Understanding Persistent Data Structures

Lesley Lai

@LesleyLai6

http://lesleylai.info/

Referential transparency

Referential transparency

- An expression always evaluates to the same result in any context

Referential transparency

- An expression always evaluates to the same result in any context
- Make both **you** and the **compiler** easier to reason about the code

Referential transparency

- An expression always evaluates to the same result in any context
- Make both **you** and the **compiler** easier to reason about the code
- What we want to achieve

Referential transparency

Referential transparency

- Not True when combine with mutation

Referential transparency

- Not True when combine with mutation
- Shared mutable states are hard to reason about

Referential transparency

- Not True when combine with mutation
- Shared mutable states are hard to reason about

```
const xs = [1, 2, 3];
const ys = xs;
console.log(xs); // [1, 2, 3]
ys[0] = 0;
console.log(xs); // [0, 2, 3]
```

Persistent data structures

- Old values are preserved
- First studied as imperative data structures

Persistent data structures

Level of persistency

Persistent data structures

Level of persistency

- partially persistent

Persistent data structures

Level of persistency

- partially persistent
- fully persistent

Persistent data structures

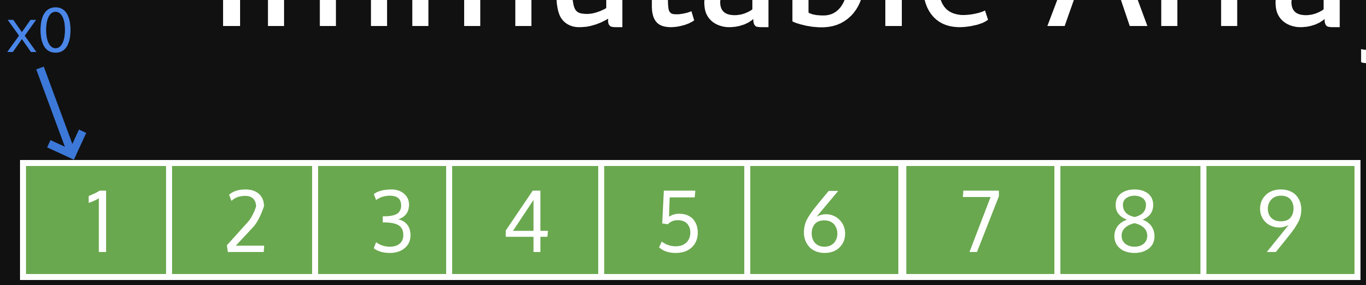
Level of persistency

- partially persistent
- fully persistent
- confluently persistent

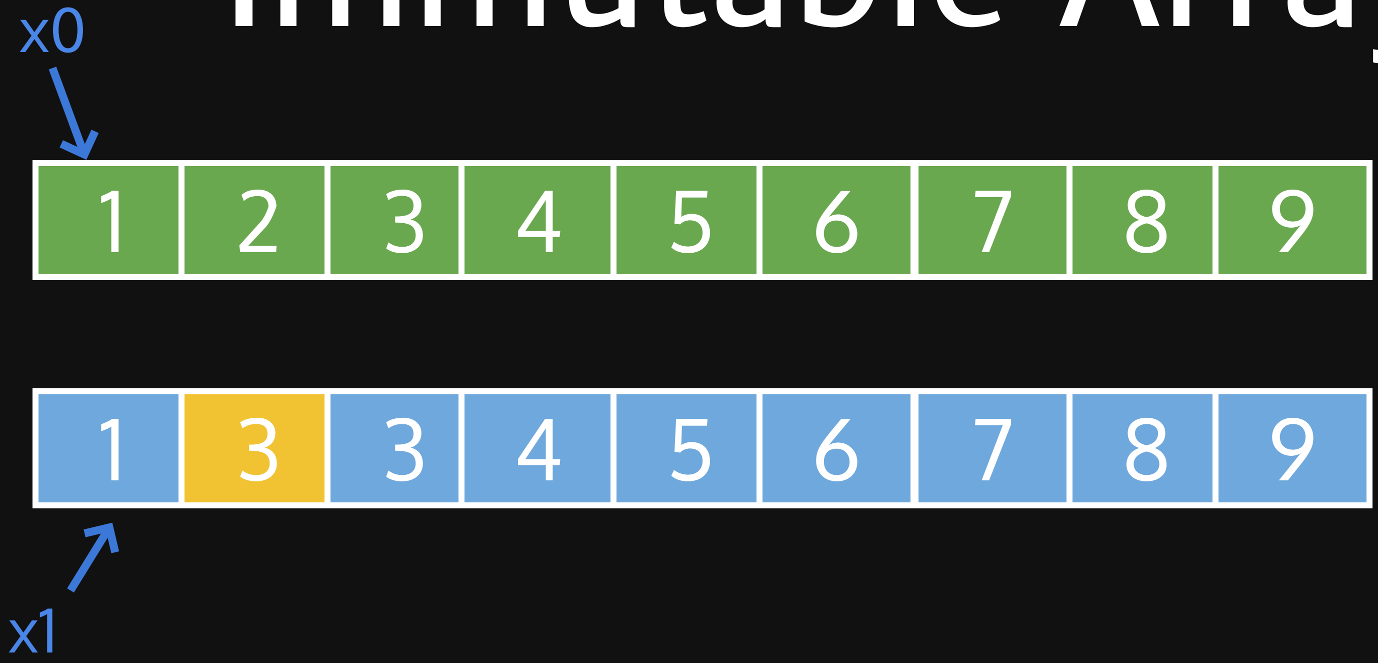
Immutable Array

```
module ImmutableArray = struct
  let set (arr : 'a array) (pos : int) (value : 'a) =
    if (pos >= 0) && (pos < (Array.length arr)) then
      let copied = Array.copy arr in
      Array.unsafe_set copied pos value;
      Some copied
    else
      None

  ...
end

let x = [|1;2;3;4;5|] (* [1,2,3,4,5] *)
let y = ImmutableArray.set x 0 42 (* Some [42,2,3,4,5] *)
```

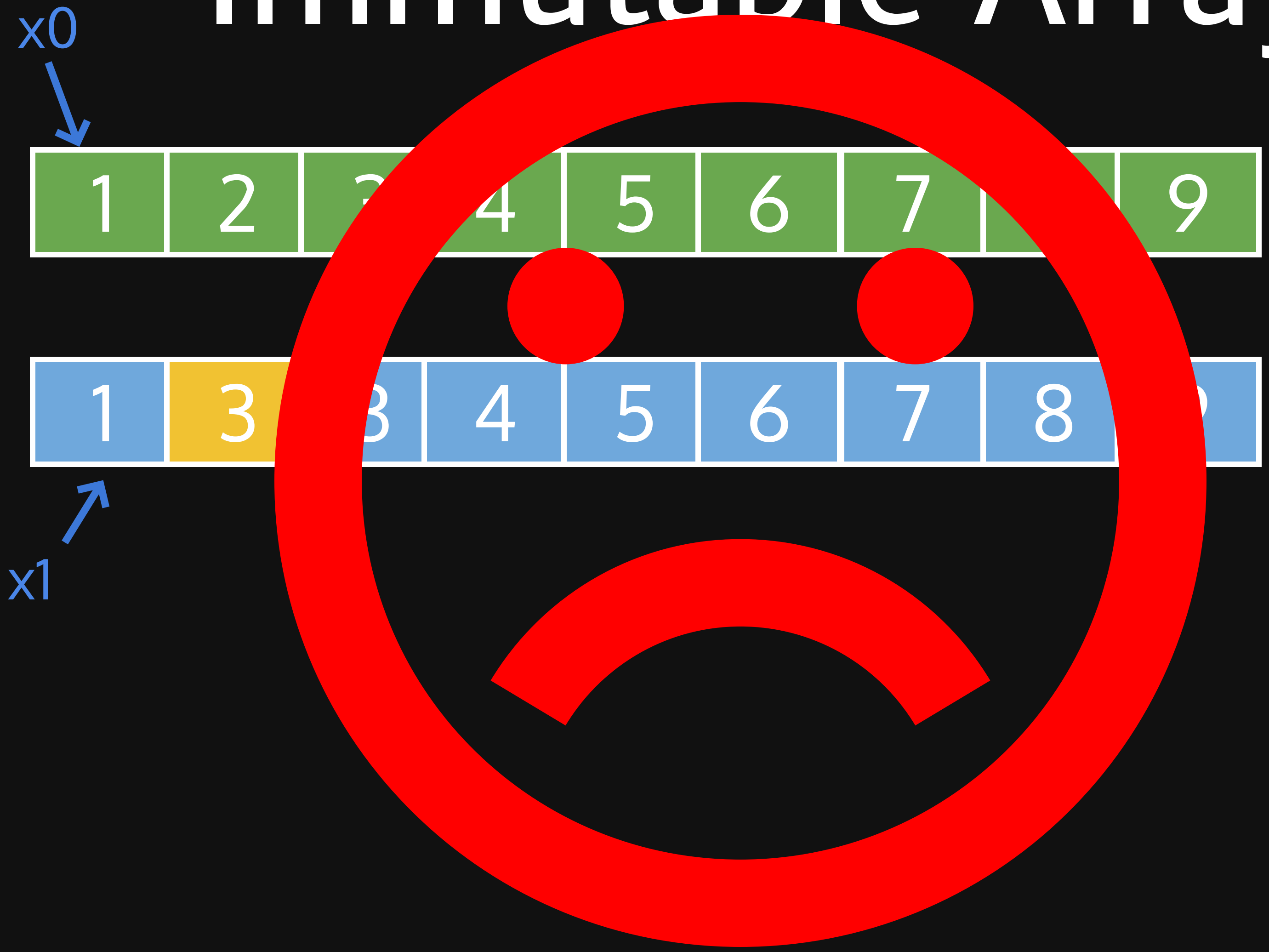
Immutable Array



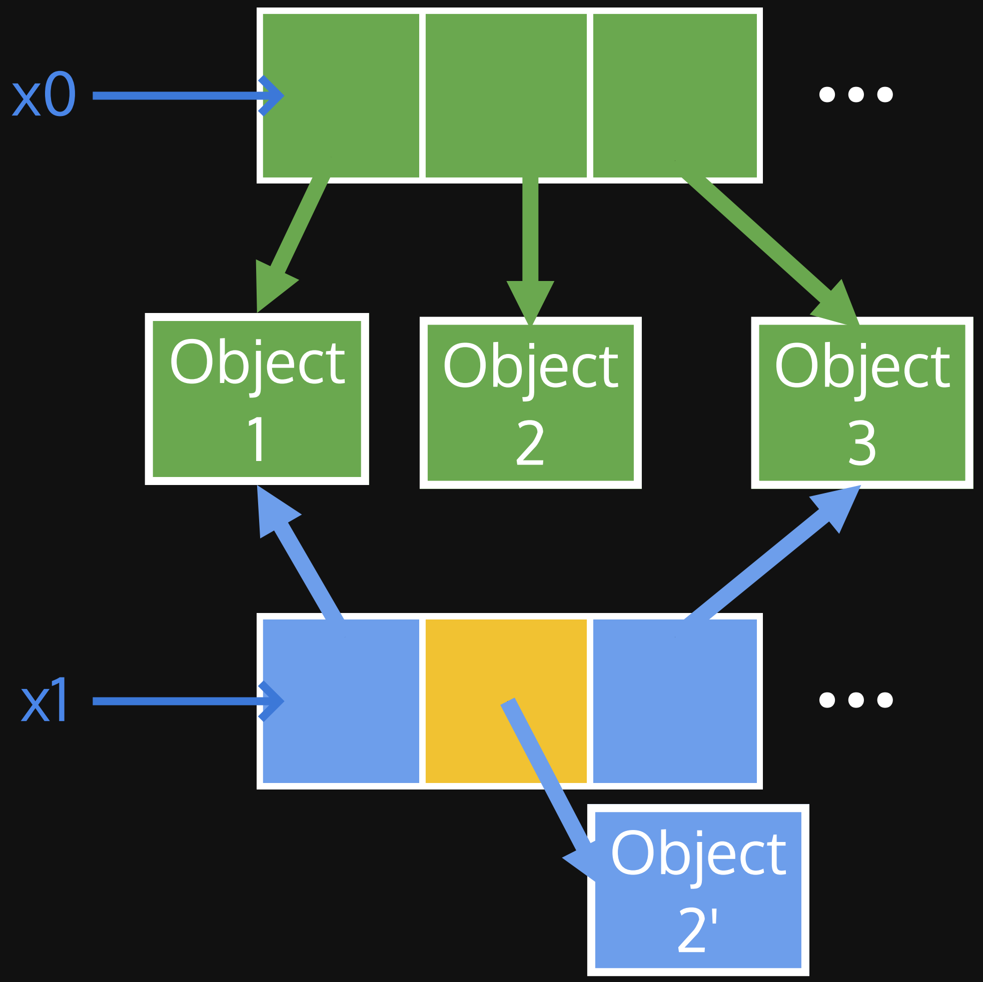
Immutable Array



Immutable Array



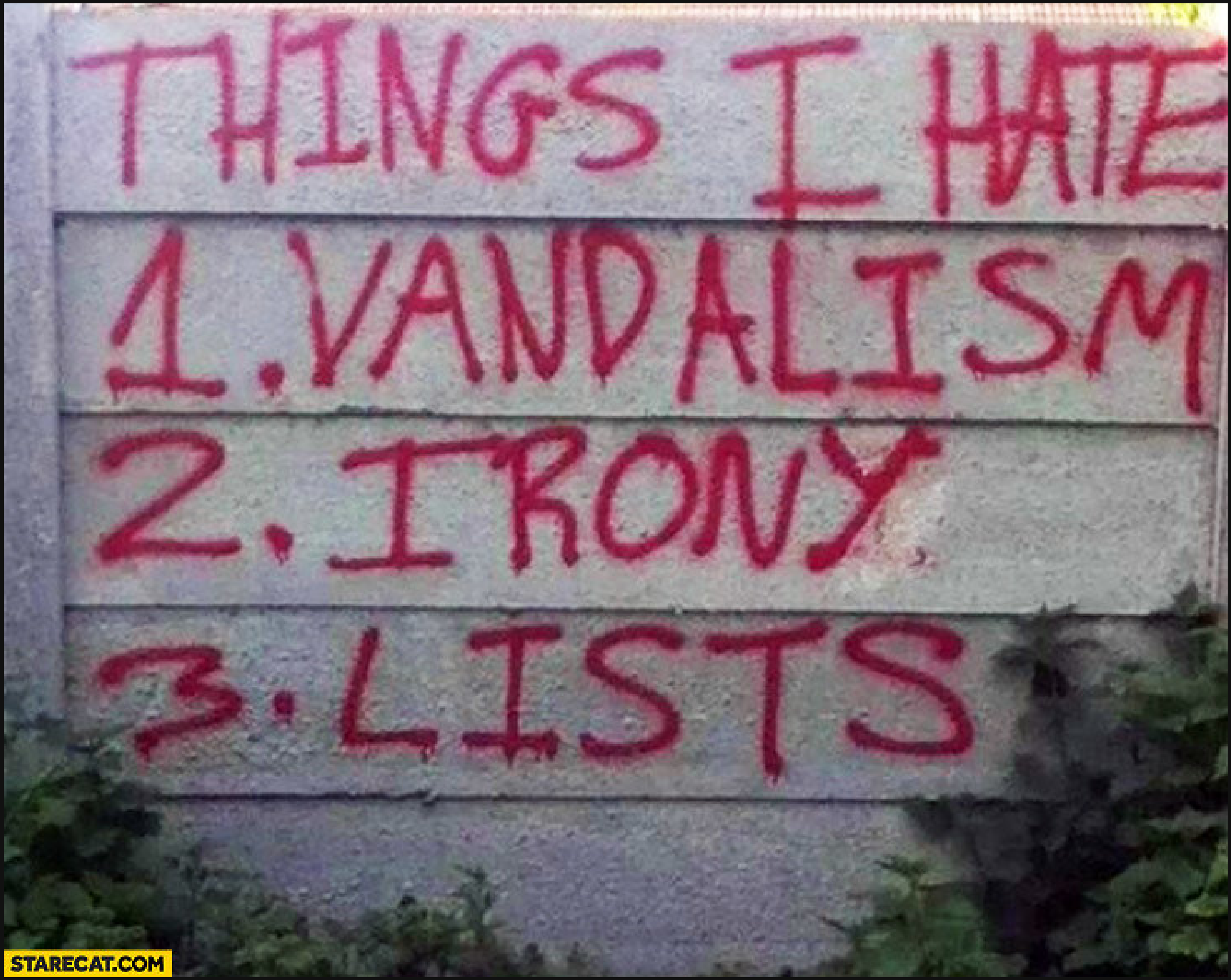
Boxed Type



Immutable Array

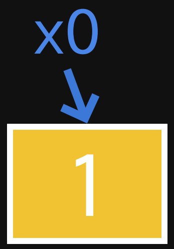
- The most efficient iteration and random access
- Compact
- All manipulations are **O(n)**

Lists



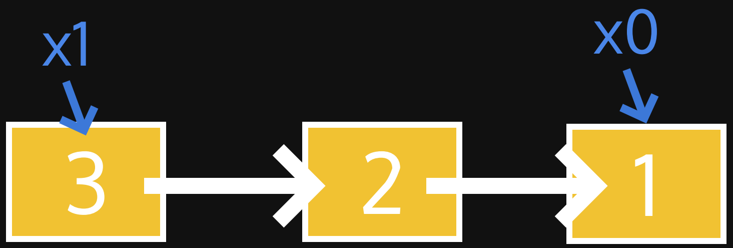
Lists

```
let x0 = [1]
```

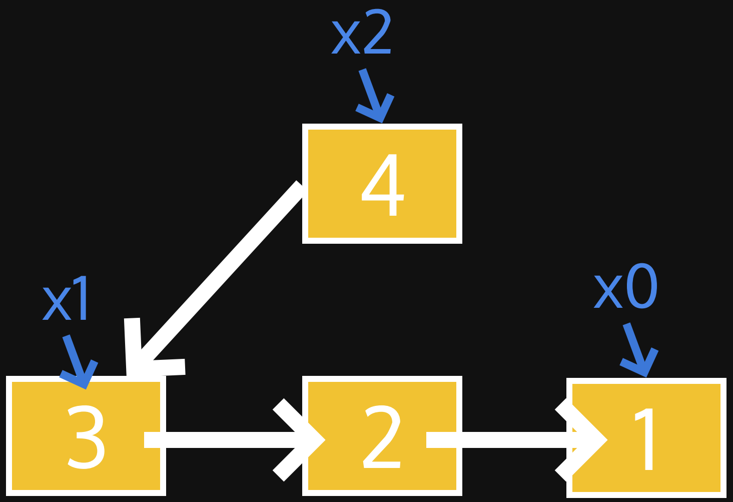


Lists

```
let x1 =
    let x0 = [1] in
    3 :: 2 :: x0
```



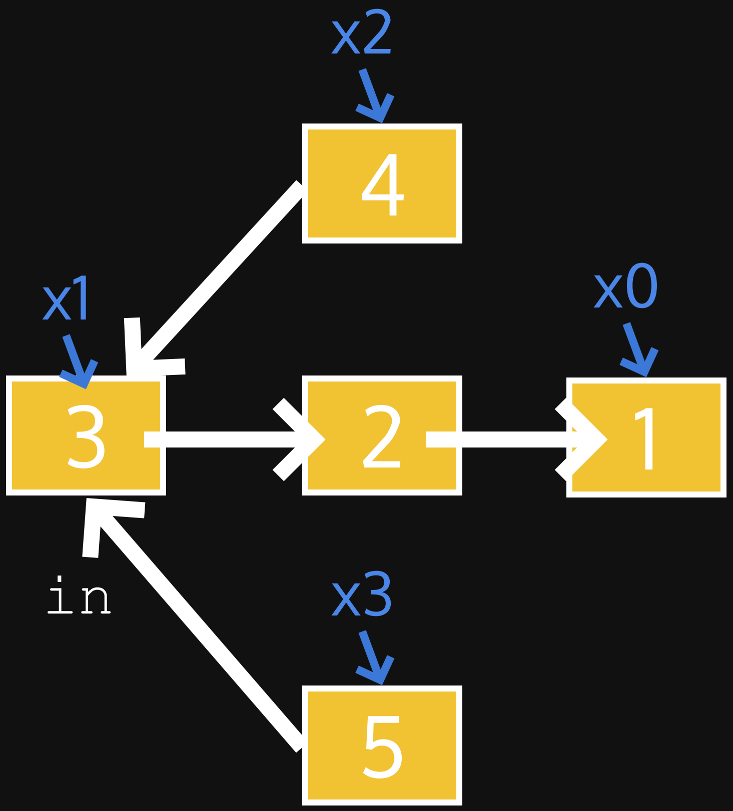
Lists



```
let x2 =
    let x0 = [1] in
    let x1 = 3 :: 2 :: x0 in
    4 :: x1
```

Lists

```
let x3 =
    let x0 = [1] in
    let x1 = 3 :: 2 :: x0 in
    let x2 = 4 :: x1 in
    5 :: x1
```



List Implementaion

```
type 'a myList =
   | Nil
   | Cons of 'a * 'a myList
```

Using List

Pattern match on the inductive definition.

Eg:

```
let map (func: ('a -> 'b)) (list: 'a myList): 'b myList =
    let rec helper xs acc =
            match xs with
              | Nil -> acc
              | Cons (head, tail) -> helper tail (Cons (func head, acc))
    in
    helper list Nil
```

Prefer higher-order functions (map/fold/...) to raw loop/recursion

But don't over generalize

```
["a"; "b"; "c"; "d"]
 |> ([1; 2; 3; 4]
     |> List.fold_left2 (fun acc x y -> (x, y) :: acc) [])
 |> List.rev
```

vs

```
let zip xs ys
  = List.fold_left2 (fun acc x y -> (x, y) :: acc) [] xs ys
  |> List.rev;

zip ["a"; "b"; "c"; "d"] [1; 2; 3; 4]
```

List Operations

O(1)

- Prepend
- Head
- Tail

O(n)

- Concat
- Insert
- Append
- Random access
- Update
- map
- filter
- foldl, foldr

List Performance

Your lists are stacks. Using them as "random accessing sequence" is wrong!

List Performance

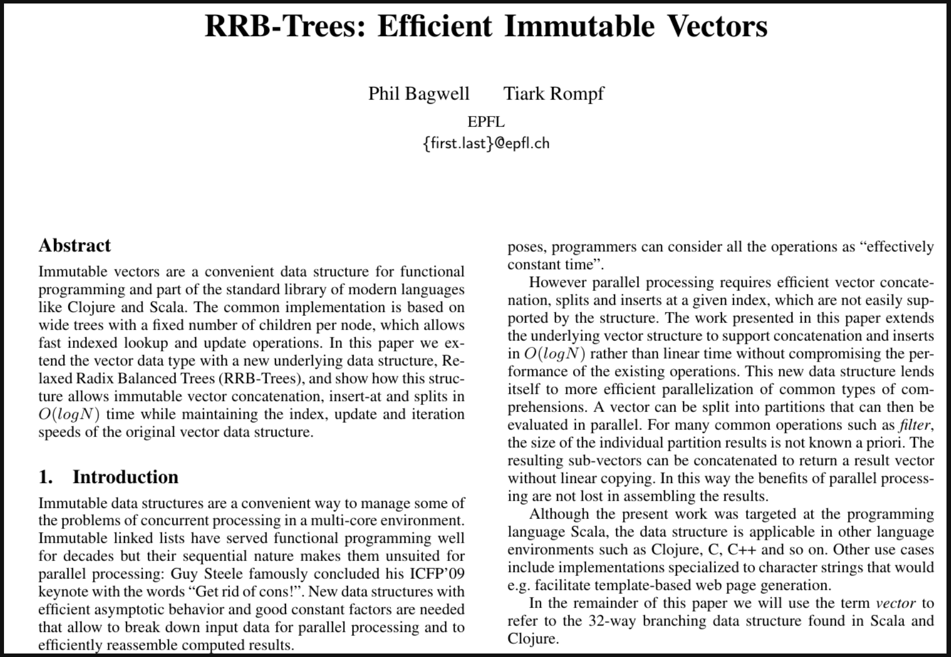
Don't do the following:

```
List.nth lst 9
```

```
if List.length xs == 0 then
    ...
else
    ...
```

```
let rec reverse (xs : 'a list) : 'a list =
  match xs with
  | [] -> []
  | head :: tail -> (reverse tail) @ [head]
```

Relaxed Radix Balanced Tree



RRB-Trees: Efficient Immutable Vectors

Phil Bagwell Tiark Rompf
EPFL
{first.last}@epfl.ch

Abstract

Immutable vectors are a convenient data structure for functional programming and part of the standard library of modern languages like Clojure and Scala. The common implementation is based on wide trees with a fixed number of children per node, which allows fast indexed lookup and update operations. In this paper we extend the vector data type with a new underlying data structure, Relaxed Radix Balanced Trees (RRB-Trees), and show how this structure allows immutable vector concatenation, insert-at and splits in $O(logN)$ time while maintaining the index, update and iteration speeds of the original vector data structure.

1. Introduction

Immutable data structures are a convenient way to manage some of the problems of concurrent processing in a multi-core environment. Immutable linked lists have served functional programming well for decades but their sequential nature makes them unsuited for parallel processing: Guy Steele famously concluded his ICFP'09 keynote with the words "Get rid of cons!". New data structures with efficient asymptotic behavior and good constant factors are needed that allow to break down input data for parallel processing and to efficiently reassemble computed results.

poses, programmers can consider all the operations as "effectively constant time".

However parallel processing requires efficient vector concatenation, splits and inserts at a given index, which are not easily supported by the structure. The work presented in this paper extends the underlying vector structure to support concatenation and inserts in $O(logN)$ rather than linear time without compromising the performance of the existing operations. This new data structure lends itself to more efficient parallelization of common types of comprehensions. A vector can be split into partitions that can then be evaluated in parallel. For many common operations such as *filter*, the size of the individual partition results is not known a priori. The resulting sub-vectors can be concatenated to return a result vector without linear copying. In this way the benefits of parallel processing are not lost in assembling the results.

Although the present work was targeted at the programming language Scala, the data structure is applicable in other language environments such as Clojure, C, C++ and so on. Other use cases include implementations specialized to character strings that would e.g. facilitate template-based web page generation.

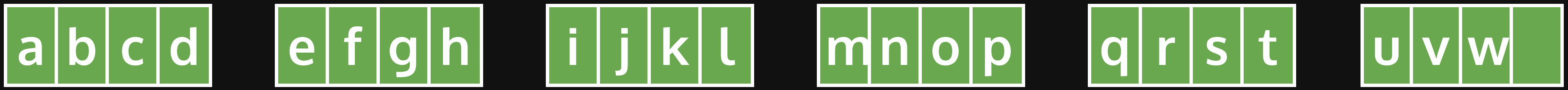
In the remainder of this paper we will use the term *vector* to refer to the 32-way branching data structure found in Scala and Clojure.

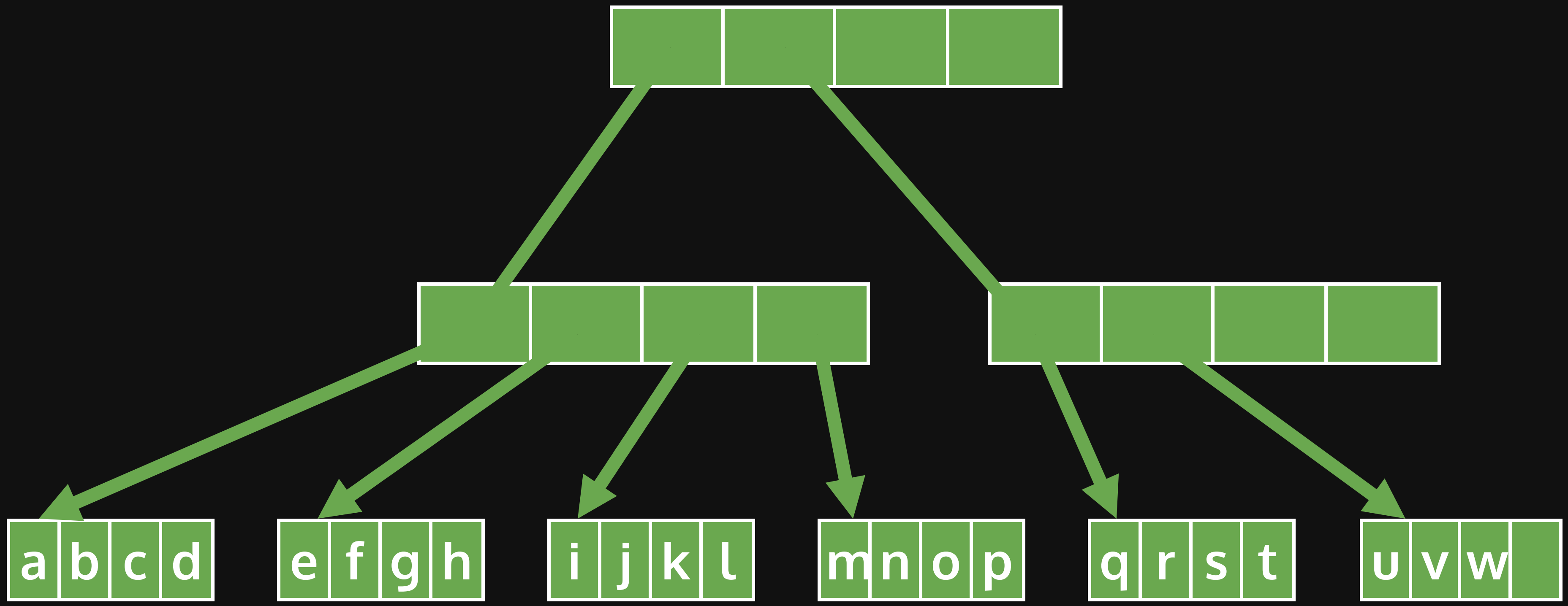
Phil Bagwell

RRB-Trees: Efficient Immutable Vectors

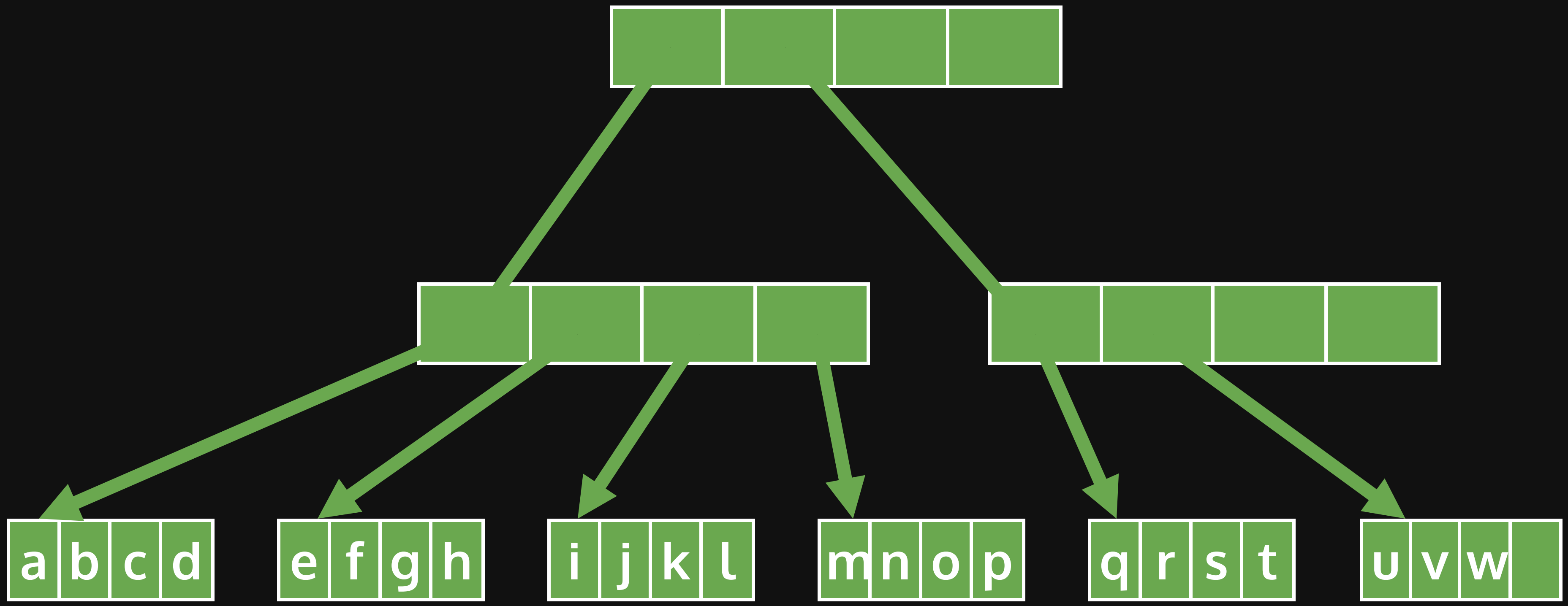
https://infoscience.epfl.ch/record/169879/files/RMTrees.pdf

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w


a b c d
e f g h
i j k l
m n o p
q r s t
u v w

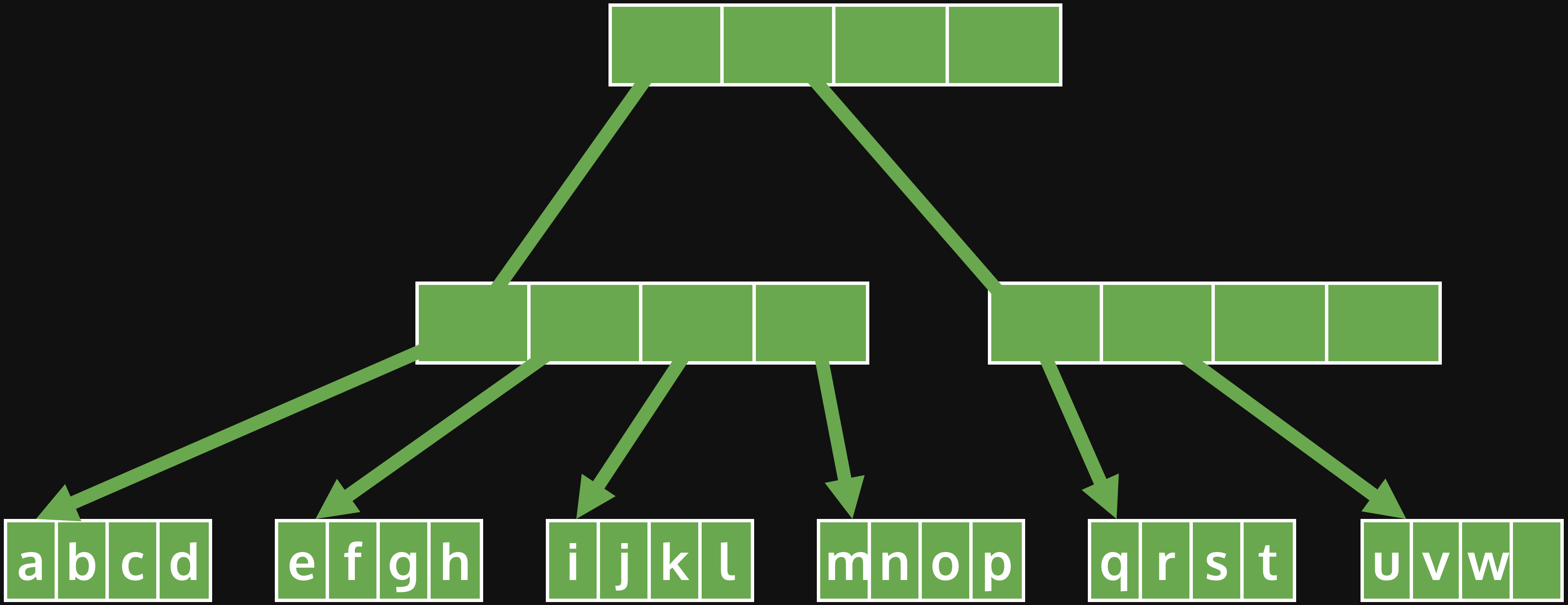

a b c d
e f g h
i j k l
m n o p
q r s t
u v w

Radix Balanced Tree



Radix Balanced Tree

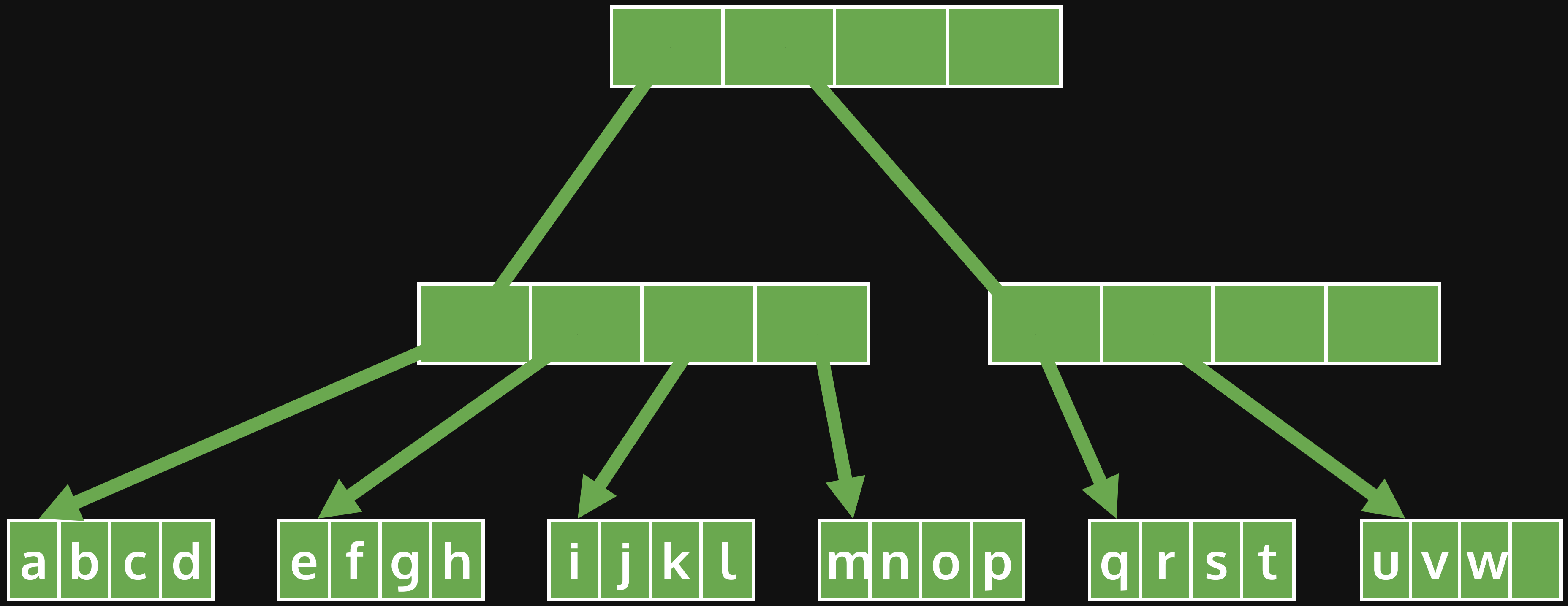
Implement a "Vector", "Dynamic Array"



Radix Balanced Tree Search

```
Vector.get 17
```

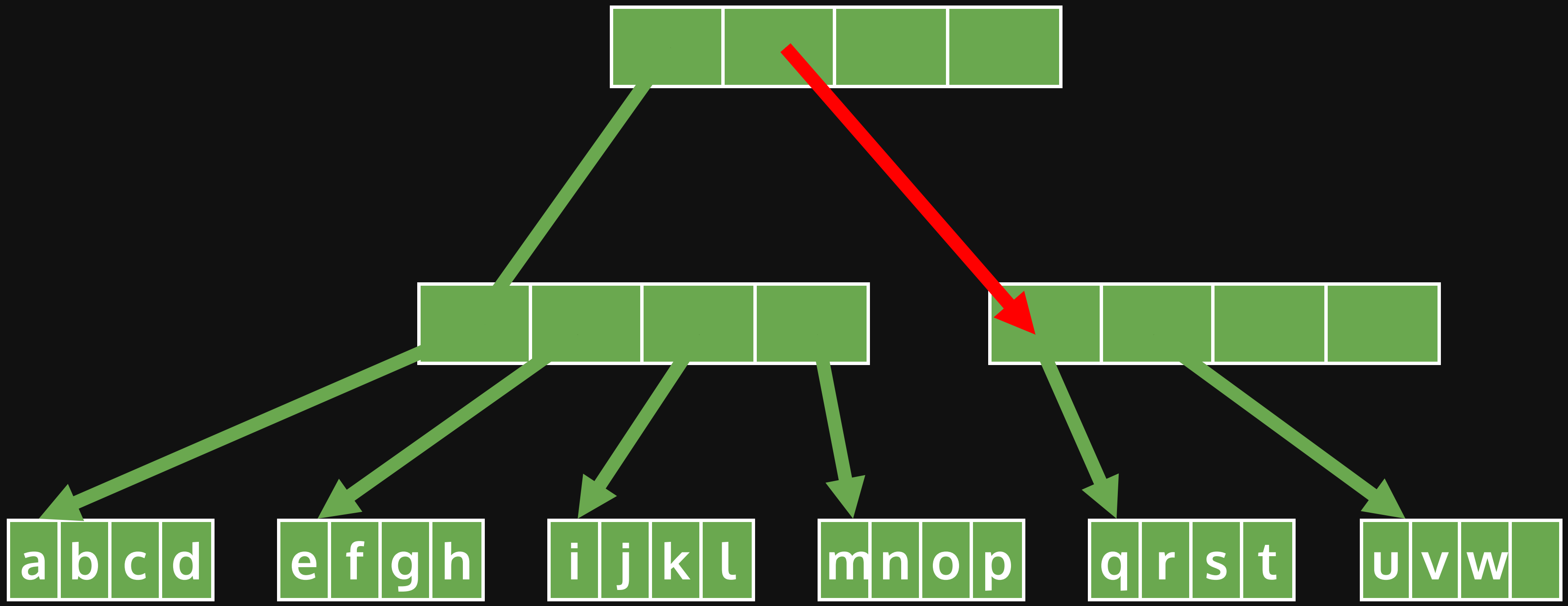
$17 \longrightarrow 01\ 00\ 01$



Radix Balanced Tree Search

```
Vector.get 17
```

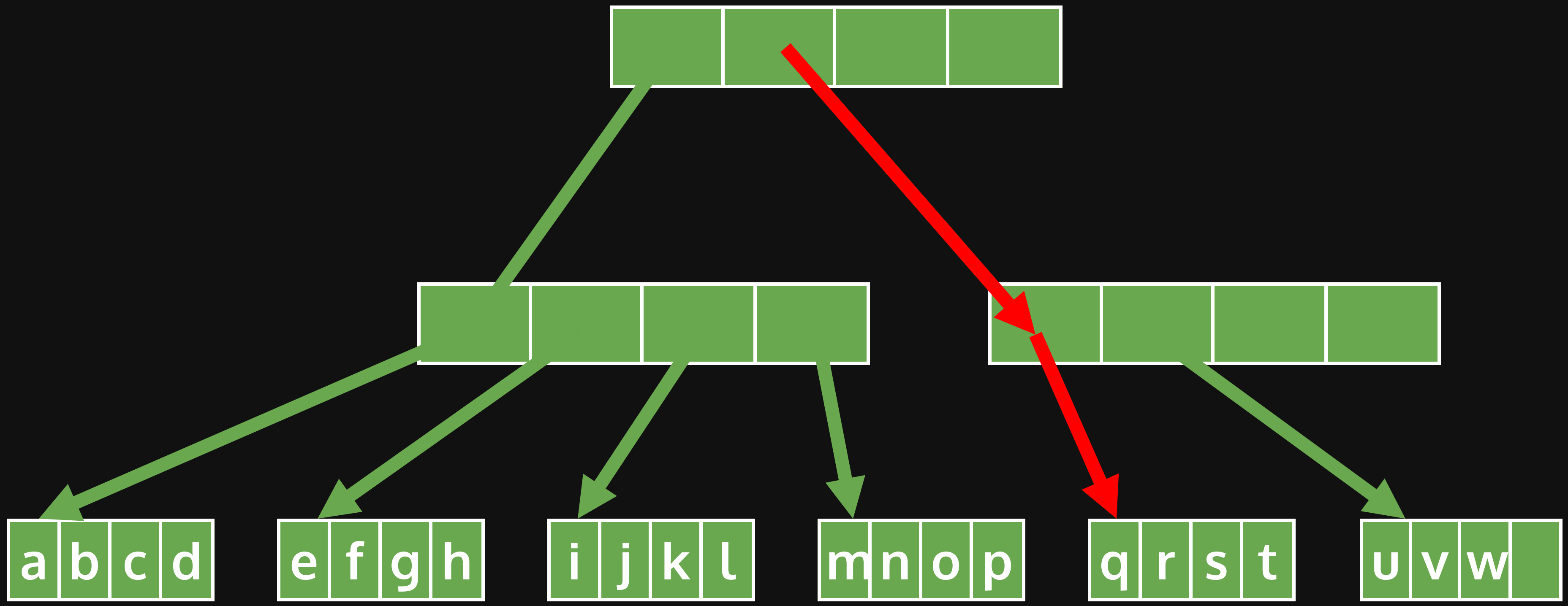
17 → 01 00 01



Radix Balanced Tree Search

```
Vector.get 17
```

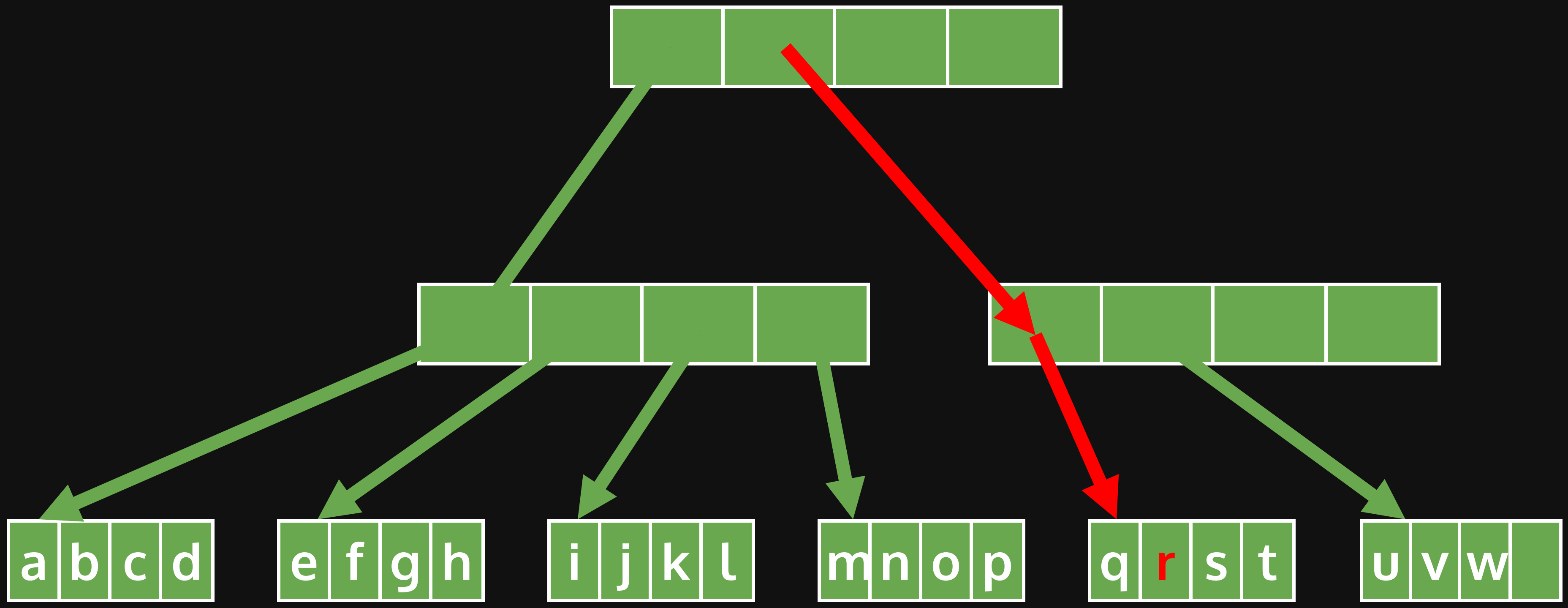
17 → 01 00 01



Radix Balanced Tree Search

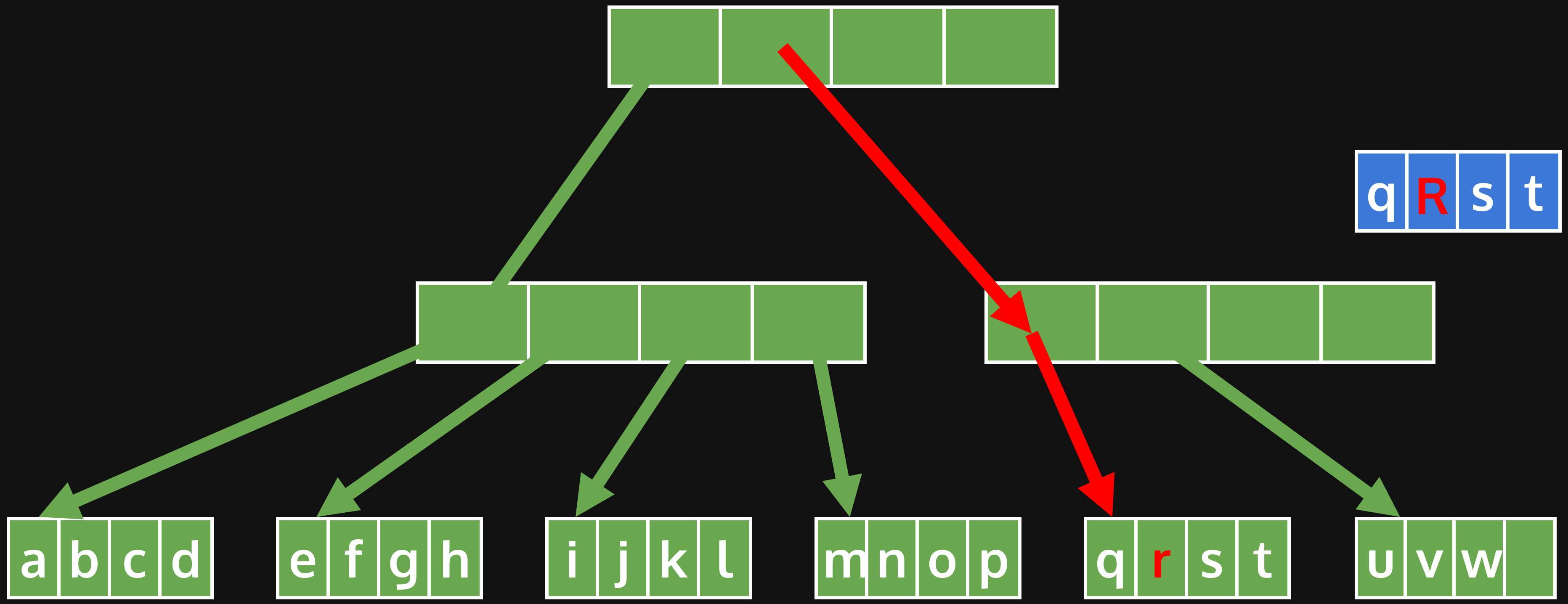
```
Vector.get 17
```

17 → 01 00 01



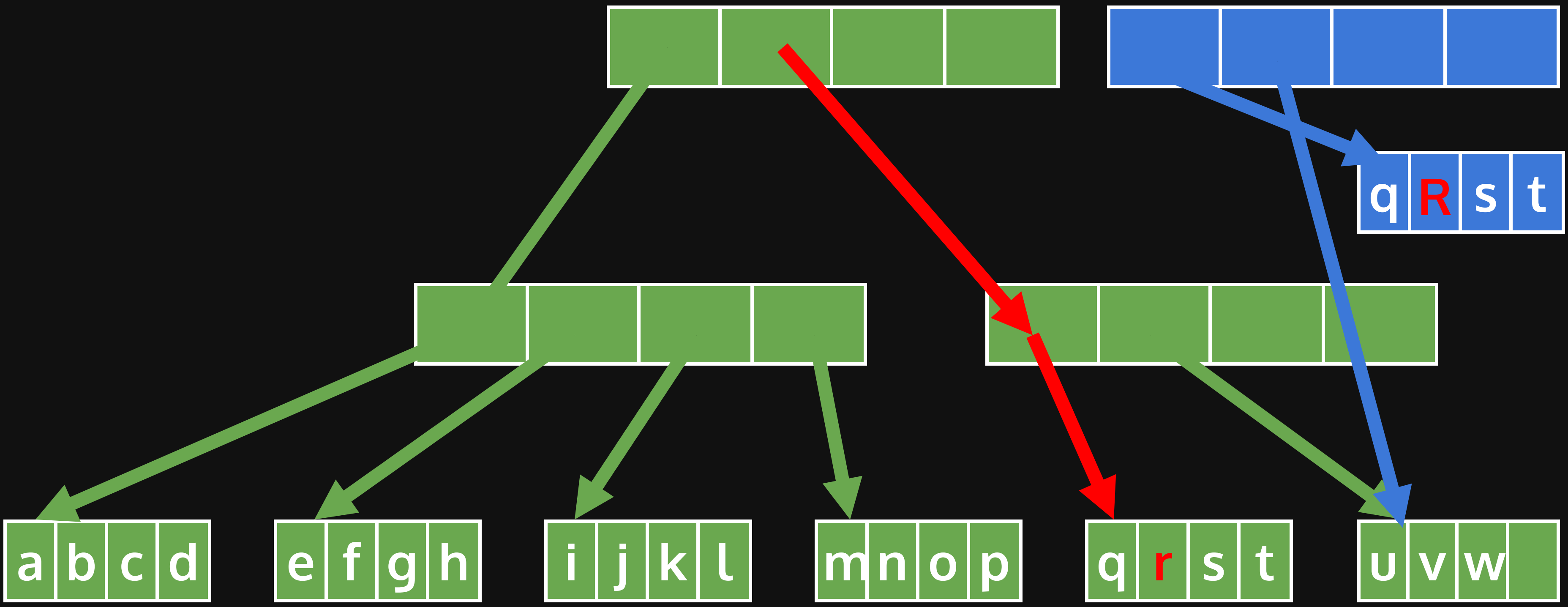
Radix Balanced Tree Update

```
Vector.set 17 "R"
```



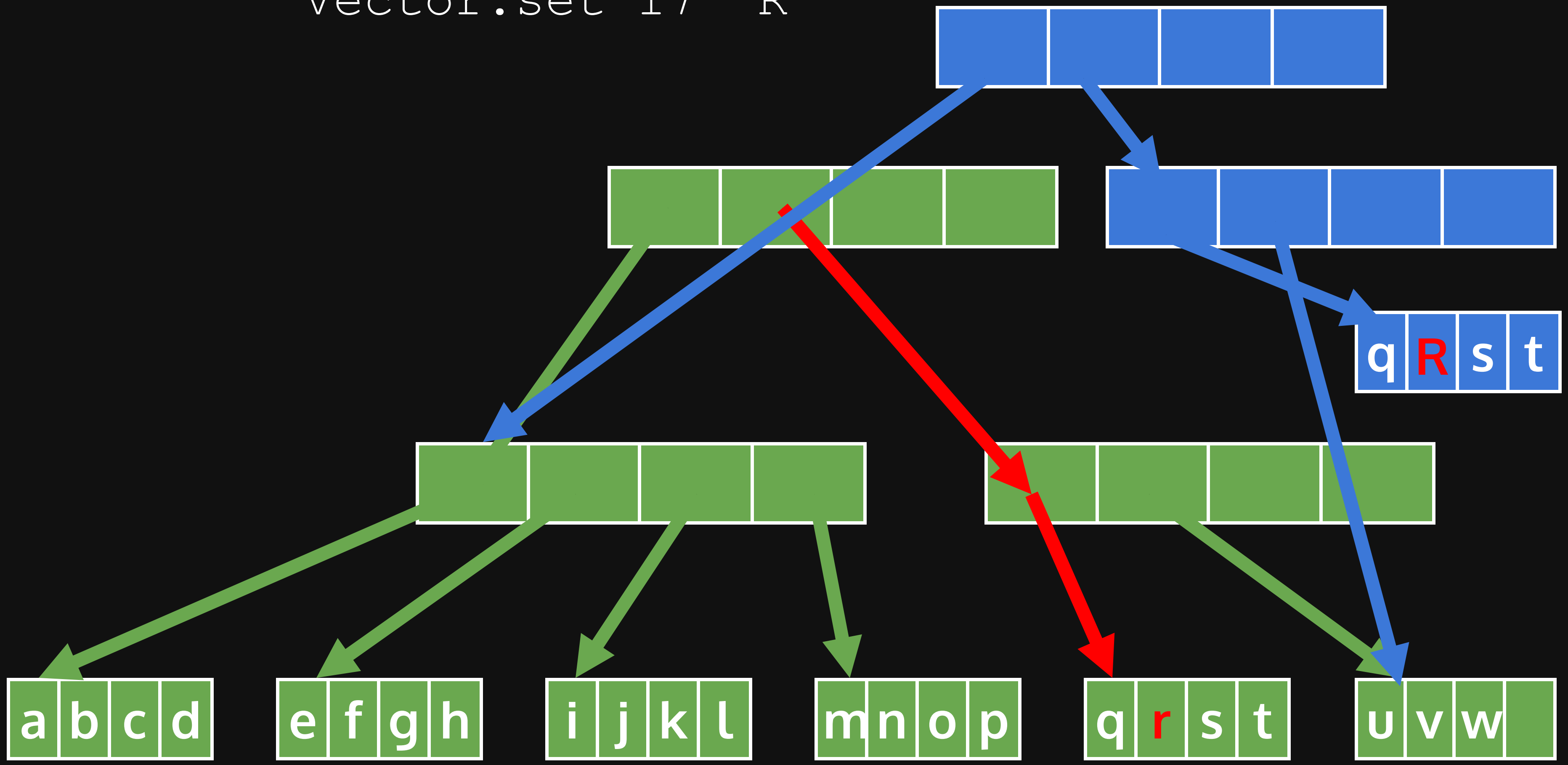
Radix Balanced Tree Update

```
Vector.set 17 "R"
```

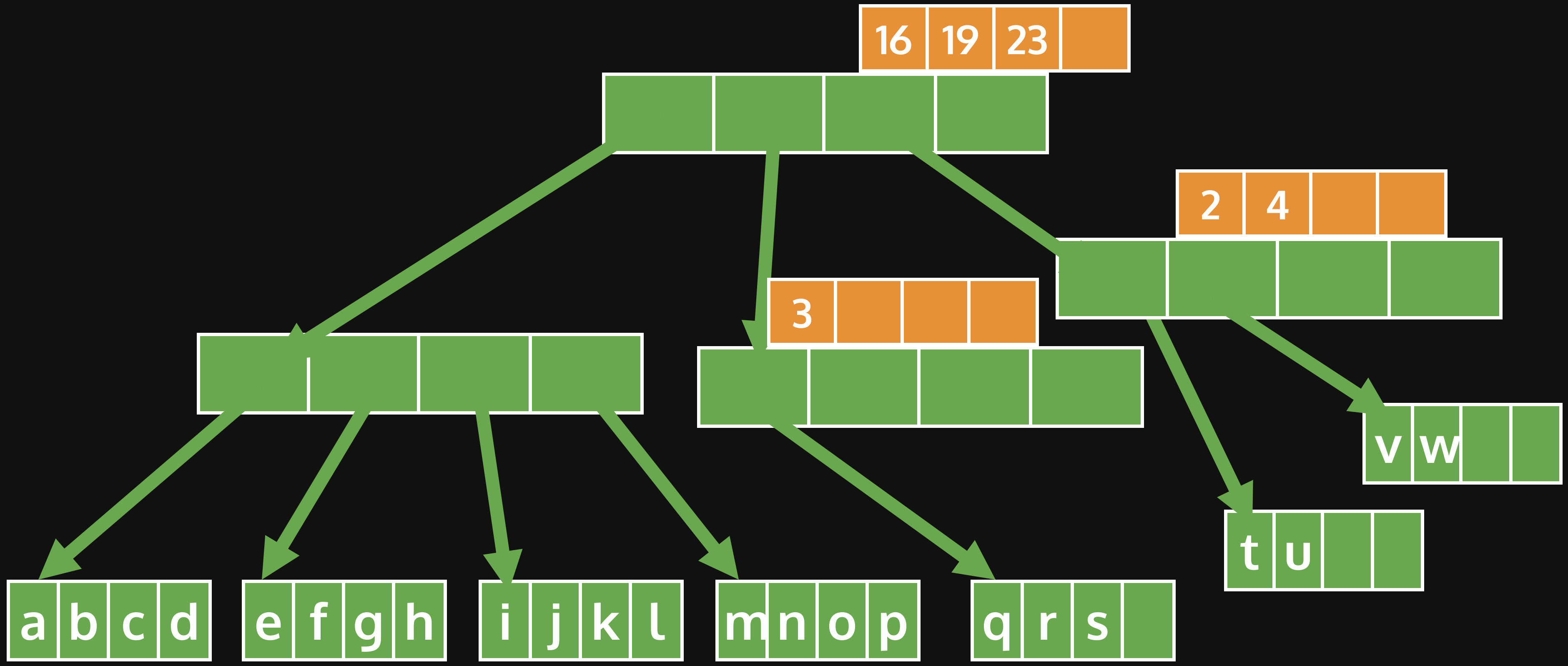


Radix Balanced Tree Update

```
Vector.set 17 "R"
```



Relaxed Radix Balanced Tree



RRB-tree Operations

O(log(n))

- Random access
- Update
- Append
- Slice left/right
- Concat
- Insert
- Prepend

Hash Array Mapped Trie

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

1. INTRODUCTION

The Hash Array Mapped Trie (HAMT) is based on the simple notion of hashing a key and storing the key in a trie based on this hash value. The AMT is used to implement the required structure efficiently. The Array Mapped Trie (AMT) is a versatile data structure and yields attractive alternative to contemporary algorithms in many applications. Here I describe how it is used to develop Hash Trees with near ideal characteristics that avoid the traditional problem, setting the size of the initial root hash table or incurring the high cost of dynamic resizing to achieve an acceptable performance. Tries were first developed by Fredkin [1960] recently implemented elegantly by Bentley and Sedgewick [1997] as the Ternary Search Trees(TST), and by Nilsson and Tikkanen [1998] as Level Path Compressed(LPC) tries. AMT performs 3-4 times faster than TST using 60 percent less space and are faster than LPC tries.

During a search bits are progressively used from the hash to traverse the trie until a key/value pair is found. During insert the AMT levels are extended using more hash bits until a new hash is differentiated from previously stored ones. It will be shown that the methods for Insert, Search and Delete are fast and independent of

Hash Array Mapped Trie

Implement "Unordered Set/Map", "Hash Set/Map", "Dictionary"

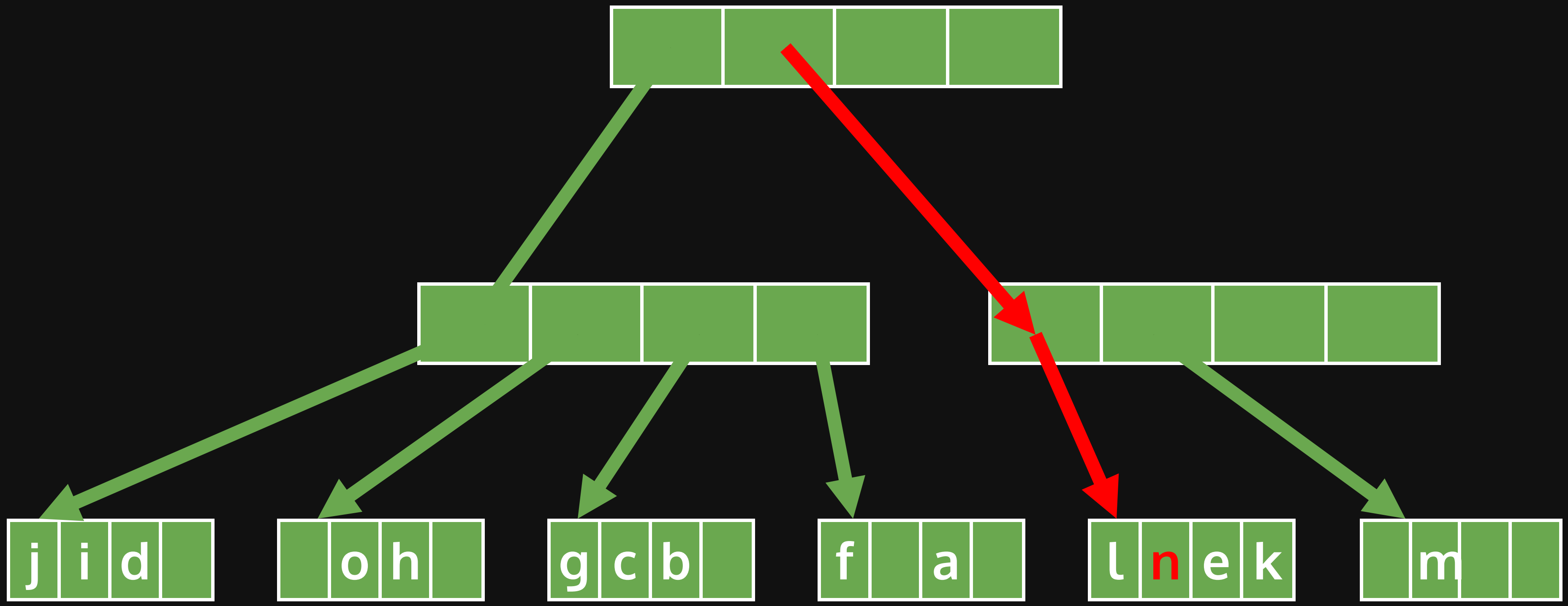
"Trie"

> *" Tries were first described by René de la Briandais in 1959. The term trie was coined two years later by Edward Fredkin, who pronounces it* ***/ˈtriː/*** *(as "tree"), after the middle syllable of retrieval. However, other authors pronounce it* ***/ˈtraɪ/*** *(as "try"), in an attempt to distinguish it verbally from "tree".*

Hash Array Mapped Trie

Using hash values as indices

hash('n') → 01 00 01



HAMT Operations

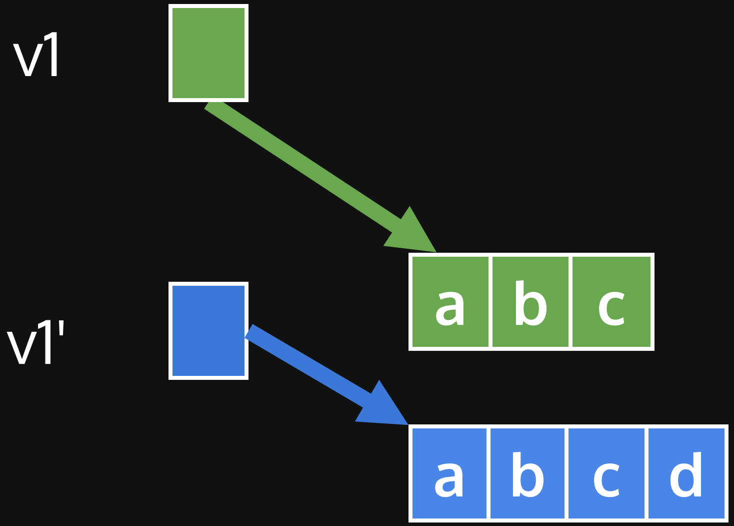
O(log(n))

- Random access
- Delete
- Insert

Common Optimizations

Left Shadow

You don't need to allocate what is not there



Hierarchical vs Flat

Performance Tradeoff of Node Size

Advantage of Small Nodes

- Less copy

- Less indirection
- Better cache locality

Advantage of Big Nodes

"Transient"

- Convert to to imperative data structure temporarily
- Clojure example:

```
(defn vrange [n]
  (loop [i 0 v []]
    (if (< i n)
      (recur (inc i) (conj v i))
      v)))

(defn vrange2 [n]
  (loop [i 0 v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))

;; benchmarked (Java 1.8, Clojure 1.7)
(def v (vrange 1000000))    ;; 73.7 ms
(def v2 (vrange2 1000000))  ;; 19.7 ms
```

Persistent Data Structure Advantages

- Minimum copying
- Compact history
- Fast comparison
- Thread safe for free

Persistence
Pays Off

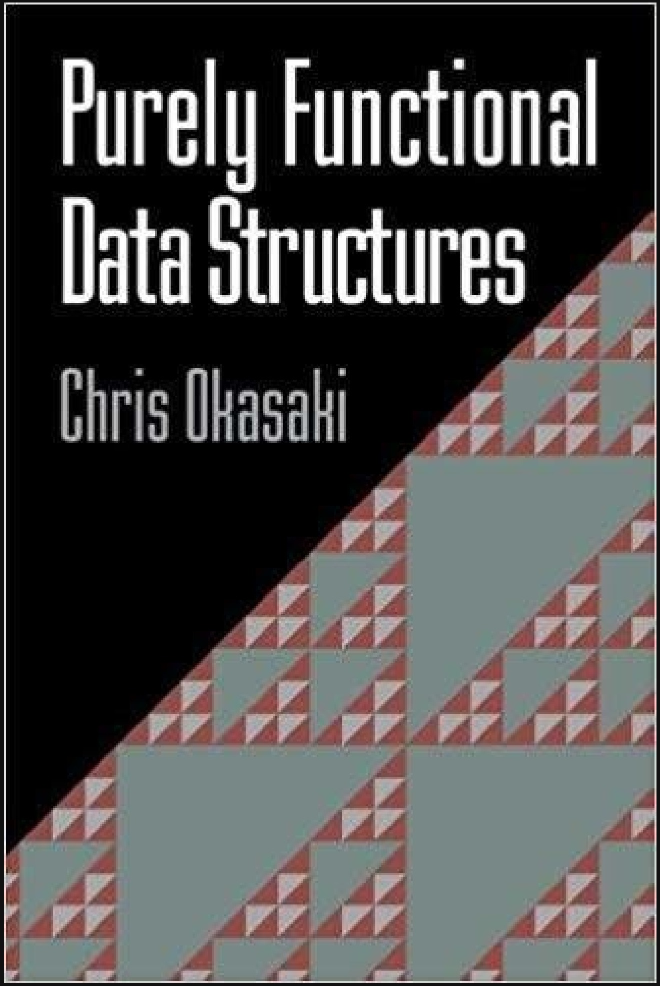
Persistent Data Structure Disadvantages

- Performance overhead
- Shared ownership (need garbage collection)

Some other interesting data structures

- Persistent red-black tree
- Finger Tree
- leftist heap, Binomial Heaps, Brodal Queue (Priority Queue)
- Physicists Queue, Banker Queue, and Real-time Queue (Lazy Queue)

Resources



Chris Okasaki
Purely Functional Data Structures
https://www.amazon.com/Purely-Functional-Data-Structures-Okasaki/dp/0521663504

Resources

Dr. Matthew Hammer
CSCI 4830-016 Principles of Functional Programming
http://matthewhammer.org/courses/pfp-s18/

Thank you!