UNIVERSITY OF NAIROBI

FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTING AND INFORMATICS

ANALYSIS AND DESIGN OF ALGORITHMS

PROJECT NUMBER 1

BY

NAME	REG NO:
IGADWA DEBI TELOORI	P15/1906/2020
WEKULO FIONA ABIGAEL	P15/1905/2020
MWANGI ENOCH BABU	P15/140432/2020
KARANJA SAMUEL MWANGI	P15/1933/2020

LECTURER: ELISHA OPIYO

January 2023

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

In computer science, a list is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. Lists are an important data structure in many programming languages, and are used to store collections of items that can be modified and processed. Lists are similar to arrays, but can contain elements of different types, and can be resized dynamically. Lists are also known as sequences or arrays in some programming languages.

Lists are often implemented as dynamic arrays, which means that the size of the list can grow or shrink as needed to accommodate the number of elements in the list. Lists can be accessed by their position in the list, using an index starting at 0 for the first element. Lists also support operations such as inserting and deleting elements, searching for elements, and sorting the elements.

There are several types of lists that can be used in programming, including linked lists, arrays, and dynamic arrays. Each type of list has its own set of characteristics and behaviors, and is best suited for certain types of applications.

In general, lists are a powerful and flexible data structure that can be used to store and manipulate large amounts of data in a variety of applications.

1.2 OBJECTIVES

2. Investigate various lists
3. Implement a list.
4. Investigate the performance of your implementation theoretically.
5. Investigate the performance of your implementation experimentally using two different search methods on input streams of sizes 50, 200 and 500.
6. In each case find the best- and worst-case time duration.

1.3 SIGNIFICANCE OF STUDY

1. Efficiency: Algorithms are used to process large amounts of data, and it is important to design algorithms that can do so efficiently in order to save time and resources. By analyzing algorithms, we can determine how fast they run and how much memory they use, which can help us choose the most efficient algorithm for a particular task.
2. Correctness: Algorithms are used to solve problems and make decisions, and it is important that they produce the correct results. By designing and analyzing algorithms, we can ensure that they are correct and reliable.
3. Problem-solving: Designing algorithms requires problem-solving skills and the ability to think logically and abstractly. This can help improve problem-solving skills and develop critical thinking.
4. Innovation: Algorithms are at the heart of many modern technologies and innovations. By designing and analyzing algorithms, we can create new and improved ways to solve problems and perform tasks.

CHAPTER 2: INVESTIGATING VARIOUS LISTS

A list is an abstract data type that represents a sequence of elements. Some common operations on lists include inserting and deleting elements, searching for a particular element, and iterating through the elements in the list.

There are several different ways to implement a list, including using an array, a linked list, or a dynamic array.

Arrays: An array-based list stores the elements in a contiguous block of memory, which allows for fast access to any element using its index. However, inserting and deleting elements can be inefficient, as it may require shifting all of the elements after the inserted or deleted element to make room for the new element or fill the gap left behind.

Linked lists**:** A linked list, on the other hand, stores the elements in a chain of nodes, where each node contains a reference to the next element in the list. This allows for efficient insertion and deletion, as only the references in the neighboring nodes need to be updated. However, accessing an element by its index can be slow, as it requires traversing the list from the beginning.

A doubly linked list is a type of list in which each node contains references to both the next and previous nodes in the list. This allows you to easily traverse the list in both directions, as well as perform operations such as inserting and deleting nodes from the list. One advantage of using a doubly linked list is that you can easily traverse the list in either direction, which can be useful in certain situations. However, doubly linked lists can take up more memory than some other types of lists, because each node requires references to both the next and previous nodes.

Dynamic arrays**:** A dynamic array uses an array to store the elements but automatically resizing the array when it becomes full or when elements are deleted. This allows for fast access to elements using their index, as well as efficient insertion and deletion.

Stacks**:** A stack is a type of list that follows the last-in, first-out (LIFO) principle. This means that the last item added to the stack is the first one to be removed. Stacks are used for storing temporary data and implementing undo/redo functionality.

Queues**:** A queue is a type of list that follows the first-in, first-out (FIFO) principle. This means that the first item added to the queue is the first one to be removed. Queues are used for storing data that needs to be processed in a specific order, such as tasks in a task scheduler.

Trees: A tree is a type of list that has a hierarchical structure, with a root node and one or more levels of child nodes. Trees are used for storing data in a way that allows for fast searching and sorting.

CHAPTER 3: ANALYSIS AND DESIGN OF ALGORITHMS

In computer science, a search is an algorithm that helps to find an item in a collection of items. A search algorithm takes in a collection of items and a target item, and returns a value indicating whether the target item is present in the collection. The search algorithms used in this study include:

3.1 Linear search:

This algorithm searches through the items one by one, in sequence, until it finds the target item or determines that the item is not present. Linear search has a time complexity of O(n), meaning that the time taken to perform the search increases linearly with the number of items in the collection. Linear search is easy to implement and understand, but it is not very efficient when the collection is large. If the collection contains a million items, the worst-case time complexity of linear search would be O (1 million), which is much slower than other search algorithms such as binary search. Linear search is best suited for small collections or for situations where the cost of comparing items is very high and the collection is not expected to grow much over time.

PSEUDOCODE	ALGORITHM

def linear_search (collection, target):
 	
for item in collection:
 	
if item == target:
 	
return True
 	
return False
	This implementation iterates through the items in the collection one by one, and returns True as soon as it finds the target item. If the target item is not present in the collection, the function will return False after it has checked all of the items.

3.2 Binary search

This is an efficient search algorithm that works on sorted collections. It has a time complexity of O(log n), meaning that the time taken to perform the search increases logarithmically with the number of items in the collection. Binary search is more efficient than linear search, but it requires that the collection be sorted. It is also not well-suited for situations where the collection is expected to change frequently, as the collection must be kept sorted in order for the algorithm to work correctly.

PSEUDOCODE	ALGORITHM

def binary_search (collection, target):
low = 0
high = len(collection) - 1
while low <= high:
mid = (low + high) // 2
if collection[mid] == target:
return True
elif collection[mid] < target:
low = mid + 1
else:
high = mid - 1
return False
	The basic idea behind binary search is to repeatedly divide the search space in half until the target item is found or it is determined that the item is not present in the collection. This is done by comparing the target item to the middle element of the collection. If the target item is smaller than the middle element, then the algorithm searches the left half of the collection. If the target item is larger than the middle element, then the algorithm searches the right half of the collection. This process is repeated Until the target item is found or the search space is empty.

CHAPTER 4: IMPLEMENTATION AND TESTING

4.1 IMPLEMENTATION

The list used in this study is an array. It is implemented using the Python language.

4.1.1 LINEAR SEARCH

This implementation first reads in a stream of 50,200 or 500 integers from the user and stores them in a list called collection. It then prompts the user for a target item to search for and stores it in the variable target. Finally, it calls the linear_search function to search for the target item in the collection, and prints the result of the search. If the target item is found in the collection, the program will print "Target found in collection". If the target item is not found in the collection, the program will print "Target not found in collection".

THEORETICAL PERFORMANCE

The time complexity of linear search is O(n), meaning that the time taken to perform the search increases linearly with the number of items in the collection.

Best case

In the best-case scenario, linear search will find the target item on the first iteration of the loop and will return immediately, with a time complexity of O (1) for all the input stream sizes. This occurs when the target item is the first item in the collection.

Worst case

In the worst-case scenario, linear search will have to iterate through the entire collection before it determines that the target item is not present, with a time complexity of O(n). This occurs when the target item is not present in the collection. Hence if we are using linear search to search through an input stream of 50, 200 and 500 integers, the time complexity of the search would be O(50), O(200), O(500) respectively = O(n), where n is the number of items in the collection. This means that the time taken to perform the search would be directly proportional to the number of items in the collection.

EXPERIMENTAL PERFORMANCE

INPUT STREAM OF 50	
Best case duration	Worst case duration
Beginning of search= 1673105517.5148091 End of search= 1673105517.5148091 Total time of search= 0.0 seconds	Beginning of search= 1673105559.6055453 End of search= 1673105559.6055455 Total time of search= 0.0000002 seconds
INPUT STREAM OF 200	
Best case duration	Worst case duration

Beginning of search= 1673105701.0689535 End of search= 1673105701.0689535 Total time of search= 0.0 seconds	Beginning of search= 1673105730.244737 End of search= 1673105730.244745 Total time of search= 0.000008 seconds
INPUT STREAM OF 500	
Best case duration	Worst case duration
Beginning of search= 1673105826.647175 End of search= 1673105826.647175 Total time of search= 0.0 seconds	Beginning of search= 1673105969.744874 End of search= 1673105969.744874 Total time of search= 0.000020 seconds

4.1.2 BINARY SEARCH

This implementation first reads in a stream of 50, 200 or 500 integers from the user and stores them in a list called collection. It then sorts the collection using the sort method. It then prompts the user for a target item to search for and stores it in the variable target. Finally, it calls the binary_search function to search for the target item in the collection, and prints the result of the search.

If the target item is found in the collection, the program will print "Target found in collection". If the target item is not found in the collection, the program will print "Target not found in collection".

It is important to note that the binary_search function will only work correctly if the collection is sorted. If the collection is not sorted, the function may return incorrect results or may not terminate.

THEORETICAL PERFORMANCE

The time complexity of binary search is O(log n), meaning that the time taken to perform the search increases logarithmically with the number of items in the collection.

Best case

In the best-case scenario, binary search will find the target item on the first iteration of the loop and will return immediately, with a time complexity of O (log 1) = O (1). This occurs when the target item is the middle element of the collection.

Worst case

In the worst-case scenario, binary search will have to iterate logarithmically with the number of items in the collection before it determines that the target item is not present. This occurs when the target item is not present in the collection and is not equal to the middle element of the collection at any point during the search. Hence if we are using binary search to search through an input stream of 50, 200 and 500 integers, the time complexity of the search would be O(log 50), O(log 200), O(log 500) respectively = O(log n), where n is the number of items in the collection.

EXPERIMENTAL PERFORMANCE

INPUT STREAM OF 50	
Best case duration	Worst case duration

Beginning of search= 1673106023.0343013 End of search= 1673106023.0343013 Total time of search= 0.0 seconds	Beginning of search= 1673106132.0217109 End of search= 1673106132.0217110 Total time of search= 0.0000001 seconds
INPUT STREAM OF 200	
Best case duration	Worst case duration
Beginning of search= 1673106220.8492794 End of search= 1673106220.8492794 Total time of = 0.0 seconds	Beginning of search= 1673106243.7166564 End of search= 1673106243.7166568 Total time of = 0.0000004 seconds
INPUT STREAM OF 500	
Best case duration	Worst case duration
Beginning of search= 1673106319.65079 End of search= 1673106319.65079 Total time of search= 0.0 seconds	Beginning of search= 1673106349.7625413 End of search= 1673106349.7625423 Total time of search= 0.0000010 seconds

CHAPTER 5: CONCLUSION

In conclusion, binary search is more efficient than linear search as it takes less time even on the worst-case scenario on increasing input size.