CSE100 Final Review

This review doc aims to provide a concise summary of the concepts, algorithms and lecture slides covered in CSE110. *Created by M.*, feel free to collaborate.

Midterm 1 Topics

- Time and space complexity
- C++ (compare to java, classes, reference, const, vector, I/O, template)
- Big O: upper
- Big Theta: tight
- Big Omega: lower
- Tree
- Binary search Tree (Average-case time complexity computation)
- Iterators(pointers)
- KD-Tree(Insertion: left<curr<=right)
- Multi-way Tree
- Ternary Search Tree

```
int a = 5;                     // create a regular int
int b = 6;                     // create a regular int
const int * ptr1 = &a;         // can change what ptr1 points to, but can't modify the actual data pointed to
int const * ptr2 = &a;         // equivalent to ptr1
int * const ptr3 = &a;         // can modify the data pointed to, but can't change what ptr3 points to
const int * const ptr4 = &a; // can't change what ptr2 points to AND can't modify the actual object itself

ptr1 = &b;                     // valid, because I CAN change what ptr1 points to
*ptr1 = 7;                      // NOT valid, because I CAN'T modify the data pointed to

*ptr3 = 7;                     // valid, because I CAN modify the data pointed to
ptr3 = &b;                     // NOT valid, because I CAN'T change what ptr3 points to

ptr4 = &b;                     // NOT valid, because I CAN'T change what ptr4 points to
*ptr4 = 7;                     // NOT valid, because I can't modify the the data pointed to
```

With **references**, the `const` keyword isn't too complicated. Basically, it prevents modifying the data being referenced via the `const` reference. Below are some examples with explanations:

```
int a = 5;             // create a regular int
const int b = 6;       // create a const int

const int & ref1 = a; // creates a const reference to 'a' (can't modify the value of a using ref1)
                       // This is OK.  Even though a is allowed to change, it's OK to have a reference that
                       // does not allow you to change it because nothing unexpected will happen.
int const & ref2 = a; // equivalent to ref1

ref1 = 7;              // NOT valid, because ref1 can't modify the data

const int & ref3 = b; // valid, because you can have const reference to const data
int & ref4 = b;        // NOT valid, because you CAN'T have a non-const reference to const data
                       // ref4 might change the data but b says it shouldn't be changed, which is unexpected

int & const ref5 = a; // invalid syntax (const must come before the & symbol)
```

- **Traversal orders**:
    - Pre-order traversal - Root, Left, Right
    - In-order traversal - Left, Root, Right
    - Post-order traversal - Left, Right, Root
- **Depth First Search**: search as far down a single path as possible before backtracking

- Pseudo code (iterative) - LIFO
```
DFS(start):
    Initialize stack
    Push start onto the stack
    while stack is not empty:
        pop node curr from top of stack
        visit curr
        for each of curr's children, n
            push n onto the stack
// Done
```
- Pseudo code (recursive) -
```
DFS(start):
    for each of start's children, n:
        visit n
        DFS(n)
```

- **<u>Breadth First Search</u>**: visit children first before searching down a path
  - Pseudo code (iterative) - FIFO
```
BFS(start):
    Initialize queue
    Push start onto the queue
    while queue is not empty:
        pop node curr from front of queue
        visit curr
        for each of curr's children, n
            push n onto the queue
// Done
```
- **Multiway-Trie**
  - Pros
    - O(D), where D is length of the longest string.(Better than BST which has O(DlogC), C is the number of characters
  - Cons
    - Creating an extra node(edge) for each character in the structure; wastes a lot of space
  - Important Notes:
    - The structure of a trie does *not* depend on the insertion order (unlike a BST)
  -

<u>Midterm 2</u>

- Hashing, properties of hash functions, hash table, optimization based on probability theory, collision resolution strategies(linear, double, random, separate chaining, cuckoo), hash map(map abstract data type)

- Markov text generation

- AVL tree; single/double/edge case rotation, runtime complexity proof;

- Red-black Tree; general structure, coloring, insertion in different cases, insertion, removal

- Graph; types of graphs, representation(adjacency matrix, adjacency list)

- Encoding with graph: huffman algorithm(huffman total length<fixed length coding)

- Graph Search; BFS, BST's queue ADT implementation; DFS, DFS's stack implementation; runtime: **O(|V| + |E|).**

- Shortest path; Dijkstra's algorithm on weighted and unweighted, runtime **O(|V| + |E| log |E|)**, A* graph search (modification on priority function: f(n) = g(n) + h(n), best estimate cost)

==

Final:

==

- Disjoint set
- P, NP complete, NP hard
- **KD Tree - (pa2 point comparison)**
    - Multidimensional search tree, each level uses different dimensions interchangeably. Worst-case runtime $O(k*n^{(1 - 1/i)})$
    - Nearest neighbor
        - Start from the **root**, stored as **closest point and shortest distance**
        - Recursively go left or right based on dimension, to a leaf
        - Store every point's distance from queried point
        - Compare current distance with shortest distance, update if necessary
        - **Check the other subtree when climbing up (single dimension check)**

- **Binary Search Tree**
    - Space complexity O(n)
    - In-order traversal - Left, Root, Right (ordered)
    - Successor
        - Case 1: go right once, all the way left
        - Case 2: traverse up, find a node is a left child of its parent, then its parent will be the successor
    - Runtime:

        - Assumption: 1. All keys are equally likely to be searched; 2. All insertion orders are equally likely.
        - Find average # of comparisons needed to find an element in specific BST; then average this value over all possible BSTs with N nodes.
- **Multiway Trie**
    - Worst find depends on the length of the query **word - O(k)**
    - Store characters by index (eg: ‘c’ is represented by ‘c’ - ‘a’)
    - Remove
        - Simply set **word** to false
        - If not found in MWT, do nothing
    - Fields
        - **Word -** indicates if current node is a word
        - **Node * nexts[n] -** an array of size of the possible number of characters
    - Don’t depend on the order we insert elements

- **Ternary Search Tree (TST)**
    - Find algorithm
        - **If** *letter* < *node*'s label:
            - **If** left, traverse down to *left*. Otherwise, **failed**
        - **If** *letter* > *node*'s label:
            - **If** *right*, traverse down to *right*. Otherwise, **failed.**
        - **If** *letter* = *node*'s label:
            - **If** last letter of *key* and if *node* is labeled as a "word”, **found.**
            - **If not, failed.**
            - **Otherwise**, if *middle*, traverse down to *node*'s middle child and set *letter* to the next character of *key*; if not, we have failed (*key* does not exist in this Ternary Search Tree)
    -
- **Hashing**
    - Faster find array[i] at **O(1);** Size is **Prime**
    - Hash function
        Key + hash function = hash value;
        Necessary: **if k = l, then h(k) = h(l)**; Nice to have: if k != l, then h(k) != h(l)
    - Hash Table: array of size M
        Average complexity = O(1), independent of # elements it stored.
        Impossible to iterator in order: **unordered set**.
    - Optimization
        Reasonable capacity (load factor), good indexing (hash function).

Probability theory: P(no collisions) = 1 * (m-1)/m * (m-2)/m …. (m-n-1)/m; expected collisions: **M ~= 1.3N**.
Load factor: performance depend on load factor, not number of elements; insertion is bad when load factor > 70%.

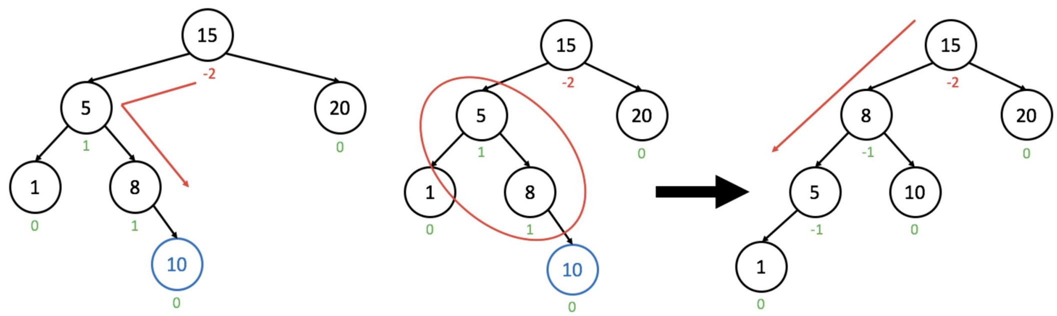
- **Resize** - double to the nearest **prime**

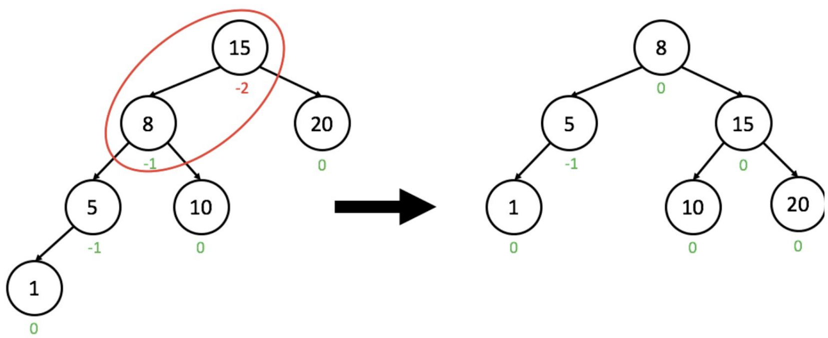
- **Resolution of collision**
  - Linear probing: Best O(1), Worst O(n)
    - If occupied, go to next slot: index = (index + 1) % size
    - **Open addressing** (the keys are open to move to an address other than the address to which they initially hashed.)
    - Remove
      - **delete** flag needed, cannot actually remove element
    - Pros - when not full, average performance **O(1)**
    - Cons
      - **clumps**
      - Unequal probability of getting filled
  - Double Hashing
    - H_1 to calculate the index, H_2 to calculate the offset
    - Next Index = (Index + offset) % size
  - Random Hashing
    - Key as seed, generate a sequence of index
  - Separate Chaining: Best O(1); Worst O(n)(for find and remove in worst case, insert still O(1))
    - Keep pointers as linked lists - new key to the **front**
    - **Closed addressing** (the key *must* be located in the original address)
    - Pros:
      - In general, the average-case performance is considered much better than **Linear Probing** and **Double Hashing** as the amount of keys approaches, and even *exceeds*, *M*, the capacity of the **Hash Table**
    - Cons:
      - Requires extra space for pointers
      - All the data in our **Hash Table** is no longer huddled near 1 memory location (since pointers can point to memory anywhere), and as a result, this poor locality causes poor cache performance (i.e., it often takes the computer longer to find data that isn't located near previously accessed data)
  - Cuckoo Hashing: Best O(1); Worst O(1)
    - **Two hash functions, both return a position in Hash table**
    - Two hash tables **(d ≥ 2) (size = M / d)**
      - If collision, kick out item in T_1, insert the kicked item into T_2
      - **Infinite cycle - kicking out the same element**

        - find & delete
            - Need to check two tables
    - Bloom Filter
        - If words are real, give true; if words are not real, might be wrong
        - Fill the bit
            - Multiple hash functions, fill every bit at all hash values
        - When checking if in the table
            - Check corresponding indice
            - **Might be wrong** - slot filled by other words
    - Count-min - sacrificing exact solution to solve heavy hitters
        - Heavy Hitters: Given a set of elements of size n, some much smaller value k, determine which elements occur at least **n/k** times.
        - Multiple hash functions
        - Find the **min** occurrence among the values at corresponding indice

- **Markov Text Generation**
    - Train: build model based on data, for each new word keep track of next words, calculate probability
    - Generate: find the current word, generate next word based on probability

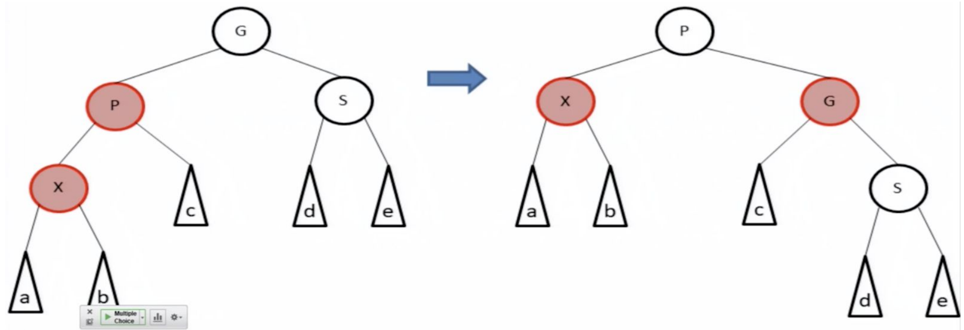
- **AVL Tree**
    - a **Binary Search Tree** in which, for all nodes in the tree, the *heights* of the two child subtrees of the node differ by at most one
    - **balance factor -**
        - **right_height - left_height**
        - Is etheri 1, 0, -1
    - Time complexity
        - **O(log(n))**
    - To keep balanced → **Rotation**
        - **Single rotation on the node out of balance**
            - Check if has two child - If yes, connect children
        - **Double rotation**





- **Red-Black Tree**
    - Properties
        - All nodes must be "colored" either **red** or **black**
        - **root** must be **black**
        - If a node is **red**, all of its children must be **black**
        - For any given node *u*, every possible path from *u* to a "null reference" (i.e., an empty left or right child) must contain the same number of **black** nodes
    - Null reference is also colored **black**
    - *some* **Red-Black Trees** are *also* **AVL Trees**, *not all* **Red-Black Trees** are **AVL Trees**.
    - Height
        - At most 2 times of the height of AVL Tree with the same amount of elements. (consider remove all red nodes, then all black nodes same height)
        - **O(log(n))**
    - To keep balanced - insertion
        - **Recoloring - (primary - top black, children red)**
            - Newly inserted node marked as **red**
            - If newly inserted node is a child of **black node, we are done!**
            - If newly inserted node is a child of **red node**
                - **If nodes are straight line**
                    - **single rotation**
                    - **recoloring**

        - **Else**
            - **Double rotation**
            - **Recoloring**
    - **Parent's sibling is Red**
        - As going down the tree, check nodes' two children
        - Recoloring before insertion
            - Once found two siblings are red, recolor and rotate



If X's Parent (P) is red, P is a left child of G, X is a left child of P, (and P's sibling (S) is black), then Rotate P right, flip colors of P and G

- **AVL vs Red-Black**
    - AVL Trees perform better with find operation
    - RB Trees perform better with insert and remove operations.
    - Red-Black Trees are the Binary Search Tree of choice for ordered data structures in many programming languages (e.g. the C++ map or set)
    - Red-Black Tree
        - out of balance, take roughly the same amount of time to **remove or insert** an element in comparison to the corresponding AVL Tree (around 2 log *n* vs. around 2 log *n* operations).
        - pretty balanced, take roughly ***half*** the time to **remove or insert** an element in comparison to the corresponding AVL Tree (around log *n* vs. around 2 log *n* operations).

- **Worst-Case Time Complexity (AVL Tree)**
    - Find: **O(log *n*)** — AVL Trees must be balanced by definition
    - Insert: **O(log *n*)** — AVL Trees must be balanced by definition, and the rebalancing is O(log *n*)
    - Remove: **O(log *n*)** — AVL Trees must be balanced by definition, and the rebalancing is O(log *n*)
- **Average-Case Time Complexity (AVL Tree)**
    - Find: **O(log *n*)**
    - Insert: **O(log *n*)**
    - Remove: **O(log *n*)**

- **Best-Case Time Complexity (AVL Tree)**
    - Find: **O(1)** — If the query is the root
    - Insert: **O(log *n*)** — AVL Trees must be balanced, so we have to go down the entire O(log *n*) height of the tree
    - Remove: **O(log *n*)**
- **Space Complexity (AVL Tree)**
    - **O(*n*)** — Each node contains either 3 pointers (parent, left child, and right child) and the data, so O(1) space for each node, and we have exactly *n* nodes

- **Worst-Case Time Complexity (Red-Black Tree)**
    - Find: **O(log *n*)** — Red-Black Trees must be balanced
    - Insert: **O(log *n*)** — Red-Black Trees must be balanced, so we have to go down the entire O(log *n*) height of the tree, and the rebalancing occurs with O(1) cost during the insertion
    - Remove: **O(log *n*)** — Red-Black Trees must be balanced, and the rebalancing occurs with O(1) cost during the removal
- **Average-Case Time Complexity (Red-Black Tree)**
    - Find: **O(log *n*)**
    - Insert: **O(log *n*)**
    - Remove: **O(log *n*)**
- **Best-Case Time Complexity (Red-Black Tree)**
    - Find: **O(1)** — If the query is the root
    - Insert: **O(log *n*)** — Red-Black Trees must be balanced, so we have to go down the entire O(log *n*) height of the tree, and the rebalancing occurs with O(1) cost during the insertion
    - Remove: **O(log *n*)**
- **Space Complexity (Red-Black Tree)**
    - **O(*n*)** — Each node contains either 3 pointers (parent, left child, and right child) and the data, so O(1) space for each node, and we have exactly *n* nodes

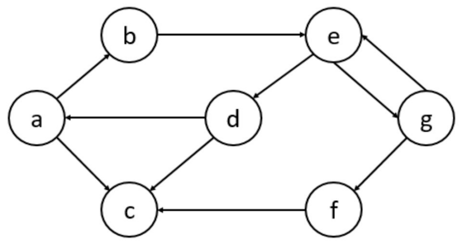
- **Huffman Encoding** - Tree Construction
    - Encoding depends on the **frequencies** of symbols
    - Construction:
        - Have a bunch of nodes (some subtrees that only have **root**)
        - Pick the least frequent nodes and make a **Tree**
            - **Mark left edge 1, right 0**
        - Repeat **step-2** till no subtrees exists
    - once have a tree, we can start encoding
    - Decryption

- Traverse down the tree by bits
- Lossless encoding(shorter bytes than fixed length)
- **Runtime analysis**
  - **File - O(n)**
  - **Use priority queue - O(clogc)**

- **Graph**
  - Nodes, edges
  - Unstructured, sequential, hierarchical, structured/connected/disconnected
  - Directed, undirected, weighted
  - Representation
    - **Adjacency Matrix**: coming from row i going to column j. **O(V^2)**

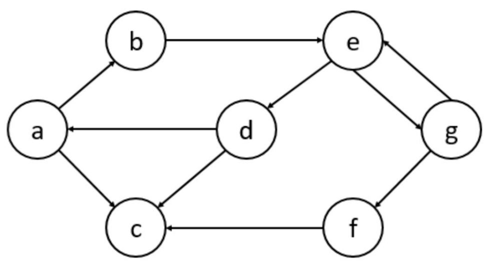
Image: https://ucarecdn.com/
85b37548-4042-42d5-8f67-a35788db286c/

	a	b	c	d	e	f	g
							0
b	0	0	0	0	1	0	0
c	0	0	0	0	0	0	0
d	1	0	1	0	0	0	0
e	0	0	0	1	0	0	1
f	0	0	1	0	0	0	0
g	0	0	0	0	1	1	0



- **Adjacency List:** smaller storage, O(V+E). More dense: O(V+V^2)

```
a: {b, c}
b: {e}
c: {}
d: {a, c}
e: {d, g}
f: {c}
g: {e, f}
```



- Storage - |V| + |E|

Graph Search

- BFS: Search all nodes on the same layer before continuing to next layer, **queue ADT(first in, first out)**
  - **Application: shortest path search (unweighted)**

```
BFS(s):
    initially, give all vertices in the graph a distance of INFINITY  O(|V|)
    start at s; give s distance = 0  O(1)
    enqueue s into a queue  O(1)
    while the queue is not empty:  O(|E|)
        dequeue the vertex v from the head of the queue  O(1)
        for each of v's adjacent nodes that have not yet been visited:  O(|E|)
            mark its distance as as 1 + distance to v  O(1)
            enqueue it in the queue  O(1)
```

-

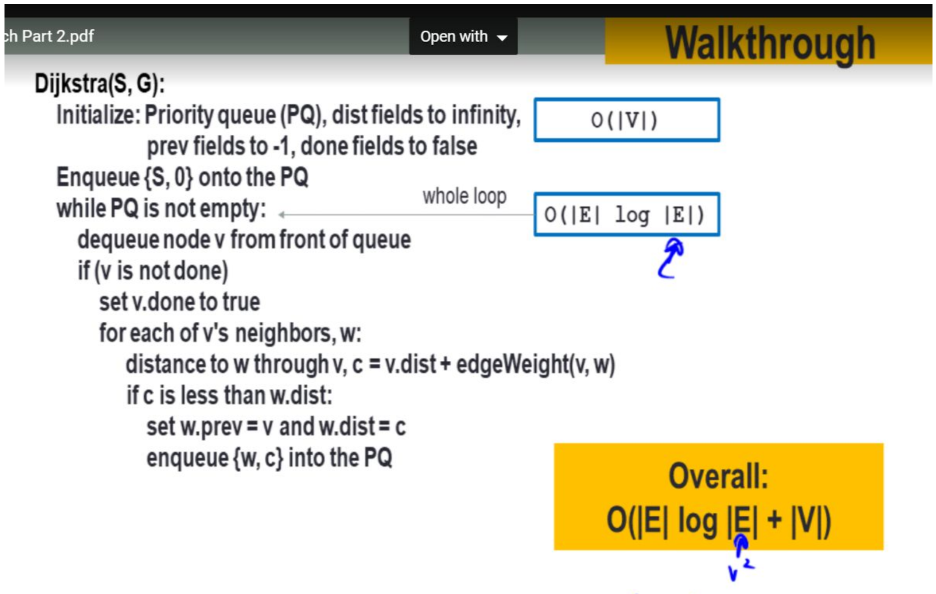
- DFS: Search from root to leaf, go all the way down. **Stack(first in, last out)**
  - **Runtime: O(V + E).**

```
DFS(u,v):
    s = an empty stack
    push (0,u) to s // (0,u) -> (length from u, current vertex)
    while s is not empty:
        (length,curr) = s.pop()
        if curr == v: // if we have reached the vertex we are searching for
            return length
        for all outgoing edges (curr,w) from curr: // otherwise explore all neighbors
            if w has not yet been visited:
                add (length+1,w) to s
    return "FAIL" // if we reach this point, then no path exists from u to v
```

  -

Shortest Path

- Dijkstra's Algorithm: (**BFS based**)
  - **no negative weight edges can exist in the graph**
  - lowest overall path weight. Search for next shortest and see which vertex it discovers.
  - Implementation: **Priority Queue** ADT, store one immediate vertex at the end of path
  - Fields of node: Distance, previous, done
  - Each vertex: first time removed, **shortest path found**
  - Runtime: O(V + **E log(E)**)
    - Insertion of priority queue - **O(ElogE)**

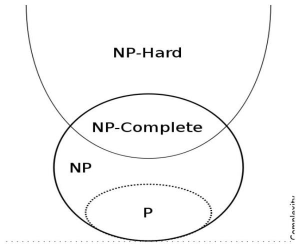


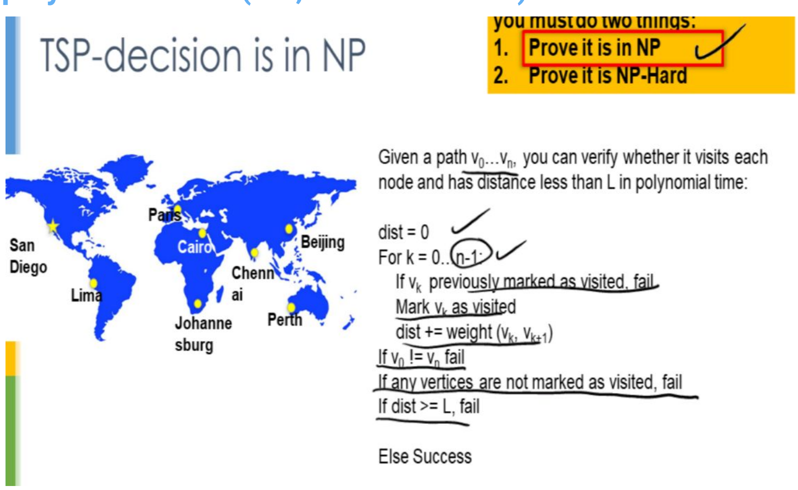
- A star search
  - **Priority function: f(n) = g(n) {distance from start to vertex n} + h(n) {heuristic estimated cost from vertex n to goal vertex}**

Minimum Spanning Tree Creation

- N-1 edges(N nodes)
- No cycle
- **Prim's Algorithm(similar to Dijkstra)**
  - Starting at a vertex
  - Enqueue every adjacent vertex
  - Dequeue the very first vertex and connect with its previous
  - Done when queue is empty
- **Kruskal's Algorithm**
  - Create a forest of vertices
  - Create a priority queue containing all the edges, ordered by edge weight (**not accumulative**)
  - While fewer than |*V*| - 1 edges have been added back to the forest:
    - Deque the smallest-weight edge (*v*,*w*, cost), from the priority queue
    - **If:** *v* and *w* already belong to the **same tree** in the forest, go to deque
    - **Else:** Join those vertices with that edge and continue
  - **O(|E| log|E|) - (logE due to heap insertion)**
- **Kruskal's Algorithm** and **Prim's Algorithm** can easily work with **negative edge**

P, NP, NP hard and NP complete:

- **Relationship as follows:**
- 

  P ≠ NP
- **P:**
  - solve in polynomial time, NP: prove in polynomial time
- **NP-complete**
  - No poly solution but can check a solution in poly time
- **NP-hard:**
  - hardest problem of NP - Prove to be **NP(TSP problem)**
  - A problem *H* is NP-Hard when every problem *L* in NP can be "reduced", or transformed, to problem *H* in polynomial time. As a result, if someone were to find a polynomial-time algorithm to solve any NP-Hard problem, this would give polynomial-time algorithms for all problems in NP
- **an NP-Hard problem is considered NP-Complete if it can be verified in polynomial time (i.e., it is also in NP).**
- 

Prove NP-hard(TSP problem-> using hamiltonian cycle->using reduction):

- **Disjoint set:**
  - **two operations:**

  - **Union -** merge two sets that have elements **u & v**
  - **Find -** return the set element **e** belongs
- **sentinel nodes**
  - Nodes that **do not** have "parent"
  - Represents a single set
- **Representation in array**
- eg:

Index 0	Index 1	Index 2	Index 3	Index 4
A	**B**	**C**	**D**	**E**
-1	**0**	**-1**	**-1**	**-1**

- **Union operation mechanism**
  - Weighted union: make the root larger set
  - Path compression
  - Perform find operation on each element to get sentinels
  - And **connect two sentinels**
    - Union by size
      - Sentinel of smaller set becomes the child
      - **Worst-case** find O(logn)
    - Union by height
      - Sentinel of smaller height set becomes the child
      - **Worst-case** find O(logn)
- **Path Compression :**
  - In find operation, **Re-attach** each node to sentinel along the way up
  - worst case O(1) after path compression for **all operations**

- **Greedy Algorithm**
  - Always select the locally largest step toward the goal
  - **Not always the optimal solution tho** (coins problem from CSE 20)