

CSC263 Notes

Average-Case Running-Time

Average-Case Analysis: Finding a weighted average of running-time on different inputs.

- 1) Define sample space/input set $S_n = \{\text{all inputs of size } n\}$
 - Limit sample space to those representative of all possible algorithm outcomes (ie. no two inputs in S_n should result in the exact same behaviour)
- 2) Define discrete random variable X , function's input, with probability distribution $p_X(x)$ on any $x \in S_n$
 - More "common" algorithm behaviours have higher probabilities of occurring
 - Usually stick to uniform probabilities
- 3) Define $t(x)$ representing running-time for some $x \in S_n$
- 4) Compute average-case running-time $t_n = \mathbb{E}[t(X)] = \sum_{x \in X} t(x)p_X(x)$

eg. Find average-case running-time of LinSearch.

Sample Space: There're infinitely many options:

$$S_n = \{([l_1, \dots, l_n], x) : l_1, \dots, l_n, x \in \mathbb{R}\}$$

But many have duplicate behaviours; we only need input types representative of all possible behaviours:

$$S_n = \{([1, \dots, n], 0), ([1, \dots, n], 1), \dots, ([1, \dots, n], n)\}$$

This is not the only possible choice, but it's simple and representative of all possible locations of x (and thus all possible behaviours of LinSearch).

Probability Distribution: With no extra info, assume uniform: $p_X([1, \dots, n], i) = \frac{1}{n+1}$ for $i \in \{0, \dots, n\}$

$$\begin{aligned} \therefore t_n &= \mathbb{E}[t(X)] \\ &= \sum_{x \in X} t(x)p_X(x) \\ &= \sum_{i=0}^n t([1, \dots, n], i)p_X([1, \dots, n], i) \\ &= \underbrace{t([1, \dots, n], 0)}_{\text{special case } =0} \frac{1}{n+1} + \sum_{i=1}^n t([1, \dots, n], i) \frac{1}{n+1} \\ &= \frac{n}{n+1} + \sum_{i=1}^n \frac{i}{n+1} \quad (\text{ignore constant-time part of running-time}) \\ &= \frac{n}{n+1} + \frac{n(n+1)}{2(n+1)} \\ &= \frac{n}{2} + \frac{n}{n+1} \in \Theta(n) \end{aligned}$$

```
def LinSearch(L, x):  
    # Pre: L is a linked list, x is a number  
    # Post: Return node containing x (or None)  
    curr = L.first  
    while curr is not None and curr.item != x:  
        curr = curr.next  
    return curr
```

Priority Queues and Heaps

Abstract Data Type (ADT): A theoretical model of a data type, how it stores data, its operations

Data Structure: An implementation of an ADT in a program

Priority Queue ADT: A collection where items with highest “priorities” are removed first

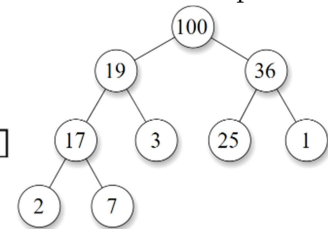
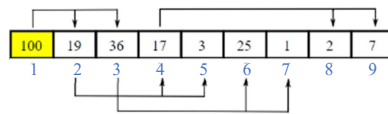
Operation	Description	Speed of Implementations		
		Unsorted List	Sorted List	Heap
insert(Q, x)	Add x to Q with priority x.p	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
max(Q)	Find and return max of Q	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
extract_max(Q)	Pop the max of Q	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Heap Property: Of a tree, if a parent node value is \geq/\leq (for a max-heap/min-heap) its descendant(s).

Almost-Complete: A tree, if all levels are full, except maybe the last, & bottom-level nodes as far left as possible.

Heap: An almost-complete binary tree satisfying the heap property.

- Store binary tree/heap as a list ordered by level, left-to-right
- Indices start at 1 for convenience
- For the tree vertex at list index i ,
 - Parent: index $\lfloor \frac{i}{2} \rfloor$
 - Left child: index $2i$
 - Right child: index $2i + 1$



Heap Implementation	Comments
<pre>def max(Q): return Q[1]</pre>	Constant-time, $\mathcal{O}(1)$
<pre>def insert(Q, x): Q.append(x) while x.p > x.parent.p: swap x, x.parent</pre>	Add item at end of tree (preserve tree structure) Keep swapping item with parents (satisfy heap property)
<pre>def extract_max(Q): swap Q[1], Q[-1] x, m = Q[1], Q.pop(-1) while x.left.p > x.p or x.right.p > x.p: swap x, max(x.left, x.right) return m</pre>	Swap index 1 (max) and -1 (max) (preserve tree structure) Call new index 1 x. Keep swapping x with higher-value child (satisfy heap property) Pop out max (last item)

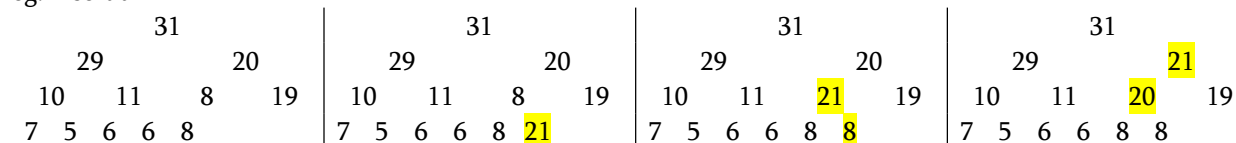
For insert(Q, x):

- Constant-time work at each level
- In worst-case, x moves from bottom \rightarrow top.
- *As Tree:* There're $\log_2 n$ levels, so $\mathcal{O}(\log_2 n)$
- *As List:* Swap index i with $\frac{i}{2}$ until index 1, can be done $\log_2 n$ times, so $\mathcal{O}(\log_2 n)$

For extract_max(Q):

- Constant-time work at each level
- In worst-case, x moves from top \rightarrow bottom
- *As Tree:* There're $\log_2 n$ levels, so $\mathcal{O}(\log_2 n)$
- *As List:* Swap index i with $2i$ until index -1 , can be done $\log_2 n$ times, so $\mathcal{O}(\log_2 n)$

eg. insert a 21

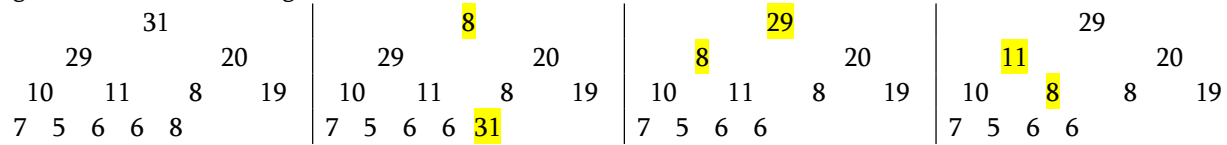


[31, 29, 20, 10, 11, 8, 19, 7, 5, 9, 9, 8, 21]

[31, 29, 20, 10, 11, 21, 19, 7, 5, 9, 9, 8, 8] – swap $Q[13]$ with $Q[\lfloor \frac{13}{2} \rfloor] = Q[6]$

[31, 29, 21, 10, 11, 20, 19, 7, 5, 9, 9, 8, 8] – swap $Q[6]$ with $Q[\lfloor \frac{6}{2} \rfloor] = Q[3]$

eg. extract_max, returning 31



[8, 29, 20, 10, 11, 8, 19, 7, 5, 9, 9, 8, 31] – swap $Q[1]$ with $Q[-1]$
 [29, 8, 20, 10, 11, 21, 19, 7, 5, 9, 9, 8] – swap $Q[1]$ with $Q[2(1)] = Q[2]$
 [29, 11, 20, 10, 8, 20, 19, 7, 5, 9, 9, 8, 8] – swap $Q[2]$ with $Q[2(2) + 1] = Q[5]$

HeapSort: Sorting algorithm in $\mathcal{O}(n \log n)$ that converts lists to heaps and repeatedly calls extract_max().

```
def to_heap(L):
    for i in range(len(L) // 2, 0, -1):
        x = L[i]
        while x.left.p > x.p or x.right.p > x.p:
            swap x, max(x.left, x.right)

def heapsort(L):
    Q = to_heap(L)
    l = []
    while Q.size > 0:
        l.append(extract_max(Q))
    return l
```

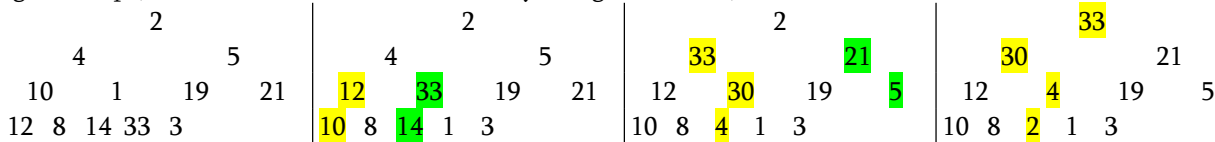
Start at the last non-leaf node (at index $\lfloor \frac{n}{2} \rfloor$)

Going backwards, swap values with children so the tree becomes a heap from the bottom-up, kinda recursively (base case: leafs are heaps)

to_heap is $\mathcal{O}(n)$

Extract-max is $\mathcal{O}(\log n)$, and is called n times (though input size decreases each time), but it still makes for $\mathcal{O}(n \log n)$ steps.

eg. to_heap (underline shows the leafs currently being re-ordered)



[2, 4, 5, 12, 33, 19, 21, 10, 8, 14, 1, 3] – swap $Q[4]$ with $Q[8]$; swap $Q[5]$ with $Q[10]$
 [2, 33, 21, 12, 30, 19, 5, 10, 8, 4, 1, 3] – swap $Q[2]$ with $Q[5]$, then $Q[10]$; swap $Q[3]$ with $Q[7]$
 [33, 30, 21, 12, 4, 19, 5, 10, 8, 2, 1, 3] – swap $Q[1]$ with $Q[2]$, then $Q[5]$, then $Q[10]$

Inner while loop of to_heap is $\mathcal{O}(\lg n)$ (travelling from parent to child)

So running-time of to_heap is

$$\# \text{ nodes in level} \times \text{height of level} = \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots + 1 \cdot \lg n = \sum_{i=2}^{\lg n} \frac{n}{2^i} i = n \sum_{i=2}^{\lg n} \frac{i}{2^i} \approx 2n \in \mathcal{O}(n)$$

Dictionaries, AVL Trees, and Hashing

Dictionary ADT: A set of elements with unique keys

Operation	Description
search(D, k)	Find $x \in D$ where $x.key == k$. If $x \notin D$, return null.
insert(D, x)	Add x with $x.key$ to D . Overwrites data with the same key as $x.key$.
delete(D, x)	Remove x from D . If $x \notin D$, do nothing. <i>(delete by key using delete(D, search(D, k)))</i>

Operation	Speed of Implementations							
	Unsorted		Sorted		Trees		Direct- Access Table	Hash Table
	List	Linked List	List	Linked List	Binary Search	Balanced (eg. AVL)		
search(D, k)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
insert(D, x)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
delete(D, x)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$

- Insert's property of replacing duplicate keys increases many running-times to n
- Binary search trees can be lopsided in shape, causing $\mathcal{O}(n)$ running-time
- For some reason, we don't count traversal as part of running-time when doing delete, so it's $\mathcal{O}(1)$

Binary Search Tree Implementation	Comments
<pre>def search(D, k): if D is None: return None elif k == D.key: return D.item elif k < D.key: return search(D.left, k) else: # k > D.key return search(D.right, k)</pre>	<p>We're assuming D is also binary tree.</p> <p>Constant-time work In worst-case, number of recursive calls equal to height,</p> <p>Thus $\mathcal{O}(h)$, and $\log_2 n \leq h \leq n$</p>
<pre>def insert(D, x): if D is None or x.key == D.key: D = x elif x.key < D.key: insert(D.left, x) else: # x.key > D.key insert(D.right, x)</pre>	<p>Same as above.</p> <p>Assume everything mutates.</p>
<pre>def delete(D, x): if D is None: pass elif x.key == D.key: if D.left is None or D.right is None: D = D.left/right # whichever one that isn't None else: D = extract_min(D.right) # D = extract_max(D.left) also works elif x.key < D.key: delete(D.left, x) else: # k > D.key delete(D.right, x)</pre>	<p>Recall for deleting a node x:</p> <p>If x is not fully-subtreed: - Promote other subtree to original. - If x is leaf, promoted subtree is NAN</p> <p>If x is fully-subtreed (extract_min): - Replace x with left-most subtree of x.right (smallest value larger than x)</p>

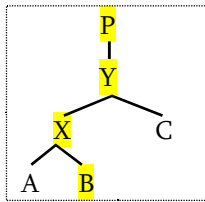
Balanced Search Tree: A tree that rebalances itself as items are inserted/removed to maintain $\Theta(\log n)$ height.

Balance Factor: Of node n , the value $n.right.height - n.left.height$ (count empty sub-leafs as height -1)

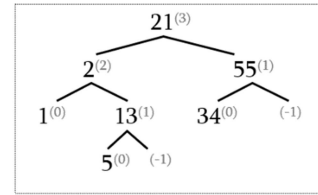
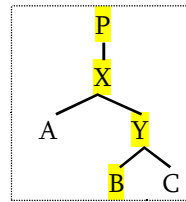
AVL-Invariant: For node n , if $|n.right.height - n.left.height| \leq 1$

AVL-Balanced: A binary tree, if all nodes satisfy the AVL-Invariant

AVL (Adelson-Valesky and Landis) Tree: A binary search tree that is AVL-balanced (ie. leaf levels differ by ≤ 1).



→ Right **Rotation** on Y →
 ← Left **Rotation** on X ←



Right Rotation

```
def rotate_right(D):
    D.right, D.right.right = D, D.right
    D.right.left = D.left.right
    D, D.left = D.left, D.left.left
```

Left Rotation

```
def rotate_left(D):
    D.left.item, D.left.left = D.item, D.left
    D.left.right = D.right.left
    D.item, D.right = D.right.item, D.right.right
```

Augmentation: Adding fields to a basic data structure that doesn't quite solve a problem (inefficient, or does not support a select task), and adjusting original operations to keep fields up to date (eg. AVL trees augment BSTs)

AVL Tree Implementation

```
def rebalance_right(D):
    D.height = max(D.left.height, D.right.height) + 1

    # Left side is longer
    if D.balance_factor == -2:
        if D.left.left.height == D.right.height + 1:
            rotate_right(D) # Case 1
        else:
            rotate_left(D.left)
            rotate_right(D) # Case 2

    # Identical case for rebalance_left(D)
    # D.balance_factor == 2, and you flip left/right

def search(D, k):
    bin_search(D, k)

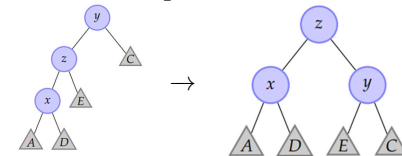
def insert(D, x):
    if D is None or x.key == D.key:
        D = x
    elif x.key < D.key:
        insert(D.left, x)
        rebalance_right(D) # Left side risks being longer
    else: # x.key > S.item.key
        insert(D.right, x)
        rebalance_left(D) # Right side risks being longer

def delete(D, x):
    if D is None:
        pass
    elif x.key == D.key:
        if D.left.height > D.right.height:
            D = extract_max(D.left)
        else:
            D = extract_min(D.right)
    elif x.key < D.key:
        delete(D.left, x)
        rebalance_left(D) # Right side risks being longer
    else: # k > S.root.key
        delete(D.right, x)
        rebalance_right(D) # Left side risks being longer

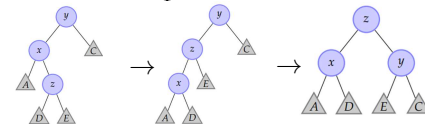
def extract_max(D, x): # Identical case for extract_min(D)
    if D.right is None: # Just switch right/left
```

Comments

Case #1 corresponds to



Case #2 corresponds to



Same implementation as in BSTs.

Same as BSTs, but each node from the root to x needs to rebalance itself.

Unlike BST deletion, it matters if we replace the deleted x with $\max(x.\text{left})$ or $\min(x.\text{right})$.

```

    if D is a right child:
        D.parent.right = D.left
    else:
        D.parent.left = D.left
    return D.pop()
else:
    max = extract_max(D.right)
    rebalance_right(D) # Left side risks being longer
    return max

```

Hash Table: A length- m array D . You can pick m . Each array element is called a **bucket/slot**

Hash Function: A function $h: K \rightarrow \{0, \dots, m-1\}$ mapping possible keys to a **home bucket** – a hash table index.

- Good hash functions are designed so that:
 - $h(k)$ depends on the entire key k (eg. if the key is a string, don't depend on a substring)
 - $h(k)$ spreads keys evenly across $\{0, \dots, m-1\}$ if $m < |K|$
 - $h(k)$ is computationally efficient
- Ultimately, hashing boils down to converting keys \rightarrow int/float $\rightarrow \{0, \dots, m-1\}$
 - *Step 1:* If keys are very different data types, one option is using their memory addresses (ints)
 - *Step 2:* Use multiplication for floats, use division and mod m for integers
 - Good hashing functions are hard to make! Don't worry about the implementation in CSC263.

Perfect Hash Functions: Hash functions with $m \geq |K|$, so there are enough array slots for all keys.

- Unrealistic, as we want small memory (ie. low m) and many allowed keys (ie. high $|K|$)
- **Direct Address Tables:** Hash tables that directly store values at an index determined by the key.

Direct Address Table Implementation	Comments
<pre>def search(D, k): return D[h(k)]</pre>	Think of D as the dictionary and the array at the same time.
<pre>def insert(D, x): D[h(x.key)] = x.item</pre>	Recall that indexing is $\mathcal{O}(1)$
<pre>def delete(D, x): D[h(x.key)] = None</pre>	Assume the hash function is $\mathcal{O}(1)$

Collisions: When two keys map to the same index; inevitable when $m < |K|$. Solve with close/open addressing.

Closed Addressing/Chaining: Where each index stores a linked list of key-value pairs instead of a value

Closed-Addressing Implementation	Comments
<pre>def search(D, k): lst = D[h(k)] return lst.find(k).value</pre>	Pretend linked list contains x , which has x .key with x .value; the list is searchable based on k ; and None is returned on failure to find. $\mathcal{O}(n)$.
<pre>def insert(D, x): lst = D[h(x.key)] lst.insert(x, 0)</pre>	Insertion at the start of a linked list is $\mathcal{O}(1)$, making this $\mathcal{O}(1)$
<pre>def delete(D, x): lst = D[h(x.key)] lst.delete(x)</pre>	If every key gets mapped to the same index, $\mathcal{O}(n)$ (since list delete includes list search). But what exactly happens depends on h ; getting n steps is statistically pretty much impossible.

Average-Case Running-Time Analysis, Closed Addressing, Searching and Deleting

Simple Uniform Hashing Assumption: That $\mathbb{P}(h(k) = i) = \frac{1}{m}$ for any key k , index i of the length- m array Finding the list, $D[h(k)]$, is one step.

Traversing the list is $L(k) \leq \text{len}(\text{lst})$ steps.

$$\begin{aligned}
 T(n) &= 1 + L(k) \\
 \mathbb{E}[T(n)] &= 1 + \mathbb{E}[L(k)] \\
 &= 1 + \sum_{k \in K} \mathbb{P}(K = k) L(k) \\
 &= 1 + \sum_{i=0}^{m-1} \sum_{k \in K, h(k)=i} \mathbb{P}(K = k) L(k) \quad (\text{group keys by bucket}) \\
 &\leq 1 + \sum_{i=0}^{m-1} \sum_{k \in K, h(k)=i} \mathbb{P}(K = k) \text{len}(\text{lst}_i) \\
 &= 1 + \sum_{i=0}^{m-1} \text{len}(\text{lst}_i) \mathbb{P}(h(k) = i) \\
 &= 1 + \sum_{i=0}^{m-1} \text{len}(\text{lst}_i) \cdot \frac{1}{m} \\
 &= 1 + \frac{1}{m} \sum_{i=0}^{m-1} \text{len}(\text{lst}_i) \\
 &= 1 + \frac{n}{m}
 \end{aligned}$$

Recall the length of all sublists sum to n .

Load Factor: Of a hash table, the value $\frac{n}{m}$; the ratio of all stored keys to hash table size.

Open Addressing: Where each index contains one key, and keys whose indices are occupied get new indices.

- We must choose m such that $m \geq n$
- The “parameterized” hash function is $h: K \times \mathbb{N} \rightarrow \{0, \dots, m-1\}$.
- **Probe Sequence:** Of key k , the valid indices it can have: $h(k, 0), h(k, 1), h(k, 2), \dots$

Open-Addressing Implementation	Comments
<pre>def search(D, k): i = 0 while D[h(k, i)] is not None and D[h(k, i)].key != k: i += 1 return D[h(k, i)]</pre>	<p>Iterate through $h(k, i)$ for $i \in \mathbb{N}$ until None or the right value is found. $\mathcal{O}(n)$.</p>
<pre>def insert(D, x): i = 0 while D[h(k, i)] is not None: i += 1 D[h(k, i)] = x</pre>	<p>Note we should also check $i < m$. $\mathcal{O}(n)$. EDIT THIS!!!! KEY K whatever</p>
<pre>def delete(D, x): i = 0 while D[h(k, i)] is not None and D[h(k, i)].key != k: i += 1 if D[h(k, i)].key == key: [h(k, i)] = Deleted</pre>	<p>Deleting will leave a “gap” value in the probe sequence. We need a sentinel/flag/dummy value, Deleted, so searching won’t end at the deleted value. $\mathcal{O}(n)$.</p>

Linear Probing: Setting alternate indices to linear offsets $h(k, i) = h(k) + \alpha i \pmod{m}$

Quadratic Probing: Setting alternate indices to quadratic offsets $h(k, i) = h(k) + \alpha i + \beta i^2 \pmod{m}$

Double Hashing: Given two hash functions h_1, h_2 , $h(k, i) = h_1(k) + i \cdot h_2(k) \pmod{m}$

Clustering: Phenomenon where contiguous index ranges are occupied, causing insertions to take a long time

- Linear probing is worst – let $\alpha = 1$, insert 100 keys $D[0]$, then insert key at $D[1]$ – 99 index checks
- Quadratic probing better, but bad – insert 100 keys at $D[0]$, then insert key at $D[0]$ – 100 index checks
- Double hashing has m^2 possible probe sequences, making clusters unlikely!
 - Only works well when $m > 2n$. Still, when implemented well, less overhead than chaining
 - Average-case analysis is messy, but same results as closed-addressing

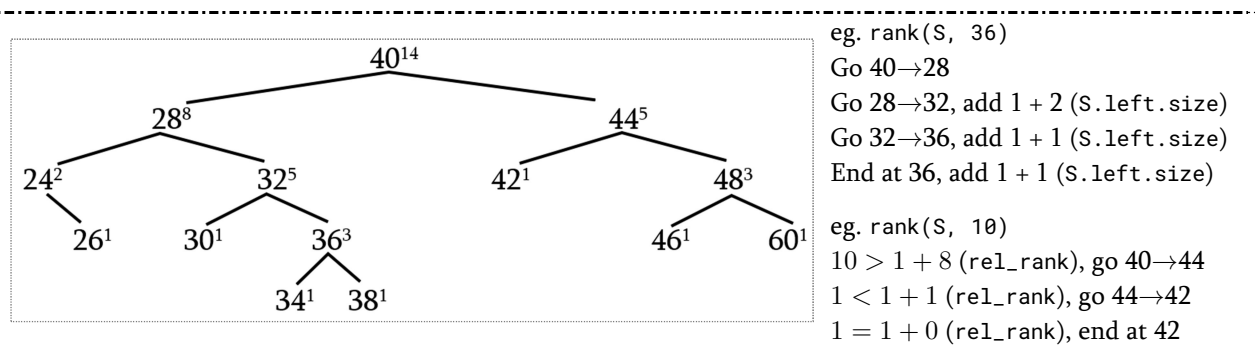
AVL Trees	Hashing
Slow $\mathcal{O}(\log n)$ speed, but allows <u>ordered</u> processing of values	Ideal for <u>everything else</u> that isn’t ordered. As hash functions spread keys out evenly across indices, we can’t keep orderings. But $\Theta(1)$ average speed!

Ordered Sets

Ordered Set ADT: A set of ordered elements

Operation	Description	Speed of Implementation	
		AVL Tree	AVL Tree (Augmented)
search(S, x)	Find $x \in S$. If $x \notin S$, return null.	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
insert(S, x)	Add x to S.	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
delete(S, x)	Remove x from S. If $x \notin S$, do nothing.	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
select(S, k)	Return the k-th smallest item from S.	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
rank(S, x)	Return x's rank in terms of the smallest items in S.	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Augmented AVL Tree Implementation	Comments
<pre>def rank(S, x): if S is None: return None elif S == x: return 1 + S.left.size elif S > x: return rank(S.left) else: # S < x return 1 + S.left.size + rank(S.right) def select(S, k): rel_rank = 1 + S.left.size if S is None: return None elif rel_rank == k: return S elif rel_rank > k: return select(S.left, k) else: # rel_rank < k return select(S.right, k - rel_rank)</pre>	<p>If $x \notin S$, then we return null.</p> <p>Size is an augmented field representing the size of a node's subtrees.</p> <p>On insertion/deletion, update size after the rebalancing calls with $S.size = 1 + self.left.size + self.right.size$</p> <p>On rotation, updating size is easy – just recalculate height of all nodes involved in the rotation.</p> <p>Both operations on the left are $\mathcal{O}(\log n)$</p>



Quicksort

Quicksort: Recursive sorting algorithm that divides into two lists whose elements are below/over a pivot item.

- Worst case of $\Theta(n^2)$ – the partitions always divide into sizes of 1 and $n - 1$
- Average case of $\Theta(n \log n)$

```
def quicksort(A):
    if len(A) <= 1:
        return A
    pivot = A[0] // Deterministic quicksort
    l1, l2 = partition(A[1:], pivot)

    l1 = quicksort(l1)
    l2 = quicksort(l2)
    return l1 + [pivot] + l2

def partition(A, pivot):
    l1, l2 = [], []
    for item in A:
        if item <= pivot:
            l1.append(item)
        else:
            l2.append(item)
    return l1, l2
```

Randomized Quicksort: Quicksort that selects a random pivot.

- Quicksort is fast for mixed lists but predictably slow on almost-sorted lists – this is no good!
- For large lists, the probability of many bad partitions is negligibly tiny, so random pivot is fine.

Average-Case Running-Time Analysis, Quick Sort

Define sample space $S_n = \{\text{all permutations of } [1, \dots, n]\}$

Define A , the list we get, a random variable with uniform probability distribution

Define $T(A)$, number of comparisons done in running Quick Sort on A

Want $\mathbb{E}[T(A)]$, expected comparisons done in running Quick Sort on A

Since $\mathbb{E}[T(A)]$ is a constant factor away from running-time, this is a fine substitute

Define $X_{i,j} = \begin{cases} 1 & i, j \text{ are compared during QuickSort} \\ 0 & \text{else} \end{cases}$ for $1 \leq i < j \leq n$

i, j are compared at most once. For i, j to be compared,

- i, j are in the same sublist
- One of i, j is chosen as the pivot

Also, while $\{i, \dots, j\}$ are still in the same sublist during run-time, nothing between i, j is chosen as pivot before one of i, j ; otherwise, i, j are compared to that pivot and put into separate sublists.

The probability i, j is chosen as pivot over all values in $\{i, \dots, j\}$ is $\frac{2}{j-i+1}$

If we iterate through every unique i, j combination, then

$$\begin{aligned}
 T(A) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j} \\
 \therefore \mathbb{E}[T(A)] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{i,j}] \\
 &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{P}[X_{i,j} = 1] \\
 &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
 &= 2 \sum_{i=1}^{n-1} \sum_{j'=1}^{n-i} \frac{1}{j'+1} \quad (\text{set } j' = j - i) \\
 &= 2 \left(\sum_{j'=1}^{n-1} \frac{1}{j'+1} + \dots + \sum_{j'=1}^1 \frac{1}{j'+1} \right) \\
 &= 2 \left(\frac{n-1}{1+1} + \frac{n-2}{2+1} + \frac{n-3}{3+1} + \dots + \frac{n-(n-1)}{(n-1)+1} \right)
 \end{aligned}$$

$$\begin{aligned}
 &= 2 \sum_{i=1}^{n-1} \frac{n-i}{i+1} \\
 &= 2 \left(\sum_{i=1}^{n-1} \frac{n+1-(i+1)}{i+1} \right) \\
 &= 2 \sum_{i=1}^{n-1} \left(\frac{n+1}{i+1} - 1 \right) \\
 &= 2 \sum_{i=1}^{n-1} \frac{n+1}{i+1} - 2 \sum_{i=1}^{n-1} 1 \\
 &= 2(n+1) \sum_{i=1}^{n-1} \frac{1}{i+1} - 2(n-1) \in \Theta(n \log n)
 \end{aligned}$$

We know this as from the Harmonic Series, $\sum_{i=1}^n \frac{1}{i} \approx \ln(n + \gamma)$, where $\gamma \approx 0.577$ is the Euler-Mascheroni constant.

So, $\sum_{i=1}^{n-1} \frac{1}{i+1} \in \Theta(\log n)$.

Note:

It doesn't matter if something outside of $\{i, \dots, j\}$ is chosen as pivot first; all of $\{i, \dots, j\}$ would go still to the same sublist.

Amortized Analysis

Amortized Analysis: The worst-case averaged cost per operation in a predefined sequence of m operations

- Relies on knowing a distinct property of the operation – some methods better for some problems

Aggregating Method: Calculating worst-case total cost of m operations, $T(m)$, dividing by m

Accounting Method: Assign each operation a constant value/“charge” k , such that $T(m) \leq mk$

- Can be difficult to work with $T(m) \leq mk$ directly; consider alternative interpretations:
 - Find a line $y = \alpha x$ where $\alpha \geq$ cumulative cost of m operations, $\sum_{i=1}^m T(i)$
 - “Charge” each operation $\$k$ and subtract the running-time/“cost”; the remainder is “credit” and carries over to the next operation. We want Credit ≥ 0 for all operations
 - Credit Invariant:** Similar to loop invariant – a statement always holding about $T(m)$, usually relating leftover credit to the operation number.

eg. Empty max-heap, insert m times.

Since insert is $\mathcal{O}(\log n)$, we get $T(m) = \log 1 + \dots + \log m \leq m \log m$, or at most $\frac{T(m)}{m} = \log m$ in amortized time

eg. Binary counter data structure that only increments by 1. Flipping 1 bit is 1 step.

Decimal	Binary	Bits Flipped	<u>Aggregating Method</u>
0	000...00000		The number of bits flipped during each increment differs depending on the number it is changing to:
1	000...0000 <u>1</u>	1	<ul style="list-style-type: none"> The 1st bit changes every 1 increment
2	000...0001 <u>0</u>	2	<ul style="list-style-type: none"> The 2nd bit changes every 2 increments
3	000...0001 <u>1</u>	1	<ul style="list-style-type: none"> The 3rd bit changes every 4 increments
4	000...0010 <u>0</u>	3	After m increments,
5	000...0010 <u>1</u>	1	<ul style="list-style-type: none"> The 1st bit changes m times
6	000...0011 <u>0</u>	2	<ul style="list-style-type: none"> The 2nd bit changes $\frac{m}{2}$ times
7	000...0011 <u>1</u>	1	<ul style="list-style-type: none"> The 3rd bit changes $\frac{m}{4}$ times
8	000...0100 <u>0</u>	4	<ul style="list-style-type: none"> The $\log_2 m$-th bit changes 1 time
9	000...0100 <u>1</u>	1	So the worst-case total number of steps for m operations is
10	000...0101 <u>0</u>	2	$T(m) = \sum_{i=0}^{\log_2 m - 1} \frac{m}{2^i} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m \text{ (trust me)}$
11	000...0101 <u>1</u>	1	So the amortized time is at most $\frac{T(m)}{m} = 2$ steps, or $\mathcal{O}(1)$!
12	000...0110 <u>0</u>	3	<u>Accounting Method</u>
13	000...0110 <u>1</u>	1	Let us “charge” \$2 per increment operation.
14	000...0111 <u>0</u>	2	Consider an arbitrary bit. In some increment operation, we flip it from 0 to 1, “costing” \$1 – store the extra \$1 as “credit”
15	000...0111 <u>1</u>	1	In a later increment operation, the bit ends up flipped 1 to 0 again, “costing” \$1 – we use the stored \$1 to cover that cost.
16	000...1000 <u>0</u>	5	Thus there is always enough “credit” to pay for an increment and $T(m) \leq 2m$, so the amortized time is $2 \in \mathcal{O}(1)$.
17	000...1000 <u>1</u>	1	
18	000...1001 <u>0</u>	2	
19	000...1001 <u>1</u>	1	
20	000...1010 <u>0</u>	3	
21	000...1010 <u>1</u>	1	
22	000...1011 <u>0</u>	2	
...	

Line Intercept Method: My own method to find the “charge” of a $\mathcal{O}(1)$ amortized operation.

- Find two operation numbers x_1, x_2 that have higher running-times than all previous operations (disclaimer: the x_i must occur predictably in a pattern with no ceilings/floors)
- Find the cumulative running-time for those operation numbers, y_1, y_2
- Find the slope of the line passing (x_1, y_1) and (x_2, y_2) , a lower-bound to the amortized cost.

eg. Binary counter data structure, line intercept method (This method is informal! Use only for rough work.)

Note that operation number 2^k for some $k \in \mathbb{N}$ has running-time k , the highest running-time “so far”.

Consider operation numbers $x_1 = 2, x_2 = 4$ with cumulative running-times $y_1 = 3, y_2 = 7$.

The line passing $(2,3)$ and $(4,7)$ is $y = 2x - 1$, so the amortized time is at least $2 \in \mathcal{O}(2)$.

Dynamic Array: An array of initial capacity 1 that, when full, copies all elements to a new array of double size.

- Saves more memory compared to an array of predefined size
- **Expansion Factor:** The factor α multiplied to the capacity in each list expansion (ie. 2 by default).
 - If $\alpha < 2$, the list resizes more often, leading to higher costs
 - If $\alpha > 2$, the list resizes less often, leading to lower costs

$$T(n) = \begin{cases} 2n + 1 & n = \alpha^k \text{ for some } k \in \mathbb{N} \\ 1 & \text{else} \end{cases}$$

eg. Dynamic array insertion on an empty list has amortized time of $\mathcal{O}(1)$.

Accounting Method

The running-time is $T(n) = \begin{cases} 2n + 1 & n = 2^k \text{ for some } k \in \mathbb{N} \\ 1 & \text{else} \end{cases}$ - 1 to insert, $2n$ to access/copy all n elements

Consider $k = 5$, we claim $T(m) \leq 5m$ and proceed with complete induction:

Base Case: $m = 1$

$$T(1) = 2(1) + 1 = 3 < 5 = 5(1)$$

Recursive Case: $\forall m \geq 1, (\forall m' \leq m, T(m') \leq T(m)) \Rightarrow T(m + 1)$

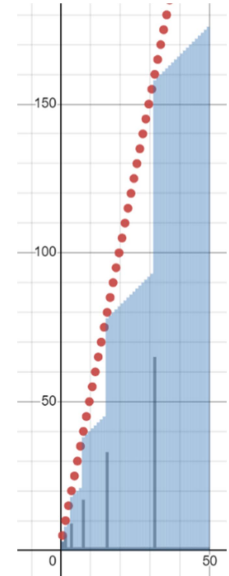
Case 1: $m + 1 \neq 2^k$ for some $k \in \mathbb{N}$

Then $T(m + 1) = T(m) + 1 \leq 5m + 1 < 5(m + 1)$

Case 2: $m + 1 = 2^k$ for some $k \in \mathbb{N}$

Then $\frac{m+1}{2} = 2^{k-1}$, and $T(\frac{m+1}{2}) = T(\frac{m+1}{2} + 1) = \dots = T(m) = 1$

$$\begin{aligned} T(m + 1) &= T\left(\frac{m+1}{2}\right) + T\left(\frac{m+1}{2} + 1\right) + \dots + T(m) + T(m + 1) \\ &= T\left(\frac{m+1}{2}\right) + 1 + \dots + 1 + 2m + 1 \\ &= T\left(\frac{m+1}{2}\right) + \left(\frac{m+1}{2} - 1\right) + 2m + 1 \\ &\leq 5\left(\frac{m+1}{2}\right) + \frac{5m+1}{2} \end{aligned} \rightarrow \begin{aligned} &= \frac{10m+6}{2} \\ &= 5m+3 \\ &< 5(m+1) \end{aligned}$$



How did get $k = 5$? Line intercept method works. Also, think of a single array item; you need \$1 to insert it, \$2 to access/copy itself once, and \$2 to recursively access/copy another item that accessed/copied itself once. You can also prove a credit invariant – each newly-created slot (when the list expands) will have \$2 credit.

Aggregating Method

The total running-time is

$$\begin{aligned} T(m) &= \sum_{i=0}^{m-1} \begin{cases} 2i + 1 & i = 2^k \text{ for some } k \in \mathbb{N} \\ 1 & \text{else} \end{cases} \\ &= \sum_{i=0}^{m-1} \left[1 + \begin{cases} 2i & i = 2^k \text{ for some } k \in \mathbb{N} \\ 0 & \text{else} \end{cases} \right] \\ &= m + \sum_{i=0}^{\lfloor \log_2(m-1) \rfloor} 2(2^i) \\ &= m + 2(2^{\lfloor \log_2(m-1) \rfloor + 1} - 1) \\ &= 4(2^{\lfloor \log_2(m-1) \rfloor + 1}) + 4m - 2 \end{aligned}$$

Consider the bound

$$\begin{aligned} \log_2(m-1) - 1 &\leq \lfloor \log_2(m-1) \rfloor \leq \log_2(m-1) \\ 2^{\log_2(m-1)-1} &\leq 2^{\lfloor \log_2(m-1) \rfloor} \leq 2^{\log_2(m-1)} \\ \frac{1}{2}(m-1) &\leq 2^{\lfloor \log_2(m-1) \rfloor} \leq m-1 \\ 3m-4 &\leq 4(2^{\lfloor \log_2(m-1) \rfloor + 1}) + 4m - 2 \leq 5m - 6 \\ \therefore 3m-4 &\leq T(m) \leq 5m - 6 \end{aligned}$$

So the amortized cost is

$$3 - \frac{4}{m} \leq \frac{T(m)}{m} \leq 5 - \frac{6}{m}$$

In other words, 3 to 5-ish steps, constant-time.

Graphs

Graph ADT: A tuple $G = (V, E)$ of a set of vertices and set of edges. By convention, $|V| = n, |E| = m$

Undirected Graph: Graphs where edges have no direction – they just connect nodes

Directed Graph: Graphs where edges have directions – nodes point to other nodes. Self-cycles are allowed

Neighbourhood: Of a node, all nodes directly connected to it by an edge

Path: Of length n , an ordered collection of $n + 1$ nodes connected by edges

Cycle: A path where the first/last vertices are the same. Accepts cycles of length 1 (in directed graphs) and 2

Simple: A path/cycle, if there are no duplicate vertices or edges.

Adjacency Matrix: Square matrix A that stores graph edges where $A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \text{else} \end{cases}$ for $i, j \in \{1, \dots, n\}$

- For undirected graphs,
 - $A_{i,i} = 0$ Since nodes cannot connect to themselves
 - A is symmetric Since if (v_i, v_j) is an edge, then (v_j, v_i) is the same edge. Redundant space!
- Space complexity: $\Theta(n^2)$
- Query if neighbours, two nodes: $\Theta(1)$ – return $A_{i_1, j_1} = A_{i_2, j_2}$, for nodes $(i_1, j_1), (i_2, j_2)$
- Query neighbourhood of node: $\Theta(n)$ – return $\{A_{i, j_k}\}_{k \in \{1, \dots, n\}} \cup \{A_{i_k, j}\}_{k \in \{1, \dots, n\}}$, for node (i, j)

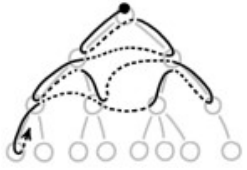
Adjacency Lists: A list of lists V where $V[i]$ is a list of every node to which node i points

- Space complexity: $\Theta(n + m)$
- Query if neighbours, two nodes: $\Theta(n)$ – return u in v . neighbours (a list)
- Query neighbourhood of node: $\Theta(1)$ – return v . neighbours

Breadth-First Search (BFS): Algorithm to find nodes directly/indirectly connected to a source node $s \in V$.

Iteration i finds all nodes whose distance from s is i . Requires augmenting vertices with:

- $s.parent$ Predecessor/parent of v . Initialize as nothing
- $s.found$ False for not found (by the search), True for found.
- $s.dist$ Distance of v to u via the discovered path between them. Initialize as ∞ .

Breadth-First Search	Comments
<pre>def bfs(G, s): # Initialize vertices for v in G.V: v.found = v == s v.dist = 0 if v == s else ∞ Q = Queue() enqueue(Q, s) while not is_empty(Q): u = dequeue(Q) for v in neighbourhood(G, u): if not v.found: v.found = True v.parent = u v.dist = u.dist + 1 enqueue(Q, v)</pre>	<p>First, initialize vertices. Then, initialize the source node.</p> <p>For each node, find/fill info for undiscovered neighbours, mark them discovered, and repeat.</p> <p>The found nodes/edges <u>form a tree!</u></p> <p>Running-time of $\Theta(n + m)$</p> <ul style="list-style-type: none"> • At most n vertices enqueued/dequeued • So all adjacency lists are found ≤ 1 time • Constant-work for each item in adjacency list (accessed in call to <code>neighbourhood</code>) – across all loops, look at every edge in every adjacency list, so m items 

Proof, correctness of $s.dist$

Define $\delta(u, v) = \min(\text{dist}(u, v))$

Show $v.dist = \delta(s, v)$

Lemma 1: For edge (u, v) , we have $\delta(s, v) \leq \delta(s, u) + 1$

Minimum distance from s to u is $\delta(s, u)$. Take edge from u to v , get path from s to v , length $\delta(s, u) + 1$

Minimum distance from s to v is thus $\delta(s, v) \leq \delta(s, u) + 1$

Lemma 2: $\delta(s, v) \leq v.dist$

By definition, $\delta(s, v)$ is minimum distance from s to v , so BFS's $v.dist$ can't be smaller than $\delta(s, v)$.

Lemma 3: At any point in loop with $Q = [v_1, \dots, v_k]$, $v_1.dist \leq \dots \leq v_k.dist$ and $v_k.dist \leq v_1.dist + 1$

Can use proof by induction based on number of enqueues – below is more informal

Start with single u in the queue.

When u is dequeued, enqueue v_1, \dots, v_{n_1} where $v_i.dist = u.dist + 1$, queue has two sizes.

When v_i is dequeued, enqueue w_1, \dots, w_{n_2} where $w_i.dist = v_i.dist + 1$, queue has two sizes.

To dequeue w_1 , must first dequeue all previous v_i , removing a size. Newly enqueued values add a size.

Suppose, for contradiction, that $\exists v \in V, v.dist \neq \delta(s, v)$

By Lemma 2, $\delta(s, v) < v.dist$

By PWO, there exists a $v_0 \in V$ where $\delta(s, v_0) < v_0.dist$ with the minimum $\delta(s, v_0)$

We know $v_0 \neq s$, since $\delta(s, s) = 0 = s.dist$

Thus, there exists shortest path from s to v_0 . Let u_0 be second-last node of the path, and as $\delta(s, v_0)$ is minimal,
 $\delta(s, v_0) = \delta(s, u_0) + 1$

Then $v_0.dist > \delta(s, v_0) = \delta(s, u_0) + 1 = u_0.dist + 1$ (since u_0 is on shortest path from s to v_0)

Consider when u_0 is about to be dequeued by BFS:


If v_0 is undiscovered, BFS finds it, sets $v_0.dist = u_0.dist + 1$, a contradiction

If v_0 is already in the queue, by Lemma 3, $v_0.dist \leq u_0.dist + 1$, a contradiction

If v_0 has already been dequeued, then $v_0.dist \leq u_0.dist$, a contradiction

Depth-First Search (DFS): Algorithm to find nodes directly/indirectly connected to a source node $s \in V$. Finds the deepest/farthest/child nodes first. Requires augmenting vertices with:

- $s.parent$ Predecessor/parent of v . Initialize as nothing
- $s.time_found$ Number representing when the node is detected
- $s.time_done$ Number representing when the node is fully explored

Depth-First Search	Comments
<pre> time = 0 def dfs_find_all(G): # Initialize vertices for v in G.V: v.time_found = ∞ v.time_done = ∞ time = 0 for v in G.V: if v.time_found == ∞: dfs_visit(G, v) def dfs_visit(G, v): time += 1 v.time_found = time # Do something with v for u in neighbourhood(G, v): if u.time_found == ∞: u.parent = v dfs_visit(G, v) time += 1 v.time_done = time </pre>	<p>This is actually a template for an algorithm that iterates through every node using a depth-first search.</p>  <p>Checking $v.time_found == \infty$ is equivalent to checking $v.found == \text{False}$.</p> <p>The time counter starts at 1, and increases every time a node is visited/fully-explored.</p> <p>The algorithm finds every not-found neighbour of a vertex and recursively visits them.</p> <p>The found nodes/edges <u>form a tree!</u></p> <p>Running-time of $\Theta(n + m)$</p> <ul style="list-style-type: none"> • <code>dfs_visit</code> called 1 time for all vertices • So adjacency lists are examined once

Edge Classification: Classification of edges based on the tree generated by the DFS

- **Tree Edge:** Traversed in DFS
- **Back Edge:** Not traversed in DFS, points from child to parent in DFS tree; creates cycles
- **Forward Edge:** Not traversed in DFS, points from parent to child in DFS tree; only in undirected graphs
- **Cross Edge:** Not traversed in DFS, two connecting nodes are unrelated; only in undirected graphs

Parentheses Theorem: After the DFS, for any nodes $a, b \in V$,

- a is ancestor of $b \Leftrightarrow a.timeFound < b.timeFound < b.timeDone < a.timeDone$
- b is ancestor of $a \Leftrightarrow b.timeFound < a.timeFound < a.timeDone < b.timeDone$
- a, b unrelated $\Leftrightarrow a.timeFound < a.timeDone < b.timeFound < b.timeDone$ or $b.timeFound < b.timeDone < a.timeFound < a.timeDone$

Topological Sorting: Ordering vertices in directed graphs so $(u, v) \in E \Rightarrow u$ appears before v

- G is topologically sortable $\Leftrightarrow G$ has no cycle \Leftrightarrow DFS finds no back edges
- Findable by ordering vertices in DFS by descending finish time

Proof: Topological sorting of G findable by sorting DFS vertices in descending finish time

Consider recursive call of `dfs_visit`(G, u), just found node v .

Then $u.timeFound < \infty$ and $u.timeDone = \infty$

If $v.timeFound = \infty$ (ie. unexplored),

v is child of u , $u.timeFound < v.timeFound < v.timeDone < u.timeDone$

If $v.timeFound < \infty, v.timeDone < \infty$ (ie. already explored),

v is unrelated to u , $v.timeDone < u.timeFound < u.timeDone$ – topological sort not applicable

If $v.timeFound < \infty, v.timeDone = \infty$ (ie. still exploring),

We must have been previously exploring v , got to u , and now found a back edge to v again – cycle!

Directed Acyclic Graph (DAG): A directed graph with no cycles

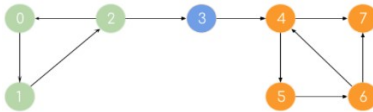
Weakly-Connected Component: A group of vertices V where $\forall u, v \in V$, there's a path $u \rightarrow v$ or $v \rightarrow u$

Strongly-Connected Component: A group of vertices V where $\forall u, v \in V$, there's a path $u \rightarrow v$ and $v \rightarrow u$

- If strongly-connected components are bunched into a single node, graph becomes a DAG
- Findable by running DFS on G with vertices processed according to a topological ordering of G but with edges pointed in reverse – every DFS tree is a strongly-connected component

eg. “Kosaraju’s Algorithm”, strongly-connected components

The flipped graph is



After DFS, a topological ordering is

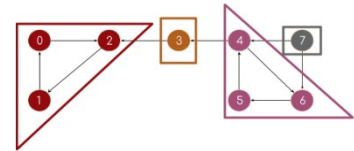
$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$

DFS on node 0 returns the left triangle. (nodes 1, 2 found, skip them)

DFS on node 3 returns one point

DFS on node 4 returns the right triangle (nodes 5, 6 found, skip them)

DFS on node 7 returns one point



Weighted Graph: A graph with numbers assigned to edges, usually the “cost/distance/weight” of the edge.

Spanning Tree: Of a graph, a tree containing all its nodes and a subset of its edges.

- Acyclic
- Connected
- $n - 1$ edges
- Adding any edge creates a cycle
- Removing any edge disconnects it into two subtrees

Minimum Spanning Tree: A minimally-weighted spanning tree of a connected, undirected, weighted graph

Prim’s Algorithm, Minimum Spanning Tree	Comments
<pre>def prim(G, s): T = Tree() T.V.add(s) Q = MinPriorityQueue() for v in G.V: v.priority = 0 if v == s else ∞ v.parent = None Q.enqueue(v) while not Q.is_empty(): # Add node with lowest edge weight to tree u = Q.extract_min() if u != s: T.E.add((u.parent, u)) # Find minimum edge weights of newly-added node for v in neighbourhood(G, u): if v in Q and weight((u, v)) < v.priority: v.parent = u v.priority = weight((u, v)) Q.bubble_up(v) # As v.priority changed # Ideally, we should also keep track of v's index in Q # so that bubble_up doesn't have to search for it</pre>	<p>Pick a source $s \in V$.</p> <p>Find all edges that point to a <u>new node</u>.</p> <p>Pick edge with <u>lowest weight</u>, add new node.</p> <p><u>Repeat</u> until done!</p> <p>The implementation uses a min priority queue storing nodes, where priority is the currently-found minimum edge weights.</p> <p>Running-time of $\Theta((n + m) \log n)$</p> <ul style="list-style-type: none"> • Initialization is $\Theta(n)$ • Main queue loop is n iterations. ExtractMin is $\Theta(\log n)$ • Inner for loop, over main loop, finds all m adjacency lists. Heap bubble up (ie. the helper function swapping v with parents until heap property) is $\Theta(\log n)$ <p>Possible to reduce to $\Theta(m + n \log n)$ using Fibonacci Heaps (not in CSC263)</p>

Kruskal's Algorithm, Minimum Spanning Tree, Version #1	Comments
<pre>def kruskal(G): T = Tree() E = sorted(list(G.E)) # assume this sorts by weight for e in E: bfs(T, e[0]) if e[1].found: T.V.update({e[0], e[1]}) T.E.add(e) return T</pre>	<p>Sort edges by ascending weight</p> <p>Iterate through edges, add edge and nodes if <u>not connected by tree's current edges</u>.</p> <p>Running-time of $\Theta(mn)$</p> <ul style="list-style-type: none"> For loop is m iterations BFS is $\Theta(n)$ (since $m \leq n$, tree!) <p>We will see a more efficient version of this.</p>

Disjoint Sets

Disjoint Set ADT: A collection of non-empty and disjoint sets S_1, \dots, S_k where $S_1 \cup \dots \cup S_k \subseteq S$ for some S

Operation	Description	Speed of Implementations					
		#1	#2	#3	#4	#5	#6
make_set(D, x)	Add set {x} to D.	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
find_set(D, x)	Return the set containing x (or None)	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log m)$	$\mathcal{O}(\log^* m)$
union(D, x, y)	Replace the sets s1, s2 having x, y with $s1 \cup s2$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(\log m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

- By convention, we identify sets with a unique representative element from the set
 - make_set sets x to be the representative
 - find_set returns the set's representative
 - union creates a new representative (depends on implementation)
- Assume each element of each set can be accessed from outside via an attribute

Circular Linked List: A linked list where the last node points to the front.

- Think of disjoint sets like dictionaries with a list of linked list nodes (a single item of a disjoint set): keys are **node items**, values are **indices of the nodes**. Linked lists are disjoint sets.
 - In truth, assume we can access each element of each linked list directly via some outer attribute/pointer, and there is no need for a wrapper class and list of linked list nodes.
- Idea #1:** First linked list node is representative, each node contains a boolean field is_rep

Idea #1 Implementation	Comments
<pre>def make_set(D, x): D.nodes.append(Node(x, True)) # is_rep True D[x] = len(D.nodes) - 1</pre>	<p>Assume the dictionary contains a list of nodes, and nodes start as representatives.</p> <p>In Python, we'll represent storing a pointer to 1 outside as a dictionary assigning key x to 1's index.</p>
<pre>def find_set(D, x): n = D.nodes[D[x]] if n is None: return None while not n.is_rep: n = n.next return n.item</pre>	<p>Find the linked list node.</p> <p>If it exists, iterate through the linked list until it finds the representative (all lists have 1 rep.)</p> <p>Running-time of $\mathcal{O}(m)$, where m is the size of the linked list.</p>
<pre>def union(D, x, y): n1 = D.nodes[D[x]] n2 = D.nodes[D[y]] n1.next = n2 if n2.next is None else n2.next n2.next = n1 if n1.next is None else n1.next n2.is_rep = False D.delete(y)</pre>	<p>Find representatives of both lists, swap what they point to. This combines both lists like a set.</p> <p>Remove the second representative's representative status, and remove the pointer/key from the 2nd list.</p> <p>Assume dictionaries delete by key.</p>

Amortized Worst-Case Running-Time Analysis in Idea #1

Instead of calculating the exact cost of operations, just find a lower and upper bound in this case.

Upper Bound: The worst operation is FindSet, which is $\mathcal{O}(m)$. The largest possible list size is m after m operations. Total is $\mathcal{O}(m^2)$ time.

Lower Bound: Consider $\frac{m}{3}$ MakeSet, $\frac{m}{3} - 1$ Union, and $\frac{m}{3} + 1$ FindSet. The last operation is on list of size $\frac{m}{3}$, so running-time is $\mathcal{O}(\frac{m}{3})$, which is done $\frac{m}{3} + 1$ times, so total is $\mathcal{O}(\frac{m^2}{9})$ time.

- **Idea #2:** Instead of `is_rep`, consider a field `rep` that points to the representative.

Idea #2 Implementation	Comments
<pre>def make_set(D, x): n = Node(x) n.rep = n D.nodes.append(n) D[x] = len(D.nodes) - 1</pre>	Exact same code as #1, but setting the node's representative to point to itself.
<pre>def find_set(D, x): n = D.nodes[D[x]] if n is None: return None return n.rep</pre>	No more looping results in a running-time of $\mathcal{O}(1)$
<pre>def union(D, x, y): n1 = D.nodes[D[x]] n2 = D.nodes[D[y]] n1.next = n2 if n2.next is None else n2.next n2.next = n1 if n1.next is None else n1.next while n2.rep != n1: n2.rep = n1 n2 = n2.next D.delete(y)</pre>	<p>Now we have to iterate through every element of one set, changing the attribute <code>rep</code> to point to the new representative, resulting in $\mathcal{O}(m)$ time.</p> <p>If we have $\frac{m}{2}$ MakeSet, $\frac{m}{2}$ Union, and constantly append single items to a large set, the result is amortized $\Omega(m^2)$ time total, so no improvement.</p>

- **Idea #3:** Idea #2, but list nodes also store field `size` (initialized to 1), the list's size ("Union-by-weight").

Idea #3 Implementation	Comments
<pre>def union(D, x, y): n1 = D.nodes[D[x]] n2 = D.nodes[D[y]] n1.next = n2 if n2.next is None else n2.next n2.next = n1 if n1.next is None else n1.next max = n1 if n1.size >= n2.size else n2 min = n2 if n1.size >= n2.size else n1 max.size += min.size while min.rep != max: min.rep = max min = min.next D.delete(y)</pre>	<p>Keep track of the size of each linked list, and append the longer list to the shorter list. This way, we iterate as many times as the shorter list.</p> <p>Only need to update size in the new representative.</p> <p>Consider $m + n$ operations, m MakeSet, n Union, where $n < m$.</p> <p>In worst-case for Union, lists appends to a bigger list, so list size at least doubles. Since m items in total, unions that cause doubling occur at most $n = \log_2 m$ times. Total time is $\mathcal{O}(n + m \log m) \in \mathcal{O}(m \log m)$</p>

Inverted Tree: A tree whose nodes store parents instead of children. The parent of the root is itself.

- **Idea #4:** Like #1, but instead of linked lists, we have inverted trees. Roots are representatives.

Idea #4 Implementation	Comments
<pre>def make_set(D, x): n = Node(x) n.parent = n D.nodes.append(n) D[x] = len(D.nodes) - 1 def find_set(D, x): n = D.nodes[D[x]] if n is None: return None while n != n.parent: n = n.parent return n def union(D, x, y): n1 = find_set(D, x) n2 = find_set(D, y) y.parent = x</pre>	<p>Exact same code as #2, but setting the parent to itself instead of setting the representative.</p> <p>Similar code to #1, where we keep finding parents until we find the root, which is the representative.</p> <p>We find the representative of both sets, and set the parent of one to the other.</p> <p>Worst case is $\Theta(m^2)$ if we create a super-long m height tree and perform FindSet on the bottom node.</p>

- **Idea #5:** Like #4, but nodes also store field size (initialized to 1), the tree's size like #3

Idea #5 Implementation	Comments
<pre>def union(D, x, y): n1 = find_set(D, x) n2 = find_set(D, y) max = n1 if n1.size >= n2.size else n2 min = n2 if n1.size >= n2.size else n1 max.size += min.size min.parent = max</pre>	<p>Similar to #3. We append the smaller tree to the larger tree, and only update size in the representative.</p> <p>The worst case is also the same as #3, $\Theta(\log m)$, where we combine trees with sizes equal to powers of 2.</p>

- **Idea #6:** Like #4, but nodes also store field rank, an upper bound on height. Also, “path compression”

Idea #6 Implementation	Comments
<pre>def find_set(D, x): n = D.nodes[D[x]] if n is None: return None elif n != n.parent: n.parent = find_set(n.parent.item) return n.parent def union(D, x, y): n1 = find_set(D, x) n2 = find_set(D, y) max = n1 if n1.rank >= n2.rank else n2 min = n2 if n1.rank >= n2.rank else n1 max.rank += 1 if max.rank == min.rank else 0 min.parent = max</pre>	<p>Recursive algorithm that first finds the node. If the node is its own parent, return the node. If the node is not its own parent, set its parent to the recursive call on the node's parent.</p> <p>This “flattens” the tree since every traversed node's parent becomes the root node.</p> <p>Similar to #5, but rank is only updated when the two trees are of equal rank. This is because normally, FindSet will flatten the trees out completely.</p> <p>The worst case is very complex (don't need to know), $\Theta(\log_2^* m)$, where $\log_2^* n = \begin{cases} 0 & n \leq 1 \\ 1 + \log_2^*(\log_2 n) & n > 1 \end{cases}$ “# of times $\log_2 n$ must be applied to get $n \leq 1$”</p>

Kruskal's Algorithm, Minimum Spanning Tree, Version #2	Comments
<pre> def kruskal(G): T = Tree() D = DisjointSets() E = sorted(list(G.E)) # assume this sorts by weight for v in G.V: make_set(D, v) for e in E: if find_set(e[0]) != find_set(e[1]): union(D, e[0], e[1]) T.V.update({e[0], e[1]}) T.E.add(e) return T </pre>	<p>Running-time of $\Theta(m \log m + n)$</p> <ul style="list-style-type: none"> • Sorting is $\Theta(m \log m)$ • First for loop is n iterations • Second for loop is m iterations • Body of for loop is $\mathcal{O}(\log^* m)$ iterations at best, which is $\mathcal{O}(\log m)!$

Lower Bounds

Algorithm Complexity: Analysis of running-time of one algorithm/data structure to solve a task

Problem Complexity: Analysis of running-time all possible algorithms/data structures to solve a task

- In practice, arguments are based in computational models

Comparison Trees: Models of comparison-based algorithms, where nodes are comparisons to make, edges are possible outcomes connecting to subsequent nodes/comparisons to make.

- Leaves are all possible outcomes
- **Informational Theory:** Worst case running-time is $\Omega(h)$, where h is the tree's height

<p>eg. Searching values in a sorted list (returning the value's index, -1 if not inside).</p> <p>Pick any algorithm and input size n, which will result in a comparison tree.</p> <p>Tree has $1 +$ leaves for all $n + 1$ possible outcomes (n outcomes for every index plus -1 for failure)</p> <p>Then # of leaves $\geq n + 1$</p> <p>If comparisons go three ways, then tree is ternary, so</p> $ \begin{aligned} h &\geq \log_3 n \\ &\geq \log_3 (\# \text{ of leaves}) \\ &\geq \log_3 (n + 1) \in \Omega(\log n) \end{aligned} $ <p>If comparisons go two ways, then tree is binary, so</p> $ \begin{aligned} h &\geq \log_2 n \\ &\geq \log_2 (\# \text{ of leaves}) \\ &\geq \log_2 (n + 1) \in \Omega(\log n) \end{aligned} $ <p>Recall worst-case running-time is $\Omega(h)$ time, so in other words $\Omega(\log n)$</p>	<p>eg. List sorting – rephrase the problem as returning a permutation of $[0, \dots, n - 1]$ where the i-th element is the index of $A[i]$ in the sorted version of an input list.</p> <p>There are $n!$ possible outcomes</p> <p>Then # of leaves $\geq n!$</p> <p>If comparisons go k ways, then</p> $ \begin{aligned} h &\geq \log_k n \\ &\geq \log_k (\# \text{ of leaves}) \\ &\geq \log_k (n!) \\ &= \log n + \log(n - 1) + \dots + \log 1 \\ &\geq n \log n \in \Omega(n \log n) \end{aligned} $
--	---

eg. Searching values in an unsorted list (returning the value's index, -1 if not inside).

Comparisons are two-way (testing equality)

$n + 1$ outcomes means the informational theory says $\log(n + 1)$ is a lower bound. But this isn't a tight bound...

Adversarial Argument: Every correct algorithm has worst-case problem of $\geq T(n)$

- Usually prove the contrapositive: If worst case is $< T(n)$, algorithm is not correct

eg. Searching values in an unsorted list (returning the value's index, -1 if not inside).

Count comparison performed on an input of size n

Assume we have an algorithm that performs $\leq n - 1$ comparisons

Show there exists an input which algorithm returns an incorrect output

- Assume each comparison the algorithm uses is of form "if $x = A[i]$ "
- Construct an input where $x \neq A[i]$ always holds – every checked index does not contain x
- Only $n - 1$ comparisons were made, so ≥ 1 index is not checked
- Eventually, we reach a leaf in the comparison tree with output k .
 - If $k = -1$, build the input so the unchecked index contains x , making the algo wrong!
 - If $k \neq -1$, build the input so the unchecked index doesn't contain x , making the algo wrong!

eg. Find the max of a list

Assume an algorithm performs $\leq n - 2$ comparisons

Assume every comparison is of form "if $A[i] > A[j]$ "

There are ≥ 2 elements not checked with each other ("undetermined")

Then when comparison tree reaches a leaf, let i be the chosen max.

For some j , i vs. j is undetermined. Build the input so $A[j] > A[i]$.
