CS 483 - Data Structures and Algorithm Analysis

Lecture VII: Chapter 6, part 2

R. Paul Wiegand

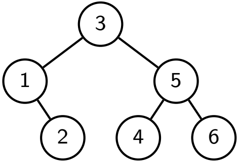
George Mason University, Department of Computer Science

March 22, 2006

Outline

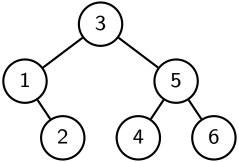
Binary Search Trees

- *binary search tree* — A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value
- Operations: SEARCH, INSERT, & DELETE
- Average case for these: $\Theta(\lg n)$



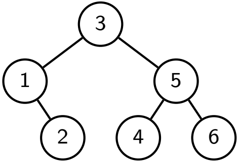
Binary Search Trees

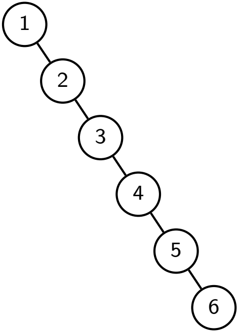
- *binary search tree*— A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value
- Operations: SEARCH, INSERT, & DELETE
- Average case for these: $\Theta(\lg n)$
- Worst case for these: $\Theta(n)$
- This occurs when the tree is *unbalanced* (wide diversity of path lengths from leaf nodes to root)



Binary Search Trees

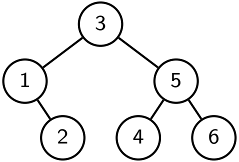
- *binary search tree*— A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value
- Operations: SEARCH, INSERT, & DELETE
- Average case for these: $\Theta(\lg n)$
- Worst case for these: $\Theta(n)$
- This occurs when the tree is *unbalanced* (wide diversity of path lengths from leaf nodes to root)
- In the most severe case, the tree becomes a list whose height is $O(n)$

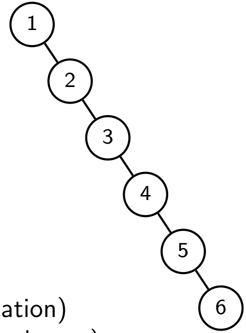




Binary Search Trees

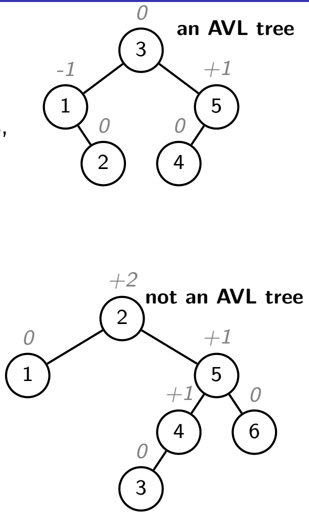
- *binary search tree*— A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value
- Operations: SEARCH, INSERT, & DELETE
- Average case for these: $\Theta(\lg n)$
- Worst case for these: $\Theta(n)$
- This occurs when the tree is *unbalanced* (wide diversity of path lengths from leaf nodes to root)
- In the most severe case, the tree becomes a list whose height is $O(n)$
- Two high-level for avoiding unbalanced trees:
  - Balance an unbalanced tree (instance simplification)
  - Allow more elements in a node (representation change)



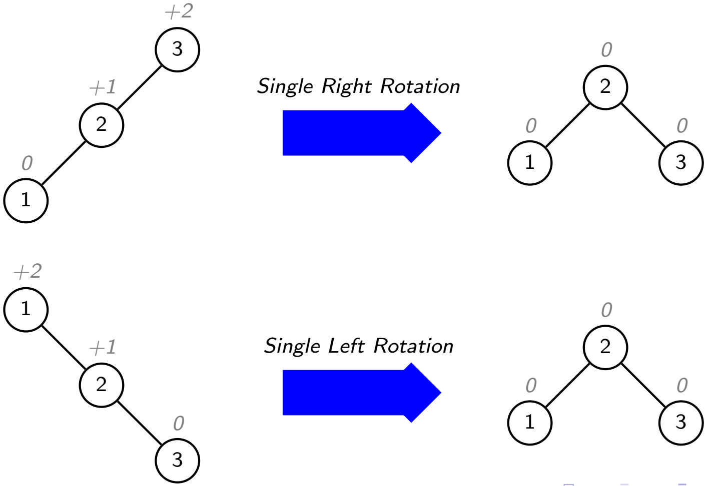


AVL Trees

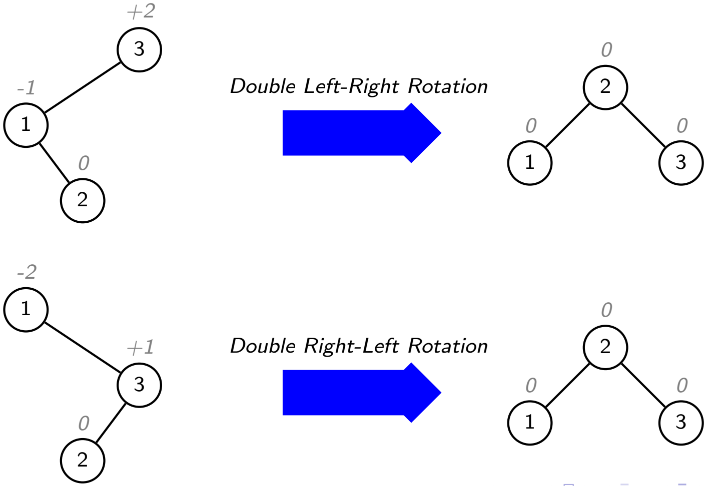
- Methods for transforming unbalanced trees to balanced trees include AVL trees, red-black trees, and splay trees
- *Balance factor*— the difference between the heights of the left and right subtrees
- *AVL tree*— a binary search tree in which the balance factor of every node is $\{+1, 0, -1\}$
- The trick is to *maintain* the AVL property when nodes are inserted or deleted
- To do so, there are four special transformations:
  - Single-right, single-left rotation
  - Double left-right, double right-left rotation



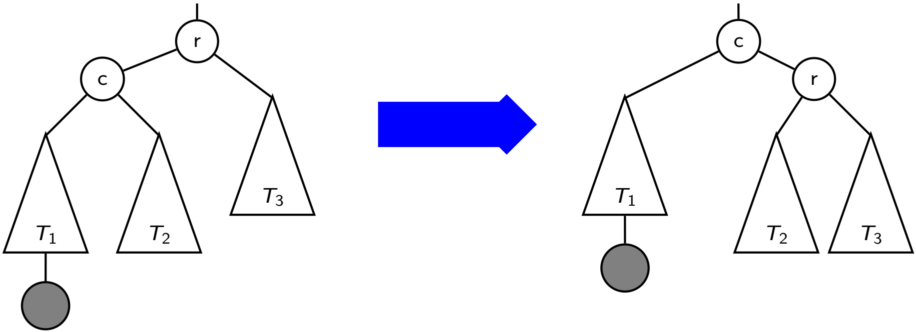
Right & Left Rotations



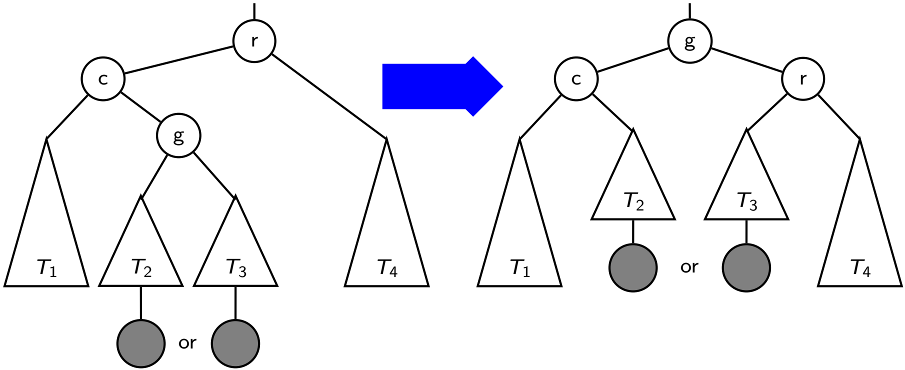
Left-Right & Right-Left Rotations



General Single-Right Rotation



General Double Left-Right Rotation



Analyzing AVL Trees

- Rotations are complicated operations, but still constant time
- Tree traversal efficiency depends on height of the tree
- The Height h of any AVL tree with n nodes can be bound by $\lg n$
- So SEARCH, INSERT, and even DELETE are in $\Theta(\lg n)$.
- Cost: Frequent rotations (high constant values in running-time)

Analyzing AVL Trees

- Rotations are complicated operations, but still constant time
- Tree traversal efficiency depends on height of the tree
- The Height h of any AVL tree with n nodes can be bound by $\lg n$
- So SEARCH, INSERT, and even DELETE are in $\Theta(\lg n)$.
- Cost: Frequent rotations (high constant values in running-time)

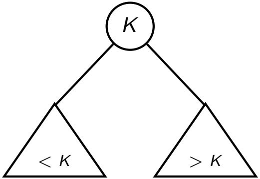
Something to Ponder:

Is it better to accept a linear worst case situation when the average is $\Theta(\lg n)$ (binary search tree), or to slow all operations down by a constant factor to ensure a $\lg n$ bound in all cases (AVL tree)?

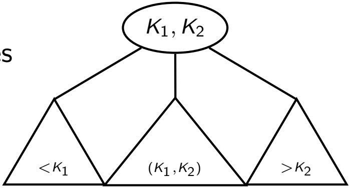
2-3 Trees

One may also change the representation by allowing more nodes (e.g., 2-3 trees, 2-3-4 trees, and B-trees)

2-node — Contains a single key K and (up to) two subtrees. The left subtree contains nodes with key values less than K, the right contain values greater than K



3-node — Contains two keys K_1 and K_2, and (up to) three subtrees. The left subtree contains nodes with key values less than K_1, the right contain values greater than K_2, the middle contain values in (K_1, K_2)



Searching in 2-3 Trees

- For a 2-node: Compare the search key to the key at the node
  - If they are the same, return the node
  - If the search key is less, traverse left
  - If the search key is greater, traverse right

- For a 3-node: Compare the search key to two keys at the node
  - If the search key is equal to either node keys, return the node
  - If the search key is less than the first node key, traverse left
  - If it is between the two keys, traverse middle
  - If it is greater than the second node key, traverse right

Inserting in 2-3 Trees

- If tree is empty, make a 2-node at the root for the inserted key
- Otherwise,
  - Insert at a leaf (i.e., SEARCH)
  - If the leaf is a 2-node, insert the key in that node in the correct order
  - If the leaf is a 3-node, split the node up
    - The smallest key becomes a left 2-node
    - The largest key becomes a right 2-node
    - The middle key is promoted to the parent
    - Note: This promotion can force a split in the node above

Example: Inserting in a 2-3 Tree

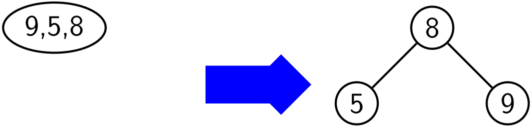
Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:

Example: Inserting in a 2-3 Tree

Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:

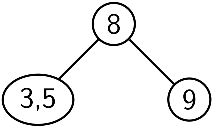
Example: Inserting in a 2-3 Tree

Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:



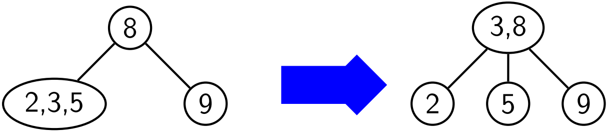
Example: Inserting in a 2-3 Tree

Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:



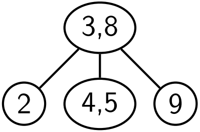
Example: Inserting in a 2-3 Tree

Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:



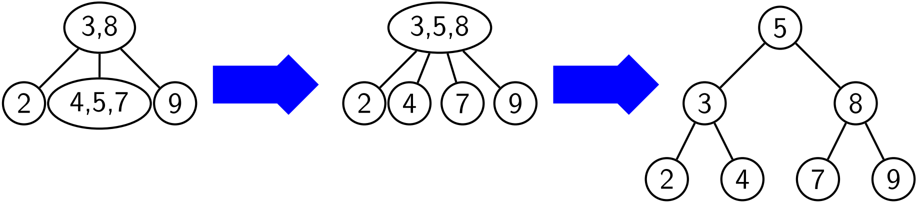
Example: Inserting in a 2-3 Tree

Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:



Example: Inserting in a 2-3 Tree

Inserting: $\langle 9, 5, 8, 3, 2, 4, 7 \rangle$:



Analyzing 2-3 Trees

Consider a 2-3 tree of height h with n nodes in it.

- Upper bound: All nodes are 2-nodes,
  $n \geq 1 + 2 + \ldots + 2^h = 2^{h+1} - 1$
  $\therefore h \leq \lg(n+1) - 1$
- Lower bound: All nodes are 3-nodes,
  $n \leq 2 \cdot 3^0 + 2 \cdot 3^1 + \cdots + 2 \cdot 3^h = 3^{h+1} - 1$
  $\therefore h \geq \log_3(n+1) - 1$
- So the height is bounded by $\Theta(\log n)$
- Basic operations are, as well

Introduction to Heaps

- Heaps are *incompletely* ordered data structures suitable for *priority queues*
  - FIND item with highest priority
  - DELETE item with highest priority
  - ADD NEW ITEM TO THE SET

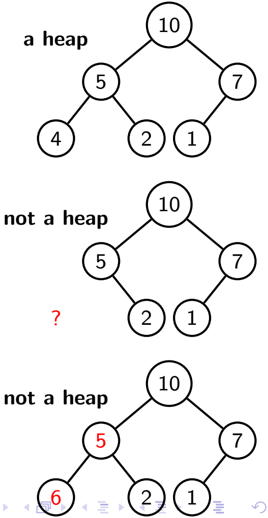
Definition

A *heap* can be defined as a binary tree that meets the following conditions:

1. It is *essentially complete* (all $h-1$ levels are full, level h has only left-most leaves)
2. *Parental dominance*— Key at each node is $\geq$ its children

Introduction to Heaps

- Heaps are *incompletely* ordered data structures suitable for *priority queues*
  - FIND item with highest priority
  - DELETE item with highest priority
  - ADD NEW ITEM TO THE SET



Definition

A *heap* can be defined as a binary tree that meets the following conditions:

1. It is *essentially complete* (all $h-1$ levels are full, level h has only left-most leaves)
2. *Parental dominance*— Key at each node is $\geq$ its children

Fun Facts about Heaps

- The height of an essentially complete binary tree with n nodes is always $\lfloor \lg n \rfloor$
- The root node of a heap always has the largest key value
- Any subtree of a heap is also a heap
- A heap can be implemented as an array
  - Store values top-down, left-to-right
  - Parent nodes in first $\lfloor n/2 \rfloor$ positions, leaf keys in last $\lceil n/2 \rceil$
  - Children of a key in position $i \in [1, \lfloor n/2 \rfloor]$ will be at $2i$ and $2i+1$
  - A parent of a key in position $j \in [\lceil n/2 \rceil, n]$ will be at $\lfloor n/2 \rfloor$
  - Alternate heap definition:

$$H[i] \geq max\{H[2i], H[2i+1]\} \;\; \forall i \in [1, \lfloor n/2 \rfloor]$$

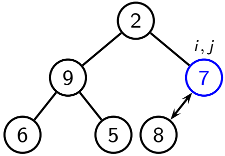
parents **children**

i	0	1	2	3	4	5	6
$H[i]$		10	5	7	4	2	1

Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat



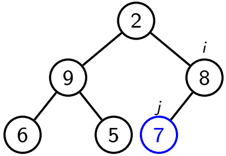
$C_{worst}(n) = 2(n - \log(n+1))$

$\therefore C(n) \in O(n)$

Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat



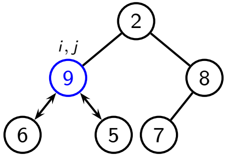
$$C_{worst}(n) = 2(n - \log(n+1))$$

$$\therefore C(n) \in O(n)$$

Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat



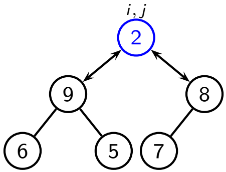
$C_{worst}(n) = 2(n - \log(n+1))$

$\therefore C(n) \in O(n)$

Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat



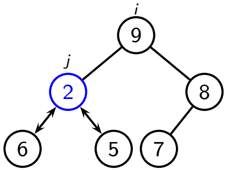
$C_{worst}(n) = 2(n - \log(n+1))$

$\therefore C(n) \in O(n)$

Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat



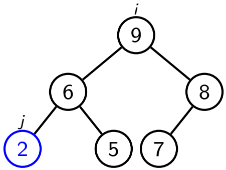
$C_{worst}(n) = 2(n - \log(n+1))$

$\therefore C(n) \in O(n)$

Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat



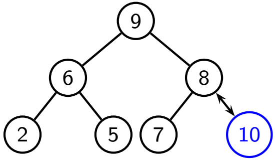
$C_{worst}(n) = 2(n - \log(n+1))$

$\therefore C(n) \in O(n)$

Top-Down Heap Construction

Top-down heap construction maintains heap properties as nodes are inserted.

- Repeatedly insert new nodes at the bottom of the heap
- Each insert:
  - Compare inserted node to parent
  - If parental dominance condition is not met, swap nodes
  - Repeat until condition met or root is reached



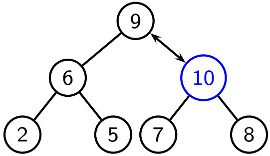
Comparisons needed for inserts are bounded by the heap height:

$C_{insert}(n) = O(\lg n)$
$\therefore C(n) \in O(n \lg n)$

Top-Down Heap Construction

Top-down heap construction maintains heap properties as nodes are inserted.

- Repeatedly insert new nodes at the bottom of the heap
- Each insert:
  - Compare inserted node to parent
  - If parental dominance condition is not met, swap nodes
  - Repeat until condition met or root is reached



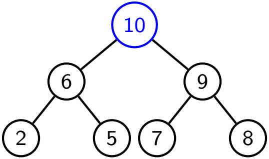
Comparisons needed for inserts are bounded by the heap height:

$C_{insert}(n) = O(\lg n)$
$\therefore C(n) \in O(n \lg n)$

Top-Down Heap Construction

Top-down heap construction maintains heap properties as nodes are inserted.

- Repeatedly insert new nodes at the bottom of the heap
- Each insert:
  - Compare inserted node to parent
  - If parental dominance condition is not met, swap nodes
  - Repeat until condition met or root is reached

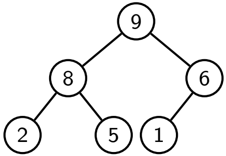


Comparisons needed for inserts are bounded by the heap height:

$C_{insert}(n) = O(\lg n)$
$\therefore C(n) \in O(n \lg n)$

Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the hep size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction

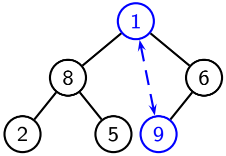


Comparisons needed for delete are bounded by twice the height:

$C_{delete}(n) = O(\lg n)$

Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the hep size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction



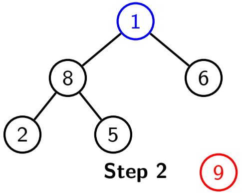
Step 1

Comparisons needed for delete are bounded by twice the height:

$C_{delete}(n) = O(\lg n)$

Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the hep size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction

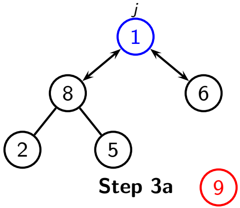


Comparisons needed for delete are bounded by twice the height:

$C_{delete}(n) = O(\lg n)$

Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the hep size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction

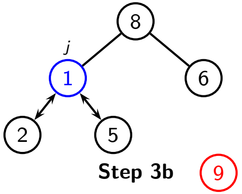


Comparisons needed for delete are bounded by twice the height:

$C_{delete}(n) = O(\lg n)$

Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the hep size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction

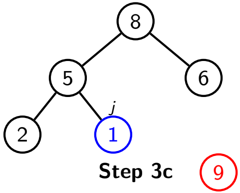


Comparisons needed for delete are bounded by twice the height:

$C_{delete}(n) = O(\lg n)$

Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the hep size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction



Comparisons needed for delete are bounded by twice the height:

$C_{delete}(n) = O(\lg n)$

HeapSort

- Two stage process:
  1. Construct a heap
  2. Apply root-deletion $n-1$ times
- Bottom-up heap construction is $O(n)$
- The deletes are *slightly* more complicated to analyze because the size changes with each deletion:

$$
\begin{aligned}
C(n) &\leq 2\lfloor \lg(n-1) \rfloor + 2\lfloor \lg(n-2) \rfloor + \cdots + 2\lfloor \lg 1 \rfloor \\
&\leq 2\sum_{i=1}^{n-1} \lg i \\
&\leq 2\sum_{i=1}^{n-1} \lg(n-1) = 2(n-1)\lg(n-1) \\
&\leq 2n\lg n \\
C(n) &\in O(n\lg n)
\end{aligned}
$$

Evaluating Polynomials

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Given some polynomial evaluate it at a specified x
- Example: $p(x) = 2x^2 - 3x + 1$
- Brute force: $p(2) = 2*(2*2) - 3*(2) + 2 = 4$

Evaluating Polynomials

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Given some polynomial evaluate it at a specified x
- Example: $p(x) = 2x^2 - 3x + 1$
- Brute force: $p(2) = 2 * (2 * 2) - 3 * (2) + 2 = 4$

3 multiplications

Evaluating Polynomials

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Given some polynomial evaluate it at a specified x
- Example: $p(x) = 2x^2 - 3x + 1$
- Brute force: $p(2) = 2*(2*2) - 3*(2) + 2 = 4$ — 3 multiplications
- In general for brute force:
  - $a_n x^n = a_n * x * x * x \cdots$ requires n multiplications
  - $a_{n-1} x^{n-1}$ requires $n-1$ multiplications
  - ...

Evaluating Polynomials

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Given some polynomial evaluate it at a specified x
- Example: $p(x) = 2x^2 - 3x + 1$
- Brute force: $p(2) = 2*(2*2) - 3*(2) + 2 = 4$ — 3 multiplications
- In general for brute force:
  - $a_n x^n = a_n * x * x * x \cdots$ requires n multiplications
  - $a_{n-1} x^{n-1}$ requires $n-1$ multiplications
  - ...
  - $\sum_{i=0}^{n} i \in O(n^2)$

Evaluating Polynomials

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Given some polynomial evaluate it at a specified x
- Example: $p(x) = 2x^2 - 3x + 1$
- Brute force: $p(2) = 2*(2*2) - 3*(2) + 2 = 4$

3 multiplications

- In general for brute force:
  - $a_n x^n = a_n * x * x * x \cdots$ requires n multiplications
  - $a_{n-1} x^{n-1}$ requires $n-1$ multiplications
  - ...
  - $\sum_{i=0}^{n} i \in O(n^2)$
- Is there a better way?

Horner's Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

$$p(x) = a_n x^n + + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0$$

Horner's Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

$$\begin{aligned} p(x) &= a_n x^n + + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \\ &= (a_n x + + a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \end{aligned}$$

Horner's Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

$$\begin{aligned} p(x) &= a_n x^n + +a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0 \\ &= (a_n x + +a_{n-1})x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0 \\ &= \left((a_n x + +a_{n-1})x + a_{n-2}\right) x^{n-2} + \cdots + a_1 x + a_0 \end{aligned}$$

Horner's Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

$$
\begin{aligned}
p(x) &= a_n x^n + + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \\
&= (a_n x + + a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \\
&= ((a_n x + + a_{n-1}) x + a_{n-2})\, x^{n-2} + \cdots + a_1 x + a_0 \\
&= (\ldots (a_n x + a_{n-1}) x + \ldots)\, x + a_0
\end{aligned}
$$

Horner's Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

$$\begin{aligned} p(x) &= a_n x^n + +a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \\ &= (a_n x + +a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \\ &= ((a_n x + +a_{n-1}) x + a_{n-2}) \, x^{n-2} + \cdots + a_1 x + a_0 \\ &= (\ldots (a_n x + a_{n-1}) x + \ldots) \, x + a_0 \end{aligned}$$

One multiplication (& one addition) per coefficient $\therefore O(n)$

For example: $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$. What is $p(3)$?

$\vec{a}$	2	-1	3	1	-5
x	$P = a_4$	$P = Px + a_3$	$P = Px + a_2$	$P = Px + a_1$	$P = Px + a_0$
$x = 3$	2	$2 \cdot 3 - 1 = 5$	$5 \cdot 3 + 3 = 18$	$18 \cdot 3 + 1 = 55$	$55 \cdot 3 - 5 = 160$

Binary Exponentiation Basics

x^n, where $x = a$

- A degenerate polynomial evaluation problem of interest is a^n

Binary Exponentiation Basics

x^n, where $x = a$

- A degenerate polynomial evaluation problem of interest is a^n
- Suppose we have a representation of n as a binary string of length ℓ:
  $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$ e.g., $n = 13 = 1101_2$

Binary Exponentiation Basics

x^n, where $x = a$

- A degenerate polynomial evaluation problem of interest is a^n
- Suppose we have a representation of n as a binary string of length ℓ:
  $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$ e.g., $n = 13 = 1101_2$
- Can interpret bits as coefficients, write a polynomial where $x = 2$:
  $p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$ e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Binary Exponentiation Basics

x^n, where $x = a$

- A degenerate polynomial evaluation problem of interest is a^n
- Suppose we have a representation of n as a binary string of length ℓ:
  $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$ e.g., $n = 13 = 1101_2$
- Can interpret bits as coefficients, write a polynomial where $x = 2$:
  $p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$ e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- We can now rewrite a^n:
  $a^{p(x)} = a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0}$ e.g., $a^{1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0}$

Binary Exponentiation Basics

x^n, where $x = a$

- A degenerate polynomial evaluation problem of interest is a^n
- Suppose we have a representation of n as a binary string of length ℓ:
  $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$ e.g., $n = 13 = 1101_2$
- Can interpret bits as coefficients, write a polynomial where $x = 2$:
  $p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$ e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- We can now rewrite a^n:
  $a^{p(x)} = a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0}$ e.g., $a^{1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0}$
- So we can accumulate the product in the exponent by Horner's rule

Binary Exponentiation Basics

x^n, where $x = a$

- A degenerate polynomial evaluation problem of interest is a^n
- Suppose we have a representation of n as a binary string of length ℓ:
  $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$ e.g., $n = 13 = 1101_2$
- Can interpret bits as coefficients, write a polynomial where $x = 2$:
  $p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$ e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- We can now rewrite a^n:
  $a^{p(x)} = a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0}$ e.g., $a^{1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0}$
- So we can accumulate the product in the exponent by Horner's rule
- Writing p as the current product, we recognize that:
  $$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0 \\ (a^p)^2 \cdot a & \text{if } b_i = 1 \end{cases}$$

Left-to-Right Binary Exponentiation

LeftToRightExp($a, b(n)$)

$p \longleftarrow a$
for $i \leftarrow \ell$ downto 0 do
  $p \longleftarrow p \cdot p$
  if $b_i = 1$ then $p \leftarrow p \cdot a$
return p

- Number of multiplications bounded by the number of 1-bits
- This is bounded by ℓ, the length of b
- $\ell - 1 = \lfloor \lg n \rfloor$
- $\therefore M(n) = O(\lg n)$
- But we must have binary string to begin with!

For example: a^{13} where $n = 13 = 1101_2$:

binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$
example	3	$(9) \cdot 3 = 27$	$(27)^2 = 729$	$(729)^2 \cdot 3 = 1,594,323$

Right-to-Left Binary Exponentiation

- Can re-express a^n:
  $a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0} =$
  $a^{b_\ell 2^\ell} \cdot \ldots \cdot a^{b_i 2^i} \cdot \ldots \cdot a^{b_0}$
- We recognize that:
  $a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$
- This is also $O(\lg n)$
- Also relies on having an available binary string

Right-to-Left Binary Exponentiation

- Can re-express a^n:
  $a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0} =$
  $a^{b_\ell 2^\ell} \cdot \cdots \cdot a^{b_i 2^i} \cdot \cdots \cdot a^{b_0}$
- We recognize that:
  $a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$
- This is also $O(\lg n)$
- Also relies on having an available binary string

RightToLeftExp($a, b(n)$)

```
t ⟵ a
if b_0 = 1 then p ⟵ a
else p ⟵ 1
for i ← 1 to ℓ do
  t ⟵ t · t
  if b_i = 1 then p ← p · t
return p
```

Right-to-Left Binary Exponentiation

- Can re-express a^n:
  $a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0} =$
  $a^{b_\ell 2^\ell} \cdot \ldots \cdot a^{b_i 2^i} \cdot \ldots \cdot a^{b_0}$
- We recognize that:
  $a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$
- This is also $O(\lg n)$
- Also relies on having an available binary string

RightToLeftExp($a, b(n)$)

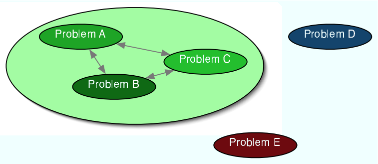
```
t ⟵ a
if b_0 = 1 then p ⟵ a
else p ⟵ 1
for i ← 1 to ℓ do
  t ⟵ t · t
  if b_i = 1 then p ← p · t
return p
```

For example: a^{13} where $n = 13 = 1101_2$:

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms of a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator
$3^5 \cdot 3^8 = 1,594,323$	$3 \cdot 3^4 = 243$		3	example

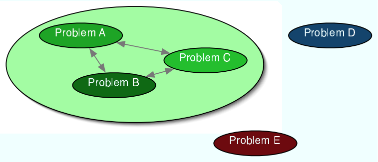
"Reducing" Problems

- Not called "reducing" because the problem gets smaller or even (necessarily) easier
- Comp Sci's transform one problem into another as a means of classifying problems
- Properly classified, the *space* of unique problems is reduced

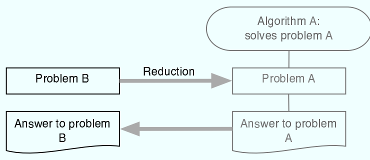


"Reducing" Problems

- Not called "reducing" because the problem gets smaller or even (necessarily) easier
- Comp Sci's transform one problem into another as a means of classifying problems
- Properly classified, the *space* of unique problems is reduced



- Also reduce problems as a means of solving problems using known & proven methods
- Or when another view gives us some additional insight about the original problem



Least Common Multiple

The *least common multiple* of two positive integers m and n, $\text{lcm}(m, n)$, is the smallest integer that is divisible by both m and n.

$$24 = 2 \cdot 2 \cdot 3 \cdot 2$$
$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$\text{lcm}(24, 60) = (2 \cdot 2 \cdot 3) \cdot (2 \cdot 5)$$

- Middle school method:
  - Compute the prime factors of m and n
  - Multiply common factors by the uncommon factors

Least Common Multiple

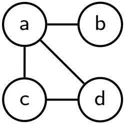
The *least common multiple* of two positive integers m and n, $\text{lcm}(m,n)$, is the smallest integer that is divisible by both m and n.

$$24 = 2 \cdot 2 \cdot 3 \cdot 2$$
$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$\text{lcm}(24,60) = (2 \cdot 2 \cdot 3) \cdot (2 \cdot 5)$$

- Middle school method:
  - Compute the prime factors of m and n
  - Multiply common factors by the uncommon factors
- Alternatively:
  - Note: The product of $\text{lcm}(m,n)$ and $\text{gcd}(m,n)$ includes every factor exactly once
  - In other words: $\text{lcm}(m,n) \cdot \text{gcd}(m,n) = m \cdot n$
  - $\therefore \text{lcm}(m,n) = \frac{m \cdot n}{\text{gcd}(m,n)}$
  - So, if we can solve gcd, we can solve lcm
  - gcd can be computed efficiently via Euclid's algorithm

Counting Paths in a Graph

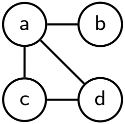
- How many paths of length k are there between any pair of nodes in a graph?
- We could perform a graph search and count the paths ...
- But there's a cool little trick:
    - Consider the adjacency matrix A



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Counting Paths in a Graph

- How many paths of length k are there between any pair of nodes in a graph?
- We could perform a graph search and count the paths ...
- But there's a cool little trick:
  - Consider the adjacency matrix A
  - Recall: $A^2 = A \cdot A$ and $A_{ij} = \{0, 1\}\ \forall i, j$
  - So by matrix multiplication, A^2_{ij} is the sum of all situations in which the i is connected to some other node *and* that node is connected to j

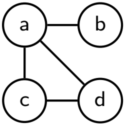


$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

Counting Paths in a Graph

- How many paths of length k are there between any pair of nodes in a graph?
- We could perform a graph search and count the paths ...
- But there's a cool little trick:
  - Consider the adjacency matrix A
  - Recall: $A^2 = A \cdot A$ and $A_{ij} = \{0, 1\}\ \forall i, j$
  - So by matrix multiplication, A^2_{ij} is the sum of all situations in which the i is connected to some other node *and* that node is connected to j
  - $A^k = A \cdot A \cdot A \cdots$
  - The value at A^k_{ij} will be the number of paths of length k that connect i and j



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

Optimization

- One optimization problem is *maximization* — $\text{argmax}\{f(x)\}$, find the argument value for x that gives us $\max\{f(x)\}$
- We may also be asked to *minimize* a function
- It turns out that this is the *same problem*: $\max\{f(x)\} = -\max\{-f(x)\}$
- This works for virtually any domain — so if you can solve maximization, you can solve minimization
- Moreover, the standard calculus method is a type of reduction:
    - Calculate the derivative, $f'(x) = \frac{d}{dx} f(x)$
    - Solve for $f'(0)$
    - Assuming the derivatives can be calculated, this reduces to the problem of finding critical points

Linear Programming

- Linear programming problems involve optimizing a linear function subject to linear constraints
- There exists a general form for many LP problems:

$$\begin{array}{rl} \text{maximize} & c_1x_1 + \cdots + c_nx_n \\ \text{subject to} & a_{i1}x_1 + \cdots + a_{in}x_n\{\leq, =, \geq\}b_i \;\; \forall i \in [1, m] \\ & x_1 \geq 0, \ldots, x_n \geq 0 \end{array}$$

- Many (*many*) problems in computer science can be reduced to such problems (e.g., the *fractional knap-sack problem*)
- There are a variety of well-known methods for solving them:
  - The *simplex method*, which has an exponential worst-case bound, but whose average case is typically quite good
  - Karmarkar's algorithm, which guarantees a polynomial worst-case bound and has done well empirical
- A much harder, related class of problems are *integer linear programming*, which are known to be NP-hard in general (e.g., the *0-1 knap-sack problem*)

Book Topics Skipped in Lecture

- In section 6.6:
  - *Reduction to Graph Problems* (pp. 239–240)

Assignments

- This week's assignments:
  - Section 6.3: Problems 1, 4, & 7
  - Section 6.4: Problems 1 & 6
  - Section 6.5: Problems 4, 6, 7 & 8
  - Section 6.6: Problems 1, 8, & 9