CSE 373 Midterm Review

Spring 2014 - Nicholas Shahan

Stacks LIFO - Last In First Out

Main Operations

- push - place element on the top of the stack
- pop - remove element at the top of the stack

Runtime: push O(1) pop O(1)

Disadvantage: O(n) searches

Queues FIFO - First In First Out

Main Operations

- enqueue - place element at the back
- dequeue - remove element from the front

Runtime: enqueue O(1) dequeue O(1)

Disadvantage: O(n) searches

Binary Search Tree Dictionary

- Nodes can have two children
- Order Property
  - Child to the left is less than parent
  - Child to the right is greater than parent

Runtime: insert O(n) find O(n)

Disadvantage: Can become unbalanced

AVL Tree Dictionary

- Nodes can have two children
- Order Property
  - Child to the left is less than parent
  - Child to the right is greater than parent
- Balance Property
  - Height of children never differs by more than 1

Runtime: insert O(logn) find O(logn)

Trade-off: Sorted but operations not O(1)

AVL Tree Dictionary

Single Rotations - Imbalanced at:

- right-right
- left-left

Double Rotations - Imbalanced at:

- right-left
- left-right

Heaps Priority Queue

Structure property

- A *complete* binary tree

Heap property **(Not a binary search tree)**

- The priority of every (non-root) node is less important than the priority of its parent

Runtime: insert O(logn) deletemin O(logn) findmin O(1)

Trade-off: Always fast access to the min but access to any other elements O(n)

Heaps Priority Queue

insert - place element as a new leaf and *Percolate Up*.

deletemin - remove element at the root, place the last leaf element at the root position and *Percolate Down.*

Heaps Priority Queue

- Imagine as a *Binary Tree.*
- Implement as an *Array.*

From node `i`

- left child: `i*2`
- right child: `i*2+1`
- parent: `i/2`

Disjoint Sets Union-Find

Main Operations

- union - join two sets together
- find - identify the set containing an element

Runtime: union O(1)

find O(logn) a single time or O(log*) (amortized)

Trade-off: Very fast but limited operations

Disjoint Sets Union-Find

- Imagine as a forest of *Up Trees*.
- Implement as an *Array.*

- All nodes (except roots) point to their parent.
- Roots record the size of their tree.

Disjoint Sets Union-Find

Optimizations

- Union by Size - when performing a union the smaller tree joins the larger.
- Path Compression - during a find, after traversing the tree up to the root, connect all nodes on the path directly to the root.

Hash Tables Dictionary

Keys are “hashed” to determine which bucket to store the value in.

Runtime: insert O(1) find O(1) *assuming a good hash function and proper table size

Can be much worse O(n)

Trade-off: Operations are O(1) but unsorted

Hash Tables Dictionary

“Hash” the key: **`h(key) % TableSize`**

Collision: When two keys “hash” to the same bucket.

Collision Resolution: Rules used when a collision occurs.

Collision Resolution Hash Tables

Separate Chaining

- All buckets point to a linked list containing all the values in that bucket.
- With a good hash function, not as bad as it sounds since collisions should be rare.

Collision Resolution Hash Tables

Linear Probing

- If a bucket is occupied, use the next one. If that is occupied, use the next one…
- Probe Sequence:

  0^{th} probe: **`h(key) % TableSize`**

  1^{st} probe: **`((h(key) + 1) % TableSize`**

  2^{nd} probe: **`((h(key) + 2) % TableSize`**

  3^{rd} probe: **`((h(key) + 3) % TableSize`**

  i^{th} probe: **`((h(key) + i) % TableSize`**

Collision Resolution Hash Tables

Quadratic Probing

- If a bucket is occupied, look for a new location with a series of probes that increase quadratically
- Probe Sequence:

0^{th} probe: **(h(key) % TableSize** **$0^2=0$**

1^{st} probe: **((h(key) + 1) % TableSize** **$1^2=1$**

2^{nd} probe: **((h(key) + 4) % TableSize** **$2^2=4$**

3^{rd} probe: **((h(key) + 9) % TableSize** **$3^2=9$**

i^{th} probe: **((h(key) + i^2) % TableSize**

Collision Resolution Hash Tables

Double Hashing

- If a bucket is occupied, hash the key with a ***different*** hash function and add to the original location.
- Probe Sequence:

  0^{th} probe: **`h(key) % TableSize`**

  1^{st} probe: **`((h(key) + g(key)) % TableSize`**

  2^{nd} probe: **`((h(key) + 2*g(key)) % TableSize`**

  3^{rd} probe: **`((h(key) + 3*g(key)) % TableSize`**

  i^{th} probe: **`((h(key) + i*g(key)) % TableSize`**

You all know this already...

Test Tips

- Relax
- Read all the instructions carefully - some details are easy to miss *(I speak from experience)*
- If you feel stuck, move on and come back after completing some other problems.