Lecture 3:
Asymptotic Analysis
Trees & Tree Traversals
Stacks and Queues
Binary Search Trees (and other trees)
WARNING
BEWARE OF MATH
Program Efficiency

• How much memory a program uses
  – Memory is cheap
• How much time a program uses
Testing Time Efficiency

• Benchmarks
  – Hardware specs?
  – Which programming language?
  – Other processes running?

• Estimate with statement count?
  – int a = 1; // 1 “unit” of time?
  – int[] b = new int[100]; // 100 “units” of time?
  – b[15] = 2 + 2; // 1… 2… 3 (?) “units” of time?
Big-Oh Notation

• Classify program running times into groups
  – E.g. $O(n^2)$, $O(2^n)$

• Formal definition (version 1):

\[ f(n) \in O(g(n)) \]

\[ \exists M, N > 0 : \forall n > N, f(n) < M \times g(n) \]
Big-Oh Notation

• Classify program running times into groups
  – E.g. $O(n^2), O(2^n)$
• Formal definition (version 1):

$$f(n) \in O(g(n))$$

There exist positive constants $M$ and $N$ such that for all $n > N$, $f(n) < M \times g(n)$
Big-Oh Notation

• Classify program running times into groups
  – E.g. \(O(n^2), O(2^n)\)

• Formal definition (version 1):

\[ f(n) \in O(g(n)) \]

There exist positive constants \(M\) and \(N\) such that for all \(n > N\), \(f(n) < M \times g(n)\)

(Upper bound on worst case running time)
Big-Oh Example 1

There exist positive constants $M$ and $N$ such that for all $n > N$, $f(n) < M \times g(n)$

\[ f(n) = 3n + 5 \]

\[ g(n) = n^2 + 2n + 3 \]

\[ N = 2 \]

\[ M = 1 \]
Input interpretation:

plot $\frac{3 + 2n + n^2}{5 + 3n}$ $n = 0$ to $3$

Plot:

Source: wolframalpha.com
Input interpretation:

\[
\text{plot } \frac{3 + 2n + n^2}{5 + 3n} \quad n = 0 \text{ to } 100
\]
Big-Oh Notation

• Classify program running times into groups
  – E.g. $O(n^2)$, $O(2^n)$
• Formal definition (version 2):

$$f(n) \in O(g(n))$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$
Big-Oh Example 2

\[ f(n) = 3n + 5 \]
\[ g(n) = n^2 + 2n + 3 \]

\[
\lim_{{n \to \infty}} \frac{f(n)}{g(n)} = \lim_{{n \to \infty}} \frac{3n + 5}{n^2 + 2n + 3} = \lim_{{n \to \infty}} \frac{3n}{n^2}
\]

\[
\lim_{{n \to \infty}} \frac{n}{n^2} = \lim_{{n \to \infty}} \frac{1}{n} = 0 < \infty
\]
Shortcuts

• Get rid of additive terms with smaller growth order
  – E.g. $n^2 + 2n + 3$ becomes $n^2$

• Get rid of constant factors
  – E.g. $10n^2$ becomes $n^2$

• NOTE: Can’t drop non-constant factors!
  – Common mistake: $n \log^2(n)$ becomes $n$
Big-Oh Example 3

\[ f(n) = 3n \log_3(3n) \]
\[ g(n) = 5n \log_2(n) \]

\[ \lim_{n \to \infty} \frac{3n \log_3(3n)}{5n \log_2(n)} = \lim_{n \to \infty} \frac{3 \log_3(3n)}{5 \log_2(n)} \]
\[ \lim_{n \to \infty} \frac{\log_3(3n)}{\log_2(n)} = \lim_{n \to \infty} \frac{\log_3 3 + \log_3 n}{\log_3 2} \]
\[ \lim_{n \to \infty} \frac{\log_3 n}{\log_3 n} = 1 < \infty \]
\( \Omega \)

- \( \Omega: f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n)) \)
  - Lower bound / best case

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0
\]
• $\Theta$:

\[
f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in O(g(n)) \\ f(n) \in \Omega(g(n)) \end{cases} \text{ and } \lim_{n \to \infty} \frac{f(n)}{g(n)} = c; \ 0 < c < \infty
\]

– Tight bound
Big-Oh Example 4

\[ f(n) = 3n + 5 \]
\[ g(n) = n^2 + 2n + 3 \]

• Showed: \[ f(n) \in O(g(n)) \]

• Shortcut:
  – Show: \[ f(n) \in \Theta(n) \] and \[ g(n) \in \Theta(n^2) \]
  – Use hierarchy of common time complexity classes
<table>
<thead>
<tr>
<th>Notation</th>
<th>Common name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(1) )</td>
<td>Constant</td>
</tr>
<tr>
<td>( \Theta(\log n) )</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>( \Theta(n^c), 0 &lt; c &lt; 1 )</td>
<td>Polynomial</td>
</tr>
<tr>
<td>( \Theta(n) )</td>
<td>Linear</td>
</tr>
<tr>
<td>( \Theta(n \log n) )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>( \Theta(n^2) )</td>
<td>Quadratic</td>
</tr>
<tr>
<td>( \Theta(n^3) )</td>
<td>Cubic</td>
</tr>
<tr>
<td>( \Theta(n^c), c &gt; 1 )</td>
<td>Polynomial</td>
</tr>
<tr>
<td>( \Theta(c^n), c &gt; 1 )</td>
<td>Exponential</td>
</tr>
<tr>
<td>( \Theta(n!) )</td>
<td>Factorial</td>
</tr>
</tbody>
</table>
Source: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/
Big O, Ω, Θ

• How well does a program scale?
  – Youtube video quality
  – Video game graphics settings
  – Google results
  – Facebook users
Story time

• Simplex algorithm
  – Developed by George Dantzig of UC Berkeley
  – Solves linear programs:
    
    $$\begin{align*}
    \text{min } & x_1 + x_2 \\
    x_1 + 2x_2 & \geq 4 \\
    3x_1 + x_2 & \geq 5 \\
    x_1 & \geq 0, \quad x_2 \geq 0
    \end{align*}$$

  – Worst case exponential time, but fast enough polynomial time in practice
Linked List

- Each node has at most one "next" node:
Tree

• Each node can have more than one “next” (child) node:
Terminology

• **Node:**
  – Contains an item
  – Similar to linked list node

• **Root:**
  – “Top” level / “first” node
  – Has no parent
  – Similar to linked list head
  – Tree can only have one root

• **Leaf**
  – “Bottom” / “last” node
  – Has no children
  – Similar to last node of linked list
  – Tree can have multiple leaf nodes
Terminology

• Children
  – Tree nodes can have multiple children
  – Similar to linked list node’s “myNext”

• Parent
  – Each tree node can only have 1 parent
  – Tree nodes can share a parent (siblings)
  – Similar to linked list node’s “myPrev”
Terminology

• **Edge**
  – Connects two nodes (pink arrows)

• **Path**
  – A series of edges that connect two nodes through intermediary nodes

• **Depth**
  – Length of path from root node

• **Height**
  – Length of path to deepest descendant
Linked List -> Tree

• TreeNode has instance variable ArrayList<TreeNode> children
  – ListNode has instance variable myNext
• Tree has instance variable myRoot
  – LinkedList has instance variable myHead
• TreeNode and ListNode both have myItem
Tree Uses

• Parsing text
  – Parse tree: sentence structure (e.g. English)
  – Abstract Syntax Tree: source code structure (e.g. Java)
Tree Uses

- Directory structure:
  - Children: everything inside a directory
  - Parent: directory containing stuff
Project 2

• File / folder compression
• Specs out this weekend
• Much more difficult than project 1
  – So start early!
Midterm 1 Post-mortem

• Mean: 28/45, Standard deviation: 9
Upcoming Events

• Tonight: CS61BL dessert potluck / board game night!
  – 7pm to 11pm
• Friday: 61ABC Hackathon
  – 5pm to midnight
• Both in Wozniak Lounge (438 Soda)
• Free food!
Stack

• Last In, First Out (LIFO)
  – Most recent item added is next one removed
• Like dish tower at a buffet
Stack

• Last In, First Out (LIFO)
  – Most recent item added is next one removed
• Like dish tower at a buffet
Stack

• Last In, First Out (LIFO)
  – Most recent item added is next one removed
• Like dish tower at a buffet
Stack

• Last In, First Out (LIFO)
  – Most recent item added is next one removed
• Like dish tower at a buffet
Stack Operations

- **push**: add item to front / top of stack
- **pop**: remove first item of stack
- **peek**: access first item of stack w/o returning it
- **empty**: check if stack has any items in it
### Constructor Summary

**Constructors**

**Constructor and Description**

**Stack()**  
Creates an empty Stack.

### Method Summary

#### All Methods

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean</code></td>
<td><code>empty()</code></td>
</tr>
<tr>
<td></td>
<td>Tests if this stack is empty.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E</code></td>
<td><code>peek()</code></td>
</tr>
<tr>
<td></td>
<td>Looks at the object at the top of this stack without removing it from the stack.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E</code></td>
<td><code>pop()</code></td>
</tr>
<tr>
<td></td>
<td>Removes the object at the top of this stack and returns that object as the value of this function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E</code></td>
<td><code>push(E item)</code></td>
</tr>
<tr>
<td></td>
<td>Pushes an item onto the top of this stack.</td>
</tr>
</tbody>
</table>
Stack Example 1

// start with empty stack
// start with empty stack

push(1)
// start with empty stack

push(1)

push(2)
// start with empty stack
push(1)
push(2)
pop() // 2

Top of stack 1
Stack Example 1

// start with empty stack
push(1)
push(2)
pop() // 2
peek() // 1
// start with empty stack
push(1)
push(2)
pop() // 2
peek() // 1
empty() // false
Stack Example 1

// start with empty stack
push(1)
push(2)
pop() // 2
peek() // 1
empty() // false
pop() // 1

Top of stack
Stack Example 1

// start with empty stack

push(1)
push(2)

pop() // 2
peek() // 1

empty() // false

pop() // 1
empty() // true

Top of stack
Stack Uses

• Call stack (function calls)
  – Make a function call: push stack frame onto stack
  – Exit a function: pop off stack
  – Stack overflow: too many stack frames on stack
Stack Uses

- Parenthesis matching
  - \[1 + (2 \times 3) / (4 - 5)\]
- HTML tag matching
  - <a> stuff </a>
- LaTeX \begin{} \end{} matching
Stack Example 2

( { [ ] } ) ( )

// start with empty stack
// start with empty stack
push (
Stack Example 2

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

// start with empty stack
push ( 
push { 

Top of stack
Stack Example 2

// start with empty stack
push ( 
push { 
push [ 
Top of stack
Stack Example 2

( { [ ] } ) ( )

// start with empty stack
push ( 
push { 
push [ 
pop ]

Top of stack
// start with empty stack
push ( 
push { 
push [ 
pop ] 
pop } 
)
Stack Example 2

 functioning as a stack: 

push ( 
push { 
push [ 
pop ] 
pop } 
push ( 

// start with empty stack

Top of stack
Stack Example 2

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

// start with empty stack
push (  
push {  
push [  
push [  
pop ]  
pop ]  
pop }  
push (  
pop )
```

Top of stack
Stack Example 2

```
// start with empty stack
push ( 
push { 
push [ 
pop ] 
pop } 
pop ) 
push ( 
pop ) 
pop )
```

---

Top of stack
// start with empty stack
Stack Example 3

// start with empty stack
push (  

Top of stack

(  

[  

{ ] [ ] } ( ) )
Stack Example 3

( { [ ] } ( ) )

// start with empty stack
push ( 
push { 

Top of stack
Stack Example 3

( { [ ] } ) ( )

// start with empty stack
push (
push {
// error

Top of stack
Implementing a Stack

- **Linked list**
  - Push: append to front
  - Pop: remove from front
- **Array list**
  - Keep track of last element’s index
  - Push: add to end of list
  - Pop: remove from end of list
Queues

• First In, First Out (FIFO)
  – Element that has been in queue for the longest gets out first

• Like people standing in a line
Queues

• First In, First Out (FIFO)
  – Element that has been in queue for the longest gets out first

• Like people standing in a line
Queues

• First In, First Out (FIFO)
  – Element that has been in queue for the longest gets out first

• Like people standing in a line
Queues

• First In, First Out (FIFO)
  – Element that has been in queue for the longest gets out first
• Like people standing in a line
Operations

• add / enqueue: add element to back of queue
• peek / element: access first element w/o removing it
• poll / remove / dequeue: remove first element and return it
Implementing Queues

• Linked list
  – Keep a pointer to end of linked list
  – Enqueue: add to end
  – Dequeue: remove from front
  – (or vice versa)
Queue Example

(start with empty queue)

Front / back of queue
Queue Example

(start with empty queue)
enqueue(1)
Queue Example

(start with empty queue)

enqueue(1)
enqueue(2)
Queue Example

(start with empty queue)
enqueue(1)
enqueue(2)
decqueue() // 1

Front of queue  Back of queue

2
Queue Example

(start with empty queue)
enqueue(1)
enqueue(2)
dequeue() // 1
peek() // 2

Front of queue

2

Back of queue
Queue Example

(start with empty queue)

enqueue(1)
enqueue(2)
denqueue() // 1
peek() // 2
denqueue() // 2

Front / back of queue
Tree Traversal

• Fringe: group of nodes to be visited next

• Algorithm:
  – Add root to fringe
  – While fringe is not empty:
    • Remove and process a node from fringe
    • Add its children to the fringe
Tree Traversals: Depth First Search

• Node order:
  – Start at root
  – Go as deep as you can (choosing children arbitrarily)
  – Backtrack to unvisited nodes

• Use stack for fringe
Top of stack
Top of stack

1
2
3
4
5
6
7
Top of stack

7
Tree Traversal

• Fringe: group of nodes to be visited next

• Algorithm:
  – Add root to fringe
  – While fringe is not empty:
    • Remove and process a node from fringe
    • Add its children to the fringe
Tree Traversal: Breadth First Search

- Node order:
  - Start at root
  - Visit all nodes at next depth level
  - Repeat

- Use queue for fringe
Front / back of queue
Binary Trees

• Each node has up to 2 children
• Instead of ArrayList of children, nodes have myLeft and myRight instance variables
Binary Tree Traversals

- Start at root
- A: Process left subtree
- B: Process current node’s element
- C: Process right subtree
- Preorder: BAC
- Inorder: ABC
- Postorder: ACB
Preorder, inorder, or postorder?
Preorder, inorder, or **postorder**?
Preorder, inorder, or postorder?
Preorder, inorder, or postorder?
Preorder, inorder, or postorder?
Preorder, inorder, or postorder?
Binary Tree Array Representation

• Root at index 1
• If node is at index $i$
  – myLeft at index $2i$
  – myRight at index $2i + 1$
• Nothing at index 0
Binary Tree Array Example

```
X  1  2  3  4  5  6  7
```

```
X  1  2  3  4  5  7
```
Types of Binary Trees

• Full: Every node has 2 children (except for leaves)
• Complete: Every level is as filled up as much as possible except possibly the bottom-most level, where all nodes are as left as possible
• Balanced: Height of left and right subtree of every node differ by at most 1
Complete Binary Tree
Balanced Binary Tree
Binary Search Trees

- Invariants:
  - No nodes with duplicate keys
  - At each node, every node in the left subtree has key less than the current node’s key
  - At each node, every node in the right subtree has key greater than the current node’s key
  - Subtrees rooted at each node’s children must also be binary search trees
NOT A BST!
Binary Search Tree

• If reasonably well-balanced, searching for a key in a binary search tree takes $O(\log n)$ for a tree with $n$ nodes
Inorder traversal of this BST?
Trie

• Each node’s key is determined by keys of all nodes in path from root
• Useful as a dictionary for storing words!
Trie Example

c
t
cat
two
can
cap
cat
too
two
See you tonight (maybe)