データ構造
with Haskell

目次

- Stack
- Queue
- Deque

目次

- Stack
- Queue
- Deque

- 全部配列を使えば一瞬じゃないか！
  いいかげんにしろ！

残念ながら

配列なんて無いよ

Haskellのデータ構造

- 配列は一応あるよ
  - ただし代入に O(N) かかる

Haskellのデータ構造

- リストというのを使います
- 次の操作が可能
  - 先頭に値を追加 O(1)
  - 先頭から削除　O(1)
  - n番目にアクセス O(n)

- 厳密にはちょっと違うけどだいたいあってる

Haskellのリスト

- [3, 1, 4, 1, 5]　--リストの例
- 追加
  - 9 : [3, 1, 4, 1, 5] → [9, 3, 1, 4, 1, 5]
- 先頭を得る
  - head [3, 1, 4, 1, 5] → 3
- 先頭以外を得る
  - tail [3, 1, 4, 1, 5] → [1, 4, 1, 5]
  - tail [1] → [ ] -- 空のリスト

Haskellのリスト

- 先頭にくっつける (:)
- 先頭を参照する head
- 先頭以外を参照する tail

- これらは以降たくさん出てくるのでよく覚えていてね

1つめのデータ構造

Stack

Stack

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の先頭から値を取り出す（pop）

Stack:図示

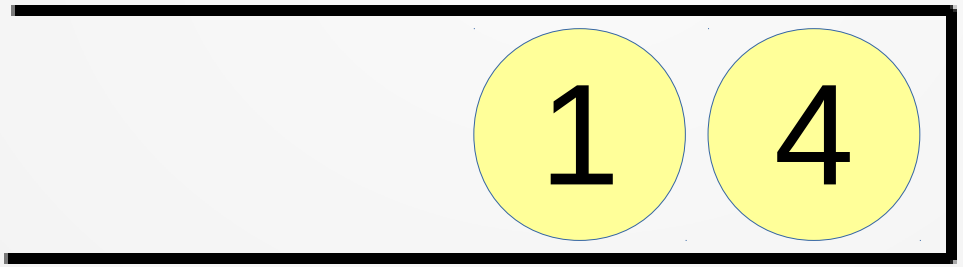
- 空のStackがあるじゃろ
  - 左側が先頭

Stack:図示

- push(4)

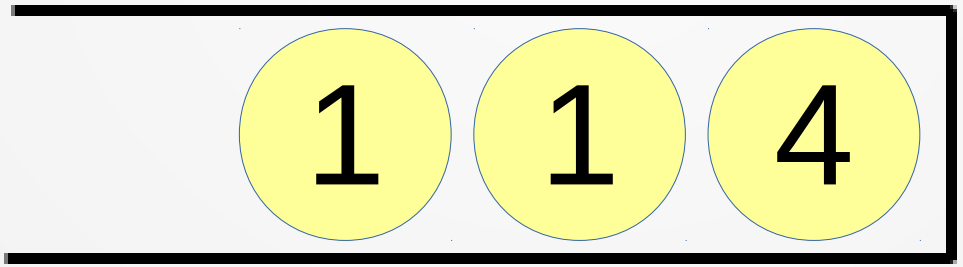
Stack:図示

- push(4), push(1)



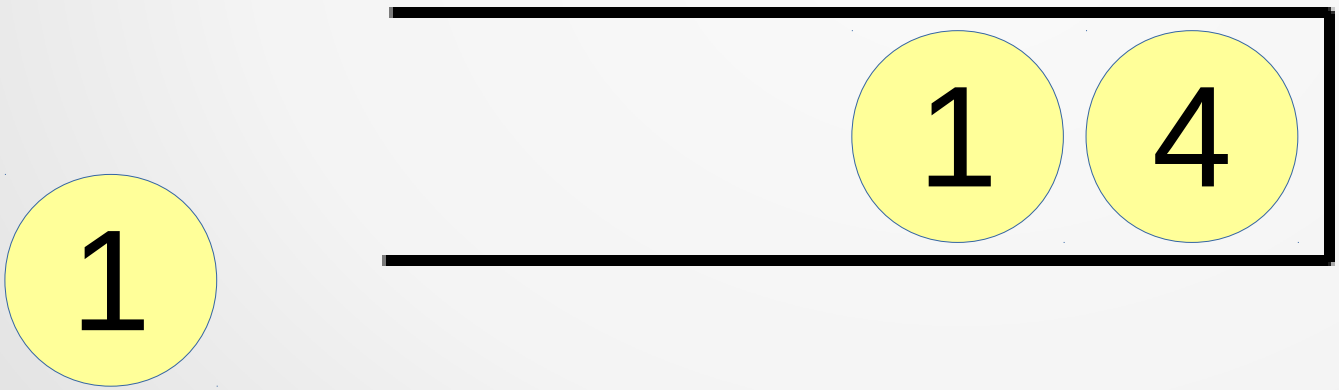
Stack:図示

- push(4), push(1), push(1)



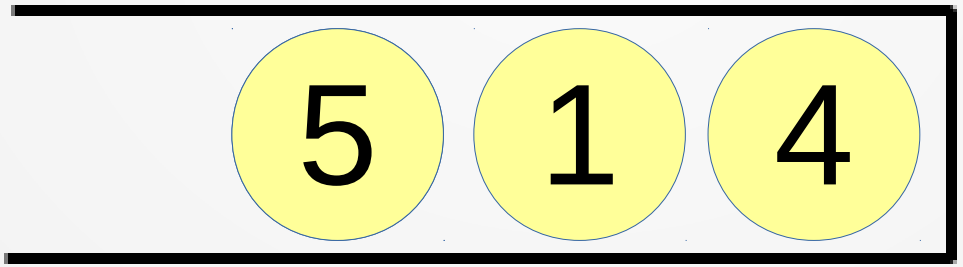
Stack:図示

- push(4), push(1), push(1), pop()



Stack:図示

- push(4), push(1), push(1), pop(), push(5)



Stack

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の先頭から値を取り出す（pop）

Stack

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の先頭から値を取り出す（pop）

- すごい既視感

Haskellのデータ構造（再掲）

- リストというのを使います
- 次の操作が可能
  - 先頭に値を追加 O(1)
  - 先頭から削除　O(1)
  - n番目にアクセス O(n)

- 厳密にはちょっと違うけどだいたいあってる

Haskellのデータ構造（再掲）

- リストというのを使います
- 次の操作が可能
  - 先頭に値を追加 O(1)
  - 先頭から削除　O(1)
  - n番目にアクセス O(n)

- 厳密にはちょっと違うけどだいたいあってる

Stack

- スタック
- リストをそのまま使える
  - やったぜ

Stack: 実装

(ソースコードをここに書く)

(説明をここに書く)

Stack: 実装

```
Data Stack a = S [a]
```

Stack型の定義

Stack: 実装

```
Data Stack a = S [a]

push (S xs) x = S (x:xs)
```

pushの定義

Stack: 実装

```
Data Stack a = S [a]

push (S xs) x = S (x:xs)

pop (x:xs) = (x, S xs)
```

popの定義

Stack: 実装

```
Data Stack a = S [a]

push (S xs) x = S (x:xs)

pop (x:xs) = (x, S xs)
```

空のStackをpopしないことは
良心にまかせる

Stack: 実装

```
Data Stack a = S [a]

push (S xs) x = S (x:xs)

pop (x:xs) = (x, S xs)
```

完成

1つめのデータ構造

Queue

Queue

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の末尾から値を取り出す（pop）

Queue: 図示

- 空のQueueがあるじゃろ
  - 左側から入って、右側から出る

Queue: 図示

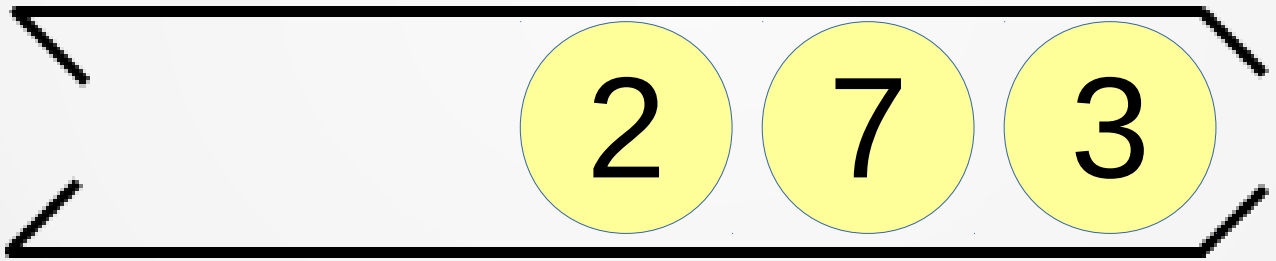
- push(3)

Queue: 図示

- push(3), push(7)

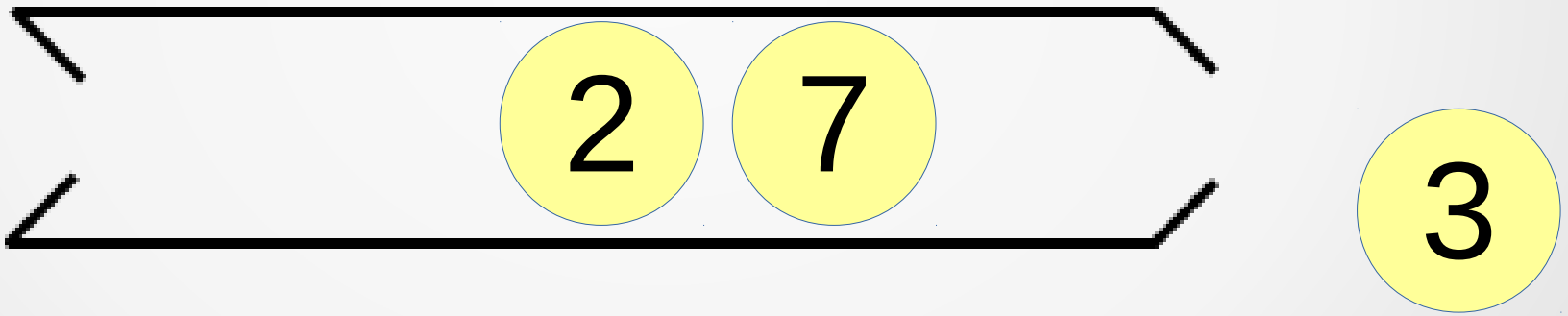
Queue: 图示

- push(3), push(7), push(2)



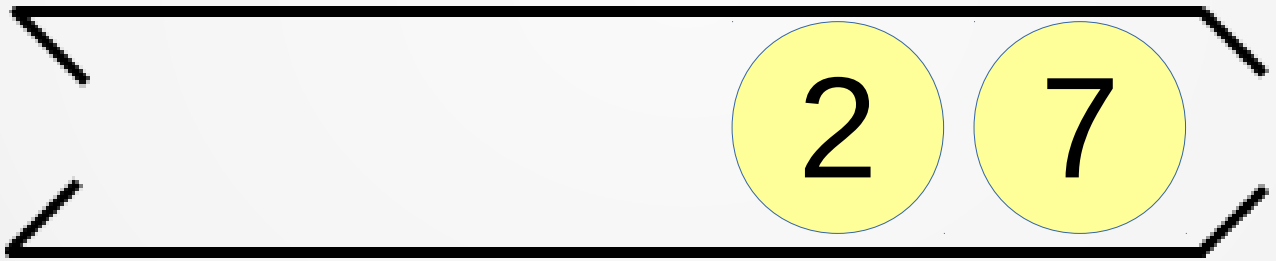
Queue: 图示

- push(3), push(7), push(2), pop()



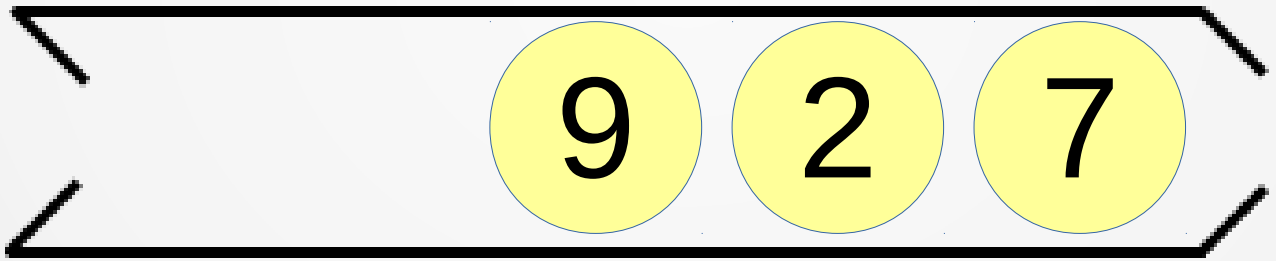
Queue: 图示

- push(3), push(7), push(2), pop()



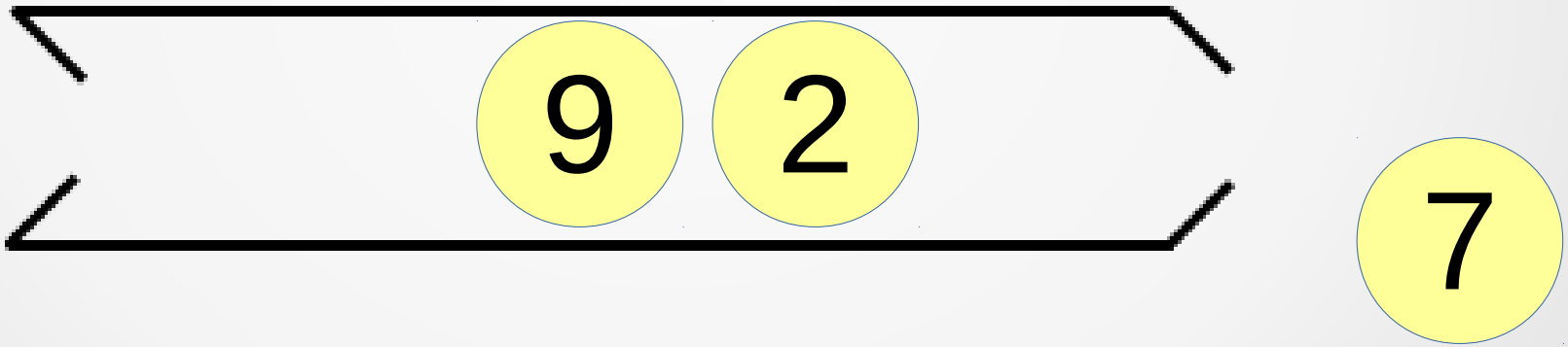
Queue: 图示

- push(3), push(7), push(2), pop(), push(9)



Queue: 図示

- push(3), push(7), push(2), pop(), push(9) pop()



Queue

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の末尾から値を取り出す（pop）

Queue

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の末尾から値を取り出す（pop）
  - リストの末尾の操作は厄介

Queue

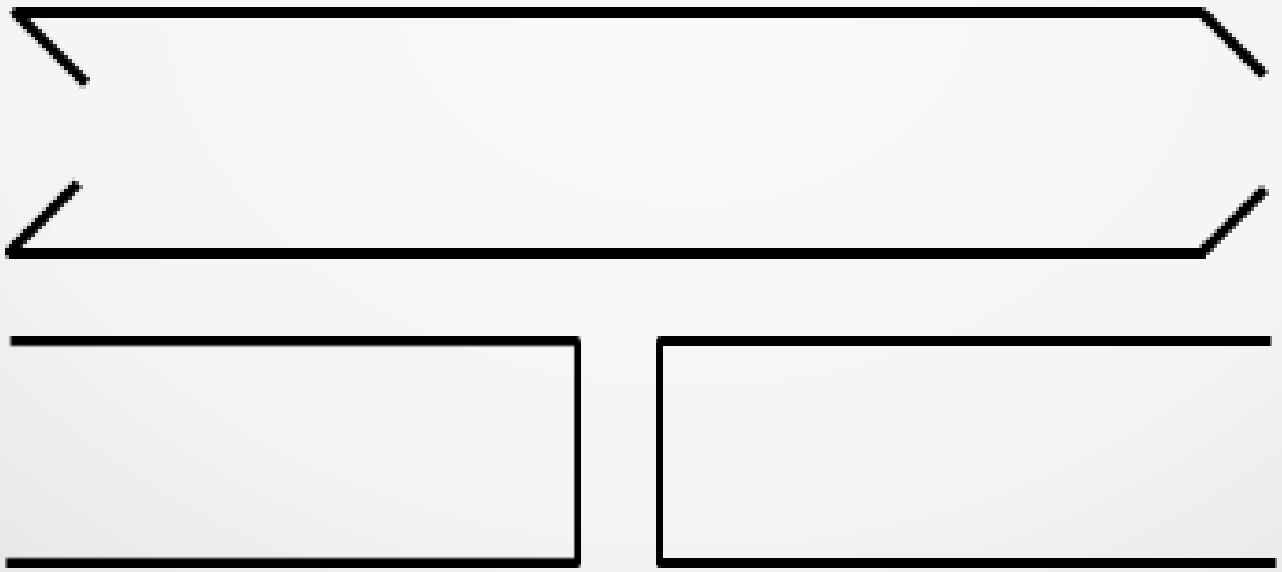
- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（push）
  - 列の末尾から値を取り出す（pop）
  - リストの末尾の操作は厄介
  - しかしデータ構造の申し子である皆さんなら対処法をすぐに思いつくはず

有名なことわざ

QueueはStack2つで作れる

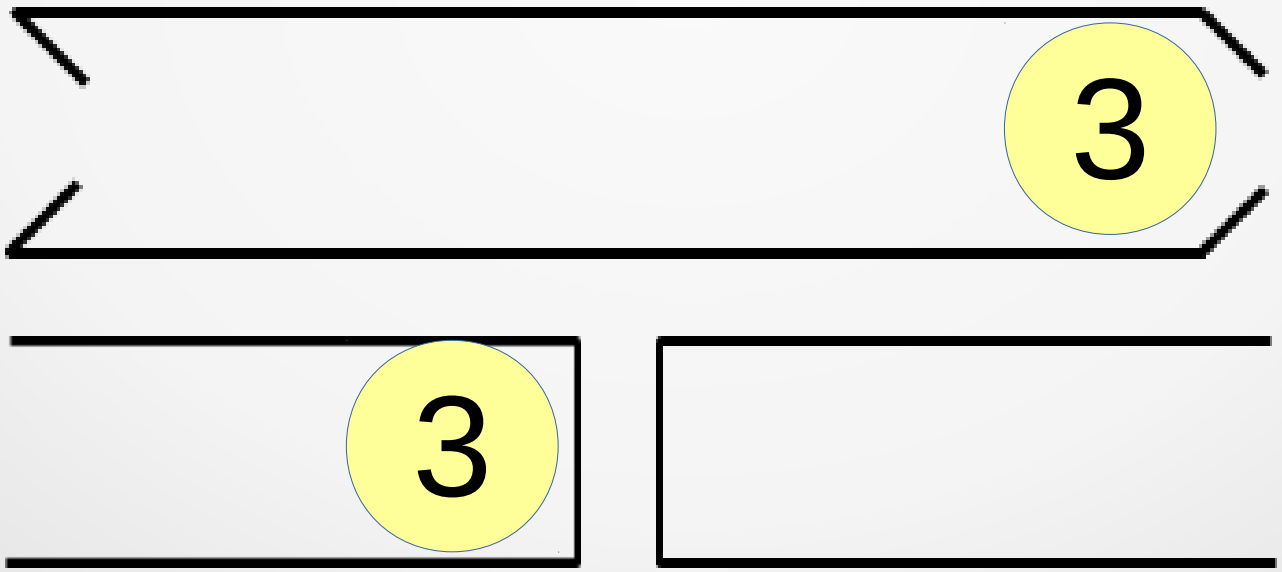
Queue: 図示

- 空のQueueの下に2つのStackがあるじゃろ



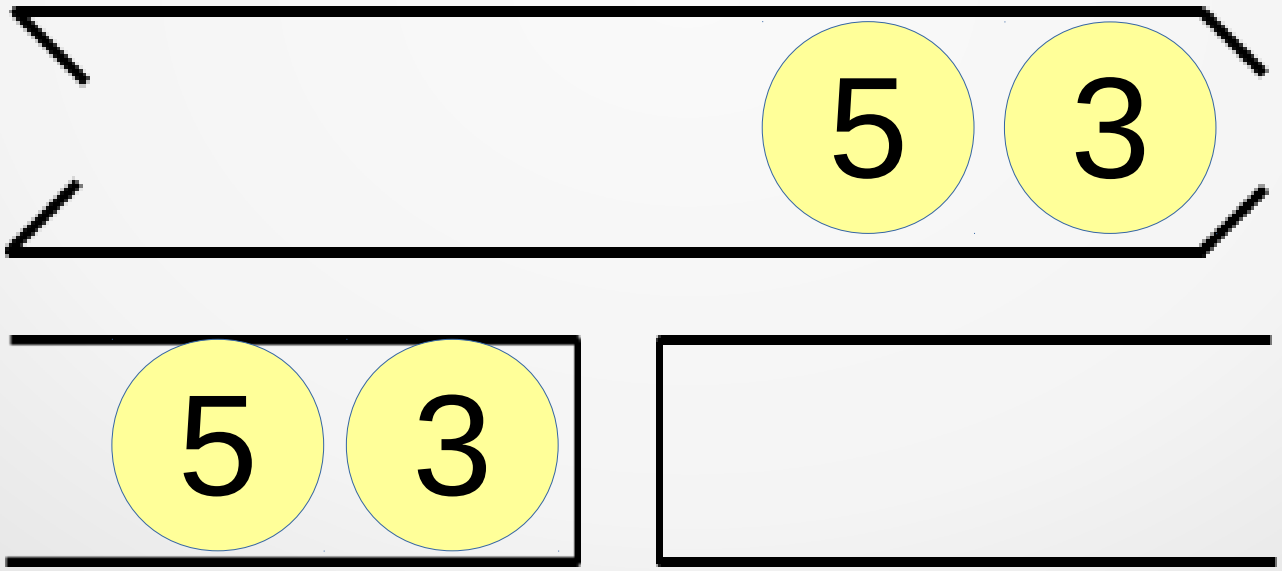
Queue: 図示

- push(3)



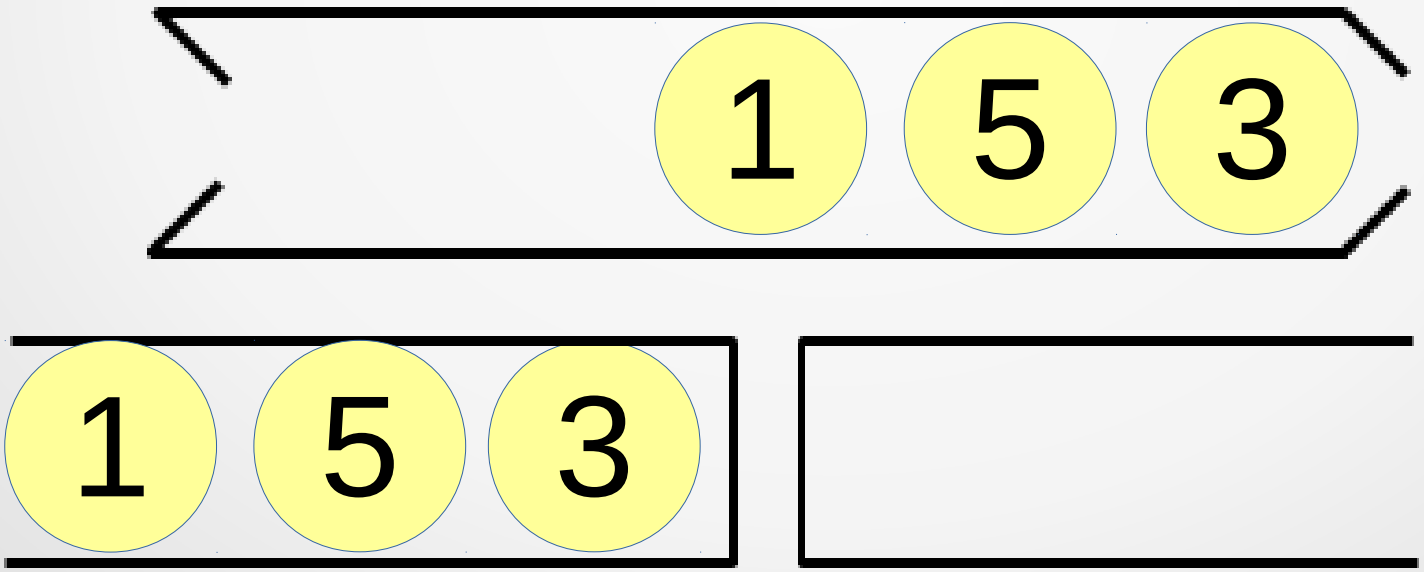
Queue: 図示

- push(3), push(5)



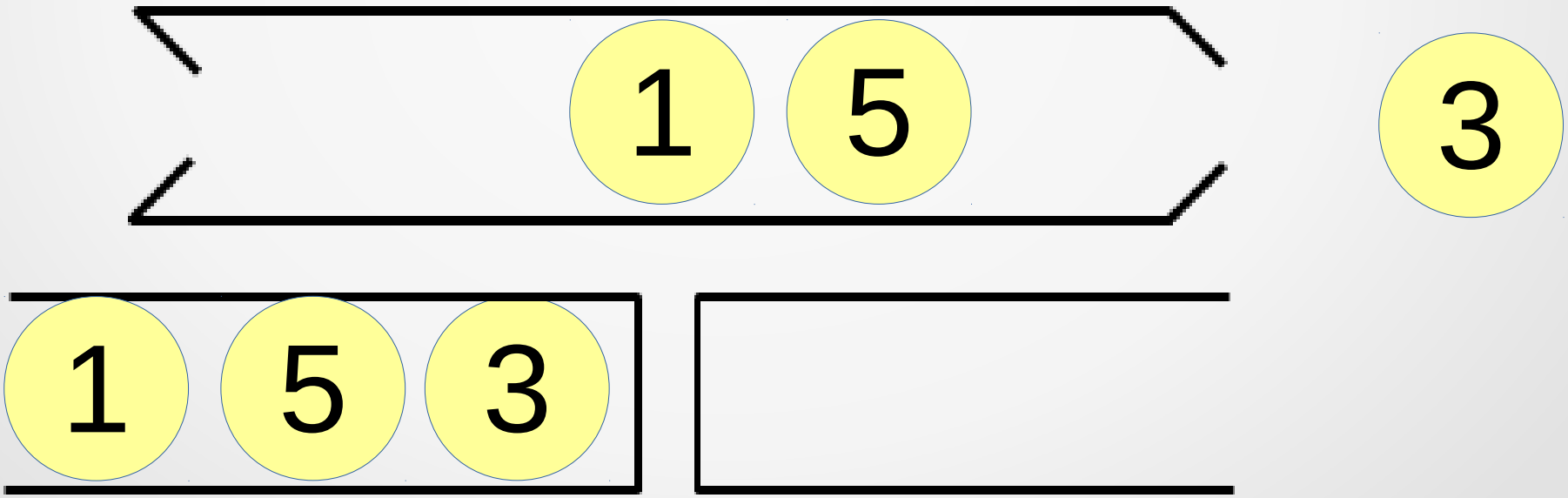
Queue: 図示

- push(3), push(5), push(1)



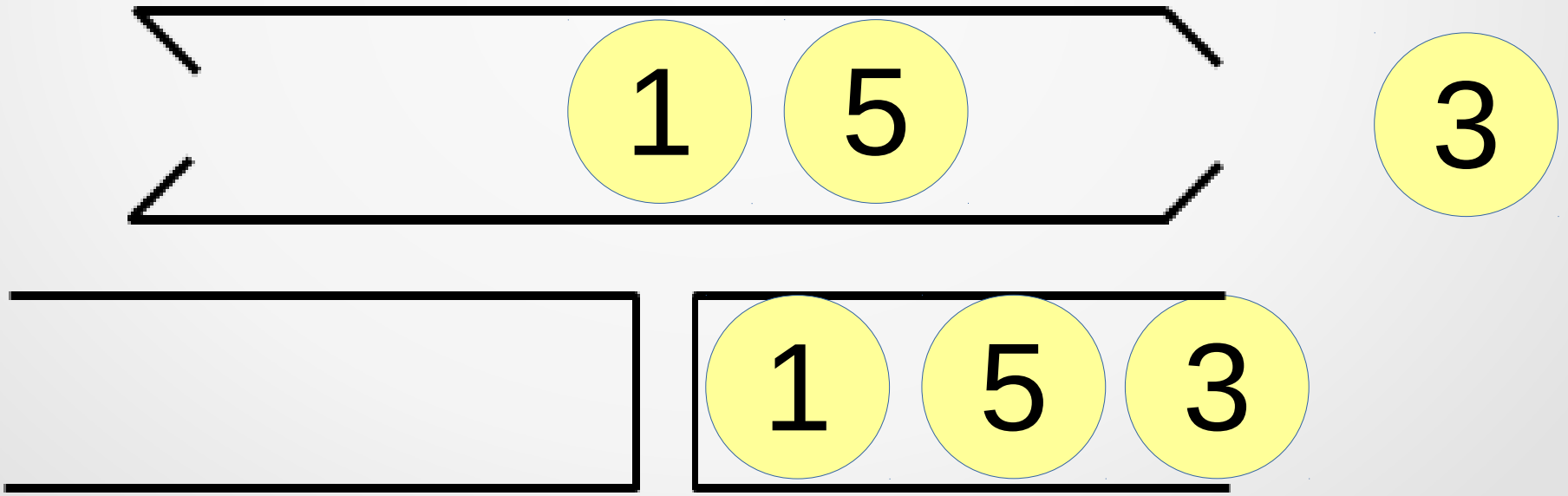
Queue: 図示

- push(3), push(5), push(1), pop()



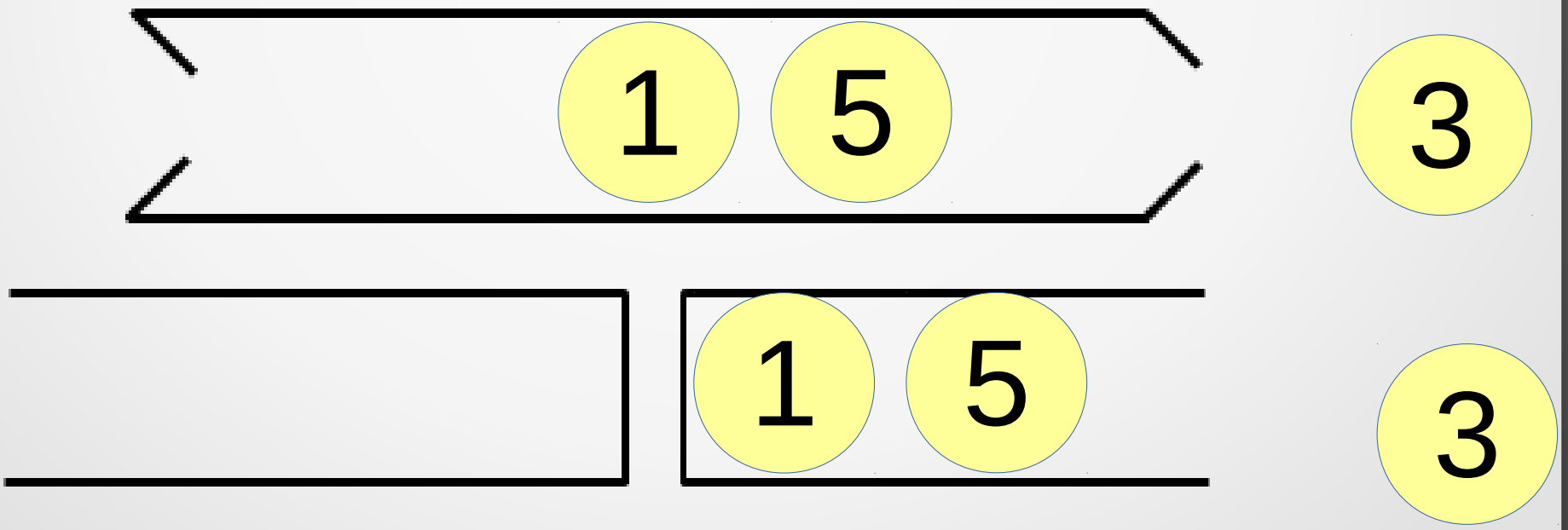
Queue: 図示

- push(3), push(5), push(1), pop()
- pop時右側が空だったらｴｲﾔｯする



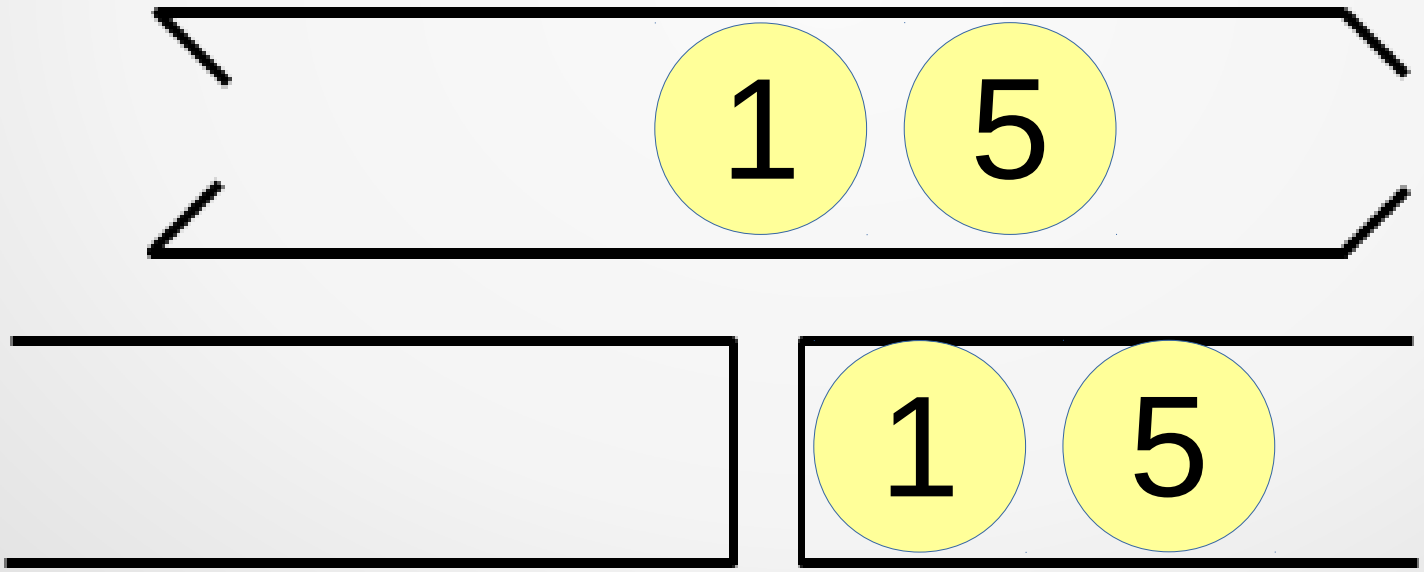
Queue: 図示

- push(3), push(5), push(1), pop()



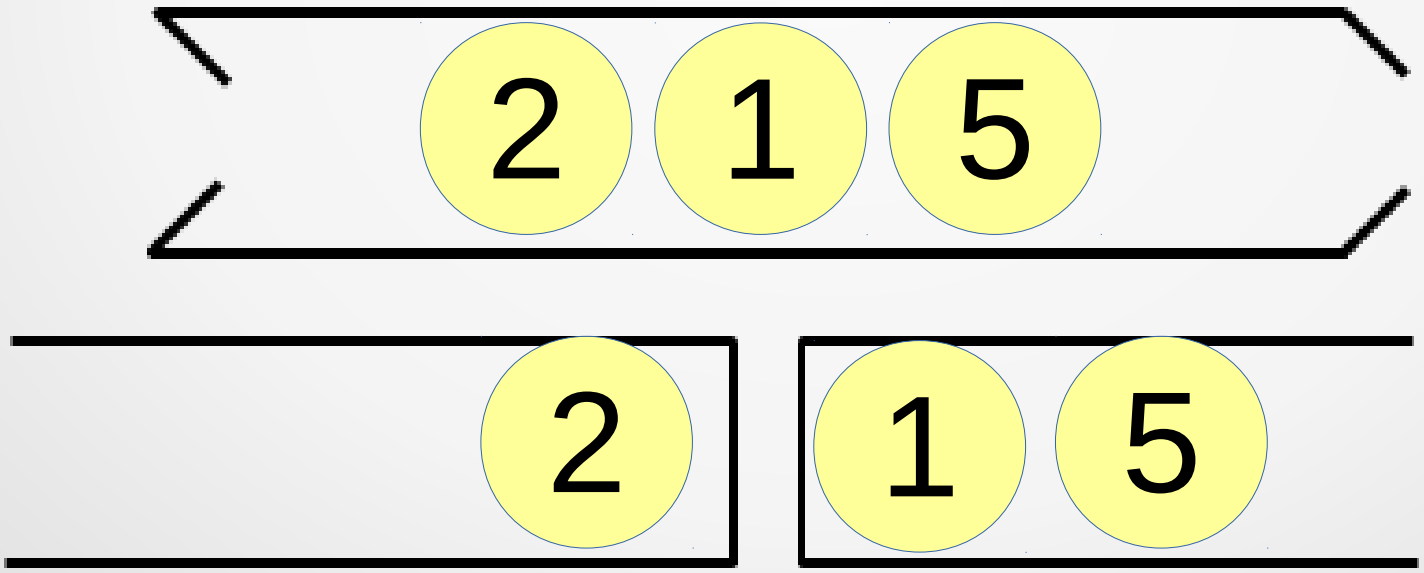
Queue: 図示

- push(3), push(5), push(1), pop()



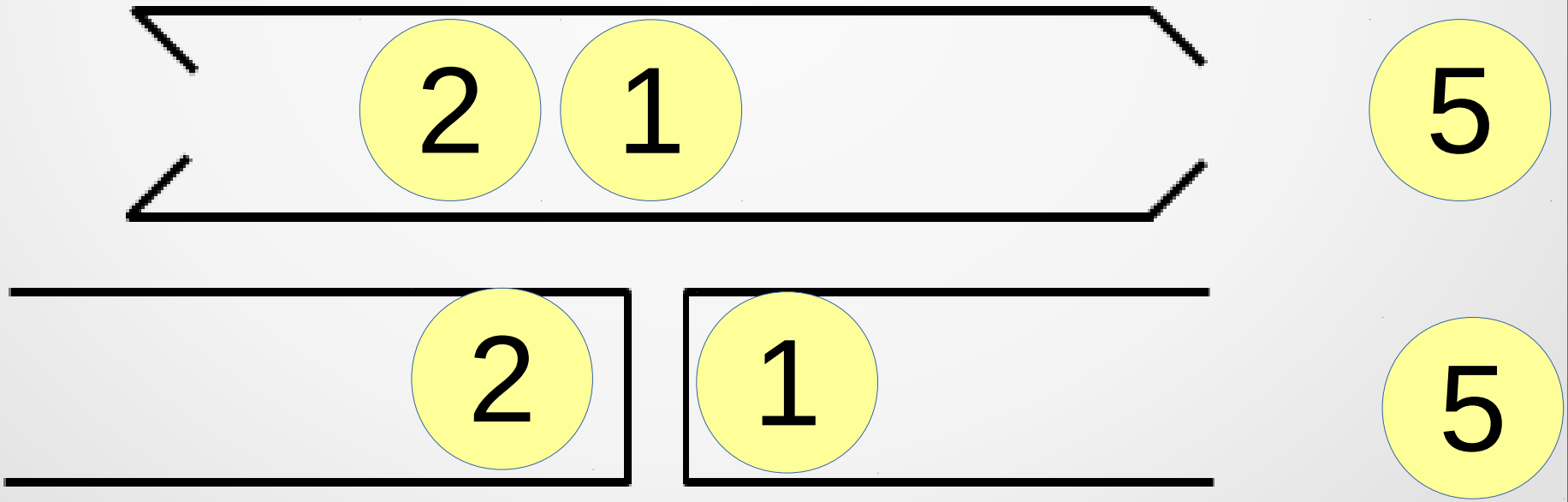
Queue: 图示

- push(3), push(5), push(1), pop(), push(2)



Queue: 図示

- push(3), push(5), push(1), pop(), push(2), pop()

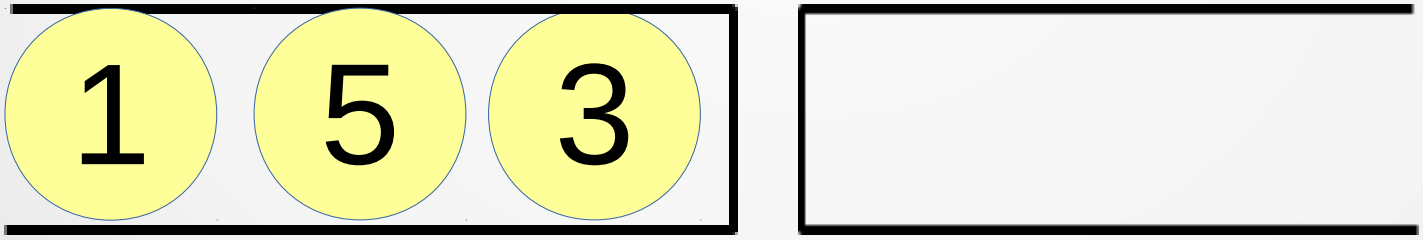


Queue(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！
- 一番ややこしいのはpopするときに右側のStackが空だった時

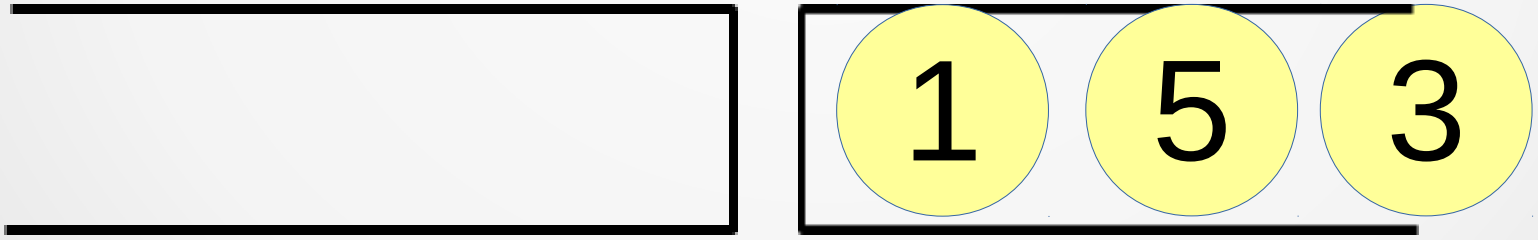
Queue(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！
- 一番ややこしいのはpopするときに右側のStackが空だった時



Queue(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！
- 一番ややこしいのはpopするときに右側のStackが空だった時



エイヤッ

Queue(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！
- 一番ややこしいのはpopするときに右側のStackが空だった時
- Stackがただのリストであったことを思い出すと、エイヤッはただのリスト反転
  - 標準で reverseという関数がある

Queue: 実装

(実装をここに書く)

(説明をここに書く)

Queue: 実装

```
Queue a = Q [a] [a]
```

Queueの型を定義　リスト2つ

Queue: 実装

```
Queue a = Q [a] [a]

push (Q xs ys) x = Q (x:xs) ys
```

push の実装 (Stackと同様)

Queue: 実装

```
Queue a = Q [a] [a]

push (Q xs ys) x = Q (x:xs) ys

pop (Q xs y:ys) = (y (Q xs ys))
```

右側が空じゃない時のpop

Queue: 実装

```
Queue a = Q [a] [a]

push (Q xs ys) x = Q (x:xs) ys

pop (Q xs y:ys) = (y (Q xs ys))
pop (Q xs []) =
          pop(Q [] (reverse xs))
```

右側が空の時のpop

3つめのデータ構造

Deque

Deque

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　　（pushF）
  - 列の末尾に値を追加　　（pushB）
  - 列の先頭から値を取り出す（popF）
  - 列の末尾から値を取り出す（popB）

Deque: 図示

- ここに空のDequeがあるじゃろ
  - 左が先頭(Front), 右が末尾 (Back)

Deque: 図示

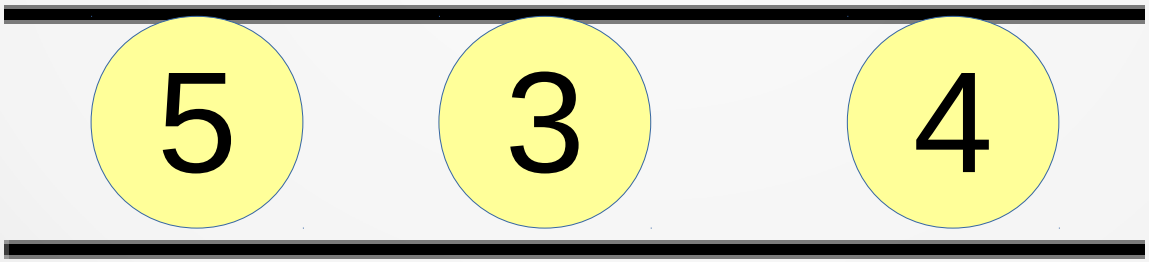
- pushF(3)

Deque: 図示

- pushF(3), pushB(4)

Deque: 図示

- pushF(3), pushB(4), pushF(5)

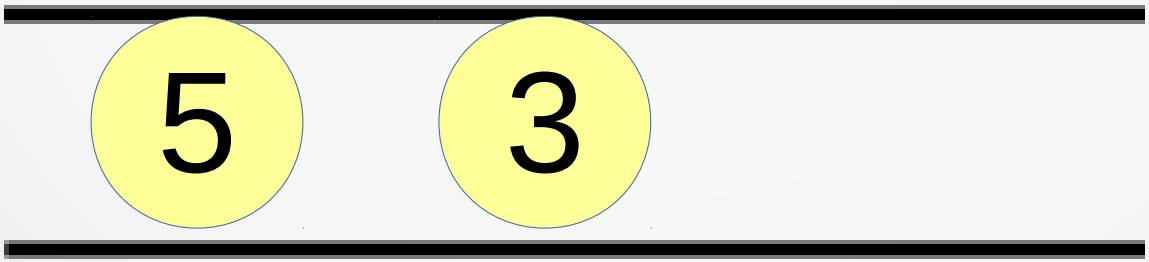


Deque: 図示

- pushF(3), pushB(4), pushF(5), popB()

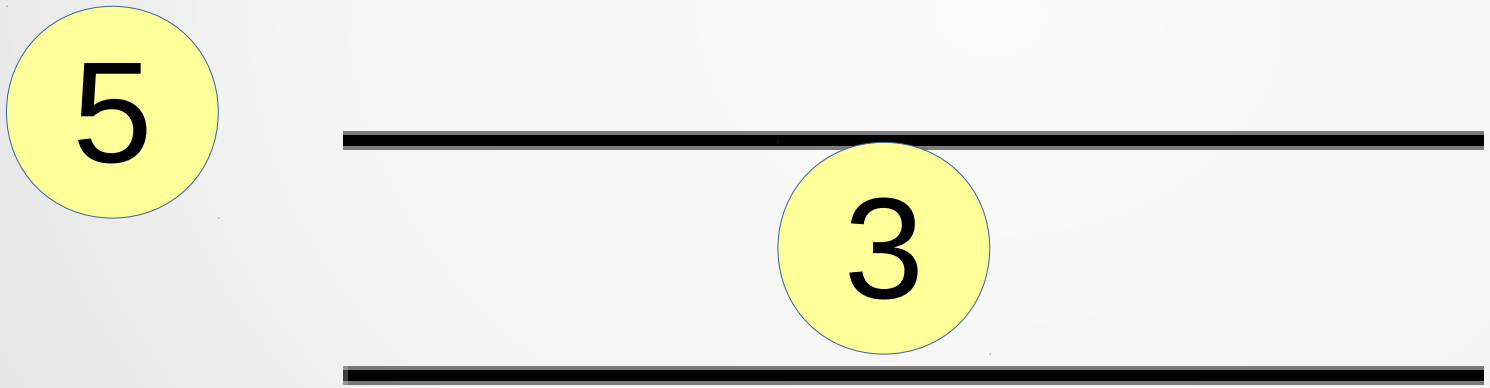
Deque: 図示

- pushF(3), pushB(4), pushF(5), popB()



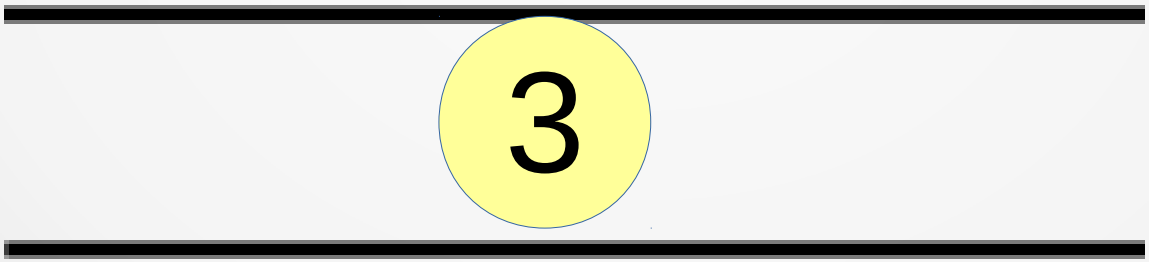
Deque: 図示

- pushF(3), pushB(4), pushF(5), popB(), popF()



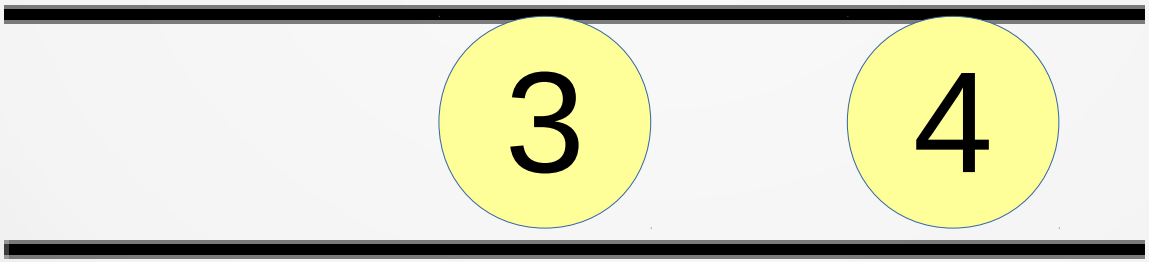
Deque: 図示

- pushF(3), pushB(4), pushF(5), popB(), popF()



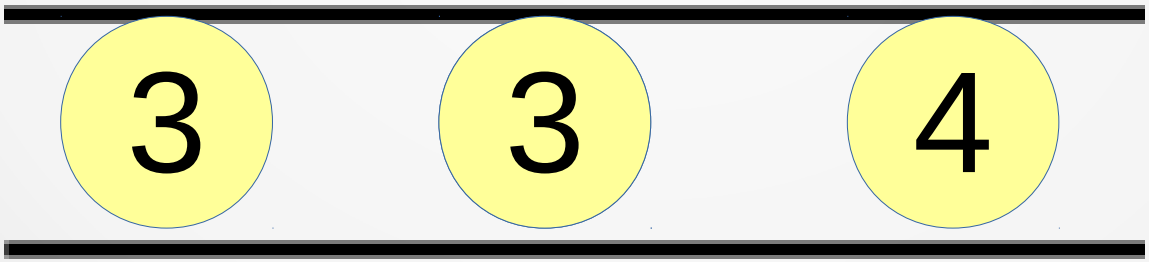
Deque: 図示

- pushF(3), pushB(4), pushF(5), popB(), popF()
  pushB(4)



Deque: 図示

- pushF(3), pushB(4), pushF(5), popB(), popF()
  pushB(4), pushF(3)



Deque

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　（pushF）
  - 列の末尾に値を追加　（pushB）
  - 列の先頭から値を取り出す（popF）
  - 列の末尾から値を取り出す（popB）

Deque

- 次の操作に対応したデータ構造
  - 列の先頭に値を追加　（pushF）
  - 列の末尾に値を追加　（pushB）
  - 列の先頭から値を取り出す（popF）
  - 列の末尾から値を取り出す（popB）

Queue

- 次の
  - 列
  - 列
  - 列
  - 列

有名なことわざ

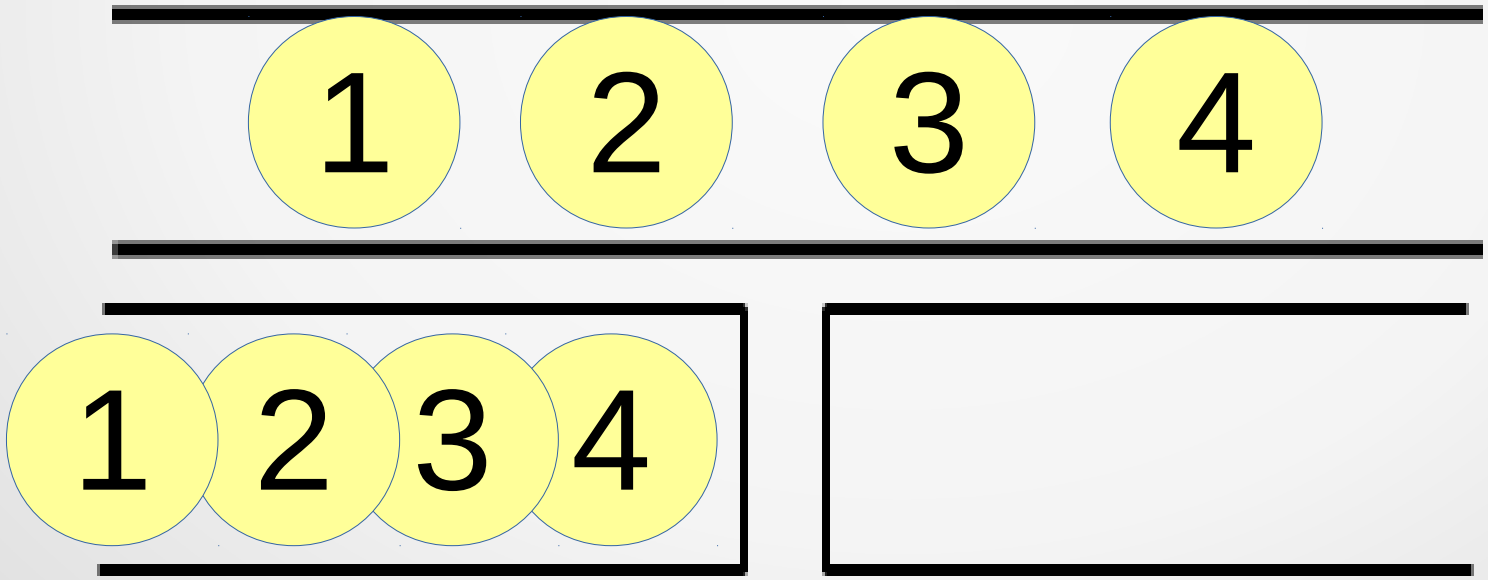
DequeはStack2つで作れる

Deque（Stack × 2）: 図示

- 空のDequeの下にStackが2つあるじゃろ
  - 左が先頭(Front), 右が末尾 (Back)

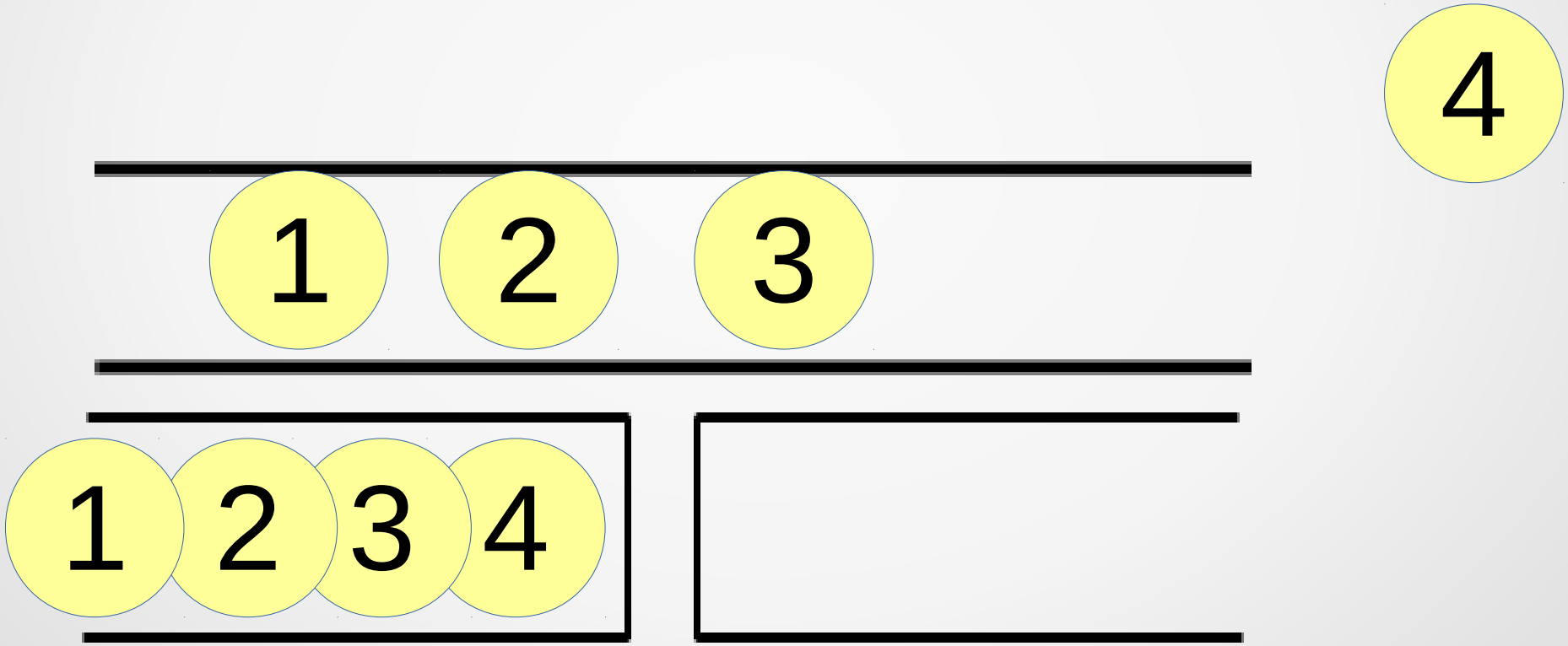
Deque（Stack × 2）: 図示

- pushF(1), pushF(2), pushF(3), pushF(4)



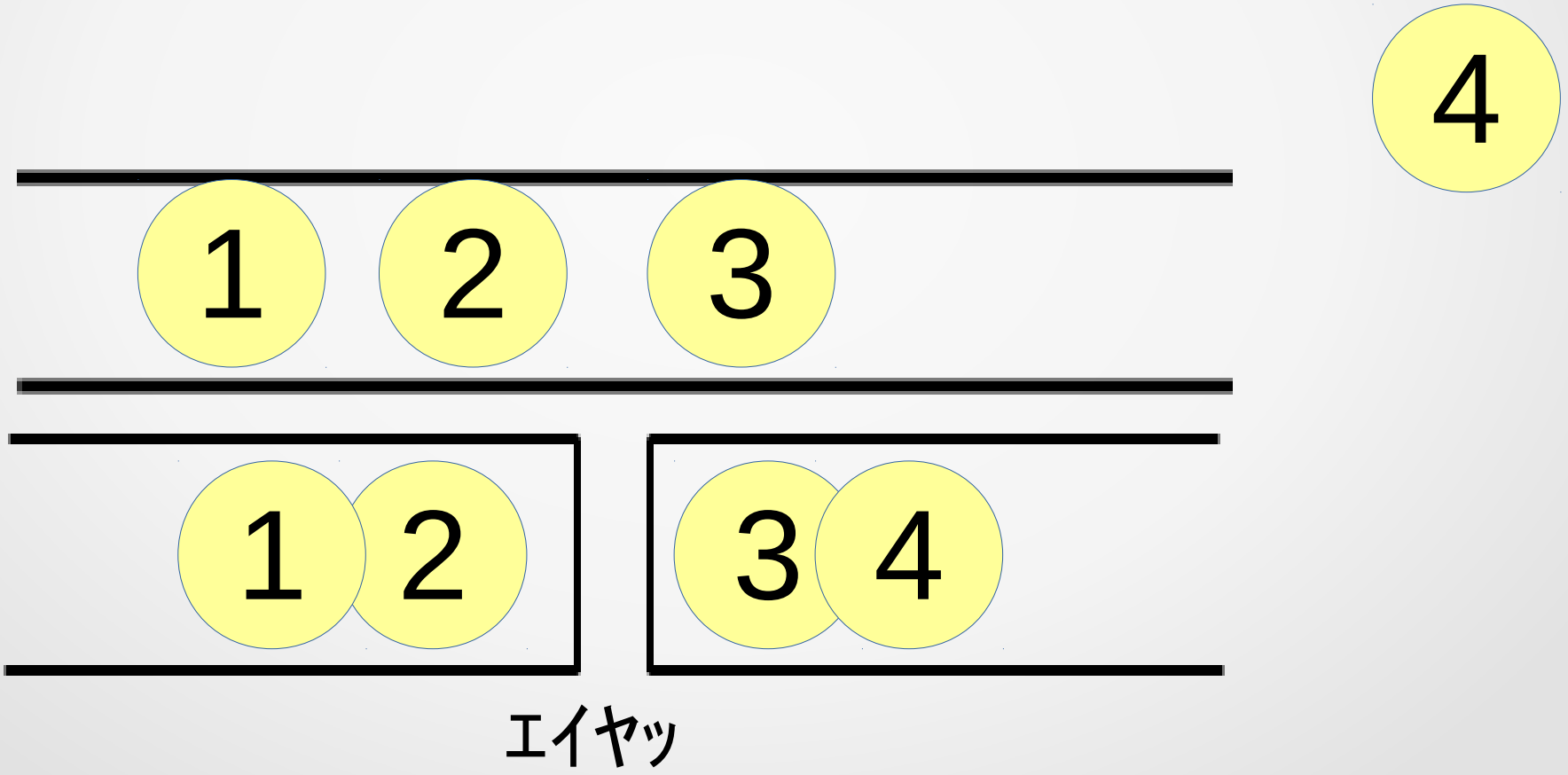
Deque（Stack × 2）: 図示

- pushF(1), pushF(2), pushF(3), pushF(4)
  popB()



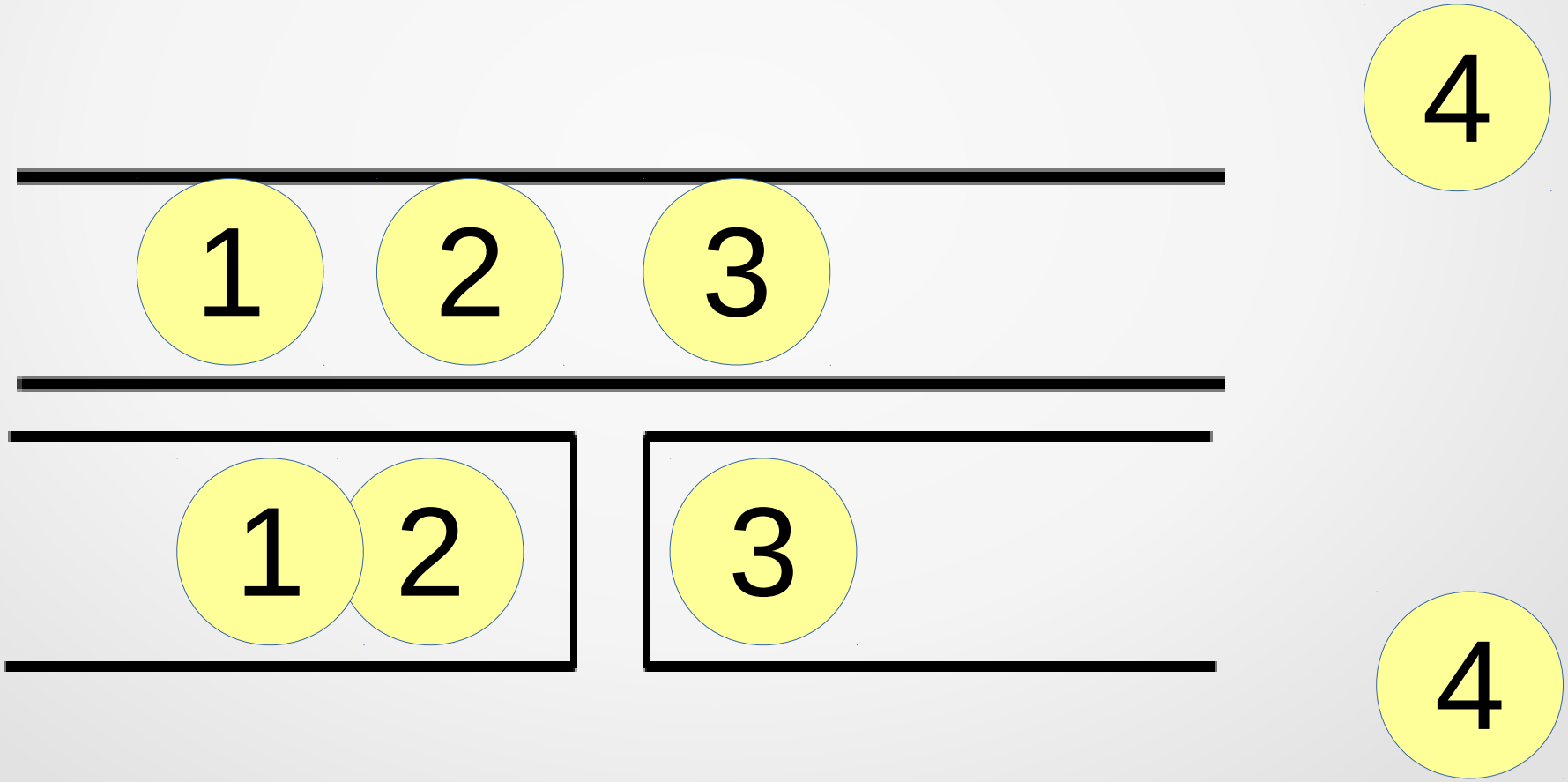
Deque（Stack × 2）: 図示

- pushF(1),　pushF(2), pushF(3), pushF(4)
  popB()



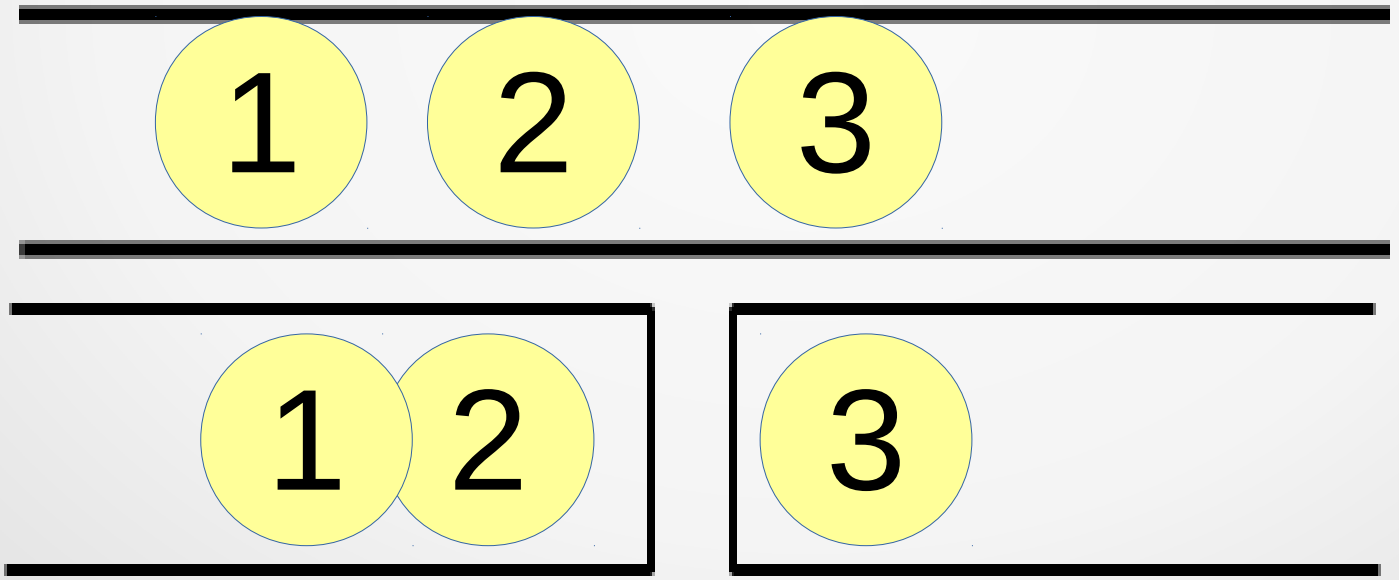
Deque（Stack × 2）: 図示

- pushF(1), pushF(2), pushF(3), pushF(4)
  popB()



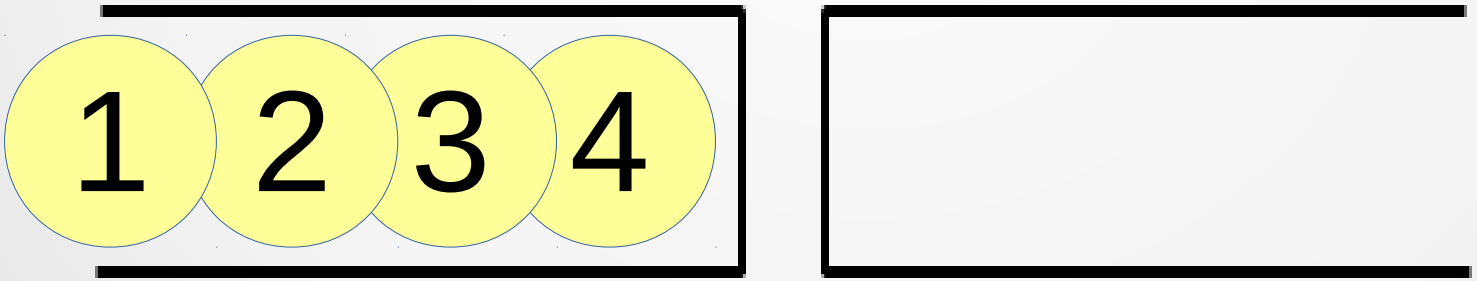
Deque（Stack × 2）: 図示

- pushF(1), pushF(2), pushF(3), pushF(4)
  popB()



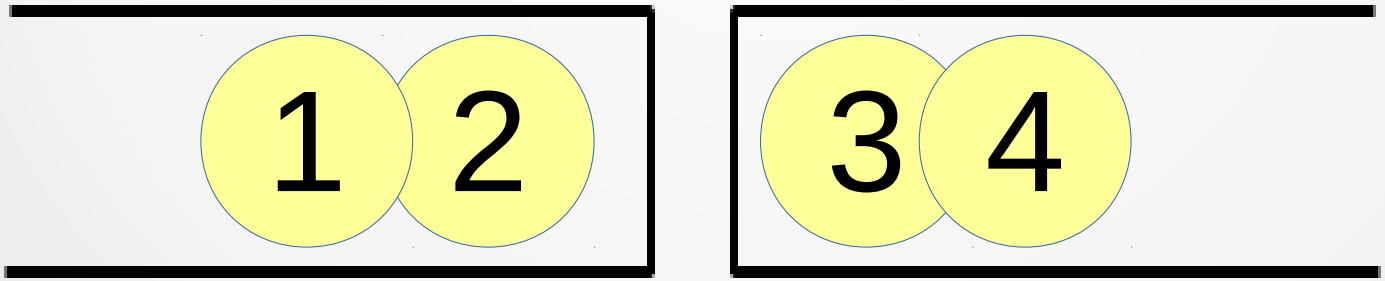
Deque(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！(デジャヴ)
- 一番ややこしいのはpopするときにしたい方のStackが空だった時 (デジャヴ)



Deque(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！(デジャヴ)
- 一番ややこしいのはpopするときにしたい方のStackが空だった時 (デジャヴ)



エイヤッ

Deque(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！（デジャヴ）
- 一番ややこしいのはpopするときにしたい方のStackが空だった時（デジャヴ）
- エイヤッは空じゃないStackの中身を半分ずつにわける操作
  - リストの最初 k 個分をとりだす take
  - リストの最初 k 個を除いたリストを得る drop
  - これらを使えば便利

Deque(Stack×2)

- Stackはさっき実装したばかりだしやるだけじゃん！(デジャヴ)
- 一番ややこしいのはpopするときにしたい方のStackが空だった時 (デジャヴ)
- ｴｲﾔｯは空じゃないStackの中身を半分ずつにわける操作
  - 空じゃない方の奥側半分を逆転して空の方につっこむ
  -

Deque: 実装

(実装をここに書く)

(説明をここに書く)

Deque: 実装

```
Deque a = D [a] [a]

pushF (D xs ys) x = D (x:xs) ys
pushB (D xs ys) y = D xs (y:ys)
```

pushFとpushB (簡単）

Deque: 実装

```
Deque a = D [a] [a]

popF (D x:xs ys) = (x, (D xs ys))
```

左側が空じゃない時のpopF

Deque: 実装

```
Deque a = D [a] [a]

popF (D x:xs ys) = (x, (D xs ys))
popF (D [] ys) = popF (D lh rh)
 where lh = take h ys
       rh = reverse (drop h ys)
       h  = (length ys) `div` 2
```

左側が空の時のpopF

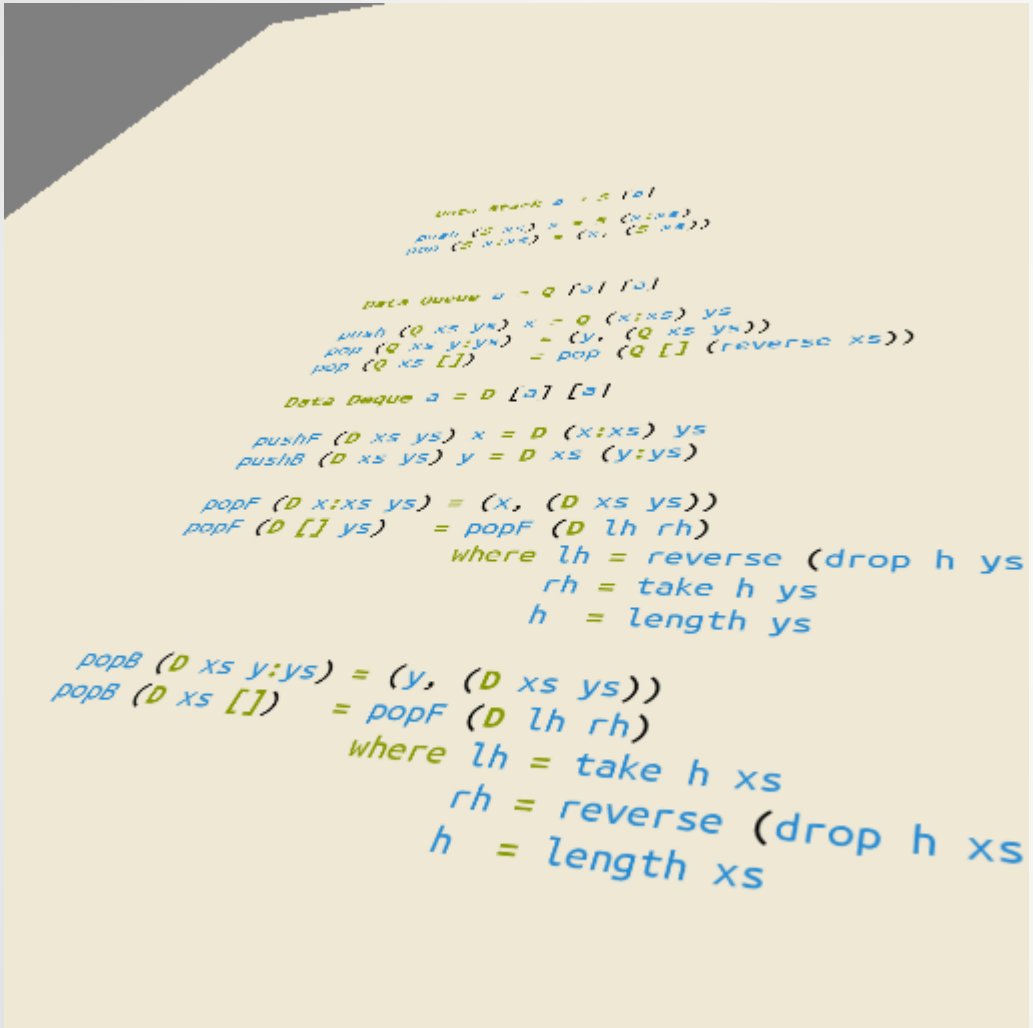
Deque: 実装

```
Deque a = D [a] [a]

popB (D xs y:ys) = (y, (D xs ys))
popB (D xs []) = popB (D lh rh)
 where lh = reverse (drop h xs)
       rh = take h xs
       h  = (length xs) `div` 2
```

popBも同様

クリックしてタイトルを　　！



```
data Queue a = Q [a] [a]

push (Q xs ys) x = Q (x:xs) ys
pop (Q xs y:ys) = (y, (Q xs ys))
pop (Q xs []) = pop (Q [] (reverse xs))
```

```
data Deque a = D [a] [a]

pushF (D xs ys) x = D (x:xs) ys
pushB (D xs ys) y = D xs (y:ys)

popF (D x:xs ys) = (x, (D xs ys))
popF (D [] ys)   = popF (D lh rh)
                   where lh = reverse (drop h ys
                         rh = take h ys
                         h  = length ys

popB (D xs y:ys) = (y, (D xs ys))
popB (D xs [])   = popF (D lh rh)
                   where lh = take h xs
                         rh = reverse (drop h xs
                         h  = length xs
```

やったぜ