

Programiranje 2

10. predavanje

Saša Singer

singer@math.hr

web.math.hr/~singer

PMF – Matematički odjel, Zagreb

Sadržaj predavanja

- Vezane liste (nastavak):
 - Ubacivanje na pravo mjesto. “Insertion Sort”.
 - Izbacivanje traženog elementa.
 - Složenije operacije na listama.
 - Spajanje (konkatenacija) dvije liste.
 - Sortirano spajanje dvije liste — “merge”.
 - Sortiranje liste. “MergeSort”.

Informacije

Termin drugog kolokvija je:

- četvrtak, 3. 7., u 12 sati.

Kako sad stvari stoje, popravni kolokvij je u “jesen”:

- srijeda, 10. 9., u 12 sati.

Praktični kolokvij (treći i zadnji krug) — termini su:

- subota, 31. 5., praktikum 2 — 3 grupe, start u 13 sati.

Prijave još mogu — na “mojim” vratima!

Napomena: od prvog popravka PK i nadalje, nema

- odbijanja zadataka s matricama, ili uvodjenja svojih granica na dimenzije nizova i matrica.

Postoji i dinamička alokacija!

Informacije (nastavak)

Na službenom webu, pod materijalima, od nedavno postoji i

- službeni podsjetnik = “šalabahter”,
- tj., kratki opis svih **bitnih** elemenata C-a.

Bitno: Smijete ga koristiti i na kolokviju!

- Zato se se “priviknite” na njega.

Ali, nemojte ga nositi na kolokvij — to je **zabranjeno!**

- Dobit ćete ga na kolokviju — to se “dijeli” kao poklon (umjesto sendviča, kave, i sl.)

Razlog: Nećemo “obrađene” šalce — s “osobnim dodacima”.

Vezane liste (nastavak)

Sadržaj

- Vezane liste (nastavak):
 - Ubacivanje na pravo mjesto. “Insertion Sort”.
 - Izbacivanje traženog elementa.
 - Složenije operacije na listama.
 - Spajanje (konkatenacija) dvije liste.
 - Sortirano spajanje dvije liste — “merge”.
 - Sortiranje liste. “MergeSort”.
 - Složenost MergeSorta.

Sortirano ubacivanje elementa u listu

Primjer. Imamo vezanu listu cijelih brojeva, zadanu pokazivačem **prvi** na prvi element. Dodatno,

- prepostavljamo da je lista **uzlazno** sortirana, od početka prema kraju liste.

U tu listu treba **ubaciti** jedan element, zadan pokazivačem **novi**, ali tako da

- **nova** (“povećana”) lista, također, bude **uzlazno** sortirana.

Dogovor: kao i uvijek, polazna lista smije biti **prazna**!

Drugim riječima, zadani element treba

- **ubaciti** na njegovo “**pravo**” mjesto u listi.

Sortirano ubacivanje elementa u listu (nastavak)

Posao koji treba napraviti ima **dva** dijela:

- prvo treba **pronaći** to “**pravo**” mjesto na koje treba ubaciti element (“traženje po listi”);
- zatim treba **ubaciti** zadani element na to **mjesto**.

Ova **dva** dijela još treba korektno “**povezati**”. Dakle:

- Što treba “**pronaći**” u **prvom** dijelu,
- da bismo korektno “**ubacili**” element u **drugom** dijelu?

Bez obzira na to **gdje** je “**pravo**” mjesto elementa u listi,

- za sva **ubacivanja** u listu **uvijek** koristimo isti **princip**:
 - “ubaci **iza** nekog” — kao u funkciji **ubaci_iza!**

Sortirano ubacivanje elementa u listu (nastavak)

Ubacivanje elementa u listu ima dvije operacije — i treba ih napraviti točno tim redom. Prvo:

- novom elementu treba postaviti sljedećeg,
- a zatim:
- prethodnom elementu treba promijeniti sljedećeg na novog, a
 - ako prethodnog nema, onda pokazivač **prvi** treba postaviti na novog.

A to znači da traženje pravog mesta u listi mora naći

- pokazivač **preth** na “prethodni” element — onaj iza kojeg ubacujemo novog.

Sortirano ubacivanje elementa u listu (nastavak)

S druge strane, zato što je lista **uzlazno** sortirana, **traženje** “pravog” mesta ide tako da:

- preskačemo sve elemente čiji sadržaj je (strogo) manji od sadržaja **novog**, a
- stajemo na prvom elementu čiji sadržaj je **veći ili jednak** od sadržaja **novog** (ako takav **postoji**).

Sad **oprez!** Tako nalazimo element **ispred** kojeg treba ubaciti novog. Zato koristimo **dva** pokazivača:

- **pom** — na element čiji **sadržaj testiramo** i
- **preth** — na element **ispred** tog.

Napomena: **pom** možemo i izbaciti! Po logici stvari, uvijek vrijedi **pom = preth->sljed**, ali se lakše čita kad imamo oba.

Funkcija sortirano_ubaci

```
lista sortirano_ubaci(lista prvi, lista novi)
{
    /* Insertion sort. */
    /* Ne provjerava novi != NULL. */

    lista preth, pom;
    int trazi;

    if (prvi == NULL) return novi;
```

Funkcija sortirano_ubaci — nastavak

```
pom = prvi;  
preth = NULL;      /* Nije nuzno. */  
  
while (pom != NULL && pom->broj < novi->broj) {  
    preth = pom;  
    pom = pom->sljed;  
}  
}
```

Funkcija sortirano_ubaci — nastavak

```
/* Ovo ispod je:  
     ubaci_iza(prvi, preth, novi); */  
/* To radi i za prvi == NULL. */  
if (pom == prvi) {  
    novi->sljed = prvi;  
    prvi = novi;  
}  
else {  
    novi->sljed = preth->sljed;  
    preth->sljed = novi;  
}  
  
return prvi;  
}
```

Sortiranje ubacivanjem — Insertion Sort

Sortirano ubacivanje jednog elementa može se iskoristiti i kao algoritam za sortiranje niza podataka.

Sortiranje ubacivanjem ili Insertion Sort:

- Na početku je sortirana lista prazna.
- Nakon toga, sortirano ubacujemo jedan po jedan element, s tim da svaki element sadrži po jedan podatak iz niza kojeg treba sortirati.
- Postupak ubacivanja ponavljamo sve dok ne ubacimo sve takve elemente.

Uočite da je upravo vezana lista “zgodna” struktura za realizaciju ovog algoritma (za razliku od polja).

Insertion Sort (nastavak)

U tom algoritmu, **nije** bitno kako **nastaju** pojedini **elementi** sortirane liste.

- Možemo redom čitati **brojeve** (sadržaje) i **kreirati** pripadne elemente — jedan po jedan, ili
- startati s postojećem **nesortiranom** listom elemenata, iz koje **izbacujemo** jedan po jedan element, na pr. s **početka**.

U programu **1_5.c** dan je primjer takvog sortiranja učitanog niza — kreiranjem element po element.

Napomena. Složenost ovog algoritma je $O(n^2)$. U praksi se koristi samo za **vrlo kratke** liste. Postoji i puno **bolji** algoritam za sortiranje liste (v. **MergeSort**, malo kasnije).

Zadatak. Napravite **Insertion Sort** zadano niza na **polju**.

Izbacivanje traženog elementa iz liste

Primjer. Imamo vezanu listu cijelih brojeva, zadanu pokazivačem **prvi** na prvi element.

- Iz te liste treba **obrisati** prvi element s **parnim** brojem (kao sadržajem).
- Ako takvog elementa **nema**, lista se **ne mijenja**.

Dogovor: Polazna lista smije biti **prazna**!

Opet, bez obzira na to **gdje** se **nalazi** “traženi” element u listi,

- za sva **izbacivanja** ili **brisanja** iz liste **uvijek** koristimo isti **princip**:
 - “izbaci ili obriši **iza** nekog” — kao u funkciji **obrisi_iza**!

Izbacivanje traženog elementa iz liste (nast.)

Izbacivanje elementa iz liste ima dvije operacije.

- Prvo treba naći traženi element u listi (onaj kojeg treba izbaciti).

Zatim, ako ga nađemo:

- prethodnom elementu treba promijeniti sljedećeg na onog iza traženog, a
- ako prethodnog nema, onda pokazivač prvi treba postaviti na onog iza traženog.

Dakle, traženje tog elementa u listi mora naći

- pokazivač preth na “prethodni” element — onaj iza kojeg izbacujemo traženog.

Izbacivanje traženog elementa iz liste (nast.)

Kao i kod ubacivanja, koristimo dva pokazivača:

- `pom` — na element čiji sadržaj testiramo i
- `preth` — na element ispred tog.

Napomena: Zbog `pom = preth->sljed`, opet `pom` možemo izbaciti.

Naravno, kao i uvijek, još treba paziti na

- praznu listu i
- izbacivanje s početka — kao u funkciji `obrisi_iza`.

Provjerite da ranije uočeni nedostaci funkcije `obrisi_iza`

- `prvi == NULL` i `preth->sljed == NULL`

ovdje `nisu` bitni — tj. da ne mogu dovesti do greške!

Funkcija obrisi_prvi_parni

```
lista obrisi_prvi_parni(lista prvi)
{
    /* Brise prvi parni element u listi.
       Ako ga nema - ne radi nista. */

    lista preth, pom;

    if (prvi == NULL) return NULL;      /* Ne treba. */
```

Funkcija obrisi_prvi_parni — nastavak

```
pom = prvi;
preth = NULL;      /* Ovdje je nuzno. */

while (pom != NULL && pom->broj % 2 != 0) {
    preth = pom;
    pom = pom->sljed;
}
```

Funkcija obrisi_prvi_parni — nastavak

```
/* Test jesmo li nasli parnog! */

if (pom != NULL) {
    /* Ovo ispod je:
       obrisi_iza(prvi, preth); */
    /* To radi i za preth == NULL. */
    if (preth == NULL) {
        pom = prvi;
        prvi = prvi->sljed;
    }
    else {
        pom = preth->sljed;      /* Ne treba. */
        preth->sljed = pom->sljed;
    }
}
```

Funkcija obrisi_prvi_parni — nastavak

```
    free(pom);
        /* Ne treba pom = NULL; */
    }

    return prvi;
}
```

U programu **1_6.c** kreiramo listu

- $2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow \text{NULL}$.

a zatim iz nje **6** puta brišemo prvi parni element. Zadnje
brisanje ne mijenja listu. Završna lista je

- $5 \rightarrow 9 \rightarrow \text{NULL}$.

Zadaci na temu — ubaci/izbaci

Zadatak. Napišite funkciju `izbaci_iza` sa zaglavljem

```
lista izbaci_iza(lista prvi, lista preth,
                  lista *p_izbacen);
```

Funkcija treba:

- iz liste zadane **pokazivačem prvi** na prvi element,
- **izbaciti** element koji se nalazi odmah **iza** elementa na kojeg pokazuje **preth**.

Za razliku od funkcije `obrisi_iza`, **izbačeni** element se **ne briše**, već treba

- **vratiti** pokazivač na njega — kroz **varijabilni** argument `*p_izbacen` (slično kao u funkciji `ubaci_na_kraj`).

Zadaci na temu — ubaci/izbac (nastavak)

Vrijednost funkcije je (kao i inače):

- pokazivač na prvi element dobivene liste.

Napišite funkciju tako da radi “korektno” u svim mogućim slučajevima:

- `prvi == NULL` — ne radi ništa, samo vraća `NULL` i `*p_izbacen = NULL`,
- `preth == NULL` — izbacuje prvi element liste,
- `preth == zadnji`, tj. `preth->sljed == NULL` — ne izbacuje ništa i uredno vraća `*p_izbacen = NULL`.

Zadaci na temu — ubaci/izbaci (nastavak)

Zadatak. Vezana lista **brojeva** zadana je **pokazivačem** **prvi** na prvi element. Napišite funkciju koja **rastavlja** tu listu u dvije liste, tako da

- **prva** lista sadrži samo elemente s **parnim** sadržajem, a
- **druga** lista sadrži samo elemente s **neparnim** sadržajem iz polazne liste.

Funkcija treba **vratiti** **pokazivače** na te **dvije** liste kroz varijabilne argumente.

Rastav liste treba napraviti

- samo **promjenama** **veza** (pokazivača).

Zabranjeno je mijenjati **sadržaje** elemenata!

Zadaci na temu — ubaci/izbaci (nastavak)

Varijacije. Relativni poredak elemenata u dobivenim listama:

- smije biti **bilo koji**,
- **mora** biti **isti** kao u polaznoj listi.

Zadaci na temu — preuredi listu

Zadatak. Vezana lista **brojeva** zadana je **pokazivačem prvi** na prvi element. Napišite funkciju koja **preuređuje** tu listu, tako da

- na **početku** liste budu svi elementi s **parnim** sadržajem, a
- na **kraju** liste (iza svih parnih) budu svi elementi s **neparnim** sadržajem iz polazne liste.

Funkcija treba **vratiti pokazivač** na **prvi** element **preuređene** liste.

Preuređenje liste treba napraviti

- samo **promjenama veza** (pokazivača).

Zabranjeno je mijenjati **sadržaje** elemenata!

Zadaci na temu — preuređi listu (nastavak)

Varijacije. Relativni poredak elemenata u dobivenim listama:

- smije biti **bilo koji**,
- **mora** biti **isti** kao u polaznoj listi.

Unutar funkcije, smijete koristiti

- rastav liste u **dvije** liste,
- spajanje dvije liste — konkatenacija (v. sljedeći primjer).

Probajte zadatak riješiti **bez** toga, tj. “**unutar**” samo jedne liste!

Zadaci na temu — preuredi listu (nastavak)

Zadatak. Vezana lista **brojeva** zadana je **pokazivačem prvi** na prvi element. Napišite funkciju **okreni_listu**

```
lista okreni_listu(lista prvi);
```

koja preuređuje tu listu, tako da

- okreće naopako, tj. **invertira** poređak elemenata u listi.

Funkcija treba **vratiti** pokazivač na **prvi** element
“preokrenute” liste.

Invertiranje liste treba napraviti

- samo **promjenama veza** (pokazivača).

Zabranjeno je mijenjati **sadržaje** elemenata!

Spajanje (konkatenacija) dvije liste

Primjer. Imamo dvije vezane liste cijelih brojeva, zadane pokazivačima `prvi_1` i `prvi_2` na prvi element odgovarajuće liste. Te dvije liste treba spojiti u jednu listu, tako da

- druga lista bude **iza** prve (poredak elemenata u pojedinoj listi se **ne mijenja**).

Ova operacija se još zove i **konkatenacija** — po analogiji s konkatenacijom **dva stringa** (funkcija `strcat`).

Dogovor: kao i uvijek, obje polazne liste smiju biti **prazne!**

Drugim riječima, u **spojenoj** listi

- sljedbenik **zadnjeg** elementa prve liste mora biti **prvi** element druge liste.

Spajanje dvije liste (nastavak)

Posao koji treba napraviti ima **dva** dijela:

- prvo treba pronaći **zadnji** element u prvoj listi (kao u funkciji **trazi_zadnji**);
- zatim treba postaviti njegovog **sljedbenika** na **prvi** element **druge** liste.

Funkcija spoji_dvije

```
lista spoji_dvije(lista prvi_1, lista prvi_2)
{
    lista pom = prvi_1;

    if (prvi_1 == NULL) return prvi_2;

    for ( ; pom->sljed != NULL; pom = pom->sljed);

    pom->sljed = prvi_2;

    return prvi_1;
}
```

Funkcija spoji_dvije_rek

Konkatenaciju možemo napraviti i **rekurzivnom** funkcijom:

```
lista spoji_dvije_rek(lista prvi_1, lista prvi_2)
{
    if (prvi_1 == NULL) return prvi_2;

    if (prvi_1->sljed == NULL)
        prvi_1->sljed = prvi_2;
    else
        spoji_dvije_rek(prvi_1->sljed, prvi_2);
        /* Ne trebamo vrijednost funkcije! */

    return prvi_1;
}
```

Spajanje dvije liste — demo-program

Program `1_7.c` kreira dvije liste

- 1 → 3 → 5 → 7 → NULL,
- 2 → 4 → 6 → 8 → NULL,

a zatim ih spaja — konkatenira. Dobivena lista je

- 1 → 3 → 5 → 7 → 2 → 4 → 6 → 8 → NULL.

Program `1_7r.c` radi to isto, ali

- sve funkcije za rad s listama su **rekurzivne!**

Pitanje: Zašto to je ili **nije** dobro?

Kad imamo dvije već **sortirane** liste, kao u ovom primjeru, onda ih **nema** smisla nesortirano spajati (konkatenirati).

- Kako ih treba **spojiti** da dobijemo **sortiranu** listu?

Sortirano spajanje (merge) dvije sortirane liste

Primjer. Imamo dvije vezane liste cijelih brojeva, zadane pokazivačima `prvi_1` i `prvi_2` na prvi element odgovarajuće liste. Dodatno, prepostavljamo da su

- obje liste već **uzlazno sortirane**, od početka prema kraju liste.

Te dvije liste treba **spojiti** u jednu listu, ali tako da

- spojena lista, također, bude **uzlazno sortirana**.

Ova operacija se još zove i **sortirano spajanje** (engl. **merge**).

Dogovor: kao i uvijek, obje polazne liste smiju biti **prazne!**

Merge dvije sortirane liste (nastavak)

Skica algoritma.

- Ako je barem jedna od dvije liste **prazna**, rezultat je ona **druga** lista (može i ona biti **prazna**).

U protivnom, **obje** liste su **neprazne**.

Ideja za nastavak algoritma — “pametni” **Insertion Sort**:

- Na početku je **spojena** lista **prazna**.
- Zatim “gradimo” **spojenu** listu — element po element, tako da stalno bude uzlazno **sortirana**.

Za to nam **ne treba** opći algoritam za **sortirano ubacivanje** (funkcija **sortirano_ubaci**). Tu **iskoristimo** pretpostavku da su polazne liste već uzlazno **sortirane** — i to je “**pametno**”!

Merge dvije sortirane liste (nastavak)

Zato što su liste **uzlazno sortirane**,

- **najmanji** element u svakoj u njih je baš **prvi** element.
- Usporedimo te **prve** elemente i **manji** od njih je sigurno **najmanji** element u **obje** liste.
- Izbacimo ga iz odgovarajuće liste — s **početka**,
- i **ubacimo** ga **na kraj** spojene liste!

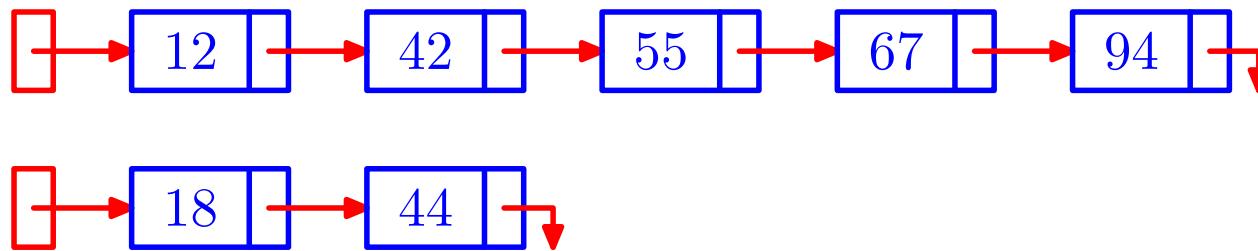
Ovo ponavljamo sve dok su **obje** liste **neprazne**.

Na kraju, kad ostanemo s jednom “**nepotrošenom**” listom,

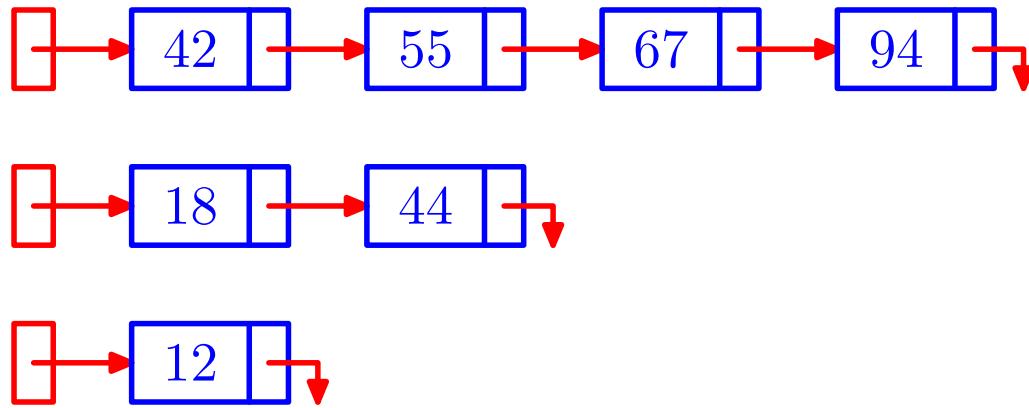
- **spojimo tu** listu **na kraj** dotad spojene liste (konkatenacija).

I nije tako komplikirano!

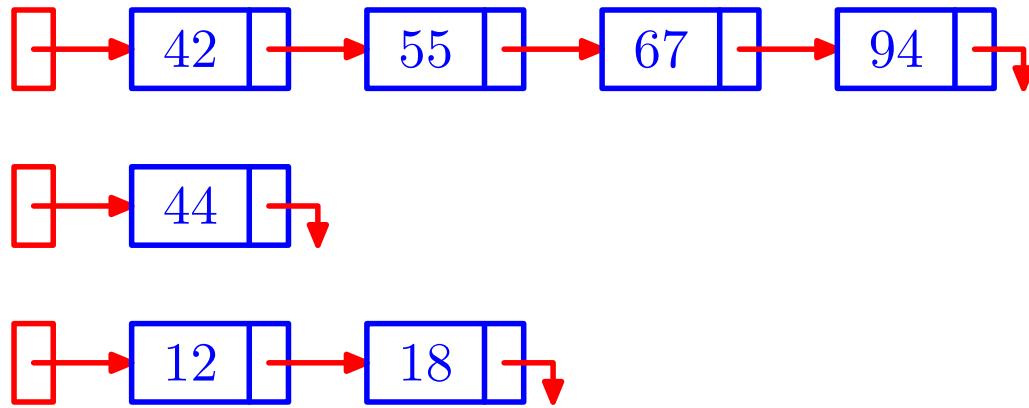
Merge dvije sortirane liste — primjer



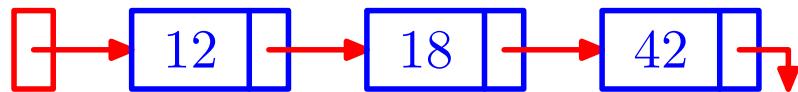
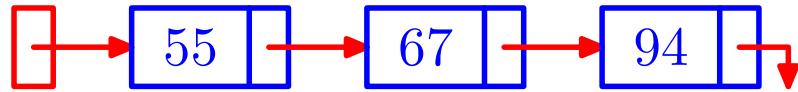
Merge dvije sortirane liste — primjer



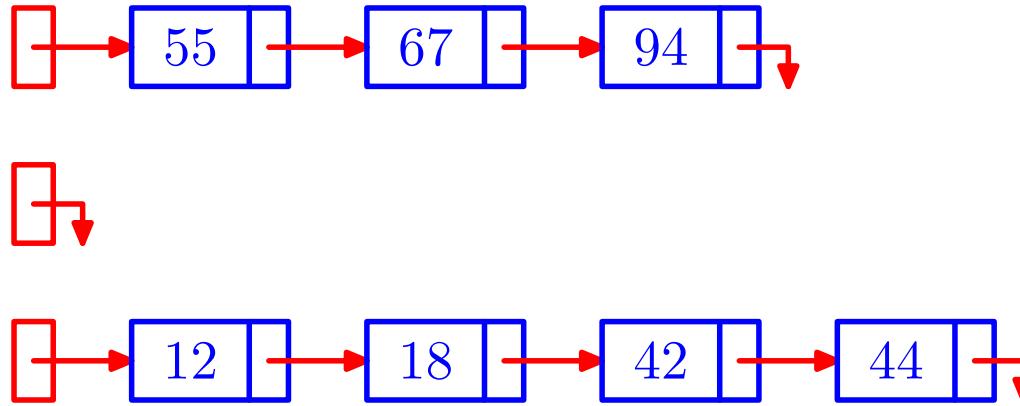
Merge dvije sortirane liste — primjer



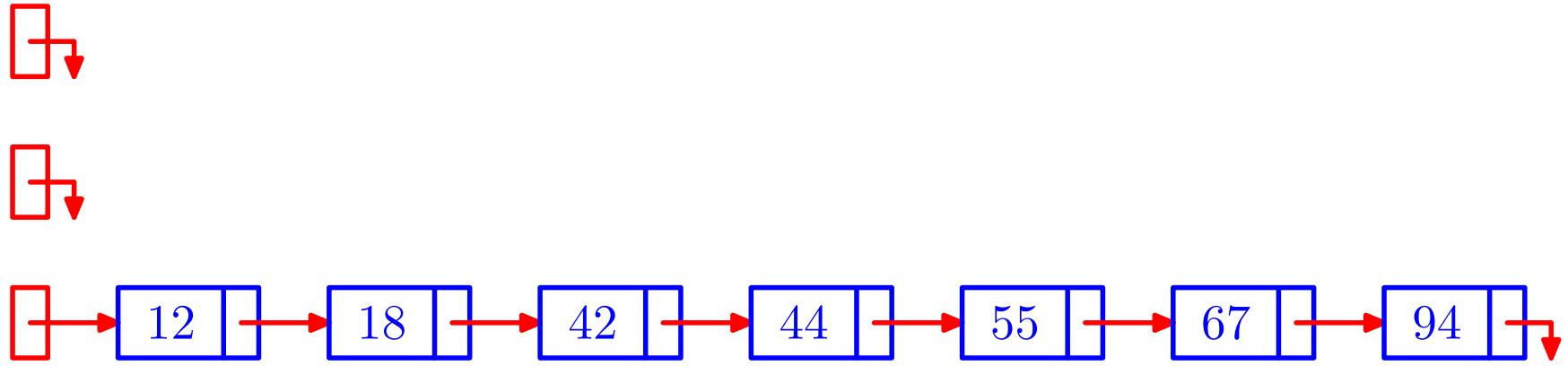
Merge dvije sortirane liste — primjer



Merge dvije sortirane liste — primjer



Merge dvije sortirane liste — primjer



Funkcija merge

```
lista merge(lista prvi_1, lista prvi_2)
{
    /* Sortirano spaja dvije liste (merge). */

    lista prvi = NULL, zadnji, pom;

    if (prvi_1 == NULL) return prvi_2;
    if (prvi_2 == NULL) return prvi_1;
```

Funkcija merge — nastavak

```
while (prvi_1 != NULL && prvi_2 != NULL) {
    if (prvi_1->broj <= prvi_2->broj) {
        pom = prvi_1;
        prvi_1 = prvi_1->sljed;
    }
    else {
        pom = prvi_2;
        prvi_2 = prvi_2->sljed;
    }
}
```

Funkcija merge — nastavak

```
    if (prvi == NULL)
        prvi = pom;
    else
        zadnji->sljed = pom;
    zadnji = pom;
}

if (prvi_1 == NULL)
    zadnji->sljed = prvi_2;
if (prvi_2 == NULL)
    zadnji->sljed = prvi_1;

return prvi;
}
```

Merge dvije sortirane liste — demo-program

Program `1_8.c` kreira iste dvije liste kao u prošlom primjeru

- `1 -> 3 -> 5 -> 7 -> NULL,`
- `2 -> 4 -> 6 -> 8 -> NULL,`

a zatim ih sortirano spaja. Dobivena lista je

- `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> NULL.`

Na kraju, program još okreće ovu listu (`okreni_listu`).

Program `1_8r.c` radi to isto, s tim da je

- funkcija `merge` implementirana rekurzivno i zove se `merge_rek`,
- a rekurzivne su i sve ostale funkcije za rad s listama.

Pitanje: Zašto to je ili nije dobro?

Sortiranje spajanjem — MergeSort

Primjer. Imamo jednu vezanu listu cijelih brojeva, zadanu pokazivačem **prvi** na prvi element liste.

- Tu listu treba **uzlazno sortirati** — po sadržaju elemenata, od početka prema kraju liste, i
- **vratiti** pokazivač na početak **sortirane** liste.

Dogovor: polazna lista smije biti **prazna!**

Sortiranje liste treba napraviti

- samo **promjenama veza** — pokazivača.

Zabranjeno je mijenjati **sadržaje** elemenata!

Sortiranje spajanjem — MergeSort (nastavak)

Za sortiranje liste koristimo MergeSort algoritam. To je **rekurzivni** algoritam za sortiranje niza podataka

- baziran na operaciji merge — sortiranom spajanju već sortiranih nizova podataka.

U našem primjeru je niz podataka spremljen u obliku liste.

Osnovna ideja MergeSort algoritma:

- Podijeli nesortirani niz podataka na dva dijela (podniza) — podjednake “duljine”.
- Rekursivno sortiraj svaki od nastalih podnizova.
- Sortirano spoji (merge) ta dva već sortirana podniza u jedan sortirani niz.

Sortiranje spajanjem — MergeSort (nastavak)

Rekurzivno “raspolavljanje” i sortiranje radimo sve dok ne dobijemo “trivijalan” niz:

- prazan ili jednočlan.

Ovakvi nizovi su “već sortirani”.

Dakle, ako je ulazna lista **prazna** ili **jednočlana**

- **nema** rekurzije, već samo vraćamo (ulazni) pokazivač na tu listu.

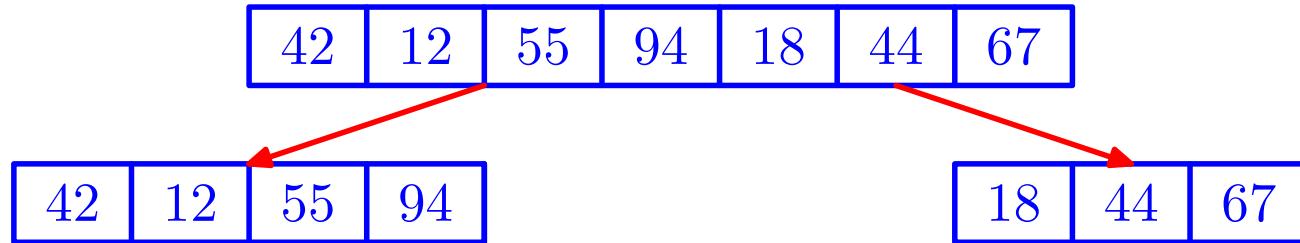
Uočite da je **vezana lista**

- “idealna” struktura za **realizaciju MergeSort algoritma**, za razliku od **polja**. Što je **problem** kod **polja**?

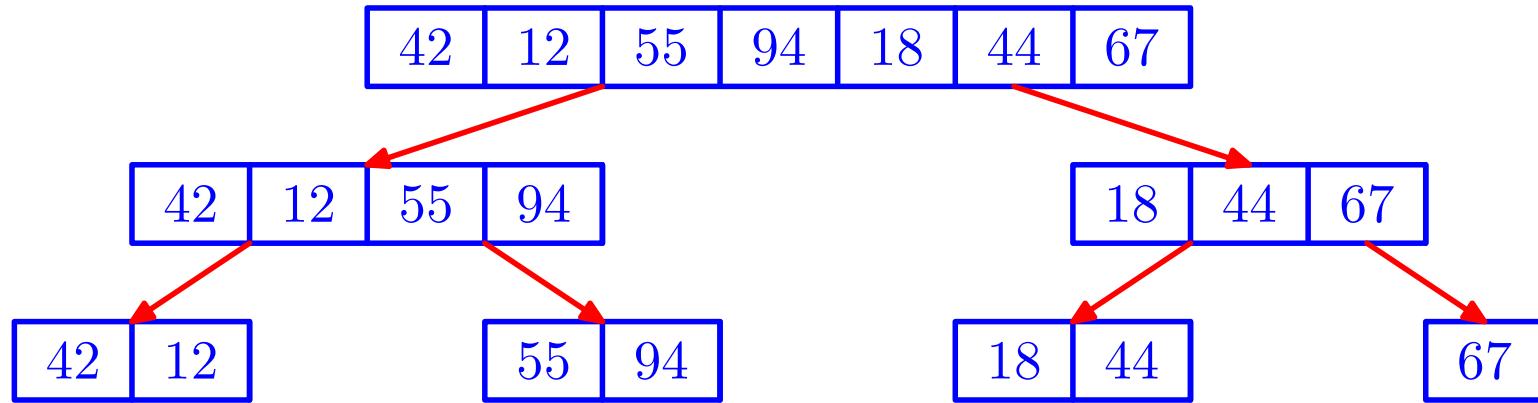
MergeSort — primjer

42	12	55	94	18	44	67
----	----	----	----	----	----	----

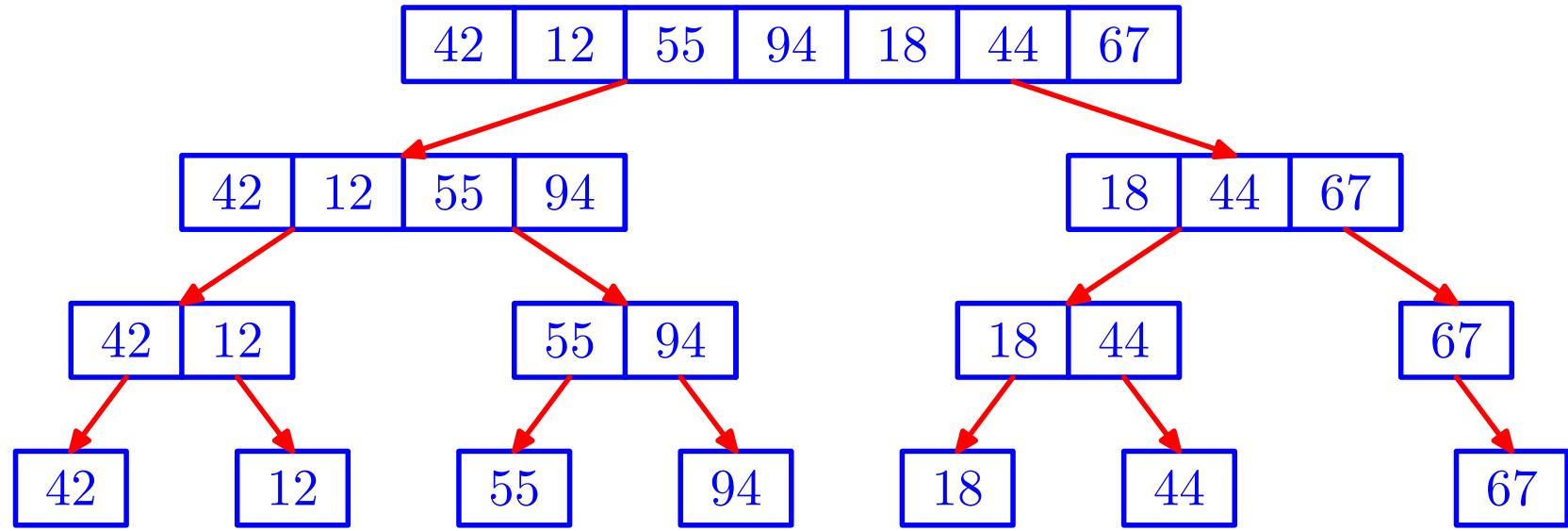
MergeSort — primjer



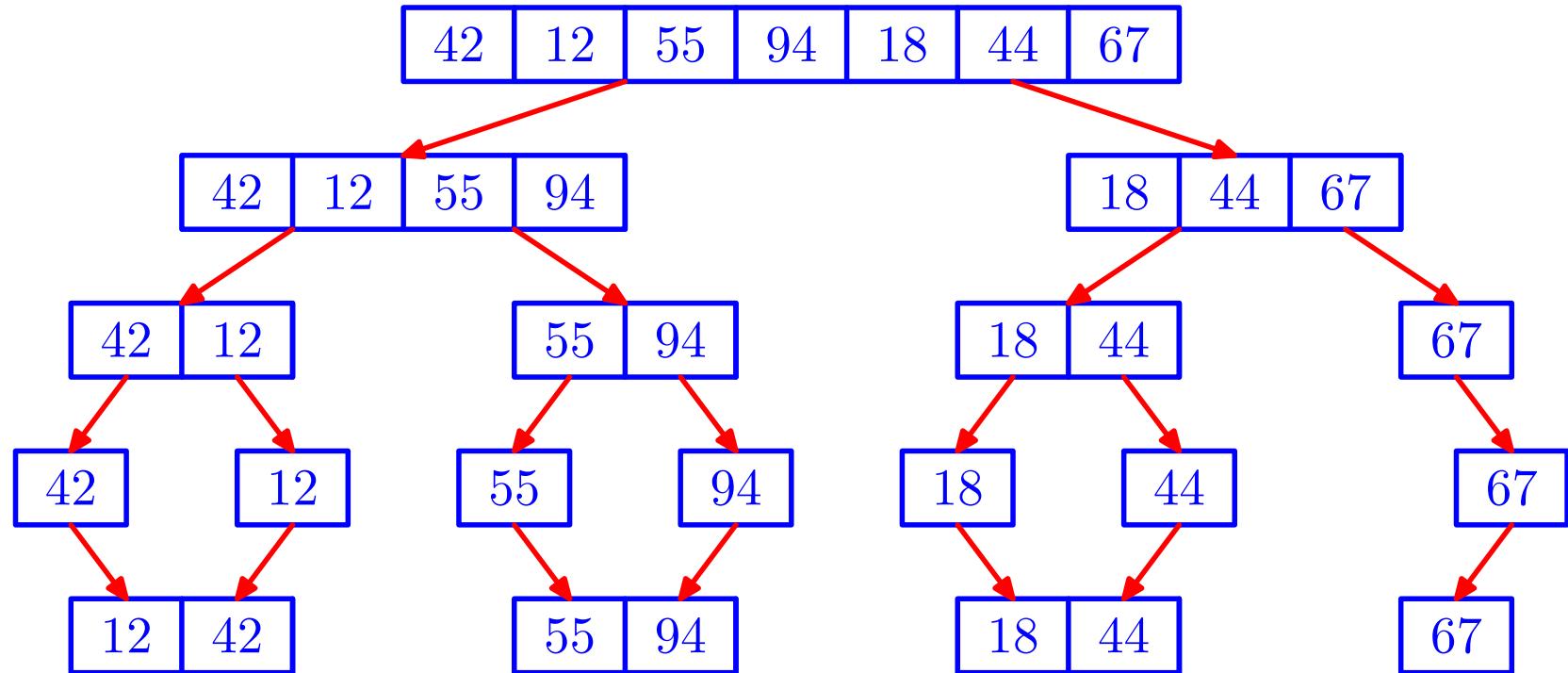
MergeSort — primjer



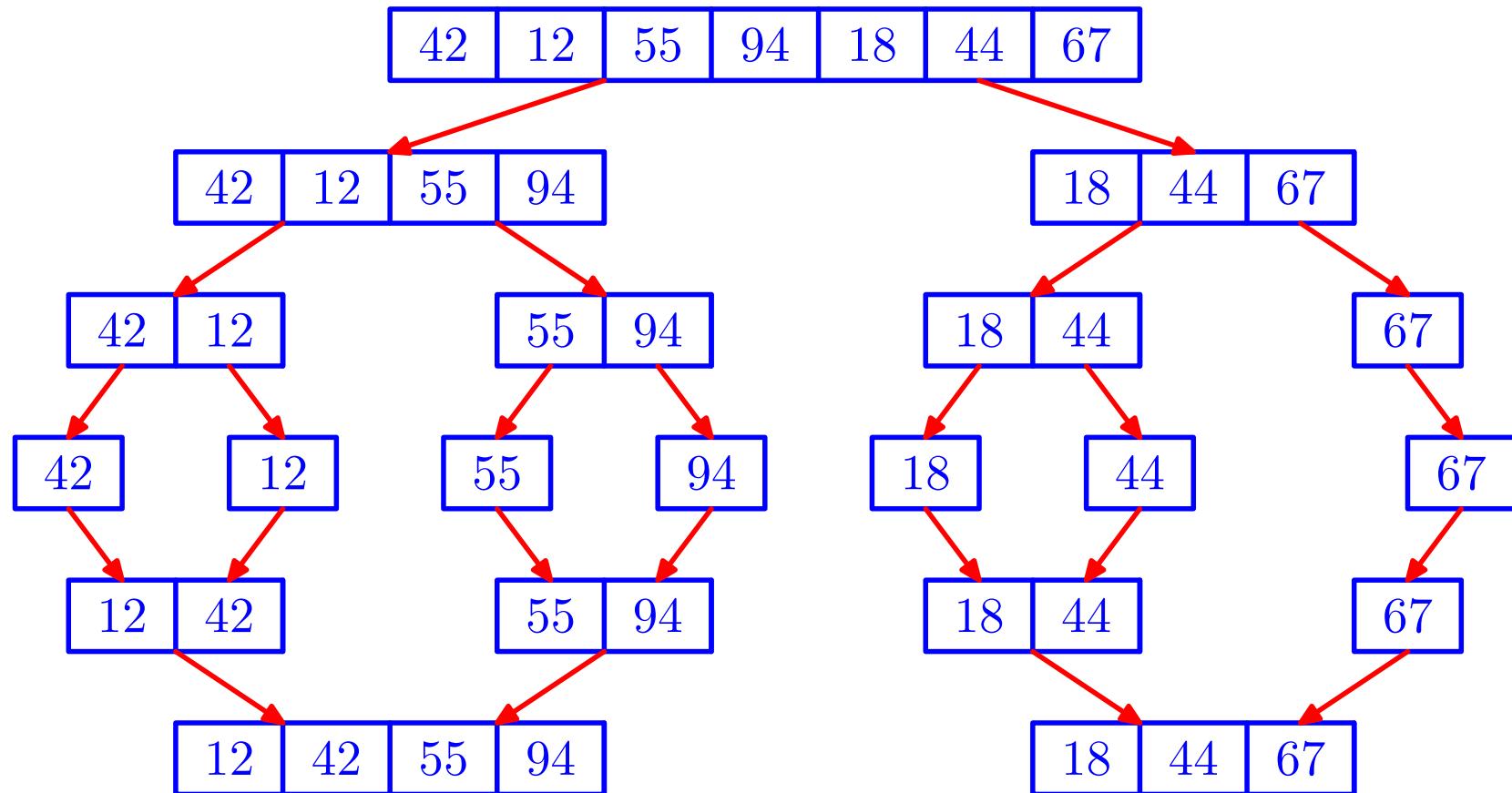
MergeSort — primjer



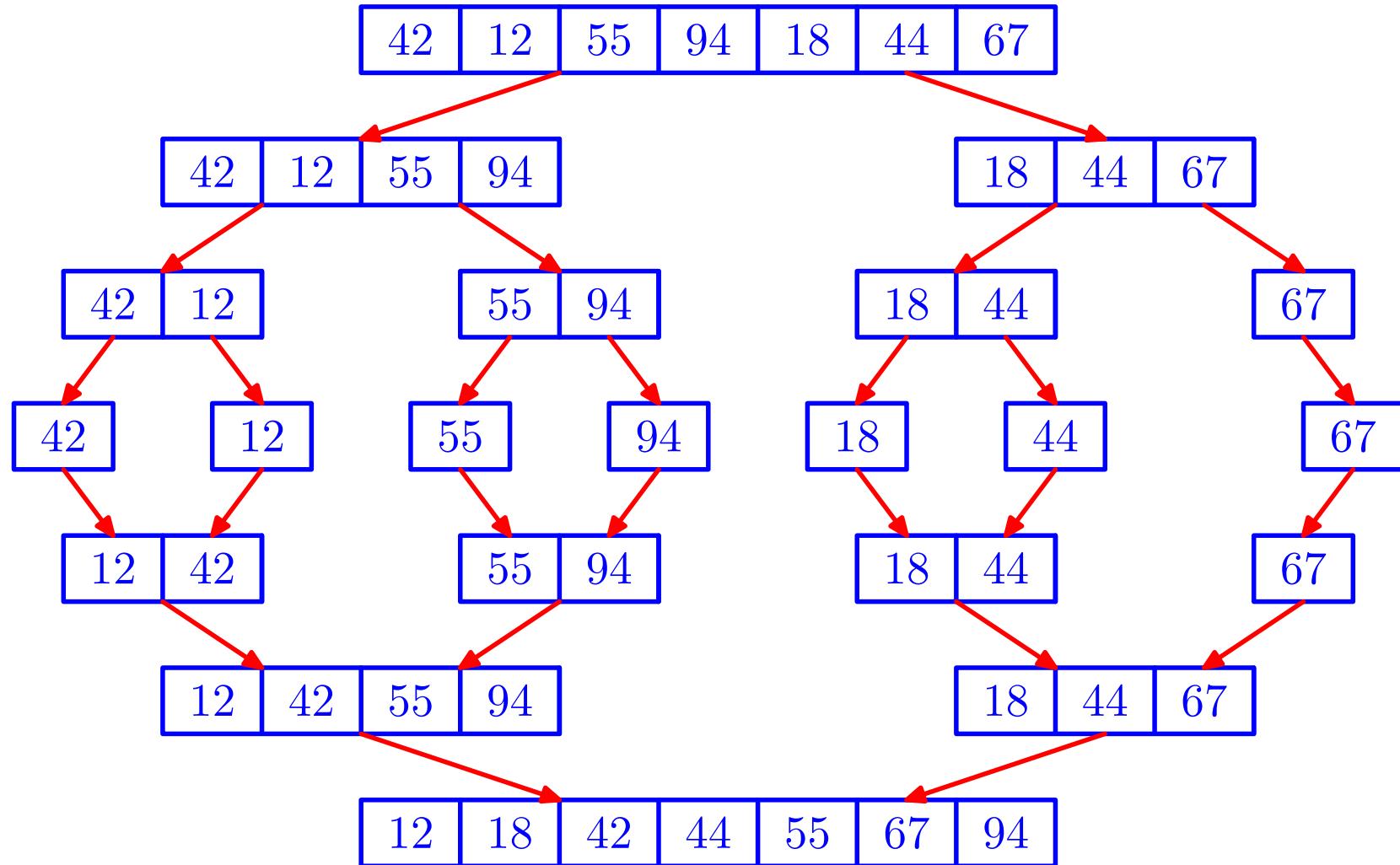
MergeSort — primjer



MergeSort — primjer



MergeSort — primjer



Sortiranje spajanjem — MergeSort (nastavak)

“Raspolavljanje” liste možemo realizirati na dva načina:

- preko **pokazivača**, pažljivim “šetanjem” po listi — elegantnije, ali sporije (v. funkciju `merge_sort`),
- **brojanjem** elemenata u listi — brže, ali treba dodatni ulazni argument (napravite **sami**).

Složenost MergeSort algoritma: $O(n \log n)$.

Dokaz. Slijedi iz prethodne **slike**. Neka je n broj elemenata u nizu i neka je

$$2^{k-1} < n \leq 2^k,$$

tj. 2^k je **najmanja** potencija broja 2 veća ili **jednaka** n , odnosno, $k = \lceil \log_2 n \rceil$. Na **slici** je $n = 7$ i $k = 3$.

Složenost MergeSorta (nastavak)

Uzmimo da polazni nesortirani niz ima razinu (nivo) 0.

Nakon toga imamo točno k “horizontalnih” razina

- raspolavljanja podnizova — u gornjem dijelu slike, i
- sortiranog spajanja podnizova — u donjem dijelu slike.

Dakle, broj “radnih” razina na slici je $2k \approx 2 \log_2 n$.

Uočimo da svako raspolavljanje i svaki merge

- traje linearno u duljini “većeg” odgovarajućeg niza, jer svaki element u “većem” nizu “prolazimo” najviše jednom.

Ako nije očito, pogledajte pažljivo “raspolavljanje” u funkciji `merge_sort` i funkciju `merge`.

Složenost MergeSorta (nastavak)

Kad to pozbrajamo za sve podnizove na istoj razini,

- jer svaka razina ima najviše n “radnih” elemenata, zaključujemo da
 - na svakoj razini imamo $O(n)$ operacija.

Kad pomnožimo broj razina i broj operacija, dobivamo da

- ukupno ima $O(n \log n)$ operacija.

Autor MergeSorta je John von Neumann, 1945. godine.

- To je prvi program napisan za računalo koje sprema i programe (von Neumannov model).
- Računalo se zvalo EDVAC.

Funkcija merge_sort

```
lista merge_sort(lista prvi)
{
    /* Sortira listu Merge_Sort algoritmom. */

    lista zadnji, prvi_2;

    /* Test na praznu ili jednoclangu listu. */

    if ((prvi == NULL) || (prvi->sljed == NULL))
        return prvi;
```

Funkcija merge_sort — nastavak

```
/* ‘‘Raspolovi’’ listu. Pokazivac zadnji  
pokazuje na zadnjeg u prvom dijelu.  
Pokazivac prvi_2 je trenutno pomocni  
i sluzi za raspalavljanje liste. */
```

```
zadnji = prvi;  
prvi_2 = prvi->sljed;
```

Funkcija merge_sort — nastavak

```
/* Pomicemo zadnjeg za JEDNO mjesto,  
   a prvi_2 za DVA mesta, sve dok  
   prvi_2 ne stigne do kraja liste. */  
  
while ((prvi_2 != NULL) &&  
       (prvi_2->sljed != NULL)) {  
  
    zadnji = zadnji->sljed;  
    prvi_2 = prvi_2->sljed->sljed;  
}  
  
/* NE VALJA: (iz programa na webu)  
   prvi_2 = zadnji->sljed->sljed;  
*/
```

Funkcija merge_sort — nastavak

```
/* Pokazivac zadnji sad korektno pokazuje
na zadnjeg u prvom dijelu.
Pokazivac prvi_2 postavljamo na prvog
u drugom dijelu (prvi iza zadnjeg) i
korektno zavrsavamo prvi dio. */

prvi_2 = zadnji->sljed;
zadnji->sljed = NULL;

prvi = merge(merge_sort(prvi),
            merge_sort(prvi_2));

return prvi;
}
```

MergeSort — demo-program

Program `1_9.c` kreira jednu listu oblika

- `42 -> 12 -> 55 -> 94 -> 18 -> 44 -> 67 -> NULL,`

(standardni primjer niza podataka za algoritme sortiranja), a zatim sortira tu listu MergeSort algoritmom. Dobivena lista je

- `12 -> 18 -> 42 -> 44 -> 55 -> 67 -> 94 -> NULL.`

Program `1_9w.c` radi to isto, s puno ispisa.

Zadatak. Napišite funkciju za MergeSort liste kojoj stiže broj elemenata u listi.

Zadatak. Napišite funkciju za MergeSort na polju. Za merge (spajanje) smijete koristiti jedno dodatno pomoćno polje!