CSCE 750
10/3/2023

Hash tables (cont.)
Binary Search Trees

Hash table: array $H[0..(m-1)]$ with m entries

"Random-looking" but deterministic & easy to compute hash function

$h(k) \in [0..(m-1)]$ for any key k

Collision: $k_1 \neq k_2$ but $h(k_1) = h(k_2)$

Handling collisions:

- Chaining — $h[j]$ is a linked list of all keys k such that $h(k) = j$.
- Open addressing — all items stored in the array H itself. If inserting k results in a collision; ($h(k)$ ~~is~~ is occupied) then look for another location.

  Different strategies:

  - try $h(k)+1$, $h(k)+2$, ... until empty space found. [sequential search]

    Naive & poorly performing

  - better: try a different $h_2(k)$. If this is occupied, then sequential search

Analysis (with chaining)

Uniform simple hashing assumption:

$$\forall j_{\,0 \le j \le m-1} \quad \Pr_{k}[h(k) = j] = \frac{1}{m}$$

Reasonable assumption if you have a sufficiently "crazy" hash function.

<u>Worst Case</u>: All keys hash to the same index:

$\left.\begin{array}{l} \text{Insert} — O(1) \\ \text{Search (unsuccessful)} — \Theta(n) \\ \text{Delete} — \end{array}\right]$ no better than a linked list

<u>Expected</u> time for an unsuccessful search given the uniform simple hashing assumption is

$$A(n) = \underbrace{\Theta(1)}_{\substack{\text{constant}\\ \text{time to}\\ \text{compute}\\ h(k)}} + \Theta\left(\sum_{j=0}^{m-1} \underset{\substack{\uparrow \\ \Pr[h(k)=j]}}{\frac{1}{m}} \cdot \text{Length}(H[j])\right)$$

$$= \Theta(1) + \Theta\left(\frac{1}{m} \sum_{j=0}^{m-1} \text{Length}(H[j])\right)$$

$$= \Theta\left(1 + \frac{n}{m}\right)$$

n = # items in the hash table

m = # array entries

If $n = O(m)$, then this is constant time.

Binary Search Trees (BSTs)

Collection of items with comparable keys.

Supporting

Insert
Search
Delete
⋮

Rooted, ordered Binary tree of keys (assume no duplicate keys)

k = root key $\Rightarrow$ left subtree is a BST containing only keys $< k$ &
right subtree is a BST containing only keys $> k$.

Search, Insert, Delete all take worst-case time $\Theta(h)$ where h is the height of the tree.

* Avoid BSTs where h is big.

Generally want $h = O(\lg n)$ (n = # items)

Some standard techniques to ~~key~~ keep a BST balanced so that $h = O(\lg n)$ always:

- AVL trees
- Red-Black trees (2-3 trees)

Randomization? Does that help?

~~If items~~

With n items, there are $n!$ many permutations. If one of these perms is chosen uniformly at random for insertion order into the BST, then the expected ~~height~~ time for each op is $O(\lg n)$.

[Analysis is the same (essentially) as randomized quicksort]

Can't count on a random insertion order but can add randomness to the algo to mimic a random insertion order. Treap (tree/heap hybrid).

Def: A treap T is a binary tree of items that each have a comparable key and a numerical priority, such that

T is a BST of the keys (ignoring priorities)

& T is a max heap of the priorities (ignoring keys)

each nonroot priority is < parent priority

{assume no duplicate priorities}

Fact:
The shape of T depends only on the set of (key, priority) pairs.

Proof: by induction on the size of T

{base case: T is empty}

T nonempty: root is item with highest priority [regardless of anything else]

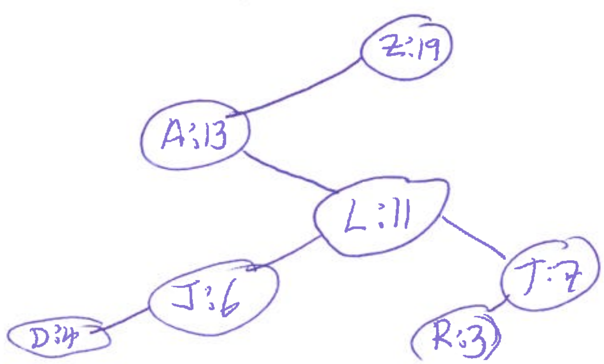
Left subtree: determined by the root key among set of all keys.

Inductive hypothesis: root.left shape is uniquely determined

Same for the right subtree.

Ex: key:priority items

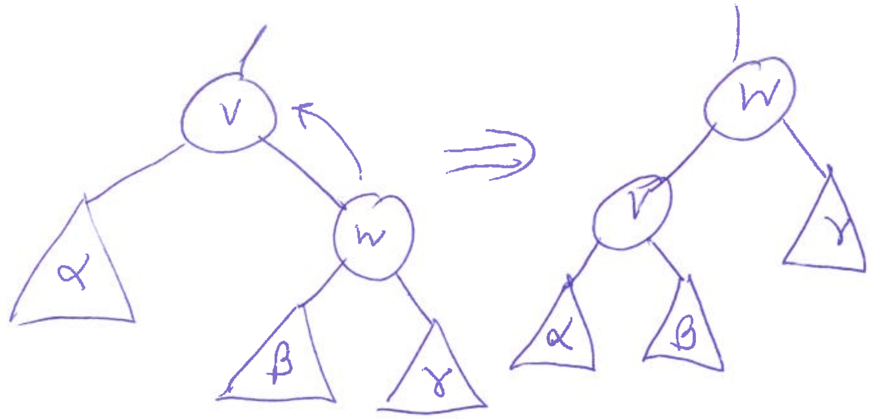
~~J:6~~, ~~A:13~~, ~~R:3~~, ~~T:7~~, ~~Z:19~~, ~~L:11~~, ~~D:4~~



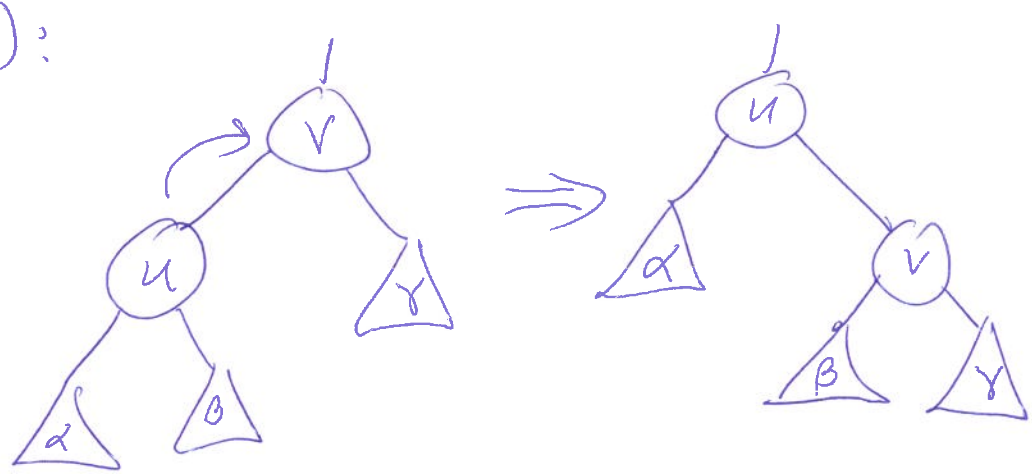
Rotations 2 types:

O(1) time

LeftRotate(v):



RightRotate(v):



~~Implementing a BST using a randomized treap~~
~~treap~~

Inserting into a treap:

1) Insert as usual with a BST (new item at a leaf)
2) Go up the search path, ~~given~~ making rotations until the max-heap order is restored.