Freeze After Writing:
Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Aaron Turon
Neelakantan R. Krishnaswami
Ryan R. Newton

POPL '14, January 23, 2014

Freeze After Writing

- LVars
- Quasi-Deterministic

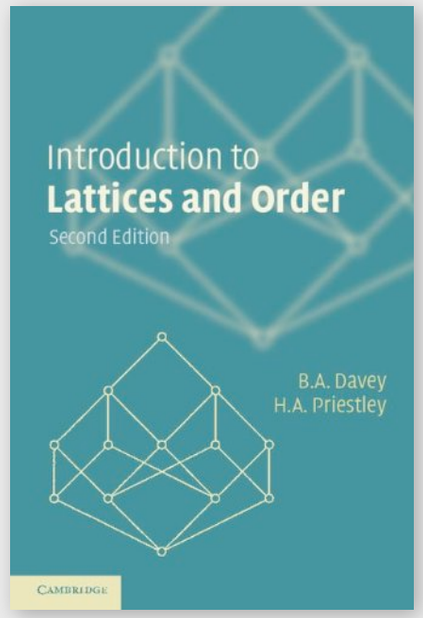
Parallel Programming

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```



```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)] (Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
][(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

```
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)][Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1)][(Book,1),(Shoes,1)
] [(Shoes,1)] [(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

Terminal — bash — 90×27

bash

```
landin:lvar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main             ( map-ioref-data-race.hs, map-ioref-data-race.o )
Linking map-ioref-data-race ...
while true; do ./map-ioref-data-race +RTS -N2; done
[(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1
)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1
),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

```
s,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Sho
es,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Bo
ok,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(B
ook,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(
Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][
(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)]
[(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),
(Shoes,1)][(Book,1),(Shoes,1)] [(Book,1)] [(Book,1),(Shoes,1)][(Book,1),(Shoes,1)
][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)] [(Book,1)] [(Book,1),(Shoes,1)
] [(Shoes,1)] [(Shoes,1)][(Book,1),(Shoes,1)][(Shoes,1)][(Book,1),(Shoes
,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book
,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)] [(Shoes,1)] [(Boo
k,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoes,1)][(Book,1),(Shoe
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async $ atomicModifyIORef cart
                (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async $ atomicModifyIORef cart
                     (\m -> (insert Book 1 m, ()))
       a2 <- async $ atomicModifyIORef cart
                     (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async $ atomicModifyIORef cart
                     (\m -> (insert Book 1 m, ()))
       a2 <- async $ atomicModifyIORef cart
                     (\m -> (insert Shoes 1 m, ()))
       res <- async $ do waitBoth a1 a2
       wait res                readIORef cart
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       a1 <- async $ atomicModifyIORef cart
                     (\m -> (insert Book 1 m, ()))
       a2 <- async $ atomicModifyIORef cart
                     (\m -> (insert Shoes 1 m, ()))
       res <- async $ do waitBoth a1 a2
                         readIORef cart
       wait res
```

```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async $ atomicModifyIORef cart
        (\c -> (insert Book 1 c, ()))
  a2 <- async $ atomicModifyIORef cart
        (\c -> (insert Shoes 1 c, ()))
  res <- async $ do waitBoth a1 a2
                    readIORef cart
  wait res

main = do v <- p
          putStr $ show $ toList v
```

Deterministic,
but only because we put the `wait`s
in the right places

```haskell
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async $ atomicModifyIORef cart
        (\c -> (insert Book 1 c, ()))
  a2 <- async $ atomicModifyIORef cart
        (\c -> (insert Shoes 1 c, ()))
  res <- async $ do waitBoth a1 a2
                    readIORef cart
  wait res

main = do v <- p
          putStr $ show $ toList v
```

Deterministic,
but only because we put the `waits`
in the right places

```haskell
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Book 1 cart
  fork $ insert Shoes 1 cart
  return cart

main = do
  putStr $ show $ toList $
    fromIMap $ runParThenFreeze p
```

Deterministic by construction

Freeze After Writing

- LVars
- Quasi-Deterministic

Parallel Programming

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

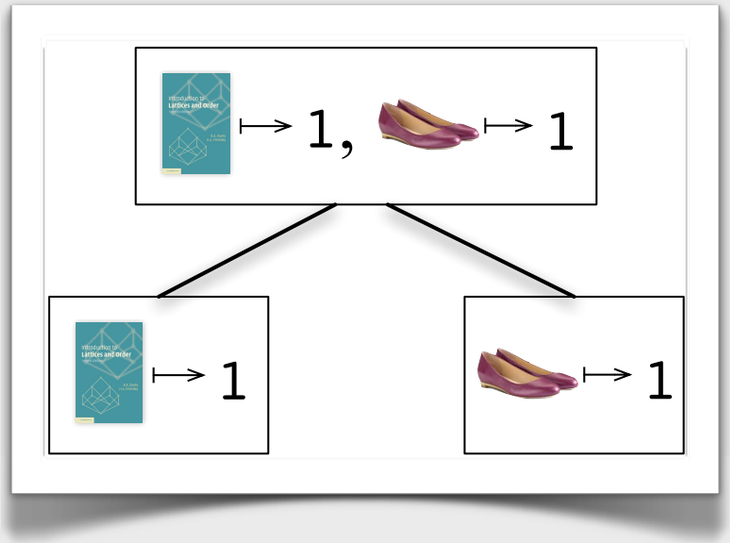
```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]

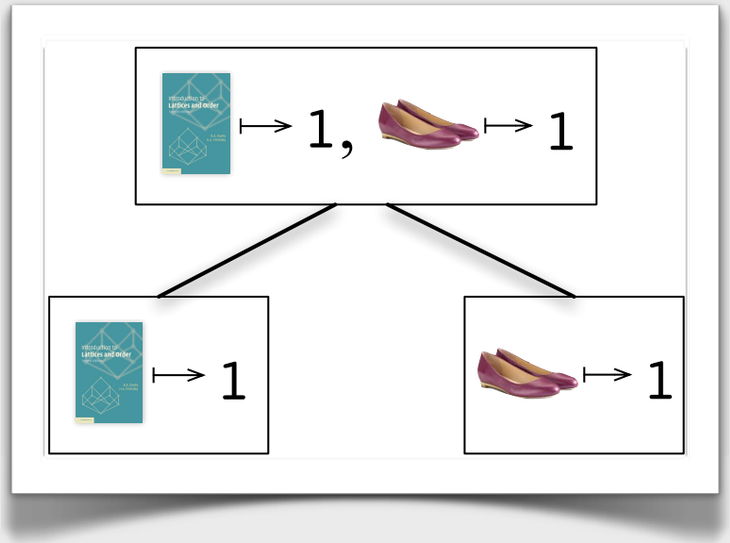
```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```



IVars: single writes, blocking (but exact) reads

[Arvind *et al.*, 1989]

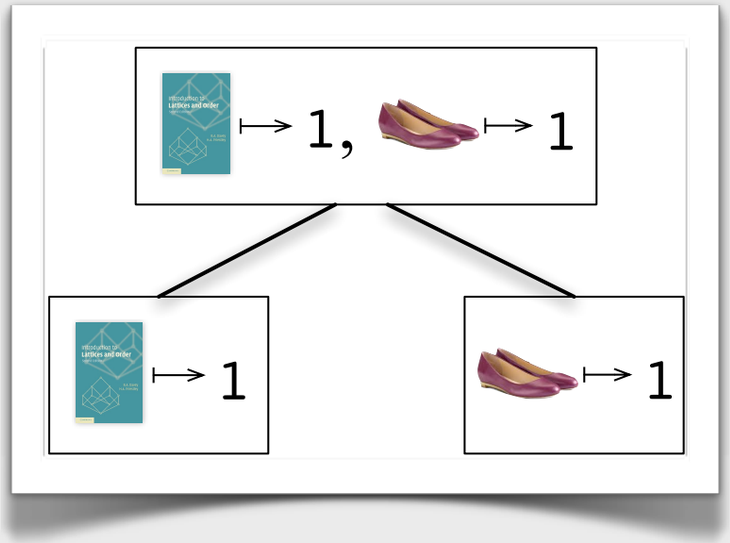
```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```



IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: multiple *least-upper-bound* writes, blocking *threshold* reads

```haskell
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
       async $ atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))
       async $ atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))
       res <- async $ readIORef cart
       wait res
```

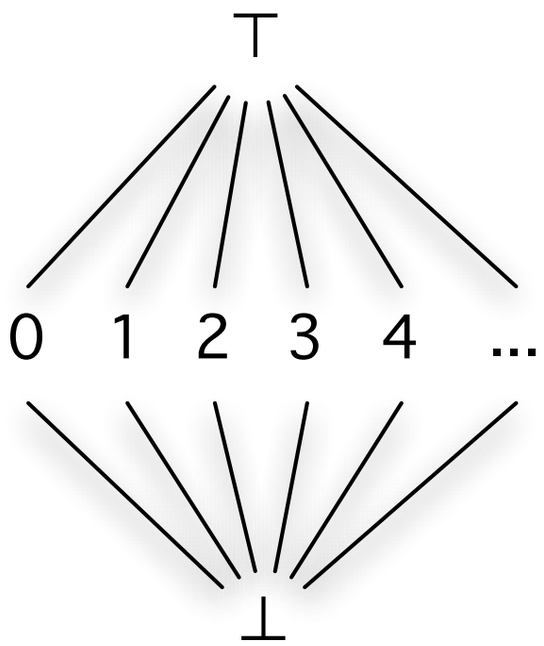


IVars: single writes, blocking (but exact) reads
[Arvind *et al.*, 1989]

LVars: multiple *least-upper-bound* writes, blocking *threshold* reads

* actually a bounded join-semilattice

num

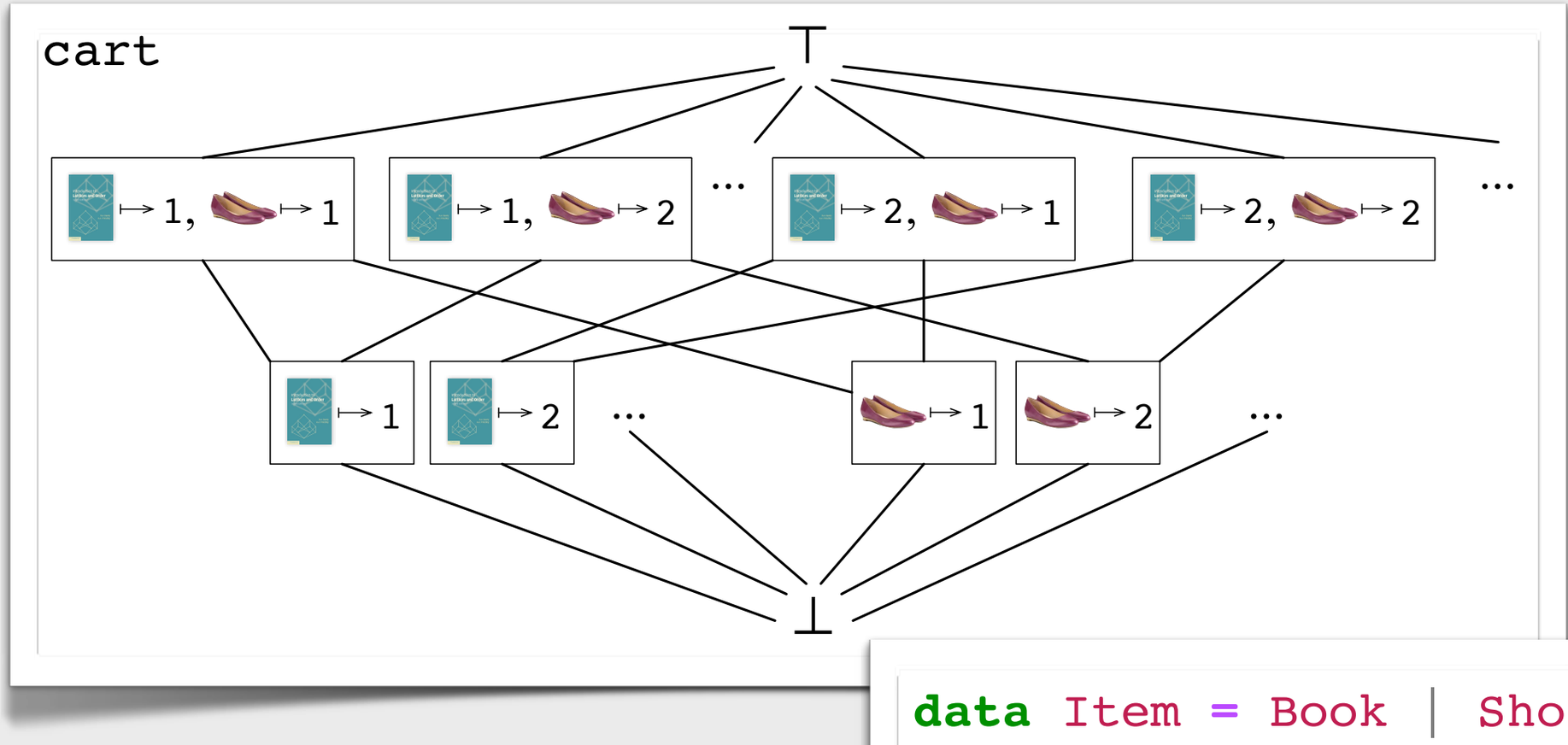


Raises an error, since 3 ⊔ 4 = ⊤

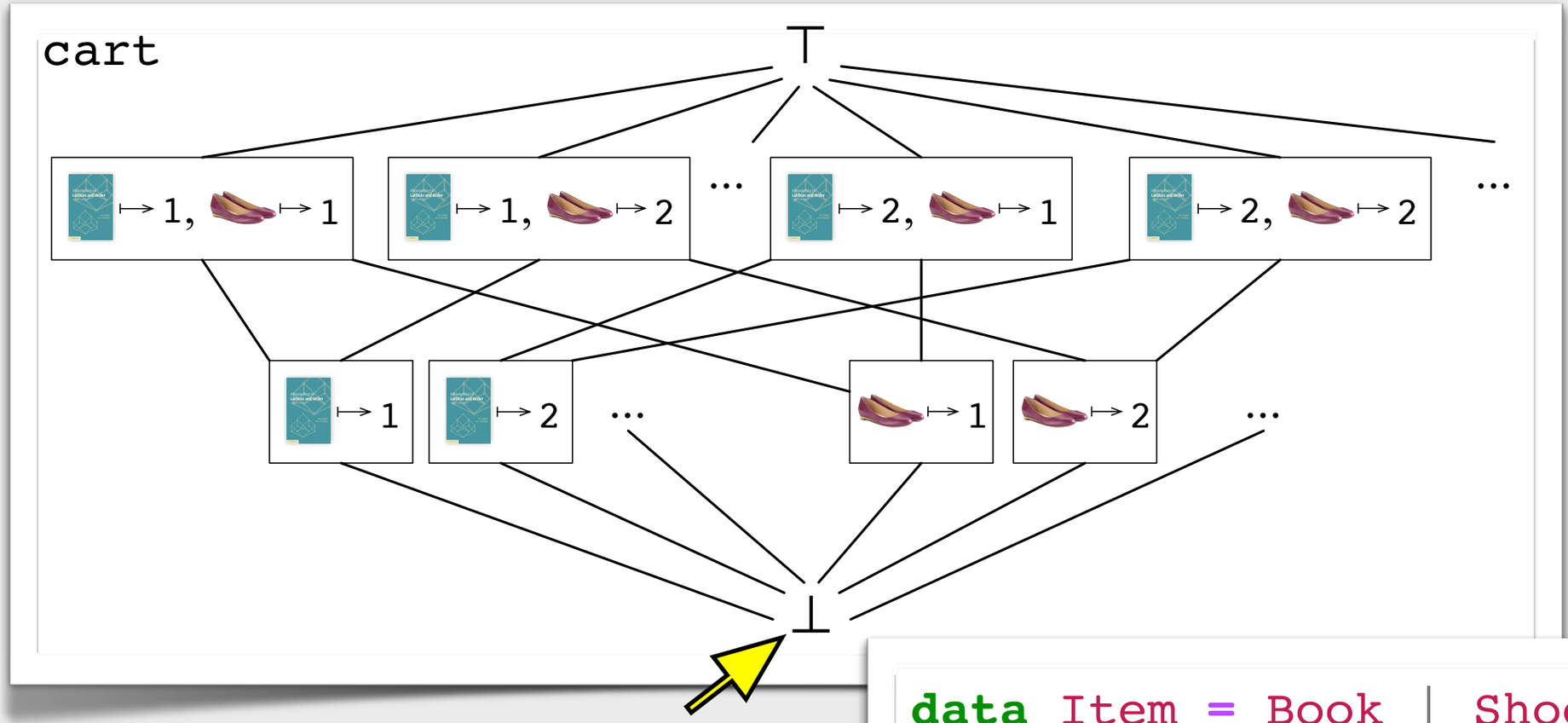
```
do
   fork (put num 3)
   fork (put num 4)
```

Works fine, since 4 ⊔ 4 = 4

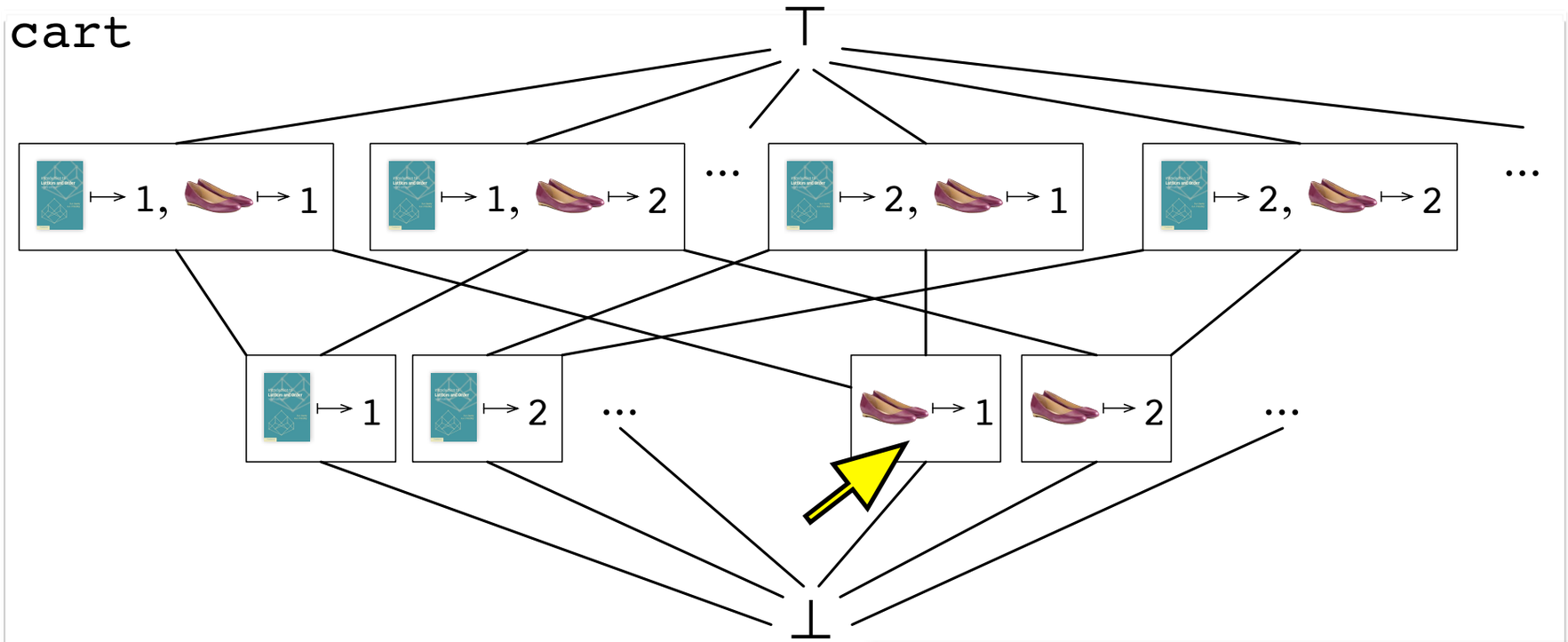
```
do
   fork (put num 4)
   fork (put num 4)
```



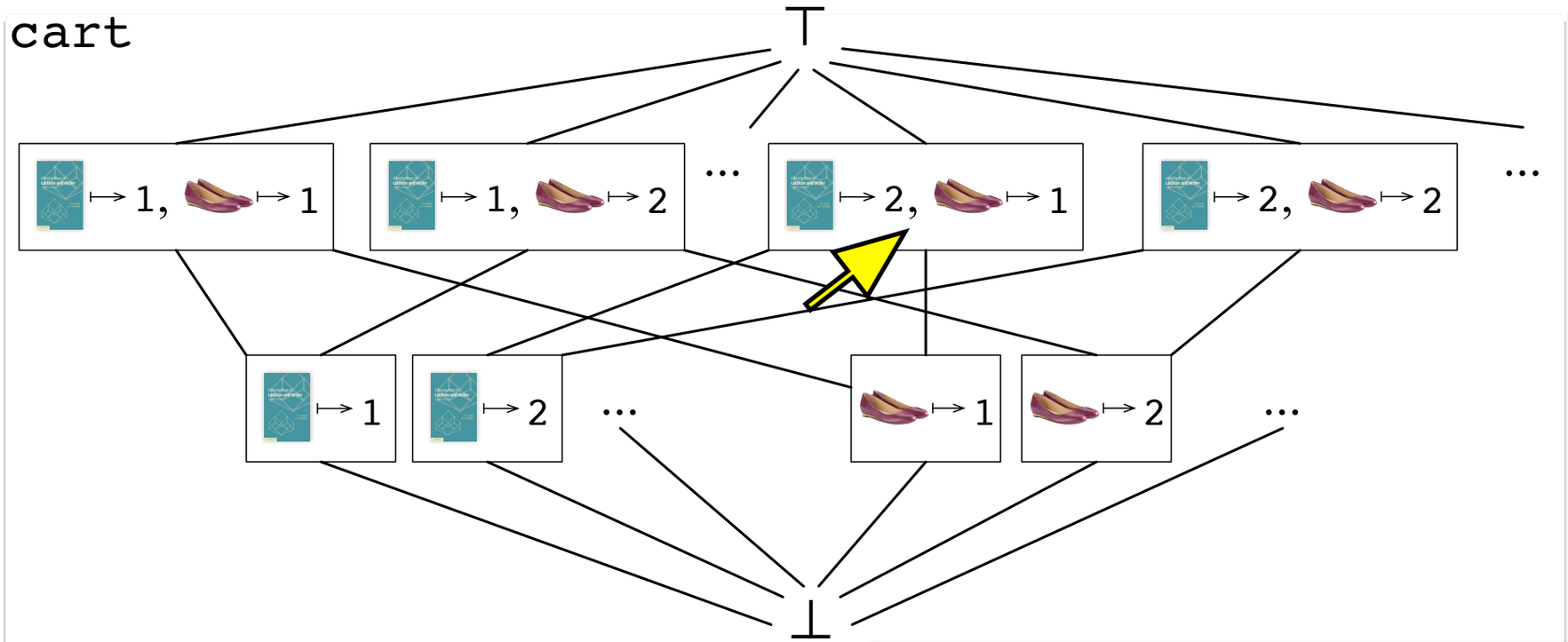

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



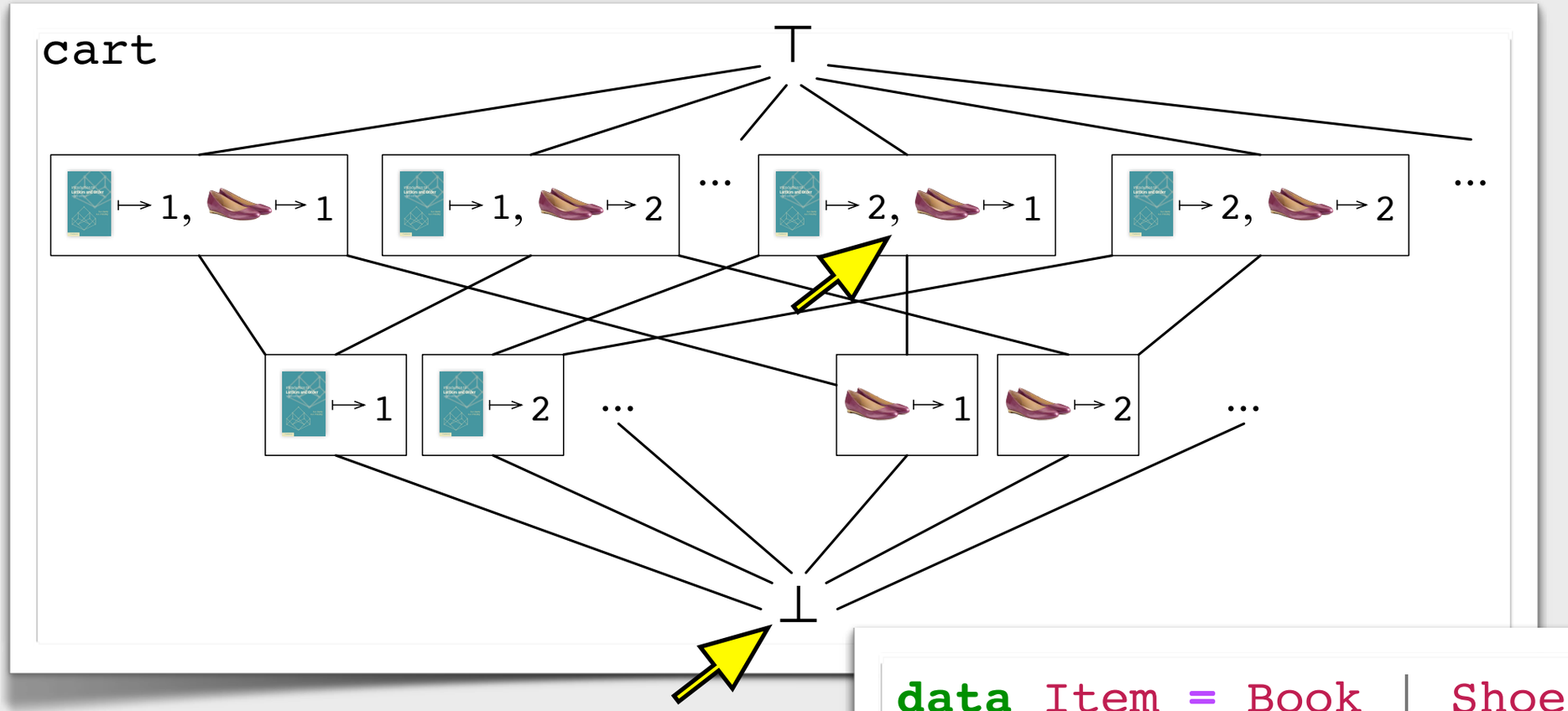

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



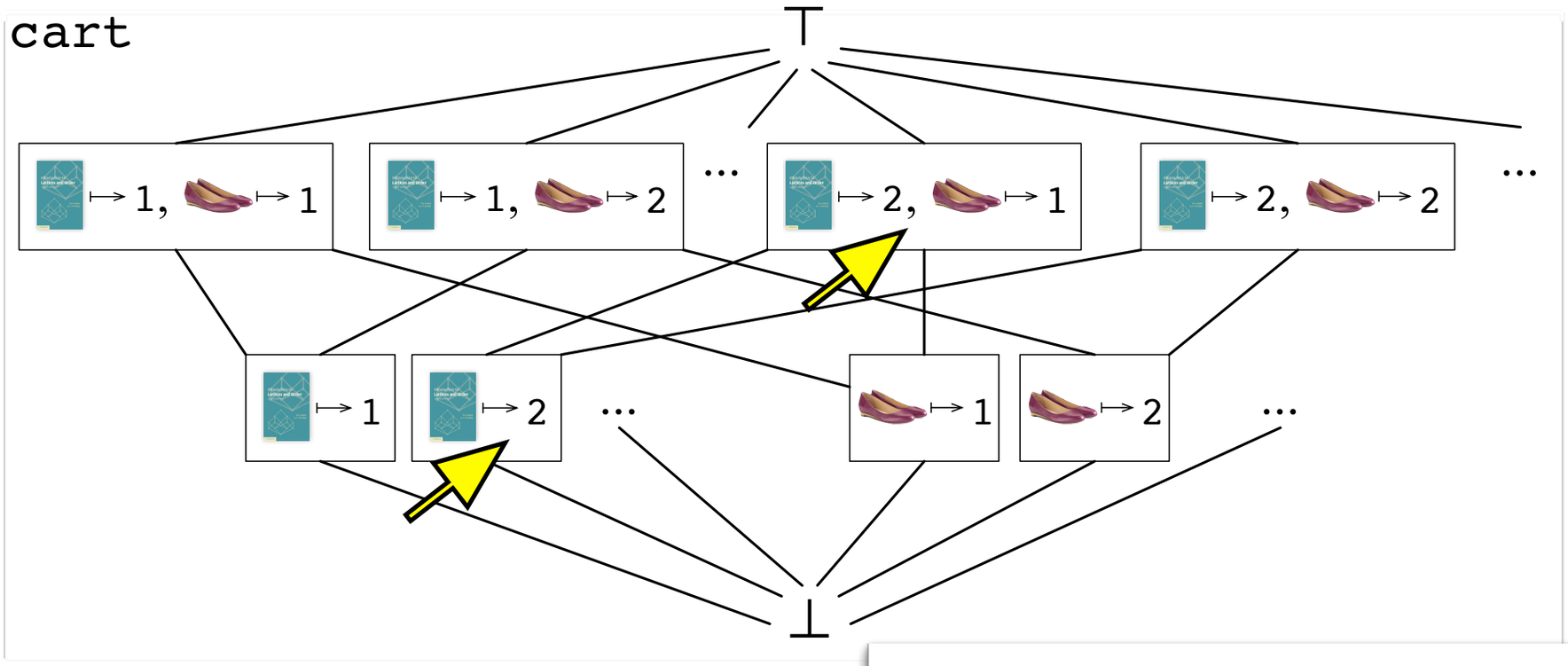

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



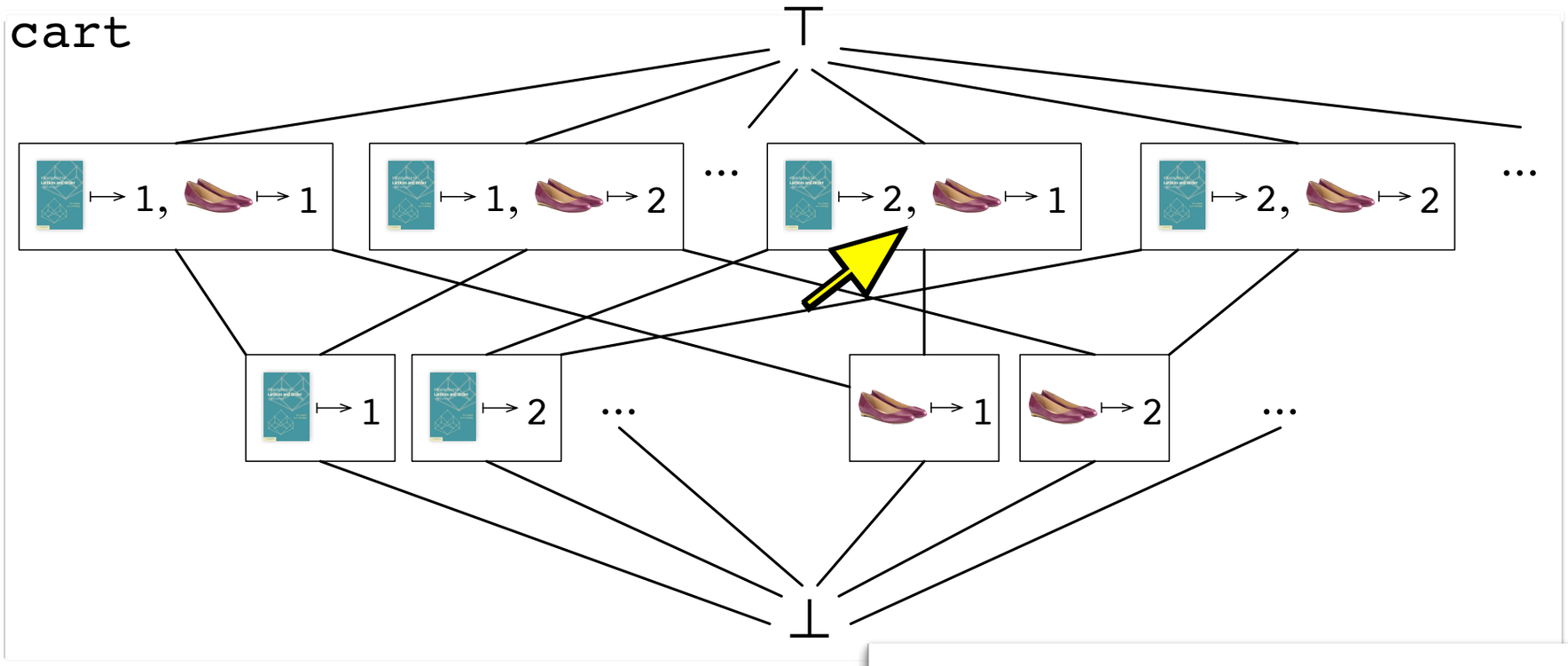

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



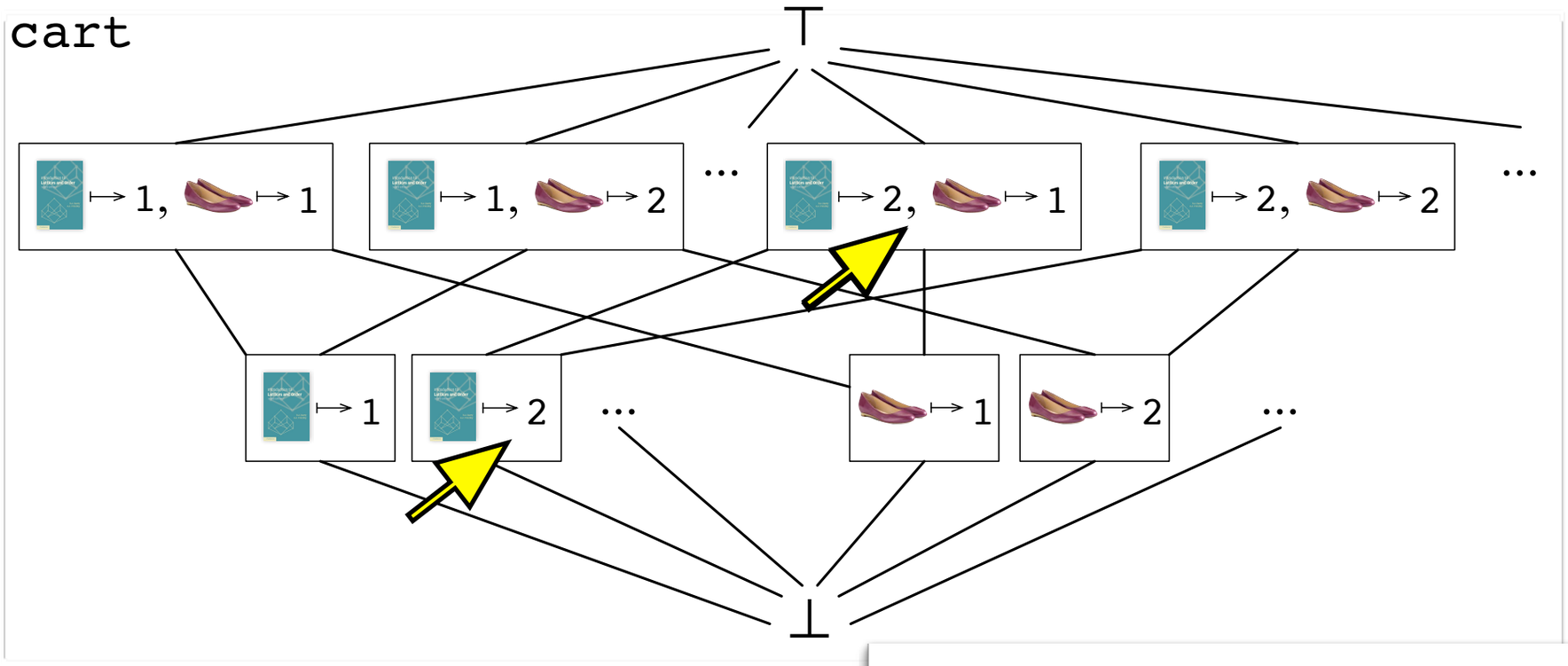

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



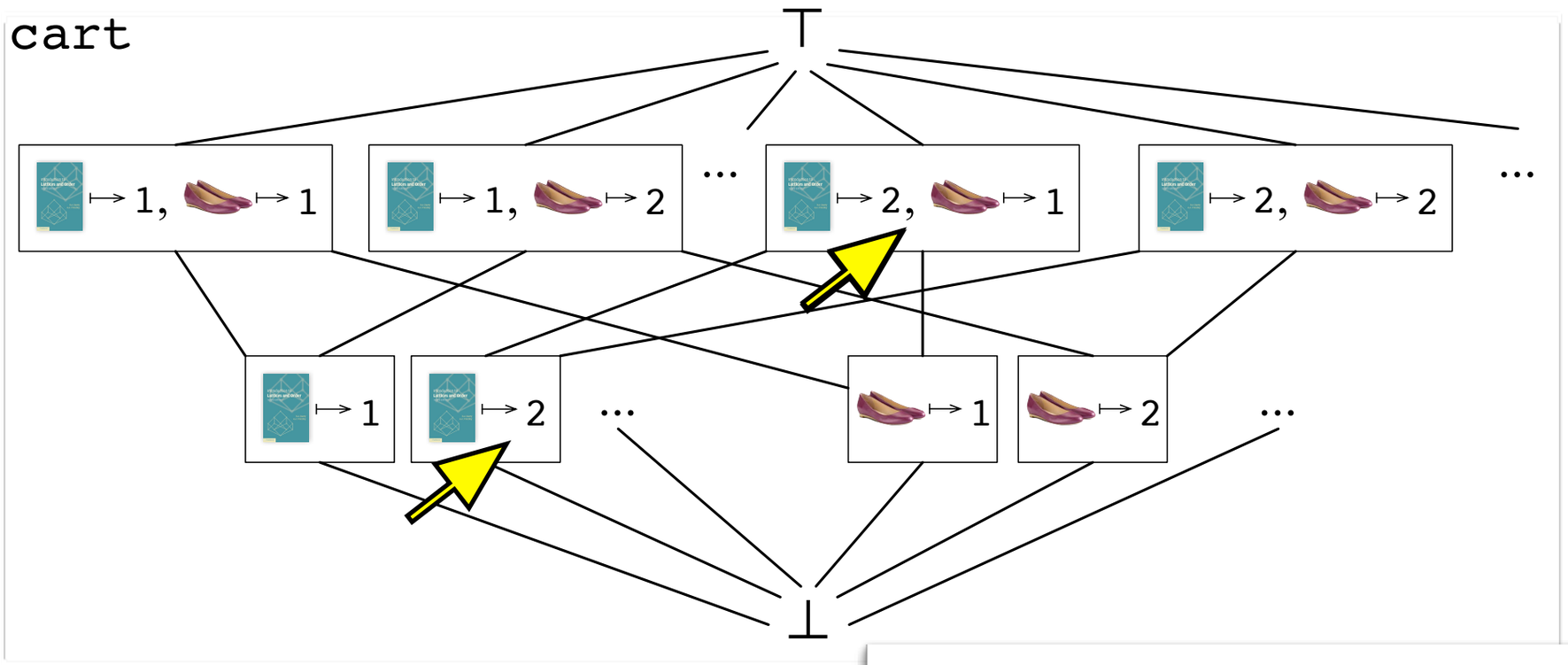

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```




```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```

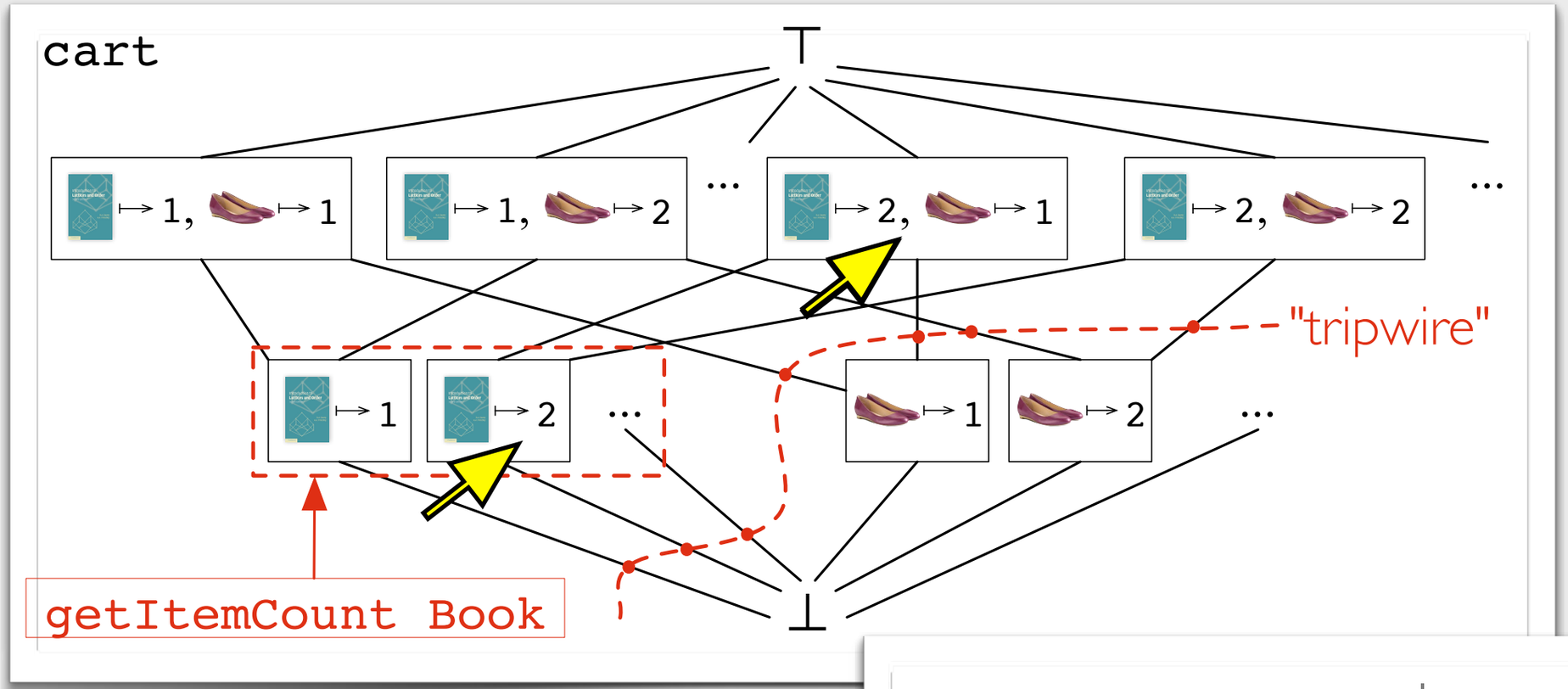



```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



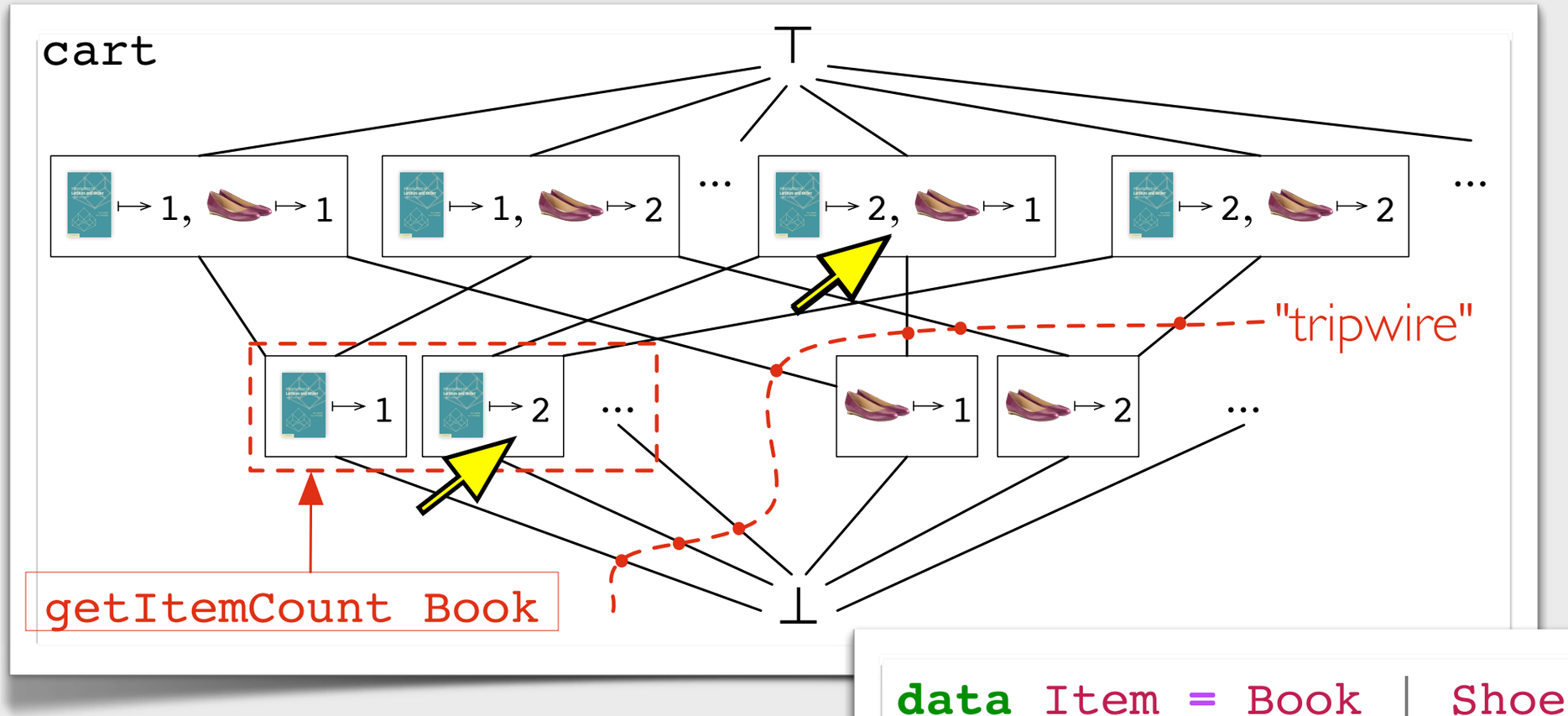

```
{(Book,1), (Book,2), ...}
```

```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```




```
{(Book,1), (Book,2), ...}
```

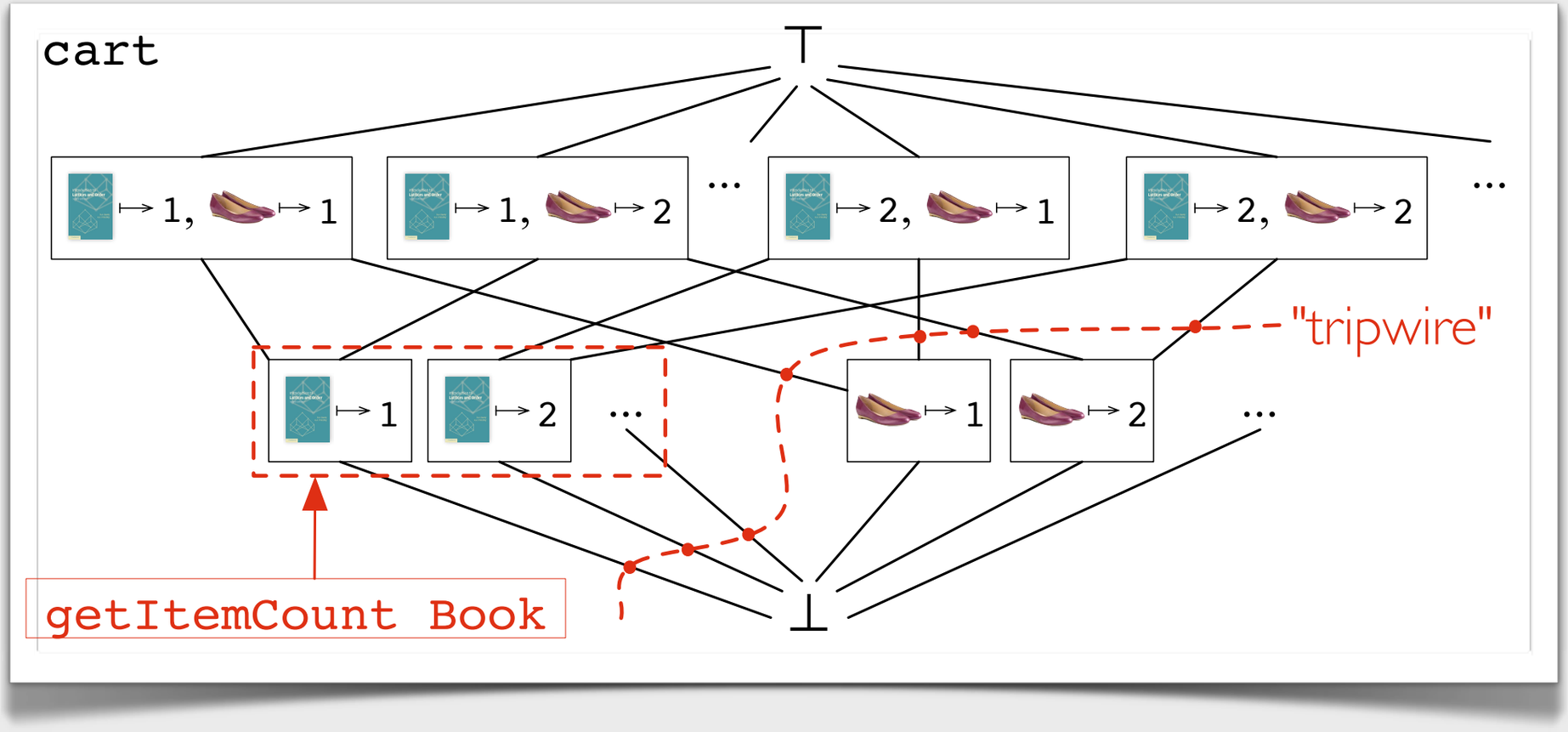
```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



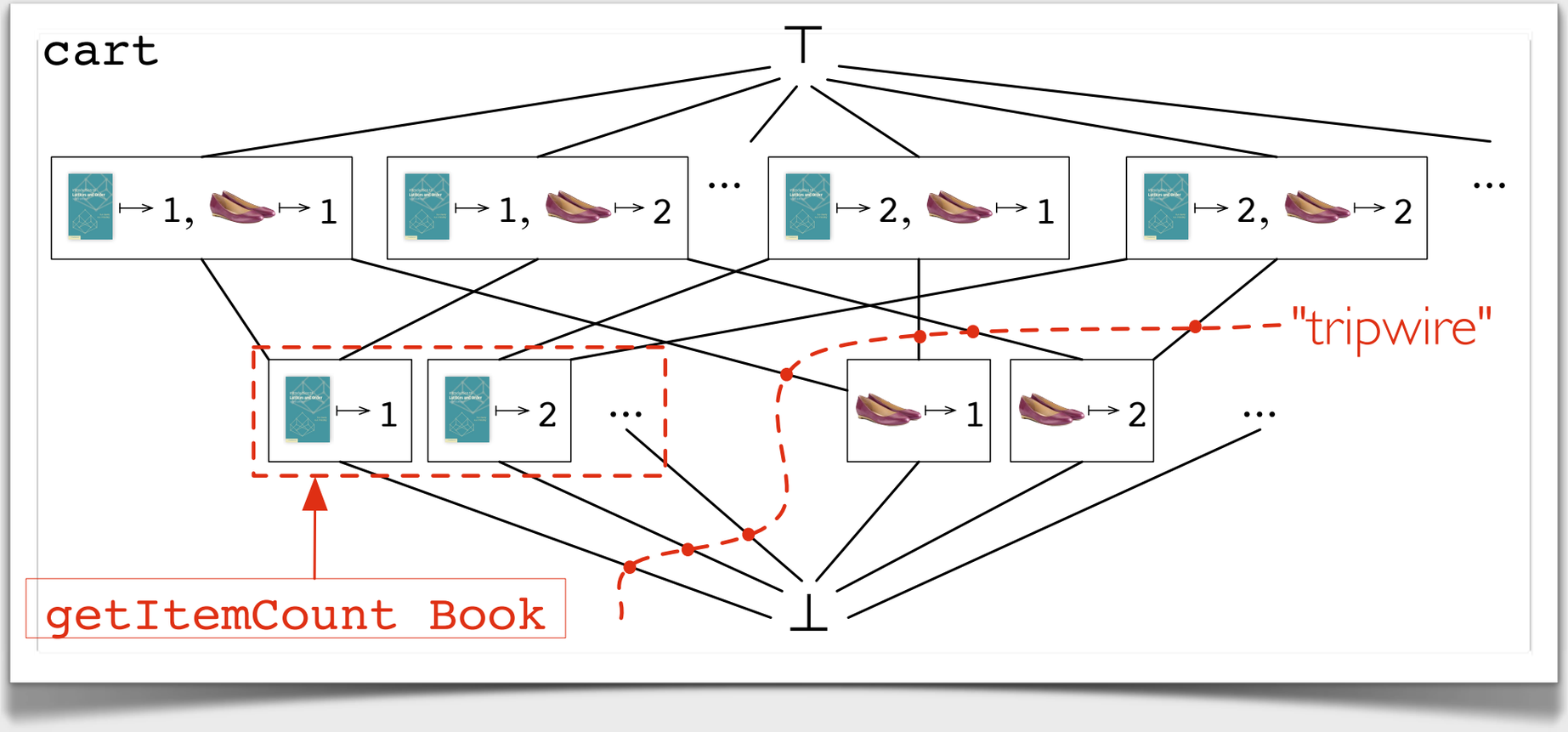

```
{(Book,1), (Book,2), ...}
```

The threshold set must be *pairwise incompatible*

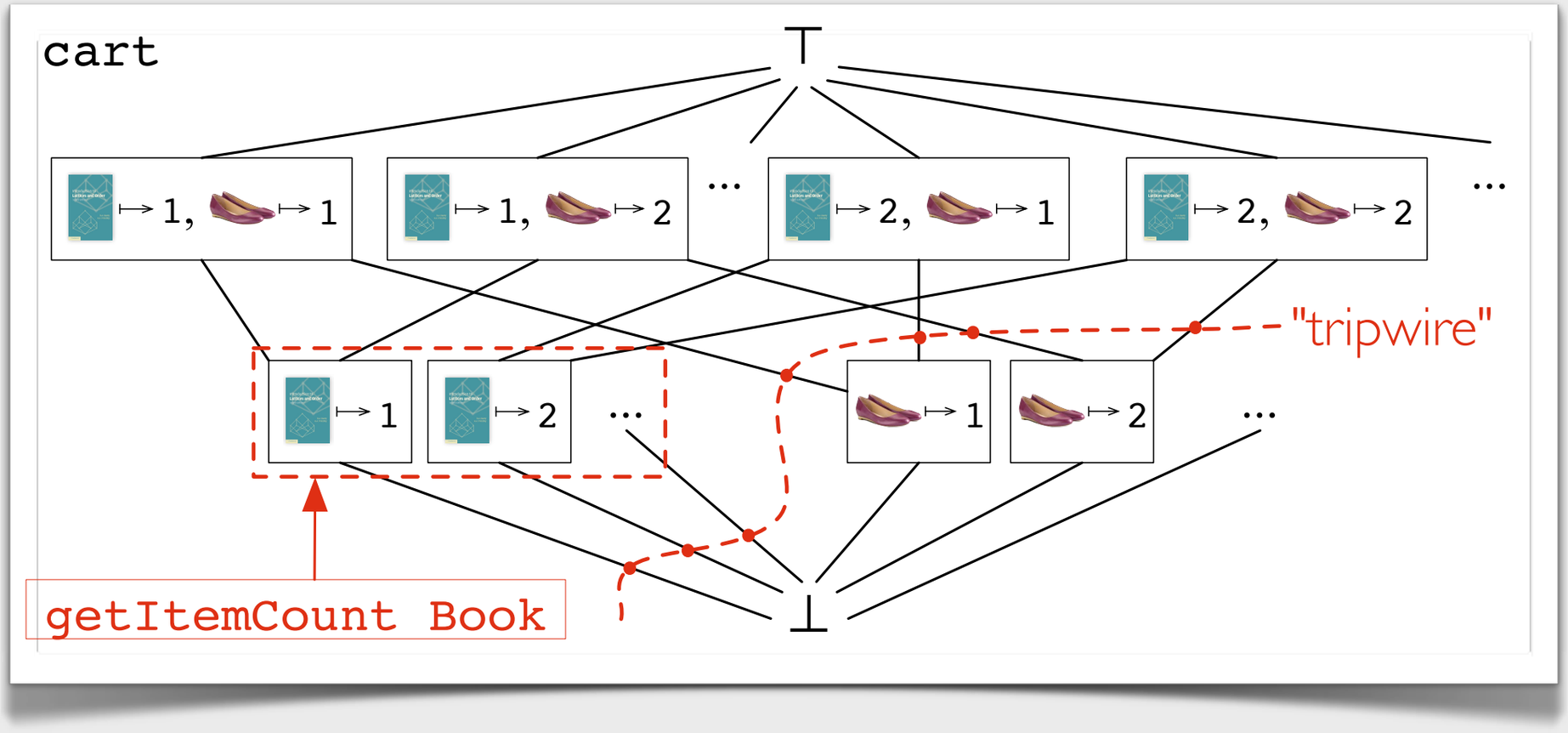
```haskell
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
```



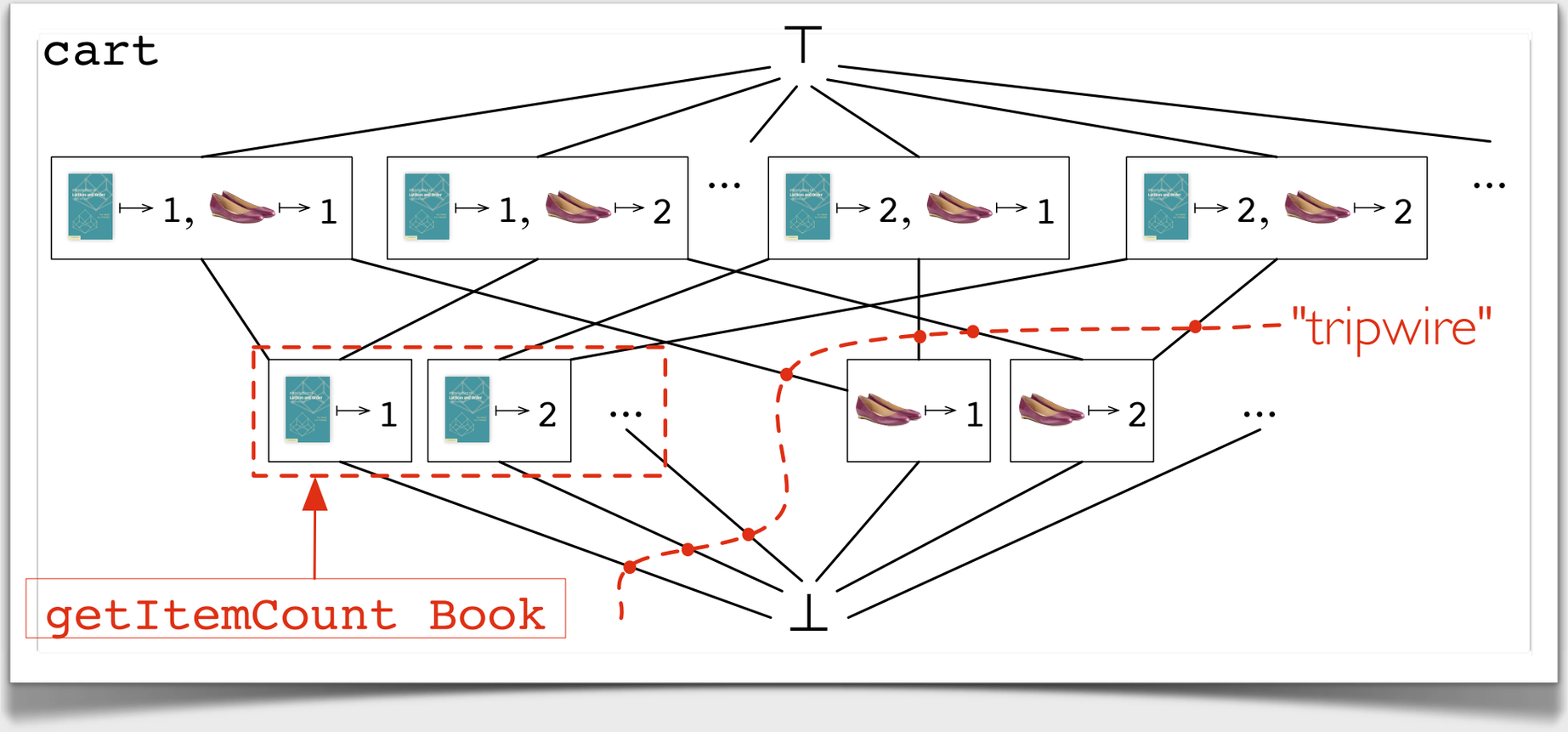
✖ Can't see the exact, complete contents of the cart



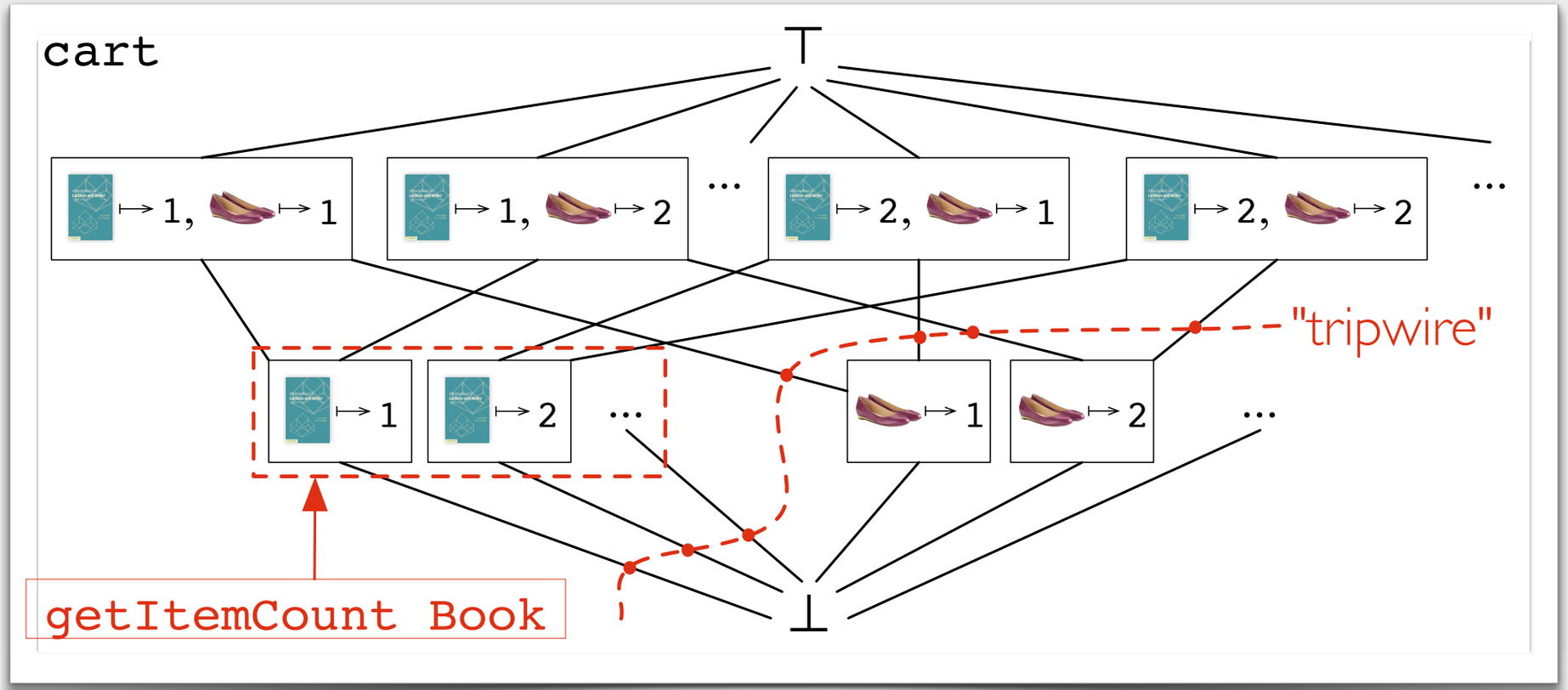
- ✗ Can't see the exact, complete contents of the cart
- ✗ Can't iterate over the items in the cart



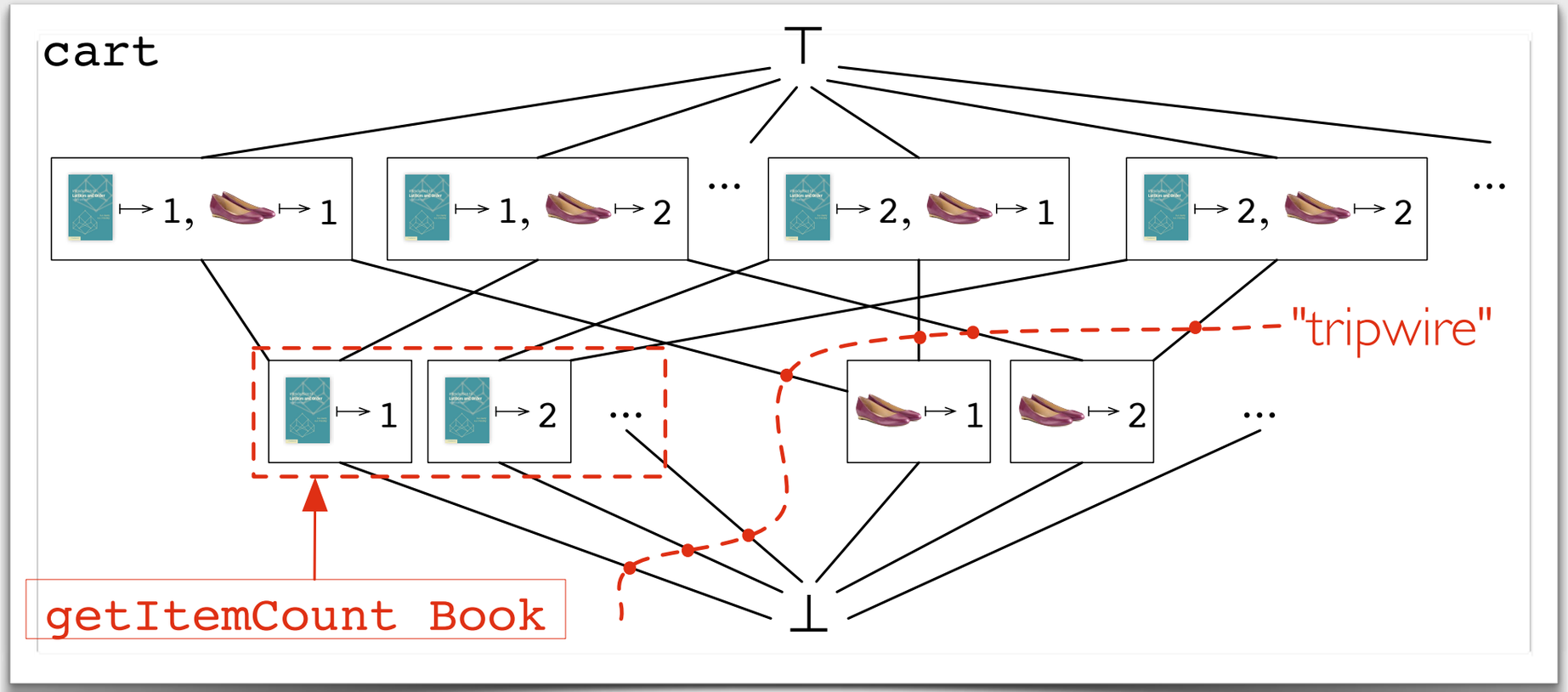
- ✗ Can't see the exact, complete contents of the cart
- ✗ Can't iterate over the items in the cart
- ✗ Can't determine if an item *isn't* in the cart



- ✗ Can't see the exact, complete contents of the cart
- ✗ Can't iterate over the items in the cart
- ✗ Can't determine if an item *isn't* in the cart
- ✗ Can't react to writes that we weren't expecting



- ✓ Can see the exact, complete contents of the cart
- ✓ Can iterate over the items in the cart
- ✓ Can determine if an item *isn't* in the cart
- ✓ Can react to writes that we weren't expecting



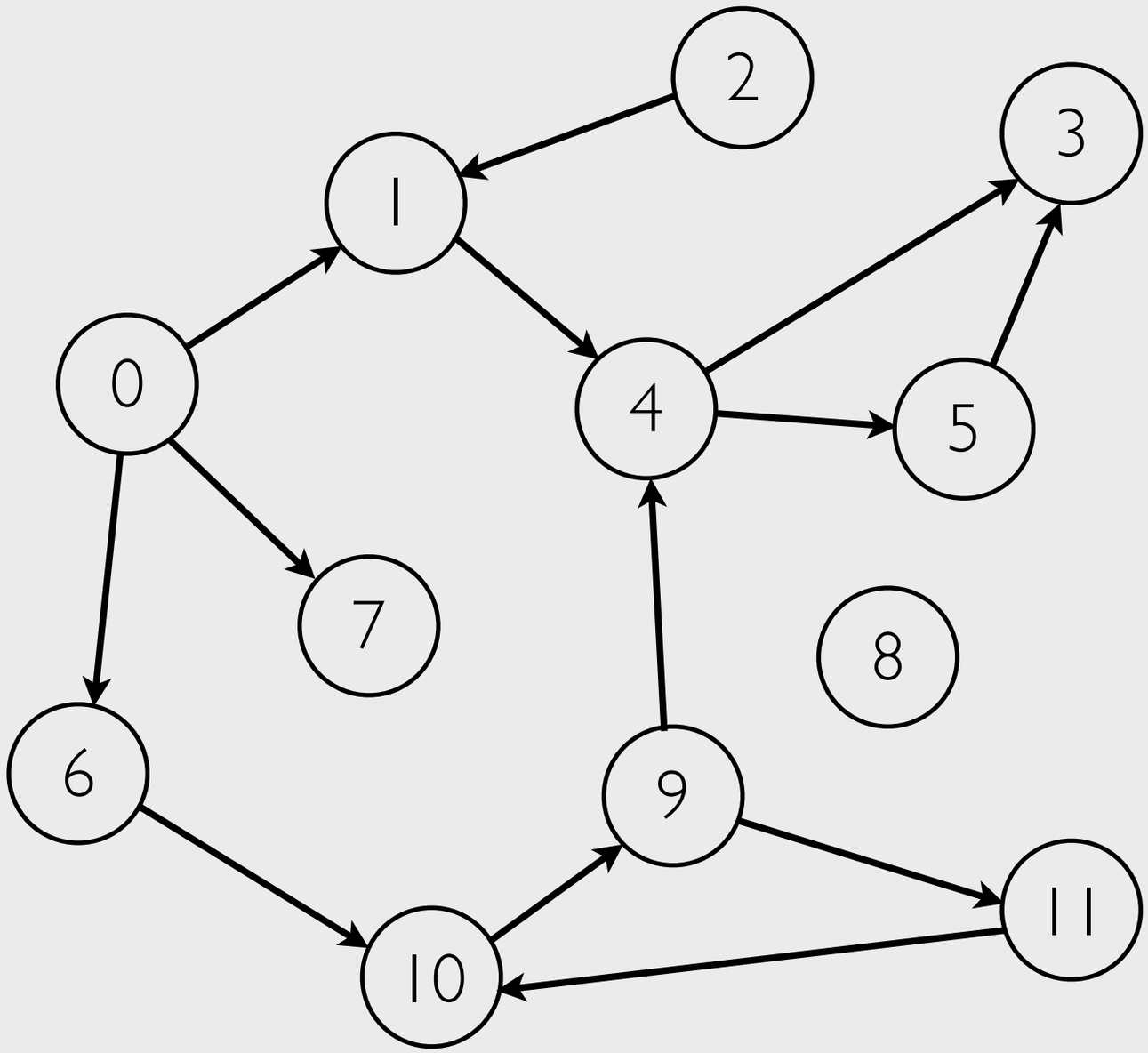
- ✓ Can see the exact, complete contents of the cart
- ✓ Can iterate over the items in the cart
- ✓ Can determine if an item *isn't* in the cart
- ✓ Can react to writes that we weren't expecting

} handlers, quiescence, freezing

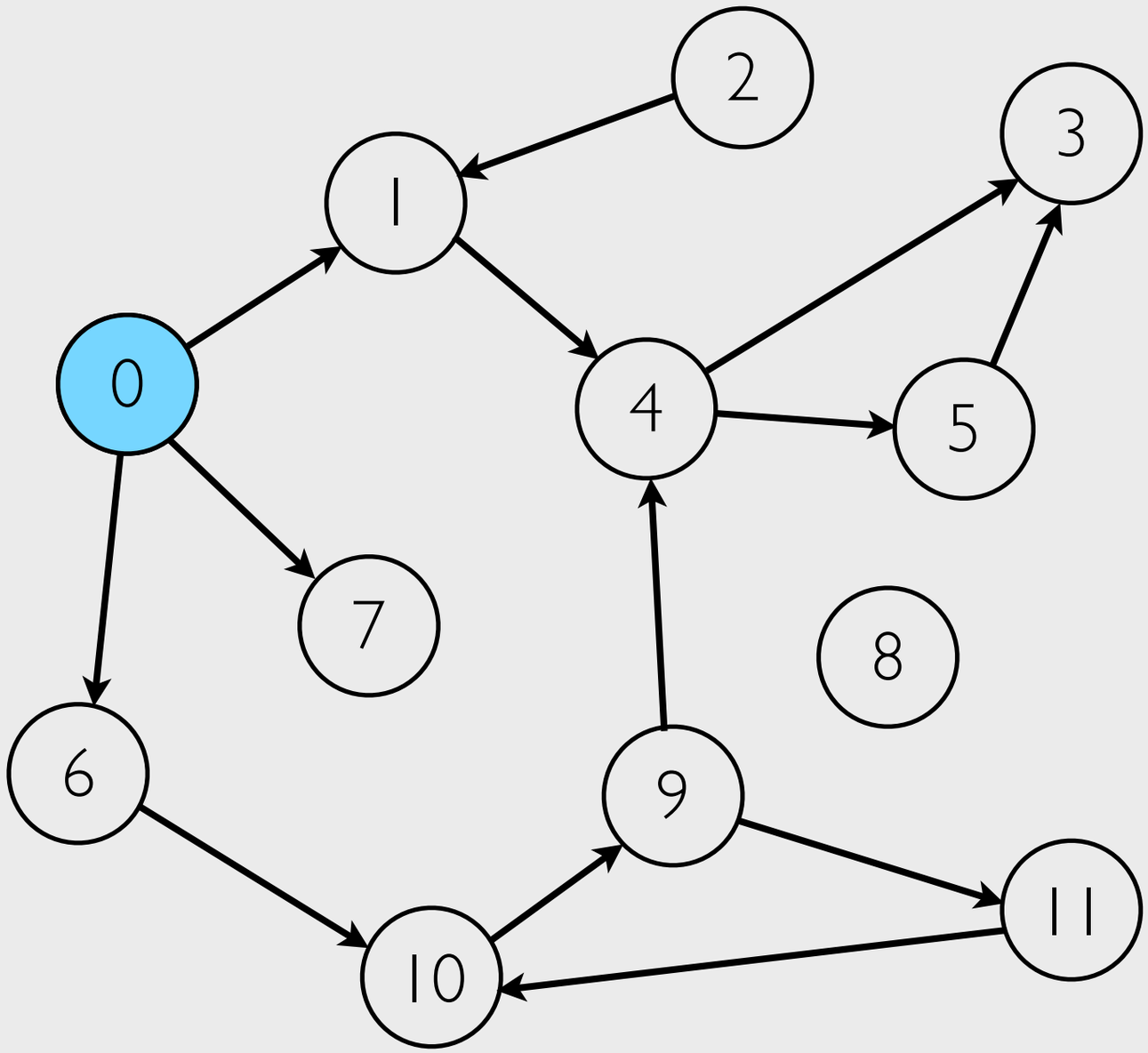
Freeze After Writing

- LVars
- Quasi-Deterministic

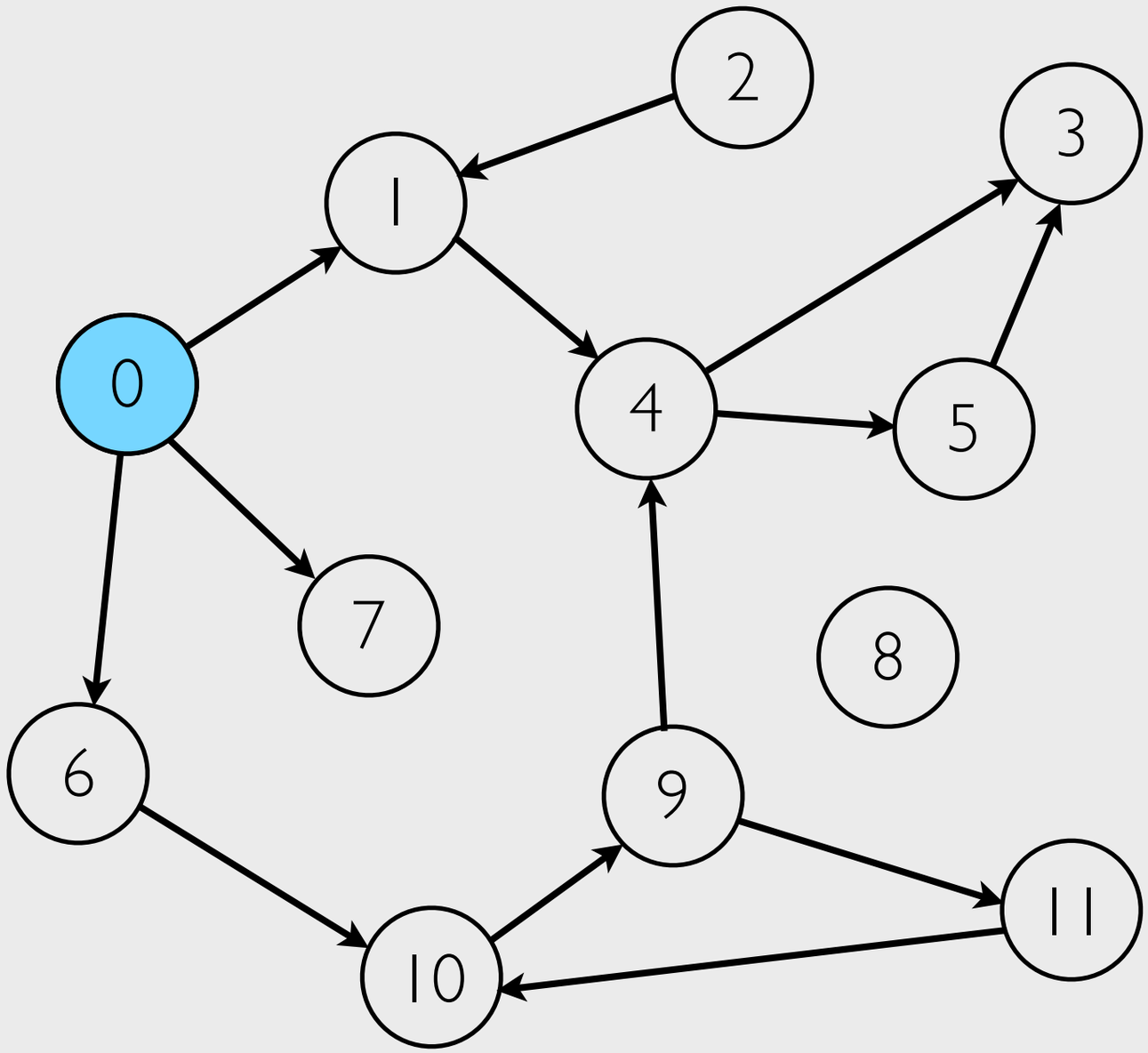
Parallel Programming



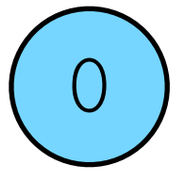
seen nodes

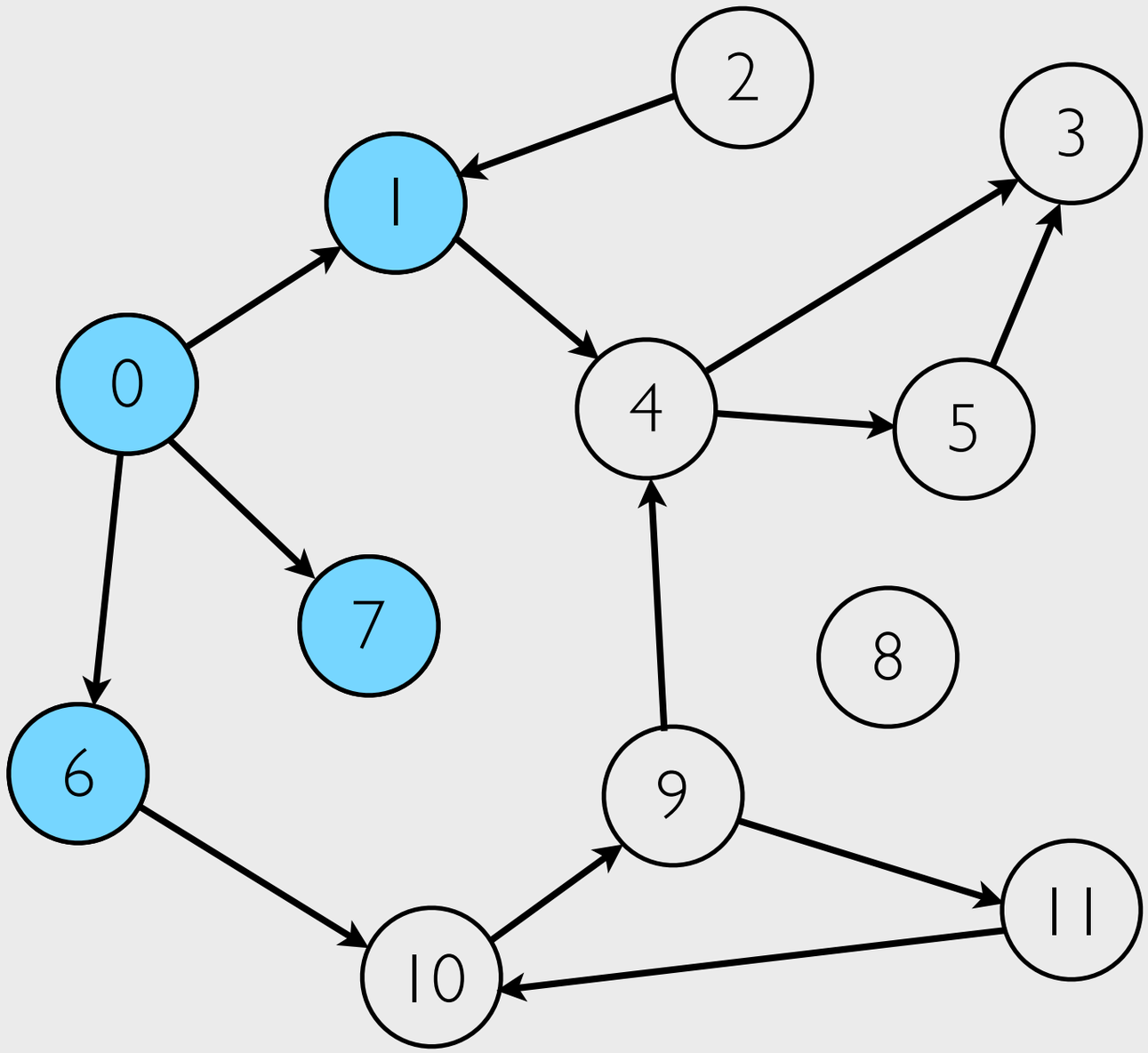


seen nodes

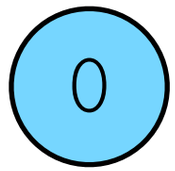


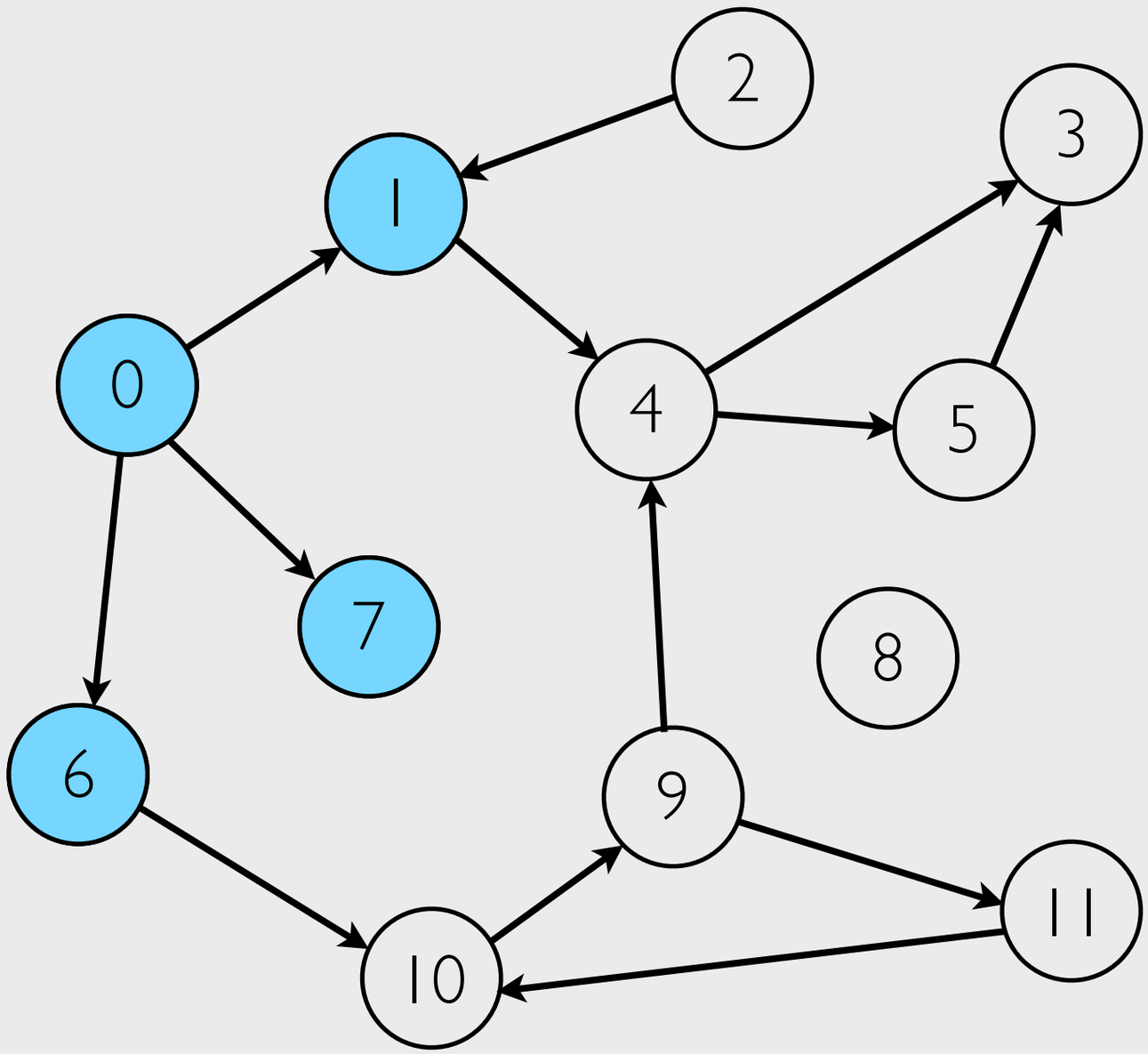
seen nodes



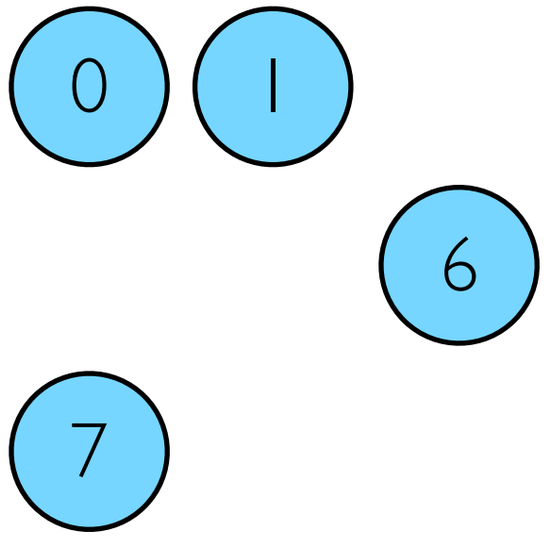


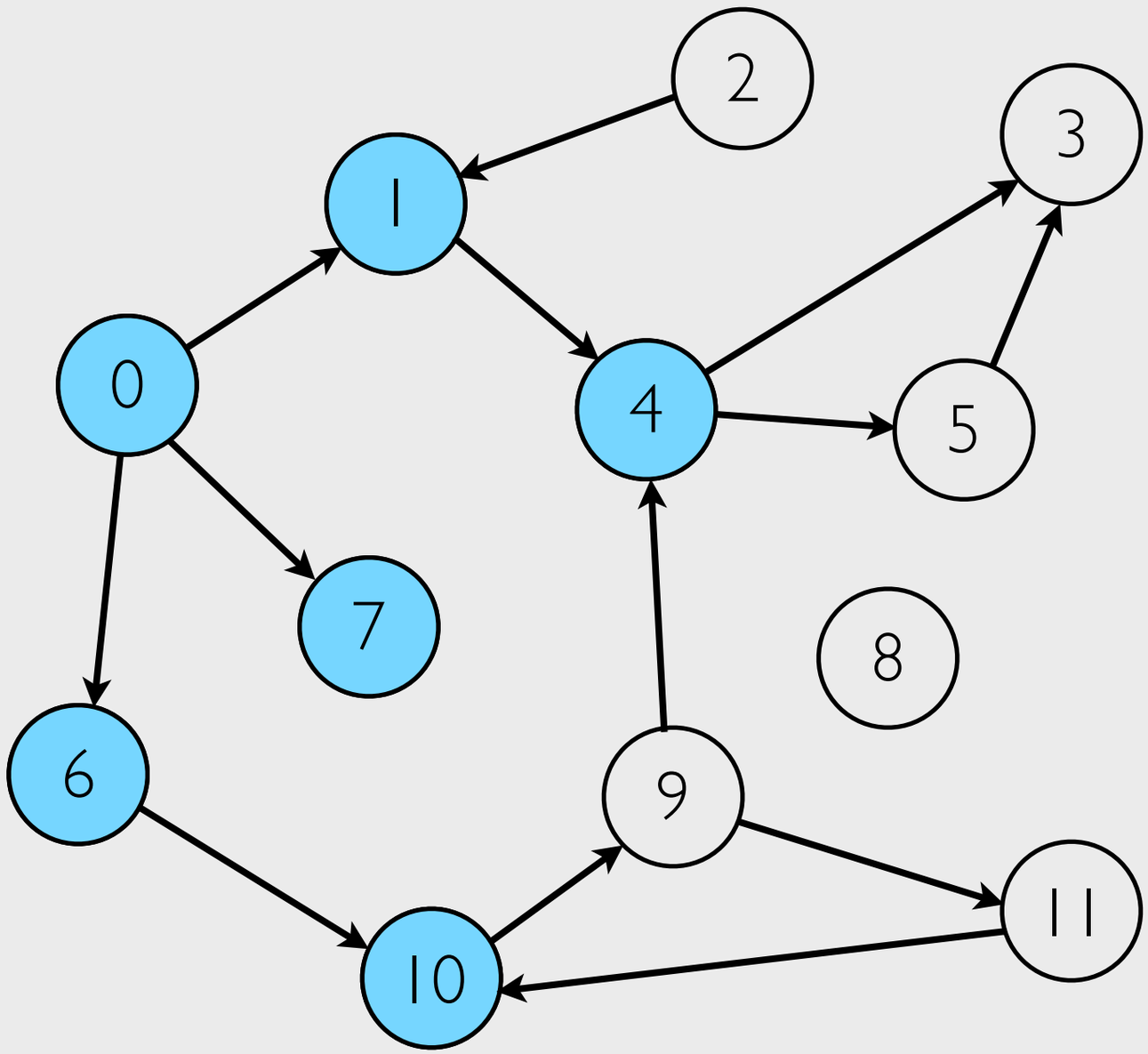
seen nodes

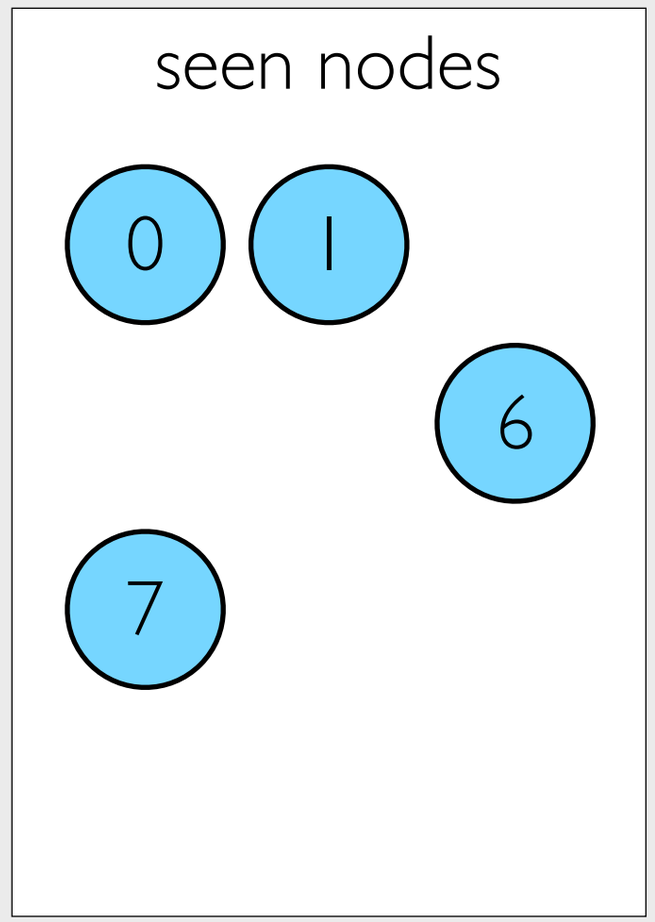


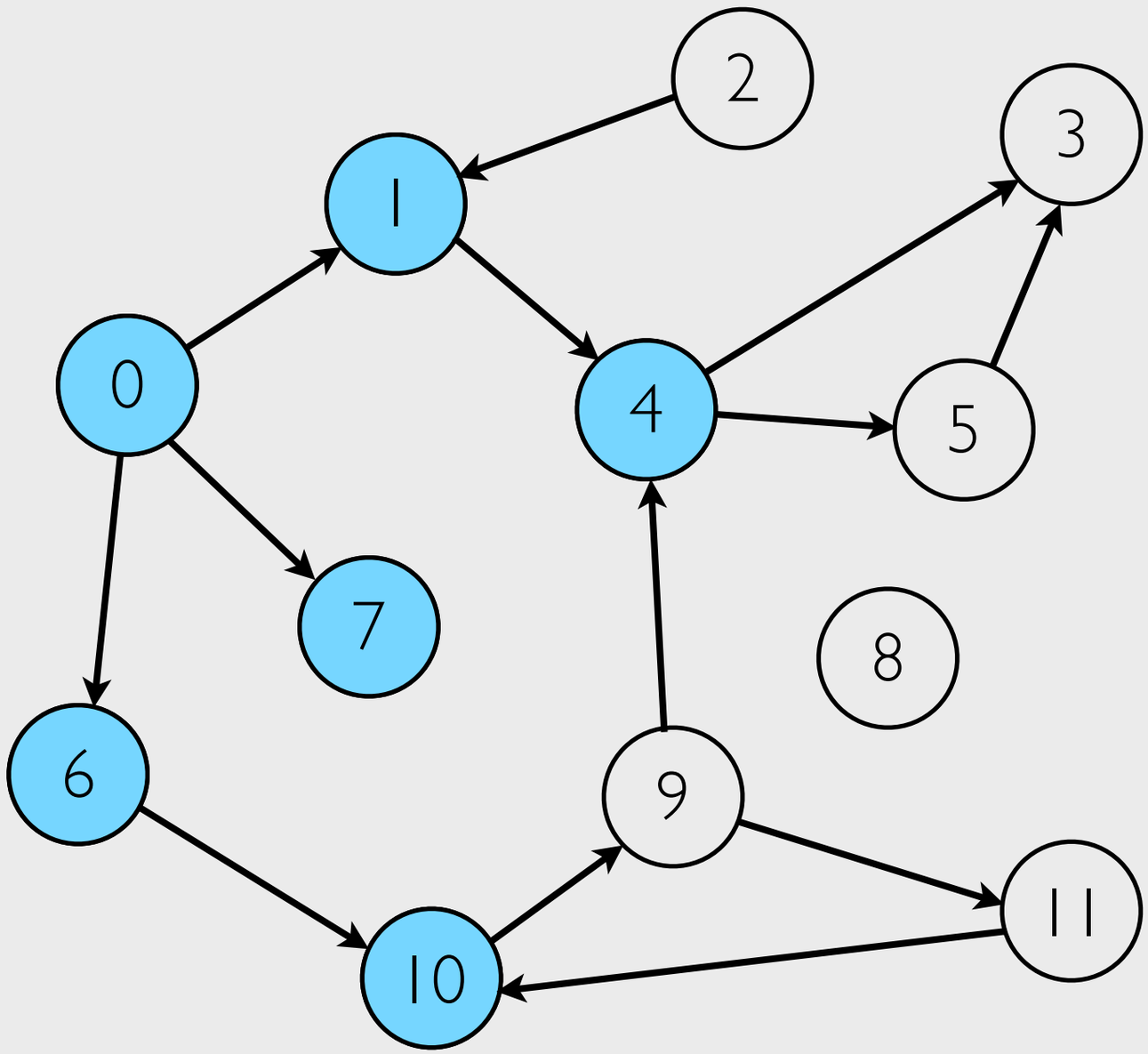


seen nodes

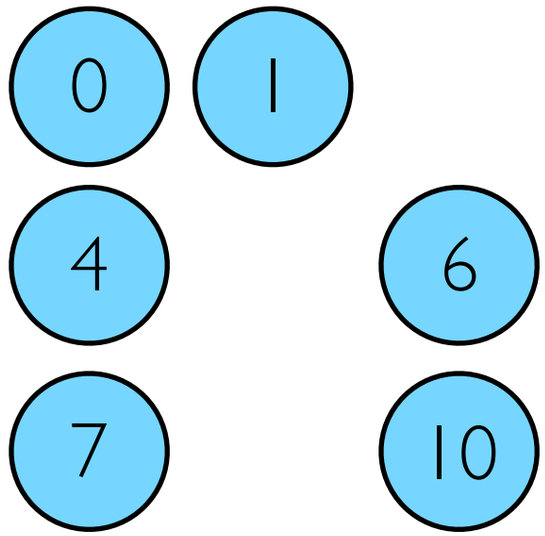


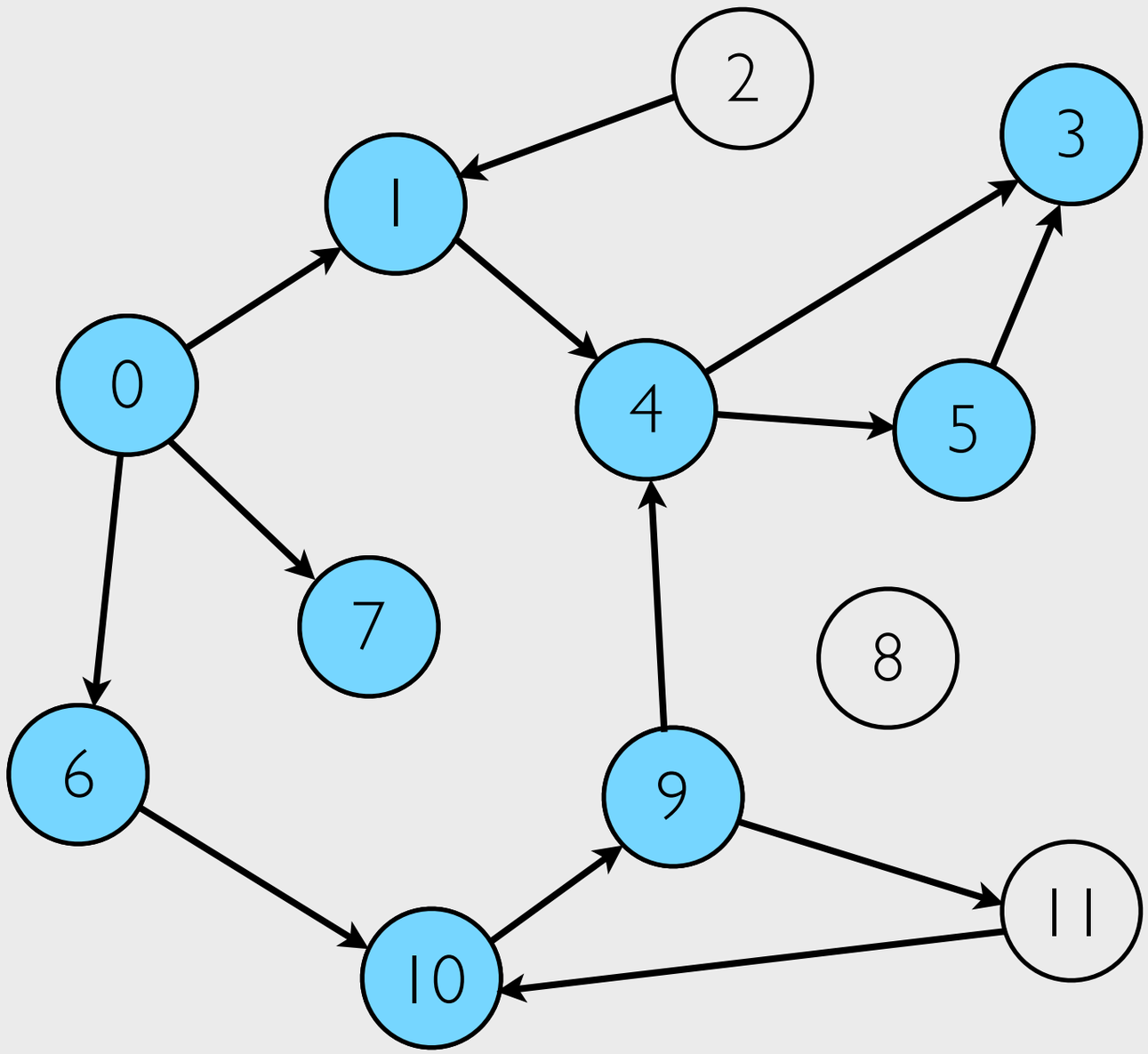

2
1
3
0
4
5
7
8
6
9
11
10


seen nodes
0
1
6
7

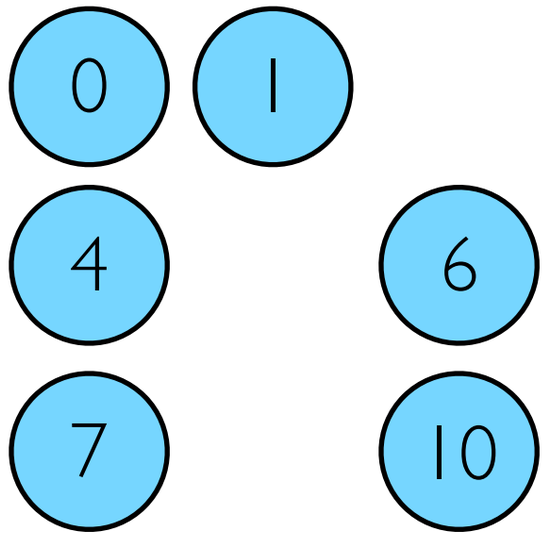


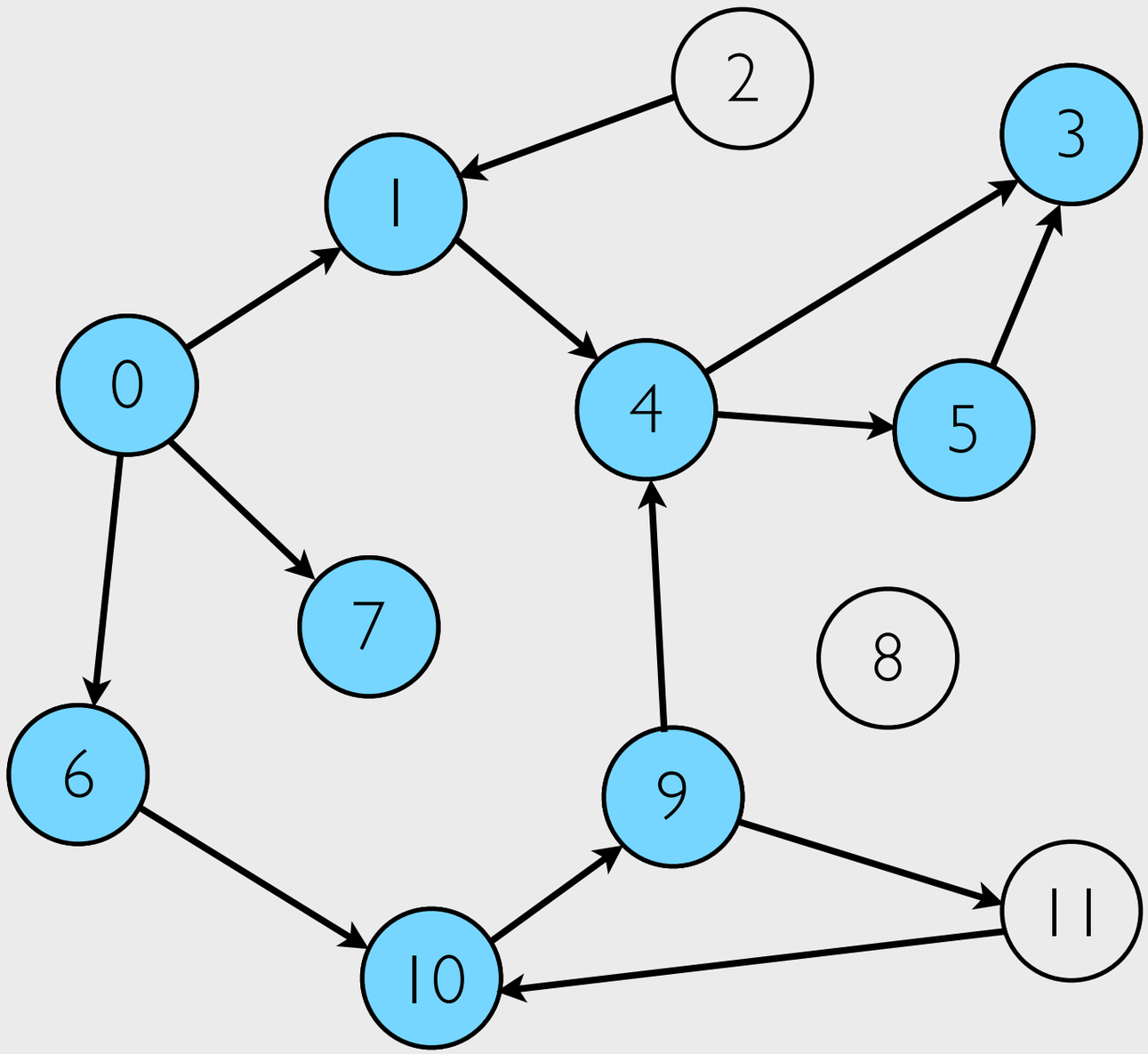
seen nodes



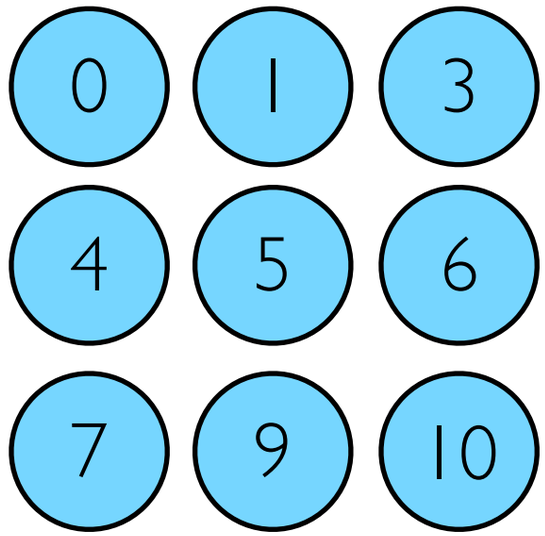


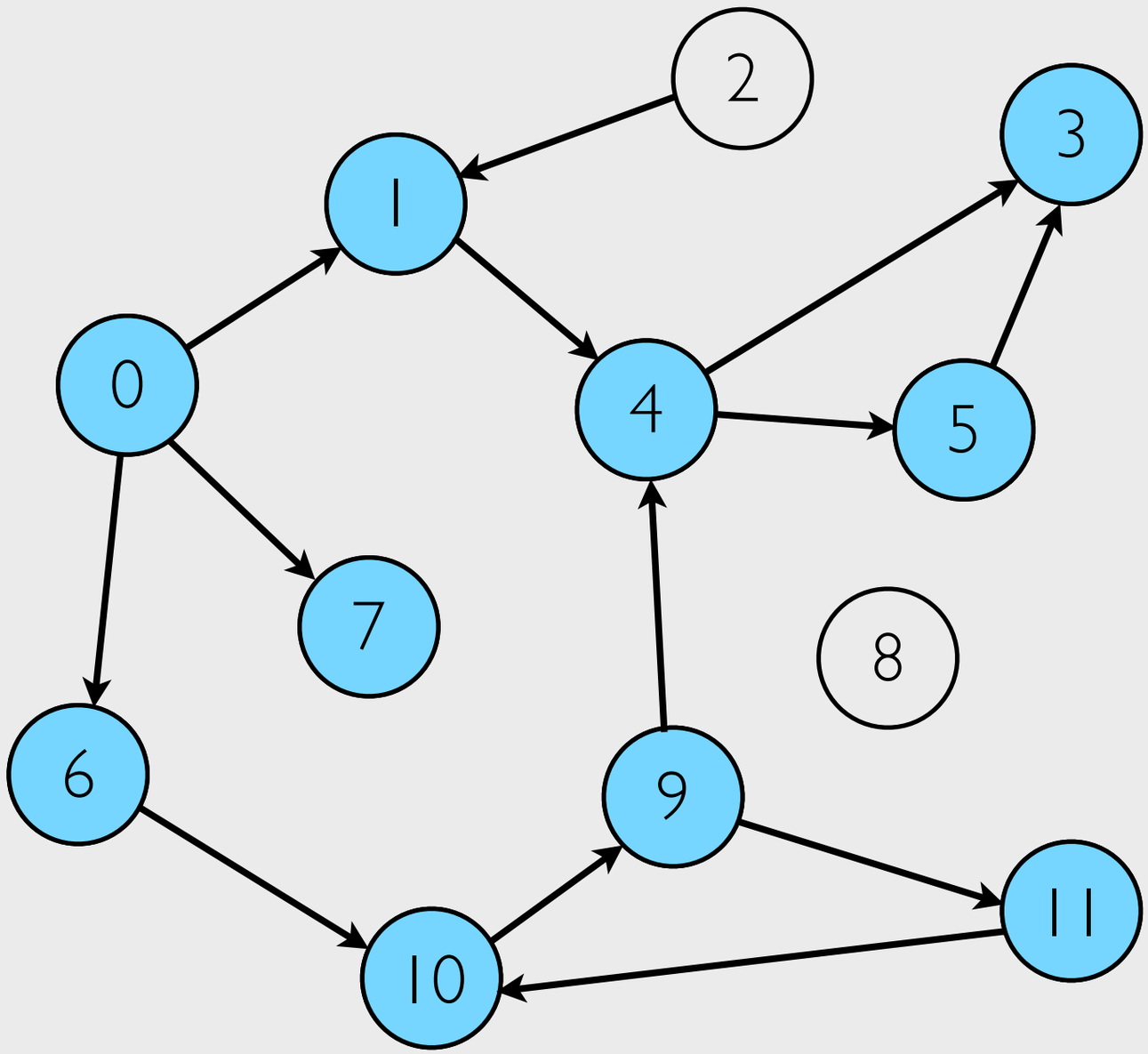
seen nodes



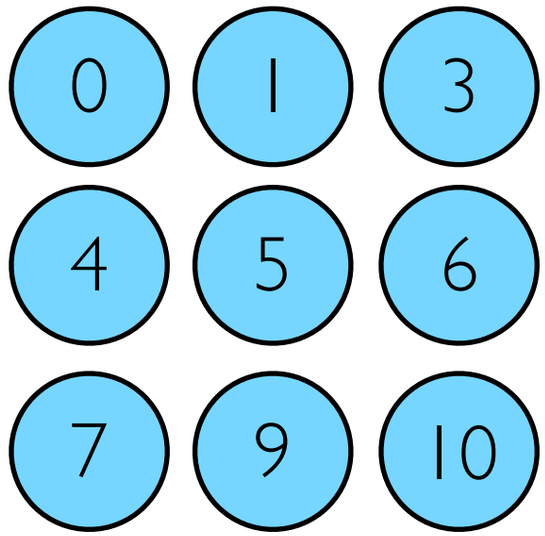


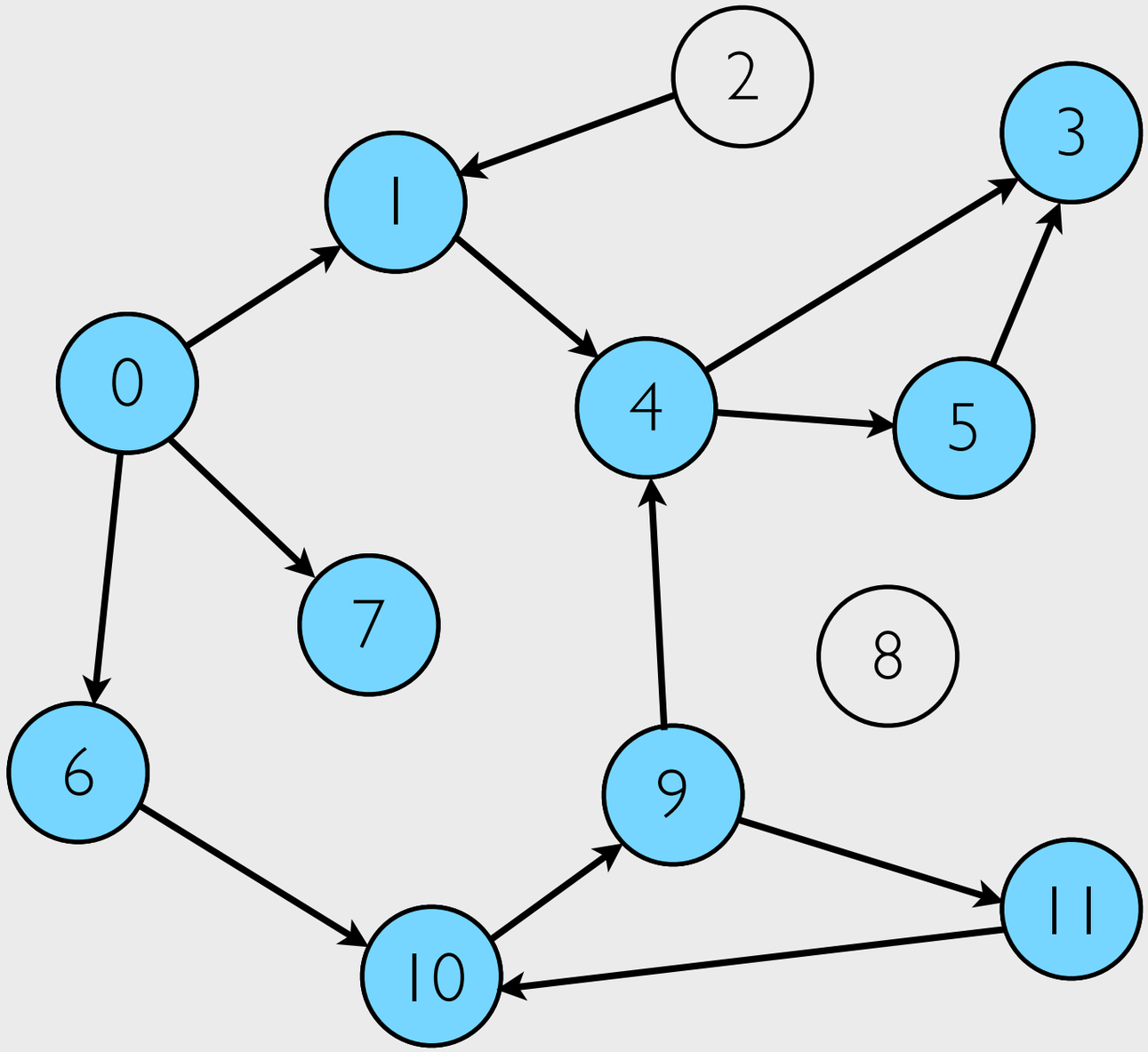
seen nodes



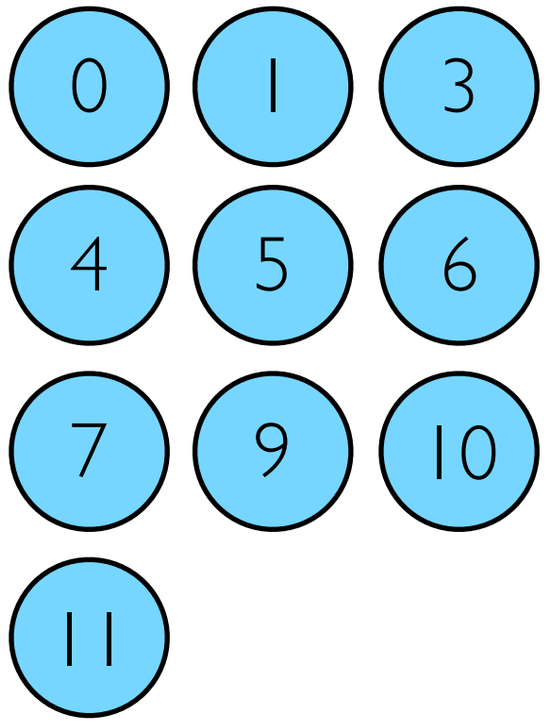


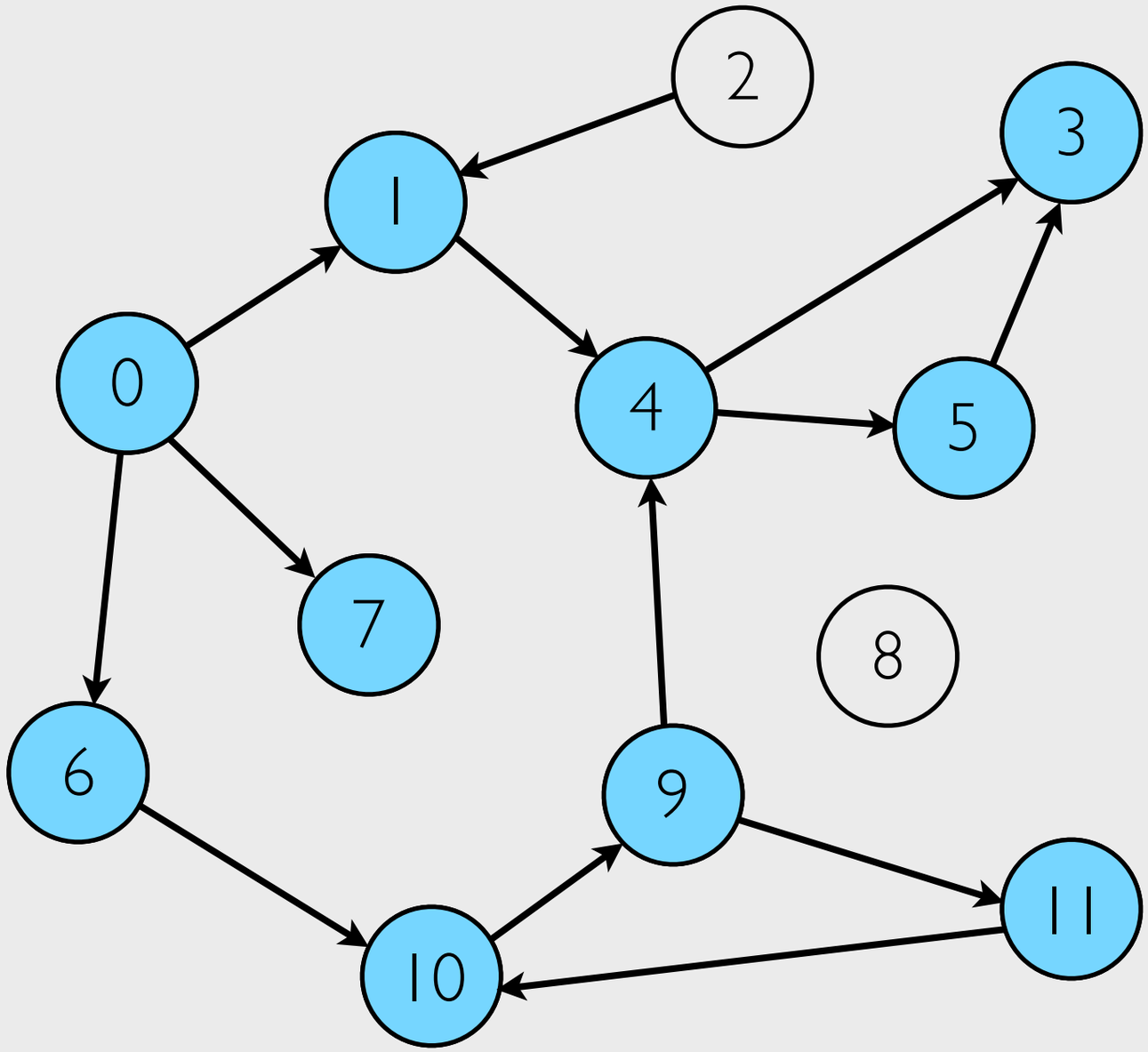
seen nodes



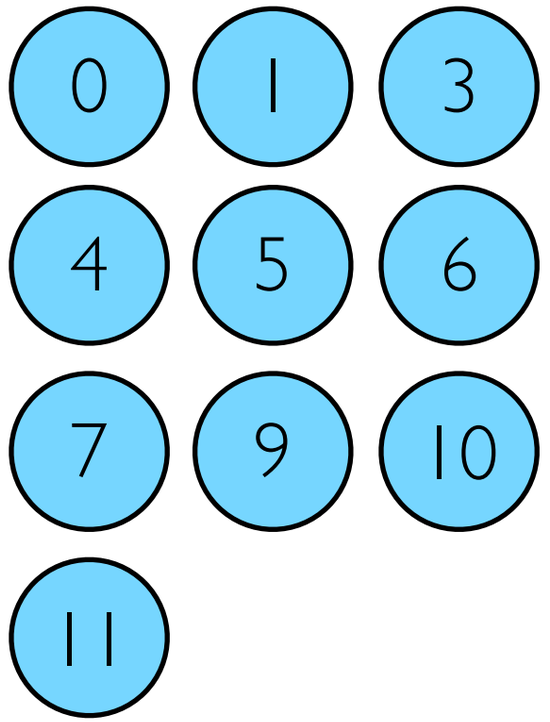


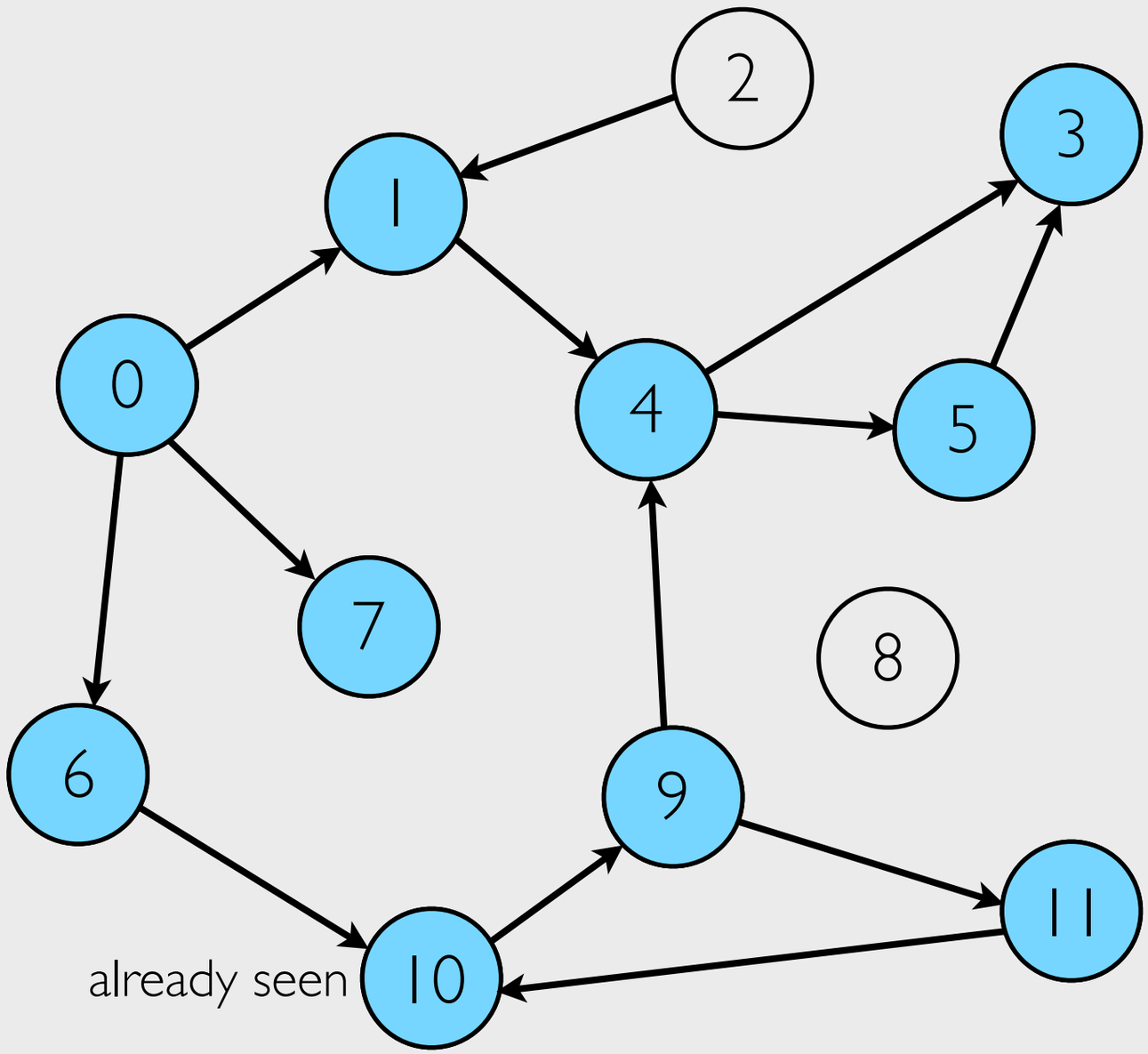
seen nodes



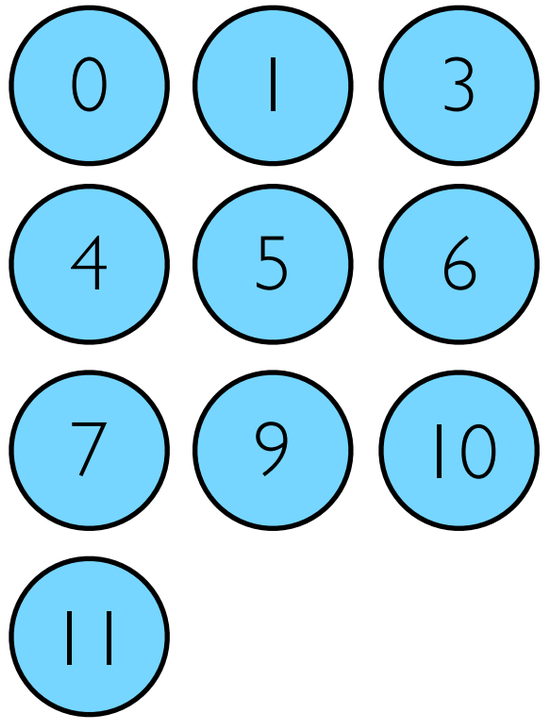


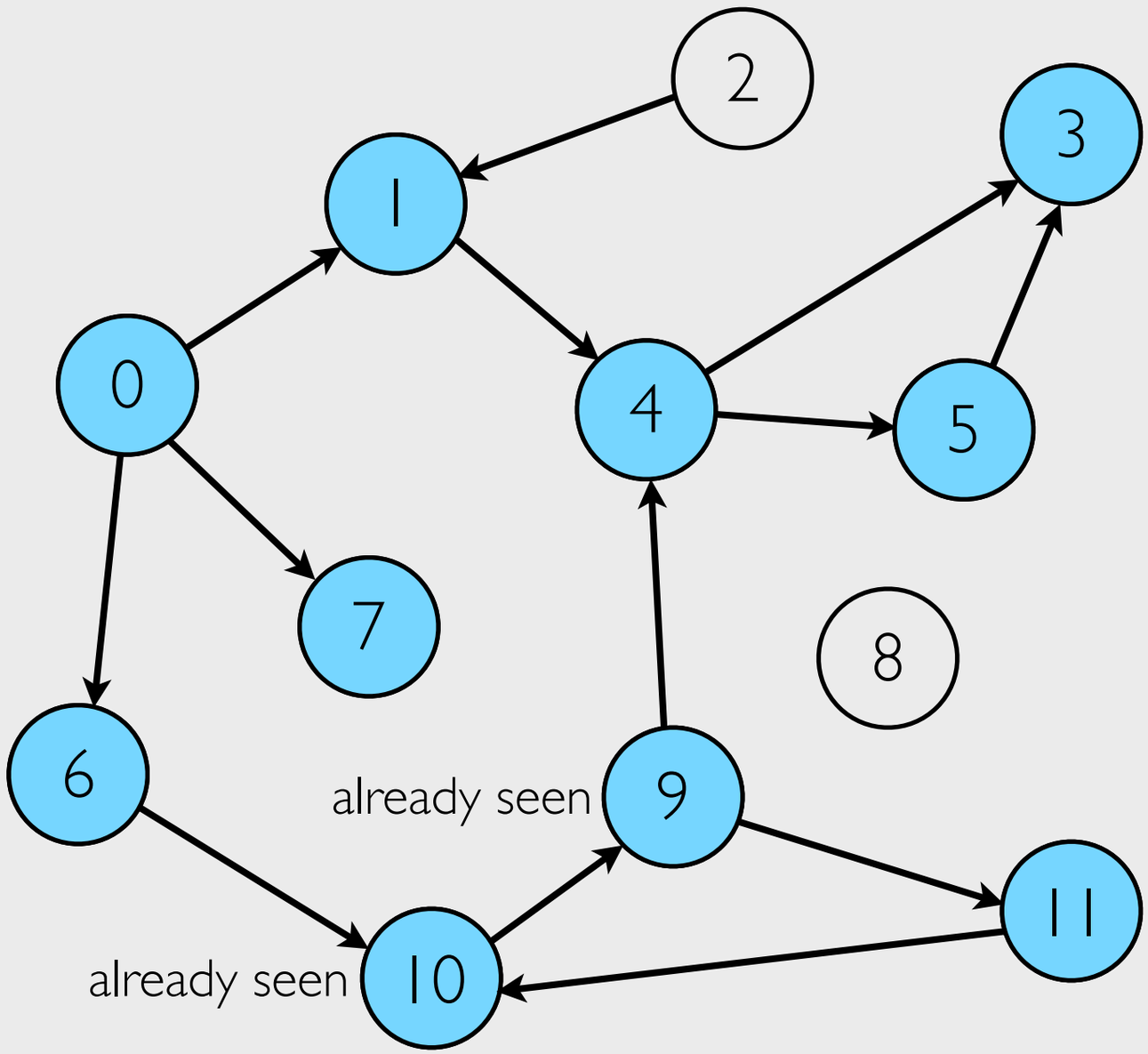
seen nodes



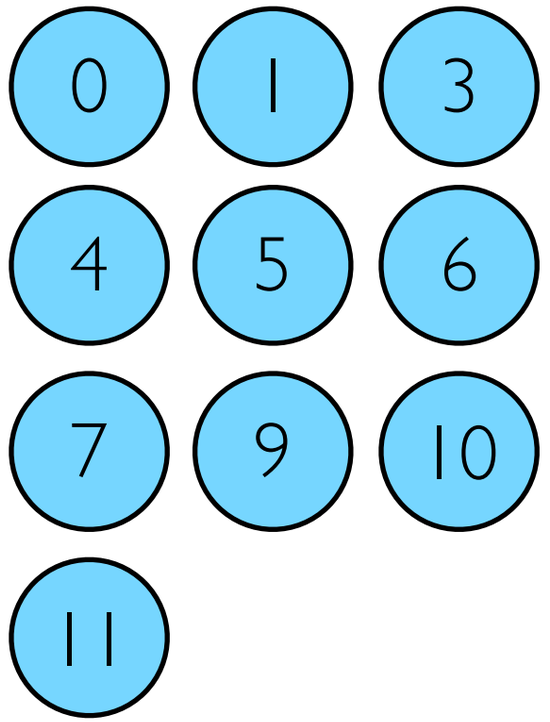


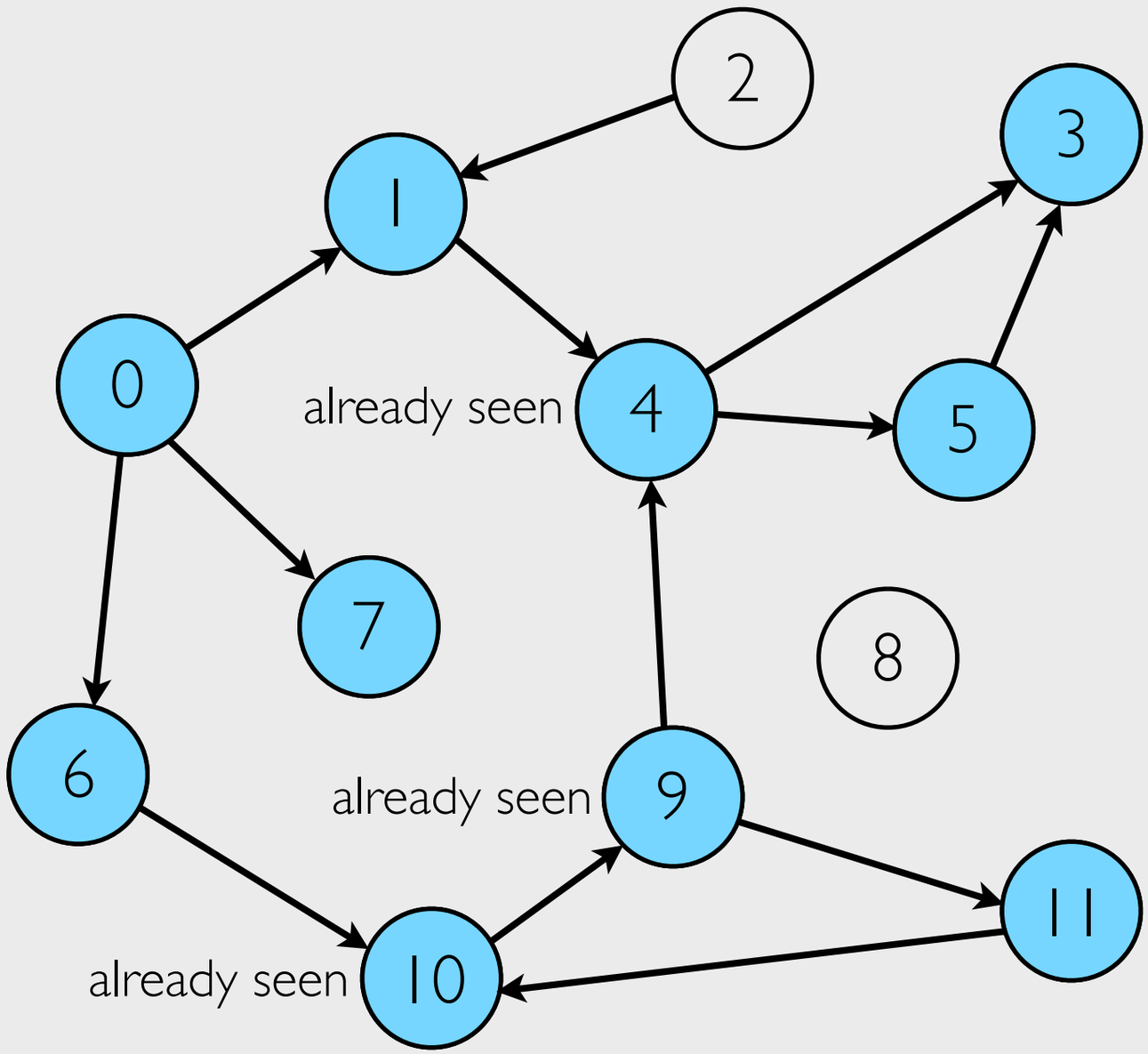
seen nodes

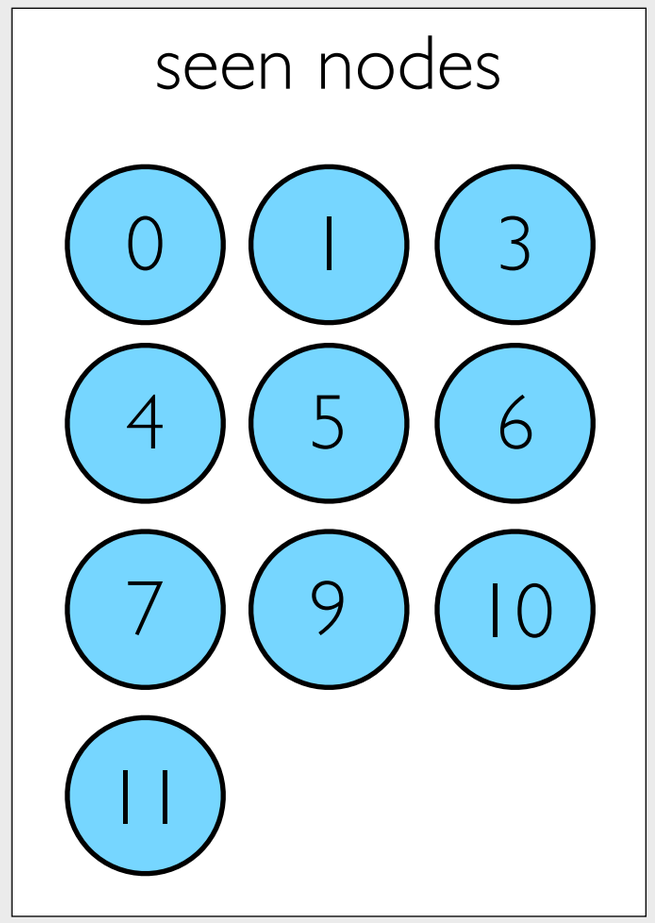


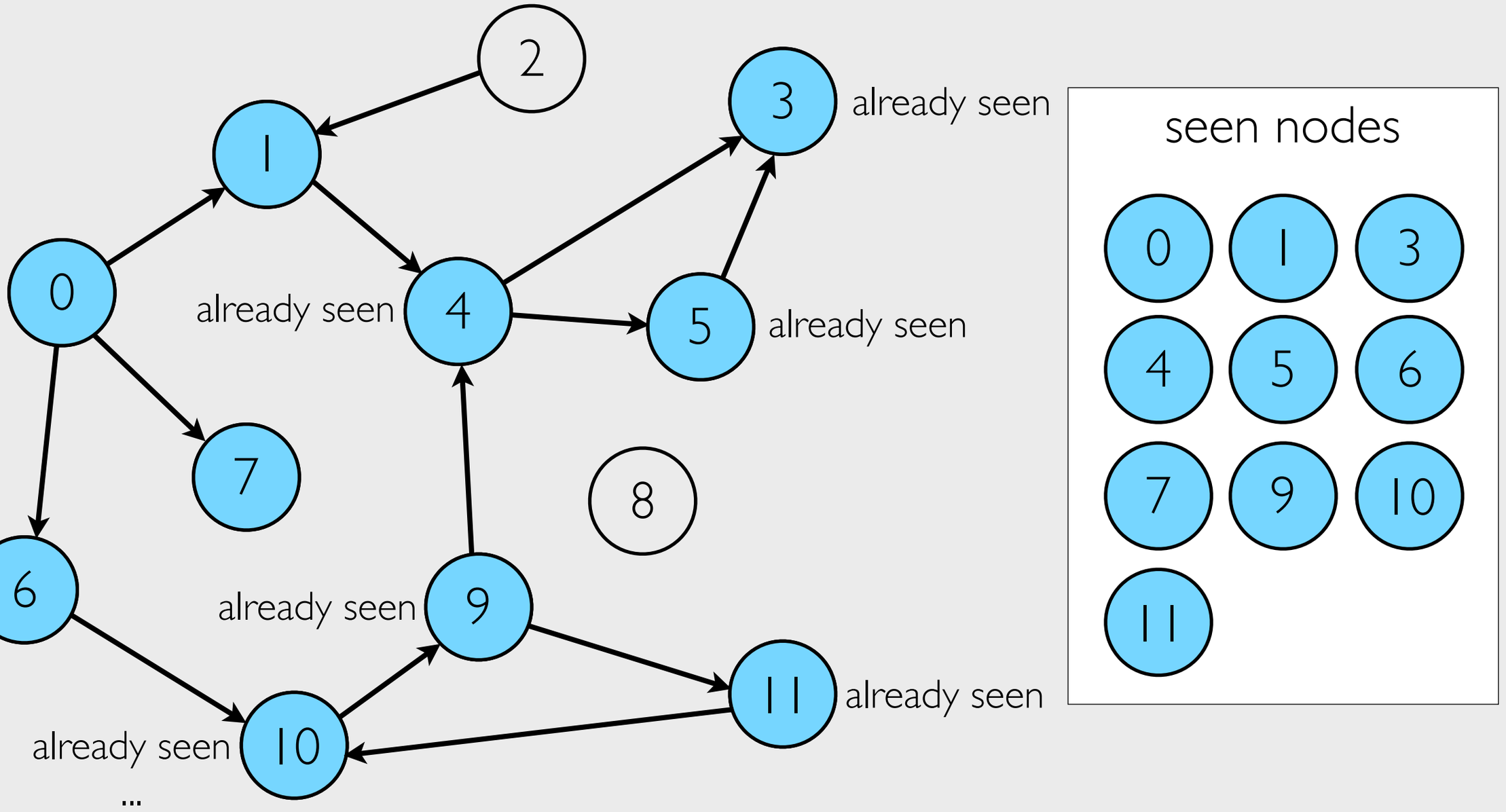


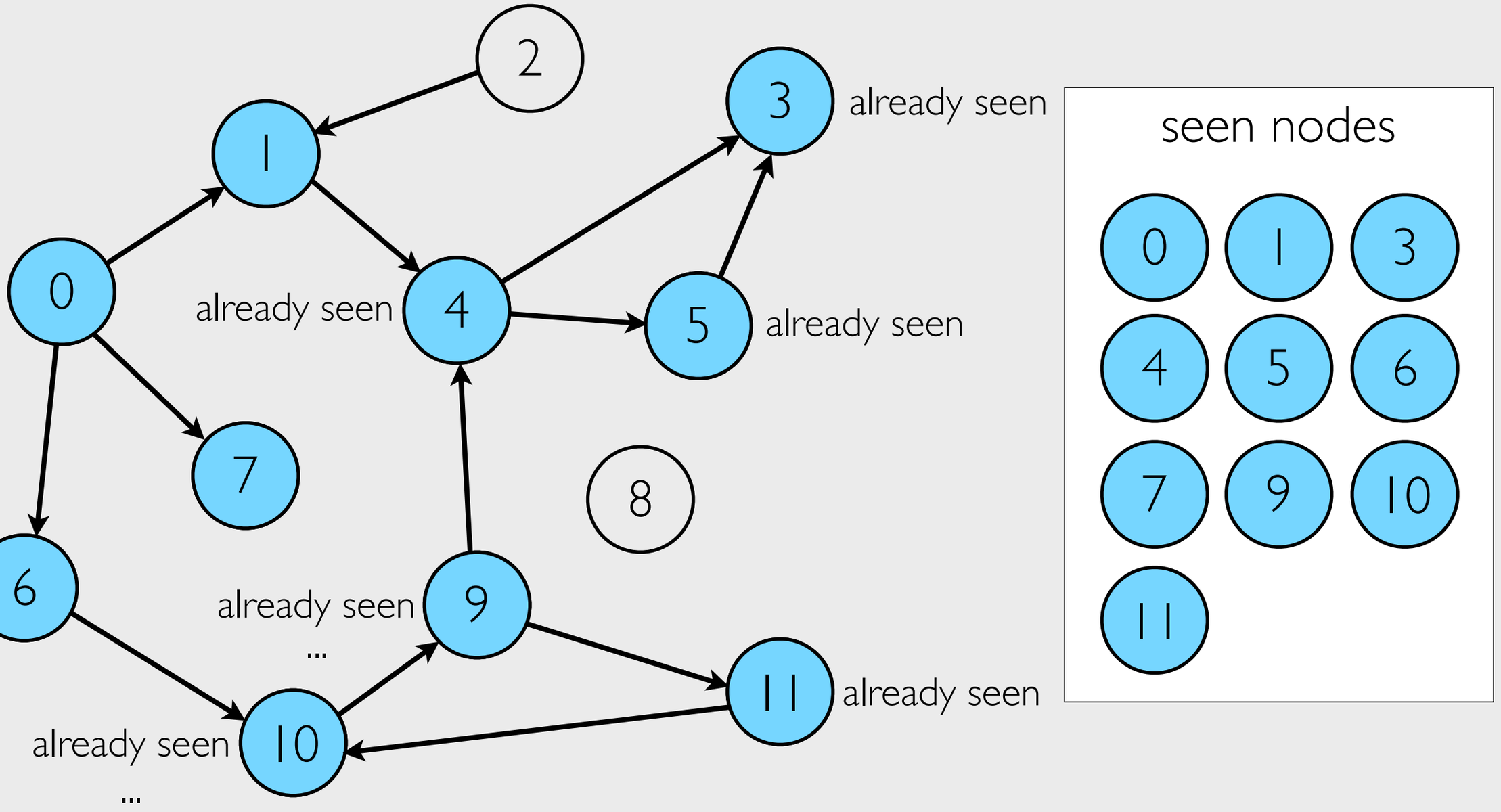
seen nodes

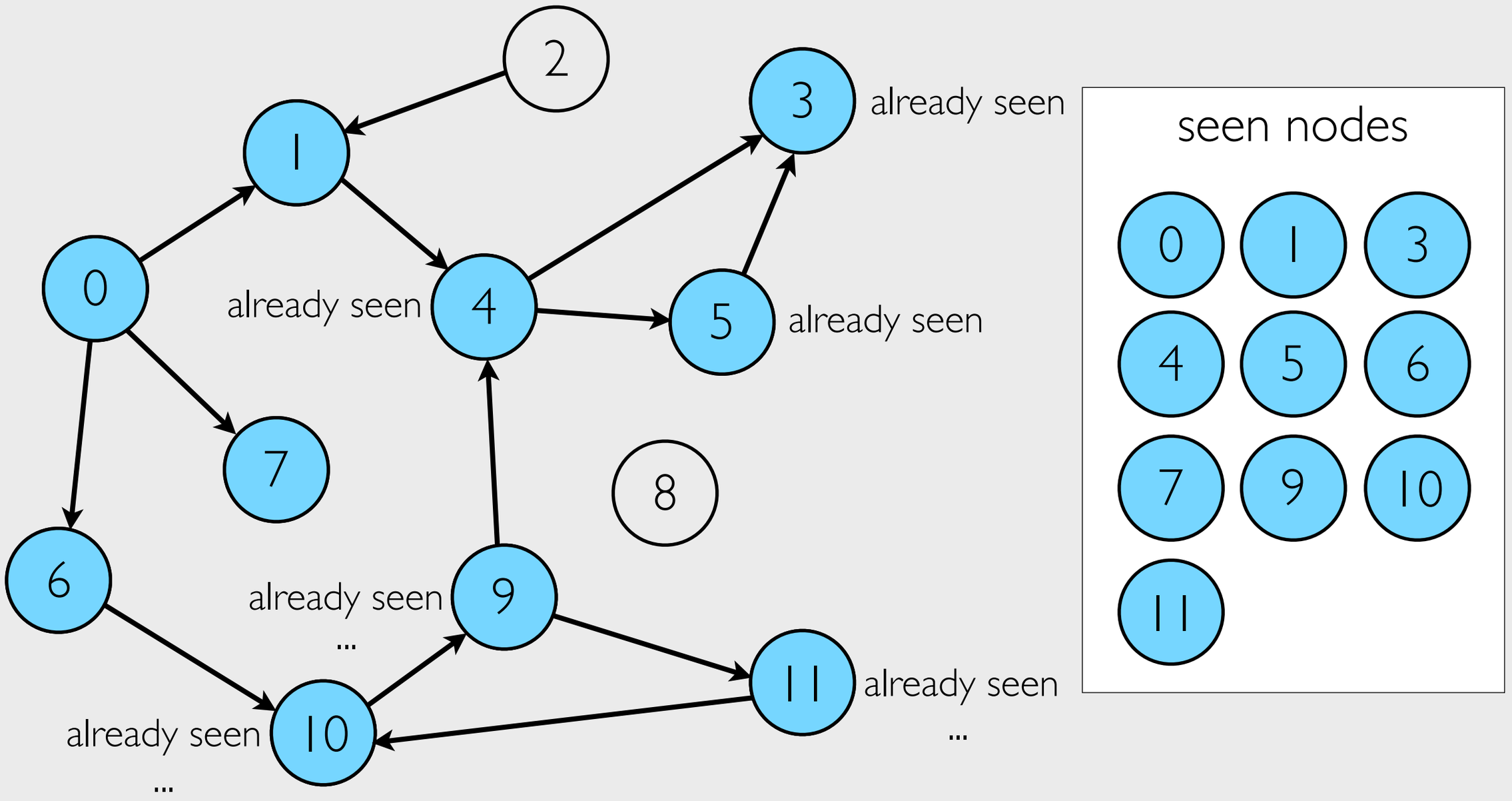



2
1
3
0
already seen
4
5
7
8
6
already seen
9
11
already seen
already seen
10

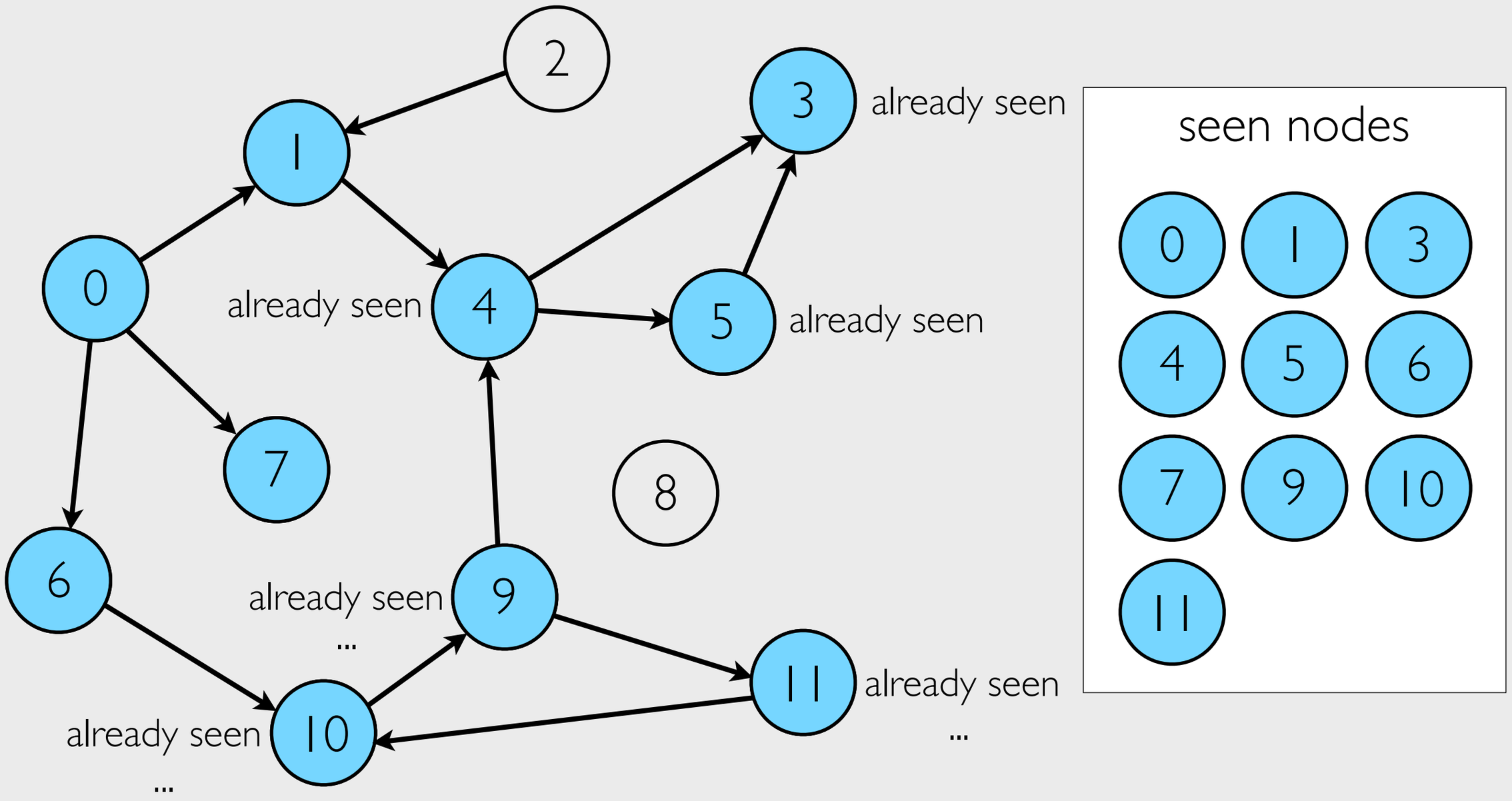

seen nodes
0
1
3
4
5
6
7
9
10
11


2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
9
11
already seen
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11


2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
...
9
11
already seen
already seen
...
10
seen nodes
0
1
3
4
5
6
7
9
10
11

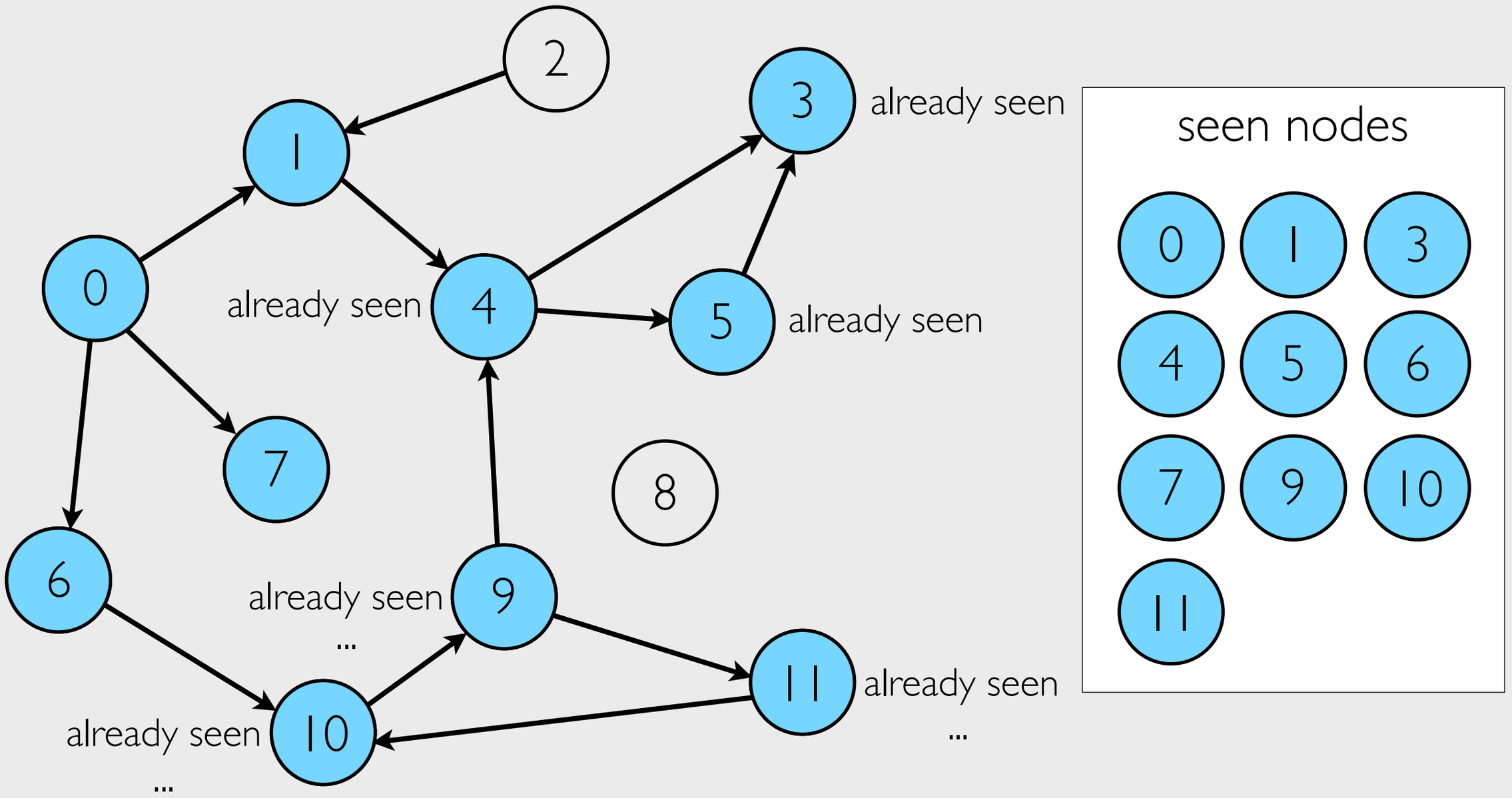

2
1
3
already seen
0
already seen
4
5
already seen
7
8
6
already seen
...
9
11
already seen
...
already seen
10
...
seen nodes
0
1
3
4
5
6
7
9
10
11

Events are updates that change an LVar's state



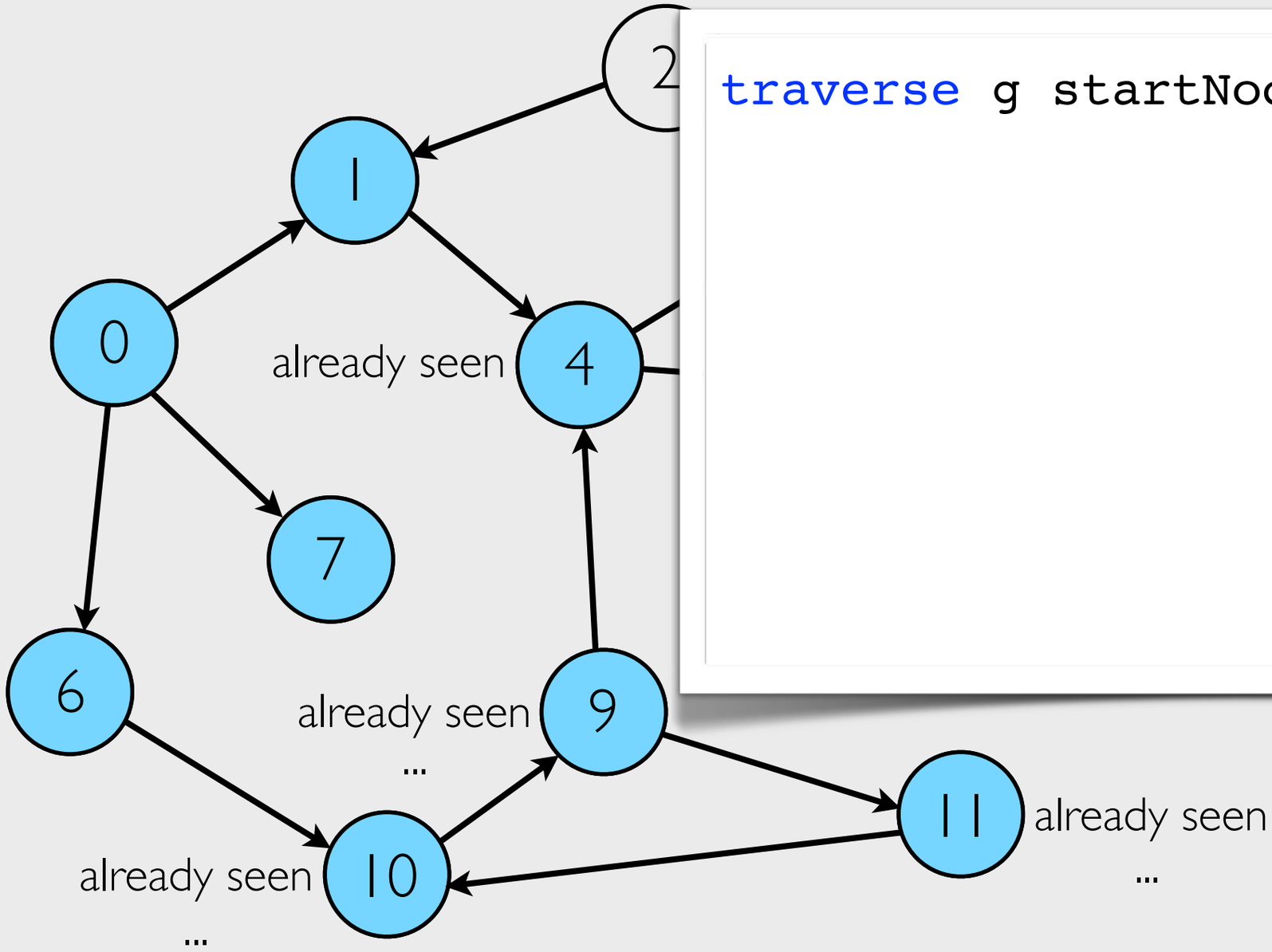
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response



Events are updates that change an LVar's state

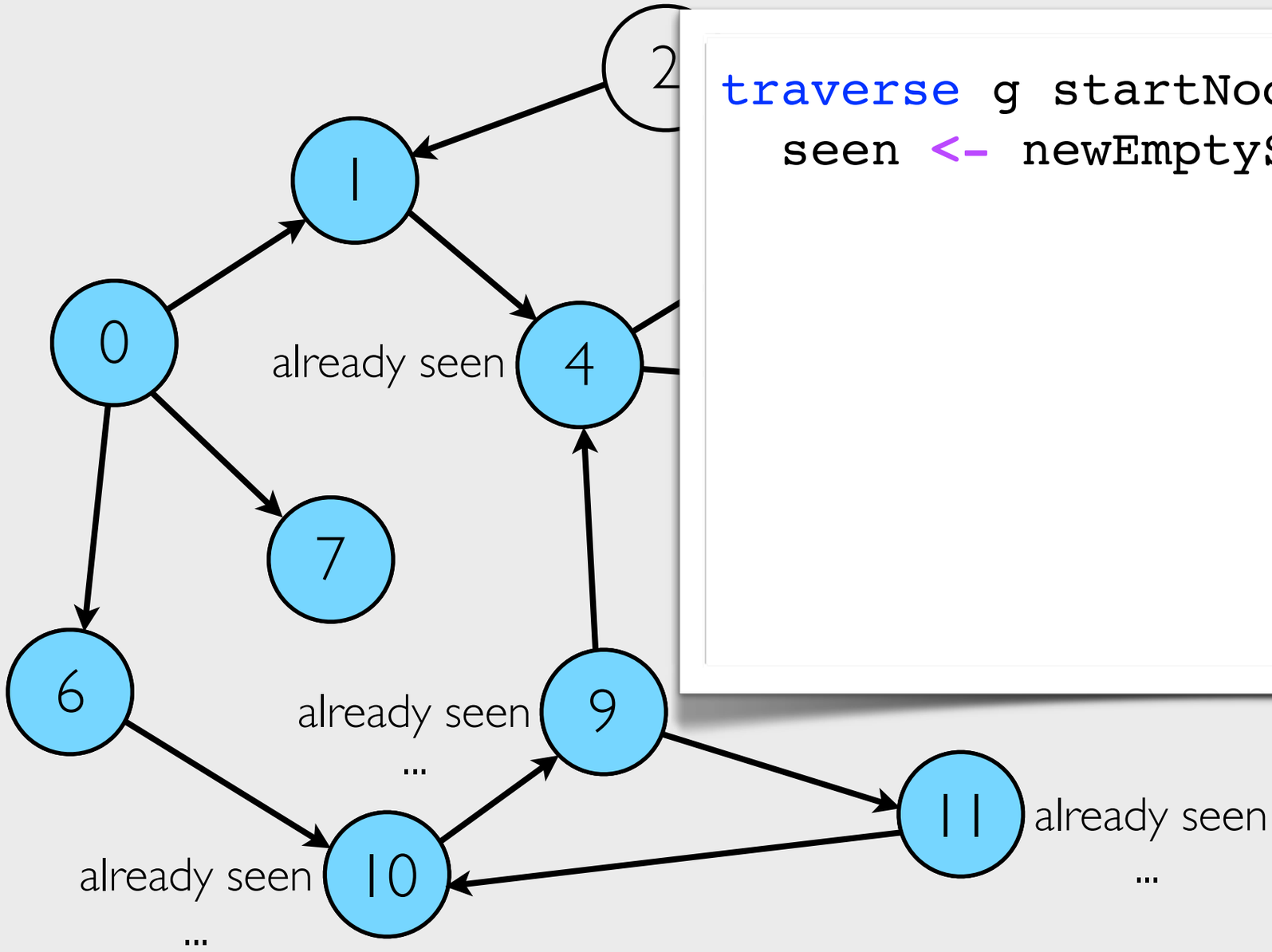
Event handlers listen for events and launch callbacks in response




```
traverse g startNode = do
```

Events are updates that change an LVar's state

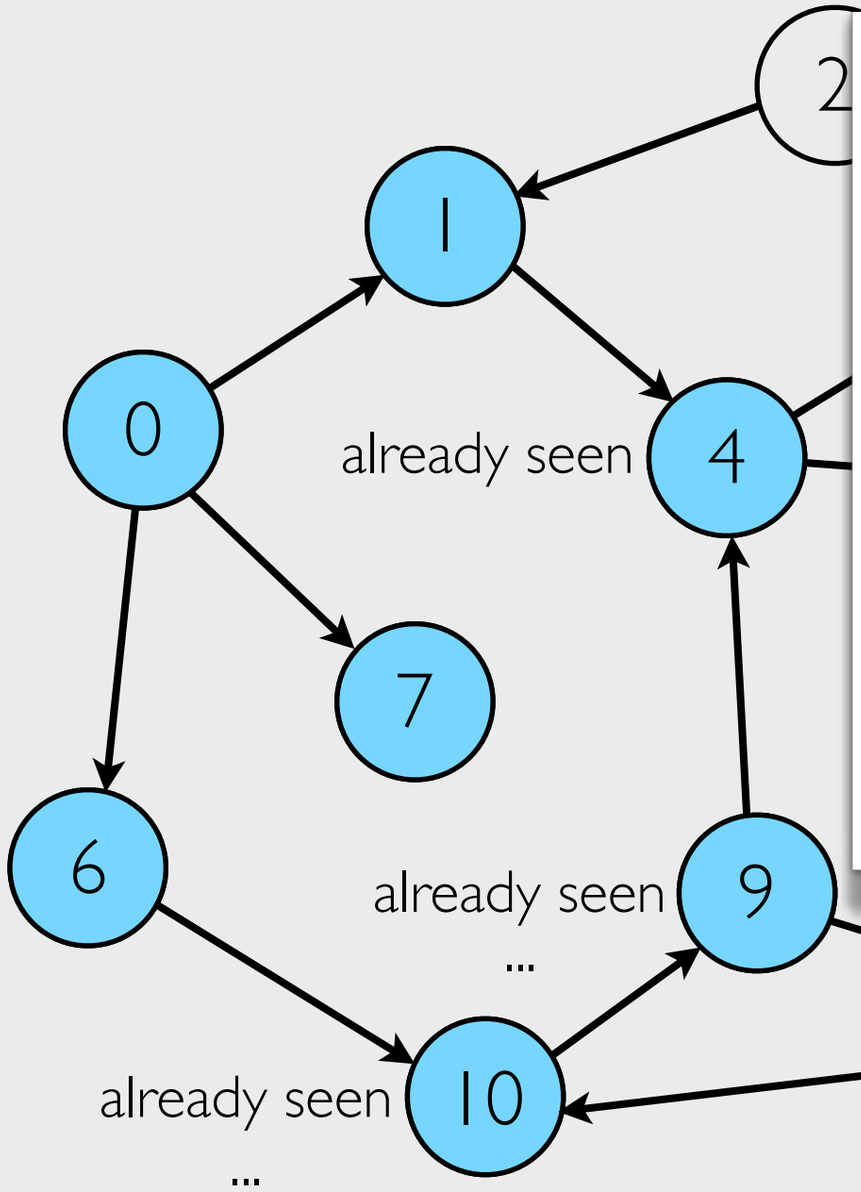
Event handlers listen for events and launch callbacks in response




```
traverse g startNode = do
  seen <- newEmptySet
```

Events are updates that change an LVar's state

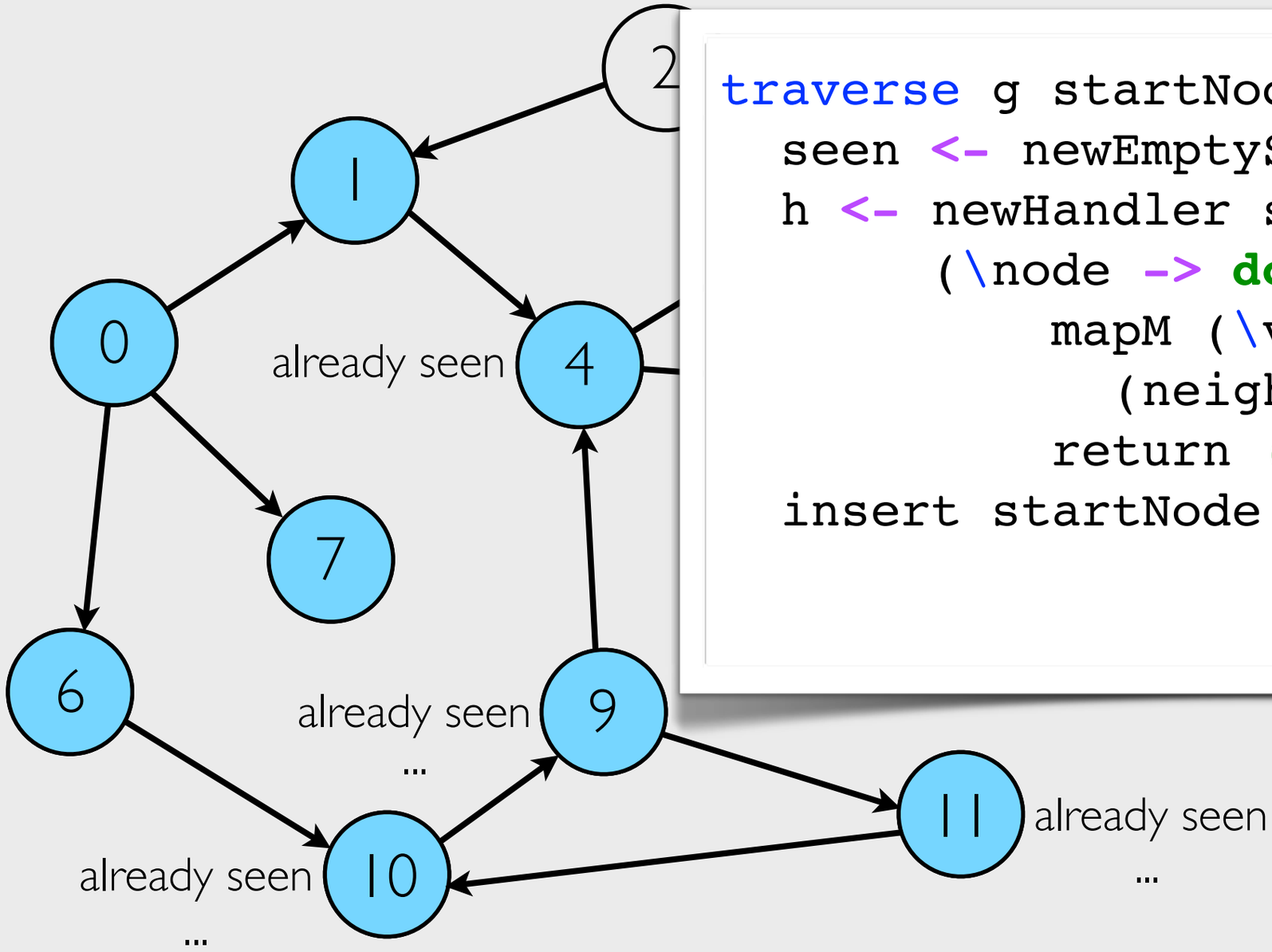
Event handlers listen for events and launch callbacks in response




```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

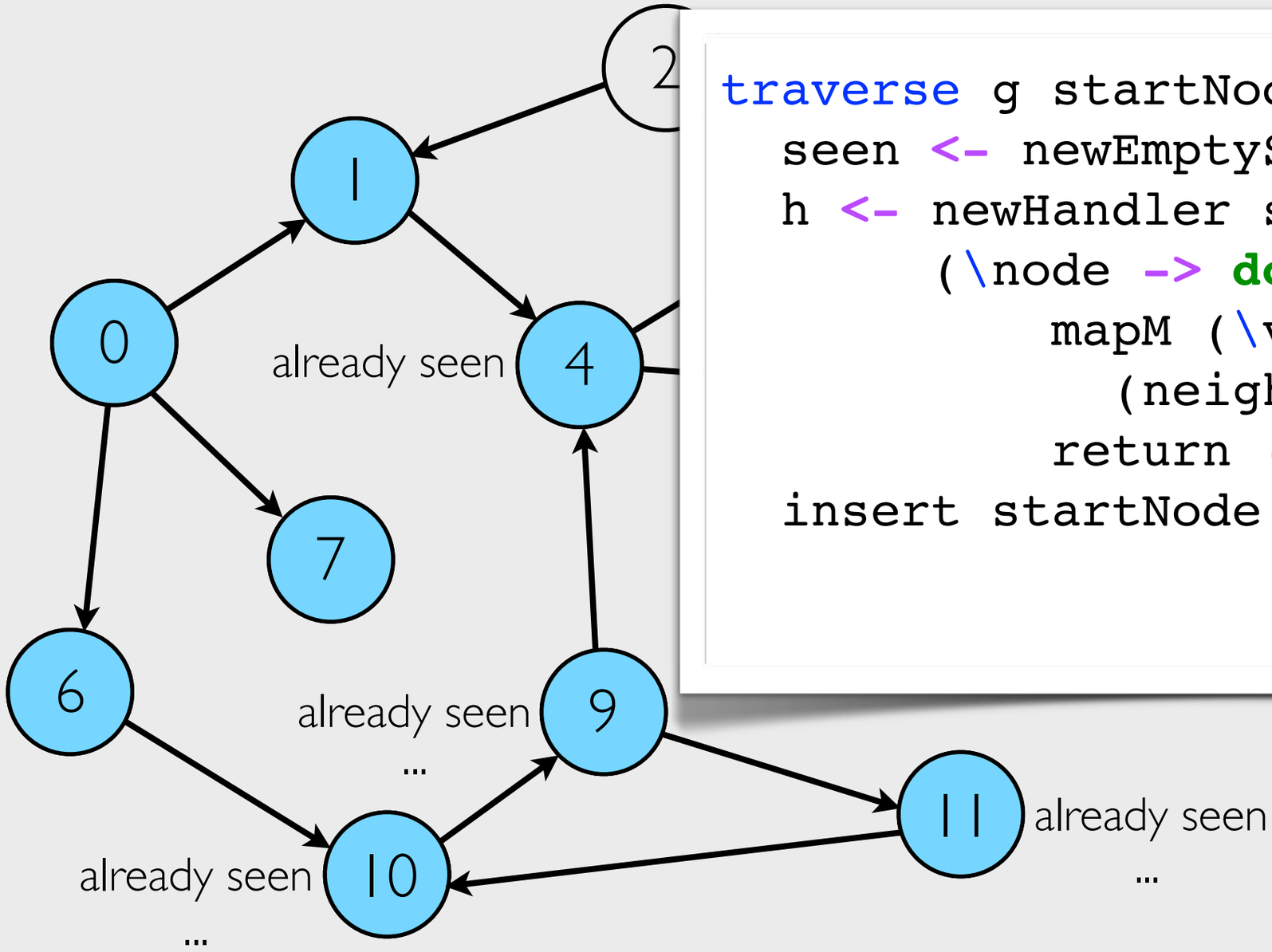



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running

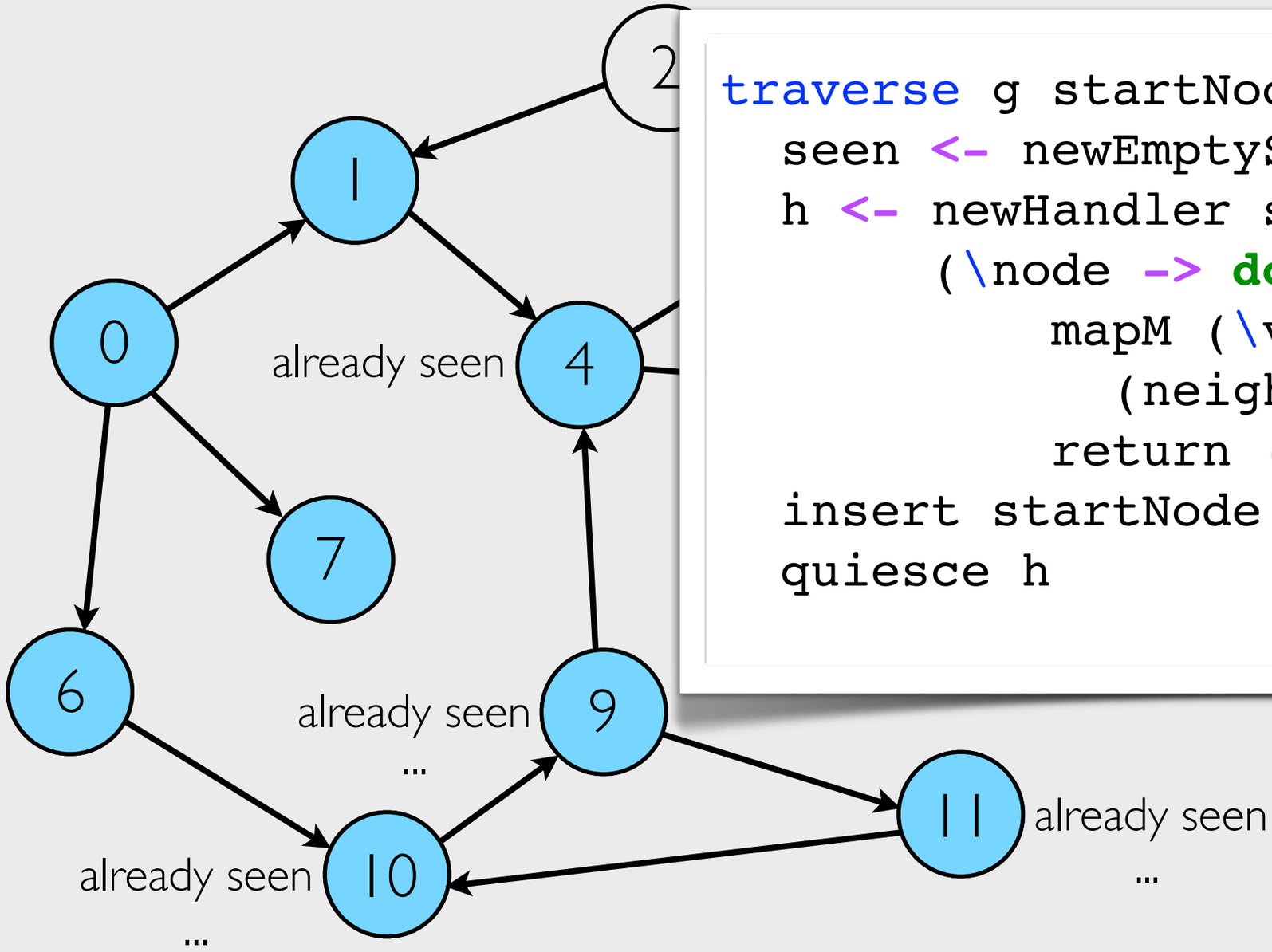



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running

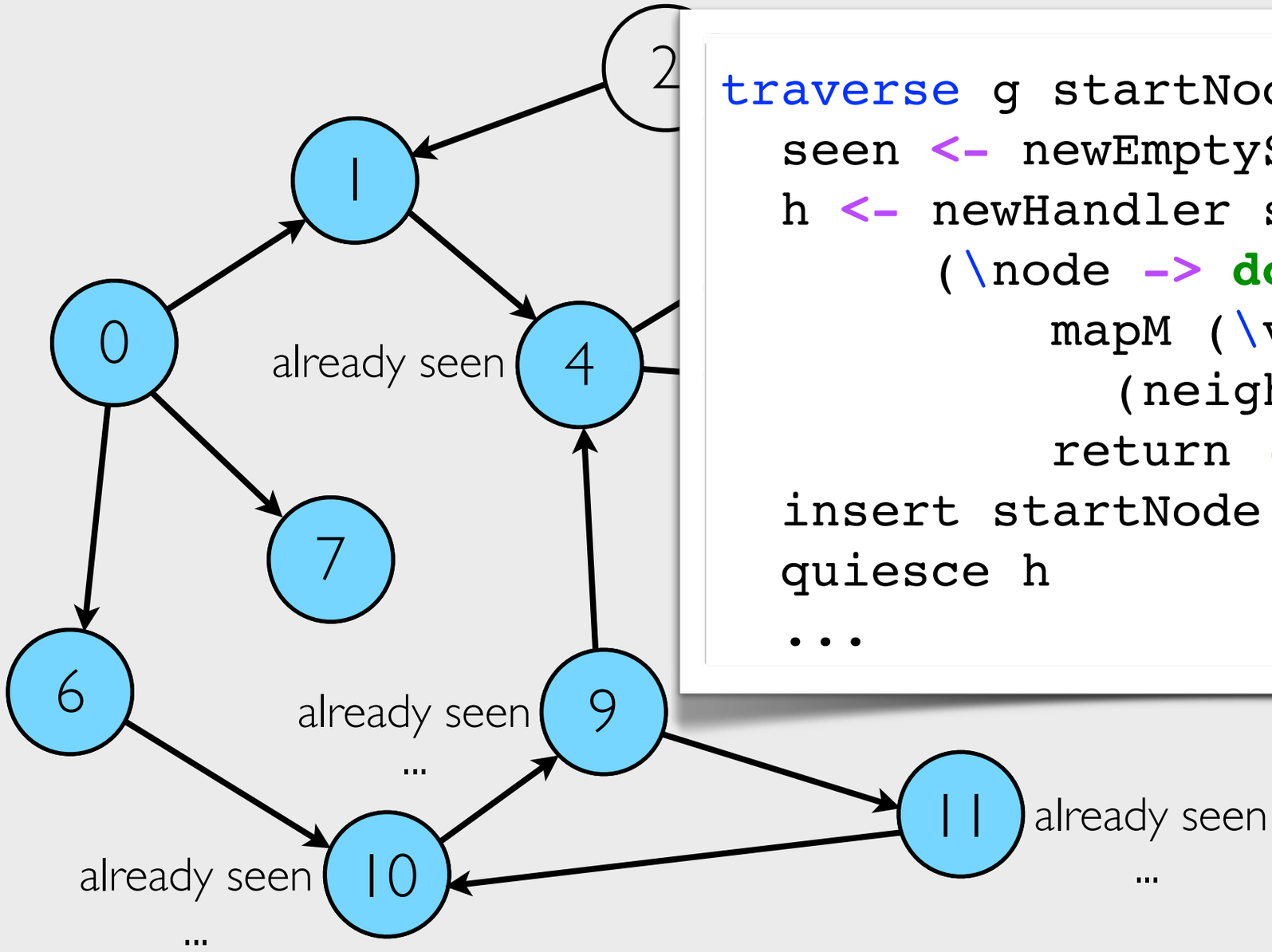



```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
```

Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

`quiesce` blocks until all callbacks launched by a given handler are done running




```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
            mapM (\v -> insert v seen)
              (neighbors g node)
            return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  ...
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
       (\node -> do
           mapM (\v -> insert v seen)
             (neighbors g node)
           return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longmapsto^* \sigma'$ *and* $\sigma \longmapsto^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

```
do
                                    nsert v seen)
                                    g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```

`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

```
do
                                    nsert v seen)
                                    g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```

[(Book,1),(Shoes,1)]

[(Book,1)]

[(Shoes,1)]

`freeze`: exact non-blocking read

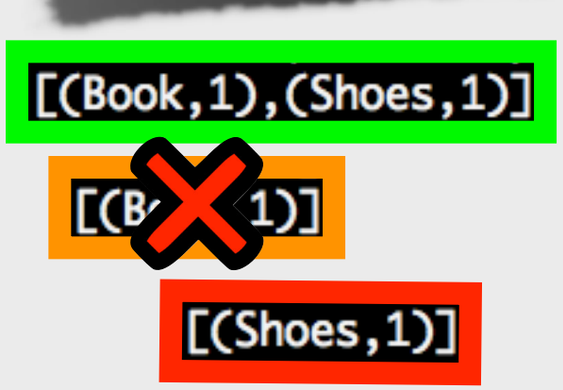
Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

```
do
                                    nsert v seen)
                                    g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```



`freeze`: exact non-blocking read

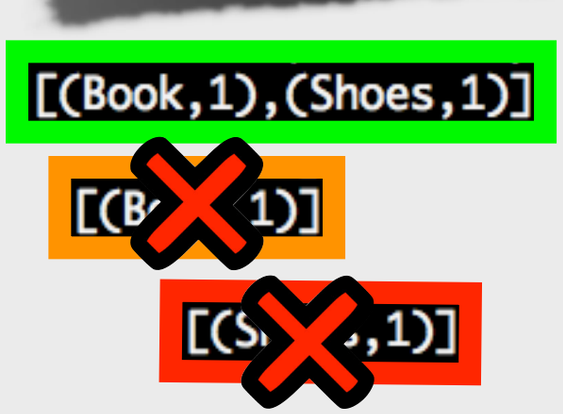
Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.

```
do
                                    nsert v seen)
                                    g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```



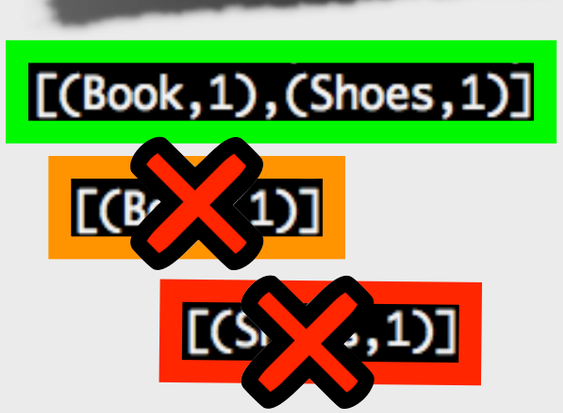
`freeze`: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**`freeze`** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). *If* $\sigma \longhookrightarrow^* \sigma'$ *and* $\sigma \longhookrightarrow^* \sigma''$, *and neither* σ' *nor* σ'' *can take a step, then either:*

1. $\sigma' = \sigma''$ *up to a permutation on locations* π, *or*
2. $\sigma' =$ **error** *or* $\sigma'' =$ **error**.



or error.

```
do
                                  nsert v seen)
                                  g node)
                  return ())
  insert startNode seen
  quiesce h
  freeze seen
```

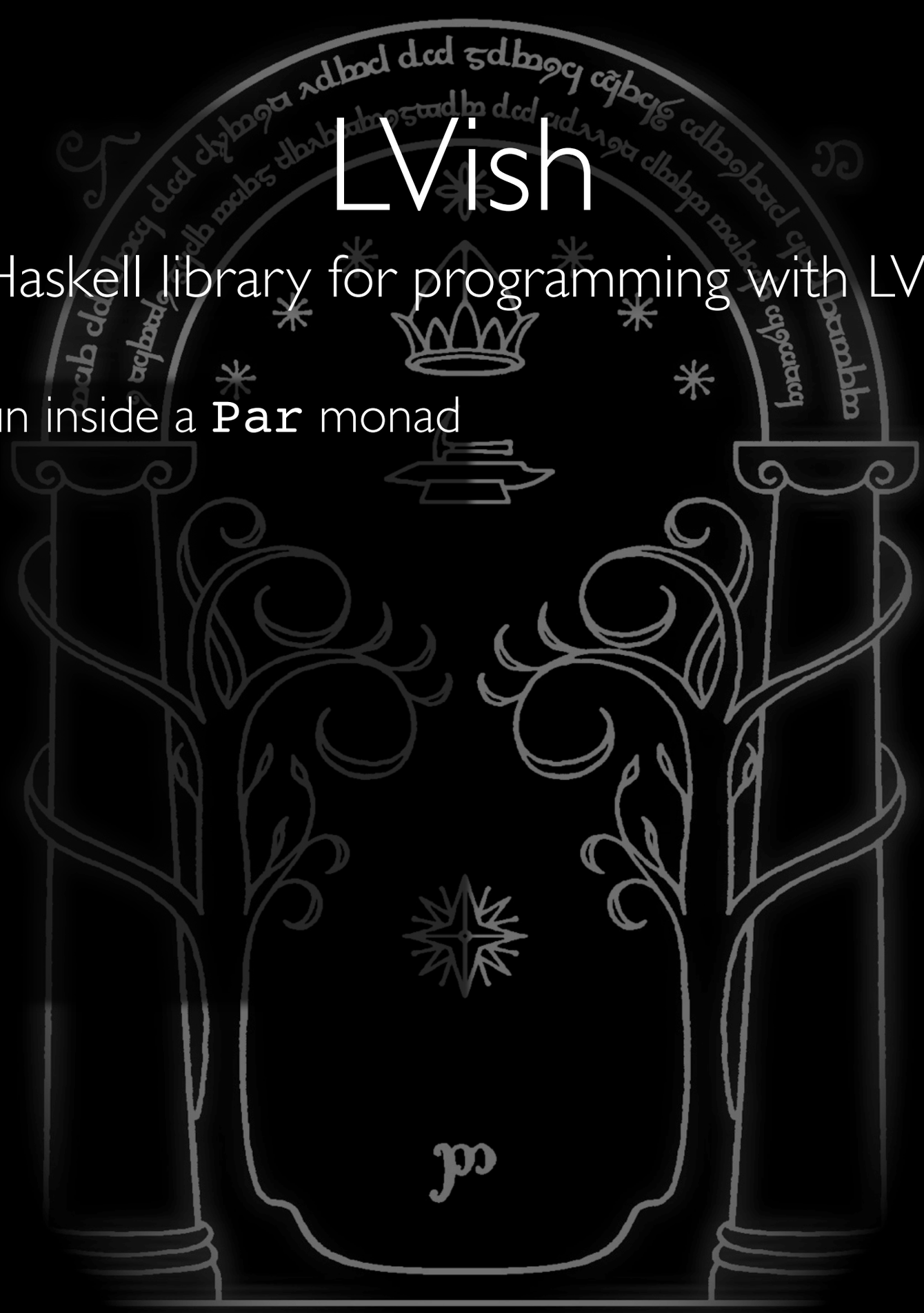
LVish

a Haskell library for programming with LVars

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad



LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

`Par` computations indexed by *effect level*

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart
```

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart
```

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart

main = do
  putStr $ show $ toList $
    fromIMap $ runParThenFreeze p
```

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

LVar data structures: sets, maps, *etc.*

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart

main = do
  putStr $ show $ toList $
    fromIMap $ runParThenFreeze p
```

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

LVar data structures: sets, maps, *etc.*

Implement your own LVars, too

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart

main = do
  putStr $ show $ toList $
    fromIMap $ runParThenFreeze p
```

LVish

a Haskell library for programming with LVars

LVar operations run inside a **`Par`** monad

Library-level threads

`Par` computations indexed by *effect level*

`runParThenFreeze` captures the freeze-after-writing idiom

LVar data structures: sets, maps, *etc.*

Implement your own LVars, too

`cabal install lvish` today!

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart

main = do
  putStr $ show $ toList $
    fromIMap $ runParThenFreeze p
```

More in the paper and TR

More in the paper and TR

The LVish calculus and its quasi-determinism proof

More in the paper and TR

The LVish calculus and its quasi-determinism proof

Gory details of the LVish scheduler implementation

More in the paper and TR

The LVish calculus and its quasi-determinism proof

Gory details of the LVish scheduler implementation

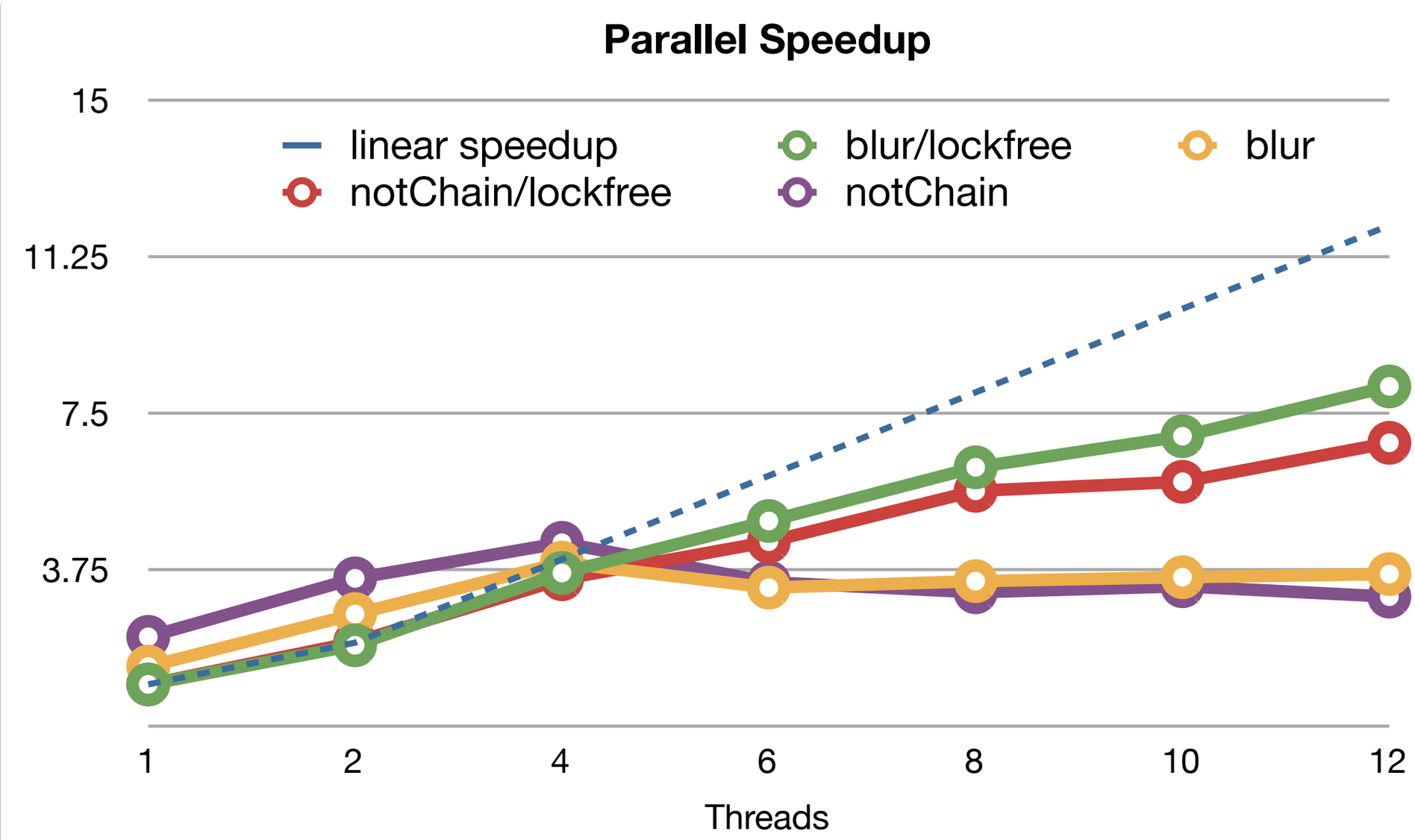
Case study: parallel *k*-CFA written with the LVish library

More in the paper and TR

The LVish calculus and its quasi-determinism proof

Gory details of the LVish scheduler implementation

Case study: parallel *k*-CFA written with the LVish library



Thank you!

Email: lkuper@cs.indiana.edu
Project repo: github.com/iu-parfunc/lvars
Code from this talk: github.com/lkuper/lvar-examples
More LVars papers: cs.indiana.edu/~lkuper
Research blog: composition.al