Analysis of Algorithms

BLG 335E

Project 2 Report

Şafak Özkan Pala
palas21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 14.12.2023

1. Implementation

1.1. Headers

In this assignment five header files are used for specific purposes.

- **<iostream>** : It is included for handling input and output operations.
- **<vector>** : It is included for the storage of the data.
- **<string>** : It is included for string operations.
- **<fstream>** : It is included for file operations.
- **<chrono>** : It is included for time-related operations.

1.2. Struct

struct City is used for storing the name and the population of the city.

1.3. Functions

1.3.1. Custom Functions:

- **takeDataFromCsvFile** : This function takes vector and filename as arguments and for every line of the file it seperates elements than assign them to proper part of the struct after that push back the struct into the vector.
- **customSwap** : This function takes references of the two cities and swaps them using temporary variable.
- **printCities** : This function takes vector and iterates through it while printing the data into the terminal.
- **writeToAFile** : This function takes vector and iterates through it while printing the data into the file.

1.3.2. Heap Functions:

- **max_heapify** : This function takes vector, index and size of the vector and with the given index finds the unproper element and place it into the correct place with compare it with each child. If it is not fit than swaps with corresponding child and this process progress until the element is in the correct place.
- **build_max_heap** : This function takes vector and vector size than iterates through every non-leaf element and max_heapify them.

- **heapsort** : This function takes heapified vector and using heapsort algorithm it takes the biggest element which is the first and starting with the end of the vector it swaps it with that element. After that it max heapifies the vector and continue this process until the vector is sorted.
  This functions time complexity is $O(n\log(n))$ since it iterates through the half of the input size whcih has a complexity of $O(n)$ and every time it calls the max_heapify function which takes $O(\log n)$ time we can obtain $O(n\log(n))$ by multiplying the these two complexities.

- **max_heap_insert** : This function takes heapified vector, a new city element and the size of the vector and add the new city at the end of the vector. After that it calls the heap_increase_key function.

- **heap_extract_max** : This function takes the heapified vector and its size than takes the first element of the vector and stores it into the temporary element after that swaps it with the last element than decrease the size of the vector and calls the max_heapify function than return the temporary element.

- **heap_increase_key** : This function takes heapified vector, index of the changing element, key value for the changing element and the size of the vector. If the new value is smaller than the current value we print an error message and return without doing anyting. If it is not we change the population of the index element and check wheter it is bigger than its parent. If it is than change it with the parent and change the index to parents. After that we continue this prosess until the parent is bigger than changed element.

- **heap_maximum** : This function takes heapified vector and its size and return the first element which is the maxiumum.

1.3.2.1. D-ary Heap Functions:

- **dary_max_heapify** : This function takes vector, index, d which is the number of children and size of the vector and with the given index finds the unproper element and place it into the correct place with compare it with each child. If it is not fit than swaps with corresponding child and this process progress until the element is in the correct place.

- **dary_build_max_heap** : This function takes vector, its size and d than iterates through every non-leaf element and dary_max_heapify them.

- **dary_calculate_height** : This function takes vector and iterates though its element taking the first child of every parent until the reach beyond the size of the vector than height is increased in every iteration than it is returned.

- **dary_extract_max** : This function takes vector, its size and the child number of every parent, if heap is empty error message will be printed and return if it is not the max element which is the first element of the vector stored into a temporary variable than size of the vector is decreased and the dary_max_heapify function is called to make the vector heap again and temporary element is returned. Except dary_build_max_heap function every line of the function runs in $O(n)$ time. But dary_build_max_heap function runs in $O(\log_d n)$ time so it is also this functions time complexity

- **dary_insert_element** : This function takes vector, its size, d and new element. It pushes the new element into the vector and calls the function dary_increase_key. In the function every line is takes $O(1)$ time except dary_increase_key function. Its complexity is $O(n \log_d(n))$. Therefore it is also this functions complexity

- **dary_increase_key** : This function takes vector, its size, index of changing element, key value and d. It controls the size of the heap if heap is empty it prints an error message and return if it is not it assign new key value to the element in the given index. Than calculate the height of the heap for iteration. Until reaching the height or parents population is bigger function iterates through the heap by changing the parent at each time and decreasing the height. In the function every operation takes $O(1)$ except while loop. In total it takes $O(1)$ time but for every iteration which is d it makes this process so total time complexity of this function is $O(n \log_d(n))$.

- **dary_heapsort** : This function takes heapified vector and using heapsort algorithm it takes the biggest element which is the first and starting with the end of the vector it swaps it with that element. After that it max heapifies the vector and continue this process until the vector is sorted.
  This functions time complexity is $O(n \log_d(n))$ since it iterates through the half of the input size whcih has a complexity of $O(n)$ and every time it calls the max_heapify function which takes $O(\log_d(n))$ time we can obtain $O(n \log_d(n))$ by multiplying the these two complexities.

2. Heapsort vs Quicksort

2.1. Time Difference

Total time taken functions are given below in the tables.

QuickSort	Population1	Population2	Population3	Population4
Last Element	89293740 ns	1443036117 ns	602097537 ns	4926545 ns
Random Element	4194948 ns	2855417 ns	3872431 ns	4981370 ns
Median of 3	4046170 ns	3259709 ns	3807761 ns	4700102 ns
Heapsort	9678662 ns	8211671 ns	7996395 ns	8818207 ns

Both Quicksort and Heapsort generally have $O(n\log(n))$. average and best-case time complexities, Quicksort can outperform Heapsort in practice due to its smaller constant factors. However, Heapsort's worst-case time complexity might be better in situations where worst-case performance is critical. By randomizing the data Quicksort's worst-case time complexity disadventegous can be eleminated and it can outperforms the Heapsort.

2.2. Comparison Difference

Total comparison numbers are given below in the tables.

QuickSort	Population1	Population2	Population3	Population4
Last Element	9004989	95309721	72827050	234018
Random Element	225260	219191	221441	214263
Median of 3	198198	196976	206015	204321
Heapsort	172558	194077	174489	184682

According to results these assumptions can be make. (Due to my poor implemtation of QuickSort it can be misleading so i am going to talk about proper implementations)

QuickSort:

- **Average Case Comparisons:** In the average case, QuickSort makes approximately $O(n\log n)$ comparisons.The algorithm's efficiency be better in scenarios with random data.
- **Best Case Comparisons:** In the best case, QuickSort also makes $O(n\log n)$ comparisons.
- **Worst Case Comparisons:** The worst-case scenario, where QuickSort can make $O(n^2)$ comparisons. This occurs when poorly chosen pivots lead to unbalanced partitions. However, good pivot selection strategies can eleminate this scenario.

HeapSort:

- **Consistent Comparisons:** HeapSort, in its typical implementation, consistently makes $O(n \log n)$ comparisons in all cases. Data does not affect its comparison number

In practice, QuickSort often shows better average-case performance. The ability of QuickSort to adapt to different input distributions contributes to its efficiency. On the other hand, HeapSort's consistent $O(n \log n)$ comparison count makes it more predictable, especially when worst-case behavior is on the table.

2.3. Strength and Weaknesses

Strengths and Weaknesses of QuickSort:

Strengths:

- **Efficiency in Practice:** QuickSort is often faster in practice Its average-case performance is generally superior to many other sorting algorithms.
- **Adaptability:** QuickSort adapts well to different input distributions. It efficiently handles a wide range of scenarios, making it versatile for various data if the data is randomized. If it is not than there might be a big problem for this algorithm.
- **In-Place Sorting:** QuickSort can be implemented as an in-place sorting algorithm, which means it doesn't require additional memory proportional to the input size.

Weaknesses:

- **Worst-Case Complexity:** The worst-case time complexity of QuickSort is $O(n^2)$, which occurs when chosen pivots lead to unbalanced partitions. It can be a concern for specific data.
- **Stability:** By default, QuickSort is not stable. This can be a drawback in scenarios where stability is crucial.
- **Dependency on Pivot Selection:** The efficiency of QuickSort is highly dependent on the pivot selection strategy. Poor pivot choices can decrease performance.

Strengths and Weaknesses of HeapSort:

Strengths:

- **Consistent Performance:** HeapSort consistently performs with $O(n \log n)$ time complexity in all cases Its predictability makes it suitable for scenarios where worst-case behavior is a significant concern.
- **Worst-Case Efficiency:** HeapSort guarantees $O(n \log n)$ time complexity in the worst case, making it more predictable than QuickSort.

- **No Dependency on Input Distribution:** HeapSort's performance is not affected by the input distribution, making it a reliable choice for various data.

Weaknesses:

- **Slower in Practice:** HeapSort may be slower in practice compared to QuickSort, especially for small data.
- **Not In-Place:** HeapSort typically requires additional memory for the heap data structure, making it less memory-efficient than QuickSort.
- **Complex Implementation:** The implementation of HeapSort involves heap construction and manipulation, which may be more complex for programmer compared to QuickSort.

In summary, the choice between QuickSort and HeapSort depends on the specific requirements and data of the sorting task. QuickSort is often preferred for its efficiency in practice, while HeapSort's consistent performance and worst-case guarantees make it a reliable choice in critical applications.