Data Structures

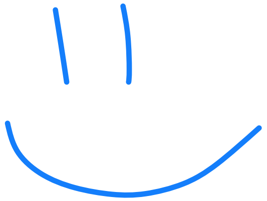
Linked List 3

& Array Lists

CS 225

Brad Solomon

September 6, 2024



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)

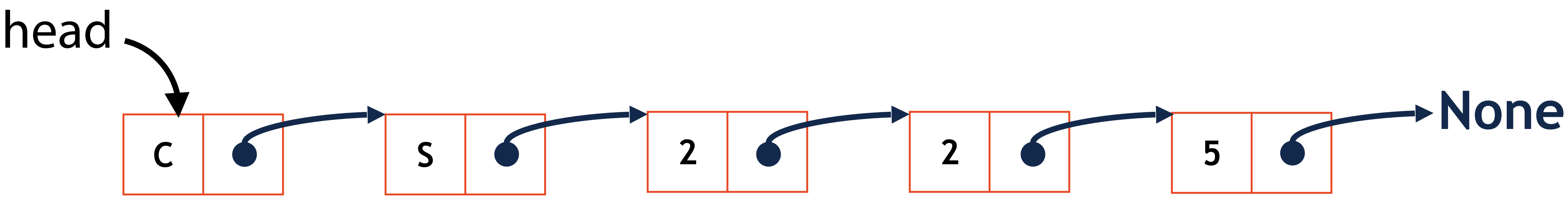
Discuss data variables for implementing array lists

Explore the List ADT (as an array list)

↳ Big O

List Implementations

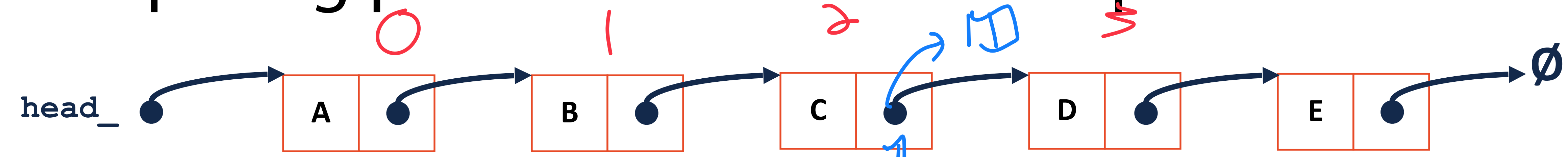
1. Linked List



2. Array List

C	S	2	2	5					

Comparing pointer to reference-to-pointer



2 diff imp, see Wednesday!

Alias to this arrow

↑

← cannot go back

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

↳ Not our implementation

We chose not to do this!

↳ Gives us address to Node @ 3

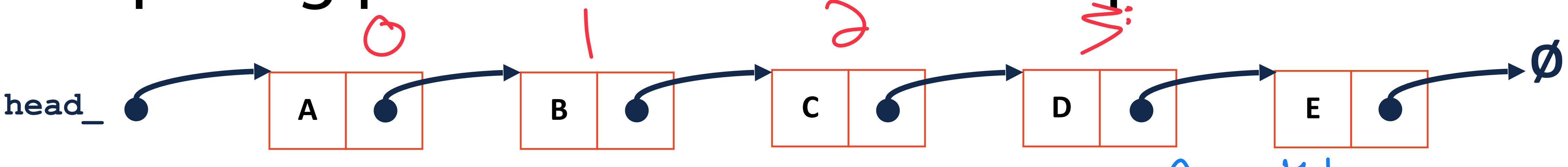
But doesnt let us change what node is 3

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

alias of

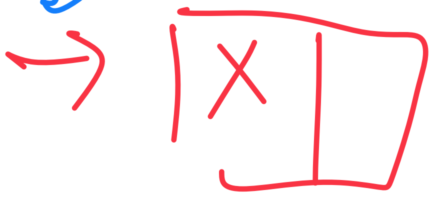
C → next

Comparing pointer to reference-to-pointer



↑ from this
curr

↓ to this



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

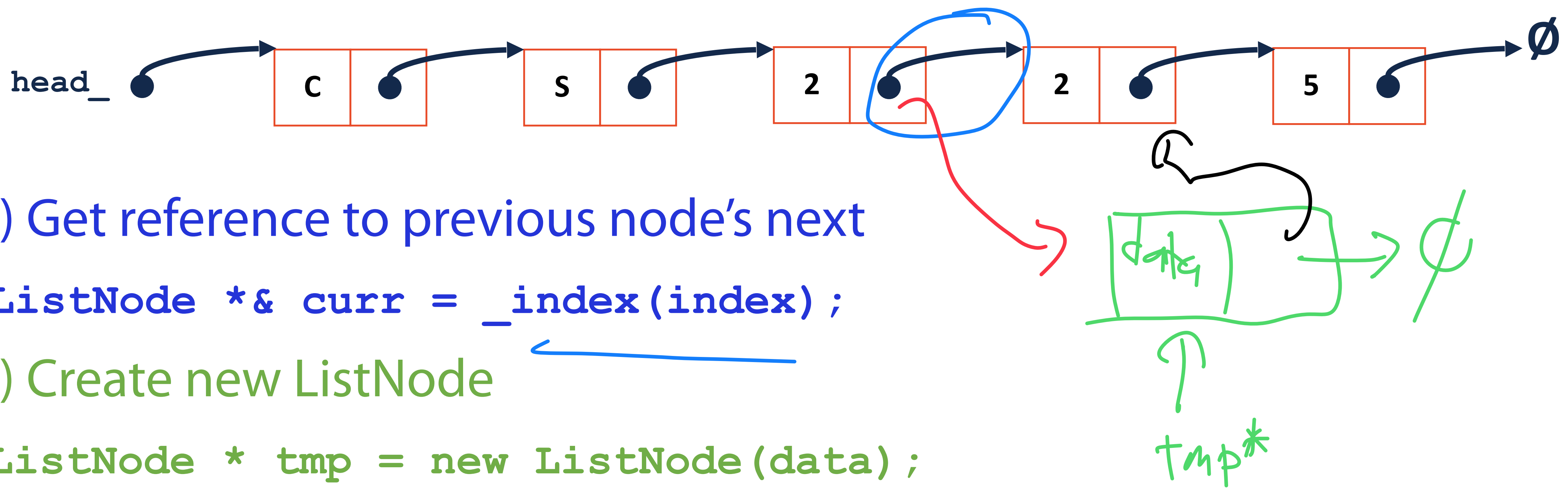
curr = new ListNode(x)

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

alias of

C → next

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

```cpp

template <typename T>
void List<T>::insertAtFront(const T& t)
{
  ListNode *tmp = new ListNode(t);

  tmp->next = head_;

  head_ = tmp;

}
```

```cpp

template <typename T>
void List<T>::insert(const T & data,
unsigned index) {



  ListNode *& curr = _index(index);



  ListNode * tmp = new ListNode(data);



  tmp->next = curr;



  curr = tmp;
}
```

List Random Access [ ]

L[7] → item @ 1

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

↳ Get Value

↳ Set (or change value)

What return type should this function have?

↳ pointer would work | reference will work

List.hpp

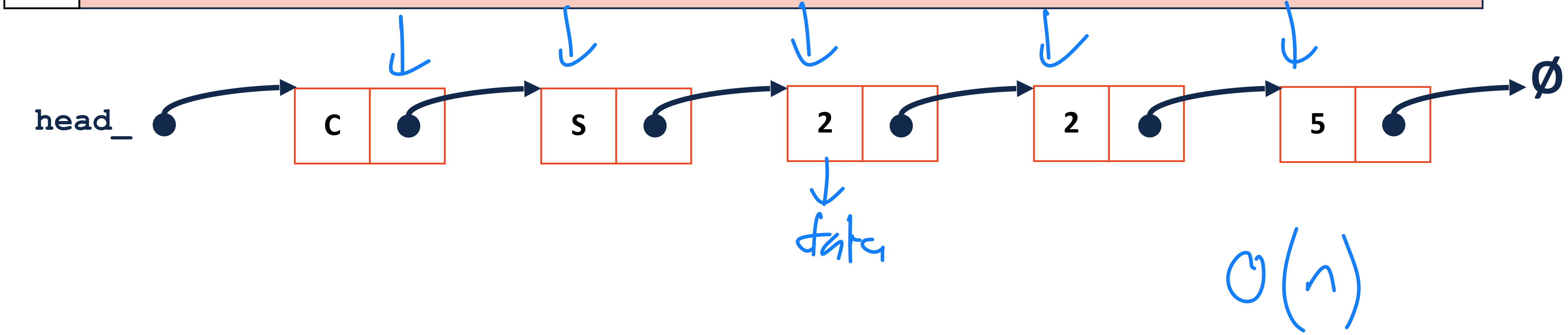
```
template <typename T>
T & List<T>::operator[](unsigned index) {

    ListNode *& val = _index(index);

    return val->data;

}
```

O(n)



head_

C

S

2

2

5

Ø

data

O(n)

List.hpp

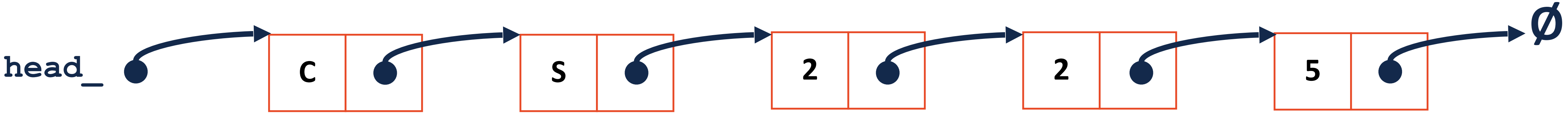
```
template <typename T>
T & List<T>::operator[](unsigned index) {


ListNode *&new_node = _index(index);


return new_node->data;


}
```

Join Code: 225



What is the Big O of random access?

Linked List: remove(<parameters>)

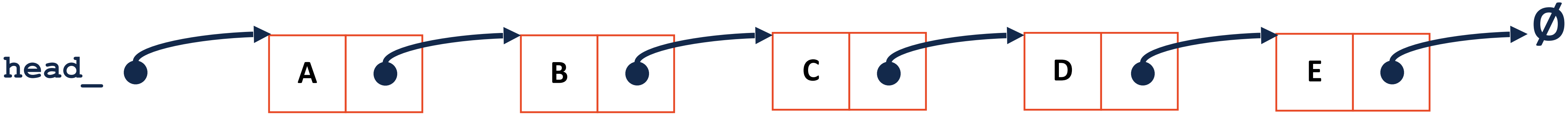
What input parameters make sense for remove?

↳ unsigned index

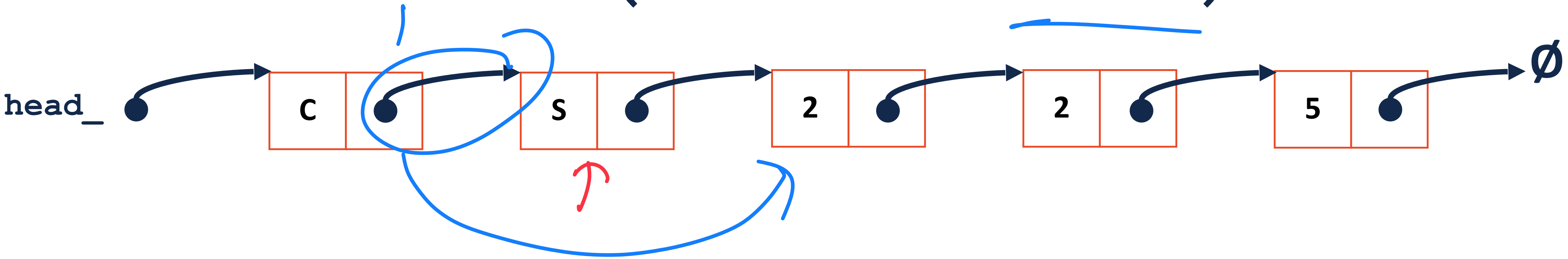
↳ T & value

↳ ListNode *& ref pointer

↳ remove bulk



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



```
ListNode * tmp = n;

n = n -> next;
delete tmp;
```

List.hpp

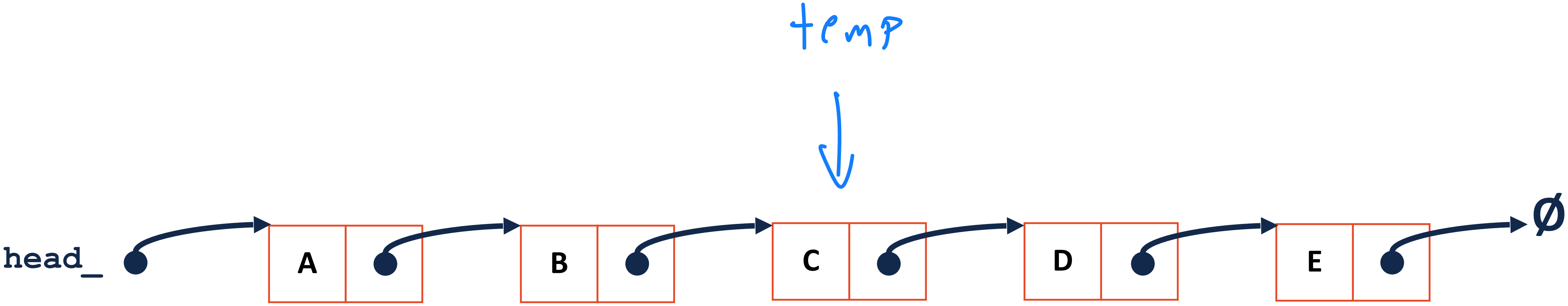
```
template <typename T>
T List<T>::remove(ListNode *& node) {

ListNode * temp = node;        // O(1)
node = node->next;             // O(1)
T data = temp->data;
delete temp;                   // O(1)
return data;

}
```

```
ListNode *& t;
      remove(t);
```

O(1)



Linked List: remove(T & data)

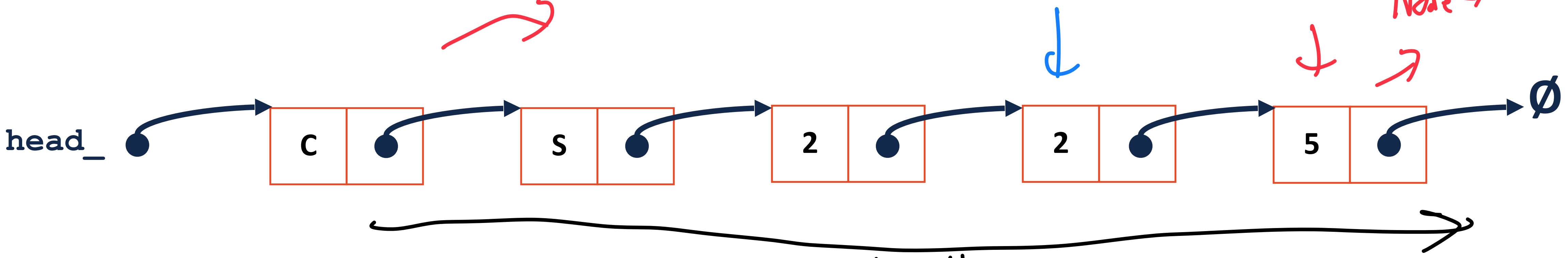
remove(5)

List Node * prev

List Node * curr

curr = curr -> next

Node -> data == 9

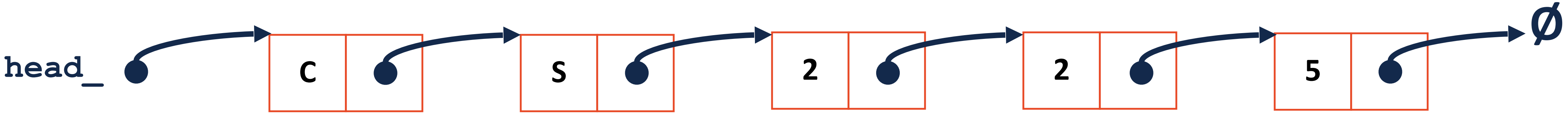


List Node *& curr

N items

O(N)

Linked List: remove



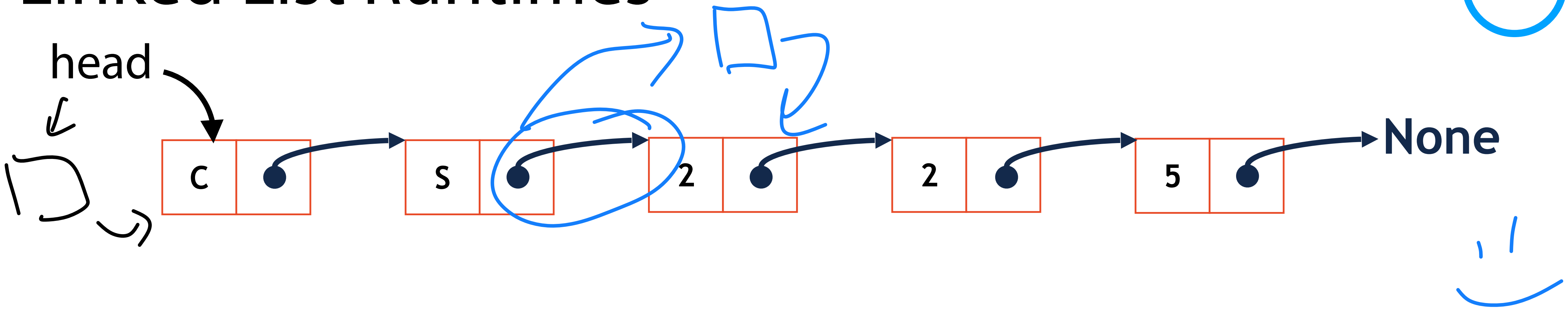
Running time for **remove(ListNode *&)**

↳ $O(1)$

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

↳ $O(n)$

Linked List Runtimes



	@Front	@RefPointer	@Index
Insert	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Thinking critically about linked lists…

When would we use insert/delete on a reference to a pointer?

↳ If we have already run find()

↳ Loop over list

↳ $O(n)$

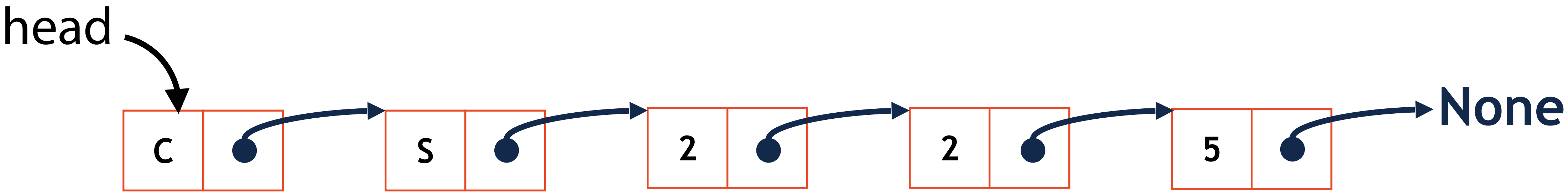
$O(1)$ for each mod

What is the runtime to find an item of interest?

↳ $O(n)$

List Implementations

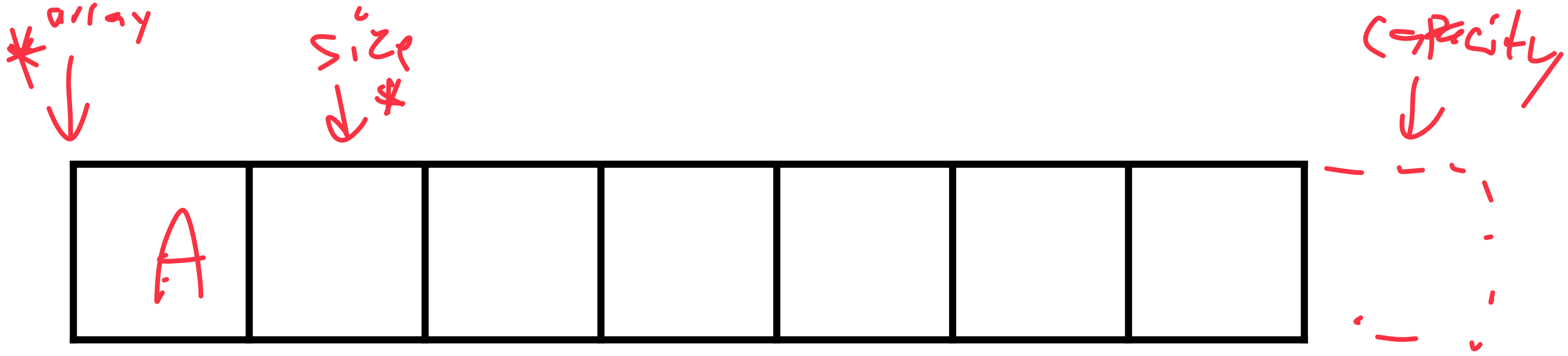
1. Linked List



2. Array List

C	S	2	2	5					

Array List



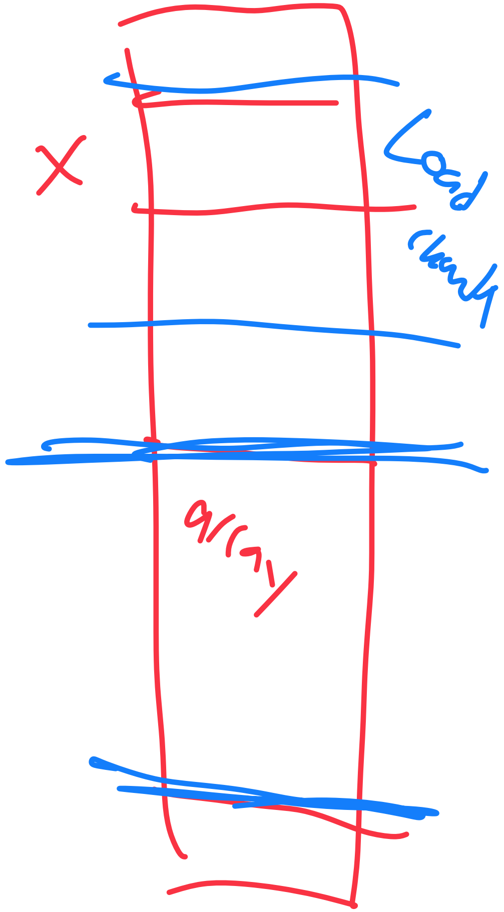
An array is allocated as continuous memory.

Three values are necessary for efficient array usage:

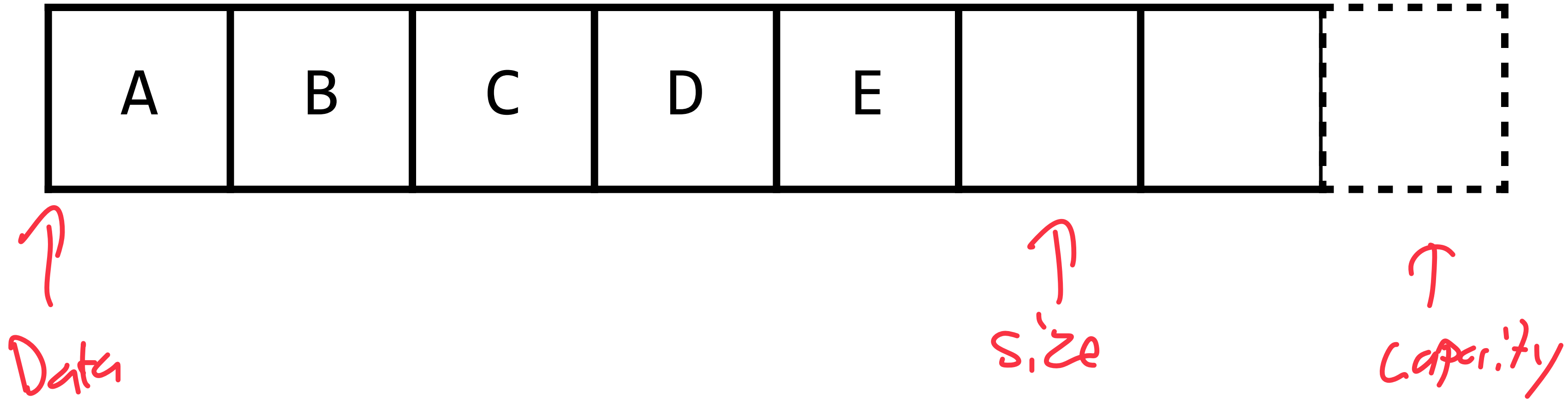
1) Data : the start location of my array

2) Size : How many things are in our array

3) Capacity : The max space allocated



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

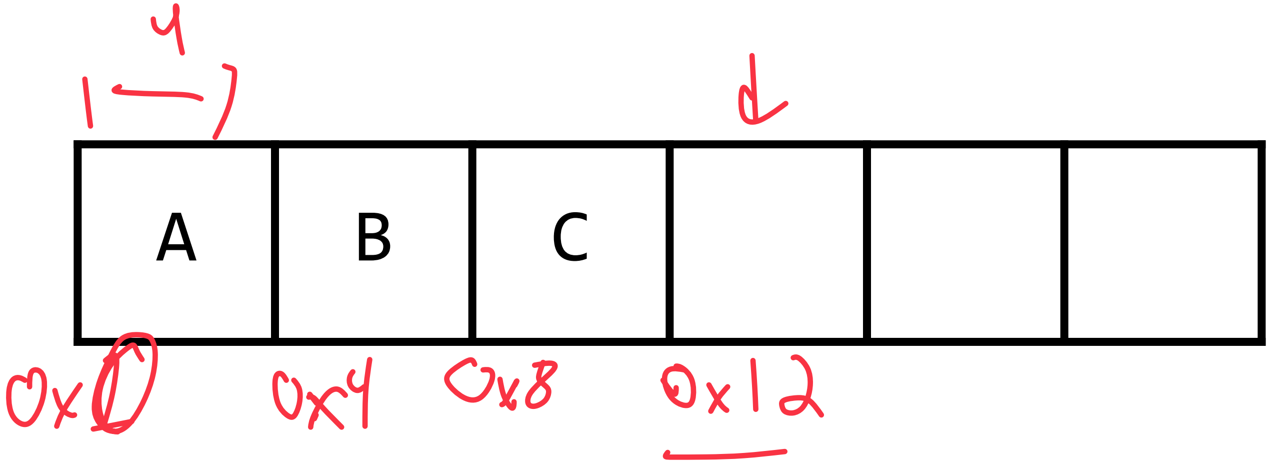
```
#pragma once

template <typename T>
class List {
public:
    /* --- */
private:
  T *data_;

  T *size_;

  T *capacity_;

    /* --- */
};
```



sizeof(4) = 4

int *p

p++;

start p p+1

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

size - data = 3 ← O(1)

0x12 0x0

List.h

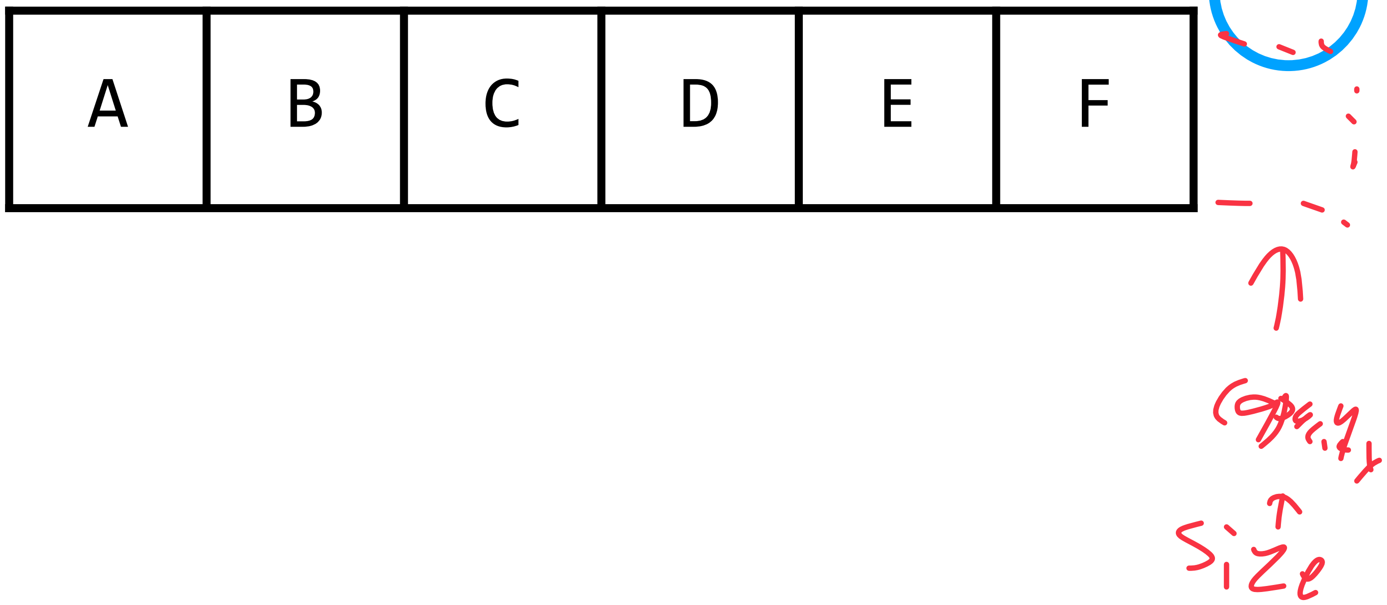
```
#pragma once

template <typename T>
class List {
public:
    /* --- */
private:
  T *data_;

  T *size_;

  T *capacity_;

    /* --- */
};
```



How do I know if I’m at capacity?

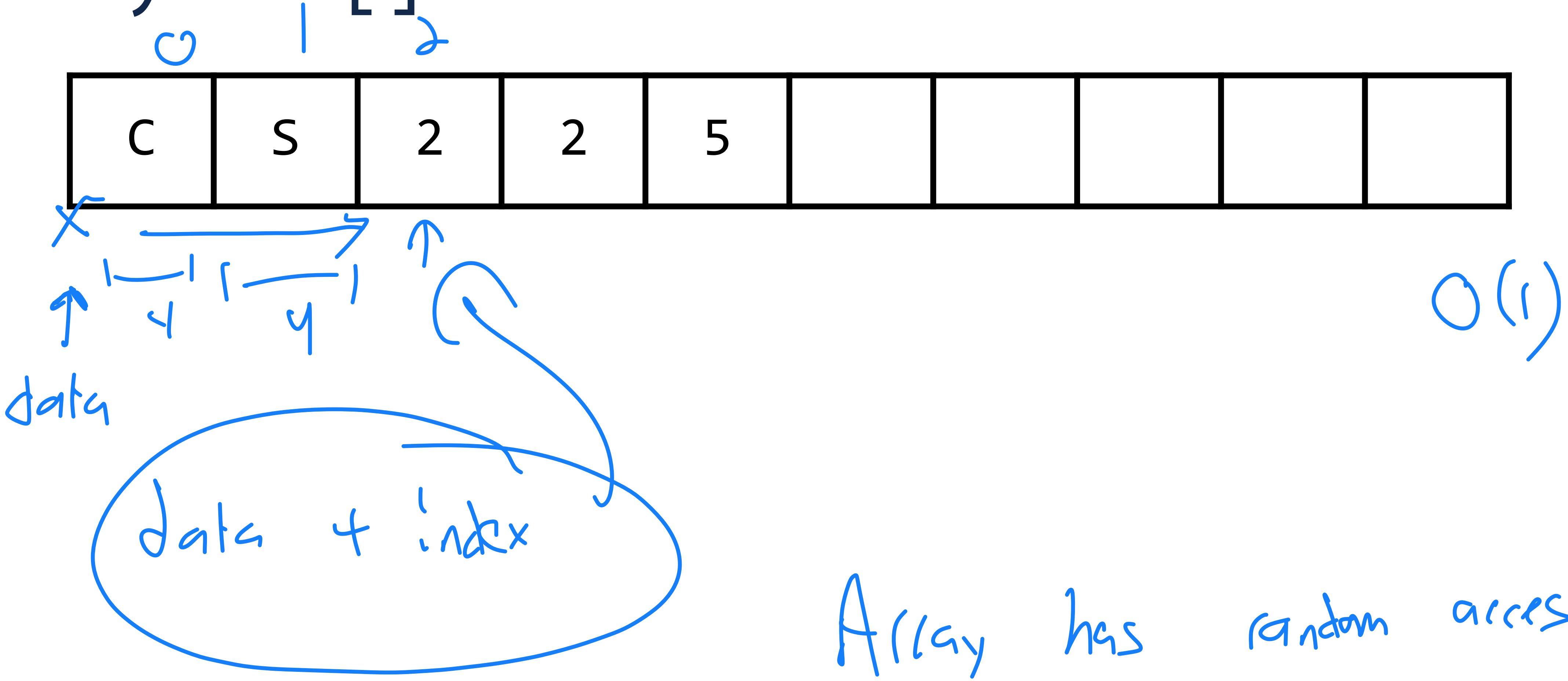
size == capacity, my array is full

↳ Dont insert

O(1)

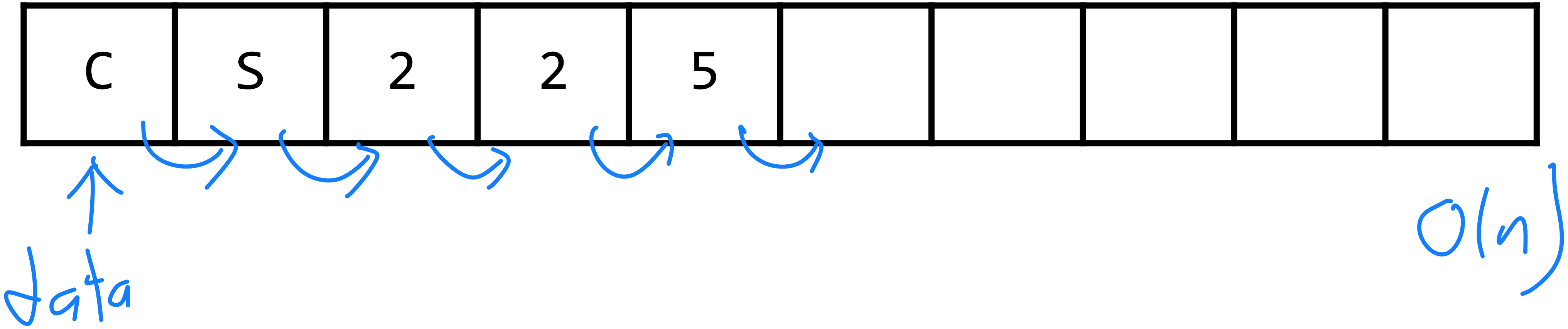
Array List: []

index



Array has random access

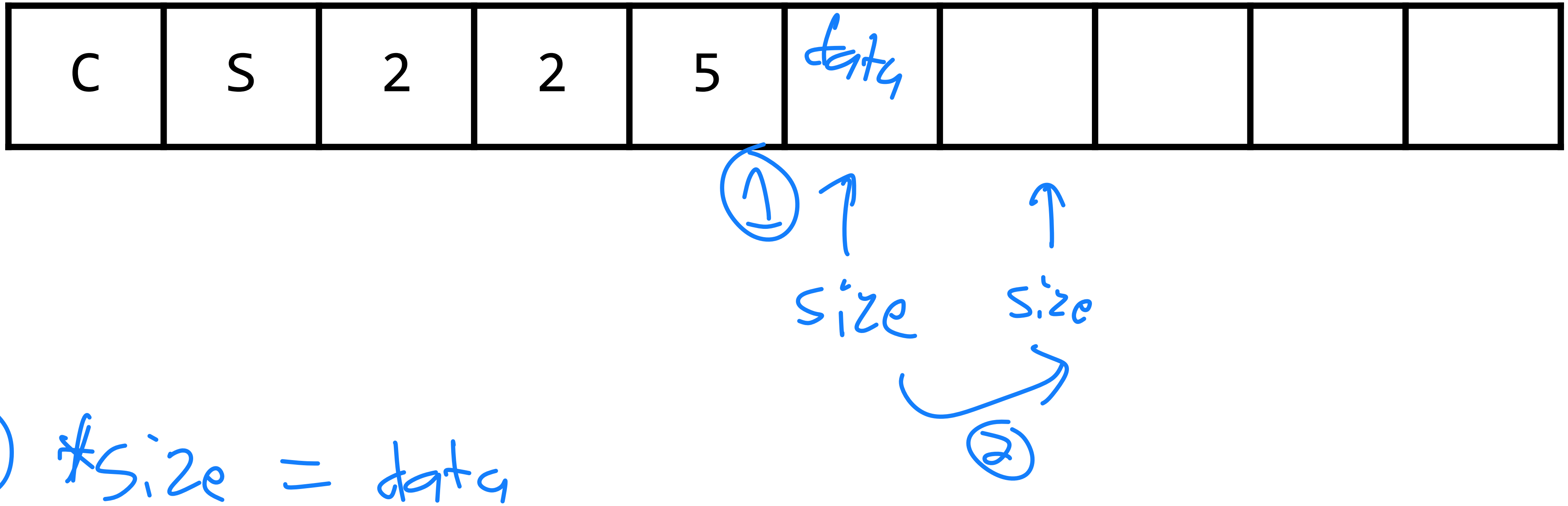
Array List: `insertFront(data)`



insert at front moves all items

Array List: `insertBack(data)`

* Not at capacity

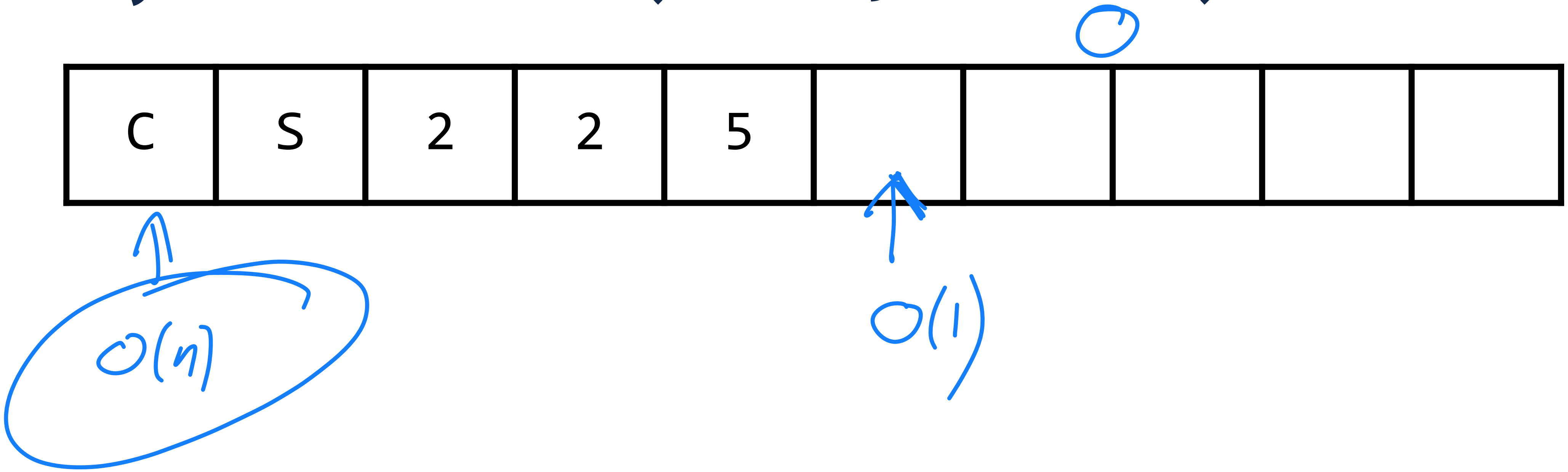


① *size = data

② size++;

O(1)

Array List: `insert(data, index)`



C	S	2	2	5					

Array List: `addspace(data)`

N	O	S	P	A	C	E