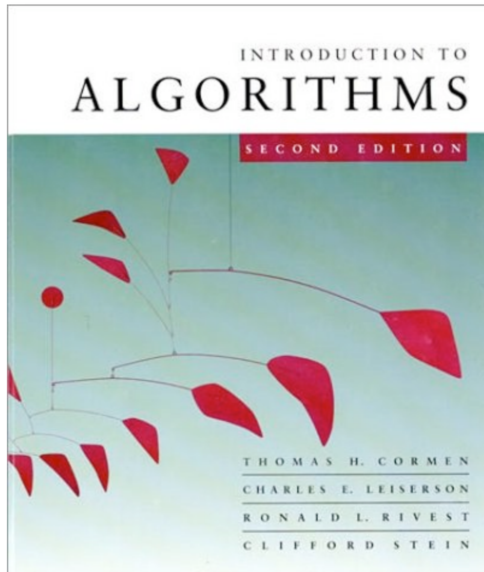
Algorithms and Data Structures

CSE 465

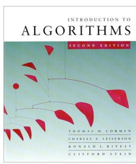


LECTURE 9

Abstract Data Types

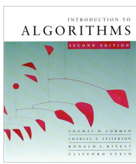
Queues

Sofya Raskhodnikova and Adam Smith



Data Structures

- So far in this class: designing algorithms
  - Inputs and outputs were specified, we wanted to design the fastest algorithm
  - The representation was fixed (e.g. a sorted array)
- Another important question:
  - How can we represent information so that there are fast algorithms for performing important operations?
  - This is the study of **data structures**



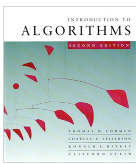
Some important data structures

- arrays
- linked lists
- graphs
- binary search trees
- heaps

What about…

- stacks?
- queues?

Not exactly data structures.
These are **abstract data types**
(note: the text book doesn't distinguish data structures from abstract data types, but we will in this class)

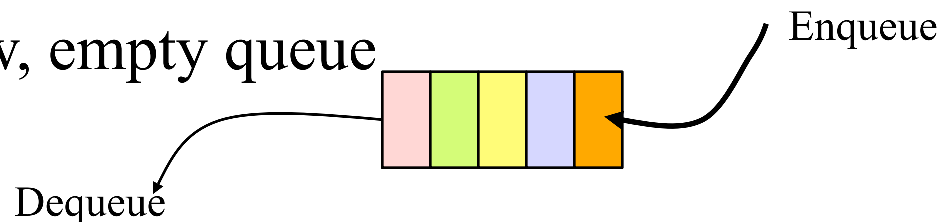


Abstract Data Types

- “Interface” between the real data and the outside world
- Collection of operations to be performed on data
- No algorithms!
  - Just a description of desired outcomes
- Important tool in the design of computer programs
  - First, figure out what you need to do with your data
  - Worry about implementing it later.
- Sort of like a “class”, an “interface” or a “template” in object-oriented programming (but not exactly like any of these)

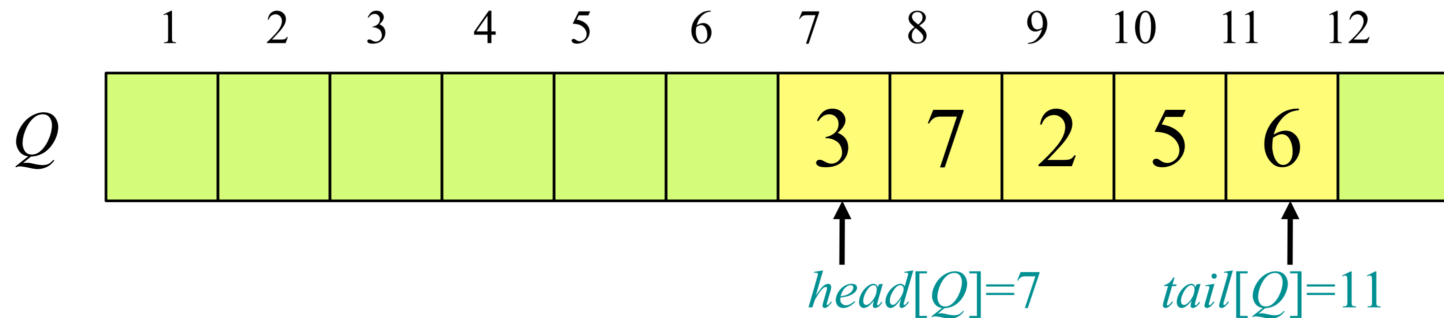
Example: Queues

- Suppose you manage the list of cases waiting for trial at a courthouse
  - You maintain a “bunch” of court cases
  - As cases come in you add them to your list
  - When the court finishes a trial, you find the next case in line and it goes to trial
  - What’s the ADT you’re using?
- A **Queue** holds a *set* of elements and supports
  - Enqueue(Q, x): add x to the rear of the queue
  - Dequeue(Q): get element from the front of the queue and remove it from the queue
  - MakeNew(): create a new, empty queue

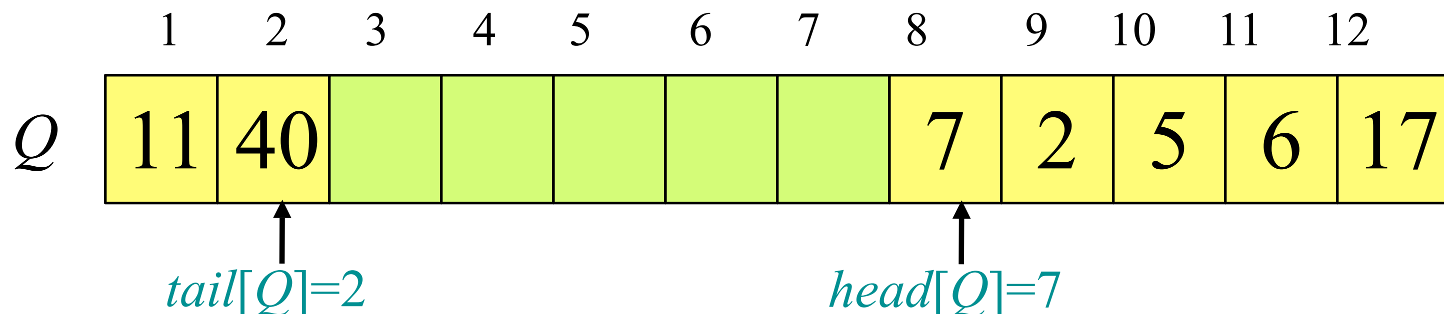


How should we implement a queue?

- One option: an array along with two indices *head* and *tail*

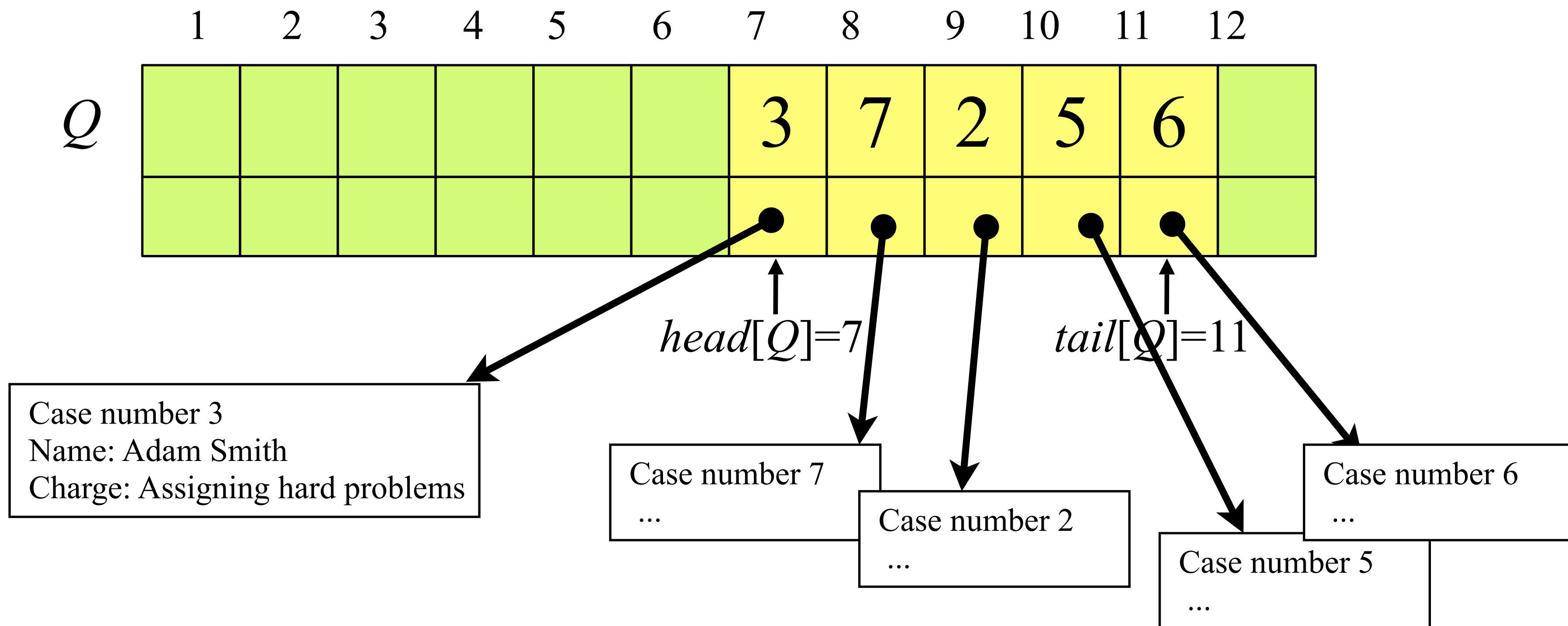


- As elements are added, increment *tail*[*Q*]
- As elements are removed, increment *head*[*Q*]
- Wrap around as necessary
- After Enqueue(*Q*,17), Enqueue(*Q*,11), Enqueue(*Q*,40), Dequeue(*Q*), we get:



Satellite data

- May have other "satellite data" along with each record (case details, name of plaintiff, etc)
- Typically: include a pointer for each element



Pseudocode

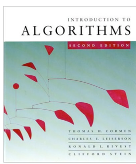
ENQUEUE(Q, x)

1. $Q[tail[Q]] \leftarrow x$
2. **if** $tail[Q] = length[Q]$
3. **then** $tail[Q] \leftarrow 1$
4. **else** $tail[Q] \leftarrow tail[Q] + 1$

Notice that this code doesn't handle what happens when the queue fills up or when it is empty!

DEQUEUE(Q,x)

1. $x \leftarrow Q[tail[Q]]$
2. **if** $head[Q] = length[Q]$
3. **then** $head[Q] \leftarrow 1$
4. **else** $head[Q] \leftarrow head[Q] + 1$
5. **return** x



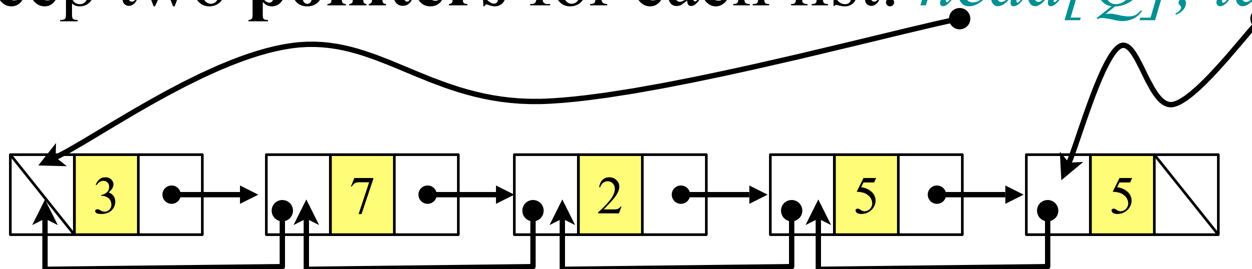
How long do the operations take?

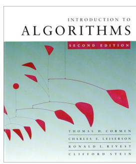
- Enqueue: $O(1)$
- Dequeue: $O(1)$
- MakeNew: $O(1)$ if memory implemented well
- Storage space = length of array n
  - Maximum queue size limited to n
  - Wastes space is size of L is much smaller than n

- What do you do when queue is full?
  - Crash the program? (sometimes)
  - Better solution: allocate bigger array

What about using a linked list?

- Dynamic structure uses memory flexibly
- **Doubly linked list** is a data structure
  - ➢ collection of nodes
  - ➢ Each node has at least three fields
    - next (pointer)
    - previous (pointer)
    - key (depends on application: case number?)
    - may have satellite data here too
  - ➢ Keep two **pointers** for each list: *head[Q], tail[Q]*





Pseudocode for list-based queue

ENQUEUE(Q, x)

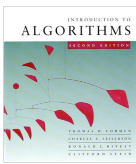
1. $newnode \leftarrow$ New node
2. $key[newnode] \leftarrow x$
3. $prev[newnode] \leftarrow tail[Q]$
4. $next[newnode] \leftarrow$ NIL
5. $next[tail[Q]] \leftarrow newnode$
6. $tail[Q] \leftarrow newnode$

   ▷ Note no check for an empty list.

DEQUEUE(Q, x)

1. $oldnode \leftarrow head[Q]$
2. $head[Q] \leftarrow next[oldnode]$
3. $prev[head[Q]] \leftarrow$ NIL
4. **return** $key[oldnode]$

   ▷ Note: no deallocation, no check for an empty list.



What about using a linked list?

- How long do operations take?
  - Enqueue: O(1)
  - Dequeue: O(1)
  - MakeNew: O(1)
  - Storage: O(size(Q)), i.e. the number of elements currently in the queue
- Better storage use than array, right?
  - But constants are better for arrays
  - Clever allocation of memory can make array also use O(size(Q)) memory (we may see this in later lectures)