ALDA VT12: Linjära datastrukturer samt introduktion till algoritmanalys

Leon Hennings
leonh

Kamyar Sajjadi
kamy-saj

19 januari 2012

1 Lazy Deletion

```
1  import java.util.Collection;
2  import java.util.ConcurrentModificationException;
3  import java.util.Iterator;
4  import java.util.List;
5  import java.util.ListIterator;
6  import java.util.NoSuchElementException;
7
8  /**
9   * Dokumentationen for metoderna finns i interfacet.
10  * @author henrikbe
11  */
12 public class SimpleLinkedList<E> implements List<E>
13 {
14
15    private static class Element<E>
16    {
17       public E data;
18       public Element<E> prev;
19       public Element<E> next;
20       public boolean deleted = false;
21
22       public Element()
23       {
24       }
25
26       public Element(E data, Element<E> prev, Element<E> next)
27       {
28          this.data = data;
29          this.prev = prev;
30          this.next = next;
31          prev.next = this;
32          next.prev = this;
33
34       }
35    }
36
37    private class SimpleLinkedListIterator implements ListIterator<E>
```

```
38      {
39
40          private Element<E> current;
41          private int currentIndex;
42          private int expectedModCount;
43
44          public SimpleLinkedListIterator(Element<E> current, int currentIndex)
45          {
46              this.current = current;
47              this.currentIndex = currentIndex;
48              this.expectedModCount = modCount;
49          }
50
51          @Override
52          public void add(E element)
53          {
54              throw new UnsupportedOperationException("add is not supported");
55          }
56
57          /*
58           * hasNext() kommer att returnera true om current
59           * element har en next som ej är deleted. Om next
60           * är markerad som deleted så ska den gå vidare
61           * till nästa element. Detta kommer att ske tills
62           * vi kommer till ett objekt som ej är marketat som
63           * deleted och ej är tail:n.
64           */
65          @Override
66          public boolean hasNext()
67          {
68              Element<E> tmp = current;
69
70              while(tmp.next != tail && tmp.next.deleted)
71              {
72                  tmp = tmp.next;
73              }
74              if(tmp.next != tail)
75                  return true;
76              else
77                  return false;
78          }
79
80          /*
81           * Denna metod är i stortsätt som hasNext().
82           *  Skillnaden är att den kollar om det finns en
83           * prev i current. Den gör givetvis samma kontroll
84           * så att inte prev är deleted, om den är flaggad
85           * som deleted så går den vidare och kollar föregående.
86           */
87          @Override
88          public boolean hasPrevious()
89          {
90              Element<E> tmp = current;
91
```

```
92              while(tmp != head && tmp.prev.deleted)
93              {
94                  tmp = tmp.prev;
95              }
96              if(tmp != head)
97                  return true;
98              else
99                  return false;
100         }
101
102         /*
103          * next() returnera <E> från nästa Element. Om nästa
104          * är flaggad som deleted så kommer den att loopa
105          * tills ett element hittas som ej är flaggat som deleted.
106          * När elementet är hittat så kommer iteratorns
107          * "pekare" att ändras fram ett steg.
108          */
109         @Override
110         public E next()
111         {
112             if (!hasNext())
113                 throw new NoSuchElementException();
114             if (modCount != expectedModCount)
115                 throw new ConcurrentModificationException();
116             do
117             {
118                 current = current.next;
119             } while (current.deleted);
120
121             currentIndex++;
122             return current.data;
123         }
124
125         @Override
126         public int nextIndex()
127         {
128             return currentIndex;
129         }
130
131         /*
132          * Metoden returnerar föregåendes data och flyttar
133          * iteratorn ett steg bakåt. Om det föregående elementet
134          * är deleted så ska iteratorn ej ställa sig på det
135          * utan den ska loopa tills den hittar ett objekt som
136          * ej är deleted och ställa sig där.
137          */
138         @Override
139         public E previous()
140         {
141             if (!hasPrevious())
142                 throw new NoSuchElementException();
143             if (modCount != expectedModCount)
144                 throw new ConcurrentModificationException();
145             E data = current.data;
```

```
146             do
147             {
148                 current = current.prev;
149             } while (current.deleted); //fortsätt så länge deleted är sant
150 
151             currentIndex--;
152             return data;
153         }
154 
155         @Override
156         public int previousIndex()
157         {
158             return currentIndex - 1;
159         }
160 
161         @Override
162         public void remove()
163         {
164             throw new UnsupportedOperationException("remove is not supported");
165         }
166 
167         @Override
168         public void set(E element)
169         {
170             throw new UnsupportedOperationException("set is not supported");
171         }
172     }
173 
174     private int size;
175     private int modCount;
176     private int sumDeleted; // Summan av samtliga som är flaggade som deleted
177     private Element<E> head;
178     private Element<E> tail;
179 
180     public SimpleLinkedList()
181     {
182         clear();
183     }
184 
185     private void checkIndex(int index, int upperBoundary)
186     {
187         if (index < 0 || index > upperBoundary)
188             throw new IndexOutOfBoundsException(String.format(
189                     "Illegal index %d. Acceptable range is 0 to %d", index,
190                     upperBoundary));
191     }
192 
193     /*
194      * Hämtar det Element som ligger på det angivna
195      * indexet. Den börjar i head.next, dvs det
196      * första Elementet i vår länkade lista. Sedan
197      * så stegar den fram tills den kommit till
198      * det angivna indexet. Om Elementet är flaggat
199      * som deleted så kommer index att ökas med 1.
```

```
200      * Detta innebär att ett extra varv kommer att
201      * köras för varje Element som är deleted.
202      */
203     private Element<E> getElement(int index)
204     {
205         Element<E> temp = head;
206         for (int n = 0; n < index; n++)
207         {
208             temp = temp.next;
209             if(temp.deleted)
210                 index++;
211         }
212         return temp;
213     }
214 
215     @Override
216     public boolean add(E element)
217     {
218         new Element<E>(element, tail.prev, tail);
219         size++;
220         modCount++;
221         return true;
222     }
223 
224     @Override
225     public void add(int index, E element)
226     {
227         checkIndex(index, size());
228         Element<E> temp = getElement(index);
229         new Element<E>(element, temp, temp.next);
230         size++;
231         modCount++;
232     }
233 
234     @Override
235     public boolean addAll(Collection<? extends E> c)
236     {
237         for (E element : c)
238             add(element);
239         return c.size() > 0;
240     }
241 
242     @Override
243     public boolean addAll(int index, Collection<? extends E> c)
244     {
245         for (E element : c)
246             add(index++, element);
247         return c.size() > 0;
248     }
249 
250     /*
251      * I denna metod har vi lagt till sumDeleted
252      */
253     @Override
```

```
254     public void clear()
255     {
256         head = new Element<E>();
257         tail = new Element<E>();
258         head.next = tail;
259         tail.prev = head;
260         size = 0;
261         sumDeleted = 0; // sumDeleted sätts till 0
262         modCount++;
263     }
264 
265     @Override
266     public boolean contains(Object o)
267     {
268         for (E element : this)
269             if (o == null ? element == null : o.equals(element))
270                 return true;
271         return false;
272     }
273 
274     @Override
275     public boolean containsAll(Collection<?> c)
276     {
277         for (Object o : c)
278             if (!contains(o))
279                 return false;
280         return true;
281     }
282 
283     @Override
284     public E get(int index)
285     {
286         checkIndex(index, size() - 1);
287         return getElement(index + 1).data;
288     }
289 
290     @Override
291     public int indexOf(Object o)
292     {
293         int index = 0;
294         for (E element : this)
295         {
296             if (o == null ? element == null : o.equals(element))
297                 return index;
298             else
299                 index++;
300         }
301         return -1;
302     }
303 
304     @Override
305     public boolean isEmpty()
306     {
307         return size() == 0;
```

```
308     }
309
310     @Override
311     public Iterator<E> iterator()
312     {
313        return listIterator();
314     }
315
316     @Override
317     public int lastIndexOf(Object o)
318     {
319        ListIterator<E> iterator = listIterator(size());
320        while (iterator.hasPrevious())
321        {
322           E element = iterator.previous();
323           if (o == null ? element == null : o.equals(element))
324              return iterator.nextIndex();
325        }
326        return -1;
327     }
328
329     @Override
330     public ListIterator<E> listIterator()
331     {
332        return new SimpleLinkedListIterator(head, 0);
333     }
334
335     @Override
336     public ListIterator<E> listIterator(int index)
337     {
338        checkIndex(index, size());
339        return new SimpleLinkedListIterator(getElement(index), index);
340     }
341
342     @Override
343     public boolean remove(Object o)
344     {
345        int index = indexOf(o);
346        if (index >= 0)
347        {
348           remove(index);
349           return true;
350        }
351        else
352        {
353           return false;
354        }
355     }
356
357     /*
358      * Markerar elementet som deleted samt att den tar
359      * bort datan som elementet håller. När hälften av
360      * listan är markerad som deleted så kommer den att
361      * länka om samtliga element.
```

```
362      */
363     @Override
364     public E remove(int index)
365     {
366         checkIndex(index, size() - 1);
367         Element<E> removed = getElement(index+1);
368         E data = removed.data;
369 
370         getElement(index+1).data = null;
371         getElement(index+1).deleted = true;
372         sumDeleted++;
373         modCount++;
374         size--;
375 
376         if(sumDeleted >= (size()+sumDeleted)/2)
377             deleteAllDeleted();
378 
379         return data;
380     }
381 
382     /*
383      * Metod för att länka om alla element som är deleted.
384      * Om next är flaggad som deleted så tar den nästa
385      * element efter next och ändrar dens pekare till det
386      * nuvarande elementet. När omlänkningen är klar så
387      * sätts sumDeleted till 0.
388      */
389     private void deleteAllDeleted()
390     {
391         Element<E> element = head;
392 
393         while(element != tail)
394         {
395             if(element.next.deleted)
396             {
397                 element.next.next.prev = element;
398                 element.next = element.next.next;
399             }
400             element = element.next;
401         }
402         sumDeleted = 0;
403     }
404 
405     @Override
406     public boolean removeAll(Collection<?> c)
407     {
408         boolean changed = false;
409         for (Object o : c)
410         {
411             while(remove(o))
412             {
413                 changed=true;
414             }
415         }
```

```
416         return changed;
417      }
418
419      @Override
420      public boolean retainAll(Collection<?> c)
421      {
422         boolean changed = false;
423         int n = 0;
424         while (n < size())
425         {
426            if (c.contains(get(n)))
427            {
428               n++;
429            }
430            else
431            {
432               remove(n);
433               changed = true;
434            }
435         }
436         return changed;
437      }
438
439      @Override
440      public E set(int index, E newElementValue)
441      {
442         checkIndex(index, size() - 1);
443         Element<E> e = getElement(index + 1);
444         E oldValue = e.data;
445         e.data = newElementValue;
446         return oldValue;
447      }
448
449      @Override
450      public int size()
451      {
452         return size;
453      }
454
455      public String toString()
456      {
457         StringBuilder buffer = new StringBuilder();
458
459         buffer.append("[");
460         Iterator<E> iter = iterator();
461         while (iter.hasNext())
462         {
463            buffer.append(iter.next());
464            if (iter.hasNext())
465               buffer.append(", ");
466         }
467         buffer.append("]");
468
469         return buffer.toString();
```

```
470     }
471
472     // Har nedanfor bryter vi mot kontraktet for listan. Metoderna ar inte
473     // "optional", men de tillfor inget till uppgiften.
474
475     @Override
476     public List<E> subList(int fromIndex, int toIndex)
477     {
478         throw new UnsupportedOperationException();
479     }
480
481     @Override
482     public Object[] toArray()
483     {
484         throw new UnsupportedOperationException();
485     }
486
487     @Override
488     public <T> T[] toArray(T[] arg0)
489     {
490         throw new UnsupportedOperationException();
491     }
492
493 }
```

2 Två stackar i en array

```
1  import java.util.EmptyStackException;
2  public class DoubleStack<E>{
3      /*
4       *En array som ska innehålla de två stackarna
5       *En int som ska ska hålla index för första platsen i den andra stacken
6       */
7      private E[] stackar;
8      private int stackBorder;
9      /*
10      *Konstruktor som initierar stackar till en generisk array mha cast.
11      *@SuppressWarnings för att ignorera varningen castet genererar.
12      *Arrayens storlek sätts till 2.
13      *stackBorder sätts till 1 för att indikera index motsvarande andra
14        *stackens första plats.
15      */
16     @SuppressWarnings({"unchecked"})
17         public DoubleStack(){
18             stackar = (E[]) new Object[2];
19             stackBorder = 1;
20         }
21     /*
22      *pushToFirst tar ett element och lägger in det överst i den första stacken.
23      *Första halvan av arrayen itereras över från index 0 tills en plats hittas
24      *som är null varpå elementet placeras där.
25      *Ifall ingen tom plats hittas kallas expand för att skapa mer utrymme,
26      *Därefter itererar man igen över stacken och lägger elementet i den första
27      *av de tomma platserna.
```

```
28          */
29      public void pushToFirst(E elem){
30          boolean bool = true;
31          while(bool){
32              for(int i = 0; i<stackBorder ; i++){
33                  if(stackar[i]==null){
34                      stackar[i]=elem;
35                      bool = false;
36                  }
37              }
38              if(bool)
39                  expand();
40          }
41      }
42      /*
43       *Fungerar på samma sätt som pushToFirst med skillnaden att den börjar
44       *iterera över arrayen från stackBorder till slutet av den.
45       */
46      public void pushToSecond(E elem){
47          boolean bool = true;
48          while(bool){
49              for(int i = stackBorder; i<stackar.length; i++){
50                  if(stackar[i]==null){
51                      stackar[i]=elem;
52                      bool = false;
53                  }
54              }
55              if(bool)
56                  expand();
57          }
58      }
59
60      /*
61       *expand kallas från push metoderna för att göra plats för ytterligare
62       *element när någon av stackarna är fulla.
63       *En temporär array, dubbelt så stor som stackar[] skapas och värdena
64         *kopieras till den. Detta görs i en for loop som gör hälften så många
65         *iterationer som stackars längd. För varje iteration läggs elementen
66         *på n:te platsen i stackarna ett och två till i temp.
67       *@SuppressWarnings används även här för att ignorera castet av temp arrayen.
68       *Slutligen dubblas stackBorder och stackar sätts till temp.
69       */
70      @SuppressWarnings({"unchecked"})
71          private void expand(){
72              E[] temp = (E[]) new Object[stackar.length*2];
73              for(int i = 0; i<stackBorder ; i++){
74                  temp[i]=stackar[i];
75                  temp[(stackBorder*2)+i] = stackar[stackBorder+i];
76              }
77              stackBorder = stackBorder*2;
78              stackar = temp;
79          }
80
81      /*
```

```
82        *peekFirst returnerar det senaste pushade värdet i första stacken.
83        *Den itererar bakifrån från slutet på stack ett tills den
84        *stöter på en plats som inte är tom.
85        *Om inget element påträffas kastas EmptyStackException.
86        */
87       public E peekFirst(){
88           for(int i=stackBorder-1; i>=0;i--){
89               if(stackar[i]!=null)
90                   return stackar[i];
91           }
92           throw new EmptyStackException();
93       }
94
95       /*
96        *Samma som peekFirst fast på andra stacken.
97        *Börjar iterera från slutet av arrayen istället.
98        */
99       public E peekSecond(){
100          for(int i=stackar.length; i>=stackBorder-1; i--){
101              if(stackar[i]!=null)
102                  return stackar[i];
103          }
104          throw new EmptyStackException();
105      }
106
107      /*
108       *popFromFirst itererar över första stacken på samma sätt som peekFirst.
109       *Enda skillnaden är att elementet tas bort innan det returneras,
110       *därav temp elementet.
111       */
112      public E popFromFirst(){
113          E temp=null;
114          for(int i=stackBorder-1; i>=0;i--){
115              if(stackar[i]!=null){
116                  temp=stackar[i];
117                  stackar[i] = null;
118                  return temp;
119              }
120          }
121
122          throw new EmptyStackException();
123      }
124
125      /*
126       *popFromSecond fungerar som popFromFirst fast självfallet med elementen
127       *från andra stacken.
128       */
129      public E popFromSecond(){
130          E temp=null;
131          for(int i=stackar.length-1; i>=stackBorder-1; i--){
132              if(stackar[i]!=null){
133                  temp=stackar[i];
134                  stackar[i] = null;
135                  return temp;
```

```
136             }
137         }
138         throw new EmptyStackException();
139     }
140 
141     /*
142      *firstIsEmpty kollar om första stacken är tom på element.
143      *Ifall elementet i botten av stacken är null innebär det
144      *att hela stacken är tom varpå false returneras.
145      */
146     public boolean firstIsEmpty(){
147         return stackar[0]==null;
148     }
149 
150     /*
151      *secondIsEmpty gör samma sak som firstIsEmpty men använder
152      *stackBorder som index för jämförelsen, för att komma åt
153      *första elementet i andra stacken.
154      */
155     public boolean secondIsEmpty(){
156         return stackar[stackBorder]==null;
157     }
158     /*
159      *searchFirst letar igenom stacken efter ett objekt och returnerar
160      *en int motsvarande hur långt ner i stacken det ligger, översta
161      *elementer får nummer ett, osv.
162      *Detta görs genom iteration från toppen av stacken neråt.
163      */
164     public int searchFirst(Object o){
165         int distance=0;
166         for(int i=stackBorder-1; i>=0;i--){
167             if(stackar[i]!=null){
168                 distance++;
169                 if(o.equals(stackar[i]))
170                     return distance;
171             }
172         }
173         return -1;
174     }
175 
176     /*
177      *searchSecond fungerar på samma vis som searchFirst fast på
178      *andra stacken.
179      */
180     public int searchSecond(Object o){
181         int distance=0;
182         for(int i=stackar.length-1; i>=stackBorder-1; i--){
183             if(stackar[i]!=null){
184                 distance++;
185                 if(o.equals(stackar[i]))
186                     return distance;
187             }
188         }
189         return -1;
```

```
190         }
191
192 }
```

3 Algoritmanalys

Denna loop itererar n gånger när n=10 så blir summan 10. Det innebär att komplexiteten är linjär O(N). Vid 10 iterationer tar det ca 600 nanosekunder. När vi ökar n till 100 tar det ca 1300 nanosekunder.

```
1 //Exempel 1
2 int sum = 0 ;
3 for(int i = 0; i<n ;i++)
4     sum++;
```

Exempel 2 kommer att ökas kvadratiskt O(N^2). Eftersom det är två loopar som har O(N) dvs är linjära så blir tidskomplexiteten N * N = N^2. För varje varv i den övre loopen kommer den undre loopen gå n gånger.

Vid mätning med nanosekunder ger n=10 ca 2900 nanosekunder samt n=100 tar det ca 160000 nanosekunder.

```
1 //Exempel 2
2 int sum = 0 ;
3 for (int i = 0 ; i<n ; i++)
4     for (int j = 0; j<n; j++)
5         sum++;
```

Första loopen är O(N) och för varje varv i den så går den inre loopen n * n varv vilket ger tidskomplexiteten N^2. O(N) * O(N^2) = O(N^3).

n = 10 tar 22000 nanosekunder

n = 100 tar 15100000 nanosekunder

```
1 //Exempel 3
2 int sum = 0 ;
3 for (int i=0; i<n; i++)
4     for (j=0; j<n*n; j++)
5         sum++;
```

Den yttre loopen går n varv, O(N). För varje varv i den yttre loopen går inre loopen i varv, där i är så många varv den yttre loopen gått. Då i närmar sig n kommer den inre loopen som mest gå n iterationer. Eftersom man i tidskomplexitets beräkning utgår från det värsta fallet så blir även den O(N). O(N) * O(N) = O(N^2).

n = 10 tar 2700 nanosekunder

n = 100 tar 103000 nanosekunder

```
1 //Exempel 4
2 int sum = 0 ;
3 for (int i = 0; i<n; i++)
4     for (int j = 0; j<i ; j++)
5         sum++;
```

Första loopen är O(N). Den andra loopen är O(N^2) då den är beroende av den första loopens iterationer. Dvs den går exponentiellt så många varv som den första loopen har gått. Den tredje loopen går lika många varv som den andra loopen och är O(N^2). O(N) * O(N^2) * O(N^2) = O(N^5)

n = 10 tar 230000 nanosekunder

n = 100 tar 700000000 nanosekunder

```
1 //Exempel 5
2 int sum = 0 ;
```

```
3 for (int i=0; i<n ; i++)
4     for (int j=0; j<i*i ; j++)
5         for (int k=0; k<j; k++)
6             sum++;
```

Den första och andra loopen är lika som i exempel 5 dvs tillsammans O(N) * O(N^2) = O(N^3). Villkoret i den andra loopen uppfylls endast när j är jämt delbart med i.

Eftersom den andra loopen kommer iterera tills j är likamed i^2 kommer den tredje loopen köras i-1 gånger per genom loopning av andra loopen.

Då i närmar sig n kommer det resultera i att ifsatsens vilkor kommer uppfyllas n-1 gånger, worst case.

Exempelvis, när i är 5 kommer den andra loopen gå 25 varv och i intervallet 0<=j<=24 kommer j vara jämnt delbart med i 4 gånger: j=5,10,15,20.

Den sista loopen kommer då köras och den är som i exempel 5 O(N^2). Detta innebär att eftersom den endast kör de gånger if satsen är sann så kommer det resultera i en tidskomplexitet av O(N^2)/O(N) = O(N).

Den totala tidskomplexiteten bli då O(N) * O(N^2) * O(N) = O(N^4)

n = 10 tar 37000 nanosekunder

n = 100 tar 28600000 nanosekunder

```
1 //Exempel 6
2 int sum = 0 ;
3 for (int i=1; i<n; i++)
4     for (int j=1; j<i*i ; j++)
5         if ( j % i == 0 )
6             for (int k=0; k<j; k++)
7                 sum++;
```

4 Poolfråga

Lifo är ett begrepp som nämns i samband med stackar. Vad innebär det? Ge exempel på situationer då stackar är lämpliga och där de inte är lämpliga att använda istället för t ex array eller länkad lista. Visa hur man med hjälp av en stack kan göra Depth-first search i ett binärträd och varför lifo är fördelaktigt i den här situationen.

För godkänt betyg måste du förklara lifo och kunna ge exempel på tillfällen då stackar är bra eller dåliga att använda. Dessutom bör grundläggande förståelse för hur en algoritm för depth-first serach uppvisas. För högre betyg ska lämpliga argument ges på fördelar med att använda lifo i depth-first search. Även en bra implementation ska beskrivas.