Sorting and Searching, practicum 1

Efficiëntie van geavanceerde sorteeralgoritmes

Kostis Thanos
IS-204
laatste versie 27-10-2016

Inhoudsopgave

Inleiding

In opdracht van de Hogeschool van Amsterdam moet ik (Kostis Thanos) voor het vak Sorting and Searching een aantal opdrachten maken en die vervolgens weer verwerken in verslagen. Dit verslag betreft opdracht 1, waarin ik het geavanceerde sorteeralgoritme “Quicksort” implementeer. Vervolgens test ik zijn tijdsefficiëntie en ik bepaal zijn “Big O” waarde. Hetzelfde doe ik voor de “Median-of-3-partitioning” versie van de Quicksort, waarna ik ze allebei met elkaar vergelijk om een conclusie te trekken. Als laatst maak ik een zogenoemde “Binary Search Tree” van de cijfers van de studenten (de cijfers zijn van 1,0 tot en met 9,9).

Quicksort

Om de studenten te generen heb ik de klasse hieronder geprogrammeerd.

```
public class StudentenGenerator {

	public static Comparable[] maakStudentenAan(int aantal){
		Random random = new Random();
		final Comparable[] studenten =(Comparable[]) new Student[aantal];

		for(int i = 0; i < aantal; i++){
			final double dbl =((random == null ?
			new Random() : random).nextDouble() * (9) + 1);
			Comparable student = new Student(i + 1,
String.format(Locale.US,"%." + 1 + "f", dbl));
			studenten[i] = student;
		}
		return studenten;
	}
```

Die vervolgens zo wordt aangeroepen in de Main:

```
int aantalStudenten = 50;
Comparable[] studenten =  StudentenGenerator.maakStudentenAan(aantalStudenten);
```

Om de Quicksort te laten werken moeten we eerst een Shuffle-methode hebben, die ziet er als volgt uit:

```
public class Shuffle {
	public static void shuffle(Comparable[] array){
		{
		    // If running on Java 6 or older, use `new Random()` on RHS here
		    Random rnd = new Random();
		    for (int i = array.length - 1; i > 0; i--)
		    {
		      int index = rnd.nextInt(i + 1);
		      // Simple swap
		      Comparable temp = array[index];
		      array[index] = array[i];
		      array[i] = temp;
		    }
		}
	}
}
```

De Quicksort methode en de Partition methode die hij aanroept zien er zo uit:

```
public class Quick{
	public static void sort(Comparable[] a){
		Shuffle.shuffle(a); // Eliminate dependence on input.
		sort(a, 0, a.length - 1);
	}

	private static void sort(Comparable[] a, int lo, int hi){
		if (hi <= lo){
			return;
		}
		int j = Partition.partition(a, lo, hi); // Partition (see page 291).
		sort(a, lo, j-1); // Sort left part a[lo .. j-1].
		sort(a, j+1, hi); // Sort right part a[j+1 .. hi].
	}
}

public class Partition {
	public static int partition(Comparable[] a, int lo, int hi){
		// Partition into a[lo..i-1], a[i], a[i+1..hi].
		int i = lo, j = hi+1; // left and right scan indices
		Comparable v = a[lo]; // pivot v
		while (true){ // Scan right, scan left, check for scan complete, and
exchange.
			while(less(a[++i], v)){
				if (i == hi){
					break;
				}
			}
			while (less(v, a[--j])){
				if (j == lo){
					break;
				}
			}
			if (i >= j){
				break;
			}
			exch(a, i, j);
		}

		exch(a, lo, j); // Put v = a[j] into position
		return j; // with a[lo..j-1] <= a[j] <= a[j+1..hi].
	}

	private static boolean less(Comparable v, Comparable w){
		return v.compareTo(w) < 0;
	}

	private static void exch(Comparable[] a, int i, int j){
		Comparable t = a[i]; a[i] = a[j]; a[j] = t;
	}
}
```

Daarna wordt de studentenLijst van 50 studenten als volgt in de main geshuffled en gesorteerd:

```
System.out.println("\nShuffle");
Shuffle.shuffle(studenten);
for(int i = 0; i < aantalStudenten; i++){
	System.out.println(studenten[i]);
}

Quick.sort(studenten);
System.out.println("\nQuicksort");
for(int i = 0; i < aantalStudenten; i++){
	System.out.println(studenten[i]);
}
```

De uitkomst van de shuffle gevolgt door de Quicksort :

Shuffle	Quicksort
41 2.6	19 9.7
19 9.7	50 9.5
3 4.1	26 9.2
37 3.8	33 9.0
48 5.5	38 9.0
32 1.5	22 8.8
6 4.8	39 8.7
18 5.9	46 8.4
13 2.3	31 8.3
36 1.3	21 8.1
8 1.7	14 7.0
14 7.0	40 6.3
15 1.9	28 6.2
2 3.2	18 5.9
47 5.2	24 5.8
50 9.5	42 5.8
43 3.1	48 5.5
38 9.0	12 5.2
44 4.9	45 5.2
24 5.8	47 5.2
5 2.2	34 4.9
23 4.2	44 4.9
25 3.2	6 4.8
29 4.6	9 4.8
1 2.6	16 4.7
16 4.7	29 4.6
33 9.0	11 4.4
9 4.8	23 4.2
28 6.2	3 4.1
30 1.5	37 3.8
27 2.3	35 3.6
21 8.1	17 3.3
34 4.9	2 3.2
49 1.5	25 3.2
17 3.3	43 3.1
22 8.8	10 3.0
45 5.2	1 2.6
10 3.0	41 2.6
7 2.1	13 2.3
42 5.8	27 2.3
46 8.4	5 2.2
4 1.5	7 2.1
39 8.7	15 1.9
11 4.4	8 1.7
35 3.6	20 1.7
31 8.3	4 1.5
26 9.2	30 1.5
40 6.3	32 1.5
12 5.2	49 1.5
20 1.7	36 1.3

Conclusie: De methode functioneert correct.

Efficiëntie Quicksort meten

Om de efficiëntie te meten luidde de opdracht als volgt:
Onderzoek de efficiëntie van het sorteeralgoritme door een aantal experimenten uit te voeren1. Genereer (in een loop) verschillende lijsten van 10.000, 20.000, 40.000, 80.000 en 160.000 studenten met hun resultaten. Sorteer vervolgens deze lijst op de gewenste manier

Om de tijd te meten maakte ik de klasse stopwatch:

```
public class StopWatch {
        private final long start;

            public StopWatch() {
                start = System.currentTimeMillis();
            }

            public double elapsedTime() {
                long now = System.currentTimeMillis();
                return (now - start) / 1000.0;
            }
}
```

In de code maakte ik een array van de cijfers gevolgt door de code voor de tijstest:

```
int[] aantallenStudenten = {10000,20000,40000,80000,160000};

              //Tijdstest voor Quicksort
              for (int k = 0; k < aantallenStudenten.length; k++){
                    int aantal = aantallenStudenten[k];
                    Comparable[] groteStudenten =
StudentenGenerator.maakStudentenAan(aantal);
                    Shuffle.shuffle(groteStudenten);
                    StopWatch stopWatch1 = new StopWatch();
                    Quick.sort(groteStudenten);
                    System.out.println("Aantal seconden voor " + aantal + "
studenten (Quicksort) = " + stopWatch1.elapsedTime());
              }
```

Daaruit volgde de volgende resultaten:

```
Aantal seconden voor 10000 studenten (Quicksort) = 0.234
Aantal seconden voor 20000 studenten (Quicksort) = 0.257
Aantal seconden voor 40000 studenten (Quicksort) = 0.314
Aantal seconden voor 80000 studenten (Quicksort) = 0.619
Aantal seconden voor 160000 studenten (Quicksort) = 1.245
```

Verbetering Quicksort

Om de Quicksort te “verbeteren” heb ik de Median-of-3-partitioning als volgt geïmplementeerd :

```
public class Quick3way{
	public static void sort(Comparable[] a, int lo, int hi){ // See page 289 for
public sort() that calls this method.
		if (hi <= lo) return;
		int lt = lo, i = lo+1, gt = hi;
		Comparable v = a[lo];
		while (i <= gt){
			int cmp = a[i].compareTo(v);
			if (cmp < 0) exch(a, lt++, i++);
			else if (cmp > 0) exch(a, i, gt--);
			else i++;
		} // Now a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
		sort(a, lo, lt - 1);
		sort(a, gt + 1, hi);
	}

	private static void exch(Comparable[] a, int i, int j){
		Comparable t = a[i]; a[i] = a[j]; a[j] = t;
	}
}
```

Verbetering Quicksort meten

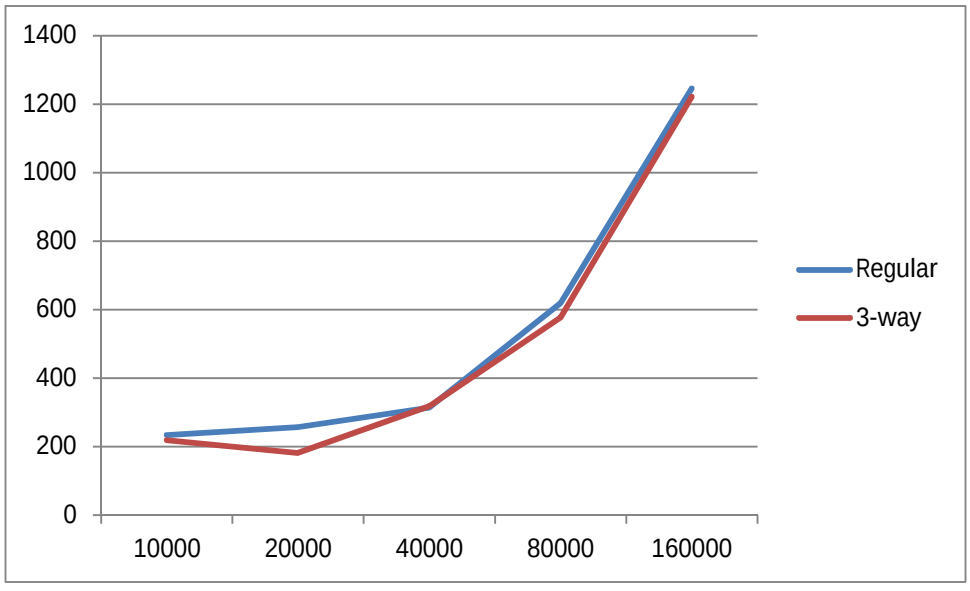
Om de verbetering van de Quicksort te meten doen we in principe het zelfde als met de originele Quicksort:

```
//Tijdstest voor verbeterde Quicksort
for (int k = 0; k < aantallenStudenten.length; k++){
      int aantal = aantallenStudenten[k];
      Comparable[] groteStudenten =
      StudentenGenerator.maakStudentenAan(aantal);
      Shuffle.shuffle(groteStudenten);
      StopWatch stopWatch2 = new StopWatch();
      Quick3way.sort(groteStudenten, 0, aantal - 1);
      System.out.println("Aantal seconden voor " + aantal + "
      studenten (Quick3Way) = " + stopWatch2.elapsedTime());
}
```

Met als uitkomst de volgende resultaten:

```
Aantal seconden voor 10000 studenten (Quick3Way) = 0.219
Aantal seconden voor 20000 studenten (Quick3Way) = 0.181
Aantal seconden voor 40000 studenten (Quick3Way) = 0.318
Aantal seconden voor 80000 studenten (Quick3Way) = 0.577
Aantal seconden voor 160000 studenten (Quick3Way) = 1.221
```

Als we de resultaten van de reguliere en de verbeterde quicksort in een grafiek stoppen krijgen we het volgende plaatje:



(Op de y-as zijn het aantal milliseconden, op de x-as het aantal objecten)
3-Way-Partitioning (3WP) lijkt (bijna) altijd iets sneller te zijn. Een rare afwijking is de tijdsvergelijking van 10000 naar 20000 objecten bij de 3WP die sneller lijkt te gaan bij 20000 objecten t.o.v. de 10000.

Binary Search Tree (BST)

Als laatst maken we de BST aan de hand van de volgende code:

```
public class BST<Key extends Comparable<Key>, Value>{
	private Node root; // root of BST
	public BST(){
		//root = new Node(null, null, 0);
	}
	private class Node{
		private Key key; // key
		private Value val; // associated value
		private Node left, right; // links to subtrees
		private int N; // # nodes in subtree rooted here
		public Node(Key key, Value val, int N)
		{ this.key = key; this.val = val; this.N = N; }
	}
	public int size(){
		return size(root); }
	private int size(Node x){
		if (x == null){
			return 0;
		} else {
			return x.N;
		}
	}

	public Value get(Key key){
		return get(root, key); }

	private Value get(Node x, Key key){ // Return value associated with key in
the subtree rooted at x;
	// return null if key not present in subtree rooted at x.
		if (x == null) return null;
		int cmp = key.compareTo(x.key);
		if (cmp < 0) return get(x.left, key);
		else if (cmp > 0) return get(x.right, key);
		else return x.val;
	}

	public Node put(Key key, Value value){
		Node temp =  put(root, key, value);
		if (root == null){
			root = temp;
		}
			return temp;
		}

	private Node put(Node x, Key key, Value val)
	{
		// See page 399.
	// Change key's value to val if key in subtree rooted at x.
	// Otherwise, add new node to subtree associating key with val.
	if (x == null) {
		return new Node(key, val, 1);
	}
	int cmp = key.compareTo(x.key);
	if (cmp < 0) x.left = put(x.left, key, val);
```

```
        else if (cmp >= 0) x.right = put(x.right, key, val);
        x.N = size(x.left) + size(x.right) + 1;
        return x;
        }

// See page 407 for min(), max(), floor(), and ceiling().
        public Key min(){
        return min(root).key;
        }

        public Key max(){
        return max(root).key;
        }

        private Node min(Node x){
        if (x.left == null) return x;
        return min(x.left);
        }

        private Node max(Node x){
        if (x.right == null) return x;
        return max(x.right);
        }

        public Key floor(Key key){
        Node x = floor(root, key);
        if (x == null) return null;
        return x.key;
        }

        public Key ceiling(Key key){
        Node x = ceiling(root, key);
        if (x == null) return null;
        return x.key;
        }

        private Node floor(Node x, Key key){
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp < 0) return floor(x.left, key);
        Node t = floor(x.right, key);
        if (t != null) return t;
        else return x;
        }

        private Node ceiling(Node x, Key key){
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp > 0) return ceiling(x.right, key);
        Node t = ceiling(x.left, key);
        if (t != null) return t;
        else return x;
        }

// See page 409 for select() and rank().
        public Key select(int k){
              return select(root, k).key;
```

```
	}

	private Node select(Node x, int k)
	{ // Return Node containing key of rank k.
	if (x == null) return null;
	int t = size(x.left);
	if (t > k) return select(x.left, k);
	else if (t < k) return select(x.right, k-t-1);
	else return x;
	}

	public int rank(Key key){
		return rank(key, root);
		}

	private int rank(Key key, Node x){ // Return number of keys less than x.key
in the subtree rooted at x.
		if (x == null){
			return 0;
		}
		int cmp = key.compareTo(x.key);
		if (cmp < 0){
			return rank(key, x.left);
		}
		else if (cmp > 0){
			return 1 + size(x.left) + rank(key, x.right);
		} else {
			return size(x.left);
		}
	}

// See page 411 for delete(), deleteMin(), and deleteMax().
	public void deleteMin()
	{
	root = deleteMin(root);
	}
	private Node deleteMin(Node x)
	{
	if (x.left == null) return x.right;
	x.left = deleteMin(x.left);
	x.N = size(x.left) + size(x.right) + 1;
	return x;
	}
	public void delete(Key key)
	{ root = delete(root, key); }
	private Node delete(Node x, Key key)
	{
	if (x == null) return null;
	int cmp = key.compareTo(x.key);
	if (cmp < 0) x.left = delete(x.left, key);
	else if (cmp > 0) x.right = delete(x.right, key);
	else
	{
	if (x.right == null) return x.left;
	if (x.left == null) return x.right;
	Node t = x;
	x = min(t.right); // See page 407.
	x.right = deleteMin(t.right);
	x.left = t.left;
```

```
        }
        x.N = size(x.left) + size(x.right) + 1;
        return x;
        }

// See page 413 for keys().
        public Iterable<Key> keys(){
              return keys(min(), max());
        }

        public Iterable<Key> keys(Key lo, Key hi)
        {
        Queue<Key> queue = new LinkedList<Key>();
        keys(root, queue, lo, hi);
        return queue;
        }
        private void keys(Node x, Queue<Key> queue, Key lo, Key hi)
        {
        if (x == null) return;
        int cmplo = lo.compareTo(x.key);
        int cmphi = hi.compareTo(x.key);
        if (cmplo < 0) keys(x.left, queue, lo, hi);
        if (cmplo <= 0 && cmphi >= 0) queue.add(x.key);
        if (cmphi > 0) keys(x.right, queue, lo, hi);
        }
}
```

In de Main roepen we de code als volgt aan:

```
              BST<Double, Integer> bst = new BST<Double, Integer>();

              for(Comparable student : studenten){
                    bst.put(Double.parseDouble(
                    ((Student)(student)).getCijfer()),(((Student)
                    student).getStudentNummer()));
              }
              System.out.println("\nBST");

              List<String> lijstBST = new ArrayList<String>();
              for(Comparable student : studenten){

                    Student deStudent = (Student)(student);
                    int rank = bst.rank(Double.parseDouble(deStudent.getCijfer()));
                    if(lijstBST.contains(deStudent.getCijfer())){
                          continue;
                    } else {
                          lijstBST.add(deStudent.getCijfer());
                    System.out.println(((Student)student).getCijfer() + " " +
                    rank);
              }
```

De output van 50 studenten is als volgt:

```
BST
Cijfer Rank
10.0   48
9.9    47
9.8    46
9.7    44
9.4    43
9.2    42
9.1    41
9.0    40
8.8    39
8.5    37
8.3    36
8.1    34
8.0    33
7.6    32
7.3    31
7.2    30
6.6    27
6.4    25
6.3    24
6.0    23
5.9    22
5.6    21
5.5    20
5.2    19
4.9    18
4.7    17
4.4    16
4.3    14
4.2    13
4.1    12
4.0    11
3.8    10
2.7    9
2.6    7
2.5    6
2.2    4
2.1    3
1.8    2
1.4    1
1.2    0
```

Conclusie: de BST werkt correct.