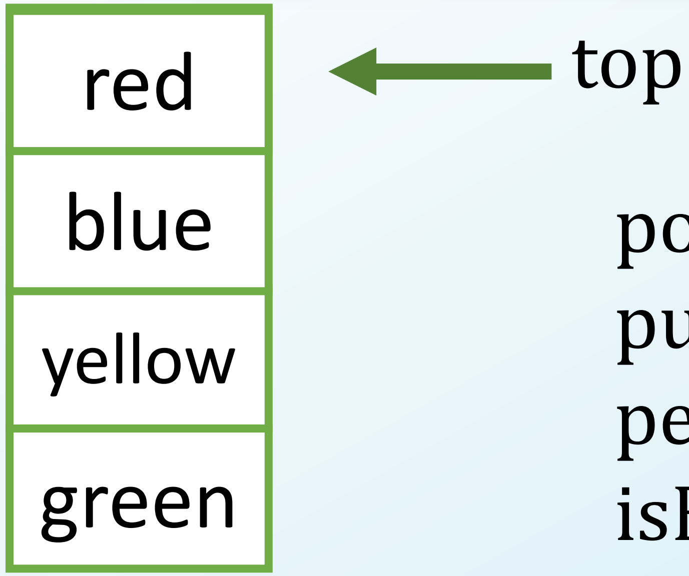
CTEC2901 DATA STRUCTURES AND ALGORITHMS

Lecture 5 Implementation of Stacks and Queues

Implementing a Stack

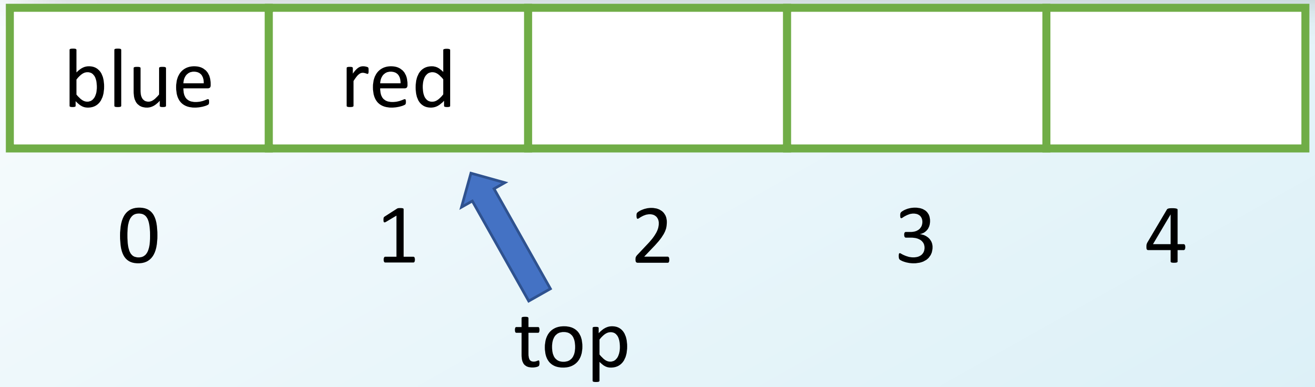


Recall what the stack operations are:

pop() - returns the top element & removes it
push(item) – pushes the item to the top
peek() - returns the top element without removing
isEmpty() - returns a boolean

A stack can be implemented in various ways, e.g. as an array or as a linked list.

Implementing a Stack – Array-based



top = 1 at this moment

A stack can be implemented by storing the elements in an array.

A variable top keeps track of the last element added.

push – The element is added to the next space (top+1).

pop – The element is removed from the last used space (top).

Implementing a Stack – Array-based

The Stack class would have a private variable representing the array.

```
public class Stack{
    private String[ ] arr;
    private int top;

    public Stack(){
        top = -1;
        arr = new String[10];
    }
}
```

The constructor initialises the array and the top variable.

Why does it start with -1?

There are no elements in it initially.

Implementing a Stack – Array-based

```
public void push(String str){
    arr[top] = str;
    top = top + 1;
}
```

Problems: What about when the array is full? It has a fixed size.

What if there's nothing in the array and pop is called?

```
public String pop(){
    String item = arr[top];
    arr[top] = null;
    top = top – 1;
    return item;
}
```

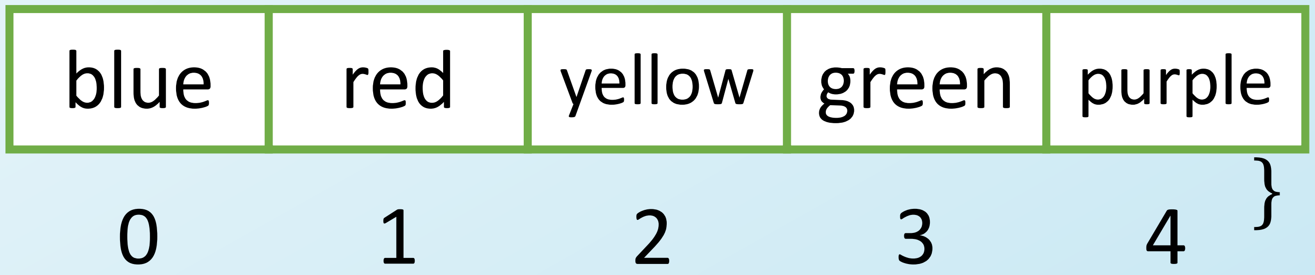
The location is set to null so that it can be garbage collected.

Implementing a Stack – Array-based

Solution: Keep a count of the elements.

```
public void push(String str){
    if (count < MAX_SIZE){
        arr[top] = str;
        top = top + 1;
    }else{
        // What should happen?
    }
}
```

```
public String pop(){
    if (count > 0){
        String item = arr[top];
        arr[top] = null;
        top = top – 1;
        return item;
    }else{
        // throw exception
    }
}
```



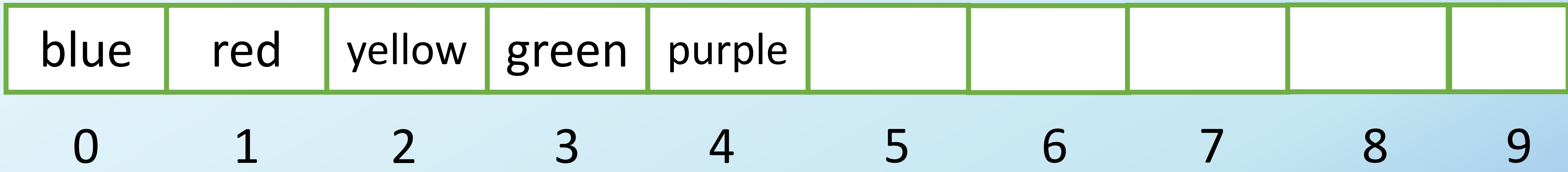
Implementing a Stack – Array-based

What to do when the array is full:

Increase the array. How can a fixed size array be increased?

Create a new array and copy all the elements over to the new array.

```
String[] arr2 = new String[MAX_SIZE*2];
for (int i = 0; i < arr.length; i++){
    arr2[i] = arr[i];
}
```



Implementing a Stack – Array-based

Why is doubling better than increasing by one?

Each increase means an O(n) copying operation.

Suppose that you start an array of 5 elements.

When the 6th element is added, the size will be increased to 10.

When the 7th element is added, it will fit without resizing. etc. ....

When the 11th element is added, the size will increase to 20.

It will be: 5, 10, 20, 40, 80, 160, 320, ... Works out to be O(n).

If it was increased by 1 each time, then it would be:

5, 6, 7, 8, 9, 10, 11, 12, ... There would be so many more increases.

This would be $O(n^2)$.

Implementing a Stack – Array-based

When popping, the array is halved only when the number of elements reaches a **quarter** of the size.

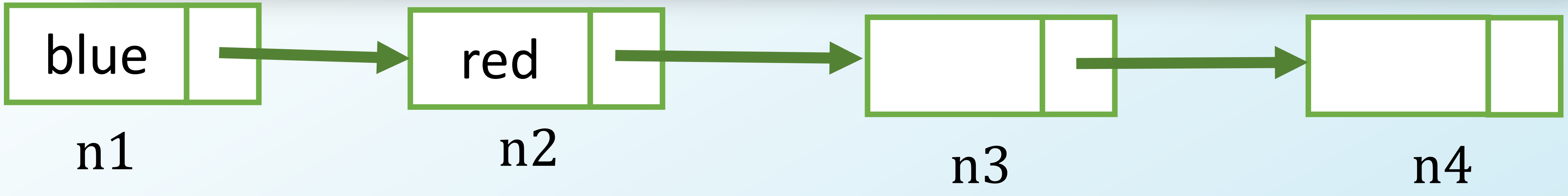
This is to prevent **thrashing**.

Thrashing is when there is a sequence of push, pop, push, pop, causing the array to be doubled, then halved, then doubled, then halved, etc.

This would be very expensive.

By halving only when it is ¼ full, the array is always between 25% and 100% full.

Implementing a Stack – Linked-list-based

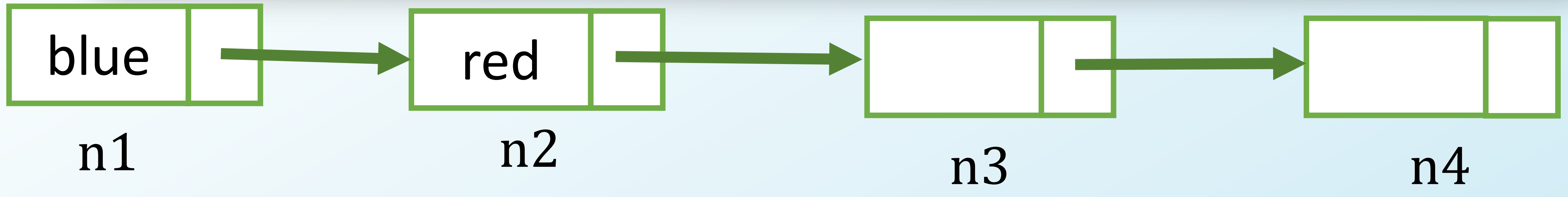


For a singly-linked list, pushing and popping is all done at the front.
Why? We have a head pointer so adding/removing at the front is O(1).

```
public class Stack{
    private LinkedList list;

    public Stack{
        list = new LinkedList();
    }
    ... } // end of class
```

Implementing a Stack – Linked-list-based

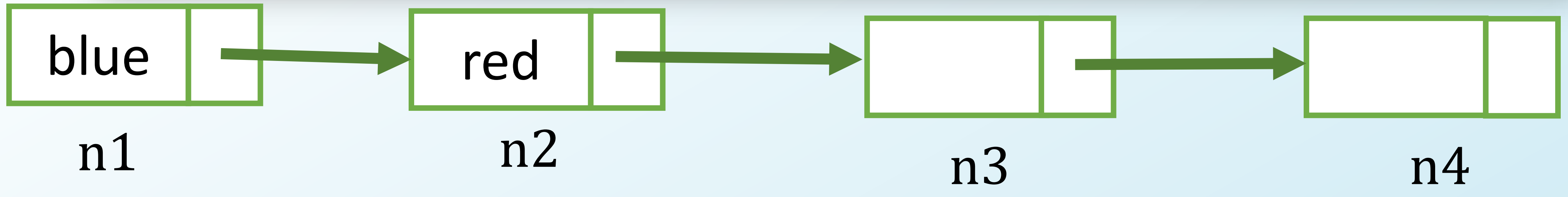


```
public void push(String str){
    Node head = list.getHead();
    Node n = new Node(str, head);
    list.setHead(n);
}
```

Creates a new Node to store the new element and attaches it in front of the head.

Sets the head to point to the new node.

Implementing a Stack – Linked-list-based



```
public String pop(){
    Node n = list.getHead();
    list.setHead(n.next());
    return n.getItem();
}
```

Returns the element at the start of the list.

Sets the head to point to the next node in the list.

Implementing a Stack

Comparison:

pop and push are both constant O(1) when using a linked list.

Linked lists allow the stack to increase/decrease.

When using arrays, push and pop have worst-case complexity of O(n), but when averaged over the number of operations, it comes to an ***amortised*** complexity of O(1).

Stack Example 1 – Matching parentheses

Stacks can be used to check that all brackets match in a string, e.g. in some code:

```
if ((x < 4) && (y > 2 || (z > 3)))
```

Parse the string, one character at a time.

Every time you get an opening bracket, push it on the stack.

Every time you get a closing bracket, pop from the stack.

- nothing on the stack means more closing brackets than opening.

Something still on the stack at the end means more opening brackets.

Stack Example 2 – Postfix Notation

Normally we use infix notation, e.g. 4 + 5 * 6

Postfix notation: 4 5 6 * +

Infix: (2 + 4) * 9 + 6 * 8

Postfix: 2 4 + 9 * 6 8 * +

Stack Example 2 – Postfix Notation

Postfix: 2 4 + 9 * 6 8 * +

Stack-based solution:

Parse the string from left to right.

If it is an operand, e.g. 2, push it onto the stack.

If it is an operator, e.g. +, pop 2 elements from the stack, perform the operation and push the answer back.

Stack Example 2 – Postfix Notation

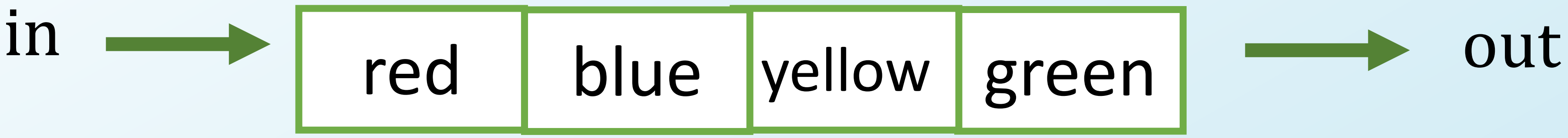
Postfix: 2 4 + 9 * 6 8 * +

1. push 2
2. push 4
3. pop 2 and 4, add them and push 6 on the stack
4. push 9
5. pop 9 and 6, multiply them and push 54
6. push 6
7. push 8
8. pop 6 and 8, multiply them and push 48
9. pop 48 and 54, add them and push 102

Answer = 102

Queues

Stacks are LIFO (last-in first-out), queues are FIFO (first-in first-out).



Elements go in one end and come out the other end.

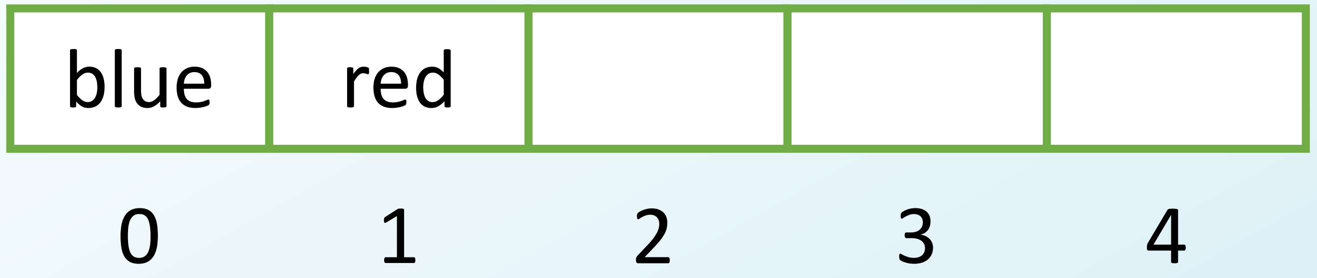
enqueue(item) – puts the element into the queue

dequeue() – removes the element from the other end

isEmpty() – checks if it is empty

peek() – returns the element from the other end but without removing

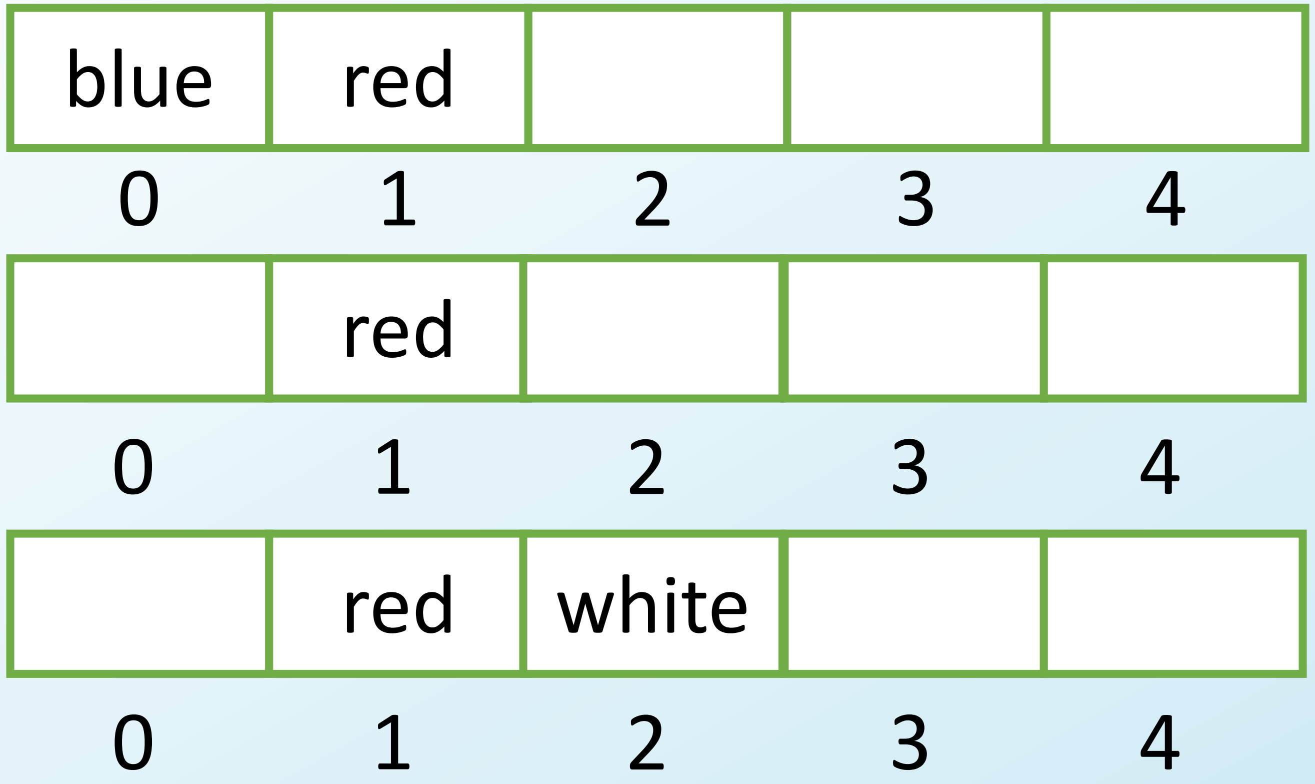
Implementing a Queue – Array-based



For stacks, we could just add and remove from one end, so it could be done in O(1) (apart from when it has to be resized).

Can the same thing work for queues?

Implementing a Queue – Array-based



front = 0,
rear = 1

Dequeue blue

front = 1,
rear = 1

Enqueue white

front = 1,
rear = 2

Everything keeps shifting. Moving it back would cost O(n).

Implementing a Queue – Array-based

Solution: Use a circular array.

After it reaches the end, the next element goes in the start if there's space.

blue	red	pink	green	white
0	1	2	3	4

Enqueue white

front = 0,
rear = 4

	red	pink	green	white
0	1	2	3	4

Dequeue blue

front = 1,
rear = 4

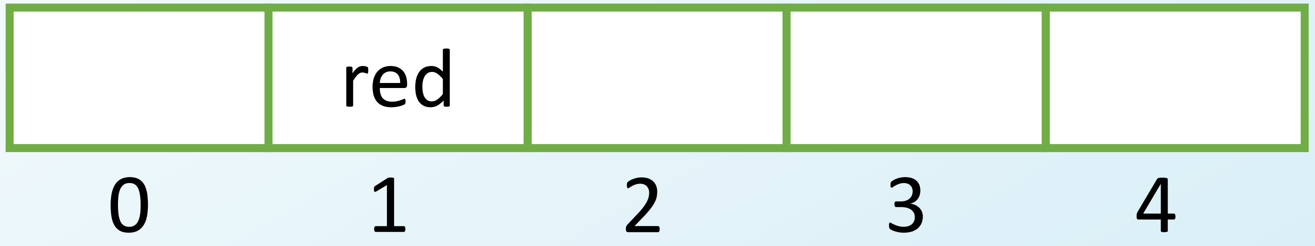
lilac	red	pink	green	white
0	1	2	3	4

Enqueue lilac

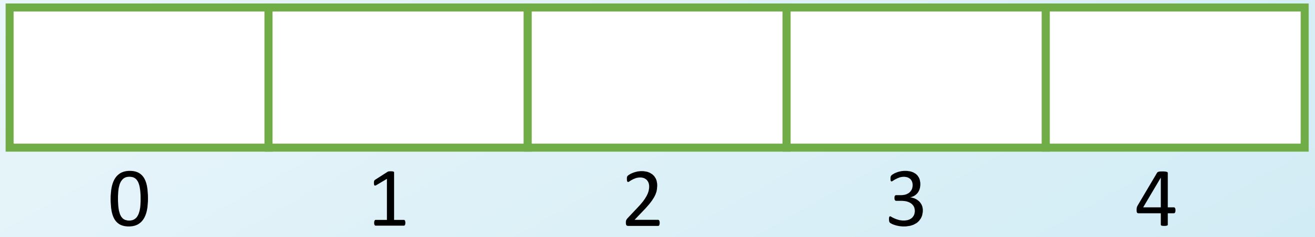
front = 1,
rear = 0

Implementing a Queue – Array-based

How do you know if the queue is empty? Front is one ahead of rear.



front = 1,
rear = 1



Dequeue red

front = 2,
rear = 1

Implementing a Queue – Array-based

Wait, but what about when it is full? Front is one ahead of rear.

lilac	red		green	white
0	1	2	3	4

front = 3,
rear = 1

lilac	red	pink	green	white
0	1	2	3	4

Enqueue pink

front = 3,
rear = 2

It's the same for both empty queues and full queues!

Solution 1: Keep a count

Solution 2: Maintain a gap in the queue. This is more efficient.

Implementing a Queue – Array-based

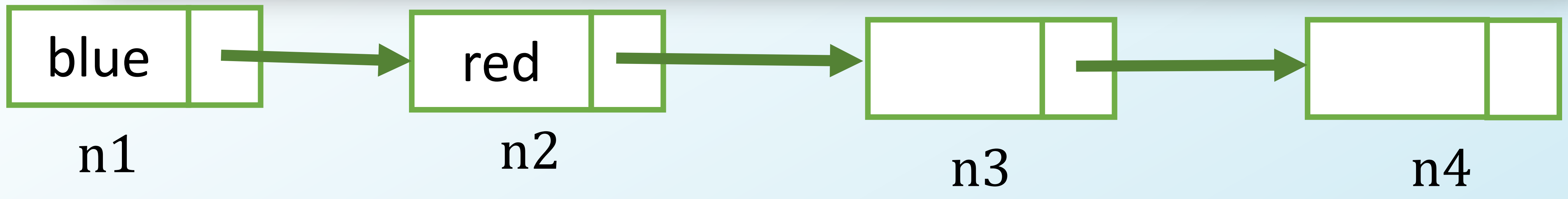
How do you get it to be a circular array though?

Use mod, e.g. 0 % 5 = 0, 1 % 5 = 1, …   5 % 5 = 0, 6 % 5 = 1, …

Enqueue: Instead of rear = rear + 1,
rear = (rear + 1) % array.length()

Dequeue: Instead of front = front + 1,
front = (front + 1) % array.length()

Implementing a Queue – Linked-list-based



For a stacks, it was simple – just add and delete from the head end.

Problem: For queues, adding and deleting are on different ends.

Recall that adding to the end of a singly-linked list takes O(n)

… except if there’s a tail pointer! Then it’s O(1).

Deleting from the tail end is O(n), so keep both head and tail pointers.

Deques

Deques (double-ended queues) are a combination of stacks and queues.

Add and remove from both ends.

Elements still can’t be removed from the middle.

Operations:
addFront()
addRear()
removeFront()
removeRear()
etc.

Implementations of deques can also be used to implement queues or stacks, as these are special cases of deques.