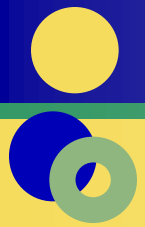
Sortiranje listi



Strukture podataka i algoritmi 2

Sortiranje listi

- Na prethodnim časovima smo se upoznali sa algoritmima za sortiranje niza
- Sada ćemo razmotriti algoritme za sortiranje listi
- U principu svaki postupak za sortiranje niza se može adaptirati za sortiranje listi, ali neki postupci su prirodniji listama (i obratno)
  - Npr. sortiranje umetanjem je prirodno listama
    - Element se trivijalno umeće na neko drugo mesto u listi
    - Kod nizova smo pak morali pomerati elemente

Sortiranje listi

- Prikazaćemo tri postupka za sortiranje jednostruko povezanih listi: sortiranje umetanjem, sortiranje spajanjem (engl. merge sort), i quick sort.

```
public class SortableList<T extends Comparable<T>> {
	private class Node {
		T info;
		Node next;
		public Node(T info) { this.info = info; }
	}

	private Node root = null;

	public void add(T info) {
		Node newElement = new Node(info);
		newElement.next = root;
		root = newElement;
	}

	public void insertionSort() { ... }
	public void mergeSort()     { ... }
	public void quickSort()     { ... }
}
```

Sortiranje umetanjem

- Ako je lista prazna ili jednoelementna tada je ona sortirana
- Neka je levi deo liste sortiran, a desni nesortiran
  - *lastSorted* pokazivač na poslednji element u sortiranom delu liste
  - *firstUnsorted* pokazivač na prvi element u nesortiranom delu liste,
    firstUnsorted = lastSorted.next
  - Razlikujemo tri slučaja
    - **[1] firstUnsorted >= lastSorted:** nema umetanja, sortirani deo liste se trivijalno povećava (*lastSorted = firstUnsorted*)
    - **[2] firstUnsorted < root:** *firstUnsorted* treba umetnuti pre *root*
    - **[3] Inače:** *firstUnsorted* treba umetnuti iza *lastLeq*
      - *lastLeq:* pokazivač na poslednji element između *root* i *firstUnsorted* koji je manji ili jednak od *firstUnsorted*
      - *lastLeq* se lako pronalazi sekvencijalnim pretraživanjem liste od korena ka kraju.

```java
public void insertionSort() {
	if (root == null || root.next == null)
		return;

	Node lastSorted = root;
	while (lastSorted.next != null) {
		Node firstUnsorted = lastSorted.next;

		// firstUnsorted >= lastSorted -- nema umetanja
		if (firstUnsorted.info.compareTo(lastSorted.info) >= 0)
			lastSorted = firstUnsorted;
		else
		// firstUnsorted < root -- umetanje na pocetak
		if (firstUnsorted.info.compareTo(root.info) < 0) {
			lastSorted.next = firstUnsorted.next;
			firstUnsorted.next = root;
			root = firstUnsorted;
		}
		else {
			// pronalazenje lastLeq
			Node current = root, lastLeq = null;
			while (current.info.compareTo(firstUnsorted.info) <= 0) {
				lastLeq = current; current = current.next;
			}
			// umetanje firstUnsorted iza lastLeq
			lastSorted.next = firstUnsorted.next;
			firstUnsorted.next = lastLeq.next;
			lastLeq.next = firstUnsorted;
		}
	}
}
```

Primer

```
public class ExSortableList {
	public static void main(String[] args) {
		SortableList<Integer> l = new SortableList<Integer>();
		for (int i = 0; i < 20; i++) {
			l.add((int) (Math.random() * 10));
		}
		l.print();
		l.insertionSort();
		l.print();

		SortableList<String> ls = new SortableList<String>();
		ls.add("Zika"); ls.add("Mika");
		ls.add("Pera"); ls.add("Aca");
		ls.insertionSort();
		ls.print();
	}
}
```

```
4 8 5 8 6 2 4 3 0 9 1 1 1 0 2 0 8 7 7 0
0 0 0 0 1 1 1 2 2 3 4 4 5 6 7 7 8 8 8 9
Aca Mika Pera Zika
```

Sortiranje spajanjem (engl. merge sort)

- Merge sort je još jedan od algoritama baziranih na principu “zavadi pa vladaj”
- Merge sort je linearitamske vremenske složenosti O(n logn)
- Osnovna ideja:
  - Podelimo listu L u dve balansirane liste L1 i L2 (dužine L1 i L2 se razlikuju najviše za 1)
  - Sortiramo L1 i L2 merge sortom
  - Spojimo dve sortirane liste L1 i L2 u finalnu sortiranu listu L
- Stoga se merge sort sastoji od dva podproblema:
  - Podele liste u dve balansirane liste
  - Spajanje sortiranih listi u sortiranu listu

Podela liste u dve podliste L1 i L2

- Neka je **start** pokazivač na koren orginalne **liste koja ima bar dva elementa**
- Neka su **l1** i **l2** pokazivači na korene podlisti L1 i L2
- Uvešćemo još dva pokazivača
  - **l1End** – pokazivač na poslednji element u L1
  - **l2End** – pokazivač na poslednji element u L2
- Podelu liste započinjemo tako što inicijalizujemo pokazivače
  - `l1 = start     ; l1End = l1;`
  - `l2 = start.next; l2End = l2;`
- Drugim rečima na početku **l1** i **l1End** pokazuju na prvi element orginalne liste, a **l2** i **l2End** na drugi element orginalne liste

Podela liste u dve podliste L1 i L2

- Koreni L1 i L2 su sada postavljeni i ne menjaju se više, ažuriramo samo pokazivače na poslednje elemente L1 i L2
  - Drugim rečima, imamo dodavanje elemenata iz orginalne liste na kraj liste L1 ili kraj liste L2
- Neka je **current** prvi element iz L koji nije ubačen u L1 ili L2
- Na početku **current = start.next.next**
- Ostatak podele realizujemo tako što:
  - **current** dodamo na kraj liste L1
  - **current = current.next** → povratimo esencijalno svojstvo **current** pokazivača
  - **current** dodamo na kraj liste L2
  - **current = current.next** → povratimo esencijalno svojstvo **current** pokazivača
  - Ponavljamo prethodne korake do kraja orginalne liste

Podela liste u dve podliste L1 i L2

- Još o dve stvari treba povesti računa prilikom podele:
  - Tekući element koji se dodaje u listu L1 može biti poslednji element orginalne liste (ako lista ima neparan broj elemenata).
    - Drugim rečima **current** može postati ***null*** pošto smo mu posle dodavanja na kraj liste L1 povratili esencijalno svojstvo da je prvi element koji nije dodan u L1 ili L2
    - Stoga pre dodavanja u L2 treba proveriti da li je **current** različito od ***null***
  - Pošto je postupak podele završen treba postaviti **"uzemljenja"** za liste L1 i L2
    - **l1End.next = null;**
    - **l2End.next = null;**

Podela liste u dve podliste L1 i L2

```
Node l1 = start, l1End = l1;
Node l2 = start.next, l2End = l2;

Node current = start.next.next;
while (current != null) {
     // dodaj current u l1
     l1End.next = current;
     l1End = current;

     // pomeri current za jedno mesto u desno
     current = current.next;

     // dodaj current u l2
     if (current != null) {
          l2End.next = current;
          l2End = current;
          current = current.next;
     }
}

// postavi uzemljenja
l1End.next = null;
l2End.next = null;
```

Spajanje sortiranih listi

- Uvešćemo sledeće pokazivače:
  - **root**: pokazivač na koren liste koja se dobija spajanjem dve sortirane liste L1 i L2
  - **last**: pokazivač na poslednji element u spojenoj listi
  - **l1**: pokazivač na prvi element iz L1 koji nije dodat u spojenu listu
  - **l2**: pokazivač na prvi element iz L2 koji nije dodat u spojenu listu
- Na početku **l1** i **l2**, prirodno, pokazuju na korene liste L1 i L2
- Spajanje realizujemo u tri jednostavna koraka
  1. Određivanje korena spojene liste
  2. Dodavanje elemenata iz L1 i L2 u spojenu listu
  3. Kalemljenje ostaka kada se jedna od listi “isprazni”.

Spajanje sortiranih listi

- **Prvi korak: određivanje korena spojene liste**
  - Ako je prvi element L1 manji od prvog elementa L2 tada
    - `root = l1;`
    - `l1 = l1.next; // povraćaj esencijalne osobine l1`
  - Inače
    - `root = l2;`
    - `l2 = l2.next; // povraćaj esencijalne osobine l2`

- Prvi element spojene liste je na početku istovremeno i njen poslednji element:
    - `last = root`

Spajanje sortiranih listi

- **Drugi korak: dodavanje elemenata u spojenu listu**

- Prirodno, razlikujemo dva slučaja
  - **l1.info < l2.info:** dodajemo l1 na kraj spojene liste
    - `last.next = l1;`
    - `last = l1;    // povraćaj esencijalne osobine last`
    - `l1 = l1.next; // povraćaj esencijalne osobine l1`
  - **l1.info >= l2.info:** dodajemo l2 na kraj spojene liste
    - `last.next = l2;`
    - `last = l2;    // povraćaj esencijalne osobine last`
    - `l2 = l2.next; // povraćaj esencijalne osobine l2`

- Prethodnu operaciju ponavljamo dokle god postoje oba l1 i l2

Treći korak (kalemljenje ostatka)

- Posmatrajmo liste
  - L1 = (2, 4, 7, 8, 10)
  - L2 = (1, 3, 5, 6)
- Nakon drugog koraka spajanja imaćemo situaciju
  - Spojena = (1, 2, 3, 4, 5, 6)
  - **l1** pokazuje na 7
  - **l2** = null
- Stoga je “ostatak” iz L1 potrebno nakalemiti na kraj spojene liste.
- U opštem slučaju imamo

l1 == null ? last.next = l2 : last.next = l1;

```
private Node merge(Node l1, Node l2) {
	Node root = null;

	if (l1.info.compareTo(l2.info) < 0) {
		root = l1;
		l1 = l1.next;
	} else {
		root = l2;
		l2 = l2.next;
	}

	Node last = root;
	while (l1 != null && l2 != null) {
		if (l1.info.compareTo(l2.info) < 0) {
			last.next = l1;
			last = l1;
			l1 = l1.next;
		} else {
			last.next = l2;
			last = l2;
			l2 = l2.next;
		}
	}

	last.next = l1 == null ? l2 : l1;

	return root;
}
```

Merge sort

```
public void mergeSort() {
     if (root != null)
          root = mergeSort(root);
}

private Node mergeSort(Node start) {
     // jednoelementna lista je sortirana
     if (start.next == null)
          return start;

     //... podela liste na dva dela l1 i l2

     l1 = mergeSort(l1);
     l2 = mergeSort(l2);

     // spojanje sortiranih listi
     return merge(l1, l2);
}
```

Quick sort

- Quick sort je baziran na ideji pivota i podele liste na dve liste naspram pivota
- Neka je L = (G | R) lista koja se sortira
- Za pivota selektujemo G: element L koji se najjednostavnije i najbrže dobavlja
- Na osnovu R formiramo dve liste
  - M – elementi iz R koji su manji od pivota
  - V – elementi iz R koji su veći ili jednaki od pivota
- Sortiramo M i V: M’ = sorted(M), V’ = sorted(V)
- Sortirana lista L’ ima oblik: L’ = (M’ | G | V’)

Quick sort

- Formiranje lista M i V se realizuje jednostavno jednim obilaskom liste G

- Sortirana lista L’ ima oblik: L’ = (M’ | G | V’)
  - M’ može biti prazna lista: tada je koren L’ G, a ne prvi element iz M’ (koji ne postoji)
  - Ako M’ nije prazna, tada G treba nakalemiti na kraj M’ → moramo se “došetati” do kraja liste M’
  - Kalemljenje V’ na G je trivijalna operacija

Quick sort

```
public void quickSort() {
      if (root != null)
            root = quickSort(root);
}

private Node quickSort(Node start) {
      // lista koja ima jedan element je sortirana
      if (start.next == null)
            return start;

      // za pivota selektujemo prvi element liste start
      Node pivot = start;

      // smaller je pokazivac na koren liste koja sadrzi
      // elemente manje od pivota, a greater je pokazivac
      // na koren liste koja sadrzi elemente vece od pivota
      Node smaller = null, greater = null;

      //... to be continued
}
```

```
Node current = pivot.next;
while (current != null) {
	Node afterCurrent = current.next;

	if (current.info.compareTo(pivot.info) < 0) {
		// dodajemo current na pocetak liste smaller
		current.next = smaller;
		smaller = current;
	} else {
		// dodajemo current na pocetak liste greater
		current.next = greater;
		greater = current;
	}

	current = afterCurrent;
}

if (smaller != null) smaller = quickSort(smaller);
if (greater != null) greater = quickSort(greater);

pivot.next = greater;
if (smaller == null) return pivot;
else {
	Node tmp = smaller;
	while (tmp.next != null) tmp = tmp.next;
	tmp.next = pivot;
	return smaller;
}
```