Analysis of Algorithms

BLG 335E

Project 2 Report

Ekin Taşyürek
tasyurek19@itu.edu.tr

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

1. Implementation

1.1. Max Heapify

The `max_heapify` function takes a vector of `City` objects, which is assumed to represent a binary heap, and checks to see if the subtree rooted at a certain index still has the max-heap property. This is required for the heap sort algorithm, which extracts the largest member from the heap periodically to form a sorted array.

Time complexity: $O(logn)$

Example function call: `./Heapsort population1.csv max_heapify out.csv i50`

1.2. Build Max Heap

The `build_max_heap` function takes a vector of `City` objects and builds a max-heap from it. It begins with the last non-leaf node and iteratively applies `max_heapify` to verify that each subtree has the max-heap property. Iterating through the items from the last non-leaf node to the root, the function calls `max_heapify` at each step to produce a valid max-heap.

Time complexity: $O(n)$

Example function call: `./Heapsort population1.csv build_max_heap out.csv`

1.3. Heapsort

The `heapsort` function takes a vector of `City` objects and sorts it in ascending order using the heap sort algorithm. It first constructs a max-heap from the input array, then continually takes the maximum element from the heap, swapping it with the last unsorted element while keeping the max-heap attribute. The function calls `build_max_heap` to initialize the heap, and then iterates through the array, exchanging the maximum element with the last unsorted element and restoring the max-heap attribute.

Time complexity: $O(n \log n)$

Example function call: `./Heapsort population1.csv heapsort out.csv`

1.4. Max Heap Insert

The `max_heap_insert` function takes a vector of `City` objects and inserts a new city into the max-heap. The new city is appended to the end of the array, and the heap is adjusted by swapping the new element with its parent until the max-heap property is restored. The function appends the new city to the heap array, then changes the heap structure by swapping the new element with its parent until the max-heap attribute is restored.

Time complexity: $O(\log n)$

Example function call: `./Heapsort population1.csv max_heap_insert out.csv k_Kayseri_500000`

1.5. Heap Extract Max

The `heap_extract_max` method takes a vector of `City` objects, extracts the maximum city (found at the max-heap's root), and restores the max-heap attribute. It swaps the root with the last element, then removes the last element and applies `max_heapify` to keep the max-heap structure. The function extracts the maximum city (placed at the root), replaces it with the heap's final city, removes the heap's last city, and then applies `max_heapify` to keep the max-heap attribute.

Time complexity: $O(\log n)$

Example function call: `./Heapsort population1.csv heap_extract_max out.csv`

1.6. Heap Increase Key

The `heap_increase_key` function takes a vector of `City` objects, an index i, and a new key value. It then adjusts the max-heap structure by swapping the city with its parent until the max-heap property is restored by updating the key of the city at index i. The function modifies the heap structure by swapping the city with its parent until the max-heap property is restored by updating the population key of the city at index i.

Time complexity: $O(\log n)$

Example function call: `./Heapsort population1.csv heap_increase_key out.csv i30 k500000`

1.7. Heap Maximum

The `heap_maximum` function takes a vector of `City` objects, applies `max_heapify` to assure the max-heap property, and returns the city with the highest population (placed at the max-heap's root). The function uses `max_heapify` to ensure the max-heap property, then returns the city with the most people (placed at the root of the max-heap).

Time complexity: $O(n)$

Example function call: `./Heapsort population1.csv heap_maximum out.csv`

1.8. Build D-ary Heap

The `build_dary_heap` function constructs a D-ary max-heap from a vector of `City` objects. It initiates the process by starting from the last non-leaf node and iteratively applies `dary_max_heapify` to ensure that each subtree maintains the max-heap property in the context of a D-ary heap. The function iterates through the elements from the last non-leaf node to the root, invoking `dary_max_heapify` at each step to establish a valid D-ary max-heap.

Time complexity: $O(n)$

Example function call: `./Heapsort population1.csv build_dary_heap out.csv d5`

1.9. D-ary Heap Height Calculation

The `dary_calculate_height` function computes the height of a D-ary heap given its element count n and base d. It divides the number of elements by d recursively until n equals zero, incrementing the height at each step. The function divides the number of elements by d iteratively until n equals zero, incrementing the height at each step.

Time complexity: $O(\log_d n)$

Example function call: `./Heapsort population1.csv dary_calculate_height out.csv d5`

1.10. D-ary Heap Extract Max

The `dary_extract_max` function takes a vector of `City` objects representing a D-ary heap and returns the largest city (placed at the heap's root). If the heap is empty, it returns a `City` object by default. If the heap is not empty, the function extracts the maximum city, replaces it with the last city in the heap, and eliminates the final city.

Time complexity: $O(1)$

Example function call: `./Heapsort population1.csv dary_extract_max out.csv d5`

1.11. D-ary Heap Insert Element

The function `dary_insert_element` accepts a vector of `City` objects representing a D-ary heap, a new city (`newCity`), and the base (`d`). It adds the new city to the heap, altering the heap structure by comparing population values to their parent until the D-ary heap property is met.The function appends the new city to the heap array and changes the heap structure by comparing population values to their parent until the D-ary heap property is satisfied.

Time complexity: $O(\log_d n)$

Example function call: `./Heapsort population1.csv dary_insert_element out.csv d5 k_Kayseri_500000`

1.12. D-ary Heap Increase Key

The function `dary_increase_key` accepts a vector of `City` objects representing a D-ary heap, an index i, a new key value, and the base d. It updates the city's population key at index i and changes the heap structure by comparing population values to their parent until the D-ary heap property is satisfied. The function updates the city's population key at index i and modifies the heap structure by comparing population values with their parent until the D-ary heap property is satisfied.

Time complexity: $O(\log_d n)$

Example function call: `./Heapsort population1.csv dary_increase_key out.csv i10 d5 k500000`

1.13. HeapSort vs QuickSort

QuickSort is a divide-and-conquer sorting algorithm that works by splitting an array into two sub-arrays and sorting them recursively. It chooses a pivot element and positions it so that items less than the pivot are on its left and elements greater are on its right.

HeapSort is a binary heap data structure-based comparison sorting method. It creates a max-heap and extracts the maximum element repeatedly, placing it at the end of the array. After that, the heap property is restored, resulting in a sorted array.

	Population1	**Population2**	**Population3**	**Population4**
HeapSort	11531875 ns	12551209 ns	12364875 ns	12870000 ns
QS Last Element	144697000 ns	2134767667 ns	902100459 ns	21341417 ns
QS Random Element	26829750 ns	23395166 ns	18705875 ns	27872000 ns
QS Median of 3	26294250 ns	26282250 ns	14450000 ns	26925625 ns

Table 1.1: Comparison of different pivoting strategies on input data.

Regardless of the input distribution, HeapSort consistently displays competitive runtime efficiency across different datasets. The elapsed time for QuickSort Last Element is much longer. This implies that utilizing the final element as the pivot could result in poor partitioning for some input distributions. QuickSort Random Element and QuickSort Median of 3 outperform QuickSort Last Element. The randomized and median-based partitioning algorithms provide more balanced partitions, which improves runtime performance.

- **QuickSort**

  Time Complexity Average Case: $O(n \log n)$

  Time Complexity Worst Case: $O(n^2)$

  Number of Comparisons Average Case: $O(n \log n)$

  Number of Comparisons Worst Case: $O(n^2)$

  QuickSort, on average, requires fewer comparisons than HeapSort, especially when using efficient pivot selection strategies.

- **HeapSort**

  Time Complexity Average/Worst Case: $O(n \log n)$

  Number of Comparisons Average/Worst Case: $O(n \log n)$, each element may be compared with its children during the heap construction and with its children and possibly with its parent during the heapify step.

In circumstances where a constant and predictable runtime is required, HeapSort may outperform QuickSort. HeapSort's $O(n \log n)$ worst-case guarantee, for example,

makes it a dependable choice for working with sensitive systems or real-time applications where worst-case performance is a problem.

QuickSort may be preferable in situations where average-case performance is more important than worst-case guarantees. QuickSort frequently outperforms other sorting algorithms on random or partially sorted data, making it useful for general-purpose sorting applications.

HeapSort delivers consistent performance but lacks adaptability, whereas QuickSort provides average-case efficiency while taking into account worst-case possibilities.