Data Structures

Lists and List ADT

CS 225

August 25, 2023

Brad Solomon & G Carl Evans

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Define the functions and operations of the List ADT

Discuss list implementation strategies

Explore how to code and use a linked list

Practice fundamentals of C++ in the context of lists

Pointer-to-constant vs constant pointer

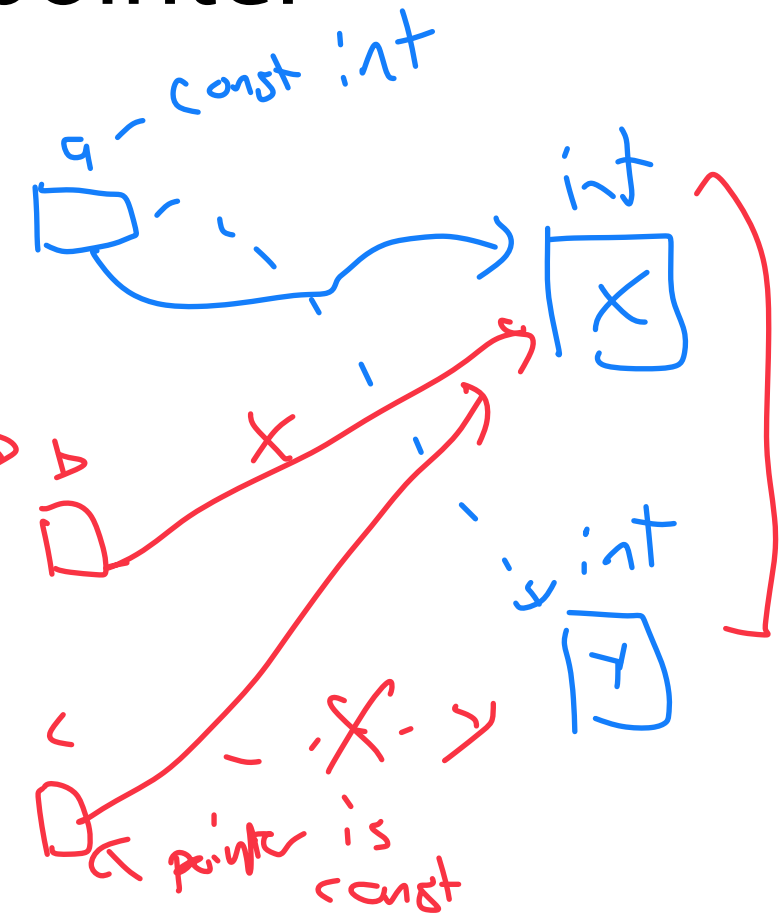
```
1  int x = 3;
2  int y = 2;
3  // *** A ***
4  const int* a = &x;
5
6
7  a = &y;
8
9  // *** B ***
10 const int* b = &x;
11
12
13 *b = y;
14
15 // *** C ***
16 int* const c = &x;
17
18
19 c = &y;
20
21 // *** D ***
22 int* const d = &x;
23
24
25 *d = y;
```

(const int)* a = &x;

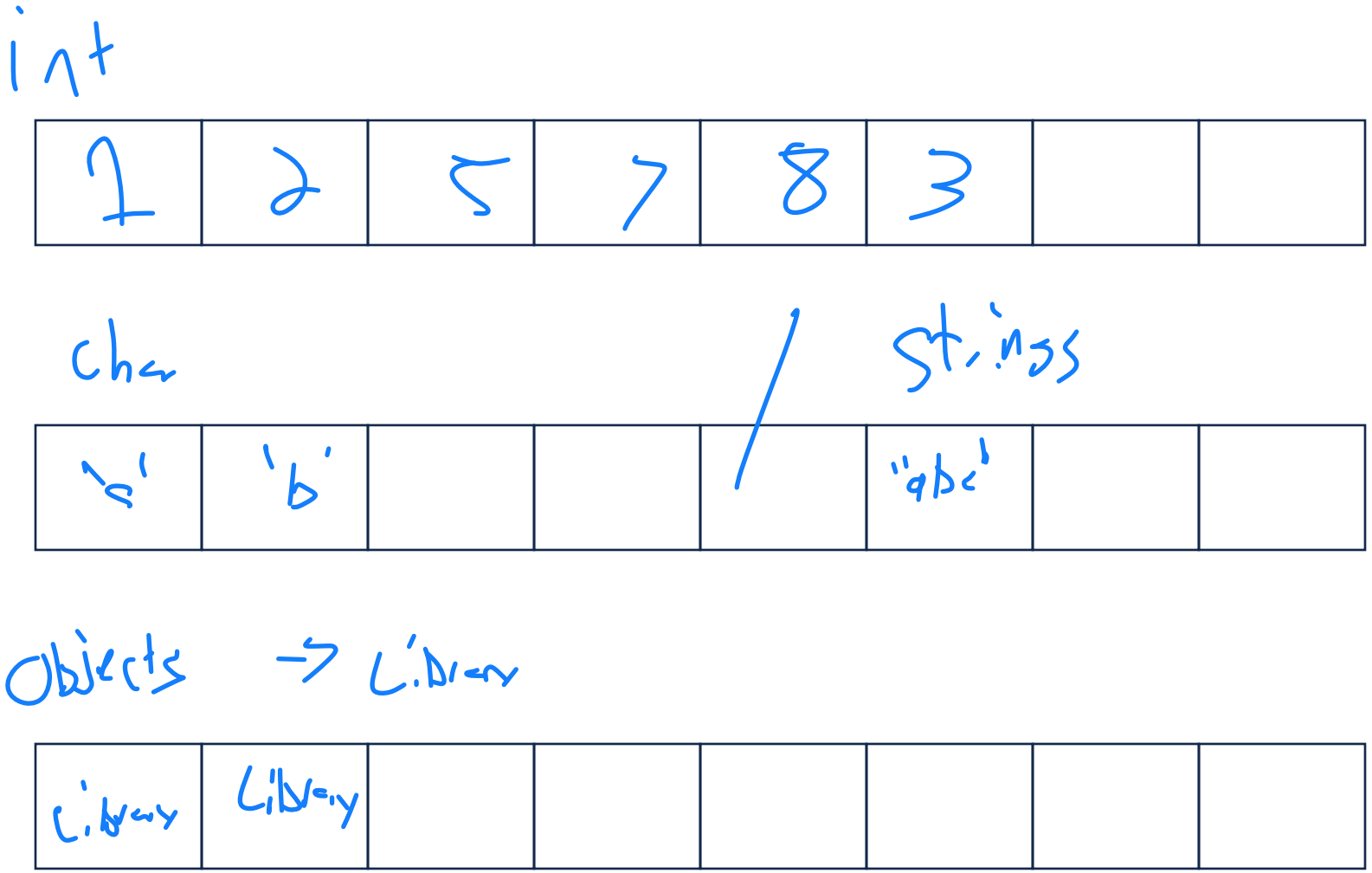
(const int)* b = &x;

(int*) const c = &x;

(int*) const d = &x;



What types of “stuff” do we want in our list?



Templates

int, float, string, ...

1) A way to write generic code

2) Only compiled when it is used

3) A template is a recipe for code

Compiler handles this

template1.cpp

```
1
2  template <typename T>
3  T maximum(T a, T b) {
4    T result;
5    result = (a > b) ? a : b;
6    return result;
7  }
```

int a = 5;

int b = 3;

return maximum(a, b)

float x = 2.7

float y = 3.9

maximum(x, y)

Abstract Data Types

A way of describing a data type as a combination of:

- **Data** being stored by the data type
- **Operations** that can be performed on the data type

The actual implementation details of the ADT aren't relevant!

List ADT (What do we want our list to do?)

Create an empty list

getItem(index)

addItem()

removeItem()

length()

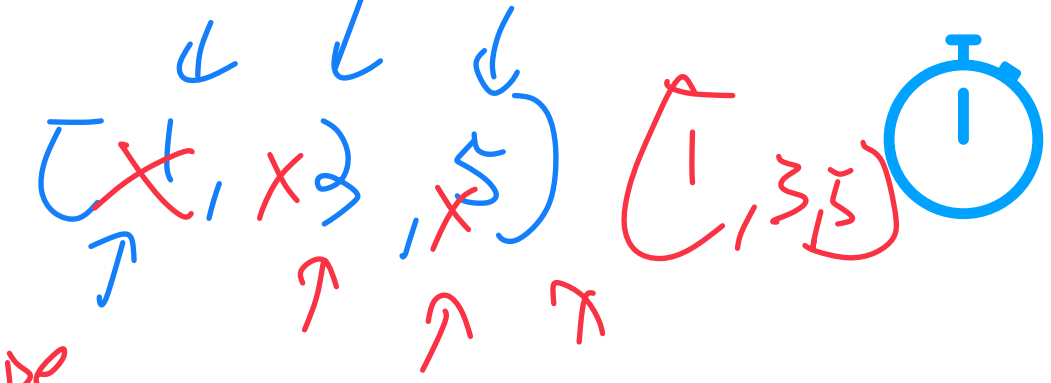
setItem(index, value) (=)

isEmpty()

sort()

findMax()

List ADT



A list is an ~~**ordered**~~ collection of items

Items can be either **heterogeneous** or **homogenous**

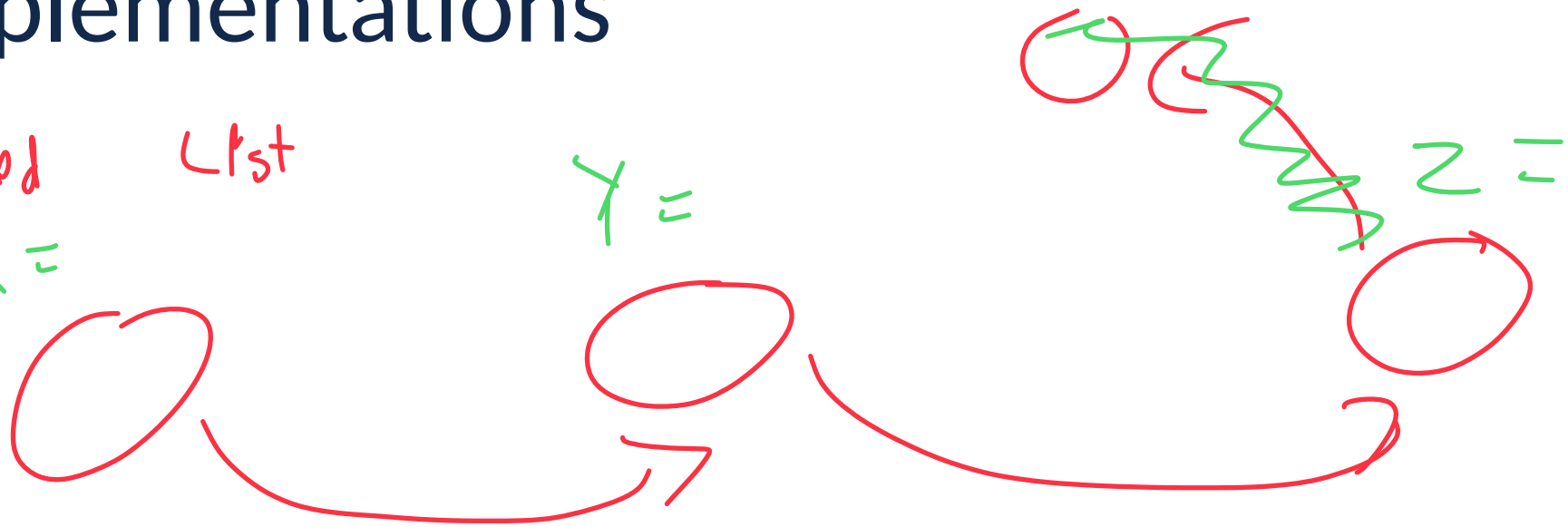
The list can be of a **fixed size** or is **resizable**

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

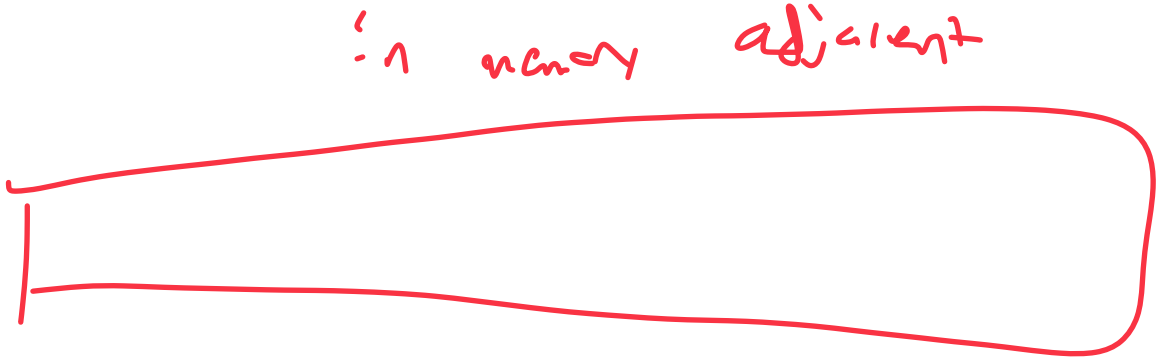
List Implementations

1. Linked List

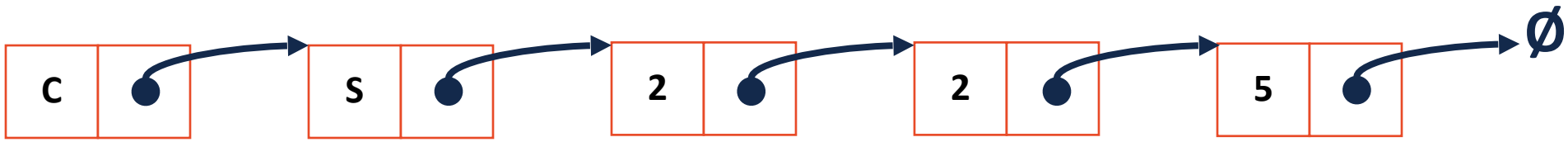



```
ListNode X= LN(1)
Y = LN(2)
Z = LN(3)
X.next = Y
Y.next = Z
ListNode *head = &X
```

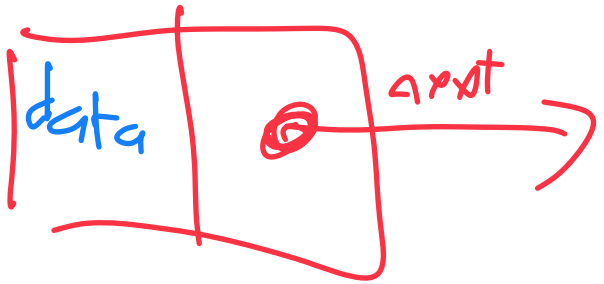
2. Array List



Linked List

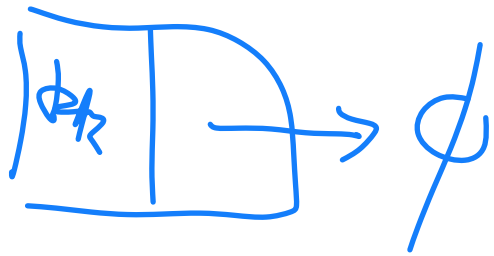



```
class CistNode {
    T & data;
    CistNode * next;
};
```



List.h

```
28 class ListNode {
29   T & data;
30   ListNode * next;
31   ListNode(T & data) : data(data), next(NULL) { }
32 };
```

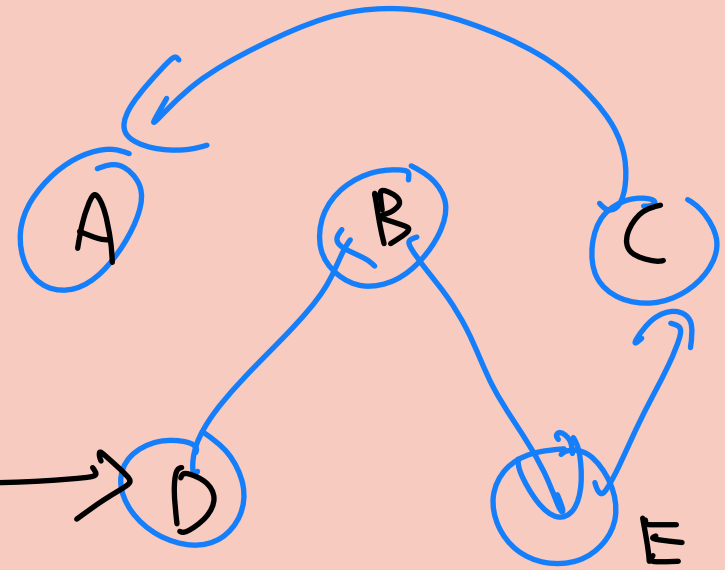


List.h

```
1   #pragma once
2
3   template <typename T>
4   class List {
5     public:
      /* ... */
28    private:
29      class ListNode {
30        T & data;
31        ListNode * next;
32        ListNode(T & data) :
           data(data), next(NULL) { }
33      };
34
35      ListNode *head_;
36
37      /* ... */
38
39  };
    ...
    ...
79  #include "List.hpp" // **A**
```

List.hpp

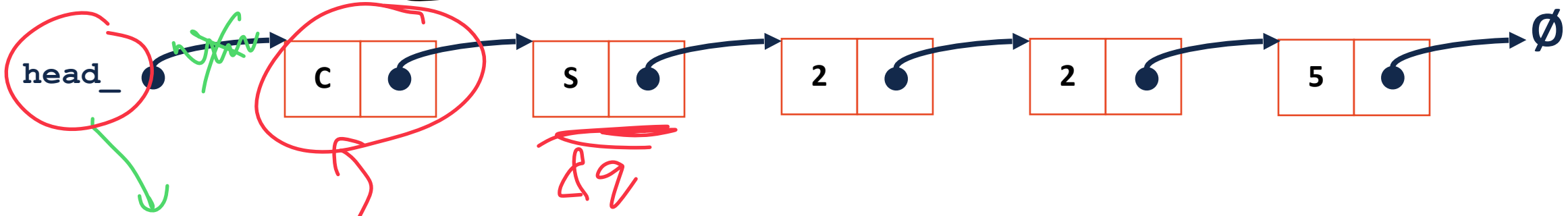
```
1   #include "List.h" // **B**
2
3   template <typename T>
4   void List<T>::insertAtFront(const T& t) {
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22  }
```

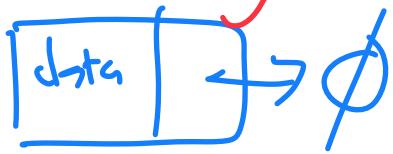


int insert at front (

ListNode (int) → insert At front

Linked List: `insertAtFront(data)`





n.next = head_

ListNode
td data = C

ListNode * next = q

1) Make new ListNode in w/ data

2) Make new Node's next point to head

3) Set head to point to n

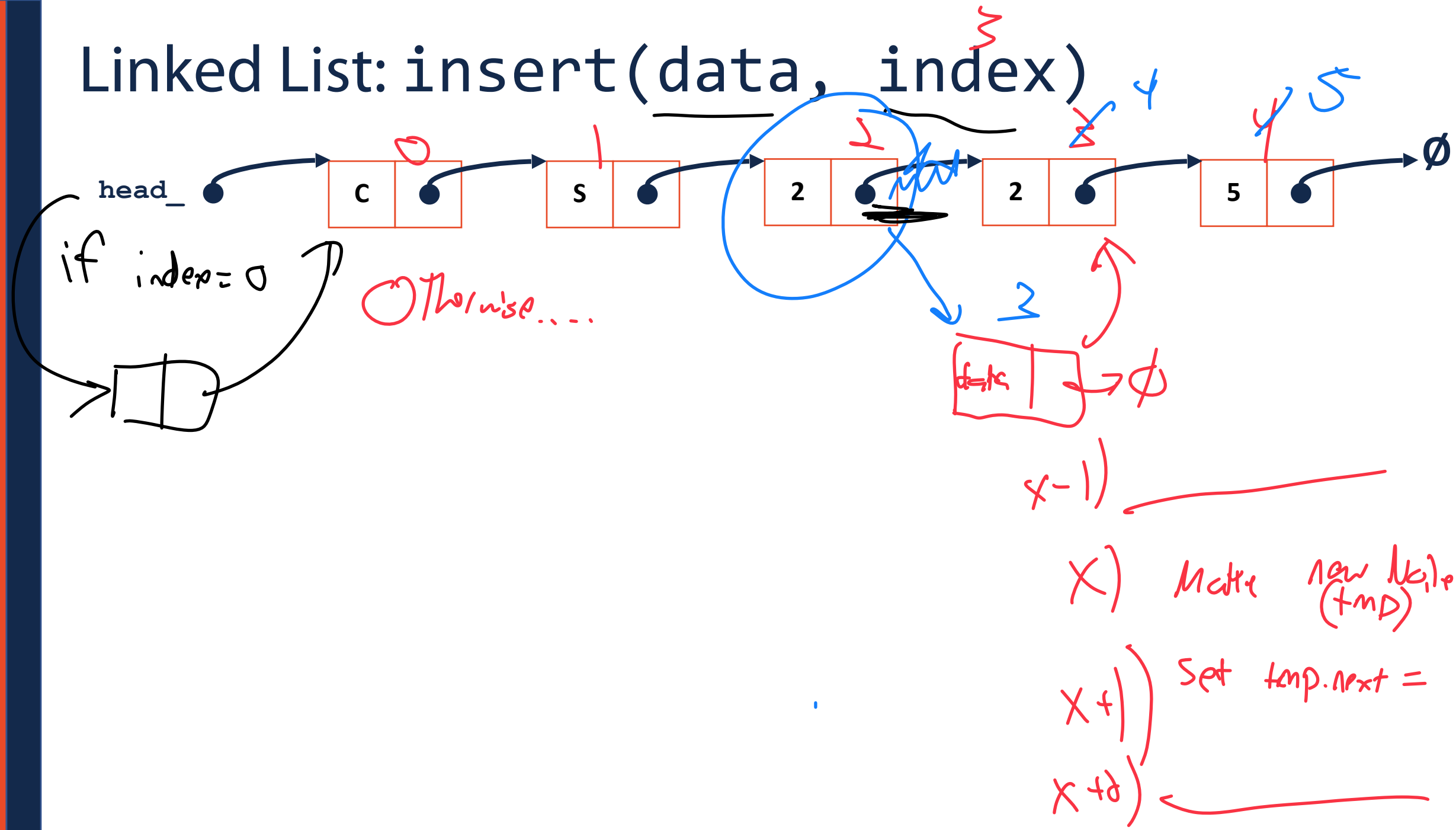
List.h

```
#pragma once

template <typename T>
class List {
  public:
    /* ... */
  private:
    class ListNode {
      T & data;
      ListNode * next;
      ListNode(T & data) :
        data(data), next(NULL) { }
    };

    ListNode *head_;

    /* ... */

};

...

#include "List.hpp"
```

List.hpp

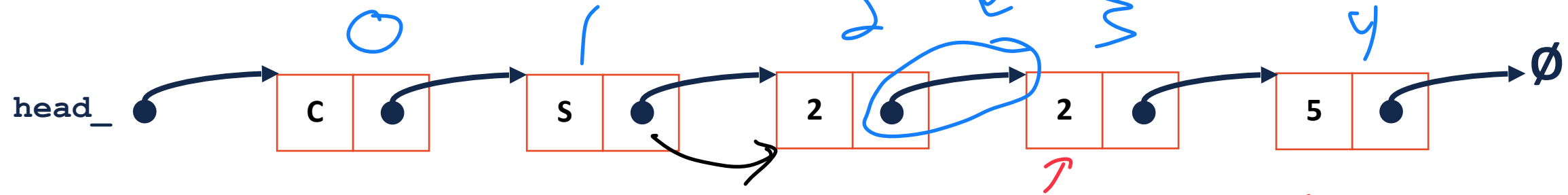
```
template <typename T>
void List<T>::insertAtFront(const T& t)
{
  1) Create new ListNode

  ListNode *tmp = new ListNode(data);

  tmp->next = head_;

  head_ = tmp;

}
```

Linked List: `insert(data, index)`



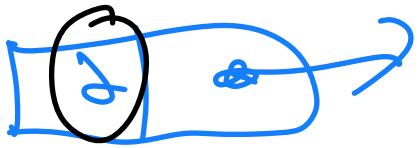
Linked List: _index(index)

What should the return type of _index( ) be?

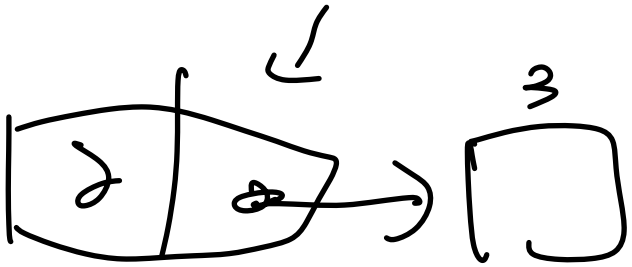


[template <class T>]

(A) T &



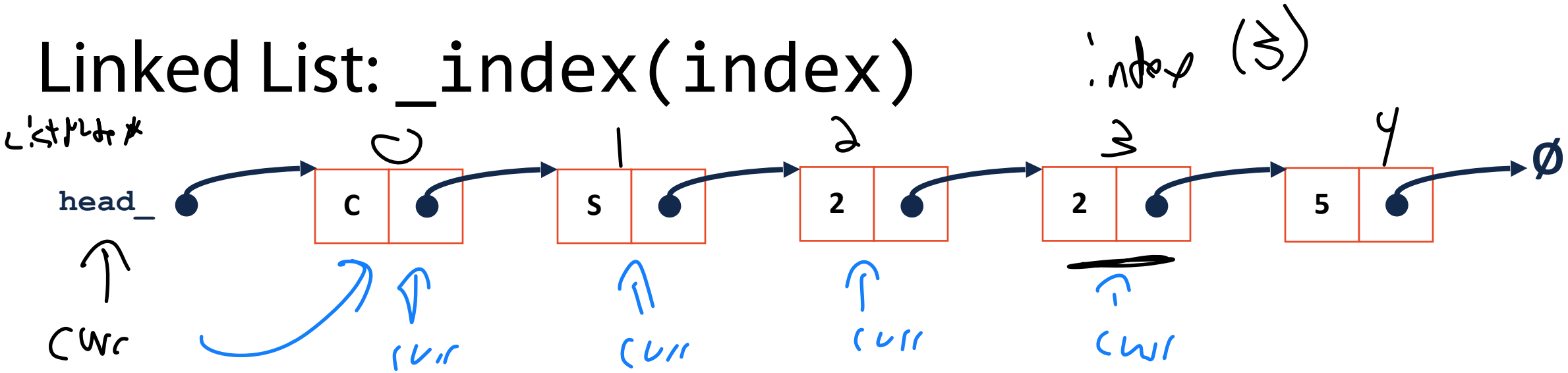
(B) ListNode



(C) ListNode *

(D) ListNode *&

Linked List: _index(index)



"walk" by setting curr = curr -> next

We will start from here on Monday!

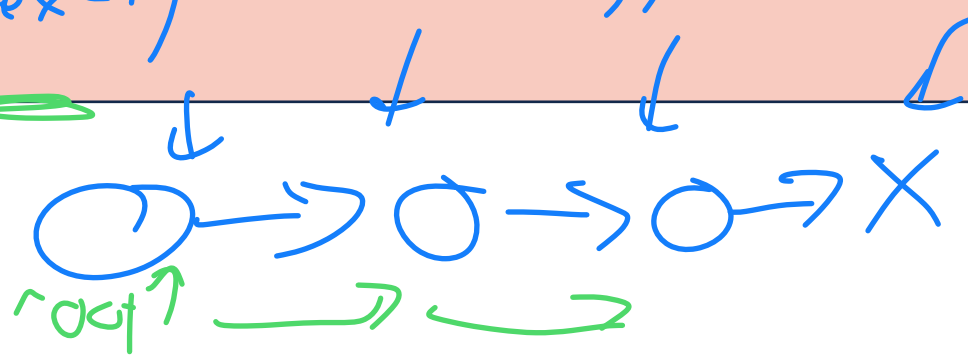
List.hpp

```
58 template <typename T>
59 typename List<T>::ListNode *& List<T>::_index(unsigned index){
60         return _index(index, head_)
61 }
```

helper function

```
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 }
```

```
if (root == nullptr){
    return root;
}

return _index( index-1, root->next);

if (index == 0){
    return root
}
```



List.hpp

```
58 template <typename T>
59 typename List<T>::ListNode *& List<T>::_index(unsigned index){
60         return _index(index, head_)
61 }
```

NULL

```
63 template <typename T>
64 typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66 if (index == 0 || node == nullptr){
67     return node;
68 }
69
70 return _index(index - 1, root -> next);
71
72
73 }
```

root -> next == 2

root

Rewrite this for Monday!

(Discuss on Monday)

Note: nullptr can come on insert as written

List.hpp

```
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 *thru = head;
7      for (unsigned i = 0; i < index - 1; i++) {
8         thru = thru->next;
9      }
10     return thru->next;
11   }
12 }
```

What is the running time for iterative index?

What is the running time for recursive index?