Data Structures and Algorithms
Stacks and Queues

Chris Brooks

Department of Computer Science
University of San Francisco
Stacks and queues are two of the most common data structures. Based on the List, but with more limited access to elements. Simplicity allows for efficiency, easy implementation.
A Stack is a Last-In, First-Out (LIFO) data structure.

Like a stack of trays in a cafeteria.

Stack Operations:

- Add an element to the top of the stack (push)
- Remove the top element (pop)
- Look at the top element (peek)
- Check if the stack is empty
6-2: Uses of a stack

6 Postfix notation calculator
   ▶ 3 4 + -> 7
   ▶ 3 4 + 5 - -> 2
6 Parenthesis checker
6 Recursion
6 Interrupt handling
6-3: Stack Implementation

Array:

- Like the list, we allocate memory in advance.
- Keep a pointer for the 'top' of the stack.
Array:

- Stack elements are stored in an array
- Top of the stack is the *end* of the array
  - If the top of the stack was the beginning of the array, a push or pop would require moving all elements in the array
- Push: `data[top++] = elem`
- Pop: `elem = data[--top]`
Array Implementation:
  push
  pop
  empty()
Array Implementation:

- push: $\Theta(1)$
- pop: $\Theta(1)$
- empty(): $\Theta(1)$

- Pros: fast, easy to implement
- Cons: Resizing the stack is $\Theta(n)$
Linked List:
Linked List:

6 Stack elements are stored in a linked list
6 Top of the stack is the *front* of the linked list
6 push: top = new Link(elem, top)
6 pop: elem = top.element(); top = top.next()
6-9: $\Theta()$ For Stack Operations

Linked List Implementation:
- push
- pop
- empty()
6-10: Θ(1) For Stack Operations

Linked List Implementation:

- push: Θ(1)
- pop: Θ(1)
- empty(): Θ(1)

- **Pros:** Fast, can grow/shrink dynamically,
- **Cons:** Allocating/freeing memory may be costly.
A Queue is a Last-In, First-Out (FIFO) data structure. Like a line at the post office or grocery store.

Queue Operations:

- Add an element to the end (tail) of the Queue (enqueue)
- Remove an element from the front (head) of the Queue (dequeue)
- Check the item at the front of the queue (peek)
- Check if the Queue is empty
6-12: Uses of a queue

6 Scheduling
   ▶ You’ll get lots of experience with queues in OS

6 Searching (web crawling)

6 Handling requests

6 Print queues
Linked List:
6-14: Queue Implementation

Linked List:

- Maintain a pointer to the first and last element in the Linked List
- Add elements to the back of the Linked List
- Remove elements from the front of the linked list
  - Enqueue: tail.setNext(new link(elem,null));
    tail = tail.next()
  - Dequeue: elem = head.element();
    head = head.next();
6-15: $\Theta()$ For Queue Operations

Linked List Implementation:

- enqueue $\Theta(1)$
- dequeue $\Theta(1)$
- empty() $\Theta(1)$

- Pros: Fast, can grow/shrink dynamically,
- Cons: Allocating/freeing memory may be costly.
Array:

- Can’t just use a flat array, as with a stack.
- Queue will wind up ’sliding up’ the array.
6-17: Queue Implementation

Array:

- Store queue elements in a circular array
- Maintain the index of the first element (head) and the next location to be inserted (tail)

  Enqueue: \( \text{data}[\text{tail}] = \text{elem}; \)
  \( \text{tail} = (\text{tail} + 1) \mod \text{size} \)

  Dequeue: \( \text{elem} = \text{data}[\text{head}]; \)
  \( \text{head} = (\text{head} + 1) \mod \text{size} \)
Array Issues:

- What do head and tail look like when the queue is empty?
- What about when there is one element in the queue?
- What about when the queue is full?
Array Implementation:
  - enqueue $\Theta(1)$
  - dequeue $\Theta(1)$
  - empty() $\Theta(1)$

- Pros: Fast
- Cons: Issues with full vs. empty, fixed size
“Minimum Stacks” have one additional operation:

- minimum: return the minimum value stored in the stack

Can you implement a $O(n)$ minimum?
“Minimum Stacks” have one additional operation:

- minimum: return the minimum value stored in the stack

Can you implement a $O(n)$ minimum?
Can you implement a $\Theta(1)$ minimum?

push, pop must remain $\Theta(1)$ as well!