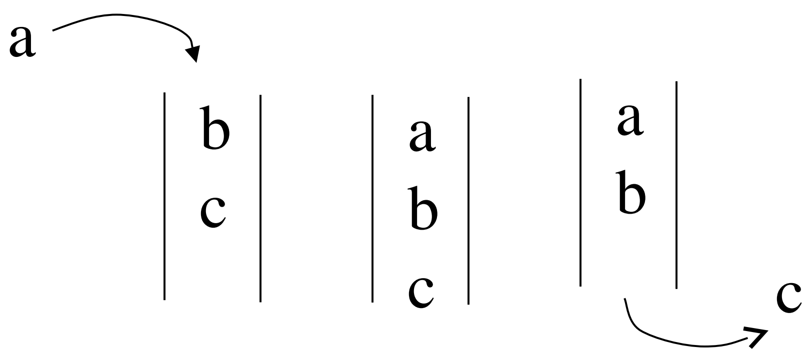
Computer Science 112
Data Structures

Lecture 09:
Binary Search

Review: Queues

- **First in first out: Queue**



Operations

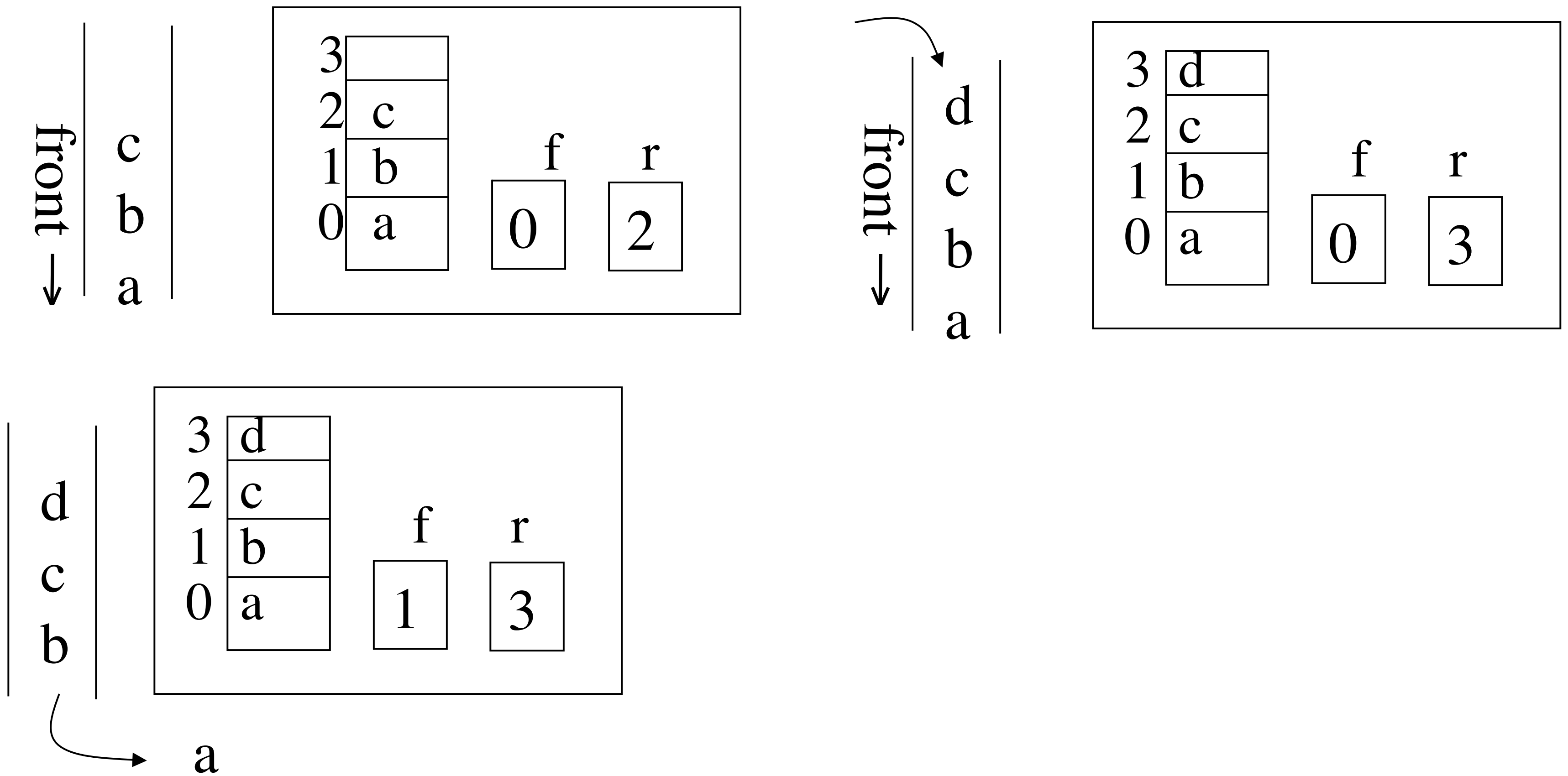
- Queue<T>
  - public void enqueue(T data)
  - public T dequeue( )
  - public boolean isEmpty( )
  - public int size( )
  - public void clear( )

Implementing Queues

- Queues need to be accessed at both ends, so implementations are a bit messier
  - Arrays: need two ints to keep track of both front and back
  - linked lists: use circular lists or have two pointers

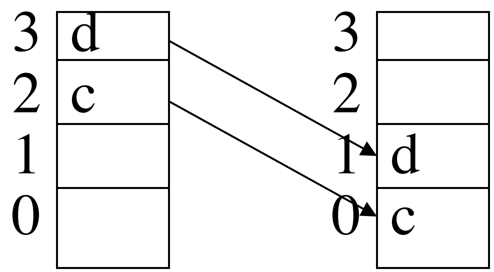
Queues as arrays

- **Keep track of both front & rear**



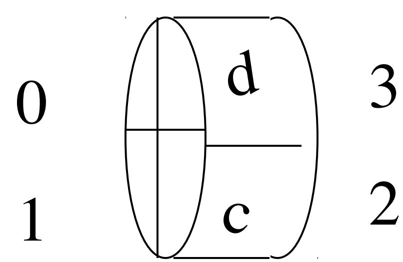
Queues as arrays

- **Problem: how to reuse space emptied by dequeue?**
  - **Could move data down: O(n)**



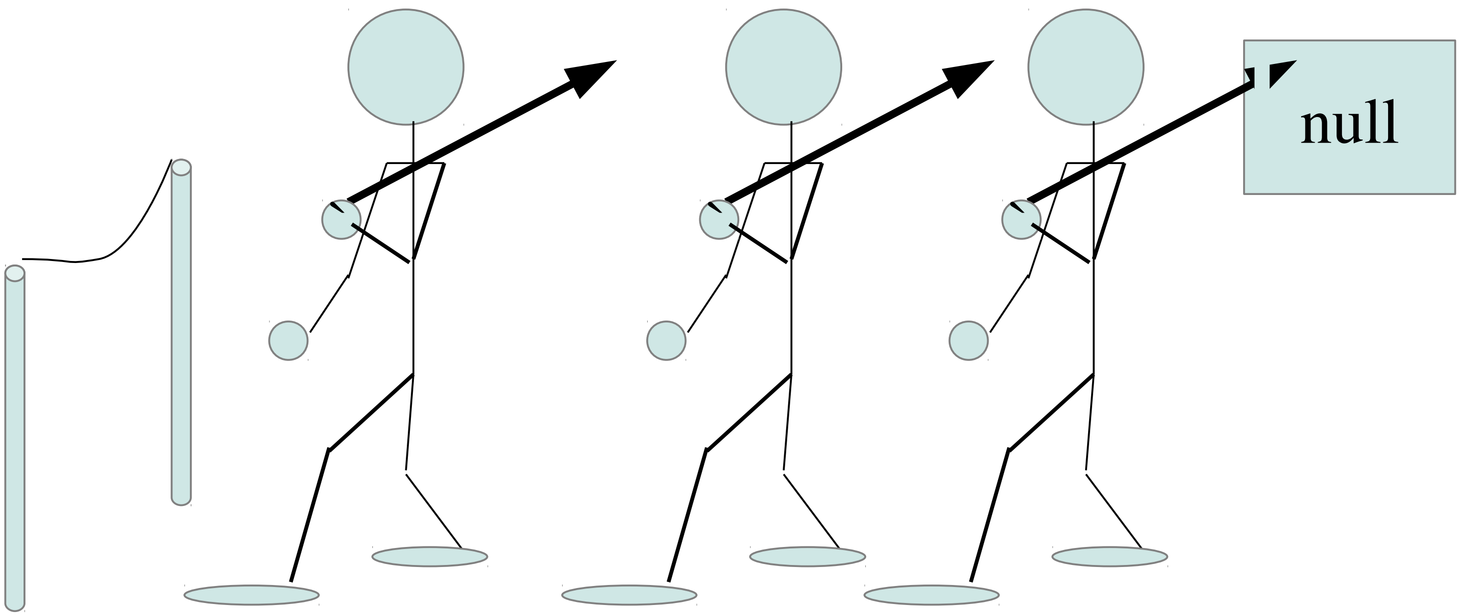
  - **Treat array as circular**

    front = (front + 1) % size



Queues as linked lists

- **Problem: Need to access both ends**
- **Solution: Linked list with head/tail pointer or circular linked list**
- **Which end of the list should be the front of the queue?**
  - **enqueue is O(1) time whether at head or tail**
  - **dequeue is O(1) at head but O(n) at tail (Why?)**
  - **so more efficient when front is head**


null

Sequential Search

- **Is target value in this linked list / array?**

=> Is target the first element? No

Is target the second element? No

…

Is target the 35th element? Yes!

Cost

- **Operation to count:**
  - **test if target equals element**
- **Best case? target in first element: 1**
- **Worst case? target in last element or not there at all: n**
- **Average case …**

Input Cases

- **We group the inputs into *cases*.**
- **A case is either**
  - **A specific input**

    target is 5, array is {4, 6}
  - **Or a group of inputs, all with the same cost**

    inputs where target is not in the array

Average Cost

- when we say "average cost" we mean a probability-weighted average.
- If all input cases are equally likely, average cost is

$$\frac{\sum_i C(i)}{N}$$

- C(i) is the cost of input case i
- N is number of different input cases

Average Cost
Equal Probabilities

- **Suppose 3 possible input cases, with equal probabilities:**

Input Case i	Cost C(i)
Target in element 0	1
Target in element 1	2
Target in element 2	3

Average Cost = Total Cost / N
= (1+2+3) / 3 = 2

Average Cost
Different Probabilities

- **If different input cases have different probabilities, average cost is**

$$\sum_i \left(C(i) * P(i) \right)$$

- **C(i) is the cost of input case i**
- **P(i) is the probability of input case i**

Average Cost
Different Probabilities

- **Suppose 3 possible input cases:**

i	Input Case i	Cost C(i)	Probability P(i)	C(i)*P(i)
1	target not in array	2	1/2	1.00
2	target in element 0	1	1/4	.25
3	target in element 1	2	1/4	.50
	Average Cost			1.75

Average Cost of Sequential Search

- **Target is in the array, equal probability at each position, length n**

  Average cost = (1 + 2 + … + n) / n

  = (n*(n+1)/2) / n

  = (n+1) / 2

Failed search

- **Array not in order**
- **Assume target is** ***not*** **found (failure)**
- **Worst case = best case = average case = n comparisons**

Ordered Array

- Suppose we do linear search in an <u>ordered</u> array:

```
for (int j = 0; j < n; j++){
    if  ( ! (a[j] < target) ) {
        if (a[j] == target){
          return true;
        } else {    // a[j] > target
          return false;
} } }
return false;
```

Ordered Array

- **Suppose search fails. What is average cost?**

Array Contents	**gap i**	**C(i)**
	1	1
15		
	2	2
23		
	3	3
37		
	4	4
48		
	5	4

$$\Sigma\, C(i) / N$$
$$= (1+2+..+N + N)/N$$
$$= (N+1)/2 + 1$$

Block Search

- **Sorted array, size n**
- **think of it as m blocks of s elements each**
- **compare target with last element of first block:**
  a[s-1]
  - **if target == a[s-1] target has been found**
  - **else if target < a[s-1] search from start of block to a[2-1]**
  - **otherwise, redo on next block**
- **What is worst case cost? What is average cost?**

Search an ordered array

- **Question game**
  - **You think of a number between 1 and 1,000**
  - **I will guess a number**
  - **You tell me if your number is**
    - the same as my guess,
    - bigger than my guess,
    - smaller than my guess

Search an ordered array

- In an ordered array, one test can rule out a whole region of the array

```
if (a[10] < target){
    // if we get here target can't be in a[j] for
    // 0 <= j <= 10
```

Binary Search

??	??	??	??	??	??	??
0	1	2	3	4	5	6

Search for 30

left	right	mid
0	6	

Binary Search

??	??	??	29	??	??	??
0	1	2	3	4	5	6

Search for 30

left	right	mid
0	6	3

Binary Search

??	??	??	29	??	??	??
0	1	2	3	4	5	6

```
Search for 30
 left   right   mid
  4       6
```

Binary Search

??	??	??	29	??	35	??
0	1	2	3	4	5	6

Search for 30

left	right	mid
4	6	5

Binary Search

??	??	??	29	??	35	??
0	1	2	3	4	5	6

Search for 30

left	right	mid
4	4	

Binary Search

22	25	26	29	30	35	??
0	1	2	3	4	5	6

Search for 30

left	right	mid
4	4	4

Binary Search

??	??	??	??	??	??	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
0	6	

Binary Search

??	??	??	29	??	??	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
0	6	3

Binary Search

??	??	??	29	??	??	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
4	6	

Binary Search

??	??	??	29	??	35	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
4	6	5

Binary Search

??	??	??	29	??	35	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
4	4	

Binary Search

??	??	??	29	30	35	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
4	4	4

Binary Search

??	??	??	29	30	35	??
0	1	2	3	4	5	6

Search for 32

left	right	mid
5	4	

See BinarySearch.java

See RecursiveBinSearch.java

How Many Questions

- How many questions for 1000 numbers?
  - Each question eliminates 1/2 of possibilities
  - Therefore each question doubles the size of array you can search
  - Therefore size = $2^{questions}$
  - Therefore $\log_2(size)$ = questions
  - $Log_2 (1000) \approx 10$
  - Answer: 10 questions

Searching an array Performance

- **Search among 1 <u>M</u>illion entries**
- **Check 1 million entries per second**
  - **Sequential search**
    - **1 million operations needed**
    - **Requires 1 second**
  - **Binary earch**
    - **20 operations needed**
    - **Requires 20 microseconds**
    - **50,000 times faster**

Searching an array Performance

- **Search among 1 Billion entries**
- **Check 1 million entries per second**
  - **Sequential search**
    - **1 billion operations needed**
    - **Requires 1000 seconds - about 20 minutes**
  - **Binary search**
    - **30 operations needed**
    - **Requires 30 microseconds**
    - **30 million times faster**