Krisha Sharma

CSE13s

Winter 2022 Long

01/23/2022

Assignment 3 DESIGN.pdf

Description of Program:

Assignment 3: Putting your affairs in order, is an assignment where we will be sorting pseudo-randomized arrays using 4 different arrays. The 4 arrays are Insertion Sort, Batcher Sort, Heap Sort, and recursive Quicksort. Each of these arrays

Files to be included in directory "asgn3":

- `batcher.c`
  - This file implements Batcher Sort
- `batcher.h`
  - This file specifies the interface to `batcher.c`
- `insert.c`
  - This file implements Insertion Sort
- `insert.h`
  - This file specifies the interface to `insert.c`
- `heap.c`
  - This file implements Heap Sort
- `heap.h`
  - This file specifies the interface to `heap.c`

- `quick.c`
  - This file implements recursive Quicksort
- `quick.h`
  - This file specifies the interface to `quick.c`
- `set.h`
  - This file implements and specifies the interface for the set ADT
- `stats.c`
  - This file implements the statics module
- `stats.h`
  - This file specifies the interface to the statistics module
- `sorting.c`
  - This file contains `main()` and *may* contain any other functions necessary to complete assignment 3.
- Makefile
  - The file that formats the program into clang-format and compiles it into program executable "`sorting`" with `make sorting` /`make all` from Makefile.
  - `CC = clang` must be specified
  - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified
  - `make` must build the `sorting` executable, as it should `make all` and `make sorting.`
  - `make format` should format all the source code, including the header files
- README.md

  - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
  - Describes design for the program thoroughly with pseudocode and visualizations.
- WRITEUP.pdf
  - This document must be a PDF
  - What did you learn from the different sorting algorithms? Under what conditions do sorts perform well? Under what conditions do sorts perform poorly? What conclusions can you make from your findings?
  - Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements. The graphs must be produced using either `gnuplot` or `matplotlib`.
  - Analysis of the graphs produced.

General Notes:

- Assignment 3 - The Sorting Assignment
  - During this assignment, we will be sorting pseudo-randomized arrays using 4 different arrays.
  - The 4 arrays are Insertion Sort, Batcher Sort, Heap Sort, and recursive Quicksort.
    - Insertion Sort: taking things that are out of order, parsing through them starting at the first element and then putting them in the same place.

- Heapsort: establishes a partial order over all the elements in an array. It will first build a heap and then fix the heap.
    - Heapsort structure is based on a concept called a binary tree. Binary trees are rooted at some node, and in a binary tree any node can have at most 2 "children" but it is also possible for a node to have one "child" or no "children". In heapsort, you can have a max-heap and a min-heap.
        - For a max-heap, any parent node must have a value that is greater than or equal to the value of their "children".
        - For a min-heap, any parent node must have a value that is less than or equal to the value of their "children".
    - In the Heapsort algorithm, we will create for this assignment, max-heap is what will be used the most. The heap is represented as an array in which for any index k, the index of the left "child" is 2k and the index of the right "child" is 2k + 1.
    - The root node is always going to be the first element and in a max-heap, the root node is always going to be the largest element in the array
- Quicksort: Possibly the fastest recursive sort on average, Quicksort (sometimes called a partition-exchange sort) works as a divide-and-conquer algorithm. To start, it partitions an array into two subsequent arrays by selecting an element from an array and designating it as a pivot, then elements that are less than the pivot are placed to the

left-sub array, and elements that are greater than or equal go to the right-sub array.

- Quicksort uses a subroutine called `partition()` which will place elements less than the pivot to the left side of the array and elements that are greater than or equal to the pivot to the right side, additionally, it will also return the index that indicates the division between the partitioned part of the array.

■ Batchers Odd-Even Merge Sort: sorts the even and odd subsequences of an array.

- It is a sorting network

■ The first task in this assignment is to implement all of the above sort methods.

- The interface for all these methods are given as the header files `insert.h`, `batcher.h`, `heap.h`, and `quick.h`

■ The second task in this assignment is to implement a test harness for the sorting algorithms made in the first task. To make the test harness I will first need to make an array of pseudorandom elements to test each of the sorts with.

- The test harness must be in the file `sorting.c`

■ Lastly, gather the statistics about each sort and its performance

- Record the size of the array, the number of moves required, and the number of comparisons required.

- A code is given in set.h that accounts for the command-line options outlined in the assignment 3 instruction pdf.
  - Set.h handles the command line options
- `Malloc()` or `calloc()` handles dynamic memory allocation

Pseudocode

- **Insertion Sort:**
  - Insertion sort is $o(n^2)$ because the function iterates over $o(n)$ twice.
  - The pseudocode written for Insertion Sort is taken from the python pseudocode given in the assignment 3 instruction pdf provided by the professor and it is also based on Eugene's section video on January 21st.
  - Python pseudocode:

```
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

  - Pusedocode explained:

insertion sort (given some array A)

initialize i = 1

for i in range of 1 and the length of the array copy what index you are on, set j = 1

save the value, temp = A[i]

while j is greater than 0 and the value (temp) is less than A[j - 1]

set A[j] = A[j - 1]

decrease the count by 1, j -= 1

reset the value, A[j] = temp

- **Heapsort:**
  - The pseudocode written for Heapsort is taken from the python pseudocode given in the assignment 3 instruction pdf provided by the professor and it is also based on Eugene's section video on January 21st.
  - Python pseudocode:

```
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left
def fix_heap(A: list, first: int, last: int):
    found = false
    mother = first
    great = max_child(A, mother, last)
    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
```

```
                A[mother - 1], A[great - 1] = A[great -
                1], A[mother - 1]
                mother = great
                great = max_child(A, mother, last)
            else:
                found = true
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)
def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1],
        A[first - 1]
        fix_heap(A, first, leaf - 1)
```

- Pseudocode explained:

  the first thing we are going to do in heap sort is built_heap

  build_heap

  - make sure that the array obeys the heap property

    take the last index of the array and divide by 2 to get the middle (start at one level above the bottom of the "binary tree")

    get the height of the binary tree, using $log_2(n)$

max_child

set left equal to 2 * first

set right equal to left + 1

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

return right

return left

fix_heap ($log_2(n)$)

inatilize found as false

initialize mother as first

initialize great and set it equal to max_child(A, mother, last)

while the mother is greater than or equal to last divided by 2

A[mother - 1] = A[great - 1], swap the values of mother and great

A[great - 1] = A[mother - 1], swap the values of mother and great

mother = great

set great equal to the max child value

heap_sort

initialize first = 1

initialize last as the length of the array

build the heap

for leaf in range of last, first and -1

A[first - 1] = A[leaf - 1], swapping the values

A[leaf - 1] = A[first - 1], swapping the values fix_heap(A, start at the root which is first, leaf - 1 which is where the array ends)

- **Quicksort:**
  - Quicksort will make use of a function `partition()` that will be written to help divide and sort the elements in the array
  - The pseudocode written for Quicksort is taken from the python pseudocode given in the assignment 3 instruction pdf provided by the professor and it is also based on Eugene's section video on January 21st.
  - Python pseudocode:

```
def partition(A: list, lo: int, hi: int):
     i = lo - 1
     for j in range(lo, hi):
          if A[j - 1] < A[hi - 1]:
               i += 1
               A[i - 1], A[j - 1] = A[j - 1], A[i -
               1], A[i], A[hi - 1] = A[hi - 1], A[i]
     return i + 1
def quick_sorter(A: list, lo: int, hi: int):
     if lo < hi:
          p = partition(A, lo, hi)
          quick_sorter(A, lo, p - 1)
          quick_sorter(A, p + 1, hi)
def quick_sort(A: list):
     quick_sorter(A, 1, len(A))
```

- Pseudocode explained:

  partition (given some array A)

      set i equal to the low - 1

      for j in range of the low and high

          if the array at index j - 1 is less that the array at index high - 1

              increment the count by 1, i += 1

              A[i - 1] = A[j - 1]

              A[j - 1] = A[i - 1]

              A[i] = A[high - 1]

              A[high - 1] = A[i]

          return the value of i + 1, return i + 1

  quick sorter

      if the low is less than the high

          p = partition(A, low, high)

          recursively call quick_sorter(A, low, p - 1)

          recursively call quick_sorter(A, p + 1, high)

  quick sort

      quick_sorter(A, 1, length of array A)

- **Batchers Sort:**
  - Batchers Sort
  - The pseudocode written for Batchers Sort is taken from the python pseudocode given in the assignment 3 instruction pdf provided by the professor and it is also based on Eugene’s section video on January 21st.

- Python pseudocode:

```
def comparator(A: list, x: int, y: int):
    if A[x] > A[y]:
        A[x], A[y] = A[y], A[x]
def batcher_sort(A: list):
    if len(A) == 0:
        return
    n = len(A)
    t = n.bit_length()
    p = 1 << (t - 1)
    while p > 0:
        q = 1 << (t - 1)
        r = 0
        d = p
        while d > 0:
            for i in range(0, n -d):
                if(i & p) == r:
                    comparator(A, i, i + d)
            d = q - p
            q >>= 1
            r = p
        p >>= 1
```

- Pseudocode explained:

  comparator

```
        if A[x]  is greater than A[y]
                swap the indices, A[x] = A[y]
                swap the indices A[y] = A[x]
batcher sort
        if the length of the array is equal to 0
                return
        set n equal to the length of the array
        set t equal to n.bit_length()
        set p = 1 << (t - 1), shift the bit p to the left 1
        while p is greater than 0:
                set q = 1 << (t - 1), shift the bit q to the left 1
                set r = 0
                set d = p
                while d is less than 0:
                        for i in range(0, n -d):
                                if(i & p) == r:
                                        comparator(A, i, i + d)
                        set d = q - p
                        q >>= 1, shift the q bit to the right 1
                        set r = p
                p >>= 1, shift the p bit to the right 1
```

Error Handling

- I haven't run into any errors yet, but I will eventually as I start testing my program more. This section is where I will explain the errors I ran into while completing this assignment.

Citations

- This section of my DESIGN.pdf will contain my citations for this assignment. It will detail what sections I went to and what they helped me with, as well as any other resources I used to complete this assignment.
- I watched Eugene's recorded section video that he posted to Yuja to help me get started on this assignment. The section was on the 21st of January, and in it, Eugene talks about assignment 3, sorting, time complexity, sets, and dynamic memory allocation `[malloc()/calloc()]`