

Data Structures

Linked Lists & (maybe) Array Lists

CS 225

September 2, 2026

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

BUILD PRODUCTS THAT MATTER!

FALL 2026 RECRUITMENT

Join our 12-week PM fellowship.
No experience needed.

APPLY HERE



APPLY BY

Sept 5

linktr.ee/productspaceui



U. of I.'s premier student-run consulting group

Some of our clients have been:



Microsoft



reddit



Who are we?

U. of I.'s premier student-run consulting firm,
consistently delivering outstanding business
solutions to real-world problems.

Info Nights

Monday, August 24th 6 PM | CIF Room 4025

Monday, August 31st 7 PM | Wohlers Room 141

**Please dress business casual*

Case Training + Meet & Greet

Wednesday, September 2nd, 6 PM | CIF Room 4025

**Please dress business casual*

Women in Consulting Night

Friday, September 4th 5 PM | Wymer 1010

**Please dress smart casual*

Application Deadlines

Cycle 1 - Friday, August 28th @ 11:59 PM

Cycle 2 - Friday, September 4th @ 11:59 PM



@otcr_consulting



**www.recruit.otcr-
consulting.com**

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):

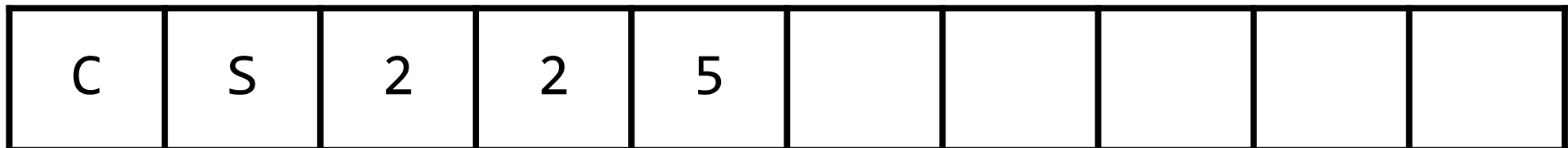
1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations

1. Linked List



2. Array List



Where we left off...



1. Singly linked list only accesses forward —>

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

A review on memory management

A pointer stores a memory address

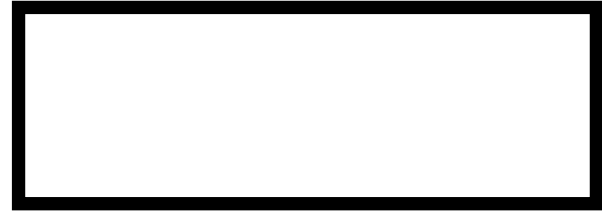
```
int a = 3;
```

```
int *p = &a;
```

a



p



A review on memory management

A reference assigns a new name to a variable

```
int a = 3;
```

$y = a$



```
void blackBox(int & y) {  
    y = 4;  
}
```

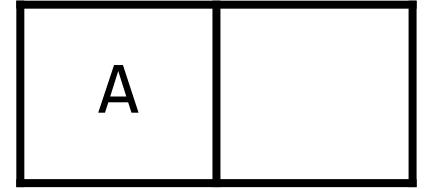
```
blackBox(a) ;
```

A review on memory management

Now lets think in terms of linkedLists...

```
ListNode* a = new ListNode(A) ;
```

$y = a$



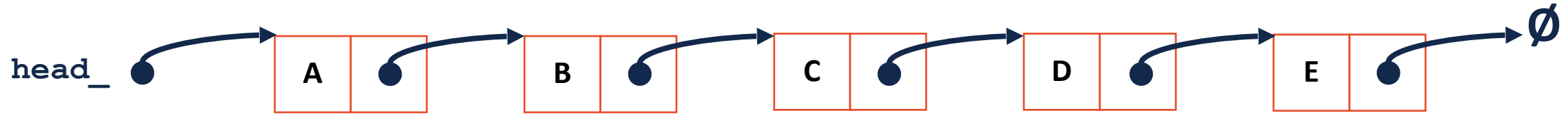
```
void blackBox(ListNode* & y) {  
    y = new ListNode(B) ;  
}
```

```
blackBox(a->next) ;
```

```
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 }
```



Comparing pointer to reference-to-pointer



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

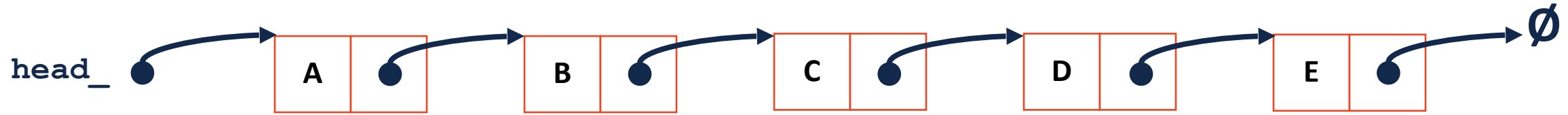
We can access **curr->data** and **curr->next** but not **previous.next**

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

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

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

Comparing pointer to reference-to-pointer



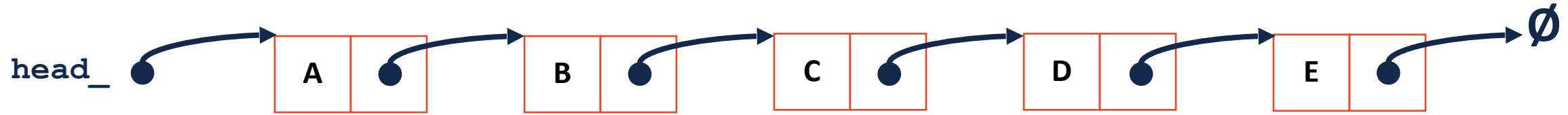
Does this code work?

```
void insertAtNode(ListNode* target, int value) {  
    ListNode* newNode = new ListNode(value);  
    newNode->next = target;  
    target = newNode;  
}
```



Join Code: 225

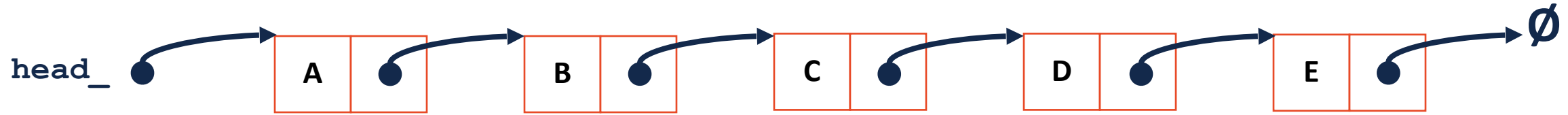
Comparing pointer to reference-to-pointer



This does not work! The pointer is a local stack object!

```
void insertAtNode(ListNode* target, int value) {  
    ListNode* newNode = new ListNode(value);  
    newNode->next = target;  
    target = newNode; // Only updates local copy!  
}
```

Comparing pointer to reference-to-pointer



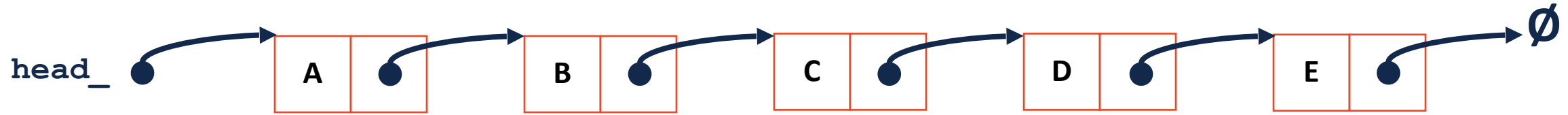
Does this code work?

```
void insertAtNode(ListNode*& target, int value) {  
    ListNode* newNode = new ListNode(value);  
    newNode->next = target;  
    target = newNode;  
}
```



Join Code: 225

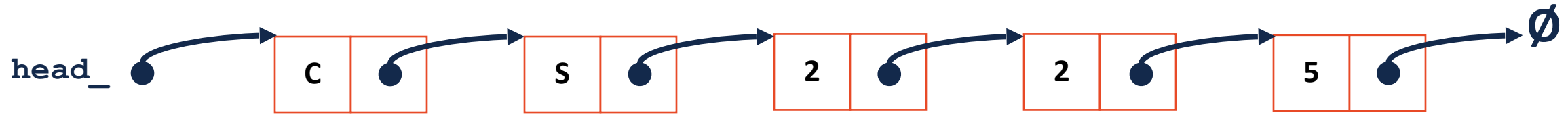
Comparing pointer to reference-to-pointer



Yes — because the reference is to a specific memory allocation

```
void insertAtNode(ListNode*& target, int value) {  
    ListNode* newNode = new ListNode(value);  
    newNode->next = target;  
    target = newNode;  
}
```

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
16     tmp->next = curr;
17
18
19
20     curr = tmp;
21 }
22
```

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

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>]

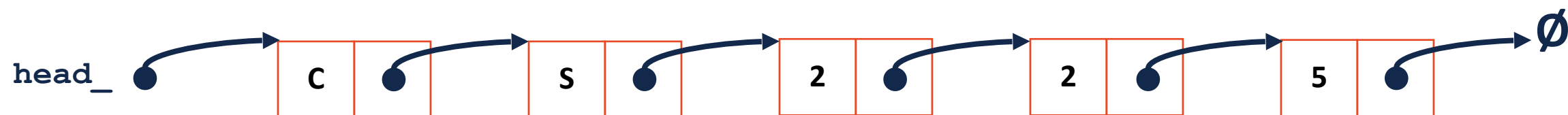
(A) T &

(B) ListNode

(C) ListNode *

(D) ListNode *&

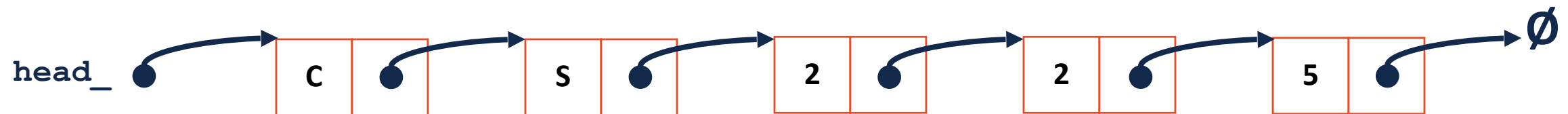
```
48 template <typename T>
49 T & List<T>::operator[] (unsigned index) {
50
51
52
53
54
55
56
57
58 }
```



```
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 }
```



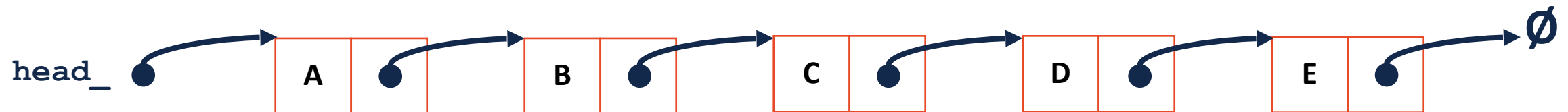
Join Code: 225



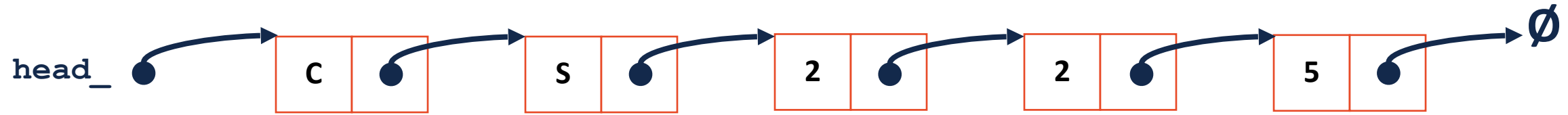
What is the Big O of random access?

Linked List: remove(<parameters>)

What input parameters make sense for remove?

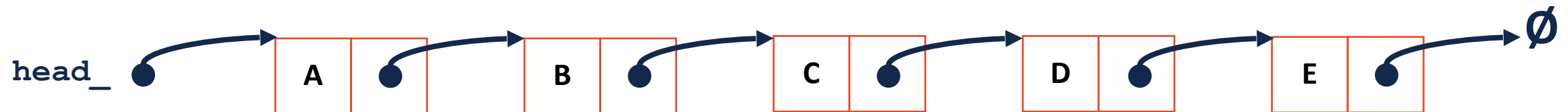


Linked List: remove(ListNode *& n)

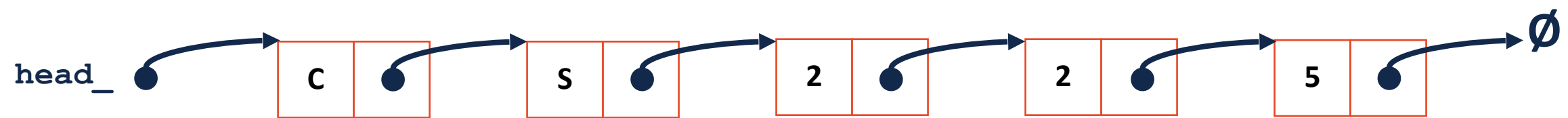




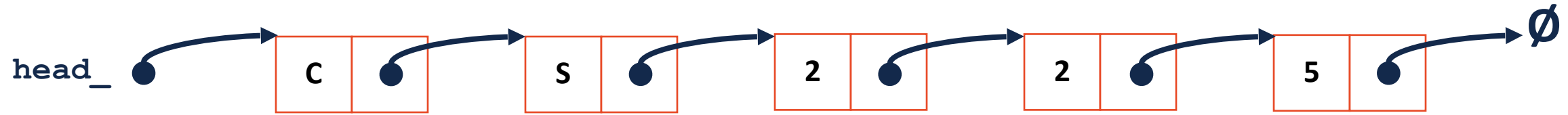
```
103 template <typename T>
104 T List<T>::remove(ListNode *& node) {
105
106     ListNode * temp = node;
107     node = node->next;
108     T data = temp->data;
109     delete temp;
110     return data;
111
112 }
```



Linked List: remove(T & data)



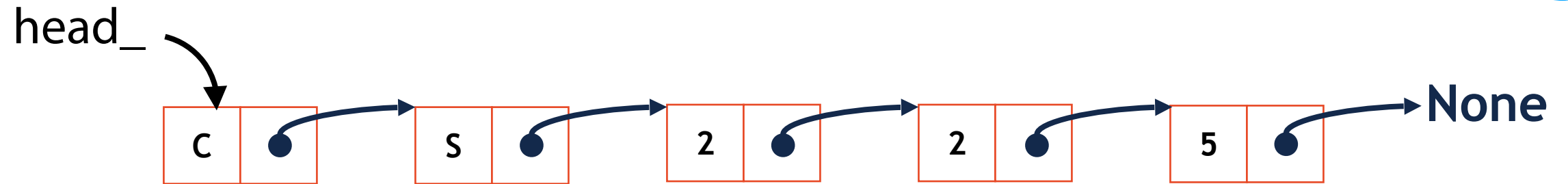
Linked List: remove



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

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

Linked List Runtimes



@Front

@RefPointer

@Index

Insert

Delete

Thinking critically about linked lists...



What is the runtime to filter a list of all odd valued items?

(A) $O(1)$

(B) $O(\log n)$

(C) $O(n)$

(D) $O(n^2)$

Thinking critically about linked lists...

An $O(n)$ runtime seems bad (and is) for finding a single item.

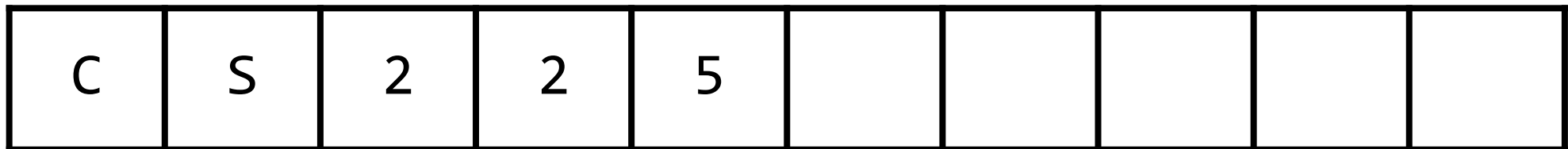
When might you want to use a linked list?

List Implementations

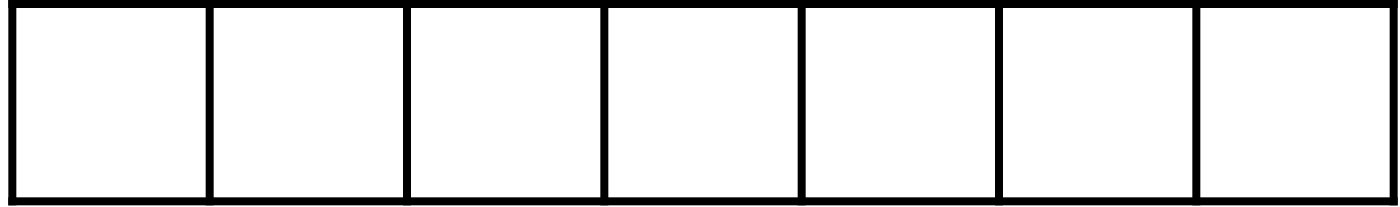
1. Linked List



2. Array List



Array List



An array is allocated as continuous memory.

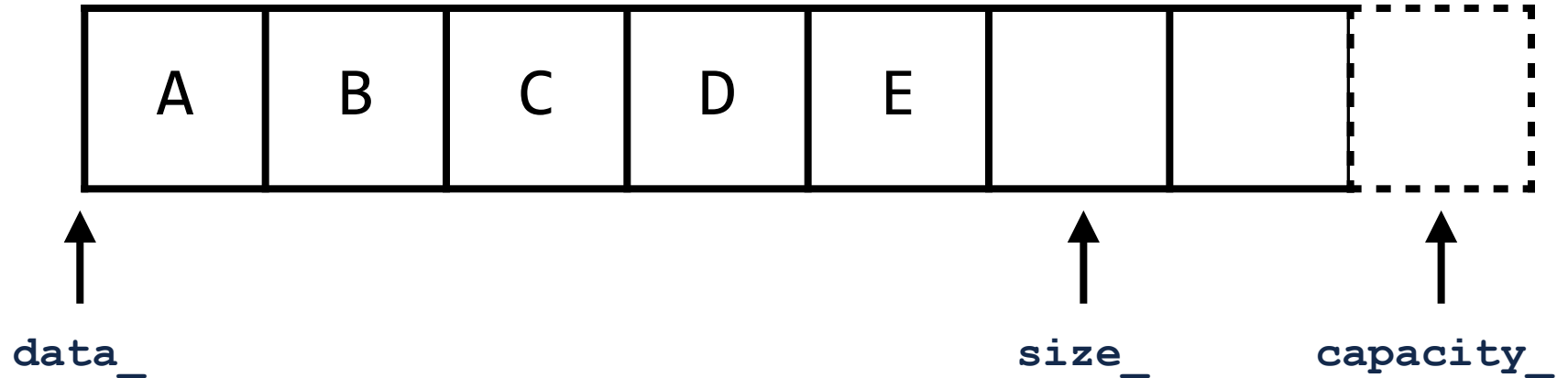
Three values are necessary for efficient array usage:

1)

2)

3)

Array List

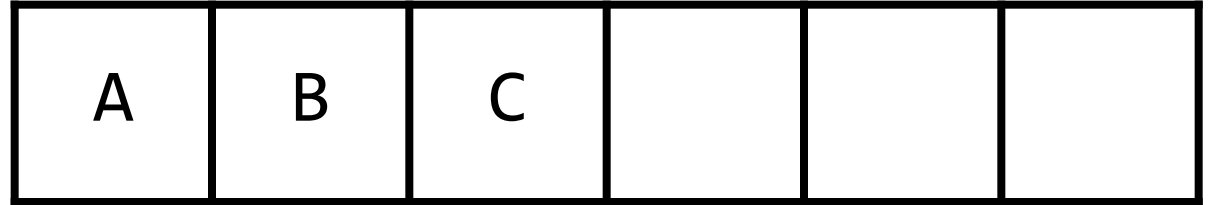


In C++, vector is implemented as:

- 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

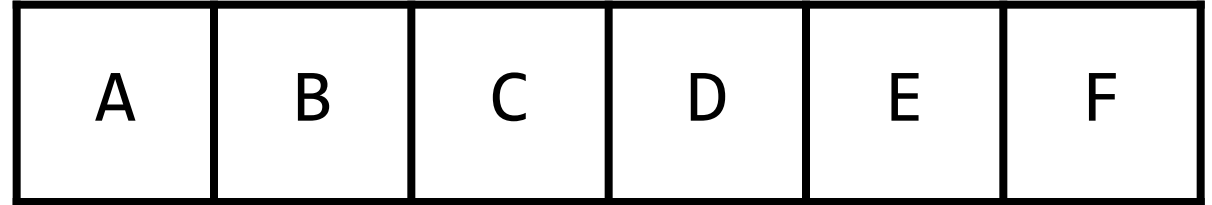
```
1  #pragma once
2
3  template <typename T>
4  class List {
5  public:
6      /* --- */
7  ...
8  private:
9      T *data_;
10
11     T *size_;
12
13     T *capacity_;
14
15     /* --- */
16
17 };
```



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

List.h

```
1  #pragma once
2
3  template <typename T>
4  class List {
5  public:
6      /* --- */
7  ...
8  private:
9      T *data_;
10
11     T *size_;
12
13     T *capacity_;
14
15     ...
16     /* --- */
17 };
```



How do I know if I'm at capacity?

Array List: []

c	s	2	2	5					
---	---	---	---	---	--	--	--	--	--

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
---	---	---	---	---	---	---