Rekursion auf Listen

In dem Informatikbuch und im Zentralabitur werden Sortier-Algorithmen gerne auf Listen angewendet. Hier werden wir uns einige davon ansehen.

Sortieren duch Einfügen – iterativ auf Listen

Zur Erinnerung: Sortieren durch Einfügen auf einem Array:

Strategie: (Sortieren durch Einfügen (Insertionsort) auf einem Array)

Gegeben sei ein Feld f [1] ... f [N] mit N ≥ 2 ganzen Zahlen.

Das erste Element gilt als bereits sortiert.

Vertausche nacheinander die Elemente von 2 bis N solange mit ihren linken Nachbarn, bis sie an der korrekten Position im bereits sortierten Feld angekommen sind.

Nach N - 1 Durchläufen ist die Folge sortiert.

Sortieren durch Einfügen auf Listen:

Strategie: (Sortieren durch Einfügen (Insertionsort) auf einer Liste)

Gegeben sei ein eine Liste mit mit N ≥ 2 ganzen Zahlen.

Lege eine leere Ergebnisliste an.

Entnehme jeweils das erste Element der Originalliste und füge es an der richtigen Position der Ergebnisliste ein.

Am Ende ist die Originalliste leer und die Ergebnisliste sortiert.

Quelltext: Sortieren durch Einfügen (Insertionsort) auf einer Liste

```
public class Insertionsort {

  public static void insertionSort(List <Integer> liste){

    List <Integer> sortierteListe = new List<Integer>();

    while (!liste.isEmpty()) {
      liste.toFirst();
      int akt = liste.getContent();
      sortierteListe.toFirst();
      while (sortierteListe.hasAccess() &&
                                sortierteListe.getContent()<akt) {
        sortierteListe.next();
      }

      if (sortierteListe.hasAccess()) {
        sortierteListe.insert(akt);
      }else{
        sortierteListe.append(akt);
      }

      liste.remove();
    }

    liste.concat(sortierteListe);
  }

  public static void listeAusgeben(List <Integer> liste){
    liste.toFirst();
    while (liste.hasAccess()) {
      System.out.print(" "+liste.getContent()+"-->");
      liste.next();
    }
    System.out.println("null");
  }

  public static void main(String[] args) {

    List <Integer> liste = new List<Integer>();

    liste.append(12);
    liste.append(14);
    liste.append(11);
    liste.append(17);
    liste.append(19);
    liste.append(11);
    liste.append(18);
    liste.append(9);

    listeAusgeben(liste);
    insertionSort(liste);
    listeAusgeben(liste);
  }
```

}

Aufwandsbetrachtung: Sortieren durch Einfügen – iterativ

Zur Erinnerung: Bei dem Algorithmus auf Array-Basis ergab sich als Aufwand:

Sortieren durch Einfügen	BC	AC	WC
Vertauschungen:	$O(1)$	$O(N^2)$	$O(N^2)$
Vergleiche:	$O(N)$	$O(N^2)$	$O(N^2)$
Aufwand gesamt:	$O(N)$	$O(N^2)$	$O(N^2)$

Betrachten wir nun den Aufwand der Sortierung auf Listen:

Letztlich muss jedes Element aus der Originalliste in die neue Liste überführt werden. Damit ist der Aufwand mindestens O(N).

Best Case:

Im besten Fall liegen die Elemente in der Originalliste umgekehrt sortiert vor (also absteigend sortiert). Damit kann jeweils das erste Element entnommen und in der neuen Liste vorne eingefügt werden. Damit wäre der Aufwand in diesem Fall O(N).

Worst Case:

Im schlechtesten Fall ist die Liste bereits sortiert. Damit wird jeweils das kleinste Element aus der Originalliste entnommen und die Ergebnisliste muss komplett durchlaufen werden, um das Element hinten anzufügen. Aufwand: $O(N^2)$.

Average Case:

Im mittleren Fall muss pro Element etwa die Hälfte der Ergebnisliste durchlaufen werden, womit auch hier der Aufwand $O(N^2)$ ist.

Sortieren durch Einfügen	BC	AC	WC
Aufwand gesamt:	$O(N)$	$O(N^2)$	$O(N^2)$

Der Aufwand ist also bei beiden Varianten in etwa gleich.

Auf den folgenden Seiten werden wir eine Variante des Quicksortalgorithmus auf Listen kennen lernen. Interessanterweise ist dieser einfacher als die Variante mit Arrays.

Quicksort – rekursiv auf Listen

Zur Erinnerung: Strategie von Quicksort

Wähle in der zu sortierenden Folge das mittlere Element (als Pivot-Element) aus und verschiebe alle Elemente, die kleiner sind als dieses mittlere Element, nach links und die, die größer sind, nach rechts. Anschließend wird der gleiche Algorithmus auf die Teilfolge links angewandt und anschließend auf die Teilfolge rechts. Wenn eine Teilfolge nur noch aus einem oder aus keinem Element mehr besteht, endet die Rekursion.

Quelltext Quicksort auf Arrays: (mittleres Element ist das Pivot-Element)

```
public static void quicksort(int[] a, int linkeGrenze, int rechteGrenze) {

  int lo = linkeGrenze;
  int hi = rechteGrenze;
  int mitte;              // Stelle des mittleren Elements
  int pivot;              // Wert des mittleren Elements

  if (rechteGrenze > linkeGrenze) {

    // ein beliebiges Pivot-Element aussuchen
    mitte = (linkeGrenze + rechteGrenze) / 2;
    pivot = a[mitte];

    // wiederholen, bis lo und hi sich kreuzen
    while (lo <= hi) {

      // finde ein Element, was größer oder gleich dem Pivot-Element ist
      while ((lo < rechteGrenze) && (a[lo] < pivot))
        lo = lo+1;

      // finde ein Element, was kleiner oder gleich dem Pivot-Element ist
      while ((hi > linkeGrenze) && (a[hi] > pivot))
        hi=hi-1;

      // wenn sich die Zeiger noch nicht überkreuzt habe, tauschen
      if (lo <= hi) {
        int dummy;
        dummy = a[hi];
        a[hi] = a[lo];
        a[lo] = dummy;

        // beide Zeiger um eins weiterrücken
        lo=lo+1;
        hi=hi-1;
      }
    }
    // wenn es links vom Pivot-Element noch Elemente gibt, Rekursion
    if (linkeGrenze < hi)
      quicksort(a, linkeGrenze, hi);

    // wenn es rechts vom Pivot-Element noch Elemente gibt, Rekursion
    if (lo < rechteGrenze)
      quicksort(a, lo, rechteGrenze);
  }
}
```

Für die Implementation des Quicksort-Algorithmus auf Listenbasis ist es geschickter als Pivot-Element nicht das mittlere Element der Liste, sondern das erste Element der Liste zu verwenden. Grundsätzlich ist es egal, welches Element man nimmt.

Im schlechtesten Fall wählt man immer das größe oder immer das kleinste Element, womit die Vorteile des Quicksortalgorithmus dahin sind. Dieser ungünstige Fall hat aber nichts damit zu tun, ob man das Pivot-Element aus der Mitte oder vom Anfang entnimmt.

Strategie: Quicksort auf Listen

Die Strategie ist es, das erste Element der Liste zu entnehmen und zu löschen. Dieses Element ist das Pivot-Element.

Anschliessend werden die Elemente nach und nach der Liste entnommen, gelöscht und an zwei neue Listen angehängt. Die linke Liste enthält dann alle Elemente die kleiner sind als das Pivot-Element und die rechte Liste enthält alle Elemente, die größer sind.

Beide Teillisten werden dann rekursiv sortiert und anschließend werden die linke Liste, das Pivot-Element und die rechte Liste wieder zusammengefügt.

Sollte bei einem Aufruf die Liste nur noch aus einem oder keinem Element bestehen, so ist nichts mehr zu tun.

Quelltext Quicksort auf Listen: (erstes Element ist das Pivot-Element)

```
public class Quicksort {

  public static void quicksort(List<Integer> liste){

    if (laenge(liste) > 1) {
      List<Integer> links  = new List<Integer>();
      List<Integer> rechts = new List<Integer>();

      liste.toFirst();
      int pivot = liste.getContent();
      liste.remove();

      while(!liste.isEmpty()){
        int aktuell = liste.getContent();

        if(aktuell < pivot) {
          links.append(aktuell);
        }else{
          rechts.append(aktuell);
        }
        liste.remove();
      }
      quicksort(links);
      quicksort(rechts);
      liste.concat(links);
      liste.append(pivot);
      liste.concat(rechts);
    }
  }
```

```
  public static int laenge(List <Integer> liste){
    int laenge = 0;

    liste.toFirst();

    while (liste.hasAccess()) {
      laenge++;
      liste.next();
    }
    return laenge;
  }

  public static void listeAusgeben(List <Integer> liste){
    liste.toFirst();
    while (liste.hasAccess()) {
      System.out.print(" "+liste.getContent()+"-->");
      liste.next();
    }
    System.out.println("null");
  }

  public static void main(String[] args) {

    List<Integer> liste = new List<Integer>();

    liste.append(12);
    liste.append(14);
    liste.append(11);
    liste.append(17);
    liste.append(19);
    liste.append(19);
    liste.append(11);
    liste.append(18);
    liste.append(9);

    listeAusgeben(liste);

    quicksort(liste);

    listeAusgeben(liste);
  }
}
```

Aufwandsbetrachtung: Quicksort auf Listen

Best Case:

Im besten Fall teilt das Pivotelement die Liste in etwa zwei gleich große Listen. Damit sind ähnlich wie bei der binären Suche etwa $\log_2 N$ Rekursionsebenen vorhanden.

Auf der ersten Ebene müssen alle N Elemente betrachtet und einsortiert werden. In Ebene zwei müssen alle Elemente in der linken Liste und alle Elemente in der rechten Liste angeschaut und aufgeteilt werden usw.

Insgesamt müssen auf jeder Ebene etwa N Elemente betrachtet werden. Damit ergibt sich im Best-Case ein Aufwand von etwa $O(N \log_2 N)$.

Worst Case:

Im schlechtesten Fall ist das Pivot-Element immer das kleinste oder größe Element. Damit wird das Pivot-Element abgespalten und immer ist die linke oder rechte Liste leer und die andere enthält alle Elemente.

Damit ist die Rekursionstiefe nicht mehr nur $\log_2 N$, sonder N. Auf jeder Ebene ist immer genau ein Element weniger zu betrachten.

Damit ergibt sich im Worst-Case ein Aufwand von etwa $O(N^2)$.

Average Case:

Im durchschnittlichen Fall ergibt sich ein Aufwand von ebenfalls $O(N \log_2 N)$.

Fazit:

Insgesamt haben wir etwa den gleichen Aufwand, wie bei dem Quicksort-Algorithmus auf Array-Basis:

Vergleich der Sortierverfahren	**BC**	**AV**	**WC**
Sortieren durch Auswahl	$O(n^2)$	$O(n^2)$	$O(n^2)$
Sortieren durch Einfügen	$O(n^2)$	$O(n^2)$	$O(n^2)$
Sortieren durch Vertauschen	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

Aufgabe:

Implementiere den Algorithmus Sortieren duch Auswahl auf Listenbasis. Orientiere dich dabei an dem Verfahren Sortieren durch Einfügen. Eine Vorlage findest du im Tauschordner.

Gib zudem eine Einschätzung bezüglich der Laufzeit ab und vergleiche diese mit dem Sortierverfahren Sortieren durch Einfügen.