Datenstrukturen und Algorithmen
Übung 5 – Elementare Datenstrukturen

Musterlösung

Theoretische Aufgaben

1. Geben Sie eine in Zeit $\Theta(n)$ laufende, nichtrekursive Prozedur an, welche die Reihenfolge einer **einfach** verketteten Liste aus n Elementen umkehrt. Die Prozedur sollte nur konstant viel Speicherplatz benutzen (abgesehen vom Speicherplatz, der für die Liste selbst gebraucht wird). **(1 Punkt)**

```
REVERSE-LINKED-LIST(L)
1  R := NIL // Reversed list head
2
3  cur := L.kopf
4  while cur ≠ NIL
5        next := cur.next;
6        cur.next := R;
7        R := cur;
8        cur := next;
9  L.kopf := R
```

2. Schreiben Sie Pseudocode, um eine Warteschlange mit einer einfach verketteten Liste zu implementieren. Ihre Lösung soll Code für die Operationen ENQUEUE und DEQUEUE enthalten. Nehmen Sie an, die Listenelemente hätten ein Feld *next* mit dem Zeiger auf das nächste Element, und ein Feld *key* mit dem Schlüssel. Die Operationen ENQUEUE und DEQUEUE sollten noch immer in Zeit $\mathcal{O}(1)$ arbeiten.

Die Queue Q hat ein Attribut *head*, das auf den Anfang der Linked List zeigt, und ein Attribut *tail* für das letzte Element; beide sind mit NIL initialisiert.

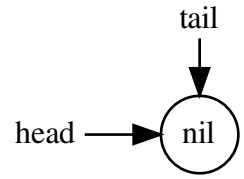
Die Methode MAKE-ELEMENT$(key, next)$ soll ein Listenelement mit den jeweiligen Attributen erzeugen.

```
ENQUEUE(Q,k)
1  E := MAKE-ELEMENT(k,NIL)
2  if Q.tail == NIL
3        // Empty list
4        Q.head := E
5  else
6        Q.tail.next := E
7  Q.tail := E
```

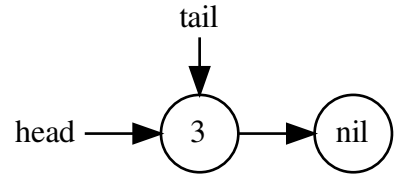
```
DEQUEUE(Q)
1  if Q.head == NIL
2        return NIL
3  else
4        k := Q.head.key
5        Q.head := Q.head.next
6        if Q.head == NIL
7              Q.tail := NIL
8        return k
```

Illustrieren Sie den Ablauf der folgenden Operationen, indem Sie für jeden Schritt die Liste darstellen und gegebenenfalls den Rückgabewert angeben: *ENQUEUE(3); ENQUEUE(5); DEQUEUE(); ENQUEUE(2); DEQUEUE(); ENQUEUE(8); ENQUEUE(9); DEQUEUE(); DEQUEUE(); DEQUEUE(); DEQUEUE()* **(1 Punkt)**

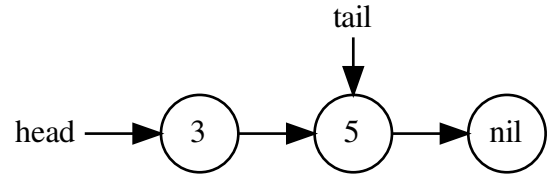
Initial:



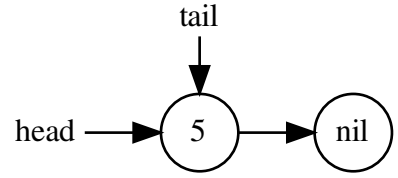
Nach ENQUEUE(3):



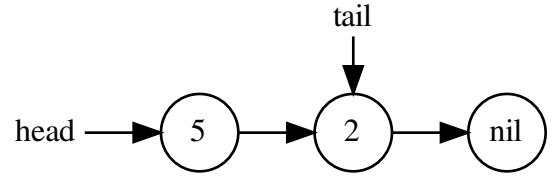
Nach ENQUEUE(5):



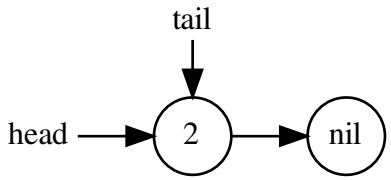
Nach DEQUEUE() = 3:



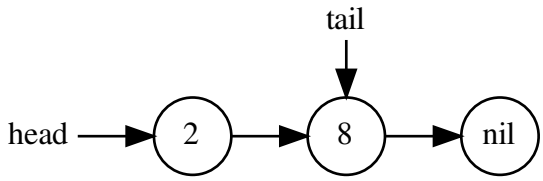
Nach ENQUEUE(2):



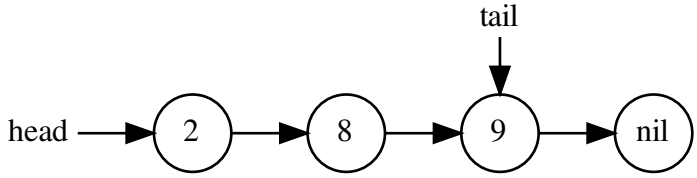
Nach DEQUEUE() = 5:



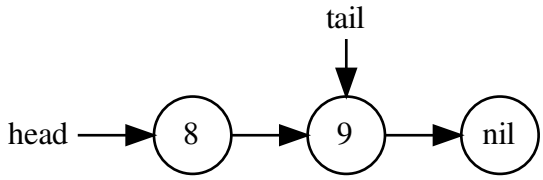
Nach ENQUEUE(8):



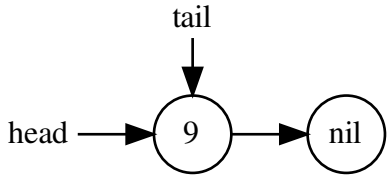
Nach ENQUEUE(9):



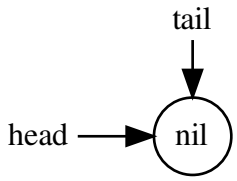
Nach DEQUEUE() = 2:



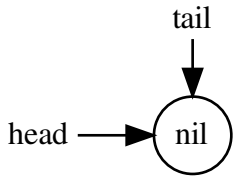
Nach DEQUEUE() = 8:



Nach DEQUEUE() = 9:



Nach DEQUEUE() = NIL:



3. Schreiben Sie Pseudocode für eine rekursive Prozedur, die alle Knoten eines gerichteten Baumes mit unbeschränktem Grad besucht und jeweils den Schlüssel des Knotens ausgibt. Nehmen Sie an, die Knoten des Baumes hätten folgende Felder: *key* für den Schlüssel, *left-child* für den Zeiger auf das sich am weitesten links befindende Kind und *right-sibling* für den Zeiger auf das rechte Geschwister. **(1 Punkt)**

```
VISIT(n)
1  if n == NIL
2       return
3  PRINT(n.key)
4  VISIT(n.left-child)
5  VISIT(n.right-sibling)
```

4. Schreiben Sie Pseudocode für eine *nicht-rekursive Prozedur*, die alle Knoten eines gerichteten Baumes mit unbeschränktem Grad besucht und jeweils den Schlüssel des Knotens ausgibt. Verwenden Sie dazu einen Stack. Nehmen Sie an, der Stack unterstützt die Operationen PUSH(Node) und POP, wobei *node* ein Knoten des Baumes ist. Der Rückgabewert von POP ist ein Knoten *node* oder NIL wenn der Stack leer ist. **(1 Punkt)**

```
VISIT-NONRECURSIVE(N)
1  S := NEW-STACK()
2  PUSH(S,N)
3  while S.length > 0
4       N := POP(S)
5       if N ≠ NIL
6            PRINT(n.key)
7            PUSH(S,n.left-child)
8            PUSH(S,n.right-sibling)
```

5. Geben Sie Pseudocode für eine Methode MERGE an, die zwei *sortierte* einfach verkettete zyklische Listen als Parameter annimmt und diese in linearer Zeit zu einer einzelnen *sortierten* Liste zusammenfügt. Die ursprünglichen Listen dürfen dabei zerstört werden und es soll nur konstant viel zusätzlicher Speicher verwendet werden.

```
MERGE(L,R)
 1  A := EMPTY-CYCLIC-LIST() // With A.tail = NIL
 2  l := L.head
 3  r := R.head
 4  while l ≠ L.nil ∨ r ≠ R.nil
 5       if r == r.nil ∨ (l ≠ l.nil ∧ l.key ≤ r.key)
 6            A.tail.next := l
 7            A.tail := l
 8            l := l.next
 9       else
10            A.tail.next := r
11            A.tail := r
12            r := r.next
13  A.tail.next := A.nil
14  return A
```

Hinweis: Es geht auch noch etwas effizienter: Wenn man am Ende der einen Liste angekommen ist, kann man den Rest der anderen Liste ans Ergebnis direkt in einem Schritt anhängen, statt alle Elemente einzeln nacheinander anzuhängen.

Wieso ist die Zeitkomplexität quadratisch statt linear, wenn der MERGE Pseudocode aus Kapitel 2, Seite 32 im Buch direkt verwendet wird, die Felder A, L, R aber durch verkettete Listen ersetzt werden? **(1 Punkt)**

Der Zugriff per Index i auf eine Linked List braucht i Schritte, statt nur konstante Zeit wie bei einem Array.

Praktische Aufgaben

In dieser Aufgabe werden Sie einen k-d-Tree implementieren. Wir stellen auf Ilias Code zur Verfügung, auf dem Sie aufbauen können. Der Code enthält eine Klasse `KDTreeTester`, welche das Programm startet und ein Fenster mit zufällig generierten Punkten anzeigt. Die Variabeln w, h und n steuern die Grösse des Fensters und die Anzahl Punkte.

Die Klasse *KDTreeVisualization* enthält verschiedene Funktionen zum Generieren und Anzeigen der zufälligen Punkte. Die Punkte werden in einer verketteten Liste gespeichert. Die Klasse enthält auch eine Funktion um die Reihenfolge der Punkte in der verketteten Liste zu visualisieren.

Eine detaillierte Beschreibung von k-d-Bäumen finden Sie auf Wikipedia.

1. Schreiben Sie eine Funktion, die für einen gegebenen Punkt seinen nächsten Nachbarn in der Liste sucht. Implementieren Sie dazu die Funktion `listSearchNN` in `KDTreeVisualization`. Sie können Ihre Funktion testen, indem Sie im Menu *Search "Search List for NN"* wählen. Die aufgerufene Funktion sucht den nächsten Nachbarn für x Punkte und misst dabei die benötigte Zeit. **(1 Punkt)**

```
public class KDTreeVisualization extends Component {
// [...]
private Point listSearchNN(Point p){
    Iterator<Point> it = points.iterator();
    double closest_sqnorm = Double.POSITIVE_INFINITY;
    Point closest = null;
    while(it.hasNext())
    {
        Point q = it.next();
        // compare squared distances (dx^2 + dy^2)
        // instead of distances (sqrt(dx^2 + dy^2))
        // saves computing the square root.
        double d = p.distanceSq(q);
        if (d < closest_sqnorm) {
            closest_sqnorm = d;
            closest = q;
        }
    }
    return closest;
}
// [...]
}
```

2. Implementieren Sie die Funktion `createKDTree`, welche die Punkte aus der Liste in einem *k*-d-Baum speichert. Nutzen Sie dazu die innere Klasse `TreeNode`. Ein Objekt dieser Klasse repräsentiert einen Knoten im *k*-d-Baum. Die Variable `kdRoot` soll die Wurzel Ihres Baumes enthalten.

   Um die Punktelisten zu sortieren, können Sie die Klasse *PointComparator* und die Java Funktion `Collections.sort(List list, Comparator c)` verwenden.

   Sie können den *k*-d-Baum mit *Visualize kd-Tree* anzeigen lassen. **(2 Punkte)**

```java
public class KDTreeVisualization extends Component {
// [...]
public void createKDTree() {
    kdRoot = createKD(new ArrayList<Point>(points), 0);
}

private TreeNode createKD(List<Point> pts, int axis) {
    if (pts.size() == 0)
        return null;
    Collections.sort(pts, new PointComparator(axis));
    int mid = pts.size() / 2;
    // make sure all elements to the left of `mid` are <= mid,
    // and all elements to the right of `mid` are > mid:
    while(mid+1 < pts.size() && pts.get(mid+1) == pts.get(mid))
        mid += 1;
    axis = (axis + 1) % 2; // axis = cycle(axis)
    var node = new TreeNode(pts.get(mid));
    node.left = createKD(pts.subList(0, mid), axis);
    node.right = createKD(pts.subList(mid+1, pts.size()), axis);
    return node;
}
// [...]
}
```

   Korrekturhinweis: Das Verschieben des Medians zum sicherstellen der Trennung zwischen kleiner-gleichen und grösSeren Elementen (oder andersrum) ist nicht gefordert, im Pseudocode hatten wir es auch der Einfachheit halber ausgelassen.

3. Schreiben Sie nun eine Funktion, welche die Suche nach dem nächsten Nachbarn auf dem *k*-d-Baum durchführt. Implementieren Sie dazu die Funktion `treeSearchNN`. Vergleichen Sie dann die Laufzeit der Suche auf dem *k*-d-Baum mit der Suche auf der Liste. Führen Sie dazu eine Suche nach dem nächsten Nachbarn für verschiedene Mengen zu suchender Punkte und mehrere unterschiedliche Punktemengen von unterschiedlicher Grösse durch. Stellen Sie die Resultate in einer Liste und grafisch dar. Stimmen Ihre Messungen mit der theoretisch erwarteten Zeitkomplexität überein (Suche in der Liste: $\mathcal{O}(n)$, Suche im Baum: $\mathcal{O}(\lg n)$)? **(2 Punkte)**

   Wir modifizieren `PointComparator` geringfügig, um die Distanz zur Trennebene direkt berechnen zu können:

```java
public class PointComparator implements
 ↪  java.util.Comparator<Point> {
// [...]

public int compare(Point a, Point b) {
```

```
    return signum(signedDistance(a,b));
}

public long signedDistance(Point a, Point b)
{
    if (dimension == 0){
        return a.x - b.x;
    } else {
        return a.y - b.y;
    }
}
// [...]
}
```

Der eigentliche Such-Code in `KdTreeVisualisation.java`:

```
public class KDTreeVisualization extends Component {
// [...]
private Point treeSearchNN(Point p){
    Point r =  doTreeSearchNN(p, kdRoot, 0, null);
    /*
       // test correctness by comparing to list search result:
       Point test = listSearchNN(p);
       if (p.distance(r) > p.distance(test)) {
       System.out.println("bad result" + p + r + test);
       }
       */
    return r;
}

private Point doTreeSearchNN(
        Point p,       // query point
        TreeNode node, // search this sub-tree
        int axis,      // axis of `node`
        Point closest) // best candidate found so far (or null)
{
    double closest_sqnorm = Double.POSITIVE_INFINITY;
    if (closest != null) {
        closest_sqnorm = p.distanceSq(closest);
    }

    var pc = new PointComparator(axis);
    var dist = pc.signedDistance(node.position, p);
    TreeNode nearSide, farSide;
    // nearSide: subtree that would contain `p`
    // farSide: the other subtree
    if (dist >= 0.0) {
        nearSide = node.left;
        farSide = node.right;
    } else {
        nearSide = node.right;
        farSide = node.left;
    }
```

```
38      if (nearSide != null) {
39          closest = doTreeSearchNN(p, nearSide, (axis+1) % 2,
             ↪  closest);
40          closest_sqnorm = p.distanceSq(closest);
41      }
42
43      // compare with point at current `node`
44      double hereDist = p.distanceSq(node.position);
45      if (hereDist < closest_sqnorm) {
46          closest = node.position;
47          closest_sqnorm = hereDist;
48      }
49
50      if (farSide != null) {
51          // if the 'far' subtree exists, see if it is possible
52          // for it to contain any points closer to the query
53          // point `p` than what we have already found.
54          // all points in `farSide` lie beyond the hyperplane,
55          // so they cannot be closer than `dist`
56          double hyperplaneDistSq = dist * dist;
57          if (hyperplaneDistSq < closest_sqnorm) {
58              // there is a chance that a point beyond the
59              // separating hyperplane is closer to the query
60              // point than `closest` is
61              closest = doTreeSearchNN(p, farSide, (axis+1) % 2,
                 ↪  closest);
62          }
63      }
64      return closest;
65  }
66  // [...]
67  }
```