Data Structures

Table of Contents

Introduction

Data structures is a general term referring to all of the different ways in which data can be stored and worked with. Each individual data structure is defined by how it's stored in memory, the relationship that individual data items have to each other, and the various operations associated with it. The study of data structures is an intersection between mathematics and computer science. Any given data structure isn't better than another. Rather, different data structures have different properties and their utility is situation dependent. The underlying implementation of every data structure uses an **array** or it uses **nodes** in which case it's called a **linked** data structure. If a data structure is implemented using an array it means all of the items are contiguous meaning they are in consecutive blocks of memory. If a data structure is a linked data structure then it means all of the items are non-contiguous meaning they are not in consecutive blocks of memory, and to accommodate for this each item contains a link/connection to other items. The reason the items of an array are contiguous is because there is one single memory allocation that happens at once for all of the items. The reason the items of a linked data structure are not contiguous is because there are separate memory allocations for each individual node, and every time memory is allocated the location of its allocation isn't guaranteed to be contiguous to the previous memory allocation. There are three types of linked data structures called **linked lists**, **trees**, and **graphs**. This document doesn't discuss graphs and only discusses linked lists and trees. Arrays, linked lists, and trees are the three most fundamental data structures because not only are they data structures in and of themselves but they are used to build all other data structures (with the exception of graphs). The underlying implementation of every other data structure uses an array, linked list, or tree, and some data structures can be implemented using more than one of them.

This document has two main purposes. First, it provides conceptual explanations and visual representations of many of the most important and fundamental data structures. Learning and understanding data structures is all about visualization. It starts off by discussing the three most fundamental data structures described earlier and goes on to discuss all of the other data structures that can be built off of those three. Second, it discusses what time complexity is in general and also the time complexities of the operations associated with each individual data structure with an emphasis on Big-O notation. Data structures are intimately connected to time complexity because they can't be studied to any level that is considered adequate without also studying time complexity. A major part of what defines each data structure and makes it unique are the time complexities of its various operations. Lastly, there is a third and smaller purpose which is to provide some information about various sorting algorithms and specifically give an introduction to quicksort which is one of the fastest and most commonly used sorting algorithms. Oftentimes sorting algorithms are studied in conjunction with data structures.

For any given data structure or topic discussed in this document there is always much more discussion to be had. Ultimately, this document just provides a summary of a lot of the material that would be in any generic college-level data structures course. A person who reads this document will gain a good understanding of all of the most important and fundamental data structures and will gain a good understanding of time complexity in general and the time complexities of individual data structures

Additional Notes

There is limited discussion on data structures as they relate to code. When this does happen the C programming language is used (the code is also indistinguishable from C++ and is valid C++ code). This is arbitrary and data structures are independent of any one programming language. However, C is a good language to use when studying data structures and arguably the best. For example, C has pointers which are useful when studying linked data structures because it provides a further understanding of what it actually means for two nodes to be linked to each other in regards to the use of addresses. Many other languages do not have pointers.

Addresses are unsigned integers in a computer's memory. For example, a 64-bit machine has addresses that are 64-bit (8 byte) unsigned integers. 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 and linked implementations then generally speaking it can be said that if there are an unknown amount of items the array implementations present a downside due to the potential resize operations at runtime whereas linked implementations do not require resize operations. On the contrary, array implementations are better than linked implementations in regards to cache performance. Those are just a couple of examples and there is more discussion to be had.

The study of time complexity in this document is limited to Big-O notation but there are actually three major types:

- $O(N)$ 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 are actually mathematical definitions in regards to upper and lower bounds of equations. For example, for some equation $T(N)$ representing the runtime 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's acceptable 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.

There are many algorithms, and specifically operations performed on data structures, for which the implementations have both an iterative and recursive version. It is important to fully understand both versions and have the ability to write the code for both versions. However, in practice there are cases for which it would be better to use iteration. The use of recursion always poses a potential downside due to the overhead of the recursive function calls. At a minimum this can significantly increase the runtime in comparison to the iterative version and at a maximum this can cause a crash due to stack overflow. For example, calculating the Nth number in the fibonacci sequence can be implemented both iteratively and recursively. In practice the recursive version will take an unreasonable amount of time as N increases and will eventually cause stack overflow. The same is with data structure operations. For example, an item can be inserted into an ordered linked list both recursively and iteratively. The iterative implementation works for a list of any size and the recursive implementation can cause stack overflow even when the list is not that large. However, the statement that iteration should be used over recursion does not always apply and is nuanced. For example, there are cases where the performance difference between the two versions would largely be negligible or stack overflow would only happen in the recursive version for an astronomically large amount of items. An example of this would be inserting into an AVL tree. The time complexity of inserting into an AVL tree is O(lgN) because it's self-balancing so even inserting something as large as the 1 trillionth item is not slow because lg(1000000000000) = 39 which means at most 39 recursive function calls will have to happen. The extra runtime that would be caused by the overhead of these recursive function calls would be negligible in most contexts, and the amount of items that would have to be inserted into the AVL tree for stack overflow to potentially happen is so large it would never actually happen in practice. As another example, there are cases where the operation being done is by definition recursive or the recursive version is simpler than the iterative version so from the standpoint of learning something or simplicity recursion is better. An example of this would be tree traversals. A tree traversal is by definition a recursive operation and its iterative version is more complicated to implement because it requires the use of a stack. Ultimately, just know that for any given operation whose implementation has both recursive and iterative versions, in practice it might be better to implement the iterative version and it is just situation dependent. However, regardless of whether that's true, it's important to fully understand both versions and be able to write the code for both versions.

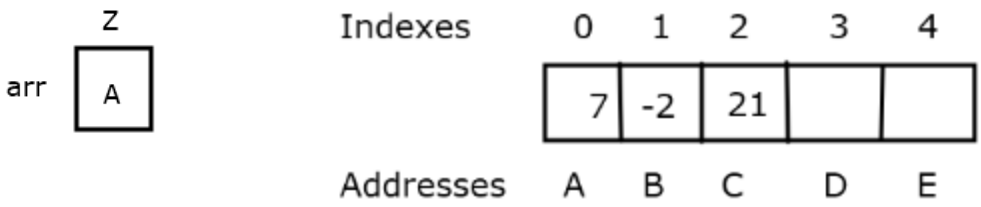
Array

Array: A collection of items stored in one contiguous block of memory with a fixed capacity where each item is associated with an index. Indexes are unsigned integers with the first index starting at 0 and increasing by 1. An array is one of the most fundamental data structures because it's used as the underlying implementation of other data structures including vectors, stacks, FIFO (regular) queues, heap priority queues, and hash tables.

```
int arr[5];
```

This says that *arr* is an array whose items are integers, has a capacity of 5 integers, and whose valid range of indexes are in the range of [0, 4]. Its capacity is fixed at 5 meaning it can't be resized to hold more integers. The value of the last valid index is always 1 less than the capacity because indexes start at 0. The first integer is at index 0 denoted by arr[0], the second integer is at index 1 denoted by arr[1], … the fifth integer is at index 4 denoted by arr[4].

As discussed in the introduction the word *contiguous* means the items are all in consecutive blocks in memory. The memory in a computer is a sequence of bytes. Each byte is a series of 8 bits and has an address. An analogy would be each byte is a house and the address of each byte is the address of each house. The address of the first byte is 1 Main Street, the address of the second byte is 2 Main Street, and so on. On a 64-bit machine addresses are 64-bit unsigned integers (8 bytes) and integers (int) are 32-bit signed integers (4 bytes). Using the array *arr* in the above example, this means that somewhere in memory there is a series of 20 sequential bytes allocated for the 5 integers in the array. The first integer is composed of bytes 1 - 4 and its address is the address of byte 1. The second integer is composed of bytes 5 - 8 and its address is the address of byte 5, … the fifth integer is composed of bytes 17 - 20 and its address is the address of byte 17. Additionally, the identifier *arr* holds the address of the first integer in the array which is the address of byte 1. Since addresses are 8 bytes that means that in addition to allocating 20 bytes for the array an additional 8 bytes must be allocated for *arr* so it can hold that 8 byte address. This means that *arr* isn't actually the array. It holds the address of the array but isn't the array itself. In fact, the array doesn't really have a name at all. It's the unnamed sequence of 5 integers somewhere in memory that can be accessed by using *arr* because *arr* holds the address of the first integer in the array, and by having access to the first integer in the array there is access to all other integers in the array because they're *contiguous* in memory. So although *arr* isn't actually technically the array for simplicity sake it's ok to think of it as the array when talking about it and referring to it. The diagram below would be if *arr* had 3 items added to it after it was created. Memory can't be empty so indexes 3 and 4 would actually contain whatever arbitrary garbage integer values are stored in the uninitialized integers. Also notice how *arr* has an address *Z* because it's a variable itself and all variables have addresses.



Below is what is actually happening in memory when the array *arr* is declared. The sequence of bits (all the 0s) are the binary representations of all of the integers in the array. For example, the first sequence of 32 bits is the binary representation of the first integer in the *arr*. This is bytes 1 - 4 with each byte having address A, A2, A3, and A4 respectively where A, A2, A3, and A4 are some 64-bit unsigned integers. The second sequence of 32 bits is the binary representation of the second integer in the array. This is the bytes 5 - 8 with each byte having address B, B2, B3, and B4 respectively where B, B2, B3, and B4 are some 64-bit unsigned integers.

The array of integers somewhere in memory

A	A2	A3	A4	B	B2	B3	B4	
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	…
byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	…
first integer				second integer …				
arr[0]				arr[1] …				

The fact that all of the bits are 0 is just for demonstration. The integers in *arr* are uninitialized so the bits are actually the binary representation of whatever arbitrary garbage integers value are stored in the uninitialized integers.

Additionally, *arr* is also somewhere in memory holding the address of the first integer in the array which is the address of the first byte in the sequence of 20 bytes. Again, the fact that all of the bits are 0 is just for demonstration. In reality, it would be the binary representation of the 64-bit integer which is the address of the first integer in the array referred to as A.

arr somewhere in memory - the 64-bit unsigned integer representation of A

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Lastly, it's important to understand what arr[0], arr[1], … and arr[4] actually mean.

- *arr* - By itself it evaluates as the address of the first integer in the array.
- [ ] - In a declaration like
  ```
  int arr[10]
  ```
  it means the data being declared is an array but in the context of after a declaration like
  ```
  arr[1]
  ```
  it's the dereference operator. Dereference means take the address of *something* and go to the actual *something* (in this case, the *something* is an integer).
- arr[0] - Go to the address of the integer at index 0 which is the first integer in the array. Then, dereference the address which actually then goes to the first integer in the array. So, arr[0] evaluates as the first integer in the array.
- arr[1] - Go to the address of the integer at index 1 which is the second integer in the array. Then, dereference the address which actually then goes to the second integer in the array. So, arr[1] evaluates as the second integer in the array.

Putting all of that together, it should be clear what *contiguous* actually means and why it’s important in arrays.

Linked List

Linked List: A linked data structure that is a collection of items that are non-contiguous in memory in which every item is stored in a node and is linked to the next item and potentially the previous item forming a list. It's a *list* of nodes that are all *linked* to each other hence the name *linked list*. These links are made using node pointers. Within each node are node pointers holding the addresses of the nodes it's linked to. 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 keeps track of the head or both the head and the tail. A linked list is also commonly referred to as just a **list**. A linked list is one of the most fundamental data structures because it's used as the underlying implementation of other data structures including stacks, FIFO (regular) queues, and hash tables.

- **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 can 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 can 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 stores the address of 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 holding the address of the tail node.
- A linked list can have zero or more nodes. If it has zero nodes then it's considered empty in which case the head pointer is set to NULL (and tail pointer if the tail is being kept track of). If it has one node then that one node is both the head and the tail. If it has more than one node then the head and tail are different nodes.
- *Linked list operation specific names:* There are no special names for some linked list operations with the exception of prefixing the location of the operation. For example, *head_insert* is used to refer to inserting an item at the head.

```
// 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;
};
```

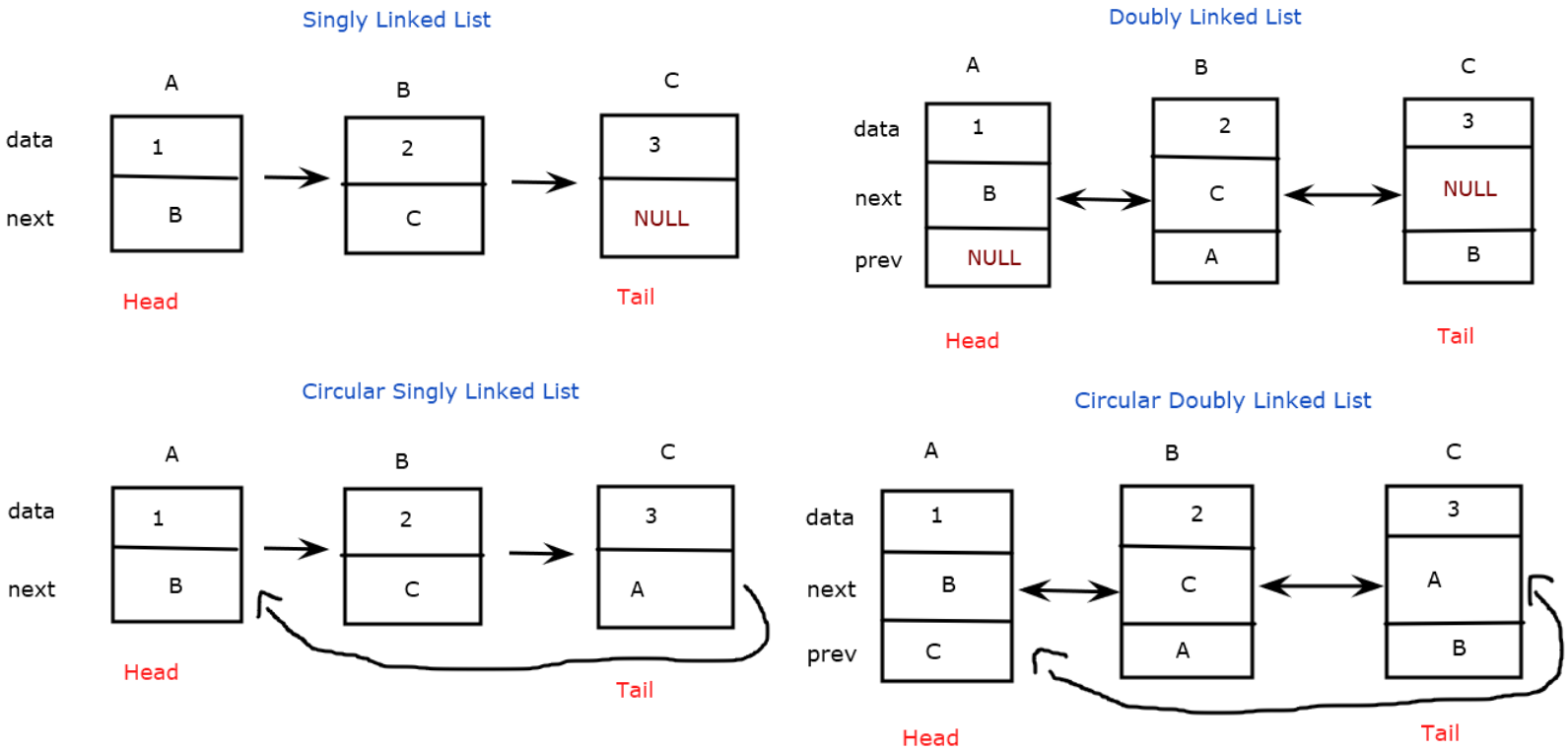
```
// optional linked list structure
typedef struct list {
    Node* head;
    Node* tail;  // optional
    int size;    // optional
} List;
```

Tracking the size and tail node means getting the size and the operations associated with the tail (like tail_insert) are O(1) operations instead of O(N). Implement this as necessary.

It's optional but nice to have a separate structure for the linked list. A single Node pointer could be used to track the head of the linked list but a node itself isn't really a linked list but rather a single entity in the linked list. Additionally, if the tail and/or size had to be kept track of then it makes much more sense to put everything in one structure.

If the linked list is being implemented by using a Node pointer just to track the head then for any given operation that may update the head there are always two versions of the implementation. The **pass-by-value** version will pass the head pointer by value as an argument and update the head by returning a pointer to the updated head, and the **pass-by-reference** version will pass the head pointer by reference as an argument and update the head with assignment. It's important to always be able to understand and implement both versions for any given operation. For example, the two versions of inserting at the head are below:

```
// function declarations
Node* head_insert_pass_by_value(Node* head, int data);
void head_insert_pass_by_reference(Node** pHead);

// function calls
Node* list = NULL;
list = head_insert_pass_by_value(list, 1);
head_insert_pass_by_reference(&list, 2);
```



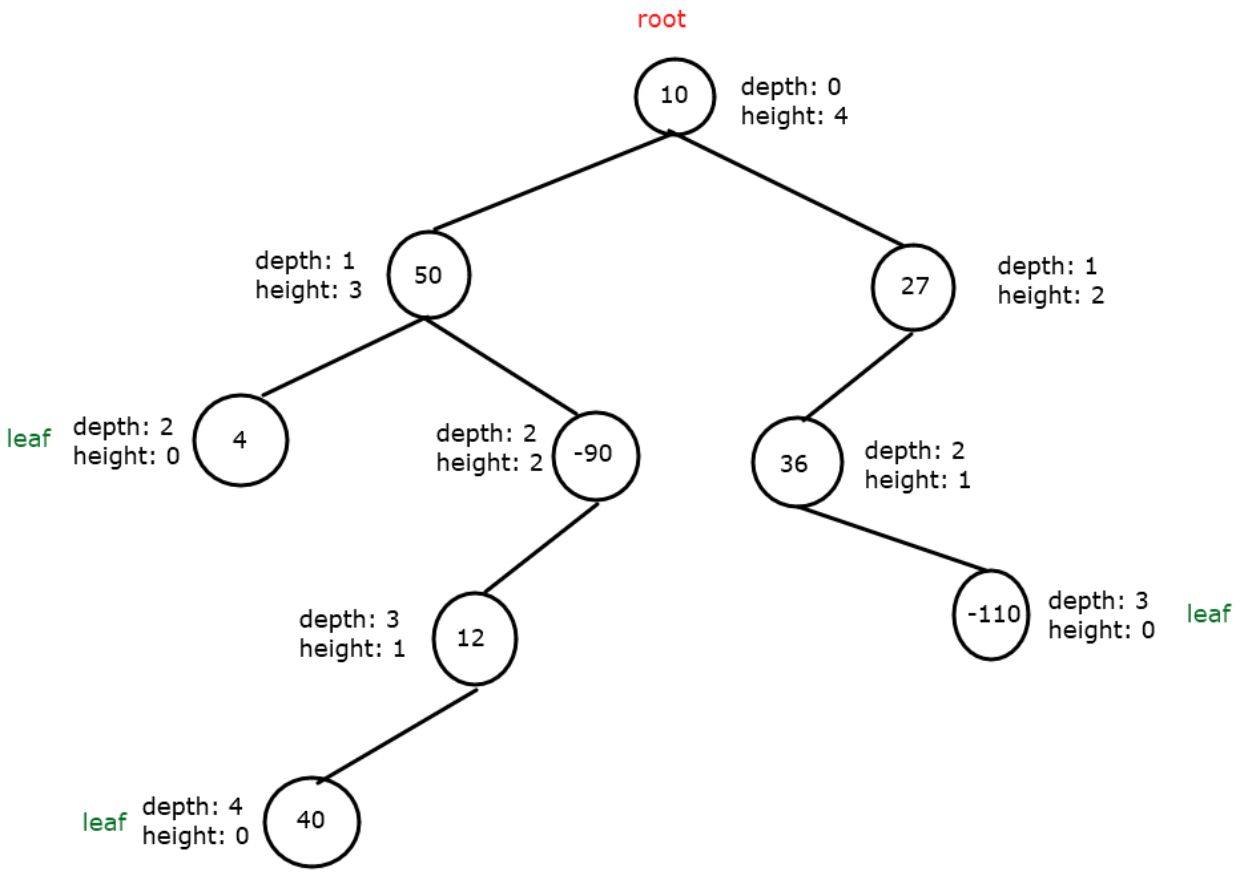
Tree

Tree: A linked data structure that is a collection of items that are non-contiguous in memory in which every item is stored in a node and is linked to other items via parent-to-child relationships. These links are made using node pointers. Within each node are node pointers holding the addresses of the nodes it's linked to. It's called a tree because the nodes branch out from the base of the tree just like branches on an actual tree. Unlike an actual tree whose base is at the bottom and whose branches grow upwards, a tree data structure has its base at the top and its nodes branch out going downwards so in that sense it's like an upside down actual tree. A tree is one of the most fundamental data structures because it's used as the underlying implementation of other data structures including binary search trees, AVL trees, heap priority queues, and binomial heap priority queues.

- Every node in the tree has a parent node with the exception of the **root** which is the first node in the tree. The root is like the base of an actual tree.
- The parent-to-child relationship is one-way meaning a node can directly access its children but not its parent. In other words, it's possible to go from a parent node down to a child node but not from a child node up to a parent node. The implication of this is that with the exception of the root node it's not possible to access every single node in the tree from any given node. This is similar to how in a singly linked list each node can access the node after it but not the node before it.
- Every node in the tree has zero or more children and each node that has zero children is called a **leaf**. Collectively, all of the nodes with zero children are the **leaves** of the tree. The non-leaf nodes are like the parts of the branches in the middle of an actual tree and the leaf nodes are like the leaves and terminating branches of an actual tree.
- The amount of children that every node can have is dependent on the tree. For example, a tree could allow each node to have up to two children, up to three children, or place no limit on how many children it can have. Most commonly, a tree has nodes that can have up to two children called the **left child** and the **right child**. In that case there would be 4 possible combinations of child nodes: no children, only a left child, only a right child, or both a left and a right child. If a particular child doesn't exist then the pointer to that child node is set to NULL.
- Every node has a **depth** which is the number of levels down it is from the root. Another way to define it is the number of edges from the node to the root. The root's depth is 0.
- Every node has a **height** which is the number of levels up it is from the leaf it's connected to that it's furthest away from. Another way to define it is the number of edges on the longest path from the node to a leaf it's connected to. The longest path means if a node is connected to multiple leaves then the longest path is the path to the leaf for which the most number of edges have to be traversed.
- For any given node all of the nodes to its left are called its **left subtree** and all of the nodes to its right are called its **right subtree**. In that sense, a tree is just a collection of subtrees.
- A tree can have zero or more nodes. If it has zero nodes then it's considered empty in which case the root node pointer is set to NULL. If it has one node then that one node is both the root and a leaf. If it has more than one node then it's not possible for the root to be a leaf.
- Trees are the data structure in which recursion is most commonly used in its operations.

The following diagram is a tree where the root is 10 the three leaves 4, 40, and -110. The integers that are in each node are arbitrary and have no relationship to each other. Take 50 as an example:

- It has a depth of 1 because it's 1 level down from the root.
- It has a height of 3 because it's 3 levels up from the leaf it's furthest away from. Notice how it's connected to two leaves (4 and 40) but 40 is the one that it's furthest away from.
- It has a left subtree composed of 4.
- It has a right subtree composed of -90, 12, and 40.
- 50 cannot access 10, 27, 36, or -110 because the parent-to-child relationship is one way.



Tree Traversals

Tree Traversals: A reference to the various ways in which a tree can be navigated. Different tree traversals visit the nodes of a tree in different orders. The implementation of tree traversals is most easily done recursively but it can also be done iteratively with a stack. 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 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:* 10, 50, 4, -90, 12, 40, 27, 36, -110
- *in-order traversal:* 4, 50, 40, 12, -90, 10, 36, -110, 27
- *post-order traversal:* 4, 40, 12, -90, 50, -110, 36, 27, 10

Important Note

It's important to take a pause and emphasize that as discussed in the introduction an array, linked list, and tree are together the three most fundamental data structures because the underlying implementation of all other data structures uses one of these three (again, with the exception of graphs). Having a complete understanding of arrays, linked lists, and trees is the foundation to understanding all of the other data structures. If this understanding isn't had or is lacking in some way it's important to spend more time reviewing these three data structures before continuing on. Listed below are the data structures that use an array, linked list, and/or tree in its underlying implementation:

- **vector:** array
- **stack:** array or linked list
- **FIFO (regular) queue:** array or linked list
- **priority queue - heap:** array or tree
- **priority queue - binomial queue/heap:** tree
- **binary search tree:** tree
- **AVL tree:** tree
- **hash table:** array (open addressing) or both an array and linked list (separate chaining)

Note: As will be discussed next, a vector is an array with a non-fixed capacity which means it can accommodate for an unknown amount of elements. This means that above when saying an array can be used in the implementation of something what is actually being used is a vector because a data structure should always be able to work for an unknown amount of elements. For example, in the array implementation of a stack what is actually being used is a vector. However, the term *array* is used above because the actual underlying fundamental data structure being used is an array - a vector is itself an array just with the addition of having a non-fixed capacity.

Vector

Vector: An array with a non-fixed capacity. This means the vector is resized when it can no longer fit more items unlike a traditional array. The implications of this are that if there are an unknown amount of items then a vector must be used. A traditional array can only be used if there are a known amount of items in which case the array is created with a large enough capacity to begin with. Additionally, although it's acceptable to insert and remove items at any location in the vector, this is traditionally done at the back. Since a vector is an array its items are contiguous in memory.

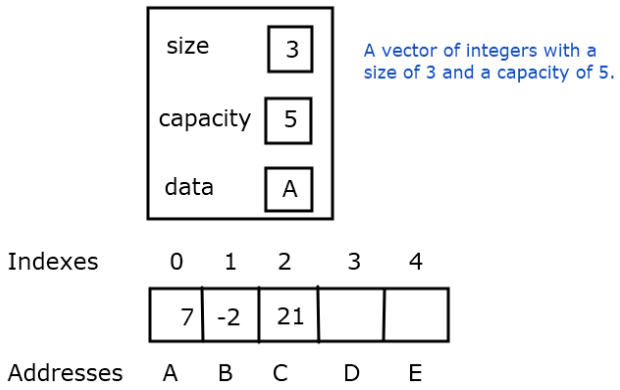
- *Vector operation specific names:* The terms **push back**, **pop back**, and **at** are used to refer to inserting an item at the back, removing the back item, and accessing an item at any index. If an item is inserted or removed at a location other than the back then the generic terms *insert* and *remove* can be used.

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 inserting a new item then a resize operation on the array is 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 decreased performance during runtime. For example, increasing the capacity by 1 saves memory but decreases performance due to more resize operations.
- **array** - The actual array, called *data*, in this example.

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

Below is what the array *arr* would look like from the array section if it were in vector form. In this case, if an item is added to the vector when the size equals the capacity then the vector is resized which *arr* would not be able to do.



Stack

Stack: A collection of items where only the **top** can be worked with. Items can only be added to the top, only the top item can be removed, and only the top item can be accessed just like a stack of plates. A stack can be implemented using an array or linked list. In the array implementation the top item is always at index [size - 1] and the items are contiguous in memory. In the linked list implementation the top item is always at the head of the list and the items are non-contiguous in memory because it's a linked data structure.

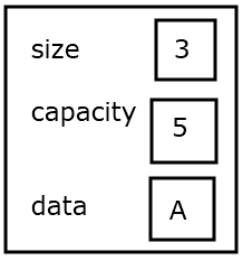
- The key difference between a stack vs. an array or linked list is that in a stack only the top is accessible. For example, in a stack of integers only the top integer can be accessed at index [size - 1] whereas in an array of integers any integer at any index can be accessed. Similarly, in a linked list stack of integers only the top integer can be accessed at the head node whereas in a regular linked list of integers any integer at any node can be accessed.
- *Stack operation specific names:* The terms **push**, **pop**, and **top** are used to refer to inserting an item into the stack, removing an item from the stack, and accessing the top item.

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

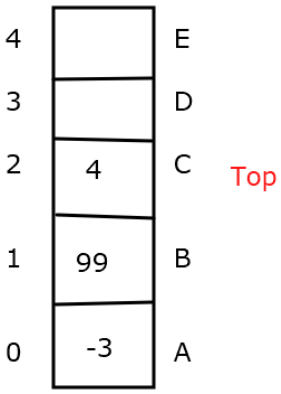
// linked list implementation
Use one of the linked list node structures. Then create another structure like below.

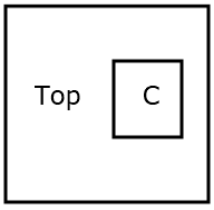
```
typedef struct stack {
    Node* top;
    int size;  // optional
} Stack;
```

Tracking the size in a linked list implementation means getting the size is an O(1) operation instead of O(N). Implement this as necessary.

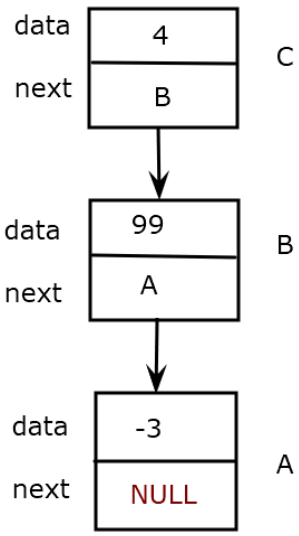


Vector implementation of a stack with a size of 3 and a capacity of 5





Singly linked list implementation of a stack a size of 3



FIFO (Regular) Queue

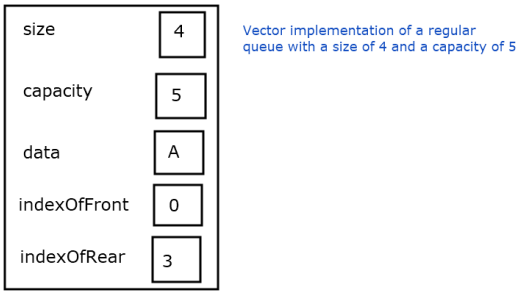
FIFO Queue: A collection of items where the arrangement of the items is based on the order in which they entered the queue. Items are added to the **back**, removed from the **front**, and only the front item can be accessed. FIFO stands for *first-in-first-out*. The first item in is the first item out. In other words, the first item ever put into the queue is the first item ever removed from the queue, and more generally when an item is removed from the queue it is always the least recently added item. A FIFO queue can be casually referred to as a *regular queue* or just *queue* since it's 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. A queue can be implemented using an array or linked list. In the array implementation the indexes of the back and front are tracked and the items are contiguous in memory. There is also an implementation that only requires tracking one of them and the other can be calculated using the index of the one being tracked and modular arithmetic. In the linked list implementation the front is the head of the list, the back is the tail of the list, and the items are non-contiguous in memory because it's a linked data structure.

- *FIFO queue operation specific names:* The terms **enqueue**, **dequeue/service**, and **front** are used to refer to adding an item to the queue, removing an item from the queue, and accessing the front item.

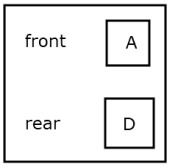
```
// array implementation
typedef struct queue {
    int size;
    int capacity;
    int* data;
    int front;  // index of front
    int back;  // index of back
} Queue;
```

```
// linked list implementation
Use one of the linked list structures.
Then create another structure like below.

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

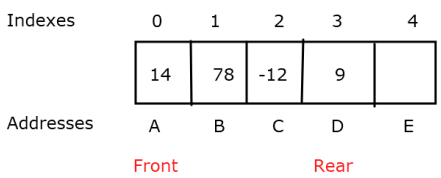
Tracking the size in a linked list implementation means getting the size is an O(1) operation instead of O(N). Implement this as necessary. Additionally, the array implementation should be a *wraparound queue* to improve efficiency and save memory. Using the example below, if the front is index 1 and the rear is index 4 then when adding a new item the rear becomes index 0 instead of resizing the array and using index 5. This prevents an unnecessary resize operation

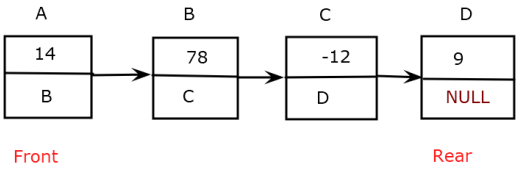


Vector implementation of a regular queue with a size of 4 and a capacity of 5



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 unlike a FIFO queue.

- *Priority queue operation specific names:* same as FIFO queue.

A priority queue can be implemented in multiple ways and its definition is independent of any specific implementation. As long as the arrangement of the items is based on their priority then it's a priority queue. The first idea that may come to mind would be using an array and when an item is added it finds its correct position by starting at index 0 and then going through all of the indexes one-by-one until it finds its place. Upon finding its place it's stored in that index and then all items of lower priority then have to be shifted over one-by-one. This is inefficient and a more efficient implementation can be achieved 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 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's not at the *back* because there is no *back*.
- A heap by definition is a tree but it can also be implemented using an array. In the array implementation the front is always at index 0 and the items are contiguous in memory. In the tree implementation the front is always at the root and the items are non-contiguous in memory because it's a linked data structure.
- Don't confuse the heap data structure with the area in memory also called the heap that is used with the *malloc*, *calloc*, and *realloc* functions. They are two different things that have nothing to do with each other that happen to have the same name.

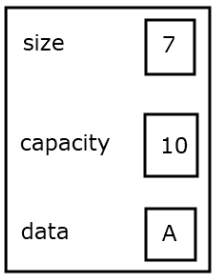
```
// array implementation
    typedef struct priority_queue {
        int size;
        int capacity;
        Item* data;
    } Priority_queue;

typedef struct item {
    int data_item;
    int priority_level;
} Item;
```

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.

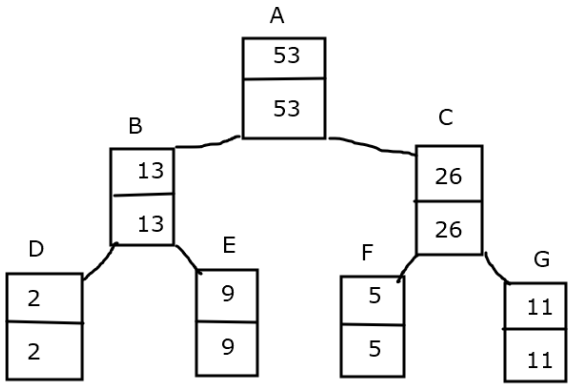
For the purposes of making the demonstration more simple, the data item and priority level are the same meaning 53 also has a priority level of 53. What matters though is the priority level and not the data item. For example, if 53 had a priority level of 80 and 13 had a priority level of 90 then 13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

Heap Diagram 1: 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

- **Heap property:** Every node in the heap has no children or is larger than its children if it has children (**max heap**), or every node has no children or is smaller than its children if it has children (**min heap**).
- Items are inserted top-to-bottom, left-to-right.
- A heap is a **left complete tree**.
  - **Complete:** Nodes are inserted in the proper order.
  - **Full:** If a node has children it has all of them that it can have.

Index Calculations For Heap As Array

The array implementation of a heap uses 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's so common in computer science that it's abbreviated as lg. Additionally, lgN really means floor(lgN) which means the result is rounded down to the nearest integer. For example, lg10 = 3 because lg10 is approximately 3.3 which gets rounded down to 3.

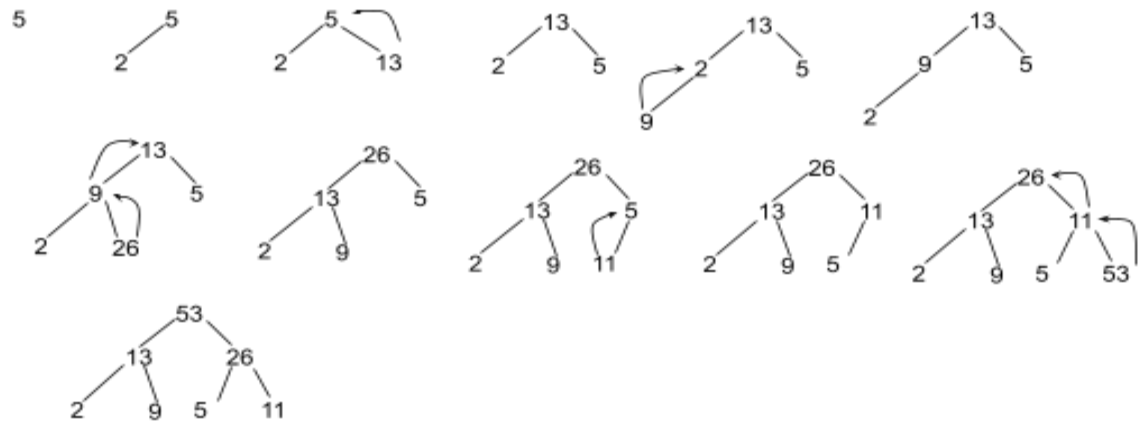
Operation	Time	
enqueue	O(lgN)	at most lgN nodes are traversed/at most lgN fix-ups operations
service	O(lgN)	at most lgN nodes are traversed/at most lgN fix-down operations
merge	O(NlgN)	1 - remove an item from one heap (lgN) 2 - insert it into the next heap (lgN) 3 - do this for N items in that one heap (N) 4 - 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 for array, root for tree)
empty:	O(1)	check if it's empty (size is 0 for array, root is NULL for tree)

Note: The descriptions below are for max heaps. The same process applies for min heaps, just change the logic for the priority levels.

Inserting Into A Heap

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 its parent if it has a higher priority than its parent.

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

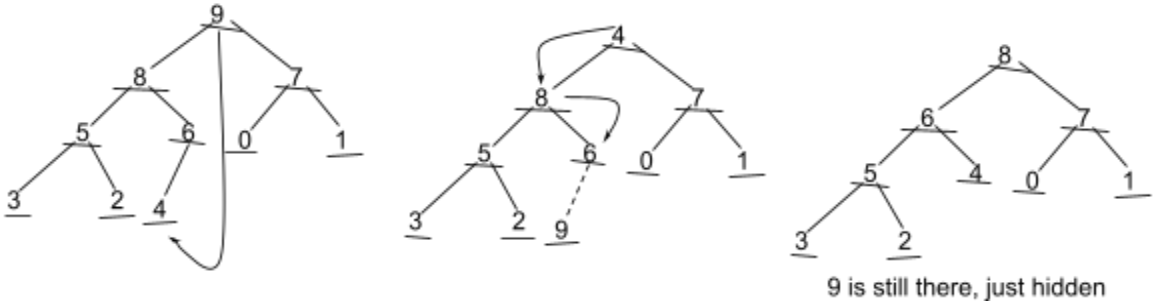


Removing From A Heap

Dequeue/Service: Only the highest priority item can be removed from the heap. This is achieved as follows - Swap the first and the last item with each other. Then, remove the last item (which was previously the first item) from consideration and perform the **fix-down** operation on the first item (which was previously the last item) 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 that 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.

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



Merging Two Heaps

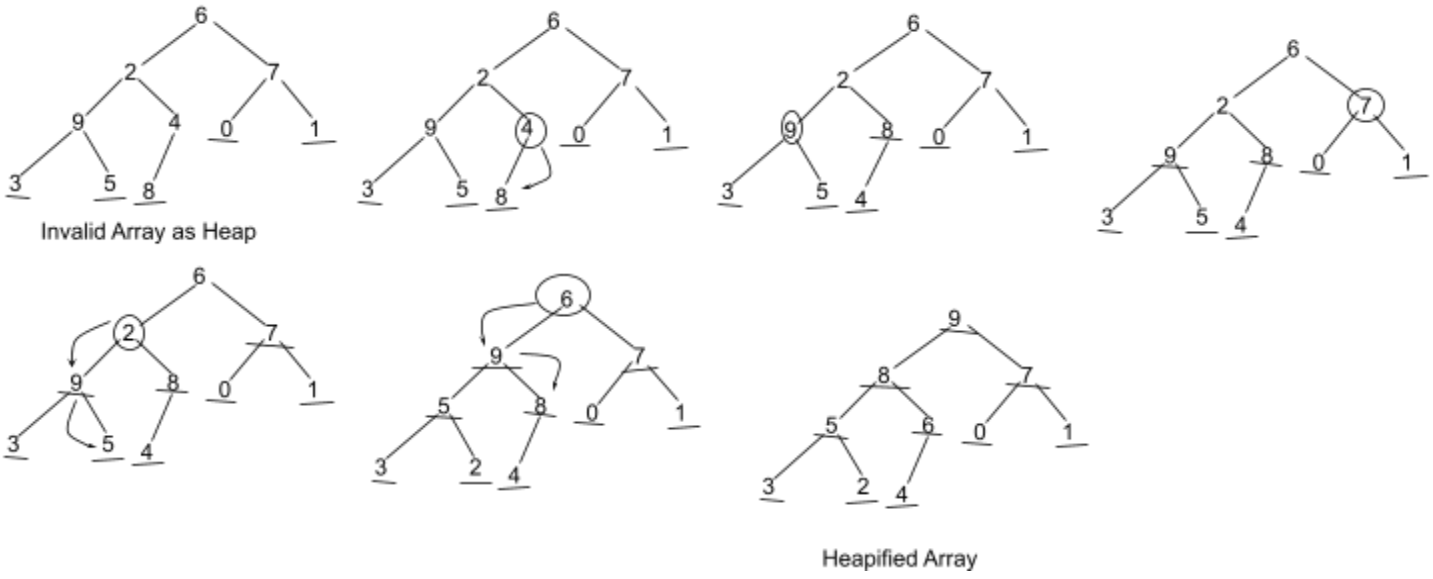
Merging Two Heaps: When two heaps are merged all of the items from one heap are added to the other heap which has a time complexity of O(NlgN). This is achieved as follows - Remove an item from one heap (lgN), insert it into the next heap (lgN), and do this for N items in that one heap so that's 2NlgN 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** which is discussed in the next section.

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 is being implemented as an array then the same steps could be followed using the index calculations.

- 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 fix-down 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's not a valid heap since it's not larger than all of its children
    - Call fix-down on 2 and fix-down until it's in the proper position. Now 6 is the next potential non-heap.
- Examine 6 - 6 isn't 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. This would be done by taking something that's not a heap (an array or tree with items in random order) and then heapifying it. 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 using fix-up. This has a time complexity of O(NlgN) because inserting to the heap is O(lgN) and this must be done for N items. This is less efficient than O(N).
- Heapify isn't 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's the most efficient way to create a heap, and it's used in the heap sort sorting algorithm. However, it's 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). So, put the terms *heap priority queue*, *fix-up*, and *fix-down* together and then put the terms *heapify, heap sort*, and *fastest way to create a heap* together.

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 called a binomial heap. Since a binomial queue is a tree its items are non-contiguous in memory because it's a linked data structure.

- *Binomial queue/heap operation specific names:* same as FIFO (regular) queue and heap priority queue.

Binomial Heap Time Complexities

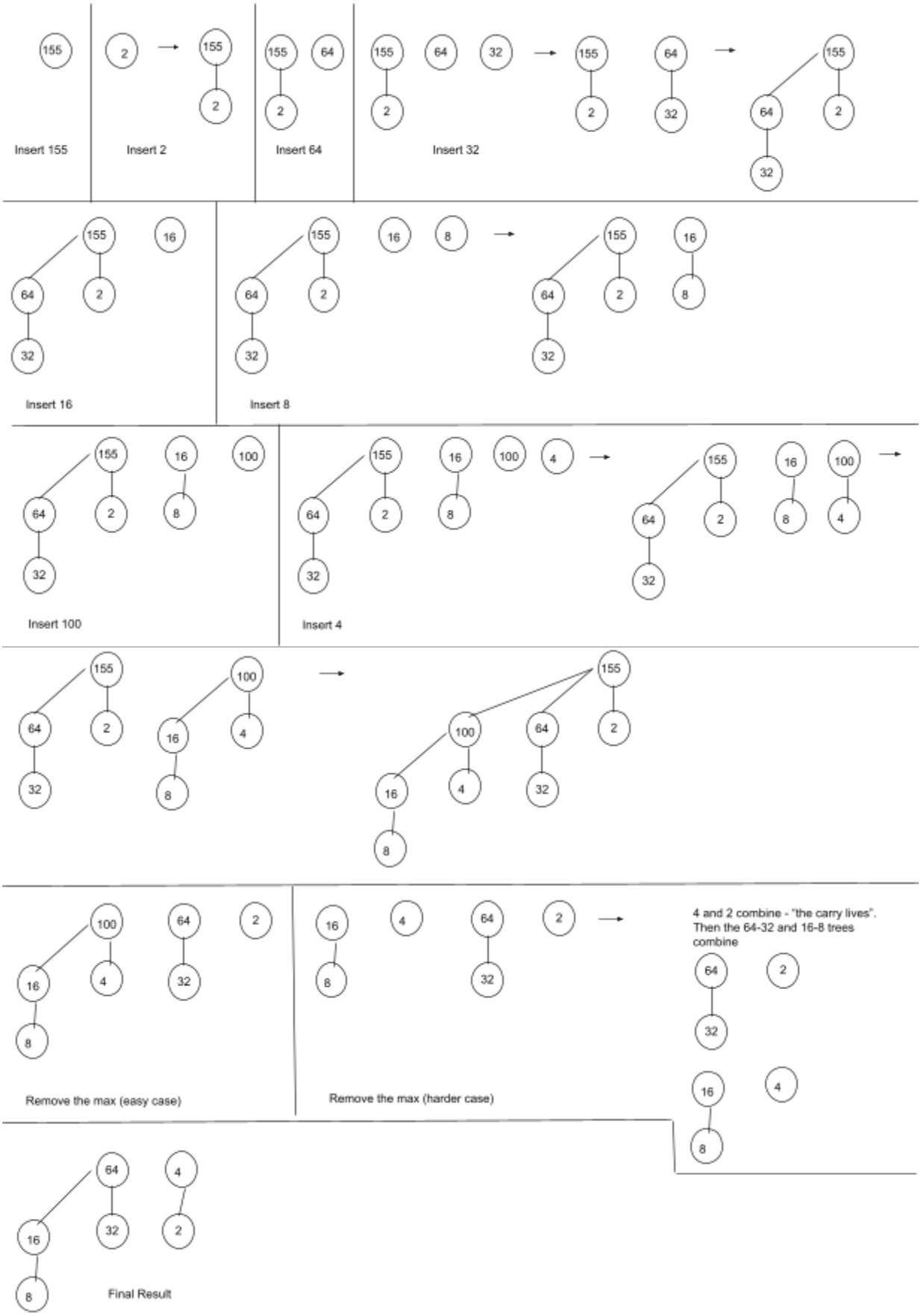
Operation	Time	
enqueue	O(lgN)	lgN merges will have to happen at most
service	O(lgN)	lgN merges will have to happen at most
merge	**O(1)**	the two trees merge and the higher priority root wins
front	O(1)	highest priority item is at the root
empty:	O(1)	hust check if it's empty (root is NULL)

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 binomial queue itself can be represented as a binary number. For example, a binomial queue 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 queue of size 7 would be 111 since there would be a 1 tree (2^0 place), a 2 tree (2^1 place), and a 7 tree (2^2 place) 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

Binary Search Tree

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. Since a binary tree is a tree its items are non-contiguous in memory because it's a linked data structure.

- The binary tree is considered the most basic of all trees.
- The binary tree isn't self-balancing which has negative implications. What is meant by self-balancing is discussed more in the section on **AVL trees** which is the next section.

```
// 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;
    int size;  // optional
} Binary_tree;
```

Tracking the size means getting the size is an O(1) operation instead of O(N). Implement this as necessary.

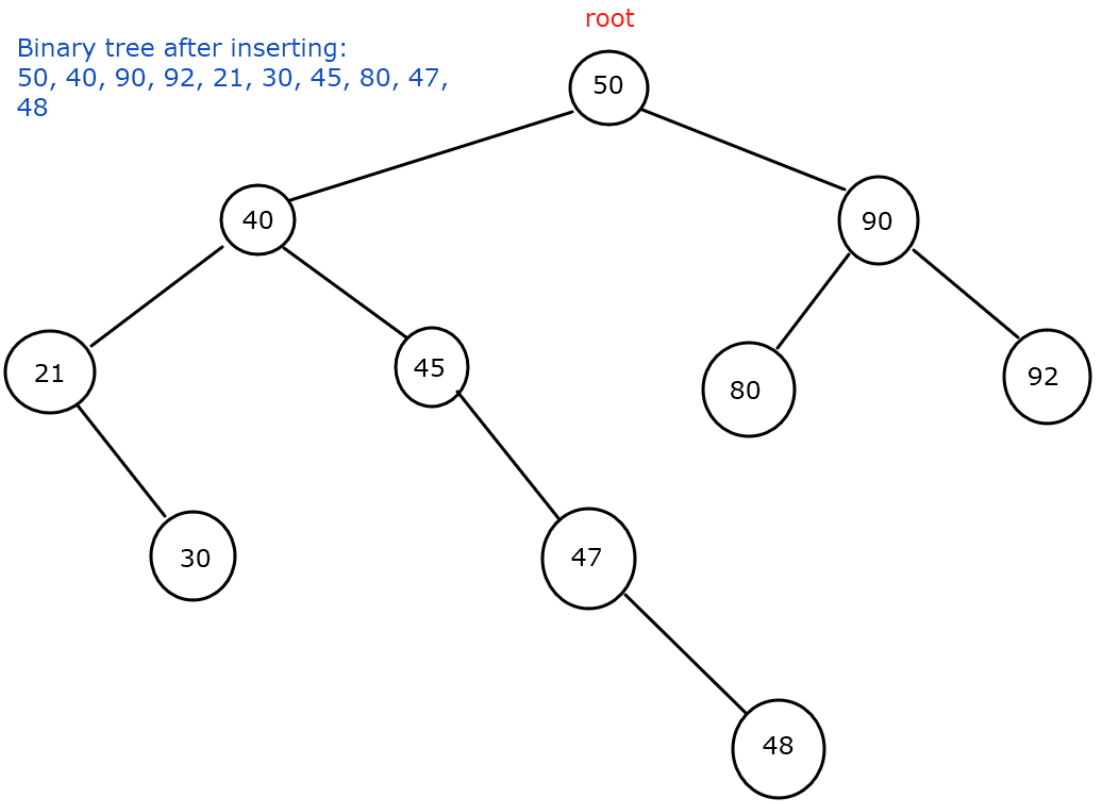
It's optional but nice to have a separate structure for the binary tree. A single `Node` pointer could be used to track the root of the binary tree but a node itself isn't really a binary tree but rather a single entity in the binary tree. Additionally, if size had to be kept track of then it makes much more sense to put everything in one structure.

Like with linked lists, if the binary tree is being implemented by using a `Node` pointer just to track the root then for any given operation that may update the root there are always two versions of the implementation. The **pass-by-value** version will pass the root pointer by value as an argument and update the root by returning a pointer to the updated root, and the **pass-by-reference** version will pass the root pointer by reference as an argument and update the root with assignment. It is important to always be able to understand and implement both versions for any given operation. For example, the two versions of adding to the tree are below:

```
// function declarations
Node* bst_insert_pass_by_value(Node* root, int data);
void bst_insert_pass_by_reference(Node** pRoot);

// function calls
Node* bst = NULL;
bst = bst_insert_pass_by_value(bst, 1);
bst_insert_pass_by_reference(&bst, 2);
```

Inserting Into a Binary Tree

The best case scenario for inserting into the binary tree has a time complexity of Ω(lgN). 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 isn't 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 have to be traversed for the item to reach its correct position to be inserted into at the lowest depth of the tree. In order to guarantee that the best case scenario always happens and have a worst case time complexity of O(lgN) an AVL tree must be used.



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/absolute value of the depth of its left subtree would be 2. So, the difference is 0 - 2 = -2. Since |-2| = 2 which isn't < 2, this would violate the self-balancing principle and the tree would need to rebalance. Rebalancing is done via *left rotations* and *right rotations*. Since an AVL tree is a tree its items are non-contiguous in memory because it's a linked data structure.

```
// 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;
    int size;  // optional
} AVL_tree;
```

Tracking the size means getting the size is an O(1) operation instead of O(N). Implement this as necessary.

It's optional but nice to have a separate structure for the AVL tree. A single `Node` pointer could be used to track the root of the AVL tree but a node itself isn't really an AVL tree but rather a single entity in the AVL tree. Additionally, if size had to be kept track of then it makes much more sense to put everything in one structure.

Like with binary trees and linked lists, if the AVL tree is being implemented by using a `Node` pointer just to track the root then for any given operation that may update the root there are always two versions of the implementation. The **pass-by-value** version will pass the root pointer by value as an argument and update the root by returning a pointer to the updated root and the **pass-by-reference** version will pass the root pointer by reference as an argument and update the root with assignment. It is important to always be able to understand and implement both versions for any given operation. For example, the two versions of adding to the tree are below:

```
// function declarations
Node* avl_tree_insert_pass_by_value(Node* root, int data);
void avl_tree_insert_pass_by_reference(Node** pRoot);

// function calls
Node* avl = NULL;
avl = avl_tree_insert_pass_by_value(avl, 1);
avl_tree_insert_pass_by_reference(&avl, 2);
```

Magnitude: The AVL tree is based on the principle that if the magnitude 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.

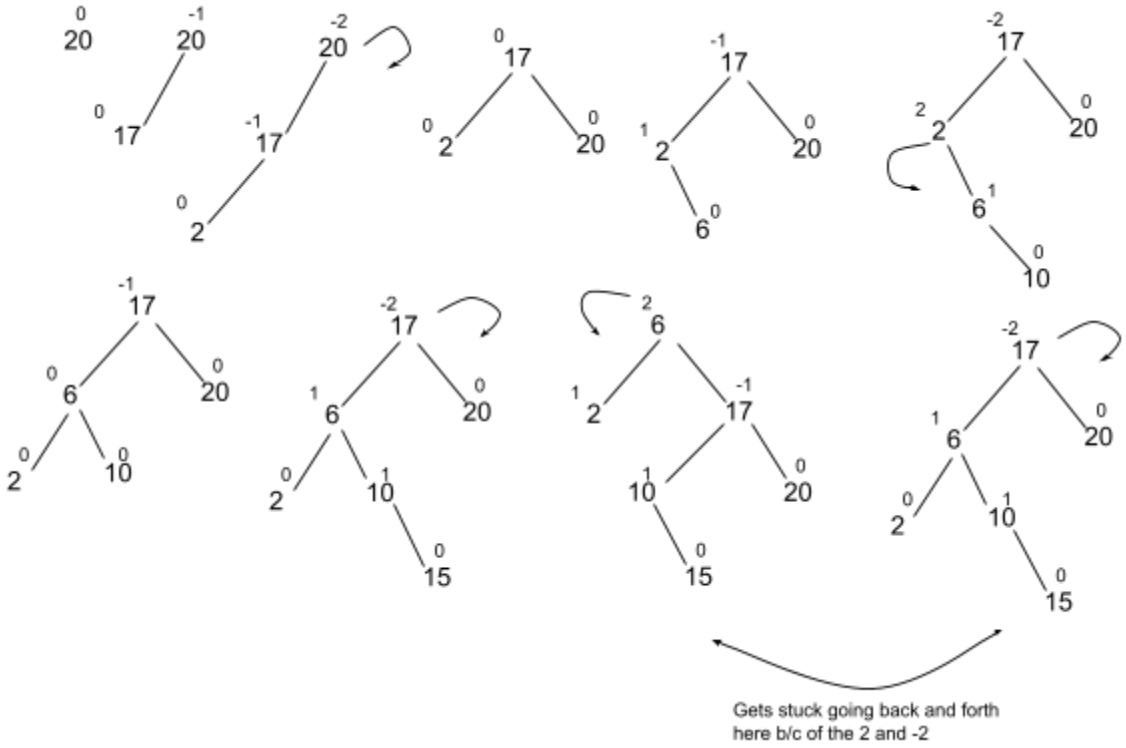
- A node in the tree is *left heavy* or *leans to the left* if the depth of its left subtree is greater than the depth of its right subtree which results in a balance factor of -1 or -2.
- A node in the tree is *right heavy* or *leans to the right* if the depth of its right subtree is greater than the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

- The parent is right heavy (2) but its right child is left heavy (-1) or the parent is left heavy (-2) but its 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 its child.

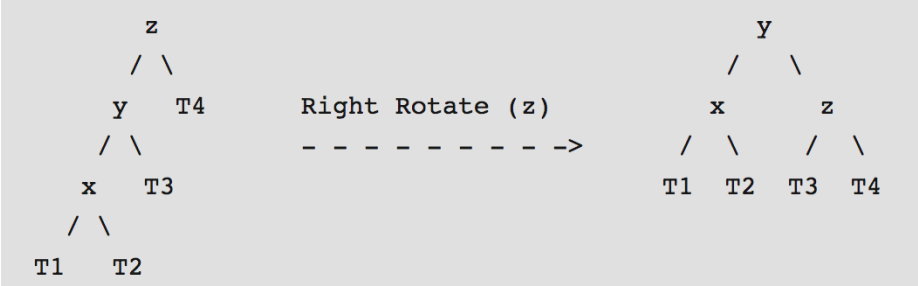
- The parent is right heavy (2) and its right child is also right heavy (1) or the parent is left heavy (-2) and its left child is also left heavy (-1).
- In these two situations the double rotation doesn't have to happen.

Simple Cases - One Rotation

The parent and its child lean the same way.

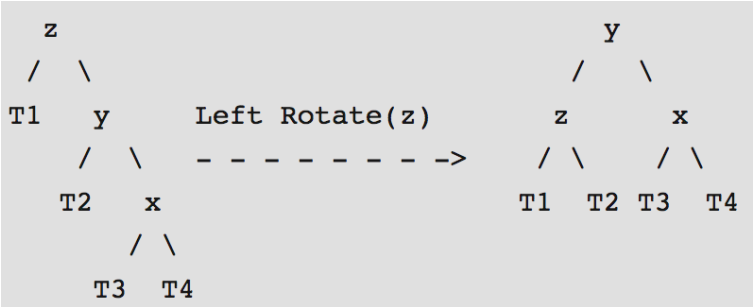
Left-Left: The parent (root) is left heavy (-2) and its left child is left heavy (-1). Perform a right rotation on the parent.

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



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.

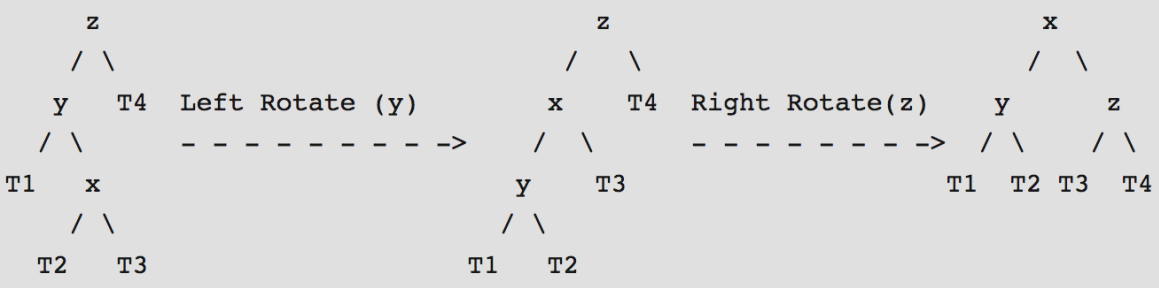


Complex Cases - Two Rotations

The parent and its child lean the opposite way.

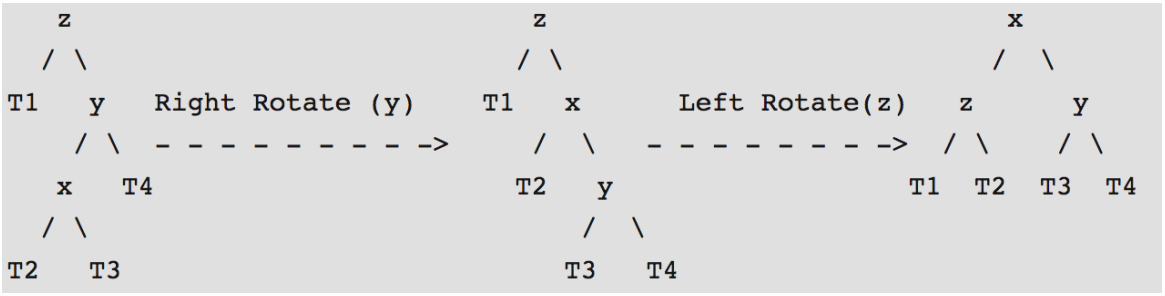
Left-Right: The parent is left heavy (-2) and its left child is right heavy (1). Perform a left rotation on the left child of the root and then perform a right rotation on the root.

- 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.



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.



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

Hash Table

Hash Table: An array where a **key** in the form of a string is used as an index. This associates each item directly with a string instead of directly with an index. Then, a **hashing function** takes the string and converts it into an index in a process called **hashing** the string. A hash table can be implemented using an array or a combination of an array and a linked list list as is described below. In the array implementation the items are contiguous in memory. In the linked list implementation the items are non-contiguous in memory because it's a linked data structure.

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, a hash table with a capacity of 1000 has indexes in the range of [0, 999]. Modular arithmetic must be used for the hashing function to always return an index within the range.
- It tries to avoid **collision** as much as possible which 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 is written to avoid collision as much as possible though it's not possible to avoid it entirely

Hash tables have two common implementations 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 an 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.

There is much more discussion to be had on how hash tables get implemented using either open addressing or separate chaining. This is just a general introduction to the concept 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 runtime 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 runtime 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 runtime of an algorithm purely from an abstract mathematical perspective in relation to N.

Note: As discussed in the introduction, Big-O time complexity doesn't actually mean worst case and it has a more precise mathematical definition in regards to an upper bound of an equation that would be studied in a more advanced algorithms course. In the discussion below there are times where the description of Big-O deviates from what it actually means. This is intentional because this document, and specifically this section on time complexity, is meant for someone learning about data structures and time complexity for the first time in which it's acceptable to think of Big-O as the worst case. Generally, when someone learns about data structures and time complexity they first learn the slightly incorrect definition of Big-O in an introductory data structures course because it's more conducive to learning, and then later on they learn the technically correct definition in a more advanced algorithms course.

The study of time complexity is about what happens to the runtime as N increases (goes out to infinity). It's about the rate of growth. For example, exponential time is slower than polynomial time because as N increases eventually the exponential runtime will surpass the polynomial runtime and the difference between the runtimes will increase as well. This means exponential time grows at a faster rate than polynomial time. However, this doesn't mean that there aren't smaller values of N for which this isn't 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 and the difference between 2^N and N^4 increases. This is why $O(N^4) < O(2^N)$. Time complexity can be visualized on a graph where the x-axis represents N and the y-axis represents the time. It's helpful to do this because it provides a clear picture on how the rates of growth compare to each other.

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's important to think about time complexity in regards to 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. This can be seen with hash tables and quicksort (discussed later on). 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 versions with increasing values of c)	…
$O(c^N)$	exponential time where c is some positive constant	…
O(N!)	factorial time	Slowest

$O(c)$ **constant time** means the amount of time an operation takes in the worst case isn't proportional to N but rather 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. Another way to think of this is that the operation always takes c units of time. It could be any number but traditionally a 1 is put there like O(1) for convenience. 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 runtime in comparison to all of the others because as N increases the runtime of the operation doesn't increase. Notice how the definition doesn't actually have anything to do with speed. It just very specifically says that N doesn't affect the runtime. Hypothetically, there could be a constant time operation that takes a long time to complete like if $c = 100$ trillion. However, in practice constant time operations are generally fast and c would not be that high. For example, inserting at the head of a linked list which is a constant time operation is only a few steps. Regardless, what really matters is that the runtime doesn't increase as N gets larger. So, even if hypothetically c were that high, eventually the runtimes of all the other time complexities will surpass it as N increases.

O(lgN) **logarithmic time** 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 $T(N) = lgN$. Another way to think of this is that the operation takes lgN units of time to complete in the worst case. As discussed previously lgN really means floor(lgN) which means the result is rounded down to the nearest integer. For example, lg10 = 3 because lg10 is approximately 3.3 which gets rounded down to 3.

O(N) **linear time** 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 takes N units of time to complete in the worst case.

O(NlgN) **log-linear time** means the amount of time an operation takes in the worst case is log-linearly proportional to N. On a graph this is the equation T(N) = NlgN. Another way to think of this is that the operation takes NlgN units of time to complete in the worst case. This combines linear and logarithmic time together in the form of the equation NlgN which is multiplying N by lgN.

$O(N^2)$ **quadratic time** 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 takes N^2 units of time to complete in the worst case.

$O(N^C)$ **polynomial time** means the amount of time an operation takes in the worst case is polynomially proportional to N for some polynomial equation of degree *c* with base N 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 takes N^C units of time to complete in the worst case. Linear and quadratic are both versions of polynomial time with degrees of 1 and 2 respectively. The runtime increases as *c* increases meaning $N^2 < N^3 < N^4$…

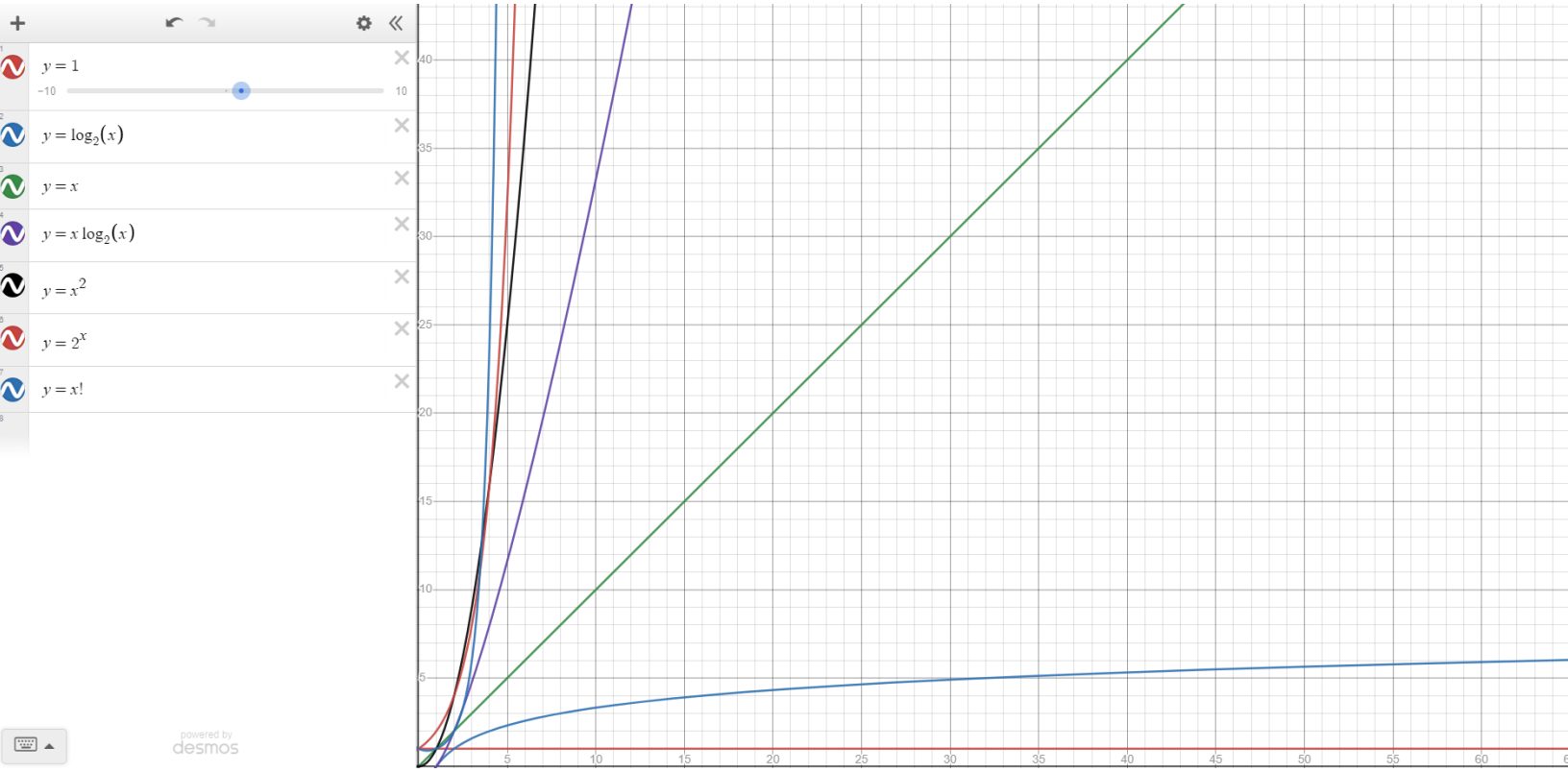
$O(c^N)$ **exponential time** means the amount of time an operation takes in the worst case is exponentially proportional to N for some exponential equation of degree N with base *c* where *c* is some positive constant. On a graph this is the equation $T(N) = c^N$. Another way to think of this is that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

O(N!) **factorial time** means the amount of time an operation takes in the worst case is factorially proportional to N. On a graph this is the equation T(N) = N!. Another way to think of this is that the operation takes N! units of time in the worst case. N! means take the product of all integers starting at N down to 1. For example, if N = 5 then $N! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

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

O(1)	1 unit of time	constant	*Fastest*
O(lgN)	3 units of time	lg10	…
O(N)	10 units of time	10	…
O(NlgN)	30 units of time	10 x lg10	…
$O(N^2)$	100 units of time	10^2	…
$O(2^N)$	1024 units of time	2^{10}	…
O(N!)	3628800 units of time	10!	Slowest

How the rates of growth compare to each other can be seen on the graph below.
As expected, O(1) < O(lgN) < O(N) < O(NlgN) < $O(N^2)$ < $O(2^N)$ < O(N!) as N increases.



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

The Big-O time complexities for all of the data structures discussed are listed starting on the next page.The *DS specific name* section shows the special names that some data structures use for generic operations i.e. some data structures use another name other than *insert* for inserting a new item. These are the names that could be given to functions in the implementation. If it's blank it means there is no special name for it in the data structure and the generic name can be used.

Notes

The following are all constant time operations: accessing an index in an array, accessing the head in a linked list and the tail if it's being kept track of, and accessing the root of a tree. This is being mentioned here because the time complexities of many operations are what they are because of this. For example, the *head_insert* operation in a linked list is constant time because accessing the head is constant time.

Unlike the other two fundamental data structures - arrays and linked lists - there are no time complexities for trees in general because every type of tree is unique.

Amortized analysis: https://en.wikipedia.org/wiki/Amortized_analysis
Regular Big-O analysis examines the worst case when an operation is performed one single time. Amortized analysis examines time complexity from a different perspective which is when an operation is repeatedly done over a period of time and calculates the average runtime of an operation in that context. The $^+$ symbol is used to denote amortized time. For example, pushing back into a vector is O(N) only on the rare occasion that a resize operation happens but in all other cases it's guaranteed to be O(1). The amortized analysis of this operation shows that pushing back is $O(1)^+$ when done repeatedly. Consider the example where an empty vector has a capacity of 1 million items. The first 1 million push back operations are all guaranteed to be O(1). That's quite a lot of push backs that are O(1). So, saying that push back is O(N) does not accurately depict what happens most of the time. That's not to say that it doesn't matter that it's O(N) because there are many situations where it could matter especially in real time systems where performance is especially important and that one time that the push back operation is O(N) actually could be a problem. Amortized analysis is just helpful to use in conjunction with Big-O analysis to paint a bit more of an accurate overall picture. It is used in this document in the data structures for which an array can be used in their implementation.

Array/Vector (unordered)

Note: Time complexities for the ordered version are not listed because keeping items in order is not what a vector does (an array could do this just not a vector). Hypothetically it could be implemented this way but it's just not what is meant by a traditional vector.

Common Operations	DS specific name	Location	Time
insert	vector_push_back	back	O(N) or $O(1)^+$
insert		anywhere	O(N)
remove	vector_pop_back	back	O(1)
remove		anywhere	O(N)
access	vector_at	anywhere	O(1)
search (specific item)		anywhere	O(N)

- *insert (back):* insert at index [size].
  - O(N) because a resize operation may happen but all other cases are O(1).
  - $O(1)^+$ is the amortized time.
- *insert (anywhere):* insert at any index.
  - O(N) because a resize operation may happen and because even if a resize operation doesn't happen any items at indexes greater than or equal to the index where the item is being inserted into have to shift over to the right by 1 index which at most is N items if the item is added to index 0.
- *remove (back):* remove from index [size - 1].
- *remove (anywhere):* remove an item from any index - any items at indexes greater than the index of the item being removed have to shift over to the left by 1 index which at most is N items if the item being removed is at index 0.
- *access:* access any index.
- *search:* all N indexes are searched if the item doesn't exist or is at the back.

Linked List (unordered and unordered)

Notes: There are no special names for some linked list operations with the exception of prefixing the location of the operation. For example, *head_insert* is used to refer to inserting an item at the head. Additionally, the time complexities are all the same for an ordered and unordered linked list with the exception of insert which will be specified. For an ordered linked list, there is no head/tail insert because the item has to go into the correct position in the linked list. Similarly, for an unordered linked list there are only head and tail inserts.

Common Operations	DS specific name	Location	Time
insert (unordered)		head	O(1)
insert (unordered, tail tracked)		tail	O(1)
insert (unordered, tail not tracked)		tail	O(N)
insert (ordered)		anywhere	O(N)
remove		head	O(1)
remove (tail tracked)		tail	O(1)
remove (tail not tracked)		tail	O(N)
remove (specific item, tail tracked/not tracked)		anywhere	O(N)
access		head	O(1)
access (tail tracked)		tail	O(1)
access (tail not tracked)		tail	O(N)
access (specific item, tail tracked/not tracked)		anywhere	O(N)
search (specific item, tail tracked/not tracked)		anywhere	O(N)

- *insert (unordered, head):* the head is tracked.
- *insert (unordered, tail tracked):* the tail is tracked.
- *insert (unordered, tail not tracked):* all N nodes are traversed to reach the tail.
- *insert (ordered):* all N nodes are traversed if the item ends up at the tail.
- *remove (head):* the head is tracked.
- *remove (tail tracked):* the tail is tracked.
- *remove (tail not tracked):* all N nodes are traversed to reach the tail.
- *remove (specific item):* all N nodes are traversed if the item doesn't exist, if it's at the tail and the search starts at the head, or if it's at the head and the search starts at the tail if the tail is being tracked.
- *access (head):* the head is tracked.
- *access (tail tracked):* the tail is tracked.
- *access (tail not tracked):* all N nodes are traversed to reach the tail.
- *access (specific item):* all N nodes are traversed if the item doesn't exist, if it's at the tail and the search starts at the head, or if it's at the head and the search starts at the tail if the tail is being tracked.
- *search (specific item):* all N nodes are traversed if the item doesn't exist, if it's at the tail and the search starts at the head, or if it's at the head and the search starts at the tail if the tail is being tracked.

Stack

Common Operations	DS specific name	Location	Time
insert (array)	stack_push	top	O(N) or $O(1)^+$
insert (linked list)	stack_push	top	O(1)
remove	stack_pop	top	O(1)
access	stack_top	top	O(1)

- *insert (array):* insert at index [size].
    - O(N) because a resize operation may happen but all other cases are O(1).
    - $O(1)^+$ is the amortized time.
- *insert (linked list):* insert at the head, linked lists don't have resize operations.
- *remove/access:* in an array this is removing/accessing at index [size - 1] and in a linked list this is removing/accessing the head.

FIFO Queue

Common Operations	DS specific name	Location	Time
insert (array)	queue_enqueue	back	O(N) or $O(1)^+$
insert (linked list)	queue_enqueue	back	O(1)
remove	queue_dequeue/service	front	O(1)
access	queue_front	front	O(1)

- *insert (array):* insert at index [back + 1] or [0] depending on the situation in a wraparound queue.
    - O(N) because a resize operation may happen but all other cases are O(1).
    - $O(1)^+$ is the amortized time.
- *insert (linked list):* insert at the tail, linked lists don't have resize operations.
- *remove/access:* in an array this is removing/accessing at index [front] and in a linked list this is removing/accessing the head.

Priority Queue - Heap

Commons Operations	DS specific name	Location	Time
insert (array)	pqueue_enqueue	based on priority	O(N) or $O(lgN)^+$
insert (tree)	pqueue_enqueue	based on priority	O(lgN)
remove (highest priority)	pqueue_dequeue/service	front	O(lgN)
access (highest priority)	pqueue_front	front	O(1)
merge			O(NlgN)

- *insert (array):* insert at an index based on priority level - at most lgN indexes are traversed/at most lgN fix-up operations happen.
  - O(N) because a resize operation may happen but all other cases are O(lgN).
  - $O(lgN)^+$ is the amortized time.
- *insert (tree):* insert at a node position based on priority level - at most lgN nodes are traversed/at most lgN fix-up operations happen, trees don't have resize operations.
- *remove:* in an array this is removing from index [0] and in a tree this is removing from root - at most lgN indexes/nodes are traversed/at most lgN fix-down operations happen.
- *access:* in an array this is accessing at index [0] and in a tree this is accessing the root.
- *merge:* remove an item from one heap (lgN), insert it into the next heap (lgN), do this for N items in that one heap (N), so that'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	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(1)

- *insert:* insert at a node position based on priority level - at most lgN tree merges.
- *remove:* remove the root - at most lgN tree merges.
- *access:* access the root.
- *merge:* the items at the root 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.

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/remove/search:* the binary tree is not self-balancing so there are two versions of the worst case - there is a tree that is a diagonal line going down to the left/right because all items inserted have been less than/greater than all items previously inserted and the item being added, removed, or searched for is smaller/larger than all other items in the tree - in either case, all N nodes are traversed.

AVL Tree

Note: The rotations in AVL trees are constant time operations so they don't affect the time complexities of the insert/remove operations because constants are ignored in time complexities since they're negligible.

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

- *insert/remove/search:* the AVL tree is self-balancing so for all operations at most lgN nodes are traversed/a depth of lgN has to be gone down to.

Hash Table

Note: The worst case scenarios can easily be avoided with a good implementation so the average time complexities are also shown because this is easily achievable in practice.

Common Operations	Location	Time (Average)	Time (Worst)
insert	could end up anywhere	Θ(1)	O(N)
remove	could be anywhere	Θ(1)	O(N)
search (some specific item)	could be anywhere	Θ(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's reasonable to achieve constant time operations.

Sorting Algorithms

Listed below are some of the most common sorting algorithms and how they implement the sorting process should be fully understood.

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

Other good sorting algorithms to know are.

- radix sort
- counting sort
- bucket sort

The first sorting algorithms listed - 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 it seems counterintuitive a non-comparison based sorting algorithm will sort items without ever actually comparing them.

Quick sort, shell sort, heap sort, and merge sort all have an average runtime of $\Theta(NlogN)$ but in practice quick sort actually performs better than the other 3. Additionally, it has been proven mathematically that any comparison based sorting algorithm can't 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's 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 isn't 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.

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 doesn't 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 doesn't 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 doesn't belong on the right,
L: there's nothing that doesn't 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 doesn't belong on the right pivot
L: 5 doesn't 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 doesn't belong on the right pivot
L: 6 doesn't 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 doesn't belong on the right pivot
L: 7 doesn't 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 doesn't 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 doesn't belong on the right pivot
L: 3 doesn't 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 doesn't 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's 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 doesn't 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 1.3×10^{30}.
However, 100! is so much larger and is approximately 9.3×10^{157}.
As a fraction this is $\frac{1.3 \times 10^{30}}{9.3 \times 10^{157}}$.

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's 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 items and then pick the median of them.

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	$\Omega(N)$	$\Theta(NlogN)$	$O(NlogN)$
Heap Sort	$\Omega(NlogN)$	$\Theta(NlogN)$	$O(NlogN)$
Quick Sort	$\Omega(NlogN)$	$\Theta(NlogN)$	$O(N^2)$
Merge Sort	$\Omega(NlogN)$	$\Theta(NlogN)$	$O(NlogN)$