CSC263: Data Structures and Analysis - Notes

Harsh Jaluka

Feb 2022

Contents

1 Abstract data types

An **abstract data type (ADT)** is a set of objects together with operations.

Examples:

- Integers: The integers (objects) with operations add, compare and multiply
- Stacks: A pile of things (objects) with operations push, pop and isEmpty

A **data structure** is an implementation of an ADT (a way to represent objects and an algorithm for every operation)

Stack implementations:

1. Linked lists: a linked list and a pointer to the head of the list
2. Arrays: an array with an attribute for size

The key point is that the **abstract data type** describes the 'what' (the objects and the operations) and the **data structure** describes the 'how' (the implementation).

2 Analysing runtime complexity

2.1 Definition: complexity

Complexity is the amount of resources required by the algorithm as a function of the input size.

Resources usually refer to the running time (or memory/space).

Input size is problem dependent.

- lists: the length of the list
- graphs: the number of nodes and edges
- numbers: the number of bits needed to represent the number (or the number itself)

Sometimes we count the exact number of operations but oftentimes we use **asymptotic notation**: 'Big O', 'Big Omega', 'Big Theta'.

We will take runtime cases: best case, average case and worst case.

2.2 Definitions: asymptotic notation

$$\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 \in \mathbb{R}^+ \text{ such that } 0 \leq f(n) \leq cg(n) \ \ \forall n \geq n_0\}$$
$$\Omega(g(n)) = \{f(n) : \exists c, n_0 \in \mathbb{R}^+ \text{ such that } 0 \leq cg(n) \leq f(n) \ \ \forall n \geq n_0\}$$
$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 \in \mathbb{R}^+ \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \ \forall n \geq n_0\}$$

Examples:

- $f(n) = 2n^2 + n \in \mathcal{O}(n^2), \mathcal{O}(n!), \mathcal{O}(n^n), \mathcal{O}(n^{1000})$
- $f(n) = 2n^2 + 2 \in \Omega(n^2), \Omega(1), \Omega(n)$
- $f(n) = 2n^2 + 2 \in \Theta(n^2)$

We say that Θ gives a 'tight bound'

Intuitively,

- $f(n) = \mathcal{O}(g(n))$ means the function $f(n)$ grows slower or at the same rate as $g(n)$

- $f(n) = \Omega(g(n))$ means the function $f(n)$ grows faster or at the same rate as $g(n)$
- $f(n) = \Theta(g(n))$ means that the function $f(n)$ grows at the same rate as $g(n)$

Note: the usage of equality is an abuse of notation; set inclusion '$\in$' is more correct, mathematically speaking.

2.3 Worst-case analysis

The question we are seeking to answer with worst-case analysis is what is the maximum possible running time of an algorithm for an input of size n?

If the following two conditions are met,

- Big-Omega (lower bound): exhibit one (family of inputs for each n) input on which the algorithm executes at least $cg(n)$ steps.
- Big-O (upper bound): argue that the algorithm executes no more than $cg(n)$ steps on any input of size n.

then we can say that the worst case running time is $\Theta(g(n))$.

Example:

```
    def hasTwentyOne(L):
1         j = L.head
2         while j != None and j.key != 21:
3             j = j.next
4         return
```

Some of the ways to count the number of operations is the number of comparisons, the total number of lines executed or the number of times line 3 is executed.

Most of the times, the way to count the number of operations will be given in the question. Sometimes, the question will delegate this responsibility to us.

For example, in this case, we will choose to count the number of comparisons and the input size will be the length of the list n.

One family of bad inputs is lists of length n with no occurrences of 21. Line 2 will be executed n times, which means $2n$ comparisons will be executed and then 1 final comparison will be executed when `j == None`. There will be a total of $2n + 1$ comparisons.

Now, we need to show that any input of length n will take no more than $2n+1$ comparisons. Each time line 2 is executed, either we finish early or execute line 3. Each time line 3 is executed, we move to the next element in the list. Hence, we can execute line 3 at most n times, after which `j` has to be `None`. So, we can conclude that we can execute line 2 at most n times when followed by line 3 and then a final 1 time after which line 4 will be executed and the program will terminate. Hence, we can have at most $2n+1$ comparisons.

Having shown an upper bound and a family of inputs that give the upper bound, we can conclude that the worst-case number of comparisons is $2n + 1$.

What about the best-case?

The best case is **not** the empty list. It is crucial to remember that we cannot make any choices about the value of n; we want to find the best case for an arbitrary n. It can be verified that in this case, it would be when the first element of the given list is 21.

2.4 Average-case analysis

In practice, it is possible that the worst case rarely shows up. To get the expected running time for an arbitrary input, we perform **average-case analysis**.

The average-case analysis depends on the probability of each input.

Example (continued):

Let us do the average-case analysis for the example `hasTwentyOne`.

The number of comparisons depends on when we see the 21 for the **first** time. If the first 21 is at location k, then there will be $2k$ comparisons. If there is no 21, then there will be $2n + 1$ comparisons.

We want to divide our possible inputs into classes/sets in which each element execute the same number of operations. One possible way to do this is

- x_0: all inputs with no 21 in the list
- x_1: all inputs where 21 is at the first element
- x_2: all inputs where 21 is at the second position

  $\vdots$
- x_n: all inputs where 21 is at the nth position

For an algorithm A, we denote the sample space of all inputs of size n with S_n.

Let $t_n(x)$ be the number of steps executed by A on an input x in S_n

Finally, the expected running time of all possible values

$$\begin{aligned} E[t] &= \sum_t t \cdot P[t] \\ &= \sum_{k=0}^{n} t_n(x_k) P(x_k) \\ &= (2n+1)P(x_0) + \sum_{k=1}^{n} 2kP(x_k) \end{aligned}$$

Let's assume each position in the list can be 21 with probability p. Then, the probability that any position is not 21 is $(1-p)$.

$$\begin{aligned} E[t] &= (2n+1)(1-p)^n + \sum_{k=1}^{n} 2k(1-p)^{k-1}p \\ &= (2n+1)(1-p)^n + \frac{2p}{1-p}\sum_{k=1}^{n} k(1-p)^k \\ &= (1-p)^n\left[1 - \frac{2}{p}\right] + \frac{2}{p} \qquad \text{(use arithmetic-geometric series)} \end{aligned}$$

As $n \to \infty$, $(1-p)^n \to 0$ and $E[t] \to \frac{2}{p} \in \Theta(1)$ since p is a constant.

Handy summation formulas:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \qquad \text{(arithmetic series)}$$

$$\sum_{k=0}^{n} a^k = \frac{a^{n+1}-1}{a-1} \qquad \text{(geometric series)}$$

$$\sum_{k=1}^{n} ka^k = \frac{a^{n+1}(n(a-1)-1)+a}{(1-a)^2} \qquad \text{(arithmetic-geometric series)}$$

General procedure for doing average-case analysis:

1. Define the possible set of inputs and a probability distribution over this set.
2. Define the 'basic operations' being counted. This is often the operation that happens the most frequently or the most expensive operation.
3. Define the random variable T over this probability space to represent the running time of the algorithm.
4. Compute the expected value of T, using the chosen probability distribution and formula for T.
5. Since the average number of steps is counted as an exact expression, it can be written as a Θ expression.

3 Heaps

3.1 Motivation: priority queues

Consider a scenario where the following ADT would be required.

Data: A collection of items which each have a priority

Operations: Insert(x, priority), FindMax, ExtractMax

Some conceivable implementations, with the corresponding worst-case complexities for each operation, are given in the following table

	Insert	FindMax	ExtractMax
Unsorted linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n) + \Theta(1)$
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n) + \Theta(1)$
Sorted array (asc.)	$\Theta(n)$	$\Theta(1)$	$\Theta(1) + \Theta(1)$
Ordered linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1) + \Theta(1)$
Binary search tree	$\Theta(n)$	$\Theta(n)$	$\Theta(n) + \Theta(1)$

While binary trees have, by far, has the worst worst-case complexity of all the proposed implementations, we note that this is mainly because we are allowing these trees to be extremely *unbalanced*. Indeed, our worst case for a binary search tree is just (a more limited version of) an unsorted linked list. However, we also know that if we could somehow keep the tree balanced, then the maximum depth of the tree would be $\log_2 n$ and we would get $\Theta(\log n)$ runtime for all three operations.

The question is, how can we ensure that the tree remain balanced? This brings us to a new data structure called *heaps*.

3.2 Definitions

Definition: nearly complete binary tree

A nearly complete binary tree is a binary tree where every row is completely filled except possibly the lowest row. Moreover, the lowest row is completely filled from the left (i.e. there cannot be 'gaps').

Definition: heap

A heap is a nearly complete binary tree that is stored in an array (we will discuss this later in the chapter). Moreover, a heap must satisfy a property, which is, appropriately, called

the 'heap property'.

Intuitively, the heap property dictates a relationship between parents and children and maintains sortedness in an extremely loose sense.

Definition: heap property

There are two kinds of heap properties:

- A 'max heap' satisfies the heap property that the value at every node is greater than or equal to the value of its immediate children.
- A 'min heap' satisfies the property that the value at every node is less than or equal to the value of its immediate children.

Recall that our main goal is to come up with an efficient implementation of priority queues. So, we have to discuss how the three operations Insert, FindMax and ExtractMax would be implemented.

3.3 Insert

When a new item is inserted, it is placed at the lowest row in the leftmost empty node. One can imagine that this would be problematic as the heap property may be violated. This is true, but we will ensure that the new tree has the *structure* of a heap before making sure the heap property is held.

To maintain the heap property, we compare the new node to its parent. Assume we are working with a max heap; the argument is analogous for a min heap. If it is less than or equal to its parent, then the heap property is preserved and we are done. If not, then we swap the parent with the child and check the heap property again with the new parent's parent. This process is referred to as **'bubble up'**. The running time is $\Theta(\log n)$ since there can be at most h swaps where $h = \log_2 n$ is the height of the tree.

3.4 FindMax

The max (for max heaps) is always at the root.

Moreover, the running time of FindMax is $\Theta(1)$.

3.5 ExtractMax

The following is the algorithm for extracting the max from the heap.

1. Remove and return root element
2. Restore shape first and then fix heap property. To do this, put the rightmost element in the lowest row at the root
3. 'Bubble down' or 'max-heapify' to fix heap property:

   Swap the root with the larger of the two children. Compare the again with its new children and if the root is smaller than its children, swap with the larger of them. Repeat until heap property is fixed.

In the worst case, we may have to make as many swaps as the height of the tree. So, the (worst-case) running time is $\Theta(\log n)$.

Notice that there was a detail that we swept under the rug here. In step 2, we want to replace the root with the rightmost element in the lowest row. Pictorially, one can easily read off the the desired element. However, in code, this is not possible. So, how do we actually reach the rightmost element in the lowest row? The answer lays within the ways the heaps are stored.

3.6 Storage

We use an array to store heaps. By convention, we use 1-based indexing so the root is at element 1. The array is the elements in the tree top to bottom, left to right. For a node at position i, the left child is at $2i$, the right child is at $2i + 1$ and the parent is at $\lfloor \frac{i}{2} \rfloor$.

We also keep track of an attribute called `heapsize` which, as one would expect, keeps track of the number of elements in the heap. To answer the question that we ended the previous section with, we can simply access the element at index `heapsize` to get the rightmost element in the lowest row.

This ends our discussion of using heaps as an implementation for priority queues. It turns out, however, that heaps also provide for a really efficient sorting algorithm. The next section explores this.

3.7 Heapsort

We will first describe the algorithm to turn a heap into a sorted list. Recall that the root of the heap stores the maximum element.

The key idea is to remove the root, put it in an array at the end, decrement the heapsize and restore the heap property. This is almost identical to the algorithm for ExtractMax. Indeed, here are we just calling ExtractMax repeatedly and storing the returned elements until we have the sorted list.

The storing is done 'in place' since the replacement item for the root was in the position where we wanted to put the root anyway. In other words, no extra space is needed to sort the array.

The pseudocode for `maxHeapify` is given by

```
def maxHeapify(L, i):
    l = left(i)         # l is the index of the left child
    r = right(i)        # r is the index of the right child
    if l <= L.heapsize and L[l] > L[i]:
        largest = l
    else:
        largest = i
    if r <= L.heapsize and L[r] > L[largest]:
        largest = r
    if largest != i:    # max heap property violated
        exchange L[i] with L[largest]
        maxHeapify(L, largest)
```

What is the complexity of maxHeapify?

Each call to ExtractMax takes $\Theta(\log n)$ steps. Since we are calling ExtractMax n times, Heapsort takes $\mathcal{O}(n \log n)$ steps. However, this is not sufficient to show that it is $\Theta(n \log n)$ since the n decreases over time as we remove elements from the heap. It can be proved that this algorithm is $\Theta(n \log n)$ but we will postpone this discussion for later.

We've discussed how to go from a heap to a sorted list. However, typically, we want a procedure to sort unsorted lists. So, we need an efficient procedure to turn unsorted lists into heaps.

3.8 Building a heap

First approach:

Start with an empty heap and repeatedly call Insert. If there are n items in the list, then the last insert takes $\log(n)$ time. There are n inserts and all inserts are less than $\log(n)$ times. Hence, the build time is $\mathcal{O}(n \log n)$.

Can we get a tight bound? We need to provide a family of inputs. Let us add the elements in increasing order. Notice that at least $\lceil \frac{n}{2} \rceil$ nodes are leaves and each of those take at least $\log n - 1$ time. Hence, we have $\Omega(\frac{n}{2} \log(n-1)) = \Omega(n \log n)$. Finally, we can conclude $\Theta(n \log n)$

Second approach:

We make calls to maxHeapify for every element that is not a leaf, bottom up. We skip the leaves because each leaf is a 1-element heap to begin with. Each call takes $\mathcal{O}(\log n)$, so we immediately get a bound of $\mathcal{O}(n \log n)$.

To get a tight bound, notice that the running time of maxHeapify(L,i) is proportional to the height of the tree rooted at i.

- $\frac{n}{2}$ leaves at height 0 require 0 swaps
- $\frac{n}{4}$ nodes at height 1 require 1 swap
- $\frac{n}{8}$ nodes at height 2 require 2 swaps

$$\vdots$$

- 2 nodes at height $\log n - 1$ require $\log n - 1$ swaps
- 1 node at height $\log n$ requires $\log n$ swaps

A heap contains at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes at height h (this is a generalisation from the few cases examined above). Hence, the total time

$$\begin{aligned}
\sum_{h=1}^{\lfloor \log n \rfloor} h \cdot \lceil \frac{n}{2^{h+1}} \rceil &= \mathcal{O}(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}) \\
&= \mathcal{O}(n \sum_{h=1}^{\infty} \frac{h}{2^h}) \\
&= \mathcal{O}(n \frac{1/2}{(1-1/2)^2})
\end{aligned}$$

$$= \mathcal{O}(2n)$$
$$= \mathcal{O}(n)$$

where we use Gabriel's staircase series:

$$\sum_{k=1}^{\infty} kr^k = \frac{r}{(1-r)^2} \qquad (0 < r < 1)$$

This bound of $\mathcal{O}(n)$ is better than the first approach. The reason behind this is that in the first approach, the most expensive operations - the ones where the leaves of the final tree are added and we perform bubble-up - were also the most common ones. In the second approach however, the most expensive operation is calling max-heapify on the root, which only occurs once. As we go down the depth of the tree and max-heapify is called more frequently, it becomes less expensive since there are less elements left to bubble down to.

4 Dictionaries

Operations:

1. Insert(S, x)
2. Search(S, k)
3. Delete(S, x)

where S is the dictionary, x is a key-value pair and k is the key.

Implementations (with worst-case complexities):

Data structure	Search	Insert	Delete
Unsorted array	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
Sorted array	$\Theta(\log n)$	$+\Theta(n)$	$+\Theta(n)$
Unsorted linked list	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
Sorted linked list	$\Theta(n)$	$\Theta(1)$	$+\Theta(1)$
Direct-access table*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hash table (best case)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hash table (worst case)	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
Binary search tree	$\Theta(n)$	$+\Theta(1)$	$+\Theta(n)$
Balanced binary search tree	$\Theta(\log n)$	$+\Theta(\log n)$	$+\Theta(\log n)$

Times for Insert and Delete are in addition to the time to first Search for the element.

* The catch is that direct-access tables take up a huge amount of space. Everything else we have considered takes $\mathcal{O}(n)$ space but direct access tables need one location per *possible* key. If keys are 32-bit integers, we would need $\Omega(2^{32})$ space.

Hash table: map all possible keys onto a smaller set of actual keys from which you do direct access. The problem is that collisions will occur if there are not enough slots for keys. The worst case for hash tables is if all the keys go to the same slot. This essentially becomes an unsorted linked list. We will return to hashing later but for now, it's very efficient because the average case turns out to be $\Theta(1)$.

Balanced binary search tree: After insertion or deletion, we will need to correct the balance. There are a number of algorithms for this; red-black trees, AVL trees and 234 trees.

5 Binary search trees

The binary search tree property is that the left child is less than or equal to the node is less or equal to than the right child.

Since there are no duplicate keys in regular dictionaries, we can drop the 'equal to' requirement.

The height is defined as the number of edges to the lowest leaf or the lowest depth. To get the max height, every time an element is inserted it must be the min/max of the remaining elements.

When a node is deleted, we must replace it with the successor (chosen for convenience, in theory it could be predecessor as well). The successor is the minimum value in the right subtree.

Note that a successor can never have two children because by definition it cannot have any element smaller than it (and hence there cannot exist a left child). If its a leaf, then no change has to be done. If it has one child, the child is promoted when the successor is replaced at the root.

The worst case is $\Theta(height)$. In the worst tree, the height is n. So, the worst case is $\Theta(n)$. Even if we had already searched, we could still need $\Theta(n)$ to find the successor.

The major drawback right now is that the structure of the tree is completely dependent on the order of insertion. When this leads to unbalanced trees, we get a larger running time than necessary.

5.1 Duplicate keys

What happens if we allow duplicate keys in our dictionary?

For each strategy, we will change the pseudo code and analyse the total time taken to insert n identical items into an empty tree.

Approach 1: always go right

Each insert takes kC time where k is the depth $+1$ where the new element is added and C is the complexity of Insert. Hence, the total time is $C\frac{n(n+1)}{2} \in \Theta(n^2)$.

Approach 2: strictly alternate using a flag to keep track

Worst case time to insert into a tree is $\Theta(\log m)$ for a tree that currently has m values. So,

the total time is $c \log 1 + c \log 2 + \cdots + c \log n$.

$$= c \sum_{i=1}^{n} \log i < cn \log n \in \mathcal{O}(n \log n)$$

To show a tight bound, consider

$$c \sum_{i=1}^{n} \log(i) > c \sum_{i=n/2}^{n} \log(n/2) = c \frac{n}{2} \log(\frac{n}{2}) \in \Omega(n \log n)$$

Therefore, we have $\Theta(n \log n)$

Approach 3: randomly pick a side

On average, we would end up with the same tree but in the worst case, the tree could get quite bad (as in approach 1).

Approach 4: Keep a chain of values inside the node

6 AVL Trees

In a regular binary search tree, the shape is determined by the insertion order and the relative values.

Definition (height of a tree): The height of a tree rooted at v is the number of edges on the longest path from v to a leaf. The height of an empty tree (*ad hoc*) is defined to be -1.

Definition (balance factor): The balance factor (BF) of a node m is given by

$$BF(\text{node } m) = h_R - h_L$$

where h_R = height of m's right subtree and h_L = height of m's left subtree.

Definition (AVL balanced): If $BF(m) = 0$, we say that m is balanced. If $BF(m) = 1$, we say that m is AVL balanced (right-heavy). If $BF(m) = -1$, we say that m is AVL balanced (left-heavy). For all other cases, we say that the tree is not AVL balanced.

6.1 Search

The height of an AVL tree h has an upper bound $h \leq 1.44 \log_2(n+2)$ (see proof in notes for AVL trees).

Search in an AVL tree is just BST search since the BST property is satisfied. However, since the complexity of search for BSTs is $\Theta(height)$, which in this case would be $\Theta(\log(n))$.

6.2 Insert

Properties of a single rotation:

1. Insertion only affects the balance factors of its ancestors (justification in the supplementary pdf on Quercus)
2. The root BF depends on $h(A), h(C), h(E)$
3. The overall height of the rotated tree remains the same (as before insertion), so that nothing beyond it is affected in terms of height or balance factors.

Sometimes, a single rotation does not work. This is when the longer subtree is in the 'middle'. In these situations, we first do a rotation so that the longer piece is outside and then we do the normal rotation in whichever direction necessary.

The complexity of Insert is $\Theta(\log n)$ ($\Theta(\log n)$ contribution from finding the place at which it needs to be inserted and checking whether we need rotation, $\Theta(1)$ contribution from rotation, if necessary).

6.3 Delete

- All cases end up being equivalent to deleting a leaf

A problem aries when the tree is already left-heavy and we delete the bottom most node in the right subtree (analogously for right-heavy trees).

After deletion, consider the first node which goes out of AVL balance and denote it the root. If the balance factor at the top most node on the left subtree is 0 or -1, then we can do a single right rotation on the root to restore AVL balance. In this case, the resulting tree after deletion *might* have a smaller height by 1.

If the balance factor is 1, then we have to do a left rotation on the topmost node of the left subtree followed by a right rotation on the root. In this case, the resulting tree will have a smaller height by 1.

If the height decreases, then the tree may go out of AVL balance. In particular, this depends on the balance factor of the parent of the root. So, we may have to do $\log n$ rotations all the way up to the actual root of the tree, which makes for $\mathcal{O}(\log n)$ worst case running time (rotations take constant time).

The deletion algorithm is as follows:

```
find node p to delete

on route from p back to root at each node i:

if BF(i) was previously 0:
    update BF and exit
if BF(i) was previously +1 or -1:
    if it becomes 0 after deletion, then we shortened tree rooted at i
    so continue up path
if BF(i) was previously +1 or -1 and change makes it +2 or -2:
    do appropriate rotation
    if rotation shortened tree rooted at i, then continue up tree
    if not, stop
```

7 Hashing

Definitions:

- universe of keys U: the set of all possible keys
- hast table T: an array with m positions where each location is often called a slot or a bucket
- hash function $h : U \to \{0, 1, \cdots, m-1\}$: maps keys to an array position; to access key k (or its data), we will examine $T(h(k))$
- collision: when $x \neq y$ but $h(x) = h(y)$, we say there is a collision

Since $|U| > m$, collisions are unavoidable (pigeonhole principle). So, we must have a strategy to deal with them and the first one that we will look at is closed addressing (also known as chaining).

7.1 Closed addressing

With this strategy, we assign to each bucket a linked list of items that hash to it.

Worst-case analyses for insert, search and delete:

The worst-case for insert is when all n keys have been hashed to the same bucket and the new key we are inserting is not in the dictionary. To insert this key, we apply the hash function, which is constant time (assuming arithmetic operations take constant time) and go to the respective bucket. Now, we have to search through the entire list before adding the key because keys have to be unique in dictionaries. So, we will make n comparisons and then only can we conclude that this key does not exist, and hence we can add it to the end of the linked list. This takes $\Theta(n)$ time.

The worst-case for search is when all n keys are hashed to the same bucket. Then, there is a linked list of length n and to find the key we are looking for, we must check every single element of the linked list since there is no predefined order on it. Hence, the worst-case time complexity is $\Theta(n)$.

The worst-case for delete is the same as the one for search. First, finding the element takes $\Theta(n)$. However, since we are working with linked lists, deleting it is only a matter of updating the link of the previous element, which is constant time. Hence, if we have already found the element, then deletion takes $\Theta(1)$ but the overall process takes $\Theta(n)$.

Note that we are only guaranteed that a bucket with all n keys in the hash table is possible when $|U| > m(n-1)$. To see why, let us add all keys from U to our hash table. Assume there is no bucket with n keys. Then there are at most $n(m-1)$ keys in the table. However, we know that $|U| > m(n-1)$, which gives us a contradiction.

Another thing to note is that $|U| > m(n-1)$ is a sufficient condition, but not a necessary one. It is conceivable to construct hash functions that map n keys to the same bucket and for whom such a large universe would not be necessary.

7.1.1 Average-case runtime of search

The simple uniform hashing assumption is the assumption that a key is equally likely to hash to any bucket. The consequence is the expected number of keys in each bucket is the same $\frac{n}{m}$. This value is given a special name: the load factor α.

Since we are doing an average case analysis, we need to have a probability distribution over all the inputs. In this set of notes, we will assume that we are equally likely to search for any key in U.

For the running time analysis, let the time to compute the hash function and the number of keys we need to compare be the operations that we count. We will divide our analysis into two parts; first, we will look at the keys are in the table and then we will look at those that are not.

Part 1: assume that the key k is not in the table

Compute $h(k)$ and then traverse the entire list in $T(h(k))$. Since we assumed that the expected number of keys in each bucket is $\frac{n}{m}$, the number of comparisons will be $\frac{n}{m}$ and we will have the expected running time $E[T] = 1 + \frac{n}{m} = \Theta(1 + \frac{n}{m})$. (Note: we don't drop the 1 because we don't have information about α).

Part 2: assume that the key k is in the table

If $X_1, X_2, \cdots, X_n$ are the values we inserted into the table, then the probability that we selected one of the X_i's is $\frac{1}{n}$. Having chosen X_i, the time taken to search for this key is 1 for computing the hash function and some number of comparisons in the linked list. To find this number, we must first note that $n - i$ items were inserted into T after X_i and hence, $\frac{n-i}{m}$ of these items are expected in each bucket. Since we always add newer elements to the front of the linked list, we only have to make $\frac{n-i}{m} + 1$ comparisons. Finally,

$$E[T] = 1 + \frac{1}{n}\sum_{i=1}^{n}(\frac{n-i}{m} + 1)$$

$$
\begin{aligned}
&= 2 + \frac{n}{m} - \frac{1}{mn}\sum_{i=1}^{n} i \\
&= 2 + \frac{n}{m} - \frac{n+1}{2m} \\
&= 2 + \frac{n-1}{2m} \\
&= 2 + \frac{\alpha}{2} - \frac{\alpha}{2n} \\
&= \Theta(1+\alpha)
\end{aligned}
$$

Since, in both cases, we have the same average running time, we can conclude that search is $\Theta(1+\alpha)$ overall. This implies that if m is large enough, we have $\Theta(1)$ search performance.

7.2 Open addressing

Instead of storing a reference to a linked list at every bucket, the open addressing strategy looks for another bucket if the bucket that is hashed to is already taken. Hence, all the keys are kept in the table.

Instead of the hash function depending only on k, as with closed addressing, the hash function now depends on k and an integer i. This i will run from 0 to $m-1$ and is introduced to obtain a sequence of m buckets as i varies. The reason behind this is that we will traverse the sequence generated by key k and put k in the first empty bucket in the sequence. Naturally, then, we would like this sequence to be a permutation of $\{0, \cdots, m-1\}$ because we want to ensure that every key is mapped to an empty bucket. We refer to the bucket given by $h(k,0) = h'(k)$ as the 'home bucket' and we refer to the bucket that the key actually ends up in, appropriately enough, as the 'actual bucket'.

An example of such a hash function is $h(k,i) = (h'(k) + i) \mod m$ where $h'(k) = k \mod m$ is a hash function for closed addressing. This sort of approach is called linear probing because it finds the home bucket and if it is full, then successively checks the next bucket until it finds an empty one. However, there are issues with linear probing. One of the biggest ones is that it leads to clustering. If it so happens that a number of keys are hashed to consecutive buckets and form a cluster, then any key that maps to anywhere in the cluster will continue to enlarge the cluster. This leads to worse performances as the cluster increases.

What about if we used a larger step size? With this approach, while the table would not *look* clustered to us, the problem would still be there because there would be a chain of buckets to visit and if some sequence of them happens to be full, clustering occurs again.

We are then inclined to think that perhaps the linearity in the search is the problem. What

about if we used a random step size? This may better the performance of insert, but it is a nightmare for search because the key is equally likely to be anywhere in the table if not in its home bucket. So, randomness is not an approach.

Naturally, then, we take a nonlinear approach and as we will see, this indeed does solve the problem. Take $h(k,i) = (h'(k) + c_1 i + c_2 i) \mod m$. With this, as i increases, the offset from the home bucket increases as well, as desired. A minor problem that remains with this approach is that any two keys that map to the same bucket have the same step sizes. This can easily be fixed by making the step size dependent on k. An example is $h(k,i) = (h_1(k) + h_2(k)i) \mod m$. This is called **double hashing**.

Note that we still want the hash function to output a sequence that is a permutation of the buckets and this needs to be ensured by requiring $h_2(k)$ and m to be relatively prime. One common technique is to make m a power of 2 and $h_2(k)$ odd.

7.2.1 Deletion

When we delete an element from the hash table, we cannot simply set the bucket to be null because this may cause problems for future search and delete queries. Here is an illustration of why.

Assume we delete the key at bucket i and set the bucket to null. Then, any search query for a key that passes through bucket i will stop there because, a priori, one would (correctly) think that if the key were in the table, it would be at bucket i (Since we pick the first empty bucket found).

The solution to this is to mark a bucket with a tombstone whenever we delete its key. This would signify to search and delete that they should not stop probing at that bucket. Values can still be inserted in that bucket and the tombstone can be removed when a value is inserted.

7.3 Hash functions

In a good hash function, we want

- $h(x)$ to be efficient to compute
- $h(x)$ to spread out the values (not cluster them)
- $h(x)$ to depend on every part of the key (even for complex objects)

In practice, it is difficult to achieve all three. So, we make trade-offs. For example, if we want $h(x)$ to depend on every character of a very long string, then we would have to sacrifice efficiency.

Steps to make a hash function

1. From string or key to natural number k

   e.g. ASCII code with each position of string as a digit in base 128. For instance, `key` maps to $107(128)^2 + 101(128) + 121(128)^0$. However, we cannot use this natural number as our hash bucket because that would create very large hash buckets.

2. From natural number k to the range $\{0, \cdots, m-1\}$

   - Division method: $h(k) = k \mod m$
     - usually avoid an m that is a power of 2 since that is equivalent to taking lower-order bits of k. So, we choose a prime near the m that we want.
   - Multiplication method: $0 \leq \lfloor m(kA - \lfloor kA \rfloor) \rfloor < m$
     - multiply k by A where $0 < A < 1$
     - keep only the fractional part
     - multiply by m
     - take the floor

     What makes a good A? There are a lot of guidelines, see textbook for more information.

7.4 Hashing vs balanced trees

If we make the table an appropriate size, we control α. Hashing is $\Theta(1)$ average case. So, why would we ever use a tree for a dictionary?

1. If we cannot tolerate the worst case. Worst case of a balanced tree is $\Theta(\log n)$ as opposed to $\Theta(n)$ for hash tables.

2. If we need a list of all the keys, we would have to search through all the buckets for a hash table.

8 Augmented data structures

Definition: An **augmented data structure** is an existing data structure that has been modified to store additional information and/or perform additional operations.

1. Choose data structure to augment
2. Determine the additional information
3. Check that additional information can be maintained during each original operation
4. Implement new operations

8.1 Ordered sets

Apart from INSERT, DELETE and SEARCH, we define two new operations:

- RANK(k): return the 'rank' of key k

  The rank is a way to define position in an ordered list. It is typically defined from lowest to highest.

- SELECT(r): return the key with rank r

Approach 1: AVL tree without modification

We will carry out an in-order traversal of the tree and keep track of the number of nodes visited until we reach either the desired key or the desired rank. The worst-case for this approach is $\Theta(n)$. All the other operations remain the same because the tree remains the same. They all remain $\Theta(\log n)$. So, the problem with this approach is the RANK and SELECT are too slow.

Approach 2: AVL tree with additional field rank[x]

Although RANK and SELECT become $\Theta(\log n)$, and SEARCH remains the same, both INSERT and DELETE's worst-case running times becomes $\Theta(n)$ because all ranks may have to be updated when, for example, an element that is smaller than all the elements in the tree is added or when the smallest element is removed.

Approach 3: AVL tree with an additional field n.size() for each node that stores the number of keys in the subtree rooted at n

The rank of the root of the tree is the size of the left subtree +1. For any root, we define its local (relative) rank to be the size of the left subtree +1.

SELECT has $\Theta(\log n)$

For RANK, the local rank of a node m in the tree rooted at m is given but its left tree's size +1. When we recurse on the left subtree, we add 0. When we recruse on the right subtree, we add the parent's local rank (to some variable, say current rank). When we have reached the node we are looking for, its true rank is given by the current rank + its local rank. This also has $\Theta(n)$

SEARCH remains the same $\Theta(\log n)$.

For INSERT, as we go down the tree to insert, simply add one to each node we travel through. Also $\Theta(\log n)$.

For DELETE, remove one from path from root to leaf. Also $\Theta(\log n)$.

However, since this is an AVL tree, we have to consider rotations. This requires changing the tree's sizes when doing rotations. This is easy, still in constant time.

Take-away:

1. Augmenting an AVL tree with size of subtree rooted at m, allowed us to efficiently implement RANK and SELECT.
2. Use known data-structure with extra information
3. Extra information must be maintained on original operations

9 Quicksort

9.1 The algorithm

```
1    def quickSort(array):
2        if len(array) < 2:
3            return array{:}     # makes a copy
4        else:
5            pivot = array[0]
6            smaller, bigger = partition(array[1:], pivot)
7            smaller = quickSort(smaller)
8            bigger = quickSort(bigger)
9            return smaller + [pivot] + bigger
10
11    def partition(array, pivot):
12        smaller = []
13        bigger = []
14        for item in array:
15            if item <= pivot:
16                smaller.append(item)
17            else: bigger.append(item)
18        return smaller, bigger
```

9.2 Running-time analysis

Upper bound: We will count the number of comparisons in line 15.

Note that each element of S is pivot at most once and at most all other elements are compared to the pivot. So, every pair is compared at most once and there are $\binom{n}{2}$ pairs if $|S| = n$. Hence,

$$T(n) \leq \binom{n}{2} \in \mathcal{O}(n^2)$$

Lower bound: We will use the family of inputs of `[n, n-1, n-2, ···, 2, 1]`.

Let $C(n)$ denote the number of comparisons on any such input of length n.

In line 6, the pivot n gets compared in the helper function `partition` to all the other $n-1$ values and yields `smaller = [n-1, ···, 1]` and `bigger = []`. All other comparisons happen in recursive call `quickSort(smaller)`, which takes $C(n-1)$ comparisons, by

definition. Hence, $C(n)$ satisfies the recurrence $C(n) = n - 1 + C(n-1)$ for $n > 1$ and $C(1) = 0$. Hence,

$$C(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n-1)(n)}{2} \in \Omega(n^2)$$

Since we have shown an upper bound of n^2 and a family of inputs with this bounnd, we can conclude that the worst-case running time is $\Theta(n^2)$.

If it is an $\Theta(n^2)$ worst-case, why is it called 'quick'? This is because the average case running time is quite good.

Average-case analysis:

Let us assume that the inputs are all permutations of `[1,2,3, ⋯, n]` and a uniform distribution. (We've chosen that there are no duplicates and that we are equally likely to get any random input.)

Let T be a random variable that counts the number of comparisons. We want to find $E[T]$.

Define an indicator random variable X_{ij} where $i, j \in \{1, 2, \cdots, n\}$ and $i < j$

$$X_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

Hence,

$$T = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

$$E[T] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ji}]$$

$E[X_{ij}]$ is the probability that i and j are compared. For i and j to be compared, either one has to be the pivot. For either one to be the pivot, we cannot pick any value between i and j to be a pivot (in this case, i will go into `smaller` and j into `bigger` and will not be compared with each other).

In any iteration, we are equally likely to pick any one of $\{i, i+1, \cdots, j-1, j\}$. Hence, the probability that i and j will be compared is $\frac{2}{j-i+1}$. So,

$$E[T] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Let $k = j - i$

$$\begin{aligned} E[T] &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\ &= 2(n-1) \sum_{k=1}^{n} \frac{1}{k} \\ &= 2(n-1) \log(n) \in \mathcal{O}(n \log n) \end{aligned}$$

Let us do it again to show $\Theta(n \log n)$

$$\begin{aligned} E[T] &= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \\ &= 2 \sum_{k=1}^{n-1} \frac{(n-k)}{k+1} \\ &= 2 \sum_{k=1}^{n-1} \frac{n+1-(k+1)}{k+1} \\ &= 2(n+1) \sum_{k=1}^{n-1} \frac{1}{k+1} - 2(n-1) \\ &= 2(n+1)(\log(n-1)-1)) - 2(n-1) \in \Theta(n \log n) \end{aligned}$$

9.3 Quicksort summary and randomised quicksort

- Quicksort performs well on random input.
- Quicksort performs poorly on nearly sorted inputs.

Can we guarantee the 'average' performance on 'average'?

Yes, if we randomise the choice of pivot.

Randomised Quicksort:

- Average-case analysis depended on each permutation equally likely

- Instead of relying on distribution of inputs, randomise algorithm by picking random element as pivot
- Random behaviour of algorithm on any fixed input is equivalent to fixed behaviour of algorithm on uniformly random input

Expected worst-case time of randomised algorithm on every single input is $\Theta(n \log n)$ (i.e. if we ran quicksort on any particular input multiple times and average the time, it would be $\Theta(n \log n)$.

Take away: Randomised algorithm are good when there are lots of good choices but hard to find a choice that is guaranteed to be good.

10 Amortized Analysis

Often, we perform a sequence of operations on data structures and are interested in the time complexity of the entire sequence. In an amortized analysis, we find the average performance of each operation using the worst case time for a sequence of operations. This is different from average-case analysis in that there is no probability involved; amortized analysis guarantees the average performance of each operation in the worst case.

Definition: worst-case sequence complexity (WCSC) (on m operations)

The worst-case sequence complexity on m operations is the maximum time over all sequences of m operations.

Remark: The WSCS is less than m multiplied by the maximum worst-case time complexity over all operations in the sequence of m operations.

Definition: amortized sequence complexity (ASC)

The amortized sequence complexity is the worst-case sequence complexity of m operations divided by m.

This value oftentimes makes more sense (than average-case analysis) in real situations.

There are three approaches to calculating the amortized complexity:

1. Aggregate: add together the costs from m operations and divide by m

   $\rightarrow$ works well when there is only one operation (simply because it is the same operation does not imply that the cost of each operation is the same)

   $\rightarrow$ also can work when there are multiple operations

2. Accounting: charge each operation a 'fee', pay the real cost and then bank the overcharge on a specific piece of the data structure

3. Potential: charge every operation. but now the overcharges are stored as 'potential energy' in the data structure overall

We will cover the aggregate method and the accounting method in more detail. See CLRS for more about the potential method.

10.1 Aggregate method

10.1.1 Binary counters

A binary counter is a sequence of k bits (with k fixed) with a single operation: INCREMENT

```
INCREMENT(A)
1   i <- 0
2   while i < length[A] and A[i] = 1
3       do A[i] <- 0
4       i <- i + 1
5   if i < length[A]
6       then A[i] <- 1
```

Note that the zeroth digit flips every time, the first digit every 2 times, the second digit every 4 times and in general, the kth digit every 2^k times. Over m operations, the zeroth digit changes m times, the first digit $\lfloor m/2 \rfloor$ times, the second $\lfloor m/4 \rfloor$ times and the kth digit $\lfloor m/2^{k-1} \rfloor$ times. The sum of all bits is then

$$\sum_{i=0}^{k-1} \lfloor \frac{m}{2^i} \rfloor < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

Hence, the total running time for m operations is $\mathcal{O}(2m) = \mathcal{O}(m)$. Hence, the amortized cost per operation is $\frac{\mathcal{O}(m)}{m} = \mathcal{O}(1)$

10.1.2 Multipop stacks

A multipop stack is a stack with the regualar PUSH and POP operations, each of which take $\mathcal{O}(1)$ time. In addition, we have a MULTIPOP operation, which takes $\mathcal{O}(\min(|S|, k))$

```
MULTIPOP(S, k)
1    while not STACK-EMPTY(S) and k != 0
2       do POP(S)
3       k <- k - 1
```

In n operations, at most there are n calls to PUSH, which costs a total of n. If $|S|$ is at most n, then in total we can have at most n calls to POP (either by calling POP or MULTLIPOP). So, the total cost has to be less than or equal to $2n$, which is $\mathcal{O}(n)$. Finally, it follows that the amortized cost of a single operation is $\mathcal{O}(1)$

10.2 Accounting method

With the accounting method, we charge each operation an amortized cost (aka a fee or a charge). The amortized cost can be more or less than the actual cost and it can be different for different types of operations.

For an operation i, let C_i be the actual cost and $\hat{C}_i$ be the amortized cost. When $\hat{C}_i > C_i$, the credit left over is assigned to an associated element in the data structure. When $\hat{C}_i < C_i$, the difference has to be paid for by the existing credit. In other words, the data structure can never be in debt.

$$\sum_{i=1}^{k} \hat{C}_i \geq \sum_{i=1}^{k} C_i \qquad (\forall k)$$

10.2.1 Multipop stacks

Recall that the multipop stack had operations PUSH, POP, MULTIPOP. They cost $1, 1$ and $\min(|S|, k)$ respectively. Let us set the amortized cost to be 2 for PUSH and 0 for both POP and MULTIPOP.

Claim: (credit invariant) Each item in the stack has \$1 credit

Proof: We use induction.

Base case: In an empty stack, there are no items and hence the statement is vacuously true.

Induction step: Assume the credit invariant is true is for an existing stack. We will show that the credit invariant holds after performing any of PUSH, POP or MULTIPOP.

PUSH: We charge \$2 and the actual cost was \$1, so we are left with \$1 on the newly pushed element, as desired.

POP: We charge \$0 but pay for the POP with \$1 credit on that element (which we know exists due to the induction hypothesis). Since nothing has changed to the remaining elements, the credit invariant holds.

MULTIPOP: We charge \$0 but it costs however many items were popped off. For each item, we charge the \$1 credit on that item and the remaining items' credits are unchange, showing that the credit invariant holds.

□

Since the credit invariant says that the credit is greater than or equal to 0, we are never

in debt. It follows that each operation's amortized cost is less than or equal to 2, which implies $\mathcal{O}(1)$ per operation.

10.2.2 Binary counters

Recall the binary counter from the aggregate method:

A binary counter is a sequence of k bits (with k fixed) with a single operation: INCREMENT

```
INCREMENT(A)
1   i <- 0
2   while i < length[A] and A[i] = 1
3       do A[i] <- 0
4       i <- i + 1
5   if i < length[A]
6       then A[i] <- 1
```

Note that the zeroth digit flips every time, the first digit every 2 times, the second digit every 4 times and in general, the kth digit every 2^k times.

We will determine the average performance of INCREMENT using the accounting method.

We will charge \$2 for each call to INCREMENT. The actual cost if \$1 per bit that flips in each call.

Claim: (credit invariant) At any step in the sequence, each bit in the counter that is equal to 1 has a \$1 credit.

Proof: (by induction)

Base case: The counter is 0 and hence has no bits equal to 1. The statement is vacuously true.

Induction step: Assume the credit invariant is true at some value of the counter. We will show that it remains true after calling INCREMENT.

Note that the cost of flipping any 1 to 0 is paid for by the \$1 credit on each 1. Note also that only one of the 0's flips to 1 and it is paid for by \$1 out of \$2 from the fee paid by calling INCREMENT. No other bits change.

Then, it follows that the invariant still holds.

□

This shows that the data structure never goes in debt. Finally, the amortized cost per operation is then less than or equal to 2, which implies $\mathcal{O}(1)$ average performance of INCREMENT.

10.3 Dynamic arrays

So far in CSC263, we have always assumed that the arrays we use have a size that is sufficient for whatever we wish to do. This gave us $\mathcal{O}(1)$ access to read or write any individual array element. In real life, however, we sometimes do not know beforehand the size of the array we will require. While we could use a linked data structure (a linked list or a binary search tree, for instance), we lose the constant time access to an element with these structures.

Instead, high-level programming languages often use **dynamic arrays**. Python's lists and Java's ArrayLists are examples of dynamic arrays. Dynamic arrays are structures that behave like arrays but grow or shrink in size as needed.

Before we can understand how dynamic arrays work, it must be noted that we can access elements in constant time for (normal) arrays because the array is stored in one contiguous block of memory. So, if we want constant access time in dynamic arrays, we would have to make sure that if the array size is increased to store more elements, the new elements are stored immediately after the original array.

This is problematic, because there may not be empty memory space remaining directly after the array. The solution to this is that we simply find a new location in memory that has sufficient space for the new larger sized array and we copy the old array and put it in the new location with the new elements.

Here is the pseudocode for one implementation of insertion into a dynamic array.

```
def insert(A, x):
""" Instance attributes:
    - A.size = number of total elements in the array
    - A.allocated = total number of elements for which memory space is
      currently allocated
    - A.array = array of size A.allocated that holds the elements
      themselves
"""
    # Check whether current array is full
    if A.size == A.allocated:
        new_array = new array of length (A.allocated * 2)
        copy all elements of A.array into new_array
```

```
        A.array = new_array
        A.allocated = A.allocated * 2

    # insert the new element into the first empty slot
    A.array[A.size] = x
    A.size = A.size + 1
```

To the user of this ADT, it appears as if A is the array, but it is actually a wrapper around the array A.array. Let us try to find the average performance of `insert` over m operations using the aggregate method.

$$WCSC = m + 2 \sum_{i=0}^{\lfloor \log_2(m-1) \rfloor} 2^i$$

Now, note that

$$\sum_{i=0}^{b} 2^i = 2^{b+1} - 1$$

So, we now have

$$\begin{aligned} WCSC &= m + 2 \cdot (2^{\lfloor \log_2(m-1) \rfloor + 1} - 1) \\ &\leq m + 2(2 \cdot 2^{\log(m-1)} - 1) \\ &= 5m - 6 \\ &= \mathcal{O}(m) \end{aligned}$$

This implies that the amortized complexity of a single operation is $\mathcal{O}(1)$.

Let us now try to find the amortized complexity of `insert` using the accounting method.

We will charge \$5 per call to insert and store the credit on the element added.

Claim: (credit invariant) When the array is full and we need to expand, each element in the 2nd half of the full list has a \$4 credit.

Proof: (by induction)

Base case: The first time that the array is full is when it A.allocated = A.size = 1. In this case, since there is no 'second half' of the list when the length of the list is 1, the statement is vacuously true.

Induction step: Assume that the array is full and each element in the 2nd half of the full list has a \$4 credit. When we add a new element, a new array of twice the length is created

and all the original elements are copied to it. Note that the second half of the original array now have \$0 each because reading and writing each element costs \$2 and we had to do this for the entire array. Now, as we add new elements and the longer array starts to fill up, each element, after being added, will have a credit of \$4 because the amortized cost is \$5 and it takes \$1 to insert an element when there is space in the array. Finally, when the last element in the new array is filled, the entire second half will have \$4 credit, as desired.

□

End of midterm 2 content!

11 Graphs

11.1 Definitions

Definition: graphs, undirected, directed

A **graph** is a tuple of two sets $G = (V, E)$, where V is the set of vertices, and E is the set of edges on those vertices. An **undirected** graph is a graph where order does not matter in the edge tuples: (v_1, v_2) is equal to (v_2, v_1). Because the order doesn't matter, we sometimes see the edge in an undirected graph expressed as a set $\{v_1, v_2\}$. A **directed** graph is a graph where the tuple order does matter: (v_1, v_2) and (v_2, v_1) represent different edges in a directed graph.

Definition: self-loops

A self-loop (in a directed graph) is an edge going from a vertex to itself. Self-loops are not allowed in undirected graphs.

Definition: adjacent

If (u, v) is an edge in an undirected graph, then vertex v is said to be adjacent to vertex u, and this relationship is symmetric: vertex u is also adjacent to vertex v. In a directed graph, v is adjacent to u if there is an edge (u, v).

Definition: incident

In an undirected graph, we say that an edge (u,v) is incident on vertices u and v. In a directed graph, the terminology differentiates between the beginning and ending vertex of an edge. So edge (u,v) which leaves vertex u is said to be incident from vertex u and is incident to (or enters) vertex v.

Definition: degree

In an undirected graph, the degree of a vertex v is the number of edges incident on v. In a directed graph, the in-degree of vertex v is the number of edges incident to v and the out-degree is the number of edges incident from v.

Definition: neighbourhood

A neighbourhood of a particular vertex v, is the set of vertices that share an edge with v. To be more precise, in an undirected graph, the neighbourhood of vertex v is the set of vertices u such that (u, v) is an edge. In a directed graph, the in-neighbourhood of vertex v is the set of vertices u such that (u, v) is an edge, and the out-neighbourhood of vertex v is the set of vertices u such that (v, u) is an edge.

Definition: weighted graphs

In a weighted graph, we associate an additional real number with each edge and call this value the edge weight or the edge cost. Weighted graphs can be directed or undirected.

Definition: paths, simple paths

A path is a sequence of edges connected to each other: (v,u1), (u1,u2), ..., (uk-1,x) and its length is the number of edges on the path. A path is called simple if it has no repeated edge or vertex.

Definition: cycle, acyclic

A cycle is a path with the same starting and ending vertex. In an undirected graph, a path forming a cycle must have no repeated edges (and must therefore contain at least 3 vertices). In a directed graph, a cycle must have at least 1 edge. A self-loop is a cycle of length 1. A simple cycle has no repeated edge or vertex (other than the same first and last vertex, of course). A graph is acyclic if it contains no cycle.

Definition: connected

An undirected graph is connected if it contains a path between any two vertices.

Definition: trees

A tree is an undirected connected graph that is acyclic. This definition may be different from the trees you have encountered so far in your CS experience, which have all been rooted trees. In a rooted tree, one vertex is designated as the root. Sometimes the term free tree is used for an undirected connected acyclic graph without a specific vertex designated as the root.

Definition: forests

Finally, a forest is a collection of (zero or more) disjoint trees.

11.2 Representation

How do we represent graphs?

- Adjacency list representation: make a list of the vertices and for each vertex, give sublist of adjacent vertices. A common implementation is a fixed array of vertices and a linked list for each vertex.
- Adjacency matrix representation: make a matrix with the columns and rows labelled by the vertices in the graph. Put a 1 in entry (a, b) if vertex b is adjacent to vertex

a and 0 if not. When the graph is undirected, the matrix is symmetric along the diagonal with 0s on the diagonal.

11.3 Complexities

Space complexities:

- Adjacency list: $\Theta(|V| + |E|)$

  - either V or E can dominate depending on the graph

- Adjacency matrix: $\Theta(|V|^2)$

So, an adjacency list can save space in a sparser graph.

Time complexities:

- Add/remove vertex:

  - Adjacency list:

  - Adjacency matrix:

- Add/remove edge:

  - Adjacency list:

  - Adjacency matrix:

- Edge query: given vertices u, v, check if (u, v) in E

  - Adjacency list: $\mathcal{O}(|V|)$

  - Adjacency matrix: $\Theta(1)$

- Neighbourhood query: given a vertex v, return the (in/out) neighbourhood as a set of vertices

  - Adjacency list: it is much more efficient to find the out-neighbourhood of a vertex using an adjacency list.

  - Adjacency matrix:

11.4 Breadth First Search (BFS)

Starting from a source vertex s in V, we visit every vertex v that is reachable from s. In the process, we find paths from s to each reachable v. The paths make a BFS tree that is rooted at s. This works on both directed and undirected graphs.

Progress is kept track of by colouring each vertex

1. White: not yet discovered (all vertices start as white)
2. Grey: discovered but not fully explored
3. Black: fully explored

Breadth first search keeps track of the parent of v in the BFS tree (also known as the predecessor) and the distance (or depth) from s to v.

The grey vertices are stored in `Queue` and are handled in FIFO order.

Convention: explore in order of the vertices in the given adjacency list.

11.5 The BFS algorithm

```
BFS(G=(V,E,s):
1    for all v in V:
2       colour[v] <- white
3       d[v] <- inf
4       π[v] NIL
5   colour[s] <- grey
6   d[s] <- 0
7   initialize empty queue Q
8   ENQUEUE(Q,s)
# loop invariant Q contains exactly the grey vertices
9   while Q not empty:
10      u <- DEQUEUE(Q)
11      for each edge (u,v) in E:
12          if colour[v] == white:
13              colour[v] <- grey
14              d[v] <- d[u] + 1
15              π[v] <- u
16              ENQUEUE(Q,v)
17      colour[u] <- black
```

11.6 Depth First Search (DFS)

The basic idea is to search through the graph going as deep as possible before we backtrack to explore the edges of already discovered vertices. Similar to BFS, each vertex is coloured

1. White: not yet discovered
2. Grey: discovered but not fully explored
3. Black: fully explored

Note that fully explored means something different for DFS (as opposed to BFS)

Instead of storing distance from source vertex, we store timestamps

1. d[v]: discovery time (time at which first discovered, white $\rightarrow$ grey)
2. f[v]: finish time (time at which fully explored, grey $\rightarrow$ black)

It is natural to write DFS recursively. Since DFS is commonly used to find connected components, the main algorithm does not take a source vertex but rather calls a helper function DFS-VISIT(G, s) repeatedly on each s in V in adjacency list order (just convention).

11.7 The DFS algorithm

```
DFS(G=(V,E)):
1   for each v in V:
2       colour[v] <- white
3       f[v] <- d[v] <- ∞
4       π[v] <- NIL
5   time <- 0       # global
6   for each v in V:
7       if colour[v] == white:
8           DFS-VISIT(G,v)

DFS-VISIT(G=(V,E),u):
1   colour[u] <- grey
2   d[u] <- time <- time + 1
3   for each (u,v) in E:
4       if colour[v] == white:
5           π[v] <- u
```

```
6            DFS-VISIT(G,v)
7    colour[u] <- black
8    f[u] <- time <- time + 1
```

DFS-VISIT is only called on white vertices and then those vertices are immediately painted grey. So, DFS-VISIT is called once on each vertex.

Outisde of recursive calls, each execution of DFS-VISIT examines the adjacency list for one vertex. So, the total running time is $\mathcal{O}(|V| + |E|)$.

11.8 Topological sort

1. For directed acyclic graphs only
2. A linear ordering of all vertices in graph G such that if G contains (u, v) then u comes before v in the ordering
3. Algorithm: run DFS on G, collect finish times and as each vertex finishes, insert it into the front of a list

Recall that an undirected graph is connected if there is always a path between any two nodes. What about a directed graph? Since a directed graph could be 'visually connected' in the sense that there is always a line between two nodes, all nodes may not be reachable from each other. So, we do not define connectedness for directed graphs and instead say that a directed graph is **strongly connected** if for any pair of vertices u and v, there is a path from u to v and v to u.

Now, given a directed graph, what algorithm can we use to test whether it is strongly connected? Pick a vertex s in the directed graph G and call DFS-VISIT(G, s). If $f[s] = 2|v|$ then we have a path from s to v for all v in V. Now, make a new graph G' whose edges are all the edges in G but reversed. Run DFS-VISIT(G', s) and if $f[s] = 2|v|$ then we have a path from v to s for all v in V.

12 Minimum spanning trees

Minimum spanning trees are only generated for connected and undirected graphs. In practice, they are usually weighted as well.

Two ideas for the algorithm

1. Take a graph and remove expensive edges until we get an MST

2. Start with nothing and add edges from the graph making a forest/tree as we go, until we have an MST

We will go with approach 2 since approach 1 is much harder to implement.

Greedy minimum spanning tree:

```
GREEDY-MST(G=(V,E), w:E→ ℝ)
1    T ← {} # invariant: T is a subset of some MST of G
2    while T is not a spanning tree:
3        find e ''safe for T''
4        T ← T ∪ {e}
```

An edge is "safe" if and only if $T \cup \{e\}$ is a subset of some MST of G. But this is a circular definition.

Theorem: a safe edge

If G is a connected, undirected, weighted graph, T is a subset of some MST of G, and e is an edge of minimum weight whose end points are in different connected components of T, then e is safe for T.

The proof is given in the text (CLRS).

12.1 Prim's algorithm

The idea is to pick a root vertex and grow the minimum spanning tree T by connecting an isolated vertex to T. At each step, pick the cheapest edge that connects a vertex from T to one not in T. In pseudocode, this is

```
1    for each v in V:
2        priority[v] ← ∞
3        π[v] ← NIL
4    priority[r] ← 0
5    Q ← V
6    while Q is not empty:
7        u ← ExtractMin(Q)
8        T ← T ∪ {π[u], u} # except when u = r
9        for each v in adj[u]
10           if v in Q and w(u,v) < priority[v]:
11               priority[v] ← w(u,v)
12               decrease priority(Q, v, new priority)
13               π[v] ← u
```

What is the complexity of Prim's algorithm?

The initial for-loop takes $\Theta(|V|)$ time. Recall that the complexity of building a heap is $\mathcal{O}(|V|)$. Now consider the while-loop. Instead of considering the complexity of each iteration, we instead realise that line 7 iterates exactly $|V|$ times and the for-loop within iterates exactly $2|E|$ times. Recall that ExtractMin costs $\mathcal{O}(\log|V|)$ and hence line 7 gives a total contribution of $\mathcal{O}(|V|\log|V|)$. Note that line 12 takes $\mathcal{O}(\log|V|)$ and that this depends on the implementation of the priority queue (in our case, a binary heap). Hence, the entire for-loop takes $\mathcal{O}(|E|\log|V|)$ time. We end up with $\mathcal{O}((|V|+|E|)\log|V|)$ overall which we simplify to $\mathcal{O}(|E|\log|V|)$ since $|E| \gg |V|$ for a connected graph.

12.2 Kruskal's algorithm

The idea is not to build a tree until the end but instead build a forest and merge trees in our forest until we have only one tree. At each step, we pick the cheapest edge in the whole graph (that is not yet in our forest) that does not make a cycle and add it to our forest. In pseudocode, this is

```
1    MST-KRUSKAL(G=(V,E), w:E→R)
2    sort edges so w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
3    for i ← 1 to m:
4        let (u_i, v_i) = (e_i)
5        if u_i, v_i in different connected components of T:
6            T ← T ∪ {e_i}
```

Line 2 takes $\mathcal{O}(|E|\log|E|)$ time. Before we can figure out the complexity of line 5, we must figure out an implementation.

We could do DFS/BFS, which would cost $\mathcal{O}(|V|)$ per iteration. Hence, the entire loop would take $\mathcal{O}(|E||V|)$ time. This is rather expensive. Fortunately, there is an easier implementation which makes use of a data type called disjoint sets. This is the focus of our next section.

13 Disjoint sets

Recall that two sets are disjoint if and only if their intersection is empty. This data type will have three operations:

- MAKE-SET(x): Given an element x that doesn't already belong to a set, create a new set containing only x and designate x as the representative.

- FIND-SET(x): Given an element x, find the representative of the set that contains x (or NIL if x is not in any set).
- UNION(x,y): Given 2 elements x and y, let S_x denote the set containing x and S_y the set containing y. Make a new set $S_x \cup S_y$ and designate a representative. Finally, remove S_x and S_y from the ADT.

We will propose 7 (that's right) different implementation for disjoint sets. First, however, we will work out `MST-KRUSKAL` with disjoint sets.

```
1    MST-KRUSKAL(G=(V,E), w:E→R)
2    sort edges so w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
3    for all v in V:
4       MAKE-SET(v)
5    for i ← 1 to m:
6       let (u_i, v_i) = (e_i)
7       if FIND-SET(u_i) != FIND-SET(v_i):
8           UNION(u_i, v_i)
9           T ← T ∪ {e_i}
```

- Disjoint set ADT manages sets only. The client code manages elements.
- Each element x manages a pointer to DJS data structure
- Complexity analyzed using worst-case sequence complexity, in the context of Kruskal's algorithm
- Each analysis based on m operations where n is the number of `MAKE-SET`s

1. Circular linked list
   - `MAKE-SET` takes $\mathcal{O}(1)$ time
   - `FIND-SET(x)` takes $\mathcal{O}(n)$ time
   - `UNION(x,y)` takes $\mathcal{O}(1)$ time
   - WCSC: m operations with n `MAKE-SET`s
     Consider the case with $n = \frac{m}{4}$ `MAKE-SET`s, $\frac{m}{4}$ `UNION`s and $\frac{m}{2}$ `FIND-SET`s (called on the 2nd element). Since each call to `FIND-SET` will take $\Omega(\frac{m}{4})$ time and since there are $\frac{m}{2}$ calls, the total time will be $\Omega(m^2)$.
     For an upper bound, note that since the number of elements at any points is $\leq m$, the complexity of each operation is $< m$ and hence the total time is $\mathcal{O}(m^2)$.
     So, we have $\Theta(m^2)$ for m operations.

2. Add back pointers (to make `FIND-SET` less expensive)

   - `MAKE-SET` and `FIND-SET` both take $\mathcal{O}(1)$ time.
   - `UNION` does not take $\mathcal{O}(1)$ anymore (all the back pointers in either list have to be updated)
   - WCSC: m operations with n `MAKE-SET`s

     Consider $n = \frac{m}{2} + 1$ `MAKE-SET`s, $\frac{m}{2} - 1$ `UNION`s. If we always append larger list to the shorter one, we get $\Theta(n^2)$.

   How do we make this even faster? One way is to always pick the smaller list to change all the back pointers in. This can be done by carrying a size attribute

3. Union by weight (weight is the number of elements in the set)

   - when `UNION(x,y)`, we pick the new representative from the larger of the two sets (hence less number of elements have to be changed)
   - WCSC: m operations with n `MAKE-SET`s

     Since we have n `MAKE-SET`s, we never have more than n elements in total. Consider an arbitrary element x. We want an upper bound on the number of times we could update x's back pointer.

     We claim that this upper bound is $\log_2(n)$. This is because for an arbitrary element, the first time its back pointer is updated, the resulting set it is in has at least 2 elements. Indeed, every time its back pointer is updated, the size of the resulting set is at least twice. However, we can at most double the set size $\log_2(n)$ times since we start with 1 element and there can be a maximum of n elements.

     Finally, for n total elements, `UNION` would take $\mathcal{O}(n \log n)$ total time and the entire sequence $\mathcal{O}(m + n \log n)$ time.

4. Trees

   Use an inverted tree where each element has a pointer to its parent and the root of the tree is the representative (by convention, its parent points to itself)

   - `MAKE-SET` and `UNION` (after having found the set) both take $\mathcal{O}(1)$
   - `FIND-SET` is $\mathcal{O}(h)$ where h is the height of the tree.
   - `UNION(x,y)` makes the representative of `x` the representative of the new set.
   - WCSC: m operations with n `MAKE-SET`s

     Consider $n = \frac{m}{4}$ `MAKE-SET`s and $\frac{m}{4} - 1$ `UNION(x,y)`s where `x` is a singleton (we want to end up with a long chain of $\frac{m}{4}$ nodes). Finally, we want $\frac{m}{2} + 1$ calls to `FIND-SET`s on a leaf of the tree. This gives a lower bound of $\Omega(m^2)$.

For the upper bound, note that each operation must cost less than m ($1 \leq n \leq m$). Hence, over m operations, the maximum amount of time taken can be $\mathcal{O}(m^2)$.

Finally, this gives us $\Theta(m^2)$ time.

5. Tree union by weight

   Select the "heavier" tree's representative as the new root. Augment the tree to store the weight (i.e. the number of elements in the tree).

   - `MAKE-SET` and `UNION` are both $\mathcal{O}(1)$ (once we have found the representatives)
   - `FIND-SET`
   - During any sequence of m operations, n of which are `MAKE-SET`, the max height of the tree is $\mathcal{O}(\log n)$. To see this, one needs to do induction of the height of the trees. Hence, each `FIND-SET` takes $\mathcal{O}(\log n)$ time and the total time for m operations will be $\mathcal{O}(m \log n)$.

6. Path compression

   During `FIND-SET(x)`, keep track of nodes visited on the path from `x` to the root (using a stack, queue or recursion).

   Once the root (i.e. representative) is found, update the parent pointers of each node encountered to point directly to the root. This will at most double the time for `FIND-SET` but speeds up future operations.

   - WCSC: m operations with n `MAKE-SET`s
     This takes $\Theta(m \log m)$ time. The proof is messy and is omitted.

7. Union by rank (combines path compression with union by weights)

   Now that we have path compression, weight isn't nearly as important as the height of the the tree. Instead of looking at the exact weight or height, we are going to maintain an upper bound on the height of the tree. This is called the rank.

   - The rank of a leaf is 0
   - The rank of an internal node is 1+ maximum rank of its children.
   - `MAKE-SET` sets rank to 0
   - `FIND-SET` does not change the ranks (even though actual heights may change due to path compression)
   - `UNION(x,y)` makes the node with the higher rank the new root. The rank will be unchanged. If `x` and `y` have equal rank, we choose either as the new root and its rank is increased by 1. By convention, `x` will be chosen as the new representative.

- WCSC: m operations with n `MAKE-SET`s
  The time taken is $\mathcal{O}(m \log^* n)$ where $\log^*(n) = \alpha(n)$ is the inverse of the Ackermann function. It is also known as the iterated log because it is the number of times the log function function must be iteratively applied before the result is ≤ 1.
  So, any DJS would have an amortized cost of $\Omega(\alpha(n))$ for `FIND-SET`

14 Lower bounds for sorting

- We know that the existence of algorithms that run in worst-case time $\mathcal{O}(n \log n)$ confirms that sorting can be done in time $\mathcal{O}(n \log n)$, but this does not rule out the existence of other algorithms that do better.
- We know how to analyze the worst case complexity of algorithms. The worst case complexity of problems involves extra work.
- For a given problem P, C(P) is the best worst-case running time of any algorithm that solves P.
  - Provide an upper bound on C(P) - give one algorithm and analyze its time
  - Provide a lower bound on C(P) - prove that every algorithm requires a certain amount of time.
- In practice, prove lower bounds on classes of algorithms.

14.1 Comparison trees

Represent algorithms that work by piece-wise comparison of elements.

14.2 Information theory lower bounds

- Every binary tree with height h has $\leq 2^h$ leaves. Every binary tree with L leaves has height $\geq \lceil \log_2 L \rceil$
- Every comparison tree that solves a problem P has at least one leaf for each possible input. Every comparison tree for P has height $\geq \lceil \log_2 m \rceil$ where m is the number of possible outputs
- The number of ways to sort a list is $n!$. Hence, every comparison tree has height $\geq \log_2(n!)$ and every algorithm that uses only comparisons requires at least $\log_2(n!)$ comparisons.

- Note that $\log_2(n!) = \Theta(n \log n)$

  $$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 + \log 1 \\ &\leq n \log n \\ &= \mathcal{O}(n \log n)\end{aligned}$$

  However, we also have that

  $$\begin{aligned}\log(n!) &\geq \log n + \log(n-1) + \cdots \log\left(\lceil \frac{n}{2} \rceil\right) \\ &\geq \frac{n}{2} \log\left(\frac{n}{2}\right) \\ &= \Omega(n \log n)\end{aligned}$$

  which shows $\log_2(n!) \in \Theta(n \log n)$

- Every algorithm that uses only comparisons between pairs of elements to sort requires $\Theta(n \log n)$.

What about other sorts?

- Radix sort or counting sort?
- Can be done in less than $\mathcal{O}(n \log n)$ comparisons because they are not based on pair-wise comparisons (these algorithms usually restrict the domain of the problems)
- Be careful when lower bounds apply only to algorithms of a particular type