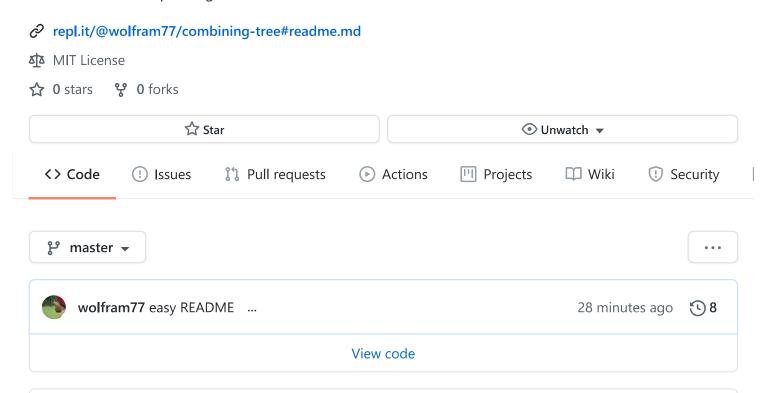
A Combining Tree is an N-ary tree of nodes, that follows software combining to reduce memory contention while updating a shared value.



# README.md

A Combining Tree is an N-ary tree of nodes, that follows software combining to reduce memory contention while updating a shared value.

The shared value is placed at the root of the tree, and threads perfrom <code>getAndOp()</code> at the leaf nodes. Each leaf node handles N threads. The combined value is then propagated up the tree by an active thread. There can be several active threads, but eventually one active thread updates the root node. It then shares this message to all other threads behind it.

This was contention at a single memory location is avoided. However, i guess that such a design is normally useful only in hardware, as it becomes too slow in software. Useful for educational purposes.

CombiningTree.getAndOp(x, op):
Gets current value, and then updates it.
x: value to OP with, op: binary op
1. Select leaf index using thread id.
2. Perform get & op at desired leaf.

0

```
CombiningTree.getAndOp(x, op, i):
Gets current value, and then updates it.
x: value to OP with, op: binary op, i: leaf index
1. Perform get & op at desired leaf (ensure in limit).
Node.getAndOp(x, op)
Gets current value, and then updates it.
x: value to OP (accumulate), op: binary operator
1. Wait until node is free.
2. Perform get & op based on 3 possible cases.
2a. Root node
2b. Active thread (first to visit node)
2c. Passive thread (visits later)
Node.getAndOpRoot(x, op)
Performs get & op for root node.
x: value to OP (accumulate), op: binary operator
1. Get old value, by combining (a).
2. Empty the node.
3. Insert a OP x
3. Return old value.
Node.getAndOpActive(x, op)
Performs get & op for active thread.
x: value to OP (accumulate), op: binary operator
1. Insert value.
2. Wait until node is full, or timeout.
3. We have the values, so start pushing.
4. Combine values into one with OP.
5. Push combined value to parent.
6. Distribute recieved value for all threads.
7. Start the pulling process.
8. Decrement count (we have our pulled value).
9. Return pulled value.
Node.getAndOpPassive(x, op)
Performs get & op for passive thread.
x: value to OP (accumulate), op: binary operator
1. Insert value.
2. Wait until active thread has pulled value.
3. Decrement count, one pulled value processed.
4. If count is 0, the node is free.
```

5. Return value of this thread.

```
## OUTPUT
3-ary 3-depth Combining tree.
Starting 25 threads doing increments ...
30: done in 51ms
14: done in 60ms
13: done in 68ms
23: done in 53ms
12: done in 56ms
21: done in 54ms
32: done in 56ms
31: done in 56ms
22: done in 54ms
28: done in 7831ms
11: done in 7836ms
10: done in 7838ms
19: done in 7833ms
20: done in 7833ms
29: done in 7831ms
24: done in 8165ms
33: done in 8160ms
25: done in 8165ms
16: done in 8167ms
34: done in 8159ms
15: done in 8167ms
26: done in 11970ms
17: done in 11972ms
27: done in 12239ms
18: done in 12241ms
Total: 2500
Was valid? true
```

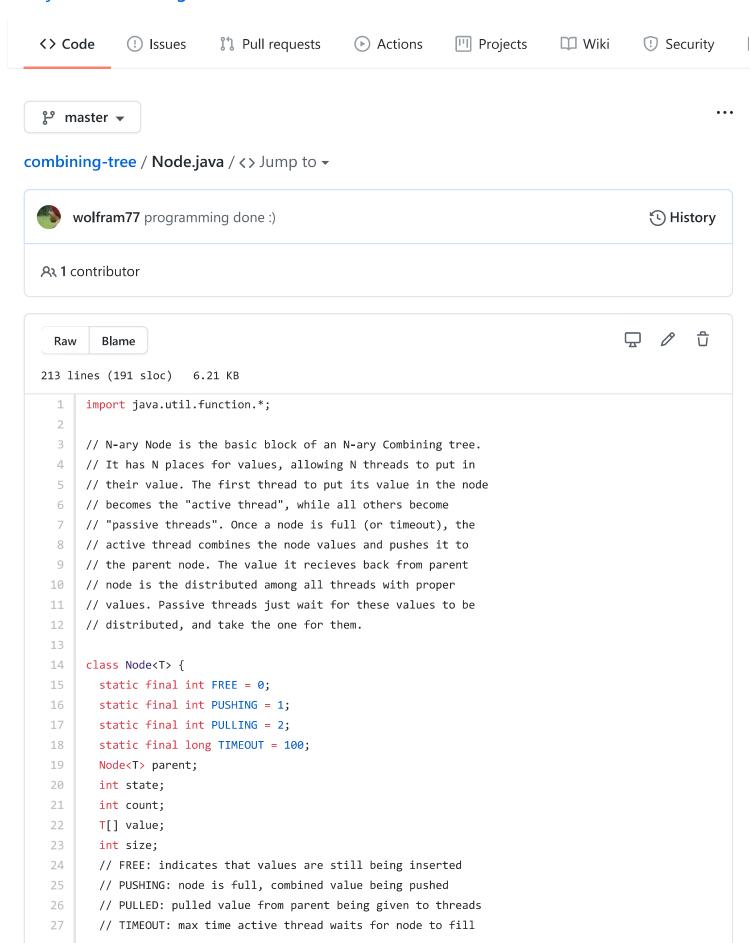
See CombiningTree.java, Node.java for code, Main.java for test, and repl.it for output.

## references

• The Art of Multiprocessor Programming :: Maurice Herlihy, Nir Shavit

#### Languages

Java 100.0%

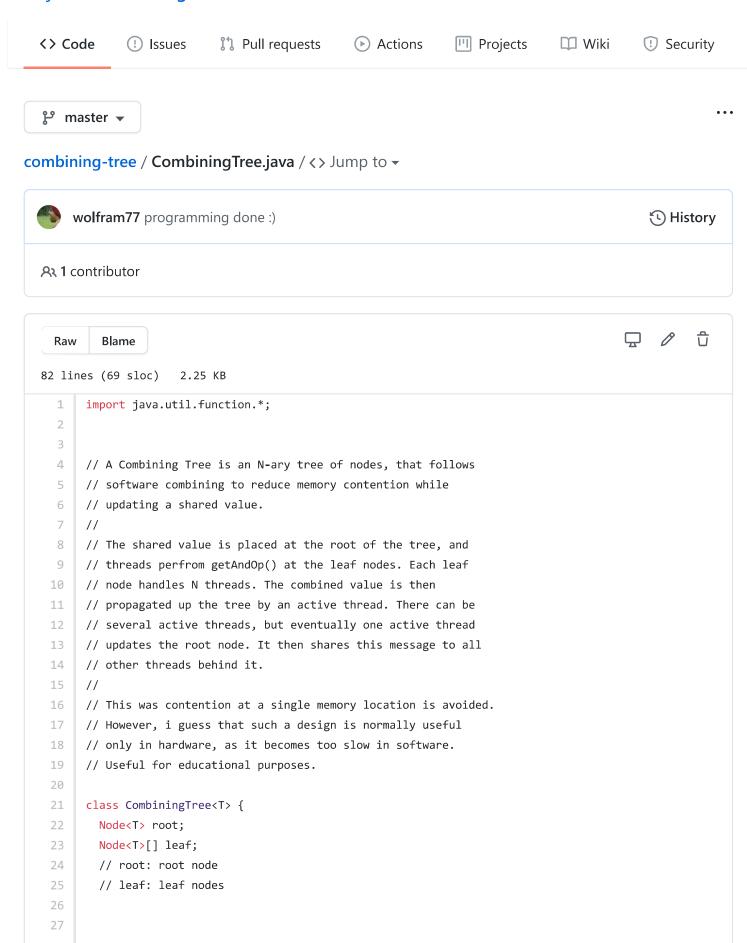


```
// parent: parent node
       // state: either FREE, PUSHING, or PULLING
29
       // count: number of values in node
30
       // value: storage place for values
32
       // size: max. no. of values allowed (arity of node, eg 2)
33
34
       public Node() {
         this(2);
37
       }
38
39
       @SuppressWarnings("unchecked")
40
       public Node(int n) {
         value = (T[]) new Object[n];
41
42
         parent = null;
43
         state = FREE;
44
         count = 0;
45
         size = n;
46
       }
47
48
49
       // Gets node value (only for root node).
50
       public synchronized T get() {
51
         return value[0];
52
       }
53
54
       // Sets node value (only for root node).
       public synchronized void set(T x) {
56
         value[0] = x;
57
         count = 1;
58
       }
60
61
       // Gets current value, and then updates it.
       // x: value to OP (accumulate), op: binary operator
62
       // 1. Wait until node is free.
63
       // 2. Perform get & op based on 3 possible cases.
64
65
       // 2a. Root node
       // 2b. Active thread (first to visit node)
       // 2c. Passive thread (visits later)
67
68
       public synchronized T getAndOp(T x,
       BinaryOperator<T> op)
       throws InterruptedException {
70
         while (state!=FREE || count==size) wait(); // 1
71
72
         if (parent==null) return getAndOpRoot(x, op); // 2a
         if (count==0) return getAndOpActive(x, op); // 2b
73
74
         return getAndOpPassive(x, op); // 2c
75
       }
```

```
76
 77
        // Performs get & op for root node.
        // x: value to OP (accumulate), op: binary operator
 78
 79
        // 1. Get old value, by combining (a).
 80
        // 2. Empty the node.
        // 3. Insert a OP x
 81
        // 3. Return old value.
 83
        private synchronized T getAndOpRoot(T x,
 84
        BinaryOperator<T> op)
 85
        throws InterruptedException {
 86
          T a = combine(op);
 87
          count = 0;
 88
          insert(op.apply(a, x));
 89
          return a;
        }
 91
 92
        // Performs get & op for active thread.
 93
        // x: value to OP (accumulate), op: binary operator
        // 1. Insert value.
        // 2. Wait until node is full, or timeout.
 96
        // 3. We have the values, so start pushing.
 97
        // 4. Combine values into one with OP.
 98
        // 5. Push combined value to parent.
        // 6. Distribute recieved value for all threads.
100
        // 7. Start the pulling process.
101
        // 8. Decrement count (we have our pulled value).
102
        // 9. Return pulled value.
103
        private synchronized T getAndOpActive(T x,
104
        BinaryOperator<T> op)
105
        throws InterruptedException {
106
          insert(x); // 1
107
          waitUntilFull(TIMEOUT); // 2
108
          state = PUSHING; // 3
109
          T = combine(op); // 4
110
          T r = parent.getAndOp(a, op); // 5
111
          distribute(r, op); // 6
          state = PULLING; // 7
112
113
          notifyAll();
                             // 7
114
          decrementCount(); // 8
115
          return r; // 9
116
        }
117
        // Performs get & op for passive thread.
118
119
        // x: value to OP (accumulate), op: binary operator
120
        // 1. Insert value.
        // 2. Wait until active thread has pulled value.
121
122
        // 3. Decrement count, one pulled value processed.
123
        // 4. If count is 0, the node is free.
```

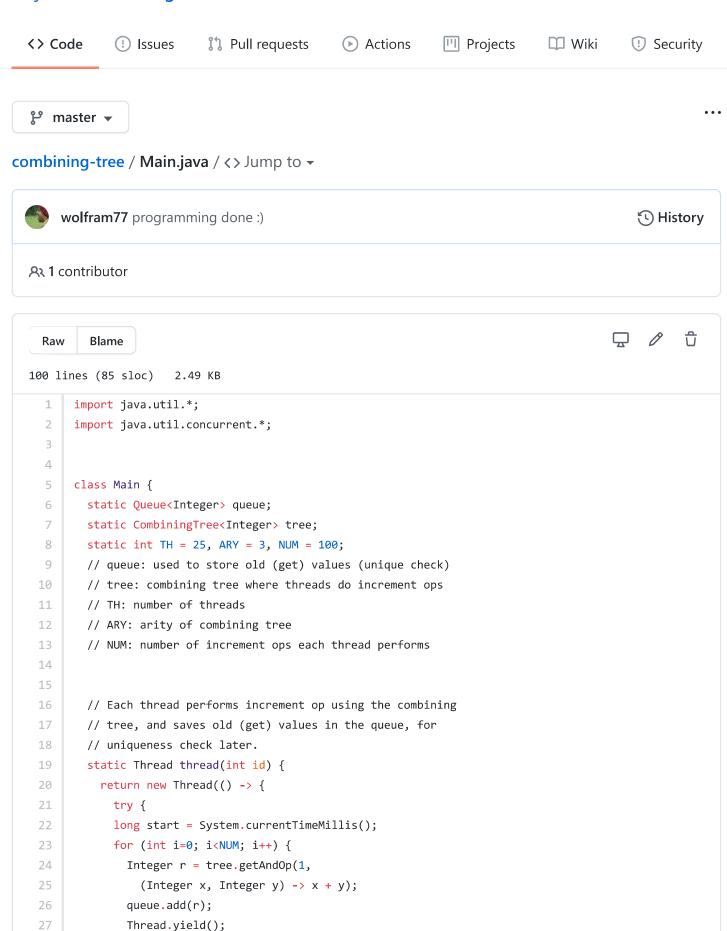
```
124
        // 5. Return value of this thread.
125
        private synchronized T getAndOpPassive(T x,
126
        BinaryOperator<T> op)
127
        throws InterruptedException {
128
          int i = insert(x); // 1
129
          while (state!=PULLING) wait(); // 2
130
          decrementCount(); // 3, 4
131
          return value[i]; // 5
132
        }
133
134
135
        // Inserts a value in the node (for a thread).
136
        // x: value to insert
137
        // 1. Wait unit node is free.
        // 2. Get index to place value in.
138
139
        // 3. Place the value.
140
        // 4. Increment number of values in node.
141
        // 5. If node is full, notify active thread.
142
        // 6. Return index where value was placed.
143
        public synchronized int insert(T x)
144
        throws InterruptedException {
145
          while (state!=FREE) wait(); // 1
146
          int i = count;
                           // 2
147
          value[i] = x;
                             // 3
148
          incrementCount(); // 4, 5
149
          return i; // 6
150
        }
151
152
153
        // Combine all values in the node (put by threads).
154
        // op: binary operator
155
        // 1. If OP is "+", combine will return sum.
        public synchronized T combine(BinaryOperator<T> op) {
156
157
          T a = value[0]; // 1
          for (int i=1; i<count; i++) // 1</pre>
158
159
            a = op.apply(a, value[i]); // 1
          return a; // 1
160
161
        }
162
163
164
        // Distribute pulled value to all threads.
        // r: pulled value, op: binary operator
165
        // T0: active thread, T1...: passive threads
166
        // 1. T0 receives r
167
168
        // 2. T1 recieves r OP v0.
        // 3. T2 recieves r OP v0 OP v1 ...
169
        public synchronized void distribute(T r,
170
171
        BinaryOperator<T> op) {
```

```
172
          for (int i=0; i<count; i++) { // 1</pre>
173
            T x = value[i];
174
                                         // 1
            value[i] = r;
175
            r = op.apply(r, x); // 2
176
          } // 3
177
        }
178
179
        // Increment count once done with insertion.
180
        // 1. Increment count.
181
        // 2. If node is full, notify active thread.
        private synchronized void incrementCount() {
182
183
          if (++count<size) return; // 1</pre>
184
          notifyAll(); // 2
185
        }
186
187
        // Decrement count once done with distribution.
188
        // 1. Decrement count.
189
        // 2. If count is zero, node is free.
190
        private synchronized void decrementCount() {
191
          if (--count>0) return; // 1
192
          state = FREE; // 2
193
          notifyAll(); // 2
194
        }
195
196
        // Wait until node is full, or timeout.
197
        // 1. Get start time.
198
        // 2. If node is full, exit.
199
        // 3. Otherwise, wait with timeout.
200
        // 4. On waking up, check current time.
201
        // 5. Reduce timeout by the elapsed time.
202
        // 6. If timeout done, exit (else retry 2).
203
        private void waitUntilFull(long w)
        throws InterruptedException {
204
205
          long t0 = System.currentTimeMillis(); //1
206
          while (count < size) { // 2</pre>
207
            wait(w);
                                 // 3
            long t = System.currentTimeMillis(); // 4
208
209
            w -= t - t0;
                            // 5
210
            if (w<=0) break; // 6
211
          }
212
        }
213
      }
```



```
28
       public CombiningTree(int depth) {
29
         this(depth, 2);
30
       }
31
32
       public CombiningTree(int depth, int ary) {
33
         Node<T>[] parent = createNodes(1, ary);
34
         root = parent[0];
         leaf = parent;
         for (int i=1; i<depth; i++) {</pre>
37
           int n = (int) Math.pow(ary, i);
38
           leaf = createNodes(n, ary);
39
           for (int j=0; j<n; j++)</pre>
             leaf[j].parent = parent[j/ary];
41
           parent = leaf;
42
         }
43
       }
44
45
       @SuppressWarnings("unchecked")
46
       private Node<T>[] createNodes(int n, int ary) {
47
         Node<T>[] a = (Node<T>[]) new Node[n];
         for (int i=0; i<n; i++)
48
49
           a[i] = new Node<>(ary);
50
         return a;
51
       }
52
53
54
       // Get value of tree (at root).
       public T get() {
56
         return root.get();
57
       }
58
59
       // Set value of tree (at root).
60
       public void set(T x) {
61
         root.set(x);
       }
62
63
64
65
       // Gets current value, and then updates it.
       // x: value to OP with, op: binary op
       // 1. Select leaf index using thread id.
67
68
       // 2. Perform get & op at desired leaf.
       public T getAndOp(T x, BinaryOperator<T> op)
       throws InterruptedException {
70
         int i = (int) Thread.currentThread().getId(); // 1
71
72
         return getAndOp(x, op, i); // 2
73
       }
74
75
       // Gets current value, and then updates it.
```

```
// x: value to OP with, op: binary op, i: leaf index
// 1. Perform get & op at desired leaf (ensure in limit).
public T getAndOp(T x, BinaryOperator<T> op, int i)
throws InterruptedException {
    return leaf[i % leaf.length].getAndOp(x, op); // 1
}
```



```
28
29
           long stop = System.currentTimeMillis();
           log(id()+": done in "+(stop-start)+"ms");
30
31
           } catch (InterruptedException e) {}
32
         });
33
       }
34
       // Check if total sum is as expected.
       // Check if all old values are unique.
37
       static boolean wasValid() {
38
         int a = tree.get().intValue();
39
         if (a != TH*NUM) return false;
         Set<Integer> s = new HashSet<>();
41
         while (queue.size()>0) {
           Integer n = queue.remove();
42
43
           if (s.contains(n)) return false;
44
           s.add(n);
45
         }
46
         return true;
47
       }
48
49
       // Setup the combining tree for threads.
50
       static void setupTreeAndQueue() {
51
         int depth = (int) Math.ceil(Math.log(TH)/Math.log(ARY));
52
         tree = new CombiningTree<>(depth, ARY);
53
         tree.set(0);
54
       }
56
       // Setup the queue for storing old values.
57
       static void setupQueue() {
58
         queue = new ConcurrentLinkedQueue<>();
59
       }
60
61
62
       // Start threads doing increments using tree.
63
       static Thread[] startOps() {
64
         Thread[] t = new Thread[TH];
65
         for (int i=0; i<TH; i++)</pre>
           t[i] = thread(i);
         for (int i=0; i<TH; i++)
67
68
           t[i].start();
         return t;
70
       }
71
72
       // Wait until all threads done with increments.
       static void awaitOps(Thread[] t) {
73
74
         try {
75
         for (int i=0; i<TH; i++)</pre>
```

```
76
            t[i].join();
 77
          } catch (InterruptedException e) {}
 78
        }
79
80
81
        public static void main(String[] args) {
 82
          setupTree();
          setupQueue();
83
84
          log(ARY+"-ary "+depth+"-depth Combining tree.");
85
          log("Starting "+TH+" threads doing increments ...");
          Thread[] t = startOps();
86
          awaitOps(t);
87
          log("Total: "+tree.get());
88
          log("\nWas valid? "+wasValid());
 89
90
        }
 91
92
93
        static void log(String x) {
94
          System.out.println(x);
 95
        }
96
97
        static long id() {
          return Thread.currentThread().getId();
98
99
        }
100
      }
```