Deeraj Gurram
dgurram@ucsc.edu
17 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 and compares performed on the array.
4. shell.h
    - Specifies the interface to shell.c
5. heap.c
    - This file contains the function heap_sort() using a max heap, which sorts an array using the heap sort algorithm. It also returns the number of moves and compares 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 and compares 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, cleans generated files, builds all executables, formats all source code files, and builds the sorting.
14. README.md
    - This file describes how to build the program using the Makefile and how to run the program using ./sorting and the command line options. It also explains the command line options that the program accepts, provides a list of known errors in the program, and gives credit to the various sources I received help from.
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.
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

Initialize a variable gap_max and set it equal to the formula as described in Dr. Long's pseudocode (log(3 + 2n) / log(3))

Initialize an array called B of size gap_max

Initialize a variable called gap and set it equal to gap_max

Create a for loop to loop through a range of length gap_max, loop from 0 to the size of gap_max, increment in steps of 1

Set array index B[i] equal to the formula as described in Dr. Long's pseudocode ((pow(3, gap) - 1) / 2)

Decrement gap by 1

Create for loop to loop through the array A and switch the elements as applicable, the loop should start at B[i] and stop before the current iterator reaches n, iterate in steps of 1

Set j equal to the nested 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 if they fit the conditions, make sure j is greater than or equal to B[i] 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 B[i] length before A[j]

Decrement j by B[i]

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

heap.c

Define a function max_child() with arguments stats, 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 stats, 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(stats, A, mother, last)

Create while loop to swap elements in the tree based on if mother is less than great, 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(stats, 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 stats, array A, the first node, and the last node

Initialize a variable called father

Create a for loop that loops over the first 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 stats, 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

Initialize a variable called leaf

Call the function build_heap with arguments stats, A, first, last

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

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

Call the function fix_heap with arguments stats, 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 stats, array A, the lo integer, and the hi integer

Set i equal to lo - 1

Create a for loop to swap the elements if the conditions are met, the loop should 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], this keeps swapping elements until they are in order

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 stats, array A, integer lo, and integer hi

Check if lo is less than hi

Create a variable p and set it equal to partition(stats, A, lo, hi)

Run quick_sorter() recursively with arguments stats, 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 stats, 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 stats, A, 1, and length of A

sorting.c

Include the following:
all .h files
various std libraries
inttypes.h
unistd.h

Include the OPTIONS “haeisqn:p:r:” to run the various test cases

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

Call the stats struct

Initialize the moves to 0

Initialize the compares to 0

Define a variable seed and set it equal to 13371453

Define a variable print_elements and set it equal to 0

Define a variable actual_elements and set it equal to 0

Initialize the variable opt and set it equal to 0

Create a boolean found and set it equal to true

Create an empty set s

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’:
    Set boolean no_input to false
    Insert HEAP into set s
    Insert SHELL into set s
    Insert INSERTION into set s
    Insert QUICK into set s

Case ‘e’:
    Set boolean no_input to false
    Insert HEAP into set s
Case ‘i’:
    Set boolean no_input to false
    Insert INSERTION into set s
Case ‘s’:
    Set boolean no_input to false
    Insert SHELL into set s
Case ‘q’:
    Set boolean no_input to false
    Insert QUICK into set s
Case ‘h’:
    Set boolean no_input to false
    Insert HELP into set s
Case ‘r’:
    Set variable seed equal to user input
Case ‘n’:
    Set variable actual_elements equal to user input
Case ‘p’:
    Set variable print_elements equal to user input

Run srandom(seed) to establish the seed pattern

Allocate the dynamic memory by setting *A = the calloc command
Create for loop to go from 0 to the length of array A and increment by 1
    Set A[iterator] equal to random() and mask

Check if no_input is true
    Print the help message

Create for loop to gro through the enum Sorts, iterator should start at HELP and stop before it reaches NUM_SORTS, increment by 1
    Check if the iterator is a member of set s
        If iterator equals HELP

Print help message
Exit out of for loop
If iterator equals INSERTION
Run the insertion sort algorithm
Print the name of the algorithm

If iterator equals SHELL
Run the shell sort algorithm
Print the name of the algorithm

If iterator equals HEAP
Run the heap sort algorithm
Print the name of the algorithm

If iterator equals QUICK
Run the quick sort algorithm
Print the name of the algorithm

Print the number of elements in the algorithm
Print the number of moves
Print the number of compares

Create a for loop to go through array A and print the formatted results, iterator should start at 0, stop when it reaches the actual_elements and increment in steps of 1
Print the results with each row having 5 columns

Reset the stats

Do srandom(seed) to reset the seed pattern

Create for loop to go from 0 to the length of array A and increment by 1
Set A[iterator] equal to random() and mask

Free the memory of A
Return the number 0