ALDA VT12: Sortering

Leon Hennings
leonh

Kamyar Sajjadi
kamy-saj

9 februari 2012

Innehåll

1 MergeSorter

```
1  public class MergeSorter<T extends Comparable<? super T>> extends
       Sorter<T> {
2
3    @Override
4    protected void doSort(List<T> list)
5    {
6      mergeSort(list);
7    }
8
9    /**
10    * Internal method for having fun
11    * @param list an array of Comparable items.
12    */
13   @SuppressWarnings("unchecked")
14   private void mergeSort(List<T> list)
15   {
16     T[] tmpArray = (T[]) new Comparable[list.size()];
17
18     mergeSort(list, tmpArray, 0, list.size() - 1 );
19   }
20
21   /**
22    * Internal method that makes recursive calls.
23    * @param list an array of Comparable items.
24    * @param tmpArray an array to place the merged result.
25    * @param left the left-most index of the subarray.
26    * @param right the right-most index of the subarray.
27    */
28   private void mergeSort(List<T> list, T[] tmpArray, int left, int
         right )
29   {
30     if(left < right)
31     {
32       int center = ( left + right ) / 2;
33       mergeSort(list, tmpArray, left, center );
34       mergeSort(list, tmpArray, center + 1, right );
35       merge(list, tmpArray, left, center + 1, right );
36     }
37   }
38
39  /**
40   * Internal method that merges two sorted halves of a subarray.
41   * @param List an array of Comparable items.
42   * @param tmpArray an array to place the merged result.
43   * @param leftPos the left-most index of the subarray.
44   * @param rightPos the index of the start of the second half.
45   * @param rightEnd the right-most index of the subarray.
46   *
47   * Det vi har ändrat här är att metoderna tar List<T> som argument
48   * och så har vi ändrat så get() används istället för
         indexparanterser
```

```
49     */
50    private void merge(List<T> list, T[] tmpArray, int leftPos, int
          rightPos, int rightEnd )
51    {
52      int leftEnd = rightPos - 1;
53      int tmpPos = leftPos;
54      int numElements = rightEnd - leftPos + 1;
55
56      // Main loop
57      while(leftPos <= leftEnd && rightPos <= rightEnd)
58        if(list.get(leftPos).compareTo(list.get(rightPos)) <= 0)
59          tmpArray[tmpPos++] = list.get(leftPos++);
60        else
61          tmpArray[tmpPos++] = list.get(rightPos++);
62
63      while(leftPos <= leftEnd)    // Copy rest of first half
64        tmpArray[ tmpPos++ ] = list.get(leftPos++);
65
66      while( rightPos <= rightEnd )  // Copy rest of right half
67        tmpArray[ tmpPos++ ] = list.get(rightPos++);
68
69      // Copy tmpArray back
70      for( int i = 0; i < numElements; i++, rightEnd-- )
71        list.set(rightEnd, tmpArray[rightEnd]);
72    }
73  }
```

2 QuickSorterFirstElement

```
1  public class QuickSorterFirstElement<T extends Comparable<? super T
       >> extends Sorter<T> {
2
3    private static final int CUTOFF = 10;
4
5    private void insertionSort(List<T> l, int left, int right) {
6      int j;
7      for (int p = left + 1; p <= right; p++) {
8        T tmp = l.get(p);
9        for (j = p; j > left && tmp.compareTo(l.get(j - 1)) < 0; j--)
             {
10         l.set(j, l.get(j - 1));
11       }
12       l.set(j, tmp);
13     }
14   }
15
16   /**
17    * Internal method for the sorting.
18    * @param list The list that we want to sort.
19    * @param left the left-most index of the subarray.
20    * @param right the right-most index of the subarray.
21    */
22   private void quicksort(List<T> l, int left, int right) {
23     if (left + CUTOFF <= right) {
24       T pivot = firstElement(l, left, right);
25
26       //Ändrat lite på index jämfört med
27       //orginalkoden (QuickSorterMedianThree)
28       int i = left-1;
29       int j = right+1;
30
31       for (;;) {
32         while (l.get(++i).compareTo(pivot) < 0) {
33         }
34         while (l.get(--j).compareTo(pivot) > 0) {
35         }
36         if (i < j)
37           swap(l, i, j);
38         else
39           break;
40       }
41
42       swap(l, i, right); // restore pivot
43
44       quicksort(l, left, i-1); // sort small elements
45       quicksort(l, i, right); // sort large elements
46     } else
47     {
48       insertionSort(l, left, right);
49     }
```

```
50    }
51
52    /**
53     *  Method to pick the pivot.
54     *  returns the first element in List l from interval
55     *  left to right and swaps it to the last place.
56     *  @return T
57     */
58    private T firstElement(List<T> l, int left, int right)
59    {
60      T first = l.get(left);
61      swap(l, left, right);
62      return first;
63    }
64
65    @Override
66    protected void doSort(List<T> l)
67    {
68      quicksort(l, 0, l.size() - 1);
69    }
70  }
```

3 QuickSorterRandom

```
1  public class QuickSorterRandom<T extends Comparable<? super T>>
       extends Sorter<T> {
2
3    private static final int CUTOFF = 10;
4
5    private void insertionSort(List<T> l, int left, int right) {
6      int j;
7      for (int p = left + 1; p <= right; p++) {
8        T tmp = l.get(p);
9        for (j = p; j > left && tmp.compareTo(l.get(j - 1)) < 0; j--)
             {
10         l.set(j, l.get(j - 1));
11       }
12       l.set(j, tmp);
13     }
14   }
15
16   /**
17    * Internal method for the sorting.
18    * @param list The list that we want to sort.
19    * @param left the left-most index of the subarray.
20    * @param right the right-most index of the subarray.
21    */
22   private void quicksort(List<T> l, int left, int right)
23   {
24     if (left + CUTOFF <= right) {
25       T pivot = random(l, left, right);
26
27       int i = left-1;
28       int j = right+1;
29
30       for (;;) {
31         while (l.get(++i).compareTo(pivot) < 0) {
32         }
33         while (l.get(--j).compareTo(pivot) > 0) {
34         }
35         if (i < j)
36           swap(l, i, j);
37         else
38           break;
39       }
40
41       swap(l, i, right); // restore pivot
42
43       quicksort(l, left, i-1); // sort small elements
44       quicksort(l, i, right); // sort large elements
45     } else
46     {
47       insertionSort(l, left, right);
48     }
49   }
```

```
50
51    /**
52     *  Method to pick the pivot.
53     *  returns a random placed T from the List i the interval
54     *  left to right and swaps it to the last place.
55     *  @return T
56     */
57    private T random(List<T> l, int left, int right)
58    {
59      Random rnd = new Random();
60      int pivoNr = rnd.nextInt(right - left + 1)+left;
61      swap(l, pivoNr, right);
62      return l.get(right);
63    }
64
65    @Override
66    protected void doSort(List<T> l)
67    {
68      quicksort(l, 0, l.size() - 1);
69    }
70  }
```

4 Analys av sorteringalgoritmer

4.1 Algoritmer

Mergesort är en sorteringsalgortim som rekursivt bryter upp det som ska sorteras till mindre listor och sedan sorteras de mindre listorna. Därefter kommer dessa sättas samman och skapa en lista som är sorterad. Innan vi börjat köra våra testfall så diskutera vi vad vi trodde skulle ta längst tid att sortera, en arraylist eller en linkedlist. Vi kom ganska snabbt fram till att en länkadlista skulle ta längre tid att sortera än den förstnämnda. Enligt teorin är mergesort en algoritm som ska ta O(N log N). Tillskillnad från quicksort som är en in-place algoritm använder sig Mergesort av mer minne när den jobbar med komplexa datatyper. Detta då den behöver skapa en extra datastruktur som tar upp mer plats i minnet.

Quicksort väljer ut ett element som blir vridpunkten (pivot) och placerar detta i slutet av listan, i det här fallet väljs element slumpmässigt. Vridpunkten kan väljas ut på olika sätt. De olika sätt vi har implementerat valet av vridpunkten är med hjälp av en randomgenerator som väljer ut ett index ur listan och sätter pivot till det element som ligger på det indexet. Den andra versionen vi implementerat går ut på att vi väljer ut det första värdet som ligger i listan och använder det som vridpunkt. Sedan finns det ytterligare ett sätt att välja privovärdet som går ut på att räkna fram medianen av det första, sista och det elementet i mitten. I quicksort så söks listan igenom från vänster efter element som är större än vridpunkten och sedan från höger i jakt på element som är mindre än vridpunkten. När två element som uppfyller detta hittats byter de plats. Detta fortsätter tills jämförelsepekaren som började från höger(j) är på ett lägre index än den som började från vänster(i), då byts vridpunkten mot det element som i pekar på. Därefter kallas quicksort rekursivt på elementen till höger och de till vänster om vridpunkten. Det är vanligt att använda sig av en annan sorteringsalgoritm för små sublistor och i det här fallet används insertionsort. Detta för att på mindre datamängder är simpla sorteringsalgoritmer mer effektiva. Detta fungerar som basecase för rekursionen.

Insertionsort är en enkel sorteringsalgoritm. Oftast använder sig andra mer avancerade sorteringsalgoritmer sig av denna sortering som basecase, tex som vi nämnde ovan quicksort. Algoritmen är relativt snabb att använda för att sortera små datastrukturer. Det absolut bästa fallet för insertionsort är när det som ska sorteras redan är sorterat. Då kommer algoritmen ha en tidskomplexitet på O(N). Även när listan är relativt sorterad så kommer insertionsort att utföras snabbt. Själva implementationen består av en yttre loop som börjar på det andra elementet i listan och för varje iteration av den yttre loopen jämförs det med alla tidigare element tills något större hittas varpå elementen byter plats. I värsta fall, t ex om hela listan är sorterad i omvänd ordning, tar insertionsort O(N^2) tid.

Selectionsort är likt insertionsort en O(N^2) sorteringsalgoritm. Selectionsort hittar det minsta värdet och byter plats med det värdet i första indexet. Sedan så går den vidare till nästa index och hittar det näst största värdet och byter plats. Detta görs med hela listan tills listan är helt sorterad. Implementationen som Henrik har bifogat ser ut på följande sätt. Den yttre loopen börjar på första elementet, på index x, och kopierar det till ett temporärt element. För varje varv söker den igenom hela resten av listan efter det minsta värdet och placerar det på platsen x varpå det temporära elementet tar det minsta elementets tidigare plats. Sedan fortsätter yttre loopen på nästa plats i listan och inre loopen söker efter det näst lägsta värdet, och så vidare till listans slut nås. Selectionsort är som insertionsort snabb på att sortera små listor. Det som skiljer sig är att selectionsort alltid är O(N^2) tillskillnad från insertionssort som i bästa fall är O(N).

4.2 Jämförelse

För att kunna resonera kring hur mycket extra tid som går åt när vi sorterar länkade listor gjorde vi både quicksortRandom och quicksortFirstElement med iteratorer och utan. Här ser vi tydligt att de med iteratorer inte helt oväntat är snabbare än de utan när sorteringen görs på en länkad lista. Det mesta av tiden går alltså åt till att iterera över listan när man inte har en iterator. Som exempel tog quicksortRandom med iterator 2,3 sekunder för att sortera en lista på

1000 element med många dubbletter av komplexa dataobjekt i slumpad ordning. Samma sortering tog det 8,88 sekunder för quicksortRandom utan iterator att utföra. Att iterera över en länkad lista utan iterator innebär att man måste stega mellan objekten lika många gånger som platsen i listan man är ute efter varje varv. Att enbart hämta det sista objektet i en (single)linked list har samma tidskomplexitet som att iterera över en arraylista av samma storlek. Detta gör t ex att en genomsnittlig toString metod skulle ta O(N^2) för att skriva ut alla element i listan. När det är en ArrayList som sorteras är iteratorn enbart negativ och de tar då istället längre tid.

Mergesort tar lång tid när de komplexa datatyperna ska sorteras. Största skillnaden uppstod när listan var slumpmässigt fylld och oändrad. Med 10000 element tog det då 2,26 sekunder att sortera heltalen och 10,08 sekunder att sortera komplexa datatyperna. Quicksort lider inte lika mycket av detta då den gör sorteringen in-place. I våra testfall var quicksort snabbare än mergesort i nästan varenda fall. Eftersom de komplexa datatyperna är arrayer med heltal kommer de behöva jämföra alla platser för de fallen som de är dubbletter varje gång det görs en compareTo. Detta drabbar quicksort hårdare än mergesort eftersom den gör fler jämförelser, dock vägs det upp av att de större datafälten tar mer tid att kopiera in i mergesorts temporära array.

Enligt vår utdata från testfallen så fungerar quicksort bättre än mergesort när det är en slumpad lista med få dubbletter av heltalsobjekt som ska sorteras. För 100000 element tog det 477 sekunder för mergesort och 316 sekunder för quicksort. När listan var slumpad och det var få dubbletter gick det fortare att sortera med mergesort. Mergesort tog då 7,09 sekunder medans quicksort tog 9,43 sekunder för 10000 element.

När det gäller de enkla sorteringsalgoritmerna är Selectionsort snabbare än både insertionsort och bubblesort i de flesta fallen, utom när listan redan är sorterad. Insertionsort och bubblesort drar jämnt strå när det gäller sorterade listor då båda är O(N) i best case. När listan inte är sorterad är bubblesort alltid långsammast.

Jämförelser med sorteringsalgoritmen som finns implementerad för collections i Java visade sig vara lönlösa då den alltid gick mycket snabbare. Endast quicksort med iterator kom i närheten men skillnaden var så stor så det inte gick att dra slutsatser om vilken sökalgoritm som användes.

5 Poolfråga

5.1 Fråga 1

Quicksort och mergesort är två vanliga sorteringsalgoritmer som båda har genomsnittliga tidskomplexiteten O(n log n). Ge exempel på fall då den ena är att föredra över den andra. För högre betyg ska du även kunna säga varför med hänsyn till hur de är implementerade.

5.2 Fråga 2

Vissa sorteringsalgoritmer innehåller ofta en implementation av en annan sorteringsalgoritm. Ge två exempel på sådana sorteringsalgoritmer. För högre betyg ska du även förklara varför och i vilket skede av sorteringen som de andra sorteringsalgoritmerna används och ge exempel på algoritmer som vanligen används i detta syfte.

5.3 Fråga 3

Ge ett exempel på ett fall då de enkla sorteringsalgoritmerna insertionsort och bubblesort är att föredra över quicksort, även för listor av större storlek. För högre betyg ska du kunna visa på en liten lista med ca 10 element hur de skulle sorteras med de olika sorteringsalgoritmerna samt hur stor skillnaden på antal operationer som görs blir.