

```
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             if (lst.GetNext().GetValue() > max)
370             {
371                 maxNode = lst.GetNext();
372                 max = lst.GetNext().GetValue();
373                 previousMaxNode = lst;
374             }
375
376             lst = lst.GetNext();
377         }
378
379         if (previousMaxNode == null)
380             lst = maxNode.GetNext();
381
382         else
383         {
384             previousMaxNode.SetNext(maxNode.GetNext());
385             maxNode.SetNext(null);
386             lst = save_lst;
387         }
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
507             else
508             {
509                 merged.SetValue(l2.GetValue());
510                 if (head == null)
511                     head = merged;
512                 l2 = l2.GetNext();
513             }
514
515             merged.SetNext(new Node<int>(0));
516             last = merged;
517             merged = merged.GetNext();
518
519         }
520
521         if (l1 != null)
522             last.SetNext(l1);
523
524         else if (l2 != null)
525             last.SetNext(l2);
526
527         return head;
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 }
```