Analysis of Algorithms

BLG 335E

Project 2 Report

KAAN KARATAŞ
karatask20@itu.edu.tr

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

1. Implementation

1.1. HEAPSORT PROCEDURES

The project is an implementation of a heap data structure, specifically focusing on the heap sort algorithm. It is designed to read a dataset of city data from a CSV file, perform various operations on the data using a heap, and then write the results to an output file. The operations include sorting the data and performing various priority queue operations. The program allows optional command-line arguments to specify parameters such as the number of children for non-leaf nodes, index values, and key values for 'increase_key' functions. The program also measures the time taken to perform the sorting operation, providing a useful benchmark for the efficiency of the heap sort algorithm. This sorting time of the operations can be used for comparisons between different algorithms such as quicksort, as well.

1.1.1. MAX-HEAPIFY Procedure

The 'max_heapify' procedure plays a pivotal role in preserving the max-heap property within a binary heap, ensuring that the highest value resides at the root. This algorithm, designed for efficiency, takes as input a vector representing the heap and an index where the max-heap property may be violated. It employs a recursive approach, comparing the node in question with its left and right children. If a child's population surpasses that of the current node, a swap occurs, and the procedure recursively applies itself to the affected child. This ensures that any potential violations are addressed throughout the heap. Crucially, 'max_heapify' operates in $O(logn)$ time, where 'n' is the size of the heap, a key factor in maintaining the efficiency of heap-based algorithms like heapsort. Its strategic swapping mechanism and recursive nature contribute to the overall integrity of the max-heap structure, essential for optimizing various heap operations. Additionally, the space complexity of 'max_heapify' is $O(1)$

1.1.2. BUILD-MAX-HEAP Procedure

The 'build_max_heap' function is a short, but important step in creating a max heap from a given dataset. The function iterates through the non-leaf nodes in reverse order, calling 'max_heapify' on each. This ensures the max-heap property, where each node is greater than or equal to its children, is maintained. The efficiency of this algorithm is $O(n)$, where n is the dataset size, making it work in linear time. The pseudo code of the procedure is quite simple, and can be observed below. The space complexity of the function is $O(1)$

Algorithm 1 MAX-HEAPIFY

```
1: function MaxHeapify(dataset, size, index)
2:     largest ← index
3:     leftChild ← 2 × index + 1
4:     rightChild ← 2 × index + 2
5:     if leftChild < size and dataset[leftChild].population > dataset[largest].population then
6:         largest ← leftChild
7:     end if
8:     if rightChild < size and dataset[rightChild].population > dataset[largest].population then
9:         largest ← rightChild
10:    end if
11:    if largest ≠ index then
12:        Swap dataset[index] and dataset[largest]
13:        MaxHeapify(dataset, size, largest) ▷ Recursively apply to the affected child
14:    end if
15: end function
```

Algorithm 2 Build Max Heap

```
1: function BuildMaxHeap(dataset, size)
2:     for i ← ⌊(size−1)/2⌋ downto 0 do
3:         MaxHeapify(dataset, size, i)
4:     end for
5: end function
```

1.1.3. HEAPSORT Procedure

The 'heapsort' function encapsulates the heapsort algorithm implemented in the prior parts of the project. Heapsort is a consistent comparison-based sorting algorithm working on $O(nlogn)$ time complexity in the worst, average, and best cases with a $O(1)$ space complexity. The algorithm consists of two main phases: building a max heap and the sorting phase.

In the first phase, a max heap is constructed from the input dataset using the previously implemented 'build_max_heap method'. This fact ensures that the largest element is residing at the root of the heap. The build_max_heap' function traverses the non-leaf nodes in reverse order, invoking 'max_heapify' to maintain the max-heap property, as discussed in the section 1.1.2.

The sorting phase follows, wherein the algorithm repeatedly extracts the maximum element from the heap and places it at the end of the array. This process, coupled with heap size reduction, continues until the entire array is sorted. The in-place nature of heapsort contributes to its efficiency and lowers the space complexity cost.

Algorithm 3 Heapsort

```
1: function Heapsort(dataset : vector of CityData, size : integer)
2:     BuildMaxHeap(dataset, size)                          ▷ Build a max-heap
3:     for i ← size − 1 downto 1 do
4:         SwapCityValues(dataset[0], dataset[i]) ▷ Swap the root with the last element
5:         MaxHeapify(dataset, i, 0)              ▷ Heapify to maintain max-heap property
6:     end for
7: end function
```

Heapsort's $O(nlogn)$ time complexity, arising from the logarithmic height of the heap and the linear max heap construction, renders it a dependable option in memory-constrained scenarios. In comparison to quicksort, heapsort lacks the average-case time efficiency of quicksort but compensates with its consistent performance, making it preferable in situations where worst-case time complexity is a critical consideration. While quicksort excels in average-case scenarios, heapsort's predictable behavior is advantageous when robustness in the face of varying inputs is paramount.

1.2. PRIORITY QUEUE OPERATIONS

1.2.1. MAX-HEAP-INSERT

The 'max_heap_insert' function adds a new element to the max heap, ensuring the max heap property is maintained. Its time complexity is $O(logn)$ due to the logarithmic height of the heap, accompanied by a space complexity of $O(1)$. This operation can be seen in applications like ticket booking systems. When a new person purchases a ticket, the system may use this operation to insert the booking into the priority queue, ensuring

that customers with higher priority are accordingly accommodated. The pseudo code is given:

Algorithm 4 Max Heap Insert

```
1: function MaxHeapInsert(dataset : vector of CityData, size : integer, city_data : CityData)
2:                                  ▷ Increase the size of the heap and add the new element
3:     size ← size + 1
4:     dataset.push_back(city_data)
5:     index ← size − 1
6:     parent ← (index − 1)/2
7:                                  ▷ Heapify to maintain the max heap property
8:     while index > 0 and dataset[parent].population < dataset[index].population do
9:         SwapCityValues(dataset[index], dataset[parent])
10:        index ← parent
11:        parent ← (index − 1)/2
12:    end while
13: end function
```

1.2.2. HEAP-EXTRACT-MAX

The 'heap_extract_max' function removes and returns the maximum element from the max heap. This operation's time complexity is $O(logn)$, and in scenarios like hospital emergency rooms, the operation proves can be utilized. As patients arrive with varying degrees of emergency, this operation would allow the system to efficiently extract and address the most critical cases first, and it would be possible to ensure closer to optimal medical attention for those in immediate need. The space complexity of the function is $O(1)$.

Algorithm 5 Heap Extract Max

```
1: function HeapExtractMax(dataset : vector of CityData, size : integer)
2:                                  ▷ Swap the root (max) with the last element
3:     SwapCityValues(dataset[0], dataset[size − 1])
4:     size ← size − 1
5:                                  ▷ Heapify to maintain the max heap property
6:     MaxHeapify(dataset, size, 0)
7:                                  ▷ Return the extracted max element
8:     return dataset[size]
9: end function
```

1.2.3. HEAP-INCREASE-KEY

The 'heap_increase_key' function increases the key of a specified element in the max heap, followed by heapification to maintain the max heap property. With a time complexity

of $O(logn)$ and a space complexity of $O(1)$, this operation may be used in cases like online games where players earn experience points over time. The procedure can be applied to update a player's level dynamically based on total experience. Pseudo code of the operation:

Algorithm 6 Heap Increase Key

```
1: function HeapIncreaseKey(dataset : vector of CityData, size : integer, index : integer, key : integer)
2:     index ← index − 1
3:     dataset[index].population ← key
4:     parent ← (index − 1)/2
5:                                          ▷ Heapify to maintain the max heap property
6:     while index > 0 and dataset[parent].population < dataset[index].population do
7:         SwapCityValues(dataset[index], dataset[parent])
8:         index ← parent
9:         parent ← (index − 1)/2
10:    end while
11: end function
```

1.2.4. HEAP-MAXIMUM

Lastly, the 'heap_maximum' function retrieves the maximum element from the max heap without removing it, operating in $O(1)$ time and space complexities. This function is efficient for scenarios where you need to inspect the highest-priority element without altering the heap, such as task management applications.In this context, the 'heap_maximum' function could be utilized to display the next upcoming task without actually removing the task from the list.

Algorithm 7 Heap Maximum

```
1: function HeapMaximum(dataset : vector of CityData)
2:     return dataset[0]
3: end function
```

1.3. d-ary HEAP OPERATIONS

The d-ary heap portion of this project extends the binary heap implementation to allow for a variable number of children for non-leaf nodes. This is achieved by introducing a parameter 'd' that specifies the maximum number of children that a node can have. The heapify, insert, and extract operations are modified to accommodate this change. The heapify operation, for instance, now compares a node with all its 'd' children to find the maximum, and the insert operation ensures that the heap property is maintained across all 'd' children of a node. This d-ary heap implementation provides a more flexible data structure that can potentially offer improved performance for certain types of data or use

cases. The program allows 'd' to be specified as an optional command-line argument, providing a convenient way to experiment with different heap structures.

1.3.1. Height Calculation

The 'dary_calculate_height' procedure computes the height of a d-ary heap, taking into account its size and the degree of each non-leaf node (d). With a logarithmic approach, the function iteratively divides the size by the degree, incrementing the height until the size reaches zero. This procedure has a time complexity of $O(log_d n)$. The space complexity of the function is $O(1)$ as well. My implementation of the procedure, which can be observed below,is a hand written reskin of the logarithmic height formula.

Algorithm 8 Method for calculating the height of a d-ary heap

```
1: function dary_calculate_height(size, d)
2:     height ← 0
3:     temp ← size
4:     while temp > 0 do
5:         temp ← temp − d^height
6:         height ← height + 1
7:     end while
8:     return height − 1
9: end function
```

1.3.2. EXTRACT-MAX Implementation

EXTRACT-MAX operation is executed by the 'dary_extract_max' function in a d-ary max-heap. It starts by swapping the root (max element) with the last element, reducing the heap size. Employing 'dary_max_heapify' then ensures the max-heap property is preserved. The space complexity of 'dary_max_heapify' function is $O(1)$, with a time complexity of $O(log_d n)$, with d as the degree and n as the number of elements. The pseudocode of the function:

Algorithm 9 Extract Max from a d-ary Heap

```
1: function DaryExtractMax(dataset, size, d)
2:     Swap(dataset[0], dataset[size − 1])
3:     size ← size − 1
4:     DaryMaxHeapify(dataset, size, 0, d)
5:     return dataset[size]
6: end function
```

1.3.3. INSERT Implementation

The 'dary_insert_element' function implements the INSERT operation for a d-ary max-heap. The function begins by adding the new element to the end of the heap array,

effectively treating it as a new leaf node. It then iteratively compares this new leaf with its parent node and swaps them if the leaf's value is greater, continuing this process up the tree until the heap property is restored. This "sift-up" operation ensures that the heap remains a valid d-ary max-heap after the insertion. The running time of the 'dary_insert_element' function is $O(log_d n)$, where 'd' is the maximum number of children of a node (the 'd' in 'd-ary') and 'n' is the number of elements in the heap. This is because in the worst case, we may have to move the new element from the leaf level to the root, which requires traversing the height of the heap, and the height of a d-ary heap with 'n' elements is $log_d n$. The space complexity of the function is $O(1)$.

Algorithm 10 Inserting a new element into a d-ary heap

```
1:  function DaryInsertElement(dataset, size, city_data, d)
2:      DaryBuildMaxHeap(dataset, size, d)        ▷ Build a max heap for the d-ary heap
3:      size ← size + 1                                  ▷ Increase the size of the heap
4:      dataset.push_back(city_data)                          ▷ Add the new element
5:      index ← size − 1
6:      parent ← (index − 1)/d
7:      while index > 0 and dataset[parent].population < dataset[index].population do
8:          SwapCityValues(dataset[index], dataset[parent])      ▷ Swap and recursively
    heapify
9:          index ← parent
10:         parent ← (index − 1)/d
11:     end while
12: end function
```

1.3.4. INCREASE-KEY Implementation

The 'dary_increase_key' function is the implementation of the INCREASE-KEY operation. This operation involves updating the key of a specified element to a new value, ensuring the max-heap property is maintained. The function begins by checking if the new key is smaller than the current key, printing a message and exiting the operation if so. If the new key is larger, the function updates the key and initiates a heapify process. Similar to 'max_heapify,' the algorithm traverses the heap upwards, swapping elements with their parent as necessary to uphold the max-heap property. The time complexity of 'dary_increase_key' is influenced by the height of the heap, considering both 'd' and 'n' in the analysis. It is $O(log_d n)$, accompanied by a space complexity of $O(1)$. This function's efficiency is crucial for scenarios requiring dynamic adjustments to element priorities in a d-ary max-heap, facilitating adaptability in applications like priority queues or resource allocation.

Algorithm 11 Method for increasing the key of an element in a d-ary heap

```
1: function dary_increase_key(dataset, size, index, key, d)
2:     if key ≤ dataset[index].population then
3:         print "New key is smaller than current key"
4:         return
5:     end if
6:     index ← index − 1
7:     dataset[index].population ← key
8:     parent ← (index − 1)/d
9:     while index > 0 and dataset[parent].population < dataset[index].population
   do
10:        swap_city_values(dataset[index], dataset[parent])
11:        index ← parent
12:        parent ← (index − 1)/d
13:    end while
14: end function
```

1.4. Comparison With Quicksort

As it can be observed in Table 1.1, Quicksort consistently outperformed Heapsort in terms of execution speed. In typical cases, Heapsort takes slightly more time in nanoseconds then Quicksort. However, Heapsort has a more consistent performence due to its static $O(nlogn)$ complexity. This difference of speed between the two algorithms also rely on the pivoting strategy used in Quicksort algorithm. As the optimal strategy for Quicksort is randomized pivoting in most scenerios, randomized pivoting is utilized as the strategy in Table 1.1.

When analyzing the comparison count between Quicksort and Heapsort, the efficiency of Quicksort predominantly hinges on the number of comparisons conducted during the partitioning part of the algorithm. On average, it manages O($n \log n$) comparisons with random pivoting and randomized data, not specially sequenced, reflecting its favorable performance. On the contrary, in the worst-case scenario, when the pivot selection made results in an unbalanced special order, as seen in scenarios like already sorted data, Quicksort's comparison count can escalate to O(n^2) which is much higher then desired. In comparison to quicksort, heapsort generally entails a reduced number of comparisons. The inherent heap property guarantees that a parent node consistently surpasses (or is surpassed by) its children, mitigating the necessity for extensive comparisons throughout the sorting procedure. In summary, comparison number in Quicksort depends on pivot selection and partitioning. Thus, in most cases, Quicksort makes more comparisons then Heapsort.

All in all, Heapsort's strengths lies on consistent complexity and performance and it may be utilized for situations with large input size and less memory, even though having a weakness of working on $O(nlogn)$, sometimes slower compared to other algorithms. On the contrast, main strength of Quicksort is it being faster in most datasets, especially

ones with smaller data size. And its main weakness is the probability of having a poor pivot selection strategy, and not having a time complexity as stable as Heapsort's complexity. However, the final decision to implement and utilize whether Quicksort or Heapsort, is dependent on the specific needs of the user and the properties of the input data.

	Population1	Population2	Population3	Population4
Quick Sort	10 839 300	8 704 200	11 017 000	12 257 600
Heap Sort	15 792 953	16 938 601	16 923 530	18 144 537

Table 1.1: Comparison of Heapsort and Quikcsort algorithms