Finals Review

Data Structures and Algorithms (Concordia University)

ANALYSIS OF ALGORITHMS

- Pseudocode: Algorithm myAlgorithm(n)
  Input:
  Output:
  Start:
- $O(1) <= O(\log n) <= O(n) <= O(n \log n) <= O(n^2) <= O(2^n) <= O(n!)$
- $f(n)$ is $O(g(n))$ if there is a c and n_0 such that $f(n) <= cg(n)$ for $n >= n_0$
- Sum of (1 + 2 + 3 + ... + n) --> n(n+1)/2

  - Properties of powers:
    $a^{(b+c)} = a^b a^c$
    $a^{bc} = (a^b)^c$
    $a^b / a^c = a^{(b-c)}$
    $b = a^{\log_a b}$
    $b^c = a^{c*\log_a b}$
  - Properties of logarithms:
    $\log_b(xy) = \log_b x + \log_b y$
    $\log_b (x/y) = \log_b x - \log_b y$
    $\log_b xa = a\log_b x$
    $\log_b a = \log_x a / \log_x b$
-
- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  big-Oh
  - $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

  big-Omega
  - $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$

  big-Theta
  - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$
-

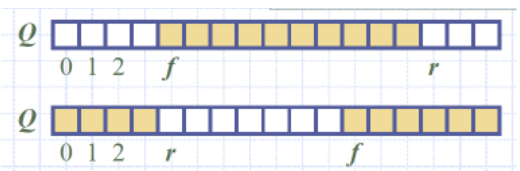
STACKS

- Last In First Out (FIFO)
- push(object): inserts element on top of stack
- pop(): removes and returns last element (null returned if none)
- top(): returns last inserted element without removing (null returned if none)
- size(): returns size of stack
- Space used is O(n)

- For arithmetic operations:
    - You need 2 stacks (one for values, one for operations)
    - Rule is that new inserted operation must be of higher precedence than the one before it (not lower or equal)

QUEUES

- First In First Out
- enqueue(object): inserts element at end of queue
- dequeue(object): removes and returns element at front of queue (returns null if empty)
- first(): returns first element without removing (returns null if empty)
- size(): returns size of queue
- queues have a front (f) and rear (r)



LISTS AND ITERATORS

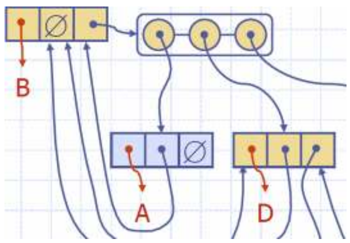
- DYNAMIC LIST:
    - implemented with an array
    - get(i): returns element at index i
    - set(i,e): sets element at index i with e and returns old element
    - add(i,e): add element e at index i, shifting the rest towards the right
    - remove(i): removes and returns element at index i, shifting the rest towards the left
    - Time complexity of add(i,e) and remove(i,e) is O(n)
    - Space used by dynamic list is O(n)
    - push(e): adds element e at the end of list; if full, create a larger array
        - Incremental strategy (increase by constant c): amortized time O(n)
        - Doubling strategy (double size of array): amortized time O(1)
- POSITIONAL LIST:
    - implemented with a doubly-linked list
    - p.getElement(): returns element stored at position p
    - first(): returns position of first element (or null if empty)

    - last(): returns position of last element (or null if empty)
    - before(p): returns position right before position p (null if p is first position)
    - after(p): returns position right after position p (null if p is last position)
    - addFirst(e): add element first and returns position of new element
    - addLast(e): add element last and returns position of new element
    - addBefore(p, e): inserts element before position p and returns position of new element
    - addAfter(p, e): inserts element after position p and returns position of new element
    - set(p, e): replaces element at position p, returns old element
    - remove(p): removes and returns element at position p

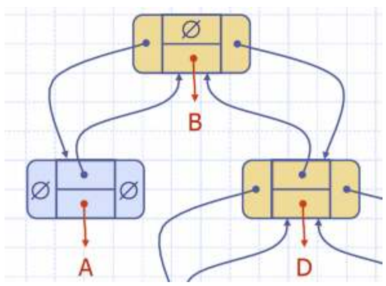
TREES

- Root: node without parent
- Internal node: node with at least one child
- External node: node without children
- Depth of node: number of ancestors
- Height of tree: maximum depth of any node
- Traversals of a tree:
    - Preorder: ROOT – LEFT – RIGHT
    - Postorder: LEFT – RIGHT – ROOT
    - Inorder: LEFT – ROOT – RIGHT (two algorithms below)
- Binary tree:
    - Each internal node has at most two children (exactly two for PROPER binary trees)
    - Recursive definition: tree with a single node, or a tree whose root has an ordered pair of children, each of which is a binary tree
    - Arithmetic operations: internal nodes are for operators and external nodes are for operands
    - Decision process: internal nodes are questions and external nodes are answers

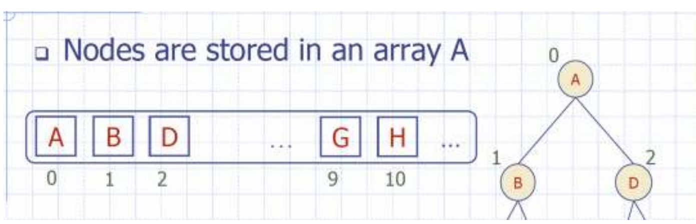
- Linked structures for trees
    - a node has
        - the element
        - the parent node
        - the sequence of children



- Linked structure for binary trees:
    - a node has
        - the element
        - the parent node
        - the left child
        - the right child



- Arrays for binary trees
    - rank(root) = 0
    - left child of parent is rank(node) = 2 * rank(parent) + 1
    - right child of parent is rank(node) = 2 * rank(parent) + 2
    - parent --> floor(i/2)



PRIORITY QUEUES

- each entry is a pair (key, value)
- insert(k,v): insert entry with key k and value v
- removeMin(): removes and returns entry with smallest key, null if PQ is empty
- min(): returns but doesn't remove key with smallest key, null if PQ is empty
- Unsorted list:
  - insert takes O(1), inserts at beginning or end of list
  - removeMin and min takes O(n), traverses whole list to find smallest key
- Sorted list:
  - insert takes O(n), traverses whole list to find where to insert item
  - removeMin and min takes O(1), smallest key is at beginning
- Priority Queue sorting:
  - inserts elements one by one with insert
  - removes elements in sorted order with removeMin
- Selection sort (unsorted sequence):
  - insert elements with insert --> O(n)
  - removing elements with removeMin takes 1 + 2 + 3 + ... + n
  - time complexity is $O(n^2)$
- Insertion sort (sorted sequence):
  - Insert elements with insert --> 1 + 2 + 3 + ... n
  - remove elements with removeMin takes O(n)
  - time complexity is $O(n^2)$
- In-place insertion sort:
  - keep sorted initial portion of sequence
  - use swaps

HEAPS

- binary tree storing keys at its nodes
- rules: except for root, key(v) >= key(parent(v))
- last node of a heap is the right most node of the maximum depth
- height of heap: a heap storing n keys has height O(log n)
- upheap runs in O(log n) and swaps the targeted key k in an upward path until it reaches the root or a node whose parent has a smaller or equal key to k
- Downheap runs in O(log n) and swaps the key k at the root in a downward path until it reaches a leaf or a node whose children have higher or equal keys to k

- Traversing a heap completely takes O(log n) time
- Heap sort:
  - space used is O(n)
  - insert and removeMin takes O(log n) time
  - Time complexity is O(n log n)
- Array-based heap: for node at rank i...
  - left child is 2i + 1
  - right child is 2i + 2
- Merging two heaps:
  - take two heaps and a key k
  - create a new heap with the key k as the rood and the two heaps as subtrees
  - downheap to restore heap-order property
- bottom-up heap construction runs in O(n) time

MAPS

- searching, inserting and deleting items (key-value entries)
- multiple entries with same key are NOT allowed
- get(k): if map has entry with key k, return the value, otherwise null
- put(k,v): put entry (k,v); if key DNE, return null, else return old value associated with key and replaces the key value
- remove(k): remove entry with key k and return the value
- Can be implemented using an unsorted list --> doubly-linked list
  - put(k,v) takes O(1)
  - get(k) and remove(k) takes O(n) because it has to traverse the entire sequence

HASH TABLES

- hash function h maps keys of a given type to an integer in a fixed interval [0, N – 1]
- a hash table for a given key type consists of:
  - hash function
  - array (called table) of size N
- hash function is usually the composition of two functions:
  - hash code = h_1: keys --> integers
  - compression function = h_2: integers --> [0, N – 1]

- Collision: when different elements are mapped to the same cell
- Ways to deal with collision (Open addressing: the colliding item is placed in a different table cell)
    - Separate chaining: every cell is a linked list (like dictionary)
    - Linear probing:
        - place the colliding item to the next available table cell (probe)
        - to handle insertions and deletions, we need an object called DEFUNCT
    - Open addressing: the colliding item is placed in a different table cell
    - Double hashing: uses a secondary hash function to place the item in the available cell (the table size N must be prime, CANNOT HAVE ZERO VALUES)
- insertion and remove takes O(n) time

BINARY SEARCH TREES

- Binary search terminates after O(log n)
- Search tables take O(log n) time
- Insert and remove takes O(n)
- to search a binary tree in increasing order, do INORDER TRAVERSAL
- property: key(left) <= key (root) <= key(right)
- external nodes DO NOT STORE items
- to search a key, we start from root and go downwards until leaf
- to insert, we search for key k; if not found, insert a node w at leaf and insert k into it, and expand w into an internal node (it will have two empty child nodes)
- deletion: if we delete a key k, make sure the array stays the same following the inorder traversal
- Space used is O(n)
- get, put, remove takes O(h)
- h is O(n) in worst case and O(log n) in best case

AVL TREES

- AVL trees are balanced
- Binary search tree that for every internal node v, the heights of the children of v can differ at most by 1
- height of an AVL tree is O(log n)
- Insertion is done as in a binary tree; by expanding an external node
- RESTRUCTURING (SINGLE ROTATIONS/DOUBLE ROTATIONS)
- Searching, insertion and removing take O(log n)
- Space is O(n)

MERGE SORT

- Divide-and-conquer algorithm (general):
    - Divide input data into two subsets
    - Solve two subsets
    - Combine two solutions of subsets
- Time complexity is O(n log n)
- Steps of Merge Sort:
    - Divide S into two sub-sequences
    - Recursively sort both sub-sequences
    - Merge both sub-sequences into one sequence
- Merging two sub-sequences takes O(n) time
- Height h of merge-sort tree is O(log n)

SUMMARY OF SORTING ALGORITHMS

Summary of Sorting Algorithms

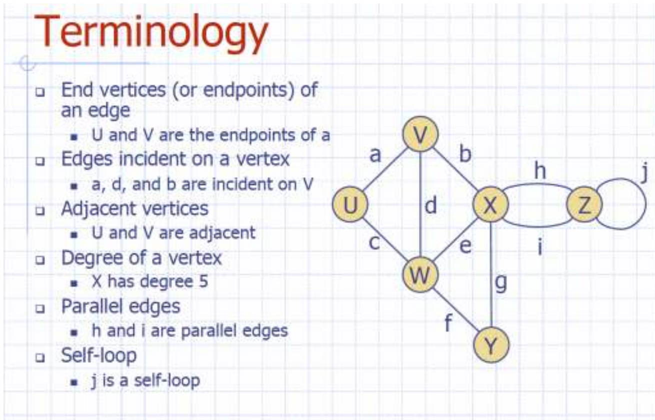
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

QUICK SORT

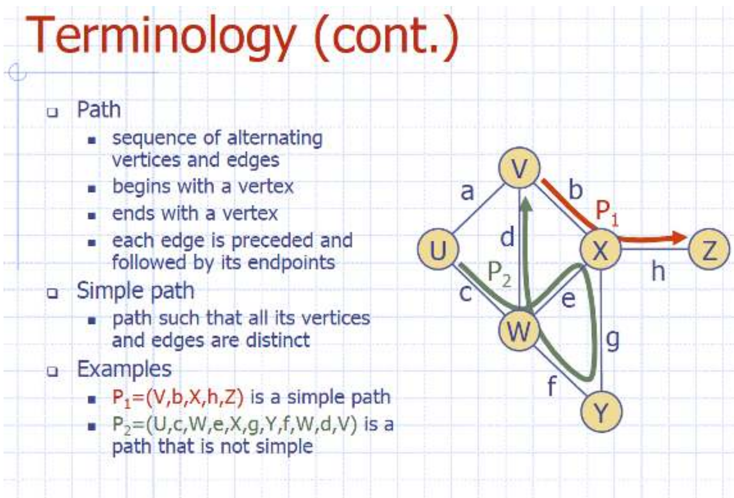
- randomized sorting algorithm
- steps:
  - Divide --> O(n) time: pick a random element x (the pivot) and divide S into three parts:
    - L elements less than x
    - E elements equal to x (stays in the same node as the element x)
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E, G
- Initial call is root
- Time complexity is $O(n^2)$
- Optimal: Sizes of L and G are each less than 3s/4
- Not optimal: one of sizes of L and G is more than 3s/4
- expected to be O(n log n)

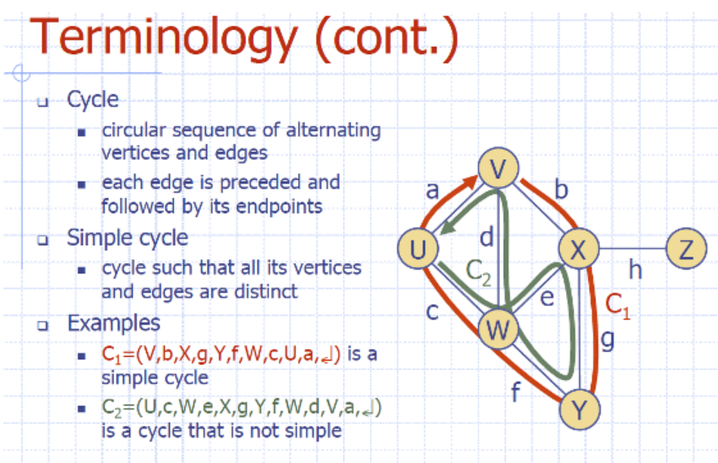
GRAPHS

- Graph is a pair (V,E) where V is a set of nodes called VERTICES and E is a collection of pairs of vertices called EDGES
- Types of edges:
  - Directed edge:
    - ordered pair of vertices (u,v)
    - first vertex u is the origin
    - second vertex v is the destination
  - Undirected edge:
    - unordered pair of vertices (u,v)
- Types of graphs
  - Directed graph:
    - all the edges are directed
  - Undirected graph:
    - all edges are undirected

- 

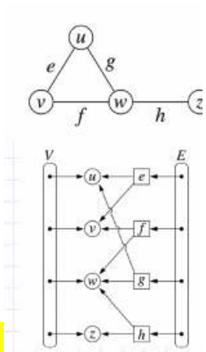
DEGREE OF VERTEX = # of edges incident to the vertex

- 

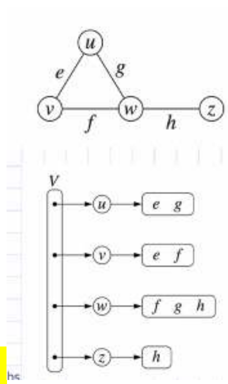
- 

-
- Notation:
  - n is number of vertices
  - m is number of edges
  - deg(v) is degree of vertex v

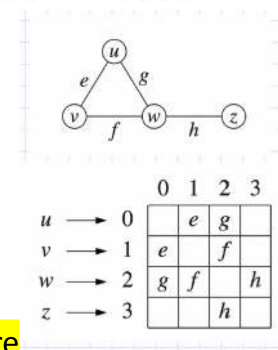
- Properties:
    - $\sum deg(v) = 2m$
    - In an undirected graph with no self-loops/no multiple edges: $m <= n(n-1)/2$
- A graph is a collection of vertices and edges
- A vertex is an object that stores an arbitrary element
- An edge stores an associated object, retrieved with element() method



- Edge list structure?



- Adjacency list structure



- Adjacency Matrix Structure

Performance

• n vertices, m edges • no parallel edges • no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

-

DFS (DEPTH-FIRST SEARCH)

- A subgraph S of a graph G is a graph such that:
  - the vertices of S are a subset of the vertices of G
  - the edges of S are a subset of the edges of G
  - A spanning subgraph is a graph that contains all the vertices of G
- A graph is connected if there is a path between every pair of vertices
- A connected component is a maximal connected subgraph of G
- A tree is an undirected graph T such that:
  - T is connected
  - T has no cycles
- A forest is an undirected graph without cycles (the connected components of a forest are trees)
- A spanning tree is not unique unless the graph is a tree
- DFS is a general technique to traverse a graph
  - visit all vertices and edges of G
  - determines whether G is connected
  - computes the connected components of G
  - computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes O(n + m) time
- Properties:
  - DFS(G,v) visits all the vertices and edges in the connected components of v

  - The discovery edges of DFS(G,v) form a spanning tree of the connected component of v
- Each vertex is labeled twice:
  - UNEXPLORED and VISITED
- Each edge is labeled twice:
  - UNEXPLORED and DISCOVERY/BACK
- Path finding: use a stack, add the vertices visited, when the destination vertex is reached, just print out contents of stack
- Cycle finding: use a stack, add the path to stack, when back edge is encountered, return the stack

BFS (BREADTH-FIRST SEARCH)

- BFS traversal:
  - visits all vertices and edges of G
  - determines whether G is connected
  - computes connected components of G
  - computes a spanning forest of G
- Takes O(n + m) time
- Properties:
  - BFS(G,v) visits all the vertices and edges of the connected components of s
  - The discovery edges labeled by BFS(G,v) form a spanning tree
  - For every vertex v in L
- How it works: traverses every node's children (PER LEVEL) until it finds unvisited node

SHORTEST PATHS

- A weighted graph is a graph where each edge has an associated numeral value (weight of the edge)
- Finding the shortest path between vertices u and v is to find the minimum total weight between u and v (length of a path is the sum of the weights)
- Properties:
  - a subpath of the shortest path is itself a shortest path
  - there is a tree of shortest paths from a start vertex to all the other vertices

    - tree of shortest paths from providence

- HAVE UNVISITED NODES AND UPDATE VALUES OF THE NODES

MST (MINIMUM SPANNING TREES)

- Create a list of visited nodes (that is empty)
- pick an arbitrary node
- from the visited nodes, pick the edge with the smallest weight and add the node to the list
- add the weight of the edges in the MST to find the MST's total edge weight