# 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?

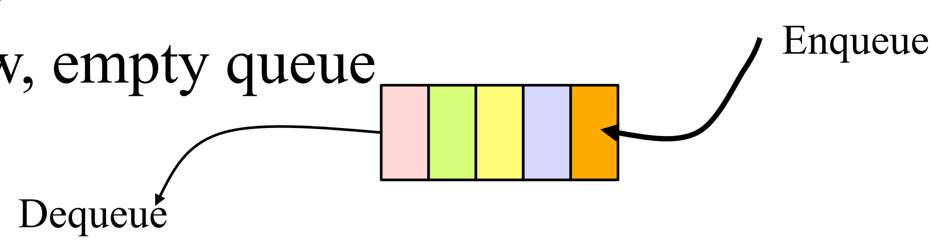
Not exactly data structures.
These are **abstract data types**
(note: the text book doesn't distinguish data structures from abstract data types, but we will in this class)

# 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)

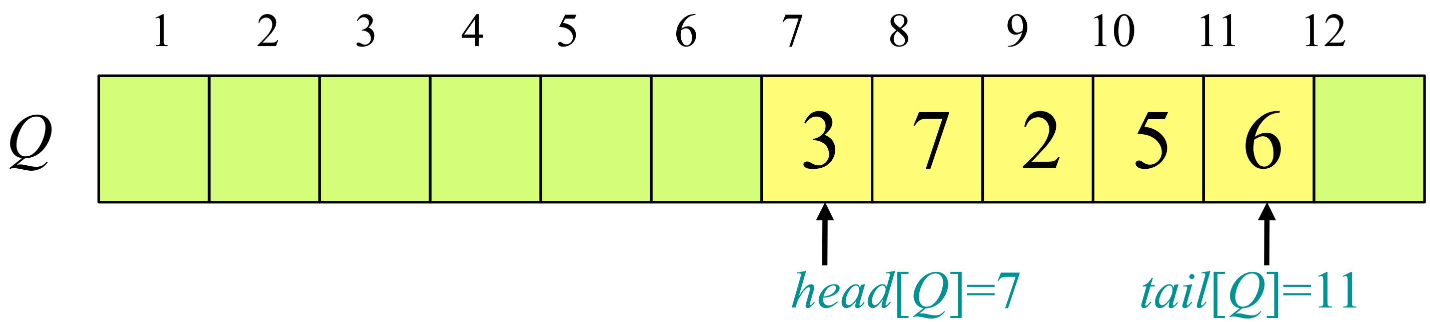
# Example: Queues

- 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

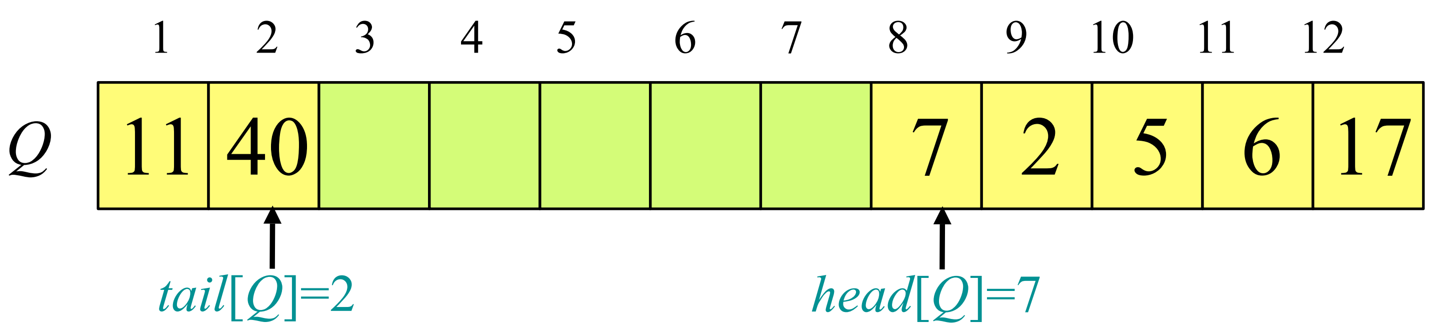


# How should we implement a queue?

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

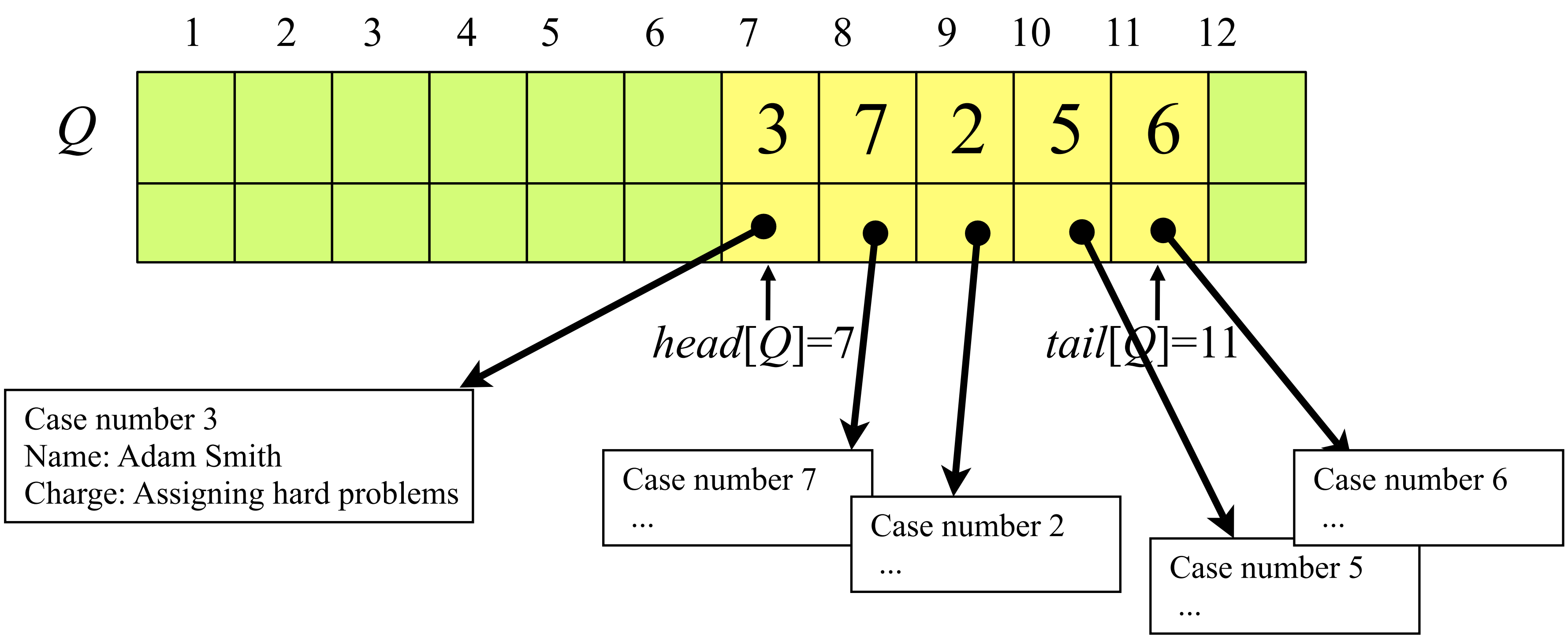



- As elements are added, increment *tail*[*Q*]
- As elements are removed, increment *head*[*Q*]
- Wrap around as necessary
- After Enqueue(*Q*,17), Enqueue(*Q*,11), Enqueue(*Q*,40), Dequeue(*Q*), we get:



# 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($Q$, $x$)

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

DEQUEUE($Q$,$x$)

$x \leftarrow Q[\overset{\text{head}}{\cancel{tail}}[Q]]$
**if** $head[Q] = length[Q]$
  **then** $head[Q] \leftarrow 1$
  **else** $head[Q] \leftarrow head[Q] + 1$
**return** $x$

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

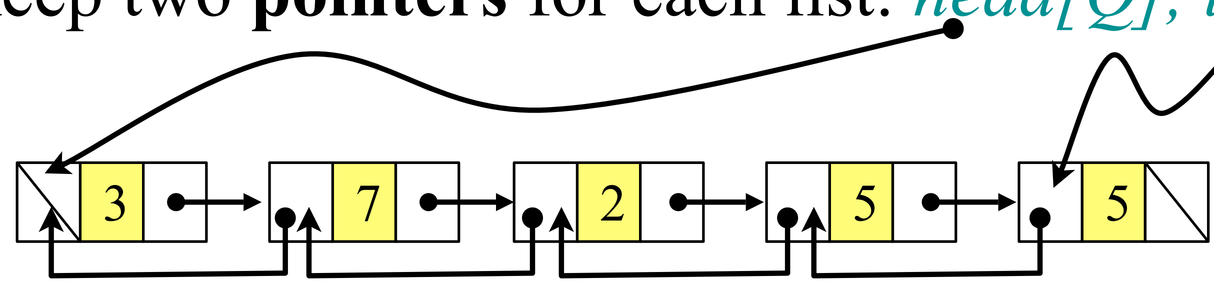
# How long do the operations take?

- 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. $tail[Q] \leftarrow newnode$

   $\triangleright$ 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** $key[oldnode]$

   $\triangleright$ 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)

# Stacks

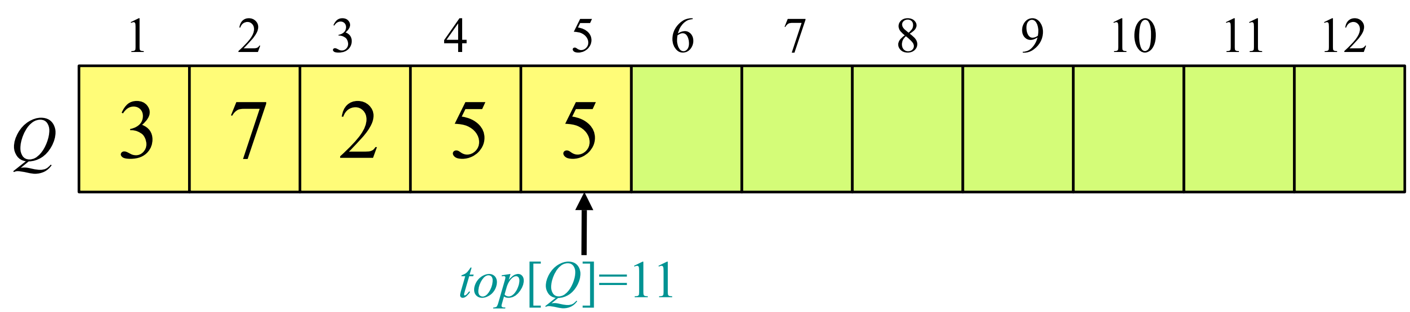
- A **stack** holds a set of elements and supports
  - Push($S$,$x$): add element $x$ to the top of stack $S$
  - Pop($S$): remove the top element from the stack and return its value
  - MakeNew(): Create a new, empty stack
-

# Example: Stacks

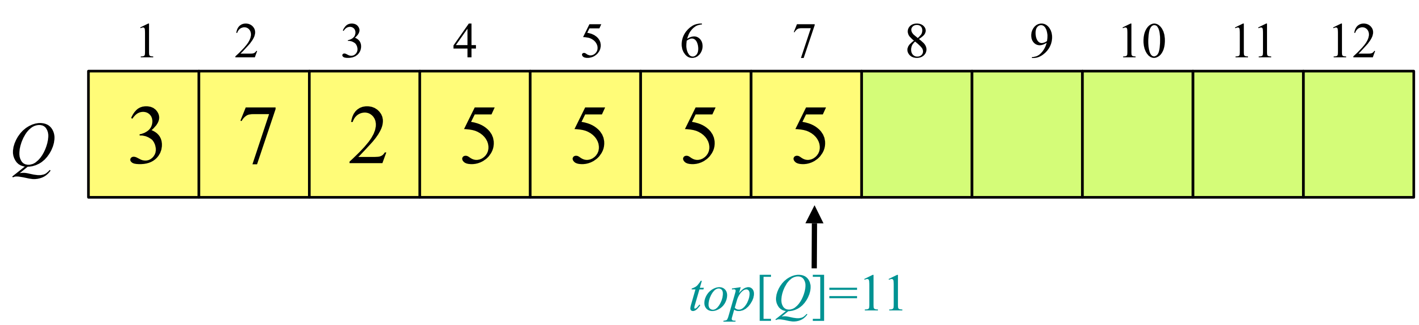
- Suppose you need to check if delimiters (parentheses and brackets) are properly balanced
  {(({})){()}})}    versus {(({}){})(})}
- Scan through the input,
  keeping a list of currently open delimiters
  - When I meet an opening delimiter, add it to my list
  - When I meet a closing delimiter,
    - check if the last thing I added to my list was of the same type.
    - If so, remove it and continue.
    - If not, then output "delimiters not matched."

# How should we implement a stack?

- Usually: an array along with an index *top*




  - As elements are added, increment *top*[*Q*]
  - As elements are removed, decrement *top*[*Q*]



# Algorithms and Data Structures

CMPSC 465

Priority Queues and
Binary Heaps

**Paul Medvedev**

*based on slides by S. Raskhodnikova , A. Smith, K. Wayne, C. Leiserson and E. Demaine.*

1

## Priority Queue Abstract Data Type

- Dynamic set of pairs (key, data), called elements
- Supports operations:
  - **MakeNewPQ**()
  - **Insert**(*S*,*x*) where *S* is a PQ and *x* is a (key,data) pair
  - **Extract-Max**(*S*) removes and returns the element with the highest key value (or one of them if several have the same value)
- Example: managing jobs on a processor, want to execute job in queue with the highest priority
- Can also get a "min" version, that supports **Extract-Min** instead of **Extract-Max**
- Sometimes support additional operations like **Delete**, **Increase-Key**, **Decrease-Key**, etc.

2

## Rooted trees

- Rooted Tree: collection of nodes and edges
  - Edges go down from root (from parents to children)
  - No two paths to the same node
  - Sometimes, children have "names" (e.g. left, right, middle, etc)

## Heaps: Tree Structure

- Data Structure that implements Priority Queue
  - Conceptually: binary tree
  - In memory: (often) stored in an array
- Max-Heap property:
  - For every rode other than root, $key[Parent(i)] \geq key[i]$
- Recall: *complete* binary tree
  - all leaves at the same level, and
  - every internal node has exactly 2 children
- Heaps are nearly complete binary trees
  - Every level except possibly the bottom one is full
  - Bottom layer is filled left to right

4

## Height

- Heap Height =
  - length of longest simple path from root to some leaf
  - e.g. a one-node heap has height 0,
    a two- or three-node heap has height 1, ...
- Exercises:
  - What is are max. and min. number of elements in a heap of height h?
    - ans: min = $2^h$,
      max = $2^{h+1}$ -1
  - What is height as a function of number of nodes n?
    - ans: floor( log(n) )

5

## Array representation

- Instead of dynamically allocated tree, heaps often represented in a fixed-size array
  - smaller constants, works well in hardware.
- Idea: store elements of tree layer by layer, top to bottom, left to right
- Navigate tree by calculating positions of neighboring nodes:
  - Left(i) := 2i
  - Right(i) := 2i+1
  - Parent(i) := floor(i/2)
- Example: [20 15 8 10 7 5 6]

6

## Review Questions

- Is the following a valid max-heap?
  - 20 10 4 9 6 3 2 8 7 5 12 1
  - (If not, repair it to make it valid)

- Is an array sorted in decreasing order a valid max-heap?

7