To-Do

- Consider moving Trees, Tries, etc. to their own sections so that everything is transferred one level up.
- Consider including language-specific examples of some ADTs, such as `std::vector` in the *Dynamic Arrays* section or `Dictionary` in the *Hash Tables* section.
- Elaborate on many sections, e.g. complexity (beyond the *Complexity Summary*) subsection + outside reference.

Data Structures

Complexity Summary

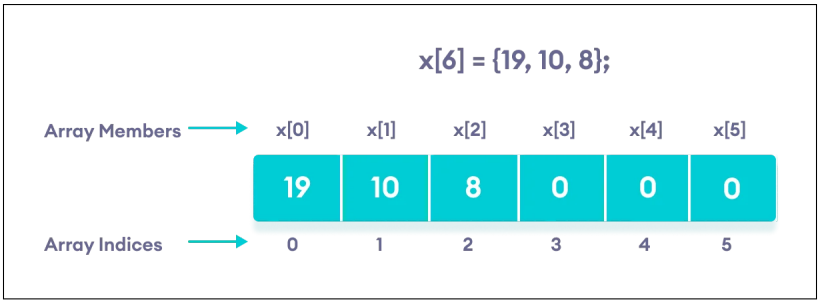
Discussion about complexity measures (big-Ω, big-θ, big-O) can be found at Programiz.

Data in this table comes lots of places including: Big-O Cheat Sheet,. . . .

Structure	Best Case				Average Case				Worst Case				Space
	Access	Search	Ins	Del	Access	Search	Ins	Del	Access	Search	Ins	Del	Worst
Array	–	–	–	–	$\theta(1)^1$	$\theta(n)^1$	$\theta(n)^1$	$\theta(n)^1$	$O(1)^1$	$O(n)^1$	$O(n)^1$	$O(n)^1$	$O(n)^1$
Linked List	–	–	–	–	–	–	–	–	$O(n)^1$	$O(n)^{12}$	$O(n)^1$	$O(n)^1$	$O(n)^1$
Hash Table Dictionary	–	–	–	–	$\theta(1)^1$	$\theta(1)^1$	$\theta(1)^1$	$\theta(1)^1$	$O(n)^1$	$O(n)^1$	$O(n)^1$	$O(n)^1$	$O(1)$
BST	–	$O(1)^3$	$O(1)^3$	$O(n)^3$	–	$O(\log n)^{23}$	$O(\log n)^{\star 3}$	$O(\log n)^3$	–	$O(n)^3$	$O(n)^3$	$O(n)^3$	$O(n)^3$
AVL tree	–	–	–	–	–	$O(\log n)^4$	$O(\log n)^4$	$O(\log n)^4$	–	$O(\log n)^4$	$O(\log n)^4$	$O(\log n)^4$	–
Red-Black tree	–	–	–	–	–	–	–	–	–	–	–	–	–
Stack	–	–	–	–	$\theta(n)^1$	$\theta(n)^1$	$\theta(1)^1$	$\theta(1)^1$	$O(n)^1$	$O(n)^1$	$O(1)^1$	$O(1)^1$	$O(n)^1$
Queue	–	–	–	–	$\theta(n)^1$	$\theta(n)^1$	$\theta(1)^1$	$\theta(1)^1$	$O(n)^1$	$O(n)^1$	$O(1)^1$	$O(1)^1$	$O(n)^1$

[1]Big-O Cheat Sheet [2]Google *Google says it's $O(n)$, but nothing else online agrees. [3]OpenGenius IQ [4]AVL Wiki

Arrays



A C-style array, with missing elements auto-set to 0

- Arrays allow storing elements of a single data type contiguously in memory.
  - Random access is a great feature.
  - Downside is that you must declare array size at creation, and they cannot dynamically grow.
  - **Complexity:**

    Search:

    When unsorted, linear can be done in $O(n)$ time, $O(1)$ space.

    When sorted, binary can be done in $O(\log(n))$ time, $O(\log(n))$ space due to recursive calls.

    Insert:

    When unsorted, insert at the end can be done in $O(1)$ time, $O(1)$ space.

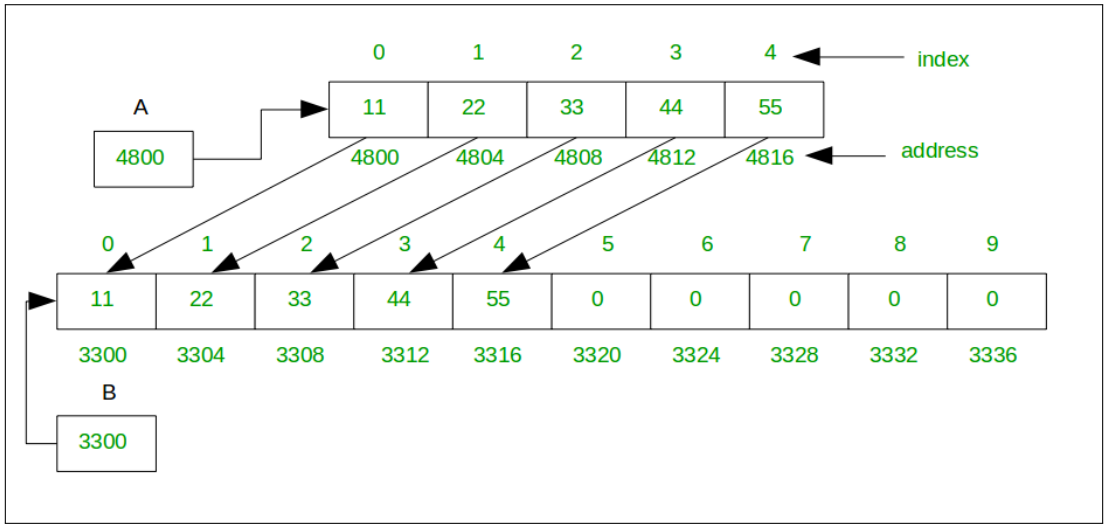
    When sorted, may require $O(n)$ time (because all elements may have to be moved), $O(1)$ space.

    Delete:

    When unsorted, delete is done by first linear searching, which requires $O(n)$ time, $O(1)$ space.

    When sorted, may require $O(n)$ time (because all elements may have to be moved), $O(\log(n))$ space (as an internal stack will be used). Say more here!

Dynamic Arrays



A dynamic array being, well...*dynamic*

Details here

Mention `std::vector` in C++!

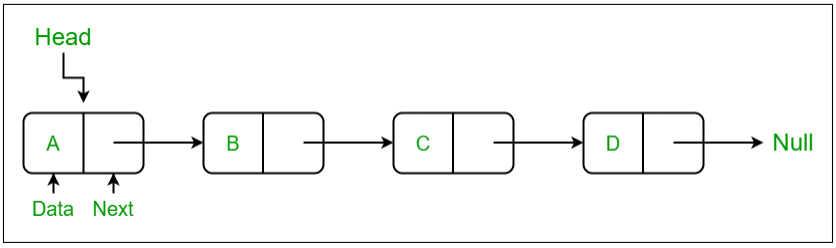
https://www.geeksforgeeks.org/how-do-dynamic-arrays-work/

https://en.wikipedia.org/wiki/Dynamic_array

Linked Lists and Variants

Details here

(Singly) Linked Lists



A (singly) linked list

- Each node stores the element we want to store + a pointer to the next item in the linked list.
  - This means that linked lists may take up more memory than an Array, because we have to store both the data *and* the pointer while an Array has only the data stored.
- Linked lists *can* grow, because they're not stored contiguously.
  - The main upshot is that the size *isn't* fixed: Linked Lists can grow dynamically.
  - One downside is that random access isn't possible: Access requires traversing the entire list from the beginning until the desired object is found.

- **Complexity:**

  Search / Access: Both $O(n)$ time, as entire list must be traversed to reach desired element.

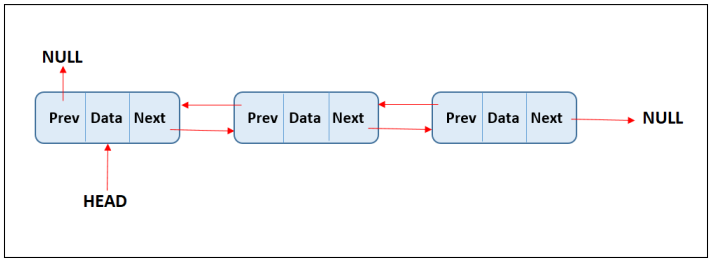
  Insert:

  - Inserting at head position is $O(1)$ time, $O(1)$ space.
  - Inserting at given position is $O(n)$ time, $O(1)$ space.
  - Inserting after a given node is $O(1)$ time, $O(1)$ space.
  - Inserting at end is $O(n)$ time, $O(1)$ space, though this can be made $O(1)$ time by keeping an extra pointer to the last node.

  Delete:

  - Deleting from head position is $O(1)$ time, $O(1)$ space.
  - Deleting from middle is $O(n)$ time (both iterative and recursive), $O(1)$ space (if iterative) or $O(n)$ space if recursive (due to recursive call stack).
  - Deleting from end is $O(n)$ time, $O(1)$ space, though this can be made $O(1)$ time by keeping an extra pointer to the last node.

Doubly Linked Lists



A doubly linked list

Details here

Circular Linked Lists and Circular Doubly Linked Lists

Details here

Comparison of Variants

fill this out more

- Singly linked lists are good for queues.
- Circular and Doubly linked lists require extra space for all the additional pointers. As space becomes cheaper, these additions are often negligible.
- Java's built-in linked list class is a circular doubly linked list by default.
- C++'s STL has a `ForwardList` which is a singly linked list and a `List` which is circular doubly linked.

Linked Lists versus Arrays

There are many factors to consider when comparing arrays to (singly) linked lists, including but not limited to:

- **Main Memory Location (in C/C++):** Arrays can be created in the stack *or* in the heap; linked lists can only be created in the heap.

  Access to data stored inside the stack is faster because it only requires one step to access. Conversely, heap data requires two steps and will be slower by default.

- **Size:** Arrays are fixed length and cannot be resized without creating a new array; by comparison, linked lists are variable-length and can grow/shrink as needed (until heap memory is full).

  This provides linked lists with greater flexibility than arrays. In addition, the fixed-size nature of an array opens the door for the possibility that the space provided will be imperfectly utilized.

- **Space Required:** Arrays only occupy the amount of space required for their data elements; linked lists require space for both data elements *and* pointers.

  For example: When sized for five 4-byte integers, arrays will occupy $5 \times 4 = 20$ bytes while the corresponding linked list will require 20 bytes plus whatever additional space the five pointers require.

- **Access:** Arrays allow for random access time; conversely, linked lists provide *sequential* access, requiring one to traverse from the beginning in order to access a random location.

  Access for arrays can be done in $O(1)$ time in all circumstances; sequential access as in linked lists requires $O(n)$ average/worst time.

- **Inserting at First Position:** Inserting at first position of an array requires shifting all subsequent elements, thus requiring $O(n)$ time; on the other hand, inserting at first position of a linked list can be done simply by rearranging a handful of pointers and thus requires only $O(1)$ time.

  Note: Movement of data in the array case can be costly in the event that the array contains complex data types. For instance, an array of size-5 consisting of 200-byte records will require 1000 bytes of data to be moved in order to insert at first position.

- **Inserting at Last Position:** Inserting at the last position of an array requires only $O(1)$ time; inserting at the last position of a linked list requires us to traverse the entire linked list first, thus requiring $O(n)$ time.

- **Deleting from First Position:** As in the insertion case, deleting the first element of an array requires data shifting and hence requires $O(n)$ time; similarly, deleting from the first position of a linked list requires only pointer management and hence is done in $O(1)$ time.

- **Deleting from Last Position:** In an array, deleting the last element can be done in $O(1)$ time; by contrast, deleting the last element of a linked list requires sequentially traversing the entire linked list, thus requiring $O(n)$ time.

- **Search:** In an array, linear search can be done in $O(n)$ time while binary search can be done in $O(\log n)$ time. Here, binary search provides a clear speedup over the linear search case.

  Linear search on linked lists still requires $O(n)$ time; however, binary search is slower—requiring $O(n \log n)$ time—because there is no quick access to the middle element of a linked list. This makes binary search a non-option in most cases.

- **Sorting:** Say array stuff here.

Certain sorting algorithms such as Insertion Sort and Merge Sort (include links) are designed for linked lists. Say more about this here.

Hash Tables

In Python, one example is a `Dictionary`. Note: Google says maps/dictionaries are *unordered*, but as of Python version 3.7, `Dictionary` objects are ordered.

- The key idea of a hash table is to use a hash function to map keys to buckets.
- Hash table is an array coupled with a hash function that takes the data ("key") as input and outputs a hash value.
- Hash tables are theoretically better than linked lists, because they offer dynamic sizing while also allowing theoretically-constant insertion, deletion, and lookup times.
- One downside is collisions, which occur when the hash function returns the same index for two different keys. See the subsection on hash functions for more details about this.
- **Complexity:**

  Search / Access: Best and average case are $O(1)$, worst cases is $O(n)$.

  Insert: Best and average case are $O(1)$, worst cases is $O(n)$.

  Delete: Best and average case are $O(1)$, worst cases is $O(n)$.
- There are two different kinds of hash tables: **Hash Sets** and **Hash Maps**.

Hash Functions

- When implementing hash tables, the best strategy is to choose a hash function that decreases the probability of collisions occurring.
- A perfect hash function would be a one-to-one mapping between the keys and buckets. This would result in no collisions.

  However, in most cases, a hash function is not perfect, and there must be a trade-off between the number of buckets and the capacity of the buckets.
- Designing a good hash function is an open problem. Here are some properties of good hash functions:
  - Should make use of all info given by a key.
  - Uniformly distributes across table.
  - Maps similar keys to very different hash values.
  - Uses only very fast operations.
- Every hash function must have a built-in collision-resolution algorithm. Such an algorithm should solve the following problems:
  - How to organize the values in the same bucket?
  - What if too many values are assigned to the same bucket?

- How to search for a target value in a specific bucket?

These questions are related to the capacity of the bucket and the number of keys which might be mapped into the same bucket according to our hash function.

- Here are some popular collision-resolution algorithms:
  - **Linear probing:**

    If a key hashes to same index as a previously-stored key, it is assigned the next available slot in the table.

    Drawbacks: Clustering.

    Once a collision occurs, you increase the chances another will occur in the same area.

    Complexity:

    Worst-case is $O(n)$ for insertion, deletion, and lookups, as it's increasingly likely that the last slot in the table is the next available slot.

  - **Separate chaining:**

    The hash table is an array of pointers to linked lists. When collision occurs, the key can be inserted in constant time at the head of the appropriate linked list.

    Drawbacks: Resolving collisions still requires a form of linear search, which is suboptimal.

    Complexity:

    Worst-case lookup is $O(n/k)$, where k is the size of the hash table.

    Note that theoretically $O(n/k) = O(n)$, so this isn't *theoretically* better than the worst-case lookup time for a linked list. Realistically, however, $O(n/k)$ can be much better.

Hash Sets

- The hash set is one of the implementations of a set data structure to store no repeated values.

Hash Maps

- The hash map is one of the implementations of a map data structure to store (key, value) pairs.

Trees

Much of the info about tries and hybrid tries comes more or less verbatim from this Dr. Sterns webpage. There's also this Stack Overflow thread with different implementation possibilities, as well as this leetcode exercise which can be solved using tries in $O(n)$ time, $O(n)$ space (compared to $O(n^2)$ time, $O(n)$ space for the naive approach).

- Trees are non-linear, hierarchical data structures used to show relations using a parent-child hierarchy.

  They are considered to be some of the most powerful and advanced data structures, and are often employed to simplify and fasten searching and sorting operations.

- Linear structures like lists couldn't illustrate the nesting complexity of e.g. a directory file structure, a family tree, the HTML Document Object Model (DOM), etc.

- Trees are commonly used to represent game states in e.g. tic tac toe.

Generalities

- Terminology:
  - Each element is called a *node*.
  - The top node is called the *root*.
  - Any subnode of a given node is called a *child node*, and the given node, in turn, is the child's *parent*.
    Every child has only a single parent but a parent can have multiple child.
  - *Sibling nodes* are nodes on the same hierarchical level.
  - Nodes higher than a given node in the same lineage are *ancestors*, and those below it are *descendants*.
  - Nodes with no children are called *leaves*.
  - The *height* of a tree is the length of the longest path to a leaf.
  - The *depth* of a node is the length of the path to its root.
  - A *binary tree* is a tree in which each node has no more than two child nodes. One or both children may be null.
  - An *n-ary tree* is a tree in which each node has no more than n child nodes. Any or all children may be null.
    Clearly, a binary tree is a 2-ary tree.
  - A *full n-ary tree* is...finish.
  - A *strict n-ary tree* is...finish.
  - A *complete n-ary tree* is...finish.
    Liken this to missing nodes / empty spaces in the array representation below.

- If a tree has n nodes, then it has $n-1$ edges. In the event that there are *more* than $n-1$ edges, the structure is a graph, not a tree.

Methods to traverse trees: reorganize this stuff!

Many search, etc. algorithms need the ability to traverse all the nodes in a given tree. There are several different strategies for tree traversal including:

Breadth-First Traversal or or Breadth-First Search (BFS):

In this method, we look at nodes at each level before looking at children.

This method traverses the tree level-by-level and depth-by-depth. If we have a tree that's $2 \leftarrow 1 \rightarrow 5$, $3 \leftarrow 2 \rightarrow 4$, and $6 \leftarrow 5 \rightarrow 4$, for example, then BFS returns $1-2-5-3-4-6-7$.

Depth-First Traversal or Depth-First Search (DFS):

In this method, we look at all descendants of a node before moving to the next node. (three versions of DFS are discussed below!)

This method explores a path all the way to a leaf before backtracking and exploring another path. If we have a tree that's $2 \leftarrow 1 \rightarrow 5$, $3 \leftarrow 2 \rightarrow 4$, and $6 \leftarrow 5 \rightarrow 7$, for example, then DFS returns $1-2-3-4-5-6-7$.

Representing Trees using Arrays and LinkedLists

- Array deets

- LinkedList deets

Computations on Graphs (do I care about this stuff?)

If so:

- Number of binary, n-ary trees.
- Height versus nodes in binary, n-ary trees.
- Internal, external nodes.

Binary Search Trees (BSTs)

A BST is a binary tree that satisfies a specific ordering property, namely that any/all left nodes (on any given subtree) are less than the root node, which are less than any/all right nodes.

Searches/Finds:

This property of BST yields efficient searches/finds, because at each operation, we've chopped off $\approx$ half of the nodes.

Insertions: Insertion works similar to searches/finds.

One possible issue that could result from this is that the resulting tree can become super-unbalanced; think of inserting 1, then 2, then 3, etc., which essentially turns your BST into a linked list.

Traversing: There are three main types of traversals. (all of these are examples of DFS!)

(assume that B is the root, A is its left child, and C is its right child)

- *pre-order* traversal, which means you visit the root first, then its left nodes, then its right nodes. In other words, $B \to A \to C$.
- *in-order* traversal, which means you visit the left nodes first, then the current node, then the right nodes. In other words, $A \to B \to C$. (these are usually preferred with BSTs because the nodes are printed in order.)

  Example use: To determine if a tree is a valid BST, you can use in-order traversal to record the nodes in order and then see if left nodes are less than parents are less than right nodes, i.e. if `x[i]<x[i+1]` for all `i=0,1,...,len-1`.
- *post-order* traversal, which means you visit the left nodes first, then the right nodes, then the current node. In other words, $A \to C \to B$.

Complexity:

According to OpenGenius IQ (and supported by other sources as well):

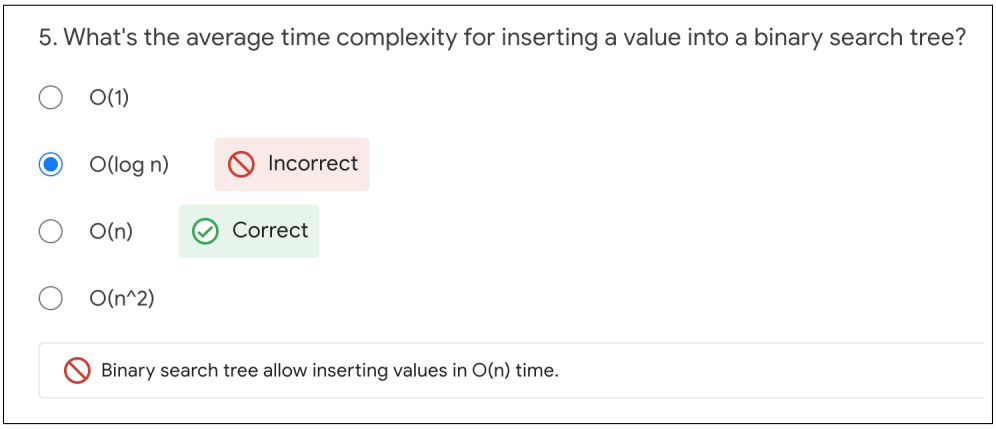
Operation	Worst Case	Average Case	Best Case	Space
Search	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
Insert	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
Delete	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$

Along the same lines, multiple sources have claimed that:

Search: $O(\log n)$ if balanced, $O(n)$ if unbalanced. Average: $O(\log n)$. (Google agrees....)

Insert: $O(\log n)$ if balanced, $O(n)$ if unbalanced. Average: $O(\log n)$. (Google says $O(n)$ instead....)

However, Google says that the average-time insertion into a BST is $O(n)$:



AVL (Adelson-Velshi and Landis) Trees

An AVL tree is a self-balancing BST.

In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, re-balancing is done to restore this property.

Relation to Red-Black (RB) Trees:

Every AVL tree can be colored red-black, but there exist RB trees that *are not* AVL-balanced.]

Complexity:

Lookup, insertion, and deletion all take $O(\log n)$ time in both the average and worst cases (where n is the number of nodes in the tree prior to the operation). Insertions and deletions may require the tree to be re-balanced by one or more tree rotations.

Rebalancing:

During insert and delete operations, a (temporary) height difference of 2 may arise, which means that the parent subtree has to be "rebalanced".

The given repair tools are the so-called *tree rotations*, because they move the keys only "vertically," so that the ("horizonta") in-order sequence of the keys is fully preserved (an essential necessity for a BST).

Let X be the node that has a (temporary) balance factor of -2 or $+2$. Its left or right subtree was modified. Let Z be the higher child (see figures 2 and 3). Note that both children are in AVL shape by induction hypothesis.

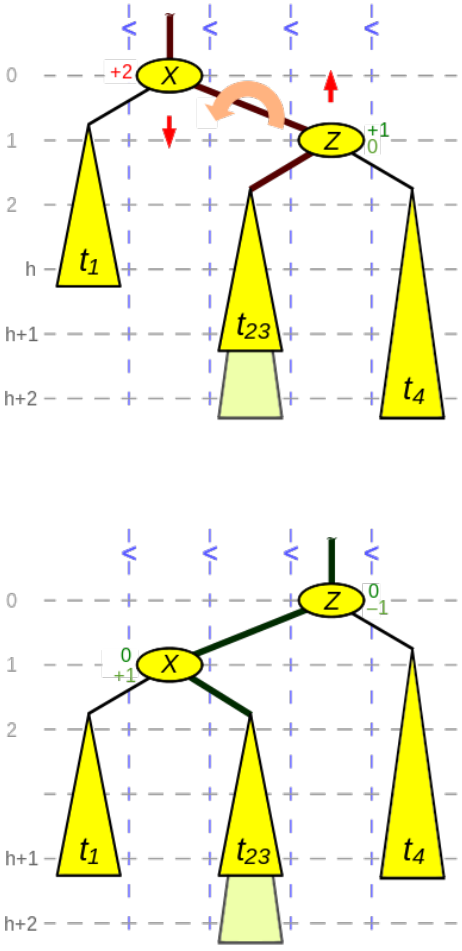
There are four possible variants of the violation, each with its own solution.

(i) Right Right. Z is a right child of its parent X and $BF(Z) \geq 0$. In this case, X is re-balanced with a `rotate_Left` simple rotation. See AVL Figure 2.

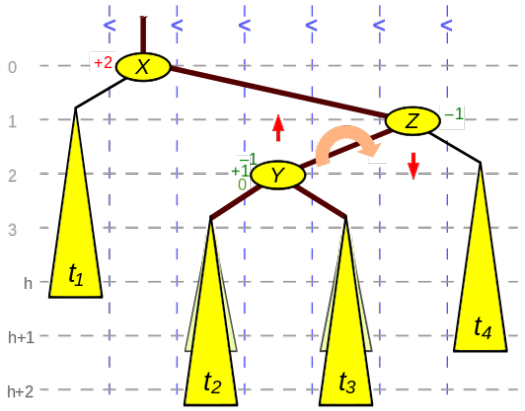
(ii) Left Left. Z is a left child of its parent X and $BF(Z) \leq 0$. In this case, X is re-balanced with a `rotate_Right` simple rotation. See (the mirror image of) AVL Figure 2.

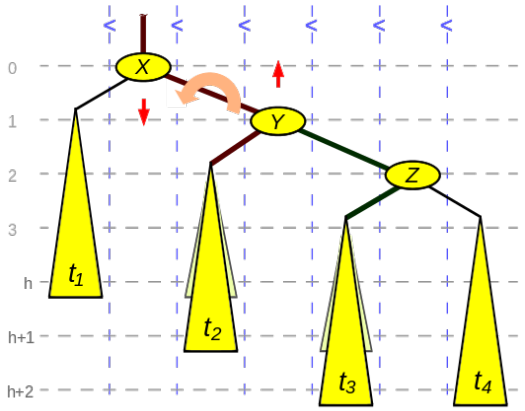
(iii) Right Left. Z is a right child of its parent X and $BF(Z) < 0$. In this case, X is re-balanced with a `rotate_RightLeft` double rotation. See AVL Figure 3.

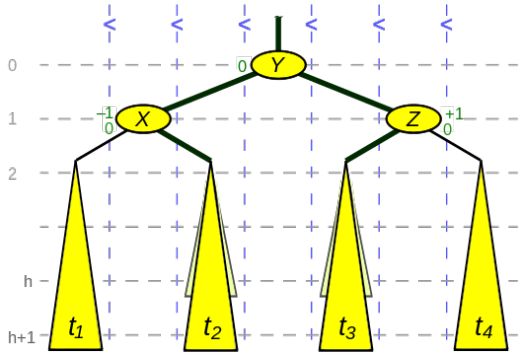
(iv) Left Right. Z is a left child of its parent X and $BF(Z) > 0$. In this case, X is re-balanced with a `rotate_LeftRight` double rotation. See (the mirror image of) AVL Figure 3.



AVL Figure 2







AVL Figure 3

Red-Black Trees

Relation to AVL Trees:

Every AVL tree can be colored red-black, but there exist RB trees that *are not* AVL-balanced.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue

eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

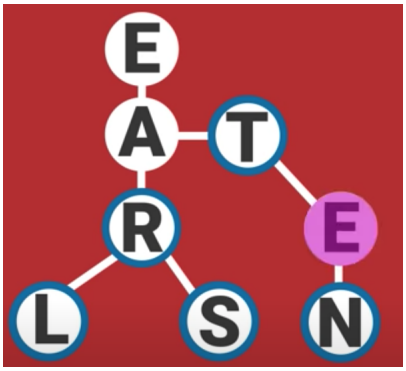
Tries (reorganize all of this stuff into hierarchical format!)

In computer science, a **trie** (also known as a *digital tree* or *prefix tree*) is a tree data structure used for locating specific keys from within a set. It is an example of a k-ary search tree.

Like other search-oriented data structures, a trie stores keys and associated values. Together, the key and value are called an entry. The keys are most often strings, with links between nodes defined by individual characters, but the value could be of any type, as the trie just stores and retrieves it as an opaque value. Often this value is a unique identifier or pointer that gets you to some data related to the key (e.g. the primary key of a record in a DBMS).

In order to access a key, the trie is traversed depth-first, following the links between nodes. The links represent characters in the key.

This is very useful for things like dictionaries:



For word `EAR`*,make* `E`→`A`→`R`*; from there, make* `EARS`*,* `EARL`*, etc., or branch off of* `EA` *to make* `EAT`*,* `EATEN`*, etc. The blue outlines note an identifier that indicates that the nodes are valid word-enders.*

The height of a trie is always equal to the length of the longest key added to it. This means that tries tend to be very wide and not terribly deep, which makes retrieval speed much quicker. Adding a new key requires creating at most `len(key)` nodes, but if your keys share common prefixes (which is typical), you will re-use nodes from previously-added keys.

Similarities with maps. Like a map, a trie can also be used to store only keys with no associated values. For example, a simple list of dictionary words might not need associated values, but you might want to find all words that start with a given prefix. In these cases we use a variant of a trie that only stores keys with no associated values. The structure is similar, but we can save a little memory in each node.

Contrasts with BSTs. Unlike a binary search tree, nodes in the trie do not store their associated key. Instead, a node's position in the trie defines the key with which it is associated. This distributes the value of each key across the entire data structure, meaning that not every node necessarily has an associated value.

Adding Keys and Values

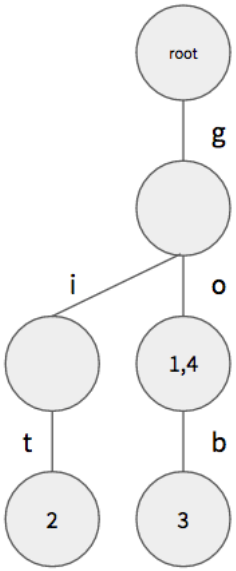
A trie constructs a tree of nodes based on the letters in the keys added to it. The tree starts with a single root node that holds no values. When a new key/value pair is added, the trie follows this algorithm:

1. let current node = root node
2. for each letter in the key,

- find the child node of current node associated with that letter
- if there is no child node associated with that letter, create a new node and add it to current node as a child associated with the letter
- set current node = child node

3. add value to current node

When indexing keys that are not unique (e.g., last names), we will often need to add the same key to the trie multiple times but with different values. For example, this shows *go* with value 1, *git* with value 2, *gob* with value 3, and (a repeat) *go* with value 4:



Retrieving Keys and Values

Tries generally allow very fast retrieval of keys starting with a prefix. To do so, follow this algorithm:

1. let current node = root node
2. for each letter in the prefix,
   - find the child node of current node associated with that letter
   - if there is no child associated with that letter, no keys start with the prefix, so return an empty/null list
   - set current node = child node
3. current node now points to the branch containing all keys that start with the prefix; recurse down the branch, gathering the keys and values, and return them

Finding the start of the branch containing all keys and values takes at most `len(prefix)` lookups, regardless of how many entries there are in the trie.

Retrieval complexity depends on the length of the key prefix, not the number of entries in the trie, so the retrieval performance remains about the same as we add more entries. Compare that with a binary search tree, which requires at most $log(n)$ comparisons to find the first matching key (where n is the number of entries in the tree).

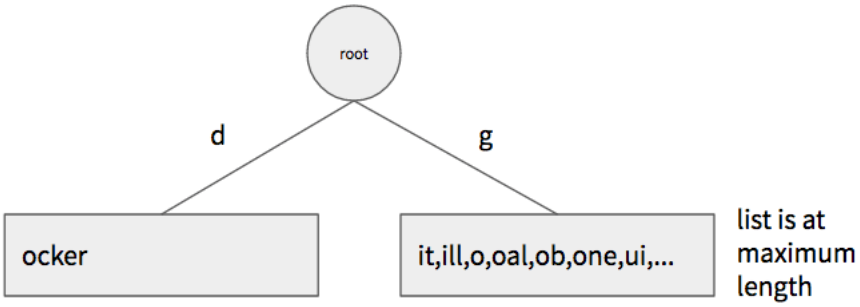
Branches for a short prefix might contain a lot of keys and values, so we typically return only the first m entries from the branch (where m is a usually-small parameter passed by the caller. To get this subset, you do a depth-first recursion down the branch, following the letters in each node according to their alphabetical order. As soon as you gather m keys and values, you return to stop the recursion.

Hybrid Tries (add sections and subsections!)

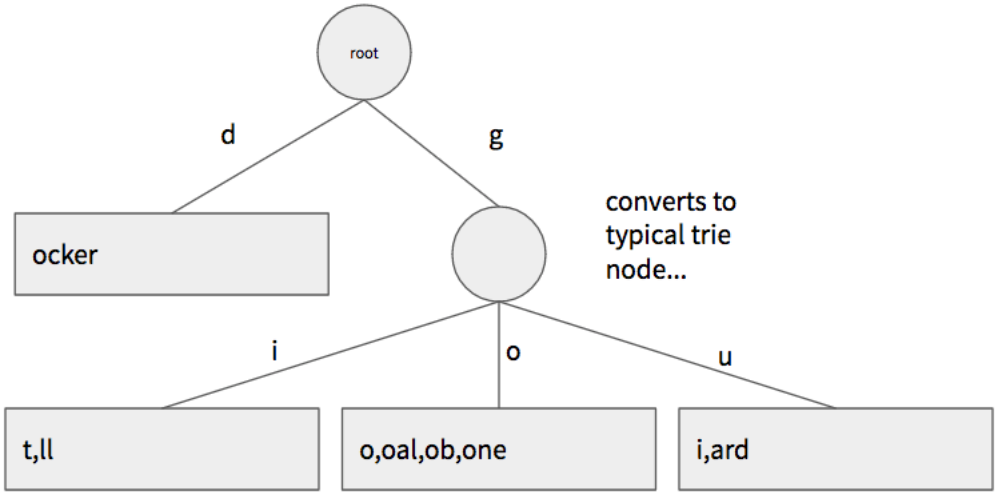
Tries offer fast retrieval, but they aren't very space-efficient unless most keys are short and share common prefixes (i.e., dictionary words). Longer keys that don't share many prefixes create a "leggy" tree structure which resembles a many-tentacled octopus!

In these situations, it's common to use a variant known as a hybrid trie. Instead of always creating a new node for each letter in the key, the leaf nodes in a hybrid trie store a list of key suffixes until those reach some maximum number, for example, 50. When the 51st key is added, the leaf node then transforms itself into a typical trie node, and creates a new leaf node for each distinct first-letter in the key suffixes it holds. Each of the new child leaf nodes then stores what remains of the key suffixes for their respective letter.

For example, consider the following:



From here, adding *guard* won't work because the right child is full, so...



...with one child for each distinct first-letter

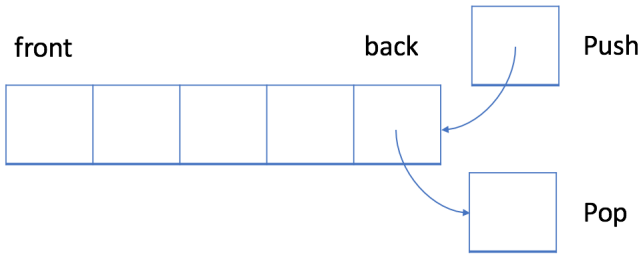
...adding *guard* turns the first level into a traditional trie structure and moves the hybrid part one level down.

Note: These diagrams depict only the keys, but you can create a hybrid trie that also stores associated values. The leaf nodes would store key-suffix and value pairs instead of just the key suffixes.

Stacks and DFS

- Stacks are like stacks of plates: Last in, first out (LIFO).
  - The insert operation for a stack is called *push*. Inserting always happens at the end.

- The delete for a stack is called *pop*. Removing always happens from the end.



- Stacks are even easier to implement than queues using any sort of dynamic array.

- A so-called `minstack` can be implemented which supports *push*, *pop*, *peek* (at the top element), and *getMin()*.

  This can be done with two stacks simultaneously by pushing on to `stack2` the smallest elements pushed on to `stack1` and by popping from `stack2` any elements equal to pops done on `stack1`.

  This may also be doable using only one stack if we push/pop pairs of elements, one of which is a tracker for the existing min.

  Does this mean `maxstack`, `avg stack`, etc., can also be done?

- Stacks can be used to implement queues (next section), but doing so requires ≥ 2 stacks.

  More details here.

Depth-First Search (DFS)

- Stacks are commonly used when doing a depth-first search (DFS).[1]
  - In the first round, the root node is processed. In the second round, choose a path a neighboring node and trace-back until a node is reached with no way to go deeper. Then, backtrack to the root and choose a new path to a new second node. From there, try subsequent paths until hitting a node that's already been visited; then, backtrack and start again.

    The processing order of the nodes is the exact opposite order as how they were added to the stack, which is Last-in-First-out (LIFO). That's why we use a stack in DFS.
  - Different from BFS, the nodes visited earlier might not be the nodes which are closer to the root node. As a result, the first path found in DFS might not be the shortest path.

- While it may seem like stacks are unnecessary when implementing DFS recursively, even recursive implementations use an implicit stack provided by the system. This is known as **the Call Stack**. Say more about recursion throughout this section.

- The advantage of the recursion solution is that it is easier to implement. However, there is a huge disadvantage: if the depth of recursion is too high, a stack overflow will result. In that case, BFS may be a good solution, as may implementing DFS using an explicit stack.

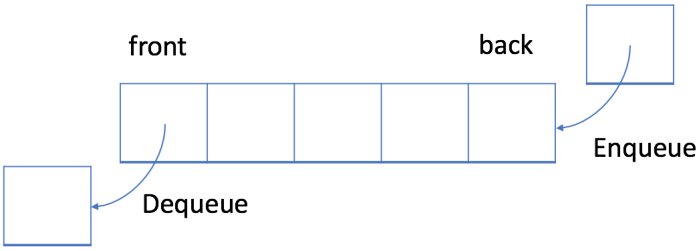
Recursion

Details here

[1] This process works for general graphs as well as trees, as described in subsequent passages.

Queues and BFS

- Queues are like lines at the movies: First in, first out (FIFO).
  - The insert operation for a queue is called *enqueue*. Inserting always happens at the end.
  - The delete for a queue is called *dequeue*. Removing always happens from the front.



- Different languages:
  - Queues are easy to implement using e.g. `ArrayLists` in Java. However, the naive implementation may be inefficient in some cases, as the movement of the start pointer means more and more space will be wasted. This will be unacceptable when facing a space limitation.

    More about this is discussed below.
  - In C++, there are multiple ways to implement a queue.

    Using an array as the physical storage, one may choose between a *one-pointer* implementation (a `rear` pointer) or a *two-pointer* implementation (a `front` pointer and a `rear` pointer). In the one-pointer approach, deleting elements takes $O(n)$ time because elements must be shifted as head elements are deleted; in the two-pointer approach, both insert and delete are $O(1)$, but unless a *circular* option is implemented, there will be wasted space as elements are inserted and deleted (see details in the **Circular Queue** subsection below).

    Using a linked list, finish here...

Circular Queue

Thoroughly discuss the wasted space issue here. Include pictures if possible!

- Even better than this linear implementation is a *circular* implementation: This can be done using a circularly-linked list or a non-circular array (using two pointers and the `mod` operation).

Breadth-First Search (BFS)

- Queues are commonly used when doing a breadth-first search (BFS).[2]
  - In the first round, we process the root node. In the second round, we process the nodes next to the root node. In the third round, we process the nodes which are two steps from the root node; so on and so forth.

    Similar to tree's level-order traversal, the nodes closer to the root node will be traversed earlier.

[2] This process works for general graphs as well as trees, as described in subsequent passages.

- If a node X is added to the queue in the kth round, the length of the shortest path between the root node and X is exactly k.

  That is to say, you are already in the shortest path the first time you find the target node.

- The root node is enqueued first. Then in each round, we process the nodes which are already in the queue one-by-one and add all their neighbors to the queue. **Note:** The newly-added nodes will *not* be traversed immediately but will be processed in the next round.

  The processing order of the nodes is the exact same order as how they were added to the queue, which is FIFO. That's why queues are used for breadth-first searches!

- Sometimes, the naive implementation will be problematic, and you'll have to ensure you never visit the same node twice (lest we get stuck in an infinite loop, like if our graph has a cycle).

  To avoid this, you can combine a hash table (hash set in particular) to the fray, as shown here. Sometimes, this is unnecessary: In tree traversal, for example, or if you want to add a node to the queue multiple times.

Priority Queues

- A priority queue is a special type of queue in which each element is associated with a priority value.

- In a (normal) queue, the first-in-first-out rule is implemented; in a priority queue, the values are removed on the basis of priority.

  Priority queues can be implemented to first remove values with the highest or lowest associated priority.[3]

- Priority queue can be implemented using an array, a linked list, a heap, or a binary search tree. Among these data structures, heaps provide an efficient implementation of priority queues.

  Here are complexity comparisons of the various data structures, according to Programiz and (for building) Wikipedia:

Operation	**Peek**	**Enqueue**	**Dequeue**	**Building**
Linked List	$O(1)$	$O(n)$	$O(1)$	–
Binary Heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$
Binary Search Tree	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$

- As discussed on Wikipedia, priority queues can be used to sort and can be implemented using around a dozen different data structures (including some very specialized heaps such as *van Emde Boas trees* and *fusion trees*).

  There is also an entire section on the time complexities of various heap data structures (which are not copied here because of how specialized they are).

- One array-based and/or queue-based implementation is as follows:

  - Determine the priorities $1, 2, \ldots, n$.

  - Create n different arrays (and/or queues) $Q_1, Q_2, \ldots, Q_n$. Array (and/or Queue) Q_i contains all the elements with priority i.

[3] If elements with the same priority occur, they are served according to their order in the queue.

- When you want to insert a new element x of priority j, use FIFO insertion of x into Q_j.
- When you want to delete a new element, you must *only* delete from Q_1 until it's empty, then from Q_2, then Q_3, etc. Individual deletions follow FIFO.

  Deletion should always be done from highest-priority non-empty array (and/or queue).

Double Ended Queues (DEQueues)

- Not strictly following FIFO.
- More details to come.

Queue	**Insert**	**Delete**
Front	No	Yes
Rear	Yes	No

DEQueue	**Insert**	**Delete**
Front	Yes	Yes
Rear	Yes	Yes

Input Restricted and Output Restricted DEQueues

- Not strictly following FIFO.
- More details to come.

Input Restricted	**Insert**	**Delete**
Front	No	Yes
Rear	Yes	Yes

Output Restricted	**Insert**	**Delete**
Front	Yes	Yes
Rear	Yes	No

Search Algorithms

Basics & Terminology

- A search algorithm is said to be *in-place* if it does not need an extra space, i.e. if it produces an output in the same memory that contains the data being sorted.

  This is done by transforming the input "in-place," so extra memory is not needed (though a small constant amount of extra space used for variables is allowed).

- A sorting algorithm is said to be *stable* if two objects with equal keys appear in the same order after sorting as they appear in the input array pre-sorting.

Complexity Summary

Most of the table summaries come from GeeksForGeeks.

Algorithm	Time Complexity			Space Complexity	Notes
	Best	Average	Worst	Worst	
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	not stable, in-place
Stable Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	stable, in-place
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	
Heap Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n \log n)$	$O(1)$	
Quick Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n^2)$	$O(\log n)$	
Merge Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n \log n)$	$O(n)$	
Bucket Sort	$\Omega(n + k)$	$\theta(n + k)$	$O(n^2)$	$O(n)$	
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(n + k)$	$O(nk)$	
Count Sort	$\Omega(n + k)$	$\theta(n + k)$	$O(n + k)$	$O(k)$	

Shell Sort	$\Omega(n)$	$\theta(n \log n)$	$O(n \log n)$	$O(1)$	
Tim Sort	$\Omega(n)$	$\theta(n^2 \log^2(n))$	$O(n^2 \log^2(n))$	$O(n)$	
Tree Sort	$\Omega(n \log n)$	$\theta(n \log n)$	$O(n^2)$	$O(n)$	
Cube Sort	$\Omega(n)$	$\theta(n \log n)$	$O(n \log n)$	$O(n)$	

Individual Algorithms

Here, we collect details for all (well, *most*...) of the sorting algorithms commonly encountered.

Selection Sort

Extended analysis done here.

The selection sort algorithm sorts an array by repeatedly finding the minimum element from the currently-unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array: (i) a subarray which is already sorted, and (ii) the remaining subarray which is unsorted. In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

Example: Start with the array {64,25,12,22,11}:

```
Step 1: Traverse whole list to find minimium = 11.
Step 2: Swap element 1 for 11. Result: {11,25,12,22,64}.
Step 3: Traverse list starting at 25 to find minimum = 12.
Step 4: Swap element 2 for 12. Result: {11,12,25,22,64}.
Step 5: Traverse list starting at 25 to find minimum = 22.
Step 6: Swap element 3 for 22. Result: {11,12,22,25,64}.
Step 7: Traverse list starting at 25 to find minimum = 25. Do nothing.
Step 8: Traverse list starting at 64 to find minimum = 64. Do nothing.
```

- **Complexity:** The complexity here is $O(n^2)$, since you need one loop to select an element of the array followed by a second to compare that element with all the others.

- **Stability:** This algorithm *isn't* stable.

  However, it can be made stable by implementing stable seleection sort instead. The idea here is: Instead of swapping, the minimum element may be placed in its final position by placing the number in the desired position and then pushing every subsequent element one step forward (instead of swapping).

  This is similar to the methodology employed by Insertion Sort.

Bubble Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu,

pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Insertion Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Heap Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Quick Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Merge Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bucket Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Radix Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Count Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Shell Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Tim Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a,

magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Tree Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Cube Sort

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.