Lineární datové struktury

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Prezentace ke dni 12. února 2020





Osnova přednášky

Lineární datové struktury

- Abstraktní datové struktury
- Zásobník
  - Popis
  - Implementace
  - Ukázka použití
- Fronta
  - Popis
  - Použití
  - Implementace

Datové struktury

- Algoritmy (počítačové programy) pracují s daty.
- Práce s daty obnáší manipulaci s daty – čtení, zápis, vyhledávání, agregace atd.
- Cílem je efektivní manipulace s daty.
- Mezi daty existují strukturální vazby – například balíček karet, první karta, poslední karta, předchozí karta, počet karet.
- **Datová struktura** – způsob uložení dat v počítači a možnosti manipulace s těmito daty.
- Specializace datových struktur na různé úlohy.
- Datové struktury jsou obsaženy v takřka každém programu.
- Efektivní datová struktura je obvykle základem pro efektivního algoritmu.

Abstraktní datové struktury (ADS)

- **Neformální definice**: Abstraktní datová struktura není závislá na vlastní implementaci a je definována pouze pomocí povolených operací na této struktuře.
- Formálně lze definovat jako matematický model.
- Nepřímá definice pomocí operací – neříkáme co to je, ale co s tím lze dělat.
- Nezajímá nás konkrétní implementace, ale rozhraní, které struktura poskytuje.
- Zjednodušují návrh algoritmů a přinášejí abstrakci do programování.
- Konkrétní implementace je označována pojmem datová struktura.
- Implementace je, v konečném důsledku, ale také důležitá.

Abstraktní datové struktury – dělení

- Počet prvků
  - neměnný – statická ADS, např. pole pevné velikosti,
  - proměnný – dynamická ADS, počet složek je proměnný, mezi operace patří vložení, odebrání prvku
- Datový typ prvků
  - homogenní – všechny prvky jsou shodného datového typu,
  - heterogenní – prvky jsou různého datového typu
- Existence jednoho, bezprostředního, následníka
  - lineární – existuje, např. pole, zásobník, fronta, seznam, ...
  - nelineární – neexistuje, např. strom, graf, ...

Abstraktní datové struktury – typičtí představitelé

Pole (Array, Vector) – obvykle dynamická velikost

Zásobník (Stack)

Fronta (Queue)

Seznam (List, Collection) – mnoho variant

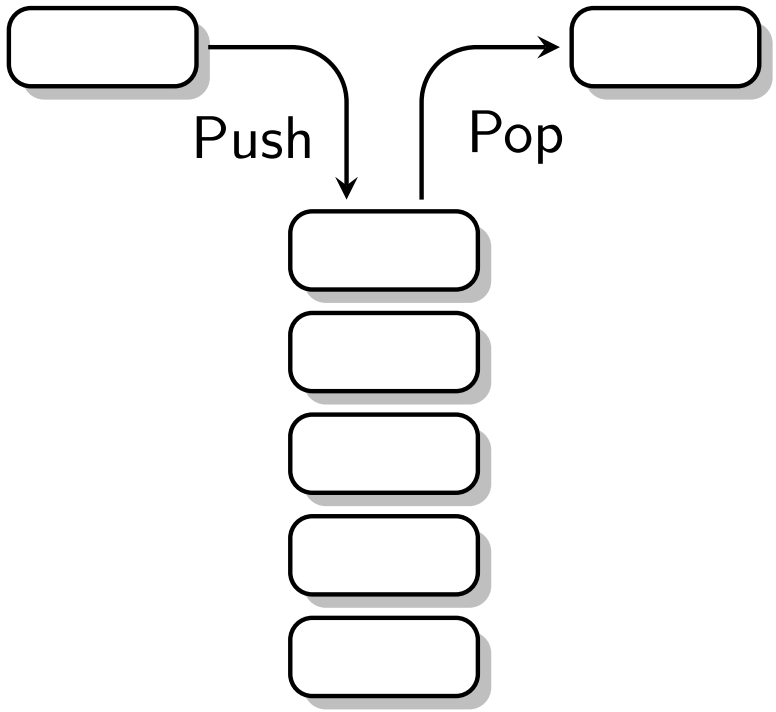
Slovník (Dictionary) – uložení dvojic (klíč, hodnota), setříděný vs. netříděný,

Množina (Set) – obdoba množiny z matematiky, setříděná vs. netříděná,

Graf (Graph) – reprezentace množiny prvků a vztahů mezi nimi, např. sociální síť

a další . . .

Zásobník (Stack)



Charakteristika

- princip **last-in, first-out**, **LIFO**
- prvek, který byl vložen poslední, je jako první ze zásobníku vyzvednut

Atributy

- prvky vkládáme na tzv. **vrchol zásobníku** (stack pointer). Ukazatel zásobníku není nutně ukazatel ve smyslu práce s dynamicky alokovanou pamětí.
- prvně vložený prvek se nazývá **dno zásobníku** (stack bottom)

Zásobník – operace

Základní operace

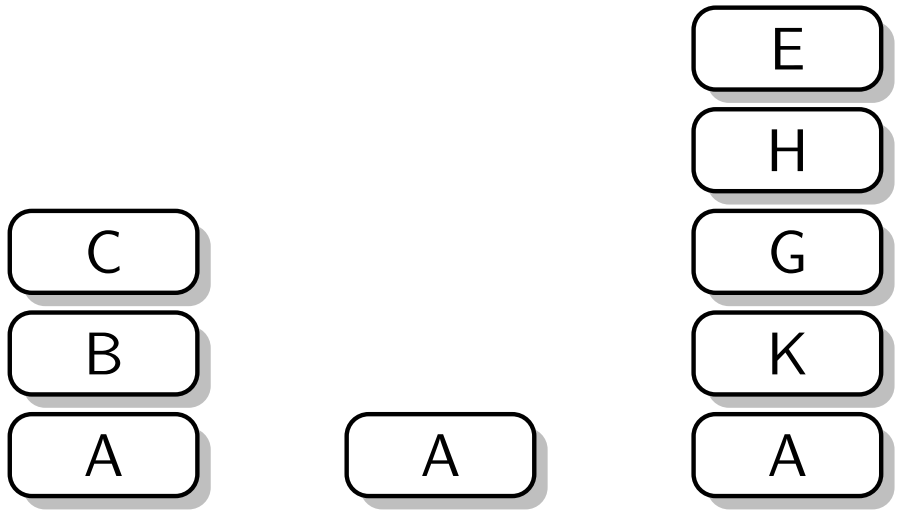
- Push – vložení prvku na vrchol zásobníku
- Pop – vyjmutí prvku z vrcholu zásobníku
- IsEmpty – test prázdnosti zásobníku
- Top – vrátí prvek z vrcholu zásobníku bez jeho vyjmutí

Další možné operace

- Init – inicializace zásobníku
- Clear – vyjmutí všech prvků ze zásobníku
- IsFull – test, zda je zásobník plný (pouze pro zásobník s omezenou kapacitou)

Správně implementované operace mají **konstantní** časovou složitost $O(1)$, tj. jejich časová složitost nezávisí na počtu prvků v zásobníku.

Zásobník




```
Push(A)
Push(B)
Push(C)
```

```
Pop()
Pop()
```

```
Push(K)
Push(G)
Push(H)
Push(E)
```

Zásobník – chybové stavy

- Pokud provedeme operaci Pop na prázdném zásobníku nastává tzv. **podtečení** (stack underflow).
- Pokud není možné přidat další prvek, nastává tzv. **přetečení** (stack overflow).

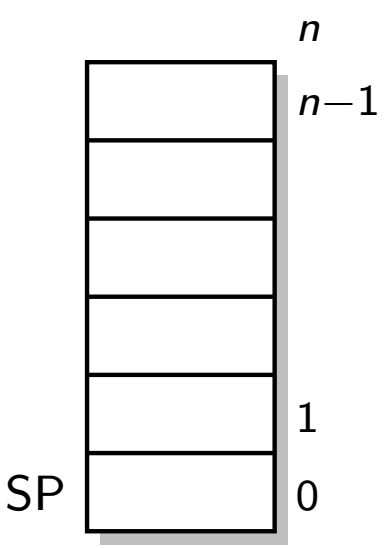
Zásobník – možnosti implementace

Pomocí pole

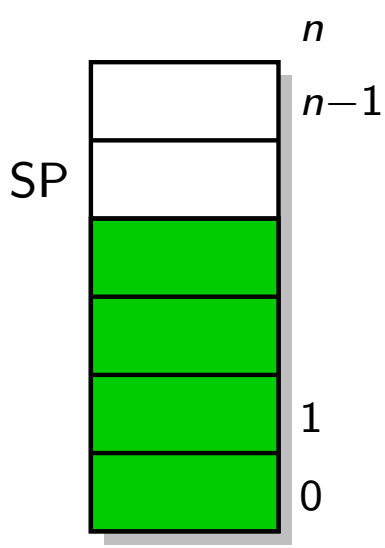
- pole délky n
- dno zásobníku na indexu 0
- stack pointer
  - index v poli, číslo typu `int`
  - neukazuje na poslední vložený prvek, nýbrž
  - ukazuje na **první volnou pozici** – dodržuje zvyklosti C++ o polích
  - prázdný zásobník – stack pointer roven 0
  - plný zásobník – stack pointer roven n

Pomocí spojových struktur – dynamická alokace paměti, pointery

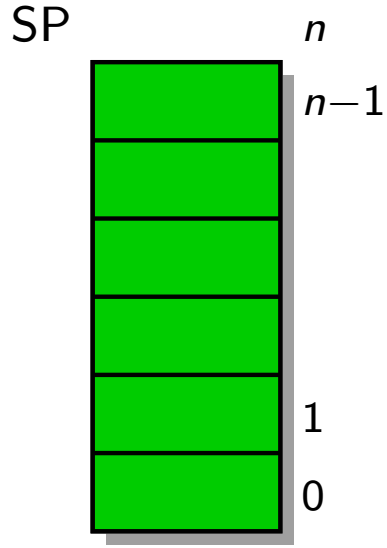
Zásobník – implementace pomocí pole



Prázdný



Částečně zaplněný



Plný

Zásobník – deklarace

```
1 const int StackSize = 100;
2
3 struct Stack
4 {
5    char Items[StackSize];
6
7    int StackPointer;
8 };
```

Zásobník – implementace

```
1  void Init(Stack& S)
2  {
3     S.StackPointer = 0;
4  }
5
6  void Push(Stack& S, const char X)
7  {
8     S.Items[S.StackPointer] = X;
9     S.StackPointer += 1;
10 }
11
12 char Pop(Stack& S)
13 {
14    S.StackPointer -= 1;
15    return S.Items[S.StackPointer];
16 }
17
18 char Top(const Stack& S)
19 {
```

Zásobník – implementace (pokrač.)

```
20    return S.Items[S.StackPointer - 1];
21 }
22
23 bool IsEmpty(const Stack& S)
24 {
25    return S.StackPointer == 0;
26 }
27
28 bool IsFull(const Stack& S)
29 {
30    return S.StackPointer == StackSize;
31 }
32
33 void Clear(Stack& S)
34 {
35    S.StackPointer = 0;
36 }
```

Zásobník – využití zásobníku

- volání funkcí (metod)
- vyhodnocování aritmetických výrazů
- odstranění rekurze
- zásobníkově orientované jazyky, například PostScript, PDF
- testování parity závorek, HTML/XML značek

Zásobník – testování parity závorek

Zadání

Máme dán matematický výraz `Expr`, který obsahuje kulaté, hranaté a složené závorky. Úkolem je implementovat funkci `CheckParenthesesParity`, která bude vracet `true` pokud je výraz správně uzávorkován, jinak bude vracet `false`.

Ukázka

Výraz `Expr`	Výsledek
`()`	`true`
`)(`	`false`
`(()())`	`true`
`[()()]`	`true`
`{()()()[()()]}`	`true`
`[()()`	`false`
`[](()]`	`false`
`[[()]`	`false`

Zásobník – testování parity závorek (pokrač.)

Princip řešení

- při průchodu řetězcem se musí pravé závorky objevovat v přesně opačném pořadí než levé závorky,
- levé závorky je nutné si pamatovat tak, aby aktuální, posledně načtená, levé závorka byla zpracována jako první; první načtená levá závorka musí přijít na řadu až jako poslední,
- zkontrolovaný pár závorek vyřadíme z dalšího zpracování,
- způsob pamatování levých závorek přesně odpovídá mechanismu práce zásobníku – LIFO.

Zásobník – testování parity závorek

Input : Výraz *Expr* obsahující závorky ()[]{}
Output: *true* pokud je *Expr* správně uzávorkován, jinak *false*

```
1  Init (S);
2  foreach e_i ∈ Expr do
3      switch e_i do
4          case (, [, { do
5              Push (S, e_i);
6          end
7          case ), ], } do
8              if ¬IsEmpty(S) then
9                  if Pop (S) „není správná závorka“ then
10                     return false;   // nesprávná pravá závorka
11                 end
12             end
13             else
14                 return false;           // málo levých závorek
15             end
16         end
17     end
18 end
19 return IsEmpty (S);   // nesmí zůstat žádná levá závorka
```

Zásobník – testování parity závorek

```
1  bool CheckParenthesesParity1(string Expr)
2  {
3     Stack S;
4     Init(S);
5     for (size_t i = 0; i < Expr.length(); i++)
6     {
7        switch (Expr[i])
8        {
9        case '(':
10       case '[':
11       case '{':
12          Push(S, Expr[i]);
13          break;
14       case ')':
15          if (!IsEmpty(S))
16          {
17             char ch = Pop(S);
18             if (ch != '(')
19                return false;
```

Zásobník – testování parity závorek (pokrač.)

```
20          }
21          else
22             return false;
23          break;
24       case ']':
25          if (!IsEmpty(S))
26          {
27             char ch = Pop(S);
28             if (ch != '[')
29                return false;
30          }
31          else
32             return false;
33          break;
34       case '}':
35          if (!IsEmpty(S))
36          {
37             char ch = Pop(S);
38             if (ch != '{')
```

Zásobník – testování parity závorek (pokrač.)

```
39                return false;
40            }
41            else
42               return false;
43            break;
44         }
45      }
46      return IsEmpty(S);
47   }
```

Zásobník – testování parity závorek

Opakování kódu

```
15 if (!IsEmpty(S))
16 {
17    char ch = Pop(S);
18    if (ch != '(')
19       return false;
20 }
21 else
22    return false;
```

Řešení

Do zásobníku neukládat nalezenou levou závorku, ale **očekávanou** pravou závorku.

Zásobník – testování parity závorek

```
1  bool CheckParenthesesParity2(string Expr)
2  {
3     Stack S;
4     Init(S);
5     for(size_t i = 0; i < Expr.length(); i++)
6     {
7        switch (Expr[i])
8        {
9        case '(':
10          Push(S, ')');
11          break;
12       case '[':
13          Push(S, ']');
14          break;
15       case '{':
16          Push(S, '}');
17          break;
18       case ')':
19       case ']':
```

Zásobník – testování parity závorek (pokrač.)

```
20          case '}':
21             if (!IsEmpty(S))
22             {
23                char ch = Pop(S);
24                if (ch != Expr[i])
25                   return false;
26             }
27             else
28                return false;
29             break;
30          }
31       }
32       return IsEmpty(S);
33    }
```

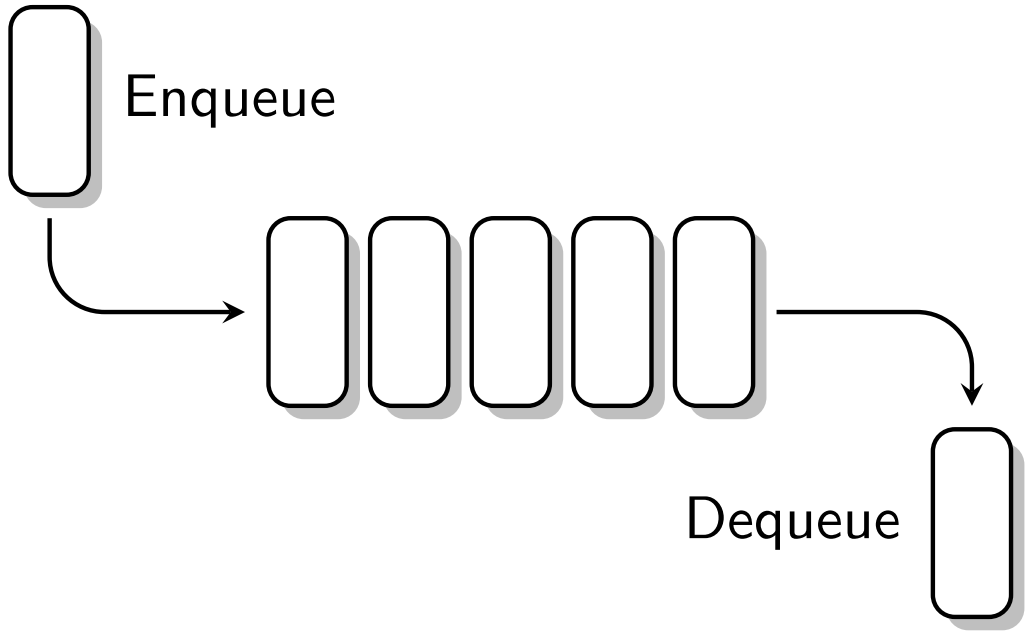
Zásobník – otázky implementace

- Jak mít v kódu (jedné funkci) více zásobníků?
- Jak zabránit, aby mohl kdokoliv v zásobníku něco měnit? Ochrana vnitřních dat.
- Lze implementovat zásobník, který by se dynamicky zvětšoval?
- Nešlo by vyrobit zásobník jako „součástku“? Třeba udělat z něj datový typ s definovanými operacemi?
- Potřebuji mít v programu zásobník čísel a znaků. To ho musím naimplementovat dvakrát? Nešlo by mít zásobník jako šablonu?

Fronta (Queue)

Charakteristika

- princip **first-in, first-out**, **FIFO**
- prvek, který byl vložen první, je také jako první z fronty vyzvednut



Atributy

- první prvek se nazývá **hlava** fronty (**head**),
- poslední prvek se nazývá **ocas** fronty (**tail**).

Fronta – operace

Základní operace

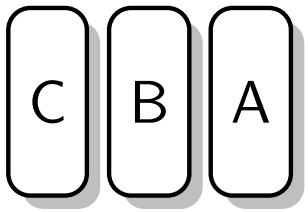
- Enqueue – vložení prvku na konec fronty
- Dequeue – vyjmutí prvku ze začátku fronty
- Peek – vrátí prvek ze začátku fronty bez jeho vyjmutí
- IsEmpty – test zda je fronta prázdná

Další možné operace

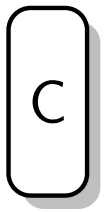
- Init – inicializace fronty
- Clear – vyjmutí všech prvků z fronty
- IsFull – test, zda je fronta zaplněna (pouze u fronty s omezenou kapacitou)

Správně implementované operace mají konstantní časovou složitost $O(1)$, tj. jejich časová složitost nezávisí na počtu prvků ve frontě.

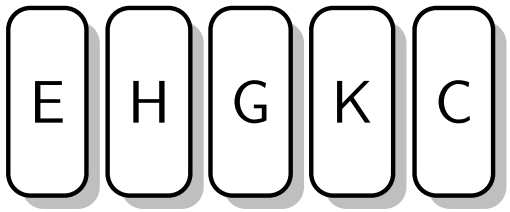
Fronta




```
Enqueue(A)
Enqueue(B)
Enqueue(C)
```




```
Dequeque()
Dequeque()
```




```
Enqueue(K)
Enqueue(G)
Enqueue(H)
Enqueue(E)
```

Fronta – chybové stavy

- Pokud provedeme operaci Dequeue na prázdné frontě nastává tzv. **podtečení** (queue underflow).
- Pokud není možné přidat další prvek, nastává tzv. **přetečení** (queue overflow).

Fronta – využití

- tisková fronta u sdílené tiskárny
- plánovač v operačním systému (více běžících procesů na jednoprocesorovém počítači $\Rightarrow$ procesy se musí střídat)
- obsluha uživatelů na serverech obecně

Fronta – plánovač OS, cyklická obsluha (round robin)

Tato část OS přiděluje jednotlivým procesům sdílený procesor:

- vybere z fronty čekajících procesů první proces,
- obnoví registry procesoru,
- spustí obnovený proces,
- po uplynutí vymezeného časového kvanta, zahájí odebírání procesoru,
- uloží obsah registrů
- zařadí tento proces na konec fronty,
- vybere z fronty čekajících procesů první proces . . .

Fronta – možnosti implementace

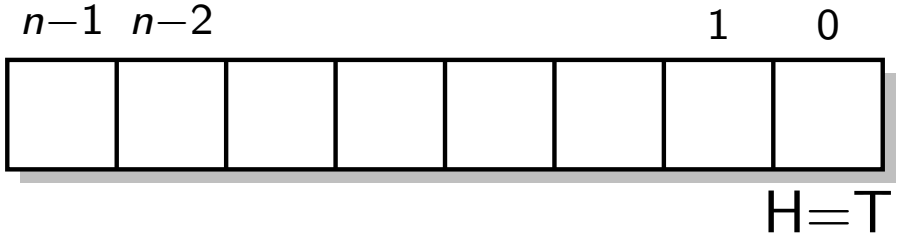
Pomocí spojových struktur – dynamická alokace paměti, pointery

Pomocí pole

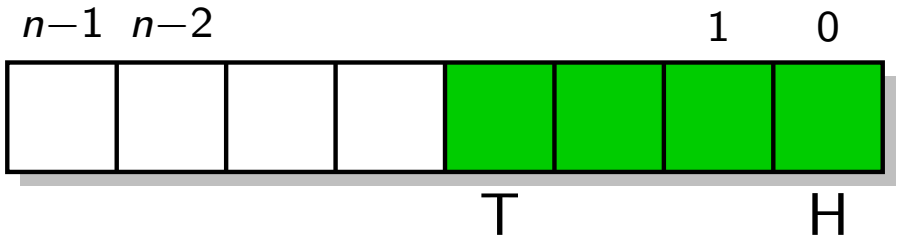
- hlava a ocas fronty jsou indexy do pole,
- jednoduché přidávání a odebírání – jak hlava, tak ocas se posunují dozadu.

Fronta – implementace v poli

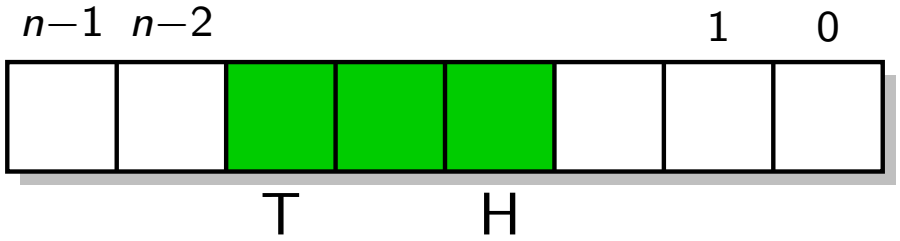
Prázdná fronta



Vloženy 4 prvky

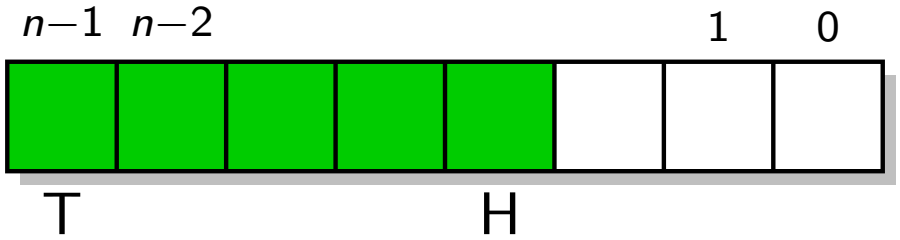


Vyjmuty 3 prvky, vloženy 2 prvky



Fronta – implementace v poli (pokrač.)

Vloženy 2 prvky



Výsledný stav:

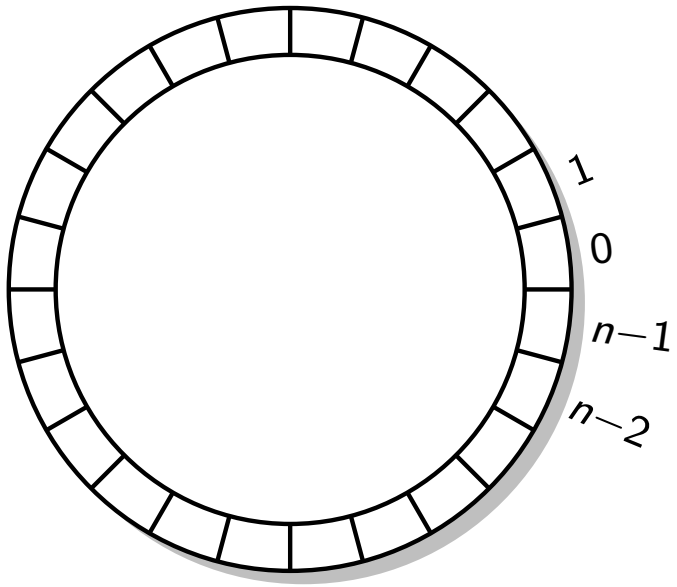
- do fronty již nelze přidat další prvky,
- ale současně fronta není plná.

Možná řešení:

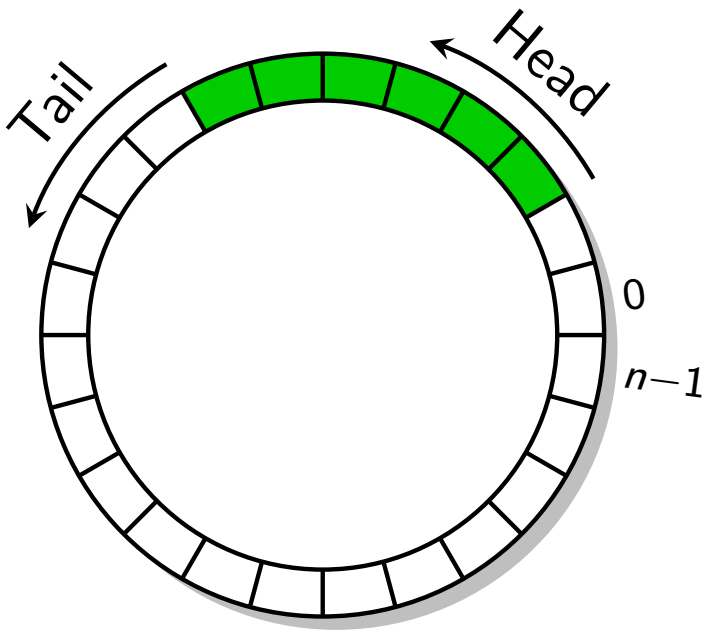
- přesouvat prvky na začátek pole – vyjmutí prvku již nebude operace s časovou složitostí $O(1)$ ale $O(n)$, bude docházet k masivnímu kopírování dat,

Fronta – kruhový buffer

Řešením je **kruhový buffer** (circular buffer) – začátek a konec pole se navzájem dotýkají.



Fronta – kruhový buffer



Charakteristika

- při vložení prvku se ocas fronty zvyšuje o jedničku,
- při odebírání prvku se hlava fronty zvyšuje o jedničku
- při překročení délky bufferu se hodnoty hlavy či ocasu vrací na první pozici tj. 0 – **modulární aritmetika**

Problém: Jak rozlišit prázdnou a zaplněnou frontu?
Řešení: V buffer je udržováno jedno prázdné místo. Ocas fronty ukazuje na první volnou pozici, ne na konec fronty.

Fronta – deklarace

```
1  const int CircularBufferSize = 5;
2
3  const int MaxQueueSize = CircularBufferSize - 1;
4
5  struct Queue
6  {
7     int Items[CircularBufferSize];
8
9     int Head;
10
11    int Tail;
12 };
```

Fronta – implementace

```
1  void Init(Queue& Q)
2  {
3     Q.Head = 0;
4     Q.Tail = 0;
5  }
6
7  void Clear(Queue& Q)
8  {
9     Q.Head = 0;
10    Q.Tail = 0;
11 }
12
13 void Enqueue(Queue& Q, const int X)
14 {
15    Q.Items[Q.Tail] = X;
16    Q.Tail = (Q.Tail + 1) % CircularBufferSize;
17 }
18
19 int Dequeue(Queue& Q)
```

Fronta – implementace (pokrač.)

```
20 {
21    int x = Q.Items[Q.Head];
22    Q.Head = (Q.Head + 1) % CircularBufferSize;
23    return x;
24 }
25
26 int Peek(const Queue& Q)
27 {
28    return Q.Items[Q.Head];
29 }
30
31 bool IsEmpty(const Queue& Q)
32 {
33    return Q.Head == Q.Tail;
34 }
35
36 bool IsFull(const Queue& Q)
37 {
38    return Q.Head == (Q.Tail + 1) % CircularBufferSize;
```

Fronta – implementace (pokrač.)

```
39 }
```

Kontrolní otázky

1. Vysvětlete co znamená, že zásobník představuje paměť typu LIFO.
2. Co je to vrchol zásobníku? Co je to dno zásobníku?
3. Jaká je teoreticky kapacita zásobníku?
4. Co je to přetečení zásobníku? Co je to podtečení zásobníku?
5. Jak se typicky nazývá operace vložení prvku do zásobníku? Popište tuto operaci.
6. Jak se typicky nazývá operace vyjmutí prvku ze zásobníku? Popište tuto operaci.
7. Závisí složitost operací na počtu prvků v zásobníku? Ano či ne?
8. Uveďte příklady použití zásobníku.
9. Do zásobníku vložím prvky v pořadí A, B, C, D. Potom je budu ze zásobníku vyjímat. Jakou dostanu posloupnost?
10. Vysvětlete co znamená, že fronta představuje paměť typu FIFO.

Kontrolní otázky (pokrač.)

11. Co je to hlava fronty? Co je to ocas fronty?
12. Závisí složitost operací na počtu prvků ve frontě? Ano či ne?
13. Co je to přetečení fronty? Co je to podtečení fronty?
14. Do fronty vložím prvky v pořadí A, B, C, D. Potom je budu z fronty vyjímat. Jakou dostanu posloupnost?
15. Jak se typicky nazývá operace vložení prvku do fronty? Popište tuto operaci.
16. Jak se typicky nazývá operace vyjmutí prvku ze fronty? Popište tuto operaci.
17. Uveďte příklady použití fronty.
18. Popište hlavní problémy při implementaci fronty v poli. Jak je lze řešit pomocí kruhové reprezentace fronty?
19. Jaký je rozdíl mezi frontou, zásobníkem a seznamem? Zaměřte se na operace vkládání a vyjímání prvků z těchto struktur.

Kontrolní otázky (pokrač.)

20. Co je to hlava seznamu, ocas seznamu?
21. Rozdíl mezi jednosměrným, obousměrným a kruhovým seznamem.
22. Popište algoritmus vyhledávání daného prvku v seznamu?
23. Lze využít algoritmus půlení intervalu pro vyhledávání v seznamu? Jaký je tu hlavní problém?
24. Co je to garbage collector?
25. Co je to iterátor?

Děkuji za pozornost