Analysis of Algorithms

BLG 335E

Project 2 Report

Emil Huseynov
150210906
huseynove21@itu.edu.tr

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

1. Implementation

1.1. Introduction

This report presents an analysis of the provided C++ code implementing the heapsort algorithm. The implementation details are examined, followed by a comparative analysis of heapsort and quicksort algorithms.

1.2. Analysis of the Heapsort Implementation(Code Structure and Components)

2.1.1 Data Structure

The code defines a ’MyPair’ struct to hold pairs of strings and integers. This struct is used to store and sort data based on the integer value while maintaining an association with the corresponding string.

2.1.2 Swap Function

A custom ’myswap’ function is implemented to exchange two ’MyPair’ elements. This function is crucial for the reordering operations within the heapsort algorithm.

2.1.3 Heap Operations

Several functions implement the core operations of the heapsort algorithm:

-heapincreasekey: Increases the value of an element in the heap, maintaining the heap property.

-maxheapify: Ensures the max-heap property of a subtree, a fundamental operation for maintaining the heap.

-buildmaxheap: Converts an unsorted vector into a max-heap.

heapsort: Performs the heapsort algorithm, repeatedly extracting the maximum element to sort the array.

2.1.4 D-ary Heap Extensions

The code includes functions for working with a d-ary heap, a generalization of a binary heap:

-maxheapifydown and heapifyup: Ensure the max-heap property in a d-ary heap.

-darycalculateheight: Calculates the height of a d-ary heap.

-daryextractmax and daryinsertelement: Extract and insert elements in a d-ary max-heap.

2.1.5 Main Function

The main function handles input parsing, file reading, and executing the specified heap operation. It demonstrates the implementation's versatility in handling various heap operations.

1.3. Comparative Analysis of Heapsort and Quicksort

3.1 Efficiency
- Heapsort provides O(n log n) performance in worst-case scenarios, making it reliable for predictable execution time. Quicksort, while faster on average, suffers in its worst case with $O(n^2)$.

3.2 Use Cases
- Heapsort's consistent performance makes it suitable for applications where data may not be randomly distributed. Quicksort is preferred for its average-case efficiency in scenarios where worst-case performance is not a critical concern.

1.4. Conclusion

The provided heapsort implementation in C++ is a comprehensive example of the algorithm, showcasing various heap operations. In comparing heapsort with quicksort, each algorithm exhibits distinct strengths and trade-offs, informing their suitability for different applications.