Tanya Gyanmote
Cruz id - tgyanmot

DESIGN DOC

Description

- In this program we are creating different sorting algorithms, the ones implemented are Shell Sort, Bubble Sort, Quick Sort, and Heap Sort. We will also create the main file to test out the moves, and compares each sort.

Main

- Create a function for -H case that prints out the usage of the program
- Create a function that will populate my array using the seed function, mtrand_seed
    - Have a for loop that will go through the array to populate each element
        - Return the array
- Have the default variables
- Create a set using set_empty
- Create cases for each type of input
    - Case a to print out all the sorts
    - Case s for shell sort
    - Case b for bubble
    - Case q for quick sort
    - Case h for heap sort
    - Case r for the seed
    - Case n for array size
        - Make sure the array size is within the range
        - If it isn't print the help message
            - And return a non-zero exit code
    - Case p for the elements of the array
    - Case h to print out the synopsis
    - Default case to print the help message, and return 1
    - In cases, a-h using the set I created use set_insert and have a number according to the sort
    - If the array size is less than the elements then change the elements to the size of the array
- For Bubble sort
    - Reset my stats
    - Create an array while calling my function that populates my array
    - Call bubble sort with the array I just populated
    - Have a temp variable to keep count of when I should create a new line
    - Have a print statement with Bubble sort, elements, moves, and compares
    - Have a for loop so the elements of the array could be printed
        - Use the print statement given in the pdf
        - Add 1 to my temp counting when I should add a new line
            - If my temp variable % 5 then its time to add a new line

            - Check after it prints all elements to make sure that if it doesn't start at the new line to print one
- For Shell sort
    - Reset my stats
    - Create an array while calling my function that populates my array
    - Call shell sort with the array I just populated
    - Have a temp variable to keep count of when I should create a new line
    - Have a print statement with shell sort, elements, moves, and compares
    - Have a for loop so the elements of the array could be printed
        - Use the print statement given in the pdf
        - Add 1 to my temp counting when I should add a new line
            - If my temp variable % 5 then its time to add a new line
        - Check after it prints all elements to make sure that if it doesn't start at the new line to print one
- For Quick sort
    - Reset my stats
    - Create an array while calling my function that populates my array
    - Call quick sort with the array I just populated
    - Have a temp variable to keep count of when I should create a new line
    - Have a print statement with quick sort, elements, moves, and compares
    - Have a for loop so the elements of the array could be printed
        - Use the print statement given in the pdf
        - Add 1 to my temp counting when I should add a new line
            - If my temp variable % 5 then its time to add a new line
        - Check after it prints all elements to make sure that if it doesn't start at the new line to print one
- For Heap sort
    - Reset my stats
    - Create an array while calling my function that populates my array
    - Call heap sort with the array I just populated
    - Have a temp variable to keep count of when I should create a new line
    - Have a print statement with heap sort, elements, moves, and compares
    - Have a for loop so the elements of the array could be printed
        - Use the print statement given in the pdf
        - Add 1 to my temp counting when I should add a new line
            - If my temp variable % 5 then its time to add a new line
        - Check after it prints all elements to make sure that if it doesn't start at the new line to print one

Shell Sort in Python

```python
def gaps(n):
  while n > 1:
    n = 1 if n <= 2 else 5 * n // 11
    yield(n)

def shell_sort(arr):
    for gap in gaps(len(arr)):
        for i in range(gap, len(arr)):
            j = i
            temp = arr[i]
            while j >= gap and temp < arr[j - gap]:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
    return arr
```

Shell

- Using the pseudocode from above
- Created a gap junction
    - If the gap is less than 2, then return the size -1
    - If the gap is less than 0 return 0
    - Else follow the equation from Knuth's gap sequence and return size times 5 - floor division of 11 to find the new gap
- Shell sort function
    - Create a for loop that goes through the gap function for every step, which starts with the first gap, checks to make sure it's greater than 0 and gets changed by the next gap
        - Creating another for loop that loops starting from the gap till the end of the array
            - Having a new variable to store I, the gap
            - Having a temp variable, where we store the value of the element of the array
            - Have a while loop that checks to see if our j, the gap is greater than our initial gap from the first for loop and if the temp variable is less than the array element of j-gap
                - If it is then change the value
            - Change j element of the array to the temp outside the loop

Bubble Sort in Python

```python
def bubble(a):
  for i in range (len(a) - 1):
    swapped = False
    for j in range (len(a) - 1, i, -1):
      if a[j] < a[j-1]:
        a[j], a[j-1] = a[j-1], a[j]
        swapped = True
    if not swapped:
      break
  return a
```

Bubble Sort

- Creating a for loop that goes through the array
  - Having a boolean to keep track of true and false
    - Having another for loop that goes through the array but started at the end of the array and decrements to the next element on the left of the last
      - If the “last” element of the array is less than the element before it then swaps the values
      - Make the boolean true if its swapped
  - If it’s not swapped then break
- Once it goes through the entire array, then it's completely swapped

Quick Sort in Python

```python
from shellsort import shellsort
SMALL = 8

def quicksort(a):
  if len(a) < SMALL:
    return shellsort(a)

  pivot = (a[0] + a[len(a) // 2] + a[-1]) // 3
  left  = [ _ for _ in a if _ <  pivot]
  mid   = [ _ for _ in a if _ == pivot]
  right = [ _ for _ in a if _ >  pivot]
  return quicksort (left) + mid + quicksort (right)
```

Quick Sort

- Using the pseudocode from above
- Have a variable small equal to 8
  - Check if the size of the array is less than 8 using small
    - if it is then call shell sort
- Have a variable for left, right, and pivot
- Left means beginning, right means end, and pivot means pivot value from the equation
- Set left to 0, the start of the array
- Set right to n_elements -1, end of the array
- Set pivot to the equation that was given above
- Create a while loop to make sure the left is always going to be less than the right because we want the array from least to greatest

- Checking to see if the pivot is greater than the left element or if the left element is equal to the pivot
    - If it decrements the left
- Checking to see if the pivot is less than the right element
    - is it then decremented right by 1
- Check if the right is greater than left
    - If it is then swap the left and right elements
- Call quicksort with the first element and left
- Call quicksort with the array of left and size of the array minus 1

```
def l_child (n):
  return 2 * n + 1

def r_child (n):
  return 2 * n + 2

def parent(n):
  return (n - 1) // 2

def up_heap (a, n):
  while n > 0 and a[n] > a[parent(n)]:
    a[n], a[parent(n)] = a[parent(n)], a[n]
    n = parent(n)

def down_heap (a, heap_size):
  n = 0  # Down heap from root
  while l_child(n) < heap_size:
    if r_child (n) == heap_size:
      bigger = l_child(n) # If there's no right child, the left is bigger
    else:
      bigger = l_child(n) if a[l_child(n)] > a[r_child(n)] else r_child(n)
    if a[n] > a[bigger]:
      break
    a[n], a[bigger] = a[bigger], a[n]
    n = bigger

def build_heap (a):
  heap = [0] * len(a)
  for n in range(len(a)):
    heap[n] = a[n]
    up_heap (heap, n)
  return heap

def heapsort (a):
  heap = build_heap (a)
  sorted_list = [0] * len (a)
  for n in range (len(a)):
    sorted_list[n], heap[0] = heap[0], heap[len(a) - n - 1]
    down_heap (heap, len(a) - n)
  return sorted_list
```

Heapsort

- Creating left child
    - Returning 2 times n plus one
- Creating right child
    - Returning 2 times n plus two
- Creating parent
    - Returning n minus 1 divided by 2
- Create up_heap check if the left child of the array is less than the parent of the array because we are trying to find the smallest number in the array, for the min heap
- Create a down heap using the pseudo from above
    - Have a variable n equal 0
    - Have a while loop to make sure the left child is less than the heap size
        - If the r child is the heap size
            - then make the smallest value left child
        - If the r child is not the heap size
            - then if the left child is less than a right child

            - using the smaller value and setting it to the left child
        - If none of the conditions above are true then make the smallest value right child
    - If element n is less than an array of then the array is sorted so you break
    - Swap the value of n element of the array and smaller element of the array if it's not the smallest
    - Change n to a smaller
- For buildheap using the puesocode
    - Create a heap array using calloc with n_elements and a size of uint32_t
    - Create a for loop that goes through the array
        - populate the array with n
        - call upheap using the heap array
    - Return heap
- For heapsort
    - Create a heap array using buildheap
    - Create a for loop that goes through the array
        - switch element n to the first element of heap
        - switch first element of heap to the last element of heap
    - free the heap after using

Files

- Bubble.c implements BubbleSort.
- Bubble.h specifies the interface to bubble.c.
- Heap.c implements HeapSort.
- Heap.h specifies the interface to heap.c
- Shell.c implements ShellSort.
- Shell.h specifies the interface to shell.c.
- Quick.c implements QuickSort.
- Quick.h specifies the interface to quick.c.
- Set.c implements set ADT
- Set.h specifies the interface for the set ADT
- Stats.c implements the statistics module.
- Stats.h specifies the interface to the statistics module.
- Mtrand.c implements the Mersenne Twister module.

- Mtrand.h specifies the interface to the Mersenne Twister module.
- Sorting.c contains main()

Credit

- All credit for pseudocode is goes to Professor Miller, which is what I used for my sorting functions