Aufgabe 1

1. FIFO
2. LIFO
3. FIFO
4. FIFO
5. FIFO
6. FIFO
7. LIFO

Aufgabe 2

(a) Setze Hase und Igel auf Element 0 der Liste. Sitzen sie irgendwann nach Verlassen des 0-ten Elements wieder auf dem gleichen Feld, so enthält die Liste einen Zyklus. Sofern die Liste einen Zyklus enthält, treffen sich die beiden nach dieser Begegnung noch einmal. Behauptung: Sie treffen sich nach n Zügen wieder.

Beweis. Der Vorsprung des Hasen wir pro Zug um 1 erhöht. Nach n Zügen wurde der Vorsprung des Hasen also um n erhöht. Da der Zyklus die Länge n hat, sitzen Hase und Igel nun also wieder auf dem gleichen Feld. □

(b)

Listing 1: haseigel.cc

```
1  #include <iostream>
2
3
4  struct IntListElem
5  {
6      IntListElem* next;
7      int value;
8  };
9
10 struct IntList
11 {
12     int count;
13     IntListElem* first;
14 };
15
16 void empty_list(IntList* l)
17 {
18     l->first = 0;
19     l->count = 0;
```

```
20 }
21
22 IntListElem* find_first_x(IntList l, int x)
23 {
24     for (IntListElem* p = l.first; p != 0; p = p->next)
25     {
26         if (p->value == x)
27         {
28             return p;
29         }
30     }
31     return 0;
32 }
33
34 void insert_in_list_cycle(IntList* list, IntListElem* where, IntListElem* ins)
35 {
36     if (where == 0)
37     {
38         ins->next = ins;
39         list->first = ins;
40         ++list->count;
41     }
42     else
43     {
44         ins->next = where->next;
45         where->next = ins;
46         list->count = list->count + 1;
47     }
48 }
49
50 void insert_in_list(IntList* list, IntListElem* where, IntListElem* ins)
51 {
52     if (where == 0)
53     {
54         ins->next = list->first;
55         list->first = ins;
56         ++list->count;
57     }
58     else
59     {
60         ins->next = where->next;
61         where->next = ins;
62         list->count = list->count + 1;
63     }
64 }
65
```

```
66  IntListElem* remove_from_list(IntList* list , IntListElem* where)
67  {
68      IntListElem* p;
69
70      if (where == 0)
71      {
72          p = list->first;
73          if (p != 0)
74          {
75              list->first = p->next;
76              list->count = list->count - 1;
77          }
78          return p;
79      }
80
81      p = where->next;
82      if(p != 0)
83      {
84          where->next = p->next;
85          list->count = list->count - 1;
86      }
87      return p;
88  }
89
90
91  void print_List(IntList l)
92  {
93      for (IntListElem* p = l.first; p != 0 and p->value != 0; p = p->next)
94      {
95          std::cout << p->value <<  " ";
96      }
97  }
98
99  int haseigle(IntList* list)
100 {
101     bool meet = false;
102     int count = 0;
103     IntListElem* hase = list->first;
104     for (IntListElem* igel = list->first; igel != 0 and hase != 0; igel = igel->next)
105     {
106         ++count;
107         hase = hase->next;
108         if (hase != 0)
109         {
110             hase = hase->next;
111             if (hase != 0)
```

```
112             {
113                 if(hase->next == igel->next and meet == false)
114                 {
115                     meet = true;
116                     count = 0;
117                 }
118                 else if(hase->next == igel->next)
119                 {
120                     return count;
121                 }
122             }
123         }
124     }
125     return 0;
126 }
127
128 void createListandhase(int n,int k)
129 {
130     IntList mylist;
131     IntList myklist;
132     bool nandk0 = false;
133     empty_list(&mylist);
134     IntListElem* nextpointer = mylist.first;
135     if (n != 0)
136     {
137         for (int i = n-1; i >= 0; --i)
138         {
139             IntListElem* newelementpointer = new IntListElem;
140             newelementpointer->value = i;
141             insert_in_list_cycle(&mylist, nextpointer, newelementpointer);
142             nextpointer = newelementpointer;
143         }
144
145         if (k != 0)
146         {
147             empty_list(&myklist);
148             IntListElem* nextkpointer = myklist.first;
149             for (int i = k-1; i >= 0; --i)
150             {
151                 IntListElem* newelementpointer = new IntListElem;
152                 newelementpointer->value = i;
153                 insert_in_list(&myklist, nextkpointer, newelementpointer);
154                 nextkpointer = newelementpointer;
155             }
156             nextkpointer->next = mylist.first;
157         }
```

```
158             else
159             {
160                 myklist = mylist;
161             }
162         }
163         else
164         {
165             if (k != 0)
166             {
167                 empty_list(&myklist);
168                 IntListElem* nextkpointer = myklist.first;
169                 for (int i = k-1; i >= 0; --i)
170                 {
171                     IntListElem* newelementpointer = new IntListElem;
172                     newelementpointer->value = i;
173                     insert_in_list(&myklist, nextkpointer, newelementpointer);
174                     nextkpointer = newelementpointer;
175                 }
176             }
177             else//n = 0 and k = 0
178             {
179                 nandk0 = true;
180             }
181         }
182         if (not nandk0)
183         {
184         std::cout << haseigle(&myklist) << std::endl;
185         }
186         else
187         {
188             std::cout << 0 << std::endl;
189         }
190 }
191
192
193
194 int main()
195 {
196     createListandhase(5,3);
197     return 0;
198 }
```

Aufgabe

(a) Einfach verkettete Liste:

Listing 2: queuea.cc

```
1  #include <iostream>
2  #include <cstdlib>
3
4  struct IntListElem
5  {
6      IntListElem* next;
7      int value;
8  };
9
10 struct IntList
11 {
12     int count;
13     IntListElem* first;
14 };
15
16 void empty_list(IntList* l)
17 {
18     l->first = 0;
19     l->count = 0;
20 }
21
22 void insert_in_list(IntList* list, IntListElem* where, IntListElem* ins)
23 {
24     if (where == 0)
25     {
26         ins->next = list->first;
27         list->first = ins;
28         list->count = list->count + 1;
29     }
30     else
31     {
32         ins->next = where->next;
33         where->next = ins;
34         list->count = list->count + 1;
35     }
36 }
37
38 IntListElem* remove_from_list(IntList* list, IntListElem* where)
39 {
40     IntListElem* p;
41
42     if (where == 0)
43     {
44         p = list->first;
45         if (p != 0)
```

```
46          {
47              list->first = p->next;
48              list->count = list->count - 1;
49          }
50          return p;
51      }
52
53      p = where->next;
54      if(p != 0)
55      {
56          where->next = p->next;
57          list->count = list->count - 1;
58      }
59      return p;
60  }
61
62  void print_List(IntList l)
63  {
64      for (IntListElem* p = l.first; p != 0; p = p->next)
65      {
66          std::cout << p->value << std::endl;
67      }
68  }
69
70  IntListElem* getfirst(IntList* list)
71  {
72      return remove_from_list(list, 0);
73  }
74
75  void appendelement(IntList* list, IntListElem* element)
76  {
77      IntListElem* letztesp = 0;
78      for (IntListElem* p = list->first; p != 0; p = p->next)
79      {
80          letztesp = p;
81      }
82      insert_in_list(list, letztesp, element);
83  }
84
85  int main()
86  {
87      int n = 100000;
88      IntList mylist;
89      empty_list(&mylist);
90      IntListElem* nextpointer = mylist.first;
91      for (int i = 0; i < n; ++i)
```

```
92      {
93          IntListElem* newelementpointer = new IntListElem;
94          newelementpointer->value = rand();
95          appendelement(&mylist, newelementpointer);
96          nextpointer = newelementpointer;
97      }
98      for (int i = 0; i < n; ++i)
99      {
100         getfirst(&mylist);
101     }
102     print_List(mylist);
103     return 0;
104 }
```

(b) Doppelt verkettete Liste:

Listing 3: queueb.cc

```
1  #include <iostream>
2  #include <cstdlib>
3
4  struct IntListElem
5  {
6      IntListElem* next;
7      IntListElem* previous;
8      int value;
9  };
10
11 struct IntList
12 {
13     int count;
14     IntListElem* first;
15     IntListElem* last;
16 };
17
18 void empty_list(IntList* l)
19 {
20     l->first = 0;
21     l->last = 0;
22     l->count = 0;
23 }
24
25 void insert_in_list(IntList* list, IntListElem* where, IntListElem* ins)
26 {
27     if (where == 0)
28     {
29         ins->next = list->first;
```

```
30          list->first = ins;
31          list->count = list->count + 1;
32          //new
33          ins->previous = 0;
34      }
35      else
36      {
37          ins->next = where->next;
38          where->next = ins;
39          list->count = list->count + 1;
40          //new
41          ins->previous = where;
42      }
43  }
44
45  IntListElem* remove_from_list(IntList* list, IntListElem* where)
46  {
47      IntListElem* p;
48
49      if (where == 0)
50      {
51          p = list->first;
52          if (p != 0)
53          {
54              list->first = p->next;
55              list->count = list->count - 1;
56          }
57          return p;
58      }
59
60      p = where->next;
61      if(p != 0)
62      {
63          where->next = p->next;
64          list->count = list->count - 1;
65          //wenn p das letze element der liste war,
66          //dann muss list->last auf das vorletzte gesetzt werden
67          if(p->next = 0)
68          {
69              list->last = where;
70          }
71      }
72      return p;
73  }
74
75  void print_List(IntList l)
```

```
76  {
77      for (IntListElem* p = l.first; p != 0; p = p->next)
78      {
79          std::cout << p->value << std::endl;
80      }
81  }
82
83  IntListElem* getfirst(IntList* list)
84  {
85      return remove_from_list(list, 0);
86  }
87
88  void appendelement(IntList* list, IntListElem* element)
89  {
90      insert_in_list(list, list->last, element);
91  }
92
93  int main()
94  {
95      int n = 100000;
96      IntList mylist;
97      empty_list(&mylist);
98      IntListElem* nextpointer = mylist.first;
99      for (int i = 0; i < n; ++i)
100     {
101         IntListElem* newelementpointer = new IntListElem;
102         newelementpointer->value = rand();
103         appendelement(&mylist, newelementpointer);
104         nextpointer = newelementpointer;
105     }
106     for (int i = 0; i < n; ++i)
107     {
108         getfirst(&mylist);
109     }
110     print_List(mylist);
111     return 0;
112 }
```

(c) Für die einfach verkettete Liste erhalten wir bei 100000 zufällig generierten Zahlen eine Laufzeit von 23.616 Sekunden, für eine doppelt verkettete Liste erhalten wir bei 100000 zufällig generierten Zahlen eine Laufzeit von 0.448 Sekungen. Bei der einfach verketteten Liste muss nämlich jedes Mal die gesamte Liste durchsucht werden, bis man das letze Element findet, um daran ein weiteres Element anzuhängen. Bei einer doppelt verketteten Liste hingegen kann man einfach direkt über den Pointer list −>last auf das letze Element zugreifen. Die getfirst -Funktion benötigt stets die gleiche Zeit, unabhängig von der Anzahl der Elemente. Der Unterschied liegt in der appendelement-Funktion. Diese hat bei einer einfach verketteten Liste eine lineare Komplexität,

bei einer doppelt verketteten Liste benötigt sie hingegen stets die selbe Laufzeit. Beim Hinzufügen und Wegnehmen von n Elementen hat also die einfach verkettete Liste eine quadratische Komplexität, die doppelt verkettete Liste hingegen eine lineare Komplexität.