Algoritmi i strukture podataka

- predavanja -

5. Lista

Osnovni pojmovi

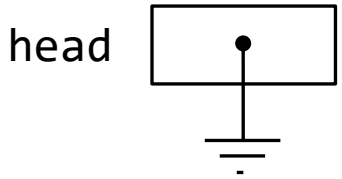
- linearna lista $A=(a_1,a_2,...a_n)$ struktura je podataka koja se sastoji od uređenog niza elemenata odabranih iz nekog skupa podataka
- za linearnu listu kažemo da je prazna ako ima $n=0$ elemenata
- elementi liste a_i nazivaju se još i atomi
- lista se može realizirati:
  - statičkom strukturom podataka (poljem)
  - dinamički (listom povezanih atoma)

Realizacija liste dinamičkom strukturom

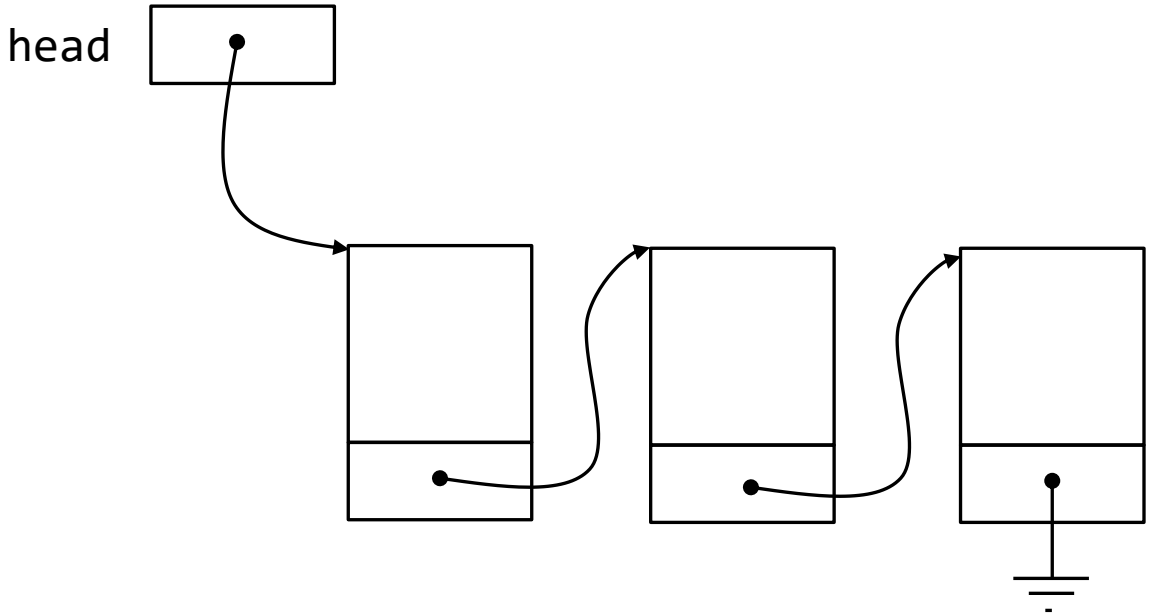
- dinamička podatkovna struktura za realizaciju liste sastoji se od pokazivača na prvi element liste i od proizvoljnog broja atoma
- svaki se atom sastoji od podatkovnog dijela i pokazivača na sljedeći element liste
- memorija za svaki atom liste zauzima se dinamički u trenutku kad je potrebna za pohranu podataka, a oslobađa kad se podatak briše
- granulacija je veličine atoma

Realizacija liste dinamičkom strukturom

- Prazna lista



- Neprazna lista

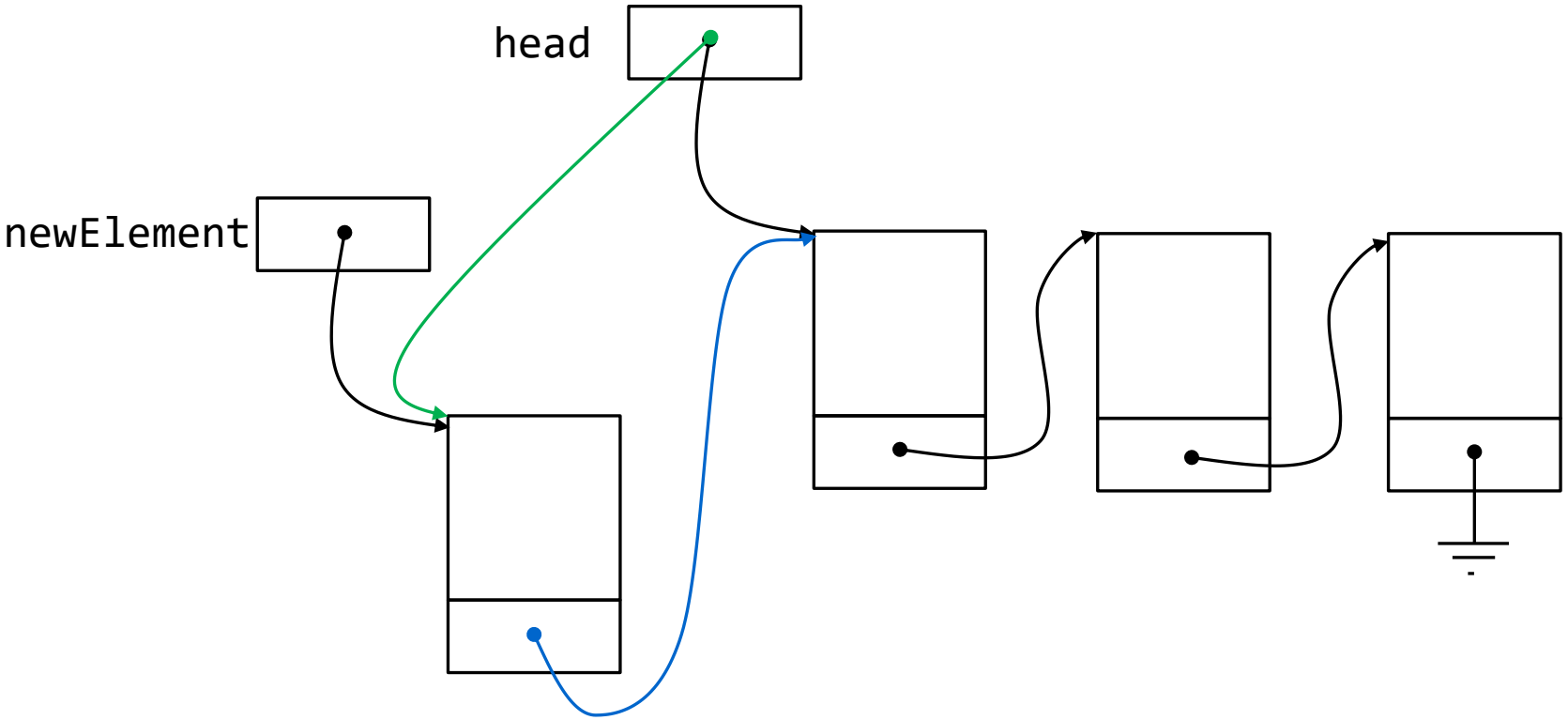


Element liste i realizacija

```
template <class X> struct ListElement {
   X item;
   ListElement<X> *next;
};
```

- Dodavanje na početak liste: O(1)
- Dodavanje na kraj liste: O(n)

Lista - umetanje na početak

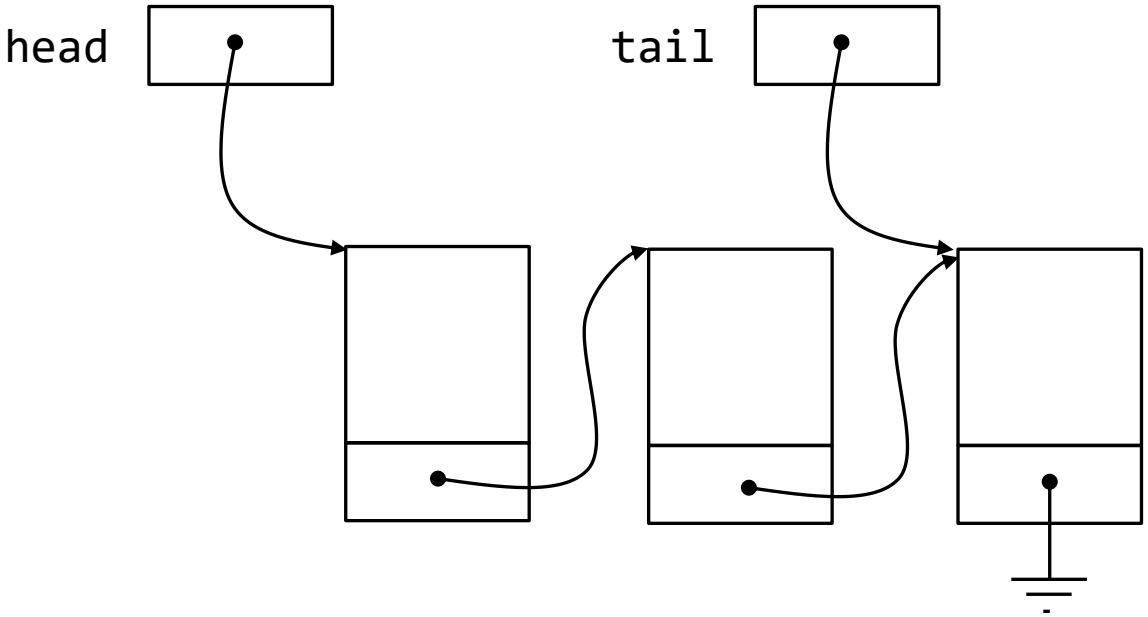



```
newElement->next = head;
head = newElement;
```

List.cpp

Realizacija liste s pokazivačem na kraj

- Dodavanje na kraj liste može biti O(1) uz:

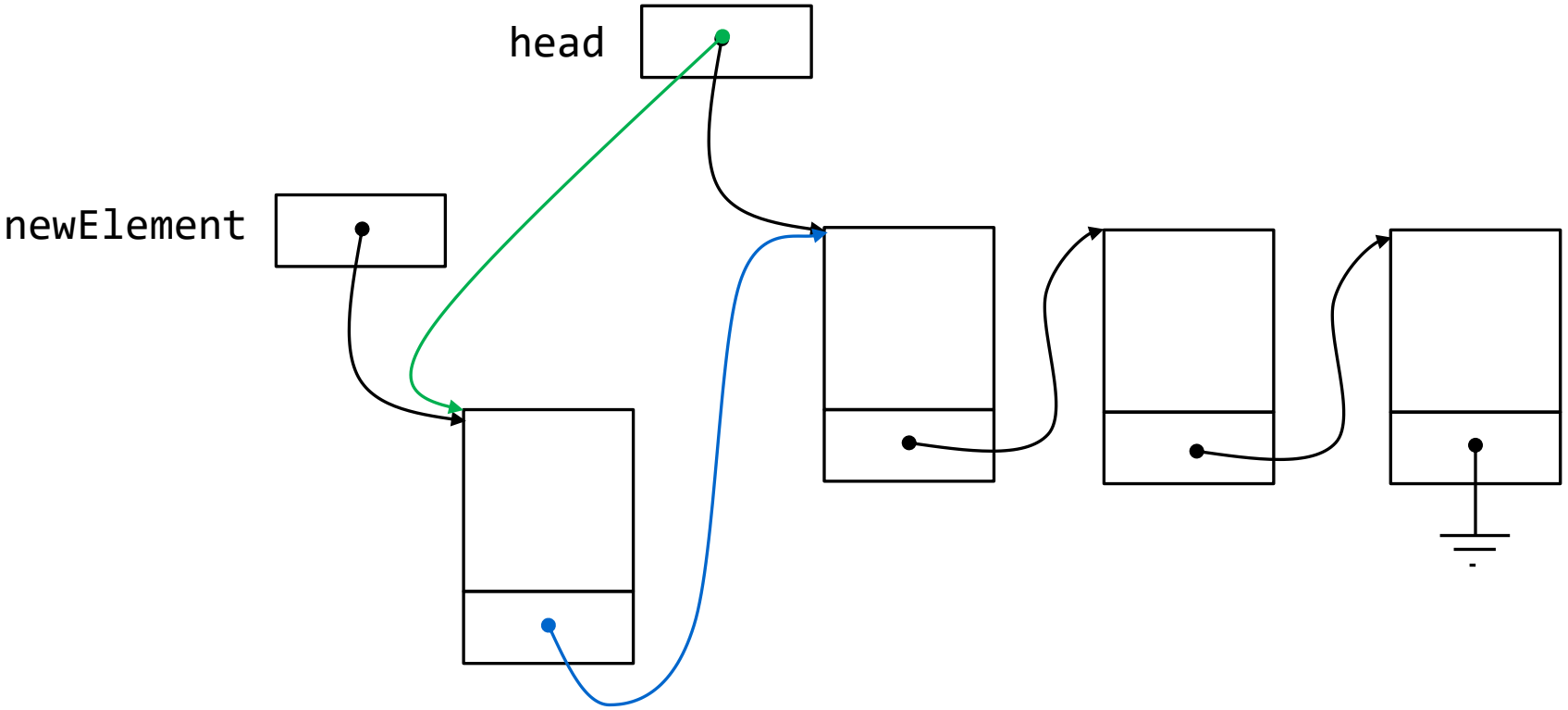


ListWithTail.cpp

Sortirana lista

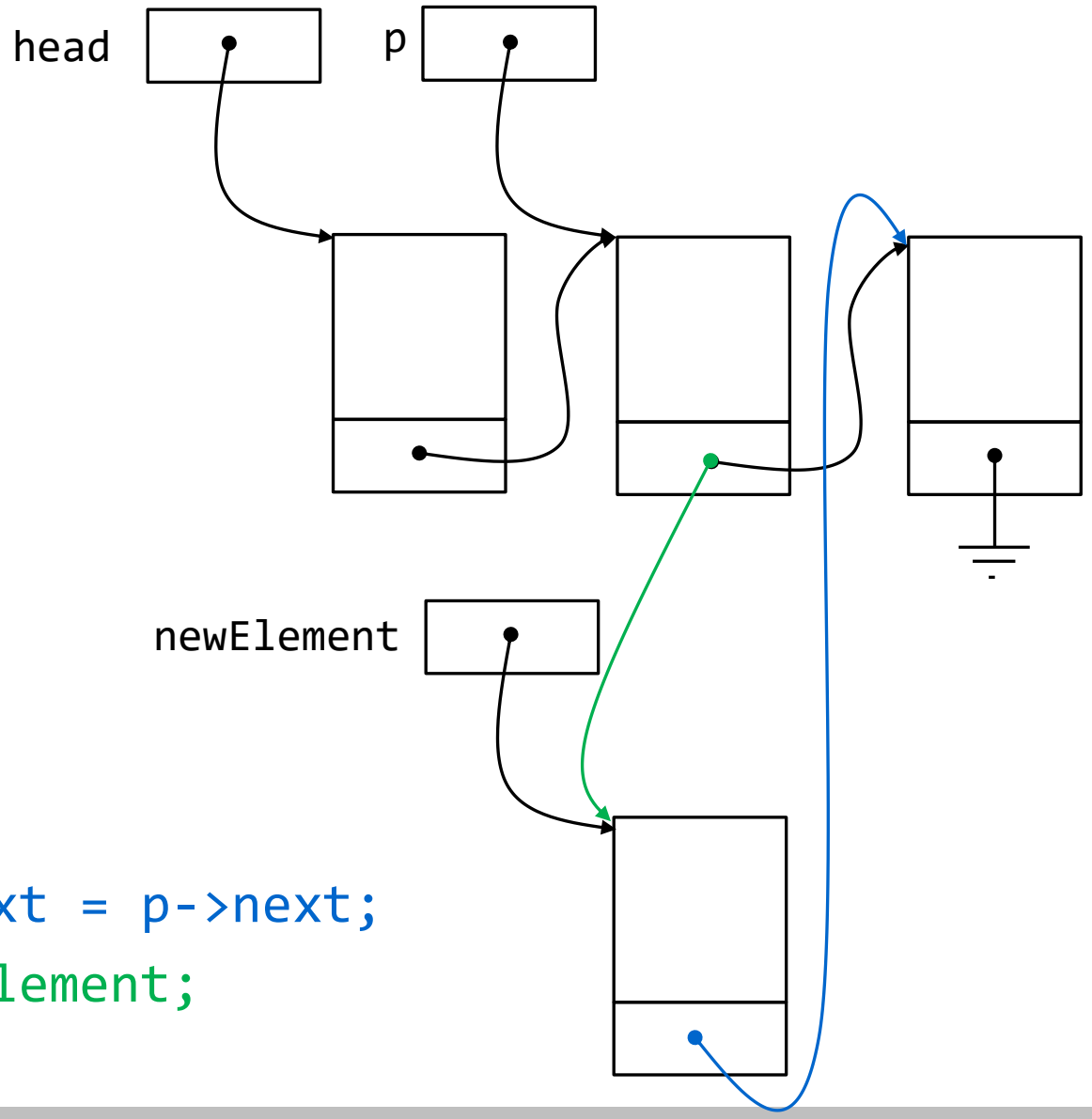
- Elementi liste sortirani po nekom redoslijedu
- Kod dodavanja elementa, potrebno je iterirati: O(n)
- Razlikuju se dva slučaja
  - Umetanje na početak
  - Umetanje iza elementa u listi

Sortirana lista – umetanje na početak




```
newElement->next = head;
head = newElement;
```

Sortirana lista - umetanje iza elementa liste

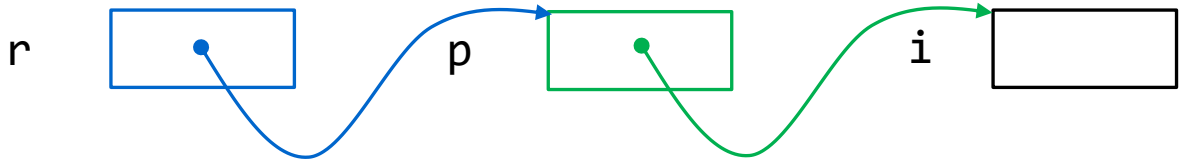



```
newElement->next = p->next;
p->next = newElement;
```

Može se objediniti, koristeći **

- Općenito

```
int i;
int *p = &i;
int **r = &p;
```



Kako čitati definiciju?

```
int *p;
```

```
int *p;
```
ali i
```
int *p;
```

p je int *

*p je int

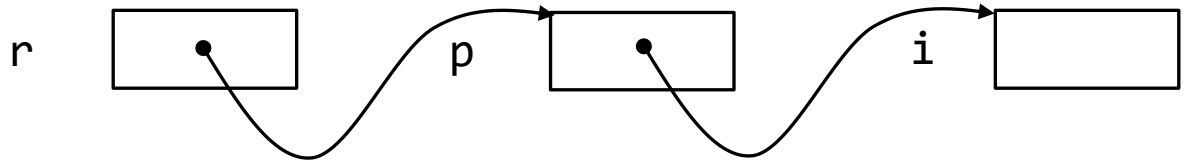
```
int **r;
```

```
int **r;
```
ali i
```
int **r;
```
ali i
```
int **r;
```

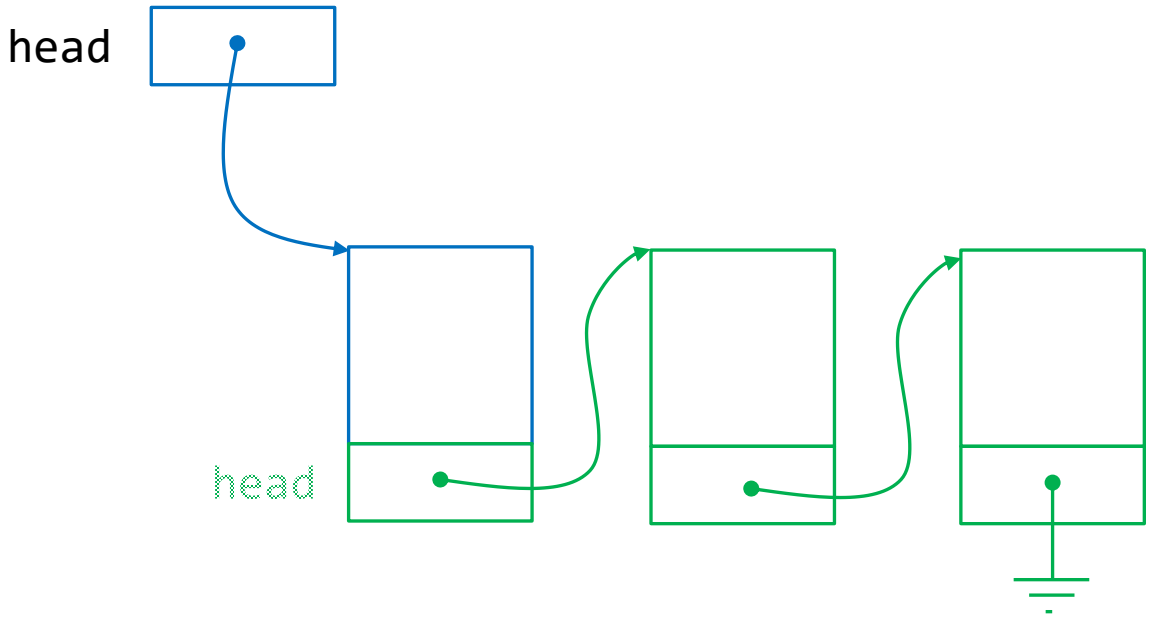
r je int **

*r je int *

**r je int

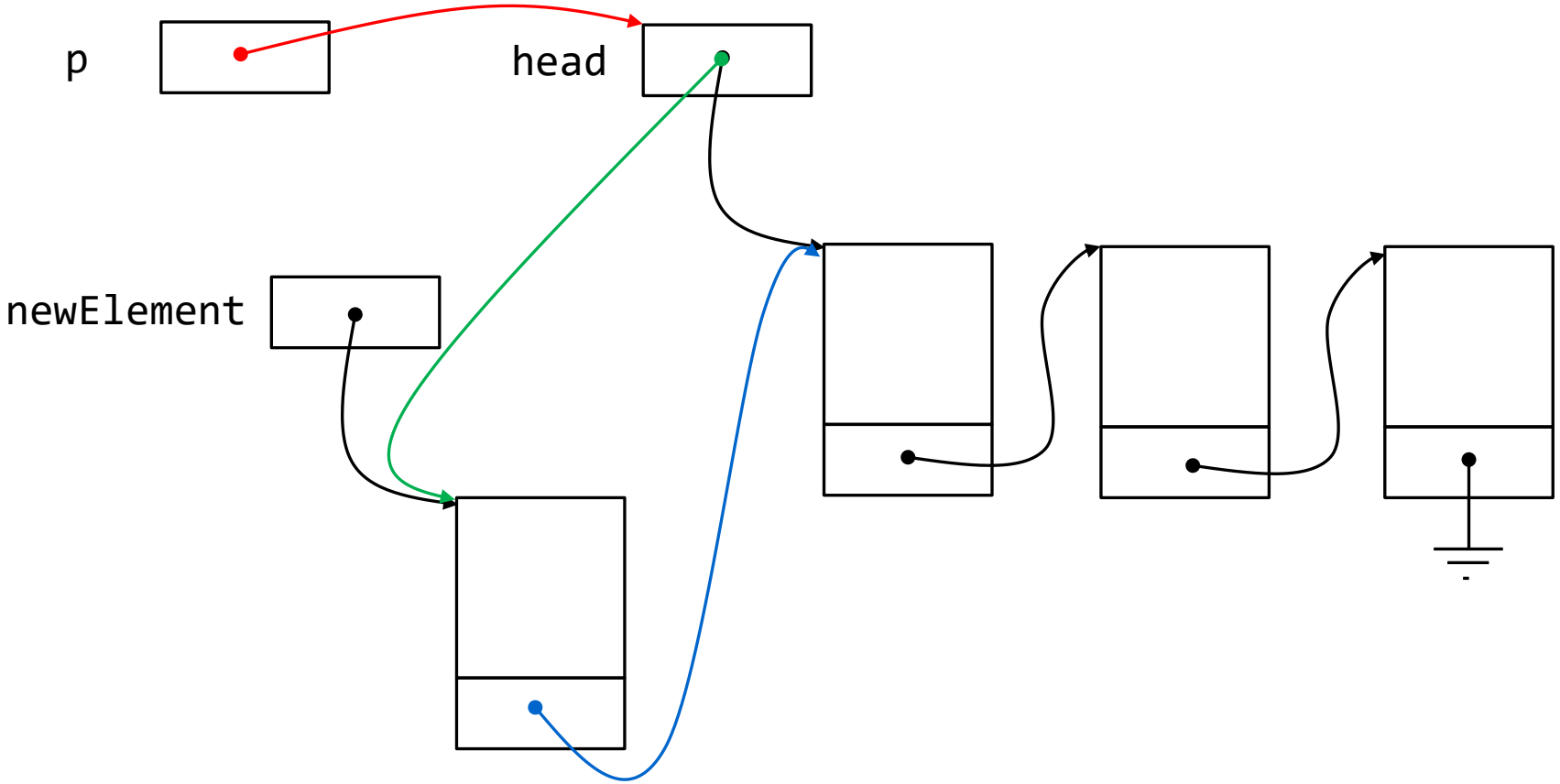


Lista je rekurzivna struktura



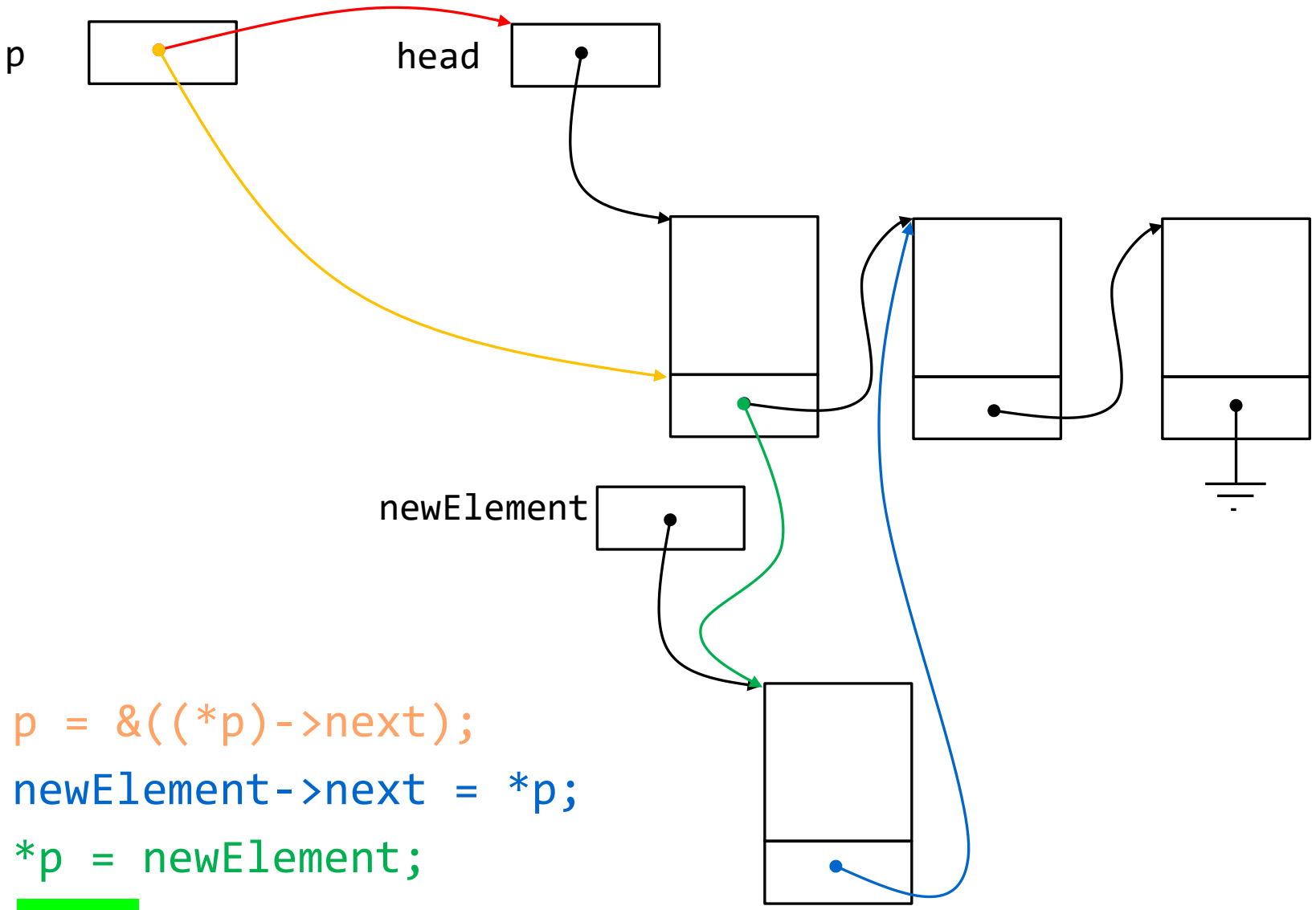
- Pokazivač na sljedeći element je glava ostatka liste

Umetanje na početak liste koristeći **




```
ListElement<T> **p = &head;
newElement->next = *p;
*p = newElement;
```

Umetanje iza elementa liste koristeći **




```
p = &((*p)->next);
newElement->next = *p;
*p = newElement;
```

List.cpp

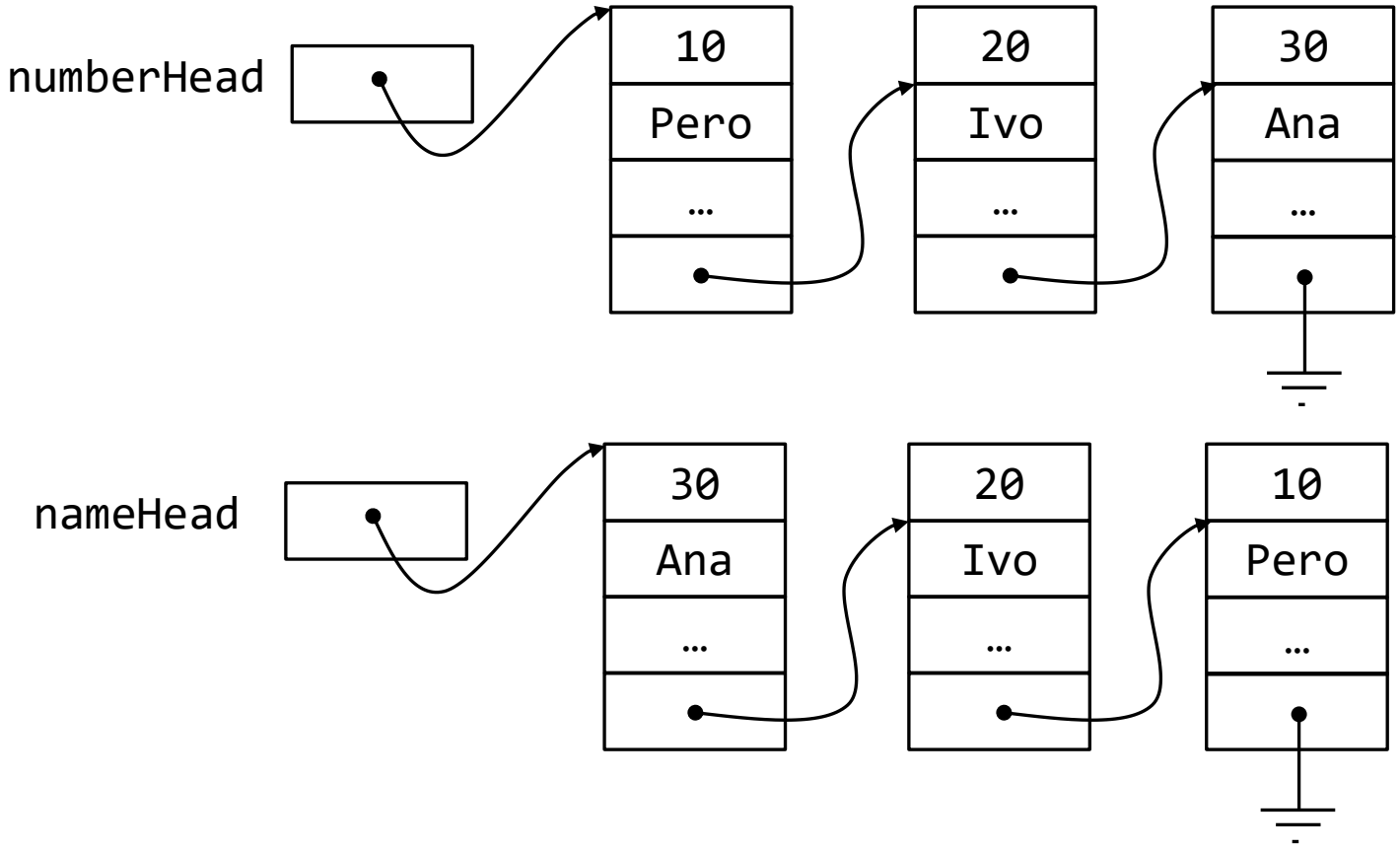
Brisanje iz liste

- Opet dva slučaja
  - Brisanje s početka liste
  - Brisanje iza elemente liste
- Mogu se objediniti koristeći **

List.cpp

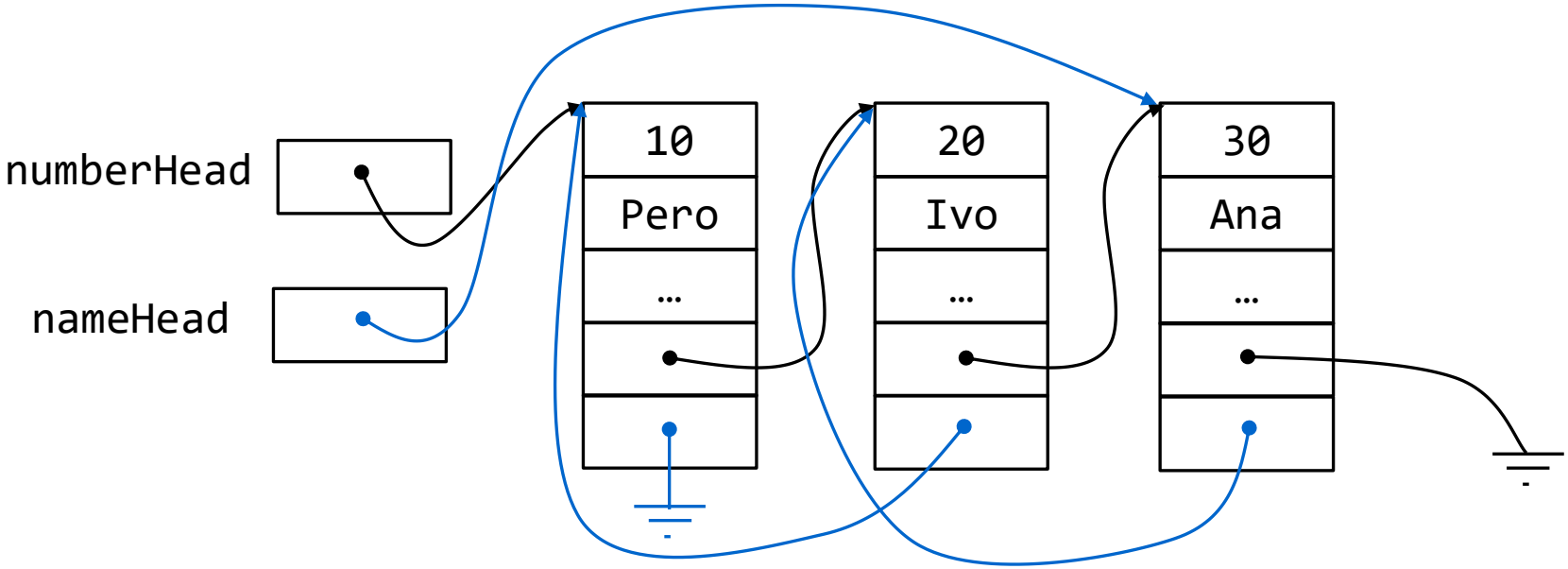
Liste s više ključeva

- Primjer: Podaci o osobi vode se sortirano po matičnom broju ali i imenu
- Dvije liste? Redundancija!!!



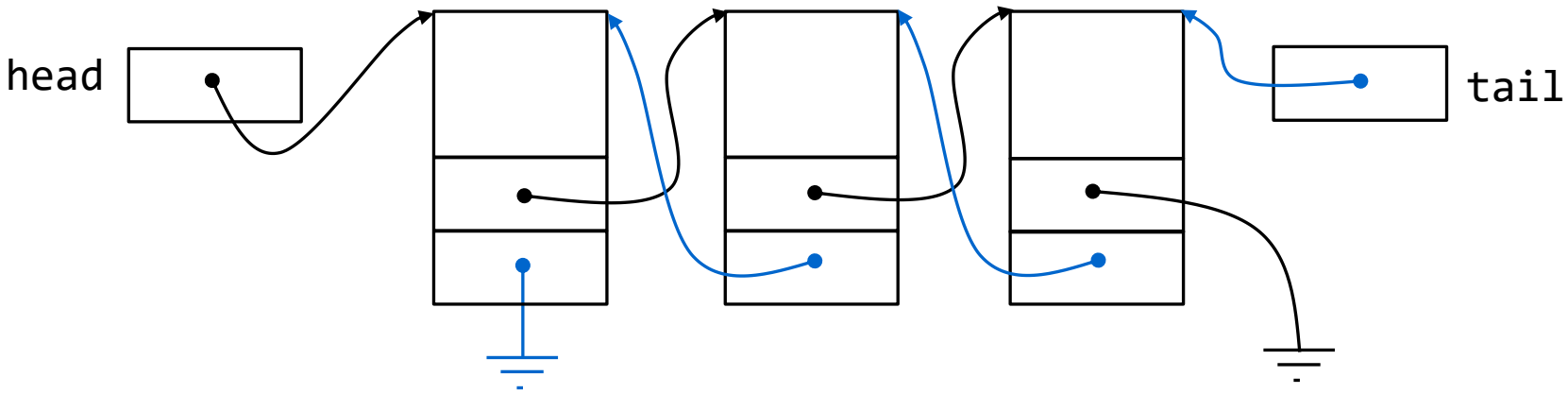
Liste s više ključeva

- Bolje: Dva pokazivača na sljedeći element



ListMultiple.cpp

Dvostruko povezana lista



DoubleLinkedList.cpp

Lista referenci

- U svim prethodnim primjerima podaci u listi su kopija podataka koji su dodani u listu
- Često će biti nužno u listu logički povezati izvorne podatke
- Za to je potrebno promijeniti strukturu elementa liste

```
template <class X> struct ListElement {
   X *item;
   ListElement<X> *next;
};
```

Lista referenci

- Potrebno je promijeniti i prototipove funkcija npr.

```
bool insert(T *data) { // umetanje na početak liste
```

- Poziv je sad

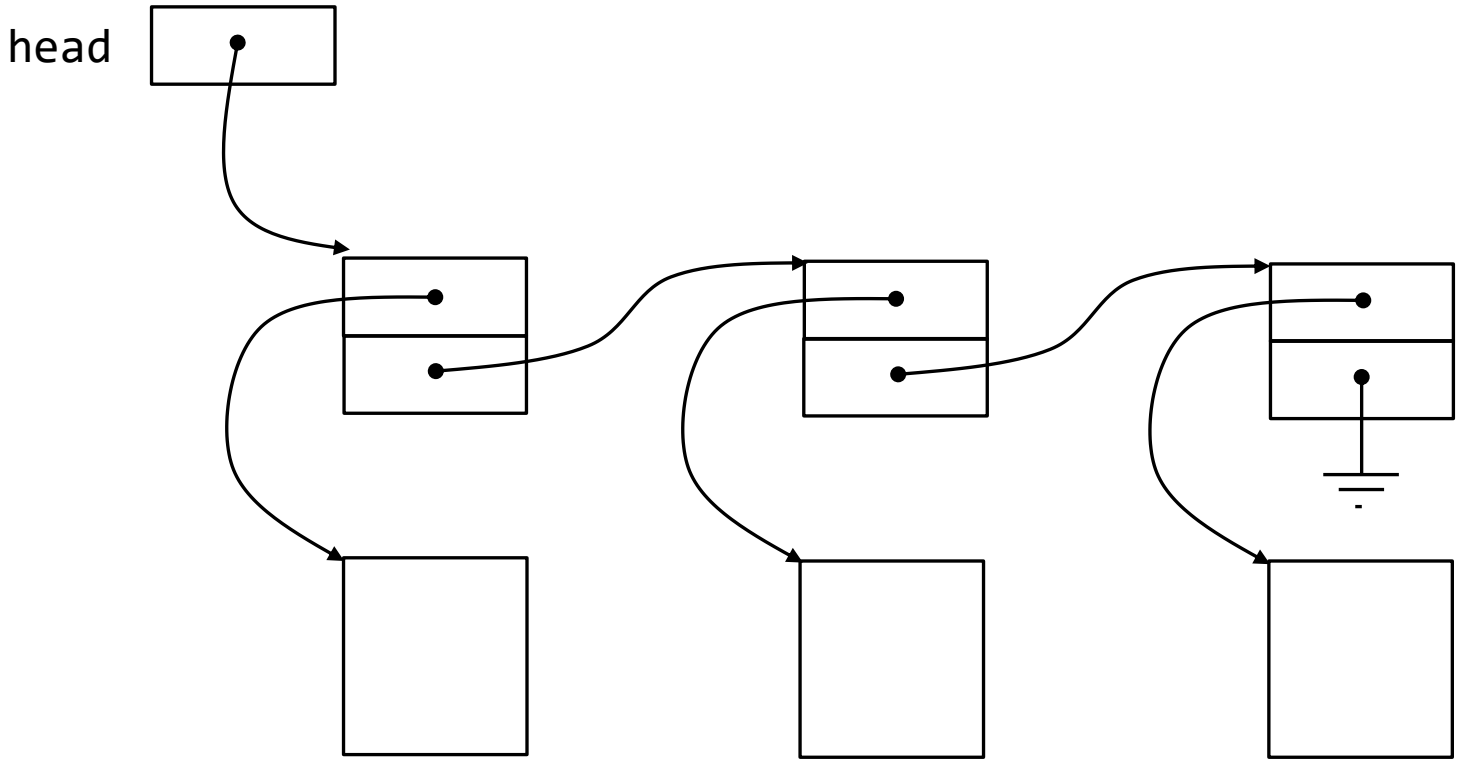
```
T data;
List<T> list;
list.insert(&data);
```

- ili, ako podatak nastaje dinamičkom rezervacijom

```
T *data;
List<T> list;
data = new T;
list.insert(data);
```

Lista referenci

- Stanje je sad sljedeće:



Implementacija iteratora u listi

- Radi uniformnog iteriranja po različitim strukturama podataka, korisno je implementirati *iterator*
- U suštini, to je pokazivač na element liste s konstruktorima:

```
template <class T> class ListIterator {
 private:
   ListElement<T> *ptr = nullptr;
 public:
   ListIterator<T>() : ptr(nullptr) {}
   ListIterator<T>(ListElement<T> *ptr) : ptr(ptr) {}
…
```

Implementacija iteratora u listi

- Moraju se implementirati operatori ++, !=, *:

```
    ListIterator<T> &operator++() { // prefix ++
       if (ptr) ptr = ptr->next;
       return *this;
    }
    ListIterator<T> operator++(int) { // postfix ++
       ListIterator<T> tmp = *this;
       ++(*this);
       return tmp;
    }
    bool operator!=(const ListIterator<T> &other) {
       return ptr != other.ptr;
     }
    T &operator*() { return ptr->data; }
};
```

Implementacija iteratora u listi

- U razredu `List` definira se tip te funkcije `begin` i `end`:

```
template <class T> class List {
   …
   typedef ListIterator<T> iterator;
   iterator begin() { return iterator(head); }
   iterator end() { return iterator(); }
};
```

- U glavnom programu:

```
int main(void) {
   List<int> l; List<int>::iterator i;
   …
   for (List<int>::iterator i; i = l.begin(); i != l.end(); i++) {
      cout << *i << " ";
   }
   ...
```

ListIterator.cpp