TUTORIAL 10.

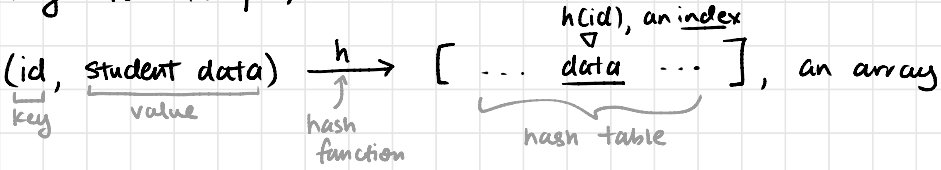
Recap: Hashing.

Hashing approximates the O(1) access times of an array, while allowing you to refer to values by some arbitrary **key** type instead of a regular integer array index. A data structure that uses hashing to store items like this is called a **hash table**.

To do this, we must define a **hash function** which maps arbitrary keys to integers in some range $\{0, 1, \ldots, N\}$, where N is the size of the associated hash table. This function should be *fast to compute* and it should try to *distribute its hash values as evenly as possible*.

$$h : \{\text{keys}\} \longrightarrow \{0, 1, \ldots, N\}$$

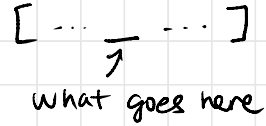
Using a function like this, we can then store (key, value) pairs by storing the items in an array, with index h(key). However, users of the hash table will just be able to look up a value directly by its key. For example,



Look up my student data using my zID instead of whatever index in the underlying array it is stored at

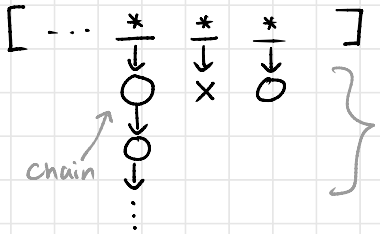
However there is an obvious problem: *what if there are more keys than indices available in the hash table?* It is inevitable then that any hash function is **not** 1:1, so we will get **hash collisions**:

$h(x) = h(y)$, but $x \neq y$!!!



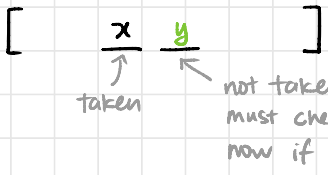
There are 3 main ways of dealing with hash collisions:

① **Separate chaining**: make each slot in the hash table a linked list, containing all of the values whose keys have the same hashes.



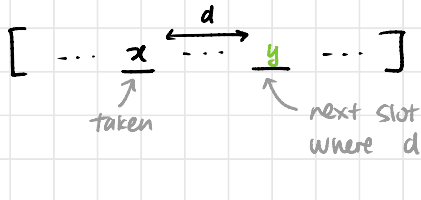
Pro: theoretically can store infinitely many values!
Con: potentially bad complexity if the chain searched is long. (Complexity can be measured by load factor; see lectures)

② **Linear probing**: when a slot is already taken, look for the next available slot to insert into. When searching, look for the value starting from the hash index until either an empty slot is encountered or you loop back to your starting point.



Must now be quite careful when deleting values.

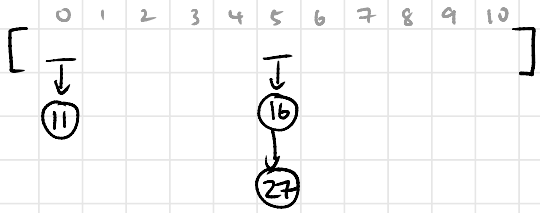
③ **Double hashing**: same as linear probing, except instead of moving in increments of 1, your increment size is determined by a 2nd hash func.



3. Insert 11 16 27 35 22 20 15 24 29 19 13

Hash function $h(x) = x \bmod 11$ = remainder when x is div. by 11

(a) Using separate chaining:



(b) Linear probing:

0	1	2	3	4	5	6	7	8	9	10
11	22	35	5		16	27	18	19	20	21

(c) Double hashing, with $h_2(x) = (x \bmod 3) + 1$:

0	1	2	3	4	5	6	7	8	9	10
11		35		22	16	27				

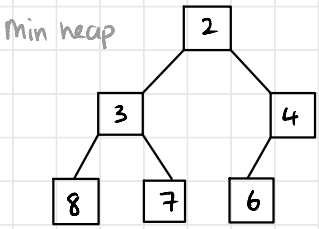
Recap: Heaps

Heaps are a special kind of tree that is ordered top to bottom.
There are 2 kinds:

1. Min heaps: each node is smaller than its children
2. Max heaps: " " greater " "

They are a very nice data structure to use as the impl. of priority q's.

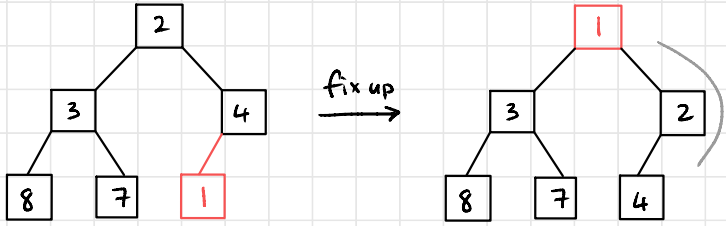
Heaps are usually stored as an array but visualised as trees. See the lectures for info about array representation.



heaps always fill from the bottom level and left to right

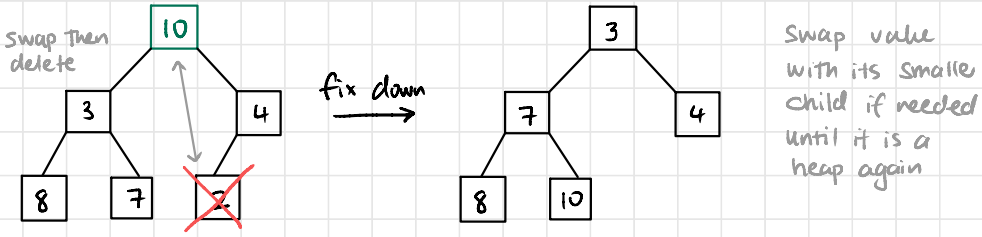
Insertion into a heap is a 2 step process:

1. Put the value down into the lowest, leftmost (free) position available
2. Do a fix up: keep comparing val with its parent and swap if the values are out of heap order; continue until you hit the root as necessary



Deletion in a heap is also a 2 step process. You always delete the root (i.e. top value):

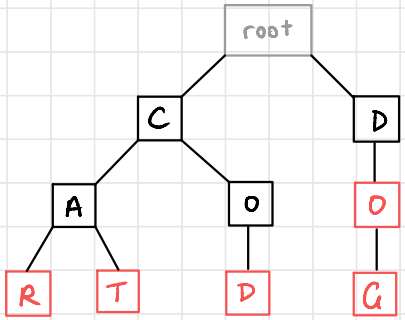
1. Swap the root val with the lowest, rightmost value, Then delete the old root
2. Do a fix down: compare the root with the smalle of its children and swap if necessary. Repeat until you hit a leaf.
   (This assumes a min heap: if max heap, compare/swap with larger child.)



Complexity of both operations is $O(\log n)$, where n = #. nodes.

Recap: Tries

Tries are a specialised tree structure which is used for efficiently storing strings/text. Each node has 1 child for all of the possible letters:



Red nodes = a word ends on that char (terminal node)

{car, cat, cod, do, dog}

A word is in a trie if there is a path with its letters that ends in a terminal node.

Insert into a trie by creating any necessary nodes missing in the trie and then set the last node to terminal.

11. Insert so boo jaws boon boat axe jaw boats sore