CS234

MODULE 8 – STACKS AND QUEUES

- Stacks
- Queues
- Deques
- Priorities Queues

Updated: 2018-06-04
What is a Stack

A Stack is an **ADT** that stores a linear collection of values in a last-in, first-out (**LIFO**) manner.

Think of it like a stack of papers. If you take the **first** paper that’s on top of the stack, it will be the **last** paper that was added (i.e. the most recent addition).
# Stack Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack()</td>
<td>Constructor</td>
<td>Makes an empty Stack</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Accessor</td>
<td>Check if Stack is empty</td>
</tr>
<tr>
<td>Push()</td>
<td>Mutator</td>
<td>Adds to the top of the Stack</td>
</tr>
<tr>
<td>Pop()</td>
<td>Mutator</td>
<td>Removes and return the top value</td>
</tr>
<tr>
<td>Top()</td>
<td>Accessor</td>
<td>Returns (but does not remove) the top value.</td>
</tr>
</tbody>
</table>
Stack Implementation

The Python List is basically a stack already

Stack.Push  =  list.append
Stack.Peek  =  list[-1]
Stack.Pop   =  list.pop

We are using the end of the list as the top of the stack.

Why?
Visualizing Stacks

Implementation View

Abstract View
Another Implementation

We can also use a linked list

Stack.push = LinkedList.add_front
Stack.peek = LinkedList.front
Stack.pop = LinkedList.remove_front

We’re using the **front** of the list as the **top** of the stack.

Why?
Array View

Stack Contents

Python List:

Stack
items

16 15 8 4
Which Implementation?

A Dynamic Array (like Python’s list) gives $O(1)$ amortized pushes and pops. However, if the stack is quite full, some operations will be quite slow.

A Linked List always has $O(1)$ complexity for pushes and pops.

Linked List wins! (Usually)
Uses for Stacks

Stacks have a lot of use.

For one thing, it’s how many language keep track of function calls.

- When function A is running, its information is on top of the stack.
- Calling function B pushes function B’s information
- When B returns, its information is popped. Now A is on top again, and can continue to run
Uses for Stacks

Programming languages have many different sorts of bracket. (Parentheses) [Brackets] {Braces} <angle brackets>, and sometimes more.

Consider the line of Python code:

```
foo({1:[2,3,'four','five', str(3+3)], 7:(4*2)},9)
```

Are the parentheses balanced? If not, where’s the problem?
Balanced Bracketing

The algorithm is simple: If you encounter an “open” bracket (meaning ‘(‘, ‘{‘, or ‘[‘) then push it to the stack.

If you encounter a “close” bracket (meaning ‘)’, ‘}’, or ‘]’) then pop from the stack and match the two.

If the stack was empty, or the top element mismatches, the brackets are unbalanced.

If the stack is not empty when the string is processed, the brackets were not balanced.
Python Code

PARTNERS = {'(':')', '[':']', '{':'}'}
def balanced(s):
    stk = Stack()
    for c in s:
        if c in PARTNERS.keys():
            stk.push(PARTNERS[c])
        elif c in PARTNERS.values():
            if stk.empty(): return False
            elif stk.top() != c: return False
            else stk.pop()
    return stk.empty()
Balanced Brackets - Example

Input
[ ( { } [ ] ( ) ) ]

▲

Stack (bottom -> top)
Balanced Brackets - Example

Input

```
[ ( { } [ ] ( ) ) ]
```

Stack (bottom -> top)

```
]
``
Balanced Brackets - Example

Input

\[ [ ( \{ \} [ ] ) ( ) ] \]

▲

Stack (bottom -> top)

\[ ] \)
Balanced Brackets - Example

Input

```
[ ( { } [ ] ( ) ) ]
```

▲

Stack (bottom -> top)

```
] ) ) }
```
Balanced Brackets - Example

Input

\[ [ ( \{ \} [ ] ( ) ) ] \]

▲

Stack (bottom -> top)

\[ ] \)
Balanced Brackets - Example

Input

[ ( { } [ ] ( ) ) ]

Stack (bottom -> top)

[ ]

▲
Balanced Brackets - Example

Input

[ ( { } [ ] ( ) ) ]

Stack (bottom -> top)

] )
Balanced Brackets - Example

Input

\[
[ ( \{ \} [ ] ( ) ) ]
\]

Stack (bottom -> top)

\[
] ) )
\]
Balanced Brackets - Example

Input
[ ( { } [ ] ( ) ) ]
▲

Stack (bottom -> top)
] )
Balanced Brackets - Example

Input
[ ( { } [ ] ( ) ) ]

Stack (bottom -> top)
]

Balanced Brackets - Example

Input

\[
[ ( \{ \} [ ] ( ) ) ]
\]

Stack (bottom -> top)
Balanced Brackets - Example

Input

[ ( { } [ ] ( ) ) ]

Stack (bottom -> top)

Traversal is complete and stack is empty.

Return true (input was balanced).
Mathematics

We have three options for where to put operators:

Infix, aka Python Style:
2 + 3 * 4 + 1 / 3

Prefix, aka Polish Notation (kinda like Racket)
+ + 2 * 3 4 / 1 3

Postfix, aka Reverse Polish Notation
2 3 4 * + 1 3 / +
Infix Notation

Evaluating infix notation is difficult to evaluate efficiently, because the order of operations makes us jump around.

A quadratic isn’t too hard to come up with:

Find leftmost operator with highest precedence:

\[ O(n) \]

Repeat until no operators found

\[ (O(n) \text{ iterations}) \]

\[ O(n) \times O(n) = O(n^2) \]
Infix Notation

If parentheses are allowed, it gets even harder!

In fact, even with parentheses it’s possible to do it in O(n) time. This is a bit complex and beyond the scope of the course, but totally doable (otherwise it would suck to run Python code!)
Prefix Notation

1. Start with an empty Stack
2. Scan input left to right, pushing onto stack
3. If there are ever two numbers in a row on top of the stack, pop 3 things (must be NUM NUM OP)
   1. Perform operation.
   2. Push answer.
   3. Repeat until only 1 number in a row
4. After processing, stack contains answer

This is O(n), but a bit complicated
Postfix Notation

1. Start with an empty Stack
2. Scan input left to right
   1. If input is a number, push it
   2. If input is an operator, pop two numbers, do calculation, push answer
3. Stack contains answer

Much nicer! Never need to peek, only pop

For this reason, old calculators used postfix notation for order of operations. Infix calculators did not support BEDMAS
### Postfix Example

<table>
<thead>
<tr>
<th>Unread Input</th>
<th>Stack (bottom -&gt; top)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 4 * + 3 /</td>
<td></td>
<td>Push 3</td>
</tr>
<tr>
<td>3 4 * + 3 /</td>
<td>3</td>
<td>Push 3</td>
</tr>
<tr>
<td>4 * + 3 /</td>
<td>3 3 4</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>* + 3 /</td>
<td>3 3 4</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>* + 3 /</td>
<td>3</td>
<td>Push L * R</td>
</tr>
<tr>
<td>+ 3 /</td>
<td>3 12</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>+ 3 /</td>
<td></td>
<td>Push L + R</td>
</tr>
<tr>
<td>3 /</td>
<td>15</td>
<td>Push 3</td>
</tr>
<tr>
<td>/</td>
<td>15 3</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>/</td>
<td></td>
<td>Push L // R</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Return Pop</td>
</tr>
</tbody>
</table>
Hardware Stacks

*Did old calculators use a dynamic array or a linked list?*

They had a memory chip that was just a stack. So, it was a fixed length array.

Computers would do the same with their memory: Stack starts at the end of memory. Each push moves back.

Runs on hope: “I hope nobody was using this memory!”

Modern machines have replaced hope with memory segments.
Stack Machines

CPUs usually come in to flavours: register and stack

A register-based CPU can only do calculations on values held in special memory called “registers”. This is what your computer / phone does.

A stack-based CPU does calculations using an operand stack. Not done physically anymore.

- Still common for virtual machines
Aside: Virtual Machines

Interpreted Languages often are compiled to machine code for a virtual machine. E.g. when you run Python code, it gets converted to Python byte-code.

The Python interpreter is simulating a CPU that evaluates this machine code. Registers are fast. A virtual register is not.

Stacks are more efficient if all memory is equal.
Queues

A Queue is the other kind of buffer. A FIFO (First-in, First-out).

If you’ve been to a grocery store, you’ve stood in a queue. North Americans tend to call them “lines”.

When you add to a Queue, the values goes to the back.

When you remove from a Queue, you take from the front.
# Queue Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue()</td>
<td>Constructor</td>
<td>Makes an empty Queue</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Accessor</td>
<td>Check if Queue is empty</td>
</tr>
<tr>
<td>Enqueue()</td>
<td>Mutator</td>
<td>Adds to the back of the Queue</td>
</tr>
<tr>
<td>Dequeue()</td>
<td>Mutator</td>
<td>Removes and return the front value</td>
</tr>
<tr>
<td>Front()</td>
<td>Accessor</td>
<td>Returns (but does not remove) the front value.</td>
</tr>
</tbody>
</table>
Queue Implementations

Hey, class, what should we do???

It’s like a Stack, but we’re adding and removing from different ends. Tricky.
Idea 1: Linked List

Seems easy. Back of the Queue is back of the linked list. Front of the Queue is front of the linked list.

We saw that with a tail reference, we can add to the back in $O(1)$ time, and still remove from the front in $O(1)$ time.

Therefore: All operations have $O(1)$ complexity.
Idea 2: Array (Python List)

Back of list is back of Queue, etc.?

Enqueue = Append: $O(1)$ [amortized]
Dequeue = pop(0): $O(n)$ (need to move all elements)

Well, that’s inefficient. Can we do better? Thoughts?
Idea 2.5: Circular Array

Take that Array, and loop it around on itself. If the length is 6 then:

After index 5 is index 0 (rather than being invalid, it wraps around)
Circular Array – Two Views
Circular Array – Example 1

We could put the values [8, 6, 7, 5, 3] from index 0 through 4:
Queue – Circular Array

We could put the values [8,6,7,5,3] from index 6 through 2:
Queue – Edge Cases

1. Start and End (inclusive)
   • What does empty look like?  Oops

2. Start and End (open end, like a Slice)
   • What does empty look like?  Start == End
   • What does full look like?  Start == End.  Oops

3. Start and Count
   • What does empty look like?  Count == 0
   • What does full look like?  Count == Length of Array
Enqueue(X)

Back = start + count
Back %= len
items[Back] = X
count += 1
Dequeue()

return = items[start]
start += 1
start %= length
count -= 1
Resizing

Often times, a circular array implementation has a fixed size

Resizing is tricky. You cannot use Python list’s automatic resizes as the wrap around will be broken.

Instead: Manually double the length, and copy over, resetting start to 0.
Implementations

Linked List: $O(1)$ for all operations. Easy to do

Dynamic Array: $O(n)$ dequeue, but $O(1)[\text{amortized}]$ enqueue. Very easy (in Python anyway) but less efficient.

Circular Array: $O(1)$ for all operations. (Amortized if you allow resizing). Complicated to get the math right (draw a diagram!)
Who Wins?

I was making a message queue (just for fun). With 100K messages per second and multiple threads enqueueing and dequeueing: There was no measurable difference.

If you can preconfigure it with a maximum queue length, circular array wins.
Deque ADT

- Short for “Double Ended Queue”
- Pronounced “Deck”
  - Pronunciations are *not* going to be tested on the exam.

Sometimes it's handy to have $O(1)$ adds and removes to both ends of a buffer. This is usually called a “double ended queue” (though a double ended stack would be just as accurate).
Deque Implementations

• Singly Linked List: Sucks
• Singly Linked List with Tail: Still Sucks
• Dynamic Array: SUCKS
• Doubly Linked List with Tail: O(1) for all operations
  • Hey, isn’t that A2Q1? Yes, yes it is.
• Dynamic Circular Array: O(1) [amortized] for all operations. But tricky.
When to use a Deque?

If you can’t decide between a stack and a queue.
   (That’s a joke. You should look at your requirements more carefully)
If you need to add and remove from both ends.
   (If you don’t, use a stack or a queue)

Because it’s (probably) a doubly linked list, it uses more memory than a stack or a queue. Use when needed.

Some scheduling algorithms use a Deque
Deques IRL

A coffee shop line is a Deque.

New customers add_back (well, the polite ones do).
When your name is called, you add_front.
Priorities Queues

A priority queue is a queue where items have a priority.

You can think of it like a sorted queue.

Priority is an integer. Smaller number = higher priority
• Think #1 priority is the first thing you should do, then the #2 priorities, etc.

If there are two #1 priorities, you still want to do first one added (FIFO)
Implementations

Sorted Linked List (just need to break ties properly).
  • Remember when I said we’d find a use for these sorted linked lists? No? Well…I did!

Sorted Circular Array (ditto)
  • This sucks, and without the need to binary search, no benefit.
Good Implementations

Both of those are $O(n)$ for enqueue and dequeue. That’s no good.

What IS good?

Skip Lists! If you sort by priority and break ties properly, you have a priority queue:

Enqueue: $O(\log p)$ where $p$ is the priority number
Dequeue: $O(1)$ (its just remove_front)
That’s Average Case Though

Good enough for me!

What else can we do?

If the priority is bounded we could just have an array of regular Queues.

If not, we could have a BST of Queues (sorted by priority).
Array of Queues

If priority is an integer between 0 and 10, you have 11 queues.

Enqueue(item, pri): Queues[pri].Enqueue(item).
  • O(1)

Dequeue(): Queues[i].Dequeue() for smallest i where the queue isn’t empty.
  • O(1) – Since 11 is a constant.
Tree of Queues

Enqueue and Dequeue will be $O(\log p)$ where $p$ is the number of different priorities currently being tracked.

To do this we need to wait for a self-balancing BST like the AVL Tree.

A min-heap will also work (we’ll see those, too).
## Postfix Example

<table>
<thead>
<tr>
<th>Unread Input</th>
<th>Stack (bottom -&gt; top)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 4 * + 3 /</td>
<td></td>
<td>Push 3</td>
</tr>
<tr>
<td>3 4 * + 3 /</td>
<td>3</td>
<td>Push 3</td>
</tr>
<tr>
<td>4 * + 3 /</td>
<td>3 3</td>
<td>Push 4</td>
</tr>
<tr>
<td>* + 3 /</td>
<td>3 3 4</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>* + 3 /</td>
<td>3</td>
<td>Push L * R</td>
</tr>
<tr>
<td>+ 3 /</td>
<td>3 12</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>+ 3 /</td>
<td></td>
<td>Push L + R</td>
</tr>
<tr>
<td>3 /</td>
<td>15</td>
<td>Push 3</td>
</tr>
<tr>
<td>/</td>
<td>15 3</td>
<td>R=Pop, L=Pop</td>
</tr>
<tr>
<td>/</td>
<td></td>
<td>Push L // R</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Return Pop</td>
</tr>
</tbody>
</table>