ANTHOLOGY OF HORRORS

Things in this leaflet:

Things that will be added:

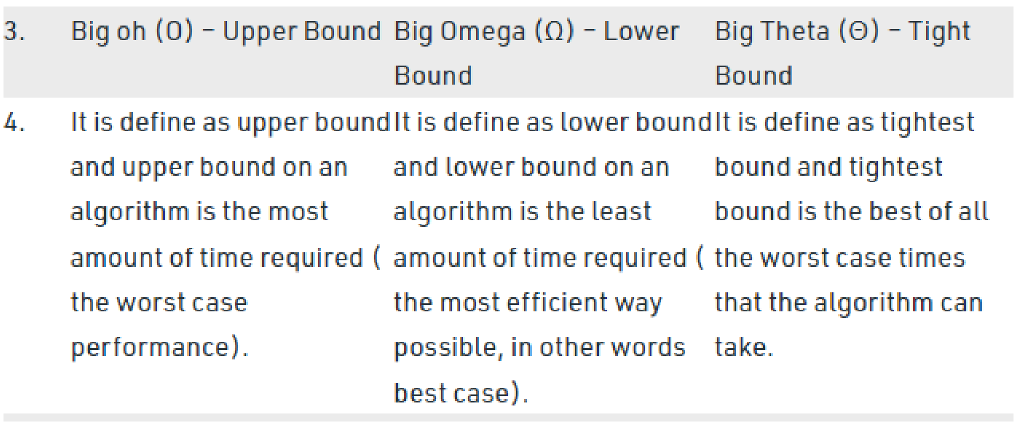
- A soul to replace the one I lost writing this.

If you have any questions about anything in this document contact me on discord! (**Balthazar#8695**)

Algorithm Analysis

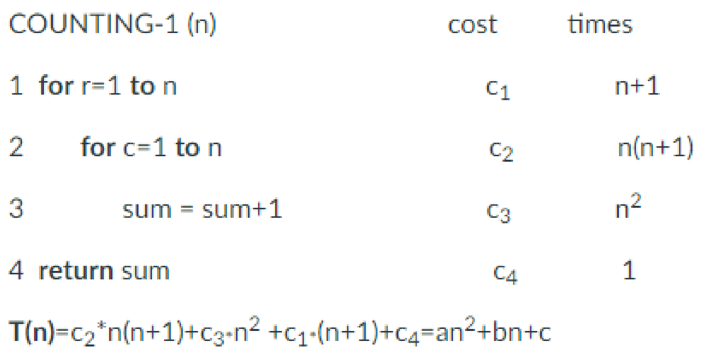
The basic topic we need to address here is time complexity. Now here are the important things to note.

There are 3 notations or ways of declaring running time of algorithms and functions



3.	Big oh (O) – Upper Bound	Big Omega (Ω) – Lower Bound	Big Theta (Θ) – Tight Bound
4.	It is define as upper bound and upper bound on an algorithm is the most amount of time required (the worst case performance).	It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).	It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.

The explanations provided should pretty much clarify them so we'll move on to the algorithm analysis.



```
COUNTING-1 (n)                  cost      times
1  for r=1 to n                 $c_1$     $n+1$
2      for c=1 to n             $c_2$     $n(n+1)$
3          sum = sum+1          $c_3$     $n^2$
4  return sum                   $c_4$     $1$
```

$$T(n)=c_2{*}n(n+1)+c_3\cdot n^2+c_1\cdot(n+1)+c_4=an^2+bn+c$$

Let's take a sample algorithm to have as a visual reference.

Here are our notes,
- the cost for each function is constant **ALWAYS!**
- What really matters is how many times it will be run!

So, if we have a **for loop**, that encompanses an entire dataset with lets say n = 10
If we start from the beginning it must run n or 10 times for index 1,2,3,4,5,6,7,8,9,10.

STOP > you see the cost in the picture is n + 1 right? Well that's because the line 1 and **ONLY** line 1, containing the for loop will run 11 times, why? Because on the 11th time it will start the iteration and realise it no longer satisfies the condition of less than or equal to n.

So what we understand here is that the **FOR LOOP** itself is run n + 1 times **BUT!** It's content is run n times, since on the last execution it won't run the code inside the loop.

A note here, the index we start the counter at changes the times run meaning if we started from 2 instead of 1 in the above picture, the times line 1 was run would be (n + 1) - 1 or n. Because we would be skipping 1 element from the n elements.

Now in line 2, there is another for loop, this for loop will again run itself n + 1 times, **BUT** it is inside another loop, and as we said whatever was inside the loop would be executed n times, so? The times run for the second loop would be n * (n+1).

We already clarified that the content of the for loop runs n times and not n+1, so the content of the second for loop will be run n * n times or n^2.

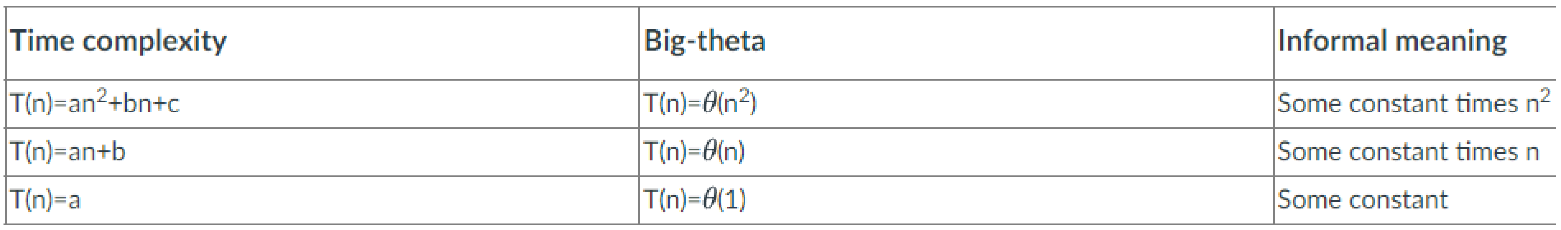
Now we don't really care about constant numbers in time complexity, we only care about the big stuff.

So if you were to look at the algorithm above, disregarding the formula below for T,
Which would be the biggest possible number?

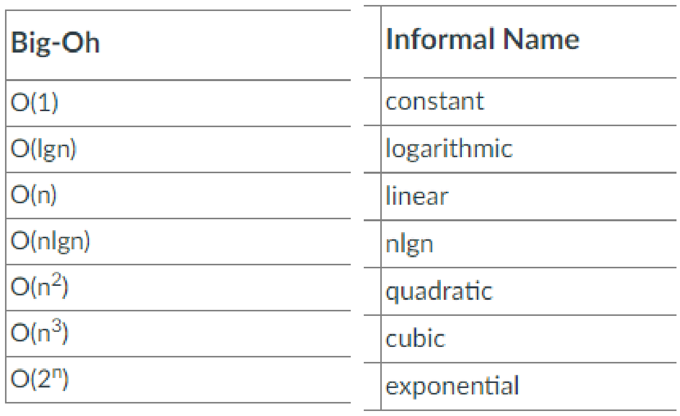
n^2, therefore, the algorithm has a time complexity of O(n^2) simple as that, now, you would probably be needed to show the T function, but i honestly don't know it so I can't explain it, what i'll do is check similar algorithms on the book and see which one fits the time complexity and just wing it. Sorry :P

What I can tell you is, no matter what you're looking at, all you have to do is find the biggest possible iteration and then remove all constants like a.

$$T(n)=c_2{}^*n(n+1)+c_3\cdot n^2+c_1\cdot(n+1)+c_4=\underset{\blacktriangle}{a}n^2+bn+c$$



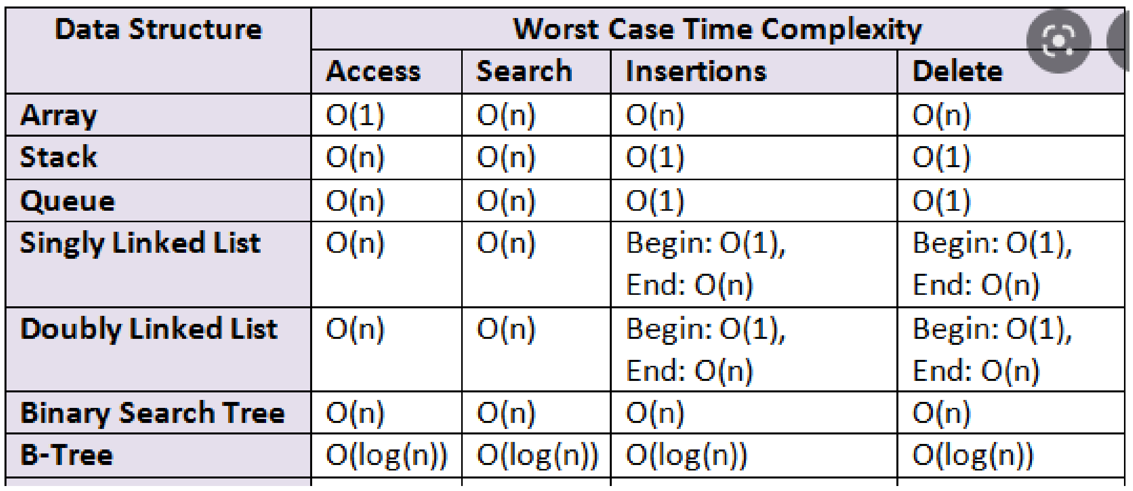
Time complexity	Big-theta	Informal meaning
$T(n)=an^2+bn+c$	$T(n)=\theta(n^2)$	Some constant times n^2
$T(n)=an+b$	$T(n)=\theta(n)$	Some constant times n
$T(n)=a$	$T(n)=\theta(1)$	Some constant



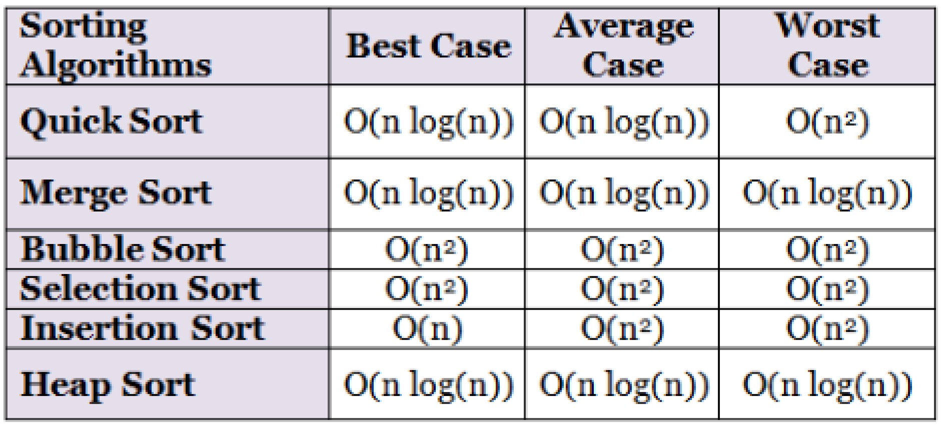
Big-Oh	Informal Name
O(1)	constant
O(lgn)	logarithmic
O(n)	linear
O(nlgn)	nlgn
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Now some tables for you to find running times quickly

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)



Data Structure	Worst Case Time Complexity			
	Access	Search	Insertions	Delete
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Singly Linked List	O(n)	O(n)	Begin: O(1), End: O(n)	Begin: O(1), End: O(n)
Doubly Linked List	O(n)	O(n)	Begin: O(1), End: O(n)	Begin: O(1), End: O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))



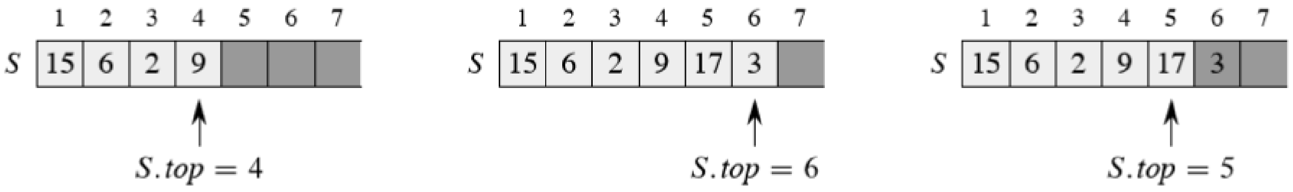
Sorting Algorithms	Best Case	Average Case	Worst Case
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n^2)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))

Stacks

Stacks are data structures that represent a set of objects just like arrays but more complex. The basic characteristic of stacks is that they follow a LIFO structure, the last element we added is the one to be removed. Elements get added to the end of the stack, which we will consider the top. So just like a PEZ dispenser we add more elements to the top and we remove elements from the top.

Dynamic data structures have a number of predefined operations that we will explore but for now you can see a list here assuming S is the stack and x is the element in question.

- SEARCH (S, k)
- INSERT (S, x)
- DELETE (S, x)
- MINIMUM (S)
- MAXIMUM (S)
- SUCCESSOR (S, x)
- PREDECESSOR (S, x)



```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE
```

As we said, stacks have an element defined as **top** and this element is always the last element we added.
NOTE! Top refers to the index not the value!

To check if it's empty it's pretty simple all we check for is if the top is 0 since the stack starts from 1 that would mean there's no elements in the stack.

```
PUSH(S, x)
1  $S.top = S.top + 1$
2  $S[S.top] = x$
```

To add a new element in the stack, it's again pretty easy, we just move the **top** one value to the right by increasing it by 1, and then we add the x value to the new top.

```
POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else $S.top = S.top - 1$
4      return $S[S.top + 1]$
```

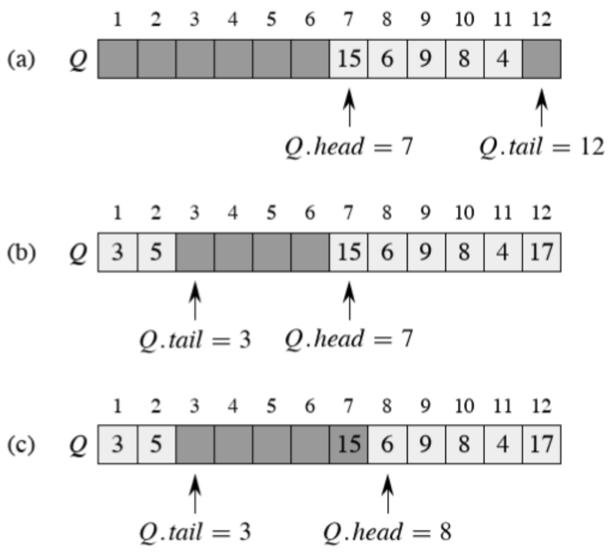
Now to remove an element from the stack we first check if it's empty (error underflow if it is) and then all we do is decrement top INDEX!! By 1.

Once again important to remember top refers to the index, **AND** we don't actually remove the element, we simply say that top is one index before as you can see in the last image above.

If we were to PUSH a new element after we popped, it would simply overwrite the previously POPed element :)

Queues

Queues are like stacks with some major differences though, they work in an opposite way than stacks as in FIFO, the first element that we added will be removed.
I will refer to them as Qs from now on for ease.
So in Qs we have 2 different values to keep track of the rabbit hole :P, we have the **head** and the **tail.**
The head is where Qs begin, and also signifies the element to be removed if need be, The tail is where the Qs end and where the next element will be added.
Note that Qs are “circular” meaning that the tail can loop around as seen in this picture.



(a) Q: 1–12; 15 6 9 8 4; Q.head = 7; Q.tail = 12
(b) Q: 3 5 … 15 6 9 8 4 17; Q.tail = 3; Q.head = 7
(c) Q: 3 5 … 15 6 9 8 4 17; Q.tail = 3; Q.head = 8

```
ENQUEUE(Q, x)
1  $Q[Q.tail] = x$
2  if $Q.tail == Q.length$
3      $Q.tail = 1$
4  else $Q.tail = Q.tail + 1$
```

```
DEQUEUE(Q)
1  $x = Q[Q.head]$
2  if $Q.head == Q.length$
3      $Q.head = 1$
4  else $Q.head = Q.head + 1$
5  return $x$
```

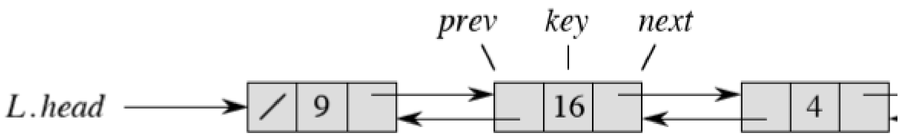
So lets talk about these operations.
We got enQ, which is similar to Push from stacks, and it basically adds a new element to the Q.
We start by using the tail index, now as you can see above, the tail index is always in the of the Q + 1, and it signifies the position where we would insert an element/

Now, this mambo jumbo here in line 2 is actually pretty simple, all it’s doing is checking if the element we just added made the tail reach the end of the line, in other words, look at image a, once we added an element in index 12 where the tail is, we would need to loop the tail around back to 1. And that’s what we do, we check if the tail is at the end, and then woop, move it back to 1. Else, if we haven’t reached the end of the line, all we have to do is move the tail one over.

Now deQ, so we said we follow LIFO so the head (1st element added) will be removed, the rest of the function is identical to enQ but with head instead of tail.

Linked Lists

So a linked list is a “list” where the elements are linked together. It can be doubly or singly linked. Meaning it can be A > B or A <> B. In a doubly linked list we can see both previous and next element, whereas in a singly linked list we can only see the next element.



Now to start off, we will look at doubly linked lists. As we can see there are multiple values we need to keep in mind.

Key - the value of the element we are looking at.
Prev - the previous element from Key
Next - the next element will from Key
Head - signifies the start of the list
NIL - signifies no element,

NOTE! Both start and tail are NIL meaning if x.prev is NIL, we’re at the start, if x.next is NIL, we’re at the end.

NOTE! If the list is circular, then head and tail are omitted and replaced by the last element as L.prev for the 1st element, and the first element as L.next for the last element.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

Now let’s talk about algos, so for searching a linked list, we start by passing the list and the element k to be found.
We assign x initially to be before the 1st element. Then we start a while loop to check for 2 things: 1st that we haven’t reached the end of the list, and 2nd that we haven’t found our element.

An important note here is that x signifies the block itself, and therefore we can call the 3 values, x.key, x.next & x.prev on x.

As the while loop runs it means it hasn’t found the element, and so it will keep looking at the next element each iteration, until finally returning to us x.
Keep in mind that it won’t return the value, it will return the “location” of the element in the list, or x
If it doesn’t find the element it will reach the NIL element and return that.

```
LIST-INSERT(L, x)
1  $x.next = L.head$
2  if $L.head \neq$ NIL
3      $L.head.prev = x$
4  $L.head = x$
5  $x.prev =$ NIL
```

Inserting an element into the list, new elements will be added at the beginning of the list.

Now I know this looks confusing but you will have your aha moment.
Okay, so we start by giving the algo an element x

The first thing we do is we put that element right before L.head by saying that x.next > after x, comes L.head.

Ok so line 2 is a conditional statement that checks to see if we have any elements in the list, ie, if the start of the List exists and it's NOT NIL.

Then we proceed to remove the previous links, we do that by saying, L.head = x, meaning the start of the list is x, and then we say before x comes nothing.

I know i might not have explained it very well if you need a further explanation message me on this.

```
LIST-DELETE(L, x)
1  if $x.prev \neq$ NIL
2      $x.prev.next = x.next$
3  else $L.head = x.next$
4  if $x.next \neq$ NIL
5      $x.next.prev = x.prev$
```

Finally we have the deletion function.
So first we pass it the element to be deleted, and then we run a check.
If, our element is not in the beginning of the list, or if x.prev is not NIL > element before x not NIL

Then we say, Take for the element, that comes before x (x.prev) assign it's next element to be the next element of x, or x.prev.next = x.next, what we do here is we essentially bypass the x element's link to the next element of the list.
Although, if we are at the start of the list, we simply just say that the new first element is the one after x, or x.next.

So we cleared out the prev link now we want to clear the next link, so we say, if x is not the last in the list, simply say that, the previous value of the element that comes after x, is the previous value of x, or x.next.prev = x.prev once again bypassing x,

And voila, x is out of the linked list, by way of unlinking it with it's next and previous elements and replacing them in the list to cover the gap.

Insertion-Sort:

```
INSERTION-SORT($A$)
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      // Insert $A[j]$ into the sorted sequence $A[1 \mathinner{.\,.} j-1]$.
4      $i = j - 1$
5      while $i > 0$ and $A[i] > key$
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```

Translation:

We initialise a **for loop** (1) to go through every element of the array, apart from the 1st one

//Note: the reason we start from the 2nd element in the array, is because we divide the array into 2 parts, anything on the left of the current index is sorted, and everything on the right isn't.

On every iteration we set the variable key to the value in the index of the array corresponding to the current iteration. We also initialise another variable. Here's the importance of this.

- Key holds the value we want to check on this iteration.
- i points to the index previous to the one in question.

// The reason for i, is because we want to look through the "sorted" portion of the array only and find the correct position to dump the key value.

So we've set up our loop up to line 4 this ensures that we will check every single element of the array.

Now we need to ensure that the element that we are examining is checked against every element of the sorted part of the array which is to its left. This is what the while loop does, it will go back from the element we are examining until it reaches 0 or, it finds the correct position to dump the element of the array.

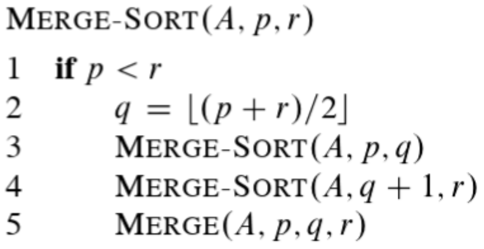
So line 6 all it does is basically move every element we checked one over to the right since it's not our magic line we want to move it one to the right.
Right after that we decrement i because we want to go check the next number in the sorted part of the array towards 0.

Once our algorithm finds a match, it simply moves our key value into that spot.

Merge-Sort:

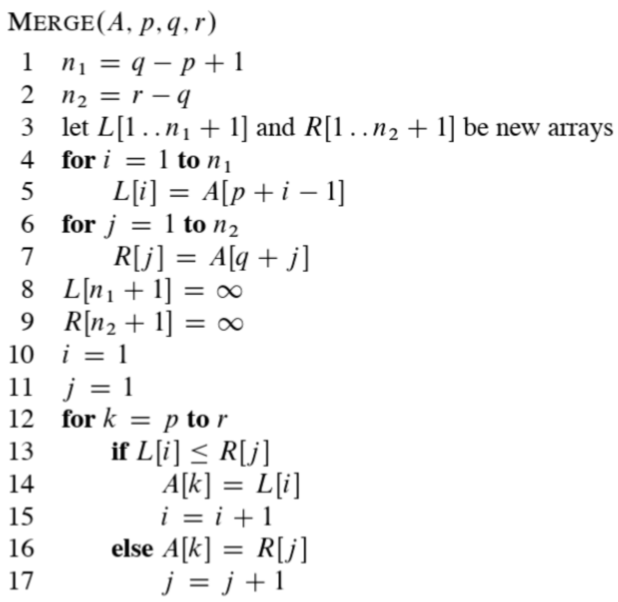
Merge-Sort (A,p,r) where A is the array we want to sort, and p is the start of the subarray, and r is the end of the subarray.

This checks that the array we gave it can be split into smaller parts line 1. The start of the array can’t be after the end :P



```
MERGE-SORT(A, p, r)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Then it assigns q, now q will be the midpoint, the point in which we will split the array in half so if M-S(A,0,13) q = 13/2 = 6 (remember floor). So then we call it recursively meaning, we split the array above from 0 to 6 and from 7 to 13, what do we do with these splits? We pass them back into the same function to be split down. Until we end up with single element arrays which are already sorted.



```
MERGE(A, p, q, r)
1  $n_1 = q - p + 1$
2  $n_2 = r - q$
3  let $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$ be new arrays
4  for $i = 1$ to $n_1$
5      $L[i] = A[p + i - 1]$
6  for $j = 1$ to $n_2$
7      $R[j] = A[q + j]$
8  $L[n_1 + 1] = \infty$
9  $R[n_2 + 1] = \infty$
10 $i = 1$
11 $j = 1$
12 for $k = p$ to $r$
13     if $L[i] \le R[j]$
14         $A[k] = L[i]$
15         $i = i + 1$
16     else $A[k] = R[j]$
17         $j = j + 1$
```

After that we call the merge function. So if we take the example array we will pass MERGE(A,0,6,13).

First we want to assign the number of values that exist on the “left part” of the new array, and then the right part. (we don’t actually create a new array up to this point, M-S just tells us the values we need to do that)

So we know we have n1= 6+1 elements “left”, and n2 =13-6 elements on the “right”

We then initialise the two new arrays, L and R, and tell them their sizes as from above so L(7) and R(7).

Then we populate them with the elements of the main array respectively.

So now we have 2 sentinel cards, these are there to help with writing the pseudo code. If we didn’t have these we would have to account for the number of elements to be checked using if statements to catch for i that’s out of bounds. It would need to check for both arrays, and then also account for each individual array being empty, telling us to dump the other array remaining elements back into A. Basically we’d have to check if the “deck” on either hand is empty.

Then we set, k to be equal to the beginning of the array, and start populating the array all the way to it’s defined end by comparing which hand L or R holds the larger value, then we increment i or j to ensure that we always pick the next L or R element depending on which one we pulled in the previous run of the for loop.

Treez

k = number of children under each node, so k-nary tree for k = 2, is a binary tree, i.e. two children.

h = the height of the tree, ie. the distance between the root, and the deepest leaf (in connections)
Important to note, nodes can also have height under the same principles.

Full tree = a tree where all nodes have 0 or k children based on a k-nary tree

Complete tree = all levels are complete except possibly the lowest, starting from left. (heaps)

Leaves = k^h will give the number of leaves

Internal Nodes = (2^h)-1

Heapsort:

The basic concepts here are, that a heap is a complete binary tree and that the following relationship exists:

Since a heap is an array represented in a binary tree, we must remember that, the parent always comes before the child, and that the formulas for calculating the indexes are

Parent floor(child_index/2)
Left parent_index * 2
Right **(**parent_index * 2) + 1

We populate the heap from the top, moving down, and always from left to right so to conform with it being a complete tree, and following the indexing of the array.

There are 2 subtypes of Heaps, a MAX and a MIN, the type refers to the relationship between nodes, MAX = children are always smaller than parents & MIN children are always bigger than parents.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

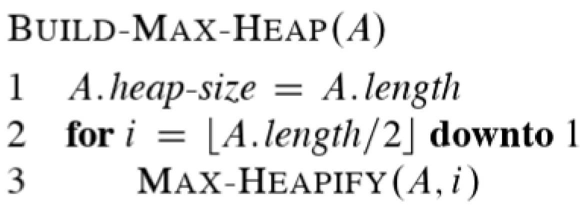
This method takes a node, and then compares it to its children to conform with the MAX heap standard.

First we find the children index in the array/heap, (function above) for parent i.

Then we look at if there are children, so if the index of l or r is greater than the size of the heap, and then we look at first the left child, and find if the parent or child is bigger. Then we compare the result of that to the right child, and see which is bigger, in the end the bigger child will move up and the parent will take its place below.

Then, we call the function recursively in order to proceed throughout the rest of the chain all the way up to the root node in a linear fashion.
This will not compare all the nodes of the heap, just the ones directly related to that subtree.

So in short, all leaves will be left alone in the initial procedure, and then all internal nodes will be compared to their children.

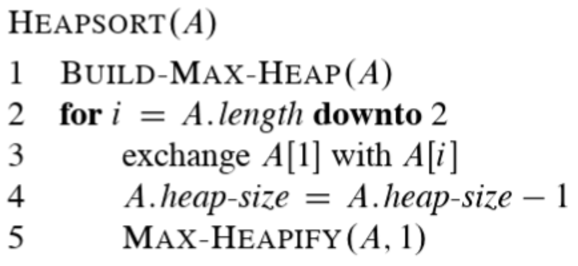


```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

To build the max-heap, we call this method in a for loop as seen here. This will ensure that every element will be sorted correctly.

NOTICE, the heap is already populated here, so we move BACKWARDS from right to left of the array/ heap that’s why there is downto 1.

We also don’t need to call this n times, only floor(n/2) since it’s a binary tree and we would be just checking everything twice. (i think)



```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Now for the actual Heap Sort function

We first call the build function, this will sort our heap in the max config as described above.

Here’s the important bit, we need to EXCHANGE, always the first element of the heap otherwise known as the root, with the i-th element of the heap.

As the root of the max heap is the biggest number, it will have to go to the end of our sorted array, HENCE… we start replacing our array from right to left, and degrement! In the for loop.

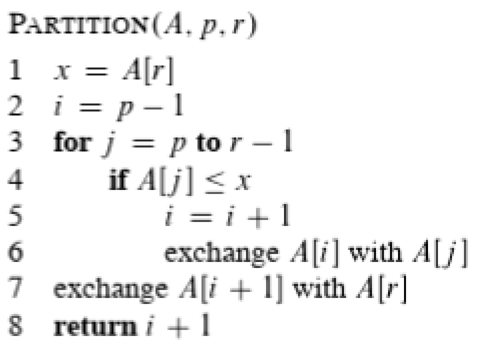
Then we call the max-heapify function so that the root of the tree that we “extracted” is replaced by the next biggest number.

If you have more questions on heap sort please let me know, it’s kind of hard to explain on paper, i’d be happy to show it graphically.

QuickSort

The best kind of sort. Nah i just like how efficiently simple it is.

So quicksort works by partitioning the array into two parts. The reasoning behind this is that quicksort sorts the numbers by selecting a pivotal element, and then comparing the array elements against it. Anything smaller than the pivot will be part of the left partition, and everything bigger than it will be part of the right.



```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

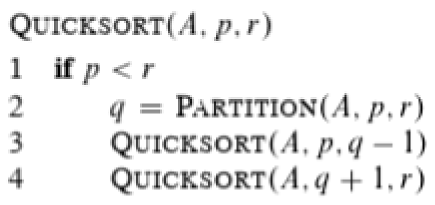
Here we pass to the method our array. A, and p = first element , and r the last element.

We then set the pivot, x which will be the last element in the passed array, and we introduce i, which will be an index value starting from -1 in code or 0 in pseudo.

Then we will iterate through the array elements, until the second to last element. Since we already have x as the pivot which is the last element.
We use 2 indexes here, i and j, so j is the element of the array that we are comparing against the pivot (x) and i is the position the next smaller than pivot element will be inserted.

To add to this visualisation, imagine that, j to r-1 are the “larger than x numbers” and 0 to i are the “smaller than x numbers.

As we iterate through the array, when we find a smaller element, we increment i to make space for the element to be moved, and then we move it.
If the next element under j is not smaller than x, nothing is done, since the element is already in the larger partition of the array.
Once we’ve reached the end of the array - 1 the for loop stops, and then we simply move the last element of the array which was our pivot, right after i, effectively in the middle of the 2 partitions.



```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

So now to analyse the sort. We give the function the initial array, A from start to finish indexes
Then it simply will check that we passed it an array that is valid for partitioning and sorting.

The we generate the q variable, this is the position of the “midpoint” of the given array, what this means is that since we run the partition, as we said we will be diving the array into 2 partitions, well that q is telling it where the first partition ends and the second begins.

The last 2 are recursive calls, so each time we split it down further and repeat the process until p is not smaller than r (well cause p will most likely be equal to r).

Direct Address Tables

Alright we now move onto the more heavy stuff, here is what we must keep in mind for these tables.

Universe (U) - Imagine an array, all the possible indexes denote the U values it doesn't mean they will be used but they are available to be used. For the purposes of this subject we will call these indexes key values.

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]
```

So if we're looking for a value all we have to do is pass the key or k, and then we will get in return the element in that particular index.

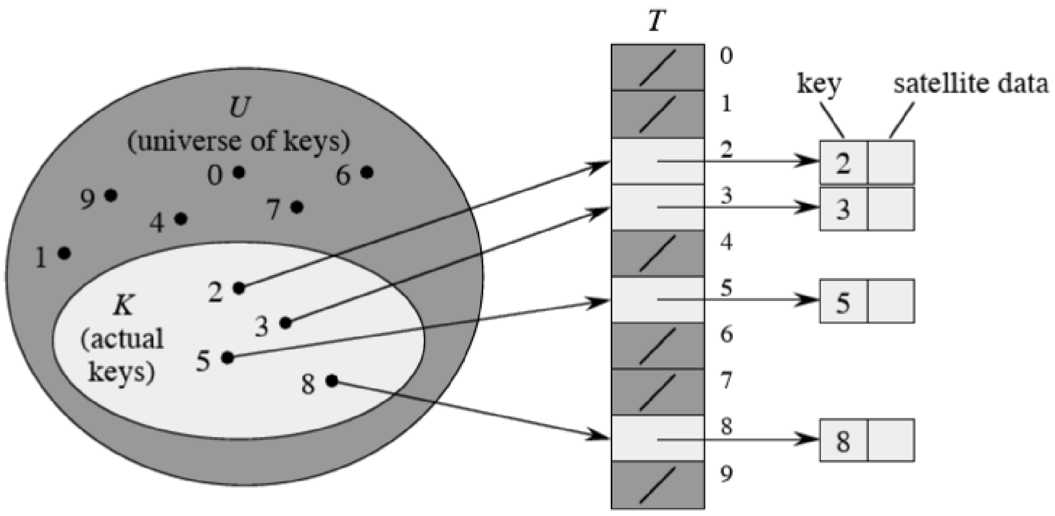
```
DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x
```

To add a new element, we simply say, that the key of element x is x.key and since we said that key must be a part of U, then it must be an index in the array, so we just add it in that position

```
DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

In delete, we don't simply remove the element, but mark it as NIL

NOTE if the universe U is large, storing a table of size |U| may be impractical or impossible. Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.



Hash Tables

Ok intro to hash tables, because direct addressing is impractical as it stores way too much data for our liking we've devised a method to take that down a notch. Here's how we do it. Important values for these:

- K - once again our beloved key value
- h(k) - This is a hashing function that takes the key value and transforms it (check hash functions) into a value we can use to store the info in our set more efficiently. (will be explained)
- M - this is the size of our hash table
- Load - How full is our hash table

So instead of saying that our key will be inserted in an index equal to that of the key, we have a function called a hash function that will transform this key to a value applicable to a smaller table with size of m.

Here's the issue though, what happens when we have two values with the same hash values meaning what happens when h(k1) = h(k2) ? we have a collision, meaning that both values will try to go into the same slot.

We can solve that using a number of methods that will be further described below:

Chaining

What chaining does is, instead of putting the value in the corresponding hashed key index, we instead institute a linked list system, where each slot has an empty linked list, and when a value is added to that slot it is essentially added to the linked list.

Let us see what that means, assume we had 2 values that landed on h(k) = 4 , we have a collision, so what we do is we take value a, and add it to the head of the linked list.
Then when value b lands at the same place, we add it to the head of the linked list and move the value a one over.

NOTE! If our hashing function is not good we could end up with all values in one slot thus defeating the purpose of a hash table

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]
```

We insert a value to the table by adding it to the head of the list, we add the value of the hash(key of x)

```
CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]
```

In search the only note is we are looking for the key k not the value x.

```
CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

And delete once again is done with the key of x.

Load Factor

Simple formula explanation here but it's important to remember.
To find the load factor we take:

$a = \frac{n}{m}$ where:

- A is the load factor
- N is the elements we are currently storing in the table
- M is the size of the table

Quick derivative formulas:

$n = a \times m$ $\qquad$ $m = \frac{n}{a}$

Hash Functions

Hash functions are the magical methods from which we transform a key into a usable key in our hash table.

Some values might not be numbers and therefore we need to convert them, that's why we've got the Radix system.

- Radix

Ok so for Radix, we are given a base value assume we have 2 scenarios

A. Base 10 with letters "pto"

p = 112, t = 116, o = 111

$$112(10)2 + 116(10)1 + 111(10)0$$

Were the number in the parenthesis represents the base, and the power represents the letters remaining

B. Base 128 with letters "pto"

$$112(128)2 + 116(128)1 + 111(128)0$$

And now we can convert letters to numbers woohoo

There are 2 main methods of hashing keys:

- Division method

We simply take a the function $key\ mod\ m$ and that will give us the hashed value of the key that we can then use to place our value in our hash table. As we said m is already defined.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k. Unless we know that all low-order p-bit patterns are equally

Translation, don't make the m be a power of 2 **:P**

- Multiplication method

I won't say a lot about it since we didn't really cover it but just so you have the formula:

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor \ ,$$

Where A is a number between 0 and 1, and remember the lines on the side mean floor value.

Open Addressing

Open addressing refers to the fact that we can now place any key value anywhere on the table instead of placing it only at the corresponding index such as with the chaining method and once again it is a method for collision resolution.

Under open addressing table operations differ a bit, we will see them bellow:

```
HASH-INSERT($T, k$)
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j] ==$ NIL
        $T[j] = k$
        return $j$
    else $i = i + 1$
until $i == m$
error "hash table overflow"
```

So we pass a key, k and the table, and then we start a Do while loop.
We of course initialise i, which will serve as our "counter" for the different hash functions under this method (will be addressed later)

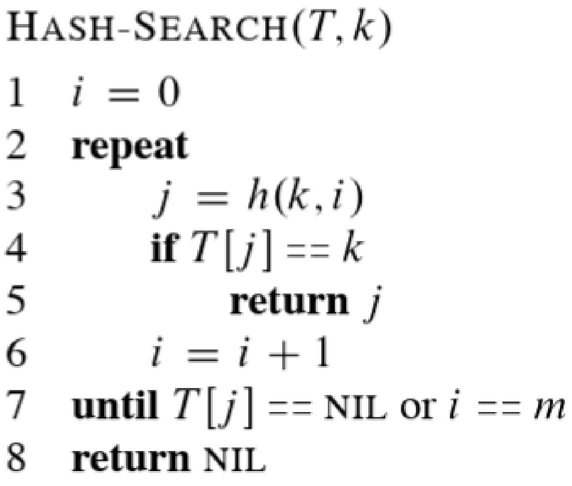
Then we name a variable J which will hold the hash value of k, with our chosen i (0 for 1st iteration and increasing afterwards)

Now if there is no element in the chosen slot, we simply place it into the slot and break out of the loop by returning the hashed index j

If there is an element we increase i by 1 and try again, this of course is repeated until we reach the end of the table which will prompt an error.

NOTE! Tables also loop around so i == m means we checked all the spots in the table and none were free.

NOTE! Both search and insert functions return j as a hashed key NOT the value of the element.



```
HASH-SEARCH($T, k$)
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j] == k$
        return $j$
    $i = i + 1$
until $T[j] ==$ NIL or $i == m$
return NIL
```

Now for search once again we pass the key, initialise a counter and start a do while loop.

We again pass the chosen hashing function with our counter.

And then we simply say, if the table in this position J, is equal to the key we're looking for then just give us the key and break out of the loop

Otherwise increment i and try again.

This loop runs with 2 conditions, if i == m, meaning we checked all the elements. Or if the slot we're looking at is NIL in which case we return NIL, as in **NOT FOUND**

Probing Methods

Now let's talk about j in both this functions, or otherwise defined as **h(k,i)**
As we said h refers to the hash function to determine the slot of placement, therefore it is used both in ascertaining the location of the next insertion and the location of the search.

Here are 3 methods for probing:

- Linear Probing:

$$h(k, i) \ = \ (h(k) \ + \ i) \, mod \, m$$

We've already said what m is, and what k is, so what does this function do?
It takes the key and the i from the function we're using it in, see above search and insert.

Now let's assume for ease we use the division method for h. So let's give some values and run an example function.
$i \ = \ 0, \ m \ = \ 28, \quad k \ = \ 156$

$h(156, 0) \ = \ (\, h(156) \ + \ 0) \, mod \, 28 \quad =>$
$(156 \, mod \, 28) \, mod \, 28 \quad =>$
$16 \, mod \, 28 \ = \ 16$

So let's assume that our 1st iteration didn't work and our i became 1 let's see it again.

$i = 1, \ m = 28, \quad k = 156$

$h(156, 1) = (h(156) + 1)\, mod\, 28 \quad =>$

$((156\, mod\, 28) + 1)\, mod\, 28 \quad =>$

$17\, mod\, 28 = 17$

It should now be clearer how linear probing works, we just look one over each time :)

- Quadratic Probing

The book complicated things a bit here, so I will try to simplify based on research online on the subject:

$$h(k, i) = (h(k) + i \times i)\, mod\, m$$

So this method will look in exponential fashion to find the slot at which our number is supposed to be inserted and found.
Let's pose 2 examples.

$i = 1, \ m = 28, \quad k = 156$

$h(156, 1) = (\, h(156) + 1^1\,)\, mod\, 28 \quad =>$

$((156\, mod\, 28) + 1)\ mod\, 28 \quad =>$

$17\, mod\, 28 = 17$

So this would be our second iteration looking at the slot +1 from the original slot we looked at and for i = 1. Let's look at the 3rd iteration.

$i = 2, \ m = 28, \quad k = 156$

$h(156, 1) = (\, h(156) + 2^2\,)\, mod\, 28 \quad =>$

$((156\, mod\, 28) + 4)\ mod\, 28 \quad =>$

$20\, mod\, 28 = 20$

So in our 3rd iteration for i = 2 we look at +4 slots to find the key, and there you have it, that's quadratic probing for you.

- Double Hashing

We basically run 2 hash functions, together. The formula is:

$$h(k, i) = ((h_1(k) + (i \times h_2(k))) \, mod \, m$$

Let's explain, the 1st iteration will have i = 0, so in any case it will point us to the actual correct slot. BUT consecutive iterations will be different. The way this is done is by defining a second hash function.

Assume the following:

$i = 1, \; m_1 = 28, \; m_2 = 27, \; k = 156$

We know that for i = 0 it will hash at slot 16, now let's see for i = 1:

$$h(156, 1) = ((156 \, mod \, 28) + (1 \times (156 \, mod \, 27))) \; mod \, 28 \;\; => \;\; (16 + 21) \, mod \, 28 \;\; => \;\; 37 \, mod \, 28 = 9$$

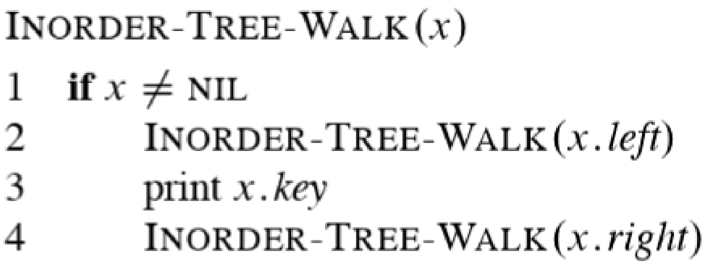
So to decipher these hieroglyphics, all it means is that on every additional iteration of the search or insert functions we will offset the slot we're looking at by 21 slots. That's what the 2nd hashing function gives us.

Binary Search Trees

Ok we already talked about trees, the difference in BSTs is, this relationship:

$$x.left \leq x \leq x.right$$

All this means is that the left child must be smaller than the parent and the right child bigger than the parent which applies throughout the tree.

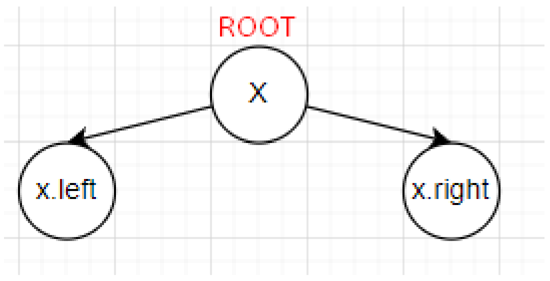


```
INORDER-TREE-WALK($x$)
if $x \neq$ NIL
    INORDER-TREE-WALK($x.left$)
    print $x.key$
    INORDER-TREE-WALK($x.right$)
```

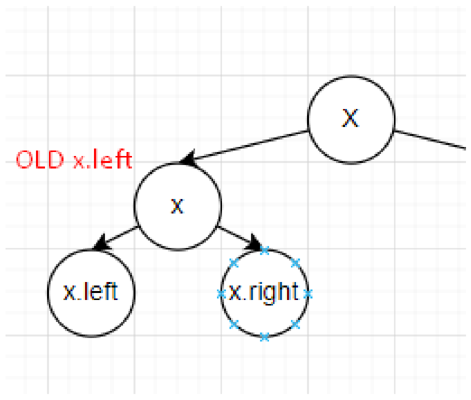
This function prints the elements of the tree in order by calling itself recursively from top or root to bottom.

It checks if the x is NIL, meaning there are no more nodes to check, and then.

It passes the left child recursively into the same search, prints the x, and then passes the right child into the same algorithm.



To quickly clarify this would be the first iteration parameters. AND, we would pass in the x.left and x.right recursively so here's how the second one would look



And printing from left to right we would always end up printing all the values in order. Pretty neat :)

Let's talk about searching a binary tree, now we already know how it looks when we print, so why would it be any different when we check for the values right ?

```
TREE-SEARCH$(x, k)$
1  if $x == \text{NIL}$ or $k == x.key$
2      return $x$
3  if $k < x.key$
4      return TREE-SEARCH$(x.left, k)$
5  else return TREE-SEARCH$(x.right, k)$
```

So we first look at the parent, and see if we are at the end of the tree or if we've simply found the element we're looking for straight up.

If not we simply check, is the key we're looking for smaller than the element we;re currently looking at? If yes we need to go to the left side of the tree and check again.

If it's bigger then we have to go to the right side of the tree and check again!

We could also search iteratively instead of recursively by:

```
ITERATIVE-TREE-SEARCH$(x, k)$
1  while $x \neq \text{NIL}$ and $k \neq x.key$
2      if $k < x.key$
3          $x = x.left$
4      else $x = x.right$
5  return $x$
```

While the value we're looking at is not NIL, and it's not equal to the value we want: Meaning we haven't reached the end of the tree or our wanted value.

Then we simply run the same checks, is it smaller? Then go left, is it bigger ? then go right.

Once we're out of the while loop we simply return the position back.

```
TREE-MINIMUM(x)
while $x.left \neq$ NIL
    $x = x.left$
return x
```

Very simple function to find the minimum element of the tree, we already said that smaller than the parent goes to the left, soooooo, if we just keep going left and down, we will eventually get the smallest element.

Basically all we;re saying is, as long as we haven’t reached the end of the tree, keep going left. Once we have, stop and return the x.

```
TREE-MAXIMUM(x)
while $x.right \neq$ NIL
    $x = x.right$
return x
```

Exactly the same logic as the minimum really no further explanation needed here.

Successor function finds us the next biggest element after x. How it works, let's see.

```
TREE-SUCCESSOR(x)
if $x.right \neq$ NIL
    return TREE-MINIMUM(x.right)
$y = x.p$
while $y \neq$ NIL and $x == y.right$
    $x = y$
    $y = y.p$
return y
```

First we check, if x has a right child, and if it does, we just want to find the smallest element in that right subtree.

What this means is, we’re looking for the next biggest element, so we go right for the biggest, but then we want the smallest of the biggest elements. So we call the tree-min function to find the smallest of the biggest :)

If x doesn’t have a child, we need to check differently, we need to go up, by looking at the parent of x or x.p. So while a parent exists and our element is not equal to the x we want, we keep going up until we find it, keep in mind that we always look in the right child for the next biggest element as our successor and not the left.

Next we’ll look at insertion and deletion which are more complex than these.

```
TREE-INSERT($T, z$)
$y$ = NIL
$x = T.root$
while $x \neq$ NIL
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$
if $y$ == NIL
    $T.root = z$          // tree $T$ was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
```

So we pass a table and the element to be inserted with the value z.

We set y to be empty and x to be the root of the table, since that's where we want to start.

So, while the element we're looking at (x) is not empty, we set y to be x.

Then we check, is z smaller or bigger than x, if it's bigger we need to go right, and left if it's smaller.

So once we've reached our desired position we need to set our element in the list, we do that by defining its parent z.p = y, where y has become the last element we checked before finding our desired position.
If it's NIL, well the parent is the root, it means the tree was empty, we defined Y as NIL,in the beginning and there were no elements.
Otherwise, we check z relationship with its parent, and decide, is it the left or right child based on if it's bigger or smaller than the parent.

Now we'll introduce transplant so we can understand deletion afterwards.

```
TRANSPLANT($T, u, v$)
if $u.p$ == NIL
    $T.root = v$
elseif $u$ == $u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$
```

So we pass in 2 nodes u and v.
So, if u's parent is NIL, then we make the root of the table be v.
Otherwise, if u is the left child, we assign v as the left child.
Otherwise if it's the right child then we assign v as the right child.

If v is not NIL, meaning it was replaced by u
then we make the parents of both children v and u to be the same value.

This is a big foggy I know, a video with visual representation could help you understand it better.

Now let’s use transplant to study deletion:

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

We pass the element to be deleted, and then,
If the left child is NIL, we transplant z we it’s right child.
Otherwise if the right child is NIL, we transplant z with its left child.
Otherwise we simply say that y is the minimum on the subtree of the right child of z.

And then we check, if y’s parent is not z, we transplant y with its right child.
Then we set the right child's parent to be y.

Then we transplant z with y, and we set the left child of y to be the left child of z.

And the left child's parent to be y.

Effectively we are finding the element we want and then we isolate it from the tree so that it is removed.

Next we’ll look at Red Black Trees

Red Black Trees

RB trees are binary trees with extra properties that we will analyse below:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red black trees basically ensure that we have an approximately balanced tree thanks to these properties. The reason why is because it restricts consecutive nodes and paths so that you can't make unbalanced trees easily.

To add to the rules,

- you can have 2 consecutive black nodes **BUT NOT** 2 consecutive red nodes.
- Height or h = the number of sides from the node in question to the NIL nodes (longest path)
- Black Height or bh = the number of black nodes to the NIL nodes not including the node in question but including the NIL node.

Now let's move onto the operations for the RB tree. Because we need to keep the tree from violating its properties, we need to have a way to perform operations on the RB tree, while keeping the properties intact.

```
LEFT-ROTATE(T, x)
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

That's why we use a function to rotate the nodes to preserve the properties.

We pass it node x, and we then set y to be the right child of x.
Then we define the right child of x to be the left child of y.

Now if the left child of y is not NIL (end of tree) then we say that its parent is x.
And then we say that y's parent is x's parent.

If x's parent is NIL (root) then y is the new root of the table.

Otherwise if x is the left node, then the left node changes to y,

Otherwise the right child is y.

Then the left child of y is x, and the parent of x is y

```
RB-INSERT($T, z$)
$y = T.nil$
$x = T.root$
while $x \neq T.nil$
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$
if $y == T.nil$
    $T.root = z$
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
$z.left = T.nil$
$z.right = T.nil$
$z.color$ = RED
RB-INSERT-FIXUP($T, z$)
```

```
RB-INSERT-FIXUP($T, z$)
while $z.p.color$ == RED
    if $z.p == z.p.p.left$
        $y = z.p.p.right$
        if $y.color$ == RED
            $z.p.color$ = BLACK
            $y.color$ = BLACK
            $z.p.p.color$ = RED
            $z = z.p.p$
        else if $z == z.p.right$
                $z = z.p$
                LEFT-ROTATE($T, z$)
            $z.p.color$ = BLACK
            $z.p.p.color$ = RED
            RIGHT-ROTATE($T, z.p.p$)
    else (same as then clause
            with “right” and “left” exchanged)
$T.root.color$ = BLACK
```

```
RB-TRANSPLANT($T, u, v$)
if $u.p == T.nil$
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
$v.p = u.p$
```

```
RB-DELETE($T, z$)
$y = z$
$y\text{-}original\text{-}color = y.color$
if $z.left == T.nil$
    $x = z.right$
    RB-TRANSPLANT($T, z, z.right$)
elseif $z.right == T.nil$
    $x = z.left$
    RB-TRANSPLANT($T, z, z.left$)
else $y$ = TREE-MINIMUM($z.right$)
    $y\text{-}original\text{-}color = y.color$
    $x = y.right$
    if $y.p == z$
        $x.p = y$
    else RB-TRANSPLANT($T, y, y.right$)
        $y.right = z.right$
        $y.right.p = y$
    RB-TRANSPLANT($T, z, y$)
    $y.left = z.left$
    $y.left.p = y$
    $y.color = z.color$
if $y\text{-}original\text{-}color$ == BLACK
    RB-DELETE-FIXUP($T, x$)
```

```
RB-DELETE-FIXUP(T, x)
while $x \neq T.root$ and $x.color$ == BLACK
    if $x == x.p.left$
        $w = x.p.right$
        if $w.color$ == RED
            $w.color$ = BLACK
            $x.p.color$ = RED
            LEFT-ROTATE($T, x.p$)
            $w = x.p.right$
        if $w.left.color$ == BLACK and $w.right.color$ == BLACK
            $w.color$ = RED
            $x = x.p$
        else if $w.right.color$ == BLACK
                $w.left.color$ = BLACK
                $w.color$ = RED
                RIGHT-ROTATE($T, w$)
                $w = x.p.right$
            $w.color = x.p.color$
            $x.p.color$ = BLACK
            $w.right.color$ = BLACK
            LEFT-ROTATE($T, x.p$)
            $x = T.root$
    else (same as then clause with "right" and "left" exchanged)
$x.color$ = BLACK
```

I cannot explain the red black trees algorithms in writing at the moment, as they are too complex and I need to prioritise and not put all my eggs in one basket. I listed all the algorithms and I will add visualisation resources and pages from the book in Added resources if you need help here.

Priority Queues

Priority Qs are just like regular Qs, except the elements inside have their own priorities, for example a min-priority Q, and max-priority Q, in which the smaller elements have higher priority and the higher elements have higher priority respectively.

Now the priority Q, is stored in a heap, and the root will be the most “prioritised” element therefore for a max-priority Q.

```
HEAP-MAXIMUM(A)
1  return $A[1]$
```

We simply return its first element as the highest priority.

Now if we were to extract the maximum element as well the operation would look a bit differently.

```
HEAP-EXTRACT-MAX(A)
1  if $A.heap\text{-}size < 1$
2      error “heap underflow”
3  $max = A[1]$
4  $A[1] = A[A.heap\text{-}size]$
5  $A.heap\text{-}size = A.heap\text{-}size - 1$
6  MAX-HEAPIFY$(A, 1)$
7  return $max$
```

First we make sure the heap is actually filled with elements, and pop an error if it isn’t

Then we simply define the max element (we already said which one it would be)

Then we redefine the root element of the heap to be the last element of the heap. The reason behind this is, we can’t simply remove any element from a heap, we can only remove the last one, so that we keep from violating the properties of the heap.

So since we removed the old root which was the max element, and we took the last element of the heap and placed it at root, we can move forward, our heap is 1 smaller so we make sure to change the size.
BUT! Since we moved the last element to root, our heap is no longer a max heap, soooo, we simply call max heapify to convert it back to a max heap.

Then we return max, which in all this has retained the old maximum value element of the heap.

A similar operation could be performed for min heap, by changing the line 6 to min-heapify.

This operation increases the value of the key (element) in the max-heap (max-priority Q)

```
HEAP-INCREASE-KEY($A, i, key$)
if $key < A[i]$
    error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$
```

We pass it the location of the element along with the new key value we want.

Then we make sure the new value is not smaller than the old one.

We set the new value, and then we need to do corrections on the heap.

While there are still elements in the heap subtree and the parent of the element we changed is smaller than the element we changed, we swap them, and go up one more.

```
MAX-HEAP-INSERT($A, key$)
$A.heap\text{-}size = A.heap\text{-}size + 1$
$A[A.heap\text{-}size] = -\infty$
HEAP-INCREASE-KEY($A, A.heap\text{-}size, key$)
```

We can also insert a new key,
First we increased the heap size and set that the last element in the heap is - infinity to make it easier for our function to execute without too much code. (this is a sentinel ofc)

Then we simply take the last element and change it to the desired key by calling the previously explained function.

All these functions can be called on a max-priority Q (max-heap) and with some modification they can be called almost identically on min-priority Q (min-heap)

Huffman Trees

Huffman encoding is an amazing subject that has to do with compression. Here are the fundamentals.
Every character is depicted in 8 bit codes (0 , 1) e.g A = 01000001 , that's 8 bits right there.

Now Huffman encoding comes in and says something amazeballz, they say well do you really need 8 bits?
What if you have the text **"BOOKEEPEROOEERRBBEE"** , that's 19 characters, under or **152 bits**. Nah way too much let's find a different way to encode it.
We know that in the text, we have the letters b , e , o , k , p , r right, so how many bits would we need to encode each letter?

Since each letter needs to be unique we can assign it a unique code, without using 8 bits (the 8 bit system was designed for 128 characters but we only have 6 here)

So how many bits do we need?
We would need to use 3 because 3 can represent a total of 8 different character combinations or permutations. Here's how:

0,0,0
0,0,1
0,1,0
0,1,1
1,0,0
1,0,1
1,1,0
1,1,1

Now we also need a frequency of appearance of each letter in our word. To keep our values clear let's put them on a table:

CHARACTER	FREQUENCY	BIT CODE	Bit Cost
b	3	000	3 x 3 = 9
e	7	001	7 x 3 = 21
o	4	010	4 x 3 = 12
k	1	011	1 x 3 = 3
p	1	100	1 x 3 = 3
r	3	101	3 x 3 = 9

So by using this fixed method, we multiply the frequencies with the bits which are 3 since we have 3 slots, and we find that our new cost for the message is **57 bits**

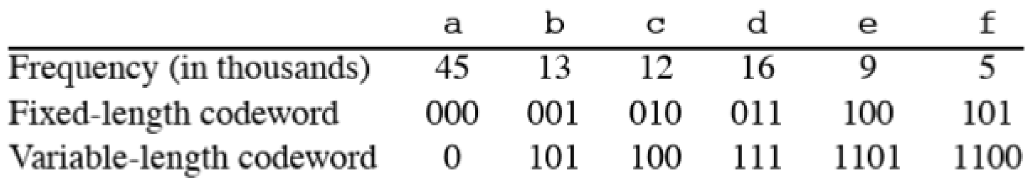
BUT! We also need to send the table in its original form, so we're sending 6 letters with 8 bits each, and then 6 codes with 3 bits each, so…. (6 x 8) + (6 x 3) = **66 bits**

So the message that originally cost **152b** now costs **123b**

NOTE! We need to send only the letters in their 8bit form, and the new codes, nothing else! So that the receiving computer can decode our message.

This is a fixed code huffman, in which all characters get the same number of bits. Now we'll look at the variable code:

The Variable code length assignment has 1 difference only, it will assign different code lengths to characters used more often and longer to the ones used less often.
Here's an example image

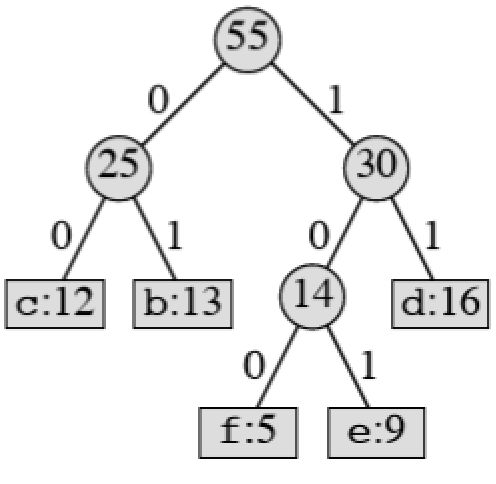


	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Now to find the Variable length code, we use a magical tree called a Huffman Tree:

Imagine a tree where all the characters we want are the leaves, and all the paths contain a 0 or 1.

In particular after we build the tree, we assign all left paths as 0s, and all right paths as 1s



Note that each letter is accompanied by its frequency :)

Now as we can see the left and right paths have 0s and 1s.

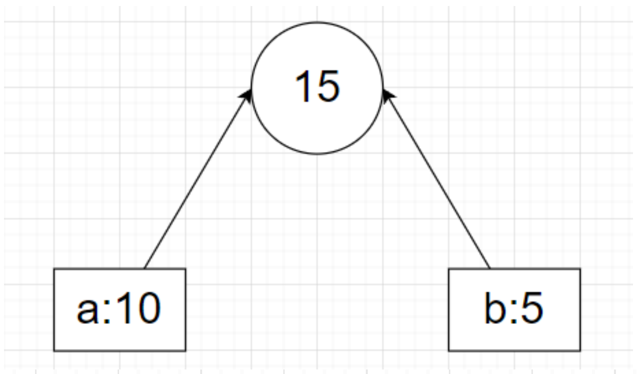
So to get the code for **c**, we would go down the path to **c**, and we get the code 0 0
For **e** we would get 1 0 0

Let's talk about how we build the Huffman tree on the next page.

So we pass the algo to create a Huffman tree, a set C, which is defined as a set of elements that have 2 attributes, a letter, and a frequency.

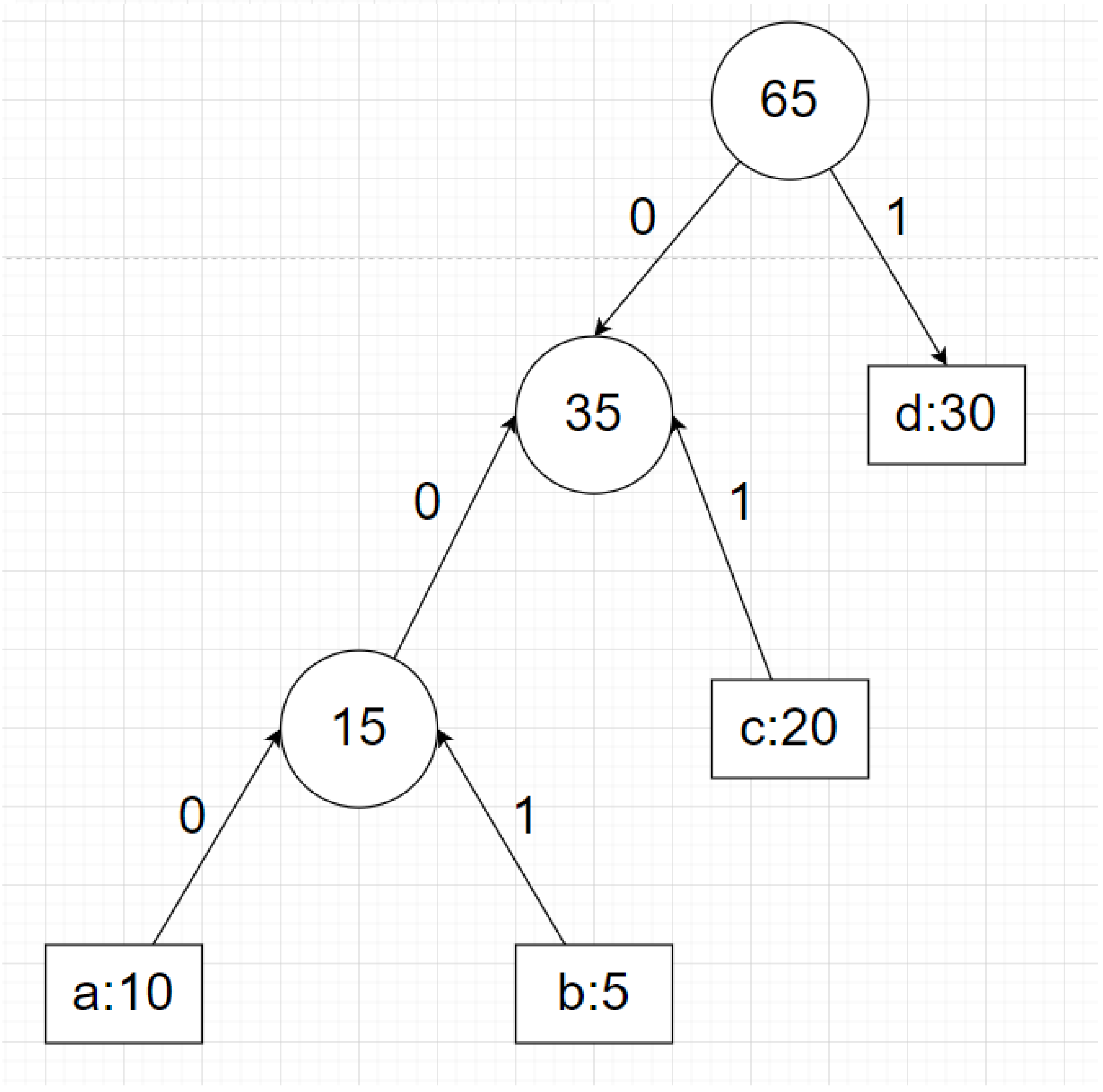
The algorithm uses the frequency of the letters to create the tree from top to bottom.
This is done by combining for example 2 elements with the lowest frequencies, and then rooting them under a parent which would hold the sum of the children elements.

Assume we had only a:10 and b:5



Their configuration would look like this, now assume we also had a c:20 and a d:30

Here's where things will spice up a bit instead of merging c and d like before, now the smallest numbers are 15, and c:20 . Mind blowing I know, so you see the higher frequency characters now have less bits representing them than the ones with lower frequencies.



a > 000
b > 001
c > 01
d > 1

As we can see this amazing tree is now a huffman tree woop.

Now let’s talk hieroglyphics, cause why not confuse you even further right?

```
HUFFMAN($C$)
$n = |C|$
$Q = C$
for $i = 1$ to $n - 1$
    allocate a new node $z$
    $z.left = x =$ EXTRACT-MIN($Q$)
    $z.right = y =$ EXTRACT-MIN($Q$)
    $z.freq = x.freq + y.freq$
    INSERT($Q, z$)
return EXTRACT-MIN($Q$)      // return the root of the tree
```

We already introduced the C, before as a set of $n: freq$ We do the same thing here.
Now **note**, we have a min-priority here, as in we prioritise smaller frequencies.

We set n to be the number of elements, and Q to be a min-priority Q with values of C.

Then we iterate through all the elements of C.
We create a new node z, we connect z with the 2 minimum items on our Q, 1 for left and one for right, then we place the frequency of the new node to be the sum of its children and we insert it into the tree.

Then we return the root of the tree which should also be the sum of all frequencies.

Graphs

There are 2 types of graphs, **directed and undirected** .

Edges are essentially the sides. Vertexes are the nodes.

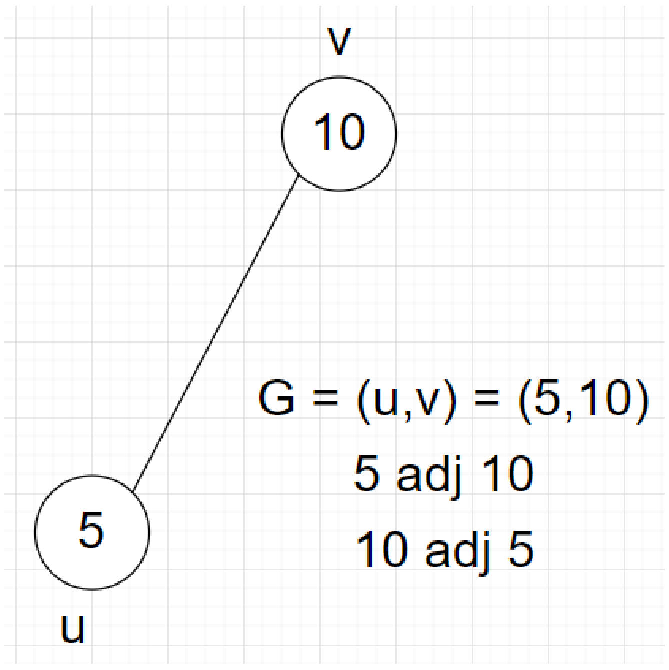
Directed Graph means that the sides or vertices of the graph carry a direction attribute, you can move along the vertice in the direction assigned but not the other way around.

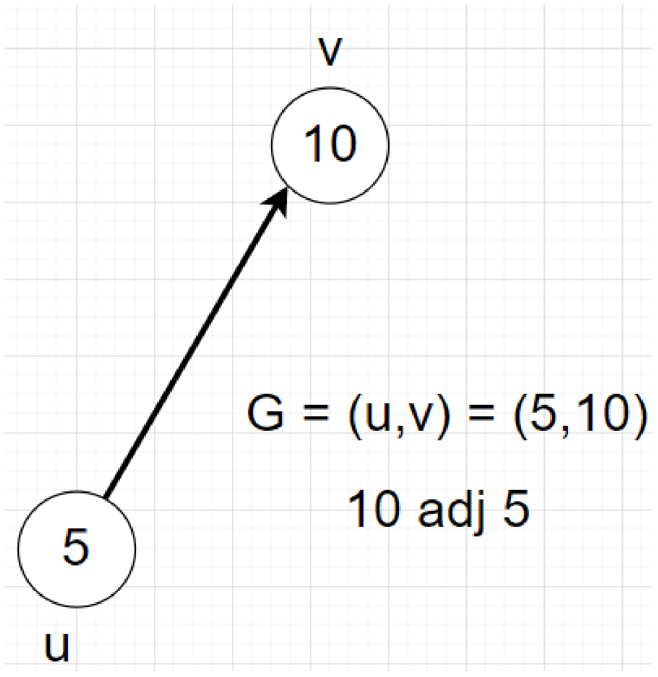
Undirected Graph means that the sides or vertices don't carry a direction attribute and you can move freely between nodes in any direction.

Adjacency

In an edge (u,v) where the u and v denote 2 different nodes, we can say that v is adjacent to u. **BUT!** We need to note if the graph or side is directed or undirected:

- Directed Graphs:
  - In edge (u,v) v is adjacent to u
  - U is not adjacent to v unless explicitly stated otherwise.
- Undirected Graphs:
  - In edge (u,v) v is adjacent to u
  - U is adjacent to v





So you see, it's the endpoint of the direction that has the adjacency and not the originator of the direction.

You can also see the symmetrical relationship in the left picture, and the non symmetrical relationship in the right.

Degrees

The edges originating or ending in a node, are called degrees.

If a node has 0 degrees it’s **isolated** (it’s a node but it’s not connected to the graph)

In an unidirectional graph if there is an edge (line) that means that both nodes connected by it have a degree > 0.

In a directional graph there are two types of degrees:

- In degree:

The edges the node is receiving from other nodes.

- Out degree:

The edges originating from the node going to other nodes

The degree of a node in this case would be *degree = (in_degree + out_degree)*

Paths

Paths are the routes we take to traverse a graph, they hold an attribute called length let’s call it k.

The attribute k is equal to the number of edges we crossed to get to the end of the path from the starting node.

- Simple Path

All nodes in the path are distinct (no double crossing a node)

- Circle

This applies for directed graphs. A path is a circle, if its starting point is also its ending point.

- Simple Circle

Basically the two conditions above smooshed together.

- Self Loop

A path with k = 1, where it starts from the node and loops back up into the same node.

- Connections
    - An undirected graph is connected if all nodes have a degree > 0 (they are all connected.)
    - A directed graph is strongly connected if every 2 nodes are reachable from each other.
- Acyclic

A graph with no simple circles.

Other

- Complete Graph

An undirected graph in which every pair of nodes is adjacent.

- Bipartite Graph

A graph that can be split in 2 parts

- Forrest

An acyclic undirected graph.

- Tree

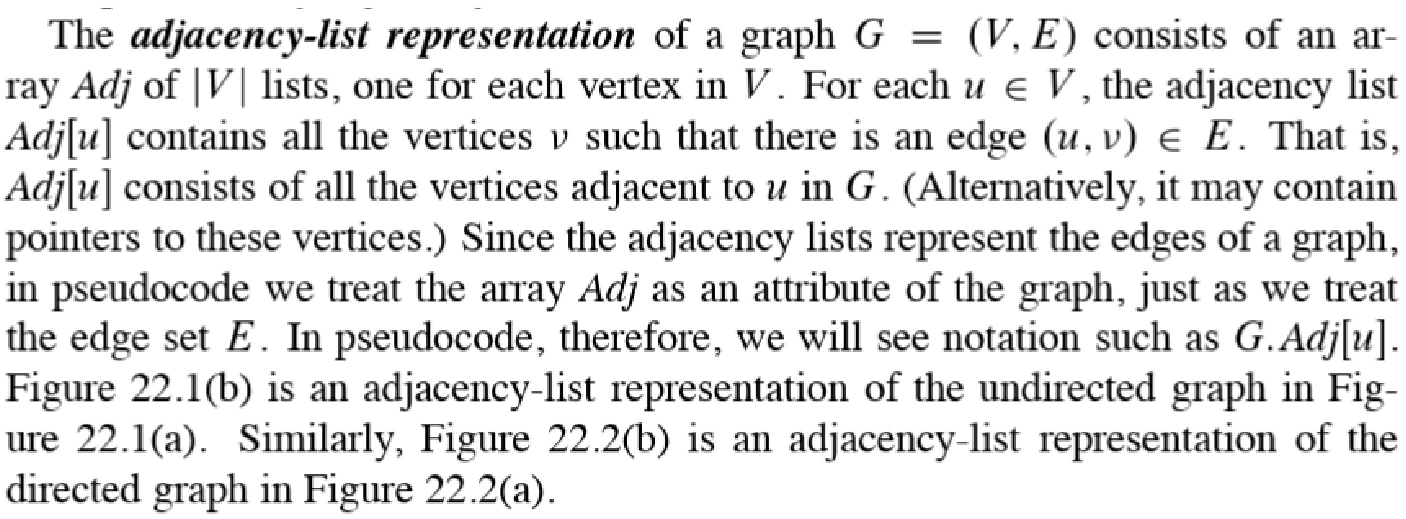
A connected acyclic undirected graph.

- DAG

The book sucked, here's wikipedia.

"a **directed acyclic graph** (**DAG** or **dag** /ˈdæɡ/ (listen)) is a directed graph with no directed cycles. That is, it consists of vertices and edges (also called *arcs*), with each edge directed from one vertex to another, such that following those directions will never form a closed loop. A directed graph is a DAG if and only if it can be topologically ordered, by arranging the vertices as a linear ordering that is consistent with all edge directions"[1]

[1] Fuck off lol im not doing sources :P



The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array *Adj* of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph, in pseudocode we treat the array *Adj* as an attribute of the graph, just as we treat the edge set E. In pseudocode, therefore, we will see notation such as $G.Adj[u]$. Figure 22.1(b) is an adjacency-list representation of the undirected graph in Figure 22.1(a). Similarly, Figure 22.2(b) is an adjacency-list representation of the directed graph in Figure 22.2(a).

For the ***adjacency-matrix representation*** of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \; , \\ 0 & \text{otherwise} \; . \end{cases}$$

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using our usual notation, such as $v.d$ for an attribute d of a vertex v. When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute f, then we denote this attribute for edge (u, v) by $(u, v).f$. For the purpose of presenting and understanding algorithms, our attribute notation suffices.

Added Resources

- Time Complexity

p.p 44-51 / Also use the canvas units for this, they have it all combined
p.p 147-150

- Stacks

p.p 229-234 / https://www.cs.usfca.edu/~galles/visualization/StackArray.html

- Queues

p.p 234-235 / https://www.cs.usfca.edu/~galles/visualization/QueueArray.html

- Linked Lists

p.p 236-240 / https://visualgo.net/en/list

- Insertion Sort

p.p 16-27 / https://visualgo.net/en/sorting

- Merge Sort

p.p 29-37 / https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

- Heap Sort

p.p 1176-1179 / https://visualgo.net/en/heap
p.p 151-162

- Quick Sort

p.p 170-180 /http://www.cs.miami.edu/home/burt/learning/Csc517.091/workbook/partition.html

- Hash Tables

p.p 253-275 / https://www.cs.usfca.edu/~galles/visualization/OpenHash.html
// https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html

- Binary Search Trees

p.p 286-298 / https://visualgo.net/bn/bst

- Red-Black Trees

p.p 308-328 // https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

- Priority Queues

p.p 162-165 //

- Huffman Trees

p.p 428-433 / https://people.ok.ubc.ca/ylucet/DS/Huffman.html

- Graphs

p.p 1169-1172 // p.p 589-592