Lecture Notes

CS126 - Design of Information Structures

Analysis of algorithms

1. **Experimental**: write the program with variable input sizes, plot results against running time.
   - Necessary to implement algorithm, to compare results, same setup is needed.

2. **Theoretical**: Use high-level description of algorithm, express running time as function with input size n

3. Big O: $O(n)$: time $\leq n$
   Little o: $o(n)$: time $< n$
   Big Omega: $\Omega(n)$: time $\geq n$
   Little Omega: $\omega(n)$: time $> n$
   Big Theta: $\Theta(n)$: time $= n$

4. Relative **growth rate**: $\lim_{n\to\infty} \frac{f(n)}{g(n)}$
   $0 \to f(n) = O(g(n))$
   $\neq 0 \to f(n) = \Theta(g(n))$
   $\infty \to g(n) = O(f(n))$
   The limit oscillates: no relation here
   Can also compare log of $f(n)$ and $g(n)$

5. $\log_b x^a = a \log_b x$

6. Amortized time: Divide total time $T(n)$ needed to perform a series of n operations, and divide it by n to find the average time of an operation: $\frac{T(n)}{n}$.

Other

1. **Recursion** has base cases (must eventually be reached) and recursive calls (to itself). **Tail** recursion: linearly recursive method makes the recursive call as its last step. Power function $p(x,n) = x^n = \begin{cases} 1 \text{if} n = 0 \\ x \times p(x, n-1) \text{else} \end{cases}$. Fibonacci recursive:

```
BinaryFib(k) =
if (k=1) { return k}
else { return BinaryFib(k-1)+BinaryFib(k-2)
}
```

   Very slow ($O(2^n)$)

2. **Theorem** Any algorithm that sorts by exchanging adjacent elements requires $\Omega(n^2)$ average time. To perform subquadratic, must do comparisons and exchanges between distant elements.

3. **Theorem** No sorting algorithm based on key comparisons can possibly be faster than $\Omega(nlogn)$

4. **Divide-and-conquer** strategy is a paradigm: **Divide** input data S in two independent subproblems S_1, S_2, **Recursive**: solve subproblems of S_1, S_2 and **Conquer**: combine solutions for S_1, S_2 into a solution for S. Base case is a problem of size 0 or 1.

5. **Iterative Substitution**: iteratively apply recurrence equation to itself and see if we can find a pattern: if $T(n) = 2T(n/2)+bn$ then do: $= 2(2T(n/2^2))+b(n/2)+bn$ etc.. $= 2^iT(n/2^i) + ibn$

Graphs

1. Edges a, c are **incident** on vertex U if they a, b are joined by U (connected by a dot). Vertices U, V are adjacent if they're connected by an edge (connected by a line).

2. **Reachability problems**: Given start vertex s of G, for all vertices v, compute the shortest path between s and v. Test whether G is connected/strongly connected (for a digraph). Compute spanning tree of G (subgraph with all its vertices). Compute connected/strongly connected components of G. Identify a cycle in G.

3. Greedy algorithm picks the optimal choice at each step, without backtracking. Can use **Ksuskal's algorithm**: at each step: select edge with the lowest cost without yielding a cycle. Can detect cycles with **Union-find** data structure. Runs in $O(|E| \log |E|)$. In connected graph with non-negative weights, shortest paths exist for any pair of vertices, but may not be unique (also negative cost may cause the algorithm to get stuck in a cycle).

4. **Dijkstra's algorithm** uses BFS. Given a start node v, it explores all directly reachable nodes, marking the cost, and marks v as explored. Next, choose one of the reached nodes with minimum cost, and treat it like v, if a shorter path to a different node is found, then update the weights. Doesn't backtrack. Runs in O(m log n).

Binary Search

Best: Θ**(logn)** **Worst:** Θ**(logn)**

Given a **sorted** list L of elements, find a key k. Keep dividing the dataset in half, and setting whichever one contains the element to be the new dataset.

```
Leftmost l, and rightmost r indecies of L.
Find middle index m = (l+r)/2, compare to k
k = m → true
k < m → r = (m − 1)
k > m → l = (m + 1)
```

Generic Merge

Best: Θ**(n+m)** **Worst:** Θ**(n+m)**

Combine two sorted subarrays (left, right) into one sorted array.

```
i = 0, j = 0, k = 0
while (i < left.size() & j < right.size()) {
if left[i] ≤ right[j] { arr[k++] = left[i++] }
else { arr[k++] = right[j++] }
Once one of the arrays is ended, can just copy the remaining elements from the second one
```

Depth-First Search

Best: Θ**(n+m)** **Worst:** Θ**(n+m)**

Find if G is connected, compute its connected components and spanning, find a cycle or a path between vertices. Visits all vertices and edges in connected component of v, discovery edges form its spanning tree. Can detect **back edge** (connects node to its ancestor), **forward edge** (to its descendant) and **biconnected components** (**Stack**)

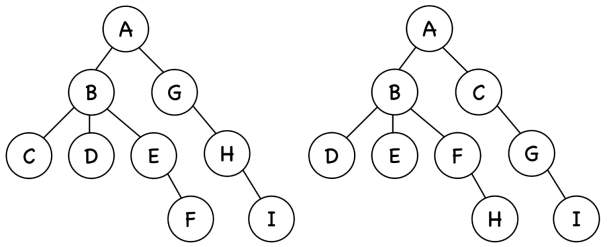
```
DFS(G, u) {   # Input:  Graph G, vertex u ∈ G
markVisited(u);
for (each of u's outoing edges, e=(u,v)) {
if (notVisited(v)) { markToBeVisited(v); DFS(G, v);}   # Add node's children to the stack
}}
```

Breadth-First Search

Best: Θ**(n+m)** **Worst:** Θ**(n+m)**

Find if G is connected, compute its connected components and spanning, find a cycle or a **shortest** path between vertices. Can detect a **cross edge** (connects two nodes on the same level, so not related). Discovery edges form a spanning tree T_s, path from root to node on level L_i has i edges in T_s and $\geq i$ in G_s (**Queue**)

```
BFS(G, u) {   # Input:  Graph G, vertex u ∈ G
markVisited(u);
for (each of u's outoing edges, e=(u,v)) {
if (notVisited(v)) { markToBeVisited(v); BFS(G, v);}   # Add node's children to the queue
}}
```



DFS uses a **Stack**

BFS uses a **Queue**

Insertion Sort

Best: $\Theta(\mathbf{n})$ **Worst:** $\Theta(\mathbf{n^2})$

Go through an array, and put every element in a correct position by comparing it to the element before.

```
for (i=1; i < arr.size(); i++) {
while i > 0, arr[i] > arr[i-1] { swap elements (arr[i], arr[i-1]); i--; }
}
```

Selection Sort

Best: $\Theta(\mathbf{n})$ **Worst:** $\Theta(\mathbf{n^2})$

Given a list of n elements, find the smallest one, and swap with the first element of the array, then find the second smallest in the subarray, and swap it with its first element and so on.

```
for (i = 0; i<size; i++) {
swap arr[i], min(arr[i:end])
}
```

Merge Sort

Best: $\Theta(\mathbf{n \cdot \log n})$ **Worst:** $\Theta(\mathbf{n \cdot \log n})$

For input sequence S with n elements: **Divide** S into sequences S_1, S_2 of around $\frac{n}{2}$ elements, **Recursively** sort S_1, S_2 and generic merge their solutions into a unique sorted sequence (**Conquer**), takes O(n) time.

```
mergeSort(S) { if S.size() > 1:
(S_1, S_2) = partitionInHalf(S)
mergeSort(S_1), mergeSort(S_2)
return merge(S_1, S_2)
}
```

Quick Sort

Best: $\Theta(\mathbf{n \cdot \log n})$ **Worst:** $\Theta(n^2)$

Use **divide-and-conquer**: given array A, divide into 2 subarrays L, R around element $q \in A$ (**pivot**) such that $\forall l \in L | x \leq q, \quad \forall r \in R | x > q$. Swap rightmost element and pivot, then whenever left pointer is $>$ than pivot, and right pointer $<$ then swap those two, repeat until left pointer index is $<$ right pointer index. Recursively sort L, R and then merge. Average runtime is $\Theta(n \cdot \log n)$. Pivot = median(first, middle, last) because random is too expensive, else may lead to worst case.

```
(L, R) = partitionInHalf(S); left = 0;  right = (size - 2);
pivot = median(arr[0], arr[size/2], arr[size-1]);
swap (arr[size-1], pivot)
while (left < right) {
while (arr[left] < pivot) {left++}
while (arr[right] > pivot) {right--}
if (left < right) { swap(arr[left], arr[right] }
} swap (pivot, original_pivot_index)
recursively split in half, and apply quicksort to each subsequence.
```

Bucket Sort

Best: $\Theta(\mathbf{n})$ **Worst:** $\Theta(\mathbf{n^2})$

For input sequence S with n elements: create n **buckets**, and put corresponding elements, into a bucket with an index of their value: buckets[arr[i]] = arr[i], value for index is optionally multiplied by a constant. Then sort every bucket using some other sorting algorithm. Bucket sort itself doesn't compare values. Entries follow FIFO order.

Stable Sort

Stable Sorting preserves the order of entries we wish to sort: assume 3' = 3, so stableSort(1, **3**, 4, **3'**) = (1, **3**, **3'**, 4)

Radix Sort

Best: $\Theta(n)$ **Worst:** $\Theta(n)$

Only used to sort numbers. Sort numbers starting at LSB (last digit) and ending at MSB (first digit). Can be done using buckets, gradually shifting one bit to the left, adding 0's in front of smaller numbers.

Selection Sort (PQ)

Best: $\Theta(n^2)$ **Worst:** $\Theta(n^2)$

Selection-sort is a variation of a Priority Queue (PQ-sort), where the queue is implemented with an **unsorted** sequence. Since insert takes O(n), and removing elements in sorted order takes $1 + 2 + ... + n$ because of a linkedlist implementation, PQ-selection sort runs in $O(n^2)$

Insertion Sort (PQ)

Best: $\Theta(n^2)$ **Worst:** $\Theta(n^2)$

Insertion-sort is a variation of a Priority Queue (PQ-sort), where the queue is implemented with a **sorted** sequence. Since insert takes O(n), and removing **all** elements also takes O(n), PQ-insertion sort runs in $O(n^2)$

Heap Sort

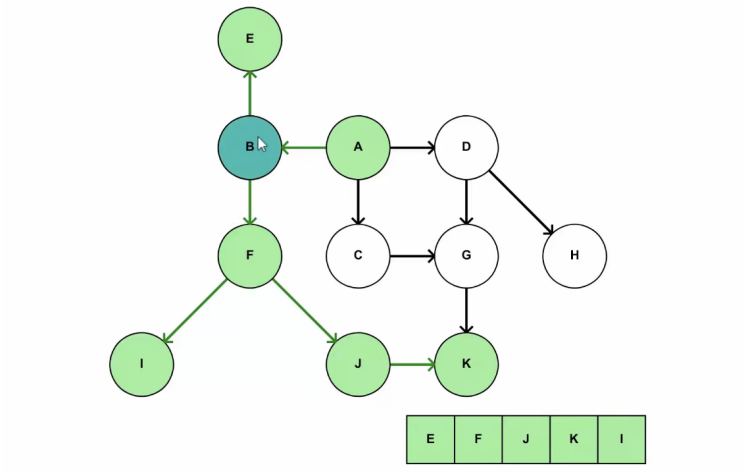
Best: $\Theta(n \cdot \log n)$ **Worst:** $\Theta(n \cdot \log n)$

Use a min-heap-based priority queue. Continuously use **heapify** algorithm, which builds a max-heap from the input. Then swap the root (largest), and the leaf(least) nodes, and consider the largest value sorted, so remove it from the heap, so now need to restore max-heap-property, so repeat. **CHECK IF THIS IS CORRECT OR NOT**

Topological Sort

Best: $\Theta(n+m)$ **Worst:** $\Theta(n+m)$

Traverse a digraph with DFS, to determine the order in which the nodes should be applied for the latter ones to be reachable (table of prerequisites). Whenever a dead end or a node where all children are explored is reached, push the it onto the stack. This sort is not unique.



Array

Array is a sequenced collection of variables all of the same type. Can't change length. Each cell has a unique index

Function	Complexity	Description
Access	$O(1)$	# get element at index i
Insertion	$O(n-i)$	# **insert** element e at index i, shifting n-i elements to the right
Deletion	$O(n-i-1)$	# remove element at index i, shifting all elements to its right to the left by 1

List (Array-based)

List is an Abstract Data Type (ADT). Implemented using an array, which, when full, can be either incremented (amortized time O(n)) or doubled (amortized time O(1)). Space complexity stays $\Theta(N)$ (capacity) and doesn't depend on its size.

Function	Complexity	Description
size()	$O(n)$	# Find length n of the array
isEmpty()	$O(1)$	# check whether size = 0
get(i)	$O(1)$	# get element at index i
set(i, e)	$O(1)$	# replace element at index i into e, and return the old value
add(i, e)	$O(n-i)$	# **insert** element e at index i, shifting n-i elements to the right
remove(i)	$O(n-i-1)$	# remove element at index i, shifting all elements to its right to the left by 1

Linked List

Singly linked list is a concrete data structure comprising a sequence of nodes each storing a value and a one-way pointer to the next node, starting at head pointer. **Doubly linked list** stores values as well as pointers both to next and previous elements. Space complexity: O(n) instead of O(N) like in an array.

Function	Complexity	Description
Insertion (start)	$O(1)$	# make element e the head of the list by setting its pointer to head.
Deletion	$O(i)$	# remove elem at ind i by bypassing (set its predecessor's pointer to its successor)
Insertion (end)	$O(n) \vee O(1)$	# make element e the tail of the list by setting list's tail's pointer to it. Constant time if stores tail, linear if not

---

Set

Set is a collection of distinct elements, **Multiset** is a set where the same element appears more than once: $\{1, 2, 1\}$. Set can be stored using array or list, where elements are stored sorted. All set operations use **generic merge**.

Function	Complexity	Description
add(e)	$O(n)$	# add the element e if not already present
remove(i)	$O(n-i-1)$	# remove element at index i, shifting all elements to its right to the left by 1
addAll(T)	$O(n_S+n_T)$	# $S \cup T$
retainAll(T)	$O(n_S+n_T)$	# $S \cap T$
addAll(T)	$O(n_S+n_T)$	# $S \backslash T$: Set difference

Stack

Stack ADT stores objects like a spring-loaded plate dispenser. Insertion and deletion follow Last-in first-out (**LIFO**) scheme. Doesn't throw exceptions, so all commands are allowed in an empty stack (return null). Used for calculators, parentheses matching, undo sequence in text editor.

Function	**Complexity**	**Description**
push()	$O(1)$	# insert an element on top
pop()	$O(1)$	# removes the top element (last). Works with empty stack.
top()	$O(1)$	# same as peek or pop but without removal. Also works with empty stack

Array implementation: start at index -1, add elements from left to right (as normal), keep track of top element index. But may run out of space, takes O(N): no matter now many elements.
Linked list implementation: add from right to left: make the top element be **head** in a singly-linked list. Doesn't have capacity limit (space used is O(n), so only takes up as much as needed, but also have to store a pointer.)

Queue

Queue ADT is used to store arbitrary object as a.. queue. Insertions (end of the queue) and deletions (front of the queue) follow first-in first-out (**FIFO**). Used for shared resources (printer queue), waiting in line. Unlike stacks, queues ADT has exceptions.

Function	**Complexity**	**Description**
enqueue()	$O(1)$	# insert element at the end of the queue
dequeue()	$O(1)$	# removes and returns element at the front of the queue. Empty queue returns null
first()	$O(1)$	# return front element without removing it. Also returns null in empty queue

Array implementation: either delete the first element O(n-i-1) or increment (bypass) front O(1), but can run out of memory, so use a circular array: front= $(\text{front} + 1)\%N$ (works same for enqueue(), dequeue()). Don't need to store end index because can use size(), and find end = $\text{front} + \text{size}\%N$.
Singly Linked list implementation does not limit capacity, but costs more than performing modular arithmetic. **Tail of Queue is Head of List**: (dequeue() takes O(n-1) because have to access second to last element's pointer). **Tail of Queue is Tail of List**: all operations take O(1).

Priority Queue

Priority Queue ADT is a queue of objects organised by importance of priority (e.g. scheduling jobs in an OS kernel). Priorities may change. Each entry is a KVP, with the key being the priority (highest is 1). Priority queues follow a total order relation $\leq$, extending **Comparator ADT**.

Function	**Unsorted**	**Sorted**	**Description**
insert(k,v)	$O(1)$	$O(n)$	# insert a value with an assigned priority
removeMin()	$O(n)$	$O(1)$	# remove and return and object with the highest priority
min()	$O(n)$	$O(1)$	# return highest priority object without removing it

Unsorted doubly linked List: Insert takes O(1), since can insert the item at beginning or end, but removeMin() and min() take O(n) since have to traverse.
Sorted List: insert takes O(n) since have to find a place to insert it, but removeMin() and min() take O(1) time (smallest element is in the beginning).

Heap

Heaps are nearly-complete binary trees (all levels are filled except for the lowest).
Max-heap: value of node $\leq$ parent, (used for heap-sort) **Downheap** restores max-heap property by moving and swapping high values down the tree until correct spot or leaf is reached.
Min-heap: value of node $\geq$ parent (used for priority queues, where to remove first element) **Upheap** restores min-heap property by moving and swapping low values up the tree until correct spot or root is reached.
The height of the tree is O(log n), so Downheap/Upheap both run in O(logn).

Function	**Complexity**	**Description**
Access	$O(\log n)$	# same as any binary tree.
findMin/Max	$O(1)$	# In min/max heaps the elements are sorted.
Insertion	$O(\log n)$	# insert element and move it to the correct position
Deletion	$O(\log n)$	# In **min-heap**, replace root key with the last node, and remove the last node, now restore heap order with downheap.

Array version of a heap: index starts at 1, then left(i)=2i, right(i)=2i+1, parent(i) = floor(i/2).

Map (Unsorted doubly-linked list)

Map (associative array) models searchable collection of key-value entries. No multiple entries with same key, used for quick searching.

Function	**List-based**	**Description**
get(k)	$O(n)$	# if map has key k, return value, otherwise null
put(k,v)	$O(n)$	# insert entry (k,v), if already exists, return old value
remove(k)	$O(n)$	# if map has key k, remove, otherwise return null
entrySet()	$O(n)$	# return iterable collection of entries (KVP)
keySet()	$O(n)$	# return iterable collection of keys
values()	$O(n)$	# return iterable collection of values

Multimap is a map that can store multiple entries with same key, so have a linked list of values corresponding to each key instead of just values.

Tree

Tree has root(no parent), internal node ($\geq$ 1 child), leaf (no child), ancestors (up), descendants (down), siblings (same parent), subtree (node, its descendants). Depth: (length of path from root to node, depth of root is 0), height (max depth).

Function	**Complexity**	**Description**
root()	$O(1)$	# return tree root node
parent(p)	$O(1)$	# return the parent of a given node
children(p)	$O(c)$	# return collection of children nodes of p
numChildren(p)	$O(1)$	# return number of children of p
isInternal(p)	$O(1)$	# check whether the node is internal
isExternal(p)	$O(1)$	# check whether the node is external
isRoot(p)	$O(1)$	# check whether the node is root

```
preOrder(v) { visit(v), for each (child w of v { preOrder(w) } } # node before descendants
postOrder(v) { for each (child w of v { postOrder(w) }, visit(v)} # node after descendants
```

Binary Tree

Binary tree ADT is a tree where each node has ≤ 2 children, in a **proper** (full) binary tree: $= 2$. Each node stores element, parent, left and right children. In proper binary trees, with num of nodes n, num internal/external nodes i, e and height h: • e = i + 1, • $\log_2$(n+1)-1 $\leq$ h $\leq$ (n-1)/2,

Function	**Complexity**	**Description**
left(p)	$O(1)$	# return left child node
right(p)	$O(1)$	# return left child node
sibling(p)	$O(1)$	# return sibling node

```
inOrder(v) { # left subtree → node, → right subtree
if (left(v)≠ null { inOrder(left(v)) }
visit(v)
if (right(v)≠ null { inOrder(right(v)) }}
```

In-order traversal can be used for arithmetic expressions, decision trees and search. Can prove theorem about comparison based sort, since the height is at least $\log(n!) = \Theta(n \cdot \log n)$

Hash Table

Hash table is a better way to implement a map. Uses **hash function** h, mapping keys to integers in a fixed interval of buckets: [0, N-1]. Contains **Hash code**: h_1 : keys $\rightarrow$ integers and **Compression function**: h_1 : integers $\rightarrow [0, N-1]$, hence item (k,v) will be stored at $i = h(k) = h_2(h_1(k))$. **Collisions**: different elements are mapped to the same cell.

Function	**Complexity**	**Description**
Access	$O(1)$	# simply use the key to retrieve the value.
put(k,v)	$O(1)$	# probe until the cell is free or marked DEFUNCT, and store
remove(k)	$O(1)$	# replace the entry with DEFUNCT sentinel

• **Hash code** generated using either integer interpretation of its bit, (but java uses on 32-bit, can combine high and low-order portion of the key to get 32-bit key), so use **polynomial hash code**: compute $\sum_{i=0}^{n-1} s[i]p^i \mod m$, for some value p, ignoring overflow (can use **Horner's** rule.), suitable for **strings**.
• **Compression functions**: **Divsion**: $h_2(y) = y \mod N$ for some prime N, or Multiply, Add, Divide (**MAD**): $h_2(y) = (ay + b) \mod N$, where $a, b \in \mathbb{Z}^+, a \mod N \neq 0$

• To resolve collisions, can use **Separate Chaining**: each cell is a linked list, containing all colliding elements: For hash table with N slots and n elements, define **load factor** $\alpha = n/N$. When $n = O(N)$ basic operations take O(1).

• Can use **Open addressing**: colliding item placed in different cell (**probe**). To decide upon a probe: $h(k, i) = h'(k) + \mathbf{f(i)}) \mod N$, so the result of the hash is offset by a number decided by a **probing function**. **Linear probing**: use $f(i) = i$, so collision moves to the next slot (suffers from **primary clustering**, which makes some parts more likely to be filled than others). **Quadratic probing**: use $f(i) = i^2$, (suffers from **secondary clustering**). **Double hashing**: use $f(i) = ih''(k)$, e.g. $h''(k) = q - k \mod q$, where $q < N$ prime. Load factor should remain below 1/2, otherwise rehash, new size prime number approx. double the size.
Open addressing uses **lazy deletion**, marking elements as deleted (with DEFUNCT) instead of removing.

Binary Search Tree (BST)

BST efficiently implements **ordered map** (keys satisfy total order) and key(leftChild)≤key(node)≤ key(rightChild), supports nearest neighbor queries. Binary tree search: value < node? left else if >? right else found.

Function	**Worst**	**Average**	**Description**
get(k)	$O(n)$	$O(\log n)$	# return value v at key k
put(k,v)	$O(n)$	$O(\log n)$	# replace value at key k or insert a new entry. Always insert as a leaf.
remove(k)	$O(n)$	$O(\log n)$	# removes entry with key k or return null. no children: just remove; 1 child: make child of the node to be child of the parent (bypass); 2 children: replace value by max(left subtree), and remove it.

Skip List

Skip list is a two-dimensional collection of positions, horizontally into **levels** (as lists S_i) and vertically: **towers** (storing same elements across consecutive lists). Each level contains two special keys: $\pm\infty$ and starting at S_1: a random subsequence of a level below: $S_0 \supseteq S_1 \supseteq ... \supseteq S_h$, where level S_h has only $\pm\infty$.

Function	**Complexity**	**Description**
next(p)	$O(1)$	# return position following p in the same level
prev(p)	$O(1)$	# return position preceding p in the same level
above(p)	$O(1)$	# return position above p in the same tower
below(p)	$O(1)$	# return position below p in the same tower
SkipSearch(k)	$O(\log n)$	# start at $S_h[0]$. For $y = key(next(p))$: $k = y \rightarrow$ return element(next(p)) $k > y \rightarrow$ "scan forward", $k < y \rightarrow$ "scan down". Return null if drop past S_0
put(k, v)	$O(\log n)$	# Use randomized algo. (toss an indep. unbiased coin). Run SkipSearch(k), if position p found, overwrite, otherwise, insert (k,v) after p in S_0, then keep adding height to its tower by flipping heads and stopping on tails.
remove(k)	$O(\log n)$	# SkipSearch(k): remove element at position p and the tower above it else return null

Implemented using **quad-node** (with pointers around). **Max height**: $n/2^i$, with average height: O(log n), required storage: O(n)

AVL Tree

Adelson-Velskii and Landis (AVL) is a balanced BST: For each node, height of left and right subtrees (**balance factor**) differ by **at most 1**. Hence, height is O(log n) from recurrence $N(h) = N(h-1) + N(h-2) + 1$ with $N(1) = 1, N(2) = 2$

Function	**Complexity**	**Description**
Insertion	$O(\log n)$	# insert like in BST, then perform Trinode restructuring to restore balance factor.
Deletion	$O(\log n)$	# same as BST, then also perform Trinode restruct.
Search	$O(\log n)$	# explain

Trinode restructuring **Runtime: O(1)**

Left-left heavy → node right rotate ↻

Left-right heavy → child left then node right 1. ↺, 2. ↻

Right-left heavy → child right then node left 1. ↻, 2. ↺

Right-right heavy → node left rotate ↺

Graph

Edge List implementation of a graph stores vertecies and edges as unordered lists V and E. Each vertex contains element, reference to position in vertex sequence. Each edge object contains: element, vertex origin and destination objects and reference to position in edge sequence.
Adjacency List: for each vertex v, store a list of edges whose entries are incident to v: $I(v)$ for a **diagraph**: store $I_{\text{out}}(v), I_{\text{in}}(v)$
Adjacency Map: improve performance of Adj. List by using hash-based map to implement $I(v)$. Let opposite endpoint of each incident edge to v be key, and the edge be the value, hence getEdge(u,v) and areAdjacent(u,v) run in **expected** O(1)
Adjacency Matrix: use $n \times n$ 2-D array, storing vertices in columns, and incident edges in rows, but bad for sparse matrices, and insert takes O(n^2)

Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
outDegree(v) inDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v) incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Digraph (directed graph)

Directed graph is one whose edges are all directed. Used for one-way streets, flights, task scheduling. If G simple, $e \leq v(v-1)$ instead of $e'leqn(n-1)/2$ for an undirected graph. Directed Acyclic Graph (**DAG**). **Topological ordering** is a numbering $v_1, ...v_n$ of its vertices s.t. for every edge $(v_i, v_j)|i < j$. **Theorem**: diagraph has topological order iff it's a DAG.

Strongly connected graph: each vertex can reach all other vertices (not necessarily directly). To detect: use DFS, if there's an unvisited node, then it's not strongly connected. Strongly connected components (**SCC**) are **maximal subgraphs** such that all vertices are reachable.

Transitive closure: if digraph G has a directed path from u to $v, u \neq v$ then G* has a **directed edge** from u to v, providing reachability info about that digraph. To compute transitive closure, represent G using adj. list or adj. map in case graph is sparse, otherwise compute reachability matrix or use dynamic programming to solve all pairwise shortest path problems. **Runs in O(n^3)**
