Dendrology

Brandon Bloom

Dendrology

Brandon Bloom

once upon a time in #clojure…

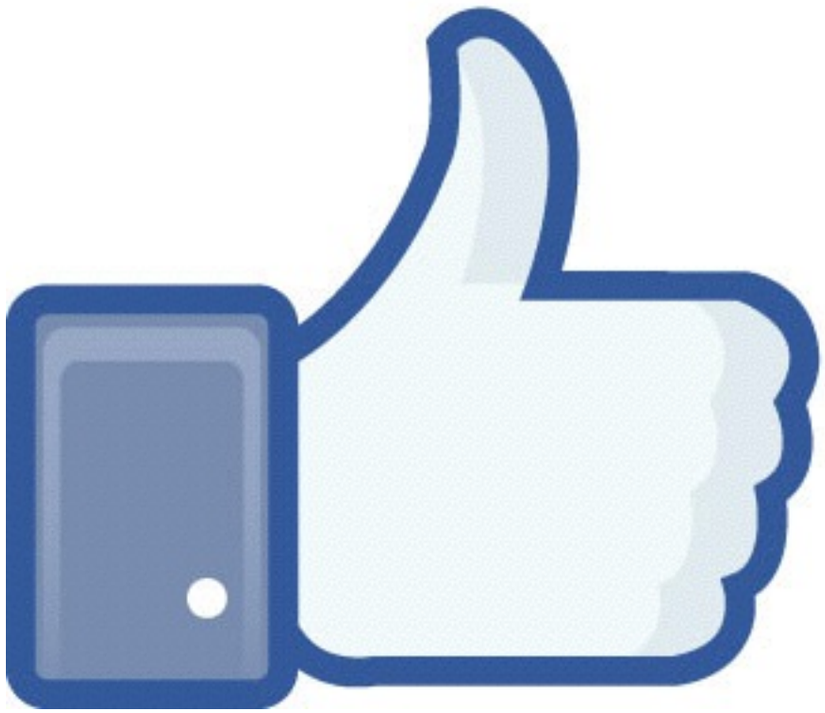
bbloom: better ways to analyze and process
trees is a bit of a new obsession of mine :-)
gfredericks: clojurebot: bbloom is a dendrologist
clojurebot: Alles klar

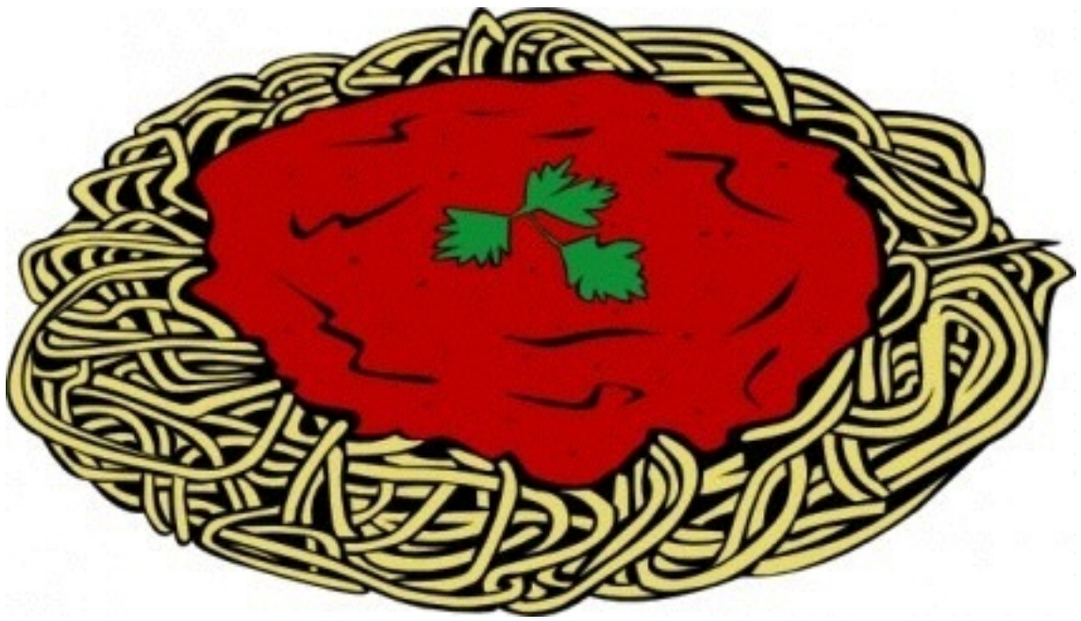
Assumptions

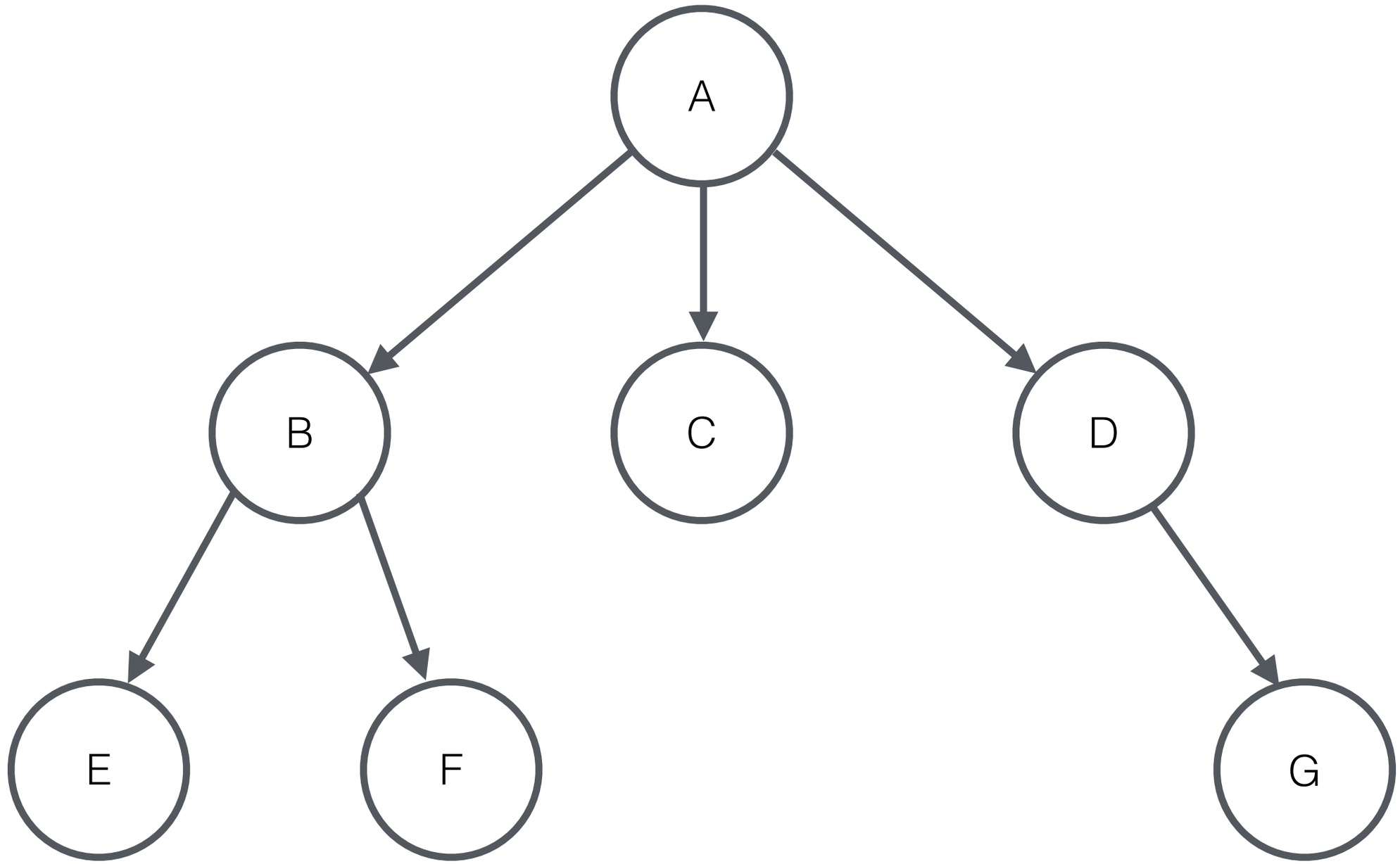
Data

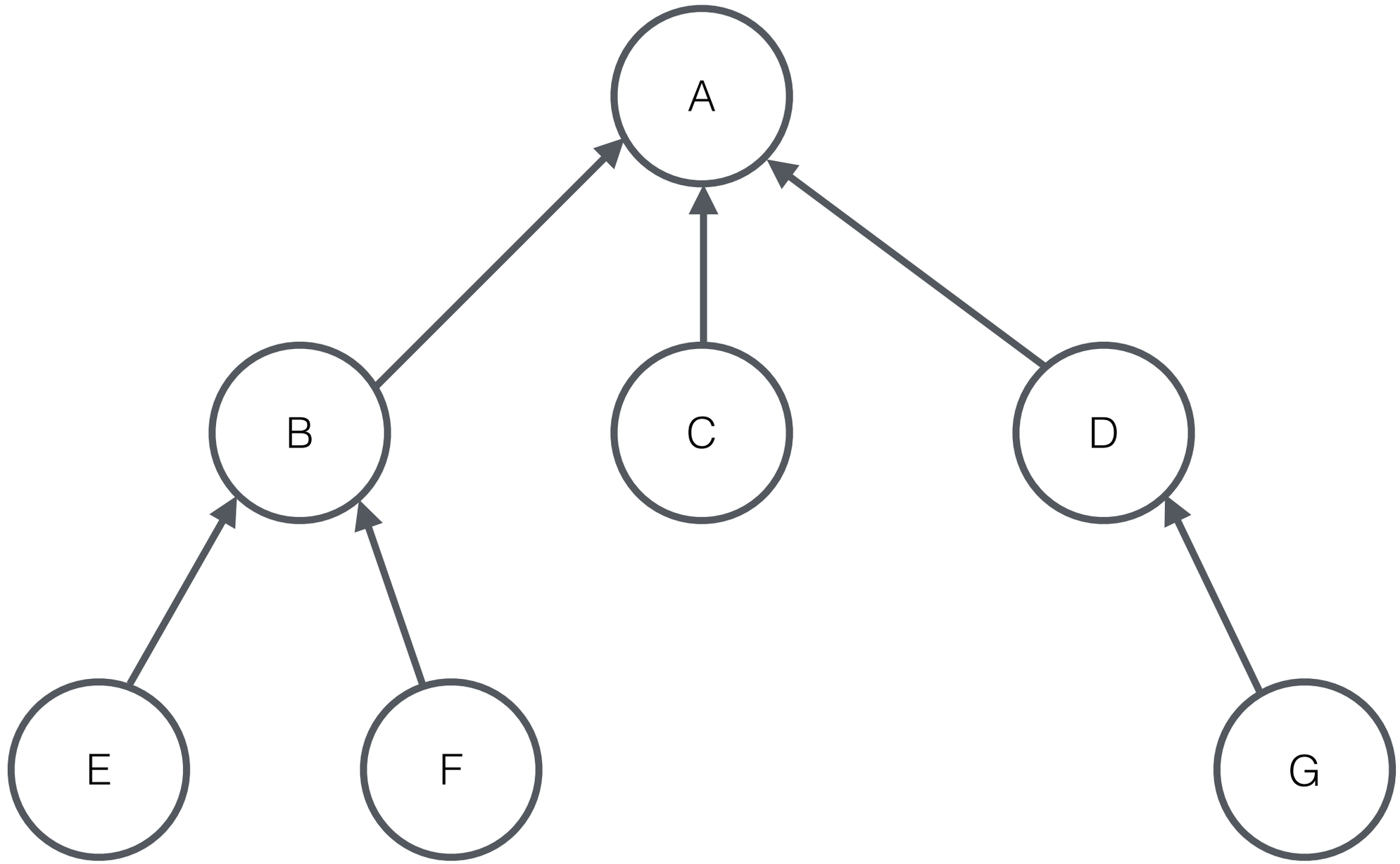
Immutability

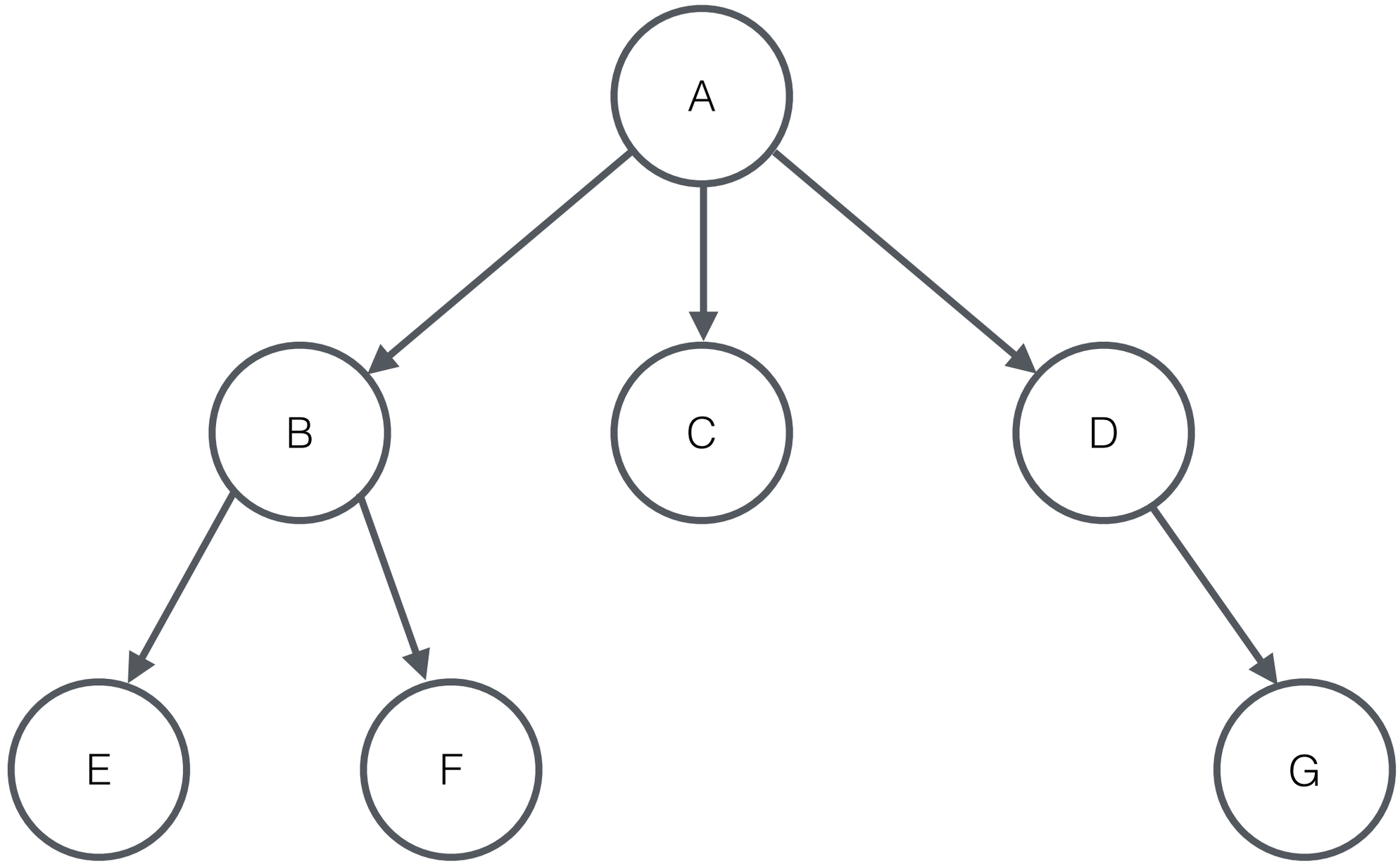
Values

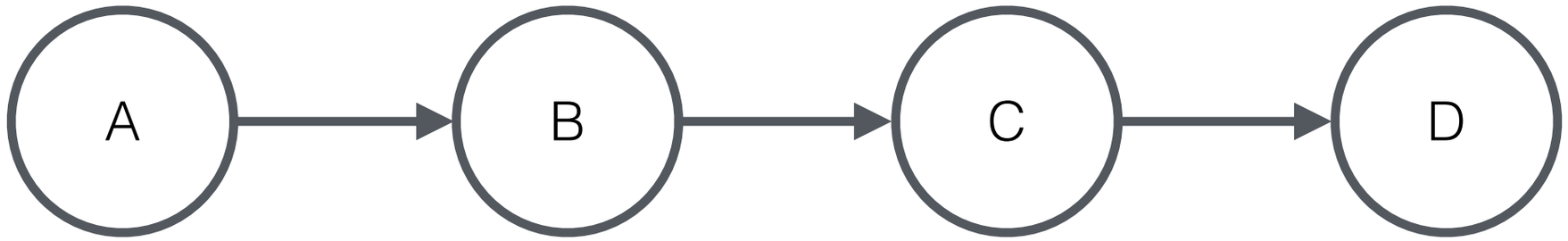


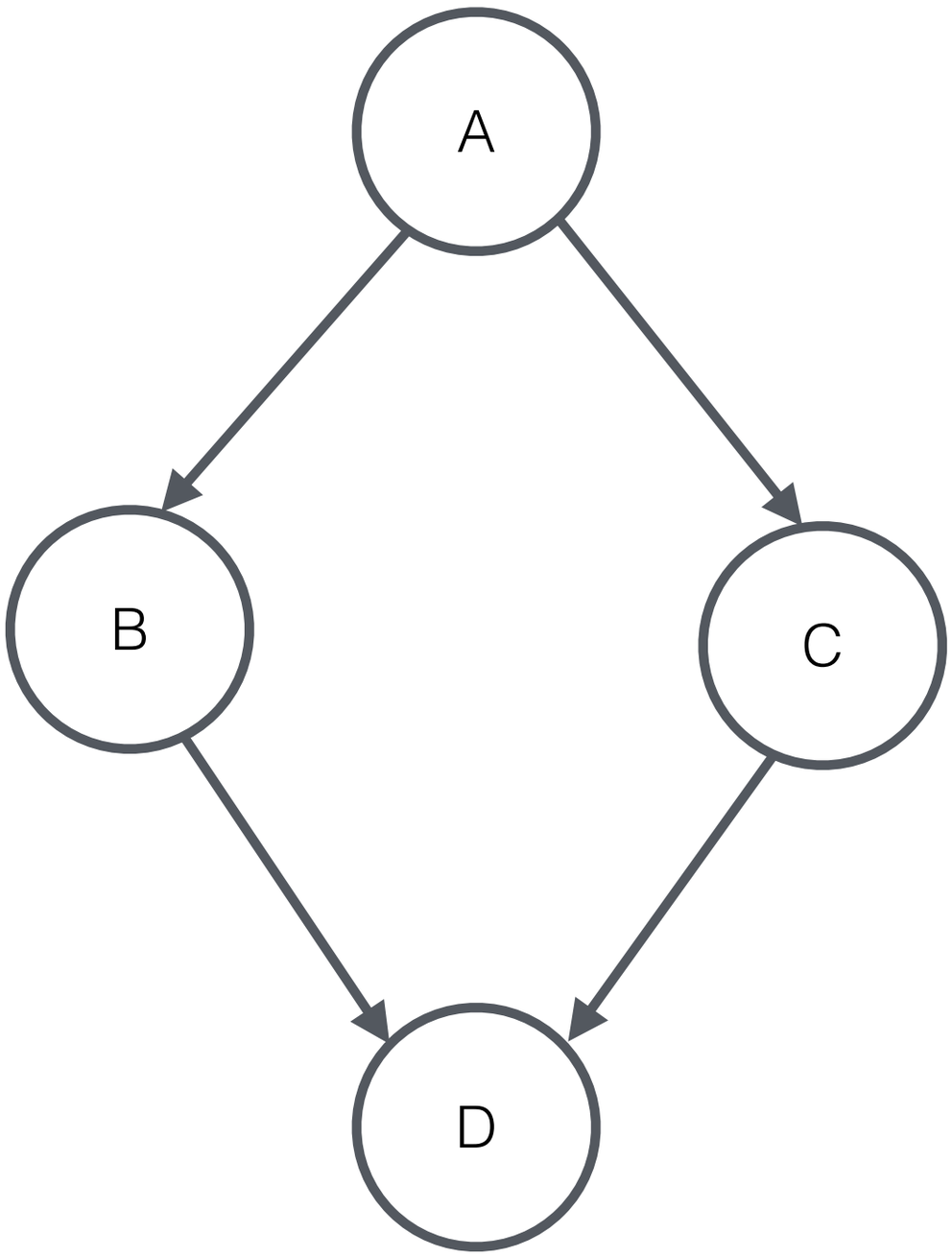


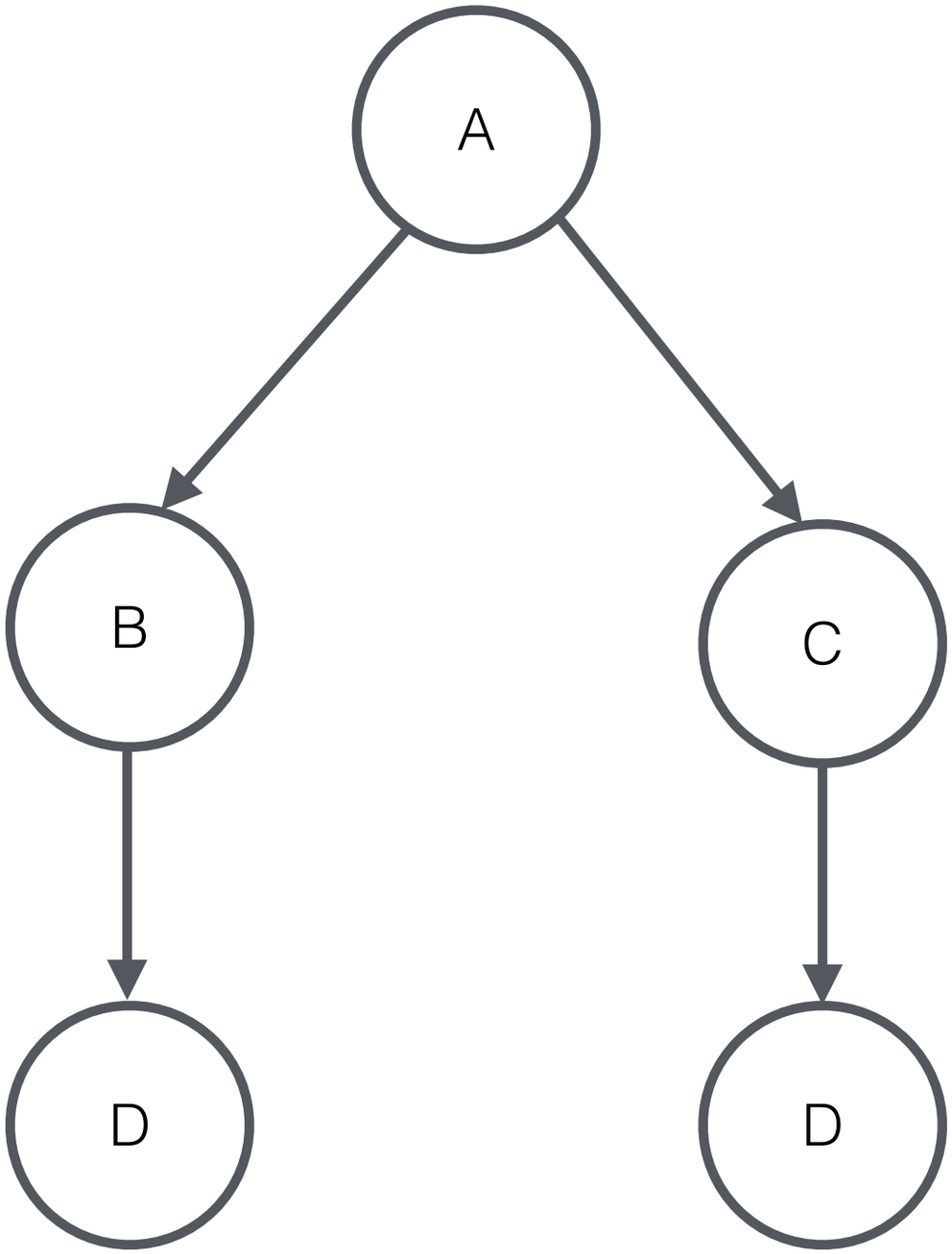

A
B
C
D
E
F
G

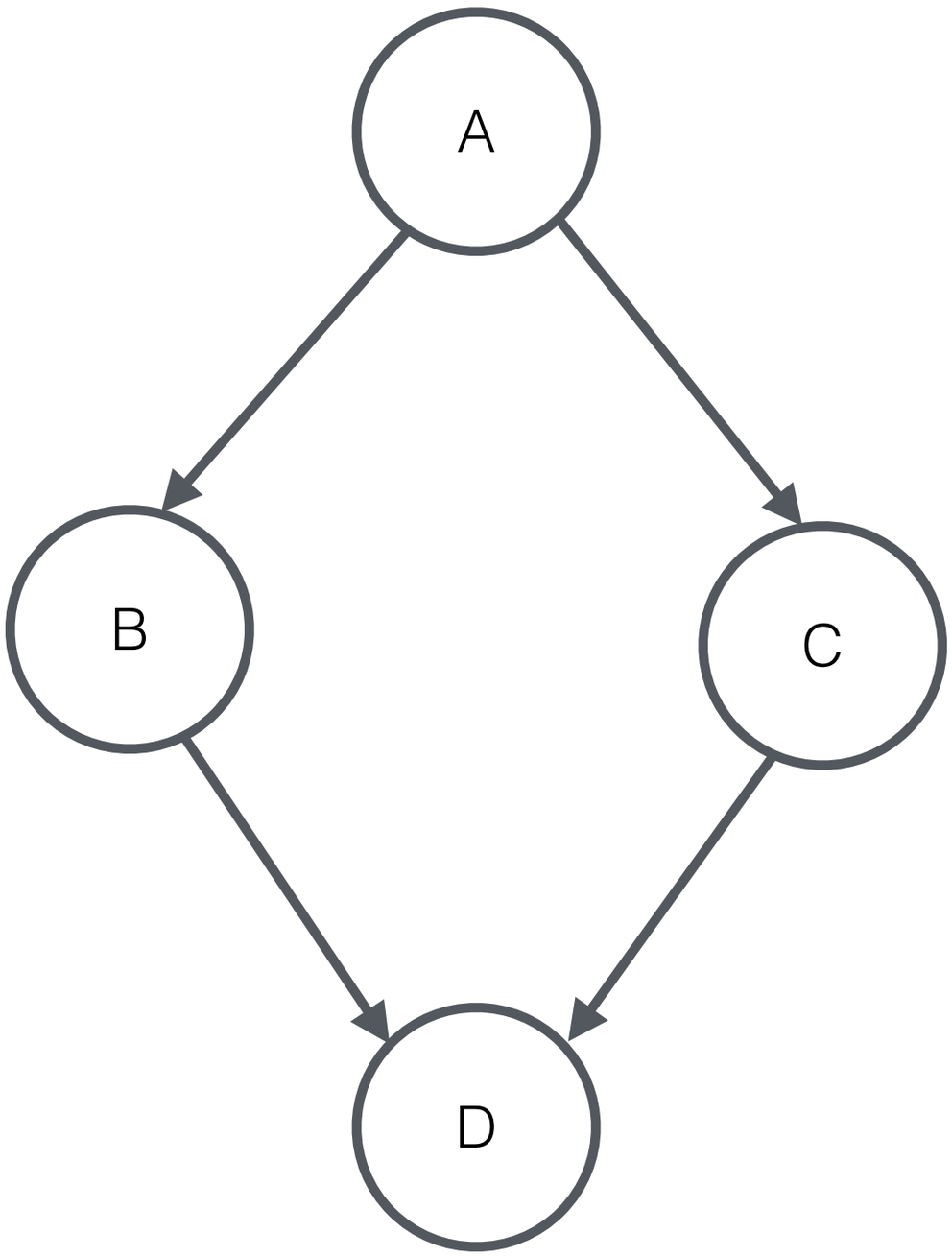

A
B
C
D
E
F
G

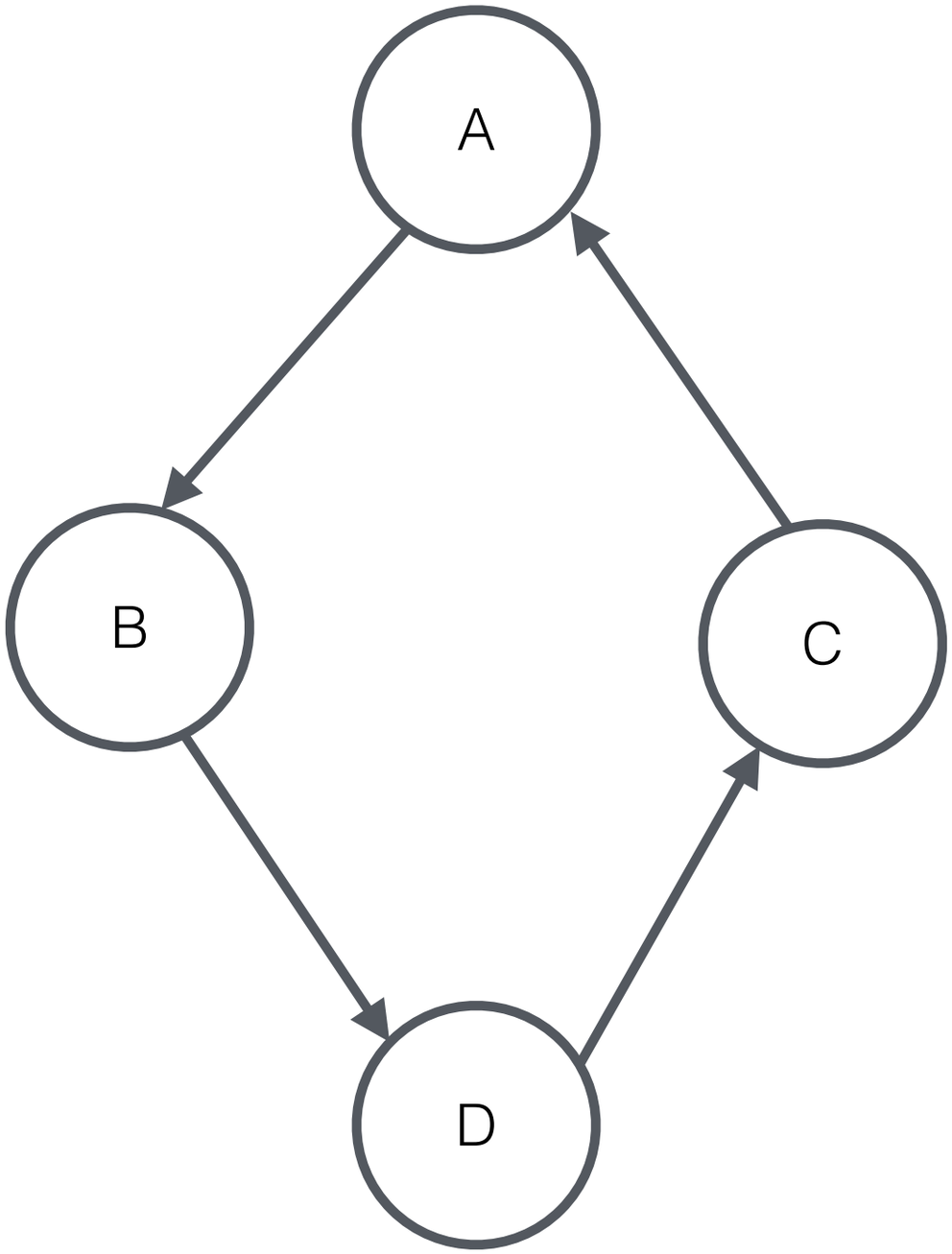

A
B
C
D
E
F
G


A
B
C
D


A
B
C
D


A
B
C
D
D

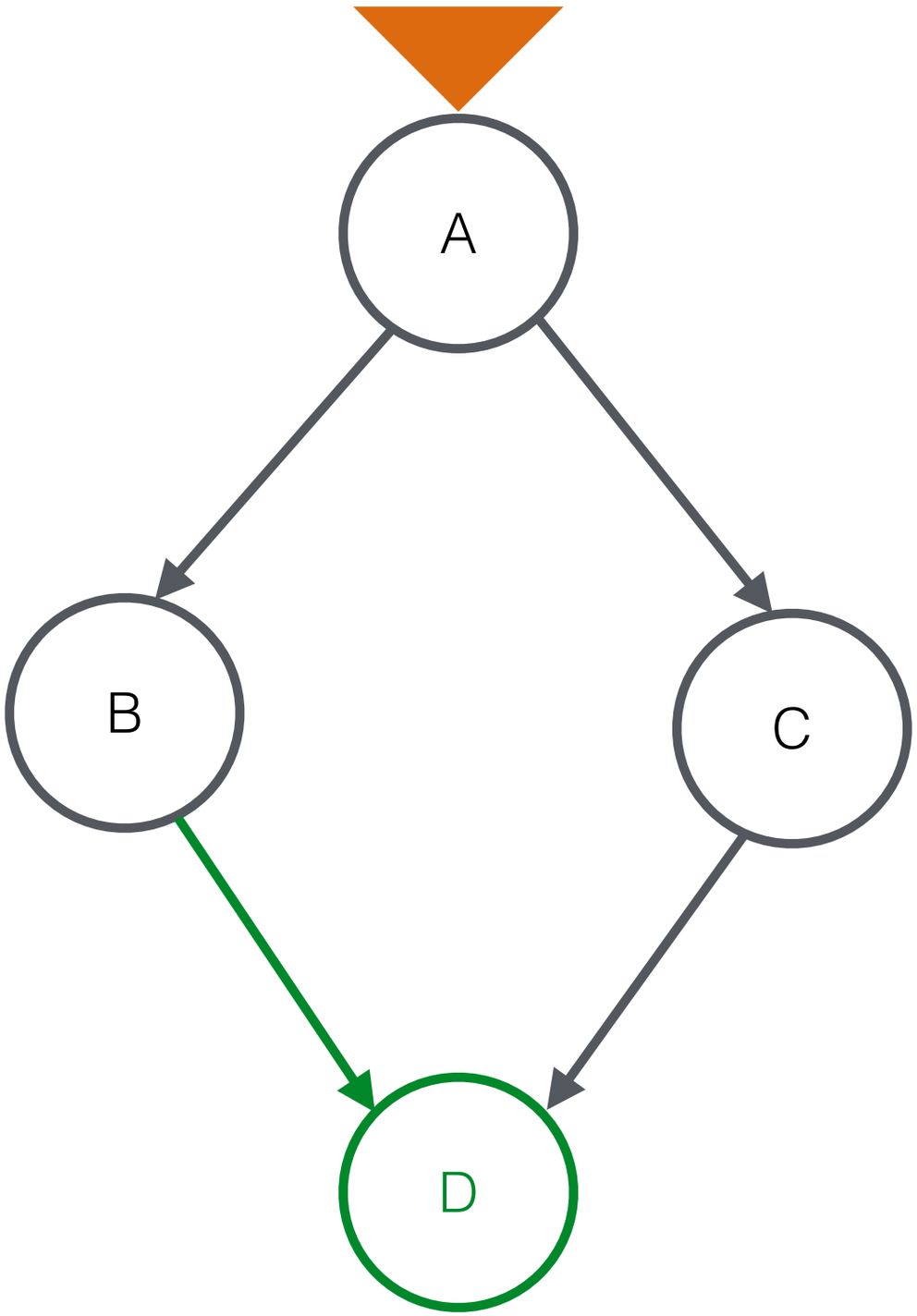

A
B
C
D


A
B
C
D

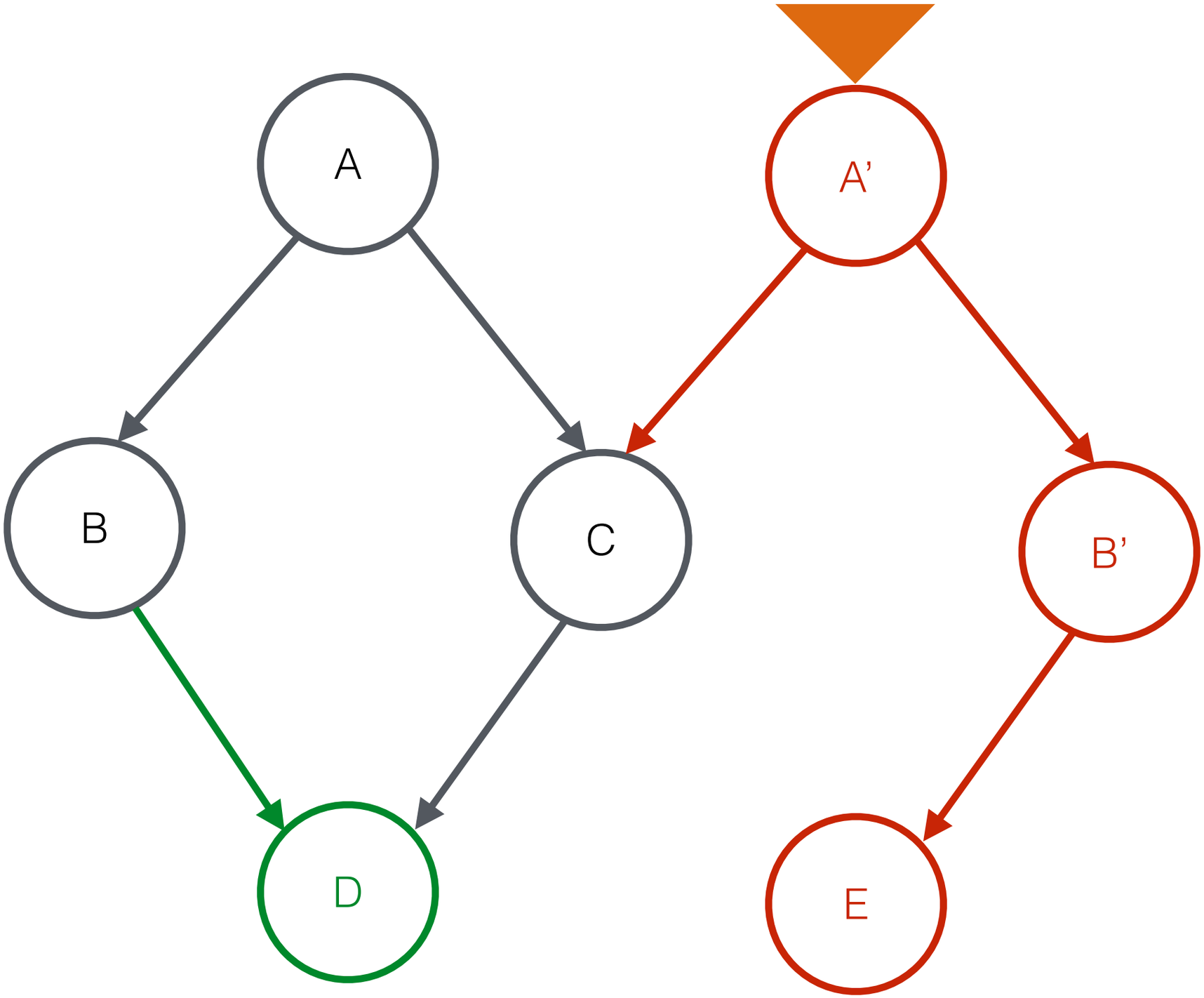
Pointer cycles are the root of all evil.

Pointer cycles are evil.

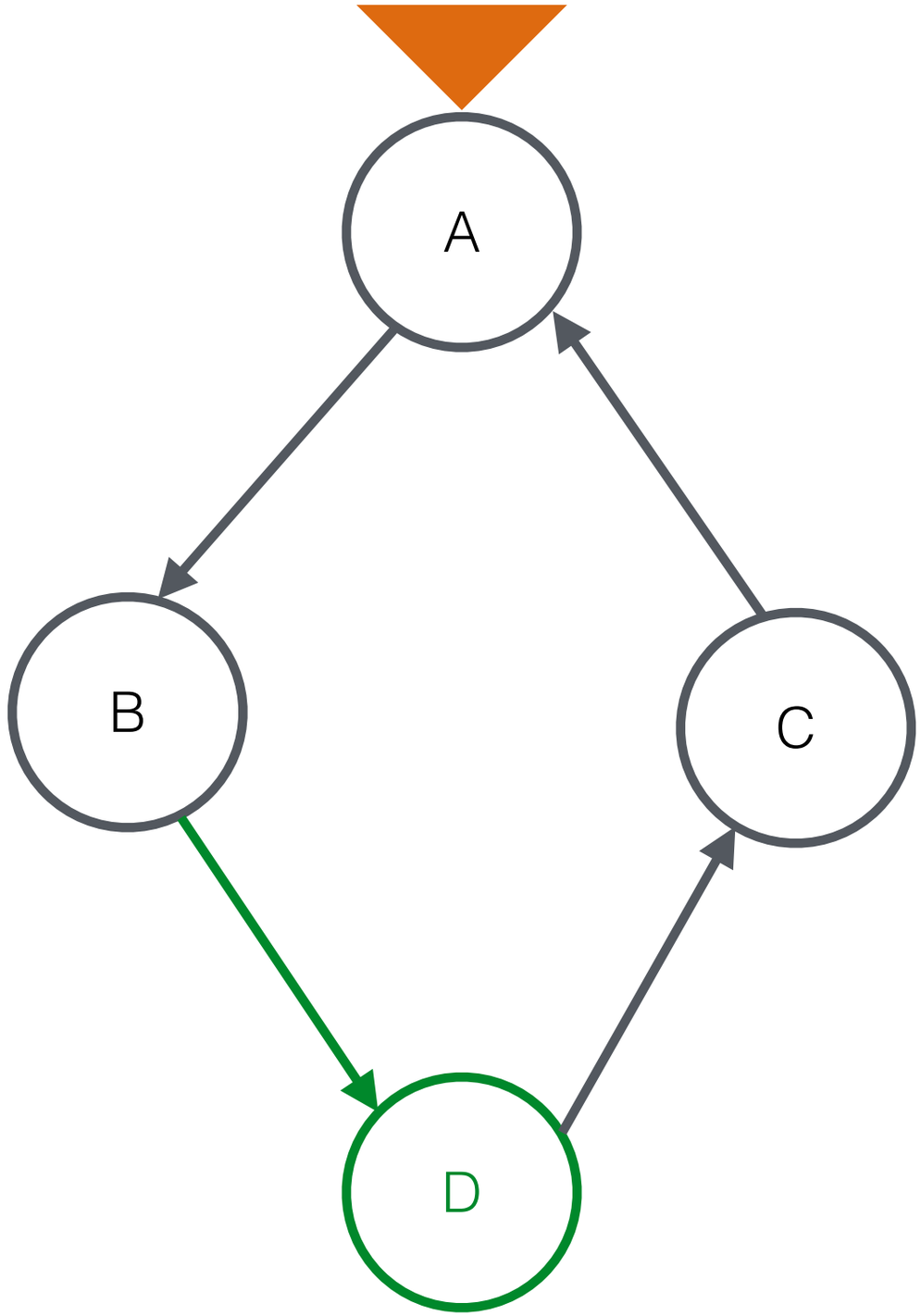
B[D]⇒B[E]



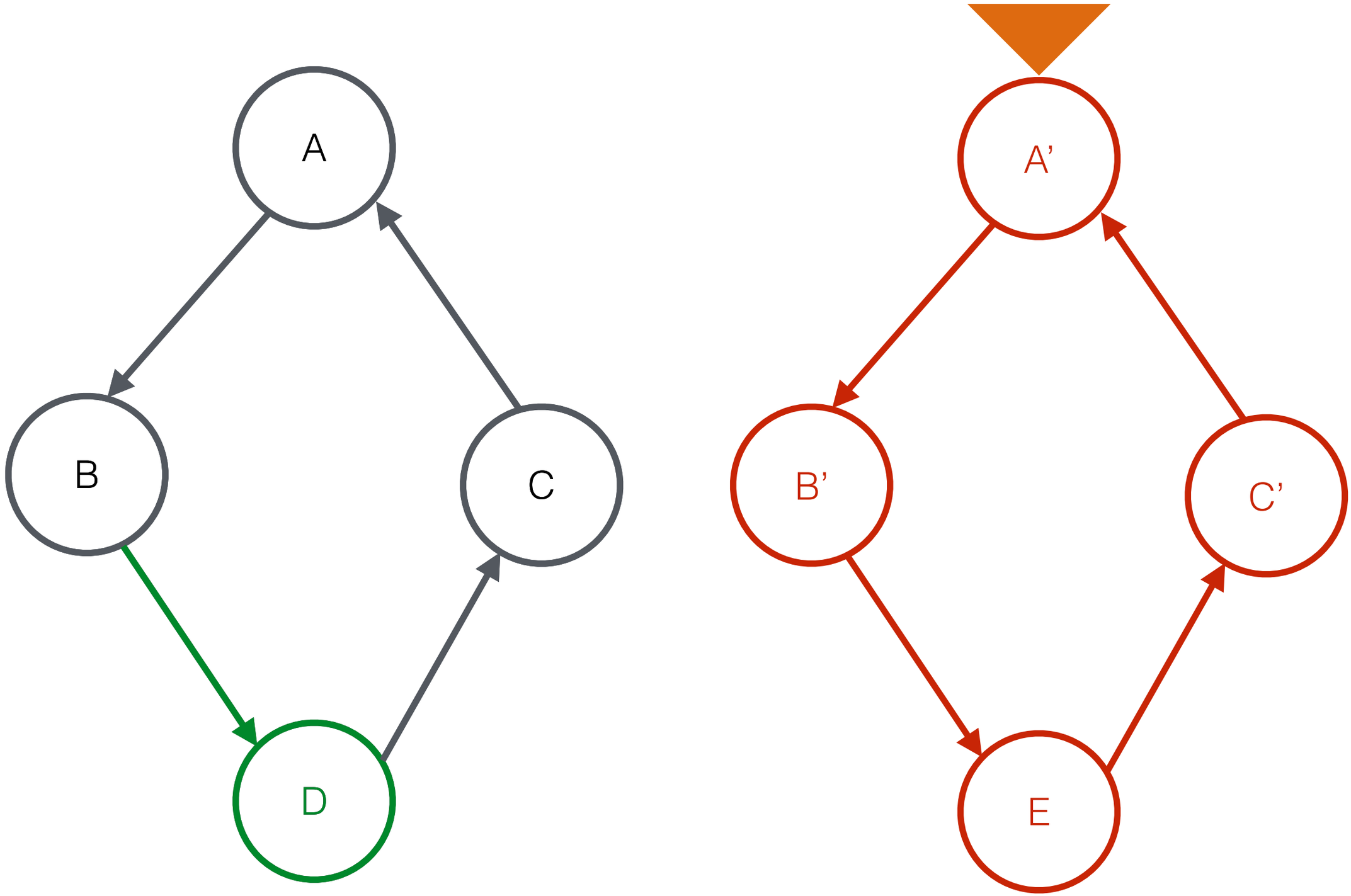
B[D]⇒B[E]

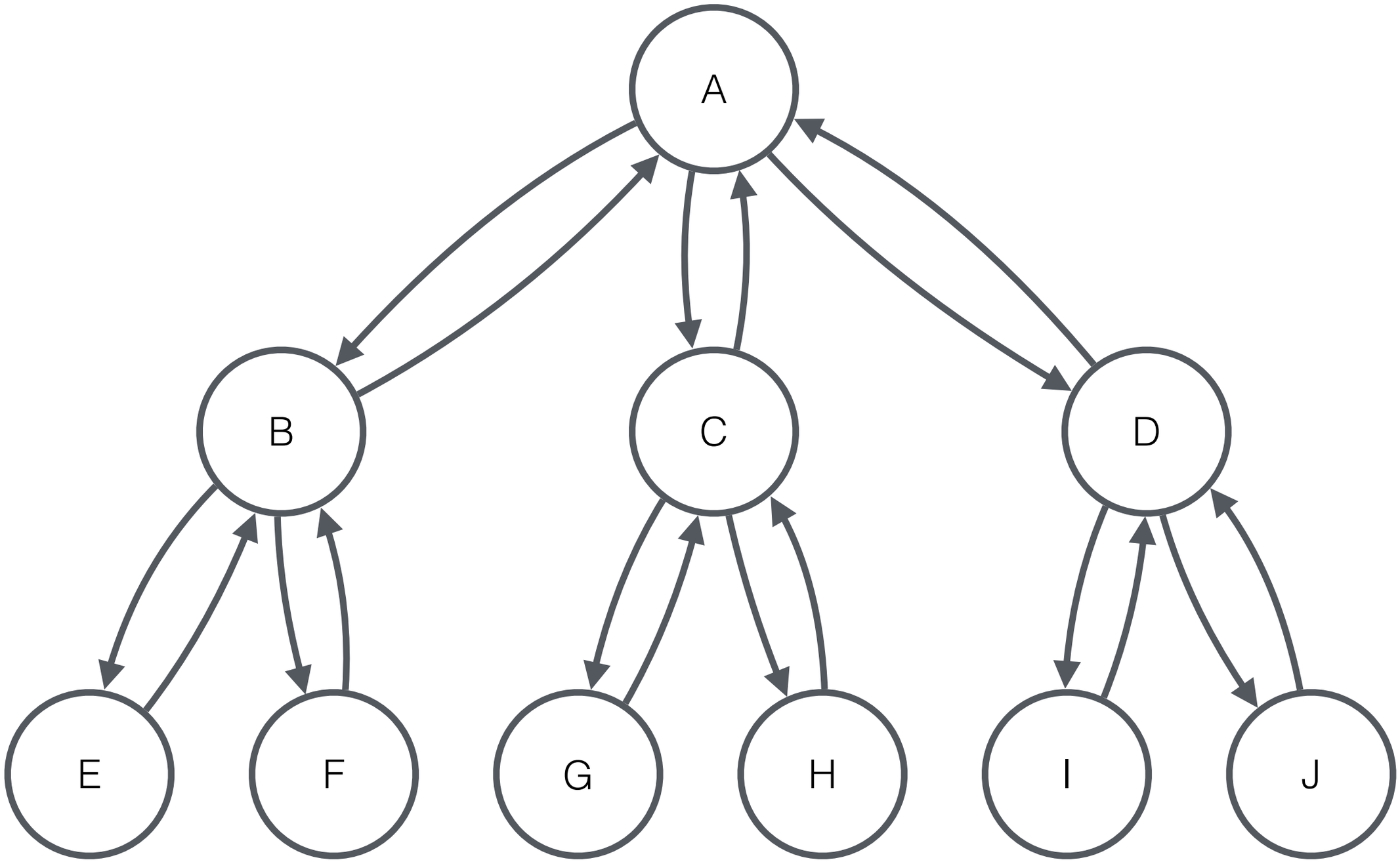


B[D]⇒B[E]

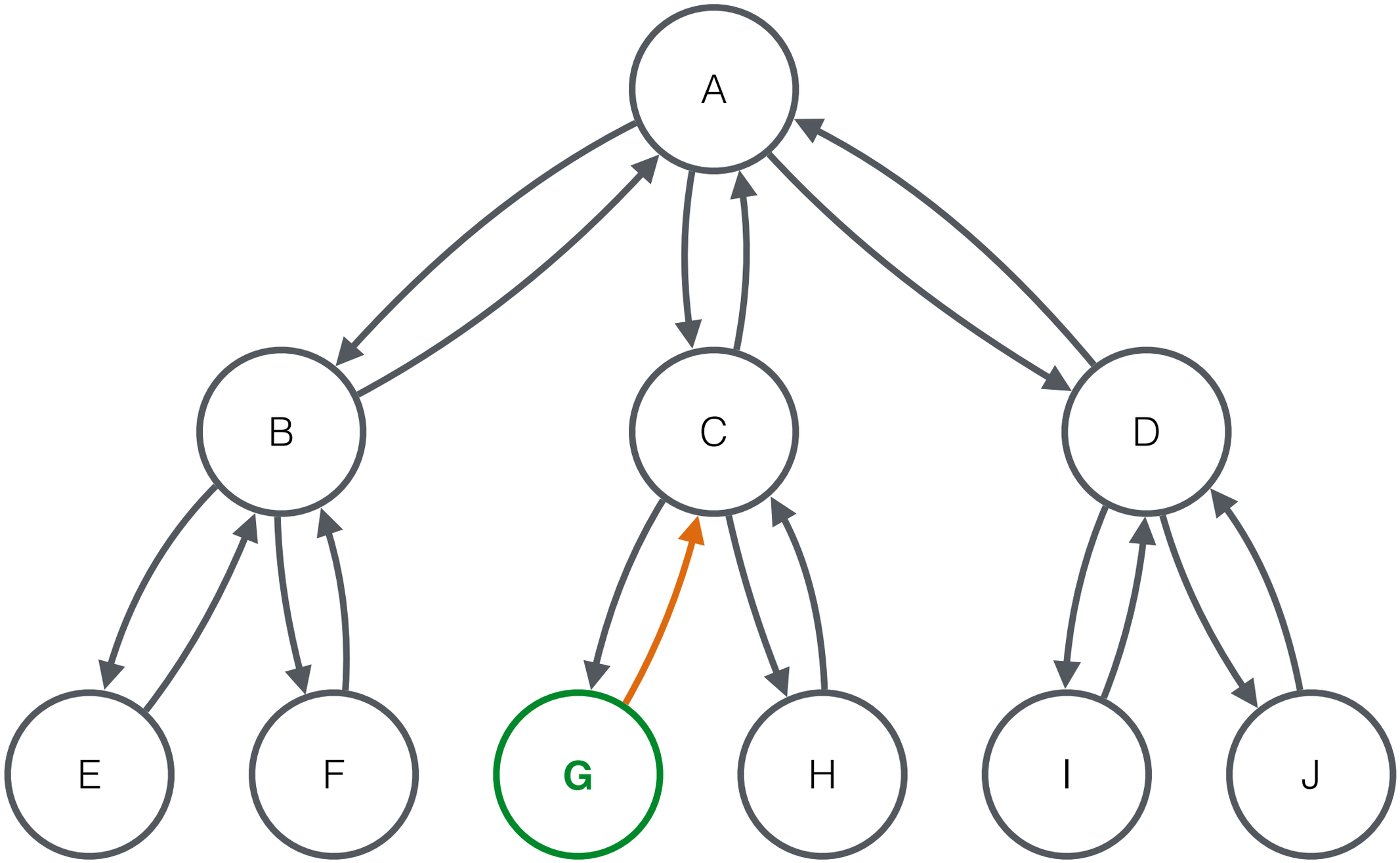


B[D]⇒B[E]

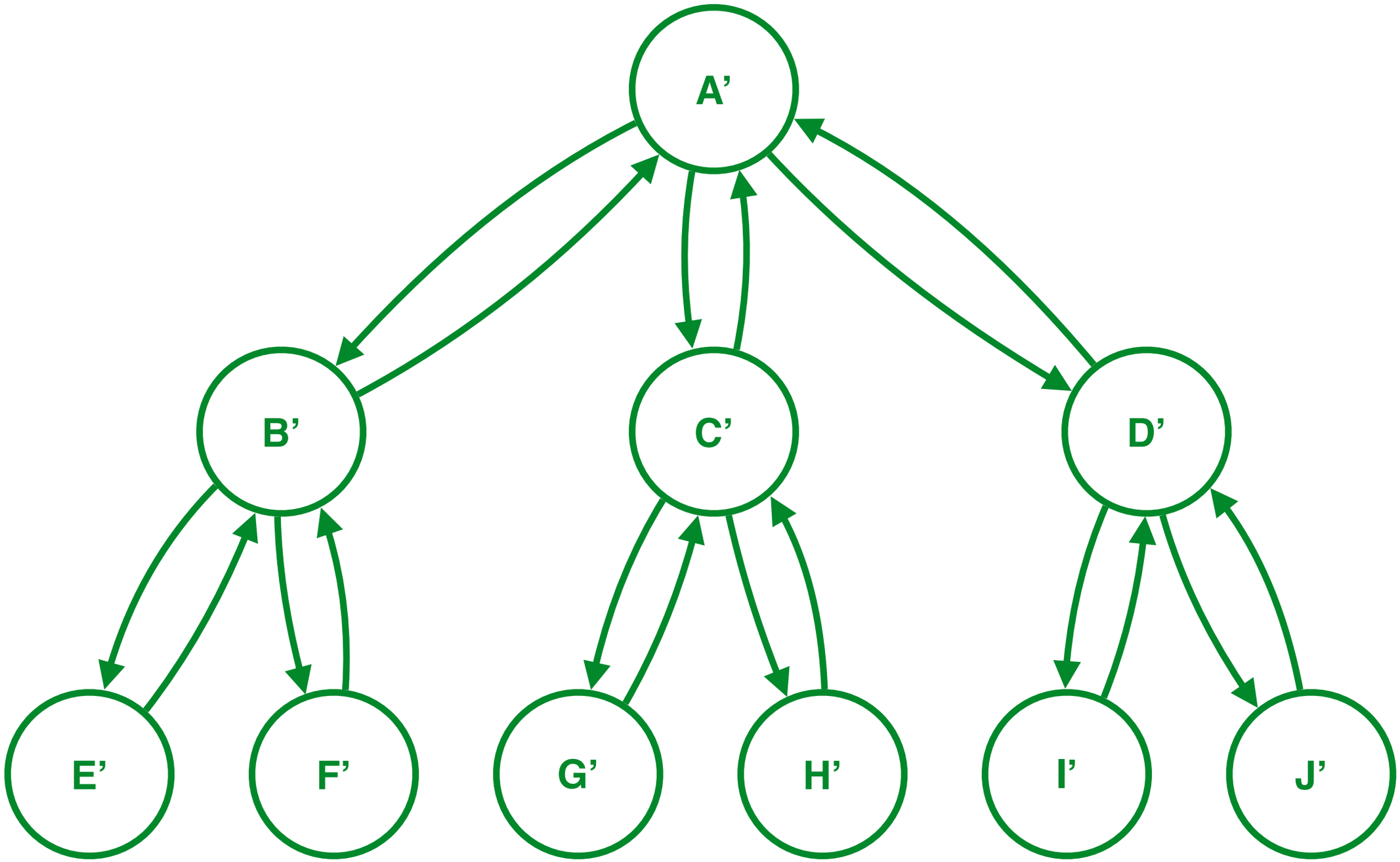



A
B
C
D
E
F
G
H
I
J

G⇒G’



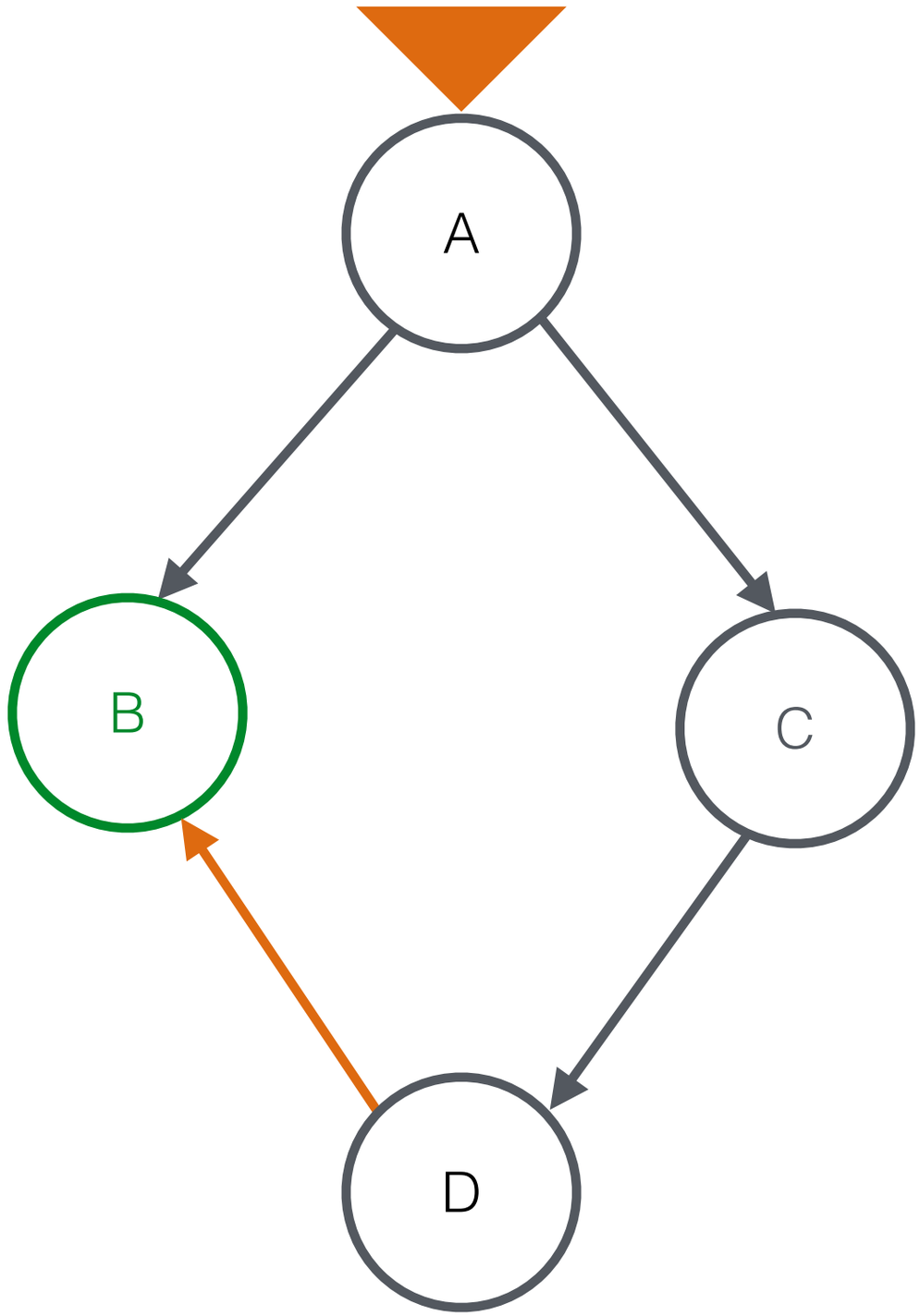
G⇒G’



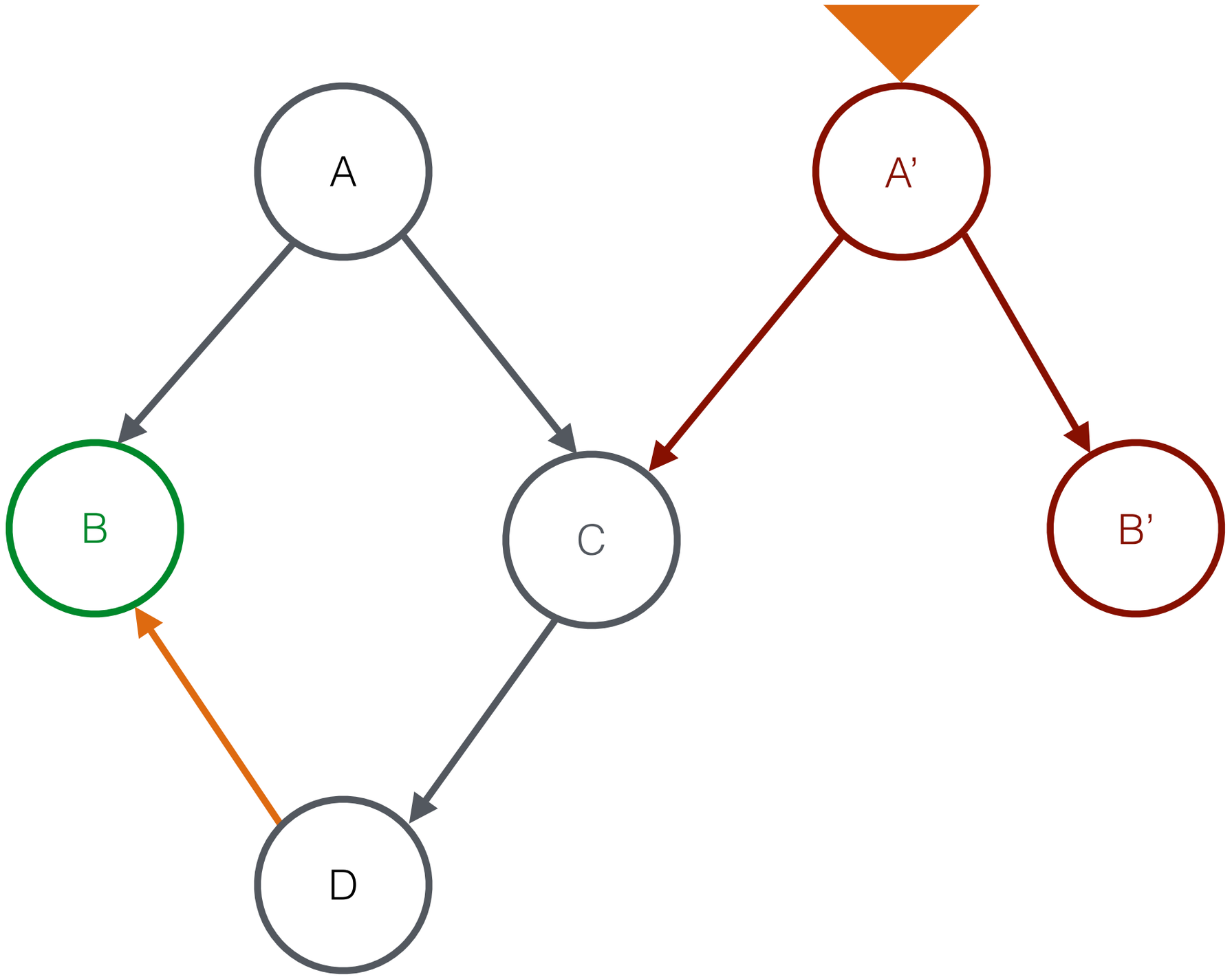
Pointer cycles are evil.

Pointers are evil.

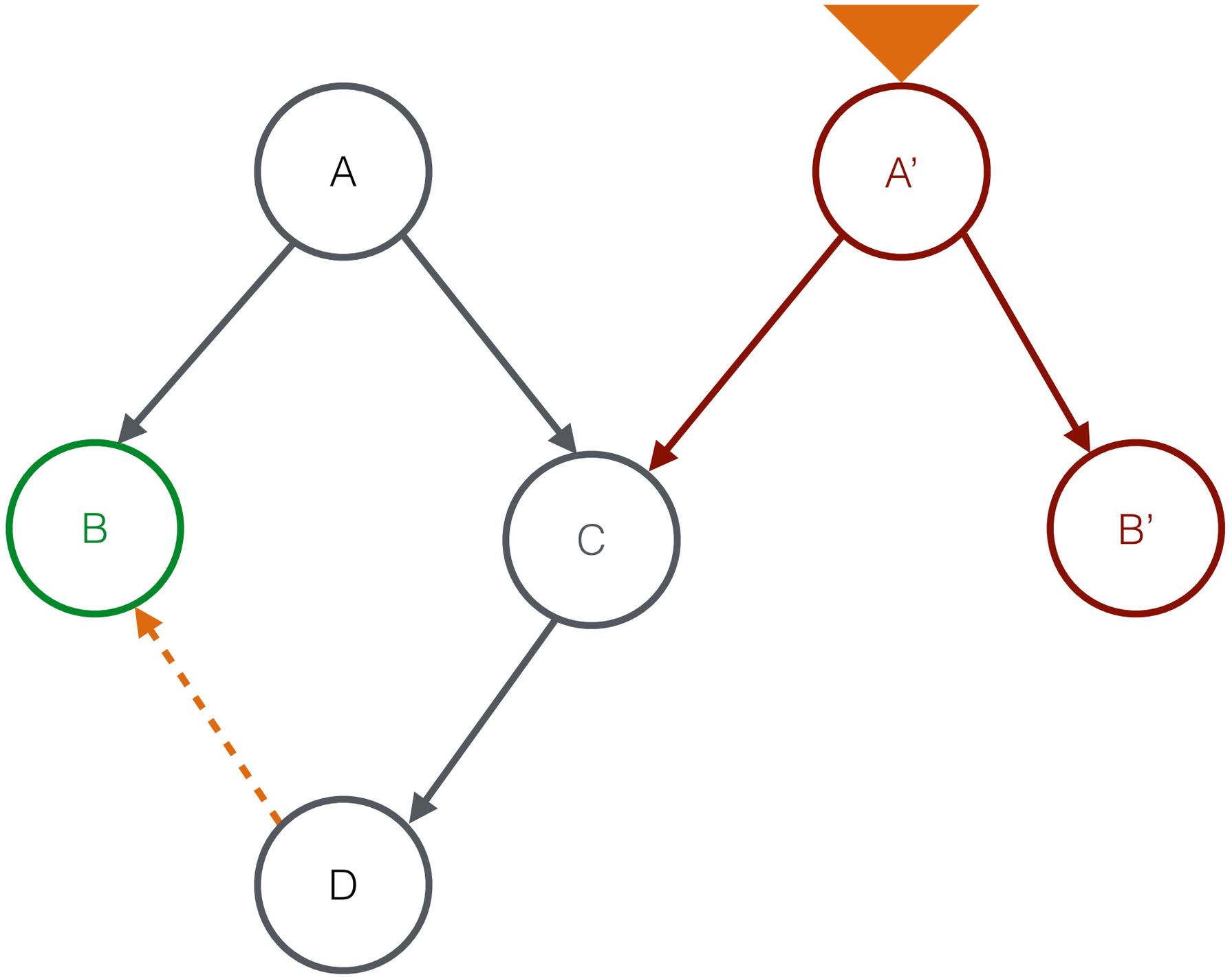
B⇒B’



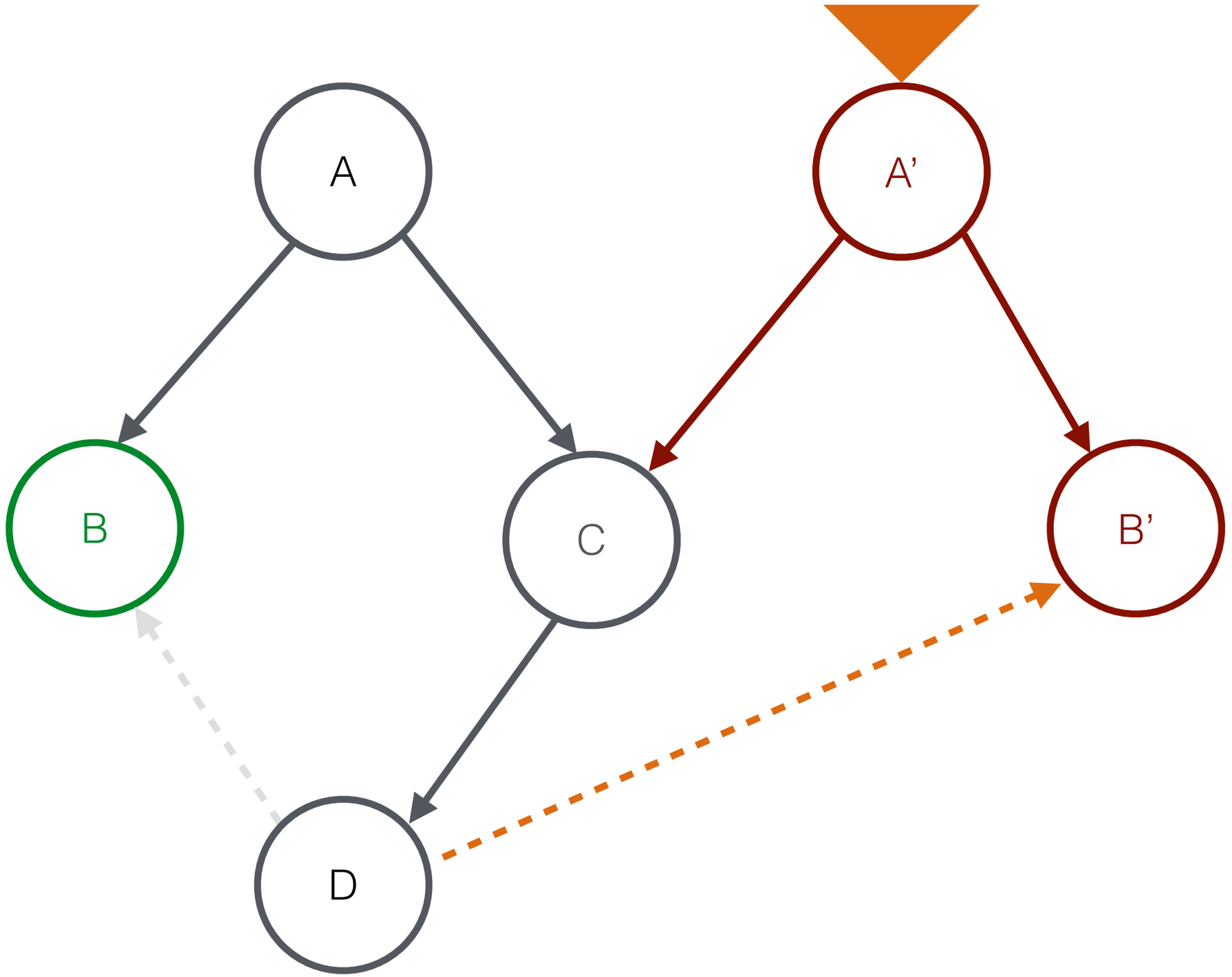
B⇒B’



B⇒B’

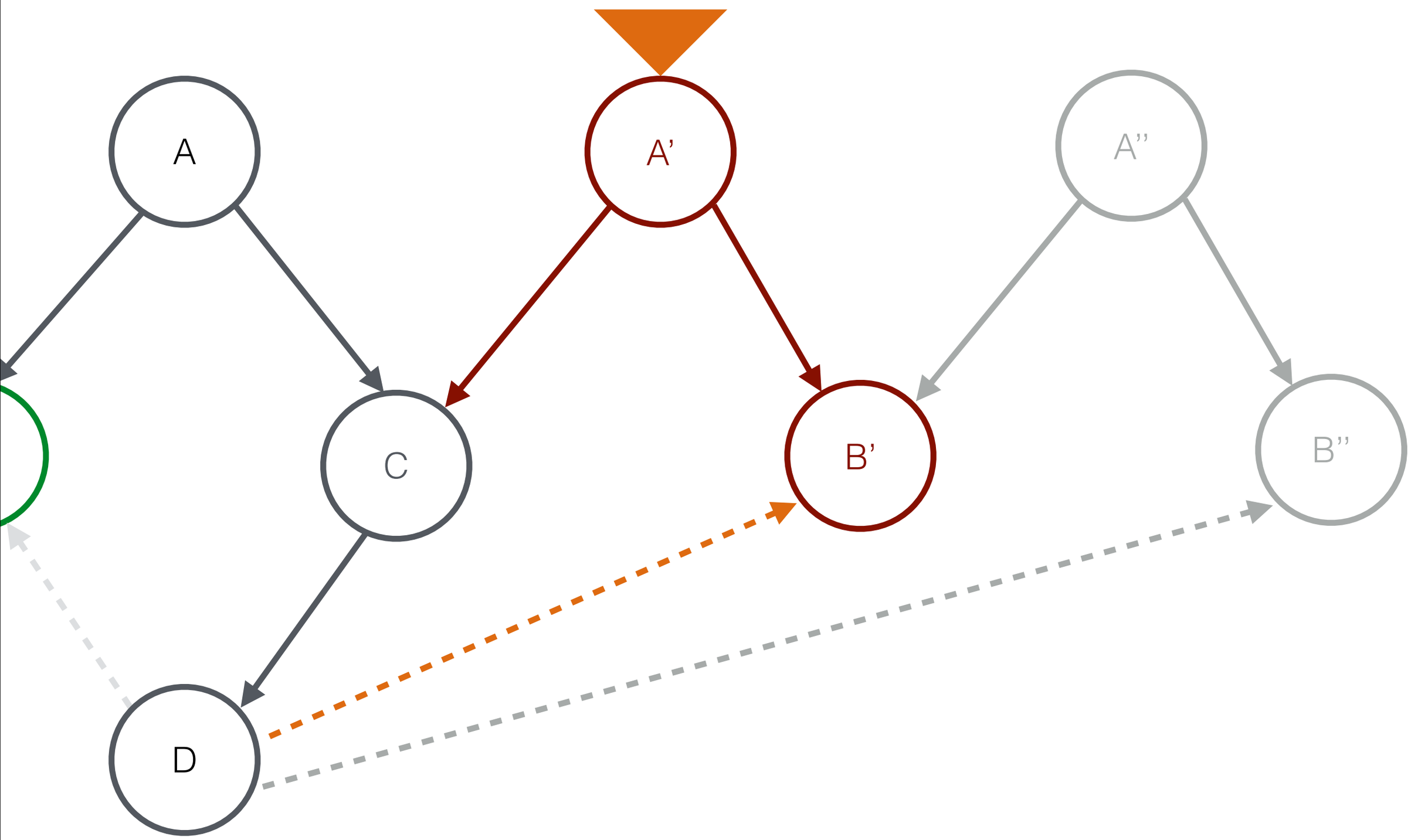


B⇒B’

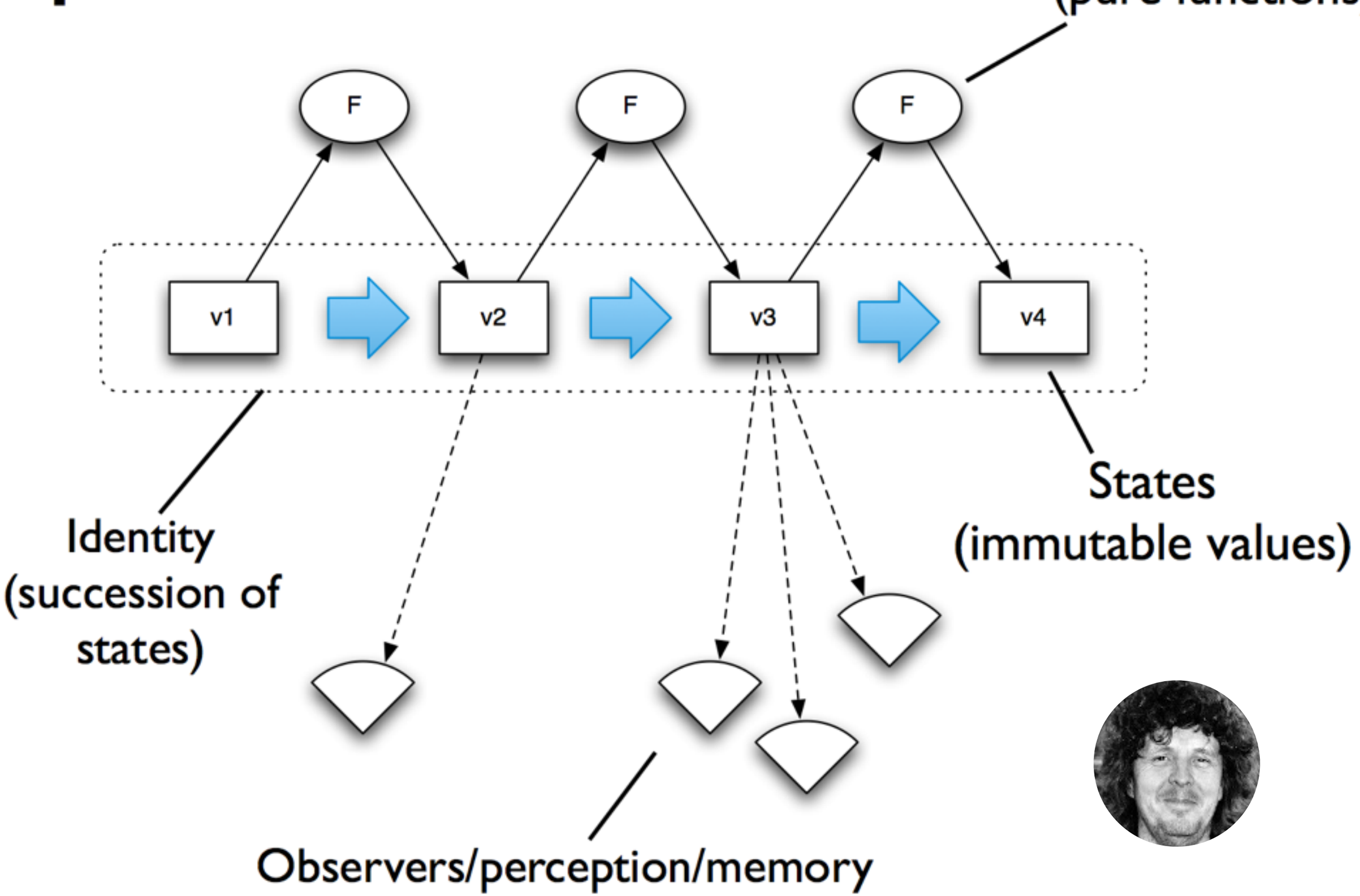


B⇒B’

B’⇒B’’

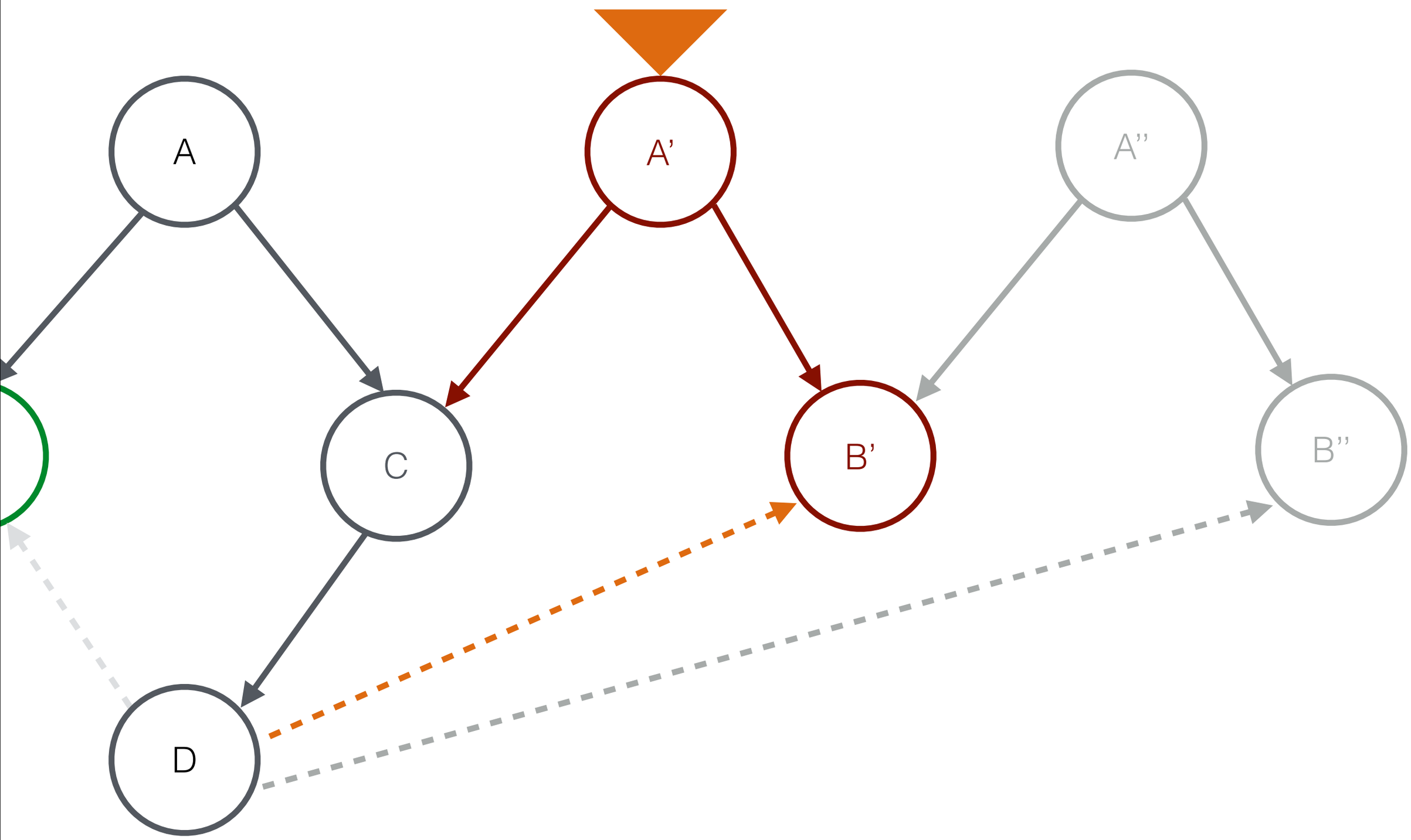


Epochal Time Model



B⇒B’

B’⇒B’’



Let’s talk about pointers and identities.

```
static byte memory[MEMORY_SIZE];

template <class T>
class Pointer {

public:

  Pointer(long p) {
    _p = p;
  }

  T deref() const {
    return !@#$%(memory + _p);
  }

  Pointer<T> add(int n) const {
    return Pointer(_p + n * sizeof(T));
  }

private:

  long _p;

};
```

```
static byte memory[MEMORY_SIZE];

template <class T>
class Reference {

public:

  Reference(long p) {
    _p = p;
  }

  T deref() const {
    return !@#$%(memory + _p);
  }



private:

  long _p;

};
```

```
template <class T>
class Reference {

public:

  Reference(long p) {
    _p = p;
  }

  T deref() const {
    return !@#$%(memory + _p);
  }



private:

  long _p;

};
```

```
template <class T>
class Reference {

public:

  Reference(long p, byte *memory) {
    _p = p;
    _memory = memory;
  }

  T deref() const {
    return !@#$%(_memory + _p);
  }

private:

  long _p;
  long _memory;

};
```

```
template <class T>
class Reference {

public:

  Reference(long p) {
    _p = p;
  }

  T deref(const byte *const memory) const {
    return !@#$%(memory + _p);
  }

private:

  long _p;

};
```

```
(defprotocol IDeref
  (deref [this]))

(defprotocol IDerefIn
  (deref-in [this context]))

(deftype Reference [p]
  IDerefIn
  (deref-in [_ memory]
    (!$%&* memory p)))
```

```clojure
(deftype DbRef [table id]
  (deref-in [_ db]
    (get-row db table id)))

(let [brandon (DbRef. :users 5)]
  (deref-in brandon *db*))
```

```clojure
(defn get-user [db id]
  (get-row db :users id))

(let [brandon 5]
  (get-user *db* brandon)
```

"http://example.com/api/users/5"

```
[:user 5]
```

5

Context is King

Symbols: The Original Identities

```
(def x "top-level")
```

```
cljs.user.x = "top-level";
```

```
(fn []
  x
  (let [x "local"]
    x))
```

var

local

```
function () {
  cljs.user.x;
  var x = "local";
  return x;
}
```

symbol resolved using context!

Context: Not Just For Identities

```
(def x "top-level")
```

```
cljs.user.x = "top-level";
```

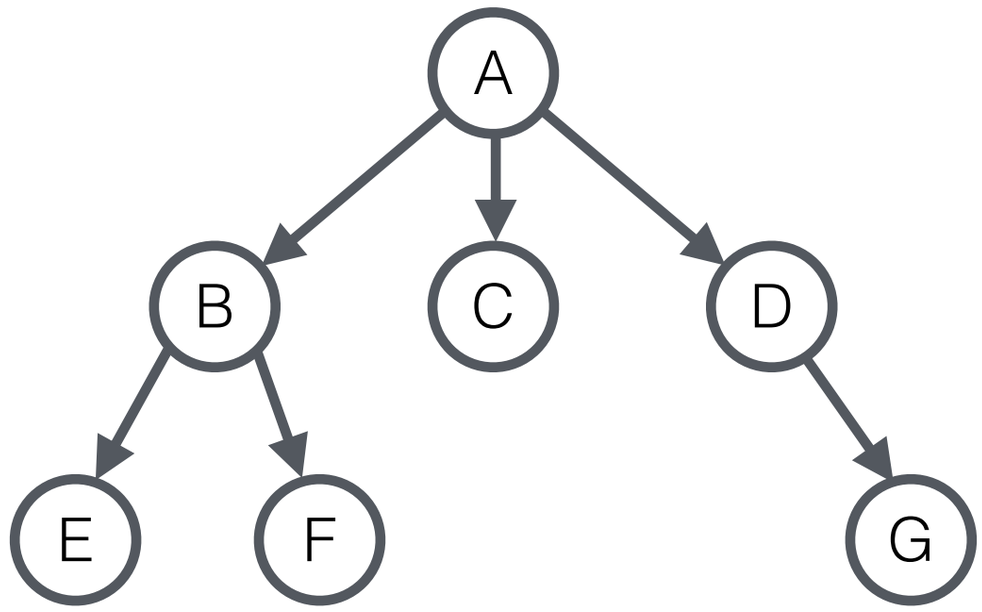
```
(fn []
  x
  (let [x "local"]
    x))
```

statement

tail position

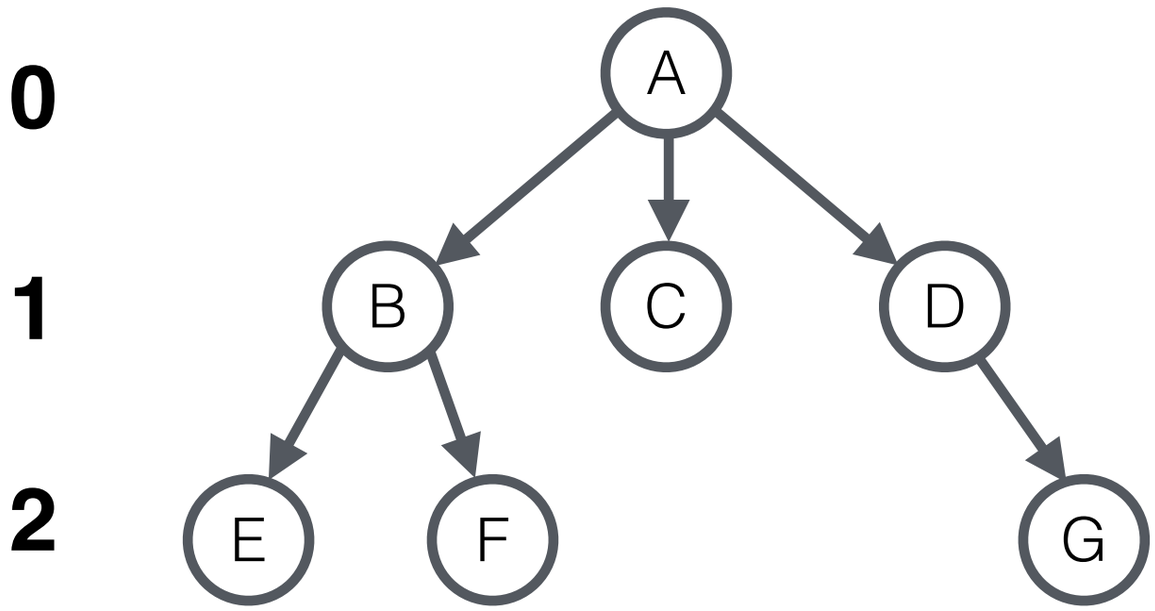
```
function () {
  cljs.user.x;
  var x = "local";
  return x;
}
```

contextual return!

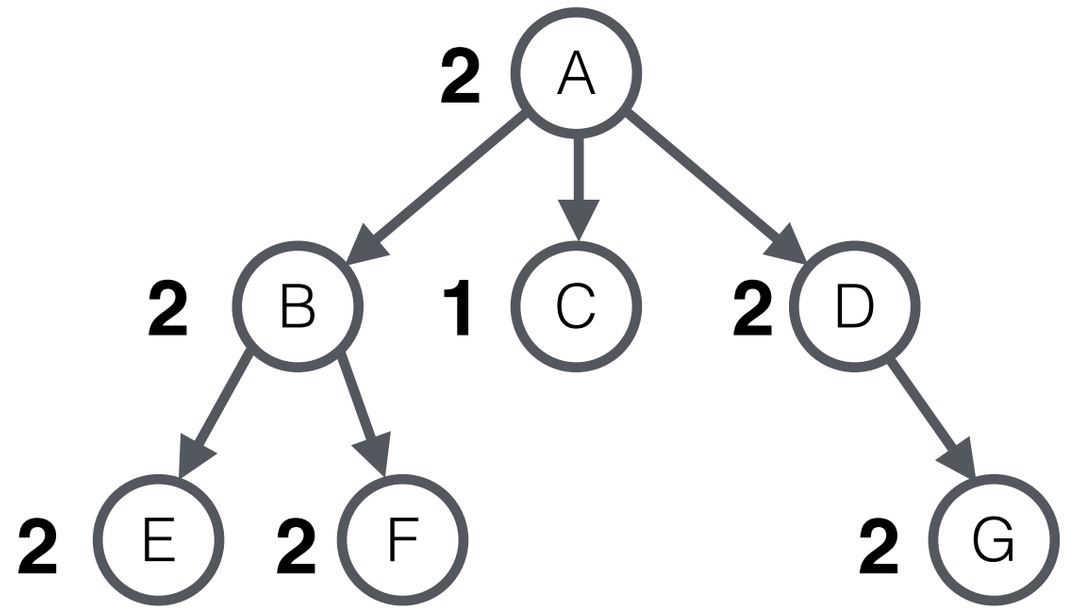



```
(def vector-tree
  ["A" ["B" ["E"] ["F"]] ["C"] ["D" ["G"]]])
```

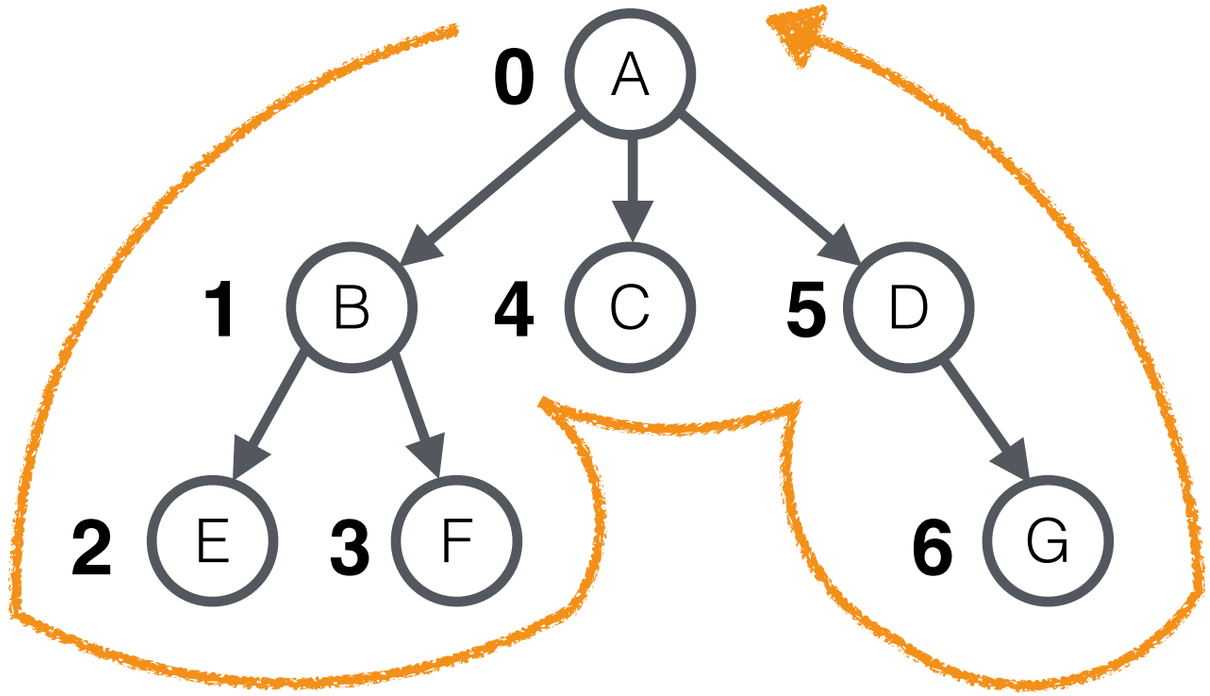
```
(def tree
  {:label "A"
   :children [{:label "B"
               :children [{:label "E"}
                          {:label "F"}]}
              {:label "C"}
              {:label "D"
               :children [{:label "G"}]}]})
```




```
(defn annotate-depth [node]
  (letfn [(f [node depth]
            (let [d (inc depth)
                  annotate-child #(f % d)]
              (-> node
                (assoc :depth depth)
                (update-in
                  [:children]
                  (mapv annotate-child %)))))]
    (f node 0)))
```

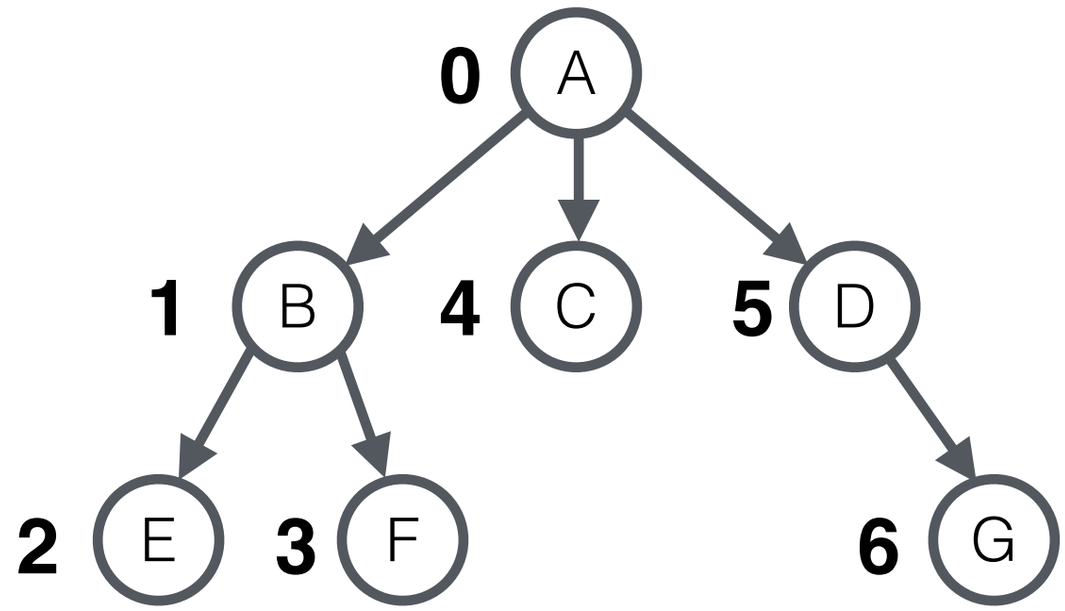



```
(defn annotate-max-depth [node]
  (let [{:keys [children]} node]
    (if (seq children)
      (let [children*
            (mapv annotate-max-depth children)]
        (assoc node
          :max-depth (apply max
                            (map :max-depth children*))
          :children children*))
      (assoc node :max-depth (:depth node)))))
```

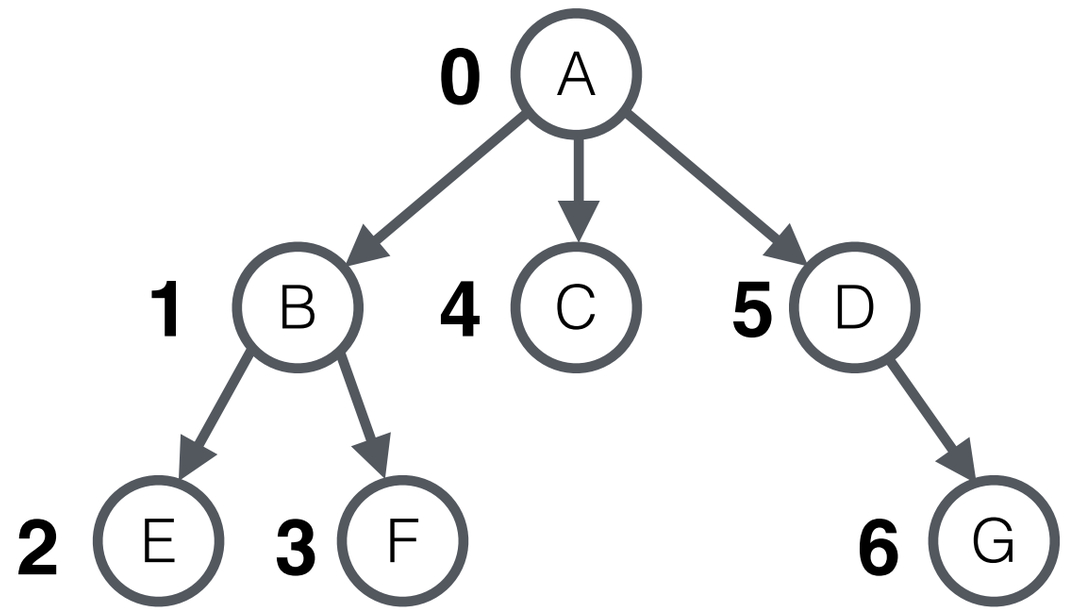


0 A
1 B
2 E
3 F
4 C
5 D
6 G

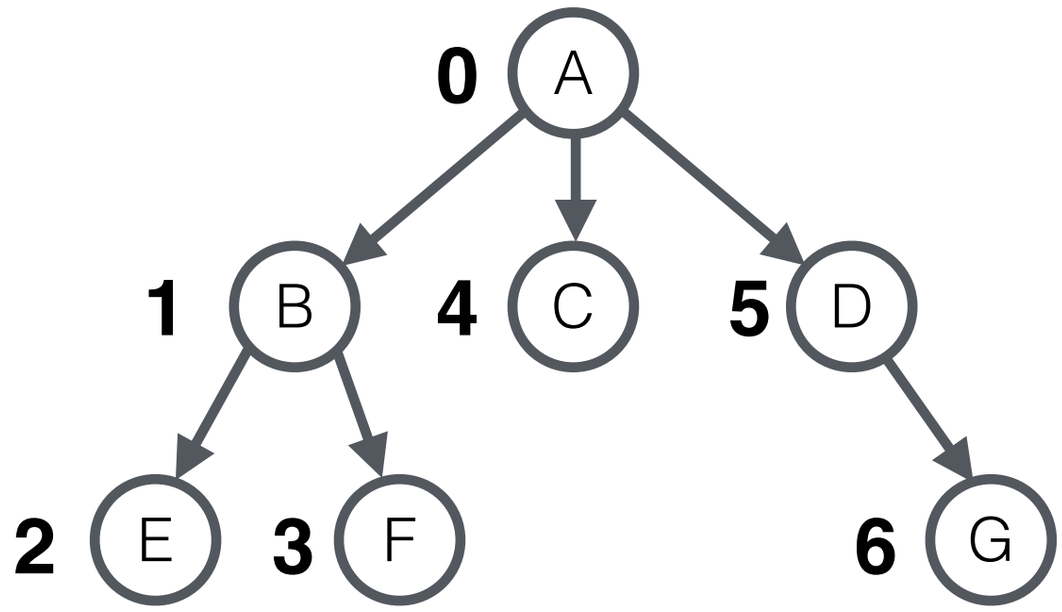
```
(defn print-depth-first-recursive [node]
  (letfn [(f [node index]
            (println index (:label node))
            (loop [nodes (:children node)
                   i index]
              (if (seq nodes)
                (recur (next nodes)
                       (f (first nodes) (inc i)))
                i)))]
    (f node 0))
  nil)
```



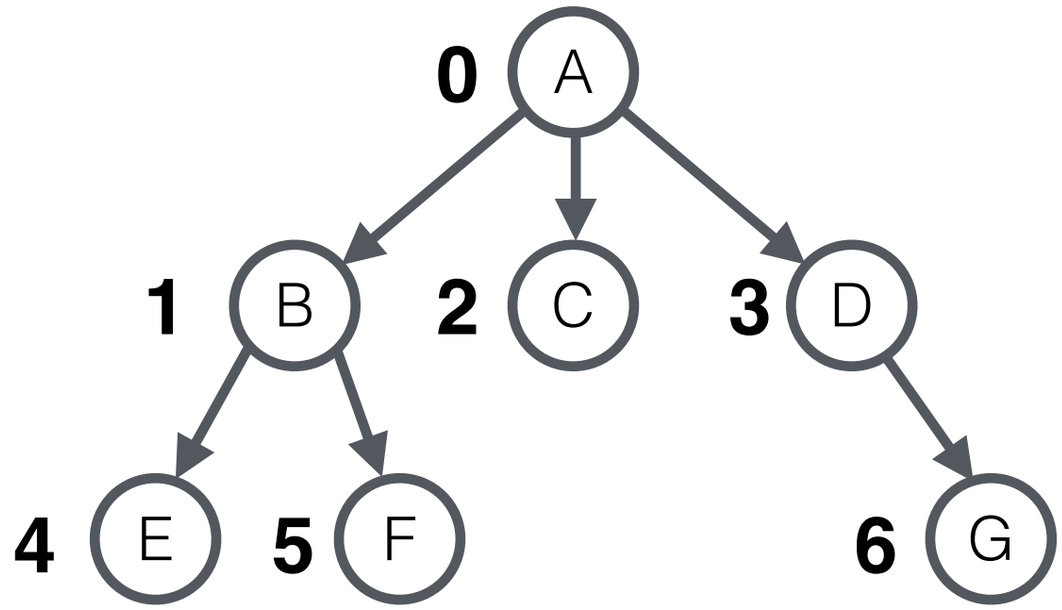

```
(defn number-depth-first-recursive [node]
  (letfn [(f [node index]
            (let [[max-index children*]
                  (reduce (fn [[i children] child]
                            (let [child* (f child (inc i))
                                  i* (:max-index child*)]
                              [i* (conj children child*)]))
                          [index []]
                          (:children node))]
              (assoc node
                     :index index
                     :children children*
                     :max-index max-index)))]
    (f node 0)))
```




```
(defn number-depth-first-stateful [node]
  (let [index (atom 0)]
    ((fn rec [n]
       (let [i @index]
         (swap! index inc)
         (-> n
           (assoc :index i)
           (update-in [:children] #(mapv rec %)))))
     node)))
```

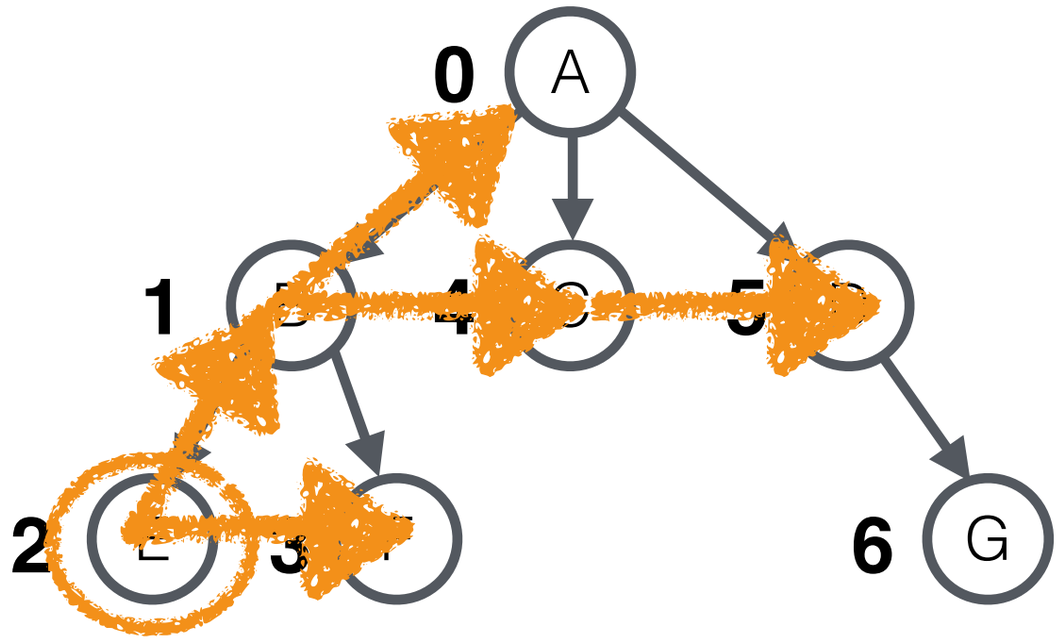



```
(defn print-depth-first-iterative [node]
  (loop [index 0
         nodes (list node)]
    (when (seq nodes)
      (let [[node & nodes*] nodes]
        (println index (:label node))
        (recur (inc index)
               (concat (:children node)
                       nodes*))))))
```

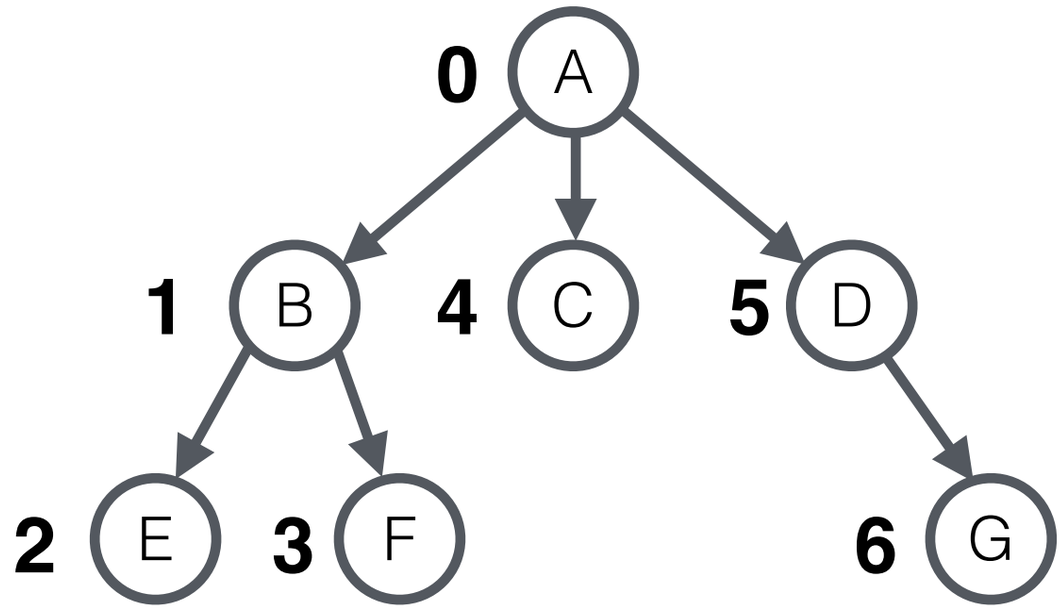


0 A
1 B
2 C
3 D
4 E
5 F
6 G

```
(defn print-breadth-first-iterative [node]
  (loop [index 0
         nodes (list node)]
    (when (seq nodes)
      (let [[node & nodes*] nodes]
        (println index (:label node))
        (recur (inc index)
               (concat nodes*
                       (:children node)))))))
```

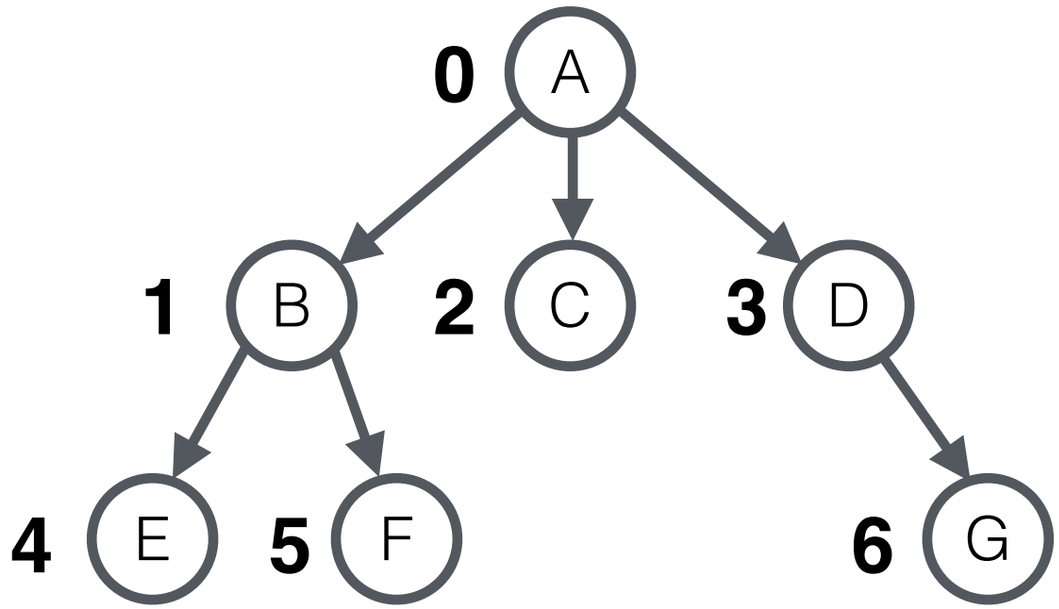



```
(defn make-zipper [root]
  (z/zipper (fn branch? [n]
              true)
            :children
            (fn make-node [n children]
              (assoc n :children (vec children)))
            root))
```




```clojure
(defn number-depth-first-zipper [node]
  (loop [index 0
         loc (make-zipper node)]
    (if (z/end? loc)
      (z/root loc)
      (let [loc* (z/edit loc assoc :index index)]
        (recur (inc index) (z/next loc*))))))
```

complete context as data!



Melt your brain:

Breadth-first numbering
Chris Okasaki

Backtracking Iterators
Jean-Christophe Filliâtre

Carefully consider:

identities + contexts

EXTRA SLIDES PAST HERE

A note about laziness

```
let cyclic = let x = 0 : y
                 y = 1 : x
             in  x

take 10 cyclic
-- [0,1,0,1,0,1,0,1,0,1]
```

```
(def cyclic
  (letfn [(x [] (cons 0 (lazy-seq (y))))
          (y [] (cons 1 (lazy-seq (x))))]
    (x)))

(take 10 cyclic)
;=> (0 1 0 1 0 1 0 1 0 1)
```

```clojure
(def cyclic
  (letfn [(x [] (cons 0 (lazy-seq (y))))
          (y [] (cons 1 (lazy-seq (x))))]
    (x)))

(take 10 cyclic)
;=> (0 1 0 1 0 1 0 1 0 1)
```

```clojure
(def cyclic
  (letfn [(x [] (cons 0 (new clojure.lang.LazySeq
                             (fn [] (y)))))
          (y [] (cons 1 (new clojure.lang.LazySeq
                             (fn [] (x)))))]
    (x)))
```