CS 240 Summary P1

Priority Queues

- Collection of objects with a priority item
- Uses max/min heaps to achieve this

Key Operation:
- item insert into PQ - $O(\log(n))$ time.
- deleteMax from PQ - $O(\log(n))$ time
- height of a min/max heap: $\Theta(\log(n))$
- build a heap - $\Theta(n\log(n))$ time
- sort using a heap - $\Theta(n\log(n))$ time

When put into an array...
- left child of $A[i]$ is $A[2i+1]$
- right child of $A[i]$ is $A[2i+2]$
- parent of $A[i]$ is $A\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right]$

Pseudo-Code: parent(v), Key(parent(v)), ← child with largest Key, heapInsert(A, x), heapDeleteMax(A), HeapSort(A)

- Check the Kth-max problem.
  - When is one better than another?

Sorting & Randomization

- Selection vs. Sorting
- Randomization as a factor - expected time
- Comparison vs. noncomparison based sorting

Key Operations:
- quick-select(A, K) has two subroutines
  - choose-pivot(A)
  - partition(A)
  - worst $\Theta(n^2)$, average $\Theta(n)$
- quick-sort(A) worst $\Theta(n^2)$, average $\Theta(n)$
  - multiple ways of calling choose-pivot, as such multiple quicksorts (see slides)

Hilroy

Sort	Running Time	Analysis	Stable?
selection	$\Theta(n^2)$	worst	Y
insertion	$\Theta(n^2)$	worst	Y
merge	$\Theta(n\log n)$	worst	Y
Heap	$\Theta(n\log n)$	worst	N
quick-sort1	$\Theta(n\log n)$	average	N
quick-sort2	$\Theta(n\log n)$	expected	N
quick-sort3	$\Theta(n\log n)$	worst	N
countsort	$\Theta(n+R)$	worst	Y
LSD radix	$\Theta(m(n+R))$	worst	Y

BST & AVL Trees

- Search, insert, delete in a BST are all $\Theta(h)$. Since $\Theta(h)$ is average $\Theta(\log n)$, so are the operations.
- $\Theta(h)$ of a BST has worst case $\Theta(n)$
- AVL trees fix this problem by ensuring that $\Theta(h)$ is $\log(n)$.

Key operations:

- search(T, x), insert(T, x), delete(T, x) in both
- fix(T) is $\Theta(\log n)$ i.e $\Theta(h)$
  - Subroutines rotate-left, rotate-right(T)
  - fix is called up to $\Theta(h)$ times, its subroutines are constant time

Pseudo Code:

- Rotations use .left, .right
- i.e newroot ← T.left, T.left ← newroot.right
- Balances also called directly
- i.e T.left.balance == -2

Double-right: -2, +1

Double-left: +2, -1

CS 240 Summary P2

Tries

- A dictionary of binary Strings
- Left children are 0; right are 1
- Keys don't get stored in a Trie, nodes are flagged to indicate a key exists.

Key Operations:

- Search(x), insert(x), delete(x) are all $\Theta(|x|)$, i.e size of binstr x.
- Augmented above for compressed tries

Skip Lists

- Randomized data structure
- Hierarchy of ordered linked lists
- Two dimensional positions: levels & towers

Key Operations:

- Skip-search(L, K), Skip-insert(S, K, v), Skip-delete(L, K) all done $O(\log n)$ expected

Pseudo Code:

- Dropping down a level is below(P)
- Scanning to the next Key is after(P)
- Use $p \leftarrow$ topmost left position to start
- Use $S \leftarrow$ stack of positions starting with p to start a return stack

Hashing

- Uses direct addressing to achieve $\Theta(1)$ operations!
- We improve upon direct addressing and use a hash function instead.
- Our hashing must have a method of dealing with collisions (Keys are NOT unique).

Key Operations

- search(K), insert(K,v), delete(K) are all O(1) operations
- Collision method 1 is Chaining - each table entry is a bucket containing an unsorted linked-list of entries
- Collision method 2 is open addressing - each table entry only holds one item, but we can hash an item to multiple locations.
- Collision method 3 is double hashing - we have two independant hashing functions to find our location
- Collision method 4 is cuckoo hashing - Again, we have two independant hashing functions. This time, we always insert a new item in its first location, by booting out its old item if necessary, and making it use its alternative hashing function.

A good hash function is efficient to compute, unrelated to patterns in our data, and depends on all parts of our key.