Adatszerkezetek és algoritmusok

Horváth Géza

negyedik előadás

Előadások témái

1. Az algoritmusokkal kapcsolatos alapfogalmak bevezetése egyszerű példákon keresztül.
2. Az algoritmusok futási idejének aszimptotikus korlátai.
3. Az adatszerkezetekkel kapcsolatos alapfogalmak. A halmaz, a multihalmaz és a tömb adatszerkezet bemutatása.
4. Az adatszerkezetek folytonos és szétszórt reprezentációja. A verem, a sor és a lista.
5. Táblázatok, önátrendező táblázatok, hash függvények és hash táblák, ütközéskezelés.
6. Fák, bináris fák, bináris keresőfák, bejárás, keresés, beszúrás, törlés.
7. Kiegyensúlyozott bináris keresőfák: AVL fák.
8. Piros-fekete fák.
9. B-fák.
10. Gráfok, bejárás, legrövidebb út megkeresése.
11. Párhuzamos algoritmusok.
12. Eldönthetőség és bonyolultság, a P és az NP problémaosztályok.
13. Lineáris idejű rendezés. Összefoglalás.

A verem, mint absztrakt adattípus

A verem olyan speciális absztrakt adattípus, melyen mindössze két módosító műveletet használhatunk. Ezek a következők:

- PUSH, ami elemek a verem tetejéhez történő **hozzáadására** szolgál, és
- POP, ami a verem tetején elhelyezkedő elem **hozzáférésére** és egyben **eltávolítására** szolgál.

Mivel ezen műveletekkel mindig a verembe legutóljára bekerült elem fog először kikerülni, ezért szokás **LIFO** (last in, first out) adattípusnak is nevezni.

Érdemes megnézni az ábrát a táblán!

A verem, mint adatszerkezet

A verem tulajdonságai:

- homogén
- dinamikus
- szekvenciális
- folytonos reprezentációval ábrázolt

A verem reprezentációja

Legyen S egy olyan verem, mely természetes számokat tartalmaz. Ha tudjuk, hogy legfeljebb n elemet tartalmazhat a verem egy adott időpontban, akkor a legkényelmesebben egy n elemű vektorban tárolhatjuk a vermet: S[1, . . . ,n]. A veremhez tartozik továbbá egy **S.top** érték, mely megmondja, hogy a verem az adott időpontban hány elemet tartalmaz.

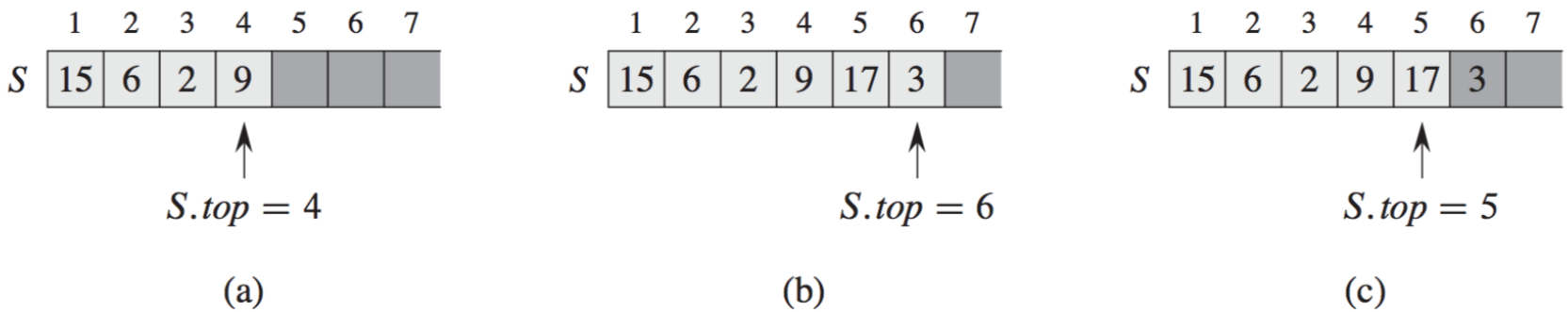


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

A verem reprezentációja

Amennyiben az S.top=0, akkor a verem üres. Ezt felhasználva könnyű létrehozni egy olyan műveletet, amit STACK-EMPTY műveletnek hívunk, és akkor add vssza igaz értéket, ha a verem üres.

A veremhez történő hozzáféréskor két probléma léphet fel.

- Ha a POP műveletet szeretnénk alkalmazni üres verem esetén, – ekkor **alulcsordulásról** beszélünk, – illetve
- ha az S.top érték nagyobbra nő az n értéknél, – ekkor pedig **túlcsordulásról** beszélhetünk.

A most ismertetésre kerülő pszeudokód nem kezeli a túlcsordulást.

A verem műveletek megvalósítása

Minden verem művelet leírható pár soros kóddal:

```
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]
```

A verem, mint adatszerkezet

Műveletek:

- adatszerkezetek **létrehozása**: folytonos reprezentációval
- adatszerkezetek **módosítása**
  - elem hozáadása: PUSH
  - elem törlése: POP
  - elem cseréje: nincs
- elem **elérése**: POP (majd vissza lehet tenni PUSH-al)

A sor, mint absztrakt adattípus

A sor olyan speciális absztrakt adattípus, melyen mindössze két módosító műveletet használhatunk. Ezek a következők:

- ENQUEUE, ami elemeknek a sor végére történő **beszúrására** szolgál, és
- DEQUEUE, ami a sor elején lévő elemhez történő **hozzáférésére** és egyben az adott elem **eltávolítására** szolgál.

Mivel ezen műveletek alkalmazásával mindig a sorba hamarabb bekerült elem fog hamarabb kikerülni, ezért szokás a sort **FIFO** (first in, first out) adattípusnak is nevezni.
A sor esetén két természetes számot szükséges nyilvántartani. Az egyik a **fej**, a másik a **farok**. Amikor az elem bekerül a sorba, akkor mindig a farok által megadott pozícióba kerül, míg törléskor mindig a fej által megadott pozícióban lévő elem fog törlődni.

A sor, mint adatszerkezet

A sor tulajdonságai:

- homogén
- dinamikus
- szekvenciális
- folytonos reprezentációval ábrázolt

A sor reprezentációja

Alapvetően 3 különböző megközelítés van a sor folytonos reprezentációját illetően.

- A naív megközelítés – nem hatékony,
- a sétáló sor – valamivel jobb, (különösen nagyobb memória esetén,)
- a ciklikus sor – ez a leghatékonyabb, így a legelterjedtebb.

Az első kettőt a táblán ismertetjük.

A sor reprezentációja – ciklikus sor

Alapvetően 3 különböző megközelítés van a sor folytonos reprezentécióját illetően.

- A naív megközelítés – nem hatékony,
- a sétáló sor – valamivel jobb, (különösen nagyobb memória esetén,)
- a ciklikus sor – ez a legelterjedtebb.

Egy n elemű Q vektorban (Q[1,...,n]) egy legfeljebb n-1 elemű ciklikus sor ábrázolható a **Q.head** és a **Q.tail** értékek segítségével, a következők szerint.

- Ha Q.head=Q.tail, akkor a sor üres.
- A sor létrehozásakor Q.head=Q.tail=1.

A sor reprezentációja – ciklikus sor

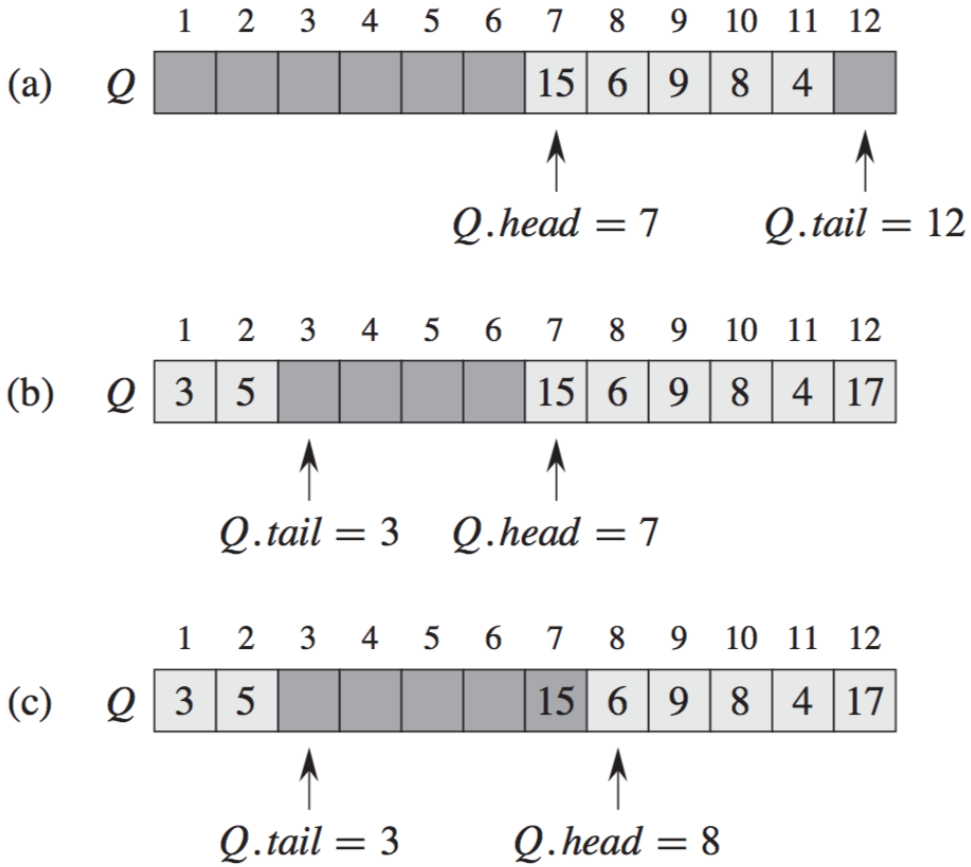


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

A sor reprezentációja – ciklikus sor

A sor használata során az alábbi problémák léphetnek fel:

- ha a DEQUEUE utasítást használjuk üres sor esetén, akkor **alulcsordulásról** beszélünk, illetve
- ha a Q.head=Q.tail+1, azaz tele van a sor, és az ENQUEUE műveletet használjuk, akkor **túlcsordulásról** beszélünk.

Az ENQUEUE és DEQUEUE műveletek pszeudokódjának ismertetésekor az egyszerűség kedvéért eltekintünk mindkét fentebbi hiba lekezelésétől. (Ugyanakkor megjegyezzük, hogy az algoritmusok megfelelő módosításával mindkét hiba előfordulása hatékonyan kezelhető.)

A sor műveletek megvalósítása

Az ENQUEUE és DEQUEUE műveletek pszeudokódjának ismertetésekor az egyszerűség kedvéért eltekintünk mindkét fentebbi hiba lekezelésétől.

```
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)
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
```

A sor, mint adatszerkezet

Műveletek:

- adatszerkezetek **létrehozása**: folytonos reprezentációval
- adatszerkezetek **módosítása**
  - elem hozáadása: ENQUEUE
  - elem törlése: DEQUEUE
  - elem cseréje: nincs
- elem **elérése**: DEQUEUE

A láncolt lista, mint absztrakt adattípus

Láncolt lista alkalmazásakor a lista elemei a lista fizikai tárolása során nem követik a listában szereplő sorrendet, hanem szétszórva helyezkednek el a memóriában. Az elemek összetettek, az adott elem értékén túl tartalmaznak egy mutatót is, mely a következő elem memóriabeli helyét tárolja. Tehát a láncolt lista minden eleme két részből áll:

- **érték,** és
- **mutató,** ami egy memóriacímet tárol.

A láncolt lista alkalmazásához feltétlen szükséges egy **head** mutató is, ami a lista első elemének memóriabeli címét tartalmazza.
Ha a head=NIL, akkor a lista üres.

A láncolt lista, mint adatszerkezet

A láncolt lista tulajdonságai:

- homogén
- dinamikus
- szekvenciális
- szétszórt (láncolt) reprezentációval ábrázolt

Láncolt lista típusok

Sokféle láncolt lista készíthető.

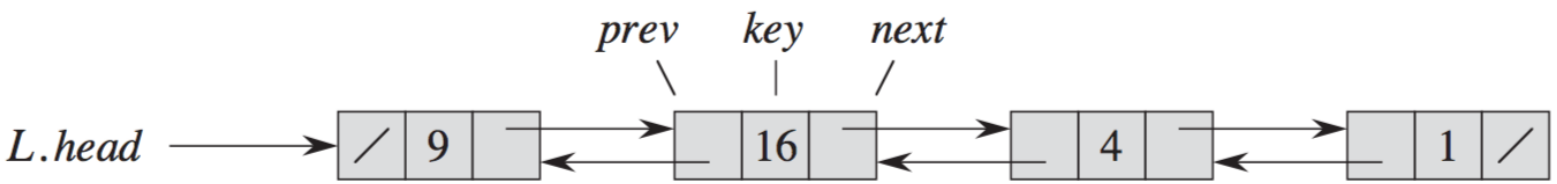
Néhány példa:

- egyirányban láncolt lista
- kétirányban láncolt lista
- ciklikus lista
- multilista

Lássuk a rajzokat a táblán!

A kétirányban láncolt lista

- A kétirányban láncolt lista minden eleme az adott elem értékén túl tartalmaz még két mutatót is: next és prev.
- Adott x listaelem esetén az x.next mutató a rákövetkező elemre, míg az x.prev mutató a megelőző elemre mutat.
- Az L.head a lista első elemére mutat.
- Ha az x.prev=NIL, akkor x a lista első eleme, míg ha az x.next=NIL, akkor x a lista utolsó eleme.



A kétirányban láncolt lista műveletei

A LIST-SEARCH(L,k) művelet megkeresi a k érték első előfordulását az L listában, lineáris keresés használatával. Amennyiben a k érték szerepel a lista elemei között, akkor a visszatérési érték a k értéket tartalmazó elemre mutató mutató, egyébként pedig NIL.

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

A kétirányban láncolt lista műveletei

A LIST-INSERT(L,x) művelet beszúrja az adott x elemet az L lista legelső pozíciójába.

```
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
```

A kétirányban láncolt lista műveletei

A LIST-DELETE(L,x) művelet eltávolitja az x elemet az L listából. Az x elem megadása a x elemre történő mutató megadásával történik. Amennyiben egy adott k értéket szeretnénk törölni, akkor először meg kell keresni az adott k érték memóriacímét a LIST-SEARCH(L,k) művelettel.

```
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
```

A kétirányban láncolt lista, mint adatszerkezet

Műveletek:

- adatszerkezetek **létrehozása**: szétszórt (láncolt) reprezentációval
- adatszerkezetek **módosítása**
  - elem hozáadása: List-Insert
  - elem törlése: List-Delete
  - elem cseréje: nincs (ebben a reprezentációban)
- elem **elérése**: List-Search

Irodalomjegyzék