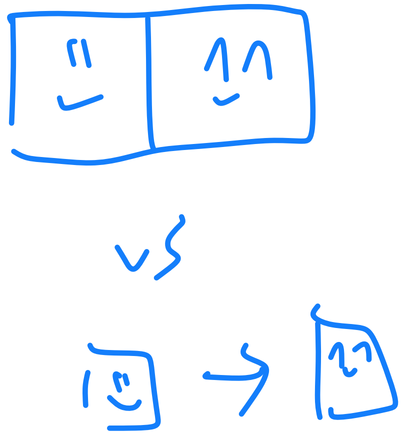
Data Structures

Linked Lists & (maybe) Array Lists

CS 225
Brad Solomon

January 30, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Review the importance of index in a linked list

Finish implementing the List ADT (as a linked list)

Discuss data variables for implementing array lists

Explore the List ADT (as an array list)

List ADT

A list is an **ordered** collection of items

Items can be either **heterogeneous** or **homogenous**

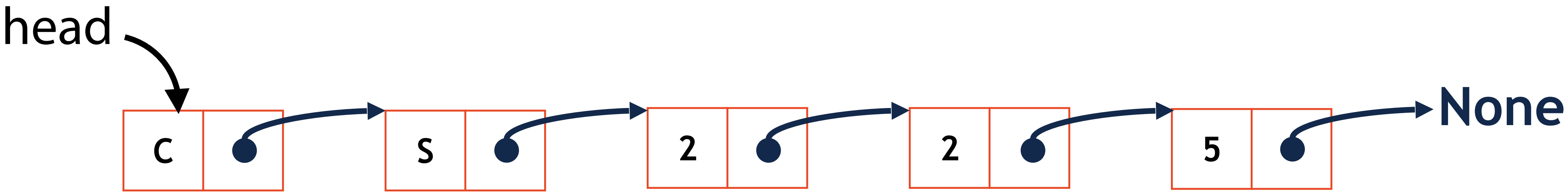
The list can be of a **fixed size** or is **resizable**

A minimal set of operations (that can be used to create all others):

1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations

1. Linked List

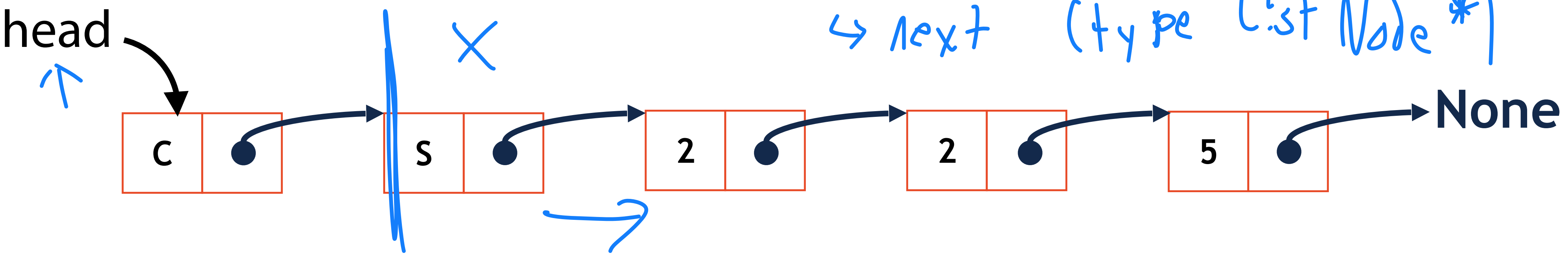


2. Array List

C	S	2	2	5					

Where we left off…

ListNode
↳ data (type T &)
↳ next (type ListNode *)



1. Singly linked list only accesses forward —>

↳ b/c only next

2. To insert in arbitrary position we need to access what value?

↳ The previous node to insert location

↳ the prev's next passed by ref

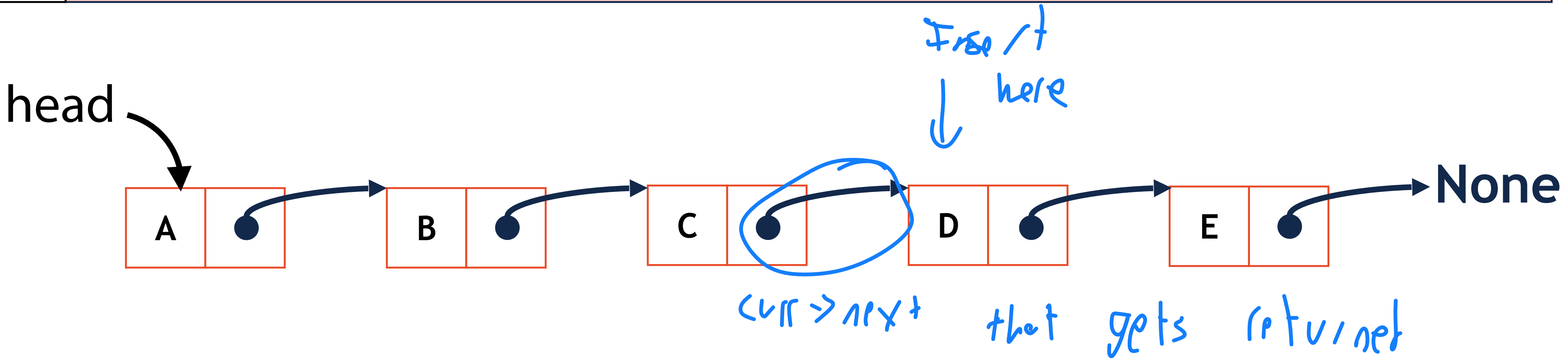
```cpp
// Iterative Solution:
template <typename T>
typename List<T>::ListNode *& List<T>::_index(unsigned index) {
  if (index == 0) { return head; }
  else {
    ListNode *curr = head;
    for (unsigned i = 0; i < index - 1; i++) {
      curr = curr->next;
    }
    return curr->next;
  }
}
```

← O(1)

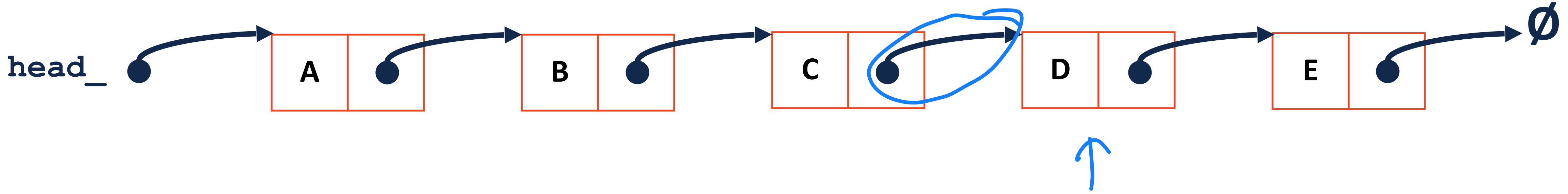
← n x

← O(1)

↖ O(1)



Comparing pointer to reference-to-pointer




```
ListNode * curr = _index(3);
```

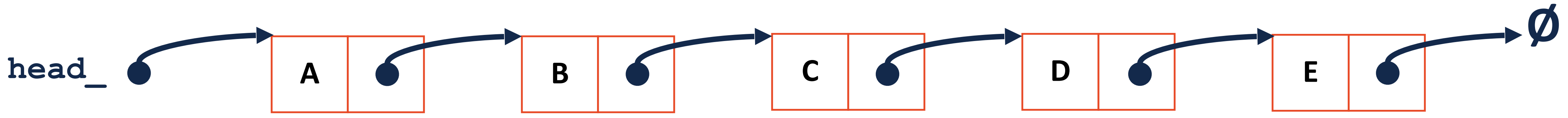
We can access `curr->data` and `curr->next` but not `previous.next`

```
ListNode *& curr = _index(3);
```

We can access `curr.data, curr.next and...`

`curr == previous.next`

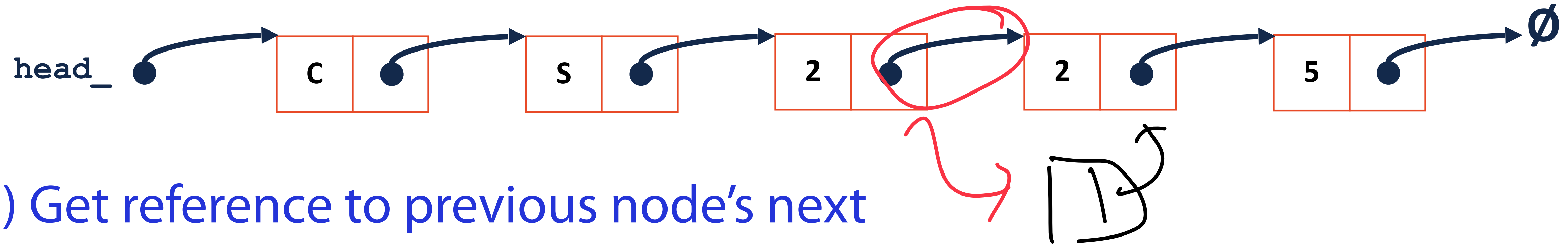
Comparing pointer to reference-to-pointer




```
ListNode * curr = _index(3);

curr = new ListNode(x);
```

```
ListNode *& curr = _index(3);

curr = new ListNode(x);
```

Linked List: `insert(data, index)`



1) Get reference to previous node's next

```
ListNode *& curr = _index(index);
```

2) Create new ListNode

```
ListNode * tmp = new ListNode(data);
```

3) Update new ListNode's next

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

ListNode(data, curr)

4) Modify the previous node to point to new ListNode

```
curr = tmp;
```

Lets compare…

List.hpp

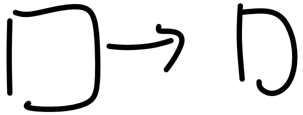
```
template <typename T>
void List<T>::insertAtFront(const T& t)
{
  ListNode *tmp = new ListNode(t);

  tmp->next = head_;

  head_ = tmp;

}
```

```
template <typename T>
void List<T>::insert(const T & data,
unsigned index) {



  ListNode *& curr = _index(index);



  ListNode * tmp = new ListNode(data);



  tmp->next = curr;



  curr = tmp;
}
```

What is the Big O of insert (given list of size n)?

O(1) / O(n)

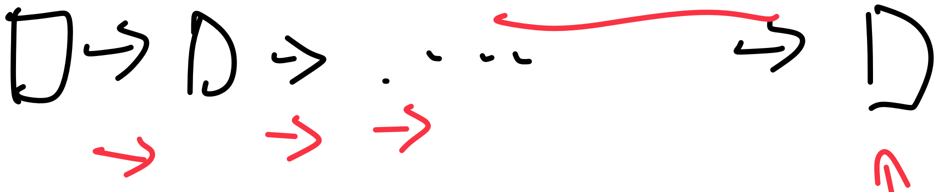
small (n=2)



↑ index of last item

1 operation

1 billion



1 billion - 1 calls to next

List.hpp

```
template <typename T>
void List<T>::insert(const T & data,
unsigned index) {


  ListNode *& curr = _index(index);
```

↳ O(n)

```
  ListNode * tmp = new ListNode(data);
```

↳ O(1)

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

↳ O(1)

```
  curr = tmp;
}
```

↳ O(1)

Join Code: 225

List Random Access [ ]

getData()

Given a list L, what operations can we do on L[ ]?

↳ getValue() , x = list[i] // get value

↳ set value() , list[i] = 10; // set value

What return type should this function have?

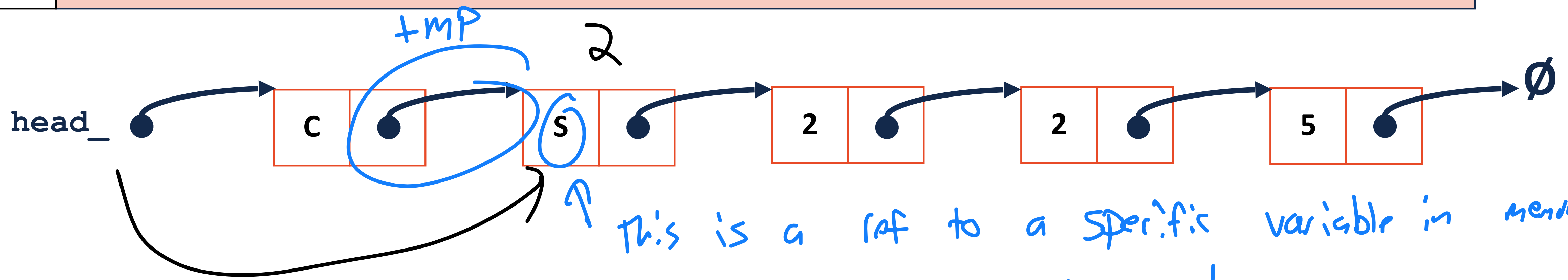
T& ✓ | ListNode ✗ | ListNode * | ListNode *&

Join Code: 225

List.hpp

```
template <typename T>
T & List<T>::operator[](unsigned index) {

  ListNode *& tmp = _index[index];      O(n)

      return tmp->data;      O(1)

}
```

Random access is O(n) in singly linked list



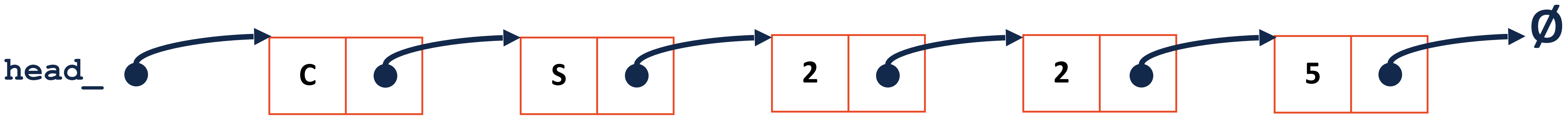
This is a ref to a specific variable in memory

NOT the value alone

List.hpp

```
template <typename T>
T & List<T>::operator[](unsigned index) {


ListNode *&new_node = _index(index);


return new_node->data;


}
```

Join Code: 225



What is the Big O of random access? $O(n)$

Linked List: remove(<parameters>)

What input parameters make sense for remove?

remove(unsigned index)

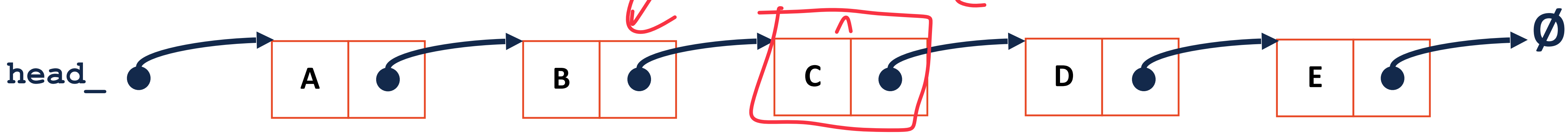
remove (ListNode *& node)

remove(ListNode * node)

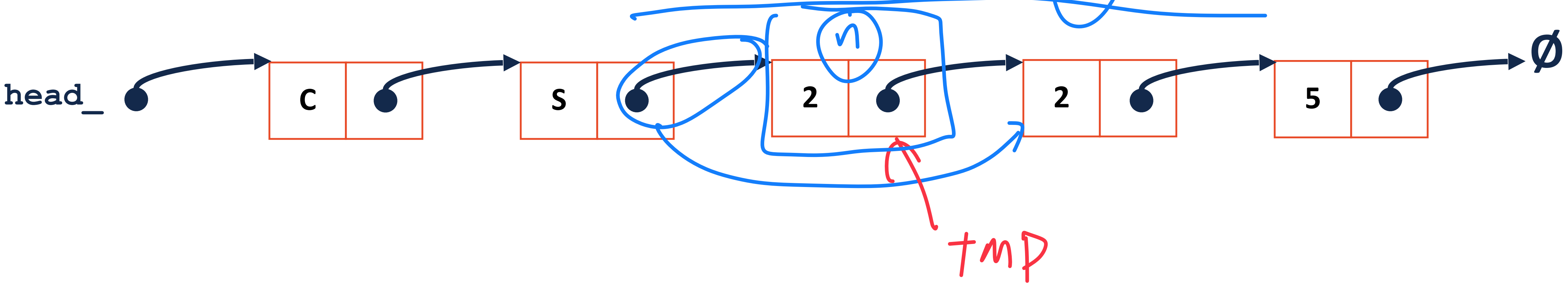
why will this be slower than ?

remove (T & value)

To delete I need but I have



Linked List: `remove(ListNode *& n)`




```
ListNode * tmp = n;

n = n->next;

delete tmp;
```

← b/c I am ref to existing pointer inside list

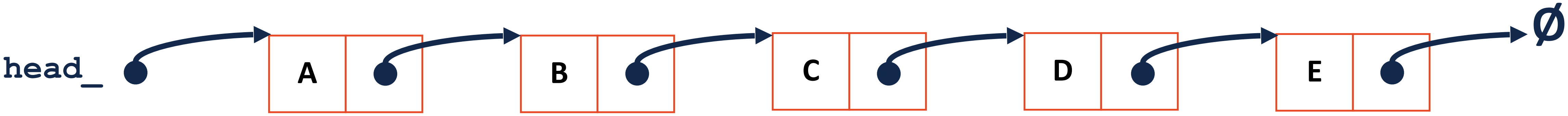
List.hpp

```
template <typename T>
T List<T>::remove(ListNode *& node) {

ListNode * temp = node;
node = node->next;
T data = temp->data;
delete temp;
return data;

}
```

O(1)
O(1)
O(1)
O(1)
O(1)

copy of T here
↳ local scope!

Tangent point: why does this return type T?
↳ return must be by value b/c it refers to local scope



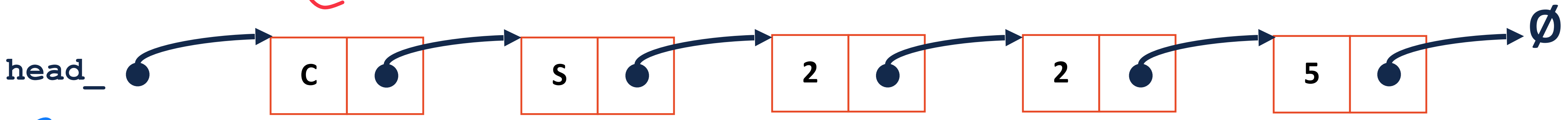
Linked List: remove(T & data)

in

↳ can not use index!

Post class Q:
You can do T data
but we frown upon it
b/c that makes a copy

curr



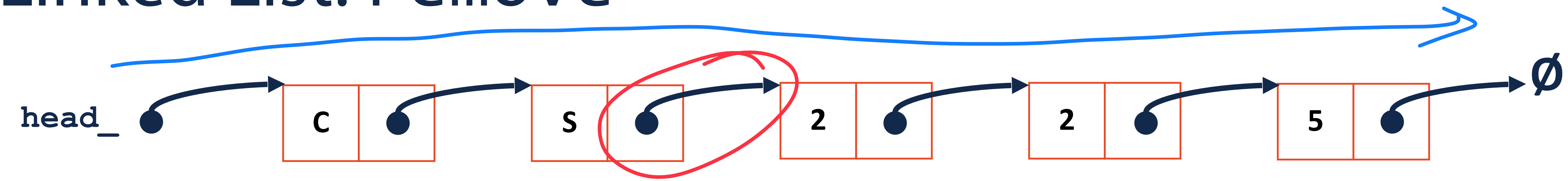
start at head

repeat calls to next

Stop when inData == curr->data

↳ or don't stop until end of list

Linked List: remove



Running time for **remove(ListNode *&)**

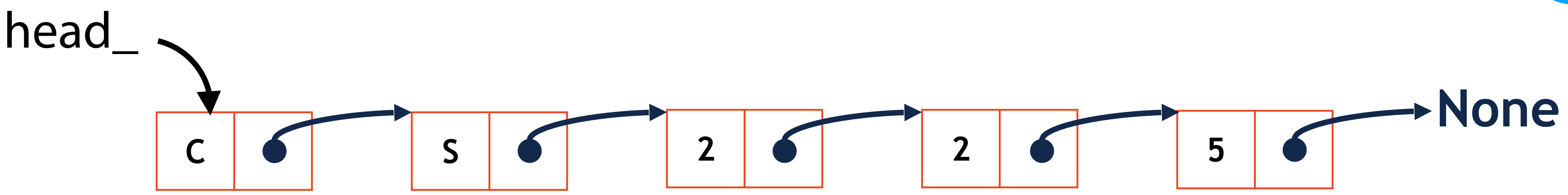
↳ O(1)

↳ Think of this like removeFront()

Running time for **remove(T & data)**

↳ O(n)

Linked List Runtimes



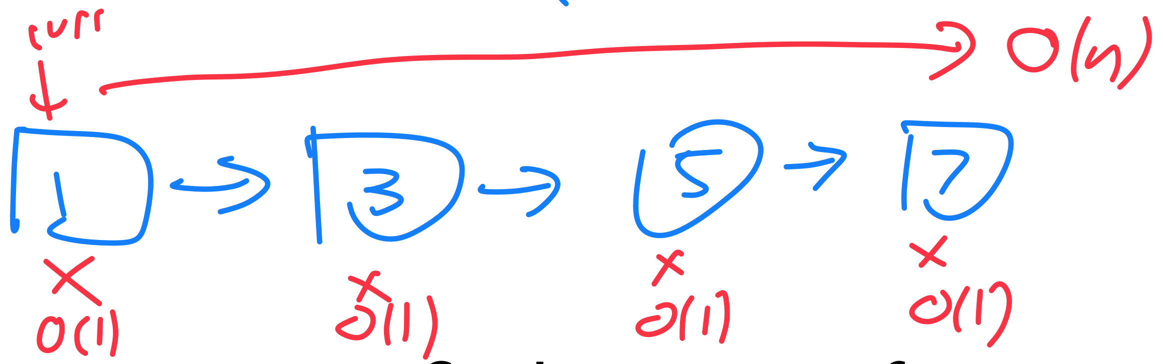
	@Front	@RefPointer	@Index or by val
Insert	$O(1)$	$O(1)$	$O(n)$
Delete	$O(1)$	$O(1)$	$O(n)$

Thinking critically about linked lists…

Imagine I want to remove all odds

What common list use case lets us take advantage of O(1) edits?

↳ If I need to walk through list O(n)



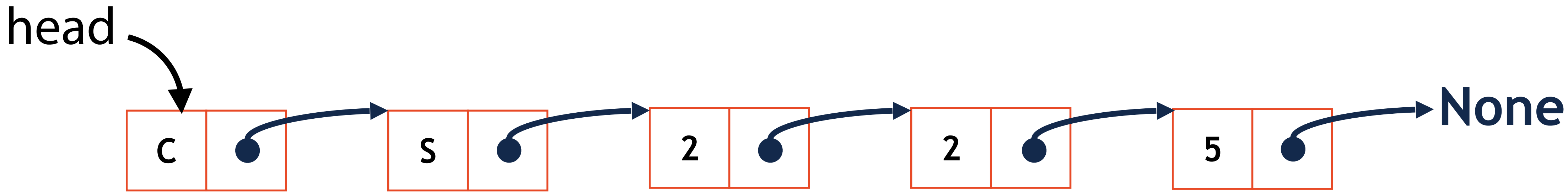
What is the runtime to find an item of interest?

↳ If I find item of interest during walk, removal is O(1)

Consider carefully how to best implement a complex function

List Implementations

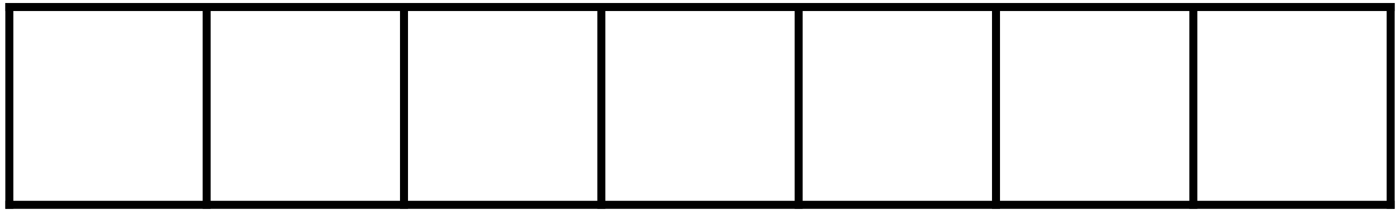
1. Linked List



2. Array List

C	S	2	2	5					

Array List



An array is allocated as continuous memory.

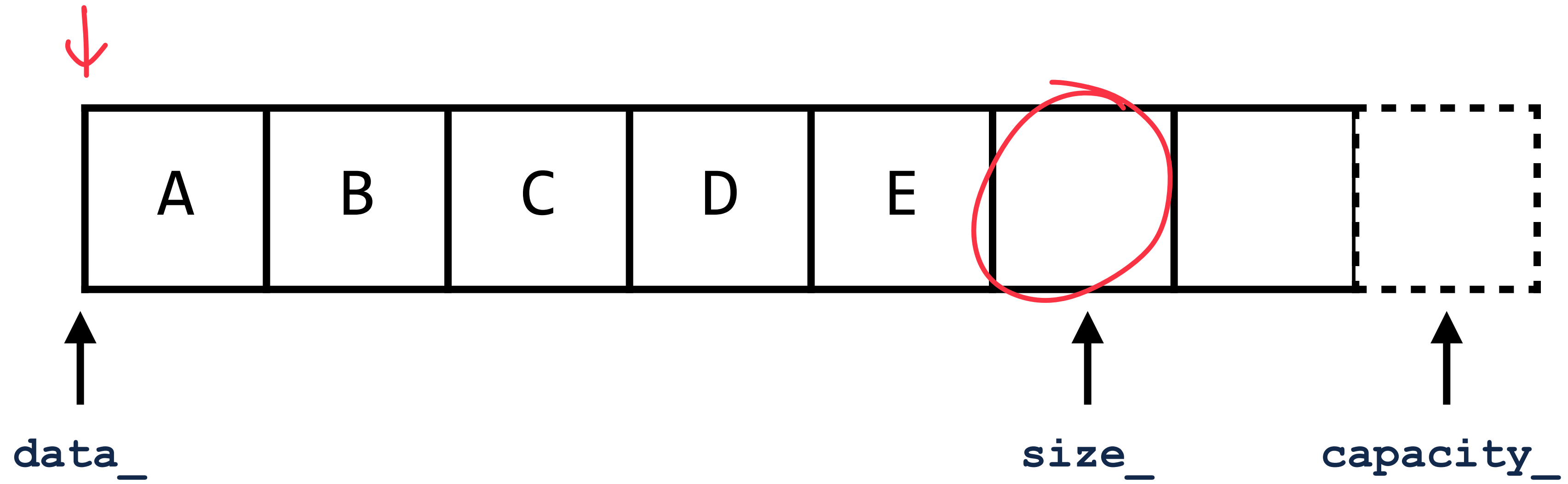
Three values are necessary for efficient array usage:

1) Data – the start position of my array (memory address)

2) Size – the current # of items in list (memory address)

3) Capacity – the max size of the list (memory address)

Array List



In C++, vector is implemented as:

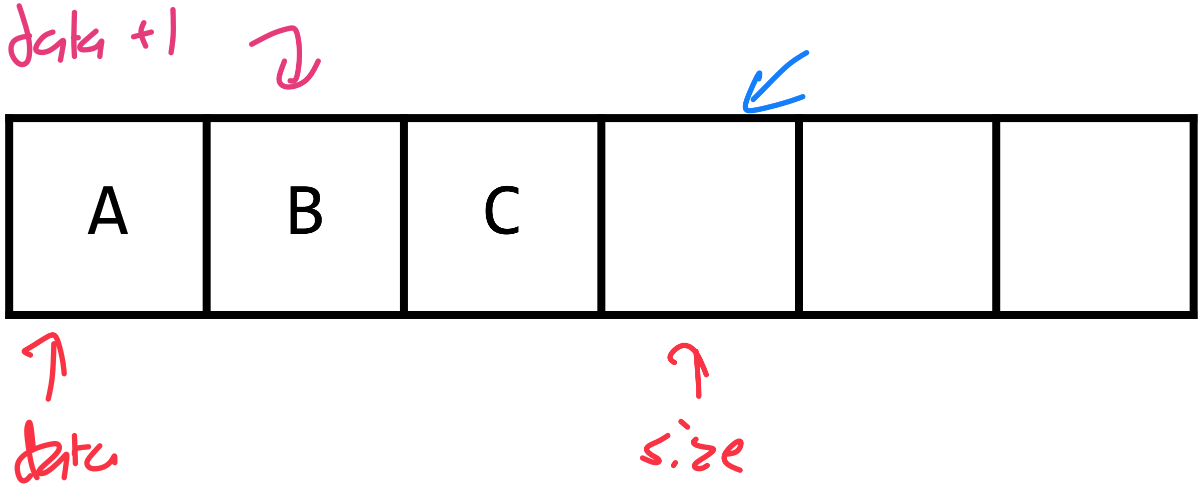
1) **Data:** Stored as a pointer to array start

2) **Size:** Stored as a pointer to the next available space

3) **Capacity:** Stored as a pointer past the end of the array

List.h

```cpp
#pragma once

template <typename T>
class List {
public:
    /* --- */
private:
  T *data_;

  T *size_;

  T *capacity_;

    /* --- */
};
```



Pointer math in C++ auto handles sizeof()

↳ int * p;

p++; ← increment by sizeof(int)

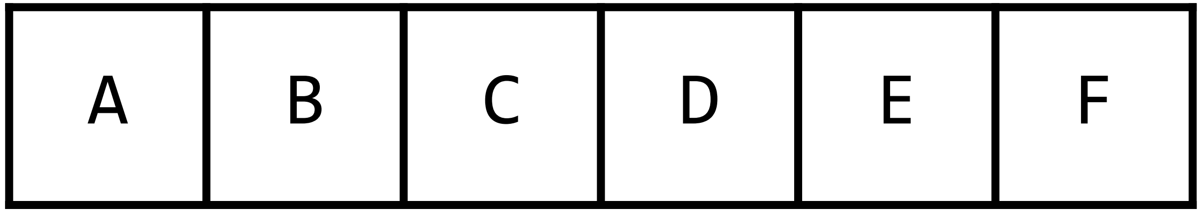
If I want to know the number of items in the array: size - data

↳ would return 3

O(1) time

List.h

```
#pragma once

template <typename T>
class List {
public:
    /* --- */
private:
  T *data_;

  T *size_;

  T *capacity_;

    /* --- */
};
```



size

capacity

How do I know if I'm at capacity? If size == capacity

Array List: [ ]

C	S	2	2	5					

data + index

$O(1)$ time

continue here next week

Array List: `insertFront(data)`

C	S	2	2	5					

Array List: `insertBack(data)`

C	S	2	2	5					

Array List: `insert(data, index)`

C	S	2	2	5					

Array List: `addspace(data)`

N	O	S	P	A	C	E