**Project One**

Nathalie Morales

CS-300: Data Structures & Algorithms

Southern New Hampshire University

06/19/2024

# Introduction

This document presents the pseudocode and runtime analysis for three data structures: Vector, Hash Table, and Binary Search Tree. Additionally, it includes an analysis of the advantages and disadvantages of each data structure.

# Pseudocode

## Vector

# Vector Pseudocode  
  
class Vector:  
 def \_\_init\_\_(self):  
 self.data = []  
   
 def insert(self, value):  
 self.data.append(value)  
   
 def delete(self, value):  
 if value in self.data:  
 self.data.remove(value)  
   
 def search(self, value):  
 for i in range(len(self.data)):  
 if self.data[i] == value:  
 return i  
 return -1

## Hash Table

# Hash Table Pseudocode  
  
class HashTable:  
 def \_\_init\_\_(self, size):  
 self.size = size  
 self.table = [[] for \_ in range(size)]  
   
 def hash\_function(self, key):  
 return key % self.size  
   
 def insert(self, key, value):  
 hash\_key = self.hash\_function(key)  
 key\_exists = False  
 for i, kv in enumerate(self.table[hash\_key]):  
 k, v = kv  
 if key == k:  
 key\_exists = True  
 break  
 if key\_exists:  
 self.table[hash\_key][i] = (key, value)  
 else:  
 self.table[hash\_key].append((key, value))  
   
 def delete(self, key):  
 hash\_key = self.hash\_function(key)  
 for i, kv in enumerate(self.table[hash\_key]):  
 k, v = kv  
 if key == k:  
 del self.table[hash\_key][i]  
   
 def search(self, key):  
 hash\_key = self.hash\_function(key)  
 for kv in self.table[hash\_key]:  
 k, v = kv  
 if key == k:  
 return v  
 return None

## Binary Search Tree

# Binary Search Tree Pseudocode  
  
class Node:  
 def \_\_init\_\_(self, key):  
 self.left = None  
 self.right = None  
 self.value = key  
  
class BinarySearchTree:  
 def \_\_init\_\_(self):  
 self.root = None  
   
 def insert(self, key):  
 if self.root is None:  
 self.root = Node(key)  
 else:  
 self.\_insert(self.root, key)  
   
 def \_insert(self, root, key):  
 if key < root.value:  
 if root.left is None:  
 root.left = Node(key)  
 else:  
 self.\_insert(root.left, key)  
 else:  
 if root.right is None:  
 root.right = Node(key)  
 else:  
 self.\_insert(root.right, key)  
   
 def delete(self, key):  
 self.root = self.\_delete(self.root, key)  
   
 def \_delete(self, root, key):  
 if root is None:  
 return root  
 if key < root.value:  
 root.left = self.\_delete(root.left, key)  
 elif key > root.value:  
 root.right = self.\_delete(root.right, key)  
 else:  
 if root.left is None:  
 return root.right  
 elif root.right is None:  
 return root.left  
 temp\_val = self.\_min\_value\_node(root.right)  
 root.value = temp\_val.value  
 root.right = self.\_delete(root.right, temp\_val.value)  
 return root  
   
 def \_min\_value\_node(self, node):  
 current = node  
 while current.left is not None:  
 current = current.left  
 return current  
   
 def search(self, key):  
 return self.\_search(self.root, key)  
   
 def \_search(self, root, key):  
 if root is None or root.value == key:  
 return root  
 if key < root.value:  
 return self.\_search(root.left, key)  
 return self.\_search(root.right, key)

# Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Operation** | **Average Case** | **Worst Case** |
| Vector | Insertion | O(1) | O(n) |
|  | Deletion | O(n) | O(n) |
|  | Search | O(n) | O(n) |
| Hash Table | Insertion | O(1) | O(n) |
|  | Deletion | O(1) | O(n) |
|  | Search | O(1) | O(n) |
| Binary Search Tree | Insertion | O(log n) | O(n) |
|  | Deletion | O(log n) | O(n) |
|  | Search | O(log n) | O(n) |

**Advantages and Disadvantages**

## Vector

Advantages:  
- Simple and easy to implement.  
- Provides random access to elements.  
- Dynamic resizing is handled automatically.  
  
Disadvantages:  
- Insertion and deletion can be slow, especially if done frequently.  
- May lead to memory wastage if the capacity exceeds the actual number of elements.

## Hash Table

Advantages:  
- Provides very fast insertion, deletion, and search operations on average.  
- Handles large datasets efficiently.  
  
Disadvantages:  
- Requires a good hash function to avoid collisions.  
- Performance can degrade in the worst case due to collisions.  
- Memory usage can be higher due to storage of additional information (e.g., linked lists for chaining).

## Binary Search Tree

Advantages:  
- Provides efficient search, insertion, and deletion operations, especially when balanced.  
- Maintains elements in a sorted order.  
  
Disadvantages:  
- Performance can degrade to O(n) in the worst case if the tree becomes unbalanced.  
- Requires additional effort to maintain balance (e.g., using AVL trees or Red-Black trees).

# Conclusion

In this project, I have explored three fundamental data structures: Vector, Hash Table, and Binary Search Tree. Each data structure offers unique advantages and is suited to different types of applications. By analyzing their pseudocode, runtime complexities, and pros and cons, I gain a deeper understanding of their practical use cases and performance characteristics.

## File Handling Pseudocode

def read\_file(file\_path):  
 with open(file\_path, 'r') as file:  
 lines = file.readlines()  
 return lines  
  
def write\_file(file\_path, data):  
 with open(file\_path, 'w') as file:  
 file.writelines(data)

## Menu Functionality Pseudocode

def display\_menu():  
 print("1. Insert Data")  
 print("2. Delete Data")  
 print("3. Search Data")  
 print("4. Print Sorted List")  
 print("5. Exit")  
  
def menu():  
 while True:  
 display\_menu()  
 choice = input("Enter your choice: ")  
 if choice == '1':  
 insert\_data()  
 elif choice == '2':  
 delete\_data()  
 elif choice == '3':  
 search\_data()  
 elif choice == '4':  
 print\_sorted\_list()  
 elif choice == '5':  
 break  
 else:  
 print("Invalid choice. Please try again.")

## Sorted List Printing Pseudocode

def print\_sorted\_list(data\_structure):  
 sorted\_list = data\_structure.get\_sorted\_list()  
 for item in sorted\_list:  
 print(item)