CSE 332 Spring 2026
Lecture 2: Algorithm Analysis pt.1

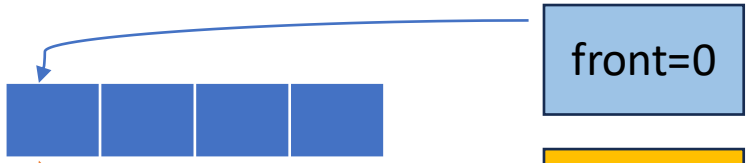
Michael Whitmeyer

http://www.cs.uw.edu/332

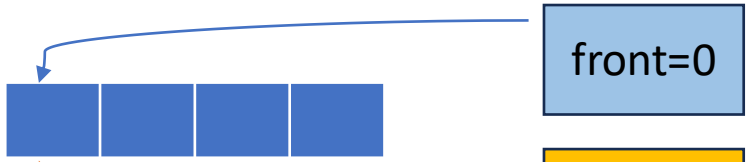
Announcements

- Exercise 0 released
  - Due Tuesday 6/30
  - There are 2 separate gradescope submissions
- Concept check 0 released
  - Helps us to get more familiar with each other!
  - Gradescope submission due Friday 6/26
  - Meet the staff activity due by 7/17
    - Come to any office hours, chat with us, ask us to mark you off for this

Circular Array Queue

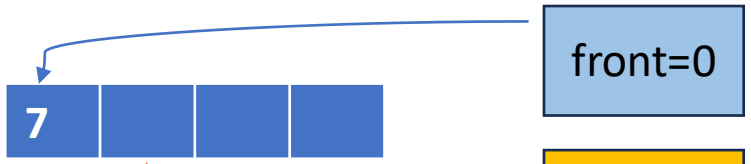


Circular Array Queue



Enqueue(7)

Circular Array Queue

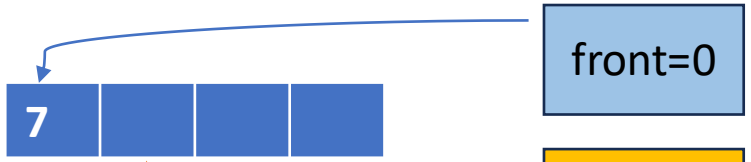


front=0

back=1

Enqueue(7)

Circular Array Queue



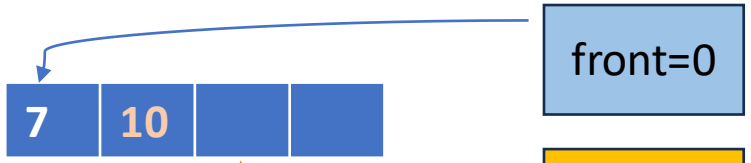
front=0

back=1

Enqueue(7)

Enqueue(10)

Circular Array Queue



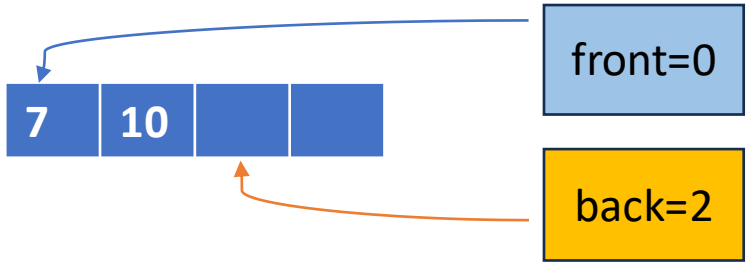
front=0

back=2

Enqueue(7)

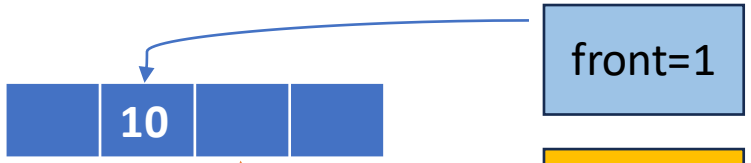
Enqueue(10)

Circular Array Queue



Enqueue(7)
Enqueue(10)
Dequeue()

Circular Array Queue

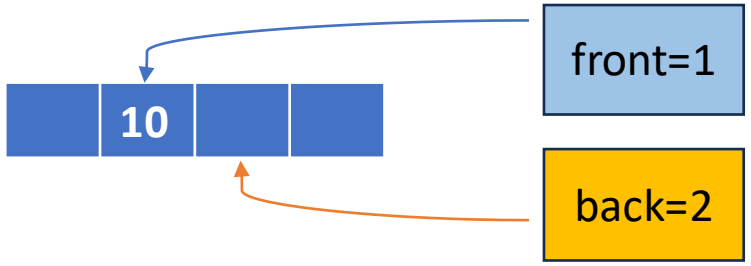


front=1

back=2

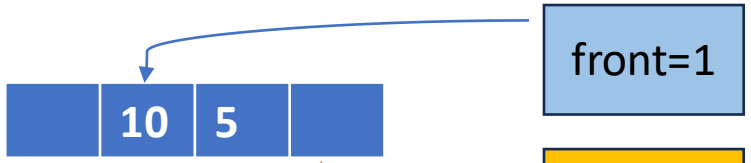
Enqueue(7)
Enqueue(10)
Dequeue()

Circular Array Queue



Enqueue(7)
Enqueue(10)
Dequeue()
Enqueue(5); Enqueue(8); Enqueue(2)

Circular Array Queue



front=1

back=3

Enqueue(7)
Enqueue(10)
Dequeue()
Enqueue(5); Enqueue(8); Enqueue(2)

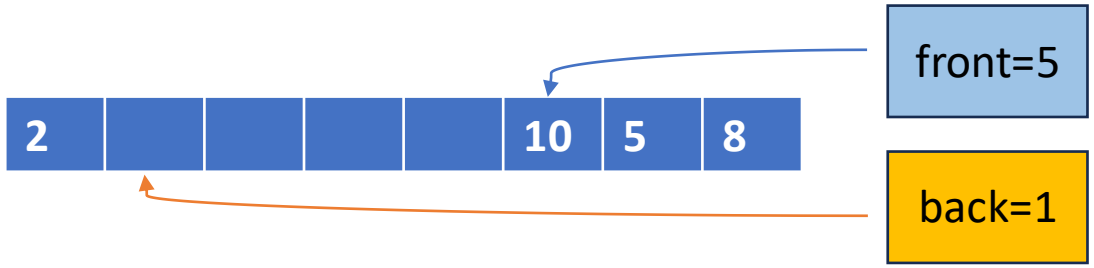
Circular Array Queue

front=1

back=0

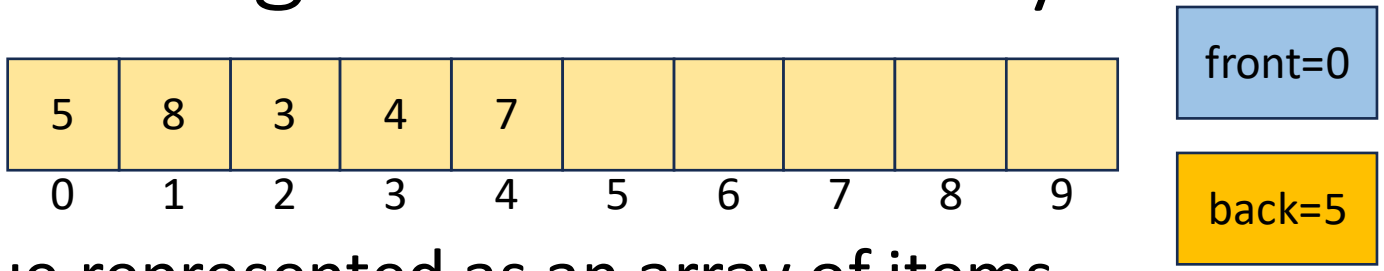
Enqueue(7)
Enqueue(10)
Dequeue()
Enqueue(5); Enqueue(8); Enqueue(2)

Circular Array Queue



Enqueue(7)
Enqueue(10)
Dequeue()
Enqueue(5); Enqueue(8); Enqueue(2)

Resizing “Circular” Array Queue



- Queue represented as an array of items
  - A “front” index to indicate the oldest item in the queue
  - A “back” index to indicate the most recent item in the queue
- enqueue Procedure:

```
enqueue(x):
    if (size == queue.length-1) then resize()
    queue[back] = x
    back = (back + 1) % queue.length
    size = size + 1
```

How do you resize?

That’s for Exercise 0!

- dequeue Procedure:

```
dequeue():
    // Assumes queue is not empty
    first = queue[front]
    front = (front + 1) % queue.length
    size = size - 1
    return first
```

- isEmpty Procedure:

```
isEmpty():
    return size == 0
```

Runtime/Resource Analysis

Why do resource analysis?

- Allows us to compare *algorithms,* not implementations
  - If my implementation on my computer is slower than your implementation on your computer, we **cannot** conclude your algorithm is better
  - Using observations couples the algorithm and implementation
- We can predict an algorithm's resource use before implementing
- Understand where the bottlenecks are in our algorithm

Process for resource Analysis

- End Result: A *function* which maps the algorithm's input size to count of resources used
  - Input of the function: **sizes** of the input
    - Number of characters in a string, number of items in a list, number of pixels in an image
  - Output of the function: **counts** of resources used
    - Number of times the algorithm adds two numbers together, number times the algorithm does a > or < comparison, maximum number of bytes of memory the algorithm uses at any time
- Important note: Make sure you know the "units" of your input and output!

Input Size: Examples

Example 1 (factoring): Suppose I want to factor a number N into two primes, p and q

Used in (email) encryption: assumes computers cannot factor large numbers quickly!

Input Size: Examples

Example 1 (factoring): Suppose I want to factor a number N into two primes, p and q

Used in (email) encryption: assumes computers cannot factor large numbers quickly!

Factor(N):

For i = 0 to i = $\sqrt{N}$:

check if (n % i == 0). If so, output p = i, q = n/i.

Input Size: Examples

Example 1 (factoring): Suppose I want to factor a number N into two primes, p and q

Used in (email) encryption: assumes computers cannot factor large numbers quickly!

Factor(N):

For i = 0 to i = $\sqrt{N}$:

check if (n % i == 0). If so, output p = i, q = n/i.

Runtime? (assume mod and division are constant time)

Input Size: Examples

Example 1 (factoring): Suppose I want to factor a number N into two primes, p and q

Used in (email) encryption: assumes computers cannot factor large numbers quickly!

Factor(N):

For i = 0 to i = $\sqrt{N}$:

check if (n % i == 0). If so, output p = i, q = n/i.

Runtime? (assume mod and division are constant time). $\sqrt{N}$

Input Size: Examples

Example 1 (factoring): Suppose I want to factor a number N into two primes, p and q

Used in (email) encryption: assumes computers cannot factor large numbers quickly!

Factor(N):

For i = 0 to i = $\sqrt{N}$:

check if (n % i == 0). If so, output p = i, q = n/i.

Runtime? (assume mod and division are constant time). $\sqrt{N}$

Input Size: $n = \log N$

Input Size: Examples

Example 1 (factoring): Suppose I want to factor a number N into two primes, p and q

Factor(N):

For i = 0 to i = $\sqrt{N}$:

check if (n % i == 0). If so, output p = i, q = n/i.

Runtime? (assume mod and division are constant time). $\sqrt{N}$

Input Size: $n = \log N$

Actual Runtime: $f(n) = \sqrt{2^n} = 2^{n/2}$

Resource Analysis – Worst Case ~~Running Time~~ Space

- An algorithm has a **worst case** ~~running time~~ space of $f(n)$ if use $f(n)$ bits of memory
  - Among all possible size-n inputs, the "worst" one will ~~do $f(n)$ "operations"~~ use $f(n)$ bits of memory
  - i.e. $f(n)$ gives the maximum ~~operation count~~ memory usage from among all inputs of size n

Resource Analysis – Best Case Running Time

- An algorithm has a **best** case running time of $f(n)$ if
  - Among all possible size-n inputs, the "**best**" one will do $f(n)$ "operations"
  - i.e. $f(n)$ gives the **minimum** operation count from among all inputs of size n

Analysis Process From 123/143

- Count the number of “primitive operations”
  - +, -, compare, arr[i], arr.length, etc
- Write that count as an expression using n (the input size)
- Put that expression into a “bucket” by ignoring constants and “non-dominant” terms, then put a $O(\ )$ around it.
  - $4n^2 + 8n - 10$ ends up as $O(n^2)$
  - $\frac{1}{2}n + 80$ ends up as $O(n)$
  - $n(n+1)$ ends up as $O(n^2)$

Analysis Process For Us

- Count the number of *chosen* operation(s)
  - Factors to consider when choosing which operation(s) to count
    - **Necessity**: should be necessary for solving the problem
    - **Frequency**: should be the most frequently done (up to a constant factor)
    - **Magnitude**: should be expensive to perform each chosen operation
- Write that count as an expression using n (the input size)
- Put an asymptotic bound on it (one of O, Ω, Θ)
  - More on this soon

Worst Case – Example 1 (Questions)

```
myFunction(List L):
    b = 55 + 5
    c = b / 3
    b = c + 100
    for i=0 to L.size:
        b = b+1
    if (b % 2 == 0):
        c = c + 1
    else:
        for i = 0 to L.size:
            c = c+1;
    return c
```

Questions to ask:

- What are the units of the input size?
- What are the operations we're counting?
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with different inputs?
- Answer:

Worst Case – Example 1 (Answers)

```
myFunction(List L)
    b = 55 + 5  } 1
    c = b / 3   } 1
    b = c + 100   } 1
    for i=0 to L.size:  } 0
        b = b + 1 } 1, n times
    if (b % 2 == 0):  } 1
        c = c + 1 } 1
    else:
        for i = 0 to L.size: } 0
            c = c + 1 } 1, n times
    return c
```

Questions to ask:

- What are the units of the input size?
  - # of items in the list
- What are the operations we're counting?
  - Arithmetic ops (+-*/) (non-loop vars)
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with different inputs?
- Answer:
  - Worst case? 1 + 1 + 1 + n + 1 + n
  - $O(n)$

Worst Case – Example 2 (Questions)

```
beAnnoying(List L)
    List A = []
    for i=0 i to L.size:
        A.append(L[i])
        for j=0 to L.size:
            print ("Hi, I'm annoying")
```

Questions to ask:

- What are the units of the input size?
- What are the operations we're counting?
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with the input size?

Worst Case – Example 2 (Answers)

```
beAnnoying(List L)
    List A = [] } 1
    for i=0 i to L.size: } 0
        A.append(L[i]) } 1, n times
        for j=0 to L.size: } 0
            print ("Hi, I'm annoying") } 1, n^2 times
```

Questions to ask:

- What are the units of the input size?
  - # items
- What are the operations we're counting?
  - appending, printing
  - Printing: $O(n^2)$
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with the input size?

Worst Case Running Time – General Guide

- Add together the time of consecutive statements
- Loops: Sum up the time required through each iteration of the loop
  - If each takes the same time, then [time per loop × number of iterations]
- Conditionals: Sum together the time to check the condition and time of the slowest branch
- Function Calls: Time of the function's body
- Recursion: Solve a recurrence relation

Defining your running time function

- Worst-case complexity:
  - max number of steps algorithm takes on “most challenging” input
- Best-case complexity:
  - min number of steps algorithm takes on “easiest” input
- Average/expected complexity:
  - avg number of steps algorithm takes on random inputs (distribution-dependent)
- Amortized complexity:
  - max total number of steps algorithm takes on M “most challenging” consecutive inputs, divided by M (i.e., divide the max total sum by M).

Amortized Analysis Analogy

- Suppose I'd like to park in a lot where they charge $10 per day to park
- If you are caught in the lot without paying you are given a warning
- If you get 3 warnings, you are charged a $25 fine, and your warnings reset.
- Should you actually pay to park?

Amortized Analysis Analogy

- Suppose I'd like to park in a lot where they charge $10 per day to park
- If you are caught in the lot without paying you are given a warning
- If you get 3 warnings, you are charged a $25 fine, and your warnings reset.
- Should you actually pay to park?
  - If you pay every day then you pay an average of $10 per day
  - If you do not pay then for every three days parking costs $0+$0+$25, for an average of $8.33 per day
    - This is an amortized analysis

ArrayList - Worst Case Time of Add

```
public void add(int value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    int[] oldData = data;
    data = new int[data.length*2];
    for(int i = 0; i < oldData.length; i++)
        data[i] = oldData[i];
}
```

- What is the worst case running time of add?
  - Input size: size of “this”
  - Operations counted: indexing
  - $O(n)$

ArrayList – Amortized Time of Add

```
public void add(int value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    int[] oldData = data;
    data = new int[data.length*2];
    for(int i = 0; i < oldData.length; i++)
        data[i] = oldData[i];
}
```

Every time we resize, we earn `data.length` more adds before the next resize!

- Amortized Analysis Idea:
  - Suppose we have a program that in total does n adds.
  - How much time was spent "on average" across all n?
- Let c be the initial size of `data`
  - The first c adds take: $c + c = 2c$
  - The next $2c$ adds: $2c + 2c = 4c$
  - The next $4c$ adds: $4c + 4c = 8c$
  - Overall: $\frac{14c}{7c} = 2c$