Data Structures

Lists and List ADT

CS 225

Brad Solomon

January 26, 2026

Department of Computer Science

Enrolling in CS 199-225

Zoom link is on Discord

James Scholar Enrollment:

1. Register for CS 225 section AH (CRN 70578)
2. Fill out the appropriate Honors Credit Learning Agreement

Check your individual college for James Scholar deadlines

Standard Enrollment:

1. Register for CS 199-225 (CRN 61879)

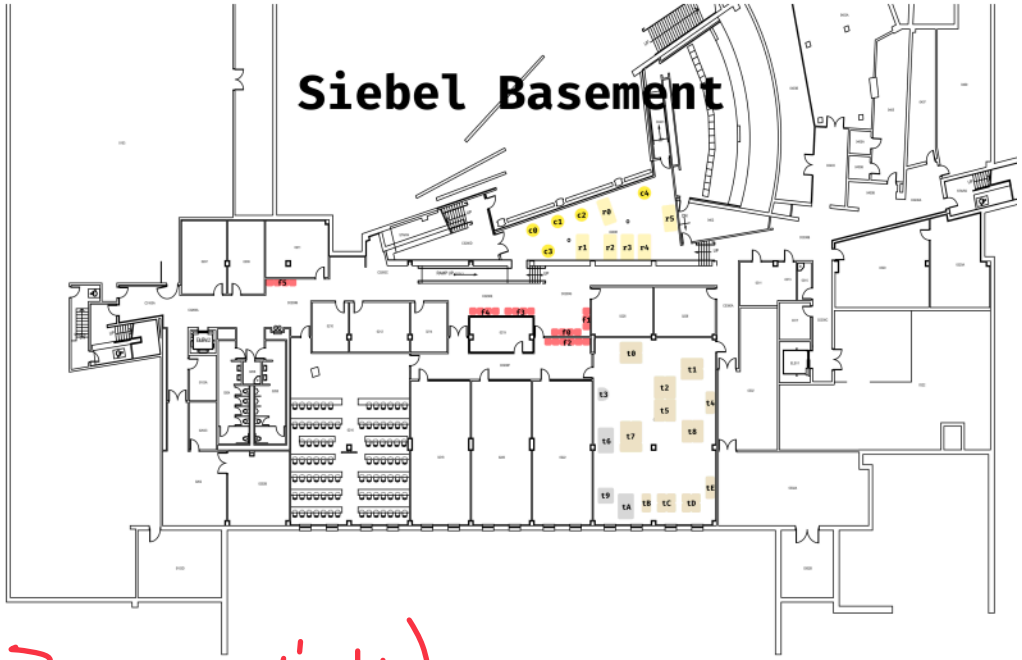
This is a pass-fail grade and a 0-credit course

Office Hour Etiquette

Schedule and link to queue on the website

Pay attention to the rules!

1. Be in Siebel Basement or online in bad weather
2. Tag questions
3. Ask **one** specific question
4. Include a specific location (Zoom link)
5. Include both your name and Discord ID



MP_Stickers Released

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

HSLA Pixel docs

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
```

Image.h

test_invert<T1_Image>
→ True/false

CT2_Image)

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/sp2026/exams/

Learning Objectives

Interface

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 - Ownership

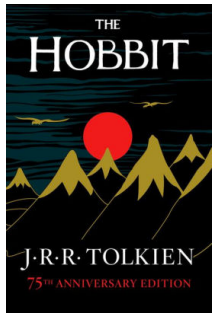
Pass by **Value:**

A local copy of the original

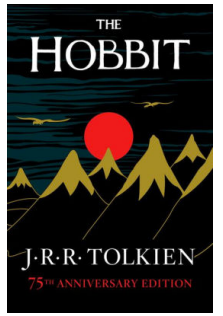
Ex: addBook(Book book)

local stack

Copy



Original



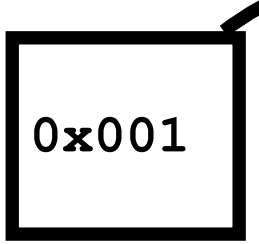
Pass by **Pointer to Value:**

An address on the heap

Ex: addBook(Book* book)

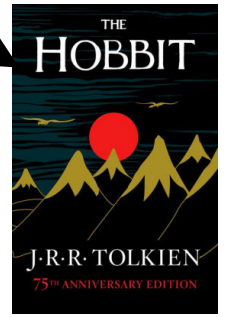
local stack pointer

P



value

value points to orig



Pass by **Reference:**

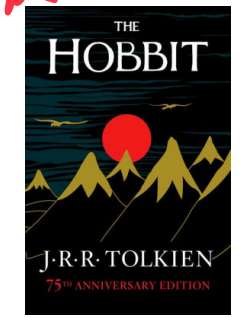
An *alias* to an existing variable

Ex: addBook(Book& book)

Local var Y ***is*** X

new name

X=



Templates

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


```
T sum (T a, T b) {
    return a+b;
}
```

```
int sum(1, 3)
float sum(2.2, 1.7)
```

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){
...
}
```

main.cpp

Sum(1,3)

int

generate

int Sum(int a, int b)

Sum<float>(1,3) → float Sum(float a, float 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
```

T is not specific, user choice (Q, myType)

Templates are very useful!

vector<int>

vector<float>

vector<string>

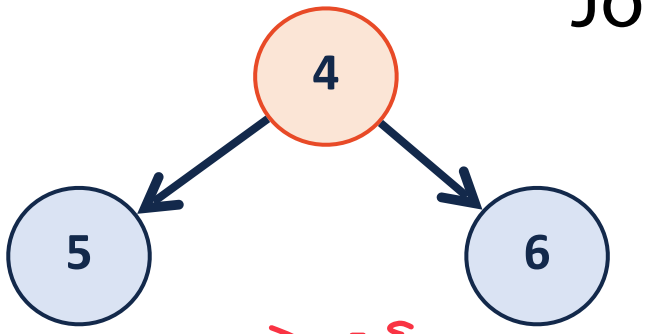
What is your favorite data structure?

Join Code: 225

A) Lists

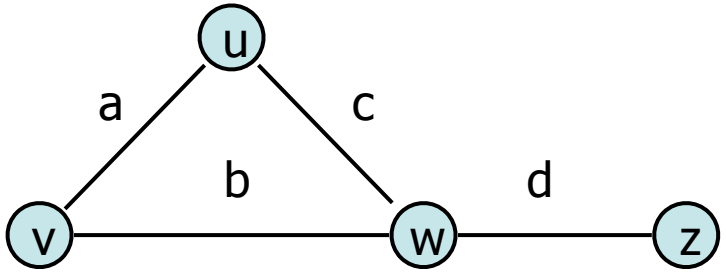
25%

B) Trees



35%

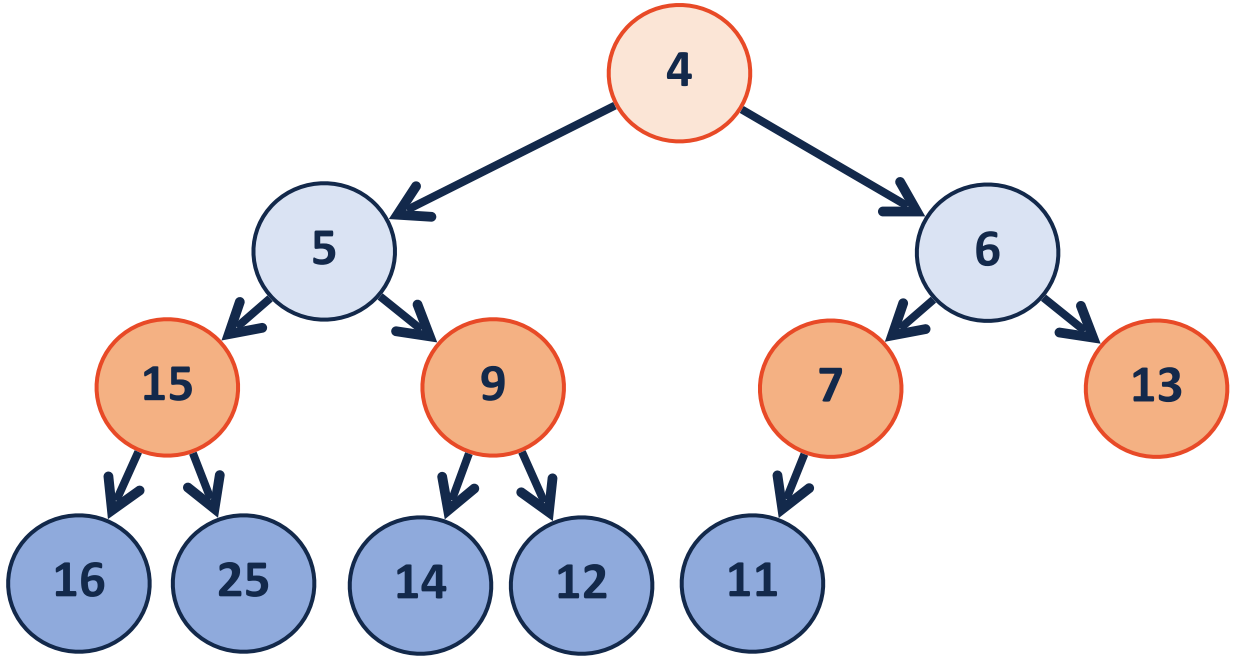
C) Graphs 22%



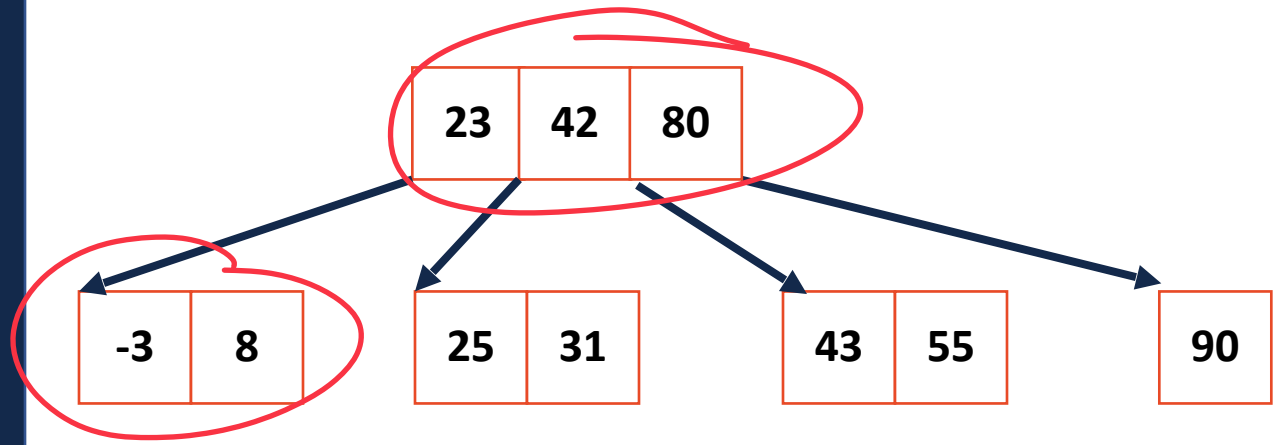
D) Hash Tables 18%

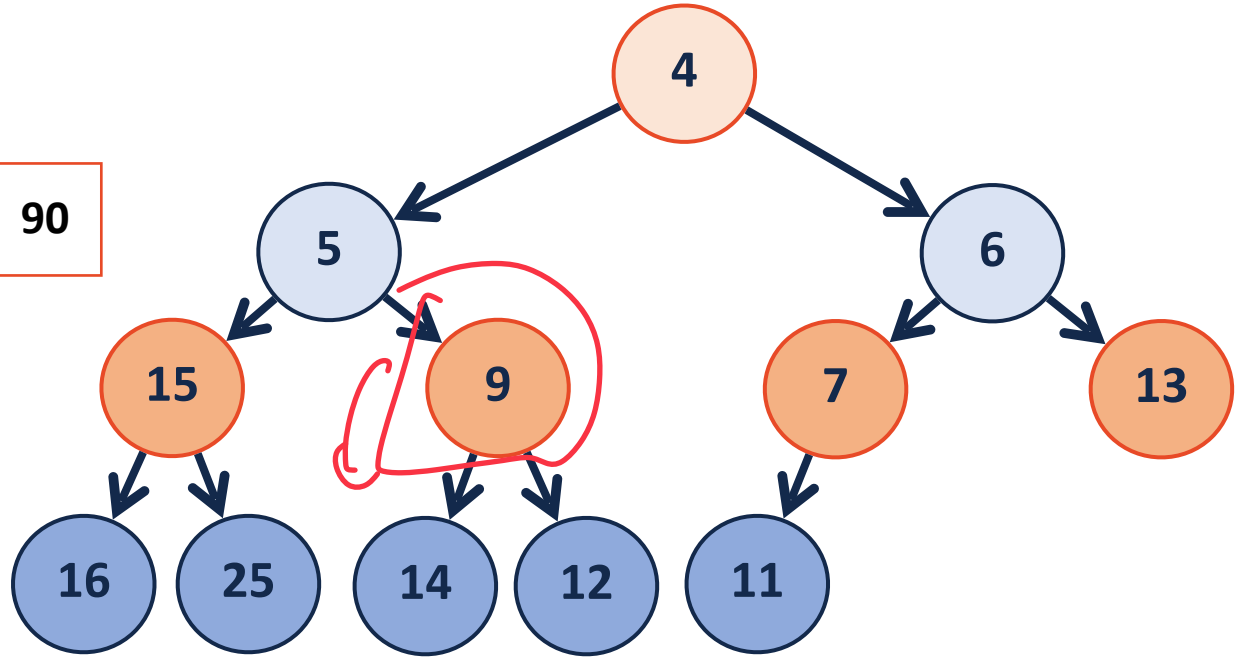
0	Apple
1	∅
2	Pear

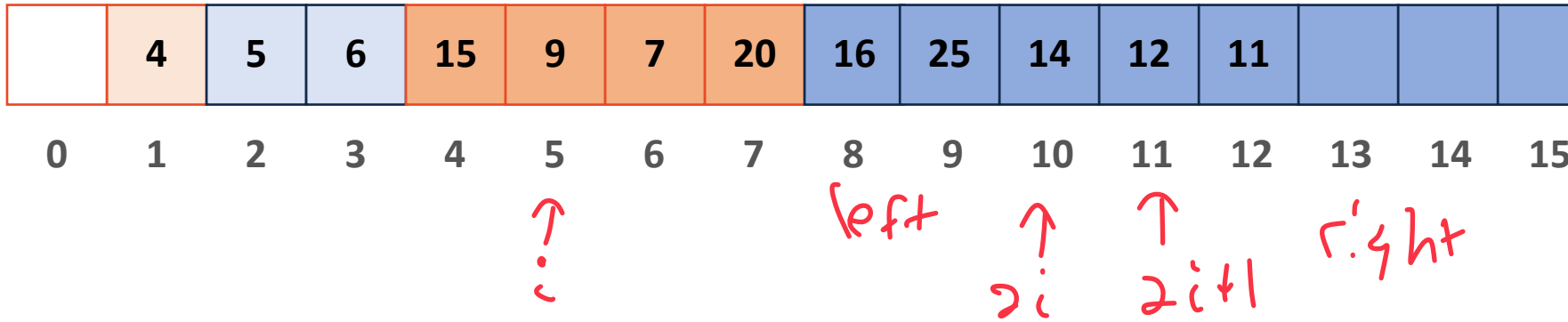
Your favorite data structure: Trees



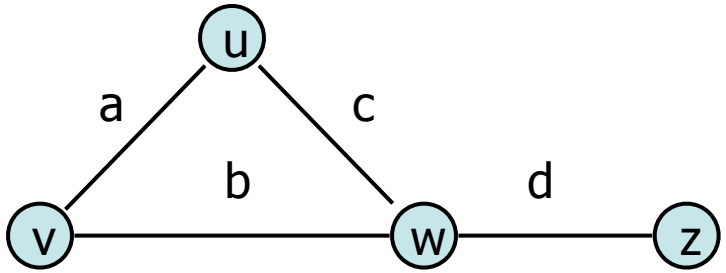
Your favorite data structure: ~~Trees~~ Lists

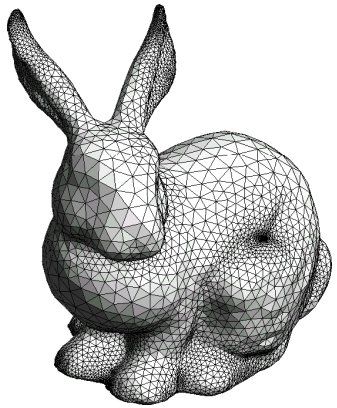


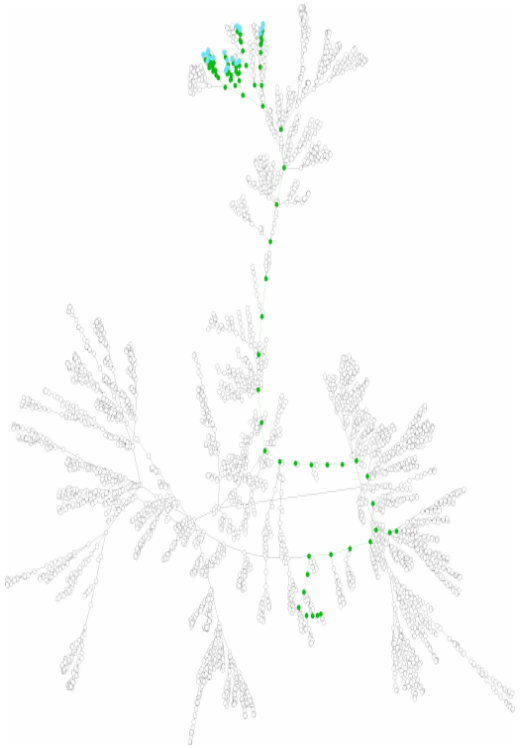


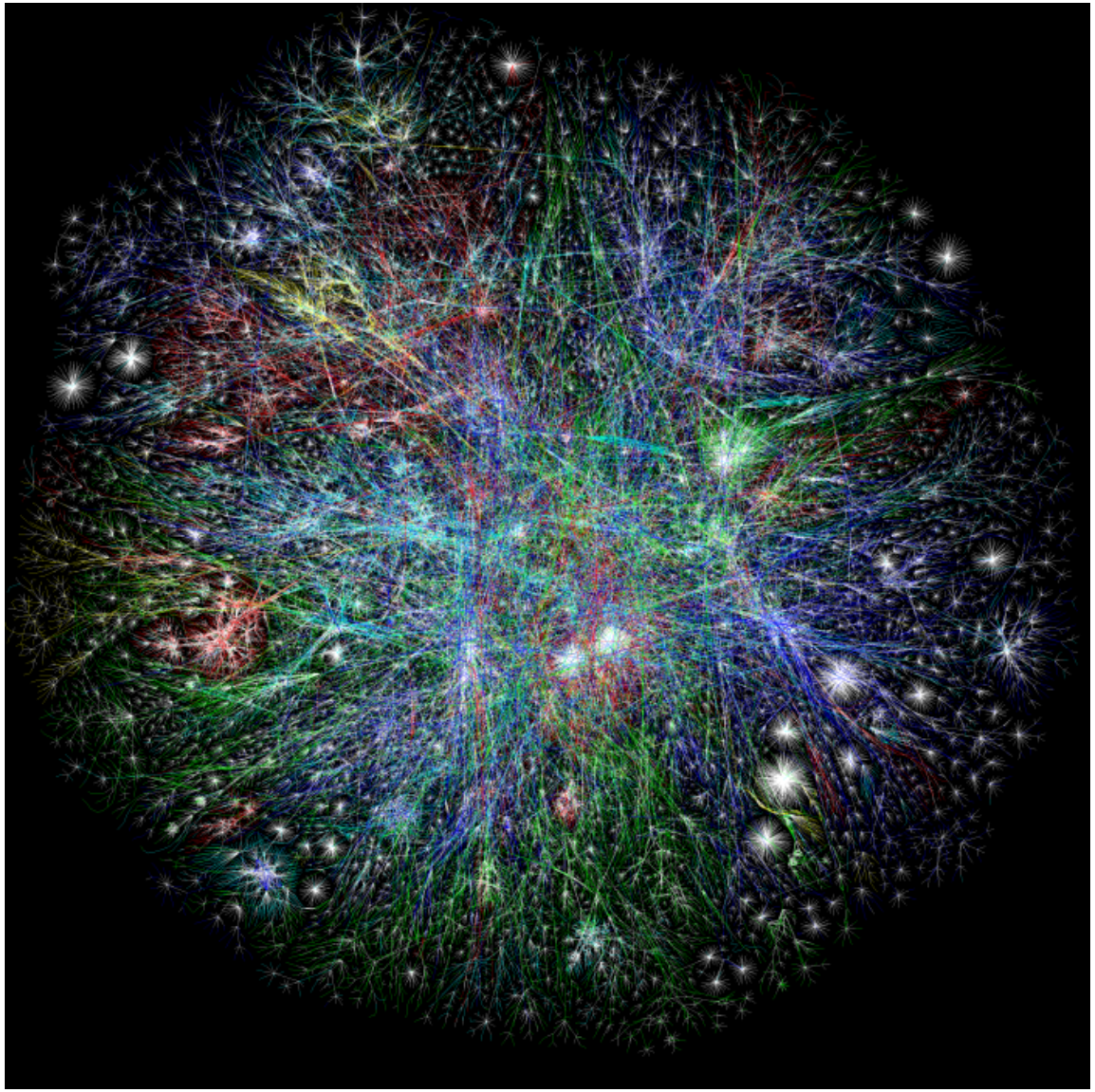


Your favorite data structure: Graphs

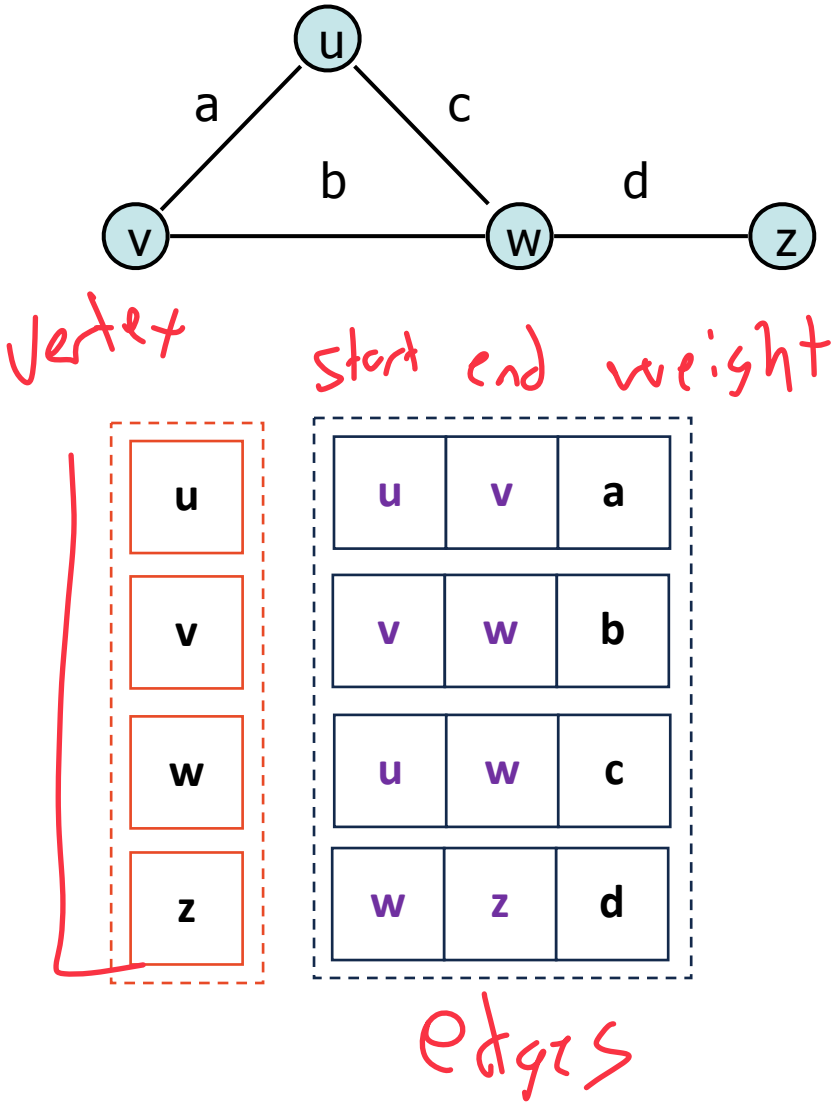




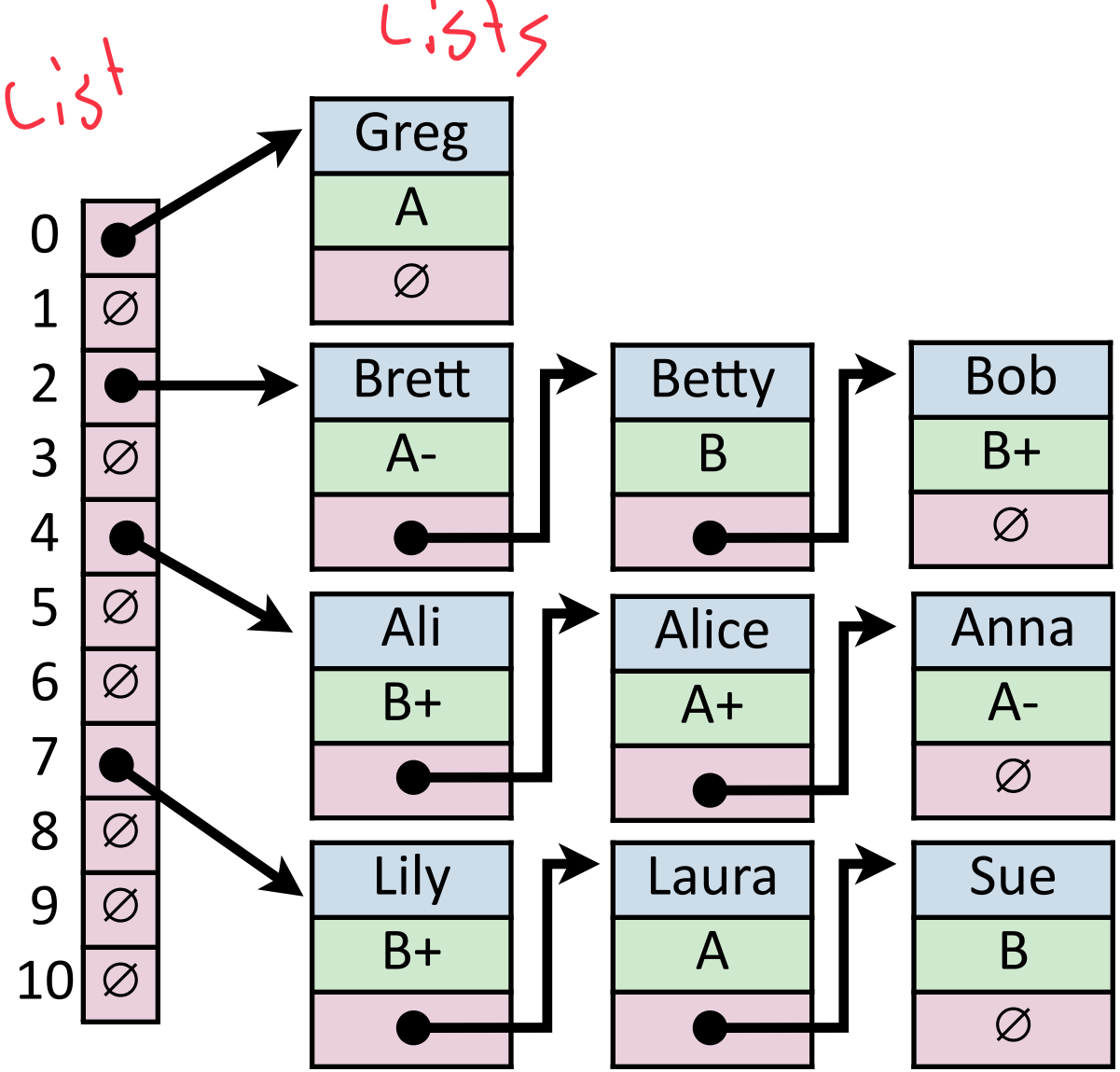




Your favorite data structure: ~~Graphs~~ Lists



Your favorite data structure: Hash Tables



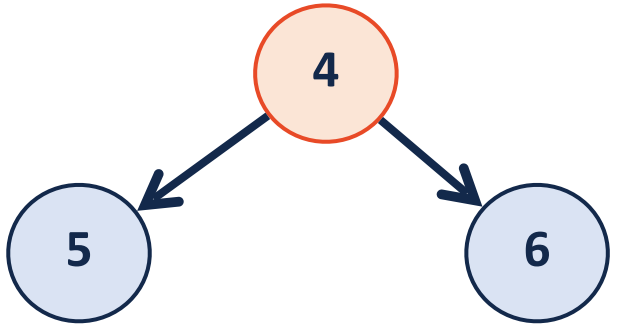
$$H = \{h_1, h_2, \ldots, h_k\}$$

1
0
1
0
1
0
0
1
0
0
0

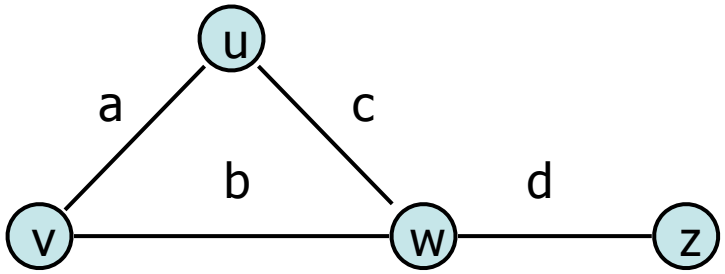
So 100% of people are excited about lists!

A) Lists

B) Trees



C) Graphs



D) Hash Tables

0	Apple
1	∅
2	Pear

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

Abstract Data Types

Interface

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?)

Store information (in an order)

↳ insert / push_back

Some data structures don't *

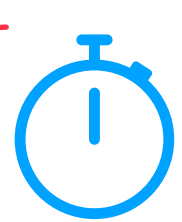
↳ remove / POP

↳ Access Data / Find

List ADT

this is not sorted

can have sorted list but can also have unsorted



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

what is order

[3, 1, 2]

3 is at position 0

(Fixed pos for each item)

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

diff types

all same type

more common

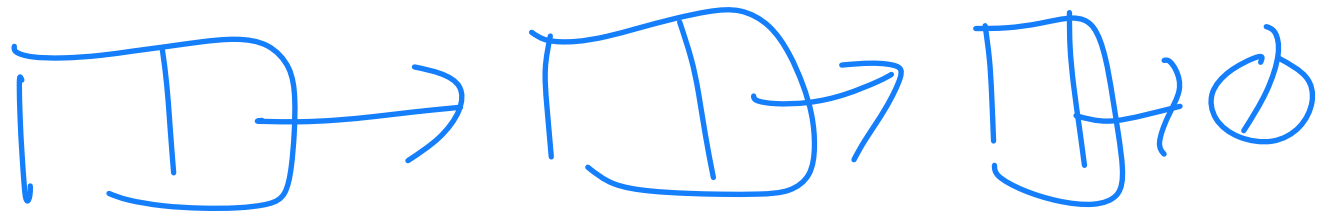
The list can be of a **fixed size** or is **resizable**

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

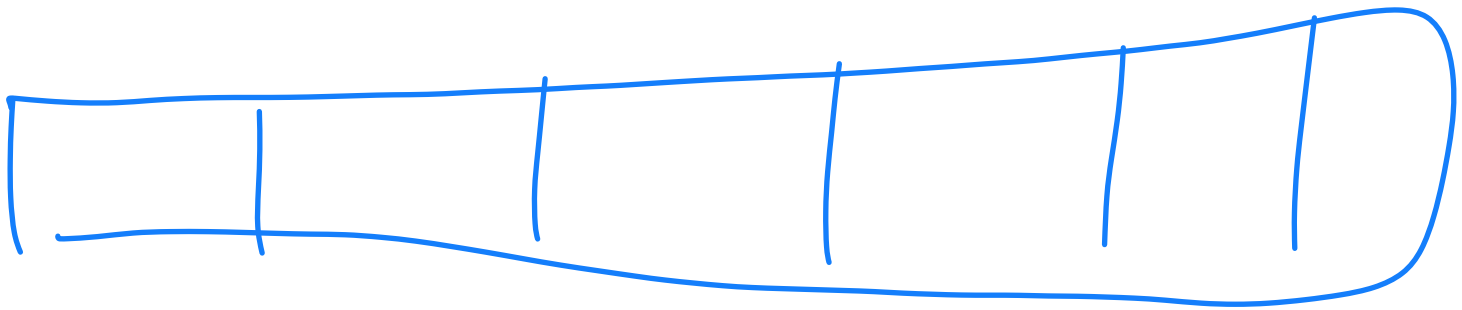
1. Insert
2. Delete
3. isEmpty ← size equivalent
4. getData (access)
5. Create an empty list ← Constructor

List Implementations

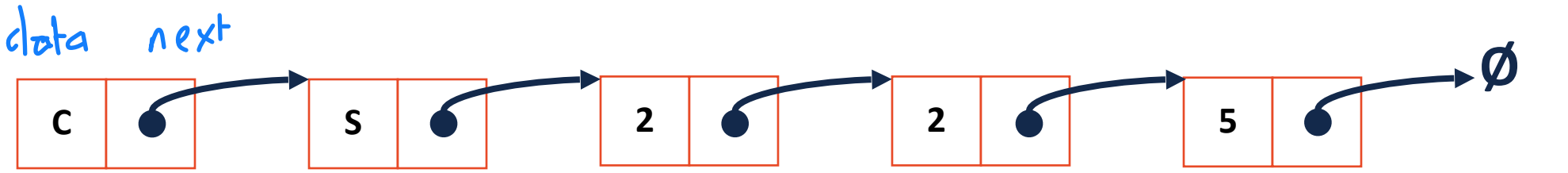
1. Linked List



2. Array



Linked List



This is a chain of objects of type ListNode

```
class ListNode{
    T & data;
    ListNode* next;

};
```

T * = & data; if data is type T &

List.h

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

← B/c T can be BIG don't use value

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

↳ what is the use case of data?

↙ style choice Pointer is OK ✓

(A stored value) [not value]

↳ b/c this matters copy/alias

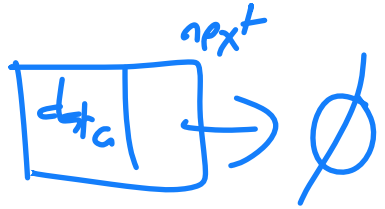
alias [↳ No pointer allocation needed
↳ An alias cannot be null ptr]

[always points to same object]

Why is **next** a pointer?

Must be pointer

↳ b/c it can be nullptr



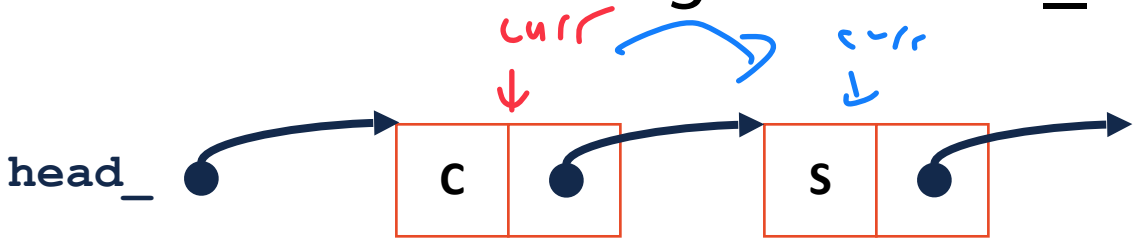
List.h

```
1   #pragma once
2
3
4   class List {
5     public:
      /* ... */
…
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
```

as a struct

front (first) item in List

How do I access list given head_?



ListNode * curr = head_;

curr = curr -> next;

Why curr needed?

↳ If we move head can never go backwards

(Next but not prev)

List.h

```
 1  #pragma once
 2
 3
 4  class List {
 5    public:
…      /* ... */
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
```

What is missing in this code?

List.h

```
1   #pragma once
2
3   template <typename T>
4   class List {
5     public:
      /* ... */
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  #include "List.hpp"
```

style choice

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  }
```

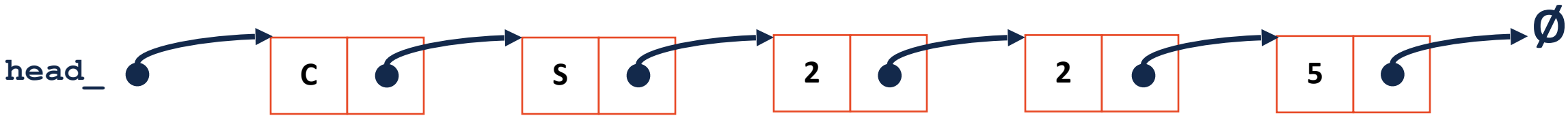
Templated functions in .hpp

Impl. of insertAtFront()

main.cpp

#include <iostream>

Linked List: `insertAtFront(data)`



Stopped here

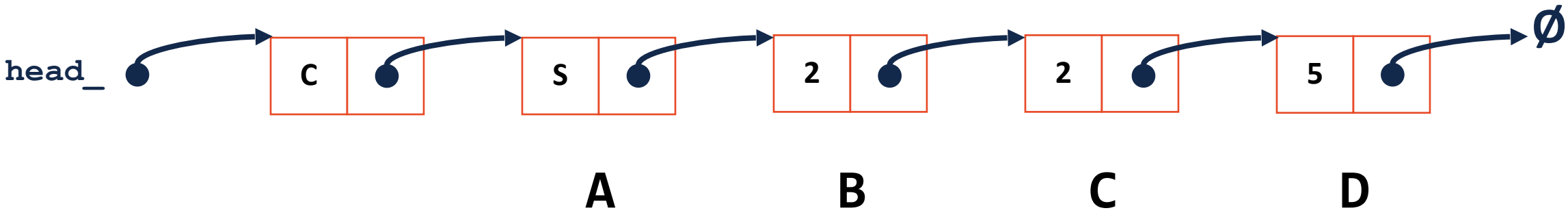
List.h

```
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"
```

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
```

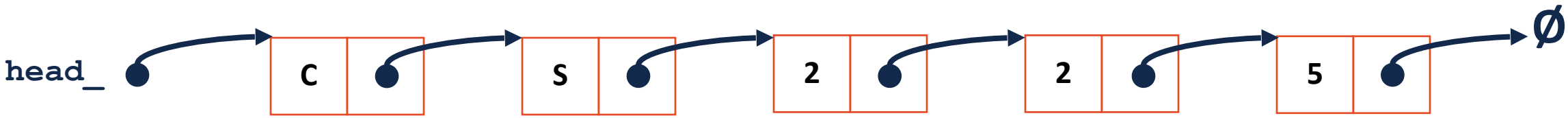
Linked List: `insert(data, index)` `insert(d,3)`



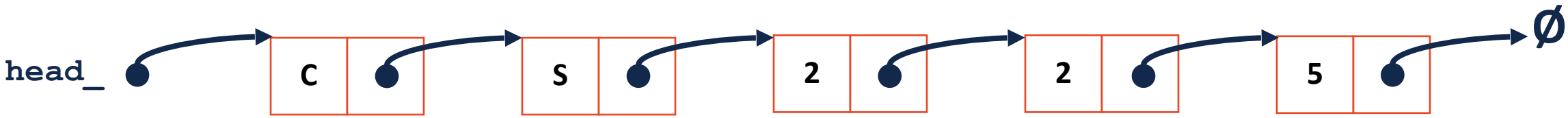
To insert a new ListNode at index 3, we need to **modify** which node?

Join Code: 225

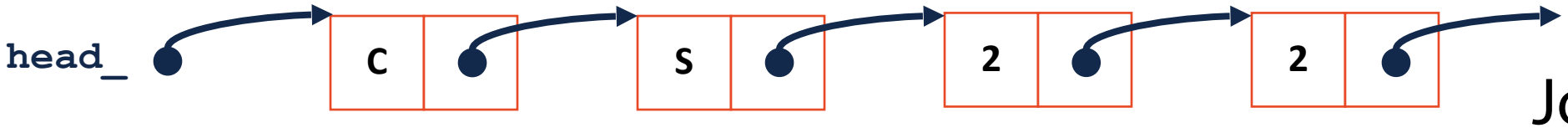
Linked List: `insert(data, index)` `insert(d,3)`



Linked List: `_index(index)`



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 *&`

```
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 }
```

List.hpp

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

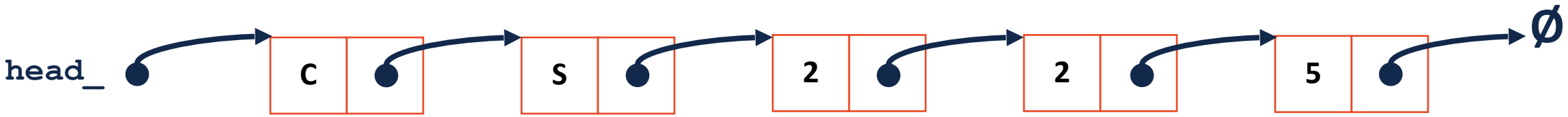
```
63 template <typename T>
64 typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66 if (index == 0 || node == nullptr){
67     return node;
68 }
69
70 return _index(index - 1, root -> next);
71
72
73 }
```

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

Which solution is better (iterative or recursive)?

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;
```

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;
}
```

Next Time: List Random Access [ ]

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

What return type should this function have?