Algorithms and Data Structures

Assignment 6: Median and Order Statistics & Element Data Structures

Nguyen The Duy Khanh

005019181

Table of Contents

1. Implementation and Analysis of the Selection Program

- Find kth smallest element in an unsorted array is a fundamental problem of computer science, and it has been applied for many wide-spread applications, particularly in data analysis and statistics, database query optimizations, etc. (e.g. finding percentiles of quartiles in the dataset [1], database indexing (2)). Before understanding the reasons why this problem has been so popular for centuries and widespread, we must understand two algorithms: the deterministic algorithm (e.g. Median of Median) and randomized implementation (e.g Randomized Quick Select)

a. Practical Implementation

- The Median of Medians[3] has three steps
    - **Step 1:** Choose a pivot (in this case, using the median of 5 by diving the arrays into five subarrays, finding the median of each subarray and then the median of the median)

```
if len(array) <= 5:
    return sorted(array)[kth]

medians = []
# Divide into 5 groups
for currentIndex in range(0, len(array), 5):
    subarray = array[currentIndex:currentIndex + 5]
    medians.append(sorted(subarray)[len(subarray) // 2])

# https://medium.com/@amit.desai03/median-of-median-on-medium-5ed518f17307
pivot = medianOfMedians(medians, len(medians) // 2)
```

Figure 1: Median of Median

- **Step 2:** Partition the array, which includes splitting and reordering the array so that all elements smaller than the pivot are on the left and all elements larger than the pivot are on the right.

```
leftArray = [element for element in array if element < pivot]
rightArray = [element for element in array if element > pivot]
pivotCount = array.count(pivot)
    You, 4 hours ago • Change easier implementation for find smal
```

Figure 2: Partition the array with the Median of Median Pivot

- **Step 3:** After partitioning the array, we will determine the kth smallest element by the following rules:
    - If the kth smallest element is in the left array, apply the previous steps to the left array.
    - If the kth smallest element is within the range of the pivot (if the pivot has duplicated values), then the kth should be the pivot
    - If the kth small element is larger than the pivot range, then the kth should be in the right array.

```
if kth < len(leftArray):
  return randomizedQuickSelect(leftArray, kth)
elif kth < len(leftArray) + pivotCount:
  return pivot

return randomizedQuickSelect(rightArray, kth - len(leftArray) - pivotCount)
```

Figure 3: Find the kth smallest element with left side, right side's partition, and pivot range

- As a result of finding the kth smallest element with a median of median is:

```
MSCS532_Assignment6 on  main via  v3.12.5 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ python3 main.py
Before finding the kth smallest element
71 122 160 186 92 88 149 130 181 175

The smallest 3 element with quickselect is 92
The smallest 6 element with quickselect is 149
The smallest 3 element with median of medians is 92
The smallest 6 element with median of medians is 149
Sorting array for easier visiblity for finding the kth smallest element
71 88 92 122 130 149 160 175 181 186
```

Figure 4: Find kth smallest element with a median of the median's output

- The same can be said for the Randomized QuickSelect, which has three steps
    - **Step 1:** Choose a pivot (in this case, randomized pivot selection)

```
pivot = random.choice(array)
```

Figure 4: Randomized Pivot Selection

    - **Step 2:** Partition the array, which includes splitting and reordering the array so that all elements smaller than the pivot are on the left and all elements larger than the pivot are on the right.

```
leftArray = [element for element in array if element < pivot]
rightArray = [element for element in array if element > pivot]
pivotCount = array.count(pivot)
    You, 4 hours ago • Change easier implementation for find smal
```

Figure 4: Partition the array with the Randomized Selection's Pivot

    - **Step 3:** After partitioning the array, we will determine the kth smallest element by the following rules:
        - If the kth smallest element is in the left array, apply the previous steps to the left array.
        - If the kth smallest element is within the range of the pivot (if the pivot has duplicated values), then the kth should be the pivot
        - If the kth small element is larger than the pivot range, then the kth should be in the right array.

```
if kth < len(leftArray):
  return randomizedQuickSelect(leftArray, kth)
elif kth < len(leftArray) + pivotCount:
  return pivot

return randomizedQuickSelect(rightArray, kth - len(leftArray) - pivotCount)
```

Figure 5: Find the kth smallest element with left side, right side's partition, and pivot range

- As a result of finding the kth smallest element with the randomized quickselect is:

```
MSCS532_Assignment6 on  main via  v3.12.5 on  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ python3 main.py
Before finding the kth smallest element
71 122 160 186 92 88 149 130 181 175

The smallest 3 element with quickselect is 92
The smallest 6 element with quickselect is 149
The smallest 3 element with median of medians is 92
The smallest 6 element with median of medians is 149
Sorting array for easier visiblity for finding the kth smallest element
71 88 92 122 130 149 160 175 181 186
```

Figure 6: Find kth smallest element with a median of median's output

b. Theoretical analysis

- The condition for the worst-case scenario is when the pivot is selected as the smallest or largest element, which leads to an unbalanced partition, needs to happen. However, due to the nature of choosing the median of the entire array, the pivot will never be the absolute worst and lead to a more balanced partitioned array. The crucial point is that in each recursive step, the algorithm reduces the size of the problem by at least $\frac{3}{10}$ of the original size (since at least $\frac{3}{10}$ of the elements is guaranteed to be in one of the partitions). Therefore, it is more guaranteed to be time complexity O(n) for the worst case and O(n) for space complexity when partitioning.

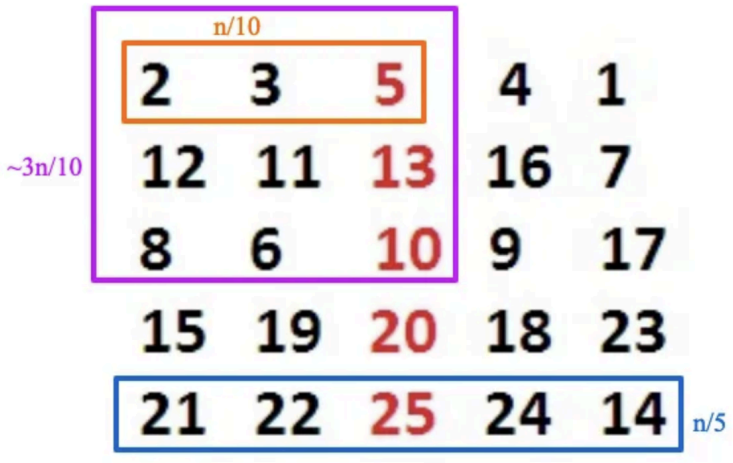

Figure 7: A guarantee of reducing $\frac{3}{10}$ elements when partitioning

- The recurrence for the Median of Medians is:

$$T(n) = T(\frac{3}{10}n) + O(n)$$

- For randomized quick selection, the algorithm chooses a pivot at random for each iteration. This randomness leads to average-case performance being much better than worst-case. On average, the random pivot will divide the array roughly in half. However, in the worst case, when the pivot is chosen consistently poor (which can be the largest element or smallest element), the partitioned array won't be balanced.
- Therefore, the expected time complexity of randomized quick selection is O(n), but the worst case can lead to $O(n^2)$, and O(n) for space complexity when partitioning.
- The recurrence for the Median of Medians is:

$$T(n) = T(\frac{n}{2}) + O(n)$$

c. Comparison

- When experimenting with Median of Median and Randomized Quick Select with different datasets, the difference between them can be clearly found:
  - With **random data** from n = 100 to 100,000 elements

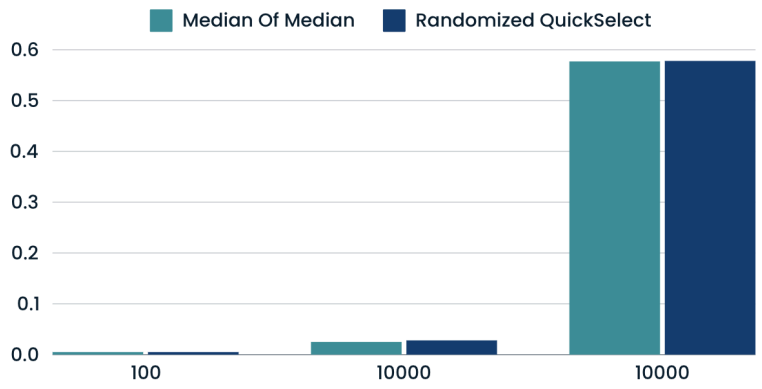

Figure 8: Random Dataset's Profiler with random data(CPU(s)))

  - With **sorted order data** from n = 100 to 100,000 elements

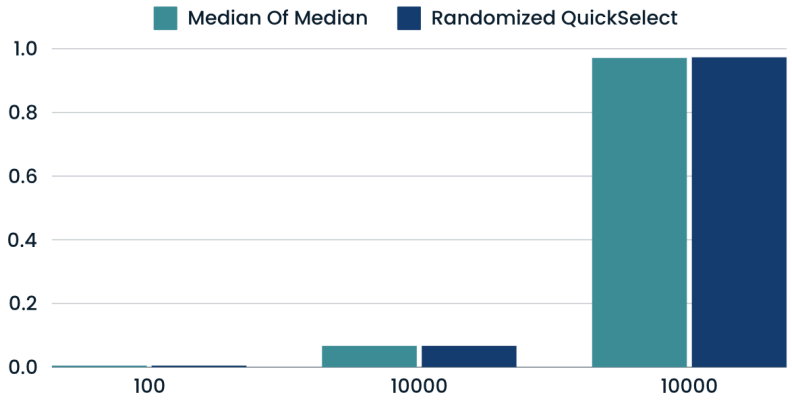

Figure 9: Random Dataset's Profiler with sorted order data (CPU(s)))

- With **reverse sorted order data** from n = 100 to 100,000 elements

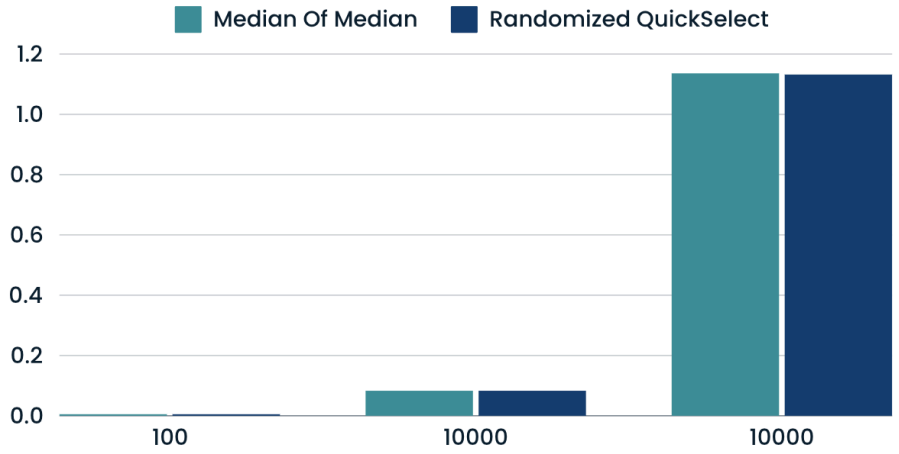

Figure 10: Random Dataset's Profiler with reverse sorted order data (CPU(s)))

- As a result of the experiment, the Median of Medians and Randomized Quick Select with Randomized Pivot Selection have the same performance time for all scenarios. Since it is rare to occur for the condition of choosing a pivot is consistently poor, the Randomized Quick Select will occur O(n) all the time. Even though there is a small increase with different datasets for both of the algorithms (e.g., 0.578s for random

data to 1.132s for sorted or reverse-sorted data), both algorithms perform respectively well, in contradiction to the algorithm for choosing the last element as pivot.

2. Element data structures implementation

a. Practical Implementation

- Array has the following operations:
    - **Insertion:** insert an element to the array with a time complexity of O(1) for the average case and worst case of O(n) for shifting elements when expanding the array size [4]

```python
# Insert an element to the array
# Time Complexity: O(n) (when shifting the elements to new array)
# https://stackoverflow.com/a/7770654
def insert(self, value: int) -> None:
  self.array.append(value)
```

Figure 11: Array's insertion in Python

    - **Remove:** insert an element to the array with a time complexity of O(1) for the average case and worst case of O(n) for shifting down elements when reducing the array size [4]

```python
# Remove an element from an array using index
# Time complexity: O(n) (when reducing the size and s
def remove(self, index: int) -> None:
  if index < 0 or index > len(self.array) - 1:
    print("Invalid index ", index)
    return
  del self.array[index]
```

Figure 12: Array's remove in Python

- **Access:** access an element in the array with a time complexity of O(1)

```python
# Access will access an element in the array
# Time Complexity: O(1)
def access(self, index: int) -> int:
    if index < 0 or index > len(self.array) - 1:
        print("Invalid index ", index)
        return -1

    return self.array[index]
```

Figure 12: Array's access in Python

- Matrice has the following operations:
    - **Insert:** will insert an array of values at a row level or column level with a time complexity of O(1) at row level and O(n m) at column level where n is the number of rows in the matrice and m is the number of columns in the matrice)

```python
# insertRow will insert an array of integer values to
# matrics at the target row
# Time Complexity: O(1) (O(n) when shifting the elements to new array)
# https://stackoverflow.com/a/7770654
def insertRow(self, row: int, values: list[int]) -> None:
    self.matrices.insert(row, values)

# insertColumn will insert all the values into target column
# Time Complexity: O(n * m)      You, yesterday • Add matrices operation
def insertColumn(self, column:int, values: list[int]) -> None:
    for index in range(len(self.matrices)):
        self.matrices[index].insert(column, values[index])
```

Figure 13: Matrice's insert in Python

- **Remove:** will remove an array of values at a row level or column level with a time complexity of O(1) at row level and O(n m) at column level where n is the number of rows in the matrice and m is the number of columns in the matrice)

```python
# Remove the row of matrics
# Time complexity: O(n) (when reducing the size and shift the elements to new array)
def removeRow(self, row: int):
    del self.matrices[row]

# Remove the column of matrics
# Time complexity: O(n * m) (when reducing the size and shift the elements to new array)
def removeCol(self, column: int):
    for row in self.matrices:
        del row[column]
```

Figure 14: Matrice's remove in Python

- **Access:** access an element in the matrice with a time complexity of O(1)

```python
# Access will access an element in the matrices
# Time Complexity: O(1)
def access(self, row: int, column: int) -> int:
  if row < 0 or row > len(self.matrices) - 1 or column < 0 or column > len(self.matrices[row]) -1:
    print("Invalid index at row ", row, " and ", column)
    return -1

  return self.matrices[row][column]
```

Figure 15: Matrice's access in Python

- Stack has the following operations (which follow the LIFO - last in, first out operation):
  - **Append:** insert an element at the end of the array with a time complexity of O(1) for the average case and worst case of O(n) for shifting elements when expanding the array size [4]

```python
# Insert an element to the stack as FILO
# Time Complexity: O(n) (when shifting the elements to new stack)
# https://stackoverflow.com/a/7770654
def insert(self, value: int) -> None:
  self.stack.append(value)
```

Figure 16: Stack's append in Python

  - **Pop:** remove the element at the end of the stack with a time complexity of O(1) for the average case and worst case of O(n) for shifting down elements when reducing the array size [4]

```python
# pop will remove the last element in the stack
# Time complexity: O(n) (when reducing the size and shift the elements to new array)
def pop(self) -> int:
  if len(self.stack) == 0:
    print("Empty stack")
    return -1

  lastElement = self.stack[len(self.stack) - 1]
  del self.stack[len(self.stack) - 1]
  return lastElement
```

Figure 17: Stack's pop in Python

- **Top:** get the top element of the stack with a time complexity of O(1)

```python
# top will get the top element of the stack
# Time complexity: O(1)
def top(self) -> int:
    if len(self.stack) == 0:
        print("Empty stack")
        return
    return self.stack[len(self.stack) - 1]
```

Figure 18: Stack's top in Python

- Queue has the following operations:
    - **Enqueue:** insert an element at the end of the array with a time complexity of O(1) for the average case and worst case of O(n) for shifting elements when expanding the array size [4]

```python
# enqueue will add an element to the queue as FIFO
# Time Complexity: O(n) (when shifting the elements to new stack)
# https://stackoverflow.com/a/7770654
def enqueue(self, value: int) -> None:
    self.queue.append(value)
```

Figure 19: Queue's enqueue in Python

    - **Dequeue:** Remove the first element of queue with a time complexity of O(1) for the average case and worst case of O(n) for shifting down elements when reducing the array size [4]

```python
# dequeue will remove the first element in the array
# Time complexity: O(n) (when reducing the size and shift the elements to new array)
def dequeue(self) -> int:
    if len(self.queue) == 0:
        print("Empty stack")
        return -1

    firstElement = self.queue[0]
    del self.queue[0]
    return firstElement
```

Figure 20: Queue's dequeue in Python

- **Is Empty:** checking whether the queue is empty

```python
# isEmpty will determine if the queue is empty or not
def isEmpty(self) -> int:
  return len(self.queue) == 0
```

Figure 21: Queue's is empty in Python

- Singly Linked List has the following operations:
  - **Insertion:** insert an element to the linked list with a time complexity of O(n) when traversing to the end of the array (if maintaining the tail, it will be O(1))

```python
# insert will insert the element at the end of linkedlist
# Time complexity: O(n)
def insert(self, value:int) -> None:
  if self.head == None:
    self.head = Node(value = value)
    return

  current = self.head
  while current.next is not None:
    current = current.next

  current.next = Node(value = value)
```

Figure 22: Linked List's insert in Python

  - **Remove:** remove an element at a given index to the linked list with a time complexity of O(n)

```python
# remove will remove the element at an index
# Time complexity: O(n)
def remove(self, index: int) -> None:
  current = self.head
  previous = None
  if current is not None and index == 0:
    value = current.value
    self.head = current.next
    return value

  currentIndex = 0
  while current is not None and currentIndex != index:
    previous = current
    current = current.next
    currentIndex +=1

  if current is None:
    return -1

  value = current.value
  previous.next = current.next
  current = None
  return value
```

Figure 23: Linked List's remove in Python

- **Search:** search the element at a given index with a time complexity of O(n)

```python
# search will search and return the element at an index
# Time complexity: O(n)
def search(self, index:int) -> int:
    current = self.head
    currentIndex = 0
    while current.next is not None:
        if currentIndex == index:
            break
        current = current.next
        currentIndex +=1

    if current == None:
        return -1

    return current.value
```

Figure 24: Linked List's search in Python

b. Performance Analysis

- When considering between LinkedList and Array [4], we need to know the following facts:
    - It's easier to store the data of different sizes in the linked list and scale the size organically, while an array can only store data at the same size, and its size needs to be known beforehand; otherwise, each insertion to the array can be costly (resizing the array leads to time complexity of O(n) [5]). However, LinkedList comes with a cost of extra memory overhead for storing data.
    - The array allows random access, while a linked list allows only sequential access to elements. Therefore, it is unsuitable for applications where it is helpful to look up an index quickly (e.g. quick sort, priority queue), and it will decrease the implementation's complexity.

- Therefore, when implementing Queue and Stack with Array or LinkedList, we would need to know the limitation of the size that those data structures will store and the use case of the designed application beforehand. In the memory-constrained environment, and if the application has a known size beforehand, an array is preferable with a bonus in ease of implementation (e.g, task scheduling system [6]). Additionally, nowadays, with the utilization of L1, L2, and L3 cache, the array is preferable to a linked list [7] when accessing a contiguous block of memory.

c. Discussions

- Each data structure has its use case since each has its advantages and disadvantages, and we need to consider the trade-off before implementing the application in a well-designed manner:
    - **Queue:** is often used for tasking scheduling systems such as a print queue system [8]. When documents are sent to a printer, they are placed in a queue. The printer processes the documents in the order they were received.
    - **Stack:** is often used in function call management [9] When a function is called, it's pushed onto the stack, and when it returns, it's popped off, maintaining the correct order of execution.
    - **Array:** Arrays are ideal for storing collections of data when the size is known and fixed, such as storing the days of the week or a list of student grades.
    - **Matrices:** is often used in graph representation. Adjacency matrices can represent graphs, where the matrix indicates whether pairs of vertices are connected by edges.

- **Linked List:** Linked lists are useful for applications that require frequent insertion and deletion of elements, such as implementing a dynamic list of students in a class where enrollments may change.

3. References

- [1] *Pavan Vadapalli* (2024). Kth Smallest Element. UpGrad. https://www.upgrad.com/tutorials/software-engineering/software-key-tutorial/kth-smallest-element/
- [2] Oracle (2006). Managing Database indexes in sorting and searching https://docs.oracle.com/cd/B40099_02/books/PerformTun/PerformTunCustConfig10.html
- [3] *Amit Desai (2019). Median of Median on Medium. Medium https://medium.com/@amit.desai03/median-of-median-on-medium-5ed518f17307*
- [4] *Jonas Klemming (2008)* Array Versus Linked List. StackOverFlow. https://stackoverflow.com/a/166966
- [5] *Karoly Horvath* (2014). Time complexity of append operation in simple array. Stackoverflow. https://stackoverflow.com/questions/22173998/time-complexity-of-append-operation-in-simple-array#:~:text=In%20case%20of%20static%20list,memory%20for%20the%20append%20operation%20.
- [6] Douglas Gregory(2023) . What data structure will be most efficient for a task scheduler? Game Development. https://gamedev.stackexchange.com/questions/208359/what-data-structure-would-be-the-most-efficient-for-a-task-scheduler

- [7] Reddit [2023]. Why aren’t linked lists more popular? https://www.reddit.com/r/learnprogramming/comments/10hwrmy/why_arent_linked_lists_more_popular/
- [8] Microsoft. View printer queue In Windows. https://support.microsoft.com/en-us/windows/view-printer-queue-in-windows-71505b3a-ba6b-14b2-b7f9-fd6204675ab5
- [9] Puneeth (2024). Functional Call Stack in C. GeeksForGeeks. https://www.geeksforgeeks.org/function-call-stack-in-c/#