Kjøretidsanalyser - gux006

Size: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid.

```
public int size() {
    return size;
}
```

addFirst: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid, uanhengig av hvor mange elementer som allerede finnes i tvekøen.

```
public void addFirst(E elem) throws DequeFullException {

    if (size == capacity) {
        throw new DequeFullException("Deque is full!");
    }
    if (isEmpty()) {
        theArray[firstIndex] = elem;
    } else {
        firstIndex = (firstIndex - 1 + capacity) % capacity;
        theArray[firstIndex] = elem;

    }
    size++;
}
```

pullFirst: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid, uanhengig av hvor mange elementer som allerede finnes i tvekøen.

```
public E pullFirst() throws DequeEmptyException {
    E objectToReturn = null;
    if (isEmpty()) {
        throw new DequeEmptyException("Deque is empty!");
    } else {
        objectToReturn = theArray[firstIndex];
        theArray[firstIndex] = null;
        firstIndex = (firstIndex + 1) % capacity;
        size--;

    }
    return objectToReturn;
}
```

peekFirst: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid.

```
public E peekFirst() throws DequeEmptyException {
    if (size == 0) {
        throw new DequeEmptyException("Deque is empty");
    }
    return theArray[firstIndex];
}
```

addLast: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid.

```
public void addLast(E elem) throws DequeFullException {
    if (size == capacity) {
        throw new DequeFullException("Deque is full!");
    } else {
        lastIndex = (lastIndex + 1) % capacity;
        theArray[lastIndex] = elem;
        size++;

    }

}
```

pullLast: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid, uanhengig av hvor mange elementer som allerede finnes i tvekøen.

```
public E pullLast() throws DequeEmptyException {
    E objectToReturn = null;
    if (isEmpty()) {
        throw new DequeEmptyException("Deque is empty!");
    } else {
        objectToReturn = theArray[lastIndex];
        theArray[lastIndex] = null;
        lastIndex = (lastIndex - 1 + capacity) % capacity;
        size--;

    }
    return objectToReturn;
}
```

peekLast: O(1)

Kompleksiteten av denne metoden er O(1), da den alltid vil bruke konstant tid.

```
public E peekLast() throws DequeEmptyException {
    if (isEmpty()) {
        throw new DequeEmptyException("Deque is empty!");
    } else {
        return theArray[lastIndex];
    }
}
```

clear: O(n)

Kompleksiteten av denne metoden er O(n), da tidskompleksiteten vil øke lineært med antall elementer som finnes i tvekøen.

```
public void clear() {
    for (E elem : theArray) {
        elem = null;
    }
    size = 0;
    firstIndex = 0;
    lastIndex = 0;

}
```

contains: O(n)

Kompleksiteten av denne metoden er O(n), da tidskompleksiteten vil øke lineært med antall elementer som finnes i tvekøen. Dette er gitt muligheten for «worst case scenario» hvor elementet finnes bakerst i tvekøen.

```
public boolean contains(Object elem) {
    for (E element : theArray) {
        if (elem == element) {
            return true;
        } else if (elem.equals(element)) {
            return true;
        }
    }
    return false;
}
```

toArray: O(n)

Kompleksiteten av denne metoden er O(n), da tidskompleksiteten vil øke lineært med antall elementer som finnes i tvekøen.

```
public E[] toArray() {
    E[] arrayToReturn = (E[]) new Object[size];
    int index = 0;
    int start = firstIndex;
    for (int i = 0; i < theArray.length; i++) {
        if (theArray[(i + start) % capacity] != null) {
            arrayToReturn[index] = theArray[(i + start) % capacity];
            index++;
        }

    }
    return arrayToReturn;
}
```

Oppgave h)

De korresponderende metodene addFirst, pullFirst, peekFirst, addLast, pullLast og peekLast har like stor kjøretid. Disse metodene krever ingen form for itterering e.l, og vil bruke konstant tid på å gjennføres uavhengig av størrelsen på tvekøen. Tidskompleksiten blir altså ikke påvirket av tvekøen.

Oppgave i)

Metoden average(Integer[] values) har tidskompleksitet O(n), da den vil bruke en lineær mengde tid basert på størrelsen på *values-arrayet*.

Metoden slidingAvg(Stack<Integer> values, int width) har tidskompleksitet O(n^2). Den vil itterere gjennom hvert element i *values-stacken*. For hver itterering, legges et nytt tall til i window-dequet. Deretter utregnes gjennomsnittet av dequet så langt ved å kalle på *average-metoden.* Dette gjøres helt til hvert tall i *values* er blitt lagt til i *window,* og gjennomsnittet så langt er regnet ut og lagt til i *avergages,* som er et array representativt for det bevegende gjennomsnittet. Til slutt returneres *avrages-arrayet.*