EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Growth of Algorithm Run Time with Size n: Big "Oh"

- O(g(n)) : Set of functions of n that grow no faster than g(n) as n gets large

$$f(n) \in O(g(n)) \text{ if and only if there exists } c > 0, N_0 > 0 \text{ such that} |f(n)| \leq c|g(n)| \text{ whenever } n > N_0$$

$$\text{Alternative: } f(n) \in O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in O(g(n)), \text{ or } f(n) \text{ is } O(g(n)), \text{ or } f(n) = O(g(n))$$

$$f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Growth of Algorithms: Other concepts

- o(g(n)) : grow strictly slower than g(n)

$$f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- Ω(g(n)) : grow no slower than g(n) as n gets large

$$f(n) \in \Omega(g(n)) \text{ if and only if there exists } c > 0, N_0 > 0 \text{ such that} |f(n)| \geq c|g(n)| \text{ whenever } n > N_0$$

- ω(g(n)) : grow strictly faster than g(n)

$$f(n) \in \omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

- Θ(g(n)) : grow similar to g(n): $f(n) \in O(g(n)), f(n) \in \Omega(g(n))$

MASTER EQUATION: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Theorem: The asymptotic Solution is:

- Define $\gamma = \log_b(a) \rightarrow a = b^\gamma$
- Then, there are three cases
  1. $T(n) \in \Theta(n^\gamma)$ if $f(n) \in O(n^{\gamma-\epsilon})$ for some $\epsilon > 0$
  2. $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\gamma+\epsilon})$ for some $\epsilon > 0$
  3. $T(n) \in \Theta(n^\gamma \log(n))$ if $f(n) \in \Theta(n^\gamma)$

Other useful recursive solutions

$T(n) = aT(n-1) + bT(n-2)$ (Discrete difference equation, constant coefficient)

Can solve via z-transforms (if you know EC 401..) or follow method below:

Solution form: k^n

Characteristic equation: Substitute into equation to get

$k^n = ak^{n-1} + k^{n-2} \Rightarrow k^2 - ak - b = 0$

Two solutions k_1, k_2 (or a repeated root).

If $k_1 \neq k_2$ then $T(n) = Ak_1^n + Bk_2^n$. Otherwise, $T(n) = Ak_1^n + Bnk_1^n$.

Mergesort: $\Theta(n \log(n))$ vs Timsort: $O(n \log(n))$

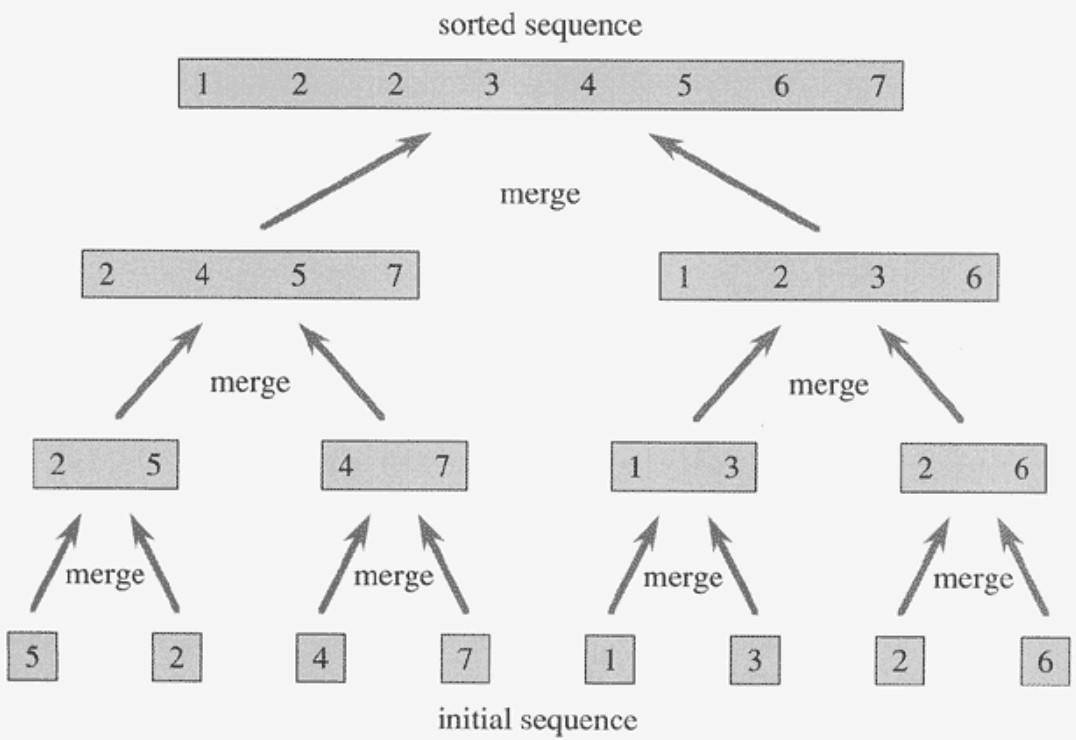
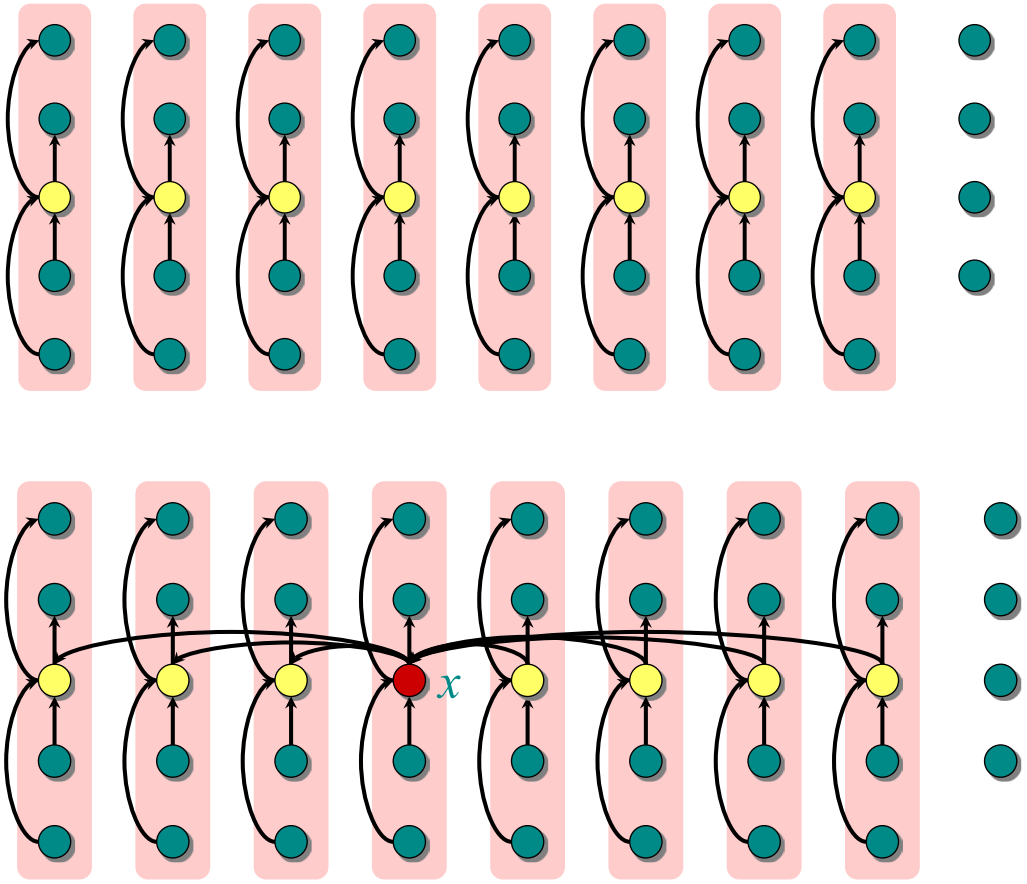


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Order Statistics: Select i-th of n is $\Theta(n)$

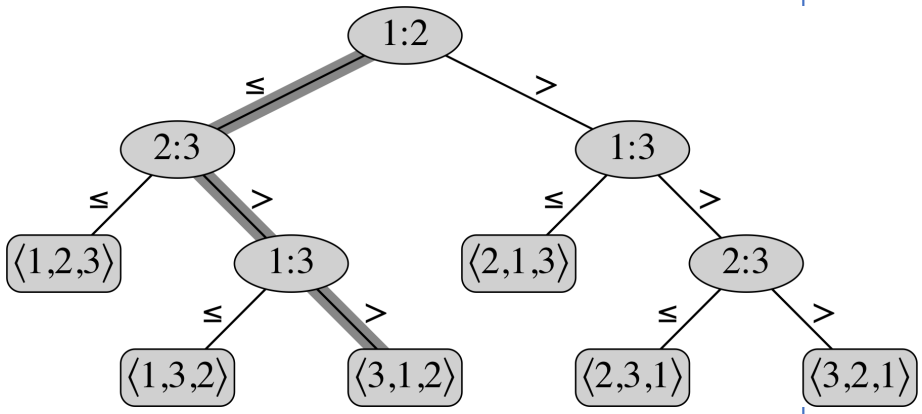


- At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians. Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\leq x$.
- Similarly, at least $3 \lfloor n/10 \rfloor$ elements are $\geq x$.
- Recursion: $T(n) \leq T(n/5) + T(7n/10) + cn$
- Result: $T(n) \in \Theta(n)$

Sorting Algorithms

- Theorem: All comparison-based algorithms are $\Omega(n \log(n))$
- Proof outline: Comparison-based algorithms are similar to binary decision trees
  - Each comparison has binary branch
  - Tree ends in leaves
  - #leaves = #permutations (n!)
  - Height of binary tree must be at least $log_2(n!) \in \Omega(n \log(n))$
  - More on binary trees later

e.g. Sort [9,1,2]



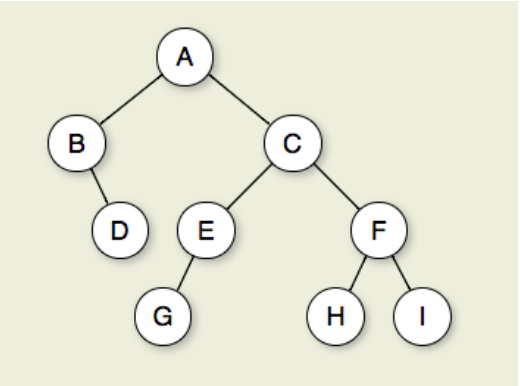
Comparison-based Sorting Algorithms

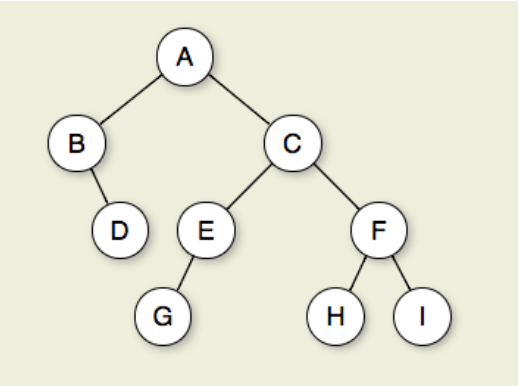
Name	Best	Average	Worst	Stable	Extra Memory	Complexity
Quicksort	$n\log(n)$	$n\log(n)$	n^2	Depends	$\log(n)$	Small
Merge Sort	$n\log(n)$	$n\log(n)$	$n\log(n)$	Yes	n	Small
Heapsort	$n\log(n)$	$n\log(n)$	$n\log(n)$	No	1	Small
Insertion Sort	n	n^2	n^2	Yes	1	Small
Shell Sort	n	$n(\log(n))^2$	$n(\log(n))^2$	No	1	Small
Bubble Sort	n	n^2	n^2	Yes	1	Small
Selection Sort	n^2	n^2	n^2	Depends	1	Small
Timsort	n	$n\log(n)$	$n\log(n)$	Yes	n	Medium

Would like optimal best case and optimal worst case! Fast & robust

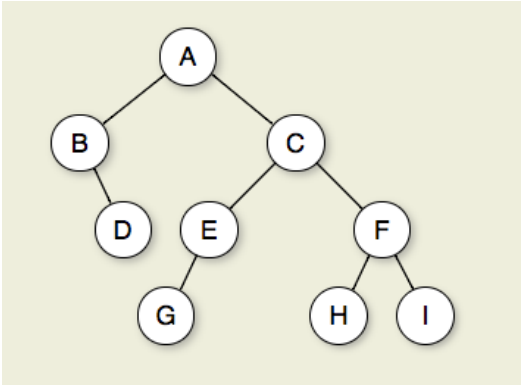
Tree vocabulary

- **Node**: an element of the tree. Contains data, pointers to other nodes
- **Root**: The entry node of the tree. Has no parent
- **Edge**: the link from a node to a child node
- **Child(X)**: A node that is linked from node X
- **Parent(X)**: The node in the tree that has X as its child
- **Sibling(X)**: a node that has the same parent as X
- **Leaf**: A node with no children nodes
- I**nternal node**: has at least one child
- **Path(**n_1, n_k**)**: a sequence $n_1, n_2, \ldots, n_k$ so n_{i+1} is child of n_i
- **Depth(n)**: number of edges on path from root to n.
- **Height**: maximum depth of any leaf
- **Ancestor**: n ancestor of m if there is path from n to m
- **Descendant**: m is a descendant of n if there is path from n to m

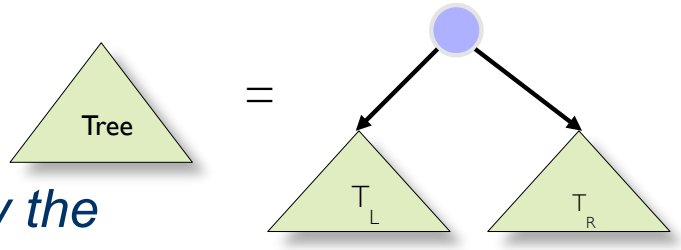




Binary Trees

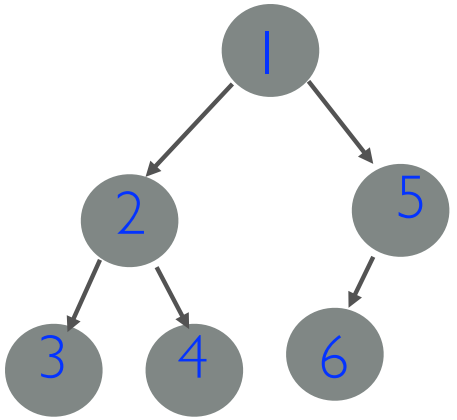


- Binary Tree: Every node has at most 2 children (0, 1 or 2)
- Recursive definition: A binary trees is null, or a singe node with a
  Right and Left Child that is a binary tree!
- Special cases:
  - Full Binary Tree: Every node has 0 or 2 children
  - Perfect Binary Tree: all interior nodes have two children and all leaves have the same depth
  - Complete Binary Tree: every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left
    as possible



Binary tree traversals

- Preorder:
```
Print [Tree]{
    Print root;
    Print Tree[LeftTree];
    Print Tree:[RightTree];
}
```
- Inorder:
```
Print [Tree]{
    Print  Tree[LeftTree];
    Print root;
    Print Tree:[RightTree]
}
```
- Postorder:
```
Print [Tree]{
    Print  Tree[LeftTree];
    Print Tree:[RightTree]
    Print root;
}
```



Pre: 1➔2➔3➔4➔5➔6

In: 3➔2➔4➔1➔6➔5

Post: 3➔4➔2➔6➔5➔1

Binary Search Tree: BST

1. BST is a Binary Tree with keys stored in each node.

2. The key (K_0) in each node is: greater or equal to all keys in T_L , the Left subtree ($K_{left} <= K_0$) less than all keys in T_R, the Right subtree ($K_0 < K_{Right}$)

3. The BST defines a partial ordered set --- as you move down to the left/right the keys decrease/increase.

4. Insert new K_{new} push down to subtree Left/Right if K_{new} <=/> K_0.

5. Delete (remove): if K_0 has two children, find **SMALLEST** key in T_R, the Right subtree.
   SMALLEST only has a right child. Right child replaces **SMALLEST** in tree, **SMALLEST** replaces K_0. K_0 is deleted.

   If K_0 has only one child, child to replace K_0 .

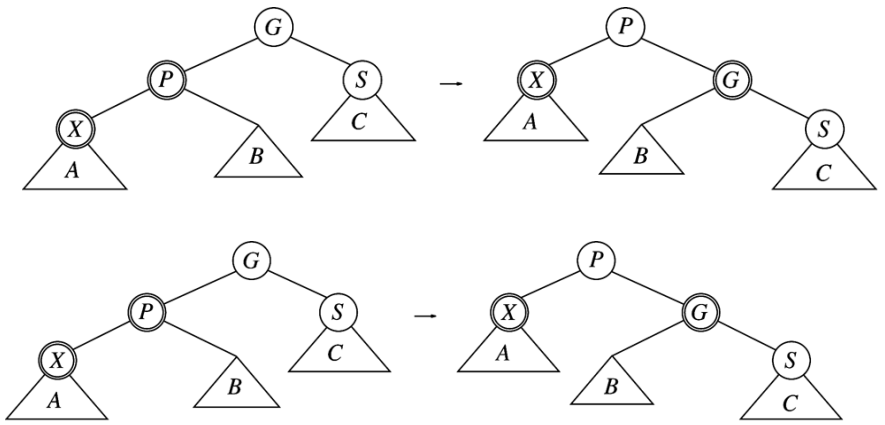
Red Black Trees

- Binary search tree
  - Each node is colored red or black.
  - The root is black
  - If a node is red, its children must be black
  - Every path from a node to a NULL reference must contain the same number of black nodes —> Black height
  - Null references are colored black
- Data structure requires an extra one- bit color field in each node

Red Black Tree Properties

- Node of black rank k has height at most 2k,
- Node of black rank k has at least 2^{k-1} descendants,
- RB Tree with n nodes has height at most 2 log(n+1)
- Proof: (More detail in book…merge red nodes with parent black nodes…)
  - Bottom layer has at least 2^k nodes. If k is black rank of node, then max number of nodes in path to leaf is K,
    min number of nodes n in path to leaf is K/2.
  - So, tree is **full** to level K/2.
  - Number of nodes n is at least $2^{K/2}$
  - So, $n + 1 \geq 2^{K/2} \Rightarrow L \leq 2\log_2(n+1)$

- Know: Insert, delete

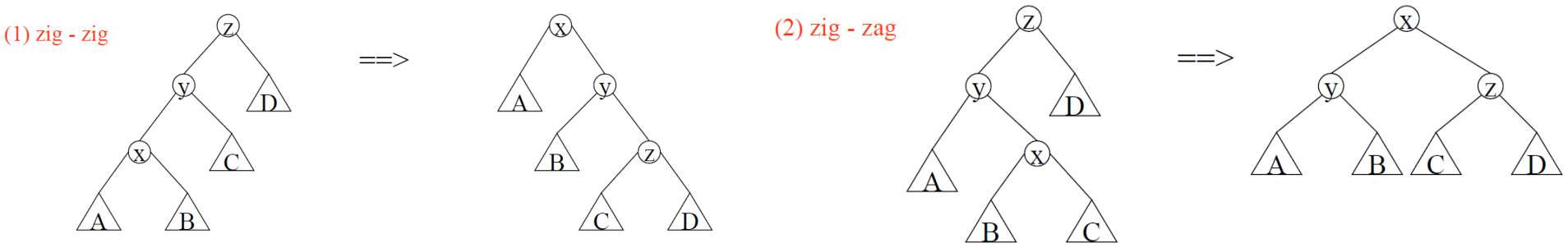


Splay Trees

Another binary tree approach: Splay trees are self-adjusting binary trees.

- "Lazy" data structure: Don't insist on balance all the time, but recover balance when needed
- Amortized complexity is O(log(n))

Know how to insert, delete, join, splay



Amortized Analysis

In the **aggregate method**, we directly evaluate $T = \sum_{i=1}^{m} t(op_i)$, and set a($op_i$) = T/m

- Assigns each operation the average of all the operation costs

In the **accounting method**, we place "credits" on the data structure for some operations redeemable for units of work in other operations

In the **potential method,** we assign a potential function to the data structure and factoring in changes to that potential for each operation to the overall runtime

Know what it is, roughly how to do it.

B+-Tree

- A B^+-tree of order M is an M-ary tree with the following properties
  - Data items are stored at the leaves
  - Non-leaf nodes store up to M-1 keys; keys at node are sorted
  - Non-leaf nodes have between $\lceil \frac{M}{2} \rceil$ and M links to children, except for root
  - The root is either a leaf or has from two to M children, 1 and M-1 keys
  - All leaves at the same depth, and contain the data items for the tree
  - Leaves have between $\lceil \frac{L}{2} \rceil$ and L data items
- KNOW: what they are, how to do inserts, deletes

Priority Queue

- Main operations
  - **Insert** (i.e., enqueue)
  - **deleteMin** (i.e., dequeue)
    - Finds the minimum element in the queue, deletes it from the queue, and returns it
  - **getMin**: Find the highest priority job
  - **deleteAny**: Delete a job from the queue
  - **Create**: Create a priority queue from a list of jobs
  - **Merge**: merge two priority queues into one
  - **decreasePriority**: decrease priority of existing job in queue

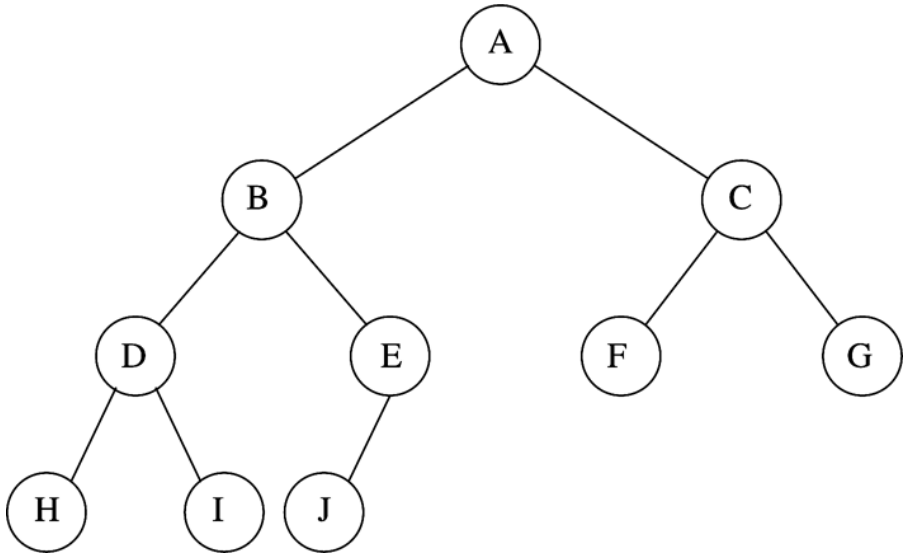
- Desired Performance
  - Goal is for operations to be fast: O(1) to O(log(n))
  - Will be able to achieve O(logN) time insert/deleteMin

Binary Heap

- A binary heap is a binary tree with two properties
  - Structure property
    - A binary heap is a complete binary tree
    - Each level is completely filled
    - Bottom level may be partially filled from left to right
  - Heap-order property
    - Parent node must have key less than or equal to the keys of its children

- Complete binary tree —> easy implementation as array
  - Height of a complete binary tree with N elements is **Floor**[logN]

Binary Heap



- Know: how to insert, remove-min, upheap, downheap, build heap complexity and operations

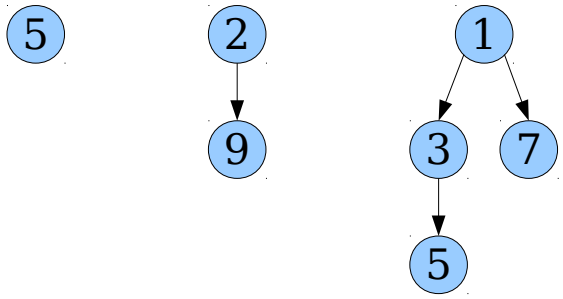
Binomial Heaps

- Forest of binomial trees (Note: not binary!)
  - Each in heap order
  - Each of a different height
- Forest: a collection of trees
- A binomial tree B_k of height k consists two B_{k-1} binomial trees
- The root of one B_{k-1} tree is the child of the root of the other B_{k-1} tree
- KNOW: what they are, how to insert (non-lazy and lazy), remove_min, join

Heap-Ordered Binomial Trees

- Heap-ordered binomial trees: Trees satisfy heap property
  - Key of parent is less than or equal to keys of children

- **Binomial heap**: A collection of heap-ordered binomial trees
  - Pointer to the tree with the min value
  - No two binomial trees are of the same order
    - Boolean representation



- Insert: O(1); Merge: O(1); Find Min: O(1); Delete Min: O(log(n))

Hash Tables

- Hashing
  - Technique supporting insertion, deletion and search in average-case constant time
  - Operations requiring elements to be sorted (e.g., FindMin) are not efficiently supported
- Generalizes an ordinary array,
  - Key property: direct addressing
  - An array is a direct-address table: Key value is position of data in array
- Basic operation should be O(1)!
-

Collision Resolution

- Chaining
- Open addressing:
  - Linear Probing: f(k) = k
  - Quadratic Probing: f(k) = k^2
  - Double Hashing: two hash functions
  - Cuckoo Hashing:

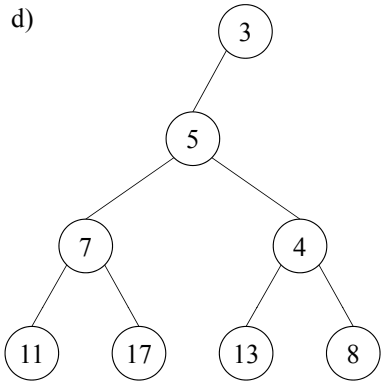
- KNOW: How to execute a hash insert with different collision resolution techniques

Fibonacci Heaps

- Fredman-Tarjan 1986
  - CLRS Chapter 19
- Binary heaps: Insert: O(log(n)), Merge: O(n), DeleteMin: O(log(n)); DecreaseKey: O(log(n))
- Lazy Binomial heaps: Insert: O(1); Merge: O(1); DeleteMin: O(log(n)) (amortized)
  DecreaseKey: O(log(n))
- Fibonacci heaps: DecreaseKey O(1) amortized
- Know: how to do inserts, remove_min, decrease key

Rank-Pairing Heaps

- Half-ordered binary trees (half-trees)
  - Root has single left child
  - Half-heap property: parent key less than or equal to left child
- Rank of a node in half trees: similar to height…
  - If node is leaf, rank = 0
  - If it is root, rank(parent) = 1+rank(left child)
  - If |rank(left child) - rank(right child)| <= 1: rank = 1 + max(rank(L),rank(R))
  - If |rank(L) - rank(R)| 1: rank(parent)= max(rank(L), rank(R))



- Insert: O(1); Merge: O(1); DeleteMin: O(log(n)) (amortized), DecreaseKey: O(1) amortized

- KNOW: what they are, how to do insert, delete_min, decrease key, how to compute ranks

String Matching: Knuth-Morris-Pratt Algorithm

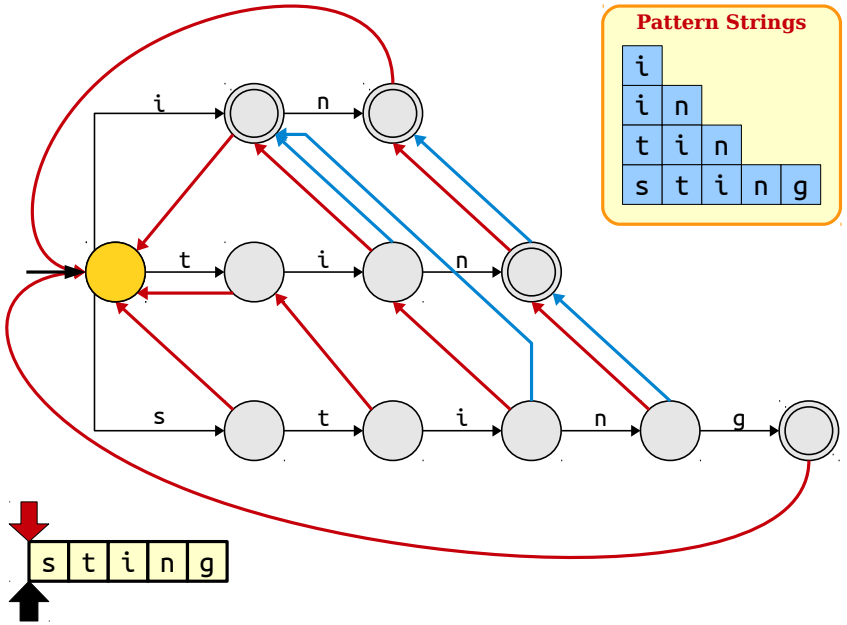
- Patterns have a prefix function
  - A string w is a **proper prefix** of a string x if x = w + v for some string non-empty string v, where + is concatenation
  - A string w is a **proper suffix** of a string x if x = v + w for some non-empty string v

Know: how to compute prefix function for a pattern, how to use a prefix function to match pattern into string

String Matching: Aho-Corasick Algorithm

- Finds k patterns with total length n in text of length m by building a suffix trie in O(n) and finding z matches in O(m+z)
  - Generalizes KMP algorithm

- KNOW:
  - How to construct tries
  - How to find suffix links
  - How to find output links
  - How to use tries with links to search a long string



Disjoint Sets

- Operations
  - Find(a): Find the subset that contains element a
  - Union(a,b): Add a relation between two elements a, b which merges the subsets containing a, b
  - Often known as Union-Find problem
  - Complexity of best algorithms: O($\log^*(n)$) amortized cost, which is approximately O(1)

- KNOW: how to merge by rank with path compression, how to update ranks

EC 504 – Fall 2020 – Take Home Midterm

Due Friday , Nov 6, 2020 at 11:59. Both written and coding problems submitted in the directory `/projectnb/alg504/yourname/MT` on your SCC account by Friday Nov 6, 2020 at 11:59PM. All your work must be your own. NO collaborations.

1. (10 pts) Answer True or False to each of the questions below. Each question is worth 2 points. Answer true only if it is true as stated, with no additional assumptions. No explanation is needed, but any explanation is likely to earn partial credit, and no explanation will not earn any credit if the answer is wrong.

   (a) All heaps can be represented by full trees.

   (b) Any sorting algorithms for 32 bit positive integers that only uses the $\leq$ comparison test must have worst case performance $\Omega(N \log(N))$.

   (c) It is possible to formulate Quick Sort to be worst case $O(N \log N)$ including the cost of picking a suitable pivot.

   (d) Given an array of N integers, the best algorithm has a worst case time to build the heap $\Theta(Nlog(N))$.

   (e) The two traversal BFS (bread first search) and DFS (depth first search) for a tree can implemented by entering the nodes one at time into Stack and Queue data structure respectively.

   (f) $N^2 e^{\ln(N^{-1})} \in \Theta(N)$.

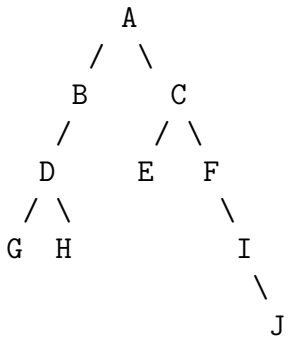
   (g) $\log(N)(1 + 1/N)^{N^2} \in \Theta(e^N log(N))$ HINT: $ln(1 + x) \simeq x$ for small x

   (h) $N^3 + 200N^2 \in \Theta(200N^3 + N^2)$

   (i) If $f(N) \in O(g(N))$ and $g(N) \in \Theta(h(N))$, then $f(N) \in O(h(N))$

   (j) If $f_i(N) \in O(N^2)$, then $\sum_{i=1}^{N} f_i(N) \in O(N^3)$

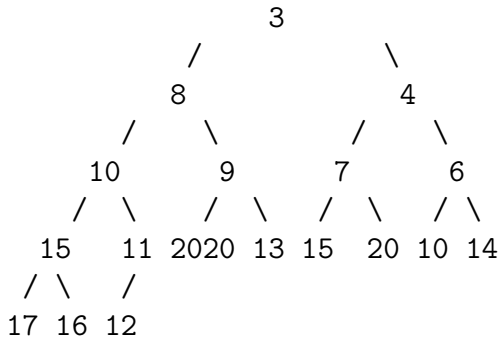
2. (10 pts) Consider the following tree:



(a) List the height and depth of each of the nodes A-J

(b) List the nodes in pre-order, in-order and post-order.

3. (15 pts) Consider the following min-heap.

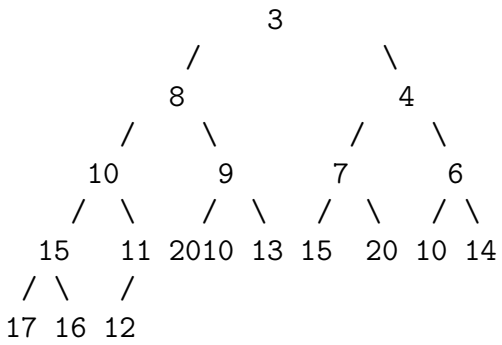


(a) Show the min-heap which results after inserting the element 2 in the heap. (Indicate the sequence of steps with arrows.) Then in this new heap show the steps required to delete element 3. Draw the final min-heap

(b) Consider min-heap as an array **int a[N+1]** with the size of heap stored in **a[0] = N**. (For example the one above has **a[0] = 18** and $a[1], a[2], \cdots, a[18]$. Describe carefully an $\Theta(N \log N)$ algorithm to sort `in place in place` the array elements $a[1], a[2], \cdots, a[N]$ in descending order. Use $a[0]$ the decreasing number remaining in the heap as you proceed with the sort.

(c) Now use $a[0]$ as a temporary to re-arrange `in place` the sort into ascending order with $O(N)$ extra swaps.

(d) Re-arrange the original min-heap (repeated below) into a max-heap by a "bottom up" $O(N)$ algorithm. Describe the steps level by level.



Do this instead for RB trees, B+-trees

4. (15 pts) This problem is to construct step by step BST and AVL trees given the following list of $N = 10$ elements.

$$8 \quad 12 \quad 2 \quad 4 \quad 8 \quad 1 \quad 36 \quad 33 \quad 35 \quad 64$$

   (a) Insert them sequentially into a BST (Binary Search Tree).

   (b) Insert them sequentially into an empty AVL tree, restoring the AVL property after each insertion. Show the AVL tree which results after each insertion and name the type of rotation (RR or LL zig-zig or versus RL or LR zig-zag).

   (c) Relative to the BST has AVL tree decreased the total height $T_H(N)$ and the total depth $T_D(N)$? Give the amount that these have change.

   (d) For $N = 2^{16} - 1$ what is the max and minimum values for $T_H(N) + T_D(N)$.

6. (10 pts) Given an array of n-distinct positive integers $a[n]$ and a second array of n-distinct positive integers $b[n]$ to construct the sum:

$$\sum_{i=0}^{n-1} a[i] * b[i] \tag{1}$$

   (a) Describe in a few words, an algorithm to find the maximun allowing different permutation both arrays $a[n]$ and $b[n]$. Give your best estimate of a bound on $T(n)$ the optimal method for this problem.

   (b) Now consider maximizing this sum by only permuting the the elements of $b[i]$. How does this maximun compare to the result of the first method? Describe you best algorithm for this method and estimate is scaling in n.

Remaining 40 points: Coding AVL inserts
We don't do coding in real-time exams; not a race!