CSC263 Notes

Data Structures and Analysis

https://github.com/ICPRplshelp/

Last updated April 15, 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.

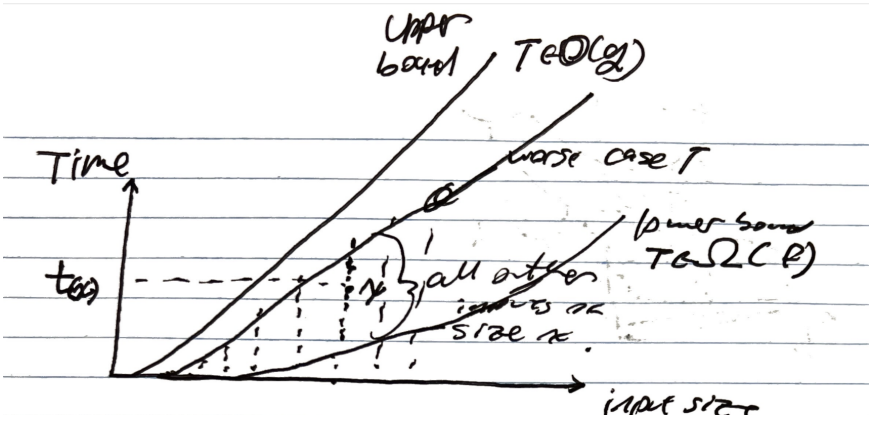

Figure 1: Upper and lower bounds of the worst case

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:

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

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

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.**

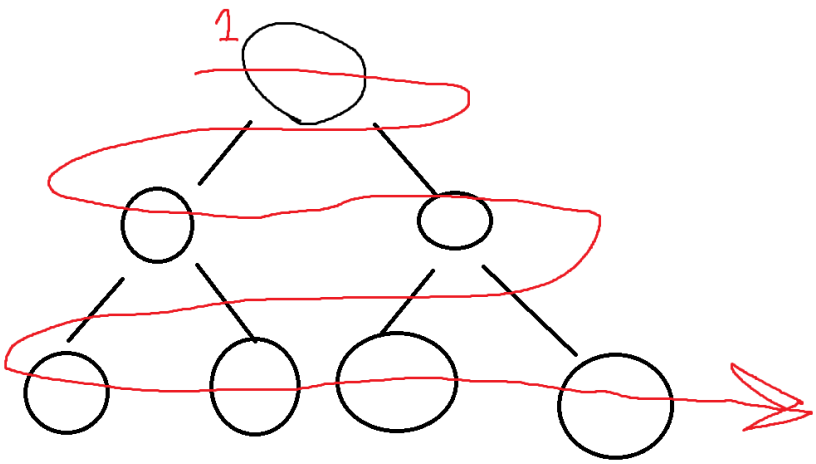

Figure 2: Indexing a heap

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.

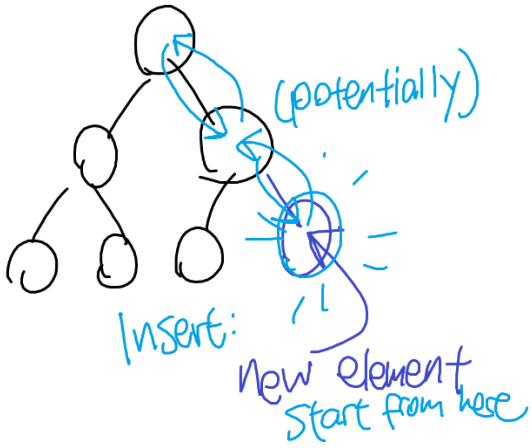

Figure 3: How insertion feels like

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 $n \geq s$.

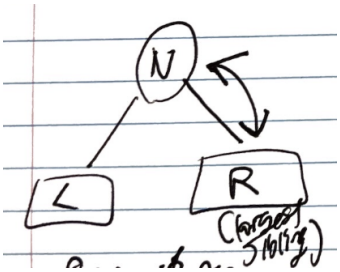

Figure 4: Largest sibling swap case

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).

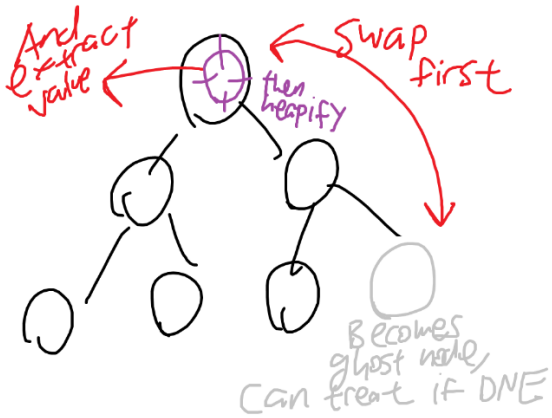

Figure 5: How extract max feels like

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

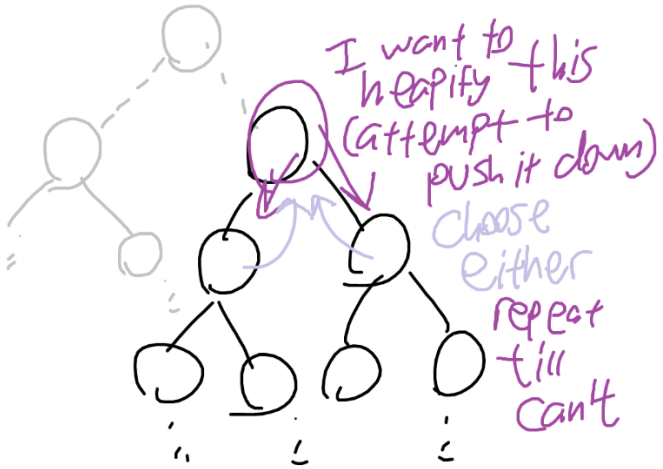

Figure 6: How it feels to heapify

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`.

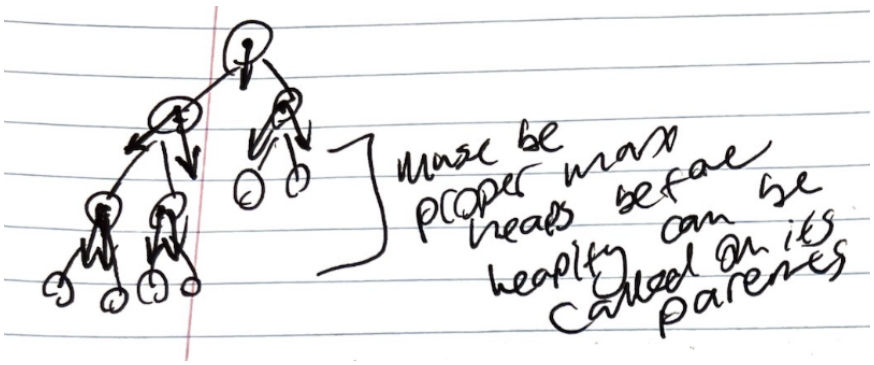

Figure 7: The idea of building a heap

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:

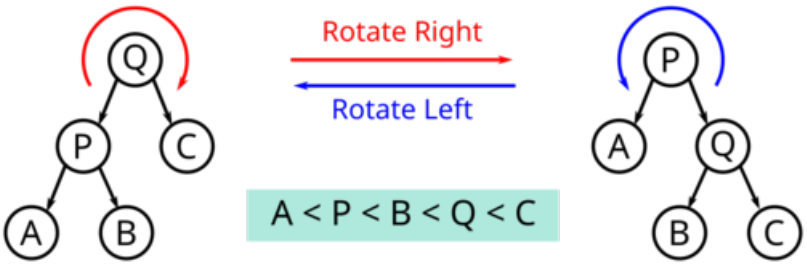

Figure 8: Tree rotations. Image from Wikipedia, see Binary Tree Rotations

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

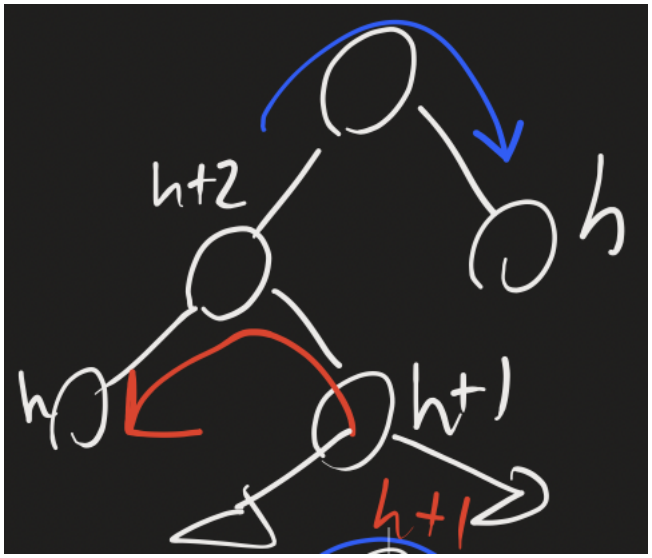

Figure 9: Left then right rotation

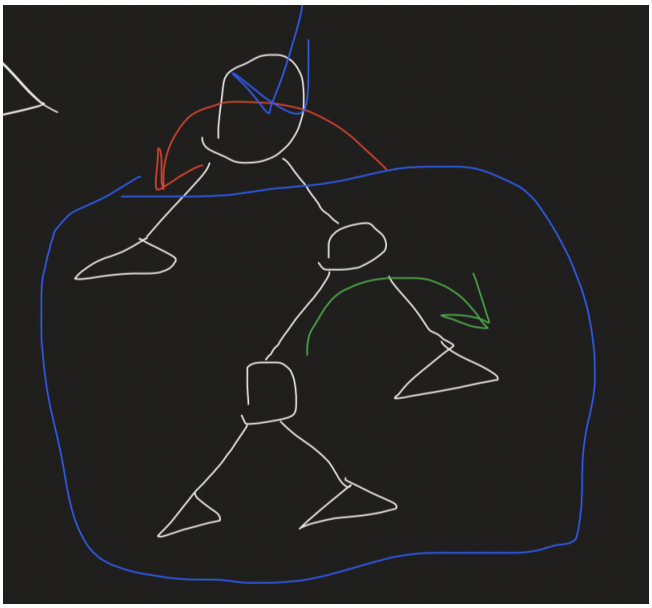

Figure 10: Right then left rotation. The root node is the target of rotation prior to being rotated. Everything in the large blue rectangle must be almost balanced, as a precondition.

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.

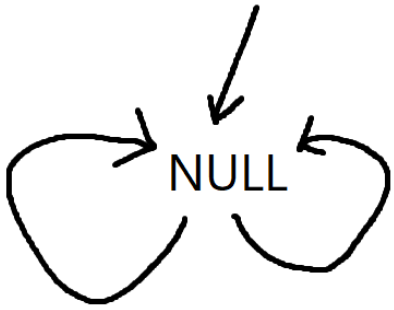

Figure 11: Null points to itself.

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
```

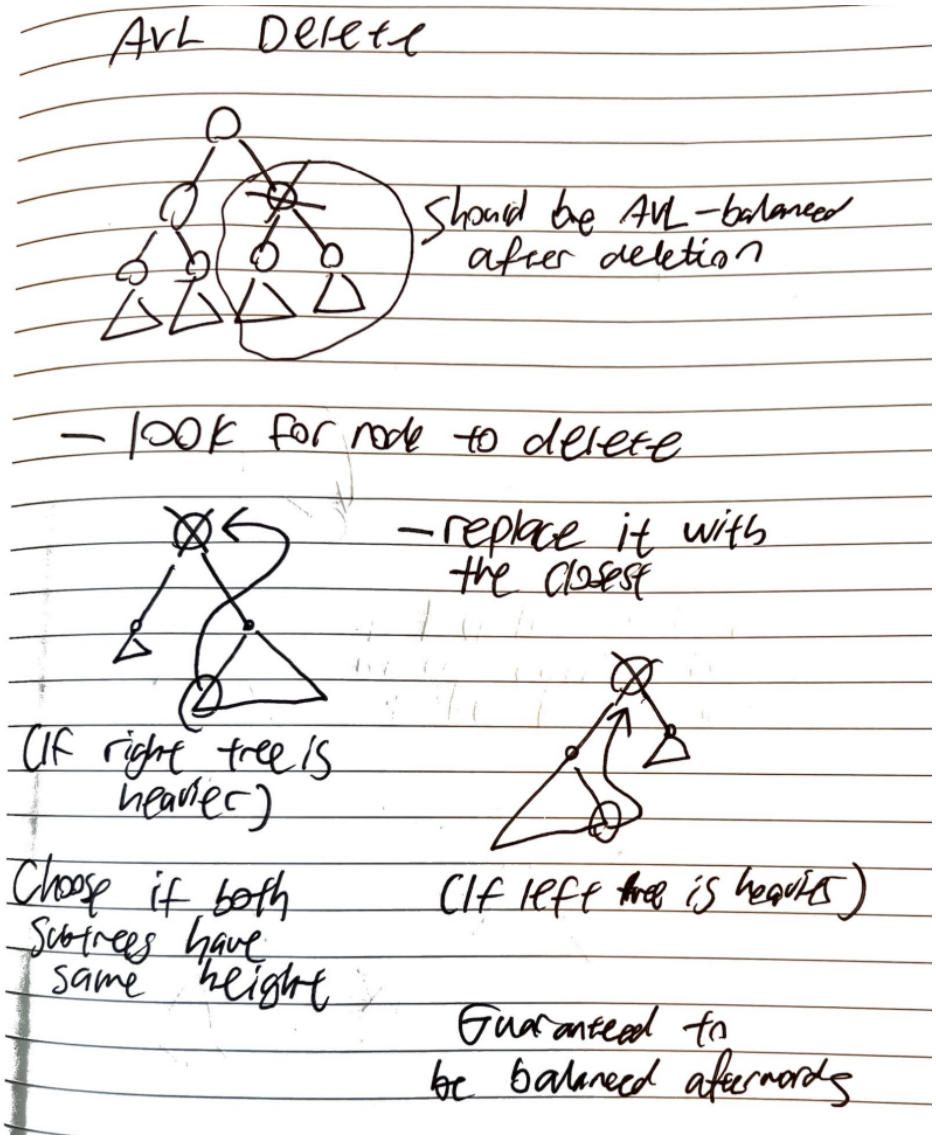

Figure 12: On AVL deletion

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 And Ordered Sets

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)`
  * Really should be `index(...)`
  * 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: $x \in S$ (ONE-INDEXED)
  * 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)`
  * Really should be `__getitem__(...)`
  * 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 $h(k)$.

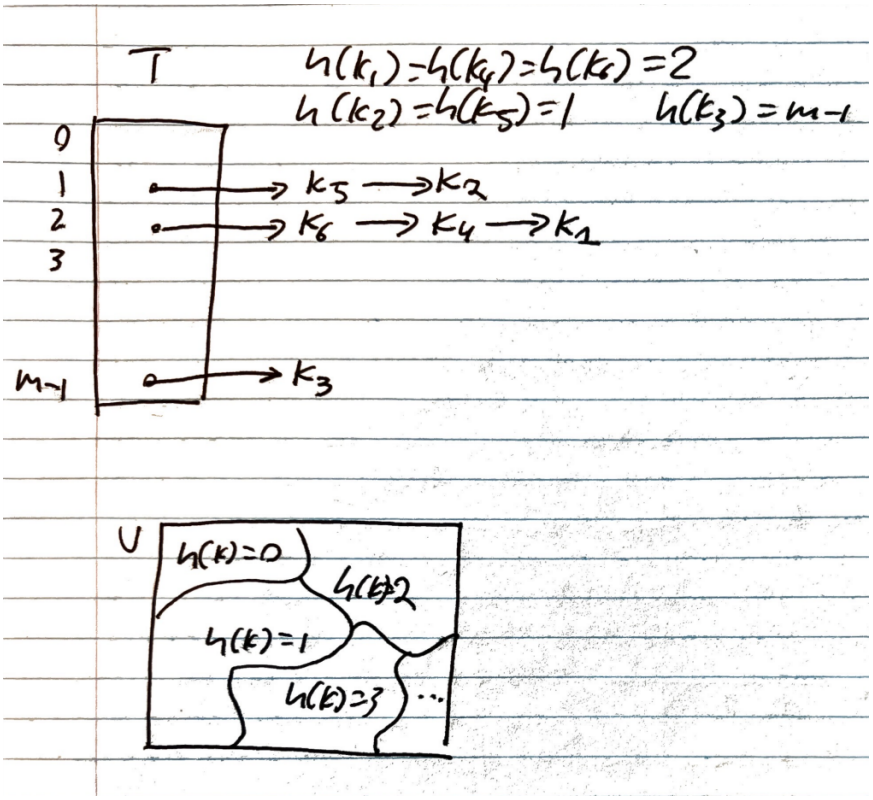

Figure 13: Collisions and Partitioning

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). m is the bucket count.
- 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.

L_i is the no. of keys in bucket i.

$$
\begin{aligned}
&= \sum_{i=0}^{m-1} \left(\sum_{k \in U : h(k) = i} \Pr[k] \cdot N(k) \right) \\
&\leq \sum_{i=0}^{m-1} \left(\sum_{k \in U : h(k) = i} \Pr[k] \cdot L_i \right) \\
&= \sum_{i=0}^{m-1} \left(L_i \cdot \sum_{k \in U : h(k) = i} \Pr[k] \right) \\
&= \sum_{i=0}^{m-1} \left(L_i \cdot \Pr\left[h(k) = i\right] \right) \\
&= \sum_{i=0}^{m-1} L_i \cdot \frac{1}{m} \quad \text{from simple uniform hashing assumption} \\
&= \frac{1}{m} \sum_{i=0}^{m-1} L_i = \frac{n}{m}
\end{aligned}
$$

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:** $h(k,\ i) = (h_1(k) + i \cdot h_2(k)) \mod m$
  - 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 when $\alpha < \frac{1}{2}$ (your hash table isn't more than half full). This means I have to deliberately not use half of the memory I set aside.
- But it's faster than making linked lists.
- Open addressing is more used in practice
- The average case analysis for this is very messy
- Use a sentinel value `DEL` to indicate deleted elements, so when I delete a value I don't need to shift everything. Skip over `DEL` as a key I'm not looking for. This prevents `SEARCH` from breaking, and `INSERT` can replace `DEL`.

5.5 Hash Functions

It's called hashing because it makes two similar values hash to very different quantities. What makes a good hash function (desired properties)?

1. $h(k)$ depends on the entire key k
2. $h(k)$ spreads out keys
3. $h(k)$ is efficient to compute

What if keys are strings? If you want your hash function to depend on every character, it wouldn't be efficient (3). If you took the first few characters, it won't depend on the entire key (1). In practice, you come up with hash functions for sufficient parts of the string.

Good hash functions exist but they are hard to develop. Use them (you don't need to implement it).

Ultimately, all hashing boils down to data (ANY STRUCTURE) $\rightarrow$ integers or floating point (through casting) $\rightarrow \{0, 1, \ldots, m = 1\}$.

The division method: $h(k) = (ak + b) \bmod m$

5.6 AVL vs. Hashing

Hashing seems like it's the best choice, because it's $\mathcal{O}(1)$ on average. You end up with this decision to make when you need a dictionary:

- Guaranteed $\mathcal{O}(\log(n))$
- Average $\mathcal{O}(1)$
  - Probably of worst case $\mathcal{O}(n)$ is less than a hardware failure

There is one thing hashing cannot do. If you have any application that requires to manipulate data in a **sorted** way, hashing is useless. If you want a list that sorts itself when you add stuff in, AVL trees are your choice.

If you don't need to process values in order, everything else, you'll use hashing.

6 Quicksort

Let's introduce Quicksort again:

```
1 def QUICKSORT(A: list[Comparable]) -> None:
2     if len(A) <= 1:
3         return A
4     select pivot element p in A
5     L = [all elements in A <= p except p itself]
6     G = [all elements in A > p]
7     Return QUICKSORT(L) + [p] + QUICKSORT(G)
```

In practice, quicksort is done in-place.

How efficient is Quicksort? $\Theta\left(n^2\right)$. If we kept choosing the largest element as the pivot, that's going to happen.

6.1 Average Case Analysis

Deterministic quicksort: Pivot $p = A[0]$ (first element)

Sample space (no repeated elements – just ignore):

$$S_n = \{\text{all permutations of } [1, 2, \ldots, n]\}$$

Assume a uniform probability distribution. Each permutation is as likely as each other

Pick a key operation that happens during the execution of the algorithm where you'll get an answer that is a constant factor of the overall runtime. What matters the most is the **partitioning**. How many comparisons between array elements does the algorithm perform?

Let $T(A) =$ no. of comparisons performed by $Quicksort(A)$. We want: $E[T(A)]$ (how many comparisons do we make on average?) There's a trick we're going to perform: can I decompose this random variable into simpler random variable?

Define $X_{i,\,j} = \left\{ \begin{array}{c} 1 \text{ if } i \text{ is compared to } j \text{ during } QS(A) \\ 0 \text{ otherwise} \end{array} \right\}$ for all $1 \leq i < j \leq n$ (look at all the possible pairs of values in $[1,\, n]$). Once two elements are compared, they will never be compared again, so they're compared once or not at all. This means that the total number of comparisons done is:

$$\begin{aligned} T(A) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,\,j} \\ E[T(A)] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,\,j}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P[X_{i,j} = 1] \end{aligned}$$

What is $P[X_{i,j} = 1]$? How likely is it that i and j are compared? They will be compared if and only if either i or j gets picked as a pivot first.

$P[X_{i,\,j} = 1] = P[i$ or j appear in A before all other values in range $[i \ldots j]]$
$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$ (double the reciprocal of the distance between the two)

Now we have an expected value expression:

$$E[T(A)] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$
$$\ldots \in \Theta(n \lg(n))$$

6.2 Randomized Quicksort

- Each call, select pivot at random
- Now, the runtime with an input is now variable (random)
- For every input, I get an expected runtime. It's going to be the same for all inputs, so the initial ordering does not matter.

Randomizing the algorithm is a powerful idea, where you can't determine ahead of time a fixed strategy that is guaranteed to work well. Where lots of the choices you could make would work well, then picking at random would work well.

7 Amortized Analysis

$$\text{AMORTIZED} = \frac{\text{WORST CASE SEQ COMPLEXITY}}{\text{NUMBER OF OPERATIONS}}$$

Worst-case sequence complexity: **max** total time to execute any sequence of m operations on a data structure (from a fixed starting point, usually starting empty).

It's not how long does it take to perform 1 operation, but how long does it take to perform many? Economies of scale

The difference from average-case:

- More realistic scenario

Example: max heaps

- I could construct it from a list by inserting each element one-by-one $\in \Theta(m \lg(m))$.

In general: worst case sequence complexity $\leq m \cdot$ max time for one operation

7.1 Binary Counters

There are algorithms where the runtime function looks like a sawtooth. For example, binary counters. A binary counter is k bits: $0_{k-1}000\ldots0_0$

There is one thing we can do with a binary counter: add 1. Incrementing the counter once gives $000\ldots001$. Doing it again gives $000\ldots010$. And so on.

If I do m of these (assume $m < k$), the total running time is the number of bits that need to be changed each increment, which is the cost – the time it takes to do each operation. Regrouping in addition requires a lot of bits to change.

If I take the runtime, I get a sawtooth pattern. Peaks occur when I add for any power of 2-th time. What is the total number of bit changes?

Bit number	Changes this many times
0	m
1	$\lfloor \frac{m}{2} \rfloor$
i	$\lfloor \frac{m}{2^i} \rfloor$

$$T = \sum_{i=0}^{\lg(m)-1} \frac{m}{2^i} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

7.2 Dynamic Arrays

An array. If I try to insert it when it is full, allocate a new array and make it **twice** as large and copy all existing elements into the new array.

Let's look at a sequence of m INSERT operations starting from an empty array with capacity 1.

The **COST**, which is per operation in the list:

$$\text{cost}(\text{INSERT}(k)) = \begin{cases} 2k-1 & \text{if } \exists n \in \mathbb{N} \text{ such that } k = 2^n + 1 \\ 1 & \text{otherwise} \end{cases}$$

$k = 2^n + 1$: read $2^n = k - 1$, write $2^n + 1 = k$, which summed up gives us $2k - 1$

The **CHARGE** – on average, over all operations. When I charge a certain amount, it has to cover:

- The cost of writing the new element (always happens and costs \$1)
- Moving each element costs \$2 in total when a total move occurs, already accounting for the cost of writing
- I need to put in credit so I can use them to copy everything when I need to. This means each array index has a credit
  - If I charge \$5 to add something, I'm good to go

7.2.1 Credit Invariant

If I want to charge \$5 for adding a new element:

A statement of the total amount of credit in a point in time. Example: Every element in the 2nd half of the array's total size (NOT number of elements) has \$4 of credit or doesn't have an element in it.

Proof. Base case: 0 elements means 0 credits, so vacuously true

Consider one insert. Assume credit invariant holds:

- Array doesn't grow, new element gets 4
- Array grows because it was full, so total credits I have is $\$4 \cdot \frac{n}{2} = \$2n$ which is enough to cover copying n elements

■

So, dynamic arrays that double size when full are 5 times as slow as regular arrays.

YOU ARE PERFORMING INDUCTION ON THE mth OPERATION
This means the format goes like follows:
Base case: does the credit invariant hold on a new data structure?
Inductive step: I'm in a state where my credit invariant holds. What happens after I perform an operation, and more importantly, does my credit invariant still hold after I perform this operation?

8 Graphs

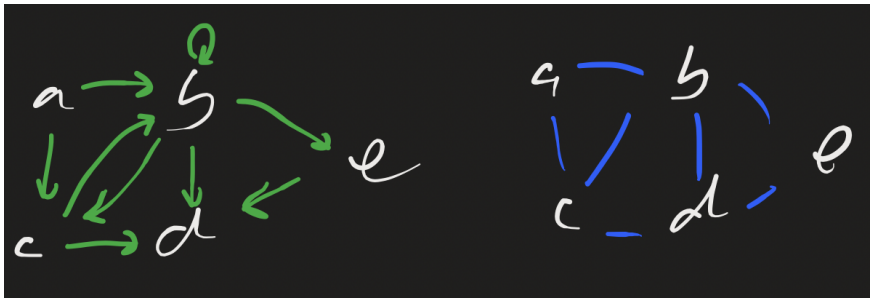

Figure 14: Directed vs. Undirected Graph

In an undirected graph, an edge is there, or is not. In a directed graph, you can have an edge from a vertex to itself, but not an undirected graph. Each arrow in the directed graph is its own edges. The edge from b to c is not the same as the edge from c to b.

A path is just a path and can be infinitely as long. Vertices and edges can be repeated. When we say path, it means this path and not a simple path.

A **simple path** is a path that does not repeat any vertices and edge. Use the word "simple" in places where it matters.

For directed graphs, you can only follow the path in the directions supported in the directed graph, just like an NFA.

A **cycle** is a sequence of edges that takes you from one vertex back to the same vertex. If it's a cycle, it's a path, but it is never a simple path.

A **simple cycle** has the same first and last vertex, but otherwise vertices and edges are traversed only once. Meaning in the left graph, $b \to c \to b$ is a simple cycle but $b - c - b$ is not. In the directed graph, $d \to d$ is a simple cycle. When cycles are talked about, it is safe to assume that they're simple cycles. If you ignore the last vertex of a simple cycle, the rest of it is a path.

A graph is a set of vertices and a set of edges: $G = (V, E)$. How do we represent them?

8.1 Adjacency Matrix

$V = \{v_1, v_2, \ldots, v_n\}$ (assume my vertex set contains n vertices). An edge is a relationship between two vertices. Give me any two vertices, and I have a potential edge. Is it in the graph, or is it not? Let's answer that explicitly for every possible edge. For instance, the undirected graph's matrix is $(a \to \ldots e)$ as follows:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

NOTE that row is the TAIL $\to$ column is the TIP. Undirected graph $\Rightarrow$ symmetric adjacency matrix. The converse is NOT true and keep the contrapositive in mind.

An adjacency matrix is a 2D array.

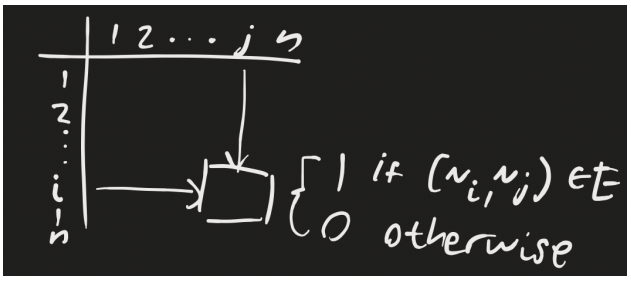

Figure 15: Adjacency matrix

Let $n = |V|$, $m = |E|$

8.1.1 Complexity

A graph really is a relationship between the integers $1 \ldots n$.

The **space** complexity is $\Theta\left(n^2\right)$

Edge queries (is $(v_i, v_j) \in E$?) takes $\Theta(1)$. Look up $A_{i,j}$.

The cost is that you're storing the answer to each question. Works well if you have a graph with a lot of edges, but not if the graph only contains a few. In practice, graphs are really large. Your matrix will be storing a lot of zeroes so that you have an explicit spot for every potential edge, where most of them aren't there.

8.2 Adjacency Lists

Storing what's there rather than everything that possibly might be there. Let $V = \{v_1, v_2, \ldots, v_n\}$ (we index at 1 when working with graphs).

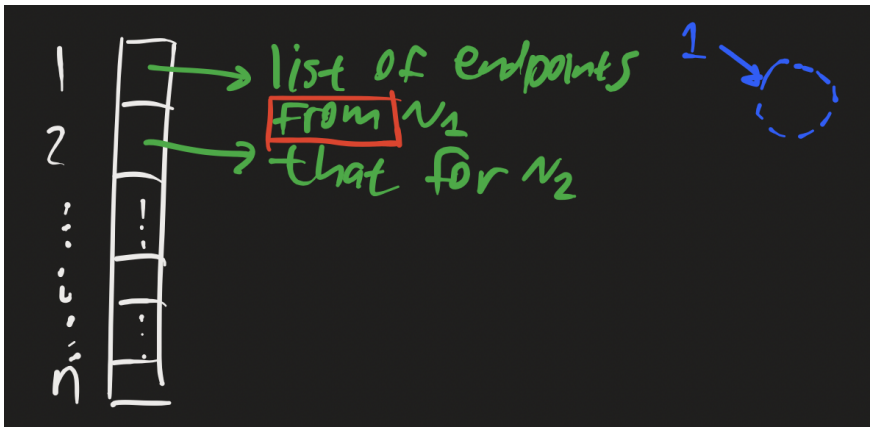

Figure 16: Adjacency list

For instance, if we look at the undirected graph at the start of this section, the adjacency list is:

$$\begin{array}{ll} a & b, c \\ b & a, c, d, e \\ c & a, b, d \\ d & b, c, e \\ e & b, d \end{array}$$

This graph is undirected, so each edge is listed twice, once from each vertex. The edge $a - b$ appears in two pieces: $(a,\ b)$ and $(b,\ a)$. This is different from directed graphs.

8.2.1 Complexity

Space: $\Theta(n + m)$

Edge query: To find if $(v_i,\ v_j) \in E$, you need to look through the list for vertex i. It could have an edge to every other vertex. That becomes $\Theta(n)$ in worst-case.

9 Breadth-First Search (BFS)

How do I loop/traverse over every edge?

Goal:

- Start from source vertex $s \in V$
- Explore every vertex that can be reached from s using only the edges to connect from one vertex to the next

Initially, we'll have s. Either given by the user, or just an arbitrary choice. Then, we can assign vertices the following:

COLOR

- Black (fully explored recursively)
- Grey (discovered but not explored)
- White (undiscovered, and we don't know if we can get to them)

$\pi[v]$: the predecessor/parent of v in BFS tree. As the search is running, I will be exploring edges that come from S. Every time I come to a new vertex; it becomes a new BFS tree edge. Those edges will end up forming a tree. The edges of that tree will only be spawned from the FIRST discovery

$d[v]$: distance (no. of edges) from s to v in BFS tree.

The key characteristic that makes this BFS is:

- Start from S
- What are all the new vertices I can get to starting from s? How do we keep track of vertices that are left to explore?
- Use a queue to keep track of the grey vertices. The ones we know we can get to but that we can't explore yet.

The BFS search algorithm goes like this:

```
def BFS(G: Graph, s: Vertex) -> None:
    # initialize all verticies
    for v in G.V:
        color[V] = WHITE
        predecessor[v] = NIL
        distance[v] = math.inf  # dist from s

    Q = Queue<Vertex>()
    color[s] = GREY
    distance[s] = 0  # distance of 0 as it is itself
    Q.add(s)

    while len(Q) != 0:
        u = Q.deque()
        for v in G.get_adjacent_vertices(u):
            if color[v] != WHITE:  # not discovered
                continue
            color[v] = GREY  # discovered
            predecessor[v] = u
            distance[v] = distance[u] + 1
            Q.add(v)
        color[u] = BLACK
```

In short, this is what the algorithm is trying to do:

- Set every vertex to WHITE, make it have no predecessor, and make all its distances infinity (another word for unknown)
- Initialize target vertex s: set it to GREY, and set its distance to 0. Enqueue it to queue Q.
- Loop back here until Q is empty, grabbing vertex u from the queue each time:
  - I want all of u's adjacent vertices. I will loop over each of them. For v in u's adjacent vertices:
    * If it's white/undiscovered, I will set the color to grey (make it discovered), set its predecessor to u, set its distance to $u + 1$ (as $s \to v$ path uses one more edge than $s \to u$), and enqueue v into Q
    * If it's discovered already, then I don't need to do anything as the Q

probably has accounted for that vertex

- Set u to black. I don't really need to do this – it is more of a flag to say that we have already looped through every adjacent vertex of u.

d is distance but abbreviated.

9.1 Running Time for BFS

- Initializing tracking info is $\Theta(n)$ time
- Initializing s is $\Theta(1)$

Observations:

- Each vertex is enqueued at most once.
- Each vertex is dequeued at most once.
  - When a vertex is dequeued, we look at its adjacency list, so each vertex's adjacency list is looped over at most once. Examined once or not at all.

Worst case:

- If I look at a graph that is connected, I look at each element once. The worst case is constant time for each adjacency list element $\Rightarrow$ time is $\Theta\left(\underset{|V|}{n} + \underset{|E|}{m}\right)$

9.2 BFS Correctness

In BFS, we look at the nodes closest to s first and the furthest from s last. Meaning:

- Iterate through everything distance 1 away, before
- Iterating through everything distance 2 away, and so on

Definition 9.1 (Minimum distance function)**.** For all $u, v \in V$, $\delta(u, v) =$ min. distance (smallest no. of edges in any path) from u to v. ∞ if there are no path between u and v.

δ says, out of EVERY path, what is the smallest number of edges. This takes very long, yet BFS is linear time.

Claim: After BFS, $\forall v \in V,$ distance$[v] = \delta(s, v)$. How do I know that the path BFS finds for each v is the shortest?

There are a few properties (lemmas) I need to rely on, some with δ and others with BFS.

1. For all edges $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$
    a. Rationale being, $s - \cdots - u - v$ screams that the distance from s to u is the distance from s to $v \pm 1$
2. $\forall v \in V$, $\delta(s, v) \leq d[v]$
    a. I do need to traverse $\delta(s, v)$ times to get to $d[v]$ in the first place
3. At any point during BFS, $Q = [v_1, v_2, \ldots, v_k]$ with $d[v_1] \leq d[v_2] \leq \ldots \leq d[v_k]$, and $d[v_k] \leq d[v_1] + 1$
    a. This means all vertices in my queue are either the same or go up by one at most once. Meaning either $[d, d, d, \ldots, d]$ or $[d, d, d, \ldots, d, d+1, d+1, \ldots, d+1]$

Proof. For a contradiction, assume $\exists v \in V$, $d[v] \neq \delta(s, v)$.

Lemma 2 $\Rightarrow d[v] > \delta(s, v)$

Let v_0 satisfy $d[v_0] > \delta(s, v_0)$ with minimum $\delta(s, v_0)$

- v_0 here is defined to be the vertex with THE minimum distance from s where BFS makes an error **!!
- So if BFS makes an error calculating distance for v_0, it will always overestimate the distance

Can $v_0 = s$. Can BFS make an error with a distance to itself? $\delta(s, s) = 0 = d[s]$. So $v_0 \neq s$.

Take the shortest path from s to v_0:

- $s - \cdots - u_0 - v_0$, where $(u_0,\ v_0)$ is the last edge on the shortest $s \to v_0$ path

Then $\delta(s,\ v_0) = \delta(s,\ u_0) + 1$

This means that $\delta(s,\ u_0) < (s,\ v_0)$

Because v_0 is the closest vertex to s where BFS calculates the wrong distance, then BFS must have calculated the correct value for $d[u_0]$

Status of v_0 just before u_0 is dequeued during BFS, each case contradicts $d[v_0] > d[u_0] + 1$ (check the slides, TODO)

■

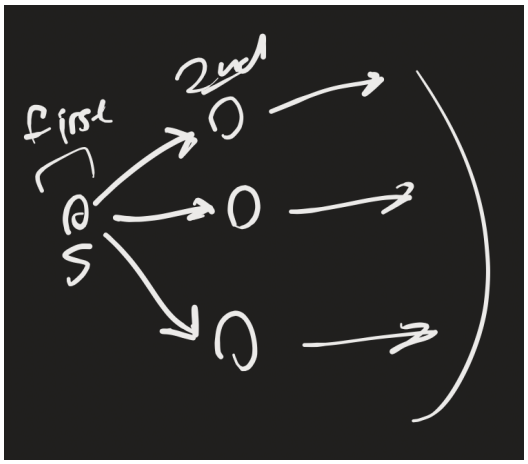

Figure 17: BFS

BFS traverses every vertex reachable from S, but not necessarily all of G. We could add an outer loop to try each vertex as a starting point.

10 Depth-First Search

In DFS, we start from a vertex s and we fully explore before looking at anything else.

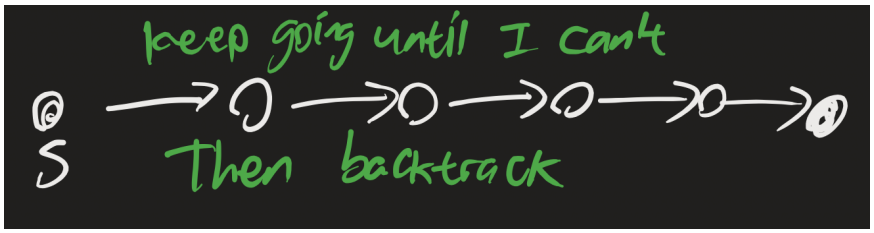

Figure 18: DFS

Some information to track for DFS:

- Timestamps
  - $d[v] =$ discovery time
  - $f[v] =$ finish time

For DFS, it is typical to have a version of it that ends up traversing every node, regardless of whether the nodes are connected or not. This means it will traverse the entire graph.

```
time = 0

def DFS(G: Graph) -> None:
  global time
  time = 0
  # initialize
  for v in G.V:  # for each vertex
    d[v] = f[v] = math.inf
    predecessor[v] = NIL
  for v in G.v:
    if d[v] == math.inf:  # not discovered
      DFS_VISIT(G, v)

def DFS_VISIT(G: Graph, v: Vertex) -> None:
  global time
  d[v] = ++time # increment time, store it in d[v]

  # do something with v; could be a callable
  DO_SOMETHING(V)

```

```
21    # explore v
22    for u in G.adj[v]:
23      if d[u] == math.inf:  # u is unexplored
24        predecessor[u] = v  # we discovered it from v
25        DFS_VISIT(G, u)
26    f[v] = ++time
```

Note that colors are omitted. However, colors can be deduced from the timestamps:

- White: $d = f = \infty$
- Grey: $d < f = \infty$
- Black: $d < f < \infty$

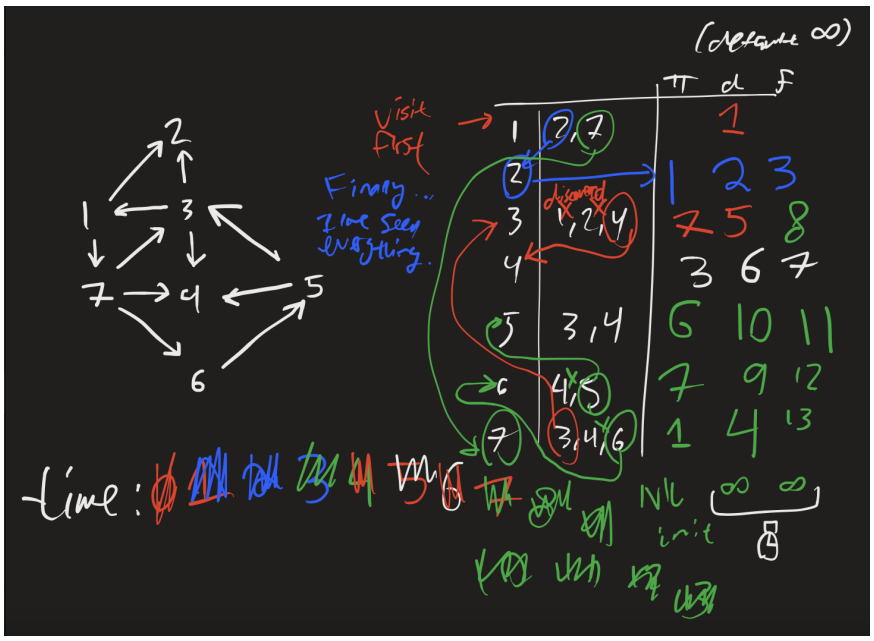

Figure 19: DFS

And if we look at the predecessors, we can form a tree:

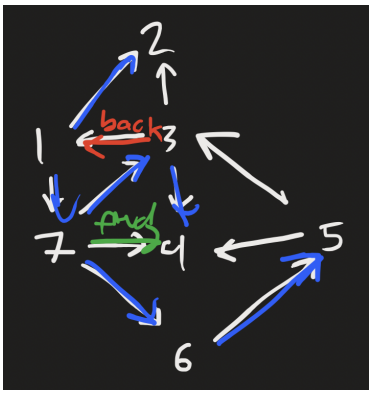

Figure 20: Depth first tree.

The edge from 3 to 1 is a back edge: it is an edge from a vertex 3 to another vertex 1 that is earlier in the depth-first tree.

The edge from 7 to 4 is called a forward edge: an edge with the property that it goes from a vertex to one of its descendants.

The edge from 5 to 3 is a cross edge (and every edge that isn't red, blue, or green). It is across different components of a tree, with the same depth.

This means every edge of the graph is either:

- A tree edge (part of the tree, blue)
- A forward edge
- A back edge (results in a cycle)
- A cross edge

Forward and cross edges are only in directed graphs.

10.1 The Parenthesis Theorem / Property

The bolded numbers here indicate that I'm now finished with a vertex. The below sequence is chronological.

$$(1, (2, \mathbf{2}), (7, (3, (4, \mathbf{4}), \mathbf{3}), (6, (5, \mathbf{5}), \mathbf{6}), \mathbf{7}), \mathbf{1})$$

If I list the discovery and finish events chronologically, I get a perfectly nested expression. I will always close 1 before closing anything within that. This comes from recursion.

For all $u, v \in V$, after DFS runs:

- Either u is an ancestor of v in the depth first tree
  - $\Leftrightarrow d[u] < d[v] < f[v] < f[u]$
- Or v is an ancestor of u
  - $\Leftrightarrow d[v] < d[u] < f[u] < f[v]$
- Or neither are ancestors of the other
  - $\Leftrightarrow d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$

This captures that there is no overlaps in the recursive structure.

10.2 Runtime for DFS

The number of iterations is uneven for the inner loop.

- Outside of recursive calls, $DFS(G)$ takes $\Theta(n)$ time.
- `DFS_VISIT(G, v)` is called **exactly** once for each $v \in V$
  - At least once because of the main loop
  - At most once because $d[\ldots]$
- Over all calls, each adjacency list is examined exactly once $\Theta(m)$ – edge count

This means that the total runtime for DFS is $\Theta(n+m)$

10.3 Topological Sorting

Cycle detection. Cycle means back edge, and we cannot do topological sort if our graph has a cycle.

Topological sorting: Order the vertices in a directed graph so that every edge (u, v) has u appear before v in the ordering

- If vertex v is after u in the list, there is no path from v to u

If a graph has no cycle, G can be topolocially sorted $\Leftrightarrow$ G contains no cycle $\Leftrightarrow$ DFS finds no back edge.

Here's the algorithm for sorting a graph:

1. Run DFS $\Theta(n+m)$
2. List the vertices in decreasing order of their finish time $\Theta(n)$

How do we know that this works? I want (u listed before $v \Leftrightarrow f[u] > f[v]$)

From our traversal of the graph above:

$$(1, (2, \mathbf{2}), (7, (3, (4, \mathbf{4}), \mathbf{3}), (6, (5, \mathbf{5}), \mathbf{6}), \mathbf{7}), \mathbf{1})$$

We take out the bolded numbers, stuff it in a list, and reverse.

$$\ll \text{later finish time}$$

$$[1, 7, 6, 3, 6, 8, \ldots]$$

See below:

- Consider any edge (u, v) at the point when DFS explores
- In `DFS_VISIT(G, u)`, processing `G.adj[u]` (the adjacency list), so I know
  - $d[u] < \infty$ and $f[u] = \infty$
- Consider possible values for $f[v]$.
  - Could v have been **discovered and fully explored?** This would mean $f[v] < \infty \Leftrightarrow$ parenthesis theorem: $f[v] < d[u] < f[u]$

    * This means that I would have already observed v earlier in the topological sorted list than u

- Is v a **vertex I've not seen before?** Then v is a descendent from $u \Leftrightarrow$ the parenthesis theorem, $d[u] < d[v] < f[v] < f[u]$
    * This means that v would be placed later than u in the topological sorted list
- Could v have been discovered but not finished yet? This means that v is currently being explored. This can only happen when u is a descendant of $v \Leftrightarrow (u, v)$ is a back edge. **That cannot happen if my graph has no back edge.** So, this is not possible if G has no cycle.

11 Connectedness

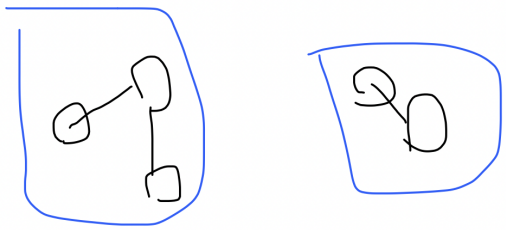

Figure 21: Each blue rectangle represents a connected component

When the graph is directed, the notion of connectedness is more complicated.

- Weak connection: path in ≥ 1 direction from one vertex to another.
- Strong connected: there is a way to use edges to go from one to the other, and back

This is not about single edges; this is about paths.

11.1 Strongly Connected Component

A strongly connected component is a subset of vertices:

- $C \subseteq V$ such that
- $\forall u \neq v \in C$, there is some path $u \to v$ and $v \to u$ within C
- C is maximal. It is not possible to add more vertices to C and keep it strongly connected.

This means if C only has one element then it is maximal, then it is strongly connected.

11.2 Finding Strongly Connected Components

It's easy in terms of what it is doing, but the reason why it works is not obvious.

1. Run DFS on the transpose of G. The transpose means reverse all edge directions.
2. Reorder the vertices in decreasing order of finish times (latest to earliest), in the graph itself $G.V$ and within each adjacency list.
    a. This impacts the order of vertices that are iterated through in the DFS algorithm.
3. Run DFS with the new ordering, with the original, non-reversed edges.
4. Each DFS tree $=$ one strongly connected component

Why does this work?

$$(\text{finally } (\text{then this after } (\text{search this next } (\text{search this first}))))$$

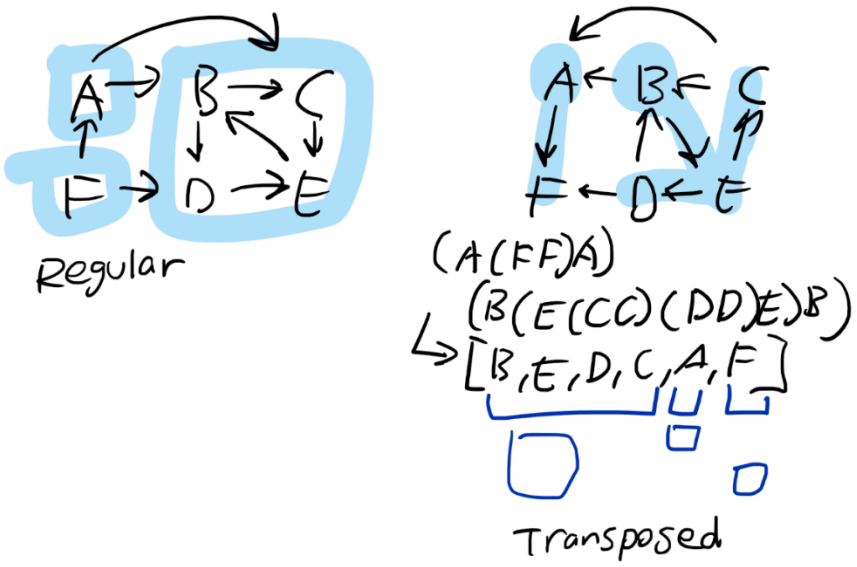

Figure 22: Strongly connected intuition

Intuition:

1. First DFS traversal generates ordering on strongly connected components
2. Second DFS traversal encounters components in reverse topological order on the original graph
    a. So, the first vertex I'll traverse to has no edges leaving out the strongly connected component
    b. The second one has edges only to the first
        i. I will encounter all vertices in that component, and any edges that leave the strongly connected component will only go to the first strongly connected component, which I will avoid
    c. The third… and so on

And this is why every individual tree I get will form a strongly connected component if one is ever made.

12 Minimum Spanning Trees

Input: **undirected**, **connected** graph. The graph has **weights** on the edges:
$\forall e \in E,\ w(e) \in \mathbb{R}$

Output: A spanning tree with the **minimum** total weight.

What's a spanning tree? A subset of edges that is acyclic, connected, and covers all the vertices.

Spanning trees all have the following properties:

- Acyclic
- Connected
- Has exactly $n-1$ edges (n is no. of vertices)
- Add any edge to a spanning tree result in one cycle being created
- Removing an edge turns the spanning tree into two subtrees

12.1 Prim's Algorithm

Intuition: start with a graph. We want a spanning tree, so it has to touch every vertex.

- Start from an arbitrary root R (just pick 1).
- Grow the tree one edge at a time

Prim's algorithm is greedy, so it makes no consideration about things it does not know.

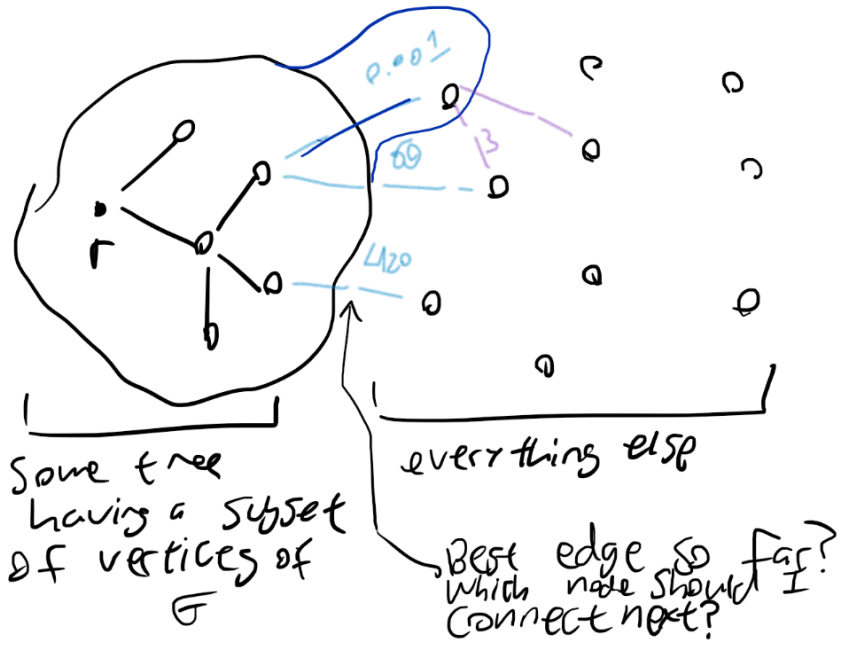

Figure 23: State of Prim's Algorithm

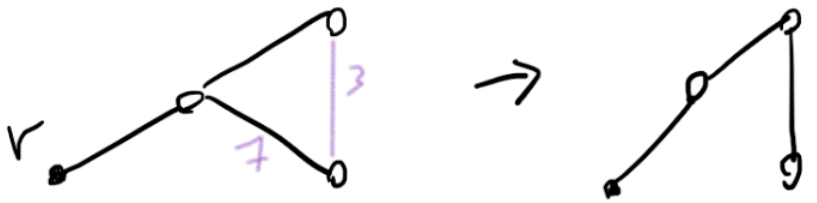

Figure 24: Replacing existing edges in Prim's algorithm. To be done every time an edge is added

Each vertex only needs to maintain once piece of information: what is the best edge to get to it? This may change over the course of the algorithm.

During the execution of the algorithm, of which vertices that I'm not connected with, currently has the smallest edge to connect to the rest of the tree?

That would use min-heaps, which would give us the best performance.

The algorithm goes like follows:

```
def Prim(G: Graph, w: Callable[[Edge], float], r: Vertex)
    -> set[Edge]:
  T = set()  # empty set, current tree edges
  # initialization
  Q = PriorityQueue()  # empty priority queue
  # The prioirity queue stores all vertices I've not
     connected yet
  # with a priority that corresponds to the weight of the
     minimum weight edge
  # that connects this vertex to the tree
  for v in G.V:
    prio[v] = math.inf  # set priority to infinity
    predecessor[v] = NIL
    Q.enqueue(Q, v)  # put all of them in the queue with no
        order

  # initialize the root
  prio[r] = 0
  Decrease_Key(Q, r)  # Make its priority smaller than
     everything else, bubbling it up to the root. O(log(n))

  # MAIN LOOP
  while not Q.IsEmpty():
    # Connect a vertex with min. priority (edge weight)
    u = Q.ExtractMin()
    if predecessor[u] != NIL:
      T = T.add((predecessor[u], u))  # add the edge to the
          tree
    # update priorities of neighbors of u. once the vertex
       is in the tree, do not undo.
    for v in G.adj[u]:
      if v in Q and w(u, v) < prio[v]:
        predecessor[v] = u
        prio[v] = w(u, v)
        Q.Decrease_Key(v)
   return T
```

12.2 Tracing Prim's Algorithm

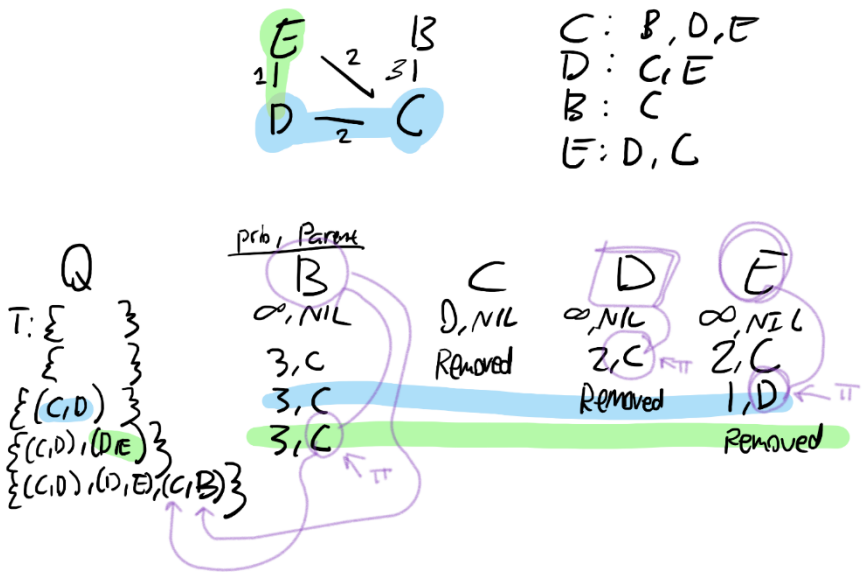

Figure 25: Tracing Prim's Algorithm

12.3 Runtime

- $\Theta(n)$ for initialization
- n iterations for the main loop, for a total of $\Theta(n \lg(n))$
  - n extract mins, where each extra min is log
- Inner loop convers all adjacency lists: $\Theta(m)$ times for a total of $\Theta(m \lg(n))$
  - The `Decrease_key` takes $\Theta(\lg(n))$ time

The total runtime is $\Theta((n+m)\lg(n))$

The reason why this takes really long is the priority queue operations. There is a data structure called Fibonacci heap. Using these heaps, you will be able to reduce the runtime to $\Theta(m + n\lg(n))$. If your graph contains a lot of edges, the runtime improves a bit.

How efficient is `Decrease_key`? I am given the name of the vertex, but I do not know where it is in the heap, taking $\Theta(n)$ time. To fix this: keep track of more information.

- We will keep track of the position of the vertex in the heap
- When I update, I need to adjust my heap algorithm to update the values in the vertices that track its position in the heap
- index$[v]$ = position of v in the min-heap, and this needs to be updated

12.4 Kruskal's Algorithm

Prim's algorithm grows trees.

Kruskal's algorithm is based on a different approach. The idea is to use edges with small weights first. The algorithm is conceptually very easy to write.

```
1 def Kruskal_but_slow(G, w: Callable[[Edge], float]) -> None
     :
2   # sort edges by weight: w(e1) <= w(e2) <= ...
3   T = set()
4   for e in sorted(G.E, key=w):
5     u, v = e  # (u, v) represnts edge e
6     if u, v are not connected using edges already in T:
7       T.add(e)
8   return T
```

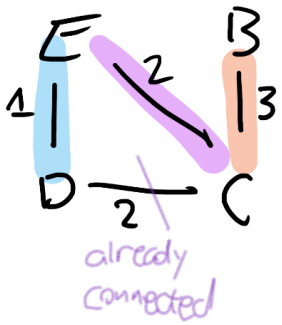

Figure 26: Kruskal's algorithm trace

How do I tell if vertices are already connected by edges of T? (This is different from detecting connected in G)

Unlike Prim's algorithm, I'm not growing a spanning tree. In a large graph, I am really "randomly" putting in edges from an already existing graph (after erasing all the edges). Piece by piece, as I add edges, I am making more components.

Figure 27: Connecting the dots

So, there is no tree until I am done. I need to form the tree afterwards. Not a great idea to run BFS or DFS on each and every node, as it would take $\Theta(mn)$ time.

Can I do better?

```
def Kruskal_but_fast(G, w: Callable[[Edge], float]) -> None
    :
  T = set()
  for v in V:
    MAKE_SET(v)
  # sort edges by weight: w(e1) <= w(e2) <= ...
  for e in sorted(G.E, key=w):
    u, v = e  # (u, v) represnts edge e
    if FIND_SET(u) != FIND_SET(v):
      UNION(u, v)  # replace
      T.add(e)
    if len(T) >= n - 1:
      return T # stop it already
  return T
```

What's the runtime?

- $m \log(m)$ for sorting
- n for initialization (calling make set n times, based on the assumption that making sets take constant time)
- The entire loop is a sequence of m disjoint set operations. So, the worst-case sequence complexity for m disjoint set operations.

13 Disjoint Sets

Objects: A collection of non-empty disjoint sets $S_1, S_2, \ldots, S_k$ where each set come from a fixed universe U.

Not every element in U needs to be in one of $S_1, S_2, \ldots, S_k$, but the same item can't appear twice.

The data structure is not responsible for keeping track of the universe, but it is responsible for keeping track of the sets. We have the following operations:

- `MAKE_SET(x : T) -> None :`
  - Precondition: $x \in U, x \notin S_1, x \notin S_2, \ldots, x \notin S_k$
  - Add new set $\{x\}$ to the collection
  - x is the representative for $\{x\}$
- `FIND_SET(x : T) -> T :`
  - Given an element, tell me what set this element belongs to. Return set S_i such that $x \in S_i$ (or `NIL` if it doesn't exist). The set is guaranteed to be unique.
  - How do we return a set? Identify each set by a representative element. For each set, we will pick one element of the set and we'll say that it's representative of the set. This is an implementation detail, so there is no one way of choosing it. So this operation really returns the **representative for the set that contains** x.

- `UNION(x : T , y : T ) -> None:`
  - Let $x \in S_x$, $y \in S_y$
  - If $S_x == S_y$, do nothing
  - Otherwise, remove S_x and S_y from the collection, and add $S_x \cup S_y$ to the collection. Pick a new representative for it.

13.1 Best Implementation?

WCSC is for performing m total operations on n distinct elements. It is not the same as the m and n as above.

We have a set that just contains the union of all elements across the entire set. Then, about each disjoint set, we store the items in a circular linked list. There are two ways we can do this:

1. Each node has a Boolean telling us if it's the representative or not
   a. All operations but `FIND_SET` are constant time, but `FIND_SET` is linear with respect to the number of items in the targeted disjoint set
2. Or each node has a pointer to the representative
   a. All operations but `UNION` are constant time, with `UNION` taking time proportional to the second set being union-ed.

So the worst case sequence complexity is $\Theta\left(m^2\right)$.

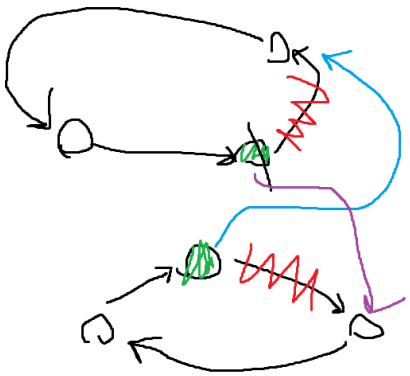

Figure 28: Union

13.2 Attempt 3

Or better?

3. Same as [2], but when I perform a `UNION`, I will append the smaller list to the larger list.
    a. I can track disjoint set sizes by augmenting the structure, and it won't change the asymptotic runtime
    b. Each node stores a pointer to the size, which is no. of elements in a set, for the representative element. The actual size is stored in the representative, and only set representatives are obliged to keep the size correct.
    c. Runtime? Worst case occurs when I am taking the union of two lists with the same size, so I can append one to the other.

For the worst case sequence complexity

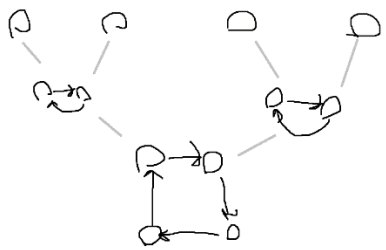

Figure 29: A sequence of MAKE_SETS

Upper bound: for any sequence of m operations, where n `MAKE_SET`s are performed, then for any element x, `.rep` is updated only when size of set that contain x at least doubles. If the list at least doubles and there are n elements in total, for each individual node, the representative pointer can be updated at most $\lg(n)$ times ($\lg(n)$ doublings means that the set that contains x has size $2^{\lg(n)} = n$ afterwards, in that sequence). So the total time is

$$\mathcal{O}(m + n\lg(n)) \in \mathcal{O}(m\lg(m))$$

13.3 Attempt 4 (Inverted Trees)

How about instead of keeping the representative fully updated, I allow the possibility of making multiple jumps. After I've done a few unions, I will have elements that have a representative, but a chain of representative. This means that I would have an inverted tree structure, where each node stores `.parent`. Note that `root.parent = root`. The `root` is the representative.

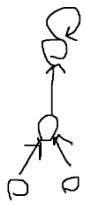

Figure 30: Inverted tree instance

For the operations:

- `MAKE_SET(x):`
  - Initialize one node where it has an edge pointing to itself. Return that node.
- `FIND_SET(x):`
  - Go to node `loc[x]` and follow parent pointers until I reach the node.
- `UNION(x, y):`
  - Trace parent pointers for the x set.
  - Trace parent pointers for the y set.
  - Update y's representative to x by changing y's `.parent` to x.

Runtime? WCSC is $\Theta\left(m^2\right)$, due to creating one long chain and performing `FIND_`SET repeatedly on the bottom node. There are ways to make this better.

13.4 Attempt 5

Attempt 4 but with "union-by-weight":

- Add `size` property to each node. `size` for root is the size of the tree, but only the representative is obliged to keep it correct.
- During `UNION`, make the smaller tree a child of the larger tree
- Like before, WCSC $\in \mathcal{O}\left(m \lg(m)\right)$
- If we kept track of the height instead, we would still get the same upper bound.

13.5 Attempt 6

From attempt 4. Inverted trees. No size, but with **path compression.** The idea behind path compression is:

- We know that we're constructing disjoint sets from the ground up, so it must always follow a property we declare
- Each node, I change their representative to point at the root. We're not going to waste time doing a `UNION` updating all the representatives, but when I do a `FIND_SET`, I have to go through all the nodes and links anyway. The first time I perform `FIND_SET`, it just points to the parent, and that will **never** happen again, in the lifetime (this means I need to track nodes I've run `FIND_SET` on… no, I don't need to).
- You can update representative ("parent" of the node) when backtracking from recursion.

WCSC? Omitted.

13.6 Attempt 7

Using attempt 6, with union by rank. The intuition:

- Keep a new `.rank` attribute in each node. `rank` is meant to represent height. `root.rank` = height of tree if no path compression has taken place. If we have path compression, the height doesn't remain constant. We don't want to recalculate height all of the time, so keep track of the height without any path compression.
- The actual height $\leq$ `root.rank`. We only update `rank` when we do union.
- `MAKE_SET(x):`
  - I make a node, pointing to itself, and its rank is 0. I return that.
- `FIND_SET(x):`
  - I do path compression
  - Ranks are unchanged
  - And all other things I would do when running this

- `UNION(x, y)`:
  - If I end up with two trees r_1, r_2 (ranks are an upper bound of the height). Whichever has the smaller rank, becomes the child of the other.
  - Ranks are unchanged, unless $r_1 = r_2$. Then, increase the new parent rank by 1.

WCSC for m operations: $\mathcal{O}(m \log^*(m))$. This is the iterated logarithm function. $\log^*(n) =$ how many times we take the log, starting from n, to get ≤ 1.

14 Lower Bounds on Problem Complexity

Until now, complexity meant one algorithm or data structure. Problem complexity is different: we're not starting with an algorithm or a data structure that is already implemented. With problem complexity, we are:

- Given a problem to solve
- We want to find the best algorithm for that problem

What does best mean? The best algorithm in the worst-case sense for this problem.

Definition 14.1 (Problem complexity)**.** For a problem, find the **minimum** worst-case complexity of **all** algorithms that solve the problem.

Conceptually, we have our problem P, and we have an algorithm A_1, A_2, ... and a whole bunch of algorithms A_k that solve the problem. Each algorithm has a certain worst-case running time WC $\Theta(t_1)$, $\Theta(t_2)$, ..., $\Theta(t_k)$. We are looking for the smallest worst-case complexity.

14.1 Proving an Upper Bound

We have a bunch of potential algorithms. How do we prove that a problem complexity is no larger than a certain running time?

- **Upper bound** $U(n)$ on problem complexity, find **one** algorithm with worst case runtime $\Theta(U(n))$.

For example, sorting has problem complexity $\mathcal{O}(n\lg(n))$ (an upper bound on the problem complexity), because heapsort takes time $\Theta(n\lg(n))$ (tight bound on the worst-case runtime of heapsort, particularly).

To conclude, sorting can be done in time NO WORSE than $\mathcal{O}(n\lg(n))$. The best algorithm won't be slower than this.

14.2 Proving a Lower Bound

Lower bound $L(n)$ on problem complexity

- Requires a general argument that **every** algorithm takes worst-case time $\Omega(L(n))$.

The difficult part: **every.** We need to come up with a bound for every algorithm.

In practice, arguments are given based on models of computation (for instance, finite state automata). We're going to focus on one particular model of computation: **comparison trees**

14.3 Comparison Trees

Comparison trees:

- Represent any **comparison-based** algorithm
  - One whose main operation is comparisons
  - By comparisons, $<, \leq, =, \neq, \ldots$, 3-way comparison (such as $x : y \Rightarrow, <, =, >$, that one function in Java)

In other words, the worst-case running time for comparison-based algorithms are always in $\Theta(\text{no. of comparisons})$.

This is a **restricted** set of algorithms we can look at. For example, we can't analyze numerical algorithms (like multiplication) using this.

14.4 Comparison Trees Example

For each input size n, we take our algorithm, and trace my algorithm on an arbitrary input size n, and I write down all comparisons performed by the algorithm.

EXAMPLE: Binary search at fixed input size n

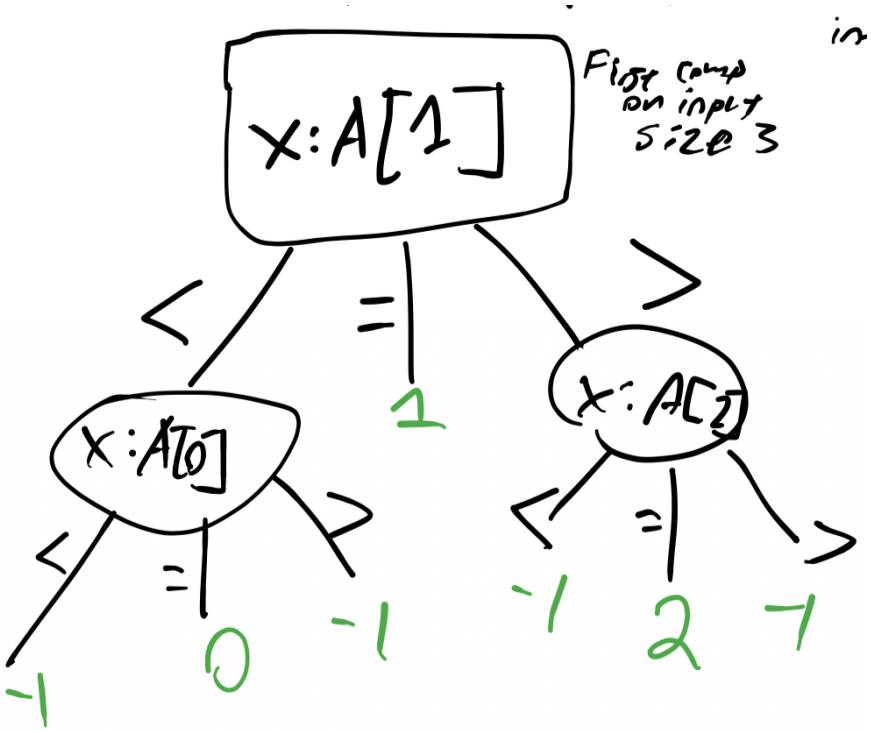

Figure 31: This tree represents the behaviour of the entire algorithm in this compact form, but only relevant for input size 3

The edges label the results of comparison. Internal node: label is **one** comparison. The leaves of this tree are the outputs.

DISTINCTION: Output and outcome:

- Output is the value returned by the algorithm

- Outcome is the information the algorithm knows at the end

Typically, the outcome contains more information than the output. For example, if I can't find x in a binary search, I would **surely** know what it is in between.

How does this help?

14.5 Information Theory Lower Bound

To prove lower bounds for all comparison-based algorithms:

Algorithm $\Rightarrow$ family of comparison trees (one tree for each input size).

In the comparison tree, each execution of algorithm in one input $=$ path from the root down to the leaf.

The worst-case running time of the algorithm is always $=$ to the height of its comparison tree.

For any problem where there are at least k different outcomes, every decision tree contains at least k leaves. It is possible for certain outcomes to appear in multiple leaves. Concisely:

$$\text{no. outcomes} \leq \text{no. leaves}$$

I get my lower bound. In any tree, with k leaves and fixed branching factor has height $\Omega(\log(k))$.

14.5.1 Example: Binary Search

Searching a sorted list:

- Input: sorted list A, value x
- Outcome: i such that $A[i] = x$ or -1 if $x \notin A$

If I have a comparison-based algorithm for this problem, it corresponds to a comparison tree. That comparison tree has at least one leaf for each possible outcome.

If there is one outcome per index plus the negative one, there are $\geq n+1$ outcomes.

The worst-case runtime is
$\in \Omega(\text{height of tree}) = \Omega(\log(\text{no. leaves})) = \Omega(\log(n+1))$.

This means every algorithm humanly possible must make at least $\Omega(\log(n+1))$ comparisons in the worst-case. If not, then not all outputs are being compared.
THERE IS NO ALGORITHM WITH A BETTER WORST-CASE RUNTIME. THERE IS NO BETTER WAY TO DO BINARY SEARCH.

So really, it's $\Omega(\log(\text{no. of outcomes}))$

14.5.2 Example: Sorting

- Input: list A
- Outcome: A permutation of $[0, 1, \ldots, n-1]$
  - [Position of $A[0]$ in sorted list, position of $A[1]$ in sorted list, ...]
  - The information I gain is the ordering I should get
- Output: Your sorted list, so only one output

The information theory lower bound: what are the number of leaves? If I have a correct tree, every possible output has to appear somewhere, so:

$$\text{no. leaves} \geq \text{no. permutations} = n!$$

So the height of comparison trees $\geq \log(n!)$

$$= \Omega(n\log(n))$$

This means that every comparison-based sorting algorithm has worst-case runtime $\Omega(n\lg(n))$.

14.5.3 Searching in an Unsorted List

Comparison trees might not work well for this.

INPUT: Unsorted array A and value x. I have no idea whether or not A is sorted.

OUTPUT: i such that $A[i] = x$ or -1 if $x \notin A$

INFORMATION THEORY LOWER BOUND

The algorithm becomes a comparison tree. How many leaves must it have?

- Must have ≥ 1 leaf per output
  - Equivalent to ≥ 1 leaf per outcome

There are $n+1$ different outputs, so the comparison tree has $\geq n+1$ leaves, 1 for each output possible. Information theory lower bound says that worst case time is

$$WC \geq \log(n+1)$$

Is this a valid lower bound? Is it true that every algorithm that searches an unsorted list takes more than $\log(n)$ steps worst case? Of course. This is a fine lower bound, but it is not a good, tight lower bound. Well, we know it can't be below that otherwise we wouldn't be able to consider all outcomes.

ISSUE: Information theory lower bound isn't good for everything.

Better lower bound? We need another technique.

14.6 Adversary / Adversarial Arguments

Goal: Every correct algorithm requires worst case time $\geq L(n)$.

Contrapositive: Every algorithm that takes worst case time $< L(n)$ is incorrect.

In practice, runtime $=$ counting key operation to use concrete and **exact** lower bounds.

14.6.1 Unsorted List

EXAMPLE: Searching an unsorted list.

- Count no. of comparisons
  - Assume all comparisons are of form $x \stackrel{?}{=} A[j]$
- $L(n) =$ what?
  - Try $L(n) = n$, because we know how to solve within that bound. Write an algorithm that solves this requiring n comparisons.
  - Argue that it is not possible to solve this problem requiring fewer than n comparisons.
  - Make an argument that **ALL** algorithms that make $< n$ comparisons are incorrect.

CONSIDER one such algorithm. Make our input list as we go (similar to how ABSURDLE works). Work through its comparison tree, with supposed inputs $A,\ x$

- For each comparison $x \stackrel{?}{=} A[j]$, set $A[j] \neq x$, go down FALSE branch
- When we reach a leaf, there are at most $n-1$ elements set in A
- There is some index in A that has not been set yet. In other words, at least one $A[i]$ is unset.
  - If algorithm returns $i \Rightarrow$ let $A[i] = 0$, where $x \neq 0$
  - If algorithm returns $-1 \Rightarrow$ let $A[i] = 1 = x$

This means that every algorithm that makes fewer than n comparisons must not have looked at one element in my list. So there is some input that the returned input is the wrong one.

So yes, we do need n comparisons to solve an unsorted list.

14.6.2 Max of a list, unsorted list

Input: List/array A, unsorted; all elements are comparable

Output: Index i such that $A[i] \geq A[j] \; \forall j \in \{0, 1, \ldots, n-1\}$

Info theory lower bound: $\Omega(\lg(n))$

Adversary argument: I need $n-1$ comparisons. Assume I have an algorithm that has $< n-1$ comparisons. I know I'll make an error somewhere. Again, construct my input list as if how ABSURDLE makes it.

- Start at the root
- Traverse comparison tree. Every comparison $A[i] \overset{?}{<} A[j]$. Every comparison eliminates one possible location for the max. I make fewer than $n-1$ comparisons, so there are at most $n-2$ comparisons. This means that $\leq n-2$ positions are eliminated that can't be the max.
- $\Rightarrow$ there are ≥ 2 possible positions for the maximum
- No matter which index is output, some input makes it incorrect.