

Persistent Data Structures

Dnepropetrovsk Lambda Club

Pavel Forkert



A man is standing on a rocky cliff edge, holding a smartphone to take a selfie. He is wearing a dark green polo shirt, blue jeans, and sunglasses. The background is a vast, turquoise-colored sea meeting a rocky coastline under a clear blue sky.

@fxposter

**Immutable
Persistent
Functional**

Data Structures

Data Structures



Ephemeral



Partially Persistent



Fully Persistent

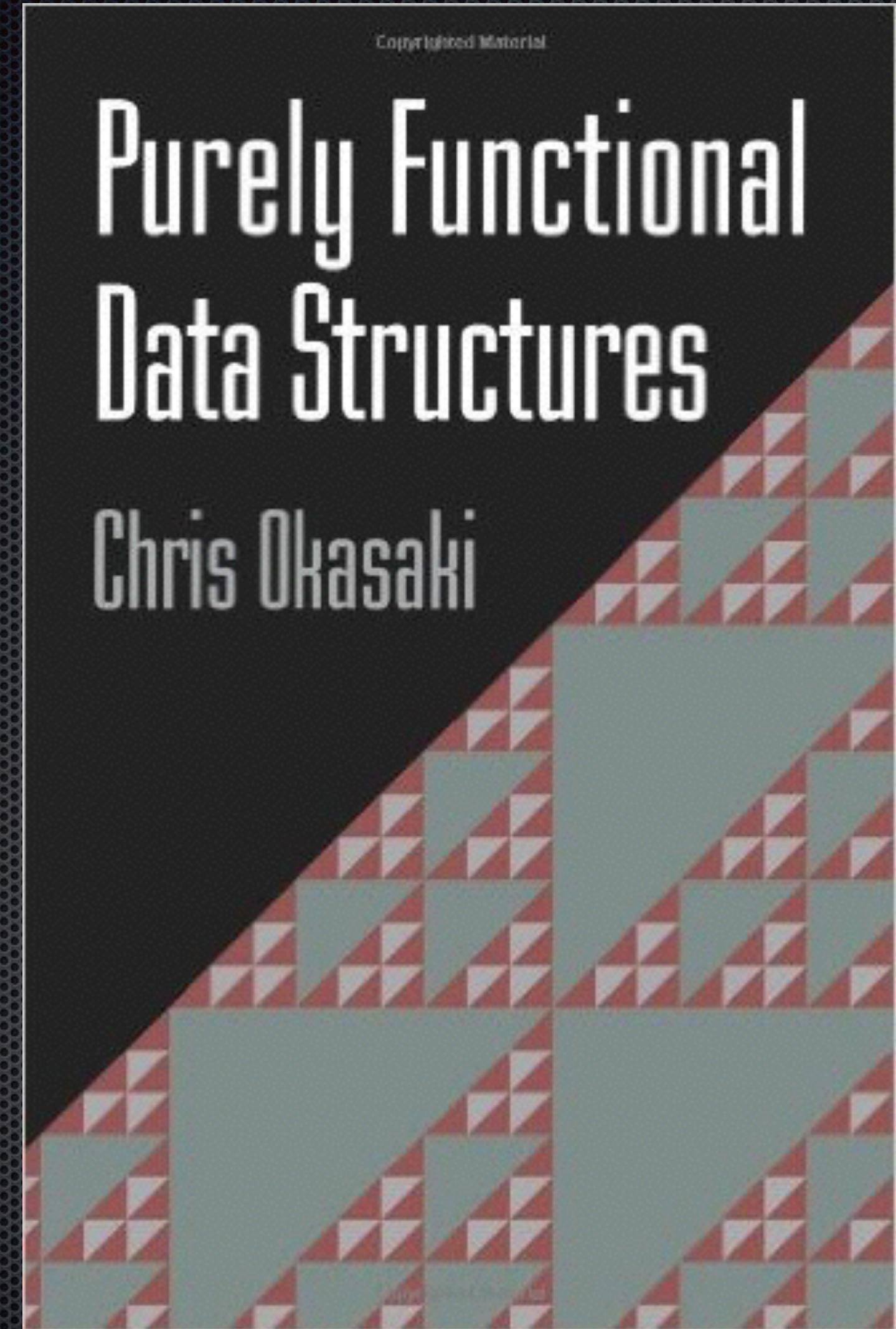
Why?

Reasoning about code

Concurrency

History (structural sharing)

Performance (sometimes)



<http://www.amazon.com/Purely-Functional-Structures-Chris-Okasaki/dp/0521663504>

Structures

1. LinkedList / Stack
2. Queue
3. TreeMap
4. Vector / Array / ArrayList
5. HashMap
6. Set

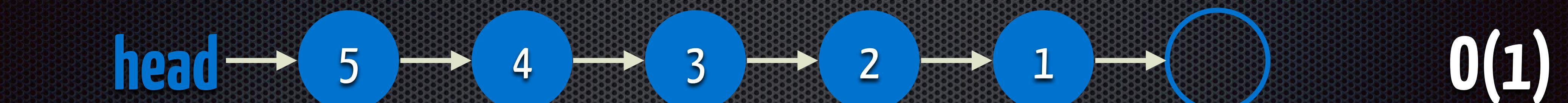
Linked List

Singly Linked List



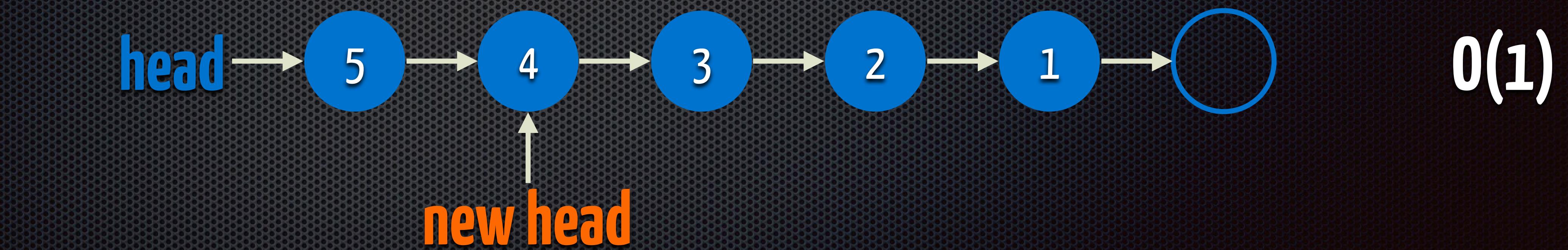
Linked List

Push



$O(1)$

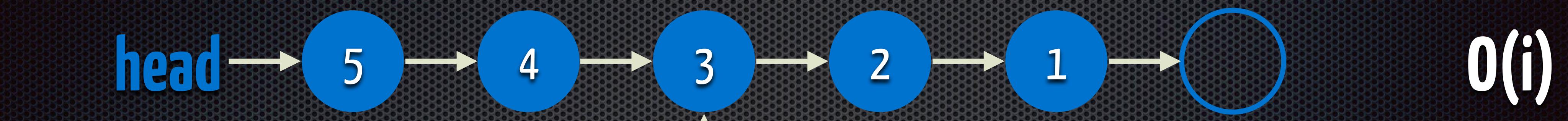
Pop



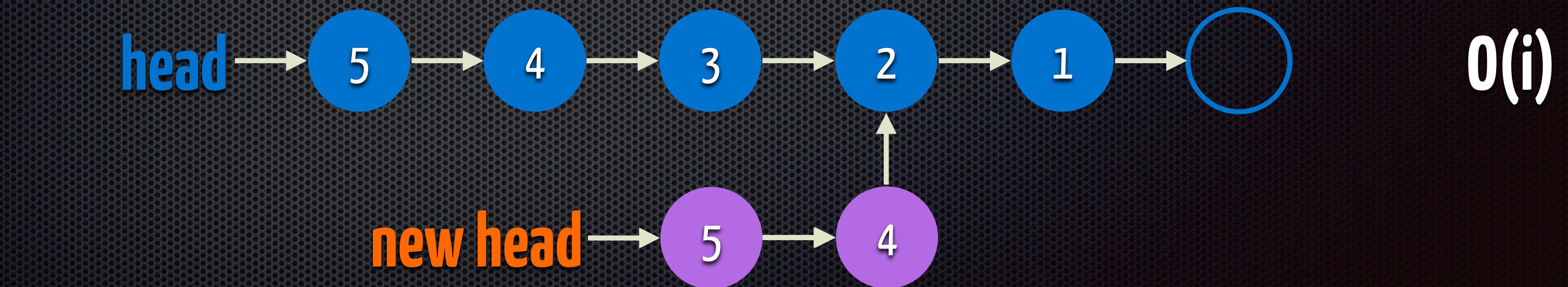
$O(1)$

Linked List

Insert



Remove



Stack

Same as Linked List

Queue

Linked List + Linked List **or** Linked List + Vector

```
class Queue {
    constructor(dequeueList, enqueueList) {
        this.dequeueList = dequeueList;
        this.enqueueList = enqueueList;
    }

    enqueue(value) { // O(1)
        if (this.dequeueList == null) {
            return new Queue(List.empty().push(value), this.enqueueList);
        } else {
            return new Queue(this.dequeueList, (this.enqueueList || List.empty()).push(value));
        }
    }

    peek() { // O(1)
        if (this.dequeueList == null) {
            throw EMPTY;
        } else {
            return this.dequeueList.head();
        }
    }

    dequeue() { // effectively O(1) time (O(n) sometimes)
        if (this.dequeueList == null) {
            throw EMPTY;
        } else {
            let tail = this.dequeueList.tail();
            if (tail.isEmpty()) {
                return new Queue(this.enqueueList ? this.enqueueList.reverse() : null, null);
            } else {
                return new Queue(tail, this.enqueueList);
            }
        }
    }
}
```

TreeMap

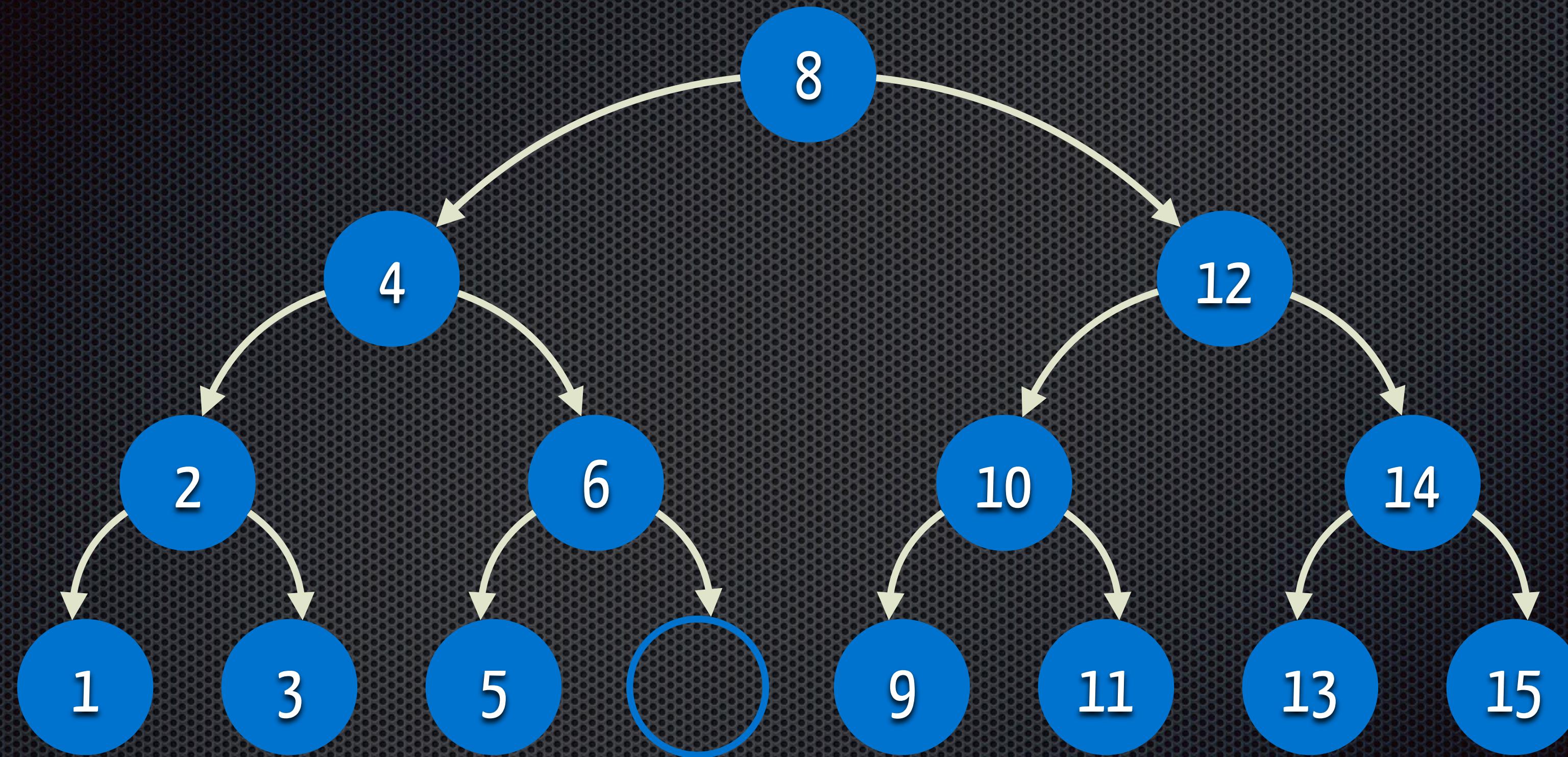
Map: (key → value) relationship

Tree: implementation detail

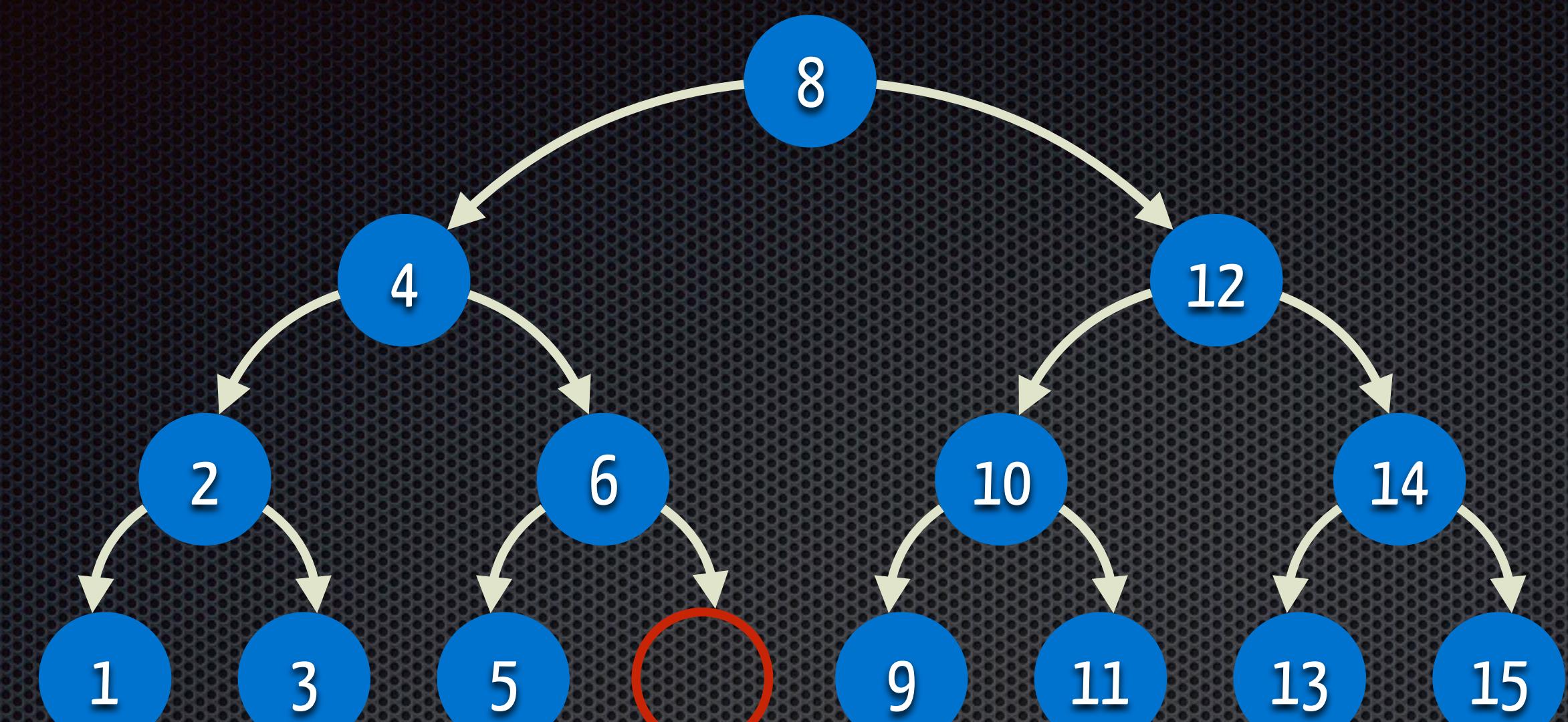
TreeMap

Binary Trees
Red-Black Trees

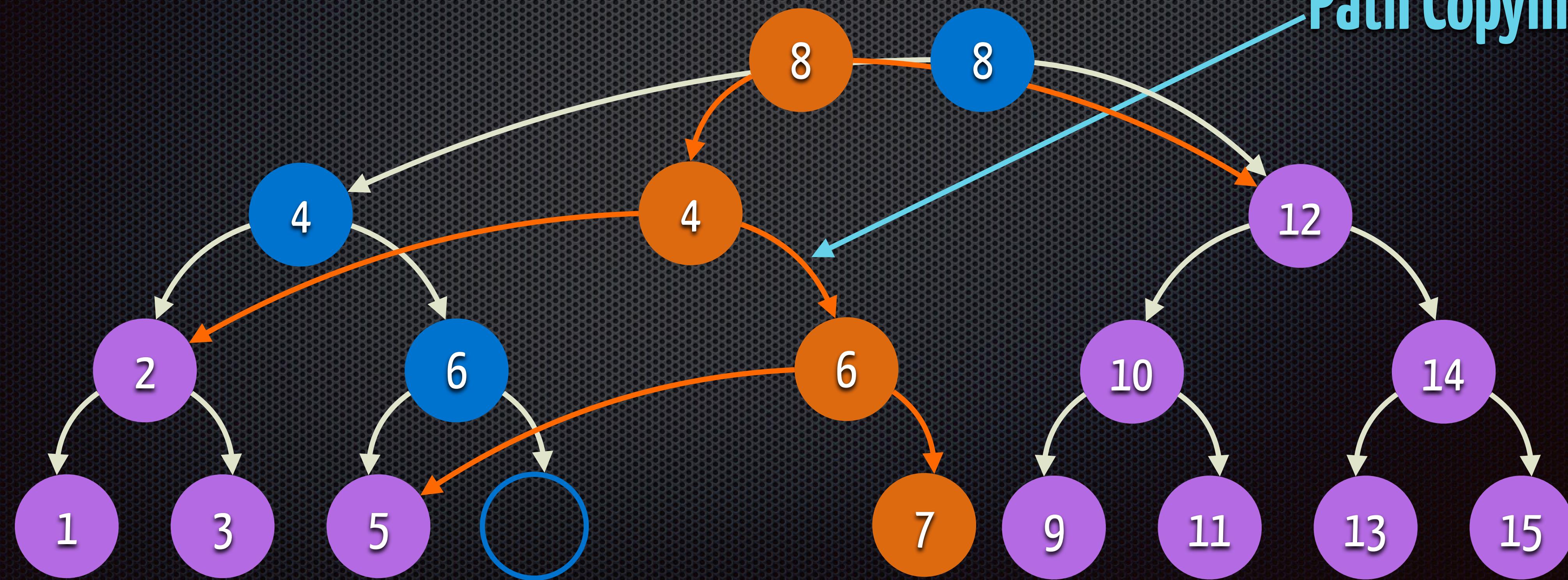
Binary Trees



Add
 $O(\log_2 n)$

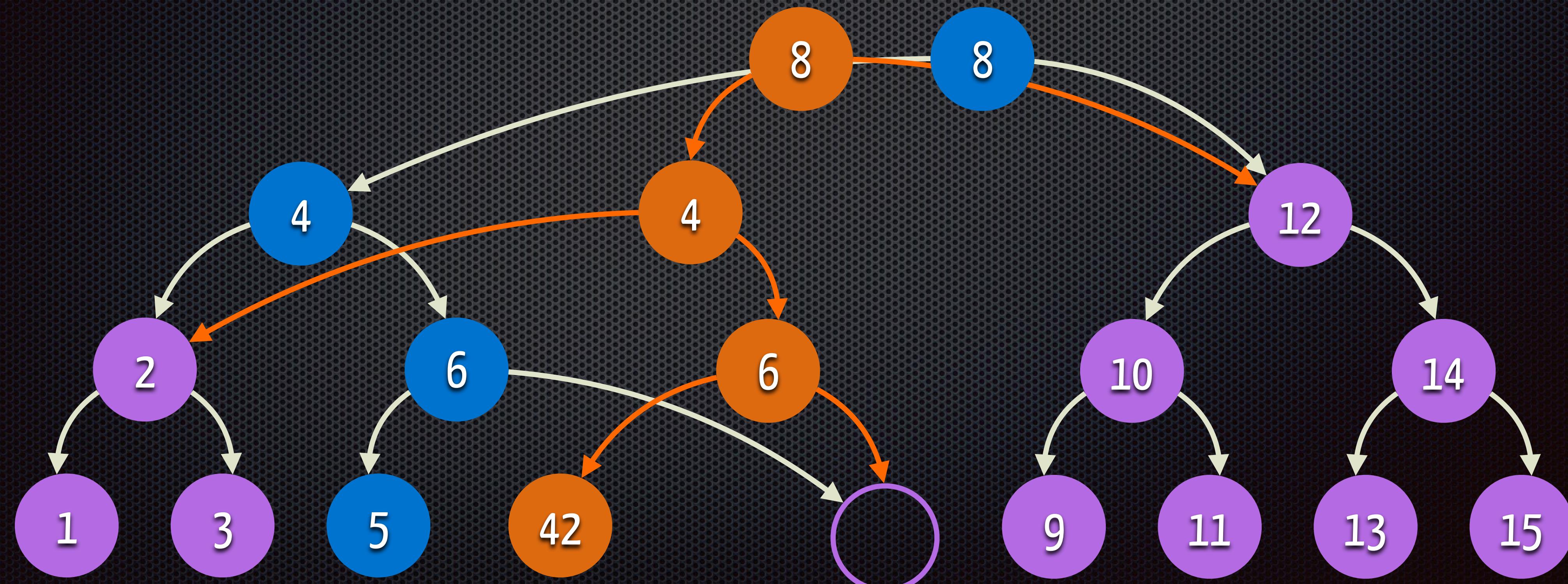
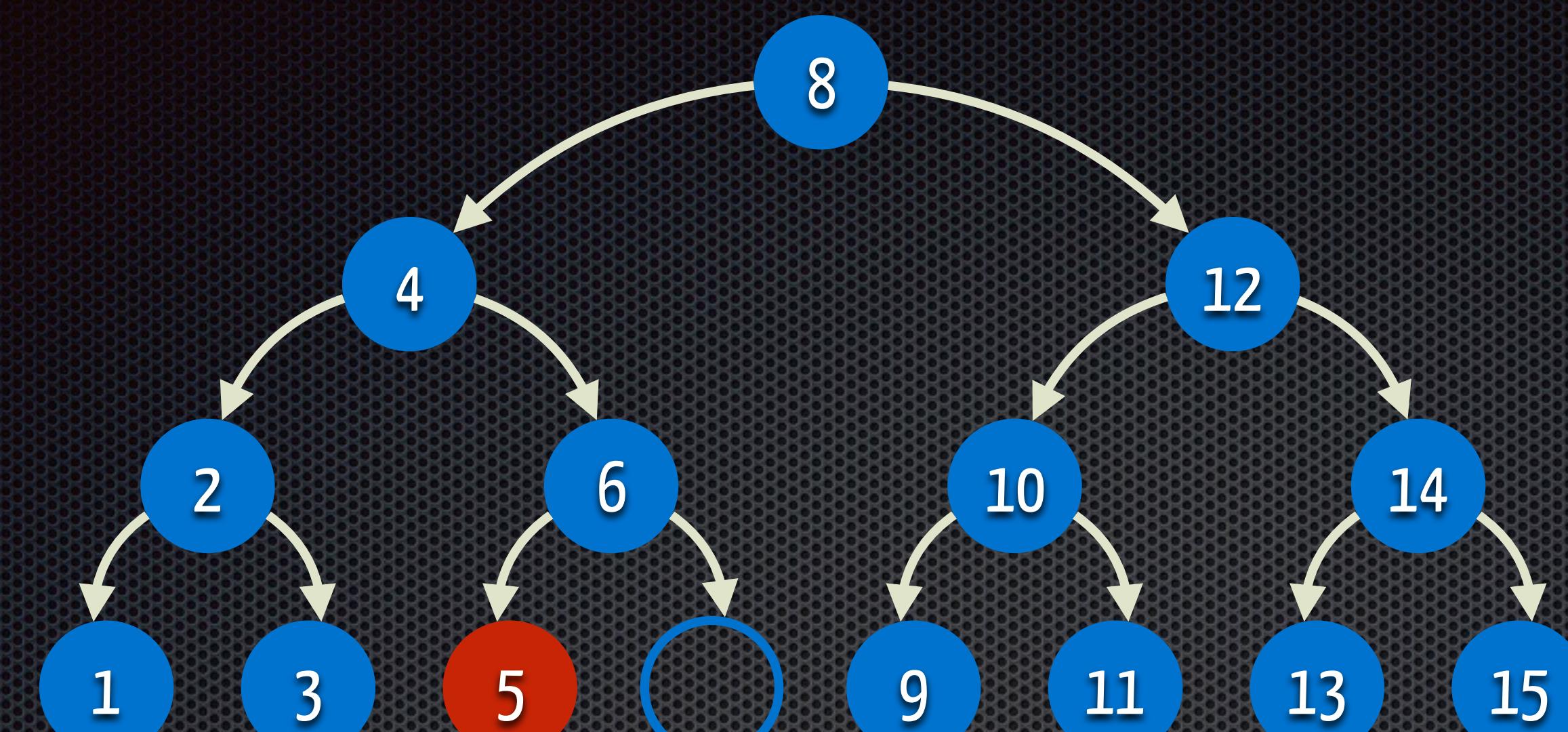


Path Copying



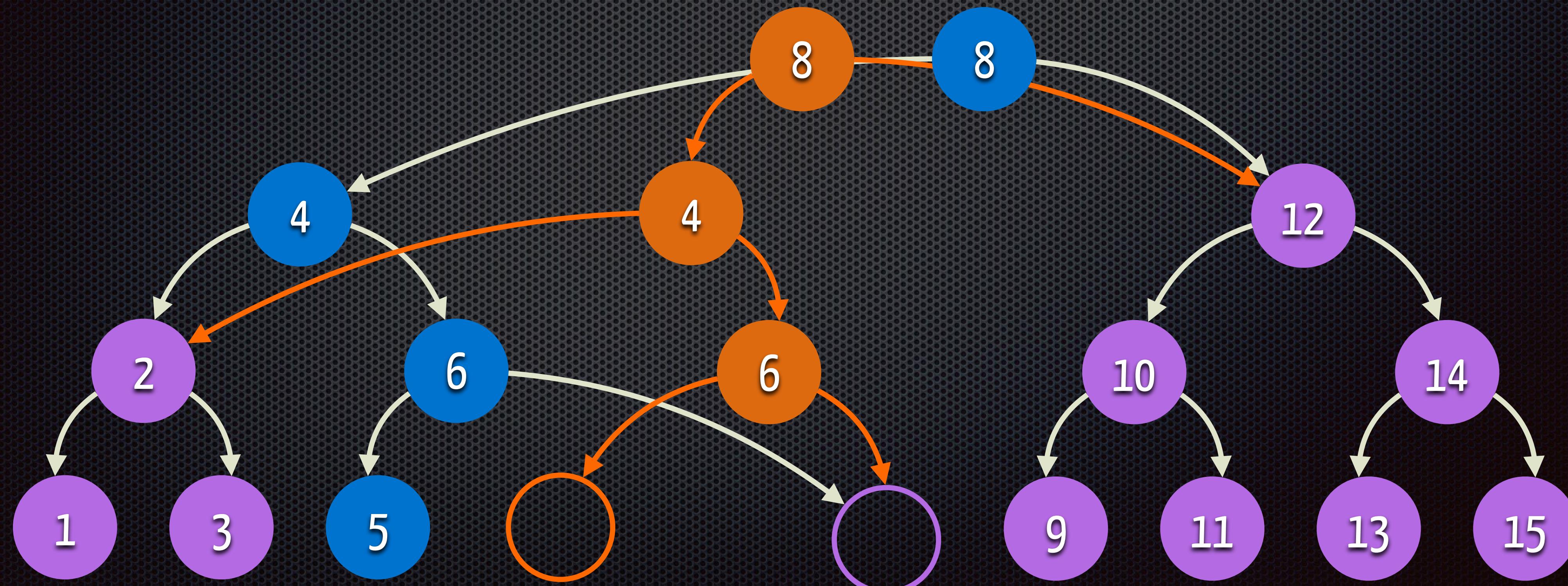
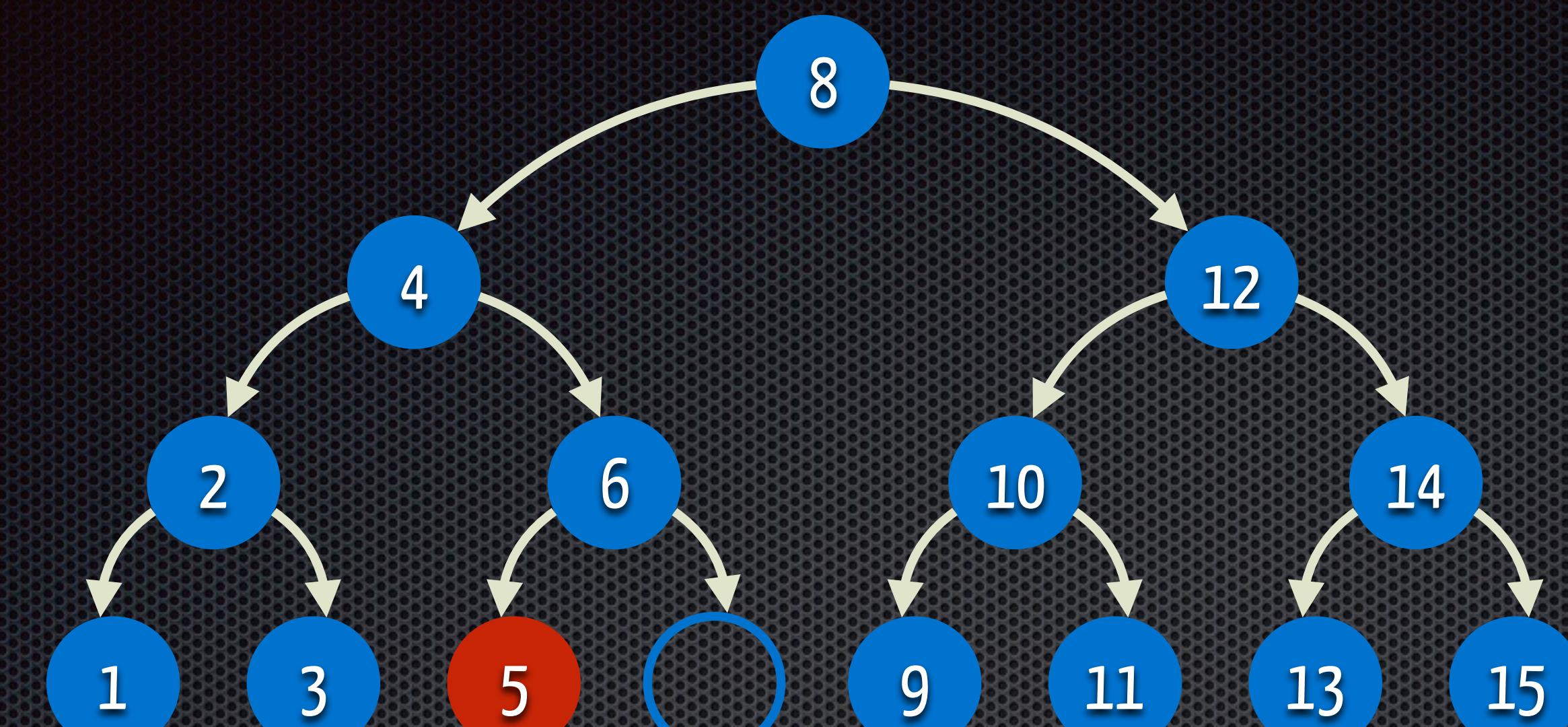
Update

$O(\log_2 n)$



Remove

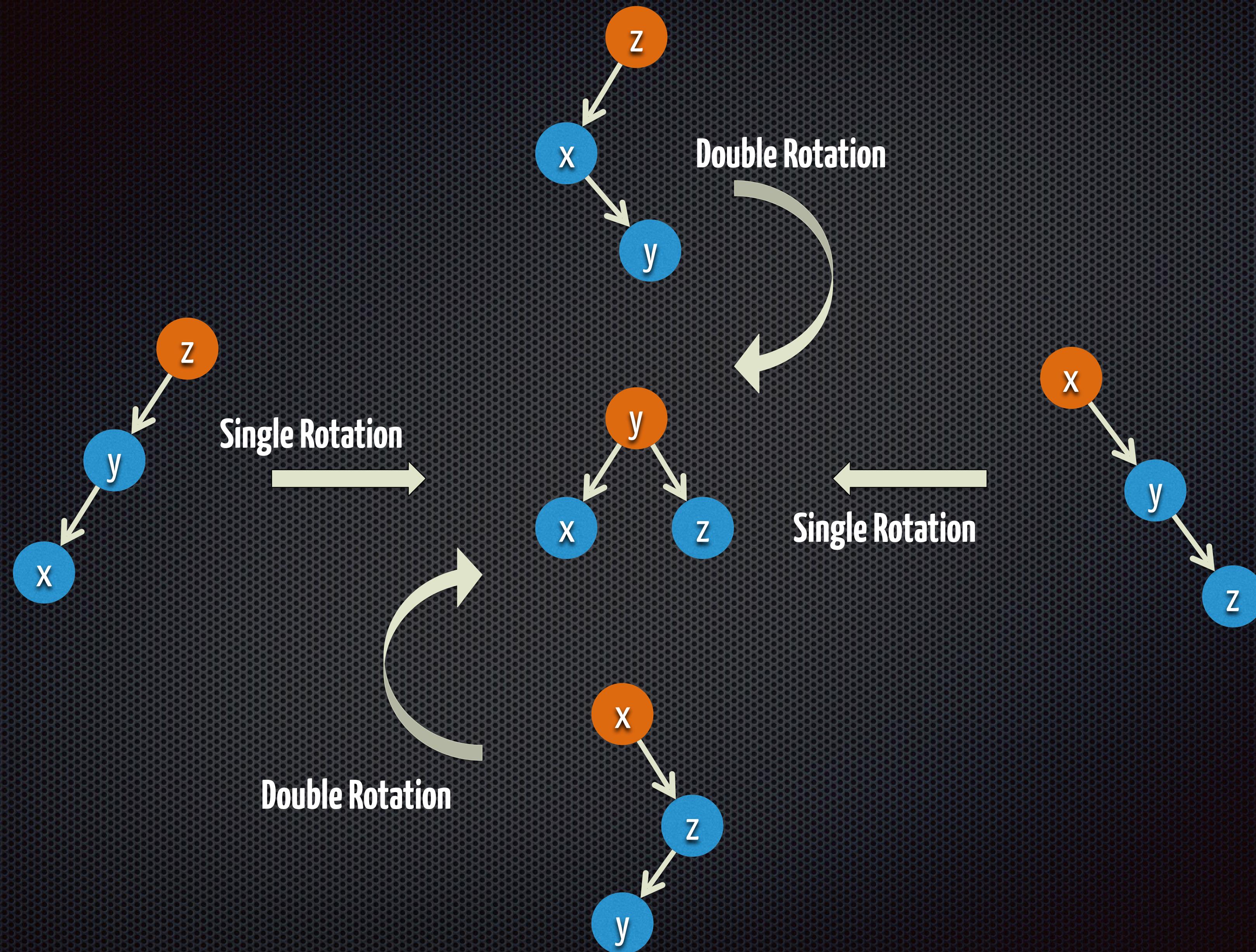
$O(\log_2 n)$



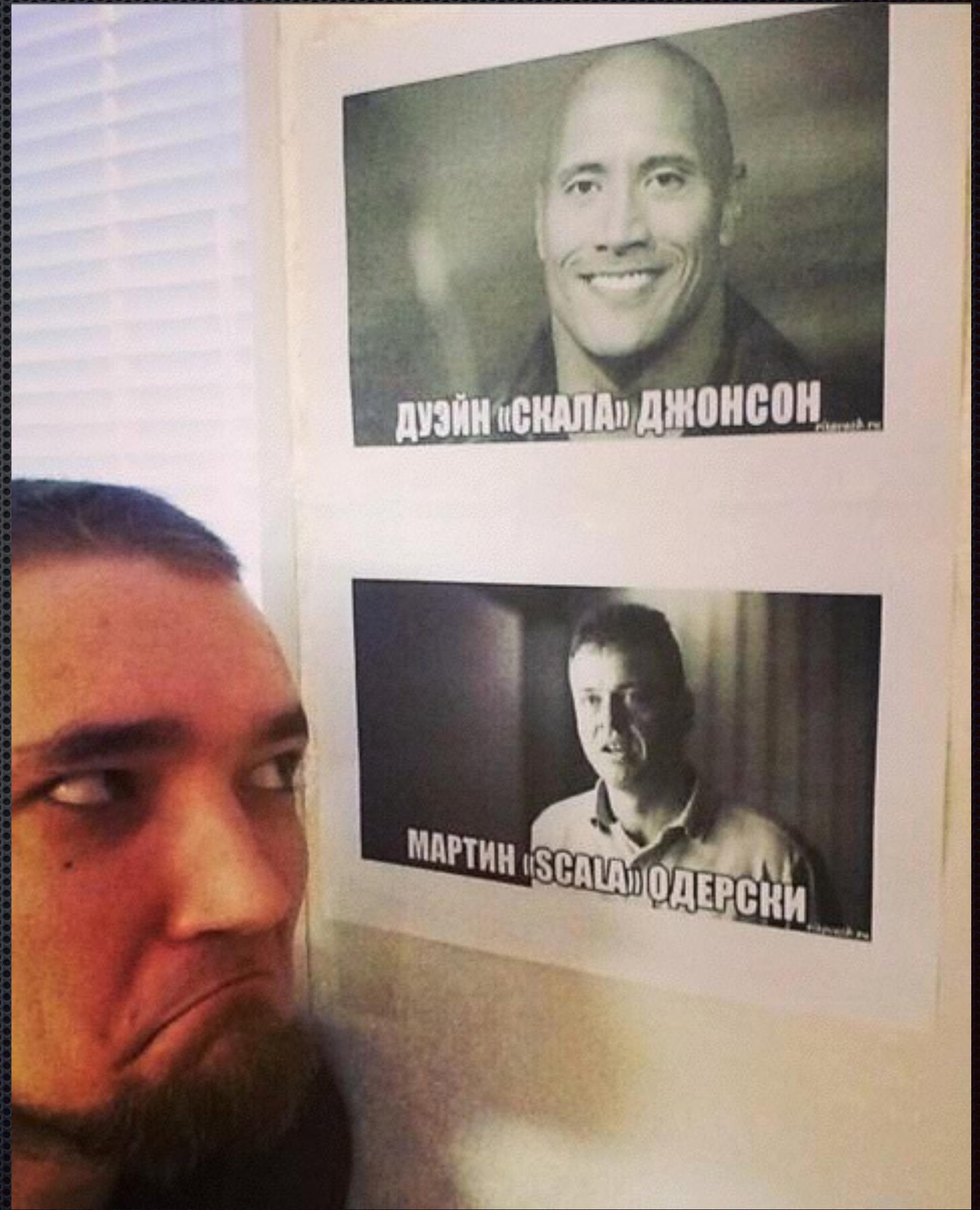
Red Black Trees

Red invariant: No red node has red parent

Black invariant: Every root-to-leaf path contains the same number of black nodes

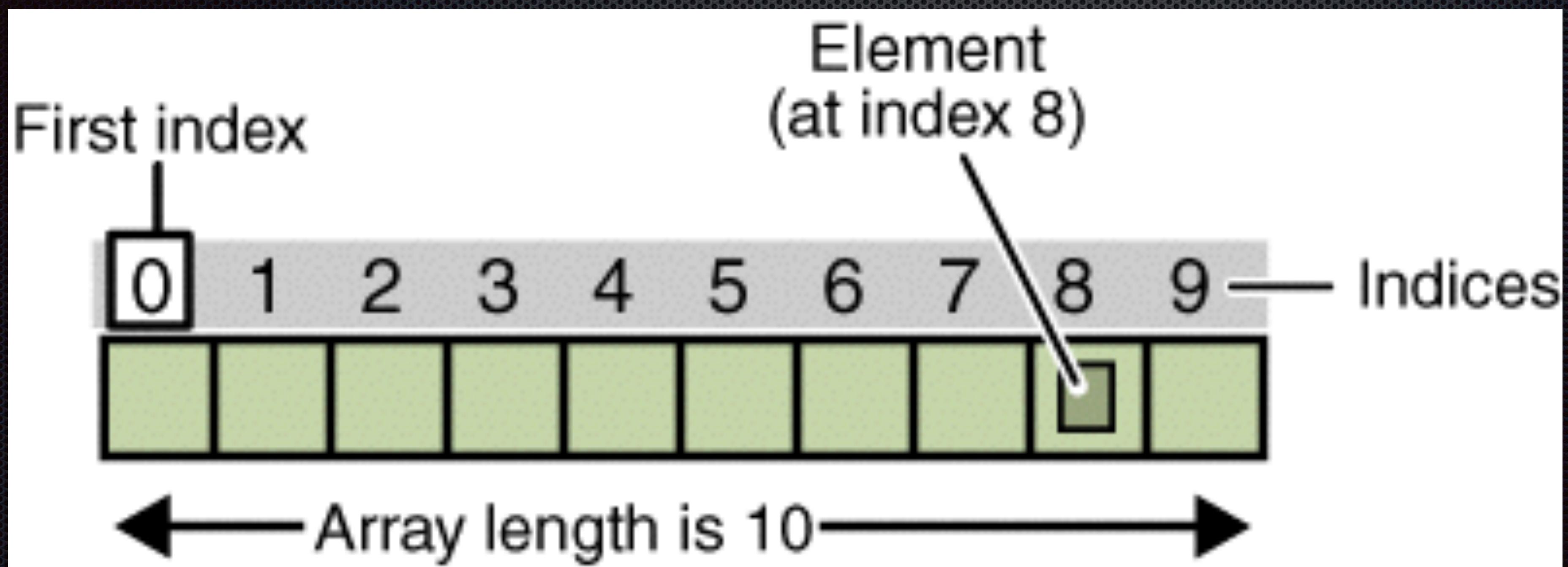


Warning!
Scala Ahead!

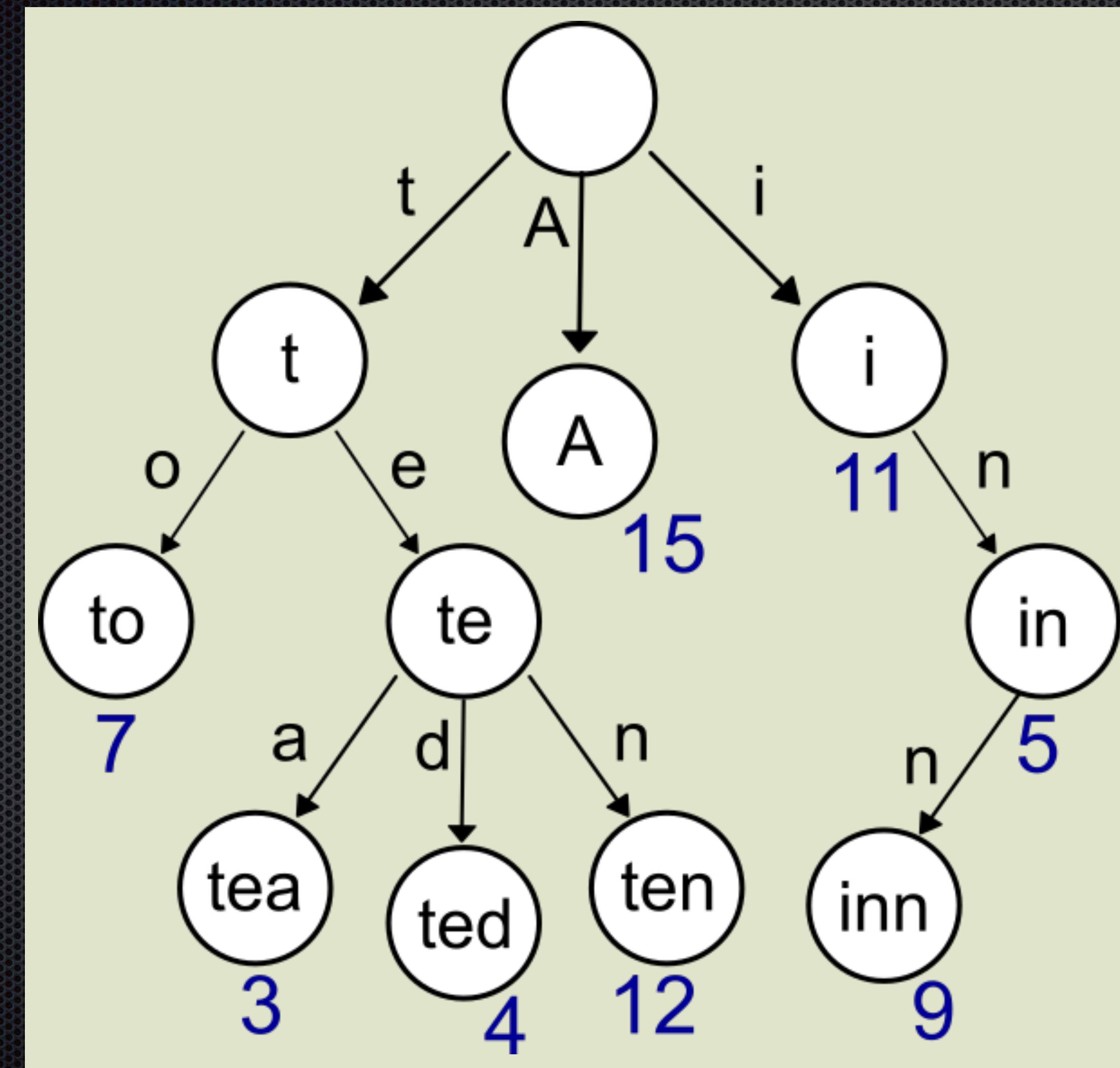


```
def balancedAdd(x: Int): Tree =  
  if (isEmpty) RedBranch(x)  
  else if (x < value) balance(isBlack, value, left.balancedAdd(x), right)  
  else if (x > value) balance(isBlack, value, left, right.balancedAdd(x))  
  else this  
  
def balance(b: Boolean, x: Int, left: Tree, right: Tree): Tree =  
  (b, left, right) match {  
    case (true, RedBranch(y, RedBranch(z, a, b), c), d) =>  
      BlackBranch(y, RedBranch(z, a, b), RedBranch(x, c, d))  
    case (true, a, RedBranch(y, b, RedBranch(z, c, d))) =>  
      BlackBranch(y, RedBranch(x, a, b), RedBranch(z, c, d))  
    case (true, RedBranch(z, a, RedBranch(y, b, c)), d) =>  
      BlackBranch(y, RedBranch(z, a, b), RedBranch(x, c, d))  
    case (true, a, RedBranch(z, RedBranch(y, b, c), d)) =>  
      BlackBranch(y, RedBranch(x, a, b), RedBranch(z, c, d))  
    case (true, _, _) => BlackBranch(x, left, right)  
    case (false, _, _) => RedBranch(x, left, right)  
  }
```

Vector



Trie

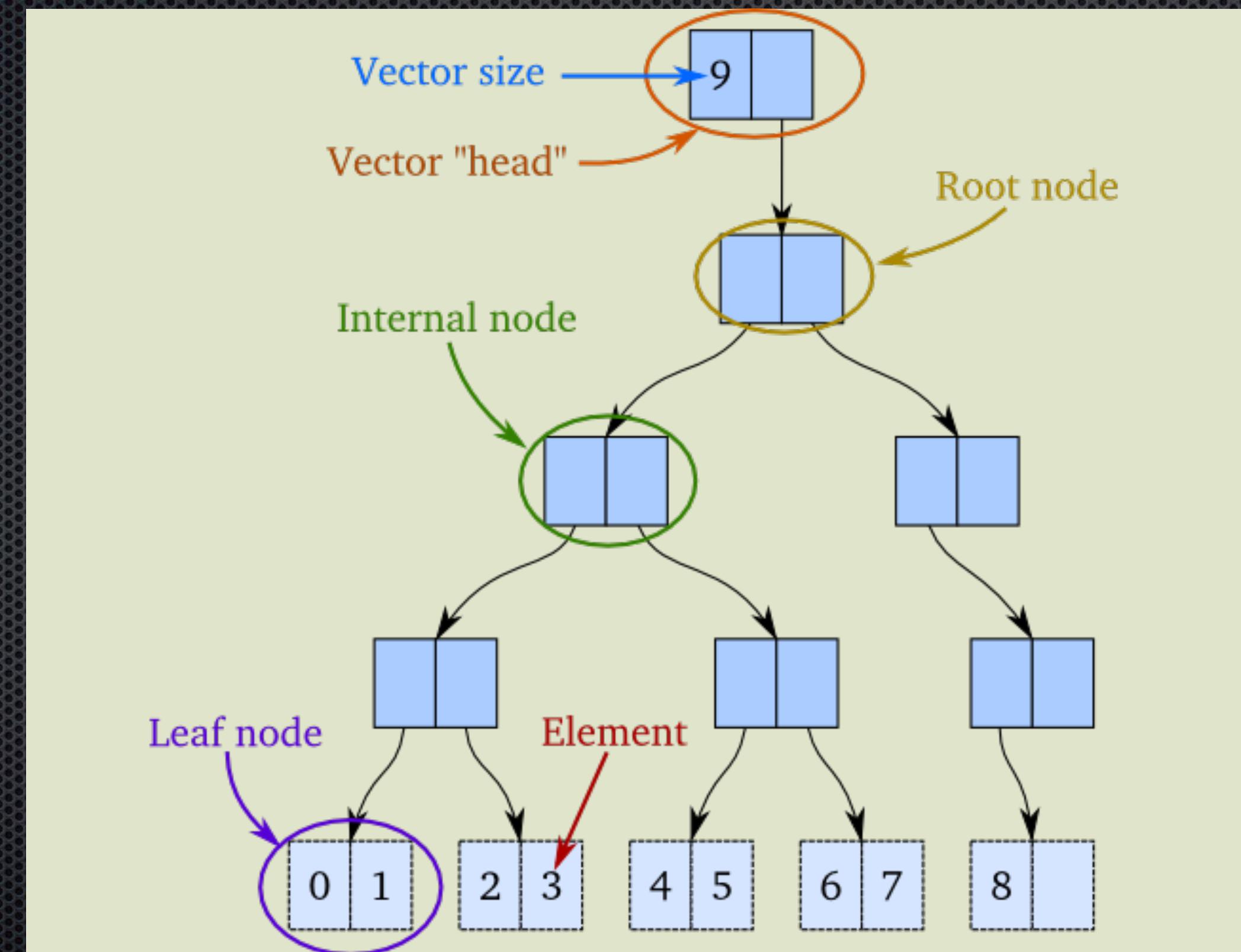


Vector

Persistent Bit-Partitioned Vector Trie

Vector

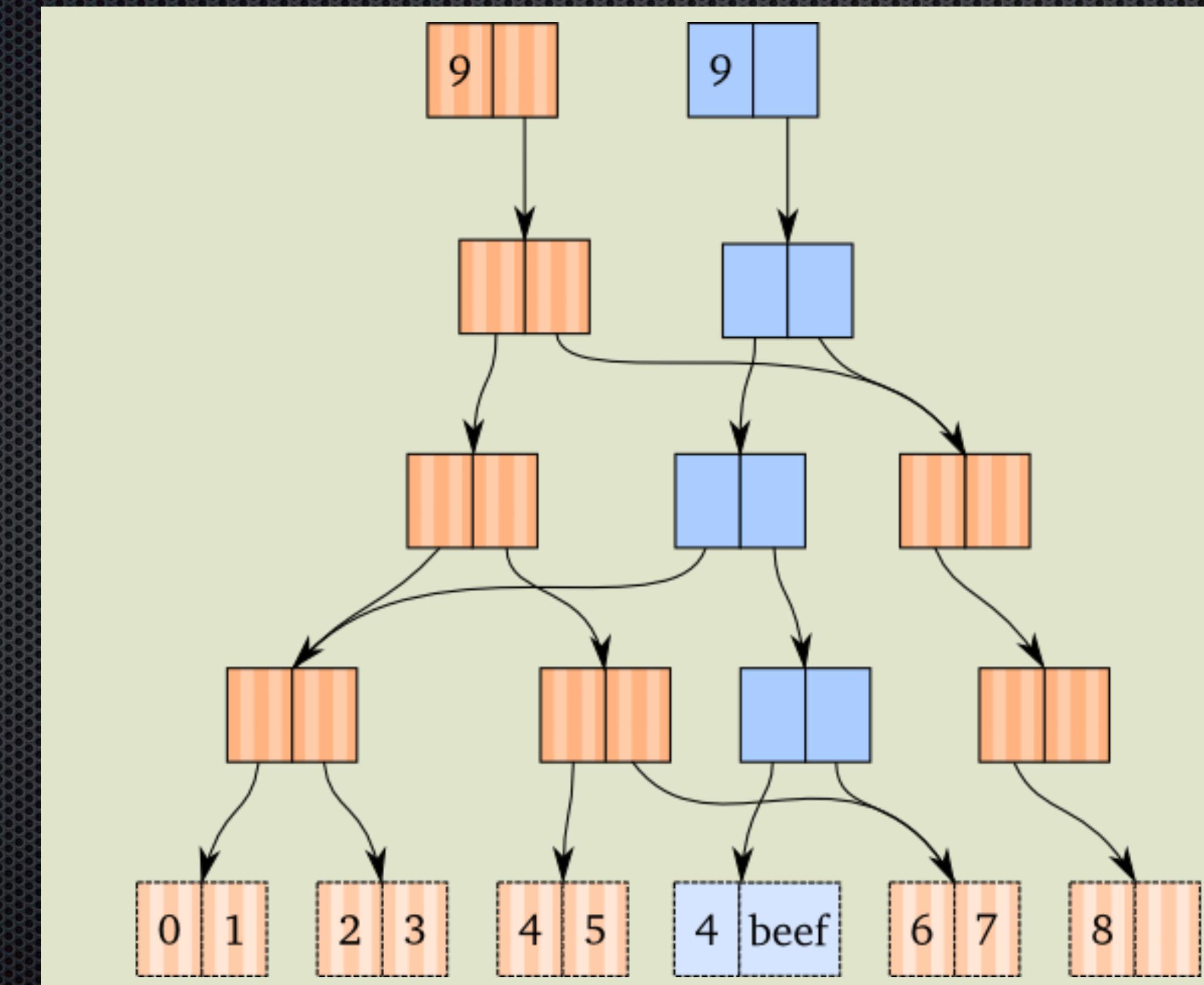
Persistent Bit-Partitioned Vector Trie



Update

$O(\log_2 n)$

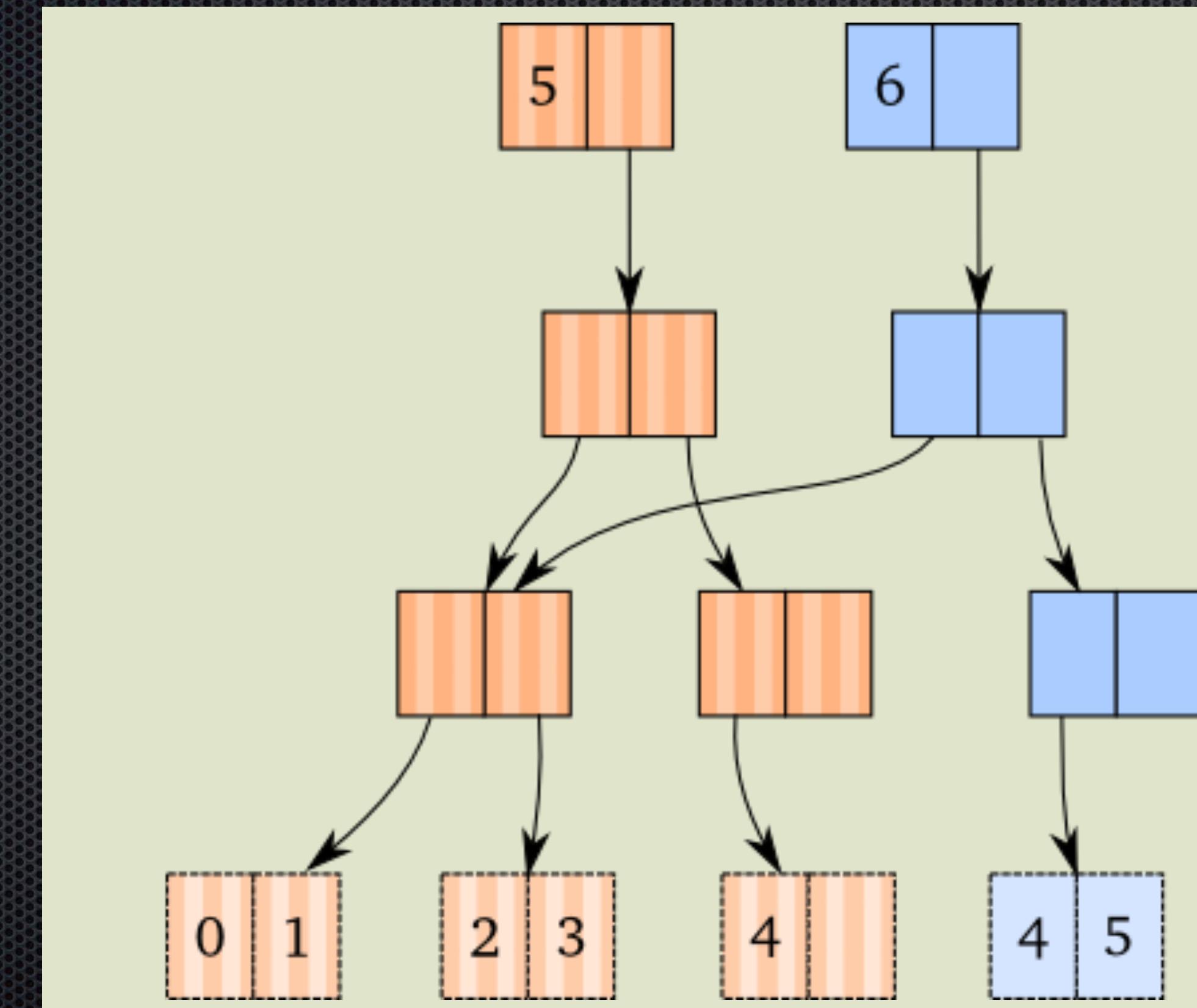
Vector Update



Push (1)

$O(\log_2 n)$

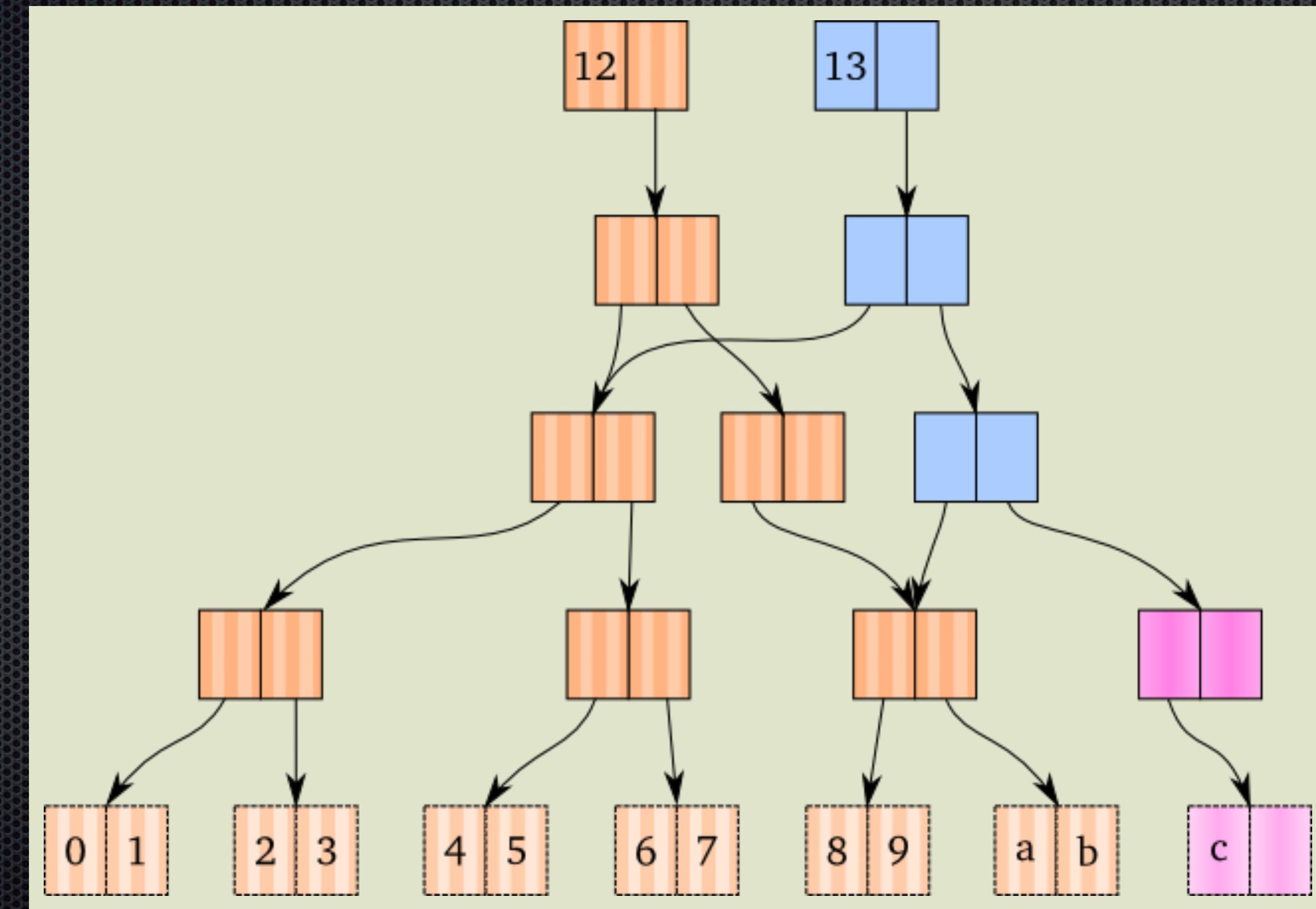
Vector



Push (2)

$O(\log_2 n)$

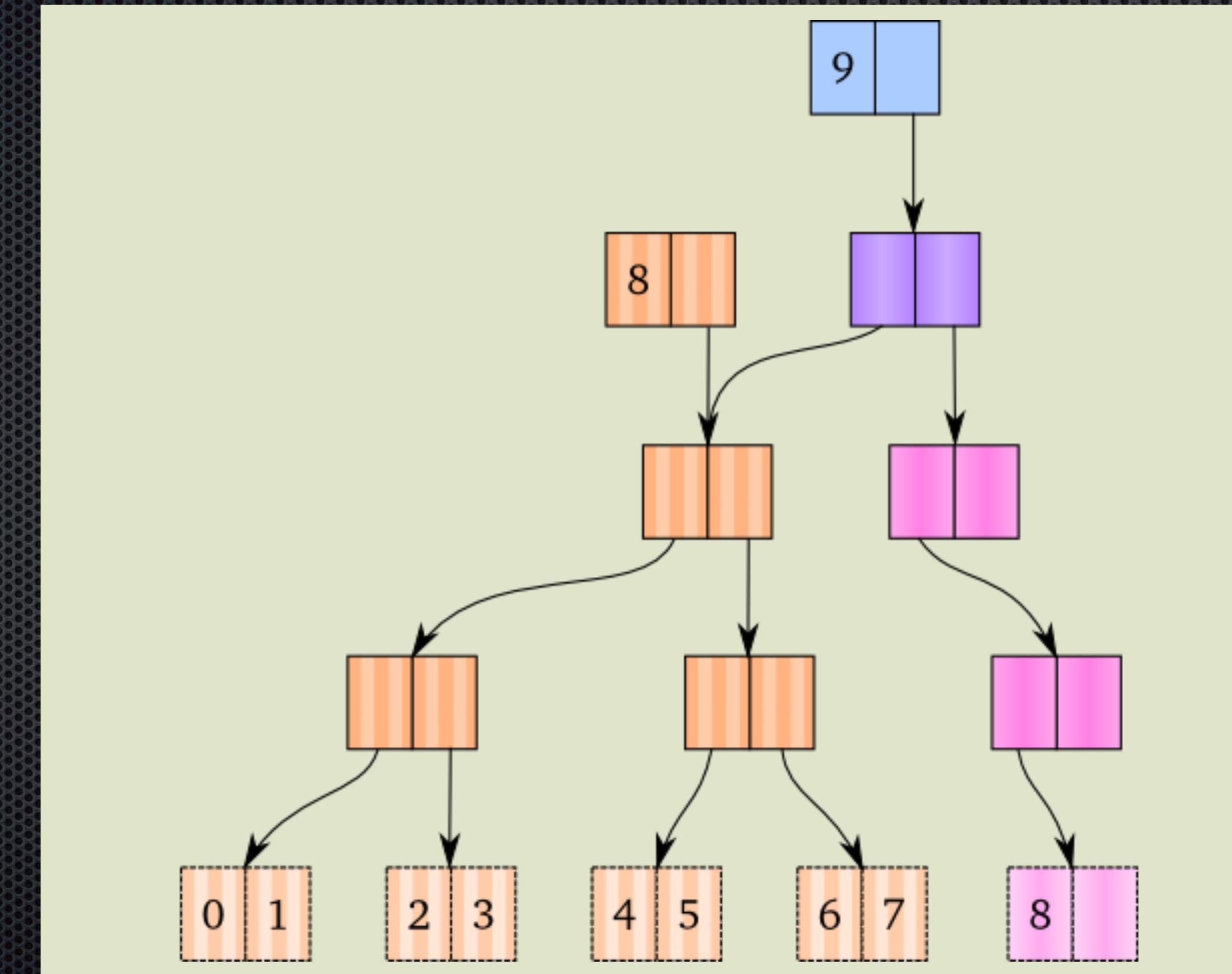
Vector



Push (3)

$O(\log_2 n)$

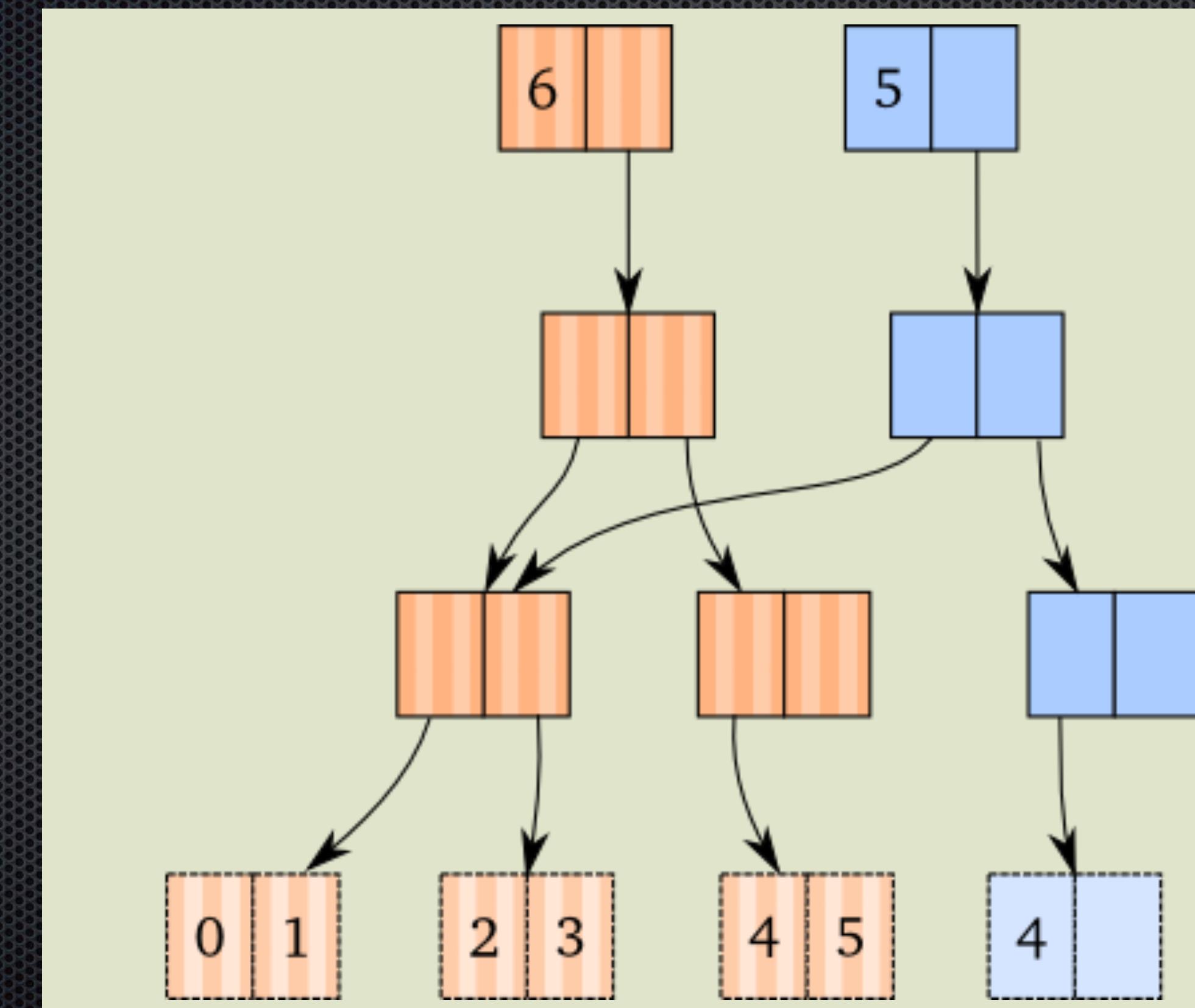
Vector



Remove (1)

$O(\log_2 n)$

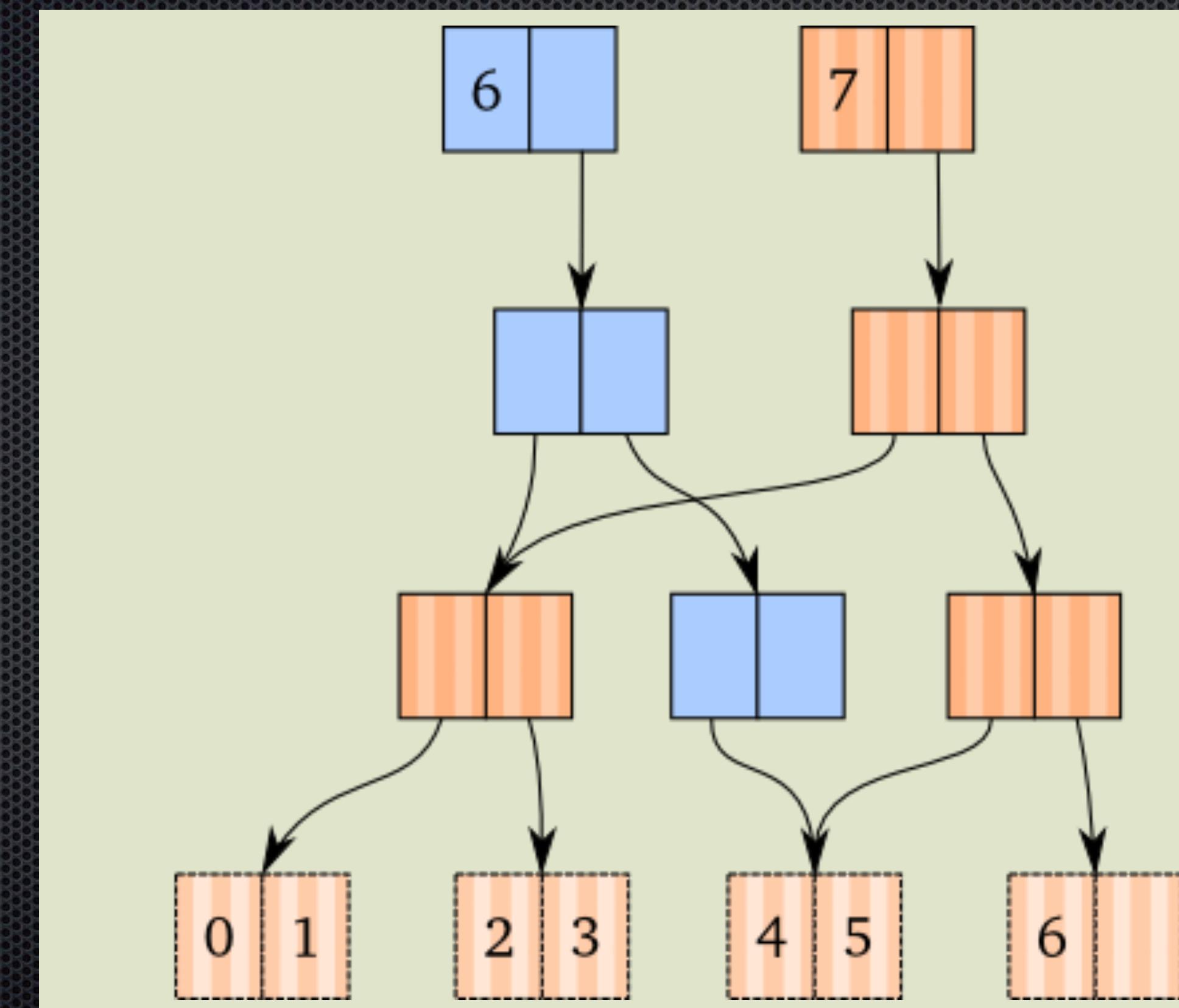
Vector



Remove (2)

$O(\log_2 n)$

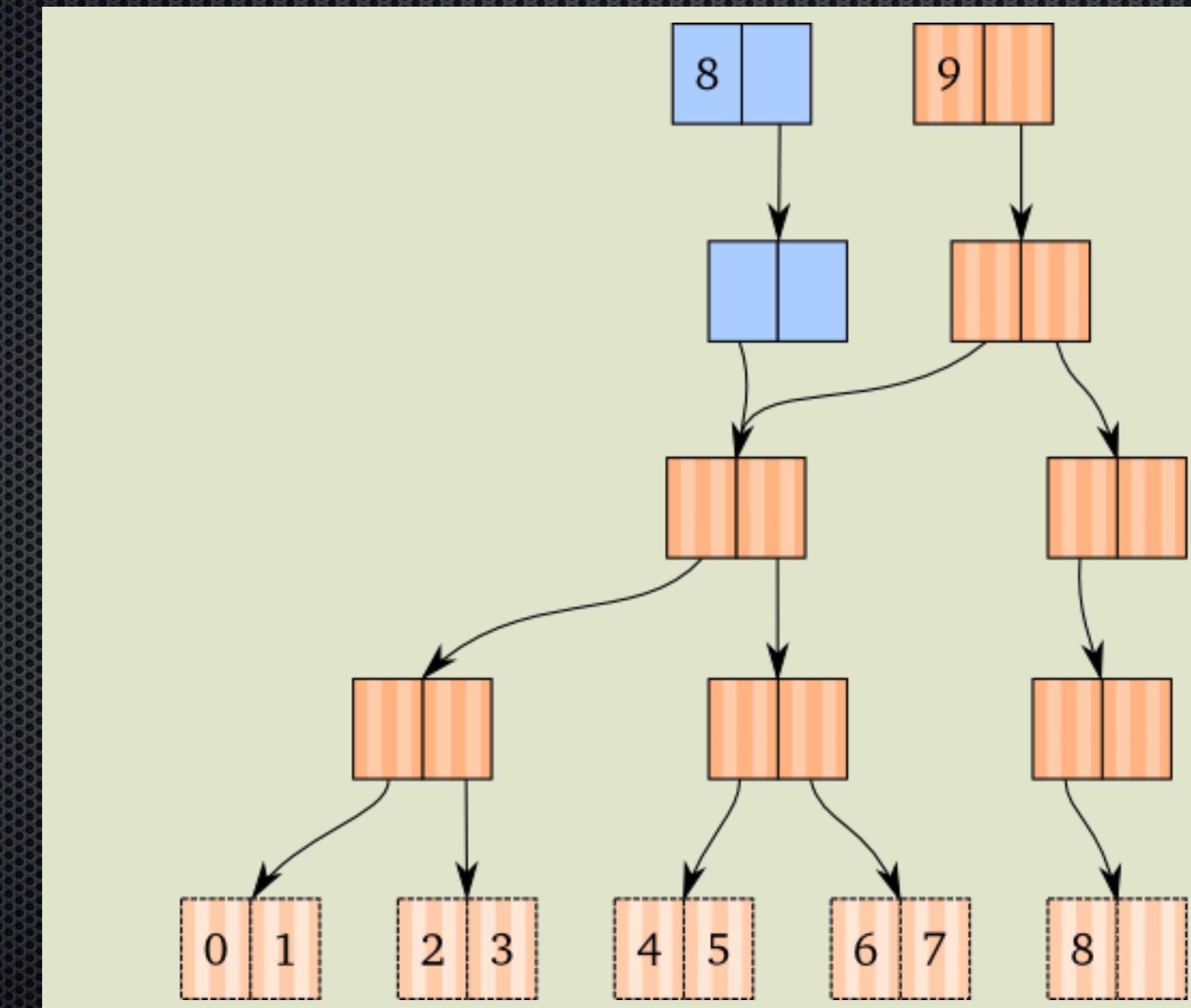
Vector



Remove (3)

$O(\log_2 n)$

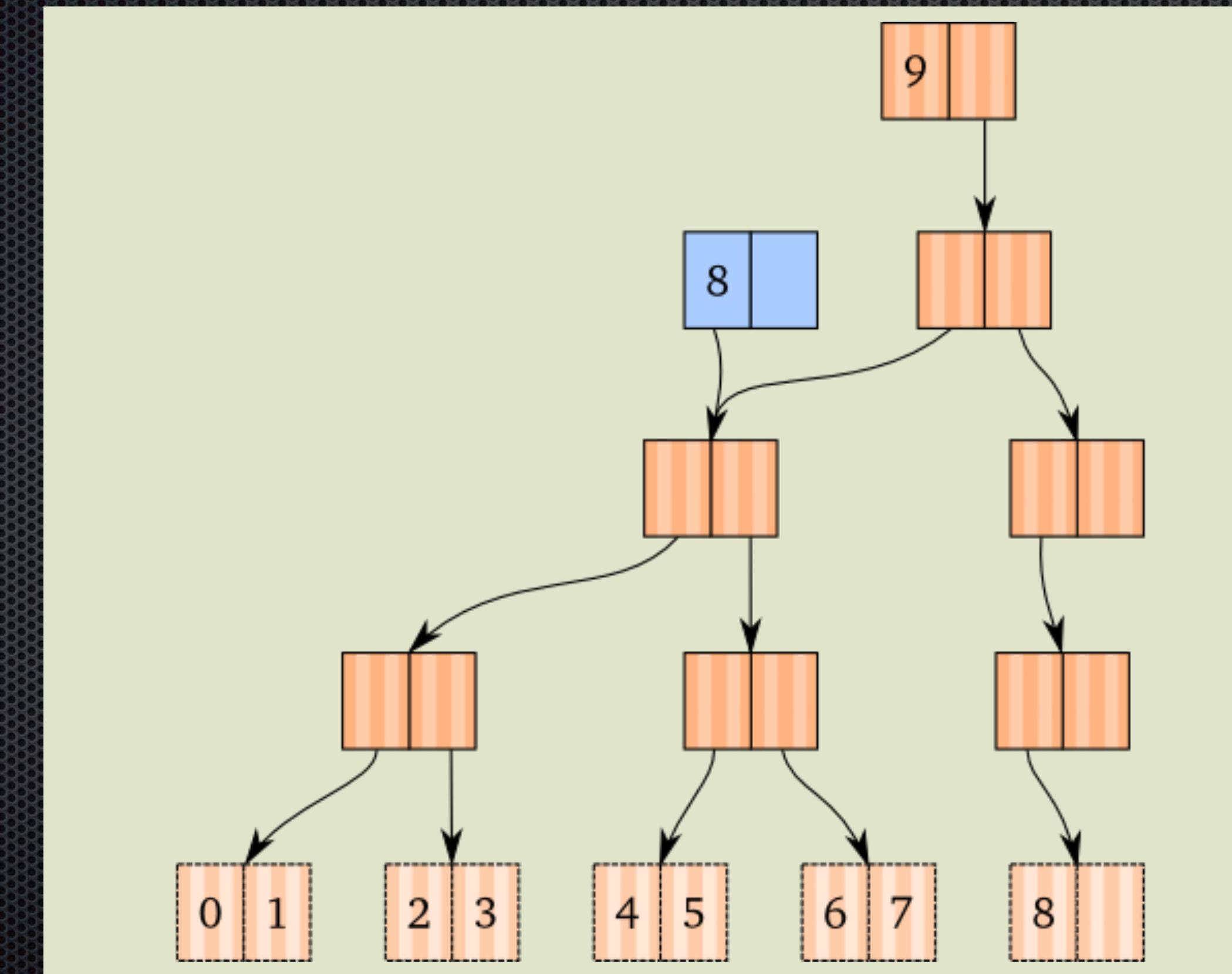
Vector



Remove (3.2)

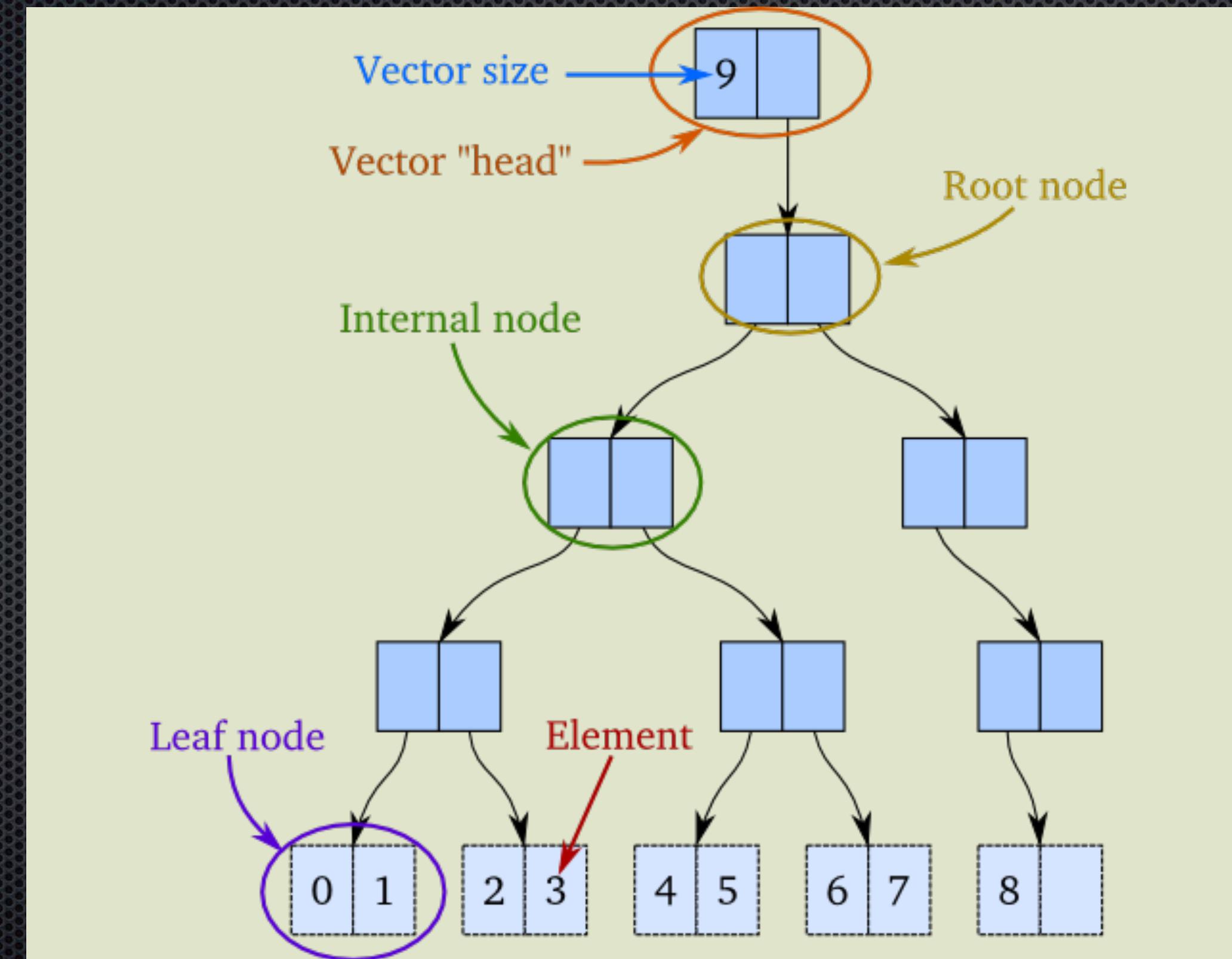
$O(\log_2 n)$

Vector



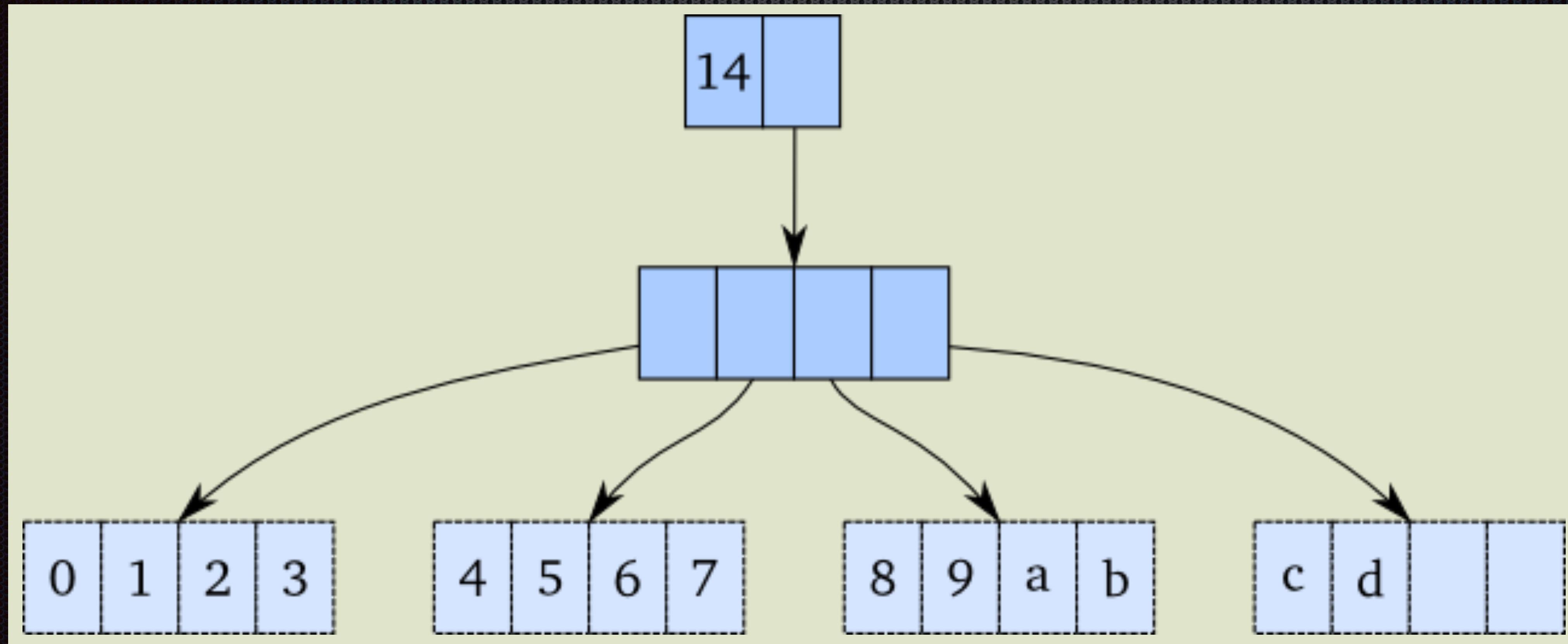
Vector

Branching Factor: 2



Vector

Branching Factor: 4



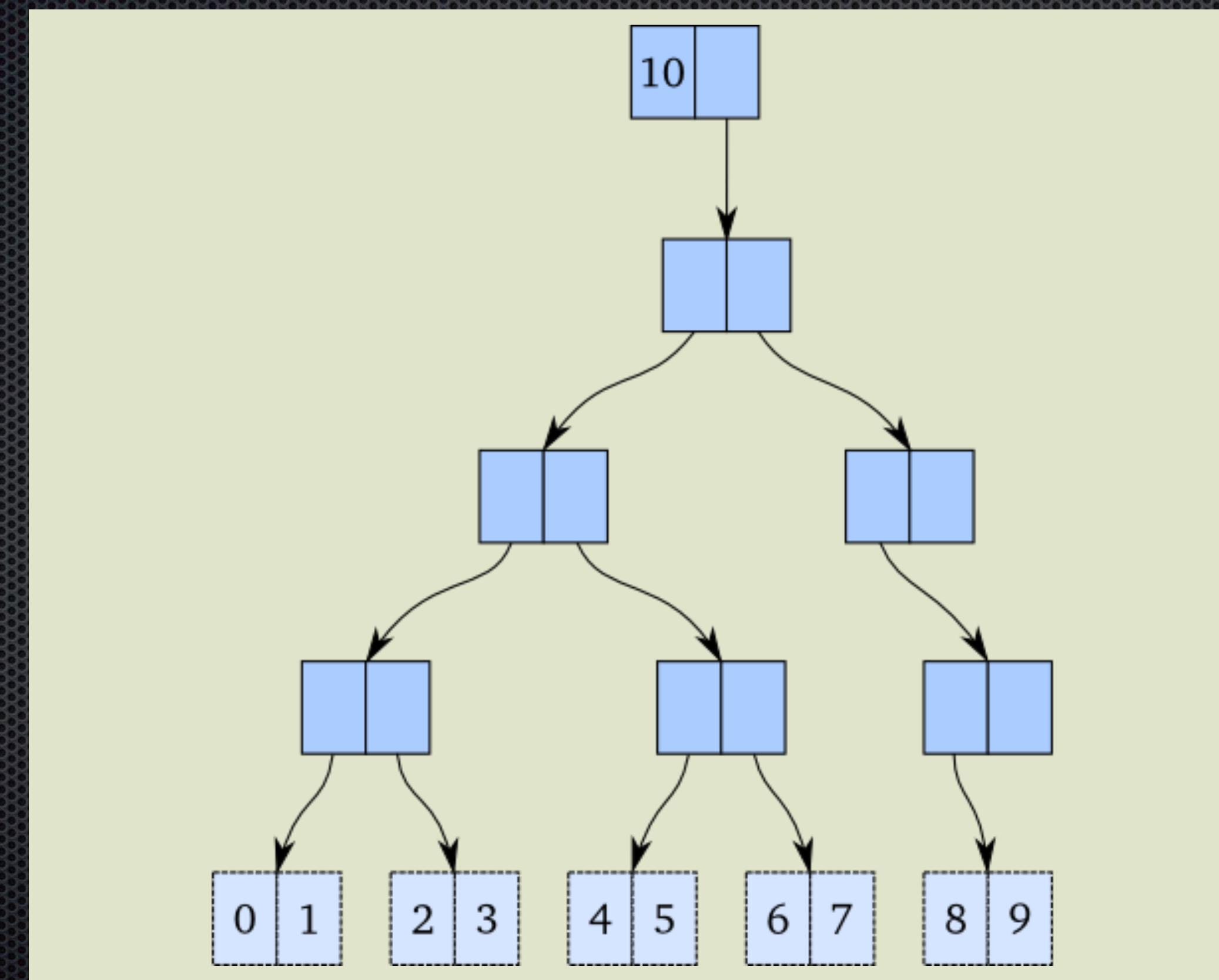
Vector

In reality branching factor is **32**

And all mentioned operations are **$O(\log_{32}n)$**

Which sometimes being promoted as **$O(1)$**

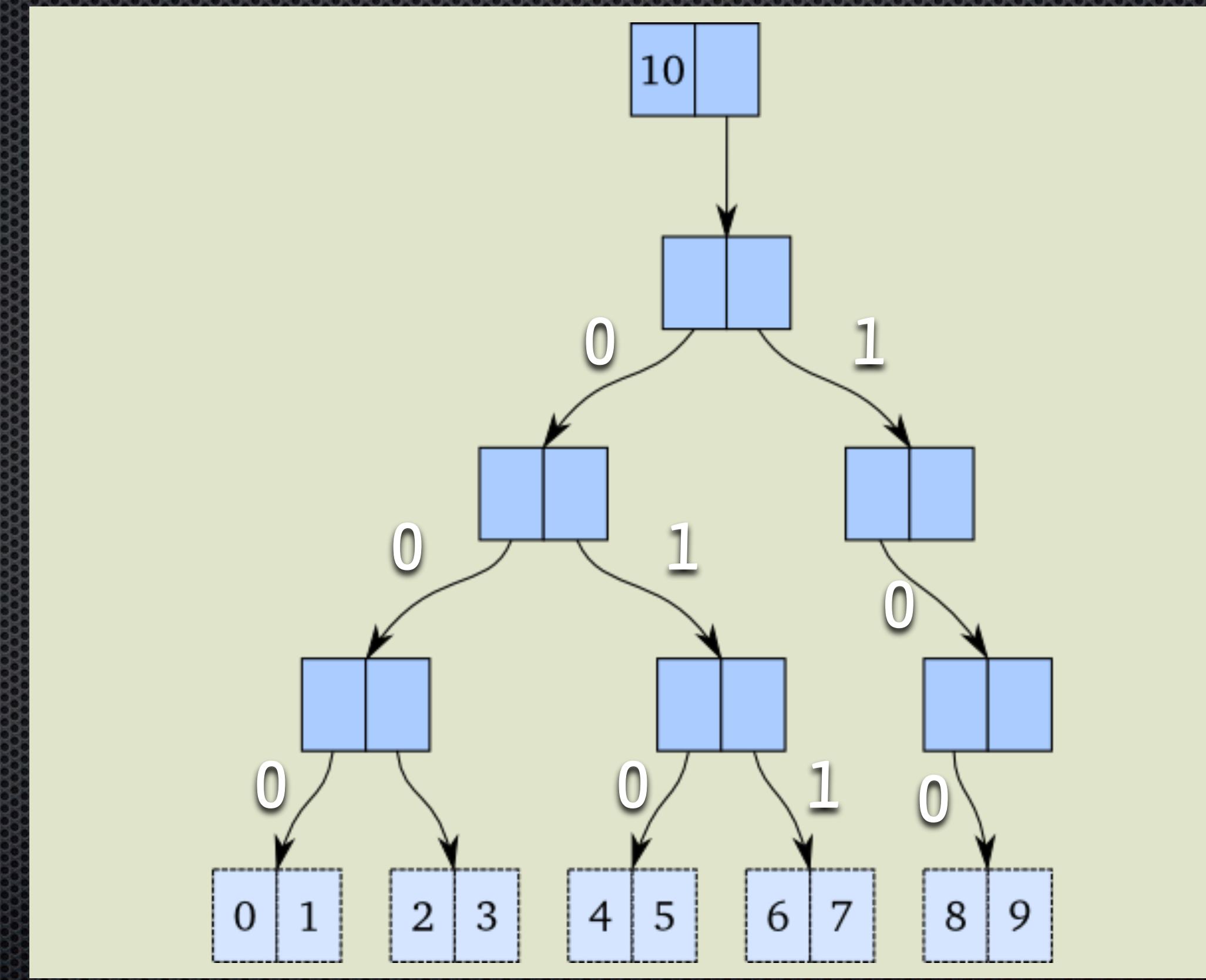
Vector Indexing



Vector

Indexing: bit-partitioned

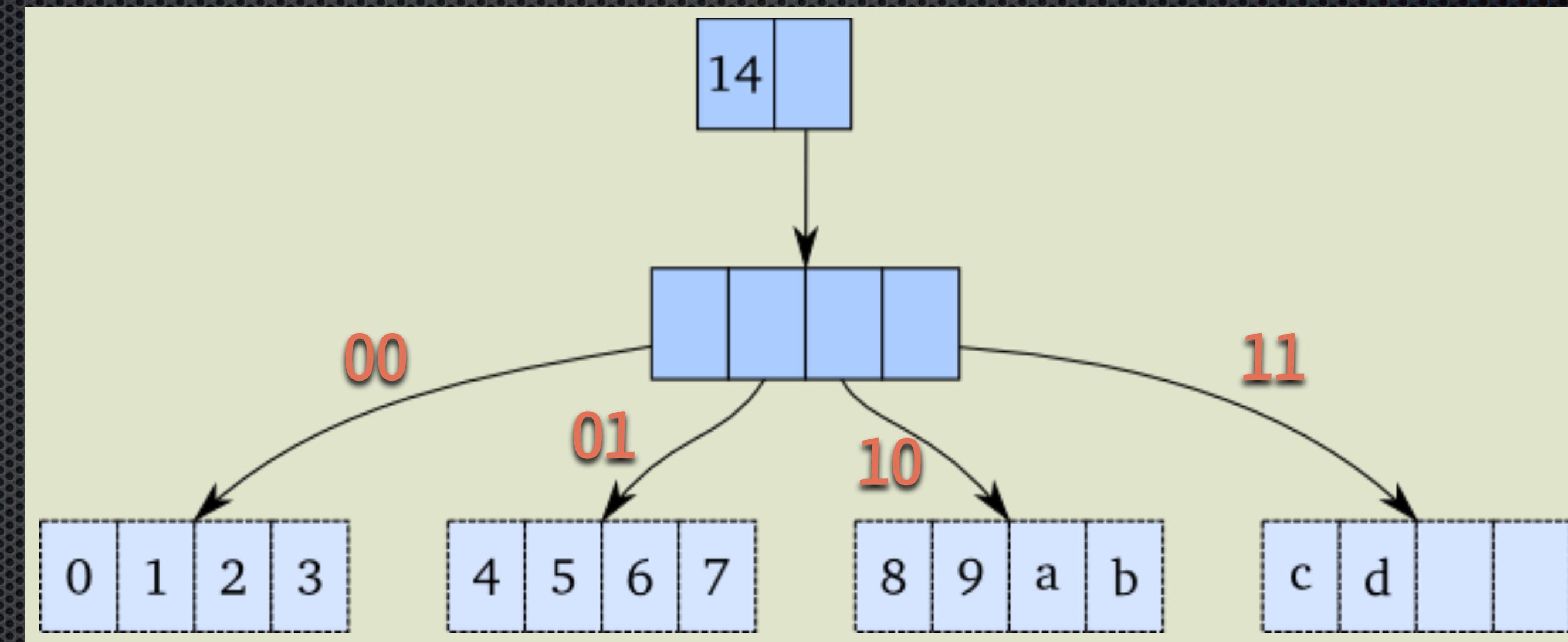
0 : 0000
1 : 0001
2 : 0010
3 : 0011
4 : 0100
5 : 0101
6 : 0110
7 : 0111
8 : 1000
9 : 1001



Vector

Indexing: bit-partitioned

0:	0000
1:	0001
2:	0010
3:	0011
4:	0100
5:	0101
6:	0110
7:	0111
8:	1000
9:	1001
a:	1010
b:	1011
c:	1100
d:	1101



Vector

Indexing: **bit-partitioned**
Branching factor: **32**

```
1 000 000 000: 0011 1011 1001 1010 1100 1010 0000 0000  
1 000 000 000: 00 11101 11001 10101 10010 10000 00000
```

Vector

1 000 000 000 000: 00 11101 11001 10101 10010 10000 00000

1 000 000 001

11101 [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

11001 [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

10101 [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

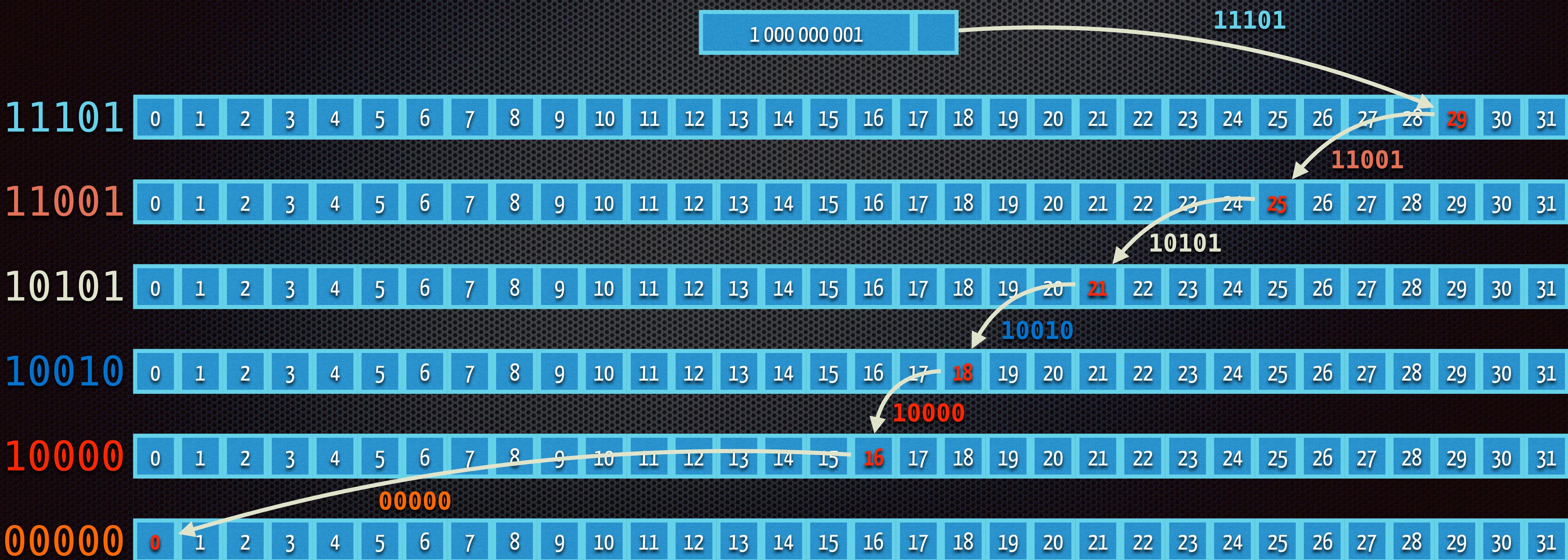
10010 [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

10000 [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

00000 [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

Vector

1 000 000 000 000: 00 11101 11001 10101 10010 10000 00000



HashMap

Map: (key → value) relationship

Hash: implementation detail

HashMap

Persistent Hash Array Mapped Trie

HashMap

Fast And Space Efficient Trie Searches

Phil Bagwell

Searching and traversing m-way trees using tries is a well known and broadly used technique. Three algorithms are presented, two have constant insert, search or delete cost, are faster than Hash Trees and can be searched twice as quickly as Ternary Search Trees (TST). The third has a $\lg N$ byte compare cost, like a TST, but is faster. All require 60% less memory space per node than TST and, unlike Hash Trees, are smoothly extensible and support sorted order functions. The new algorithms defining Array Compacted Trees (ACT), Array Mapped Trees (AMT), Unary Search Trees (UST) and their variants are discussed. These new search trees are applicable in many diverse areas such as high performance IP routers, symbol tables, lexicon scanners and FSA state tables.

Categories and Subject Descriptors: H.4.m [Information Systems]: Miscellaneous

General Terms: Searching, Database

Additional Key Words and Phrases: Trie, Tries, Search, Trees, TST, ACT, AMT, UST, Symbol Table

Ideal Hash Trees

Phil Bagwell

Hash Trees with nearly ideal characteristics are described. These Hash Trees require no initial root hash table yet are faster and use significantly less space than chained or double hash trees. Insert, search and delete times are small and constant, independent of key set size, operations are O(1). Small worst-case times for insert, search and removal operations can be guaranteed and misses cost less than successful searches. Array Mapped Tries(AMT), first described in Fast and Space Efficient Trie Searches, Bagwell [2000], form the underlying data structure. The concept is then applied to external disk or distributed storage to obtain an algorithm that achieves single access searches, close to single access inserts and greater than 80 percent disk block load factors. Comparisons are made with Linear Hashing, Litwin, Neimat, and Schneider [1993] and B-Trees, R.Bayer and E.M.McCreight [1972]. In addition two further applications of AMTs are briefly described, namely, Class/Selector dispatch tables and IP Routing tables. Each of the algorithms has a performance and space usage that is comparable to contemporary implementations but simpler.

Categories and Subject Descriptors: H.4.m [Information Systems]: Miscellaneous

General Terms: Hashing, Hash Tables, Row Displacement, Searching, Database, Routing, Routers

HashMap

```
17
18  /*
19   * A persistent rendition of Phil Bagwell's Hash Array Mapped Trie
20
21   * Uses path copying for persistence
22   * HashCollision leaves vs. extended hashing
23   * Node polymorphism vs. conditionals
24   * No sub-tree pools or root-resizing
25   * Any errors are my own
26   */
27
```

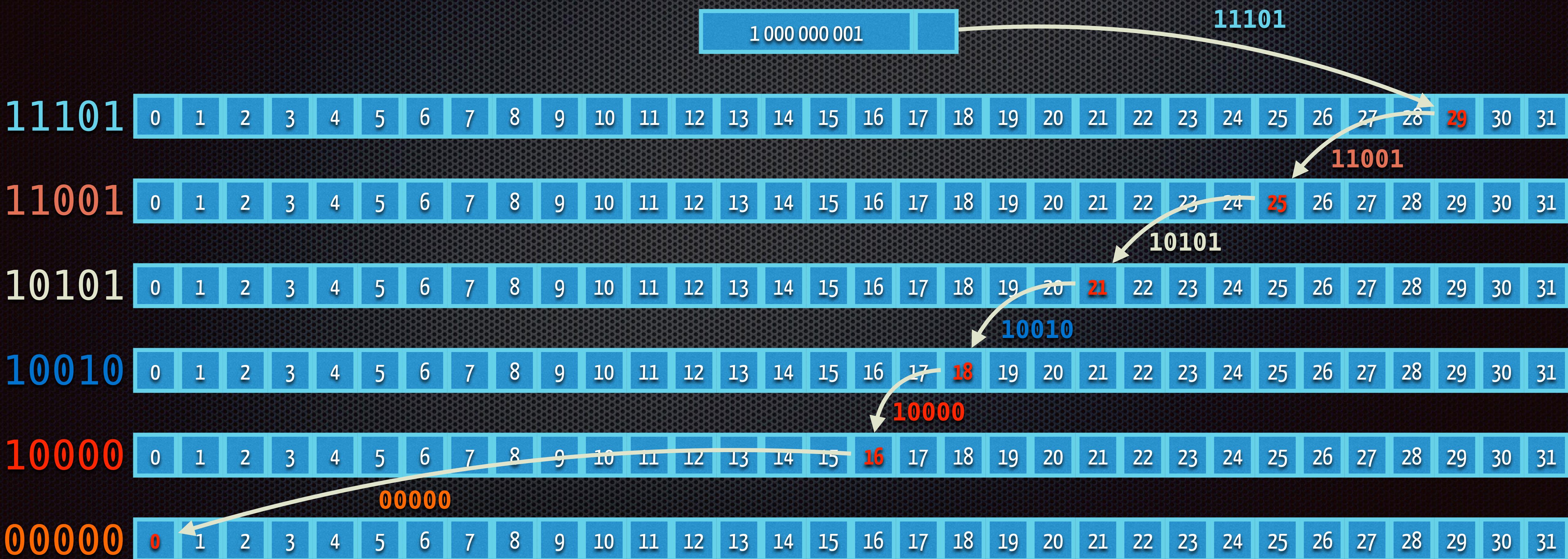
<https://github.com/clojure/clojure/blob/master/src/jvm/clojure/lang/PersistentHashMap.java#L18-L26>

HashMap

```
int hashOfKey = hash(key)  
use trie (remember vector?) to find key and value
```

Vector

1 000 000 000 000: 00 11101 11001 10101 10010 10000 00000



HashMap

We can't allocate that much memory from the beginning!

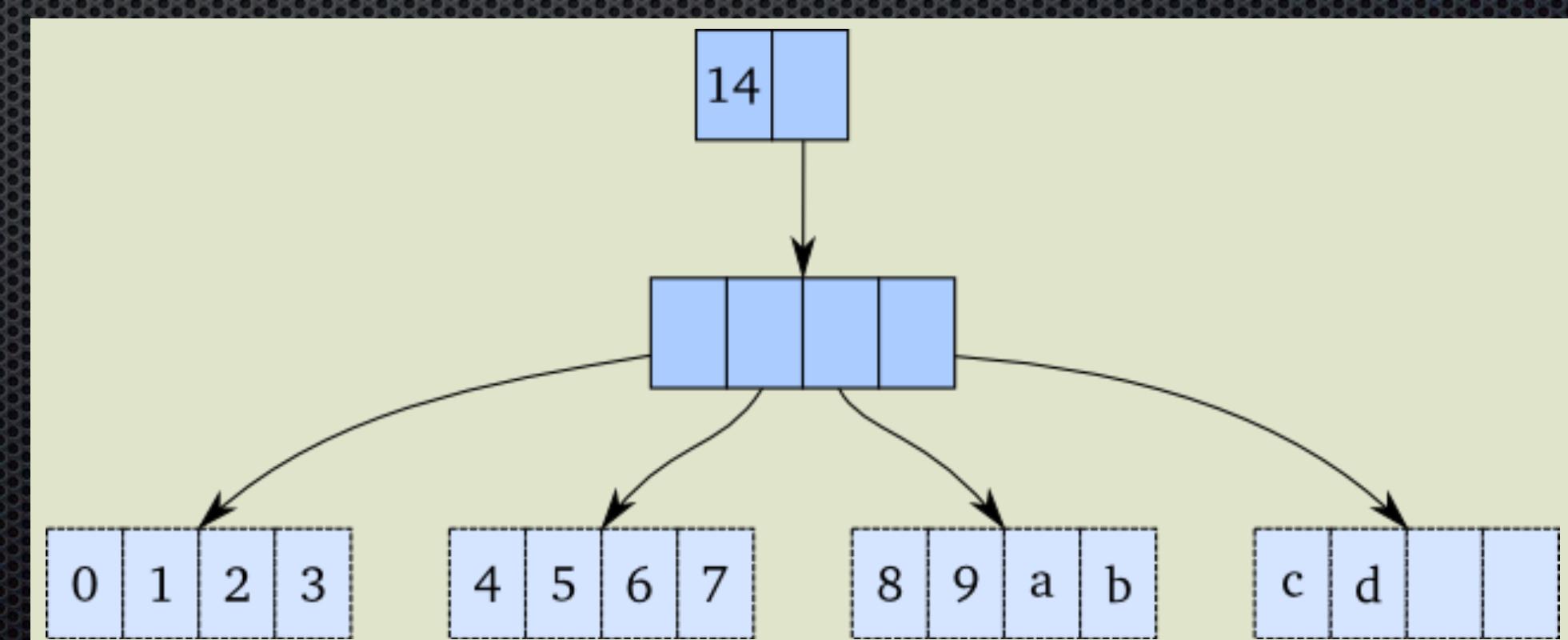
Solution: built trie incrementally!

Path copying still works as expected

HashMap

BitmapIndexedNode

```
class BitmapIndexedNode implements INode {  
    int bitmap;  
    Object[] array;  
  
    public INode assoc(int shift, int hash, Object key, Object val) {}  
    public INode without(int shift, int hash, Object key) {}  
    public IMapEntry find(int shift, int hash, Object key) {}  
}
```

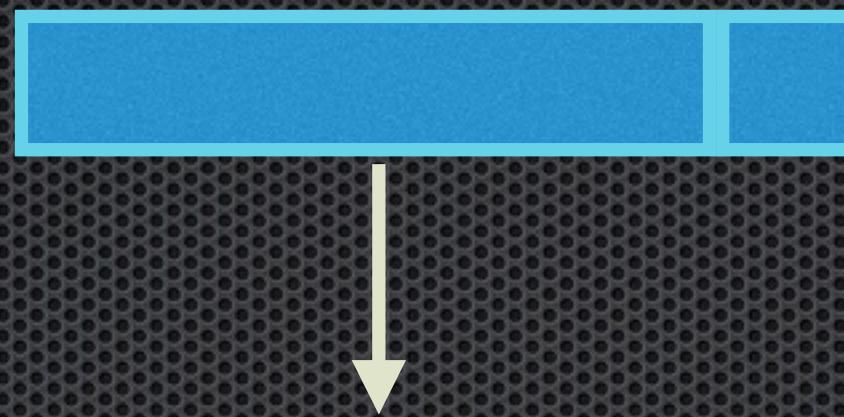


BitmapIndexedNode

add(k1 , v1)

hash(k1)==1000000003

1 000 000 003: 00 11101 11001 10101 10010 10000 00011



bitmap: 0000 0000 0000 0000 0000 0000 0000 1000

array:



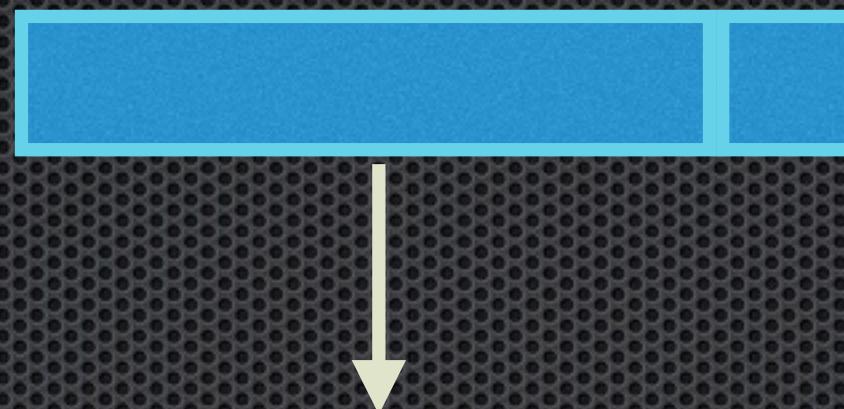
00011

BitmapIndexedNode

add(k2 , v2)

hash(k2) == 1000000000

1 000 000 000: 00 11101 11001 10101 10010 10000 00000



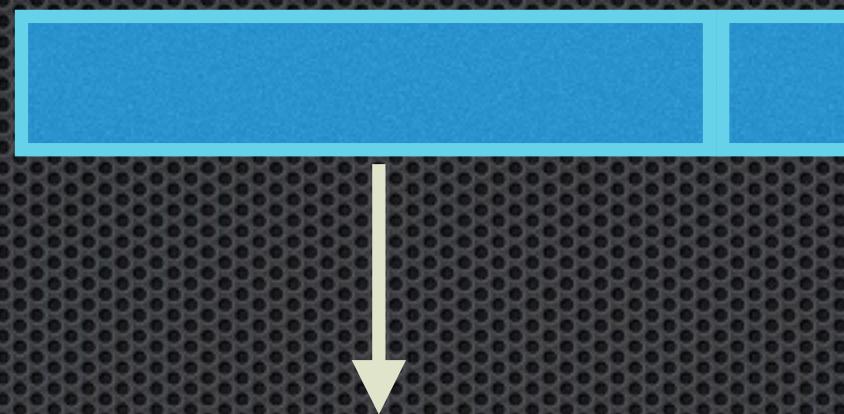
bitmap: 0000 0000 0000 0000 0000 0000 0000 0000 1001
array: k2 v2 k1 v1 00000

BitmapIndexedNode

add(k3 , v3)

hash(k3)==1000000005

1 000 000 005: 00 11101 11001 10101 10010 10000 00101



bitmap: 0000 0000 0000 0000 0000 0000 0010 1001
array: 00101

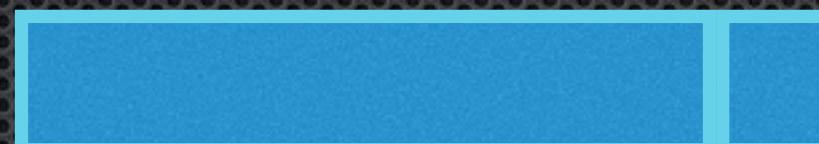


BitmapIndexedNode

add(k4 , v4)

hash(k4)==1000000032

1 000 000 032: 00 11101 11001 10101 10010 10001 00000



bitmap: 0000 0000 0000 0000 0000 0000 0000 0010 1001
array: [ref k1 v1 k3 v3]

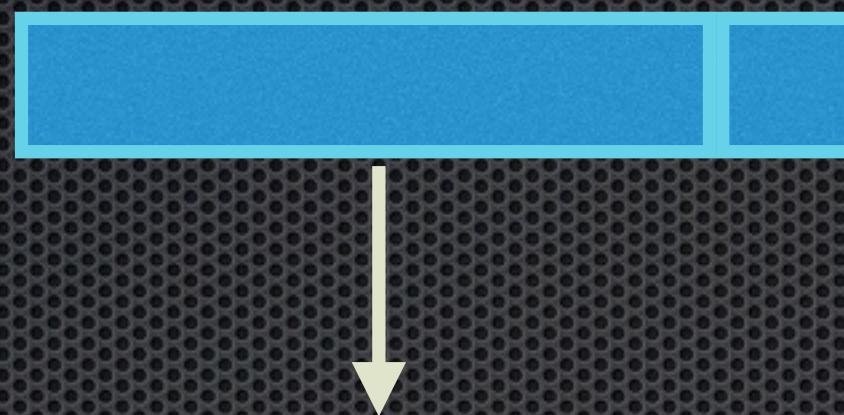


bitmap: 0000 0000 0000 0011 0000 0000 0000 0000 0000
array: [k2 v2 k4 v4]

10001
10000

HashMap

Optimisations



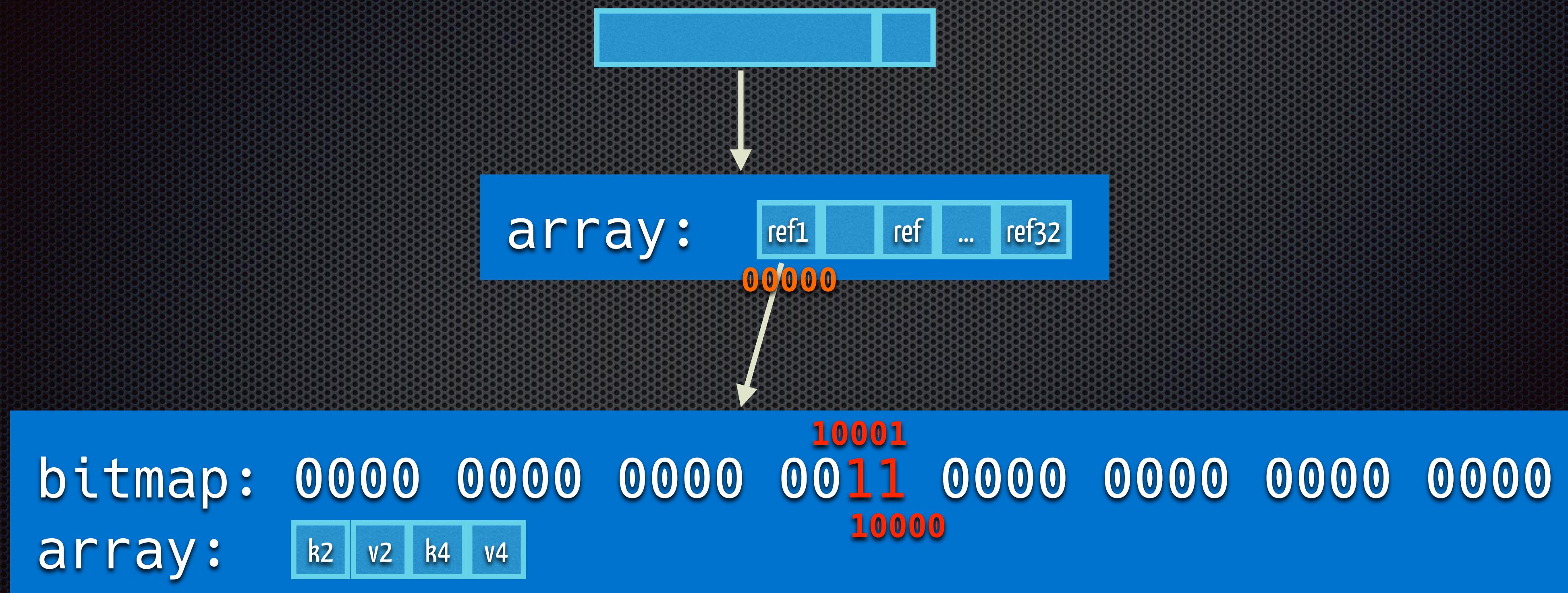
bitmap: 0000 0000 0000 0001 1111 1111 1111 1111

array: [k1 v1 k2 v2 k3 v3 ... k17 v17]

When BitmapIndexedNode's size is greater than 16
(and array size is greater than 32),
then migrate to ArrayNode

HashMap

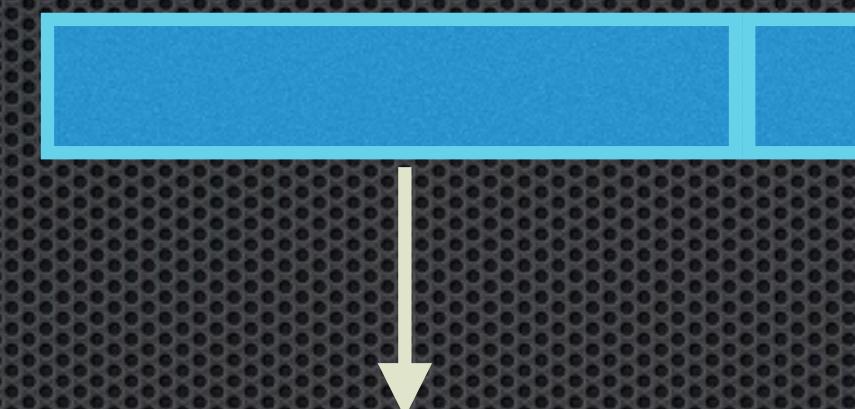
Optimisations: ArrayNode



Optimisations: HashCollisionNode

**add(k4 , v4)
hash(k4)==1000000000**

1 000 000 000: 00 11101 11001 10101 10010 10000 00000



bitmap: 0000 0000 0000 0000 0000 0000 0000 0010 1001
array: [ref k1 v1 k3 v3]

hash: 1000000000 (10000)
array: [k2 v2 k4 v4]

Set

TreeSet -> TreeMap

HashSet -> HashMap

value == key



1959	Biranda, Fredkin	Trie
1960	Windley, Booth, Colin, Hibbard	Binary Search Trees
1962	Adelson-Velsky, Landis	AVL Trees
1978	Guibas, Sedgwick	Red Black Trees
1985	Sleator, Tarjan	Splay Trees
1996	Okasaki	Purely Functional Data Structures
1998	Sedgwick	Ternary Search Trees
2000	Phil Bagwell	AMT
2001	Phil Bagwell	HAMT
2007	Rich Hickey	Clojure!

Frontend

Since immutable data never changes, subscribing to changes throughout the model is a dead-end and new data can only ever be passed from above.

This model of data flow aligns well with the architecture of React and especially well with an application designed using the ideas of Flux.

Frontend

<https://facebook.github.io/immutable-js/>

<http://redux.js.org/>

Thanks!

@fxposter