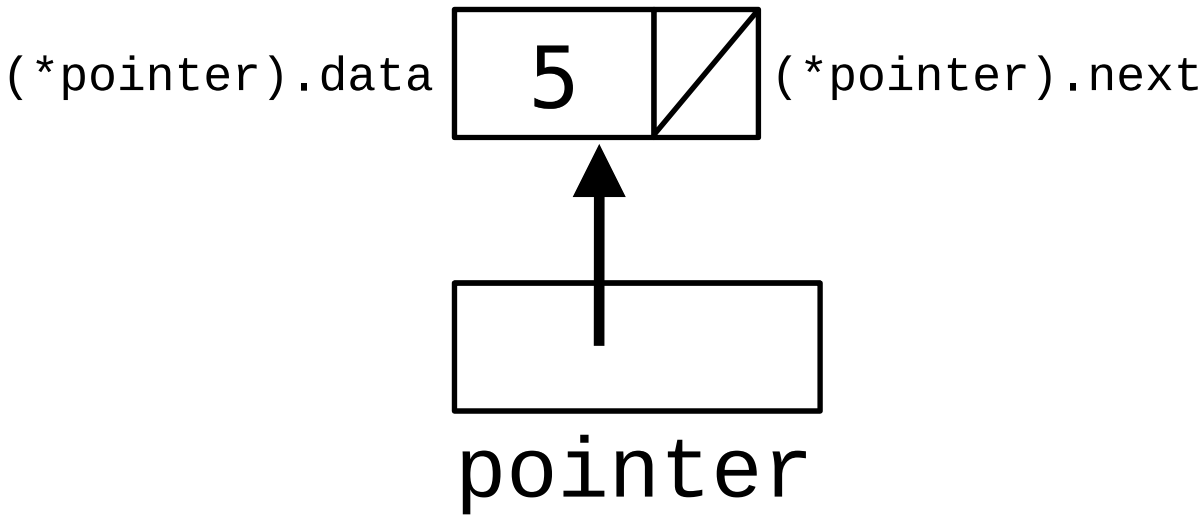
```
(*pointer).data
(*pointer).next
```



Accessing the fields of the node

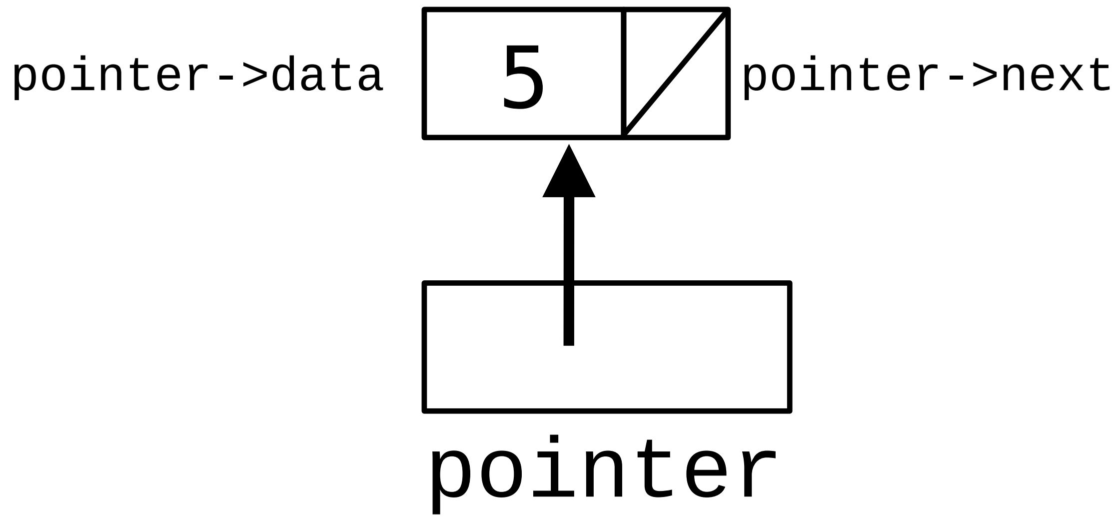
`(*pointer).data` can also be written as `pointer->data`

`(*pointer).next` can also be written as `pointer->next`



Accessing the fields of the node

```
pointer->data == 5
pointer->next == nullptr
```

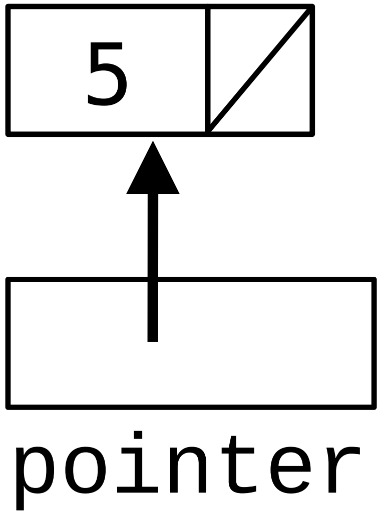


Review: delete

What does

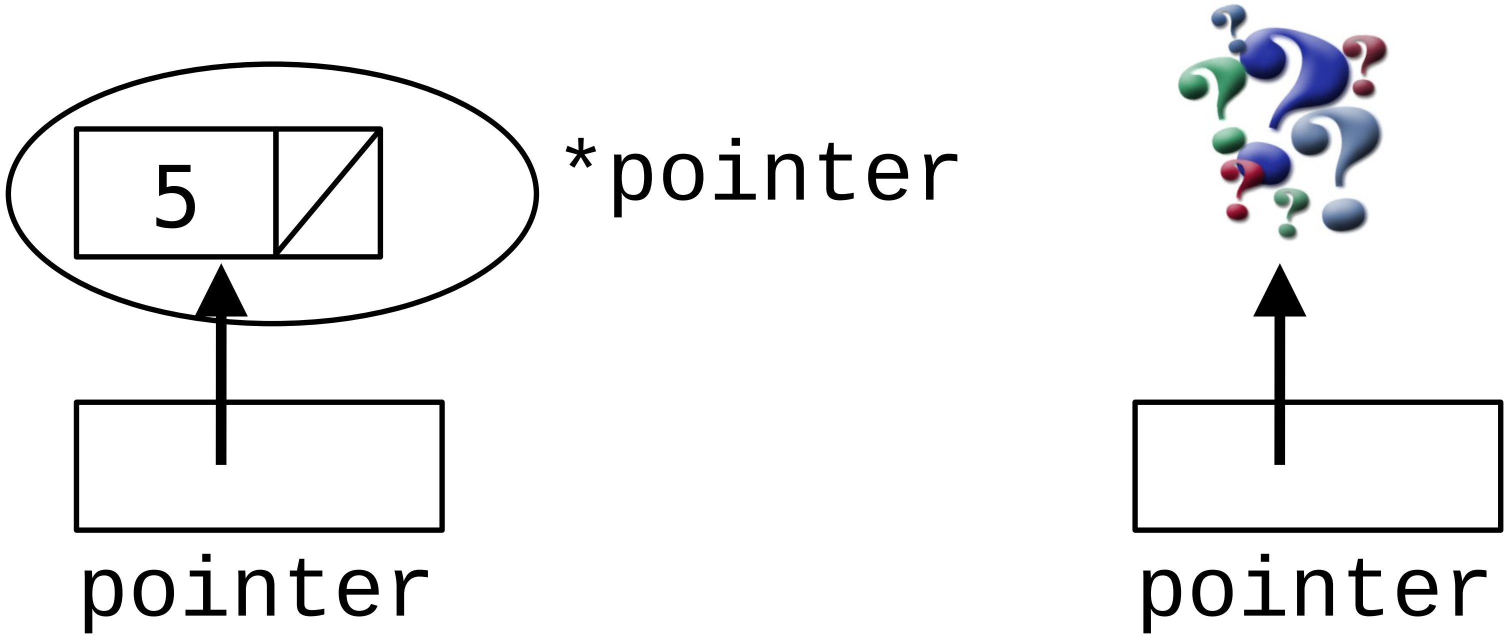
```
delete pointer;
```

do?



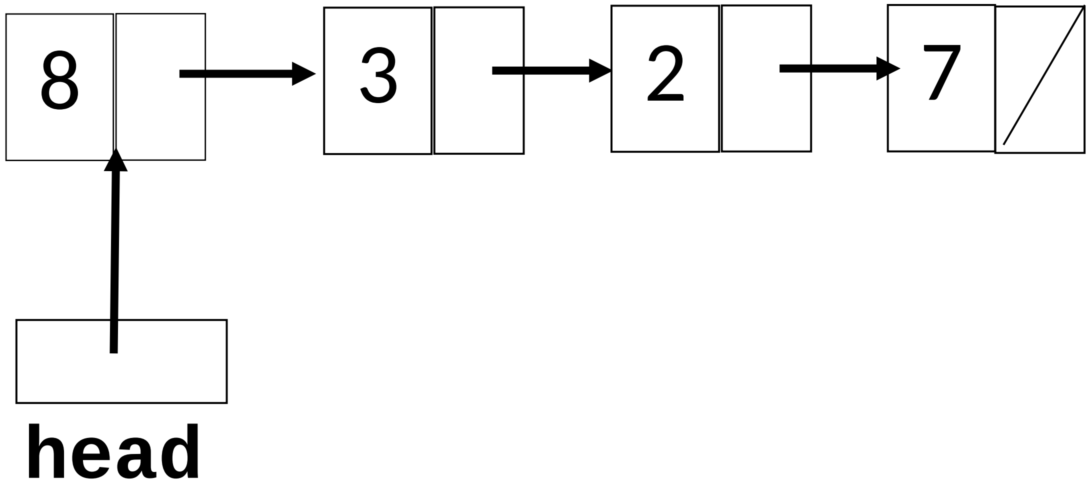
Answer

It deletes the node, but leaves the pointer dangling.



What about a linked list?

Since the `next` field can point to another node, we can link nodes together like this:



Creating a linked list

Start out with two NULL pointers to NodeType.

Code for this??

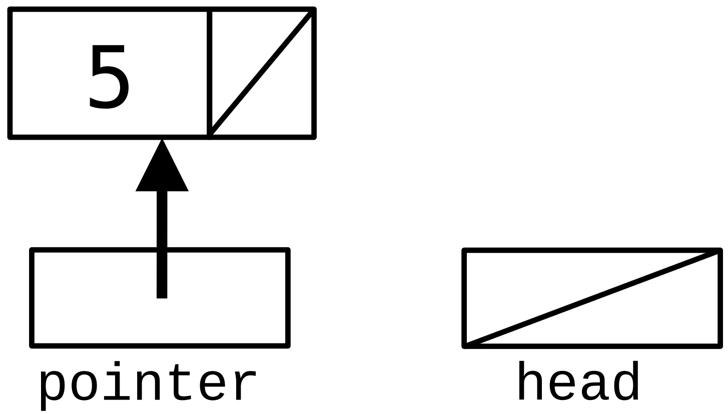
```
SLinkedList<int>::Node *pointer = nullptr;
SLinkedList<int>::Node *head    = nullptr;
```

Creating a linked list

Now create a new SinglyLinkedNode using `pointer`.

Code for this??

```
pointer = new SLinkedList<int>::Node(5);
```

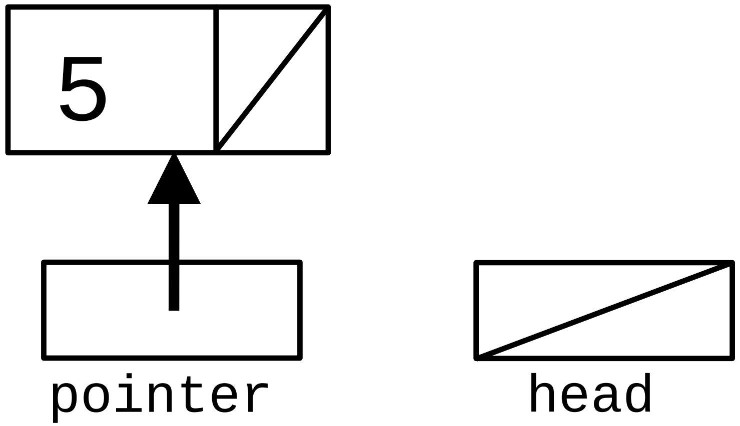


Creating a linked list

Now what happens if we do

```
pointer->next = head;
```

`head` contains `nullptr`, which gets copied to `pointer->next`

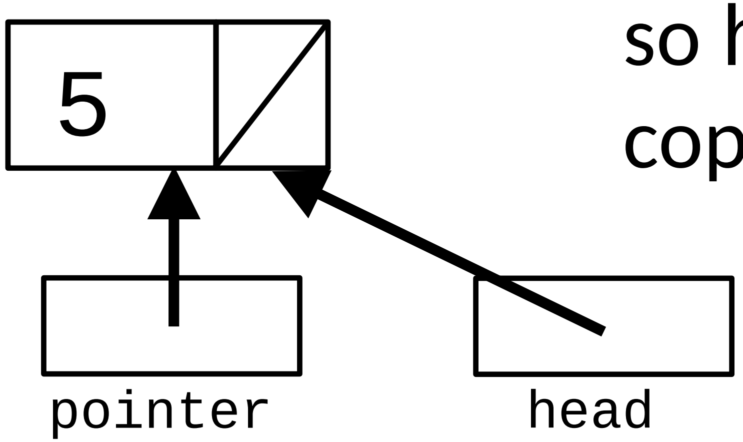


Creating a linked list

Now we want head to point to the new node,

i.e. head should contain the address of the new node.

What already has that address? **`pointer`**



so `head = pointer` will copy the address to `head`.

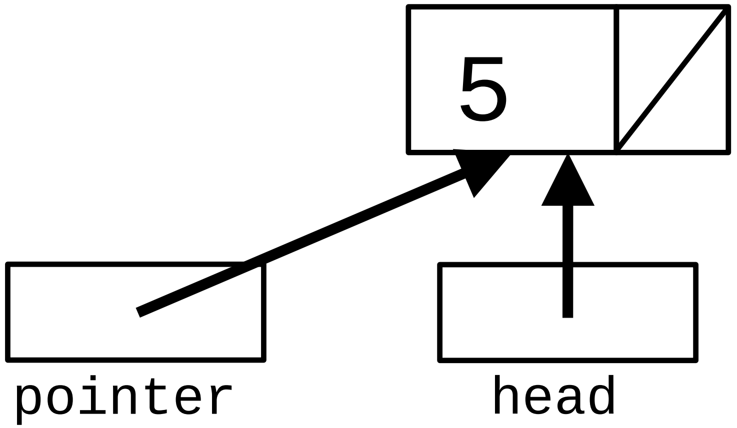
Creating a linked list

Putting the code together:

```
pointer       = new SLinkedList<int>::Node(5);
pointer->next = head;
head          = pointer;
```

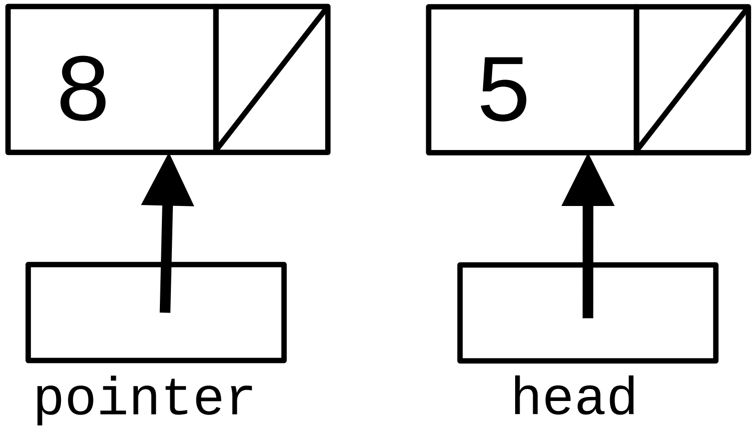
Going from code to picture

Now try seeing what happens if we repeat the same code with `data` now set to 8, starting with this picture.



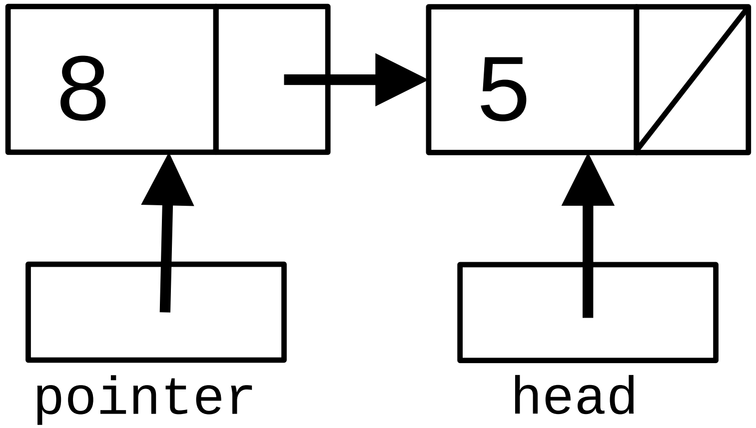
Going from code to picture

```
pointer       = new SLinkedList<int>::Node(8);
pointer->next = head;
head          = pointer;
```



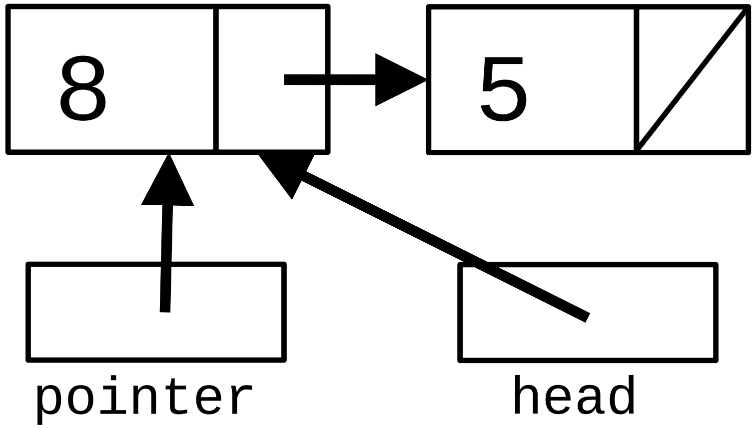
Going from code to picture

```
pointer       = new SLinkedList<int>::Node(8);
pointer->next = head;
head          = pointer;
```



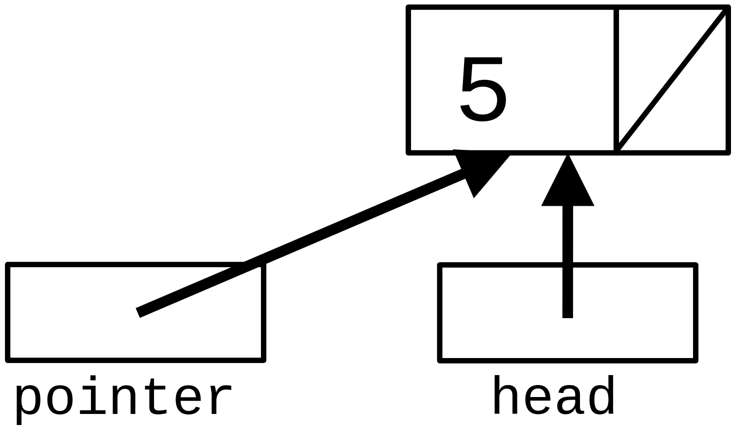
Going from code to picture

```
pointer       = new SLinkedList<int>::Node(8);
pointer->next = head;
head          = pointer;
```



So the following code will go from

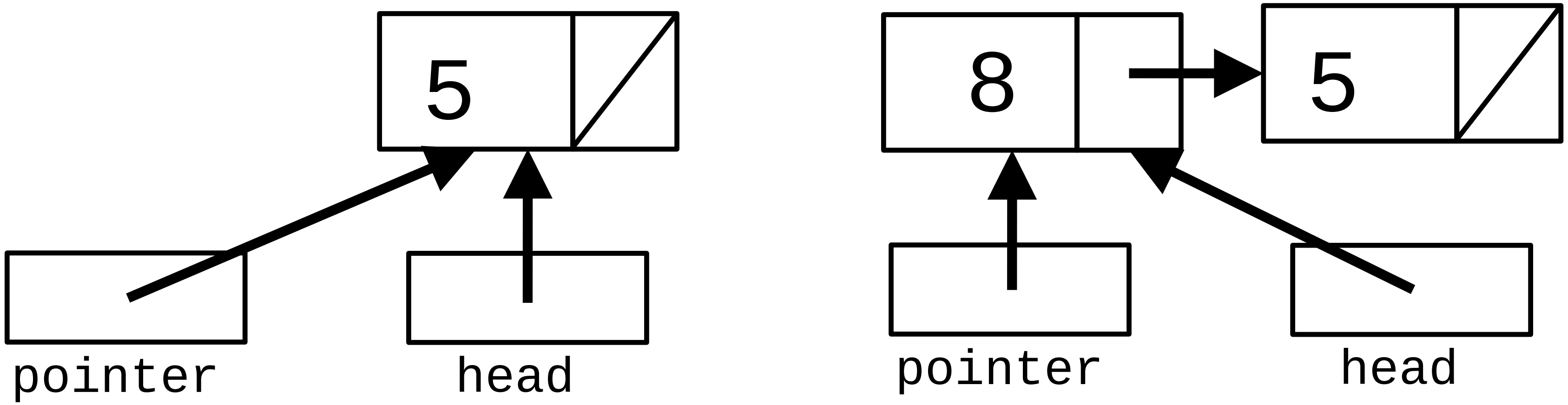
```
pointer       = new SLinkedList<int>::Node(8);
pointer->next = head;
head          = pointer;
```



to this, i.e. insert a node at the front

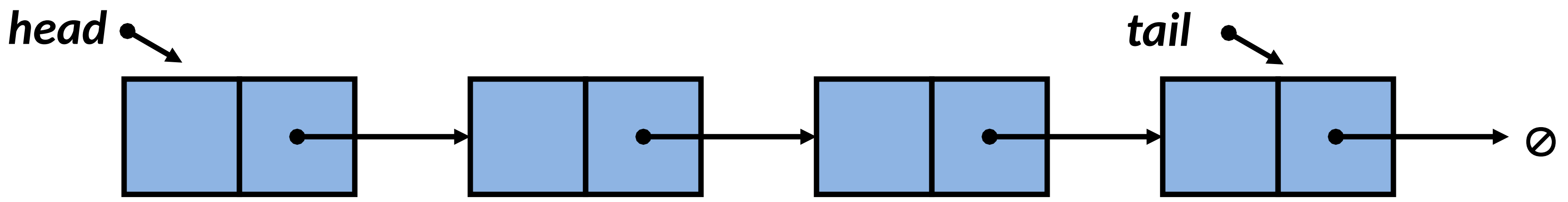
```
void
prepend( const Data_t & element );
```

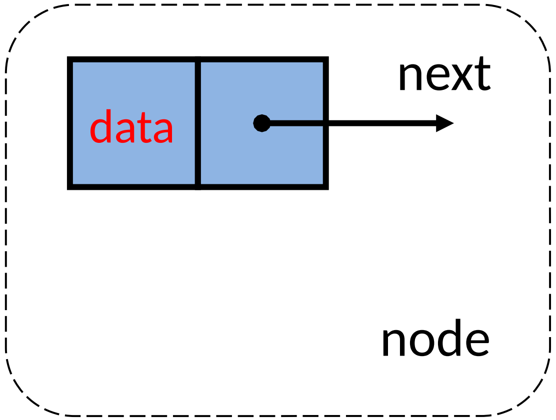
```
pointer       = new SLinkedList<int>::Node(8);
pointer->next = head;
head          = pointer;
```



Singly Linked List and Doubly Linked List

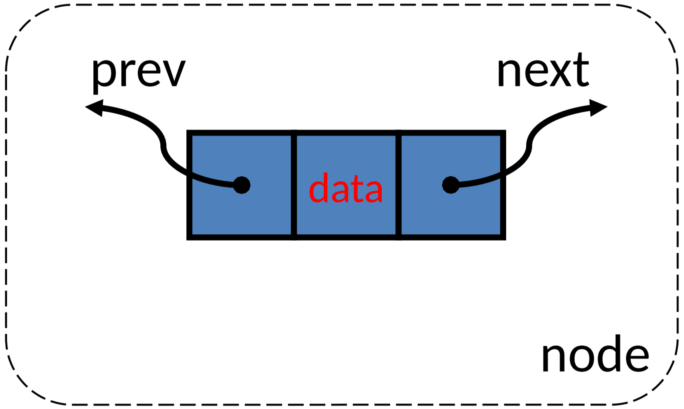
What if want to access data in reverse order?





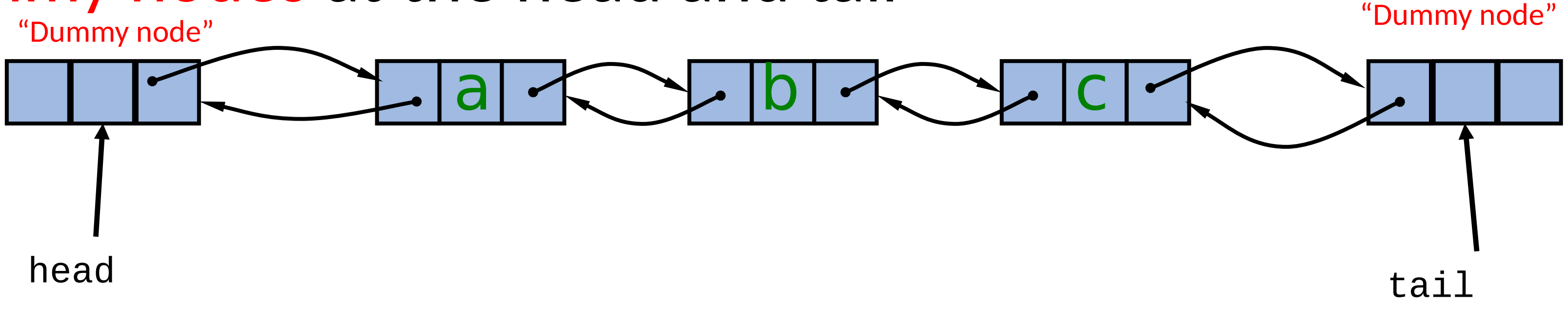
<u>Node</u>:
- element
- next pointer
- previous pointer

<u>Linked List</u>:
- length
- Head
- Tail

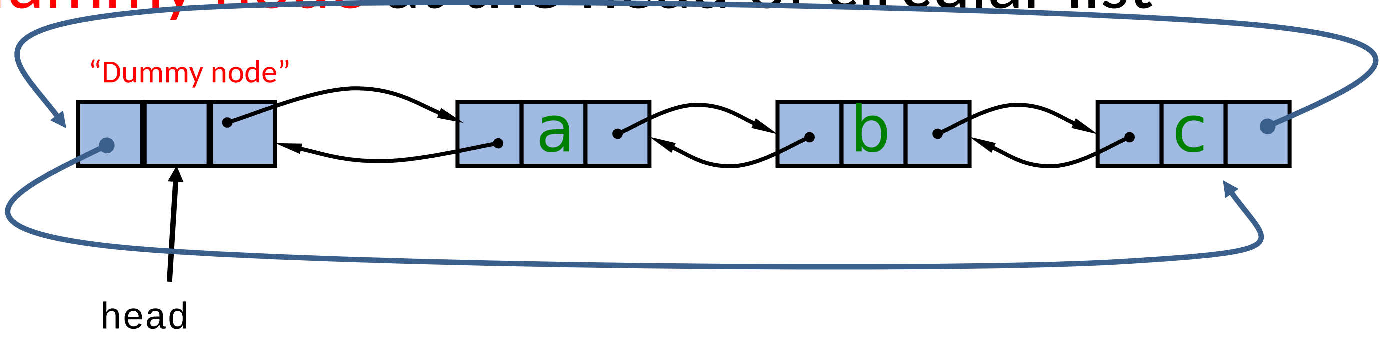


Doubly linked lists

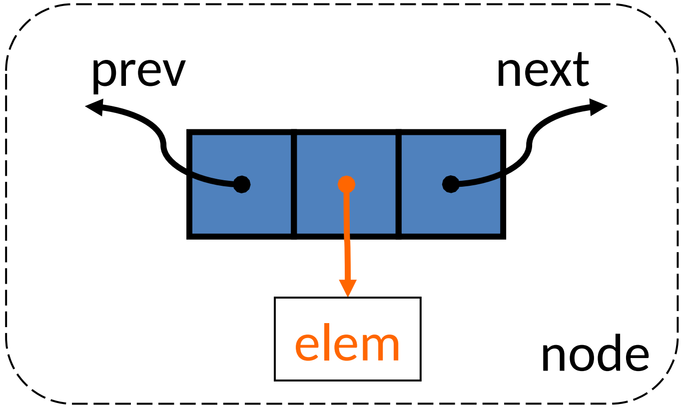
- Key ideas
  - Keep a previous pointer *in addition to a next pointer* at every node
  - Keep dummy nodes at the head and tail



  - Keep a dummy node at the head of circular list



Doubly Linked List Node




```
template <typename E>
struct DNode                             // doubly linked list node
{
   DNode (const E & element) : elem(element);

   E         elem;                       // node element value
   DNode<E> *prev = nullptr;             // previous node in the list
   DNode<E> *next = nullptr;             // next node in the list
};
```

Doubly Linked List constructor

- Create sentinels and interlink them

```
template <typename E>
DoublyLinkedList<E>:: DoublyLinkedList<E() { // constructor
   head = new Node<E>; // create dummy nodes
   tail = new Node<E>;
   // have them point to each other
   head->next = tail;
   tail->prev = head;
}
```

- Empty linked list: only has two dummy nodes