Ch6-7: Heapsort & Quicksort

25 Sep 2012
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

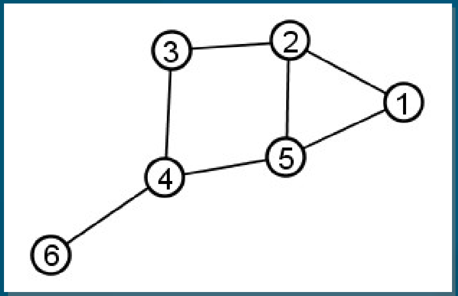
Summary of sorting algorithms

- 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: $\Theta(n)$

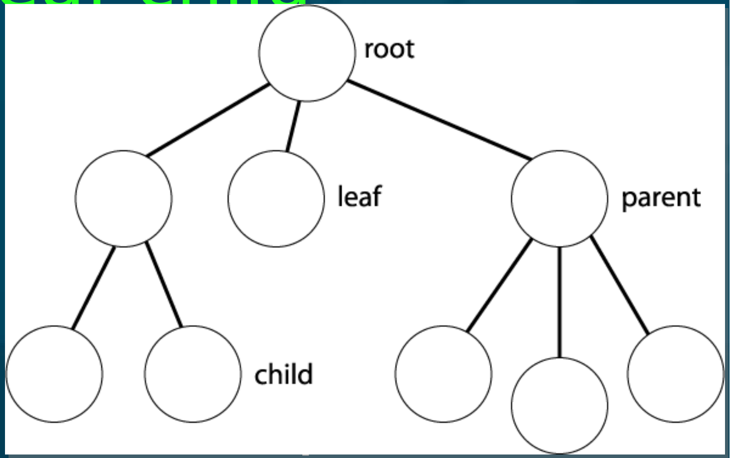
Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

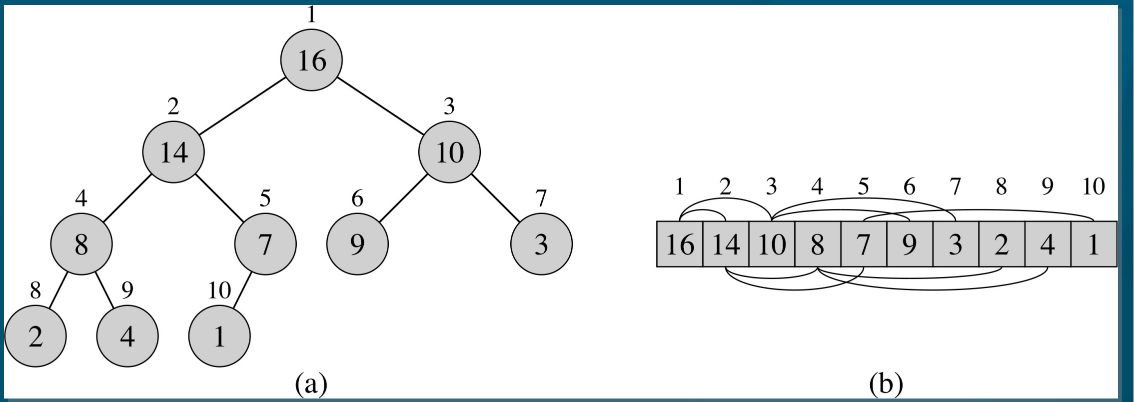
Binary trees



- 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



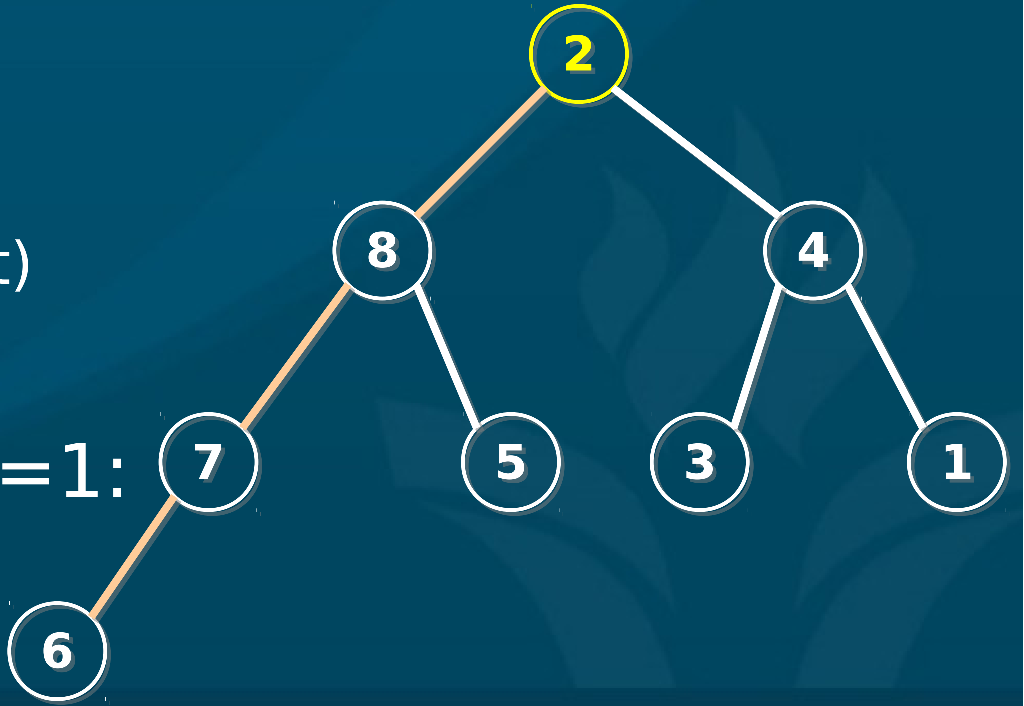
- Array storage for certain binary trees
  - Children of node i are at 2i and 2i+1
  - Must fill tree left-to-right, one level at a time
- Max-heap: value of a node is ≤ value of its parent
  - Min-heap: ≥
- max_heapify() (*O(lg n)*): reposition a given node i so it satisfies the max-heap property
-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

Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

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....

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]

Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

Binary heap for priority queue

- Binary heaps can implement a priority queue:
  - Set of items with attached priorities
- Interface (set of operations):
  - insert(A, item, pri): add item to the queue A
  - find_max(A): return item with highest priority
  - pop_max(A): same but also delete item
  - set_pri(A, item, pri): set new priority for item (must be higher than old priority)
- Setup queue by building a max-heap
  - find_max() is easy: return A[1]
  - pop_max() also easy: remove A[1] and heapify

Inserting into priority queue

- set_pri(A, i, pri): starting from i, "bubble" item up until we find the right place:
  - A[i] = pri
  - while i>1 and A[i/2] < A[i]:
    - swap(A[i/2], A[i])
    - i = i/2
  - Complexity: # iterations = Θ(lg n)
- insert(A, pri): make a new node and set its priority
  - A.length++
  - set_pri(A, A.length, pri)
  - Typically, use pre-allocated fixed-length array, and use separate variable to track size of queue
  - Complexity: same as set_pri(): Θ(lg n)

Priority queue: summary

- Build priority queue using a max-heap: $\Theta(n)$
- Get highest priority item: $\Theta(1)$
- Get and delete highest priority item: $\Theta(\lg n)$
- Set new priority for an item: $\Theta(\lg n)$
- Insert new item into queue: $\Theta(\lg n)$

Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

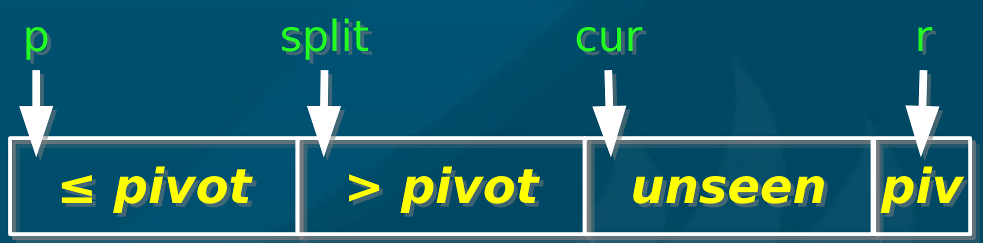
Quicksort

- Divide: partition array A[p .. r] such that:
  - max(A[p .. q-1]) ≤ A[q] ≤ min(A[q+1 .. r])
- Conquer: recurse on each part:
  - quicksort(A, p, q-1) and quicksort(A, q+1, r)
- No combine/merge step needed

- In-place sort
- Worst-case turns out to still be $\Theta(n^2)$, but average-case is $\Theta(n \lg(n))$, with small constants
- In practise, quicksort is one of the best algorithms when input values can be arbitrary

Quicksort: partition

- How to do the partitioning?
  - Pick last item as the pivot
  - Walk through array, partitioning array into items ≤ pivot and items > pivot
  - Lastly, swap pivot into place
    - partition(A, p, r):
      - pivot = A[r]
      - split = p
      - for cur = p .. r-1:
        - if A[cur] ≤ pivot:
          - swap(A[split], A[cur])
          - split++
      - swap(A[split], A[pivot])
      - return split



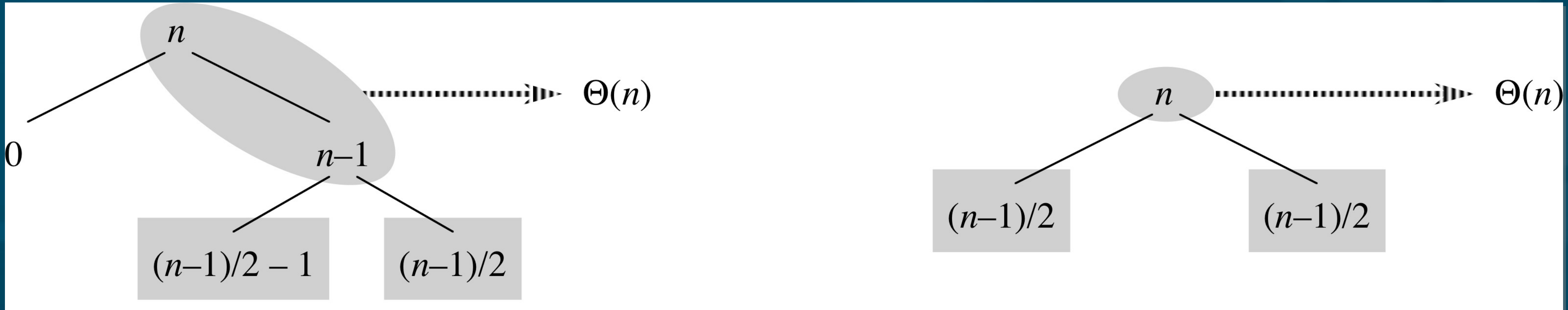
Complexity?

Quicksort: complexity

- Worst-case if every partition is the most uneven:
    - pivot (last item) is either largest or smallest item
    - $T(n) = T(n-1) + T(0) + \Theta(n)$
    - $\Rightarrow T(n) = \Theta(n^2)$
  - Example inputs that give worst case?
- Best-case if every partition is exactly in half:
    - $T(n) = 2T(n/2) + \Theta(n)$
    - $\Rightarrow T(n) = \Theta(n \lg(n))$
  - Example inputs that give best case?
- Average-case, assuming random input?

Quicksort: average case

- Not every partition will be best-case ½ – ½
  - On average, in between best and worst cases
  - Even if average split is, say, 9/10 – 1/10:
    - $T(n) = T((9/10)n) + T((1/10)n) + \Theta(n)$
    - $\Rightarrow T(n) = O(n \lg(n))$
- E.g., assume splits alternate between best+worst:
  - Only adds $O(n)$ work to each of $O(\lg n)$ levels
  - $\Rightarrow$ still $O(n \lg(n))$ (albeit w/higher constant)



(a) (b)

Quicksort with constant splits

- p.178, #7.2-5: assume every split is α vs $1-\alpha$, with constant $0 < \alpha < ½$.
  - Min/max depth of a leaf in the recursion tree?
- Min depth: follow smaller side (α) of each split
  - How many splits until reach leaf (1 item)?
    - $\alpha^m n = 1 \quad \Longrightarrow \quad m = -\lg(n) / \lg(\alpha)$
- Max depth: follow larger side ($1-\alpha$) of each split
  - How many splits until reach leaf (1 item)?
    - $(1-\alpha)^m n = 1 \quad \Longrightarrow \quad m = -\lg(n) / \lg(1-\alpha)$
- Both are $\Theta(\lg n)$, so with constant-ratio splits, depth of recursion tree is $\Theta(\lg n)$, $\Rightarrow$ total complexity is $\Theta(n \lg n)$

Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

Randomised quicksort

- We saw how giving quicksort pre-sorted data results in worst-case behaviour
  - Always chose last element (r) as pivot
- We can alleviate this risk by randomising our choice of pivot:
  - rand_partition(A, p, r):
    - swap(A[r], A[rand(p, r)])   *# swap w/random item*
    - partition(A, p, r)
  - It is still possible our random pivot choices result in worst-case $\Theta(n^2)$ time – but unlikely!

Randomised quicksort: average

- Assume items are distinct, and name them in order: $\{z_1, z_2, \ldots, z_n\}$. How many comparisons?
  - Worst case: all pairs (z_i, z_j) compared $\Longrightarrow \Theta(n^2)$
  - A pair cannot be compared >1 time, because comparisons are only made against pivots, and once a pivot is used by partition(), it is not revisited
- When is a pair (z_i, z_j) compared?
  - Only if either z_i or z_j are chosen as a pivot before any other item inbetween $\{z_i, z_{i+1}, \ldots, z_j\}$
    - (If any other item is chosen first, then z_i, z_j will be on opposite sides of the split, and will not be compared)
  - $\Rightarrow$ probability is $2(\ 1 / (j - i + 1)\ )$

Randomised quicksort: average

- Summing over all pairs (z_i, z_j):

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\left(compare\ z_i\ with\ z_j\right)$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad \left(let\ k = j-i\right)$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O\left(lg\ n\right)$$

$$= O\left(n\ lg\ n\right)$$

Outline for today

- Overview of sorting algorithms
- Binary max-heaps
  - heapify(): maintaining the max-heap property
  - build_max_heap(): creating a max-heap
  - Application: Heap Sort
  - Application: Priority Queue
- Quicksort
  - Partition & pivot
  - Randomised quicksort
  - Complexity analysis
- Review for exam next week (ch1-4)

Review for Exam1: ch1-4

- Open-book, open (paper) notes!
  - But no laptop/mobile/tablet, no communication
  - Similar to textbook Exercises
- Algorith. complexity: $\Theta(=)$, $O(\leq)$, $\Omega(\geq)$, $o(<)$, $\omega(>)$
  - Know their technical definitions!
  - Proofs!
- Solving recurrences: induction, master method
- Algorithms to be familiar with:
  - Insertion sort, bubble, merge, max subarray
  - Matrix multiply (3 algorithms!)