Multicore Programming
Programming Assignment 3 Report

1.Pseudocode Analysis

1.1 add operation

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    List<Node<T>> list = locateWindow(key);
    Node<T> pred = list.get(0);
    Node<T> cur = list.get(1);
    pred.lock();
    try {
      cur.lock();
      try {
        if (validate(pred, cur)) {
          if (cur.key == key)
            return false;
          else {
            // store the node which is currently being removed
            // if it exists, otherwise passing null instead
            Node<T> node = new Node<>(item);
            node.next = cur;
            pred.next = node;
            return true;
          }
        }
      } finally {
        cur.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

The above pseudo code is for the add operation, the original idea is from the textbook (The Art of Multiprocessor Programming Revised 1st Edition). First, we locate the window using the given item's hashcode as key. Then, we lock the window to enable mutual exclusion. Before the actual add operation, we validate the window to make sure that the window (pred and cur node) is still valid. If the window is no longer valid, return to the beginning of the while loop. If the window is valid, check whether the key already exists in the list, if it is, return false, otherwise, create a new node, insert the new node into the list. At last, unlocking the window to exit the add operation.

1.2 remove operation

```
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        List<Node<T>> list = locateWindow(key);
        Node<T> pred = list.get(0);
        Node<T> cur = list.get(1);
        pred.lock();
        try {
            cur.lock();
            try {
                if (validate(pred, cur)) {
                    if (cur.key != key)
                        return false;
                    else {
                        cur.marked = true;
                        pred.next = cur.next;
                        return true;
                    }
                }
            } finally {
                cur.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
}
```

The above pseudo code is for the remove operation , the original idea is from the text book. Same as the add operation, we first locate the window and try to lock pred and cur node. Then, if the window passes the validation, the code executes the actual remove operation. First, we check whether the key is in the list, if it is, we mark the cur node to logically remove the item from the list, otherwise we directly return false. Then, we set pred's next pointer points to cur's next node. At last, unlocking the window to exit remove operation.

1.3 replace operation

```
  public boolean replace(T oldItem, T newItem) {
    int oldKey = oldItem.hashCode();
    int newKey = newItem.hashCode();
    while (true) {
      List<Node<T>> oldWindowList = locateWindow(oldKey);
      Node<T> removePred = oldWindowList.get(0);
      Node<T> removeCur = oldWindowList.get(1);
      List<Node<T>> newWindowList = locateWindow(newKey);
      Node<T> addPred = newWindowList.get(0);
      Node<T> addCur = newWindowList.get(1);
      if (removePred.key == addPred.key && removeCur.key == addCur.key) {
        removePred.lock();
        try {
          removeCur.lock();
          try {
            if (validate(removePred, removeCur))
              return singleWindowReplace(removePred, removeCur, oldKey, newKey,
newItem);
          } finally {
            removeCur.unlock();
          }
        } finally {
          removePred.unlock();
        }
      } else if (removeCur.key <= addPred.key) {
        removePred.lock();
        try {
          removeCur.lock();
          try {
            addPred.lock();
            try {
              addCur.lock();
              try {
                if (validate(removePred, removeCur) && validate(addPred, addCur)) {
                  return doubleWindowReplace(removePred, removeCur, addPred,
addCur, oldKey, newKey, newItem);
                }
              } finally {
                addCur.unlock();
              }
            } finally {
              addPred.unlock();
            }
          } finally {
            removeCur.unlock();
          }
        } finally {
```

```
            removePred.unlock();
          }

        } else if (addCur.key <= removePred.key) {
          addPred.lock();
          try {
            addCur.lock();
            try {
              removePred.lock();
              try {
                removeCur.lock();
                try {
                  if (validate(addPred, addCur) && validate(removePred, removeCur)) {
                    return doubleWindowReplace(removePred, removeCur, addPred,
addCur, oldKey, newKey, newItem);
                  }
                } finally {
                  removeCur.unlock();
                }
              } finally {
                removePred.unlock();
              }
            } finally {
              addCur.unlock();
            }
          } finally {
            addPred.unlock();
          }
        }
      }
    }
```

The above pseudo code is for the replace operation. There are three cases in the replace operation:

1. oldItem's located window overlaps with newItem's located window. In this scenario, we only need to lock one window. The logic in the singleWindowReplace function is to replace oldItem with newItem, add the item if it is not in the list or remove the item if it is already in the list. For detailed explanation, please see below singleWindowReplace function analysis.
2. oldItem's located window is before newItem's located window. In this scenario, we first lock the oldItem's window and then lock the newItem's window. Note, the oldItem window's cur node may be the same as the newItem window's pred node, however, it doesn't matter when locking them, since we use Java's ReentrantLock, which allows multiple locking and unlocking in the same thread. After locking two windows, we need to validate whether the two windows are still valid. Then, we pass arguments to the doubleWindowReplace function. For detailed explanation of this function, please see the below analysis for doubleWindowReplace function.

3. newItem's located window is before oldItem's located window. In this scenario, we first lock the newItem's window and then lock the newItem's window. Same as the second scenario, we validate the two windows and then pass arguments to the doubleWindowReplace function.

```
private boolean singleWindowReplace(Node<T> pred, Node<T> cur, int oldKey, int
newKey, T newItem) {
      System.out.println("Enter single window replace");
      boolean result = false;
      Node<T> node = null;
      if (cur.key != newKey) {
         if (cur.key == oldKey) {
            cur.key = newKey;
            cur.item = newItem;
         } else {
            node = new Node<T>(newItem);
            node.next = cur;
            pred.next = node;
            increExpectSize();
         }
         result = true;
      } else {
         if (cur.key == oldKey) {
            cur.marked = true;
            pred.next = cur.next;
            result = true;
            decreExpectSize();
         }
      }
      System.out.println("Exit single window replace");
      return result;
   }
```

The above pseudo code is used for when there is only one window in replace operation. In this function, we only have one window, and we need to consider four scenarios:

1. If newItem is not in the list and oldItem is in the list, we replace cur node's key and item field with the newItem. This is same as we add a new item and delete the old item.
2. If newItem is not in the list and oldItem is not in the list, we only need to add a newItem into the list.
3. If newItem is in the list and oldItem is in the list, we only need to remove the oldItem from the list.
4. If newItem is in the list and oldItem is not in the list, do nothing.

```
private boolean doubleWindowReplace(Node<T> removePred, Node<T> removeCur,
Node<T> addPred, Node<T> addCur, int removeKey, int addKey, T addItem) {
```

```
        System.out.println("Enter double window replace");
        boolean result = false;
        Node<T> node = null;
        // if add key not exits
        if (addCur.key != addKey) {
            if (removeCur.key == removeKey) {
                node = new Node<T>(addItem, removeCur);
            } else {
                node = new Node<T>(addItem);
            }
            node.next = addCur;
            addPred.next = node;
            result = true;
            increExpectSize();
        }
        // if remove key exists
        if (removeCur.key == removeKey) {
            removeCur.marked = true;
            removePred.next = removeCur.next;
            if (node != null)
                node.removingNode = null;
            result = true;
            decreExpectSize();
        }
        System.out.println("Exit double window replace");
        return result;
    }
```

The above pseudo code is used for when there are two windows in the replace operation. If the newItem is not in the list, we add the newItem. The important mechanism to achieve atomicity for replace operation is to **store the oldItem's (if exists) node information in the newly added node**. In this way, if a concurrent thread performs contains operation, and find a node pointing to a removing node and the removing node is not marked, we can conclude that the node is currently under replace operation and not completed. Therefore, the contains operation should return false in this scenario. Then, when the removing oldItem operation completes, other threads which traverse the newly added node will immediately know that the node is now valid and can be accessed. Also, we reset the newly added node's pointer(pointing to the removing node) to null to reduce the useless information in the node.

1.4 contains operation

```
public boolean contains(T item) {
```

```
    int key = item.hashCode();
    Node<T> cur = head;
    while (cur.key < key) {
      cur = cur.next;
    }

    boolean result = cur.key == key && !cur.marked;
    if (cur.removingNode != null)
      result = result && cur.removingNode.marked;
    return result;
  }
```

The above pseudo code is for the contains operation. The difference between the contains operation in this program and that in the text book is that here we need to check whether the node is pointing to a removing node and whether the removing node is marked or not. As long as the pointing removing node is marked, the contains method will atomically notice that the replace operation finishes and the item is now valid in the list.

1.5 validate operation

```
  private boolean validate(Node<T> pred, Node<T> cur) {
    return !pred.marked && pred.next == cur;
  }
```

The validate operation's idea is originally from the text book. Since we validate the window within the lock block, we only need to check the pred node is not marked and pred node is still pointing to cur node. In this way, we can validate the window.

2.Test Strategy

2.1 Sanity Check

For the testing part, I create a small key space with only 10 items, and every operation will randomly pick an item from the key space. Also, I test 10 concurrent threads with 1000000 operations of each thread. After execution of all threads, I check the sanity of the list. If the list's key is not in ascending order or there exists duplicate keys, the list is not legal. I also print out the keys of the whole list (including sentinel nodes) when checking sanity of the list.

2.2 Size Check

How to check whether the list is what we expect to? I keep track of a field called **expectSize** in the my LinkedList. This field will atomically increase one if a node has been added to the list and decrease one if a node has been deleted from the list. After all threads' operations complete, our final LinkedList's size should match the expectSize.

After checking with different proportion of add, remove, replace, contains operations, I conclude that the program can produce the desired linked list correctly.

Note: I test my program with 30% add operation, 30% remove operation, 30% replace operation and 10% contains operation. One of the final linked list is as follows:

-2147483648 1 2 3 4 5 7 8 11 13 15 2147483647
The list is legal? true
Actual list size: 10
Expect list size: 10