ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP – SS 2018
Übungsabgabe 9

Konstantin Papesh

6. Juni 2018

9.1 Neue Version des "Behälters" Vector - diesmal mit OOP

9.1.1 Lösungsidee

Es ist ein Behälter mithilfe eines Vectors zu implementieren. Doch anstatt wie in *Übung 5* einen Datentyp dafür zu definieren wird diesmal mithilfe einer Klasse gearbeitet. Diese bietet ein standardisiertes Interface und vereinfacht mithilfe von Information Hiding die Sicherstellung der Integrität des Vectors. Auch kann mithilfe der *Init*-Funktion, welche als Konstruktor verwendet wird, eine initiale Kapazität festgelegt werden.

9.1.2 Implementierung

Siehe 9.2.2

9.1.3 Ausgabe

Siehe 9.2.3

9.2 "Beschränkte" Vektoren

9.2.1 Lösungsidee

Der in 9.1 erstellte Vektor soll erweitert werden, damit dieser nur bestimmte Zahlen aufnimmt. Im Fall von *a)* nur natürliche und im Fall *b)* nut Primzahlen. Dafür wird zuerst die Grundklasse *vector* geerbt, damit die Methoden dieser verwendet werden können. Dann werden die Funktionen *add* und *insertElementAt* neu implementiert.[1] Dies ist so zu bewerkstelligen, dass der übergebene *val*-Wert zuerst überprüft wird, ob er in den Vektor eingefügt werden darf.[2] Danach wird die ursprüngliche Funktion von *vector* mit *inherited FUNKTION(val)* aufgerufen, denn das Einfügen des Wertes ist ident mit der originalen Funktion, daher muss dies nicht nocheinmal ausgecoded werden.

a)

In der Funktion *isNatural* wird lediglich überprüft, ob $val >= 0$ laut Definition der natürlichen Zahlen.

b)

Die Funktion für das Ermitteln von Primzahlen ist dabei schon komplexer. Zuerst wird überprüft, ob $val <= 1$, da Zahlen kleiner gleich 1 keine Primzahlen sein können. Danach wird der Wert *val* von 2 bis $val/2 - 1$ dividiert, sobald irgendwo kein Rest bleibt ist die Zahl keine Primzahl.

9.2.2 Implementierung

Listing 9.1: vector.pas

```
1  program V_AV;
2
3  const DEFAULT_CAP_SIZE = 16;
4
5  type
6      intArray = array[1..1] of integer;
7      Vector = ^VectorObj;
8      VectorObj = OBJECT
9          PRIVATE
10             arrPtr : ^intArray;
11             capacityCount : integer;
12             top : integer; //equals size
13             initCapacity : integer;
14             PUBLIC
15                 constructor init(userCapacity : integer);
16                 destructor done;
17                 procedure add(val : integer); virtual;
18                 procedure insertElementAt(pos : integer; val : integer); virtual;
```

[1] Auch *Polymorphismus*
[2] Siehe ?? und ??

```
19                  procedure getElementAt(pos : integer; var val : integer; var ok :
      boolean);
20                  function size : integer;
21                  function capacity : integer;
22                  procedure clear;
23              PRIVATE
24                  procedure realloc;
25                  function isOutOfRange(pos : integer) : boolean;
26      end;
27      NaturalVector = ^NaturalVectorObj;
28      NaturalVectorObj = OBJECT(VectorObj)
29          PUBLIC
30              procedure add(val : integer); virtual;
31              procedure insertElementAt(pos : integer; val : integer); virtual;
32          PRIVATE
33              function isNatural(val : integer) : boolean;
34      end;
35      PrimeVector = ^PrimeVectorObj;
36      PrimeVectorObj = OBJECT(VectorObj)
37          PUBLIC
38              procedure add(val : integer); virtual;
39              procedure insertElementAt(pos : integer; val : integer); virtual;
40          PRIVATE
41              function isPrime(val : integer) : boolean;
42      end;
43
44
45  constructor VectorObj.init(userCapacity : integer);
46  begin
47      if(arrPtr <> NIL) then begin
48          writeln('Can''t initialize non-empty stack!');
49          halt;
50      end;
51      if(userCapacity <= 0) then begin
52          writeLn('No capacity given. Creating with default size ', DEFAULT_CAP_SIZE);
53          initCapacity := DEFAULT_CAP_SIZE;
54      end else
55          initCapacity := userCapacity;
56      new(arrPtr);
57      top := 0;
58      capacityCount := initCapacity;
59      GetMem(arrPtr, SIZEOF(integer) * capacityCount);
60  end;
61
62  destructor VectorObj.done;
63  begin
64      freeMem(arrPtr, SIZEOF(integer) * capacityCount);
65      arrPtr := NIL;
66  end;
67
68  procedure VectorObj.add(val : integer);
69  begin
70      if top >= capacityCount then begin
71          realloc;
72      end;
73      inc(top);
74      (*$R-*)
```

```
75      arrPtr^[top] := val;
76      (*$R+*)
77  end;
78
79  procedure VectorObj.insertElementAt(pos : integer; val : integer);
80  var i : integer;
81  begin
82      inc(top);
83      if(isOutOfRange(pos)) then
84          pos := top
85      else if pos < 0 then
86          pos := 0;
87      i := top;
88      while (i > pos) do begin
89          (*$R-*)
90          arrPtr^[i] := arrPtr^[i-1];
91          (*$R+*)
92          dec(i);
93      end;
94      (*$R-*)
95      arrPtr^[pos] := val;
96      (*$R+*)
97  end;
98
99  procedure VectorObj.getElementAt(pos : integer; var val : integer; var ok : boolean)
        ;
100 begin
101     ok := TRUE;
102     if(isOutOfRange(pos)) then begin
103         ok := FALSE;
104         val := -1;
105         exit;
106     end;
107     (*$R-*)
108     val := arrPtr^[pos];
109     (*$R+*)
110 end;
111
112 function VectorObj.size : integer;
113 begin
114     size := top;
115 end;
116
117 function VectorObj.capacity : integer;
118 begin
119     capacity := capacityCount;
120 end;
121
122 procedure VectorObj.clear;
123 begin
124     if arrPtr = NIL then begin
125         writeLn('Cannot dispose uninitialized vector!');
126         halt;
127     end;
128     freeMem(arrPtr, SIZEOF(integer) * capacityCount);
129     arrPtr := NIL;
130     init(initCapacity);
```

```
131 end;
132
133 procedure VectorObj.realloc;
134 var newArray : ^intArray;
135     i : integer;
136 begin
137     getMem(newArray, SIZEOF(INTEGER) * 2 * capacityCount);
138     for i := 1 to top do begin
139         (*$R-*)
140         newArray^[i] := arrPtr^[i];
141         (*$R+*)
142     end;
143     freeMem(arrPtr, SIZEOF(integer) * capacityCount);
144     capacityCount := 2 * capacityCount;
145     arrPtr := newArray;
146 end;
147
148 function VectorObj.isOutOfRange(pos : integer) : boolean;
149 begin
150     if pos > top then
151         isOutOfRange := TRUE
152     else
153         isOutOfRange := FALSE
154 end;
155
156 procedure NaturalVectorObj.add(val : integer);
157 begin
158     if NOT isNatural(val) then begin
159         writeln('Given value is not a natural number!');
160         exit;
161     end;
162     inherited add(val);
163 end;
164
165 procedure NaturalVectorObj.insertElementAt(pos : integer; val : integer);
166 begin
167     if NOT isNatural(val) then begin
168         writeln('Given value is not a natural number!');
169         exit;
170     end;
171     inherited insertElementAt(pos, val);
172 end;
173
174 function NaturalVectorObj.isNatural(val : integer) : boolean;
175 begin
176     if(val >= 0) then
177         isNatural := TRUE
178     else
179         isNatural := FALSE;
180 end;
181
182 procedure PrimeVectorObj.add(val : integer);
183 begin
184     if NOT isPrime(val) then begin
185         writeln('Given value is not a prime number!');
186         exit;
187     end;
```

```
188     inherited add(val);
189 end;
190 
191 procedure PrimeVectorObj.insertElementAt(pos : integer; val : integer);
192 begin
193     if NOT isPrime(val) then begin
194         writeln('Given value is not a prime number!');
195         exit;
196     end;
197     inherited insertElementAt(pos, val);
198 end;
199 
200 function PrimeVectorObj.isPrime(val : integer) : boolean;
201 var i : integer;
202 begin
203     isPrime := TRUE;
204     if(val <= 1) then begin
205         isPrime := FALSE;
206         exit;
207     end;
208     for i := 2 to val div 2-1 do
209         if val mod i = 0 then
210             isPrime := FALSE;
211 end;
212 
213 var
214     intVector : vector;
215     natVector : NaturalVector;
216     priVector : PrimeVector;
217     i : integer;
218     tVal : integer;
219     ok : boolean;
220 begin
221     New(intVector, init(4));
222     New(natVector, init(20));
223     New(priVector, init(17));
224 
225     for i := -20 to 20 do begin
226         intVector^.add(i);
227         natVector^.add(i);
228         priVector^.add(i);
229     end;
230     writeLn('Current size intVec: ', intVector^.size);
231     writeLn('Current size natVector: ', natVector^.size);
232     writeLn('Current size priVector: ', priVector^.size);
233 
234     write('intVec=[');
235     for i := 1 to intVector^.capacity do begin
236         intVector^.getElementAt(i, tVal, ok);
237         if(ok) then
238             write(tVal,',')
239     end;
240     write(']');
241     writeln;
242 
243     write('natVec=[');
244     for i := 1 to natVector^.capacity do begin
```

```
245         natVector^.getElementAt(i, tVal, ok);
246         if(ok) then
247             write(tVal,',')
248     end;
249     write(']');
250     writeln;
251 
252     write('priVec=[');
253     for i := 1 to priVector^.capacity do begin
254         priVector^.getElementAt(i, tVal, ok);
255         if(ok) then
256             write(tVal,',')
257     end;
258     write(']');
259     writeln;
260 end.
```

9.2.3 Ausgabe

```
Current size intVec: 41
Current size natVector: 21
Current size priVector: 9
intVec=[-20,-19,-18,-17,-16,-15,-14,-13,-12,-11,-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,]
natVec=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,]
priVec=[2,3,4,5,7,11,13,17,19,]
```

Abbildung 9.1: Inhalt der Vector nach einem Einfügvorgang von -20 bis +20

9.3 Verkettete Listen

9.3.1 Lösungsidee

Es ist eine verkette Liste im objektorientierten Design zu erstellen. Eine verkette Liste besteht aus einem Datentyp, welcher jeweils einen Wert enthält und einen Pointer auf das nächste Element der Liste. Die Verwaltung dieser Liste wird über das Objekt gemacht, das heißt das Objekt weiss den Startpointer der Liste und bietet einige Verwaltungsfunktionen wie das Hinzufügen, das Suchen und das Entfernen bestimmter Zahlen. Ausserdem wurde die Ausgabe der Liste zu den Funktionen hinzugefügt.

b)

Die Methode *add* wird erweitert, indem durch die existente Liste durchiteriert wird und überprüft wird, ob der Wert des nächsten Nodes größer ist als der mitgegebene Wert. Ist dies der Fall, ist der mitgegebene Wert an den momentanen Node anzuhägen. Dabei wird ein neuer Node erstellt, diesem der mitgegebene Wert gegeben, der aktuellen Node der Liste der *next* Zeiger auf die Neu-Erstellte Node gesetzt und der *next* Zeiger dieser Node auf den ursprünglichen nächsten Node der Liste. Somit wird das Element in die Liste eingepflanzt. In *contains* kann sich darauf verlassen werden, dass eine sortierte Liste vorliegt. Somit muss nur so lange durch die Liste durchiteriert werden, bis der momentane Wert größer ist als der gesuchte Wert. Ist dieser Wert bis jetzt nicht aufgetreten, wird er durch die Sortierung nach diesem Punkt auch nicht auftreten. Daher kann die Suche aufgegeben werden und ein *contains=FALSE* zurückgegeben werden. Dadurch verringert sich die Laufzeitkomplexität.

9.3.2 Implementierung

Listing 9.2: list.pas

```
1  program listOOP;
2
3  type
4      listNode = ^listPtr;
5      listPtr = record
6                  val  : integer;
7                  next : listNode;
8            end;
9      list = ^listObj;
10     listObj = OBJECT
11         private
12         startList : listNode;
13         sizeList : integer;
14         public
15             constructor init;
16             procedure add(val : integer); virtual;
17             function contains(val : integer) : boolean; virtual;
18             function size : integer;
19             procedure remove(val : integer); virtual;
20             procedure clear;
21             procedure printList;
```

```
22          private
23              procedure removeNextNode(var node : listNode);
24      end;
25      sortedList = ^sortedListObj;
26      sortedListObj = object(listObj)
27          public
28              procedure add(val : integer); virtual;
29              function contains(val : integer) : boolean; virtual;
30              procedure remove(val : integer); virtual;
31  end;
32
33  constructor listObj.init;
34  begin
35      startList := NIL;
36      sizeList := 0;
37  end;
38
39  procedure listObj.add(val : integer);
40  var nextList : listNode;
41      buffList : listNode;
42  begin
43      new(nextList);
44      nextList^.val := val;
45      if(size = 0) then
46          startList := nextList
47      else begin
48          buffList := startList;
49          while(buffList <> NIL) and (buffList^.next <> NIL) do
50              buffList := buffList^.next;
51          buffList^.next := nextList;
52      end;
53      inc(sizeList);
54  end;
55
56  function listObj.contains(val : integer) : boolean;
57  var curList : listNode;
58      finished : boolean;
59  begin
60      contains := FALSE;
61      finished := FALSE;
62      curList := startList;
63      while(NOT finished) do begin
64          if(curList^.val = val) then begin
65              contains := TRUE;
66              finished := TRUE;
67          end else begin
68              if(curList^.next = NIL) then begin
69                  finished := TRUE;
70              end else begin
71                  curList := curList^.next;
72              end;
73          end;
74      end;
75  end;
76
77  function listObj.size : integer;
78  var tSize : integer;
```

```
79      buffList : listNode;
80  begin
81      buffList := startList;
82      tSize := 0;
83      while(buffList <> NIL) do begin
84          inc(tSize);
85          buffList := buffList^.next;
86      end;
87      size := tSize;
88  end;
89
90  procedure listObj.remove(val : integer);
91  var buffList : listNode;
92  begin
93      if(size = 0) then begin
94          writeLn('Cannot remove from empty list');
95          exit;
96      end;
97      buffList := startList;
98      while(buffList^.next <> NIL) do begin
99          if(buffList^.next^.val = val) then begin
100             removeNextNode(buffList);
101         end else
102             buffList := buffList^.next;
103     end;
104     if(startList^.val = val) then begin
105         buffList := startList^.next;
106         dispose(startList);
107         dec(sizeList);
108         startList := buffList;
109     end;
110 end;
111
112 procedure listObj.clear;
113 var buffList : listNode;
114 begin
115     buffList := startList;
116     while(buffList^.next <> NIL) do begin
117         removeNextNode(buffList);
118     end;
119     dispose(startList);
120     sizeList := 0;
121     startList := NIL;
122 end;
123
124 procedure listObj.removeNextNode(var node : listNode);
125 var buffList : listNode;
126 begin
127     buffList := node^.next^.next;
128     dispose(node^.next);
129     node^.next := buffList;
130     dec(sizeList);
131 end;
132
133 procedure listObj.printList;
134 var buffList : listNode;
135 begin
```

```
136     buffList := startList;
137     while(buffList <> NIL) do begin
138         writeLn(buffList^.val);
139         buffList := buffList^.next;
140     end;
141 end;
142 
143 procedure sortedListObj.add(val : integer);
144 var nextList : listNode;
145     buffList : listNode;
146     finished : boolean;
147 begin
148     finished := FALSE;
149     new(nextList);
150     nextList^.val := val;
151     if(size = 0) then
152         startList := nextList
153     else if startList^.val >= nextList^.val then begin
154         nextList^.next := startList;
155         startList := nextList;
156     end else begin
157         buffList := startList;
158         while(buffList <> NIL) and (buffList^.next <> NIL) and (NOT finished) do
        begin
159             if(buffList^.next^.val >= val) then begin
160                 nextList^.next := buffList^.next;
161                 finished := TRUE;
162             end else
163                 buffList := buffList^.next;
164         end;
165             buffList^.next := nextList;
166     end;
167     inc(sizeList);
168 end;
169 
170 function sortedListObj.contains(val : integer) : boolean;
171 var curList : listNode;
172     finished : boolean;
173 begin
174     contains := FALSE;
175     finished := FALSE;
176     curList := startList;
177     while(NOT finished) do begin
178         if(curList^.val = val) then begin
179             contains := TRUE;
180             finished := TRUE;
181         end else begin
182             if(curList^.next = NIL) or (curList^.val < val) then begin
183                 finished := TRUE;
184             end else begin
185                 curList := curList^.next;
186             end;
187         end;
188     end;
189 end;
190 
191 procedure sortedListObj.remove(val : integer);
```

```
192 var buffList : listNode;
193     finished : boolean;
194 begin
195     finished := FALSE;
196     if(size = 0) then begin
197         writeLn('Cannot remove from empty list');
198         exit;
199     end;
200     buffList := startList;
201     while(buffList^.next <> NIL) and (NOT finished) do begin
202         if(buffList^.next^.val = val) then begin
203             removeNextNode(buffList);
204         end else
205             buffList := buffList^.next;
206         if(buffList^.val > val) then
207             finished := TRUE;
208     end;
209     if(startList^.val = val) then begin
210         buffList := startList^.next;
211         dispose(startList);
212         dec(sizeList);
213         startList := buffList;
214     end;
215 end;
216
217 var tList : sortedList;
218     i : integer;
219 begin
220     new(tList, init);
221     tList^.add(3);
222     tList^.add(3);
223     tList^.add(3);
224     tList^.add(2);
225     tList^.add(2);
226     tList^.add(2);
227     tList^.add(3);
228     writeLn('Size: ', tList^.size);
229     tList^.printList;
230     tList^.remove(3);
231     writeLn('Size: ', tList^.size);
232     tList^.clear;
233 end.
```

9.3.3 Ausgabe

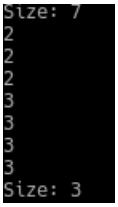

Abbildung 9.2: Inhalt der sortierten Liste