Strukture podataka

Skupovi

- Skup (*set*) predstavlja kolekciju neuređenih elemenata
- U računarstvu skupovi su promenljivi u vremenu – dodaju im se i brišu elementi – **dinamički** su.
- Algoritmi zahtevaju nekoliko tipova operacija sa skupovima
  - Dodavanje elemenata
  - Brisanje elemenata
  - Test pripadnosti
  - ...
- Elementi dinamičkih skupova mogu imati pridružene ključeve, dodatne podatke, a takođe im se tipično pristupa preko pokazivača (elementi su često predstavljeni objektima)

Operacije sa skupovima

- Razlikujemo dva skupa operacija:
  1. Upiti nad skupom
     - Search(S,k) – pretraga skupa S po ključu k
     - Minimum(S) – vraća pokazivač na elemenat sa najmanjim ključem
     - Maximum(S) - ... sa najvećim ...
     - Successor(S,x) – vraća pokazivač na sledeći elemenat od x
     - Predecessor(S,x) - ... Prethodni elemenat ...
  2. Operacije izmena skupa
     - Insert(S,x) – dodaje elemenat x u skup S
     - Delete(S,x) – briše elemenat na koji pokazuje x

Elementarne strukture podataka

- Elementarne strukture:
  - Stek (*Stack*)
  - Red (*Queue*)
  - Povezana lista (*List*)
  - Stablo (*Tree*)

- Stek i red su dinamički skupovi gde se elementi uklanjaju unapred određenim redosledom
  - Kod steka Delete briše poslednji (najmlađi) dodat elemenat. Nazivamo ga LIFO procesiranje (*last-in, first-out*)
  - Kod reda Delete briše prvi (najstariji) dodat elemenat. Nazivamo ga FIFO procesiranje (*first-in, first-out*)

Stek

- Operacije:
  - **Push** – dodavanje elementa (insert)
  - **Pop** – preuzimanje (brisanje) elementa (delete)
  - Provera: da li ima elemenata?
- Podaci: niz S[1..S.*top*] sadrži elemente
  - S[1] je elemenat na dnu, a S[S.*top*] je na vrhu steka.
  - Ako je S.*top* == 0, onda je stek prazan
  - Greška je kada se pozove Pop na praznom steku (greška tipa *underflow*)
  - Greška je kada se pozove Push na steku sa *n* elemanata (*overflow*)

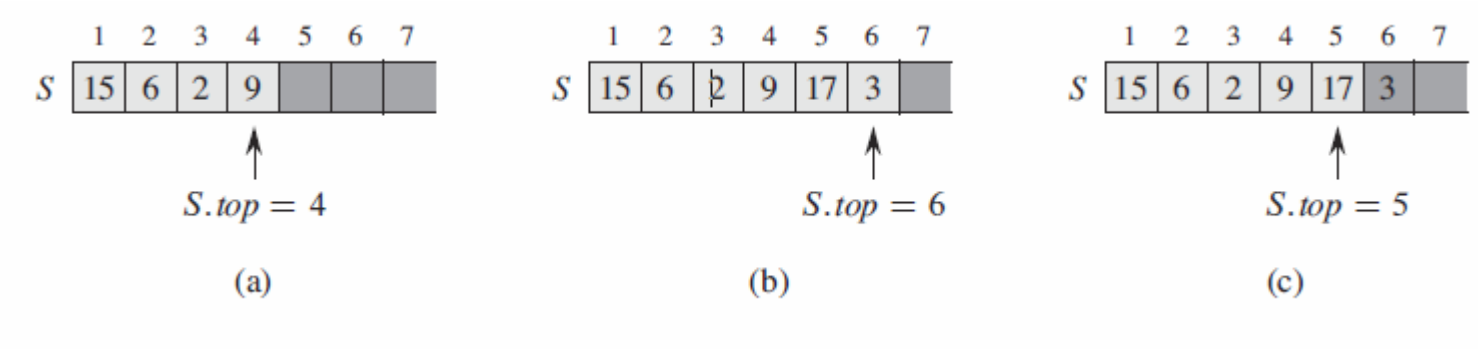
Stek primer

- Primer par operacija sa stekom...

  Push(S,17)

  Push(S,3)

  a = Pop(S)



Stek operacije

- Sve operacije su brze.

```
STACK-EMPTY(S)
1  if S.top == 0
2    return True
3  else return False

PUSH(S,x)
1  S.top = S.top + 1
2  S[S.top]= x

POP(S)
1  if STACK-EMPTY(S)
2    error “underflow”
3  else S.top = S.top – 1
4    return S[S.top + 1]
```

Red (*Queue*)

- Operacije:
  - **Enqueue** – dodavanje elementa (insert)
  - **Dequeue** – preuzimanje (i brisanje) elementa (delete)
  - Provera: da li ima elemenata?
- Podaci:
  - „glava“ (*head*) pokazuje na prvi elemenat reda.
    Preuzima se (dequeue) element sa početka – glave reda.
  - „rep“ (tail) pokazuje na poslednji elemenat reda.
    Dodavanje elementa ga smešta na kraj – rep reda.
- Implementacija (jedan način):
  - niz Q[1..*n*] je prostor za najviše *n*-1 elemenata reda
  - Q se koristi kao kružni bafer

Operacije sa redom

```
ENQUEUE(Q,x)
1  Q[Q.tail] = x
2  if Q.tail==Q.length
3     Q.tail = 1
4  else Q.tail=Q.tail+1
```

```
DEQUEUE(Q,x)
1  x = Q[Q.head]
2  if Q.head==Q.length
3     Q.head = 1
4  else Q.head=Q.head+1
5  return x
```

- U gornjim operacijama nedostaju provere
  - Greška je kada se pozove Dequeue na praznom redu (greška tipa *underflow*)
  - Greška je kada se pozove Enqueue na punom redu (*overflow*)

Primer reda

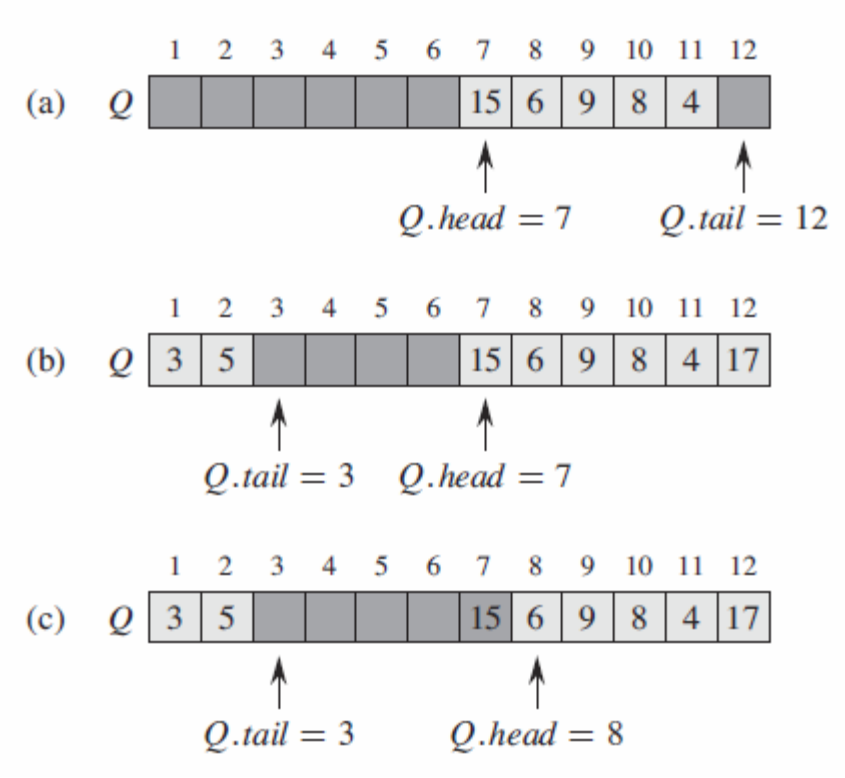
- Primer par operacija sa redom...

  Enqueue(Q,17)

  Enqueue(Q,3)

  Enqueue(Q,5)

  a = Dequeue(Q)



Povezane liste

- Liste su strukture podataka gde su elementi uređeni u linearnom poretku.
- Za razliku od nizova gde je poredak uređen preko indeksa, kod lista imamo pokazivače na susedne elemente.
- Pogodna su za:
  - Česta dodavanja i brisanja elemenata
  - Česta posećivanja elemenata niza – iteriranje kroz listu.
- Nisu pogodne za pretrage

Tipovi lista

- Razlikuju se:
  - **Jednostruko spregnute liste** gde element pokazuje na naredni element.
  - **Dvostruko spregnute liste** gde element pokazuje i na naredni i na prethodni elemenat.
  - Cirkularne liste – elementi su u „prstenu“ (prvi pokazuje na poslednji, a poslednji na prvi)
- Dodatno, lista može biti sortirana na osnovu ključa
  - Prvi elemenat je sa najmanjim ključem, a poslednji sa najvećim

Dvostruko spregnuta lista

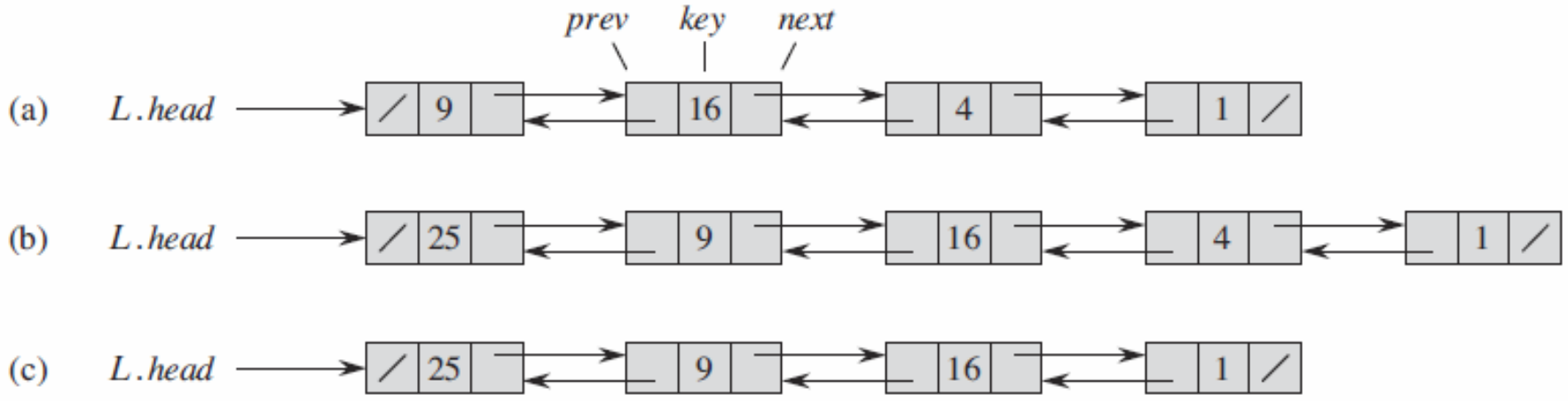
- Svaki elemenat sadrži
  - *next* – pokazivač na naredni elemenat (== Nil kada je poslednji)
  - *prev* – pokazivač na prethodni elemenat (== Nil kada je prvi)
  - *key* – ključ, prisutan kod sortiranih lista
  - drugi podaci (nisu prikazani na slici)
- Pokazivač na prvi elemenat – L.*head*
  - Ako je L.*head* == NIL lista je prazna
- Pokazivač na poslednji elemenat – L.*tail*

Primer rada sa listom

a) lista sadrži 4 elementa sa ključevima {9, 16, 4, 1}

b) Dodat je novi elemenat sa ključem 25

c) Pretraživanjem liste je dobijen pokazivač na elemenat sa ključem 4 i zatim je obrisan.



Jednostruko spregnuta lista

- Jednostavnije je od dvostruko spregnute liste
  - Elementi nemaju *prev* polje koje pokazuje na prethodni elemenat
  - Nema L.*tail*
- Omogućava kretanje od početka (L.*head*) ka narednim elementima.
- Nema mogućnost kretanja ka prethodnom elementu.
  - Jedini način da se dođe do prethodnog elementa je ponovno kretanje od L.*head*.

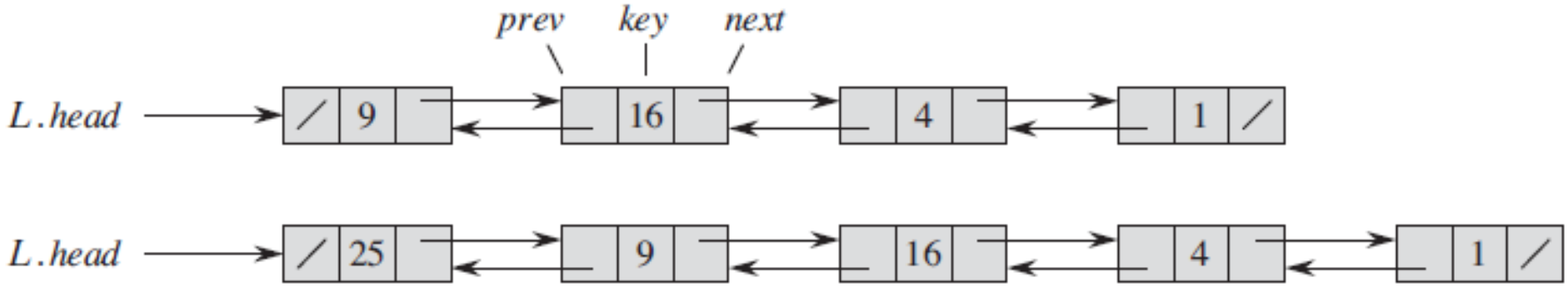
Pretraga u listi

```
LIST-SEARCH(L,k)
1  x = L.head
2  while x ≠ Nil and x.key ≠ k
3     x = x.next
4  return x
```

- Složenost $\Theta(n)$ - u najgorem slučaju traženi elemenat je na kraju liste.

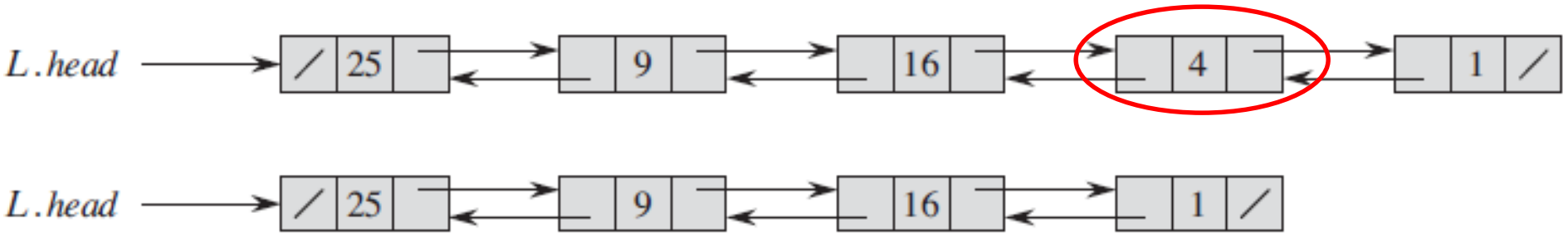
Dodavanje elementa u listu

```
LIST-INSERT(L,x)
1  x.next = L.head
2  if L.head ≠ Nil
3     L.head.prev = x
4  L.head = x
5  x.prev = Nil
```



Brisanje elementa iz liste

```
LIST-DELETE(L,x)
1  if x.prev ≠ Nil
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ Nil
5    x.next.prev = x.prev
```



Upotreba „čuvara“

- Implementacije operacija se pojednostavljuju ako se ignorišu granični uslovi.
  - Podaci se organizuju u cirkularnu listu
  - Dodaje se čuvar (*sentinel*) kao fiktivan (prazan) objekat
  - Dodaje se pokazivač na njega L.nil
  - Poređenje sa NIL se svodi na poređenje sa L.nil

```
LIST-DELETE2(L,x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

```
LIST-INSERT2(L,x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

```
LIST-SEARCH(L,k)
1  x = L.nil.next
2  while x ≠ L.nil
             and x.key ≠ k
3     x = x.next
4  return x
```

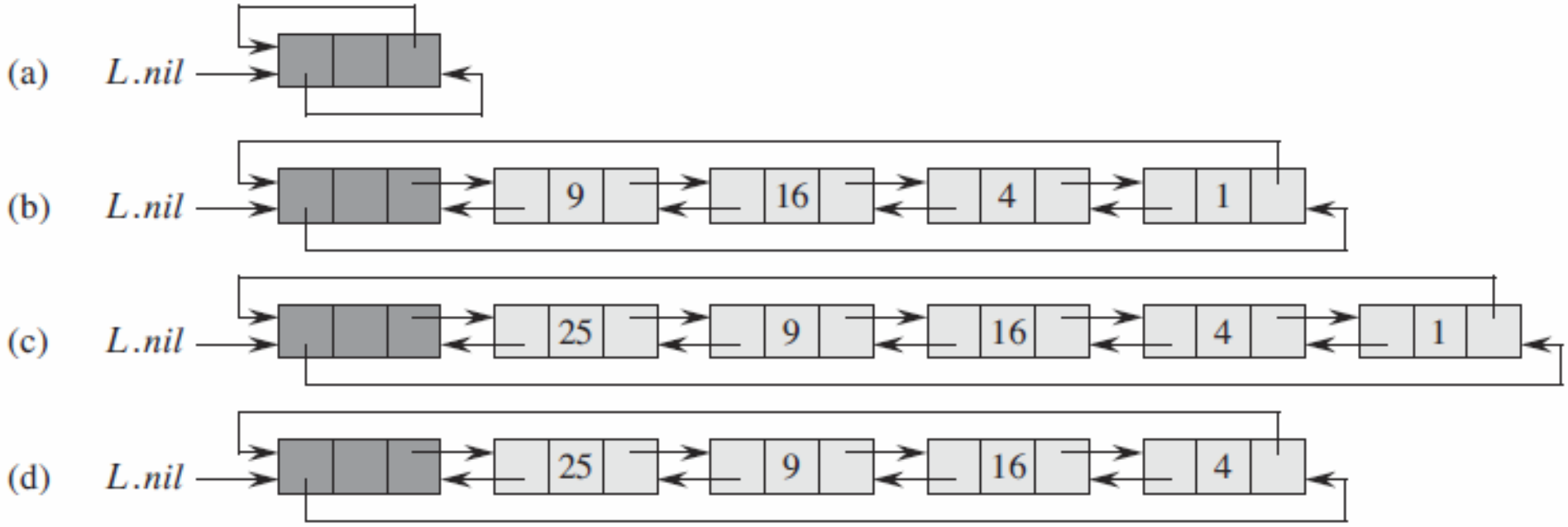
Primer liste sa „čuvarom“

a) Prazna lista (postoji samo „čuvar“)

b) Dvostruko spregnuta lista *L.head=x (x.key=9), L.tail=y* (*y.key=1*)

c) Lista nakon LIST-INSERT2(L,*x*)gde je *x.key* =25 i novi objekat postaje *L.head*

d) Lista nakon LIST-DELETE2(L,*x*)(brisanja objekta sa ključem 1)



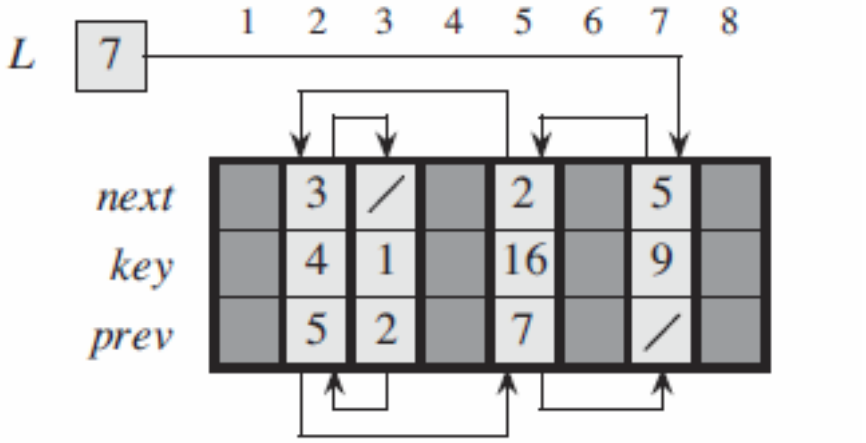
Implementacija pokazivača i objekata

- Kako implementirati pokazivače u programskim jezicima koji ih ne podržavaju?
  Rešenje: Upotreba nizova i indeksiranje elemenata.

- Primer rešenja preko:
  - Niza struktura
  - Nekoliko usklađenih nizova (gde svaki sadrži neki atribut elemenata)
  - Jednog niza (i preračunavanja veličine elementa)

Implementacija liste sa 3 niza

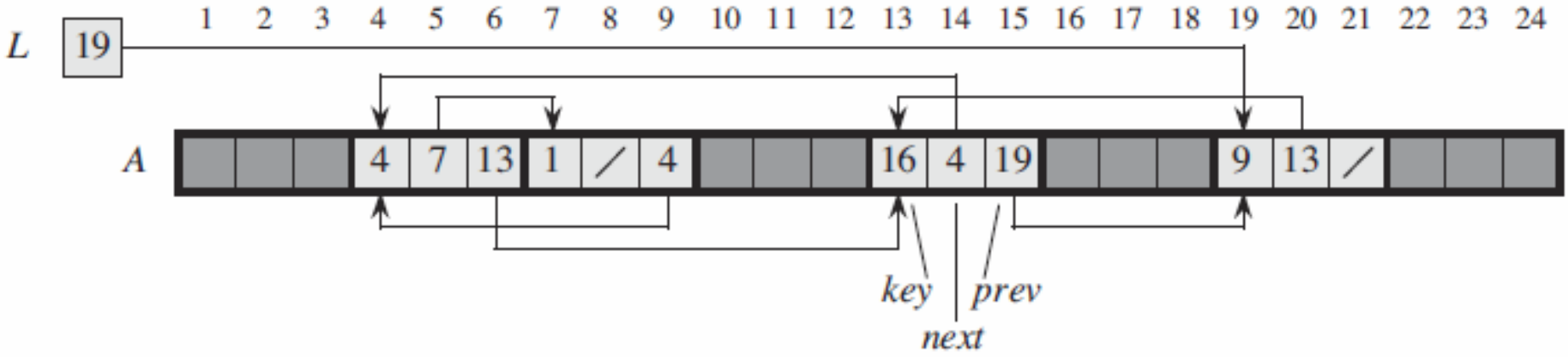
- Primer implementacije liste sadrži tri niza:
  - indeksa prethodnih elemenata
  - vrednosti ključeva
  - indeksa narednih elemenata
- Nizovi su „usklađeni“, jer se na poziciji (indeksu) *i* u svim nizovima nalaze polja jednog elementa.
- Nil vrednost se može predstaviti sa 0 ili -1.
- Primer:



Slična je implementacija upotrebom niza struktura sa poljima: next, key, prev.

Implementacija liste preko jednog niza

- Jedan niz sadrži sve elemente sa indeksima gde se nalaze prethodni i naredni elementi.
- Ograničenje: ključ i drugi korisni podaci elementa se moraju „uklopiti“ sa tipovima podataka upotrebljenim za indekse (tipično celobrojne).
- Primer:



Zauzimanje i oslobađanje elemenata liste

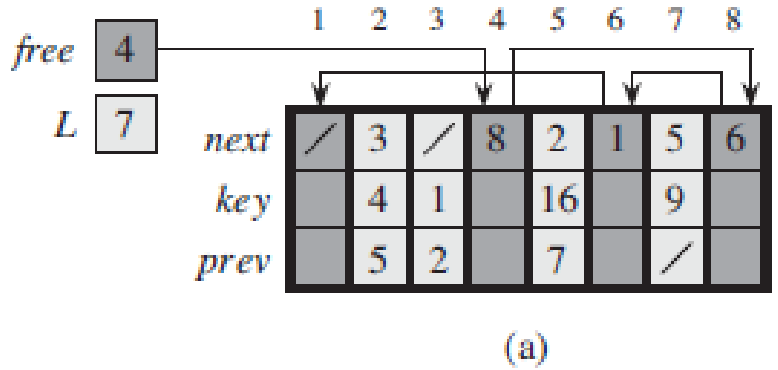
- Prethodne implementacije zahtevaju označavanje neupotrebljenih „slotova“ (potencijalnih mesta) za memorisanje elemenata.
- Pogodno je neupotrebljene „slotove“ povezati u dodatnu listu čiji je početak određen pokazivačem *free*.
- U implementaciji preko više nizova njihova dužina je *n*, a tekući broj zauzetih elemenata je *m* ($m \leq n$).

```
ALLOCATE-ELEMENT()
1  if free == Nil
2     error „out-of-space“
3  else x = free
4     free = x.next
5     return x
```

```
FREE-ELEMENT(x)
1  x.next = free
2  free = x
```

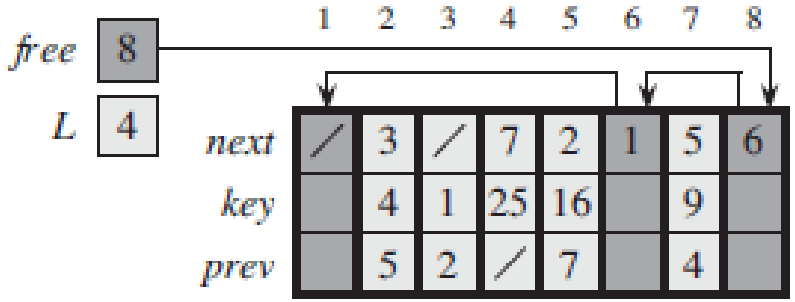
Primer zauzimanja i oslobađanja elemenata

a) Početna lista (svetlija polja) i lista slobodnih članova (tamnija polja)



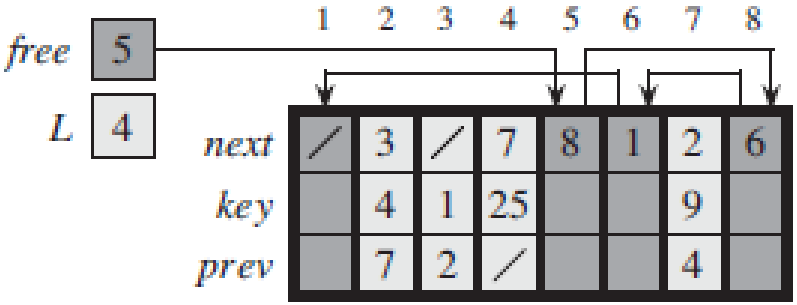
(a)

b) **ALLOCATE-ELEMENT**() (vraća indeks 4), postavljanje key[4] na 25: **LIST-INSERT(L,4)**
new free-list head je 8 (bio je next od prethodnog free-list head (4))



(b)

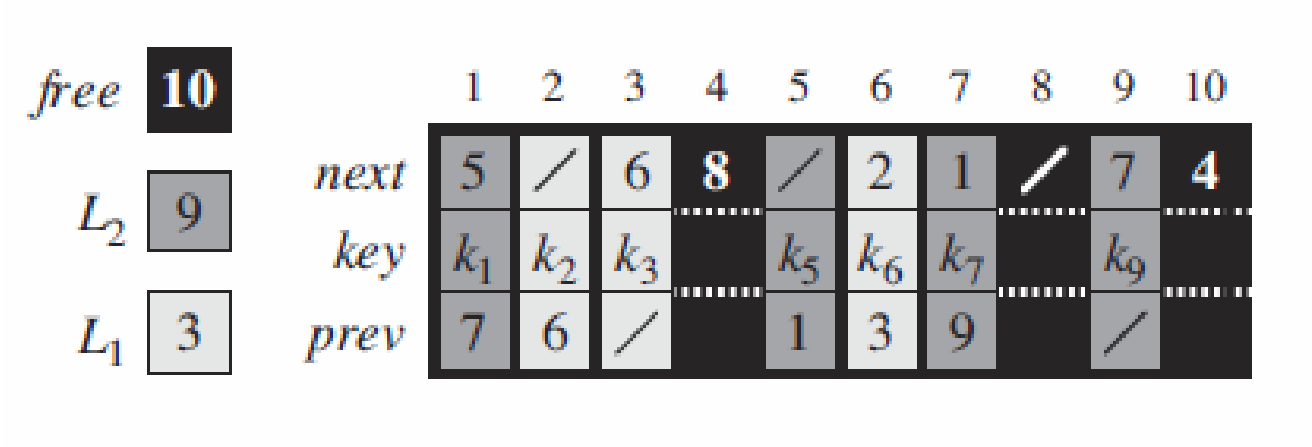
c) Nakon **LIST-DELETE(L,5)**, pozvano je **FREE-ELEMENT(5)**
new free-list head je 5 (njegov next u free-list je 8 (prethodni free-list head))



(c)

Primer 2: tri niza implementiraju nekoliko listi

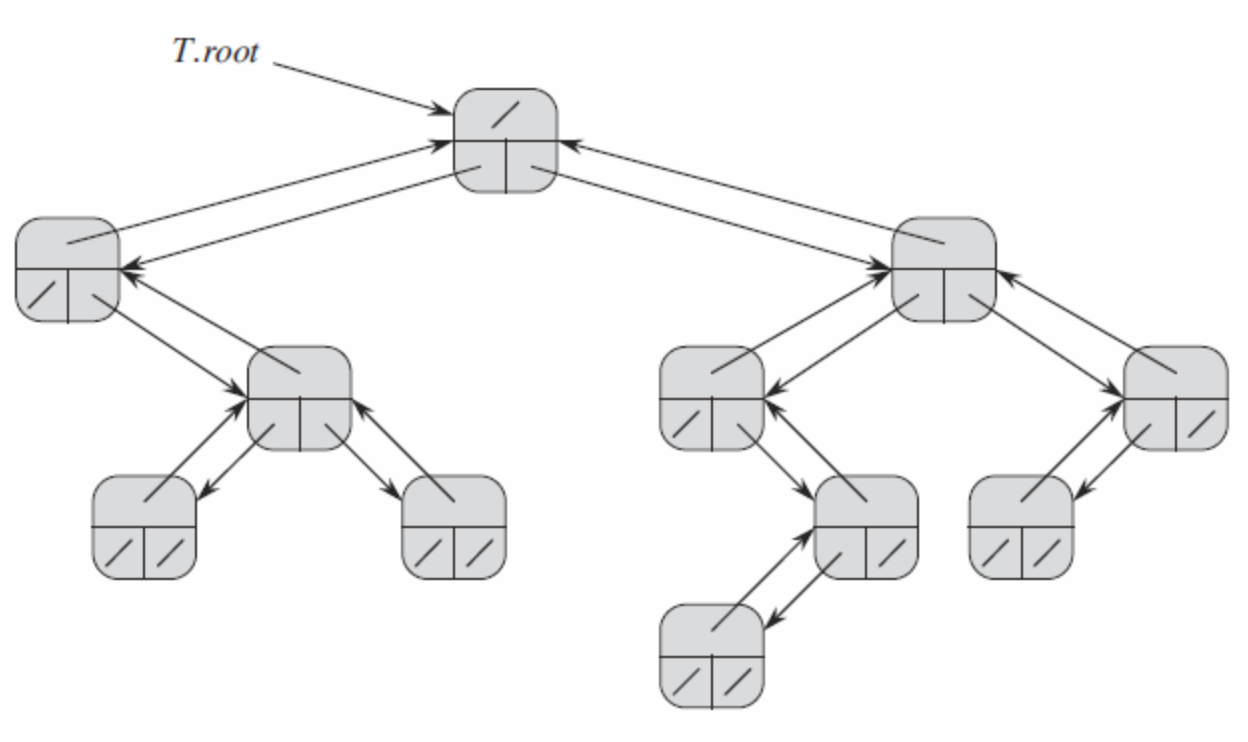
- Dve povezane liste i jedna *free* lista



Binarno stablo

- Svaki elemenat stabla sadrži pokazivače na
  - Roditelja *p,*
  - Levo dete *left,* i
  - Desno dete *right.*
- Element u korenu stabla *x* ima polje *x.p* = NIL
- Element koji nema dece (ili ima samo jedno) polja *left* i/ili *right* postavlja na NIL.
- Struktura binarnog stabla sadrži pokazivač na koren stabla T.*root*
  - Stablo bez elemenata ima T.*root* = NIL

Primer binarnog stabla



Stablo - uopšteno

- Svaki roditelj može imati više dece što se može predstaviti na razne načine.

- Ukoliko uglavnom svi roditelji imaju jednak broj dece onda se elemenat stabla može proširiti poljima: $child_1$, $child_2$, ..., $child_k$ (umesto polja *left* i *right*)

- Ukoliko broj dece varira od elementa do elementa – tada se mogu upotrebiti samo dva pokazivača:
  - na levo (prvo) dete - *left*
  - na brata/sestru - *sibling*

Primer jedne implementacije stabla