Assignment 3 - Sorting: Putting Your Affairs in Order

Teresa Joseph

CSE 13S - Professor Long

Fall 2021 - RD October 14/ FD October 17

Purpose

Assignment 3 is the implementation of several sorting algorithms. Four kinds will be implemented in this assignment: insertion sort, heap sort, shell sort, and quick sort. These algorithms will be implemented in files called insert.c, heap.c, shell.c, and quick.c respectively. One other file called sorting.c will also be made to test these files with (any possible combination of) command line arguments.

Breakdown of Algorithms/Files

Insertion

- Start with an array of size n
- For each element k in 1 <= k <= n, compare k to k-1 to see if it's in the right position
    - If k > k-1, then k is in the right position and we move to k+1
    - If k < k-1, then swap k with k-1 and compare the original k to k-2

Heap

- Heap is represented by array
    - For any index k, the left child is 2k and the right child is 2k+1
    - Parent of index k should be k/2
- To implement, 1) build heap and 2) fix heap
    - Building heap must be a max heap (parents are greater than or equal to children) -> the largest element/roof of heap is the first element of the array
    - Fixing heap involves removing the largest array elements from top of heap and placing them at the end of the sorted array (final is in increasing order)

Shell

- Start with an array of size n
- Distance between pairs of elements is called a gap
- Each iteration decreases the gap until the gap = 1 (at this point, elements are sorted)
- Similar to insertion once once gap is incorporated (using comparisons and swaps)

Quick

- Partition array into 2 sub-arrays by selecting an element from the array and designating it as a pivot
    - Elements less than the pivot go in the left sub-array and elements greater than or equal to the pivot go in the right sub-array -> use subroutine partition() for this that returns the index that indicates the division

Sorting

- Final test harness
- Create array of pseudorandom elements to test each of the sorts
- Must support any argument combination a, e, i, s, q, r, n, p, and h by using a set to track

Pseudocode

Insertion

```
define function insertion_sort(initialize stats, list A, and uint32_t n):
    for i in A until len of A (basically n):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j-1]:
            A[j] = A[j-1]
            j -= 1
        A[j] = temp
```

Heap

```
define function finding_max_child(list A, uint32_t first, uint32_t last):
    left = first*2
    right = left+1
    if right <= left and A[right-1] > A[left-1] > A[left-1]:
        return right
    else, return left
define function fix_heap(list A, uint32_t first, uint32_t last):
    found = false
    mother = first
```

```
        greatest = finding_max_child(A, mother, last)
        while (mother <= last) // 2 and not found:
                if A[mother-1] < A[greatest-1]:
                        temp = A[mother-1]
                        A[mother-1] = A[greatest-1]
                        A[greatest-1] = temp
                        mother = greatest
                        greatest = finding_max_child(A, parent, last)
                else, found = true
define function build_heap(list A, uint32_t first, uint32_t last):
        for father starting from last//2, until first-1, decreasing father by 1:
                fix_heap(A, father, last)
define function heap_sort(list A):
        first =1
        last = length of A
        build_heap(A, first, last)
        for left starting from last, until first, decreasing left by 1:
                temp = A[first-1]
                A[first-1] = A[leaf-1]
                A[leaf-1] = temp
                fix_heap(A, first, leaf-1)
```

Shell

```
define function gap(uint32_t n):
        for i starting from 0, to log(3+2*n)/log(3), decreasing i by 1:
                return (3**i-1) // 2
define function shell_sort(list A):
        for g in gaps(range of A):
                for i starting from g to range of A:
                        j = i
                        temp = A[i]
                        while j >= g and temp < A[j-g]:
```

```
                A[j] = A[j-g]
                j -= g
            A[j] = temp
```

Quick

```
define function partition(list A, uint32_t low, uint32_t high):
    i = low-1
    for j starting with low, until high, incrementing by 1:
        if A[j-1] < A[high-1]:
            i += 1
            temp = A[i-1]
            A[i-1] = A[j-1]
            A[j-1] = temp
    temp2 = A[i]
    A[i] = A[high-1]
    A[high-1] = temp2
    return i+1
define quick_sorter(list A, uint32_t low, uint32_t high):
    if low < high:
        p = partition(A, low, high))
        quick_sorter(A, low, p-1)
        quick_sorter(A, p+1, high)
def quick_sort(list A):
    quick_sorter(A, 1, length of A)
```

Sorting

```
enumerate Sorts {INSERTION, HEAP, SHELL, QUICK}
s = empty set
while option arguments are not -1:
    check each case (a, e, i, s, q, r, n, p, and h)
    for each case, insert corresponding Sorts to s, then break
if a member of Sorts is in s, then enable that member's corresponding task
    if a, employ all algorithms
```

if e, enable heap sort

if i, enable insertion sort

if s, enable shell sort

if q, enable quick sort

if r seed, set random seed to seed (default = 13371453)

if n size, set array size to size (default = 100)

if p elements, print # of elements from array (default = 100)

    if size < specified # of elements, then print entire array and nothing more

if h, print program usage

Notes

- Pseudocode follows the assignment document for the most part
- sorting.c will have all header files from the resources repository
- heap.c and quick.c algorithms make use of 1-base indexing (actual code will be done in 0-base indexing which is why "-1" is used)
- Length of arrays can be determined using sizeof
- uint32_t is used instead of int to as so account for all possible ints

Overall Description

The files insert.c, heap.c, shell.c, and quick.c will contain functions and helper functions that conduct their respective sorts as described above. These files heavily rely on the pseudocode provided in the assignment document as of now, but changes will likely be made as seen fit. sorting.c will be similar to mathlib-test.c from the previous assignment but will make use of sets instead of bools/flags.

Goals/Intended Process

- Replicate the described pseudocode for each file in C
- Make code more readable and efficient if possible
- Add sufficient comments and clean up format