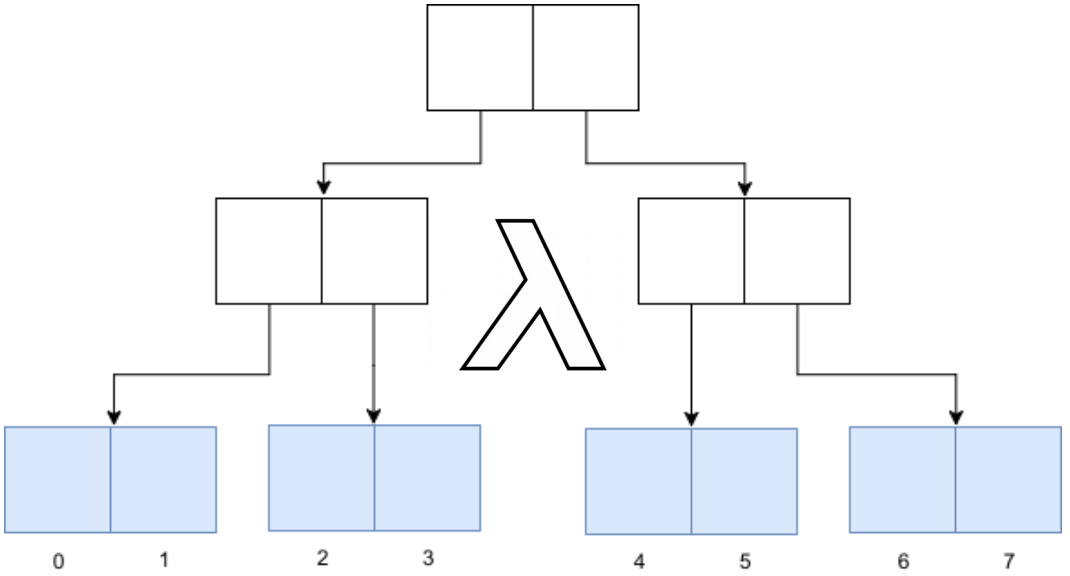
Marvellous Functional Datastructures: The Vector



Hello!

- Robert

Hello!

- Robert
- Scala & Clojure

Hello!

- Robert
- Scala & Clojure
- Obsessed with FP

Vectors: An Introduction

- What are Vectors?

Vectors: An Introduction

- What are Vectors?
- Properties:
  - Append (effectively) constant time (log32n)
  - Look-up (effectively) constant time (log32n)
  - Update (effectively) constant time (log32n)
  - Pop (effectively) constant time (log32n)
  - Iterate linear time

Vectors: An Introduction

- What are Vectors?
- **H**ash **A**rray **M**apped **T**ries (HAMT)

Vectors: An Introduction

- What are Vectors?
- **H**ash **A**rray **M**apped **T**ries (HAMT)
  - Tries?

Vectors: An Introduction

- What are Vectors?
- **H**ash **A**rray **M**apped **T**ries (HAMT)
  - Tries?
  - Trie, Radix Tree, Prefix Tree

Vectors: An Introduction

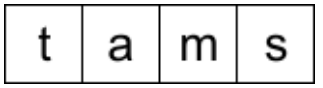
- What are Vectors?
- **H**ash **A**rray **M**apped **T**ries (HAMT)
  - Tries?
  - Trie, Radix Tree, Prefix Tree
  - Structured based on their contents

Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit

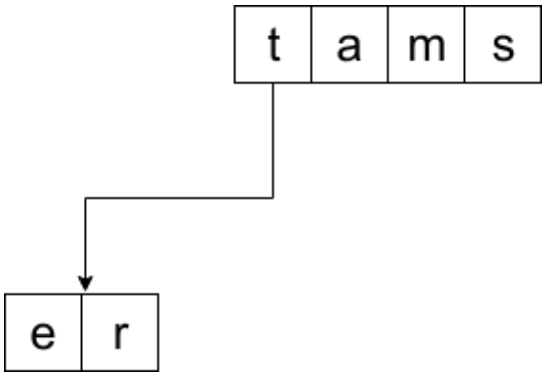
Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit



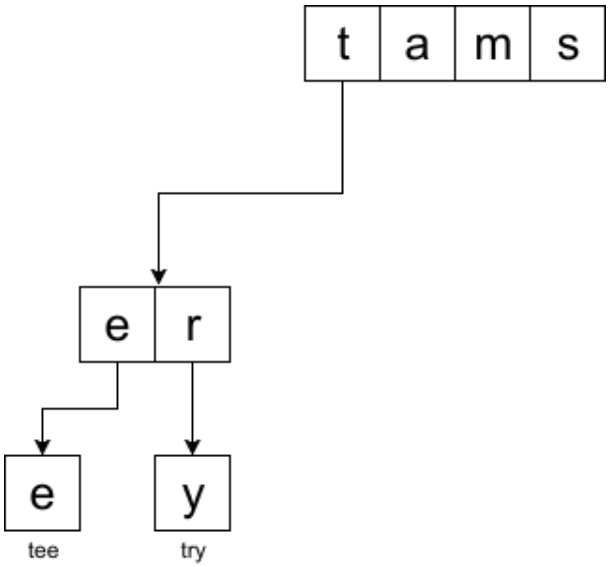
Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit



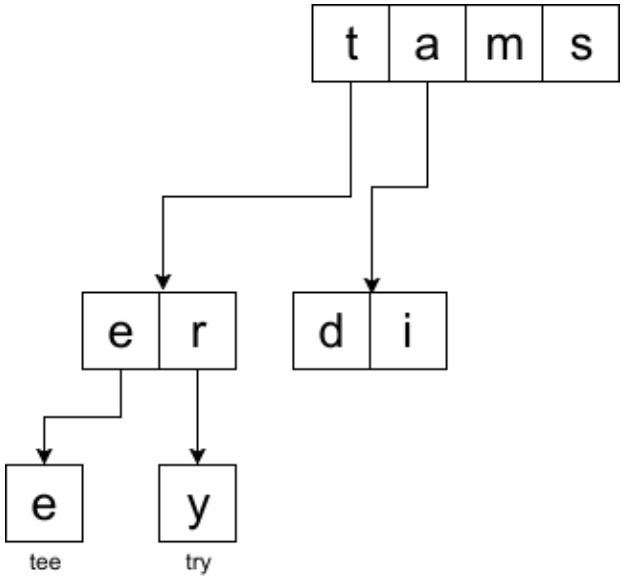
Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit



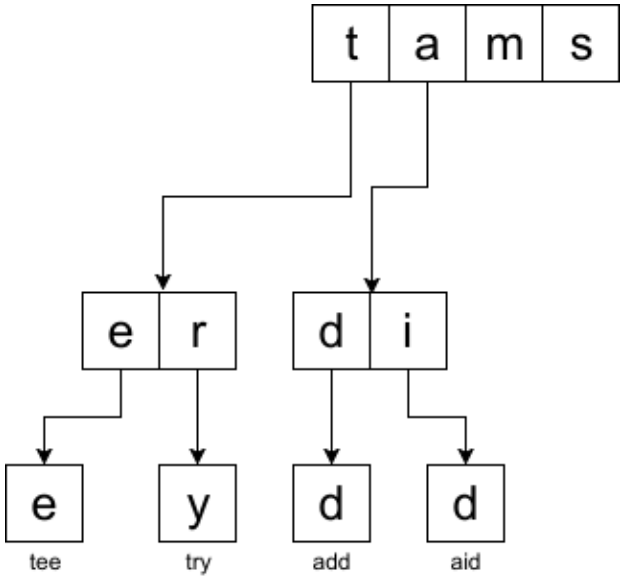
Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit



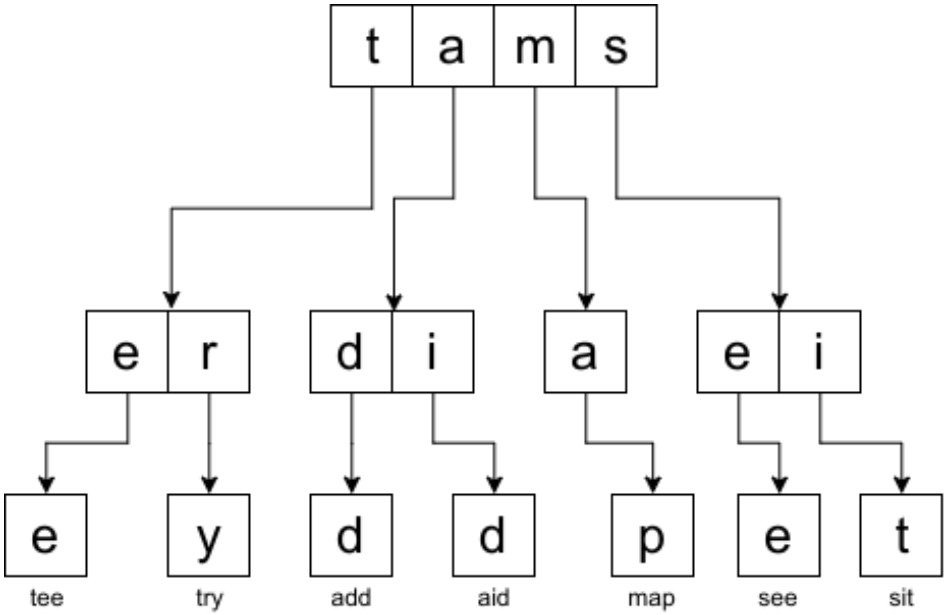
Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit



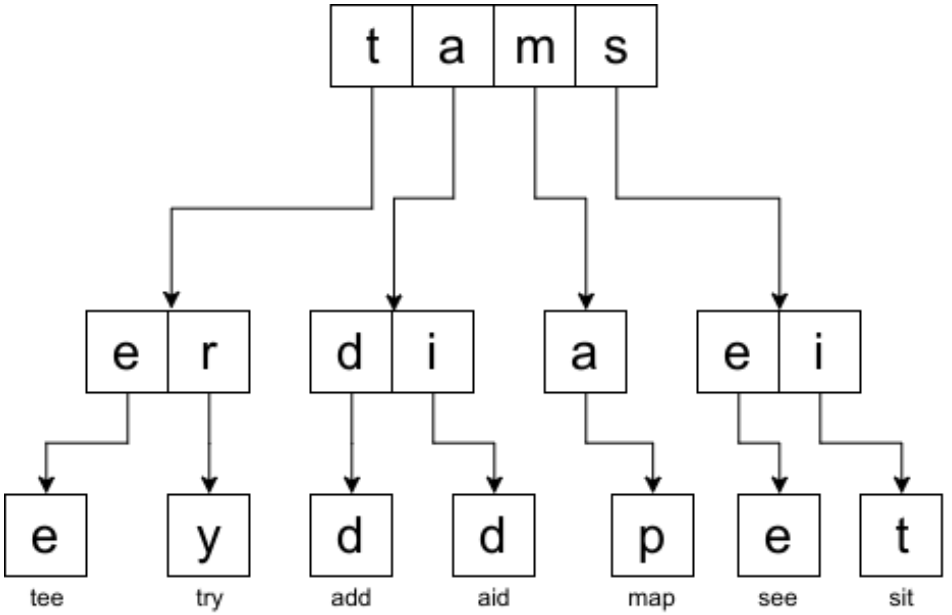
Vectors: An Introduction

- Tries:
  - tee, try, add, aid, map, see, sit



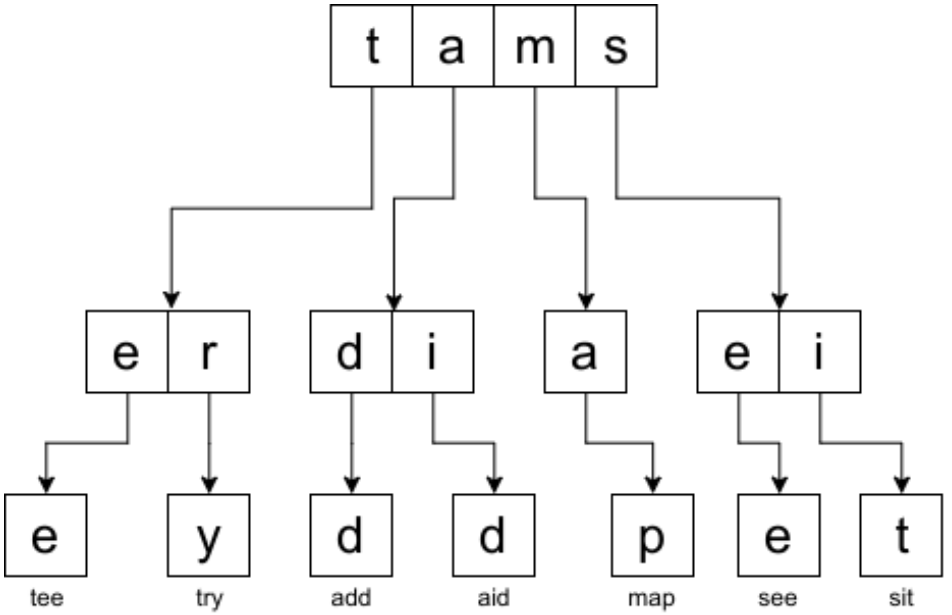
Vectors: An Introduction

- Tries:
  - Assume we want to lookup “add”



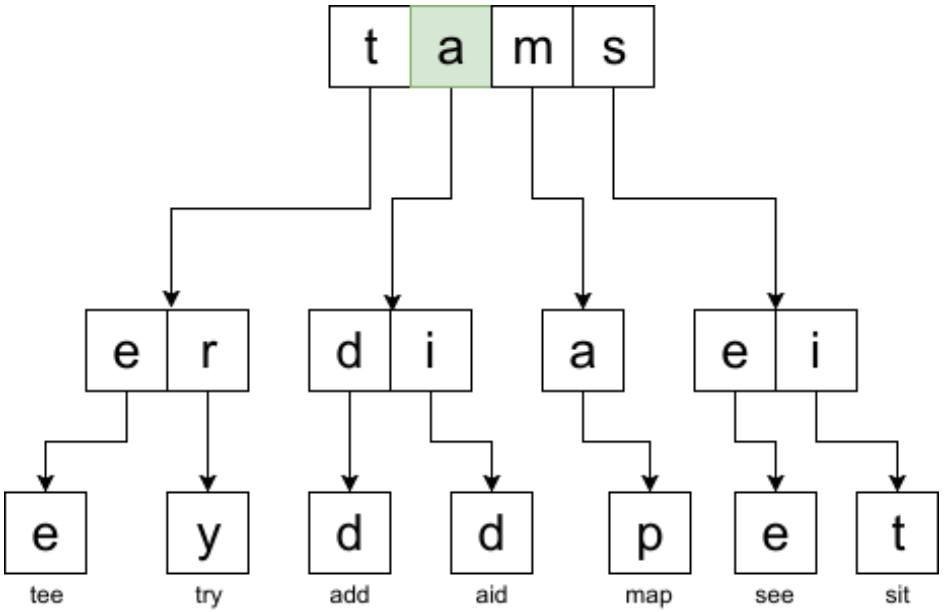
Vectors: An Introduction

- Tries:
  - Assume we want to lookup “add”
    - Split: a - d - d



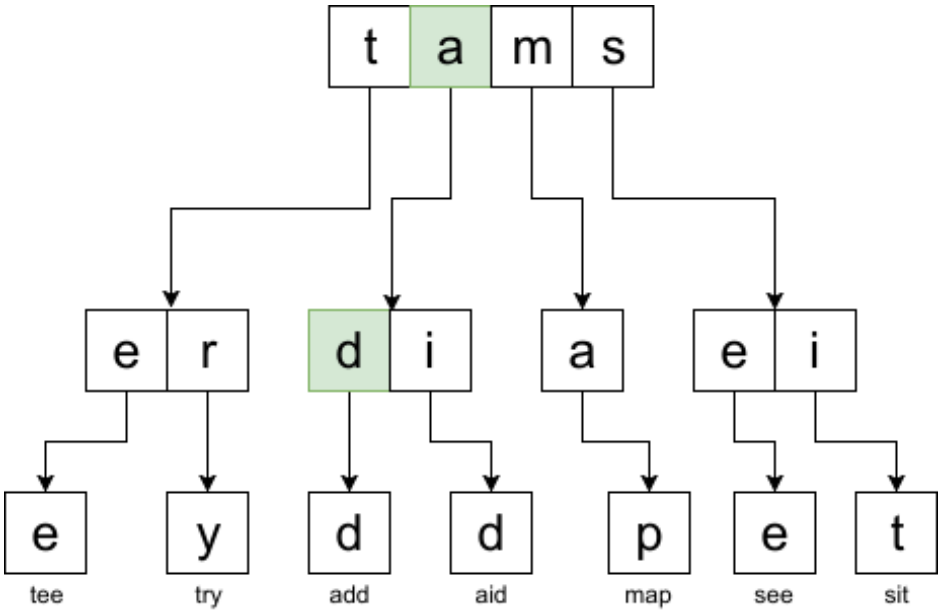
Vectors: An Introduction

- Tries:
  - Assume we want to lookup “add”
    - Split: a - d - d



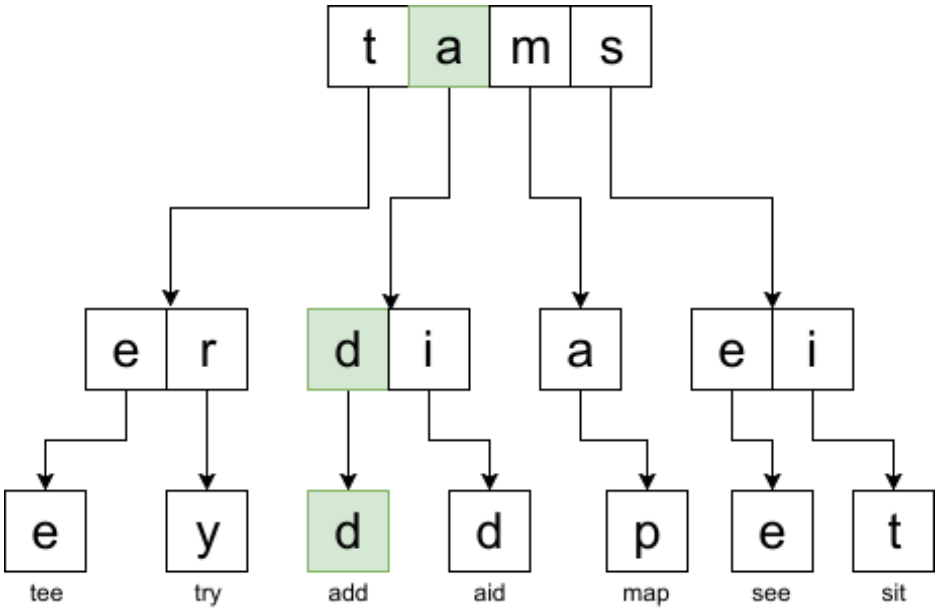
Vectors: An Introduction

- Tries:
  - Assume we want to lookup “add”
    - Split: a - d - d



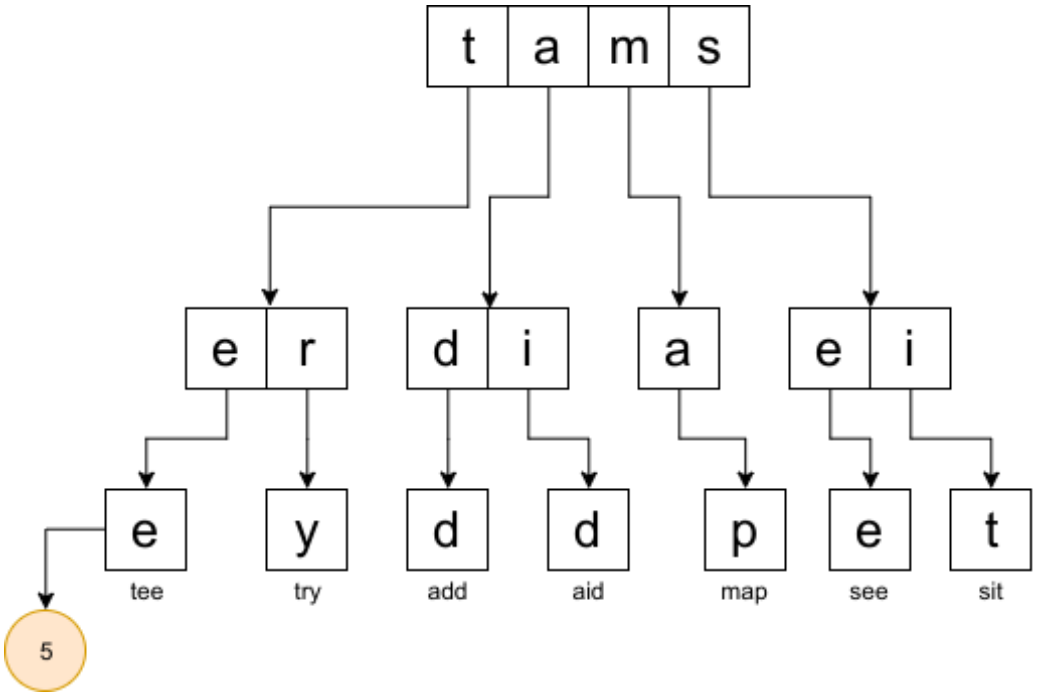
Vectors: An Introduction

- Tries:
  - Assume we want to lookup “add”
    - Split: a - d - d



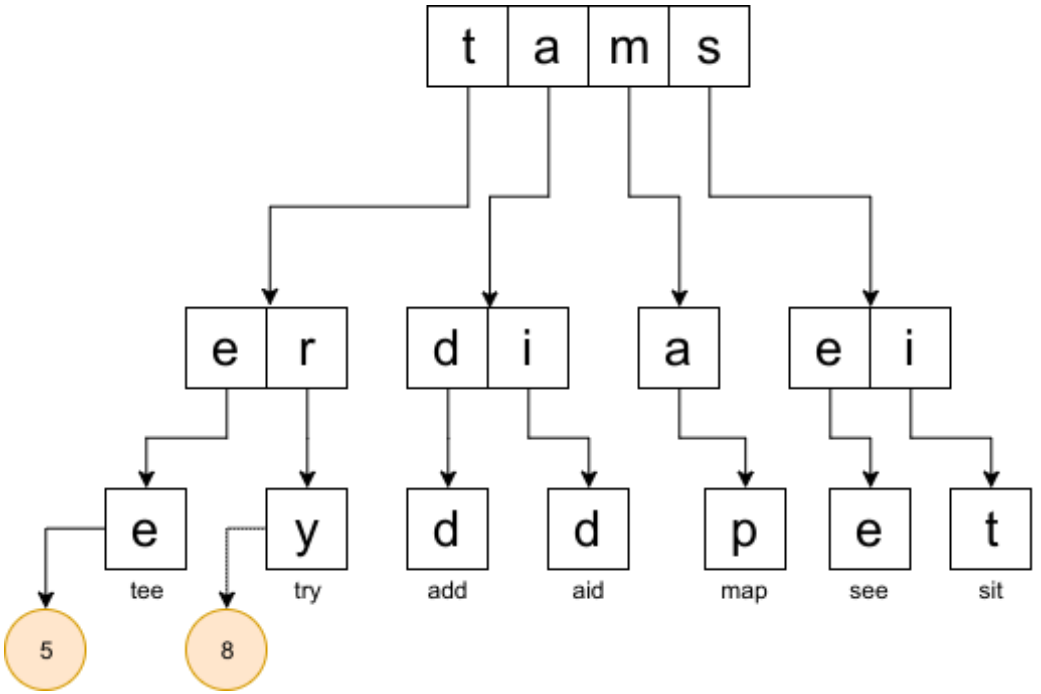
Vectors: An Introduction

- Trie Maps:
  - Associate value with leaf node



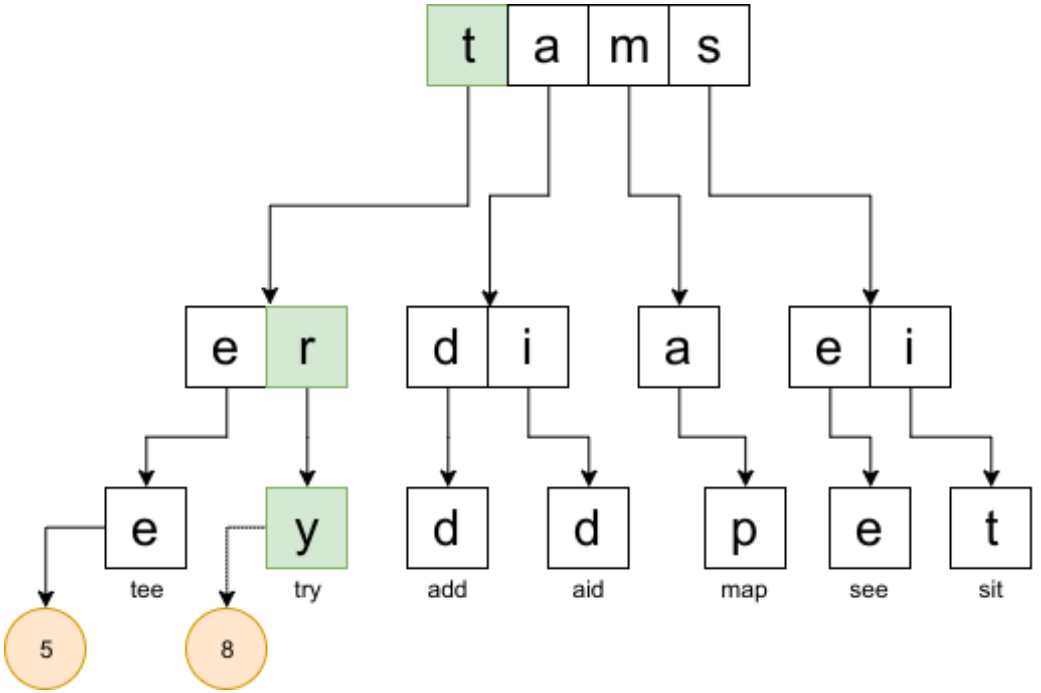
Vectors: An Introduction

- Trie Maps:
  - Associate value with leaf node



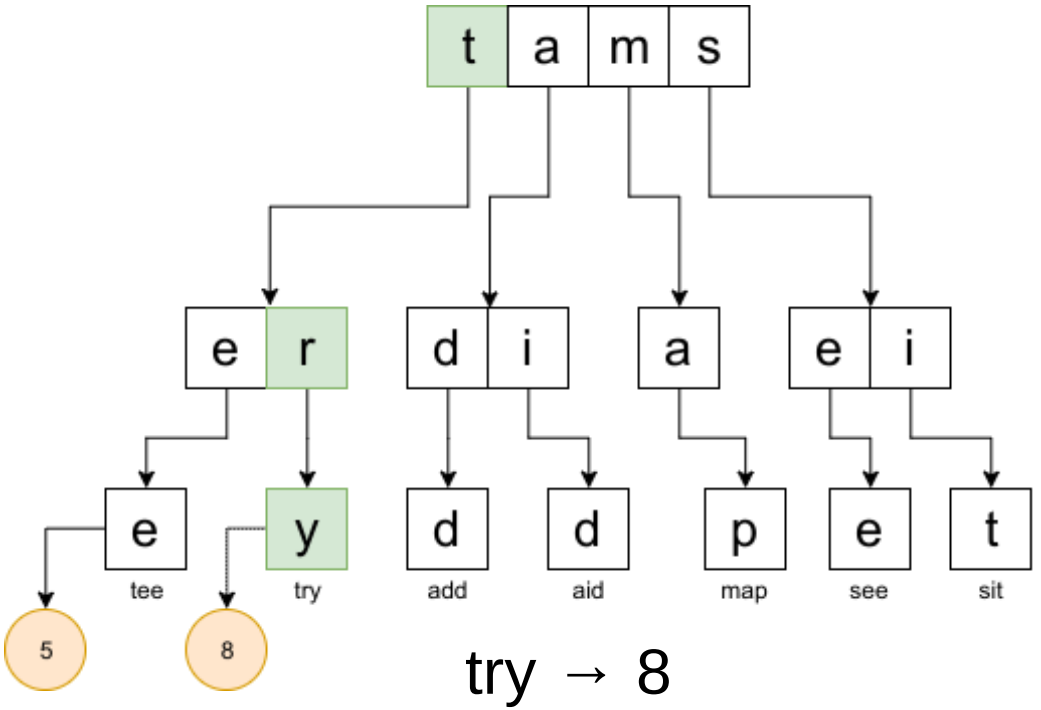
Vectors: An Introduction

- Trie Maps:
  - Associate value with leaf node
  - Look-up: “try”



Vectors: An Introduction

- Trie Maps:
  - Associate value with leaf node
  - Look-up: “try”



try → 8

Vectors: An Introduction

- Hash Array Mapped Tries:
  - Tries with hash ranges

Vectors: An Introduction

- Hash Array Mapped Tries:
  - Tries with hash ranges
  - Hash-Maps!

Vectors: An Introduction

- Hash Array Mapped Tries:
  - Tries with hash ranges
  - Hash-Maps!
- Vectors:
  - HAMTs with ranges of integers

Vectors: An Introduction

- Indexing a trie:

Vectors: An Introduction

- Indexing a trie:
    1. Fixed branching factors (power of 2):

M = 2^b

Vectors: An Introduction

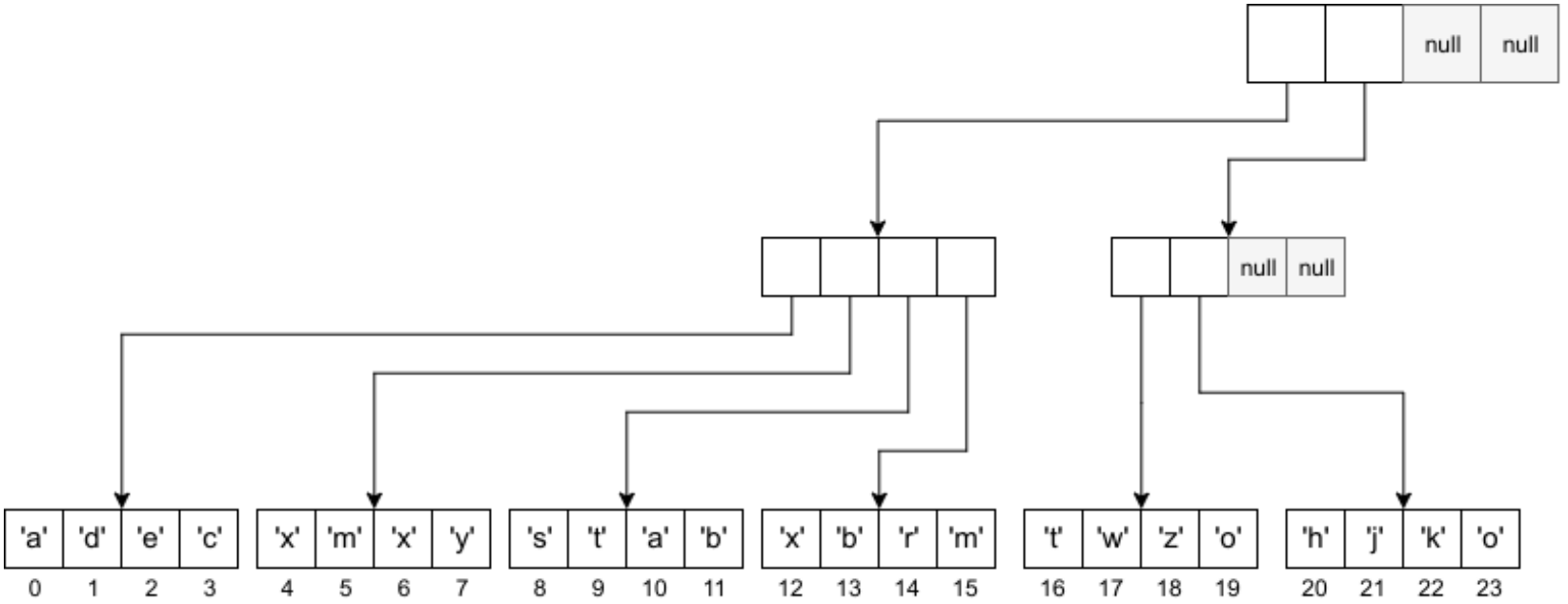
- Indexing a trie:
  1. Fixed branching factors (power of 2):

     M = 2^b

  2. Indexes are binary => Trie is bit-mapped

How to vector

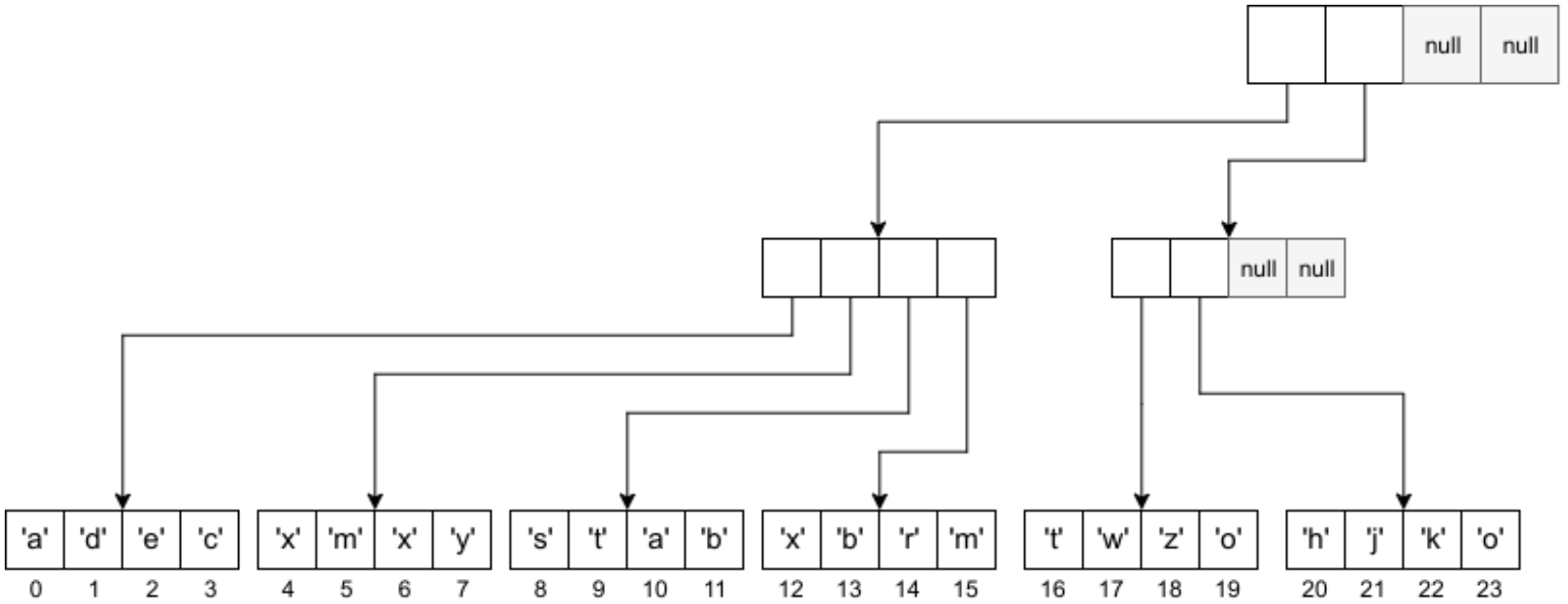
Branching factor: 4
Height: 2



How to vector

Branching factor: 4
Height: 2

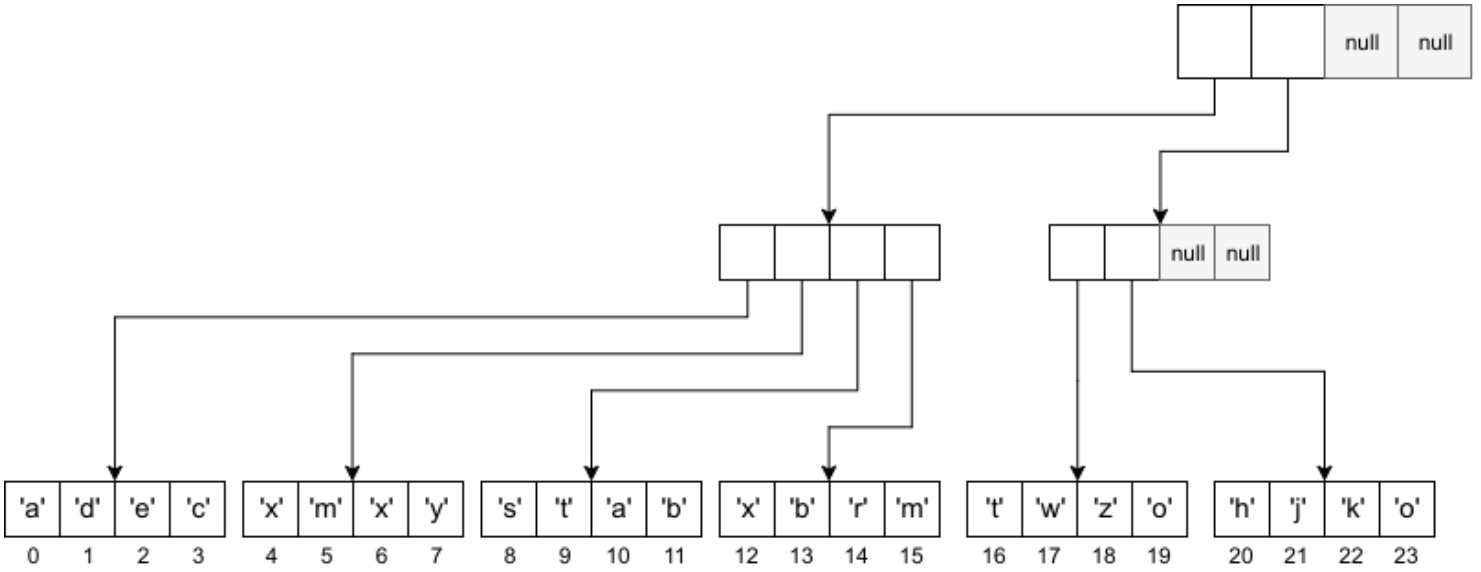
Look-up: 17



How to vector

Branching factor: 4
Height: 2

Look-up: 17

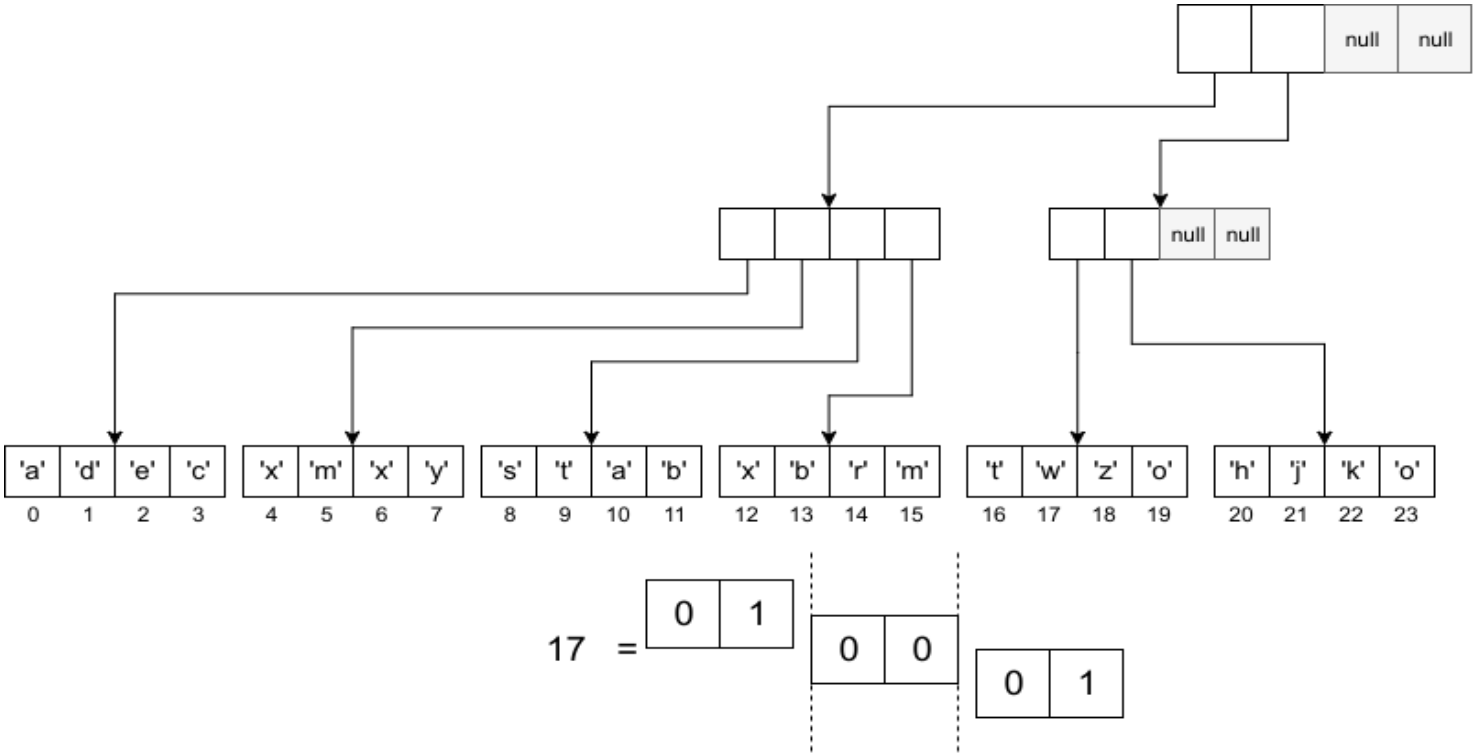


17 = | 0 | 1 | 0 | 0 | 0 | 1 |

How to vector

Branching factor: 4
Height: 2

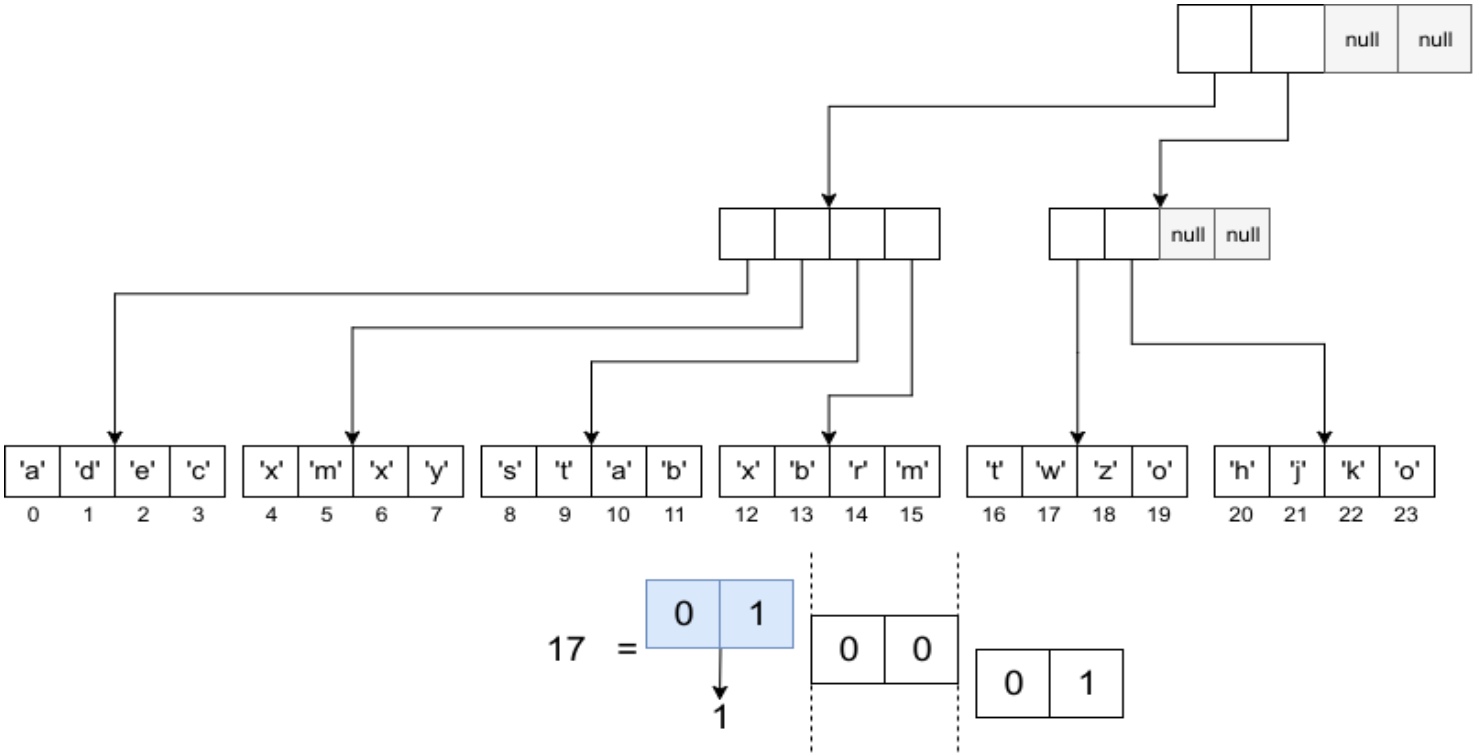
Look-up: 17



How to vector

Branching factor: 4
Height: 2

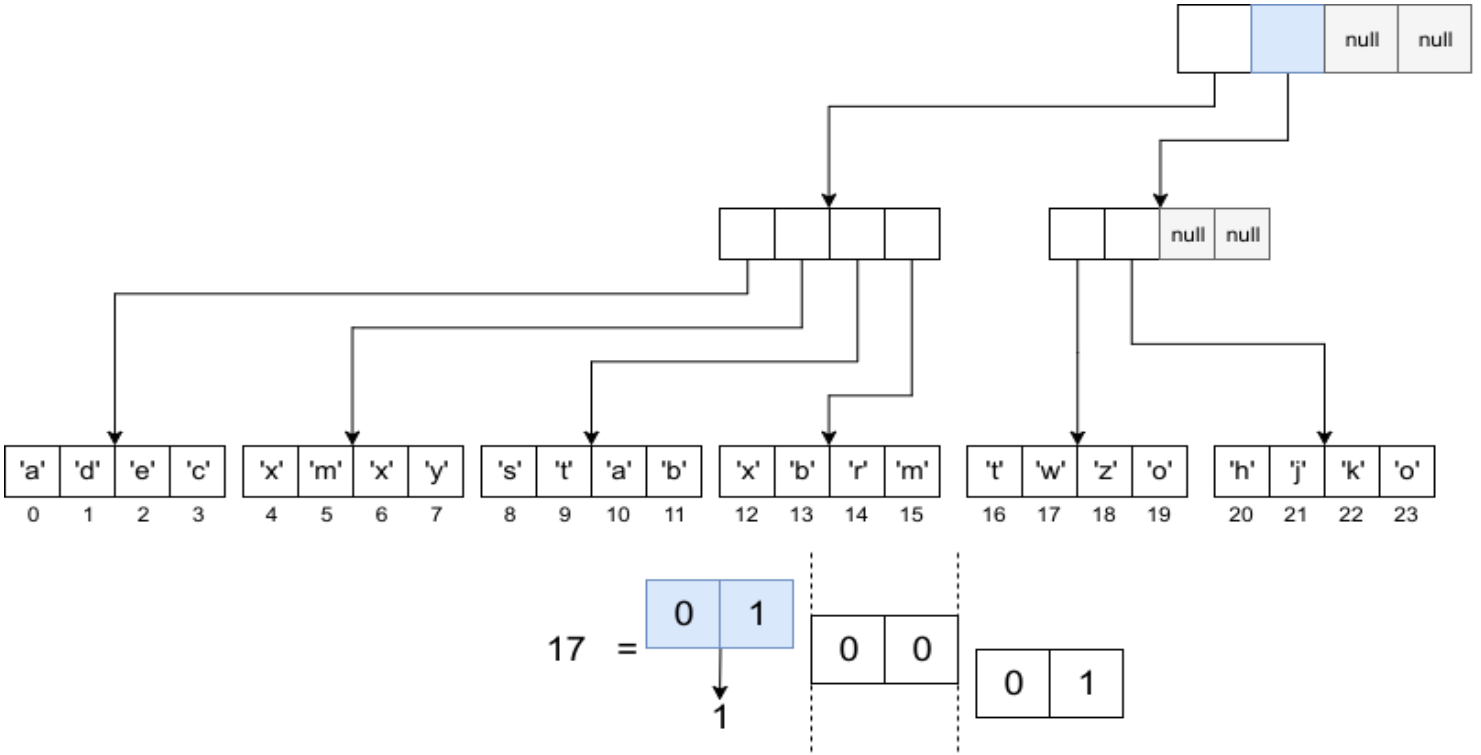
Look-up: 17



How to vector

Branching factor: 4
Height: 2

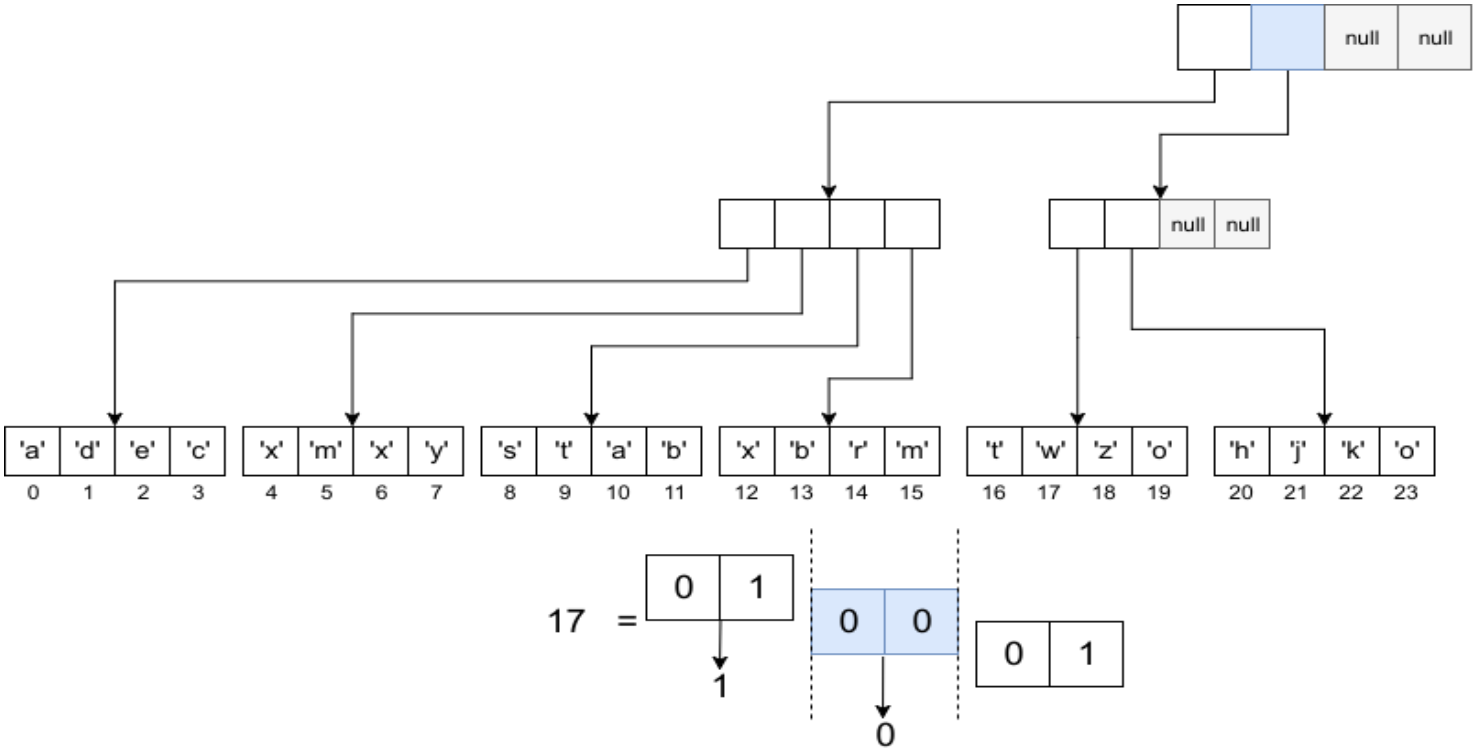
Look-up: 17



How to vector

Branching factor: 4
Height: 2

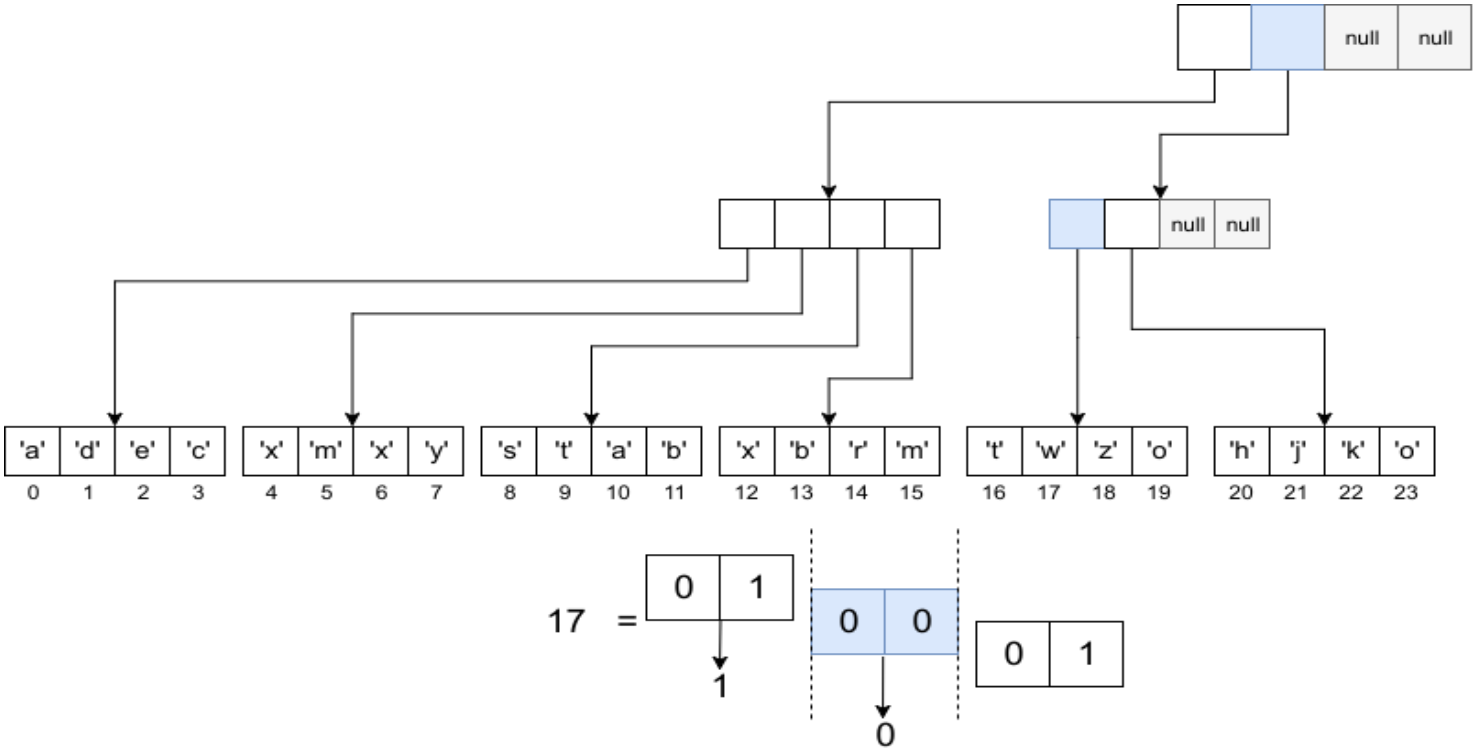
Look-up: 17



How to vector

Branching factor: 4
Height: 2

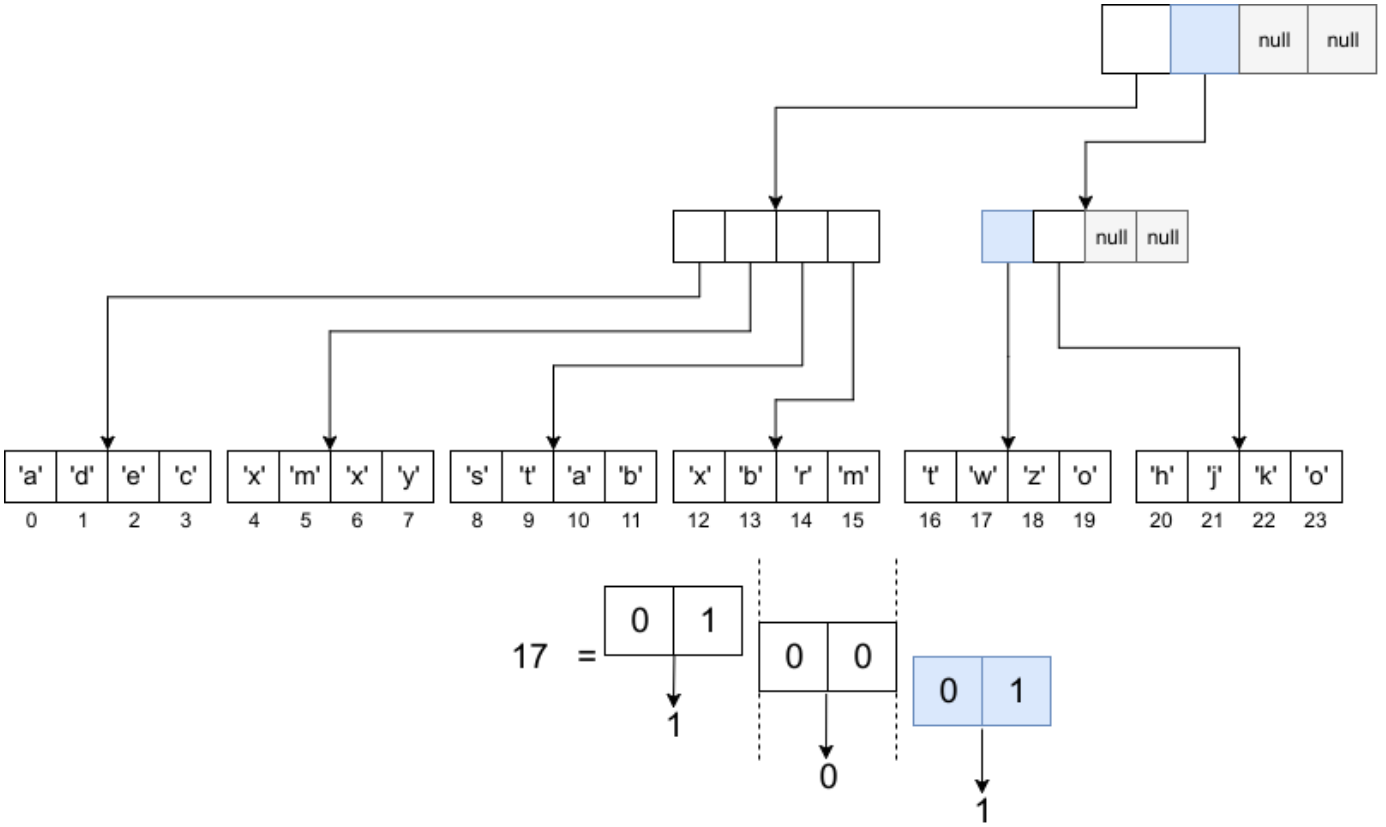
Look-up: 17



How to vector

Branching factor: 4
Height: 2

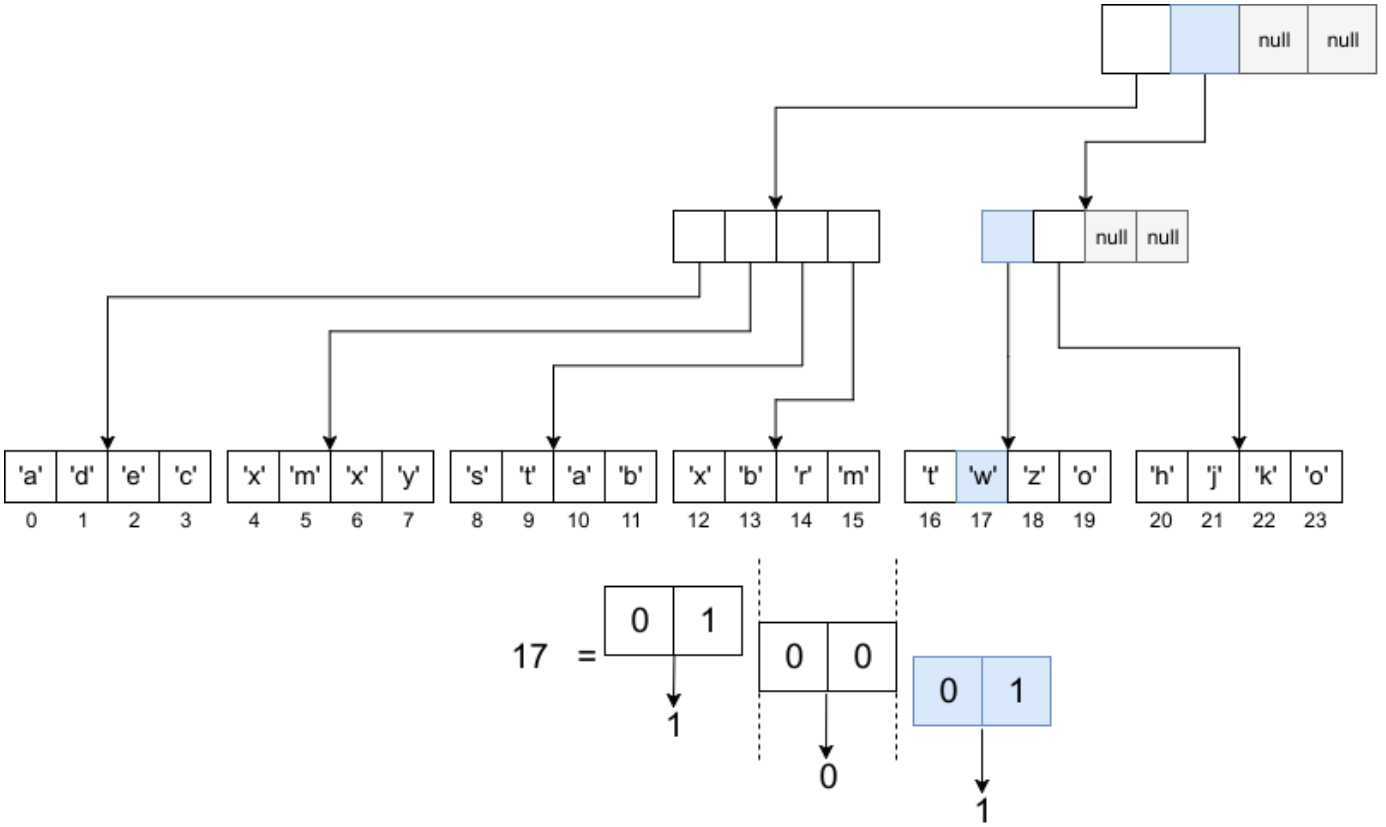
Look-up: 17



How to vector

Branching factor: 4
Height: 2

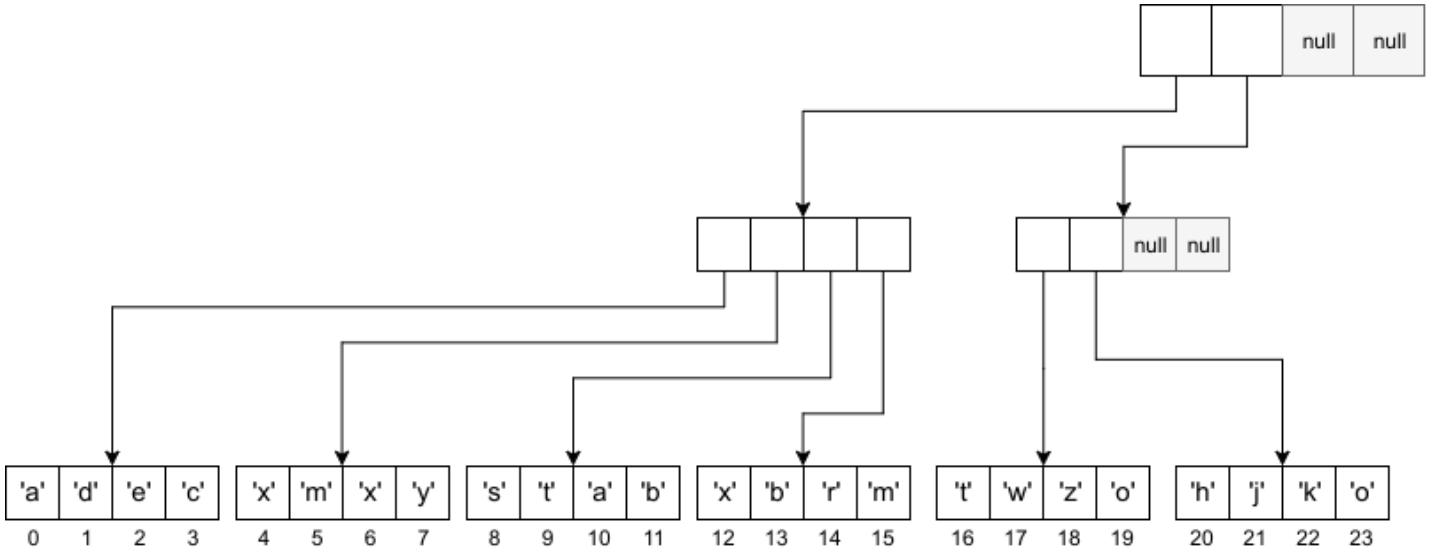
Look-up: 17



How to vector

Branching factor: 4
Height: 2

Append: ‘n’ → index 24

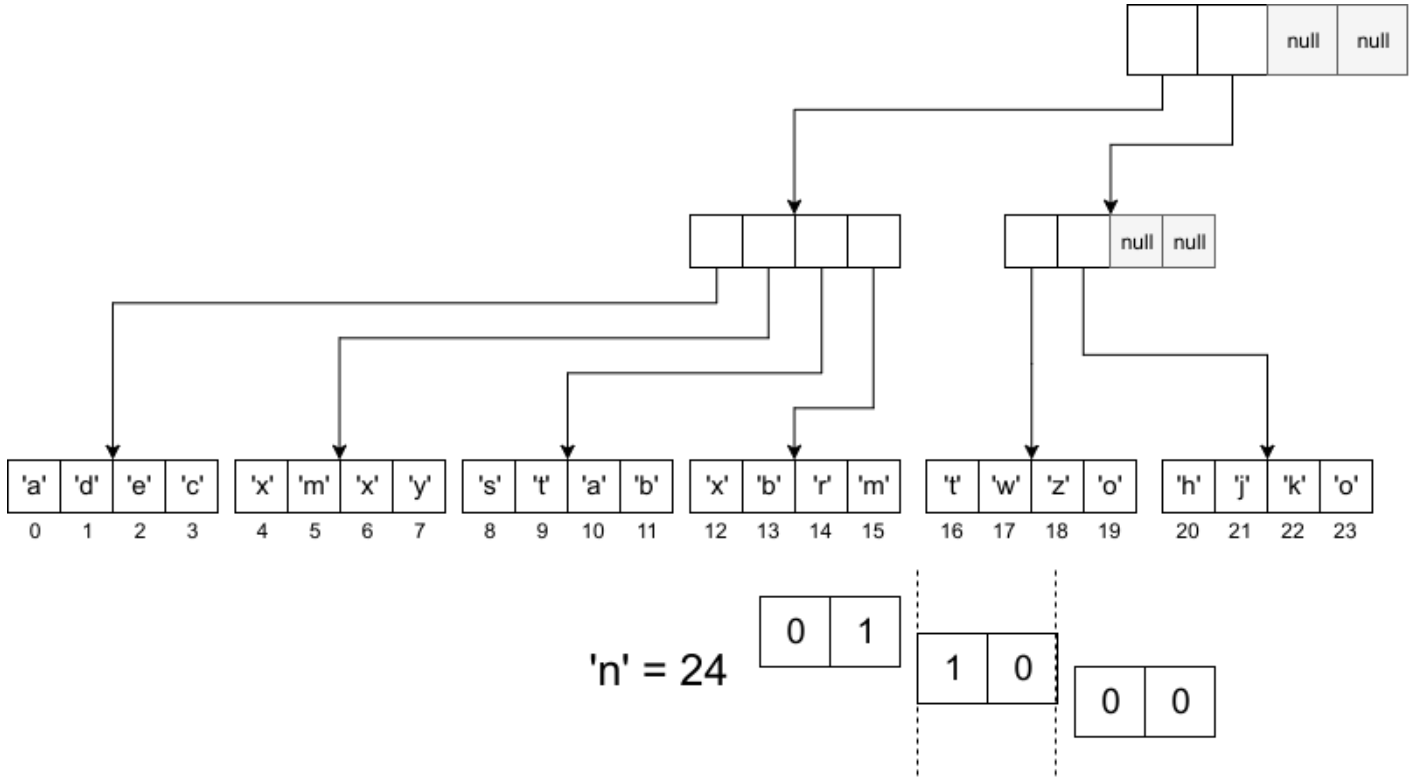


'n' = 24

How to vector

Branching factor: 4
Height: 2

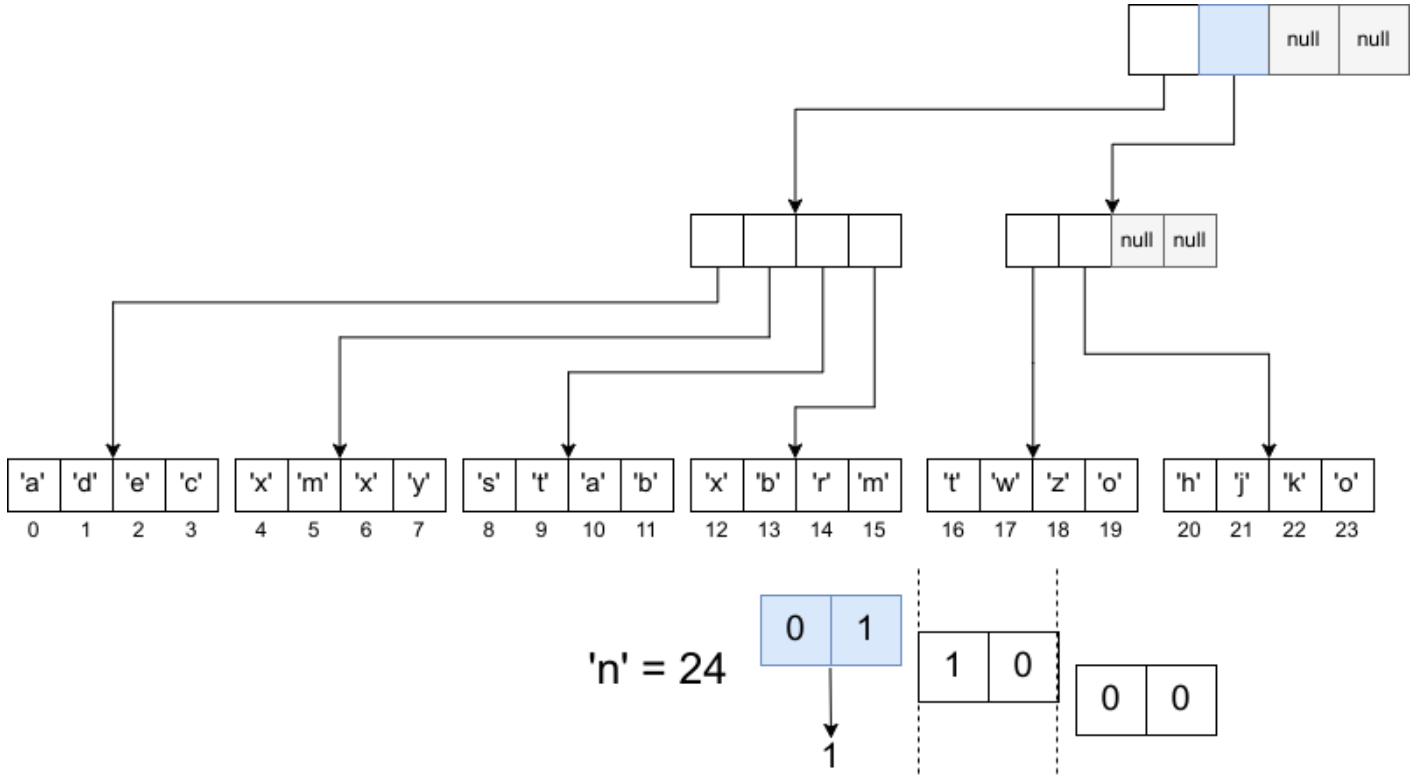
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

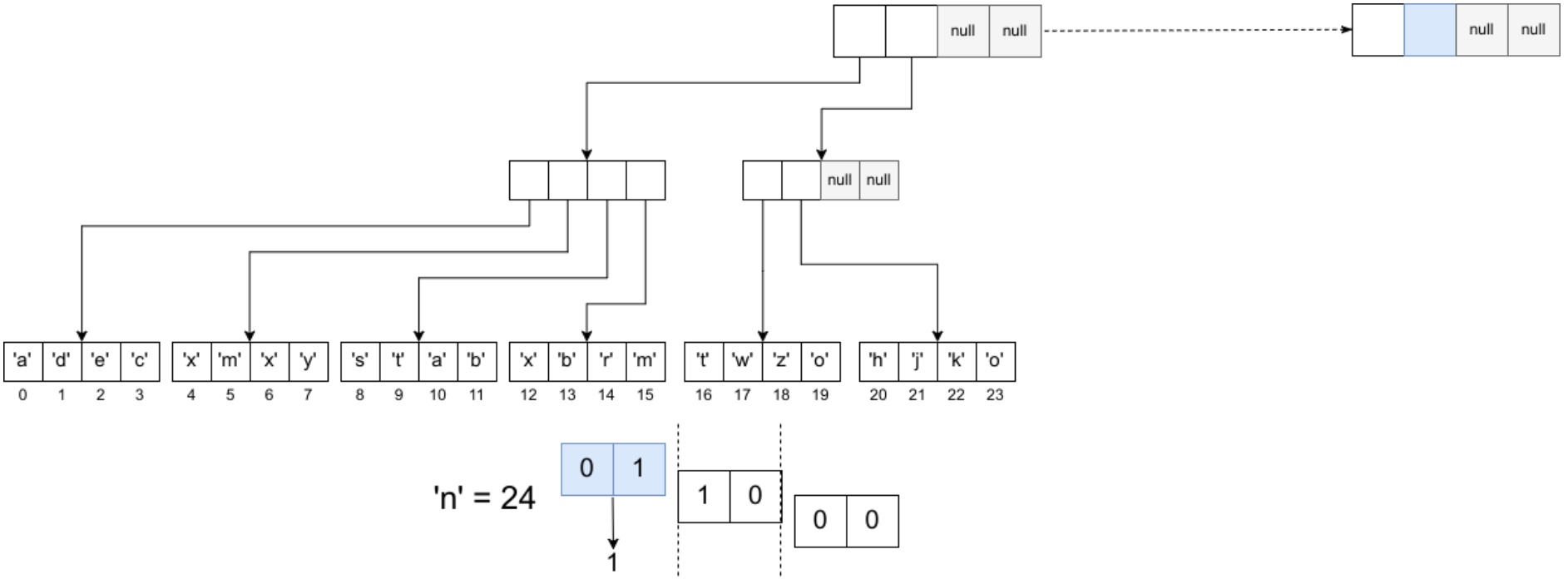
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

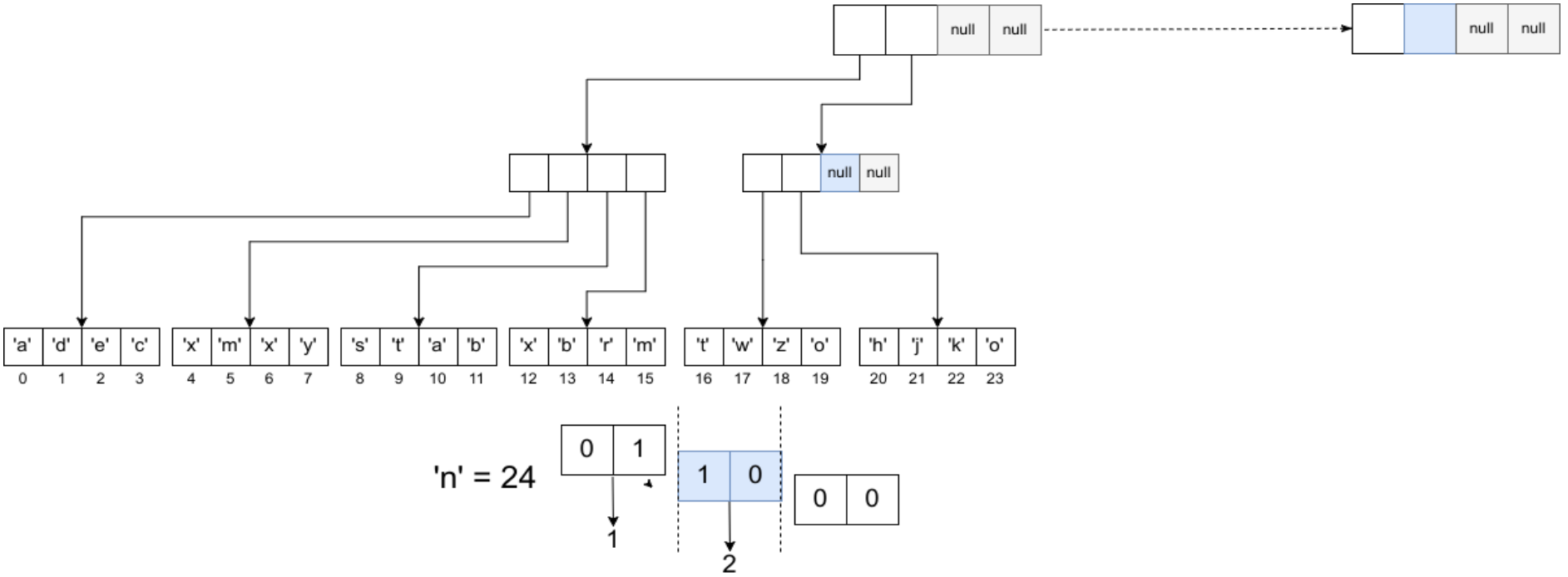
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

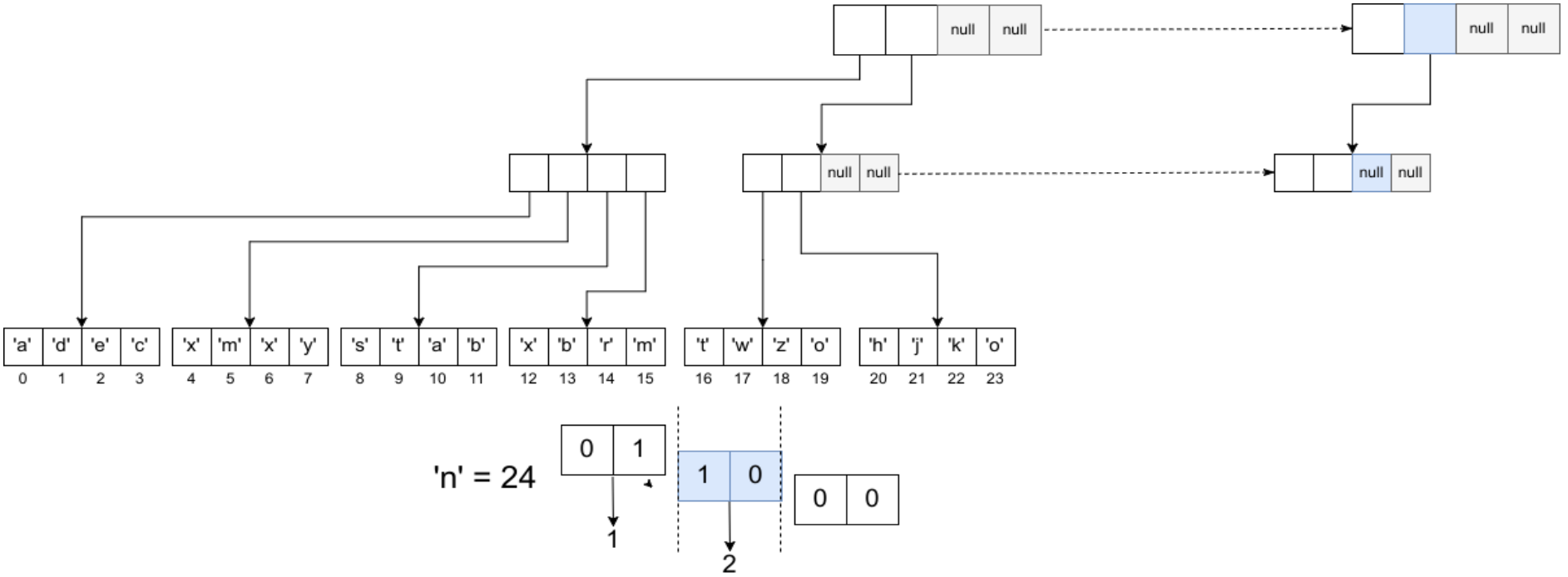
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

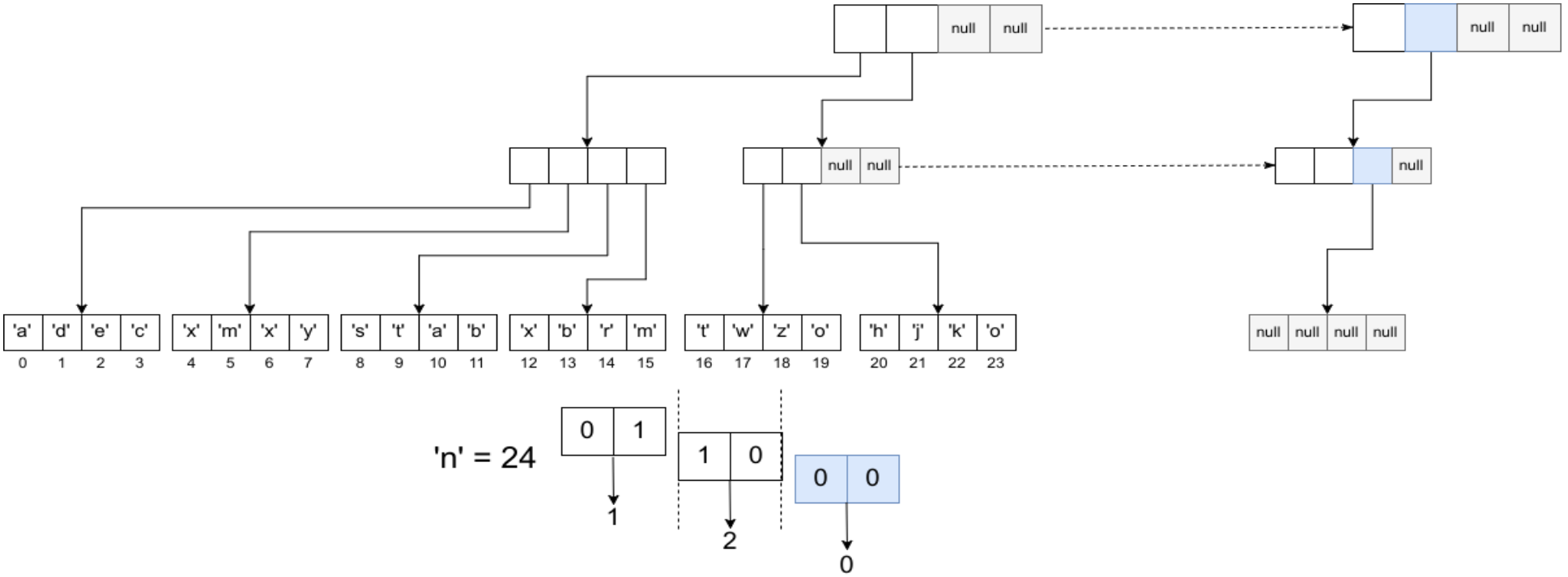
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

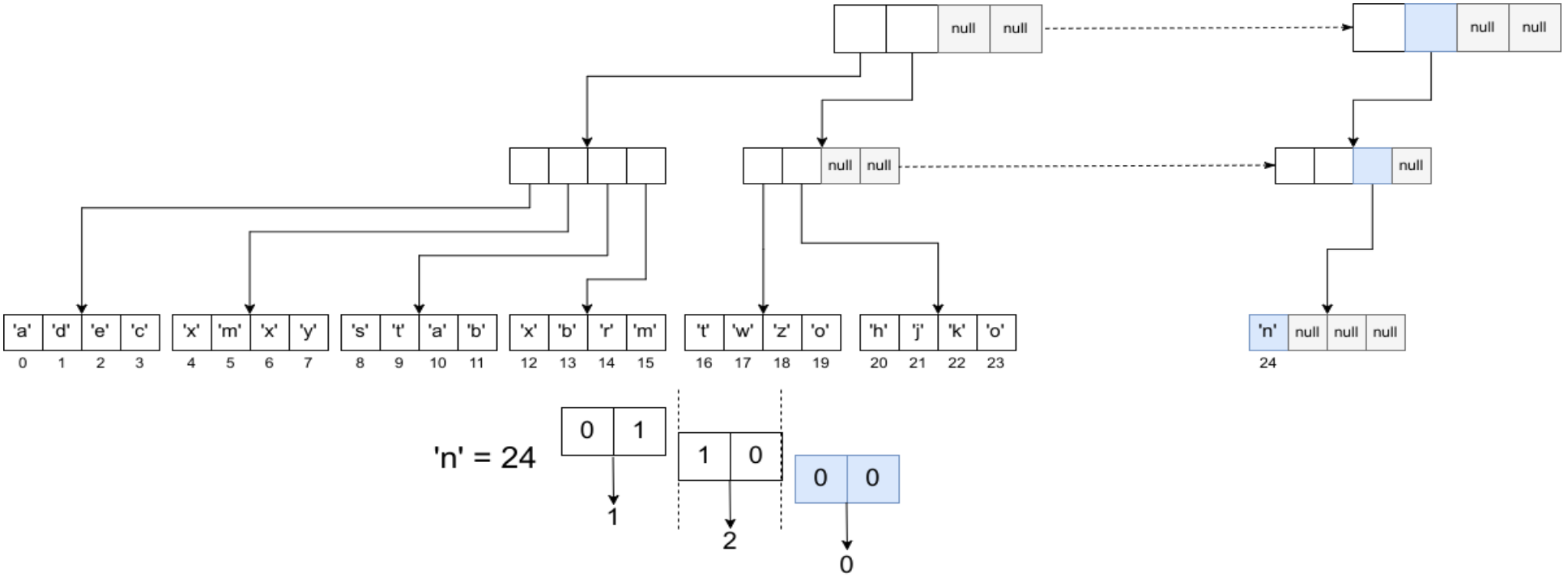
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

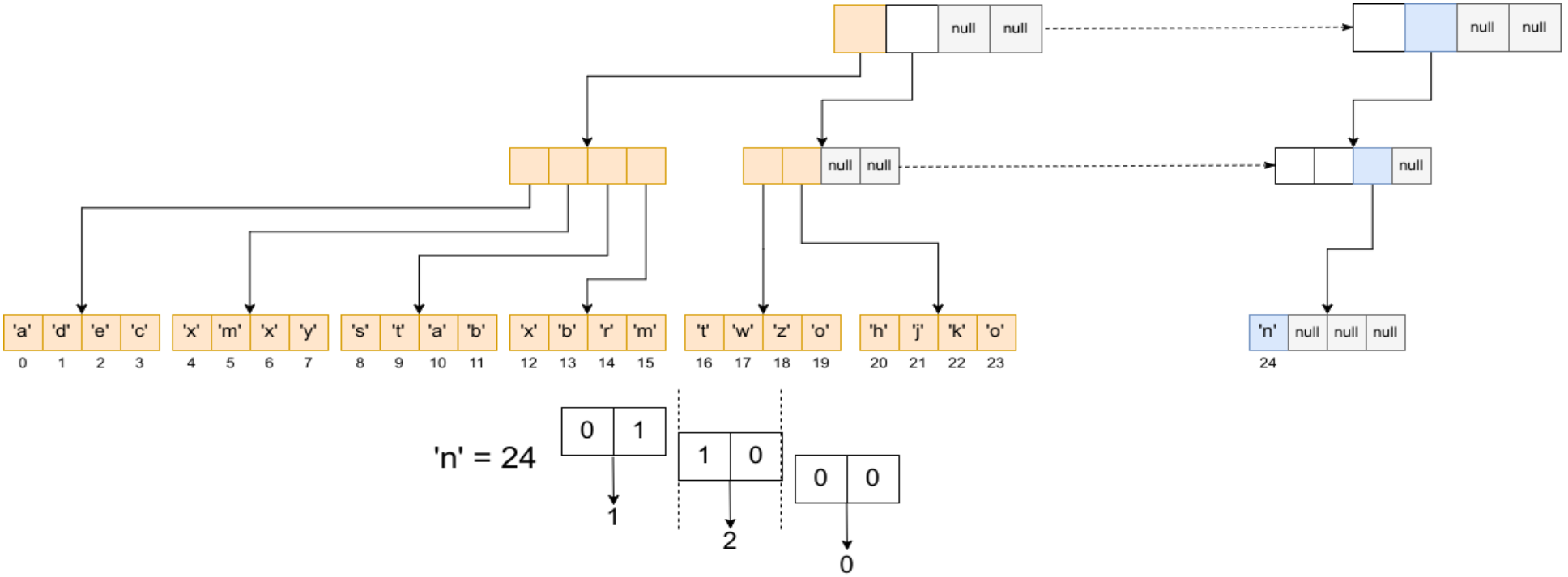
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

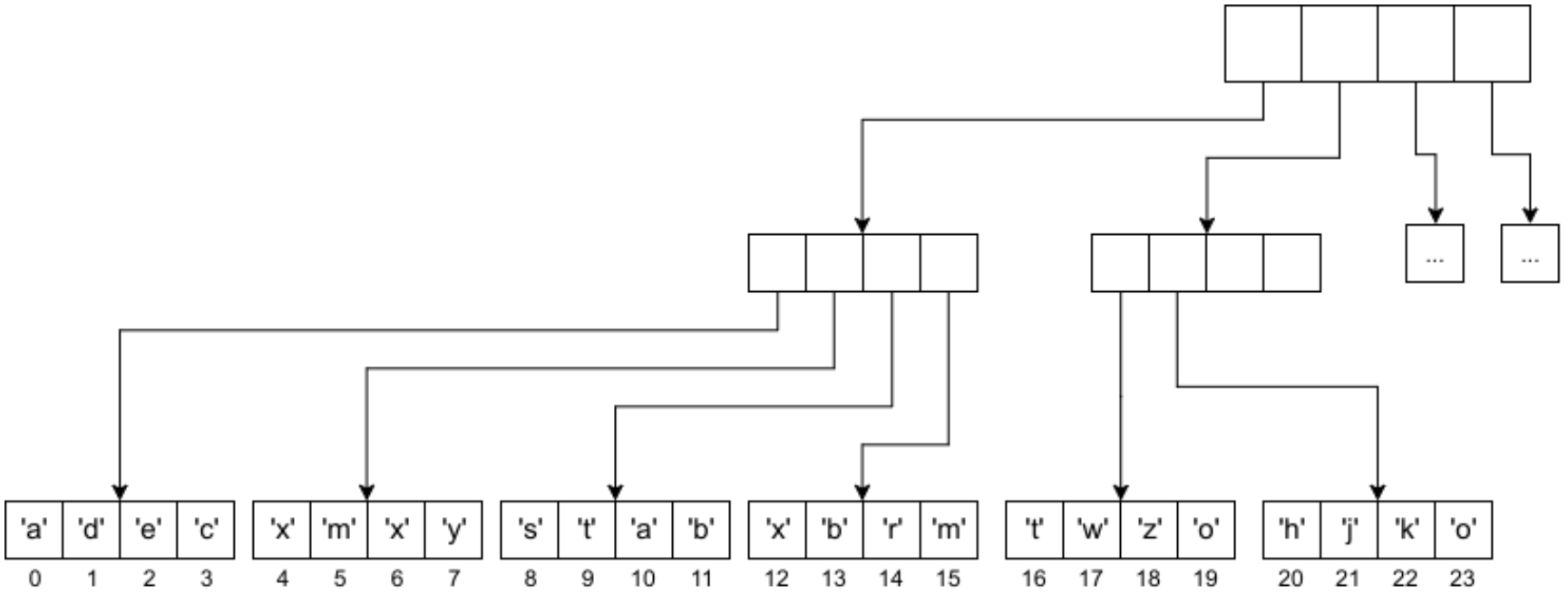
Append: ‘n’ → index 24



How to vector

Branching factor: 4
Height: 2

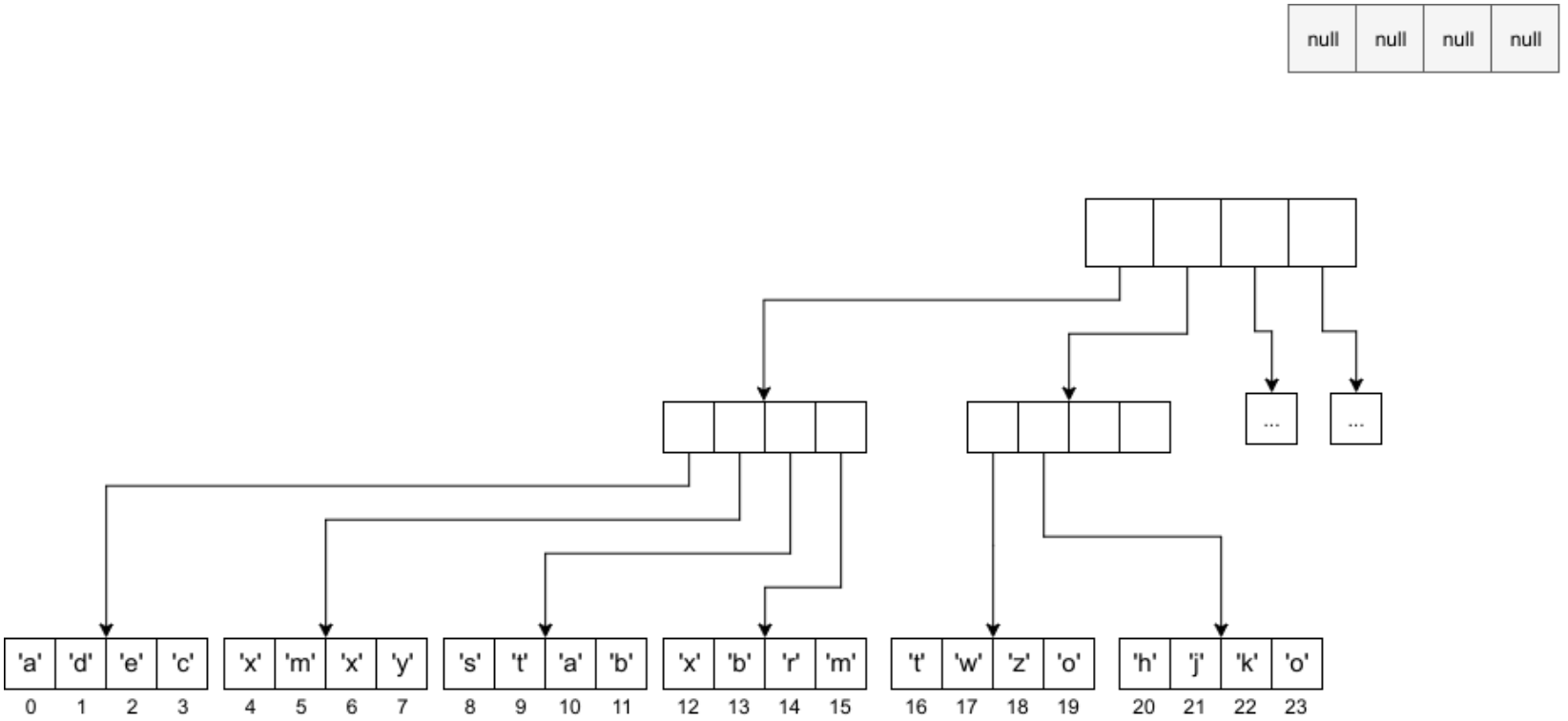
Append when full



How to vector

Branching factor: 4
Height: **3**

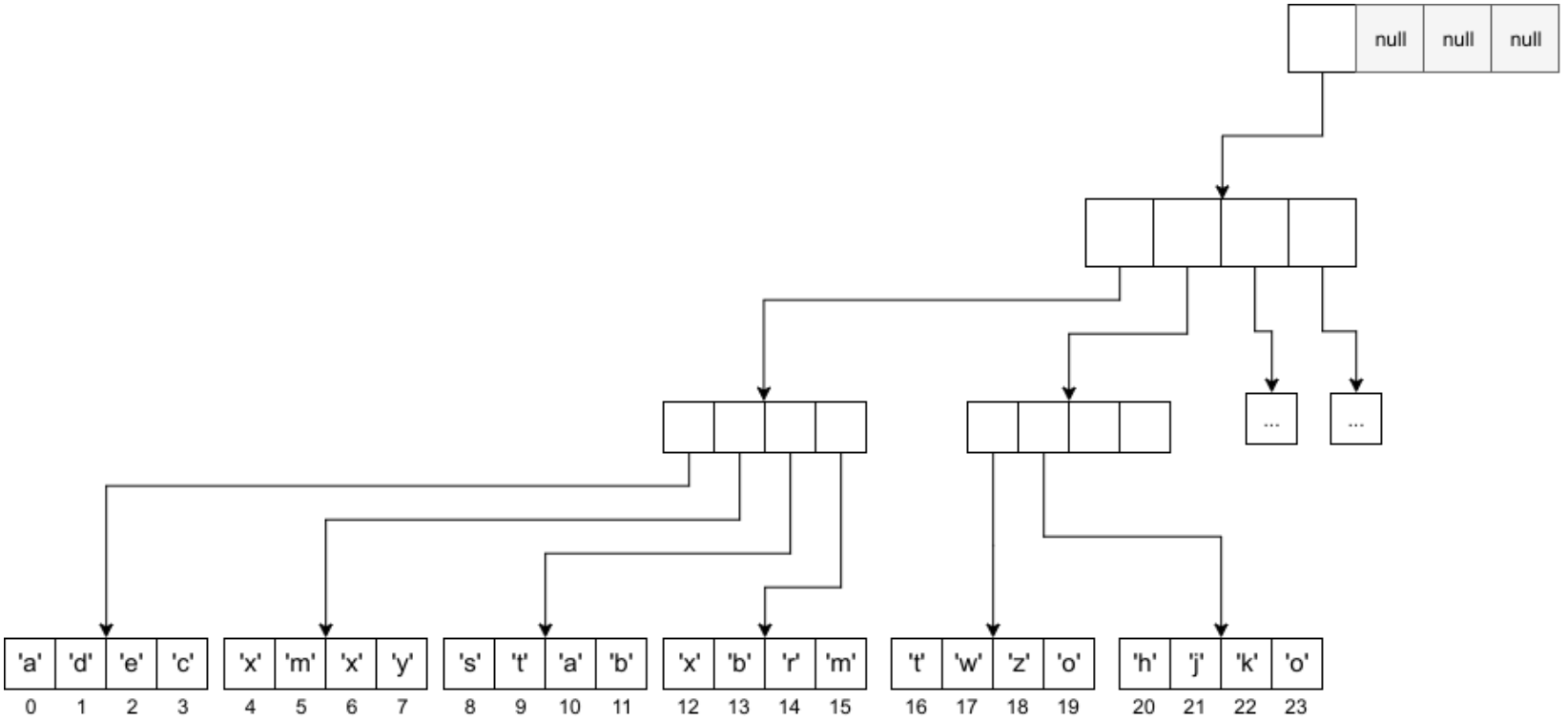
Append when full



How to vector

Branching factor: 4
Height: **3**

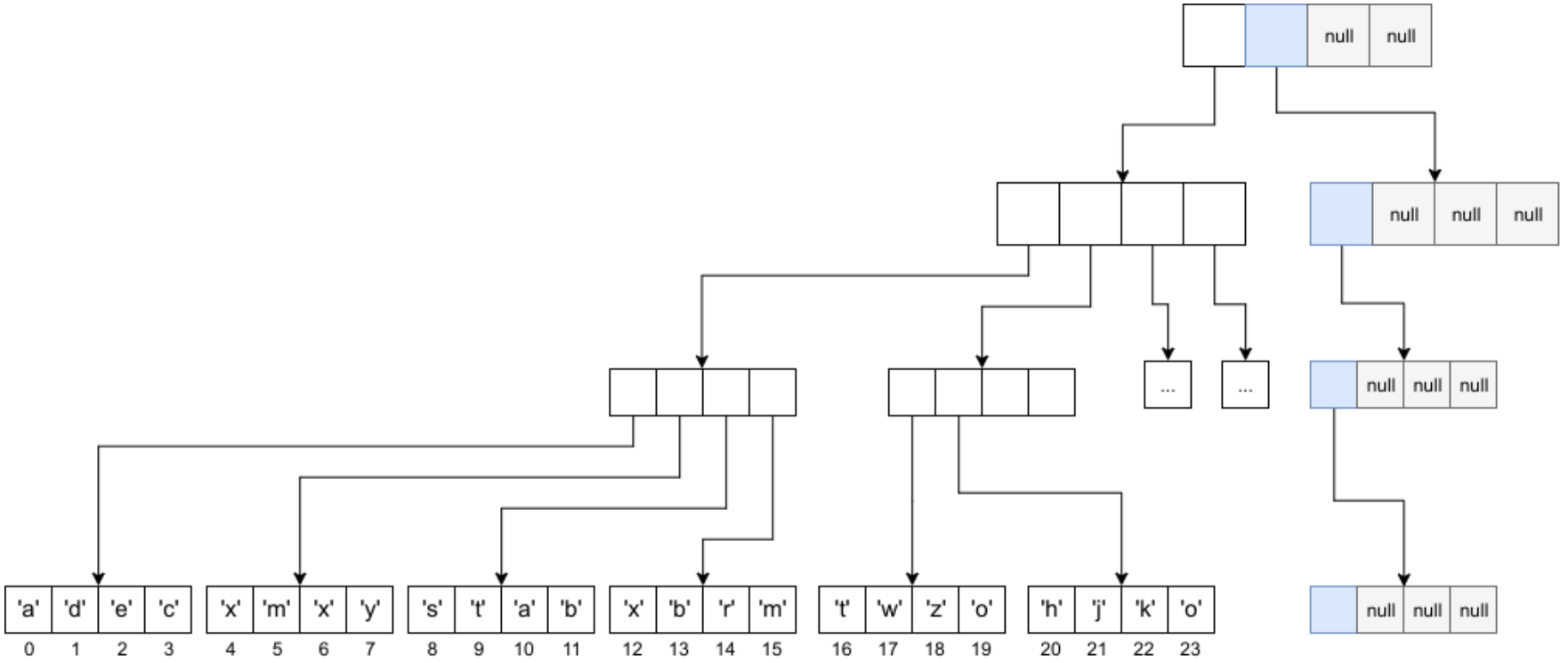
Append when full



How to vector

Branching factor: 4
Height: **3**

Append when full



How to vector (properly this time)

- You don’t implement binary nonesense

How to vector (properly this time)

- You don’t implement binary nonesense
- Calculate with bitwise operations

How to vector (properly this time)

- You don’t implement binary nonesense
- Calculate with bitwise operations
- Index with formula:

  f (i, h) = (i >> shift) & (M – 1)

  shift = h * **b** (M = 2^**b**)

  M = branching factor

Vectors: In the wild

- Why log32?

Vectors: In the wild

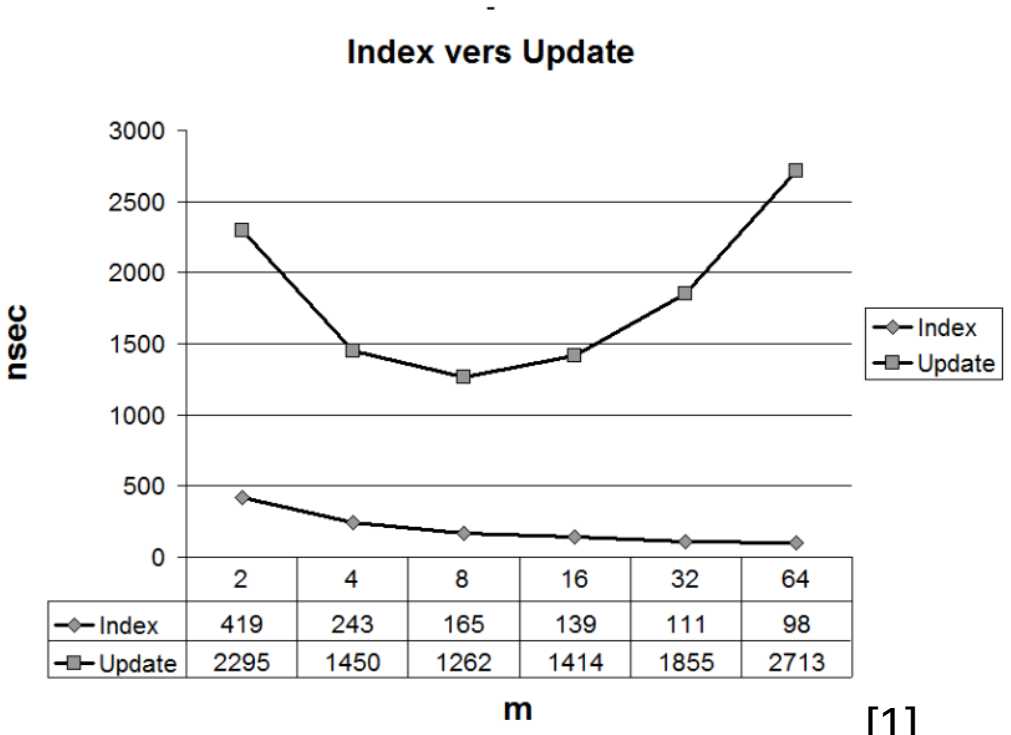
- Why log32?
  - Branching factor influences:

Vectors: In the wild

- Why log32?
  - Branching factor influences:
    - Look-up, iteration
    - Appension, update

Vectors: In the wild

- Why log32?
  - Branching factor influences:
    - Look-up, iteration
    - Appension, update



	2	4	8	16	32	64
Index	419	243	165	139	111	98
Update	2295	1450	1262	1414	1855	2713

[1]

Vectors: In the wild

- Scala Vector:
  - (on average) O(1) update
  - (on average) O(1) look-up

[1]

Vectors: In the wild

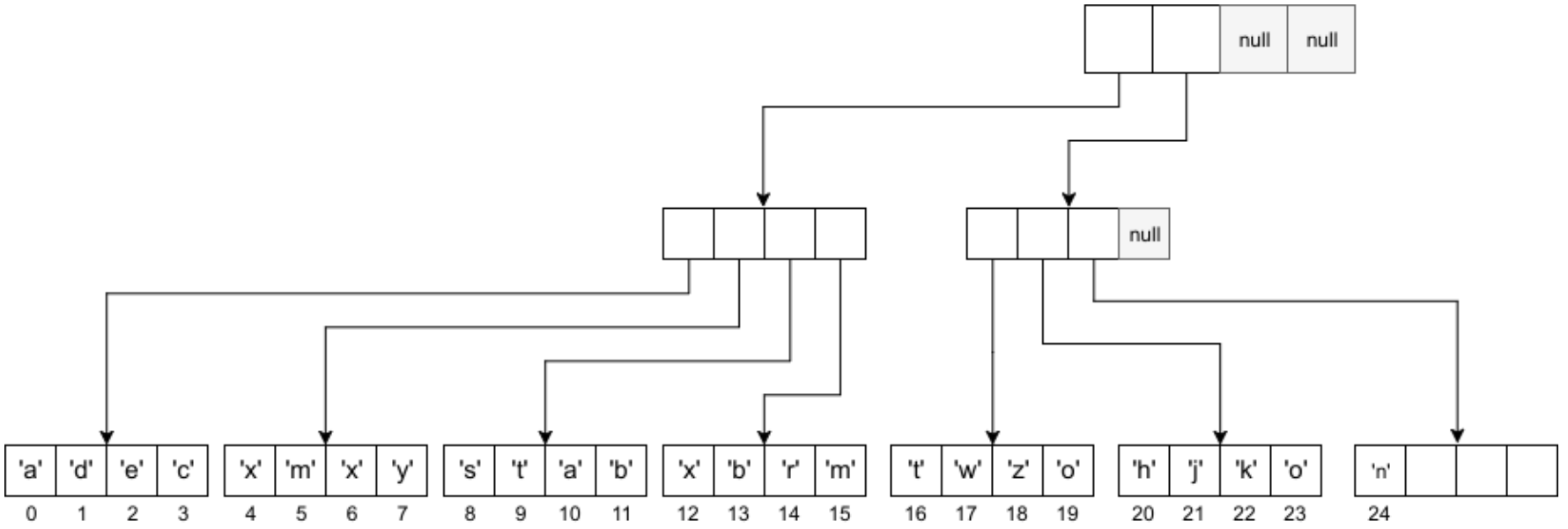
- Scala Vector:
  - (on average) O(1) update
  - (on average) O(1) look-up

- Clojure Vector:
  - O(1) appension
  - O(1) last
  - O(1) pop

[1]

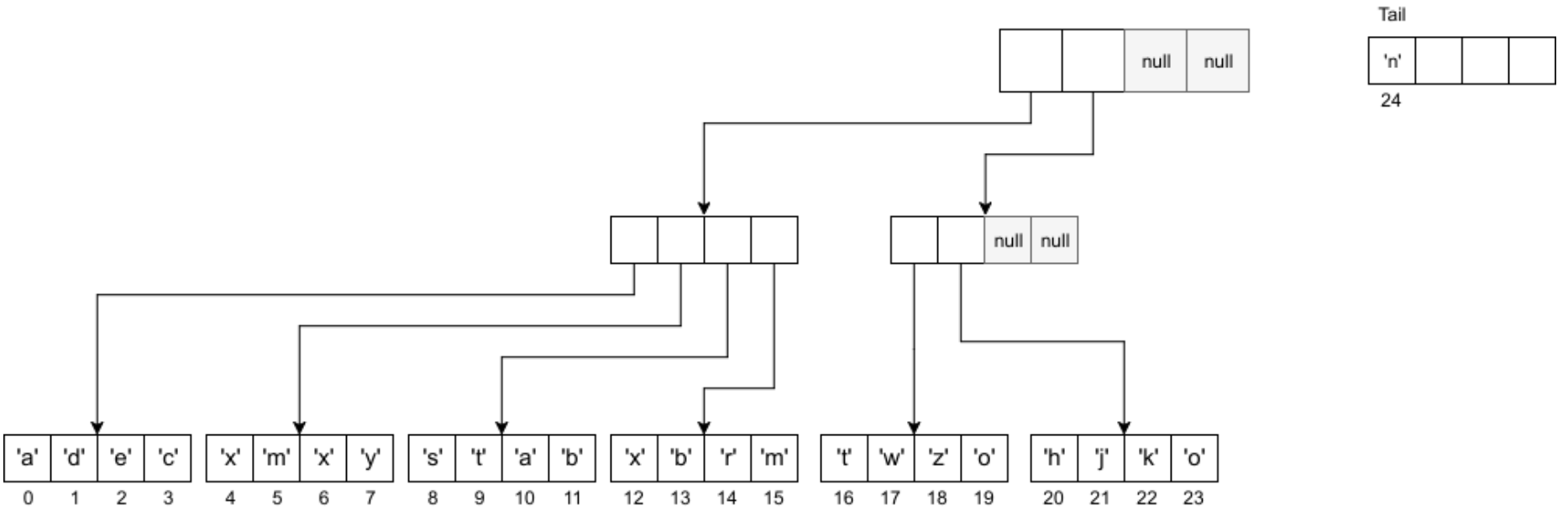
Clojure Vector

- Tail



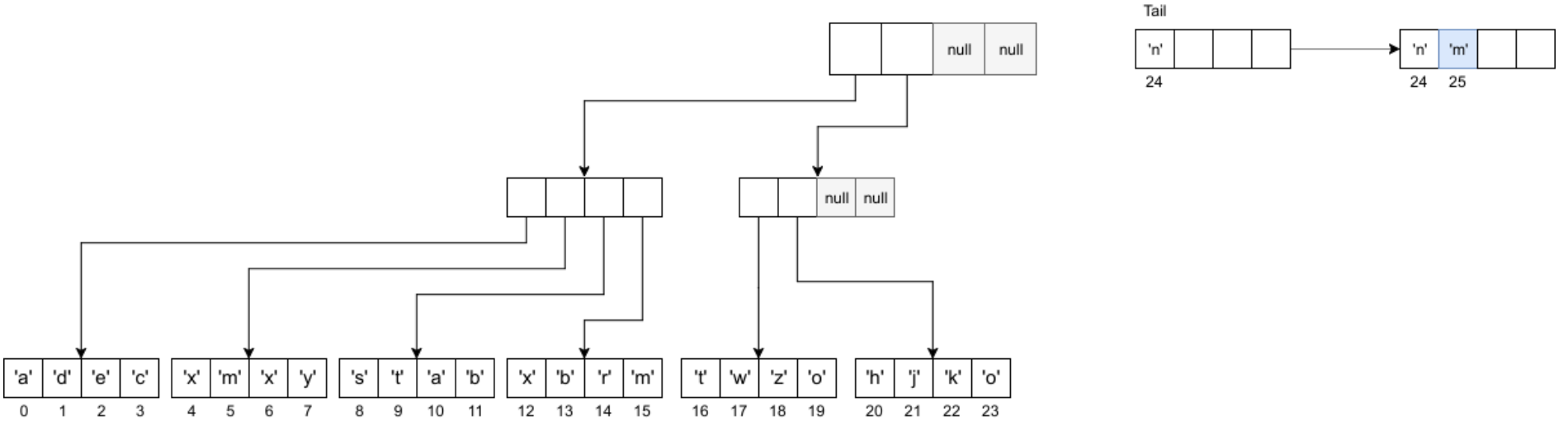
Clojure Vector

- Tail



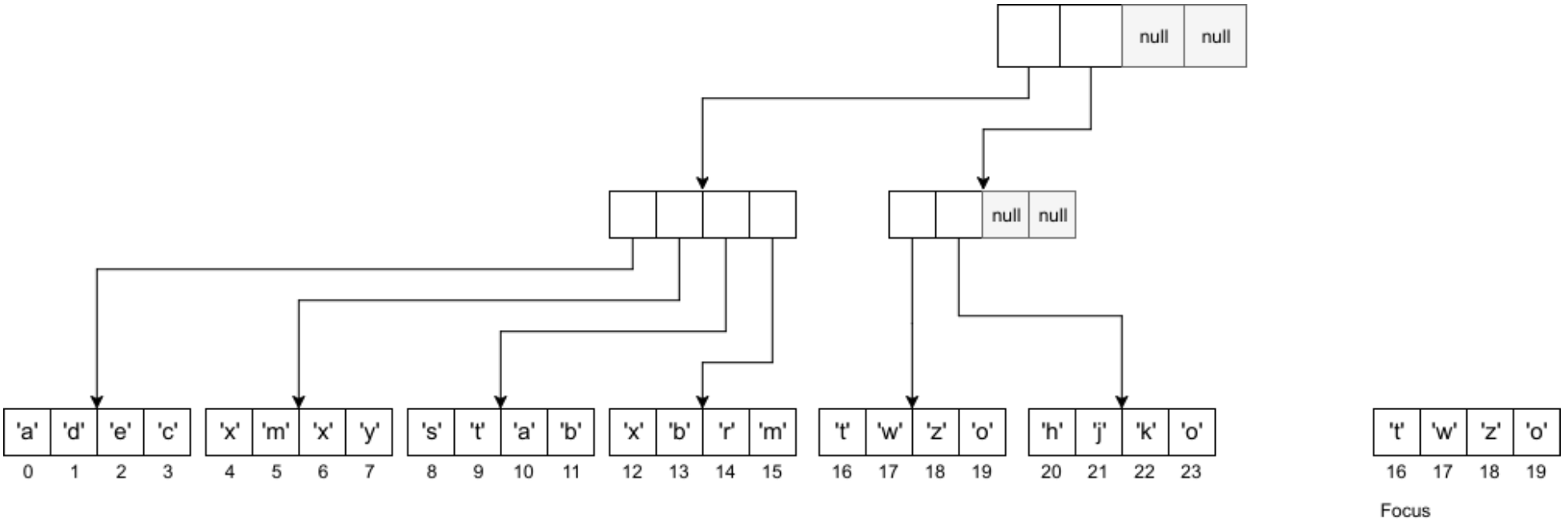
Clojure Vector

- Tail



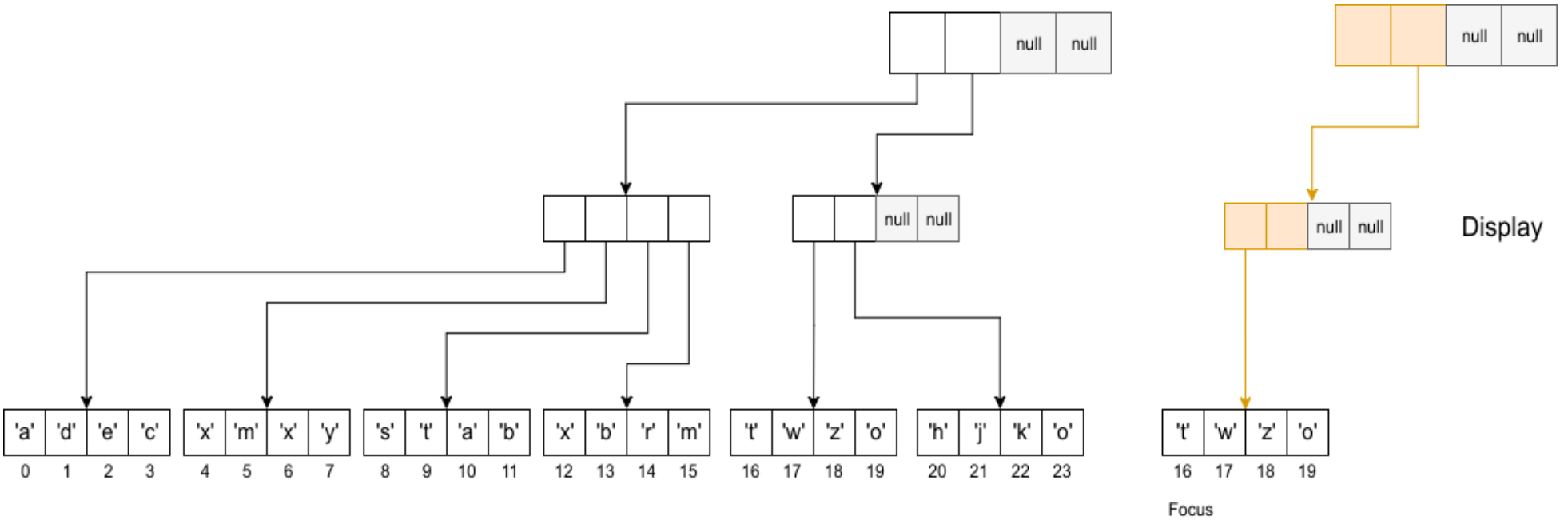
Scala Vector

- Focus



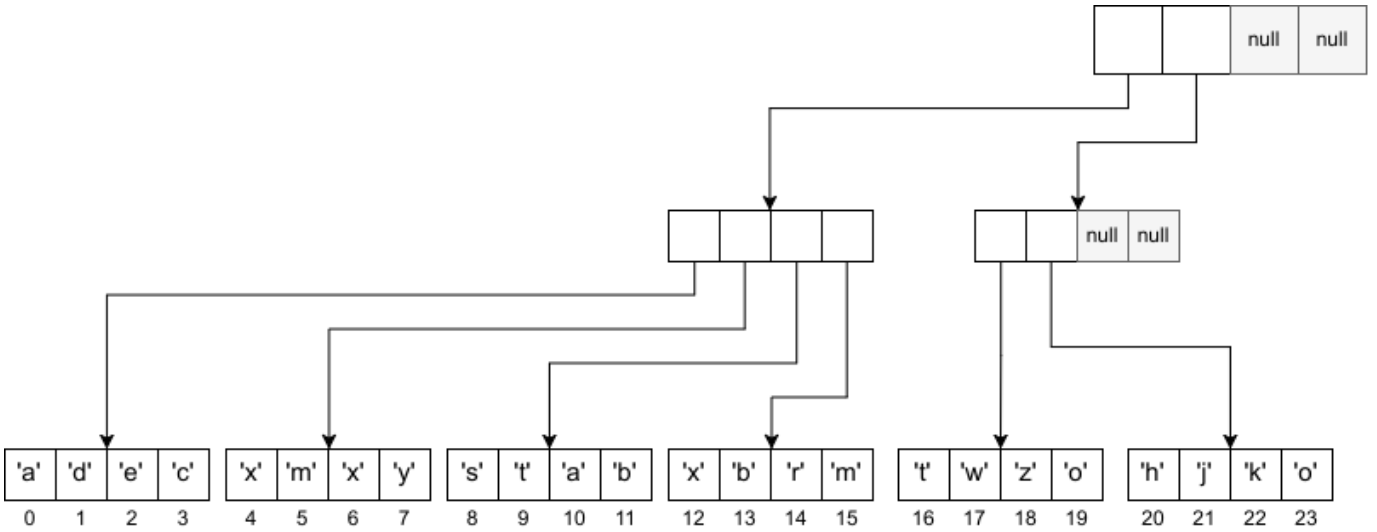
Scala Vector

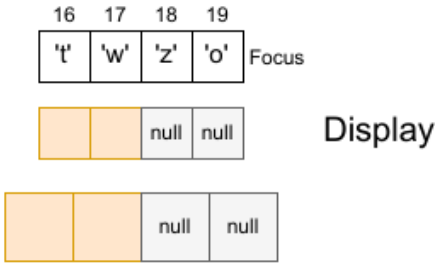
- Focus with Display



Scala Vector

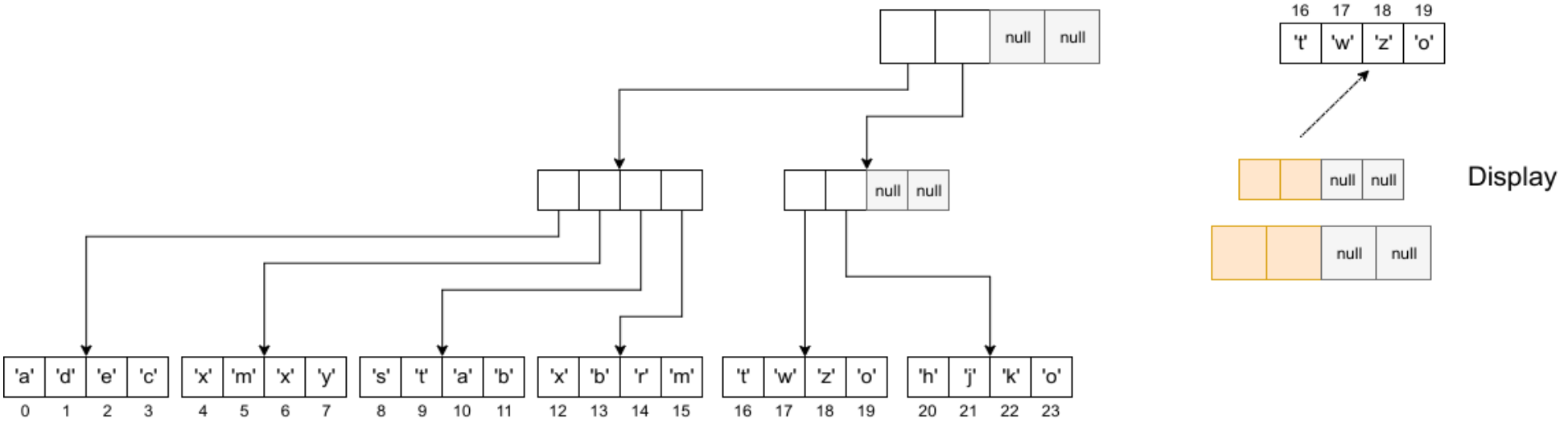
- Focus with Display





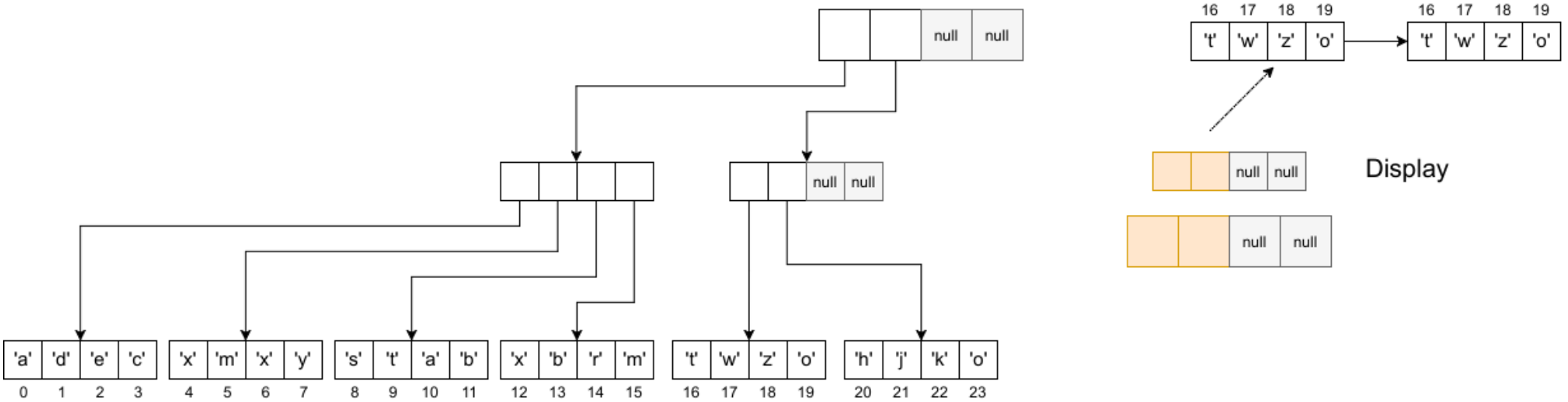
Scala Vector

- Focus with Display



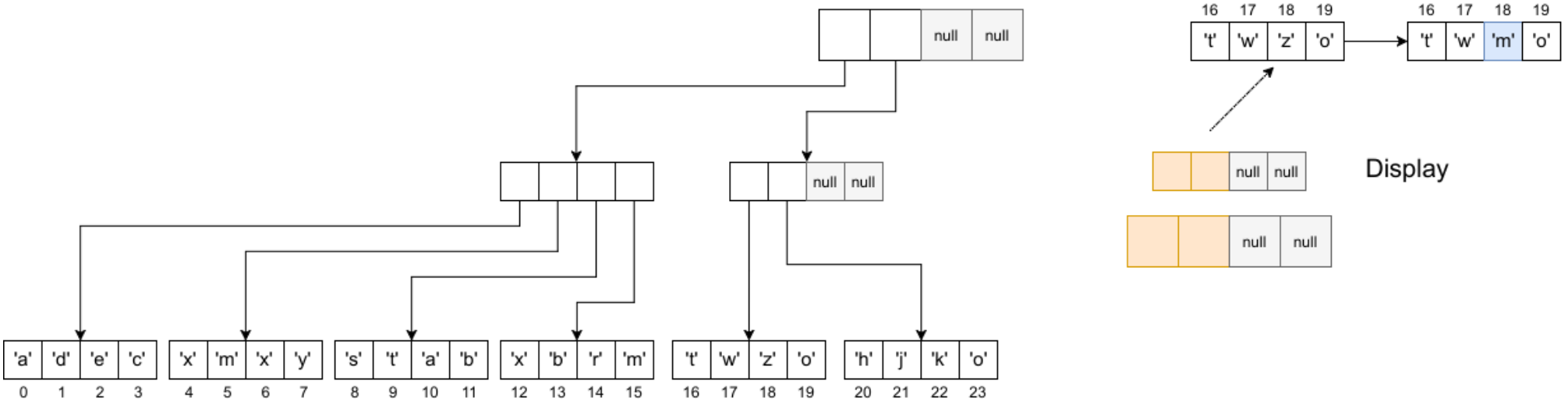
Scala Vector

- Focus with Display



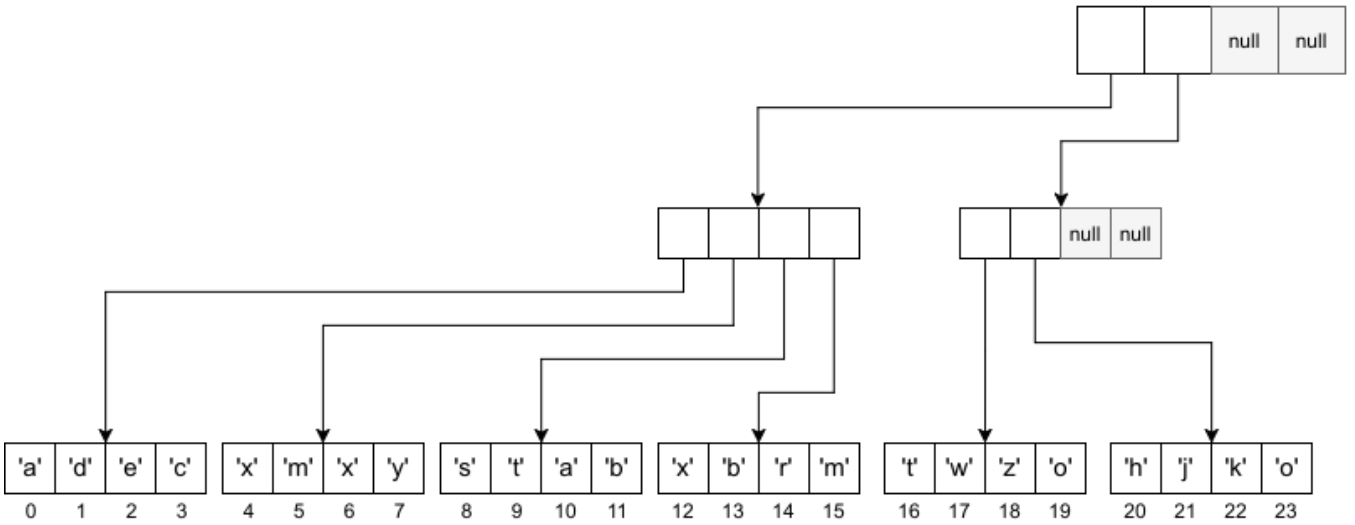
Scala Vector

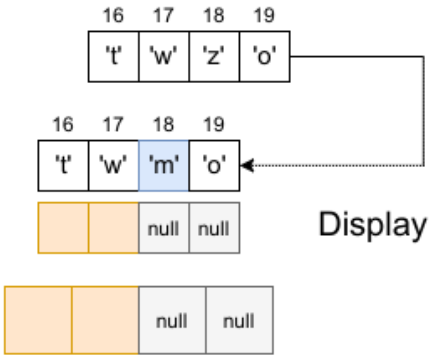
- Focus with Display



Scala Vector

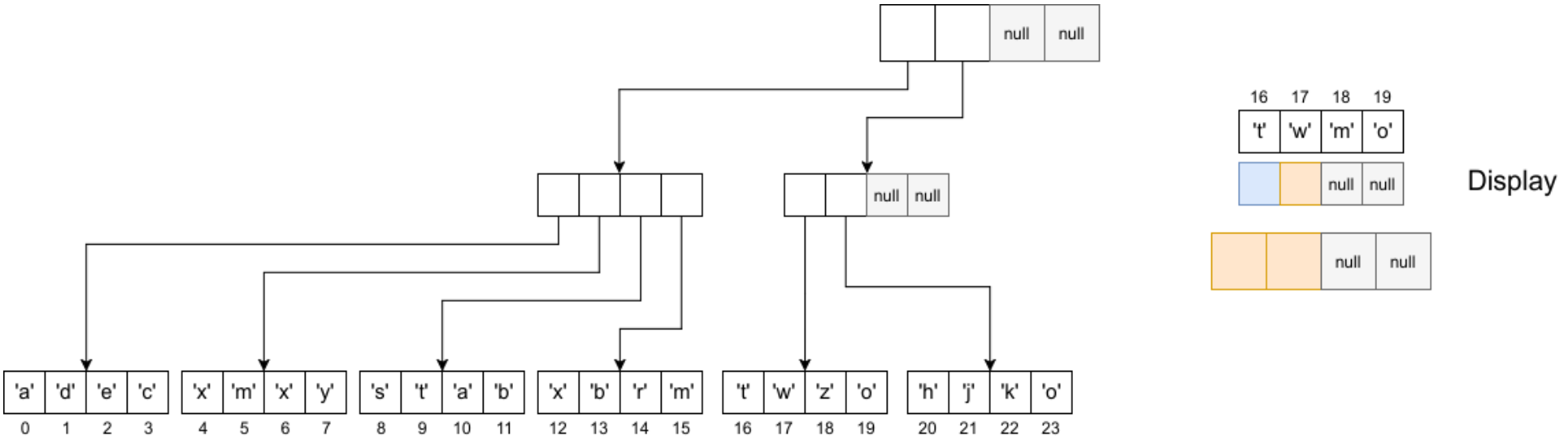
- Focus with Display





Scala Vector

- Focus with Display



Vectors: An Implementation

Thank you!

Presentation: https://github.com/AvramRobert/marvellous-functional-datastructures

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