9/29/2024

Assignment 6

Medians and Order Statistics & Elementary Data Structures

Sujan Dumaru

Student ID: 005030060

MSCS 532: Algorithms and Data Structures

Dr. Vanessa Cooper

INTRODUCTION

Like soring algorithms, selection algorithms play a crucial role in various applications, such as finding the median and order statistics. Several theoretically interesting algorithms achieve a worst-case running time of $\Theta (n)$. Notably, the Median of Medians algorithm guarantees a worst-case linear time solution, while Randomized Quickselect offers an expected linear time solution. Furthermore, elementary data structures—like arrays, stacks, queues, and linked lists—are essential for efficient data manipulation and storage. Mastering the implementation and analysis of these algorithms and data structures is vital for developing efficient systems.

SELECTION ALGORITHMS

1. Deterministic Selection Algorithm (Median of Medians)

The median-of-medians algorithm is a deterministic linear-time selection method that divides a list into sub lists (usually groups of 5 elements) and computes the approximate median of each sub list. This median is then used as a pivot for partitioning the entire array. The algorithm guarantees a worst-case time complexity of $O (n)$. The implementation of the median-of-medians algorithm is given below along with the output.

The worst-case time complexity of this algorithm is $O (n)$, achieved by ensuring that the pivot element divides the array into partitions that shrink at a guaranteed rate: $T(n) \leq T(n/5) + T(7n/10) + O(n)$ (Cormen et al., 2022). Additionally, the space complexity is $O (1)$, as the algorithm operates in place, except for temporary arrays used during partitioning.

```python
""" Implementation of Deterministic Selection Algorithm (Median-of-Medians) """

def median_of_medians(array, k):
    """ Function to determine median in the array """
    if len(array) <= 5:
        return sorted(array)[k]  # Sort the array and return the k-th smallest element

    # Divide array into groups of 5 and find medians
    sublists = [array[i:i+5] for i in range(0, len(array), 5)]
    medians = [sorted(sublist)[len(sublist) // 2] for sublist in sublists]

    # Find the median of medians which serve as pivot element
    pivot_element = median_of_medians(medians, len(medians) // 2)

    # Partition the array around the median of medians
    low = [j for j in array if j < pivot_element]
    high = [j for j in array if j > pivot_element]
    pivot_count = len(array) - len(low) - len(high)

    # Recurrence based on the position of k
    if k < len(low):
        return median_of_medians(low, k)
    elif k < len(low) + pivot_count:
        return pivot_element
    else:
        return median_of_medians(high, k - len(low) - pivot_count)

A = [1, 25, 30, 4, 5, 1000, 8, 9, 99]
B = [15, 32, 43, 74, 56, 60]
C = [1, 2, 3, 4, 5]

print(median_of_medians(A, 0)) #should be 1
print(median_of_medians(A, 7)) #should be 99
print(median_of_medians(B, 4)) #should be 60
print(median_of_medians(C, 3)) #should be 4
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python -u "z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding\MSCS532_Assignment6\median_of_medians.py"
1
99
60
4
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 1: Implementation of Median-of-Medians.

2. Randomized Selection Algorithm (Randomized Quickselect)

Randomized Quickselect is a selection algorithm designed to find the kth smallest element in an unordered list. Similar to Randomized Quicksort, it selects a random pivot, partitions the array, and recursively searches in the partition that contains the kth smallest element. Below is the implementation of Randomized Quickselect:

```python
""" Implementation of Randomized Selection Algorithm (Randomized Quickselect) """

import random

def randomized_select(array, k):
    """ Function to find the k-th smallest element in the array """
    array_size = len(array)
    if array_size == 1:
        return array[0]             # If the array is single element, return it

    # Randomly choose a pivot element
    pivot = random.choice(array)

    # Step 2: Partition the array around the pivot
    low = [i for i in array if i < pivot]
    high = [i for i in array if i > pivot]
    pivot_count = array_size - len(low) - len(high)

    # Step 3: Recur based on the position of k
    if k < len(low):
        return randomized_select(low, k)
    elif k < len(low) + pivot_count:
        return pivot
    else:
        return randomized_select(high, k - len(low) - pivot_count)

A = [1, 25, 30, 4, 5, 1000, 8, 9, 99]
B = [15, 32, 43, 74, 56, 60]
C = [1, 2, 3, 4, 5]

print(randomized_select(A, 0)) #should be 1
print(randomized_select(A, 7)) #should be 99
print(randomized_select(B, 4)) #should be 60
print(randomized_select(C, 3)) #should be 4
```

```
PROBLEMS 7   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   GITLENS

PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python -u "z
f the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding\MSCS532_Assignment6\randomized_
"
1
99
60
4
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 2: Implementation of Randomized Quickselect.

Like Randomized Quicksort, the time complexity of Randomized Quickselect depends on pivot selection. Poor pivot choices can lead to a worst-case time complexity of $O(n^2)$, though this is mitigated by random pivot selection. The average time complexity, however, remains $O(n)$. Due to the creation of new arrays during partitioning, the space complexity of this algorithm is $O(n)$.

EMPIRICAL ANALYSIS

Both the Deterministic and Randomized Selection Algorithms were tested across various input sizes and distributions to evaluate their performance relative to each other. The execution times of these algorithms were compared on random, sorted, reverse-sorted, repeated elements, and large random arrays. The results are presented below:

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python
"
RANDOM ARRAY --> Total execution time: 0.01581810000061523
SORTED ARRAY --> Total execution time: 0.01527669999632053
REVERSED SORTED ARRAY --> Total execution time: 0.013200499997765291
REPEATED ELEMENTS ARRAY --> Total execution time: 0.014407599999685772
HUGE RANDOM ARRAY --> Total execution time: 1.4527489999964018
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python
"
RANDOM ARRAY --> Total execution time: 0.015059899997140747
SORTED ARRAY --> Total execution time: 0.014889700003550388
REVERSED SORTED ARRAY --> Total execution time: 0.012959499996213708
REPEATED ELEMENTS ARRAY --> Total execution time: 0.014220799996110145
HUGE RANDOM ARRAY --> Total execution time: 1.4145977000007406
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 3: Execution times for Median-of-Medians.

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python
ct.py"
RANDOM ARRAY --> Total execution time: 0.021069799993711058 seconds
SORTED ARRAY --> Total execution time: 0.018769700000120793 seconds
REVERSED SORTED ARRAY --> Total execution time: 0.01998049999383511 seconds
REPEATED ELEMENTS ARRAY --> Total execution time: 0.010680800005502533 seconds
HUGE RANDOM ARRAY --> Total execution time: 0.6769524999981513 seconds
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python
ct.py"
RANDOM ARRAY --> Total execution time: 0.02132300000084797 seconds
SORTED ARRAY --> Total execution time: 0.018858600000385195 seconds
REVERSED SORTED ARRAY --> Total execution time: 0.020148600000538863 seconds
REPEATED ELEMENTS ARRAY --> Total execution time: 0.011519599996972829 seconds
HUGE RANDOM ARRAY --> Total execution time: 0.6942221999997855 seconds
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 4:Execution times for Randomized Quickselect.

From the results above, both algorithms demonstrate consistency in selecting elements for random, sorted, and reverse-sorted arrays, with the Median-of-Medians slightly outperforming Randomized Quickselect. However, in the case of repeated elements, Randomized Quickselect is twice as fast as in the previous input distributions, surpassing the Median-of-Medians. A similar trend is observed with large random arrays, where Randomized Quickselect also shows superior performance.

ELEMENTARY DATA STRUCTURES

Elementary data structures form the foundational building blocks for organizing and manipulating data efficiently. They provide essential ways to store, retrieve, and manage data in computer programs. Common examples include arrays, stacks, queues, linked lists, and hash tables. Some of these are explained below:

1. Arrays and Matrices:

Arrays are among the most widely used data structures in programming due to their simplicity and ease of implementation. They offer constant-time access to elements and linear-time performance for insertion and deletion operations. In Python, arrays can be easily implemented using lists, making them accessible and versatile for various applications. The implementation of basic operations along with their time complexity is shown below:

```python
""" Implementation and Analysis of Elementary Data Structures """

class Array:
    """ Implementation of Array using Python lists """
    def __init__(self):
        self.array = []

    def insert(self, value):
        """ Insert value at the end of array """
        self.array.append(value)            # Takes O(1) time

    def delete(self, value):
        """ Delete the value from the array list """
        if value in self.array:
            self.array.remove(value)        # Remove the first occurrence (O(n)) worst case

    def access(self, index):
        """ Return the value at given index from the array """
        if 0 <= index < len(self.array):
            return self.array[index]        # Access element at given index (O(1))
        else:
            raise IndexError("Index out of bounds")
```

Figure 5: Implementation of Array.

2. Stacks and Queues:

Stacks and queues are fundamental data structures that manage collections of elements in distinct ways: stacks follow the Last In, First Out (LIFO) principle, while queues adhere to the First In, First Out (FIFO) principle. Each serves different purposes in programming. Below, we provide easy implementations of both stacks and queues using arrays, along with their time complexities for basic operations:

```
class Stack:
    """ Implementation of stack using array """
    def __init__(self):
        self.stack = []

    def push(self, value):
        """ Push the value in the end of stack """
        self.stack.append(value)        # Push operation (O(1))

    def pop(self):
        """ Pop the value from the end of stack and return it"""
        if len(self.stack) != 0:
            return self.stack.pop()     # Pop operation (O(1))
        else:
            raise IndexError("Stack is empty")

class Queue:
    """ Implementation of queue using array """
    def __init__(self):
        self.queue = []

    def enqueue(self, value):
        """ Add the value in the end of queue """
        self.queue.append(value)        # Add at the end of the list (O(1))

    def dequeue(self):
        """ Remove the value from the beginning of queue and return it"""
        if len(self.queue) != 0:
            return self.queue.pop(0)    # Remove the first element and shift rest of elements (O(n))
        else:
            raise IndexError("Queue is empty")
```

Figure 6: Implementation of Stack and Queue using array.

3. Linked Lists:

Linked lists are dynamic data structures composed of a sequence of elements called nodes. Each node contains a data value and a reference to the next node, making linked lists index-less. This structure enables efficient insertion and deletion of elements, as there is no need to shift other elements. However, accessing elements requires linear time, since traversal through the entire list is necessary to reach a specific node. There are different types of linked lists which are briefly explained below:

- **Singly Linked List**: Each node contains data and a single pointer to the next node. This allows for traversal in one direction.
- **Doubly Linked List**: Each node has two pointers: one pointing to the next node and another pointing to the previous node. This allows for traversal in both directions.
- **Circular Linked List**: In this variation, the last node points back to the first node, forming a circular structure. This can be implemented as either singly or doubly linked.

The implementation for singly linked list with the basic operations is shown below:

```python
class Node:
    """ Single node class to be used in linked list """
    def __init__(self, data):
        self.data = data                # Store value in the data
        self.next = None                # Pointer to next node

class SinglyLinkedList:
    """ Implementation of singly linked list """
    def __init__(self):
        self.head = None

    def insert(self, value):
        """ Insert the new value at the beginning of the list """
        new_node = Node(value)          # Create new node with the given value
        new_node.next = self.head
        self.head = new_node            # Insert at the head of list (O(1))

    def delete(self, value):
        """ Delete the first node that contain the given value """
        current_node = self.head        # Start at the head of the list
        prev_node = None

        while current_node:
            if current_node.data == value:              # If the node is found
                if prev_node:
                    prev_node.next = current_node.next  # Skip the current node and point to next node
                else:
                    self.head = current_node.next       # Update the head if the value is in head node
                return
            prev_node = current_node
            current_node = current_node.next            # Move to next node if value is not found

    def traverse(self):
        """ Traverse through the entire linked list and return the value of all nodes """
        values = []
        current_node = self.head                # Start at the head of the list

        while current_node:
            values.append(current_node.data)    # Store the node values in the list
            current_node = current_node.next    # Move to next node until the end of the list

        print("Linked List: ", values)
        return values
```

Figure 7: Implementation of Linked List.

PERFORMANCE ANALYSIS

```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python
ment6\elementary_data_structures.py"
INSERTION TIME:
Array Insertion Time --> Total execution time: 0.0005004999984521419
Stack Push Time --> Total execution time: 0.00040470000385539606
Queue Enqueue Time --> Total execution time: 0.00038860000495333225
Linked List Insertion Time --> Total execution time: 0.001402800000505522

ACCESSING TIME:
Array Accessing Time --> Total execution time: 0.0007136999993235804

DELETION TIME:
Array Deletion Time --> Total execution time: 0.09147489999304526
Stack Pop Time --> Total execution time: 0.0006742999976268038
Queue Dequeue Time --> Total execution time: 0.08714019999752054
Linked List Deletion Time --> Total execution time: 0.0008373000018764287
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 8: Performance analysis of different operations of data structures.

The above output illustrates the execution times for various operations—such as insertion, accessing, and deletion—across Arrays, Stacks, Queues, and Linked Lists. Arrays perform well for insertion (0.00050 seconds) and accessing (0.00071 seconds) due to their *O (1)* time complexity but exhibit slow deletion (0.09147 seconds) because of the need to shift elements, leading to *O (n)* complexity.

Stacks and queues, when implemented using arrays, show similar insertion times (0.00047 seconds and 0.00038 seconds, respectively) due to the *O (1)* nature of pushing and enqueuing operations. However, queue deletion (0.08714 seconds) is slower than stack deletion (0.00067 seconds) because of the element shifting required in the underlying list causing it to take *O (n)* time. Linked lists demonstrate efficient deletion (0.00083 seconds) since they avoid shifting by adjusting pointers, but their insertion time (0.00142 seconds) is slightly higher due to overhead associated with node creation. Nevertheless, the time complexity for insertion and deletion of data in Linked list is *O (1)*.

DISCUSSION AND PRACTICAL APPLICATIONS

Arrays are suitable for scenarios requiring constant-time access, while linked lists are preferred when frequent insertions and deletions are necessary. Stacks are useful for implementing LIFO operations, such as function call management and undo mechanisms in applications, whereas queues are preferred for scheduling, breadth-first search algorithms and order maintenance. Linked lists are useful in various applications, including implementing stacks and queues, managing memory allocation in systems, and building dynamic data structures like graphs and adjacency lists. However, due to overhead associated with node creation, array is more preferred to be used in stacks and queues. Overall, linked lists provide flexibility in data manipulation and are an essential concept in computer science and programming.

CONCLUSION

This offers a thorough examination that covers both theoretical and empirical analysis aspects of selection algorithms and basic data structures. The deterministic selection algorithm offers a worst-case guarantee, making it ideal for adversarial inputs, while the randomized algorithm provides better average-case performance. It is important to have an understanding of the performance and trade-offs of elementary data structures in order to make wise choices when designing algorithms and data structures.

REFERENCE

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509