Zad. 1

1)

Niezmiennik reprezentacji - warunek, który musi zostać zachowany przed i po wywołaniu metody, dla CoarseList jest to:

1. Klucz dowolnego węzła musi być mniejszy od klucza swojego następnika (zatem lista jest posortowana i klucze są unikatowe)
2. Klucz dowolnego węzła dodanego, usuniętego lub wyszukanego musi być mniejszy od taila i większy od heada.

Mapa abstrakcji - zbiory otrzymywalne z list, dla CoarseList jest to:

Element znajduje się w zbiorze wtedy i tylko wtedy, kiedy jest osiągalny z węzła head.

2)

a) add()
pred.next = node (linijka 24)

b) remove()
pred.next = curr.next (linijka 43)

c) contains()
lock.lock()

3)

W metodach add() i remove() zajmujemy zamek zanim dokonamy wstawienia lub usunięcia węzła, zatem ustawiając w tym miejscu punkt linearyzacji twierdzenie "Element znajduje się w zbiorze wtedy i tylko wtedy, kiedy jest osiągalny z węzła head." nie jest poprawne.

4)

"Element x znajduje się w zbiorze wtedy i tylko wtedy, kiedy (*) jest osiągalny z węzła head i zamek nie jest zajęty przez metodę remove(x) lub (**) zamek jest zajęty przez metodę add(x)"

```
1  public class CoarseList<T> {
2    private Node head;
3    private Lock lock = new ReentrantLock();
4    public CoarseList() {
5      head = new Node(Integer.MIN_VALUE);
6      head.next = new Node(Integer.MAX_VALUE);
7    }
8    public boolean add(T item) {
9      Node pred, curr;
10     int key = item.hashCode();
11     lock.lock();
12     try {
13       pred = head;
14       curr = pred.next;
15       while (curr.key < key) {
16         pred = curr;
17         curr = curr.next;
18       }
19       if (key == curr.key) {
20         return false;
21       } else {
22         Node node = new Node(item);
23         node.next = curr;
24         pred.next = node;
25         return true;
26       }
27     } finally {
28       lock.unlock();
29     }
30   }
```

Figure 9.4 The CoarseList class: the add() method.

```
31   public boolean remove(T item) {
32     Node pred, curr;
33     int key = item.hashCode();
34     lock.lock();
35     try {
36       pred = head;
37       curr = pred.next;
38       while (curr.key < key) {
39         pred = curr;
40         curr = curr.next;
41       }
42       if (key == curr.key) {
43         pred.next = curr.next;
44         return true;
45       } else {
46         return false;
47       }
48     } finally {
49       lock.unlock();
50     }
51   }
```

Figure 9.5 The CoarseList class: the remove() method. All methods acquire a single lock, which is released on exit by the **finally** block.

Zad. 3

```java
public boolean contains(T item) {
  int key = item.hashCode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      // tutaj zamiana względem add / remove
      return curr.key == key;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Ponieważ przechodzimy listę, zawsze trzymając zamki dla sąsiednich węzłów mamy gwarancję, że szukany węzeł nie zostanie wstawiony pomiędzy aktualnie rozpatrywaną parą. Nasza lista jest posortowana, musimy więc jedynie znaleźć pierwszy węzeł o kluczu niemniejszym od szukanego klucza i sprawdzić czy ten klucz i klucz szukany są tym samym.

Zad. 4

Optymistyczna implementacja zakłada, że problemy synchronizacji pojawiają się dość rzadko. Z tego powodu wątki chcące wykonać operację na elementach listy nie zakładają zamków podczas przejścia przez listę. Zamki są zakładane dopiero na modyfikowane węzły listy.

Zaletą takiego podejścia jest zmniejszenie opóźnienia jeżeli problemy z synchronizacją faktycznie nie wystąpią często.

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4       Node pred = head;
5       Node curr = pred.next;
6       while (curr.key <= key) {
7         pred = curr; curr = curr.next;
8       }
9       pred.lock(); curr.lock();
10      try {
11        if (validate(pred, curr)) {
12          if (curr.key == key) {
13            return false;
14          } else {
15            Node node = new Node(item);
16            node.next = curr;
17            pred.next = node;
18            return true;
19          }
20        }
21      } finally {
22        pred.unlock(); curr.unlock();
23      }
24    }
25  }
```

```
26  public boolean remove(T item) {
27    int key = item.hashCode();
28    while (true) {
29      Node pred = head;
30      Node curr = pred.next;
31      while (curr.key < key) {
32        pred = curr; curr = curr.next;
33      }
34      pred.lock(); curr.lock();
35      try {
36        if (validate(pred, curr)) {
37          if (curr.key == key) {
38            pred.next = curr.next;
39            return true;
40          } else {
41            return false;
42          }
43        }
44      } finally {
45        pred.unlock(); curr.unlock();
46      }
47    }
48  }
```

```
67    private boolean validate(Node pred, Node curr) {
68      Node node = head;
69      while (node.key <= pred.key) {
70        if (node == pred)
71          return pred.next == curr;
72        node = node.next;
73      }
74      return false;
75    }
```

Aby metoda remove została zagłodzona ‘validate’ powinno zawsze zwracać ‘fałsz’.

1) Zatrzymajmy ‘A’ w linii 33 przed próbą wzięcia zamków.

2) Załóżmy teraz, że wątek ‘B’ dodaje między ‘pred_A’ oraz ‘curr_A’ węzeł ‘w’.

3) Dajemy wątkowi ‘A’ kontynuować i okazuje się, że ‘validate’ zwróci fałsz.

4) W czasie gdy ‘A’ zaczyna pętlę ‘while’ od nowa usuwamy węzeł ‘w’.