Algoritmy 2

Petr Osička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

O Kurzu

Přednáška

- Aula, čtvrtky od 8:00
- Zkouška: ústní

Cvičení

- Dozvíte se na cvičení

Sylabus

- Datové struktury (vyhledávání)
- Základní grafové algoritmy

Datová struktura

Systematický způsob organizace dat (v paměti počítače). Důležitý aspekt programování.

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

– **Linus Torvalds**

Data dominates.

– **Rob Pike, 5. pravidlo programování**

Datová struktura

Způsob organizace dat s nějakým účelem (co ukládáme, jak k tomu přistupujeme)

Abstraktní datová struktura

Příklady:

- *Množina*: přidáváme prvky, odebíráme prvky, je prvek v množině?, sjednocení, průnik atd.
- *Slovník*: vazba klíč a data, vytváření a rušení vazby, vyhledávání dat ke klíči, minimální a maximální klíč atd.

Teoretické partie:

- asymptotické složitosti operací v nejhorším, průměrném případě Složitosti operací závisí na počtu prvků ve struktuře.
- jak nejrychleji lze operaci udělat
- vyvážení pamětové a časové náročnosti
- vztah mezi složitostmi různými operacemi

Praktické partie

- Jak lze strukturu naprogramovat (často více možností)
- Jak ji lze optimalizovat (např. lze ji přizpůsobit fungování počítače, cache atd.)?
- Jak ji můžeme upravit, aby pro konkrétní aplikaci lépe fungovala.

Pointer Machine jako základní model

Struktura

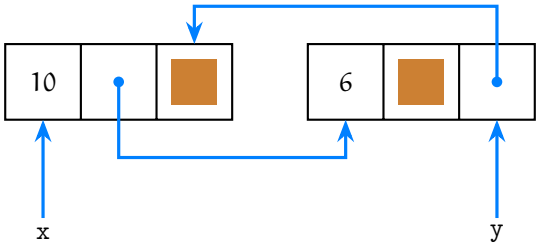
- pojmenovaný soubor pojmenovaných položek: (`jméno:typ`)
- přístup k položkám přes operátor tečky: `promenna.polozka`
- referenční sémantika

```
struct Node
   id: Key
   next: Node
   prev: Node
```

Referenční sémantika

Instance struktury odpovídá jednomu místu v paměti, kde je uložena.
Reference je jednoznačný identifikátor instance struktury, neplatná reference je `nil`.

```
x ← Node()
y ← Node()
x.id ← 10
y.id ← 6
x.next ← y
y.prev ← x
```



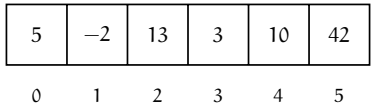
Pole

- Uložení fixního počtu prvků. Tomuto počtu se říká *velikost* pole, zadává se při vytvoření pole.
- Přístup k prvkům pomocí indexů v rozsahu `0,1, ... ,n-1`, kde `n` je velikost pole. Přístup k prvku na indexu indexu `i` je pomocí `A[i]`.
- Složitost přístupu k prvku na indexu `i` je $O(1)$, protože nezávisí na velikosti pole.

V programovacích jazycích, kde jsou prvky v poli uloženy v paměti za sebou, je operace přístupu k prvku na indexu založená na výpočtu

adresa začátku + index * velikost prvku.

```
A ← [6]Key{5, -2, 13, 3, 10, 42}     — vytvoření pole, typ je obecně []typ-prvku
B ← [5]Key     — prvky inicializovány na výchozí hodnoty
len(A)     — procedura pro velikost
```



V pointer machine mají pole referenční sémantiku. Pro objekty s referenční sémantikou máme procedury

```
clone(A)
copy(dest, src)
```

které tvoři kopii argumentu (= nově vytvořené objekty), případně zkopírují hodnoty položek či prvků z `src` do `dest` (U polí se kopíruje od začátku `src` na začátek `dest` až do konce `src` nebo `dest`)

Pole A jako podmnožina {0,1, ... ,len(A)-1}

- `A[i] = 1` pokud `i` patří do podmnožiny, jinak `A[i] = 0`
- operace vložení prvku

```
A[i] ← 1
```

- operace smazání prvku

```
A[i] ← 0
```

- test členství k v množině

```
A[k] == 1
```

- operace mají konstantní složitosti

Pole jako slovník

```
struct Array_Dict
   data: []Key
   top: Int
```

Pro jednoduchost vkládáme jenom klíče. Pole je zaplněno od začátku po index `top - 1`.

Vložení klíče `k` do `Array_Dict ar`

```
1  if (ar.top < len(ar.data))
2     ar.data[ar.top] ← k
3     ar.top ← ar.top + 1
```

Smazání prvku na indexu `i`

```
1  if (i < ar.top)
2     ar.data[i] ← ar.data[ar.top - 1]
3     ar.top ← ar.top - 1
```

Vyhledávání v poli

Chceme zjistit, na jakém indexu se nachází klíč `k`. (Bez újmy na obecnosti předpokládáme, že prohledáváme pole celé).

`array-search`
⟵ `array: []Key` — *prohledávané pole*
⟵ `k: Key` — *hledaný klíč*
⟶ `Int` — *index klíče* `k` *nebo* `-1`, *pokud v* `ar` *není*

```
1 for i in 0 ..< len(array)
2    if (array[i] = k) then return i
3 return -1
```

V nejhorším případě je složitost $\Theta(n)$, kde n je velikost pole.

Složitost v průměrném případě

- Potřebujeme znát *frekvence vyhledávání klíčů*. Zavedeme

$$f_i = \frac{\text{počet zavolání s argumentem A[i].key}}{\text{počet všech zavolání}}$$

  f_i je tedy *frekvence* hledání klíče na indexu i. Hned vidíme, že platí

$$\sum_{i=0}^{n-1} f_i = 1.$$

- *Průměrný počet porovnání* potom je

$$f_0 \cdot 1 + f_1 \cdot 2 + \cdots + f_{n-1} \cdot n$$

Pokud například $f_i = \frac{1}{n}$ pro všechna i, pak

$$\frac{1+2+3+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Průměrná složitost (úspěšného) vyhledávání je nejlepší, pokud

$$f_0 \geqslant f_1 \geqslant \cdots \geqslant f_{n-1}$$

V extrémních případech může být průměrný počet porovnání omezen konstantou.

Vyhledávání v setříděném poli

Pro všechny indexy `i,j` máme `ar[i]` $\leqslant$ `ar[j]`.

`array-search-ii`
⟵ `array: []Key` — *prohledávané pole*
⟵ `k: Key` — *hledaný klíč*
⟶ `Int` — *index klíče* `k` *nebo* `-1`, *pokud v* `ar` *není*

```
1  for i in 0 ..< len(array)
2     if (array[i] = k) then return i
3     if (array[i] > k) then return -1
4  return -1
```

Složitost v nejhorším případě zůstává lineární.

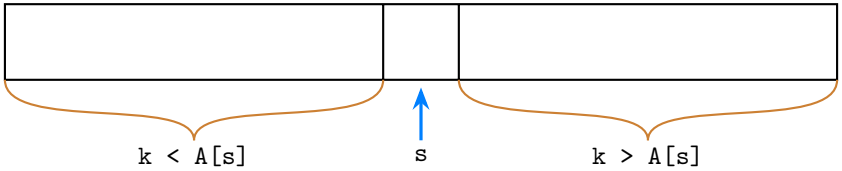
Binární vyhledávání

Při hledání klíče k v poli délky n, porovnáme jej s prvkem na indexu

$$s = \left\lfloor \frac{n}{2} \right\rfloor .$$

Pokud na indexu s klíč k není, pak

- je-li na indexu s klíč menší než k, potom se k může nacházet pouze na indexech větších než s.
- je-li na indexu s klíč větší než k, potom se k může nacházet pouze na indexech menších než s.



`binary-search`
⟵ `array: []Key` — *prohledávané pole*
⟵ `k: Key` — *hledaný klíč*
⟶ `Int` — *index klíče* `k` *nebo* `-1`*, pokud v* `ar` *není*

```
1  l ← 0
2  p ← len(array) - 1
3  while (l ⩽ p)
4     s ← floor((l + p) / 2)
5     if (array[s] = k) then return s
6     if (array[s] > k) then p ← s - 1
7     if (array[s] < k) then l ← s + 1
8  return -1
```

Analýza algoritmu

- (Mimo řádku 8) vždy platí `array[l]` $\leqslant$ k nebo k $\leqslant$ `array[p]`. Je tomu tak na začátku algoritmu, a řádky 4–7 tuto nerovnost zachovávají. (*Kdy jedna z nerovností neplatí?*)
- Každé provedení `while` zmenší `p-l+1` na polovinu původní hodnoty nebo ještě o jedna méně, minimálně však o `1`.
- Z předchozích bodů pak plyne, že pokud je k v poli, na řádku 3 jednou nastane rovnost. Nejpozději v situaci, kdy `p=l`. Naopak, pokud k v poli není, jednou dojde k tomu, že `l>p`.
- Složitost v nejhorším případě je dána rekurencí

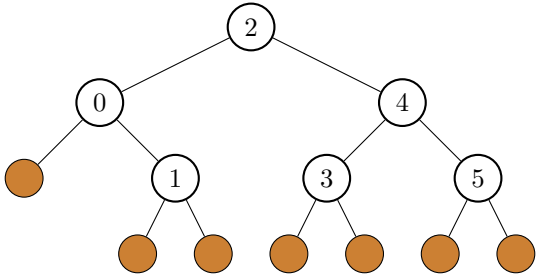
$$\begin{aligned} T(1) &= O(1), \\ T(n) &= T(n/2) + O(1), \end{aligned}$$

jejímž řešením je $\Theta(\log n)$ (*např. Master theorem*).

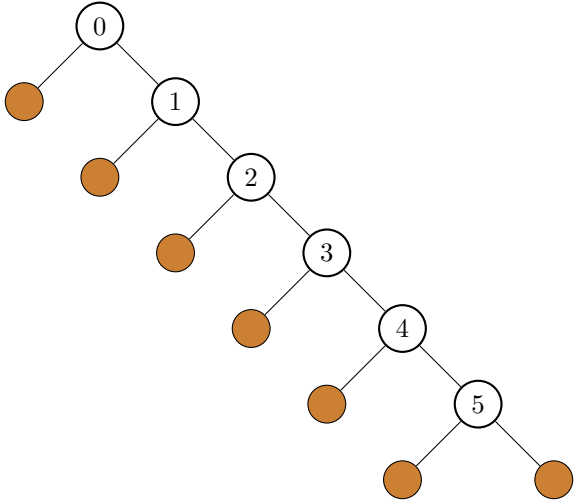
Strom vyhledání

Pro dvojici (`l`,`p`) sestavíme binární strom

- kořen: odpovídající hodnota `s`,
- vlevo: strom pro dvojici (`l`,`s-1`),
- vpravo: strom pro dvojici (`s+1`,`p`).
- pokud `l` $>$ `p`, výsledek je bronzový uzel.



Lze zobecnit na jakýkoliv způsob výběru uzlu s. Jakému principu odpovídá následující strom?



Alternativní výběr s: Interpolační vyhledávání

Předpoklad: klíče jsou čísla, v poli máme přibližně stejné rozdíly mezi sousedními čísly.

Položíme-li

$$x = \texttt{A[l]},$$
$$y = \texttt{A[p]},$$

(a máme tedy $x \leqslant k \leqslant y$), pak můžeme spočítat

$$r = (k - x)/(y - x)$$

a index s jako $l + (p - l) * r$.

Složitost: Za rozumných předpokladů je průměrná složitost $O(\lg \lg n)$

Dynamické pole

Řeší problém s fixní velikostí pole.

```
struct Dynamic-Array
   data: []Key
   top: Int
```

Umožníme přidávat a odebírat prvky, pole `data` se bude podle potřeby zvětšovat a zmenšovat

```
enlarge
⟵ A: []Key
⟶ []Key
```

```
1 new ← [2 * len(A.data)]Key
2 copy(new, A.data)
3 return new
```

```
shrink
⟵ A: []Key
⟶ []Key
```

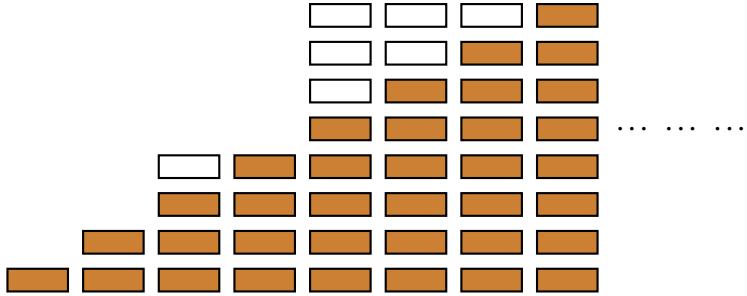
```
1 new ← [len(A.data) / 2]Key
2 copy(new, A.data)
3 return new
```

```
append
⟵ A: Dynamic-Array
⟵ k: Key ⟵ vkládaný klíč
```

```
1 if (A.data = nil) then A.data ← [1]Key
2 if (A.top = len(A.data)) then A.data ← enlarge(A.data)
3 A.data[A.top] ← k
4 A.top ← A.top + 1
```

Složitost

Měříme počet operací zápisu do pole provedených během provedení n operací `append`, pokud začínáme s čerstvě inicializovanou strukturou. Počet zápisů při i-té operaci označíme $c(i)$.



i	1	2	3	4	5	6	7	8	9	$10-16$	17
$c(i)$	1	2	3	1	5	1	1	1	9	1	17

Máme tak

$$c(i) = \begin{cases} i & \text{pokud je } i-1 \text{ mocninou 2,} \\ 1 & \text{jinak.} \end{cases}$$

Dále

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leqslant n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n = 3n. \end{aligned}$$

Na jednu operaci `append` tak průměrně (rozpočítáno přes n operací) vyjdou maximálně 3 zápisy do pole. Složitost jedné operace je tak $\Theta(1)$.

Takovému přístupu ke složitosti, kdy rozdělíme složitost provedení celé sekvence operací na jednotlivé operace, říkáme *amortizovaná složitost* (toto bereme jenom jako příklad, s amortizovanou složitostí se setkáte později během studia).

Odebírání prvků

```
unordered-remove
⟵ A: Dynamic-Array
⟵ i: Int — index odebíraného prvku
```

```
1 A.data[i] ← A.data[A.top - 1]
2 A.top ← A.top - 1
3 if (A.top ⩽ len(A.data) / 4) then A.data ← shrink(A.data)
```

```
pop
⟵ A: Dynamic-Array
⟶ Key — vrácený prvek
```

```
1 ret ← A.data[A.top - 1]
2 A.top ← A.top - 1
3 return ret
```

```
ordered-remove
⟵ A: Dynamic-Array
⟵ i: Int — index odebíraného prvku
```

```
1 if (A.top - 1 ⩽ len(A.data) / 4) then A.data ← shrink(A.data)
2 for j in i ..< A.top - 1
3    A.data[j] ← A.data[j + 1]
4 A.top ← A.top - 1
```

Složitosti

- Pokud nedojde ke zmenšení kapacity, mají `pop` a `unordered-remove` konstantní složitost, jinak lineární. Amortizovanou složitost spolu s `append` mají konstantní.
- `ordered-remove` má složitost lineární, i když nedojde ke zmenšení kapacity.

Další updaty

```
assign-at
⟵ A: Dynamic-Array
⟵ i: Int — index, kde měníme obsah
⟵ k: Key — vkládaná hodnota
```

```
1  cap ← len(A.data)
2  while (i ⩾ cap) do cap ← cap * 2
3  if (cap > len(A.data)
4     new ← [cap]Key
5     copy(new, A.data)
6     A.data ← new
7  A.data[i] ← k
8  for j in A.top ..< k do A.data[j] ← 0
9  if (A.top < k) then A.top ← k + 1
```

- na řádcích 1–6 zajistíme dostatečnou velikost `A.data` pro vložení prvku na index k
- na řádku 8 nastavíme hodnoty mezi koncem původního obsahu a nastaveným prvkem na defaultní hodnotu klíče.

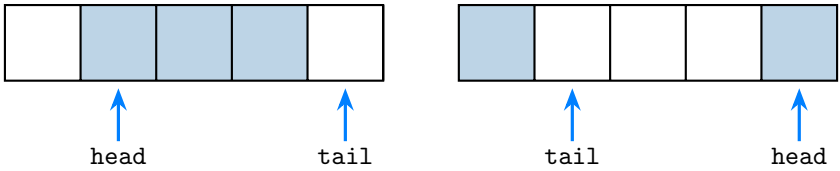
Pokud je `i ⩾ A.top` je složitost `assign-at` lineární. Může totiž dojít ke zvětšení `A.data`, a dojde k vynulování prvků na řádku 8.

Analogií `assign-at` je operace `inject-at`, při které se všechny prvky na indexech větší než `i` na o jedna větší index. (Operace je duální k `ordered-remove`).

Cyklický buffer

Původní účel: mezipamět mezi producentem a konzumentem.
Lze zapisovat a číst, čte v pořadí, ve kterém bylo zapsáno.
Iluze kruhovosti pomocí modulo aritmetiky s indexy.

```
struct Buffer
   data:[] Key
   head: Int — index pro čtení
   tail: Int — index pro zápis
```

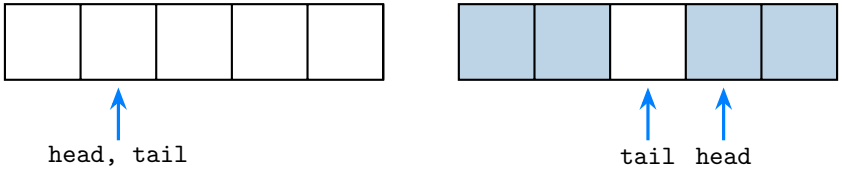


Test prázdného bufferu `Q`

```
Q.head = Q.tail
```

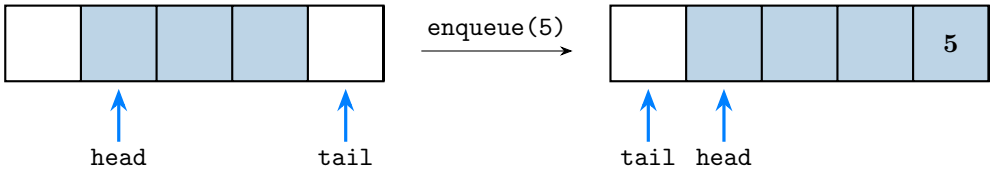
Test plného bufferu `Q`

```
(Q.tail + 1) mod len(Q.data) = Q.head
```



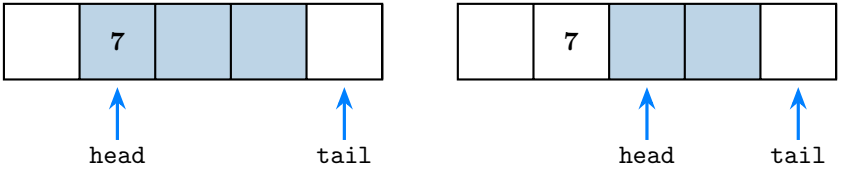
```
write
 ⟵ Q: Buffer
 ⟵ k: Key
```

```
1 if (Q.tail + 1) mod len(Q.data) = Q.head then quit — buffer je plný
2 Q.data[Q.tail] ← k
3 Q.tail ← (Q.tail + 1) mod len(Q.data)
```



```
read
⟵ Q: Buffer
⟶ Key
```

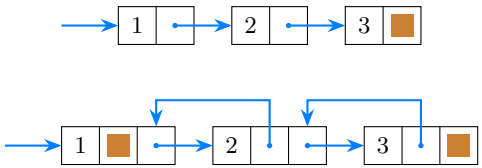
```
1  x ← Q.data[Q.head]
2  Q.head ← (Q.head + 1) mod len(Q.data)
3  return x
```



Spojový seznam

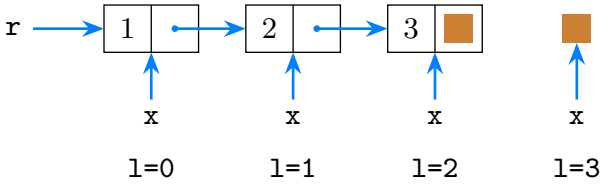
- V pointer machine: posloupnost uzlů spojených pomocí pointerů.
- Základní dělení: jednosměrný (pointer pouze na následující uzel), obousměrný
- Pamatujeme si první uzel seznamu (ten při pohledu zvnějšku reprezentuje celý seznam)
- Prázdný seznam je `nil`
- Pokud následující nebo předchozí uzel neexistuje, je příslušná položka `nil`.

```
struct Node
   id: Key
   next: Node — další uzel v seznamu
   prev: Node — předchozí uzel v seznamu, pouze o obousměrných seznamů
```



- máme-li uzel v seznamu, je to vlastně podseznam (až do konce seznamu)
- *Idiom procházení seznamu:* `x` ← `x.next`

Př. Délka seznamu:



Jednosměrné seznamy

```
search
⟵ r: Node — první uzel seznamu
⟵ k: Key — hledaný klíč
⟶ Node — uzel s klíčem k nebo nil
```

```
1 x ← r
2 while (x ≠ nil and x.id ≠ k)
3    x ← x.next
4 return x
```

Složitost

- V nejhorším případě lineární vzhledem k délce seznamu `r`.
- Lze použít analýzu analogickou sekvenčnímu vyhledávání v poli (včetně průměrného případu).

Přístup k n-tému prvku

`element-at`
⟵ `r: Node` — *první uzel seznamu*
⟵ `n: Int` — *index hledaného uzlu*
⟶ `Node` — *nalezený uzel nebo* `nil`

```
1  m ← 0
2  x ← r
3  while (x ≠ nil and m < n)
4     x ← x.next
5     m ← m + 1
6  return x
```

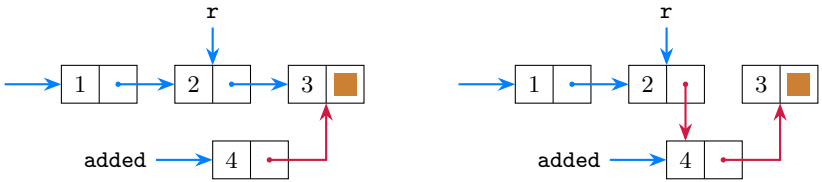
Složitost

- Složitost je lineární, odpovídá minimu z délky seznamu `r` a `n`.
- Tady je rozdíl oproti polím: nemáme přístup k n-tému prvku v konstantním čase.

Vložení uzlu za uzel

`insert-after`
⟵ `r: Node` — *uzel, za který vkládáme*
⟵ `added: Node` — *vkládaný uzel*

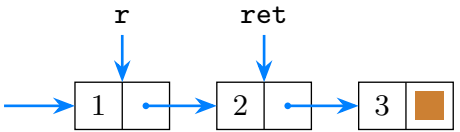
```
1  added.next ← r.next
2  r.next ← added
```

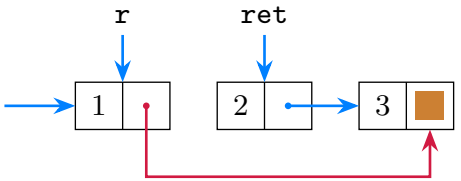


Odstranění následujícího uzlu

`remove-after`
⟵ `r:node` — *uzel, jehož následníka odstraníme*
⟶ `node` — *odstraněný uzel*

```
1  ret ← r.next
2  if (r.next ≠ nil) then r.next ← r.next.next
3  return ret
```





Přidávání uzlu

- Procedury, které by mohli změnit první uzel, musí vracet pointer na první uzel seznamu
- Použití je potom `x ← operace(x, ...)`

```
insert-at
⟵ r: Node — začátek seznamu
⟵ added: Node — přidaný uzel
⟵ n: Int — index přidaného uzlu
⟶ Node — začátek seznamu
```

```
1  if (n = 0)
2     added.next ← r
3     return added
5  x ← element-at(r, n - 1)
6  if (x ≠ nil) then insert-after(x,added)
7  return r
```

Odstranění uzlu

Odstranění prvního uzlu je speciální případ. V ostatních případech musíme nejdřív najít uzel před ostraňovaným uzlem.

```
remove-at
⟵ r: Node — začátek seznamu
⟵ n: Int — index odstraněného uzlu
⟶ Node — odstraněný uzel
⟶ Node — nový začátek seznamu
```

```
1  if (r = nil) then return nil, nil
2  if (n = 0) then return r, r.next
3  x ← element-at(r, n-1)
4  if (x ≠ nil) then x ← remove-after(x)
5  return x, r
```

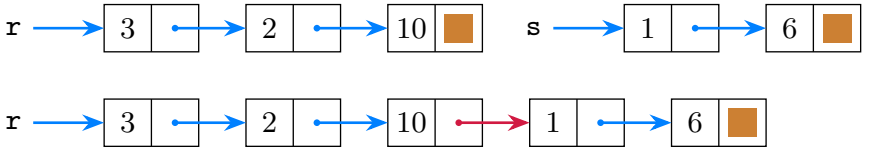
```
remove
⟵ r: Node — začátek seznamu
⟵ d: Node — odstraňovaný uzel, víme že je v seznamu
⟶ Node — nový začátek seznamu
```

```
1 if (d == r) then return r.next
2 x ← r
3 while x.next ≠ d do x ← x.next
4 remove-after(x)
5 return r
```

Spojení seznamů

`concatenate`
⟵ `r: Node` — *první uzel prvního seznamu*
⟵ `s: Node` — *první uzel druhého seznamu*
⟶ `Node` — *první uzel seznamu vzniklého spojením* `r` *a* `s`

```
1  if (r = nil) then return s
2  if (s = nil) then return r
3  x ← r
4  while (x ≠ nil) do x ← x.next
5  x.next ← s
6  return r
```



Adaptace algoritmu pro pole k práci na seznamu

Pole a seznamy uchovávají poslopnosti prvků, algoritmické myšlenky lze přenášet. Jsou nutné drobné adaptace.

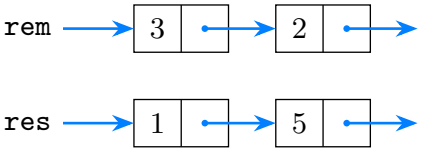
Není obtížné si představit, že v seznamu uchováváme uzly setříděny (vzestupně) podle klíčů (tj. podle položky `id`) a že máme k dispozici algoritmus

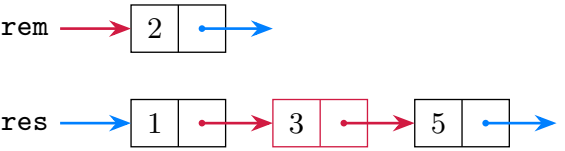
```
ordered-insert
 ⟵ r: Node — první uzel seznamu
 ⟵ added: Node — přidaný uzel
 ⟶ Node — první uzel seznamu po přidání
```

která přidá uzel do uspořádaného seznamu na správné místo tak, aby zůstal uspořádaný.

`insert-sort`
⟵ `r: Node` — *první uzel seznamu*
⟶ `Node` — *první uzel setřízeného seznamu*

```
1  result ← nil
2  remainder ← r
3  while (remainder ≠ nil)
4     x ← remainder
5     remainder ← remainder.next
6     result ← ordered-insert(result, x)
7  return result
```





```
partition
⟵ r: Node — začátek děleného seznamu (předpokládáme, že je neprázdný)
⟶ Node — uzel s pivotem
⟶ Node — začátek seznamu s klíči menšími než pivot
⟶ Node — začátek seznamu s klíči většími než pivot
```

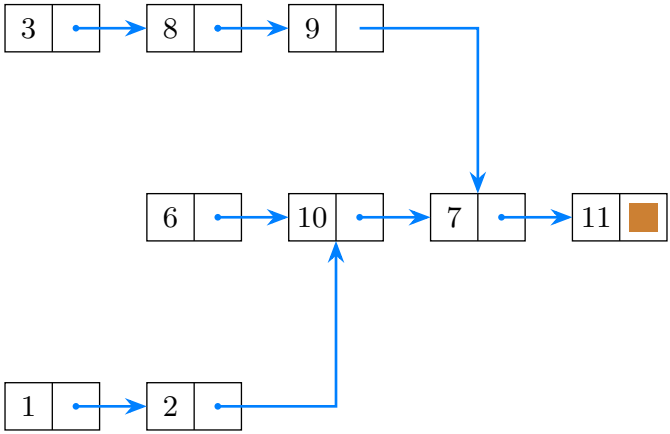
```
1  pivot ← r
2  remainder ← r.next
3  left ← nil
4  right ← nil
5  while (remainder ≠ nil)
6     x ← remainder
7     remainder ← remainder.next
8  if (x.id ⩽ pivot.id) then left ← insert-at(left, x, 0)
9  else right ← insert-at(right, x, 0)
10 return pivot, left, right
```

```
quicksort
⟵ r: Node — začátek seznamu k setřídění
⟶ Node — začátek setřízeného seznamu
```

```
1 pokud if (r == nil or r.next == nil) then return r
2 pivot, left, right ← partition(r)
3 left ← quicksort(left)
4 right ← quicksort(right)
5 insert-at(right, pivot, 0)
6 return concatenate(left, right)
```

Složitější konstrukce

Například více seznamů může sdílet konce. Použití ve specifických případech (např. seznamy jenom čteme, operace měnící seznam vytváří novou kopii seznamu, atd.)



Zarážka

- Speciální uzel na začátku seznamu. Ignorujeme jeho klíč, udržujeme jej jako představitele seznamu. Samotný seznam začíná až jeho následníkem.
- Výhoda: funkce, které modifikují seznam, nemusí vracet nový uzel.
- Nevýhoda: ztratíme flexibilitu v práci s podseznamy.

Př.

```
insert-at
 ⟵ r: Node — zarážka
 ⟵ added: Node — přidaný uzel
 ⟵ n: Int — index přidaného uzlu
```

```
1  x ← element-at(r, n)
2  if (x ≠ nil) insert-after(x,added)
```

Obousměrný seznam

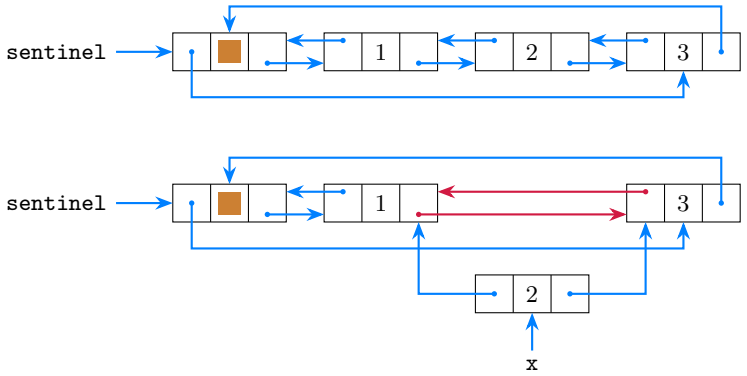
- Lze procházet oběma směry (`x ← x.next` a také `x ← x.prev`)
- Zjednoduší se operace přidání a mazání (nemusíme hledat předchozí uzel od začátku seznamu, máme-li sentinel, nepotřebujeme jej předávat proceduře)

Př.

```
remove
⟵ r: Node — mazaný uzel, který není sentinelem
```

```
1  r.prev.next ← r.next
2  if (r.next ≠ nil) then r.next.prev ← r.prev
```

Pokud dáme sentinel (ze začátku seznamu) i na konec seznamu, dostaneme cyklický seznam. Operace nad ním jsou pak velmi jednoduché, vypadnou i testy na `nil` na řádku 2.



Pozn: Odebraný uzel snadno do seznamu vrátit, pokud tam zůstanou jeho bývalí sousedi ve stejném pořadí hned za sebou.

Zásobník

Princip LIFO (last in, first out)

Hlavní operace

- `push` — vkládání prvku
- `pop` — vrátí naposled přidaný prvek, a odstraní jej

Lze implementovat pomocí pole (viz slovník pomocí pole), pomocí dynamického pole (`append`, `pop`), pomocí seznamu (`insert-at`, `remove-at` pro `n=0`).

Další operace (např. `size`, `empty?`, `full?`)

Příklad: vyhodnocení aritmetických výrazů

Dobře uzávorkovaný výraz: `(op1 operator op2)`, kde `op1` a `op2` jsou také dobře uzávorkované výrazy, nebo čísla.

Př. `(1 + ((2 + 3) * 4))`

Algoritmus pro vyhodnocení takového výrazu. Máme dva zásobníky, `Q1` pro operátory, `Q2` pro čísla.

Čteme výraz zleva doprava

- `(` přeskočíme
- narazíme-li na číslo, vložíme jej do zásobníku `Q2`
- narazíme-li na operátor, vložíme jej do zásobníku `Q1`
- narazíme-li na `)`, potom odebere z `Q1` operátor a z `Q2` dva operandy, na ně operátor aplikujeme a výsledek vložíme do `Q2`

Po přečtení výrazu obsahuje `Q2` výsledek.

Fronta

Princip FIFO (first in, first out)

Hlavní operace

- enqueue: — vložení prvku
- dequeue: — odebrání prvku

Lze implementovat pomocí (dynamického) cyklického bufferu (`write`, `read`), případně obousměrného seznamu se sentinelem (přidáváme před a odebíráme za sentinelem).