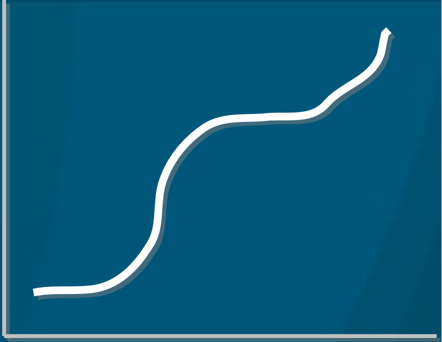
Ch4: Proofs & Recurrences
Ch6: Heap-sort

24 Sep 2013
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

- Review of discrete math:
  - Monotonicity, limits, iterated funs, Fibonacci
  - Mathematical proofs, asymptotic behaviour
- ch4: Solving recurrences
  - Proof by induction ("substitution")
  - Proof by "master method"
- ch6: Sorting: Heap-sort
  - Binary max-heaps
  - Application: Heap-sort

Discrete math review



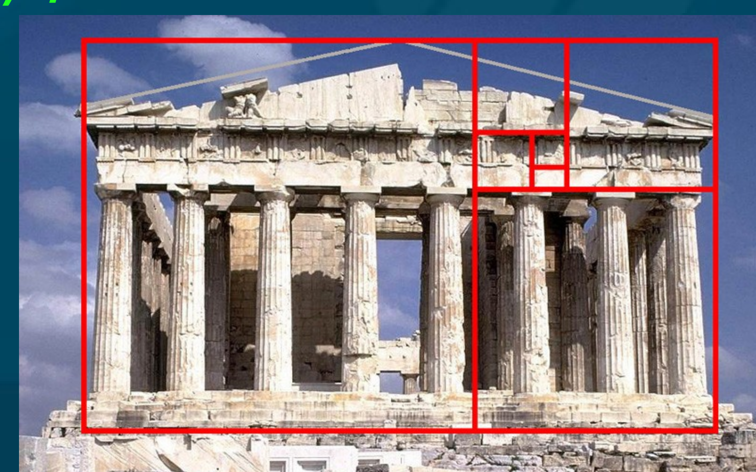
- f(x) is monotone increasing ("non-decreasing") iff $x < y \Rightarrow f(x) \leq f(y)$
- f(x) is strictly increasing iff $x < y \Rightarrow f(x) < f(y)$
- a mod n (in programming: "a % n") is the remainder of a when divided by n
  - 17 mod 5 = 2
- $\lim_{x \to a} f(x) = b$ ("limit as x goes to a of f(x) is b") means $\forall\, \varepsilon > 0, \exists\, \delta > 0: (|x - a| < \delta) \Rightarrow (|f(x) - b| < \varepsilon)$
- $\lim_{n \to \infty} f(n) = b$ ("limit as n goes to ∞ of f(n) is b") means $\forall\, \varepsilon > 0, \exists\, n_0: (n > n_0) \Rightarrow (|f(n) - b| < \varepsilon)$

Math review: iterated functions

- Iterated functions (e.g., recursion):
  - $f^{(i)}(x)$: the function f applied i times to x
    - f(f(f(… f(x) …)))
    - Not the same as $f^i(x) = (f(x))^i$
    - e.g., $\log^{(2)}(1000) = \log(\log(1000) = \log(3) \approx 0.477$
      - but $\log^2(1000) = (\log(1000))^2 = 3^2 = 9$
    - $f^{(0)}(x)$ is defined to be just x (apply f zero times)
- Iterated log: $\lg^*(n) = \min(\ i \geq 0 : \lg^{(i)}(n) \leq 1\ )$
  - "number of times lg needs to be applied to n until the result is ≤ 1"
    - $\lg^*(16) = 3$: $\lg(\lg(\lg(16))) = \lg(\lg(4)) = \lg(2) = 1$

Fibonacci and golden ratio

- The n^{th} Fibonacci number is $F_n = F_{n-1} + F_{n-2}$
  - Start with $F_0 = 0$, $F_1 = 1$
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
      - (also see Lucas numbers: $F_0 = 2$)
- Golden ratio φ (and conjugate $\tilde{\varphi}$) satisfy $x^2 = x + 1$
    - $\varphi = (1 \pm \sqrt{5})/2 \approx 1.61803\ldots$ and $-0.61803\ldots$
- #3.2-7 proves that $F_n = (\varphi^n - \tilde{\varphi}^n) / \sqrt{5}$
    - The second part $|\tilde{\varphi}^n| / \sqrt{5} < ½$, so $F_n = \lfloor \varphi^n/\sqrt{5} + ½ \rfloor$
      - i.e., $F_n = \text{round}(\varphi^n/\sqrt{5})$
      - grows exponentially!



Proving asymptotic behaviour

- e.g., p.52 #3.1-2: show that for all constants a, b, with $b>0$: $(n + a)^b = \Theta(n^b)$
  - i.e., find n_0, c_1, c_2: $\forall\, n>n_0$, $c_1 n^b \le (n + a)^b \le c_2 n^b$
  - Find lower and upper bounds on $(n + a)^b$
- We observe that $n+a \ge n/2$ if $n > 2|a|$, and that $n+a \le 2n$ if $n > |a|$
  - so $n/2 \le n+a \le 2n$, as long as $n > 2|a|$
- Then by the monotonicity of x^b ($x>1$, $b>0$),
  - $(n/2)^b \le (n + a)^b \le (2n)^b$, when $n > 2|a|$
- So we pick $n_0 = 2|a|$, $c_1 = 2^{-b}$, and $c_2 = 2^b$.

Proving asymptotic behaviour

- e.g., p.62 #3-3: $(\lg n)! = \omega(n^3)$
  - Approach: take lg of both sides
  - LHS: use Stirling: $n! = \sqrt{(2\pi n)}\,(n/e)^n\,(1 + \Theta(1/n))$
    - $\Rightarrow \lg(n!) = \Theta(n \lg n)$ *(p.58, Eq 3.19)*
    - $\Rightarrow \lg(\,(\lg n)!\,) = \Theta(\,(\lg n) \lg(\lg n)\,)$
      - Substitute $n \rightarrow \lg n$ and use monotonicity of lg
  - RHS: $\lg(n^3) = 3 (\lg n)$
    - $\lg(\lg n) = \omega(3)$, so now put it together:
  - $\lg(\,(\lg n)!\,) = \Theta(\,(\lg n) \lg(\lg n)\,)$
    $= \omega(3 \lg n)$
    $= \omega(\lg(\,n^3\,))$
  - Hence, by monotonicity of lg, $(\lg n)! = \omega(n^3)$

Outline for today

- Review of discrete math:
  - Monotonicity, limits, iterated funs, Fibonacci
  - Mathematical proofs, asymptotic behaviour
- ch4: Solving recurrences
  - Proof by induction (“substitution”)
  - Proof by “master method”
- ch6: Sorting: Heap-sort
  - Binary max-heaps
  - Application: Heap-sort
  - Application: Priority Queue
- ch7: Quick-sort

Mathematical induction

- Deduction: general principles $\Longrightarrow$ specific case
- Induction: representative case $\Longrightarrow$ general rule
- Needs at least two axioms (givens):
  - Base case: starting point, e.g., rule at n=1
  - Inductive step: if the rule holds at some n, then it also holds at n+1
- From these two axioms, we prove that the given rule holds for all (positive) n

Proof by induction: example

- Last time, we mentioned Gauss' formula for
  - $1 + 2 + \ldots + (n-1) + n = (n)(n+1)/2$
- Now we prove it by induction:
- Proof of base case (n=1): $1 = (1)(1+1)/2$
- Proof of inductive step:
  - Assume: $1 + \ldots + n = (n)(n+1)/2$
  - Want to prove: $1 + \ldots + (n+1) = (n+1)(n+2)/2$
  - i.e., prove: $(n)(n+1)/2 + (n+1) = (n+1)(n+2)/2$
    - $(n+1)(n+2)/2 = (n^2+3n+2)/2$
      $= ((n^2+n) + (2n+2))/2$
      $= (n^2+n)/2 + (2n+2)/2$
      $= n(n+1)/2 + (n+1)$

Induction for recurrences

- Proof by induction also can apply to recurrences:
- e.g., complexity of merge sort:
  - $T(1) = \theta(1)$, and
  - $T(n) = 2T(n/2) + \theta(n)$
- If we have a "guess" about the solution to $T(n)$, we can prove by induction if that guess is correct:
- Guess: $T(n) = \theta(n \lg(n))$
- Proof:
  - Base case: $T(1) = \theta(1 \lg(1)) = \theta(1)$ (i.e., constant time)
  - Inductive step: (next slide)

Inductive proof for merge sort:

- Assume: $T(m) = \theta(m \lg(m))$, for $m = n-1$
  - In fact, can assume this holds for all $m < n$
- Want to prove: $T(n) = \theta(n \lg(n))$
  - i.e., for big n, there exist c_1, c_2 such that $c_1(n \lg(n)) \le T(n) \le c_2(n \lg(n))$
- $T(n) = 2T(n/2) + \theta(n)$ (from the recurrence)
  - $\Rightarrow \exists\, c_1, c_2$: $2T(n/2) + c_1(n) \le T(n) \le 2T(n/2) + c_2(n)$
- but $T(n/2) = \theta(\ (n/2) \lg(n/2)\ )$, so
  - $\Rightarrow \exists\, c_3, c_4$: $c_3(n/2 \lg(n/2)) \le T(n/2) \le c_4(n/2 \lg(n/2))$
  - $\Rightarrow (c_3/2)(n \lg(n) - n \lg 2) \le T(n/2) \le c_4(\ldots)$
  - $\Rightarrow (c_3/2)(n \lg(n)) - (c_1 \lg 2 / 2)n \le T(n/2) \le c_4(\ldots)$

Inductive proof, continued

- Combining the two, ∃ c_1, c_2, c_3, c_4 such that:
  - $2T(n/2) + c_1(n) \le T(n) \le 2T(n/2) + c_2(n)$
  - $\Rightarrow 2(c_3/2)(n \lg(n)) - 2(c_1 \lg 2 / 2)n + c_1(n) \le T(n) \le \ldots$
  - $\Rightarrow c_3(n \lg(n)) - (c_1 \lg 2 + c_1)n \le T(n) \le \ldots$
  - $\Rightarrow c_3(n \lg(n)) - (2c_1)n \le T(n) \le c_4(n \lg(n)) - (2c_2)n$
  - $\Rightarrow c_3(n \lg(n)) \le T(n) \le c_5(n \lg(n))$
  - LHS of last step: just need $c_1>0$
  - RHS of last step: we can't choose c_2, c_4, but we can find an n_0 such that for all $n>n_0$, the $c_4(n \lg(n))$ term overwhelms the $(2c_2)n$ term
- This proves that $T(n) = \theta(n \lg(n))$

Master method for recurrences

- If the recurrence has this specific form:
  - $T(n) = a\,T(n/b) + f(n)$
    - e.g., merge sort: $a = 2$, $b = 2$, $f(n) = \theta(n)$
- Then compare $f(n)$ with $n^{\log\_b(a)}$:
  - If $f(n) = \theta(n^{\log\_b(a)})$:
    - Leaves/roots balanced: $T(n) = \theta(n^{\log\_b(a)} \lg(n))$
  - Else if $f(n) = O(n^{\log\_b(a)-\varepsilon})$ for some $\varepsilon>0$,
    - Leaves dominate the work: $T(n) = \theta(n^{\log\_b(a)})$
  - Else if $f(n) = \Omega(n^{\log\_b(a)+\varepsilon})$ for some $\varepsilon>0$ and $a\,f(n/b) \le c\,f(n)$ for some $c<1$ and big n,
    - Roots dominate the work: $T(n) = \theta(f(n))$
    - Regularity condition is fine for, e.g., $f(n) = n^k$

Master method: examples

- Merge sort: $T(n) = 2T(n/2) + \theta(n)$
  - $a=2$, $b=2$, $f(n) = \theta(n)$
  - $f(n) = \theta(n) = \theta(n^{\log_2(2)})$
    - so leaves and roots contribute work equally
  - $\Rightarrow T(n) = \theta(n^{\log_2(2)} \lg(n)) = \theta(n \lg(n))$
- Strassen matrix multiply: $T(n) = 7T(n/2) + \theta(n^2)$
  - $a=7$, $b=2$, $f(n) = \theta(n^2)$
  - $f(n) = \theta(n^2) = O(n^{\log_2(7)-\varepsilon})$
    - $\log_2 7 \approx 2.8$, so pick an ε between 0 and 0.8
    - Leaves dominate the work
  - $\Rightarrow T(n) = \theta(n^{\log_2(7)}) \approx \theta(n^{2.8})$

Gaps in master thm coverage

- Not all recurrences $aT(n/b) + f(n)$ work in master!
  - e.g., $T(n) = 2T(n/2) + n \lg(n)$
    - $n \lg(n) \neq \theta(n^{\log\_2(2)}) = \theta(n)$
    - $n \lg(n) \neq O(n^{1-\varepsilon})$, for any $\varepsilon>0$
    - $n \lg(n) \neq \Omega(n^{1+\varepsilon})$, for any $\varepsilon>0$
      (because $\lg(n) \neq \Omega(n^{\varepsilon})$ for any $\varepsilon>0$)
- Polylog extension to master theorem:
  - If $f(n) = \theta(n^{\log\_b(a)} \lg^k(n))$
    - where $\lg^k(n) = (\lg(n))^k$
    - Then $T(n) = \theta(n^{\log\_b(a)} \lg^{k+1}(n))$
  - (old case was with $k=0$)
- Above example: $T(n) = \theta(n \lg^2(n))$

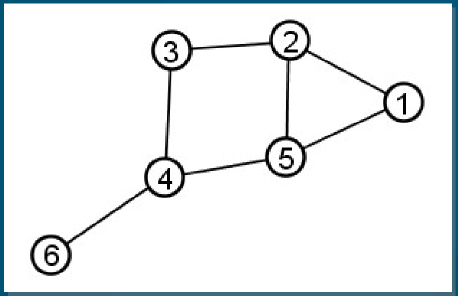
Outline for today

- Review of discrete math:
  - Monotonicity, limits, iterated funs, Fibonacci
  - Mathematical proofs, asymptotic behaviour
- ch4: Solving recurrences
  - Proof by induction (“substitution”)
  - Proof by “master method”
- ch6: Sorting: Heap-sort
  - Binary max-heaps
  - Application: Heap-sort
  - Application: Priority Queue
- ch7: Quick-sort

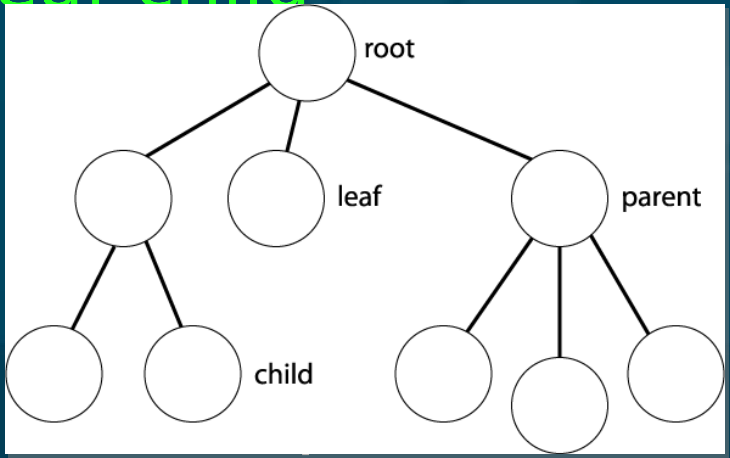
Summary of sorting algorithms

- Comparison sorts (ch2, 6, 7)
  - Insertion sort: $\Theta(n^2)$, easy to program, slow
  - Merge sort: $\Theta(n \lg(n))$, out-of-place sorting, slow due to lots of copying / memory operations
  - Heap sort: $\Theta(n \lg(n))$, in-place, uses max-heap
  - Quick sort: $\Theta(n^2)$ worst-case, $\Theta(n \lg(n))$ average, in-place, fast (small) constant factors
- Linear-time non-comparison sorts (ch8):
  - Counting sort: k distinct values: $\Theta(k)$
  - Radix sort: d digits w/k values: $\Theta(d(n+k))$
  - Bucket sort: for uniform distrib. of values: $\Theta(n)$

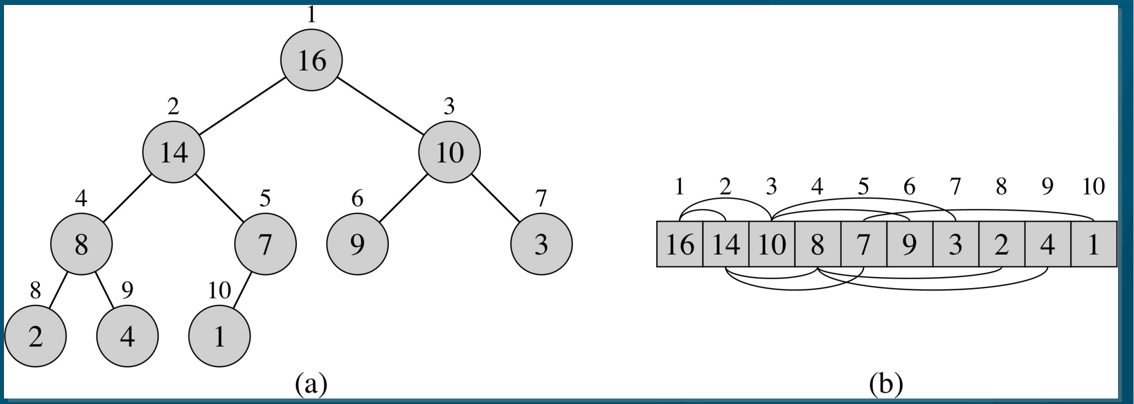
Binary trees



- Graph: collection of nodes and edges
  - Edges may be directed or undirected
- Tree: directed acyclic graph (DAG)
  - Choose a node as root
  - Parent: immediate neighbour toward root
  - Leaf: node with no children
  - Degree: maximum number of children
  - Node height: max # edges to leaf child
  - Node depth: # edges to root
  - Level: all nodes of same depth
- Binary tree: tree with degree=2



Binary heaps



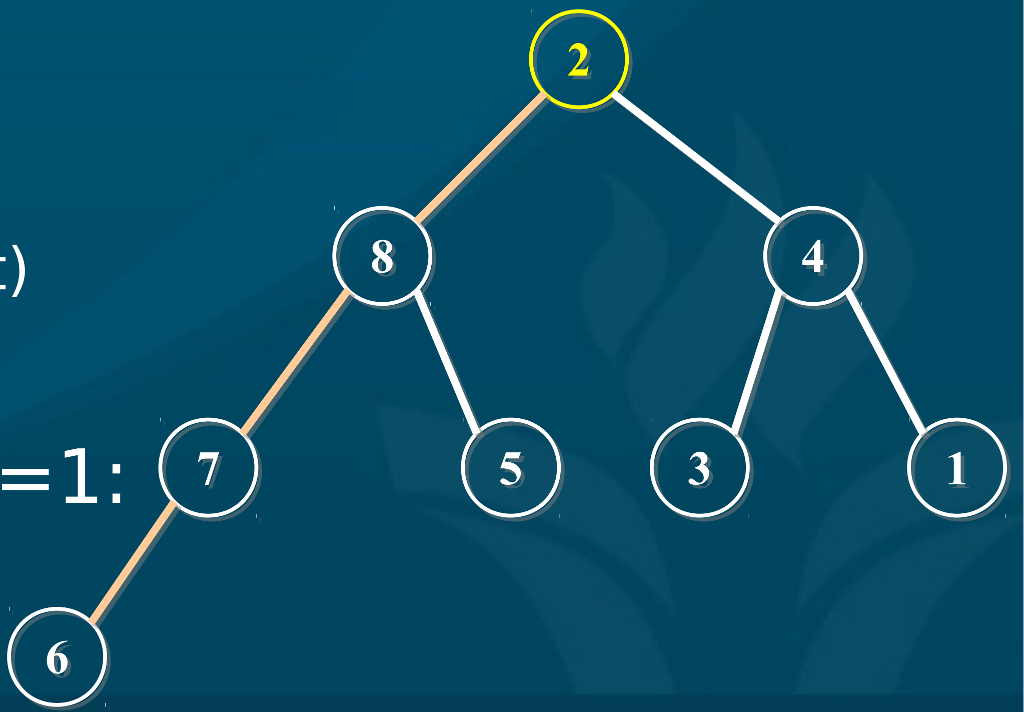
- Array storage for certain binary trees
  - Children of node i are at 2i and 2i+1
  - Must fill tree left-to-right, one level at a time
- Max-heap: value of a node is ≤ value of its parent
  - Min-heap: ≥
- max_heapify() (*O(lg n)*): reposition a given node i so it satisfies the max-heap property
- build_max_heap() (*O(n)*): construct a max-heap from an unordered array
- heapsort() (*O(n lg n)*): sort array in-place

max_heapify(): for single node

- max_heapify(A, i):
  - Precondition: left and right sub-trees of i satisfy the max-heap property
  - Postcondition: subtree at i satisfies max-heap
- Algorithm:
  - Amongst {i, left(i), right(i)}, find the largest
  - If i is not the largest, then
    - Swap i with the largest, and
    - Recurse/iterate on that subtree

max_heapify(): pseudocode

- max_heapify(A, i):
  - largest = i
  - if 2i ≤ length(A) and A[2i] > A[largest]:
    - largest = 2i
  - else if 2i+1 ≤ length(A) and A[2i+1] > A[largest]:
    - largest = 2i+1
  - if largest ≠ i:
    - swap(A[i], A[largest])
    - max_heapify(A, largest)



- A=[2, 8, 4, 7, 5, 3, 1, 6], i=1:
- Running time?

Building a max-heap

- build_max_heap(A):
  - Input: array of items in any order
  - Output: array has max-heap property
- Algorithm:
  - Leave last half of array as all leaves
  - Apply max_heapify() to each item in first half:
    - for i = floor(length(A)/2) .. 1:
      - max_heapify(A, i)
    - Descending order: each time max_heapify() is called on a node, its subtrees are already max-heaps
- Exercise: try it on [5, 2, 7, 4, 8, 1]

build_max_heap(): complexity

- Group iterations of for loop by height h of node:
  - Each call to max_heapify(i) takes $O(h)$
  - # of nodes with height h is ≤ ceil($n / 2^{h+1}$)
    - Attains that bound when tree is full
- So algorithmic complexity is $\Sigma((n / 2^{h+1}) O(h))$
    - Sum for h = 0 .. lg(n) is ≤ sum for h = 0 .. ∞
  - $= n\, O(\Sigma (1/2)^{h+1})$, where sum is for h = 0 .. ∞
  - $= O(n)$
- We can build a max heap in linear time!
  - But it's not quite a sorting algorithm….

Outline for today

- Review of discrete math:
  - Monotonicity, limits, iterated funs, Fibonacci
  - Mathematical proofs, asymptotic behaviour
- ch4: Solving recurrences
  - Proof by induction (“substitution”)
  - Proof by “master method”
- ch6: Sorting: Heap-sort
  - Binary max-heaps
  - Application: Heap-sort
  - Application: Priority Queue
- ch7: Quick-sort

Using max-heaps for sorting

- Algorithm:
  - Make array a max-heap
  - Repeat, working backwards from end of array:
    - Swap root of max-heap with last leaf of heap
    - Shrink heap by 1 and apply max_heapify()
- At each iteration of the loop:
  - First portion of array is a max-heap
  - Last portion is a sorted array (largest items)
- Complexity: $\Theta(n)$ calls to max_heapify() ($\Theta(\lg n)$)
  - $\Rightarrow \Theta(n \lg(n))$
- Exercise: try it on [5, 2, 7, 4, 8, 1]