Data Structures

Linked Lists

CS 225

Brad Solomon

August 31, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Review fundamentals of linked lists

Implement insert, index, and remove operations

Discuss pointers vs references-to-pointers

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

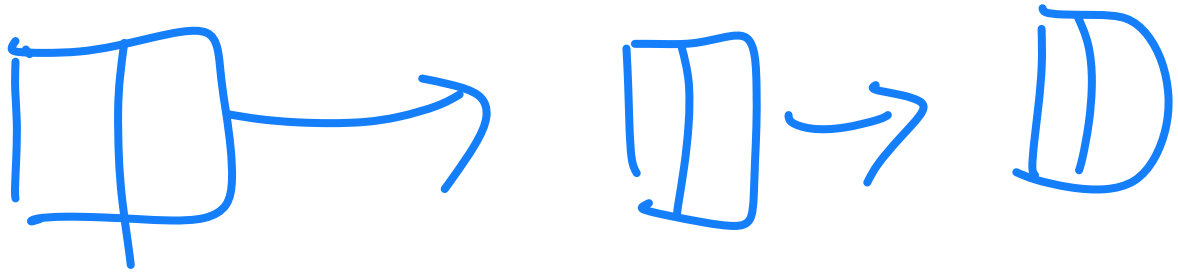
Insert

Remove

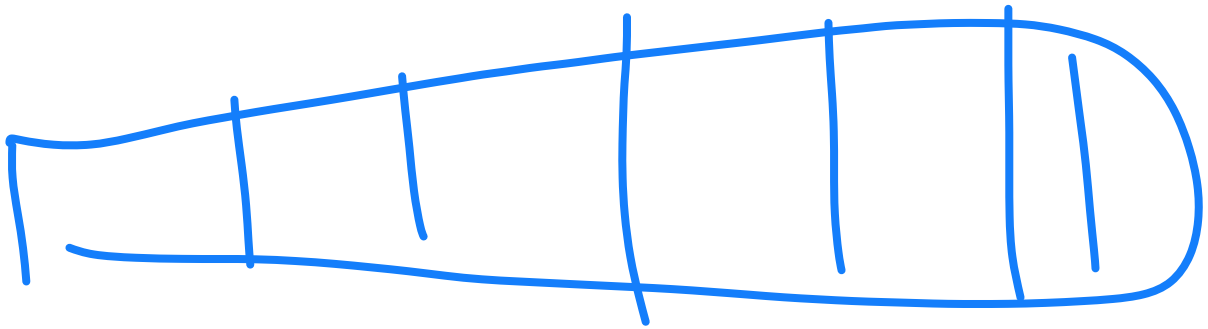
find

List Implementations

1. Linked List



2. Array List



List.h

```
 1  template <class T>
 2  class List {
 3    public:
 4      /* ... */
 5    private:
 …      class ListNode {
28        T & data;
29        ListNode * next;
30        ListNode(T & data) :
31         data(data), next(NULL) { }
32      };

33      ListNode *head_;
34  };
```

Join Code: 225

Can we access **x** from **y**?

Y → X 75% No

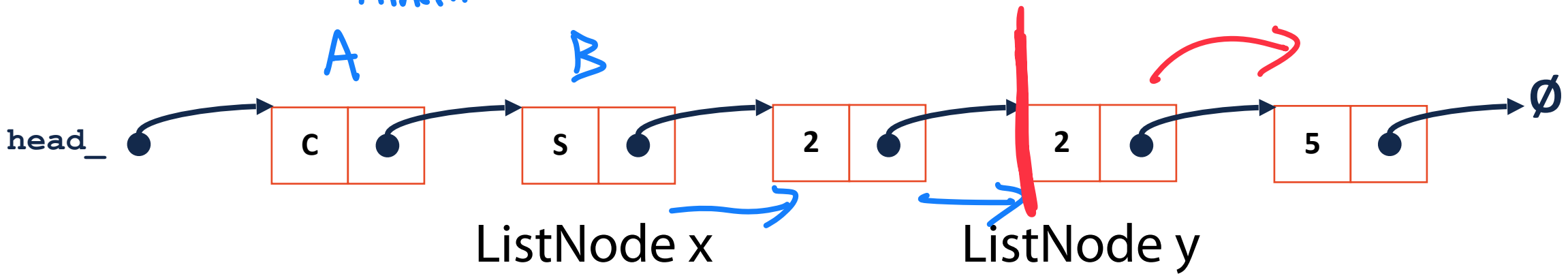
All nodes move in one direction

Can we access **y** from **x**?

X → Y 76% Yes

X → next → next

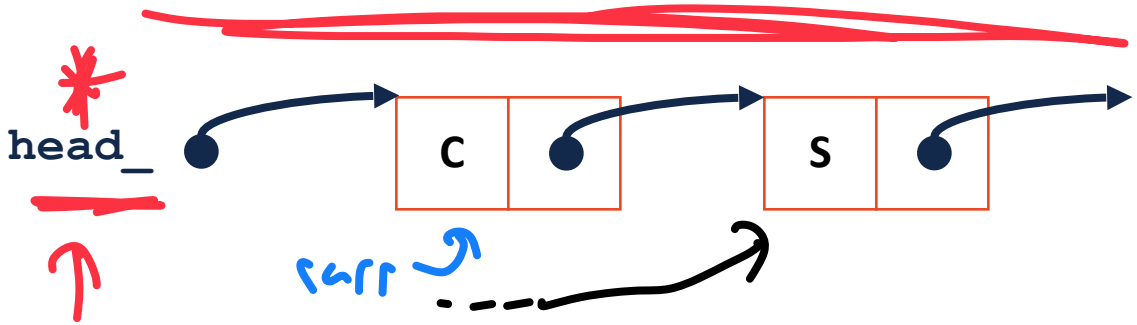
A.next = B



List.h

```
1   #pragma once
2
3
4   class List {
5     public:
      /* ... */
…
28    void insertAtFront(const T& t);
29
30    private:
31      class ListNode {
32        T & data;
33        ListNode * next;
34        ListNode(T & data) :
35         data(data), next(NULL) { }
36      };
37
38      ListNode *head_;
39
40      /* ... */
…
…
…
79
```

How do I access list given head_?



ListNode * curr = head_;

curr = curr → next;

List.h

```
1   #pragma once
2
3
4   class List {
5     public:
…       /* ... */
28    void insertAtFront(const T& t);
29
30    private:
31      class ListNode {
32        T & data;
33        ListNode * next;
34        ListNode(T & data) :
35          data(data), next(NULL) { }
36      };
37
38      ListNode *head_;
39
40      /* ... */
…
…   };
…
79
```

What is missing in this code?

List.h

```
1   #pragma once
2
3   template <typename T>
4   class List {
5     public:
      /* ... */
28    void insertAtFront(const T& t);
29
30    private:
31      class ListNode {
32        T & data;
33        ListNode * next;
34        ListNode(T & data) :
35         data(data), next(NULL) { }
36      };
37
38      ListNode *head_;
39
40      /* ... */
      ...
      };
      ...
79  #include "List.hpp"
```

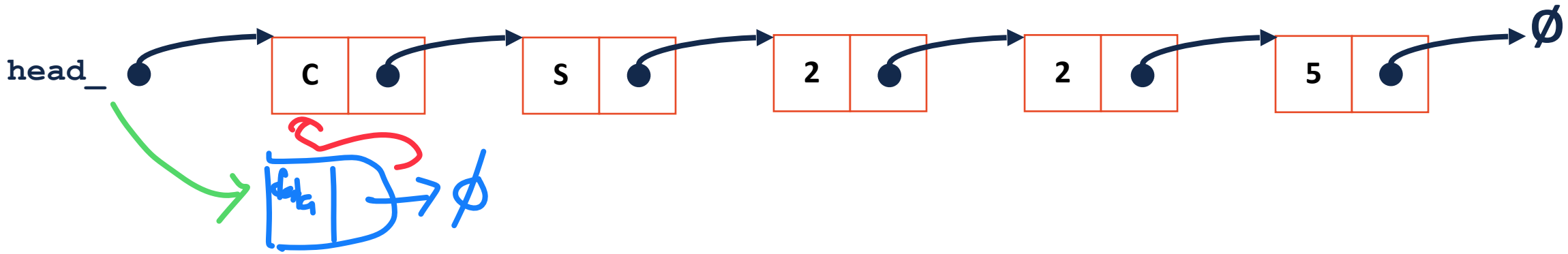
List.hpp

```
1   CS 225 style choice!
2
3
4   void List<T>::insertAtFront(const T& t)
5   {
6
7     ↳ Reminder that templates are
8       recipes!
9
10
11    ↳ Template functions written in
12      header
13
14
15
16
17
18
19
20
21
22  }
```

main.cpp

myL = new List<int>

Linked List: `insertAtFront(data)`



1) Make new Node

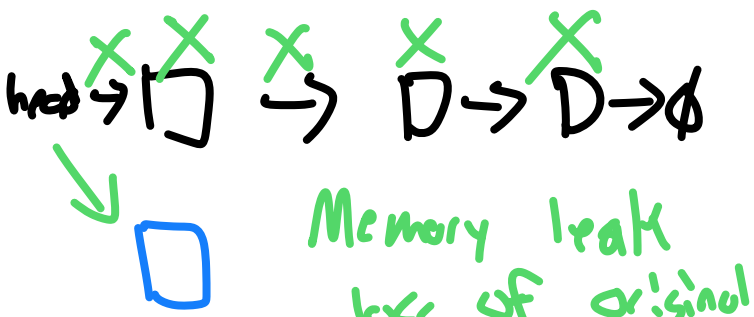
`ListNode * t = new ListNode(data, nullptr);`

2) Let new node point to head

`t->next = head_;`

3) Update head_ pointer

What happens if (1), (3), (2)?



List.h

```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
…      /* ... */
28   private:
29     class ListNode {
30       T & data;
31       ListNode * next;
32       ListNode(T & data) :
          data(data), next(NULL) { }
33     };
34
35     ListNode *head_;
36
37     /* ... */
38
39 };
…
…
79 #include "List.hpp"
```

List.hpp

```
1
2
3  template <typename T>
4  void List<T>::insertAtFront(const T& t)
5  {
6
7    ListNode *tmp = new ListNode(t);
8
9
10   tmp->next = head_;
11
12
13   head_ = tmp;
14
15 }
16
17
18
19
20
21
22
```

O(1)

O(1)

O(1)

D → D → ∅

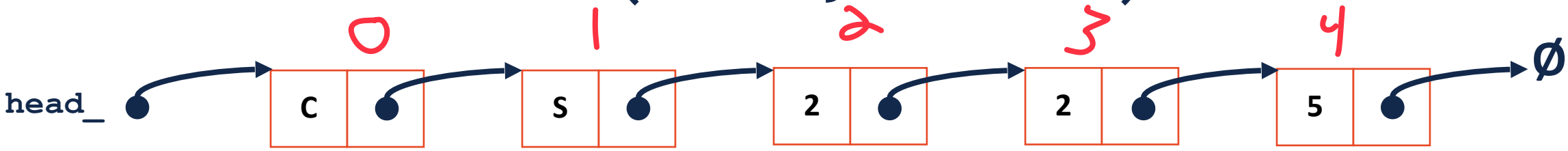
D → D → ... → D → ∅

1 billion elems

O(1)

Linked List: `insert(data, index)`

insert(X, 3)



Replacement

C → S → 2 → X → 5

↑

find index

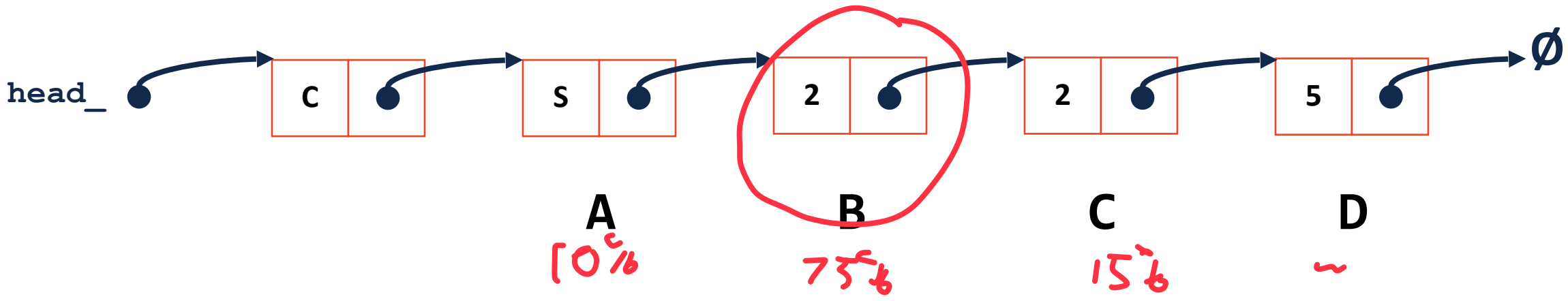
replace value

Insert

C → S → 2 → X → 2 → 5 → Ø

0 1 2 3

Linked List: `insert(data, index)` `insert(d,3)`



To insert a new ListNode at index 3, we need to **modify** which node?

↳ to insert at 3, we must have access to node 2

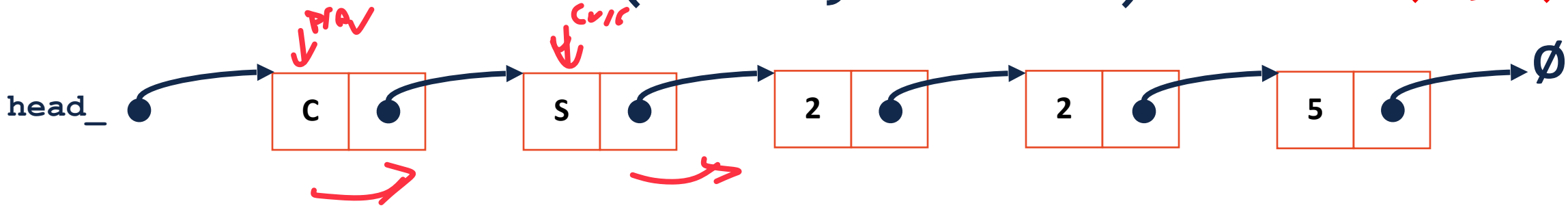
↳ We need B → next

Generally if index - 1 is prev,

we update prev → next

Join Code: 225

Linked List: `insert(data, index)` `insert(d,3)`



1) Get access to node @ position index - 1

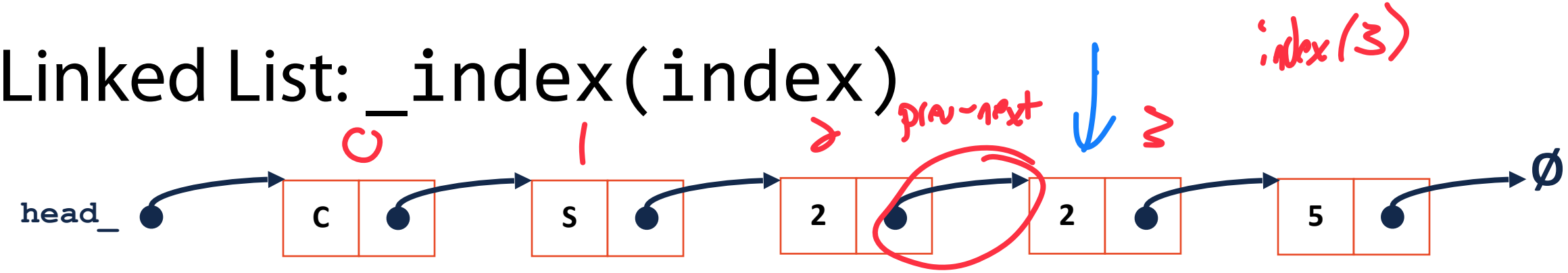
We ***could*** code up a solution to insert which uses some previous var

But lets be smarter!

Coding tip from last lecture: **Consider the entire interface**

↳ Insert ↳ Remove ↳ Find

Linked List: _index(index)



Lets write one function which is useful for insert / remove AND find

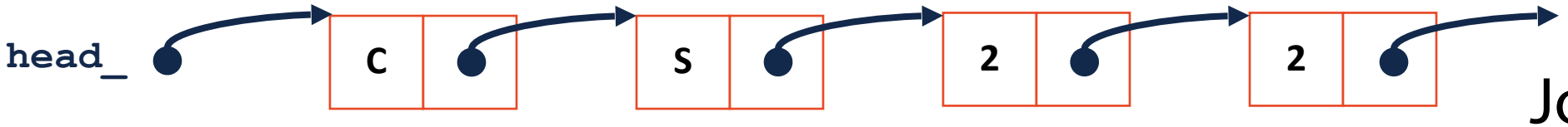
1) We need access to prev → next

2) We need the return value to be node @ index

not

index -1

Linked List: _index(index)



Join Code: 225

What should the return type of _index() be?

[template <class T>]

(A) T &

10%

(B) ListNode

5%

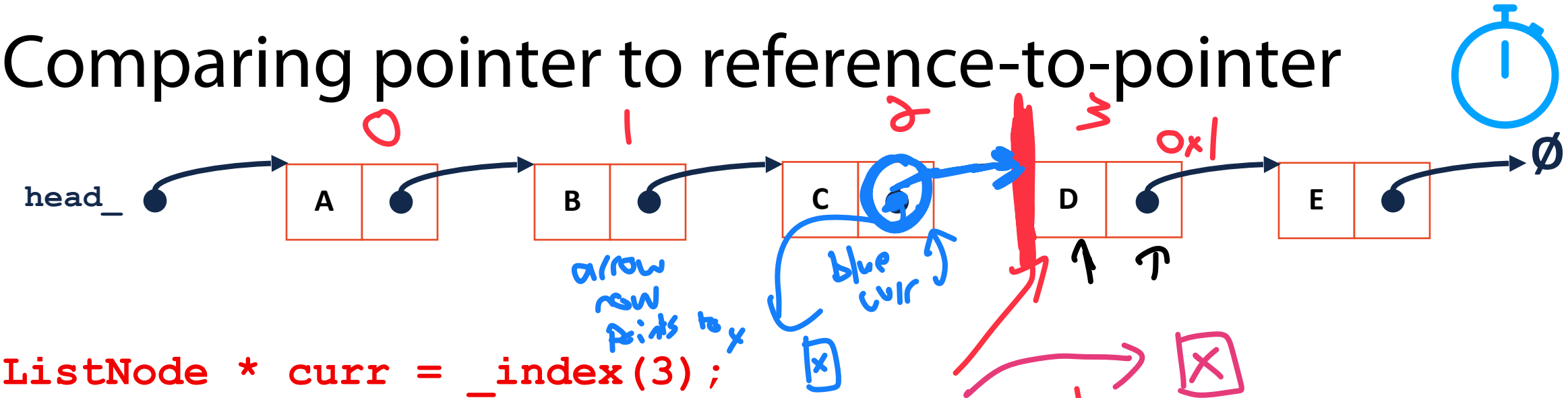
(C) ListNode *

65%

(D) ListNode *&

15 %

Comparing pointer to reference-to-pointer




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

↳ A Stark variable

curr = new ListNode(x);

↳ we can say curr→data
curr→next

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

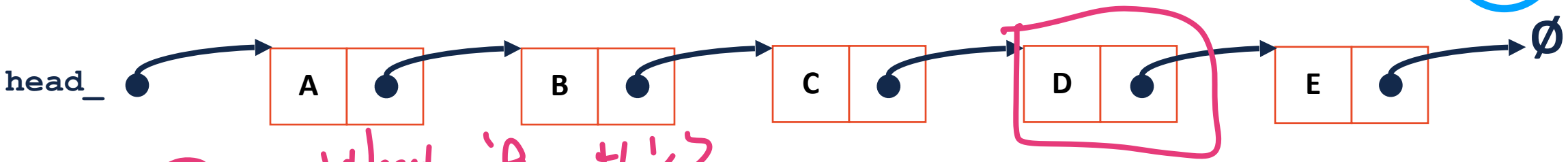
↳ An alias for the pointer prev→next

curr was 0x1
is now 0x5

→ we can say curr→data
curr→next

Curr = new ListNode(x)

Comparing pointer to reference-to-pointer



what is this?

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

↪ No diff w/ pointer version

↳ curr.data

curr.next

ListNode ** also works

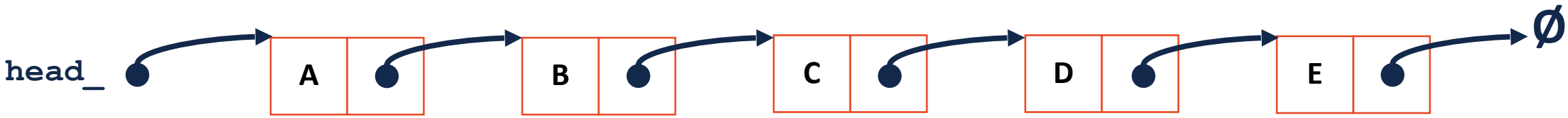
```
ListNode *& curr = _index(3);
```

This is ref to specific pointer in memory

↳ prev->next

You can do this!

Comparing pointer to reference-to-pointer




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

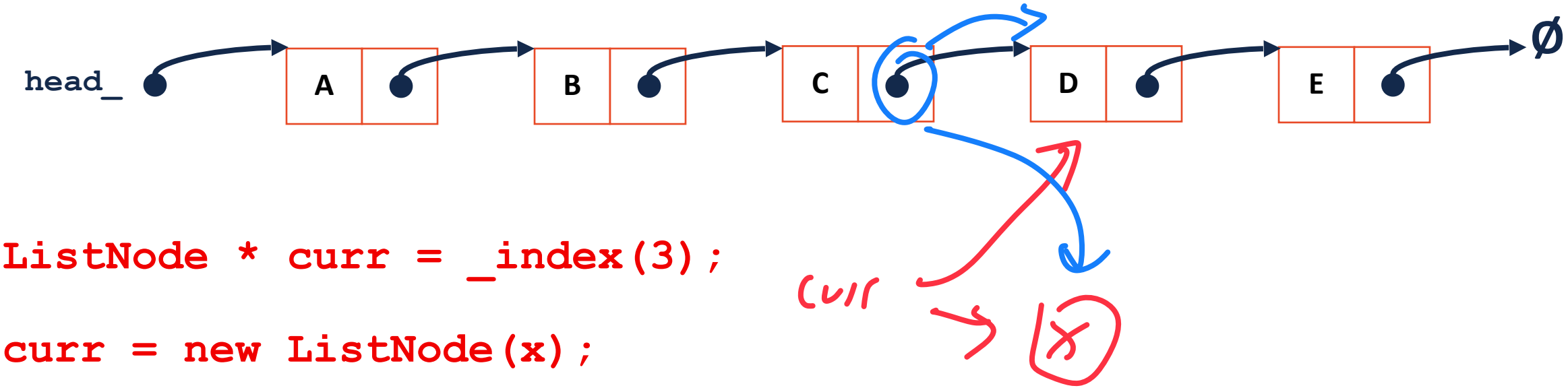
We can access **curr->data** and **curr->next** but not **previous.next**

```
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);
```

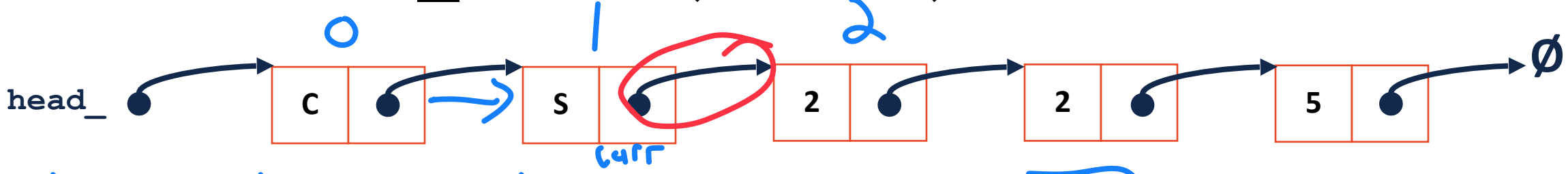
```
ListNode *& curr = _index(3);

curr = new ListNode(x);
```

easy to write!

LinkedList *& _index(index)

_index (2)



1) Get index -1 node

```
List Node * curr = head_;

for (unsigned i=0; i< index-1; i++){
        curr = curr->next;
}

return curr->next;
```

List.hpp

```
1  // Iterative Solution:
2  template <typename T>
3  typename List<T>::ListNode *& List<T>::_index(unsigned index) {
4    if (index == 0) { return head; }
5    else {
6      ListNode *curr = head;
7      for (unsigned i = 0; i < index - 1; i++) {
8          curr = curr->next;
9      }
10     return curr->next;
11   }
12 }
```

Check that index isn't out of bounds

Index returns ListNode * &

reference to existing ListNode * → head_ if index=0 or prev->next;

A brief tangent...

List.hpp

```
58 template <typename T>
59 typename List<T>::ListNode *& List<T>::_index(unsigned index){
60         return _index(index, head_)
61 }
```

```
63 template <typename T>
64 typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80 }
```

We did this w/ for

Let's try recursion on our own!

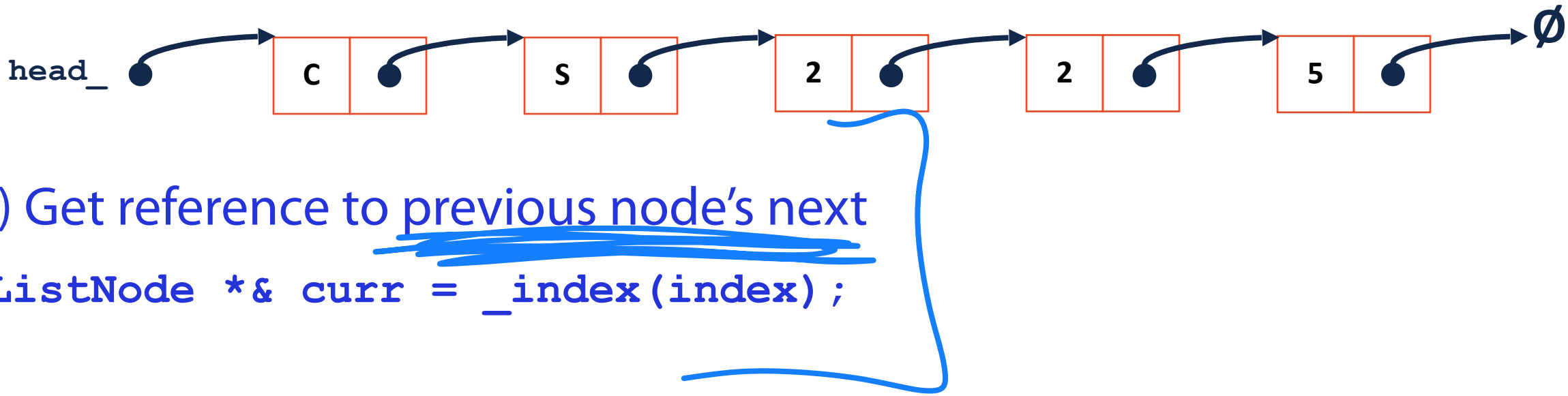
A brief tangent…

List.hpp

```
58  template <typename T>
59  typename List<T>::ListNode *& List<T>::_index(unsigned index){
60          return _index(index, head_)
61  }
```

```
63  template <typename T>
64  typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66
67
68      if (index == 0){ return root; }
69
70
71
72      if (root == nullptr){ return root; }
73
74
75
76      return _index(index - 1, root -> next);
77
78
79
80  }
```

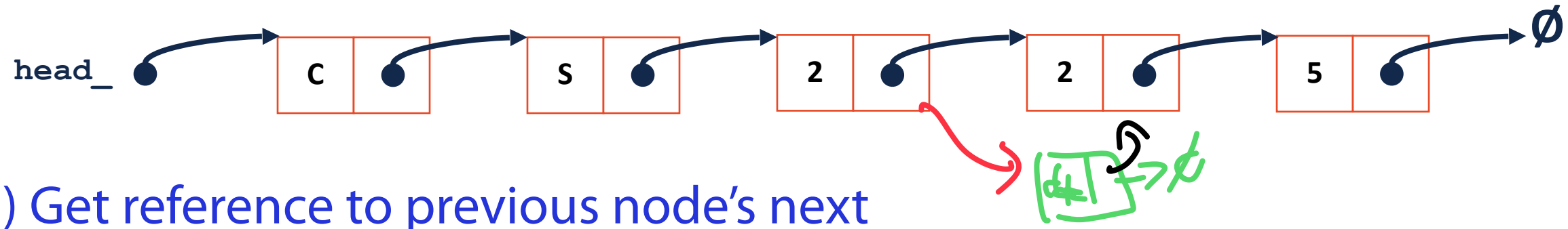
Linked List: `insert(data, index)`



1) Get reference to previous node's next

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

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;
```

4) Modify the previous node to point to new ListNode

```
curr = tmp;
```

Lets compare...

```
1
2  template <typename T>
3  void List<T>::insertAtFront(const T& t)
4  {
5    ListNode *tmp = new ListNode(t);
6
7    tmp->next = head_;
8
9    head_ = tmp;
10
11 }
12
13
14
15
16
17
18
19
20
21
22
```

!! ✓

List.hpp

```
1
2  template <typename T>
3  void List<T>::insert(const T & data,
4  unsigned index) {
5
6
7
8    ListNode *& curr = _index(index);
9
10
11
12   ListNode * tmp = new ListNode(data);
13
14
15
16   tmp->next = curr;
17
18
19
20   curr = tmp;
21 }
22
```

T/C ref to pointer, but is trivial!

What is the Big O of insert?

List.hpp

```
1
2  template <typename T>
3  void List<T>::insert(const T & data,
4  unsigned index) {
5
6
7
8    ListNode *& curr = _index(index);
9
10
11
12   ListNode * tmp = new ListNode(data);
13
14
15
16   tmp->next = curr;
17
18
19
20   curr = tmp;
21 }
22
```

Join Code: 225

List Random Access [ ]

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

```
T x = L[5];     // Get Value

L[5] = 10;      // Set Value
```

What return type should this function have?

List Random Access [ ]

What return type should this function have?

Join Code: 225

```
[template <class T>]
```

(A) `T &`

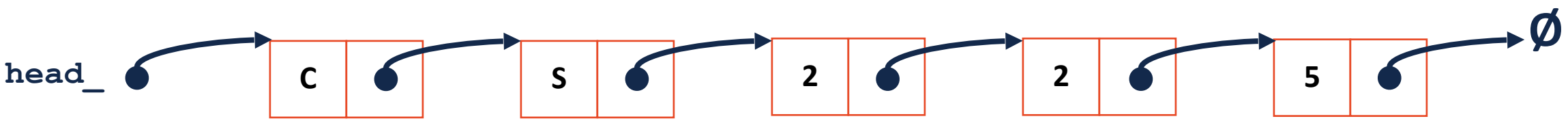
(B) `ListNode`

(C) `ListNode *`

(D) `ListNode *&`

List.hpp

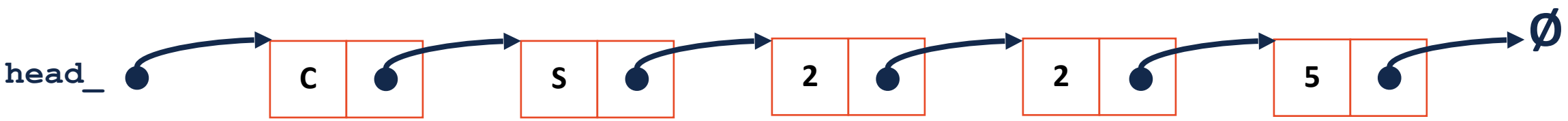
```
48 template <typename T>
49 T & List<T>::operator[](unsigned index) {
50
51
52
53
54
55
56
57
58 }
```



List.hpp

```
48 template <typename T>
49 T & List<T>::operator[](unsigned index) {
50
51
52 ListNode *&new_node = _index(index);
53
54
55 return new_node->data;
56
57
58 }
```

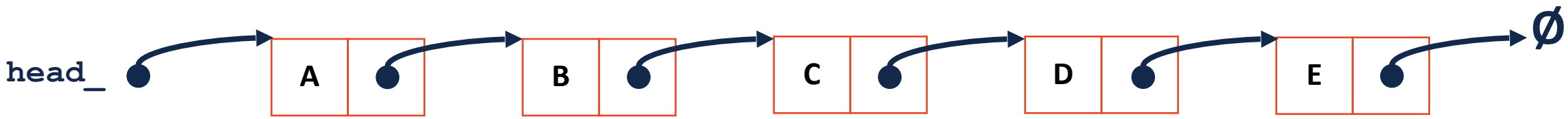
Join Code: 225



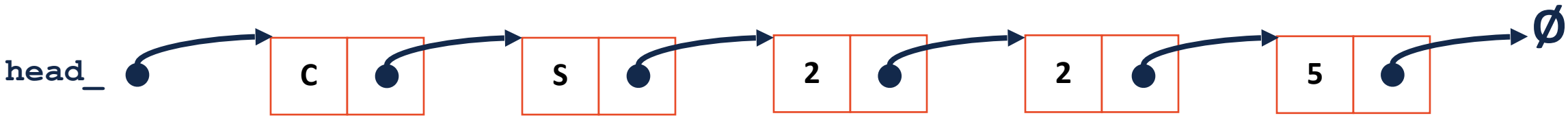
What is the Big O of random access?

Linked List: `remove(<parameters>)`

What input parameters make sense for remove?

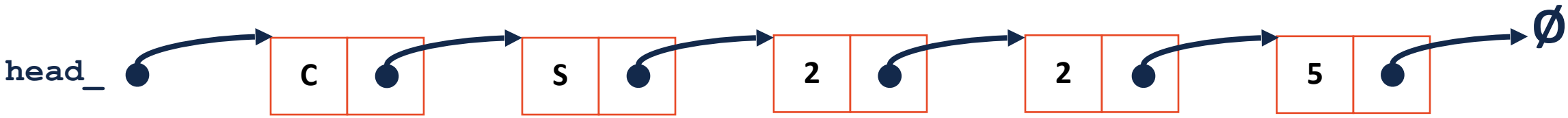


Linked List: `remove(ListNode *& n)`

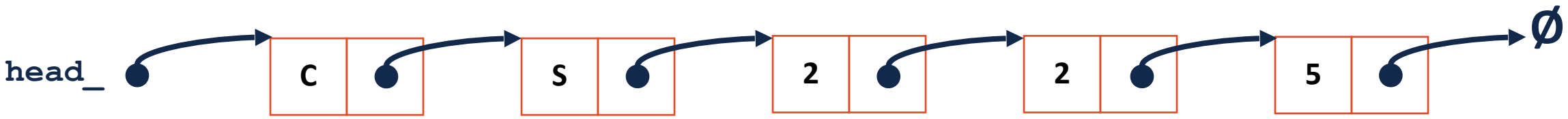


```
103 template <typename T>
104 T List<T>::remove(ListNode *& node) {
105
106
107
108
109
110
111
112 }
```

Linked List: `remove(T & data)`



Linked List: `remove`

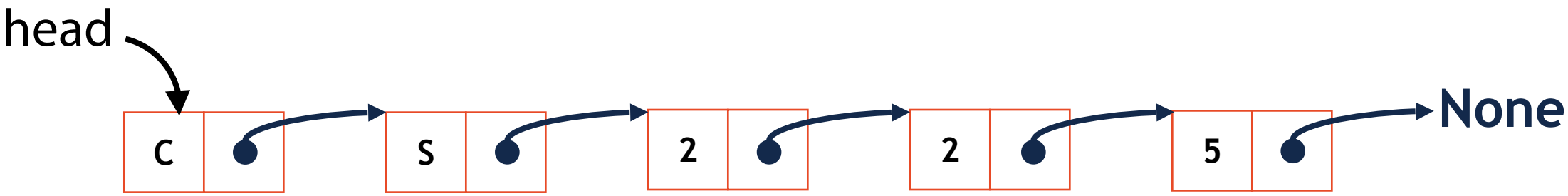


Running time for **`remove(ListNode *&)`**

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

List Implementations

1. Linked List



2. Array List