**Theorems for review**

***Introduction to Algorithms***

Tony Mo

2018.6.18

***Chapter 11 Hash Tables***

**Collision Resolution by Chaining**

**The worst-case running time**

Insertion: O(1).

Searching: Proportional to the length of the list.

Deletion: O(1) time if the lists are doubly linked.

***Simple uniform hashing***.

Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.

***Theorem 11.1***

In a hash table in which collision are resolved by chaining, an unsuccessful search takes expected time O(1+α), under the assumption of simple uniform hashing.

***Theorem 11.2***

If a hash table in which collision are resolved by chaining, a successful search takes average-case time O(1+α), under the assumption of simple uniform hashing.

**Hash Functions**

***Division method***

h(k) = k mod m

***Multiplication method***

h(k) = floor[m(kA mod 1)].

**Open Addressing**

***Linear Probing***

h(k, i) = (h’(k) + i) mod m

***Primary clustering***

Clusters arise because an empty slot preceded by 𝒊 full slots

gets filled next with probability 𝒊 + 𝟏 /𝒎.

***Quadratic probing hash function***

h(k, i) = (h’(k) + c1i + c2i^2) mod m

***Secondary clustering***

Only m distinct probe sequences are used.

***Double hashing***

h(k, i) = (h1(k) + i h2(k) ) mod m 

O(m^2) probe sequences are used.

***Theorem 11.6***

Given an open-address hash-table with load factor α= n/m < 1, the expected number of probes in an unsuccessful search is at most 1/(1 - α) assuming uniform hashing.

***Corollary 11.7***

Inserting an element into an open-address hash table with load factor α requires at most 1/(1 – α) probes on average, assuming uniform hashing.

***Theorem 11.8***

Given an open-address hash table with load factor α< 1, the expected number of successful search is at most (ln(1-α))/α.

***Chapter 12 Binary Search Tree***

***Binary-search-tree property***

Let x be a node in a binary search tree.

If y is a node in the left subtree of x, then y.key <= x.key.

If y is a node in the right subtree of x, then y.key >= x.key.

***Theorem 12.1***

If x is the root of an n-node subtree, then the callINORDER-WALK(x) takes O(n) time.

***Theorem 12.2***

The dynamic-set operations, SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in O(h) time on a binary search tree of height h.

***Theorem 12.3***

 The dynamic-set operations, INSERT and DELETE can be made to run in O(h) time on a binary search tree of height h.

***Chapter 13 Red-Black Trees***

***Red-black properties:***

Every node is either red or black.

The root is black.

Every leaf (NIL) is black.

If a node is red, then both its children are black.

For each node, all paths from the node to descendant leaves

contain the same number of black nodes.

***Lemma 13.1***

 A red-black tree with n internal nodes has height at most

2lg(n + 1).

***Chapter 15 Dynamic Programming***

***Theorem 15.1 (Optimal substructure of an LCS)***

Let X = <x1, x2, ..., xm> and Y = <y1, y2, ..., yn> be sequences, and let Z = <z1, z2, ..., zk> be any LCS of X and Y.

1. If xm = yn, then zk = xm = yn and Zk-1 is an LCS of Xm-1and Yn-1.

2. If xm ≠ yn, then zk ≠ xm implies Z is an LCS of Xm-1 and Y.

3. If xm ≠ yn, then zk ≠ yn implies Z is an LCS of X and Yn-1.

Observations:

• Optimal BST might not have smallest height.

• Optimal BST might not have highest-probability key at root.

***Chapter 16 Greedy Algorithms***

***Greedy-choice property***

A globally optimal solution can be arrived at by making a

locally optimal (greedy) choice.

In Greedy:

 Make a choice at each step.

 Make the choice before solving the subproblems.

 Solve top-down.

***Chapter 17 Amortized Analysis***

***Aggregate analysis***

In the worst case, the average cost, or amortized cost, per

operation is therefore T(n)/n.

***Accounting method***

Assign different charges to different operations.

• Some are charged more than actual cost.

• Some are charged less.

Amortized cost = amount we charge.

***Binary counter***

Charge $2 to set a bit to 1.

• $1 pays for setting a bit to 1.

• $1 is prepayment for flipping it back to 0.

• Have $1 of credit for every 1 in the counter.

• Therefore, credit ≥ 0.

Amortized cost of INCREMENT:

• Cost of resetting bits to 0 is paid by credit.

• At most 1 bit is set to 1.

• Therefore, amortized cost ≤ $2.

• For n operations, amortized cost = O(n)

***Potential method***

Like the accounting method, but it thinks of the credit as potential stored with the entire data structure.

• Accounting method stores credit with specific objects.

• Potential method stores potential in the data structure as a whole.

• Can release potential to pay for future operations.

• Most flexible of the amortized analysis methods.

Potential function φ: Di → R

C^i = ci +Δ(Di)

Total amortized cost = Σci + φD(n) – φD(0).

***Chapter 22 Elementary Graph Algorithms***

The total running time of BFS is 𝑶 (𝑽 + 𝑬).

***Shortest-path distance*** 𝜹(𝒔, 𝒗) from 𝒔 to 𝒗.

The minimum number of edges in any path from vertex 𝒔 to vertex 𝒗.

***Lemma 22.1***

Let 𝑮 = (𝑽, 𝑬) be a directed or undirected graph, and let 𝒔 ∈ 𝑽 be an arbitrary vertex. Then, for any edge (𝒖, 𝒗) ∈ 𝑬,

𝜹 (𝒔, 𝒗) ≤ 𝜹 (𝒔, 𝒖) + 𝟏

***Lemma 22.2***

Let 𝑮 = (𝑽, 𝑬) be a directed or undirected graph, and suppose that BFS is run on 𝑮 from a given source vertex 𝒔 ∈ 𝑽. Then upon termination, for each vertex 𝒗 ∈ 𝑽, the value 𝒗.𝒅 computed by BFS satisfies 𝒗. 𝒅 ≥ 𝜹 (𝒔, 𝒗) .

***Lemma 22.3***

Suppose that during the execution of BFS on a graph 𝑮 = (𝑽, 𝑬) , the queue 𝑸 contains the vertices <𝒗𝟏, 𝒗𝟐, ⋯ , 𝒗𝒓>,where 𝒗𝟏 is the head of 𝑸 and 𝒗𝒓 is the tail. Then, 𝒗𝒓. 𝒅 ≤𝒗𝟏. 𝒅 + 𝟏 and 𝒗𝒊. 𝒅 ≤ 𝒗𝒊+𝟏.𝒅 for 𝒊 = 𝟏, 𝟐, ⋯ , 𝒓 − 𝟏.

***Corollary 22.4***

Suppose that vertices 𝒗𝒊 and 𝒗𝒋 are enqueued during the execution of BFS, and that 𝒗𝒊 is enqueued before 𝒗𝒋. Then 𝒗𝒊. 𝒅 ≤ 𝒗𝒋. 𝒅 at the time that 𝒗𝒋 is enqueued.

***Theorem 22.5 (Correctness of breadth-first search)***

Let 𝑮 = (𝑽, 𝑬) be a directed or undirected graph, and suppose that BFS is run on 𝑮 from a given source vertex𝒔 ∈ 𝑽. Then, during its execution, BFS discovers every vertex 𝒗 ∈ 𝑽 that is reachable from the source 𝒔, and upon termination, 𝒗. 𝒅 = 𝜹 (𝒔, 𝒗) for all 𝒗 ∈ 𝑽. Moreover, for any vertex 𝒗 ≠ 𝒔 that is reachable from 𝒔, one of the shortest paths from 𝒔 to 𝒗 is a shortest path from 𝒔 to 𝒗.𝝅 followed by the edge 𝒗.𝝅, 𝒗 .

The running time of DFS is O(𝑽 + 𝑬) .

***Properties of depth-first search***

DFS yields valuable information about the structure of a graph.

The predecessor subgraph 𝑮𝝅 does indeed form a forest of trees.

Vertex 𝒗 is a descendent of vertex 𝒖 in the depth-first search if and only if 𝒗 is discovered during the time in which 𝒖 is gray.

Discovery and finishing times have parenthesis structure.

***Theorem 22.7 (Parenthesis theorem)***

For all u, v, exactly one of the following holds:

1. d[u] < f [u] < d[v] < f [v] or d[v] < f [v] < d[u] < f [u] and neither of u and v is a descendant of the other.

2. d[u] < d[v] < f [v] < f [u] and v is a descendant of u.

3. d[v] < d[u] < f [u] < f [v] and u is a descendant of v.

***Theorem (White-path theorem)***

v is a descendant of u if and only if at time d[u], there is a path

u ~> v consisting of only white vertices. (Except for u, which was just colored gray.)

***Theorem 22.10***

In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

***Directed acyclic graph (dag)***

A directed graph with no cycles.

***Topological sort of a dag***

A linear ordering of vertices such that if (u, v) ∈ E, then u appears somewhere before v. (Not like sorting numbers.)

Time: O(V + E).

***Lemma***

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

***Strongly connected component***

 For a directed graph 𝑮 = (𝑽, 𝑬) , a maximal set of vertices 𝑪 ⊆ 𝑽 such that for every pair of vertices 𝒖 and 𝒗 in 𝑪, we have both u ~> v and v ~> u.

The time to create 𝑮^𝑻 is 𝑶 (𝑽 + 𝑬).

**SCC(G)**

call DFS(G) to compute finishing times f [u] for all u

compute G^T

call DFS(GT), but in the main loop, consider vertices in order of decreasing f[u](as computed in first DFS)

output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

Time: 𝑶 (𝑽 + 𝑬)

***Lemma***

GSCC is a dag. More formally, let C and C’ be distinct SCCs in G, let u, v ∈ C, u’, v’ ∈ C’, and suppose there is a path u ~> u’ in G. Then there cannot also be a path v’ ~> v in G

***Lemma***

Let C and C’ be distinct SCCs is in G = (V, E). Suppose there is an edge (u, v) ∈ E such that u ∈ C and v ∈ C’ . Then f (C) > f (C’).