04-0: **Abstract Data Types**

- An Abstract Data Type is a definition of a type based on the operations that can be performed on it.
- An ADT is an *interface*
- Data in an ADT cannot be manipulated directly – only through operations defined in the interface

04-1: **Abstract Data Types**

- To define an ADT, give the operations that can be performed on it
- The ADT says nothing about *how* the operations are performed
- Could have different implementations of the same ADT
- First ADT for thie class: Stack

04-2: **Stack**

A Stack is a Last-In, First-Out (LIFO) data structure.

Stack Operations:

- Add an element to the top of the stack
- Remove the top element
- Check if the stack is empty

04-3: **Stack Implementation**

Array:

04-4: **Stack Implementation**

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]

04-5: **Stack Implementation**

- See code & Visualizaion

04-6: $\Theta()$ **For Stack Operations**

Array Implementation:
push
pop
empty()

04-7: $\Theta()$ **For Stack Operations**

Array Implementation:

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

04-8: **Stack Implementation**

Linked List:

04-9: **Stack Implementation**

Linked List:

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

04-10: **Stack Implementation**

- See code & Visualization

04-11: $\Theta()$ **For Stack Operations**

Linked List Implementation:

push
pop
empty()

04-12: $\Theta()$ **For Stack Operations**

Linked List Implementation:

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

04-13: **Queue**

A Queue is a First-In, First-Out (FIFO) data structure.

Queue Operations:

- Add an element to the end (tail) of the Queue
- Remove an element from the front (head) of the Queue
- Check if the Queue is empty

04-14: **Queue Implementation**

Linked List:

04-15: **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();

04-16: **Queue Implementation**

- See code & visualization

04-17: **Queue Implementation**

Array:

04-18: **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: data[tail] = elem;
  tail = (tail + 1) % size
- Dequeue: elem = data[head];
  head = (head + 1) % size

04-19: **Queue Implementation**

- Se code & visualization

04-20: **Modifying Stacks**

"Minimum Stacks" have one additional operation:

- minimum: return the minimum value stored in the stack

Can you implement a $O(n)$ minimum?

04-21: **Modifying Stacks**

"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!

04-22: **Modifying Queues**

- We'd like our array-based queues and our linked list-based queues to behave in the same way
- How do they behave differently, given the implementation we've seen so far?

04-23: **Modifying Queues**

- We'd like our array-based queues and our linked list-based queues to behave in the same way
- How do they behave differently, given the implementation we've seen so far?
  - Array-based queues can get full
  - How can we fix this?

04-24: **Modifying Queues**

- Growing queues

- If we do a Enqueue on a full queue, we can:
  - Create a new array, that is twice as big as the old array
  - Copy all of the data across to the new array
  - Replace the old array with a new array
- Why is this a little tricky?

04-25: **Modifying Queues**

- Growing queues
  - If we do a Enqueue on a full queue, we can:
    - Create a new array, that is twice as big as the old array
    - Copy all of the data across to the new array
    - Replace the old array with a new array
  - Why is this a little tricky?
    - Queue could wrap around the end of the array (examples!)

04-26: **Modifying Queues**

- Growing stacks/queues
  - Why do we double the size of the queue when it gets full, instead of just increasing the size by a constant amount
  - Hint – think about running times

04-27: **Modifying Queues**

- Growing stacks/queues
  - What is the running time for a single enqueue/push, if we allow the stack to grow? (by doubling the size of the stack/queue when it is full)
  - What is the running time for n enqueues/pushes?

04-28: **Modifying Queues**

- Growing stacks/queues
  - What is the running time for a single enqueue/push, if we allow the stack to grow? (by doubling the size of the stack/queue when it is full)
    - $O(n)$, for a stack size of n
  - What is the running time for n enqueues/pushes?
    - $O(n)$ – so each push/enqueue takes $O(1)$ on average

04-29: **Modifying Queues**

- Growing stacks/queues
  - What is the running time for a single enqueue/push, if we allow the stack to grow? (by adding k elements when the stack/queue is full)
  - What is the running time for n enqueues/pushes?

04-30: **Modifying Queues**

- Growing stacks/queues
  - What is the running time for a single enqueue/push, if we allow the stack to grow? (by adding k elements when the stack/queue is full)
    - $O(n)$, for a sack size of n
  - What is the running time for n enqueues/pushes?
    - $O(n * n/k) = O(n^2)$, if k is a constant
    - Each enqueue/dequeue takes time $O(n)$ on average!

04-31: **Amortized Analysis**

- Figuring out how long an algorithm takes to run on average, by adding up how long a sequence of operations takes, is called *Amortized Analysis*
- Washing Machine example
  - Cost of a washing machine
  - Amortized cost per wash
- We'll take quite a bit about amortized analysis, complete with some more formal mathematics, later in the semester.