Analysis of Algorithms

BLG 335E

Project 2 Report

MUSTAFA CAN ÇALIŞKAN
caliskanmu20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 21.11.2023

1. Implementation

1.1. General Code Structure

1.1.1. main Function

In the main function, the necessary arguments are first received. To enable proper comparison based on the names of functions (if argv[2] == heapsort), the char** data type is converted to string* data type. Then, variables i, d, and k are initialized (with d defaulting to 2). Subsequently, within a loop, the arguments are traversed and replaced with the initialized values (assuming that i, d, and k values are provided correctly for the given function). Finally, the operation specified in the last argument is executed, and the elapsed time is printed to the console.

1.1.2. max_heapify, build_max_heapify and heapsort Functions

The functions max_heapify, build_max_heapify, and heapsort were written generically based on the parameter ’d’. For instance, if only max_heapify is called, ’d’ will be set to 2. When calling one of the d_ary functions, it will operate according to the value of ’d’ specific to that function. max_heapify function compares elements in a d-ary heap structure, ensuring that a parent is larger than its children by swapping elements when necessary, thereby maintaining the maximum element at the top. The build_max_heap function constructs a maximum heap structure from an array, and heapsort uses this maximum heap to continuously move the maximum element to the end while adjusting the heap size accordingly, thereby sorting the array.

1.1.3. Priority Queue Functions

Before performing priority queue operations, the array is initially built as a max heap. Subsequently, if there’s a change in the heap, heapify operation is performed again. It was assumed that the given index in the ’heap_increase_key’ function is the index after the array has been built.

Priority queues constructed with a max heap structure can be exemplified in real-life scenarios such as an emergency priority queue determining the order of patient treatments. This structure utilizes a max heap to prioritize patients based on their conditions, ages, and urgency levels. Similarly, for airplane ticket reservations, a max heap can be employed. Airlines manage vacant seats by organizing a priority queue based on factors like the date of pre-purchased tickets, reservation time, or ticket price.

1.1.4. d-ary Functions

Similar to the priority queue functions, the array is first built as a max heap. After the operation is completed, if there's a change in the heap, it undergoes heapify again. It was assumed that the given index in the 'dary_increase_key' function is the index after the array has been built.

1.1.5. Utility Functions

Various utility functions have been written for printing the array to a given CSV file, reading from a CSV file, searching for a specific character within a given string, and swapping the positions of two pairs.

1.2. Complexity Analysis

1.2.1. max_heapify

Best case: $O(d)$
In the best-case scenario, each node's children can have smaller values than the node itself, and there might not be a need for any swap operations.

Worst case: $O(d * log_d(n))$
In the worst-case scenario, each node's children can have larger values than the node itself, leading to a swap operation at every level where each node checks d children and performs a swap.

1.2.2. build_max_heap

Best case: $O(n)$
In the best-case scenario, when max_heapify is called for each node without requiring any swap operation, the complexity for max_heapify function can indeed be just $O(d)$ for each node, as it inspects the child nodes underneath but performs no actual swapping. In this case, within the build_max_heap function, the loop runs n / d times, and during each iteration, the max_heapify function is called. Since the best-case complexity of each max_heapify call is $O(d)$, the total complexity, when multiplied by the number of operations throughout the loop, can be expressed as $O((n/d) * d)$.

Worst case: $O(n * log_d(n))$
In the worst-case scenario, during each max_heapify call in a node with d children, a swap operation might be needed with each child node, taking $log_d(n)$ steps. Thus, the complexity of each max_heapify call is $O(d * log_d(n))$. The build_max_heap function calls max_heapify (n / d) times. As each call's worst-case complexity is $O(d * log_d(n))$, the total complexity can be expressed as $O((n/d) * d * log\_d(n))$. Simplifying this

expression, it becomes $O(n * log_d(n))$.

1.2.3. heapsort

Best case: $O(n * log_d(n))$
In the best-case scenario, the function build_max_heap might have been called initially, but subsequent max_heapify operations may not require many swaps. The time complexity of build_max_heap is $O(n)$, and in the best case, the time complexity of each iteration (after creating the max heap) is O(log n). The number of iterations is n - 1, resulting in a total time complexity of $O(n * log_d(n))$ in the best case.

Worst case: $O(n * log_d(n))$
In the worst scenario, when both build_max_heap and max_heapify functions are utilized, the time complexity for building the max heap is $O(n * log_d(n))$, and each iteration afterward (once the max heap is constructed) has a time complexity of $O(log_d(n))$. Given there are n - 1 iterations, the overall time complexity in the worst case remains $O(n * log_d(n))$.

1.2.4. Priority Queue Functions and d-ary Functions

Since the build_max_heap function comes before all other functions (except dary_calculate_height) and the operations within these functions have a complexity of $O(1)$ (inserting, deleting etc.), the functions have a best-case complexity of $O(n)$ and a worst-case complexity of $O(n * log_d(n))$. Since dary_calculate_height performs a single calculation, it has a complexity of $O(1)$ in both cases.

1.3. Time Measurements

functions	d = 3	d = 5	d = 7	d = 9
dary_extract_max	1695612	1256239	1401300	533030
dary_insert_element	2327748	1196874	606085	252127
dary_increase_key	1571923	1044619	822034	590439

Table 1.1: Time measurements (in nanoseconds) using population4.csv, k_city_650000, i950

As d increases, generally, the complexity time decreases in accordance with $O(n * log_d(n))$.

1.4. Comparative Analysis

1.4.1. Elapsed Time Comparison

	population1	population2	population3	population4
Naive Quicksort	350934397	5044147475	2654739558	10293744

Table 1.2: Time measurements (in nanoseconds) using population4.csv

	population1	population2	population3	population4
Heapsort	12342115	13487375	13249565	13793841

Table 1.3: Time measurements (in nanoseconds) using population4.csv

It can be seen from the data that heapsort, compared to quicksort (including different strategies), is more stable, indicating fewer changes based on the distributions in the given input file. This situation confirms with the confirmed time complexities of quicksort (best: $O(nlogn)$, worst: $O(n^2)$) and heapsort (best: $O(n * logn)$, worst: $O(n * logn)$). Additionally, it can be observed that heapsort outperforms quicksort in overall performance.

1.4.2. Number of Comparisons

Heapsort generally tends to perform more comparison operations compared to Quicksort. While Heapsort performs $n * log(n)$ comparisons in the worst-case scenario, Quicksort has the potential to perform n^2 comparisons in its worst-case scenario. Usually, Heapsort offers more predictable performance; however, Quicksort, while often faster in average cases, may lose effectiveness in worst-case scenarios.

1.4.3. Discussions

Heapsort, with its consistent $O(n * log_n)$ time complexity, excels in scenarios where a guaranteed worst-case performance is crucial, making it reliable for large datasets. Its drawback lies in being less cache-friendly. Quicksort may outperform Heapsort in average-case scenarios, especially with smaller datasets, where cache efficiency matters. Heapsort is suited for situations requiring a dependable worst-case performance, while Quicksort's strengths shine in average-case scenarios with considerations for cache efficiency.