cse13s asgn3 DESIGN.pdf

Lucas Lee; CruzID: luclee

1/21/2022

1 Program Details

In this program we will have to make an Insertion Sort, Batcher Sort, Heapsort, and Quicksort based on python pseudocode in C. We will then implement a test function in order to test each sorting method using random seeds. This test case should gather the size of the array to sort, number of moves needed to sort the array, and number of times values are compared in the array.

2 Pseudocode and ideas

Insertion Sort:

1. insertion sort(stats struct pointer, uint32t array, length of array)
    (a) for loop from (1 - array length):
        i. j = current loop iteration
        ii. x = current array value
        iii. while j greater than 0 and x less than (array value at j-1)
            A. value at j = value at j - 1
            B. j -= 1
        iv. value at j = x

Heapsort (needs maintenance code and sort code): Maxchild (highest of the two children):

1. maxchild(list, int first, int last)
    (a) left = first * 2
    (b) right = left +1

(c) if right less than or = last and (array value at right -1) less than (array value at left -1)

i. return right

(d) return left

Fixheap(fixes heap after you change the mother and father values)

1. fixheap(array, int first, int last)

(a) found = false

(b) mother = first

(c) greater = maxchild(array, mother, last) finds greater of the two children and stores into greater

(d) while mother less than or = (last // 2) and found == false (floor division)

i. if (array at mother -1) less than (array at greater - 1)

A. switch (array at mother -1) and (array at greater -1)

B. mother = greater

C. greater = maxchild(array, mother, last)

ii. else found = true

build heap creates heap for the sort algorithm

1. buildheap(array, int first, int last)

(a) for father in range from (last // 2, to first -1, -1) (floor division on last, to first -1 stepping from -1 forward, meaning it counts backwards from last to first)

i. fixheap(array, father, last)

Sort algorithm:

1. heapsort(stats struct pointer, uint32t array, array length)

(a) first = 1, last = array length

(b) call buildheap(array, first, last)

(c) for i in range(last, first, -1) counting backwards from last to first

i. swap (array at first - 1) and (array at i -1)

ii. fixheap(array, first, i -1)

Quicksort Partitions:

1. partition(array, int low, int high)
    (a) i = low - 1
    (b) for x in range(low, high)
        i. if (array at j-1) less than (array at high -1)
            A. i++
            B. (array at i-1) swaps with (array at j -1)
        ii. (array at i) swaps with (array at high -1)
    (c) return i + 1

Quicksort Sorting Algorithm

1. quicksort(array, int low, int high)
    (a) if low less than high
        i. int p = partition(array, low, high)
        ii. quicksort(A, low, p -1)
        iii. quicksort(A, p + 1, high)

Function that actually runs this recursive function

1. quicksorter(array)
    (a) quicksort(array, 1, length(array))

Batcher's Odd-Even Merge Sort function comparator:

1. comparator(array, int x, int y)
    (a) if (array at x) greater than (array at y)
        i. swap (array at x) with (array at y)

Function that runs the batcher sort

1. batchersort(array)
    (a) if length(array) == 0 base case
        i. return nothing, exit function
    (b) n = length(array)
    (c) t = bit length of n

(d) p = 1 shifted left by (t -1) bits

(e) while p greater than 0

i. q = 1 shifted left by (t -1) bits

ii. r = 0

iii. d = p

iv. while d greater than 0

A. for x in range(0, n-d)

B. if (i and p) == r bitwise and

C. comparator(array, i, i + d)

D. d = q - p

E. q = q shifted right by 1 bit

F. r = p

v. p = p shifted right by 1 bit

3 Files

1. batcher.c and batcher.h = implementation batcher sort and header file for batcher sort

2. insert.c and insert.h = implementation of insertion sort and header file for insertion sort

3. heap.c and heap.h = implementation of heap sort and header file for heap sort

4. quick.c and quick.h = implementation of quicksort and header file for quicksort

5. set.h implements sets that we will use for bitwise operations when keeping track of command line inputs.

6. stats.c and stats.h = implements stats file we use to keep track of data from sorting, and the header file.

7. sorting.c = main function that runs and tests sorting functions using command line inputs.

8. Makefile uses clang commands, make all, make sorting, make clean, and make format

9. README.md describes how to use Makefile and compile/run the program
10. DESIGN.pdf describes how the programmer will make the code and complete the implementations of the sorting algorithms.
11. WRITEUP.pdf shows what I learned from implementing the sorting algorithms and conclusions about the project. Includes data about the sorting outputs.
12. plot.sh file that produces the graphs using sorting.c
13. figs directory that contains my graphs used in the writeup

4 Main function

Main function that takes argc and argv:

1. initialize variables as needed
2. make a variable for each sort that enables them
    (a) switch operator and check for inputs
    (b) if -a enable all sorts by adding all sorts to set
    (c) if -h enable heap sort by adding it to set
    (d) if -b enable batcher sort by adding it to set
    (e) if -i enable insertion sort by adding it to set
    (f) if -q enable quicksort by adding it to set
    (g) if -r: then set random seed to user input
    (h) if -n: then set array size to user input
    (i) if -p: then print user input elements from the array. Default is 100
    (j) if -H then print usage statement for program that tells the user how to use the program.
3. use malloc and allocate space for an array of uint32s

4. for loop for length of array that initializes the array to random values masked to 30 bits by using bitwise shifting
5. for each of enabled sorts:
    (a) run corresponding sort and print out desired output
    (b) conditionals that check if p value is 0, less than array length, or greater than array length to determine what to print out.
6. use free for every malloc command to avoid memory leaks