CS2040S

Finals Cheatsheet for 23/24 S2
zaidan s.

Asymptotic Analysis

$T(n) = O(f(n))$ | **Upper Bound**
There exists constants $c > 0$, $n_0 > 0$ such that $\forall n > n_0$, $T(n) < cf(n)$.

$T(n) = \Omega(f(n))$ | **Lower Bound**
There exists constants $c > 0$, $n_0 > 0$ such that $\forall n > n_0$, $T(n) \geq cf(n)$.

$T(n) = \Theta(f(n)) \iff T(n) = O(f(n)) \wedge T(n) = \Omega(f(n))$

Generally, $n^n \succ n! \succ a^n \succ n^a \succ log_a(n)$
$log_a(b) = log_c(a) \div log_c(b)$ $a^{log(x)} = x^{log(a)}$

Amortized Analysis

For every k operation, $T(n) \leq kT(n)$.

Binary Search

Reduce and Conquer $T(n) = T(n/2) + O(1)$ Reduce the size of search by half every single iteration.

Sorting

Sort	Worst-Case	Avg-Case	Stable?
Bubble	$O(n^2)$	$O(n^2)$	Yes
Selection	$O(n^2)$	$O(n^2)$	No
Insertion	$O(n^2)$	$O(n^2)$	Yes
Merge	$O(n^2)$	$O(nlogn)$	Yes
Quick	$O(nlogn)$	$O(nlogn)$	No

Sorts of $T(n) = T(n-1) + O(n) = O(n^2)$

BubbleSort $O(n^2)$, Stable, In-Place **Invariant**: At the end of iteration j:
the **biggest** j items are correctly sorted in the **final** j positions of the array.

SelectionSort $O(n^2)$, Not Stable, In-Place
Invariant: At the end of iteration j:
the **smallest** j items are correctly sorted in the **first** j positions of the array.

InsertionSort $O(n^2)$, Stable, In-Place
Invariant: At the end of iteration j:
the **first** j items are in sorted order.
Note: Complexity goes closer to $O(n)$ as the array becomes increasingly closer to sorted.

Sorts of $T(n) = 2T(n/2) + O(n) = O(nlogn)$

MergeSort $O(nlogn)$, Stable, Not In-Place
1. Split array into two halves
2. Sort two halves
3. Combine

QuickSort $O(nlogn)$, Not Stable, In-Place
1. Partition array into two sub-arrays around a pivot
Elements in lower subarray $< x <$ elements in upper subarray
2. Recursively sort sub-arrays

Handling Duplicates
3-Way Partition:
Pack duplicates together as a partition

Order Statistics
QuickSelect finds k smallest element, $O(logn)$
1. Choose random pivot.
2. If current index of pivot is bigger than k, recurse right. Otherwise, recurse left.

Binary Tree

Definition: A binary tree is either empty or a node pointing to two binary trees.

Traversals (generally)
In-order: Left > *SELF* > Right
Pre-order: *SELF* > Left > Right
Post-order: Left > Right > *SELF*
Level-order: BFS

Operations
delete(v) $O(height)$
1. No children (just remove)
2. 1 child (remove v, connect child(v) to parent(v))
3. 2 children (find x = successor(v), remove it temporarily, remove v, then connect it back to left(v), right(v) and parent(v))

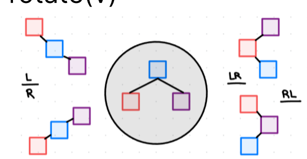
insert(v) $O(height)$
searchMin() $O(height)$ keep recursing left
searchMax() $O(height)$ keep recursing right
successor() $O(height)$
node has right subtree: searchMin(rightSubtree)
else: first parent with key in left subtree.

AVL Trees

Invariants:
1. All nodes have 0-2 children.
2. All keys in left subtree < value in node < All keys in right subtree
3. Height-balanced:
$|height(v.left) - height(v.right)| \leq 1$
If every node is height-balanced, the tree is height-balanced.

A height balanced tree is balanced ($h = O(logn)$)
Min number of nodes n: $n > 2^{h/2}$ ($h < 2logn$)
Max number of nodes n: $n \leq 2^{h+1} - 1$

Rotations:
Insertion max 2
left-hvy, not right-sub hvy: right-rotate(v)
left-hvy, right-sub hvy: left-rotate(v.left), right-rotate(v)
right-hvy, not left-sub hvy: left-rotate(v)
right-hvy, left-sub hvy: right-rotate(v.right), left-rotate(v)



Deletion $O(log(n))$
1. If node has 2 children, swap with successor.
2. Delete node from tree, reconnect children
3. Check all ancestors are height-balanced.

Tries

Time Complexity $O(L)$
Space Complexity more nodes & overhead
Node fixed amount of children for strings

(a,b)-Trees

Invariants/Rules:

1: Child-Policy:

	Key		Child	
Node Type	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

2: Key-Range: For every non-leaf node, child = keys + 1.

3: Leaf-Depth: All leafs are at same depth from root.

Insertion

1. Put element in correct order
2. If overflow, find median key in the node.
3. Put median key in parent, split current node.

Deletion

1. Delete key at present node
2. Check for orphan children/downflow
3. If orphan children, replace deleted key with successor or predecessor.
4. If downflow, check for two cases.
5. If sibling has enough keys, then share keys with the other sibling.
6. Else, merge siblings and parent.

Maximum Height $log_a(n)$

Minimum branches a.
Each non-root node $a-1$ keys.
Root node has 1 key, 2 children.

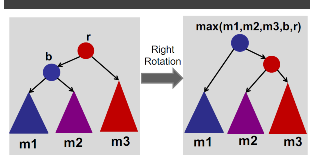
Minimum Height $log_b(n)$

Maximum branches b.
Each non-root node $b-1$ keys.

Interval Trees

Sorted: Left endpoint
Possible augmentation: **max endpoint in subtree**

Rotations (and maintaining max)



All-Overlaps (with k overlaps): $O(klogn)$
Fastest range query implementation: $O(k+logn)$

KD Trees

Generally: split by different dimensions: $x \to y \to x \to y$ for x, y 2D example

Construction Utilise *QuickSelect*: $T(n) = 2T(n/2) + O(n) = O(nlogn)$

Hash Tables

SUHA every key has an **equal probability** of being mapped to each bucket and every key is **mapped independently**
UHA every key is equally likely to be mapped to **every permutation**, independent of every other key.
Good hash function UHA/SUHA + surjective

Collisions

Chaining linked list at the same node
Insertion $O(1 + cost(h)) \implies O(1)$
Searching Worst Case: $O(n + cost(h)) \implies O(n)$
Searching Average Case: $O(\frac{n}{m} + cost(h)) \implies O(1)$

Open Addressing probe sequence until empty found
Delete Use tombstone value.
Advantages Better cache performance, lower space
Disadvantages Sensitive to hash function and load

Bloom Filter Use k hash functions

Heaps

Properties
1. Heap ordering `priority[p] > priority[c]`
2. Complete binary tree

Insert $O(logn)$ Insert at leaf, bubble
Priority Change $O(logn)$ Bubble up/down
Delete $O(logn)$ Swap with bottom right most, bubble

Array Representation
$left(x) = 2x + 1, right(x) = 2x + 2, parent = \lfloor \frac{x+1}{2} \rfloor$

HeapSort $O(nlogn)$
Unsorted array to heap $O(n)$
Heap to sorted array $O(nlogn)$

Graphs

Searching $O(V + E)$
DFS Queue implementation
BFS Stack implementation

SSSP
Bellman-Ford $O(VE)$ *Can be used for negative edge weights.*
Dijkstra $O(ElogV)$

Topological ordering
Kahn's algorithm $O(ElogV)$

Spanning tree
Prim's algorithm $O(ElogV)$ (by vertex)
Naive implementation: $O(EV)$, better for dense graphs.
Kruskal's algorithm $O(ElogV)$ (by edge)
Boruvka's algorithm $O(ElogV)$

Dynamic Connectivity

Quick-find $O(1)$ find, $O(n)$ union
Quick-union $O(n), O(n)$
Weighted-union $O(logn), O(logn)$
WU & path compression $O(\alpha(m,n)), O(\alpha(m,n))$

Dynamic Programming

Optimal sub-structure **Overlapping sub-problems**

Prize collecting $O(kE)/O(kV^2)$
Vertex cover $O(V)/O(V^2)$
APSP **Floyd-Warshall** $O(V^3)$