Lösningsförslag till tentamen

Datastrukturer, DAT037 (DAT036), 2017-04-11

1. Loopen upprepas `n` gånger, för $i$ från 0 till $\mathtt{n}-1$. Komplexiteten för `deleteMin` för en binär heap är $O(\log j)$ där $j$ är heapens storlek, här $\mathtt{n}-i$. Komplexiteten för `addLast` för en dynamisk array är $O(1)$ amorterat. Komplexiteten för $n$ anrop till `addLast` är $O(n)$. Övrigt i loopen tar konstant tid.

$$T(\mathtt{n}) = \sum_{i=0}^{\mathtt{n}-1} \log(\mathtt{n}-i) + O(\mathtt{n}) = \sum_{i=1}^{\mathtt{n}} \log i + O(\mathtt{n}) = O(\mathtt{n}\log\mathtt{n}+\mathtt{n}) = O(\mathtt{n}\log\mathtt{n})$$

2. Står felaktigt i tentauppgiften att komplexiteten ska vara $O(V)$. Det ska vara $O(V+E)$, där $E$ är antalet kanter.

```
public int addNode() {
        int i = adj.size();
        adj.add(new TreeSet<>());
        return i;
}

public void addEdge(int n1, int n2) {
        adj.get(n1).add(n2);
        adj.get(n2).add(n1);
}

public boolean isConnected() {
        if (adj.size() == 0) return true;
        Set<Integer> visited = new HashSet<>();
        traverse(0, visited);
        return visited.size() == adj.size();
}

private void traverse(int n, Set<Integer> visited) {
        if (visited.contains(n)) return;
        visited.add(n);
        for (Integer adjn : adj.get(n)) {
                traverse(adjn, visited);
        }
}
```

Komplexiteten för `isConnected`: Allt annat än anropet till `traverse` är $O(1)$. Det bortsett från första raden i `traverse` exekveras max $V$ gånger.

Varje exekvering av detta, exklusive loop-upprepningarna, tar $O(1)$ (förutsatt att hashtabellen fungerar väl). Eftersom loopen bara exekveras max 1 gång per nod så upprepas loopen och första raden i `traverse` totalt sett (för alla anrop exekveringar av `traverse`) maximalt $2E$ gånger. Varje exekvering av dessa satser tar $O(1)$ eftersom uppslag i hashtabellen och att stega fram iteratorn för en `TreeSet` tar konstant tid. Komplexiteten är alltså $O(V + E)$.

3.
```
public class MSet {
        TreeSet<Integer> a, b;
        // a innehåller den mindre halvan av elementen,
        //  dvs elementen mindre än medianen.
        // b innehåller den större halvan.
        // Om antalet element är udda så innehåller a
        // ett element fler än b.
        // Medianen är alltså hela tiden största
        // elementet i a.

        public MSet() {
                a = new TreeSet<>();
                b = new TreeSet<>();
        }

        public boolean add(int x) {
                if (a.contains(x) || b.contains(x)) return false;
                if (a.size() > b.size()) {
                        int y = a.last();
                        if (x < y) {
                                a.remove(y);
                                a.add(x);
                                b.add(y);
                        } else {
                                b.add(x);
                        }
                } else {  // a.size() == b.size()
                        if (b.isEmpty()) {
                                a.add(x);
                        } else {
                                int y = b.first();
                                if (x > y) {
                                        b.remove(y);
                                        b.add(x);
                                        a.add(y);
                                } else {
                                        a.add(x);
                                }
```

```
                }
            }
            return true;
        }

        public int deleteMedian() {
            int x = a.last();
            a.remove(x);
            if (a.size() < b.size()) {
                int y = b.first();
                b.remove(y);
                a.add(y);
            }
            return x;
        }
}
```

Komplexitet för `add`: `TreeSet`-operationerna `contains`, `remove`, `add` är alla $O(\log n)$. Allt annat i metoden tar konstant tid. Alltså: $O(\log n)$. Komplexitet för `deleteMedian`: `TreeSet`-operationerna `last`, `first`, `remove`, `add` är alla $O(\log n)$. Allt annat i metoden tar konstant tid. Alltså: $O(\log n)$.

4. Pivot-element: första elementet i delarrayen.

```
{8, 5, 7, 2, 1, 6, 4, 9}
 {4, 5, 7, 2, 1, 6}
  {2, 1}
   {1}
   {}
  {7, 5, 6}
   {6, 5}
    {5}
    {}
   {}
 {9}
```

5.
```
public class LinkedList<E> implements List<E> {
        private class Node {
                E e;
                Node next = null;
                Node(E e) {
                        this.e = e;
                }
        }
        Node first = null, last = null;
```

```
		public void addLast(E elt) {
			Node n = new Node(elt);
			if (first == null) {
				first = last = n;
			} else {
				last.next = n;
				last = n;
			}
		}

		public E get(int index) {
			Node n = first;
			for (; index > 0; index--) {
				n = n.next;
			}
			return n.e;
		}
	}
```

6. Utför beräkning av topologisk sortering som vanligt med skillnaden att noderna som står på kö läggs i en prioritetskö.

```
	public List<Integer> topol() {
		List<Integer> list = new ArrayList<>();
		Map<Integer, Integer> indeg = new HashMap<>();
		for (Set<Integer> adjNodes : adj.values()) {
			for (Integer adjNode : adjNodes) {
				int count = 0;
				if (indeg.containsKey(adjNode)) {
					count = indeg.get(adjNode);
				}
				indeg.put(adjNode, count + 1);
			}
		}
		PriorityQueue<Integer> q = new PriorityQueue<>();
		for (Integer node : adj.keySet()) {
			if (!indeg.containsKey(node)) {
				q.add(node);
			}
		}
		while (!q.isEmpty()) {
			Integer node = q.poll();
			list.add(node);
			for (Integer adjNode : adj.get(node)) {
				int count = indeg.get(adjNode) - 1;
				indeg.put(adjNode, count);
```

```
                        if (count == 0) {
                                q.add(adjNode);
                        }
                }
        }
        if (list.size() == adj.size()) {
                return list;
        } else {
                return null;
        }
}
```

Första slingan genomlöper hashtabellen med grannlistorna. För stora $V$ är hastabellens kapacitet begränsad av en $V$ multiplicerad med en konstakt faktor. Den inre for-loopen i denna uppreapas totalt $E$ gånger och `containsKey`, `get` och `put` tar idealt konstant tid. Komplexiteten för första slingan är $O(V + E)$.

Andra slingan upprepas $V$ gånger och iteratorns stegning är liksom för första $O(V)$. `containsKey` tar konstant tid och `add` $O(V)$ eftersom storleken på prioritetskön maximalt är $V$. Komplexiteten för andra slingan är $O(V \log V)$.

I tredje slingan upprepas `q.poll`, `list.add` och `q.add` maximalt $V$ gånger. Dessa har komplexiterna $O(\log V)$, $O(1)$ och $O(\log V)$. `indeg.get` och `indeg.put` upprepas maximalt $E$ gånger och har båda komplexiteten $O(1)$. Tredje slingen har alltså komplexiteten $O(V \log V + E)$.

Hela metoden har komplexiteten $O(E + V \log V)$.