Data Retrieval With Elementary Data Structures

Recapping Insertion/Deletion/Search Algorithms With Arrays And Lists

Elementary Data Retrieval

- Sequence of operations of mixed types
  - Insertion/deletion/search of items
- Collection of items: Accessed by an attribute (key)
  - Managed as arrays, linked lists (should be familiar to all already)
  - Binary search trees for better performance
- Time complexities of those operations on different data structures

Elementary Data Structures

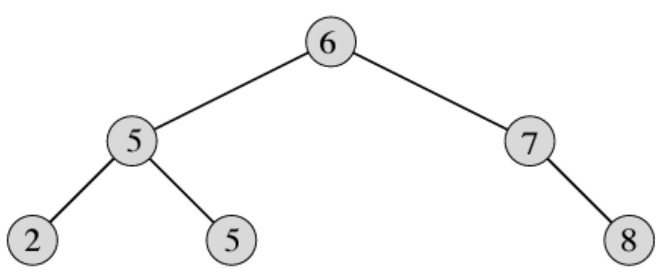
- Stacks, queues, linked lists: Undergrad prerequisites
  - Study CLRS Ch. 10 to recap or equivalent book
  - Focus on linked lists for general data retrieval operations (insert/delete/search)
  - Everyone should be able to write code for insert/delete/search on singly/doubly linked lists with *<u>pointers</u>*
- Binary tree representation using *<u>pointers</u>* (CLRS 10.4)
- Time complexities of insert/delete/search algorithms on sorted/unsorted arrays/linked lists
  - Everyone should be able to derive all these

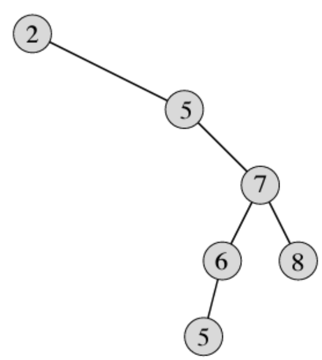
Binary Search Trees

Average-Case Logarithmic Insert/Delete/Search/Minimum/Maximum Operations

What is a Binary Search Tree (BST)?

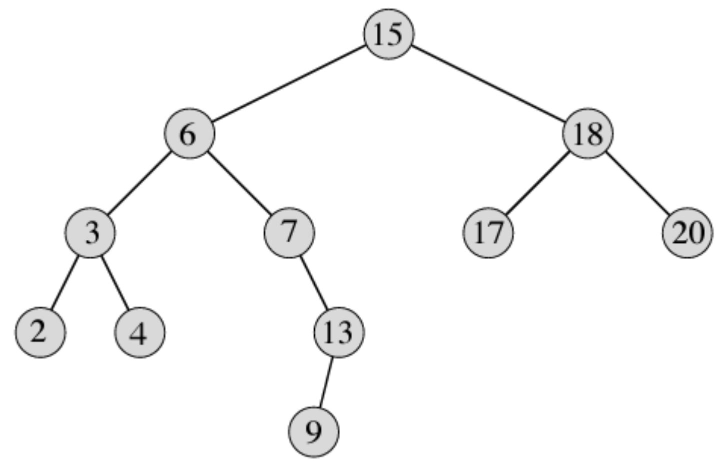
- Recursive definition
  - An empty tree is a BST.
  - A binary tree with root node r is a BST if and only if:
    - r's left/right subtree is a BST.
    - All values in r's left subtree are less than or equal to r.
    - All values in r's right subtree are greater than ("or equal to" included in CLRS) r.





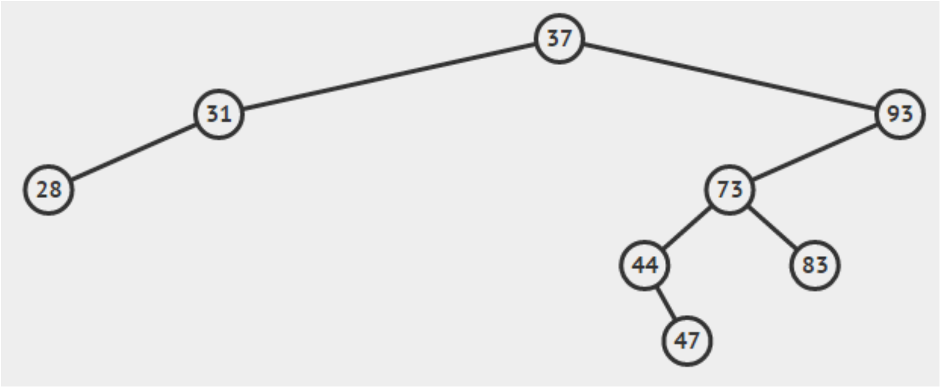
Querying Binary Search Tree

- Searching for a key
  - Very similar to binary search of a sorted array
    - The mid entry is just replaced with the tree node.
    - left = mid + 1: Traversing to the right subtree
    - right = mid – 1: Traversing to the left subtree
- Experiment BST searches at http://visualgo.net/bst
- All are $O(h)$, where h is the tree height.



Inserting in a Binary Search Tree

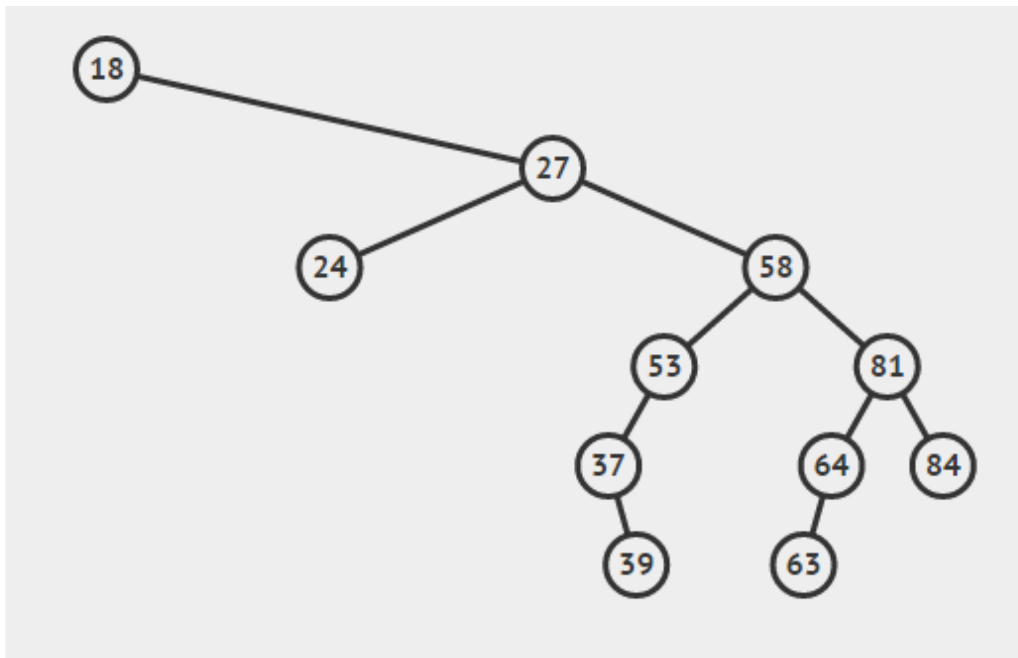
- Add a new leaf that continues to meet the BST property
- Start like search, but don't stop at a match
  - Continue until hitting a nil node
  - Add a new leaf there with the inserted value.
- http://visualgo.net/bst
- Still $O(h)$



Deleting From Binary Search Tree

- Of course search first. Return if not found.
- If the found node (call it z) is a leaf, trivial.
- If z has only one child, almost trivial.
- If z has both children,
  - Find z's right subtree's minimum (z's successor). Call it y.
  - y should be moved to z's position.
  - Filling in y's vacancy is almost trivial, as y must have no left child.
- Experiment at http://visualgo.net/bst
- Actual code (even pseudocode) can be tricky. Study CLRS 12.3 code.

BST Deletion Examples



Time Complexities of BST Operations

- All are $O(h)$
  - $h = n - 1$ in the worst case.
    - Totally skewed to one side, or zig-zag
    - Therefore, worst case BST operations are all $\Theta(n)$.
- Average case tree height
  - Expected height of a randomly built BST
  - Another probability and random variable analysis
    - See Proof of Theorem 12.4 in CLRS pp. 300-303
- Theorem 12.4: Expected height of a randomly built BST on n distinct keys is $O(\lg n)$.

Average Case Insert/Delete/Search

Operations	Unsorted arrays	Sorted arrays	Unsorted singly linked lists	Sorted singly linked lists	Unsorted doubly linked lists	Sorted doubly linked lists	BST
INSERT(A/L, i/n)	$O(n)$		$O(n)$		$O(1)$		
INSERT(A/L, k)	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(\lg n)$
DELETE(A/L, i/n)	$O(n)$		$O(n)$		$O(1)$		
DELETE(A/L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\lg n)$
SEARCH(A/L, k)	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\lg n)$
MINIMUM(A/L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(\lg n)$
MAXIMUM(A/L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(\lg n)$

Worst Case Insert/Delete/Search

Operations	Unsorted arrays	Sorted arrays	Unsorted singly linked lists	Sorted singly linked lists	Unsorted doubly linked lists	Sorted doubly linked lists
INSERT(A/L, i/n)						
INSERT(A/L, k)						
DELETE(A/L, i/n)						
DELETE(A/L, k)						
SEARCH(A/L, k)						
MINIMUM(A/L)						
MAXIMUM(A/L)						

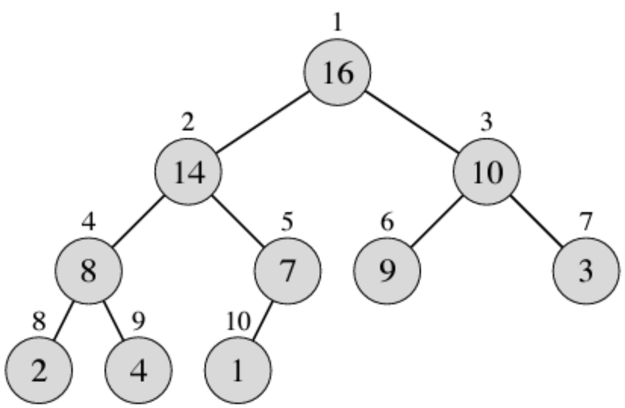
Heaps And Heapsort

When We Want O(1) MAXIMUM() (Or MINIMUM()) All The Time

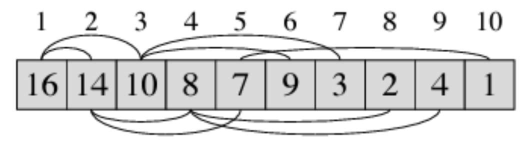
What Is a Heap?

- A data structure that's specialized for retrieving minimum (or maximum) in $O(1)$ time. This is called Priority Queue.
  - Many applications for "priority queues" in many other algorithms
  - BST can only give us $O(\lg n)$ (Even balanced BST for worst case)
- Utilize binary tree, but make sure it's as balanced as possible
  - *Complete* binary tree
    - As balanced as possible, all leaves packed to the left
  - With heap property
    - For each node, its value is less than (for min-heap) or great than (for max-heap) both of its children
  - Implemented using an array
    - No need for pointer operations/traversals

Max Heap Example (CLRS Fig. 6.1)



(a)



(b)

$$A[\textsc{Parent}(i)] \geq A[i]$$

$\textsc{Parent}(i)$

1 **return** $\lfloor i/2 \rfloor$

$\textsc{Left}(i)$

1 **return** $2i$

$\textsc{Right}(i)$

1 **return** $2i + 1$

Building A Max-Heap

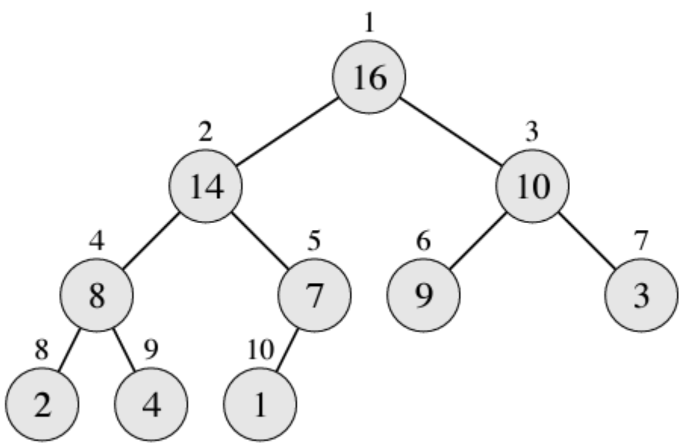
- Given an array of arbitrary values, build a max-heap.
- Two approaches:
    - Insert item by item starting from an empty heap
        - After each insertion, the resulting array must form a max-heap.
        - So fix up each inserted (appended) item by “trickling-up”.
        - n insertions, each insertion possibly taking $O(h)$, resulting in $O(n \lg n)$
    - Consider the original array as a heap
        - Of course it’s not really a heap, so fix one-by-one, from bottom up, but we do “trickling-down” here.
        - Each fix-up could possibly take $O(h)$, and there are n fix-ups possible, so this looks like another $O(n \lg n)$
        - Turns out that this is not a tight bound. It’s actually $O(n)$.
            - Analysis in CLRS 6.3

Building Max-Heap By Item-By-Item Insertions

- Given array $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$,

Building Max-Heap By Node-By-Node Fix-ups

- Given array $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$,



BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)

$l = $ LEFT(i)
$r = $ RIGHT(i)
if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY$(A, largest)$

Time Complexity Of BUILD-MAX-HEAP(A)

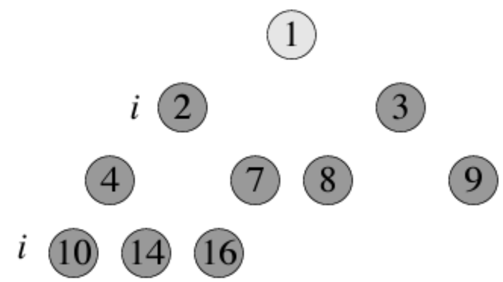
- Naïve/loose analysis: $O(\lg n)$ for each MAX-HEAPIFY(A, i), $n/2$ times, so easily $O(n \lg n)$, but this is not tight as shown below:
- Note that MAX-HEAPIFY(A, i) is not on the root (at height $h = \lfloor \lg n \rfloor$) all the time, but mostly on nodes at lower heights!
  - Up to $n/2$ nodes at height 0 (leaf), $n/4$ nodes at height 1, $n/8$ nodes at height 2, … ➔ Up to $\lceil n/2^{n+1} \rceil$ nodes at height h, where $0 \le h \le \lfloor \lg n \rfloor$
- Therefore, actual # operations is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$
$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$
$$= O(n) .$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Heapsort By Repeatedly Deleting (Extracting) Max (CLRS Fig. 6.4)

- The root of a max-heap is always the maximum of all values!
  - Remove root. Its sorted position is that of the last node of the heap.
  - Move last node in heap to root, fix-up the heap (trickle-down)
  - Then repeat this whole process until there's no node left in the heap.
- Complexity: $O(n \lg n)$ obviously.
- Experiment all heap operations at http://visualgo.net/heap



A	1	2	3	4	7	8	9	10	14	16

Heap as Priority Queue

- INSERT(S, x)
  - Insert x into queue so that GET-MAX() and EXTRACT-MAX() is efficient.
  - Place x at the end of array (last node in the heap), trickle it up. This is $O(\log n)$.
- GET-MAX(S): Always root. $O(1)$.
- EXTRACT-MAX(S): Removes & returns max of all values in queue
  - Remove root, move last heap node to root, trickle it down. This is $O(\log n)$.
- Many applications in various computer science specialty areas
  - Especially in scheduling & simulation: All about temporal priorities.
  - Also used frequently in many graph algorithms (e.g., shortest paths)