DESIGN

Pre-Lab Part 1

1. [8, 22, 7, 9, 31, 5, 13]

6 iterations [8, 7, 9, 22, 5, 13, 31]

5 iterations [7, 8, 9, 5, 13, 22, 31]

4 iterations [7, 8, 9, 9, 13, 22, 31]

3 iterations [7, 5, 8, 9, 13, 22, 31]

2 iterations [5, 7, 8, 9, 13, 22, 31]

1 iteration [5, 7, 8, 9, 13, 22, 31]

6+5+4+3+2+1 = 21 iterations

2.) We can expect dangerous scenarios like: [31, 22, 13, 9, 8, 7, 5] where the list is in reverse order and you must always keep pushing larger numbers to the top.

Pre-Lab Part 2

1.) In the worst case, the shell sort is essentially an insertion sort. It approaches $O(n^2)$ complexity. The average complexity depends on the gap sizes, because it changes the amount of comparisons you have to do.

Pre-Lab Part 3

1.) Quicksort is not doomed due to its worst case being $O(n^2)$. This is because Quicksort uses midpoints or pivots which eliminates the need to compare numbers left of pivot and sort them in ~~way~~ ordered way. It is rare for it to be $O(n^2)$ bc you would have to pick a minimum pivot out of all the numbers.

Pre-Lab Part 4

1.) I will intialize a counter for each sorting algorithm that will be incremented for each comparison. Likewise, a ~~var~~ counter will also be initialized and increment each time the algorithm swaps two numbers.

Struct Stack last in => first out System

- Use Stack.h as a interface for actual structure
- Actual definition will be in Stack.C
- Will have following components that we track
  - top (to track where to pop/push)
  - capacity (to track if we can push)
  - items (to hold actual values)

Stack_Create
- Function to instantiate a stack w/ given ~~size~~ capacity
- Each item requires space of 4 bits, (use calloc)
- top is zero
- capacity is capacity

Stack_Delete
- Function to delete stack and make it null.
- Use free() w/ a pointer to a pointer (free(&s))

Stack_Empty
- Returns true if top is at index 0
- False if otherwise

Stack_Size
- Returns index of top

Stack_Push · Place given value at index of top
· Increment top

Stack_Pop · ~~Decrement top~~ & Place value of number in ~~items~~ array of items in given x ~~and decrement top~~

Stack_print · ~~Print~~ Iterate through items and print each one, If ~~Null~~ Stack is NULL print "Stack is NULL",

---

Queue Structure · Has components
- Head (to track where to place next item)
- Tail (to track which item is to dequeue)
- Size (tracks amount of items)
- Capacity (tracks when we are full)
- items (Array to hold items)

Queue_Create · Instantiate each item w/ u bits using calloc
· head, tail, size start at zero

Queue_delete · To delete queue and prevent memory leaks use free() w/ ~~**~~ queue

queue_empty
- if size = 0 then return true
- Otherwise, return false

queue_full
- If size = capacity, return true
- Otherwise return false

queue_size
- return size

enqueue
- If full return false
- Else, increment size and place given value at ~~index~~ the tail index in item array
- Increment tail

dequeue
- If queue is empty return false
- Else decrement size
- place value at head index in item array in given x and increment head

queue_print
- iterate through all items and print each one.

Assignment 3 DESIGN

Purpose Assignment 3 towards a bubble, shell, and quick sort (w/ queue/stack). We will generate arrays of varying sizes and have the user input commands to control size, print size, sorting algorithm. Output will print statistics.

Layout/Structure

Bubble Sort Bubble sort will be a function taking in an array and a size of that array.

- It will iterate & take pairs within array.
- If one item is greater, move it rightwards
- This eventually guarantees the greatest element to be the most rightwards (excluding the previous numbers we sorted)

[1, 4, 3, 2] => [1, 3, 2, 4] ← sorted, do not touch

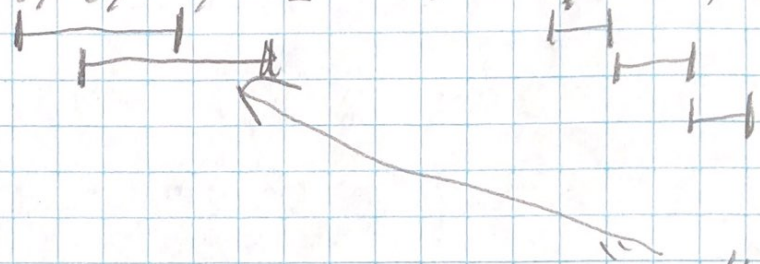
=> [1, 2, 3, 4]

- Each pair we compare, we increment a compare counter to track comparisons.
- Each time we find item greater, we swap and increment move counter

Shell Sort

- Function takes an array and size of array
- Iterates through gap sizes until we find a gap that is less than size of array
- Compare elements using gap and swap if rightwards if left item is greater than right
- Decrement gap & repeat.

Example: [1, 3, 2, 4] => [1, 3, 2, 4]



[1, 2, 3, 4]

- Increment compare counter each time we "draw" or compare pairs.
- Increment move counter each time we swap items if left item is greater than right.

Quick Sort · Function takes away and away size

- We use a pivot point; if left item of pivot is greater we move it right of pivot. If item right of pivot is less than pivot we move it left of pivot.
- Now we don't need to compare items left of pivot w/ items right of pivot. Halving comparisons
- Repeat w/ a pivot within the pivot.

Example [64, 34, 25, 12, 22, 11, 90]

=> [11] [12] [64, 34, 25, 22, 90]

Left Pivot Right

=> MERGE => [11, 12, 64, 34, 25, 22, 90]

=> [22] [25] [64, 34, 90]

Left Pivot Right

MERGE => [11, 12, 22, 25, 64, 34, 90]

=> [34] [64] [90] => MERGE [11, 12, 22, 25, 34, 64, 90]

Left Pivot Right

- Increment comparisons each comparison w/ pivot
- Increment moves each swap to the left or right of pivot.