Analysis of Algorithms

BLG 335E

Project 2 Report

Erblina Nivokazi
nivokazi21@itu.edu.tr

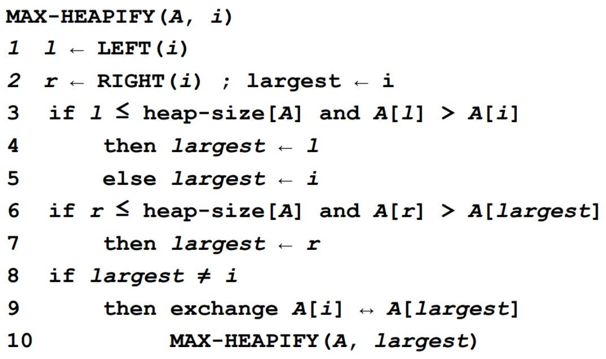
Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 14.12.2023

1. Implementation

1.1. Implementation of Max-Heapify:

1.1.1. Implementation Details

When implementing Max-Heapify I checked the pseudo-codes provide to us in the slides.



```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i) ; largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
         MAX-HEAPIFY(A, largest)
```

Figure 1.1: Max-Heapify pseudo-code

The purpose of the max-heapify function is to correct a single defect of the max-heap property in a subtree, so it iterates and fixes one node of the tree. It assumes that the binary trees rooted at the left and right children of the current node are max-heaps but that the current node selected could be smaller than its children, thus it's needed to fix it because it is violating the max-heap property. My implementation is below:

```
void max_heapify(std::vector<Population> &A, int i, int heapSize)
{
    int leftNode = 2 * i + 1;
    int rightNode = 2 * i + 2;
    int largest = i;
    if (leftNode <= heapSize && A[leftNode].population > A[largest].population)
    {
        largest = leftNode;
    }
    if (rightNode <= heapSize && A[rightNode].population > A[largest].population)
    {
        largest = rightNode;
    }
    if (largest != i)
    {
        Population temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        max_heapify(A, largest, heapSize);
    }
}
```

This implementation takes as parameters the object **Population vector**, which contains the list of the data sets that are read from the the given csv files, the variable **i**, which is an integer and contains the index of the position of the element in which we want to apply the max-heapify function, and the **heapSize** variable which is an integer and contains the size of the heap.

At first I make two new variables **leftNode** and **rightNode** which contain the integers in which the indeces of the left and right children of the index i are saved. Then with if statements we check whether or not the largest value is in the index given to us or in one of the children. If any of the children has a larger value than the parent node then they swap positions and max_heapify function is called recursively.

1.1.2. Related Recurrence Relation

The node at index I and its left and right children are compared by the function. The function switches the node with the child with the maximum value if either child has a larger value. Then, in order to preserve the max-heap property, this procedure is applied recursively to the switched child. The time complexity of `max_heapify` could be found by the recurrence relation:

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

where n represents the size of the heap.

The recursiveness of this function is shown by the recurrence relation, which indicates that the time complexity of the function is proportional to the size of the subheap that is under evaluation for the function.

1.1.3. Time and Space Complexity

The time complexity can be determined by the functions, since the function I have is recursive and it can be solved it is seen that it is logarithmic. The recurrence relation for `max_heapify` function is:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

I solved it using Master's Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Comparing with the recurrence relation, I have $a = 1$, $b = \frac{3}{2}$, and $f(n) = O(1)$. The Master Theorem states that if $f(n)$ is $O(n^c)$ for $c < \log_b a$, then concluding that the time complexity is $O(n^{\log_b a})$.

In this case:

$$\log_{\frac{3}{2}} 1 = 0$$

Since $c = 0 < \log_{\frac{3}{2}} 1$, it falls into the first case (Case 1) of the Master Theorem, and the solution is:

$$T(n) = O\left(n^{\log_{\frac{3}{2}} 1}\right) = O(\log n)$$

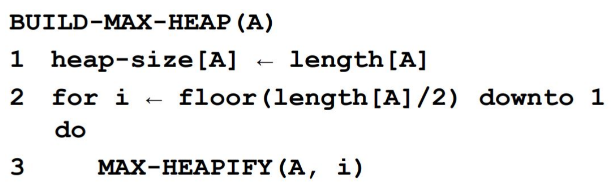
So the time complexity of `max_heapify` is $O(\log n)$.

The space complexity should be `O(log n)` because of the recursive calls made in the stack.

1.2. Implementation of Build-Max-Heap

1.2.1. Implementation Details

The implementation of the `build_max_heap` has in focus to build a heap that strictly applies to the rules of the max heap, this implementation uses `max_heapify` to apply to all the nodes beginning from the ones in the end until the ones in the beginning. I used this pseudo-code as a reference point:



```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← floor(length[A]/2) downto 1
   do
3     MAX-HEAPIFY(A, i)
```

Figure 1.2: Build-Max-Heap pseudo-code

My implementation:

```
void build_max_heap(std::vector<Population> &A)
{
    int heapSize = A.size() - 1;

    for (int i = (heapSize / 2) - 1; i >= 0; i--)
    {
        max_heapify(A, i, heapSize);
    }
}
```

The implementation takes the **object Population vector** as a parameter. the **heapSize** integer variable is retrieved to be used to find the last index's value. In the loop **integer i** to go through the nodes from the end (the leaves) to the root of the tree, thus building the Max Heap.

1.2.2. Related Recurrence Relation

The `build_max_heap` function is responsible for constructing (or as the name says building) a max-heap from an unordered array. It starts by considering the second half of the array as leaves, and then iteratively applies the `max_heapify` procedure to each node from the bottom up.

The time complexity of `build_max_heap` is determined by its recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$$

where n represents the size of the heap.

1.2.3. Time and Space Complexity

The time complexity of the `build_max_heap` function is $O(n)$. Unlike `max_heapify`, which has a logarithmic time complexity, `build_max_heap` operates in a bottom-up manner.

The function iterates through the second half of the array (from index $\frac{n}{2} - 1$ to 0) and calls `max_heapify` on each element. This ensures that each subtree that is rooted at these elements is a valid max-heap. As there are $\frac{n}{2}$ elements in this range, and each call to `max_heapify` will take constant time ($\Theta(1)$), the overall time complexity is $O(n)$. And it has space complexity of $O(1)$.

1.3. Implementation of HeapSort

My `heapsort` function is an implementation of the Heap Sort algorithm, by considering the Max-Heap data structure. The overall process involves building a Max-Heap from the input array and iteratively extracting the maximum element, moving it to the end of the array, thus being left with a sorted array in the end.

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← floor(length[A]/2) downto 1
   do
3     MAX-HEAPIFY(A, i)
```

Figure 1.3: Heapsort pseudo-code

My implementation:

```
void heapsort(std::vector<Population> &A)
{
    build_max_heap(A);

    int length = A.size();
    for (int i = length - 1; i >= 1; i--)
    {
        Population temp = A[i];
        A[i] = A[0];
        A[0] = temp;
        length--;
        max_heapify(A, 0, length);
    }
}
```

The `heapsort` function commences by calling the `build_max_heap` function to ensure that the input array satisfies the Max-Heap property. Then, a loop is executed to iteratively extract the maximum element by swapping it with the last leaf in the heap, reducing the size of the heap, and restoring the Max-Heap property using `max_heapify`.

1.3.1. Related Recurrence Relation

The `heapsort` function relies on the `max_heapify` and `build_max_heap` function to maintain the Max-Heap property during the extraction of the maximum element. The recurrence relation for `heapsort` is given by:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

where n denotes the size of the heap.

The time complexity is related to the size of the subheap that is being processed in each recursive call.

1.3.2. Time and Space Complexity

The time complexity of `heapsort` is $O(n \log n)$, where n is the size of the input array. This is because each call to `max_heapify` takes $O(\log n)$ time, and the loop iterates n times.

1.4. Implementation of Priority Queue Operations

1.4.1. Max_Heap_Insert

1.4.1.1. Implementation Details

My `max_heap_insert` function implementation is supposed to insert a new element into a Max-Heap which is represented as a vector. My code follows these steps:

```
void max_heap_insert(std::vector<Population> &A, int num)
{
    Population new_element;
    new_element.city = " ";
    new_element.population = num;
    A.push_back(new_element);
    build_max_heap(A);
}
```

In my implementation, a new `Population` object is created with a default city name and the provided population value. This new element is then added to the end of the vector representing the Max-Heap using `push_back`. Finally, the `build_max_heap` function is called to maintain the Max-Heap property after the insertion.

1.4.1.2. Related Recurrence Relation

The `max_heap_insert` function inserts a new element into a Max-Heap represented as a vector. Its time complexity is influenced by the `build_max_heap` function, thus the recurrence relation is given by:

$$T(n) = O(n)$$

where n is the size of the heap.

1.4.1.3. Time and Space Complexity

The time complexity of `max_heap_insert` is $O(n)$, where n is the size of the Max-Heap vector. This is influenced by the `build_max_heap` function, which has a time complexity of $O(n)$ per insertion and I am calling `build_max_heap` so that the array get in Max Heap structure after the addition.

The space complexity of `max_heap_insert` is $O(1)$ since it only uses a constant amount of additional space for the new element.

1.4.2. Heap_Extract_Max

1.4.2.1. Implementation Details

My implementation of the `heap_extract_max` function is designed to extract the maximum element from a Max-Heap, which is represented as a vector. My code:

```
Population heap_extract_max(std::vector<Population> &A)
{
    int heapSize = A.size();
    if (heapSize < 1)
    {
        throw std::runtime_error("Heap Underflow");
    }
    build_max_heap(A);
    Population max = A[0];
    int last = heapSize - 1;
    A[0] = A[last];
    A.pop_back();

    heapSize--;
    max_heapify(A, 0, heapSize);
    return max;
}
```

In this implementation, the function first checks if the heap size is less than 1, throwing a runtime error for heap underflow if true. Then, it builds the max-heap to ensure the structure is in the max heap position, allowing the extraction of the maximum value (`A[0]`). The maximum value is stored in the `max` variable, and the last element in the heap replaces it. The last element is then removed from the heap using `pop_back()`. The heap size is adjusted accordingly, and `max_heapify` is called to maintain the max heap property. Finally, the extracted maximum value is returned.

1.4.2.2. Related Recurrence Relation

The `heap_extract_max` function, which uses both the `build_max_heap` and the `max_heapify` function to maintain the Max-Heap property during the extraction of the maximum element, has a recurrence relation like this:

$$T(n) = O(n) + O(\log n)$$

where n represents the size of the heap.

1.4.2.3. Time and Space Complexity

The time complexity of `heap_extract_max` is $O(n)$, where n is the size of the Max-Heap vector. This time complexity is influenced by the `build_max_heap` function, which has a time complexity of $O(n)$ per extraction, and since `build_max_heap` is called after extracting the maximum element to maintain the Max-Heap structure. In other cases it would be $O(n \log n)$, but I had to make sure the heap is in Max-Heap structure so I called the `build_max_heap` function.

The space complexity of `heap_extract_max` is $O(1)$ since it only uses a constant amount of additional space for local variables.

1.4.3. Heap_Increase_Key

1.4.3.1. Implementation Details

My implementation of the `heap_increase_key` function is intended to increase the key of a specific element in a Max-Heap represented as a vector. The code follows these steps:

```
void heap_increase_key(std::vector<Population> &A, int i, int key)
{
    build_max_heap(A);
    if (key < A[i].population)
    {
        throw std::runtime_error("New key is smaller than the current key");
    }
    A[i].population = key;

    while (i > 0)
    {
        int parent = (i - 1) / 2;
        if (A[i].population > A[parent].population)
        {
            Population temp = A[i];
            A[i] = A[parent];
            A[parent] = temp;
            i = parent;
        }
        else
        {
            break;
        }
    }
}
```

In this implementation, I rebuild the max heap to ensure the correct positions of the indeces. The function checks if the new key is smaller than the current key, throwing a runtime error if true. The key of the specified element is then updated with the new value. A loop is used to check and read just the position of the new key relative to its parent by comparing population values. If the new key is greater than its parent's key, the objects are swapped, and the loop continues. The loop breaks when the new key is no longer greater than its parent's key.

1.4.3.2. Related Recurrence Relation

The heap_increase_key function, which utilizes the build_max_heap function, has a recurrence relation represented as:

$$T(n) = O(n) + \sum_{i=0}^{k} O(\log i)$$

where n denotes the size of the Max-Heap.

1.4.3.3. Time and Space Complexity

The time complexity of heap_increase_key is $O(n)$, where n is the size of the Max-Heap vector. This is influenced by the build_max_heap function since I use it in this function to ensure that the heap is in max-heap structure. Without the build_max_heap function the time complexity would've been influenced by the while loop and it would've been $O(\log n)$.

The space complexity of heap_increase_key is $O(1)$ since it only uses a constant amount of additional space for local variables.

1.4.4. Heap_Maximum

1.4.4.1. Implementation Details

My implementation of the heap_maximum function is designed to retrieve the maximum element from a Max-Heap. The code:

```
Population heap_maximum(std::vector<Population> &A)
{
    int heapSize = A.size();
    if (heapSize < 1)
    {
        throw std::runtime_error("Heap Underflow");
    }
    build_max_heap(A);
    return A[0];
}
```

In this implementation, the function first checks if the heap size is less than 1, throwing a runtime error for heap underflow if true. Then, it builds the max-heap to ensure the structure is in the max heap position. Finally, the function returns the element at the front of the vector, representing the maximum element in the Max-Heap.

1.4.4.2. Related Recurrence Relation

The `heap_maximum` function, which utilizes the `build_max_heap` function to ensure the Max-Heap property, can be described by the following recurrence relation:

$$T(n) = O(n)$$

where n represents the size of the Max-Heap.

1.4.4.3. Time and Space Complexity

The time complexity of `heap_maximum` is $O(n)$, where n is the size of the Max-Heap vector. This complexity arises from the `build_max_heap` function, which takes $O(n)$ time per extraction, as it iterates through the height of the heap while performing constant-time operations. Normally the `heap_maximum` would've taken time complexity of $O(1)$, but for this homework we had to assume or be cautious to the possibility that the input given to us is not in Max-Heap structure The function ensures the Max-Heap property.

The space complexity of `heap_maximum` is $O(1)$ as it only uses a constant amount of additional space for local variables within the function.

1.5. Implementation of d-ary Heap Operations

1.5.1. Height Calculation with dary_calculate_height function

1.5.1.1. Implementation Details

The `dary_calculate_height` function calculates the height of a d-ary heap based on the given parameters d and n. The formula that I used in my implementation is derived from James Storer's "An Introduction to Data Structures and Algorithms":

$$\text{height} = \frac{\log(n \cdot (d-1) + 1)}{\log(d)}$$

In this formula, n is the size of the heap, and d is the number of children each node can have. The function uses a logarithmic function to compute the height, and the result is returned as an integer value.

1.5.1.2. Related Recurrence Relation / Time and Space Complexity

The `dary_calculate_height` function doesn't have a recurrence relation. Instead, it directly computes the height using a formula. Therefore, its time complexity is constant ($\Theta(1)$) as it performs a fixed number of operations regardless of the input size. The space complexity is $O(1)$.

1.5.2. Dary_Extract_Max

1.5.2.1. Implementation Details

My implementation of the `dary_extract_max` function is designed to extract the maximum element from a Max-Heap and return it. The code follows these steps:

```
Population dary_extract_max(std::vector<Population> &A, int d)
{
    int heapSize = A.size();
    if (heapSize < 1)
    {
        throw std::runtime_error("Heap Underflow");
    }
    daryBuildMaxHeap(A, d);
    Population max = A[0];
    int last = heapSize - 1;
    A[0] = A[last];
    A.pop_back();
    daryMaxHeapify(A, 0, last, d);
    return max;
}
```

In this implementation, the function first checks if the heap size is less than 1, throwing a runtime error for heap underflow if true. Then, it ensures the heap is a MaxHeap with varying arity by calling `daryBuildMaxHeap`. The maximum value is stored in the `max` variable, and the last element in the heap replaces it. The last element is then removed from the heap using `pop_back()`. `daryMaxHeapify` is called to maintain the MaxHeap property, considering the varying arity. Finally, the extracted maximum value is returned.

1.5.2.2. Related Recurrence Relation

The `dary_extract_max` function, utilizing the `daryBuildMaxHeap` and `daryMaxHeapify` functions for varying arity, can be described by the following recurrence relation:

$$T(n) = O(n) + O(\log_d n)$$

where n represents the size of the Max-Heap and d is the arity parameter.

1.5.2.3. Time and Space Complexity

The time complexity of `dary_extract_max` is $O(n)$, where n is the size of the Max-Heap vector. The complexity is influenced by the `daryBuildMaxHeap` function, which has a time complexity of $O(n)$ per extraction. In a normal scenario where the input is in Max-Heap structure, the time complexity would be $O(1)$.

The space complexity of `dary_extract_max` is $O(1)$ as it only uses a constant amount of additional space for local variables within the function.

1.5.3. Dary_Insert_Element

1.5.3.1. Implementation Details

My implementation of the `dary_insert_element` function is made to insert a new element into a Max-Heap. The code follows these steps:

```
void dary_insert_element(std::vector<Population> &A, Population new_element, int d)
{
    A.push_back(new_element);
    daryBuildMaxHeap(A, d);
}
```

In this implementation, a new `Population` object is added to the end of the vector representing the Max-Heap. The function then calls `daryBuildMaxHeap` to ensure the Max-Heap property is maintained with the varying arity provided.

1.5.3.2. Related Recurrence Relation

The `dary_insert_element` function, utilizing the `daryBuildMaxHeap` function, can be described by the following recurrence relation:

$$T(n) = O(n)$$

where n represents the size of the Max-Heap, and d is the arity parameter.

1.5.3.3. Time and Space Complexity

The time complexity of `dary_insert_element` is $O(d \cdot \log n)$, where n is the size of the Max-Heap vector and d is the arity parameter. The complexity is influenced by the `daryBuildMaxHeap` function, which has a time complexity of $O(n)$ per insertion. This function iterates through the height of the heap while performing constant-time operations, and since it is called after inserting a new element to ensure the Max-Heap structure, it takes $O(d \cdot \log n)$ time. In a normal scenario where the input is already in Max-Heap structure, the time complexity would be $O(1)$.

The space complexity of `dary_insert_element` is $O(1)$ as it only uses a constant amount of additional space for local variables within the function.

1.5.4. Dary_Increase_Key

1.5.4.1. Implementation Details

My implementation of the `dary_increase_key` function is designed to increase the key of a specific element in a Max-Heap with varying arity represented as a vector. The code follows these steps:

```
void dary_increase_key(std::vector<Population> &A, int i, int k, int d)
{
    daryBuildMaxHeap(A, d);
    if (k < A[i].population)
    {
        throw std::runtime_error("New key is smaller than the current key");
    }

    A[i].population = k;
    while (i > 0)
    {
        int parent = (i - 1) / d;

        if (A[i].population > A[parent].population)
        {
            Population temp = A[i];
            A[i] = A[parent];
            A[parent] = temp;
            i = parent;
        }
        else
        {
            break;
        }
    }
}
```

In this implementation, the function first ensures the Max-Heap property by calling `daryBuildMaxHeap` with the given arity (`d`). It then compares the new key (`k`) with the current key of the specified element. If the new key is smaller, it throws a runtime error. Otherwise, it updates the key and iteratively checks and adjusts the position by swapping with the parent until the Max-Heap property is restored.“‘

1.5.4.2. Related Recurrency Relation

The `dary_increase_key` function, which relies on the `daryBuildMaxHeap` function to maintain the Max-Heap property with varying arity, can be described by the following recurrence relation:

$$T(n) = O(n) + \sum_{i=0}^{k} O(\log_d i)$$

where n denotes the size of the Max-Heap, and d is the arity parameter.

1.5.4.3. Time and Space Complexity

The time complexity of `dary_increase_key` is $O(n)$, where n is the size of the Max-Heap vector. This complexity is influenced by the `daryBuildMaxHeap` function, which has a time complexity of $O(n)$. The function iterates through the height of the heap while performing the operations and is called after updating a key to ensure the Max-Heap structure.

The space complexity of `dary_increase_key` is $O(1)$ as it only uses a constant amount of additional space for local variables within the function.

1.6. Discussion of the Outcomes

Taking the tables from the last report as a reference and running the second project with the same csv files and putting the results in the same table is helpful to draw and review the comparisons. I chose to run the Hybrid Quicksort of the first project with the Median of Three pivoting strategy and a threshhold k of 10:

	Population1	**Population2**	**Population3**	**Population4**
Hybrid/Median of 3/k=10	2243800 ns	1694500 ns	2340800 ns	3118800 ns
Heapsort	13598900 ns	14760500 ns	15047200 ns	18277300 ns

Table 1.1: Comparison of Hybrid Quicksort and Heapsort on input data.

For these Datasets it is evidently clear that he Hybrid Quicksort, in this case with pivoting strategy Median of three, is faster than the Heapsort. This is the expected outcome since with Heapsort the program will always swap the array even if it is not necessary. Whereas with Quicksort swapping is done only when necessary. The shortcome of Quicksort is that it has a worst case of $O(n)$, but Heapsort will always operate in $O(\log n)$ time.

Quicksort is known to be a widely-used sorting algorithm known that not only is efficient but also adaptable.

1. **Partitioning:** Quicksort practices a divide-and-conquer strategy, it partitions the array based on a chosen pivot. The algorithm is affected by our choice of pivot.

2. **In-Place Sorting:** Quicksort is often implemented as an in-place sorting algorithm, this minimizes extra memory requirements.

3. **Efficiency:** Quicksort exhibits good performance due to its memory access patterns, these can be both sequential and localized, which leads to efficient execution on actual real-world machines.

Heapsort, differs from Quicksort in several aspects:

1. **Heap Structure:** Heapsort builds a max-heap (or min-heap) structure and the maximum (or minimum) element is repeatedly extracted. The max heap structure makes sure that the largest element is always at the root.

2. **Not In-Place:** Heapsort is usually not implemented as an in-place algorithm. The heapsort requires additional memory to store the structure of the heap, which can impact efficiency.

3. **Less Efficient for Small Datasets:** Heapsort might take a bit longer to sort things when there is not a lot of data to sort. Quicksort tends to be quicker when there's not a lot of data to organize.

Comparing binary heap with d-ary heap

To understand the topic and the homework more, I wanted to also compare one of the heap functions with its equivalent in d-ary. In this case I increased the key at index 5 in both functions, but in d-ary I made d to be 4, and I got these running times:

	Population1	**Population2**	**Population3**	**Population4**
heap_increase_key	211626200 ns	219082200 ns	172717500 ns	144633200 ns
dary_increase_key	531400 ns	873700 ns	135900 ns	1610500 ns

Table 1.2: Comparison of Heap Increase Key and Dary Increase Key on different input data.

As can be seen in the table the d-ary increase key function is much faster than the normal heap increase function. Since in d-ary the number of children is bigger than in binary tree, the number of ancestors that a random node might have is smaller in d-ary than in binary. So evidently if a key is increased in d-ary it has to compare itself to a smaller number of parents (ancestors) than the random node from the binary tree.