**Laborator 3**

*LRU cache*

Create a data structure that supports the following operations: access and remove. The access operation inserts the item onto the data structure if it’s not already present. The remove operation deletes and returns the item that was least recently accessed.

*Hint*: Maintain the items in order of access in a doubly linked list, along with pointers to the first and last nodes. Use a symbol table with keys = items, values = location in linked list. When you access an element, delete it from the linked list and reinsert it at the beginning. When you remove an element, delete it from the end and remove it from the symbol table.

public class LRUCache<Item>

LRUCache()

int size()

void access(Item item)

Item remove()

Klasa ndihmëse:

* DoublyLinkedList.java
* SeparateChainingHashTable
* Queue.java

**DoublyLinkedList.java**

**import** java.util.Iterator;

**public** **class** DoublyLinkedList<Item> **implements** Iterable<Item> {

**public** **class** DoubleNode {

**public** Item item;

**public** DoubleNode previous;

**public** DoubleNode next;

}

**private** **int** size;

**private** DoubleNode first;

**private** DoubleNode last;

**public** **boolean** isEmpty() {

**return** size == 0;

}

**public** **int** size() {

**return** size;

}

**public** DoubleNode getFirstNode() {

**return** first;

}

**public** DoubleNode getLastNode() {

**return** last;

}

**public** Item get(**int** index) {

**if** (isEmpty()) {

**return** **null**;

}

**if** (index < 0 || index >= size) {

**throw** **new** IllegalArgumentException("Index must be between 0 and " + (size() - 1));

}

DoubleNode current;

**if** (index <= size / 2) {

current = first;

**int** count = 0;

**while** (count != index) {

current = current.next;

count++;

}

} **else** {

current = last;

**int** count = size - 1;

**while** (count != index) {

current = current.previous;

count--;

}

}

**return** current.item;

}

**public** **void** insertAtTheBeginning(Item item) {

DoubleNode oldFirst = first;

first = **new** DoubleNode();

first.item = item;

first.next = oldFirst;

**if** (oldFirst != **null**) {

oldFirst.previous = first;

}

// If the list was empty before adding the new item:

**if** (isEmpty()) {

last = first;

}

size++;

}

// Useful for LRU cache implementation

**public** DoubleNode insertAtTheBeginningAndReturnNode(Item item) {

DoubleNode oldFirst = first;

first = **new** DoubleNode();

first.item = item;

first.next = oldFirst;

**if** (oldFirst != **null**) {

oldFirst.previous = first;

}

// If the list was empty before adding the new item:

**if** (isEmpty()) {

last = first;

}

size++;

**return** first;

}

**public** **void** insertAtTheEnd(Item item) {

DoubleNode oldLast = last;

last = **new** DoubleNode();

last.item = item;

last.previous = oldLast;

**if** (oldLast != **null**) {

oldLast.next = last;

}

// If the list was empty before adding the new item:

**if** (isEmpty()) {

first = last;

}

size++;

}

**public** **void** insertBeforeNode(Item beforeItem, Item item) {

**if** (isEmpty()) {

**return**;

}

DoubleNode currentNode;

**for** (currentNode = first; currentNode != **null**; currentNode = currentNode.next) {

**if** (currentNode.item.equals(beforeItem)) {

**break**;

}

}

**if** (currentNode != **null**) {

DoubleNode newNode = **new** DoubleNode();

newNode.item = item;

DoubleNode previousNode = currentNode.previous;

currentNode.previous = newNode;

newNode.next = currentNode;

newNode.previous = previousNode;

**if** (newNode.previous == **null**) {

// This is the first node

first = newNode;

} **else** {

newNode.previous.next = newNode;

}

size++;

}

}

**public** **void** insertAfterNode(Item afterNode, Item item) {

**if** (isEmpty()) {

**return**;

}

DoubleNode currentNode;

**for** (currentNode = first; currentNode != **null**; currentNode = currentNode.next) {

**if** (currentNode.item.equals(afterNode)) {

**break**;

}

}

**if** (currentNode != **null**) {

DoubleNode newNode = **new** DoubleNode();

newNode.item = item;

DoubleNode nextNode = currentNode.next;

currentNode.next = newNode;

newNode.previous = currentNode;

newNode.next = nextNode;

**if** (newNode.next == **null**) {

// This is the last node

last = newNode;

} **else** {

newNode.next.previous = newNode;

}

size++;

}

}

**public** Item removeFromTheBeginning() {

**if** (isEmpty()) {

**return** **null**;

}

Item item = first.item;

**if** (first.next != **null**) {

first.next.previous = **null**;

} **else** { // There is only 1 element in the list

last = **null**;

}

first = first.next;

size--;

**return** item;

}

**public** Item removeFromTheEnd() {

**if** (isEmpty()) {

**return** **null**;

}

Item item = last.item;

**if** (last.previous != **null**) {

last.previous.next = **null**;

} **else** { // There is only 1 element in the list

first = **null**;

}

last = last.previous;

size--;

**return** item;

}

**public** **void** removeItem(Item item) {

**if** (isEmpty()) {

**return**;

}

DoubleNode currentNode = first;

**while** (currentNode != **null**) {

**if** (currentNode.item.equals(item)) {

removeItemWithNode(currentNode);

**break**;

}

currentNode = currentNode.next;

}

}

//Useful for LRU cache implementation

**public** **void** removeItemWithNode(DoubleNode doubleNode) {

**if** (doubleNode == **null**) {

**throw** **new** IllegalArgumentException("Node cannot be null");

}

**if** (isEmpty()) {

**return**;

}

DoubleNode previousNode = doubleNode.previous;

DoubleNode nextNode = doubleNode.next;

**if** (previousNode != **null**) {

previousNode.next = nextNode;

}

**if** (nextNode != **null**) {

nextNode.previous = previousNode;

}

**if** (doubleNode == first) {

first = nextNode;

}

**if** (doubleNode == last) {

last = previousNode;

}

size--;

}

**public** Item removeItemWithIndex(**int** nodeIndex) {

**if** (isEmpty()) {

**return** **null**;

}

**if** (nodeIndex < 0 || nodeIndex >= size()) {

**throw** **new** IllegalArgumentException("Index must be between 0 and " + (size() - 1));

}

**boolean** startFromTheBeginning = nodeIndex <= size() / 2;

**int** index = startFromTheBeginning ? 0 : size() - 1;

DoubleNode currentNode = startFromTheBeginning ? first : last;

**while** (currentNode != **null**) {

**if** (nodeIndex == index) {

**break**;

}

**if** (startFromTheBeginning) {

index++;

} **else** {

index--;

}

currentNode = startFromTheBeginning ? currentNode.next : currentNode.previous;

}

@SuppressWarnings("ConstantConditions") // If we got here, the node exists

Item item = currentNode.item;

removeItemWithNode(currentNode);

**return** item;

}

@Override

**public** Iterator<Item> iterator() {

**return** **new** DoublyLinkedListIterator();

}

**private** **class** DoublyLinkedListIterator **implements** Iterator<Item> {

**int** index = 0;

DoubleNode currentNode = first;

@Override

**public** **boolean** hasNext() {

**return** index < size();

}

@Override

**public** Item next() {

Item item = currentNode.item;

currentNode = currentNode.next;

index++;

**return** item;

}

}

}

**SeparateChainingHashTable.java**

c

**public** **class** SeparateChainingHashTable<Key, Value> {

**class** SequentialSearchSymbolTable<Key, Value> {

**private** **class** Node {

Key key;

Value value;

Node next;

**public** Node(Key key, Value value, Node next) {

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

}

**private** Node first;

**protected** **int** size;

**public** **int** size() {

**return** size;

}

**public** **boolean** isEmpty() {

**return** size == 0;

}

**public** **boolean** contains(Key key) {

**return** get(key) != **null**;

}

**public** Value get(Key key) {

**for** (Node node = first; node != **null**; node = node.next) {

**if** (key.equals(node.key)) {

**return** node.value;

}

}

**return** **null**;

}

**public** **void** put(Key key, Value value) {

**for** (Node node = first; node != **null**; node = node.next) {

**if** (key.equals(node.key)) {

node.value = value;

**return**;

}

}

first = **new** Node(key, value, first);

size++;

}

**public** **void** delete(Key key) {

**if** (first.key.equals(key)) {

first = first.next;

size--;

**return**;

}

**for** (Node node = first; node != **null**; node = node.next) {

**if** (node.next != **null** && node.next.key.equals(key)) {

node.next = node.next.next;

size--;

**return**;

}

}

}

**public** Iterable<Key> keys() {

Queue<Key> keys = **new** Queue<>();

**for** (Node node = first; node != **null**; node = node.next) {

keys.enqueue(node.key);

}

**return** keys;

}

}

**protected** **int** averageListSize;

**protected** **int** size;

**protected** **int** keysSize;

SequentialSearchSymbolTable<Key, Value>[] symbolTable;

**private** **static** **final** **int** ***DEFAULT\_HASH\_TABLE\_SIZE*** = 997;

**private** **static** **final** **int** ***DEFAULT\_AVERAGE\_LIST\_SIZE*** = 5;

// The largest prime <= 2^i for i = 1 to 31

// Used to distribute keys uniformly in the hash table after resizes

// PRIMES[n] = 2^k - Ak where k is the power of 2 and Ak is the value to subtract to reach the previous prime number

**protected** **static** **final** **int**[] ***PRIMES*** = {

1, 1, 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381,

32749, 65521, 131071, 262139, 524287, 1048573, 2097143, 4194301,

8388593, 16777213, 33554393, 67108859, 134217689, 268435399,

536870909, 1073741789, 2147483647

};

// The lg of the hash table size

// Used in combination with PRIMES[] to distribute keys uniformly in the hash function after resizes

**protected** **int** lgM;

**public** SeparateChainingHashTable() {

**this**(***DEFAULT\_HASH\_TABLE\_SIZE***, ***DEFAULT\_AVERAGE\_LIST\_SIZE***);

}

**public** SeparateChainingHashTable(**int** initialSize, **int** averageListSize) {

**this**.size = initialSize;

**this**.averageListSize = averageListSize;

symbolTable = **new** SequentialSearchSymbolTable[size];

**for** (**int** i = 0; i < size; i++) {

symbolTable[i] = **new** SequentialSearchSymbolTable<>();

}

lgM = (**int**) (Math.*log*(size) / Math.*log*(2));

}

**public** **int** size() {

**return** keysSize;

}

**public** **boolean** isEmpty() {

**return** keysSize == 0;

}

**protected** **int** hash(Key key) {

**int** hash = key.hashCode() & 0x7fffffff;

**if** (lgM < 26) {

hash = hash % ***PRIMES***[lgM + 5];

}

**return** hash % size;

}

**protected** **double** getLoadFactor() {

**return** ((**double**) keysSize) / (**double**) size;

}

**public** **boolean** contains(Key key) {

**if** (key == **null**) {

**throw** **new** IllegalArgumentException("Argument to contains() cannot be null");

}

**return** get(key) != **null**;

}

**public** **void** resize(**int** newSize) {

SeparateChainingHashTable<Key, Value> separateChainingHashTableTemp =

**new** SeparateChainingHashTable<>(newSize, averageListSize);

**for** (Key key : keys()) {

separateChainingHashTableTemp.put(key, get(key));

}

symbolTable = separateChainingHashTableTemp.symbolTable;

size = separateChainingHashTableTemp.size;

}

**public** Value get(Key key) {

**if** (key == **null**) {

**throw** **new** IllegalArgumentException("Argument to get() cannot be null");

}

**return** symbolTable[hash(key)].get(key);

}

**public** **void** put(Key key, Value value) {

**if** (key == **null**) {

**throw** **new** IllegalArgumentException("Key cannot be null");

}

**if** (value == **null**) {

delete(key);

**return**;

}

**int** hashIndex = hash(key);

**int** currentSize = symbolTable[hashIndex].size;

symbolTable[hashIndex].put(key, value);

**if** (currentSize < symbolTable[hashIndex].size) {

keysSize++;

}

**if** (getLoadFactor() > averageListSize) {

resize(size \* 2);

lgM++;

}

}

**public** **void** delete(Key key) {

**if** (key == **null**) {

**throw** **new** IllegalArgumentException("Argument to delete() cannot be null");

}

**if** (isEmpty() || !contains(key)) {

**return**;

}

symbolTable[hash(key)].delete(key);

keysSize--;

**if** (size > 1 && getLoadFactor() <= averageListSize / (**double**) 4) {

resize(size / 2);

lgM--;

}

}

**public** Iterable<Key> keys() {

Queue<Key> keys = **new** Queue<>();

**for** (SequentialSearchSymbolTable<Key, Value> sequentialSearchST : symbolTable) {

**for** (Key key : sequentialSearchST.keys()) {

keys.enqueue(key);

}

}

**return** keys;

}

}

**Queue.java**

import java.util.Iterator;

import java.util.NoSuchElementException;

public class Queue<Item> implements Iterable<Item> {

private Node<Item> first; // beginning of queue

private Node<Item> last; // end of queue

private int n; // number of elements on queue

// helper linked list class

private static class Node<Item> {

private Item item;

private Node<Item> next;

}

/\*\*

\* Initializes an empty queue.

\*/

public Queue() {

first = null;

last = null;

n = 0;

}

/\*\*

\* Returns true if this queue is empty.

\*

\* @return {@code true} if this queue is empty; {@code false} otherwise

\*/

public boolean isEmpty() {

return first == null;

}

/\*\*

\* Returns the number of items in this queue.

\*

\* @return the number of items in this queue

\*/

public int size() {

return n;

}

/\*\*

\* Returns the item least recently added to this queue.

\*

\* @return the item least recently added to this queue

\* @throws NoSuchElementException if this queue is empty

\*/

public Item peek() {

if (isEmpty()) throw new NoSuchElementException("Queue underflow");

return first.item;

}

/\*\*

\* Adds the item to this queue.

\*

\* @param item the item to add

\*/

public void enqueue(Item item) {

Node<Item> oldlast = last;

last = new Node<Item>();

last.item = item;

last.next = null;

if (isEmpty()) first = last;

else oldlast.next = last;

n++;

}

/\*\*

\* Removes and returns the item on this queue that was least recently added.

\*

\* @return the item on this queue that was least recently added

\* @throws NoSuchElementException if this queue is empty

\*/

public Item dequeue() {

if (isEmpty()) throw new NoSuchElementException("Queue underflow");

Item item = first.item;

first = first.next;

n--;

if (isEmpty()) last = null; // to avoid loitering

return item;

}

/\*\*

\* Returns a string representation of this queue.

\*

\* @return the sequence of items in FIFO order, separated by spaces

\*/

public String toString() {

StringBuilder s = new StringBuilder();

for (Item item : this) {

s.append(item);

s.append(' ');

}

return s.toString();

}

/\*\*

\* Returns an iterator that iterates over the items in this queue in FIFO order.

\*

\* @return an iterator that iterates over the items in this queue in FIFO order

\*/

public Iterator<Item> iterator() {

return new LinkedIterator(first);

}

// a linked-list iterator

private class LinkedIterator implements Iterator<Item> {

private Node<Item> current;

public LinkedIterator(Node<Item> first) {

current = first;

}

public boolean hasNext() {

return current != null;

}

public Item next() {

if (!hasNext()) throw new NoSuchElementException();

Item item = current.item;

current = current.next;

return item;

}

}

}