Cheat sheet for CS251. Useful for reviewing material as well as writing what you don’t remember well for the exam. Of course you’ll have to handwrite your material, so be sure to keep that in mind for the final.

You are highly encouraged to edit, however information must be **100% accurate**. Although this is a more in-depth cheat sheet to help you understand new information, keep the information you put here as short and simple as possible: long or super in-depth explanations are unnecessary.

No curse words of any kind, or you will be immediately kicked out of sharing.

Any sections, data structures, algorithms, methods (etc.) highlighted yellow are those that need a description. If you see any highlighted, please add on to the details for it and un-highlight it when finished.

Textbook / lectures:

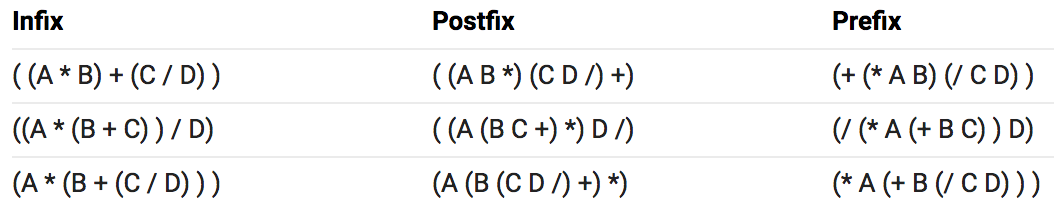
Union-Find

* Union: connect two objects; Find: is there a path between them?
  + Name objects 0 to N-1, integers as array index
  + Connected components: reflexive (p to p), symmetric (p to q, q to p), transitive (p to q, q to r, p to r)
    - Find: check if two objects in same component
    - Union: replace components with their union
      * {0, 1, 2} and {3, 5, 6}, union(2, 5) yields {0, 1, 2, 3, 5, 6}
  + Implementation
    - Constructor: union-find data structure of N objects
    - Boolean find(int obj1, int obj2)
    - Void union(int obj1, int obj2)
    - Int count(), returns number of components
* Quick find
  + Integer array of size N, objects in same component if same id
    - i 0 1 2 3 4 5 6 7 8 9
    - id[i] 0 1 9 9 9 6 6 7 8 9
  + Implementation
    - Find: check if obj1 and obj2 have same id
    - Union: change all entries with id[obj1] to id[obj2]
      * Union(3, 6) to above example yields:
      * i 0 1 2 3 4 5 6 7 8 9
      * id[i] 0 1 6 6 6 6 6 7 8 6

Proof methods

* Direct derivation: A and (A -> B) -> B
* Proof by contraposition: Original: p ⇒ q, Contrapositive: ¬ q ⇒ ¬ p
  + Prove for all possible integers, if n2 is odd, then n is odd.
  + Contrapositive: Prove that if n is even, then n2 is even.
    - Assume that n is even (n = 2k)
    - Then n2 = 4k2 = 2(2k2) = 2x, which is even
* Proof by contradiction: Original: p ⇒ q, Contradiction: use p ∧ ¬ q ⇒ False
  + Claim: If 3x = x, then x = 0.
    - If the claim is wrong, then 3x = x is true for some x ≠ 0
    - Divide both sides by x (since x ≠ 0)
    - This implies 3=1, which contradicts existing knowledge
    - Therefore, if 3x=x then x=0
* Recursion: a set of base cases and a recursive definition
  + S(1) = 2 base case; S(n) = 2S(n – 1) for n ≥ 2, yields {2, 4, 8, 16, …}
  + Often proved by mathematical induction
* Proof by induction: claim is true for all natural numbers
  + Basis: Claim is true for simple cases (n = 1)
  + Assumption: Claim is true for some integer n ≥ 1 (assume claim is true for n = k).
  + Induction: Based on the assumption, show the claim must be true
  + for n = k + 1.
  + Conclusion: Since claim is true for the base case, must be true for all subsequent cases.

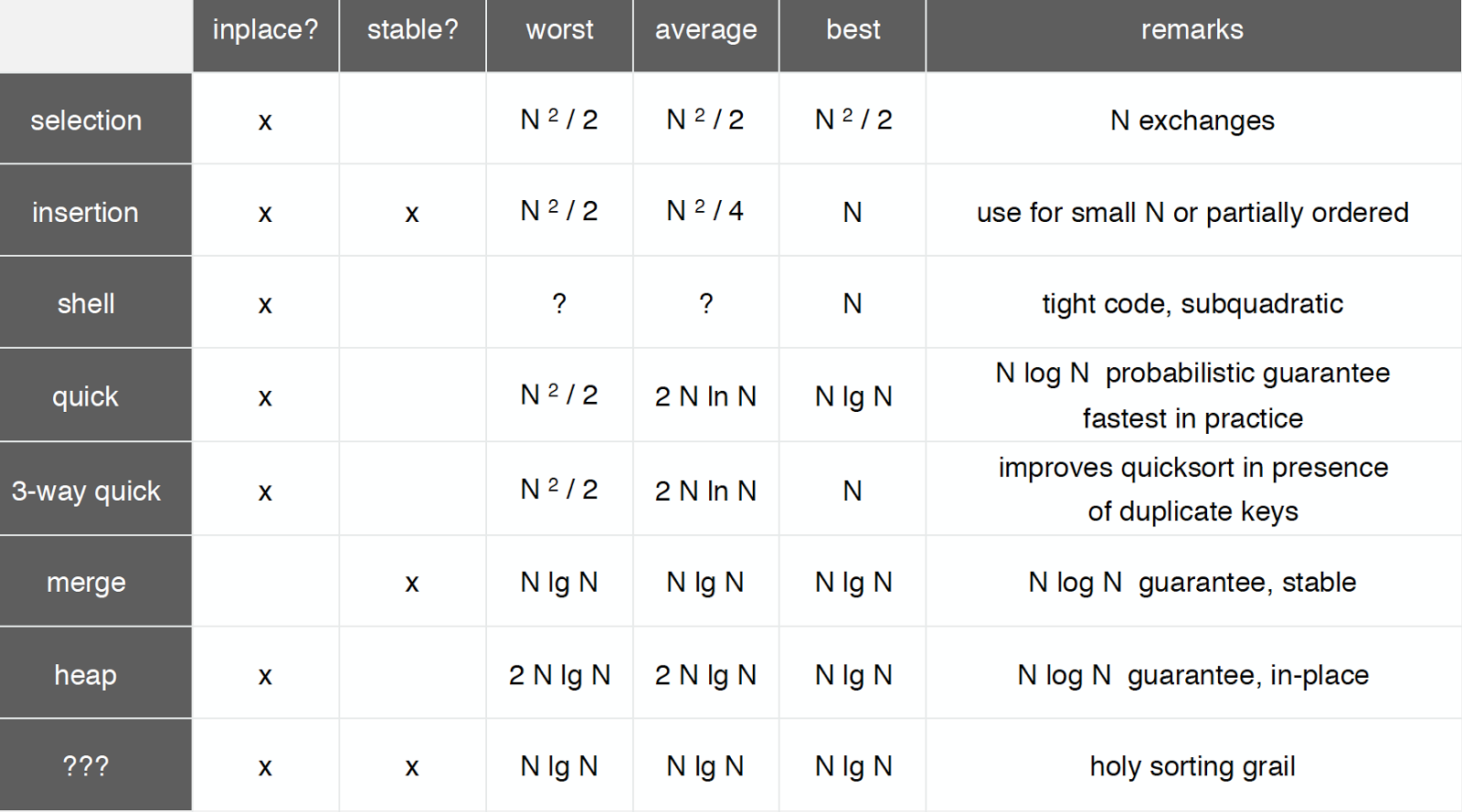
Stacks and Queues

* Linked list: sequence of nodes, each store item(s) and next node
* Doubly-linked list: each node stores item(s), next node and previous node
* Stack: remove item most recently added (LIFO, “last in first out”)
  + Push(item *obj*): move *obj* to beginning of stack (prepend)
  + Pop(): if stack isn’t empty, remove (and return) first item of stack
  + If using array implementation, use dynamic resizing:
    - If array is full, create a new array of twice the size, and copy items
      * Push: double size of array when full
      * Pop: half size of array when array is one-quarter full
* Queue: remove item least recently added (FIFO, “first in first out”)
  + Enqueue(item *obj*): insert *obj* onto queue (append)
  + Dequeue: remove item least recently added (same as pop for stack)
  + Dynamic resizing (same idea for stack)
* Generators (lecture slide 3 #44 - 48) (genericizing?)
* Iterators
  + Support iteration over stack items by client without revealing the internal representation of the stack
  + Iterable interface
    - Returns iterable, which contains “hasNext()” and “next()” methods
* Infix, postfix, prefix
  + 
  + Operators are “in” between variables, “post” after variables, “pre” before
  + Parentheses become redundant for postfix or prefix

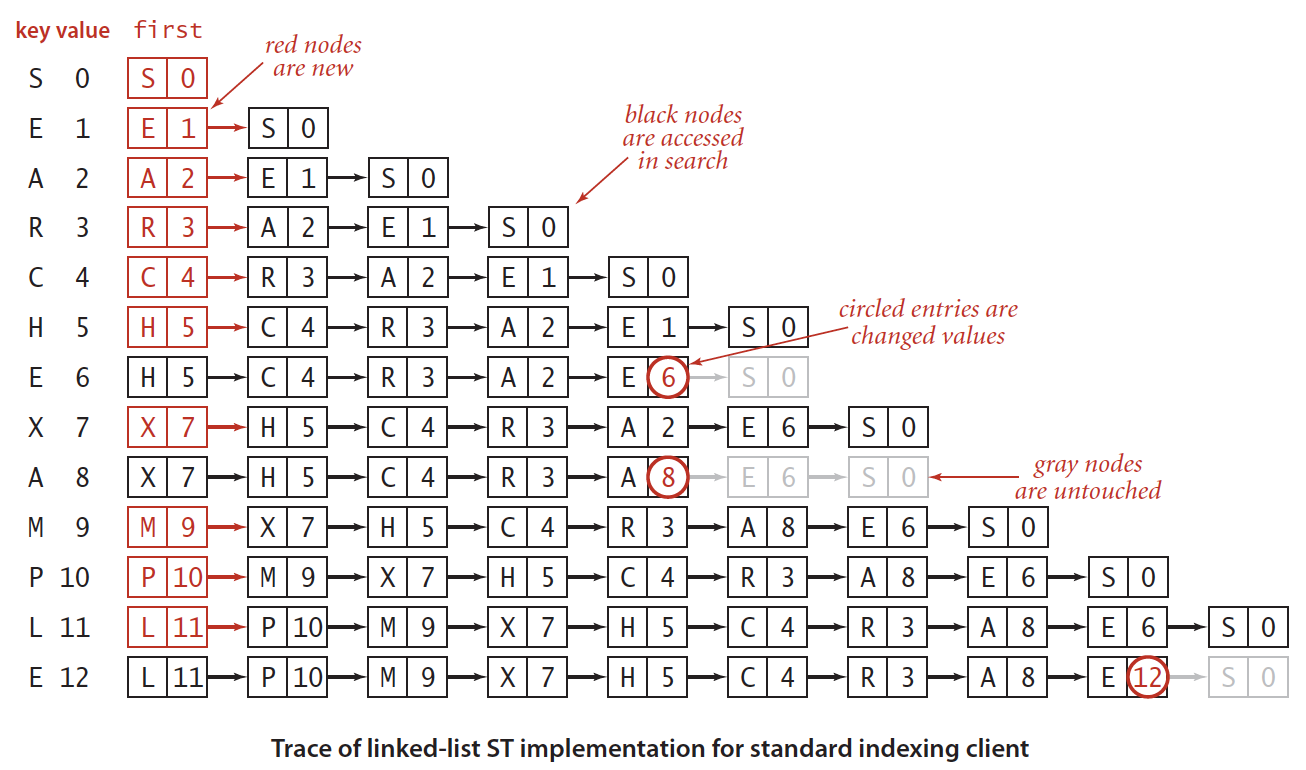
Analysis

* Amortized analysis
  + Average running time per operation over a worst-case sequence of operations
    - Useful to compute average cost per operation over a sequence of ops
* Tilde notation:
  + Take the leading term
  + 10 N2 + 22 N log N
    - In Tilde notation: ~10N2
  + Provides approximate model
* Big-theta:
  + Practically same thing as Tilde notation without coefficients
  + Asymptotic growth rate, used to classify algorithms
  + 5 N2 + 22 N log N + 3N, 10 N2
    - In Big-theta: Θ(N2)
* Big-O:
  + Worst case, upper bound
  + 22 N log N + 3 N
    - Big-O: O(N log N), but even O(N2) works
* Big Omega:
  + Best case, Lower bound
  + N3 + 22 N log N + 3 N
    - Big Omega: Ω(N2), even Ω(N) works
* Analyzing loops, recurrences.

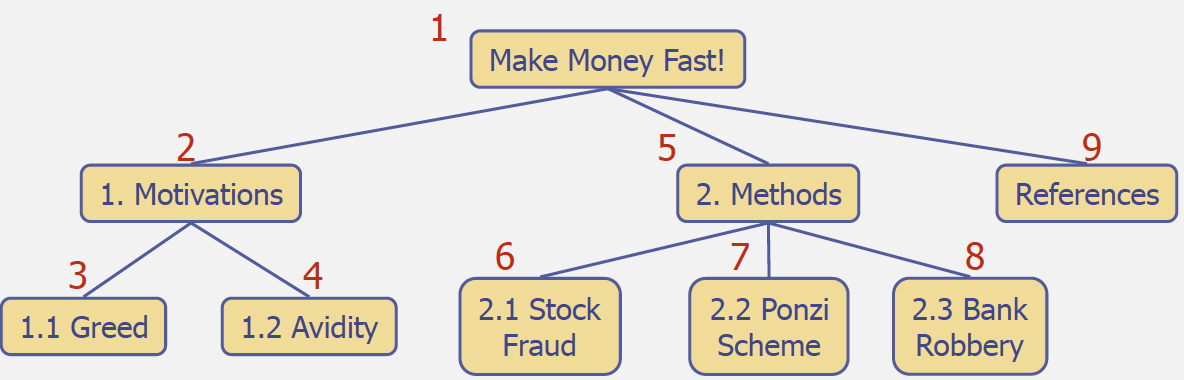
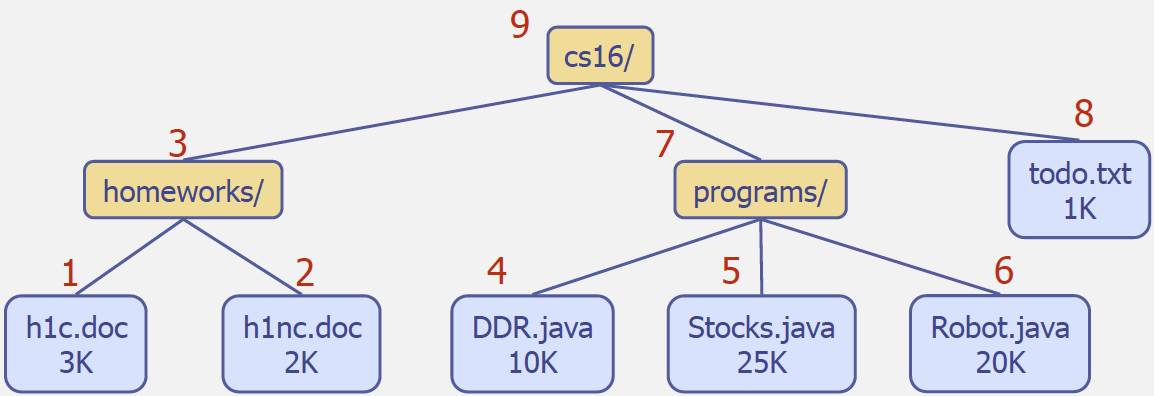
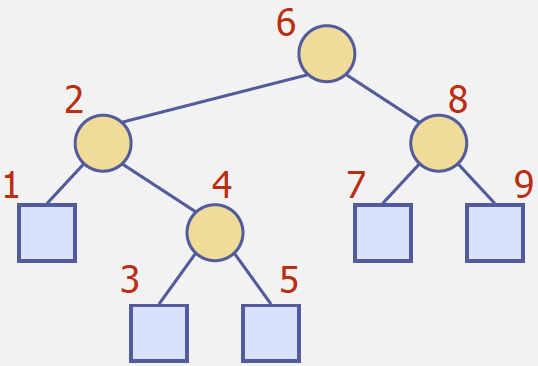
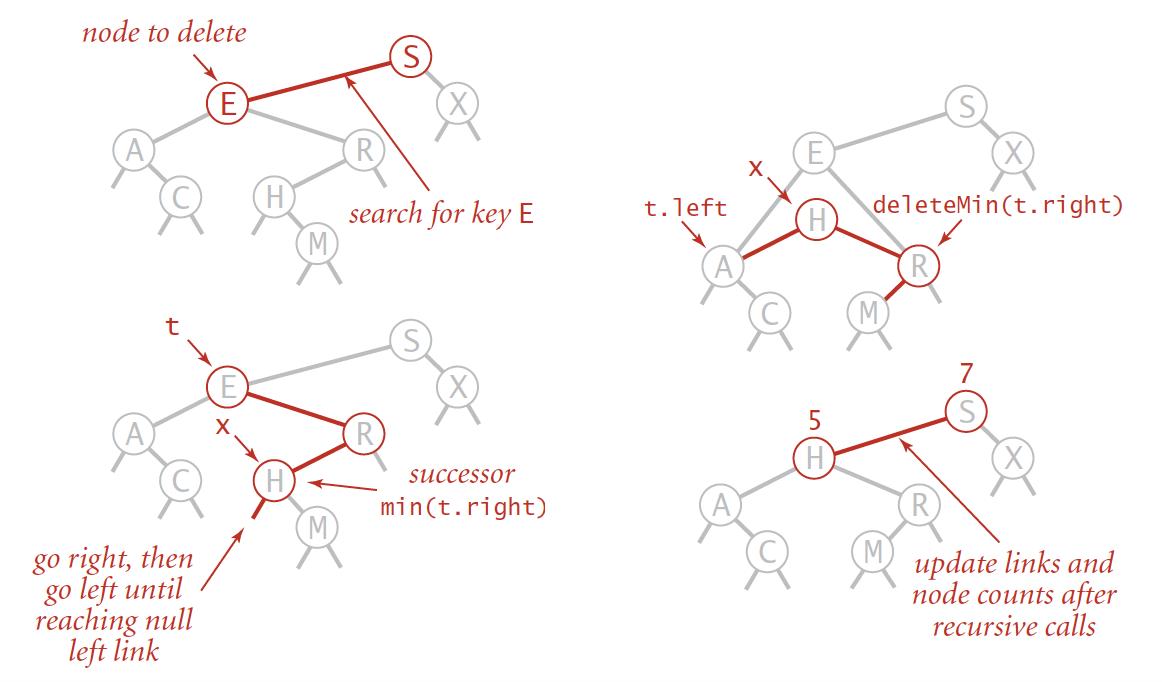
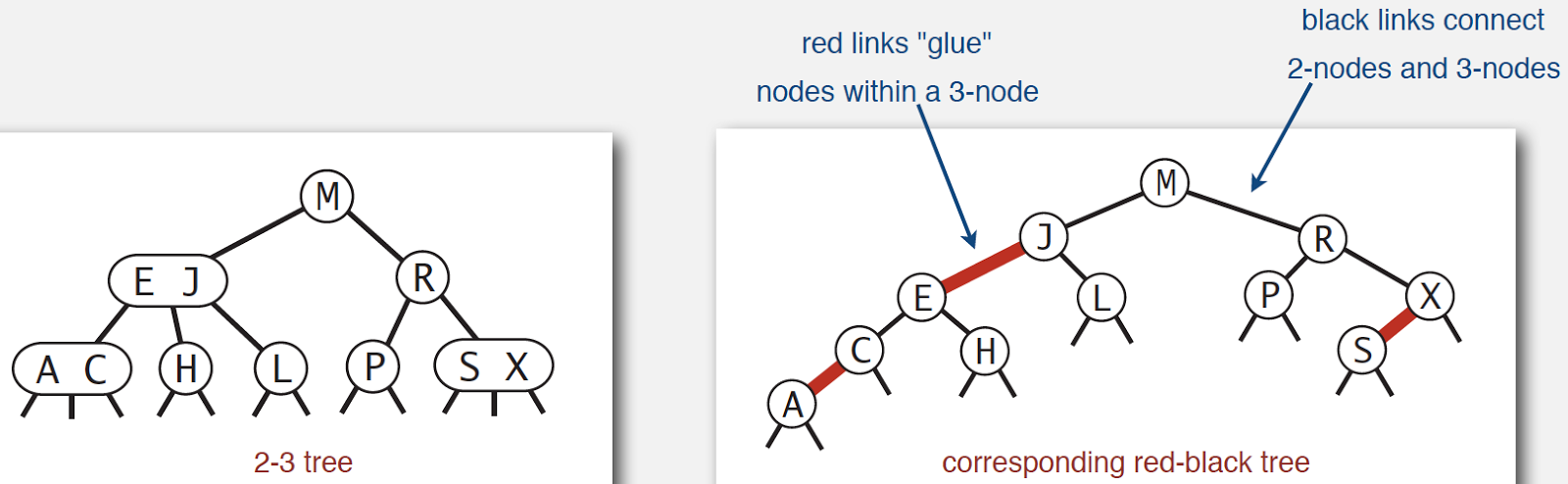
Sorts

* Selection sort: find minimum, exchange with first element
  + Then find subsequent minimum, exchange with second element (etc.)
* Insertion sort: for each element, compare with previous elements and exchange if two elements are not sorted
  + Repeat until two elements being compared are sorted
  + An inversion is a pair of keys that are out of order.
    - T R X P S: T-R, T-P, T-S, R-P, X-P, X-S (6 inversions)
    - Number of exchanges = number of inversions
  + Stable as well
* Shell-sort: insertion sort with stride length
* Mergesort: recursively divide and sort halves of array, then merge two halves
  + Bottom-up mergesort: sort sub-arrays of size 1, then 2, 4, 8, etc.
* Comparators:
  + Implements compare() function of some kind
  + (Project 2)
* Quicksort: first shuffle array, select one object (*obj1*)
  + Scan left side for object *obj2* >, then scan right side for object *obj3* < *obj1*
    - Exchange *obj2* and *obj3* *obj1*
    - Repeat scanning and exchanging objects as described
    - When pointers cross, exchange *obj1* with *obj3*
  + 3-way quicksort
* Great video animating mergesort and quicksort: <https://www.youtube.com/watch?v=es2T6KY45cA>
* Binary tree: empty or node with links to left and right binary trees
  + Balanced if every node on every level is filled
  + N nodes, height = 1 +⎣lg N⎦
  + Array representation, root node at index 1 (not 0), following nodes from left to right, top to bottom (level order)
    - Using dynamic resizing
  + Binary heap: insert at next available node, swim up if child node’s value is greater than parent node’s value
    - Delete maximum by switching root node with last node, delete last node (should have max), sink down root node
* Heapsort (max / min-oriented):
  + Start at lowest level on right side of binary tree
    - Take greater / lower of child nodes, switch with parent if child greater / less than parents
  + To remove max / min (root node)
    - Switch root value with last node value in tree
    - Delete last node (has max/min value)
    - Sink value down
    - Minimum visualization (with numbers): https://www.cs.usfca.edu/~galles/visualization/Heap.html
* 

Symbol Tables

* Key-value: get value by entering in key
* Implementation
  + Constructor: create a symbol table
  + void put(Key key, Value val) put key-value pair into the table
  + Value get(Key key) value paired with key
  + void delete(Key key) remove key-value pair from table
  + boolean contains(Key key) is there a value paired with key?
  + boolean isEmpty() is the table empty?
  + int size() number of key-value pairs in the table
  + Iterable<Key> keys() all the keys in the table
* Sequential search
  + 

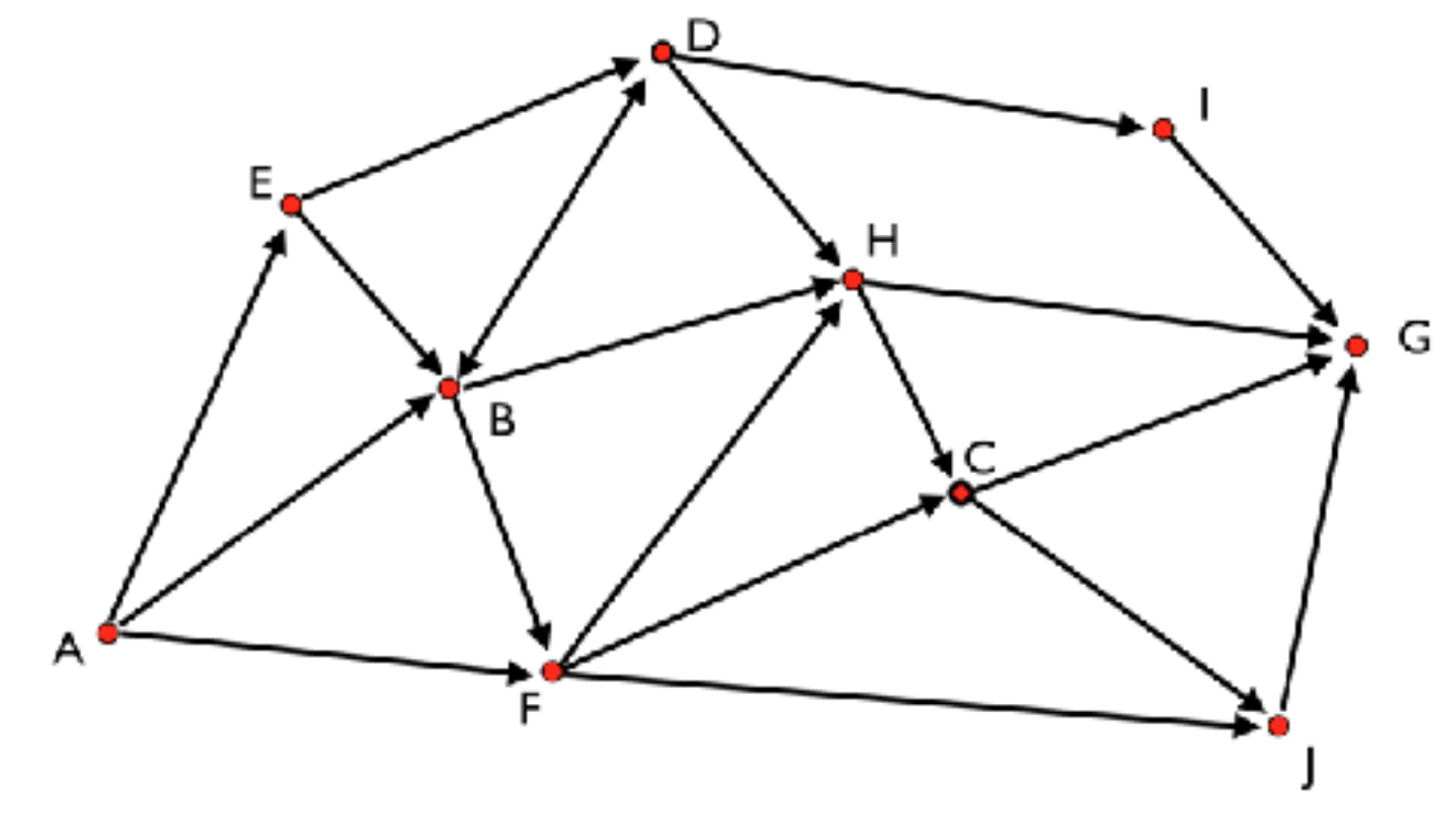
Binary Search Trees

* Preorder traversal: node visited before descendants
  + 
* Postorder traversal: node visited after descendants
  + 
* Inorder traversal: node is visited after its left subtree and before its right subtree
  + 
* Binary search trees (BST), binary tree in symmetric order
  + Each node has a key, and every node’s key is:
    - Larger than all keys in its left subtree.
    - Smaller than all keys in its right subtree
  + Key, value, left subtree, right subtree
  + Either empty or two disjoint binary trees (left, right)
  + Implementation:
    - Search: start at root node
      * If *key* < root, go to left subtree; else right subtree
      * Repeat until either found or null (*key* not in BST)
    - Insert: start at root node
      * If search *key* < root, go to left subtree; else right subtree
      * Repeat until subtree is null, place *key* in subtree (if *key* in subtree, do nothing)
  + Find minimum or maximum of BST
    - Keep going to left node until *null* (min.), keep going right for max.
  + Floor and ceiling
    - Floor: Go to the left, and then find the max of that node
    - Ceiling: Go to the right, and then find the min of that node
  + Delete a node
    - 
* 2-3 search trees
  + Allow 1 or 2 keys per node: 1 node
  + Perfect balance: root to null is same height
  + Implementation (logarithmic for both):
    - Search: (same as BST)
    - Insert: (similar to BST, until inserting *key* into 2-key node)
      * (Look at diagrams in Lecture 11, slides 10 - 39)
  + Visualization (don’t insert duplicates): <https://www.cs.usfca.edu/~galles/visualization/BTree.html>
  + Tree height: logN
* Left-leaning red-black BSTs
  + Replace 3-node with left-leaning red line, larger key is root
  + A BST such that:
    - No node has two red links connected to it.
    - Every path from root to null link has the same number of black links.
    - Red links lean left
  + 
  + Left rotation is necessary for LL, right rotation is temporary, color flip if two connecting red-lines
  + (Look at lecture 11, slides 56 - 65)
  + Height of tree is ≤ 2 lg N in the worst case

Hash Tables

* Save items in a key-indexed table (index is a function of the key)
* Properties of good hash functions.
  + Equals keys must produce the same hash value
  + Efficient to compute
  + Uniformly (equally) distributes keys
* Selecting hash table size
  + Load factor λ: “The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased”
  + Default λ is .75
  + Greater than .75 decreases overhead but increases lookup cost
  + Less than .75 vice-versa
* Uniform hashing assumption
  + Each key is equally likely to hash to an integer between 0 and M - 1
* Separate chaining and open addressing
  + Array of linked lists
  + Hash function maps object to array index *i*
  + Prepend object to linked list at *i*
  + Analysis:
* Linear Probing
  + In simplest form, hash to array index, if unavailable go to next available
  + Hash: maps key to integer i between 0 and M - 1.
  + Insert: put at table index i if free; if not try i + 1, i + 2, etc.
  + Search: search table index i; if occupied but no match, try i + 1, i + 2, etc.
* Double Hashing to resolve collisions.
  + Use linear probing, but skip a variable amount, not just 1 each time
    - Effectively eliminates clustering
    - Can allow table to become nearly full
    - Difficult to implement delete
* Rehashing
  + (Lecture 12)

Undirected Graphs

* Probably best video for searches: <https://www.youtube.com/watch?v=bIA8HEEUxZI>
* Depth-first search
  + Algorithm
    - Mark v as visited.
    - Recursively visit all unmarked
      * Vertices w adjacent to v.
    - 
    - (On midterm 1), above graph with alphabetical order:
      * Visits in following order (highlighted is already visited):
        + A B D H C G C J C H D I D B F B A E
        + Remove visited to get final order: A B D H C G J I F E
  + Implementation
    - Search(Graph G, int s) find vertices connected to s
    - boolean marked(int v) is vertex v connected to s?
    - int count() how many vertices connected to s?
  + Unvisited vertices on stack
* Breadth-first search
  + Algorithm
    - Put *s* onto FIFO queue, and mark *s* as visited.
    - Repeat until the queue is empty:
      * Remove the least recently added vertex v
      * Add each of v's unvisited neighbors to the queue, and mark them as visited.
  + Useful for shortest path (shortest number of edges)
    - BFS examines vertices in increasing distance from s
  + Unvisited vertices on queue
* Connected components
  + Initialize all vertices v as unmarked.
    - For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.

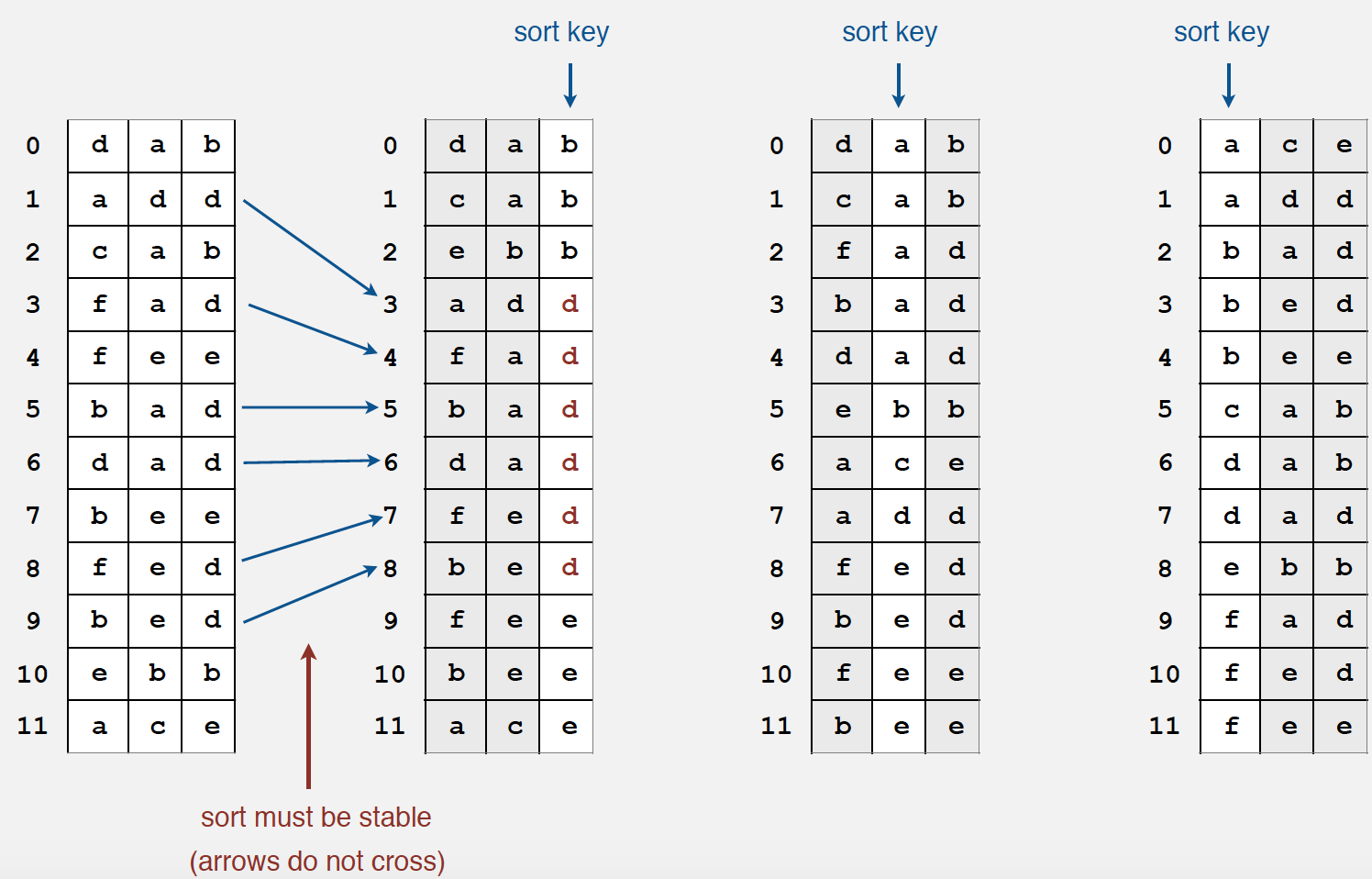
Directed Graphs

* Digraph
* Topological sort
  + Redraw so all edges point upward

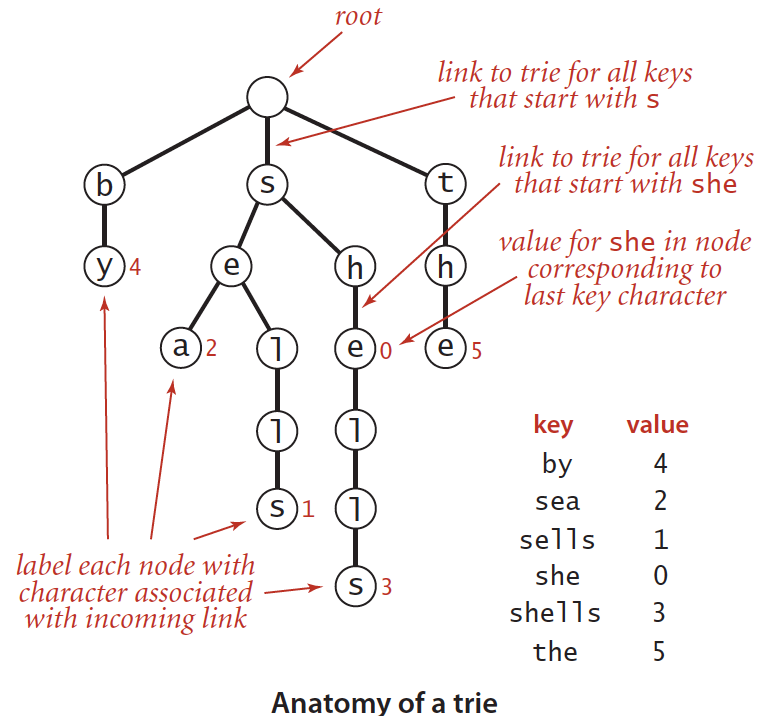
Minimum Spanning Trees

* Edge-weighted graph
  + Maintain vertex-indexed array of Edge lists (use Bag abstraction).
  + Implementation
    - Constructor: create an empty graph with V vertices (or from input stream)
    - void addEdge(Edge e): add weighted edge e
    - Iterable<Edge> adj(int v): edges incident to v
    - Iterable<Edge> edges(): all of this graph's edges
    - int V(): return number of vertices
    - int E(): return number of edges
    - String toString(): string representation
  + MST of edge-weighted graph is spanning tree whose weight is no larger than the weight of any other spanning tree
* Greedy algorithm
* Kruskal’s algorithm
  + Builds tree through taking the smallest edges as long as a cycle is not created
  + Start at smallest weighted edge and take next smallest edge (as long as cycle no created)
  + This will begin creating a forest of trees, which, given that all vertices are connected, will eventually be connected into a single MST.
* Prim’s aBuilds tree by starting at given node and adding edges with smallest weight as long as they do not create a cycle
* lgorithm
  + Start at given node and take smallest weighted edges (as long as cycle not created)
* Dijkstra’s algorithm
  + (TL;DR watch video): <https://www.youtube.com/watch?v=gdmfOwyQlcI>
  + Helps find shortest path from one node to another, based on weighted edges

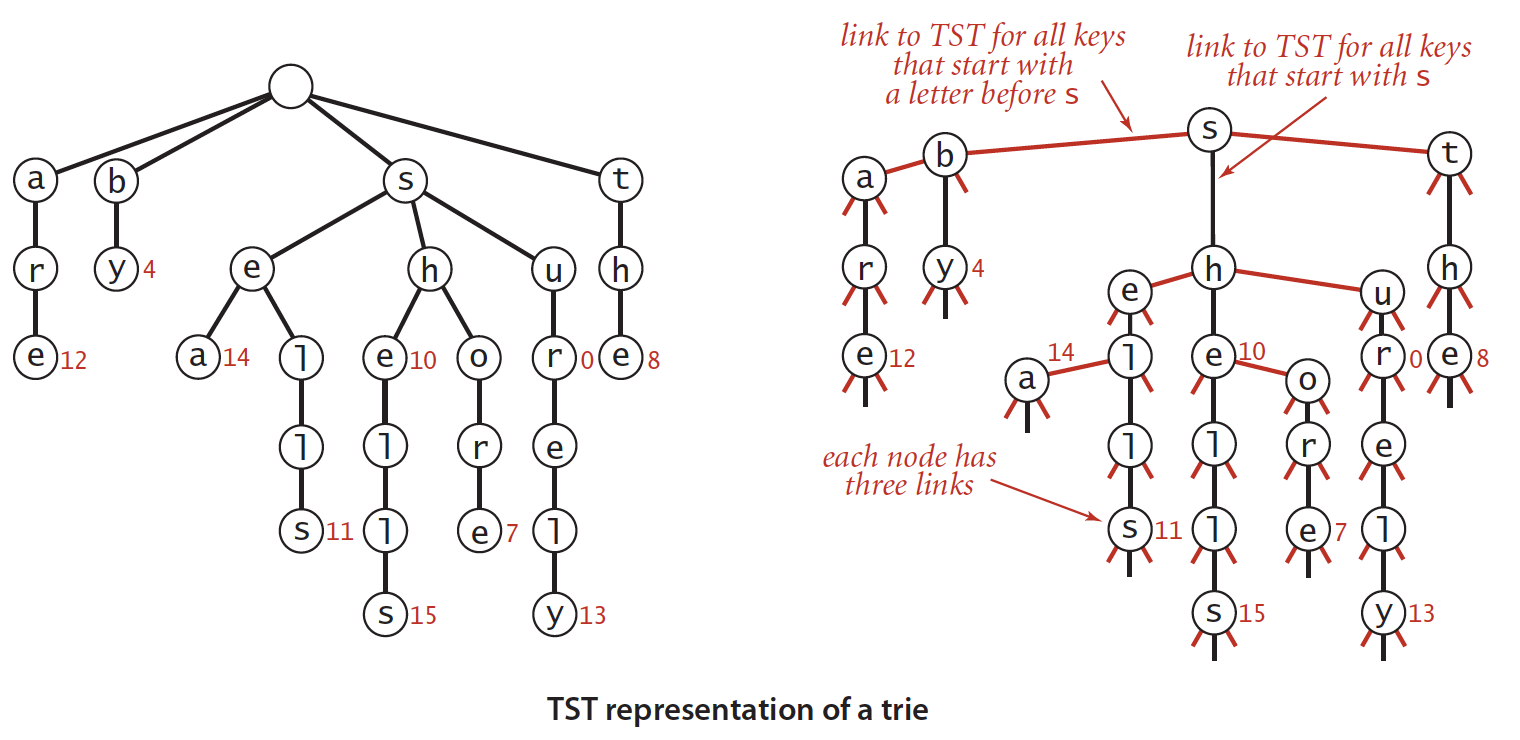
String sorts

* Key-indexed counting
  + Effective when the keys are small integers
  + Compute frequency counts
  + Transform counts to indices
  + Distribute the records
* LSD string sort
  + Consider characters from right to left.
  + Stably sort using dth character as the key (using key-indexed counting)
  + 
  + Useful for fixed-length fields
* MSD string sort
  + Same idea as string sort but consider characters from left to right
  + Useful for variable-length fields
* 3-way string quicksort

Tries

* R-way tries
  + 
  + Get(*string*)
    - Search hit: node where search ends has a non-null value
      * Ex: “shells” -> 3, “she” -> 0
    - Search miss: reach a null link or node where search ends has null value

Ex: “shell” -> null (no value), “shore” -> null (not found)

* + Put(*string*, *value*)
    - Encounter a null link: create new node.
    - Encounter the last character of the key: set value in that node.
    - Replace original value with new value if duplicate string
* Ternary search tries
  + “Ter-” = 3, so 3 children per node: smaller (left), equal (middle), larger (right).
  + Store characters and values in nodes
  + Trie to ternary search
  + 
  + Get(*string*): same for trie: “sea” -> 14
* String symbol table API

Data Compression

* Basics
* run-length coding
* Huffman compression
  + Implemented with a binary tree
  + Keys(data) always have two null leaves
  + Tree constructed using frequencies of chars
  + Decoded using preorder traversal (Root, Left, Right)
  + Fixed-length symbols with variable-length codes
  + Compress
    - Gather frequency of every possible char, insert chars into binary tree based on frequency, take each node’s value from tree, and write values to file
  + Decompress
    - Follow the binary tree and find the first matched result to decode each character
* LZW compression
  + Compress
    - Find the longest string in symbol table with prefix of unscanned input
    - Write 8-bit codeword associated with string
    - Scan 1 char past string in input
    - Associate next codeword value with char appended to string in the symbol table (char is next in input)
  + Decompress
    - Write current string value
    - Read a codeword from the input
    - Set string to value associated with codeword in symbol table
    - Associate the next unassigned codeword value to char appended to value in the symbol table (char is the first char of string)
    - Set current string value to string
  + Variable-length symbols with fixed-length codes