Lecture 4: Data Structures I

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: October 15, 2024

Announcements

1. Second Quiz, due Friday
    - Similar format to before
        - One question, select all correct answers
        - 20 minute time limit
    - Covers asymptotic (Big-O) notation
        - Lectures 03 and 04
        - Relevant reading from **CLRS**
    - Quiz is **closed resource**
        - No books, notes, internet, etc.
        - Do not discuss until after submission deadline (Friday night, after midnight)
2. Programming Assignment 1: Discuss on Thursday
    - Due 13 November
3. Attendance Code:

Meeting Goals

- Finish discussion of asymptotic notation
- Introduce Abstract Data Types:
  - Stack
  - Queue
  - Priority Queue
- Discuss array-backed and linked list-backed implementations of Stacks and Queues
- Introduce amortized analysis

Asymptotic Notation

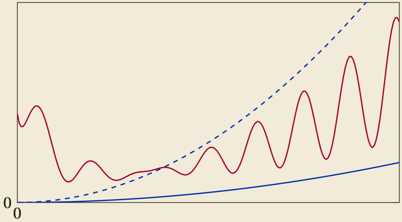
From Last Time

Definition

Suppose f and g are functions from $\mathbf{N}$ to $\mathbf{R}^+$. Then we say that $f = O(g)$ (read: f is *big O of g*) if there exist constants $N_0 \in \mathbf{N}$ and $C \in \mathbf{R}$ such that for all $n \in \mathbf{N}$

$$n \geq N_0 \implies f(n) \leq Cg(n).$$

Equivalently, $f = O(g) \iff \limsup \frac{f(n)}{g(n)} < \infty$



Proposition

Suppose $f, f_1, f_2, g, g_1, g_2, h$ are functions and a is any constant. Then:

1. $(\forall n\, f(n) \leq a) \implies f = O(1)$
2. $(\forall n\, f(n) \leq g(n)) \implies f = O(g)$
3. $f = O(g) \implies a \cdot f = O(g)$
4. $f = O(g)$ and $g = O(h) \implies f = O(h)$
5. $f = O(h)$ and $g = O(h) \implies f + g = O(h)$
6. $f_1 = O(g_1)$ and $f_2 = O(g_2) \implies f_1 \cdot f_2 = O(g_1 \cdot g_2)$

Variations of O

- $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$
  - Example: $4n^2 + 3n + 7 = \Theta(n^2)$

- $f = \Omega(g)$ if $g = O(f)$
  - Example: $0.01n^2 - 7n = \Omega(n^2)$

- $f = o(g)$ if for every $\varepsilon > 0$, there exists N_0 such that $n \geq N_0 \implies \frac{f(n)}{g(n)} < \varepsilon$.
  - Equivalently: $f = o(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
  - Example: $n^{1.999} = o(n^2)$

- $f = \omega(g)$ if $g = o(f)$
  - Example: $0.01n^{2.01} = \omega(n^2)$

Variations of O

- $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$
  - Example: $4n^2 + 3n + 7 = \Theta(n^2)$
- $f = \Omega(g)$ if $g = O(f)$
  - Example: $0.01n^2 - 7n = \Omega(n^2)$
- $f = o(g)$ if for every $\varepsilon > 0$, there exists N_0 such that $n \geq N_0 \implies \frac{f(n)}{g(n)} < \varepsilon$.
  - Equivalently: $f = o(g) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
  - Example: $n^{1.999} = o(n^2)$
- $f = \omega(g)$ if $g = o(f)$
  - Example: $0.01n^{2.01} = \omega(n^2)$

Mnemonic for Variations

Big-O	(in)equality
ω	$>$
Ω	$\geq$
Θ	$\approx$
O	$\leq$
o	$<$

Variations of O

- $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$
  - Example: $4n^2 + 3n + 7 = \Theta(n^2)$
- $f = \Omega(g)$ if $g = O(f)$
  - Example: $0.01n^2 - 7n = \Omega(n^2)$
- $f = o(g)$ if for every $\varepsilon > 0$, there exists N_0 such that $n \geq N_0 \implies \frac{f(n)}{g(n)} < \varepsilon$.
  - Equivalently: $f = o(g) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
  - Example: $n^{1.999} = o(n^2)$
- $f = \omega(g)$ if $g = o(f)$
  - Example: $0.01n^{2.01} = \omega(n^2)$

Mnemonic for Variations

Big-O	(in)equality
ω	$>$
Ω	$\geq$
Θ	$\approx$
O	$\leq$
o	$<$

More Properties

- $f_1 = O(g_1)$ and $f_2 = o(g_2) \implies f_1 \cdot f_2 = o(g_1 \cdot g_2)$
- $f_1 = \Omega(g_1)$ and $f_2 = \omega(g_2) \implies f_1 \cdot f_2 = \omega(g_1 \cdot g_2)$

Interpretation

Suppose:

- two algorithms A and B for solving the same problem
- running time of A is f, running time of B is g
- $f = o(g)$

Consider running A on a slow machine M_1 and B on a fast machine M_2. Then: regardless of how much slower M_1 is than M_2, for *sufficiently large* inputs, A will complete faster than B.

Interpretation

Suppose:

- two algorithms A and B for solving the same problem
- running time of A is f, running time of B is g
- $f = o(g)$

Consider running A on a slow machine M_1 and B on a fast machine M_2. Then: regardless of how much slower M_1 is than M_2, for *sufficiently large* inputs, A will complete faster than B.

The Moral. Efficient *algorithms* are better than faster hardware.

- little-o notation gives the "right" abstraction to formalize this relationship

Common Orders of Growth

Named orders of growth:

name	asymptotic growth
constant	$O(1)$
logarithmic	$O(\log n)$
polylogarithmic	$O(\log^c n)$
linear	$O(n)$
almost linear	$O(n \log^c n)$
quadratic	$O(n^2)$
polynomial	$O(n^c)$
exponential	$O(c^n)$

Common Orders of Growth

Named orders of growth:

name	asymptotic growth
constant	$O(1)$
logarithmic	$O(\log n)$
polylogarithmic	$O(\log^c n)$
linear	$O(n)$
almost linear	$O(n \log^c n)$
quadratic	$O(n^2)$
polynomial	$O(n^c)$
exponential	$O(c^n)$

Relationships

Between classes:

For all $a, b > 0$

- $a = o(\log^b n)$
- $\log^a n = o(n^b)$
- $n^a = o(b^n)$

Common Orders of Growth

Named orders of growth:

name	asymptotic growth
constant	$O(1)$
logarithmic	$O(\log n)$
polylogarithmic	$O(\log^c n)$
linear	$O(n)$
almost linear	$O(n \log^c n)$
quadratic	$O(n^2)$
polynomial	$O(n^c)$
exponential	$O(c^n)$

Relationships

Between classes:
For all $a, b > 0$

- $a = o(\log^b n)$
- $\log^a n = o(n^b)$
- $n^a = o(b^n)$

Within classes:
For all a, b, $a < b$

- $\log^a n = o(\log^b n)$
- $n^a = o(n^b)$
- $a^n = o(b^n)$

Example

Example

Compare the asymptotic growth of the following functions:

1. $f(n) = 2n^2 + 2^{n/2}$
2. $g(n) = \log^2 n + \sqrt{n}$
3. $h(n) = n + n\log n + n^{3/2}$

Linear ADTs and Data Structures

Abstract Data Types and Data Structures

Abstract Data Types (ADTs)

An **abstract data type** gives a formal specification of a task to be performed:

- List of supported operations (syntax)
- The effects of applying the operations (semantics)

Abstract Data Types and Data Structures

Abstract Data Types (ADTs)

An **abstract data type** gives a formal specification of a task to be performed:

- List of supported operations (syntax)
- The effects of applying the operations (semantics)

Data Structures

A **data structure** specifies

- how data is represented
- how the supported operations are performed (i.e., what algorithms are used)
- what are the costs of the operations

Abstract Data Types and Data Structures

Abstract Data Types (ADTs)

An **abstract data type** gives a formal specification of a task to be performed:

- List of supported operations (syntax)
- The effects of applying the operations (semantics)

Data Structures

A **data structure** specifies

- how data is represented
- how the supported operations are performed (i.e., what algorithms are used)
- what are the costs of the operations

Question. Why is it useful to separate ADTs from Data Structure?

- Can swap different data structures for same ADT
  - applications *using* the functionality will not be broken
  - different data structures may be more efficient in some applications
- Better abstractions
- Generic lower bounds

The Stack ADT

Stacks, Intuitively

Goal: to store a *collection* of elements

- elements arranged as in a stack of books
- can only access top-most element:
  - put a new book on the stack
  - look at the top-most book
  - remove the top-most book

The Stack ADT

Stacks, Intuitively

Goal: to store a *collection* of elements

- elements arranged as in a stack of books
- can only access top-most element:
  - put a new book on the stack
  - look at the top-most book
  - remove the top-most book

Stacks, Formally

- S is the state of the stack, initially $S = \emptyset$
- $S.\textsc{Push}(x) : S \mapsto Sx$
- $S.\textsc{Top}() :$ returns x_{n-1} where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{Pop}() : Sx \mapsto S$, returns x
- $S.\textsc{Empty}()$ returns $\textsc{True} \iff S = \emptyset$

The Stack ADT

Stacks, Intuitively

Goal: to store a *collection* of elements

- elements arranged as in a stack of books
- can only access top-most element:
  - put a new book on the stack
  - look at the top-most book
  - remove the top-most book

Stacks, Formally

- S is the state of the stack, initially $S = \varnothing$
- $S.\textsc{Push}(x) : S \mapsto Sx$
- $S.\textsc{Top}()$: returns x_{n-1} where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{Pop}() : Sx \mapsto S$, returns x
- $S.\textsc{Empty}()$ returns $\textsc{True} \iff S = \varnothing$

Tons of Applications!

- Executing programs (call stack)
- Parsing/evaluating arithmetic expression
- Syntax checking (parenthesis)
- ...

Try It Yourself!

PollEverywhere Question

What is the result of calling Top() after the following sequence stack operations:

Push(1)

Push(2)

Push(3)

Pop()

Push(4)

Push(5)

Pop()

Push(6)

Pop()

Pop()

pollev.com/comp526

Try It Yourself!

PollEverywhere Question

What is the result of calling $\textsc{Top}()$ after the following sequence stack operations:

$\textsc{Push}(1)$

$\textsc{Push}(2)$

$\textsc{Push}(3)$

$\textsc{Pop}()$

$\textsc{Push}(4)$

$\textsc{Push}(5)$

$\textsc{Pop}()$

$\textsc{Push}(6)$

$\textsc{Pop}()$

$\textsc{Pop}()$

Stacks, Formally

- S is the state of the stack, initially $S = \varnothing$
- $S.\textsc{Push}(x) : S \mapsto Sx$
- $S.\textsc{Top}()$: returns x_{n-1} where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{Pop}() : Sx \mapsto S$, returns x
- $S.\textsc{Empty}()$ returns $\textsc{True} \iff S = \varnothing$

Try It Yourself!

PollEverywhere Question

What is the result of calling TOP() after the following sequence stack operations:

PUSH(1)

PUSH(2)

PUSH(3)

POP()

PUSH(4)

PUSH(5)

POP()

PUSH(6)

POP()

POP()

Linked List Backed Stack Implementation

Idea

- Store each element in a Node
- Each Node stores
  - the value of an element in the stack
  - a *reference* to the Node storing the next element

```
class Node
    datavalue
    Node next
end class
```

ignores empty stack condition

```
class ListStack
    Node head
    procedure Push(x)
        n ← new Node
        n.data ← x
        n.next ← head
        head ← n
    end procedure
    procedure Pop
        n ← head
        head ← n.next
        return n.data
    end procedure
    procedure Top
        return head.data
    end procedure
end class
```

Issues with Linked List Stacks

Issues

- NODEs waste space
  - must store reference for each entry
- Following chains of reference is costly
  - memory access is non-local
  - sequential memory access is more efficient

```
class ListStack
    Node head ← ∅
    procedure Push(x)
        n ← new Node
        n.data ← x
        n.next ← head
        head ← n
    end procedure
    procedure Pop
        n ← head
        head ← n.next
        return n.data
    end procedure
    procedure Top
        return head.data
    end procedure
end class
```

Arrays as ADTs

Informally, arrays are indexed lists of elements:

$$a = \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline l & i & v & e & r & p & o & o & l \\ \hline \end{array}$$

Array Operations (ADT):

- **create** an array of size n
- **get** the element at index i:
  - $a[4]$ returns r
- **set** the value at index i to a prescribed value
  - $a[5] \leftarrow c$

Array Operation Costs (Data Structure)

- **create** an array of size n has cost $O(n)$
- **get** and **set** have cost $O(1)$

Array Backed Stack Implementation

Idea:

- Store elements in the stack in an array
  - access array values by *index*
  - neighboring values at adjacent indices
    $\implies$ sequential access
- Only overhead: store index of head (top)

```
class ArrayStack
    a ← new array
    head ← 0
    procedure Push(x)
        a[head] ← x
        head ← head + 1
    end procedure
    procedure Pop
        head ← head − 1
        return a[head]
    end procedure
    procedure Top
        return a[head − 1]
    end procedure
end class
```

Array Backed Stack Implementation

Idea:

- Store elements in the stack in an array
    - access array values by *index*
    - neighboring values at adjacent indices

      $\Longrightarrow$ sequential access
- Only overhead: store index of head (top)

```
class ArrayStack
    a ← new array
    head ← 0
    procedure Push(x)
        a[head] ← x
        head ← head + 1
    end procedure
    procedure Pop
        head ← head − 1
        return a[head]
    end procedure
    procedure Top
        return a[head − 1]
    end procedure
end class
```

What is the issue here?

Resizing Arrays

The Problem: Arrays are *fixed size!*

- What if we don't know the (maximum) size of the stack in advance?

Resizing Arrays

The Problem: Arrays are *fixed size!*

- What if we don't know the (maximum) size of the stack in advance?

A Solution: Make a larger array when necessary!

- Must copy contents of old array into new array…

  …this is costly!

Increasing stack capacity

```
class ArrayStack
    a ← new array
    …
    procedure IncreaseCapacity(k)
        n ← Size(a)
        b ← new array of size n + k
        for i = 0, 1, …, n − 1 do
            b[i] ← a[i]
        end for
        head ← b
    end procedure
end class
```

Resizing Arrays

The Problem: Arrays are *fixed size!*

- What if we don't know the (maximum) size of the stack in advance?

A Solution: Make a larger array when necessary!

- Must copy contents of old array into new array…

  …this is costly!

Increasing stack capacity

```
class ArrayStack
    a ← new array
    ...
    procedure IncreaseCapacity(k)
        n ← Size(a)
        b ← new array of size n + k
        for i = 0, 1, ..., n − 1 do
            b[i] ← a[i]
        end for
        head ← b
    end procedure
end class
```

Question. What is the running time of IncreaseCapacity?

Two Strategies

Design Question. When our array runs out of room, by how much should we increase the stack capacity?

Strategy 1. Increase the capacity by $k = 1$ each time.

- Why increase the size more than we need to?

Two Strategies

Design Question. When our array runs out of room, by how much should we increase the stack capacity?

Strategy 1. Increase the capacity by $k = 1$ each time.

- Why increase the size more than we need to?

Strategy 2. Increase the capacity by n each time!

- Maybe we'll need more extra space?

Two Strategies

Design Question. When our array runs out of room, by how much should we increase the stack capacity?

Strategy 1. Increase the capacity by $k = 1$ each time.

- Why increase the size more than we need to?

Strategy 2. Increase the capacity by n each time!

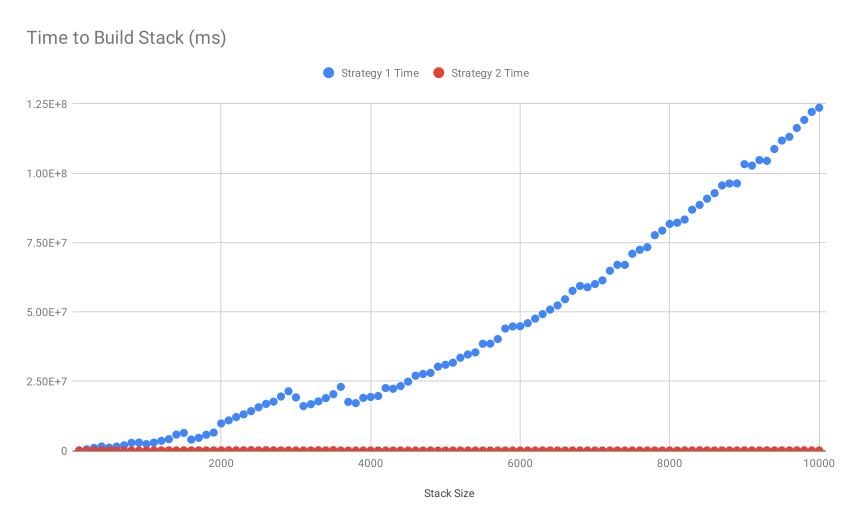
- Maybe we'll need more extra space?

PollEverywhere Question

Which strategy will lead to better performance?

pollev.com/comp526

Running Time Comparison



Understanding the Discrepancy

Question. Why was the difference in running time so dramatic?

Observation. Both strategies have *worst-case* running time of $\Theta(n)$ for INCREASECAPACITY

- Strategy 1 may incur this on *every* PUSH operation
  - Overall running time $\Theta(n^2)$

Understanding the Discrepancy

Question. Why was the difference in running time so dramatic?

Observation. Both strategies have *worst-case* running time of $\Theta(n)$ for INCREASECAPACITY

- Strategy 1 may incur this on *every* PUSH operation
  - Overall running time $\Theta(n^2)$
- For Strategy 2, INCREASECAPACITY only gets called when the stack size is $1, 2, 4, 8, \ldots, 2^k, \ldots, n$.
  - If cost of resizing n' is $c \cdot n'$, what is total resize cost?

Amortized Analysis

Goal. To analyze the worst-case running time of a *sequence* of operations.

- Amortized cost = largest average cost per operation averaged over all sequences.

Amortized Analysis

Goal. To analyze the worst-case running time of a *sequence* of operations.

- Amortized cost = largest average cost per operation averaged over all sequences.

Banker's View

- Each operation has a (financial) cost
- Cost can be paid:
  - from pocket
  - from bank account
- For each operation, can
  - withdraw from account
  - deposit to account

Amortized Analysis

Goal. To analyze the worst-case running time of a *sequence* of operations.

- Amortized cost = largest average cost per operation averaged over all sequences.

Banker's View

- Each operation has a (financial) cost
- Cost can be paid:
  - from pocket
  - from bank account
- For each operation, can
  - withdraw from account
  - deposit to account

A sequence of operations has amortized cost c if for each operation:

1. the operation is paid for (from pocket or bank account)
2. at most c value is paid from pocket and/or *deposited* during each operation

Amortized Analysis of Strategy 2

Setup. Suppose we apply Strategy 2 (double the capacity when full):

- PUSH(x) has cost $c_1 = O(1)$ if the array is not full,
- PUSH(x) has cost $c_2 = O(n)$ if the array is full.

Amortized Analysis of Strategy 2

Setup. Suppose we apply Strategy 2 (double the capacity when full):

- PUSH(x) has cost $c_1 = O(1)$ if the array is not full,
- PUSH(x) has cost $c_2 = O(n)$ if the array is full.

PollEverywhere Question

How much money must we add to our bank account after each (not full) PUSH to ensure our balance is at least c_2 before the next resize?

pollev.com/comp526

Amortized Analysis of Strategy 2

Setup. Suppose we apply Strategy 2 (double the capacity when full):

- PUSH(x) has cost $c_1 = O(1)$ if the array is not full,
- PUSH(x) has cost $c_2 = O(n)$ if the array is full.

Completing the analysis:

- If current capacity is n, last resize was at capacity $n/2$
- There were (at least) $n/2$ non-resizing PUSH operations before next resize
- Must pay c_2 for next resize
- It suffices to put $c_2/(n/2) = 2c_2/n$ in bank each operation

On each non-resizing operation, we pay c_1 out of pocket, and $2c_2/n$ into the bank
$\implies$ the amortized cost is $c_1 + 2c_2/n = O(1) + \frac{1}{n}O(n) = O(1)$.

The Moral. A single resize may cost $\Theta(n)$, but the average cost over sequences of operations is always $O(1)$ (if we're careful).

The Queue ADT

Queues, Intuitively

Goal: to store a *collection* of elements

- elements arranged as in a queue at Tesco
- new people enter the **back** of the queue
- only the person at the **front** of the queue can be removed (serviced)

The Queue ADT

Queues, Intuitively

Goal: to store a *collection* of elements

- elements arranged as in a queue at Tesco
- new people enter the **back** of the queue
- only the person at the **front** of the queue can be removed (serviced)

Queues, Formally

- S is the state of the queue, initially $S = \varnothing$
- $S.\textsc{Enqueue}(x) : S \mapsto xS$
- $S.\textsc{Front}()$: returns x_{n-1} where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{Dequeue}() : Sx \mapsto S$, returns x
- $S.\textsc{Empty}()$ returns $\textsc{True} \iff S = \varnothing$

The Queue ADT

Queues, Intuitively

Goal: to store a *collection* of elements

- elements arranged as in a queue at Tesco
- new people enter the **back** of the queue
- only the person at the **front** of the queue can be removed (serviced)

Tons of Applications!

- Scheduling
- Messaging
- ...

Queues, Formally

- S is the state of the queue, initially $S = \varnothing$
- $S.\textsc{Enqueue}(x) : S \mapsto xS$
- $S.\textsc{Front}() :$ returns x_{n-1} where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{Dequeue}() : Sx \mapsto S$, returns x
- $S.\textsc{Empty}()$ returns $\textsc{True} \iff S = \varnothing$

List Backed Queues

Idea

- Store each element in a NODE
- Store references to NODE:
  - head at the front of the queue
  - tail at the back of the queue

```
class ListQueue
    Node head
    Node tail
    procedure Enqueue(x)
        n ← new Node
        n.data ← x
        tail.next ← n
        tail ← n
    end procedure
    procedure Dequeue
        n ← head
        head ← n.next
        return n.data
    end procedure
end class
```

List Backed Queues

Idea

- Store each element in a NODE
- Store references to NODE:
  - head at the front of the queue
  - tail at the back of the queue

Issues:

- Similar to linked list stack implementation
  - Locality of reference
  - NODE memory overhead

```
class ListQueue
    Node head
    Node tail
    procedure Enqueue(x)
        n ← new Node
        n.data ← x
        tail.next ← n
        tail ← n
    end procedure
    procedure Dequeue
        n ← head
        head ← n.next
        return n.data
    end procedure
end class
```

Array Backed Queues

Idea:

- Store elements in the stack in an array
- Maintain indices of head and tail

Ignores resizing/checking if full

```
class ArrayQueue
    a ← new array, size n
    head, tail ← 0
    procedure Enqueue(x)
        a[tail] ← x
        tail ← tail + 1
    end procedure
    procedure Dequque
        head ← head + 1
        return a[head − 1]
    end procedure
end class
```

Array Backed Queues

Idea:

- Store elements in the stack in an array
- Maintain indices of head and tail

What is the problem here?

Ignores resizing/checking if full

```
class ArrayQueue
    a ← new array, size n
    head, tail ← 0
    procedure Enqueue(x)
        a[tail] ← x
        tail ← tail + 1
    end procedure
    procedure Dequque
        head ← head + 1
        return a[head − 1]
    end procedure
end class
```

Array Backed Queues

Idea:

- Store elements in the stack in an array
- Maintain indices of head and tail

The fix:

- Use *circular arrays*
- Perform index arithmetic *modulo n* (array size)
- All operations are then $O(1)$
  - amortized $O(1)$ time if resizing by doubling size

Ignores resizing/checking if full

```
class ArrayQueue
    a ← new array, size n
    head, tail ← 0
    procedure Enqueue(x)
        a[tail] ← x
        tail ← tail + 1 mod n
    end procedure
    procedure Dequque
        head ← head + 1 mod n
        return a[head − 1 mod n]
    end procedure
end class
```

The (Min) Priority Queue ADT

Priority Queues, Intuitively

Goal: to store a *collection* of elements

- Each element x has an associated *priority*, $p(x)$
- New elements **inserted** with prescribed priorities
- Can access/remove element with the *minimum* priority in the collection

The (Min) Priority Queue ADT

Priority Queues, Intuitively

Goal: to store a *collection* of elements

- Each element x has an associated *priority*, $p(x)$
- New elements **inserted** with prescribed priorities
- Can access/remove element with the *minimum* priority in the collection

Priority Queues, Formally

- S is the state of the queue, initially $S = \varnothing$
- $S.\textsc{Insert}(x, p(x)) : S \mapsto xS$
- $S.\textsc{Min}()$: returns x_0 where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{RemoveMin}() : xS \mapsto S$, returns x
- $S.\textsc{DecreaseKey}(x, p')$ $S = x_0 x_1 \cdots x_{i-1} x x_{i+1} \cdots x_{n-1} \mapsto x_0 x_1 \cdots x_{j-1} x x_j x_{i-1} x_{i+1} \cdots x_{n-1}$
  - $p(x_j) \leq p'(x) < p(x_{j+1})$

The (Min) Priority Queue ADT

Priority Queues, Intuitively

Goal: to store a *collection* of elements

- Each element x has an associated *priority*, $p(x)$
- New elements **inserted** with prescribed priorities
- Can access/remove element with the *minimum* priority in the collection

Priority Queues, Formally

- S is the state of the queue, initially $S = \varnothing$
- $S.\textsc{Insert}(x, p(x)) : S \mapsto xS$
- $S.\textsc{Min}()$: returns x_0 where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{RemoveMin}() : xS \mapsto S$, returns x
- $S.\textsc{DecreaseKey}(x, p')$ $S = x_0 x_1 \cdots x_{i-1} x x_{i+1} \cdots x_{n-1} \mapsto x_0 x_1 \cdots x_{j-1} x x_j x_{i-1} x_{i+1} \cdots x_{n-1}$
  - $p(x_j) \leq p'(x) < p(x_{j+1})$

For Next Time

- Think about implementing min priority queues with linked lists and stacks
- Consider the running times of the priority queue operations

Next Time: Trees!

- Heaps
- Binary Search Trees
- Balanced Binary Trees

Scratch Notes