

関数型データ構造

と

依存型

同時入門

前提知識

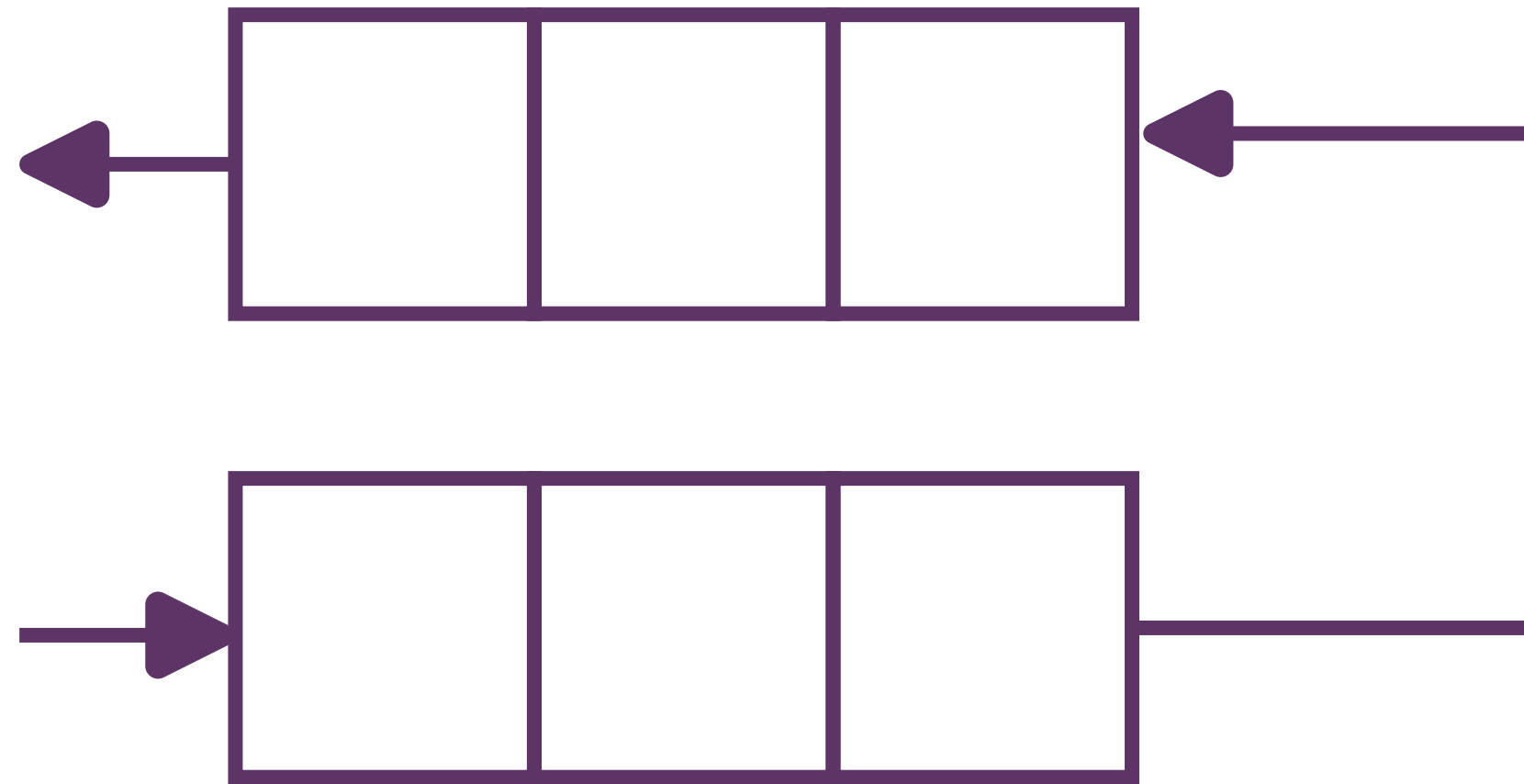
ペアノ数(Nat)

```
data Nat = Z | S Nat
```

- GADT

```
data List : (elem : Type) -> Type
where
  Nil : List elem
  (::) : (x : elem) ->
        (xs : List elem) ->
        List elem
```

Banker's Queue



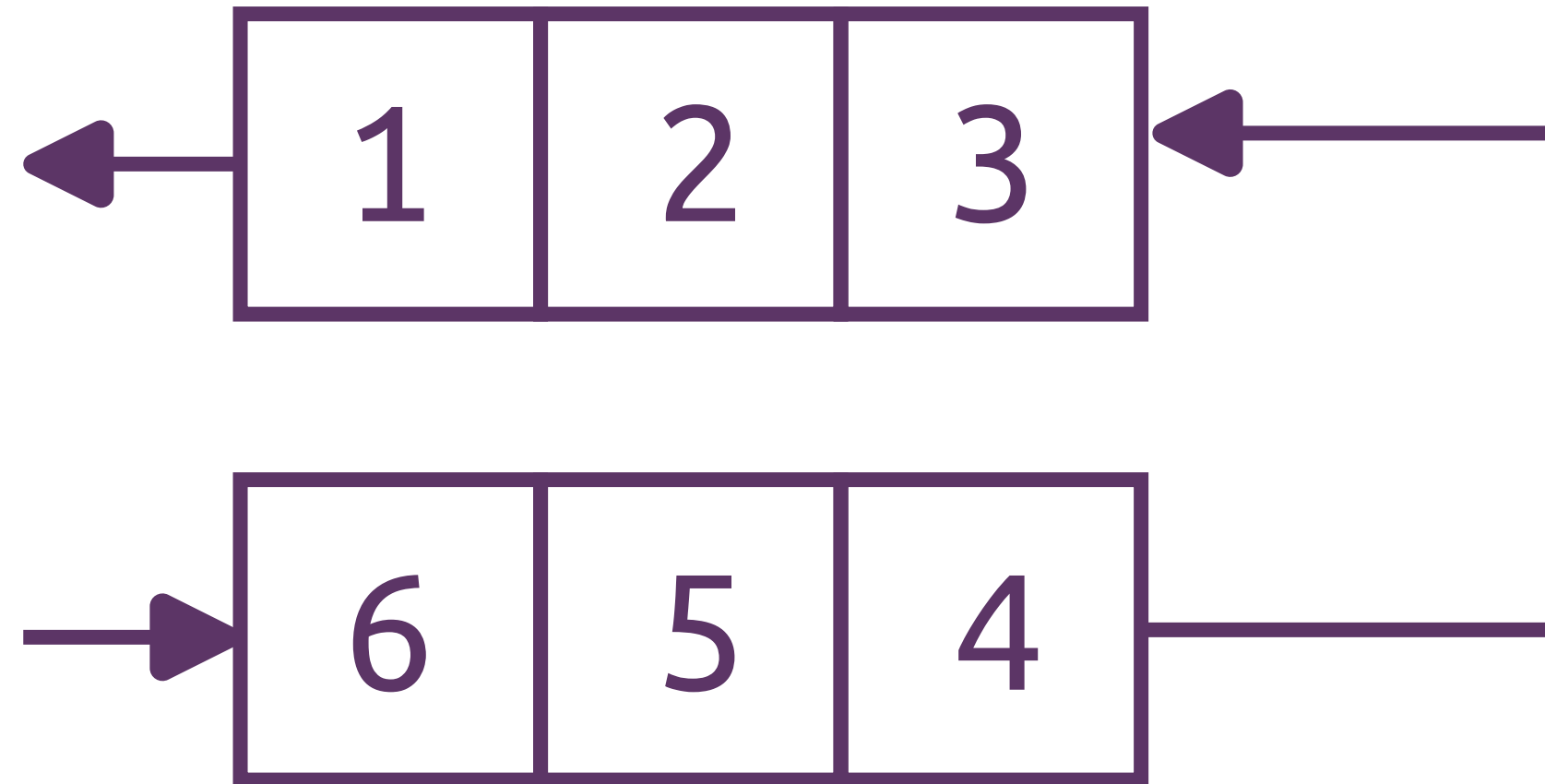
特徴1

関数型データ構造

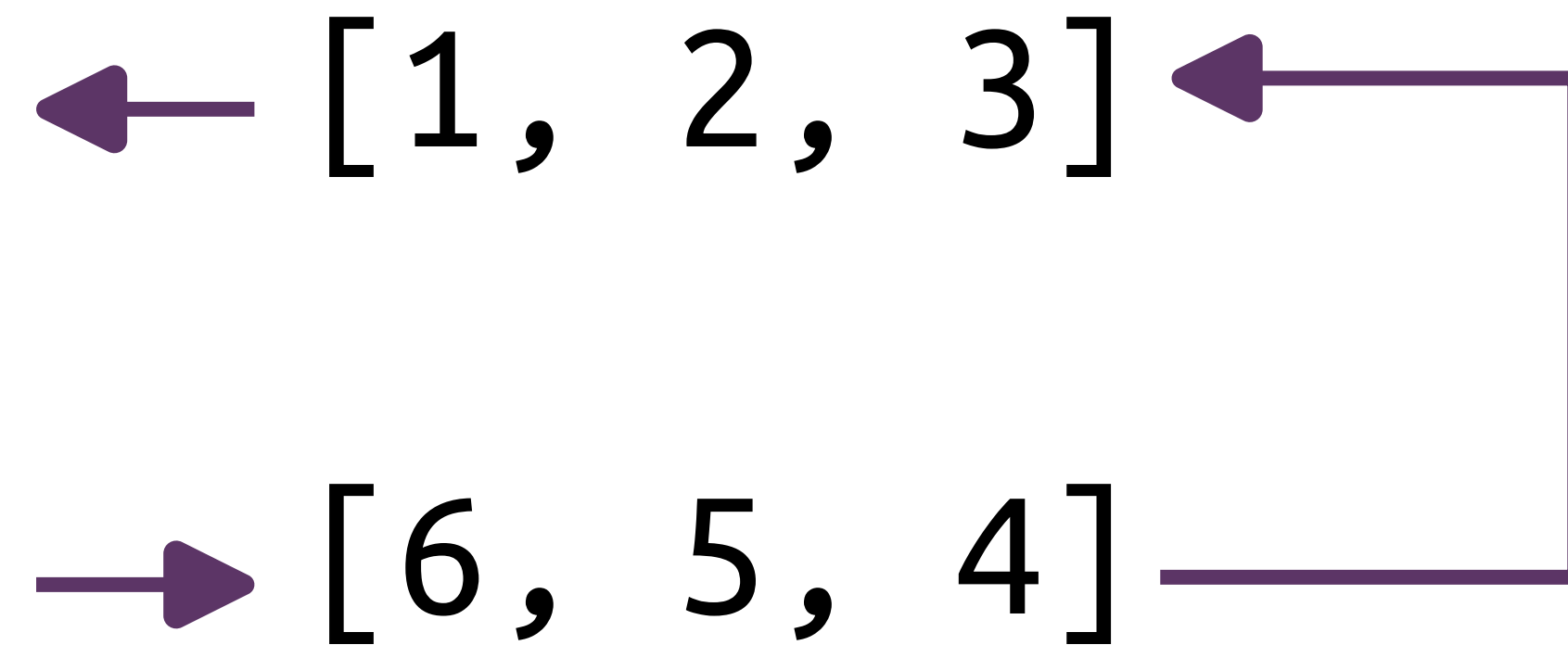
特徴2

各操作の償却計算量が $O(1)$

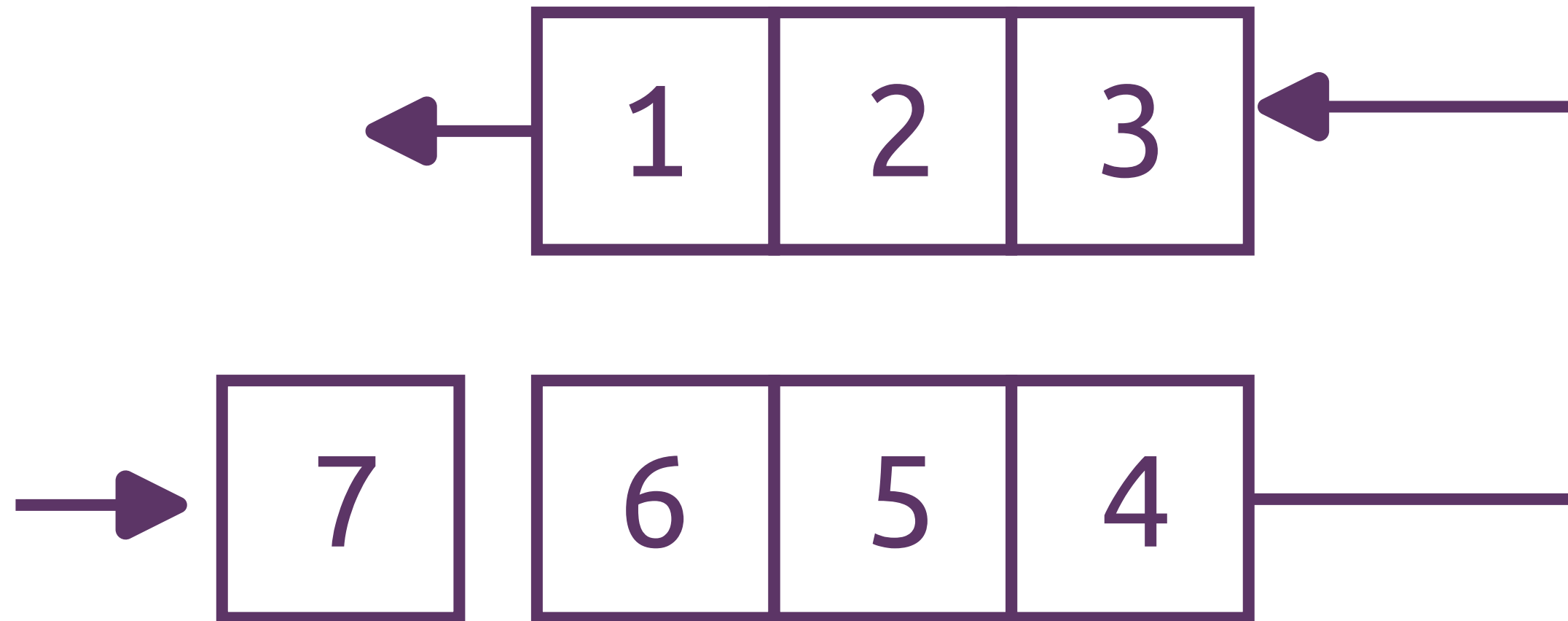
Banker's Queueのアイデア



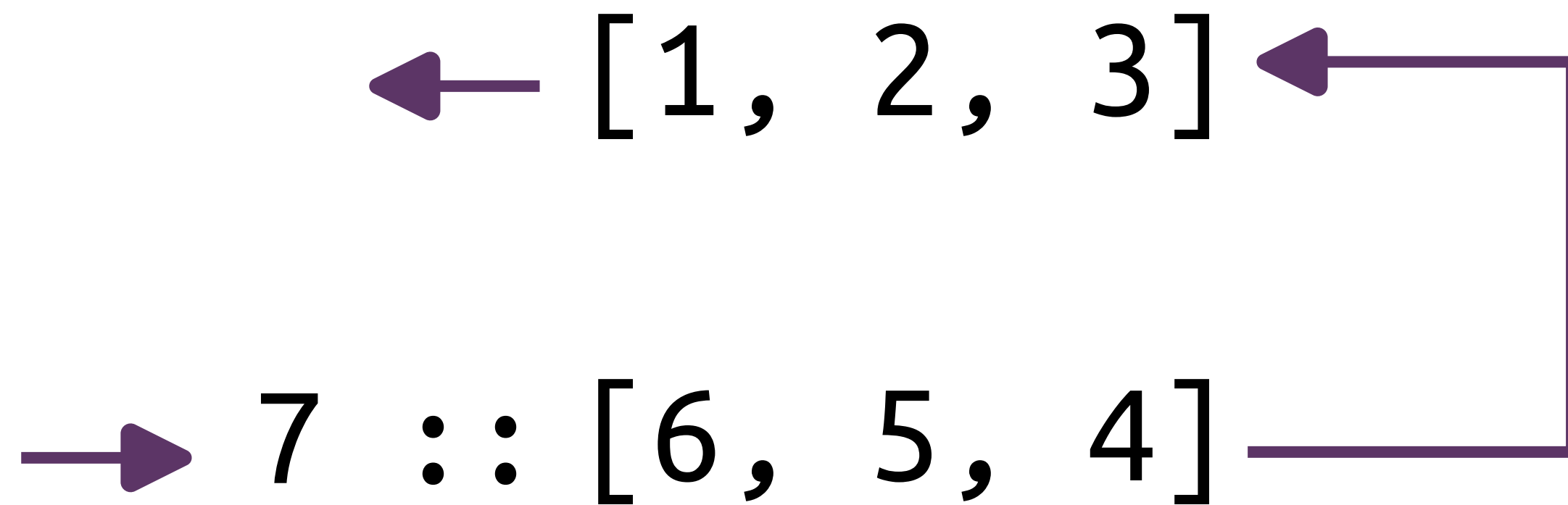
Banker's Queueのアイデア



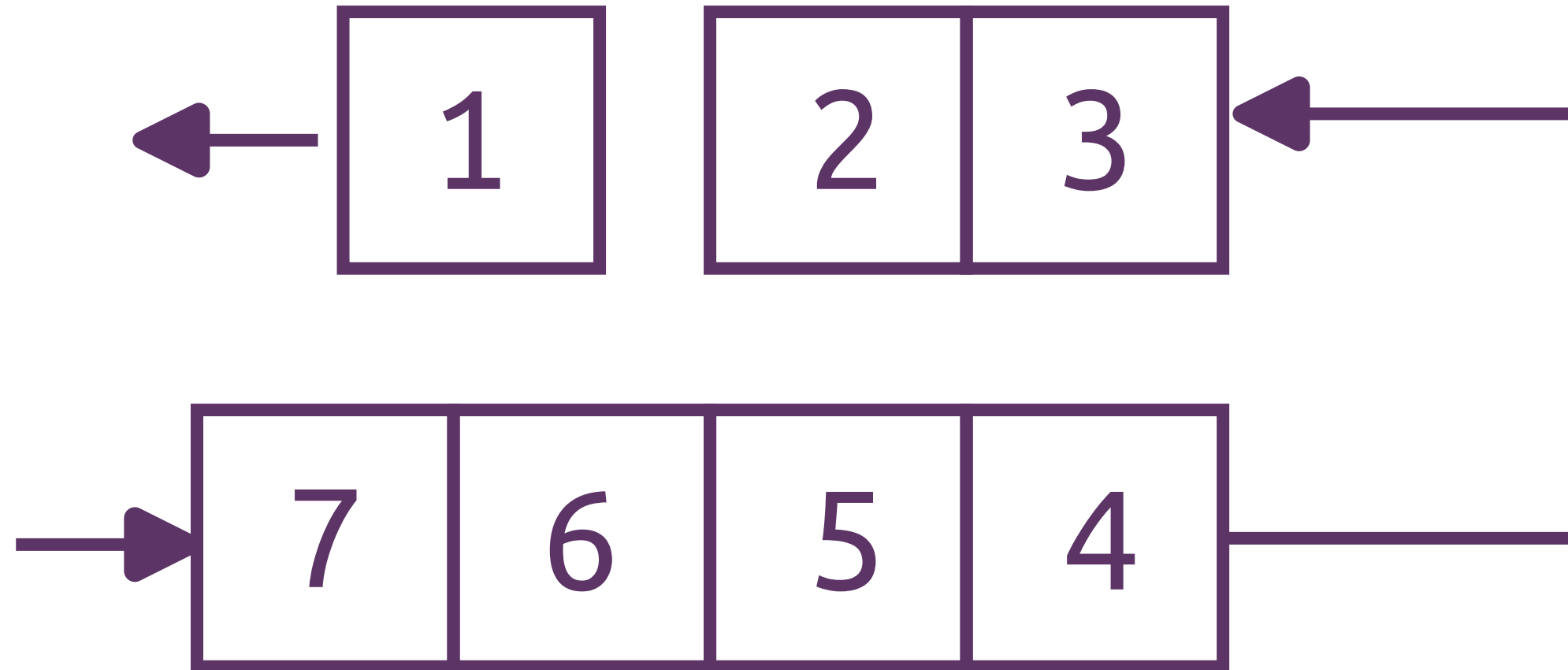
Banker's Queueのアイデア



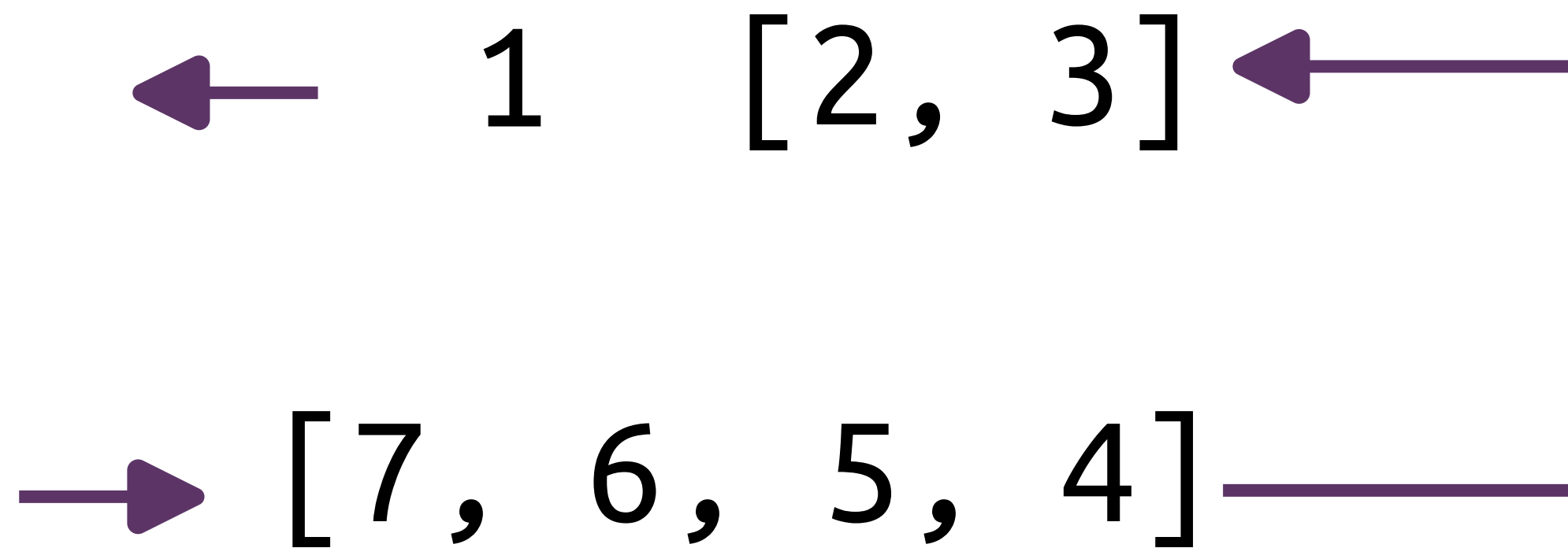
Banker's Queueのアイデア



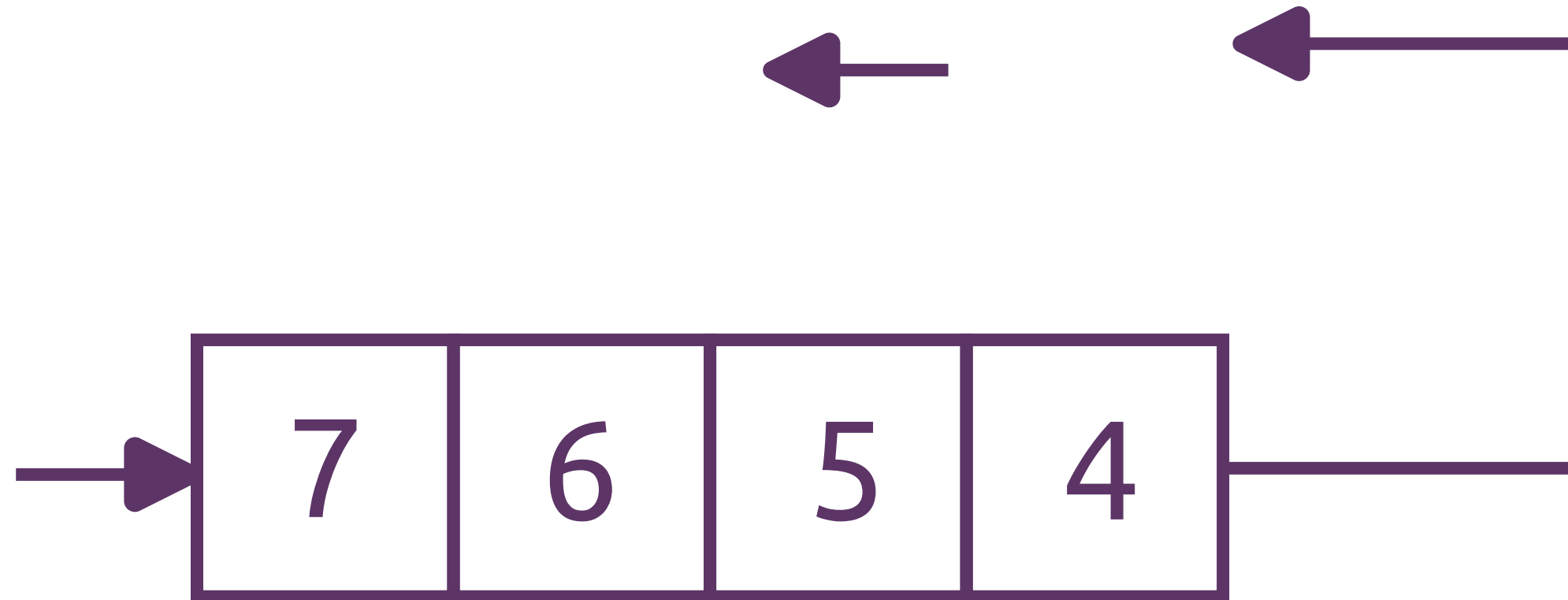
Banker's Queueのアイデア



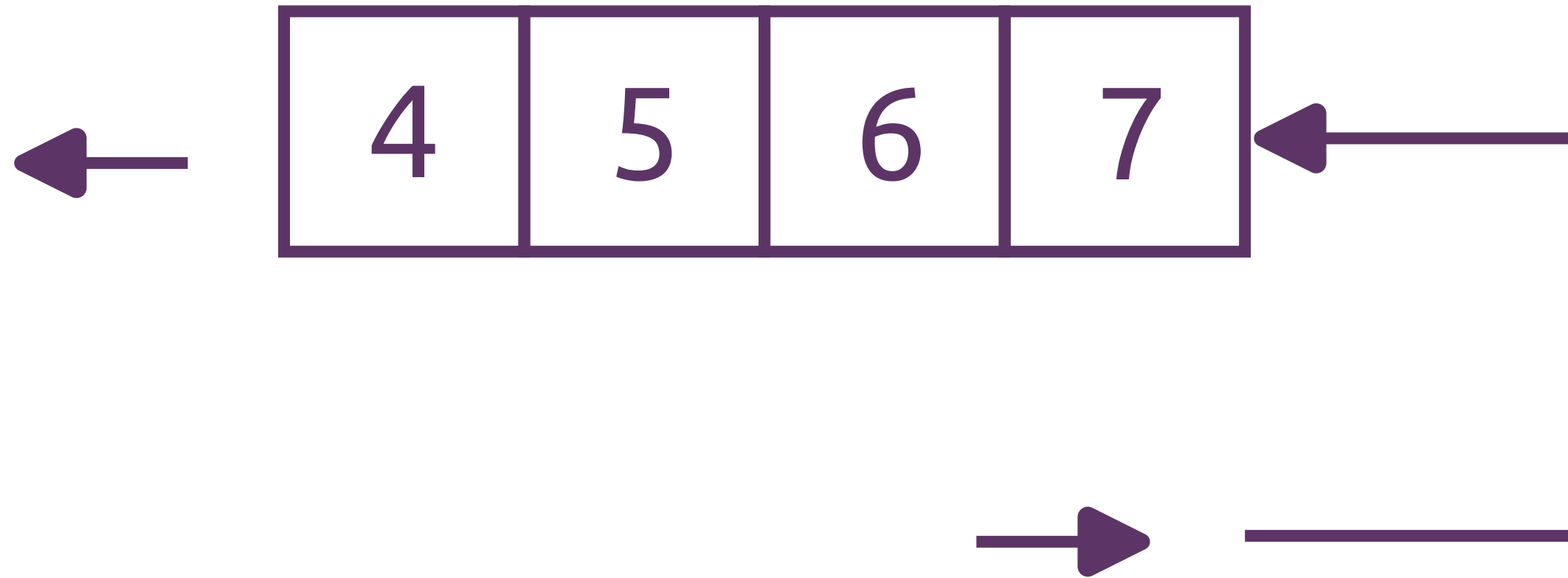
Banker's Queueのアイデア



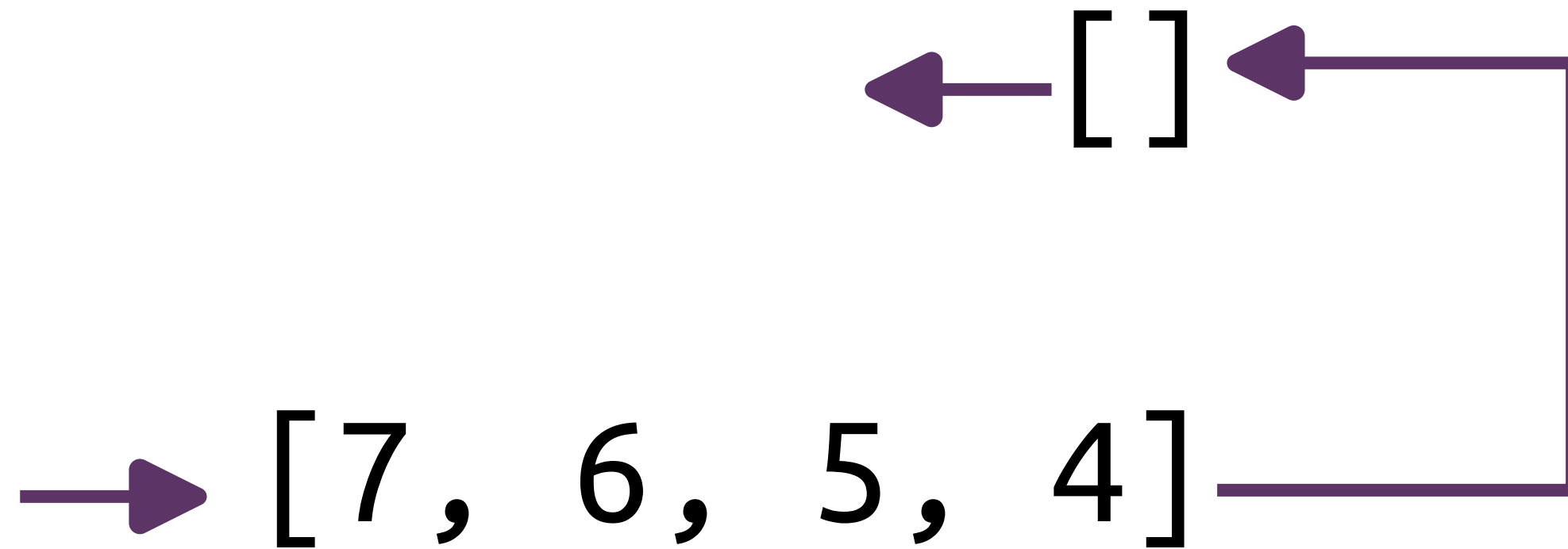
Banker's Queueのアイデア



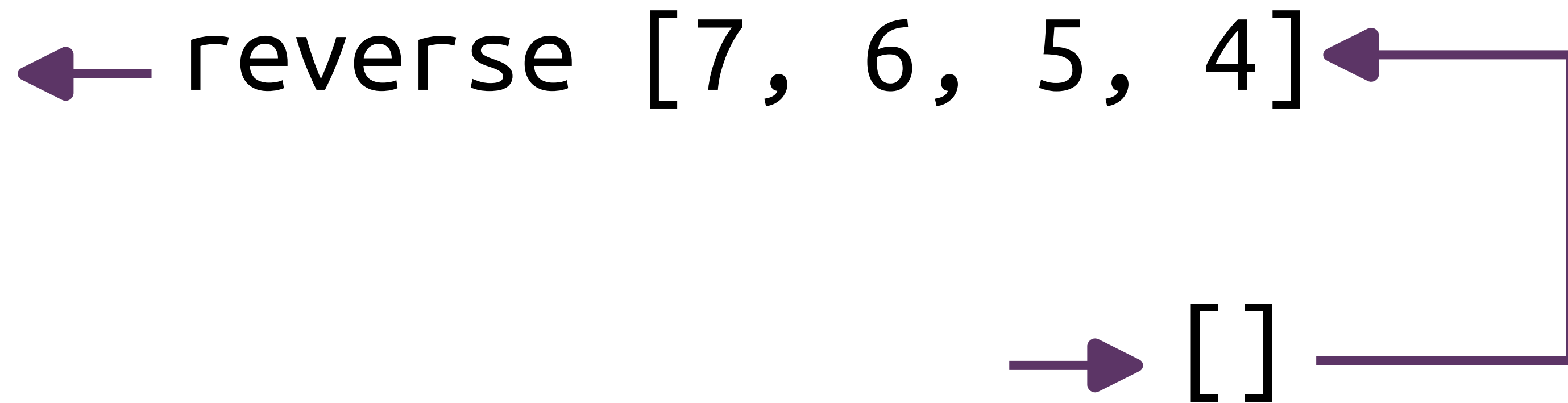
Banker's Queueのアイデア



Banker's Queueのアイデア



Banker's Queueのアイデア

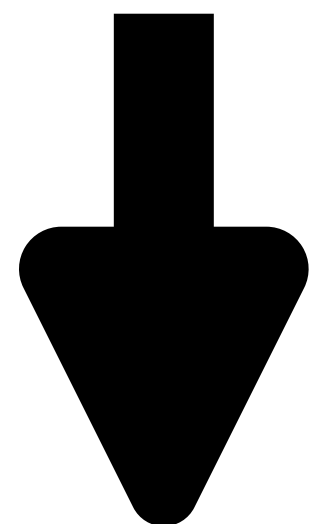


計算量

enqueueは $O(1)$

dequeueは？

基本は定数時間だが、たまにreverseが挟まる



— 1 [2, 3]
— [7, 6, 5, 4]

— reverse [7, 6, 5, 4]
— []

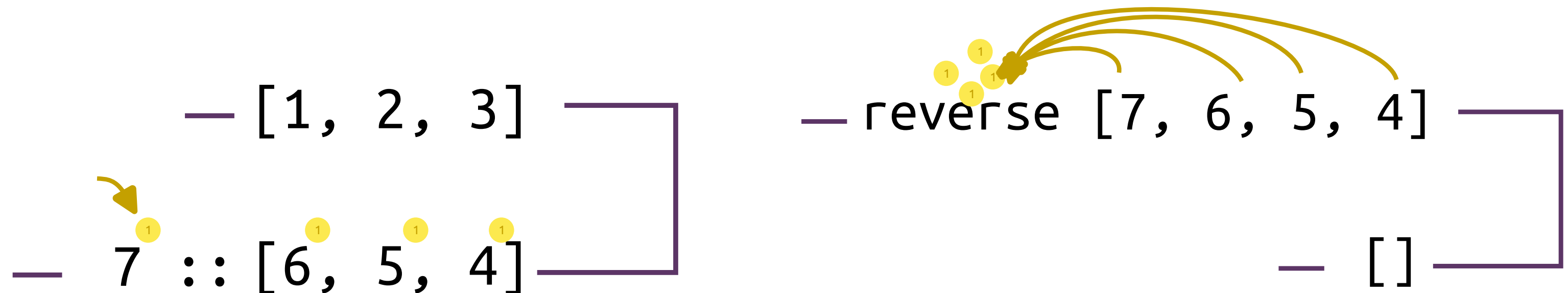
それまでの操作に依存するので
操作列で考えよう

計算量

dequeueの償却計算量は $O(1)$

償却計算量

操作列を考えたときの
平均のコスト



enqueueのときにreverseのコストを積立てていて考える

依存型の話

依存型とは

- ざっくりいうと型の位置
に値を書ける

```
v1 : Vect 3 Integer
```

```
v1 = [1, 2, 3]
```

```
v2 : Vect 2 Integer
```

```
v2 = [4, 5]
```

```
v3 : Vect 5 Integer
```

```
v3 = v1 ++ v2
```

依存型

依存型とは、計算機科学と論理学において、値に依存する型のことである

依存型のある
プログラミング言語
Idrisでの例

依存型とデータ型

```
data Vect : (len : Nat) -> (elem : Type) -> Type where  
  Nil    : Vect Z elem  
  (::)   : (x : elem) -> (xs : Vect len elem) -> Vect (S len) elem
```

```
v1 : Vect 3 Integer  
v1 = [1, 2, 3]
```

依存型とデータ型

ユーザインタフェースとしての依存型

```
head : Vect (S len) elem -> elem
```

```
head (x :: xs) = x
```

```
(++) : (xs : Vect m elem) ->  
      (ys : Vect n elem) ->  
      Vect (m + n) elem
```

```
(++) [] ys = ys
```

```
(++) (x :: xs) ys = x :: xs ++ ys
```

依存型とデータ型

長さ1以上を要求

ユースケースとしての依存型

```
head : Vect (S len) elem -> elem
```

```
head (x :: xs) = x
```

結果がMaybeにならない

```
(++) : (xs : Vect m elem,  
        ys : Vect n elem) ->  
        Vect (m + n) elem
```

長さを計算できる

```
(++) [] ys = ys
```

```
(++) (x :: xs) ys = x :: xs ++ ys
```

依存型とデータ型

実装補助としての依存型

```
data Tree : Nat -> Type -> Type -> Type where
  Leaf : k -> v -> Tree Z k v
  Branch2 : Tree n k v -> k ->
             Tree n k v -> Tree (S n) k v
  Branch3 : Tree n k v -> k ->
             Tree n k v -> k ->
             Tree n k v -> Tree (S n) k v
```

2-3木の実装

依存型とデータ型

実装補助としての依存型

木の高さを型パラメータに持つ

```
data Tree : Nat -> Type -> Type -> Type where
  Leaf : k -> v -> Tree Z k v
  Branch2 : Tree n k v -> k ->
             Tree n k v -> Tree (S n) k v
  Branch3 : Tree n k v -> k ->
             Tree n k v -> k ->
             Tree n k v -> Tree (S n) k v
```

2-3木の実装

依存型とデータ型

実装補助としての依存型

```
data Tree : Nat -> Type -> Type -> Type where
```

```
Leaf : k -> v -> Tree Z k v
```

```
Branch2 : Tree n k v -> k ->
```

```
Tree n k v -> Tree (S n) k v
```

```
Branch3 : Tree n k v -> k ->
```

```
Tree n k v -> k ->
```

```
Tree n k v -> Tree (S n) k v
```

同じ高さの木しか受け付けない

2-3木の実装

今日の話

依存型を使ってBanker's Queueを実装しよう

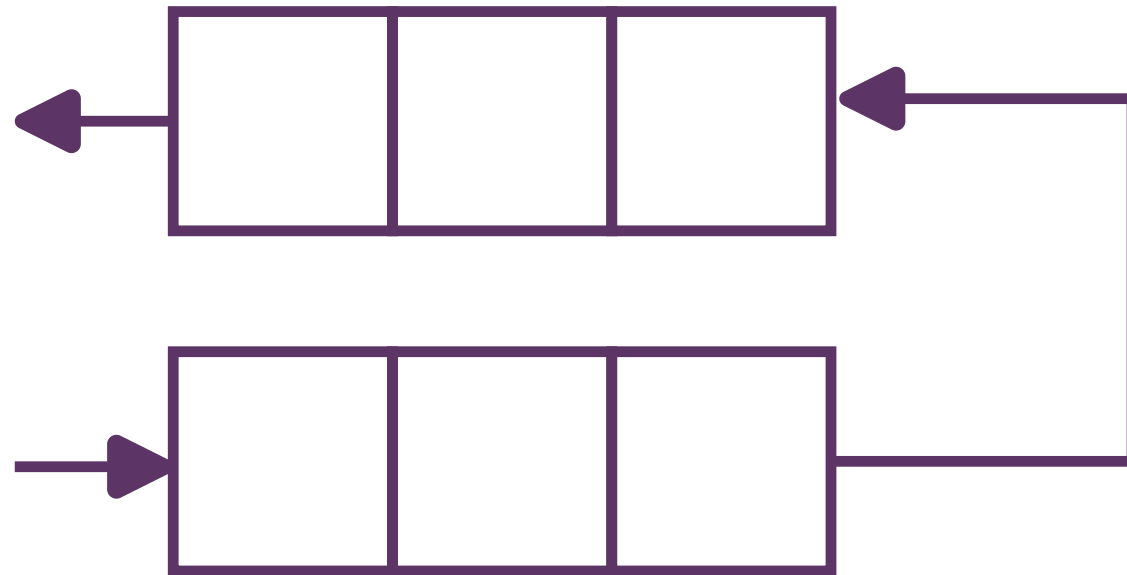
長さを型に持つ

<https://gitlab.com/blackenedgold/idris-dependent-queue>

実用上は n 以下の長さを持つQueueの方が便利だけど話がややこしくなるので今回は長さ指定
上記URLに n 以下の長さのキューの実装もある

定義

```
data Queue : Nat -> Type -> Type where  
  MkQueue : (front : Vect l a) ->  
            (rear : Vect m a) ->  
            Queue (l + m) a
```



簡単な関数

```
empty : Queue 0 a
empty = MkQueue [] []

isEmpty : Queue n a -> Bool
isEmpty (MkQueue [] []) = True
isEmpty _ = False
```

lengthは必要ない。型から取り出せる

Enqueue?

```
enqueue : Queue n a -> a -> Queue (S n) a
enqueue (MkQueue front rear) x =
  MkQueue front (x :: rear)
```

Enqueue?

- + Errors (1)

`-- Queue.idr line 42 col 3:

When checking right hand side of enqueue with expected type

`Queue (S (l + m)) a`

Type mismatch between

`Queue (l + S m) a (Type of MkQueue front (x :: rear))`

and

`Queue (S (plus l m)) a (Expected type)`

Specifically:

Type mismatch between

`plus l (S m)`

and

`S (plus l m)`

Enqueue?

Type mismatch between

`plus l (S m)`

and

`S (plus l m)`

要約すると

$$l + (1 + m) \neq 1 + (l + m)$$

人間なら目で分かるが、Idrisは分からない
ルールを教えてコンパイラを説得したい

→証明

Enqueue?

標準ライブラリにまさにな関数がある

```
plusSuccRightSucc :  
  (left : Nat) ->  
  (right : Nat) ->  
  S (left + right) = left + S right
```

Enqueue!

```
enqueue : Queue n a -> a -> Queue (S n) a
enqueue (MkQueue {l} {m} front rear) x =
  rewrite plusSuccRightSucc l m in
  MkQueue front (x :: rear)
```

Enqueue!

型からlとmを取り出す

```
enqueue : Queue n a -> a -> Queue (S n) a  
enqueue (MkQueue {l} {m} front rear) x =  
  rewrite plusSuccRightSucc l m in  
  MkQueue front (x :: rear)
```

rewriteを使って
環境の型を書き換え

Dequeue?

```
dequeue : Queue (S n) a -> (a, Queue n a)
dequeue (MkQueue (x :: xs) rear) =
    (x, MkQueue xs rear)
dequeue (MkQueue [] rear) =
    case reverse rear of
        e :: front => (e, MkQueue front [])
```

長さ1以上

Dequeue?

必ず値が返る

```
dequeue : Queue (S n) a -> (a, Queue n a)
dequeue (MkQueue (x :: xs) rear) =
  (x, MkQueue xs rear)
dequeue (MkQueue [] rear) =
  case reverse rear of
    e :: front => (e, MkQueue front [])
```

Deque?

```
- + Errors (1)
`-- Queue.idr line 50 col 16:
    When checking right hand side of Queue.case block in dequeue at Queue.idr:49:8-19 with expected type
        (a, Queue n a)

    When checking argument b to constructor Builtins.MkPair:
        Type mismatch between
            Queue (n + 0) a (Type of MkQueue front [])
        and
            Queue n a (Expected type)

    Specifically:
        Type mismatch between
            plus n 0
        and
            n
```

Deque!

```
dequeue : Queue (S n) a -> (a, Queue n a)
dequeue (MkQueue (x :: xs) rear) =
  (x, MkQueue xs rear)
dequeue {n} (MkQueue [] rear) =
  rewrite sym (plusZeroRightNeutral n) in
  case reverse rear of
    e :: front => (e, MkQueue front [])
```

利用例

```
let q: Queue 0 Int = empty in
let q: Queue 1 Int = enqueue q 1 in
let q: Queue 2 Int = enqueue q 2 in
let (e, q) = dequeue q in
-- e == 2
let (e, q) = dequeue q in
-- e == 1
-- コンパイルエラー
-- let (e, q) = dequeue q in
```

まとめ

Banker's Queueは関数型データ構造

依存型は型で値を使える

Idrisで依存型が扱える

依存型はちょっと難しい