CSC263 Notes

Data Structures and Analysis

https://github.com/ICPRplshelp/

Last updated February 14, 2023

1 Data Types, Data Structures

ADTs

- Specification
  - Objects we're working with
  - The operations (WHAT but not how)

Data structures

- Implementation (how)
  - Data
  - Algorithms

Analysis (runtime or complexity)

- Worst case
- Best case
- Upper bounds
  - $\mathcal{O}$
- Lower bounds
  - Ω
- Tight bounds
  - Θ

If we have algorithm A and input x, the runtime $t_A(x) =$ number of constant time operations independent of x.

Ultimately, we want a measure of running time that is a function of the input size. We have lots of inputs for each input size. So, if we want to prove an upper bound when looking for the worst-case running time:

- I need two functions to prove an upper bound
  - A simple algebraic expression
  - The running time
    * However, the pure runtime function's codomain isn't $\mathbb{R}^{\geq 0}$ – but rather, a list of running times. To turn it into a raw function that outputs $\mathbb{R}^{\geq 0}$, we can take the largest of the list I just described.

Worst case is just us narrowing down a bunch of possible runtimes to the worst one.

My upper bound will always be some value that is larger or equal to the worst case, and the lower bound must be below the worst case but not all the worst cases.

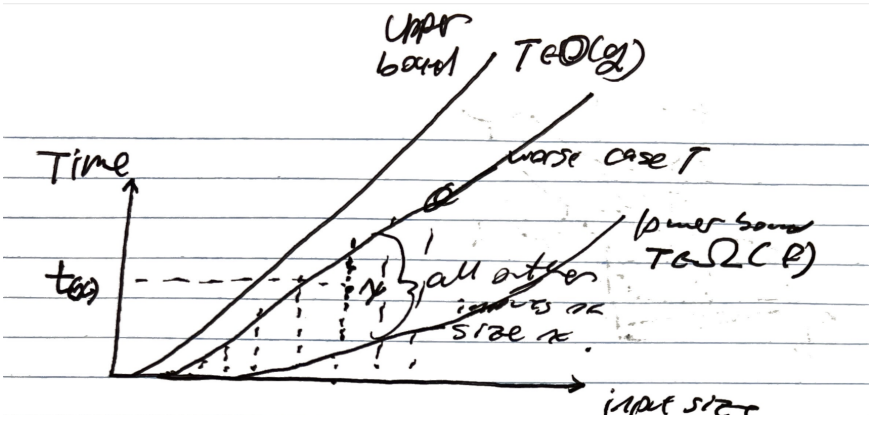

Figure 1: Upper and lower bounds of the worst case

1.1 Average-case Running Time

For a particular input size n, I have all these inputs x. We need to precisely define $S_n = \{\text{set of all inputs of size } n\}$. We need a probability distribution over our set of inputs.

- For each input $x \in S_n$, $t(x)$ is a random variable.
  - It's something that assigns a number to each element of our sample space. Now, S_n (a finite set) becomes a sample space.

Given a **discrete** probability distribution over S_n, $t(x) =$ the running time for input x. But what we'll get is that if I only know the input size, $t_n =$ **average-case running time for size n is:**

$$t_n := E[t(x)] \text{ over } S_n$$
$$= \sum_{x \in S_n} t(x) \Pr(x)$$

This summation is not as easy to figure out. There's a way we can get a particular value, but there is another way:

EX: The linear search algorithm

```
1 def LinearSearch(L: LinkedList, x: T) -> Node | None:
2     """Pre: L is a linked list, x is a value
3     Post: return the node that contains x
4     or none otherwise"""
5
6     z = L.head
7     while z != None and z.key != x:
8         z = z.next
9     return z
```

Standard linked list search.

Here's what we need to do:

Define our sample space (a family of sample space, one for each sample size):

Let n be arbitrary.

$$S_n = \{\text{every input of size } n\}$$

What should the sample space be? It shouldn't necessarily have infinitely many inputs with a given running time.

The number of different possible running times I have is finite for this algorithm.

INSIGHT. One representative input for each possible behavior. Behavior = running

time.

$$\begin{aligned} &S_3 \\ &= \{([1, 2, 3], 1), ([2, 1, 3], 1), ([2, 3, 1], 1), ([2, 3, 4], 1)\} \end{aligned}$$

Alternatively, $S_n = \{([1, 2, \ldots, n], 1), ([1, 2, \ldots, n], 2), \ldots\}$

Which is $\{([1, 2, \ldots, n], x) : x \in [0, n] \wedge x \in \mathbb{N}\}$

The probability distribution becomes important. How do we decide how likely we want each input to be? How can we tell? That is a tricky question. What are we trying to do, and there's no obvious way to choose. In practice, in any kind of real-life scenario, if you want to judge how well an algorithm performs on average if you have some idea of what your real-life inputs are going to look like. If you have no information at all, where it is all abstract, then we'll just uniformly distribute.

$\Pr([1, 2, \ldots, n], i) = \frac{1}{n+1}$ for $i = 0, 1, \ldots, n$

Now we have this, we can calculate the expected value:

$$E[t(x)] = \sum_{(L, i) \in S_n} t(L, i) \cdot \Pr(L, i)$$

$$= \sum_{i=0}^{n} t([1, 2, \ldots, n], i) \cdot \frac{1}{n+1}$$

When we're doing an average case, we cannot calculate an expected value with $\mathcal{O}$ expressions in there. We need a precise expression we can add up and average out. We need an **exact** expression

- Not in the sense that there's one right answer, but we need to fix a particular way of counting and count the same way for every input.

There is one trick: pick some representative operation that we know if we count that, the number of representative operations is $\Theta(\text{runtime})$.

In the example, the number of times `z.key == x` is run, which will be the **representative operation**

Ignore the constant time operations. The thing that matters is the loop. The loop does a constant amount of work each operation.

$$= t([1, 2, \ldots, n], 0) \cdot \frac{1}{n+1} + \sum_{i=1}^{n} i \cdot \frac{1}{n+1}$$

It's all algebra by this point. Do all of it, and you should end up with $\frac{n}{2} + \frac{n}{n+1} \in \Theta(n)$.

2 Priority Queues and Heaps

In a priority queue, we store a collection of elements. We're relying on **one** characteristic:

- Each element in the priority queue comes with its **own** priority attached to it (has something that makes each object sortable).
  - `x.priority` returns a comparable value
    * Integers are a good stand-in; however, we could use tuples or lists as a tiebreaker – the exception is that they are reversed for the context of this course. Hence, the last element takes the most precedence.
  - We don't care about its implementation. There is some built-in mechanism in the object that allows me to know its priority in constant time.

The **operations are the following:**

- `INSERT(Q, x):` add `x` to `Q`
  - Multiple elements can have the same priority.
  - If an object has multiple priorities, priorities assigned to the higher index should take precedence
- `MAX(Q):` return an element with the maximum priority.
  - If there are ties, we don't care which one is returned. Any of them could be returned. `Q` remains unchanged; this operation only queries our ADT.

- `EXTRACT_MAX(Q):` remove and return the element with the maximum priority.

The ordering for priority queues in this course will not be following the **first-in first-out** model. Drop the notion of a regular queue.
Of course, you could add a timestamp or insertion order as a tiebreaker.

2.1 How do we do it?

For the purposes of this course:

- An array will refer to an array-based list.
- A list will refer to a linked list.

The simplest data structures:

Unsorted array/list

- `INSERT`: $\mathcal{O}(1)$
- `MAX`: $\Omega(n)$
- `EXTRACT_MAX`: $\Omega(n)$
- We could do better.

Sorted array/list:

- `INSERT`: $\Omega(n)$
  - Yes, even for array-based lists. No loopholes.
- `MAX`: $\mathcal{O}(1)$
- `EXTRACT_MAX`: $\mathcal{O}(1)$

Here, $\mathcal{O}$ means "good news", and Ω means "too bad." It is simply emphasis. Everything is Θ, and all the time complexities here are **worst-case time.**

2.2 Max-Heaps

Intuition: partially sort

Structure: "almost complete" binary tree. **Everything full**, except maybe the last. On the last level, all leaves are **as far left as possible.** This **MUST** be always preserved throughout the lifetime of the max-heap.

In practice, when you have an almost-complete binary tree, the way that this is stored in memory, they generally mean a **list (or an array). They are listed in the order you would traverse then in a BREADTH FIRST SEARCH.**

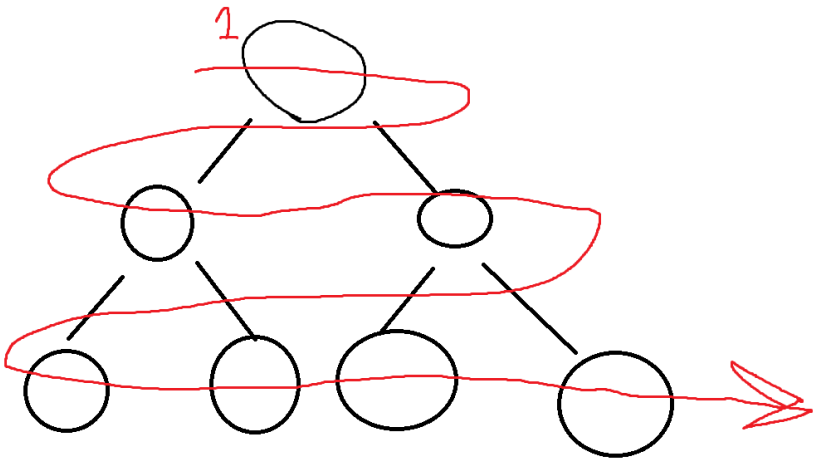

Figure 2: Indexing a heap

THE HEAP ELEMENTS START AT INDEX 1, AND WE SKIP OVER INDEX 0. Index 0 stores an empty item which we will not consider the root.

Navigation (USES BFS-LIKE ORDERING FOR INDEXING OF THE HEAP):

- For each node at index i in an array
- Parent$(i) = \lfloor \frac{i}{2} \rfloor$
  - This operation can be done extremely fast using bit shifts. That is, multiplication or floor division by a power of 2. In practice, this is implemented that way.
- LeftChild$(i) = 2i$
  - Previous item in size if it exists, otherwise we can't say anything
- RightChild$(i) = 2i + 1$
  - Next item in size if it exists, otherwise we can't say anything

Because we're working with an array (really), when we insert, we're going to add at the end. But it will mess up the ordering of the element. The same thing applies when we do extract max. We need to make sure that there are no gaps in the tree at any time.

2.3 Max-Heap Order

This is not a binary search tree. There is **no** left-to-right ordering. The only kind of ordering we have in a heap is the top-to-bottom ordering.

This property must hold for all max heaps.
EVERY NON-LEAF NODE STORES AN ELEMENT WITH PRIORITY $\geq$ THE PRIORITIES OF THE ELEMENTS IN THE NODE'S CHILDREN.
No required ordering between siblings. This means if I reflect the heap on the vertical axis, it shouldn't break this property.

View max-heaps as stairs: when you vertically go down, you should step downwards.

The highest priority of the heap is at the top. Finding the max is always easy: it is **always the top element, at index 1 (the first index).**

2.4 Heap Insertion

Insert at the end and try to push it up.

Insert any item into the heap.

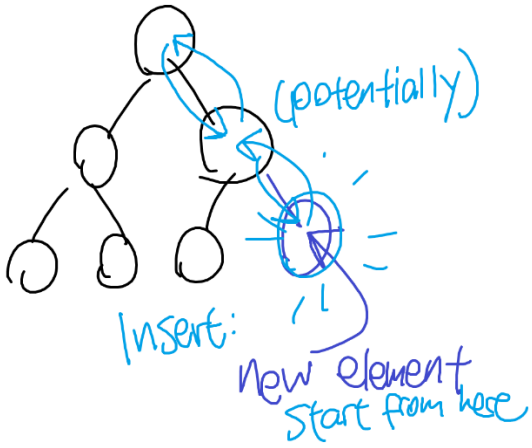

Figure 3: How insertion feels like

Do whatever we need to do to change things and preserve the entire tree structure. For `INSERT(n)`:

- Firstly, add `n` to the end. Our array would be: `[__, ..., n]`
  - Normally has extra space at the end due to array size inconsistencies
- `n` is a leaf. The only place that might be an issue is with the node and the parent.
- We swap the position of n with its parent, if $n >$ its parent.

- Then, check n with its parent (if it exists) and do the same thing as the previous step, again. Repeat that all the way up.

> Any point where `n.priority` $>$ `p.priority`, we swap. Keep doing this until that isn't the case.

Why does this not cause a problem with the neighbors: if s is a neighbor of X originally? Because $p \geq s$, and if $n > p$, then $n \geq s$.

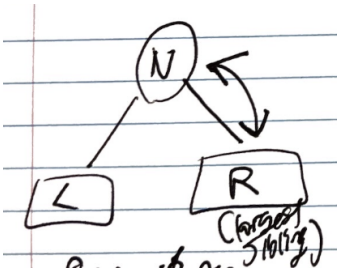

Figure 4: Largest sibling swap case

2.4.1 Heap Insertion Running Time

Constant work per level of the tree. We have a complete, balanced binary tree. The running time is $\Theta(\lg n)$, the height of the tree, for worst case.

2.5 Heap Extract Max

Move the last element to the top and heapify it (push it down).

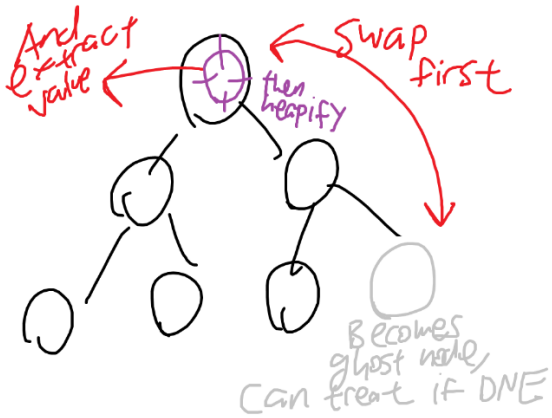

Figure 5: How extract max feels like

Remove and return the element in the heap with the highest priority. Procedure goes as follows:

- Decrease size of heap by 1: `H.size -= 1`
- Swap max value (index 1) with smallest value (index `H.size + 1`, the previous `H.size` before subtracting by 1)
- Perform Heapify on the root of the heap, now at index 1

2.6 Heapify

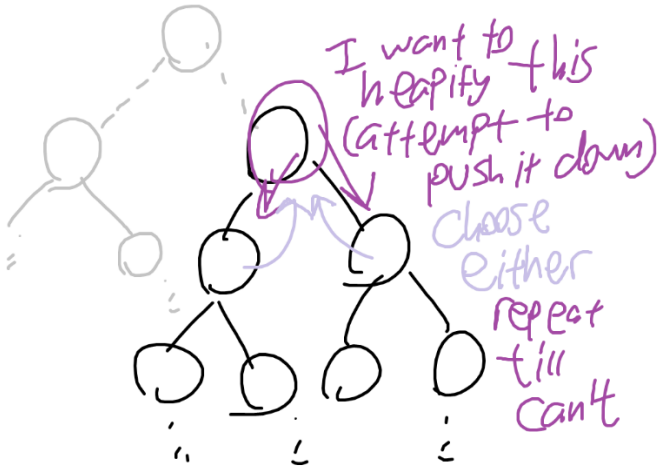

Figure 6: How it feels to heapify

Repeatedly swap element with its child of higher priority than the element until both children have smaller or equal priorities, or if I'm at a leaf.

Runtime is $\mathcal{O}(\lg(n))$. We do have to check all children (twice for a binary heap).

Heapify should only really be called if all subtrees are proper heaps.

2.7 Building a Heap

Turn the unsorted array into a heap, then run heapify on every index from right to left, starting at $\left\lfloor \frac{\text{len}(A)}{2} \right\rfloor$.

I have a whole collection of objects, and I want to put them into a heap.

- Start from collection of S
- Create a heap with elements of S

IDEA 1

One idea is to run `Insert` for each element of S into the heap. The runtime is $\Theta(n \lg n)$ in the worst case.

IDEA 2

If $S = [\_, e_1, e_2, \ldots, e_n]$, this already can represent a binary tree structure (with on constraints). Instead of starting with an empty list and copying elements into the new list one-by-one, I'll work directly with the new list and re-order the elements. I'll work from the bottom up, with all the leaves.

The leaves in my list correspond to **half** of my values of the array. For any value in the second half of the list, if I multiply the index by 2, I'm out of the list anyway.

Start with the furthest node that isn't a leaf, at index $i = \lfloor \frac{n}{2} \rfloor$, and work backwards. The roots of its subtrees are all correct heaps. Run `Heapify` on that node, and for each index.

> For each index in order from $\lfloor \frac{n}{2} \rfloor$ to 1, call `Heapify`.

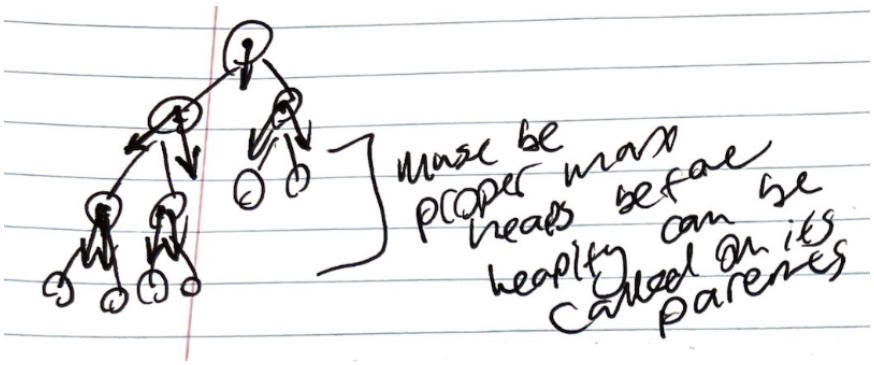

Figure 7: The idea of building a heap

2.7.1 Running Time of Building a Heap, Idea 2

In a complete binary tree with n nodes (assuming n is a power of 2):

- $\frac{n}{2}$ of them are leaves. We don't need to do work for any of them

- $\frac{n}{4}$ of them have height 1, and requires a max. of 1 level swap
- $\frac{n}{8}$ of them have height 2, and this requires a max. of 2 level swaps
- $\vdots$

As I go up the tree, the number of nodes I need to do go down quickly.

The total work is $n\sum \frac{i}{2^i} = \Theta(n)$.

2.8 Heap Sort

Suppose I have this list:

$$[4,\ 3,\ 7,\ 1,\ 8,\ 5],\ \text{len} = 6$$

I want to sort this list using heapsort. The first thing that heapsort does:

1. Build a max heap from the array: $\Theta(n)$
   a. $\rightarrow [4,\ \mathbf{8},\ 7,\ 1,\ \mathbf{3},\ 5] \rightarrow [\mathbf{8},\ 4,\ 7,\ 1,\ 3,\ 5]$
2. Call `EXTRACT-MAX` $n-1$ times: $\Theta\left(n\log(n)\right)$
   a. $\rightarrow [\mathbf{7},\ 4,\ \mathbf{5},\ 1,\ 3,\ |\ 8]$. All items on the right of the $|$ is not part of the heap, as signaled by the length of the heap, which **must** be tracked. `INSERT` overwrites the junk area, but I won't be running that.
   b. $\rightarrow [\mathbf{5},\ 4,\ \mathbf{3},\ 1,\ |\ 7,\ 8]$
   c. $\rightarrow \vdots$

After the $n-1^{\text{st}}$ `EXTRACT-MAX` call, I would already end up with a sorted list. The time complexity is $\Theta\left(n\log(n)\right)$

3 AVL Trees

A dictionary is a **set** where each element has a **unique** key: `x.key`. This means objects held in the dictionary must have its key.

Two objects can be the same for everything except its key, and a set can contain both, as they won't be equal.

The operations:

- `SEARCH(S, k)`: return element $x \in S$ with `x.key == k`. Or `NIL` if I can't find any (programming language agnostic `NULL`).
- `INSERT(S, x)`: Add x to S. If S contains element y with `y.key == x.key`, remove the old element y and replace it with x.
  - This means I have to actively look for duplicate keys to avoid duplicates.
- `DELETE(S, x)`: remove element x from S.
  - NOTE: `DELETE(S, SEARCH(S, k))` is used if you want to delete something based on a key. This implementation of `DELETE` prevents issues of needing to find a key given an element.

3.1 Data Structures / Implementations

Note that I must be able to access the length of the array in constant time. If something is sorted, the keys must be comparable. Linked lists are doubly linked lists.

Structure / OP	`SEARCH(k)`	`INSERT(x)`	`DELETE(x)`
Unsorted array	$\Theta(n)$	$\Theta(n)$ – inspect keys	$\Theta(1)$ (If I know the index. Memory leak?)
Sorted array	$\Theta(\lg(n))$	$\Theta(n)$ – array issues	$\Theta(n)$

Structure / OP	`SEARCH(k)`	`INSERT(x)`	`DELETE(x)`
Unsorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Sorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Direct access table (memory hog)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hash tables	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binary search trees	Height $\mathcal{O}(n)$	Height $\mathcal{O}(n)$	Height $\mathcal{O}(n)$
Balanced search trees	$\Theta(\lg(n))$	$\Theta(\lg(n))$	$\Theta(\lg(n))$

3.2 Binary Search Trees

Our implementation is recursive, and we're going to write them in a style, as it makes it a lot easier to talk about balancing them.

ASSUMPTIONS:

- BST nodes store the following:
  - item
    - key (required to implement dictionary)
    - value
  - left
  - right
- Dictionary only stores the root: `S.root`

The operations go as follows:

- `INSERT(S, x):`

- `S.root = BST_INSERT(S.root, x)`. This is a recursive helper and returns the root of the resulting tree.

Helper functions:

```
def BST_INSERT(root, x):
    """Add x to the tree starting at root.
    Return the root of the tree afterwards.
    """
    if root == NIL:
        root = BSTNode(x)
    # ensure recursive cases happen first
    elif x.key < root.item.key:  # INSERT LEFT
        root.left = BST_INSERT(root.left, x)
    elif x.key > root.item.key:  # INSERT RIGHT
        root.right = BST_INSERT(root.right, x)
    else:  # x.key == root.item.key
        # remove the old item to prevent
        # key duplicates
        root.item = x
    return root
```

3.2.1 Runtime For Binary Trees

For a BST, worst case is $\Theta(n)$. When I have a balanced BST, the runtime returns to $\log(n)$. This calls for balanced search trees.

3.3 AVL Trees (Balanced Binary Trees)

An AVL tree is a balanced binary tree.

How do we balance trees, and how do we fix things when things go out of balanced?

Something we do to a structure of a binary tree by rearrange a few references. We can move things around in a BST of it such that the tree structure changes, but the ordering of the values do not change.

We can use tree rotations to do this:

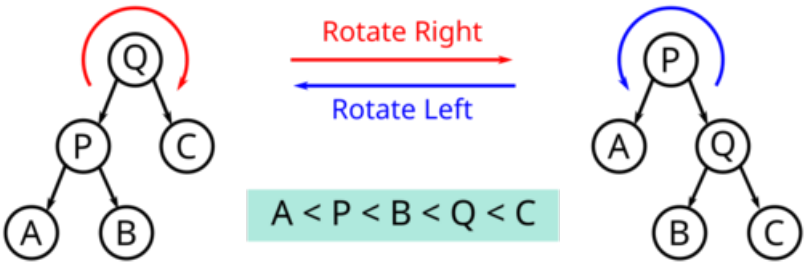

Figure 8: Tree rotations. Image from Wikipedia, see Binary Tree Rotations

When do we use this?

- The BST search algorithm doesn't need this.

We're not going to try to keep the tree perfectly always balanced. Doing so will require to move a lot of nodes at once. We need the tree to be roughly balanced.

Empty subtrees have a height of -1. They refer to `NIL`.

Completely balanced means

- All subtrees have the same height

Approximately balanced means:

- For each node:
  - `Height(left subtree)== Height(right subtree)` ± 1
  - Allow an error of 1

Property: Binary trees that are approximately balanced have height of $\Theta(\log(n))$.

So how are we going to insert:

1. Insert like normal, ignoring the fact that the insertion needs to be balanced.
2. Starting at the insertion point and work up the tree, using rotations to fix balance where needed.
3. If the left side is heavier, try to rotate it to the right. If the right side is heavier, try to rotate it to the left. You may need to double-rotate.

Tree height must be kept updated during operations, and during rotations. This can be done in constant time per node on the path from the root to the point in the tree where we made the update.

3.4 Double Rotation

To ensure that rotations keep the tree AVL-balanced, we may need to double rotate once. This occurs when a zig-zag forms when you're traversing from the root, but only targeting the subtrees of the greatest height. You **must** do this otherwise you will be caught in an infinite loop. GET RID OF ZIG-ZAGS BEFORE ROTATING

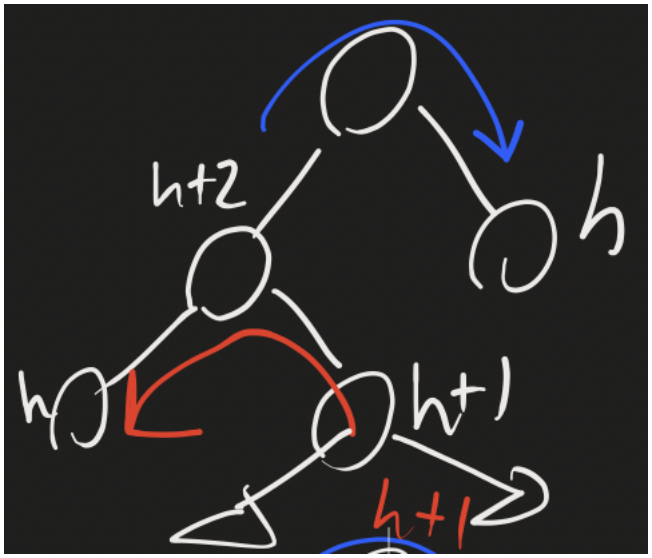

Figure 9: Left then right rotation

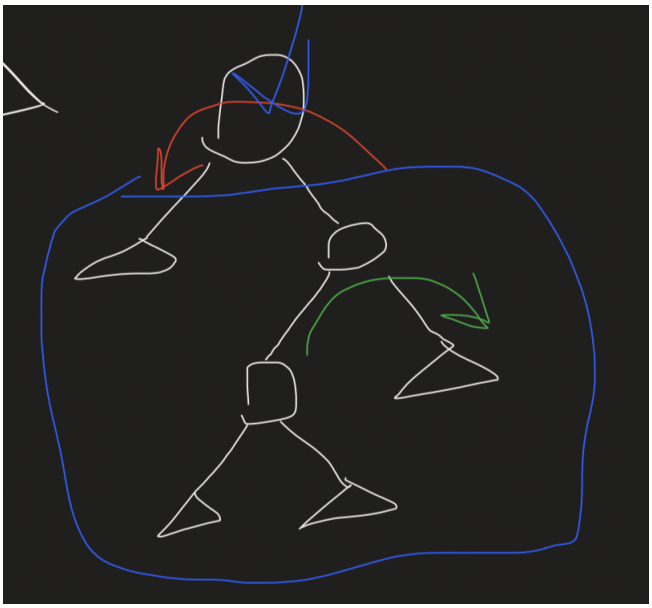

Figure 10: Right then left rotation. The root node is the target of rotation prior to being rotated. Everything in the large blue rectangle must be almost balanced, as a precondition.

3.5 Implementation of Search

Search is **identical to binary tree search.**

```
1 def AVL_SEARCH(root, k):
2   """Return the node that has key k"""
3   # identical to BST search
```

3.6 Implementation of Insert

Insertion (append to a leaf or an incomplete node):

```
1 def AVL_INSERT(root, x) -> AVLNode:
2   """Return the new root node after
3   insertion"""
4   if root = NIL:
5     root = AVLNode(x)
```

```
    # .item = x, .left = .right = NIL
    # .height = 0

  elif x.key < root.item.key:

    root.left = AVL_INSERT(root.left, x)
    root = AVL_CHECK_AND_BAL_RIGHT(root)
    # check and rebalance

  elif x.key > root.item.key:

    root.right = AVL_INSERT(root.right, x)
    root = AVL_CHECK_AND_BAL_LEFT(root)
    # check and rebalance

  else:  # x.key == root.item.key
    root.item = x
  return root
```

Check balance for one direction:

```
def AVL_CHECK_AND_BAL_LEFT(root) -> AVLNode:
  """When we have a node, and we want to
  possibly rebalance to the left (currently,
  the right MIGHT weight more)

  Return the new root of the tree.

  Pre: root != NIL and all subnodes are AVL-balanced
  """
  # recalculate the height of the root
  root.height = 1 + max(root.left.height,
                        root.right.height)
  if root.right.height > 1 + root.left.height:
    # the right subtree is too tall
    # check for double rotation
    if root.right.left.height > root.right.right.height:
       root.right = ROTATE_RIGHT(root.right)
    root = ROTATE_LEFT(root)
  return root
```

To account for `NIL`:

Every time I use DOT SOMETHING, then I need to ensure it's not NIL. I'll have more special cases than the actual code. Here's what I'm going to do:

We already have our dictionary storing a root. When I create a dictionary, I'll use a **NULL OBJECT.** This allows me to get around the need to do null checks, and the interface of the node will work perfectly.

We'll create a special node, `S.NIL = AVLNode()`, where

- We don't care about the item
- `S.NIL.left = S.NIL`
- `S.NIL.right = S.NIL`
- `S.NIL.height = -1`

No more `NullPointerException`s.

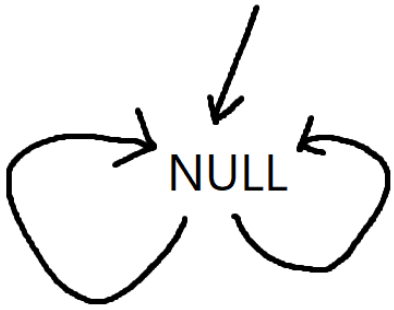

Figure 11: Null points to itself.

3.7 Implementation of Delete

This goes as follows:

```
1 def AVL_DELETE(root, x) -> AVLNode:
2     """Delete x from subtree at root (of the tree).
3     Return the new root of the tree.
4
5     The tree must be balanced afterwards and the height
```

```
    must also be rebalanced.

    Precondition: x in root (but the code
    doesn't break otherwise)
    """
    if root == NIL:
        # BRANCH SHOULD NOT HAPPEN due to preconditions
        pass  # x isn't in the tree at all
    elif x.key < root.item.key:  # <--
        root.left = AVL_DELETE(root.left, x)  # the
           subtrees are balanced
        # this causes the right side to potentially be
           heavier
        root = AVL_REBALANCE_LEFT(root)  # this
           recalculates the height for the root
        return
    elif x.key > root.item.key:  # -->
        root.right = AVL_DELETE(root.right, x)
        root = AVL_REBALANCE_RIGHT(root)
    else:  # x.key == root.item.key
        # one children case
        if root.left == NIL:
            root = root.right
        elif root.right == NIL:
            root = root.right
        else:  # two children case
            # replace x with predecessor (left) or
               successor (right)
            # whichever subtree is taller
            if root.left.right > root.right.height:  # LEFT
                HEAVY
                # find the largest value in my left subtree
                root.item, root.left = AVL_DEL_MAX(root.
                   left)
            else:
                root.item, root.right = AVL_DEL_MIN(root.
                   right)  # assume this updates ht
            root.height = 1 + max(root.left.height, root.
               right.height)  # update height
```

No rebalancing is required for the last branch because we'll always run `AVL_DEL_MAX/MIN` on the heavier subtree. So, this could either equalize the

balance, or make one side lighter by 1 than the other, but never more than 1.

The delete max helper goes like this:

```
def AVL_DEL_MAX(root) -> tuple[item, AVLNode]:
    """Delete the maximum value of this tree.
    Return the item that was deleted and the new root
    replacing it (potentially NIL)

    ENSURE the tree is balanced after removal.

    Preconditions: root != NIL, all subtrees
    are balanced
    """
    if root.right == NIL:  # base, we are AT the max
        return root.item, root.left  # left could be NIL
    else:
        item, root.right = AVL_DEL_MAX(root.right)
        root = AVL_REBALANCE_RIGHT(root)  # when we work
           our way back, we rebalance
        return item, root
```

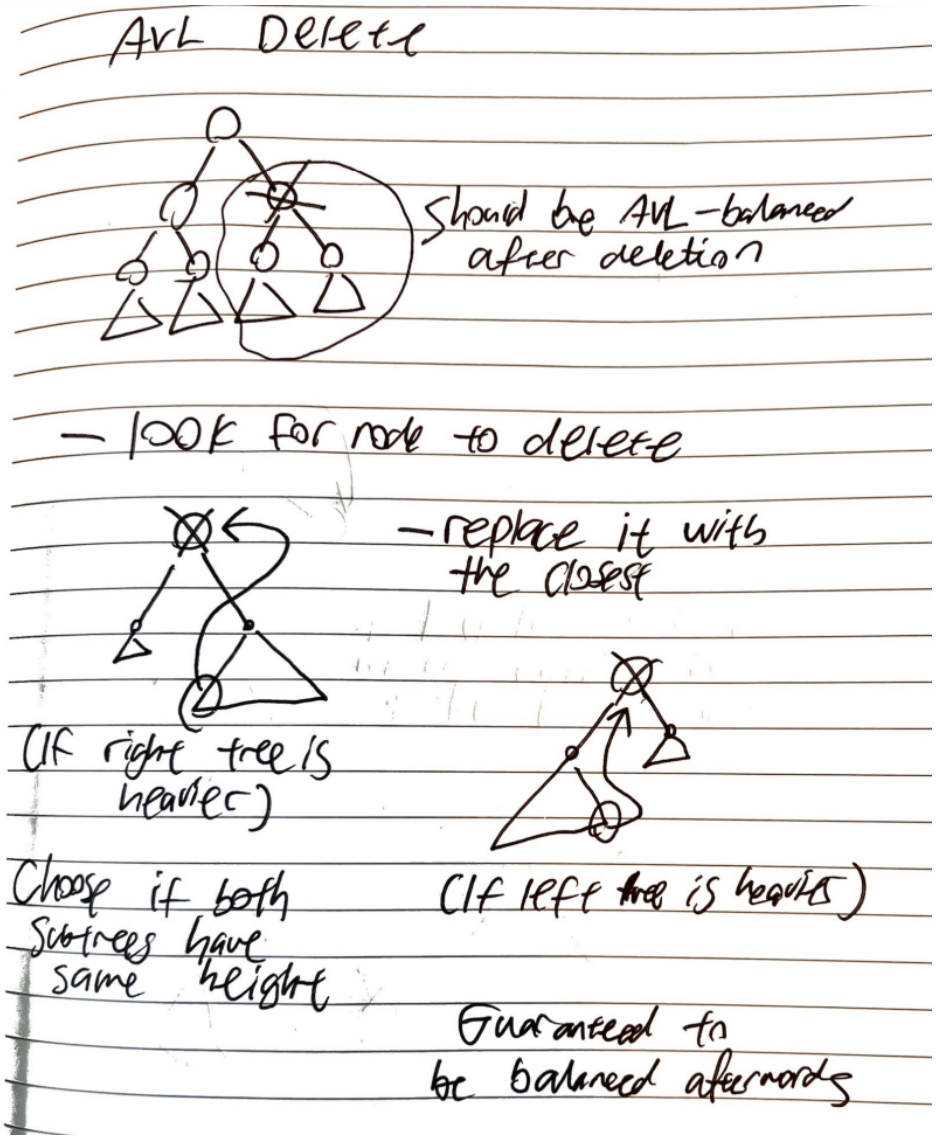

Figure 12: On AVL deletion

3.8 Runtime for AVL Tree Operations

- Performing a single rotation relinks some values and recalculates height, so it's constant time: $\Theta(1)$.
- **ALL OPERATIONS TAKE** $\Theta(\text{height}) = \Theta(\lg(n))$, **WHICH IS LOGARITHMIC TIME**

3.9 Correctness of AVL Tree Operations

A lot of the correctness comes for the BST code that forms the basis of the AVL code. The AVL code is responsible for balancing stuff. The rebalancing boils down to:

- Rebalancing works because the subtrees are always correctly AVL balanced the time we come back from the recursion.

4 Augmentation

We have a new problem that no existing data structure is entirely satisfactory. There are steps from doing this:

1. Start from a known data structure
2. Give new information to each node (e.g. AVL trees store the height in each node)
3. Adjust operations to keep the new information up to date
4. Implement any new operations

An AVL tree is an augmentation of a BST. We took BSTs, kept track of the height, and adjusted things. Many of the challenge questions in the problem sets are exactly that.

4.1 Constructing an Ordered Set ADT

NOT the order of insertion, but a set that always sorts itself. S is the set and n is its size.

- Objects: set of elements that are comparable with all of each other (NO DUPLICATES)
- Operations: They are like a dictionary; except every element we store is a key. We can store keys directly in tree nodes.
  - `SEARCH(x)`

- `INSERT(x)`
- `DELETE(x)`
- `RANK(x)`
  * Return the rank of an element x. It is the position of x in the sorted order of the elements.
  * If `RANK(x)` $= k$, then x is the x-th smallest element in the set. The smallest element is rank 1 and the last element is rank n, if we have n elements. Precondition: $x \in S$
  * If I know a value in my set, I might be curious to know where is that value situated compared to other values in the set?
- `SELECT(k)`
  * Return the element with rank k. Precondition: $1 \leq k \leq n$

4.2 How do we implement this?

IDEA 1: AVL tree with no augmentation

- SEARCH, INSERT, DELETE take $\Theta(\log(n))$ worst case.
- RANK:
  - In order traversal to count the number of values smaller than x, taking worst case $\Theta(n)$
- SELECT:
  - In order traversal required, taking worst case $\Theta(n)$

AVL trees work and allow us to do what we need to do but are not so efficient for our new operations.

IDEA 2: AVL augmented to store the rank of each node at the node, having an attribute `rank`

- RANK, SELECT, and SEARCH can be done in worst case $\Theta(\log(n))$ time as it can be done with a simple search and no need to track anything else.
- INSERT and DELETE requires me to update the rank of at most every single node, taking $\Theta(n)$ time worst case.

This doesn't work, but this is a case of "what if I can store the information I need, exactly?"

IDEA 3: Augment AVL with one new attribute (`size`) for each node, which gives me the number of nodes in all the subtrees (including the root, meaning leaves are size 1)

RANK now walks through the tree. As I'm looking for x, I can add over the size of the subtrees I've skipped. It goes like this:

`RANK(x):`

- Start with $r = 0$
- Search for x
- If I am about to recurse left:
  - Do nothing
- If I am about to recurse right:
  - Set $r = r + 1 +$ `root.left.size`
- If I find x:
  - Return $r +$ `root.left.size` $+1$ (as rank starts from 1, and I need to count my node due to that)

The worst-case runtime is $\Theta(\log(n))$.

Conceptually, at any point in time, r represents the total number of values SMALLER than the element (otherwise it would be off).

`SELECT( k ):`

- Start at the root

- If $k < 1+$ `root.left.size`:
  - Recurse left
- If $k > 1+$ `root.left.size`:
  - Recurse right with $k = k - 1 -$ `root.left.size`
    - (We need to update the relative rank of what we are looking at)
- If $k = 1+$ `root.left.size`:
  - Return `root.key`

The worst-case runtime is $\Theta(\log(n))$.

`INSERT(k):`

- Normal insert, and update `size` on the way back through the recursive calls. Note that leaves are size 1.
- If rotation occurs, I recalculate the size in constant time for the two pivot nodes in the rotation.
- The runtime is $\Theta(\lg(n))$.

5 Hashing

- **Context:** implementing dictionaries. `SEARCH, INSERT, DELETE`.
- **Ingredients**:
  - Keys belong to a LARGE universe U
  - Set up a hash table: array T (fixed size)
    - $m = \text{len}(T)$
    - Each $T[i]$ ($T[0], T[1], \ldots, T[m-1]$) is called a "bucket" / "slot" – something you can put in.

    * You're going to have the size of the array yourself. You can't tell ahead of time.
  - Hash function: $h : U \to \{0, 1, \ldots, m-1\}$
    * For all keys $k \in U$, $h(k)$ is the home bucket. It maps keys to array indices.

- **Recipes**:
  - `SEARCH(k):`
    * Look in $T[h(k)]$ (with the risk of collisions!)
  - `INSERT(x):`
    * Add or replace element in $T[h(x.key)]$
  - `DELETE(x):`
    * Remove the element from $T[h(x.key)]$

What can go wrong? Two keys going into the same slot?

5.1 Collisions

Motivation. If $|U|$ is small-ish, then set $m = \text{len}(T) = |U|$. Useful if I want to get the frequency of letters in a text file. No risk for collisions and everything takes constant time. **This is called a direct-access table, one location for each possible key.**

If U is large, I don't think I can create an array with a position for each 64-bit integer, then $m \ll |U|$ (orders of magnitude smaller). In that case, collisions are unavoidable.

Collisions: keys $k_1 \neq k_2$ such that $h(k_1) = h(k_2)$. Unavoidable when $m \ll |U|$. How do we deal with them?

1. Closed addressing (chaining)
   a. Each bucket $T[i]$ stores a linked list of elements.

b. If I know k, then I know that it has to be in the data structure stored at $T[h(k)]$

2. Open addressing

a. Each bucket $T[i]$ stores one element directly.

b. Key k may be stored somewhere else than $T[h(k)]$

The home bucket for each key is just the result of $h(k)$.

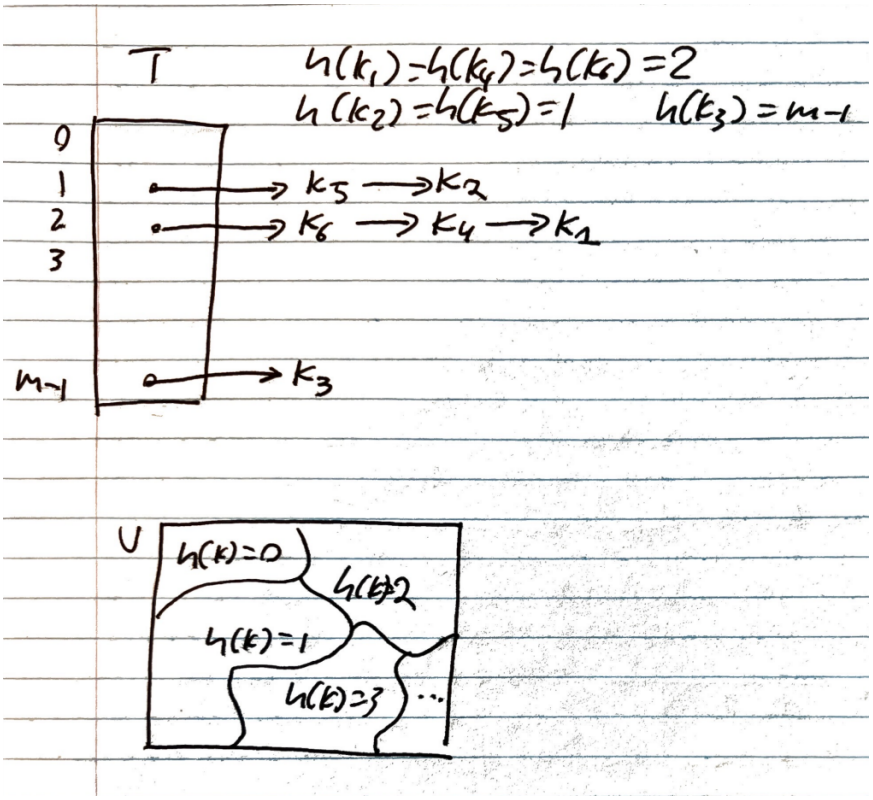

Figure 13: Collisions and Partitioning

5.2 Closed Addressing (Chaining)

Runtime?

- `DELETE(x):`

    - Once I found the linked list node
        * `SEARCH` for x
        * Remove node $\mathcal{O}(1)$

- `INSERT(x):`
    - Search for $x.key$
    - Add new node, or replace existing node in the linked list: $\mathcal{O}(1)$
- `SEARCH(k):`
    - Let $n =$ no. elements in T
    - Worst case: $\mathcal{O}(n)$, if all keys hash to same bucket, unless U is small

Insert and delete depend on search for runtime. Part of the work to do is part of search. The worst-case `SEARCH` time is $\Theta(n)$.

Log time is exponentially better than linear. In actual real use, the worst-case time $\Theta(n)$. It always works in apparently $\Theta(1)$ time.

We are used to performing worst case analysis to determine algorithm efficiency, most of the time. However, this is a case where the worst-case analysis tells us one thing, but our empirical testing tells us something else.

But worst-case is the worst case. It is true that for most algorithm, the worst case is representative of the typical behavior of the algorithm. The behavior is equally distributed between the best-case and the worst-case. So for most algorithms, if we go with the worst case, we wouldn't be far off from the typical behavior.

But for hashing, worst case happens so rarely that we'll need to look at the average case (expected). There is nothing in the analysis or the abstract algorithm that tells us that the average is better. That is an arbitrary decision. So, we need some evidence of how this plays out in practice to determine which analysis is better.

5.3 Average-Case Runtime of Search

To do an average-case analysis, I need to set up:

- A sample space (and what hash table?)
- Build a probability distribution
- Get its expected value

I'll set up the context:

- Hash table T of size n
- **Assumption**: hash function h partitions U (the universe) into roughly equal sized subsets, one for each bucket
- **Assumption:** Each key in U is equally likely (uniform distribution)
  - $\Pr[h(k) = i] = \frac{1}{m}$, choosing k at random from U. m is the number of buckets (simple uniform hashing assumption).
- If we insert n random keys into T
  - Let $L_i =$ no. of keys in $T[i]$ $\left(n = \sum_{i=0}^{m-1} L_i = L_0 + \ldots + L_{m-1}\right)$
- SEARCH(k) for random $k \in U$. How long is that going to take? Average case runtime? How many keys in T do I need to look at?
  - It depends on $N(k) =$ no. of keys examined during search for k
  - $N(k)$ is a random variable. I want to calculate $E[N(k)]$

Computing the expected value: Not by doing $E[N(k)] = \sum_{j=1}^{n} j \Pr[N(k) = j]$

Instead by doing:

$$E[N(k)] = \sum_{k \in U} \Pr[k] \cdot N(k)$$

Add up all the terms, one bucket at a time.

$$= \sum_{i=0}^{m-1} \left(\sum_{\substack{k \in U \ : \\ h(k) = i}} \Pr[k] \cdot N(k) \right)$$

$$\leq \sum_{i=0}^{m-1} \left(\sum_{\substack{k \in U \ : \\ h(k) = i}} \Pr[k] \cdot \underbrace{L_i}_{\text{no. of keys in bucket } i} \right)$$

$$= \sum_{i=0}^{m-1} \left(L_i \cdot \sum_{\substack{k \in U \ : \\ h(k) = i}} \Pr[k] \right)$$

$$= \sum_{i=0}^{m-1} \left(L_i \cdot \Pr\left[h(k) = i\right] \right)$$

$$= \sum_{i=0}^{m-1} L_i \cdot \frac{1}{m} \quad \text{from simple uniform hashing assumption}$$

$$= \frac{1}{m} \sum_{i=0}^{m-1} L_i = \frac{n}{m}$$

This has a name: $\frac{n}{m} = \alpha$ is called the load factor. This quantity represents, if I take n random keys and put them in the hash table under simple unform hashing:

- All keys are equally as likely to go anywhere
- With the probability of $\frac{1}{m}$, they would end up in the same bucket

- As you add more and more key, the probability goes up and eventually more things would go in the same bucket
- But if things were put in the hash table uniformly and randomly, if I put n keys into m buckets, $\frac{n}{m}$ is the average number of keys I would expect in each bucket.

5.3.1 Conclusion

If we make sure our hash table is large enough to have a location for each possible keys, or in other words, $m \geq n$, THEN on average, all operations take constant time (as the load factor is at most 1).

Therefore we don't bother to do anything more complicated than a singly linked list in each bucket.

If $n > m$, your runtime will depend on the load factor $\frac{n}{m}$. It will be greater than 1 here. Or we can prevent this by using a dynamic array.

All of this is an analysis for closed addressing (chaining).

5.4 Open Addressing

- Elements are stored directly in T
- We have a primary hash function $h_1 = h_1(k) =$ home bucket
- The way we hash is that we're going to use a probe sequence. When we get to a location in the hash table, and there is a collision, we are going to look at other spots in the hash table in a specific way.
- A probe sequence is what replaces the hash function: $h(k, i) =$ bucket to try after i collisions
  - Linear probing: $h(k,\ i) = (h_1(k) + i) \mod m$
    * Try the next bucket directly right. This means if we try to insert something and it collides, put it to the right if it is empty (and loop back if necessary).

    * Problem: could be responsible for more collisions. Long clusters can form with linear probing.

- Quadratic probing: $h(k,\ i) = \left(h_1(k) + ai^2 + bi\right) \mod m$, where $a,\ b$ are parameters are dependent on m.
    * Hopefully this spreads the values a bit more. If $a,\ b$ are not chosen carefully is if we roll around, we come back to locations we've been before and other locations we've never been to. Ultimate, this suffers from the same problems as linear probing: any two keys with the same home bucket will follow the same probe sequence.
- Double hashing: $h(k,\ i) = \left(h_1(k) + i \cdot \underbrace{h_2(k)}_{\substack{\text{secondary} \\ \text{hash function}}} \right) \mod m$
    * The first one h_1 tells me where to start, and h_2 tells me what to do if I encounter a collision.
    * In practice, this works quite well.

Why would somebody use this?