PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET METROPOLITAN BEOGRAD

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Implementacija listi, stekova, redova i prioritetnih redova

Lekcija 01

PRIRUČNIK ZA STUDENTE

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Lekcija 01

IMPLEMENTACIJA LISTI, STEKOVA, REDOVA I PRIORITETNIH REDOVA

- Implementacija listi, stekova, redova i prioritetnih redova
- Poglavlje 1: Apstrakcije liste u Javi
- Poglavlje 2: Lista bazirana na nizu
- Poglavlje 3: Povezane liste
- Poglavlje 4: Stekovi i redovi čekanja
- Poglavlje 5: Stekovi i redovi u Javi
- Poglavlje 6: Prioritetni red čekanja
- Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

Uvod

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upozanju sa:

- Tipovima struktura podataka koje koristimo za liste
- Implementacijom listi koje su bazirane na nizovima i povezanim strukturama podataka
- Implementacijom generičkih klasa u Javi za rad sa listama
- Izborom najefikasnije strukture podataka za postavljeni tip problema.
- Implementacijom steka i reda korišćenjem niza i povezane liste
- Implementacijom prioritenih redova korišćenjem hipa.

Liste, stekovi, redovi i redovi sa prioritetom su uobičajene strukture podataka koje se tipično obradjuju u okviru kursa Strukture Podataka. Ove strukture su podržane od strane Java API, i njhovu upotrebu smo već predstavili u prethodnom predmetu (KI201). U okviru ove lekcije ćemo prikazati kako to izgleda "ispod haube", tj kako su ove strukture podataka ustvari implementirane. Implementacija setova i mapa biće obrađena u okviru lekcije o heširanju. Kroz ove primere ćete steći sposobnost da projektujete i impementirate bilo koje tipove struktura podataka.

Poglavlje 1

Apstrakcije liste u Javi

LISTE

Lista je apstraktni tip podataka koji implementira konačan uređen skup vrednosti, gde se iste vrednosti mogu ponoviti više puta. Mogu se kreirati pomoću nizova ili povezanih struktura

Struktura podataka je kolekcija (zbirka) podataka organizovanih na neki način. Struktura ne služi samo za smeštaj (memorisanje) podataka, već
takođe podržava operacije neophodne za pristup podacima i za rad sa njima.

Na strukturu podataka, u OO smislu, se može gledati kao na neki kontejner koji čuva druge objekte koji se nazivaju podacima ili elementima.

Lista ili sekvenca je apstraktni tip podataka koji sadrži konačan uređen skup vrednosti, gde se iste vrednosti mogu ponoviti više puta. Liste su osnovni primer kontejnera, jer sadrže druge vrednosti.

Liste se dele u dve osnovne grupe:

- Liste bazirane na nizovima ili jednostavno – nizovi (array list)
- Liste bazirane na povezanim strukurama ili jednostavno - povezane liste (linked list).

S obzirom da su liste veoma važne strukture podataka u ovom delu ćemo korišćenjem osnovnih konstrukcija i šablona opisati liste kao apstraktne tipove podataka (ADT).

Kao i bilo koji drugi tip podatka, ADT se sastoji iz tri osnovna dela:

- ime ADT-a (ili ime apstraktnog tipa podatka),
- skup vrednosti koji čine ADT (oblast ili domen apstraktnog tipa) i
- skup operacija nad strukturom podatka.

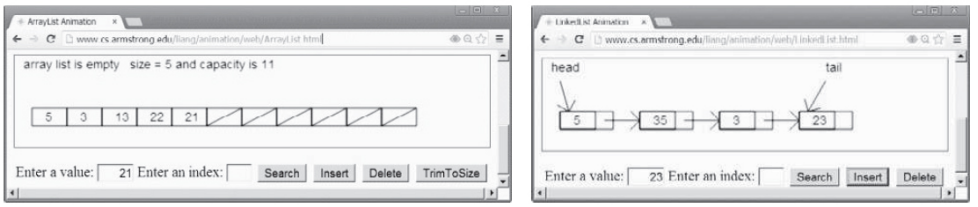
Implementacija liste kao strukture podataka može da obezbedi neke od sledećih operacija:

- konstruktor za kreiranje prazne liste
- operacija za testiranje da li je ili ne lista prazna
- operacija za pretragu elemenata liste
- operacija za dodavanje elementa u listu
- operacija za utvrđivanje prvog elementa liste ("*glava*")
- operacija za definisanje repa liste.

ANIMACIJA RADA KLASA ARRAYLIST I LINKEDLIST

Posetite sledeće linkove za interaktivni demo rada klase ArrayList i LinkedList

Da bi ste videli interaktivni demo koji opisuje ra liste bazirane na nizu i povezane liste posetite linkove www.cs.armstrong.edu/liang/animation/ArrayListAnimation.html i www.cs.armstrong.edu/liang/animation/LinkedListAnimation.html, kao što je prikazano na Slici-1.



(a) Array list animation (b) Linked list animation

Slika 1.1 Animirani alat koji omogućava uvid u način rada liste bazirane na nizu (array lists) i povezane liste (linked lists)

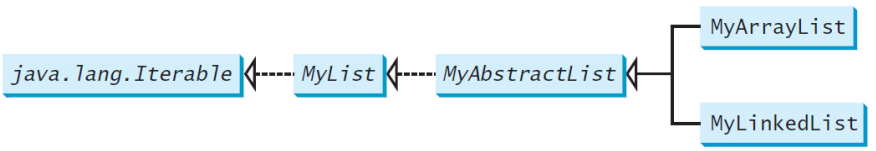
HIJERARHIJA INTERFEJSA I KLASA ZA RAD SA LISTAMA

Java Collection okvir sadrži dve klase za rad sa listama, i to: ArrayList i LinkedList. U prvoj klasi se koristi niz za čuvanje elemenata, dok se u drugoj klasi koristi povezana lista

Postoje dva načina da se implementira lista. Jedan je da se koristi niz za čuvanje elemenata, dok je drugi način preko povezanih listi. Java Collection okvir sadrži dve klase za rad sa listama, i to: ArrayList i LinkedList. U okviru ove lekcije će biti opisan postupak implementacije ove dve klase.

U nastavku, radi lakšeg opisa, nazvaćemo ove dve klase za liste, kao *MyArrayList* i *MyLinkedList*. Ove dve klase imaju zajedničke operacije ali drugačiju implementaciju.

Nazovimo osnovni interfejs za rad sa listama kao *MyList* i takođe, a osnovnu apstraktnu klasu za liste kao *MyAbstractList*. Na narednoj Slici-2 je prikazana relacija izmedju *MyList*, *MyAbstractList*, *MyArrayList*, i *MyLinkedList*.



Slika 1.2 Interfejs MyList definiše osnovni interfejs za klase MyAbstractList, MyArrayList, i MyLinkedList

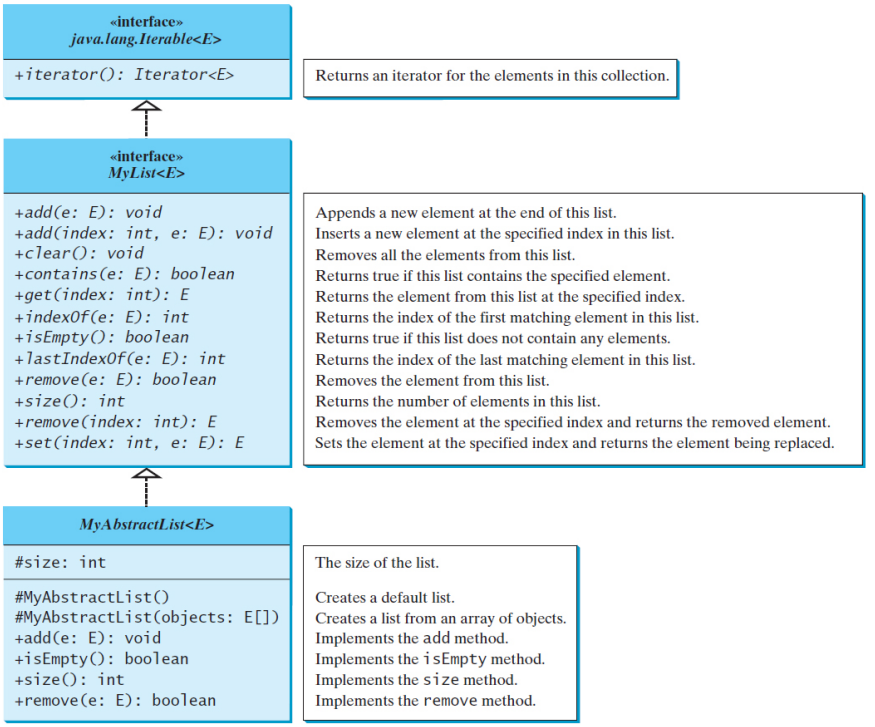
Naredni listing predstavlja izvorni kod interfejsa *MyList*.

```
public interface MyList<E> extends java.lang.Iterable<E> {
  /** Add a new element at the end of this list */
  public void add(E e);

  /** Add a new element at the specified index in this list */
  public void add(int index, E e);

  /** Clear the list */
  public void clear();

  /** Return true if this list contains the element */
  public boolean contains(E e);

  /** Return the element from this list at the specified index */
  public E get(int index);

  /** Return the index of the first matching element in this list.
   *  Return -1 if no match. */
  public int indexOf(E e);

  /** Return true if this list contains no elements */
  public boolean isEmpty();

  /** Return the index of the last matching element in this list
   *  Return -1 if no match. */
  public int lastIndexOf(E e);

  /** Remove the first occurrence of the element o from this list.
   *  Shift any subsequent elements to the left.
   *  Return true if the element is removed. */
  public boolean remove(E e);

  /** Remove the element at the specified position in this list
   *  Shift any subsequent elements to the left.
   *  Return the element that was removed from the list. */
  public E remove(int index);

  /** Replace the element at the specified position in this list
   *  with the specified element and returns the new set. */
  public Object set(int index, E e);

  /** Return the number of elements in this list */
  public int size();

  /** Return an iterator for the list */
  public java.util.Iterator<E> iterator();
}
```

METODE INTERFEJSA MYLIST I APSTRAKTNE KLASE MYABSTRACTLIST

Zajedničke operacija nad listama u Javi su definisane u interfejsu List. Ove operacija su zatim delimično implementirane u klasi AbstractList, a ostale u klasama ArrayList i LinkedList

Metode interfejsa *MyList* i metode koje su implementirane u okviru apstraktne klase *MyAbstractList* su prikazane na Slici-3:



Slika 1.3 Interfejs MyList definiše metode za manipulaciju nad listama. Klasa MyAbstractList delimično implementira MyList interfejs.

APSTRAKTNA KLASA MYABSTRACTLIST

Apstraktna klasa MyAbstractList deklariše promenljivu size koja predstavlja broj elemenata u listi, i implemetira metode isEmpty(), size(), add(E), i remove(E) interfejsa MyList

Apstraktna klasa *MyAbstractList* deklariše promenljivu *size* koja predstavlja broj elemenata u listi. Metodi *isEmpty()*, *size()*, *add(E)*, i *remove(E)* mogu biti implementirani u klasi, kao što je prikazano u sledećem kodu.

```
public abstract class MyAbstractList implements MyList {
  protected int size;
```

```
  /** Create a default list */
  protected MyAbstractList() {
  }

  /** Create a list from an array of objects */
  protected MyAbstractList(Object[] objects) {
    for (int i = 0; i < objects.length; i++)
    this.add(objects[i]);
  }

  /** Add a new element o at the end of this list */
  public void add(Object o) {
    add(size, o);
  }

  /** Return true if this list contains no elements */
  public boolean isEmpty() {
    return size == 0;
  }

  /** Return the number of elements in this list */
  public int size() {
    return size;
  }

  /** Remove the first occurrence of the element o from this list.
   *  Shift any subsequent elements to the left.
   *  Return true if the element is removed. */
  public boolean remove(Object o) {
    if (indexOf(o) >= 0) {
    remove(indexOf(o));
    return true;
    }
    else
    return false;
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Ako je list objekat klase MyList, da li se može dobiti iterator naredbom list.iterator()?

Zadatak 2. Da li se lista može kreirati naredbom new MyAbstractList() ?

Zadatak 3. Koje metode iz MyList su overrajdovane u MyAbstractList?

Zadatak 4. Koje su prednosti definisanja i MyList interfejsa kao i MyAbstractList apstraktne klase istovremeno?

Poglavlje 2

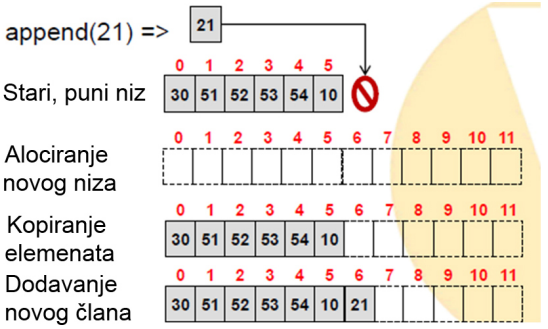
Lista bazirana na nizu

PRAVILNO KORIŠĆENJE NIZOVA

Jednom kad je niz kreiran njegova dužina ne može biti promenjena. U slučaju da je niz popunjen često se vrši kreiranje novog niza duplo veće dužine i prebacivanje podataka u njega

Niz (array) je struktura podataka fiksne dužine. Jednom kad je niz kreiran njegova dužina ne može biti promenjena. Ipak, nizovi mogu biti korišćeni za implementaciju dinamičke strukture podataka. Trik je u tome da se kreira veći niz koji će da zameni tekući niz, ako tekući niz nije u stanju da primi novi element koji želimo da ubacimo u njega.

Inicijalno, neki niz, na primer *data* koji je tipa ***E[]***, je kreiran sa nekom podrazumevanom (default) veličinom. Kada se ubacuje novi element u njega, prvo treba proveriti da li ima mesta u nizu za prihvatanje novog elementa. Ukoliko nema, kreirati novi niz koji je duplo veći od tekućeg. Kopirati elemente tekućeg niza u novi niz. Novi niz nakon toga postaje tekući (Slika-1).



Slika 2.1 Postupak proširenja niza kada nije u stanju da primi elemente koje želimo da dodamo u njega

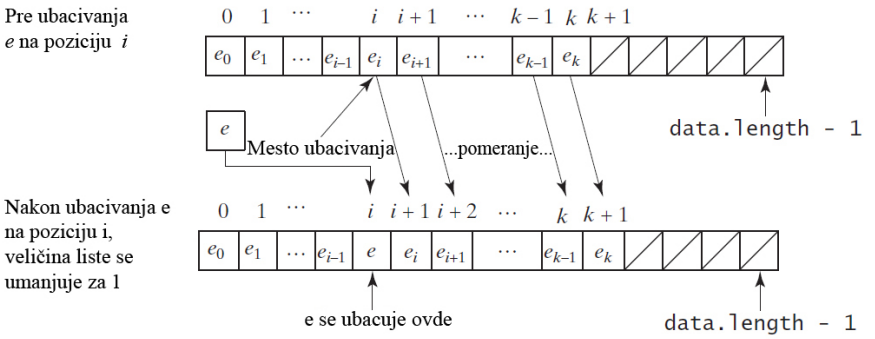
Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Niz može biti rešenje kada će biti mali broj brisanja i ubacivanja elemenata, a kada se očekuje dosta pretraživanja po nizu ili pristupanja elementima niza preko indeksa. Međutim, pristup korišćenjem nizova ima sledeće nedostatke:

UBACIVANJE I BRISANJE ELEMENATA

Pre ubacivanja novog elementa na neko mesto unutar niza, definisano indeksom, neophodno je sve elemente desno od indeksa pomeriti za jedno mesto u desno. Suprotno radimo kod brisanja.

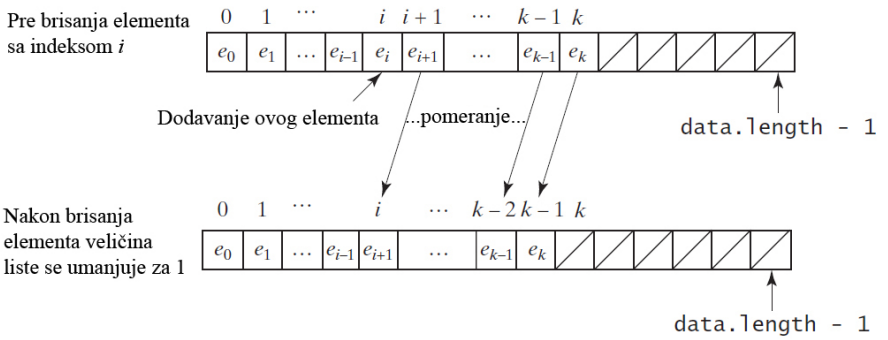
Šta se dešava ako želimo da dodamo novi element u unutrašnjost niza? Pre ubacivanja novog elementa na neko mesto unutar niza, definisano indeksom, neophodno je sve elemente desno od indeksa (zajedno sa elementom na tom indeksu) pomeriti za jedno mesto u desno, i uvećati tekući broj elemenata u nizu za jedan, kao što je prikazano na sledećoj Slici-2.



Slika 2.2 Ubacivanje novog elementa u niz zahteva da se svi elementi koji se nalaze nakon mesta ubacivanja pomere za jedno mesto u desno, tako da se oslobodi prostor za element koji se ubacuje u niz.

Napomena. Niz podataka je tipa ***E[]***. Svaka ćelija niza u Java-i ustvari sadrži referencu na objekat.

Da bi obrisali element sa određene pozicije iz niza, neophodno je pomeriti sve elemente nakon indeksa za jedno mesto u levo, i umanjiti veličinu liste za 1, kao što je prikazano na sledećoj Slici-3.

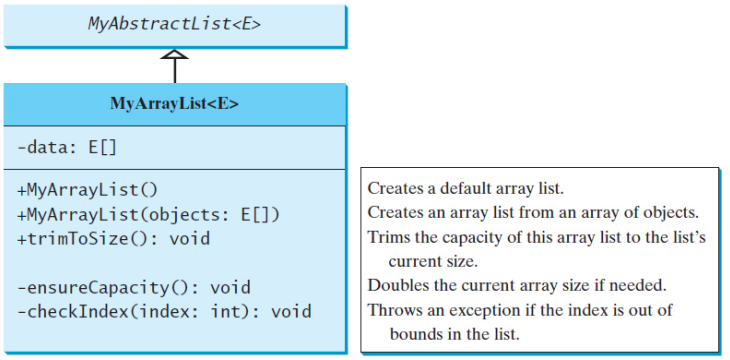


Slika 2.3 Brisanje elementa iz niza zahteva da se svi elementi nakon indeksa elementa za brisanje pomere za jednu poziciju u levo.

IMPLEMENTACIJA METODA KLASE MYARRAYLIST

Klasa MyArrayList koristi niz da bi proširila i implementirala apstraktnu klasu MyAbstractList

Kao što smo već napomenuli, koristimo naziv klase *MyArrayList* umesto*ArrayList* da ne bi došlo do konflikta sa Java Collections okvirom. Klasa*MyArrayList* koristi niz da bi proširila i implementirala apstraktnu klasu *MyAbstractList*, kao što je prikazano na sledećoj Slici-4.



Slika 2.4 Klasa MyArrayList implementira listu korišćenjem niza.

Implementacija klase *MyArrayList* je navedena u listingu koji sledi:

```
public class MyArrayList<E> extends MyAbstractList<E> {
  public static final int INITIAL_CAPACITY = 16;
  private E[] data = (E[])new Object[INITIAL_CAPACITY];

  /** Create a default list */
  public MyArrayList() {
  }

  /** Create a list from an array of objects */
  public MyArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
    add(objects[i]); // Warning: don't use super(objects)!
  }

  @Override /** Add a new element at the specified index */
  public void add(int index, E e) {
    ensureCapacity();

    // Move the elements to the right after the specified index
    for (int i = size - 1; i >= index; i--)
    data[i + 1] = data[i];

    // Insert new element to data[index]
    data[index] = e;

    // Increase size by 1
    size++;
```

```
  }

  /** Create a new larger array, double the current size + 1 */
  private void ensureCapacity() {
    if (size >= data.length) {
    E[] newData = (E[])(new Object[size * 2 + 1]);
    System.arraycopy(data, 0, newData, 0, size);
    data = newData;
    }
  }

  @Override /** Clear the list */
  public void clear() {
    data = (E[])new Object[INITIAL_CAPACITY];
    size = 0;
  }

  @Override /** Return true if this list contains the element */
  public boolean contains(E e) {
    for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return true;

    return false;
  }

  @Override /** Return the element at the specified index */
  public E get(int index) {
    checkIndex(index);
    return data[index];
  }

  private void checkIndex(int index) {
    if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException
        ("Index: " + index + ", Size: " + size);
  }

  @Override /** Return the index of the first matching element
   *  in this list. Return -1 if no match. */
  public int indexOf(E e) {
    for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return i;

    return -1;
  }

  @Override /** Return the index of the last matching element
   *  in this list. Return -1 if no match. */
  public int lastIndexOf(E e) {
    for (int i = size - 1; i >= 0; i--)
    if (e.equals(data[i])) return i;

    return -1;
```

```
}

@Override /** Remove the element at the specified position
 *  in this list. Shift any subsequent elements to the left.
 *  Return the element that was removed from the list. */
public E remove(int index) {
  checkIndex(index);

  E e = data[index];

  // Shift data to the left
  for (int j = index; j < size - 1; j++)
  data[j] = data[j + 1];

  data[size - 1] = null; // This element is now null

  // Decrement size
  size--;

  return e;
}

@Override /** Replace the element at the specified position
 *  in this list with the specified element. */
public E set(int index, E e) {
  checkIndex(index);
  E old = data[index];
  data[index] = e;
  return old;
}

@Override
public String toString() {
  StringBuilder result = new StringBuilder("[");

  for (int i = 0; i < size; i++) {
  result.append(data[i]);
  if (i < size - 1) result.append(", ");
  }

  return result.toString() + "]";
}

/** Trims the capacity to current size */
public void trimToSize() {
  if (size != data.length) {
  E[] newData = (E[])(new Object[size]);
  System.arraycopy(data, 0, newData, 0, size);
  data = newData;
  } // If size == capacity, no need to trim
}

@Override /** Override iterator() defined in Iterable */
```

```
  public java.util.Iterator<E> iterator() {
    return new ArrayListIterator();
  }

  private class ArrayListIterator
    implements java.util.Iterator<E> {
    private int current = 0; // Current index

    @Override
    public boolean hasNext() {
    return (current < size);
    }

    @Override
    public E next() {
    return data[current++];
    }

    @Override
    public void remove() {
    MyArrayList.this.remove(current);
    }
  }
}
```

OPIS KLASE MYARRAYLIST

Navodimo opis osnovnih metoda klase MyArrayList

Konstanta INITIAL_CAPACITY (line 2) se koristi da se kreiraju inicijalne vrednosti niza (line 3). Zahvaljujuči korišćenju generičkih tipova nije moguće kreirati generički niz korišćenjem sintakse new e[INITIAL_CAPACITY]. Da bi zaobišli ovo ograničenje, kreiran je niz objekata tipa Object u liniji 3 a zatim je izvršeno kastovanje u tip E[].

Primetiti da je implementacija drugog konstruktora klase MyArrayList ista kao kod klase MyAbstractList. Šta mislite, da li je moguće u toj sitauciji zameniti linije 11–12 sa naredbom super(objects)?

Metod add(int index, E e) (linije 16–28) ubacuje element e na odgovarajuću poziciju u nizu. Ovaj metod prvo poziva ensureCapacity() (line 17), koji osigurava da postoji dovoljno prostora u nizu za dodavanje novog elementa. Zatim pomera sve elemente koji su iza polja index za jednu poziciju u desno pre ubacivanja novog elementa (lines 20–21). Nakon što je element dodat, size je uvećan za 1 (line 27). Primetimo da je promenljiva size definisana ka zaštićena u okviru klase MyAbstractList, pa je dostupna u klasi MyArrayList.

Metod ensureCapacity() (lines 31–37) provera da li je niz pun. Ako jeste, onda program kreira novi niz duplo veći u odnosu na trenutni + 1, kopira elemente trenutnog niza u novi niz korišćenjem metoda System.arraycopy, i novi niz setuje za trenutni.

Metod clear() (lines 40–43) kreira novi niz , pri čemu vrednost size tretira INITIAL_CAPACITY i resetuje promenljivu size na 0. Klasa će raditi ako se obriše linija 41. Međutim, u klasi će je javiti curenje memorije (memory leak) jer su elementi još uvek u nizu, iako nisu više potrebni. Kreiranjem novog niza i referenciranjem korišćenjem promenljive data, stari niz i elementi u njemu će postati otpaci (garbage) tako da će ih automatski počistiti JVM.

Metod contains(E e) (lines 46–51) proverava da li se element e već nalazi u nizu, poređenjem e sa svakim pojedinačnim elementom niza primenom metode equals.

Metod get(int index) (lines 54–57) proverava da li je indeks niza u okviru opsega i vraća data[index] ako je index opsegu niza (validan indeks niza) .

OPIS OSTALIH METODA KLASE MYARRAYLIST

Navodimo i opis ostalih metoda klase MyArrayList

Metod checkIndex(int index) (lines 59–63) proverava da li je index validan (u opsegu). Ako nije, metod izbacuje izuzetak tipa IndexOutOfBoundsException (line 61).

Metod indexOf(E e) (lines 67–72) poredi element e sa elementima niza, počevši od prvog. Ako pronadje poklapanje, vraća se indeks na kome se javilo poklapanje, u suprotnom vrača -1.

Metod lastIndexOf(E e) (lines 76–81) poredi element e sa elementima niza, počevši od poslednjeg. Ako pronadje poklapanje, vraća se indeks na kome se javilo poklapanje, u suprotnom vrača -1.

Metod remove(int index) (lines 86–101) pomera sve elemente nakon index za jednu poziciju u levo (lines 92–93) i umanjuje size za jedan 1 (line 98). Poslednji element nije više potreban i setuje se na null (line 95).

Metod set(int index, E e) (lines 105–110) jednostavno ubacuje e u data[index], tj menja vrednost na specifičnom indeksu sa elementom e..

Metod toString() (lines 113–122) predefiniše (override) metod toString klase Object i vraća string repšrezentaciju svih elemenata liste.

Metod trimToSize() kreira novi niz čija je veličina (capacity) ista kao i trenutna veličina (size) liste (line 127), kopira elemente tekućeg niza u novi niz korišćenjem metoda System.arraycopy (line 128), i proglašava novi niz za tekući (line 129). Primetite da, ako je size == capacity, nema potrebe za rezanjem (trim) veličine niza.
Metod iterator() definisan u java.lang.Iterable interfejsu je implementian tako da vraća instancu tipa java.util.Iterator (lines 134–136). Klasa
ArrayListIterator implementira konretne metode iteratora kao što su hasNext, next, i remove (lines 143–155). Metod koristi current da označi trenutnu poziciju elementa do kog je stigao u toku putovanja kroz elemente niza (line 140).

UPOTREBA IMPLEMENTIRANE LISTE

Cilj primera je da se pokaže rad sa stringovima kao elementima liste, i primena operacija brisanja i dodavanja elemenata liste koja je kreirana korišćenjem klase MyArrayList

Naredni listing daje primer kako je moguće kreirati listu korišćenjem prethodno implementrane klase *MyArrayList*.

```
public class TestList {
  public static void main(String[] args) {
    // Create a list
    MyList<String> list = new MyArrayList<String>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.add("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    // Remove elements from the list
    list.remove("Canada"); // Same as list.remove(0) in this case
    System.out.println("(7) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(8) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.println("(9) " + list);
  }
}
```

Primer izvršavanja programa je dat na Slici-5:

```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [America, Germany, Russia, France, Norway]
(8) [America, Germany, France, Norway]
(9) [America, Germany, France]
(10) AMERICA GERMANY FRANCE
```

Slika 2.5 Primer izvršavanja programa koji pokazuje upotrebu klase MyArrayList

U programu se koristi metod *add* za dodavanje stringova u listu i metoda *remove* da bi se obrisali stringovi iz liste. Pošto klasa *MyArrayList* implementira interfejs *Iterable*, putovanje kroz elemente se može vršiti korišćenjem for-each petlje.

ZADACI ZA SAMOSTALAN RAD

Na osnovu materijala uradite samostalno sledeće zadatke

Zad 1. Koja su ograničenja array tipa podataka?

Zad 2. MyArrayList je implementirana koristeći niz koji predstavlja strukturu fiksne veličine. Zašto se onda MyArrayList računa kao dinamička struktura podataka?

Zad 3. Koja je veličina niza u MyArrayList posle izvršavanja svake od sledećih naredbi?

```
MyArrayList<Double> list = new MyArrayList<>();
list.add(1.5);
list.trimToSize();
list.add(3.4);
list.add(7.4);
list.add(17.4);
```

Zad 4. Šta nije u redu sa linijama ispod iz implementacije MyArrayList

```
for (int i = 0; i < objects.length; i++)
   add(objects[i]);
```

ako se zamene sa super(objects); ili

```
data = objects;
size = objects.length;
```

ili ako se naredna linija u implementaciji

```
E[] newData = (E[])(new Object[size * 2 + 1]);
```

zameni sa

```
E[] newData = (E[])(new Object[size * 2]);
```

program postaje netačan. Da li možete pronaći grešku?

Zad 5. Ako obrišemo sledeći kod na 41 liniji u implementaciji MyArrayList, da li će doći do problema curenja memorije?

```
data = (E[])new Object[INITIAL_CAPACITY];
```

Zad 6. Metod get(index) poziva checkIndex(index) metod (linije 59-63 u implementaciji MyArrayList) da proveri da li je indeks u granicama veličine liste i da izbaci odgovarajući izuzetak u suprotnom. Ako je dodavanje elementa add(index, e) implementirano na sledeći način:

```
public void add(int index, E e) {
    checkIndex(index);
    // Same as lines 17-27 in MyArrayList
```

Šta će se desiti ako pokrenete sledeći kod?

```
MyArrayList<String> list = new MyArrayList<>();
list.add("New York");
```

Zad 7. Napisati program koji će napraviti animaciju pretrage, ubacivanja i brisanja elementa iz liste kao što je prikazano na **Apstrakcije liste u Javi-Slika-1a**. Dugme za pretragu treba da pokrene traženje unetog elementa u listi. Dugme za brisanje treba da obriše uneti element iz liste. Dugme za umetanje treba da ubaci element na kraj liste ili da ga umetne na odgovarajuću poziciju koja se zadaje kao ulaz u program.

Poglavlje 3

Povezane liste

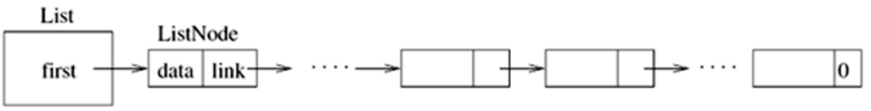
KREIRANJE ČVORA POVEZANE LISTE

U jednostruko povezanoj listi (singly linked list) čvor se sastoji od elemenata podataka (data element) i veze (pokazivača) na sledeći element u listi

Osnovni oblik povezane liste se sastoji od skupa povezanih, dinamički alociranih čvorova.

U jednostruko povezanoj listi (singly linked list) čvor se sastoji od elemenata podataka (data element) i veze (pokazivača) na sledeći element u listi. Pokazivač (pointer) je samo adresa sledećeg čvora u memoriji. Poslednji čvor u listi ima null kao *"next"* vezu.

Ključna razlika u odnosu na niz je u tome što povezna lista ne mora da bude sačuvana u kontinualnim lokacijama u memoriji, Slika-1.



Slika 3.1.1 Povezana lista

Implementacija čvora liste u programskom jeziku Java ima sledeći oblik:

```
class Node<E> {
  public E data;
 public Node<E> next;

 public Node (E data, Node<E> next) {
  this.data = data;
  this.next = next;
 }
 public Node(E e) {
  data = e;
 }
}
```

Korišćenje generičkih tipova (šablona) nam omogućuje da koristimo strukturu podatka čiji informacioni deo može da bude bilo kog tipa.

PRIMER KREIRANJA POVEZANE LISTE

Uobičajeni pokazivači koji se koriste kod jednostruko povezane liste su head i tail. Preko ovih pokazivača uvek imamo informaciju o početku i kraju liste

Analizirajmo sledeći primer:

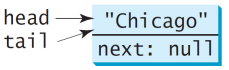
Pretpostavimo da u programu koristimo promenljive *head* i *tail* za kretanje kroz listu. *head* će da bude promenljiva koja će da se odnosi na početak liste dok će promenljiva *tail* da nam označava kraj liste. Prvi korak je deklaracija objektnih promenljivih *head* i *tail* koje u početnom trenutku nisu inicijalizovane tj, ne odgovaraju ni jednom objektu (lista je prazna):

```
Node<String> head = null;
Node<String> tail = null;
```

Sledeći korak je kreiranje novog objekta, odnosno kreiranje novog čvora i dodavanje u listu:

```
head = new Node<String>("Chicago");
tail = head;
```

Nakon izvršavanja prethodnog koda, formiraće se lista kao na Slici-2, pri čemu će i *head* i *tail* biti reference na isti objekat (*Chicago*).

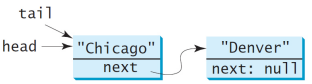


Slika 3.1.2 Dodavanje prvog čvora u listu

Sledeći korak je formiranje drugog čvora i njegovo dodavanje u listu:

```
tail.next = new Node<String>("Denver");
```

pri čemu će lista imati sledeći oblik (Slika-3):

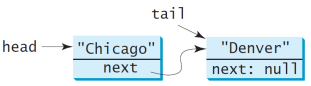


Slika 3.1.3 Dodavanje drugog čvora u listu

Nakon izvršavanja linije koda

```
tail = tail.next;
```

vrši se pomeranje reference *tail* sa prvog na drugi čvor tako da sada lista ima sledeći oblik:

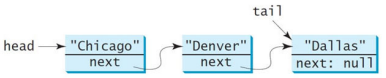


Slika 3.1.4 Lista nakon dodavanja drugog čvora i pomeranja pokazivača tail

Nakon sledeće dve linije koda:

```
tail.next = new Node<String>("Dallas");
tail = tail.next;
```

vrši se dodavanje trećeg čvora na kraj liste i pomeranje reference *tail* sa drugog na treći čvor tako da će konačan rezultat izvršavanja prethodnog koda biti prikazan na Slici-5.



Slika 3.1.5 Dodavanje trećeg čvora u listu

Kao što vidimo, svaki čvor sadrži polje *next* koje pokazuje na sledeći element liste. U slučaju da je neki čvor poslednji u listi, njegovo polje *next* će imati vrednost *null*. Kod za putovanje kroz listu može biti napisan na sledeći način:

```
Node current = head;
while (current ) {
 System.out.println(current.element);
 current = current.next;
}
```

KLASA MYLINKEDLIST

Klasa LinkedList koristi povezanu strukturu podataka da bi implementirala dinamički niz

U ovom delu ćemo opisati neke od operacija nad čvorovima liste kao što su dodavanje novog elementa liste, insertovanje elementa u unutrašnjost liste i brisanje elementa iz liste.

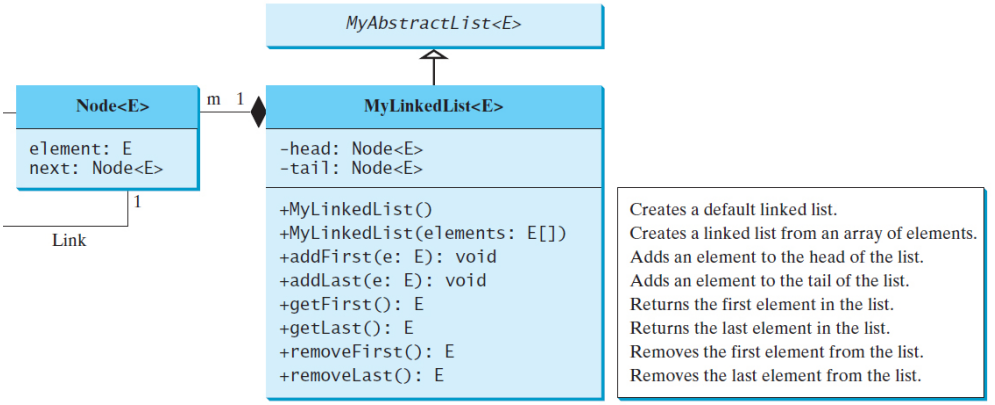
Neka je čvor liste definisan tako da se sastoji iz informacionog dela i pokazivača na sledeći čvor liste (*class Node<E>* iz ranije navedenog primera). Za početak ćemo definisatu novu klasu koju ćemo koristiti kao povezanu listu:

```
public class MyLinkedList<E>
{
 Node<E> head, tail;
 static int size;
}
```

Prethodna klasa se sastoji iz dve promenljive *head* i *tail* pomoću kojih ćemo da tačno znamo početak i kraj liste, kao i promenljive size koja će da pamti trenutan broj članova liste.

U nastavku će biti opisan postupak dodavanja i brisanja elemenata iz povezane liste, a kao primer ćemo koristiti klasu *MyLinkedList*.

Klasa *LinkedList* koristi povezanu strukturu da bi implementirala dinamički niz. U nastavku ćemo opisati postupak implementacije klase *LinkedList* korišćenjem klase *MyLinkedList* koju smo uveli ranije. Kao što smo videli u prethodnim sekcijama, ova klasa proširuje apstraktnu klasu *MyAbstractList*. Kao dodatak, ona implementira metode *addFirst*, *addLast*, *removeFirst*, *removeLast*, *getFirst*, i *getLast*, kao što je prikazano na sledećoj slici (Slika 1):



Slika 3.1.6 MyLinkedList implementira listu korišćenjem povezane liste čvorova

UPOTREBA IMPLEMENTIRANE POVEZANE LISTE

Cilj primera je da se pokaže upotreba operacija dodavanja, brisanja i pretraživanja elemenata u implementiranoj povezanoj listi koja služi kao lista tekstualnih podataka

U nastavku je primer koji testira upotrebu klase *MyLinkedList*.

```
public class TestLinkedList {
  /** Main method */
  public static void main(String[] args) {
    // Create a list for strings
    MyLinkedList<String> list = new MyLinkedList<String>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.addLast("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);
```

```
    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    list.add(0, "Poland"); // Same as list.addFirst("Poland")
    System.out.println("(7) " + list);

    // Remove elements from the list
    list.remove(0); // Same as list.remove("Australia") in this case
    System.out.println("(8) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(9) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.println("(10) " + list);
  }
}
```

Rezultat programa biće kao na sledećoj slici:

```
(1) [America]
(2) [Canada, America]
(3) [Canada, America, Russia]
(4) [Canada, America, Russia, France]
(5) [Canada, America, Germany, Russia, France]
(6) [Canada, America, Germany, Russia, France, Norway]
(7) [Poland, Canada, America, Germany, Russia, France, Norway]
(8) [Canada, America, Germany, Russia, France, Norway]
(9) [Canada, America, Russia, France, Norway]
(10) [Canada, America, Russia, France]
(11) CANADA AMERICA RUSSIA FRANCE
```

Slika 3.1.7 Rezultat izvršavanja prethodnog programa

U slučaju da niste uspeli da kreirate vašu klasu *MyLinkedList* za povezanu listu, uvek je moguće koristiti standardnu *JCF* klasu *LinkedList* za testiranje prethodnog koda.

Pogledajmo sada kako se vrši implementacija metoda klase *MyLinkedList*.

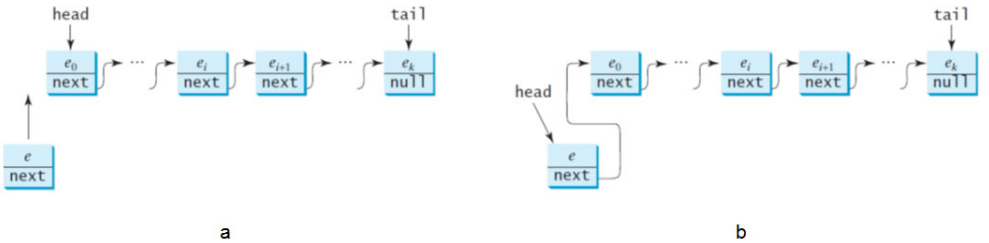
3.1 Implementacija klase LinkedList

DODAVANJE ČVORA NA POČETAK POVEZANE LISTE

Postoje dva uobičajena načina dodavanja novih elemenata u povezanu listu, a to su: dodavanja podataka na početak, odnosno dodavanje elemenata na kraj povezane liste

Metod addFirst(e) kreira novi čvor koji će da čuva vrednost elementa e, Novi čvor postaje prvi čvor u listi. Metod može biti implementiran na sledeći način:

```
public void addFirst(E e)
{
 Node<E> newNode = new Node<E>(e); // kreiranje novog cvora
 newNode.next = head;    // povezivanje novog cvora sa promenljivom head
 head = newNode; // head pokazuje na novi cvor
 size++;   // uvecati broj cvorova liste

 if (tail == null) // novi cvor je jedini cvor u listi
  tail = head;
}
```



Slika 3.2.1 Dodavanje čvora na početak povezane liste. Prvi korak je formiranje novog čvora (a), a drugi korak je postavljanje čvora na početak i pomeranje linka head.

Metod addFirst(e) kreira novi čvor koji će da prihvati element e (line 2) i ubacuje ga na početak liste (line 3), kao što je prikazano na Slici-1a.

Nakon ubacivanja, head pokazuje na ovaj novi čvor (line 4), kao što je prikazano na Slici-1b.

Ukoliko je lista prazna (line 7), i head i tail će pokazivati na novi čvoe (line 8). Nakon kreiranja novog čvora, promenljiva size se uvećava za 1 (line 5).

DODAVANJE ČVORA NA KRAJ POVEZANE LISTE

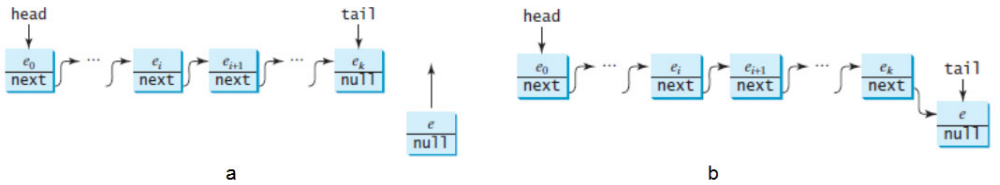
Metod addLast(e) kreira novi čvor u kome će se ubaciti element e, i dodaje ga na kraj liste

Metod addLast(e) kreira novi čvor u kome će se ubaciti element e, i dodaje ga na kraj liste. Može biti implementiran na sledeći način :

```
public void addLast(E e) {
 Node<E> newNode = new Node<E>(e); // Kreiranje novog elementa

 if (tail == null)  {
  head = tail = newNode; // Novi cvor je jedini u listi
 }
 else  {
  tail.next = newNode; // Povezati novi cvor sa poslednjim cvorom
  tail = tail.next; // tail se pomera da pokazuje na poslednji cvor
 }
```

```
 size++; // Uvecati broj cvorova liste
}
```

Na Slici-2 je prikazan postupak dodavanja novog čvora na kraj liste.



Slika 3.2.2 Dodavanje čvora na kraj liste. a) formiranje novog čvora. b) dodavanje čvora na kraj i pomeranje linka tail.

Metod addLast(e) kreira novi čvor koji će prihvatitti element e (line 2) i dodaje ga na kraj liste. Razmotrimo dva slučaja:

1. Ako je lista prazna (line 4), i head i tail će pokazivati na novi čvor (line 5);

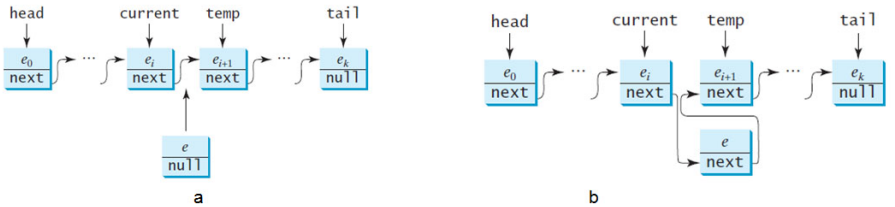
2. U suprotnom, povezati novi čvos ra poslednjim u listi (line 8). tail sada pokazuje na ovaj novi čvor (line 9). Slike-2a i 2b pokazuju novi čvor za element e, pre i posle dodavanja u listu .

U bilo kom od prethodna dva slučaja, veličina niza (size) treba da se uveća za 1 (line 12).

DODAVANJE ELEMENATA U LISTU KORIŠĆENJEM INDEKSA

Za dodavanje novog čvora u listu neophodno je uvesti dva pomoćna pokazivača current i temp preko kojih ćemo putovati kroz listu i u određenom trenutku između njih dodati novi čvor

U ovom delu je opisan postupak ubacivanja novog elementa u listu korišćenjem klase definisane u programskom jeziku Java, gde je cilj da se insertovanje izvrši na mesto u listi koje odgovara indeksu *i*. Postupak je opisan na sledećoj slici (Slika-3):



Slika 3.2.3 Ubacivanje elementa u listu na mesto koje odgovara indeksu i. Prvi korak je kreiranje novog čvora (a), a drugi je ubacivanje čvora izmedju linkova current i temp.

Implementacija ovog metoda je data u nastavku:

```
public void add(int index, E e) {
 if (index == 0)  addFirst(e);
 else if (index >= size)  addLast(e);
```

```
 else  {
  Node<E> current = head;
  for (int i = 1; i < index; i++)
   current = current.next;
  Node<E> temp = current.next;
  current.next = new Node<E>(e);
  (current.next).next = temp;
  size++;
 }
}
```

Kod ubacivanja elementa u listu postoje tri slučaja:

1. Ako je indeks 0, onda se poziva addFirst(e) (line 2) da se ubaci element na početak liste.

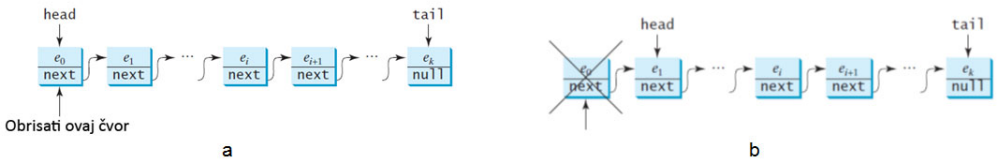
2. Ako je indeks veći ili jednak veličini liste, onda se poziva addLast(e) (line 3) da ubaci element na kraj liste.

3. Inače, treba napraviti novi čvor u listi koji će sadržati element koji se ubacuje i treba locirati njegovu poziciju. Kao što je prikazano na **Slika-3a,** novi čvor se ubacuje između čvorova **current** i **temp**. Metod dodeljuje novi čvor kao vrednost **current.next** i dodeljuje **temp** kao **next** vrednost novog čvora, kao što je prikazano na **Slika-3b**. Veličina se povećava za jedan (line 11).

BRISANJE PRVOG ČVORA IZ LISTE

Dve najuobičajene operacije kod brisanja čvorova iz povezane liste su brisanje čvora sa početka, odnosno brisanje čvora sa kraja povezane liste

Postupak brisanja čvora sa početka liste je prikazan na Slici-5:



Slika 3.2.4 Brisanje prvog čvora iz povezane liste

Kod za brisanje može biti implementiran na sledeći način:

```
public E removeFirst() {
 if (size == 0)  return null;
 else {
  Node<E> temp = head;
  head = head.next;
  size--;
  if (head == null) tail = null;
  return temp.data;
```

```
 }
}
```

Postoje dva slučaja:

1. Ako je lista prazna, nema šta da se briše, taji da se vraća vrednost **null** (line 2).

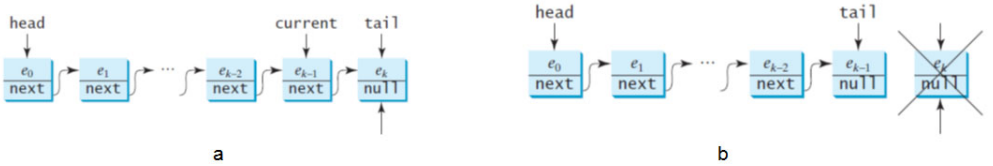
2. Inače, obriši prvi element iz liste preusmeravanjem **head** vrednosti na drugi čvor liste.

Slika-4a i **Slika-4b** prikazuju listu pre i posle brisanja. Veličina liste je smanjena za 1 (line 6). Ako lista postane prazna posle brisanja, **tail** vrednost treba postaviti na **null** (line 7).

BRISANJE POSLEDNJEG ČVORA IZ LISTE

Postoje tri slučaja: kada je lista prazna, kada ima jedan čvor i kada ima više čvorova (opšti slučaj)

Postupak brisanja poslednje čvora iz liste je prikazan na Slici-6:



Slika 3.2.5 Postupak brisanja poslednjeg elementa iz liste

Kod za brisanje može biti implementiran na sledeći način:

```
public E removeLast() {
 if (size == 0)  return null;
 else if (size == 1) {
  Node<E> temp = head;
  head = tail = null;
  size = 0;
  return temp.data;
 }
 else {
  Node<E> current = head;

  for (int i = 0; i < size - 2; i++)
   current = current.next;

  Node<E> temp = tail;
  tail = current;
  tail.next = null;
  size--;
  return temp.data;
 }
}
```

Postoje tri slučaja:

1. Ako je lista prazna, vraća se *null* (line 2).

2. Ako lista ima samo jedan čvor; *head* i *tail* postaju *null* (line 5). Posle brisanja veličina liste je jednaka nuli (line 6) i vraća se vrednost koju je imao obrisani čvor (line 7).

3. Inače, poslednji čvor se uništava (line 17) i *tail* se menja tako da pokazuje na pretposlednji element liste. **Slika-5a** i **Slika-5b** prikazuje listu pre i posle brisanja poslednjeg čvora.

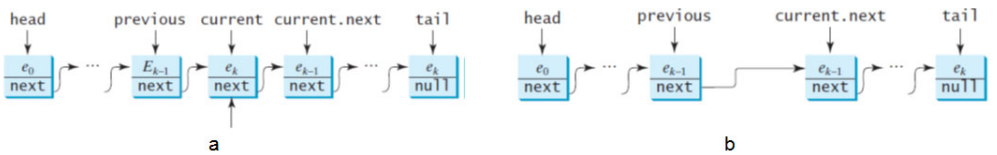
Veličina je smanjena za jedan posle brisanja (line 18) i vrednost koju je imao obrisani čvor se vraća kao rezultat (line 19).

BRISANJE ELEMENTA IZ LISTE PREKO INDEKSA

Metoda koja briše proizvoljan čvor liste preko indeksa treba da obradi sledeće slučajeve: da li je indeks validan, da li se indeks odnosi na prvi, poslednji, ili neki unutrašnji čvor

U nastavku je prikazan metod koji pronalazi čvor na određenom mestu u listi ,a zatim ga i briše iz liste:

```
public E remove(int index) {
 if (index < 0 || index >= size) return null;
 else if (index == 0)  return removeFirst();
 else if (index == size - 1)  return removeLast();
 else {
  Node<E> previous = head;

  for (int i = 1; i < index; i++){
   previous = previous.next;
  }

  Node<E> current = previous.next;
  previous.next = current.next;
  size--;
  return current.data;
 }
}
```



Slika 3.2.6 Brisanje elementa iz liste preko indeksa. Prvo se locira element sa indeksom k korišćenjem linka current (a), a zatim se vrši premošćavanje linka previous.next da ne pokazuje na current, već na current.next (b)

Razmotrimo sledeća četiri slučaja:

1. Ako je *index* izvan opsega liste (tj. *index < 0 || index >= size*), vratiti *null* (linija 2).

2. Ukoliko je *index* jednako *0*, pozvati *removeFirst()* da se obriše prvi čvor (linija 3).

3. Ako je *index* jednak *size - 1*, pozvati *removeLast()* da se obriše poslednji čvor (linija 4).

4. Ukoliko nije ništa od prethodnog, locirati čvor na indeksu *index*. Neka *current* označava taj čvor, a *previous* prethodni čvor, kao što je prikazano na Slici-6a. Dodeliti polju *current.next* vrednost *previous.next* da bi se eliminisao trenutni čvor, kao što je prikazano na Slici-6b.

KOMPLETNA IMPLEMENTACIJA KLASE MYLINKEDLIST

Klasa LinkedList proširuje klasu AbstractList i implementira sve ostale metode interfejsa List koji nisu implementirani u apstraktnoj klasi AbstractList

Kompletna implementacija klase *MyLinkedList* je prikazana u listingu koji sledi. Implementacije *get(index),*
indexOf(e), lastIndexOf(e), contains(e), i *set(index, e)* su izostavljene kako bi ostale za vežbu. Metod **iterator()** definisan u *java.lang.Iterable* interfejsu je implementiran da vrati objekat tipa *java.util.Iterator* (lines 126–128). Klasa *LinkedListIterator* implementira *Iterator* interfejs sa konkretnim metodama za **hasNext, next** i **remove** (lines 134–149). Ove implementacije koriste vrednost **current** da odrede trenutni element (line 132). Na početku, **current** pokazuje na **head** liste.

```
public class MyLinkedList<E> extends MyAbstractList<E> {
  private Node<E> head, tail;

  /** Create a default list */
  public MyLinkedList() {
  }

  /** Create a list from an array of objects */
  public MyLinkedList(E[] objects) {
    super(objects);
  }

  /** Return the head element in the list */
  public E getFirst() {
    if (size == 0) {
    return null;
    }
    else {
    return head.element;
    }
  }

  /** Return the last element in the list */
  public E getLast() {
    if (size == 0) {
    return null;
    }
```

```
  else {
  return tail.element;
  }
}

/** Add an element to the beginning of the list */
public void addFirst(E e) {
  Node<E> newNode = new Node<E>(e); // Create a new node
  newNode.next = head; // link the new node with the head
  head = newNode; // head points to the new node
  size++; // Increase list size

  if (tail == null) // the new node is the only node in list
  tail = head;
}

/** Add an element to the end of the list */
public void addLast(E e) {
  Node<E> newNode = new Node<E>(e); // Create a new for element e

  if (tail == null) {
  head = tail = newNode; // The new node is the only node in list
  }
  else {
  tail.next = newNode; // Link the new with the last node
  tail = tail.next; // tail now points to the last node
  }

  size++; // Increase size
}


@Override /** Add a new element at the specified index
 * in this list. The index of the head element is 0 */
public void add(int index, E e) {
  if (index == 0) {
  addFirst(e);
  }
  else if (index >= size) {
  addLast(e);
  }
  else {
  Node<E> current = head;
  for (int i = 1; i < index; i++) {
      current = current.next;
  }
  Node<E> temp = current.next;
  current.next = new Node<E>(e);
  (current.next).next = temp;
  size++;
  }
}
```

```
/** Remove the head node and
 *  return the object that is contained in the removed node. */
public E removeFirst() {
  if (size == 0) {
  return null;
  }
  else {
  Node<E> temp = head;
  head = head.next;
  size--;
  if (head == null) {
      tail = null;
  }
  return temp.element;
  }
}

/** Remove the last node and
 * return the object that is contained in the removed node. */
public E removeLast() {
  if (size == 0) {
  return null;
  }
  else if (size == 1) {
  Node<E> temp = head;
  head = tail = null;
  size = 0;
  return temp.element;
  }
  else {
  Node<E> current = head;

  for (int i = 0; i < size - 2; i++) {
      current = current.next;
  }

  Node<E> temp = tail;
  tail = current;
  tail.next = null;
  size--;
  return temp.element;
  }
}

@Override /** Remove the element at the specified position in this
 *  list. Return the element that was removed from the list. */
public E remove(int index) {
  if (index < 0 || index >= size) {
  return null;
  }
  else if (index == 0) {
  return removeFirst();
  }
```

```
  else if (index == size - 1) {
  return removeLast();
  }
  else {
  Node<E> previous = head;

  for (int i = 1; i < index; i++) {
      previous = previous.next;
  }

  Node<E> current = previous.next;
  previous.next = current.next;
  size--;
  return current.element;
  }
}

@Override /** Override toString() to return elements in the list */
public String toString() {
  StringBuilder result = new StringBuilder("[");

  Node<E> current = head;
  for (int i = 0; i < size; i++) {
  result.append(current.element);
  current = current.next;
  if (current != null) {
      result.append(", "); // Separate two elements with a comma
  }
  else {
      result.append("]"); // Insert the closing ] in the string
  }
  }

  return result.toString();
}

@Override /** Clear the list */
public void clear() {
  size = 0;
  head = tail = null;
}

@Override /** Return true if this list contains the element e */
public boolean contains(E e) {
  System.out.println("Implementation left as an exercise");
  return true;
}

@Override /** Return the element at the specified index */
public E get(int index) {
  System.out.println("Implementation left as an exercise");
  return null;
}
```

```
@Override /** Return the index of the head matching element in
 *  this list. Return -1 if no match. */
public int indexOf(E e) {
  System.out.println("Implementation left as an exercise");
  return 0;
}

@Override /** Return the index of the last matching element in
 *  this list. Return -1 if no match. */
public int lastIndexOf(E e) {
  System.out.println("Implementation left as an exercise");
  return 0;
}

@Override /** Replace the element at the specified position
 *  in this list with the specified element. */
public E set(int index, E e) {
  System.out.println("Implementation left as an exercise");
  return null;
}

@Override /** Override iterator() defined in Iterable */
public java.util.Iterator<E> iterator() {
  return new LinkedListIterator();
}

private void checkIndex(int index) {
  if (index < 0 || index >= size)
  throw new IndexOutOfBoundsException
      ("Index: " + index + ", Size: " + size);
}

private class LinkedListIterator
  implements java.util.Iterator<E> {
  private Node<E> current = head; // Current index

  @Override
  public boolean hasNext() {
  return (current != null);
  }

  @Override
  public E next() {
  E e = current.element;
  current = current.next;
  return e;
  }

  @Override
  public void remove() {
  System.out.println("Implementation left as an exercise");
  }
```

```
  }

  private static class Node<E> {
    E element;
    Node<E> next;

    public Node(E element) {
    this.element = element;
    }
  }
}
```

POREĐENJE KLASA MYARRAYLIST I MYLINKEDLIST

Dobro poznavanje složenosti operacija nad listama i nizovima predstavlja neophodnu osnovu za pravilan izbor strukture podataka za postavljeni tip problema

Obe klase *MyArrayList* i *MyLinkedList* mogu biti korišćenje za čuvanje liste podataka. Klasa *MyArrayList* je implementirana korišćenjem niza, dok je klasa *MyLinkedList* implementirana korišćenjem povezane liste. Na narednoj slici su ilustrovana poređenja vremenske složenosti za svaki od metoda klasa *MyArrayList* i *MyLinkedList*.

Naravno, primetimo da je klasa *MyArrayList* koju smo kreirali ista kao *java.util.ArrayList*, odnosno da je klasa *MyLinkedList* na isti način kreirana kao i klasa *java.util.LinkedList*.

Methods	*MyArrayList/ArrayList*	*MyLinkedList/LinkedList*
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

Slika 3.2.7 Vremenska složenost za metode klasa MyArrayList i MyLinkedList

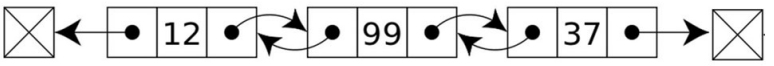
VARIJACIJE POVEZANE LISTE

Kod jednostruko povezane liste, čvor sadrži link samo na naredni čvor. Kod dvostruko povezane liste čvor sadrži link na naredni i na prethodni čvor. Kod kružne liste poslednji pokazuje na prvi

Dvostruko povezana lista

Dvostruko povezane liste su one liste kod kojih svaki element sadrži pokazivač (link) na svog prethodnika i sledbenika (Slika-4). Uvedene su kod jednostruko povezanih listi sa ciljem:

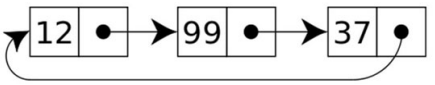
- Da se obezbedi kretanje kroz listu u oba smera
- Da se prilikom brisanja elemenata ne čuva dodatna promenljiva koja označava prethodni element liste
- Da se ne izgubi veza u nekom elementu što dovodi do toga da ostali elementi postanu nedostupni.



Slika 3.2.8 Dvostruko povezana lista koja se sastoji iz tri čvora

Kružna ili cirkularna lista

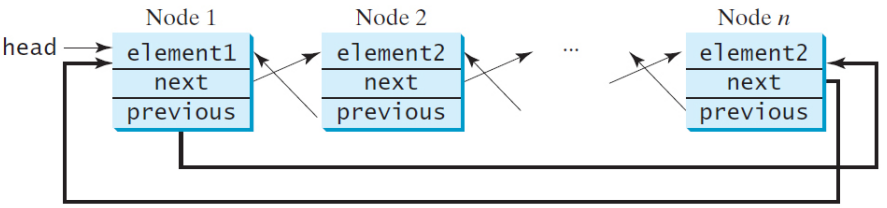
Kružne ili cirkularne liste su liste kod kojih poslednji element (čvor) pokazuje na prvi element. Stoga se kružne liste mogu posmatrati kao liste koje nemaju ni početak ni kraj (Slika-5).



Slika 3.2.9 Kružna lista. Pokazivač sa poslednjeg elementa u listi pokazuje na prvi element liste

Jedan od primera kružne liste je deljenje procesorskog vremena različitim korisnicima od strane operativnog sistema. Korisnici su rapoređeni u krug i redom im se dodeljuje deo slobodnog vremena procesora. Korisnici koji nemaju više zahteva se isključuju iz liste.

Kružna lista može biti implementirana korišćenjem dvostruko povezane liste s tim što će desni pokazivač poslednjeg čvora dvostruko povezane liste pokazivati na prvi čvor, a levi pokazivač prvog čvora liste pokazivati na poslednji čvor. To je šematski prikazano na Slici-9.



PITANJA ZA SAMOPROVERU

Pokušajte sami:

Pitanje 1. Obe klase *MyArrayList* i *MyLinkedList* se koriste za čuvanje liste objekata. Zašto su nam potrebne obe vrste listi?

Pitanje 2. Nacrtati dijagram koji prikazuje izgled povezane liste posle svake od sledećih naredbi.

```
MyLinkedList<Double> list = new MyLinkedList<>();
list.add(1.5);
list.add(6.2);
list.add(3.4);
list.add(7.4);
list.remove(1.5);
list.remove(2);
```

Pitanje 3. Koja je vremenska složenost **addFirst(e)** i **removeFirst()** metoda iz klase *MyLinkedList*?

Pitanje 4. Neka treba sačuvati listu elemenata. Ako je broj elemenata u programu fiksiran, koju strukturu podataka treba koristiti? Ako se broj elemenata menja tokom izvršavanja programa, koju strukturu treba onda koristiti?

Pitanje 5. Ako treba često dodavati ili brisati elemente na početku liste, da li je bolje koristiti onda *MyArrayList* ili *MyLinkedList*? Ako se najviše koristi uzimanje elementa iz liste na osnovu indeksa, da li je bolje koristiti *MyArrayList* ili *MyLinkedList*?

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće zadatke

Zadatak 1. (Dodavanje novih operacija u interfejs *MyList* za rad sa skupovima) Definisati sledeće metode u interfejsu *MyList* **i** implementirati ih u apstraktnoj klasi *MyAbstractList* (interfejs *MyList* i apstraktna klasa *MyAbstractList* su obrađeni na predavanjima):

```
/** Adds the elements in otherList to this list.
 * Returns true if this list changed as a result of the call */
public boolean addAll(MyList<E> otherList);
/** Removes all the elements in otherList from this list
 * Returns true if this list changed as a result of the call */
public boolean removeAll(MyList<E> otherList);
/** Retains the elements in this list that are also in otherList
 * Returns true if this list changed as a result of the call */
public boolean retainAll(MyList<E> otherList);
```

Napisati test program koji će da kreira dve liste tipa *MyArrayList*, i to *list1* i *list2*, sa inicijalnim vrednostima **{"Tom", "George", "Peter", "Jean", "Jane"}** i **{"Tom", "George", "Michael", "Michelle", "Daniel"}**, a zatim izvršiti sledeće operacije:

- Pozvati metod *list1.addAll(list2)*, a zatim prikazati liste *list1* i *list2*.
- Ponovo kreirati *list1* i *list2* sa istim inicijalnim podacima, pozvati metod *list1.removeAll(list2)*, pa opet prikazati liste *list1* i *list2*.
- Ponovo kreirati *list1* i *list2* sa istim inicijalnim podacima, pozvati metod *list1.retainAll(list2)*, pa opet prikazati liste *list1* i *list2*.

Zadatak 2. (Implementacija ostalih metoda klase*MyLinkedList*) Implementacija metoda *contains(E e)*, *get(int index)*, *indexOf(E e)*, *lastIndexOf(E e)*, i *set(int index, E e)*je preskočena na predavanjima. Stoga, neophodno je implementirati i te metode klase *MyLinkedList*.

Zadatak 3. Napisati program koji će napraviti animaciju pretrage, ubacivanja i brisanja elementa iz liste kao što je prikazano na **Apstrakcije liste u Javi-Slika-1b**. Dugme za pretragu treba da pokrene traženje unetog elementa u listi. Dugme za brisanje treba da obriše uneti element iz liste. Dugme za umetanje treba da ubaci element na kraj liste ili da ga umetne na odgovarajuću poziciju koja se zadaje kao ulaz u program..

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Napisati program koji formira jednostruko povezanu listu i ispisuje sadržaj u inverznom poretku. Unos se vrši dok se ne pređe u sledeći red.

Zadatak 2. Napisati glavni program kojim se formira lista tako da se svaki novi čvor postavlja na njen kraj.

Zadatak 3. Napisati rekurzivne funkcije za formiranje i ispis liste u inverznom poretku. Napisati i glavni program, a pritom koristiti definicije iz prethodnih zadataka.

Zadatak 4. Napisati program koji korišćenjem rekurzivne funkcije za ubacivanje novog člana u listu, formira uređenu rastuću listu. Napisati i rekurzivnu funkciju za brisanje elemenata iz liste.

Zadatak 5. Napisati program koji od n slova sa ulaza formira i ispisuje kružnu listu.

Zadatak 6. The *MyLinkedList* je jednostruko povezana lista koja omogućava prolaženje kroz listu u jednom pravcu (od početka do kraja). Modifikovati klasu *Node* tako da se doda i polje previous koje pokazuje na prethodni element liste na sledeći način:

```
public class Node<E> {
    E element;
    Node<E> next;
    Node<E> previous;
    public Node(E e) {
        element = e;
```

```
    }
}
```

Implementirati novu klasu *TwoWayLinkedList* koja koristi dvostruko povezanu listu za čuvanje elemenata. Klasa *MyLinkedList* nasleđuje *MyAbstractList*. Definisati *TwoWayLinkedList* tako da nasleđuje *java.util.AbstractSequentialList* klasu. Morate implementirati sve metode iz *MyLinkedList* ali još i *listIterator()* and*listIterator(int index).* Obe metode vraćaju *java.util.ListIterator<E*>. Prva postavlja trenutnu poziciju iteratora na početak liste, a druga na zadati indeks.

Poglavlje 4

Stekovi i redovi čekanja

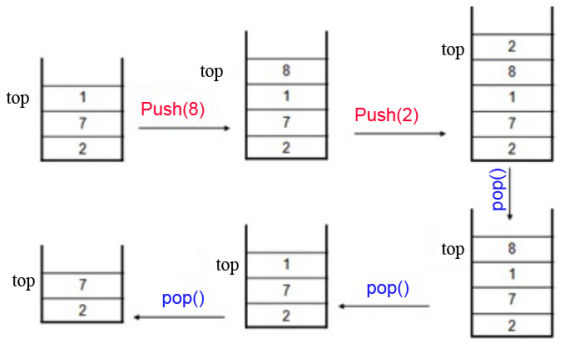
OSNOVI O STEKOVIMA

Stek je struktura podataka u kojoj se podaci dodaju i uklanjaju na samo jednom kraju koji se zove vrh steka. Stoga, za stek je dovoljno čuvati pokazivač na početak steka

Stek je struktura podataka u kojoj se podaci dodaju i uklanjaju na samo jednom kraju koji se zove vrh (engl. top) steka.

Da bi dodali (engl. push) element na stek, on mora biti stavljen na vrh steka. Da bi uklonili (engl. pop) element sa steka, on mora biti uklonjen sa vrha steka (Slika-1). Stoga, za stek je dovoljno čuvati pokazivač na početak steka. Dakle, poslednji element koji je stavljen na stek je prvi element koji će biti uklonjen sa steka, odnosno, stek je Last In First Out- *LIFO* struktura podataka.

Primeri steka su: gomila knjiga, pez dispenser, poslužavnici za kafu u restoranu. U računarstvu, stek se koristi pri izvršavanju programa, posebno za parsiranje/evaluaciju izraza.



Slika 4.1 Stek i operacije na steku

Stek je apstraktna struktura podataka koja podržava sledeće metode:

- *push(o)*: staviti objekat na vrh steka (engl. top of stack)
- *pop()*: ukloniti objekat sa vrha steka i vratiti ga kao rezultat; ako je stek bio prazan generisati izuzetak.

Treba takođe definisati i sledeće metode:

- *size()*: vraća broj objekata na steku.
- *isEmpty()*: vraća logičku vrednost (***true*** ili ***false***) koja pokazuje da li je stek prazan.

- *top()*: vraća objekat sa vrha steka, ali ga ne uklanja; ako je stek prazan ne generiše se izuzetak.

Uobičajeni interfejs za stek u Javi ima sledeći oblik:

```
interface Stack<E> {
 public boolean isEmpty();
 public E peek();
 public E pop();
 public void push(E element);
 public int size();
}
```

IMPLEMENTACIJA STEKA POMOĆU NIZA

Klasa za stek se obično kreira korišćenjem kompozicije, gde je član klase niz koji će da čuva elemente steka. Osim običnog niza često se koristi i niz tipa java.util.ArrayList

U nastavku je dat primer za implementaciju steka pomoću niza:

```
interface Stack<E> {
 public boolean isEmpty();
 public E peek();
 public E pop();
 public void push(E element);
 public int size();
}
public class ArrayStack<E> implements Stack<E> {
 private E[] elements;
 private int size;
 private static final int INITIAL_CAPACITY = 100;
 public ArrayStack() {

  elements = (E[]) new Object[INITIAL_CAPACITY];
 }
 public ArrayStack(int capacity) {
  elements = (E[]) new Object[capacity];
 }
 public boolean isEmpty() {
  return (size == 0);
 }
 public E peek() {
  if (size == 0) {
   throw new java.util.EmptyStackException();
  }
  return elements[size-1]; // top of stack
 }
 public E pop() {
  if (size == 0) {
```

```
   throw new java.util.EmptyStackException();
  }
  E element = elements[--size];
  elements[size] = null;
  return element;
 }
 public void push(E element) {
  if (size == elements.length) {
   resize();
  }
  elements[size++] = element;
 }
 public int size() {
  return size;
 }
 private void resize() {
  assert size == elements.length;
  Object[] a = new Object[2*size];
  System.arraycopy(elements, 0, a, 0, size);
  elements = (E[])a;
 }
 public static void main (String args []){
  int i=10;
  ArrayStack<Integer> ars =new ArrayStack <Integer> (100);
  System.out.println ("start " + ars.size);
  ars.push (10);
  System.out.println ("element #1 size= " + ars.size);
  ars.push (20);
  ars.push (30);
  ars.push (40);
  ars.push (50);
  ars.pop ();
  System.out.println ("peek " + ars.peek());
  ars.resize();
  System.out.println ("after resize " + ars.size);
 }
}
```

Rezultat programa je:

```
start 0
element #1 size= 1
peek 40
after resize 4
```

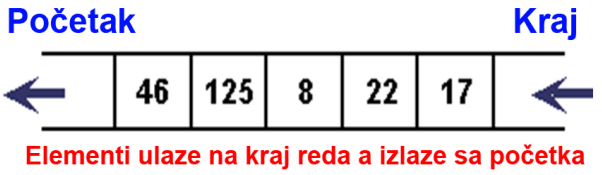
U prethodnom primeru smo kreirali generički interfejs *Stack*, a zatim smo kreirali klasu za stek *ArrayStack*, koja implementira sve metode interfejsa *Stack*. Koristimo promenljivu *size* za broj elemenata steka, a i da bi pristupili elementu na vrhu steka. Metode *pop*, *peek* i *top* izbacuju odgovarajuću vrstu izuzetka ukoliko se pokuša obavljanje nedozvoljenih operacija, pomenutih u prethodnom delu lekcije.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

RED ČEKANJA

Red čekanja je struktura podatka kod koje se brisanje elemenata vrši na jednom, a dodavanje elemenata na drugom kraju. Stoga, red radi po FIFO (first-in-first-out) principu

Red čekanja je uskladišteni skup elemenata u određenom redosledu. Radi po FIFO (First-In-First-Out) principu. Neki primeri iz realnog života su čekanje u redu u kafeteriji, čekanja na tehničku podršku i u oblasti računara, baferovani I / O.



Slika 4.2 Red čekanja

Redovi su kao i stek, poseban tip linearne liste. Red čekanja može biti implementiran na dva načina:

- korišćenjem niza
- korišćenjem povezane liste

Za razliku od steka, gde se čuva samo pokazivač na početak steka, kod reda se čuvaju pokazivači za početak i kraj liste. Jedan kraj liste se zove glava ili početak liste i sa njega se može uzeti prvi naredni element. Dodavanje elemenata u red se obavlja na kraju ili repu liste.

Složenost dodavanja ili uklanjanja elementa iz reda je konstantna, O(1).

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Interfejs za red čekanja u Javi može biti napisan na sledeći način:

```
public interface Queue<E> {
 public int size();
 public boolean isEmpty();
 public E front() throws QueueEmptyException;
 public void enqueue (E element);
 public E dequeue() throws QueueEmptyException;
}
```

RED ČEKANJA KAO POVEZANA LISTA

Kod implementacije reda korišćenjem povezane liste, potrebne su dve reference, front i rear, za praćenje stanja na oba kraja reda. Zaglavlje liste je na početku, dok je rep liste na kraju reda

Kod implementacije reda korišćenjem povezane liste, potrebne su dve reference (ili dva pokazivača), *front* i *rear*, za praćenje stanja na oba kraja reda (Slika-1). U tom slučaju se red može implementirana na neki od sledeća dva načina:

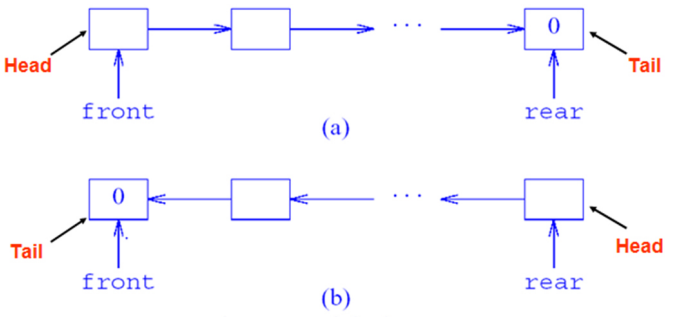
- koristiti *head* kao *front* i *tail* kao *rear* ili
- koristiti *head* kao *rear* i *tail* kao *front*

Kod reda čekanja, ubacivanja se dešavaju sa jedne strane, brisanja sa druge strane. Operacije na početku jednostruke povezane liste (engl. Singly-linked List- SLL) su operacije složenosti O(1), ali na kraju O(n), jer treba naći poslednji element.

Rešenje za postizanje O(1) u oba slučaja je sledeće:

- potreban pokazivač na prvi element liste
- dodati još jedan pokazivač za pokazivanje poslednjeg elementa liste

Zaglavlje liste je na početku reda, dok je rep liste na kraju reda. Zašto nije suprotno?



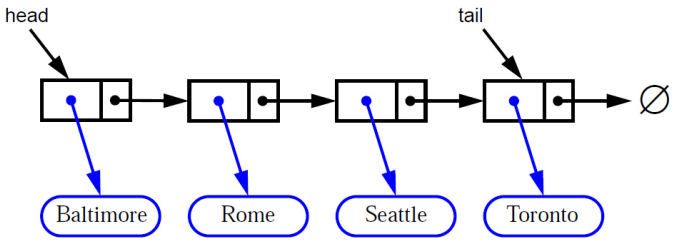
Slika 4.3 Head i tail pokazivači kod reda implementiranog pomoću povezane liste

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

BRISANJE I DODAVANJE ELEMENATA

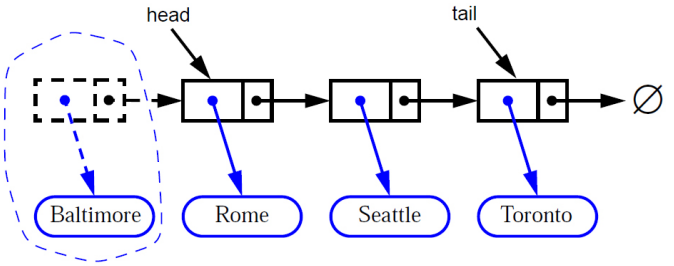
Brisanje zaglavlja se vrši prebacivanjem reference head na drugi član. Dodavanje elementa se sastoji iz: kreiranja čvora, postavljanja čvora na kraj liste i pomeranja reference tail

Pretpostavimo da početna lista ima sledeći izgled (Slika-2):



Slika 4.4 Primer red korišćenjem povezane liste

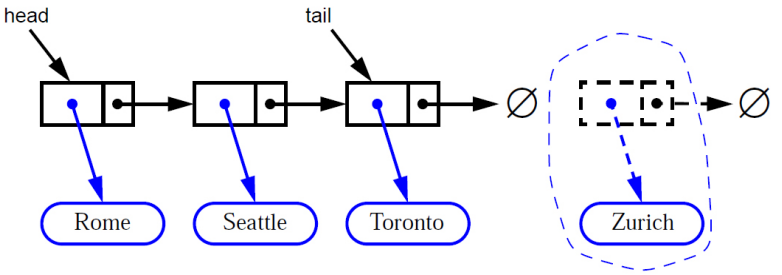
Brisanje zaglavlja se ostvaruje prebacivanjem reference *head* na drugi član (Slika-3):



Slika 4.5 Brisanje prvog elementa iz reda

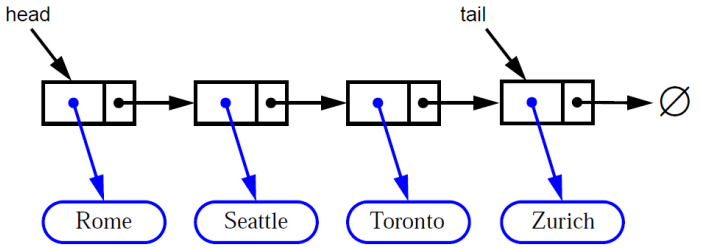
Ubacivanje podatka na početak je takođe jednostavna operacija.

Postupak dodavanja elemenata na kraj liste se sastoji iz dva koraka. Prvi je da se kreira novi čvor:



Slika 4.6 Kreiranje novog čvora, namenjenog da dodje na kraj reda

Povežite ga na kraj liste, i pomeriti referencu *tail* (Slika-5).



Slika 4.7 Pomeranje reference tail nakon dodavanja novog čvora na kraj reda

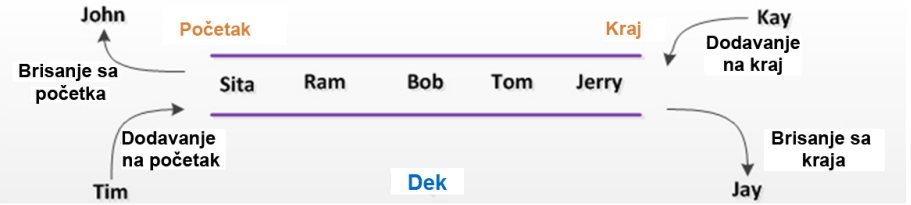
Kako bi izgledalo brisanje elementa sa kraja reda?

DVOSTRANI RED (DEQUE)

Dvostrani red ili dek je sličan steku i redu čekanja, ali dozvoljava ubacivanje i uklanjanje sa obe strane. Osnovne operacije su insertFirst(),removeFirst(),insertLast() i removeLast()

Ovaj red čekanja se zove i deque (dek). Sličan je steku i redu čekanja, ali dozvoljava ubacivanje i uklanjanje sa obe strane. Stek i red čekanja se lako mogu implementirati korištenjem deque koncepta.

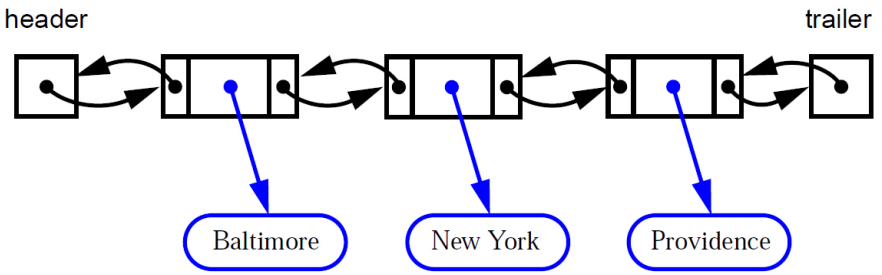
U nastavku je primer deka, i operacija sa dekom (Slika-1):



Slika 4.8 Dvostrani red

Dek može biti implementiran korišćenjem niza na isti način kao i red. Ovo možete uraditi kao zadatak za vežbu.

Međutim, korišćenje povezane liste za implementaciju deka nije baš najbolja opcija, jer je brisanje poslednjeg elementa u listi ne tako jednostava i brza operacija. Mnogo bolja opcija je korišćenje dvostruko povezane liste. Dek kao dvostruko povezana lista može biti grafički predstavljen na sledeći način:



Slika 4.9 Implementacija deka korišćenjem dvostruko povezane liste

Brisanje sa kraja jednostruko povezane liste nije baš jednostavna i brza operacija. Za implementaciju deka koristimo dvostruko povezanu listu, koja ima specijalne čvor zaglavlja, i specijalni čvor završnog elementa (čvorovi *header* i *trailer*).

Kao što znamo, čvor dvostruko povezane liste ima pokazivače (linkove) na prethodni i naredni član, tj. linkove *next* i *prev*. On pruža mogućnost upotrebe sledećih operacija:

```
setElement(Object e),
setNext(Object newNext)
setPrev(Object newPrev)
```

```
getElement()
getNext()
getPrev()
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zad 1. Koje od sledećih linija su pogrešne?

```
List<String> list = new ArrayList<>();
list.add("Tom");
list = new LinkedList<>();
list.add("Tom");
list = new GenericStack<>();
list.add("Tom");
```

Zad 2. (Uparivanje zagrada) Programski kod Java programa se sastoji iz ogromnog broja parova zagrada, kao što su:

- oble zagrade: **(** i **)**
- vitičaste zagrade: **{** i **}**
- uglaste zagrade: **[** i **]**

Primetimo da se odgovarajuće grupe zagrada ne mogu preklapati. Na primer, sledeći primer **(a{b)}** nije korektan. Napisati program koji proverava da li se Java programski kod sastoji iz pravilnih parova zagrada. Proslediti ime fajla kao argument komandne linije.

Poglavlje 5

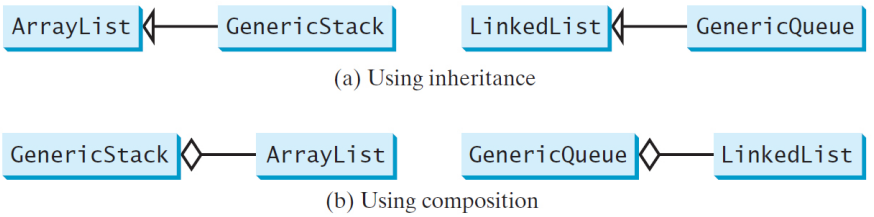
Stekovi i redovi u Javi

IMPLEMENTACIJA STEKOVA I REDOVA U JAVI

Stekovi u Javi su uglavnom implementirani korišćenjem listi baziranih na nizovima, dok su redovi uglavnom implementirani korišćenjem povezanih listi

Pošto operacije ubacivanja i brisanja na steku se obavljaju isključivo na vrhu steka, mnogo je efikasnije implementirati stek korišćenjem nizova nego povezanih listi. Kada je u pitanju red, pošto se brisanje vrši na početku liste, mnogo je efikasnije implementirati redove čekanja korišćenjem povezanih listi, nego korišćenjem listi baziranih na nizovima. U okviru ove sekcije je prikazana implementacija klase za stek korišćenjem listi baziranih na nizovima, odnosno implementacija redova čekanja korišćenjem povezanih listi.

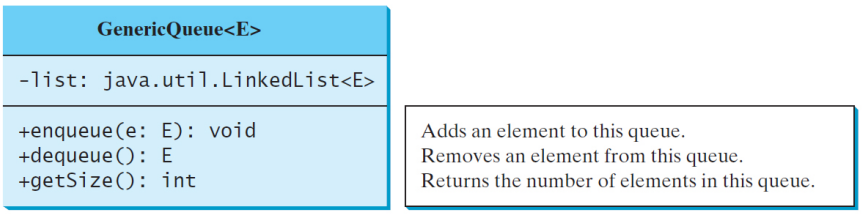
Postoje dva načina da se dizajniraju klase za stek i red:



Slika 5.1 Klase GenericStack i GenericQueue mogu biti implementirane korišćenjem nasleđivanja ili kompozicije.

- Korišćenjem nasleđivanja: Možete definisati klasu za stek koja će da nasledi klasu *ArrayList*, odnosno klasu za red koja će da nasledi (proširi) klasu *LinkedList*, kao što je prikazano na sledećoj Slici 1a.
- Korišćenjem kompozicije: Možete koristiti listu baziranu na nizu (objekat klase *ArrayList*) kao polje unutar klase koju definišete za stek, odnosno možete koristiti povezanu listu kao polje unutar generičke klase za red, kao što je prikazano na Slici 1b.

Oba pomenuta načina kreiranja su dovoljno dobar izbor, ali je za nijansu bolje koristiti kompoziciju jer vam omogućava da kreirate potpuno novu klasu za stek, odnosno red, bez nasledjivanja nepotrebnih i neodgovorajućih metoda klasa *ArrayList* odnosno *LinkedList*. Implementacija steka korićenjem koncepta kompozicije je opisana u okviru kursa CS102 – L06 (*GenericStack.java*). Na narednoj Slici 2 je prikazan UML dijagram klase *GenericQueue*.



Slika 5.2 Klasa GenericQueue koristi povezanu listu u cilju konstrukcije FIFO (first-in, first-out) strukture podataka.

GENERIČKA KLASA GENERICQUEUE

Generička klasa GenericQueue je kreirana korišćenjem kompozicije, pri čemu je povezana lista tipa java.util.LinkedList korišćena kao polje unutar generičke klase

Naredni listing prikazuje implementaciju klase *GenericQueue* korišćenjem principa kompozicije.

```
public class GenericQueue<E> {
  private java.util.LinkedList<E> list
    = new java.util.LinkedList<E>();

  public void enqueue(E e) {
    list.addLast(e);
  }

  public E dequeue() {
    return list.removeFirst();
  }

  public int getSize() {
    return list.size();
  }

  @Override
  public String toString() {
    return "Queue: " + list.toString();
  }
}
```

Povezana lista je kreirana sa ciljem da se elementi spakuju u red. Metod *enqueue(e)* dodaje novi element *e*na kraj reda. Metod *dequeue()* briše prvi element sa početka reda i kao rezultat vraća taj element. Metod *getSize()*kao rezultat vraća broj elemenata u redu.

UPOTREBA KREIRANOG STEKA I REDA

Stek je kreiran korišćenjem klase GenericStack, dok je red kreirana korišćenjem klase GenericQueue. Obe operacije push(enqueue) i pop(dequque) su vremenske složenosti O(1)

U cilju testiranja kreiranih klasa za generički stek i red koristićemo sledeći deo koda.

```
public class TestStackQueue {
  public static void main(String[] args) {
    // Create a stack
    GenericStack<String> stack = new GenericStack<String>();

    // Add elements to the stack
    stack.push("Tom"); // Push it to the stack
    System.out.println("(1) " + stack);

    stack.push("Susan"); // Push it to the the stack
    System.out.println("(2) " + stack);

    stack.push("Kim"); // Push it to the stack
    stack.push("Michael"); // Push it to the stack
    System.out.println("(3) " + stack);

    // Remove elements from the stack
    System.out.println("(4) " + stack.pop());
    System.out.println("(5) " + stack.pop());
    System.out.println("(6) " + stack);

    // Create a queue
    GenericQueue<String> queue = new GenericQueue<String>();

    // Add elements to the queue
    queue.enqueue("Tom"); // Add it to the queue
    System.out.println("(7) " + queue);

    queue.enqueue("Susan"); // Add it to the queue
    System.out.println("(8) " + queue);

    queue.enqueue("Kim"); // Add it to the queue
    queue.enqueue("Michael"); // Add it to the queue
    System.out.println("(9) " + queue);

    // Remove elements from the queue
    System.out.println("(10) " + queue.dequeue());
    System.out.println("(11) " + queue.dequeue());
    System.out.println("(12) " + queue);
  }
}
```

Kao što možemo da vidimo kreirali smo stek korišćenjem klase *GenericStack*, dok smo red kreirali korišćenjem klase *GenericQueue*. Takođe vidimo da se koriste metode *push* (*enqueue*) da bi dodali novi element u stek (red) odnosno metode *pop* (*dequeue*) da bi smo obrisali elemente sa steka (reda).

Kod steka, metod *push(e)* dodaje element na vrh steka, dok metod *pop()* element sa vrha steka i kao rezultat vraća obrisani element. Veoma je lako videti da su obe operacije *push* i *pop* kompleksnosti $O(1)$.

Kod reda, metod *enqueue(e)* dodaje element na kraj reda, dok metod *dequeue()* briše element sa početka reda. Pošto je red implementran korišćenjem povezane liste, veoma je lako zaključiti da je vremenska kompleksnost metoda *enqueue* i *dequeue* takođe $O(1)$.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zad 1. (Upotreba klase *GenericStack*) Napisati program koji prikazuje prvih 50 prostih brojeva u opadajućem poretku. Koristiti stek za čuvanje prostih brojeva.

Zad 2. (Implementacija klase *GenericQueue* korišćenjem nasleđivanja) U delu lekcije o Implementaciji stekova i redova u Javi, implementirali smo klasu *GenericQueue* korišćenjem principa kompozicije. Definisati novu klasu za red koja proširuje klasu *java.util.LinkedList*.

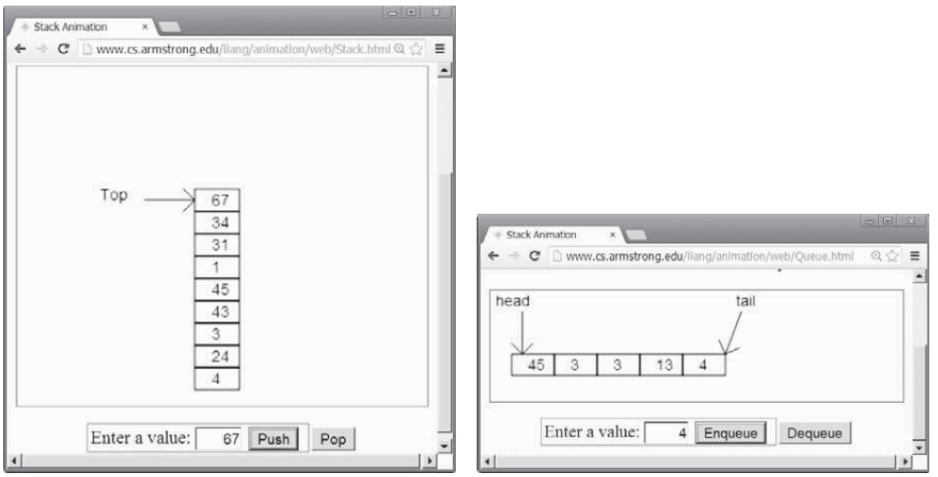
Zad 3. Napisati program koji prikazuje animaciju **push** i **pop** operacija na steku, kao što je prikazano na **Slika-1a**.

Zad 4.Napisati program koji prikazuje animaciju dodavanja i brisanja elementa iz reda, operacije **enqueue** i **dequeue** kao što je prikazano na **Slika-1b.**
Zad 5. Ako se *LinkedList* zameni sa *ArrayList* (linija 2-3) u implementaciji *GenericQueue*, koja će biti vremenska složenost za brisanje i dodavanje elementa u red?

Interaktivne aplikacije koje prikazuju rad stekova i redova su prikazane na sledećim linkovima:

www.cs.armstrong.edu/liang/animation/StackAnimation.html

www.cs.armstrong.edu/liang/animation/QueueAnimation.html



(a) Stack animation (b) Queue animation

Slika 5.3 Animacija koja prikazuje kako rade stek i red strukture.

Poglavlje 6

Prioritetni red čekanja

OSNOVI O PRIORITETNIM REDOVIMA

Kod prioritetnog reda elementima se dodeljuje odgovarajući prioritet. Stoga, pri pristupu elementima, važi da iz prioritetnog reda prvi izlazi onaj element sa najvećim prioritetom

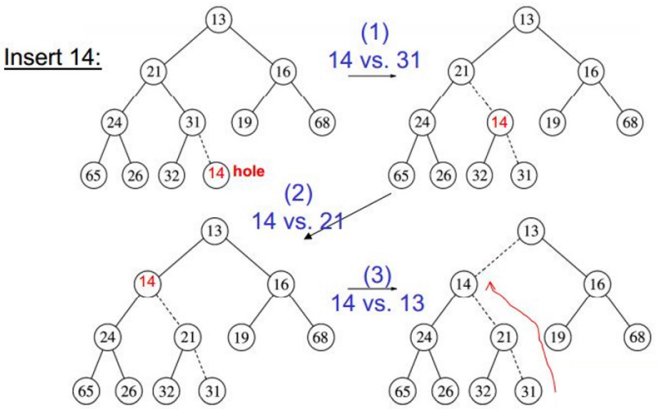
Redovi su standardni mehanizam za uređivanje elemenata, zadataka, procesa. Međutim, nekada su neki procesi mnogo važniji od drugih i moraju imati veći prioritet. U tu svrhu se uvodi prioritetni red kod koga svaki element ima prioritet ili vrednost.

Hip je jedan od načina da se predstavi red sa prioritetima. Operacije nad hipom koje je moguće uraditi su dodavanje elementa i uklanjanje onog čiji je prioritet najviši. Element koji ima najviši prioritet ima najmanji broj. Element sa prioritetom 0 je važniji i većeg prioriteta nego element sa brojem 1, i tako redom. Takođe je moguće kreirati hip gde najveći prioritet imaju elementi sa najvećom vrednošću.

Hip je binarno stablo sa svojstvima da su mu elementi, čvorovi strukture podataka i da je uređen po prioritetu elemenata.

Uvek važi pravilo da je prioritet roditeljskog čvora veći od prioriteta dece čvorova.

Prilikom dodavanja elementa, mora se poštovati dodavanje na čvor lista, nikako drugačije. Takođe, ukoliko je pravilo uređenja po prioritetu narušeno, stablo se mora reorganizovati. Postupak je opisan na sledećoj slici (Slika-1).



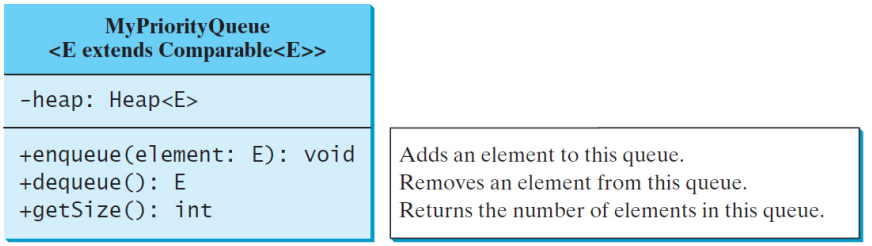
Slika 6.1 Opis operacije dodavanja čvora u hip

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

IMPLEMENTACIJA KLASE PRIORITYQUEUE

Prioritetni red može biti kreiran pomoću hipa gde se u korenu hipa stavlja objekat reda koji ima najveći prioritet. Klasa PriorityQueue je implementirana tako da u sebi sadrži objekat tipa

Prioritetni red može biti kreiran korišćenjem hipa, gde se u korenu hipa stavlja objekat reda koji ima najveći prioritet. Hip stablo smo opisali u lekciji o sortiranju (heap sort). Klasni dijagram prioritetnog reda koji je implementiran korišćenjem hipa je prikazan na sledećoj Slici-2.



Slika 6.2 Klasa MyPriorityQueue koristi hip da bi obezbedila „najveći koji uđe u hip, ide prvi napolje“ strukturu podataka (largest-in, first-out).

Implementacija hipa je prikazana u listingu koji sledi:

```
public class MyPriorityQueue<E extends Comparable<E>> {
  private Heap<E> heap = new Heap<E>();

  public void enqueue(E newObject) {
    heap.add(newObject);
  }

  public E dequeue() {
    return heap.remove();
  }

  public int getSize() {
    return heap.getSize();
  }
}
```

UPOTREBA PRIORITETNOG REDA

Prioritetni red je kreiran korišćenjem klase MyPriorityQueue. Imamo niz pacijenata koji se ubacuju u prioritetni red, a iz reda prvi izlazi pacijent sa najvećim prioritetom

Naredni listing predstavlja primer upotrebe prioritetnog reda za pacijente.

```
public class TestPriorityQueue {
  public static void main(String[] args) {
    Patient patient1 = new Patient("John", 2);
    Patient patient2 = new Patient("Jim", 1);
    Patient patient3 = new Patient("Tim", 5);
    Patient patient4 = new Patient("Cindy", 7);

    MyPriorityQueue<Patient> priorityQueue
    = new MyPriorityQueue<Patient>();
    priorityQueue.enqueue(patient1);
    priorityQueue.enqueue(patient2);
    priorityQueue.enqueue(patient3);
    priorityQueue.enqueue(patient4);

    while (priorityQueue.getSize() > 0)
    System.out.print(priorityQueue.dequeue() + " ");
  }

  static class Patient implements Comparable<Patient> {
    private String name;
    private int priority;

    public Patient(String name, int priority) {
    this.name = name;
    this.priority = priority;
    }

    @Override
    public String toString() {
    return name + "(priority:" + priority + ")";
    }

    public int compareTo(Patient o) {
    return this.priority - o.priority;
    }
  }
}
```

Rezultat programa biće

```
Cindy(priority:7) Tim(priority:5)
John(priority:2) Jim(priority:1)
```

Klasa *Patient* je definisana u linijama od 19 do 37. Kreirano je 5 pacijenata sa različitom vrednošću prioriteta. U liniji 8 je kreiran prioritetni red kao objekat klase *MyPriorityQueue*. Pacijenti se dodaju u red i linijama od 10 do 13. Zatim se u liniji 16 pacijent sa najvećim prioritetom izbacuje iz reda čekanja.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Koja je vremenska složenost **enqueue**, **dequeue** i **getSize** metoda u klasi *MyProrityQueue*?

Zadatak 2. Koje od sledećih naredbi su pogrešne?

```
MyPriorityQueue<Object> q1 = new MyPriorityQueue<>();
MyPriorityQueue<Number> q2 = new MyPriorityQueue<>();
MyPriorityQueue<Integer> q3 = new MyPriorityQueue<>();
MyPriorityQueue<Date> q4 = new MyPriorityQueue<>();
MyPriorityQueue<String> q5 = new MyPriorityQueue<>();
```

Zadatak 3. Izmeniti *MyPriorityQueue* klasu, koristiti generički parametar za poređenje objekata. Definisati novi konstruktor koji prima komparator kao argument na sledeći način:

```
PriorityQueue(Comparator<? super E> comparator)
```

Zaključak

O LISTAMA

Na osnovu svega obrađenog možemo izvesti sledeći zaključak:

Lista je apstraktni tip podataka koji implementira konačan uređen skup vrednosti, gde se iste vrednosti mogu ponoviti više puta. Liste se dele u dve osnovne grupe: Liste bazirane na nizovima, ili jednostavno – nizovi (array list), i liste bazirane na povezanim strukurama ili jednostavno - povezane liste (linked list).

Nizovi su najzastupljenije strukture podataka koje se koriste pri radu sa listama i skupovima podataka. Elementi niza su u memoriji podjednako udaljeni jedan od drugog. Povezana lista je lista elemenata koji su proizvoljno smešteni u memoriji i koji su međusobno povezani linkom (pokazivač/referenca na memorijsku adresu sledećeg člana).

Liste kao ADT se u programskom jeziku Java mogu predstaviti korišćenjem interfejsa ili apstraktnih klasa, čije se metode zatim implementiraju u konkretnim klasama za listu. Zajedničke operacija nad listama u Javi su definisane u interfejsu *List*. Ove operacije su zatim delimično implementirane u klasi *AbstraktList*, a ostale u klasama *ArrayList* i *LinkedList*.

Kod listi baziranih na nizovima, jednom kad je niz kreiran njegova dužine ne može biti promenjena. U slučaju da je niz ispunjen elementima često se vrši kreiranje novog niza duplo veće dužine i prebacivanje podataka u njega. Pre ubacivanja novog elementa na neko mesto unutar niza, definisano indeksom, neophodno je sve elemente desno od indeksa pomeriti za jedno mesto u desno.

U jednostruko povezanoj listi (*singly linked list*) čvor se sastoji od elemenata podataka (*data element*) i veze (pokazivača) na sledeći element u listi. U dvostruko povezanim listama svaki čvor osim vrednosti elementa i pokazivača *next* na sledeći čvor sadrži još jedan pokazivač i to na prethodni element. Kružnu listu možemo dobiti tako što se u polje pokazivača *next* poslednjeg čvora u listi umesto vrednosti *NULL* upiše pokazivač na prvi čvor u listi.

Klasa *ArrayList* je pogodna kada se vrši puno dodavanja/brisanja na kraju liste, ali nije pogodna kada postoji dosta ubacivanja/brisanja na početku liste. Metode *get* i *set* kod niza su dosta brze operacije. *LinkedList* je pogodna kada se vrši puno brisanja/dodavanja na početku liste, ali nije pogodna za puno dodavanja/brisanja na kraju liste. Metode *get* i *set* kod povezane liste su dosta spore operacije. Uvođenjem dvostruko povezanih listi, obezbeđujemo da su operacije dodavanja/brisanja i na početku i na kraju liste operacija složenosti O(1). Međutim, ne postoji način da ubrzamo rad operacija *get* i *set*.

STEKOVI, REDOVI, PRIORITETNI REDOVI

Na osnovu svega obrađenog možemo zaključiti sledeće:

Stek je struktura podataka u kojoj se podaci dodaju i uklanjaju na samo jednom kraju koji se zove vrh steka. Stoga, za stek je dovoljno čuvati pokazivač na početak steka. Stekovi se mogu implementirati na dva načina: korišćenjem **nizova** (statička implementacija) ili korišćenjem **povezanih listi** (dinamička implementacija). Osnovne operacije sa stekom su: **push** (dodavanje elementa na vrh steka) i **pop** (brisanje elementa sa vrha steka).

Red čekanja je struktura podatka kod koje se brisanje elemenata vrši na jednom, a dodavanje elemenata na drugom kraju. Stoga, red radi po **FIFO** (first-in-first-out) principu. Osnovne operacije sa redom su: **enqueue** (ubaci objekat na kraj reda) i **dequeue** (ukloniti element sa početka reda). Za implementaciju reda korišćenjem niza neophodne su dve promenljive: jedna za početak (front), a druga za kraj reda (rear)..

Kod implementacije reda korišćenjem povezane liste, potrebne su takođe dve reference (dva pokazivača), front i rear, za praćenje stanja na oba kraja reda. Zaglavlje liste je na početku reda, dok je rep liste na kraju reda.

Stekovi u Javi su uglavnom implementirani korišćenjem listi baziranih na nizovima, dok su redovi uglavnom implementirani korišćenjem povezanih listi

Dvostrani red ili **dek** je sličan steku i redu čekanja, ali dozvoljava ubacivanje i uklanjanje sa obe strane. Brisanje sa kraja jednostruko povezane liste nije baš jednostavna i brza operacija. Stoga je mnogo pogodnije implementirati dek korišćenjem dvostruko povezane liste.

Kod **prioritetnog reda** elementima se dodeljuje odgovarajući prioritet. Stoga, pri pristupu elementima, važi da iz prioritetnog reda prvi izlazi onaj element sa najvećim prioritetom. Prioritetni red čekanja se uglavnom kreira pomoću **hipa** (heap binarno stablo).

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, deseto izdanje, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.

[3] http://www.mycstutorials.com/articles/data_structures/linkedlists

[4] http://pages.cs.wisc.edu/~vernon/cs367/notes/4.LINKED-LIST.html#ll

[5] http://www.cs.wcupa.edu/rkline/ds/linked-lists.html

[6] https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

[7] https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm

[8] https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

[9] https://www.tutorialspoint.com/data_structures_algorithms/heap_data_structure.htm