

Data Structures

ArrayLists

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**

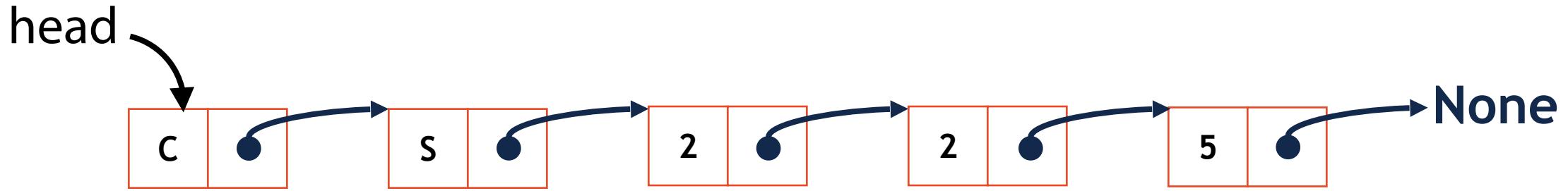
The list can be of a **fixed size** or is **resizable**

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

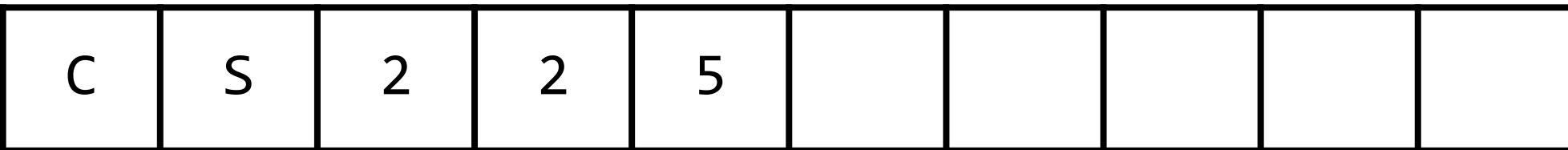
1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations

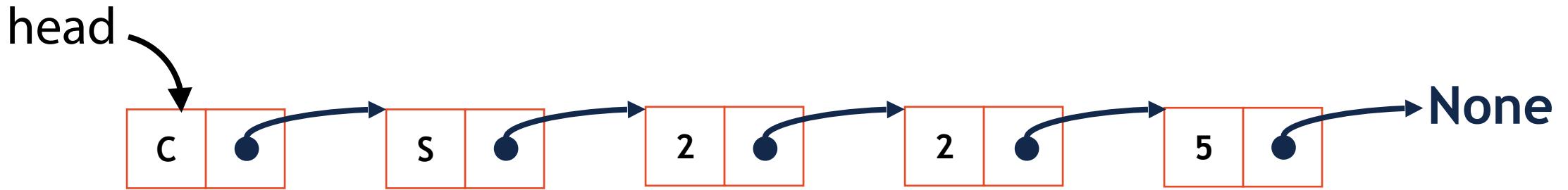
1. Linked List



2. ArrayList



Where we left off...



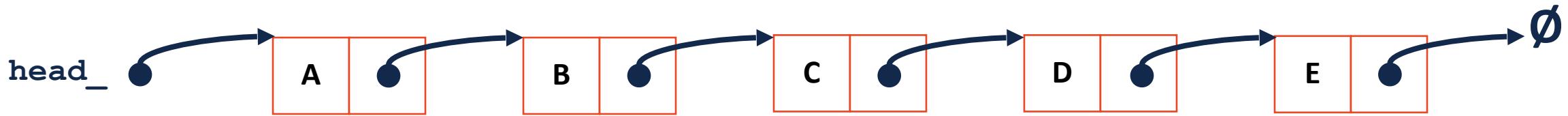
1. Singly linked list only accesses forward —>

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

```
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);
```

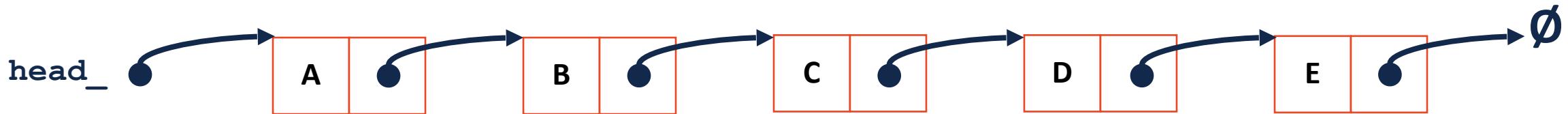
We can access **curr->data** and **curr->next** but not **previous.next**

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

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

curr == previous.next

Comparing pointer to reference-to-pointer



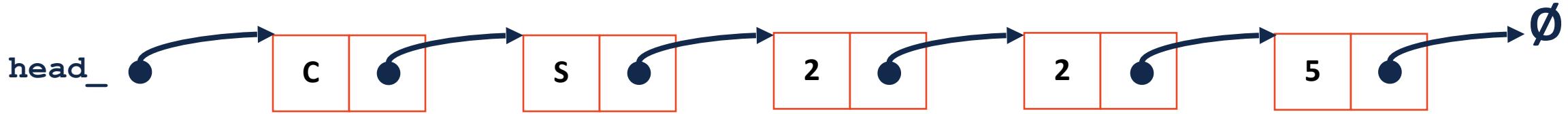
```
ListNode * curr = _index(3);
```

```
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;
```

- 4) Modify the previous node to point to new ListNode

```
curr = tmp;
```

Lets compare...

List.hpp



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

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

What is the Big O of insert?

List.hpp

```
1 template <typename T>
2 void List<T>::insert(const T & data,
3 unsigned index) {
4
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 }
```



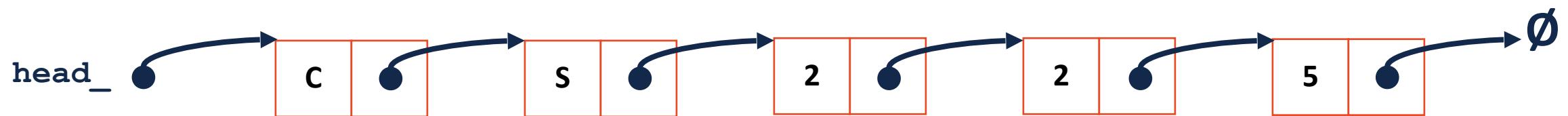
Join Code: 225

List Random Access []

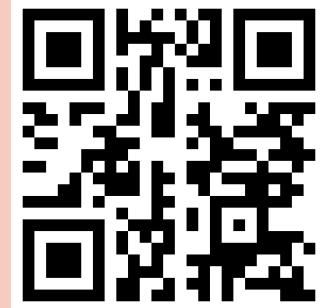
Given a list L, what operations can we do on L[]?

What return type should this function have?

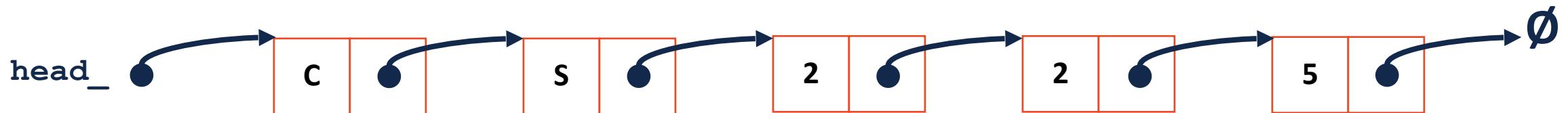
```
48 template <typename T>
49 T & List<T>::operator[] (unsigned index) {
50
51
52
53
54
55
56
57
58 }
```



```
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 }
```



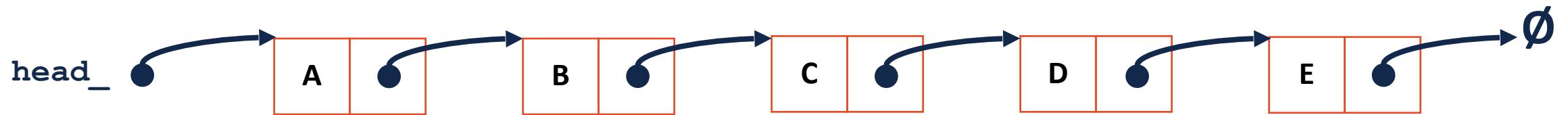
Join Code: 225



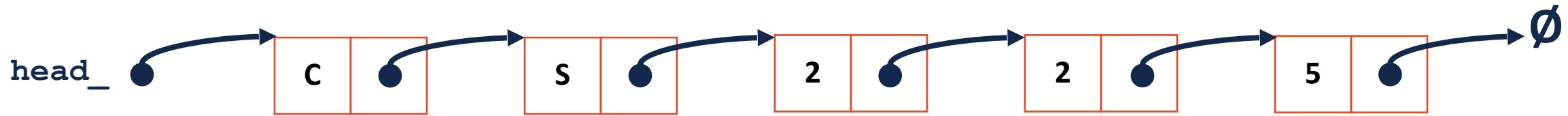
What is the Big O of random access?

Linked List: remove(<parameters>)

What input parameters make sense for remove?

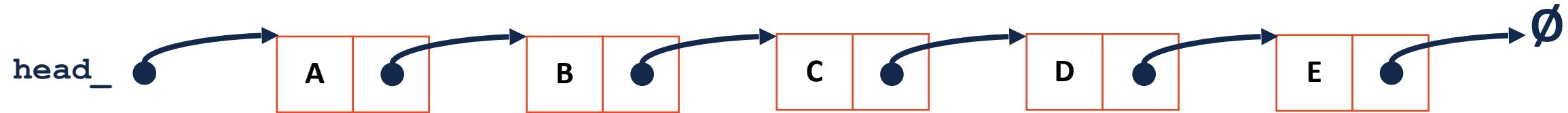


Linked List: remove(ListNode *& n)

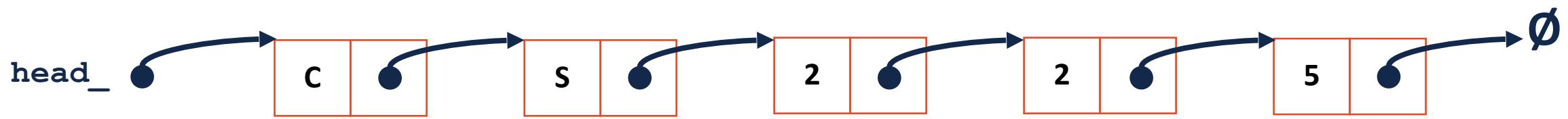




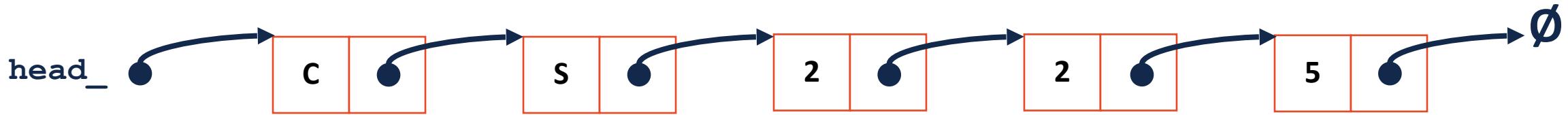
```
103 template <typename T>
104 T List<T>::remove(ListNode *& node) {
105
106     ListNode * temp = node;
107     node = node->next;
108     T data = temp->data;
109     delete temp;
110     return data;
111
112 }
```



Linked List: remove(T & data)



Linked List: remove



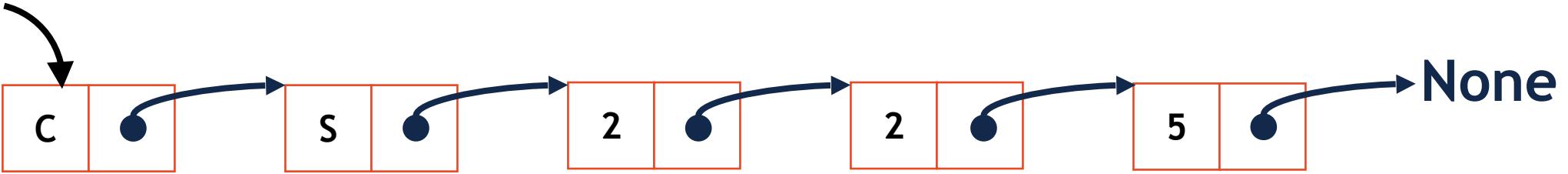
Running time for **remove(ListNode *&)**

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



Linked List Runtimes

head_



@Front

@RefPointer

@Index

Insert

Delete

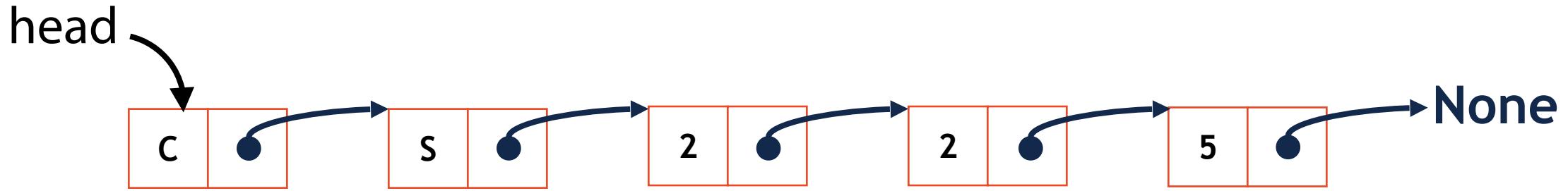
Thinking critically about linked lists...

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

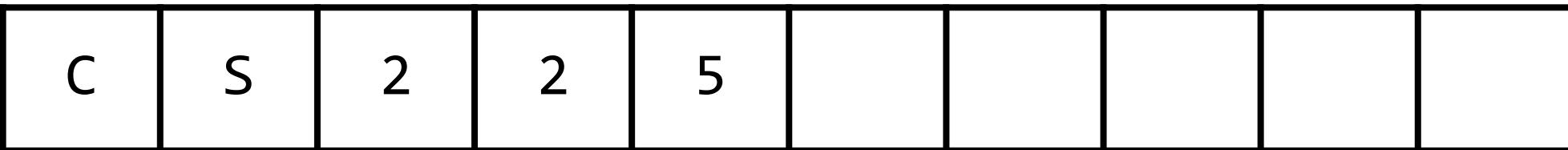
What is the runtime to find an item of interest?

List Implementations

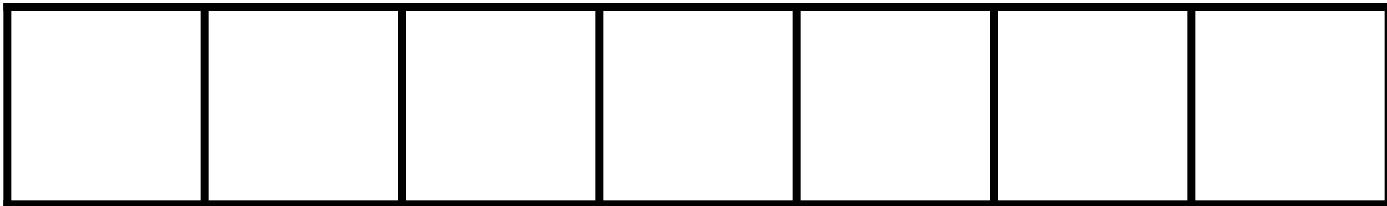
1. Linked List



2. ArrayList



Array List



An array is allocated as continuous memory.

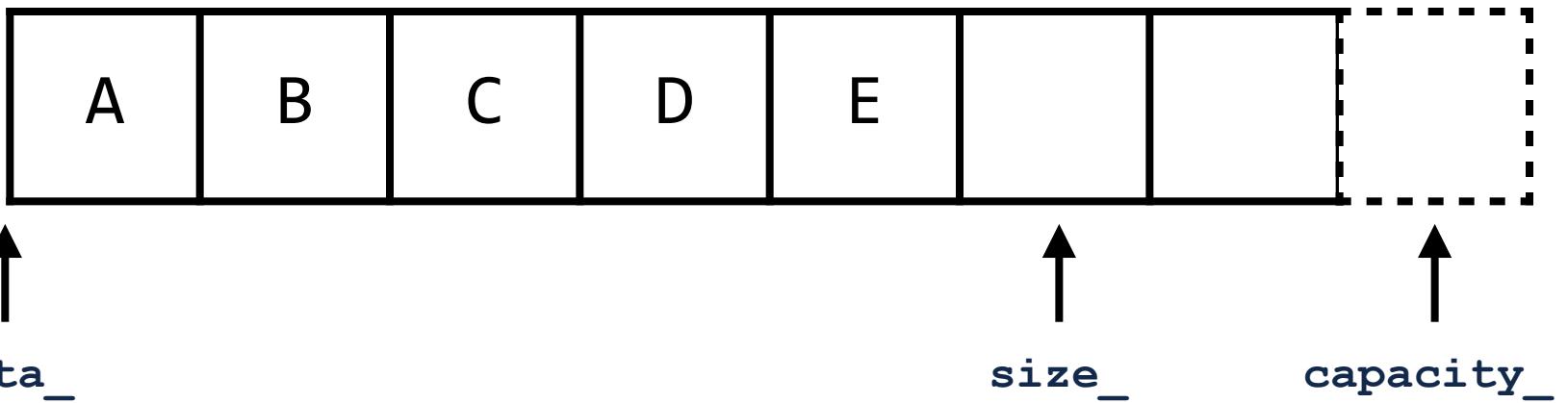
Three values are necessary for efficient array usage:

1)

2)

3)

ArrayList

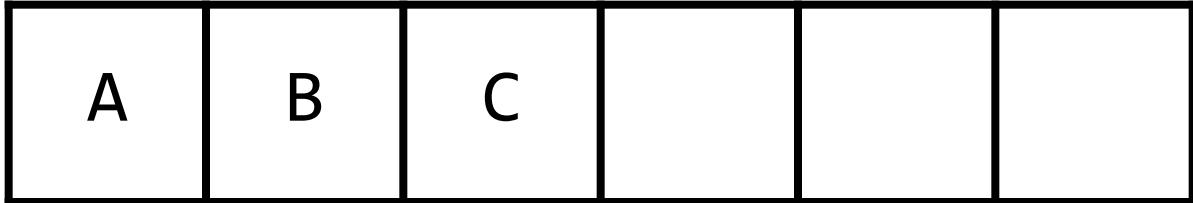


In C++, vector is implemented as:

- 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 private:
7     T *data_;
8
9     T *size_;
10
11    T *capacity_;
...
12    /* --- */
13};
```

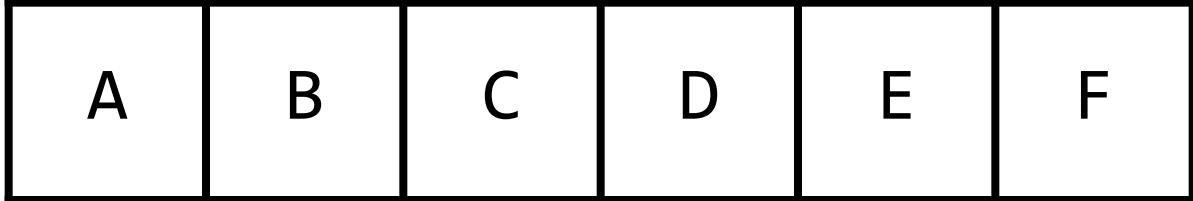


If I want to know the number of items in the array:

List.h

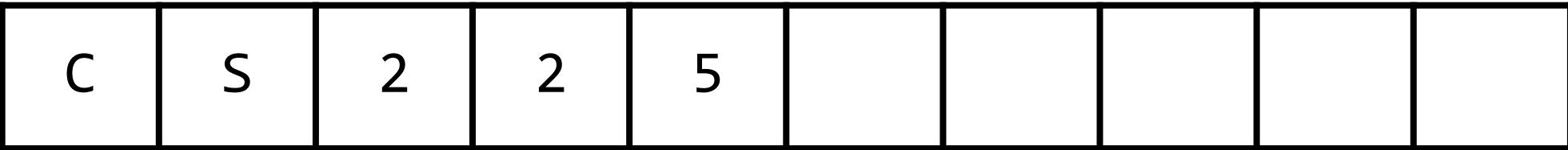


```
1 #pragma once
2
3 template <typename T>
4 class List {
5 public:
6     /* --- */
7 ...
8 private:
9     T *data_;
10
11    T *size_;
12
13    T *capacity_;
14 ...
15    /* --- */
16 };
```

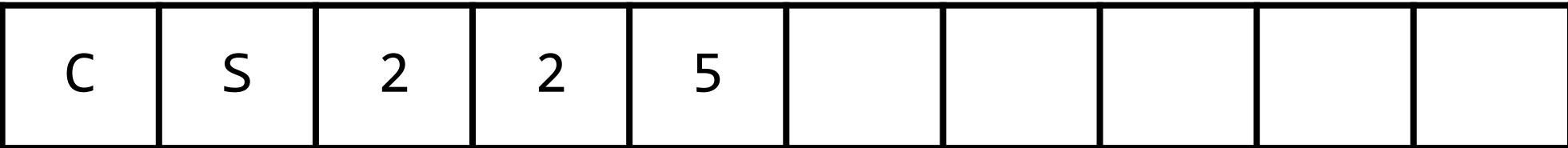


How do I know if I'm at capacity?

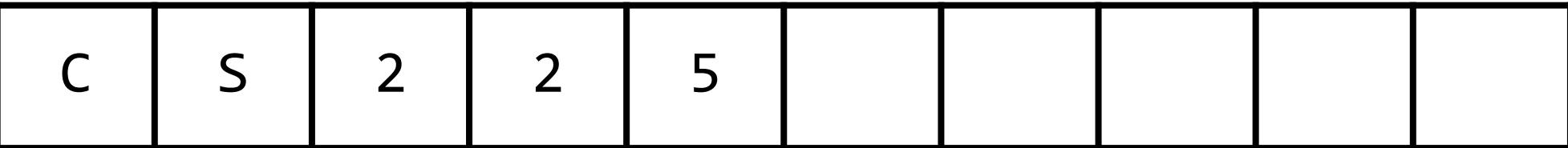
Array List: []



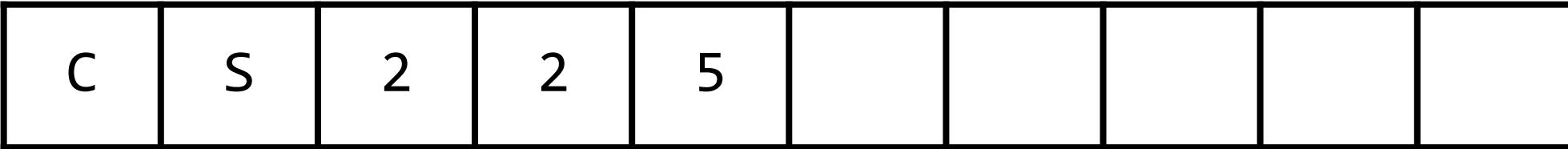
ArrayList: insertFront(data)



ArrayList: insertBack(data)



ArrayList: insert(data, index)



ArrayList: addspace(data)

