Stacks and Queues

Download Demo Code <../dsa-stacks-queues-demo.zip>

Goals

- Describe a queue data structure
- Describe a stack data structure
- Compare and contrast stacks / queues
- Implement stacks and queues in JavaScript

Lists ADT Revisited

Lists ADT

Remember: an *abstract data type* defines requirements.

ADT for list:

- Keep multiple items
- Can insert or delete items at any position
- Can contain duplicates
- Preserves order of items

Where's the Bug?

movieTicketSales.js

```
// list, in order, of people who want tickets
ticketBuyers = ["Elie", "Alissa", "Matt", "Michael"];

// ... lots of code

// sell tickets, in order
while (ticketBuyers.length) {
  buyer = ticketBuyers.pop();
  purchase(buyer);
}
```

- Is it right to sell tickets out of order?
- Of course: it's hard to see this bug 500 lines later

What's the Performance Problem?

printJob.js

```
// list of print jobs
jobs = ["resume.doc", "budget.xls", "plan.pdf", "css.css"];

// process list of print jobs in order
while (jobs.length) {
  let job = jobs.shift();
  printJob(job);
}
```

- It's ***O(n)*** to remove from start of array
  - Given that we're removing from end, a LL would be better
- Of course: it's hard to know *how* a general list will be used

Constraints Are Useful

In both cases, we only need *some* of the capability of the List ADT

- add new item (ticket buyer or print job) to end
- remove first item (buyer or job) from start

Knowing this, we could pick better data structure!

If done well, we could prevent mis-use (like buying out of order)

Let's meet two new *ADTs* for collections

Queues

Add at end, remove from beginning

Like a List, Except…

- Items are *only* added to a queue by **enqueueing** them at the *back*
- Items are *only* removed from a queue by **dequeueing** them at the *front*
- Thus, newer items are near back of queue, older items are near front
- **FIFO** for "First-in, first-out"

Typical methods

enqueue(item)
: Add to end

dequeue()
: Remove & return first item

peek()
: Return first item, but don't remove

isEmpty()

Are there items in the queue?

Sometimes there are other common methods, like ***.length()***

Sometimes ***enqueue*** and ***dequeue*** are called ***push*** and ***pop***

Implementation

What's a good implementation for queues?

- Arrays?
- Linked Lists?
- Doubly Linked List?
- Objects?

- *Array*: **no**, dequeing would be ***O(n)***
- *Linked List*: **yes**, both enqueue & dequeue are ***O(1)*** *(head is top)*
- *Doubly Linked List*: **yes**, both enqueue & dequeue are ***O(1)***
- *Object*: **no**, dequeuing is ***O(n)*** *(have to scan whole obj to find low key)*

Stacks

- "I want to order pizza for our party!"
  - In order to do that, I call the pizza place
    - They ask me how many I want
      - I put them on hold to ask my boss the budget
        - She gives amount in CAD, but pizza place takes USD
          - I look up USD→CAD conversion rates in my web browser
        - Now I can convert budget to CAD
      - Now I can tell pizza place my budget
    - …

Like function calls — you return to "previous state" when you pop top task

Like a List, Except…

- Items are *only* added to a stack by **pushing** them onto the *top*
- Items are *only* removed from a stack by **popping** them off the *top*
- Thus, newer items are near top of stack, older items are near bottom
- **LIFO** for *Last-in, first-out*
- Examples: the function call stack, most laundry hampers

Typical methods

push(item)
Add to “top” of stack

pop()
Remove & return top item

peek()
Return (but don’t remove) top item

isEmpty()
Are there items in the stack?

Implementation

What’s a good implementation for stacks?

- Arrays?
- Linked Lists?
- Doubly Linked List?
- Objects?

- *Array*: **yes**, both push & pop are ***O(1)***
- *Linked List*: **yes**, both push & pop are ***O(1)***
- *Doubly Linked List*: **yes**, both push & pop are ***O(1)***
- *Object*: **no**, popping is ***O(n)*** *(have to scan whole obj to find high key)*

Deques

An *ADT* for a “double-ended queue” — push, pop, shift & unshift

Less common than stack or queue

Use Case

A ticket buying application:

- Get in queue to buy ticket: added to end
- Buy ticket: removed from front
- Have question/concern about purchase:
  - Would be unfair to have to go to end of line for question
  - Should be next helped: pushed to front

Some task-allocation systems work this way.

Typical Methods

Method names vary across implementations, but one set:

appendleft()
Add to beginning

appendright()
Add to end

popleft()
Remove & return from beginning

popright()
Remove & return from end

peekleft()
Return (don't remove) beginning

peekright()
Return (don't remove) end

isEmpty()
Are there items in the deque?

Implementation

What's a good implementation for queues?

- Arrays?
- Linked Lists?
- Doubly Linked List?
- Objects?

- *Array*: **no**, appendleft & popleft would be ***O(n)***
- *Linked List*: **no**, popright would be ***O(n)***
- *Doubly Linked List*: **yes** — everything is ***O(1)***
- *Object*: **no**, popleft & popright would be ***O(n)***

Priority Queue

An *ADT* for a collection:

- Add item (with priority)
- Remove highest-priority item

Typical Methods

add(pri, item)
Add item to queue

poll()
Remove & return top-priority item

peek()

Return (don't remove) top-priority item

isEmpty()

Are there items in queue?

Implementation

What's a good implementation for priority queues?

- Arrays?
- Linked Lists?
- Doubly Linked List?

Consider with two strategies:

- Keep unsorted, add to end, find top priority on poll
- Keep sorted, add at right place, top priority is first

Keep unsorted, add to end, find top priority on poll:

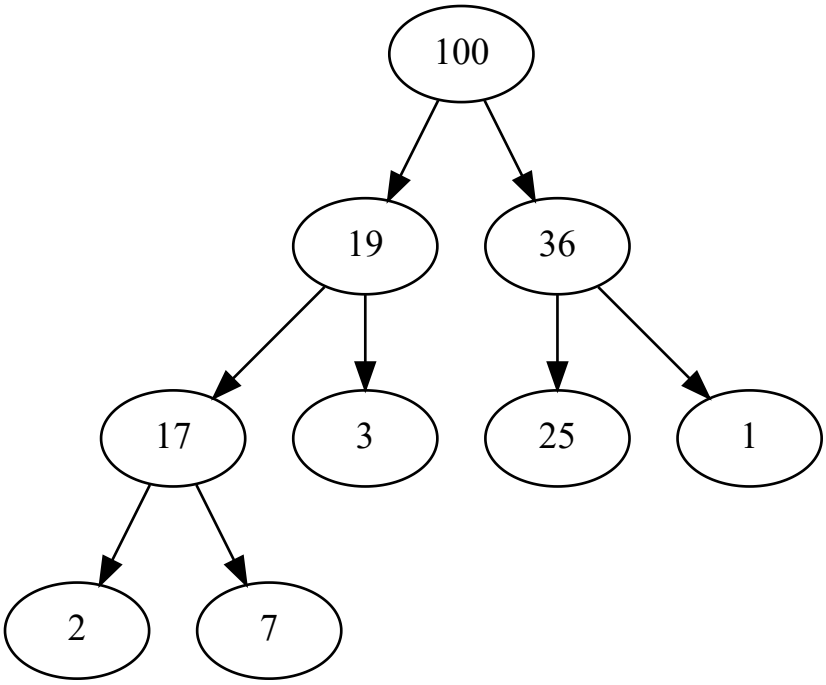
- *Array*: **no**, peek & poll would be ***O(n)***
- *Linked List*: **no**, peek & poll would be ***O(n)***
- *Doubly Linked List*: **no**, peek & poll would be ***O(n)***

Keep sorted, add at right place, top priority is first:

- *Array*: **no**, add & poll would be ***O(n)***
- *Linked List*: **no**, add would be ***O(n)***
- *Doubly Linked List*: **no**, add would be ***O(n)***

Heaps

Data structure optimized for priority queues: *heap*



Resources

Stacks and Overflows <https://medium.com/basecs/stacks-and-overflows-dbcf7854dc67>

To Queue or Not To Queue <https://medium.com/basecs/to-queue-or-not-to-queue-2653bcde5b04>

Learning to Love Heaps <https://medium.com/basecs/learning-to-love-heaps-cef2b273a238>

Rithm School Lecture on Heaps <http://curric.rithmschool.com/r9/slides/dsa-pqueues/>