CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 2: Stacks, Queues and Algorithm Analysis

Announcements

- My office hours (CSE2 344)
  - Tuesday, 10 AM – 11 AM
  - Friday, 3 PM - 4 PM
- Exercise #1: Due Monday
  - Computer set up and some programming
- Reading, Weiss
  - Algorithm Analysis: 2.1-2.4
  - Stacks and Queues: 3.1-3.7

First Example: Queue ADT

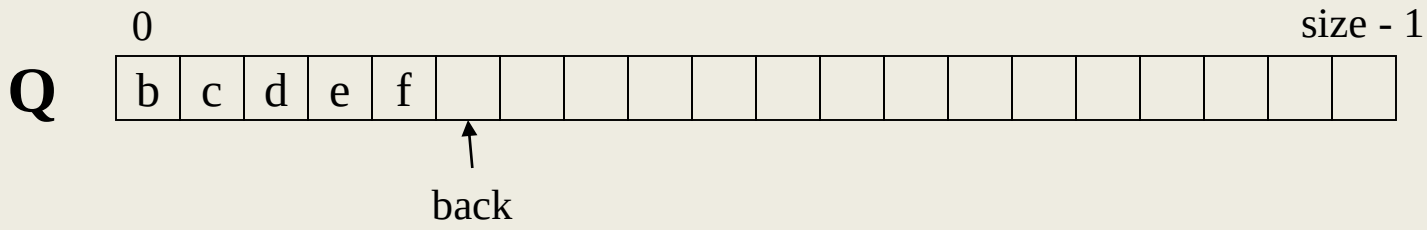
- FIFO: First In First Out
- Queue operations
  - create
  - enqueue
  - dequeue
  - is_empty

Queues in practice

- Print jobs
- File serving
- Phone calls and operators

(Later, we will consider “priority queues.”)

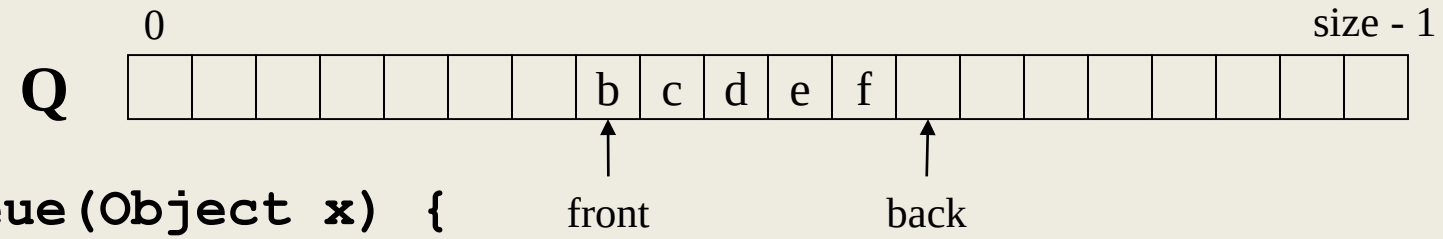
Array Queue Data Structure




```
enqueue(Object x) {
   Q[back] = x
   back = back + 1
}

dequeue() {
   x = Q[0]
   shiftLeftOne()
   back = back - 1
   return x
}
```

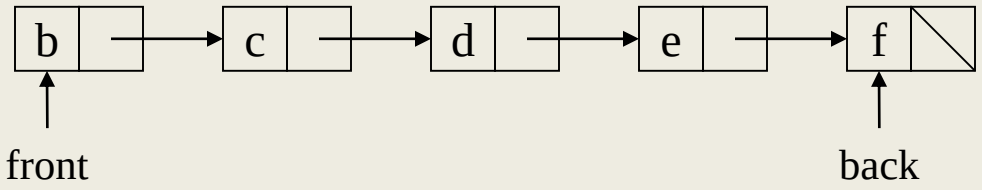
Circular Array Queue Data Structure




```
enqueue(Object x) {
    assert(!is_full())
    Q[back] = x
    back = (back + 1) % Q.size
}

dequeue() {
    assert(!is_empty())
    x = Q[front]
    front = (front + 1) % Q.size
    return x
}
```

Linked List Queue Data Structure




```
void enqueue(Object x) {
   if (is_empty())
        front = back = new Node(x)
   else {
        back.next = new Node(x)
        back = back.next
   }
}
bool is_empty() {
   return front == null
}
```

```
Object dequeue() {
   assert(!is_empty())
   return_data = front.data
   temp = front
   front = front.next
   delete temp
   return return_data
}
```

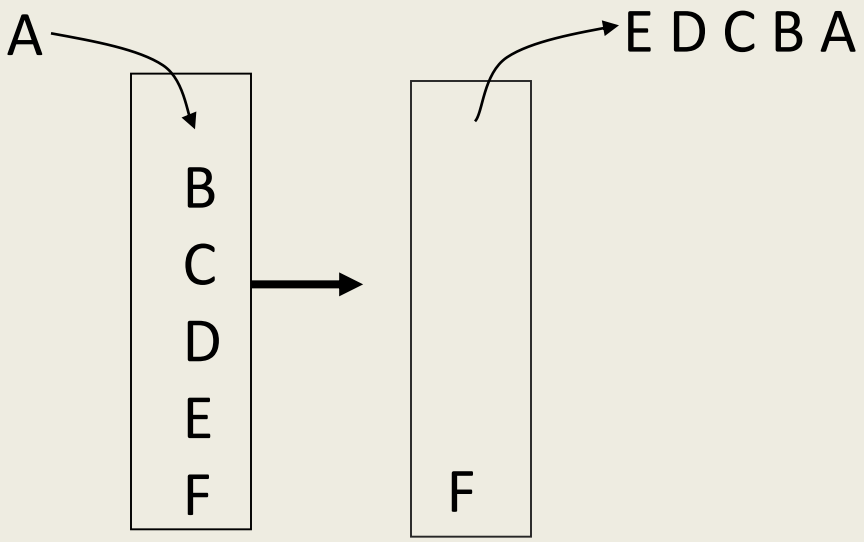
Circular Array vs. Linked List

- Advantages of a circular array

- Advantages of a linked list

Second Example: Stack ADT

- LIFO: Last In First Out
- Stack operations
  - create
  - push
  - pop
  - top
  - is_empty



Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Evaluating postfix or “reverse Polish” notation

Algorithm Analysis

Algorithm Analysis

- Correctness:
  - Does the algorithm do what is intended.
- Performance:
  - Speed: time complexity
  - Memory: space complexity
- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

How to measure performance?

Analyzing Performance

- Focus on Worst Case Time Complexity

Basic operations	Constant time
Consecutive statements	Sum of times
Conditionals	Test, plus larger branch cost
Loops	Sum of iterations
Function calls	Cost of function body
Recursive functions	Solve recurrence relation…

Worst case complexity

Problem size **N**

- **Worst-case complexity**: **max** # steps algorithm takes on "most challenging" input of size **N**

Exercise - Searching

2	3	5	16	37	50	73	75

```
bool ArrayContains(int array[], int n, int key){




}
```

Linear Search Analysis

```
bool LinearArrayContains(int array[], int n, int key ) {
   for( int i = 0; i < n; i++ ) {
        if( array[i] == key )
            // Found it!
            return true;
   }
   return false;
}
```

Best Case:

Worst Case:

Binary Search Analysis

2	3	5	16	37	50	73	75

```
bool BinArrayContains( int array[], int low, int high, int key ) {
    // The subarray is empty
    if( low > high ) return false;

    // Search this subarray recursively
    int mid = (high + low) / 2;
    if( key == array[mid] ) {
        return true;
    } else if( key < array[mid] ) {
        return BinArrayFind( array, low, mid-1, key );
    } else {
        return BinArrayFind( array, mid+1, high, key );
}
```

Best case:

Worst case:

Solving Recurrences
$T(n) = T(n/2) + 7;\ T(1) = 9$

1. Determine the recurrence relations and base cases

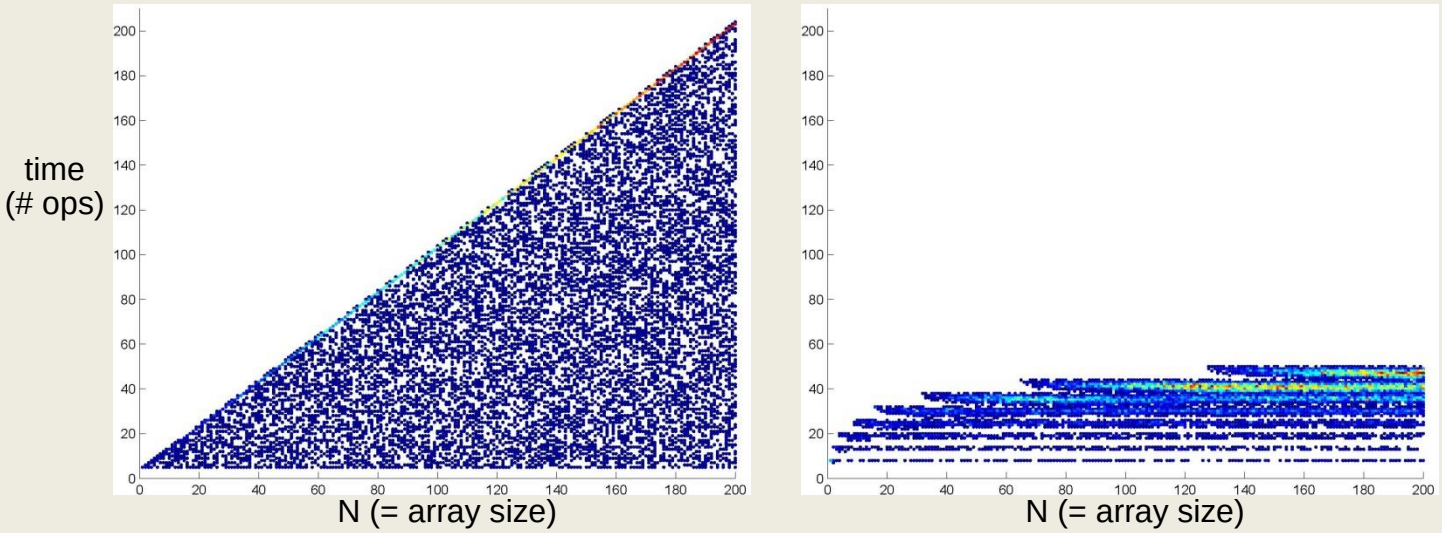
2. Expand relation in terms of number of expansions k

3. Find a closed form by setting k to value that reduces problem to the base case

Linear Search vs Binary Search

	Linear Search	Binary Search
Best Case	4	5 at [middle]
Worst Case	3n+3	$7 \lfloor \log n \rfloor + 9$

Empirical comparison



Linear search

Binary search

Asymptotic Analysis

- Consider only the *order* of the running time
  - A valuable tool when the input gets “large”
  - **Ignores** the effects of ***different machines*** or ***different implementations*** of same algorithm

Asymptotic Analysis

- To find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is $T^{LS}_{worst}(n) = 3n + 3 \in O(n)$
  - Binary search is $T^{BS}_{worst}(n) = 7\lfloor \log_2 n \rfloor + 9 \in O(\log n)$

Remember: the "fastest" algorithm has the slowest growing function for its runtime

Asymptotic Analysis

Eliminate low order terms

- $4n + 5 \Rightarrow$
- $0.5\ n \log n + 2n + 7 \Rightarrow$
- $n^3 + 3\ 2^n + 8n \Rightarrow$

Eliminate coefficients

- $4n \Rightarrow$
- $0.5\ n \log n \Rightarrow$
- $3\ 2^n \Rightarrow$

Properties of Logs

Basic:

- $A^{\log_A B} = B$
- $\log_A A =$

Independent of base:

- $\log(AB) =$
- $\log(A/B) =$
- $\log(A^B) =$
- $\log((A^B)^C) =$

Properties of Logs

Changing base $\rightarrow$ multiply by constant

- For example: $\log_2 x = 3.22 \log_{10} x$

- More generally

$$\log_A n = \left(\frac{1}{\log_B A} \right) \log_B n$$

- Means we can ignore the base for asymptotic analysis (since we're ignoring constant multipliers)