Liste

Pozicioni ADT

- Pozicioni ADT modelira pojam mesta unutar strukture podataka gde je uskladišten jedan objekat
- On daje objedinjeni prikaz različitih načina skladištenja podataka, kao što su
  - ćelija niza
  - nod povezane liste
- Samo jedan metod:
  - object element(): vraća element uskladišten na određenom položaju

List ADT

- List ADT modeluje sekvencu pozicija skladišteći proizvoljne objekte
- Uspostavlja pre/posle odnos između pozicija
- Generičke metode:
  - size(), isEmpty()

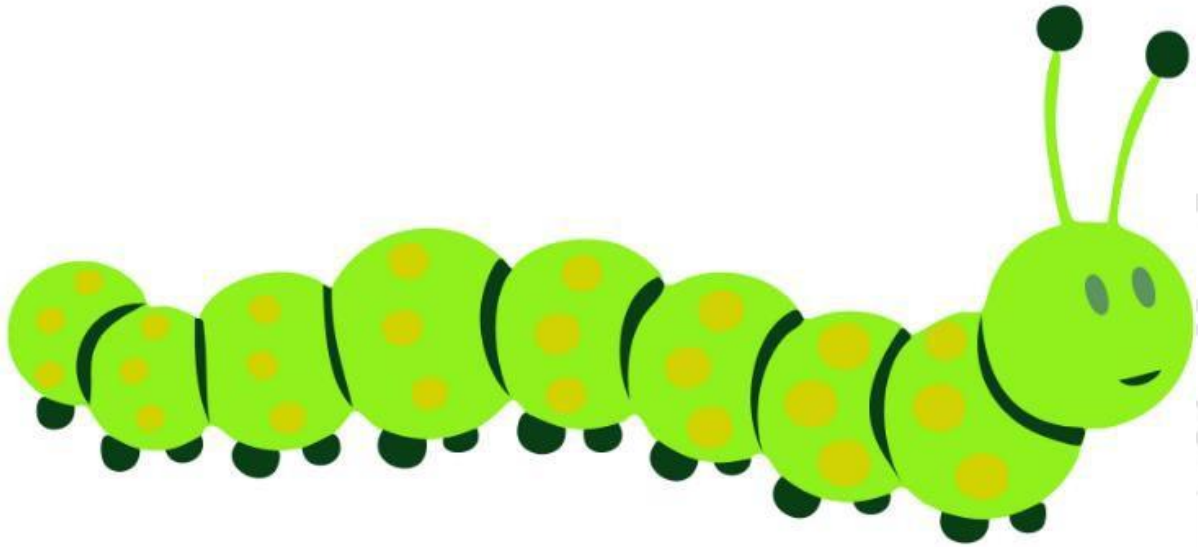
Metode pristupa:

- first(), last()
- prev(p), next(p)

Metode ažuriranja:

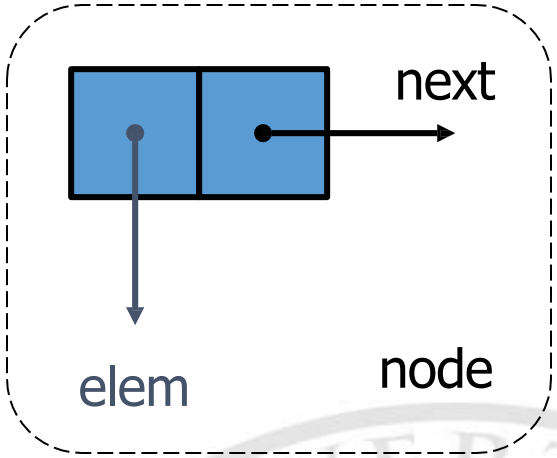
- replace(p, e)
- insertBefore(p, e), insertAfter(p, e),
- insertFirst(e), insertLast(e)
- remove(p)
- removeFirst()
- removeLast()

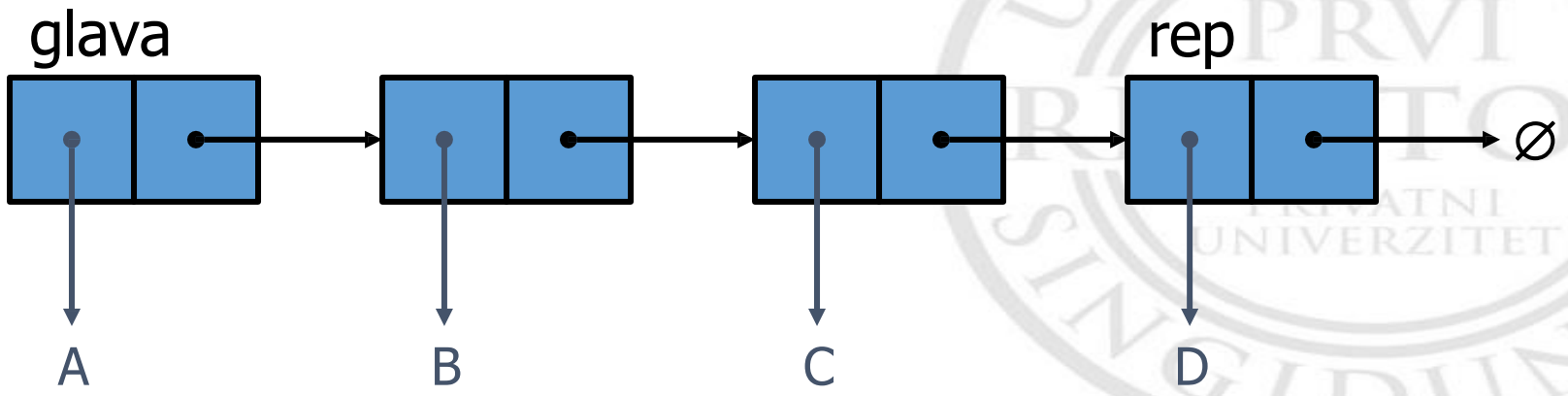
Povezane Liste



Jednostruko povezana lista

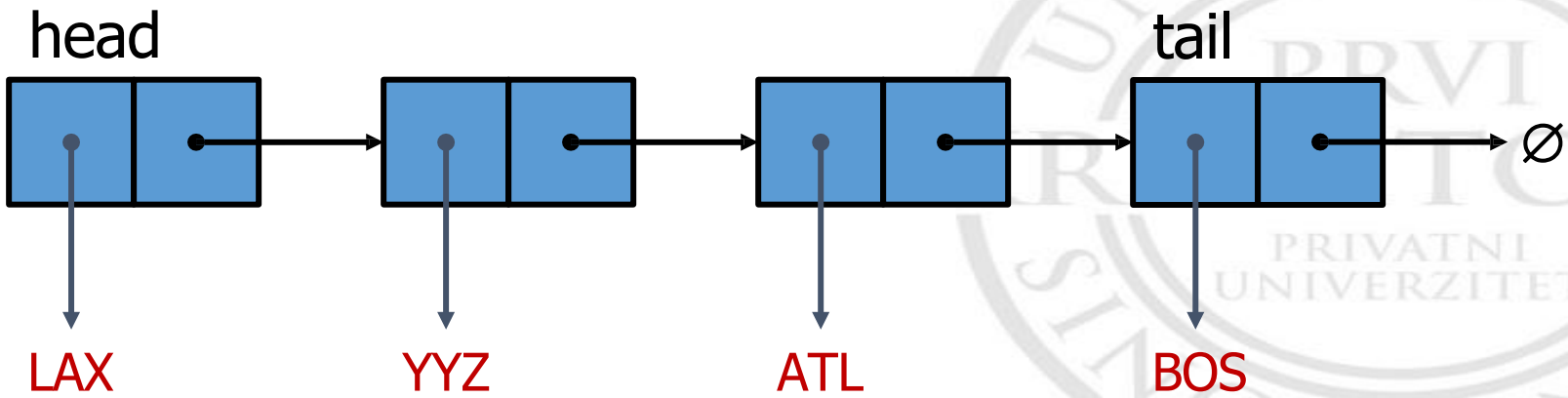
- Jednostruko povezana lista je poseban slučaj List ADT
- Svaki čvor skladišti
  - element
  - vezu sa sledećim čvrom
- Neke operacije List ADT nisu dozvoljene





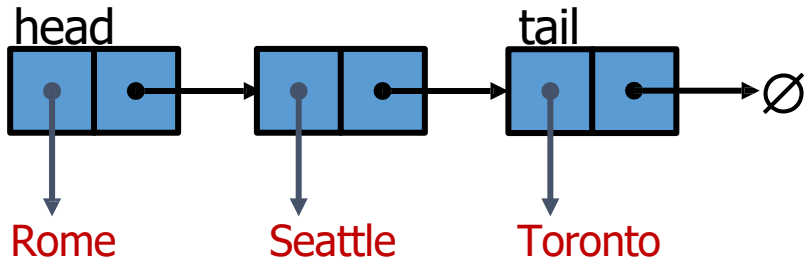
Glavne karakteristike

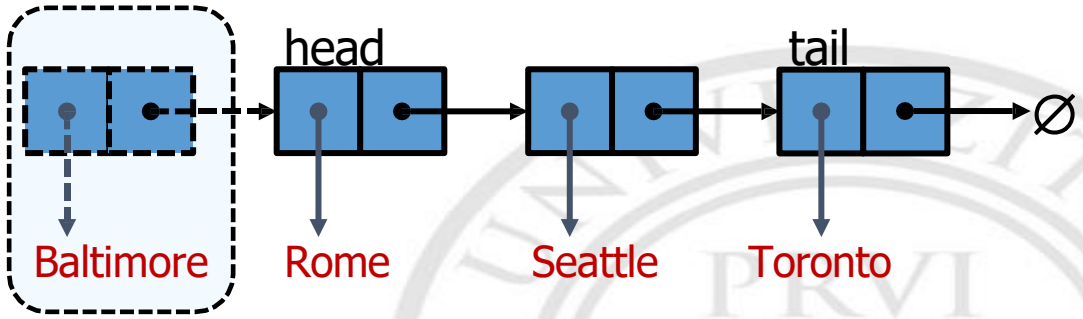
- Sledeća referenca je veza ili pokazivač na drugi čvor.
- Prvi i poslednji čvorovi se zovu glava i rep.
- Rep je nod koji ima null za sledeću referenca.
- Postoje samo dva čvora kojima se direktno pristupa: head i tail
  - Svakom drugom čvoru se pristupa sekvencijalno

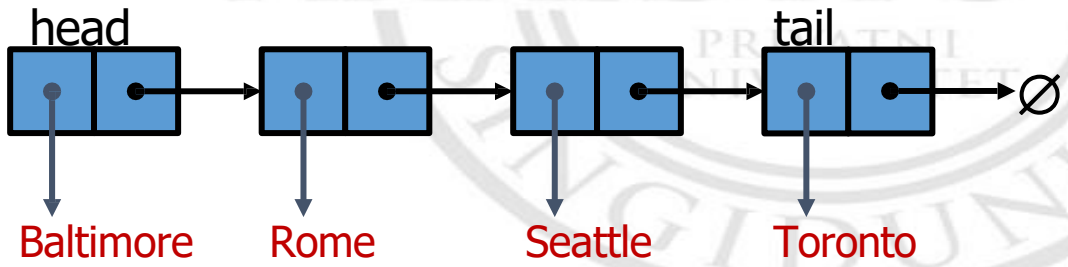


Ubacivanje na Head

1. Dodeliti novi čvor
2. Umetanje novog elementa
3. Novi čvor pokazuje na stari Head
4. Ažurira se Head da bi se usmeravao na novi čvori







Ubacivanje na Head

Algorithm *insertFirst*(*e*)

Create a new node *v*

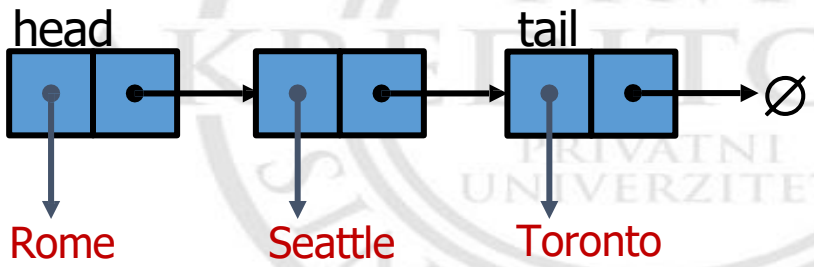
v.setElement(*e*)

v.setNext(*head*) {link *v* to its successor}

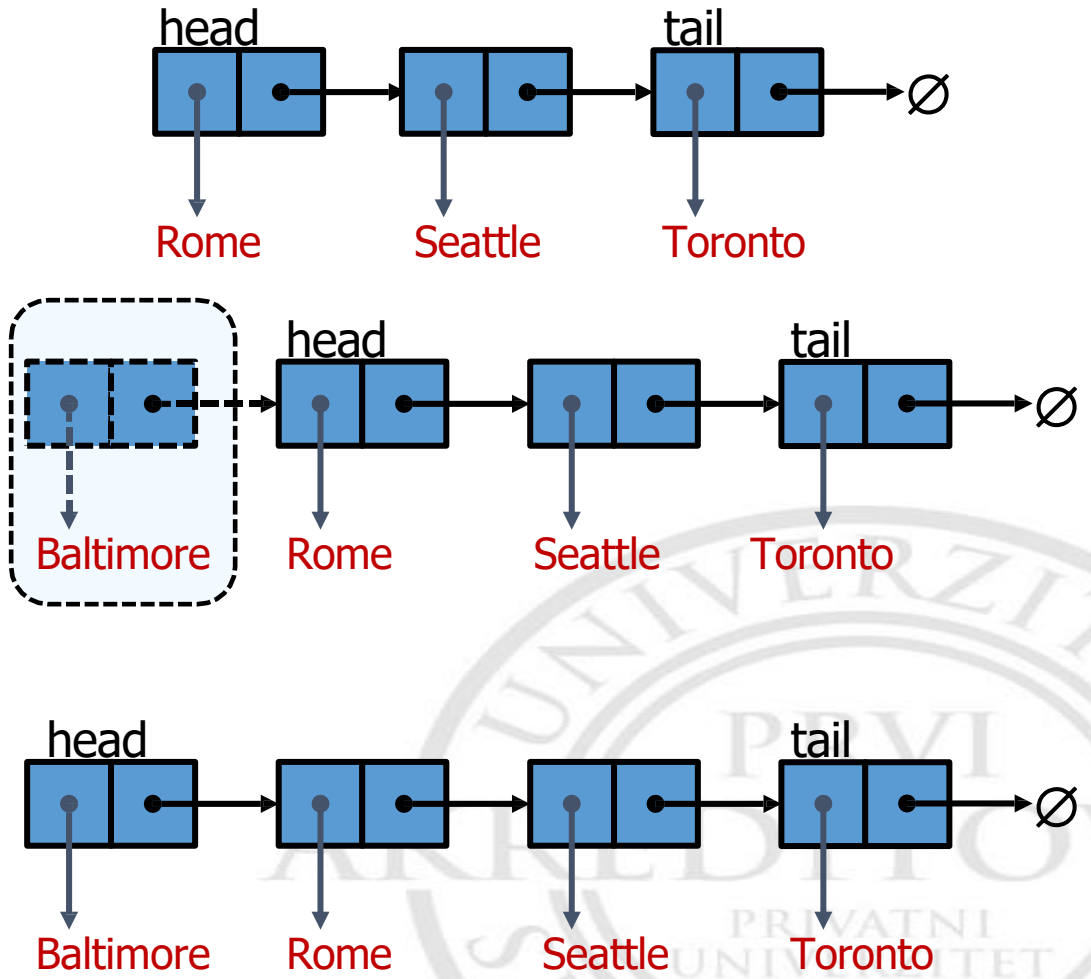
head ← *v* {link *v* to the head}

size ← *size* + 1 {increment the counter}

return *v*



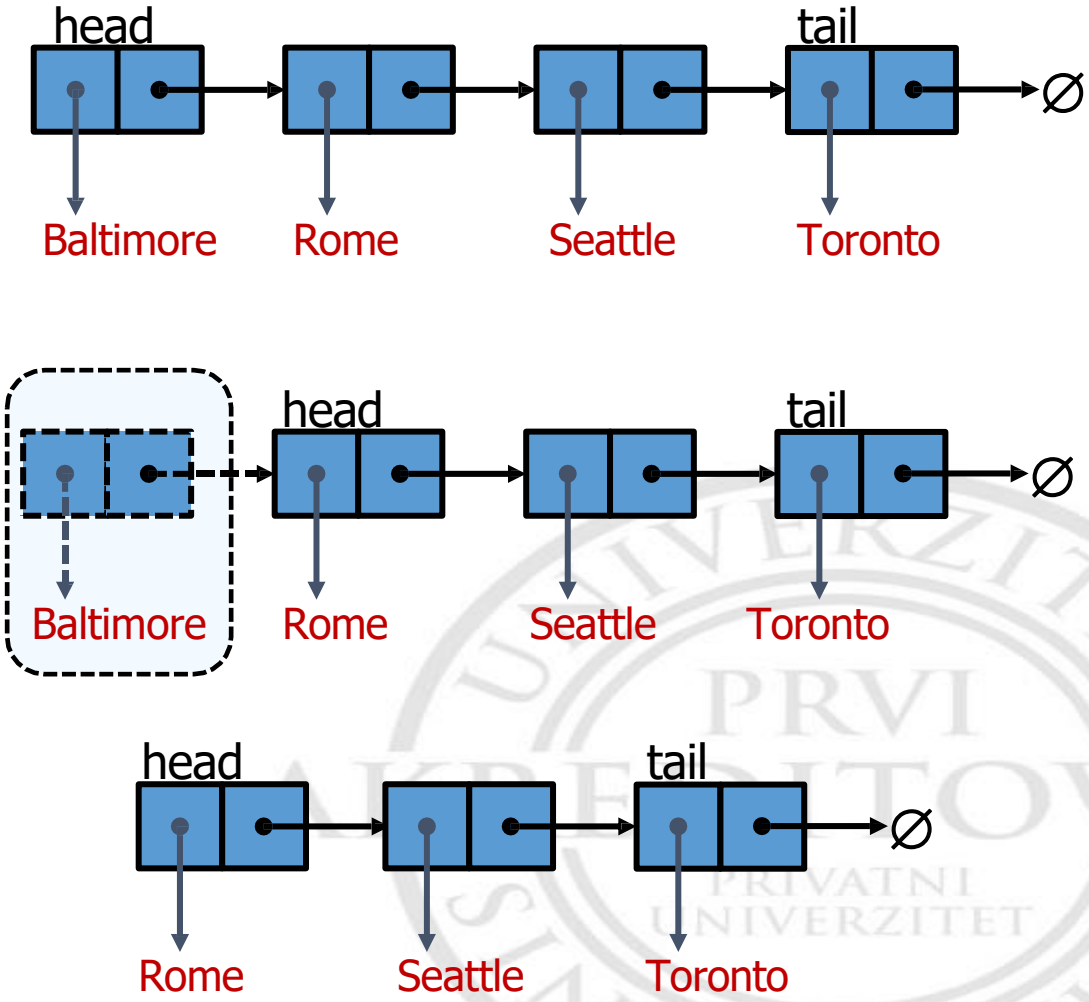
Ubacivanje na Head



- Glavi se može pristupiti za $O(1)$ vemena
- Za ubacivanje na Head potrebno $O(1)$ vremena

Uklanjanje na Headu

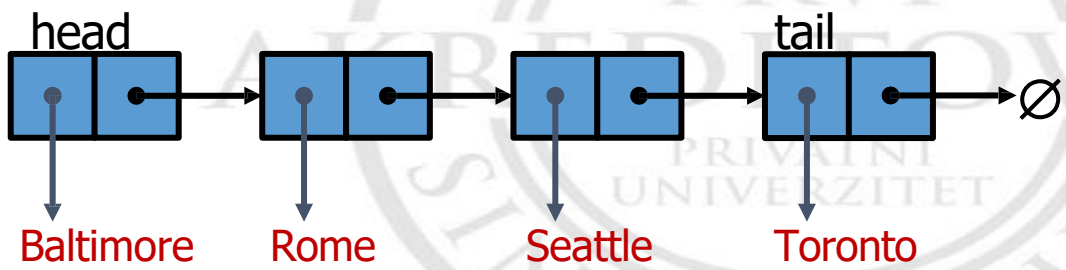
1. Ažurirati glavu da bi se pokazivala na sledeći čvor na listi
2. Dozvoliti garbage collector da odradi svoj posao



Uklanjanje na Headu

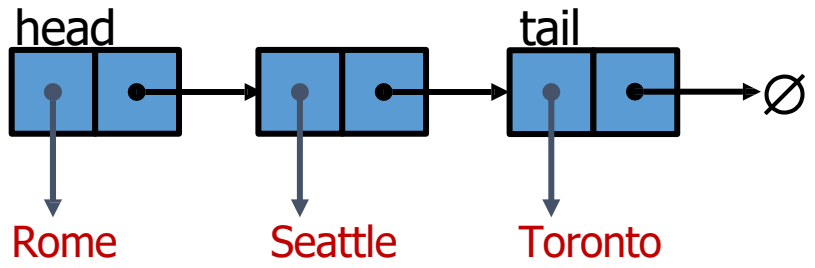
```
Algorithm removeFirst()
  if head = null then
     error "list is empty"
  t ← head
  head ← head.getNext()      {postaviti glavu da pokazuje na sledeći nod}
  t.setNext(null)            {napraviti glavin next pointer null}
  size ← size - 1            {smanjiti brojač}
  return t.element()         {vratiti element obrisanog čvora}
```

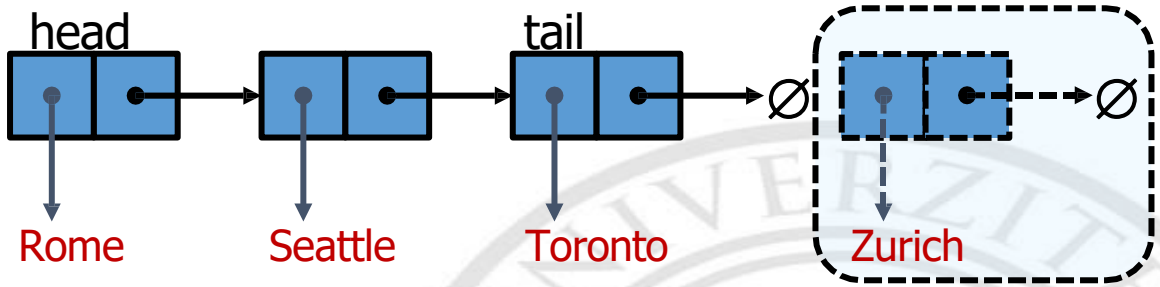
- Glavi se može pristupiti u ***O***(1) vremena
- Uklanjanje na glavi uzima ***O***(1) vremena

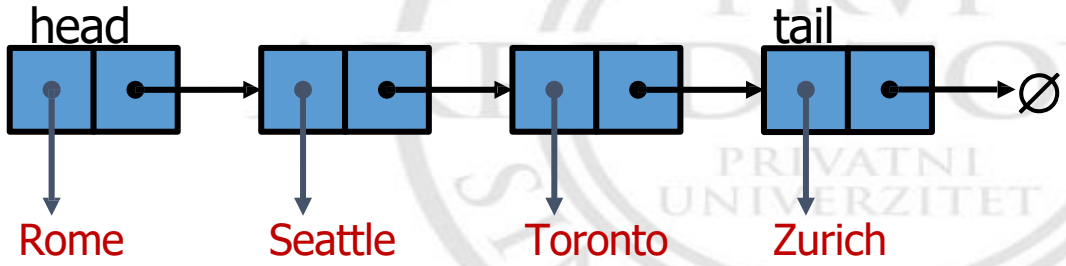


Ubacivanje na repu

1. Dodeliti novi čvor
2. Ubaciti novi element
3. Neka novi čvor point ka null
4. Neka stari poslednji čvor point ka novom čvoru
5. Ažurirati tail da point na novi čvor







Ubacivanje na repu

Algorithm *insertLast*(*e*)

Create a new node *v*

v.setElement(*e*)

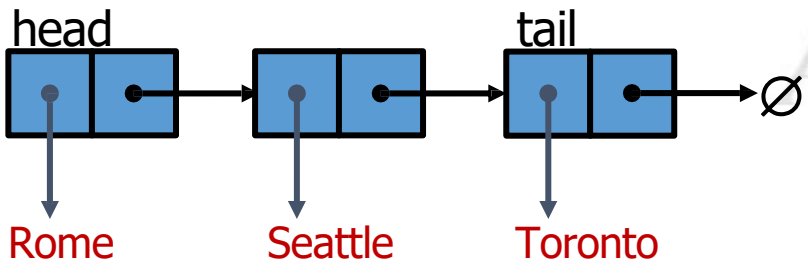
v.setNext(null)

tail.setNext(*v*) {neka satri tail čvor point ka novom čvoru}

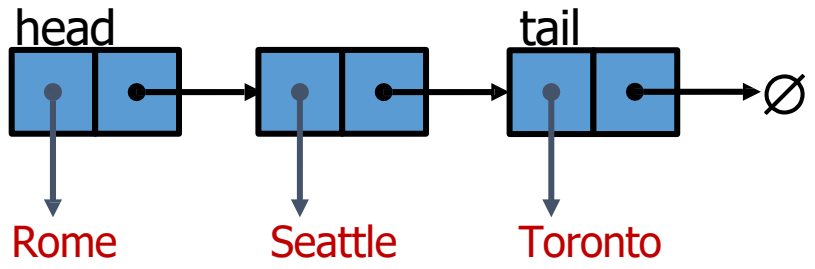
tail ← *v* {povezati *v* na tail}

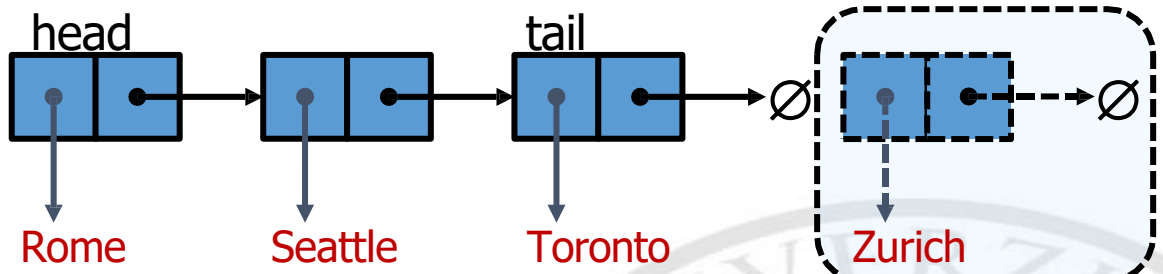
size ← *size* + 1 {povećati brojač}

return *v*

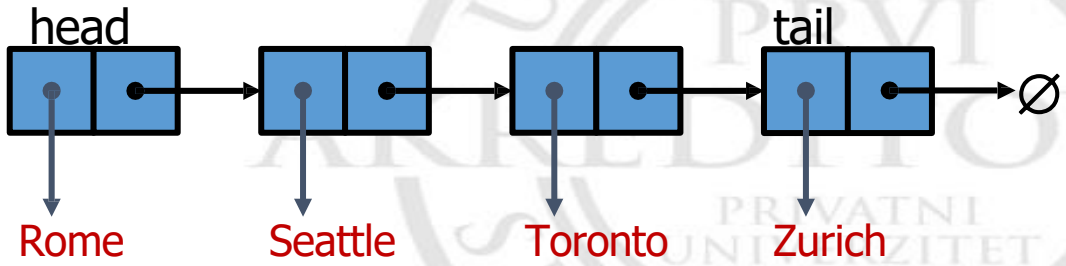


Ubacivanje na repu



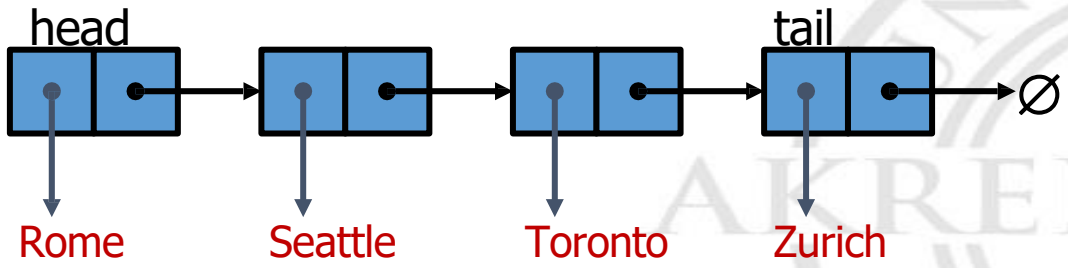


- Repu se može pristupiti u ***O***(1) vremena
- Umetanje u rep traje ***O***(1) vremena



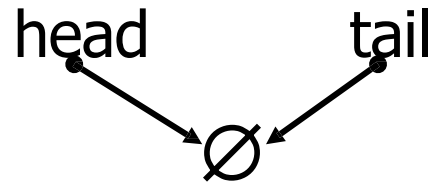
Uklanjanje na repu

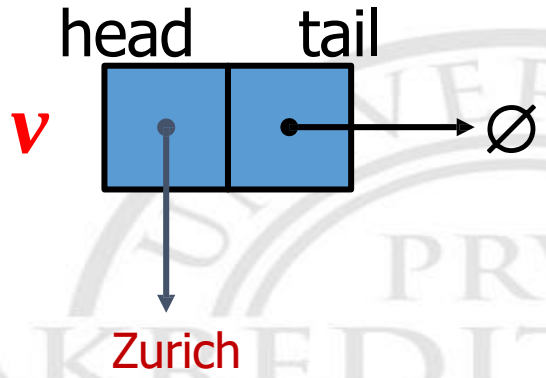
- Uklanjanje na repu za singly linked liste nije efikasno!
- Ne postoji constant-time način ažuriranja repa da pokazuje na prethodni čvor
- Ovo znači, uklanjanje na repu bi potrajalo ***O(n)***



Ubacivanje u praznu listu

- Glava i rep point na null
- Ubacivanje mora se tretirati kao poseban slučaj
- Novi čvor *v* points na null
- I head i tail point na *v*





Zašto su povezane liste važne?

Ograničenje:

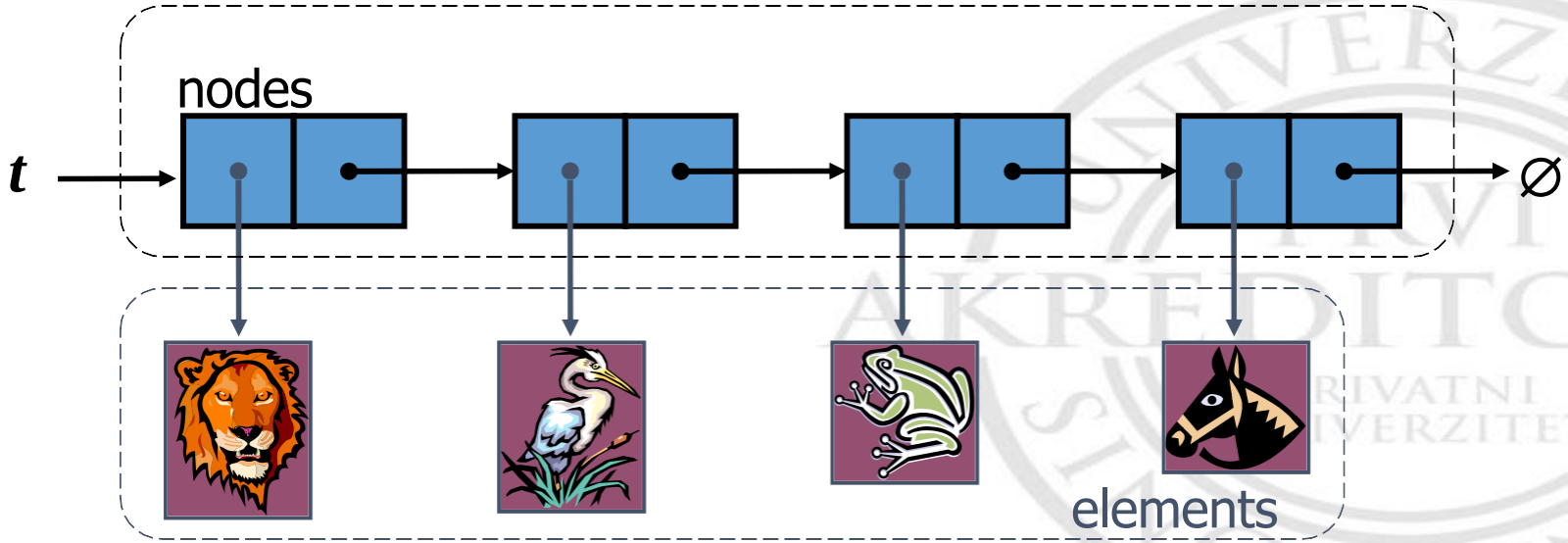
- za uklanjanje na repu je potrebno O(n)

Prednosti:

- Sve ostale operacije uzimaju O(1)
- Dovoljno su jednostavne za neke aplikacije:
  - Stacks
  - Redovi
- Enqueue/dequeue, push/pop u constant time O(1).
- Izbegavanje upućivanja na prethodni nod (doubly povezane liste), i otuda čuvanje malo prostora.

Stack sa Singly Linked Listom

- Možemo da implementiramo stack sa jednostrano povezanom listom
- Top element se smešta u prvi čvor liste
- Prostor koji se koristi je $O(n)$ i svaku operacija Stack ADT uzima $O(1)$ vremena

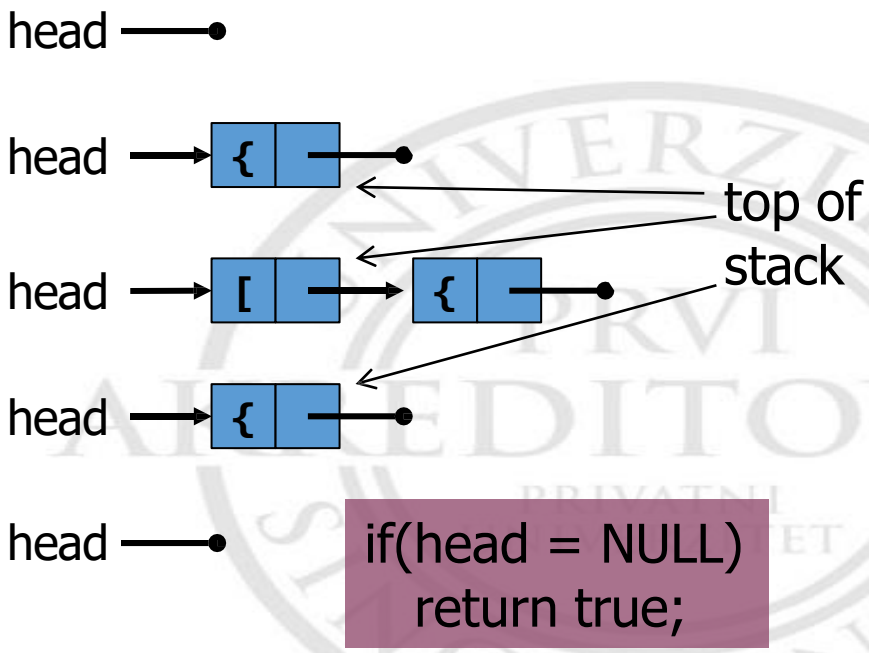


Stack sa Singly Linked Listom

- Ne treba pointer to “tail”
- Pointer na “head” je dovoljno dobar
- Primer: slaganje zagrada

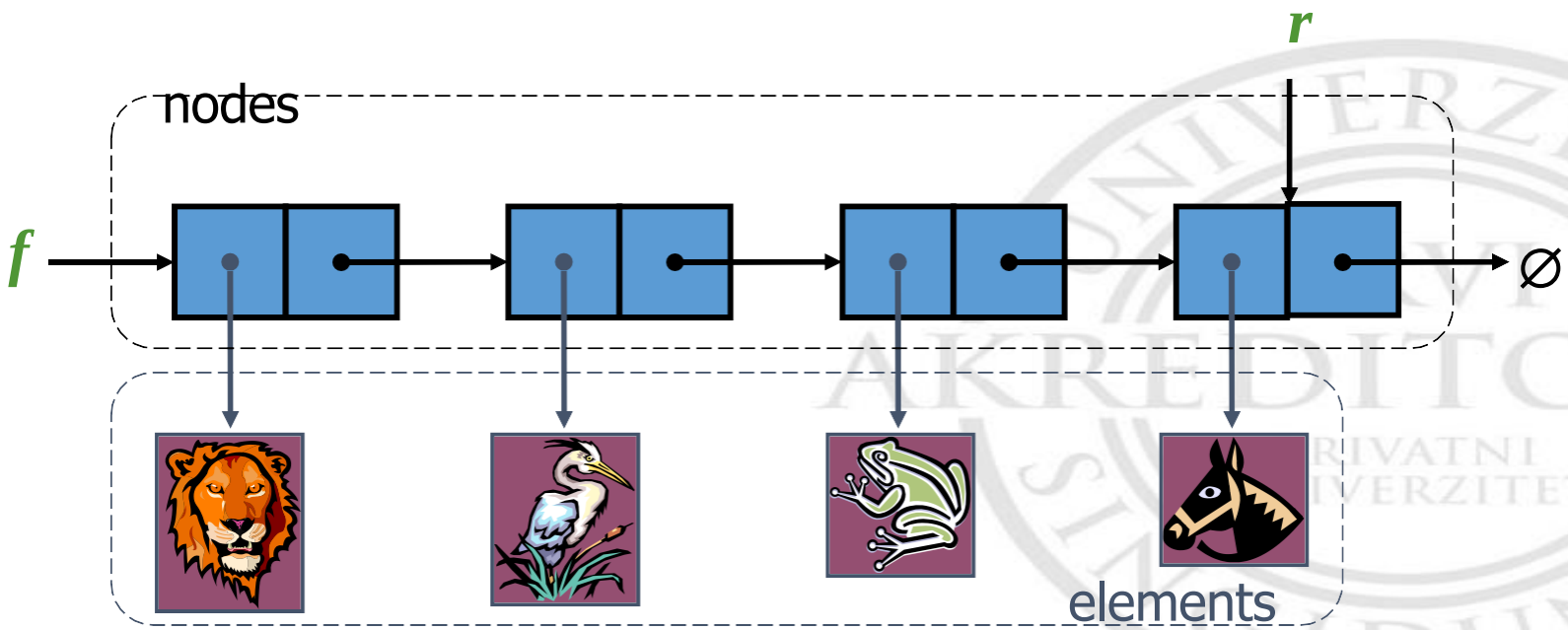
X = { [ ] }

1. Push “{“
2. Push “[“
3. Pop “[“ matches “]”
4. Pop “{“ matches “}”



Red sa jednostrano povezanom listom

- Možemo da implementiramo red sa jednostrano povezanom listom
  - front element se smešta na first čvor
  - rear element se smešta na last čvor
- Prostor koji se koristi je $O(n)$ i svaka operacija read ADT uzima $O(1)$ vremena



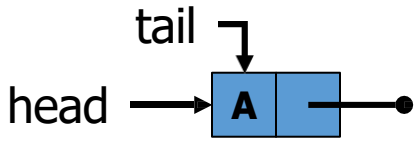
Red sa Singly Linked Listom

- enqueue(Node *v*) – ubaciti *v* na rep
- dequeue() – izbaciti čvor na glavi

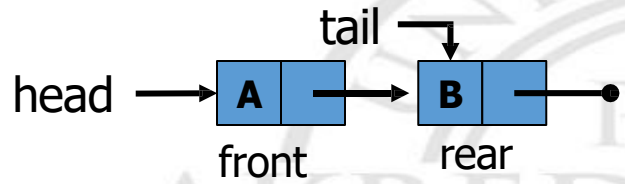
0. Empty queue:

Constant time operations
O(1)

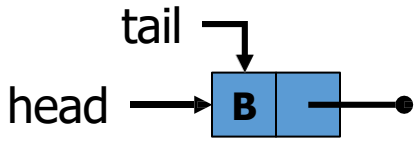
1. enqueue("A")



2. enqueue("B")



3. dequeue()



4. dequeue()

Primer implementacije

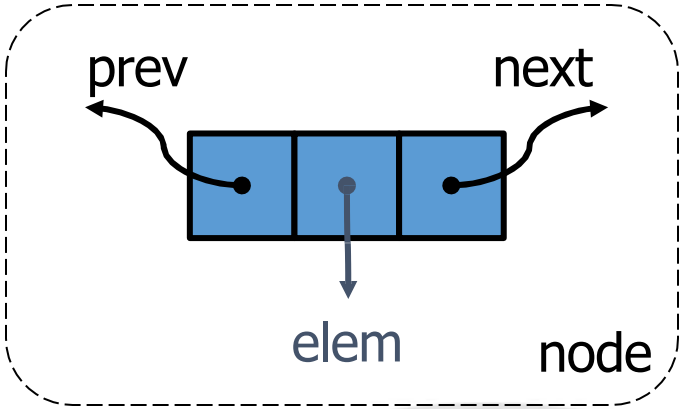
Node Class za Listu Čvorova

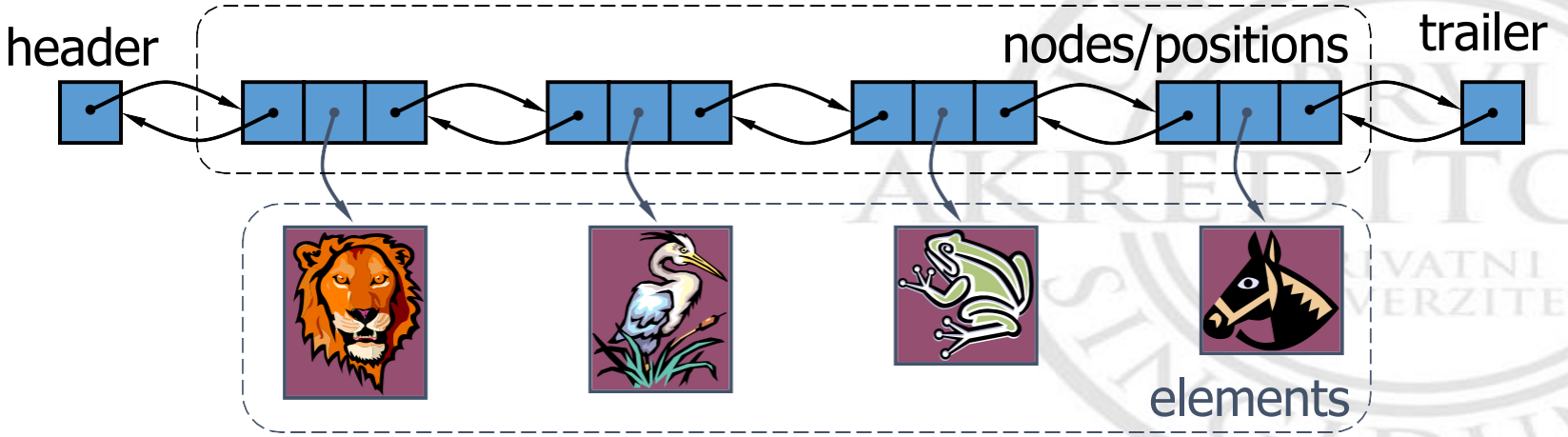
- Za jednostruko povezanu listu, moramo da kreiramo SinglyLinkedList klasu
- Ona sadrži accessor i update metode List ADT

```
public class Node      {
   // Instance variables:
   private Object element;
   private Node next;
   // Creates a node with null ref to its elem and next
    node.
   public Node()        {
     this(null, null);
   }
   // Creates a node with the given element and next
    node.
   public Node(Object e, Node n) {
       element = e;
       next = n;
   }
   // Accessor methods
   public Object getElement() {
     return element;
   }
   public Node getNext() {
     return next;
   }
   // Update methods
   public void setElement(Object newElem) {
       element = newElem;
   }
   public void setNext(Node newNext) {
       next = newNext;
   }
}
```

Dvostruko povezane liste

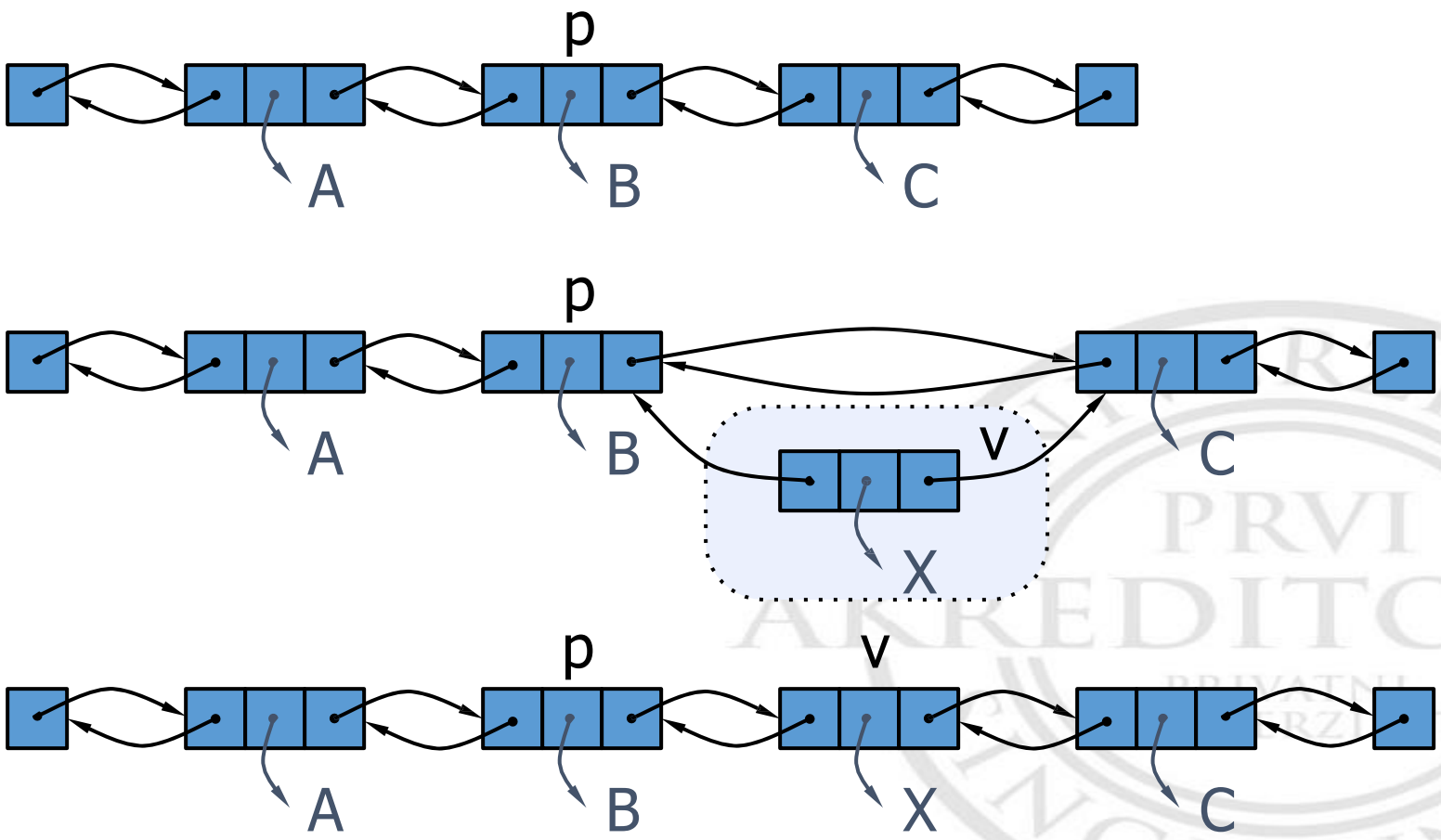
- Dvostruko povezane liste obezbedjuju prirodnu implementaciju List ADT
- Čvorovi implementiraju Poziciju i smeštaju:
  - element
  - link ka prethodnom čvoru
  - link ka sledećem čvoru
- Specijalni čvorovi trejlera i zaglavlja





Ubacivanje

- Vizualizujemo operaciju insertAfter(p, “X”), koji vraća poziciju v



Algoritam umetanja

Algorithm insertAfter(*p*,*e*):

Create a new node *v*

v.setElement(*e*)

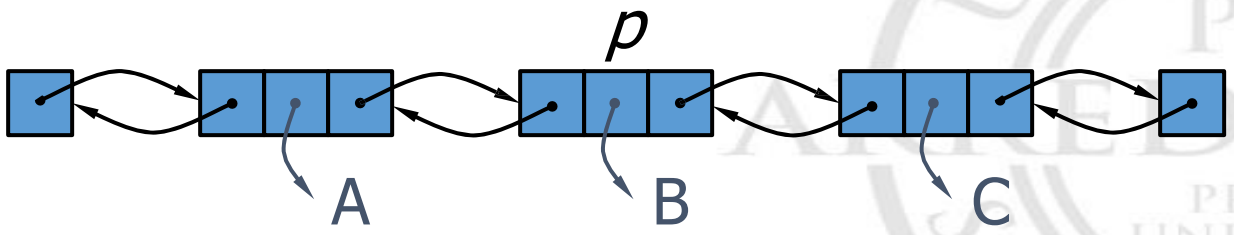
v.setPrev(*p*) {link *v* na njegovog prethodnika}

v.setNext(*p*.getNext()) {link *v* na njegovog naslednika}

(*p*.getNext()).setPrev(*v*) {link *p-ovog* starog naslednika na *v*}

p.setNext(*v*) {link *p* na njegovog novog naslednika, *v*}

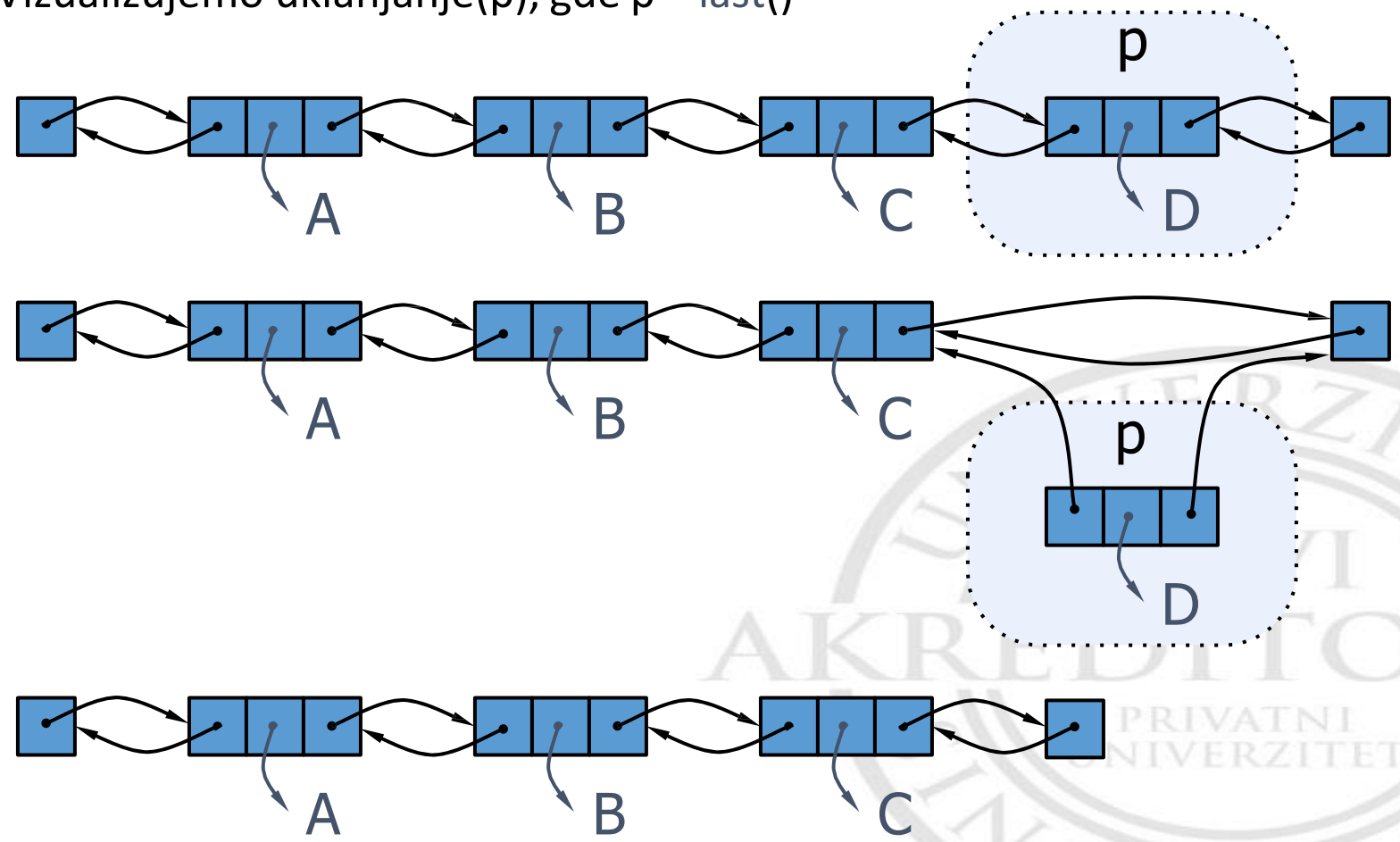
return *v* {pozicija za element *e*}



Brisanje

- Vizualizujemo uklanjanje(p), gde p = last()

radi i za druge slučajeve



Algoritam brisanja

Algorithm remove(*p*):

t = *p*.element() {temp. varijabla da čuva return value}

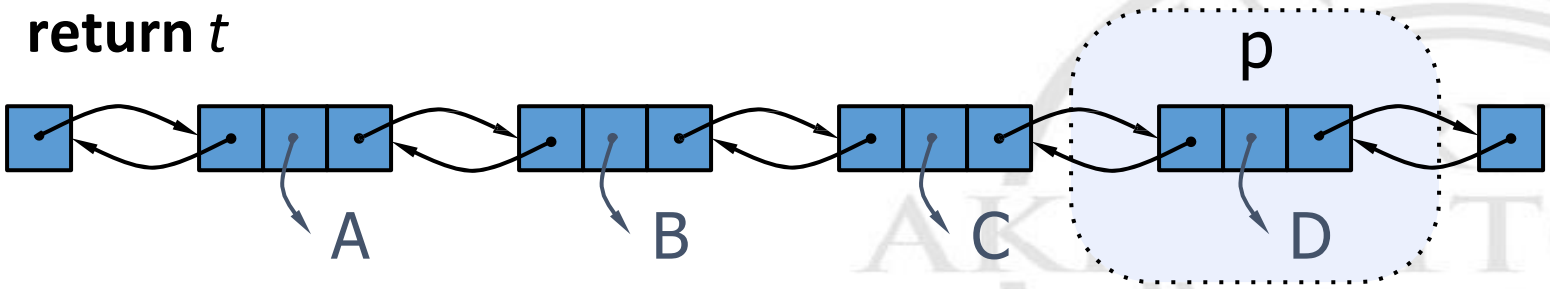
(*p*.getPrev()).setNext(*p*.getNext()) {linking out *p*}

(*p*.getNext()).setPrev(*p*.getPrev())

p.setPrev(**null**) {poništavanje pozicije *p*}

p.setNext(**null**)

return *t*



Note: remove(p) radi za bilo koji oglas na listi, ne samo za poslednji.

Performanse

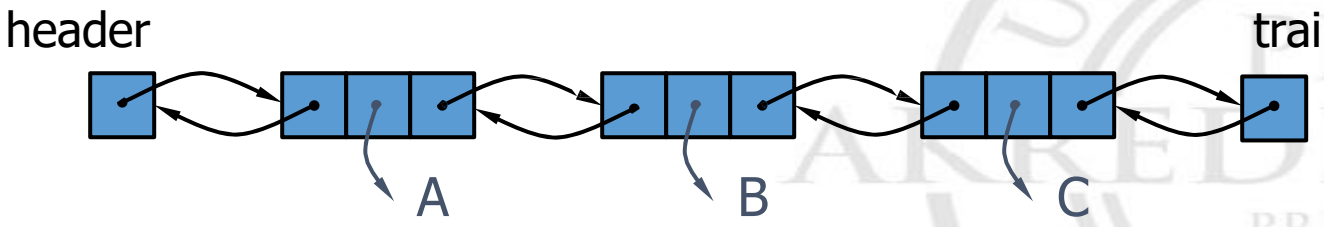
- U implementaciji List ADT putem a doubly linked liste
  - Prostor koji lista koristi sa n elemenata je $O(n)$
  - Prostor koji koristi svaka pozicija liste je $O(1)$
  - Sve operacije List ADT se izvršavaju u $O(1)$ vremena
  - Operation element() od Position ADT se izvršava u $O(1)$ vremena

Metode pristupa

- **Algorithm** first()
  return header {or header.getNext()}
- **Algorithm** last()
  return trailer {or trailer.getPrev()}
- **Algorithm** prev(p)
  return p.getPrev()
- **Algorithm** next(p)
  return p.getNext()

Ostale metode ažuriranja

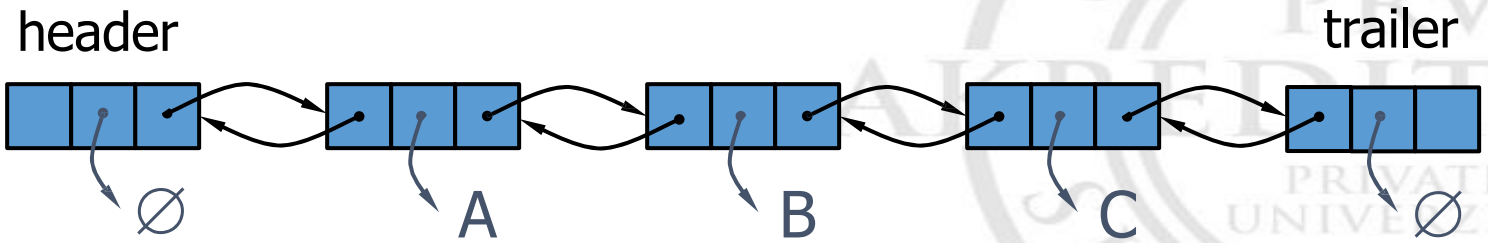
- replace(p, e)
  - p.setElement(e)
- insertBefore(p, e)
  - Slično sa insertAfter(p, e)
- insertFirst(e), insertLast(e)
  - Slično sa insertAfter and insertBefore



Saveti za implementaciju

- Korišćenje odvojenih čvorova za glavu i rep omogućava nam da koristimo iste algoritme umetanja i brisanja za prvi i poslednji čvor

- Ovo može zavisiti od programskog jezika koji se koristi za implementaciju



Reference

1. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014.
3. Java Documentation (Java SE 8):
   http://www.oracle.com/technetwork/java/javase/overview/index.html
   https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Review

1. Dajte glavne karakteristikeList ADT.
2. Diskutujte o vremenu umetanja u glavu/rep u singly-linked listama. Uradite isto za uklanjanje.
3. Uradite isto za dvostruko povezanu listu.
4. Opišite implementaciju steka u singly-linked listu. Šta je sa redom.
5. Uradite isto za dvostruko povezanu listu.
6. Uporedite ove implementacije sa onima iz array-based stack i reda.
7. Ako promenimo redosled izjava u algoritmu insertAfter, šta bi bio rezultat? Uraditi isto za brisanja.
8. Dajte pravi primer/problem koji se može primeniti/rešiti korišćenjemsingly-linked list and doubly-linked list. Diskutujte o rešenju.