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. In general, one data structure is not better than another. Rather, different data structures have different properties, and their utility is situation dependent. Additionally, in general there are array-based data structures and node-based data structures, and some data structures can be represented in both ways. The main purpose of this document is to give conceptual explanations and provide visual representations of many of the most important and fundamental data structures. In addition to discussing data structures, this document also contains some information about sorting algorithms especially in regards to quick sort.

Notes:

- The C programming 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 diagrams shown in this document 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 in this document will use integers.
- For the data structures that have both array-based and linked-list based implementations, there is no discussion on the respective advantages and disadvantages of each option since that is beyond the scope of this document. Speaking very generally, 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 much more discussion to be had.
- Time complexities for data structures are discussed in this document at times. There are three major types: O(N), Ω(N), and Θ(N). O(N) is the worst case runtime and is referred to as Big-O which stands for Big-Ordnung. Ω(N) is the best case runtime and is referred to as Big-Ω which stands for Big-Omega. Θ(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) would mean N provides an asymptotic upper bound for an operation and T(N) = Ω(N) means N provides an asymptotic lower bound for an operation. If T(N) = Ω(N) and T(N) = O(N) then T(N) = Θ(N) which means N 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 since it's a common misconception.
- There are also two other time complexities: o(N) and ω(N) referred to as little-o and little-omega. Like with Ω(N) and Θ(N), these are not discussed in this document and would be studied in a more advanced algorithms course as well.

Table of Contents

Vectors

Vector: An array with a non-fixed capacity. This is different from an array with a fixed capacity like:

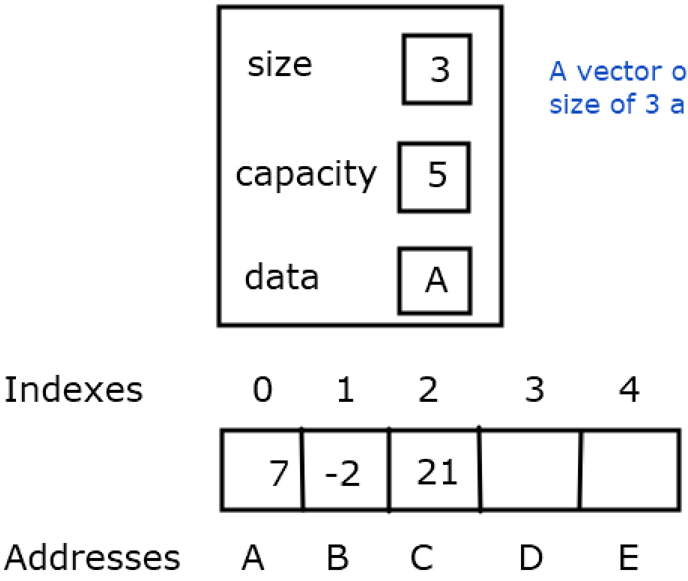
```
    int a[10];
```

That array has a capacity of 10 elements and its valid range of indexes is [0, 9]. This can never change. Unlike that array, a vector can dynamically increase its capacity during runtime.

A vector has the following components:

- **size** - The amount of items in the array (initially 0). The index of the next available position is always [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 first be performed. The array can be resized in any way to include doubling the capacity or only increasing the capacity by 1. It is a tradeoff between saving memory (increasing by 1 always uses the minimal memory necessary) and performance (increasing by 1 means more 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 Lists

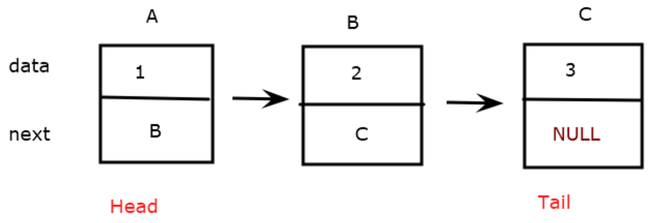
Linked List: A series of nodes all connected to each other via node pointers. It is a *list* of nodes that are all *linked* to each other, hence the name *linked list*. The first node in the list is conventionally called the **head**, and the last node in the list is conventionally called the **tail**. An implementation could keep track of both the head and the tail or just one of them.

- **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 in the list 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.

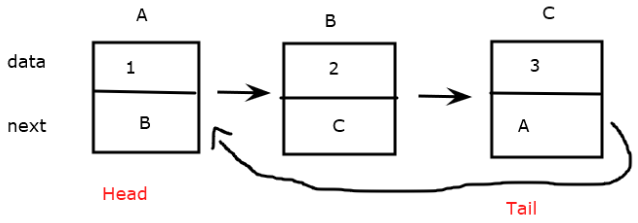
```
// singly linked list
typedef struct node Node;
struct node {
	int data;
	Node* next;
};
```

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

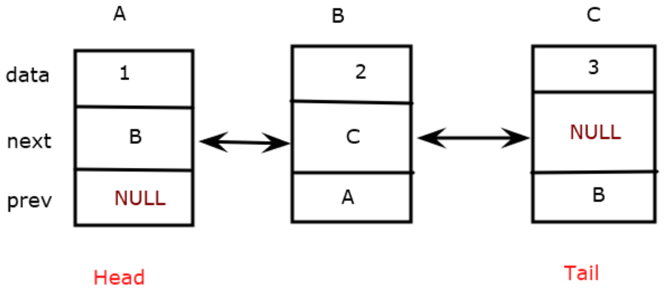
Singly Linked List



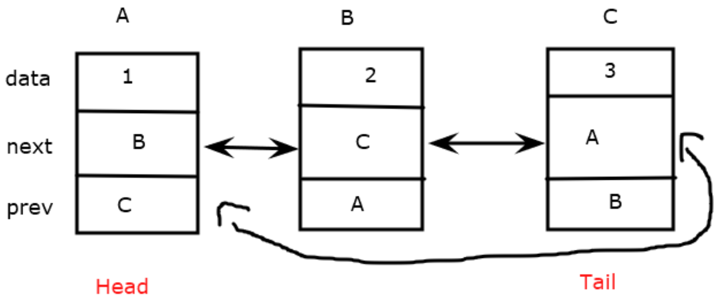
Circular Singly Linked List



Doubly Linked List



Circular Doubly Linked List



Stacks

Stack: A series of items where items are added to the top and removed from the top such as a stack of plates. A stack can be implemented using an array or linked list.

// array implementation

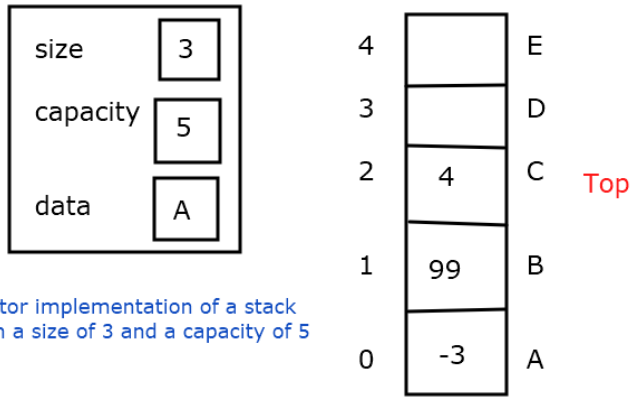
The structure is the same as the vector.

// linked list implementation

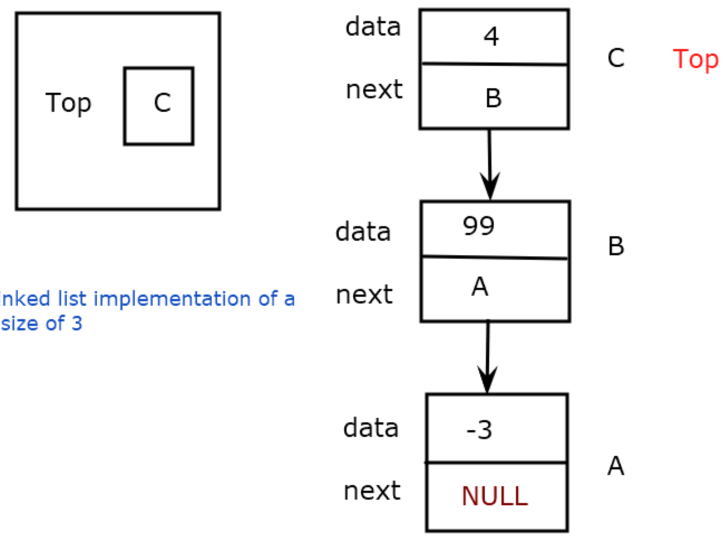
Use one of the linked list structures on page 4.

Then create an additional 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) Queues

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 though there are other types of queues (discussed later).

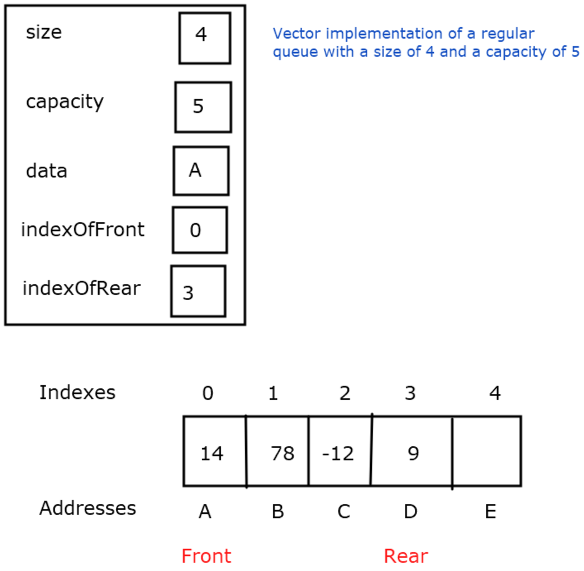
- **Front:** Contains the least recently added item and is where items are removed.
- **Rear:** Contains the most recently added item and is where items are added.
- An example of a FIFO queue is a line at a cash register.
- A regular queue can be implemented as an array or a linked list.
- The word *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

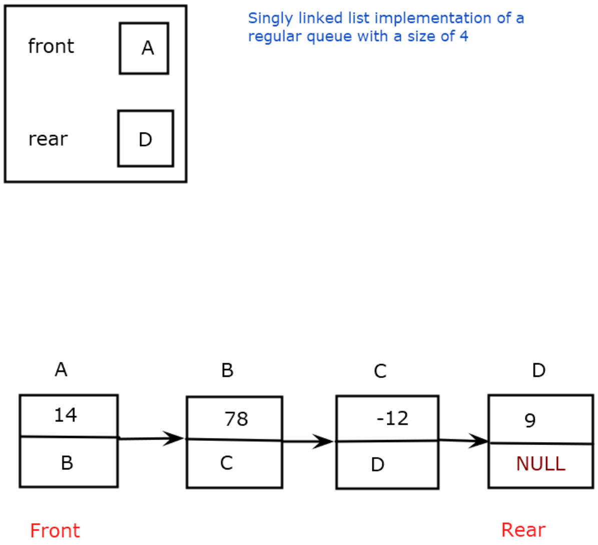
```
// array implementation
typedef struct queue {
        int size;
        int capacity;
        int* data;
        int indexOfFront;
        int indexOfRear;
} Queue;
```

// linked list implementation
Use one of the linked list structures on page 3.
Then, create an additional structure like below

```
typedef struct queue {
      Node* front;
      Node* rear;  // or Node* back
} Queue;
```

Note: In the array implementation, there is a way of doing it that doesn't require both an indexOfFront and indexOfRear. 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. For example, if the front were index 1 and the rear were index 4, when adding a new item the rear would become index 0.


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 Queues - Heaps

Priority Queue: A queue where the arrangement of the items in the queue is based on their priority, for example a number. 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. Priority queues require an understanding of trees - if this understanding is not had, read the section *Trees - Binary Search Trees* on page 14 first.

A priority queue could be naively implemented by using an array and when an item is added it finds its appropriate place by starting at index 0 and then one-by-one going through each index until it finds its place. A much more efficient implementation can be done using a **heap.**

Heap: A data structure that can be used to implement a priority queue. Note that a heap and a priority queue are not the same. A priority queue is a general term for anything that fits the definition given above. A heap is a specific data structure adhering to certain properties (discussed on the next page) and it can be used to implement a priority queue. Another type of priority queue (the **binomial heap/queue**) is discussed later. There are 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 order in which items are inserted has nothing to do with which item is at the front. The front item could be the most recently added item if it had the highest priority.
- **Back (not relevant):** In a heap, there is no back like a regular 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 in the queue is based on its priority.
- Heaps are conceptualized as trees but are in fact most easily implemented in code using vectors.
- Do not confuse the heap data structure with the area in memory also called the heap that is used with the malloc function. They are two different things that happen to have the same name.
- Heaps are more complicated than the data structures discussed previously. In the following pages on heaps, all of the diagrams must be viewed in conjunction with each other to gain a full understanding. 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. Understanding how heaps keep items in the correct priority requires viewing all three diagrams.

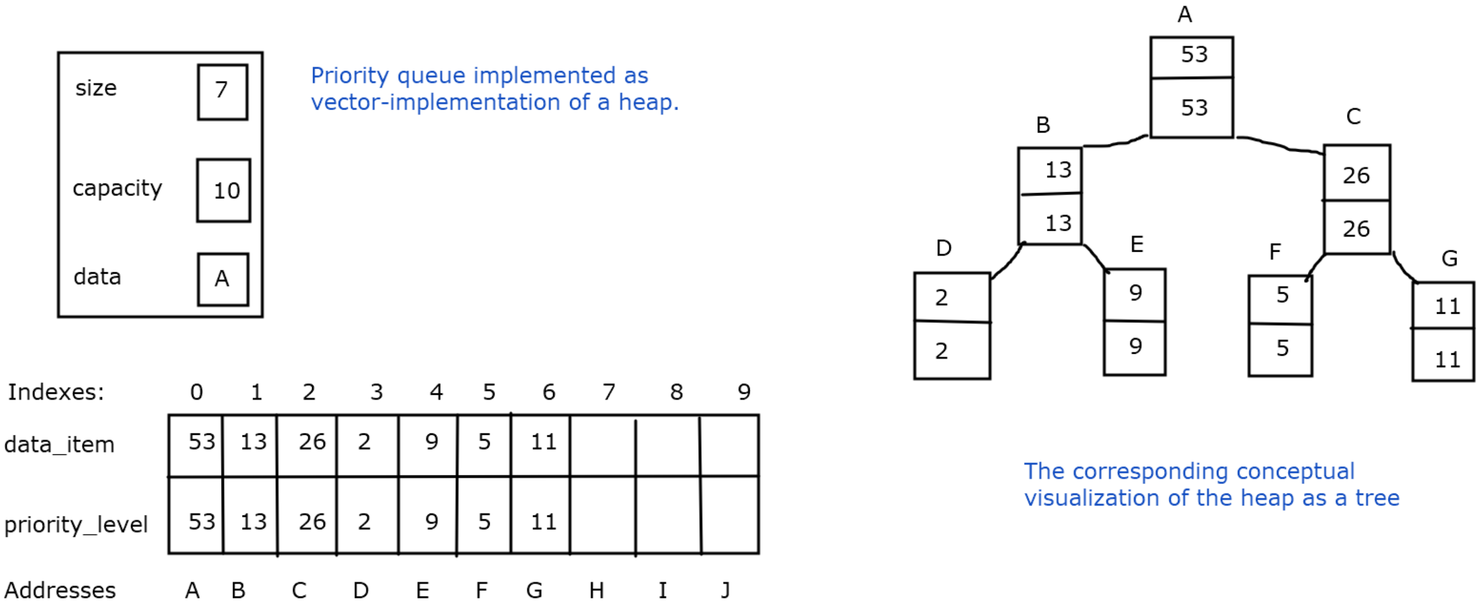
```
// array implementation - 2 structures required
typedef struct data_priority_pair {
        int data_item;
        int priority_level;
} Data_priority_pair;
```

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

This is an example of a **max-heap** - a priority queue where higher numbers have a higher priority. That doesn't have to be the case. There could also be a **min-heap** where lower numbers have a higher priority

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.

Heap Diagram 1: Visualization of the heap



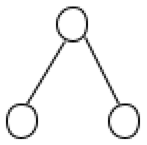
Priority queue implemented as vector-implementation of a heap.

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

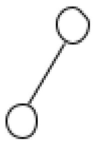
The corresponding conceptual visualization of the heap as a tree

Heap Properties:

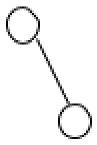
- Items are inserted top-to-bottom, left-to-right.
- Every node in the heap is bigger than its children.
- A heap is a **left complete tree**, but for convenience just a **complete tree** can be said. Also, it is arbitrary that left was chosen - it could've also been right.
    - **complete:** nodes filled in in proper order
    - **full:** if a node has children it has all of them that it can have



complete/full



complete/not full



not complete/not full

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 Operation Time Complexities

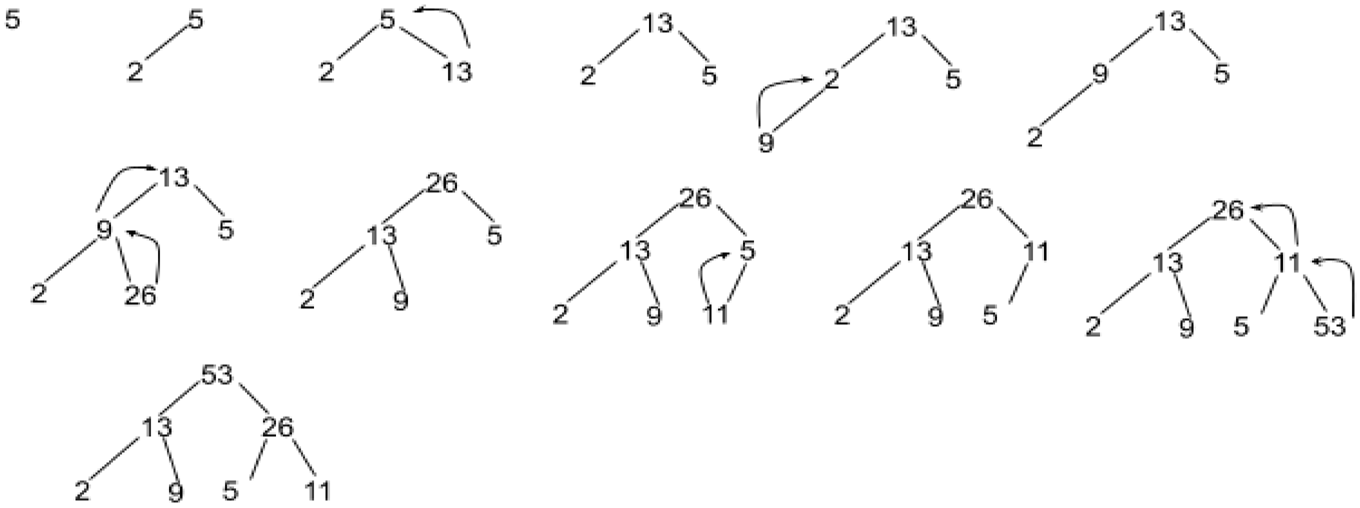
- lg is the base 2 logarithm. It is so common in computer science it's just abbreviated as lg.

<u>Operation</u>	<u>Time</u>	
enqueue:	O(lgN)	the potential fix up operations associated with enqueue is lgN
service:	O(lgN)	the fix down operations associated with service is lgN
merge:	O(NlgN)	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
front:	O(1)	highest priority item is at the front (index 0 in an array, the root in a tree)
empty:	O(1)	just check if it's empty (size is 0 in an array, the root is NULL in a tree)

Insert/Enqueue: Insert the item at the next available position, and then fix up 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 lgN (this really means floor(lgN) but it's referred to as lgN.
    - i.e. to insert a 15th item, this will be at most lg15 which equals 3. It equals 3 because 2^4 equals 16 which is larger than 15. The next lowest exponent is 3 for 2^3

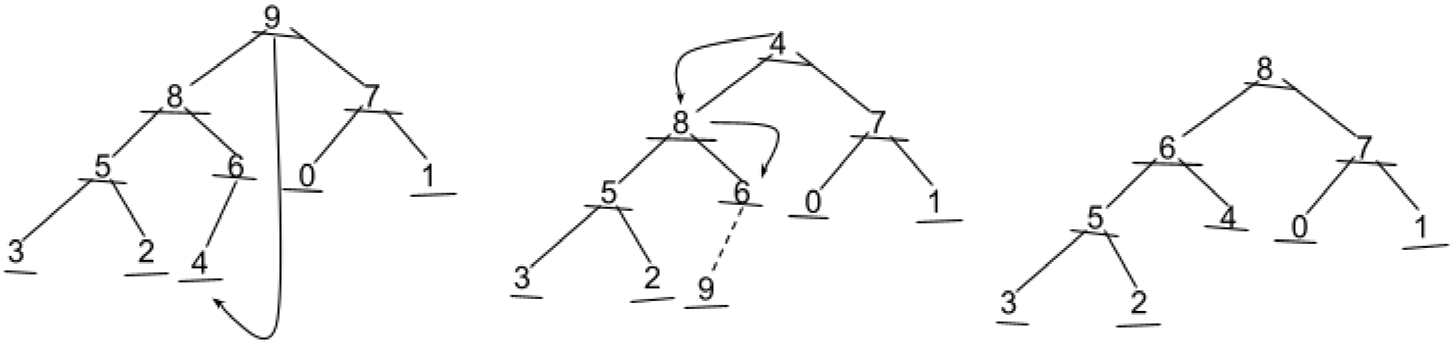
<u>Heap Diagram 2:</u> Adding to the heap - each number is also its own priority level.



Remove/Dequeue: Swap the first and the last element with each other. Then, remove the last element (which was previously the first element) from consideration, and fix down 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 an item has a lower priority than one of its children, it swaps with the higher priority child.
- If an item has a lower priority than both of its children, it swaps with the child that has the higher priority amongst the two.

Heap Diagram 3: Removing from the heap - always removes the highest priority item which is at the front.



9 is still there, just hidden

Merging Two Heaps is O(NlgN): 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.

- Though this is the best possible way to merge a heap, it is still considered inefficient because the **binomial heap** that was previously mentioned actually has a merge that is O(1).

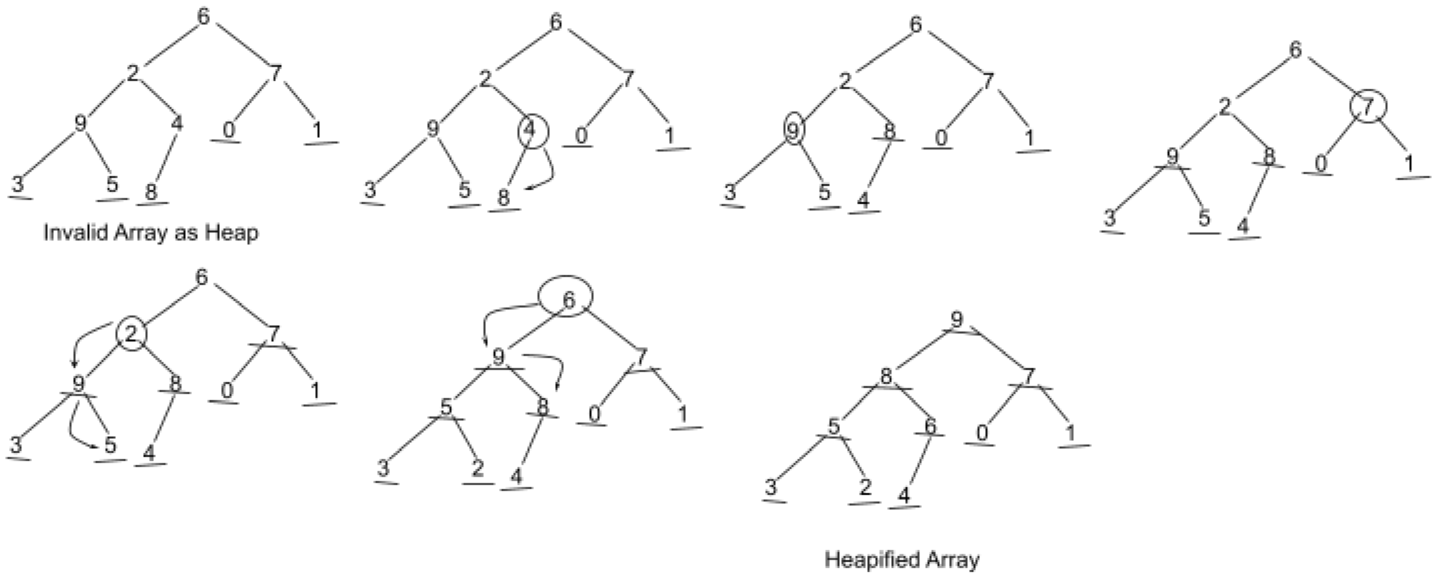
Turning an array into a heap

- Creating a heap the normal way, demonstrated previously, is O(NlgN).
- There is a faster way that is O(N) to create a heap which is to **heapify** the array. This is discussed on the next page.

Heapify

Heapify: Turn an array that's an invalid heap into a heap - used in heap sort.

- Heapify vs. heap priority queue - A heap priority queue adds items to/removes items from the queue in O(lgN) time using the fix-up and fix-down operations. Heapify takes an array and turns it into a heap in an instantaneous moment. It has nothing to do with heap priority queues.
- The array starts as an invalid heap with things randomly inserted.
- All the items in the lowest level are called **leaves** since they're at the end of the branch of the tree. The leaves have no children, and they are all **valid** heaps since they fit the heap property that they are larger than their children (or have no children).
- Start at the first non-leaf which is the first potential non-heap. 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



Invalid Array as Heap

Heapified Array

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

Priority Queues - Binomial Heaps/Queues

Binomial Heap: A type of priority queue that is a forest of trees, where two trees of the same size do not exist. The size of each tree is always a power of 2 - 1, 2, 4, 8 etc.

- When two trees of the same size exist, they combine and the higher priority root "wins" meaning they combine into one tree and the higher priority root becomes the root of the combined tree.
- When two trees combine together, the tree that "lost" goes under on the left most side of the tree that "won" which is now the root.

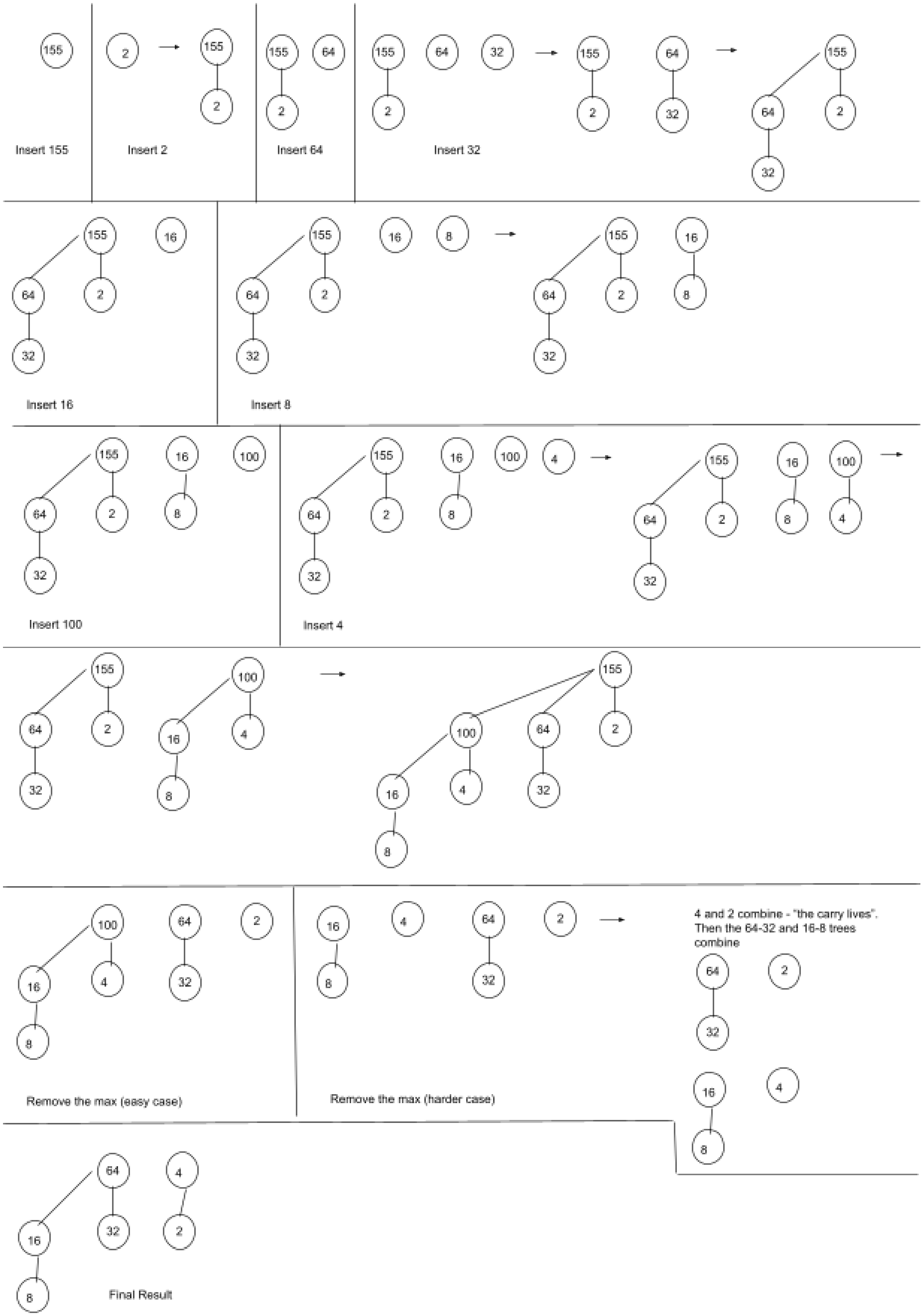
Operations:

Operation	Time
enqueue	O(lgN)
service	O(lgN)
Merge:	**O(1)**
Front:	O(1)
Empty:	O(1)

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 queue 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

Trees - Binary Search Trees

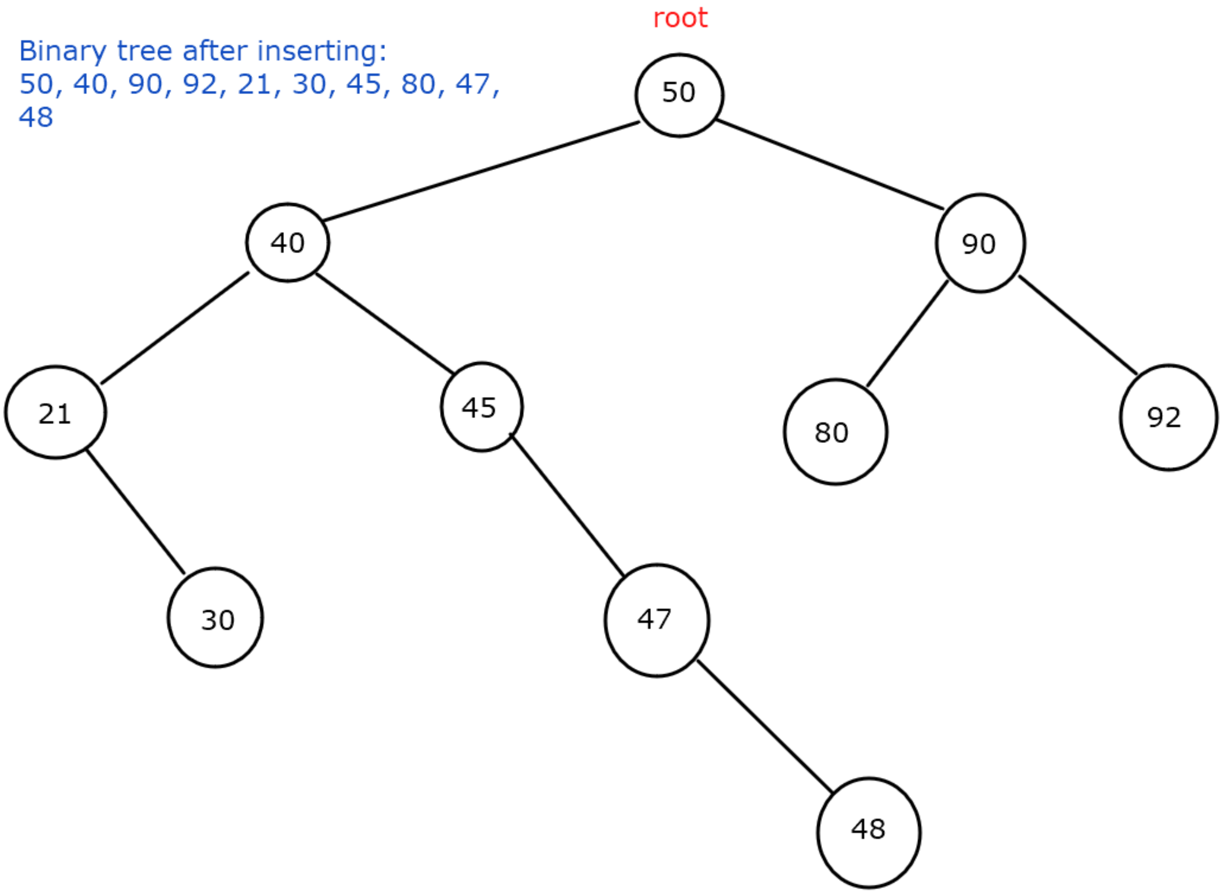
Tree: In general, a tree is a series of nodes starting at the root node where every node has a parent node (except for the root node) and child nodes.

- A 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. Also referred to as a BST or just binary tree.

- The binary tree is considered the most basic of all trees.
- The binary tree is not self balancing.

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



Binary tree after inserting:
50, 40, 90, 92, 21, 30, 45, 80, 47, 48

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

Inserting into the tree:

- best case scenario: tree is perfectly balanced
  - O(lgN) to insert one item
  - O(NlgN) to insert N items
- worst case scenario: all nodes go to the right or all go to the left in one giant diagonal line
  - O(N) for one item
  - O(N^2) to insert N items

To guarantee the best case scenario, that the tree is perfectly balanced, an AVL tree can be used which is discussed on the next page.

AVL Trees

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*.

```
typedef struct node Node;
struct node {
    int data;
    Node* left;
    Node* right;
    int height;
};
```

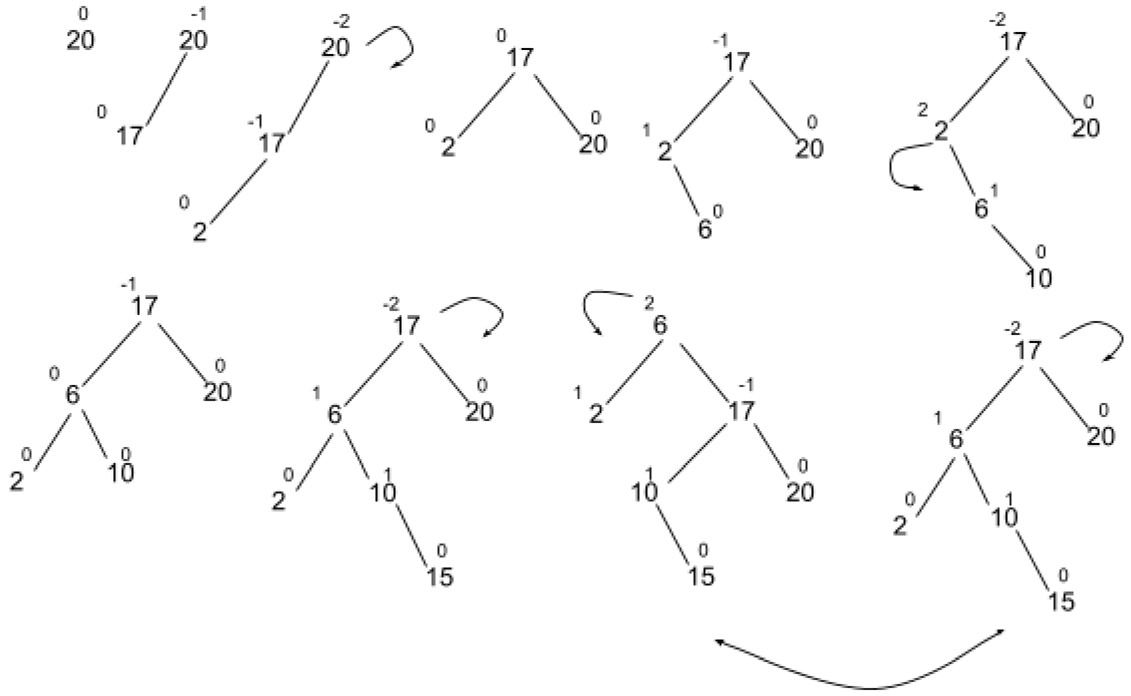
The following link provides code for an implementation of an AVL tree for integers. The logic of this code can be extrapolated to create an AVL tree for any data type.
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Magnitude: The avl tree is based on the principle that if the magnitude of the # of levels of children on the right minus the # of levels of children on the left is greater than 2, then the tree is unbalanced and rotations have to happen.

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

There are four situations that could be encountered: Two involve a single rotation, two involve two rotations. The 4 situations will be summarized below, but first it will be demonstrated why the two rotations are needed in two of the situations



Gets stuck going back and forth here b/c of the 2 and -2

Solution to this problem

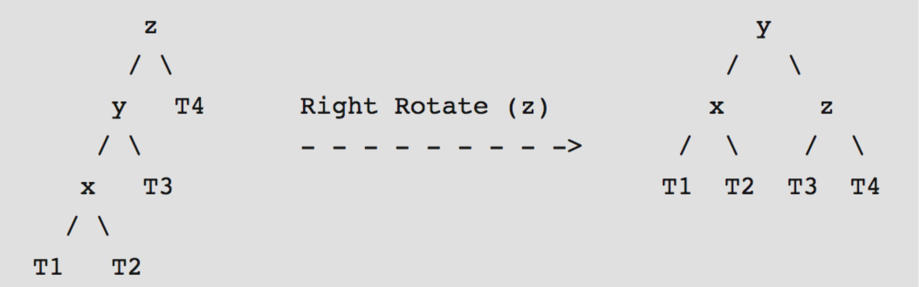
The tree gets “stuck” as seen in the above example when the children “lean” the opposite way of the parent. This means, for example, the parent is right heavy (2) but it’s right child is left heavy (-1) or the parent is left heavy (-2) but the left child is right heavy (1). In these two situations, the double rotation as seen below will need to be performed. The two simpler cases are when the parent leans the same way as the child (parent is right heavy (2), right child is right heavy also (1) or parent is left heavy (-2) and left child is left heavy also (-1).

So, in summary there are 4 cases - each case, and how to resolve each case, are discussed next.

Simple cases: parents and children lean the same way

Left-Left: parent (root) is left heavy (-2), left child is left heavy (-1)

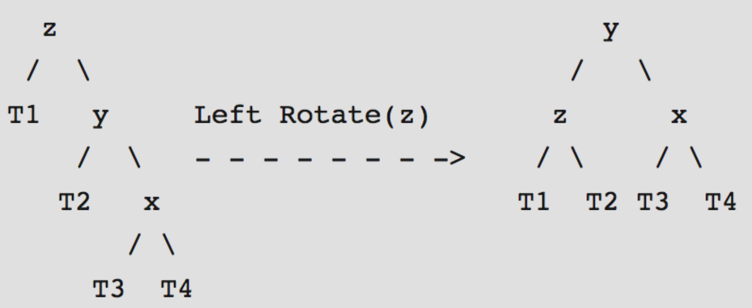
- perform a right rotation on the parent (root).



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

Right-Right: parent is right heavy (2), right child is right heavy (1)

- perform a left rotation on the parent (root).

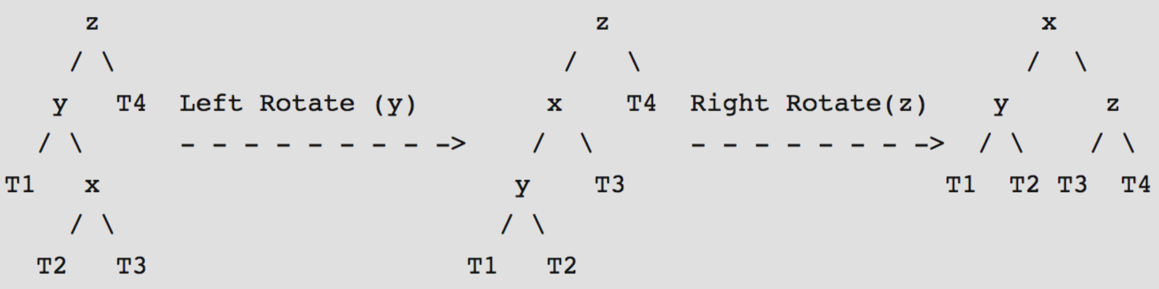


```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
      T3  T4
```

Complex Cases: parent and children lean opposite ways

Left-Right: parent is left heavy (-2), left child is right heavy (1)

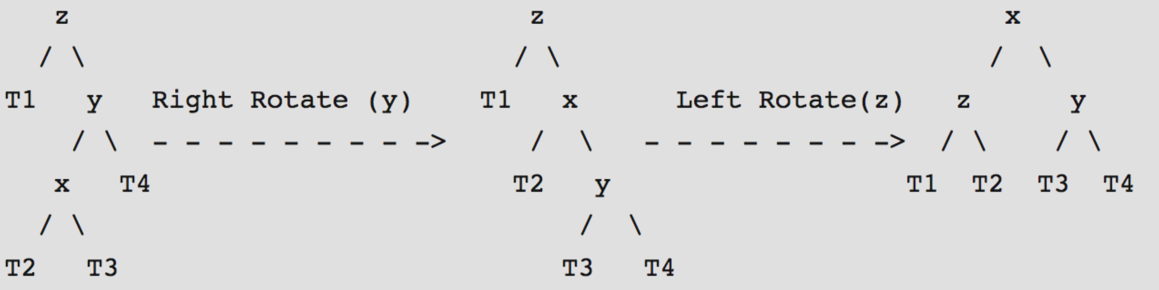
- perform a left rotation on the left child of the root
- perform a right rotation on the root



```
     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: parent is right heavy (2), right child is left heavy (-1)

- perform a right rotation on the right child of the root
- perform a left rotation on the root



```
   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
```

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

Hash Tables

Hash Tables: An array where instead of using an unsigned integer directly as an index to insert a piece of data, a key (string) is used as an index. Since a string can’t actually be used as an index because indexes need to be unsigned integers, a hashing function is developed which acts as an intermediary between the hash insert function and the piece of data actually going into the hash table. The hashing function will return a valid index within the array, and then the piece of data is inserted into the hash table.

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

- First, 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 hash function returns a number in that range, modular arithmetic must be used.
- Second, all hashing functions try to avoid **collision** as much as possible. Collision is when two the hashing function returns the same index for two unique keys. This is a result of the fact that 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. The math is the same with strings. So, a good hashing function will be written so as to avoid collision as much as possible. Collision is dealt with in two possible ways described below.

Though it isn’t valid C code, the example below is how a hash table can conceptually be thought of. In reality, there would be a hashing function that would actually take the strings “Mike”, “Sarah”, and “Tom” and convert them into valid indexes within the array called “ages”.

```
int ages[1000];    // an array to represent the ages of people
ages[“Mike”] = 18;
ages[“Sarah”] = 25;
ages[“Tom”] = 40;
```

The are two common ways of implementing hash tables - each deals with collision in its 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 general introduction and demonstration of quicksort meant for a person who has never studied it before. There is a lot to study about quicksort especially in regards to how to go about implementing the algorithm and this does not discuss everything.

Numbers in array: 9, 5, 6, 7, 2, 1, 0, 3, 8, 4

Goal: 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 first 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**

Done.

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 x 10^{30}.
However, 100! is so much larger and is approximately 9.3 x 10^{157}.
As a fraction this is $\frac{1.3\ x\ 10^{30}}{9.3\ x\ 10^{157}}$.

If you ignore the mantissa's 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. Although it's used a lot in the study of data structures it's independent of the study of data structures and can be used to calculate the running time of any algorithm. An example of this would be 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. For example, the time complexities of a linked list with N items would include calculating how long it takes to add to the head of the list and how long it takes to search for a specific item in the list in relation to N. 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.

To reiterate what was said in the introduction there are three major types of time complexities: Big-O, Big-Ω, and Big-Θ. They can be casually thought of as worst, best, and average case although they do not technically mean worst, best, and average case and are mathematical definitions meaning something else.

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 operations can avoid the worst case scenarios either by implementing them in a certain way or because the probability of the worst case scenario is so low. 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.

Big-O	Name	
O(1)	constant time	*Fastest*
O(lgN)	logarithmic time (base 2 logarithm)	…
O(N)	linear time (a version of polynomial time)	…
O(NlgN)	linear × logarithmic time	…
$O(N^2)$	quadratic time (a version of polynomial time)	…
$O(N^x)$	polynomial time (all other possible versions with different values of X)	…
$O(2^N)$	exponential time (in this case, N is the power)	…
$O(X^N)$	exponential time (all other possible versions with different values of X)	*Slowest*

An O(1) **constant time** operation means the amount of time an operation takes 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. 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, or it could take longer to complete than others when N is very small. However, in practice constant time operations are always the fastest because they are not affected by N. It's important to think about time complexities as N increases because that's what happens in real life scenarios.

An O(lgN) **logarithmic time** operation means the amount of time an operation takes is logarithmically proportional to N (base 2 logarithm). 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 8 is less than 10. To get from 3 to 4 that would be but then lg16 which is too high because 16 is greater than 10. 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 is linearly proportional to N. Another way to think of this is that the operation will take N units of time to complete.

An O(NlgN) operation is multiplying N by lgN so it combines linear and logarithmic time. 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 is quadratically proportional to N. Another way to think of this is that the operation will take N^2 units of time.

An O(N^x) **polynomial time** operation is the general formula for N raised to some power. Linear and quadratic are versions of polynomial time. The time increases as the power increases meaning $N^2 < N^3 < N^4$… even though they're all polynomial time.

An O(x^N) **exponential time** operation means the amount of time an operation takes is exponentially proportional to N. Another way to think of this is it will take x^N units of time such as 2^N units of time if x were 2. Like with polynomial time the time increases as x increases meaning $2^N < 3^N < 4^N$… even though they're all exponential time.

Consider the following scenario where N is 10 and you can see how the times compare to each other.

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

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.

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:* accessing any index in an array is O(1) and in this case it's the next available index
- *remove:* removing an item always takes the same amount of time
- *access:* accessing any index in an array is O(1)
- *search:* if the item didn't exist or was at the tail then all N items would 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 of so it can always be accessed instantaneously
- *insert (tail):* the tail node is not kept track of so all N nodes have to be traversed to reach it
- *remove (head):* the head node is kept track of so it can always be accessed instantaneously
- *remove (tail):* the tail node is not kept track of so all N nodes have to be traversed to reach it
- *remove (specific item):* if the item didn't exist or was the tail then all N items would have to be traversed
- *access(head):* the head node is kept track of so it can always be accessed instantaneously.
- *access (tail):* the tail node is not kept track of so all N nodes have to be traversed to reach it.
- *access (specific item):*if the item didn't exist or was the tail then all N items would have to be traversed

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/remove/access:* the top of the stack is kept track of and in a linked list-based implementation this would be accessing the head which is O(1) and in an array-based implementation this would be accessing an index which is O(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:* the back of the queue is kept track of and in a linked list-based implementation this would be accessing the tail which is O(1) and in an array-based implementation this would be accessing an index which is O(1).
- *remove/access:* the front of the queue is kept track of and in a linked list-based implementation this would be accessing the head which is O(1) and in an array-based implementation this would be accessing an index which is O(1).

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:* accessing any index in an array is O(1) and in this case it's 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 at the root node and accessing the root of a tree is always O(1)
- *merge:* the higher priority item becomes the root in the merged tree and the lower priority item becomes its child and this always takes the same amount of time so it's O(1)

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)

- *insert:* worst case is all items inserted are less than all items previously inserted which creates a tree that's a diagonal line going down to the left or all items inserted are greater than all items previously inserted which creates a tree that's a diagonal line going down to the right - in either case, all N nodes would have to be traversed every time a new item is added for the item to find its correct position in the tree.
- *remove:* worst case is all items inserted are less than all items previously inserted which creates a tree that's a diagonal line going down to the left or all items inserted are greater than all items previously inserted which creates a tree that's a diagonal line going down to the right - in the worst case all N items would have to be searched if the item being removed is at the deepest level in the tree.
- *search:* worst case is all items inserted are less than all items previously inserted which creates a tree that's a diagonal line going down to the left or all items inserted are greater than all items previously inserted which creates a tree that's a diagonal line going down to the right - in the worst case all N items would have to be searched if the item being searched for is at the deepest level in the tree.

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

Sorting Algorithms Time Complexities

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

In order to understand how a sorting algorithm works and see how it sorts the items it is very helpful to have a visualizer of them.