# Assignment - 6

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 24, 2024

Github : https://github.com/VijayaKrishnaSameerajJonnavithula/Assignment-6

**Part 1: Implementation and Analysis of Selection Algorithms**

**Deterministic Algorithm: Median of Medians**

In the worst situation, the Median of Medians algorithm guarantees that finding the kth smallest element will take O(n) time.

Steps:

* Group the array into no more than five elements.
* Determinate the median for each group after sorting them.
* A recursive call to the same function yields to the median of the medians.
* Divide the array according to the median of the medians.

Recur: Return the pivot if k equals the pivot index.

Recur in the left subarray if k is less.

Recur in the right subarray otherwise.

Randomized Algorithm: Randomized Quickselect :

Randomized the average time complexity of Quickselect is O(n), while in the worst scenario, it is O(n 2).

Steps:

• Select a pivot at random from the array.

• Divide the array according to this pivot.

• Recur: Return the pivot if k equals the pivot index.

• Recur in the left subarray if k is less.

• Recur in the right subarray otherwise.

Code :

import random

# Deterministic Algorithm: Median of Medians

def median\_of\_medians(arr, k):

# Base case: if the array is small, sort it and return the k-th element

if len(arr) <= 5:

return sorted(arr)[k]

# Step 1: Divide the array into sublists of at most 5 elements

sublists = [arr[i:i + 5] for i in range(0, len(arr), 5)]

# Step 2: Find the median of each sublist

medians = [sorted(sublist)[len(sublist) // 2] for sublist in sublists]

# Step 3: Recursively find the median of the medians

pivot = median\_of\_medians(medians, len(medians) // 2)

# Step 4: Partition the array based on the pivot

low = [x for x in arr if x < pivot]

high = [x for x in arr if x > pivot]

# Find the rank of the pivot

pivot\_rank = len(low)

# Step 5: Recur based on the pivot's rank and k

if k < pivot\_rank:

return median\_of\_medians(low, k)

elif k > pivot\_rank:

return median\_of\_medians(high, k - pivot\_rank - 1)

else:

return pivot

# Randomized Algorithm: Randomized Quickselect

def randomized\_partition(arr, low, high):

pivot\_index = random.randint(low, high)

arr[pivot\_index], arr[high] = arr[high], arr[pivot\_index] # Move pivot to end

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] <= pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

def randomized\_quickselect(arr, low, high, k):

if low == high: # Only one element in the array

return arr[low]

pivot\_index = randomized\_partition(arr, low, high)

if pivot\_index == k:

return arr[pivot\_index]

elif k < pivot\_index:

return randomized\_quickselect(arr, low, pivot\_index - 1, k)

else:

return randomized\_quickselect(arr, pivot\_index + 1, high, k)

# Function to call randomized\_quickselect conveniently

def find\_kth\_smallest(arr, k):

return randomized\_quickselect(arr, 0, len(arr) - 1, k)

# Main function to test both algorithms

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

# Test array and k value

test\_array = [3, 2, 9, 1, 5, 7, 8, 6, 4]

k = 4 # Find the 4th smallest element (0-based index, so it's the 5th smallest in 1-based)

# Deterministic Algorithm Output

deterministic\_result = median\_of\_medians(test\_array.copy(), k)

print(f"Deterministic Algorithm: {k + 1}th smallest element is {deterministic\_result}")

# Randomized Algorithm Output

randomized\_result = find\_kth\_smallest(test\_array.copy(), k)

print(f"Randomized Algorithm: {k + 1}th smallest element is {randomized\_result}")

Output :

A screenshot of a computer program

Description automatically generated

Median of Medians (Deterministic):

Because of balanced partitioning, which guarantees that subproblems diminish linearly, the time complexity is O(n).

For in-place operations, the space complexity is O(1), with a recursive overhead of O(log⁡𝑛) O(logn).

Quickselect via Random Selection:

Time complexity: Average 𝑂 (𝑛) O(n), and in the worst scenario, 𝑂 (𝑛 2) O(n 2).

Space Complexity: recursive overhead, O(1) for in-place operations

O(logn) 𝑂 (log ⁡ 𝑛)

Comparison :

Code:   
import random

import time

import matplotlib.pyplot as plt

import numpy as np

# --- Median of Medians (Deterministic) ---

def median\_of\_medians(arr, k):

if len(arr) <= 5:

return sorted(arr)[k]

sublists = [arr[i:i + 5] for i in range(0, len(arr), 5)]

medians = [sorted(sublist)[len(sublist) // 2] for sublist in sublists]

pivot = median\_of\_medians(medians, len(medians) // 2)

low = [x for x in arr if x < pivot]

high = [x for x in arr if x > pivot]

pivot\_rank = len(low)

if k < pivot\_rank:

return median\_of\_medians(low, k)

elif k > pivot\_rank:

return median\_of\_medians(high, k - pivot\_rank - 1)

else:

return pivot

# --- Randomized Quickselect ---

def randomized\_partition(arr, low, high):

pivot\_index = random.randint(low, high)

arr[pivot\_index], arr[high] = arr[high], arr[pivot\_index]

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] <= pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

def randomized\_quickselect(arr, low, high, k):

if low == high:

return arr[low]

pivot\_index = randomized\_partition(arr, low, high)

if pivot\_index == k:

return arr[pivot\_index]

elif k < pivot\_index:

return randomized\_quickselect(arr, low, pivot\_index - 1, k)

else:

return randomized\_quickselect(arr, pivot\_index + 1, high, k)

def find\_kth\_smallest\_random(arr, k):

return randomized\_quickselect(arr, 0, len(arr) - 1, k)

# --- Empirical Comparison ---

def compare\_algorithms():

input\_sizes = [100, 500, 1000, 5000, 10000]

times\_mom = []

times\_rqs = []

for size in input\_sizes:

# Generate a random array for each test case

arr\_random = random.sample(range(1, 100000), size)

k = random.randint(0, size - 1)

# Measure time for Median of Medians

start\_time = time.time()

median\_of\_medians(arr\_random.copy(), k)

times\_mom.append(time.time() - start\_time)

# Measure time for Randomized Quickselect

start\_time = time.time()

find\_kth\_smallest\_random(arr\_random.copy(), k)

times\_rqs.append(time.time() - start\_time)

# Plot the results

plt.figure(figsize=(10, 6))

plt.plot(input\_sizes, times\_mom, label="Median of Medians", marker='o')

plt.plot(input\_sizes, times\_rqs, label="Randomized Quickselect", marker='x')

plt.xlabel("Input Size (n)")

plt.ylabel("Time (seconds)")

plt.title("Comparison of Median of Medians vs Randomized Quickselect")

plt.legend()

plt.grid(True)

plt.show()

# --- Running the comparison ---

compare\_algorithms()

Output :

A screen shot of a computer program

Description automatically generated

A graph with a line

Description automatically generated

Explanation:

Functions:

* median\_of\_medians(arr, k): The deterministic Median of Medians technique is implemented to determine the lowest element by k.
* To get the kth smallest element, the Randomised Quickselect algorithm is implemented as randomized\_quickselect(arr, low, high, k).
* algorithms\_compare(): a function to evaluate how well the two algorithms perform with varying input sizes. In addition to running both algorithms and recording their runtimes, it creates random input arrays.

Comparing empirically:

* We use input sizes of 100, 500, 1000, 5000, and 10,000 elements to test the algorithms.
* We choose a k at random for each input size and calculate how long each algorithm takes.
* The runtime for each algorithm is stored in the times\_mom and times\_rqs lists, respectively.
* Lastly, we use matplotlib to plot the runtimes.

Visualization

Each algorithm's runtime is displayed versus the size of the input. This enables us to compare the performance of each method graphically as the size of the inputs increases.

Key Points:

* In the worst scenario, the deterministic median of medians ensures O(n) time.
* The average time for Randomised Quickselect is O(n), however in extreme instances, it may deteriorate to O(n 2).
* Because of its additional overhead, the comparison will probably reveal that Median of Medians performs worse on average, but in the worst scenario, its time complexity is assured.
* Randomised Quickselect should perform quicker on average, but because of its worst-case 𝑂(𝑛2) O(n 2) behaviour, it may see runtime spikes for bigger inputs.

**Part 2: Elementary Data Structures Implementation and Discussion**

Matrices and Arrays

Basic data structures called arrays are used to hold a group of items in a single, continuous block of memory. Simply put, matrices are 2D arrays with rows and columns storing the elements.

Stacks

A Last In, First Out (LIFO) data structure is called a stack. Only the elements at the top of the stack can be added or removed.

Queues

First In, First Out (FIFO) data structures are what a queue is. Elements are taken out of the front and added to the back.

Linked Lists

Each element in a linked list points to the one after it in a linear arrangement. A singly linked list containing fundamental operations will be implemented.

Rooted Trees

A tree that has a specified root node is said to be rooted. Nodes are connected by edges, and each node may have more than one child.

Code :

# --- Array Implementation ---

class Array:

def \_\_init\_\_(self):

self.data = []

def insert(self, value):

self.data.append(value)

def delete(self, index):

if 0 <= index < len(self.data):

del self.data[index]

def access(self, index):

if 0 <= index < len(self.data):

return self.data[index]

return None

# --- Stack Implementation ---

class Stack:

def \_\_init\_\_(self):

self.data = []

def push(self, value):

self.data.append(value)

def pop(self):

if not self.is\_empty():

return self.data.pop()

return None

def peek(self):

if not self.is\_empty():

return self.data[-1]

return None

def is\_empty(self):

return len(self.data) == 0

# --- Queue Implementation ---

class Queue:

def \_\_init\_\_(self):

self.data = []

def enqueue(self, value):

self.data.append(value)

def dequeue(self):

if not self.is\_empty():

return self.data.pop(0)

return None

def peek(self):

if not self.is\_empty():

return self.data[0]

return None

def is\_empty(self):

return len(self.data) == 0

# --- Singly Linked List Implementation ---

class Node:

def \_\_init\_\_(self, data=None):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def insert(self, value):

new\_node = Node(value)

if self.head is None:

self.head = new\_node

else:

current = self.head

while current.next:

current = current.next

current.next = new\_node

def delete(self, value):

current = self.head

if current and current.data == value:

self.head = current.next

return

while current and current.next:

if current.next.data == value:

current.next = current.next.next

return

current = current.next

def traverse(self):

elements = []

current = self.head

while current:

elements.append(current.data)

current = current.next

return elements

# --- Rooted Tree Implementation ---

class TreeNode:

def \_\_init\_\_(self, data=None):

self.data = data

self.children = []

def add\_child(self, child\_node):

self.children.append(child\_node)

def remove\_child(self, child\_node):

self.children.remove(child\_node)

# --- Example Usage and Output ---

# Array Example

array = Array()

array.insert(10)

array.insert(20)

array.insert(30)

array.delete(1)

print("Array after insertions and deletion:", array.data)

# Stack Example

stack = Stack()

stack.push(10)

stack.push(20)

stack.push(30)

print("Stack after pushes:", stack.data)

stack.pop()

print("Stack after pop:", stack.data)

print("Stack peek:", stack.peek())

# Queue Example

queue = Queue()

queue.enqueue(10)

queue.enqueue(20)

queue.enqueue(30)

print("Queue after enqueue:", queue.data)

queue.dequeue()

print("Queue after dequeue:", queue.data)

print("Queue peek:", queue.peek())

# Linked List Example

linked\_list = LinkedList()

linked\_list.insert(10)

linked\_list.insert(20)

linked\_list.insert(30)

print("Linked List after insertions:", linked\_list.traverse())

linked\_list.delete(20)

print("Linked List after deletion:", linked\_list.traverse())

# Rooted Tree Example

root = TreeNode(1)

child1 = TreeNode(2)

child2 = TreeNode(3)

root.add\_child(child1)

root.add\_child(child2)

print("Rooted Tree structure:")

print("Root:", root.data)

for child in root.children:

print("Child node:", child.data)

Output :

A screen shot of a computer program

Description automatically generated

A computer screen with text on it

Description automatically generated

**Performance Analysis**

Arrays:

* O(1) for appending at the end and O(n) for inserting in the middle (because of shifting elements) are the two methods for insertion.
* Deletion at any arbitrary location: 𝑂(𝑛) O(n) (shifting required).
* For direct access to an element by index, use O(1).

Stacks:

* Pushing an element to the top is done with 𝑂(1) O(1).
* The removal of an element from the top is represented by Pop: O(1).
* Examine: 𝑂 (1) O(1) (looking at the top element).
* Space Complexity: 𝑂(𝑛) O(n), where n is the number of stack items.

Queues :

* Adding an element to the back of the queue is done by putting it in O(1).
* Dequeue: 𝑂 (𝑛) O(n) (when using an array, shifting causes an element to be removed from the front).
* Viewing the front element, peek: 𝑂( 1) O(1).
* The complexity of space is O(n)(O(n).

Linked Lists:

* Insertion: 𝑂(1) O(1) if the head or tail is being inserted, 𝑂(n) if the position is to be arbitrary.
* Deletion: O(1) if the head is being deleted, O(n) if a single node is being deleted.
* Traversal: O(n) 𝑂(𝑛).
* Because each node needs more memory for the subsequent pointer, the space complexity is O(n).

Rooted Trees :

* Insertion is O(1) (inserting a child to the root), but for a deep tree, it is typically O(n).
* The traversal is 𝑂(𝑛) O(n) (going through each node in the tree).
* The complexity of space is O(n)(O(n).

**Discussion of Practical Applications**

Arrays:  
It works best when you know how many elements you'll need in advance and you require random access to them. Perfect for situations when you regularly need to retrieve elements by index, including when storing fixed-size data, sorting algorithms, or searching.

Stacks:  
used for backtracking algorithms, expression evaluation, depth-first search (DFS) in trees and graphs, and undo systems in applications.

Queues:  
Perfect for managing requests in systems like as print queues or server request handling, job scheduling, and breadth-first search (BFS).

Linked Lists:  
Helpful when you anticipate a large number of insertions and deletions or when you are unsure of the collection's size beforehand. When constructing intricate data structures like hash tables or adjacency lists in graphs, for example, they enable dynamic memory allocation.

Rooted Trees:

Utilized in databases (B-trees for indexing), file systems, and scheduling job organization, among other hierarchical data representations.

Depending on the use case, each data structure has advantages and disadvantages. In hierarchical representations, rooted trees are utilized, linked lists are appropriate for dynamic data, stacks and queues are ideal for LIFO and FIFO operations, and arrays are excellent for fast access. Selecting the best data structure for the task at hand is made easier by having a thorough understanding of time complexities and practical applications.

REFERENCE

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.

This is one of the most authoritative texts on algorithms. The Median of Medians algorithm is introduced in the context of selection algorithms.

Relevant Chapter: "Selection" (Chapter 9).

Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L., & Tarjan, R. E. (1973). "Time Bounds for Selection." Journal of Computer and System Sciences, 7(4), 448-461.

The original paper that introduces the deterministic selection algorithm, Median of Medians. It proves the worst-case linear time complexity for the selection problem.

DOI: 10.1016/S0022-0000(73)80012-6

GeeksforGeeks. (n.d.). Median of Medians Algorithm to Find the Kth Smallest Element.

This article gives a clear, step-by-step breakdown of the Median of Medians algorithm and its implementation in Python.

Link

Wikipedia. (n.d.). Median of Medians.

An accessible overview of the Median of Medians algorithm, explaining its approach and time complexity.