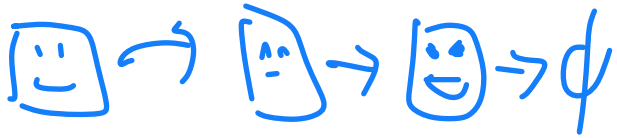
Data Structures

Linked Lists

CS 225

Brad Solomon

January 28, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Announcements

↳ If you have just joined, sign up to take exam 0

↳ If you cannot, email cs225admin!

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**

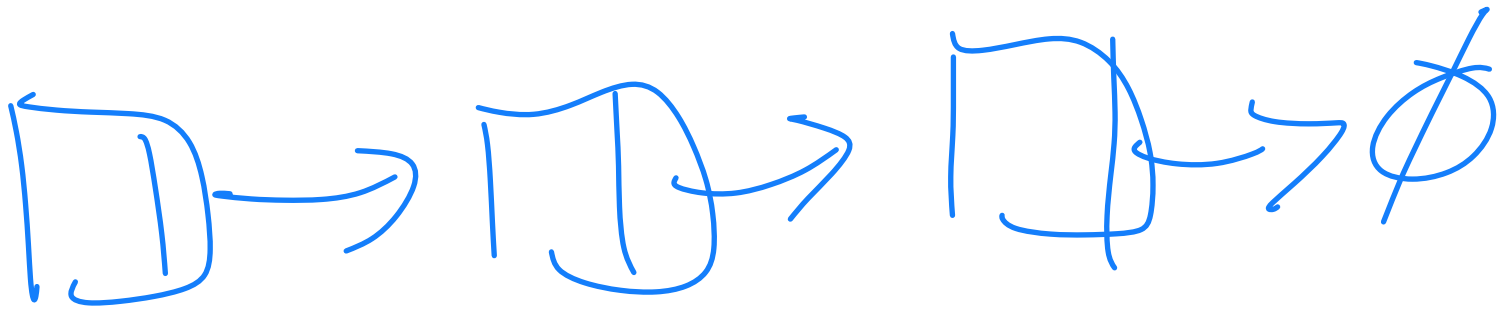
The list can be of a **fixed size** or is **resizable**

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

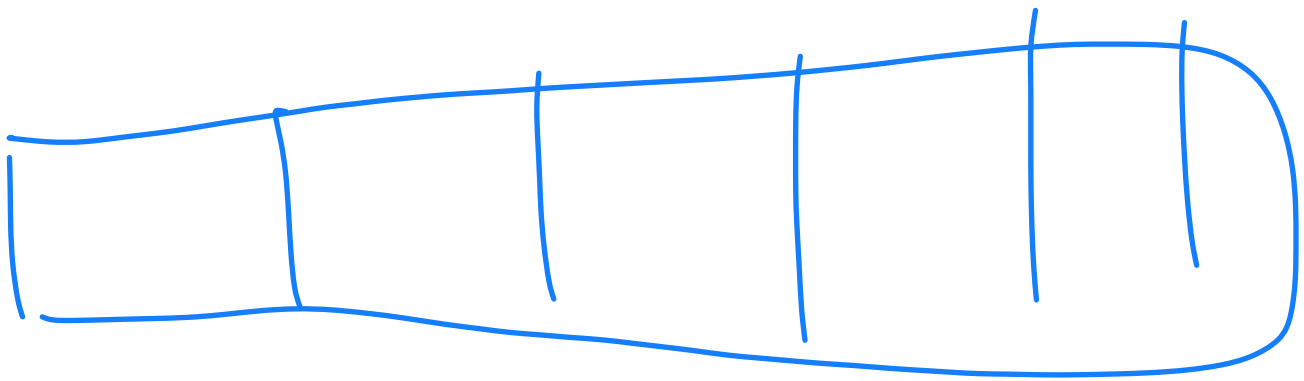
1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations (How)

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**?

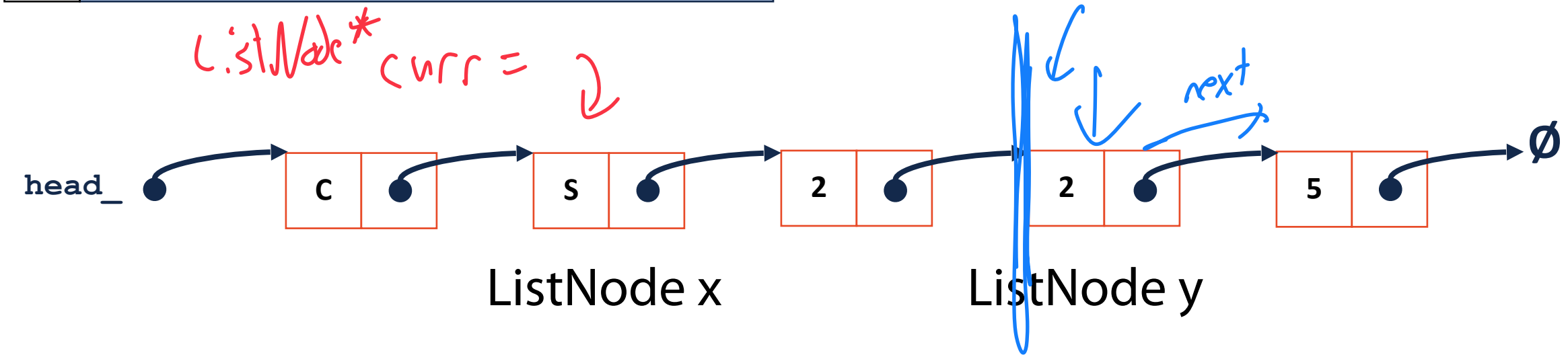
↳ 10% Yes

↳ 90% No

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

↳ Yes! Yis curr->next->next

ListNode* curr = ?



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
```

What is missing in this code?

↳ Templated

List.h

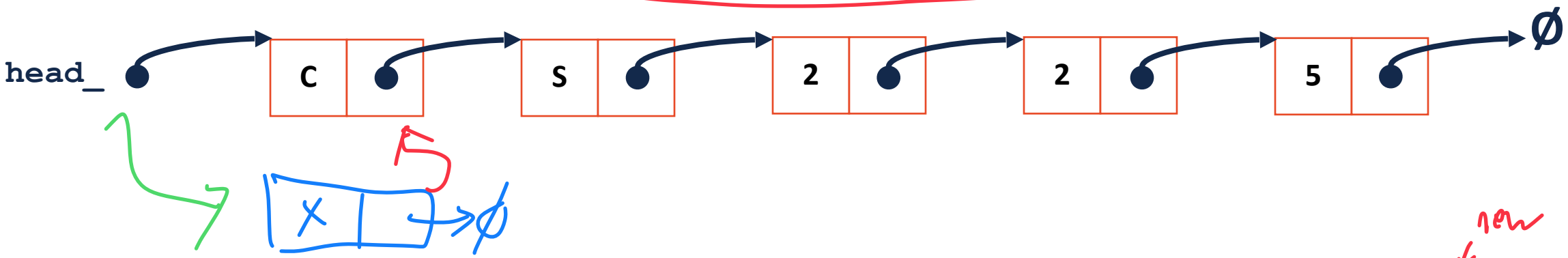
```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
…      /* ... */
28   void insertAtFront(const T& t);
29
30   private:
31     class ListNode {
32       T & data;
33       ListNode * next;
34       ListNode(T & data) :
35         data(data), next(NULL) { }
36     };
37
38 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

List.hpp

Templates go in .hpp

```
1
2
3
4  void List<T>::insertAtFront(const T& t)
5  {
6    ↳ Templated code not used unless
7        called
8
9    ↳ will generate the specific types
10       as needed
11
12
13
14
15
16
17
18
19
20
21
22 }
```

Linked List: `insertAtFront(data)`

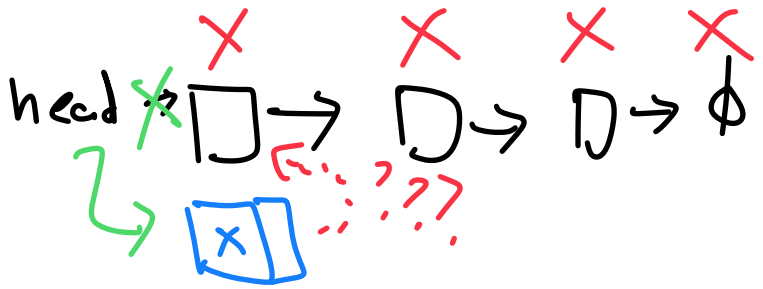


1) Make a new ListNode* tmp
   - data = x
   - next = nullptr

2) Set tmp->next = head_

3) Set head_ to be tmp

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



List.h

```
#pragma once

template <typename T>
class List {
  public:
    /* ... */
  private:
    class ListNode {
      T & data;
      ListNode * next;
      ListNode(T & data) :
       data(data), next(NULL) { }
    };

    ListNode *head_;

    /* ... */

};


#include "List.hpp"
```

List.hpp

```
template <typename T>
void List<T>::insertAtFront(const T& t)
{

  ListNode *tmp = new ListNode(t);

  tmp->next = head_;

  head_ = tmp;

}
```

O(1)

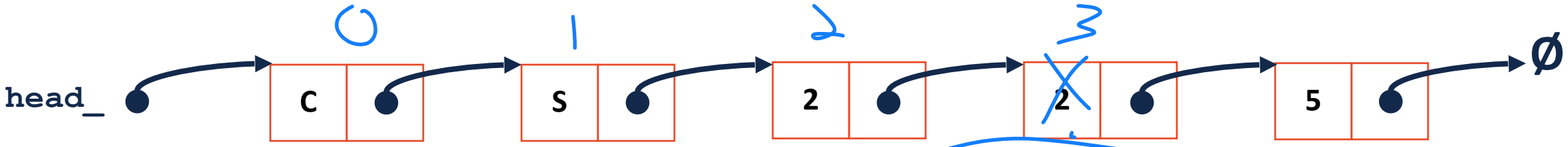
O(1)

O(1)

```
List myL = new List();
myL.insertAtFront();
```

Linked List: `insert(data, index)`

insert(X, 3)



Replacement

Put X in data@3

Not what we will do!

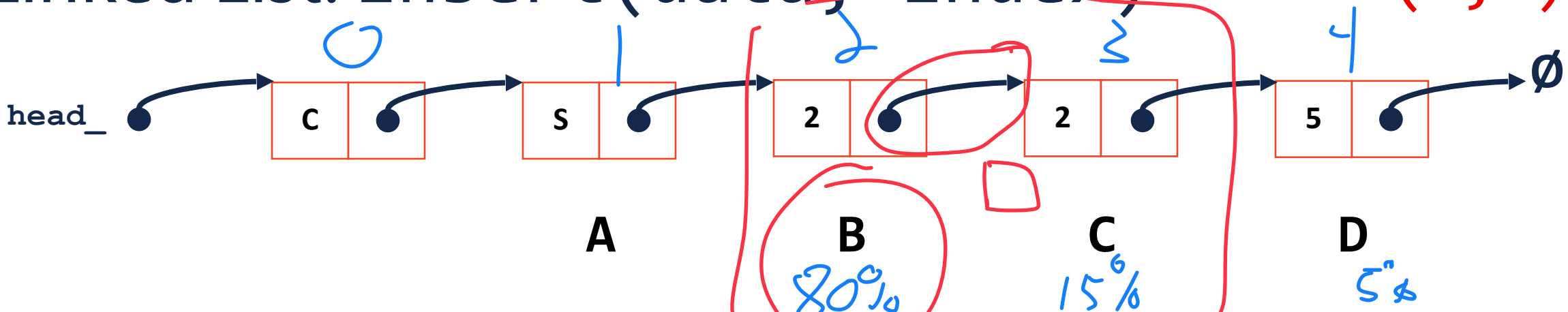
Insert

0 1 2 3

C → S → 2 → X → 2 → 5

Shifted over the rest

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

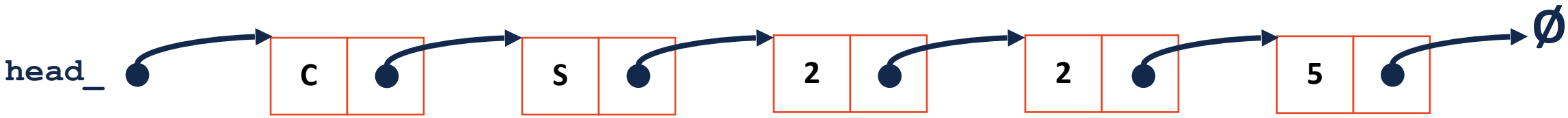


To insert a new ListNode at index 3, we need to **modify** which node?

↳ To add new item at index 3, we need B->next

Join Code: 225

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



1) Get access to node @ position index - 1

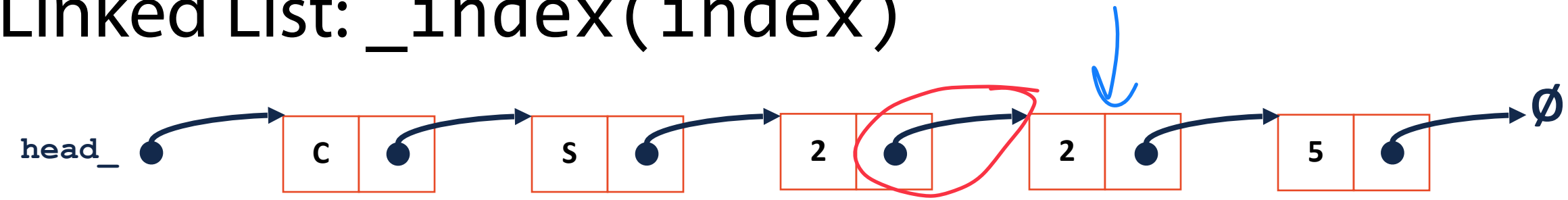
We ***could*** code up a solution to insert which uses some previous var

But lets be smarter!

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

↳ Insert() ↳ Remove() ↳ Find()

Linked List: _index(index)

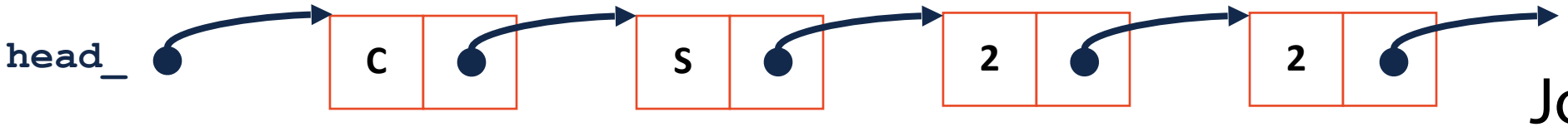


Lets write one function which is useful for insert / remove AND find

1) It must have access to previous nodes next (index - 1)

2) When I call find(3), want node @ 3

Linked List: _index(index)



Join Code: 225

What should the return type of _index() be?

[template <class T>]

(A) T &

10%

(B) ListNode

5%

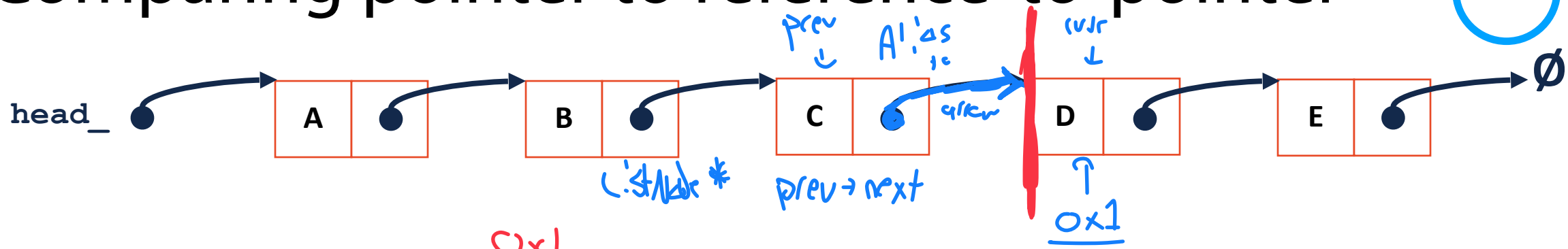
(C) ListNode *

65%

(D) ListNode *&

20%

Comparing pointer to reference-to-pointer



0x1

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

↳ curr → data

↳ curr → next

0x1

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

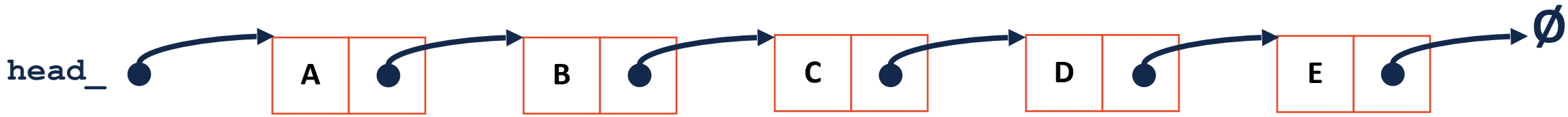
1) I have pointer to curr

↳ curr → data ↳ curr → next

B/c I am reference I can modify this value!

curr = new ListNode!

Comparing pointer to reference-to-pointer



Why can't we dereference curr to make changes?

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

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

reference to a pointer

& * also works

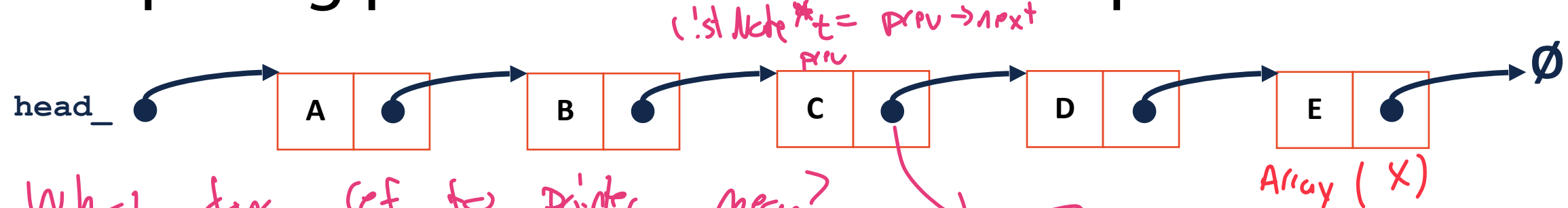
```
ListNode *& curr = _index(3);
```

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

& * ← gibberish

curr == previous.next

Comparing pointer to reference-to-pointer



What does ref to pointer mean?

```
ListNode * curr = _index(3);
```
(0x1)

```
curr = new ListNode(x);
```
← 0x5

This does not modify my list!

int x = 7

x = 5

```
ListNode *& curr = _index(3);
```
← returning prev->next (0x1)

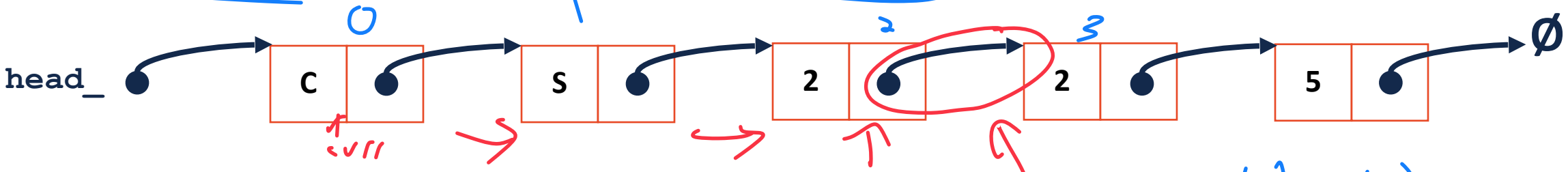
```
curr = new ListNode(x);
```

This does modify list

prev->next = 0x5

LinkedList *& _index(index)

index - 1 times




```
{
    ListNode * curr = head_;

    for (unsigned i = 0; i < index-1; i++){

        curr = curr->next;

    }

    return curr->next;
}
```

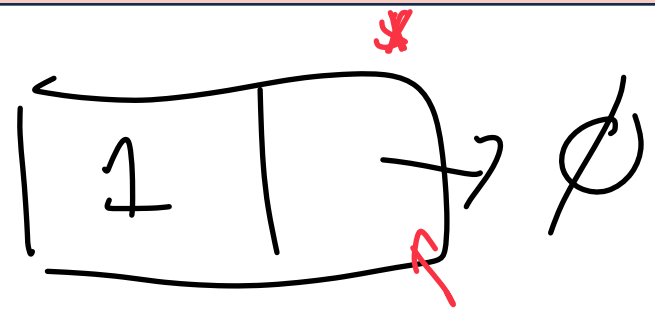
_index(3)

ref to pointer

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 }
```

maybe add edge checker so we don't nullptr->next



main.cpp return curr.next

1

curr.next has value nullptr but does exist in memory

t = L._index(1)

t = new ListNode();

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                  Try on our to implement recursively
68
69
70
71          Base case
72
73
74
75
76          recursive step
77
78          combining step
79
80  }
```

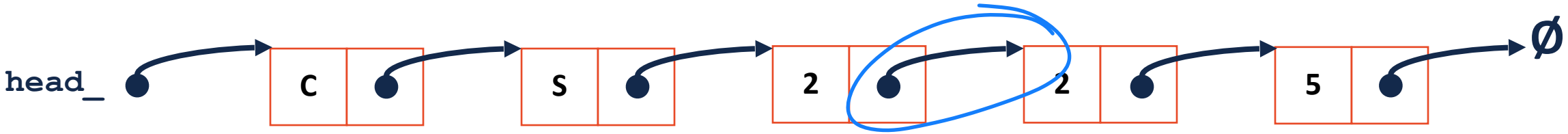
A brief tangent…

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  }
```

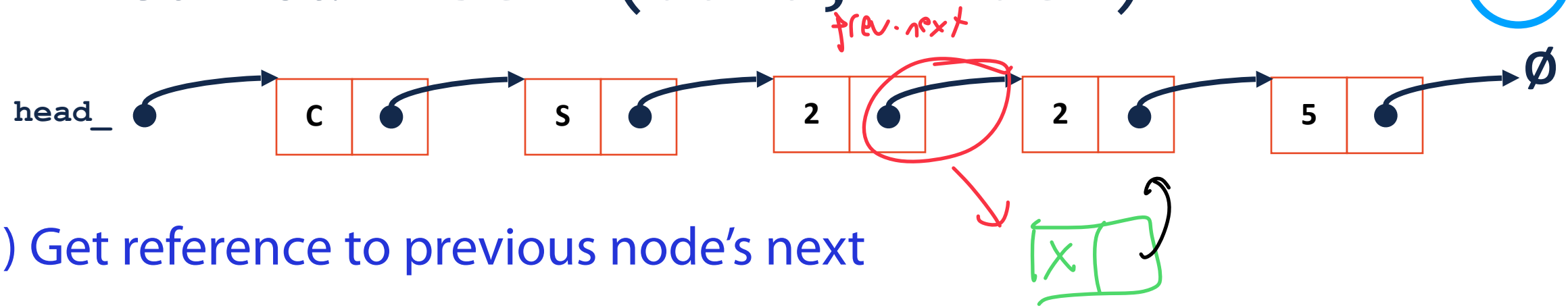
Linked List: `insert(data, index)`



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
```

Ref to a pointr

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[ ]?

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 &`

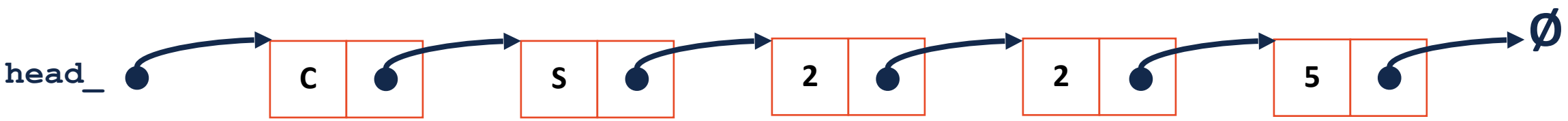
(B) `ListNode`

(C) `ListNode *`

(D) `ListNode *&`

List.hpp

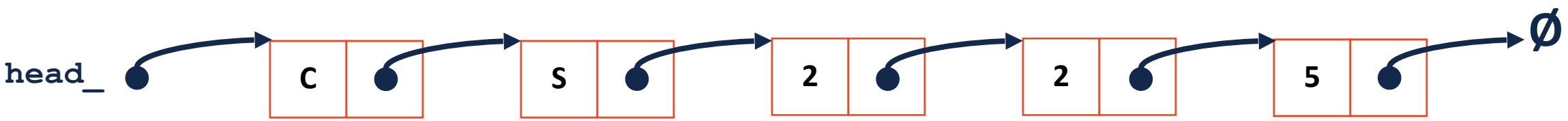
```
48 template <typename T>
49 T & List<T>::operator[](unsigned index) {
50
51
52
53
54
55
56
57
58 }
```



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 }
```

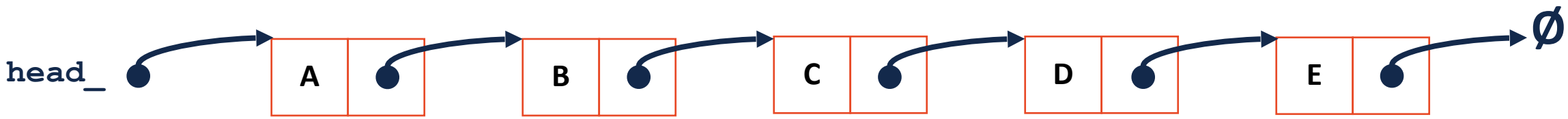
Join Code: 225



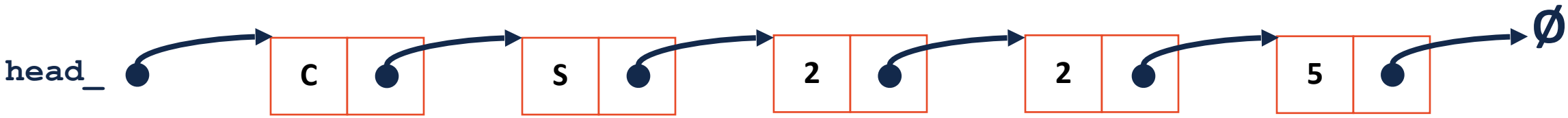
What is the Big O of random access?

Linked List: `remove(<parameters>)`

What input parameters make sense for remove?

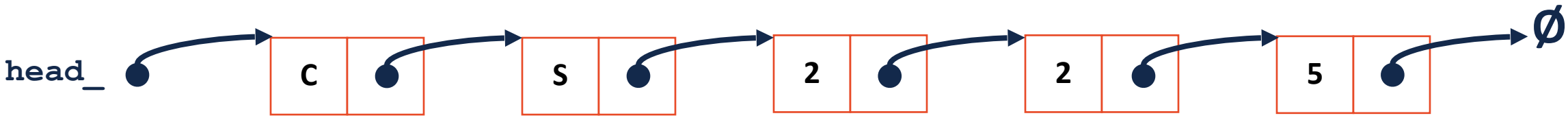


Linked List: `remove(ListNode *& n)`

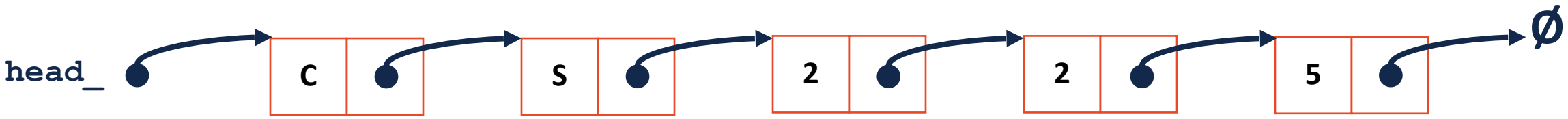


```
103 template <typename T>
104 T List<T>::remove(ListNode *& node) {
105
106
107
108
109
110
111
112 }
```

Linked List: `remove(T & data)`



Linked List: `remove`

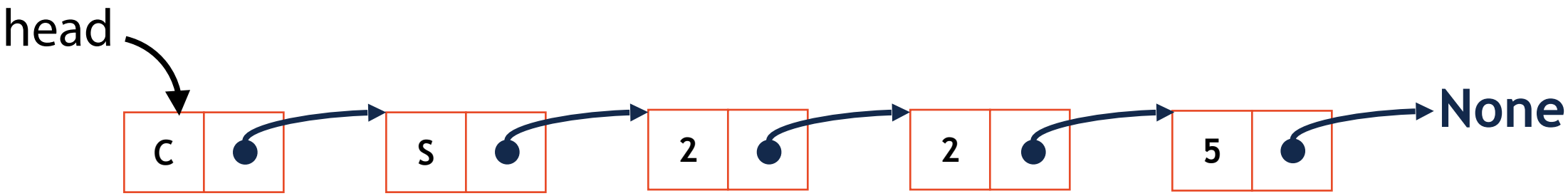


Running time for **`remove(ListNode *&)`**

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

List Implementations

1. Linked List



2. Array List