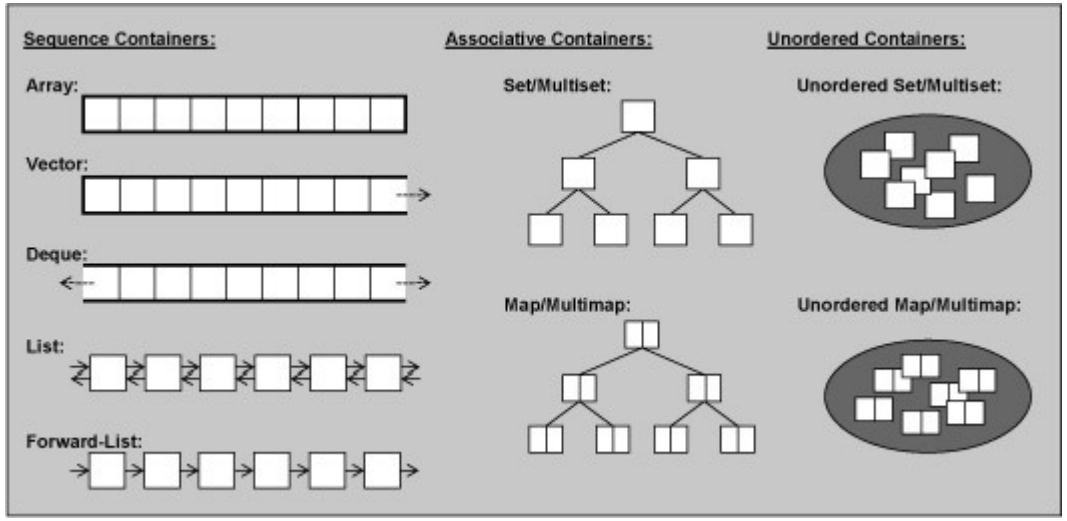
CALIFORNIA STATE UNIVERSITY FULLERTON

CPSC 131 – Data Structures

Singly and Doubly Linked List Abstract Data Types



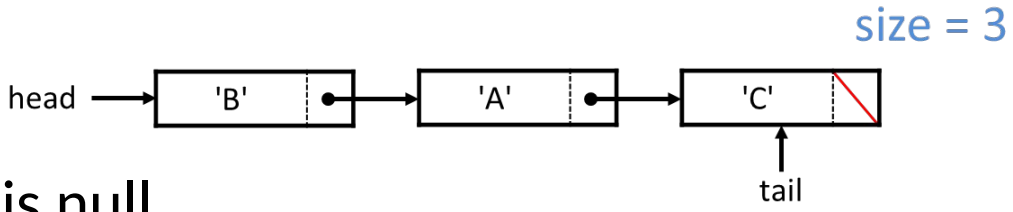
Professor T. L. Bettens

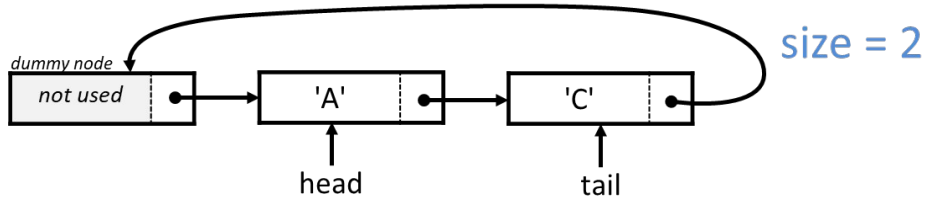
Spring 2023

Linked List Abstract Data Type

Definitions:

Node	- an object containing data and link(s) to adjacent node(s)
Head	- the first node in the list
Tail	- the last node in the list
Null-terminated	- a list whose link in the last node is null
Circular	- a list whose link in the last node is the first node

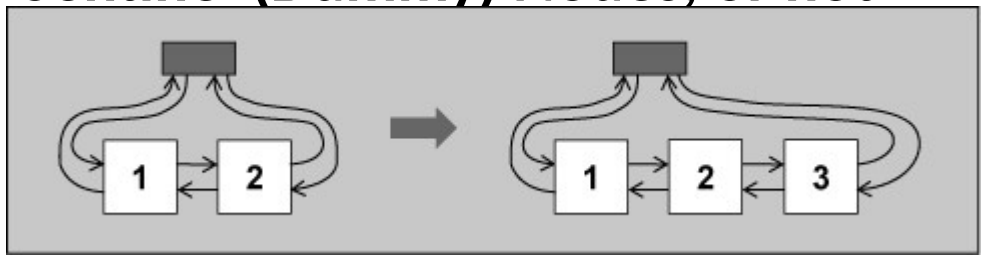


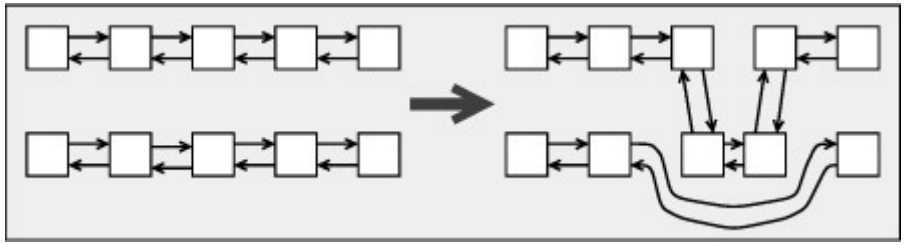


CPSC 131 T. L. Bettens

Linked List Abstract Data Type
Two flavors, Singly and Doubly Linked

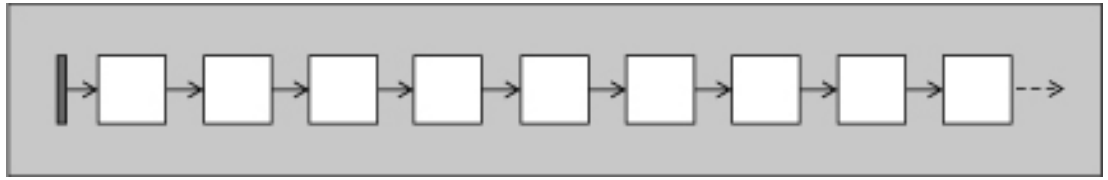
Sentinel (Dummy) Nodes, or not





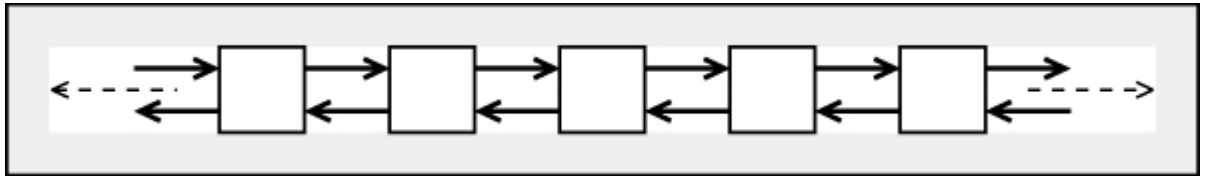
Singly Linked List

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



Doubly Linked List

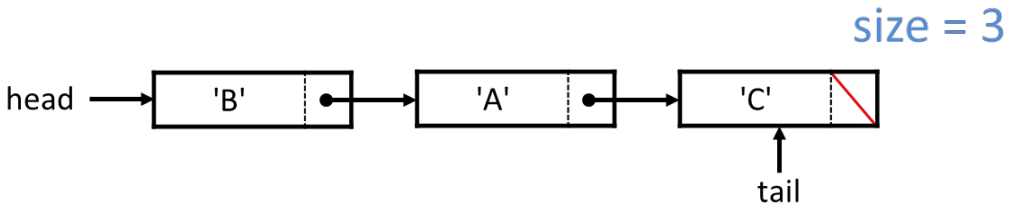
- Can move in both directions
- Can delete “current” element
- Can insert before or after “current”

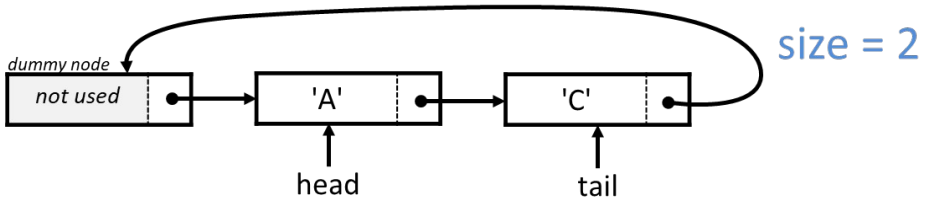


SinglyLinkedList
DoublyLinkedList

Linked List Abstract Data Type
The Abstraction - What can I do to a Linked List?

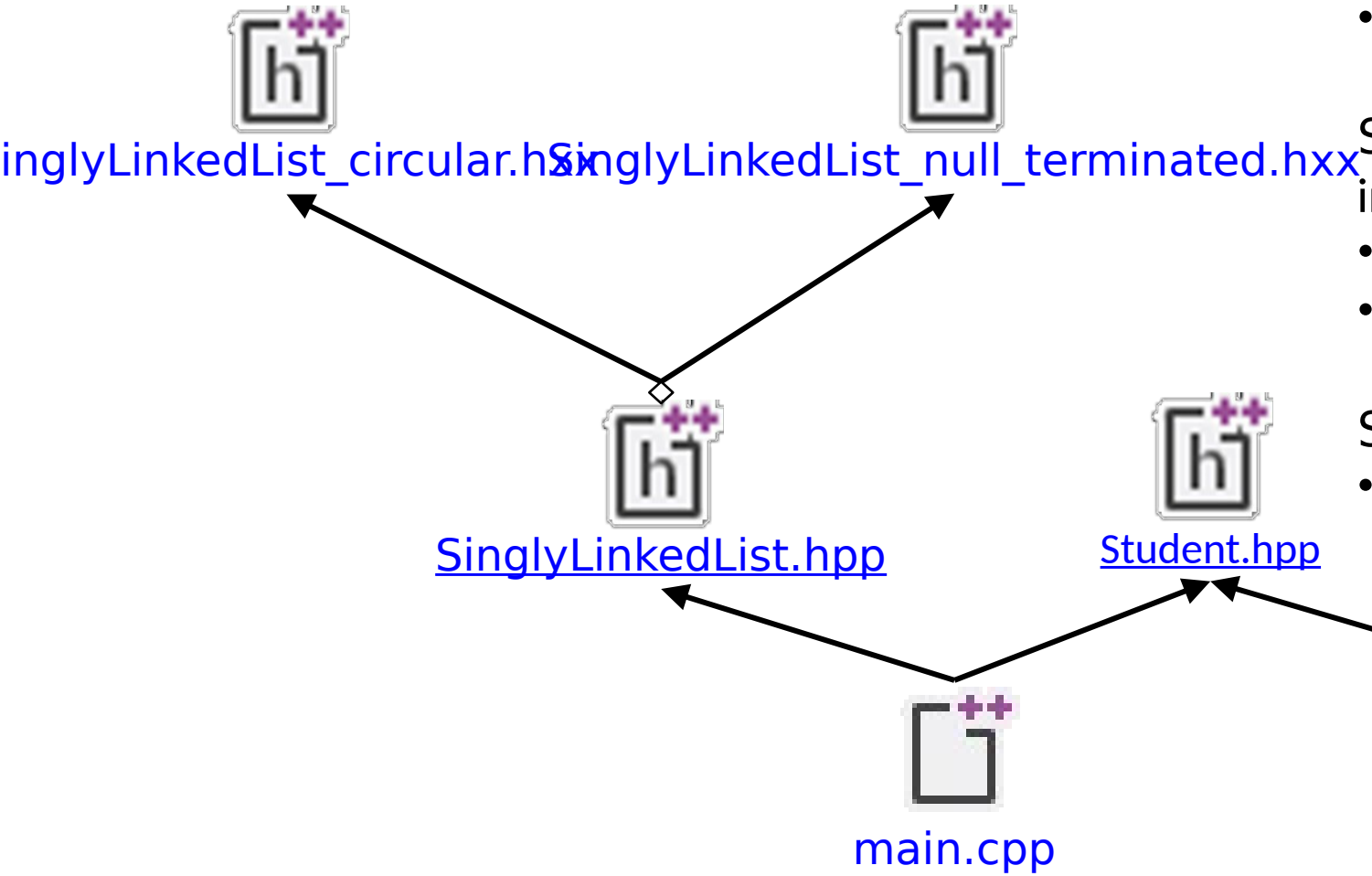
- Construct, destruct, assign
- Copy, compare
- Iterate
- Access elements
  - front, back
- Query
  - empty, size
- Operations
  - Insert, erase, clear
  - push_front, push_back, pop_front, pop_back





CPSC 131 T. L. Bettens

Singly Linked List Implementation Example



`main.cpp` includes both

- SinglyLinkedList.hpp and Student.hpp

`SinglyLinkedList.hpp` conditionally includes either

- SinglyLinkedList_null_terminated.hxx or
- SinglyLinkedList_circular.hxx

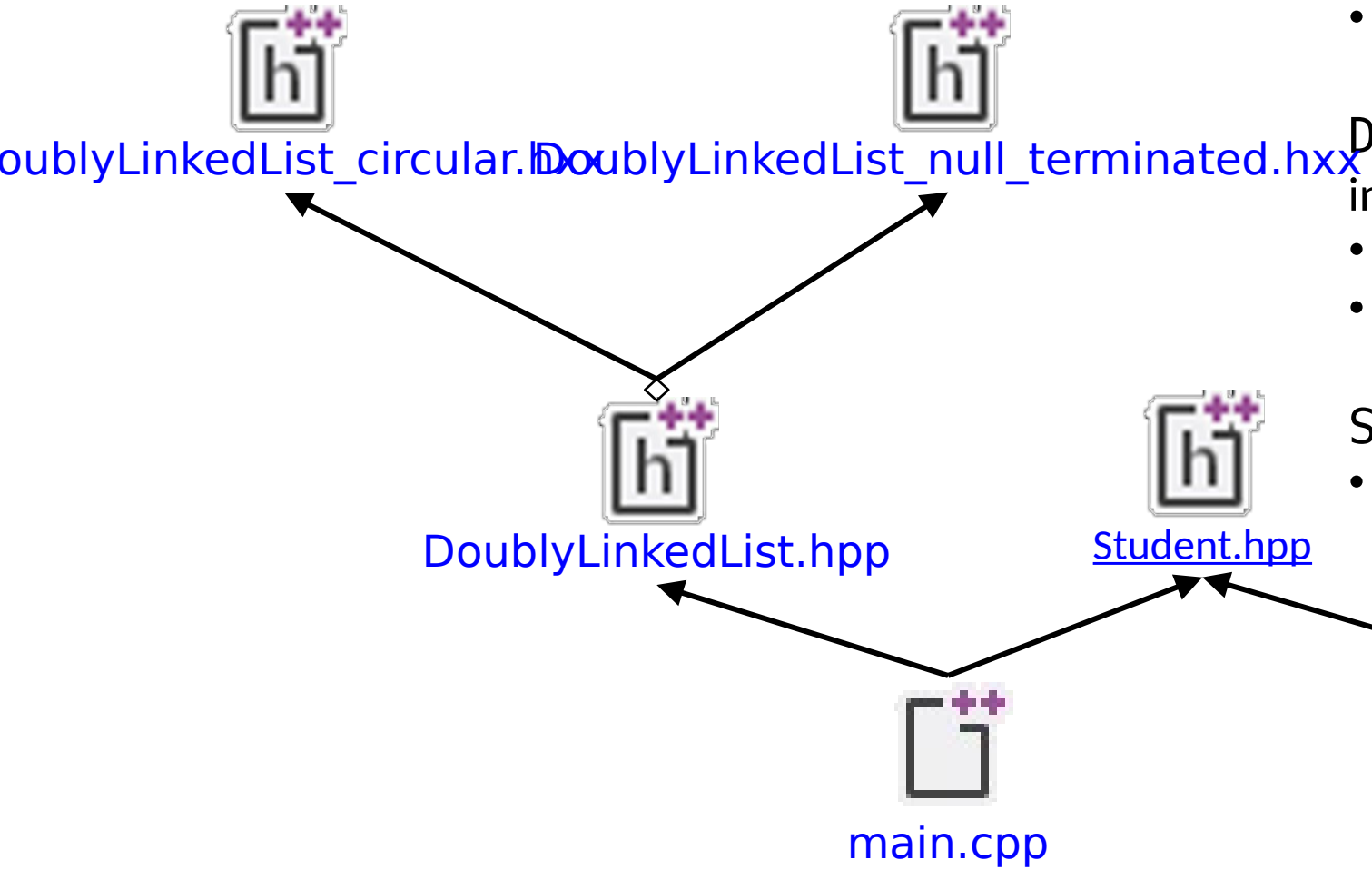
`Student.cpp` includes only

- Student.hpp

In Class Sketching Activity

Using the Implementation Examples, for each of the major operations of Singly Linked Lists, step through the code and sketch the resulting structure

Doubly Linked List Implementation Example



`main.cpp` includes both

- DoublyLinkedList.hpp and Student.hpp

`DoublyLinkedList.hpp` conditionally includes either

- DoublyLinkedList_null_terminated.hxx or
- DoublyLinkedList_circular.hxx

`Student.cpp` includes only

- Student.hpp

In Class Sketching Activity

Using the Implementation Examples, for each of the major operations of Doubly Linked Lists, step through the code and sketch the resulting structure

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)

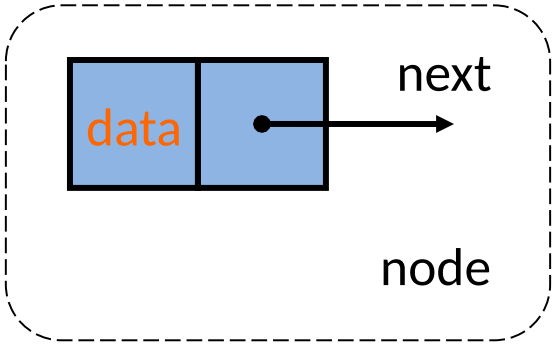
Function	Analysis – std::forward_list<T> (Singly Linked List)	Analysis – std::list<T> (Doubly Linked List)
push_front	O(1)	same
resize	O(n)	same
find	O(n) linear search from begin() to end() *(i.e. head to tail)*	same

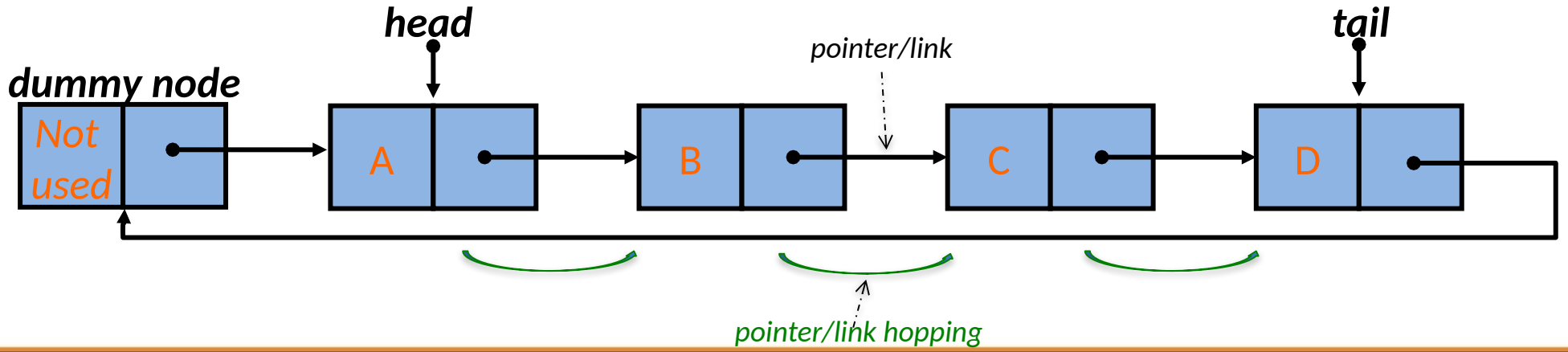
Analysis of the Vector Abstract Data Type
Complexity Analysis (5)

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

Circular Singly Linked Lists w/one dummy node

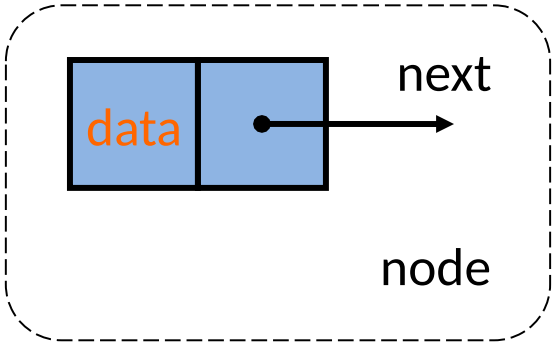
- A singly linked list is a concrete **data structure** consisting of a sequence of nodes
- Each node stores
  - Data element
  - link to the next node

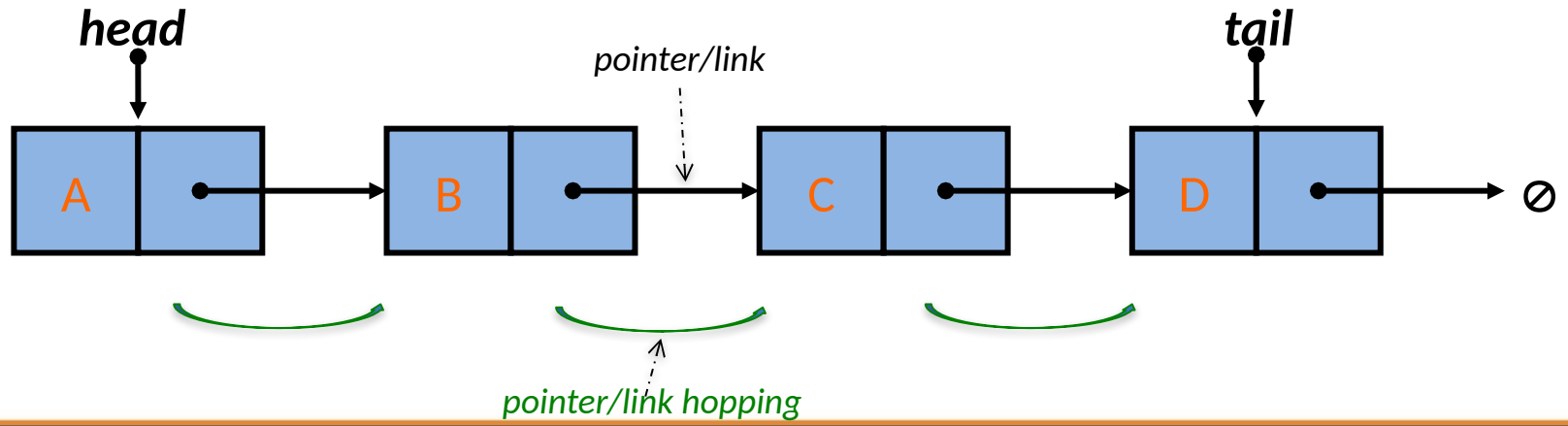




Null-Terminated Singly Linked Lists

- 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



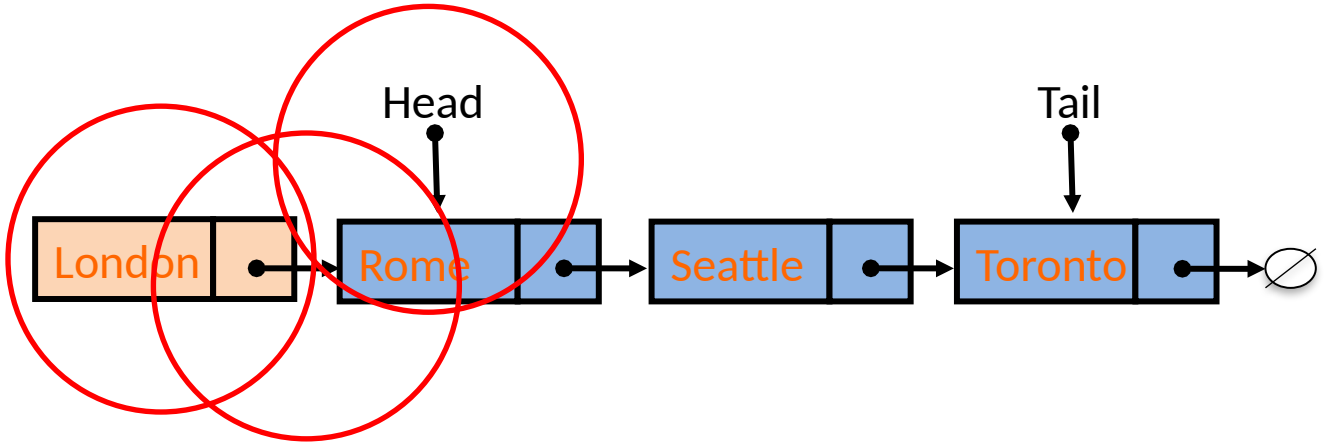


Inserting at the Head

1. Allocate and populate a new node

2. Have new node point to old head

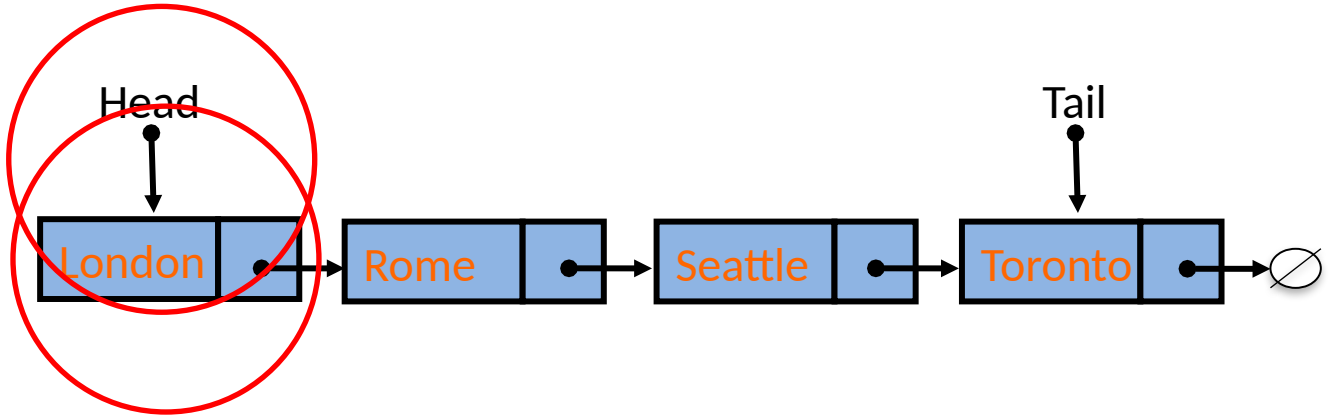
3. Update head to point to new node



Deleting at the Head

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

2. Delete the former first node



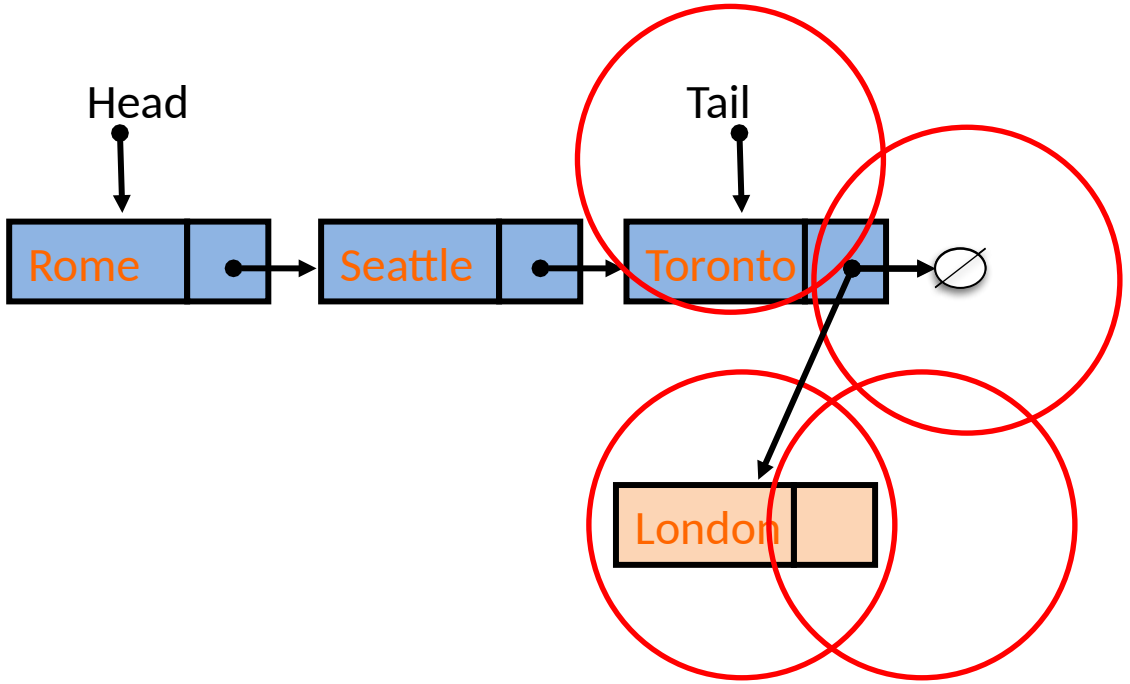
Inserting at the Tail

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



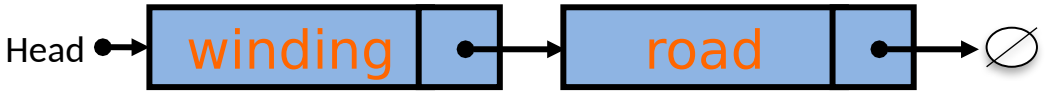
Draw data structure for this code

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();
```



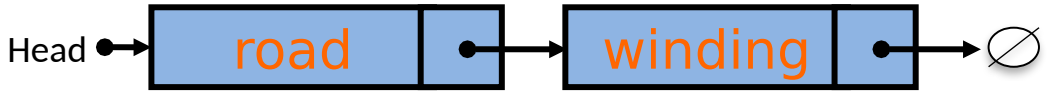
Draw data structure for this code

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();
```



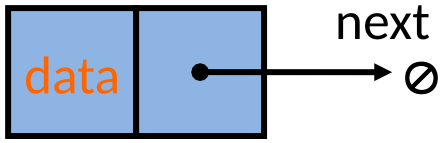
Nodes

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

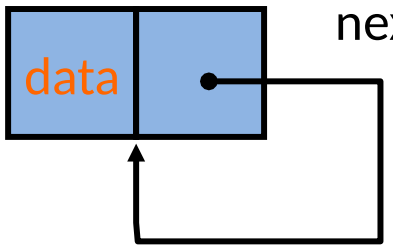
Null-Terminated SLL Node definition




```
template<typename T>
struct SinglyLinkedList<T>::Node
{
  Node() = default;
  Node( T value ) : _data{ std::move(value) } {}

  T       _data = T{};
  Node * _next = nullptr;
};
```

Circular SLL Node definition

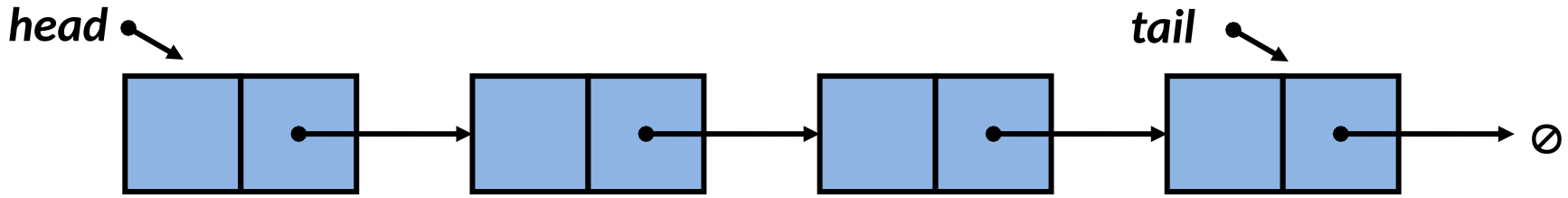


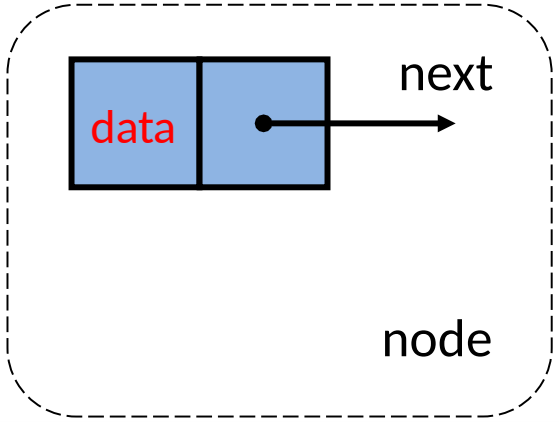

```
template<typename T>
struct SinglyLinkedList<T>::Node
{
  Node() = default;
  Node( T value ) : _data{ std::move(value) } {}

  T       _data = T{};
  Node * _next = this;
};
```

Singly Linked List and Doubly Linked List

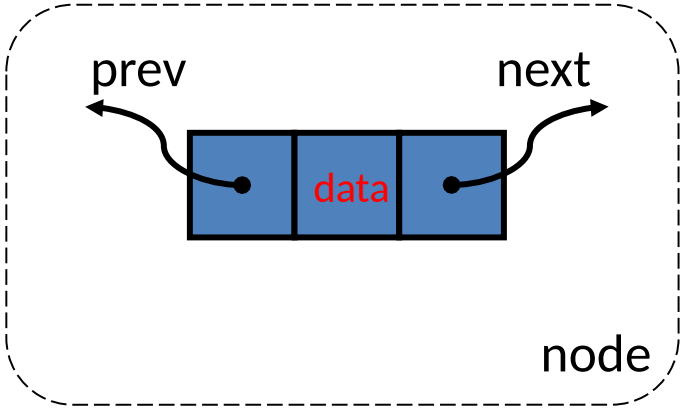
What if want to access data in reverse order?





Node:
- element
- next pointer
- previous pointer

Linked List:
- length
- Head
- Tail



Doubly Linked List Node

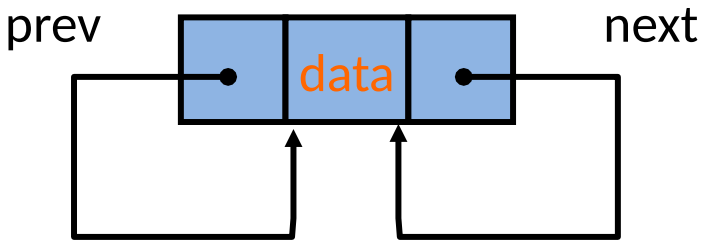
Null-Terminated DLL Node definition


```
template<typename T>
struct DoublyLinkedList<T>::Node
{
  Node() = default;
  Node( const T value ) : _data( std::move(value) )
{}

  T       _data = T{};

  Node * _next = nullptr;
```

Circular DLL Node definition




```
template<typename T>
struct DoublyLinkedList<T>::Node
{
  Node() = default;
  Node( T value ) : _data( std::move(value) ) {}

  T       _data = T{};
  Node * _next = this;
  Node * _prev = this;
};
```

Circular Doubly Linked Lists w/one dummy node

- Key ideas
  - Keep a previous pointer *in addition to a next pointer* at every node
  - Keep a dummy node at the head of circular list