Spojová implementace lineárních datových struktur

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Prezentace ke dni 12. února 2020



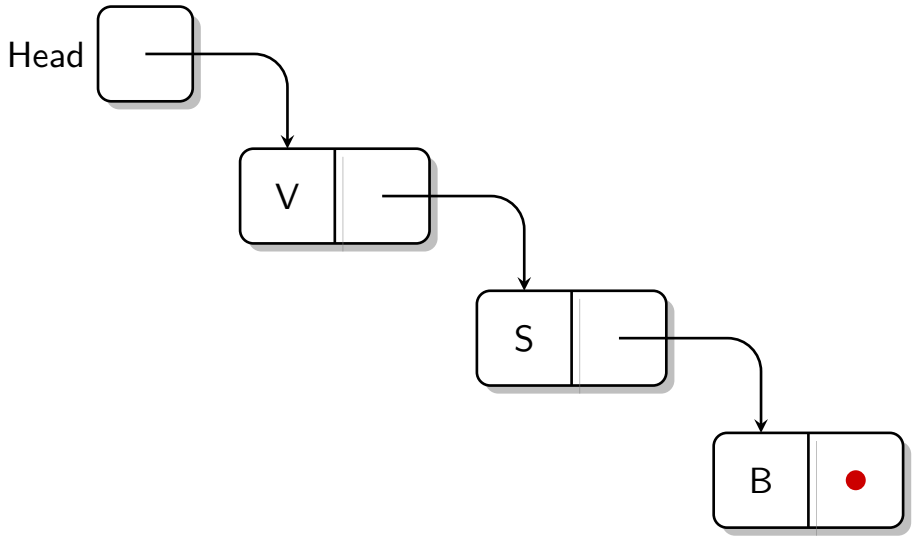


Osnova přednášky

Spojová implementace lineárních datových struktur

- Seznam
  - Popis
  - Použití
- Implementace obousměrného seznamu
  - Implementace pomocí pole
  - Spojová implementace

Seznam (List)



Charakteristika

- nejobecnější lineární datová struktura,
- operace nejsou striktně určeny,
- existuje mnoho variant.

Atributy

- atributy seznamu závisí na konkrétní implementaci,
- v nejjednodušším případě jen nutný odkaz na první prvek seznamu, tzv. **hlavu seznamu**.

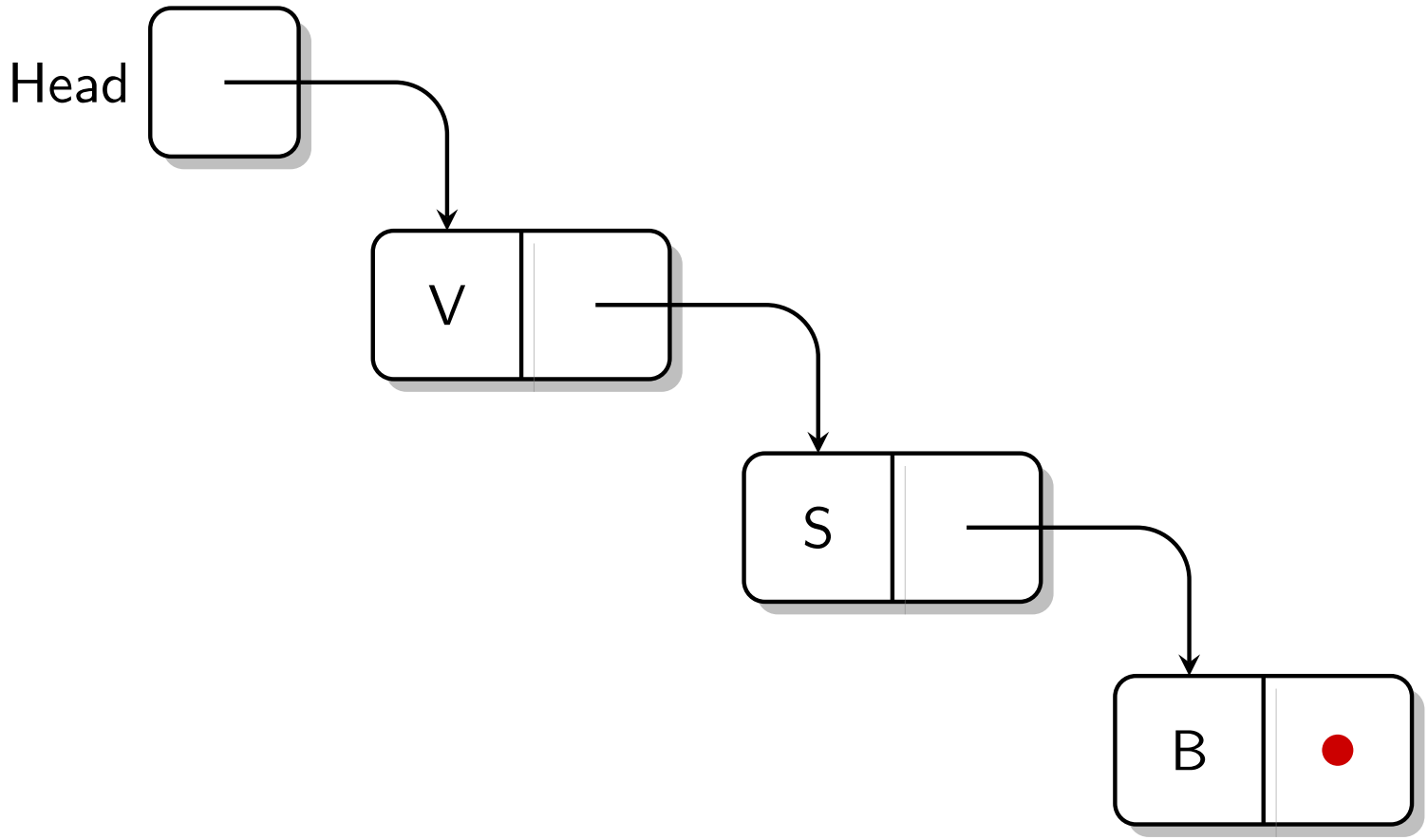
Seznam – jednosměrný seznam

- složen z položek (items),
- položky obsahují data a odkaz na další položku,
- každá položka odkazuje na následovníka,
- k položkám lze přistupovat sekvenčně,
- přímý přístup na základě indexu je složitý (průchod cyklem),
- pohyb seznamem dozadu je také problém,
- konec seznamu – speciální odkaz „nikam“, většinou pojmenován **nil**, **NULL**, **nullptr** či **Nothing**.

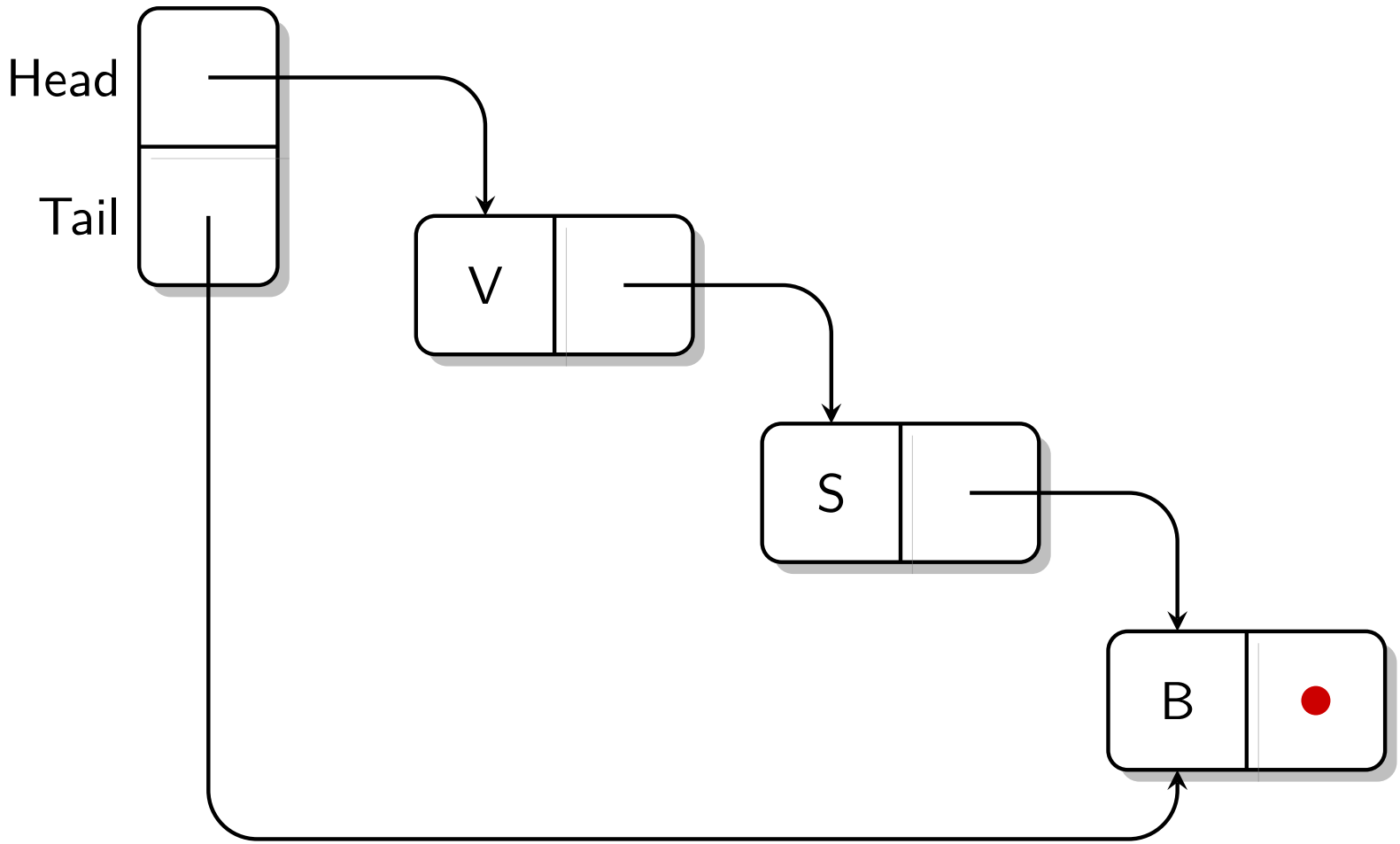
Seznam, varianty

- jednosměrný seznam – nejjednodušší varianta, odkaz pouze na následníka,
- obousměrný seznam – položka obsahuje odkaz na předchůdce i následníka,
- kruhový seznam – hlava a ocas seznamu splývají.

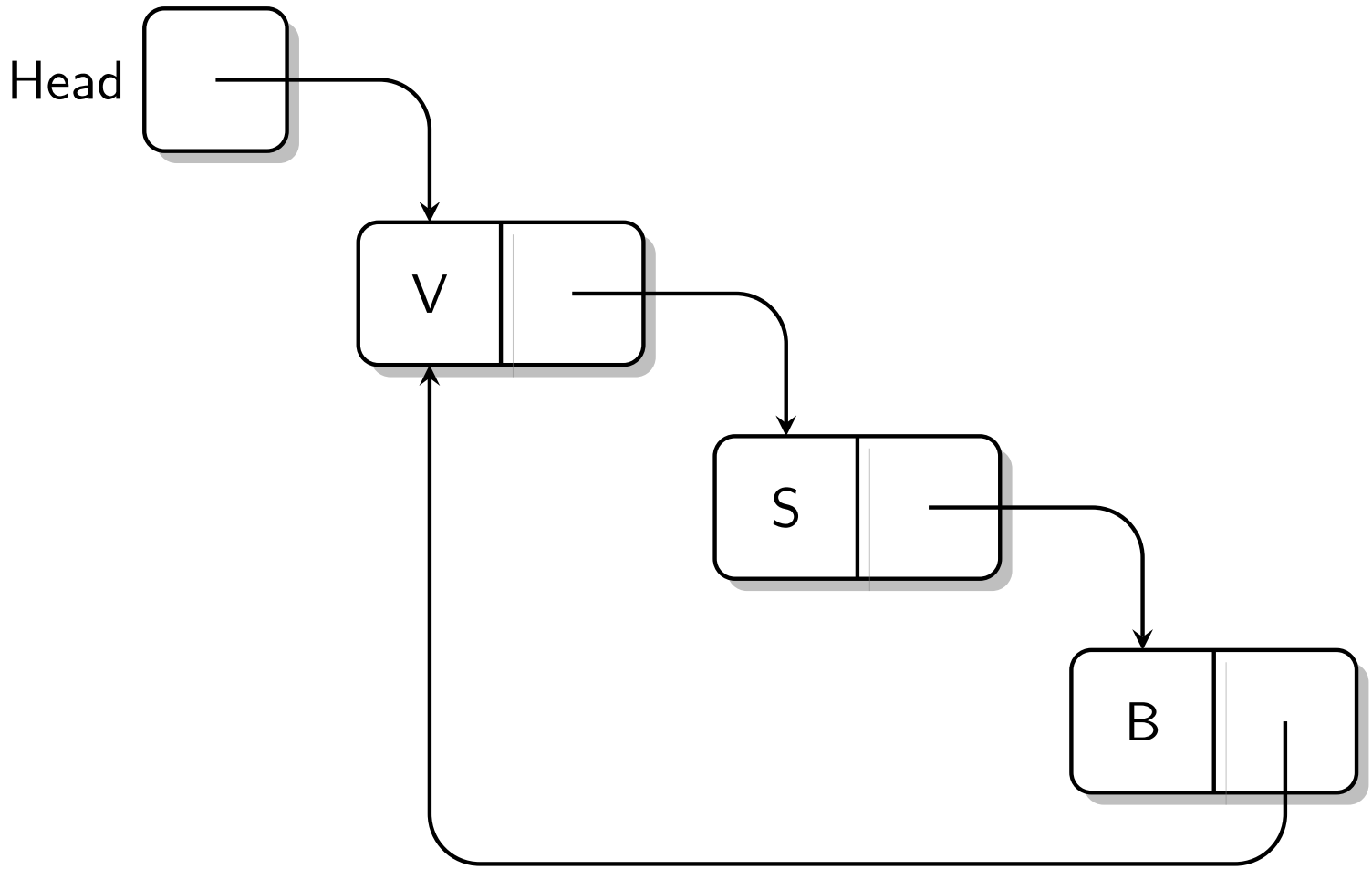
Seznam – jednosměrný, jen hlava seznamu



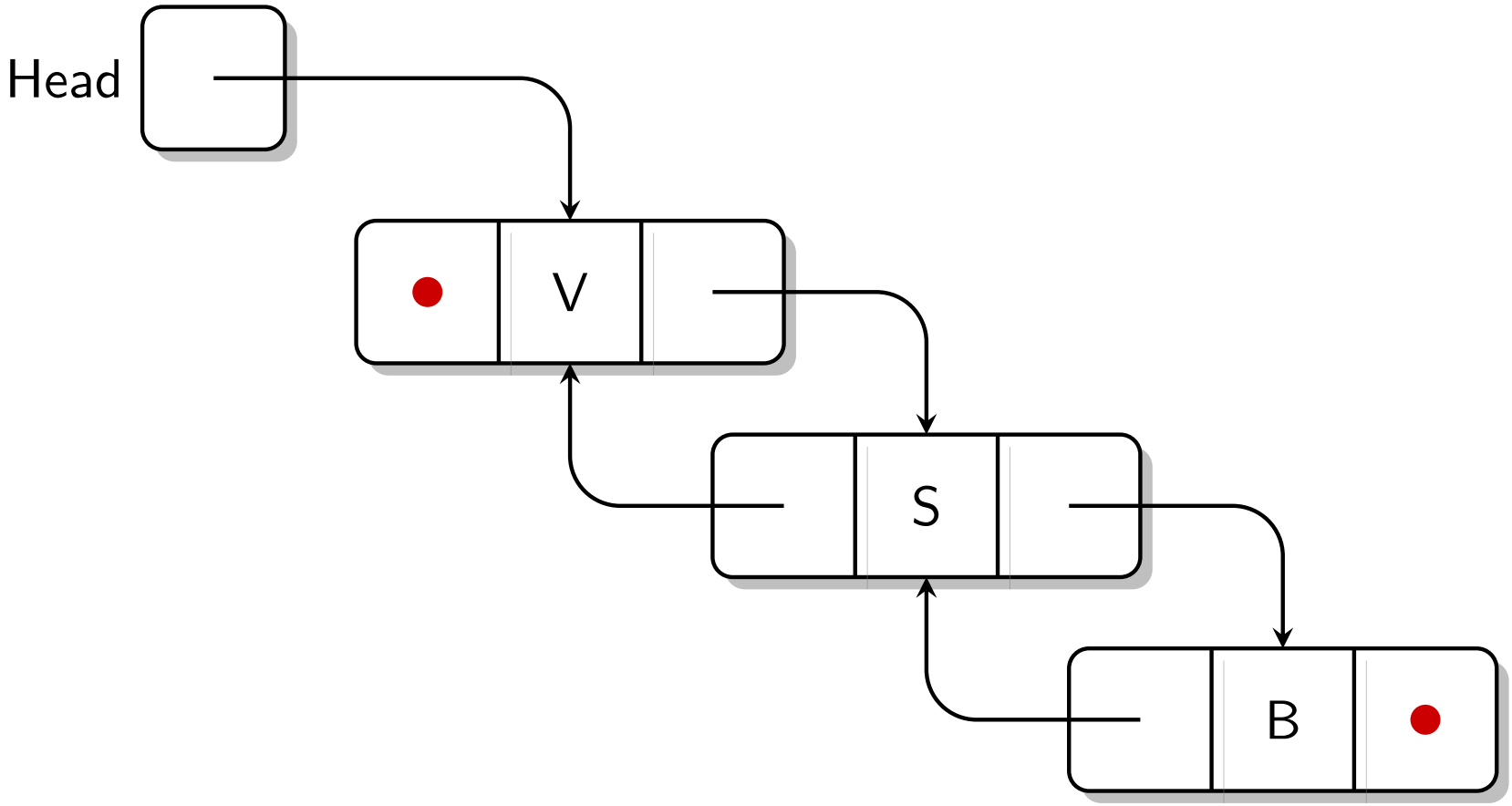
Seznam – jednosměrný, hlava i ocas seznamu



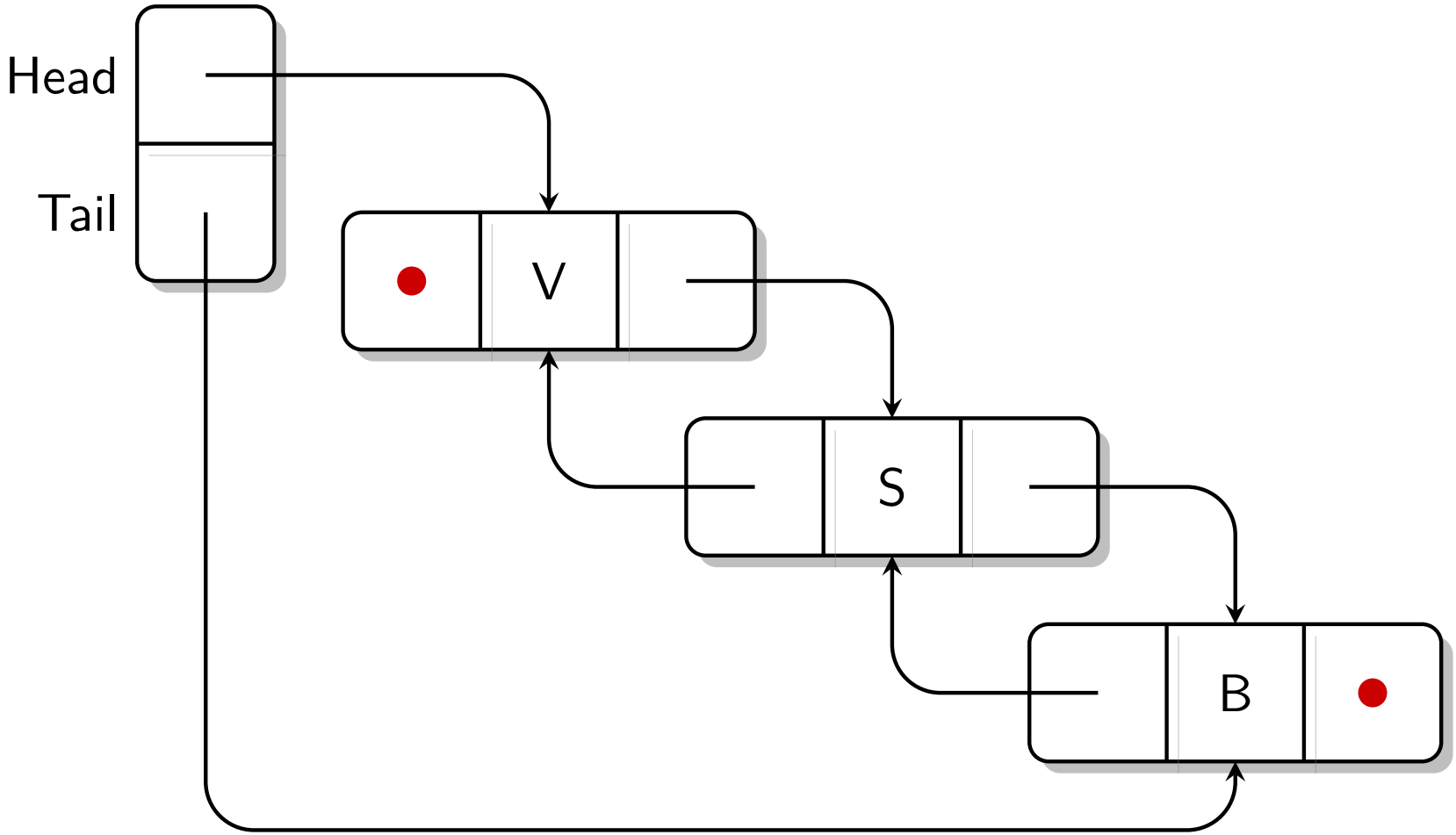
Seznam – jednosměrný, kruhový



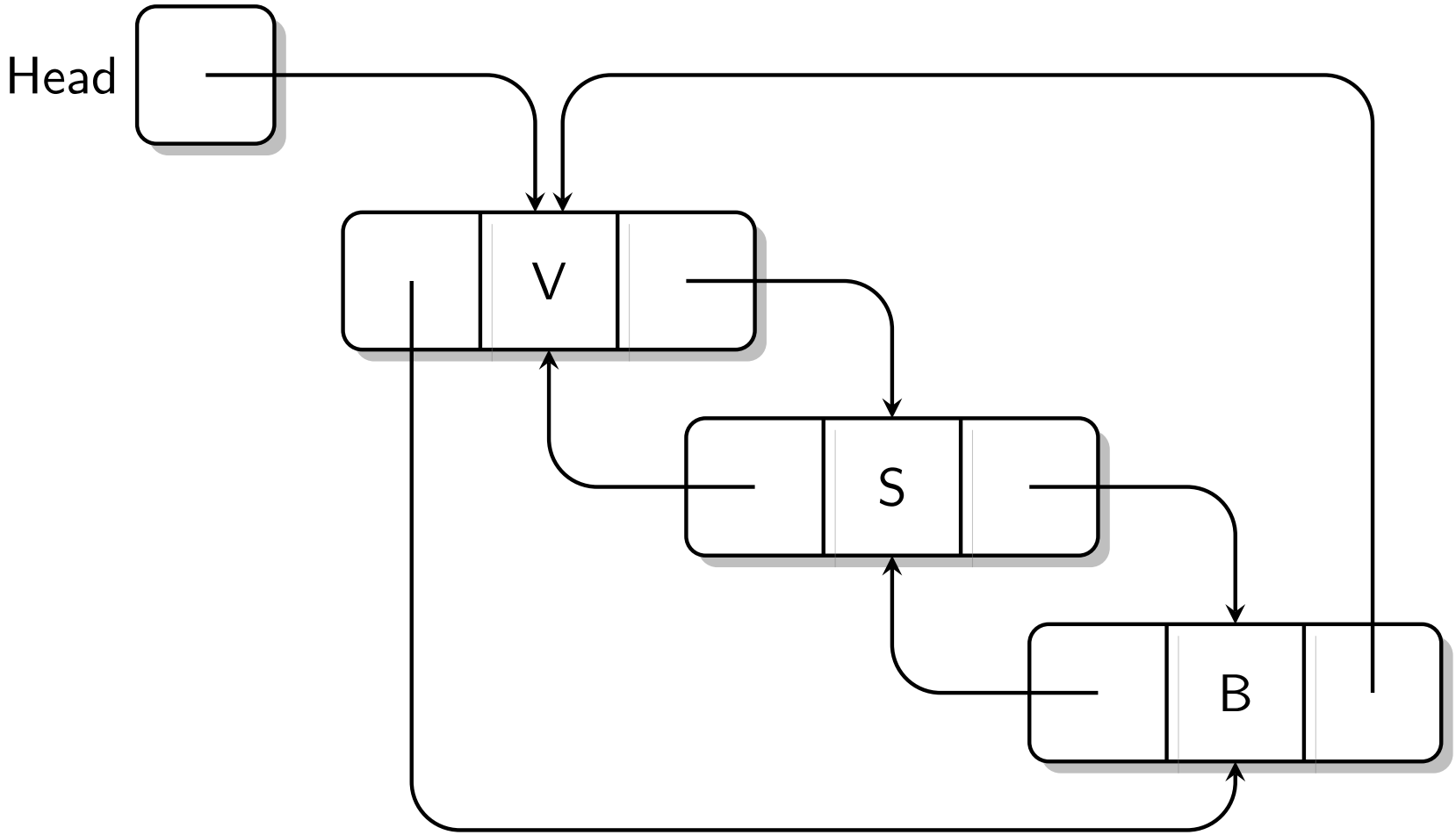
Seznam – obousměrný, jen hlava seznamu



Seznam – obousměrný, hlava i ocas seznamu



Seznam – obousměrný, kruhový



Seznam – použití

- obecná datová struktura,
- dynamicky se měnící kolekce prvků,
- čili kdekoliv – například velice primitivní databáze,
- otázkou ale je složitost jednotlivých operací se seznamem – třídění, vyhledávání prvku atd.

Seznam – možnosti implementace

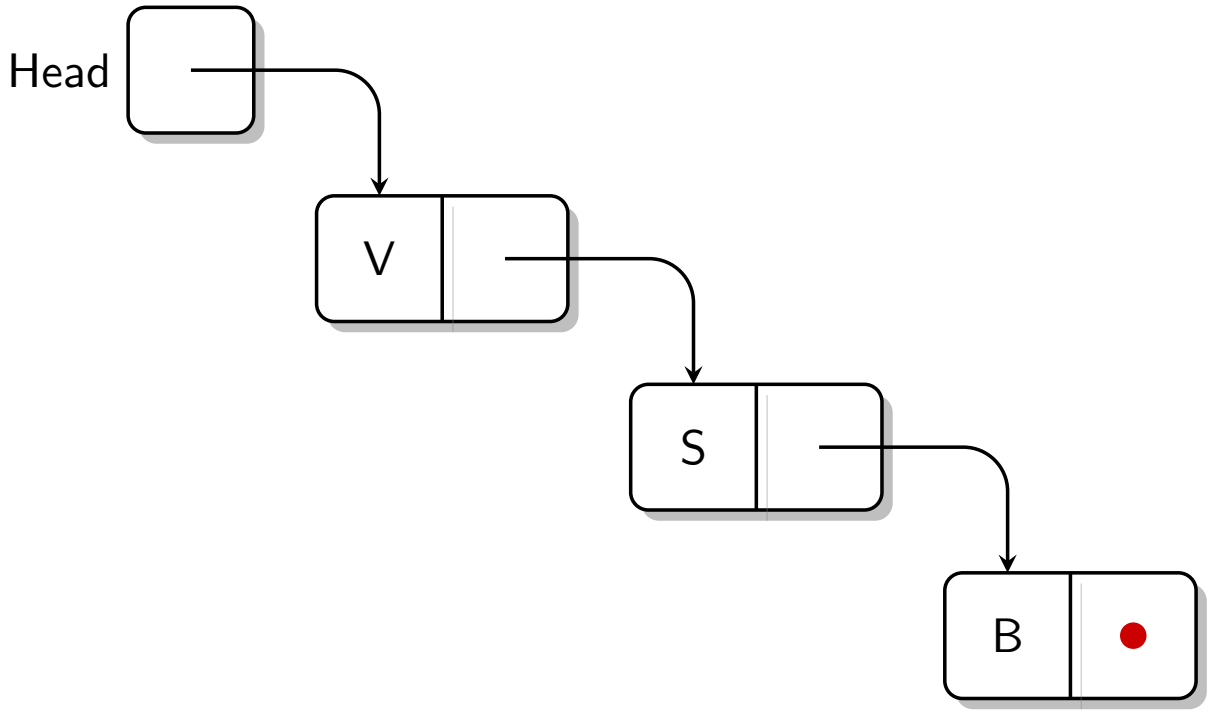
- pomocí pole nebo
- pomocí spojových struktur

Seznam – implementace pomocí pole

- pole délky n – omezená kapacita, ale rychlá manipulace,
- odkazy na předchůdce resp. následníka jsou reprezentovány indexy do pole – čísly `int`,
- uložení položek seznamu v poli nemusí odpovídat logickému uspořádání položek v seznamu,
- správa paměti:
  - evidence volného prostoru v poli,
  - přidělování prostoru z evidence pro nové položky při vkládání,
  - navracení prostoru do evidence při mazání položky.

Logická struktura vs. uložení dat

Logická struktura



Paměťová struktura

Head: 8
Free: 3

	0	1	2	3	4	5	6	7	8	9
Value			**S**			**B**			**V**	
Next	1	4	5	9	null	null	7	0	2	6

Garbagge Collector (GC)

- mechanismus pro evidenci volného prostoru v poli,
- doslova „sběr odpadu“ – sběr nepoužívané paměti,
- index **Free** ukazuje na první volnou položku,
- v nepoužívaných položkách seznamu je využita složka **Next**, která odkazuje na další volnou položku,
- při alokaci položky si vezmeme první volnou a **Free** posuneme přes **Next** na další volnou položku,
- při dealokaci vložíme uvolňovanou položku před položku na kterou odkazuje **Free**,
- GC tedy pracuje na principu LIFO, chová tedy jako **zásobník** volného místa.

Seznam pomocí pole – deklarace struktur

Konstanty

```
1 const int ListSize = 5;
2
3 const int Nothing = -1;
```

Položka seznamu

```
1 struct ListItem
2 {
3    string Value;
4    int Prev;
5    int Next;
6 };
```

Vlastní seznam

```
1 struct List
2 {
3    ListItem Items[ListSize];
4    int Head;
5    int Tail;
6    int Free;
7 };
```

Seznam pomocí pole – deklarace funkcí

```
1  void Init(List& L);
2  void Init(List& L, const string& InitValue);
3  void Init(List& L, const string InitValues[], const int N);
4  void Init(List& L, const List& Other);
5
6  void Prepend(List& L, const string& NewValue);
7  void Append(List& L, const string& NewValue);
8  void InsertBefore(List& L, const int CurrentItem, const string&
       NewValue);
9  void InsertAfter(List& L, const int CurrentItem, const string& NewValue
       );
10
11 void RemoveHead(List& L);
12 void RemoveTail(List& L);
13 void RemoveFirst(List& L, const string& ValueToRemove);
14 void RemoveLast(List& L, const string& ValueToRemove);
15 void RemoveAll(List& L, const string& ValueToRemove);
16 void Remove(List& L, const int ItemToRemove);
17 void Clear(List& L);
```

Seznam pomocí pole – deklarace funkcí (pokrač.)

```
18
19 int Search(const List& L, const string& Value);
20 int ReverseSearch(const List& L, const string& Value);
21
22 bool Contains(const List& L, const string& Value);
23
24 bool IsEmpty(const List& L);
25 int Count(const List& L);
26
27 void Report(const List& L);
28 void ReportStructure(const List& L);
29
30 void InternalCreateSingleElementList(List& L, const string& NewValue);
31 void InternalRemove(List& L, const int ItemToDelete);
32 int InternalAllocateItem(List& L);
33 void InternalFreeItem(List& L, const int Index);
```

Seznam pomocí pole – implementace

```
1  void Init(List& L)
2  {
3     Clear(L);
4  }
5
6  void Init(List& L, const string& InitValue)
7  {
8     Init(L);
9     InternalCreateSingleElementList(L, InitValue);
10 }
11
12 void Init(List& L, const string InitValues[], const int N)
13 {
14    Init(L);
15    for (int i = 0; i < N; i++)
16    {
17       Append(L, InitValues[i]);
18    }
19 }
```

Seznam pomocí pole – implementace (pokrač.)

```
20
21 void Init(List& L, const List& Other)
22 {
23    Init(L);
24    for (int p = Other.Head; p != Nothing; p = Other.Items[p].Next)
25    {
26       Append(L, Other.Items[p].Value);
27    }
28 }
29
30 void Prepend(List& L, const string& NewValue)
31 {
32    if (IsEmpty(L))
33    {
34       InternalCreateSingleElementList(L, NewValue);
35    }
36    else
37    {
38       int NewItemIndex = InternalAllocateItem(L);
```

Seznam pomocí pole – implementace (pokrač.)

```
39        L.Items[NewItemIndex].Value = NewValue;
40        L.Items[NewItemIndex].Prev = Nothing;
41        L.Items[NewItemIndex].Next = L.Head;
42        L.Items[L.Head].Prev = NewItemIndex;
43        L.Head = NewItemIndex;
44     }
45  }
46
47  void Append(List& L, const string& NewValue)
48  {
49     if (IsEmpty(L))
50     {
51        InternalCreateSingleElementList(L, NewValue);
52     }
53     else
54     {
55        int NewItemIndex = InternalAllocateItem(L);
56        L.Items[NewItemIndex].Value = NewValue;
57        L.Items[NewItemIndex].Prev = L.Tail;
```

Seznam pomocí pole – implementace (pokrač.)

```
58        L.Items[NewItemIndex].Next = Nothing;
59        L.Items[L.Tail].Next = NewItemIndex;
60        L.Tail = NewItemIndex;
61     }
62  }
63
64  void InsertBefore(List& L, const int CurrentItem, const string&
        NewValue)
65  {
66     if (CurrentItem != Nothing)
67     {
68        if (CurrentItem == L.Head)
69        {
70           Prepend(L, NewValue);
71        }
72        else
73        {
74           int NewItemIndex = InternalAllocateItem(L);
75           L.Items[NewItemIndex].Value = NewValue;
```

Seznam pomocí pole – implementace (pokrač.)

```
76          int P = L.Items[CurrentItem].Prev;
77          L.Items[P].Next = NewItemIndex;
78          L.Items[NewItemIndex].Prev = P;
79          L.Items[CurrentItem].Prev = NewItemIndex;
80          L.Items[NewItemIndex].Next = CurrentItem;
81       }
82    }
83 }
84
85 void InsertAfter(List& L, const int CurrentItem, const string& NewValue
       )
86 {
87    if (CurrentItem != Nothing)
88    {
89       if (CurrentItem == L.Tail)
90       {
91          Append(L, NewValue);
92       }
93       else
```

Seznam pomocí pole – implementace (pokrač.)

```
94          {
95              int NewItemIndex = InternalAllocateItem(L);
96              L.Items[NewItemIndex].Value = NewValue;
97              int N = L.Items[CurrentItem].Next;
98              L.Items[N].Prev = NewItemIndex;
99              L.Items[NewItemIndex].Next = N;
100             L.Items[CurrentItem].Next = NewItemIndex;
101             L.Items[NewItemIndex].Prev = CurrentItem;
102         }
103     }
104 }
105
106 void RemoveHead(List& L)
107 {
108     if (!IsEmpty(L))
109     {
110         InternalRemove(L, L.Head);
111     }
112 }
```

Seznam pomocí pole – implementace (pokrač.)

```
113
114 void RemoveTail(List& L)
115 {
116    if (!IsEmpty(L))
117    {
118       InternalRemove(L, L.Tail);
119    }
120 }
121
122 void RemoveFirst(List& L, const string& ValueToRemove)
123 {
124    int p = Search(L, ValueToRemove);
125    if (p != Nothing)
126    {
127       InternalRemove(L, p);
128    }
129 }
130
131 void RemoveLast(List& L, const string& ValueToRemove)
```

Seznam pomocí pole – implementace (pokrač.)

```
132 {
133    int p = ReverseSearch(L, ValueToRemove);
134    if (p != Nothing)
135    {
136       InternalRemove(L, p);
137    }
138 }
139
140 void RemoveAll(List& L, const string& ValueToRemove)
141 {
142    int p;
143    while ((p = Search(L, ValueToRemove)) != Nothing)
144    {
145       InternalRemove(L, p);
146    }
147 }
148
149 void Remove(List& L, const int ItemToRemove)
150 {
```

Seznam pomocí pole – implementace (pokrač.)

```
151     if (ItemToRemove != Nothing)
152     {
153        InternalRemove(L, ItemToRemove);
154     }
155 }
156
157 void Clear(List& L)
158 {
159     L.Head = Nothing;
160     L.Tail = Nothing;
161     for (int i = 0; i < ListSize; i++)
162     {
163        L.Items[i].Next = i + 1;
164     }
165     L.Items[ListSize - 1].Next = Nothing;
166     L.Free = 0;
167 }
168
169 int Search(const List& L, const string& Value)
```

Seznam pomocí pole – implementace (pokrač.)

```
170 {
171    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
172    {
173       if (L.Items[p].Value == Value)
174       {
175          return p;
176       }
177    }
178    return Nothing;
179 }
180
181 int ReverseSearch(const List& L, const string& Value)
182 {
183    for (int p = L.Tail; p != Nothing; p = L.Items[p].Prev)
184    {
185       if (L.Items[p].Value == Value)
186       {
187          return p;
188       }
```

Seznam pomocí pole – implementace (pokrač.)

```
189     }
190     return Nothing;
191 }
192
193 bool Contains(const List& L, const string& Value)
194 {
195     return Search(L, Value) != Nothing;
196 }
197
198 bool IsEmpty(const List& L)
199 {
200     return L.Head == Nothing;
201 }
202
203 int Count(const List& L)
204 {
205     int counter = 0;
206     for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
207     {
```

Seznam pomocí pole – implementace (pokrač.)

```
208         counter += 1;
209     }
210     return counter;
211 }
212
213 void Report(const List& L)
214 {
215     for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
216     {
217         cout << L.Items[p].Value << "\t";
218     }
219     cout << endl;
220 }
221
222 void ReportStructure(const List& L)
223 {
224     cout << "L.Head: " << L.Head << endl;
225     cout << "L.Tail: " << L.Tail << endl;
226     for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
```

Seznam pomocí pole – implementace (pokrač.)

```
227     {
228         cout << "Item address: " << p << endl;
229         cout << "Value: " << L.Items[p].Value << endl;
230         cout << "Prev: " << L.Items[p].Prev << endl;
231         cout << "Next: " << L.Items[p].Next << endl;
232         cout << endl;
233     }
234 }
235
236 void InternalCreateSingleElementList(List& L, const string& NewValue)
237 {
238     L.Head = L.Tail = InternalAllocateItem(L);
239     L.Items[L.Head].Value = NewValue;
240     L.Items[L.Head].Prev = Nothing;
241     L.Items[L.Head].Next = Nothing;
242 }
243
244 void InternalRemove(List& L, const int ItemToDeleteIndex)
245 {
```

Seznam pomocí pole – implementace (pokrač.)

```
246     if (ItemToDeleteIndex == L.Head && ItemToDeleteIndex == L.Tail)
247     {
248        L.Head = Nothing;
249        L.Tail = Nothing;
250     }
251     else
252     {
253        if (ItemToDeleteIndex == L.Head)
254        {
255           L.Head = L.Items[L.Head].Next;
256           L.Items[L.Head].Prev = Nothing;
257        }
258        else
259        {
260           if (ItemToDeleteIndex == L.Tail)
261           {
262              L.Tail = L.Items[L.Tail].Prev;
263              L.Items[L.Tail].Next = Nothing;
264           }
```

Seznam pomocí pole – implementace (pokrač.)

```
265          else
266          {
267             int P = L.Items[ItemToDeleteIndex].Prev;
268             int N = L.Items[ItemToDeleteIndex].Next;
269             L.Items[P].Next = N;
270             L.Items[N].Prev = P;
271          }
272       }
273    }
274    InternalFreeItem(L, ItemToDeleteIndex);
275 }
276
277 int InternalAllocateItem(List& L)
278 {
279    if (L.Free != Nothing)
280    {
281       int p = L.Free;
282       L.Free = L.Items[L.Free].Next;
283       return p;
```

Seznam pomocí pole – implementace (pokrač.)

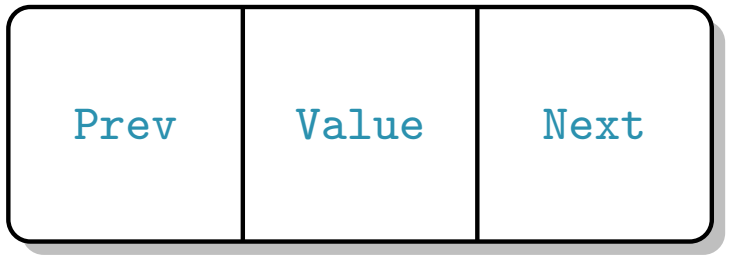
```
284     }
285     cerr << "List is full" << endl;
286     return Nothing;
287 }
288
289 void InternalFreeItem(List& L, const int Index)
290 {
291     L.Items[Index].Next = L.Free;
292     L.Free = Index;
293 }
```

Spojová implementace

- jednotlivé položky seznamu jsou samostatně alokovány heap managerem,
- vazby mezi položkami realizovány pomocí ukazatelů,
- velikost seznamu omezena dostupnou pamětí,
- nebezpečí úniku paměti (memory leak) – nutnost precizní práce s ukazateli a korektní dealokace položek.

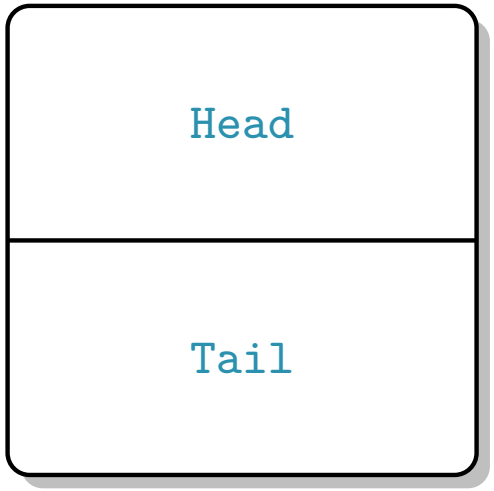
Položka seznamu

```
1 struct ListItem
2 {
3    string Value;
4    ListItem* Prev;
5    ListItem* Next;
6 };
```



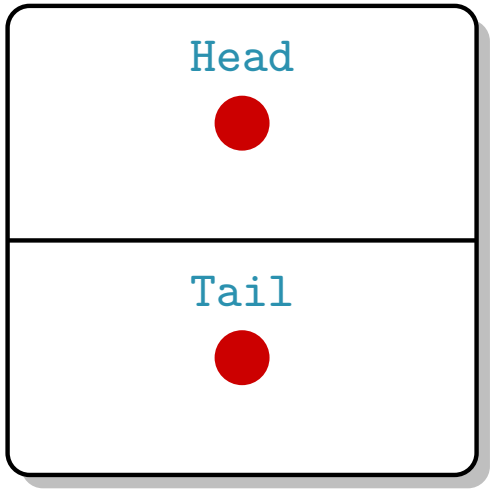
Datová reprezentace seznamu

```
1 struct List
2 {
3    ListItem* Head;
4    ListItem* Tail;
5 };
```



Inicializace prázdného seznamu

```
1 void Init(List& L)
2 {
3    L.Head = nullptr;
4    L.Tail = nullptr;
5 }
```



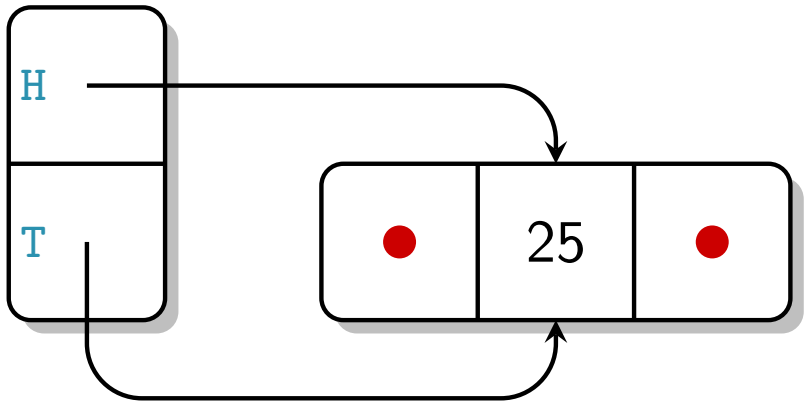
Inicializace seznamu jedním prvkem

```
7 void Init(List& L, const string& InitValue)
8 {
9    InternalCreateSingleElementList(L, InitValue);
10 }
```

Inicializaci převedeme na volání další, interní, funkce – vytvoření seznamu s jediným prvkem se totiž v kódu několikrát opakuje.

Vytvoření seznamu s jediným prvkem

```
232 void InternalCreateSingleElementList(List& L, const string& NewValue)
233 {
234     L.Head = L.Tail = new ListItem;
235     L.Head->Value = NewValue;
236     L.Head->Prev = nullptr;
237     L.Head->Next = nullptr;
238 }
```



Vložení položky do seznamu

Vložení položky do seznamu můžeme rozdělit na dvě fáze:

1. alokace nové položky v paměti a
2. zapojení nové položky do struktury seznamu – navázání vazeb mezi stávajícími položkami a novou položkou. Zde mohou nastat celkem čtyři situace, podle toho kam novou položku vkládáme:
   - 2.1 před hlavu seznamu – funkce `Prepend`,
   - 2.2 za ocas fronty – funkce `Append`,
   - 2.3 před danou položku – funkce `InsertBefore` a
   - 2.4 za danou položku – funkce `InsertAfter`.

Vložení položky do seznamu – před hlavu seznamu

Zde mohou nastat dvě situace:

1. seznam je prázdný – vytváříme seznam s jediným prvkem a
2. seznam není prázdný.

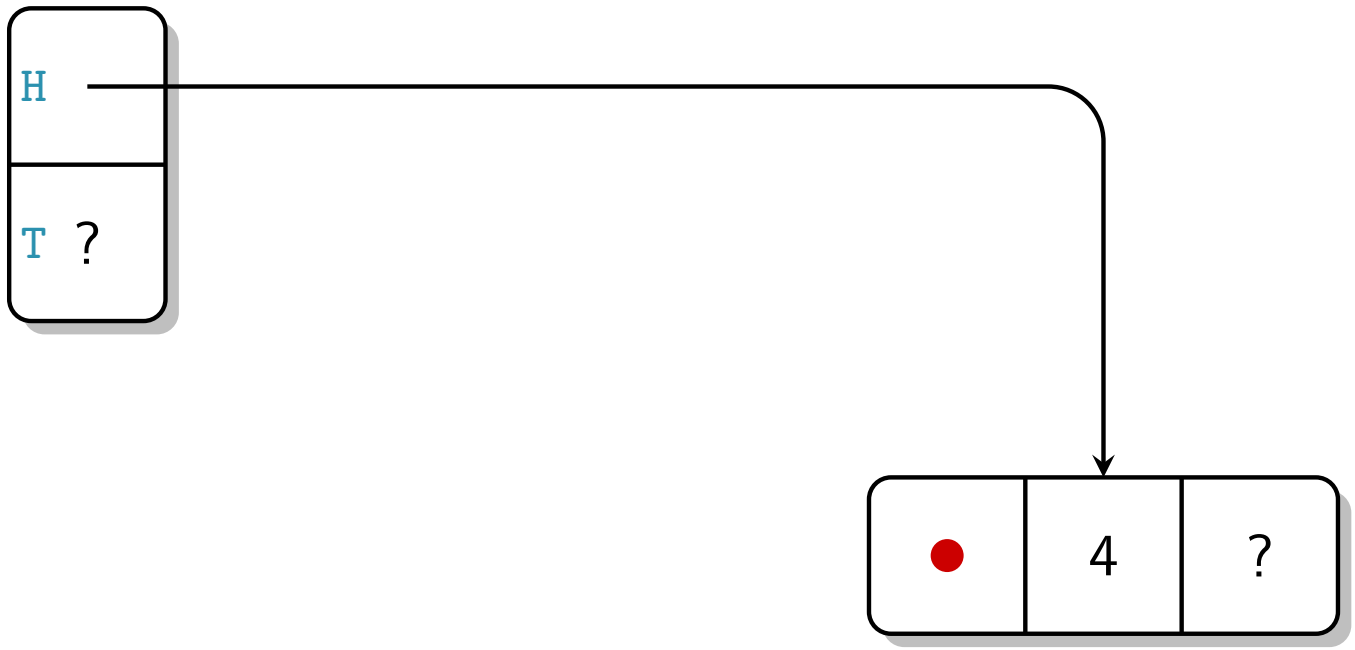
```
30 void Prepend(List& L, const string& NewValue)
31 {
32    if (IsEmpty(L))
33    {
34       InternalCreateSingleElementList(L, NewValue);
35    }
36    else
37    {
38       ListItem* NewItem = new ListItem;
39       NewItem->Value = NewValue;
40       NewItem->Prev = nullptr;
41       NewItem->Next = L.Head;
42       L.Head->Prev = NewItem;
43       L.Head = NewItem;
```

Vložení položky do seznamu – před hlavu seznamu (pokrač.)

```
44      }
45  }
```

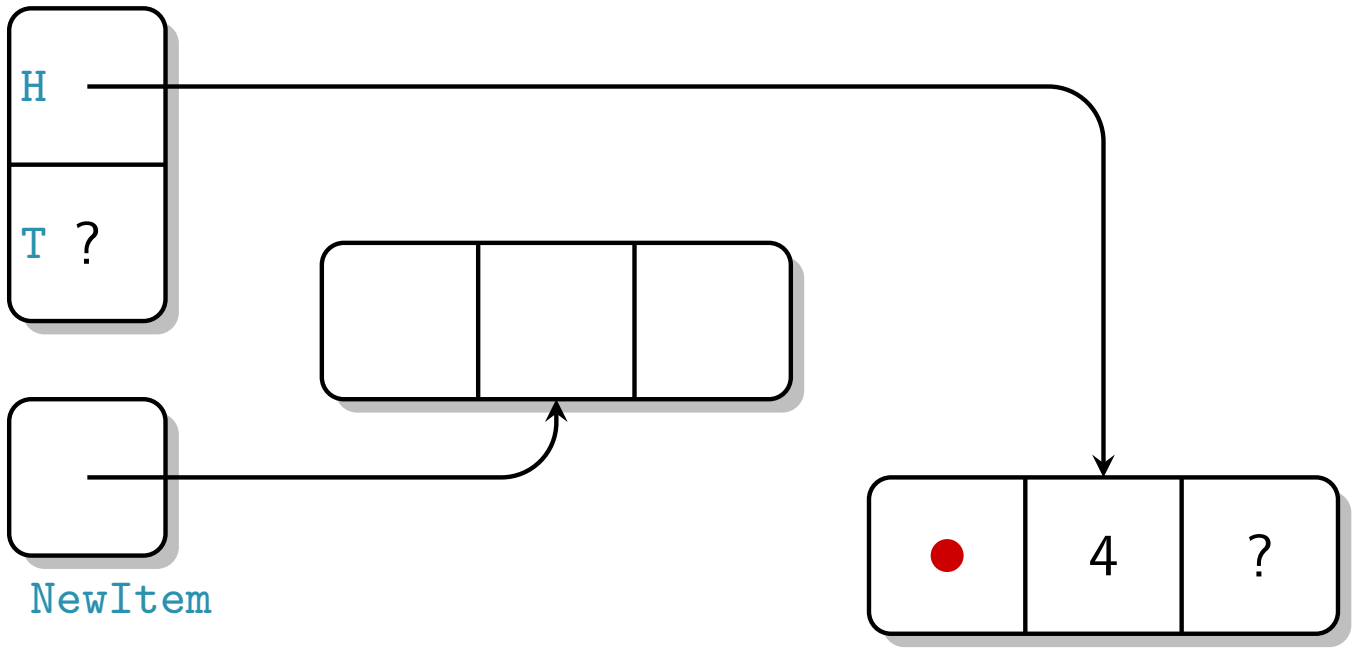
Vložení položky před hlavu seznamu – neprázdný seznam 1

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



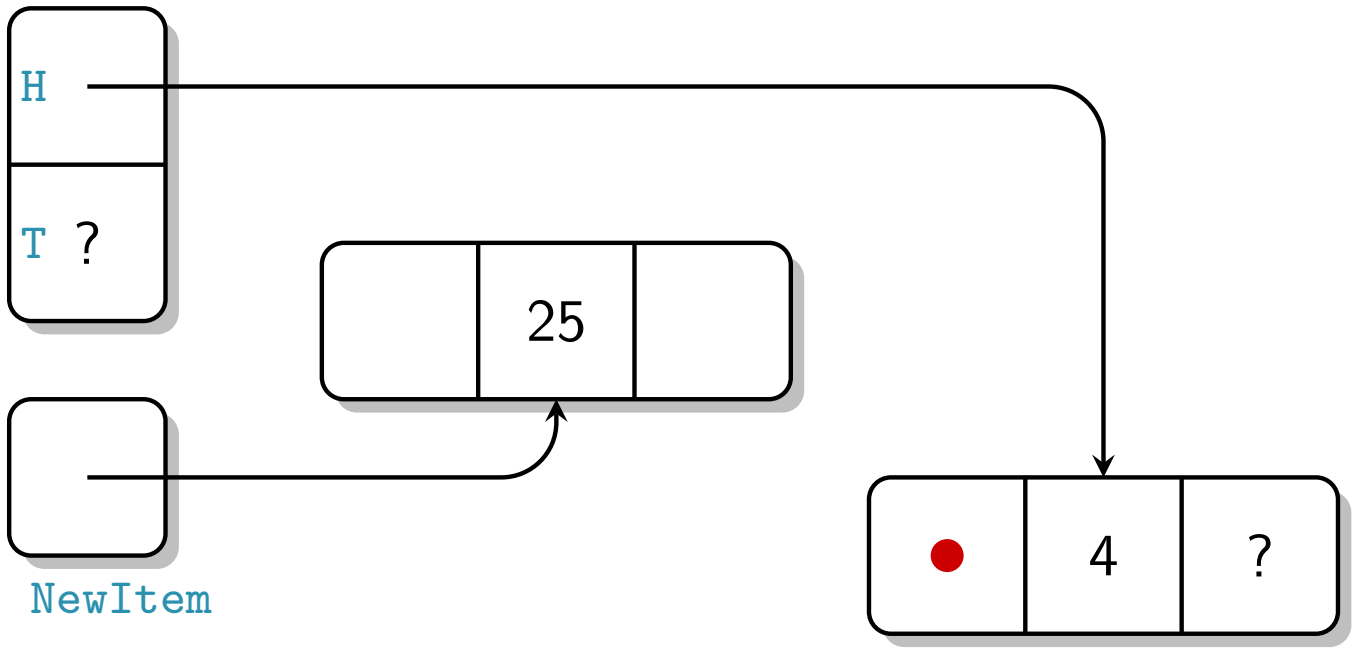
Vložení položky před hlavu seznamu – neprázdný seznam 2

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



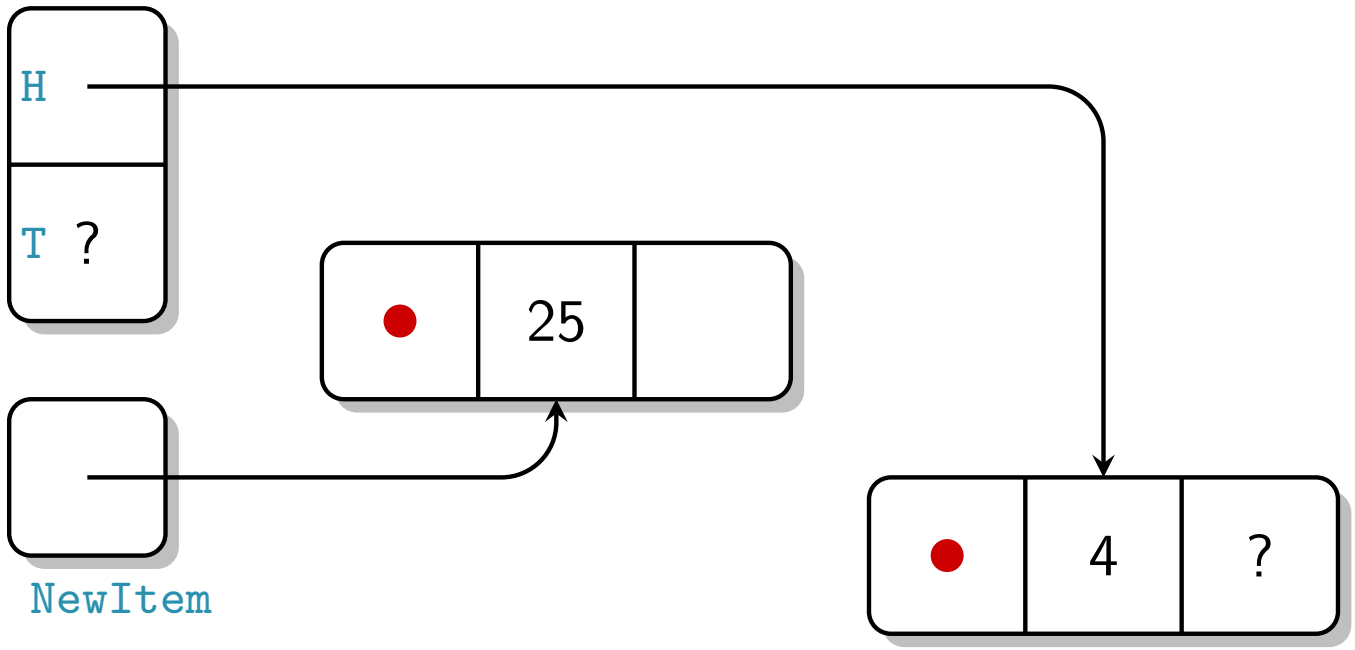
Vložení položky před hlavu seznamu – neprázdný seznam 3

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



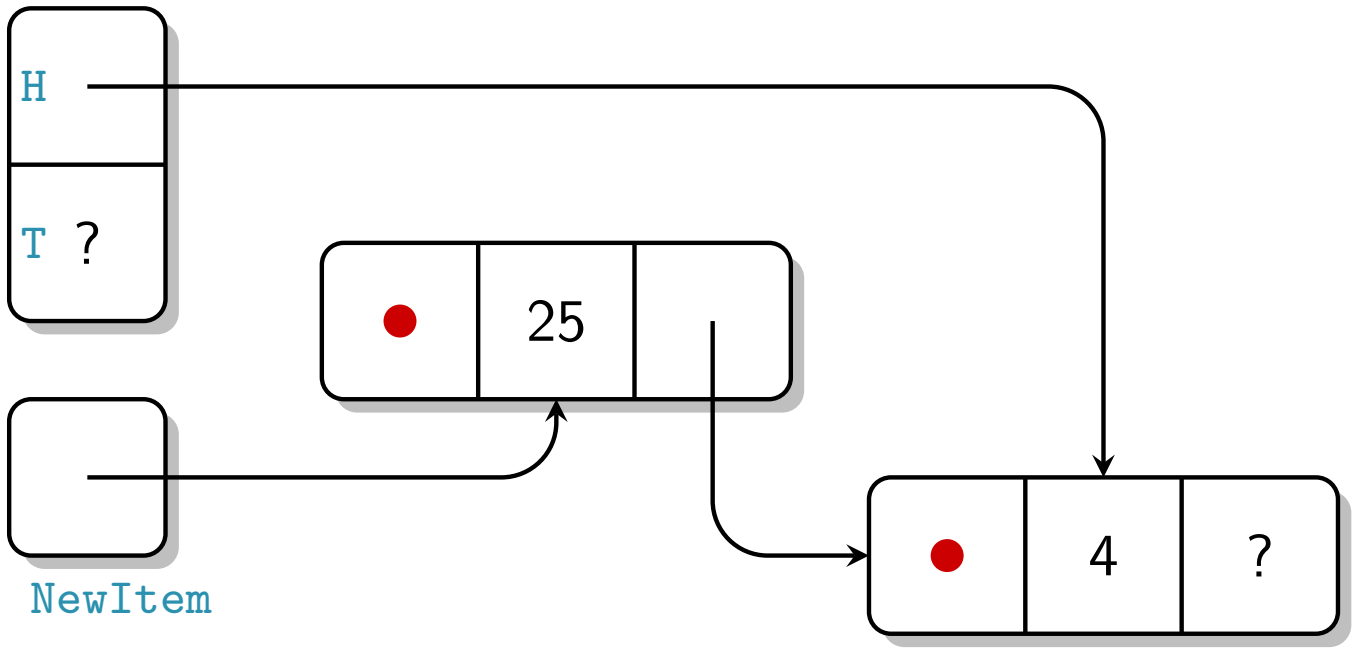
Vložení položky před hlavu seznamu – neprázdný seznam 4

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



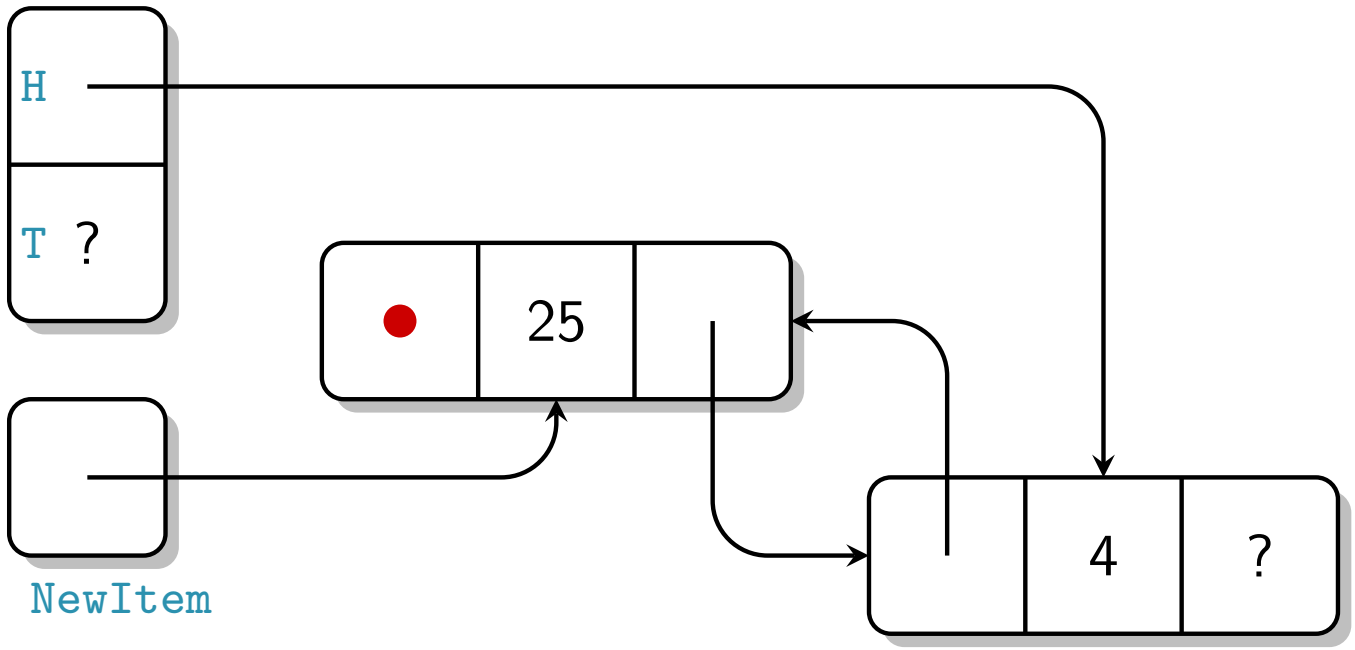
Vložení položky před hlavu seznamu – neprázdný seznam 5

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



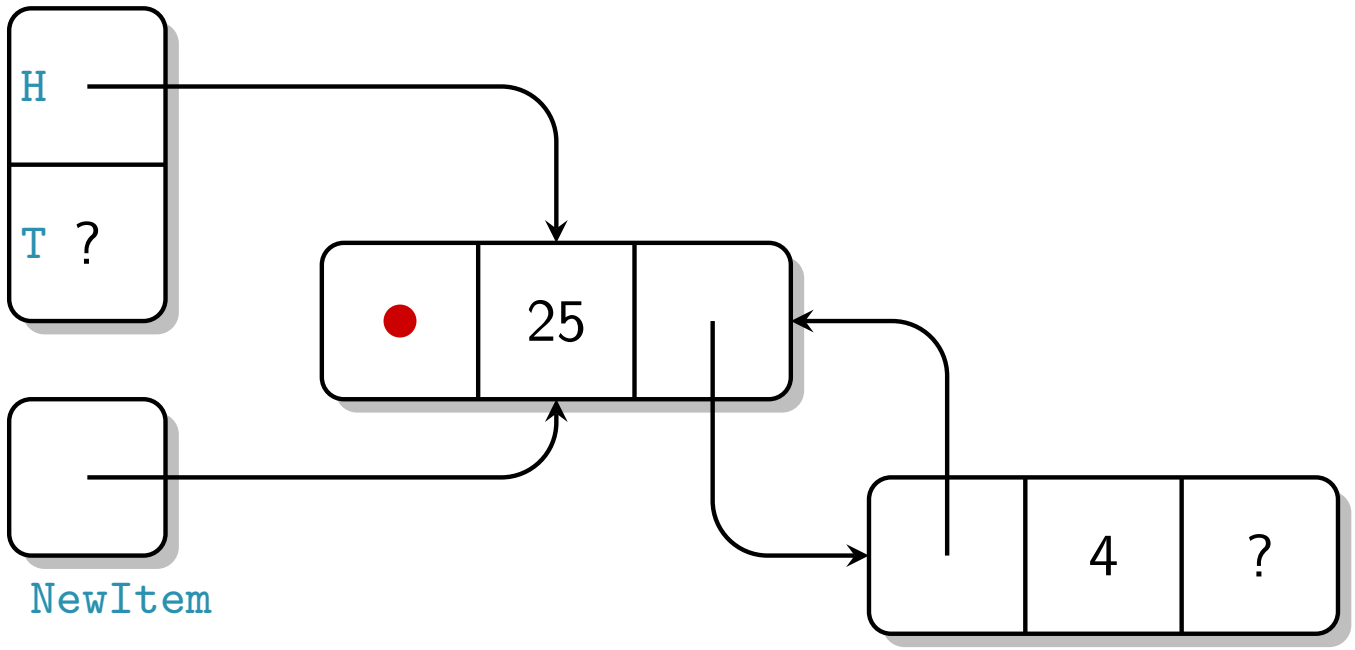
Vložení položky před hlavu seznamu – neprázdný seznam 6

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



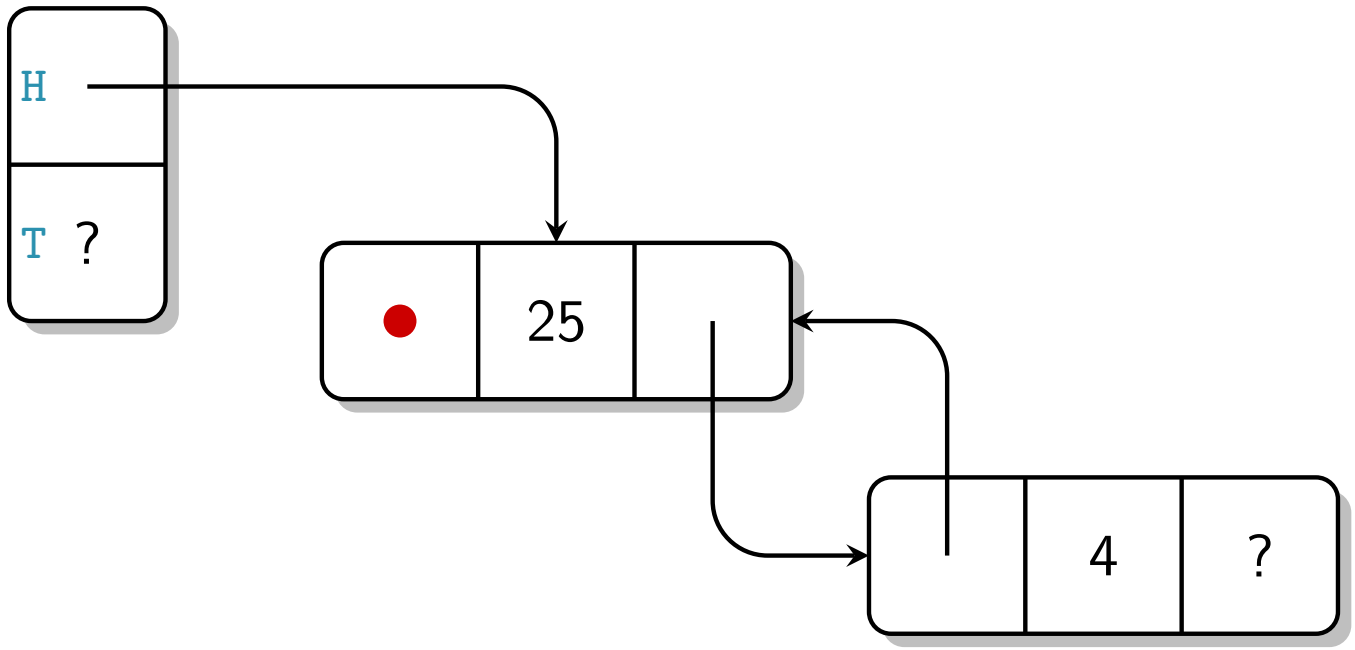
Vložení položky před hlavu seznamu – neprázdný seznam 7

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



Vložení položky před hlavu seznamu – neprázdný seznam 8

```
38 ListItem* NewItem = new ListItem;
39 NewItem->Value = NewValue;
40 NewItem->Prev = nullptr;
41 NewItem->Next = L.Head;
42 L.Head->Prev = NewItem;
43 L.Head = NewItem;
```



Vložení položky do seznamu – na konec seznamu

Zde mohou nastat dvě situace:

1. seznam je prázdný – vytváříme seznam s jediným prvkem a
2. seznam není prázdný.

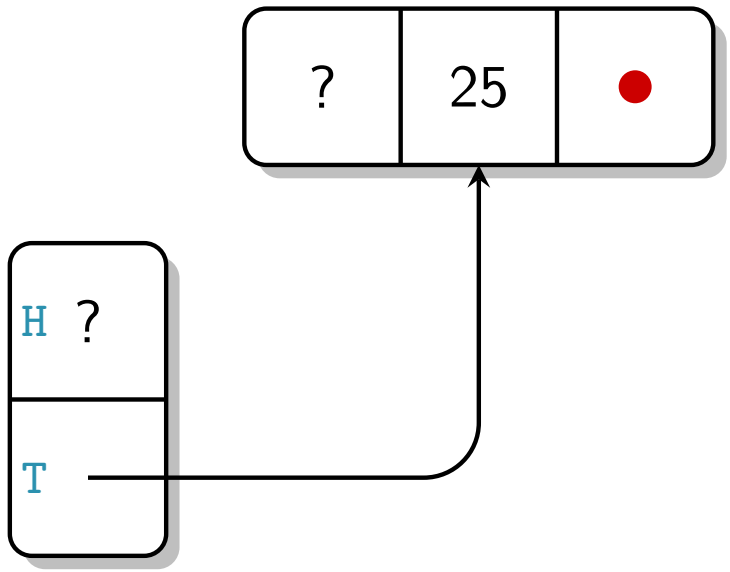
```
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
54    {
55       ListItem* NewItem = new ListItem;
56       NewItem->Value = NewValue;
57       NewItem->Next = nullptr;
58       L.Tail->Next = NewItem;
59       NewItem->Prev = L.Tail;
60       L.Tail = NewItem;
```

Vložení položky do seznamu – na konec seznamu (pokrač.)

```
61      }
62  }
```

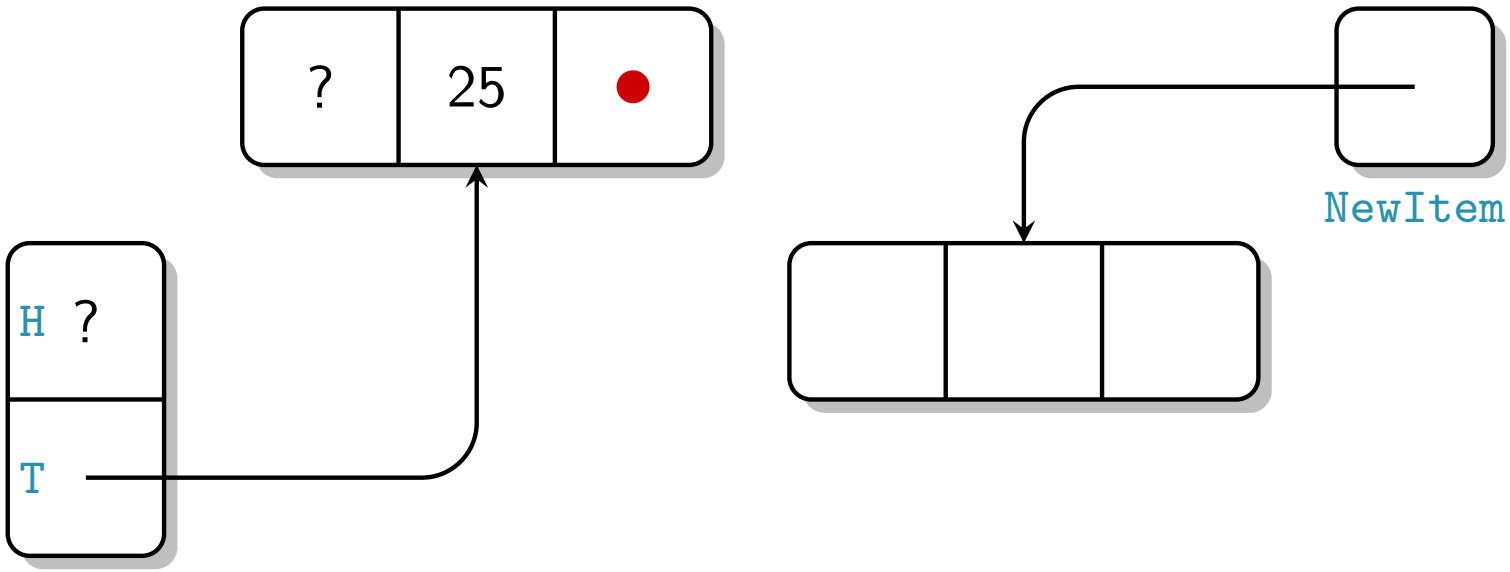
Vložení položky na konec seznamu – neprázdný seznam 1

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



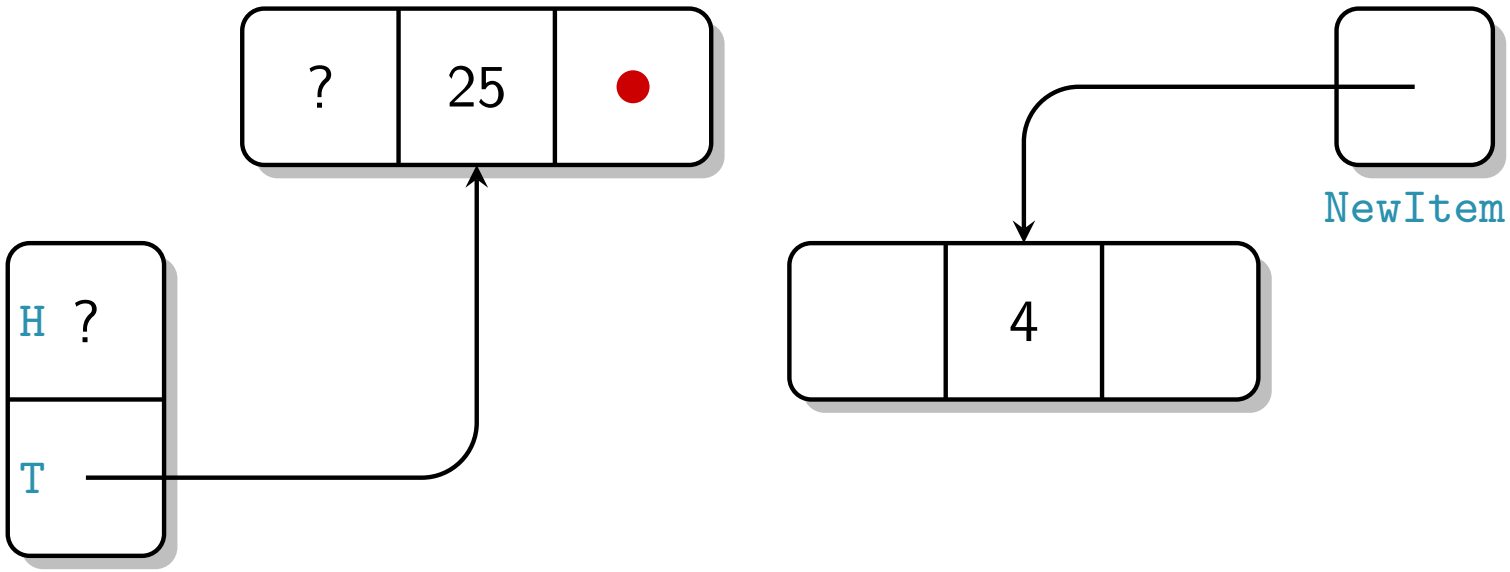
Vložení položky na konec seznamu – neprázdný seznam 2

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



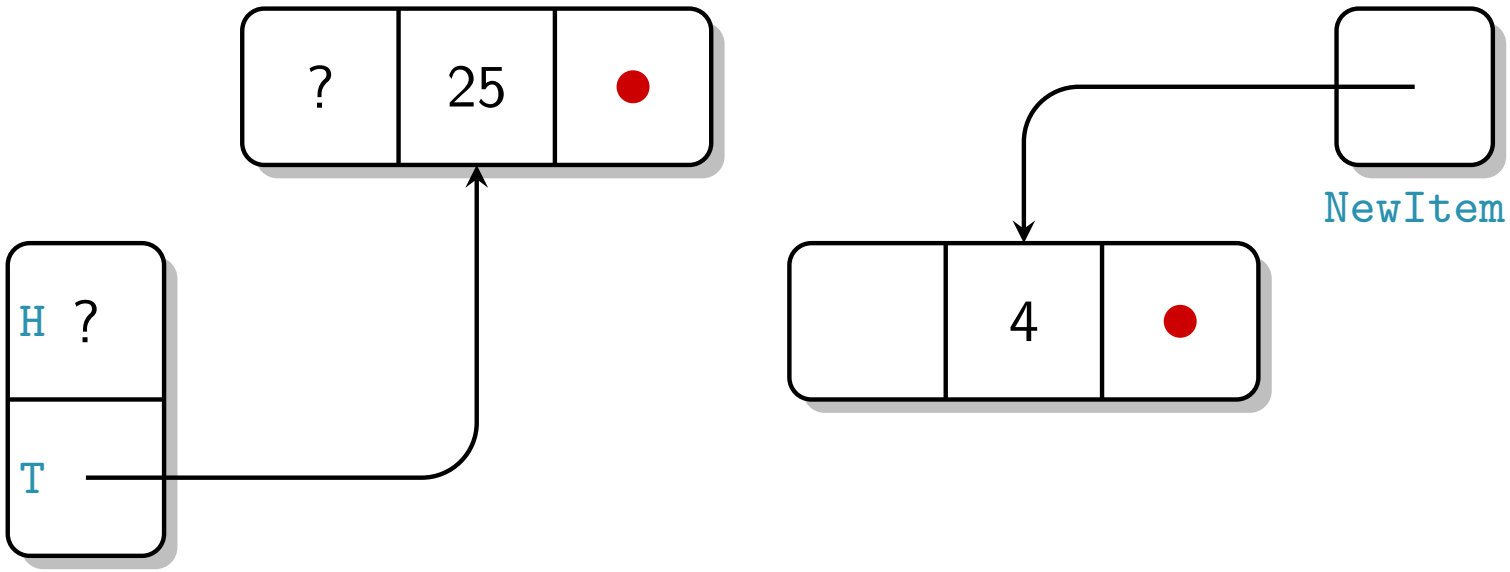
Vložení položky na konec seznamu – neprázdný seznam 3

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



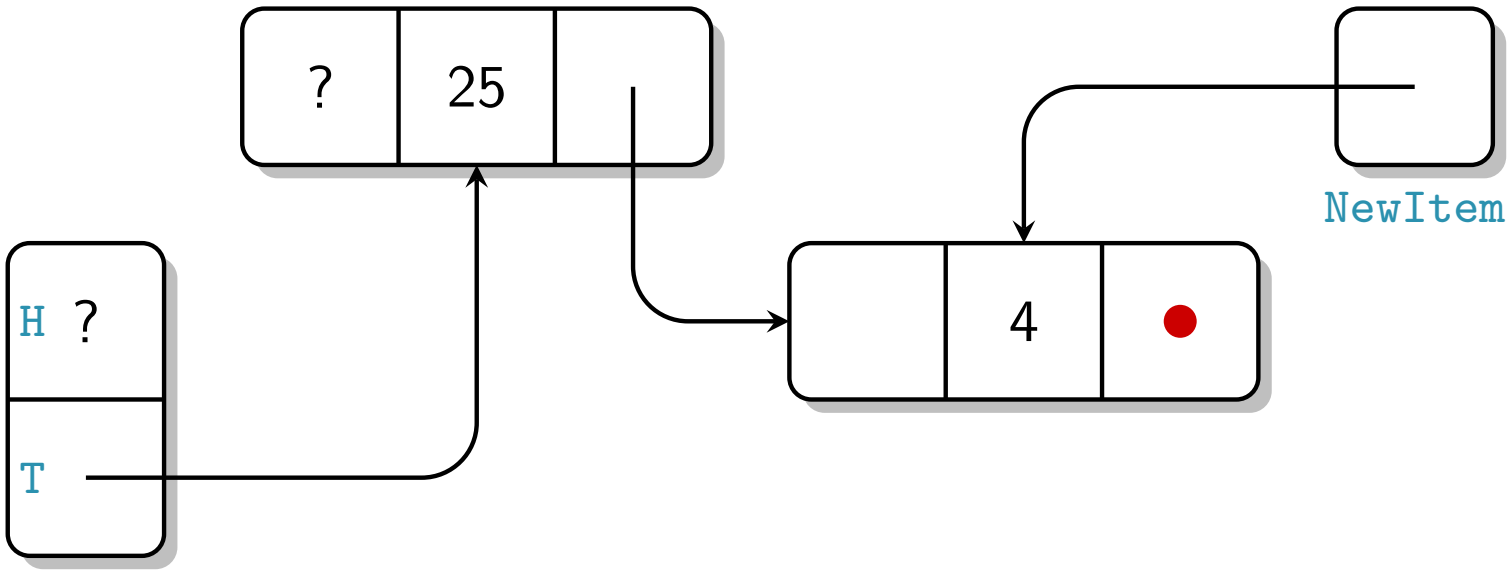
Vložení položky na konec seznamu – neprázdný seznam 4

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



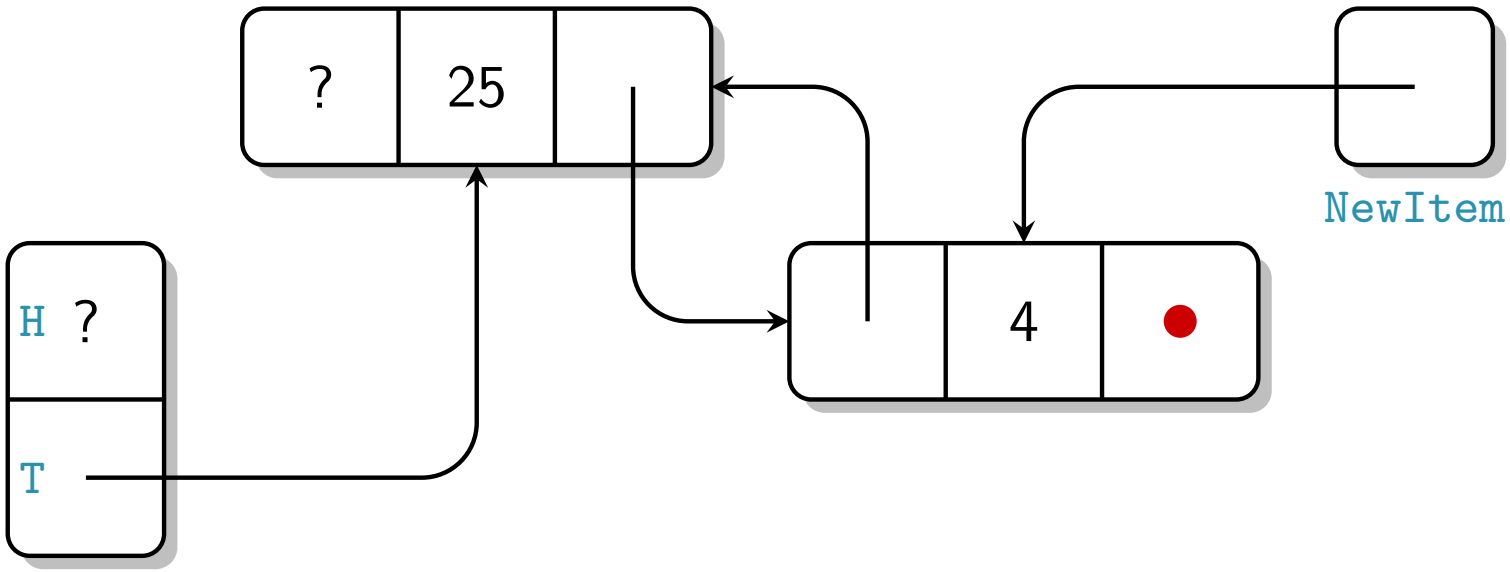
Vložení položky na konec seznamu – neprázdný seznam 5

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



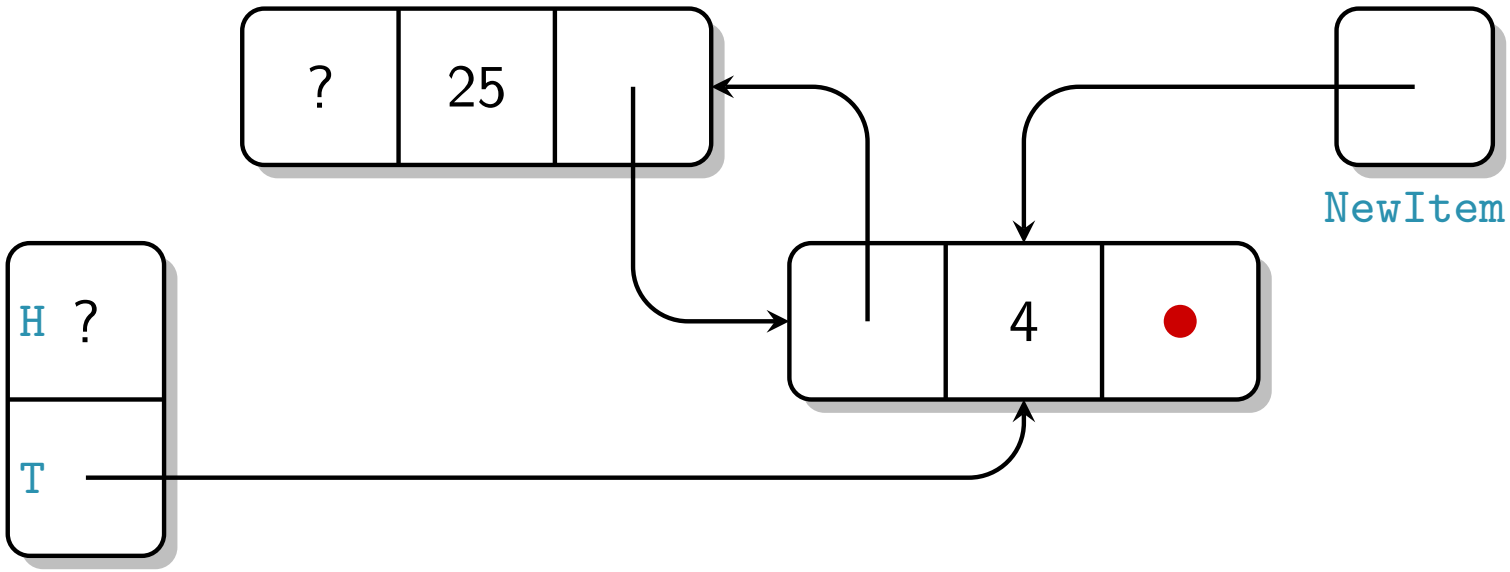
Vložení položky na konec seznamu – neprázdný seznam 6

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



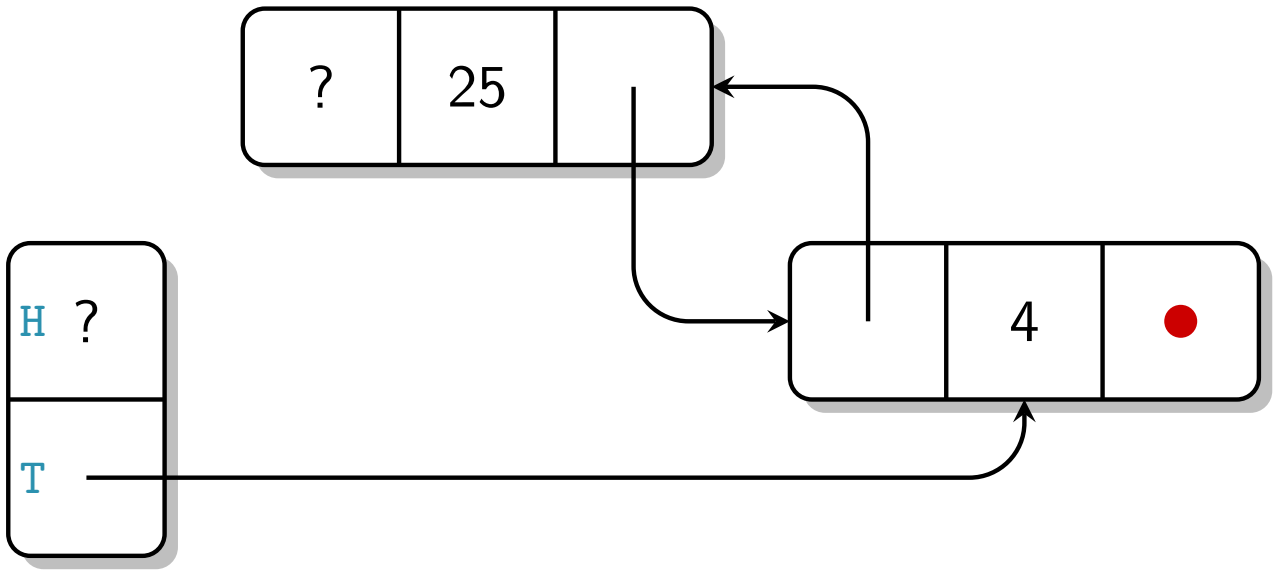
Vložení položky na konec seznamu – neprázdný seznam 7

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



Vložení položky na konec seznamu – neprázdný seznam 8

```
55 ListItem* NewItem = new ListItem;
56 NewItem->Value = NewValue;
57 NewItem->Next = nullptr;
58 L.Tail->Next = NewItem;
59 NewItem->Prev = L.Tail;
60 L.Tail = NewItem;
```



Vložení položky do seznamu – před danou položku

Zde mohou nastat dvě situace:

1. daná položka je hlavou seznamu – využijeme funkci `Prepend` a
2. daná položka je obecně uvnitř seznamu.

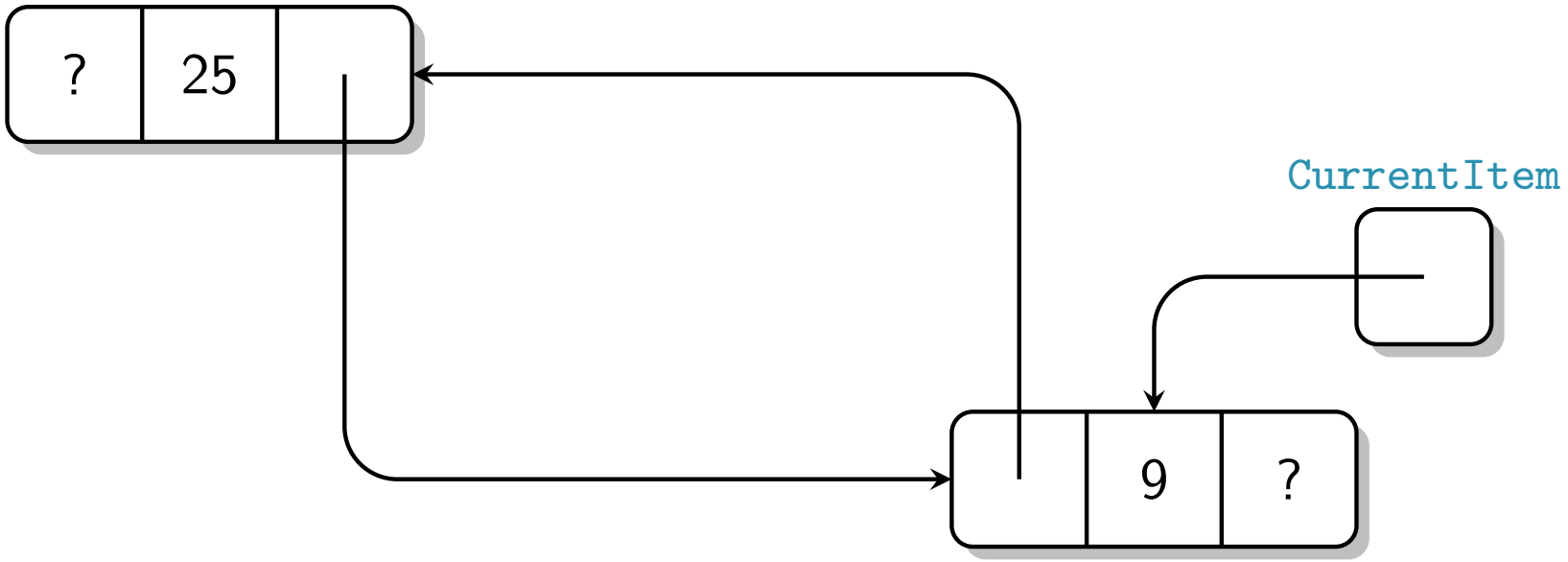
```
64 void InsertBefore(List& L, ListItem* CurrentItem, const string&
       NewValue)
65 {
66    if (CurrentItem != nullptr)
67    {
68       if (CurrentItem == L.Head)
69       {
70          Prepend(L, NewValue);
71       }
72       else
73       {
74          ListItem* NewItem = new ListItem;
75          NewItem->Value = NewValue;
76          ListItem* P = CurrentItem->Prev;
```

Vložení položky do seznamu – před danou položku (pokrač.)

```
77            P->Next = NewItem;
78            NewItem->Prev = P;
79            CurrentItem->Prev = NewItem;
80            NewItem->Next = CurrentItem;
81        }
82    }
83 }
```

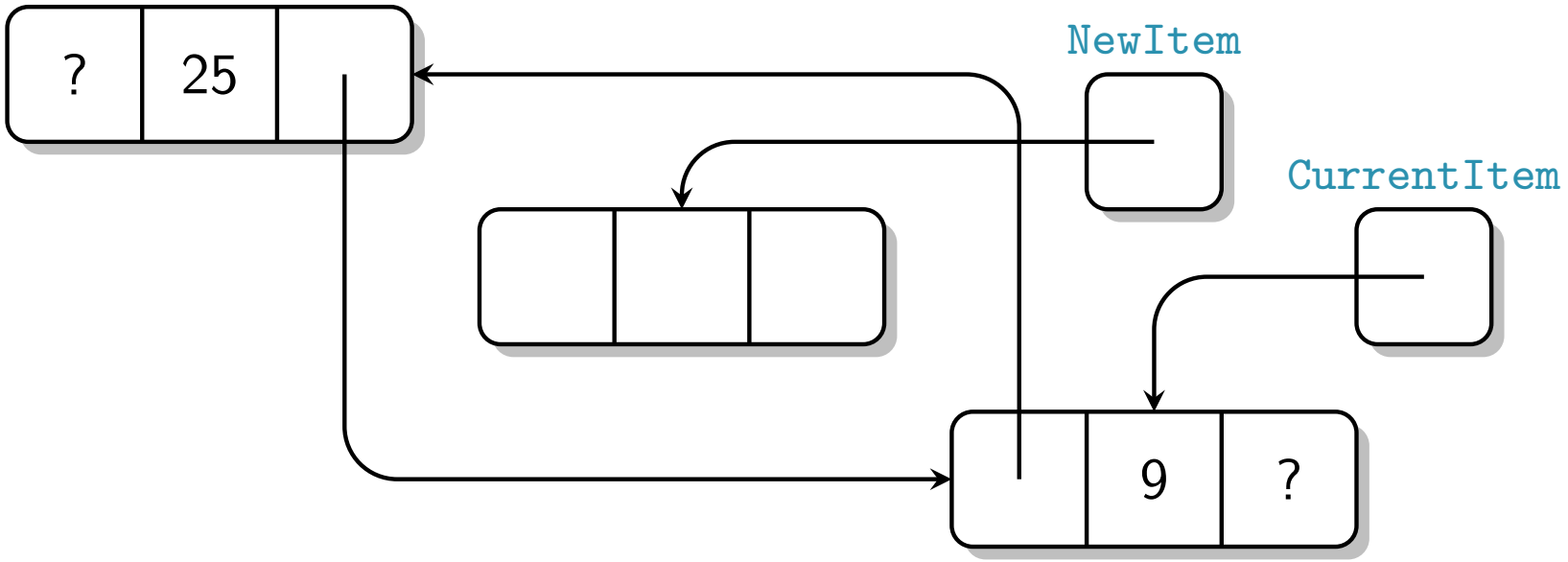
Vložení položky do seznamu před danou položku – obecně 1

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



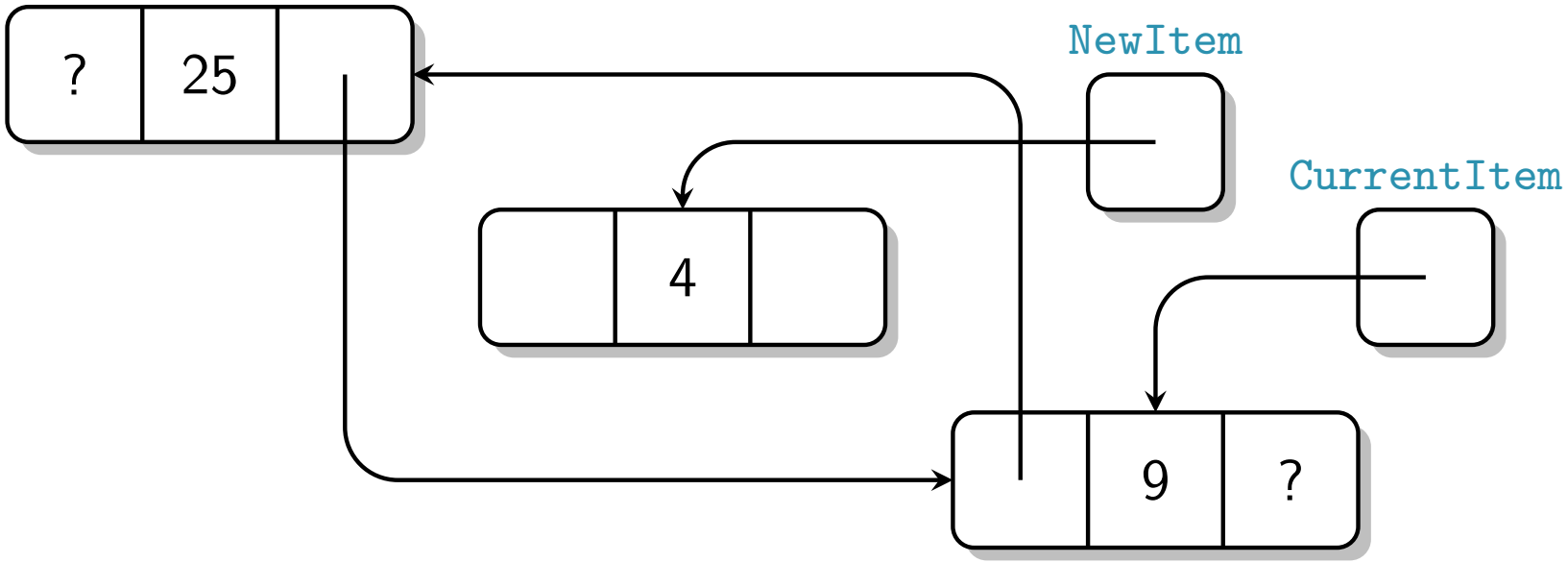
Vložení položky do seznamu před danou položku – obecně 2

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



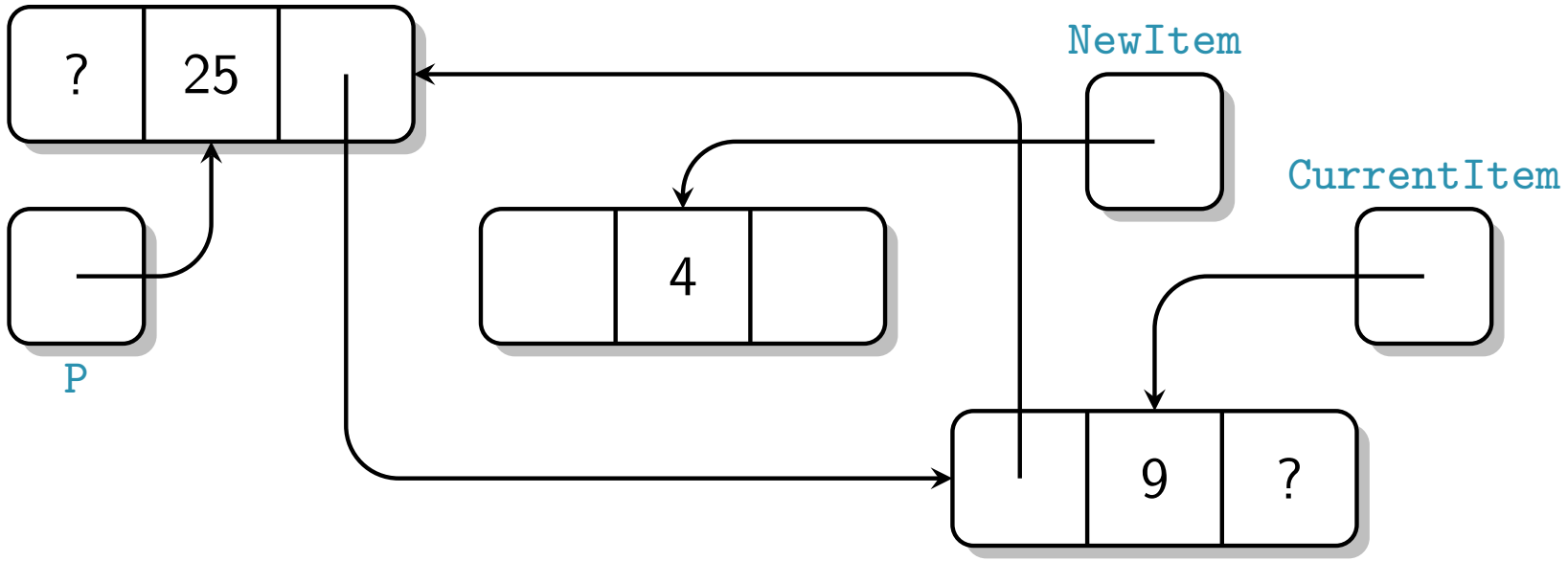
Vložení položky do seznamu před danou položku – obecně 3

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



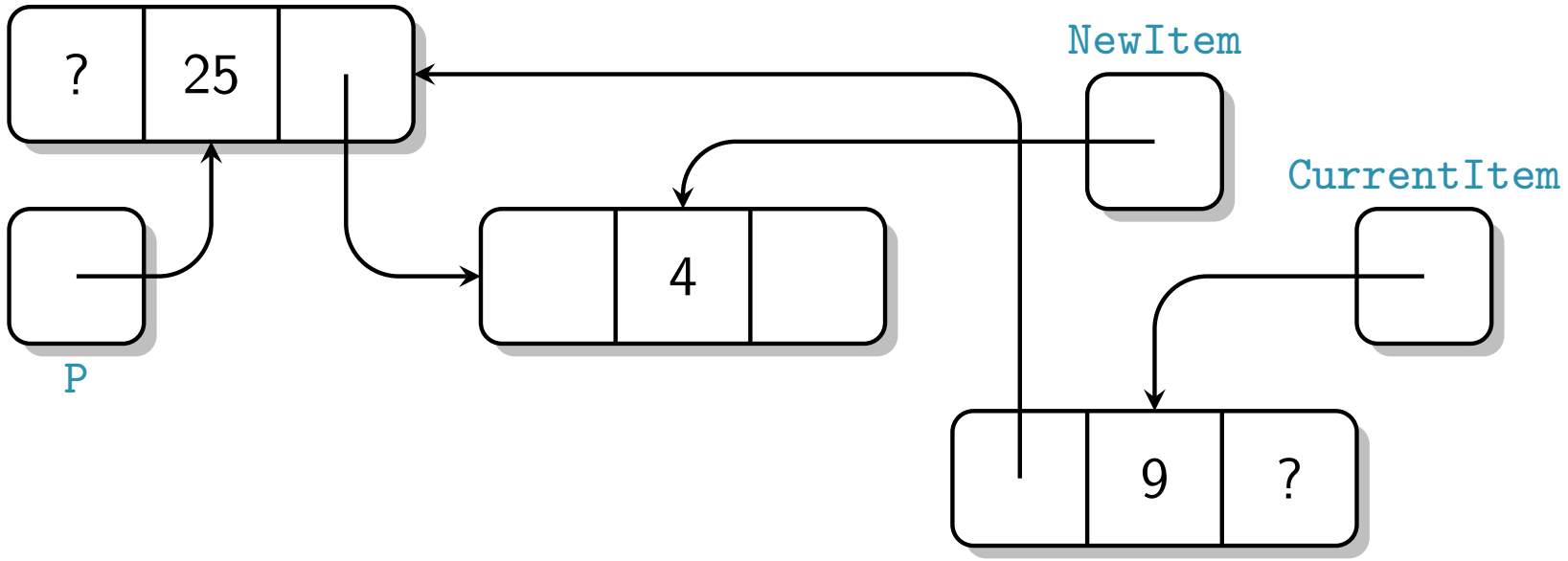
Vložení položky do seznamu před danou položku – obecně 4

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



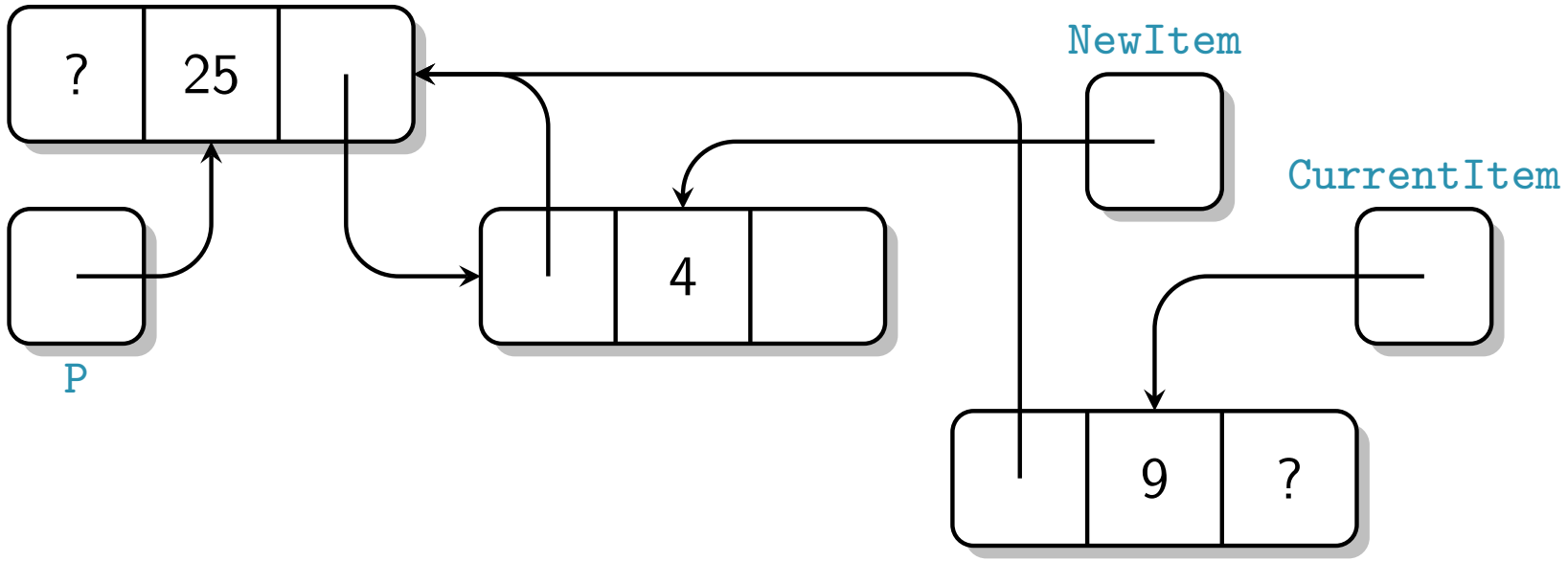
Vložení položky do seznamu před danou položku – obecně 5

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



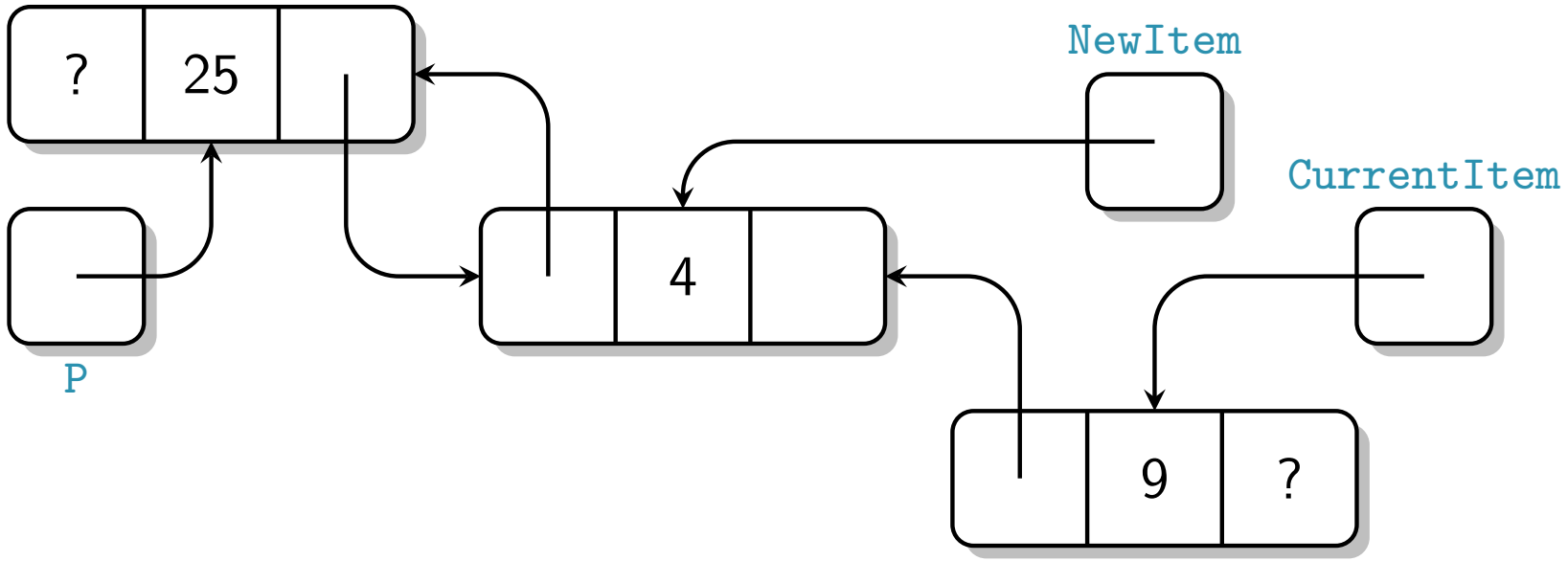
Vložení položky do seznamu před danou položku – obecně 6

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



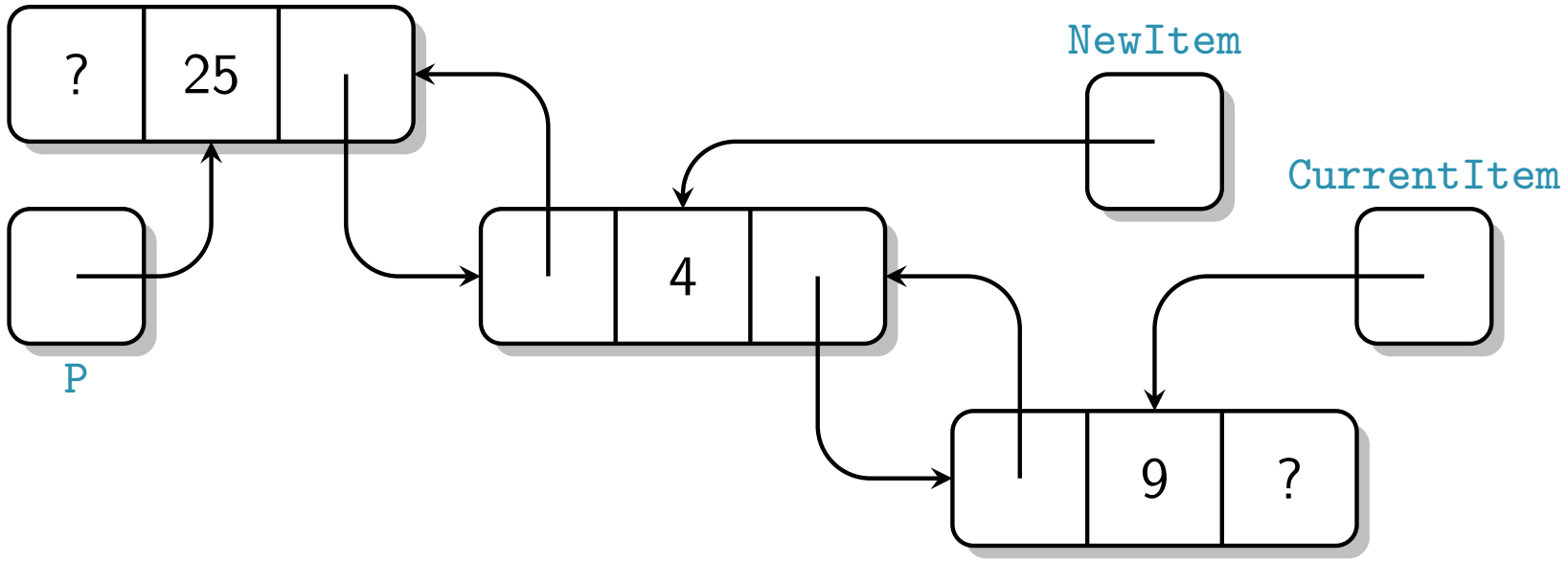
Vložení položky do seznamu před danou položku – obecně 7

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



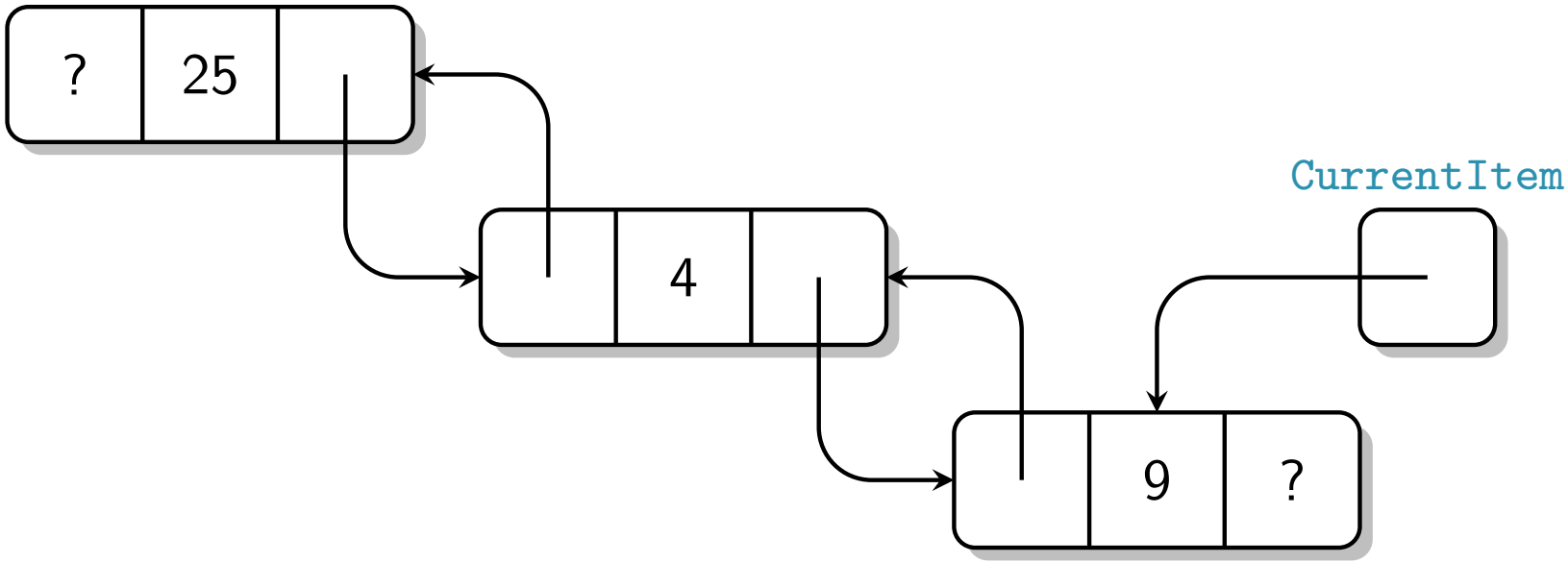
Vložení položky do seznamu před danou položku – obecně 8

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



Vložení položky do seznamu před danou položku – obecně 9

```
74 ListItem* NewItem = new ListItem;
75 NewItem->Value = NewValue;
76 ListItem* P = CurrentItem->Prev;
77 P->Next = NewItem;
78 NewItem->Prev = P;
79 CurrentItem->Prev = NewItem;
80 NewItem->Next = CurrentItem;
```



Vložení položky do seznamu – za danou položku

Zde mohou nastat dvě situace:

1. daná položka je ocasem seznamu – využijeme funkci `Append` a
2. daná položka je obecně uvnitř seznamu.

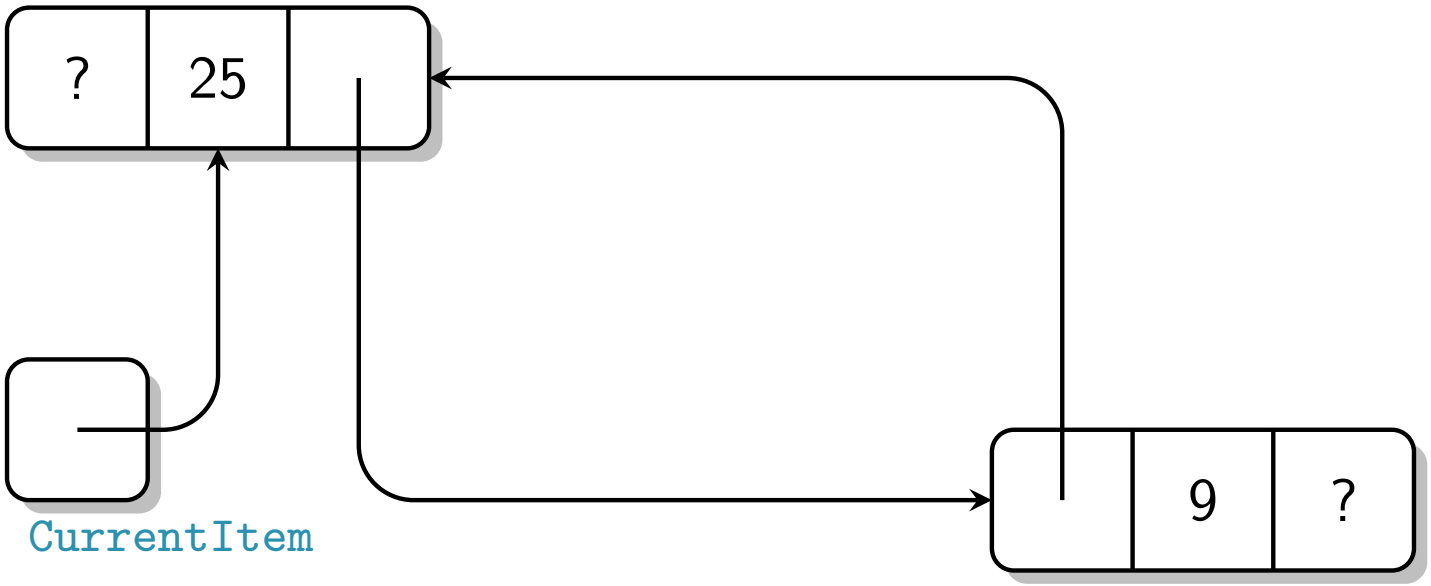
```
85 void InsertAfter(List& L, ListItem* CurrentItem, const string& NewValue
       )
86 {
87    if (CurrentItem != nullptr)
88    {
89       if (CurrentItem == L.Tail)
90       {
91          Append(L, NewValue);
92       }
93       else
94       {
95          ListItem* NewItem = new ListItem;
96          NewItem->Value = NewValue;
97          ListItem* N = CurrentItem->Next;
```

Vložení položky do seznamu – za danou položku (pokrač.)

```
98              N->Prev = NewItem;
99              NewItem->Next = N;
100             CurrentItem->Next = NewItem;
101             NewItem->Prev = CurrentItem;
102         }
103     }
104 }
```

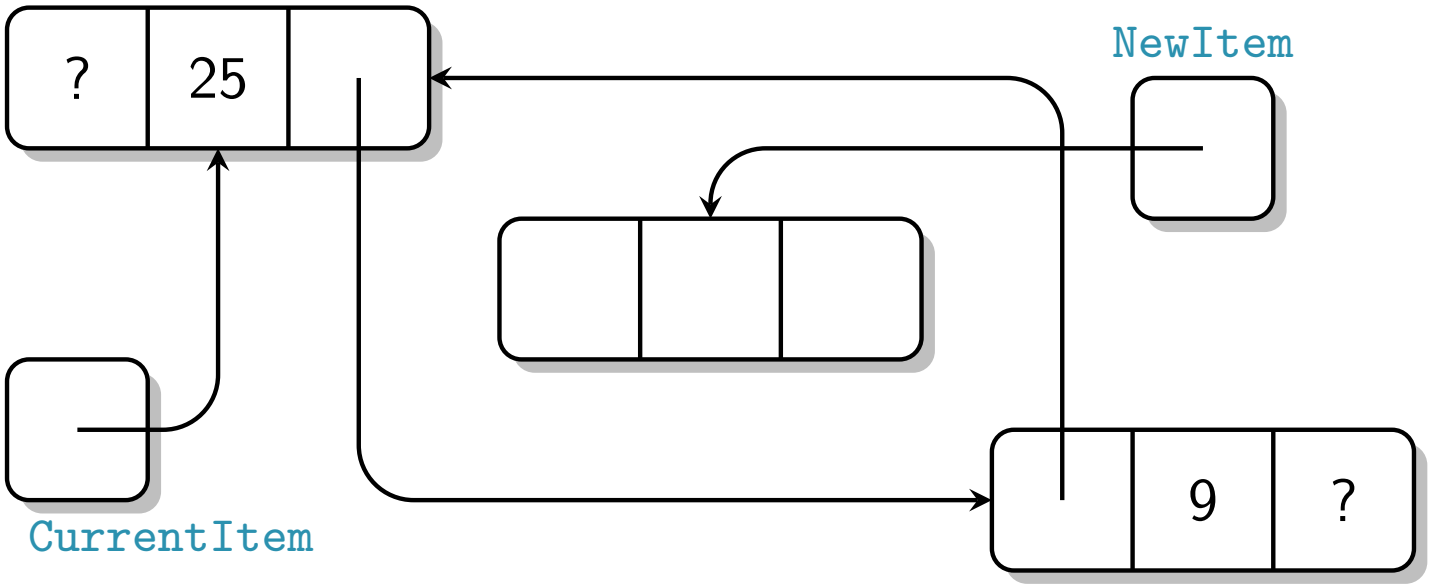
Vložení položky do seznamu za danou položku – obecně 1

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



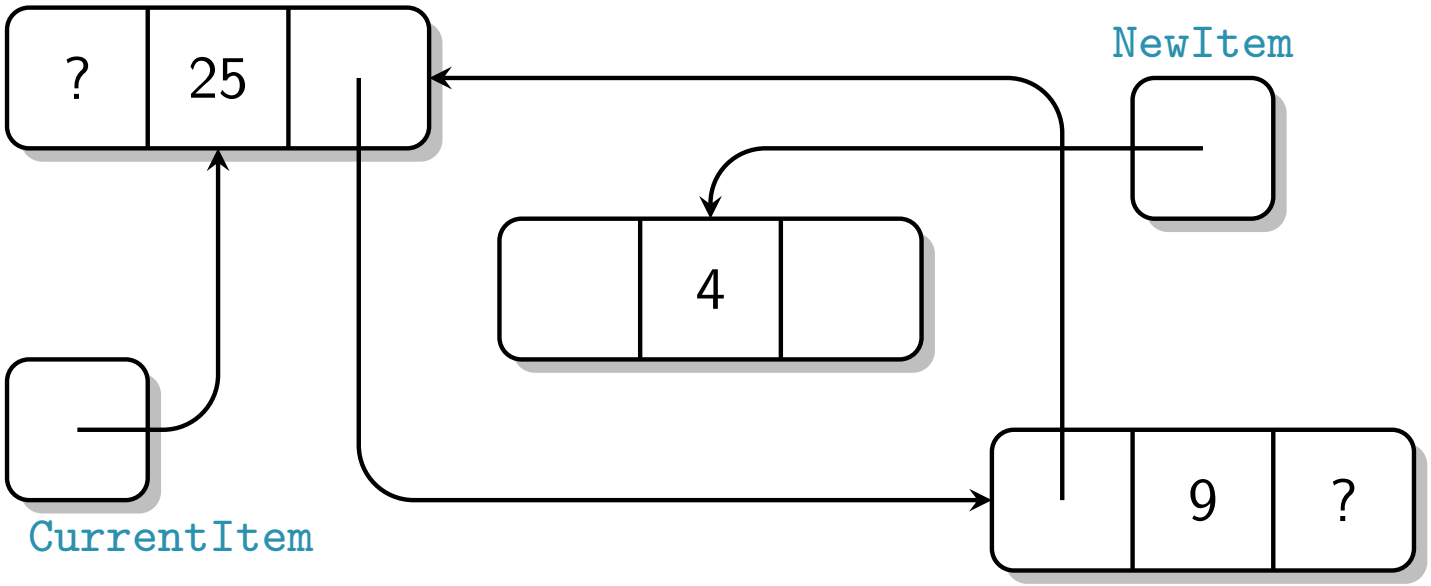
Vložení položky do seznamu za danou položku – obecně 2

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



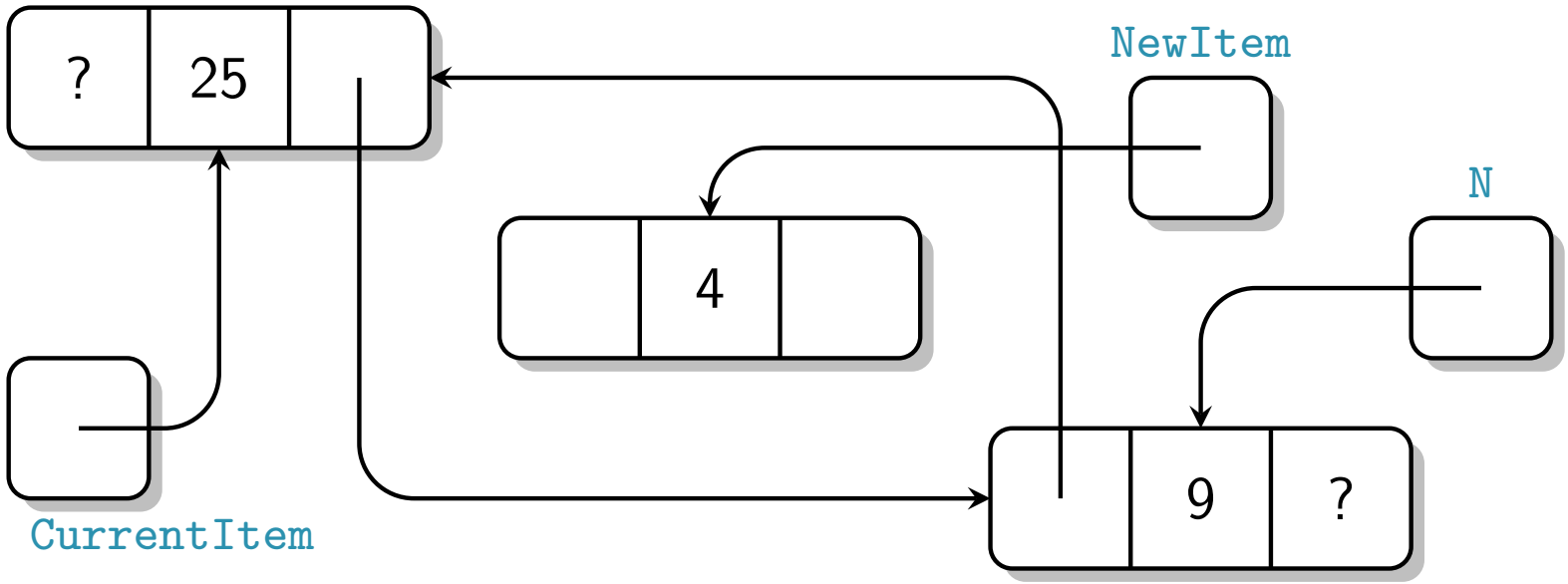
Vložení položky do seznamu za danou položku – obecně 3

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



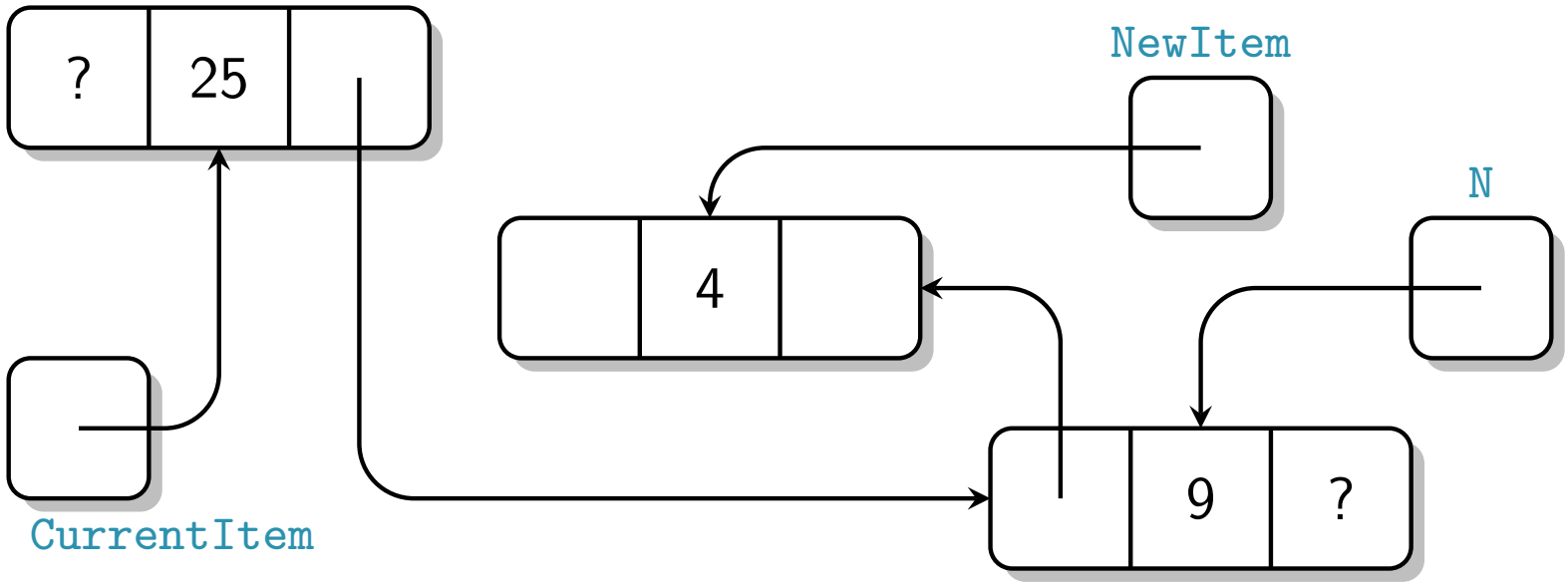
Vložení položky do seznamu za danou položku – obecně 4

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



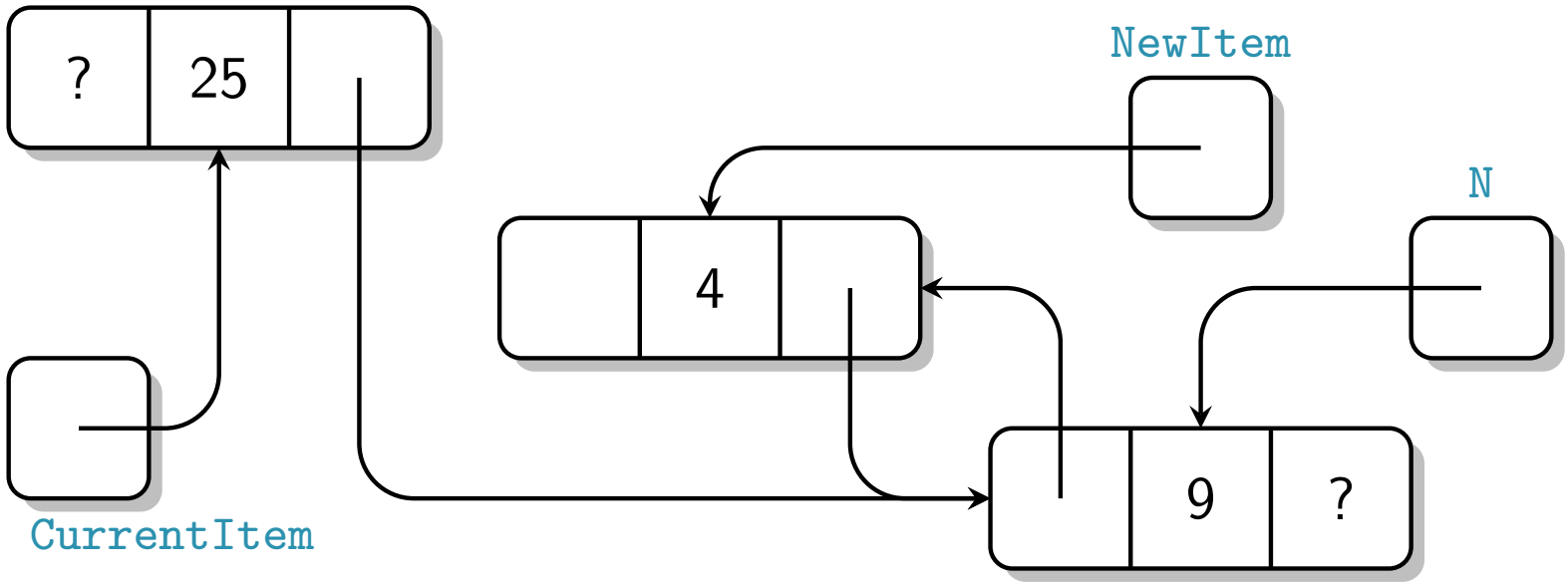
Vložení položky do seznamu za danou položku – obecně 5

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



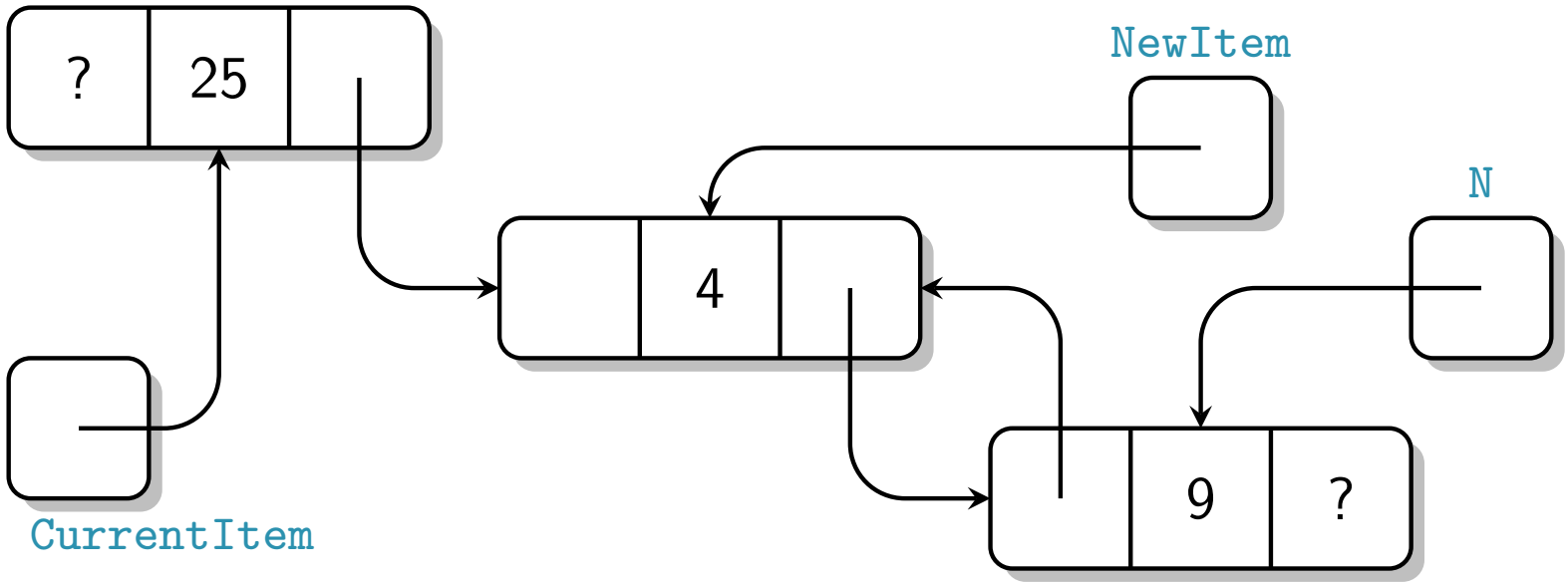
Vložení položky do seznamu za danou položku – obecně 6

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



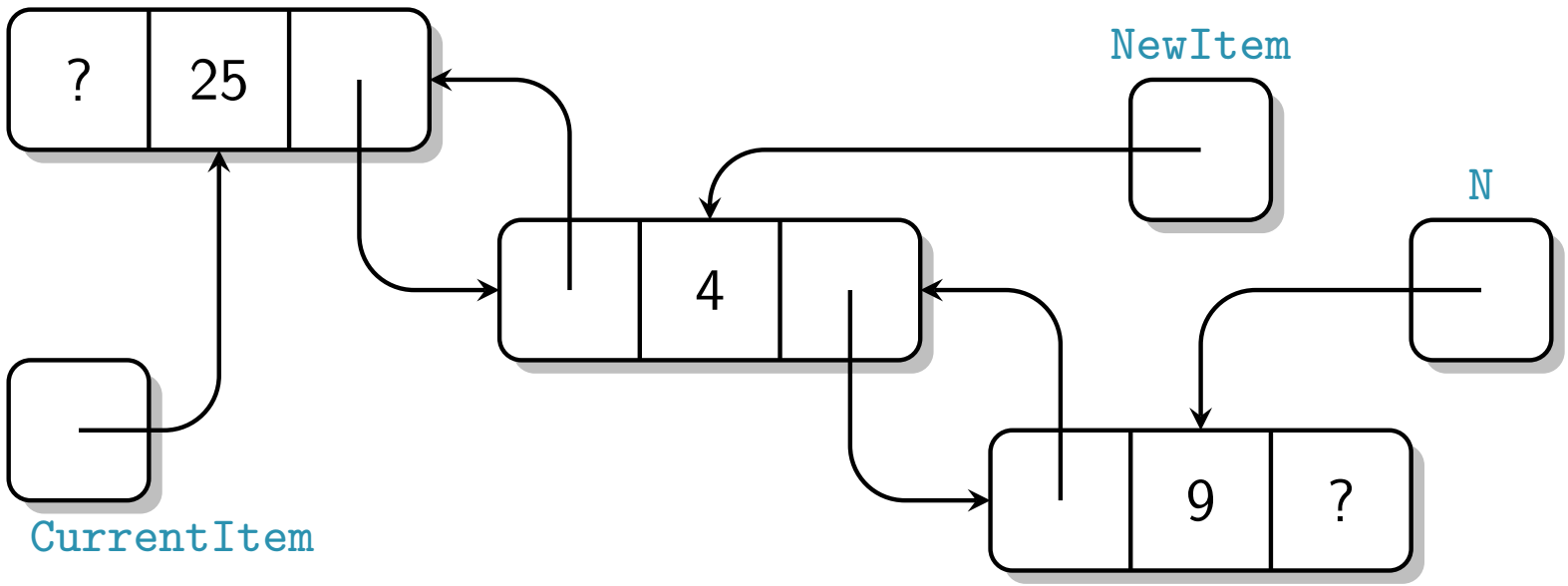
Vložení položky do seznamu za danou položku – obecně 7

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



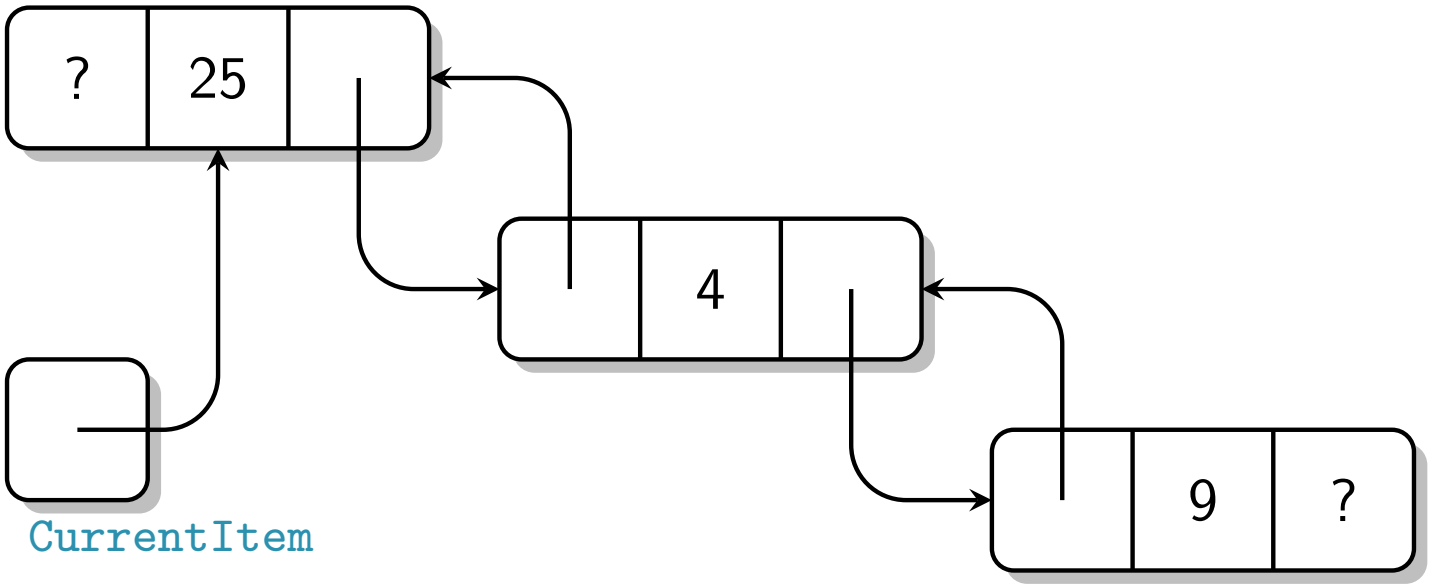
Vložení položky do seznamu za danou položku – obecně 8

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



Vložení položky do seznamu za danou položku – obecně 9

```
95 ListItem* NewItem = new ListItem;
96 NewItem->Value = NewValue;
97 ListItem* N = CurrentItem->Next;
98 N->Prev = NewItem;
99 NewItem->Next = N;
100 CurrentItem->Next = NewItem;
101 NewItem->Prev = CurrentItem;
```



Odebrání položky ze seznamu

Odebrání položky ze seznamu můžeme rozdělit na dvě fáze:

1. vlastní odebrání položky ze seznamu – odstranění vazeb mezi ostatními položkami seznamu a odebíranou položkou. Zde mohou nastat celkem čtyři situace:
   - 1.1 seznam obsahuje jedinou položku – řádky 242 až 246,
   - 1.2 odebíraná položka je hlavou seznamu – řádky 249 až 253,
   - 1.3 odebíraná položka je ocasem seznamu – řádky 256 až 260 nebo
   - 1.4 odebíraná položka leží obecně uvnitř seznamu – řádky 263 až 266.
2. uvolnění položky z paměti – řádek 270

Odebrání položky ze seznamu (pokrač.)

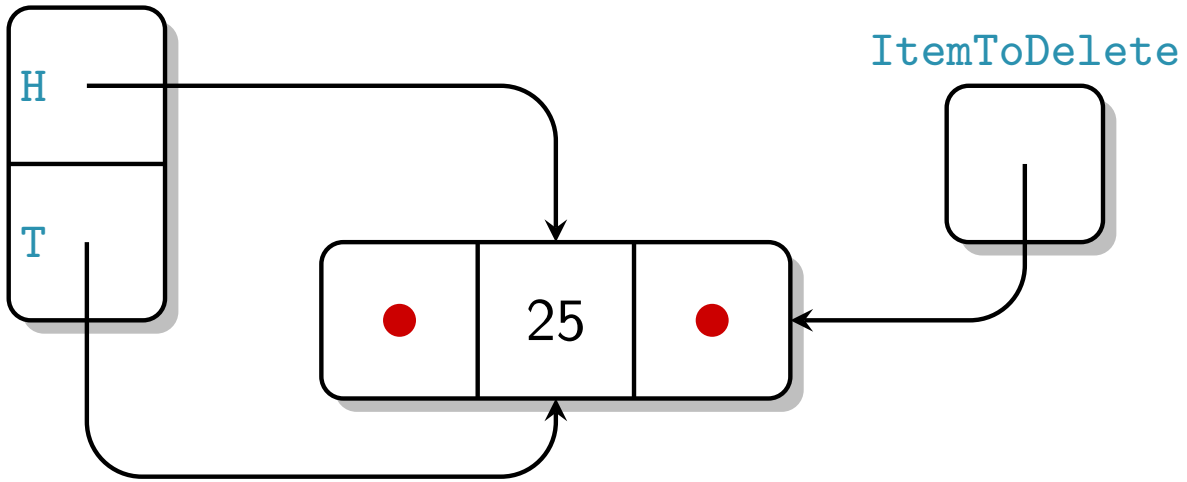
```
240 void InternalRemove(List& L, const ListItem* ItemToDelete)
241 {
242    if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243    {
244       L.Head = nullptr;
245       L.Tail = nullptr;
246    }
247    else
248    {
249       if (ItemToDelete == L.Head)
250       {
251          L.Head = L.Head->Next;
252          L.Head->Prev = nullptr;
253       }
254       else
255       {
256          if (ItemToDelete == L.Tail)
257          {
258             L.Tail = L.Tail->Prev;
```

Odebrání položky ze seznamu (pokrač.)

```
259             L.Tail->Next = nullptr;
260          }
261          else
262          {
263             ListItem* P = ItemToDelete->Prev;
264             ListItem* N = ItemToDelete->Next;
265             P->Next = N;
266             N->Prev = P;
267          }
268       }
269    }
270    delete ItemToDelete;
271 }
```

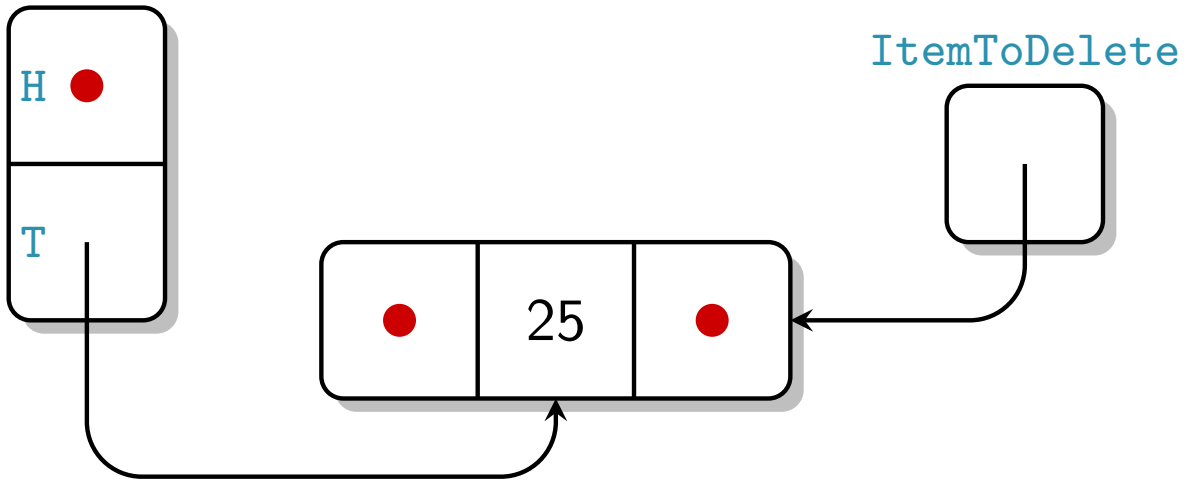
Odebrání položky ze seznamu – jediná položka 1

```
242 if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243 {
244     L.Head = nullptr;
245     L.Tail = nullptr;
246 }
```



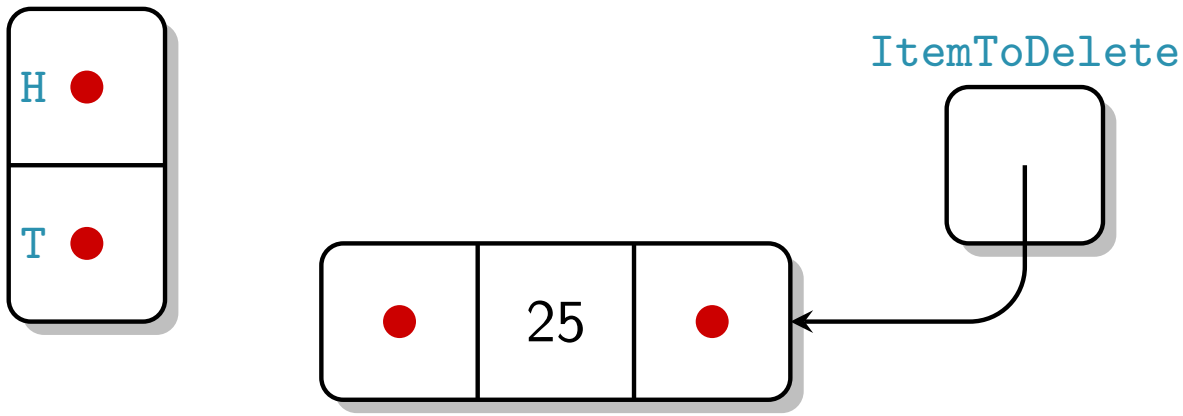
Odebrání položky ze seznamu – jediná položka 2

```
242 if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243 {
244     L.Head = nullptr;
245     L.Tail = nullptr;
246 }
```



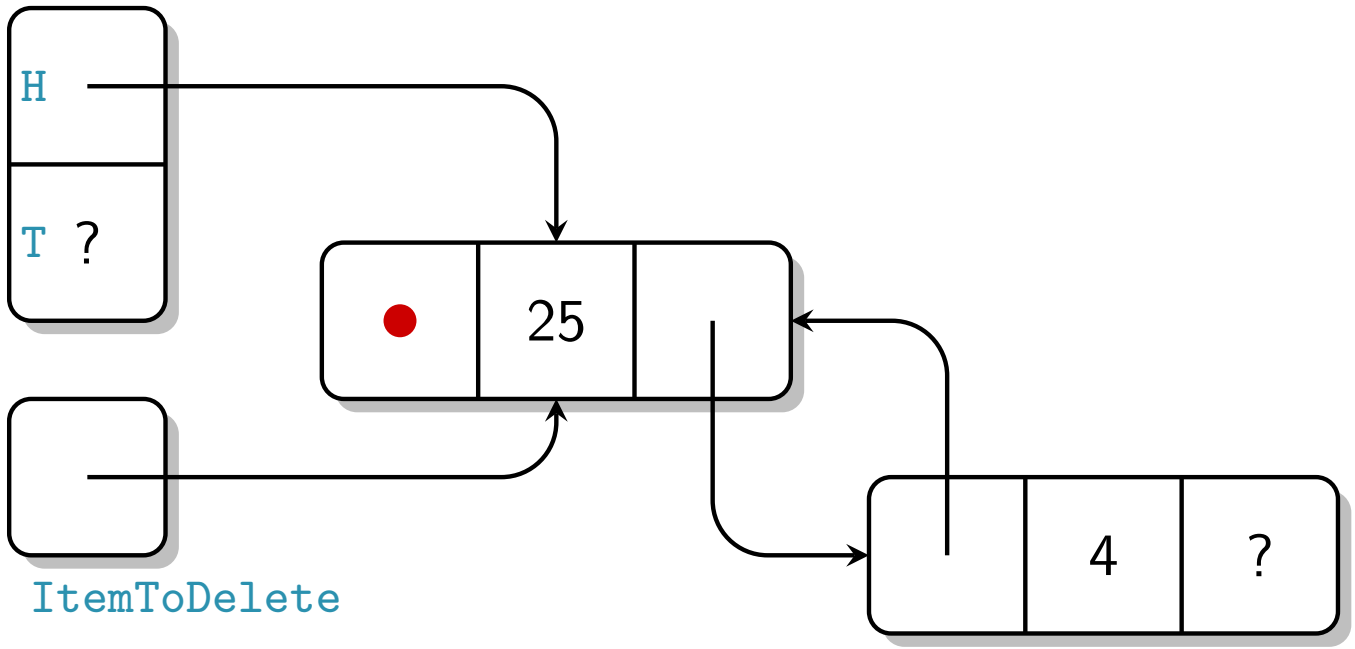
Odebrání položky ze seznamu – jediná položka 3

```
242 if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243 {
244     L.Head = nullptr;
245     L.Tail = nullptr;
246 }
```



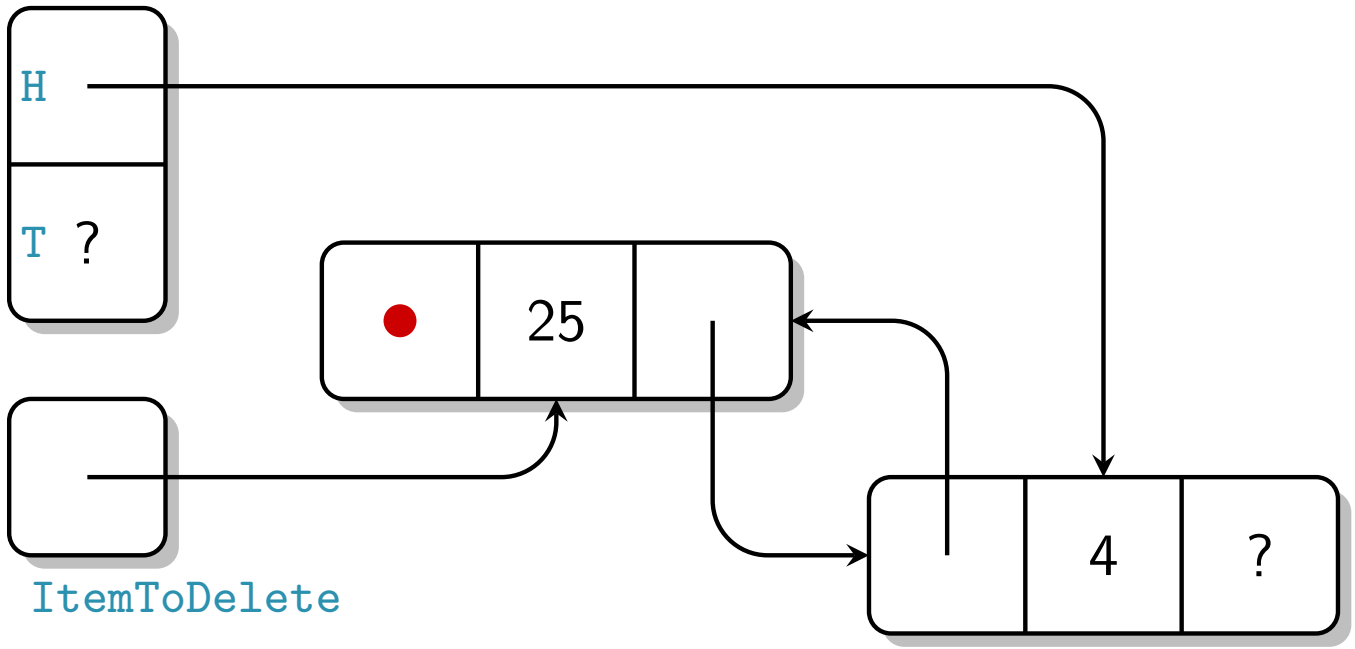
Odebrání položky ze seznamu – hlava 1

```
249 if (ItemToDelete == L.Head)
250 {
251     L.Head = L.Head->Next;
252     L.Head->Prev = nullptr;
253 }
```



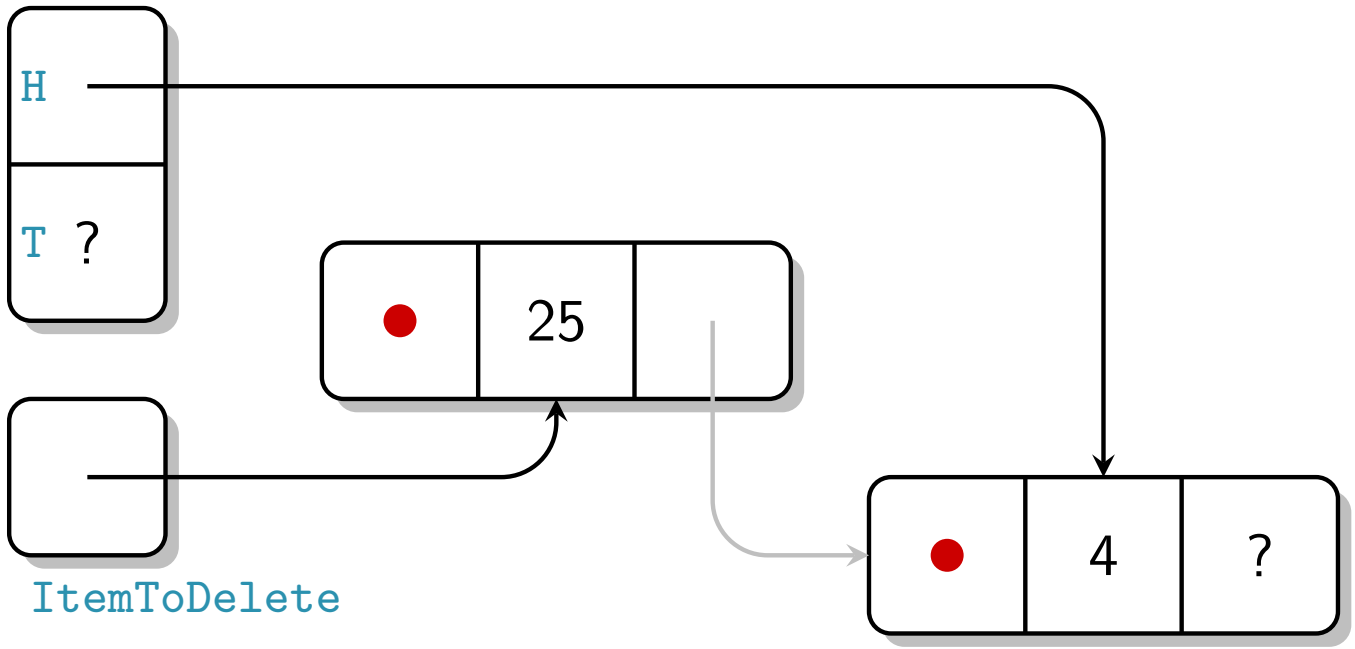
Odebrání položky ze seznamu – hlava 2

```
249 if (ItemToDelete == L.Head)
250 {
251     L.Head = L.Head->Next;
252     L.Head->Prev = nullptr;
253 }
```



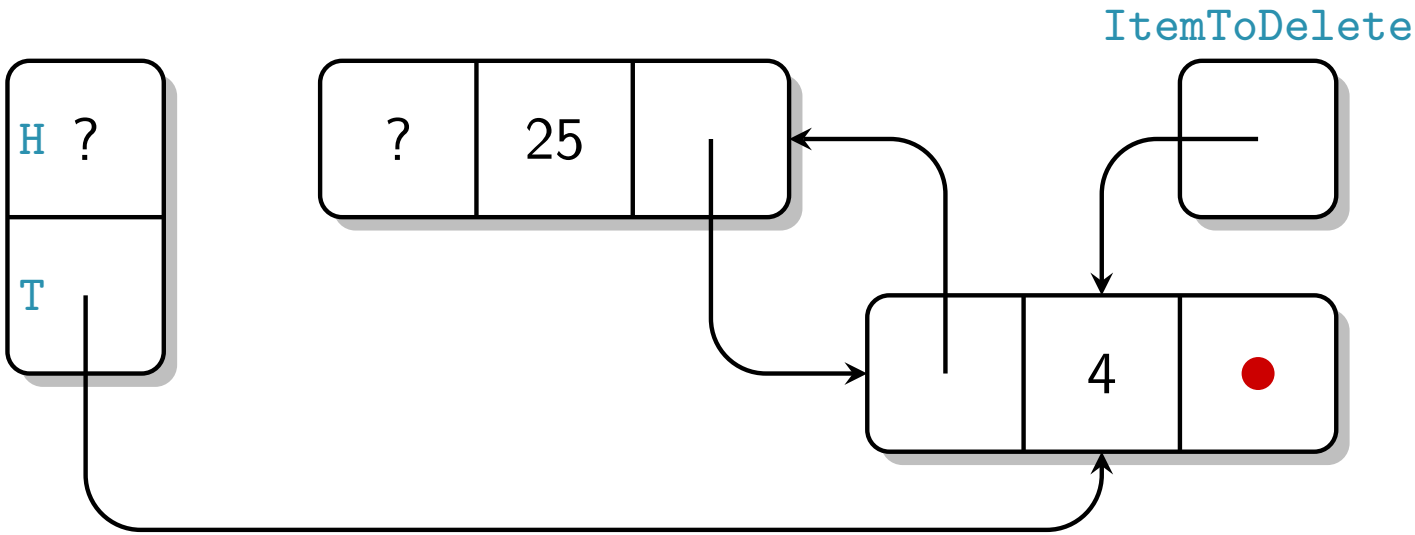
Odebrání položky ze seznamu – hlava 3

```
249 if (ItemToDelete == L.Head)
250 {
251     L.Head = L.Head->Next;
252     L.Head->Prev = nullptr;
253 }
```



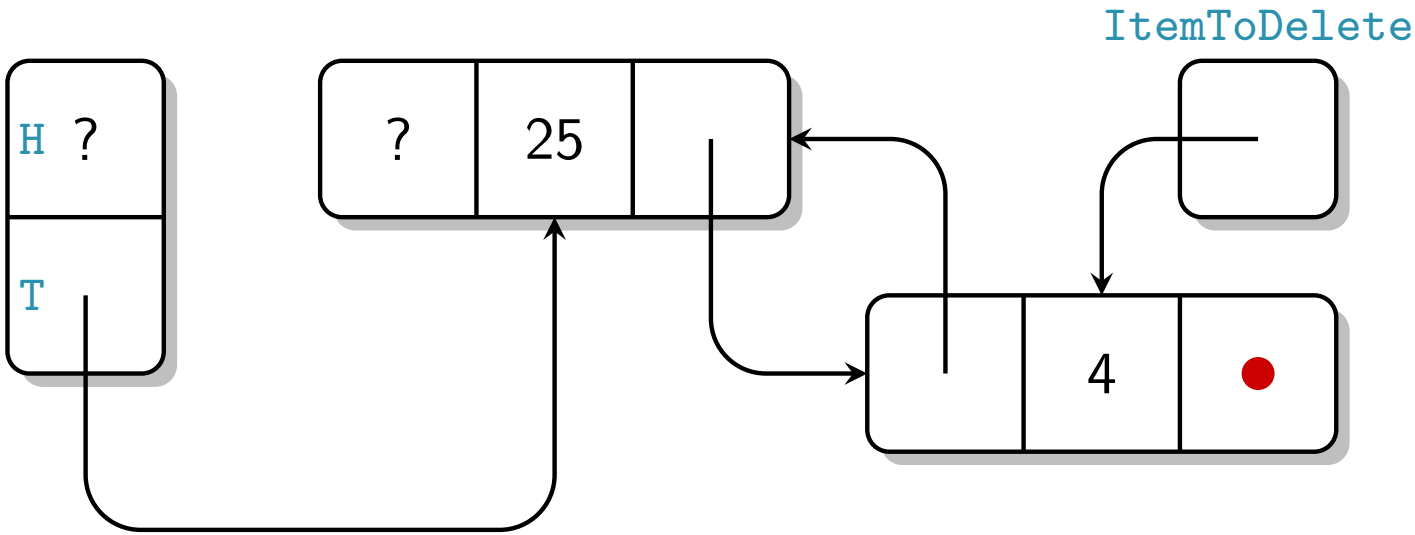
Odebrání položky ze seznamu – ocas 1

```
256 if (ItemToDelete == L.Tail)
257 {
258    L.Tail = L.Tail->Prev;
259    L.Tail->Next = nullptr;
260 }
```



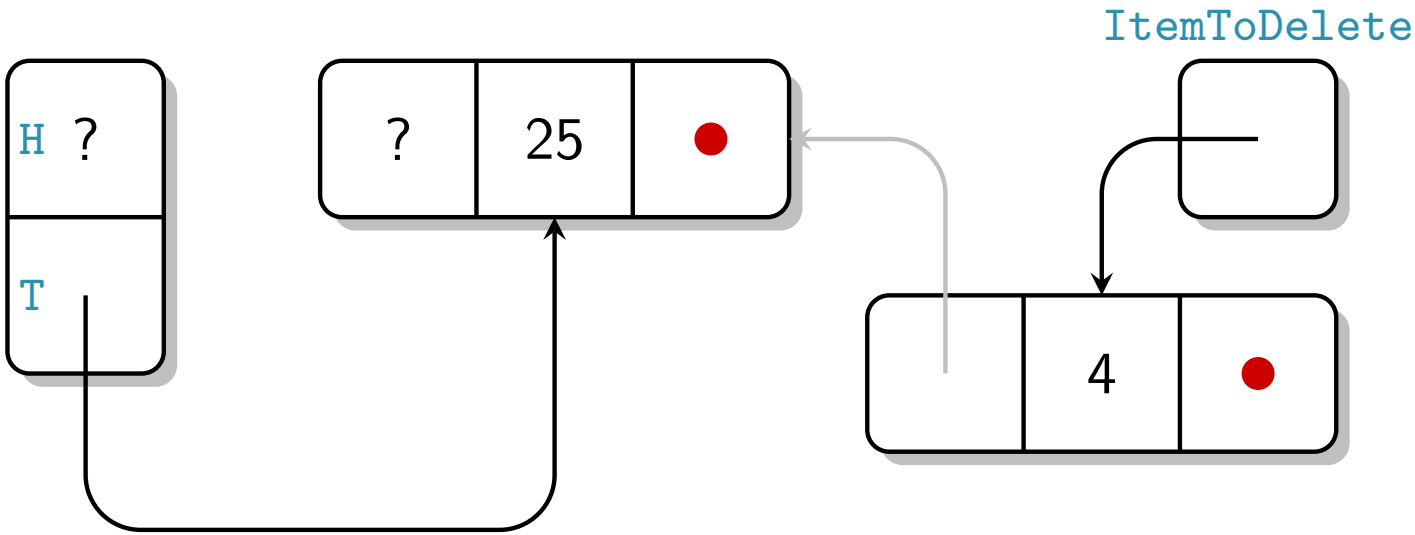
Odebrání položky ze seznamu – ocas 2

```
256 if (ItemToDelete == L.Tail)
257 {
258    L.Tail = L.Tail->Prev;
259    L.Tail->Next = nullptr;
260 }
```



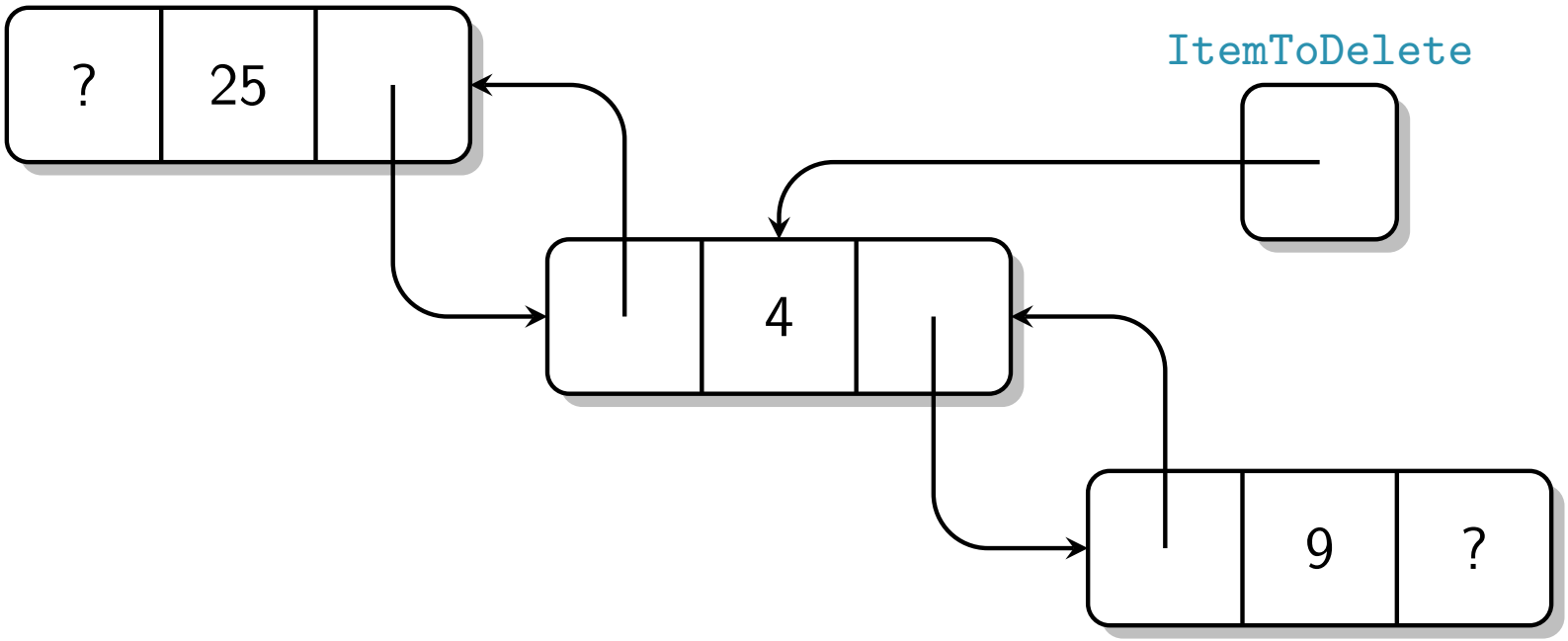
Odebrání položky ze seznamu – ocas 3

```
256 if (ItemToDelete == L.Tail)
257 {
258     L.Tail = L.Tail->Prev;
259     L.Tail->Next = nullptr;
260 }
```



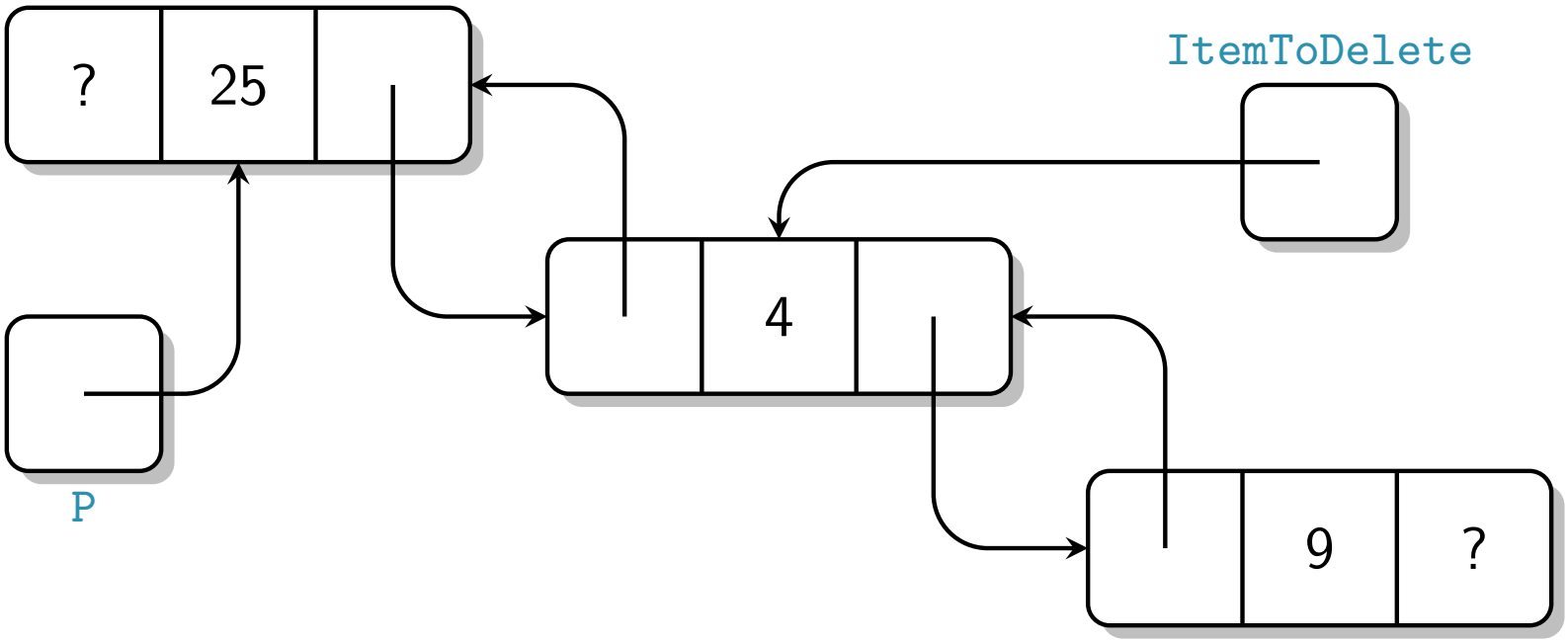
Odebrání položky ze seznamu – obecná pozice 1

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



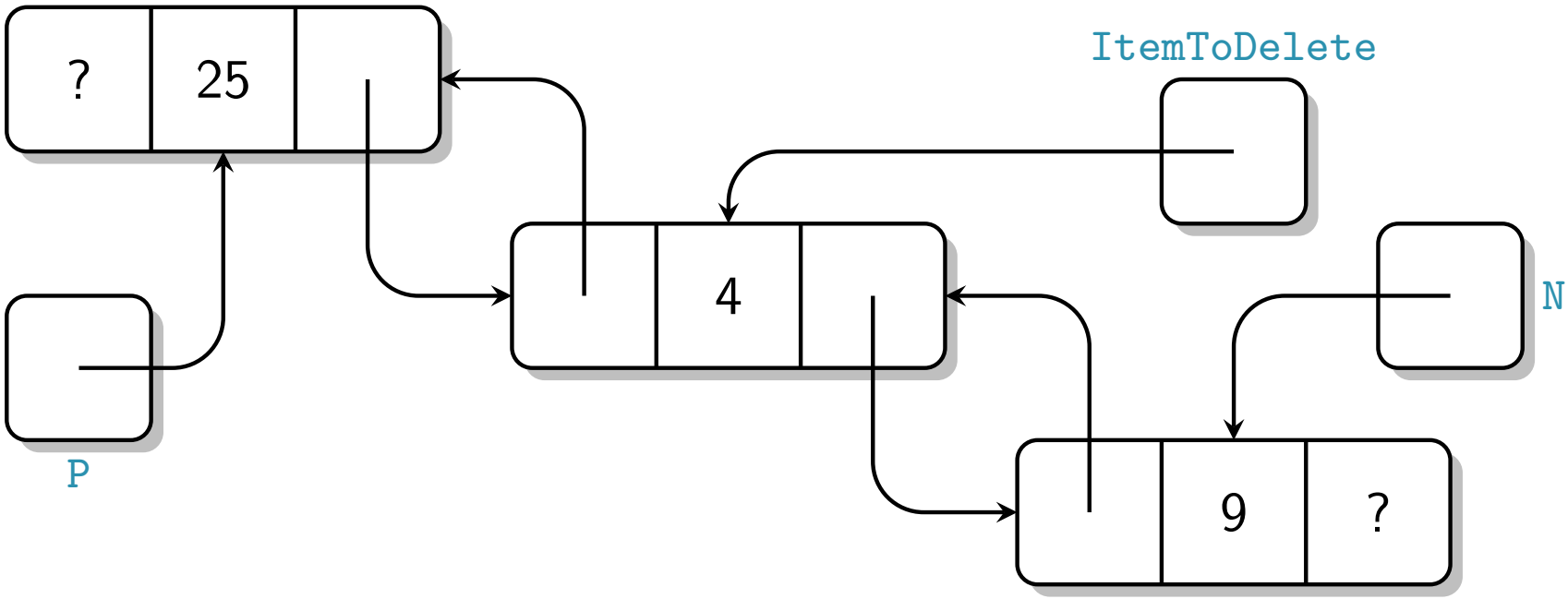
Odebrání položky ze seznamu – obecná pozice 2

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



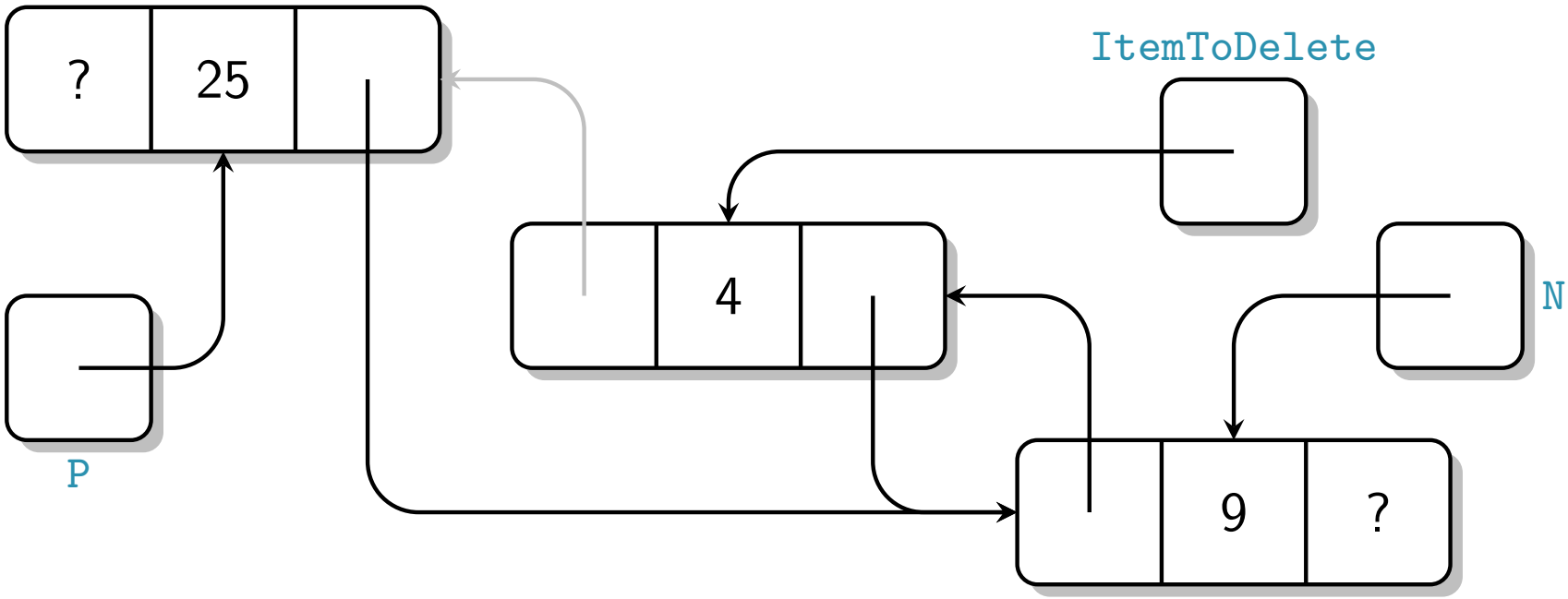
Odebrání položky ze seznamu – obecná pozice 3

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



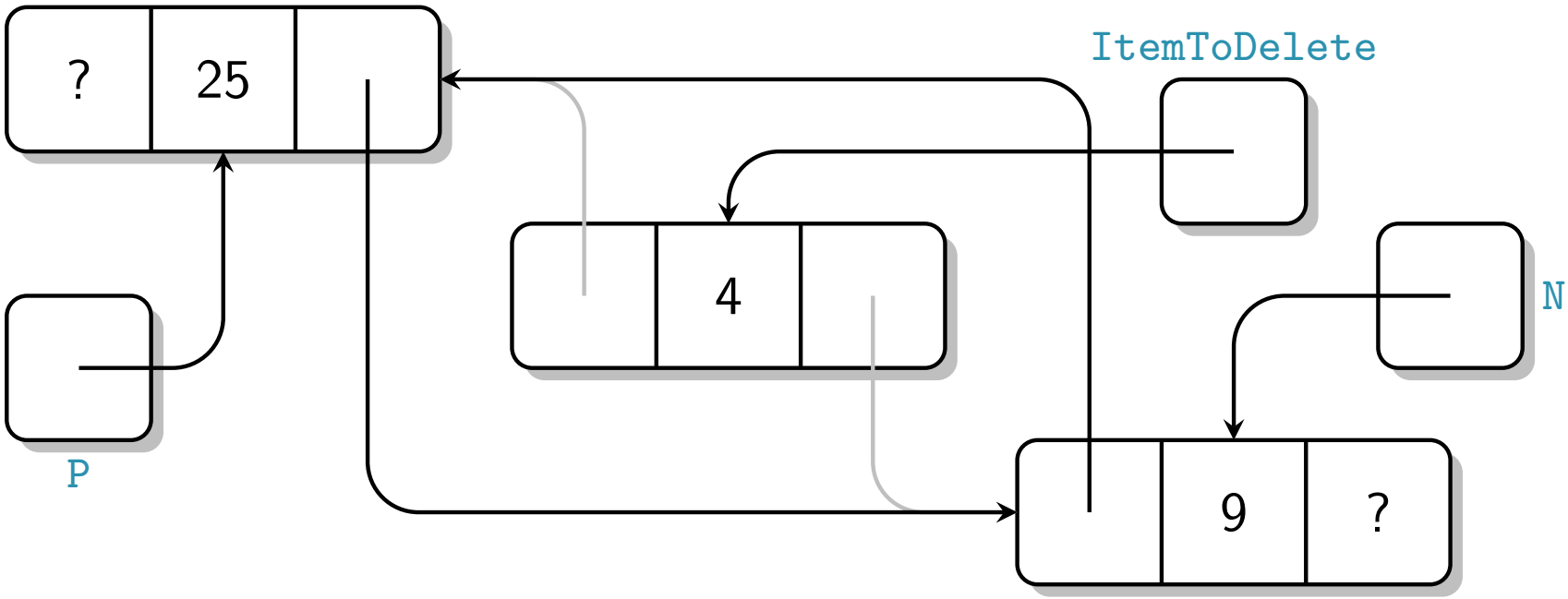
Odebrání položky ze seznamu – obecná pozice 4

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



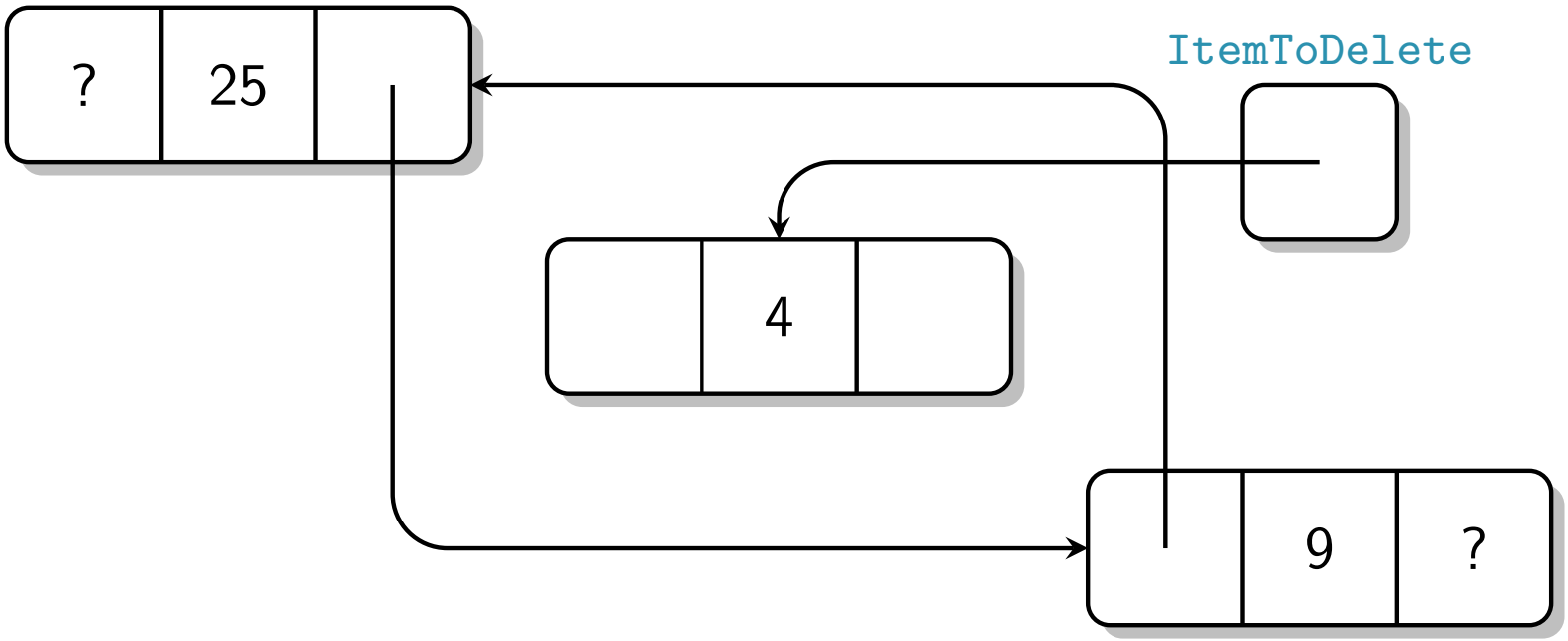
Odebrání položky ze seznamu – obecná pozice 5

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



Odebrání položky ze seznamu – obecná pozice 6

```
263 ListItem* P = ItemToDelete->Prev;
264 ListItem* N = ItemToDelete->Next;
265 P->Next = N;
266 N->Prev = P;
```



Spojová implementace obousměrného seznamu – deklarace

```
1  void Init(List& L);
2  void Init(List& L, const string& InitValue);
3  void Init(List& L, const string InitValues[], const int N);
4  void Init(List& L, const List& Other);
5
6  void Prepend(List& L, const string& NewValue);
7  void Append(List& L, const string& NewValue);
8  void InsertBefore(List& L, ListItem* CurrentItem, const string&
       NewValue);
9  void InsertAfter(List& L, ListItem* CurrentItem, const string& NewValue
       );
10
11 void RemoveHead(List& L);
12 void RemoveTail(List& L);
13 void RemoveFirst(List& L, const string& ValueToRemove);
14 void RemoveLast(List& L, const string& ValueToRemove);
15 void RemoveAll(List& L, const string& ValueToRemove);
16 void Remove(List& L, const ListItem* ItemToRemove);
```

Spojová implementace obousměrného seznamu – deklarace (pokrač.)

```
17 void Clear(List& L);
18
19 ListItem* Search(const List& L, const string& Value);
20 ListItem* ReverseSearch(const List& L, const string& Value);
21
22 bool Contains(const List& L, const string& Value);
23
24 bool IsEmpty(const List& L);
25 int Count(const List& L);
26
27 void Report(const List& L);
28 void ReportStructure(const List& L);
29
30 void InternalCreateSingleElementList(List& L, const string& NewValue);
31 void InternalRemove(List& L, const ListItem* ItemToDelete);
```

Spojová implementace obousměrného seznamu – implementace

```
1  void Init(List& L)
2  {
3     L.Head = nullptr;
4     L.Tail = nullptr;
5  }
6
7  void Init(List& L, const string& InitValue)
8  {
9     InternalCreateSingleElementList(L, InitValue);
10 }
11
12 void Init(List& L, const string InitValues[], const int N)
13 {
14    Init(L);
15    for (int i = 0; i < N; i++)
16    {
17       Append(L, InitValues[i]);
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
18    }
19 }
20
21 void Init(List& L, const List& Other)
22 {
23    Init(L);
24    for (ListItem* p = Other.Head; p != nullptr; p = p->Next)
25    {
26       Append(L, p->Value);
27    }
28 }
29
30 void Prepend(List& L, const string& NewValue)
31 {
32    if (IsEmpty(L))
33    {
34       InternalCreateSingleElementList(L, NewValue);
35    }
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
36    else
37    {
38       ListItem* NewItem = new ListItem;
39       NewItem->Value = NewValue;
40       NewItem->Prev = nullptr;
41       NewItem->Next = L.Head;
42       L.Head->Prev = NewItem;
43       L.Head = NewItem;
44    }
45 }
46
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
54    {
55       ListItem* NewItem = new ListItem;
56       NewItem->Value = NewValue;
57       NewItem->Next = nullptr;
58       L.Tail->Next = NewItem;
59       NewItem->Prev = L.Tail;
60       L.Tail = NewItem;
61    }
62 }
63
64 void InsertBefore(List& L, ListItem* CurrentItem, const string&
       NewValue)
65 {
66    if (CurrentItem != nullptr)
67    {
68       if (CurrentItem == L.Head)
69       {
70          Prepend(L, NewValue);
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
71        }
72        else
73        {
74           ListItem* NewItem = new ListItem;
75           NewItem->Value = NewValue;
76           ListItem* P = CurrentItem->Prev;
77           P->Next = NewItem;
78           NewItem->Prev = P;
79           CurrentItem->Prev = NewItem;
80           NewItem->Next = CurrentItem;
81        }
82     }
83 }
84
85 void InsertAfter(List& L, ListItem* CurrentItem, const string& NewValue
       )
86 {
87     if (CurrentItem != nullptr)
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
88      {
89          if (CurrentItem == L.Tail)
90          {
91              Append(L, NewValue);
92          }
93          else
94          {
95              ListItem* NewItem = new ListItem;
96              NewItem->Value = NewValue;
97              ListItem* N = CurrentItem->Next;
98              N->Prev = NewItem;
99              NewItem->Next = N;
100             CurrentItem->Next = NewItem;
101             NewItem->Prev = CurrentItem;
102         }
103     }
104 }
105
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
106 void RemoveHead(List& L)
107 {
108    if (!IsEmpty(L))
109    {
110       InternalRemove(L, L.Head);
111    }
112 }
113
114 void RemoveTail(List& L)
115 {
116    if (!IsEmpty(L))
117    {
118       InternalRemove(L, L.Tail);
119    }
120 }
121
122 void RemoveFirst(List& L, const string& ValueToRemove)
123 {
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
124     ListItem* p = Search(L, ValueToRemove);
125     if (p != nullptr)
126     {
127         InternalRemove(L, p);
128     }
129 }
130
131 void RemoveLast(List& L, const string& ValueToRemove)
132 {
133     ListItem* p = ReverseSearch(L, ValueToRemove);
134     if (p != nullptr)
135     {
136         InternalRemove(L, p);
137     }
138 }
139
140 void RemoveAll(List& L, const string& ValueToRemove)
141 {
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
142     ListItem* p;
143     while ((p = Search(L, ValueToRemove)) != nullptr)
144     {
145         InternalRemove(L, p);
146     }
147 }
148
149 void Remove(List& L, const ListItem* ItemToRemove)
150 {
151     if (ItemToRemove != nullptr)
152     {
153         InternalRemove(L, ItemToRemove);
154     }
155 }
156
157 void Clear(List& L)
158 {
159     while (!IsEmpty(L))
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
160    {
161       InternalRemove(L, L.Head);
162    }
163 }
164
165 ListItem* Search(const List& L, const string& Value)
166 {
167    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
168    {
169       if (p->Value == Value)
170       {
171          return p;
172       }
173    }
174    return nullptr;
175 }
176
177 ListItem* ReverseSearch(const List& L, const string& Value)
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
178 {
179    for (ListItem* p = L.Tail; p != nullptr; p = p->Prev)
180    {
181       if (p->Value == Value)
182       {
183          return p;
184       }
185    }
186    return nullptr;
187 }
188
189 bool Contains(const List& L, const string& Value)
190 {
191    return Search(L, Value) != nullptr;
192 }
193
194 bool IsEmpty(const List& L)
195 {
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
196     return L.Head == nullptr;
197 }
198
199 int Count(const List& L)
200 {
201     int counter = 0;
202     for (ListItem* p = L.Head; p != nullptr; p = p->Next)
203     {
204         counter += 1;
205     }
206     return counter;
207 }
208
209 void Report(const List& L)
210 {
211     for (ListItem* p = L.Head; p != nullptr; p = p->Next)
212     {
213         cout << p->Value << "\t";
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
214     }
215     cout << endl;
216 }
217
218 void ReportStructure(const List& L)
219 {
220     cout << "L.Head: " << L.Head << endl;
221     cout << "L.Tail: " << L.Tail << endl;
222     for (ListItem* p = L.Head; p != nullptr; p = p->Next)
223     {
224         cout << "Item address: " << p << endl;
225         cout << "Value: " << p->Value << endl;
226         cout << "Prev: " << p->Prev << endl;
227         cout << "Next: " << p->Next << endl;
228         cout << endl;
229     }
230 }
231
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
232 void InternalCreateSingleElementList(List& L, const string& NewValue)
233 {
234    L.Head = L.Tail = new ListItem;
235    L.Head->Value = NewValue;
236    L.Head->Prev = nullptr;
237    L.Head->Next = nullptr;
238 }
239
240 void InternalRemove(List& L, const ListItem* ItemToDelete)
241 {
242    if (ItemToDelete == L.Head && ItemToDelete == L.Tail)
243    {
244       L.Head = nullptr;
245       L.Tail = nullptr;
246    }
247    else
248    {
249       if (ItemToDelete == L.Head)
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
250         {
251             L.Head = L.Head->Next;
252             L.Head->Prev = nullptr;
253         }
254         else
255         {
256             if (ItemToDelete == L.Tail)
257             {
258                 L.Tail = L.Tail->Prev;
259                 L.Tail->Next = nullptr;
260             }
261             else
262             {
263                 ListItem* P = ItemToDelete->Prev;
264                 ListItem* N = ItemToDelete->Next;
265                 P->Next = N;
266                 N->Prev = P;
267             }
```

Spojová implementace obousměrného seznamu – implementace (pokrač.)

```
268         }
269     }
270     delete ItemToDelete;
271 }
```

Srovnání implementací seznamu

Pole

```
236 void InternalCreateSingleElementList(List& L, const string& NewValue)
237 {
238    L.Head = L.Tail = InternalAllocateItem(L);
239    L.Items[L.Head].Value = NewValue;
240    L.Items[L.Head].Prev = Nothing;
241    L.Items[L.Head].Next = Nothing;
242 }
```

Spojové struktury

```
232 void InternalCreateSingleElementList(List& L, const string& NewValue)
233 {
234    L.Head = L.Tail = new ListItem;
235    L.Head->Value = NewValue;
236    L.Head->Prev = nullptr;
237    L.Head->Next = nullptr;
238 }
```

Srovnání implementací seznamu (pokrač.)

Pole

```
169 int Search(const List& L, const string& Value)
170 {
171    for (int p = L.Head; p != Nothing; p = L.Items[p].Next)
172    {
173       if (L.Items[p].Value == Value)
174       {
175          return p;
176       }
177    }
178    return Nothing;
179 }
```

Spojové struktury

```
165 ListItem* Search(const List& L, const string& Value)
166 {
167    for (ListItem* p = L.Head; p != nullptr; p = p->Next)
168    {
169       if (p->Value == Value)
170       {
171          return p;
172       }
173    }
174    return nullptr;
175 }
```

Srovnání implementací seznamu (pokrač.)

Pole

```
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L, NewValue);
52    }
53    else
54    {
55       int NewItemIndex = InternalAllocateItem(L);
56       L.Items[NewItemIndex].Value = NewValue;
57       L.Items[NewItemIndex].Prev = L.Tail;
58       L.Items[NewItemIndex].Next = Nothing;
59       L.Items[L.Tail].Next = NewItemIndex;
60       L.Tail = NewItemIndex;
61    }
62 }
```

Spojové struktury

```
47 void Append(List& L, const string& NewValue)
48 {
49    if (IsEmpty(L))
50    {
51       InternalCreateSingleElementList(L,
               NewValue);
52    }
53    else
54    {
55       ListItem* NewItem = new ListItem;
56       NewItem->Value = NewValue;
57       NewItem->Next = nullptr;
58       L.Tail->Next = NewItem;
59       NewItem->Prev = L.Tail;
60       L.Tail = NewItem;
61    }
62 }
```

Kontrolní otázky

1. Máme danou strukturu a čtyři proměnné:

```
1 struct MyStruct
2 {
3    int Id;
4    double Value;
5 };
6
7 MyStruct a, b;
8 MyStruct *p, *q;
```

   Co se stane pokud provedu `a = b;`? A co pokud provedu `p = q;`?

2. Jak alokuji dynamicky strukturu z předchozí otázky? Pomocí jakého operátoru?
3. Jak uvolním z paměti dynamicky alokovanou strukturu, kterou jsem vytvořil v předchozí otázce?
4. Mám deklarovanou proměnnou `MyStruct a;`. Jak se dostanu k položce `Value` v proměnné `a`?

Kontrolní otázky (pokrač.)

5. Mám deklarovanou proměnnou `MyStruct *p;`. Jak se dostanu k položce `Value` v proměnné `p` pomocí jediného operátoru (bez dereference)?
6. Mám deklarovanou proměnnou `MyStruct *p;`. Jak se dostanu k položce `Value` v proměnné `p` pomocí operátoru dereference?
7. Mám následující kód:

```
1 double *p = new double;
2 delete p;
3 *p = 15;
```

   Co tento kód způsobí?

8. Mám následující kód:

```
1 double *p = new double;
2 p = NULL;
```

   Co tento kód způsobí? K čemu v paměti dojde?

Kontrolní otázky (pokrač.)

9. Mám následující kód:

```
1 double *p = new double;
2 p = NULL;
3 *p = 15;
```

   Co tento kód způsobí? K čemu v paměti dojde?

10. Mám seznam implementovaný pomocí pole. Musím tento seznam (především to pole) explicitně uvolňovat z paměti pomocí operátoru `delete`?
11. Mám seznam implementovaný pomocí dynamicky alokovaných struktur. Musím tento seznam explicitně uvolňovat z paměti pomocí operátoru `delete`?
12. Co je to zarážka (angl. sentinel)? Co nahrazuje?
13. K čemu slouží zarážka?

Kontrolní otázky (pokrač.)

14. Jak se změní algoritmus vkládání prvku na začátek obousměrného seznamu, pokud využijeme zarážku?
15. Jak se změní algoritmus smazání prvku z obousměrného seznamu, pokud využijeme zarážku?
16. Kdy je vhodné využít zarážku? Zaměřte se na spotřebovanou paměť a počet prvků v uvažovaném seznamu.
17. Kdy je vhodné použít seznam implementovaný v poli a kdy pomocí dynamických struktur? Zaměřte se na rychlost operací vložení (s tím spojené alokování položky), operací mazání (s tím spojené dealokovaní položky). Dále se zaměřte na možnou změnu kapacity seznamu.
18. Jestliže budu k seznamu přistupovat pouze pomocí funkcí `Insert`, `Delete` nebo `Search` a změním implementaci seznamu. Musím měnit i okolní kód, kde se seznam využívá?
19. Pokud implementuji frontu pomocí dynamických struktur, potřebuji kruhový buffer a podobné „triky“ s polem?

Kontrolní otázky (pokrač.)

20. Zkuste navrhnout efektivní algoritmus pro třídění seznamu. Uvědomte si, že k položkám seznamu můžeme přistupovat pouze sekvenčně, jako například k datům v souboru. Jaký Vám známý algoritmus pro třídění lze na tento problém aplikovat?
21. Zkuste navrhnout rekurzívní algoritmus pro vyhledání prvku v obousměrném seznamu.
22. Jak spočítáte počet položek v seznamu? Má na tento algoritmus vliv zda je seznam obousměrný nebo jednosměrný?
23. Jak byste v obousměrném seznamu implementovali mazání k-té položky.
24. Popište algoritmus mazání položky z obousměrného seznamu. Položka určená ke smazání se musí nejprve vyhledat podle dat uložených v položce.
25. Popište algoritmus vkládání na začátek obousměrného seznamu.

Děkuji za pozornost