Assignment 3 Design

Caitlin Smith

February 5, 2023

Program Description

In this assignment, there are multiple files containing the sorting methods (Shell, Heap, Quick, and Batcher). The statistics of the moves and compares used are tracked for the sorts using stats.h. There is a program to then test the sorting using randomized arrays. It utilizes command-line arguments in the main function in order to then use a set to track what functions need to be run. The results can be printed to the standard output.

1 shell.c

Purpose: This contains the functions for Shell Sort. It uses stats.h to track the number of moves and compares. Shell uses gaps.h in order to produce an array of gaps to be used. Below is the python code referenced for this sort.

Shell Sort in Python

```
def shell_sort(arr):
    for gap in gaps:
        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
```

Pseudocode:

Include the shell.h, stats.h, and gaps.h files.
Create a function void shell_sort(Stats *stats, uint32_t *arr, uint32_t length) that will sort the passed array pointer and track stats.
Use a for loop to iterate through the gaps array in gaps.h.
    Set a variable gap to the g index of the gaps array.
    Create a for loop to run from i = the current gap until the end of the array.

Set a variable j to equal i.
Set a temp to equal the array at index i using stats move.
Create a while loop to run while j is greater than or equal to the gap, and the temp is less than the array at the index j minus the gap. Use the stats compare function.
Use stats move to set the array at j to the array at j minus the gap.
Subtract the gap from j.
Set the array at j to the temp using stats move.

2 heap.c

Purpose: This contains the functions for Heap Sort. It uses stats.h to track the number of moves and compares. Below is the python code referenced for this sort.

Heap maintenance in Python

```
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
```

Heapsort in Python

```
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:

Include the heap.h and stats.h.

Create a function max_child taking in an array and variables for first and last.
Set a variable left to 2 times first.
Set a variable right to left plus 1.
If right is less than or equal to last and the array at right minus 1 is greater than the array at left minus 1 (use the stats compare function).
Return Right.

Return left.

Create a function fix_heap taking in an array and variables for first and last.
Set a boolean variable found to false.
Set a variable mother to first.
Set a variable great to what is returned from calling max_child passing the array, mother and last.
Create a while loop that will run as long as mother is less than or equal to last divided by 2, and the bool found is false.

Using the stats compare function, if the array at mother minus 1 is less than the array at great minus 1

Use the stats swap function with the array at mother minus 1 and the array at great minus 1.

Set great to the output of max_great passing the array, mother and last.

Else, set found to be true.

Create a function build_heap taking in an array and variables for first and last.
Create a for loop to run from i = last divided by 2 until i reaches first minus 1 with steps of -1.

Call fix_heap passing the array, i and last.

Create a function void heap_sort(Stats *stats, uint32_t *A, uint32_t n) that will sort the passed array pointer and track stats.
Set a variable first to 1.
Set a variable last to the length of the array.
Call build_heap, passing the array and the first and last variables.
Create a for loop to run from i = last until i = first with steps of -1.

Use the stats swap function, passing the array at first minus 1 and i minus 1.

Call fix_heap, passing the array, first and i minus 1.

3 quick.c

Purpose: This contains the functions for Quick Sort. It uses stats.h to track the number of moves and compares. Below is the python code referenced for this sort.

Partition in Python

```
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
```

Recursive Quicksort in Python

```
# A recursive helper function for Quicksort.
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:

Include the quick.h and stats.h.

Create a function partition taking in an array and variables for low and high.
Set a variable i to low minus 1.
Create a for loop to run from j = low until it reaches high.
    Using stats compare, if the array at j minus 1 is less than the array at high minus 1
        Increment i by 1.
        Use stats swap to switch the array at i minus 1 and the array at j minus 1.
Use stats swap to switch the array at i and the array at high minus 1.
Return the variable i plus 1.

Create a function quick_sorter taking in an array and variables low and high.
If low is less than high
    Set a variable p to the result of calling partition passing the array, low and high.
    Call quick_sorter passing the array, low and p minus 1.
    Call quick_sorter passing the array, p plus 1 and high.

Create a function void quick_sort(Stats *stats, uint32_t *A, uint32_t n) that will sort the passed array pointer and track stats.
Call quick_sorter, passing the array, 1 and the length of the array.

4 batcher.c

Purpose: This contains the functions for utilizing a set in the main test file. It uses stats.h to track the number of moves and compares. Below is the python code referenced for this sort.

Merge Exchange Sort (Batcher's Method) in Python

```python
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:

Include the batcher.h and stats.h.

Create a function comparator taking in an array and variables for x and y.
Using stats compare, if the array at x is greater than the array at y
    Use stats swap to switch the array at x and y.

Create a function void batcher_sort(Stats *stats, uint32_t *A, uint32_t n) that will sort the passed array pointer and track stats.
If the length of the array is zero, return.
Set a variable i to n.
Set a variable t to 0.
Create a while loop to track t number of times i can be divided by 2. This gets the number of bits needed for the number n.
Set a variable p to 1 left shifted by t minus 1.
Create a while loop that will run while p is greater than zero.
    Set a variable q to 1 left shifted t minus 1.
    Set a variable r to 0.
    Set a variable d to p
    Create a while loop to run while d is greater than zero.

Create a for loop to run from i = 0 until i is n minus d.
If i and p is equal to r, call comparator passing the array, i and i plus d.
Set d to q minus p.
Right shift q by 1.
Set r to p.
Right shift p by 1.

5 set.c

Purpose: This contains the functions for utilizing a set in the main test file. It will track which command-line options are specified when the program is called.
Pseudocode:

Include the set.h file.

Create a function set set_empty(void).
Return 0.

Create a function set set_universal(void).
Return the complement of a set that is 0.

Create a function set set_insert(Set s, uint8_t x).
Use the bit-wise operator OR and shifting to set the designated bit to 1.

Create a function set set_remove(Set s, uint8_t x).
Use the bit-wise operator AND and shifting to set the designated bit to 0.

Create a function bool set_member(Set s, uint8_t x).
Use the bit-wise operator AND and shifting to see if the given x location of the set is set to 1. If the result is non-zero, return true. If the result is 0, return false.

Create a function set set_union(Set s, Set t).
Use the bit-wise operator OR to see all of the elements that are in both sets passed.

Create a function set set_intersect(Set s, Set t).
Use the bit-wise operator AND to see the elements that are in both sets.

Create a function set set_difference(Set s, Set t).
Use the bit-wise operator AND with the s and the negation of t. This finds the difference between the two sets. Create a function set set_complement(Set s).

Use the bit-wise operator NOT to find the complement of the set.

6 sorting.c

Purpose: This file contains the main function. It utilizes command-line arguments in order to then use a set to track what functions need to be run. They are tested using randomized arrays. The results and statistics can be printed to the standard output.
Pseudocode:

Include the necessary standard and the created h files.
Define OPTIONS to be "ahbsqr:n:p:H".

The following will all be within the main().
Initialize int opt.
Create variables for seed, elements and print size. These defaults are used if they are not specified in the command-line.
Create an empty set.

Create a while loop with getopt.
    Case a will set all of the corresponding bits for all of the sorts.
    Case h will set the bit for Heap Sort.
    Case b will set the bit for Batcher Sort.
    Case s will set the bit for Shell Sort.
    Case q will set the bit for Quick Sort.
    Case r will set the seed to the user input.
    Case n will set the number of elements to the user input.
    Case p will set the print size to the user input.
    Case H will print the help message.
    The default prints the help message.

    If the first bit in the set is 1, Heap Sort will be run.
    Allocate memory for stats and the array using the number of elements.
    Use srandom() to fill it with random numbers.
    Use a for loop to iterate the array and bit mask to 30 bits.
    Reset stats and then call the Heap Sort function.
    Print the results and stats.
    Free stats and the array.

    If the second bit in the set is 1, Batcher Sort will be run.
    Allocate memory for stats and the array using the number of elements.
    Use srandom() to fill it with random numbers.

Use a for loop to iterate the array and bit mask to 30 bits.
Reset stats and then call the Batcher Sort function.
Print the results and stats.
Free stats and the array.

If the third bit in the set is 1, Shell Sort will be run.
Allocate memory for stats and the array using the number of elements.
Use srandom() to fill it with random numbers.
Use a for loop to iterate the array and bit mask to 30 bits.
Reset stats and then call the Shell Sort function.
Print the results and stats.
Free stats and the array.

If the fourth bit in the set is 1, Quick Sort will be run.
Allocate memory for stats and the array using the number of elements.
Use srandom() to fill it with random numbers.
Use a for loop to iterate the array and bit mask to 30 bits.
Reset stats and then call the Quick Sort function.
Print the results and stats.
Free stats and the array.

Return 0.

7 Makefile

Purpose: This is a file that is used to clean, format, and compile the entire program.