Caterin Duarte Reynosa
cduarter@ucsc.edu

CSE13S Fall 2020
Assignment 5: Sorting Putting your affairs in order
Design Document

Goal

The goal for this assignment is to implement different sorting algorithms and to familiarize with how they work, as well their time complexity. We are also going to be working with sets and the stack in order to get a deeper understanding on how they work.

PreLab questions

Prelab Part 1

1.) Sort 8 22 7 9 31 5 13 would take me 6 rounds of swapping to order this set
    a.) 8 7 9 22 5 13 31
    b.) 7 8 9 5 13 22 31
    c.) 7 8 5 9 13 22 31
    d.) 7 5 8 9 13 22 31
    e.) 5 7 8 9 13 22 31
    f.) 8 7 9 22 5 13 31
2.) In the worst case scenario, we can expect o(n^2) comparisons if the smallest number is at the end. It will be the worst because all the numbers will have to be compared. We can expect to see Comparisons= (n-1) amount of comparisons based on the size of the array. For example, if we an array that has the elements 10 6 12 1, we would implement the bubble sorting algorithm as such: it would take
    a.) Original: 10 6 12 1
    b.) 1st pass: 6 10 1 12
    c.) 2nd pass: 6 1 10 12
    d.) 3rd pass: 1 6 10 12
    e.) 4th pass: 1 6 10 12
3.) We can revise the bubble sorting algorithm so that the smallest elements float to the top if we reverse the order in which we compare the elements. That way, we will aim to move things to the front faster than the elements being moved to the back.

Source: https://www.sparknotes.com/cs/sorting/bubble/section1/

PreLab Part 2

1.) The worst time complexity for shell sort depends on the gap sequences because it compares elements that are specific ‘gap’ apart. Since the gap is smaller or larger for different sequences, it would change the worst time complexity for them. If the gap is small, it will cause the worst time complexity for shell sort. Even though this is considered the worst case for the sorting algorithm, we can change this by changing the gap size to a bigger number. Sources used: https://en.wikipedia.org/wiki/Shellsort

PreLab Part 3

1.) Even though quick sort's worst time complexity case is o(N^2), people still believe it's a good way to sort. This is because it's worst case complexity can be avoided more easily than other algorithms by using a different pivot such as the right pivot. Sources: https://www.geeksforgeeks.org/quicksort-better-mergesort/

PreLab Part 4

1.) I plan to keep track of the number of moves in my program by creating two functions, one for comparing and one for swapping, that will keep track of how many comparisons and how many swappings I do. Both of the functions would take in two paraments, the two cells that will be compared or swapped. I will also try to include them in a new file so as to not disrupt the other files.

Summary + Rules

- Implement a bubble sort
- Implement a shell sort
- Implement a quicksort
- Implement a stack ADT
- Implement heapsort
- Implement set ADT + getop

Function/Goal of each file I have to implement

- bubble.c
    - Implement a bubble algorithm using the python pseudo
- Shell.c
    - Implement a shell sorting algorithm using the python pseudo
- Quick.c
    - Implement a quicksort algorithm using the python pseudo
- Stack.c
    - Stack *stack_create(void)
        - Allocated memory for the stack
    - void stack_delete(Stack **s)
        - Frees memory for the stack
    - bool stack_empty(Stack *s)
        - Checks if a stack is empty or not
    - bool stack_push(Stack *s, int64_t x)
        - Pushes an item to the stack
    - bool stack_pop(Stack *s, int64_t *x)
        - Pops an item off the stack
    - void stack_print(Stack *s)
        - Prints out the stack
- Heap.c
    - Implement a heap sorting algorithm using the python pseudo
- Set.c
    - Set set_empty(void)
        - returns an empty set
    - bool set_member(Set s, uint8_t x)
        - Checks to see if a set is present using bit manipulation

    - Set set_insert(Set s, uint8_t x)
        - Makes x equals to 1 using bit manipulation
    - Set set_remove(Set s, uint8_t x)
        - Clears a specific bit to 0 using bit manipulation
    - Set set_intersect(Set s, Set t)
        - Calculates the bits in both set s and set t and return a new set
    - Set set_union(Set s, Set t)
        - United set s and set t into one and then returns is
    - Set set_complement(Set s)
        - Returns the complement of a set
    - Set set_difference(Set s, Set t)
        - Finds the difference between set s and elements not in set t
- Stats.c
    - Keep track of the comparisons and moves made by the sorting algorithms
    - Void comparing (expt 1 , exp2)
        - Keeps track of how many comparisons (comparing between two elements in an array)
    - Void swapping (exp 1, exp2)
        - Keeps tracks of many many swappings were made

Pseudocode

- bubble.c
    - The algorithm bubble sort is a simple stable sorting algorithm that sorts by comparing elements and swapping them. The algorithm compares adjacent elements and swaps them if the left one is larger than the right element. This happens until the largest elements bubble to the back of the elements. Once you have the largest element at the end of the array, you can ignore it next iteration and repeat the process until we only have no swaps.
    - Examples of bubble sort
        - [8 9 12 16 18 1]
        - [8 9 12 16 1 | 18]
        - [8 9 12 1 | 16 18]
        - [8 9 1 | 12 16 18]
        - [8 1 | 9 12 16 18]
        - [1 | 8 9 12 16 18]
        - [1 8 9 12 16 18]
    - Even though this array only had one element out of order, it still took a while for it to be sorted because the average time complexity of bubble sort is o(N^2)
- Shell.c
    - The shell sort algorithm compares elements that are certain distances, also known as gap, and swaps them so that the biggest number is at the end and the smallest number is in the front. As the iterations increment, the gap gets smaller. During the last iteration, all the numbers are compared.
    - Examples of shell sort: [8 9 12 16 18 1]
    - [8 1 12 16 18 9]

    - [8 1 9 16 18 12]
    - [8 1 9 12 18 16]
    - [1 8 9 12 18 16]
    - [1 8 9 12 16 18]
    - As we can see from the sorted arrays, shell sort is a little faster than bubble sort.
- Quick.c
    - The quick sort algorithm is one of the fastest algorithms because it uses a divide and conquer method. In order to sort using quick sort, we first pick a pivot in the array (which is an element in the array) and pick the left and right elements to compare. The left element of the pivot represents smaller numbers while the right element represents bigger numbers. Starting from the right and left element, we use those to compare them with the pivot element. If the left element is larger than the pivot element, wait. If the right element is smaller than the pivot, wait. Then compare both right and left and swap them. Do this until the algorithm is sorted.
    - Example: [8, 9, 12, 16, 18, 1]
    - [1 9 12 16 18 8]
    - [1 8 12 16 18 9]
    - [1 8 9 16 18 12]
    - [1 8 9 12 16 18]
- Stack.c
    - Stack *stack_create(void)
        - Use malloc to allocate memory for the stack
        - Stack_create = (stack)malloc (1, size of capacity)
        - Top of stack = 0
        - Capacity = min_capacity
        - Allocate memory for the items
    - void stack_delete(Stack **s)
        - Free (items)
        - Free stack
        - Stack = null
    - bool stack_empty(Stack *s)
        - If top of stack == 0
            - Return true
        - Else
            - Return false
    - bool stack_push(Stack *s, int64_t x)
        - If top of stack == capacity
            - Capacity = capacity x 2
            - Items = reallocate (items, capacity)
            - If items == nulls
                - Return false
            - Else
                - Top = x

            - Increment the top
            - Return true
    - bool stack_pop(Stack *s, int64_t *x)
        - If top = 0
            - Return false
        - else :
            - Deference x
            - Pop it
            - Pass it back to x
            - Return true
    - void stack_print(Stack *s)
        - Prints out the stack
        - Loops
- Heap.c
    - Heap sorting is a sorting algorithm that quickly sorts an array. In an array, the first element (in index 1) will be the root of the heap. The elements 2k and 2k+1 will be the children. We swap the first element with the last element of the array and the heapify the remainder of the array. This causes a decrease in the heap by one and will continue until all the elements are sorted correctly.
    - Example: [8, 9, 12, 16, 18, 1]
        - [16, 9, 12, 8, 1, 18]
        - [12, 9, 1, 8, 16, 18]
        - [9, 8 ,1, 12, 16, 18]
        - [8, 1 9, 12, 16,18]
        - [1 , 8, 9, 12, 16, 18]
- Set.c
    - Set set_empty(void)
        - returns 0
    - bool set_member(Set s, uint8_t x)
        - Shift by x % 32 and & with set s
    - Set set_insert(Set s, uint8_t x)
        - Shift by x % 32
        - Or the shift with s
    - Set set_remove(Set s, uint8_t x)
        - Shift by ~x % 32
        - S anded with the shift
    - Set set_intersect(Set s, Set t)
        - S & and t
    - Set set_union(Set s, Set t)
        - S or t
    - Set set_complement(Set s)
        - Not s
    - Set set_difference(Set s, Set t)
        - S anded with not t

-
- Stats.c
    - Void comparing (expt 1 , exp2)
        - If expt 1 > exp 2
            - Counter += 1
        - If expt 2 > expt 1
            - Counter += 1
    - Void swapping (exp 1, exp2)
    - If swap(exp1, exp2)
        - Counter += 1
    - If swap(exp1, exp 2)
        - Counter += 1