Analysis of Algorithms

BLG 335E

Project 2 Report

ÖMER YILDIZ
yildizom20@itu.edu.tr

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

MAX-HEAPIFY Procedure

The MAX-HEAPIFY procedure is vital for maintaining the max-heap property within a binary heap. Given an array representation of a nearly complete binary tree and a target index, the procedure compares the element at the specified index with its left and right children. If necessary, it swaps the element with the larger of its children and recursively applies MAX-HEAPIFY to the affected subtree. This ensures that the subtree rooted at the given index remains a max-heap.

The time complexity of MAX-HEAPIFY is crucial to its efficiency. At each level of recursion, the procedure compares the element with its children, and the recursion proceeds to the child with the maximum value. The worst-case scenario occurs when the element needs to traverse the entire height of the tree, which is $\log n$ in a binary heap of size n. Therefore, the time complexity of MAX-HEAPIFY is $O(\log n)$.

This operation is fundamental to the efficiency of other heap operations, especially during the construction of a heap and the HEAPSORT algorithm.

BUILD-MAX-HEAP Procedure

The BUILD-MAX-HEAP procedure constructs a max-heap from an unordered input array. Starting from the last non-leaf node and moving upwards, it applies the MAX-HEAPIFY procedure to each node. This ensures that each subtree rooted at a non-leaf node is a max-heap. The procedure runs in linear time, $O(n)$, where n is the size of the input array.

The intuition behind the linear time complexity lies in the fact that nodes lower in the tree have larger subtrees, and therefore, fewer nodes require the MAX-HEAPIFY operation. The procedure only needs to be applied to non-leaf nodes, and since there are $n/2$ non-leaf nodes in a heap of size n, the overall complexity is $O(n)$.

HEAPSORT Procedure

The HEAPSORT procedure utilizes the BUILD-MAX-HEAP and MAX-HEAPIFY procedures to sort an array in place. The algorithm begins by building a max-heap from the input array using BUILD-MAX-HEAP. After constructing the heap, the largest element (at the root) is swapped with the last element in the array, and the heap size is reduced. The MAX-HEAPIFY procedure is then applied to the root to restore the max-heap property. This process is repeated until the entire array is sorted.

The time complexity of HEAPSORT is $O(n \log n)$. The BUILD-MAX-HEAP step takes $O(n)$ time, and the subsequent MAX-HEAPIFY operation, performed n times, contributes $O(n \log n)$ to the overall complexity.

HEAPSORT offers an in-place sorting solution with a relatively favorable time complexity compared to other algorithms.

Heapsort and D-ary Max Heap with Priority Queue Operations

MAX-HEAP-INSERT

Listing 1: MAX-HEAP-INSERT Operation

```
void maxHeapInsert(CityData* heap, int& heapSize, const CityData& key, in
    // Insert key into the max heap
    heap[heapSize] = key;
    dary_heapify_Up(heap, heapSize, d, counter);
    heapSize++;
}
```

Explanation: The MAX-HEAP-INSERT operation inserts a new element into the max heap represented as an array. It maintains the max-heap property using the `dary_heapify_Up` operation.

HEAP-EXTRACT-MAX

Listing 2: HEAP-EXTRACT-MAX Operation

```
CityData heapExtractMax(CityData* heap, int& heapSize, int d, ComparisonC
    // Extract the maximum element from the max heap
    if (heapSize == 0) {
        std::cerr << "Heap␣is␣empty.␣Cannot␣extract␣max." << std::endl;
        return CityData(); // Assuming an empty CityData as an indicator
    }

    CityData maxElement = heap[0];
    heap[0] = heap[--heapSize];
    dary_heapify_down(heap, 0, heapSize, d, counter);

    return maxElement;
}
```

Explanation: The HEAP-EXTRACT-MAX operation removes and returns the maximum element from the max heap represented as an array. It maintains the max-heap property using the `dary_heapify_down` operation.

HEAP-INCREASE-KEY

Listing 3: HEAP-INCREASE-KEY Operation

```
void heapIncreaseKey(CityData* heap, int index, const CityData& newKey, i
    // Increase the key of a specified element in the max heap
    if (newKey.population < heap[index].population) {
        std::cerr << "New␣key␣is␣smaller␣than␣the␣current␣key.␣Cannot␣inc
    } else {
        heap[index] = newKey;
        dary_heapify_Up(heap, index, d, counter);
    }
}
```

Explanation: The HEAP-INCREASE-KEY operation increases the key of a specified element in the max heap represented as an array. It maintains the max-heap property using the `dary_heapify_Up` operation.

HEAP-MAXIMUM

Listing 4: HEAP-MAXIMUM Operation

```
CityData heapMaximum(CityData* heap) {
    // Return the maximum element from the max heap without removing it
    return heap[0];
}
```

Explanation: The HEAP-MAXIMUM operation returns the maximum element from the max heap represented as an array without removing it.

Example Usage in a Real-World Scenario

Assume we are managing a database of cities with their populations and we want to use heaps for various operations.

1. MAX-HEAP-INSERT:

Suppose we want to add a new city to our database.

Listing 5: Example MAX-HEAP-INSERT Usage

```
CityData* cityHeap = new CityData[MAX_SIZE];
int heapSize = 0;
ComparisonCounter counter{0};

CityData newCity("NewCity", 1000000);
maxHeapInsert(cityHeap, heapSize, newCity, 3, counter);
```

2. HEAP-EXTRACT-MAX:

Let’s say we want to find the most populated city and remove it from our database.

Listing 6: Example HEAP-EXTRACT-MAX Usage

```
CityData* cityHeap = new CityData[MAX_SIZE];
int heapSize = 0;
ComparisonCounter counter{0};

CityData mostPopulatedCity = heapExtractMax(cityHeap, heapSize, 3, counte
```

Here, we extract the city with the highest population from our max heap.

3. HEAP-INCREASE-KEY:

Suppose the population of an existing city has increased, and we need to update its data in the database.

Listing 7: Example HEAP-INCREASE-KEY Usage

```
CityData* cityHeap = new CityData[MAX_SIZE];
int heapSize = 0;
ComparisonCounter counter{0};

CityData existingCity("ExistingCity", 2000000);
heapIncreaseKey(cityHeap, 2, existingCity, 3, counter);
```

Here, we increase the population key of an existing city in the max heap, ensuring the heap property is maintained.

4. HEAP-MAXIMUM:

Suppose we want to know which city is currently the most populated without removing it.

Listing 8: Example HEAP-MAXIMUM Usage

```
CityData* cityHeap = new CityData[MAX_SIZE];
int heapSize = 0;
ComparisonCounter counter{0};

CityData mostPopulatedCity = heapMaximum(cityHeap);
```

In this scenario, we retrieve information about the city with the highest population without altering the database.

Performance Analysis of HeapSort and QuickSort

(a) Elapsed Time Comparison

1. **dummyData:**
    - QuickSort: 9128 ns
    - HeapSort: 54287 ns
    - **Observation:** QuickSort outperforms HeapSort in terms of elapsed time on this small dataset.
2. **population1 (Random values):**
    - QuickSort: 652478 ns
    - HeapSort: 7635625 ns
    - **Observation:** QuickSort is faster than HeapSort on this larger dataset with random values.
3. **population2 (Sorted values):**
    - QuickSort: 43615404 ns
    - HeapSort: 8848125 ns
    - **Observation:** QuickSort takes longer on this dataset where values are already sorted.
4. **population3 (Reverse-sorted values):**
    - QuickSort: 51927958 ns
    - HeapSort: 7696000 ns

- **Observation:** HeapSort outperforms QuickSort on this dataset with reverse-sorted values.

(b) Number of Comparisons Comparison

1. **dummyData:**
    - QuickSort: 70 comparisons
    - HeapSort: 146 comparisons
    - **Observation:** QuickSort requires fewer comparisons on this small dataset.
2. **population1 (Random values):**
    - QuickSort: 294589 comparisons
    - HeapSort: 336175 comparisons
    - **Observation:** QuickSort has fewer comparisons on this larger dataset with random values.
3. **population2 (Sorted values):**
    - QuickSort: 111746 comparisons
    - HeapSort: 363517 comparisons
    - **Observation:** QuickSort performs fewer comparisons on this dataset with sorted values.
4. **population3 (Reverse-sorted values):**
    - QuickSort: 3809363 comparisons
    - HeapSort: 3875907 comparisons
    - **Observation:** QuickSort requires fewer comparisons on this dataset with reverse-sorted values.(population3 and population4 results are nearly same thats why ı did not add population4 to the report)

(c) Insights into Strengths and Weaknesses

QuickSort:

- **Strengths:**
    - Efficient on average, especially with random datasets.
    - Performs fewer comparisons.
- **Weaknesses:**
    - Can be slower on already sorted or reverse-sorted datasets.

HeapSort:

- **Strengths:**
  - Consistent performance across different datasets.
- **Weaknesses:**
  - Tends to have higher elapsed time and more comparisons compared to QuickSort.

Scenario Analysis:

- Use QuickSort when the dataset is large and partially ordered.
- Use HeapSort when a worst-case guarantee is essential, or the dataset characteristics are unknown.

Summary

The choice between QuickSort and HeapSort depends on the dataset characteristics. QuickSort generally performs well on average, while HeapSort provides more consistent performance across various scenarios. Understanding the nature of the data is crucial for selecting the most appropriate algorithm.