Mhammed Alhayek

RUID: 156-00-6919

Programming Methodology Homework 5

Question 1: code submitted on Sakai.

Question 2:

1. An efficient algorithm for combining two heaps, one with size n and the other with size 1:

If heap1 is size n and heap2 is size 1, the algorithm could simply be peeking at the top of heap2 and adding that value to heap 1. The pseudo code is as follows:

heapCombine(heap1: heap, heap2: heap) : heap{

heap1.add(heap2.peekTop())

return heap1;

}

The efficiency of this algorithm is going to be based off the efficiency of the add() function and the peekTop() function. The add function in the worst case scenario has to swap array entries on a path from a leaf to a root, therefore it makes at most ceil(log2(n+1)) swaps, which is O(logn) efficiency. The peekTop() function is O(1), therefore the efficiency of heapCombine is simply O(logn).

1. An efficient algorithm for combining two heaps of size n is:

Create a new array or size 2n and copy the contents of the arrays of the two heaps to that array. Then create a new heap using the new array to build the heap. To build the new heap using the new array, you must call the heapCreate() function, which calls the heapRebuild(int index) function using the index for ((2n-1)/2) all the way down to 0.

The first calls to heap rebuild will make at most 1 swaps. This will be when the index passed is at level (h-1). The next calls at level (h-2) will make at most 2 swaps, then the calls at level (h-3) at most 3 swaps, until finally the last calls will make at most log(2n) swaps (2n because it is two heaps of size n). The algorithm will make roughly (1 + 2 + 3 + … + log(2n) ) moves, which is roughly 2n. Also, copying the two heaps of size n is O(2n) complexity. Thus the efficiency of this algorithm is O(2n + 2n) = O(n) complexity.

The pseudo code is as follows:

heapCombine(heap1: heap, heap2: heap){

Create an array of size 2\*n

Copy contents of both heaps to this array

combinedHeap = ArrayMaxHeap(arr, 2\*n)

return combinedHeap

}

The constructor ArrayMaxHeap(arr,2\*n) uses the heapCreate() function on the array arr to create the heap, as explained above.

1. An efficient algorithm for combining two heaps of arbitrary sizes n and m is: Create a new array or size n + m and copy the contents of the arrays of the two heaps to that array. Then create a new heap using the new array to build the heap.

Based off the explanation in part b, the efficiency of this algorithm would be O((n+m) + (n+m)) = O(n+m). If n>m, this could be rewritten as O(n).

The pseudo code is as follows:

heapCombine(heap1: heap, n: int, heap2: heap, m:int){

Create an array of size n+m

Copy contents of both heaps to this array

Copy contents of both heaps to this array

combinedHeap = ArrayMaxHeap(arr, n+m)

return combinedHeap

}

The constructor ArrayMaxHeap(arr,2\*n) uses the heapCreate() function on the array arr to create the heap, as explained above.

1. implementation submitted on sakai.

Question 3: Code Submitted on Sakai.

Unsorted Inputs Data:

|  |  |
| --- | --- |
| Unsorted Inputs | |
| Number of Elements | HeapSort Run Time |
| 4 | 39 |
| 16 | 352 |
| 64 | 2302 |
| 256 | 12609 |
| 1024 | 64832 |
| 4096 | 316241 |
| 65536 | 6896675 |
| 1048576 | 139686014 |
| 16777216 | 2704214817 |

Plot:

Plot is on a log-log graph with a base of 2.0.

Analysis:

Based off the data above and the plot, it can be found that the complexity of the Heap Sort algorithm I wrote is roughly O(nlog(n)). This algorithm is significantly faster than the algorithms for selection, insertion, or bubble sort. The heap sort algorithm was able to handle the arrays of dus-20 and dus-24, which had over one million elements each. This shows that the algorithm is very practical and is ok to use even for large inputs. The algorithms runtime is very similar to that of merge sort, however, it doesn’t use the auxiliary array, making it more space efficient and thus preferred over merge sort. Overall however, Quick Sort is a little faster and is usually the preferred sorting algorithm.

Sorted Inputs Data:

|  |  |
| --- | --- |
| Sorted Inputs | |
| Number of Elements | Heap Sort Run Time |
| 4 | 43 |
| 16 | 367 |
| 64 | 2400 |
| 256 | 13365 |
| 1024 | 68171 |
| 4096 | 330900 |
| 65536 | 6489444 |
| 1048576 | 126902988 |
| 16777216 | 2412298523 |

Plot: plot is on a log-log graph with a base of 2.0.

Analysis: Even with a sorted input, the heap sort algorithm’s efficiency is about the same as with an unsorted input. Heap Sorts worst case and average case efficiency is the same, O(nlogn). It will take roughly the same amount of time for heap sort to sort an array, regardless of the original order of elements in the array.