Data Structures

The list below contains all of the data structures discussed in this document - explanations are discussed in the context of the C programming language.

- **Vectors**
- **Linked Lists**
    - Singly Linked Lists
    - Doubly Linked Lists
    - Circularly Linked Lists
- **Stacks** - Array-based and Linked List-based implementations
- **Regular Queues** - Array-based and Linked List-based implementations
- **Priority Queues**
    - Heaps - Array-based implementation with conceptualization as a tree
    - Binomial Heaps/Queues - Conceptual visualization
- **Trees**
    - Binary Trees
    - AVL Trees
- **Hash Tables**
    - Open Addressing
    - Separate Chaining

Notes:

- Addresses are unsigned integers in a computer's memory. For the purposes of demonstration, the diagrams shown in this document will use uppercase letters as addresses.
- A data structure can be used for any type of data. For the purposes of demonstration, the examples shown in this document will use integers.

Time complexities of data structures are discussed at points, but the main point of this document is to give conceptual explanations of data structures and provide visual representations. There are many available resources online that give comprehensive overviews of the time complexities of all the data structures such as this website: https://www.bigocheatsheet.com/

Vectors

Vector: An array that can dynamically increase its capacity. A vector is identical to a static array like:

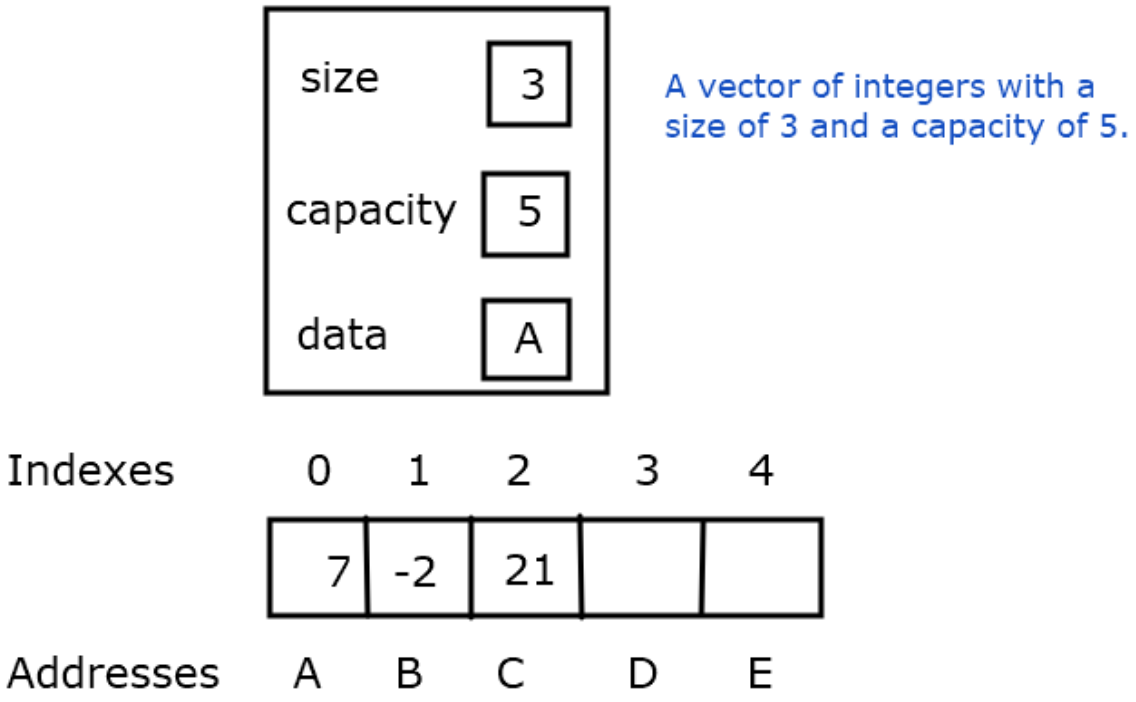
```
int a[10];
```

The only difference is a static array's capacity (in this case 10) cannot change.

A vector must have the following components:

- **size** - The amount of items in the array (initially 0). The index of the next available position is always [size], and the index of the most recently added item is [size - 1].
- **capacity** - The amount of items the array can hold. If the size equals the capacity when adding a new item, a resize operation on the array must first be performed. The array can be resized in any way, and a common convention is to double the capacity.
- **array** - The actual array, called *data*, in this example.

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



Linked Lists

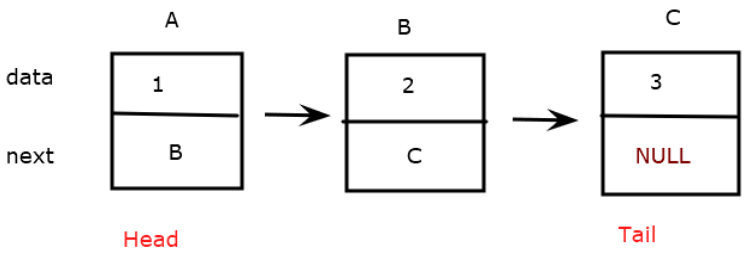
Linked List: A series of nodes all connected to each other via node pointers. It is a *list* of nodes that are all *linked* to each other, hence the name *linked list*. The first node in the list is conventionally called the **head**, and the last node in the list is conventionally called the **tail**.

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

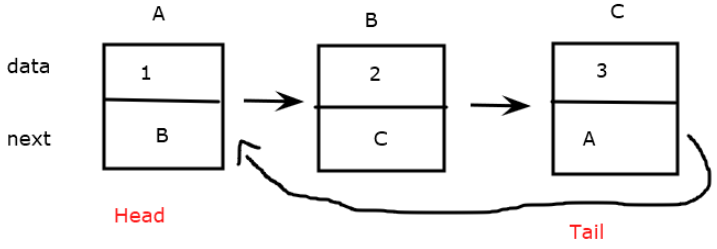
```
// singly linked list
typedef struct node Node;
struct node {
    int data;
    Node* next;
};
```

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

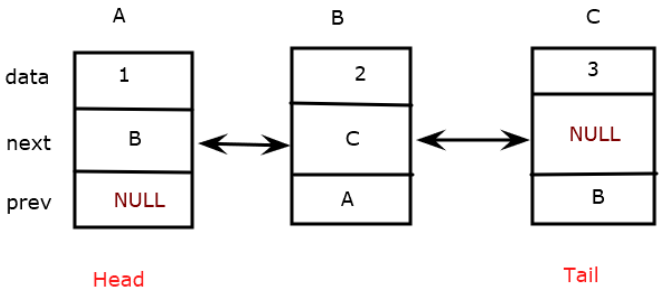
Singly Linked List



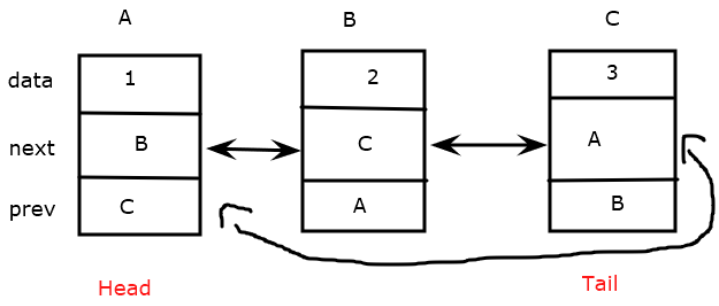
Circular Singly Linked List



Doubly Linked List



Circular Doubly Linked List



Stacks

Stack: A series of data items where items are added to the top and removed from the top such as a stack of plates.

- A stack can be implemented using an array or linked list.

// array implementation

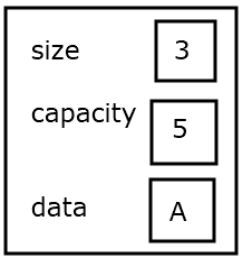
The structure is the same as the vector.

// linked list implementation

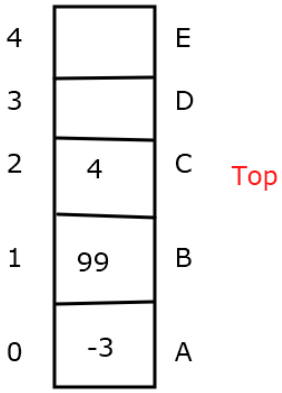
Use one of the linked list structures on page 3..

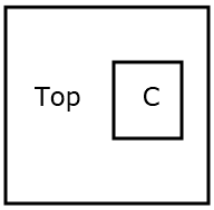
Then create an additional structure like below

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

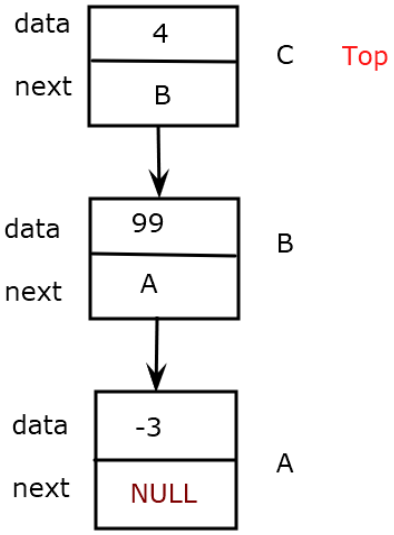


Vector implementation of a stack with a size of 3 and a capacity of 5





Singly linked list implementation of a stack a size of 3



Regular Queues

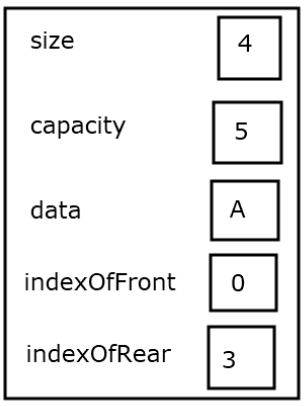
Regular Queue: A queue where the arrangement of items is based on the order in which the items entered the queue.

- **Front:** Contains the least recently added item and is where items are removed.
- **Rear:** Contains the most recently added item and is where items are added.
- So, items are inserted at the rear of the queue and removed from the front of the queue such as in a line of people waiting to get served at a store's cash register.
- A regular queue can be implemented as an array or a linked list.
- The word *enqueue* is used to refer to adding to the queue, and the words *dequeue* or *service* are used to refer to removing from the queue

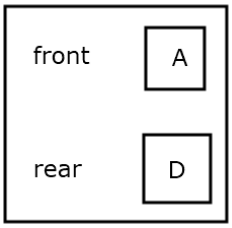
```
// array implementation
typedef struct queue {
    int size;
    int capacity;
    int* data;
    int indexOfFront;
    int indexOfRear;
} Queue;
```

```
// linked list implementation
Use one of the linked list structures on page 3.
Then, create an additional structure like below
typedef struct queue {
    Node* front;
    Node* rear;  // or Node* back
} Queue;
```

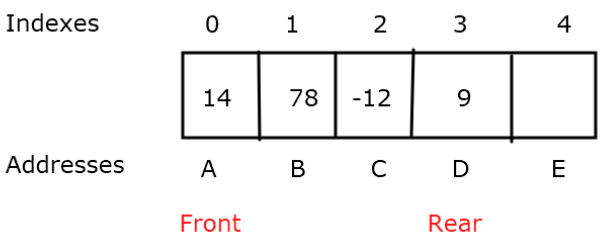
Note: In the array implementation, there is a way of doing it that doesn't require both an indexOfFront and indexOfRear. This is done by using some basic mathematical calculations involving modular arithmetic. Also, the array implementation should be a *wraparound queue* to save memory space. For example, if the front were index 1 and the rear were index 4, when adding a new item the rear would become index 0.

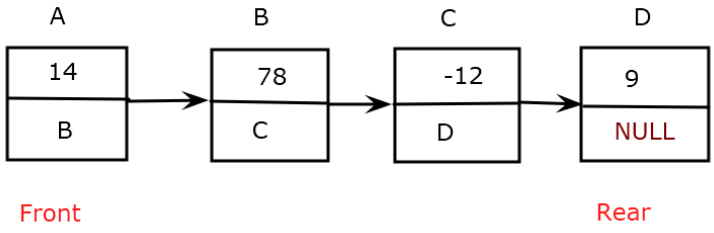


Vector implementation of a regular queue with a size of 4 and a capacity of 5



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





Priority Queues - Heaps

Priority Queue: A queue where the arrangement of the items in the queue is based on their priority, for example a number. Higher numbers could have a higher priority, or lower numbers could have a higher priority. Regardless, the arrangement has nothing to do with the order in which the items entered the queue. Priority queues require an understanding of trees - if this is not had, read the section *Trees - Binary Trees* first.

- **Front:** Contains the item with the highest priority and is where items are removed.
- **Back:** There is no back like a regular queue since there is no default position that an item goes to when it's added. When an item is added to the priority queue, the position it goes to in the queue is based on its priority. If it's the highest priority item, it goes to the front. Else, it finds its appropriate place within the queue based on its priority.
- **Heap:** A priority queue can be implemented as a **heap**. Note that a priority queue can also be implemented with a more advanced data structure called a **binomial heap** (also referred to as a **binomial queue**). This is discussed in a later section. For now, the information below refers to regular heaps.
    - Heaps are conceptualized as trees but are in fact most easily implemented using vectors.
    - Heaps are more complicated than the data structures discussed previously. In the following pages on heaps, all of the diagrams must be viewed in conjunction with each other to gain a full understanding. The first diagram shows a standstill snapshot of a heap at some given moment. The following two diagrams demonstrate adding to the heap and removing from the heap. Understanding how heaps keep items in the correct priority requires viewing all three diagrams.

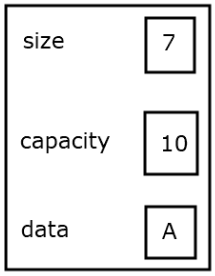
```
// array implementation - 2 structures required
typedef struct data_priority_pair {
    int data_item;
    int priority_level;
} Data_priority_pair;
```

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

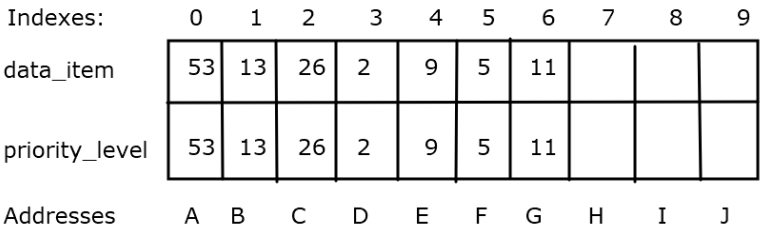

**** Note ****

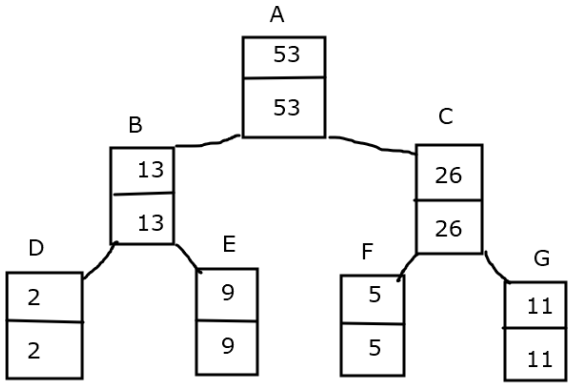
- This is an example of a **max-heap** - a priority queue where higher numbers have a higher priority. That doesn't always have to be the case. There could also be a **min-heap** where lower numbers have a higher priority
- For the purposes of making the demonstration more simple, the data item and priority level are the same meaning 53 also has a priority level of 53. What matters though, is the priority level and not the data item. For example, if 53 had a priority level of 80 and 13 had a priority level of 90, then 13 would be at the front of the priority queue because 90 > 80.

Heap Diagram 1: Visualization of the heap



Priority queue implemented as vector-implementation of a heap.





The corresponding conceptual visualization of the heap as a tree

Heap Properties:

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

- Use the following formulas to calculate the indexes of the parent, right child, and left child nodes where k = index of current item.
    - parent index: (k - 1)/2 left child index: 2k + 1 right child index: 2k + 2

  Take 13 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

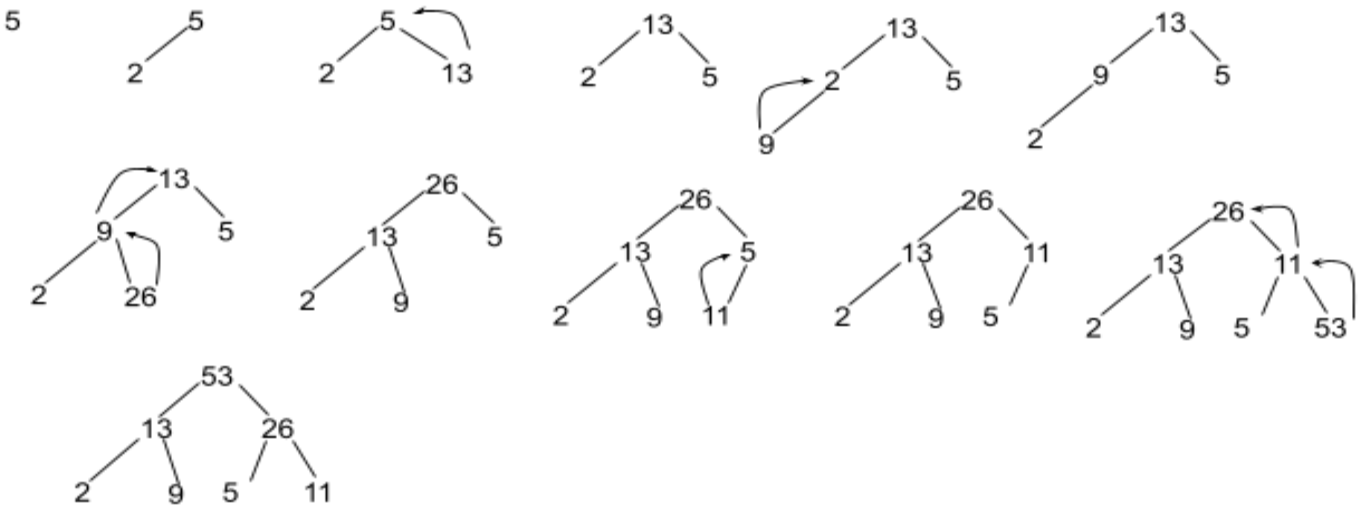
Heap Operations

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

Insert/Add/Enqueue: Insert the item at the next available position, and then fix up until it's in the correct position. Fix up means the item will swap with the parent if it has a higher priority than the parent.

- Cost of insert is lgN (this really means floor(lgN) but it's referred to as lgN.
    - i.e. to insert a 15th item, this will be at most lg15 which equals 3. It equals 3 because 2^4 equals 16 which is larger than 15. The next lowest exponent is 3 for 2^3

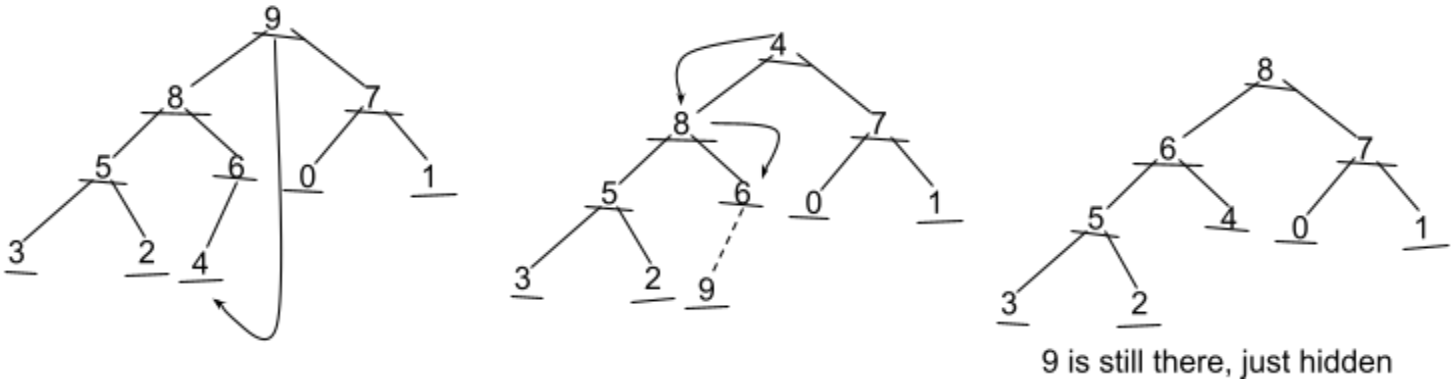
Heap Diagram 2: Adding to the heap - each number is also its own priority level.



Remove/Dequeue: Swap the first and the last element with each other. Then, remove the last element (which was previously the first element) from consideration, and fix down the first element (which was previously the last element) until it's in the correct position. Fix down means the following:

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

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



Merging Two Heaps (NlgN): Remove an element from one heap (lgN), insert it into the next heap (lgN), and do this for N elements in that one heap so it's 2NlgN which is NlgN

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

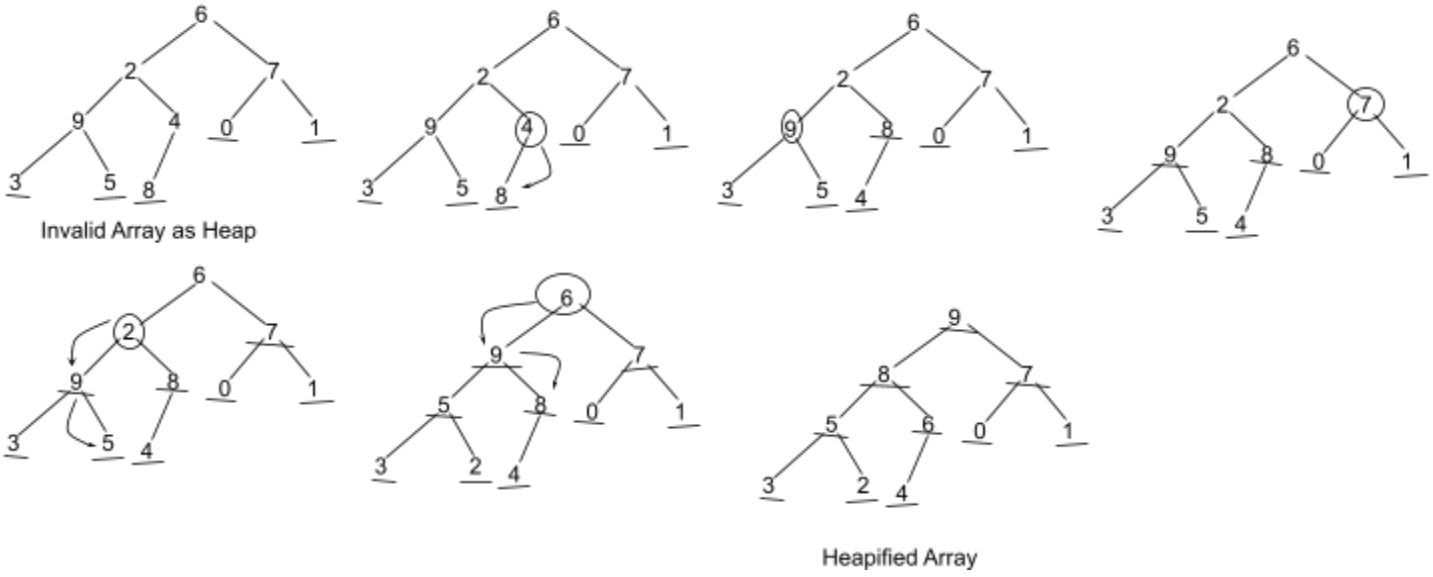
Turning an array into a heap

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

More About Heaps - Heapify

Heapify: Turn an array that's an invalid heap into a heap

- the array starts as an invalid heap with things randomly inserted.
- all the items in the lowest level are called **leaves** since they're at the end of the branch of the tree. The leaves have no children.
    - The leaves are all **valid** heaps since they fit the heap property that they are larger than their children (even though they don't actually have children, since they don't have any they are technically larger than their children).
- Begin the sorting process at the first non-leaf which is the first potential non-heap. If given size, the first non-leaf is the halfway point. So, index of first non-leaf is (size / 2 - 1)...-1 because indexes are 1 smaller than size. In this example, 4 is the first non-leaf node
    - All of 4's children are heaps. Call fixdown on 4 since 8 is larger than 4.
    - Now 9 is the first potential non-heap.
- Examine 9 - 9 is a valid heap since it's larger than all of its children. 7 is the next potential non-heap.
- Examine 7 - 7 is a valid heap since it's larger than all of its children. 2 is the next potential non-heap.
- Examine 2 - It is not a valid heap since it's not larger than all of its children
    - Call fix down on 2 and fix it down until it's in the proper position. When choosing among children, choose the higher priority child. First that's 9, then 5. Now 6 is the next potential non-heap
- Examine 6 - 6 is not a valid heap since it's not larger than all of its children.
    - Call fix down on 6. Once again, it fixes down until it's in the proper position
- Index 0 has been reached so the array has been heapified



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

Priority Queues - Binomial Heaps/Queues

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

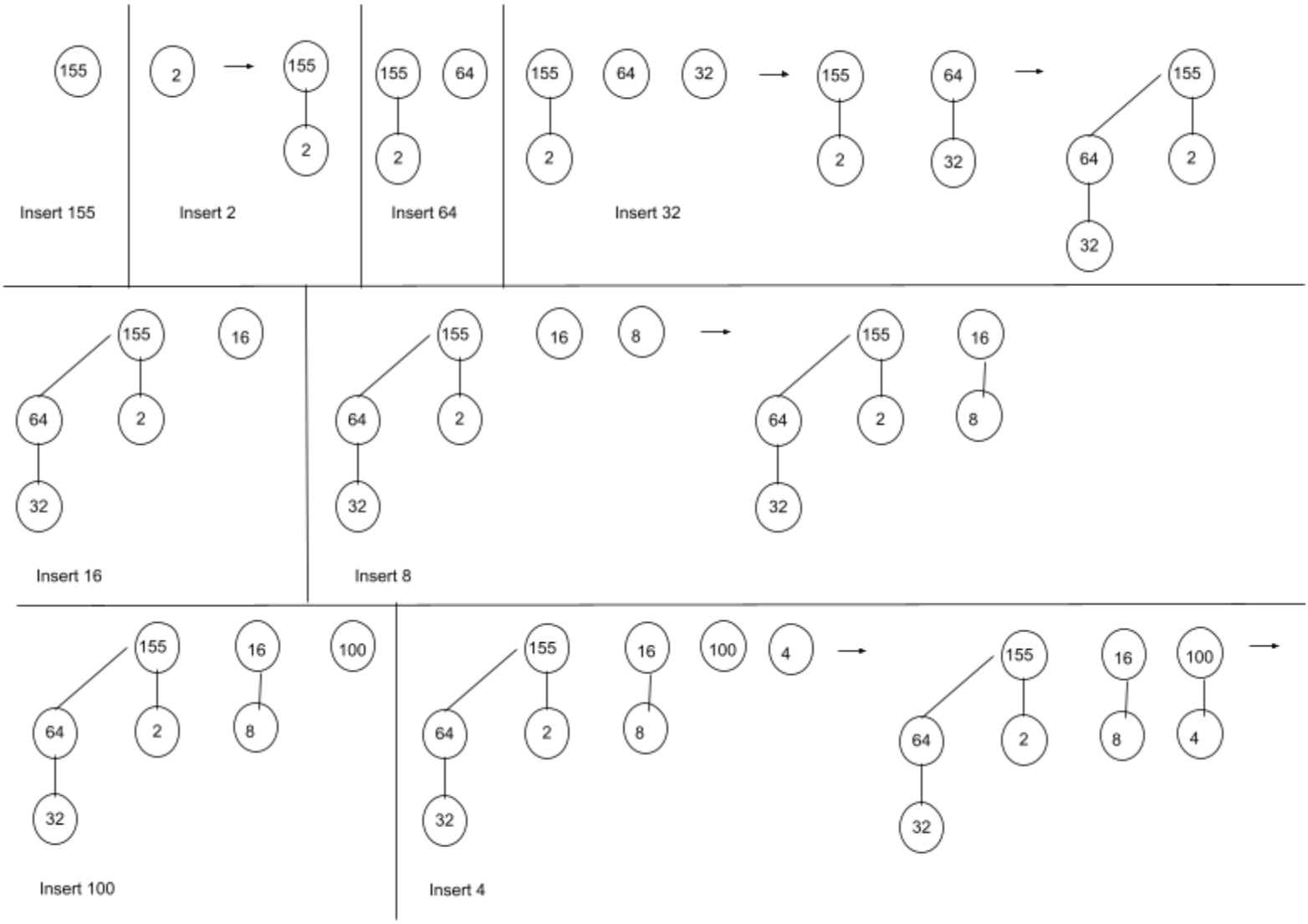
- When two trees of the same size exist, they combine and the higher priority root "wins" meaning they combine into one tree and the higher priority root becomes the root of the combined tree.
- When two trees combine together, the tree that "lost" goes under on the left most side of the tree that "won"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

Operations:

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

Binomial heaps and binary

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



Trees - Binary Trees

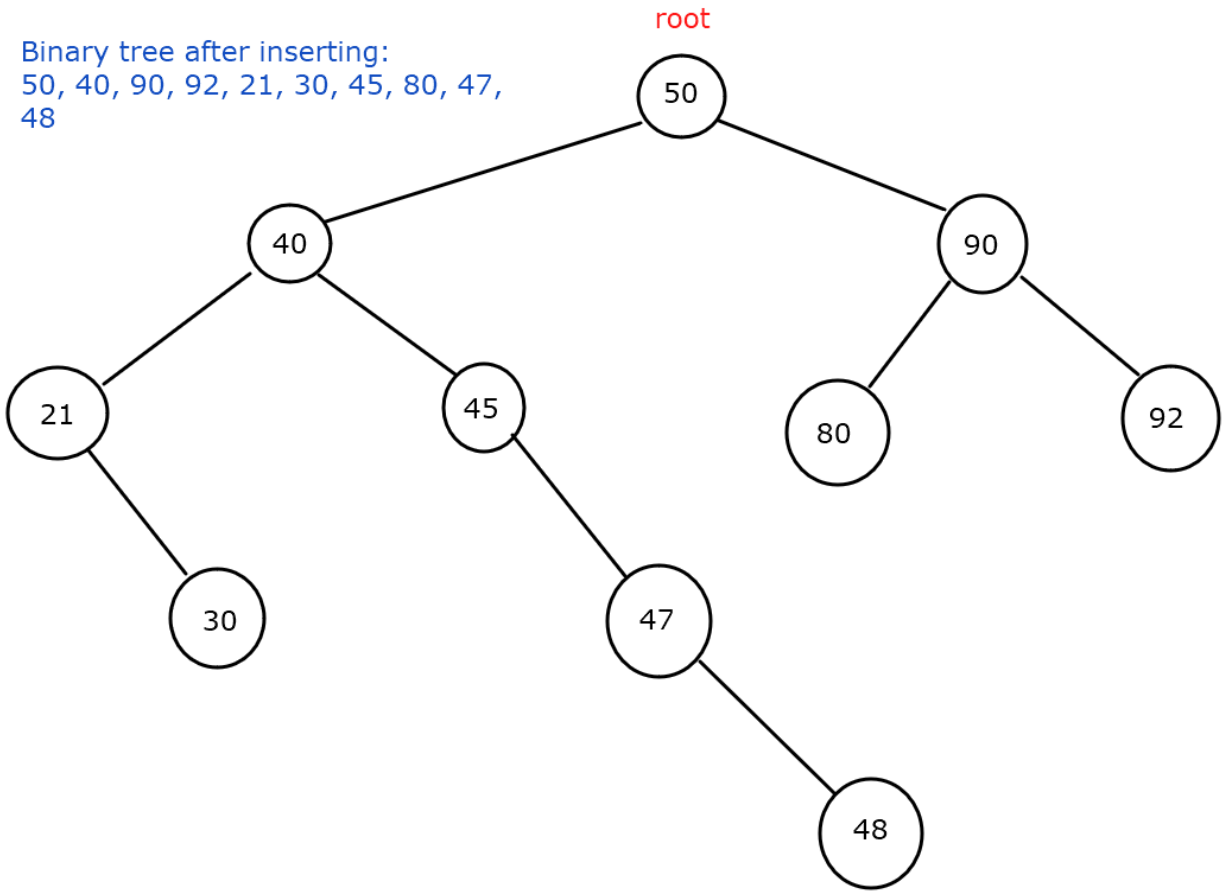
Tree: In general, a tree is a series of nodes starting at the root node where every node has a parent node (except for the root node) and child nodes.

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

Binary 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.

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

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



Navigating Trees

Pre-order traversal: SLR (**self** left right)

- each node visits itself first, then its left subtree and then its right subtree,
- used for copying a tree.
- *memorization technique*: pre - self comes before left and right, pre means before

In-order traversal: LSR (left **self** right)

- each node visits its left subtree, then itself, then its right subtree.
- used for print things in the tree in order
- *memorization technique*: in - self is in between/in the middle of left and right

Post-order traversal: LRS (left right **self**)

- each node visits its left subtree, its right subtree, then itself
- used to delete a tree
- *memorization technique*: post - self comes after left and right, post means after

Using the example of the binary tree on the previous page, the order in which each item from the tree would be printed using the three traversal techniques would be as follows

- *pre-order traversal:* 50, 40, 21, 30, 45, 47, 48, 90, 80, 92
- *in-order traversal:* 21, 30, 40, 45, 47, 48, 50, 80, 90, 92
- *post-order traversal:* 30, 21, 48, 47, 45, 40, 80, 92, 90, 50

Inserting into the tree:

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

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

Trees - AVL Trees

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

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

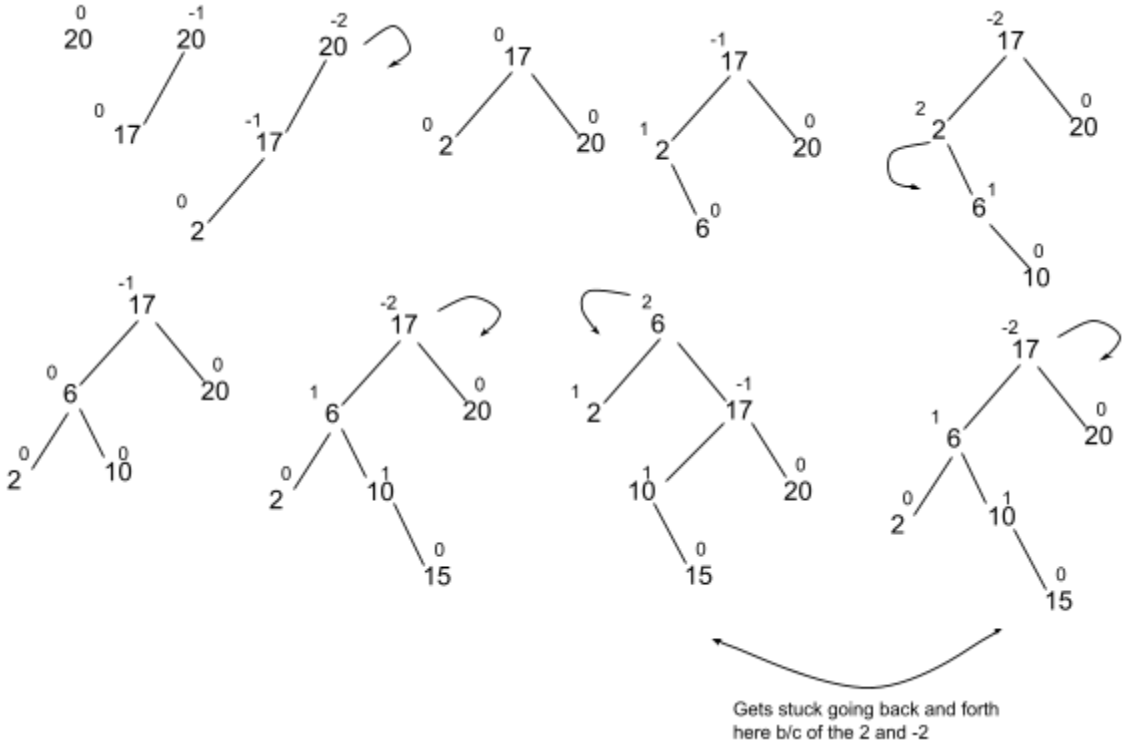
The following link provides code for an implementation of an AVL tree for integers. The logic of this code can be extrapolated to create an AVL tree for any data type.
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

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

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

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

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



Solution to this problem

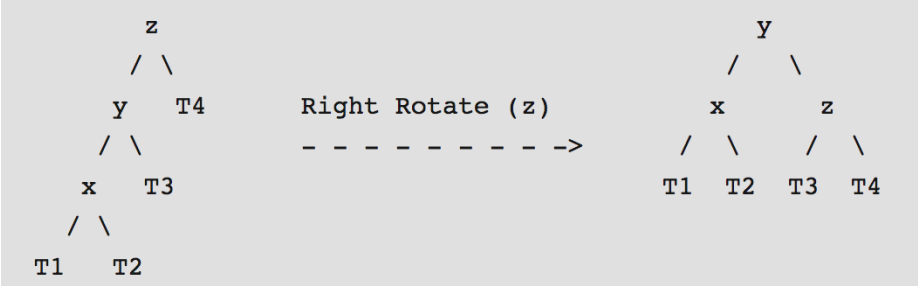
The tree gets "stuck" as seen in the above example when the children "lean" the opposite way of the parent. This means, for example, the parent is right heavy (2) but it's right child is left heavy (-1) or the parent is left heavy (-2) but the left child is right heavy (1). In these two situations, the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

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

Simple cases: parents and children lean the same way

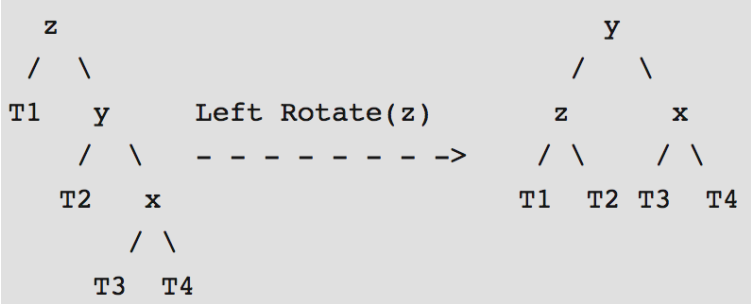
Left-Left: parent (root) is left heavy (-2), left child is left heavy (-1)

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



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

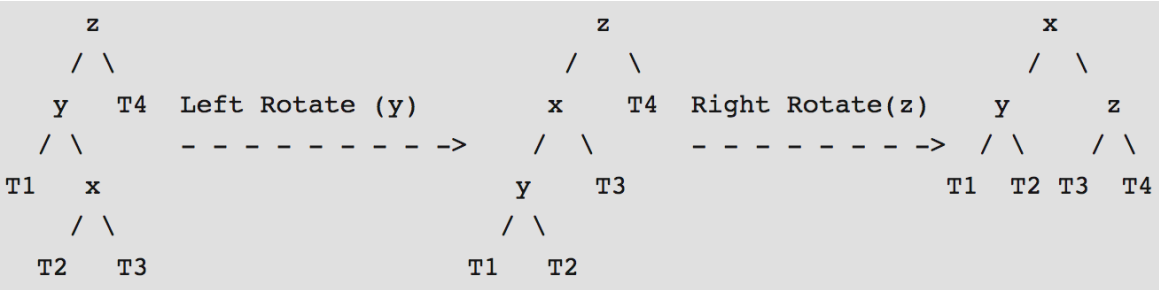
- perform a left rotation on the parent (root).



Complex Cases: parent and children lean opposite ways

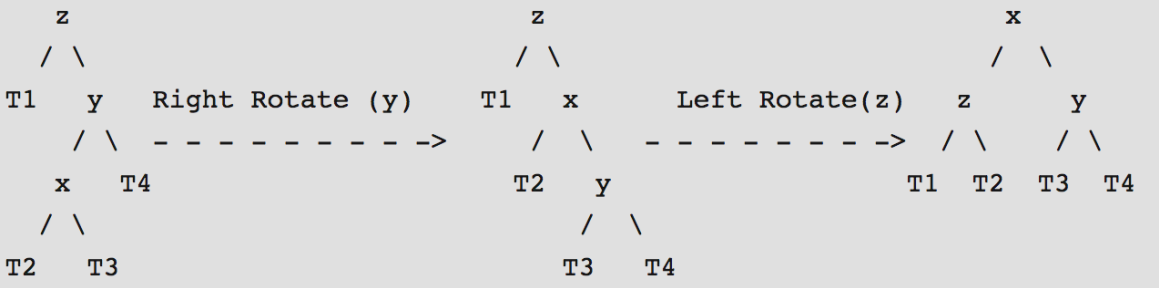
Left-Right: parent is left heavy (-2), left child is right heavy (1)

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



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

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



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

Hash Tables

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

- A hashing function can be written in an infinite number of ways, but every hashing function has two things in common:
    - First, it involves modular arithmetic. Since the hash table is an array, there is a restricted range of valid indexes. For example, if the hash table has a capacity of 1000, then the valid indexes are [0, 999]. To guarantee that the hash function returns a number in the range of [0, 999], modular arithmetic must be used.
    - Second, all hashing functions (in the context of a hash table, hashing does have other applications like cryptography) try to avoid collision as much as possible. Collision is when the hashing function returns the same index for two unique strings. This is a result of the fact that a property of modular arithmetic is that two unique inputs can result in the same output. For example, 25 % 10 and 35 % 10 both result in the same remainder of 5. The math is the same with strings. So, a good hashing function will be written so as to avoid collision as much as possible. There are various ways to deal with collision, but it should be avoided as much as possible in order to make the hash table as efficient as possible as a data structure.

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

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

- The most common ways of implementing hash tables are **open addressing** and **separate chaining**. This video has a good explanation of them: https://www.youtube.com/watch?v=KyUTuwz_b7Q

Sorting Algorithm Descriptions

NOTE: for the sake of simplicity, all of these descriptions are for sorting a list from least to greatest. The same logic can be applied for going greatest to least.

*** THESE CANNOT BE SHOWN TO STUDENTS SINCE DURING EXAMS THEY ARE ASKED TO GIVE DEFINITIONS. USE THIS FOR YOUR OWN REFERENCE. ***

bubble sort: go through the list n - 1 times starting from the beginning comparing adjacent elements. Swap them if they are out of order.

selection sort:

- starting from 0, select the i^{th} element and swap it with the smallest element in the list. Do this for n - 1 elements.
- starting at the beginning of the array, find the smallest element in the list and swap it with the element at the current position. Move onto the next element, and do repeat this process for up to but not including the last element.

A truly precise definition would be something like “starting from 0, select the i^{th} element and swap it with the smallest element in the list. Do this for n - 1 elements.” or “starting at the beginning of the array, find the smallest element in the list and swap it with the element at the current position. Move onto the next element, and repeat this process for every element up to but not including the last element.”

insertion sort: assume the first element in the list is a sorted list. For the rest of the elements in the list starting at the second, swap it to the left until it's in the correct position.

Shell Sort: h sort the elements with decreasing values of h until and including h = 1 where h sorting is insertion sort but instead of comparing adjacent elements, you compare elements that are h away.

Heap sort: heapify the array then remove the max n - 1 times. Heapify means call fixdown on every element of the array except for leaf nodes (a more detailed description of heap sort is given in previous pages).

quick sort: select a pivot. All items that are less than the pivot go on the left of the pivot. All items that are greater than the pivot go on the right of the pivot. Then, quicksort the halves (each side of the now sorted pivot).

Quicksort Demonstration

Numbers in array: 9, 5, 6, 7, 2, 1, 0, 3, 8, 4
Goal: all numbers less than pivot should be on the left of the pivot, all numbers larger than the pivot should be on the right of the pivot. Algorithm is described below

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

P = pivot, R = right scanner, L = left scanner.
Note: often the pivot will just be randomly selected. Here, arbitrarily the pivot is always selected as the leftmost item.

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

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

R: 4 does not belong on the right,
L: there's nothing that does not belong on the left so scanners meet

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

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

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

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

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

Swap items at scanners and continue scanning

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

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

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

Swap items at scanners and continue scanning

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

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

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

Swap items at scanners and continue scanning

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

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

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

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

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

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

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

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

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

Swap items at scanners and continue scanning

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

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

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

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

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

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

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

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

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

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

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

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

Right half of 2

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

Size is 1 so 3 is now sorted

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

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

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

R: 5 does not belong on the right of the pivot

L: meets right scanner

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

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

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

Left half of 7

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

R: 6 is fine, meets left scanner at 5

L: meets right scanner

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

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

Left half of 5. Nothing there

Right half of 5:

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

6 is size 1. It's now sorted

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

Right half of 7

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

8 is size 1 It's now sorted.

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

Done.

Extra notes on quicksort

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

The point of quicksort is you swap something to be put in the middle so everything on the left and right is on the proper side of the pivot. When the pivot always swaps to the end (like 9 in 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. For example, if a 100 size list, Worst case scenario odds would be $\frac{2}{100} \cdot \frac{2}{99} \cdot \frac{2}{98} \cdot \frac{2}{97} \cdots = \frac{2^{100}}{100!}$
    - the 2 comes from there are two worst case indexes (the last and first), and the decreasing denominator represents every time you select a new pivot, the size is 1 less one item has been sorted
- 2^{100} is a very large number, but 100! is so much larger the fraction is actually very small and so small it will never happen.

Another way to pick pivot (slightly better, but more runtime)

- randomly select 3 elements and pick the median of them.

On average, quicksort will perform better than all other sorts. Typically it's NlgN

- it can actually be proved mathematically that any sort that involves comparisons cannot be faster than NlgN