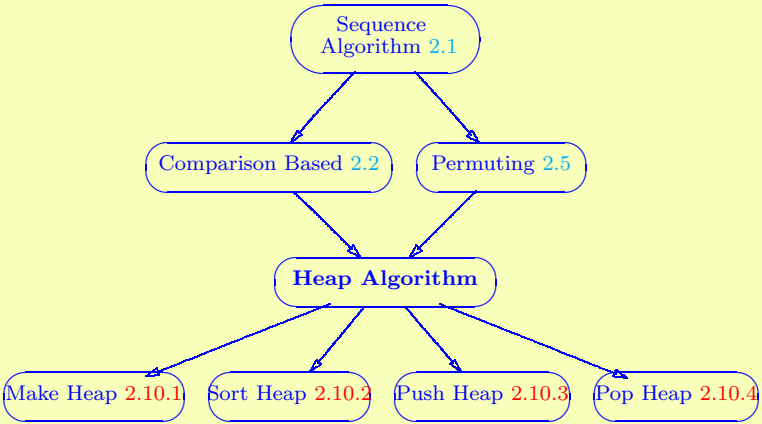
2.10. Heap Algorithm

Section authors: Brad King, Garrett Yaun, and Jeff Banks



A *Heap Algorithm* takes advantage of a heap property in a randomly-accessible sequence of elements. A *heap* represents a particular organization of a random access data structure (2). Given a range $[first, last)$, we say

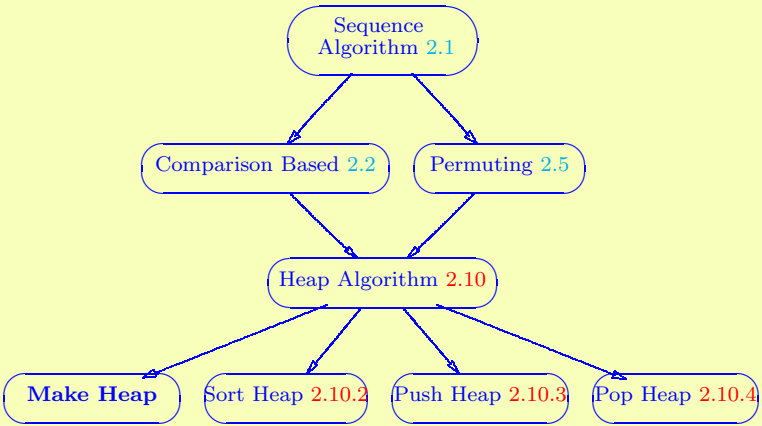
that the range represents a heap if two key properties are satisfied:

- The value pointed to by *first* is the largest value in the range.
- The value pointed to by *first* may be removed by a pop operation or a new element added by a push operation in logarithmic time. Both the pop and push operations return valid heaps.

The largest element is distinguished according to a given *strict-weak-ordering*. How this ordering is defined and provided to the algorithms is an implementation detail and is not covered here.

Refinement of: Comparison Based (§2.2) and Sequence Permuting (§2.5)

2.10.1. Make Heap



Refinement of: Heap Algorithm (§2.10), and therefore of Comparison Based (§2.2) and Sequence Permuting (§2.5)

Prototype: (1) From STL

```
template <class RandomAccessIterator>
void make_heap(RandomAccessIterator first,
               RandomAccessIterator last);
```

Input: Unordered sequence in range $[\text{first}, \text{last})$.

Output: Elements in range $[\text{first}, \text{last})$ form a heap.

Effects: Permutes the input sequence such that the result satisfies the heap property for a particular strict-weak-ordering.

Asymptotic complexity: Let $N = \text{last} - \text{first}$.

- Average case (random data): $O(N)$
- Worst case: $O(N)$

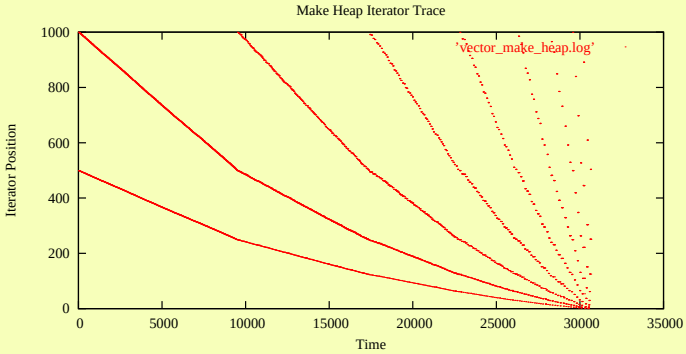
Operation Counts in the Average Case:

N	Comparisons	Assignments	Iterator	Integer
10	11.2	24.2	106.8	159.6
20	27.3	51.7	237.0	345.5
40	59.0	106.4	497.4	722.7
60	92.7	161.4	763.4	1099.3
80	125.9	217.2	1033.4	1488.3
160	255.4	435.1	2080.8	2994.5
320	520.1	877.5	4218.8	6042.2
640	1044.6	1752.4	8444.2	12084.9

Value Comparisons (§A.1.1) : $1.65N - 0.69 \lg N - 3.02$
Value Assignments (§A.1.2) : $2.76N - 0.46 \lg N - 2.41$
Iterator Operations (§A.1.3) : $13.32N - 5.03 \lg N - 11.80$
Integer Operations (§A.1.4) : $19.03N - 5.41 \lg N - 12.51$

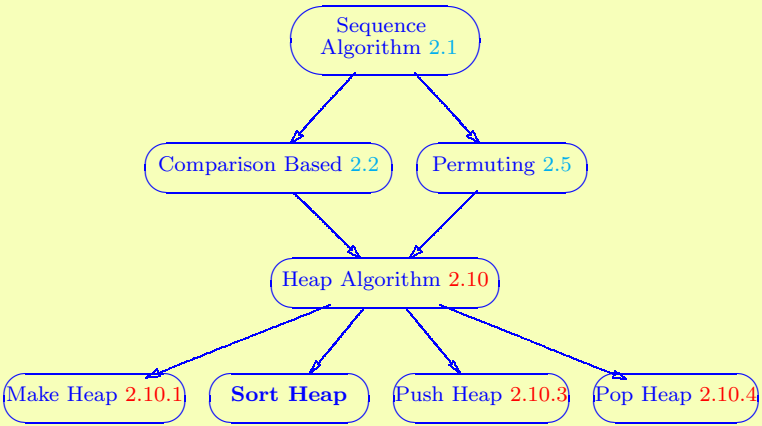
See Appendix (§A) for a description of how the data were collected and processed to produce the above equations.

Iterator Trace:



An unordered sequence of 1000 elements is converted into a heap. Make-heap is implemented by heapifying and combining small sub-heaps into a final single heap. At any time t, the number of iterator trails intersecting with a vertical line indicates the height of the heaps currently getting built. At about time 9500, all heaps of height 1 have been built. At about time 17500, all heaps of height 2 have been built. The process continues until the heap of height $\lceil \lg 1000 \rceil$ have been built.

2.10.2. Sort Heap



Refinement of: Heap Algorithm (§2.10), and therefore of Comparison Based (§2.2) and Sequence Permuting (§2.5)

Prototype: (1) From STL

```
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first,
               RandomAccessIterator last);
```

Input: Elements in range $[\text{first}, \text{last})$ form a heap.

Output: Elements in range $[\text{first}, \text{last})$ are in sorted order.

Effects: Permutes the input heap such that the elements are sorted according to the same strict-weak-ordering used to define the original heap. The result may no longer be a heap.

Asymptotic complexity: Let $N = \text{last} - \text{first}$.

- Average case (random data): $O(N \lg N)$
- Worst case: $O(N \lg N)$

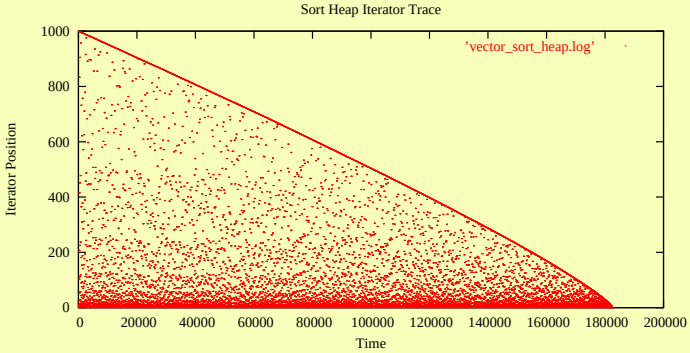
Operation Counts in the Average Case:

N	Comparisons	Assignments	Iterator	Integer
10	20.4	60.0	288.8	288.1
20	60.9	142.0	735.6	749.2
40	161.4	322.7	1775.2	1834.2
60	278.4	519.9	2942.4	3059.6
80	403.8	726.8	4187.4	4382.2
160	968.8	1613.2	9643.6	10171.0
320	2254.1	3537.9	21796.4	23183.9
640	5158.9	7724.1	48755.6	52131.1

Value Comparisons (§A.2.1) : $0.98N \lg N - 1.05N - 16.35 \lg N + 98.57$
Value Assignments (§A.2.2) : $0.98N \lg N + 2.95N - 15.32 \lg N + 94.81$
Iterator Operations (§A.2.3) : $7.87N \lg N + 3.26N - 126.37 \lg N + 782.24$
Integer Operations (§A.2.4) : $8.81N \lg N - 0.03N - 175.06 \lg N + 1071.58$

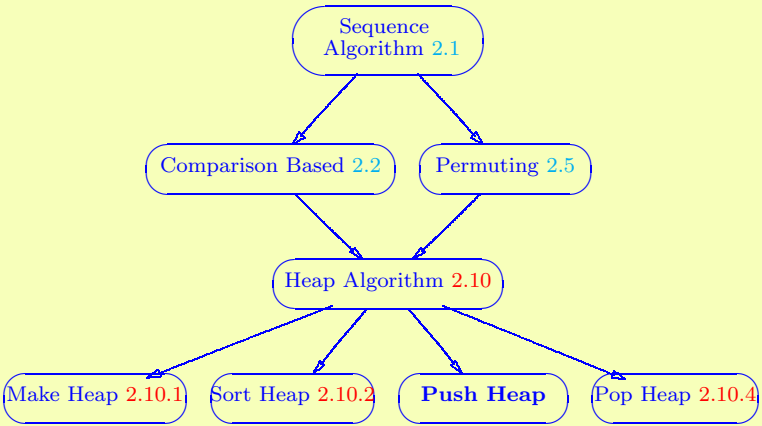
See Appendix (§A) for a description of how the data were collected and processed to produce the above equations.

Iterator Trace:



A heap of 1000 elements is being sorted. Sort-heap is implemented by repeatedly performing a pop-heap until the heap is empty. The trace simply shows the iterator trace for each pop-heap in turn. Since the heap is smaller by one element after each pop-heap, the highest iterator position steadily decreases with time.

2.10.3. Push Heap



Refinement of: Heap Algorithm (§2.10), and therefore of Comparison Based (§2.2) and Sequence Permuting (§2.5)

Prototype: (1) From STL

```
template <class RandomAccessIterator>
void push_heap(RandomAccessIterator first,
               RandomAccessIterator last);
```

Input: Elements in range $[\text{first}, \text{last} - 1)$ form a heap. Element at position $\text{last} - 1$ is to be inserted.

Output: Elements in range $[\text{first}, \text{last})$ form a heap.

Effects: Inserts a new value into the heap.

Asymptotic complexity: Let $N = \text{last} - \text{first}$.

- Average case (random data): $O(\lg N)$
- Worst case: $O(\lg N)$

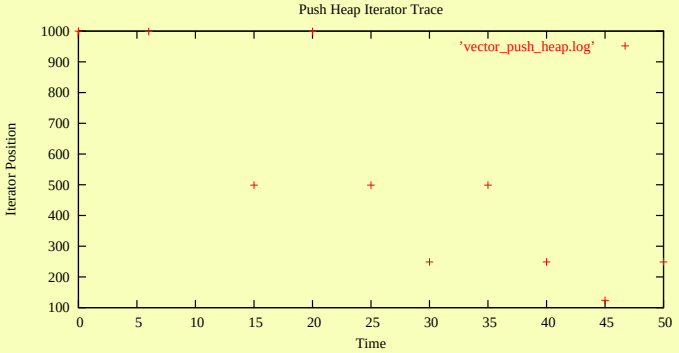
Operation Counts in the Average Case:

N	Comparisons	Assignments	Iterator	Integer
10	4.8	9.3	47.6	50.6
20	5.6	9.6	52.6	56.0
40	6.7	10.7	61.2	64.4
60	7.2	11.4	65.8	71.4
80	6.4	10.4	61.0	65.4
160	7.7	11.7	71.4	78.2
320	8.9	12.9	80.2	87.0
640	10.9	14.9	94.6	102.2

Value Comparisons (§A.3.1) : $1.02 \lg N + 0.84$
Value Assignments (§A.3.2) : $1.02 \lg N + 4.89$
Iterator Operations (§A.3.3) : $8.10 \lg N + 15.43$
Integer Operations (§A.3.4) : $9.07 \lg N + 14.18$

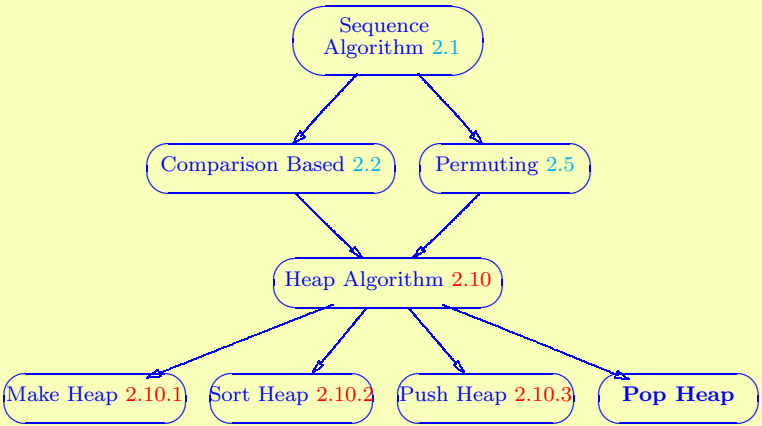
See Appendix (§A) for a description of how the data were collected and processed to produce the above equations.

Iterator Trace:



An element at position 999 is inserted into the existing heap of 999 elements in the range [0, 999). The iterator trace shows the progress of walking up the heap until the proper location is found. At time 6, the value x to be inserted is read. At time 15, x's parent node at position 499 is read and its value is compared to x. A violation of the heap property is detected, and fixing it accesses the positions again at times 20 and 25. The parent of node 499 then compared against x. This process continues until the proper position is found at node 249, and x is assigned to this node at time 50.

2.10.4. Pop Heap



Refinement of: Heap Algorithm (§2.10), and therefore of Comparison Based (§2.2) and Sequence Permuting (§2.5)

Prototype: (1) From STL

```
template <class RandomAccessIterator>
void pop_heap(RandomAccessIterator first,
              RandomAccessIterator last);
```

Input: Elements in range $[\text{first}, \text{last})$ form a heap.

Output: Elements in range $[\text{first}, \text{last}-1)$ form a heap. Value that was previously at first has been removed from the heap and placed at $\text{last} - 1$.

Effects: Removes the value from the heap that is considered largest by the strict-weak-ordering.

Asymptotic complexity: Let $N = \text{last} - \text{first}$.

- Average case (random data): $O(\lg N)$
- Worst case: $O(\lg N)$

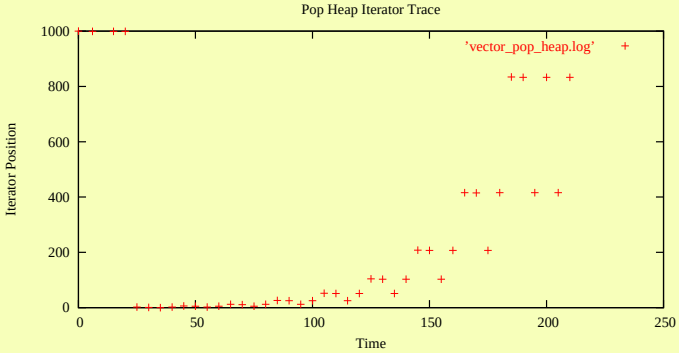
Operation Counts in the Average Case:

N	Comparisons	Assignments	Iterator	Integer
10	3.7	7.7	39.4	42.4
20	4.7	8.7	47.4	51.4
40	5.5	9.5	53.8	58.0
60	6.3	10.3	59.8	64.2
80	6.4	10.4	61.0	64.8
160	7.5	11.5	70.0	76.4
320	8.5	12.5	77.6	85.0
640	9.6	13.6	86.0	92.6

Value Comparisons (§A.4.1) : $0.99 \lg N + 0.29$
Value Assignments (§A.4.2) : $0.99 \lg N + 4.29$
Iterator Operations (§A.4.3) : $7.95 \lg N + 12.00$
Integer Operations (§A.4.4) : $8.92 \lg N + 10.73$

See Appendix (§A) for a description of how the data were collected and processed to produce the above equations.

Iterator Trace:



The root element is removed from a heap of 1000 elements and placed at the end of the sequence. Through time 25, the algorithm is reading the old value x from position 999 and copying the root element from position 0 to position 999. It then starts at the empty root element, and moves this hole down the heap by repeatedly selecting the greater of the two children and copying it to its parent node. At time 200, the bottom of the heap is reached. The last three accesses correspond to the push-heap operation of element x starting at the hole at the bottom of the heap.

A. Run-Time Analysis

The STL (1) implementations of all four algorithms from the library of GCC 2.95.3 were used to measure operation counts. Each algorithm was run on N randomly generated elements, where N ranged from 10 to 10000, in steps of 10. There were 10 independent trials for each value of N. For each type of operation, the average of the counts across all 10 trials was recorded as the count for that operation.

The following types of operations were recorded using the operation counting library:

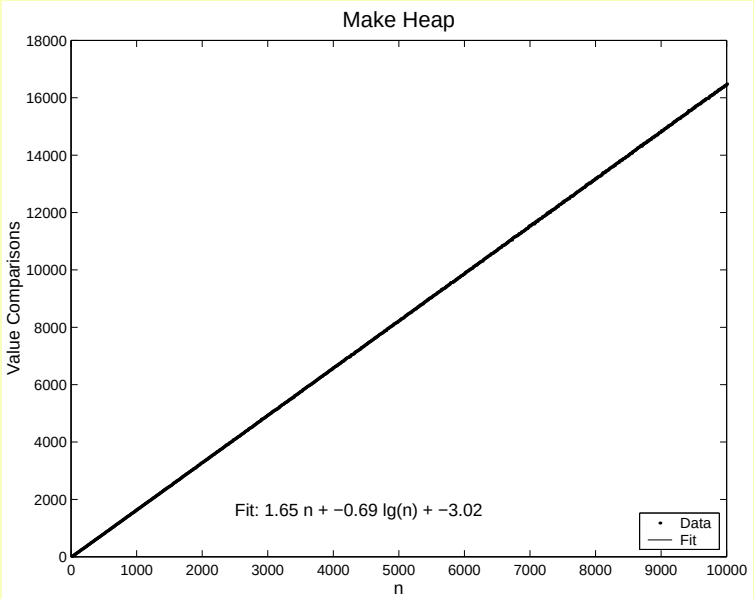
Category:	**Operations Used:**
Value Comparisons:	<
Value Assignments:	= and Copy-Construction
Iterator Operations:	Total `iterator_counter`
Integer Operations:	Total `difference_counter`

Using the average operation totals for each N, a least-squares fit was performed to obtain the constants in the running-time equations. There were four algorithms, and four operation categories in each, for a total of

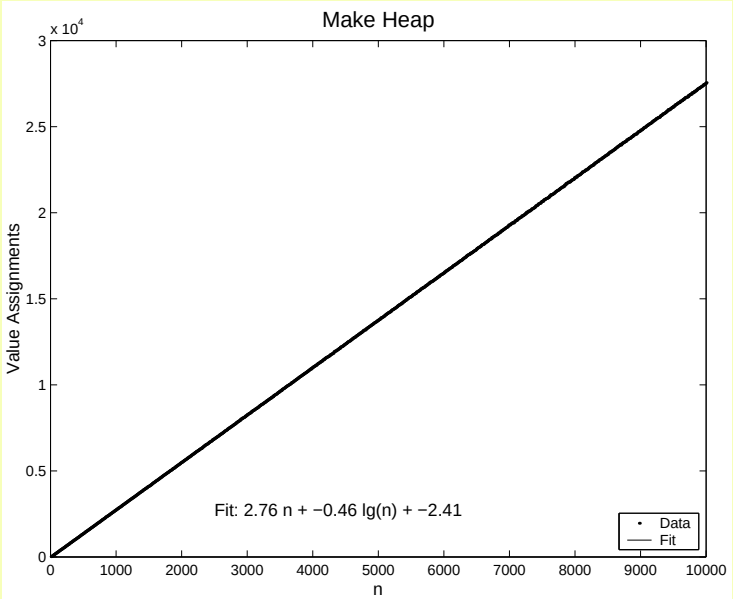
sixteen least-squares fit operations. The remaining sections show the plots of each set of data with its corresponding fit.

A.1. Make Heap Fits

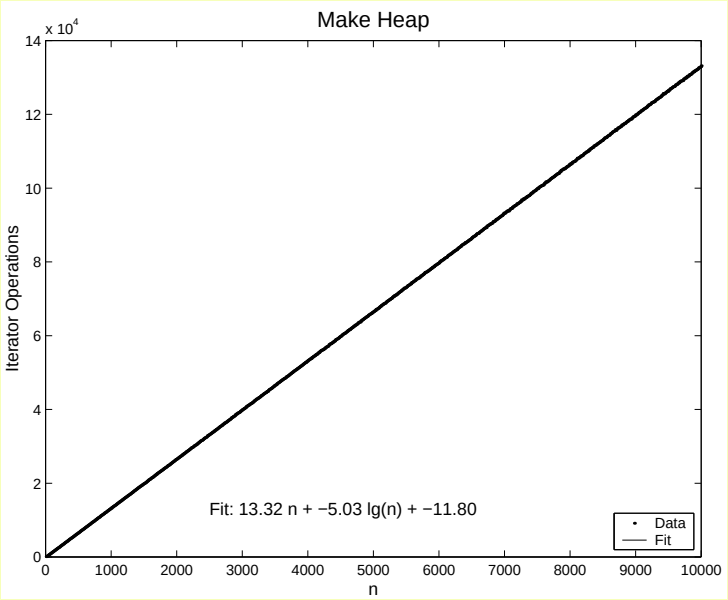
A.1.1. Make Heap Value Comparisions



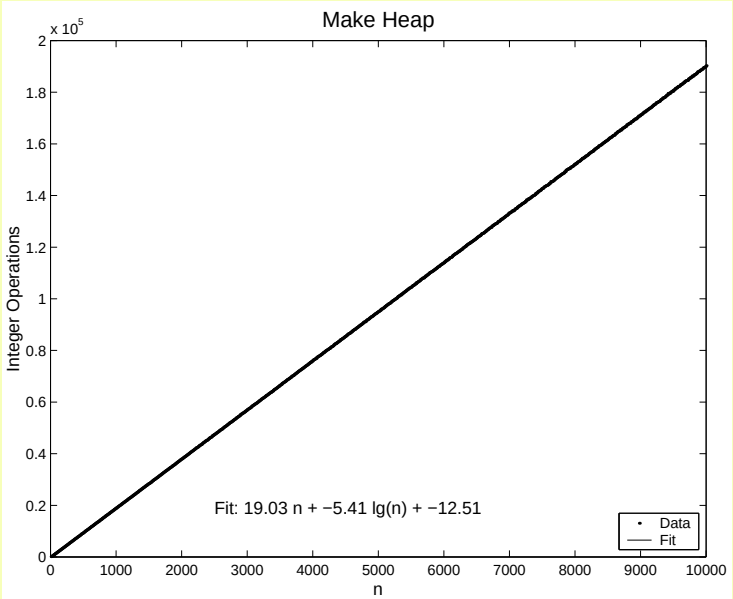
A.1.2. Make Heap Value Assignments



A.1.3. Make Heap Iterator Operations

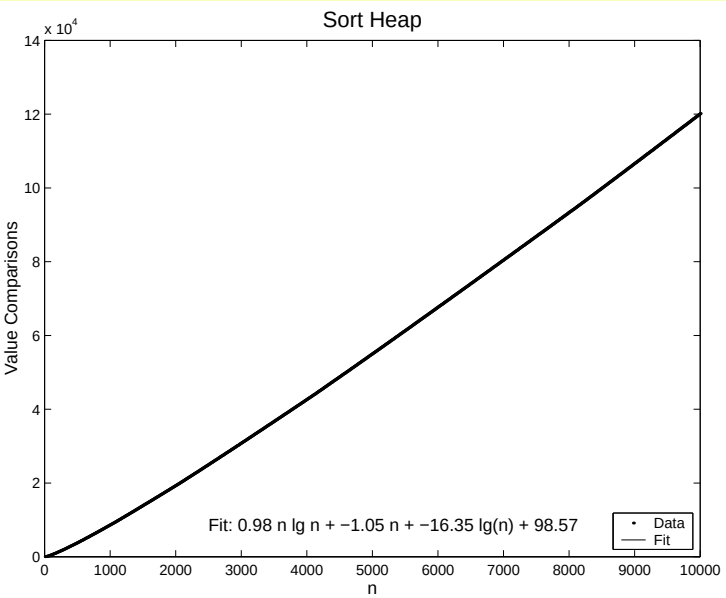


A.1.4. Make Heap Integer Operations

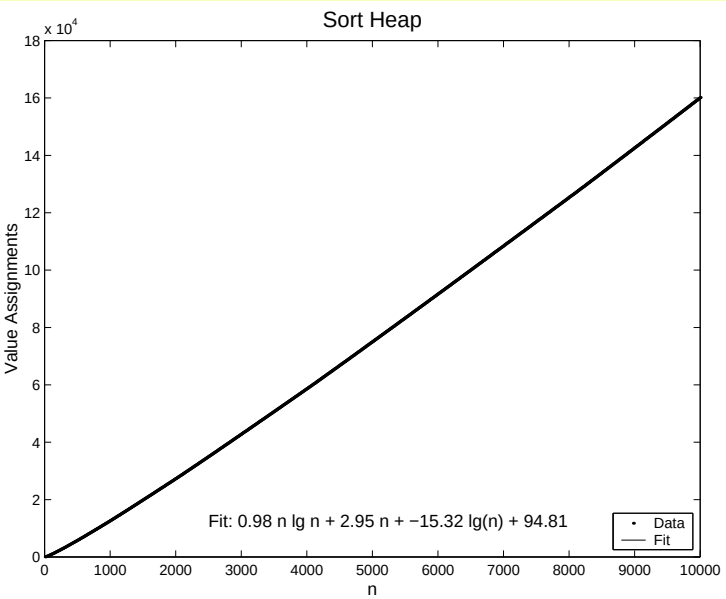


A.2. Sort Heap Fits

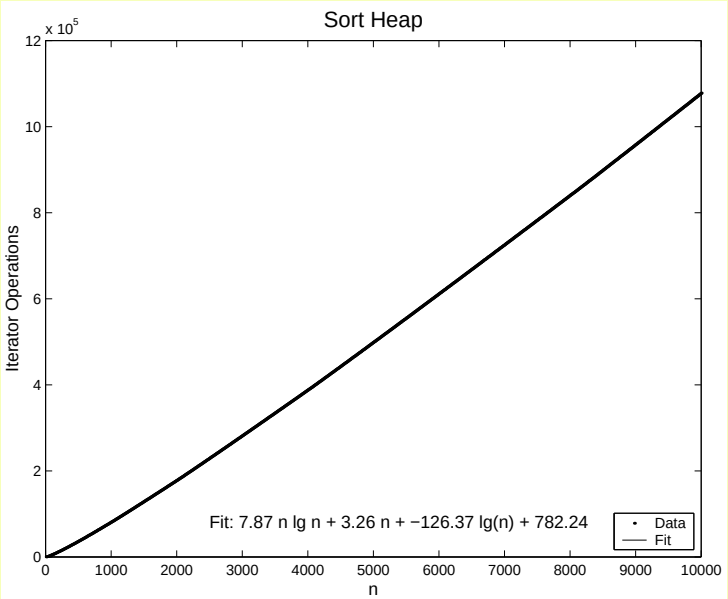
A.2.1. Sort Heap Value Comparisions



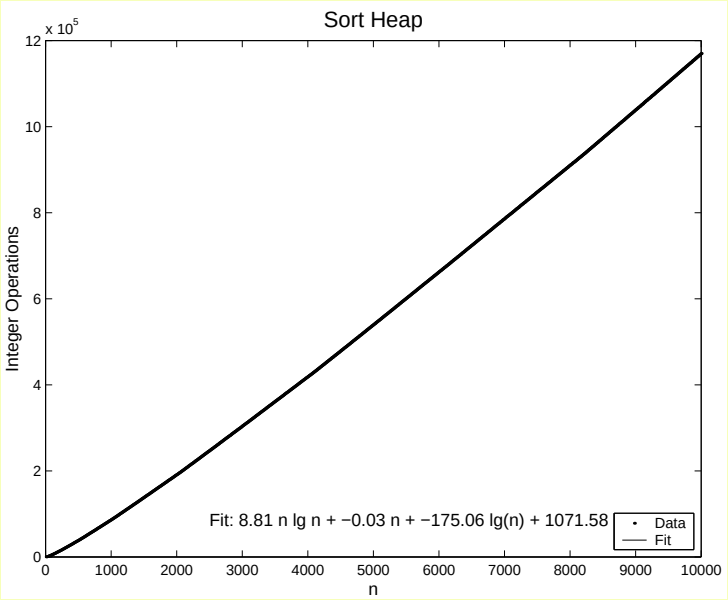
A.2.2. Sort Heap Value Assignments



A.2.3. Sort Heap Iterator Operations

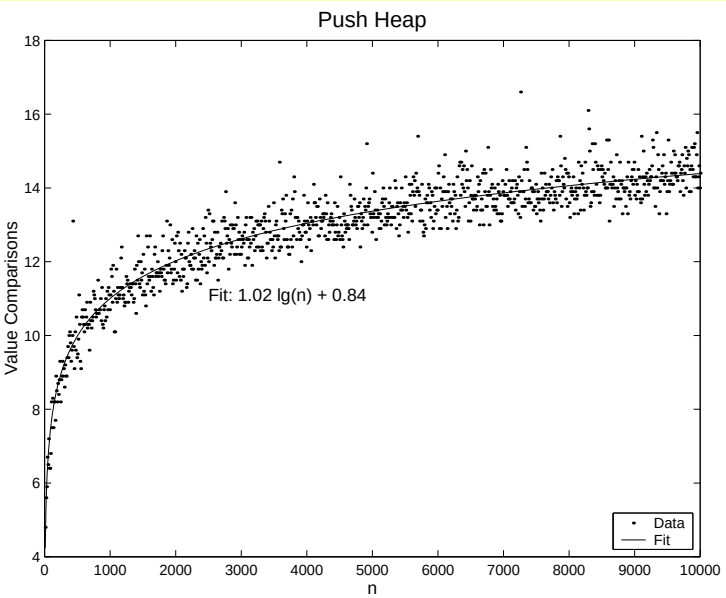


A.2.4. Sort Heap Integer Operations

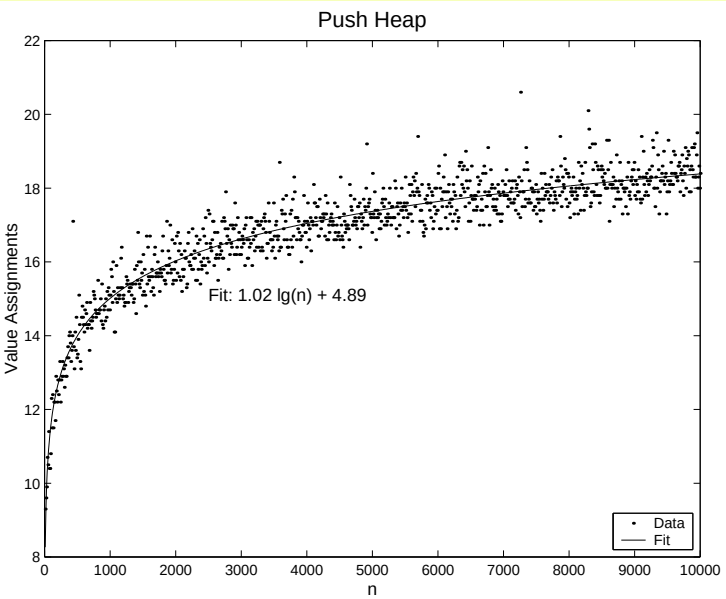


A.3. Push Heap Fits

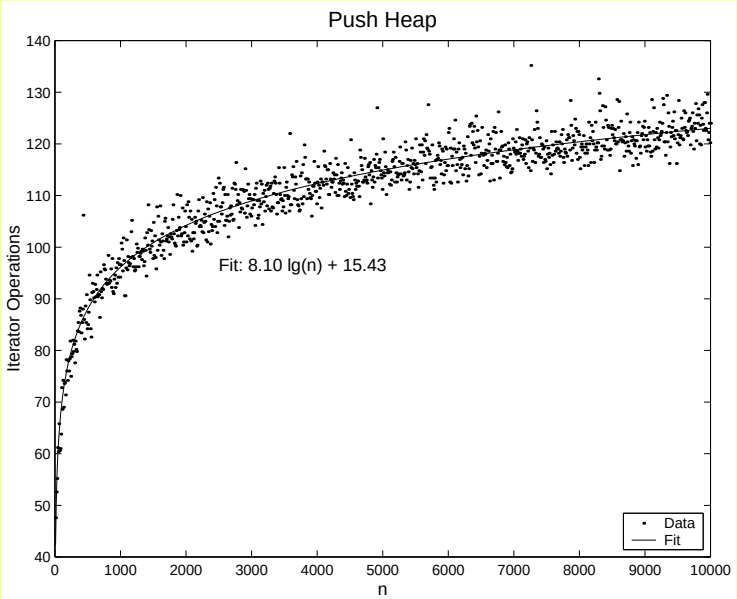
A.3.1. Push Heap Value Comparisions



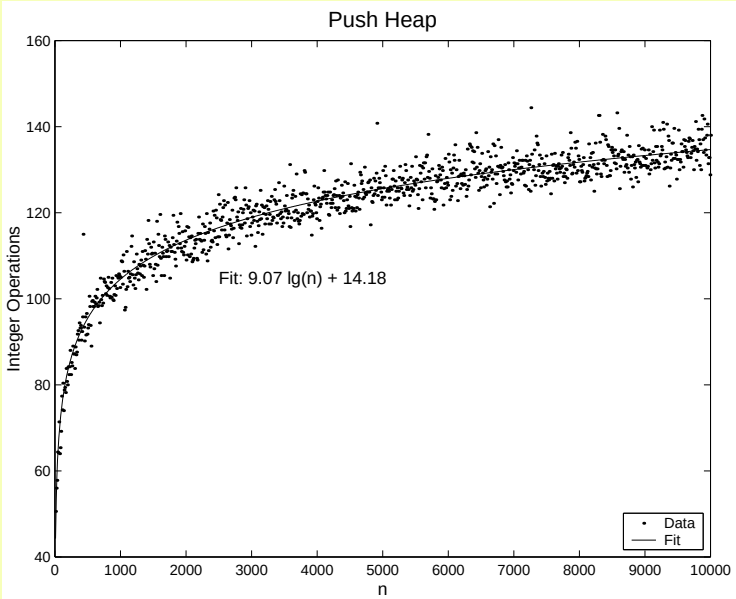
A.3.2. Push Heap Value Assignments



A.3.3. Push Heap Iterator Operations

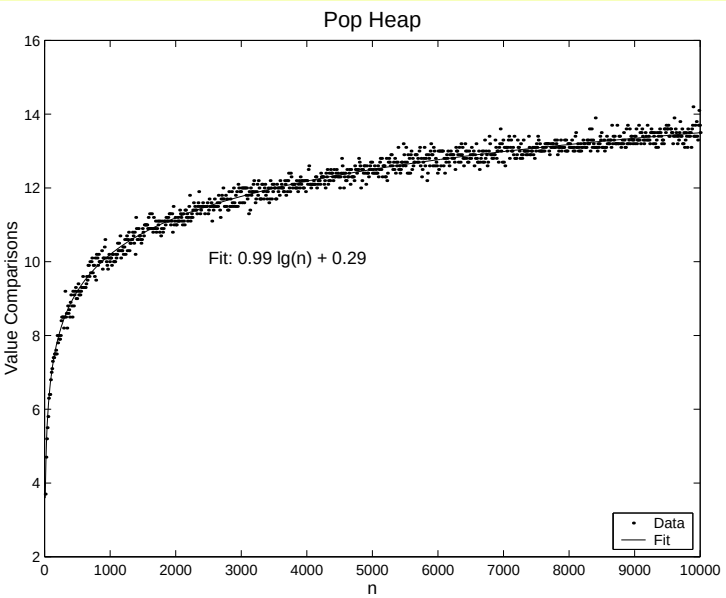


A.3.4. Push Heap Integer Operations

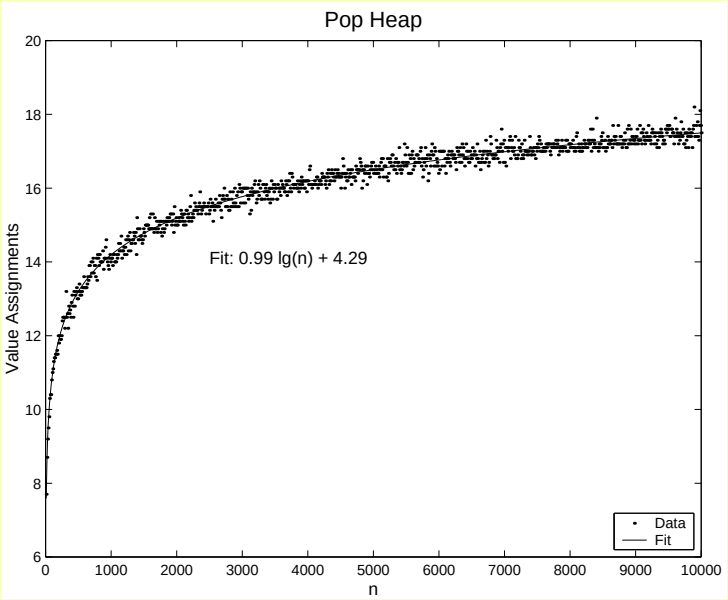


A.4. Pop Heap Fits

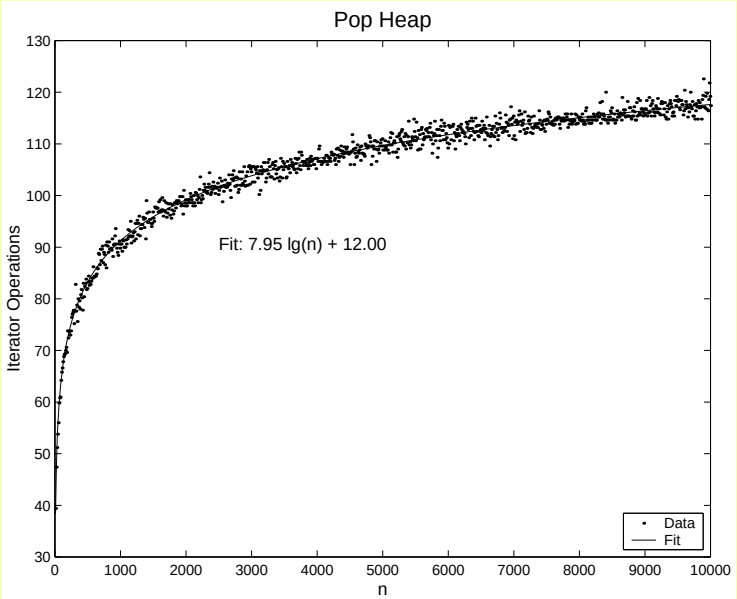
A.4.1. Pop Heap Value Comparisions



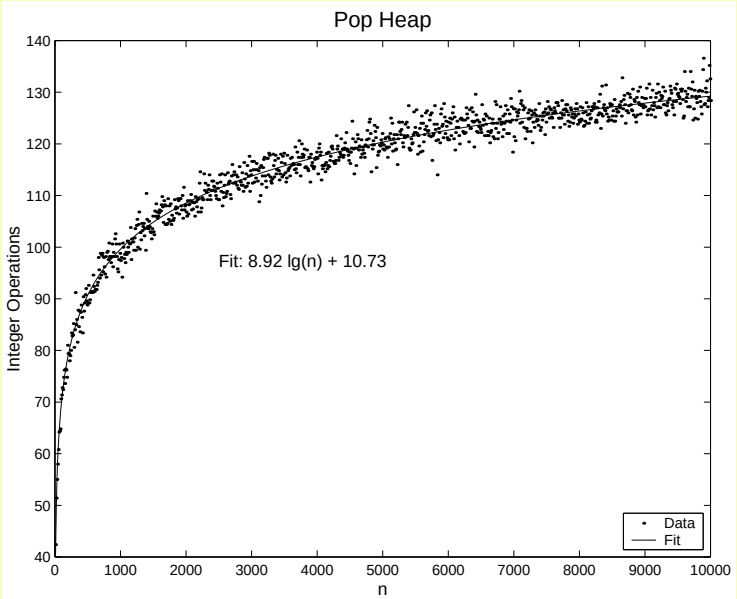
A.4.2. Pop Heap Value Assignments



A.4.3. Pop Heap Iterator Operations



A.4.4. Pop Heap Integer Operations



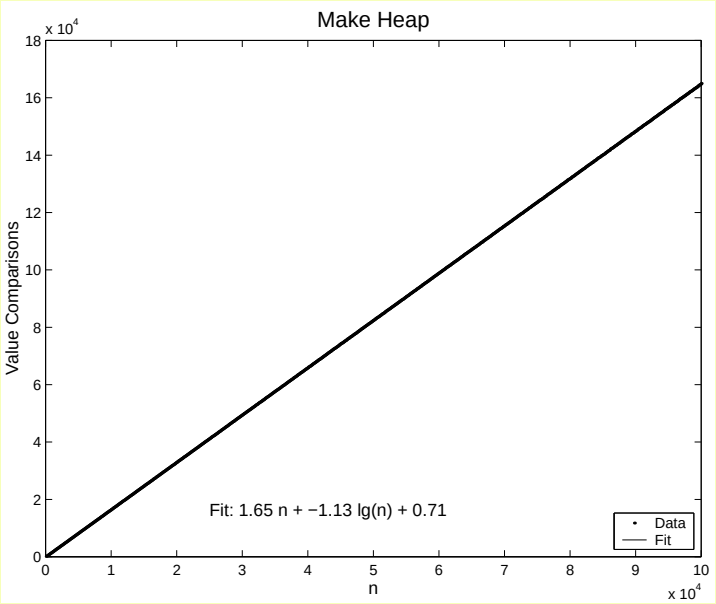
B. Analysis with Larger N

All the tests, data collection, and analyses that were performed as described in Appendix (§A) used a range of sequence sizes from 10 to 10000, in steps of 10. The entire process was repeated for sequence sizes ranging from 100 to 100000, in steps of 100. The results of this analysis are shown in this section.

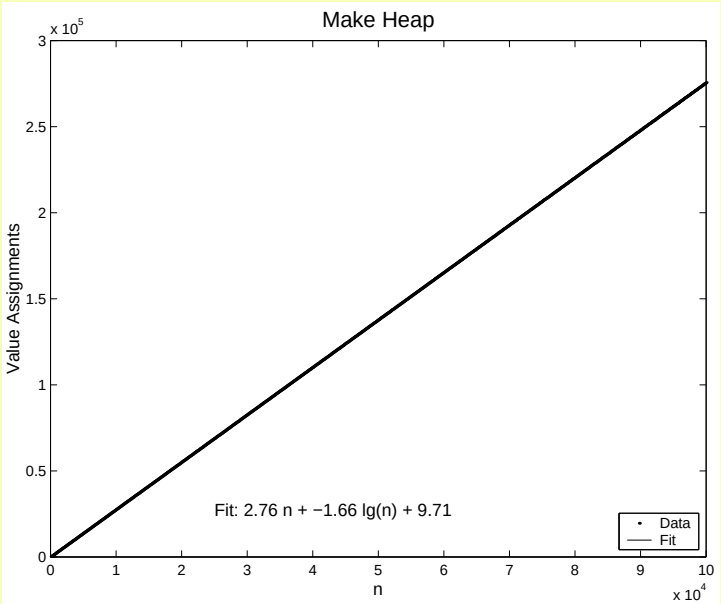
We decided to favor the results from the analysis of the smaller data sets because we believe that the large constants that appear in this section are the result of over-fitting by using too many terms. This is espe-ically noticeable for the sort-heap fits. Further analysis with correlation coefficients could be used to reveal the terms that are actually important for each fit, but such analysis is beyond the scope of this course.

B.1. Make Heap Fits

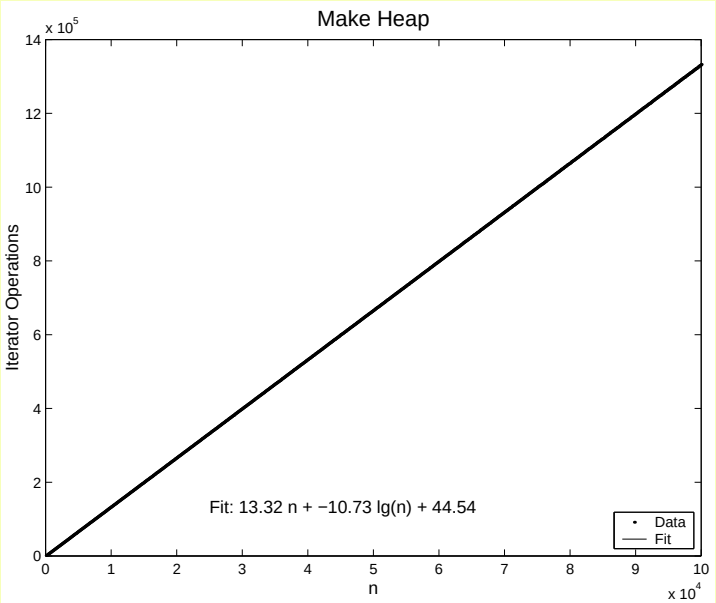
B.1.1. Make Heap Value Comparisions



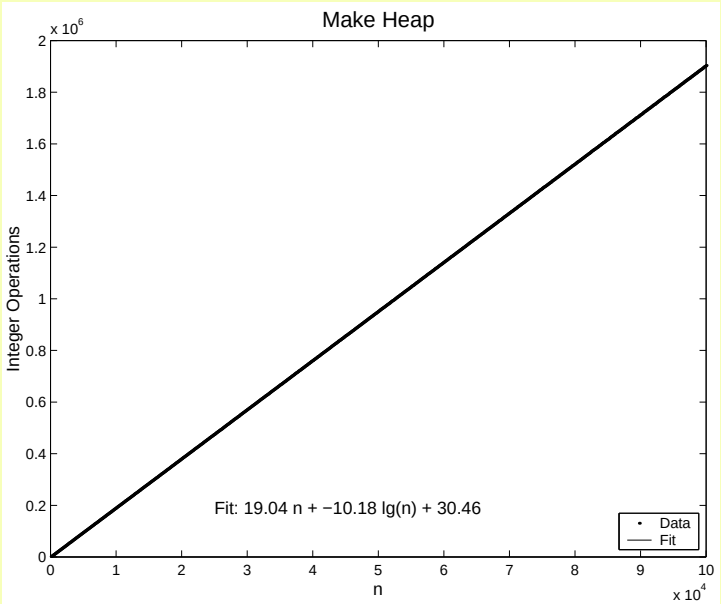
B.1.2. Make Heap Value Assignments



B.1.3. Make Heap Iterator Operations

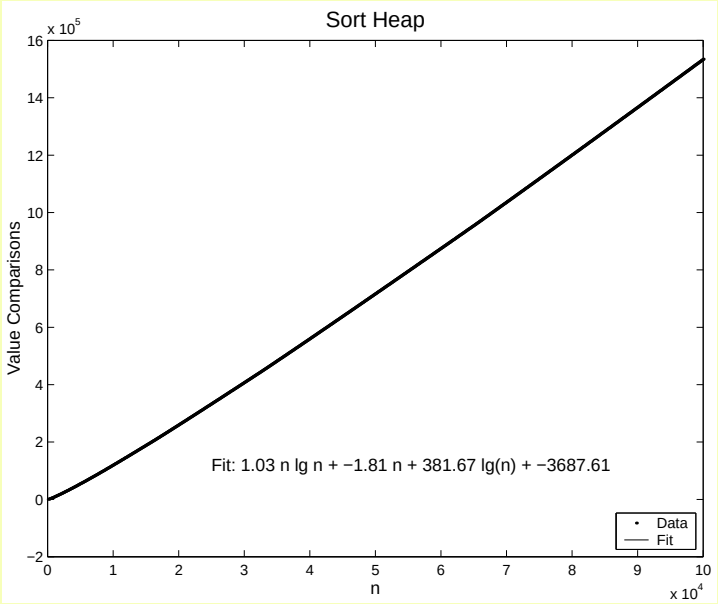


B.1.4. Make Heap Integer Operations

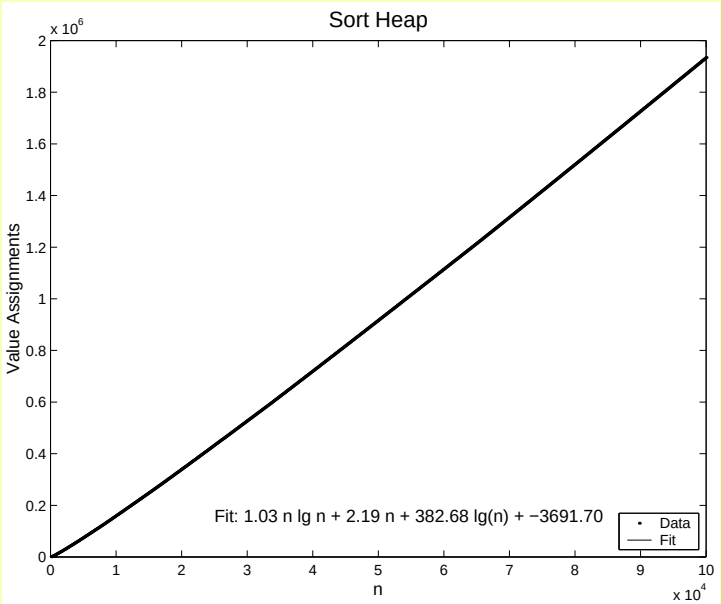


B.2. Sort Heap Fits

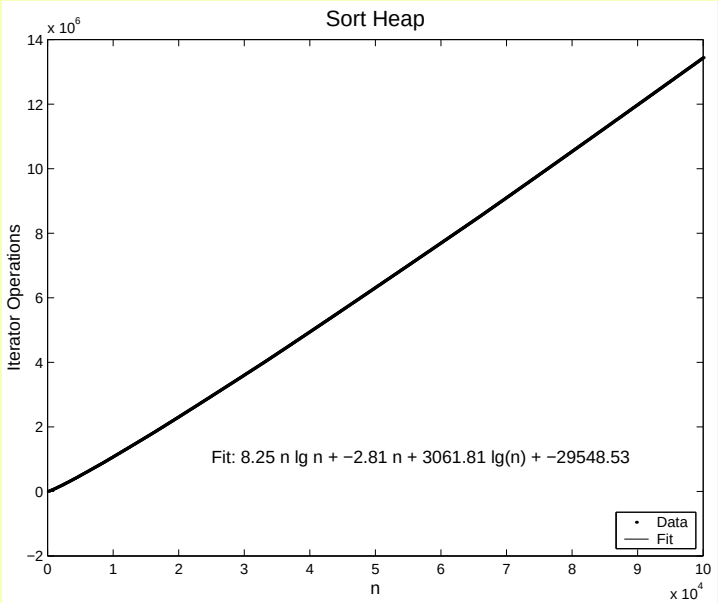
B.2.1. Sort Heap Value Comparisions



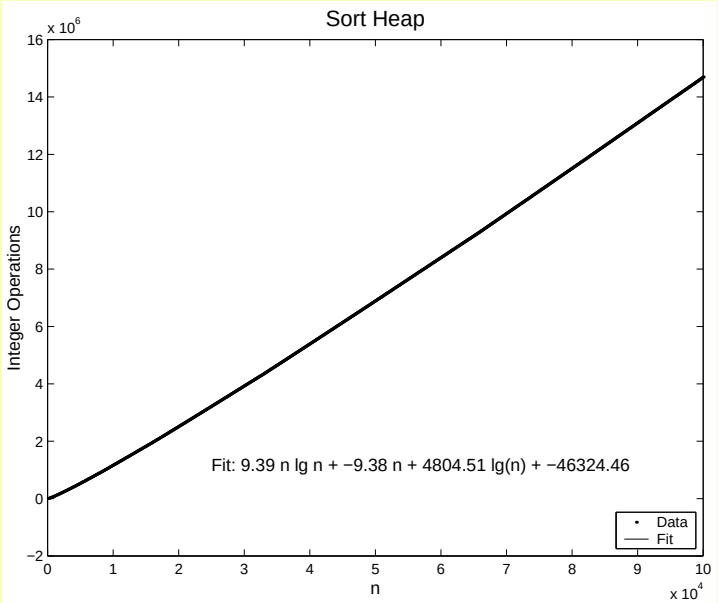
B.2.2. Sort Heap Value Assignments



B.2.3. Sort Heap Iterator Operations

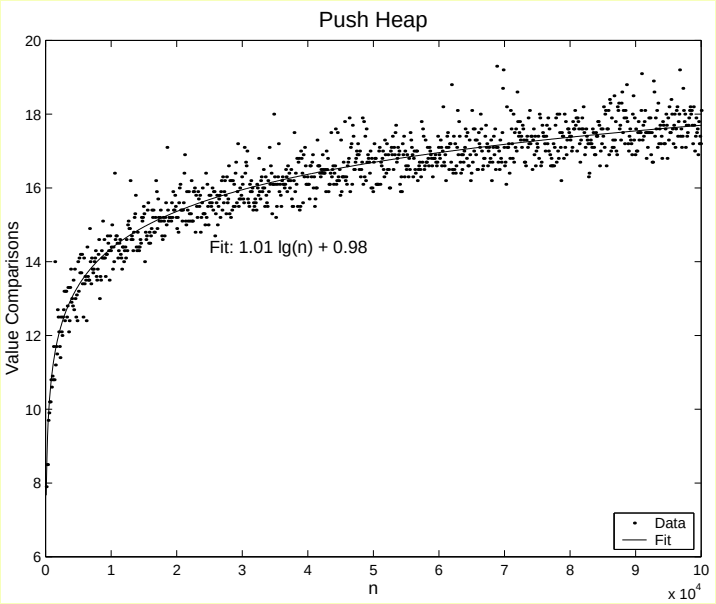


B.2.4. Sort Heap Integer Operations

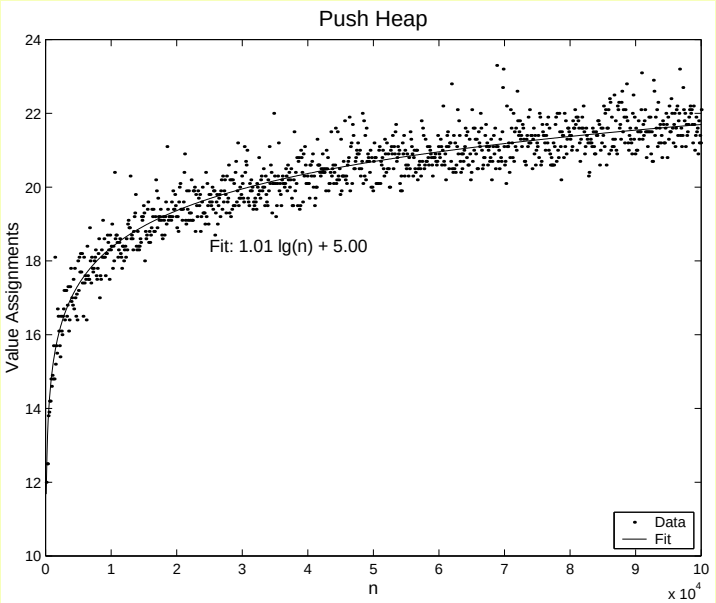


B.3. Push Heap Fits

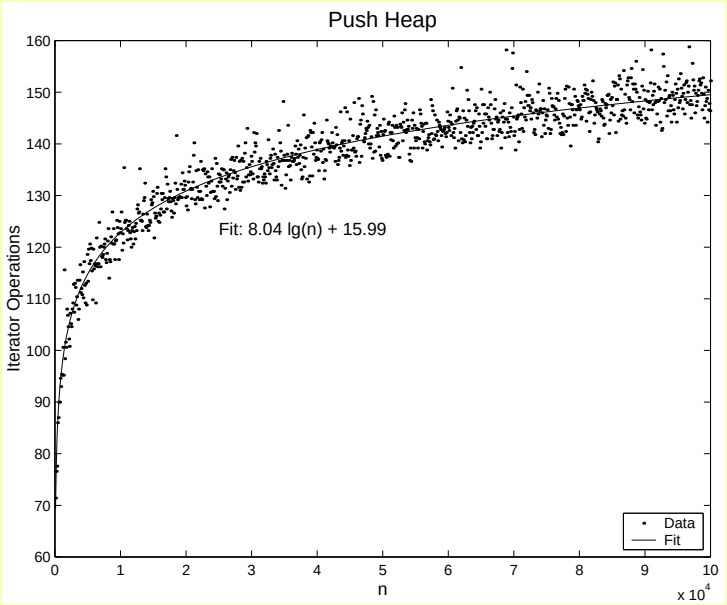
B.3.1. Push Heap Value Comparisions



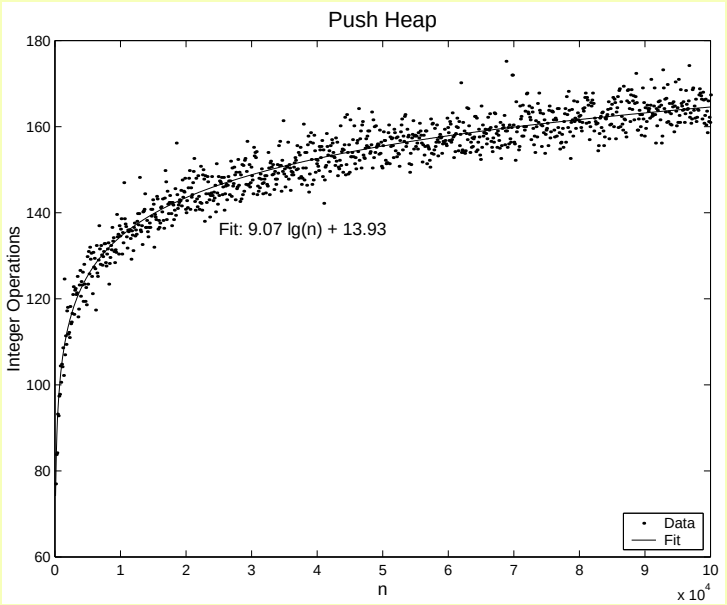
B.3.2. Push Heap Value Assignments



B.3.3. Push Heap Iterator Operations

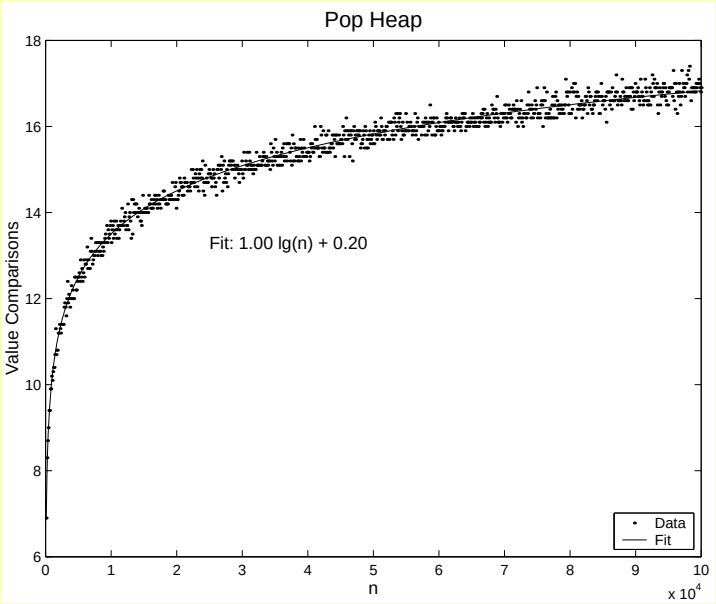


B.3.4. Push Heap Integer Operations

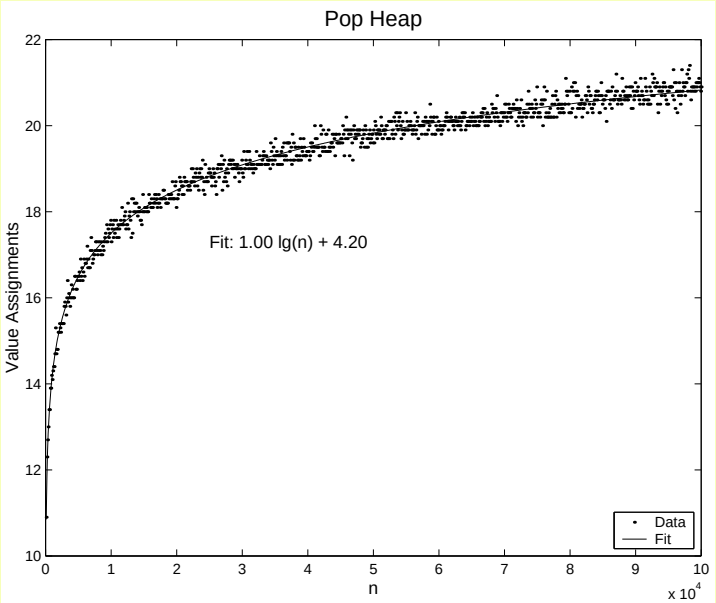


B.4. Pop Heap Fits

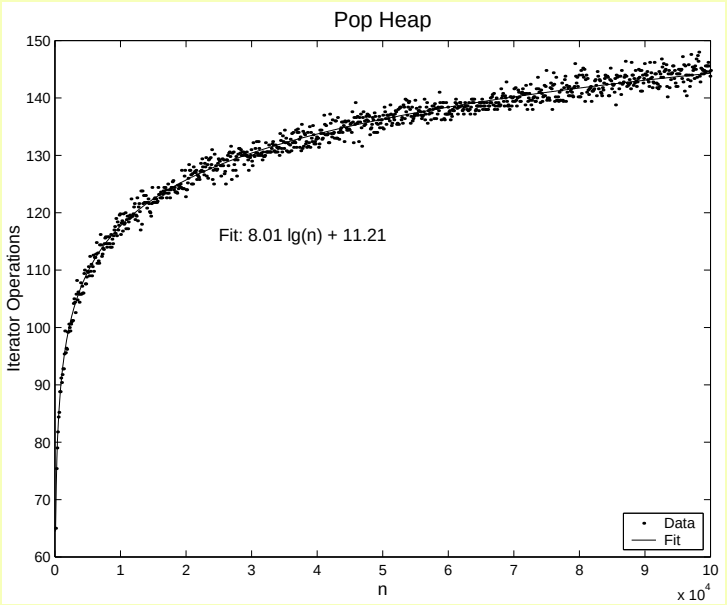
B.4.1. Pop Heap Value Comparisions



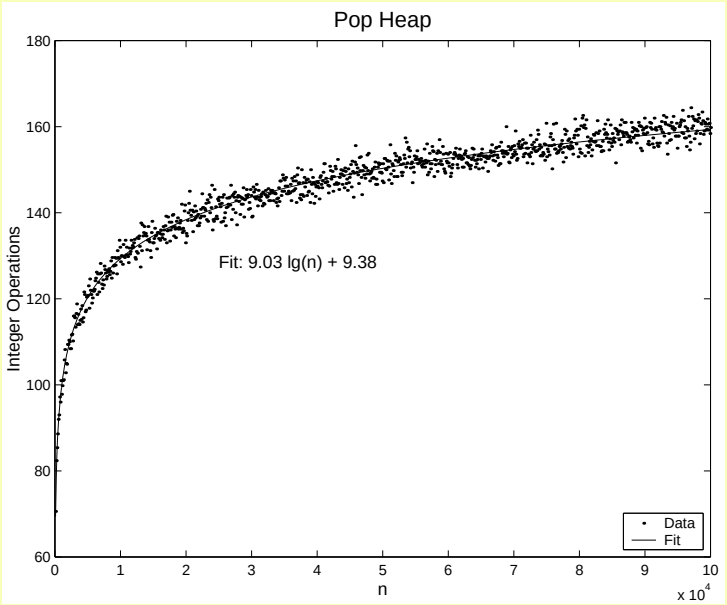
B.4.2. Pop Heap Value Assignments



B.4.3. Pop Heap Iterator Operations



B.4.4. Pop Heap Integer Operations



References

[1] SGI Standard Template Library Reference
http://www.sgi.com/tech/stl

[2] David Musser, *STL Tutorial and Reference Guide*, Addison-Wesley, Reading, MA, 1997.