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. This assignment will act as a means of comparing each kind of sort and their efficiencies.

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
        use move so that temp = A[i]
        use comparison so that while j > 0 and temp < A[j-1]:
            move A[j-1] to A[j]
            Decrease j by 1
        move temp to A[j]
```

Heap

```
define function finding_max_child(stats, list A, uint32_t first, uint32_t last):
    left = first*2
    right = left+1
    if right <= left and comparison of A[right-1] > A[left-1] is > 0:
        return right
    else, return left
define function fix_heap(stats, list A, uint32_t first, uint32_t last):
    found = false
```

```
        mother = first
        greatest = finding_max_child(stats, A, mother, last)
        while (mother <= last) // 2 and not found (aka, not false):
                if comparison of A[mother-1] and A[greatest-1] < 0:
                        swap A[mother-1] and A[greatest-1]
                        mother = greatest
                        greatest = finding_max_child(stats, A, parent, last)
                else, found = true
define function build_heap(stats, list A, uint32_t first, uint32_t last):
        for father starting from last//2, until first-1, decreasing father by 1:
                fix_heap(stats, A, father, last)
define function heap_sort(stats, list A, int n):
        first =1
        last = n
        build_heap(stats, A, first, last)
        for left starting from last until first, decreasing left by 1:
                swap A[first-1] and A[leaf-1]
                fix_heap(stats, A, first, leaf-1)
```

Shell

```
create static/global variables
        i = 0
        gap_array[32] (the biggest it can be with uint32_t values)
        array_size
define function gaps(int n):
        create variable range in terms of an int = log(3+2*n)/log(3)
        create index counter = 0
        for i starting from range, to 0, decreasing i by 1:
                gap_array[increase index by 1] = (3**i-1) // 2 (this will yield different values with
                every call)
define function shell_sort(stats, list A, int n):
        call gaps(n)
```

```
    for g starting from 0 to array_size, increasing g by 1:
        gap = gap_array[g]
        for i starting from gap to range of n, increasing i by 1:
            j = i
            move A[i] to temp
            use comparison so that while j >= g and temp < A[j-g]:
                move A[j-g] to A[j]
                decrease j by gap
            move temp to A[j]
```

Quick

```
define function partition(stats, list A, int low, int  high):
    i = low-1
    for j starting with low, until high, incrementing by 1:
        if comparison of A[j-1] and A[high-1] < 0:
            i += 1
            swap temp and A[i-1]
    swap A[i-1] and A[high-1]
    return i+1
define quick_sorter(stats, list A, uint32_t low, uint32_t high):
    if low < high:
        call partition(stats, A, low, high) as p
        use recursion so that quick_sorter(stats, A, low, p-1)
        use recursion so that quick_sorter(stats, A, p+1, high)
def quick_sort(stats, list A, int n):
    call quick_sorter(stats, A, 1, n)
```

Sorting

include all header files

define options for the command line arguments

define each bit to represent each of the four sort types

define static variables for seed, array size, and print_size (r, n, and p respectively)

define function array

```
        create pointer = malloc(size of array * sizeof the 32-bit int)
        set the random seed
        for every iter in the size of the array:
                (pointer+iter) = random() and max of 30 bit hex
        return pointer
define function random elements (for the sorts)
        set the random seed
        for every iter in the size of the array:
                (pointer+iter) = random() and max of 30 bit hex
        return pointer
define function print_results for printing statistics
        print name, elements, moves, and compares
        check that print_size is not greater than size (if so, set print_size to size)
        for iters in print_size, increasing iters by 1:
                print element of that iter according to pointer
main function(argc, argv):
        s = empty set
        create Stats for each of the four sort types
        opt = 0
        use getopt with options defined earlier:
                use switch to check each option (a, e, i, s, q, r, n, p, and h)
                for a, e, i, s, and q: use insert_set to add corresponding bit that represents the sort
                for r, n, and p: set seed, size, and print_size to their respective opt arguments
                for h: print the program usage
        create test_pattern pointer
        set test_pattern = malloc(size of array * sizeof the 32-bit int) (from earlier)
        [note: if test_pattern is null, it is invalid and should return -1]
        check if member_set has each sort's corresponding bit, and if so:
                use random function with test_pattern, size, and seed
                use rest(address of sort's Stats)
                all (sort name)_sort function with address of sort's Stats, test_pattern, and size
```

use print_result with address of sort's Stats, name of the sort, and test_pattern

free test_pattern (takes care of malloc)

return 0

Command Line Options

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
- All four sorting algorithms will use <stdio.h> and "stats.h"
    - heap.c will also include <stdbool.h> for setting the boolean found
    - shell.c will also include <math.h> for calculating log
- The length of arrays will be n
- 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)
- 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 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, but changes were made as seen fit. sorting.c is somewhat similar to mathlib-test.c from the previous assignment, but makes use of malloc and sets instead of booleans.

Other (visualized sorts with examples)