HeapSort and Data Structure Design

Suhas Arehalli

COMP221 - Spring 2024

1 Data Structure Design

- New design principle!
- **Idea:** Take a slow, inuitive algorithm and replace (or design!) a data structure that optimizes the speed of core operations.
- This is what COMP128 was all about!
  - Common data structure patterns that optimize common operations on collections!
  - i.e., min-heaps optimize repeatedly finding the minimum element of a collection.

2 HeapSort

Algorithm 1 Pseudocode for SelectionSort

```
function SelectionSort(Array A)
    for i ← 1 . . . N do
        for j ← i + 1 . . . N do
            if A[j] < A[i] then
                Swap(A[i], A[j])
            end if
        end for
    end for
end function
```

- To get to HeapSort, it helps to start with SelectionSort (Alg. 1).
- First, it helps to identify what the inner loop of selection is doing: Finding the minimum values in the unsorted part of the array ($A[i \dots N]$) and placing it at $A[i]$ each iteration. We can make this clearer by factoring out the minimum-finding part of the algorithm, resulting in Alg. 2.
- Note that findMin here is a linear algorithm to find the minimum element in a shrinking sub-array. This is an $\Theta(n)$ operation that finds the minimum over a shrinking subarray!

Algorithm 2 Pseudocode for SelectionSort, with findMin factored out

```
function SelectionSort(Array A)
    for i ← 1 . . . N do
        min ← i − 1 + findMin(A[i . . . N])
        Swap(A[i], A[min])
    end for
end function
function findMin(Array A)
    minIndex ← i
    for j ← i + 1 . . . N do
        if A[j] < A[minIndex] then
            minIndex ← j
        end if
    end for
end function
```

  - $A[i \dots N]$ is our collection, and we move the minimum element to $A[i]$ so that it's not part of the subarray on the next iteration.
  - Thus, we remove the minimum element from our collection of unsorted elements and place it at $A[i]$ at each step.

- This is exactly the use case that min-heaps were developed for! Repeatedly removing the minimum element from a collection! Min-heaps optimize this operation to be $\Theta(\log n)$, rather than $\Theta(n)$ like findMin
- This gets us to HeapSort!

Algorithm 3 Pseudocode for HeapSort

```
function HeapSort(Array A)
    Let h be a min-heap
    for i ← 1 . . . N do
        h.insert(A[i])
    end for
    for i ← 1 . . . N do
        A[i] ← h.removeMin()
    end for
end function
```

3 Proof of Correctness

Our proof of correctness is blessedly straightforward, particularly if we are familiar with the proof of correctness for SelectionSort!

We begin with a loop invariant for the second for-loop: *After the iteration where $i = k$, $A[1 \dots k]$ is sorted AND that all elements in $A[1 \dots k]$ are less than or equal to elements in h.*

Base Case: Before the first iteration, when $i = 0$, We must show that $A[1 \dots 0]$ is sorted and that elements in $A[1 \dots 0]$ are less than or equal to elements in h. Since $A[1 \dots 0] = \emptyset$ contains no elements, it's by definition sorted and all 0 of it's elements are smaller than elements in h. We're done!

Inductive Step: By our inductive hypothesis, We can assume after the iteration where $i = k$, $A[1 \dots k]$ is sorted and that elements in $A[1 \dots k]$ are less than or equal to elements in h

After the iteration where $i = k + 1$, we need to show that $A[1 \dots k + 1]$ is sorted, and that all elements in $A[1 \dots k + 1]$ are less than or equal to elements in h. Let's start working at it

We can immediately see that each iteration of the loop only changes $A[i]$; Every other position of the array is untouched. We can also note that no new elements are added to h. Thus, elements in $A[1 \dots k]$ remain less than elements in h, and so for the second half of our loop invariant, we only need to show that $A[k + 1]$ is less than or equal to elements that remain in the heap. This follows directly from the correctness of our REMOVEMIN operation for our heap — REMOVEMIN is meant to return exactly the smallest element in the heap!

What remains is showing that $A[1 \dots k + 1]$ is sorted. Again, $A[1 \dots k]$ is untouched and began sorted, so it remains sorted. We also know from our inductive hypothesis that since $A[k] \in A[1 \dots k]$, $A[k]$ is less than or equal to every element that began the iteration in h. We then note that since $A[k]$ is assigned the return value of REMOVEMIN, the value at $A[k + 1]$ began the iteration in h! With those two facts together, we know that $A[k] \leq A[k+1]$. This, combined with our observation that $A[1 \dots k]$ is sorted, will let us conclude that $A[1 \dots k + 1]$.

We now showed that after the $i = k + 1$ iteration, $A[1 \dots k + 1]$ sorted and that elements in $A[1 \dots k + 1]$ are less than or equal to elements in h. That concludes the proof.

4 Runtime Analysis

Here, I'll present a coarse analysis of worst-case time complexity and let you fill in the formal details. The first loop calls INSERT N times, and we recall that INSERT for heaps is a $\Theta(\log N)$ operation, making the loop $\Theta(N \log N)$. Similarly, the second loop calls REMOVEMIN N times, and since REMOVEMIN is $\Theta(\log N)$, the loop runs in $\Theta(N \log N)$ time. Thus, the entire function runs in $\Theta(N \log N)$ time!

Here, it might be useful to note that Skiena presents us with a faster way of constructing a heap with the elements of A: We can do it in $\Theta(N)$ time instead of $\Theta(N \log N)$! However, this won't reduce the asymptotic time complexity of HEAPSORT, since the time complexity of the second for-loop will dominate the time complexity of building the heap.