Analysis of Algorithms

BLG 335E

Project 2 Report

Fatih Baskın
150210710
baskin21@itu.edu.tr

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

1. Implementation

1.1. Explaination of the Functions Implemented

1.1.1. Heap Fuctions

1.1.1.1. MAX-HEAPIFY

MAX-HEAPIFY function requires parent, left_child and right_child methods to work. Their implementation is shown below:

```
static int left_child(int idx) { return 2 * idx; }
static int right_child(int idx) { return (2 * idx) + 1; }
static int parent(int idx) { return idx / 2; }
```

Since this is a binary heap with 1 indexing, the parent is the floor of $index/2$, left child is $2 \times index$ and right child is $2 \times index + 1$. MAX-HEAPIFY function looks like:

```
void BinaryHeap::max_heapify(std::vector<City *> &vect,
                             int heap_size, int idx)
{
    if (idx >= heap_size)
        return;
    int largest = idx;
    int left = BinaryHeap::left_child(idx);
    if (left <= heap_size
     && vect[left]->mPopulation > vect[largest]->mPopulation)
        largest = left;
    int right = BinaryHeap::right_child(idx);
    if (right <= heap_size
     && vect[right]->mPopulation > vect[largest]->mPopulation)
        largest = right;
    if (largest != idx)
    {
        City *temp = vect[largest];
        vect[largest] = vect[idx];
        vect[idx] = temp;
        max_heapify(vect, heap_size, largest);
    }
}
```

MAX-HEAPIFY compares the current element with left child and right child and swaps the current element with the largest one, and continues to do comparisons recursively from its new position until it is a leaf. The complexity of this operation is $O(log)$ which is the height of the tree.

1.1.1.2. BUILD-MAX-HEAP

BUILD-MAX-HEAP uses the MAX-HEAPIFY procedure. It starts to heapify the array starting from the last parent, which is the floor of $n/2$ to the top. This ensures that subtrees are already heapified and the newly added nodes will not fail this property when they are heapified too.

At first glance, the complexity is $O(nlogn)$ but it is $O(n)$, since most of the heapify calls are near leaves. The math behind this looks like: $\Sigma_{h=0}^{logn} \frac{n}{2^(h+1)} = n \times \Sigma_{h=0}^{\inf} \frac{1}{2^h} = n \times 2 = O(n)$.

1.1.1.3. HEAPSORT

In HeapSort, the heap property is used to sort the elements. The top point is always the maximum value in a heap, so the maximum is put to the end of the array and the end of the array is put to the top. Since the end was a leaf node, which is close to being the minimum element of the array, heapify is called. Heapify's complexity would be $O(logn)$ since the top one is close to the minimum so the heapify procedure would work until the bottom, which would yield $O(logn)$ complexity. So, the complexity would be $\Sigma_{i=1}^{n} log(n-1)$ which is upper bounded by $O(nlogn)$

1.1.2. Priority Queue Functions

1.1.2.1. MAX-HEAP-INSERT

To insert an element to a max heap, the heap size is extended by one and the element is put at the end, then it is compared with its parent and swapped with it until the parent is larger than the new element or the new element is the new head. Since the new element can be the new head, the upper bound for the run time complexity is $O(h) = O(logn)$ where the h is the height.

Heap insert can be used in a CPU scheduler, more urgent tasks with higher priority values should be returned and coming tasks can have varying levels of urgency.

1.1.2.2. HEAP-EXTRACT-MAX

Extract max removes the top element from the heap array and returns it. It works by swapping the last element with the top element and resizing the heap so the old top is excluded from the heap array. Then the new top is very close to the minimum value since it was at the bottom of the heap, heapify procedure is called to restore the heap property. Since heapify is called one times, the complexity is upper bounded by $O(logn)$

To further exemplify the CPU scheduler, important tasks can be taken and when they are taken, they should be removed from the scheduler. This is an example of the extract-max.

1.1.2.3. HEAP-INCREASE-KEY

Incease key function takes a key and a value argument. The i^{th} element in the array is set to the new value if it is larger than the old value. Then, since it is increased, it is compared with its parents and swapped up until it is the new top or the new parent is larger than the current element. The runtime complexity is upper bounded by $O(logn)$ since the key can be the minimum element and the new value can make it the new top element. In this case, the function would work height times, which is the floor of $logn$.

To further exemplify the CPU scheduler, the increase key can be used to increase the urgency of a given task.

1.1.2.4. HEAP-MAXIMUM

Directly returns the top element of the heap array, which is indexed by 1. Run time complexity is $O(1)$.

Heap maximum can be used to compare the current task with the task at the top of the scheduler. If the top of the scheduler is more urgent, current task can be left and new task can be taken. This comparison can be made swiftly by using the heap-maximum method.

1.1.3. D-ary Heap Functions

1.1.3.1. DARY-CALCULATE-HEIGHT

The height of a d-ary heap can't be calculated using the logarithm directly, the height should be calculated mathematically. The inner details of this function is like:

```
int DaryHeap::dary_calculate_height(int dary_heap_size, int d)
{
    int height = 0;
    int limit = 1;
    int increment = d;
    while (limit < dary_heap_size)
    {
        limit += increment;
        increment *= d;
        height += 1;
    }
    return height;
}
```

Initially, the height is 0. The limit is the maximum number of elements that a given height can hold. While the limit is smaller than the number of elements, height is incremented. As height is incremented, the limit is increased by increment value, which is initially d and multiplied with d in each step.

Since this function works with a while loop, while loop can't be executed by more than h times. The height value upper bounds heap size since $d^(h+1) > heap\_size$ so the running time complexity is $log_d(n) + 1$ which is $O(log_d(n))$

1.1.3.2. DARY-EXTRACT-MAX

This function works similar to the extract max of a binary heap, the only caveat is since each node can have d children, it is necessary to calculate the indexes of children correctly. I have found a formula with my experimentation:

```
d = 5
1 -> 2, 3, 4, 5, 6
2 -> 7, 8, 9, 10, 11
3 -> 12, 13, 14, 15, 16
4 -> 17, 18, 19, 20, 21
5 -> 22, 23, 24, 25, 26
6 -> 27, 28, 29, 30, 31

d = 8
1 -> 2,3,4,5,6,7,8,9
2 -> 10,11,12,13,14,15,16,17
3 -> 18,19,20,21,22,23,24,25
4 -> 26,27,28,29,30,31,32,33
5 -> 34,35,36,37,38,39,40,41
6- > 42,43,44,45,46,47,48,49
7 -> 50,51,52,53,54,55,56,57
8 -> 58,59,60,61,62,63,64,65

childs = (p - 1)*d + 2, (p - 1)*d + 3, ..., (p - 1)*d + d + 1
parent = (c - 2)/d + 1
```

The listing above shows how I derived the formula for child nodes and parent nodes. Using this formula, I have rewritten the heapify function, which compares with d different values in the children array. The time complexity of this operation is $O(d \times log_d n)$ since there are d comparisons in each step and the height is upper bounded by ceil of $log_d n$.

1.1.3.3. DARY-INSERT-ELEMENT

Insert element works very similarly to binary heap's insert element, and with the method I used for children and parents, I do the insertion operation. First, I add element to the end of the array and then I swap with parent until the element is smaller than its parent or the top element. The time complexity of this operation is $O(log_d n)$ since the height is upper bounded by ceil of $log_d n$.

1.1.3.4. DARY-INCREASE-KEY

The method used here is very similar to binary heap's increase key, the only difference is in the calculation of the parent node. I use the parent strategy explained above to find the parents. First, the value at the key index is set to the value provided if the provided value is greater than the key's value. Then, since its value is increased, the key is moved upwards by swapping it with its parents until the parent is larger than the key or the key is the top of the heap. Again, time complexity of this operation is $O(log_d n)$ since the height is upper bounded by ceil of $log_d n$.

1.1.4. DARY-HEAPSORT

The dary-HeapSort works very similar to binary heap's HeapSort, the only difference being in the heapify procedure. Since the heapify is called n times and heapify's running time complexity is $O(d \times log_d n)$ then the complexity of dary-Heapsort is $O(d \times nlog_d n)$.

1.2. Runtime Difference Between QuickSort and HeapSort

	Population1	**Population2**	**Population3**	**Population4**
Randomized QuickSort	3799397	3810187	4019540	4095764
HeapSort	8021385	7953106	7024836	9701198
4ary-Heapsort	113264273	116604860	106797928	128324470

Table 1.1: Comparison of the running time of QuickSort and HeapSort in terms of nanoseconds.

The average time complexity of the randomized HeapSort is $O(nlogn)$ as previously discussed in the previous homework. The time complexity of the HeapSort is also $O(nlogn)$ since it consists of putting the heap's maximum (top) to the end of the array and repeating max-heapify procedure over and over again. HeapSort will call max-heapify for each element in the heap, n times, and max-heapify procedure takes $O(logn)$ time since it does comparisons for left and right children at max height times. Therefore the running complexity of HeapSort is $\Sigma_{i=1}^{n} c_1 \times log(n-i) + c_2$ which is upper bounded by $O(nlogn)$.

At first glance, QuickSort and HeapSort have the same complexity, but often case, HeapSort works slower than the QuickSort, as shown in the experiment table. In our case, HeapSort worked approximately two times slower than QuickSort. This is probably due to binary HeapSort requiring two comparisons (left and right child) for each swap operation, unlike where QuickSort's swap operations only occur while finding the partition index, and it happens only n times for each depth.

To simplify the calculations of the number of comparisons, I will assume that max-heapify procedure works $logn$ times and partition divides the array perfectly into two in every step. So, the number of comparisons would be for HeapSort = $\Sigma\Sigma_{i=1}^{n} 2 \times log(n)$ and for the QuickSort, it would be $\Sigma_{i=0}^{logn-1} 2^i \times \frac{n}{2^i}$.

If we calculate for $n = 64$, number of comparisons for QuickSort would be $1 \times 64 + 2 \times \frac{64}{2} + 4 \times \frac{64}{4} + 8 \times \frac{64}{8} + 16 \times \frac{64}{16} + 32 \times \frac{64}{32} = 64 + 64 + 64 + 64 + 64 + 64 = 384$ and for HeapSort, the calculation would be $64 \times 2 \times log64 = 128 \times 6 = 768$ which coincides with the results of our experimentation.

There is also one hurdle that HeapSort has, which is the build-max-heap procedure. It has $O(n)$ complexity, and it's overhead is significant in small sample sizes.

Even though QuickSort has clear advantages as a result of our experiments, HeapSort can outperform QuickSort in cases where there are a lot of repetitions. If there are a lot of number repetitions, then randomized QuickSort's partition will work very poorly and its time complexity will be $O(n^2)$ in such case. Even the randomization optimization would not work in such cases. But HeapSort is consistent with its working time and it will work in $O(nlogn)$ time, but probably faster since most of the values would be the same and max-heapify would not work deeper in the heap tree.

To conclude, QuickSort works fast with a randomized input and a low number of repetitions. If the array is pre-sorted, quickSort works slowly and this is mitigated by using randomized pivoting. But randomization can't optimize the case of a high number of repeated values, where partition divides the array very poorly. HeapSort on the other hand, on average works slower compared to randomized QuickSort, but it works faster than QuickSort in the case of a high number of repeated values.

And finally, there is a huge difference in the running time of D-ary HeapSort and regular HeapSort, this difference is probably due to the overhead from calculating the d children of a node and passing them by value.