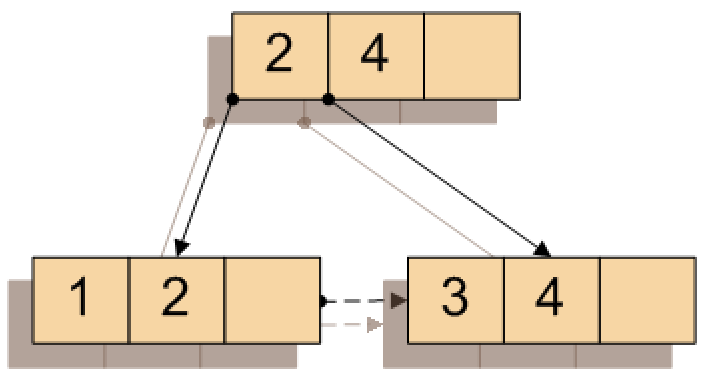
A Symmetric Concurrent B-Tree Algorithm

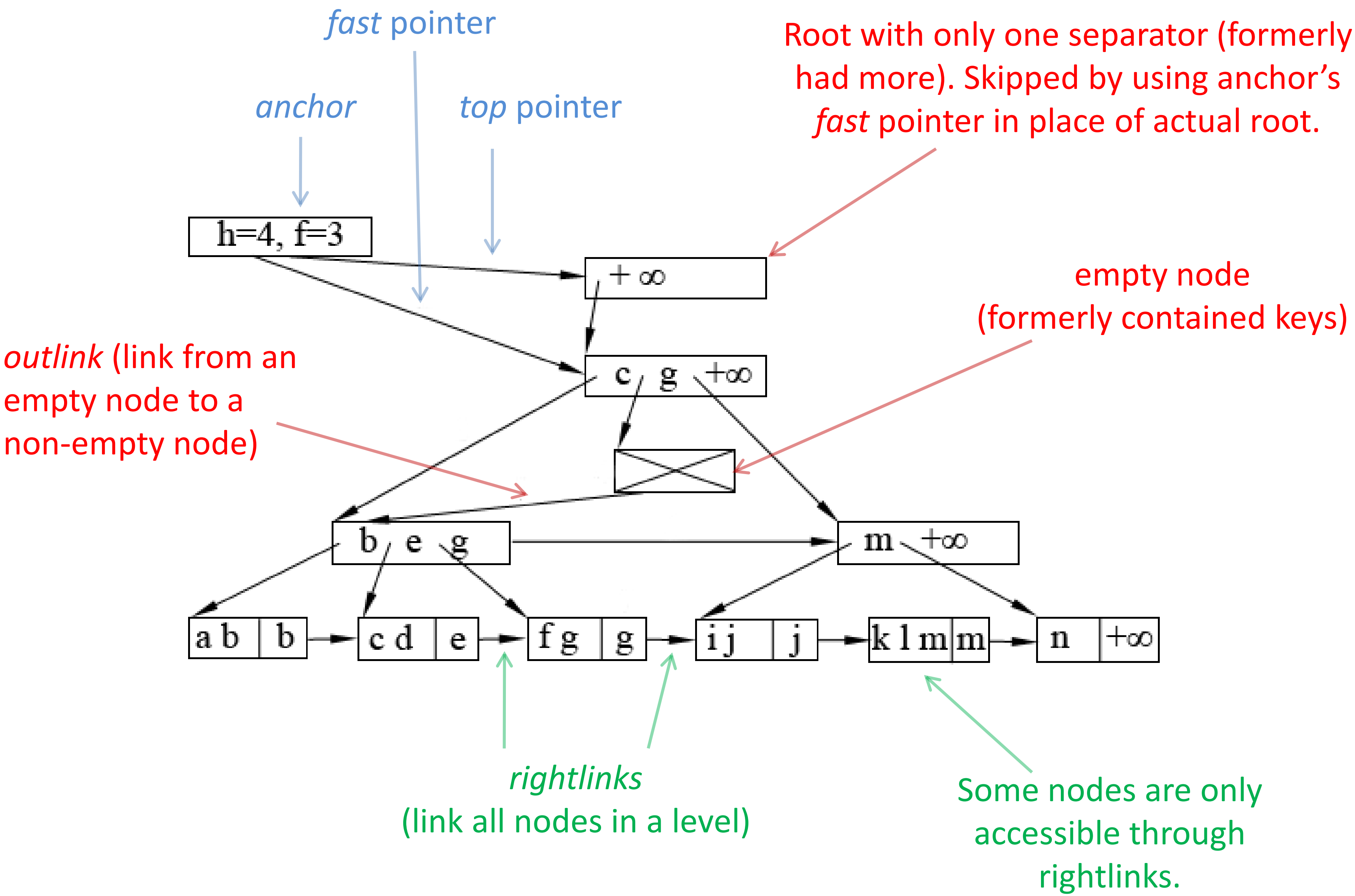
Part 2: Implementation

Overview: B-Link Tree

- Introduced by Lehman & Yao, 1981. Merge operation (to balance after deletions) added by Lanin & Shasha, 1986.
- Like a B+ tree, actual keys are in leaf nodes. “Separators” in internal nodes are for navigation only.
- Contains extra links (“rightlinks” and “outlinks”) so that threads searching for something will still be able to find it if the tree is reconfigured during their search.
  - This allows it to use less locking than other types of concurrent B-tree algorithm.



Example of a B-Link Tree (diagram from Lanin & Shasha):



Structure Properties

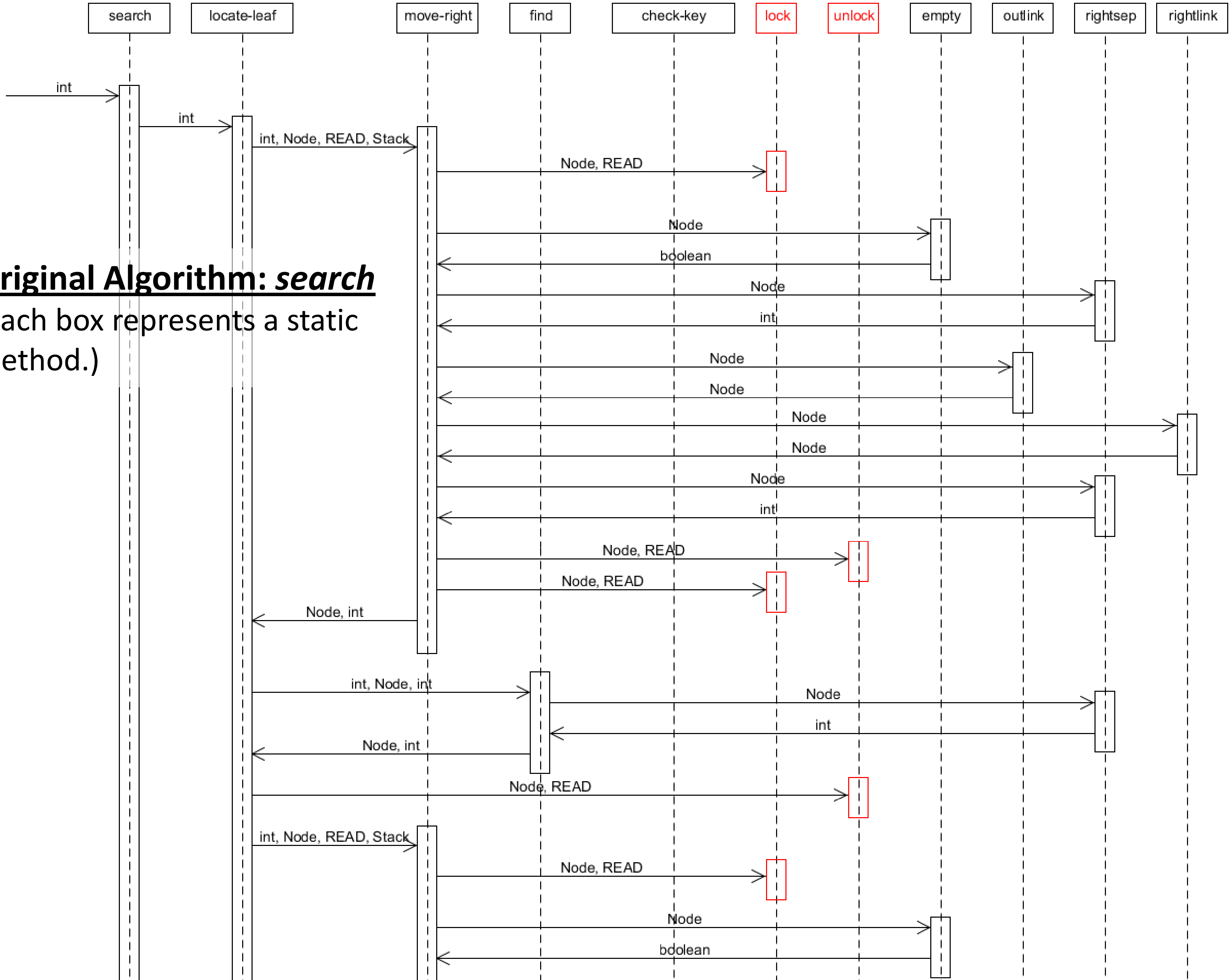
- Form is less strict than a normal B-tree or B+ tree.
- Most important properties:
  - All keys in each level must be sorted.
  - All leaves must be on the same level.
  - No key can move left after tree modification, or appear to the left of where an above separator indicates.
  - If a key is located (or moved) farther to the right, it can be found by following rightlinks.
- A search can be started at any level.
  - Starting at the leaf level is equivalent to searching a sorted list.

Implementation

- The original algorithm, written in 1986 with Pascal in mind, is structured in a very non-object oriented way.
  - E.g. Static methods call other static methods which in turn call other static methods, with enumerator parameters that choose their behavior.
- My implementation is a combination of object-oriented and non-object-oriented techniques.

Original Algorithm: *search*

(Each box represents a static method.)



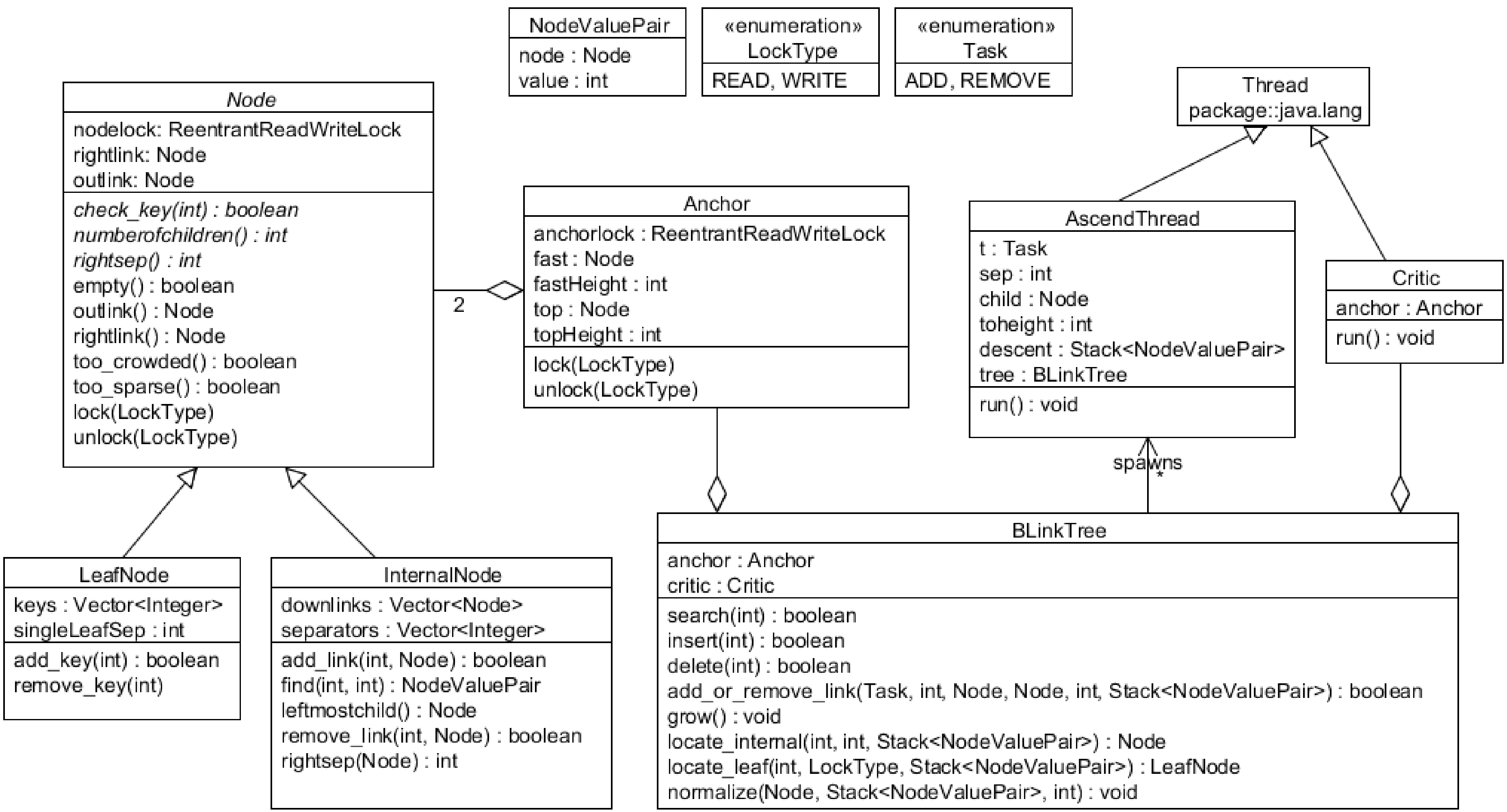
Java Implementation

- Many methods in the original algorithm take only a Node, or Node & lock type, as a parameter.
- For my implementation, these were assembled into Node objects.
- Most other static methods were placed in a BLinkTree object.

Implementation: Concurrency

- The algorithm details a fine-grained locking scheme, using read and write locks.
  - Recall: The purpose of the algorithm is to minimize locking.
- This was translated directly to Java using *ReentrantReadWriteLock*.
- *CyclicBarrier* was used to coordinate threads during testing.
- *Synchronized* methods were used for things external to the algorithm itself.
  - E.g. Writing to *System.out*, verifying correctness of the structure.

Java Implementation: Classes
(shows methods from original algorithm only)



Correctness Testing

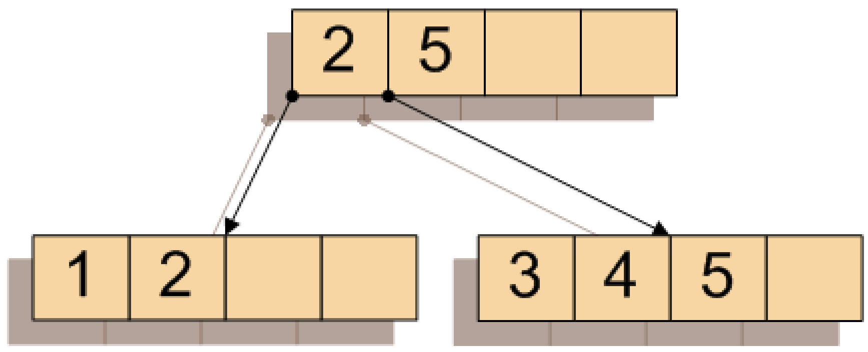
- The most important factor is that every item in the tree can be found at all stages of the tree's life, including while it is being modified by other threads.
- Each test run was started with a set of odd numbers inserted into the tree.
- About 50% of the threads searched for these numbers during initial testing and during the performance tests.
- Insertion and deletion were made to only modify even numbers, so that the search threads would always find what they were looking for, provided the code was correct.
- Any time a search thread did *not* find a key, it kept a record of this.
- The total number of keys not found was counted after each test.
- After initial debugging was complete, every single test run finished without any un-found keys.

Performance Testing

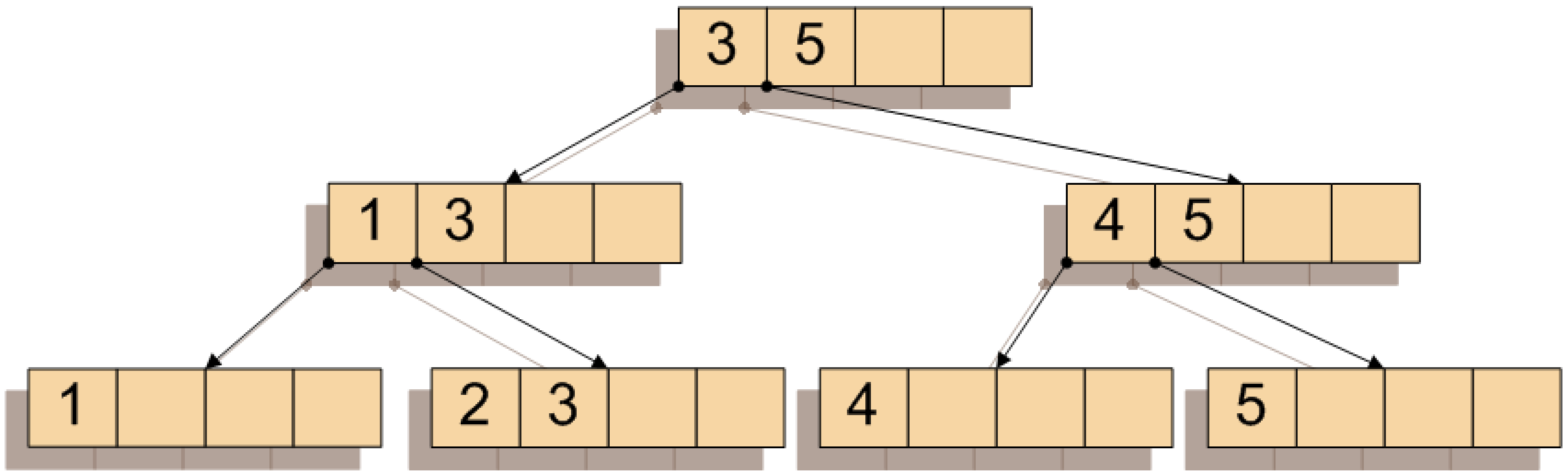
- Used:
  - MTL
  - My PC (4 core, 4GB RAM)

- Tested against:
  - Sequential version
    - All locks and thread-spawning were removed.
  - No-merge version
    - Merge operation was removed.
    - Tree is not rebalanced to correct for nodes that have too few keys after deletion.
    - Similar to Lehman & Yao's original B-link algorithm, which this algorithm was based on.

B-tree with merge:

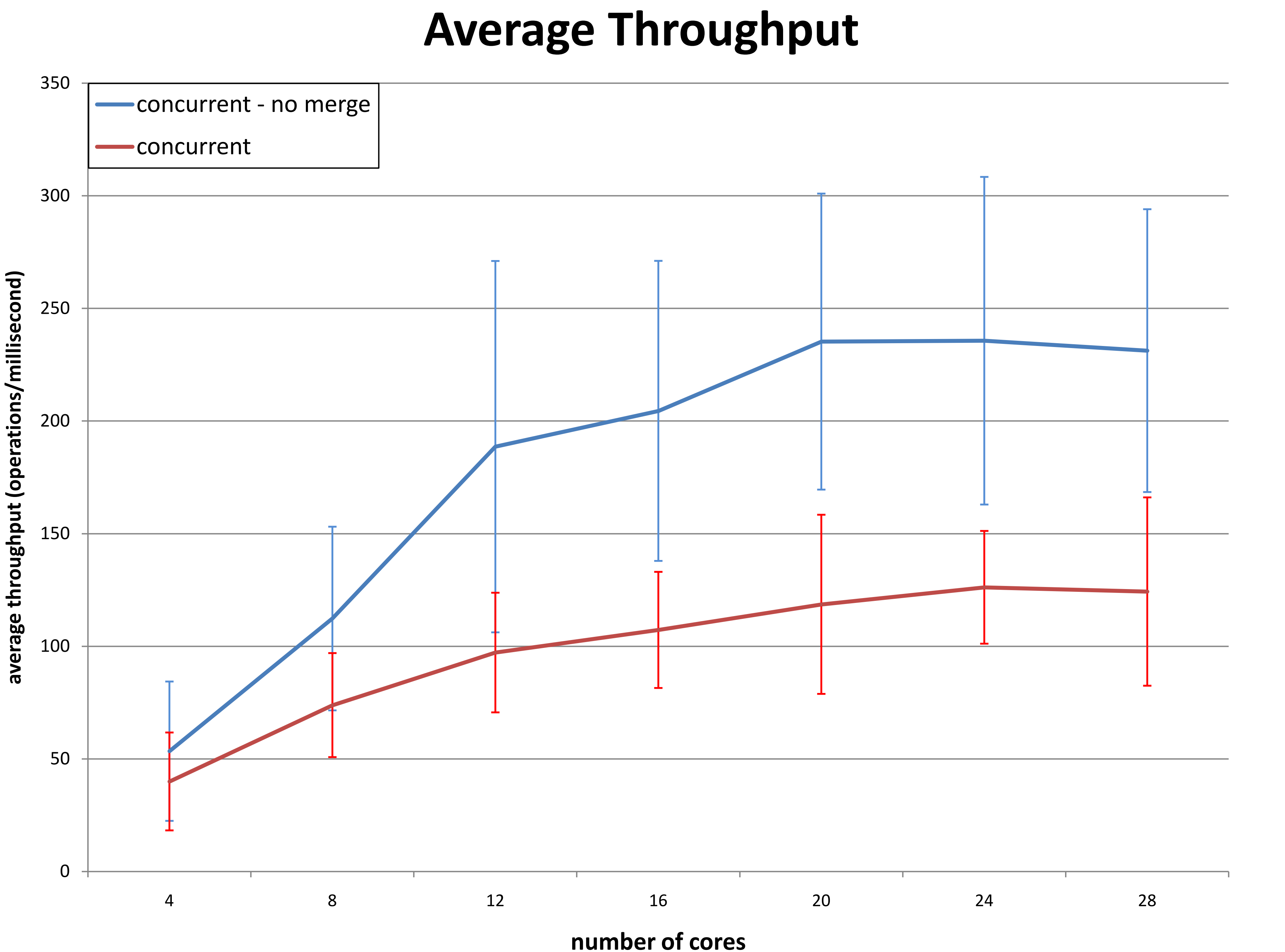


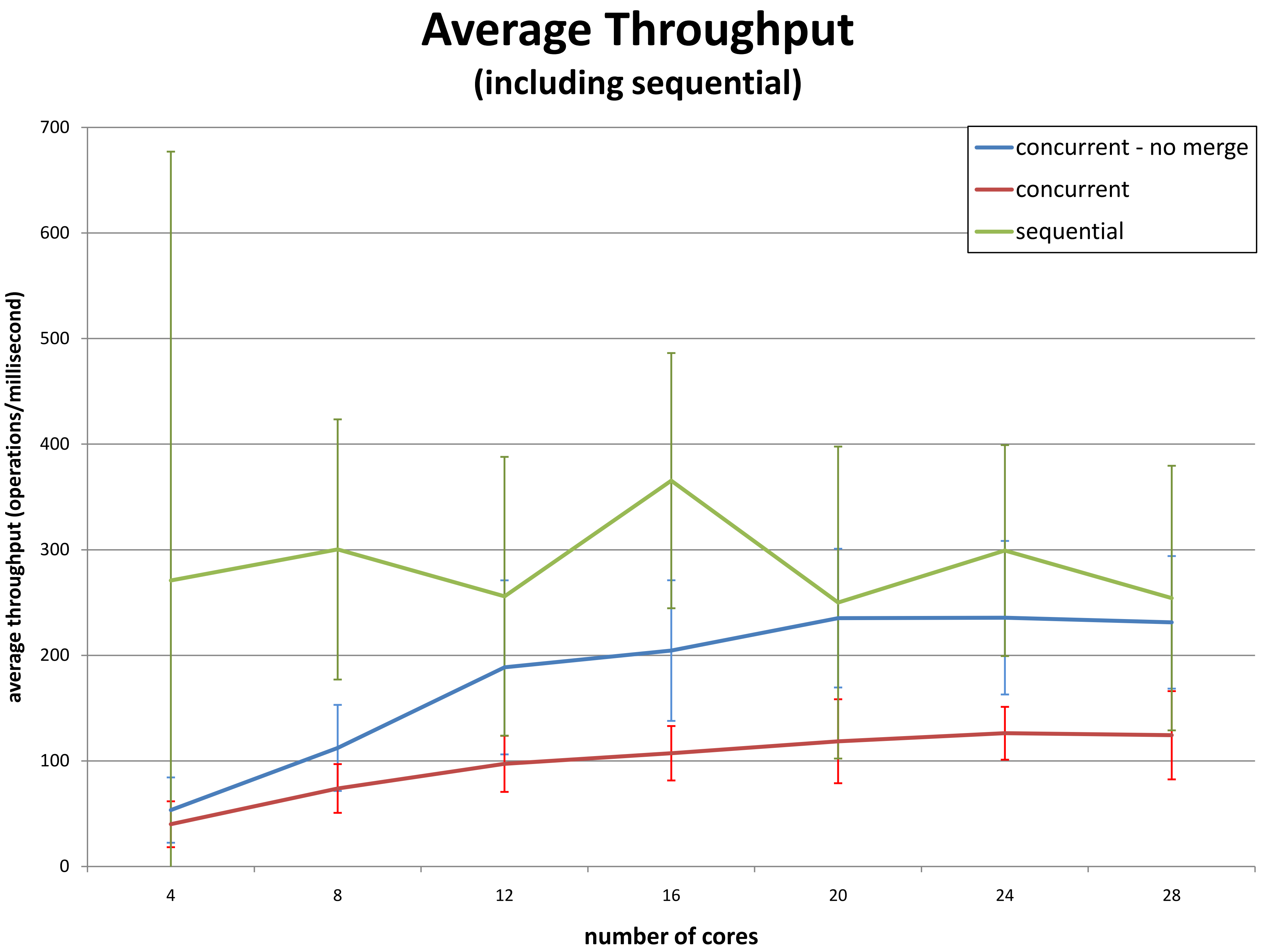
B-tree without merge:



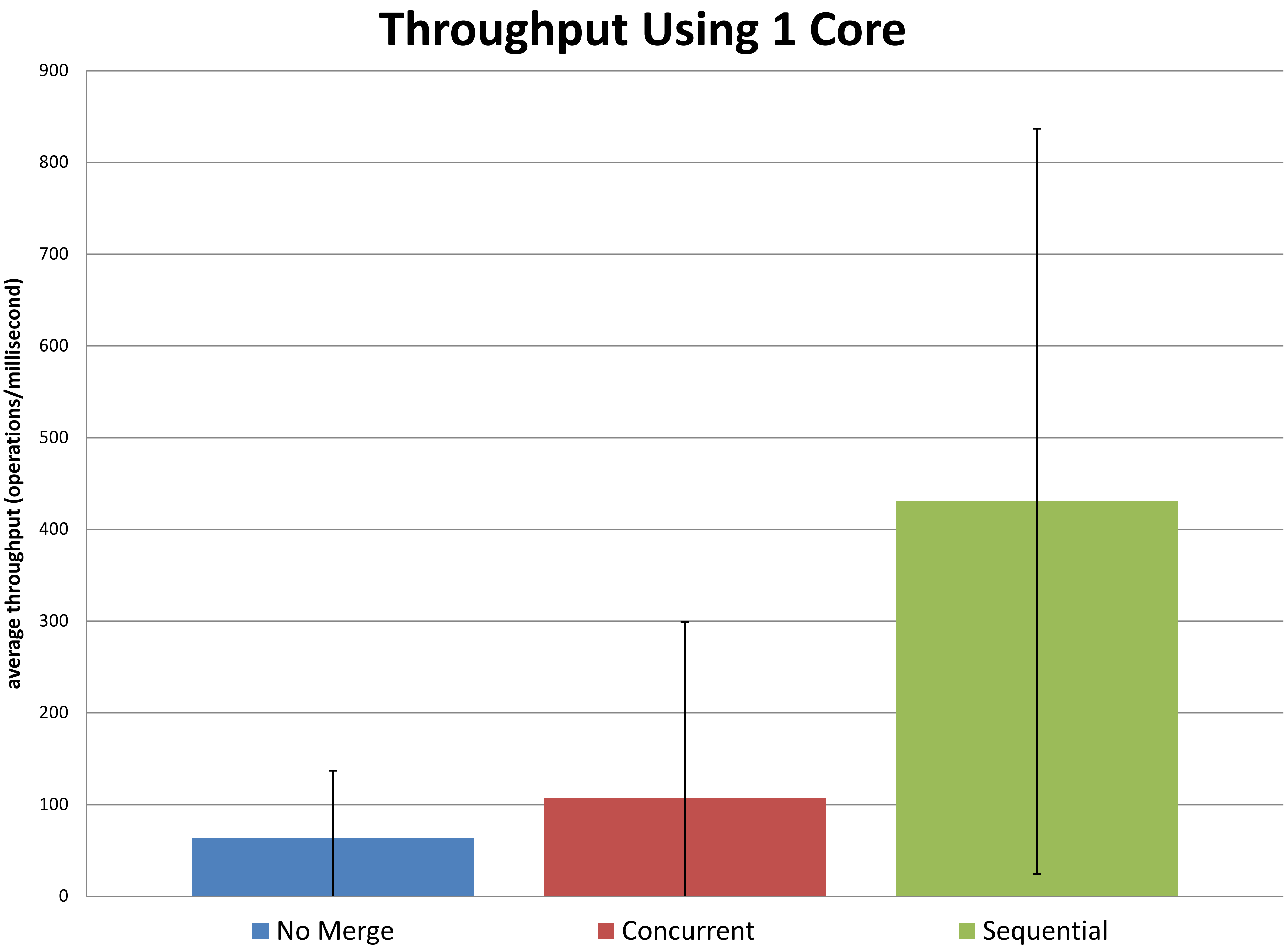
Variables

- Size of nodes = 8
- Number of threads
- Number of cores
- Size of tree
- Maximum key in tree
  - Affects maximum size (no duplicate keys)
- Proportion of threads for each of:
  - Search
  - Insert
  - Delete

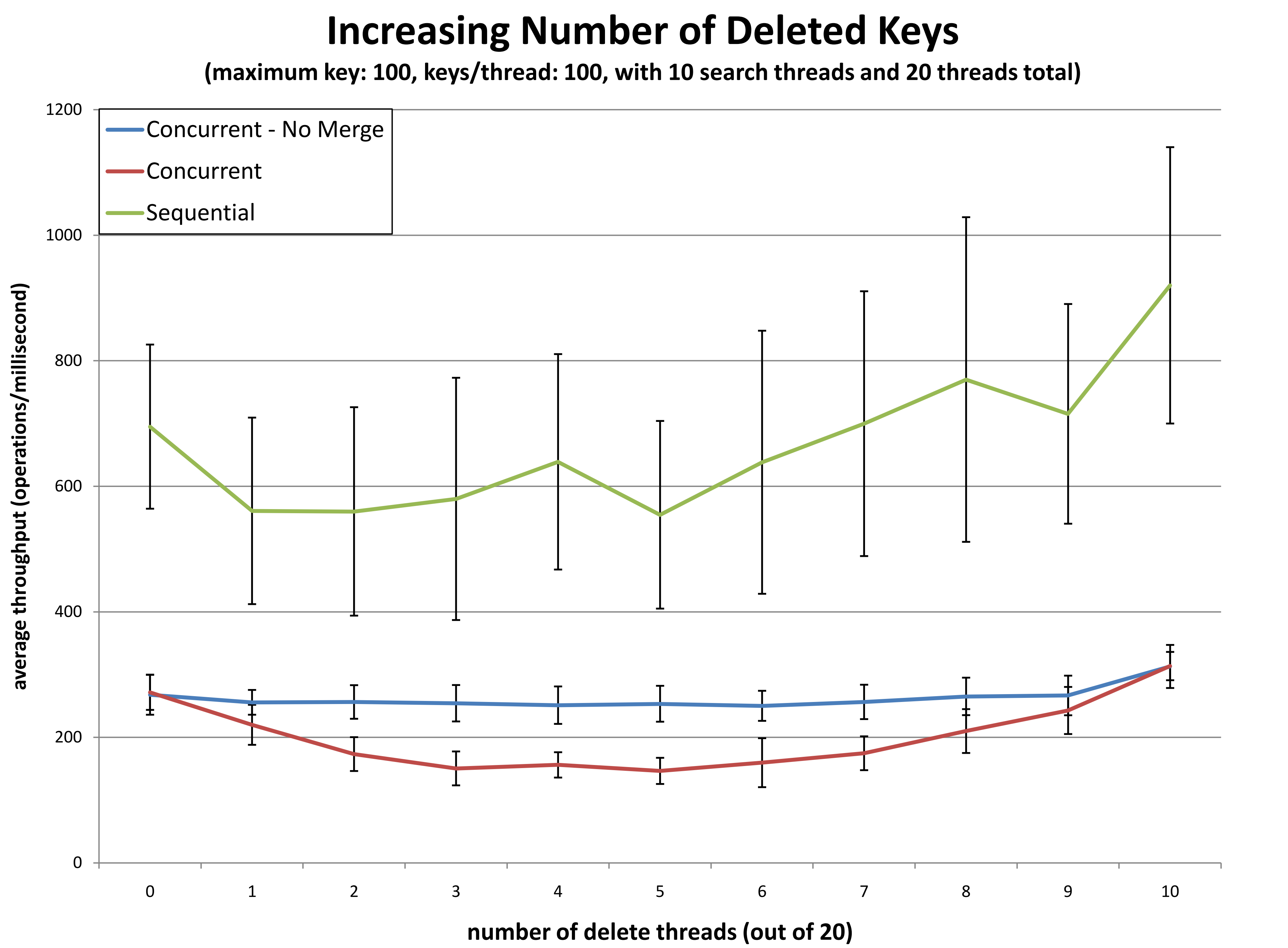

Average Throughput
concurrent - no merge
concurrent
average throughput (operations/millisecond)
0
50
100
150
200
250
300
350
4
8
12
16
20
24
28
number of cores



Average Throughput
(including sequential)
concurrent - no merge
concurrent
sequential
average throughput (operations/millisecond)
number of cores
0
100
200
300
400
500
600
700
4
8
12
16
20
24
28



Throughput Using 1 Core
average throughput (operations/millisecond)
900
800
700
600
500
400
300
200
100
0
No Merge
Concurrent
Sequential



Increasing Number of Deleted Keys
(maximum key: 100, keys/thread: 100, with 10 search threads and 20 threads total)
Concurrent - No Merge
Concurrent
Sequential
average throughput (operations/millisecond)
0
200
400
600
800
1000
1200
0
1
2
3
4
5
6
7
8
9
10
number of delete threads (out of 20)

Questions?