Wormhole

A Fast Ordered Index for In-memory Data Management

von Markus Heidrich

28.01.2020

Ablauf

- Intro
- O-Notation
- Hashtabelle
- B+ Baum
- Prefix Baum
- Evaluation
- Zusammenfassung
- Quellen

Motivation

- Große Datenmengen
- Viel Memory
- Wenig Zeit
- Key-value Store
- Range query

O-Notation

- O: $\leq$
- o: <
- Ω: $\geq$
- ω: $>$

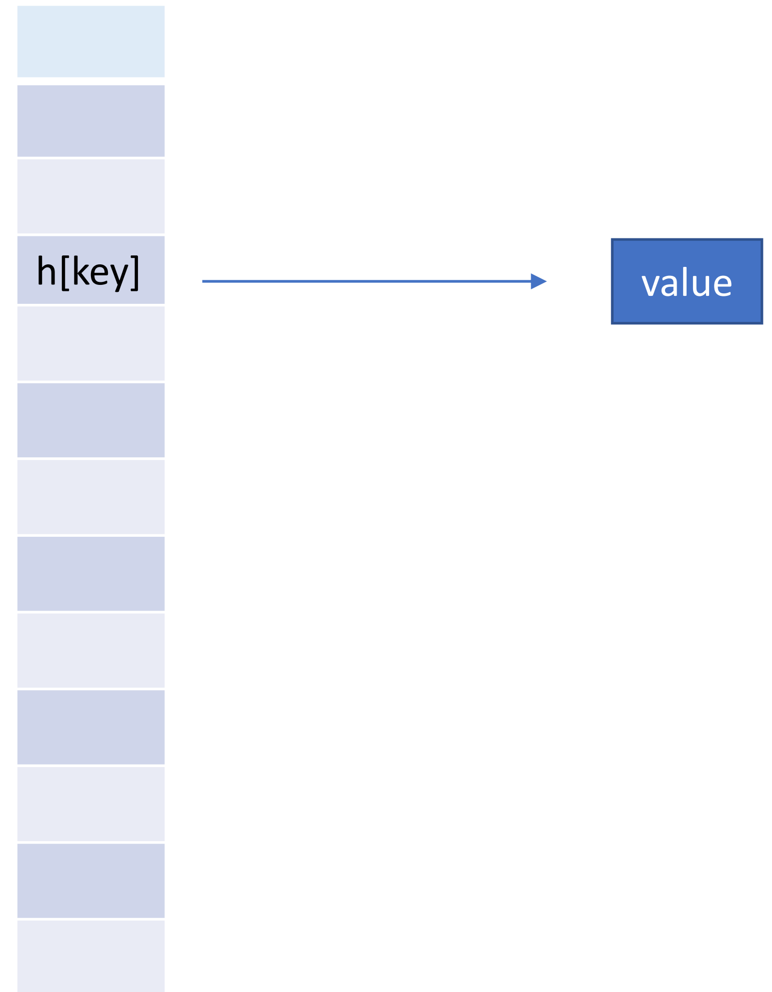
O-Notation

Bsp:

```
1
2  for i=0, i<n, i++:
3          do stuff
4
5
6  for i=0, i<50000000, i++:
7          do stuff
8
9
10 for i=1, i<n, i**2:
11         do stuff
12
13
14
15
16
17
18
```

Hashtabelle

- Key Value Paare
- Berechne Index aus h[key]
- SET, DEL und SEARCH in O(1)
- Kollisionsbehandlung



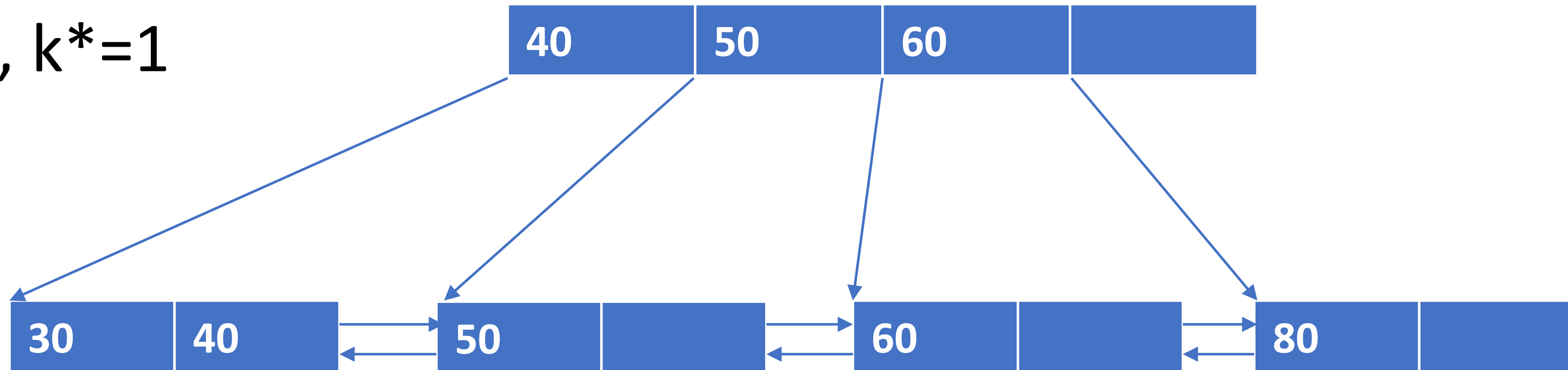
B+ Baum

- $h^* \geq 0$, k, k* > 0
- Jeder Pfad von Wurzel zu Blatt ist h*-1 lang
- Jeder innere Knoten hat mind k+1 Kinder
- Jeder innere Knoten hat höchstens 2k+1 Kinder
- Jedes Blatt hat mind k* und höchstens 2k* Einträge

- O(log N) lookup

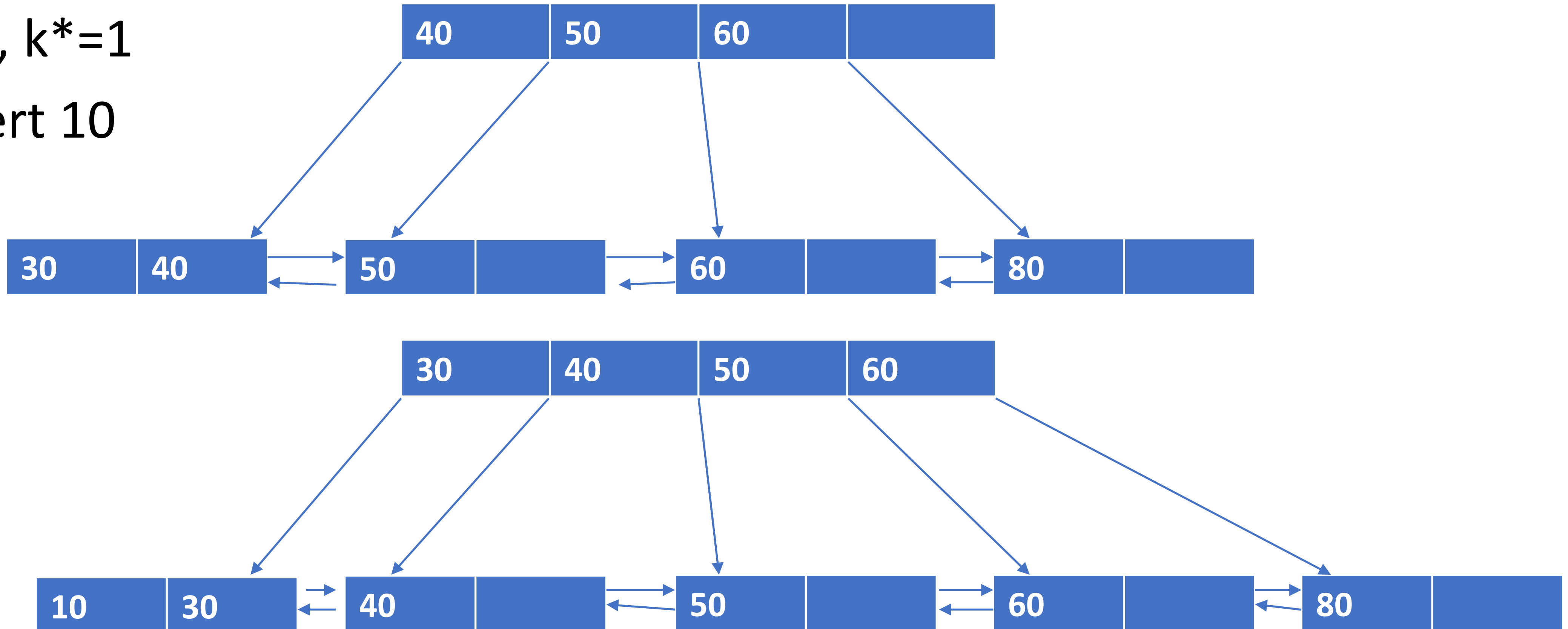
B+ Baum

- k=2, k*=1



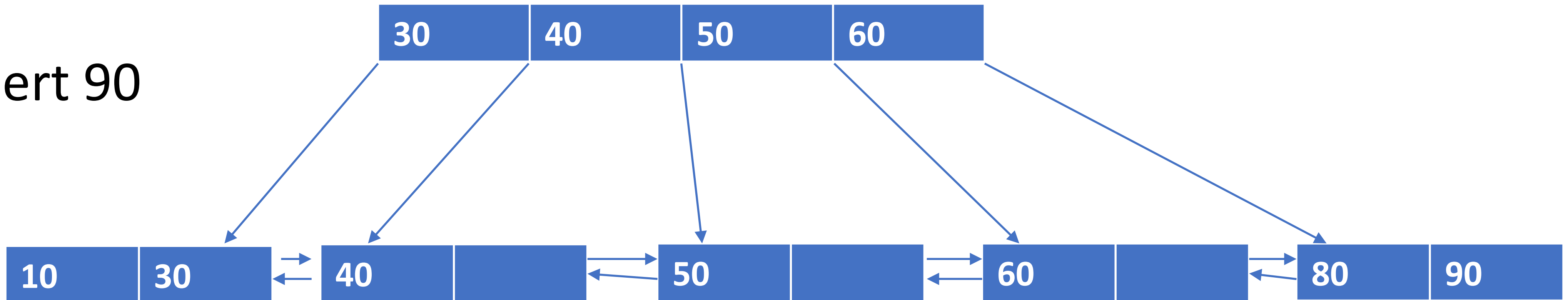
B+ Baum

- k=2, k*=1
- Insert 10

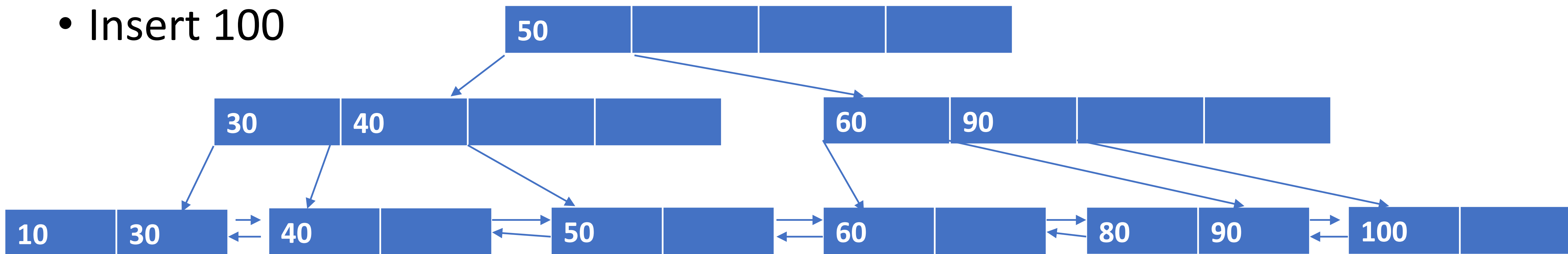


B+ Baum

- Insert 90



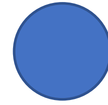
- Insert 100



Prefix Baum

- Auch Trie genannt
- Baum aus Strings
- O(L) wobei L Länge der Strings ist
- Höhere Platzkomplexität
- Auto-complete

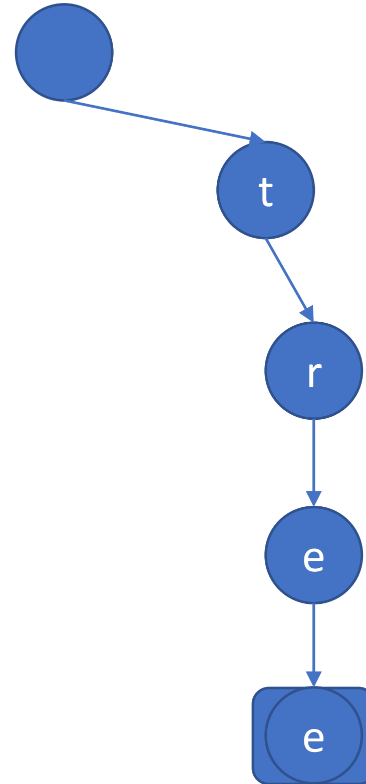
Prefix Baum



- Insert `tree`

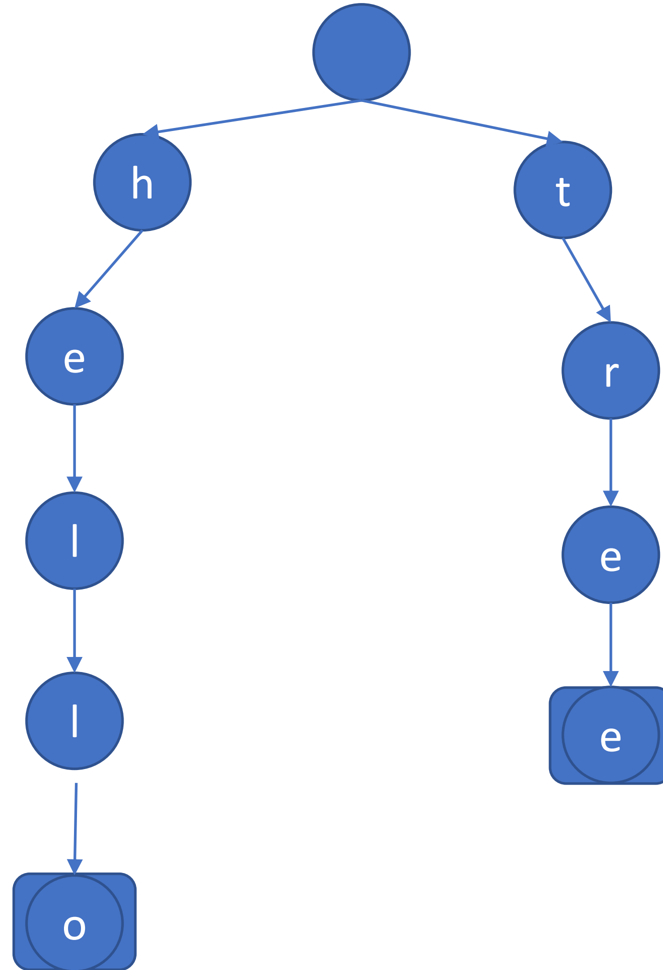
Prefix Baum

- Insert `tree`



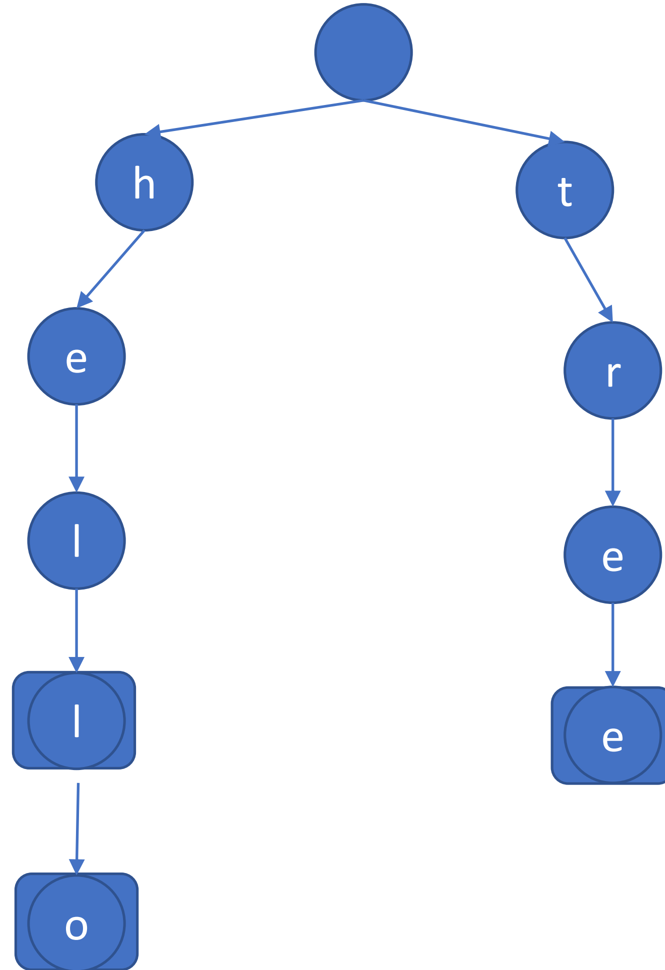
Prefix Baum

- Insert `hello`



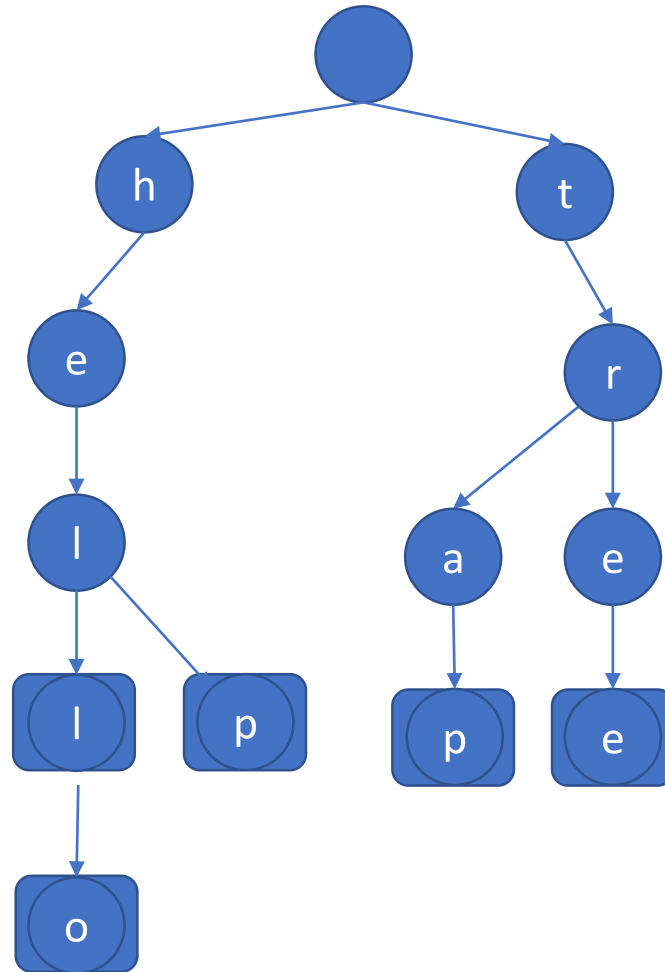
Prefix Baum

- Insert `hell`



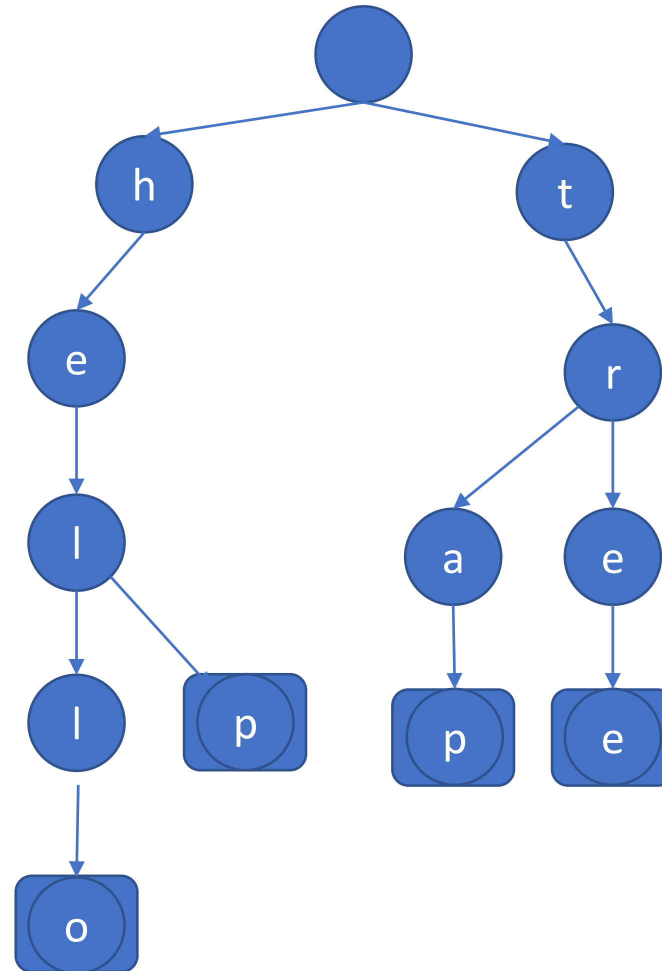
Prefix Baum

- Insert `help`
- Insert `trap`



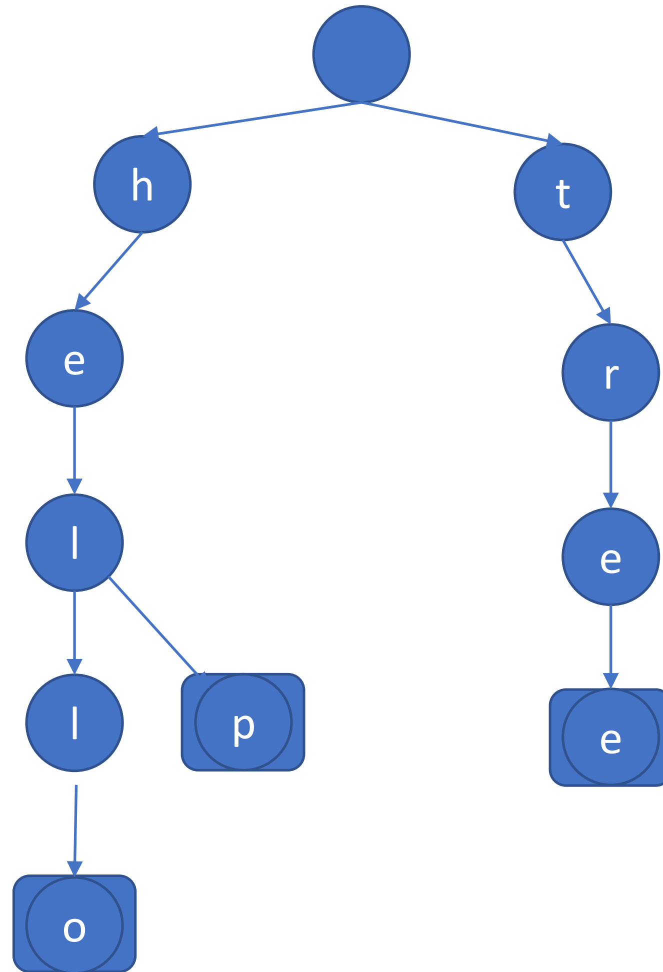
Prefix Baum

- Delete `hell`

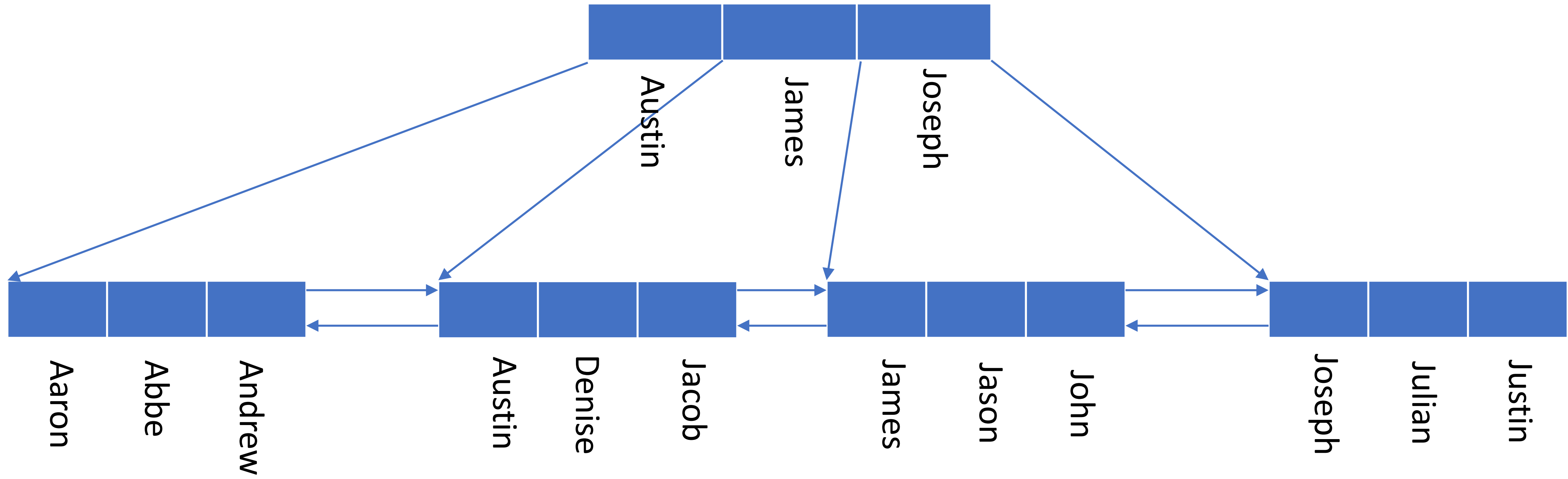


Prefix Baum

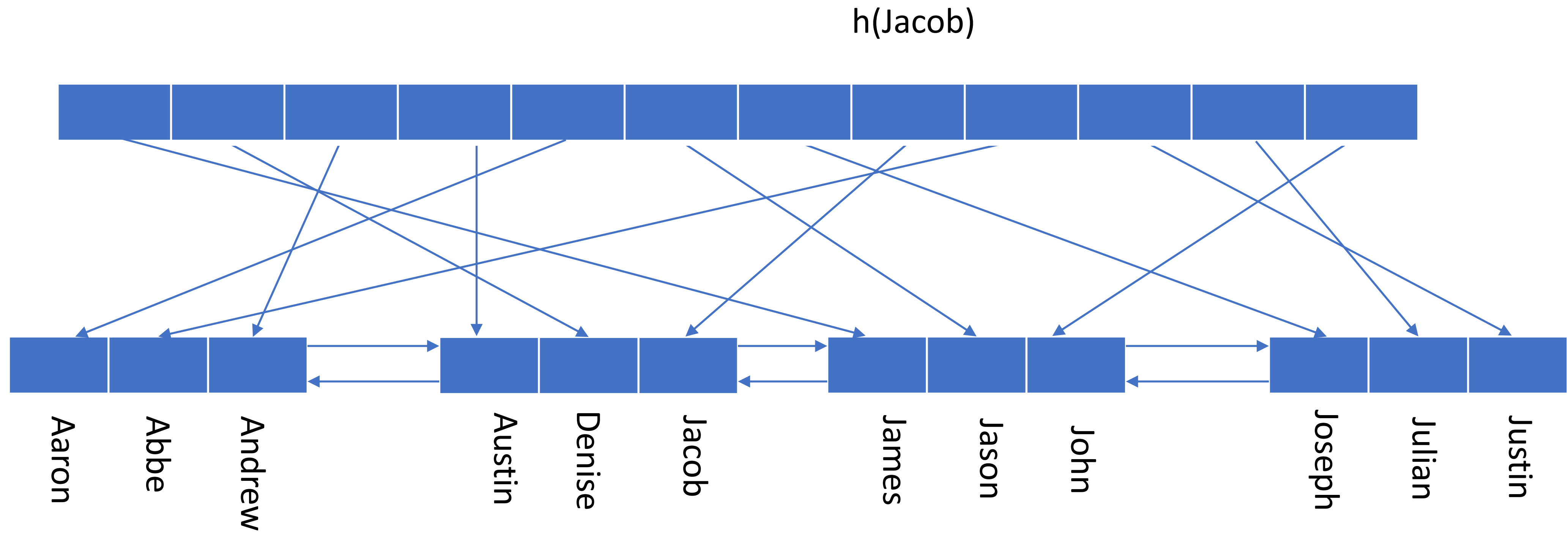
- Delete `trap`



Wormhole [1]

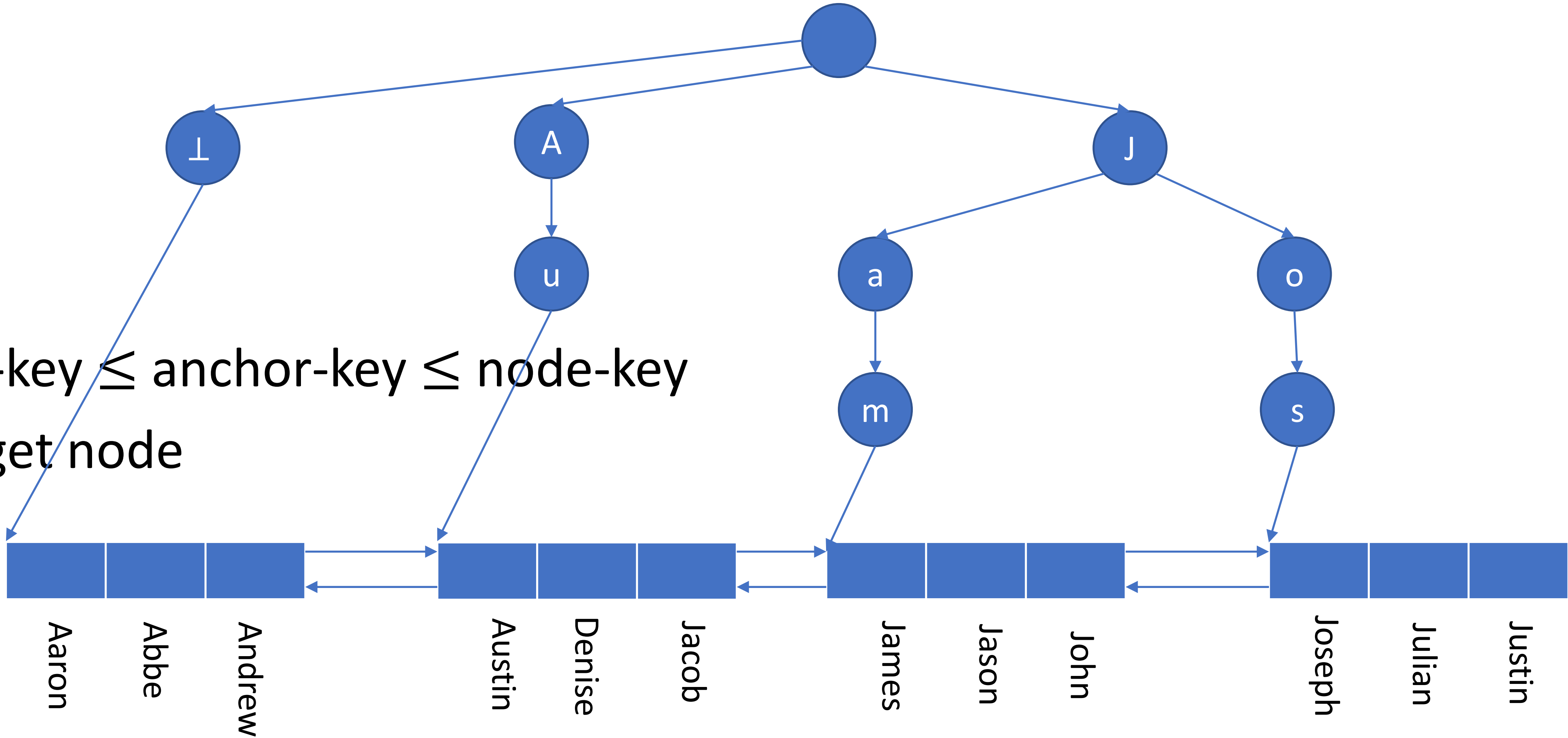


Wormhole [1]

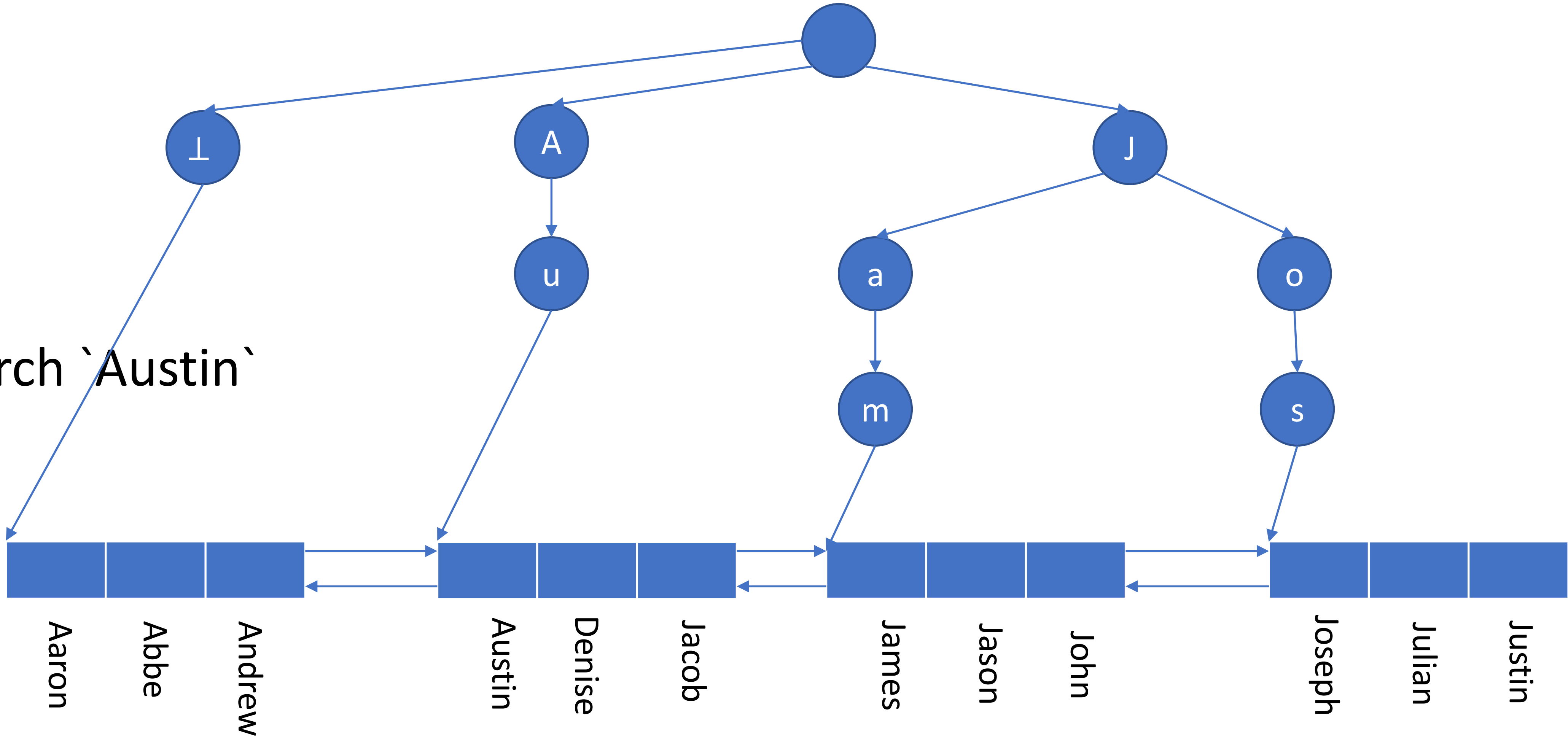


Wormhole [1]

- left-key ≤ anchor-key ≤ node-key
- Target node



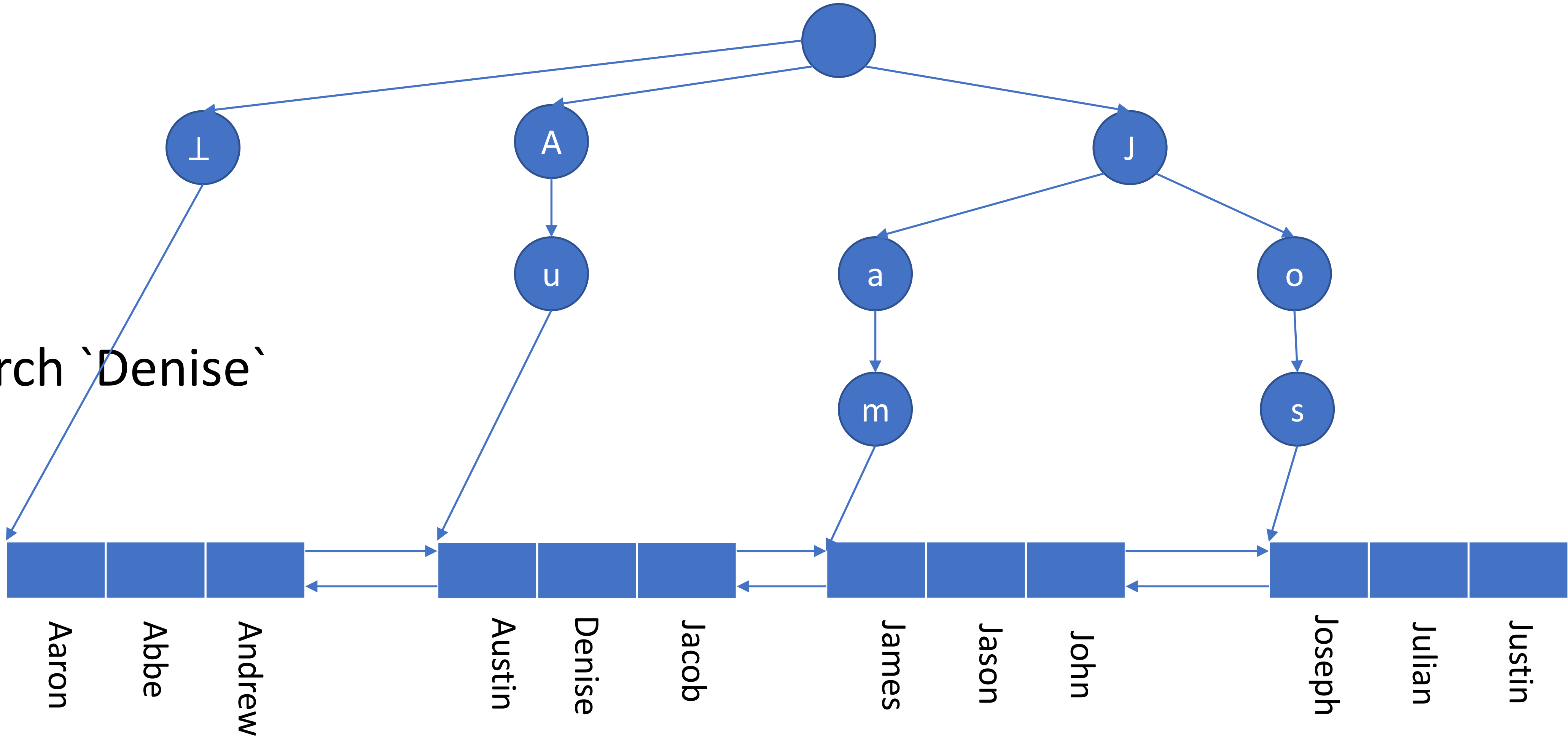
Wormhole [1]



- Search `Austin`

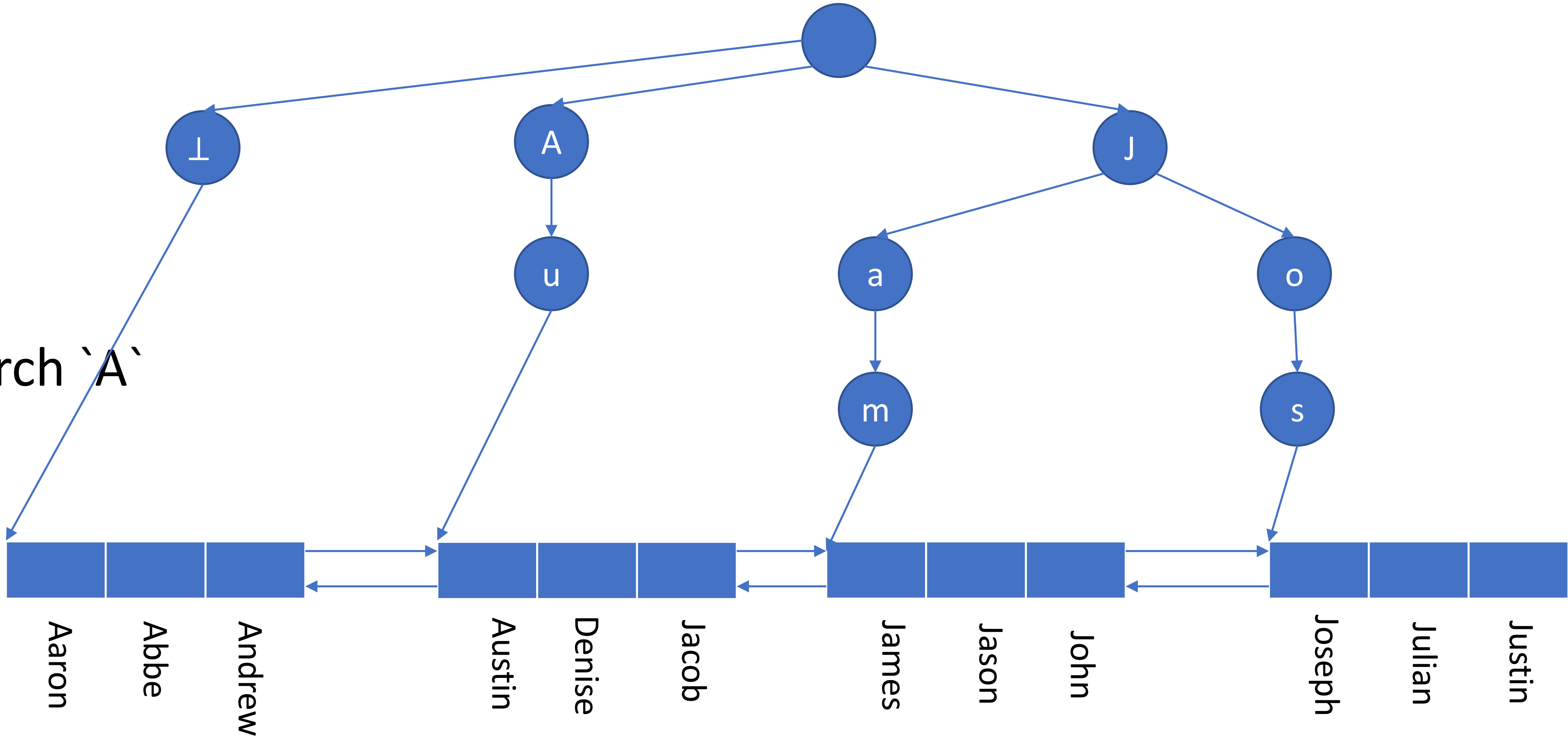
Wormhole [1]

- Search `Denise`

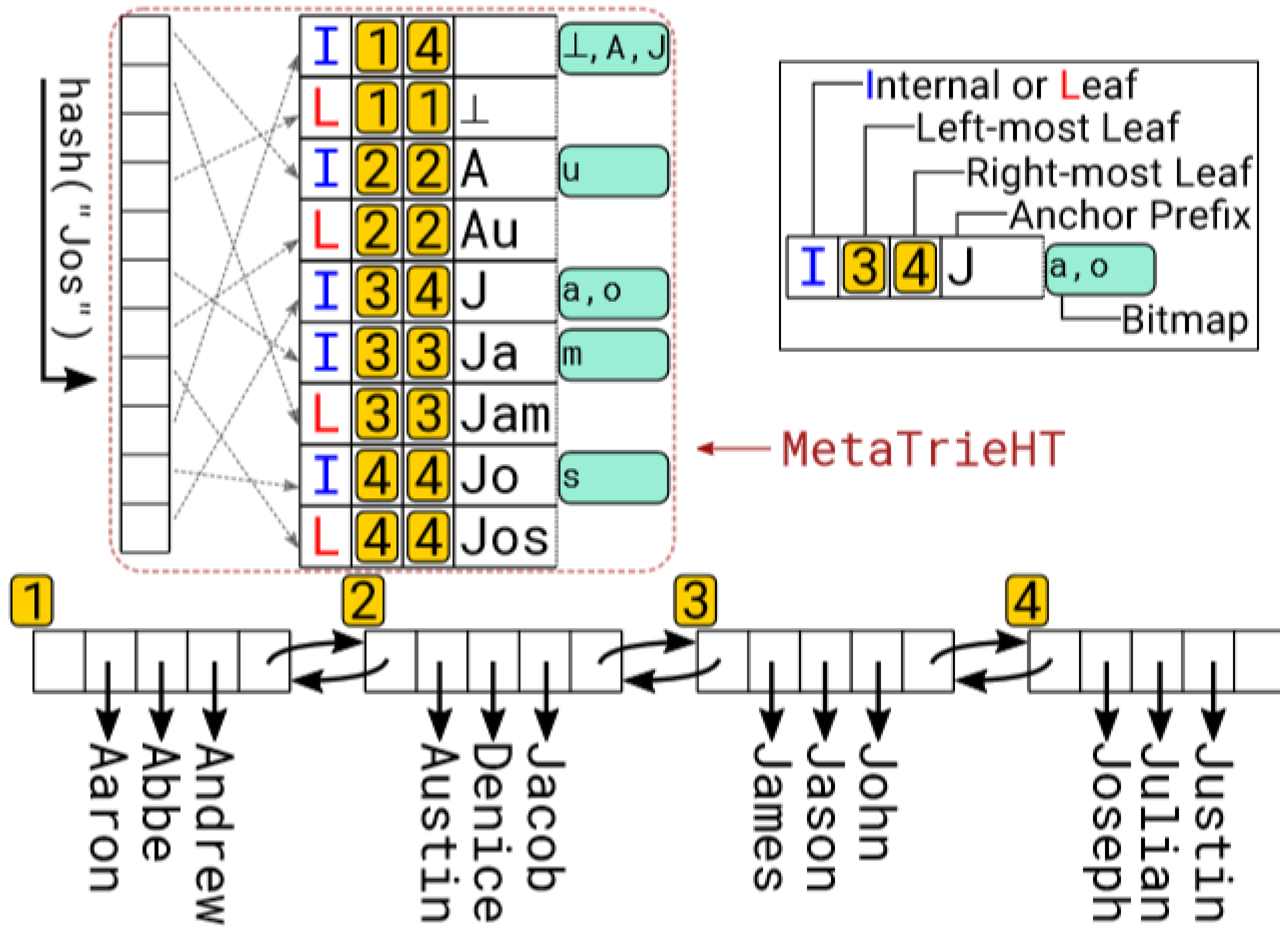


Wormhole [1]

- Search `A`



Wormhole [1]

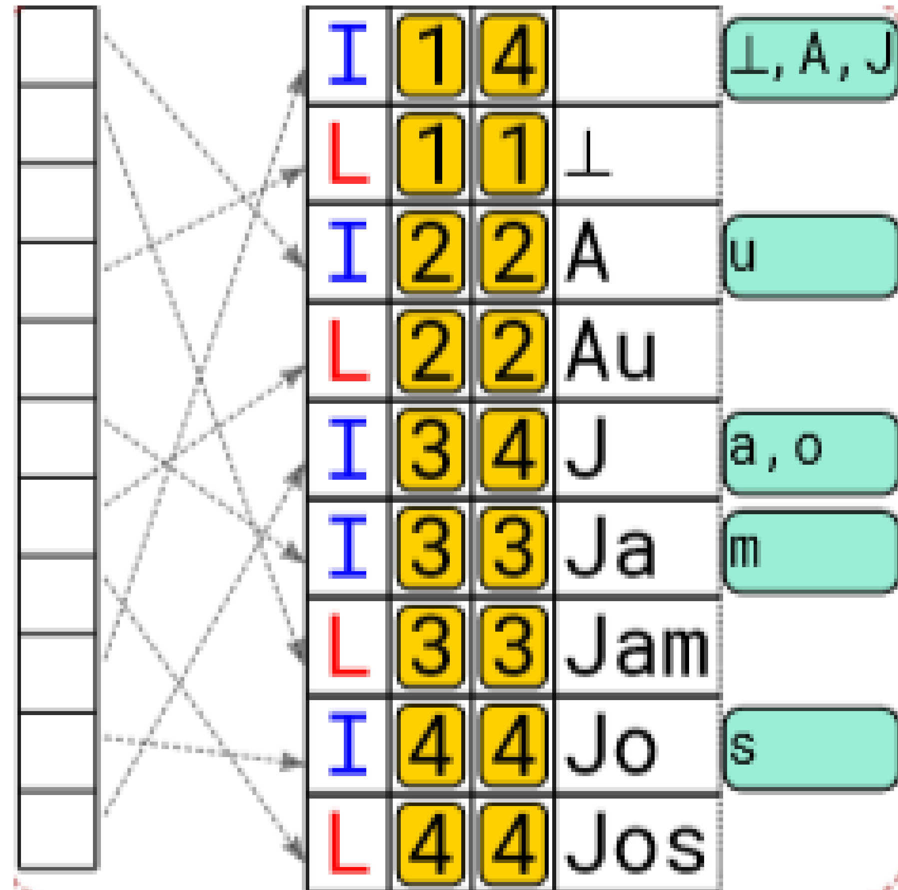


Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

```
1: function searchLPM(search_key, L_anc, L_key)
2:     m ← 0;     n ← min(L_anc, L_key)+1
3:     while (m+1) < n do
4:         prefix_len ← (m+n)/2
5:         if search_key[0 : prefix_len-1] is in the trie then
6:             m ← prefix_len
7:         else  n ← prefix_len
8:     return search_key[0 : m-1]
```

- search_key = James
- L_key = 5
- L_anc = 4

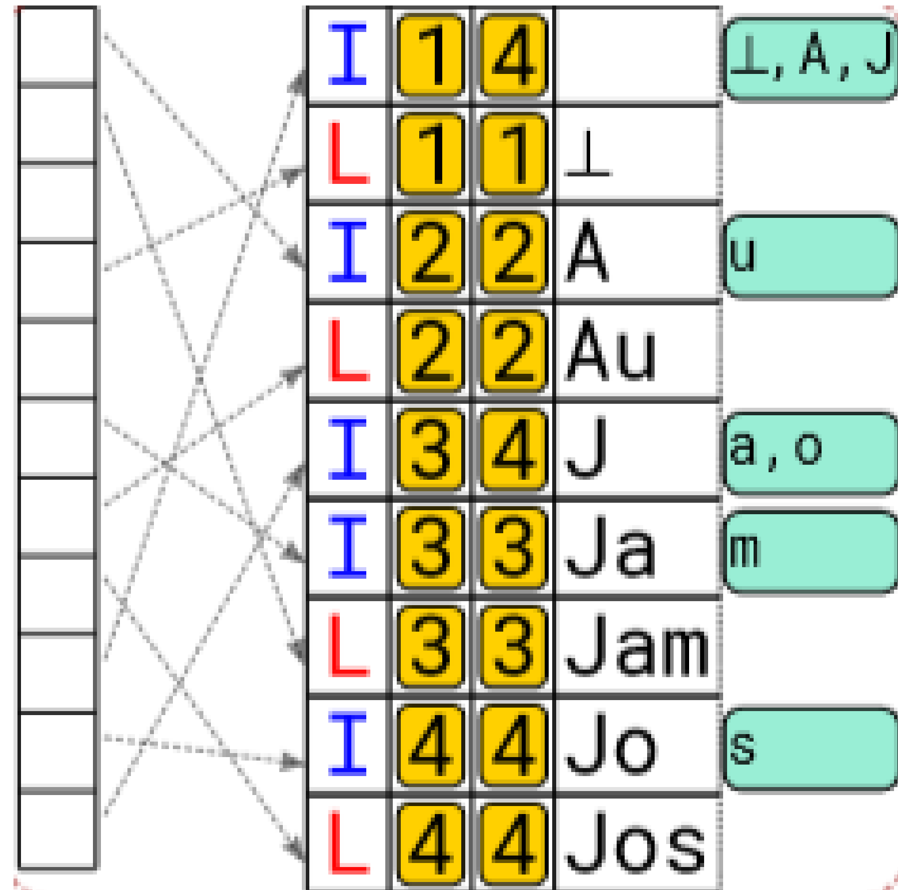


Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

```
1: function searchLPM(search_key, L_anc, L_key)
2:     m ← 0;     n ← min(L_anc, L_key)+1
3:     while (m+1) < n do
4:         prefix_len ← (m+n)/2
5:         if search_key[0 : prefix_len-1] is in the trie then
6:             m ← prefix_len
7:         else    n ← prefix_len
8:     return search_key[0 : m-1]
```

- m = 0
- n = 4

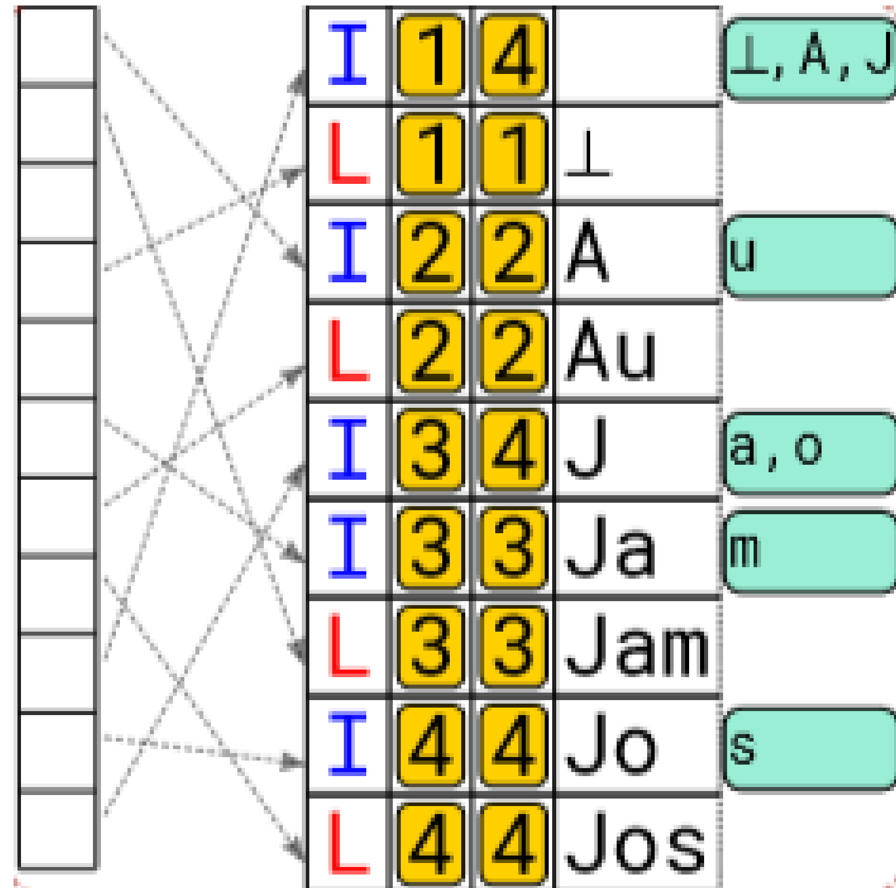


Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

```
1: function searchLPM(search_key, L_anc, L_key)
2:     m ← 0;     n ← min(L_anc, L_key)+1
3:     while (m+1) < n do
4:         prefix_len ← (m+n)/2
5:         if search_key[0 : prefix_len-1] is in the trie then
6:             m ← prefix_len
7:         else   n ← prefix_len
8:     return search_key[0 : m-1]
```

- m = 0
- n = 4
- prefix_len = 2

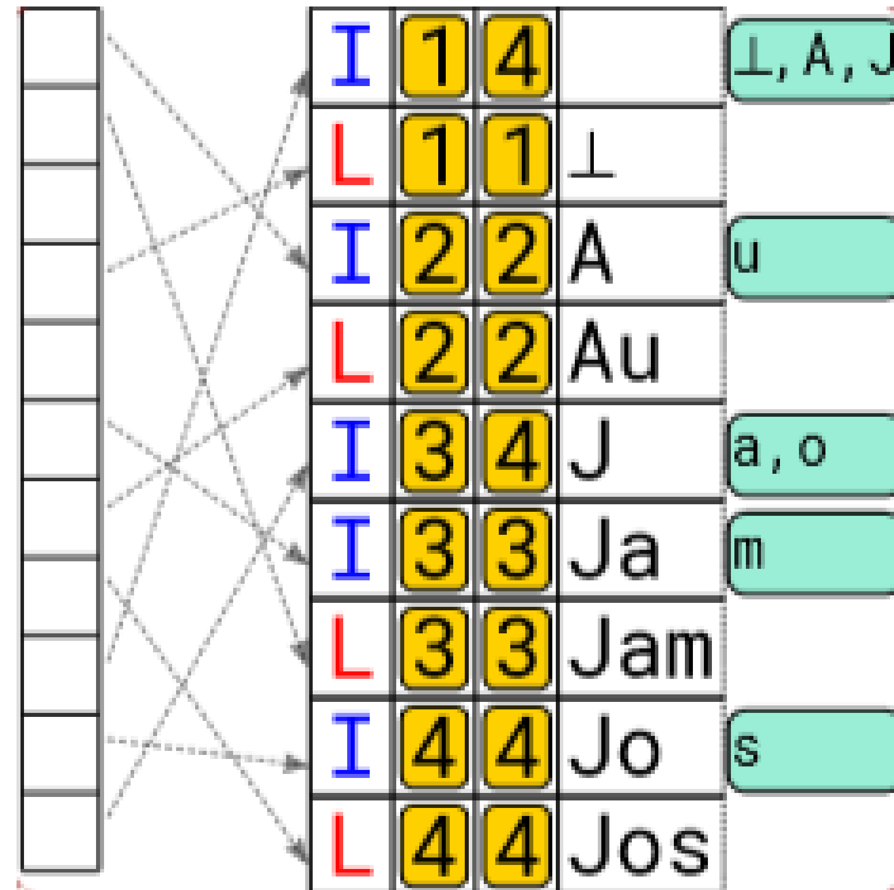


Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

1: **function** searchLPM(search_key, L_{anc}, L_{key})
2: m ← 0; n ← min(L_{anc}, L_{key})+1
3: **while** (m+1) < n **do**
4: prefix_len ← (m+n)/2
5: **if** search_key[0 : prefix_len-1] is in the trie **then**
6: m ← prefix_len
7: **else** n ← prefix_len
8: **return** search_key[0 : m-1]

- m = 2
- n = 4
- prefix_len = 2

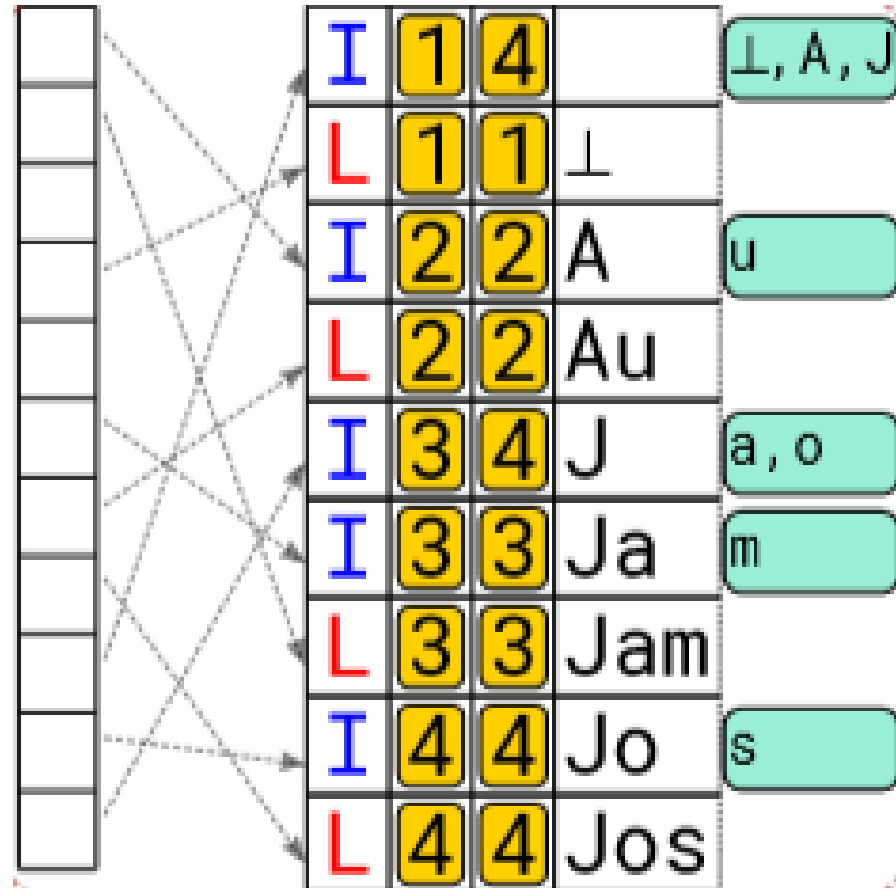


Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

```
1: function searchLPM(search_key, L_anc, L_key)
2:     m ← 0;     n ← min(L_anc, L_key)+1
3:     while (m+1) < n do
4:         prefix_len ← (m+n)/2
5:         if search_key[0 : prefix_len-1] is in the trie then
6:             m ← prefix_len
7:         else   n ← prefix_len
8:     return search_key[0 : m-1]
```

- m = 2
- n = 4
- prefix_len = 3

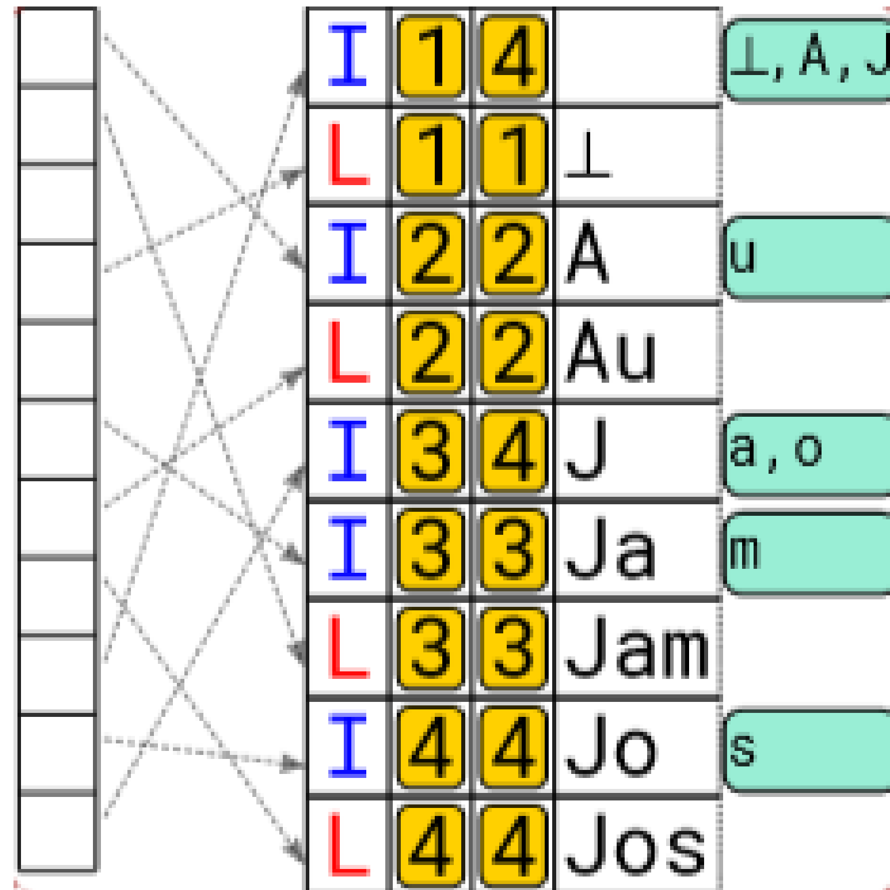


Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

```
1: function searchLPM(search_key, L_anc, L_key)
2:     m ← 0;     n ← min(L_anc, L_key)+1
3:     while (m+1) < n do
4:         prefix_len ← (m+n)/2
5:         if search_key[0 : prefix_len-1] is in the trie then
6:             m ← prefix_len
7:         else    n ← prefix_len
8:     return search_key[0 : m-1]
```

- m = 3
- n = 4
- prefix_len = 3



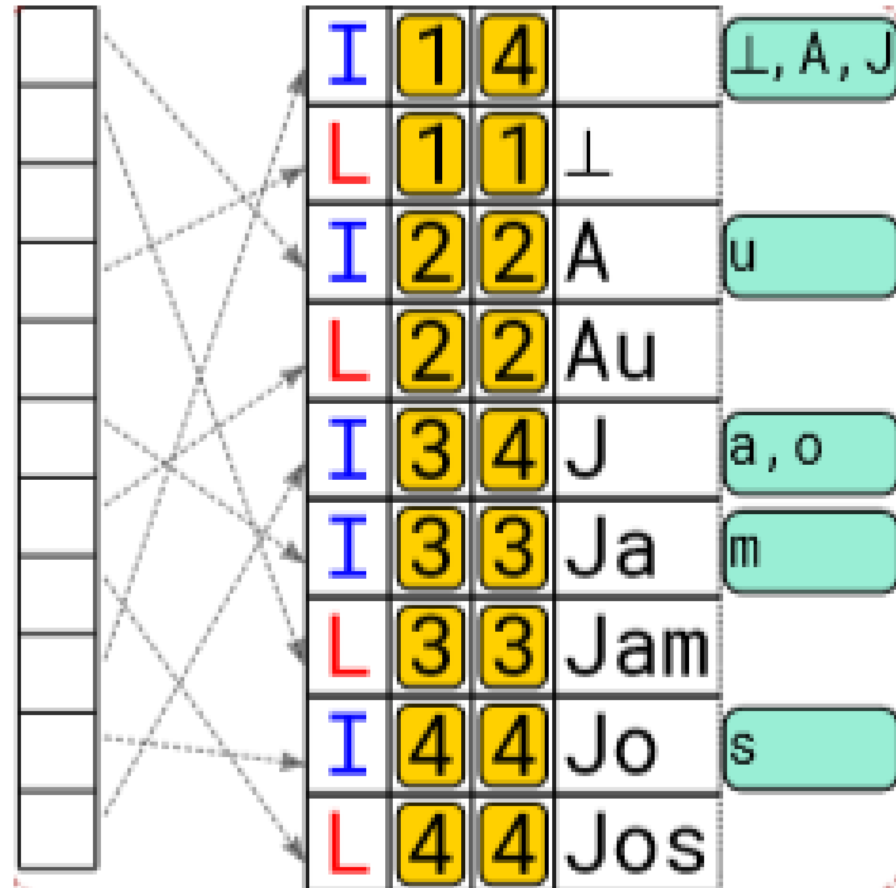
Wormhole [1]

Algorithm 1 Binary Search on Prefix Lengths

1: **function** searchLPM(search_key, L_{anc}, L_{key})
2: m ← 0; n ← min(L_{anc}, L_{key})+1
3: **while** (m+1) < n **do**
4: prefix_len ← (m+n)/2
5: **if** search_key[0 : prefix_len-1] is in the trie **then**
6: m ← prefix_len
7: **else** n ← prefix_len
8: **return** search_key[0 : m-1]

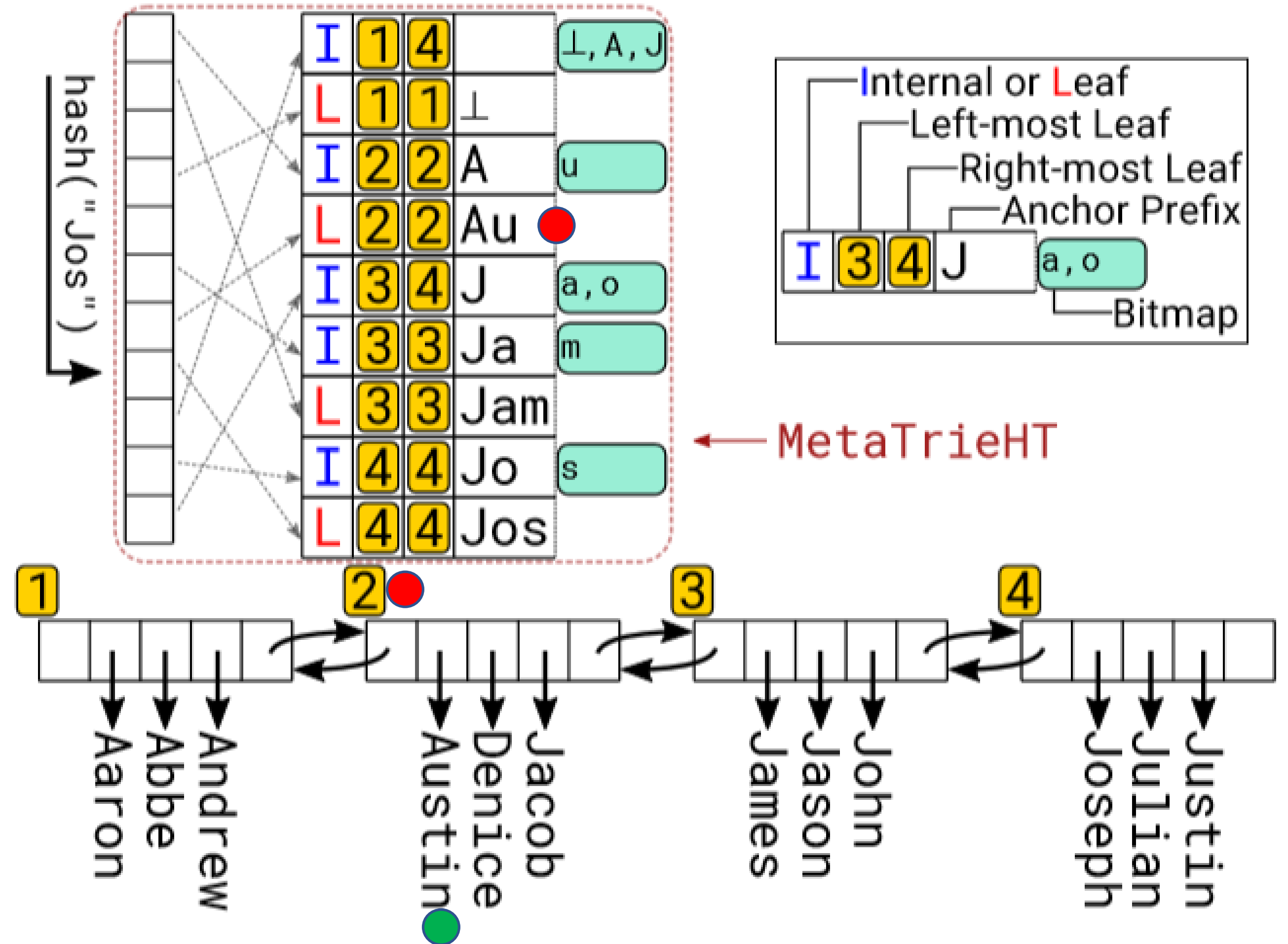
- return Jam

- O(log(min(L_anc,L_key)))



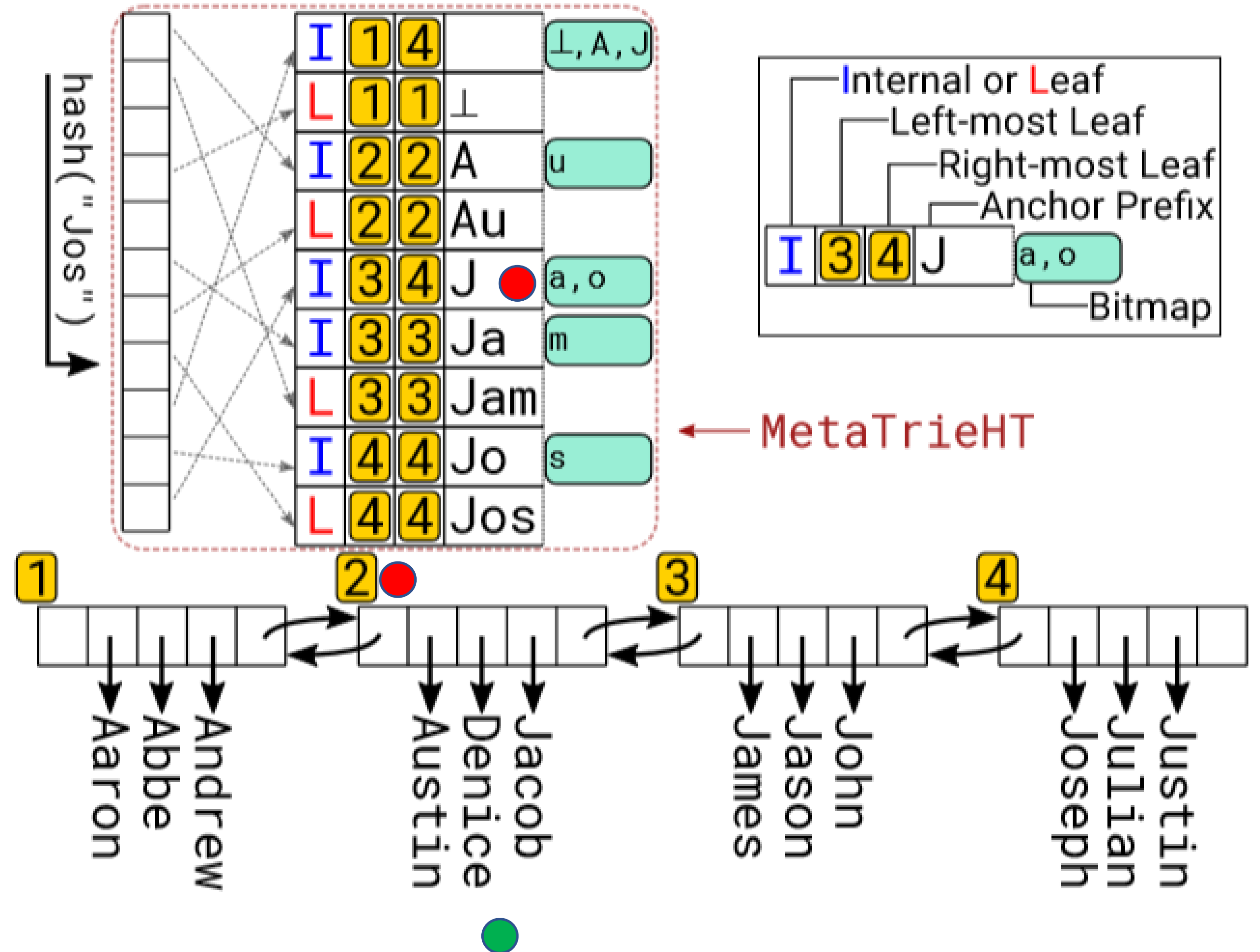
Wormhole [1]

```
function searchTrieHT(wh, key)
  node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
  if node.type = LEAF then return node
  else if node.key.len = key.len then
    leaf ← node.leftmost
    if key < leaf.anchor then leaf ← leaf.left
    return leaf
  missing ← key.tokens[node.key.len]
  sibling ← findOneSibling(node.bitmap, missing)
  child ← htGet(wh.ht, concat(node.key, sibling))
  if child.type = LEAF then
    if sibling > missing then child ← child.left
    return child
  else
    if sibling > missing then return child.leftmost.left
    else return child.rightmost
```



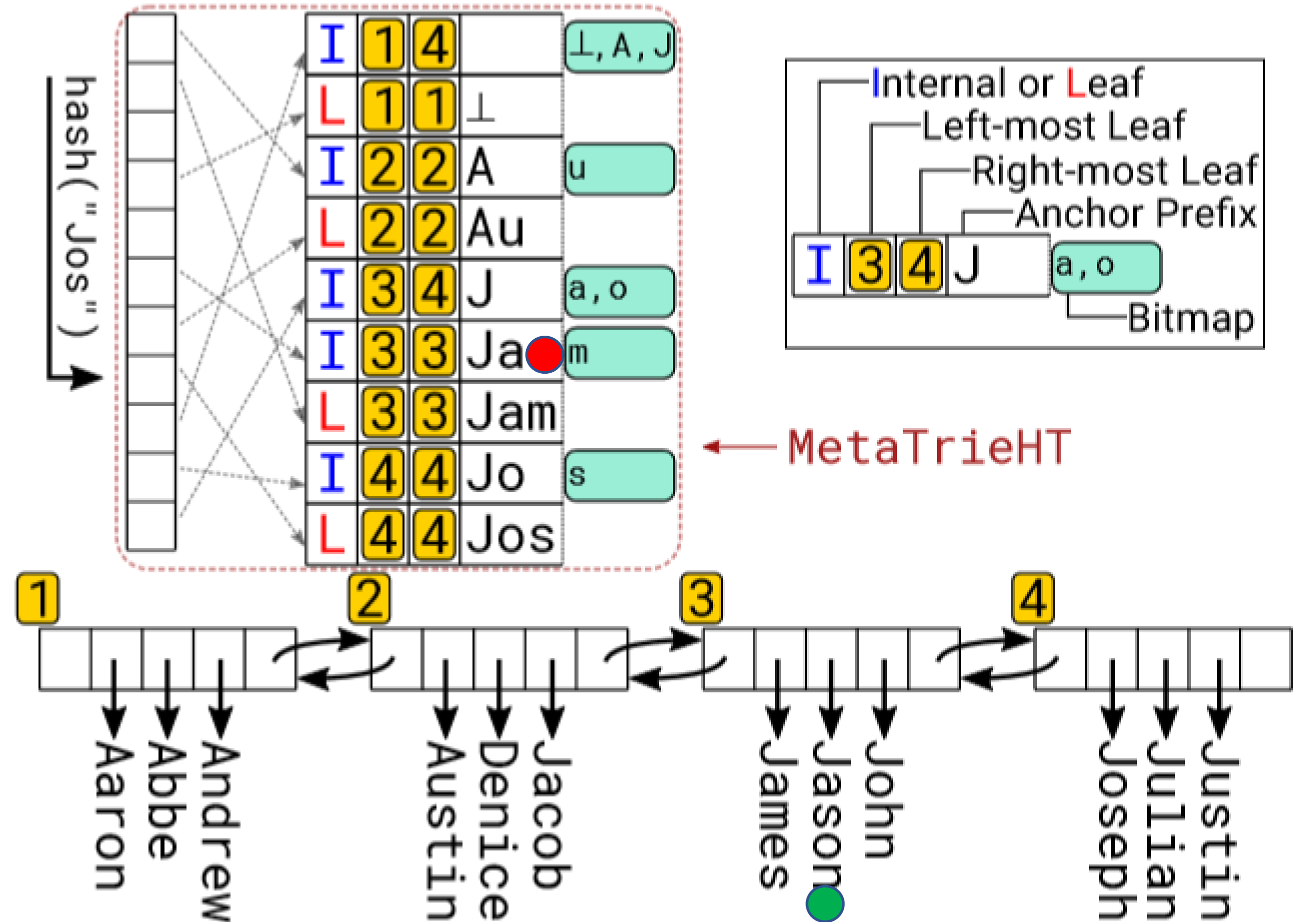
Wormhole [1]

```
function searchTrieHT(wh, key)
    node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
    if node.type = LEAF then return node
    else if node.key.len = key.len then
        leaf ← node.leftmost
        if key < leaf.anchor then leaf ← leaf.left
        return leaf
    missing ← key.tokens[node.key.len]
    sibling ← findOneSibling(node.bitmap, missing)
    child ← htGet(wh.ht, concat(node.key, sibling))
    if child.type = LEAF then
        if sibling > missing then child ← child.left
        return child
    else
        if sibling > missing then return child.leftmost.left
        else return child.rightmost
```



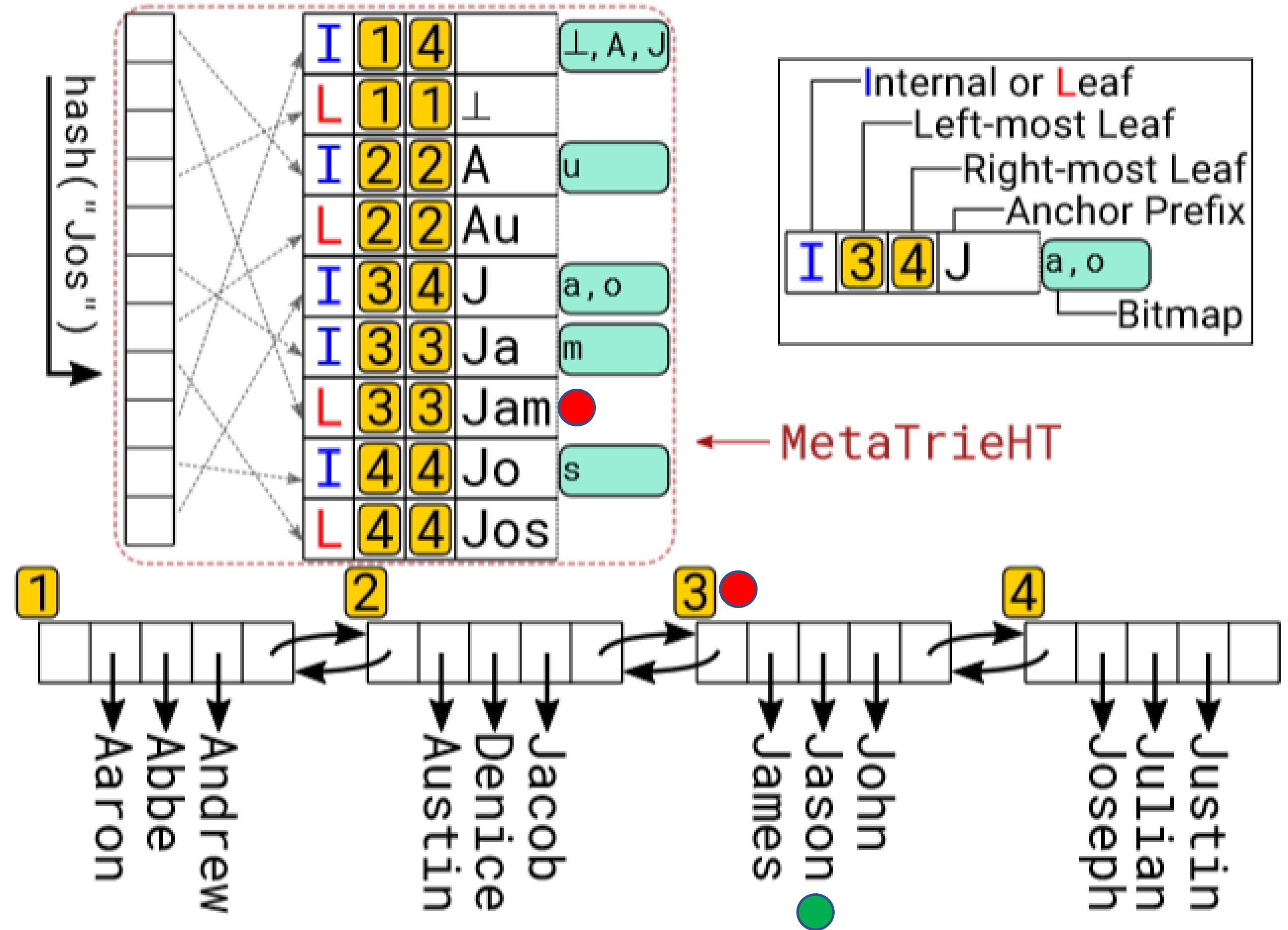
Wormhole [1]

```
function searchTrieHT(wh, key)
    node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
    if node.type = LEAF then return node
    else if node.key.len = key.len then
        leaf ← node.leftmost
        if key < leaf.anchor then leaf ← leaf.left
        return leaf
    missing ← key.tokens[node.key.len]
    sibling ← findOneSibling(node.bitmap, missing)
    child ← htGet(wh.ht, concat(node.key, sibling))
    if child.type = LEAF then
        if sibling > missing then child ← child.left
        return child
    else
        if sibling > missing then return child.leftmost.left
        else return child.rightmost
```



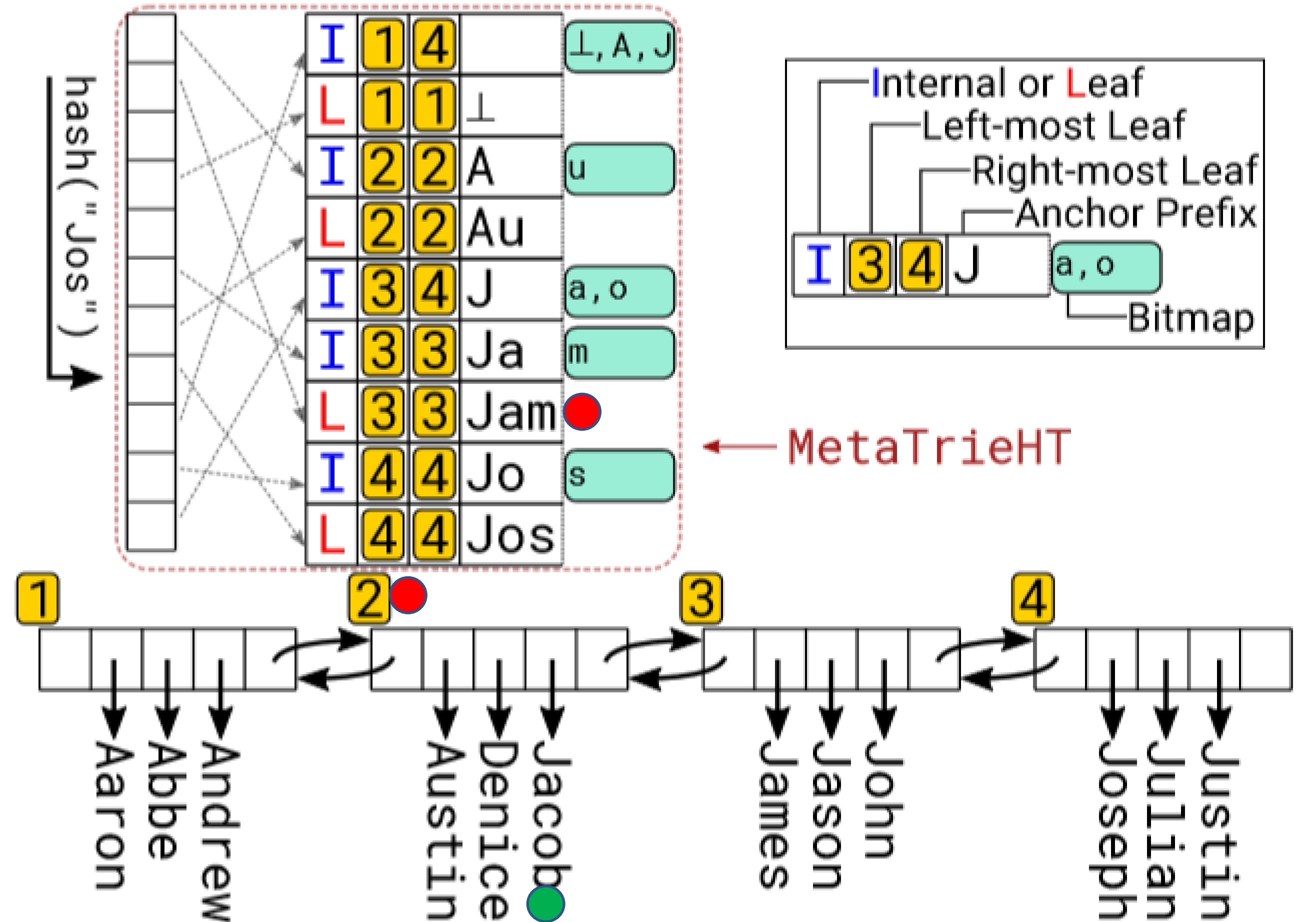
Wormhole [1]

```
function searchTrieHT(wh, key)
    node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
    if node.type = LEAF then return node
    else if node.key.len = key.len then
        leaf ← node.leftmost
        if key < leaf.anchor then leaf ← leaf.left
        return leaf
    missing ← key.tokens[node.key.len]
    sibling ← findOneSibling(node.bitmap, missing)
    child ← htGet(wh.ht, concat(node.key, sibling))
    if child.type = LEAF then
        if sibling > missing then child ← child.left
        return child
    else
        if sibling > missing then return child.leftmost.left
        else return child.rightmost
```



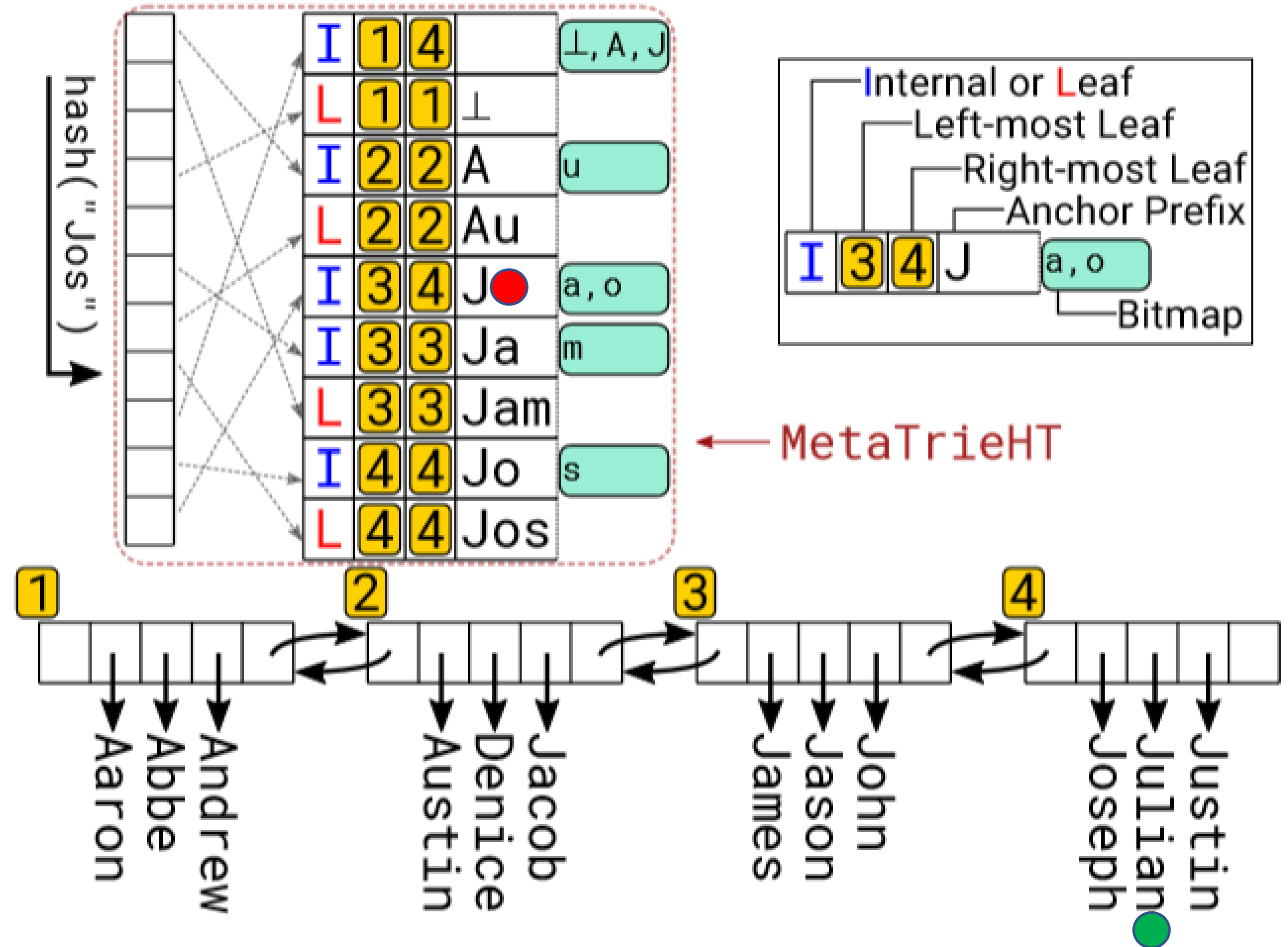
Wormhole [1]

```
function searchTrieHT(wh, key)
    node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
    if node.type = LEAF then return node
    else if node.key.len = key.len then
        leaf ← node.leftmost
        if key < leaf.anchor then leaf ← leaf.left
        return leaf
    missing ← key.tokens[node.key.len]
    sibling ← findOneSibling(node.bitmap, missing)
    child ← htGet(wh.ht, concat(node.key, sibling))
    if child.type = LEAF then
        if sibling > missing then child ← child.left
        return child
    else
        if sibling > missing then return child.leftmost.left
        else return child.rightmost
```



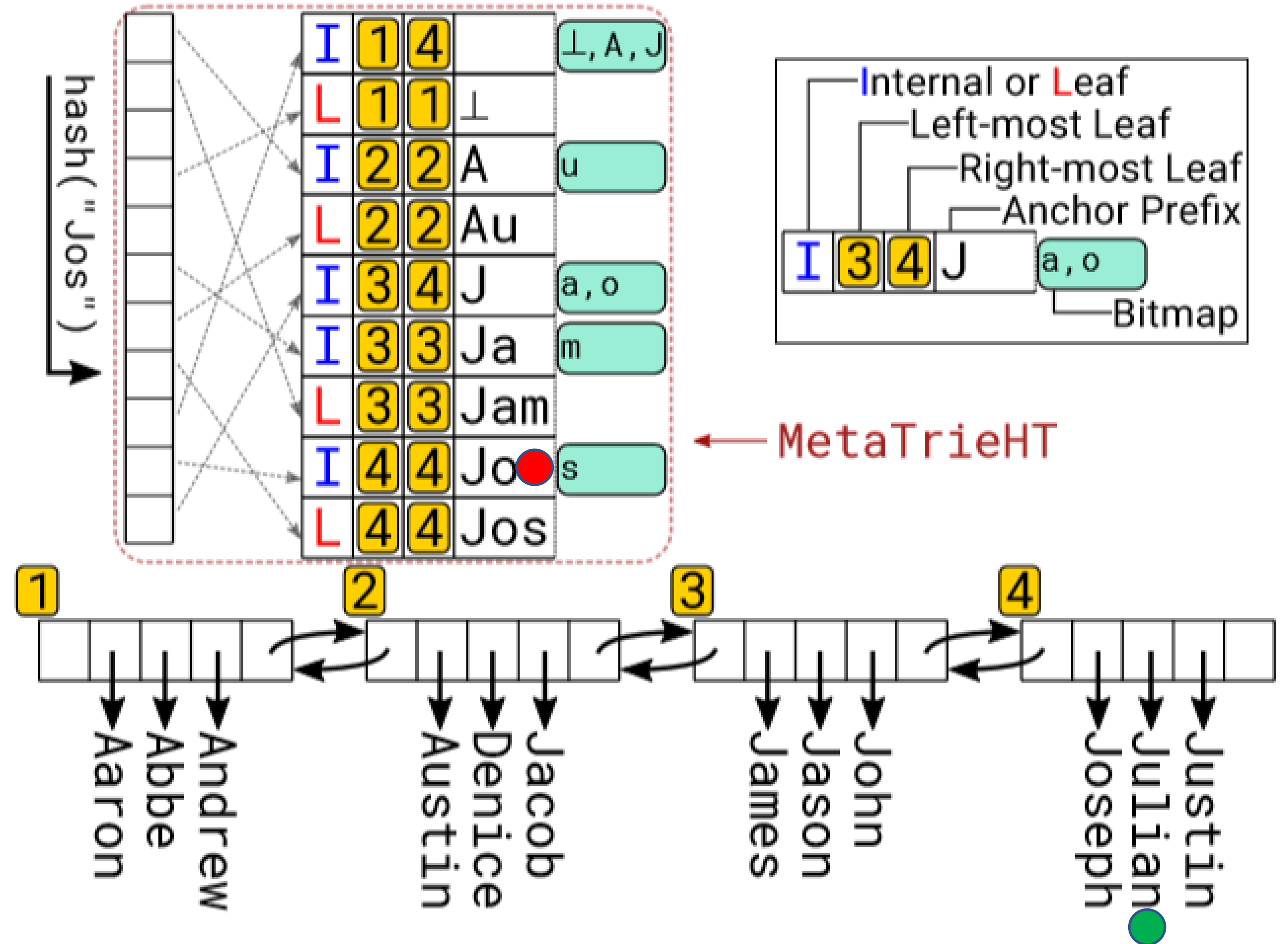
Wormhole [1]

```
function searchTrieHT(wh, key)
    node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
    if node.type = LEAF then return node
    else if node.key.len = key.len then
        leaf ← node.leftmost
        if key < leaf.anchor then leaf ← leaf.left
        return leaf
    missing ← key.tokens[node.key.len]
    sibling ← findOneSibling(node.bitmap, missing)
    child ← htGet(wh.ht, concat(node.key, sibling))
    if child.type = LEAF then
        if sibling > missing then child ← child.left
        return child
    else
        if sibling > missing then return child.leftmost.left
        else return child.rightmost
```



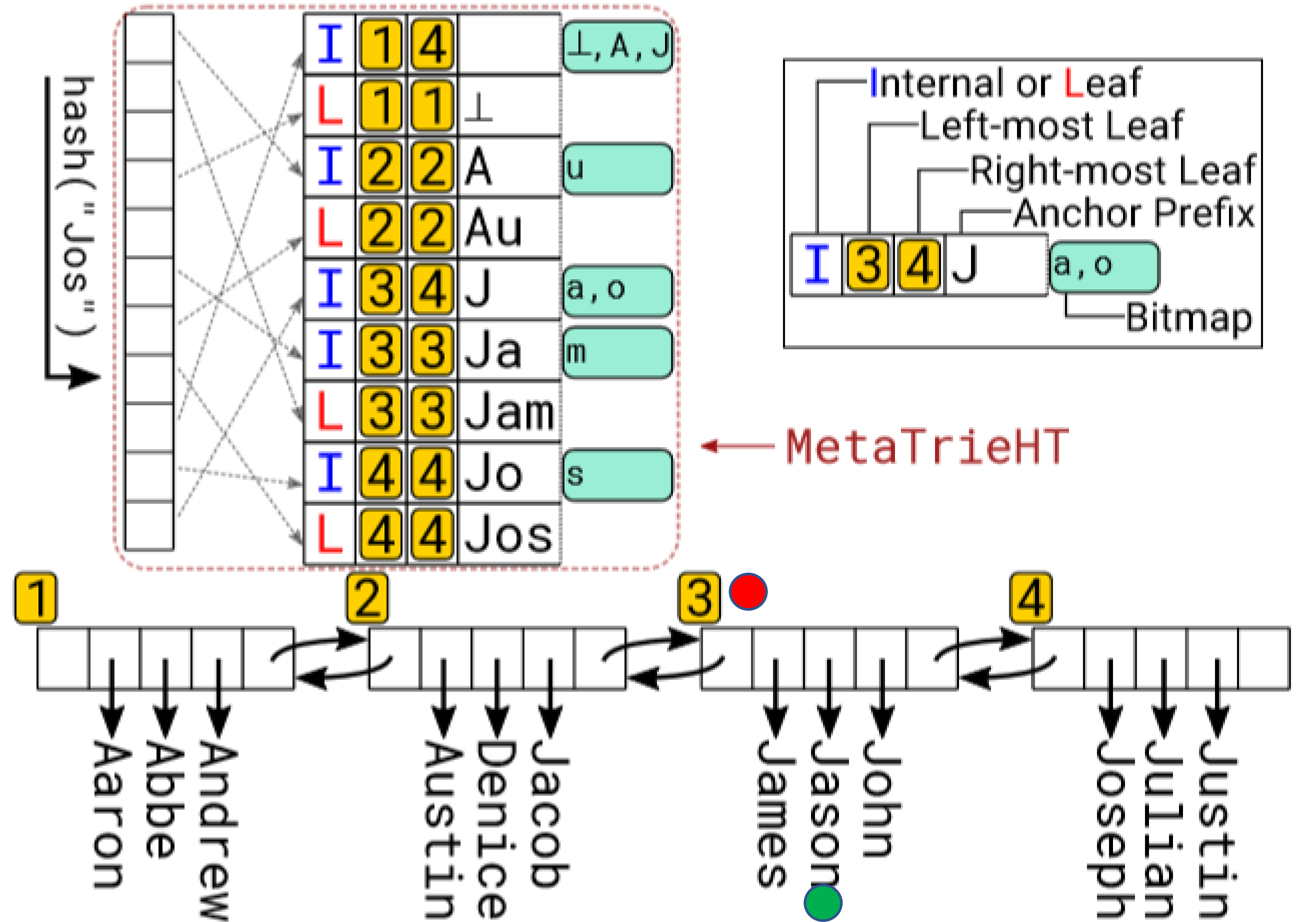
Wormhole [1]

```
function searchTrieHT(wh, key)
    node ← searchLPM(wh.ht, key, min(key.len, wh.maxLen))
    if node.type = LEAF then return node
    else if node.key.len = key.len then
        leaf ← node.leftmost
        if key < leaf.anchor then leaf ← leaf.left
        return leaf
    missing ← key.tokens[node.key.len]
    sibling ← findOneSibling(node.bitmap, missing)
    child ← htGet(wh.ht, concat(node.key, sibling))
    if child.type = LEAF then
        if sibling > missing then child ← child.left
        return child
    else
        if sibling > missing then return child.leftmost.left
        else return child.rightmost
```



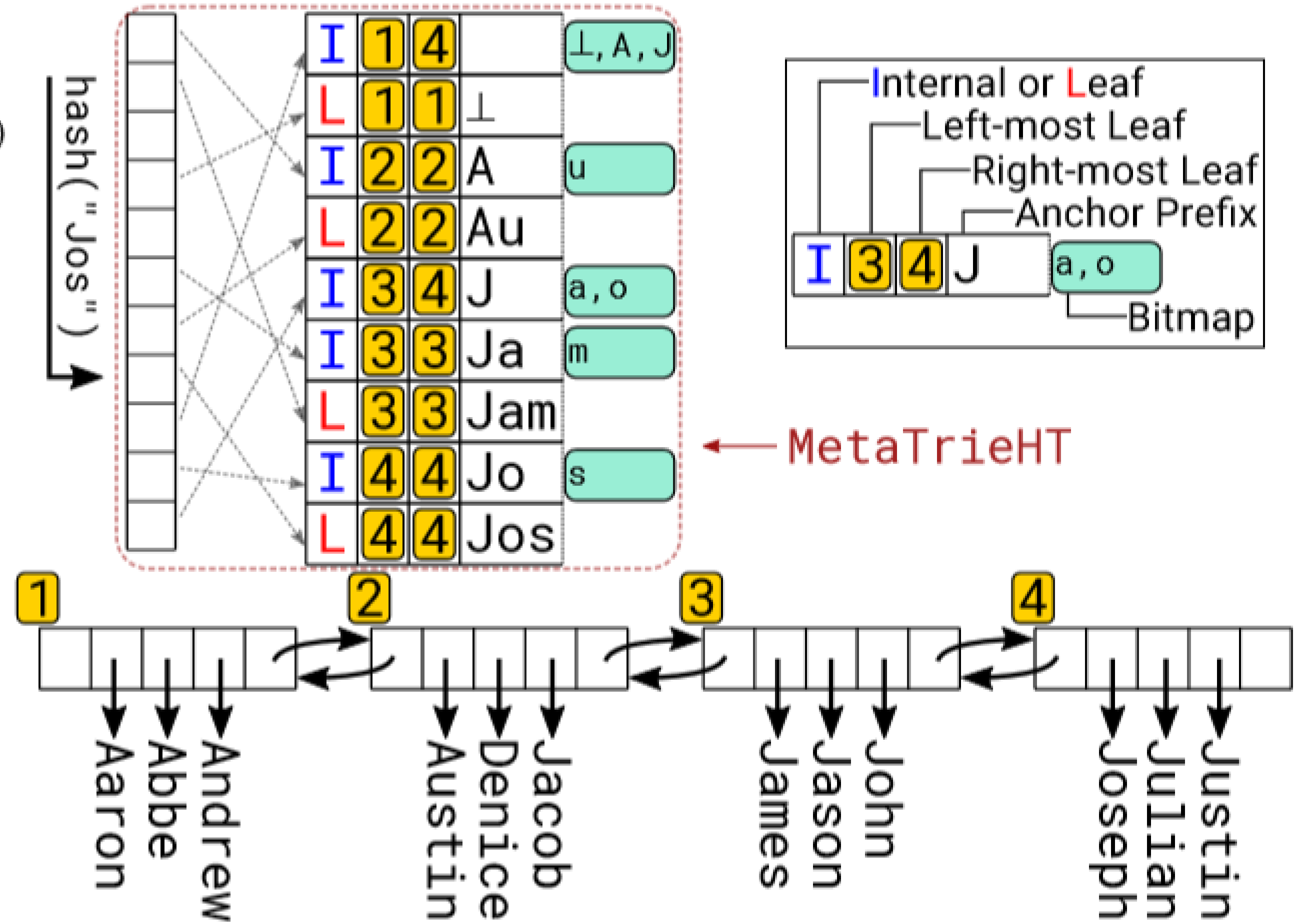
Wormhole [1]

function GET(wh, key)
  leaf ← searchTrieHT(wh, key);   i ← pointSearchLeaf(leaf, key)
  if (i < leaf.size) and (key = leaf.hashArray[i].key) **then**
    return leaf.hashArray[i]
  else return NULL



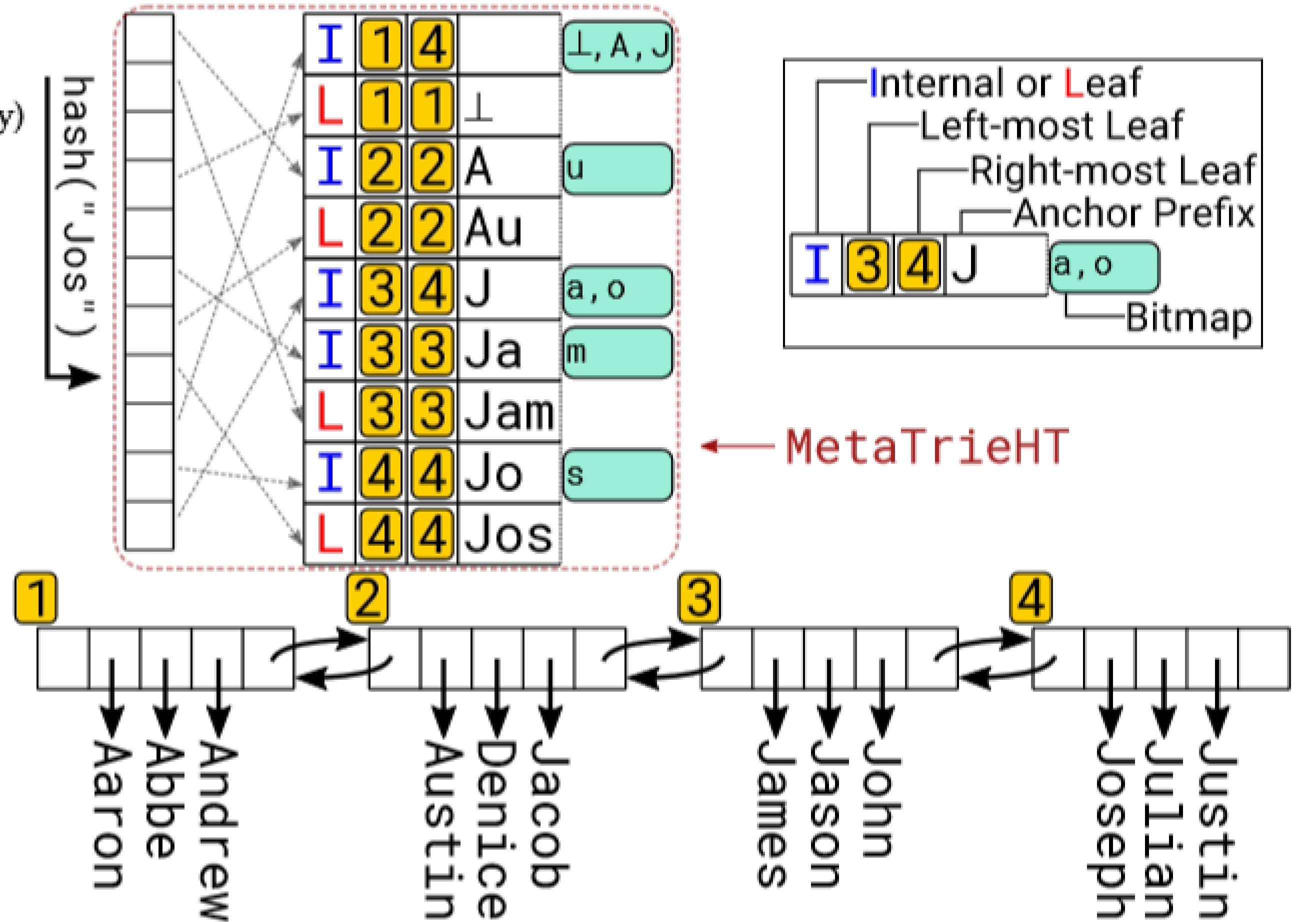
Wormhole [1]

```
function DEL(wh, key)
    leaf ← searchTrieHT(wh, key);        i ← pointSearchLeaf(leaf, key)
    if (i < leaf.size) and (key = leaf.hashArray[i].key) then
        leafDelete(leaf, i)
        if (leaf.size + leaf.left.size) < MergeSize then
            merge(wh, leaf.left, leaf)
        else if (leaf.size + leaf.right.size) < MergeSize then
            merge(wh, leaf, leaf.right)
```

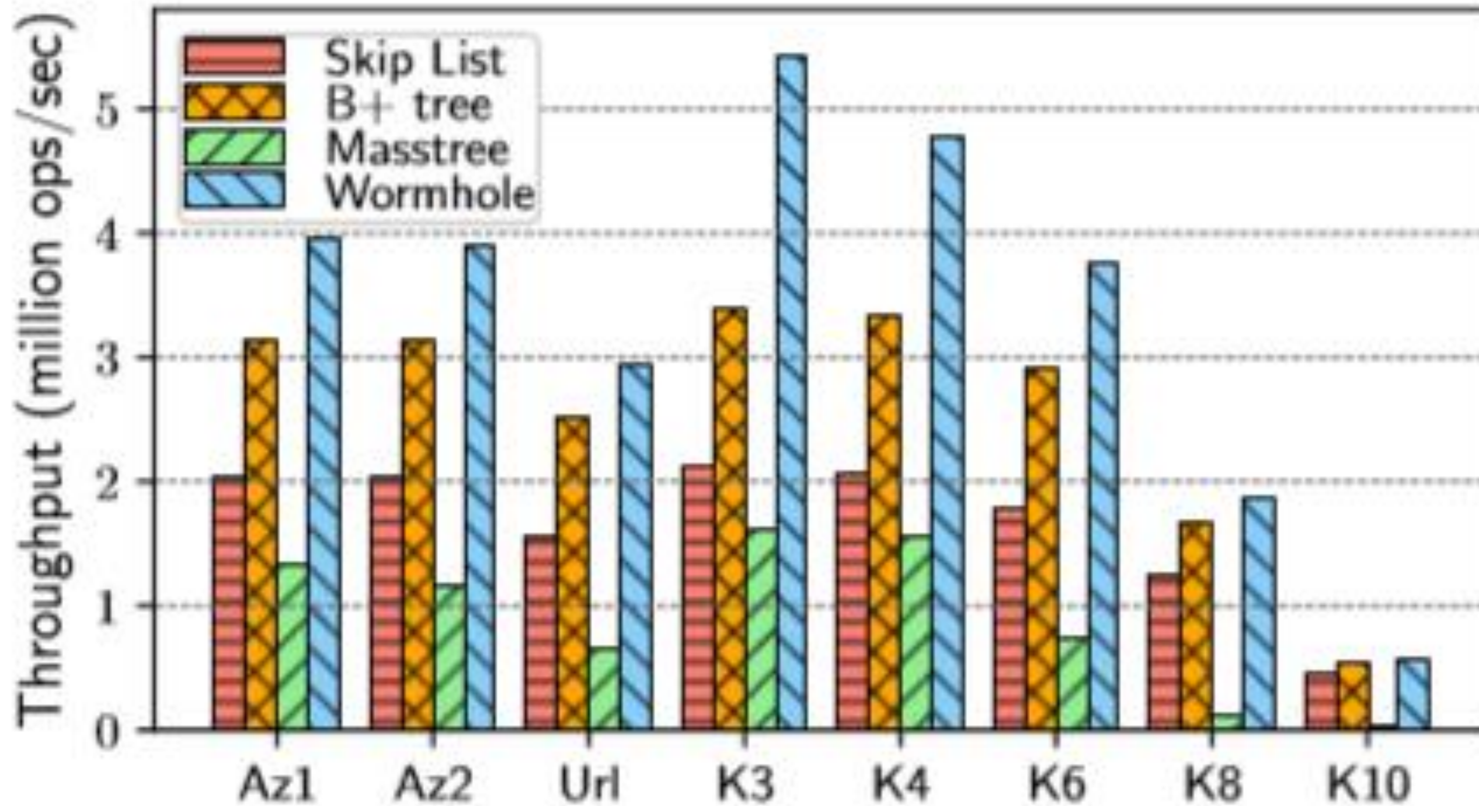


Wormhole [1]

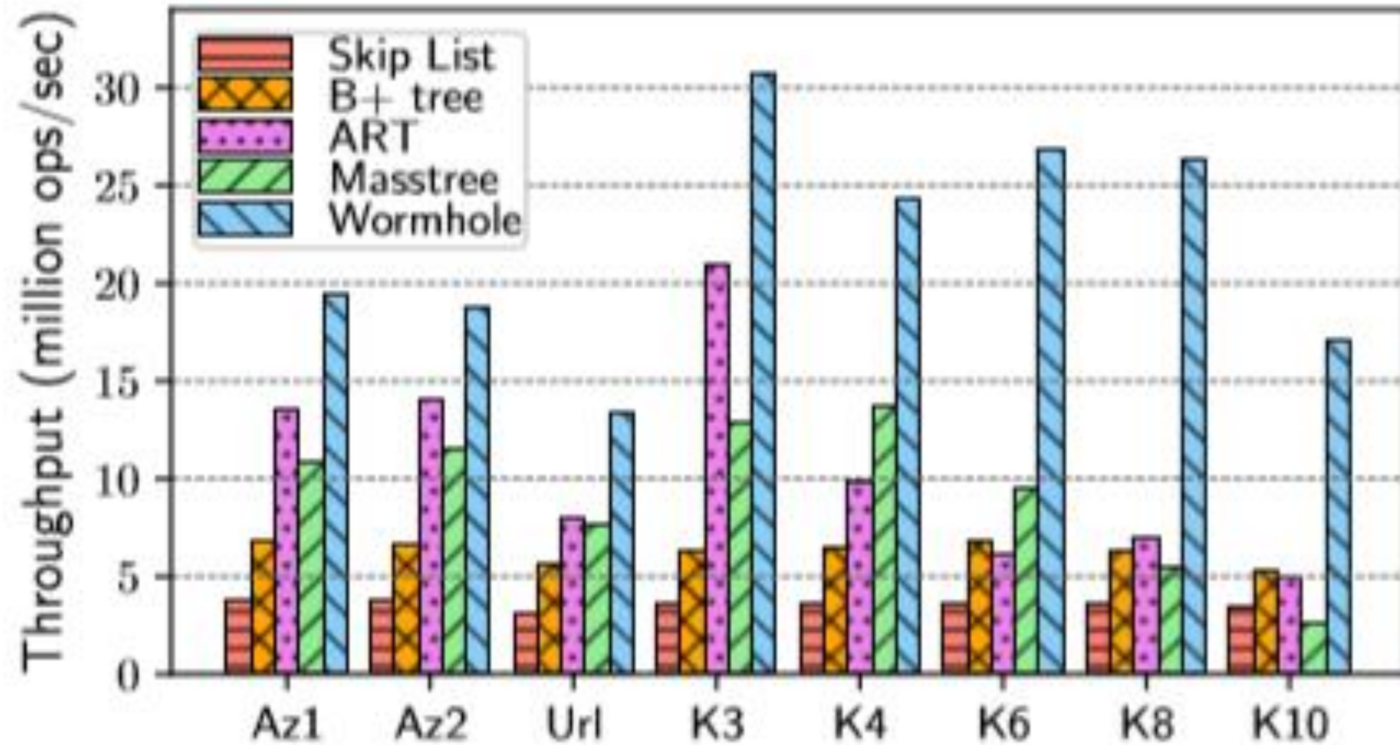
```
function SET(wh, key, value)
    leaf ← searchTrieHT(wh, key);        i ← pointSearchLeaf(leaf, key)
    if (i < leaf.size) and (key = leaf.hashArray[i].key) then
        leaf.hashArray[i].value ← value
    else
        if leaf.size = MaxLeafSize then
            left, right ← split(wh, leaf)
            if key < right.anchor then
                leaf ← left
            else leaf ← right
        leafInsert(leaf, key, value)
```



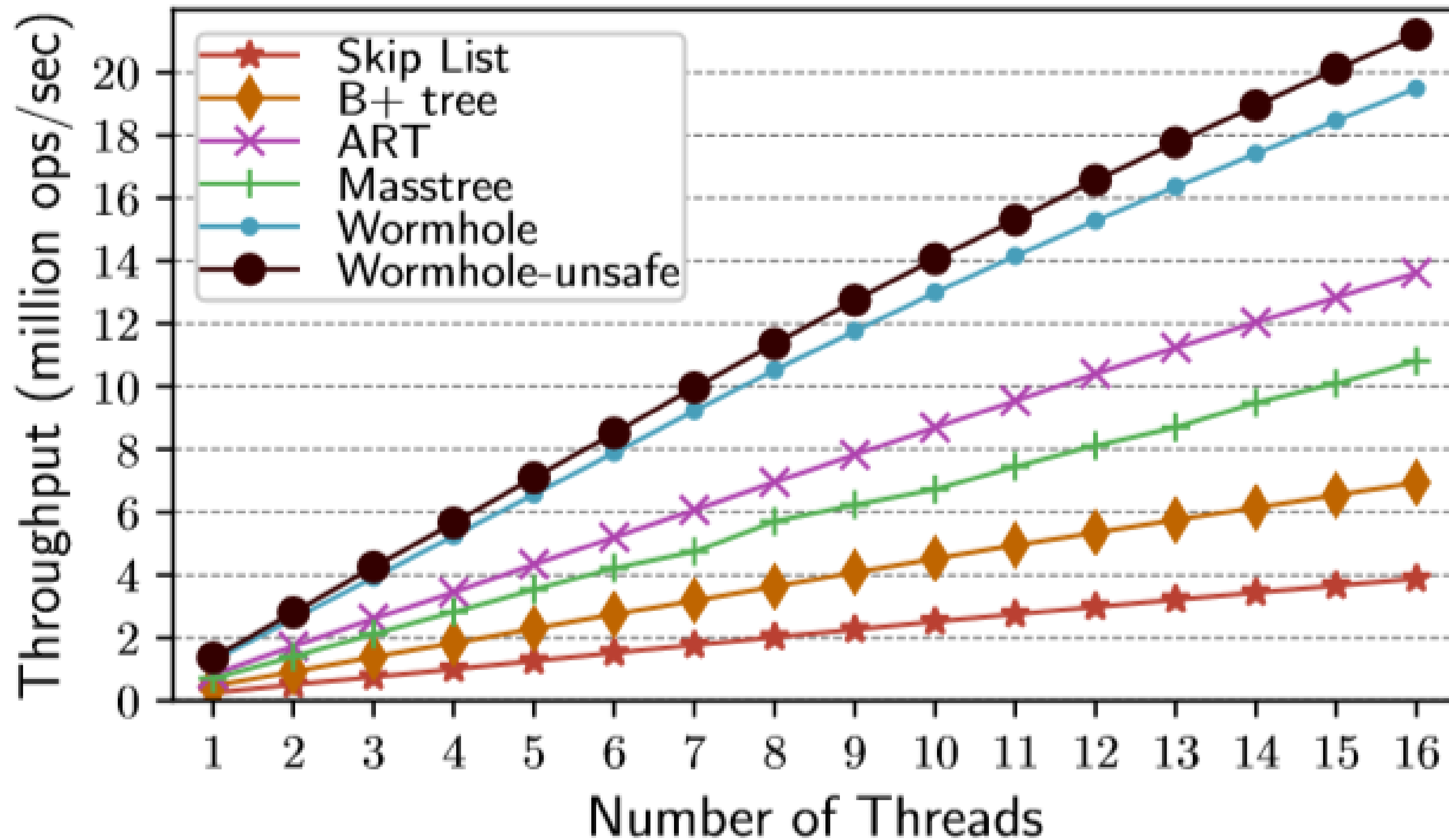
Throughput of range lookups [1]



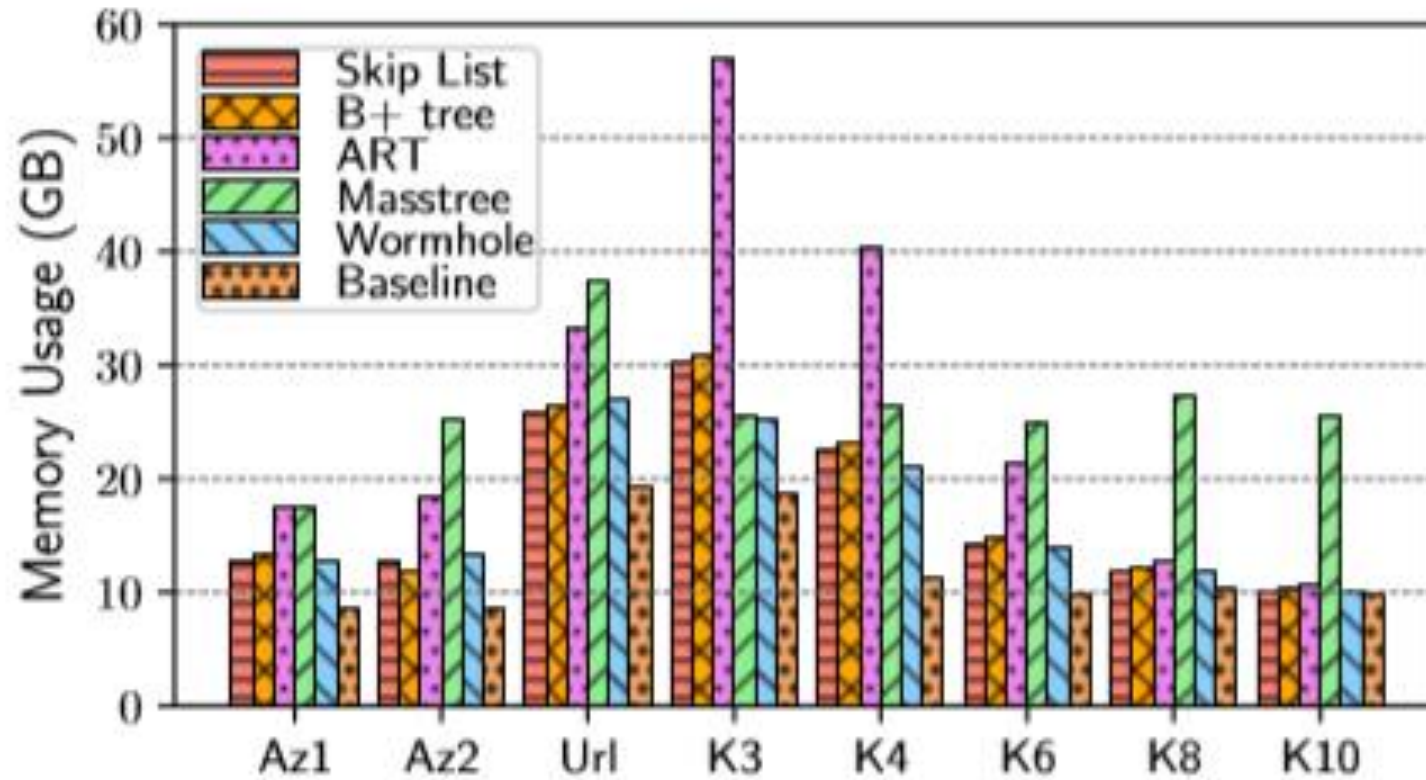
Lookup throughput on local CPU [1]



Lookup throughput with threads [1]



Memory usage of the indexes [1]



Zusammenfassung

	GET	SET	DEL	GET_RANGE
Hashtabellen	O(1)	O(1)	O(1)	nein
B+Baum	O(log N)	O(log N)	O(log N)	O(log N + k)
Präfix Baum	O(L)	O(L)	O(L)	ja
Wormhole	O(log L)	O(log L)	O(log L)	O(log L + k)

Zusammenfassung

- Aus B+, Trie und Hash Table neue Index Struktur zsm gebaut
- GET, SET, DEL(, range search) in O(log L)
- Platzkomplexität vergleichbar mit B+ Baum
- Niedrige Zeitkomplexität

Quellen

1. Paper: Wormhole: A Fast Ordered Index for In-memory Data Management
2. AD Folien WS 18/19
3. GDB Folien WS 18/19
4. https://www.geeksforgeeks.org/trie-insert-and-search/
5. https://www.geeksforgeeks.org/advantages-trie-data-structure/