Data Structures

Data structures is a general term referring to all of the various ways to store data, and the study of data structures is an intersection between mathematics and computer science. Any given data structure is not better than another. Rather, different data structures have different properties and use cases so their utility is situation dependent. The underlying implementation of every data structure is either array-based or node-based (linked list or tree) and some data structures can be implemented in both ways. This document provides conceptual explanations and visual representations of many of the most important and fundamental data structures and additionally discusses some of their time complexities with an emphasis of Big-O notation. There is also some information about various sorting algorithms and specifically an introduction to quicksort.

Notes:

- The C language is used when discussing data structures as they relate to code.
- Addresses are unsigned integers in a computer's memory. For the purposes of demonstration the visual representations will use uppercase letters as addresses.
- A data structure can be used for any type of data. For the purposes of demonstration the examples shown will use integers.
- For the data structures that have both array-based and linked-list based implementations, there is no discussion on the advantages and disadvantages of each option since that is beyond the scope of this document. Generally speaking, it can be said that if there are an unknown amount of elements then array-based implementations present a downside due to the potential resize operations at runtime whereas linked list-based implementations do not require resize operations. On the contrary, array-based implementations are better than linked list-based implementations in regards to cache performance. Those are just a couple of examples and there is more discussion to be had.
- Time complexities for data structures are discussed in this document at times. There are three major types: $O(N)$, $\Omega(N)$, and $\Theta(N)$. $O(N)$ is the worst case runtime and is referred to as Big-O which stands for Big-Ordnung. $\Omega(N)$ is the best case runtime and is referred to as Big-Ω which stands for Big-Omega. $\Theta(N)$ is the average case runtime and is referred to as Big-Θ which stands for Big-Theta.
- Big-O, Big-Ω, and Big-Θ do not *technically* mean worst, best, and average case. They actually are mathematical definitions in regards to upper and lower bounds of equations. For example, for some equation $T(N)$ representing the running time of an operation, $T(N) = O(N^2)$ would mean N^2 provides an asymptotic upper bound for an operation and $T(N) = \Omega(N^2)$ means N^2 provides an asymptotic lower bound for an operation. If $T(N) = \Omega(N^2)$ and $T(N) = O(N^2)$ then $T(N) = \Theta(N^2)$ which means N^2 provides an asymptotic tight upper and lower bound for an operation. This is something that would be studied in a more advanced algorithms course. For the purposes of this document and for beginning to learn data structures, it is ok to think of them as the worst, best, and average case and that's how they should be thought of. This is just being mentioned so that there is no misinformation on what they actually mean.
- There are also two other time complexities: $o(N)$ and $\omega(N)$ referred to as little-o and little-omega. Like with $\Omega(N)$ and $\Theta(N)$, these are not discussed in this document and would be studied in a more advanced algorithms course.

Table of Contents

Section	Page
Vector	3
Linked List	4
Trees (General) and the Binary Search Tree	5
Tree Traversals	7
Important Note	8
Stack	9
FIFO (Regular) Queue	10
Priority Queue - Heap	11
Priority Queue - Binomial Queue/Heap	18
AVL Tree	20
Hash Table	24
Sorting Algorithms	25
Time Complexity (General Discussion)	31
Data Structure Time Complexities (Big-O/Worst Case)	34
Sorting Algorithm Time Complexities	38

Vector

Vector: An array with a non-fixed capacity. The array will resize itself if it runs out of space. This is different from an array with a fixed capacity like:

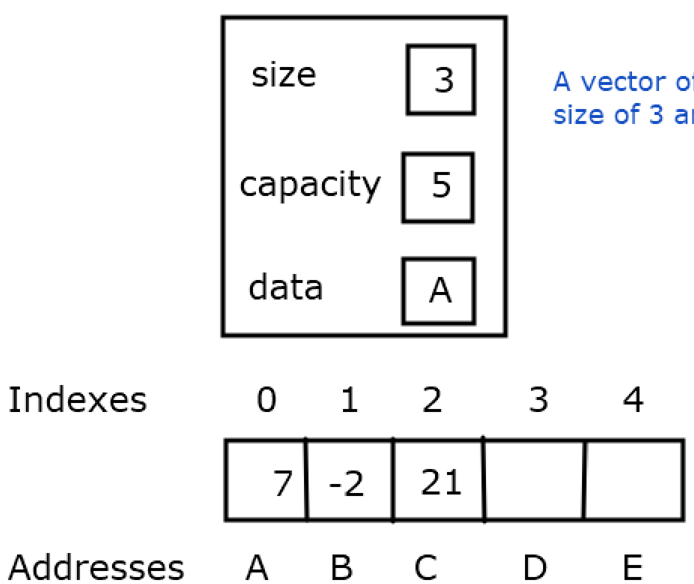
```
int a[10];
```

That array has a capacity of 10 elements and its valid range of indexes is [0, 9]. This can never change. A vector is one of the most fundamental data structures because it is also used to implement other data structures as will be seen later in this document.

A vector has the following components:

- **size** - The amount of items in the array which is initially 0. The index of the next available position is [size] and the index of the most recently added item is [size - 1].
- **capacity** - The amount of items the array can hold. If the size equals the capacity when adding a new item, a resize operation on the array must be performed. The array can be resized in any way to include doubling the capacity, increasing the capacity by some proportion (for example, increasing it by ⅓ of the current capacity) or only increasing the capacity by 1. Any given option involves a tradeoff between saving memory and performance during runtime. For example, increasing the capacity by 1 is good for saving memory because it uses the minimum amount of memory necessary but is bad for performance because more resize operations have to happen. On the contrary, doubling the capacity is bad for saving memory since it unnecessarily uses up extra memory if there are unoccupied indexes but is good for performance because less resize operations have to happen.
- **array** - The actual array, called *data*, in this example.

```
typedef struct vector  {
        int size;
        int capacity;
        int* data;
 } Vector;
```



A vector of integers with a size of 3 and a capacity of 5.

Linked List

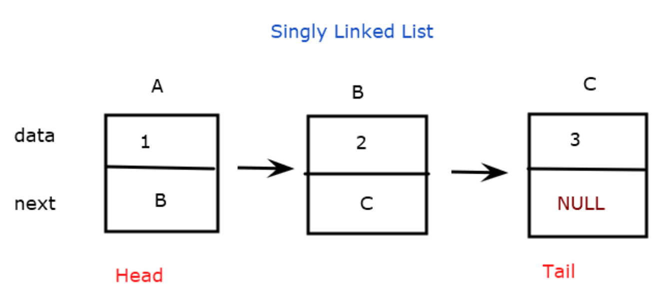
Linked List: A series of nodes all connected to each other via node pointers. It is a *list* of nodes that are all *linked* to each other hence the name *linked list*. The first node in the list is conventionally called the **head** and the last node in the list is conventionally called the **tail**. An implementation could keep track of both the head and the tail or just one of them. A linked list is one of the most fundamental data structures because it is also used to implement other data structures as will be seen later in this document.

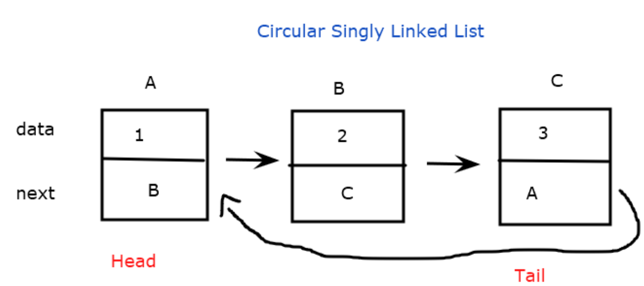
- **Singly Linked List** - Each node contains some data and a pointer to the next node conventionally called *next*. The *next* pointer of the tail node is set to NULL because there is no next node. This is how the end of the could be identified when traversing through the list.
- **Doubly Linked List** - Identical to a singly linked list with one addition - each node contains a pointer to the previous node conventionally called *prev*. The *prev* pointer of the first node is set to NULL because there is no previous node. This is how the beginning of the list could be identified when traversing through the list.
- **Circular Linked List:** A singly or doubly linked list where the final node is connected to the first node. The list can therefore be traversed like traversing the perimeter of a circle. In the case of a singly linked list, the *next* pointer of the tail node points to the head node. In the case of a doubly linked list, this is also true with the addition of the *prev* pointer of the head node pointing to the tail node.

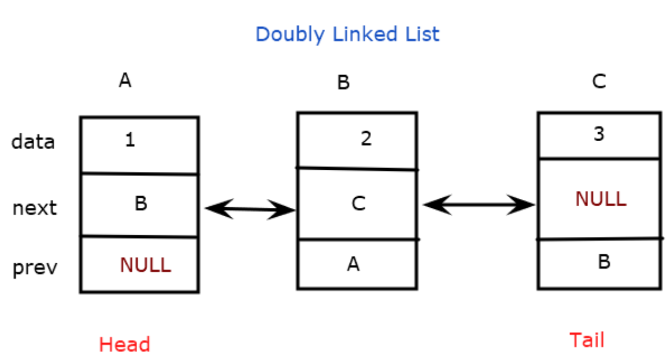
```
// singly linked list
typedef struct node Node;
struct node {
    int data;
    Node* next;
};
```

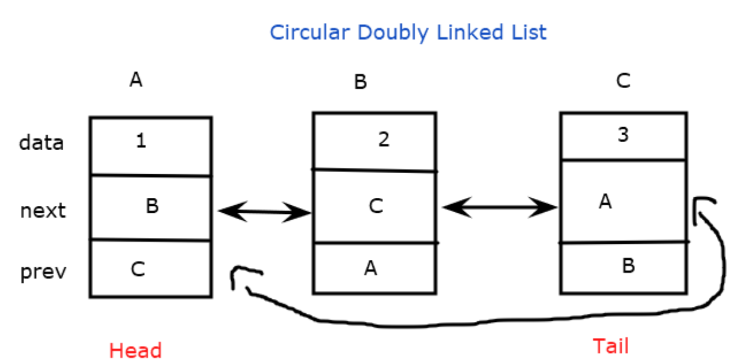
```
// doubly linked list
typedef struct node Node;
struct node {
    int data;
    Node* next;
    Node* prev;
};
```

```
// optional linked list structure
typedef struct list {
    Node* head;
    Node* tail;  // optional
} List;
```









Trees (General) and the Binary Search Tree

Tree: In general, a tree is a series of nodes starting at the root node where every node has a parent node (except for the root node) and child nodes. A tree is one of the most fundamental data structures because many data structures are implemented as trees as will be seen later in this document.

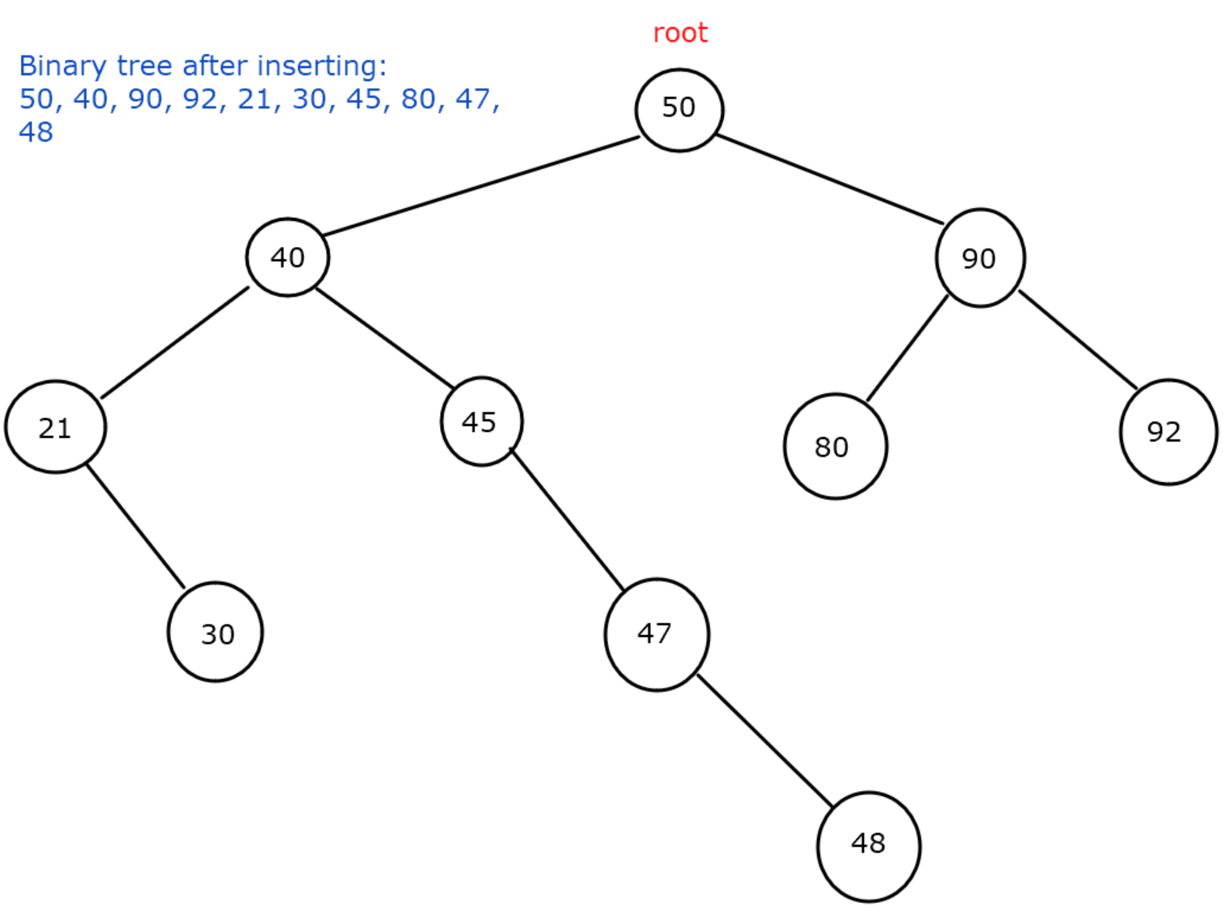
- A node can have the capability of having any number of children. Most commonly it has the capability of having two children: a left child and a right child.
- If a particular child node does not exist then the pointer to that child node is set to NULL.

Binary Search Tree: A tree adhering to the **binary tree property** - everything less than a node's data goes to the left and everything greater than a node's data goes to the right. A binary search tree is also referred to as a BST or just binary tree.

- The binary tree is considered the most basic of all trees. This is why it's being discussed first.
- The binary tree is not self-balancing. What is meant by self-balancing is discussed in the section on AVL trees.

```
// binary tree node structure
typedef struct node Node;
struct node {
	int data;
	Node* left;
	Node* right;
};
```

```
// optional binary tree structure
typedef struct binary_tree {
	Node* root;
} Binary_tree;
```



Inserting Into a Binary Tree

The best case scenario for inserting into the binary tree has a time complexity of $\Omega(\lg N)$. This is the case for when the tree is balanced meaning for any given node the difference between the depths of its left and right subtrees has a magnitude no greater than 1. However, the binary tree is not self-balancing so this is unlikely to happen and in the worst case scenario its time complexity is $O(N)$. This is because in the worst case scenario every item inserted is less than all items previously inserted creating a tree that is a diagonal line going down to the left or every time inserted is greater than all items previously inserted creating a tree that is a diagonal line going down to the right. In either case, all N nodes would have to be traversed for the item to reach its correct position to be inserted into at the bottom of the tree. In order to guarantee that the best case scenario always happens and have a worst case time complexity of $O(\lg N)$ an AVL tree must be used which is discussed later.

Tree Traversals

Tree Traversals: Tree traversals refer to the various ways by which a tree can be navigated. Different tree traversals will visit the nodes of a tree in different orders. The three basic traversals are described below.

Pre-order traversal: SLR (**self** left right)
- Each node visits itself first, then its left subtree and then its right subtree,
- Used for copying a tree.
- *Memorization technique*: pre - self comes before left and right, pre means before

In-order traversal: LSR (left **self** right)
- Each node visits its left subtree, then itself, then its right subtree.
- Used for print things in the tree in order
- *Memorization technique*: in - self is in between/in the middle of left and right

Post-order traversal: LRS (left right **self**)
- Each node visits its left subtree, its right subtree, then itself
- Used to delete a tree
- *Memorization technique*: post - self comes after left and right, post means after

Using the example of the binary tree on the previous page, the order in which each item from the tree would be printed using the three traversal techniques would be as follows
- *pre-order traversal:* 50, 40, 21, 30, 45, 47, 48, 90, 80, 92
- *in-order traversal:* 21, 30, 40, 45, 47, 48, 50, 80, 90, 92
- *post-order traversal:* 30, 21, 48, 47, 45, 40, 80, 92, 90, 50

-

Important Note

A vector (array), linked list, and tree are together the three most fundamental data structures because all other data structures are implemented using one of those three, and some data structures can be implemented using more than one of them. So, understanding all other data structures discussed hereafter requires first understanding these three fundamental data structures. In regards to trees specifically the binary tree was discussed at first because it's the simplest of all the trees so it's the easiest way to learn trees at first.

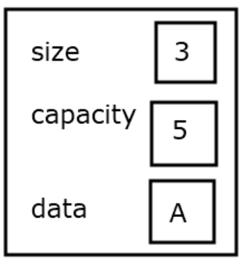
Stack

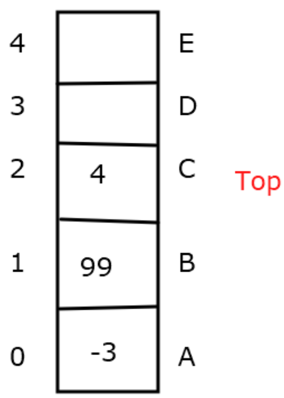
Stack: A series of items where only the top item can be accessed. Items are added to the top and removed from the top just like a stack of plates. A stack can be implemented using an array or linked list.

// array implementation
The structure is the same as the vector.

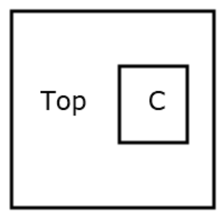
// linked list implementation
Use one of the linked list structures (page 4). Then create another structure like below

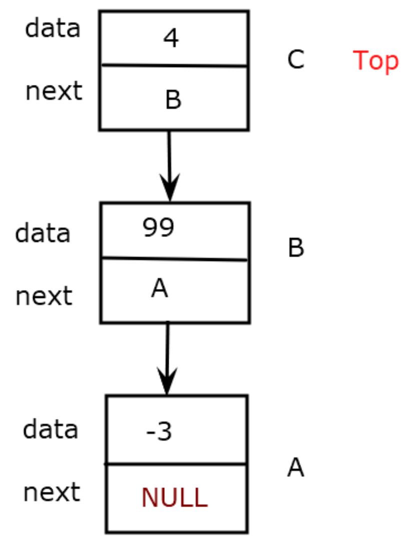
```
typedef struct stack {
    Node* top;
} Stack;
```





Vector implementation of a stack with a size of 3 and a capacity of 5





Singly linked list implementation of a stack a size of 3

FIFO (Regular) Queue

FIFO Queue: A queue where the arrangement of items is based on the order in which the items entered the queue. Items are added to the back and removed from the front. FIFO stands for *first-in-first-out*. The first item in is the first item out - in other words, when an item is removed from the queue at any given moment it was always the least recently added item. The FIFO queue can be casually referred to as a regular queue since it is what is traditionally thought of when the word queue comes to mind. An example of a FIFO queue is a line at a cash register.

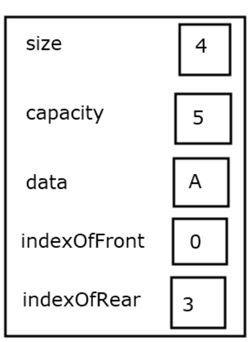
- **Front:** Contains the least recently added item and is where items are removed.
- **Back/Rear:** Contains the most recently added item and is where items are added.
- A FIFO queue can be implemented as an array or a linked list.
- The word *enqueue* is used to refer to adding to the queue and the words *dequeue* or *service* are used to refer to removing from the queue

```
// array implementation
typedef struct queue {
      int size;
      int capacity;
      int* data;
      int front;  // index of front
      int back;  // index of back
} Queue;
```

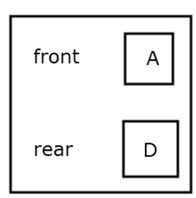
// linked list implementation
Use one of the linked list structures (page 4).
Then create another structure like below

```
typedef struct queue {
      Node* front;  // front/head of the list
      Node* back;  // back/tail of the list
} Queue;
```

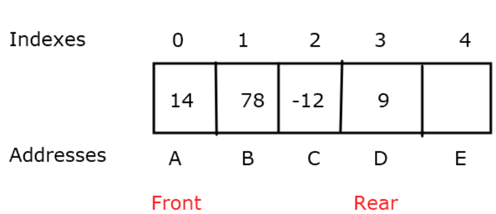
Note: The array implementation doesn't require tracking the back index. This is done by using some basic mathematical calculations involving modular arithmetic. Also, the array implementation should be a *wraparound queue* to improve efficiency and save memory. Using the example below, if the front were index 1 and the rear were index 4 then when adding a new item the rear would become index 0 instead of resizing the array and using index 5. If that were done it would waste memory since index 0 is unoccupied and it would decrease performance because a resize operation is unnecessarily happening.

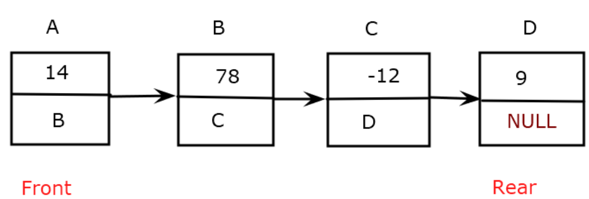


Vector implementation of a regular queue with a size of 4 and a capacity of 5



Singly linked list implementation of a regular queue with a size of 4





Priority Queue - Heap

Priority Queue: A queue where the arrangement of the items in the queue is based on their priority. The most common example of this is using a number as a priority. Higher numbers could have a higher priority or lower numbers could have a higher priority. Regardless, the arrangement has nothing to do with the order in which the items entered the queue.

A priority queue can be implemented in multiple ways and its definition is independent of any specific implementation. The first idea that may come to mind would be using an array and when an item is added it finds its appropriate place by starting at index 0 and then going through all of the indexes until it finds its place. Upon finding its place it is stored in that index and then all items of lower priority then have to be shifted over one by one. This would be a valid priority queue but it is a naive implementation because it's inefficient. A much more efficient implementation can be done using a **heap.**

Heap: A tree satisfying the **heap property**. The heap property has two versions so there are two versions of heaps. A **max heap** would satisfy the **max heap property** which means every node either has no children or is larger than its children if it has children, and a **min heap** would satisfy the **min heap** property which means every node either has no children or is smaller than its children if it has children. Notice how the definition of a heap never mentioned a priority queue nor does the definition of a priority queue mention a heap. A priority queue is a general term for a queue where the arrangement of the items in the queue is based on their priority and a heap is a specific data structure satisfying the heap property. Then, it happens to be that one way in which a priority queue can be implemented is with a heap but a priority queue and a heap are technically two separate things by definition. There are also other applications of heaps aside from priority queues such as the heap sort sorting algorithm.

- **Front:** Contains the item with the highest priority and is where items are removed. Unlike a FIFO queue the front item doesn't have to be the least recently added item since the order of items is based on their priority levels.
- **Back (not relevant):** In a heap, there is no back like a FIFO queue since there is no default position that an item goes to when it's added. When an item is added to the priority queue the position it goes to queue is based on its priority. For example, on the next page there is an example heap where 2 is the lowest priority item but notice how it is not at the "back" because there is no back.
- Heaps are trees but are in fact most easily implemented in code using arrays.
- Do not confuse the heap data structure with the area in memory also called the heap that is used with the malloc/calloc/realloc functions. They are two different things that have nothing to do with each other that happen to have the same name.
- In the following pages all of the diagrams must be viewed in conjunction with each other to gain a full understanding of heaps and how they keep items in the correct priority. The first diagram shows a standstill snapshot of a heap at some given moment. The following two diagrams demonstrate adding to the heap and removing from the heap.

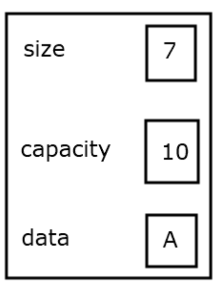
```c
// array implementation
typedef struct item {
        int data_item;
        int priority_level;
} Item;

typedef struct priority_queue {
        int size;
        int capacity;
        Item* data;
} Priority_queue;
```

Heap Visualization

Below is a standstill snapshot of a heap at some point in time. The example is a **max heap** where higher numbers have a higher priority level but recall there could also be a **min heap** where lower numbers have a higher priority level.

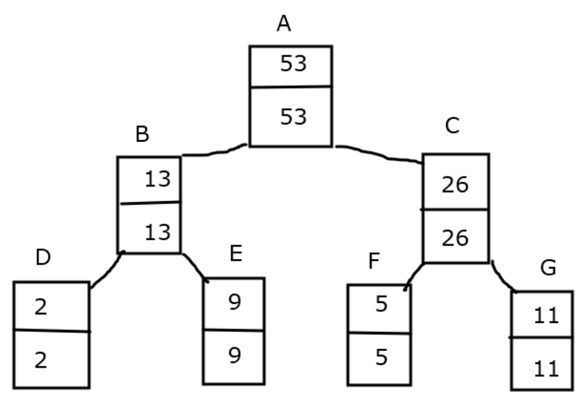
For the purposes of making the demonstration more simple, the data item and priority level are the same meaning 53 also has a priority level of 53. What matters though, is the priority level and not the data item. For example, if 53 had a priority level of 80 and 13 had a priority level of 90, then 13 would be at the front of the priority queue because 90 > 80.

<u>Heap Diagram 1:</u> Heap Visualization



Priority queue implemented as vector-implementation of a heap.

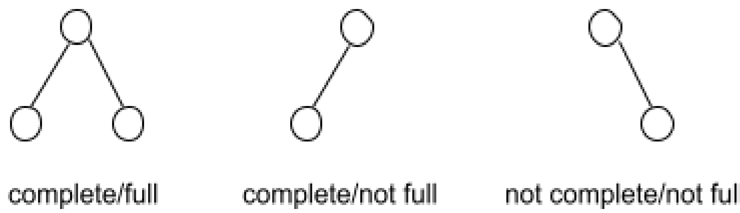
Indexes:	0	1	2	3	4	5	6	7	8	9
data_item	53	13	26	2	9	5	11			
priority_level	53	13	26	2	9	5	11			
Addresses	A	B	C	D	E	F	G	H	I	J



The corresponding conceptual visualization of the heap as a tree

Heap Properties

- Items are inserted top-to-bottom, left-to-right.
- Every node in the heap is bigger than its children if it has children for a max heap or every node is smaller than its children if it has children for a min heap .
- A heap is a **left complete tree**.
    - **Complete:** nodes are inserted in the proper order.
    - **Full:** <u>if</u> a node has children it has all of them that it can have.



Index Calculations For Heap As Array

When implementing the heap with an array use the following formulas to calculate the indexes of the parent, right child, and left child nodes where k = index of current item:

- parent index: $(k - 1)/2$ left child index: $2k + 1$ right child index: $2k + 2$

Take 13 on the previous page as an example which is at index 1:

parent index	(1 - 1) / 2	=	0	correct
left child index	2(1) + 1	=	3	correct
right child index	2(1) + 2	=	4	correct

Heap Time Complexities

Note: lg is the base 2 logarithm. It is so common in computer science it's just abbreviated as lg.

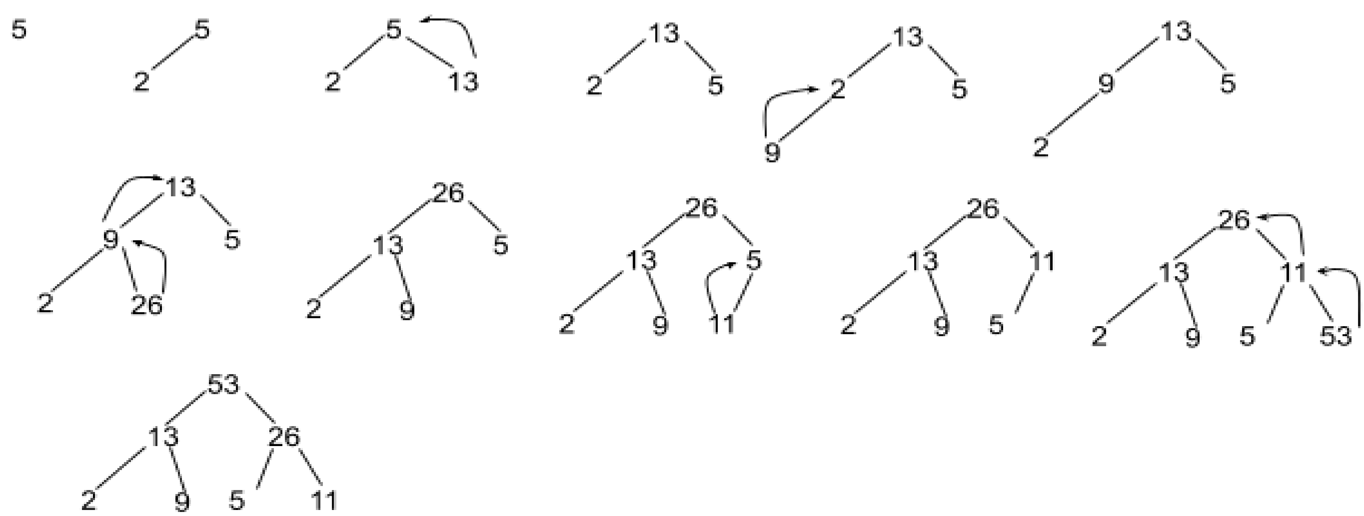
<u>Operation</u>	<u>Time</u>	
enqueue	O(lgN)	lgN fix-up operations will have to happen at most
service	O(lgN)	lgN fix-down operations will have to happen at most
merge	O(NlgN)	Remove an element from one heap (lgN)... Insert it into the next heap (lgN)... Do this for N elements in that one heap (N)... So that's 2NlgN which is NlgN because constants are ignored in time complexities since they're negligible
front	O(1)	Highest priority item is at the front (index 0 in an array, root in a tree)
empty:	O(1)	Just check if it's empty (size is 0 in an array, root is NULL in a tree)

Inserting Into A Heap

Insert/Enqueue: When a new item is inserted into a heap the position it goes to is based on its priority. This is achieved as follows - Insert the item at the next available position and then perform the **fix-up** operation until it's in the correct position. Fix-up means the item will swap with the parent if it has a higher priority than the parent.

- Cost of insert is O(lgN) (this really means floor(lgN) but it's referred to as lgN).
    - Ex: The cost to insert the 10th item is lg10 which is 3. It equals 3 because lg8 = 3 and lg16 = 4. This means lg10 is somewhere between 2^3 and 2^4 and is approximately $2^{3.3}$. The 3.3 gets rounded down to the next integer which is 3. This means that at most 3 fix-up operations will have to happen in the worst case to insert the 10th item.

<u>Heap Diagram 2:</u> Inserting into a heap - each number is also its own priority level.

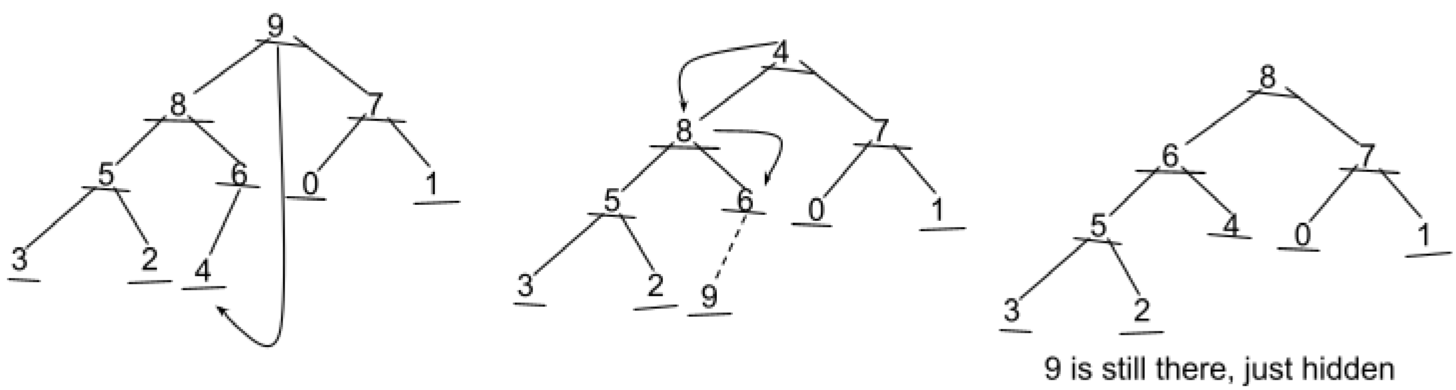


Removing From A Heap

Remove/Dequeue: Only the highest priority item can be removed from the heap. This is achieved as follows - Swap the first and the last element with each other. Then, remove the last element (which was previously the first element) from consideration and perform the **fix-down** operation on the first element (which was previously the last element) until it's in the correct position. Fix down means the following:

- If the item has a higher priority than both its children, it doesn't swap.
- If the item has a lower priority than one of its children, it swaps with the higher priority child.
- If the item has a lower priority than both of its children, it swaps with the child that has the higher priority amongst the two.

<u>Heap Diagram 3:</u> Removing from a heap - each number is also its own priority level.



Merging Two Heaps

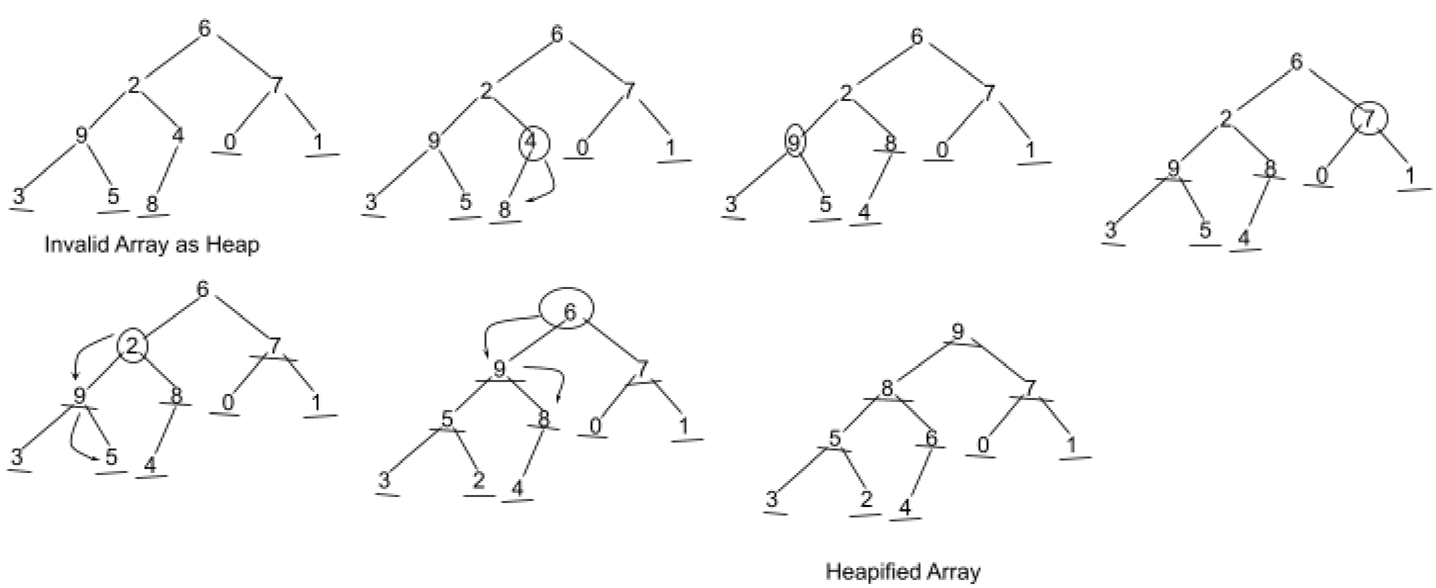
Merging Two Heaps: When two heaps are merged all of the items from one heap are added to the other heap which has a time complexity of O(NlgN). This is achieved as follows - Remove an element from one heap (lgN), insert it into the next heap (lgN), and do this for N elements in that one heap so that's 2NlgN which is NlgN because constants are ignored in time complexities since they're negligible.

- An O(1) constant time merge operation in a priority queue is achievable by using a different data structure to implement the priority queue called a **binomial queue/heap**. This data structure is discussed later.

Heapify

Heapify: Turn something that's not a heap (either a tree or an array) into a heap. The example below demonstrates how heapify works using a heap as a tree. If the heap were being implemented as an array then the same steps could be followed using the index calculations on page 12.

- The tree starts as an invalid heap with items randomly inserted.
- All the items in the lowest level are called **leaves** since they're at the end of the branch of the tree and have no children. The leaves are all **valid** heaps since they satisfy the heap property.
- Start at the first non-leaf which is the first potential non-heap. If the heap were an array the index of the first non-leaf is (size / 2 - 1). In this example, 4 is the first non-leaf node.
    - All of 4's children are heaps. Call fixdown on 4 since 8 is larger than 4.
    - Now 9 is the first potential non-heap.
- Examine 9 - 9 is a valid heap since it's larger than all of its children. 7 is the next potential non-heap.
- Examine 7 - 7 is a valid heap since it's larger than all of its children. 2 is the next potential non-heap.
- Examine 2 - It is not a valid heap since it's not larger than all of its children
    - Call fix down on 2 and fix it down until it's in the proper position. Now 6 is the next potential non-heap.
- Examine 6 - 6 is not a valid heap since it's not larger than all of its children.
    - Call fix down on 6. Once again, it fixes down until it's in the proper position
- Index 0 has been reached so the array has been heapified



Original Array Representation 6 2 7 9 4 0 1 3 5 8
Final Array Representation 9 8 7 5 6 0 1 3 2 4

More on heapify

- Heapify has a time complexity of O(N).
- Heapify is used in the heap sort sorting algorithm which has a time complexity of O(NlogN).
- Heapify is the most efficient way to create a heap. The other way a heap could be created would be by creating a heap priority queue as described on the previous pages where items are added in one at a time and then they’re fixed-up until they're in the correct position. This has a time complexity of O(NlgN) which is less efficient than O(N).
- Heapify is not used in heap priority queues because it would be less efficient than using the fix-up and fix-down operations. Recall that adding an item to and removing an item from a heap priority queue both have time complexities of O(lgN) due to the associated fix-up and fix-down operations. As an alternative to using the fix-up and fix-down operations, when an item is added to or removed from a heap priority queue the heapify operation could be performed to turn the heap priority queue back into a valid heap after adding a new item or removing an item. But, because heapify has a time complexity of O(N) this would result in the time complexity of adding and removing to also be O(N) instead of O(lgN) when using the fix-up and fix-down operations. O(N) is slower than O(lgN) so this would be less efficient.

Putting all of that together - Heapify has a time complexity of O(N), it is the most efficient way to create a heap, and it is used in the heap sort sorting algorithm. However, it is not used to create a heap priority queue because it would result in the add and remove operations both having a time complexity of O(N) instead of O(lgN).

Priority Queue - Binomial Queue/Heap

Binomial Queue: A type of priority queue that is a forest of trees where two trees of the same size don't exist and the size of each tree is always a power of 2. When an item is added or removed it must be checked if there are any trees of the same size. If this is true then the two trees of the same size combine and the higher priority root "wins". This means that they combine into one tree, the higher priority root becomes the root of the combined tree, and the lower priority root becomes a child of the higher priority root. Specifically, it will become the leftmost child.

- A binomial queue is also referred to as a binomial heap.

Binomial Heap Time Complexities

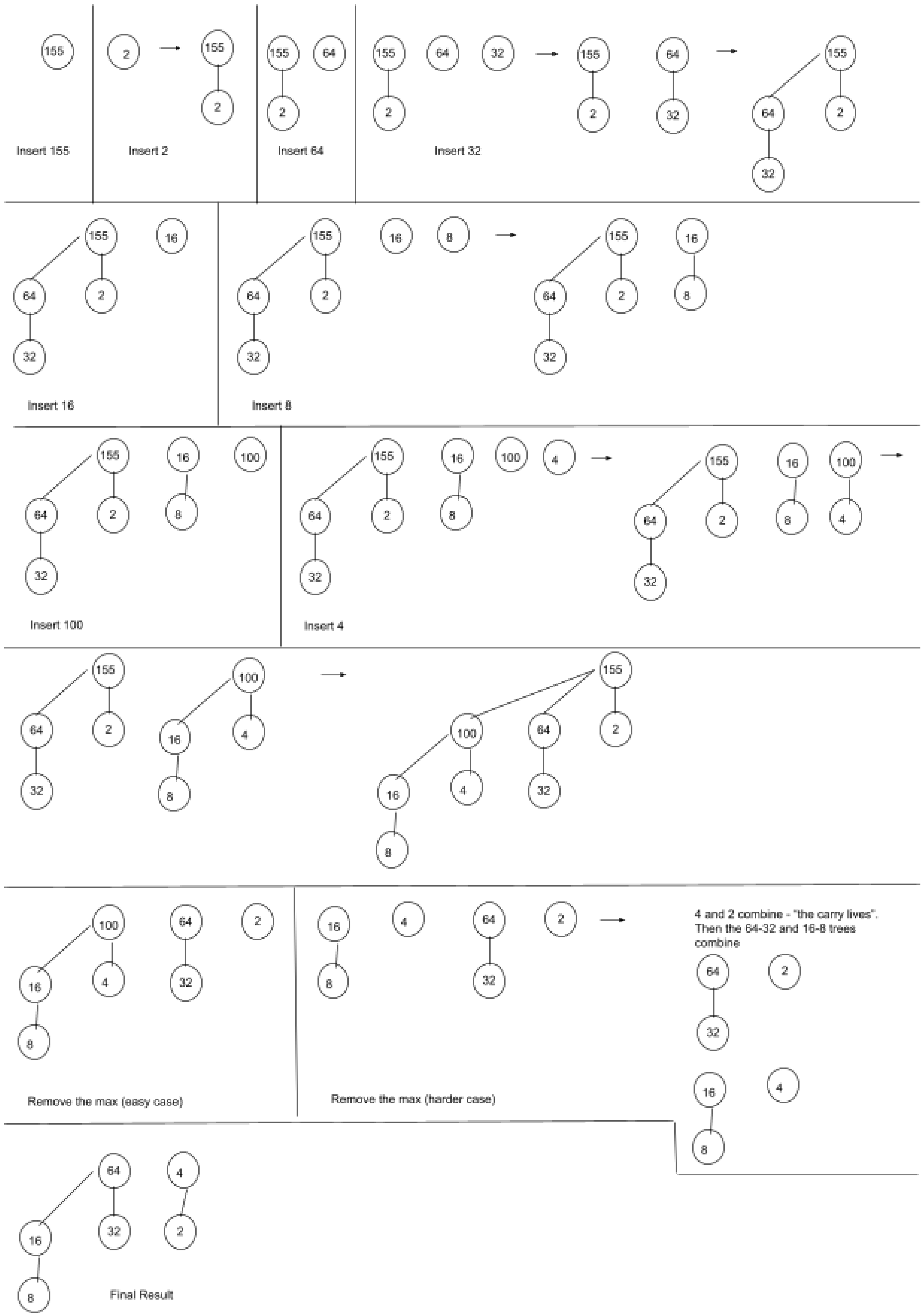
Operation	Time	
enqueue	O(lgN)	lgN merges will have to happen at most
service	O(lgN)	lgN merges will have to happen at most
merge	**O(1)**	The two trees merge and the higher priority root "wins"
front	O(1)	Highest priority item is at the root
empty:	O(1)	Just check if it's empty (root is NULL)

Notice how the time complexities are the same as a heap with the exception of the merge operation.

Binomial Heaps and Binary

- As said previously, each tree in the binomial queue is a size that is power of 2.
- Each tree is composed of the trees smaller than it. For example, a size 16 tree would have a root (+1) connected to an 8 tree (+8 = 9), a 4 tree (+4 = 13), a 2 tree (+2 = 15) and a 1 tree (+1 = 16).
- The heap itself can be represented as a binary number. For example, a binomial heap of size 8 would have one 8 tree which is 1000 in binary. So, the 8 tree fills up the 2^3 place and all the others are empty since they have no trees. A binomial heap of size 7 would be 111 since there would be a 1 tree, 2 tree, and a 7 tree but no 8 tree yet

An example of a binomial heap is shown on the next page.


Insert 155
Insert 2
Insert 64
Insert 32
Insert 16
Insert 8
Insert 100
Insert 4
Remove the max (easy case)
Remove the max (harder case)
4 and 2 combine - "the carry lives". Then the 64-32 and 16-8 trees combine
Final Result

AVL Tree

AVL tree: A self-balancing binary tree. Self-balancing means that for any given node the difference in magnitude of the depth of the left subtree and the right subtree is always less than 2. For example, if a node had no right child and it had a left child which also had a child, then the magnitude of the depth of its right subtree would be 0, and the magnitude of the depth of its left subtree would be 2. So, the difference is 0 - 2 = -2. Since |-2| = 2 which is not < 2, this would violate the self-balancing principle and the tree would need to rebalance. Rebalancing is done via *left rotations* and *right rotations*.

```
// AVL tree node structure
typedef struct node Node;
struct node {
	int data;
	Node* left;
	Node* right;
	int height;
};
```

```
// optional AVL tree structure
typedef struct avl_tree {
	Node* root;
} AVL_tree;
```

The following link provides code for an implementation of an AVL tree of integers. The logic of this code can be extrapolated to create an AVL tree of any data type.
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

A good AVL Tree Visualizer program can be found at this link:
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Magnitude: The AVL tree is based on the principle that if the magnitude/absolute value of (the depth of its right subtree) - (the depth of its left subtree) is greater than or equal to 2 then the tree is unbalanced and rotations have to happen.

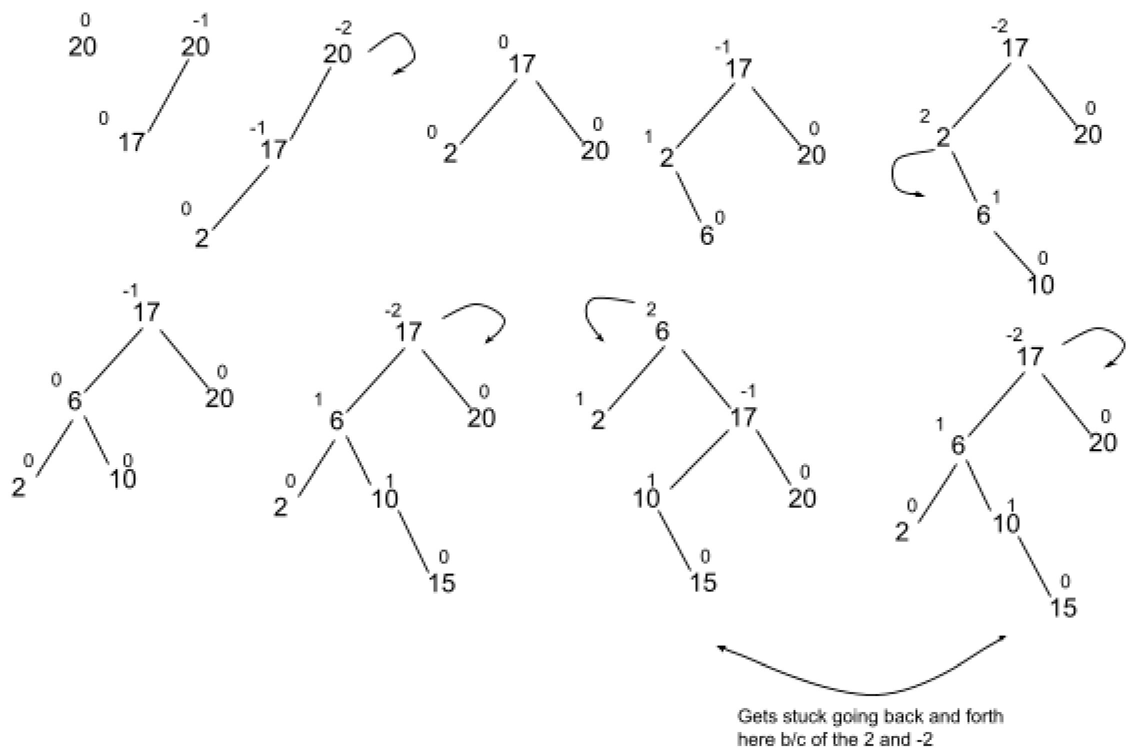
- A node in the tree is *left heavy* or *leans to the left* if the depth of its left subtree is greater than the depth of its right subtree which results in a balance factor of -1 or -2.
- A node in the tree is *right heavy* or *leans to the right* if the depth of its right subtree is greater than the depth of its left subtree which results in a balance factor of 1 or 2.

Left rotation: Right child of the previous root becomes the new root. Previous root becomes the left child of the new root. The left child of the right child (if it exists) becomes the right child of the previous root. The textbook calls this "right rotation" because the root rotates off the right child. Either name is fine just as long as its known what it's referring to

Right rotation: Left child of the previous root becomes the new root. Previous root becomes the right child of the new root. The right child of the left child (if it exists) becomes the left child of the previous root. The textbook calls this "left rotation" because the root rotates off the left child. Either name is fine just as long as its known what it's referring to

Self-Balancing

In regards to self-balancing there are four possible cases that can occur. Two of them require only one rotation and two of them require two rotations since they're more complicated. The four cases are summarized on the next page but first it must be demonstrated why two of the cases require two rotations instead of just one.



The tree gets stuck as seen in the above example when the children lean the opposite way of the parent.

- The parent is right heavy (2) but its right child is left heavy (-1) or the parent is left heavy (-2) but the left child is right heavy (1).
- In these two situations the double rotation as seen on the next page needs to happen.

The two simpler cases are when the parent leans the same way as the child.

- The parent is right heavy (2) and its right child is also right heavy (1) or the parent is left heavy (-2) and its left child is also left heavy (-1).
- In these two situations the double rotation does not have to happen.

Simple Cases - One Rotation

The parent and its child lean the same way.

Left-Left: The parent (root) is left heavy (-2) and its left child is left heavy (-1). Perform a right rotation on the parent.

- z is left heavy (-2) and y is left heavy (-1). Perform a right rotation on z.

```
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

Right-Right: The parent is right heavy (2) and its right child is right heavy (1). Perform a left rotation on the parent.

- z is right heavy (2) and y is right heavy (1). Perform a left rotation on z.

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

Complex Cases - Two Rotations

The parent and its child lean the opposite way.

Left-Right: The parent is left heavy (-2) and its left child is right heavy (1). Perform a left rotation on the left child of the root and then perform a right rotation on the root.

- z is left heavy (-2) and y is right heavy (1). Perform a left rotation on y and then perform a right rotation on z.

```
     z                               z                           x
    / \                            /   \                       /   \
   y   T4   Left Rotate (y)       x     T4  Right Rotate(z)   y      z
  / \       - - - - - - - - ->  /  \        - - - - - - - ->  / \    / \
T1    x                         y    T3                    T1   T2 T3   T4
     / \                       / \
   T2   T3                   T1   T2
```

Right-Left: The parent is right heavy (2) and its right child is left heavy (-1). Perform a right rotation on the right child of the root and then perform a left rotation on the root.

- z is right heavy (2) and y is left heavy (-1). Perform a right rotation on y and then perform a left rotation on z.

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->      /  \   - - - - - - - ->  / \    / \
   x   T4                      T2    y                 T1  T2  T3  T4
  / \                               /  \
T2   T3                           T3    T4
```

Hash Table

Hash Table: An array where a key in the form of a string is used as an index. This associates data with a string. Since a string can't literally be used as an index a hashing function converts the string into an index but this is hidden in the implementation.

The example below shows how a hash table can conceptually be thought of in conjunction with how it would actually get implemented. The ht_insert function would hash the strings "Mike" and "Sarah" to convert them into indexes in the array ages.

```
int ages[1000];        // an array to represent the ages of people

ages["Mike"] = 18;               // invalid C code but conceptually how it's thought of
ht_insert(ages, 1000, "Mike", 18);  // actual implementation

ages["Sarah"] = 25;              // invalid C code but conceptually how it's thought of
ht_insert(ages, 1000, "Sarah", 25);  // actual implementation
```

A hashing function can be written in an infinite number of ways but every hashing function has two things in common:

- It uses modular arithmetic. Since the hash table is an array, there is a restricted range of valid indexes. For example, if the hash table had a capacity of 1000 then the range of the valid indexes is [0, 999]. To guarantee that the hashing function returns a number in that range modular arithmetic must be used.
- It tries to avoid **collision** as much as possible. Collision is when the hashing function returns the same index for two or more unique keys. This can happen because a property of modular arithmetic is that two unique inputs can result in the same output. For example, 25 % 10 and 35 % 10 both result in the same remainder of 5. A good hashing function will be written so as to avoid collision as much as possible though it is not possible to avoid it entirely

The are two common ways of implementing hash tables that each deal with collision in their own way:

- **Open Addressing:** The hash table is just an array. If collision doesn't happen, the new item is inserted at the index returned by the hashing function. If collision does happen, the array is traversed until another unoccupied index is found. The traversal can be implemented in various ways. A linear probe is common which is when subsequent indexes are checked one-by-one. A quadratic probe is also common which involves squaring some number to calculate the next index to go to.
- **Separate Chaining:** The hash table is an array of linked lists. If collision doesn't happen, a new linked list is created at the index with that item. If collision does happen, the new item is added to the linked list. There is no need to traverse through the indexes to find another available one.

Hash tables are great because not only do they have a lot of utility (it is very common to want to associate data with a string), they are also an incredibly efficient data structure. On average, inserting, deleting, and accessing an item in a hash table is a constant time operation.

Sorting Algorithms

The definitions for the sorting algorithms listed below should be known. The official precise definitions cannot be shown since coming up with those definitions is exam material.

- bubble sort
- selection sort
- insertion sort
- shell sort
- heap sort
- quick sort

Other good sorting algorithms to know are:

- merge sort
- radix sort
- counting sort
- bucket sort

Note that bubble sort, selection sort, insertion sort, shell sort, heap sort, quick sort, and merge sort are all comparison based sorting algorithms whereas radix sort, bucket sort, and counting sort are all non-comparison based sorting algorithms. Though this seems counterintuitive, a non-comparison based sorting algorithm will sort elements without ever actually comparing them.

Quick sort, shell sort, heap sort, and merge sort all have an average runtime of $\Theta(N\log N)$ but in practice quick sort actually performs better than the other 3. Additionally, it has been proven mathematically that any comparison based sorting algorithm cannot be faster than NlogN. This means that quick sort is the fastest of all comparison based sorting algorithms. Interestingly, some of the non-comparison based sorting algorithms can achieve $\Theta(N)$ runtimes in particular circumstances so it is possible to get a linear runtime for a sorting algorithm. However, studying the actual mathematics behind the time complexities of sorting algorithms is something that is reserved for a more advanced algorithms course. For now, just know that comparison based sorting algorithms can’t have a faster runtime than NlogN, the fastest of the comparison based sorting algorithms is quick sort, and a sorting algorithm with a linear runtime is possible with a non-comparison based sorting algorithm.

Quicksort Demonstration

This is just a basic introduction to quicksort meant for someone who has never studied it before. There is a lot more information about it that is not discussed here especially in regards to different ways in which the algorithm can be implemented. This is something that would be studied in a more advanced algorithms course.

<u>Array:</u> [9 5 6 7 2 1 0 3 8 4]

<u>Goal:</u> all numbers less than pivot should be on the left of the pivot, all numbers larger than the pivot should be on the right of the pivot. Algorithm is described below

- Select a pivot
- put in left/right scanners. The left scanner starts at the pivot, the right scanner starts at the end (initially end of the array, when you're quick sorting the halves it's the last unsorted item going towards the right)
- move the right scanner <u>first</u> until it finds something that does not belong on the right/should be on the left/is smaller than the pivot (3 ways of saying the same thing). Or, go until it meets the left scanner.
- move the left scanner until it finds something that does not belong on the left/should be on the right/is larger than the pivot (3 ways of saying the same thing). Or, go until it meets the right scanner.
- Cases:
    - if the scanners do not meet, swap the items where the left scanner and right scanner are. Continue scanning.
    - if the scanners do meet, swap the item where they have met with the item in the pivot.

P = pivot, R = right scanner, L = left scanner.

Pivot Selection: Often the pivot will be randomly selected and the advantage of doing this is discussed after the demonstration. Here, the pivot is always arbitrarily selected as the leftmost item.

Start. Randomly selected the first item as pivot. Scanners go into appropriate positions

9	5	6	7	2	1	0	3	8	4
PL									R

R: 4 does not belong on the right,

L: there's nothing that does not belong on the left so scanners meet

9	5	6	7	2	1	0	3	8	4
P									LR

Swap pivot with scanners and rest/quick sort the halves. Technically the right half of 9 will be quicksorted but there's nothing there to the right so it just does the left half

4	5	6	7	2	1	0	3	8	**9**
PL								R	

R: 3 does not belong on the right pivot
L: 5 does not belong on the left of pivot

4	5	6	7	2	1	0	3	8	**9**
P	L						R		

Swap items at scanners and continue scanning

4	3	6	7	2	1	0	5	8	**9**
P	L						R		

R: 0 does not belong on the right pivot
L: 6 does not belong on the left of pivot

4	3	6	7	2	1	0	5	8	**9**
P		L				R			

Swap items at scanners and continue scanning

4	3	0	7	2	1	6	5	8	**9**
P		L				R			

R: 1 does not belong on the right pivot
L: 7 does not belong on the left of pivot

4	3	0	7	2	1	6	5	8	**9**
P			L		R				

Swap items at scanners and continue scanning

4	3	0	1	2	7	6	5	8	**9**
P			L		R				

R: 2 does not belong on the right pivot
L: meets right scanner.

4	3	0	1	2	7	6	5	8	**9**
P				LR					

Swap with pivot. 4 is now sorted. Quick sort the halves

2	3	0	1	**4**	7	6	5	8	**9**
P				LR					

Left half of 4 (could've done right half, doesn't matter)

2	3	0	1	**4**	7	6	5	8	**9**
PL			R						

R: 1 does not belong on the right pivot
L: 3 does not belong on the left of pivot

2	3	0	1	**4**	7	6	5	8	**9**
P	L		R						

Swap items at scanners and continue scanning

2	1	0	3	**4**	7	6	5	8	**9**
P	L		R						

R: 0 does not belong on the right of the pivot
L: meets right scanner

2	1	0	3	**4**	7	6	5	8	**9**
P		LR							

Swap with pivot. 2 is now sorted. Quick sort the halves

0	1	**2**	3	**4**	7	6	5	8	**9**
P				LR					

Left half of 2 (could've done right half, doesn't matter)

0	1	**2**	3	**4**	7	6	5	8	**9**
PL	R								

R: 1 is fine, moves on to 0. meets left scanner
L: nothing

0	1	**2**	3	**4**	7	6	5	8	**9**
PLR									

Swap with pivot (swaps with itself so nothing changes but 0 is now considered sorted). Quick sort halves

0	1	**2**	3	**4**	7	6	5	8	**9**
	PLR								

Left half of 0. Nothing there. Right half of 0. Size is 1 so already sorted (this can be coded so it recognizes when the size is 1. It is based on if the scanners met each other). 1 is now sorted. Left half of 2 is now sorted
Right half of 2

0	**1**	**2**	3	**4**	7	6	5	8	**9**
			PLR						

Size is 1 so 3 is now sorted

0	**1**	**2**	**3**	**4**	7	6	5	8	**9**
			PLR						

Left half of 4 now sorted. Quick sort right half of 4

0	**1**	**2**	**3**	**4**	7	6	5	8	**9**
					PL			R	

R: 5 does not belong on the right of the pivot
L: meets right scanner

0	**1**	**2**	**3**	**4**	7	6	5	8	**9**
					P		LR		

Swap with pivot. 7 is now sorted. Quick sort the halves

0	**1**	**2**	**3**	**4**	5	6	**7**	8	**9**
					P		LR		

Left half of 7

0	**1**	**2**	**3**	**4**	5	6	**7**	8	**9**
					PL	R			

R: 6 is fine, meets left scanner at 5
L: meets right scanner

0	**1**	**2**	**3**	**4**	**5**	6	**7**	8	**9**
					PLR				

Swap with pivot (swaps with itself so nothing changes but 5 is now considered sorted). Quick sort halves.

Left half of 5. Nothing there

Right half of 5:

0	**1**	**2**	**3**	**4**	**5**	6	**7**	8	**9**
						PLR			

6 is size 1. It's now sorted

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	8	**9**
						PLR			

Right half of 7

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	8	**9**
								PLR	

8 is size 1. It's now sorted.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

<u>Done.</u>

Extra Notes On Quicksort

Worst case scenario: $O(N^2)$

The point of quicksort is you swap something to be put in the middle so everything on the left and right is on the proper side of the pivot. When the pivot always swaps to the end (like 9 in the previous example) and that happens every time, that is bad. However, this is a problem that's easily fixed.

Fix:

The underlying problem with the worst case scenario involves picking a bad pivot. So, if you randomly select a pivot every time this won't happen. Technically, it could lead to worst case performance but the odds are so low it doesn't happen in practice. For example, for an array of 100 numbers the worst case scenario odds would be $\frac{2}{100} \cdot \frac{2}{99} \cdot \frac{2}{98} \cdot \frac{2}{97} \ldots = \frac{2^{100}}{100!}$

The numerator is 2 because every time there are two possible worst case indexes (the first and the last). The denominator decreases by 1 because every time a new pivot is selected the size of the subarray being sorted has decreased by 1 due to the previous number being sorted.

2^{100} is a very large number and is approximately 1.3×10^{30}.

However, 100! is so much larger and is approximately 9.3×10^{157}.

As a fraction this is $\frac{1.3\, x\, 10^{30}}{9.3\, x\, 10^{157}}$.

If you ignore the mantissas since they're negligible then this is the fraction $\frac{10^{30}}{10^{157}} = \frac{1}{10^{127}}$.

This number is so small that although it is technically possible for the worst case scenario to occur in that it has a non-zero probability, the probability is so low it will never realistically happen in practice. Additionally, the probability will get smaller and smaller as the amount of numbers being sorted increases. For example, if the size is increased from 100 to 150 which isn't that much, the probability becomes $\frac{1}{10^{202}}$. Now imagine if there were 10,000 or 1,000,000 numbers being sorted. The probability would continue to become astronomically lower and lower.

Final note on pivot selection: In addition to randomly selecting a pivot, another way to pick a pivot which is slightly better would be to randomly select 3 elements and then pick the median of them.

Time Complexity (General Discussion)

Time complexity is a type of computational complexity that calculates how much time a particular algorithm takes to execute as a function of some input. It's independent of the study of data structures and can be used to calculate the running time of any algorithm such as sorting algorithms. With data structures specifically the input is the amount of items that are in the data structure traditionally referred to as N, and the algorithms are the various operations associated with the data structure. So, the time complexities of a particular data structure calculate how much time the operations performed on the data structure take in relation to the amount of items in the data structure. In practice, there are many other factors aside from N that affect an algorithm's running time when it's actually implemented on a real computer. This includes the features of the machine it's running on such as the hardware of the machine, the software configurations of the machine, cache performance, and space complexity among others. Time complexity for data structures is independent of all of those other factors and studies the running time of an algorithm purely from an abstract mathematical perspective in relation to N.

The time complexities below are some of the most common Big-O time complexities which are how long a particular operation takes in the worst case. It is important to think about time complexities from the perspective of the worst case scenarios because achieving the best or average case scenarios can't be relied upon. In practice, various algorithms can avoid the worst case scenarios by implementing them in a certain way such that the probability of the worst case scenario is so low it won't realistically happen. An example of this was seen when discussing the quicksort sorting algorithm. However, much of the time avoiding the worst case scenario can't be relied upon especially with data structures.

Big-O	Name	
O(c)	constant time where *c* is some positive constant	*Fastest*
O(lgN)	logarithmic time (base 2 logarithm)	…
O(N)	linear time (polynomial time, c = 1)	…
O(NlgN)	linear × logarithmic time	…
$O(N^2)$	quadratic time (polynomial time, c = 2)	…
$O(N^c)$	polynomial time (all other possible versions with increasing values of c)	…
$O(2^N)$	exponential time (c = 2)	…
$O(c^N)$	exponential time (all other possible versions with increasing values of c)	*Slowest*

The study of time complexity is about what happens as N gets larger (goes out to infinity). For example, exponential time is slower than polynomial time because as N gets larger there will come some point where the exponential time equation always has a larger result than the polynomial time equation. However, this doesn't mean that there aren't smaller values of N for which this won't be true. For example, compare exponential time $O(2^N)$ to polynomial time $O(N^4)$. For all values of N starting at 1 up to 15, N^4 is greater than 2^N. At N = 16 they are equal. Then starting at N = 17 and for all values thereafter 2^N is greater than N^4. So, this is why $O(N^4) < O(2^N)$. What matters is what happens as N gets larger. For all of these time complexities they can be visualized on a graph with the x-axis representing N and the y-axis representing the time. As N gets larger it could be clearly seen that O(c) < O(lgN) < O(N)...

An O(c) **constant time** operation means the amount of time an operation takes in the worst case is independent of N. In other words, the operation always takes the same amount of time regardless of how many items are in the data structure. On a graph this is the equation T(N) = *c* where *c* is some positive constant. It could be any number but traditionally a 1 is put there like O(1) for convenience. Another way to think of this is that the operation will take *c* units of time. So, if *c* = 1 the operation always takes 1 unit of time or if *c* = 5 the operation always takes 5 units of time. The consequence of this is that constant time operations have the fastest running time in comparison to all of the others because as N gets larger the runtime of the operation does not increase. Notice how the definition doesn't actually have anything to do with speed. It just very specifically says that N does not affect the runtime. Hypothetically, there could be a constant time operation that takes a long time to complete like if *c* = 100 million. However, in practice constant time operations are generally fast and *c* would not be that high, and even if hypothetically it were that high eventually the results of all the other time complexities will surpass it as N gets larger.

An O(lgN) **logarithmic time** operation means the amount of time an operation takes in the worst case is logarithmically proportional to N (base 2 logarithm). On a graph this is the equation T(N) = lgN. Note that it technically means floor(lgN) where the actual result is rounded down to the next integer. Another way to think of this is that the operation will take lgN units of time to complete. For example, if N is 10 then that is lg10 which equals 3. It equals 3 because lg8 = 3 and lg16 = 4. This means lg10 is somewhere between 2^3 and 2^4 and is approximately $2^{3.3}$. The 3.3 gets rounded down to the next integer which is 3.

An O(N) **linear time** operation means the amount of time an operation takes in the worst case is linearly proportional to N. On a graph this is the equation T(N) = N. Another way to think of this is that the operation will take N units of time to complete.

An O(NlgN) operation means the amount of time an operation takes in the worst case is proportional to N in the form of the equation NlgN which is multiplying N by lgN so it combines linear and logarithmic time. On a graph this is the equation T(N) = NlgN. Another way to think of this is that the operation will take NlgN units of time to complete.

An O(N^2) **quadratic time** operation means the amount of time an operation takes in the worst case is quadratically proportional to N. On a graph this is the equation T(N) = N^2. Another way to think of this is that the operation will take N^2 units of time.

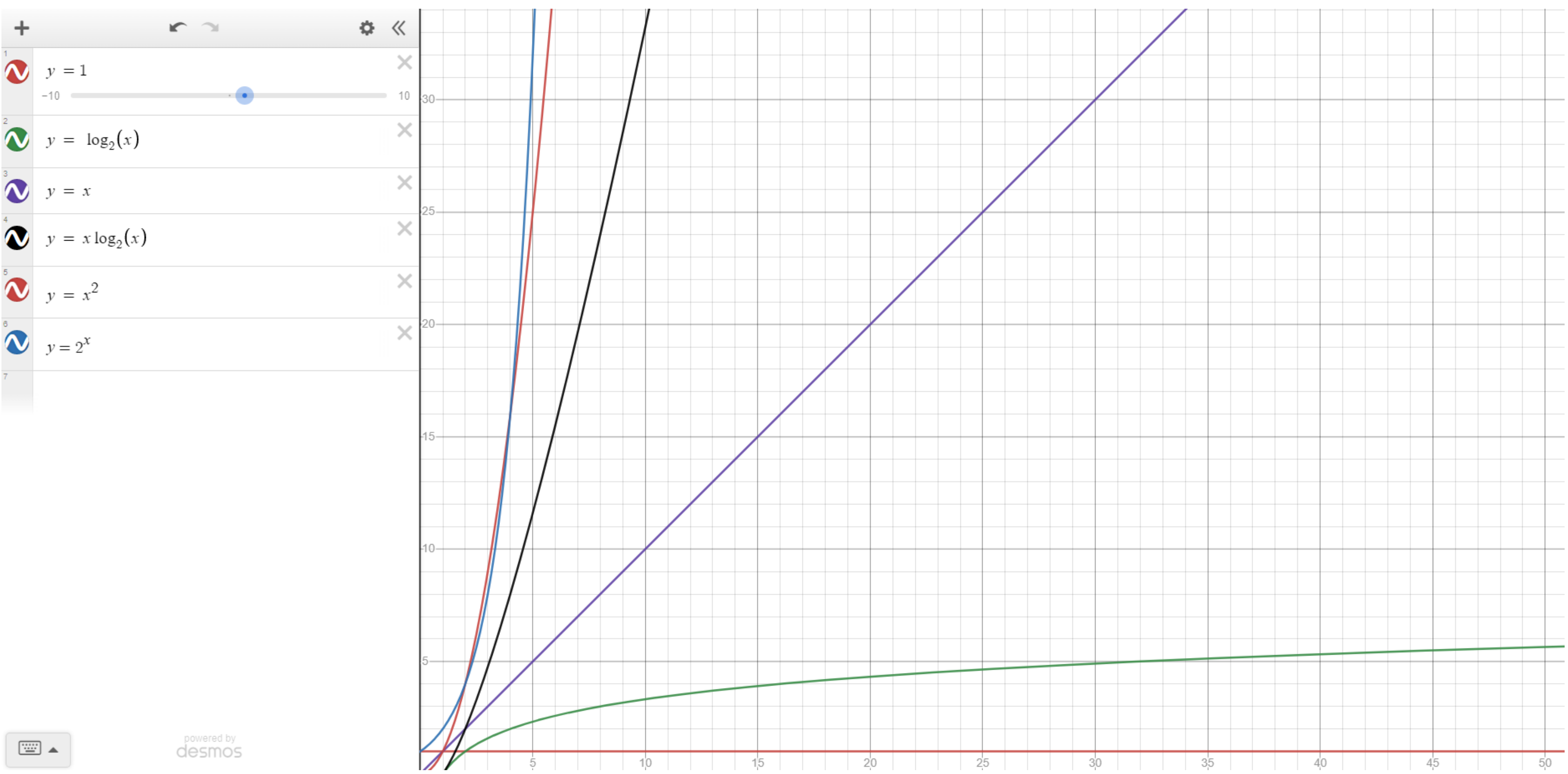
An O(N^c) **polynomial time** operation means the amount of time an operation takes in the worst case is polynomially proportional to a polynomial of degree *c* where *c* is some positive constant. On a graph this is the equation T(N) = N^c. Another way to think of this is that the operation will take N^c units of time. Linear and quadratic are both versions of polynomial time. The runtime increases as *c* increases meaning $N^2 < N^3 < N^4$…

An $O(c^N)$ **exponential time** operation means the amount of time an operation takes in the worst case is exponentially proportional to some exponential equation with base *c* where *c* is some positive constant. On a graph this is the equation $T(N) = c^N$. Another way to think of this is that the operation will take c^N units of time. Like with polynomial time the runtime increases as *c* increases meaning $2^N < 3^N < 4^N$…

Consider the example where N = 10 and it can be seen how the times compare to each other.

O(1)	Some constant amount of time.	Doesn't increase as N increases	*Fastest*
O(lgN)	3 units of time because lg10 = 3	Increases as N increases	…
O(N)	10 units of time	Increases as N increases	…
O(NlgN)	30 units of time because 10 x lg10 = 30	Increases as N increases	…
$O(N^2)$	100 units of time because $10^2 = 100$	Increases as N increases	…
$O(2^N)$	1024 units of time because $2^{10} = 1024$	Increases as N increases	*Slowest*

They can also be seen on a graph so it can be seen what happens as N gets larger. As is expected, O(1) < O(lgN) < O(N)... as N increases.



Data Structure Time Complexities (Big-O/Worst Case)

This is not a comprehensive overview for every possible operation in every possible situation. It just gives the time complexities of some of the most common operations.

Notes: The following are all O(1) constant time operations: accessing an index in an array, accessing the head/tail node in a linked list (if they're kept track of), and accessing the root node of a tree. This is being mentioned here so it doesn't have to be repeatedly stated.

Vector (unordered) - same as an unordered array

Common Operations	DS specific name	Location	Time
insert	vector_push_back	back	O(1)
remove	vector_pop_back	back	O(1)
access	vector_at	any index	O(1)
search (specific item)		could be anywhere	O(N)

- *insert:* insert into the next available index which is index [size]
- *remove:* remove the item at the back which is index [size - 1]
- *access:* any index
- *search:* if the item didn't exist or was at the back then all N items have to be traversed

Linked List (unordered, access to head only)

Common Operations	DS specific name	Location	Time
insert	head_insert	head	O(1)
insert	tail_insert	tail	O(N)
remove		head	O(1)
remove		tail	O(N)
remove (specific item)		could be anywhere	O(N)
access		head	O(1)
access		tail	O(N)
access (specific item)		could be anywhere	O(N)
search (specific item)		could be anywhere	O(N)

- *insert (head):* the head node is kept track
- *insert (tail):* the tail node is not kept track of so all N nodes have to be traversed
- *remove (head):* the head node is kept track of
- *remove (tail):* the tail node is not kept track of so all N nodes have to be traversed
- *remove (specific item):* if the item didn't exist or was the tail then all N items have to be traversed
- *access(head):* the head node is kept track of
- *access (tail):* the tail node is not kept track of so all N nodes have to be traversed
- *access (specific item):* if the item didn't exist or was the tail then all N items have to be traversed

Binary Search Tree

Common Operations	DS specific name	Location	Time
insert		could end up anywhere	O(N)
remove		could be anywhere	O(N)
search (some specific item)		could be anywhere	O(N)

- For all of the worst case scenarios there is either a tree that is a diagonal line going down to the left because all items inserted have been less than all items previously inserted (case 1) or there is a tree that is a diagonal line going down to the right because all items inserted have been greater than all items previously inserted.
- *insert:* there is a *case 1/case 2* tree and the item being inserted is less than/greater than all items previously inserted - in either case, all N nodes would have to be traversed for the item to reach its correct position to be inserted into at the bottom of the tree
- *remove:* there is a *case 1/case 2* tree and the item being removed is the smallest/largest item in the tree - in either case, all N nodes would have to be traversed to reach it
- *search:* there is a *case 1/case 2* tree and the item being searched for is the smallest/largest item in the tree - in either case, all N nodes would have to be traversed to reach it

Stack

Common Operations	DS specific name	Location	Time
insert	stack_push	top	O(1)
remove	stack_pop	top	O(1)
access	stack_top	top	O(1)

- *insert:* in a linked list-based implementation this would be inserting at the head and in an array-based implementation this would be inserting at the the top which is index [size]
- *remove/access:* in a linked list-based implementation this would be removing/accessing the head and in an array-based implementation this would be removing/accessing the top which is index [size - 1]

FIFO Queue

Common Operations	DS specific name	Location	Time
insert	queue_enqueue	back	O(1)
remove	queue_dequeue/service	front	O(1)
access	queue_front	front	O(1)

- *insert:* in a linked list-based implementation this would be inserting at the tail and in an array-based implementation this would be inserting at the back which is index [back + 1] or [0] depending on the situation in a wraparound queue
- *remove/access:* in a linked list-based implementation this would be removing/accessing the head and in an array-based implementation this would be removing/accessing the front which is index [front]

Priority Queue - Heap

Commons Operations	DS specific name	Location	Time
insert	pqueue_enqueue	based on priority	O(lgN)
remove (highest priority)	pqueue_service	front	O(lgN)
access (highest priority)	pqueue_front	front	O(1)
merge			O(NlgN)

- *insert:* there will be at most lgN fix-up operations that have to happen
- *remove:* there will be at most lgN fix-down operations that have to happen
- *access:* in a tree-based implementation the highest priority item is always at the root node and in an array-based implementation the highest priority item is always at the front which is index [0]
- *merge:* remove an element from one heap (lgN), insert it into the next heap (lgN), and do this for N elements in that one heap so it's 2NlgN which is NlgN because constants are ignored in time complexities since they're negligible

Priority Queue - Binomial Queue/Heap

Commons Operations	DS specific name	Location	Time
insert	bqueue_enqueue	based on priority	O(lgN)
remove (highest priority)	bqueue_service	front	O(lgN)
access (highest priority)	bqueue_front	front	O(1)
merge			O(1)

- *insert:* there will be at most lgN tree merges
- *remove:* there will be at most lgN tree merges
- *access:* the highest priority item is always at the root node
- *merge:* the items at the root nodes of each tree are compared - the higher priority item becomes the root in the merged tree and the lower priority item becomes its child and this process always takes the same amount of time regardless of N

AVL Tree

Common Operations	DS specific name	Location	Time
insert		could end up anywhere	O(lgN)
remove		could be anywhere	O(lgN)
search (some specific item)		could be anywhere	O(lgN)

- *insert:* at most lgN nodes will have to be traversed for the item to find its correct position in the tree
- *remove:* at most lgN nodes will have to be traversed to find the item being removed
- *remove:* at most lgN nodes will have to be traversed to find the item being searched for

Hash Table - The worst case scenarios can easily be avoided with a good implementation so the average time complexities are also shown since that is what usually happens in practice.

Common Operations	Location	Time (Average)	Time (Worst)
insert	could end up anywhere	$\Theta(1)$	$O(N)$
remove	could be anywhere	$\Theta(1)$	$O(N)$
search (some specific item)	could be anywhere	$\Theta(1)$	$O(N)$

- Discussing why the average case time complexities are constant time and the worst case time complexities are linear time requires a much more in depth discussion on hash tables, and within that discussion there would have to be two sub-discussions - one for open addressing and one for separate chaining. This is beyond the scope of this document. For now, just know that if the hash table implementation avoids collision as much as possible then in practice it is reasonable to achieve constant time operations.

Sorting Algorithm Time Complexities

Sorting Algorithm	Best	Average	Worst
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$
Shell Sort	$\Omega(N)$	$\Theta(N\log N)$	$O(N\log N)$
Heap Sort	$\Omega(N\log N)$	$\Theta(N\log N)$	$O(N\log N)$
Quick Sort	$\Omega(N\log N)$	$\Theta(N\log N)$	$O(N^2)$
Merge Sort	$\Omega(N\log N)$	$\Theta(N\log N)$	$O(N\log N)$