Data Structures

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

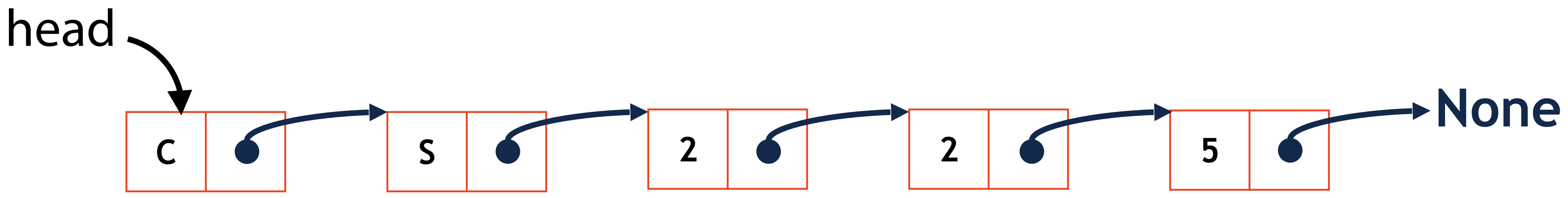
Finish implementing the List ADT (as a linked list)

Discuss data variables for implementing array lists

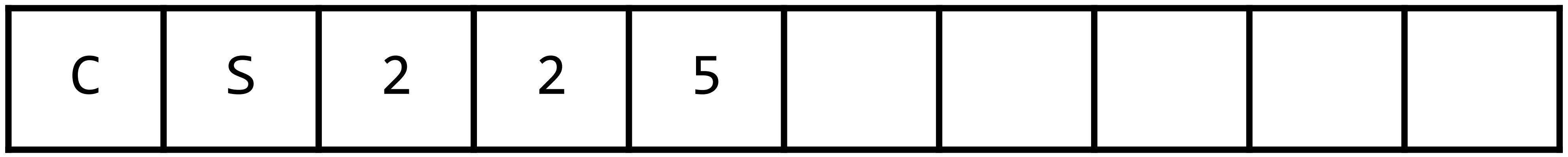
Explore the List ADT (as an array list)

List Implementations

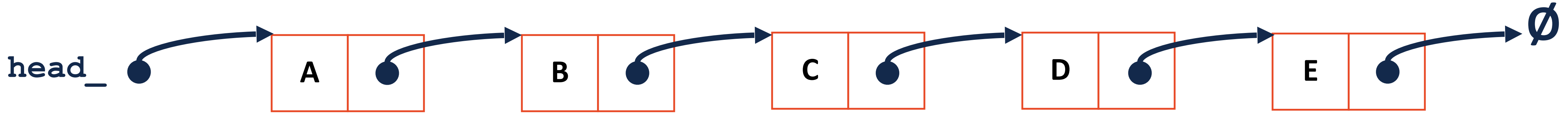
1. Linked List



2. Array List



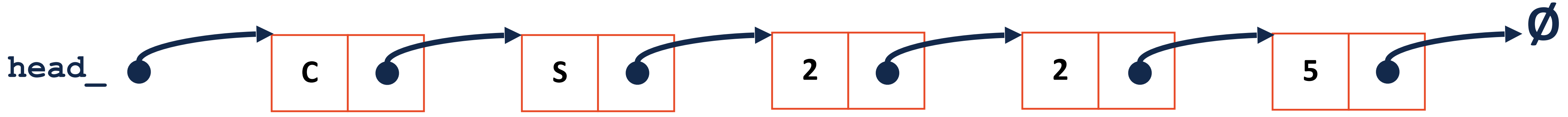
Comparing pointer to reference-to-pointer



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

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

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

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

  tmp->next = head_;

  head_ = tmp;

}
```

```
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;
}
```

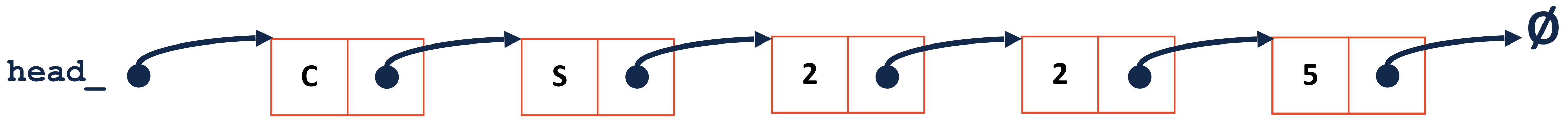
List Random Access [ ]

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

What return type should this function have?

List.hpp

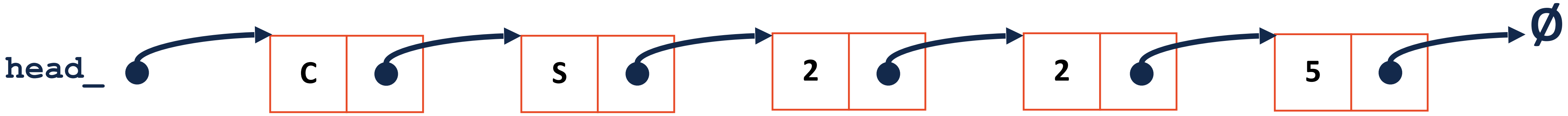
```
template <typename T>
T & List<T>::operator[](unsigned index) {










}
```



List.hpp

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


ListNode *&new_node = _index(index);


return new_node->data;


}
```

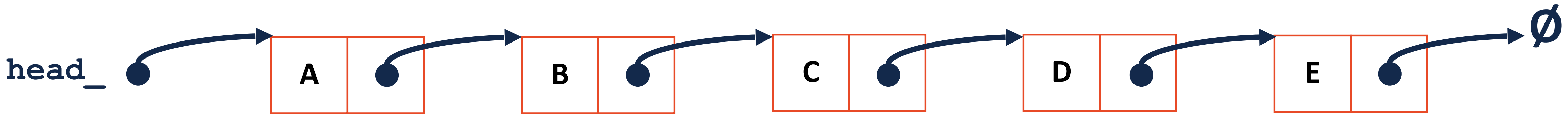
Join Code: 225



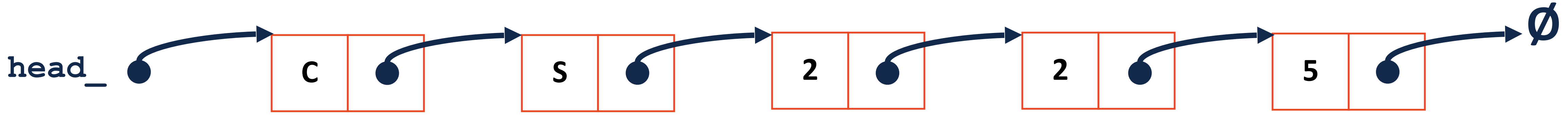
What is the Big O of random access?

Linked List: `remove(<parameters>)`

What input parameters make sense for remove?

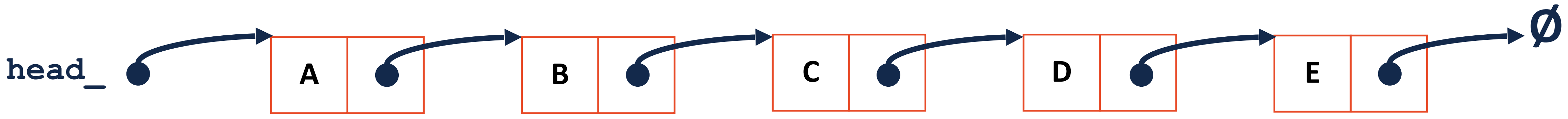


Linked List: `remove(ListNode *& n)`

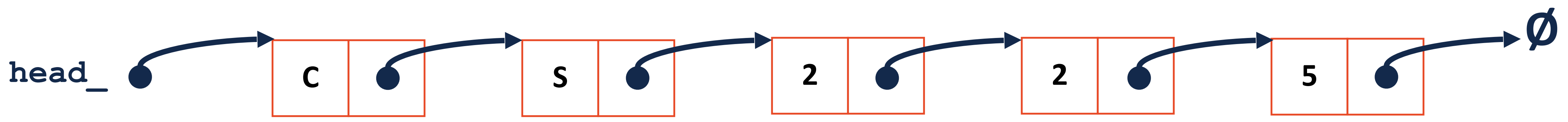


List.hpp

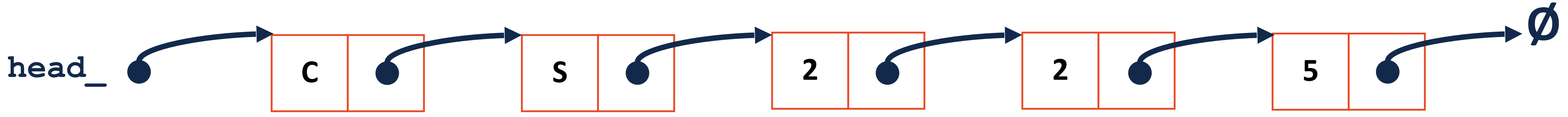
```
template <typename T>
T List<T>::remove(ListNode *& node) {

ListNode * temp = node;
node = node->next;
T data = temp->data;
delete temp;
return data;

}
```



Linked List: `remove(T & data)`



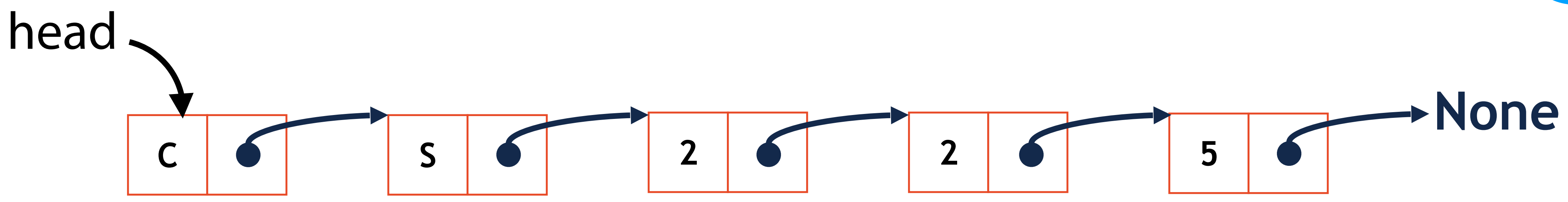
Linked List: remove



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

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

Linked List Runtimes



	@Front	@RefPointer	@Index
Insert			
Delete			

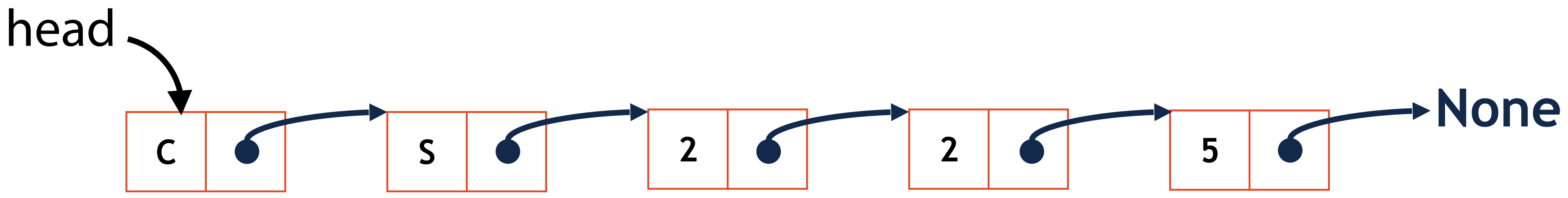
Thinking critically about linked lists…

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

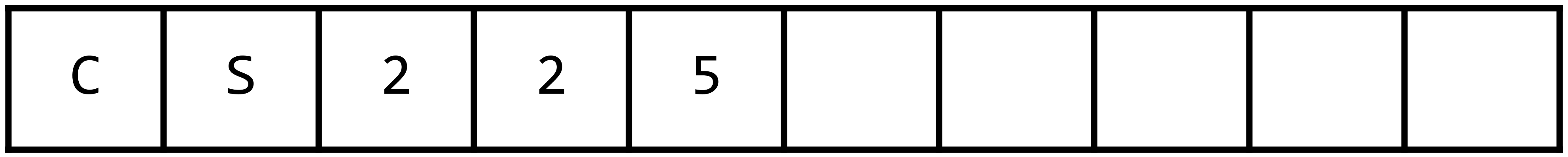
What is the runtime to find an item of interest?

List Implementations

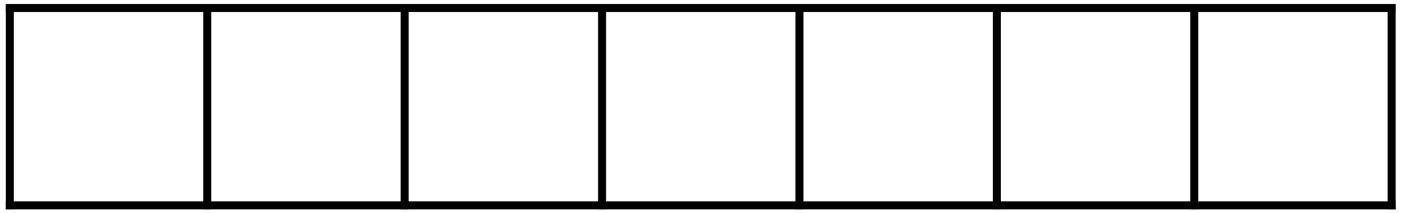
1. Linked List



2. Array List



Array List



An array is allocated as continuous memory.

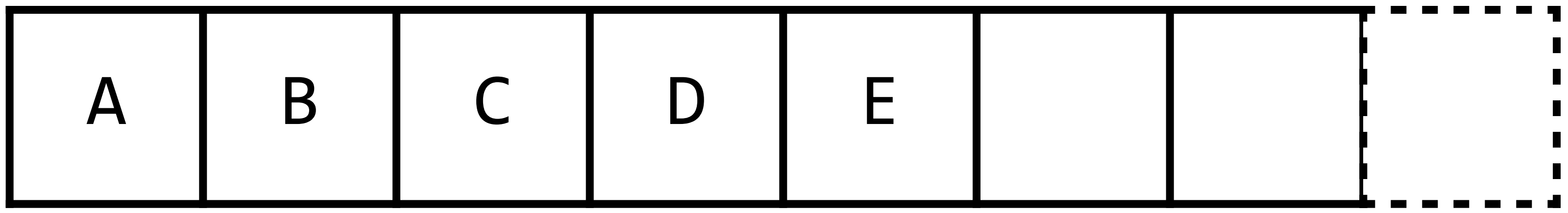
Three values are necessary for efficient array usage:

1)

2)

3)

Array List



In C++, vector is implemented as:

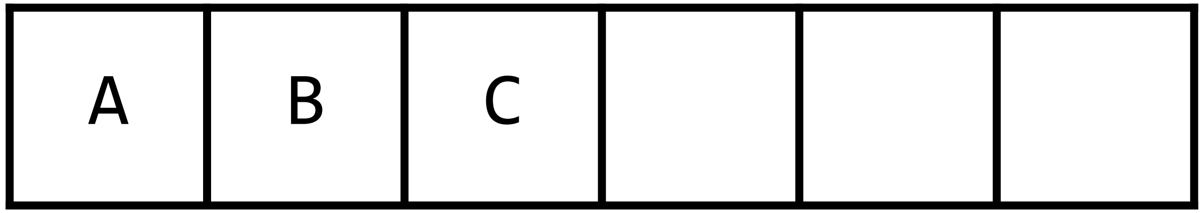
1) **Data:** Stored as a pointer to array start

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

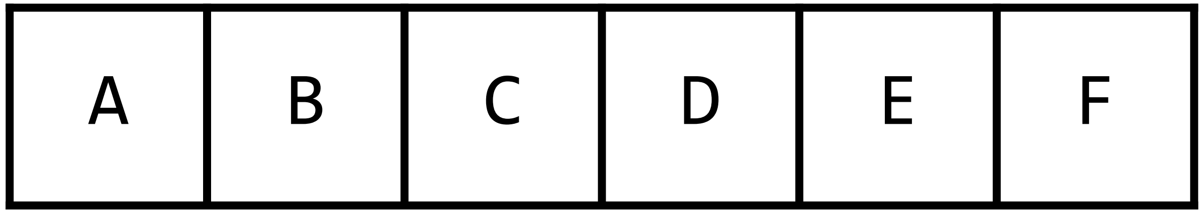
```cpp
#pragma once

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

  T *size_;

  T *capacity_;

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



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

List.h

```
#pragma once

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

  T *size_;

  T *capacity_;

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

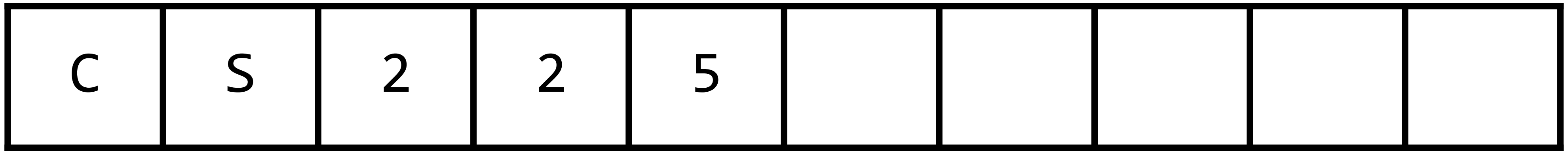


How do I know if I'm at capacity?

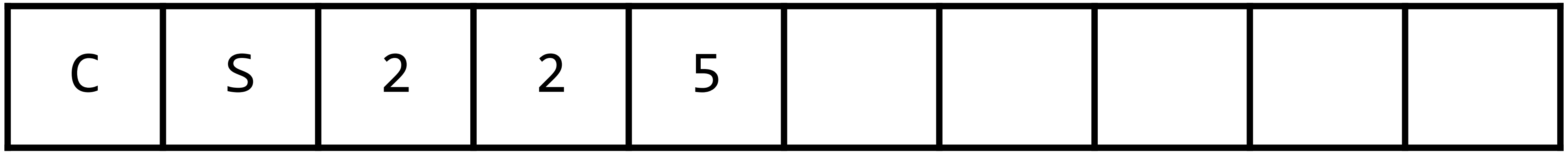
Array List: [ ]

C	S	2	2	5					

Array List: `insertFront(data)`



Array List: `insertBack(data)`



Array List: `insert(data, index)`

C	S	2	2	5					

Array List: `addspace(data)`

N	O	S	P	A	C	E