

Data Structures

Lists and List ADT

CS 225

August 28, 2026

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

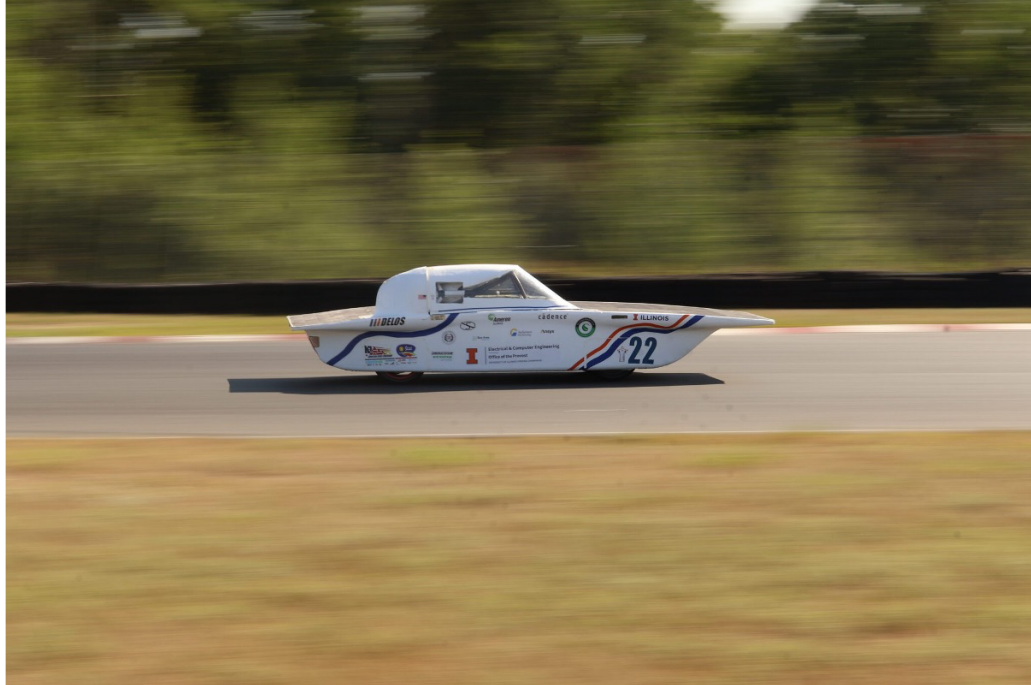
Department of Computer Science

Illini Solar Car Strategy & Telemetry

Strategy

How to drive the car most efficiently? (not always fast)

Physics simulation, data analysis, statistics, and ML



Telemetry

Ingest real-time data, show visualizations, run strategy models

Frontend, backend, databases, cloud

Interest
Form
>>>>



Bonus Video: Practice Exam 0 Releasing Today

Goes over every practice problem on exam

Gives some loose tips on study / exam habits

Look at it after running exam yourself!

MP_Stickers Releasing Today

An introduction to assignments in CS 225

A focused look at memory management / Rule of Three

Practice with Image processing / code testing

No extra credit / early submission for this MP!

Be sure to set up your Illinois CS 225 Git repo!

General MP Tips

Start by reading the web page!

Take a look at the doxygen for **interface** requirements

The **implementation** is decided by you!

Most autograder tests are provided for you locally

MP_Stickers Highlights

Part 2: StickerSheet

Big Picture: Memory Management! (Who owns what?)

Read the docs!

◆ render()

Image StickerSheet::render () const

Renders the whole **StickerSheet** on one **Image** and returns that **Image**.

The base picture is drawn first and then each sticker is drawn in order starting with layer zero (0), then layer one (1), and so on.

If a pixel's alpha channel in a sticker is zero (0), no pixel is drawn for that sticker at that pixel. If the alpha channel is non-zero, a pixel is drawn. (Alpha blending is awesome, but not required.)

The returned image always includes the full contents of the picture and all stickers. It should expand in each corresponding direction if stickers go beyond the edge of the picture.

Returns

An **Image** object representing the drawn scene

Exam 0 (1/26 — 1/29)



An introduction to CBTF exam environment / expectations

Quiz on foundational knowledge from all pre-reqs

Practice questions can be found on PL

Topics covered can be found on website

Registration open now

<https://courses.engr.illinois.edu/cs225/exams/>

Exam 0 (9/2 — 9/4)



An introduction to CBTF exam environment / expectations

Quiz on foundational knowledge from all pre-reqs

Practice questions can be found on PL

Topics covered can be found on website

Registration open now :)

<https://courses.engr.illinois.edu/cs225/exams/>

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

Last time: Memory management

Local memory on the stack is managed by the computer

Heap memory allocated by **new** and freed by **delete**

Pass by value makes a copy of the object

Pass by pointer can be dereferenced to modify an object

Pass by reference modifies the object directly

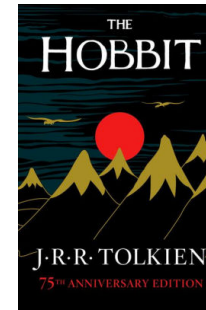
Memory Management - Parameters

Pass by **Value**:

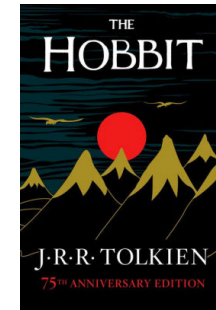
A local copy of the original

Ex: `addBook(Book book)`

Copy



Original

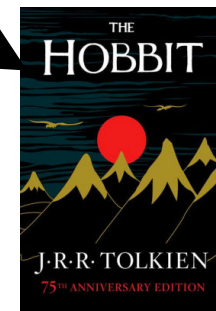
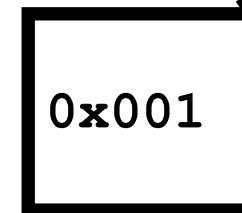


Pass by **Pointer to Value**:

An address on the heap

Ex: `addBook(Book* book)`

P



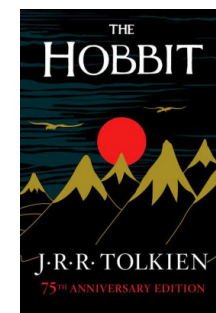
Pass by **Reference**:

An *alias* to an existing variable

Ex: `addBook(Book& book)`

Local var Y
is X

X=

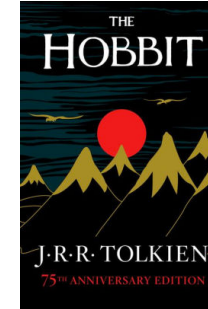


Memory Management - Ownership

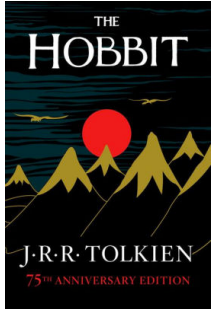
Pass by **Value**:

Local scope owns local copies
(They are stack copies)

Copy



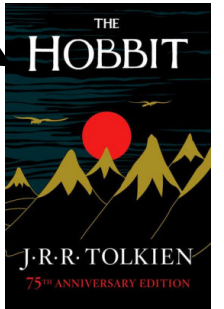
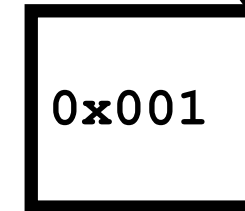
Original



Pass by **Pointer to Value**:

Local scope owns local pointer
NOT original data value

P

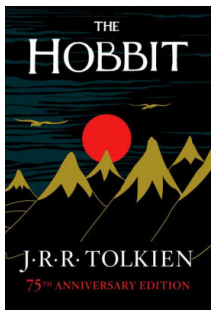


Pass by **Reference**:

Compiler alias with local scope
NOT original data value

Local var Y
is X

X=



Memory Management - Parameters

Which implementation do you prefer?

```
1 class Library {
2 public:
3     int numBooks;
4     std::string * titles;
5 };
6
7
8 // *** Function A ***
9 std::string getFirstBook(Library l){
10     return (l.numBooks > 0) ? l.titles[0] : "None";
11 }
12
13
14 // *** Function B ***
15 std::string getFirstBook(Library * l){
16     return(l->numBooks > 0) ? l->titles[0] : "None";
17 }
18
19
20 // *** Function C ***
21 std::string getFirstBook(Library & l){
22     return (l.numBooks > 0) ? l.titles[0] : "None";
23 }
24
```

The Rule of Three

If it is necessary to **define any one** of these three functions in a class, it will be necessary to **define all three** of these functions:

1. Destructor — Called when we delete object
2. Copy Constructor — Make a new object as a copy of an existing one
3. Copy assignment operator — Assign value from existing X to Y

'The Rule of Zero'

A corollary to Rule of Three

Classes that **declare** custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership. Other classes **should not declare** custom destructors, copy/move constructors or copy/move assignment operators

— Scott Meyers

Memory Management — Ownership

```
1 class Book{
2 public:
3     Book(std::string title, std::string author);
4
5     std::string title;
6
7
8     std::string author;
9
10 };
11
12 std::vector<Book*> universe_of_books;
13
14 universe_of_books.push_back(new Book("The
15 Hobbit", "J.R.R. Tolkien"));
16
17 library.addBook(universe_of_books.back());
18
```

```
1 class Library {
2 public:
3     int numBooks;
4     std::string * titles;
5     ~Library();
6     Library( int num, std::string* list );
7 };
8
9 Library::~~Library() {
10     delete[] titles;
11     titles = nullptr;
12 }
13
14 Library::Library(int num, std::string* list) {
15     numBooks = num;
16     titles = new std::string[ num ];
17     std::copy(inList, inList + num, titles);
18 }
19
20 int main() {
21     std::string myBooks[3] = {"A", "B", "C"};
22     Library L1( 3, myBooks );
23     Library L2( L1 );
24     return 0;
25 }
```

```
1 class Library {
2 public:
3     int numBooks;
4     std::string * titles;
5     ~Library();
6     Library( int num, std::string* list );
7 };
8
9 Library::~~Library() {
10     delete[] titles;
11     titles = nullptr;
12 }
13
14 Library::Library(int num, std::string* list) {
15     numBooks = num;
16     titles = new std::string[ num ];
17     std::copy(inList, inList + num, titles);
18 }
19
20 int main() {
21     std::string myBooks[3] = {"A", "B", "C"};
22     Library L1( 3, myBooks );
23     Library L2( L1 );
24     return 0;
25 }
```

Whats wrong with this code?

- A. Can't create L2 Library obj
- B. Don't delete either Library
- C. The second object being deleted crashes



Questions?



Templates

A way to write generic code whose type is determined during compilation



Templates

A way to write generic code whose type is determined during compilation

1. Templates are a recipe for code using generic types



Templates

A way to write generic code whose type is determined during compilation



1. Templates are a recipe for code using generic types

2. The compiler uses templates to generate C++ code **when needed**

```
T sum(T a, T b) {  
    ...  
}
```

template1.hpp



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

Templates are very useful!

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

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

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

MP_Stickers Highlights

Part 1: test_invert()

Your first example of a templated function!

```
1  template <class T>
2  bool test_invert(){
3      // Your code here
4      // (As this is the only
5  templated function, we aren't
... using a .hpp file in this
28 instance)
29     return false;
30 }
31
32
33
34
```

What is your favorite data structure?

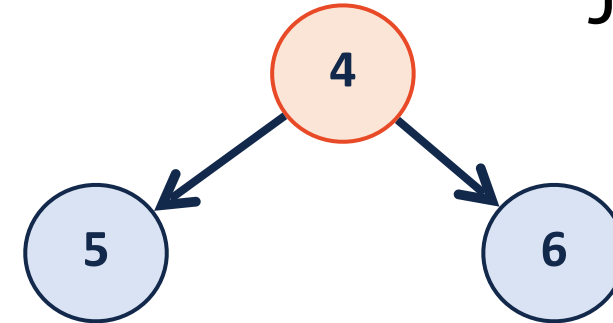


Join Code: 225

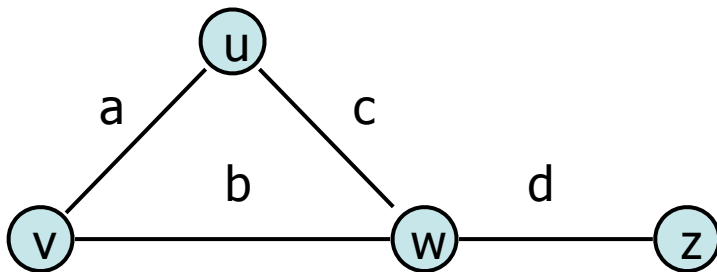
A) Lists



B) Trees



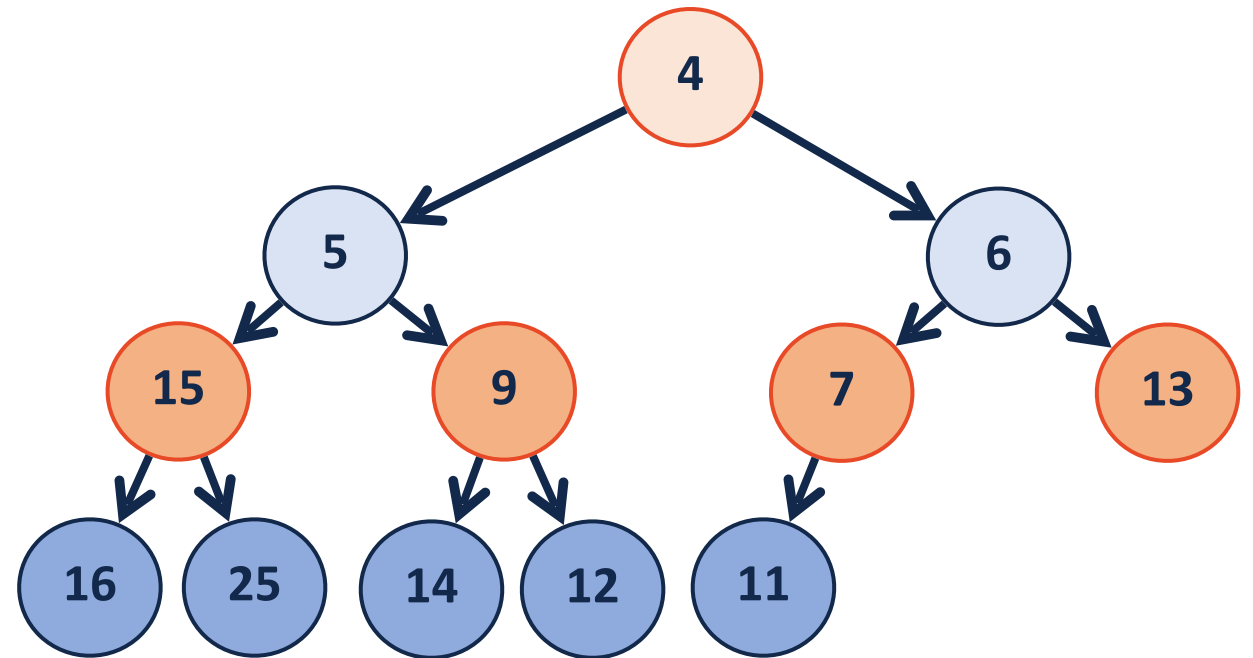
C) Graphs



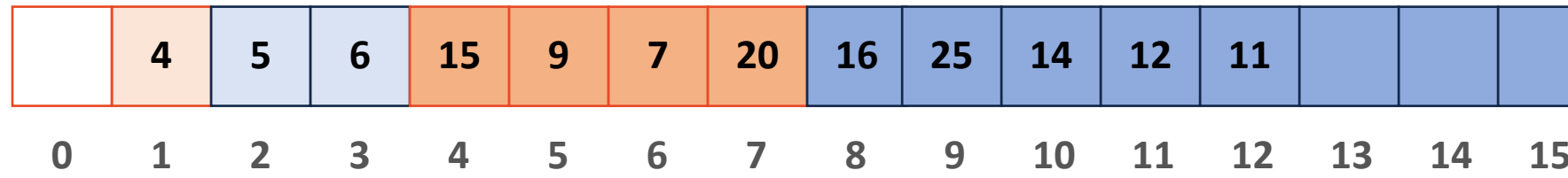
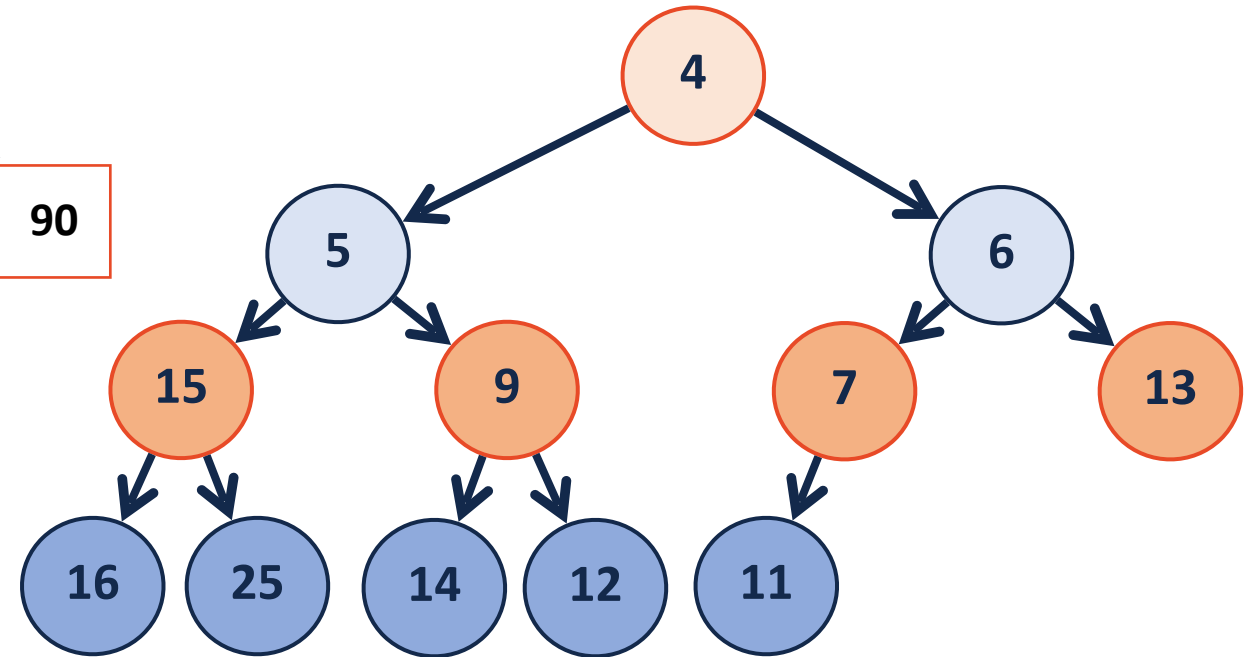
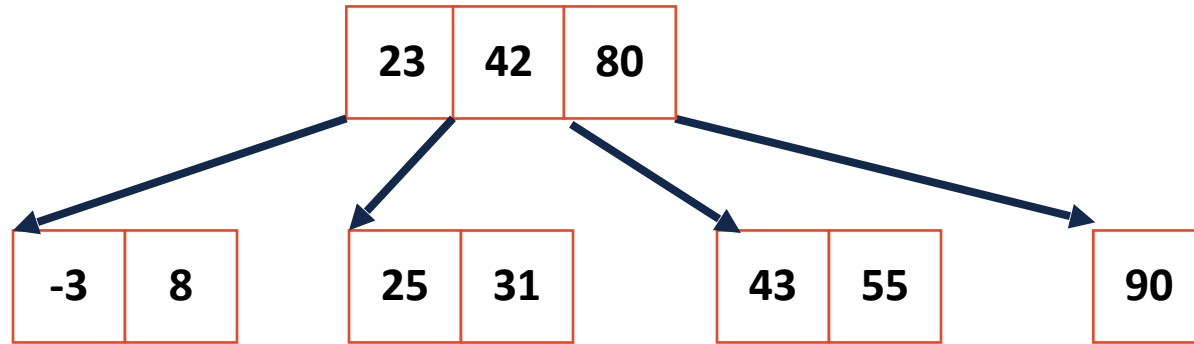
D) Hash Tables

0	Apple
1	∅
2	Pear

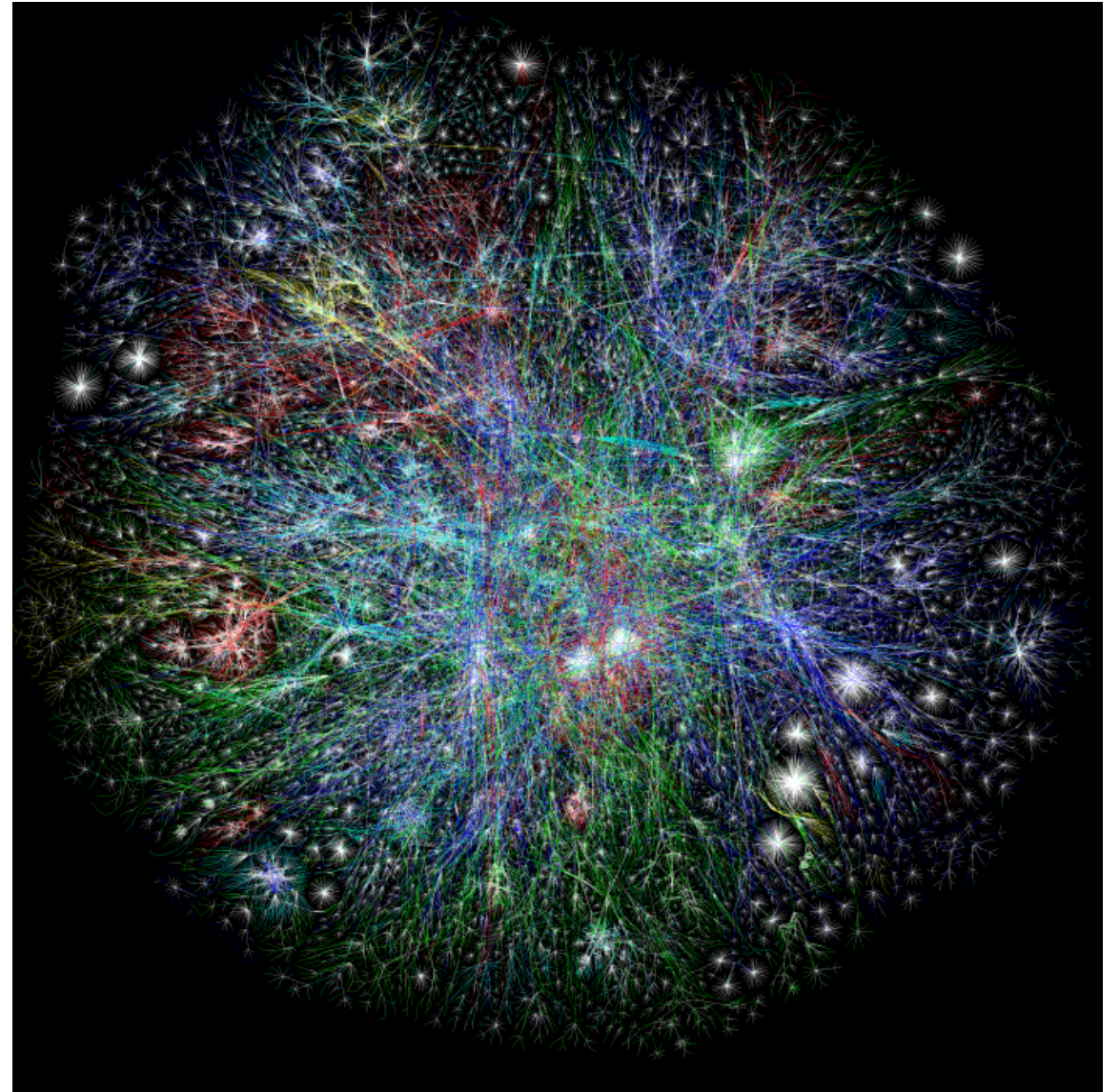
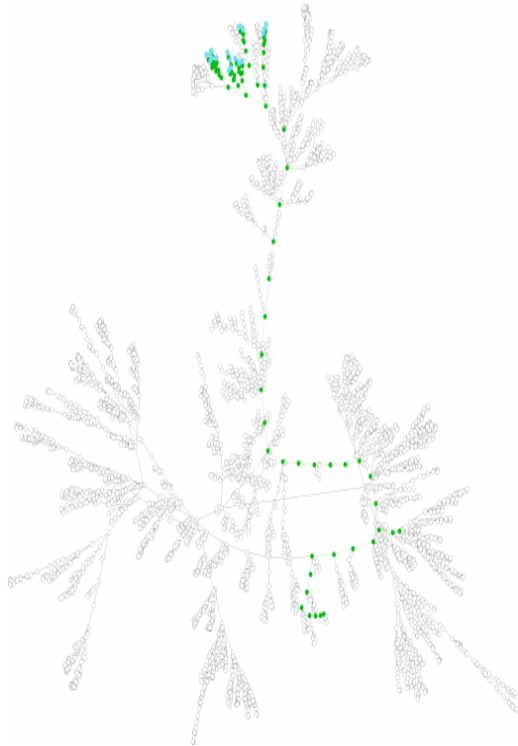
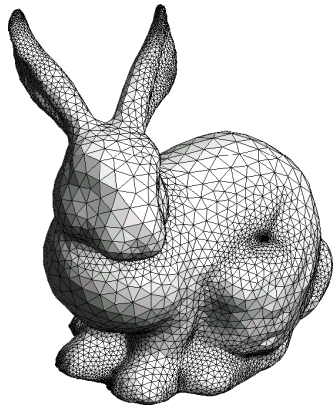
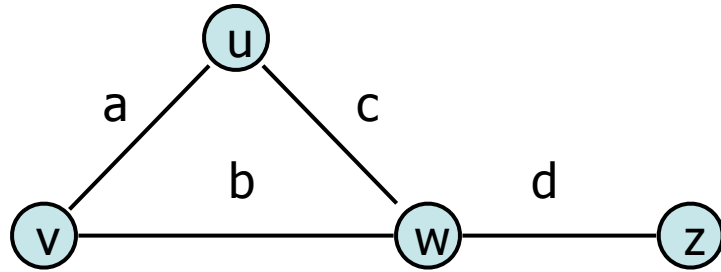
Your favorite data structure: Trees



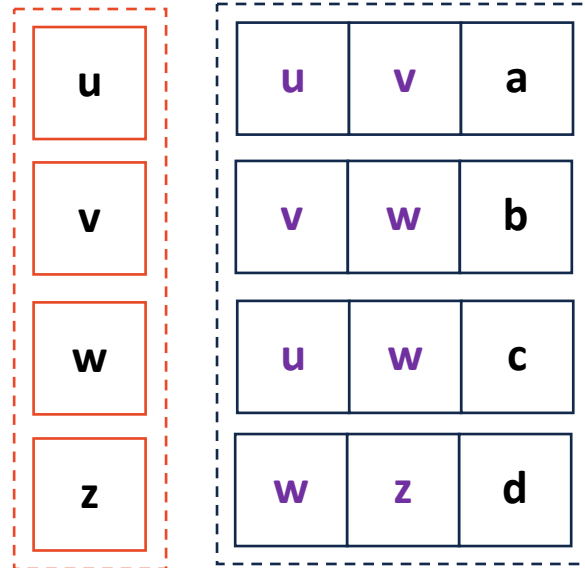
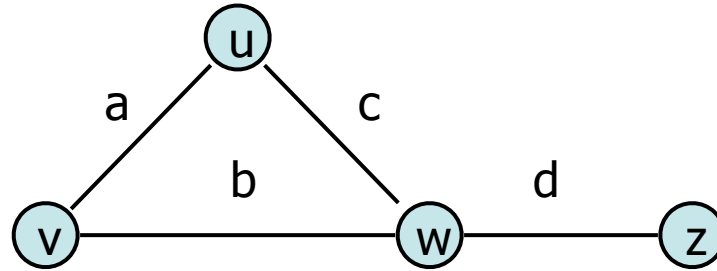
Your favorite data structure: ~~Trees~~ Lists



Your favorite data structure: Graphs

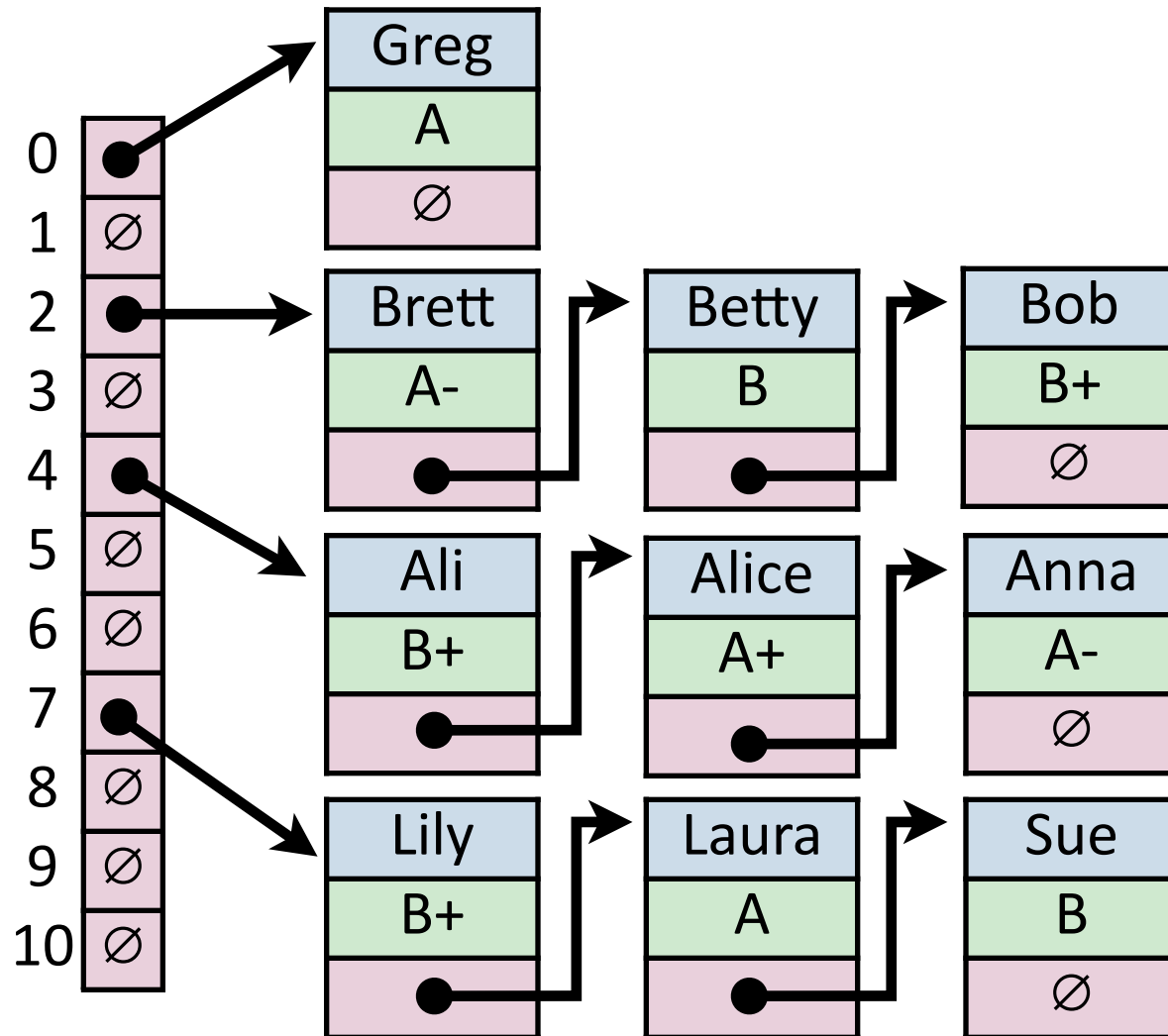


Your favorite data structure: ~~Graphs~~ *Lists*



Your favorite data structure: Hash Tables

$$H = \{h_1, h_2, \dots, h_k\}$$



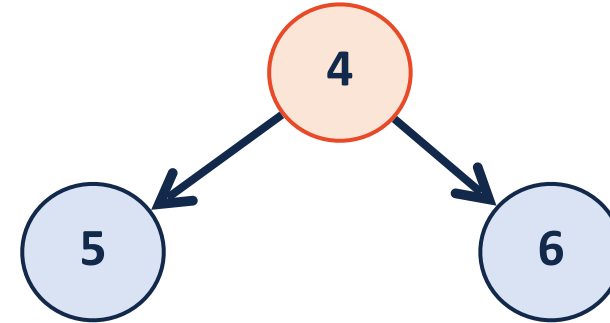
1
0
1
0
1
0
0
1
0
0
0

So 100% of people are excited about lists!

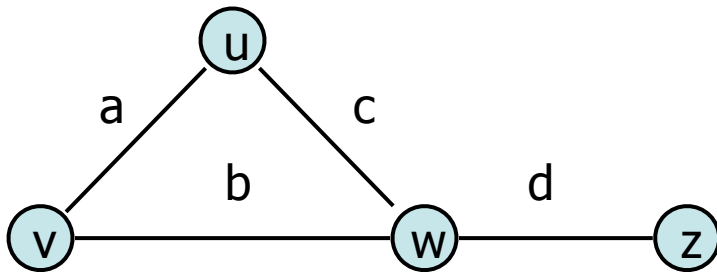
A) Lists



B) Trees



C) Graphs



D) Hash Tables

0	Apple
1	$\emptyset$
2	Pear

Note: Not every tree / graph / hash is actually a list :)

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?)



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):

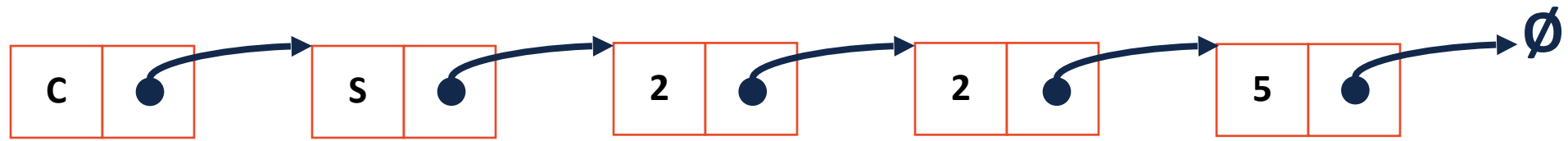
1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations

1.

2.

Linked List



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

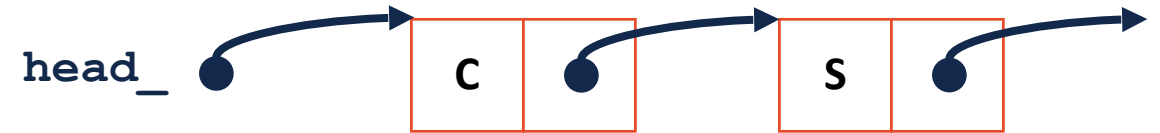
Why is **data** stored as a reference?

Why is **next** a pointer?

List.h

```
1  #pragma once
2
3
4  class List {
5  public:
6      /* ... */
28  void insertAtFront(const T& t);
29
30  private:
31      class ListNode {
32          T & data;
33          ListNode * next;
34          ListNode(T & data) :
35              data(data), next(NULL) { }
36      };
37
38      ListNode *head_;
39
40      /* ... */
...
... };
...
79
```

How do I access list given head_?



List.h

```
1  #pragma once
2
3
4  class List {
5      public:
6          /* ... */
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28      void insertAtFront(const T& t);
29
30      private:
31          class ListNode {
32              T & data;
33              ListNode * next;
34              ListNode(T & data) :
35                  data(data), next(NULL) { }
36          };
37
38          ListNode *head_;
39
40          /* ... */
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```

What is missing in this code?

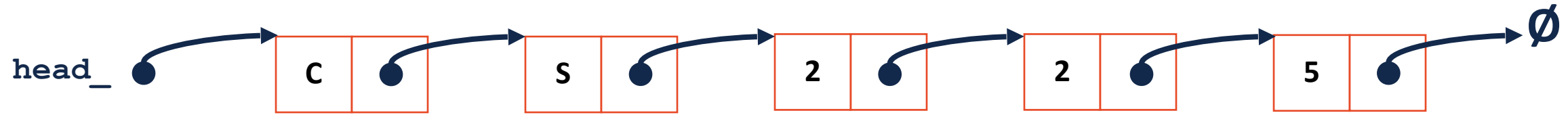
List.h

```
1  #pragma once
2
3  template <typename T>
4  class List {
5      public:
6          /* ... */
7
28     void insertAtFront(const T& t);
29
30     private:
31         class ListNode {
32             T & data;
33             ListNode * next;
34             ListNode(T & data) :
35                 data(data), next(NULL) { }
36         };
37
38         ListNode *head_;
39
40         /* ... */
41
...     };
...
79 #include "List.hpp"
```

List.hpp

```
1
2
3
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 }
```

Linked List: insertAtFront(data)



List.h

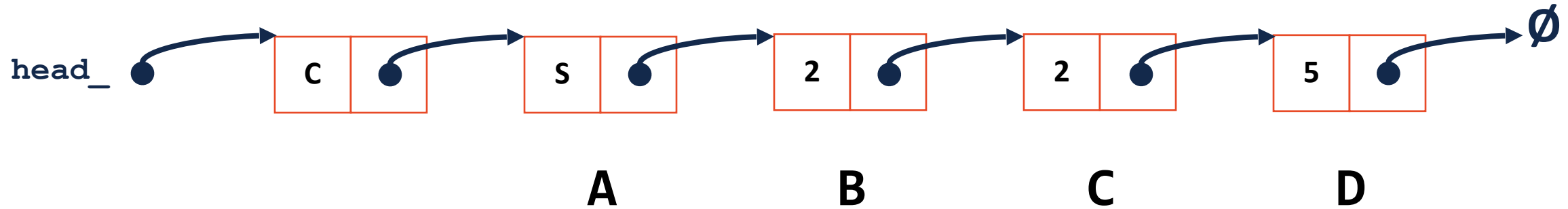
```
1 #pragma once
2
3 template <typename T>
4 class List {
5     public:
6         /* ... */
7
8     private:
9         class ListNode {
10             T & data;
11             ListNode * next;
12             ListNode(T & data) :
13                 data(data), next(NULL) { }
14
15         };
16
17         ListNode *head_;
18
19         /* ... */
20
21 };
22
23 ...
24
25 ...
26
27
28 #include "List.hpp"
```

List.hpp

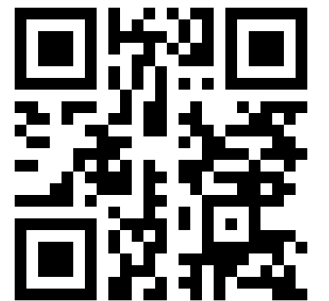


```
1
2
3 template <typename T>
4 void List<T>::insertAtFront(const T& t)
5 {
6
7     ListNode *tmp = new ListNode(data);
8
9
10    tmp->next = head_;
11
12
13    head_ = tmp;
14
15 }
16
17
18
19
20
21
22
```

Linked List: insert(data, index) **insert(d, 3)**

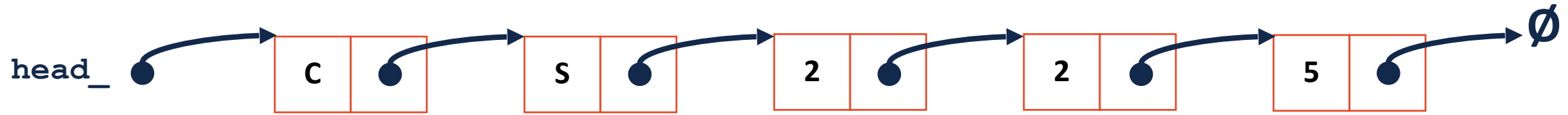


To insert a new ListNode at index 3, we need to **modify** which node?



Join Code: 225

Linked List: `insert(data, index)` `insert(d, 3)`



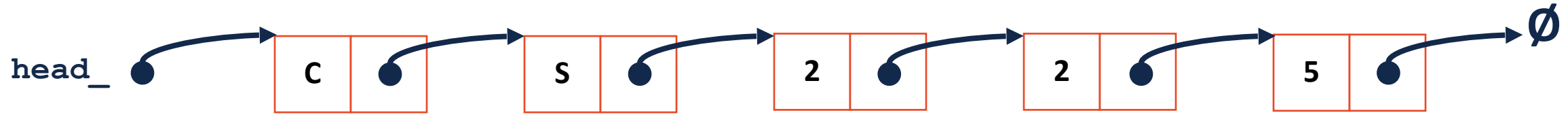
1) Get access to node @ position index - 1

We ***could*** code up a solution to insert which uses some previous var

But lets be smarter!

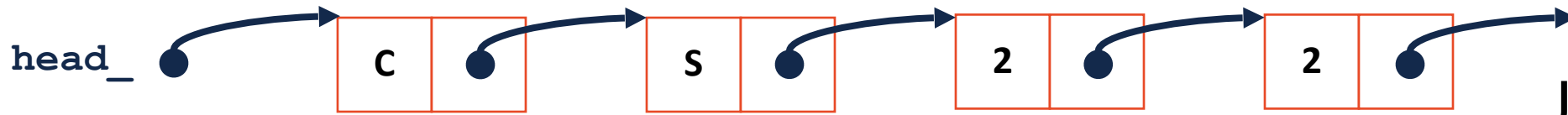
Coding tip from earlier lecture: **Consider the entire interface**

Linked List: `_index(index)`



Lets write one function which is useful for insert / remove AND find

Linked List: `_index(index)`



Join Code: 225

What should the return type of `_index()` be?

[template <class T>]

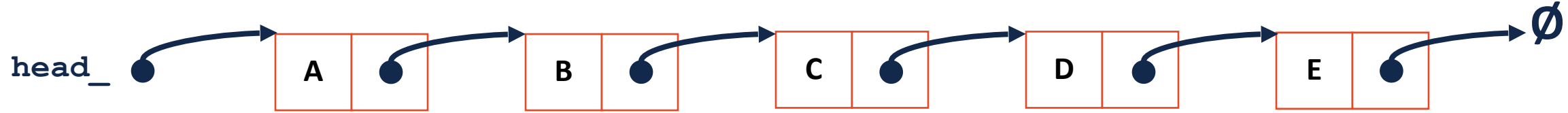
(A) T &

(B) ListNode

(C) ListNode *

(D) ListNode *&

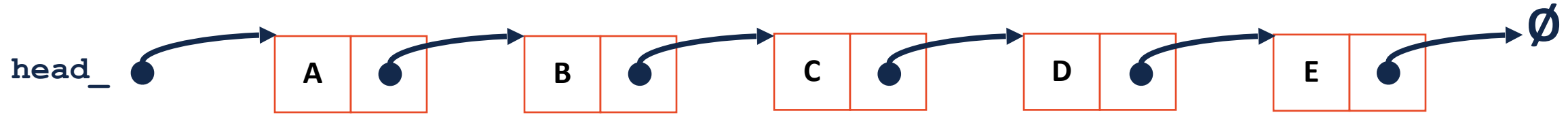
Comparing pointer to reference-to-pointer



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

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

Comparing pointer to reference-to-pointer



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

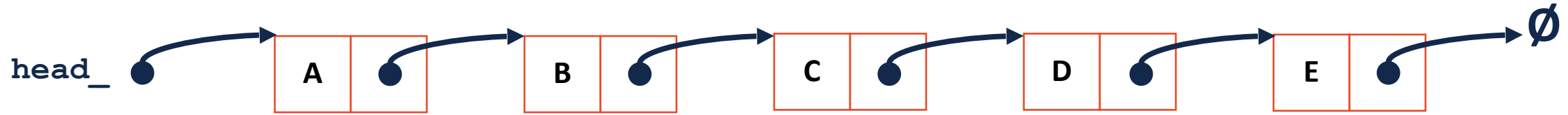
We can access **curr->data** and **curr->next** but not **previous.next**

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

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

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

Comparing pointer to reference-to-pointer



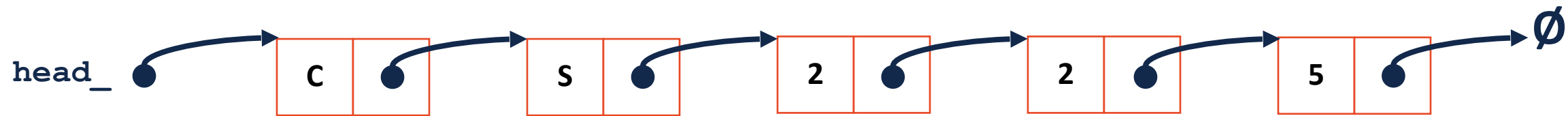
```
ListNode * curr = _index(3) ;
```

```
curr = new ListNode(x) ;
```

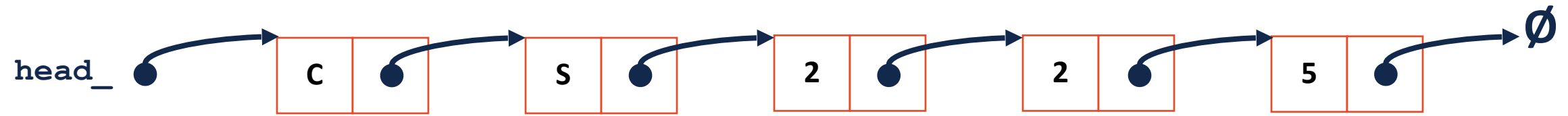
```
ListNode *& curr = _index(3) ;
```

```
curr = new ListNode(x) ;
```

LinkedList *&_index(index)



Linked List: `_index(index)`



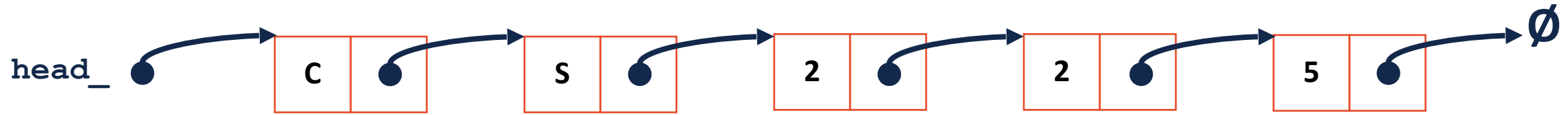
A brief tangent...

List.hpp

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

```
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
74
75
76
77
78
79
80 }
```

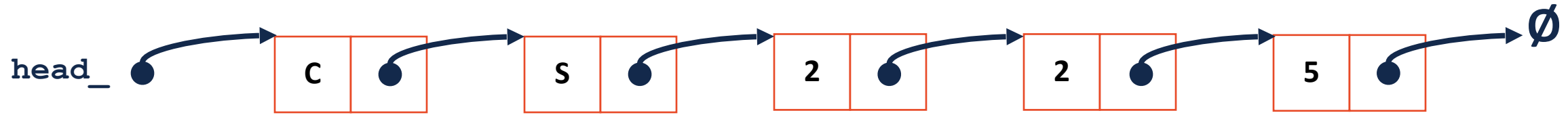
Linked List: insert(data, index)



1) Get reference to previous node's next

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

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

What is the Big O of insert?

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

Next Time: List Random Access []

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

What return type should this function have?