Types of Objects

Vectors: An array that can dynamically increase its capacity. In Computing I, you used static arrays like

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

This is not a vector since its capacity, 10, can never change.
To create a vector, you create a single structure with a size, capacity, and the array.

- size - how many elements are in the array (initially 0). The index of the next available position is always [size]
- capacity - how many elements the array can hold. If the size equals the capacity when you go to add a new element, you must do a resize operation on the array.
- data - the array. This example arbitrarily uses integers. You can have an array of any data type including all primitive types (int, char, float, double, long int etc) and any structures you make.

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

Linked List: A series of nodes all connected to each other via node pointers

- Singly linked list - each node contains a pointer to the next node. This pointer is set to NULL for the last node in the list since there is no next node.
- Doubly linked list - each node contains a pointer to the next and previous node. The "next" pointer of the last node is set to NULL since there is no next node, and the "previous" pointer of the first node is set to NULL since there is no previous 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;
};
```

Stack: A series of data items all stacked on top of each other and the only one that is accessible is the top. Things are added to the top and removed from the top. Think of a stack of plates. Can represented in multiple ways including an array and linked list

// array implementation
The structure is the same as the vector. The top item is always at index [size - 1].

// linked list implementation
Use one of the linked list structures above.
Then create an additional structure like below

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

Queue: There are multiple types of queues:

- *Regular queue:* items are inserted at the back of the queue and removed from the front of the queue. Think of a line of people in a grocery store. Can be represented using an array or linked list.

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

```
// linked list implementation
```

Use one of the linked list structure above.
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 just by using a bit of math to calculate the other.

- *Priority queue:* items are organized according to priority. The highest priority item is always at the front. Represented using the heap data structure (described below). You can represent a heap as a tree or as an array.

Heap: used for priority queues as previously discussed. A more detailed explanation is later in this document. For the array implementation, the structures are below. Note how one of the structures is actually just a vector.

```
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* queue;
} Priority_queue;
```

Tree: in general, a tree is a series of nodes starting at the root node, and every node has pointers to left and right children. A node may or may not have a left and/or right child. When it doesn't, the pointer is set to NULL. Every node does have a parent, except for the root node. There are different types of trees described below

- **Binary tree:** a tree where everything less than a node's data goes to the left, and everything greater than a node's data goes to the right. Not self-balancing
```
typedef struct node Node;
struct node {
    int data;
    Node* left;
    Node* right;
};
```
- **AVL tree:** a self balancing binary tree
```
typedef struct node Node;
struct node {
    int data;
    Node* left;
    Node* right;
    int height;
};
```
  This is the data structure that must be used in the evil hangman lab. Use this link to not only understand AVL trees but also as some starter code for the AVL tree interface (referred to as the associative array in the lab).
  https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Binomial Heap: a type of priority queue that is a forest of trees, where two trees of the same size do not exist. This is taught at the conceptual level and not with code, and is discussed in greater detail later in this document.

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, you write a hashing function that uses modular arithmetic to convert the string into an unsigned integer that is within the range of the valid indices of the array. For example, if the hash table had a capacity of 1000, the hashing function would return a number in the range [0, 999].

- Though this isn't valid C code (again, you have to write the hash function), this is how it can conceptually be thought of.
```
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

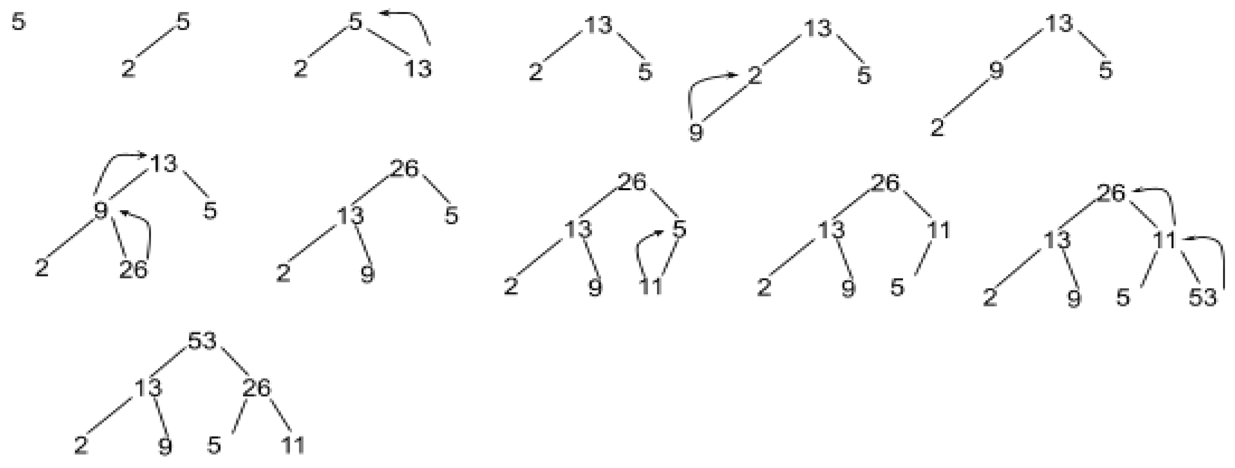
Heap

- Type of priority queue. Items are inserted top to bottom left to right .
- every node in the heap is bigger than its children
- *insert (enqueue):* Insert the item at the next available position, and then fix up until it's in the correct position. Fix up means the item will swap with the parent if it has a higher priority than the parent.
    - Cost of insert is lgN (this really means floor(lgN) but it's referred to as lgN.
        - i.e. to insert a 7th item, this will be at most lg7 which equals 2. To insert an 8th item, it becomes at most lg8 which equals 3 since at most something will have to swap up 3 levels
- *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.

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

- When using a heap in an array (k = index)
    - parent index: (k - 1)/2 right child index: 2k + 2 left child index: 2k + 1

<u>Insert:</u> 5, 2, 13, 9, 26, 11, 53



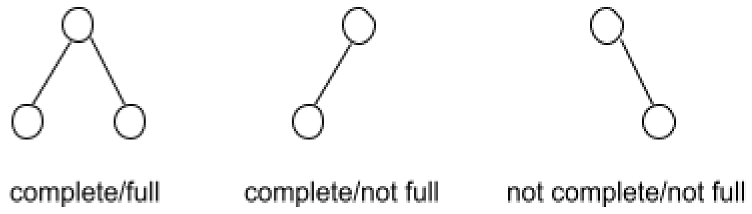
Indices of array:	0	1	2	3	4	5	6
Items in array:	53	13	26	2	9	5	11

Merging the heap

- inefficient way: NlgN
    - remove elements from one heap (lgN), insert into next heap (lgN), and must do it 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's still "inefficient" since there's a special kind of heap called a **binomial heap** discussed later that gets a O(1) merge.

A heap is a **left complete tree**, but for convenience we just say complete tree (it could be right, doesn't matter, we just arbitrarily chose left).

- **complete:** nodes filled in in proper order
- **full:** if a node has children it has all of them that it can have

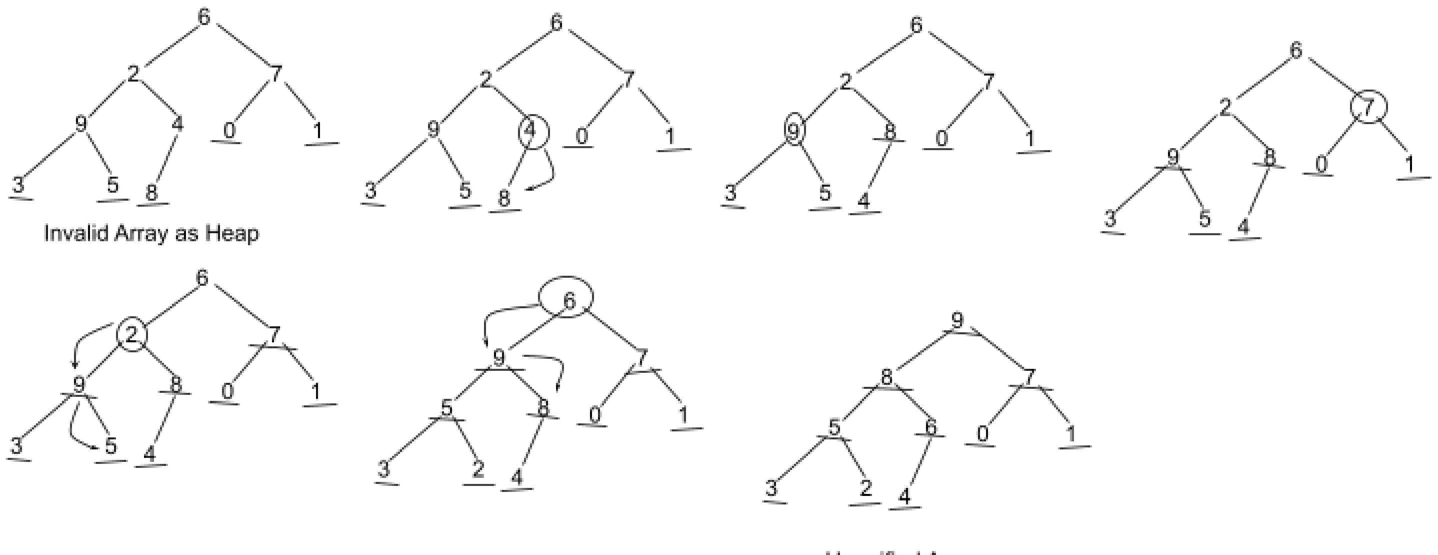


Turning an array into a heap

- creating a heap the normal way (as indicated on the previous page) is O(NlgN). Although the above example used a tree, an array can represent the tree. So, every time you inserted something, using the index rules (index of parent is (k + 1)/2, left/right child 2k+1/2k+2) you create the heap
- There is a faster way that is O(N) and this is called **heap sort** that involves **heapifying** an array

Heapify

- 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. That is 4 in the example below.
    - 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
- Now we're at index 0 and the array has been heapified.


Invalid Array as Heap

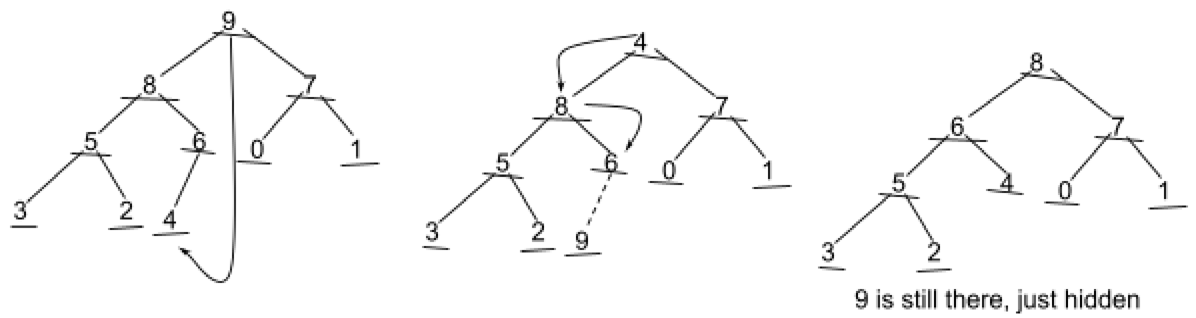
Heapified Array

9 8 7 4 5 0 1 3 2 4

- heapify is one part of **heap sort** whose description is below

Heap sort

1) heapify the array (turn it into a heap, as demonstrated above)
2) remove the max n - 1 times and now it's been sorted (1 max heap removal demonstrated below)
    a) call fix down on every element of the array starting at the end of the array ignoring leaf nodes (once again, leaf nodes are ones that are already heaps since they have no children, which means they're larger than they're children and they can't possibly be invalid heaps.
    b) if given size, the first non leaf (index where you want to start, the first node from last that

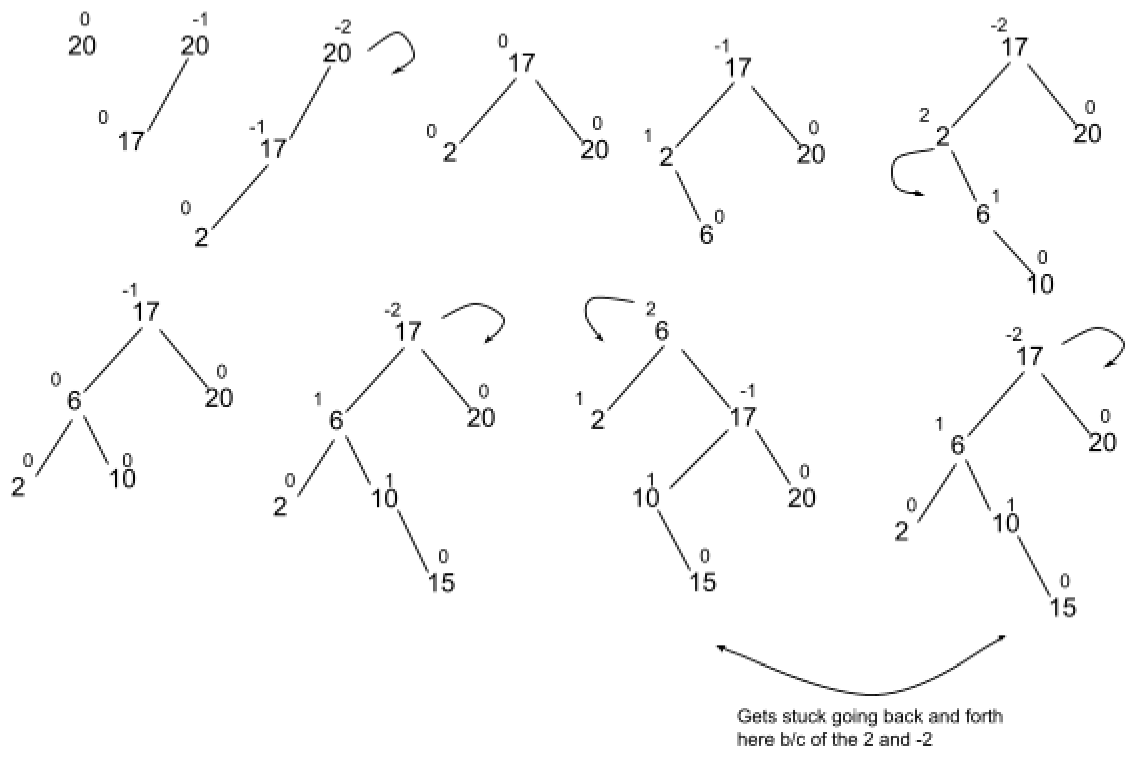


has a child) is just the halfway point. So, index of first non-leaf is (size / 2 - 1)...-1 because indexes are 1 smaller than size

- NOTE: the 9 gets a dashed line since every time the max is removed, the size is decreased by 1. However, when we say the max is "removed", it's not actually being removed. It is still in the array, it's just the integer representing the size of the array has decreased by 1 so it's hidden.

AVL Tree

- As previously stated, and AVL tree is a self balancing binary tree.
- **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)

Simple cases: parents and children lean the same way

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

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

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

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

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

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

Complex Cases: parent and children lean opposite ways

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

- perform a left rotation on the <u>left child</u> of the root
- perform a right rotation on the <u>root</u>

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

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

- perform a right rotation on the <u>right child</u> of the root
- perform a left rotation on the <u>root</u>

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

AVL Tree Visualizer
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Navigating Trees

Note: in all of these, left comes before right. These orders are the orders used when doing tree functions recursively.

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

Note about tree_insert

- 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

Sorting Algorithm Descriptions

The definitions of the following sorting algorithms shown below should be known. The ideal definitions cannot be shown here since that is exam material

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

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**

<u>Done.</u>

Extra notes on quicksort

Worst case scenario: O(N^2)

The point of quicksort is you swap something to be put in the middle so everything on the left and right is on the proper side of the pivot. When the pivot always swaps to the end (like 9 in 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 $\frac{2}{100} \cdot \frac{2}{99} \cdot \frac{2}{98} \cdot \frac{2}{97} \ldots = \frac{2^{100}}{100!}$
    - 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

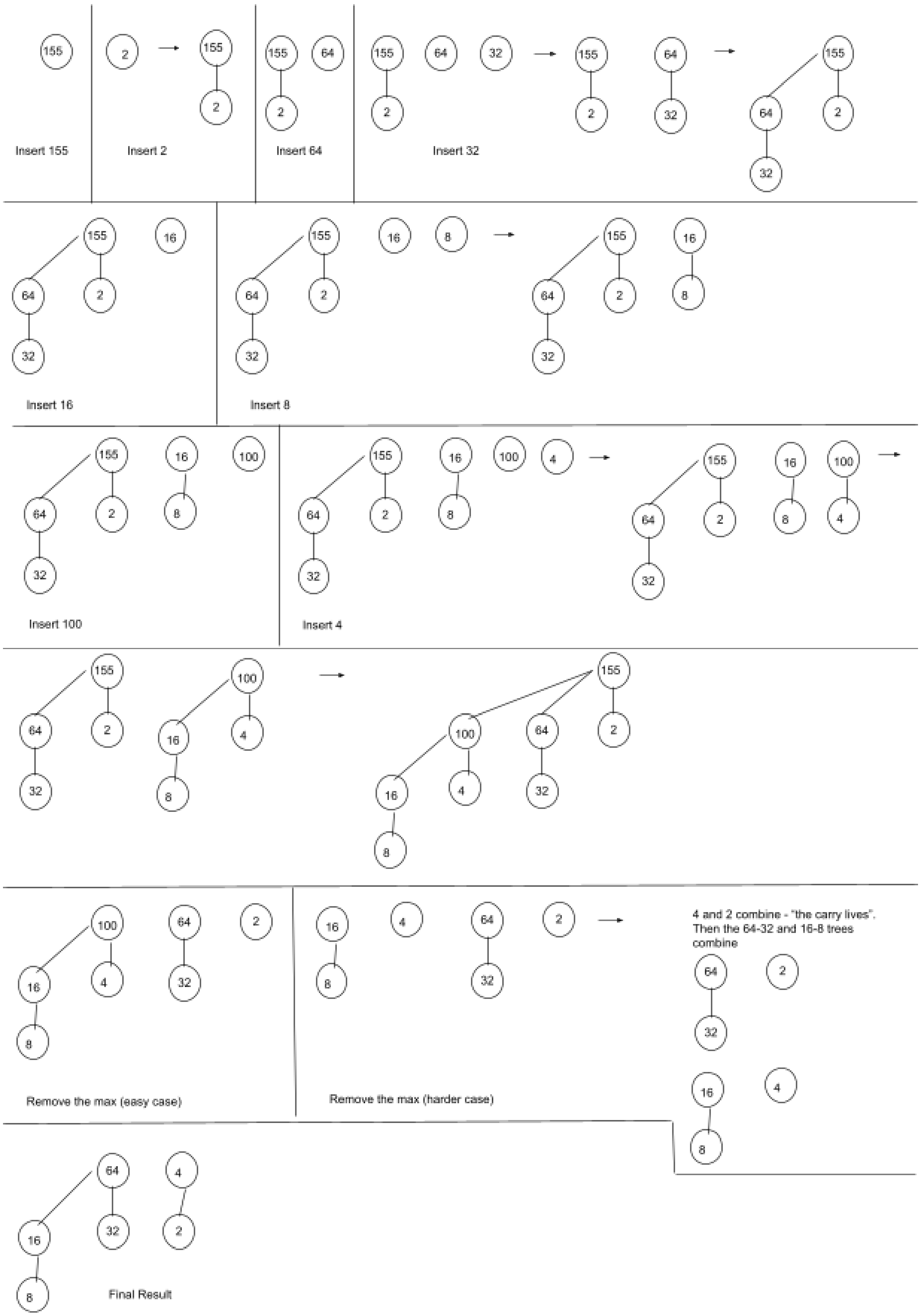
Binomial Heap (Binomial Queue)

Binomial heap: a forest of trees where 2 trees of the same size cannot exist.

- When this happens, 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, put the tree (that does not have the root that will become the new root) under on the left most side of the tree that is acting as the root.
- node insertion: O(lgN). To insert a 7th node would be lg7 = 2. To insert an 8th node would be lg8 = 3
- **left heap property**
- Removal of max: O(lgN)
- Merge: O(1) constant
- The carry lives

Binomial heaps and binary

- Binomial heaps are a forest of trees, and each tree can be a size that is powers 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



4 and 2 combine - "the carry lives". Then the 64-32 and 16-8 trees combine