Data Structures

Array Lists 2

CS 225

September 6, 2023

Brad Solomon & G Carl Evans

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exam 1 Practice Exam Available

Practice exams give a rough idea of the format and style of questions

They are not exhaustive nor meaningfully repeatable

↳ September 11th

Lab and MP Feedback

Student feedback makes this class better

Weekly opportunities to provide anonymous feedback on Prairielearn

Entirely optional and very short!

Learning Objectives

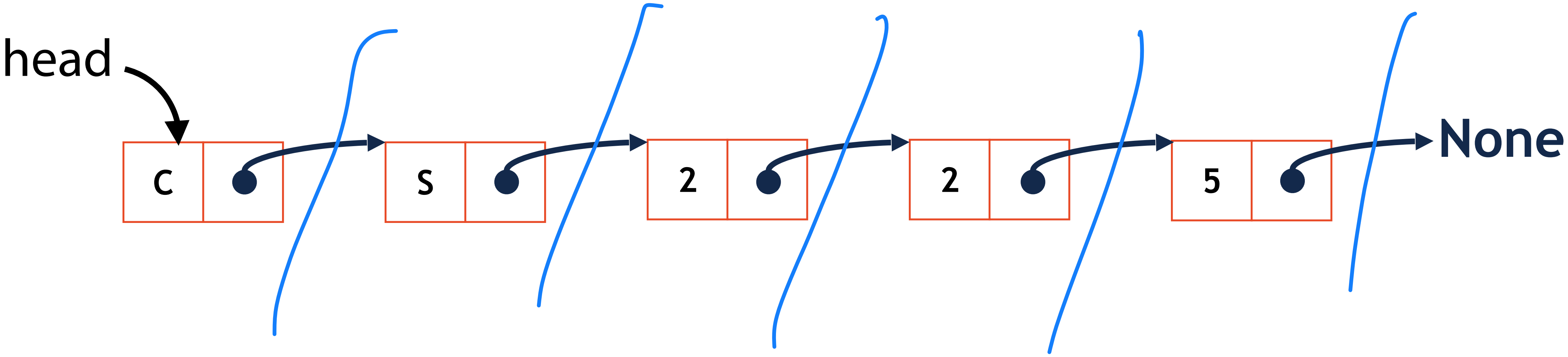
Review array list implementation

Discuss array resizing

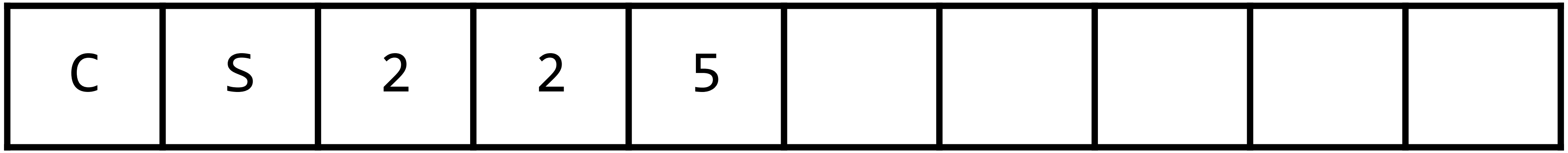
Consider extensions to lists

List Implementations → ADT

1. Linked List



2. Array List



List.h

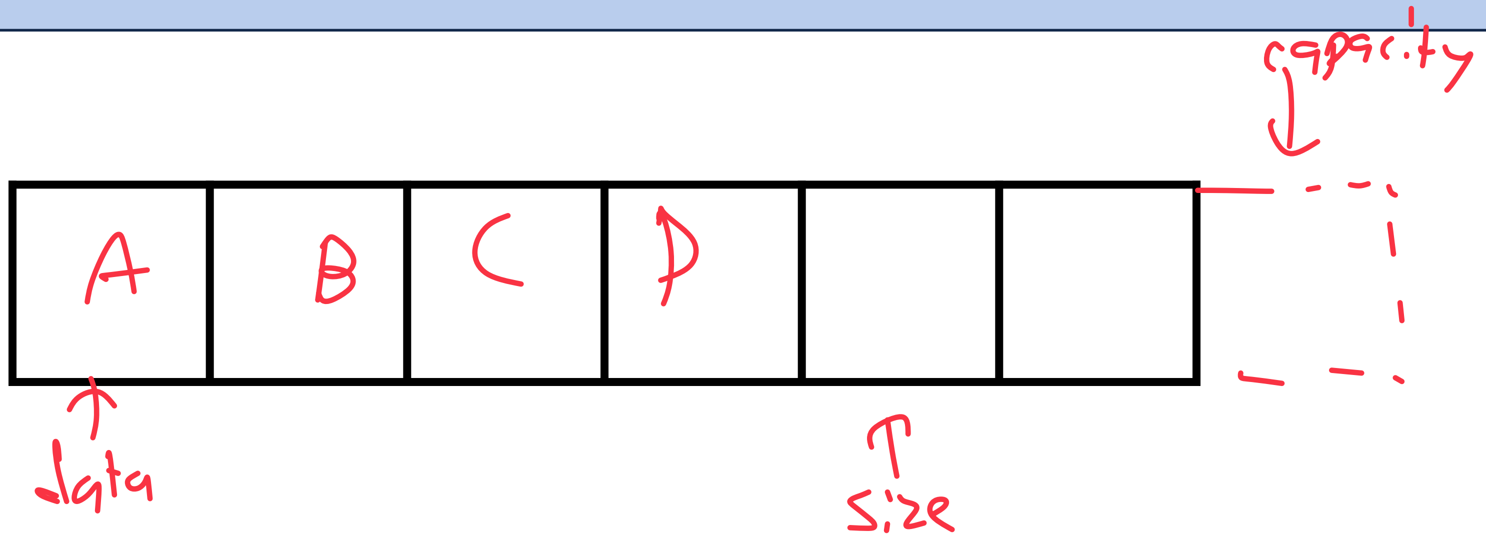
```
#pragma once

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

  T *size;

  T *capacity;

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



Array List

Index

start + size · index

O(1)

C	S	2	2	5					

Insert

O(n)

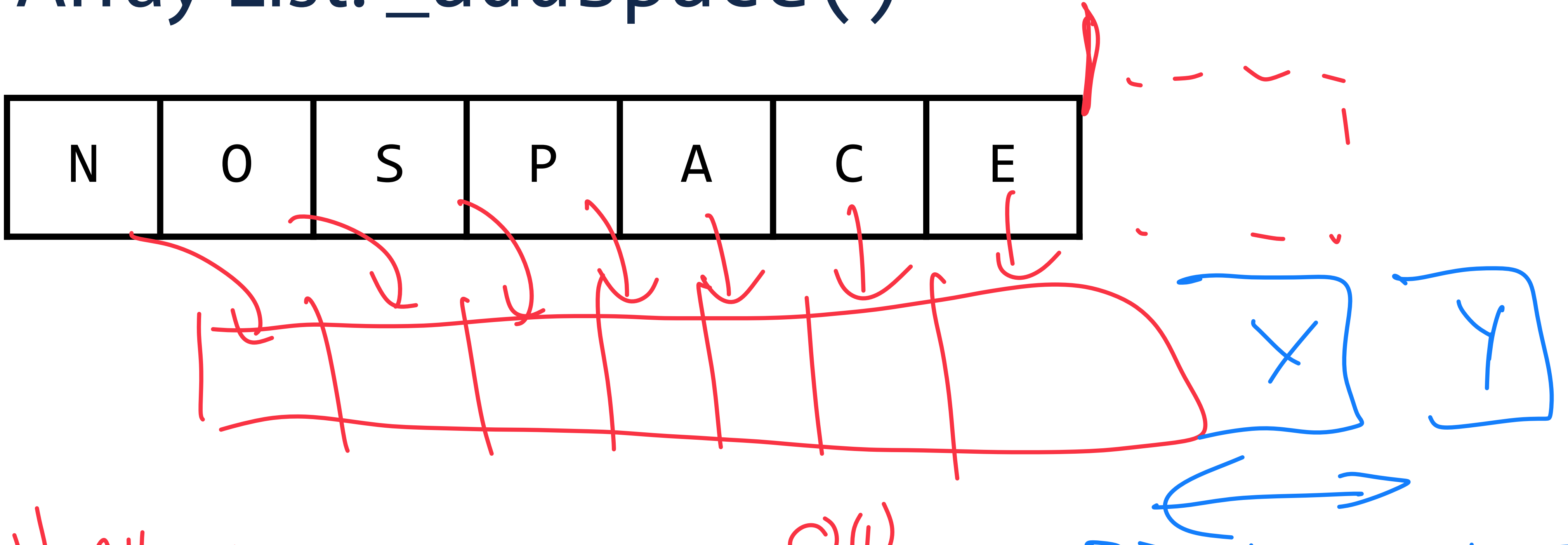
C	S	2	2	5					

Remove

O(n)

C	S	2	2	5					

Array List: _addspace()



1) Allocate a new array O(1)

2) Copy over everything O(n)

?? How much space to allocate

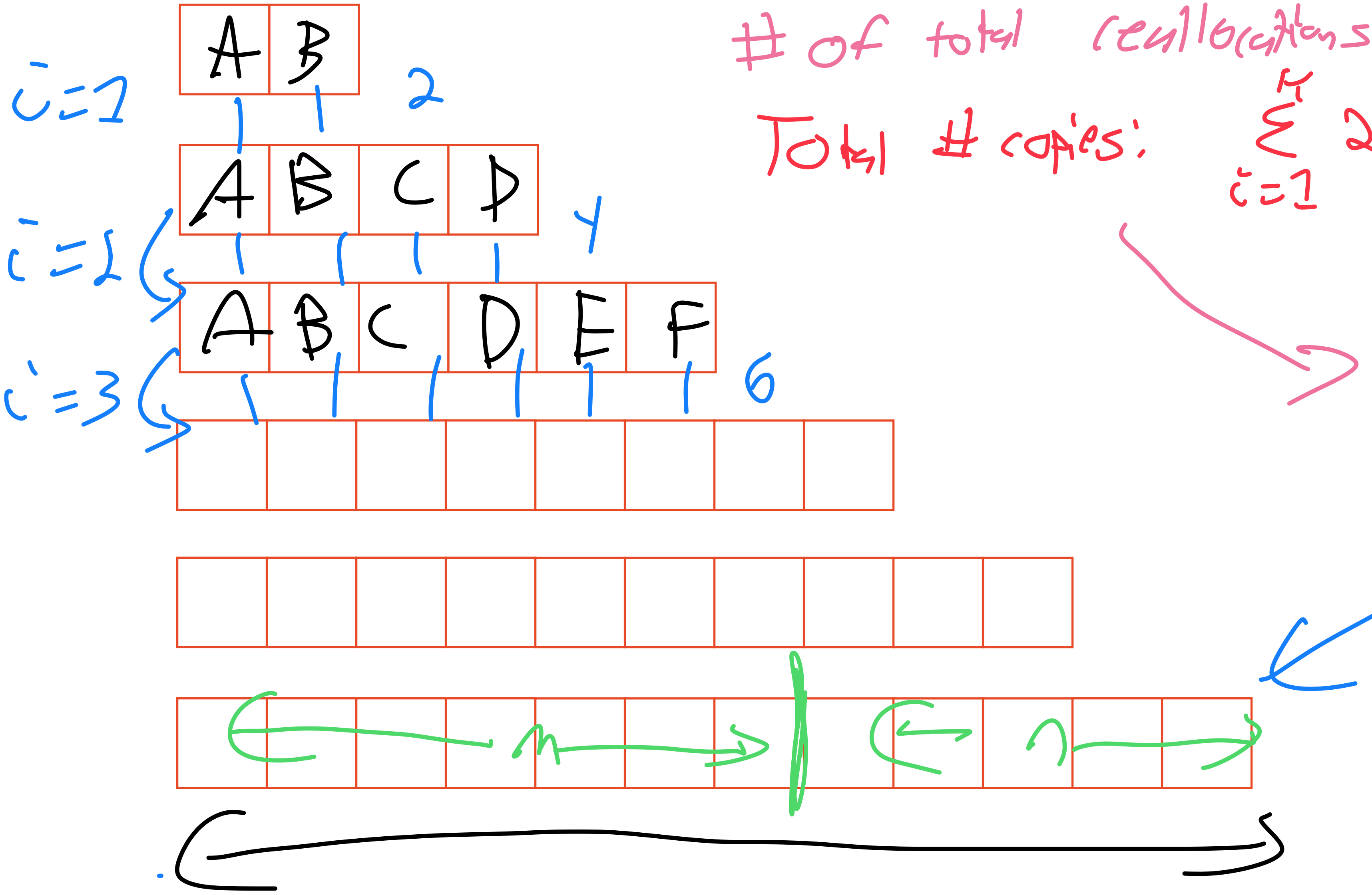
Resize Strategy: +2 elements every time

n total # of items

# of copies at reallocation i: 2i

# of total reallocations: n/2

Total # copies: $\sum_{i=1}^{k} 2i = k(k+1) = \frac{n}{2}\left(\frac{n}{2}+1\right) = \frac{n^2+2n}{4}$ + m·n



Resize Strategy: +2 elements every time

Amortized or expected

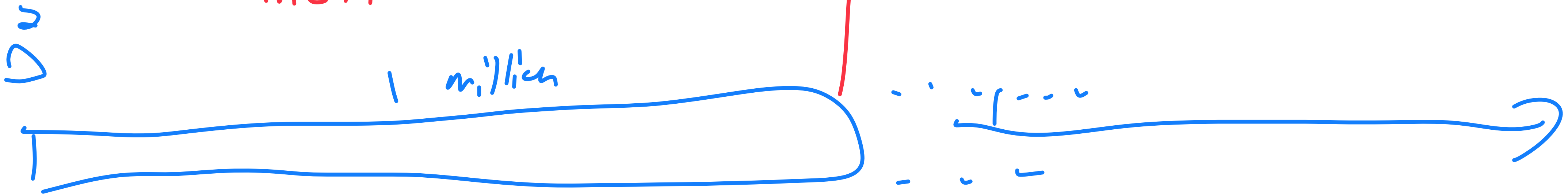
Total copies for n inserts: $\dfrac{n^2 + 2n}{4}$

1 insert: $\dfrac{n^2 + 2n}{4n}$

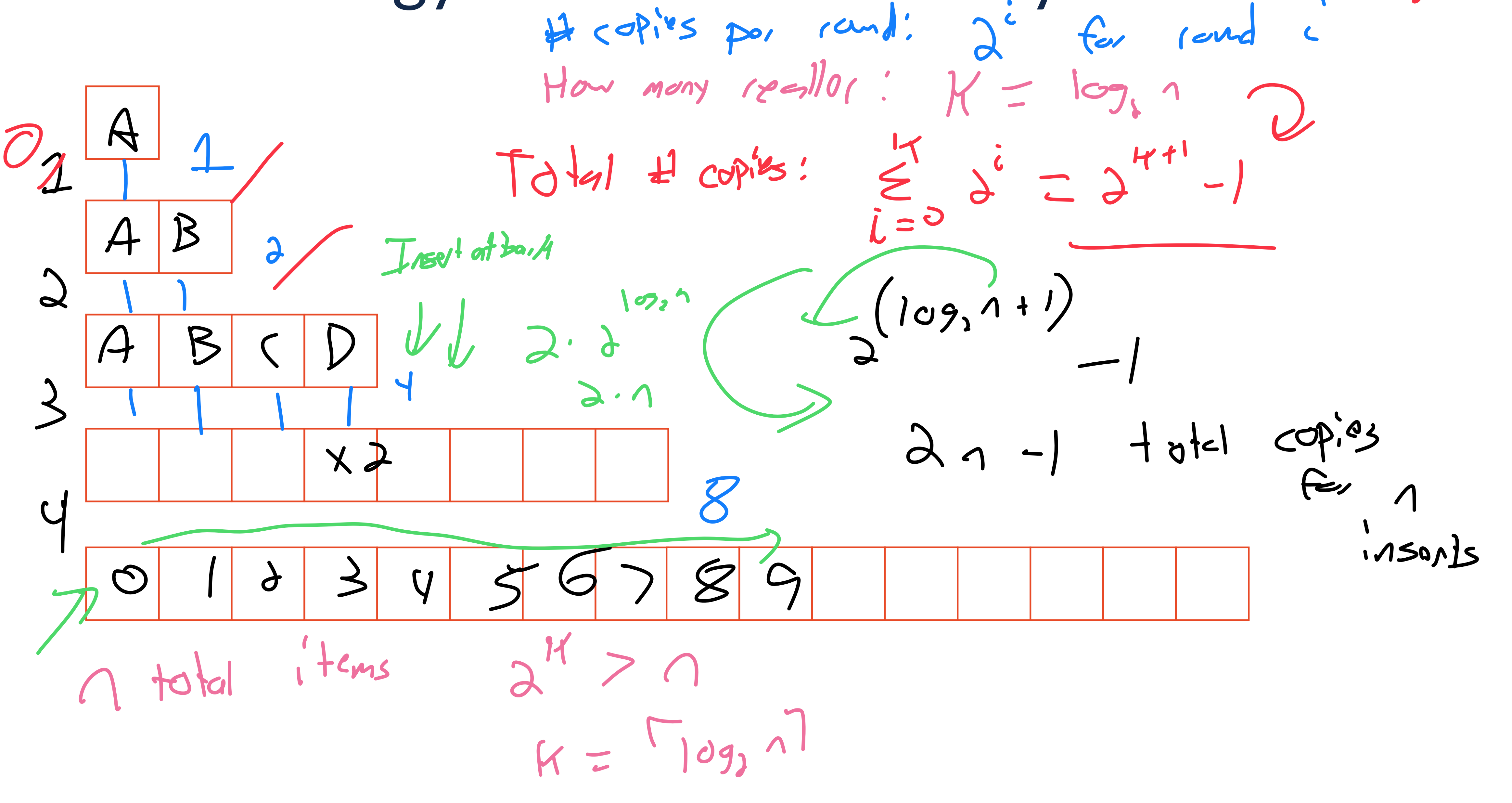
Expected copies for one insert: $\dfrac{n}{4} + \dfrac{1}{2}$

Big O of insert: $O(n)$

↳ worst case for 1 insert



Resize Strategy: x2 elements every time



Resize Strategy: x2 elements every time

Amortized

Total copies for n inserts: $2n - 1$

divide by n

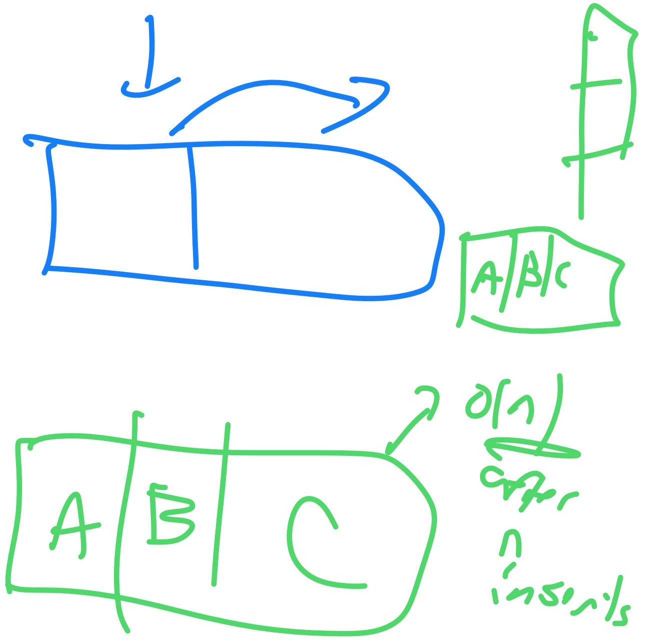
1 insert: $2 - \frac{1}{n}$

Big O: O(n)

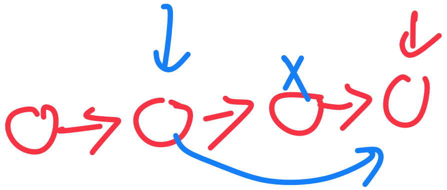
Expected insert cost is O(1)*

All logic here for insert at back

b/c don't consider shuffling



Array Implementation



	Singly Linked List	Array
Look up **arbitrary** location ↳ index (random access)	O(n)	O(1)
Insert after **given** element ↳ pointer to object (*&)	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

LL.insert(3)

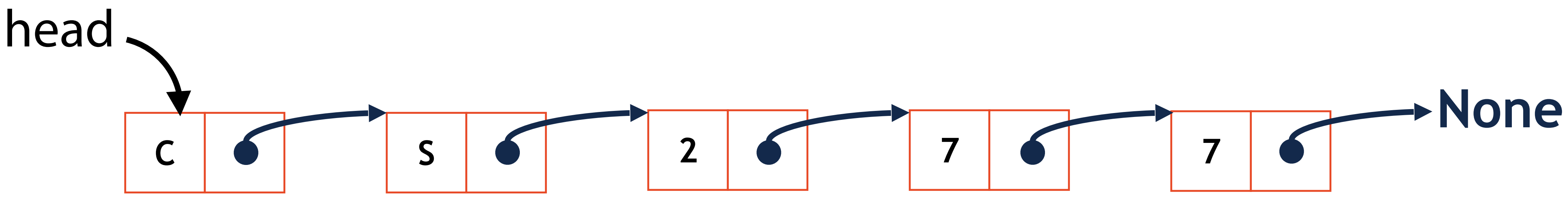
Thinking critically about lists: tradeoffs

The implementations shown are foundational.

Can we make our lists better at some things? What is the cost?

Thinking critically about lists: tradeoffs

Getting the size of a linked list has a Big O of: O(n)



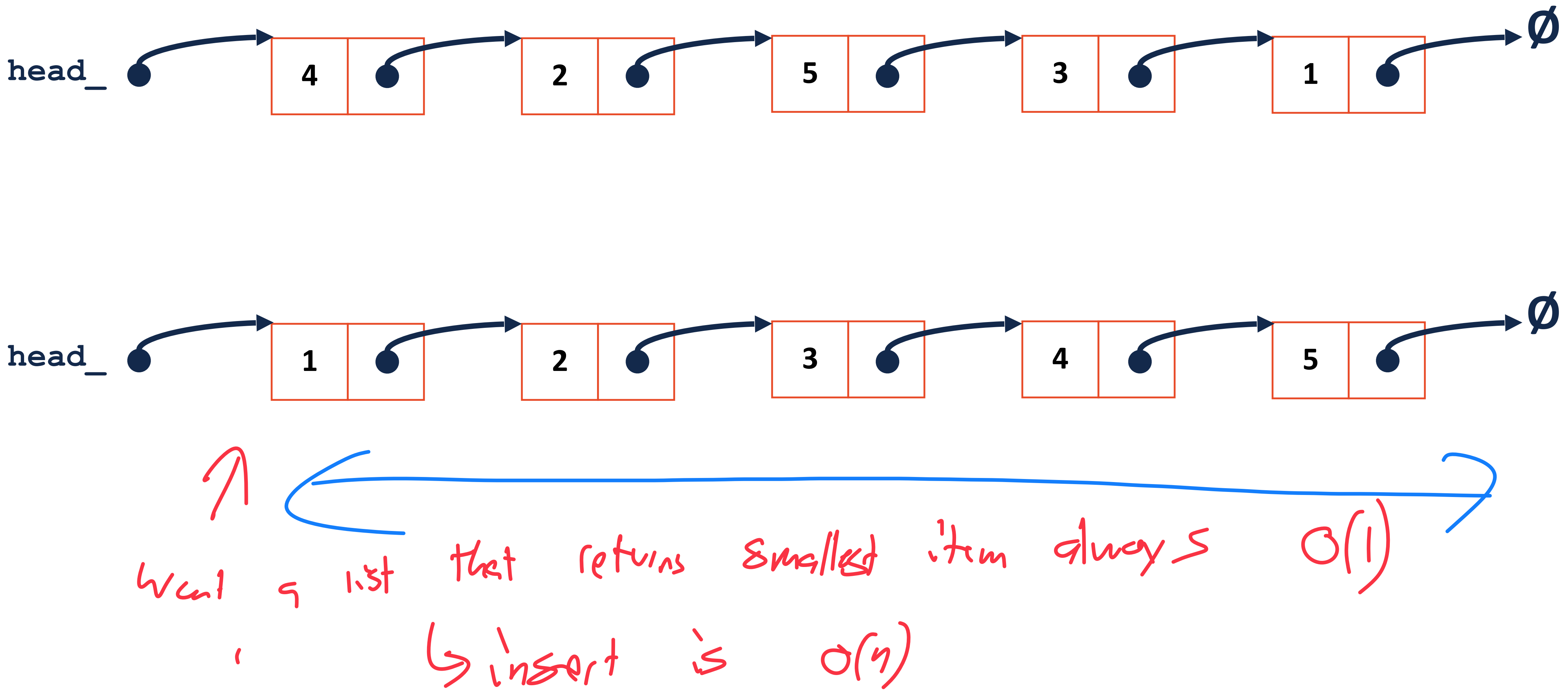
List:

private
ListNode * head;
unsigned size;

memory increased by 4 bytes

increase by O(1)

Thinking critically about lists: tradeoffs



Thinking critically about lists: tradeoffs

find(7)

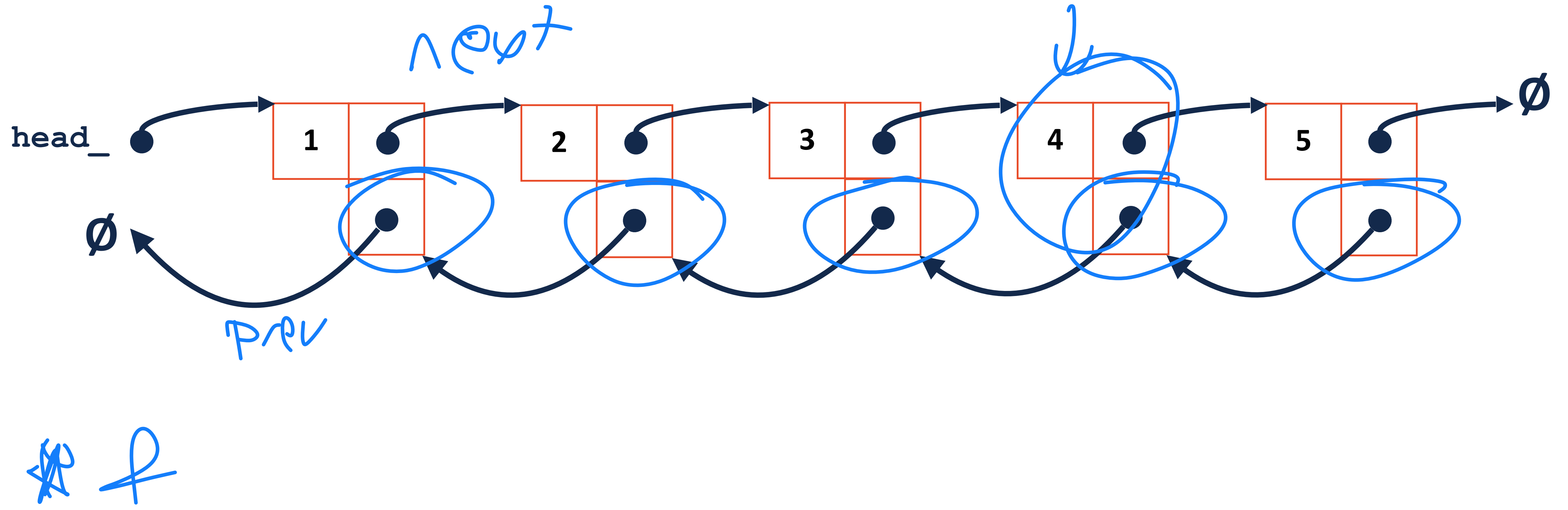
2	7	5	9	7	14	1	0	8	3

0	1	2	3	5	7	7	8	9	14

5 < 7

7 = 7

Thinking critically about lists: tradeoffs



Thinking critically about lists: tradeoffs

When we discuss data structures, consider how they can be modified or improved!

Can we make a 'list' that is O(1) to insert and remove?
What is our tradeoff in doing so?

↳ I can only insert & remove from specific places

LL: my head is O(1) insert / remov

Array*: first available space (T* size)

* is not full

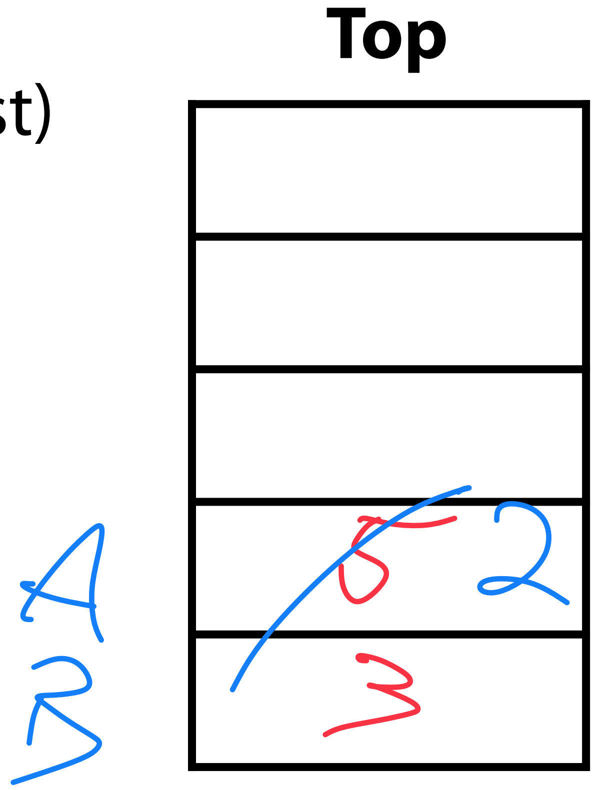
Stack Data Structure

A **stack** stores an ordered collection of objects (like a list)

However you can only do two operations:

Push: Put an item on top of the stack

Pop: Remove the top item of the stack (and return it)




```
push(3); push(5); pop(); push(2)
```

Stack Data Structure

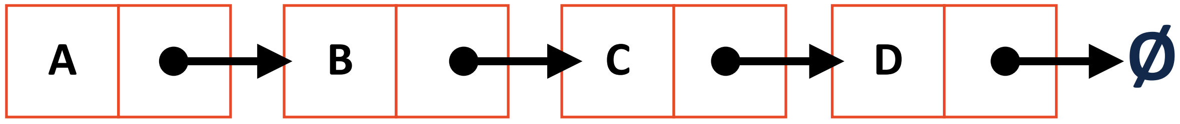
The **call stack** is a key concept for understanding recursion

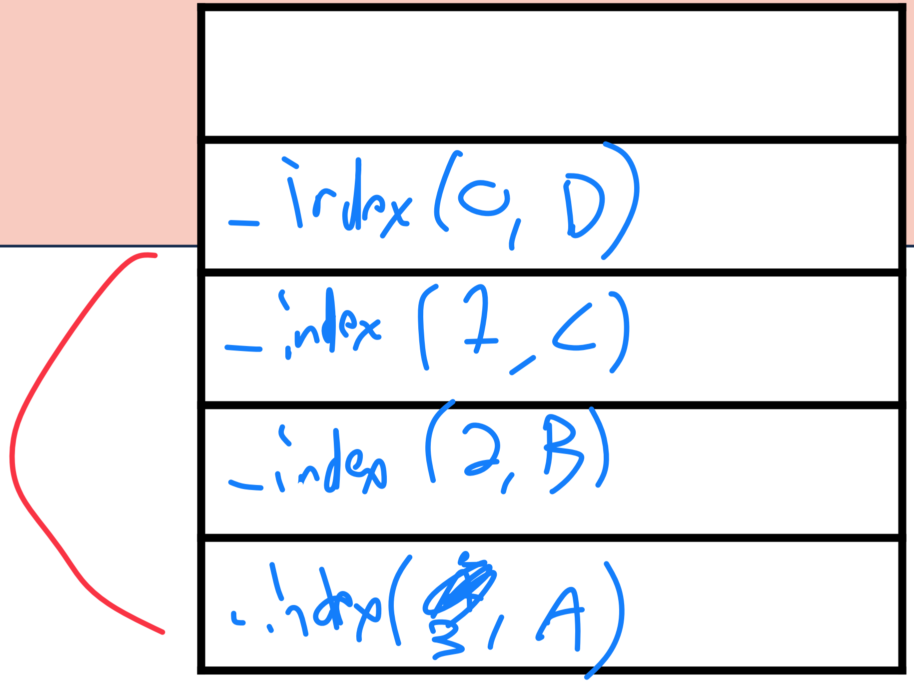
```
template <typename T>
typename List<T>::ListNode *& List<T>::_index(unsigned index, ListNode *& root){

    if (index == 0){ return root; }
    if (root == nullptr){ return root; }

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

}
```



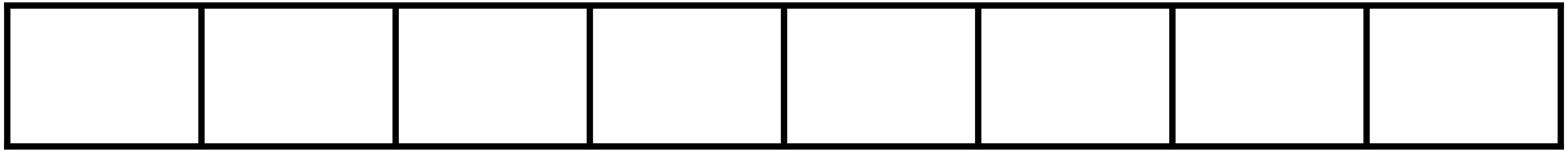


Stack Data Structure

C++ has a built-in stack

Underlying implementation is vector or deque

```
#include <stack>
int main() {
  stack<int> stack;
  stack.push(3);
  stack.push(8);
  stack.push(4);
  stack.pop();
  stack.push(7);
  stack.pop();
  stack.pop();
  stack.push(2);
  stack.push(1);
  stack.push(3);
  stack.push(5);
  stack.pop();
  stack.push(9);
}
```



Stack ADT

- [Order]:
- [Implementation]:
- [Runtime]:

Queue Data Structure

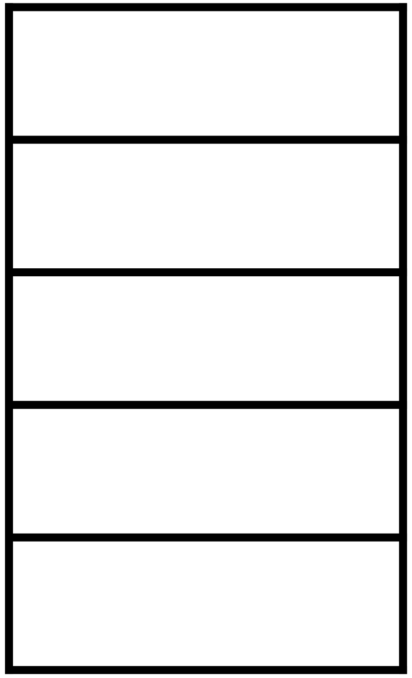
A **queue** stores an ordered collection of objects (like a list)

However you can only do two operations:

Enqueue: Put an item at the back of the queue

Dequeue: Remove and return the front item of the queue

Front



```
enqueue(3); enqueue(5); dequeue(); enqueue(2)
```