

תכונות סגירות של שפות

שפות חסרות הקשר	שפות רגולריות	
X	X	חלקיות \ הכלה
X	V	משלים
X	V	חיתוך
V	V	איחוד
V	V	היפוך
V	V	שרשור

פעולות על string

Properties	Description
Clone()	Make clone of string.
CompareTo()	Compares two specified String objects and returns an integer that indicates their relative position in the sort order.
Contains()	Returns a value indicating whether a specified substring occurs within this string.
EndsWith()	Determines whether the end of this string instance matches the specified string.
Equals()	Determines whether this instance and another specified String object have the same value
IndexOf(String)	Reports the zero-based index of the first occurrence of the specified string in this instance.
ToLower()	Returns a copy of this string converted to lowercase.
ToUpper()	Returns a copy of this string converted to uppercase.
Insert()	Returns a new string in which a specified string is inserted at a specified index position in this instance.
LastIndexOf(String)	Reports the zero-based index position of the last occurrence of a specified string within this instance.
Remove()	This method deletes all the characters from beginning to specified index position.
Replace()	This method helps to replace the character.
Split()	This method splits the string based on specified value.
StartsWith(String)	Determines whether the beginning of this string instance matches the specified string.
Substring()	Retrieves a substring from this instance. The substring starts at a specified character position and continues to the end of the string.
Trim()	It removes extra whitespaces from beginning and ending of string.

Random

```
Random rnd = new Random();
int num = rnd.Next(); // int as param -> 0 < random < x
```

Node Utils Section

```
1 using System;
2 using System.CodeDom;
3 using System.CodeDom.Compiler;
4 using System.Collections.Generic;
5 using System.Runtime.ExceptionServices;
6 using System.Runtime.InteropServices.WindowsRuntime;
7 using System.Security.Policy;
8
9 namespace Unit4
10 {
11     public static class NodeUtils
12     {
13         public static Node<T> CreateListFromArray<T>(T[] arr)
14         {
15             Node<T> head = null;
16
17             for (int i = arr.Length-1; i >= 0; i--)
18             {
19                 head = new Node<T>(arr[i], head);
20             }
21
22             return head;
23         }
24
25         public static Node<T> CreateListFromArrayR<T>(T[] arr, int i)
26         {
27             Node<T> head = new Node<T>(arr[i]);
28
29             if (i <= 0)
30                 return head;
31
32             head = new Node<T>(arr[i - 1], head);
33
34             return CreateListFromArrayR(arr, i - 1);
35         }
36
37         public static void PrintList<T>(Node<T> l)
38         {
39             while(l != null)
40             {
41                 Console.Write(l.GetValue() + "-->");
42                 l = l.GetNext();
43             }
44             Console.WriteLine();
45         }
46
47         public static void PrintListR<T>(Node<T> l)
48         {
49             if (l == null)
```

```
50         return;
51
52         Console.Write(l + "-->");
53
54         PrintList(l.GetNext());
55     }
56
57     public static Node<T> CloneList<T>(Node<T> l)
58     {
59         if (l == null)
60             return null;
61
62         Node<T> new_head = new Node<T>(l.GetValue());
63         Node<T> pos = new_head;
64
65         while(l.GetNext() != null)
66         {
67             l = l.GetNext();
68             pos.SetNext(new Node<T>(l.GetValue()));
69             pos = pos.GetNext();
70         }
71
72         return new_head;
73     }
74
75     public static bool CompareList(Node<int> left, Node<int> right)
76     {
77         bool equal = true;
78
79         while(left != null && right != null && equal)
80         {
81             if ((left.GetNext() == null && right.GetNext() != null) ||
82                 (left.GetNext() != null && right.GetNext() == null))
83                 equal = false;
84
85             equal = left.GetValue() == right.GetValue() && equal;
86
87             left = left.GetNext();
88             right = right.GetNext();
89         }
90
91         return equal;
92     }
93
94     public static bool CompareListR(Node<int> left, Node<int> right)
95     {
96         if (left == null && right == null)
97             return true;
```

```
198         if ((left.GetNext() == null && right.GetNext() != null) ||  
199             (left.GetNext() != null && right.GetNext() == null))  
200             return false;  
201         return left.GetValue() == right.GetValue() && CompareListR  
202             (left.GetNext(), right.GetNext());  
203     }  
204     public static int CountList<T>(Node<T> lst)  
205     {  
206         int cnt = 0;  
207         while(lst != null)  
208         {  
209             cnt++;  
210             lst = lst.GetNext();  
211         }  
212         return cnt;  
213     }  
214     public static int CountListR<T>(Node<T> lst)  
215     {  
216         if (lst == null)  
217             return 0;  
218         return 1 + CountListR(lst.GetNext());  
219     }  
220     public static int SumList(Node<int> lst)  
221     {  
222         int sum = 0;  
223         while (lst != null)  
224         {  
225             sum += lst.GetValue();  
226             lst = lst.GetNext();  
227         }  
228         return sum;  
229     }  
230     public static int SumListR(Node<int> lst)  
231     {  
232         if (lst == null)  
233             return 0;  
234         return lst.GetValue() + SumListR(lst.GetNext());  
235     }
```

```
145
146     public static bool IsExist(Node<int> lst, int val)
147     {
148         bool found = false;
149
150         while (!found && lst != null)
151         {
152             found = lst.GetValue() == val;
153             lst = lst.GetNext();
154         }
155
156         return found;
157     }
158
159     public static bool IsExistR(Node<int> lst, int val)
160     {
161         if (lst == null)
162             return false;
163
164         return lst.GetValue() == val || IsExistR(lst.GetNext(), val);
165     }
166
167     public static int FindMax(Node<int> lst)
168     {
169         int max = int.MinValue;
170
171         while (lst != null)
172         {
173             if (lst.GetValue() > max)
174             {
175                 max = lst.GetValue();
176             }
177             lst = lst.GetNext();
178         }
179
180         return max;
181     }
182
183     public static int FindMaxR(Node<int> lst)
184     {
185         if (lst == null)
186             return int.MinValue;
187
188         return Math.Max(lst.GetValue(), FindMaxR(lst.GetNext()));
189     }
190
191     public static void AbsValue(Node<int> lst)
192     {
193         while (lst != null)
```

```
194         {
195             lst.SetValue(Math.Abs(lst.GetValue()));
196             lst = lst.GetNext();
197         }
198     }
199
200     public static void AbsValueR(Node<int> lst)
201     {
202         if (lst == null)
203             return;
204
205         lst.SetValue(Math.Abs(lst.GetValue()));
206
207         AbsValue(lst.GetNext());
208     }
209
210     public static Node<T> GetNodeRef<T>(Node<T> lst, int loc)
211     {
212         Node<T> r = null; // reference
213
214         int save_i = 1;
215
216         for (int i = 1; i <= loc && lst != null; i++)
217         {
218             r = lst;
219             save_i = i;
220             lst = lst.GetNext();
221         }
222
223         if (save_i < loc)
224             return null;
225         return r;
226     }
227
228     public static Node<T> GetNodeRefR<T>(Node<T> lst, int loc)
229     {
230         if (loc == 0)
231             return lst;
232
233         if (lst == null)
234             return null;
235
236         return GetNodeRef<T>(lst.GetNext(), loc - 1);
237     }
238
239     public static bool IsSorted(Node<int> lst)
240     {
241         bool is_sorted = true;
242     }
```

```
243         while (lst.GetNext() != null && is_sorted)
244         {
245             is_sorted = lst.GetValue() <= lst.GetNext().GetValue();
246             lst = lst.GetNext();
247         }
248
249         return is_sorted;
250
251     }
252
253     public static bool IsSortedR(Node<int> lst)
254     {
255         if (lst.GetNext() == null)
256             return true;
257
258         return lst.GetValue() <= lst.GetNext().GetValue() && IsSortedR ↗
            (lst.GetNext());
259     }
260
261     public static int CountSeqs(Node<int> lst, int n)
262     {
263         int seq_cnt = 0;
264         int curr_seq_len = 1;
265
266         while(lst != null)
267         {
268             if (lst.GetValue() == n)
269             {
270                 if (curr_seq_len == 1)
271                     seq_cnt++;
272
273                 curr_seq_len++;
274             }
275
276             else
277             {
278                 curr_seq_len = 1;
279             }
280
281             lst = lst.GetNext();
282         }
283
284         return seq_cnt;
285     }
286
287     public static void PrintAtoB(Node<int> lst, int a, int b)
288     {
289         for (int i = 0; i < a-1; i++)
290         {
```



```
291         lst = lst.GetNext();
292     }
293
294     for (int i = 0; i < b-a+1; i++)
295     {
296         Console.Write(lst + "-->");
297         lst = lst.GetNext();
298     }
299 }
300
301 public static Node<int> RemoveDuplicates(Node<int> lst)
302 {
303     Node<int> new_lst = new Node<int>(lst.GetValue());
304     lst = lst.GetNext();
305     Node<int> head = new_lst;
306
307     while(lst != null)
308     {
309         if (!IsExist(head, lst.GetValue()))
310         {
311             new_lst.SetNext(new Node<int>(lst.GetValue()));
312             new_lst = new_lst.GetNext();
313         }
314
315         lst = lst.GetNext();
316     }
317
318     return head;
319 }
320
321 public static bool BalancedList(Node<int> lst)
322 {
323     int sum = 0;
324     int cnt = 0;
325
326     Node<int> save_lst = lst;
327
328     while(lst != null)
329     {
330         sum += lst.GetValue();
331         cnt++;
332         lst = lst.GetNext();
333     }
334
335     double avg = (double)(sum) / cnt;
336
337     lst = save_lst;
338
339     int aboveAvg = 0;
```

```
340         int belowAvg = 0;
341
342         while(lst != null)
343         {
344             if (lst.GetValue() > avg)
345                 aboveAvg++;
346
347             else if (lst.GetValue() < avg)
348                 belowAvg++;
349
350             lst = lst.GetNext();
351         }
352
353         return aboveAvg == belowAvg;
354     }
355
356     public static (Node<int>, Node<int>) RemoveMax(Node<int> lst)
357     {
358         Node<int> save_lst = lst;
359
360         Node<int> maxNode = lst;
361         Node<int> previousMaxNode = null;
362         int max = lst.GetValue();
363         previousMaxNode = null;
364
365
366         while(lst.GetNext() != null)
367         {
368
369
370             if (lst.GetNext().GetValue() > max)
371             {
372                 maxNode = lst.GetNext();
373                 max = lst.GetNext().GetValue();
374                 previousMaxNode = lst;
375             }
376
377             lst = lst.GetNext();
378         }
379
380         if (previousMaxNode == null)
381             lst = maxNode.GetNext();
382
383         else
384         {
385             previousMaxNode.SetNext(maxNode.GetNext());
386             maxNode.SetNext(null);
387             lst = save_lst;
388         }
```

```
389         return (maxNode, lst);
390     }
391
392     public static Node<int> InsertToSortedList(Node<int> lst,
393         Node<int> node)
394     {
395         Node<int> save_lst = lst;
396         Node<int> previous = null;
397
398         while (lst != null && lst.GetValue() < node.GetValue())
399         {
400             previous = lst;
401             lst = lst.GetNext();
402         }
403
404         if (previous == null)
405             lst = new Node<int>(node.GetValue(), lst);
406
407         else
408         {
409             node.SetNext(lst);
410             previous.SetNext(node);
411             lst = save_lst;
412         }
413
414         return lst;
415     }
416
417     public static Node<int> Sort(Node<int> lst)
418     {
419         Node<int> sorted = new Node<int>(lst.GetValue());
420
421         while(lst.GetNext() != null)
422         {
423             lst = lst.GetNext();
424             Node<int> new_node = new Node<int>(lst.GetValue());
425             sorted = InsertToSortedList(sorted, new_node);
426         }
427
428         return sorted;
429     }
430
431     public static Node<T> ReverseList<T>(Node<T> lst)
432     {
433         Node<T> curr = lst;
434         Node<T> next = lst.GetNext();
435         Node<T> next_next = lst.GetNext().GetNext();
436
```

```
437
438         while(next != null)
439         {
440             next.SetNext(curr);
441             curr = next;
442             next = next_next;
443             if (next_next != null)
444                 next_next = next_next.GetNext();
445         }
446
447         lst.SetNext(null);
448
449         return curr;
450     }
451
452     public static Node<T> ReverseListR<T>(Node<T> lst)
453     {
454         if (lst.GetNext() == null)
455             return lst;
456
457         Node<T> reversed = ReverseListR(lst.GetNext());
458         lst.GetNext().SetNext(reversed);
459         lst.SetNext(null);
460
461         return reversed;
462     }
463
464     public static int NegativeSequence(Node<int> lst)
465     {
466         int max_seq_len = 0;
467         int curr_seq_len = 1;
468
469         while(lst.GetNext() != null)
470         {
471             if (lst.GetValue() < 0 && lst.GetNext().GetValue() < 0)
472             {
473                 curr_seq_len++;
474             }
475
476             else if (curr_seq_len > 1)
477             {
478                 if (curr_seq_len > max_seq_len)
479                     max_seq_len = curr_seq_len;
480                 curr_seq_len = 1;
481             }
482
483             lst = lst.GetNext();
484
485         }
```

```
486
487     return max_seq_len;
488 }
489
490 public static Node<int> MergeSortedLists(Node<int> l1, Node<int> l2)
491 {
492
493     Node<int> merged = new Node<int>(0);
494     Node<int> head = null;
495     Node<int> last = null;
496
497     while (l1 != null && l2 != null)
498     {
499         if (l1.GetValue() < l2.GetValue())
500         {
501             merged.SetValue(l1.GetValue());
502             if (head == null)
503                 head = merged;
504             l1 = l1.GetNext();
505         }
506         else
507         {
508             merged.SetValue(l2.GetValue());
509             if (head == null)
510                 head = merged;
511             l2 = l2.GetNext();
512         }
513
514         merged.SetNext(new Node<int>(0));
515         last = merged;
516         merged = merged.GetNext();
517     }
518
519     if (l1 != null)
520         last.SetNext(l1);
521
522     else if (l2 != null)
523         last.SetNext(l2);
524
525     return head;
526 }
527
528
529
530 public static Node<int> MergeSortedLists2(Node<int> l1, Node<int> l2)
531 {
532     Node<int> next_l1, next_l2 = null;
```

```
533         Node<int> head = null;
534
535         while(l1 != null && l2 != null)
536         {
537             if (l1.GetValue() > l2.GetValue())
538             {
539                 if (head == null)
540                     head = l2;
541
542                 next_l2 = l2.GetNext();
543                 l2.SetNext(l1);
544                 l2 = next_l2;
545             }
546
547             else
548             {
549                 if (head == null)
550                     head = l1;
551
552                 next_l1 = l1.GetNext();
553                 l1.SetNext(l2);
554                 l1 = next_l1;
555             }
556         }
557
558
559         return head;
560     }
561
562     public static void AddFollowingNum(Node<int> lst, int val)
563     {
564         while(lst.GetValue() != val)
565         {
566             lst = lst.GetNext();
567         }
568
569         lst.SetNext(new Node<int>(val + 1, lst.GetNext()));
570     }
571
572     public static Node<int> ListsIntersection(Node<int> l1, Node<int> l2)
573     {
574         Node<int> head = null;
575         Node<int> save_head = null;
576
577
578         while(l1 != null)
579         {
580             if (IsExist(l2, l1.GetValue()))
```

```
581         {
582             if (head == null)
583                 head = new Node<int>(l1.GetValue());
584             else
585             {
586                 head.SetNext(new Node<int>(l1.GetValue()));
587                 head = head.GetNext();
588             }
589
590             if (save_head == null)
591                 save_head = head;
592         }
593
594         l1 = l1.GetNext();
595     }
596
597     return save_head;
598 }
599
600 public static bool Compare(Node<int> left, Node<int> right)
601 {
602     bool equals = true;
603
604     while(left.GetNext() != null && right.GetNext() != null)
605     {
606         equals = left.GetValue() == right.GetValue();
607
608         left = left.GetNext();
609         right = right.GetNext();
610     }
611
612     return equals;
613 }
614
615 public static Node<T> GetMiddle<T>(Node<T> lst)
616 {
617     Node<T> slow = lst;
618     Node<T> fast = lst;
619
620     while(fast.GetNext() != null)
621     {
622         slow = slow.GetNext();
623         fast = fast.GetNext().GetNext();
624     }
625
626     return slow;
627 }
628
629 public static bool IsPalindrome(Node<char> lst)
```

```
630     {
631         Node<char> rlst = ReverseList(GetMiddle(lst));
632
633         bool pali = true;
634         while (pali && lst != rlst && lst != null)
635         {
636             pali = lst.GetValue() == rlst.GetValue();
637             lst = lst.GetNext();
638             rlst = rlst.GetNext();
639         }
640
641         return pali;
642     }
643 }
644 }
645 }
646 }
```


Round Node Utils Section

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Unit4
8 {
9     class RoundNodeUtils
10    {
11        public static void MakeListRound<T>(Node<T> lst)
12        {
13            Node<T> head = lst;
14
15            while (lst.GetNext() != null)
16            {
17                lst = lst.GetNext();
18            }
19
20            lst.SetNext(head);
21        }
22
23        public static void PrintRoundList<T>(Node<T> lst)
24        {
25
26            Node<T> pos = lst.GetNext();
27
28            Console.Write(lst + "-->");
29
30            while (pos != lst)
31            {
32                Console.Write(pos + "-->");
33                pos = pos.GetNext();
34            }
35        }
36
37        public static void DisconnectRoundList<T>(Node<T> lst)
38        {
39            Node<T> pos = lst.GetNext();
40
41            while (pos.GetNext() != lst)
42            {
43                pos = pos.GetNext();
44            }
45
46            pos.SetNext(null);
47        }
48
49        public static bool IsRoundList<T>(Node<T> head)
```

```
50     {
51         Node<T> pos = head.GetNext();
52
53         while (pos != head && pos != null)
54         {
55             pos = pos.GetNext();
56         }
57
58         return pos == head;
59     }
60
61     public static int ListLength<T>(Node<T> head)
62     {
63         Node<T> pos = head.GetNext();
64         int cnt = 1;
65
66         while (pos != head)
67         {
68             pos = pos.GetNext();
69             cnt++;
70         }
71
72         return cnt;
73     }
74
75     public static int SumList(Node<int> head)
76     {
77         Node<int> pos = head.GetNext();
78
79         int sum = head.GetValue();
80
81         while(pos != head)
82         {
83             sum += pos.GetValue();
84             pos = pos.GetNext();
85         }
86
87         return sum;
88     }
89
90     public static Node<T> RemoveHead<T>(Node<T> head)
91     {
92         Node<T> pos = head;
93
94         while(pos.GetNext() != head)
95         {
96             pos = pos.GetNext();
97         }
98     }
```

```
99         pos.SetNext(head.GetNext());
100         return pos.GetNext();
101     }
102
103     public static void RemoveLast<T>(Node<T> head)
104     {
105         Node<T> pos = head;
106
107         while (pos.GetNext().GetNext() != head)
108         {
109             pos = pos.GetNext();
110         }
111
112         pos.SetNext(head);
113     }
114
115     public static bool IsExist(Node<int> head, int value)
116     {
117         Node<int> pos = head.GetNext();
118
119         bool found = false;
120
121         while (pos != head && !found)
122         {
123             found = pos.GetValue() == value;
124             pos = pos.GetNext();
125         }
126
127         return found;
128     }
129
130     public static Node<int> RemoveEven(Node<int> head)
131     {
132         Node<int> last = head;
133         Node<int> pos = head.GetNext();
134
135         while (pos != head)
136         {
137             if (pos.GetValue() % 2 == 0)
138             {
139                 last.SetNext(pos.GetNext());
140             }
141
142             else
143                 last = pos;
144
145             pos = pos.GetNext();
146         }
147     }
```

```
148         if (pos.GetValue() % 2 == 0)
149         {
150             last.SetNext(pos.GetNext());
151             return pos.GetNext();
152         }
153
154         return head;
155     }
156
157     public static void AddToEven(Node<int> head)
158     {
159         Node<int> last = head;
160         Node<int> pos = head.GetNext();
161
162         while(pos != head)
163         {
164             if (pos.GetValue() % 2 == 0)
165             {
166                 last.SetNext(new Node<int>(pos.GetValue() - 1));
167                 last.GetNext().SetNext(pos);
168                 last = pos;
169             }
170
171             else
172                 last = pos;
173
174             pos = pos.GetNext();
175         }
176
177         if (pos.GetValue() % 2 == 0)
178         {
179             last.SetNext(new Node<int>(pos.GetValue() - 1));
180             last.GetNext().SetNext(pos);
181         }
182     }
183
184     public static Node<T> AddToLoop<T>(Node<T> head, Node<T> new_node)
185     {
186         Node<T> pos = head.GetNext();
187
188         while (pos.GetNext() != head)
189             pos = pos.GetNext();
190
191         pos.SetNext(new_node);
192         new_node.SetNext(head);
193
194         return new_node; // new_node => new list head
195     }
196 }
```

```
197
198     public static void SumNeighbors(Node<int> head)
199     {
200         Node<int> last = head;
201         Node<int> pos = head.GetNext();
202
203         while(pos != head)
204         {
205             AddToLoop(pos, new Node<int>(last.GetValue() + pos.GetValue()));
206             last = pos;
207             pos = pos.GetNext();
208         }
209         AddToLoop(pos, new Node<int>(last.GetValue() + pos.GetValue()));
210     }
211
212     public static bool HasLoop<T>(Node<T> head)
213     {
214         Node<T> curr = head;
215         Node<T> next = head.GetNext();
216         Node<T> next_next = next.GetNext();
217
218         while (next != null && curr != next_next)
219         {
220             next.SetNext(curr);
221
222             curr = next;
223             next = next_next;
224             if (next_next != null)
225                 next_next = next_next.GetNext();
226         }
227
228         bool foundLoop = curr == next_next;
229
230         return foundLoop;
231     }
232
233     public static void CreateLoopList<T>(Node<T> lst, int n)
234     {
235         Node<T> pos = lst;
236         for (int i = 0; i < n; i++)
237         {
238             pos = pos.GetNext();
239         }
240
241         while (lst.GetNext() != null)
242             lst = lst.GetNext();
243     }
```

```
244         lst.SetNext(pos);
245     }
246
247     public static Node<T> IntersectionPoint<T>(Node<T> rLst)
248     {
249         Node<T> turtle = rLst;
250         Node<T> rabbit = rLst;
251
252         bool found_intersect = true;
253
254         while (rabbit.GetNext().GetNext() != null && found_intersect)
255         {
256             for (int i = 0; i < 2; i++)
257                 rabbit = rabbit.GetNext();
258
259             turtle = turtle.GetNext();
260
261             found_intersect = rabbit != turtle;
262         }
263
264         if (rabbit == turtle)
265             return rabbit;
266
267         return null;
268     }
269
270     public static Node<T> CrossSection<T>(Node<T> lst)
271     {
272         Node<T> intersect = IntersectionPoint(lst);
273
274         while (intersect != lst)
275         {
276             intersect = intersect.GetNext();
277             lst = lst.GetNext();
278         }
279
280         return intersect;
281     }
282
283     public static void PrintLoopRoundList<T>(Node<T> lst)
284     {
285         Node<T> intersect = IntersectionPoint(lst);
286         while (lst != intersect)
287         {
288             Console.Write(lst.GetValue() + "-->");
289             lst = lst.GetNext();
290         }
291     }
292
```

```
293         Console.Write("(");
294         Console.Write(lst + "-->");
295         lst = lst.GetNext();
296
297         while(lst != intersect)
298         {
299             Console.Write(lst + "-->");
300             lst = lst.GetNext();
301         }
302
303         Console.Write(")");
304     }
305 }
306 }
307
```


Fork Utils Section

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Unit4
8 {
9     class ForkNodeUtils
10    {
11        public static void CreateForkList<T>(Node<T> ls1, Node<T> ls2, int n)
12        {
13            while(ls2.GetNext() != null)
14                ls2 = ls2.GetNext();
15
16            for (int i = 0; i < n-1; i++)
17                ls1 = ls1.GetNext();
18
19            ls2.SetNext(ls1);
20        }
21
22        public static Node<T> GetForkMeeting<T>(Node<T> ls1, Node<T> ls2)
23        {
24            int size_ls1 = NodeUtils.CountList(ls1);
25            int size_ls2 = NodeUtils.CountList(ls2);
26
27            int diff = Math.Abs(size_ls2 - size_ls1);
28
29            if (size_ls1 > size_ls2)
30            {
31                for (int i = 0; i < diff; i++)
32                    ls1 = ls1.GetNext();
33            }
34
35            else
36            {
37                for (int i = 0; i < diff; i++)
38                    ls2 = ls2.GetNext();
39            }
40
41            while(ls1 != ls2 && ls1 != null && ls2 != null)
42            {
43                ls1 = ls1.GetNext();
44                ls2 = ls2.GetNext();
45            }
46
47            if (ls1 == null || ls2 == null)
48                return null;
```

```
49
50         return ls1;
51     }
52
53     public static Node<T> MergeFrokList<T>(Node<T> l1, Node<T> l2)
54     {
55         Node<T> merge = GetForkMeeting(l1, l2);
56
57         int size_l1 = NodeUtils.CountList(l1);
58         int size_l2 = NodeUtils.CountList(l2);
59
60         Node<T> head = size_l1 > size_l2 ? l1 : l2;
61         Node<T> pos = head;
62
63         while (pos.GetNext() != null)
64             pos = pos.GetNext();
65
66         Node<T> conn = size_l1 > size_l2 ? l2 : l1;
67
68         pos.SetNext(conn);
69
70         while (conn.GetNext() != merge)
71             conn = conn.GetNext();
72
73         conn.SetNext(null);
74
75
76         return head;
77     }
78
79 }
80 }
81
```

Queue Utils Section

```
1 using System;
2 using System.CodeDom.Compiler;
3 using System.Collections.Generic;
4 using System.Data.SqlClient;
5 using System.Runtime.Remoting.Messaging;
6 using System.Security;
7
8 namespace Unit4
9 {
10     class QueueUtils
11     {
12         public static Queue<T> CreateQueueFromArray<T>(T[] arr)
13         {
14             Queue<T> s = new Queue<T>();
15
16             for (int i = 0; i < arr.Length; i++)
17             {
18                 s.Insert(arr[i]);
19             }
20
21             return s;
22         }
23         public static void SpilledOn<T>(Queue<T> dest, Queue<T> src)
24         {
25             while (!src.IsEmpty())
26             {
27                 T temp = src.Remove();
28                 dest.Insert(temp);
29             }
30         }
31
32         public static Queue<T> Clone<T>(Queue<T> s)
33         {
34             Queue<T> t = new Queue<T>();
35             Queue<T> s2 = new Queue<T>();
36
37             SpilledOn(t, s);
38
39             while(!t.IsEmpty())
40             {
41                 T temp = t.Remove();
42                 s.Insert(temp);
43                 s2.Insert(temp);
44             }
45
46             return s2;
47         }
48         public static int GetSize<T>(Queue<T> q)
49         {
```

```
50         int count = 0;
51
52         Queue<T> temp = Clone(q);
53         while (!temp.IsEmpty())
54         {
55             temp.Remove();
56             count++;
57         }
58
59         return count;
60     }
61
62     public static int GetSum(Queue<int> q)
63     {
64         int sum = 0;
65
66         Queue<int> temp = Clone(q);
67         while (!temp.IsEmpty())
68         {
69
70             sum+=temp.Remove();
71         }
72
73         return sum;
74     }
75
76     public static bool IsExist<T>(Queue<T> q, T e)
77     {
78         Queue<T> temp = Clone(q);
79
80         while (!temp.IsEmpty())
81         {
82             if (EqualityComparer<T>.Default.Equals(temp.Remove(), e))
83                 return true;
84         }
85         return false;
86     }
87
88     public static void LastToFirst<T>(Queue<T> q)
89     {
90         Queue<T> temp = new Queue<T>();
91
92         int l = GetSize(q) - 1;
93
94         for (int i = 0; i < l; i++)
95         {
96             temp.Insert(q.Remove());
97         }
98     }
```

```
99         while(!temp.IsEmpty())
100         {
101             q.Insert(temp.Remove());
102         }
103     }
104
105     public static bool IsSorted(Queue<int> q)
106     {
107         Queue<int> temp = Clone(q);
108
109         bool sorted = true;
110         int last = q.Head();
111         while(!temp.IsEmpty() && sorted)
112         {
113             int curr = temp.Remove();
114             sorted = last <= curr;
115             last = curr;
116         }
117
118         return sorted;
119     }
120
121     public static void InsertToSorted(Queue<int> q, int val)
122     {
123         Queue<int> temp = new Queue<int>();
124
125         bool found = false;
126
127         while (!q.IsEmpty())
128         {
129             int curr = q.Remove();
130             if (val < curr && !found)
131             {
132                 temp.Insert(val);
133                 temp.Insert(curr);
134                 found = true;
135             }
136             else
137             {
138                 temp.Insert(curr);
139             }
140         }
141
142     }
143
144     if (!found)
145     {
146         temp.Insert(val);
147     }
```

```
148
149         SpilledOn(q, temp);
150     }
151
152     public static int FindMin(Queue<int> q)
153     {
154         Queue<int> temp = Clone(q);
155
156         int min = int.MaxValue;
157
158         while(!temp.IsEmpty())
159         {
160             int val = temp.Remove();
161             if (val < min)
162                 min = val;
163         }
164
165         return min;
166     }
167
168     public static int FindMax(Queue<int> q)
169     {
170         Queue<int> temp = Clone(q);
171
172         int max = int.MinValue;
173
174         while (!temp.IsEmpty())
175         {
176             int val = temp.Remove();
177             if (val > max)
178                 max = val;
179         }
180
181         return max;
182     }
183
184     public static void RemoveMin(Queue<int> q)
185     {
186         Queue<int> temp = Clone(q);
187
188         int min_index = 0;
189         int min = int.MaxValue;
190         int cnt = 0;
191
192         while (!temp.IsEmpty())
193         {
194             int val = temp.Remove();
195             if (val < min)
196             {
```



```
197         min = val;
198         min_index = cnt;
199     }
200     cnt++;
201 }
202
203 SpilledOn(temp, q);
204
205 int l = GetSize(temp);
206
207 for (int i = 0; i < l; i++)
208 {
209     int val = temp.Remove();
210     if (i != min_index)
211     {
212         q.Insert(val);
213     }
214 }
215
216 }
217
218 public static void RemoveMax(Queue<int> q)
219 {
220     Queue<int> temp = Clone(q);
221
222     int max_index = 0;
223     int max = int.MinValue;
224     int cnt = 0;
225
226     while (!temp.IsEmpty())
227     {
228         int val = temp.Remove();
229         if (val > max)
230         {
231             max = val;
232             max_index = cnt;
233         }
234         cnt++;
235     }
236
237     SpilledOn(temp, q);
238
239     int l = GetSize(temp);
240
241     for (int i = 0; i < l; i++)
242     {
243         int val = temp.Remove();
244         if (i != max_index)
245         {
```

```
246         q.Insert(val);
247     }
248 }
249
250
251 public static void SortQueue(Queue<int> q)
252 {
253     int l = GetSize(q);
254     Queue<int> temp = new Queue<int>();
255
256     for (int i = 0; i < l; i++)
257     {
258         int val = FindMin(q);
259         temp.Insert(val);
260         RemoveMin(q);
261     }
262
263     SpilledOn(q, temp);
264 }
265
266 public static void Reverse<T>(Queue<T> q)
267 {
268     int l = GetSize(q);
269
270     Queue<T> new_q = new Queue<T>();
271     Queue<T> save_q = new Queue<T>();
272     SpilledOn(save_q, q);
273
274     for (int i = 0; i < l; i++)
275     {
276         Queue<T> temp = Clone(save_q);
277         for (int j = 0; j < l - i - 1; j++)
278             temp.Remove();
279         q.Insert(temp.Remove());
280     }
281 }
282
283 public static void RemoveDuplicates(Queue<int> q)
284 {
285     Queue<int> temp = new Queue<int>();
286     SpilledOn(temp, q);
287
288     while (!temp.IsEmpty())
289     {
290         int val = temp.Remove();
291         if (!IsExist(q, val))
292             q.Insert(val);
293     }
294 }
```

```
295
296     public static int Count(Queue<int> q, int n)
297     {
298         Queue<int> temp = Clone(q);
299
300         int cnt = 0;
301
302         while (!temp.IsEmpty())
303         {
304             if (temp.Remove() == n)
305                 cnt++;
306         }
307
308         return cnt;
309     }
310
311     public static void RemoveSpec(Queue<int> q, int val)
312     {
313         Queue<int> temp = new Queue<int>();
314         SpilledOn(temp, q);
315
316         while (!temp.IsEmpty())
317         {
318             int curr = temp.Remove();
319             if (curr != val)
320                 q.Insert(curr);
321         }
322     }
323
324     public static void InsertAtPos<T>(Queue<T> q, T e, int n)
325     {
326     }
327
328     // --- Bagrut Exercices ---
329     public static int ToNumber(Queue<int> q)
330     {
331         int num = 0;
332
333         while (!q.IsEmpty())
334         {
335             int val = q.Remove();
336
337             num *= 10;
338             num += val;
339         }
340
341         return num;
342     }
343
```

```
344     public static int BigNumber(Queue<Queue<int>> q)
345     {
346         int max = int.MinValue;
347
348         Queue<Queue<int>> clone = Clone(q);
349
350         while (!clone.IsEmpty())
351         {
352             int val = ToNumber(clone.Remove());
353             if (val > max)
354                 max = val;
355         }
356
357         return max;
358     }
359
360     // ---
361     public static void ConnectQueues<T>(Queue<T> q1, Queue<T> q2)
362     {
363         Queue<T> temp = Clone(q2);
364
365         while (!temp.IsEmpty())
366             q1.Insert(temp.Remove());
367     }
368     public static Queue<int> DoublesToPali(Queue<int> qd)
369     {
370         if (GetSize(qd) == 2)
371             return qd;
372
373         Queue<int> res = new Queue<int>();
374         res.Insert(qd.Remove());
375         int val = qd.Remove();
376         ConnectQueues(res, DoublesToPali(qd));
377         res.Insert(val);
378
379         return res;
380     }
381
382     // ---
383     public static bool IsIdentical(Queue<int> q1, Queue<int> q2)
384     {
385         Queue<int> copy1 = Clone(q1);
386         Queue<int> copy2 = Clone(q2);
387
388         bool identical = true;
389
390         while(identical && !copy1.IsEmpty() && !copy2.IsEmpty())
391         {
392
```

```
393         identical = copy1.Remove() == copy2.Remove();
394
395         if ((!copy1.IsEmpty() && copy2.IsEmpty()) ||
396             (copy1.IsEmpty() && !copy2.IsEmpty()))
397             identical = false;
398     }
399     return identical;
400 }
401
402 public static bool IsSimilar(Queue<int> q1, Queue<int> q2)
403 {
404     int size = GetSize(q1);
405
406     bool similar = false;
407
408     for (int i = 0; i < size && !similar; i++)
409     {
410         similar = IsIdentical(q1, q2);
411         LastToFirst(q1);
412     }
413
414     return similar;
415 }
416
417
418 // -- Bagrut 2023
419 public static bool TwoSum(Queue<int> q, int x)
420 {
421     bool found = false;
422     Queue<int> copy = Clone(q);
423
424     while (!found && !copy.IsEmpty())
425     {
426         Queue<int> temp = Clone(copy);
427         int head = temp.Remove();
428
429         while(!temp.IsEmpty() && !found)
430         {
431             found = head + temp.Remove() == x;
432         }
433
434         copy.Remove();
435     }
436
437     return found;
438
439 }
440 }
```

```
441
442     }
443 }
444
445
446
447
448
449
450
451
452
453
454
455
```