Vectors

… backed by Radix-Balanced trees

Chaitanya Koparkar

It’s fast!

Op	Time complexity
lookup	$O(\log_m n)$
update	$O(m * \log_m n)$
cons/snoc	$O(m * \log_m n)$
concat	$O(n)$
empty?	$O(1)$
toList	$O(n)$
fromList	$O(n)$
length	$O(1)$

Branching factor (m) = 4

It’s fast!

Op	Time complexity
lookup	$O(\log_m n)$
update	$O(m * \log_m n)$
cons/snoc	$O(m * \log_m n)$
concat	$O(n)$
empty?	$O(1)$
toList	$O(n)$
fromList	$O(n)$
length	$O(1)$

Branching factor (m) = 4

RRB Vector: A Practical General Purpose Immutable Sequence

Nicolas Stucki[†] Tiark Rompf [‡] Vlad Ureche[†] Phil Bagwell [*]

[†]EPFL, Switzerland: {first.last}@epfl.ch
[‡]Purdue University, USA: {first}@purdue.edu

Abstract

State-of-the-art immutable collections have wildly differing performance characteristics across their operations, often forcing programmers to choose different collection implementations for each task. Thus, changes to the program can invalidate the choice of collections, making code evolution costly. It would be desirable to have a collection that performs well for a broad range of operations.

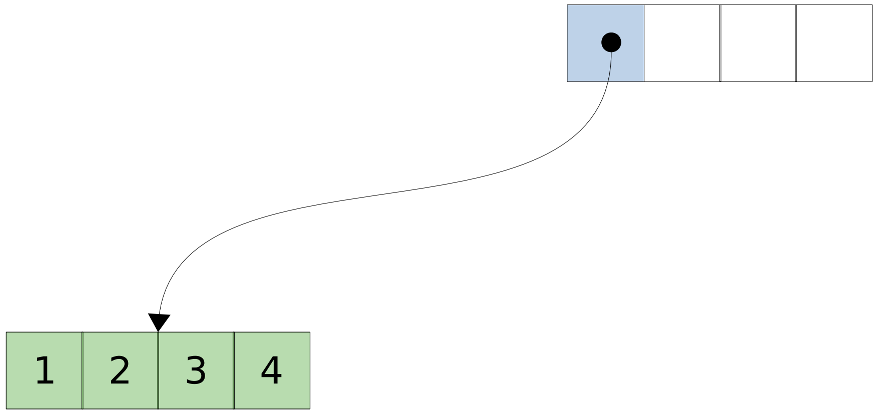
To this end, we present the `RRB-Vector`, an immutable sequence collection that offers good performance across a large number of sequential and parallel operations. The underlying innovations are: (1) the Relaxed-Radix-Balanced (RRB) tree structure, which allows efficient structural reorganization, and (2) an optimization that exploits spatio-temporal locality on the RRB data structure in order to offset the cost of traversing the tree.

In our benchmarks, the `RRB-Vector` speedup for parallel operations is lower bounded by $7\times$ when executing on 4 CPUs of 8 cores

Bulk operations on immutable collections lend themselves to implicit parallelization. This allows the execution to proceed either sequentially, by traversing the collection one element at a time, or in parallel, by delegating parts of the collection to be traversed in different execution contexts and combining the intermediate results. Therefore, the bulk operations allow programs to scale to multiple cores without explicit coordination, thus lowering the burden on programmers.

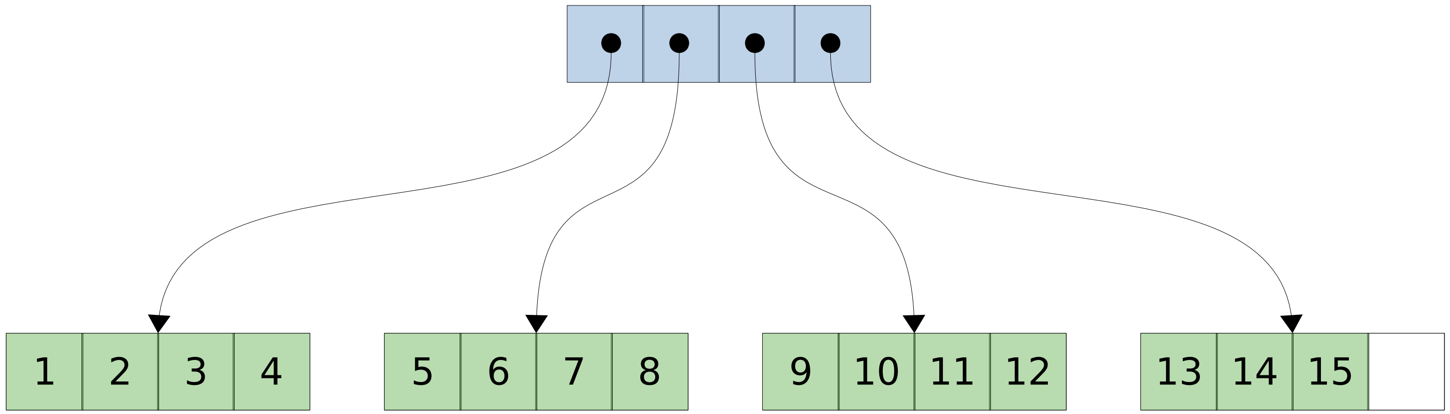
Most state-of-the-art collection implementations are tailored to some specific operations, which are executed very fast, at the expense of the others, which are slow. For example, the ubiquitous `Cons` list is extremely efficient for prepending elements and accessing the head of the list, performing both operations in $\mathcal{O}(1)$ time. However, it has a linear $\mathcal{O}(n)$ cost for reading and updating random elements. And although sequential scanning is efficient, requiring $\mathcal{O}(1)$ time per element, it cannot benefit from parallel execution,

RB-Tree



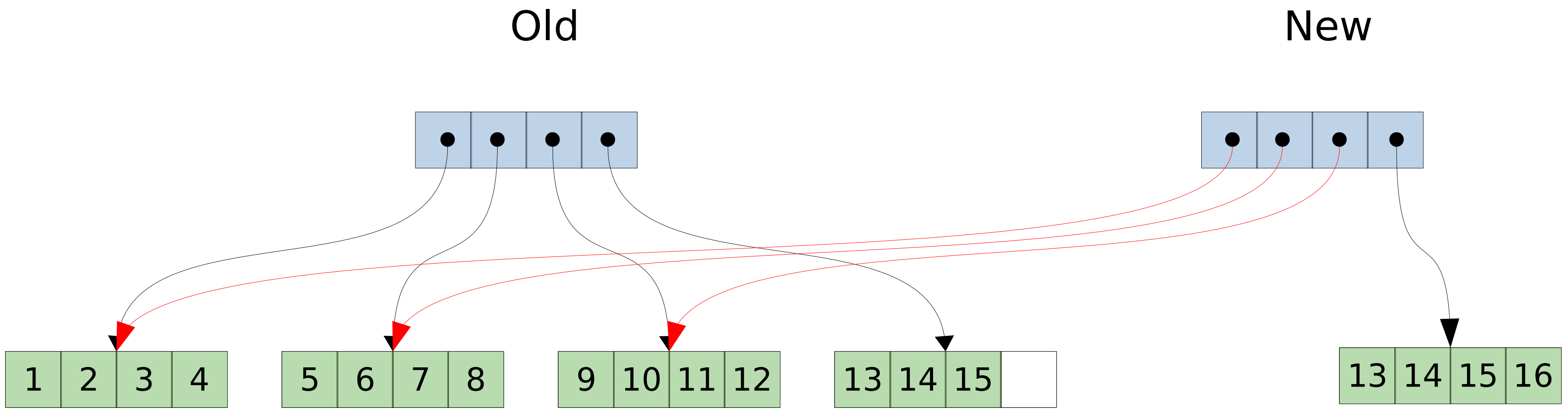
Branching factor (m) = 4

RB-Tree



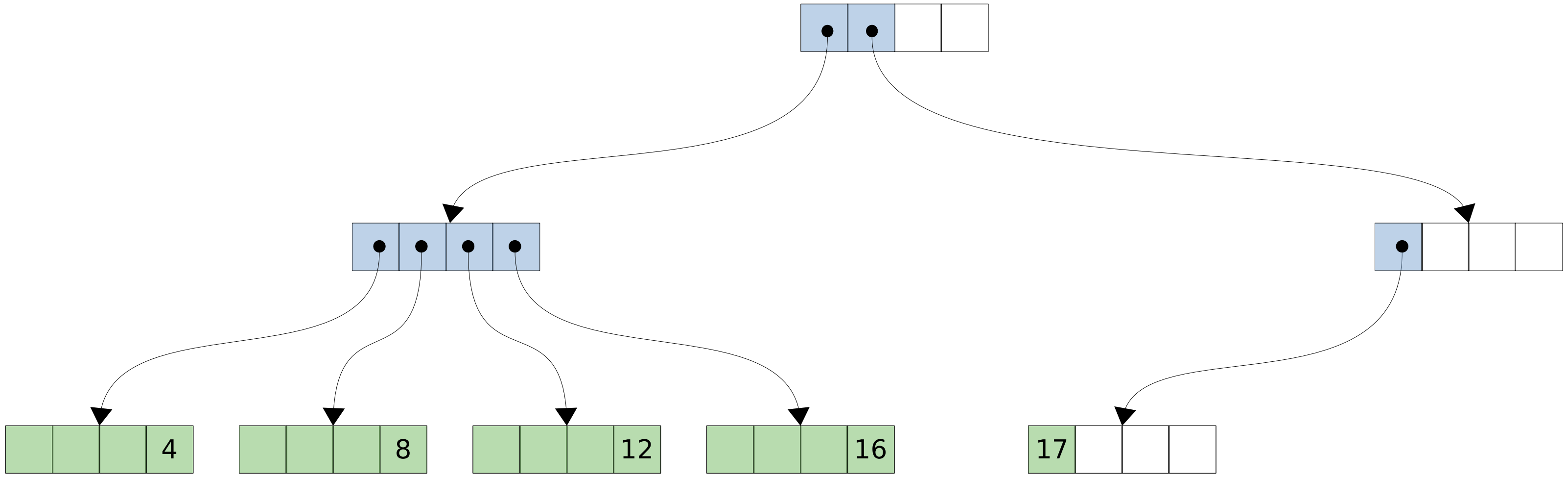
Branching factor (m) = 4

RB-Tree



Branching factor (m) = 4

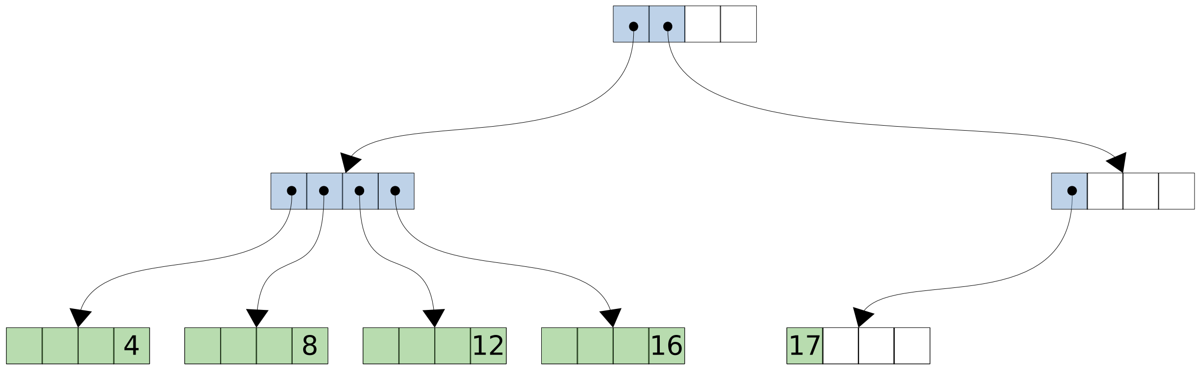
RB-Tree



Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```



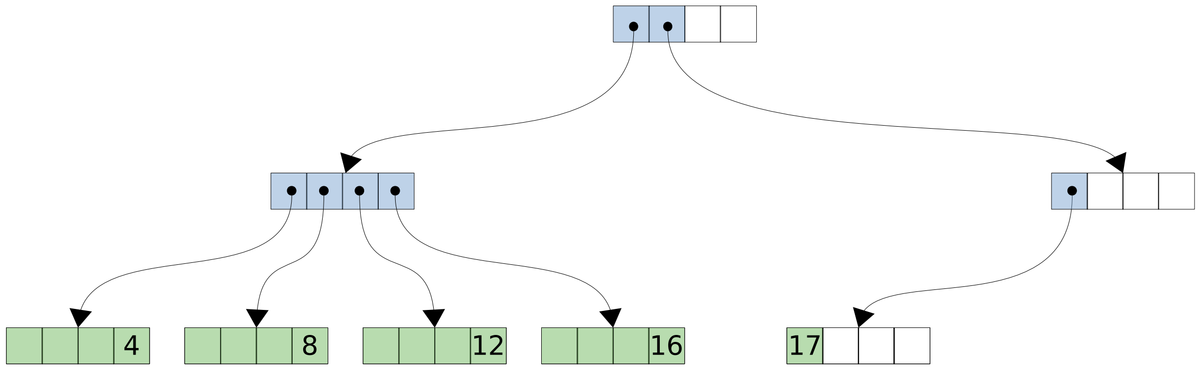
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) Convert `idx` to binary



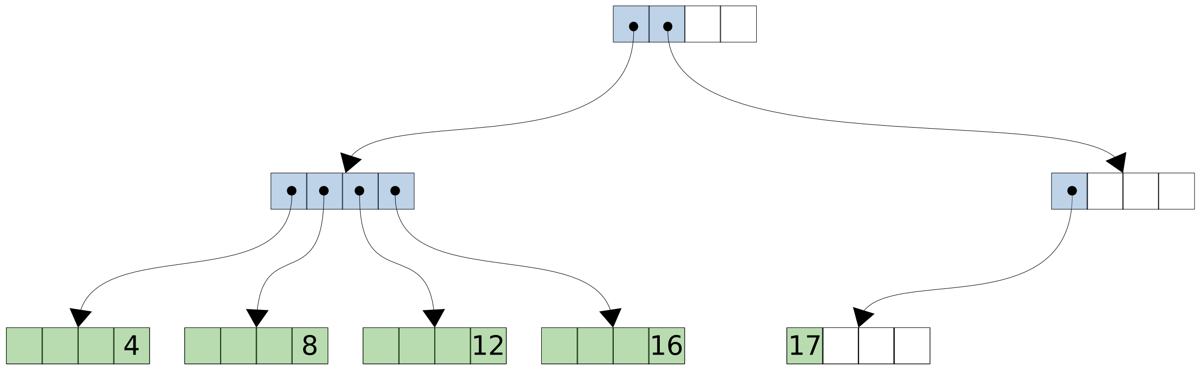
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000



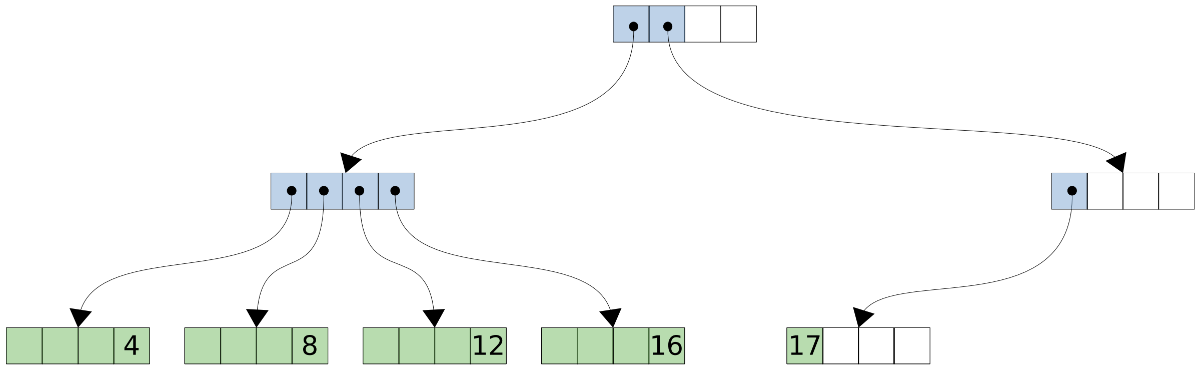
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) Split it into $\log_2 m$ parts



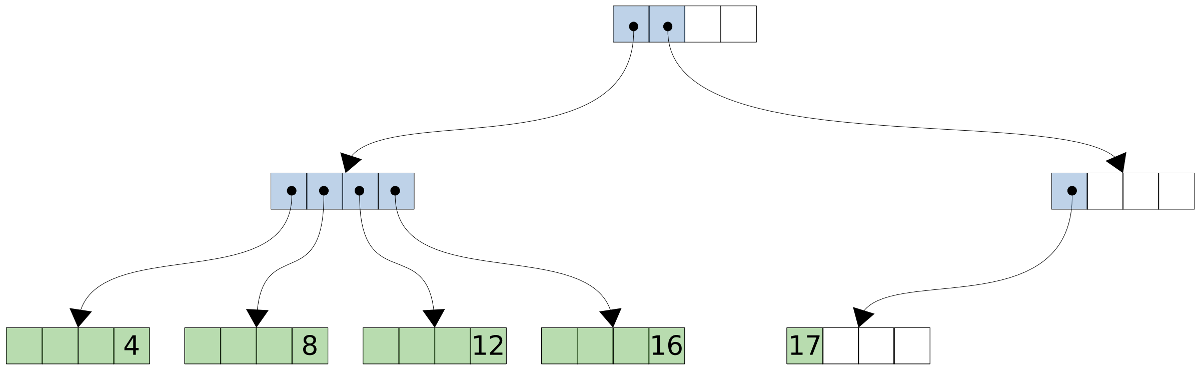
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) 01 00 00



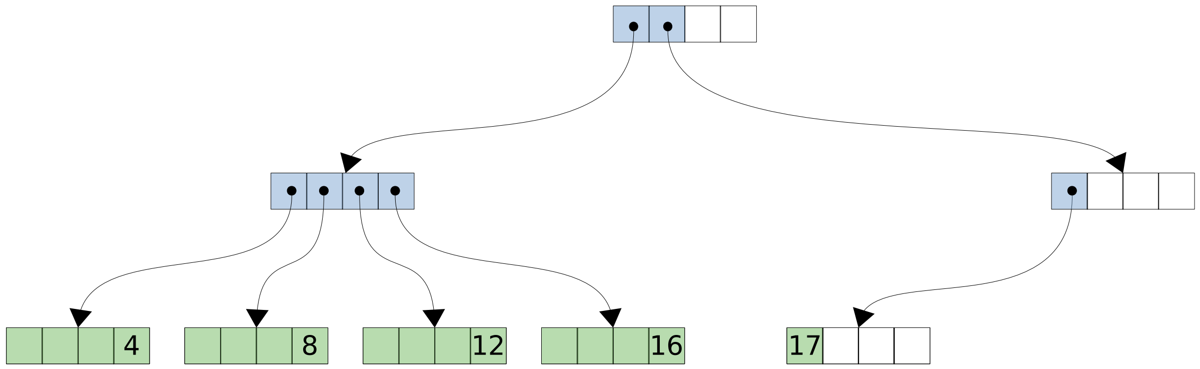
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) 01 00 00
3) Each part is an index!



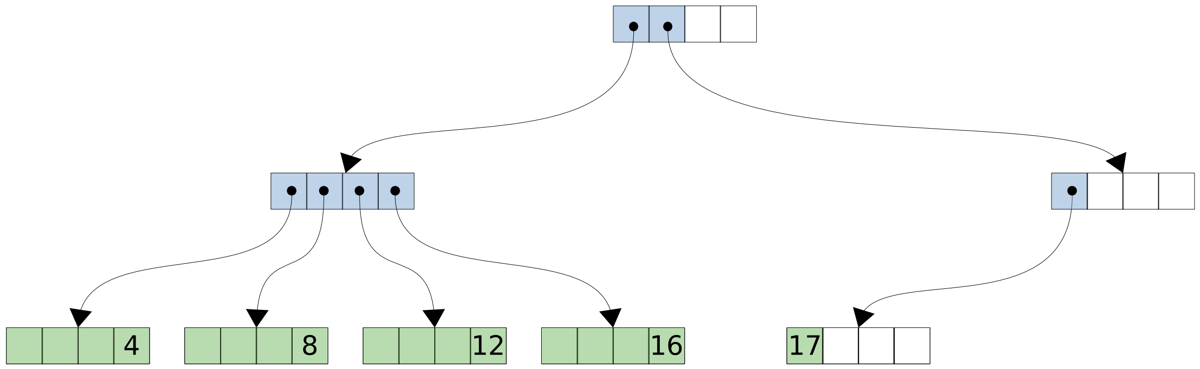
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) 01 00 00
3) Each part is an index!



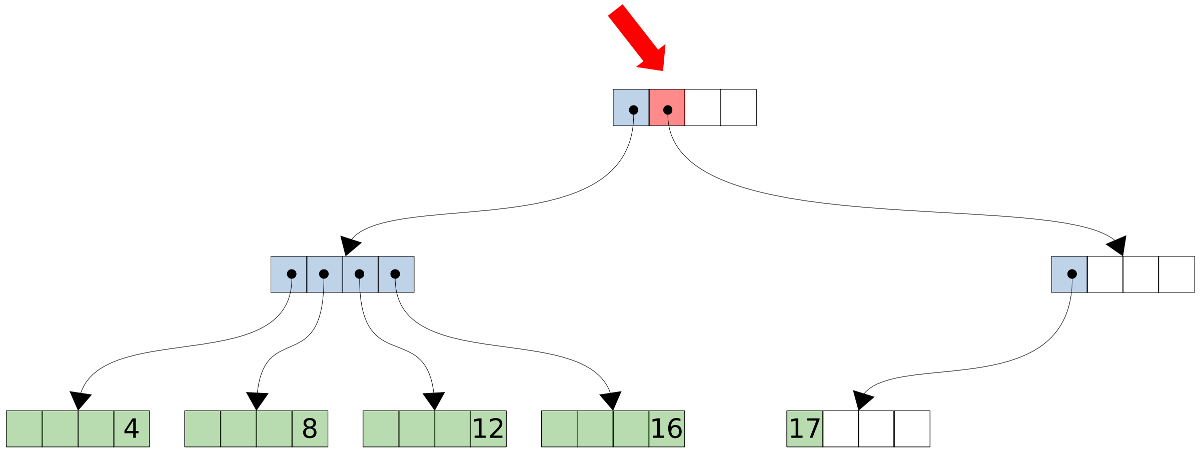
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) 01 00 00
3) Each part is an index!



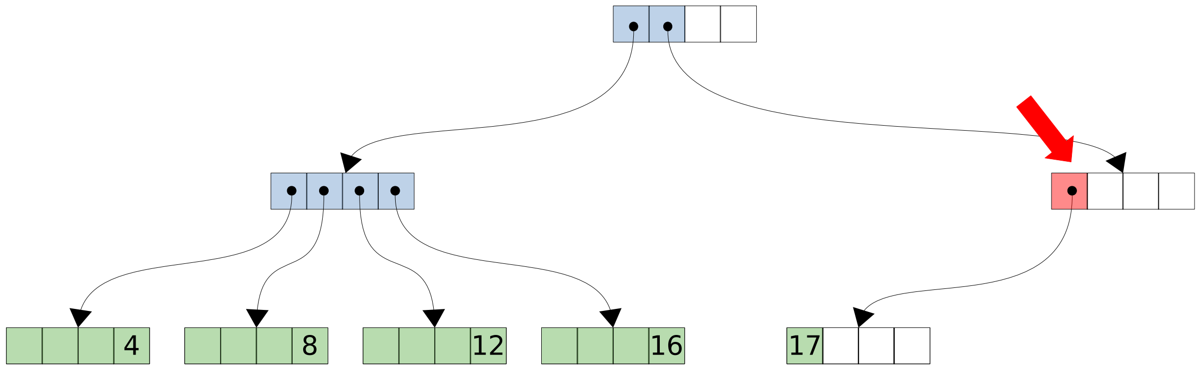
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) 01 00 00
3) Each part is an index!



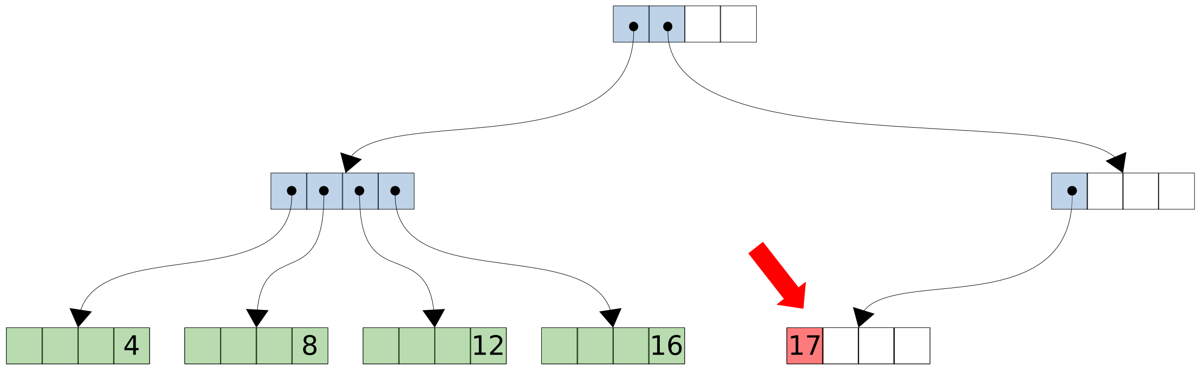
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

Algorithm:

1) 16 == 10000
2) 01 00 00
3) Each part is an index!



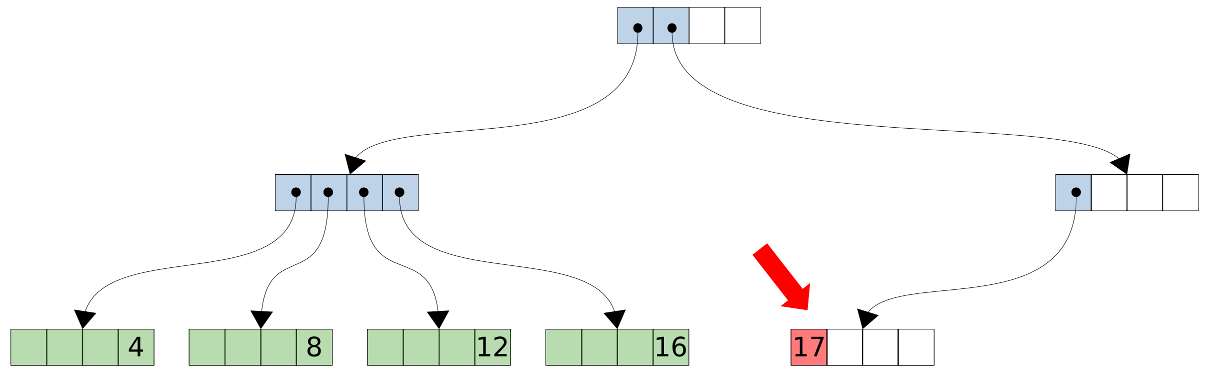
Branching factor (m) = 4

Lookup

```
lookup vec 16 == 17
```

~~Algorith~~

~~1) 16 == 1000~~
~~2) 01 00 00~~
~~3) Each [illegible] s an [illegible] !~~



Branching factor (m) = 4

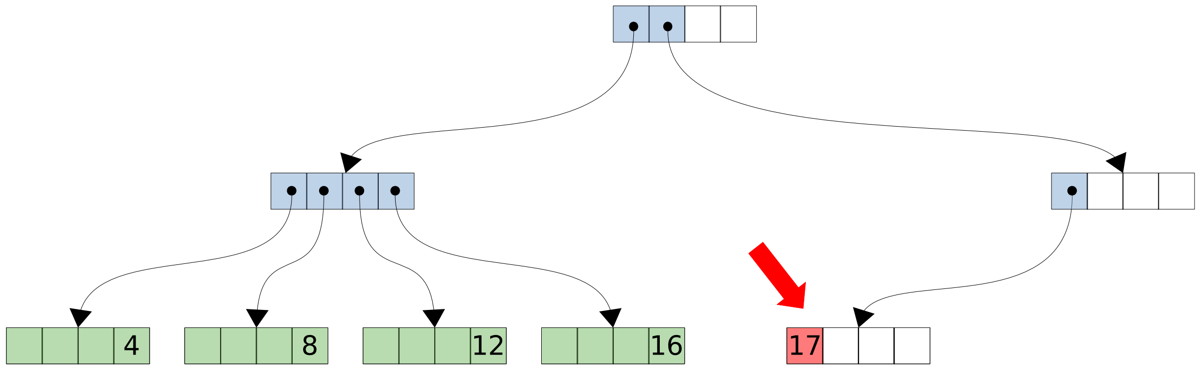
Lookup

```
lookup vec 16 == 17
```

~~Algorith~~

1) ~~16 == 1000~~
2) ~~01 00 00~~
3) ~~Each~~ ~~s an i~~ ~~!~~

```
quot i (m ^ ht) `mod` m
```



Branching factor (m) = 4

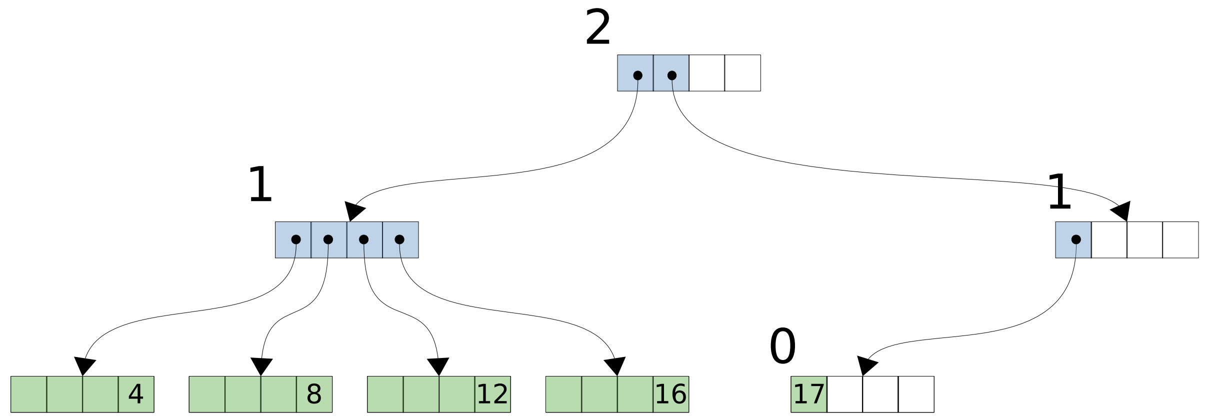
Lookup

```
lookup vec 16 == 17
```

~~Algorithr~~

~~1) 16 == 1000~~
~~2) 01 00 00~~
~~3) Each s an i !~~

```
quot i (m ^ ht) `mod` m
```



Branching factor (m) = 4

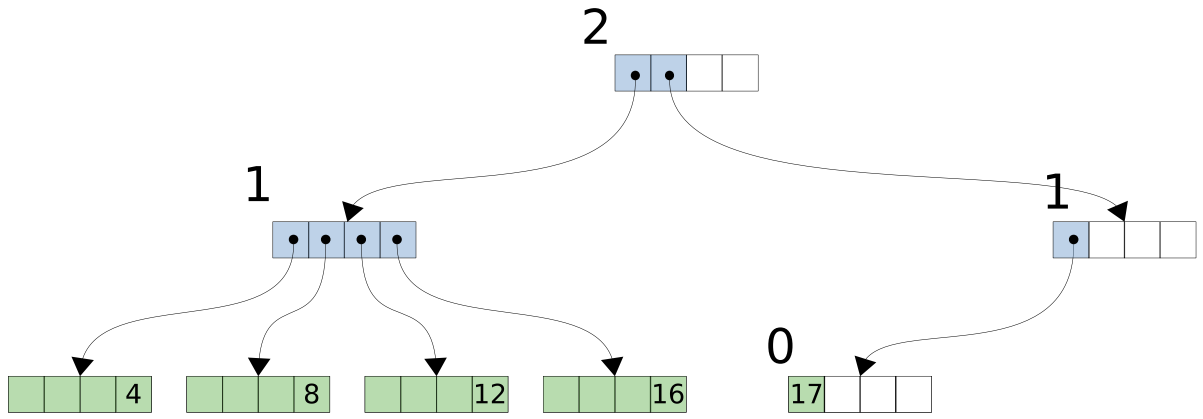
Lookup

```
lookup vec 16 == 17
```

~~Algorith~~

1) ~~16 == 1000~~
2) ~~01 00 00~~
3) ~~Each s an i !~~

```
quot i (m ^ ht) `mod` m
```



Branching factor (m) = 4

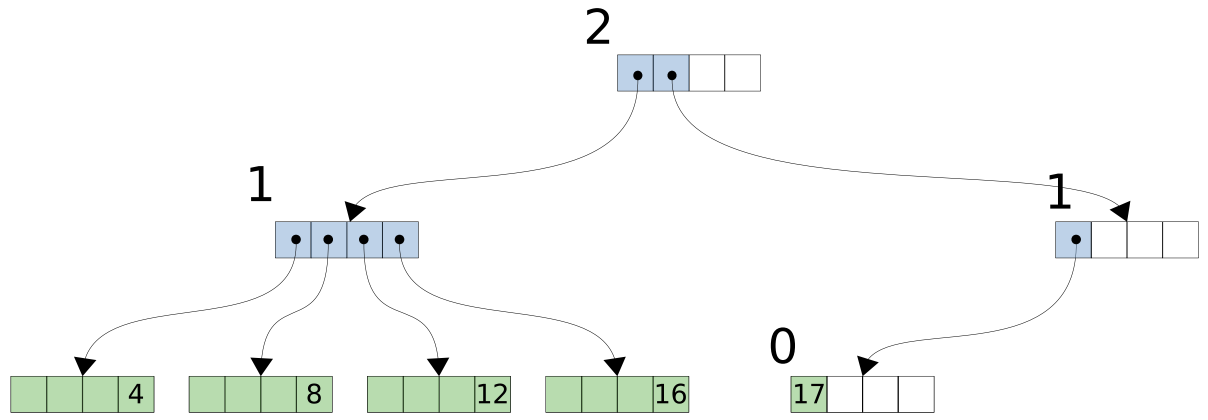
Lookup

```
lookup vec 16 == 17
```

~~Algorith~~

1) ~~16 == 1000~~
2) ~~01 00 00~~
3) ~~Each r s an i !~~

```
quot i (m ^ ht) `mod` m
```



Branching factor (m) = 4

RB-Tree