CS2040S Final Summary

By Chen Yiyang, AY20/21 Sem 2

Order of Growth

O for upper bound, Ω for lower bound

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

- i.e. upper and lower bound the same

Some common ones:

$$T(n) = \log(n!) = O(n \log n)$$

$T(\text{fib}(n)) = O(\phi^n)$, $\phi = 0.618...$ golden ratio

Master Theorem (extra)

$$T(n) = aT(\frac{n}{b}) + f(n), \qquad a \geq 0, b > 1$$

$$= \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

[Algo] Binary Search

Invariant:

[1] A[begin] <= key <= A[end]

[2] range <= n/2^k for iteration k

Criteria for using Binary Search: **monotonic**

Peak Finding --->

```
FindPeak(A, n)
    if A[n/2] is a peak then
        return n/2
    else if A[n/2+1] > A[n/2] then
        search for peak in right half
    else if A[n/2-1] < A[n/2] then
        search for peak in left half
```

[Algo] Sorting

Quick Sort

~ Improvement 1 -

Probabilistic Quick Sort: repeat partitioning until an acceptably good pivot is found. E(X) ~ Geom r.v.

-> always O(nlogn)

~ Improvement 2 – Double Pivot -> 3 regions: smaller, equal, and larger than pivot

Quick Select

To find the k-th largest / smallest item.

Similar to Quick Sort, but only need to recurse on one subarray every time.

~ Time: O(n), if probabilistic

[DS] AVL Tree

Balanced Tree: height of tree = O(logn)

AVL: height-balanced, all nodes' children's height differ by at most 1

Left/right Heavy: left/right child larger height

Operations

Successor -->

```
FUNCTION successor(node)
  IF node.right != null THEN
    return searchMin(node.right)
  ENDIF

  parent <- node.parent
  child <- node
  WHILE ((parent != null) && (child == parent.right))
    child <- parent
    parent <- child.parent
  ENDWHILE
  return parent
END
```

Rotate

left-/right-rotate == root goes to left / right

(Remember to update height/weight/…)

(intuition: treat the above 1 or 2 operations as 1 "rotate set", which reduces height difference by 1)

If `v` is out of balance and left-heavy:

- `v.left` is balanced or left-heavy: `rightRotate(v)`
- `v.left` is right-heavy: `leftRotate(v.left)` then `rightRotate(v)`

If `v` is out of balance and right-heavy:

- `v.right` is balanced or right-heavy: `leftRotate(v)`
- `v.right` is left-heavy: `rightRotate(v.right)` then `leftRotate(v)`

Insert

~ Time O(logn)

~ Rotations: 2 (i.e. "1 set") –> check upwards and adjust first unbalanced

Delete

First delete (using successor) then adjust

~ Time O(logn)

~ Rotations: O(logn) (i.e. "O(logn) sets") -> check all the way up till root & adjust all unbalanced

delete(u) based on cases of 0, 1, 2 children.

[DS] Trie

Tree but not binary tree. Each node stores a char & a special char (for terminal of a string) -> Each root-to-leaf path represents a word

~ rationale: to minimize string comparisons

~ Time: O(L) for search/insert

Space: O(nL) = O(size of text * overhead)

L: max length, n: #strings

[DS] Augmented Tree

Augmented: modify ADT and add in extra data for certain features / functionality.

Order Statistics

Goal: find the k-th largest / smallest item in the tree (i.e. search by rank)

Extra Data: **weight** of each node

~ rank **in subtree** = left.weight+1

~ Time: O(logn) for insert/delete/query

Interval Query

Goal: find an interval / range that covers the current query point

Extra Data: **max endpoint** of the subtree rooted at current node

```
FUNCTION intervalSearch(x):
  c <- root
  WHILE (c not null AND x not in interval of c) DO
    IF (c.left null OR x > c.left.max) THEN
      c <- c.right
    ELSE
      c <- c.left
    ENDIF
  ENDWHILE
  return c.interval
END
```

~ Intuition: always go left unless it is obviously wrong to go left

~ Time: O(logn) for insert/delete/query

~ Limitations: 1) not find the smallest / most suitable interval 2) some (right) intervals will never be returned

Orthogonal Range Search (1D)

Goal: find all the points in a query interval (opposite of Interval Query)

Augmentation: each **leaf** node in a bBST stores data, and each **non-leaf** stores the **largest value in its left subtree**.

~ query(): find "split node" then do left and right traversal (both symmetric)

```
FUNCTION leftTraversal(v, low, high)
  // rightTraversal symmetric
  IF (low <= v.key) THEN
    all-leaf-traversal(v.right)
    leftTraversal(v.left, low, high)
  ELSE
    leftTraversal(v.right, low, high)
  ENDIF
END
```

~ Time: O(logn+k) query, O(logn) insert/delete

n: #nodes in total, k: #nodes in range

[DS] Hashing

(notations: n #items, m #buckets)

Symbol table

~ do not support predecessor/successor

~ keys are **immutable** and no duplicates

Collision: distinct keys hashed to same bucket

Simple Uniform Hashing Assumption:

[1] Each key, equally likely to be mapped to each bucket (among all buckets)

[2] Each key is mapped independently

Double Hashing (f, g hash functions)

$$h(k, i) = f(k) + (i \times g(k)) \bmod m$$

~ g(k) = 1 for linear probing

Collision Solutions

Solution 1: Chaining

Every bucket is a linked list, not a var. (can insert even if n>m)

~ Time for insert O(1) (to front of LL)

~ Expected search time: 1+n/m = O(1)

Expected max cost for insert when n=m: O(logn)

~ Max. search time: O(n)

Solution 2: Open Addressing

Probe a seq. of buckets until an empty one is found and place the item there. (cannot insert once n==m)

~ Linear probing: probe linear seq.

~ Implementation

~ insert: probe until an empty or DELETED bucket and insert item there

~ search: probe until found or not found if probe one empty or all m probed.

~ delete: search for the item and mark the bucket as DELETED (not empty!!!)

~ Grow table size: m -> 2*m when full

~ Time: <= 1/(1- α) for all operations

α = n/m, **load** of hash table

More Hashing Techniques

Table Resizing

(assume **chaining**, **Simple Hashing Assumption**)

~ m *= 2, when n == m

m /= 2, when n <= m/4

~ amortised cost O(1) for insertion, delete

Fingerprint Hash Table (FHT)

~ each item **Boolean**, array space O(logm)

~ possible false positives but no false negatives (don't have but say have, not the other way)

$$P(\text{no false +ve}) = (1 - \frac{1}{m})^n \approx (\frac{1}{e})^{n/m}$$

Bloom Filter

~ k FHTs, with different hashing functions

~ P(no false +ve) = P(no false +ve in FHT)^k

~ all operations now O(k)

[DS] Graph & SSSP

Searching

~ BFS (using Queue) / DFS (using Stack)

~ both time O(V+E) on adjList, O(V^2) on adjMatrix

~ if to find SP on unweighted graph, only BFS

relax(u,v) : dist[v] = min(dist[v], dist[u] + e[u,v]) :D

[Algo] SSSP - Bellman Ford

~ early stop when no relaxed after |E| checks

~ works with -ve edge; can be used to identify -ve cycle

~ **invariant**: after n iterations, n-hops estimates correct

~ Time Complexity O(VE)

```
n = V.length
for i <- 1 TO n:
  for edge IN graph:
    relax(e)
  ENDFOR
ENDFOR
```

[Algo] SSSP – Dijkstra's

~ only works for non -ve edges

~ start w. empty Shortest-Path-Tree, repeat

1. consider vertex w. shortest estimate
2. add vertex to tree
3. relax all outgoing

Analysis

~ insert, extractMin V times each; relax E times

~ overall time O(ElogV), space O(V) (for pq.)

[Algo] Topological Ordering (DAG)

DAG always have topoOrder, but not necessarily unique.

[1] Post Order DFS

~ add elements from end to front

~ time O(V+E)

[2] Kahn's - O(ElogV)

Repeat:
S = nodes in G that have *no* incoming edges.
Add nodes in S to the topo-order
Remove all edges adjacent to nodes in S
Remove nodes in S from the graph

~ use pq. for nodes, where key = # incoming edges

[3] Kahn's – O(E+V)

Use a queue. Enqueue nodes with in-deg. 0 then decrease its adjacent nodes' in-deg. & dequeue.

Other Graph Problems
Longest Path on DAG -> topoOrder
Shortest Path on Tree -> BFS/DFS relax
^ assume tree edges **undirected**!!!

[Algo] Minimum Spanning Tree
MST Properties:
[1] No cycle
[2] If you cut a MST, 2 pieces are MST each
[3] Cycle property: for every cycle, max weight edge not in MST
  ~ min. weight edge of cycle might not in MST
[4] Cut property: for every partition/cut, min. edge of the cut in MST

MST Algorithms
Prim's
~ Start with a node, extend to others with shortest edge. DecreaseKey using **Heap**
~ each node insert & extractMin once, each edge decreaseKey once -> **overall O(ElogV)**
Kruskal's
~ "greedy" on edges.
~ Start with min.edge, connect components and skip edge connecting nodes of same component.
~ O(ElogE) sort, then UFDS -> **overall O(ElogV)**
Boruvka's
~ each Boruvka step, add every component's min. adjacent edge to MST. Start with each node one component, repeat.
~ O(V+E) DFS/BFS for all min. adjacent edges, O(V) for new components -> **overall O(ElogV)**
~ Boruvka good at parallelism

MST Variants
MST on DAG with one root
-> for every node except root, add min. incoming edge to MST
Max Spanning Tree
(Assume still |V|-1 edges in the tree in total)
[1] Run Kruskal's in reverse (max. heap) OR
[2] Negate all edges then run MST algorithms
Steiner Tree 2-Approx.
Goal: find a tree that must spans certain vertices.
~Steps: [1] calculate APSP, [2] construct new graph on required nodes, [3] now do MST on new graph.

[DS] Heap
~ max/min heap: biggest/smallest @ root
~ impt. properties
  [1] Heap Ordering
  [2] Complete Binary Tree
~ stored as array. (index manip. -> parent-children)
~ max height: floor(logn)

Operations
~ bubbleUp, bubbleDown: O(logn) each
~ insert: to last index, then bubbleUp, O(logn)
~ increaseKey/decreaseKey: update & bubbleUp/bubbleDown
~ delete: swap with last, delete, then bubbleDown the swapped element, O(logn)
~ extractMin == delete the root

Heap Sort
~ heap to sorted array: O(nlogn)
~ unsorted array to heap: O(n)!!!

[DS] Union Find Disjoint Set
Quick Find: componentId[] stores which component
  ~ O(1) find, O(n) union
Quick Union: parent[] stores direct parent
  ~ O(n) find, O(n) union
Weighted Union: parent[], size[]
  ~ merge smaller into larger -> "more balanced"
  ~ O(logn) find, O(logn) union
Path Compression – when traversing to root, set parent of each visited node to root
  ~ with weighted union: O(α(m, n)) find, union

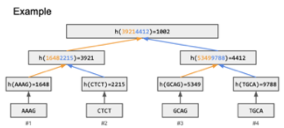
[Algo] Dynamic Programming
Two properties:
[1] Optimal Substructure
[2] Overlapping subproblems
~ if [1] but no [2] -> divide & conquer
Example: Floyd-Warshall for APSP

Recitations
Rec. 03 – Fisher-Yates / Knuth Shuffle --->

```
PROCEDURE KnuthShuffle(A[1..n])
  FOR i <- 2 TO n
    r <- random(1, i)
    swap(A, i, r)
  ENDFOR
END
```

Rec. 04 – (a,b) Tree & B-Tree
B-Tree == (B, 2B) Tree
Definition of (a, b)-Tree
[1] Each non-leaf min. a (2 for root), max. b children.
[1.1] #children == #key+1 for non-leaf; min. (a-1), max (b-1) keys for leaf
[2] Key-ordering for children.
[3] All leaf nodes same height (growing upwards)
Basic Operations:
~ split: give 1 key to parent, and then split this node & its remaining keys to 2 new nodes.
~ merge: pull 1 key down from parent, and merge the two keys divided by this key into one new child node.
~ share: share == merge + split
Other Operations:
~ insert: insert, then check up to root, split if necessary
~ delete: delete, then check up to root, merge/share if necessary.
Analysis:
~ Height min O(log_b n), max O(log_b n)
~ Time: O(logn) for insert/delete/search
~ same order as normal bBST, but smaller coefficient -> shorter tree and better for caching / mem. Access.

Rec. 05 – Merkle Tree
A bBST where leaf nodes store hash of its data & each non-leaf node takes and concatenates the values of children and hash, which is stored as its own value. Same root value indicates all info in subtree identical.



Tutorials
Tut. 03 – Order Maintenance
~ insertAfter(A, B): by insert B as successor of A in bBST, then rotate. Similar process for insertBefore(A, B)
~ isAfter(A, B): to check if B after A in tree, compare their ranks

Tut. 04 – kd-Tree
A tree structure that is used to describe points in a 2D plane. Each node is a (x, y) pair denoting a point and it divides horizontally or vertically (alternate by levels in tree) points into left and right children.
~ search: O(logn) - recursive alternate binary search on x- / y-values.
~ construct tree from unsorted array: O(nlogn) – QuickSelect median, partition.
~ smallest x / y: T(n) = 2T(n/4) + O(1) = O(√n)

Tut. 04 – Finger Search
Augmented bBST that supports search from one node to another (nearby one)
~ Use a B-Tree where each node has a parent point and is linked to nodes of same level. Record min & max of all values in subtree at each node.
~ search(x,y): from x to y (assume y>x), check whether to go up or go right neighbor (go right only when y in the neighbor's min-max range). Once go right or arrive at root, search for y normally in the subtree
  ~ Time: O(logd), d: diff. in x and y's rank.

Miscellaneous (Credits: Jovyn Tan)
2D Orthogonal Range Search
- `insert(key)`, `insert(key)` $\Rightarrow O(\log n)$
- `2D_query()` $\Rightarrow O(\log^2 n + k)$ (space is $O(n \log n)$)
  - build x-tree from x-coordinates; for each node, build a y-tree from y-coordinates of subtree
- `2D_buildTree(points[])` $\Rightarrow O(n \log n)$

Sorting Comparison

sort	best	average	worst	stable?	memory
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	$O(n)$
quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	$O(1)$
heap	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	×	$O(n)$

Sorting Invariants --->

sort	**invariant** (after k iterations)
bubble	largest k elements are sorted
selection	smallest k elements are sorted
insertion	first k slots are sorted
merge	given subarray is sorted
quick	partition is in the right position

DS's Time Complexity

data structure	search	insert
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree (kd/(a, b)/bst)	$O(\log n), O(h)$	$O(\log n), O(h)$
trie	$O(L)$	$O(L)$
heap	$O(n)$	$O(\log n), O(h)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n)$	$O(1)$
open addressing	$\frac{1}{1-\alpha} = O(1)$	$O(1)$
priority queue	(contains) $O(1)$	$O(\log n)$
skip list	$O(\log n)$	$O(\log n)$

$$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$$
$$T(n) = T(n/2) + O(n) \Rightarrow O(n)$$
$$T(n) = 2T(n/2) + O(1) \Rightarrow O(n)$$
$$T(n) = T(n/2) + O(1) \Rightarrow O(\log n)$$
$$T(n) = 2T(n-1) + O(1) \Rightarrow O(2^n)$$
$$T(n) = 2T(n/2) + O(n \log n) \Rightarrow O(n(\log n)^2)$$
$$T(n) = 2T(n/4) + O(1) \Rightarrow O(\sqrt{n})$$
$$T(n) = T(n-c) + O(n) \Rightarrow O(n^2)$$