Zad. 2

Regularny M-wartościowy rejestr implementujemy jako tablicę M rejestrów boolowskich. Początkowo rejestr jest ustawiony na wartość zero, o czym świadczy r_bit[0] == true.

```
1  public class RegMRSWRegister implements Register<Byte> {
2    private static int RANGE = Byte.MAX_VALUE - Byte.MIN_VALUE + 1;
3    boolean[] r_bit = new boolean[RANGE]; // regular boolean MRSW
4    public RegMRSWRegister(int capacity) {
5      for (int i = 1; i < r_bit.length; i++)
6        r_bit[i] = false;
7      r_bit[0] = true;
8    }
9    public void write(Byte x) {
10     r_bit[x] = true;
11     for (int i = x - 1; i >= 0; i--)
12       r_bit[i] = false;
13   }
14   public Byte read() {
15     for (int i = 0; i < RANGE; i++)
16       if (r_bit[i]) {
17         return i;
18       }
19     return -1; // impossible
20   }
21 }
```

Figure 4.8 The RegMRSWRegister class: a regular *M*-valued MRSW register.

Aby wykazać poprawność, musimy pokazać, że jest regularny, czyli poniższe zdania muszą być zawsze prawdziwe:

---

dla każdego i nie jest prawdą, że $R^i \rightarrow W^i$ (*),

dla każdych i oraz j nie jest prawdą, że $W^i \rightarrow W^j \rightarrow R^i$ (**),

(*)

Lemat 1. Wywołanie metody read() zawsze zwróci wartość dla bitu od 0 do M-1, ustawionego przez jakieś write()

Następująca własność jest niezmienna: jeżeli wątek odczytujący przeczyta r_bit[j], to jakiś bit z indeksem j lub wyższym, zapisany w wyniku wywołania metody write(), jest ustawiony na true. Inicjalizując rejestr nie ma wątków odczytujących i konstruktor ustawia r_bit[0] = true.
Załóżmy, że czytamy j-ty bit, a jakiś k-ty jest ustawiony na true dla k>=j:

1) Jeżeli wątek czytający przejdzie na j + 1 to znaczy, że j jest ustawione na false, czyli k>j
2) Wątek zapisujący może ustawić bit na false tylko gdy ustawi wyższy na true

Zatem lemat jest prawdziwy.

(**)
Lemat 2: Konstrukcja z kodu jest regularnym rejestrem M-wartościowym typu MRSW.

Dla dowolnego odczytu ‘x’ niech będzie wartością zapisaną przez ostatnie write(), które nie nakłada się z innymi operacjami. W momencie zakończenia zapisu x-ty bit zostanie ustawiony na true, a wszystkie mniejsze na false. Z lematu 1. wiemy, że jeśli read() zwraca wartość inną niż x to znaczy, że zwraca wartość ustawioną przez obecny zapis.

Zatem rejestr jest regularny.

Zad. 3

```
1  public class AtomicMRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
3    private StampedValue<T>[][] a_table; // each entry is an atomic SRSW register
4    public AtomicMRSWRegister(T init, int readers) {
5      lastStamp = new ThreadLocal<Long>() {
6         protected Long initialValue() { return 0; };
7      };
8      a_table = (StampedValue<T>[][]) new StampedValue[readers][readers];
9      StampedValue<T> value = new StampedValue<T>(init);
10     for (int i = 0; i < readers; i++) {
11       for (int j = 0; j < readers; j++) {
12         a_table[i][j] = value;
13       }
14     }
15   }
16   public T read() {
17     int me = ThreadID.get();
18     StampedValue<T> value = a_table[me][me];
19     for (int i = 0; i < a_table.length; i++) {
20       value = StampedValue.max(value, a_table[i][me]);
21     }
22     for (int i = 0; i < a_table.length; i++) {
23       if (i == me) continue;
24       a_table[me][i] = value;
25     }
26     return value;
27   }
28   public void write(T v) {
29     long stamp = lastStamp.get() + 1;
30     lastStamp.set(stamp);
31     StampedValue<T> value = new StampedValue<T>(stamp, v);
32     for (int i = 0; i < a_table.length; i++) {
33       a_table[i][i] = value;
34     }
35   }
36 }
```

FIGURE 4.12

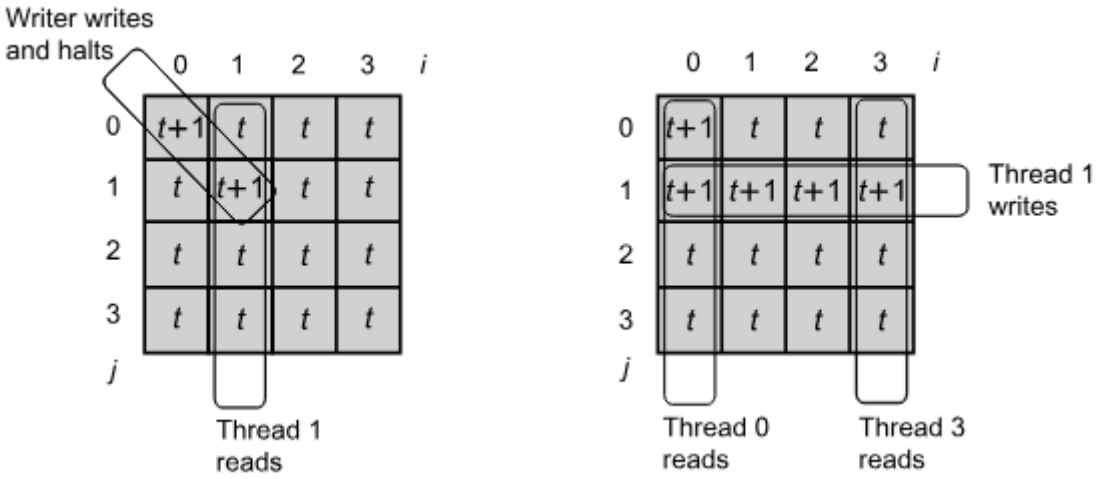
The `AtomicMRSWRegister` class: an atomic MRSW register constructed from atomic SRSW registers.

Chcemy pokazać, że podany rejestr jest rejestrem atomowym MRSW, czyli chcemy pokazać, że spełnia podane warunki:

dla każdego i nie jest prawdą, że $R^i \rightarrow W^i$ (*),

dla każdych i oraz j nie jest prawdą, że $W^i \rightarrow W^j \rightarrow R^i$ (**), oraz

dla dla każdych i oraz j jeśli $R^i \rightarrow R^j$ to $i \leq j$ (***).



(*) Żaden read() nie może zwrócić wartości z przyszłości. ok

(**) Z kodu widać, że write() zapisuje zawsze rosnące timestamps. Można zauważyć, że maksymalny timestamp wzdłuż wiersza czy kolumny także rośnie. Zatem jeśli write() zapisze coś to kolejne wywołanie read() odczyta z przekątnej maksymalny timestamp większy lub równy temu z write(). ok

(***) Jeżeli wywołanie read() przez jeden wątek całkowicie poprzedza wywołanie go przez drugi to pierwszy zapisze swoją wartości z timestamp w wierszu wątku drugiego. Stąd drugi wybierze wartość z większym lub równym timestamp. ok

Zatem jest atomowy.

Zad 4.

```
1  public class AtomicMRMWRegister<T> implements Register<T>{
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
3    public AtomicMRMWRegister(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      StampedValue<T> value = new StampedValue<T>(init);
6      for (int j = 0; j < a_table.length; j++) {
7        a_table[j] = value;
8      }
9    }
10   public void write(T value) {
11     int me = ThreadID.get();
12     StampedValue<T> max = StampedValue.MIN_VALUE;
13     for (int i = 0; i < a_table.length; i++) {
14       max = StampedValue.max(max, a_table[i]);
15     }
16     a_table[me] = new StampedValue(max.stamp + 1, value);
17   }
18   public T read() {
19     StampedValue<T> max = StampedValue.MIN_VALUE;
20     for (int i = 0; i < a_table.length; i++) {
21       max = StampedValue.max(max, a_table[i]);
22     }
23     return max.value;
24   }
25 }
```

FIGURE 4.14

Atomic MRMW register.

(*) Każdy write() następujący po read() będzie miał timestamp wyższy niż jakikolwiek inny zapisany w momencie jego działania, zatem read() nie może odczytać write() wywołanego po nim. ok

(**) Załóżmy, że wywołanie write() przez A poprzedziło wywołanie write() przez B, które z kolei poprzedziło read() przez C. Jeśli A = B, to późniejszy write() nadpisuje a_table[A], a read() nie zwraca wartości wcześniejszego write(). Jeśli A =/= B, to ponieważ timestamp A jest mniejszy niż timestamp B, każdy C, który widzi oba, zwraca wartość B (lub wartość z wyższym timestamp), zatem nie odczytuje A. ok

(***) Załóżmy, że wywołanie read() przez A jest przed wywołaniem read() przez B oraz wywołanie write() przez C jest przed wywołaniem write() przez D. Możemy pokazać, że jeśli A zwróci D to B nie może zwrócić C.
Jeżeli timestamp C jest mniejszy od timestamp D to A zwróci D a zatem B nie zwróci C.
Jeżeli timestamp C jest równy timestamp D to wątki musiały wykonać zapis współbieżnie, a z kolejności zapisu C<D, więc jeżeli A odczyta timestamp D to B również go odczyta. ok

Zatem jest atomowy.

Zad 5.

```
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class MergeSortConcurrent {

  static class Task {
    public String type;
    public int l, r, m, parent;

    Task(String type, int l, int m, int r, int parent) {
      this.type = type;
      this.l = l;
      this.m = m;
      this.r = r;
      this.parent = parent;
    }
  }

  public static void main(String[] args) {
    int[] array = {10, 8, 7, 6, 5, 4, 3, 2, 1, 3, 5, 6, 34, 78};

    AtomicInteger sorted = new AtomicInteger(0);
    AtomicInteger[] children_sorted = new AtomicInteger[array.length];

    for (int i=0; i < array.length; i++) {
      children_sorted[i] = new AtomicInteger(0);
    }

    AtomicInteger merge_index = new AtomicInteger(0);

    Task[] awaiting_merges = new Task[array.length];

    ConcurrentLinkedQueue<Task> queue = new ConcurrentLinkedQueue<>();
    queue.offer(new Task("SORT", 0, -1, array.length-1, -1));

    int THREAD_COUNT = 5;
    Thread[] workers = new Thread[THREAD_COUNT];

    for (int i=0; i < THREAD_COUNT; i++) {
      workers[i] = new Thread(() -> {
        while (sorted.get() == 0) {
          Task t = queue.poll();

          if (t != null) {
            if (t.type.equals("SORT")) {
              if (t.l == t.r) {
```

```
      if (children_sorted[t.parent].getAndIncrement() == 1) {
        queue.offer(awaiting_merges[t.parent]);
      }
    } else {
      int idx = merge_index.getAndIncrement();
      t.m = (t.l + t.r) / 2;

      awaiting_merges[idx] = new Task("MERGE", t.l, t.m, t.r, t.parent);

      queue.offer(new Task("SORT", t.l, -1, t.m, idx));
      queue.offer(new Task("SORT", t.m + 1, -1, t.r, idx));
    }

  } else {

    int[] temp = new int[array.length];
    int leftSize = t.m - t.l + 1;
    int rightSize = t.r - t.m;

    int k = t.l;
    int leftPos = t.l, rightPos = t.m + 1;

    System.arraycopy(array, leftPos, temp, leftPos, leftSize);
    System.arraycopy(array, rightPos, temp, rightPos, rightSize);

    while (leftPos <= t.m && rightPos <= t.r) {
      if (temp[leftPos] <= temp[rightPos]) {
        array[k++] = temp[leftPos++];
      } else {
        array[k++] = temp[rightPos++];
      }
    }

    while (leftPos <= t.m) {
      array[k++] = temp[leftPos++];
    }

    while (rightPos <= t.r) {
      array[k++] = temp[rightPos++];
    }

    if (t.parent == -1) {
      sorted.getAndIncrement();
    } else if (children_sorted[t.parent].getAndIncrement() == 1) {
        queue.offer(awaiting_merges[t.parent]);

    }
  }
```

```
        }
      }
    });

    workers[i].start();
}

for (Thread worker : workers) {
    try {
        worker.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

for (int i : array) {
    System.out.printf("%d ", i);
} }}
```