Deeraj Gurram
dgurram@ucsc.edu
14 October 2021

CSE 13S Fall 2021
Assignment 3 - Sorting: Putting your affairs in order
Design Document

Description of the Program:

This assignment uses 4 separate programs to implement different sorting methods. Insertion sort is the first program, and compares each element to the previous element to see if it is in the correct spot or not. Shell sort compares elements that are spread apart by some gap length and puts the elements in the correct order, until the gap length is 1. Our heapsort will use a max heap where the parent node must have a value that is greater than or equal to the value of its children. After an array is ordered from largest value to smallest, the second routine will remove elements from the top of the heap and place it at the end of the sorted array. The last program is the quicksort, which partitions the array into 2 sub-arrays. Elements that are less than a chosen value from the array called the pivot are placed in one sub-array, and the rest are placed in the other sub-array. Then the quicksort is called recursively on each sub-array until the array is sorted. This assignment also includes a self-created test harness called sorting.c which serves as the code tester, and supports multiple command line options to run tests, input values for variables, and specify element sizes.

Deliverables:

1. insert.c
    - This file contains the function insertion_sort() which sorts an array using the insertion sort algorithm. It also returns the number of moves and compares that were performed on the array.
2. insert.h
    - Specifies the interface to insert.c
3. shell.c
    - This file contains the function shell_sort() which sorts an array using the shell sort algorithm. It also returns the number of moves, compares and swaps performed on the array.
4. shell.h
    - Specifies the interface to shell.c
5. heap.c
    - This file contains the function heap_sort() which sorts an array using the heap sort algorithm. It also returns the number of moves, compares, and swaps performed on the array.
6. heap.h

    - Specifies the interface to heap.c
7. quick.c
    - This file contains the function quick_sort() which sorts an array using the quick sort algorithm. It also returns the number of moves, compares, and swaps performed on the array.
8. quick.h
    - Specifies the interface to quick.c
9. set.h
    - Implements and specifies the interface for the set ADT
10. stats.c
    - Implements the statistics module
11. stats.h
    - Specifies the interface to the statistics module
12. sorting.c
    - Contains the main() function. This file is the testing harness which runs all the aforementioned sorts and may contain other functions necessary to complete the assignment.
13. Makefile
    - This file clang formats all program files, builds all executables, and formats all source code files.
14. README.md
    - This file describes how to use the program as well as the Makefile. It also explains the command line options that the program accepts.
15. DESIGN.pdf
    - This program describes the assignment and all required program files. It also contains a short description of each deliverable, an explanation on how the program works with supporting pseudocode, and a section that gives credit to all sources I received help from.
16. WRITEUP.pdf
    - This file includes an explanation on what I learned from the different sorting algorithms, an analysis on which conditions the sorts performed well under, graphs explaining the performance of sorts on a variety of inputs, and an examination of each graph that is produced.

Top Level Design/ Pseudocode:

insert.c

Define the function insertion_sort() with arguments stats, the array, and length of the array

    Initialize a variable j to hold a copy of the current iteration of the loop

Initialize a variable temp to hold the array's current element

Create for loop to go through and sort A, loop from 1 to the length of the array and increment in steps of 1

Set j equal to the for loop iterator (Serves to reset the variable j for the while loop)

Set temp equal to the current array element

Create while loop to compare the current element with the preceding one, make sure j is greater than 0, and the current element's value is less than the preceding element's value

Set the current element equal to the preceding element (this allows the move to occur)

Decrement j by 1 to compare the preceding array elements (to see if the current value should continue to be moved down)

Set the most preceding element equal to temp which holds the smaller value (this places the smallest elements in the correct position)

shell.c

Import the log function

Define the function shell_sort() with arguments stats, the array A, and length of the array

Initialize a variable j to manipulate the main iterator without actually changing its value

Initialize a variable temp to store the current element's value

Create a for loop to loop through each element in the array gap_length, loop from 0 to the length of this array, increment in steps of 1

Create for loop to loop through the current element, and end on the length of A

Set j equal to the iterator

Set temp equal to the current element

Create while loop to go through the elements of A and swap the values of the gap separated elements, make sure j is greater than or equal to gap and temp is less than the element that is gap length before the current element

Set the current element of A equal to the element that is gap length before A[j]

Decrement j by the current element of array gap_length

Set A[j] equal to A[i], which effectively swaps the values that were separated by the current value of gap_length

Define the function gaps() with the argument n (an integer that contributes to the range)

Create an array called gap_length of size (log(3 + 2n) / log(3)) to contain the gap length of each run through of the for loop

Create a for loop that creates the gap length for each run through of shell_sort, The loop should start at the value decided by the formula (log(3 + 2n) / log(3)), stop the loop before the iterator reaches 0, and decrement the iterator after each loop run

Create if statement and make sure iterator is greater than 0

Define a variable j (This variable is a copy of the iterator, but it takes care of the exponential portion of the next part)

Set j equal to j times i (This takes care of the 3^i)

Append the gap length of this iteration with the formula (((3*j) - 1) / 2) to the new array gap_length

heap.c

Define a function max_heap() with arguments array A, the first node, and the last node

Initialize a variable called left to hold the left child's value and set it equal to 2 * first

Initialize a variable called right to hold the right child's value and set it equal to left + 1

Check if right is less than or equal to last and A[right - 1] is greater than A[left - 1]

Return the variable right

This if statement allows us to return the largest values and place them at the top of the heap

Return the variable left, left is returned only if the right side is less than A[left - 1]

Define a function fix_heap() with arguments array A, the first node, and the last node

Initialize a boolean found and set it equal to false

Initialize a variable mother and set it equal to argument first

Initialize a variable great and set it equal to function max_child(A, mother, last)

Create while loop to swap elements in the tree based on if one child is greater than the other, make sure mother is less than or equal to last divided by 2, and that found is true

Check if A[mother -1] is less than A[great -1]

If this is true, then swap the values of A[mother -1] and A[great -1]

Set mother equal to great

Set great equal to max_child(A, mother, last) this checks the next children in the array A

Create else statement

Set found equal to true if if statement condition is not met

Define a function build_heap() with arguments array A, the first node, and the last node

Create a for loop that loops over the latter half of A, the loop should start at argument last divided by 2, and stop when the iterator is at first - 1, with the iterator decrementing in steps of 1

Call the function fix_heap() on father to build this part of the heap, the arguments passed should be A, father, last

Define a function heap_sort() to sort the array A, it should take the argument array A

Set the argument first equal to 1

Set the variable last equal to the length of A

Call the function build_heap with arguments A, first, last

Create for loop to swap the elements of A sort it in increasing order, the loop should iterate from last, until the iterator reaches first, and decrements in steps of 1

Swap the values of A[first - 1] with A[A[iterator] - 1]

Call the function fix_heap with arguments A, first, leaf - 1

quick.c

Define a function partition() to go through the array A and swap the values until they're in increasing order, this function takes the arguments array A, the lo integer, and the hi integer

Set i equal to lo - 1

Create a for loop to iterate from the argument lo until the iterator reaches hi, in steps of 1

Check if A[iterator - 1] is less than A[hi - 1]

Increment i by 1

Swap the values of A[i - 1] and A[iterator - 1]

Swap the values of A[i] and A[hi - 1]

Return i + 1

Define a function quick_sorter() to recursively go through the array A and partition each sub-array into smaller sub-arrays until A is in increasing order, this function takes the arguments array A, integer lo, and integer hi

Check if lo is less than hi

Create a new array p and set it equal to partition(A, lo, hi)

Run quick_sorter() recursively with arguments A, lo, p - 1, this will split the smaller sub-array into more sub-arrays that are increasing in order

Run quick_sorter() recursively with arguments A, p + 1, hi, this will split the larger sub-array into more sub-arrays that are increasing in order

Define a function quick_sort() with the argument array A

Run the function quick_sorter() with arguments A, 1, and length of A

sorting.c

Note: I haven't implemented the switch method for sorting.c yet, so I don't know how to run the tests for most of these cases

Include the OPTIONS "aeisqrsph" to run the various test cases

Create function main() with arguments argc, **argv

Initialize the variable opt and set it equal to 0

Create a while loop to go through all command line inputs that were specified, the loop should make sure (opt equals getopt() which has arguments argc, argv, OPTIONS) and that this whole thing doesn't equal -1. The -1 signals that all command line options have been read

Create a switch statement with argument opt, to check for each case that was specified in OPTIONS

Case 'a': Run all the tests

Case 'e': Run the Heap Sort

Case 'i':

Run the function insertion_sort() with arguments &stats, array A, variable size

Create for loop to loop through the size of the array and print each element until iterator reaches p, increment in steps of 1

Print the current value of the array A

Print the number of moves

Print the number of compares

Case 's': Run the Shell Sort

Case ‘q’: Run the Quick Sort

Case ‘r’:

Set variable seed equal to user input

If seed isn’t specified, set seed equal to 13371453

Case ‘n’:

Set variable size equal to user input

If size isn’t specified, set size equal to 100

Case ‘p’:

Set variable p equal to user input

If p isn’t specified, set p equal to 100

Case ‘h’:

Print out the program usage

Print out each option with description on each option

Credit:

I based all of my code off of the Python pseudocode provided by Dr. Long

I based my sorting.c test harness off of the code from TA Eugene Chou’s section video