Analysis of Algorithms

BLG 335E

Project 2 Report

Abdullah Jafar Mansour Shamout
shamout21@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

1.1.1. Implementation Details

The max heapify's goal is to put the ith element in the correct place in a portion of the array that “almost” has the heap property. My implimentation can be seen below:

```
void max_heapify(std::vector<Population> &array, const int &i, const
    int &array_size) {
     int left = 2 * i + 1;
     int right = 2 * i + 2;
     int largest;

     if (left < array_size && array[left].population > array[i].
         population)
          largest = left;
     else
          largest = i;

     if (right < array_size && array[right].population > array[largest
         ].population)
          largest = right;

     if (largest != i) {
          swap_array_value(array, i, largest);
          max_heapify(array, largest, array_size);
     }
}
```

Listing 1.1: MAX-HEAPIFY Implementation

This implementation takes a vector of `Population` objects, an index `i`, and the size of the array as parameters. It ensures the max-heap property is preserved.

This implementation works as follows:

1. Calculate the indices of the left and right children of the current node.

2. Compare the values of the current node, left child, and right child to find the index of the largest element.

3. If the largest element is not at the current node's index, swap the values and recursively apply MAX-HEAPIFY on the affected child.

In essence, MAX-HEAPIFY ensures that the subtree rooted at the current node conforms to the max-heap property, allowing for efficient heap operations like building a heap, sorting with heap sort, and priority queue operations.

1.1.2. Time, Space complexities and related recurrence relations

The time complexity of the MAX-HEAPIFY procedure is $O(\log n)$, where n is the size of the heap because it can go as far as the height of the heap which is $O(\log n)$. The recurrence relation can be expressed as $T(n) = T(2n/3) + O(1)$ a visual proof can be seen in the figure below that I drew, it comes from the fact that a heap is a nearly complete binary tree, so work can be divided unequally as in 2/3 of the tree to 1/3 worst case.

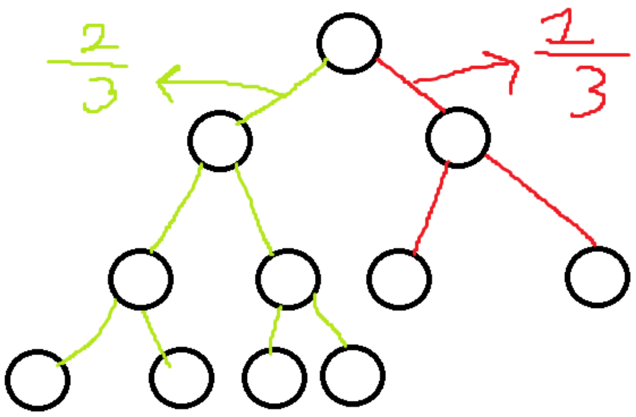

Figure 1.1: visual proof

The space complexity is $O(1)$ due to the fact that we aren't creating any new variables besides temporary ones which are negligible, and I am not considering stack calls as space usage, but if they were to be considered then it will be $O(\log n)$ because that's how many calls there are. The same goes for every other function in this report

All complexities were found using the master method.

1.2. BUILD-MAX-HEAP

1.2.1. Implementation Details

The BUILD-MAX-HEAP procedure aims to convert an unordered array into a max-heap. My implementation is provided below:

```
void build_max_heap(std::vector<Population> &array, const int &
    array_size) {
     for (int i = (array_size / 2) - 1; i >= 0; i--) {
          max_heapify(array, i, array_size);
     }
}
```

Listing 1.2: BUILD-MAX-HEAP Implementation

This implementation takes a vector of `Population` objects and the size of the array as parameters, building a max-heap from it.

The BUILD-MAX-HEAP process works as follows:

1. The loop iterates from the last internal node (array_size$/2-1$) to the root (0).
2. At each iteration, it applies the MAX-HEAPIFY procedure to ensure the current subtree rooted at the index maintains the max-heap property.
3. The result is a max-heap that covers the entire array.

1.2.2. Time, Space complexities and related recurrence relations

The time complexity of BUILD-MAX-HEAP is $O(n)$, where n is the size of the array. This linear time complexity arises because each call to MAX-HEAPIFY operates in $O(\log n)$ time, and there are $n/2$ calls made during the BUILD-MAX-HEAP process.

The space complexity is $O(1)$ since the algorithm operates in-place and does not use additional space.

The recurrence relation for the BUILD-MAX-HEAP process can be expressed as follows:

$$T(n) = \sum_{i=1}^{n/2} T(2i) + O(\log n)$$

This recurrence relation accounts for the fact that each call to MAX-HEAPIFY is $O(\log n)$, and there are $n/2$ calls which when solved gives $O(n)$.

In conclusion, BUILD-MAX-HEAP is a crucial step in preparing an array for efficient heap operations, and its time complexity is $O(n)$ with a space complexity of $O(1)$.

1.3. HEAPSORT

1.3.1. Implementation Details

The HEAPSORT procedure utilizes the max-heap data structure to efficiently sort an array in ascending order. My implementation is provided below:

```
void heapsort(std::vector<Population> &array, const int &array_size)
   {
    build_max_heap(array, array_size);
    for (int i = array_size - 1; i > 0; i--) {
        swap_array_value(array, 0, i);
        max_heapify(array, 0, i);
    }
}
```

Listing 1.3: HEAPSORT Implementation

This implementation takes a vector of `Population` objects and the size of the array as parameters, sorting the array in-place using the HEAPSORT algorithm.

The HEAPSORT process works as follows:

1. Build a max heap from the input array using the BUILD-MAX-HEAP procedure.

2. Iterate through the array in reverse, swapping the current root (maximum) element with the last element at each step.

3. After each swap, restore the max-heap property on the reduced heap (excluding the sorted elements).

1.3.2. Time, Space complexities and related recurrence relations

The time complexity of HEAPSORT is $O(n \log n)$, where n is the size of the array. This is because the build_max_heap operation takes $O(n)$ time, and the subsequent n max_heapify operations each take $O(\log n)$ time but is in a loop that runs n-1 times.

The space complexity is $O(1)$ since the algorithm operates in-place and does not use additional space.

The recurrence relation for the time complexity can be expressed as follows:

$$T(n) = O(n) + \sum_{i=1}^{n} O(\log i)$$

This recurrence relation considers the $O(n)$ time complexity of building the max heap and the $O(\log i)$ time complexity of each max_heapify operation. and by solving it we get $O(n \log n)$

In conclusion, HEAPSORT provides an efficient $O(n \log n)$ sorting algorithm based on the max-heap data structure.

1.4. MAX-HEAP-INSERT

1.4.1. Implementation Details

The MAX-HEAP-INSERT procedure inserts a new element into the max-heap and puts it in the correct position. My implementation is provided below:

```
void max_heap_insert(std::vector<Population> &array, const Population
    &key) {
    build_max_heap(array, array.size());
    array.push_back(Population{key.name, -1});

    int i = array.size() - 1;
    if (key.population < array[i].population) {
        std::cerr << "new key is smaller than current key" << std::
            endl;
        return;
    }
    array[i].population = key.population;
    while (i > 0 && array[(i - 1) / 2].population < array[i].
       population) {
        swap_array_value(array, i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}
```

Listing 1.4: MAX-HEAP-INSERT Implementation

This implementation takes a vector of `Population` objects and a key to insert. It ensures the array remains a max-heap after inserting the new element.

The MAX-HEAP-INSERT process works as follows:

1. Build a max heap from the input array using the BUILD-MAX-HEAP procedure since this operation requires a max heap.

2. Add a new element with the specified key to the end of the array, initially setting its population to -1 (representing negative infinity) in the pseudo code shown in class.

3. Update the population of the newly inserted element to the actual key value.

4. Ensure the max-heap property is maintained by adjusting the element's position in the heap. This would usually be done using a call to HEAP-INCREASE-KEY, but since it has to be a functional separate function too through the command line, I added a part of its code here instead of calling it separately.

1.4.2. Time, Space complexities and related recurrence relations

The time complexity of MAX-HEAP-INSERT is $O(\log n)$, where n is the size of the array if we ignore the building of the max heap process. This is because the operation involves inserting an element into the heap and then performing a logarithmic number of swaps to restore the max-heap property but creating the heap itself is of complexity $O(n)$.

The space complexity is $O(1)$ since the algorithm operates in-place and does not use additional space.

The recurrence relation for the MAX-HEAP-INSERT process can be expressed as follows:

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

This recurrence relation considers the time complexity of the while loop during the element insertion and the time complexity of the max_heapify operation.

In conclusion, MAX-HEAP-INSERT efficiently inserts a new element into a max-heap with a time complexity of $O(\log n)$ if we ignore the building process and a space complexity of $O(1)$.

1.5. HEAP-EXTRACT-MAX

1.5.1. Implementation Details

The HEAP-EXTRACT-MAX procedure removes and returns the maximum element from the max-heap. My implementation is provided below:

```
Population heap_extract_max(std::vector<Population> &array) {
    if (array.size() < 1) {
        std::cerr << "heap underflow" << std::endl;
        return Population{"", -1};
    }
    build_max_heap(array, array.size());
    Population max = array[0];
    array[0] = array[array.size() - 1];
    array.pop_back();
    max_heapify(array, 0, array.size() - 1);
    return max;
}
```

Listing 1.5: HEAP-EXTRACT-MAX Implementation

This implementation takes a vector of `Population` objects as a parameter and returns the maximum element while maintaining the max-heap property.

The HEAP-EXTRACT-MAX process works as follows:

1. Check for heap underflow (array size less than 1).

2. Build a max heap from the input array using the BUILD-MAX-HEAP procedure since it is required for this operation.

3. Extract the maximum element (located at the root).

4. Swap the root with the last element and remove the last element from the array.

5. Restore the max-heap property by applying the MAX-HEAPIFY procedure to the modified heap.

1.5.2. Time, Space complexities and related recurrence relations

The time complexity of HEAP-EXTRACT-MAX is $O(\log n)$, where n is the size of the array if we ignore the building heap procedure. This is due to the build_max_heap operation which takes $O(n)$ time.

The space complexity is $O(1)$ since the algorithm operates in-place and does not use additional space.

The recurrence relation for the HEAP-EXTRACT-MAX process can be expressed as follows:

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

This recurrence relation considers the time complexity of the build_max_heap operation and the max_heapify operation.

In conclusion, HEAP-EXTRACT-MAX removes and returns the maximum element from a max-heap with a time complexity of $O(\log n)$ if we ignore the building heap procedure which would raise the complexity to $O(n)$. the space complexity is $O(1)$.

1.6. HEAP-INCREASE-KEY

1.6.1. Implementation Details

The HEAP-INCREASE-KEY procedure increases the key of a specified element in the max-heap. My implementation is provided below:

```
void heap_increase_key(std::vector<Population> &array, int i, const
    int &key) {
     build_max_heap(array, array.size());
     if (key < array[i].population) {
          std::cerr << "new key is smaller than current key" << std::
              endl;
          return;
     }
     array[i].population = key;
     while (i > 0 && array[(i - 1) / 2].population < array[i].
         population) {
          swap_array_value(array, i, (i - 1) / 2);
          i = (i - 1) / 2;
     }
}
```

Listing 1.6: HEAP-INCREASE-KEY Implementation

This implementation takes a vector of `Population` objects, an index `i`, and a key as parameters. It increases the key of the specified element while maintaining the max-heap property.

The HEAP-INCREASE-KEY process works as follows:

1. Build a max heap from the input array using the BUILD-MAX-HEAP procedure.
2. Check if the new key is greater than the current key. If not, print an error message.
3. Update the key of the specified element and ensure the max-heap property is maintained by adjusting its position in the heap.

1.6.2. Time, Space complexities and related recurrence relations

The time complexity of HEAP-INCREASE-KEY is $O(\log n)$, where n is the size of the array if we ignore the fact that we are building the max heap at the start because it has a complexity of $O(n)$. This is because the main operation in HEAP-INCREASE-KEY involves increasing the key of a specific element and then performing a logarithmic number of swaps to restore the max-heap property.

The space complexity is $O(1)$ since the algorithm operates in-place and does not use additional space.

The recurrence relation for the HEAP-INCREASE-KEY process can be expressed as follows:

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

This recurrence relation considers the time complexity of the while loop during the key increase which can go as far as the height and the logarithmic time complexity of the max_heapify operation.

In conclusion, HEAP-INCREASE-KEY efficiently increases the key of a specified element in a max-heap with a time complexity of $O(\log n)$ if the building is ignored, and a space complexity of $O(1)$.

1.7. HEAP-MAXIMUM

1.7.1. Implementation Details

The HEAP-MAXIMUM procedure returns the maximum element from the max-heap without removing it. My implementation is provided below:

```
Population heap_maximum(std::vector<Population> &array) {
    if (array.size() < 1) {
        std::cout << "heap underflow" << std::endl;
        return Population{"", -1};
    }
    build_max_heap(array, array.size());
    Population max = array[0];
    return max;
}
```

Listing 1.7: HEAP-MAXIMUM Implementation

This implementation takes a vector of `Population` objects as a parameter and returns the maximum element without altering the max-heap structure.

The HEAP-MAXIMUM process works as follows:

1. Check for heap underflow (array size less than 1).

2. Build a max heap from the input array using the BUILD-MAX-HEAP procedure since this operation requires a max heap.

3. Return the maximum element (located at the root) without removing it.

1.7.2. Time, Space complexities and related recurrence relations

The time complexity of HEAP-MAXIMUM is $O(1)$ if we ignore the fact that we are calling build_max_heap since its a bit different that what this function actually aims to do. The maximum element is always at the root of the max-heap, and its retrieval does not involve any additional operations.

The space complexity is $O(1)$ since the algorithm operates in-place and does not use additional space.

In conclusion, HEAP-MAXIMUM efficiently returns the maximum element from a max-heap with a constant time complexity of $O(1)$ and a space complexity of $O(1)$.

1.8. D-ARY-CALCULATE-HEIGHT

1.8.1. Implementation Details

The D-ARY-CALCULATE-HEIGHT function calculates the height of a D-ary heap given the array size and the degree d. My implementation is provided below:

```
int dary_calculate_height(const int &array_size, const int &d) {
    return std::ceil(std::log(array_size) / std::log(d));
}
```

Listing 1.8: D-ARY-CALCULATE-HEIGHT Implementation

This implementation takes the array size and the degree d as parameters and returns the height of the corresponding D-ary heap.

The D-ARY-CALCULATE-HEIGHT process works by utilizing the logarithmic base-d formula to determine the height of the D-ary heap, I wrote this formula by looking at a normal heap with 2 children and generalising it to d children. Since we have 'level' to the power of d children in each level, taking the log of the number of values that we have with base d, fills the levels value by value so we get an integer, if not then its not a complete tree its nearly complete, but to count that layer to I ceil the value.

1.8.2. Time, Space complexities and related recurrence relations

The time complexity of D-ARY-CALCULATE-HEIGHT is $O(1)$ if you consider taking the log operation constant, or $O(\log n)$ if you consider the operation to be taking log n time if its done with a loop. In my code it can be seen that I implemented this function using the log function, but there is also another implementation using a loop that runs $O(\log n)$ times to find the height.

The space complexity is also $O(1)$ since the algorithm operates without using additional space.

In conclusion, D-ARY-CALCULATE-HEIGHT efficiently calculates the height of a D-ary heap with constant time complexity $O(1)$ and constant space complexity $O(1)$.

1.9. D-ARY-EXTRACT-MAX

1.9.1. Implementation Details

The D-ARY-EXTRACT-MAX procedure removes and returns the maximum element from the D-ary max-heap. My implementation is provided below:

```
Population dary_extract_max(std::vector<Population> &array, const int
    &d) {
    if (array.size() < 1) {
        std::cerr << "heap underflow" << std::endl;
        return Population{"", -1};
    }
    dary_build_max_heap(array, array.size(), d);
    Population max = array[0];
    array[0] = array[array.size() - 1];
    array.pop_back();
    dary_max_heapify(array, 0, array.size() - 1, d);
    return max;
}
```

Listing 1.9: D-ARY-EXTRACT-MAX Implementation

This implementation takes a vector of `Population` objects and the degree d as parameters and returns the maximum element from the D-ary max-heap.

The D-ARY-EXTRACT-MAX process works as follows:

1. Check for heap underflow (array size less than 1).

2. Build a D-ary max heap from the input array using the D-ARY-BUILD-MAX-HEAP procedure.

3. Return the maximum element (located at the root) after removing it.

It follows the same methodology as the normal HEAP-EXTRACT-MAX

1.9.2. Time, Space complexities and related recurrence relations

The time complexity of D-ARY-EXTRACT-MAX is $O(d \log_d n)$, where n is the size of the heap and d is the degree if we ignore the building procedure which is of complexity $O(n)$. The d before the log comes from how many children we need to compare to get the max

The recurrence relation for the HEAP-INCREASE-KEY process can be expressed as follows:

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

The space complexity is $O(1)$ since the algorithm operates in-place without using additional space.

In conclusion, D-ARY-EXTRACT-MAX efficiently removes and returns the maximum element from a D-ary max-heap with time complexity $O(d \log_d n)$ and space complexity $O(1)$ if we ignore the building procedure.

1.10. D-ARY-INSERT-ELEMENT

1.10.1. Implementation Details

The D-ARY-INSERT-ELEMENT procedure inserts a new element into the D-ary max-heap. My implementation is provided below:

```
void dary_insert_element(std::vector<Population> &array, const
    Population &key, const int &d) {
     dary_build_max_heap(array, array.size(), d);
     array.push_back(Population{key.name, -1});
     int i = array.size() - 1;
     if (key.population < array[i].population) {
          std::cerr << "new key is smaller than current key" << std::
              endl;
          return;
     }
     array[i].population = key.population;
     while (i > 0 && array[(i - 1) / d].population < array[i].
         population) {
          swap_array_value(array, i, (i - 1) / d);
          i = (i - 1) / d;
     }
}
```

Listing 1.10: D-ARY-INSERT-ELEMENT Implementation

This implementation takes a vector of `Population` objects, a key to insert, and the degree d as parameters.

The D-ARY-INSERT-ELEMENT process works as follows:

1. Build a D-ary max heap from the input array using the D-ARY-BUILD-MAX-HEAP procedure because this procedure requires a max heap.

2. Add the key to the end of the array with a population of -1 which acts as a -ve infinity as shown in the pseudo code in class.

3. Increase the key of the last element using D-ARY-INCREASE-KEY, which is handled within this function.

1.10.2. Time, Space complexities and related recurrence relations

The time complexity of D-ARY-INSERT-ELEMENT is $O(d \log_d n)$, where n is the size of the heap and d is the degree if we ignore the building procedure of the d-ary max heap. The recurrence relation involves the D-ary max-heap operations.

The recurrence relation for the HEAP-INCREASE-KEY process can be expressed as follows:

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

The space complexity is $O(1)$ since the algorithm operates in-place without using additional space.

In conclusion, D-ARY-INSERT-ELEMENT efficiently inserts a new element into a D-ary max-heap with time complexity $O(d \log_d n)$ and space complexity $O(1)$.

1.11. D-ARY-INCREASE-KEY

1.11.1. Implementation Details

The D-ARY-INCREASE-KEY procedure increases the key of a specified element in the D-ary max-heap. My implementation is provided below:

```
void dary_increase_key(std::vector<Population> &array, int i, const
    int &key, const int &d) {
     dary_build_max_heap(array, array.size(), d);
     if (key < array[i].population) {
          std::cerr << "new key is smaller than current key" << std::
              endl;
          return;
     }

     array[i].population = key;
     while (i > 0 && array[(i - 1) / d].population < array[i].
         population) {
          swap_array_value(array, i, (i - 1) / d);
          i = (i - 1) / d;
     }
}
```

Listing 1.11: D-ARY-INCREASE-KEY Implementation

This implementation takes a vector of `Population` objects, an index of the element to increase, a new key, and the degree d as parameters.

The D-ARY-INCREASE-KEY process works as follows:

1. Build a D-ary max heap from the input array using the D-ARY-BUILD-MAX-HEAP procedure.

2. Check if the new key is greater than the current key. If not, print an error message.

3. Update the key of the specified element.

4. Move up the heap, swapping elements if necessary, to maintain the D-ary max-heap property.

1.11.2. Time, Space complexities and related recurrence relations

The time complexity of D-ARY-INCREASE-KEY is $O(d \log_d n)$, where n is the size of the heap and d is the degree if we ignore the building max d-ary heap procedure which takes $O(n)$ complexity.

The recurrence relation for the HEAP-INCREASE-KEY process can be expressed as follows:

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

The space complexity is $O(1)$ since the algorithm operates in-place without using additional space.

In conclusion, D-ARY-INCREASE-KEY efficiently increases the key of a specified element in a D-ary max-heap with time complexity $O(d \log_d n)$ and space complexity $O(1)$.

1.12. Priority Queue Operation Real World Example

As per requested in the question sheet; Imagine already having a max-heap holding the data of multiple cities and their populations:

- **Inserting a New City:**
  A new city can be established with a population due to various factors such as border changes. To insert this new city into the existing max-heap, the `MAX-HEAP-INSERT` procedure from Section 1.4 can be utilized.

- **Cutting Down a Big City:**
  If a city becomes too large and needs to be divided into smaller cities for easier management, the procedure of extracting the maximum city from the heap and subsequently inserting the smaller cities would be practical. This operation is performed using the `HEAP-EXTRACT-MAX` procedure described in Section 1.5.

- **Proposal for Data Manipulation:**
  In the scenario of preparing a proposal to perform the extraction and insertion of cities without modifying the current data, the `HEAP-MAXIMUM` procedure outlined in Section 1.7 would be employed. This procedure retrieves the maximum element (city) from the heap without altering the underlying data.

- **Editing Increased Population:**
  If the population of a city increases and needs to be updated in the max-heap, the `HEAP-INCREASE-KEY` procedure from Section 1.6 is suitable. This procedure efficiently adjusts the key (population) of the specified city in the heap while maintaining the max-heap property.

1.13. Results

	Population1	**Population2**	**Population3**	**Population4**
Random Hybrid K=25	12890072 ns	8545700 ns	10519199 ns	10088399 ns
Heap Sort	14910697 ns	12412500 ns	13892698 ns	11650499 ns

Table 1.1: Comparison of different pivoting strategies on input data of quicksort with heapsort

Discussion:

From the table we can see that heapsort is slower than quicksort with random pivot and a hybrid switch to insertion sort at 25 elements throughout all population files.

That can be attributed to the fact that HeapSort has a higher constant factor in its time complexity, making it less efficient for smaller datasets. HeapSort's time complexity is generally $O(n \log n)$, but its constant factor can make it slower for practical purposes.

Quicksort, on the other hand, has a lower constant factor and performs well on average. The hybrid strategy of switching to insertion sort at a threshold of 25 elements might contribute to its efficiency for smaller datasets, as insertion sort is more efficient for small lists.

By constant factor I meant all other operations besides important funciton calls, so that includes swaps and comparisons .... etc. we see that in the heapsort code, there is more comparisons that in quicksort which is what lead to the bigger constant factor.

We also see that heapsort has a guaranteed $n \log n$ complexity, which basic quicksort doesn't provide unless we employ specific pivoting strategies and randomizations that decrease the probability of it giving us an n^2 complexity.

Scenario-Based Considerations:

Heapsort:

- Best for situations where ensuring worst-case performance is still good is crucial.
- Suitable for scenarios with tight memory constraints, as heapsort can be implemented with minimal additional memory usage.

Quicksort:

- Preferred in situations where average-case performance is a priority, and worst-case behavior is not as critical.
- Well-suited for scenarios where the input data is random or nearly random, as quicksort tends to perform better in such cases.

d (Number of children)	2	4	10	100	1000
dary_calculate_height (ns)	8460.0	7605.0	12330.0	8070.0	8535.0
dary_extract_max (ns)	852280.2	427205.2	236655.1	128030.0	113475.05
dary_insert_element (ns)	856115.25	426390.15	240170.05	120260.0	105590.0
dary_increase_key (ns)	867125.25	418140.05	231640.1	129605.05	105210.05

Table 1.2: Comparison of different d-ary heap operations on different d values using population 4.

For getting the table above, I wrote the following python script that would print the outputs directly into LaTeX that I can just copy paste:

```
import subprocess

d_values = [2, 4, 10, 100, 1000]
num_iterations = 20  # Number of times to repeat each command

commands = [
    "./Heapsort population4.csv dary_calculate_height height.csv d{}"
       ,
    "./Heapsort population4.csv dary_extract_max height.csv d{}",
    "./Heapsort population4.csv dary_insert_element height.csv
       k_ituland_721 d{}",
    "./Heapsort population4.csv dary_increase_key height.csv i721
       k999999 d{}"
]

results = {
    "dary_calculate_height": {},
    "dary_extract_max": {},
    "dary_insert_element": {},
    "dary_increase_key": {}
}

for command in commands:
    for d in d_values:
        total_time = 0
        for _ in range(num_iterations):
            full_command = command.format(d)
            output = subprocess.check_output(full_command, shell=True
               , text=True)
            time_taken = int(output.split()[-2])
            total_time += time_taken

        average_time = total_time / num_iterations
        results[command.split()[2]][d] = average_time

print("\\textbf{d (Number of children)} & " + " & ".join(map(str,
   d_values)) + " \\\\ \\hline")
for command, values in results.items():
    print("\\textbf{" + command + " (ns)} & " + " & ".join(map(str,
       values.values())) + " \\\\ \\hline")
```

Listing 1.12: Your Python Script

it runs each command 20 times and gets me the average.

As for its discussion;

We can see a global trend that with increasing d we decrease the time taken, this could be due to the fact that when we increase the number of children, we decrease the height of the heap, and since all of the operations depend on the height of the tree, decreasing it increases the speed of all of those operations.

Also by editing population4.csv and doubling its size, we get the table below

d (Number of children)	2	4	10	100	1000
dary_calculate_height (ns)	12705.0	10340.0	9465.0	10765.0	9765.0
dary_extract_max (ns)	1991970.2	1074110.1	461965.0	271175.0	275670.1
dary_insert_element (ns)	1834160.1	928270.1	494550.1	298480.05	243680.0
dary_increase_key (ns)	1759520.35	912315.15	459100.1	251045.0	204900.1

Table 1.3: doubling population4 size

We see that if we increase n "size of the heap/data" the time also increases.