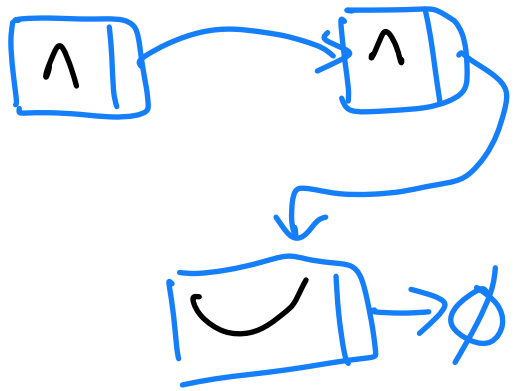
Data Structures

Lists and List ADT

CS 225

Brad Solomon

August 29, 2025



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

CS 225 Honors Class Registration Fixed

James Scholar Enrollment:

1. Register for CS 225 section AH (CRN 75824)
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 40944)

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

No class Labor Day (Monday September 1st)

Things to keep in mind for next week:

- Study for exam 0
- Complete POTDs for extra credit
- Set up your own machine!
- Join the CS 225 Discord!
  - Ask questions anonymously in class!
  - Request start of lecture music

Staff office hours start Tuesday**

Practice exam 0 Bonus video

Harsha @ Siebel 2322
4:30 - 6:00 PM
Today

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 - Ownership

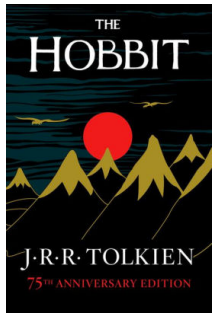
Pass by **Value:**

A local copy of the original

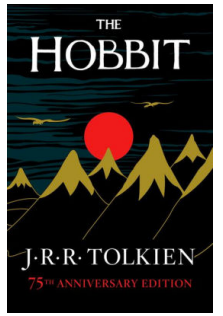
Ex: addBook(Book book)

Stack

Copy



Original



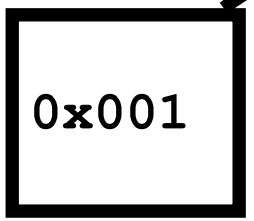
Pass by **Pointer to Value:**

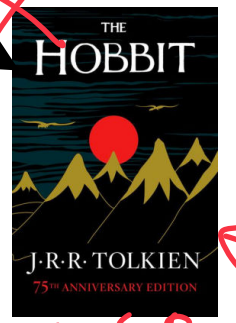
An address on the heap

Ex: addBook(Book* book)

Don't do this!

P





Owner of Book

Pass by **Reference:**

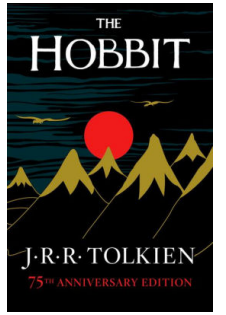
An *alias* to an existing variable

Ex: addBook(Book& book)

Local var Y

is X

X=



Templates

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


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

Templates

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

1. Templates are a recipe for code using generic types

Templates

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

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(2, 4);
```

```
int sum(int a, int b)
```

```
<float> sum (2, 4);
```

```
float sum (float a, float b)
```

Compiler generated code

Is whatever I want

template1.cpp h

```
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
```

Max (2, 4)

Max <float> (2.2, 4.1)

Max <Book> ← Crash

test <t1_ Image>

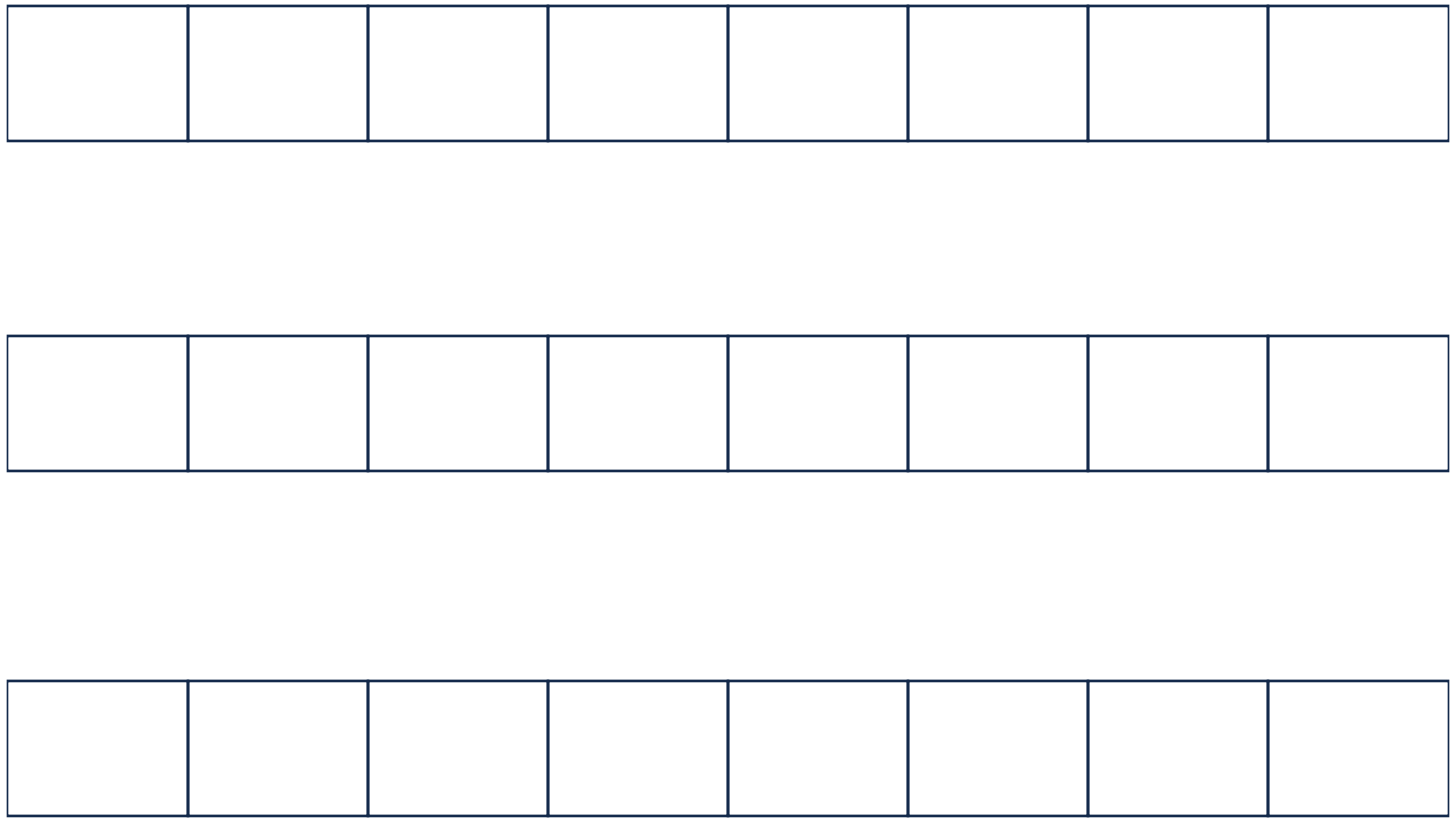
<t2_ Image>

. . .

- - -

Further justification to choose

Templates are very useful!

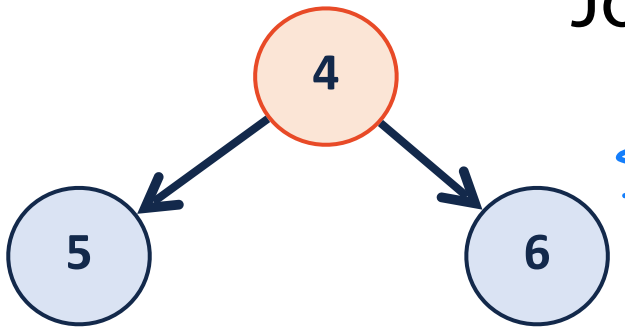


What is your favorite data structure?

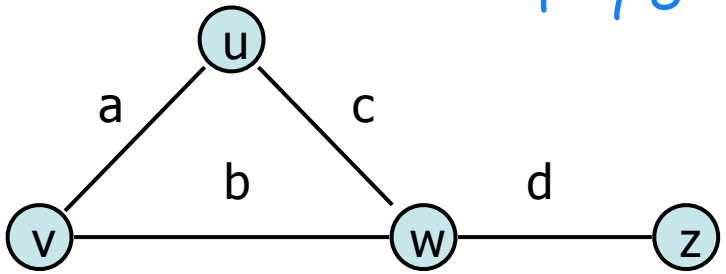
Join Code: 225

A) Lists 34%

B) Trees 35%



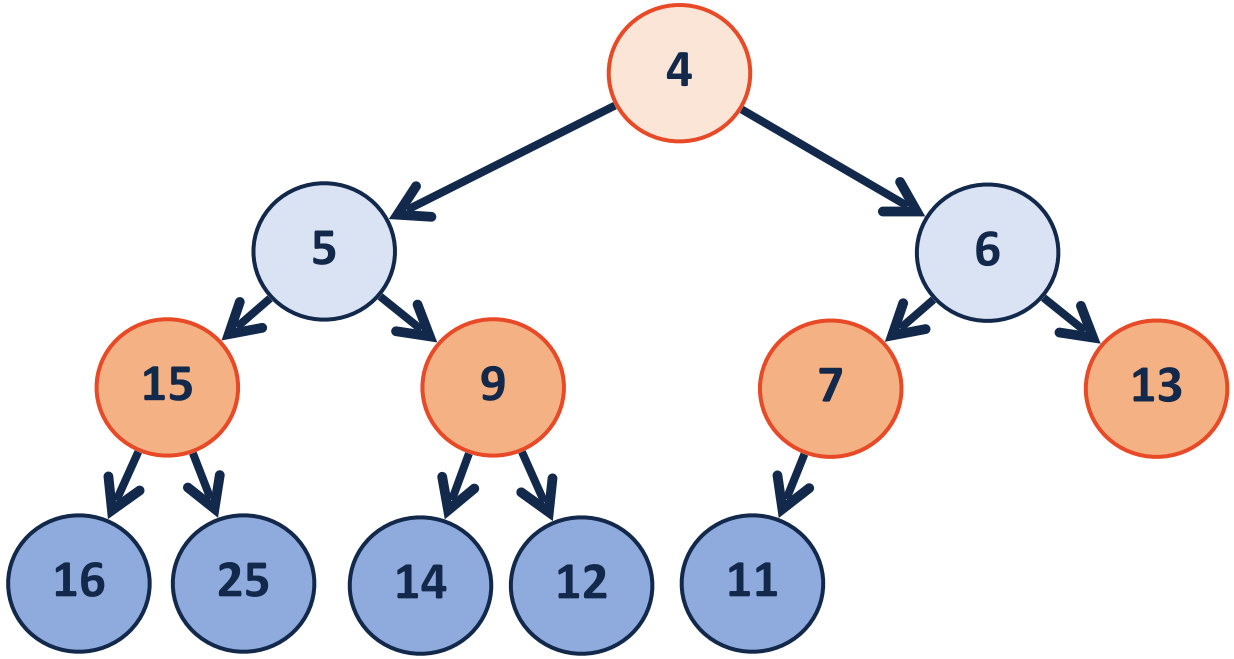
C) Graphs 11%



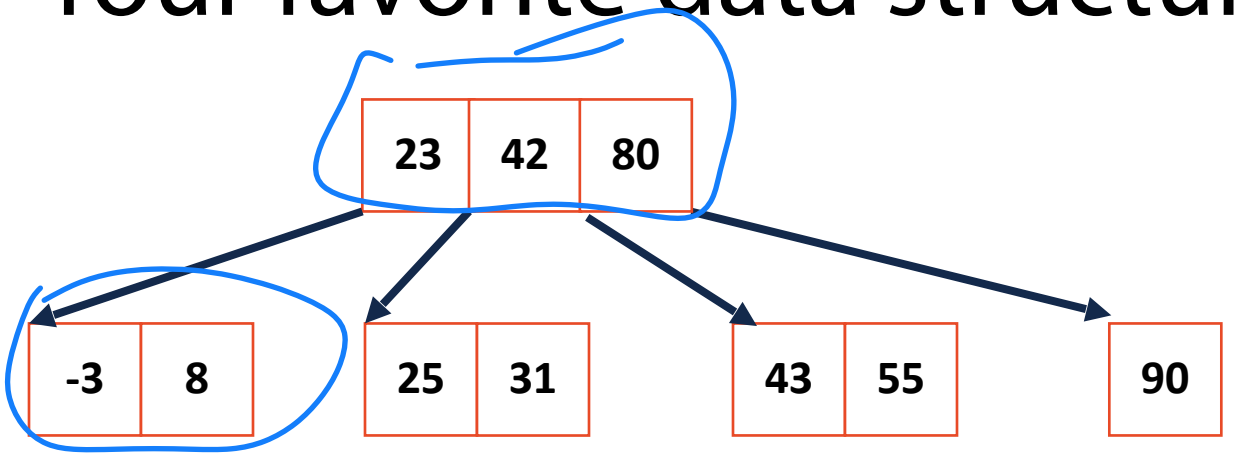
D) Hash Tables 21%

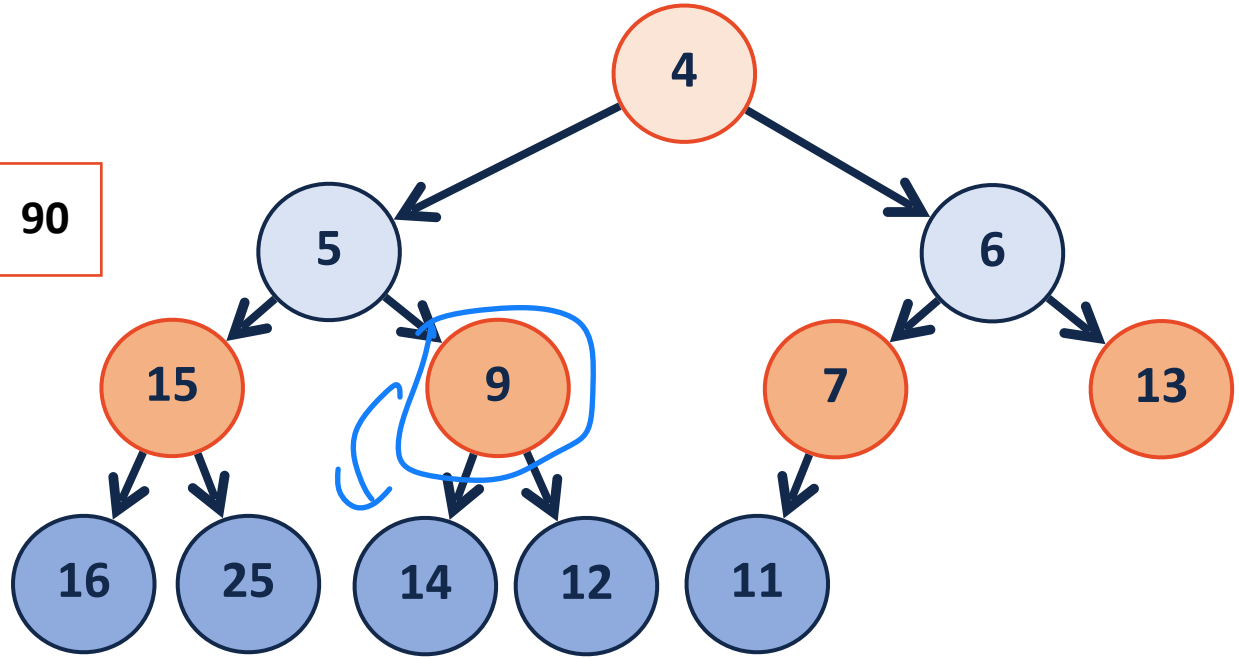
0	Apple
1	∅
2	Pear

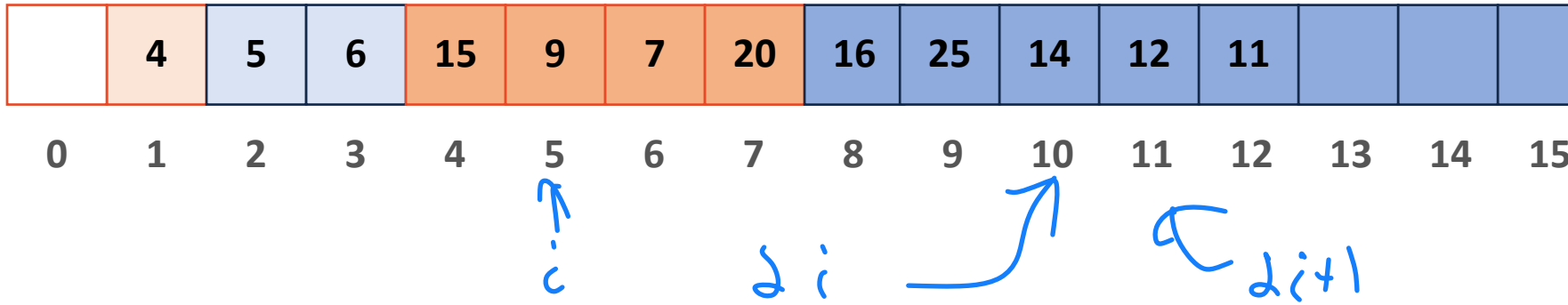
Your favorite data structure: Trees



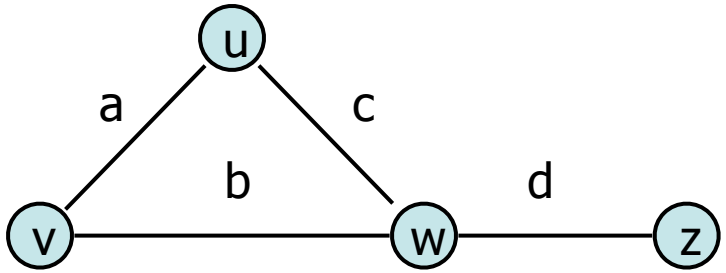
Your favorite data structure: ~~Trees~~ Lists

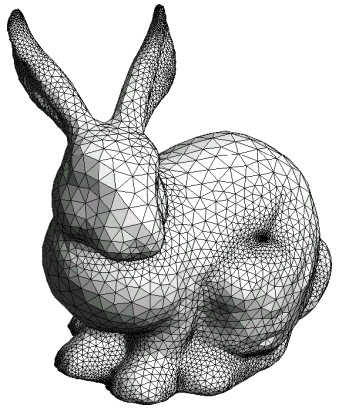


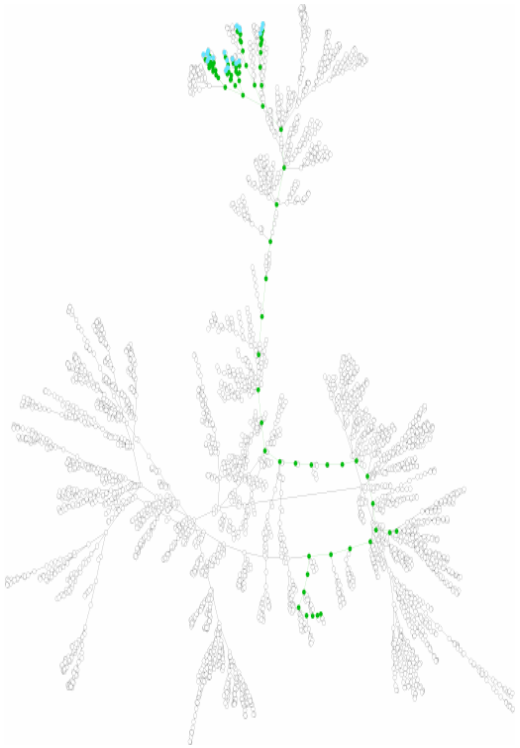


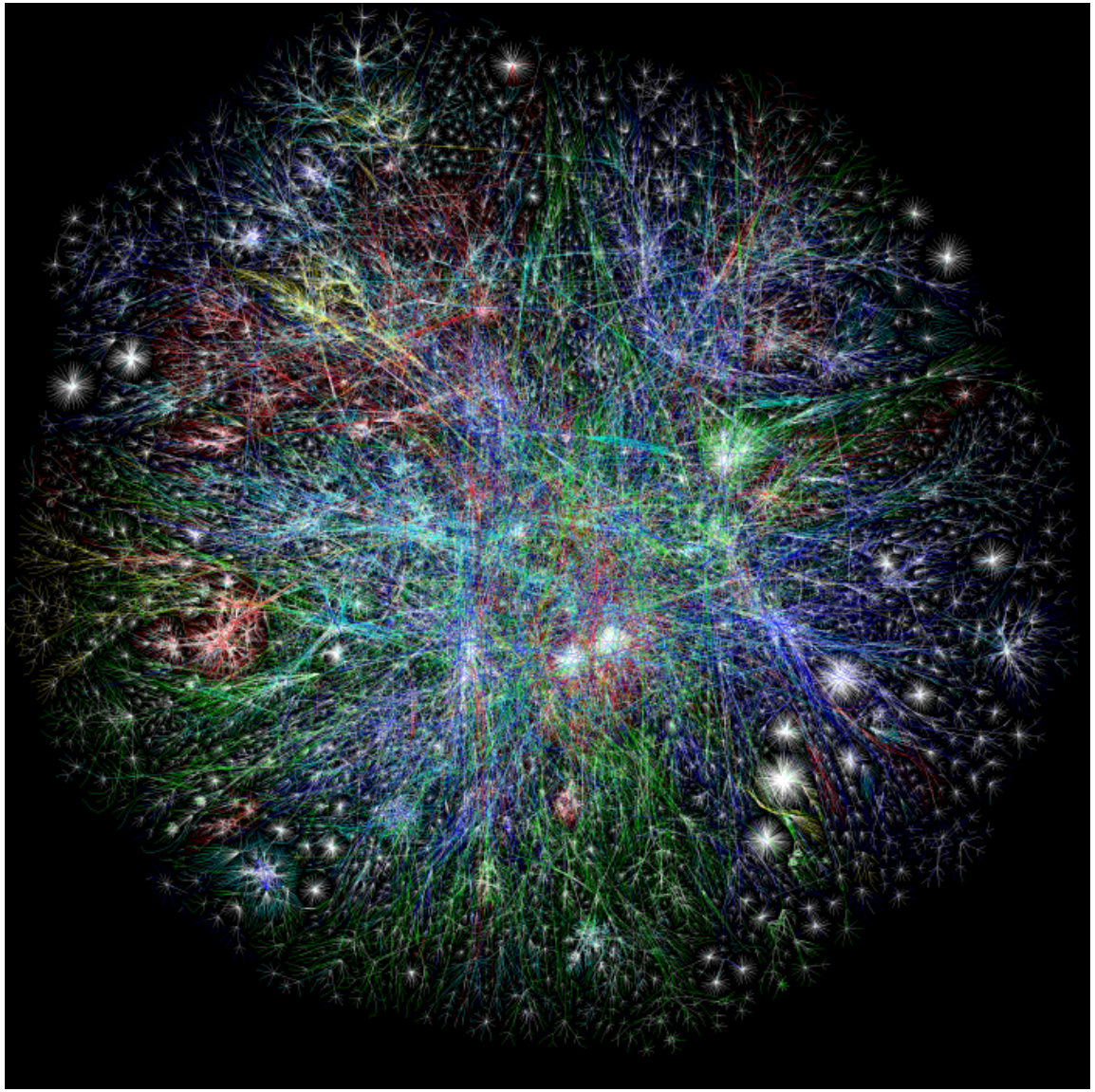


Your favorite data structure: Graphs

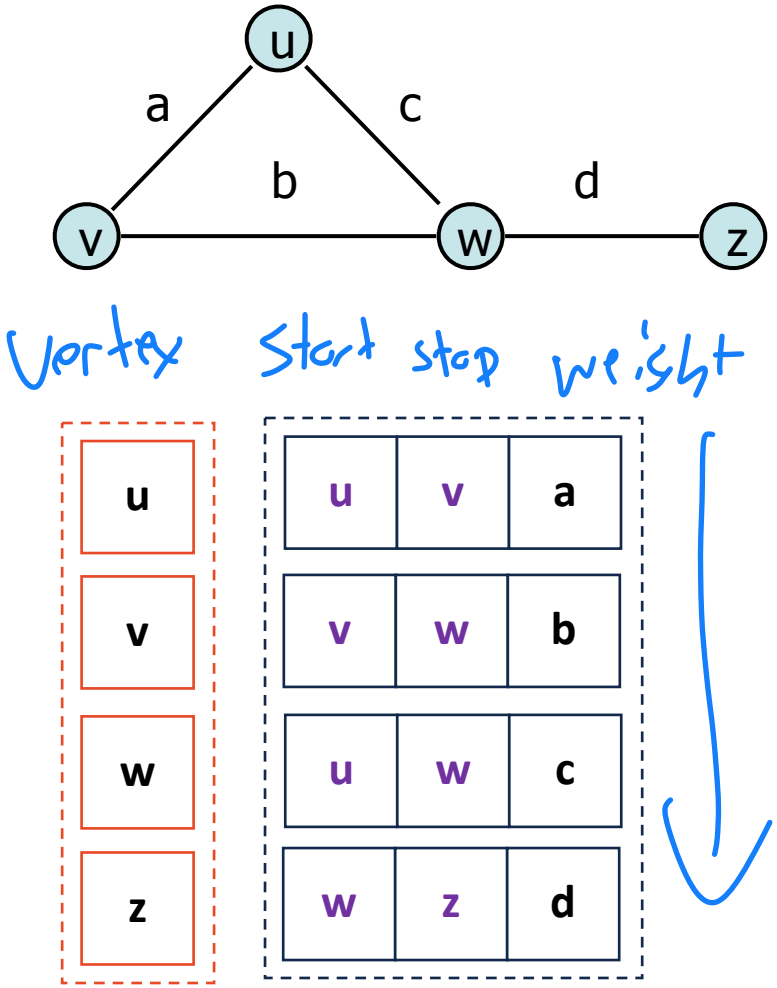




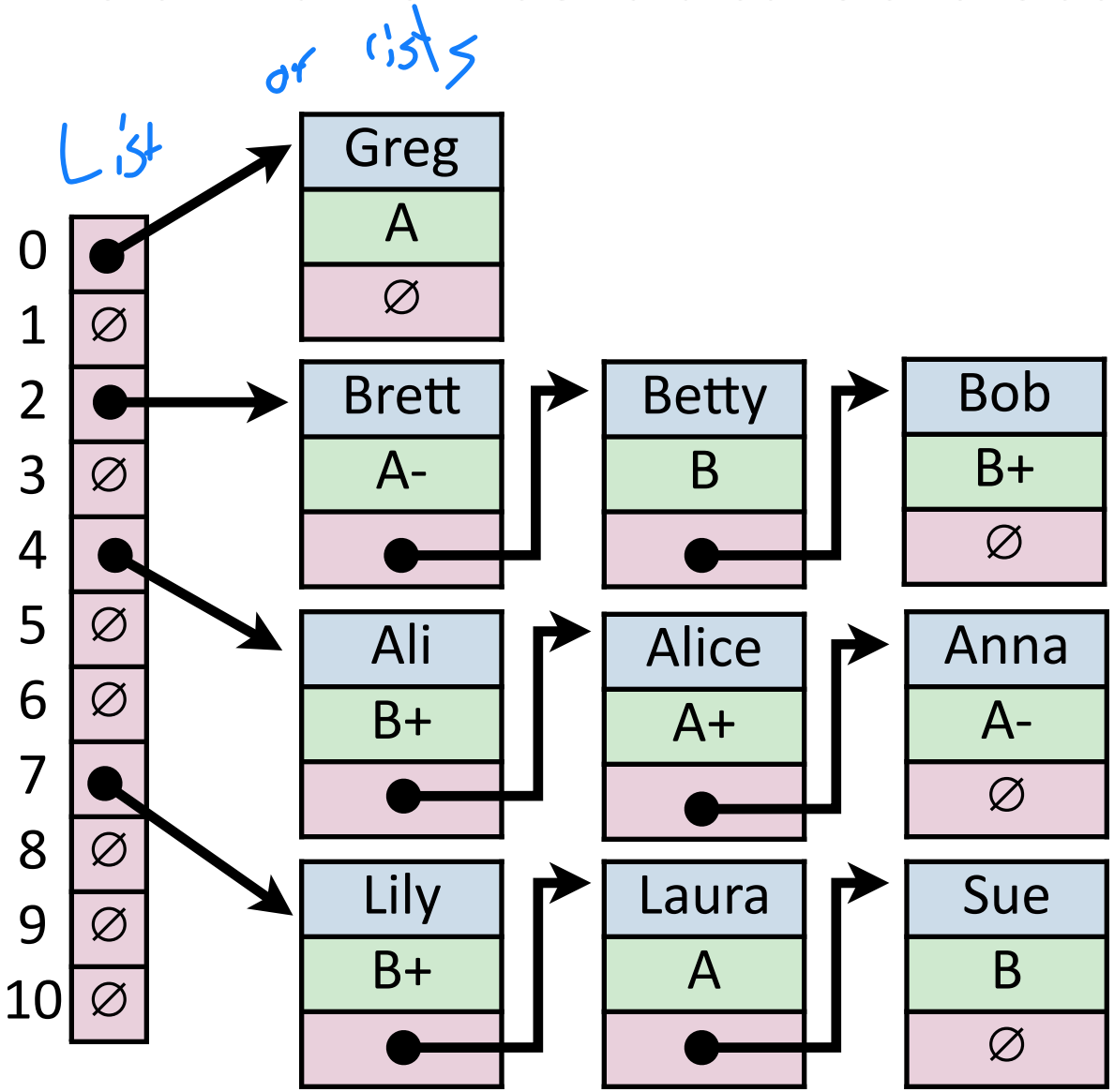




Your favorite data structure: ~~Graphs~~ Lists



Your favorite data structure: Hash Tables



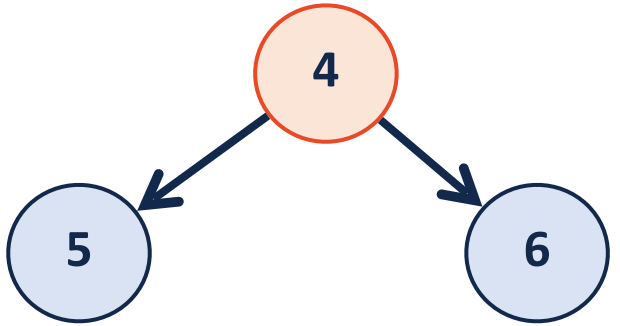
$$H = \{h_1, h_2, \ldots, h_k\}$$

1
0
1
0
1
0
0
1
0
0
0

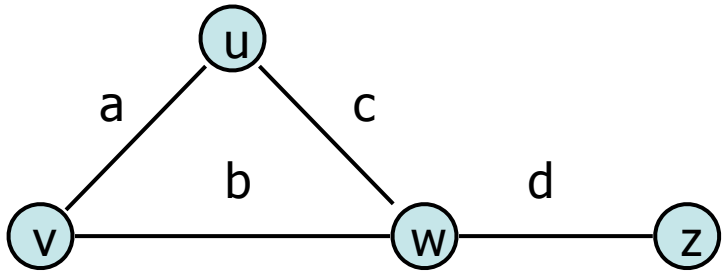
So 100% of people are excited about lists!

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

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?)

Insert() (anywhere in list)

Remove()

Find() (by value, by index, ...?)

Replace

Sort() <-> reverse()

Length()

Splice()

Construction

List ADT

Not the same as sorted

[5, 1, 3]

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

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

↳ mixed data type

↳ all same data type

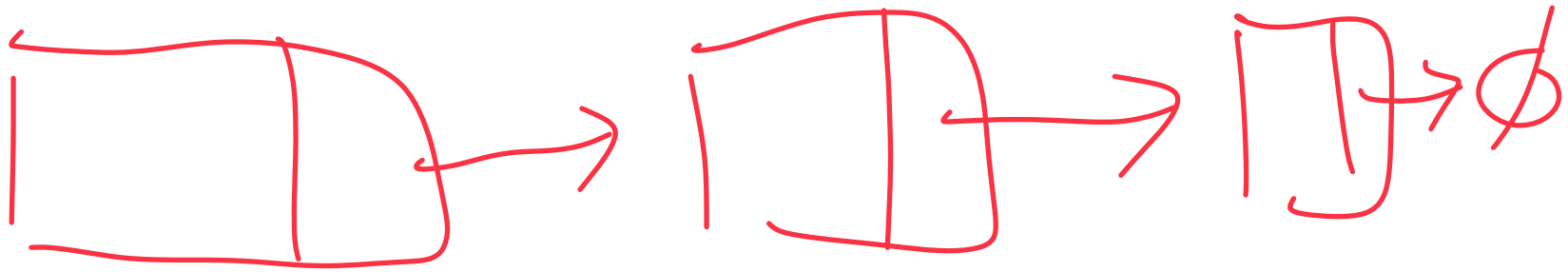
The list can be of a **fixed size** or is **resizable**

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

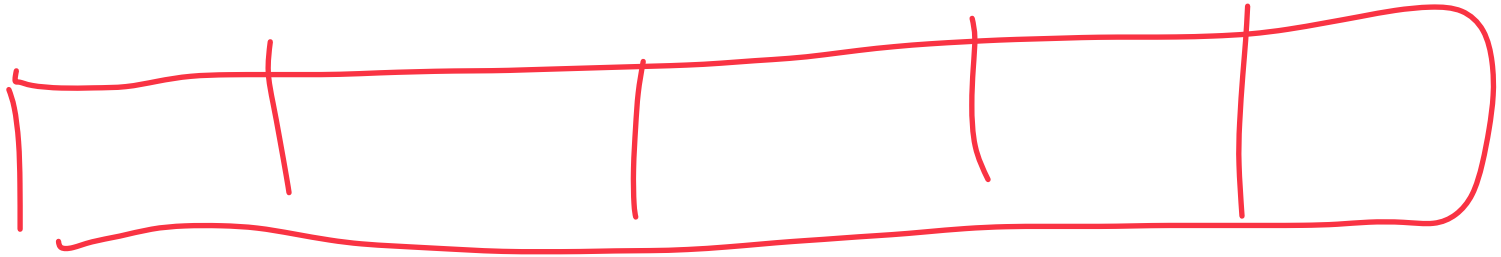
1. Insert
2. Delete
3. isEmpty
4. getData
5. Create an empty list

List Implementations

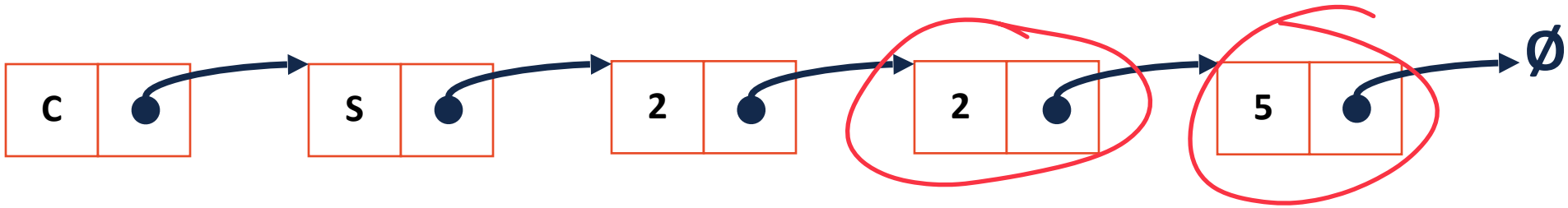
1. Linked List



2. Array List



Linked List

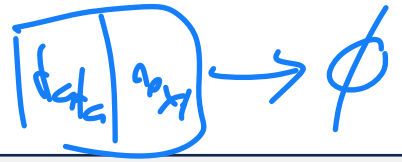


A list implemented using List Nodes

```
class ListNode {
    T& data;

    ListNode* next;

};
```



List.h

```
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?

Align
- ↳ We cannot change what we are pointing at (const pointer)
- ↳ Cannot be nullptr
- ↳ No pointer overhead

Why is next a pointer?

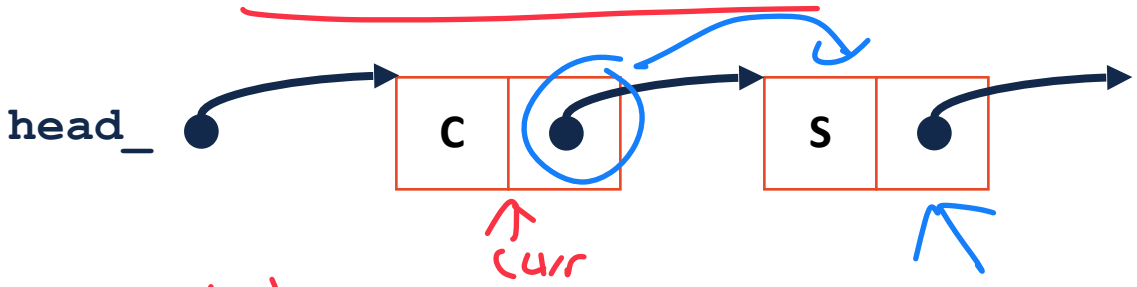
- ↳ we want to be able to change this!
- ↳ Can be nullptr! Default value

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
```

struct

How do I access list given head_?



ListNode* curr = head_;

curr = curr->next;

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?

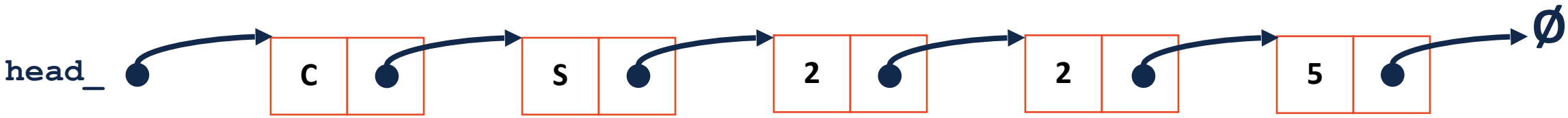
List.h

```
 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"
```

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)`



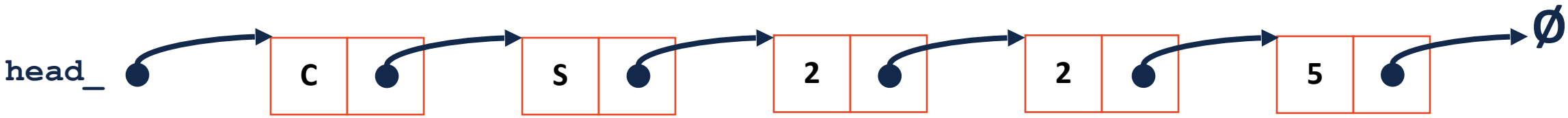
List.h

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

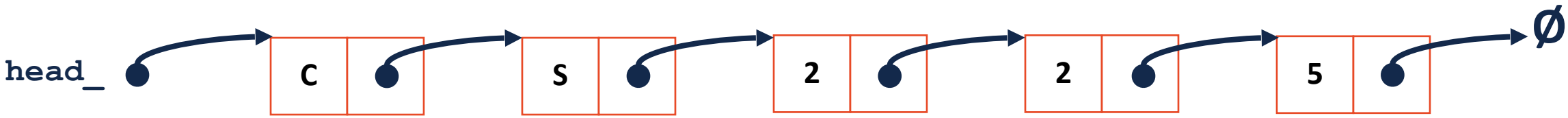
List.hpp

```
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
```

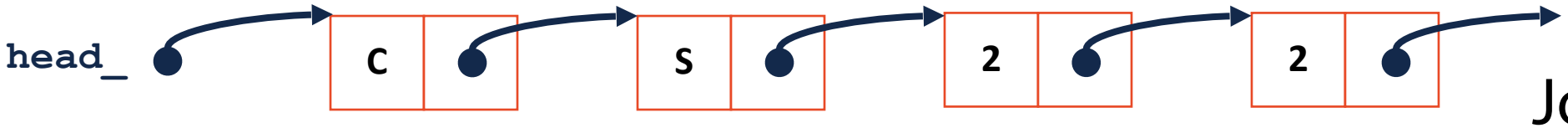
Linked List: `insert(data, index)`



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 }
```

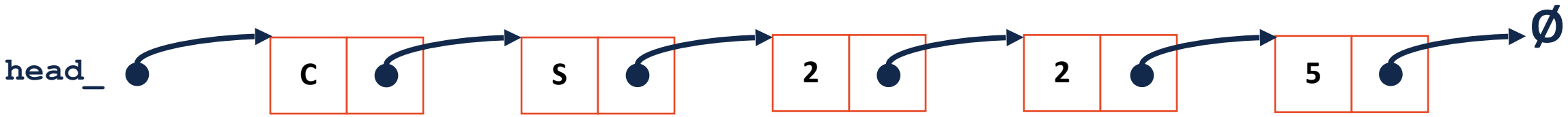
```
63 template <typename T>
64 typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){
65
66 if (index == 0 || node == nullptr){
67     return node;
68 }
69
70 return _index(index - 1, root -> next);
71
72
73 }
```

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?