

Data Structures

Linked Lists

CS 225

August 31, 2026

Brad Solomon



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

List Implementations

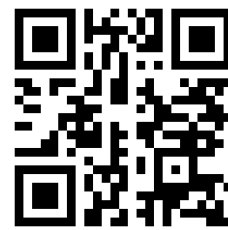
1. Linked List

2. Array List

List.h

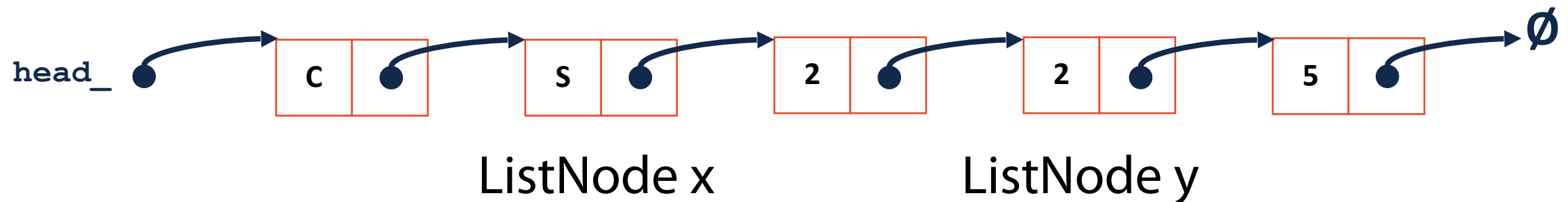
```
1  template <class T>
2  class List {
3  public:
4      /* ... */
5  private:
6      ...
7      class ListNode {
8          ...
9          T & data;
10         ListNode * next;
11         ListNode(T & data) :
12             data(data), next(NULL) { }
13     };
14
15     ListNode *head_;
16 };
17
```

Join Code: 225



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

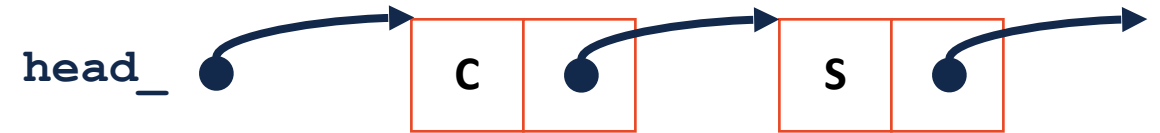
Can we access **y** from **x**?



List.h

```
1  #pragma once
2
3
4  class List {
5      public:
6          /* ... */
7
8      void insertAtFront(const T& t);
9
10     private:
11         class ListNode {
12             T & data;
13             ListNode * next;
14             ListNode(T & data) :
15                 data(data), next(NULL) { }
16         };
17
18         ListNode *head_;
19
20         /* ... */
21
22     };
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```

How do I access list given head_?



List.h

```
1  #pragma once
2
3
4  class List {
5      public:
6          /* ... */
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
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          /* ... */
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```

What is missing in this code?

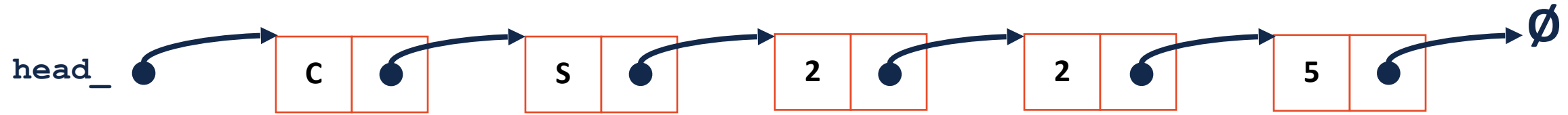
List.h

```
1  #pragma once
2
3  template <typename T>
4  class List {
5      public:
6          /* ... */
7
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         /* ... */
41
...     };
...
79 #include "List.hpp"
```

List.hpp

```
1
2
3
4  void List<T>::insertAtFront(const T& t)
5  {
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22 }
```

Linked List: insertAtFront(data)



List.h

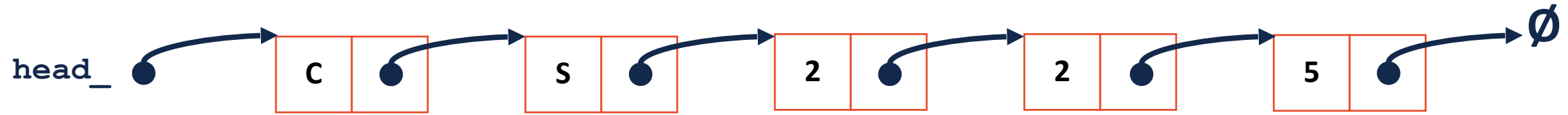
```
1 #pragma once
2
3 template <typename T>
4 class List {
5     public:
6         /* ... */
7
8     private:
9         class ListNode {
10             T & data;
11             ListNode * next;
12             ListNode(T & data) :
13                 data(data), next(NULL) { }
14         };
15
16         ListNode *head_;
17
18         /* ... */
19
20 };
21
22 ...
23
24 ...
25
26 #include "List.hpp"
```

List.hpp

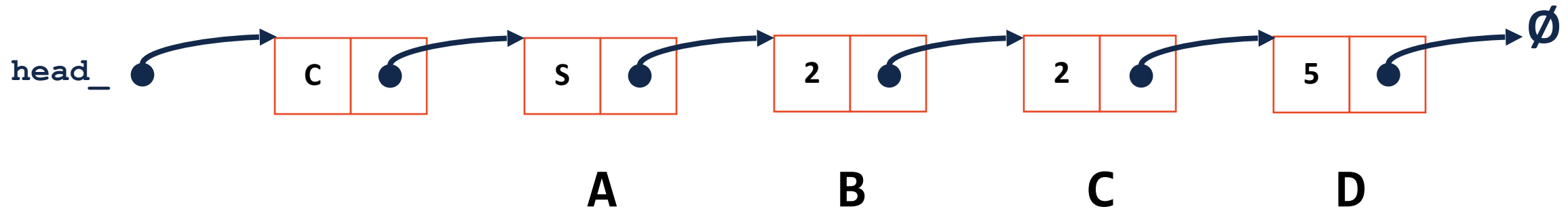


```
1
2
3 template <typename T>
4 void List<T>::insertAtFront(const T& t)
5 {
6
7     ListNode *tmp = new ListNode(t);
8
9
10    tmp->next = head_;
11
12
13    head_ = tmp;
14
15 }
16
17
18
19
20
21
22
```

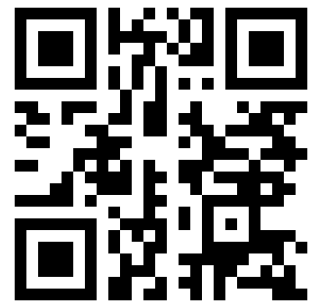
Linked List: insert(data, index)



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

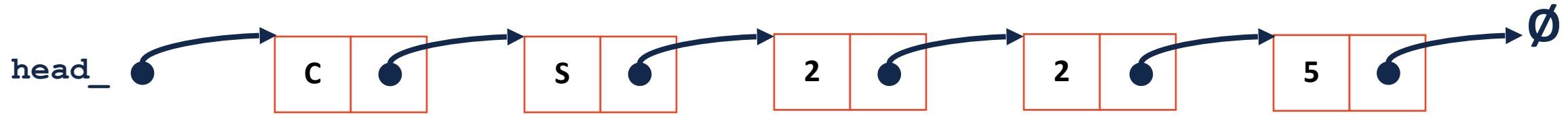


To insert a new ListNode at index 3, we need to **modify** which node?



Join Code: 225

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



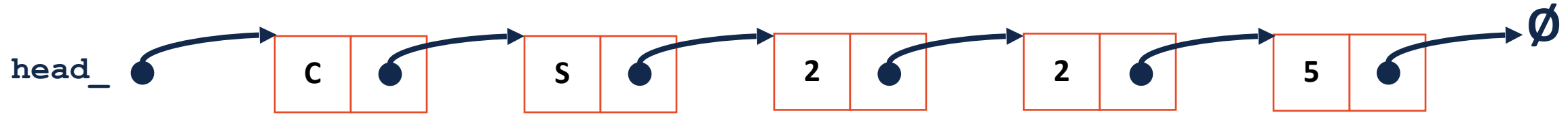
1) Get access to node @ position index - 1

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

But lets be smarter!

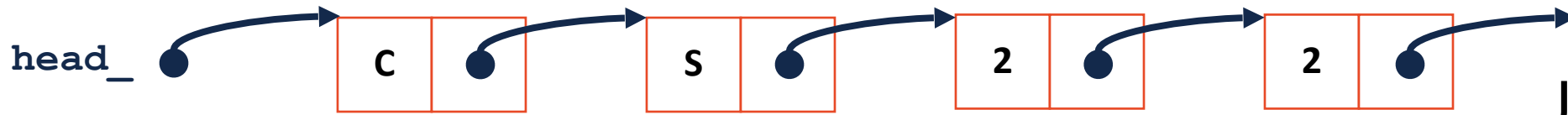
Coding tip from last lecture: **Consider the entire interface**

Linked List: `_index(index)`



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

Linked List: `_index(index)`



Join Code: 225

What should the return type of `_index()` be?

[template <class T>]

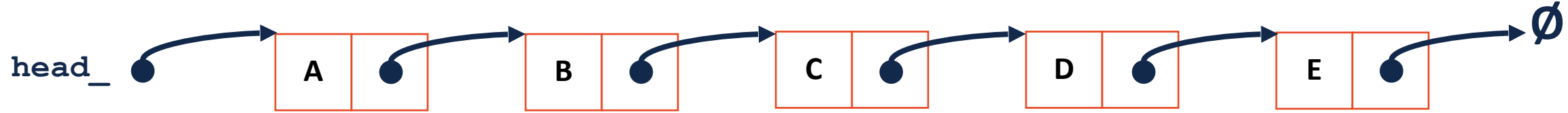
(A) T &

(B) ListNode

(C) ListNode *

(D) ListNode *&

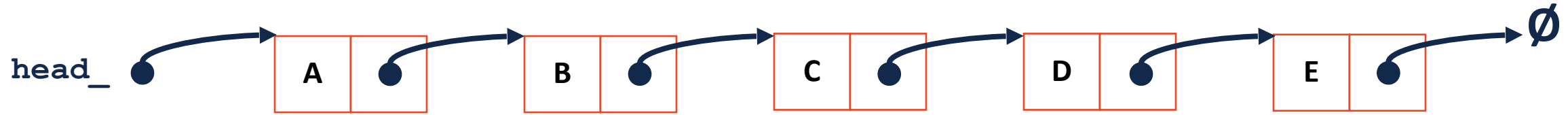
Comparing pointer to reference-to-pointer



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

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

Comparing pointer to reference-to-pointer



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

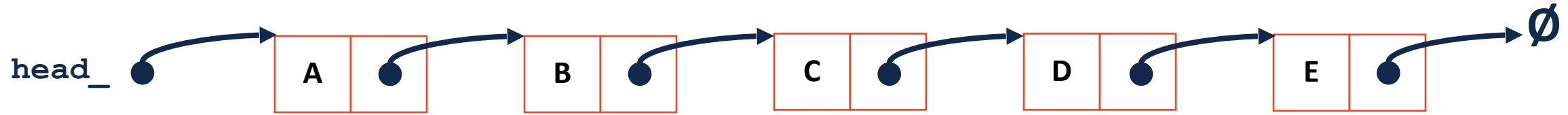
We can access **curr->data** and **curr->next** but not **previous.next**

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

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

```
curr == previous.next
```

Comparing pointer to reference-to-pointer



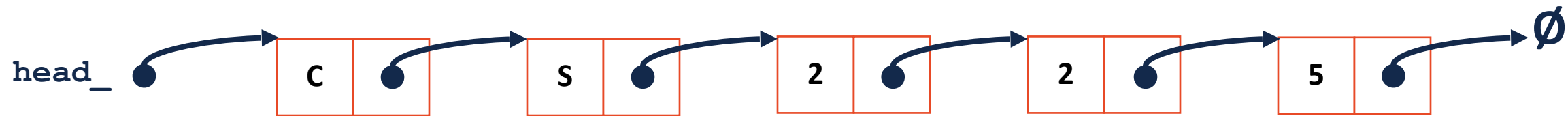
```
ListNode * curr = _index(3);
```

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

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

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

LinkedList *&_index(index)





```
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 }
```

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 }
```

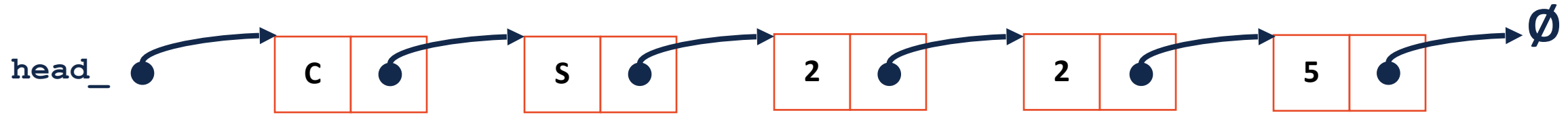
A brief tangent...

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 }
```

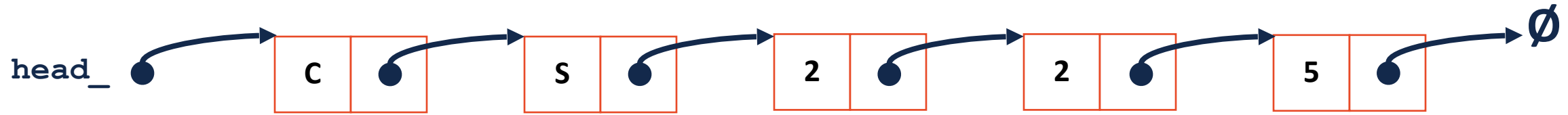
Linked List: insert(data, index)



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...

List.hpp

```
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
```

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

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>]

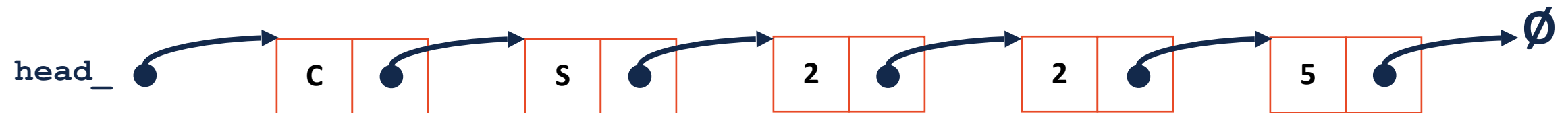
(A) T &

(B) ListNode

(C) ListNode *

(D) ListNode *&

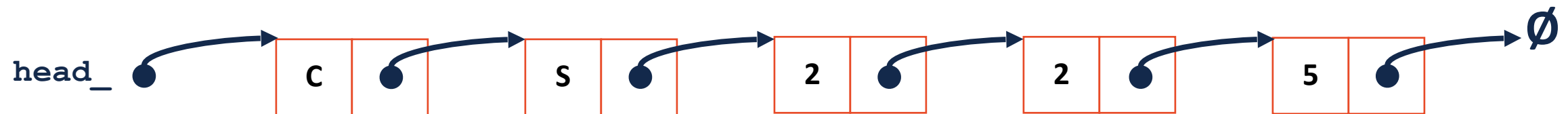
```
48 template <typename T>
49 T & List<T>::operator[] (unsigned index) {
50
51
52
53
54
55
56
57
58 }
```



```
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 }
```



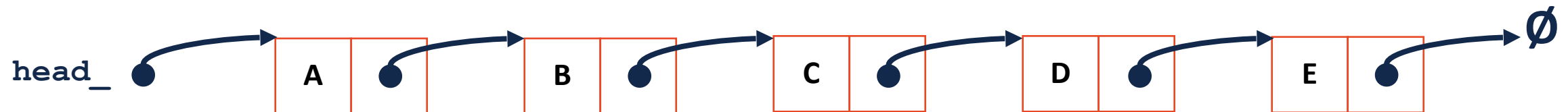
Join Code: 225



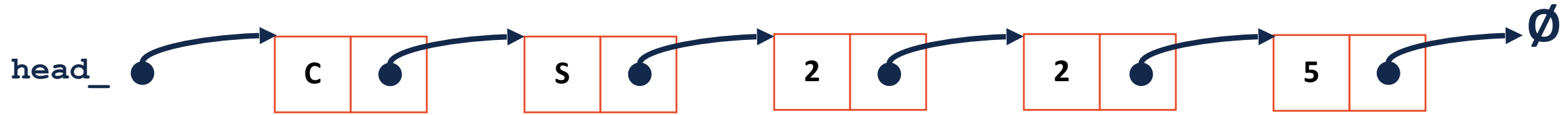
What is the Big O of random access?

Linked List: remove(<parameters>)

What input parameters make sense for remove?

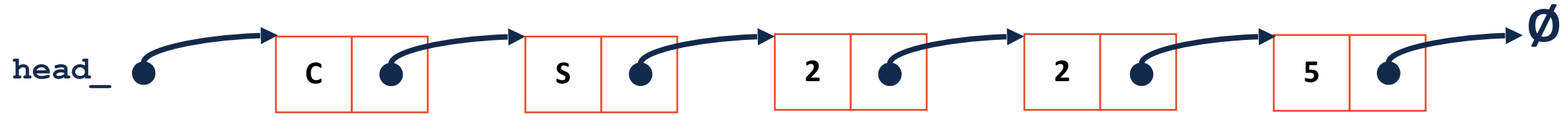


Linked List: remove(ListNode *& n)

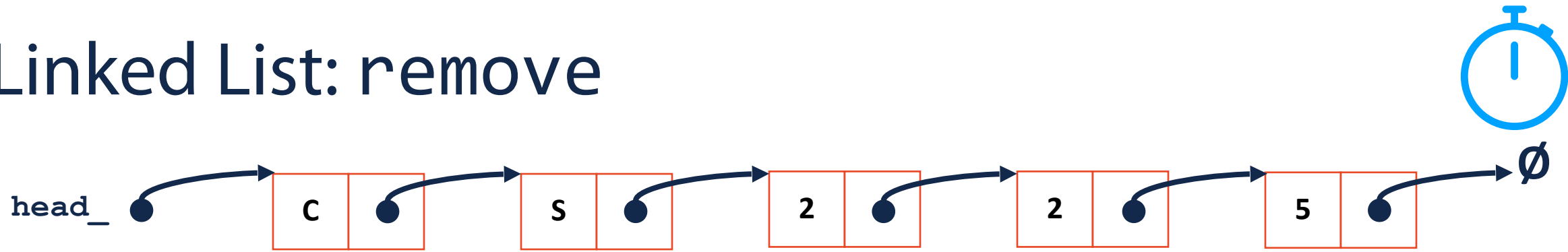


```
103 template <typename T>
104 T List<T>::remove(ListNode *& node) {
105
106
107
108
109
110
111
112 }
```

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

