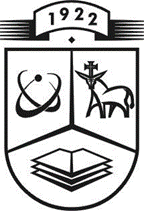
KAUNO TECHNOLOGIJOS UNIVERSITETAS



**TECHNOLOGINIO PROJEKTO**

**ATASKAITA**

**Studentas**

**Matas Palujanskas**

**KAUNAS 2022**

**TURINYS**

[Dekas 3](#_Toc121689741)

[Dvikrypčio susietojo sąrašo realizacija 3](#_Toc121689742)

[Ciklinio masyvo realizacija 6](#_Toc121689743)

[Greitaveikos tyrimas 9](#_Toc121689744)

[Asimptotiniai sudėtingumai 9](#_Toc121689745)

[Kompiuterio charakteristikos 9](#_Toc121689746)

[Greitaveikos rezultatai 10](#_Toc121689747)

[Išvados 16](#_Toc121689748)

# Dekas

Dekas arba dviguba eilė (angl. Double Ended Queue) – tai tiesinė duomenų struktūra, kurioje elementus galima išimti ir įdėti tiek iš priekio, tiek iš galo. Dekas gali būti taikomas visur, kur galima taikyti steką arba eilę. Algoritmuose, kuriuose reikia dirbti su slenkančiu langu.

Deke saugomi duomenys yra Integer tipo, gali būti įdedami, tiek į pradžią tiek į galą.

Dekas buvo realizuotas dvikrypčio susietojo sąrašo pagrindu ir ciklinio masyvo pagrindu.

Susietasis dvikryptis sąrašas leidžia šalinti ir įterpti tiek eilės priekyje, tiek gale. Dvikrypčiame susietajame sąraše šios operacijos yra O(1) sudėtingumo, todėl ši duomenų struktūra puikiai tinka dekui realizuoti.

Realizuojant deką ciklinio masyvo pagrindu, algoritmas yra panašus kaip ir realizuojant eilę, tačiau šiuo atveju reikia realizuoti daugiau operacijų ir atlikti papildomų tikrinimų. Reikia leisti pradžios (angl. front) ir pabaigos (angl. rear) rodyklėms judėti laisvai po masyvą tiek pirmyn, tiek atgal.

# Dvikrypčio susietojo sąrašo realizacija

//Deque LinkedList class  
public class DequeLinkedList<E> implements Deque<E> {  
  
 private Node<E> head; // Link to the first element  
 private Node<E> tail; // Link to the last element  
 private int size = 0; // Size  
  
 public DequeLinkedList()  
 {  
 tail = null;  
 head = null;  
 }  
  
 //Adding element to front  
 @Override  
 public void addFirst(E elementToAdd) {  
 if (elementToAdd == null) throw new IllegalArgumentException();  
  
 Node<E> node = new Node<>(null, elementToAdd, null);  
 if (head == null) {  
 head = node;  
 tail = head;  
 }  
 else  
 {  
 node.next = head;  
 head.previous = node;  
 head = node;  
 }  
 size++;  
 }  
  
 //Returning first element  
 @Override  
 public E getFirst() {  
 if (this.head == null)  
 {  
 return null;  
 }  
 return this.head.value;  
 }

//Removing first element  
 @Override  
 public E removeFirst() {  
 if (isEmpty())  
 {  
 throw new NoSuchElementException();  
 }  
 Node<E> toReturn = head;  
 head = head.next;  
 size--;  
 return toReturn.value;  
 }  
  
 //Adding element to the end  
 @Override  
 public void addLast(E elementToAdd) {  
 if (elementToAdd == null) throw new IllegalArgumentException();  
  
 Node<E> node = new Node<>(null, elementToAdd, null);  
 node.value = elementToAdd;  
  
 if (head == null)  
 {  
 tail = node;  
 head = tail;  
 }  
 else  
 {  
 tail.next = node;  
 node.previous = tail;  
 tail = node;  
 }  
 size++;  
 }  
  
 //Returning last element  
 @Override  
 public E getLast() {  
 if (this.tail == null)  
 {  
 return null;  
 }  
 return this.tail.value;  
 }  
  
 //Removing last element  
 @Override  
 public E removeLast() {  
 if (isEmpty())  
 {  
 throw new NoSuchElementException();  
 }  
 Node<E> toReturn = tail;  
 tail = tail.previous;  
 tail.next = null;  
 size--;  
 return toReturn.value;  
 }  
  
 //Checking if list is empty  
 @Override  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 //Node class  
 private static class Node<E> {  
 private E value;  
 private Node<E> previous;  
 private Node<E> next;  
  
 private Node(Node<E> previous, E current, Node<E> next) {  
 this.value = current;  
 this.previous = previous;  
 this.next = next;  
 }  
 }  
 public int size()  
 {  
 return this.size;  
 }  
 public String toString()  
 {  
 StringBuilder deque = new StringBuilder();  
  
 if (isEmpty())  
 {  
 deque.append("Deque is empty.");  
 return deque.toString();  
 }  
  
 Node<E> first = head;  
 deque.append("Deque elements: [ ").append(first.value).append(" ");  
 while (first.next != null)  
 {  
 deque.append(first.next.value).append(" ");  
 first = first.next;  
 }  
 deque.append("]");  
  
 return deque.toString();  
 }  
}

# Ciklinio masyvo realizacija

//Deque Array class  
public class DequeArray<E> implements Deque<E> {  
  
 private Object[] elements;  
 private int headIndex; //head index  
 private int tailIndex; //tail index  
 private int numberOfElements; //amount of elements  
  
 public DequeArray(int capacity) {  
 if (capacity < 1) {  
 throw new IllegalArgumentException("Capacity must be 1 or higher");  
 }  
 elements = new Object[capacity];  
 }  
  
 //Ensuring capacity  
 private void ensureCapacity()  
 {  
 if (size() == elements.length)  
 {  
 Object[] ensuredArray = new Object[elements.length \* 2];  
 for (int i = 0; i < numberOfElements; i++)  
 {  
 ensuredArray[i] = elements[headIndex];  
 headIndex = (headIndex + 1) % elements.length;  
 }  
 headIndex = 0;  
 tailIndex = numberOfElements;  
 elements = ensuredArray;  
 }  
 }  
  
 //Adding element to the front  
 @Override  
 public void addFirst(E element) {  
 ensureCapacity();  
  
 headIndex = decreaseIndex(headIndex);  
 elements[headIndex] = element;  
 numberOfElements++;  
 }  
  
 //Returning first element  
 @Override  
 public E getFirst() {  
 return elementAtHead();  
 }  
  
 //Removing first element  
 @Override  
 public E removeFirst() {  
 E element = elementAtHead();  
 elements[headIndex] = null;  
 headIndex = increaseIndex(headIndex);  
 numberOfElements--;  
 return element;  
 }  
  
 //Adding element to the end  
 @Override  
 public void addLast(E element) {  
 ensureCapacity();  
  
 elements[tailIndex] = element;  
 tailIndex = increaseIndex(tailIndex);  
 numberOfElements++;  
 }  
  
 //Returning last element  
 @Override  
 public E getLast() {  
 return elementAtTail();  
 }  
  
 //Removing last element  
 @Override  
 public E removeLast() {  
 E element = elementAtTail();  
 tailIndex = decreaseIndex(tailIndex);  
 elements[tailIndex] = null;  
 numberOfElements--;  
 return element;  
 }  
  
 //Checking if array is empty  
 @Override  
 public boolean isEmpty() {  
 return false;  
 }  
 private E elementAtHead() {  
 if (isEmpty()) {  
 throw new NoSuchElementException();  
 }  
 E element = (E) elements[headIndex];  
 return element;  
 }  
  
 private E elementAtTail() {  
 if (isEmpty()) {  
 throw new NoSuchElementException();  
 }  
 E element = (E) elements[decreaseIndex(tailIndex)];  
 return element;  
 }  
  
 private int decreaseIndex(int index) {  
 index--;  
 if (index < 0) {  
 index = elements.length - 1;  
 }  
 return index;  
 }  
  
 private int increaseIndex(int index) {  
 index++;  
 if (index == elements.length) {  
 index = 0;  
 }  
 return index;  
 }  
  
 //Size of an array  
 public int size()  
 {  
 return numberOfElements;  
 }  
  
 //ToString method  
 public String toString()  
 {  
 String dequeElements = "";  
  
  
 for (int i = 0,j=0; j<elements.length;i=(i+1)% elements.length,j++)  
 {  
  
 if (elements[i]!=null)  
 dequeElements = dequeElements + elements[i].toString()+ " ";  
 else if (elements[j]!=null)  
 {  
 //System.out.println("j is: " +j);  
 dequeElements = dequeElements + elements[j].toString()+ " ";  
 }  
 }  
 return dequeElements;  
 }  
}

# Greitaveikos tyrimas

Atlikti greitaveikos tyrimai naudojant Java JMH ir palyginti atskirų realizacijų metodai (addFirst, addLast, removeFirst, removeLast, getFirst, getLast).

Testuojant greitaveiką, buvo generuojami atsitiktiniai Integer tipo elementai.

# Asimptotiniai sudėtingumai

Paveikslėlis, kuriame yra stalas

Automatiškai sugeneruotas aprašymas

# Kompiuterio charakteristikos

Procesorius: AMD Ryzen 7 4800H 8 -Core Processor 2.90 GHz

Installed RAM: 16 GB, (usable 15.4 GB)

# Greitaveikos rezultatai

addFirst metodo greitaveikos rezultatai:

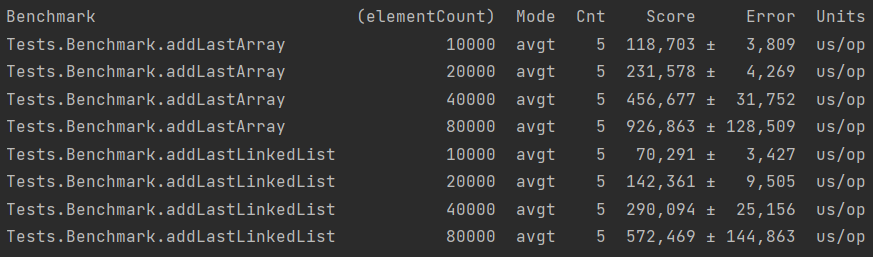
Paveikslėlis, kuriame yra žinutė, elektroniniai prietaisai

Automatiškai sugeneruotas aprašymas

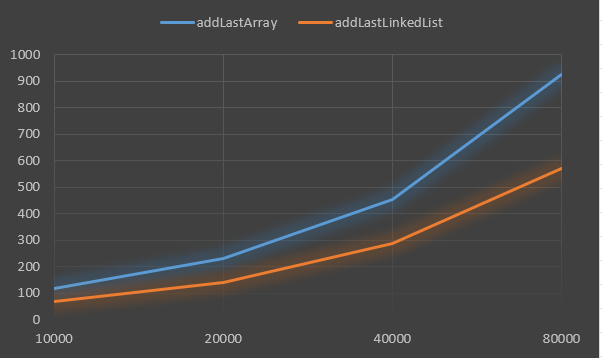
Testavimo grafikas:



addLast greitaveikos rezultatai:



Testavimo grafikas:

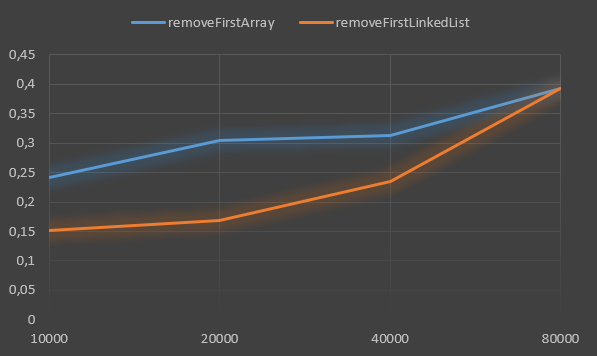


removeFirst greitaveikos rezultatai:

Paveikslėlis, kuriame yra žinutė, elektroniniai prietaisai

Automatiškai sugeneruotas aprašymas

Testavimo grafikas:

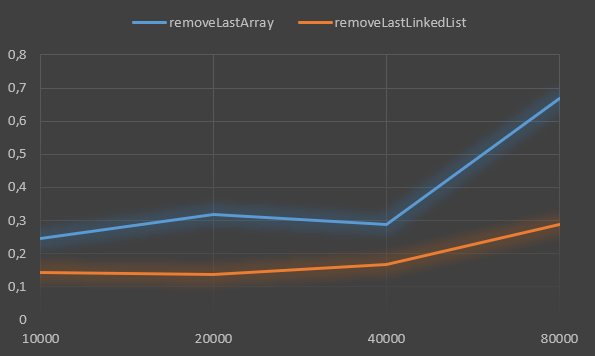


removeLast greitaveikos rezultatai:

Paveikslėlis, kuriame yra žinutė, elektroniniai prietaisai, klaviatūra, kompiuteris

Automatiškai sugeneruotas aprašymas

Testavimo grafikas:

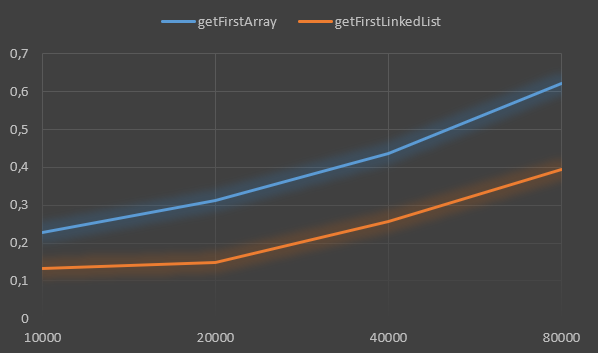


getFirst greitaveikos rezultatai:

Paveikslėlis, kuriame yra žinutė, elektroniniai prietaisai, klaviatūra

Automatiškai sugeneruotas aprašymas

Testavimo grafikas:

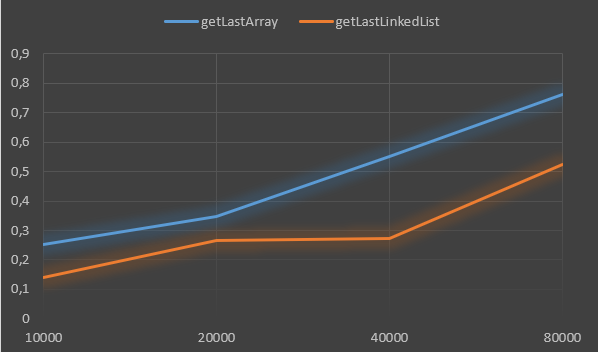


getLast greitaveikos rezultatai:

Paveikslėlis, kuriame yra žinutė, elektroniniai prietaisai, klaviatūra, kompiuteris

Automatiškai sugeneruotas aprašymas

Testavimo grafikas:



# Išvados

* Dekas buvo sėkmingai realizuotas tiek susietojo sąrašo, tiek ciklinio masyvo pagrindu.
* Testuojami metodai atitiko teorinius asimptotinius sudėtingumus, visi jie buvo O(1).
* Iš gautų greitaveikos rezultatų galime pastebėti, jog Deko realizacija susietojo sąrašo pagrindu veikia greičiau nei ciklinio masyvo. Taip yra todėl, nes masyvo pagrindu realizuotas Dekas, didėjant elementų kiekiui turi didinti ir talpą, tuo tarpu susietajame sąraše elementai yra „kabinami“ į grandinėlę. Gauti rezultatai atitinka ir įžangoje aptartas skirtingas Deko realizacijas, jog dvikrypčio susieto sąrašo duomenų struktūra puikiai tinka Dekui realizuoti.