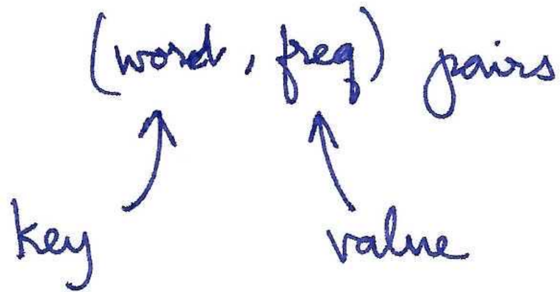
Hash Table



Keys are distinct

Insert a word	Insert(key)
Update freq for word	Update(key, val)
Find	Find(key)
Delete	Delete(key)

Q. Suppose we use an AVL tree for these operations. How long does each operation take?

(R) $\Theta(1)$ ~~(G)~~ $\Theta(\log n)$ (B) $\Theta(n)$

Hash tables perform operations in $\Theta(1)$.

(1) Memory allocation is complicated.

(2) Deletion is complicated

(3) Hash tables are randomized data structures.

Hash function Initialize size to m.

(Ideally m should be n, # keys to be inserted)

Keys are from a set K

K ← ints, strings, objects, ---

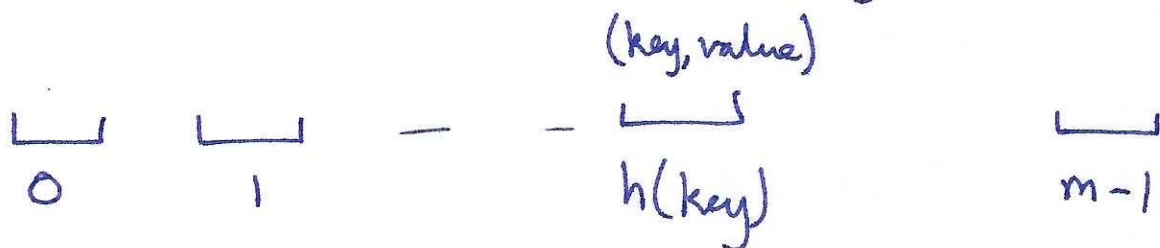
$h: K \longrightarrow \{0, 1, 2, \ldots m-1\}$

If K was int, $h(x) = x \pmod{m}$

h is typically a "random" function

Chaining Create m bins/buckets.

Store (key, value) pair in bin $h(\text{key})$



$h(\text{key}) = h(\text{key}')$ key = key' Collision

Each bin is stored as a linked list.

Q. Suppose h is random function, and n keys are stored. What is the average ~~bin size?~~ bin size?

(R) $\Theta(1)$ ✓(G) $\Theta\left(\frac{n}{m}\right)$ (B) $\Theta\left(\frac{m}{n}\right)$

m bins

n keys distributed among m bins.

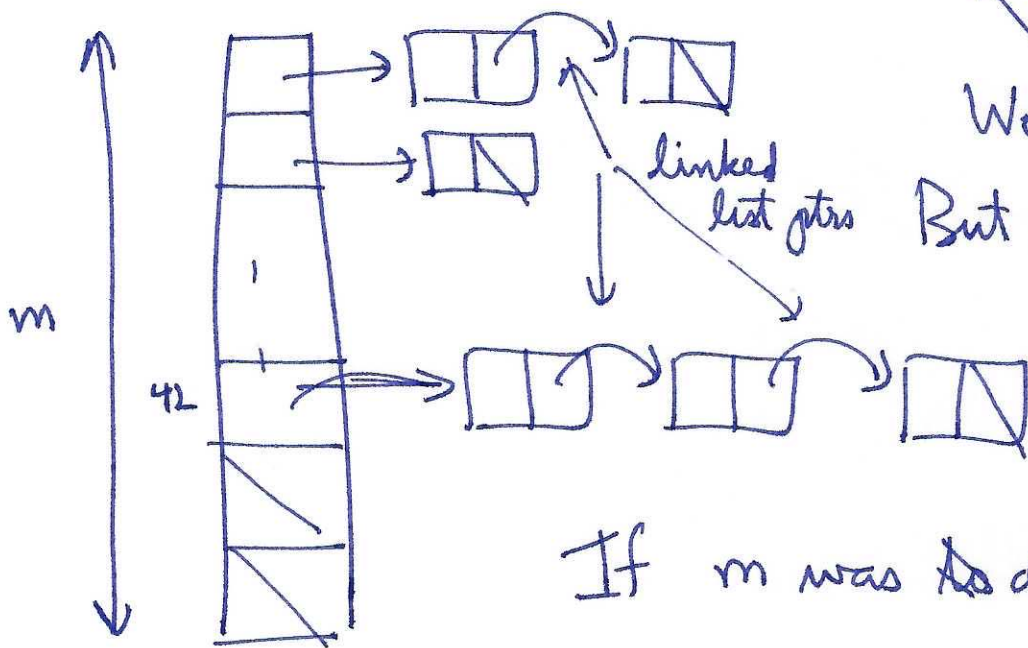
Each bin, on average, has n/m ~~bins~~ keys.

Hash table is an array of m linked lists.

insert/find/delete/update (key)   "the"

$h(\text{"the"}) = 42$

1. Go to the ~~linked list at~~ $h(\text{key})$ linked list.
2. Insert/find/delete/update key in that linked list



Worst-case is $\Theta(n)$

But if h was random, each list should have size $\Theta\left(\frac{n}{m}\right)$

If m was close to n $(m = \Omega(n))$,

$\Theta\left(\frac{n}{m}\right) = \Theta(1)$. Constant time for all operations.

More bins (m is larger)

$\Rightarrow$ Operations are faster

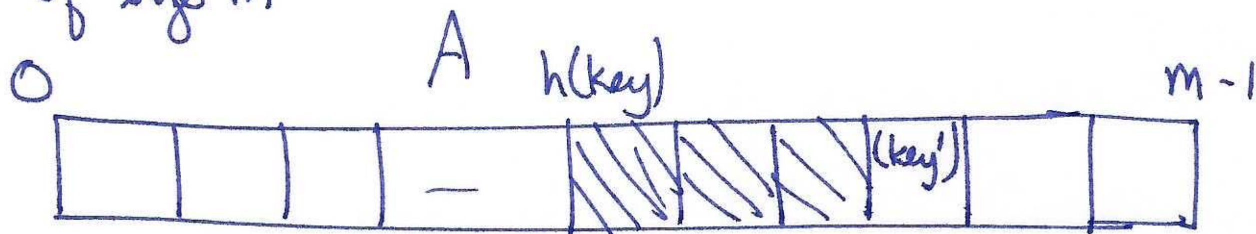
Chaining needs $m + n$ pointers/memory over storing the data.

m → head ptrs

n → linked list internal ptrs

Open addressing   Saves pointers. Less memory

Hash table is just an array of (key, value) pairs of size m



Store (key, value) at $A[h(key)]$

What happens if you insert (key', value') such that $h(key) = h(key')$?

Collision

Linear probing

Go to the next index, and try that.

Try $A[h(key)]$. Full / taken

$A[h(key)+1]$.

$A[h(key)+2]$. ... until you find an empty spot.

As long as $m > n$, there will be an empty spot.

Find(key)

1. Set $i = h(key)$

~~2. while (true)~~

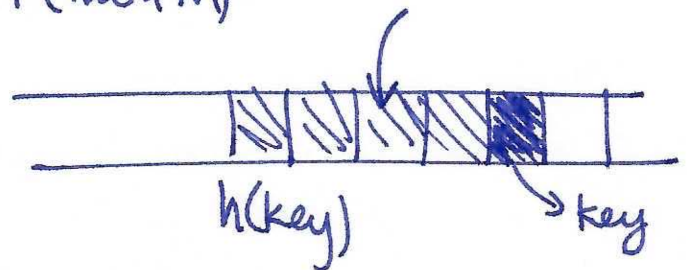
2. while (A[i] is not empty)

   (a) Check if key at A[i] is key we're searching for

   (b) If so, FOUND. So return True / A[i]

   (c) else, $i = i+1 \pmod{m}$

3. Return False



If key was inserted, then it must be between A[h(key)] and the first empty cell.

To delete, you cannot remove an element. Just mark it as deleted. COMPLICATED!

Punchlines:

Hash tables do insert/delete/find in $\Theta(1)$.

Randomized and memory intensive.

Arrays (sorted/unsorted)
Linked List
Stack
Queue
Binary Heap / Priority Queue ← Extract Max
BST (self Balanced / AVL / Red-Black) ← Ultimate data structure
Hash Tables

} Basic Data Structures

Know the operations supported by these data structures and their running times.

Other than self-balancing trees and hash tables, know how to implement all other data structures

---

Divide and Conquer

Technique to devise faster algorithms, using recursion

Given a problem (esp. on arrays)

(1) Divide into subproblems

(2) Solve each recursively

(3) Put together ~~sub~~ subsolutions to get solution for the main problem

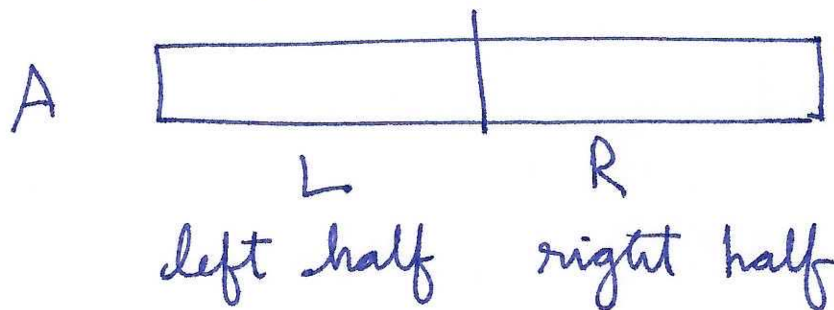
↑ You only need to figure out Step (3).

Mergesort:

(int)

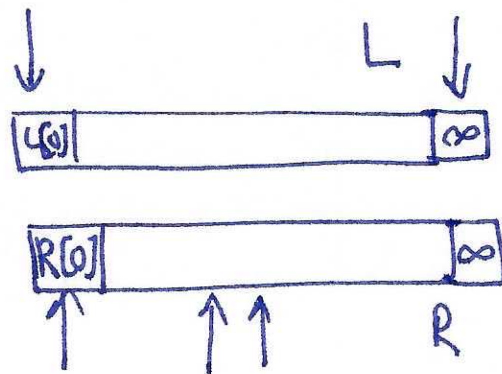
Sorting array of size n in $\Theta(n \log n)$.

(We have seen Heapsort which also runs in $\Theta(n \log n)$.)



Suppose L and R are already sorted

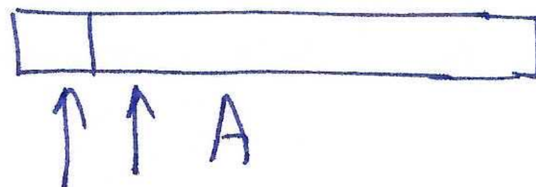
How to sort $A = L \cup R$ ← union



Merge(L, R)

↳ Takes two sorted arrays and output sort of union

1. Initialize $i_L = i_R = 0$
2. Initialize output array A of size $|L \cup R|$.
   Init $i_A = 0$



3. while("condition")
   (a) Compare $L[i_L]$ and $R[i_R]$ and store min as $A[i_A]$.
   (b) i_A++; (c) Increment either i_L or i_R, depending on min

"Condition": while we haven't reached end of L and the end of R