### Screen Captures

*All tests use the same fruit dataset for consistency.*

**Binary Tree Search**

**A black background with white text

AI-generated content may be incorrect.**

**Binary Search**

**A black background with white text

AI-generated content may be incorrect.**

**Sequential Search**

**A black background with white text

AI-generated content may be incorrect.**

The binary tree search is good for finding elements in a balanced tree and so long as it's balanced it guarantees a time complexity of O(log n). The binary search tree performs comparatively to the binary search. But if the binary tree is unbalanced, basically a linked list, it devolves to O(n) making it less efficient than the Binary Search. Sequential Search is less efficient than both at the best case, and the same as a binary search tree in its worst case due to it's O(n) time complexity, making it not suitable for large datasets.

**Bubble Sort**



**Merge Sort**



**Quick Sort**



The merge sort method is the most efficient for working with large lists, it guarantees an O(n log n) for time complexity. Compared to the quick sort they're about the same for large lists but can downgrade to O(n^2). Bubble sort is the least efficient, since it guarantees an O(n^2) time complexity, it does poorly compared to the other two and does terrible with large datasets.

### Python Code

Binary\_tree\_string\_list.py

from binary\_tree import BinaryTree

class BinaryTreeStringList:

    def \_\_init\_\_(self):

        self.tree = BinaryTree()

    def add(self, string):

        self.tree.add(string)

    def find(self, string):

        return self.tree.find(string)

Binary\_tree\_test.py

import timeit

from binary\_tree\_string\_list import BinaryTreeStringList

def test\_binary\_tree():

    bt\_list = BinaryTreeStringList()

    strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon",

               "mango", "nectarine", "orange", "papaya", "quince", "raspberry", "strawberry", "tangerine", "ugli", "watermelon"]

    for s in strings:

        bt\_list.add(s)

    # Test finding an existing word

    time\_existing = timeit.timeit(lambda: bt\_list.find("kiwi"), number=1000)

    print(f"Binary Tree Search (existing): {time\_existing:.10f} seconds")

    # Test finding a non-existing word

    time\_non\_existing = timeit.timeit(lambda: bt\_list.find("blueberry"), number=1000)

    print(f"Binary Tree Search (non-existing): {time\_non\_existing:.10f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_binary\_tree()

Binary\_tree.py

class TreeNode:

    def \_\_init\_\_(self, key):

        self.left = None

        self.right = None

        self.val = key

class BinaryTree:

    def \_\_init\_\_(self):

        self.root = None

    def add(self, key):

        if self.root is None:

            self.root = TreeNode(key)

        else:

            self.\_add(self.root, key)

    # Recursive helper function to add a key to the tree

    def \_add(self, node, key):

        if key < node.val:

            if node.left is None:

                node.left = TreeNode(key)

            else:

                self.\_add(node.left, key)

        else:

            if node.right is None:

                node.right = TreeNode(key)

            else:

                self.\_add(node.right, key)

    def find(self, key):

        return self.\_find(self.root, key)

    # Recursive helper function to find a key in the tree

    def \_find(self, node, key):

        if node is None:

            return None

        if node.val == key:

            return node.val

        elif key < node.val:

            return self.\_find(node.left, key)

        else:

            return self.\_find(node.right, key)

binary\_string\_list.py

class BinaryStringList:

    def \_\_init\_\_(self):

        self.internal\_list = []

    def add(self, string):

        self.internal\_list.append(string)

        self.internal\_list.sort()  # Ensures the list is sorted for binary search

    def find(self, string):

        left, right = 0, len(self.internal\_list) - 1

        while left <= right:

            mid = (left + right) // 2

            if self.internal\_list[mid] == string:

                return string

            elif self.internal\_list[mid] < string:

                left = mid + 1

            else:

                right = mid - 1

        return None

binary\_test.py

import timeit

from binary\_string\_list import BinaryStringList

def test\_binary():

    bin\_list = BinaryStringList()

    strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon",

               "mango", "nectarine", "orange", "papaya", "quince", "raspberry", "strawberry", "tangerine", "ugli", "watermelon"]

    for s in strings:

        bin\_list.add(s)

    # Test finding an existing word

    time\_existing = timeit.timeit(lambda: bin\_list.find("kiwi"), number=1000)

    print(f"Binary Search (existing): {time\_existing:.10f} seconds")

    # Test finding a non-existing word

    time\_non\_existing = timeit.timeit(lambda: bin\_list.find("blueberry"), number=1000)

    print(f"Binary Search (non-existing): {time\_non\_existing:.10f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_binary()

sequential\_string\_list.py

class SequentialStringList:

    def \_\_init\_\_(self):

        self.internal\_list = []

    def add(self, string):

        self.internal\_list.append(string)

    def find(self, string):

        for item in self.internal\_list:

            if item == string:

                return item

        return None

sequential\_test.py

import timeit

from sequential\_string\_list import SequentialStringList

def test\_sequential():

    seq\_list = SequentialStringList()

    strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon",

               "mango", "nectarine", "orange", "papaya", "quince", "raspberry", "strawberry", "tangerine", "ugli", "watermelon"]

    for s in strings:

        seq\_list.add(s)

    # Test finding an existing word

    time\_existing = timeit.timeit(lambda: seq\_list.find("kiwi"), number=1000)

    print(f"Sequential Search (existing): {time\_existing:.10f} seconds")

    # Test finding a non-existing word

    time\_non\_existing = timeit.timeit(lambda: seq\_list.find("blueberry"), number=1000)

    print(f"Sequential Search (non-existing): {time\_non\_existing:.10f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_sequential()

bubble\_string\_list.py

class BubbleStringList:

    def \_\_init\_\_(self):

        self.internal\_list = []

    def add(self, string):

        self.internal\_list.append(string)

    def sort(self):

        n = len(self.internal\_list)

        for i in range(n):

            for j in range(0, n-i-1):

                if self.internal\_list[j] > self.internal\_list[j+1]:

                    self.internal\_list[j], self.internal\_list[j+1] = self.internal\_list[j+1], self.internal\_list[j]

bubble\_test.py

import timeit

from bubble\_string\_list import BubbleStringList

def test\_bubble():

    bubble\_list = BubbleStringList()

    strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon",

               "mango", "nectarine", "orange", "papaya", "quince", "raspberry", "strawberry", "tangerine", "ugli", "watermelon"]

    for s in strings:

        bubble\_list.add(s)

    # Test sorting

    time\_sort = timeit.timeit(lambda: bubble\_list.sort(), number=1)

    print(f"Bubble Sort: {time\_sort:.10f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_bubble()

merge\_string\_list.py

class MergeStringList:

    def \_\_init\_\_(self):

        self.internal\_list = []

    def add(self, string):

        self.internal\_list.append(string)

    def sort(self):

        self.internal\_list = self.\_merge\_sort(self.internal\_list)

    def \_merge\_sort(self, lst):

        if len(lst) > 1:

            mid = len(lst) // 2

            left\_half = lst[:mid]

            right\_half = lst[mid:]

            self.\_merge\_sort(left\_half)

            self.\_merge\_sort(right\_half)

            i = j = k = 0

            while i < len(left\_half) and j < len(right\_half):

                if left\_half[i] < right\_half[j]:

                    lst[k] = left\_half[i]

                    i += 1

                else:

                    lst[k] = right\_half[j]

                    j += 1

                k += 1

            while i < len(left\_half):

                lst[k] = left\_half[i]

                i += 1

                k += 1

            while j < len(right\_half):

                lst[k] = right\_half[j]

                j += 1

                k += 1

        return lst

merge\_test.py

import timeit

from merge\_string\_list import MergeStringList

def test\_merge():

    merge\_list = MergeStringList()

    strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon",

               "mango", "nectarine", "orange", "papaya", "quince", "raspberry", "strawberry", "tangerine", "ugli", "watermelon"]

    for s in strings:

        merge\_list.add(s)

    # Test sorting

    time\_sort = timeit.timeit(lambda: merge\_list.sort(), number=1)

    print(f"Merge Sort: {time\_sort:.10f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_merge()

quick\_string\_list.py

class QuickStringList:

    def \_\_init\_\_(self):

        self.internal\_list = []

    def add(self, string):

        self.internal\_list.append(string)

    def sort(self):

        self.\_quick\_sort(0, len(self.internal\_list) - 1)

    def \_quick\_sort(self, low, high):

        if low < high:

            pi = self.\_partition(low, high)

            self.\_quick\_sort(low, pi - 1)

            self.\_quick\_sort(pi + 1, high)

    def \_partition(self, low, high):

        pivot = self.internal\_list[high]

        i = low - 1

        for j in range(low, high):

            if self.internal\_list[j] < pivot:

                i += 1

                self.internal\_list[i], self.internal\_list[j] = self.internal\_list[j], self.internal\_list[i]

        self.internal\_list[i + 1], self.internal\_list[high] = self.internal\_list[high], self.internal\_list[i + 1]

        return i + 1

quick\_test.py

import timeit

from quick\_string\_list import QuickStringList

def test\_quick():

    quick\_list = QuickStringList()

    strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape", "honeydew", "kiwi", "lemon",

               "mango", "nectarine", "orange", "papaya", "quince", "raspberry", "strawberry", "tangerine", "ugli", "watermelon"]

    for s in strings:

        quick\_list.add(s)

    # Test sorting

    time\_sort = timeit.timeit(lambda: quick\_list.sort(), number=1)

    print(f"Quick Sort: {time\_sort:.10f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_quick()