

Lecture 3: Access Methods



Logistics

- Turing Point quiz details
- Intro sheet (due on Gradescope)
- Assignment 1 (External sorting, due on Gradescope)

Recap

- Storage Management
- Smart Pointers and Pages
- Slotted Page
- Buffer Management
- 2Q Policy

Lecture Overview

- Indexing
- Hash Table
- Double Hashing
- Range Query
- B+Tree
- Trie
- RTree

Periodic Table



Periodic Table of System Design Principles

Towards a Periodic Table of Computer System Design Principles

JOY ARULRAJ, Georgia Institute of Technology

System design is often taught through domain-specific solutions specific to particular domains, such as databases, operating systems, or computer architecture, each with its own methods and vocabulary. While this diversity is a strength, it can obscure cross-cutting principles that recur across domains. This paper proposes a preliminary “periodic table” of system design principles distilled from several domains in computer systems. The goal is a shared, concise vocabulary that helps students, researchers, and practitioners reason about structure and trade-offs, compare designs across domains, and communicate choices more clearly. For supporting materials and updates, please refer to the repository at: <https://github.com/jarulraj/periodic-table>.

<https://github.com/jarulraj/periodic-table>
<https://news.ycombinator.com/>



Periodic Table of System Design Principles

Towards a Periodic Table of Computer System Design Principles 3

Str	Eff	Sem	Dist	Plan	Oper	Rel	Sec
Si	Sc	Al	Lt	Ep	Ad	Ft	Sy
Mo	Rc	Lu	Dc	Cm	Ec	Is	Ac
Co	Wv	Se	Fp	Cp	Wa	At	Lp
Ex	Cc	Fs	Lo	Gd	Au	Cr	Tq
Pm	Bo	Ig		Bb	Ho		Cf
Gr	Ha			Ah	Ev		Sa
	Op						
	La						

Periodic Table of System Design Principles

-  **Mo – Modularity**
- Partition the system into cohesive units with minimal interfaces, so that each unit can be reasoned about, replaced, or evolved independently. This principle focuses on decomposition: choosing boundaries to favor clear separation of concerns so that each responsibility sits in one module.
- **Example:** The OSI model decomposes communication into standardised layers with well-defined boundaries that permit independent development and substitution [48].

Periodic Table of System Design Principles

- **■ Co – Composability**
- Design components that can be safely and flexibly recombined; rely on explicit contracts and type-constrained interfaces so that every legal composition remains correct, letting components be assembled like interchangeable bricks. Unlike modularity, this principle focuses on re-composition: making sure the components can be combined safely and flexibly.
- **Example:** Unix programs (e.g., grep, sort, uniq) read from stdin and write to stdout, letting the user compose complex text processing pipelines [41].

Periodic Table of System Design Principles

-  **Pm – Policy/Mechanism Separation**
- Separate what should be done (policy) from how it is carried out (mechanism) by exposing a common interface through which multiple policies can plug into the same mechanism.
- **Example:** Hydra has a kernel of generic mechanisms (scheduling, paging, protection) and moved resource-allocation policies to user-level modules [32].

Index



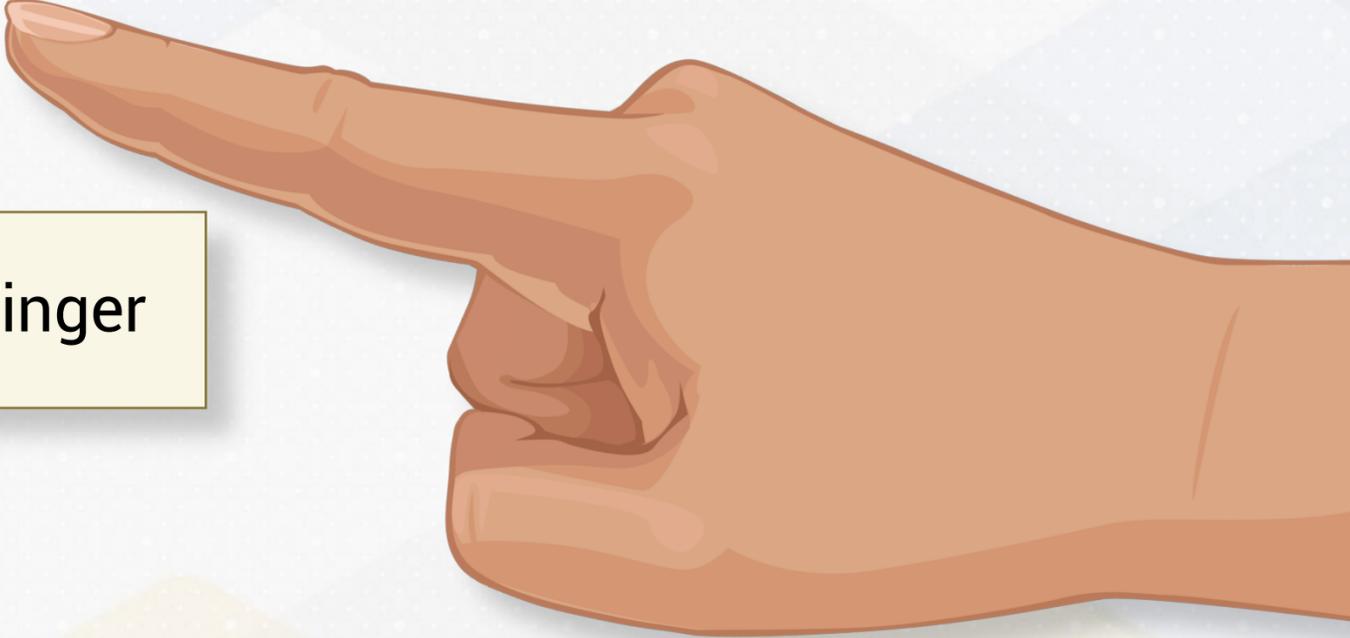
Indexing in Database Systems

INDEX

Klein, Arno	249
La Duke, Elliott W.	377
Lawless, Frank R.	307
Layer, John G.	330
Leffel, John C.	357
Leffel, Edward	359
Leonard, Frederick P.	371
Lewis, Edward	338
Lewis, Frank E.	290
Lewis, James R.	327
Lewis, Thompson P.	339
Llewelyn, Edgar J.	331
....., George T.	334
....., Francis B.	298
....., Enoch W.	387
....., James F.	389
....., Noah	389
McReynolds, Samuel M.	296
MacGregor, Francis B.	244
Macy, Carlos B.	287
Marvel, Alexander L.	400
Marvel, Thomas	397
Meinschein, Conrad	338
Menzies, G. V.	217
Menzies, Winston	218
Miller, Lorenz C.	385
Moeller, John H.	256
Montgomery, Samuel B.	282
Morrow, Lannie G.	350
Moye, James H.	362

Book Index

Index Finger

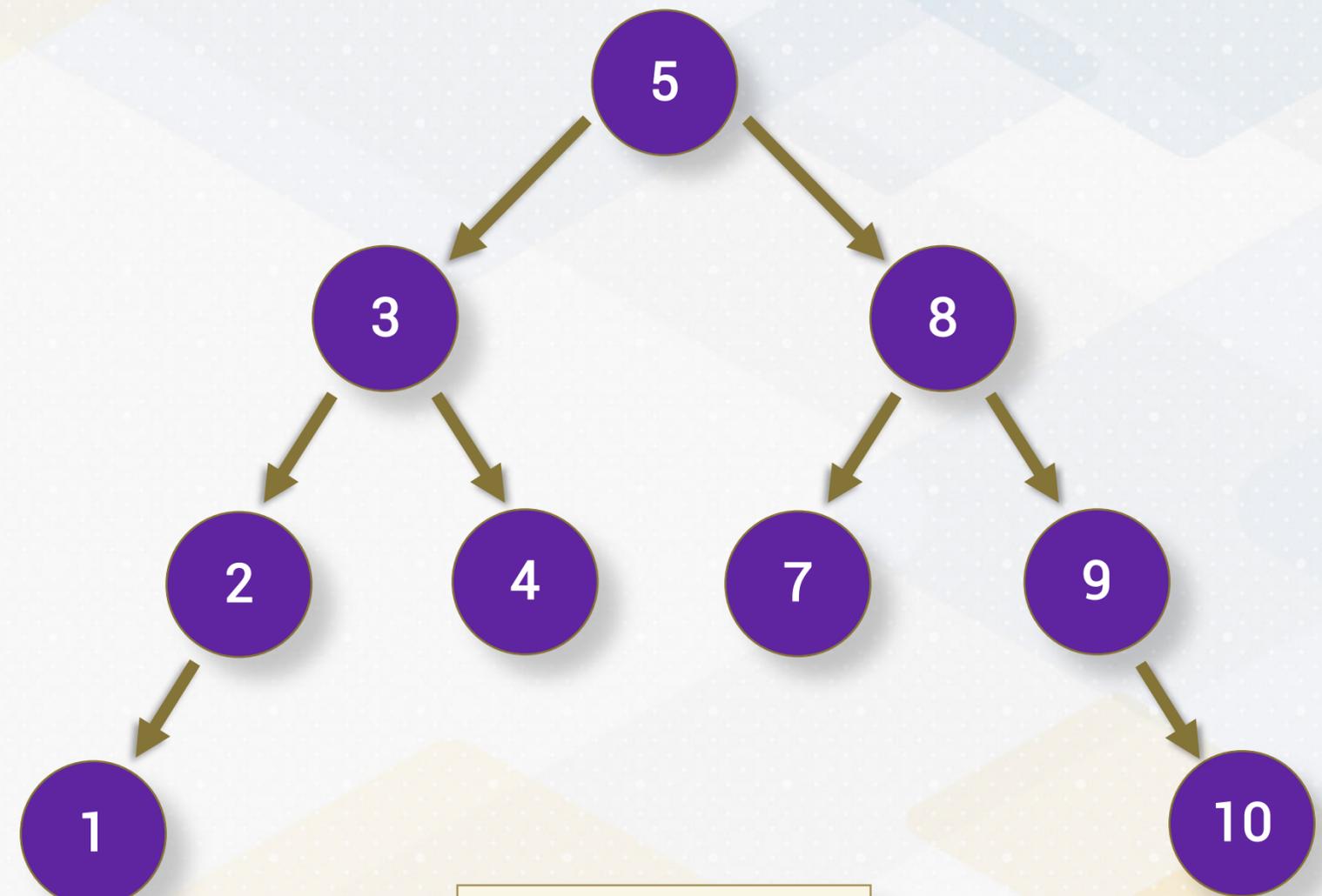


Indiana State Library and Historical Bureau, Public domain, via Wikimedia Commons



Indexing in C++: ordered_map

Key	Value
1	Apple
2	Banana
3	Cherry
4	Date
5	Elderberry
6	Fig
7	Grape
8	Honeydew
9	Ilama
10	Jackfruit



$O(\log N)$

Indexing in C++: unordered_map

Key	Value
2	Banana
6	Fig
1	Apple
3	Cherry
4	Date
5	Elderberry
8	Honeydew
7	Grape
10	Jackfruit
9	Ilama

Avg Case

$O(1)$

Worst Case

$O(N)$

map vs unordered_map

	MAP	UNORDERED MAP
Ordering	Sorted based on key	Not sorted based on key
Implementation	Binary Search Tree	Hash Table
Average Case Time	$O(\log N)$	$O(1)$
Worst Case Time	$O(\log N)$	$O(N)$

HashIndex in BuzzDB

insertOrUpdate

getValue

```
class HashIndex {  
private:  
    std::unordered_map<int, int> hash_index;  
public:  
    void insertOrUpdate(int key, int value) {  
        hash_index[key] = value;  
    }  
    int getValue(int key) const {  
        auto it = hash_index.find(key);  
        return it != hash_index.end() ? it->second : -1;  
    }  
};
```

Integration of HashIndex in BuzzDB

During the scanning phase, each tuple's key-value pair is inserted into the HashIndex.

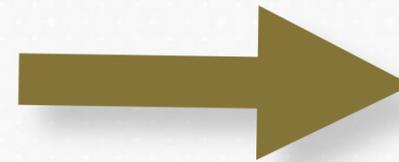
```
void BuzzDB::scanTableToBuildIndex() {  
    // Iterating over pages and tuples  
    for (size_t page_itr = 0; page_itr < num_pages; page_itr++) {  
        // Extract key-value pairs and insert them into the index  
        int key = loadedTuple->fields[0]->asInt();  
        int value = loadedTuple->fields[1]->asInt();  
  
        index.insertOrUpdate(key, value);  
    }  
}
```

Custom HashTable

Hash Index Class
(this version)

Hash Index Class
(next version)

`std::unordered map`



Custom Hash Table

Hash Table



Hash Table

Think of a hash table like a cabinet of drawers.
Each drawer can hold a piece of paper with the **key** and **associated value**.

0	1	2	3	4	5	6	7	8	9
					15				
					Apple				



DIRECT ACCESS USING HASH FUNCTION

Inserting Key-Value Pair

Insert (15, Apple)

0	1	2	3	4	5	6	7	8	9
					15				
					Apple				



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 15 \% 10 = 5 \text{ (since } 15 = 10 * 1 + 5 \text{)}$$

Inserting Key-Value Pair

Insert (26, Grape)

0	1	2	3	4	5	6	7	8	9
					15	26			
					Apple	Grape			



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 26 \% 10 = 6 \text{ (since } 26 = 10 * 2 + 6)$$

Inserting Key-Value Pair with Collision

Insert (56, Kiwi)

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 56 \% 10 = 6$$

Finding Key-Value Pair

Find value associated with key 15 = Apple

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 15 \% 10 = 5$$

Finding Key-Value Pair

Find value associated with key 56 = Kiwi

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 56 \% 10 = 6$$

Finding Key-Value Pair

Find value associated with key 86 = KEY NOT FOUND

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 86 \% 10 = 6$$

Hash Function



Hash Function

Hash function maps keys to slots in the hash table

```
size_t hashFunction(int key) {  
    return key % totalSlots; // Our simple formula  
}
```

$$\text{HASH}(\text{KEY}) = \text{KEY} \% \text{TABLE_SIZE}$$

Collision Handling

Linear Probing

Move to the next available slot until we find one empty.

0	1	2	3	4	5	6	7	8	9
80	11				15	26	56	18	79
Lime	Date				Apple	Grape	Kiwi	Fig	Pear

If slot 8 is taken, check slot 9, then 0, 1, and so on.

"Linear" Probing

Formula to find the index of the next slot I after K "probes" is "linear" with respect to K.

$$\text{INDEX } I = (\text{HASH}(\text{KEY}) + K) \% \text{TABLE_SIZE}$$

$$\text{HASH}(\text{KEY}) = \text{KEY} \% \text{TABLE_SIZE}$$

HashEntry Struct

HashEntry Struct

Represents a piece of paper containing **key** and **value**.

```
struct HashEntry {  
    int key, value;  
    HashEntry(int k, int v) : key(k), value(v) {}  
};
```

HashIndex

HashTable is simply an array of drawers. Each drawer can either hold a `HashEntry` or be empty (indicating no value).

```
class HashIndex {  
    std::vector<std::optional<HashEntry>> hashTable;  
  
    static const size_t capacity = 100; // Hard-coded capacity  
    HashIndex() { hashTable.resize(capacity); }  
};
```

Inserting Key-Value Pair

insertOrUpdate

Uses linear probing to find the next available slot for insertion or locate an existing key for updating.

```
void insertOrUpdate(int key, int value) {
    size_t index = hashFunction(key);
    do {
        if (!hashTable[index]) {
            hashTable[index] = HashEntry{key, value, true}; // Insert new entry
            break;
        } else if (hashTable[index]->key == key) {
            hashTable[index]->value += value; // Update existing entry
            break;
        }
        index = (index + 1) % capacity; // Linear probing to next slot
    } while (index != originalIndex); // Returned to starting point
}
```

Finding Key-Value Pair

getValue

The retrieval process employs linear probing as well to navigate through potential collision sequences.

```
int getValue(int key) const {
    size_t index = hashFunction(key);
    do {
        if (hashTable[index] && hashTable[index]->key == key) {
            return hashTable[index]->value; // Key found
        }
        index = (index + 1) % capacity; // Continue probing
    } while (index != originalIndex);
    return -1; // Key not found
}
```

Operation Complexity

N = Number of key-value pairs in the hash table.

	NO/FEW COLLISIONS	LOTS OF COLLISIONS
INSERT	$O(1)$	$O(N)$
FIND	$O(1)$	$O(N)$

Clustering Problem with High Collision Rates

Bad Hash Function: $\text{HASH}(\text{KEY}) = 5$

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$\text{HASH}(\text{KEY}) = 5$

Quadratic Probing



Quadratic Probing

Formula to find the index of the next slot I after K “probes” is “quadratic” with respect to K .

$$\text{INDEX } I = (\text{HASH}(\text{KEY}) + K^2) \% \text{TABLE_SIZE}$$

$$\text{HASH}(\text{KEY}) = \text{KEY} \% \text{TABLE_SIZE}$$

Inserting 2

Insert (2, Apple)

0	1	2	3	4	5	6	7	8	9
		2							
		Apple							



$$\text{HASH}(2) = 2; K = 0; \text{INDEX } I = (2 + 0^2) \% 10 = 2$$

Inserting 12

Insert (12, Grape)

0	1	2	3	4	5	6	7	8	9
		2	12						
		Apple	Grape						



$$\text{HASH}(12) = 2; K = 1; \text{INDEX } I = (2 + 1^2) \% 10 = 3$$

Inserting 22

Insert (22, Kiwi)

0	1	2	3	4	5	6	7	8	9
		2	12			22			
		Apple	Grape			Kiwi			



$$\text{HASH}(22) = 2; K = 2; \text{INDEX } I = (2 + 2^2) \% 10 = 16 \% 10 = 6$$

Inserting 32

Insert (32, Fig)

0	1	2	3	4	5	6	7	8	9
	32	2	12			22			
	Fig	Apple	Grape			Kiwi			



$$\text{HASH}(32) = 2; K = 3; \text{INDEX } I = (2 + 3^2) \% 10 = 11 \% 10 = 1$$

Quadratic Probing

Probe sequence follows a quadratic formula.

```
void insertOrUpdate(int key, int value) {
    size_t originalIndex = hashFunction(key);
    bool inserted = false;
    int i = 0; // Attempt counter
    do {
        if (!hashTable[index].exists) {
            ...
        } else if (hashTable[index].key == key) {
            ...
        }
        i++;
        index = (originalIndex + i * i) % capacity; // Quadratic probing
    } while (index != originalIndex && !inserted);
}
```

Benefits of Quadratic Probing

Better spread of keys; higher search efficiency

0	1	2	3	4	5	6	7	8	9
	32	2	12			22			
	Fig	Apple	Grape			Kiwi			

Limitation of Quadratic Probing

Secondary clustering of all keys hashed to 2.

0	1	2	3	4	5	6	7	8	9
	32	2	12			22			
	Fig	Apple	Grape			Kiwi			

Double Hashing



Double Hashing

Double hashing

Uses a secondary hash function to calculate the probe step, offering a unique probe sequence for each key.

$$\text{INDEX } I = (\text{HASH1}(\text{KEY}) + K * \text{HASH2}(\text{KEY})) \% \text{TABLE_SIZE}$$

$$\text{HASH1}(\text{KEY}) = \text{KEY} \% \text{TABLE_SIZE}$$

$$\text{HASH2}(\text{KEY}) = 1 + \text{KEY} \% (\text{TABLE_SIZE} - 1)$$

Inserting 2

Insert (2, Apple)

0	1	2	3	4	5	6	7	8	9
		2							
		Apple							



$$\text{HASH1}(2) = 2; \text{HASH2}(2) = 1 + (2 \% 9) = 1 + 2 = 3$$
$$K = 0; \text{INDEX } I = (2 + 0 * 3) \% 10 = 2$$

Inserting 12

Insert (12, Grape)

0	1	2	3	4	5	6	7	8	9
		2				12			
		Apple				Grape			



$$\text{HASH1}(12) = 2; \text{HASH2}(12) = 1 + (12 \% 9) = 1 + 3 = 4$$
$$K = 1; \text{INDEX } I = (2 + 1 * 4) \% 10 = 6 \% 10 = 6$$

Inserting 22

Insert (22, Kiwi)

0	1	2	3	4	5	6	7	8	9
		2				12	22		
		Apple				Grape	Kiwi		



$$\text{HASH1}(22) = 2; \text{HASH2}(22) = 1 + (22 \% 9) = 1 + 4 = 5$$
$$K = 1; \text{INDEX } I = (2 + 1 * 5) \% 10 = 7 \% 10 = 7$$

Inserting 32

Insert (32, Fig)

0	1	2	3	4	5	6	7	8	9
		2				12	22	32	
		Apple				Grape	Kiwi	Fig	



$$\text{HASH1}(32) = 2; \text{HASH2}(32) = 1 + (32 \% 9) = 1 + 5 = 6$$
$$K = 1; \text{INDEX } I = (2 + 1 * 6) \% 10 = 8 \% 10 = 8$$

Benefit of Double Hashing

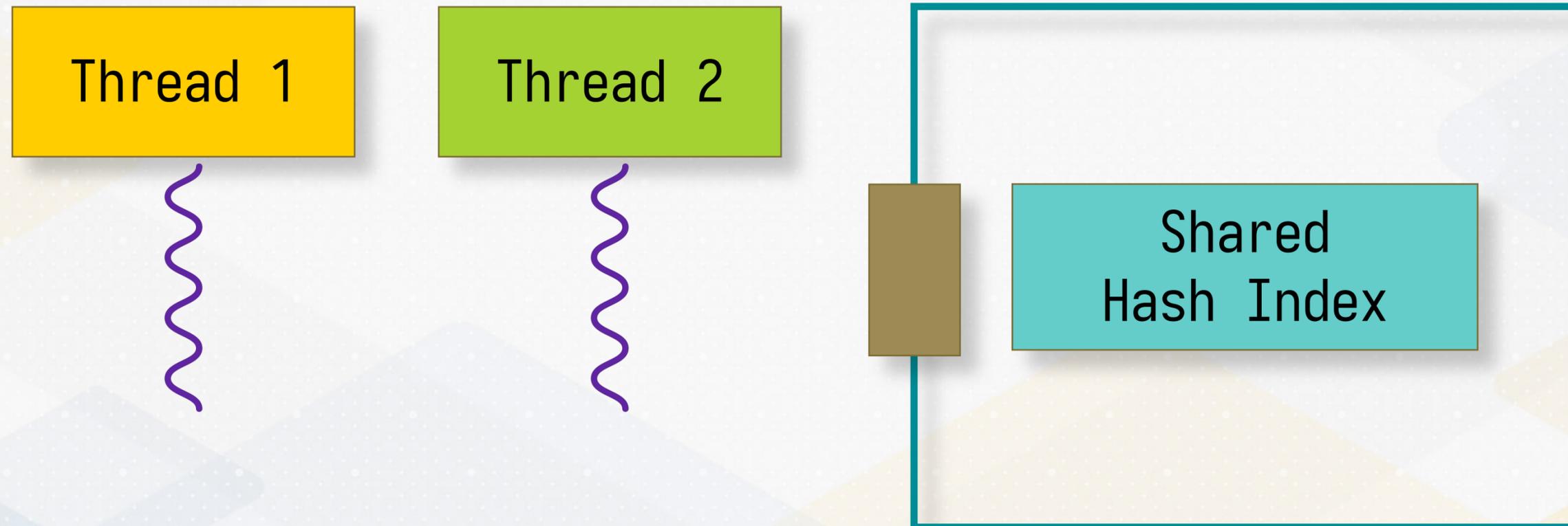
0	1	2	3	4	5	6	7	8	9
		2				12	22	32	
		Apple				Grape	Kiwi	Fig	

Parallel Index Construction



Parallel Index Construction

With multi-core CPUs, parallelizing index construction offers a significant performance boost by **distributing the workload across multiple threads.**

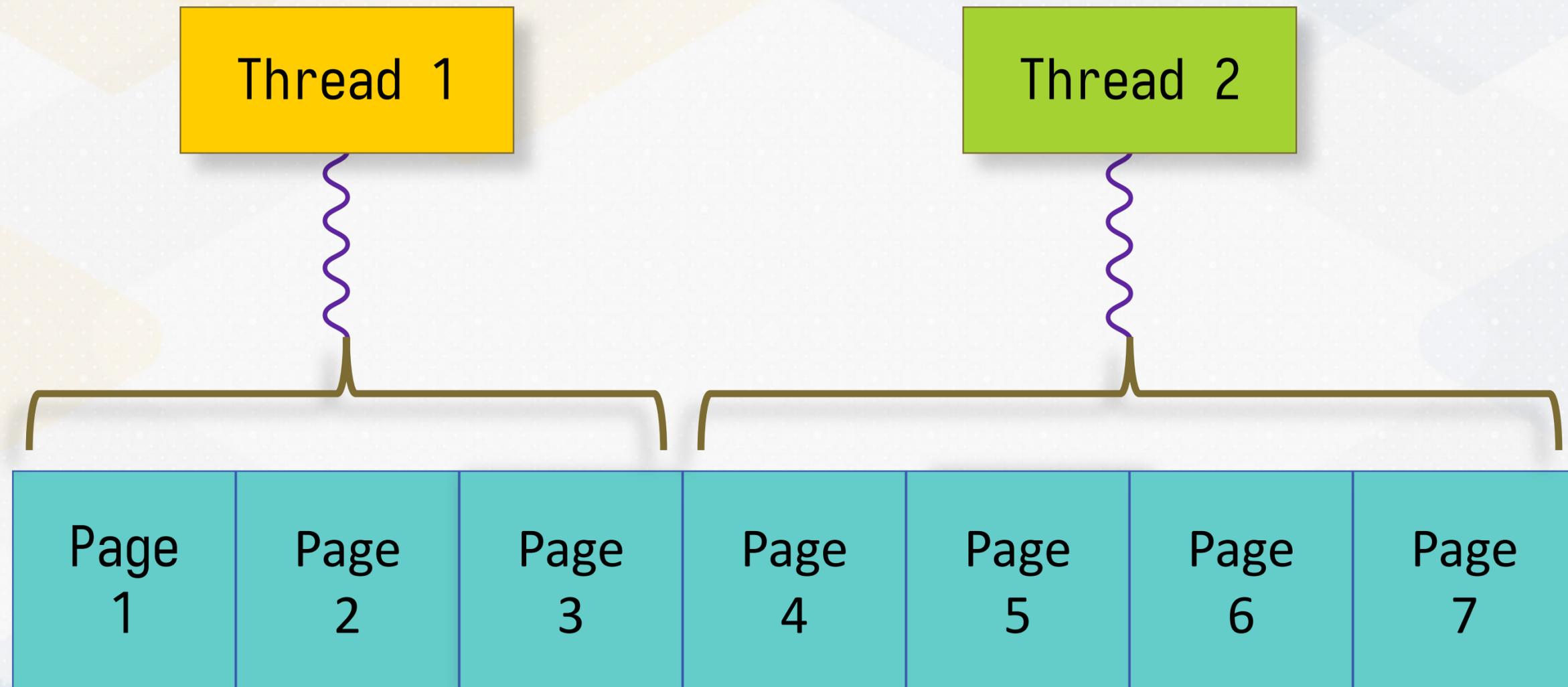


Page Assignment

Divide the total number of pages (`num_pages`) by the number of available threads (`num_threads`), assign each thread a specific range of pages to process.

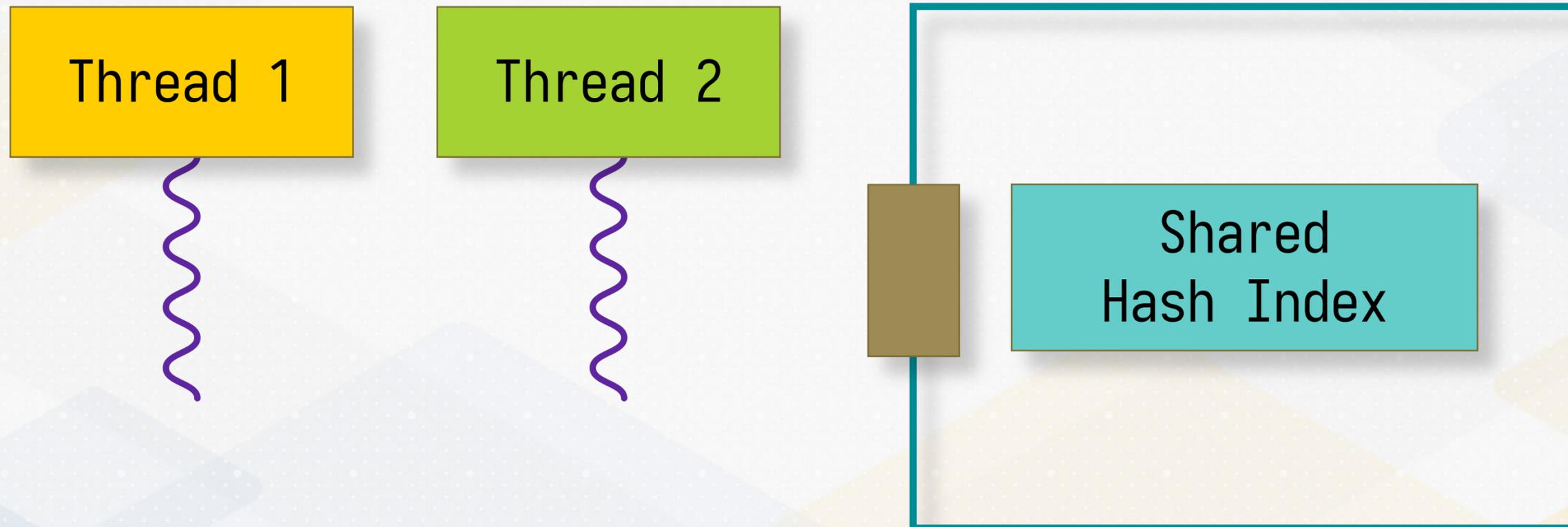
```
void parallelProcessPages(size_t num_threads = 5) {
    auto num_pages = buffer_manager.getNumPages();
    size_t pages_per_thread = num_pages / num_threads;
    std::vector<std::thread> threads;
    for (size_t i = 0; i < num_threads; i++) {
        size_t start_page = i * pages_per_thread;
        size_t end_page = ...; // Last thread gets any remaining pages
        threads.emplace_back(&BuzzDB::processPageRange, this, start_page,
end_page);
    }
    ...
}
```

Page Assignment



Thread Safety

With parallel index construction, the challenge is that concurrent operations on the index by multiple threads may lead to inconsistent state.



Thread Safety

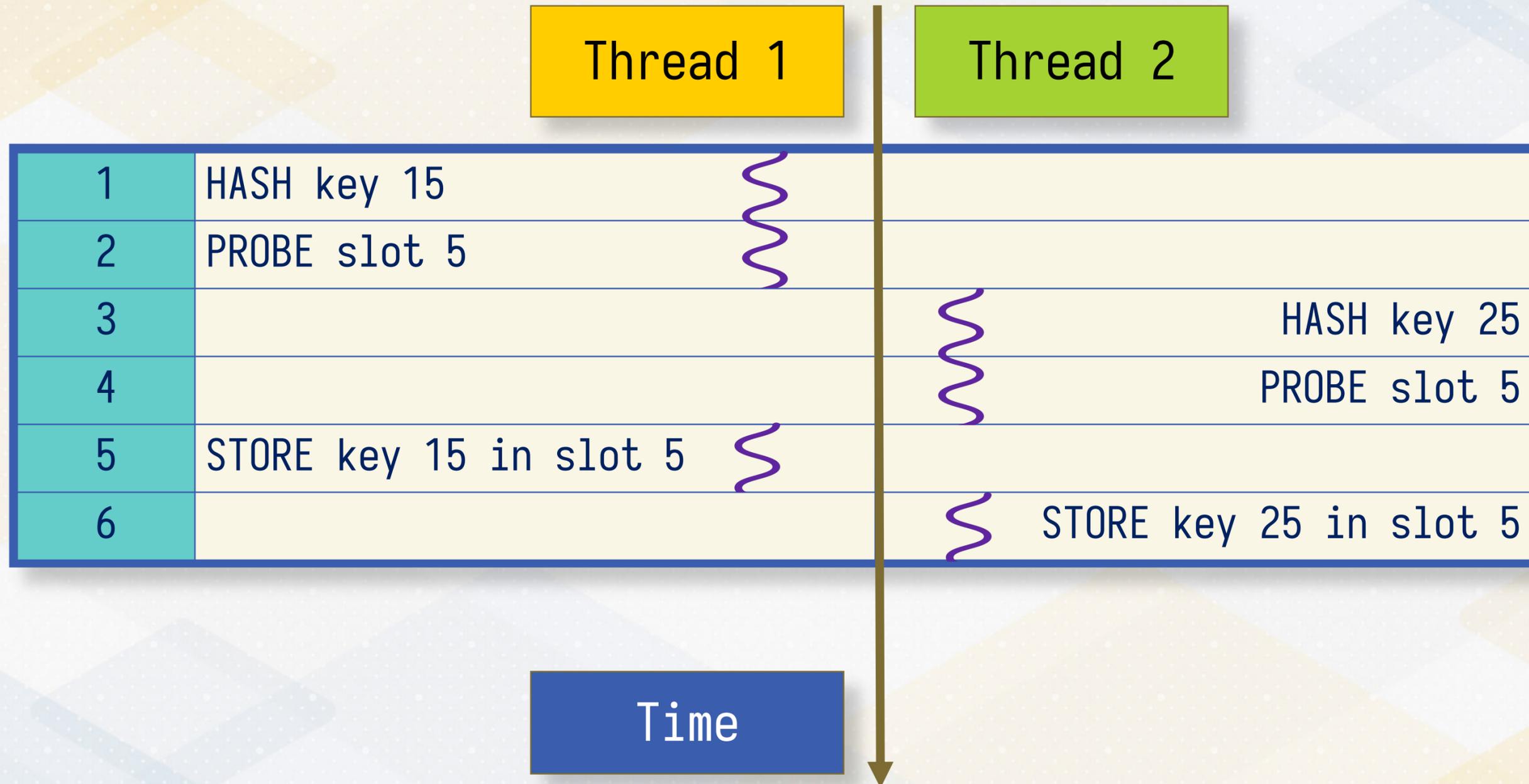
Thread 1: Insert (15, Fig) **Thread 2:** Insert (25, Pear)

0	1	2	3	4	5	6	7	8	9
					25				
					Pear				



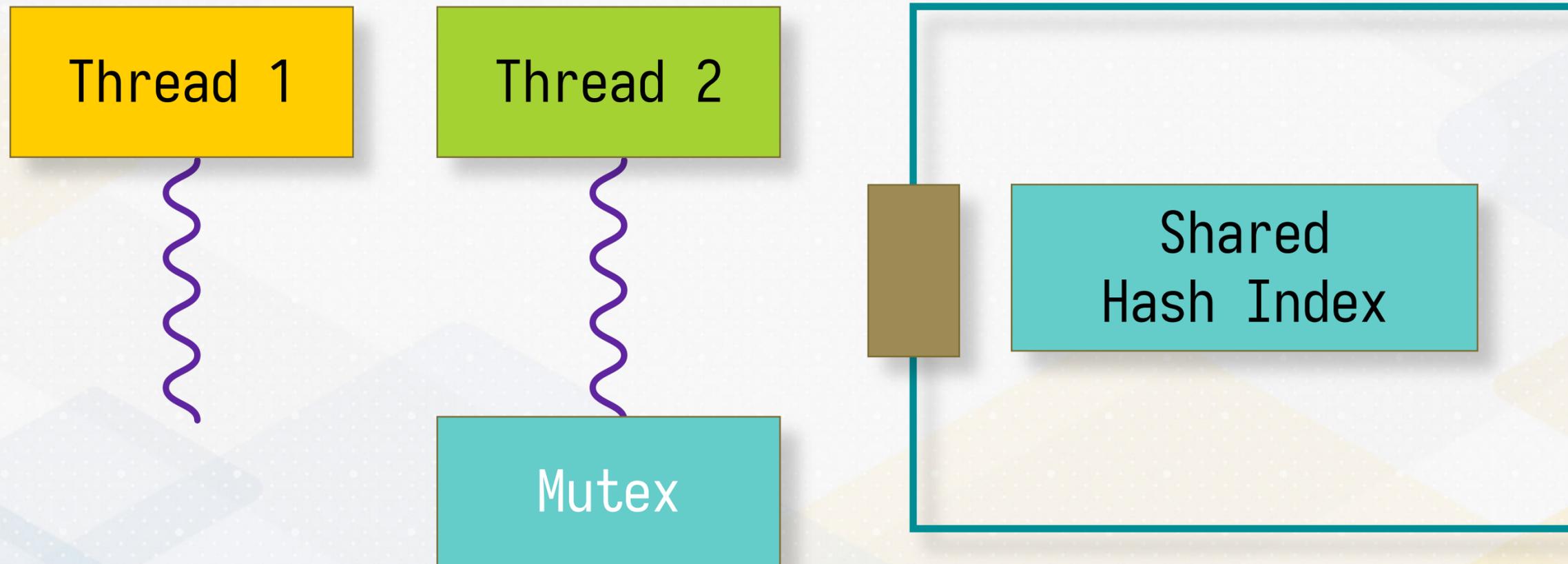
$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10$$

Race Condition



std::mutex

Mutex "serializes" access to the shared index structure.

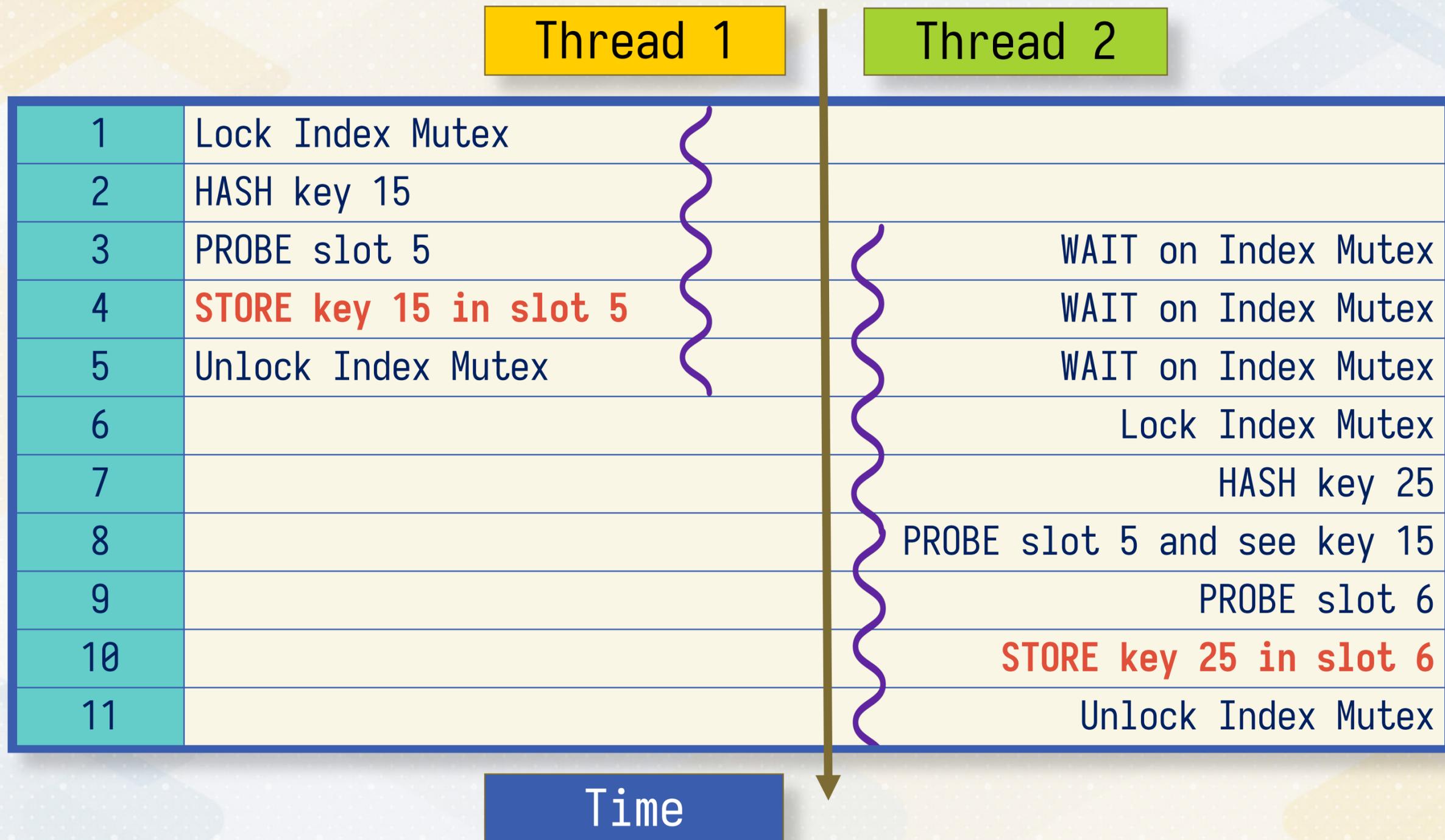


std::lock_guard

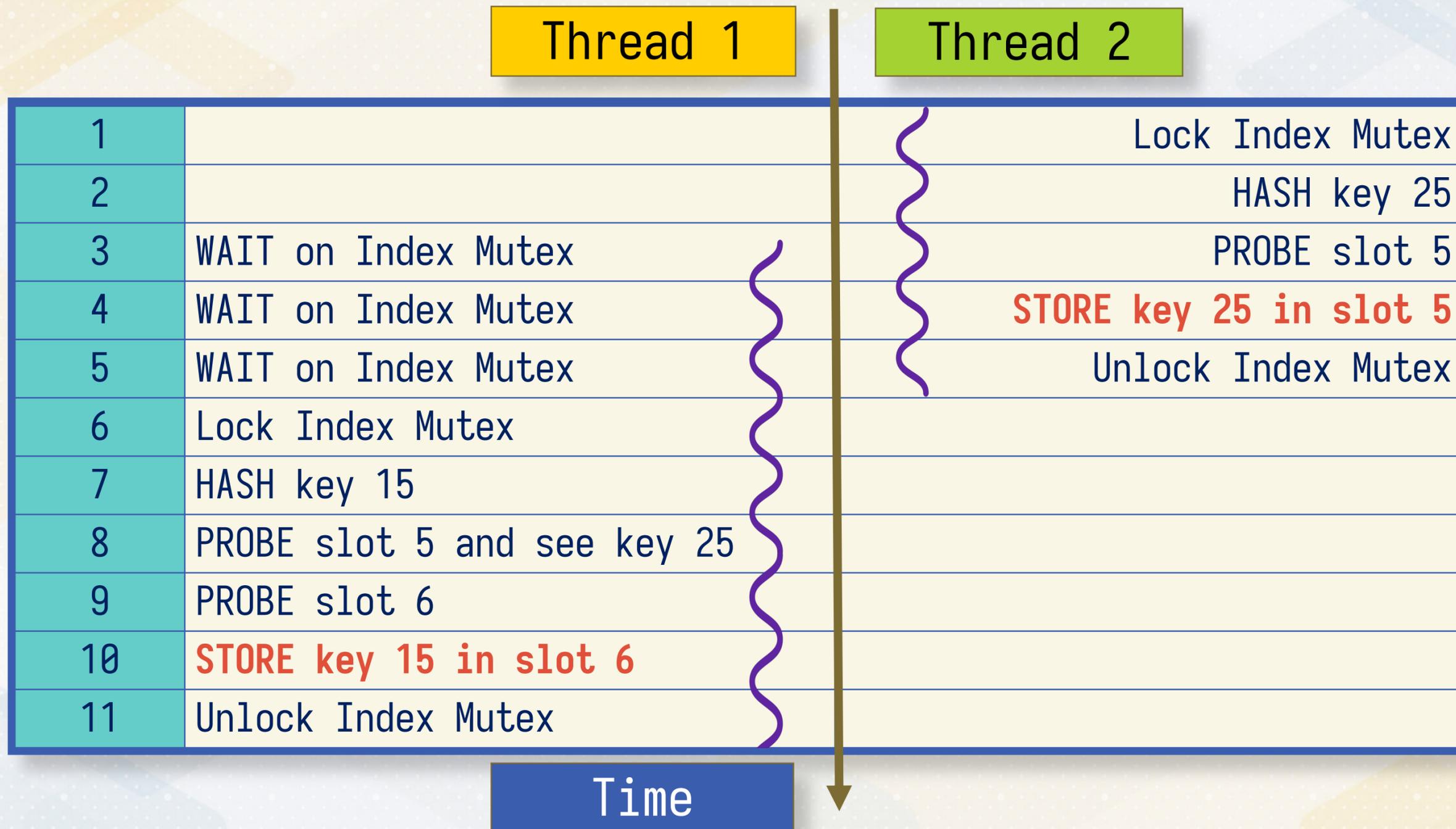
lock_guard automatically acquires a lock on creation and releases it on destruction.

```
class HashIndex {  
private:  
    mutable std::mutex indexMutex; // Mutex for thread-safe access  
  
    void insertOrUpdate(int key, int value) {  
        // RAII-style mutex management  
        std::lock_guard<std::mutex> guard(indexMutex);  
        // Perform thread-safe update on the index  
    }  
};
```

Thread Safety with Mutex



Thread Safety with Mutex



Order of Thread Execution



0	1	2	3	4	5	6	7	8	9
					15	25			
					Fig	Pear			



0	1	2	3	4	5	6	7	8	9
					25	15			
					Pear	Fig			

Range Query



Range Query vs Point Query

Range Query

Tuples where a column is in a Range of Values

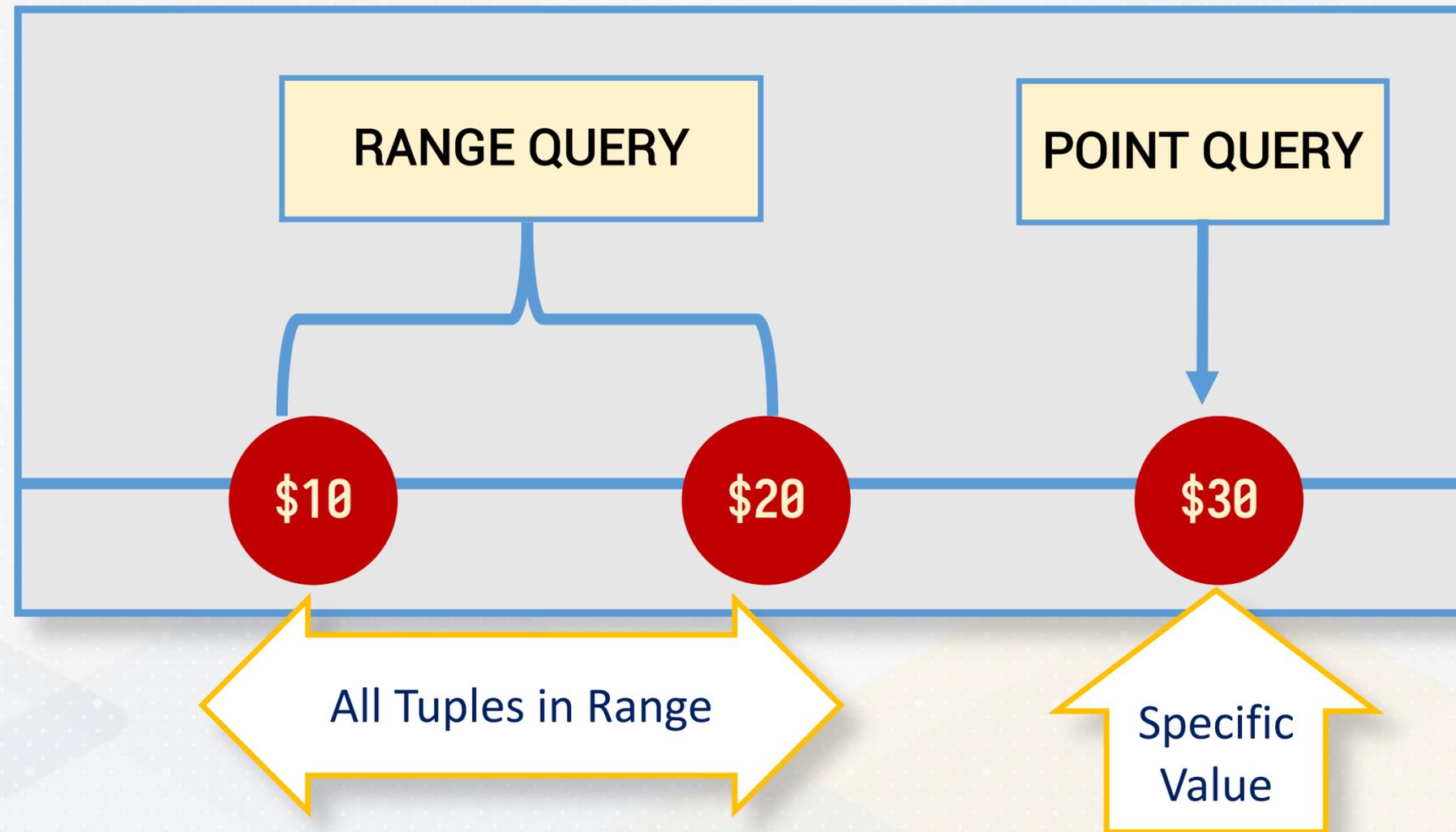
Products priced between \$10 and \$20

Point Query

Tuples where a column is equal to a Single Value

Products priced \$30

Range Query vs Point Query



Point Query Using Hash Table Index

Look up a key in the hash table with $O(1)$ average complexity.

```
std::vector<int> tuple_ids = productIndex.getValue(30);
```

POINT QUERY



0	1	2	3	4	5	6	7	8	9
\$30		\$12			\$25		\$17		
[103, 105]		[104]			[101]		[106, 102]		

Point Query Using Hash Table Index

Hash Table

Not ordered based on key

Iterate through entire table

RANGE QUERY

0	1	2	3	4	5	6	7	8	9
\$30		\$12			\$25		\$17		
[103, 105]		[104]			[101]		[106, 102]		

Point Query Using Hash Table Index

```
std::vector<int> rangeQuery(int lowerBound, int upperBound) const {  
    std::vector<int> results;  
    for (size_t i = 0; i < capacity; ++i) {  
        if (hashTable[i].exists && hashTable[i].key >= lowerBound &&  
            hashTable[i].key <= upperBound) {  
            results.push_back(hashTable[i].value);  
        }  
    }  
    return results;  
}
```

Inefficient Range Query in BuzzDB



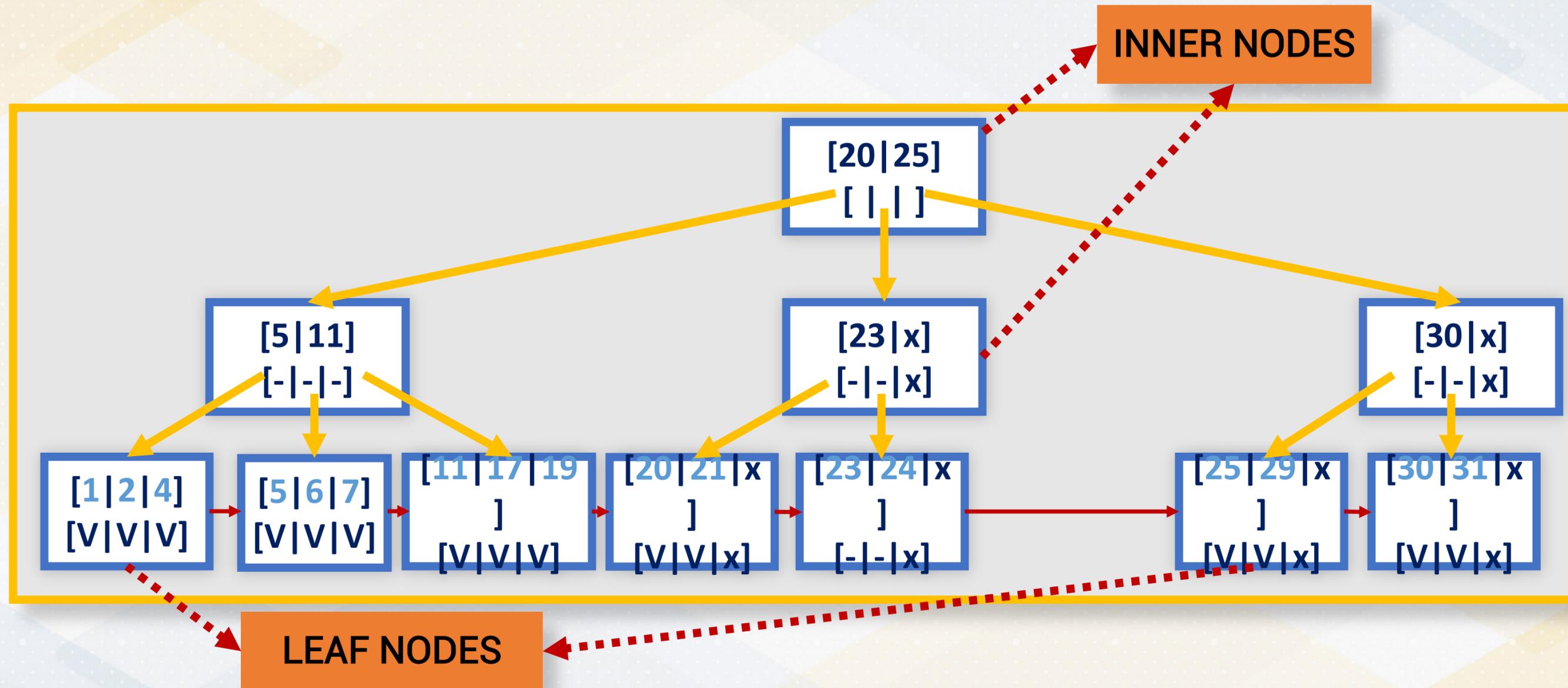
buzzDB

```
void performRangeQueryWithHashIndex(int lowerBound, int upperBound) {  
    auto results = index.rangeQuery(lowerBound, upperBound);  
    std::cout << "Found " << results.size() << " records in the range.\n";  
}
```

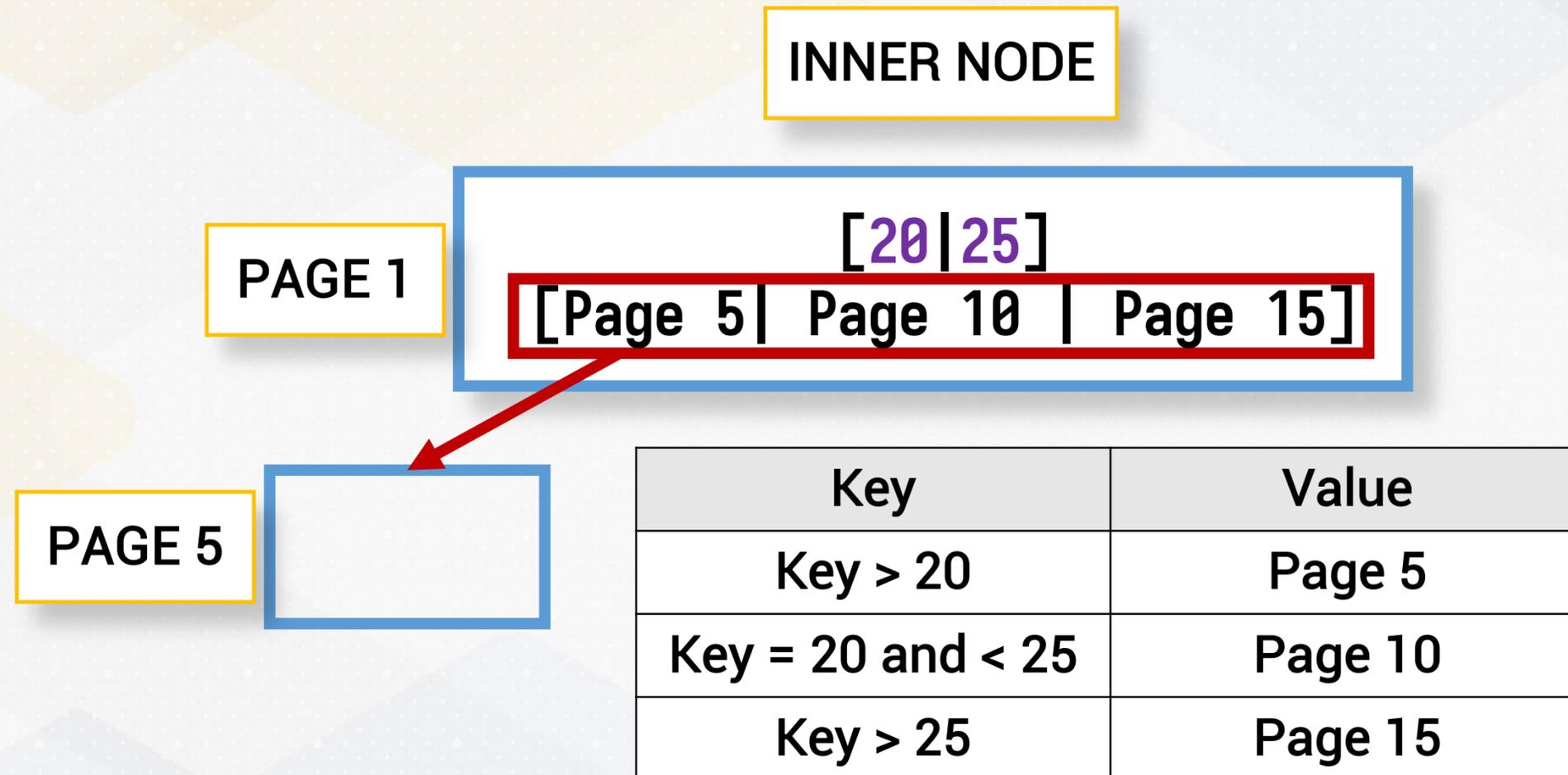
B+Tree



B+Tree Structure



Inner Node



Leaf Node

LEAF NODE

PAGE 8

[1 | 2 | 4]

[Tuple 310 | Tuple 102 | Tuple 115]

Key	Value
Key = 1	Tuple 310
Key = 2	Tuple 102
Key = 4	Tuple 115

Key and Value Types

ENTRIES

Entry	Key	Value
Entry 1	"max_file_size"	"10 MB"
Entry 2	"default_language"	"English"
Entry 3	"session_timeout"	"30 minutes"

LEAF NODE

PAGE 8

```
[ "max_file_size" | "default_language" | "session_timeout" ]  
  ["10 MB" | "English" | "30 minutes"]
```

Entry in C++

```
struct Entry {  
    Key key;  
    Value value; // Used only in leaf node entry  
    NodePtr next; // Used only in inner node entry  
  
    Entry(Key k, Value v) : key(k), value(v), next(nullptr) {}  
};
```

B+Tree Node in C++



isLeaf?

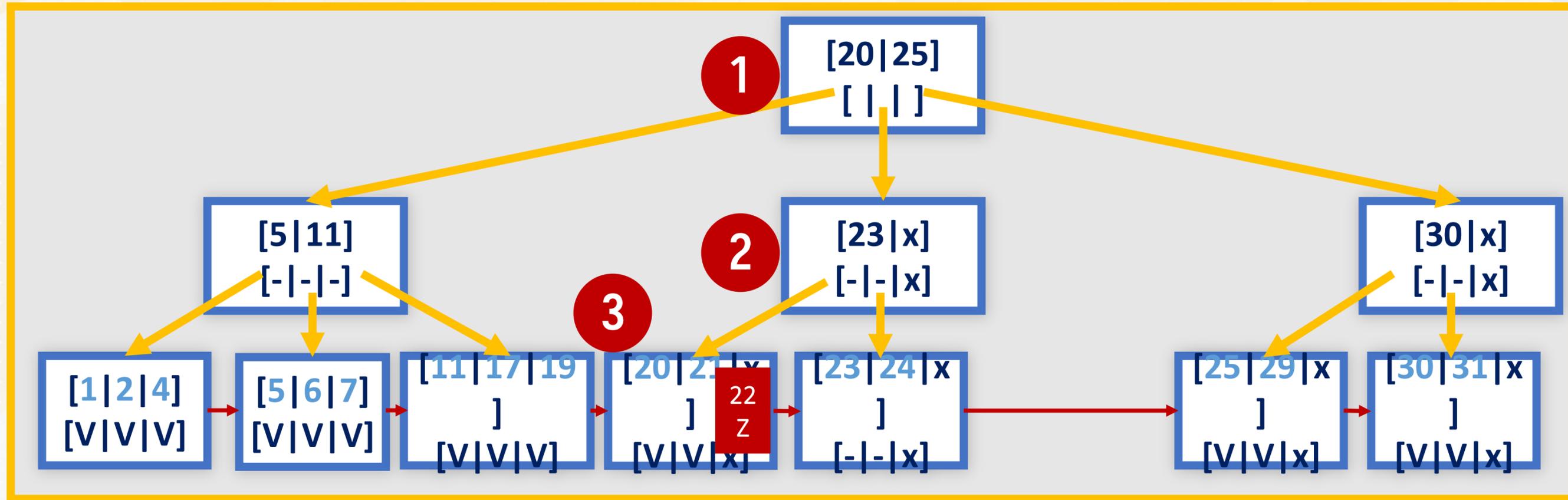
Vector of Entries

Entry = {Key-Value}

Next Pointer

```
struct Node {  
    bool isLeaf;  
    std::vector<Entry> entries;  
    std::unique_ptr<Node> next;  
  
    Node(bool leaf) : isLeaf(leaf), next(nullptr) {}  
};
```

Inserting a Key-Value Pair



Inserting a Key-Value Pair

Insertion process is recursive

```
void insertOrUpdate(const Key &key, const Value &value) {  
    insertOrUpdateInternal(key, value, root);  
}
```

Recursive Descent in an Inner Node

Find the appropriate child node by comparing keys.

```
void insertOrUpdateInternal(const Key &key, const Value &value, NodePtr &node) {  
    ...  
    else {  
        bool found = false;  
        for (auto it = node->entries.begin(); it != node->entries.end(); ++it) {  
            if (key < it->key) {  
                insertOrUpdateInternal(key, value, it->next);  
                found = true;  
                break;  
            }  
        }  
    }  
}
```

Insertion in a Leaf Node

If key does not exist in leaf node, insert it at the correct position to maintain sorted order.

```
void insertOrUpdateInternal(const Key &key, const Value &value, NodePtr &node) {
    if (node->isLeaf) {
        // Search for the key in the leaf node
        auto it = std::find_if(node->entries.begin(), node->entries.end(),
                               [&](const Entry &entry) { return entry.key == key; });
        // Key not found, proceed to insert in sorted order
        auto comp = [](const Entry &entry, const Key &k) { return entry.key < k; };
        it = std::lower_bound(node->entries.begin(), node->entries.end(), key,
                              comp);
        node->entries.insert(it, Entry(key, value));
    }
    ...
}
```

Binary Search using a Lambda Function

 **Function**

Anonymous

Short Functions

Comparator

```
// Key not found, proceed to insert in sorted order
auto comp = [](const Entry &entry, const Key &k)
             { return entry.key < k; };
it = std::lower_bound(node->entries.begin(),
                     node->entries.end(),
                     key, comp);
node->entries.insert(it, Entry(key, value));
```

Binary Search using a Lambda Function

 Function

lower_bound

iterator

Sorted Order

```
// Key not found, proceed to insert in sorted order
auto comp = [](const Entry &entry, const Key &k)
             { return entry.key < k; };
it = std::lower_bound(node->entries.begin(),
                     node->entries.end(),
                     key, comp);
node->entries.insert(it, Entry(key, value));
```

Node Split



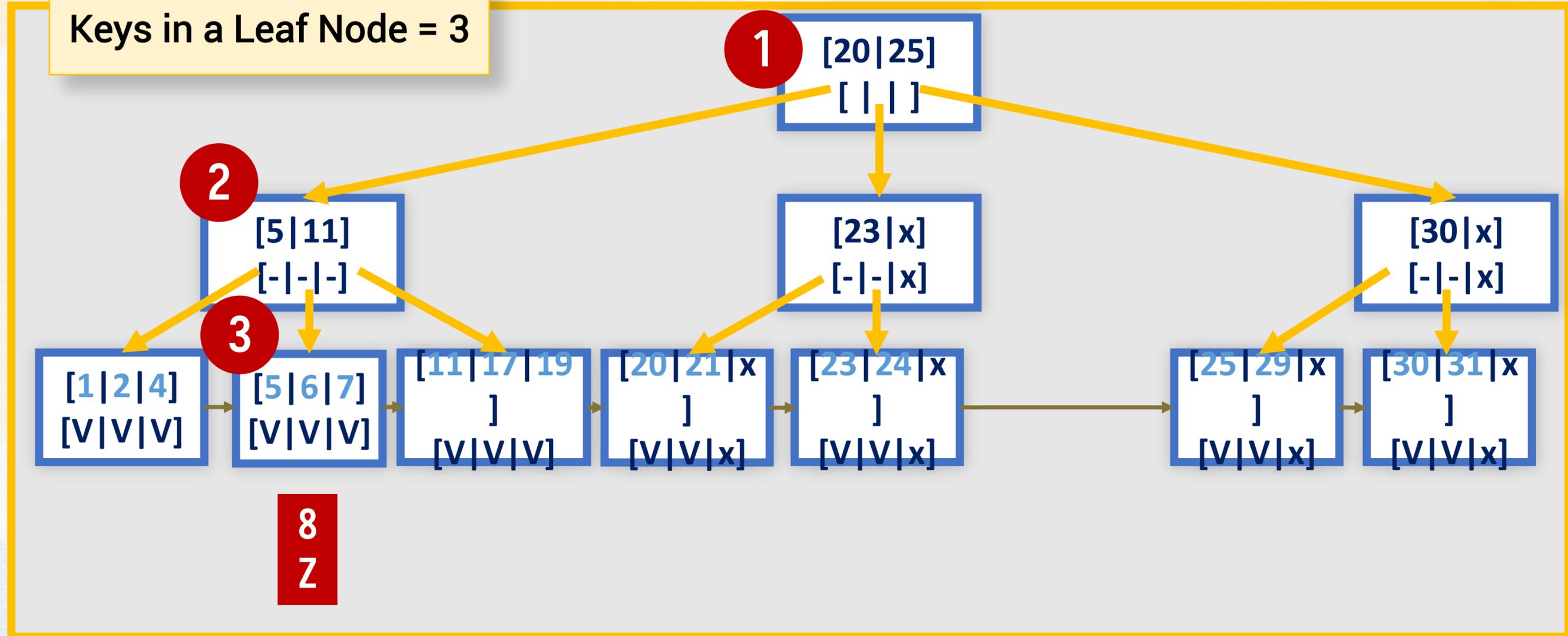
Separate Vectors for Keys and Values

Node contains keys and values in separate vectors.

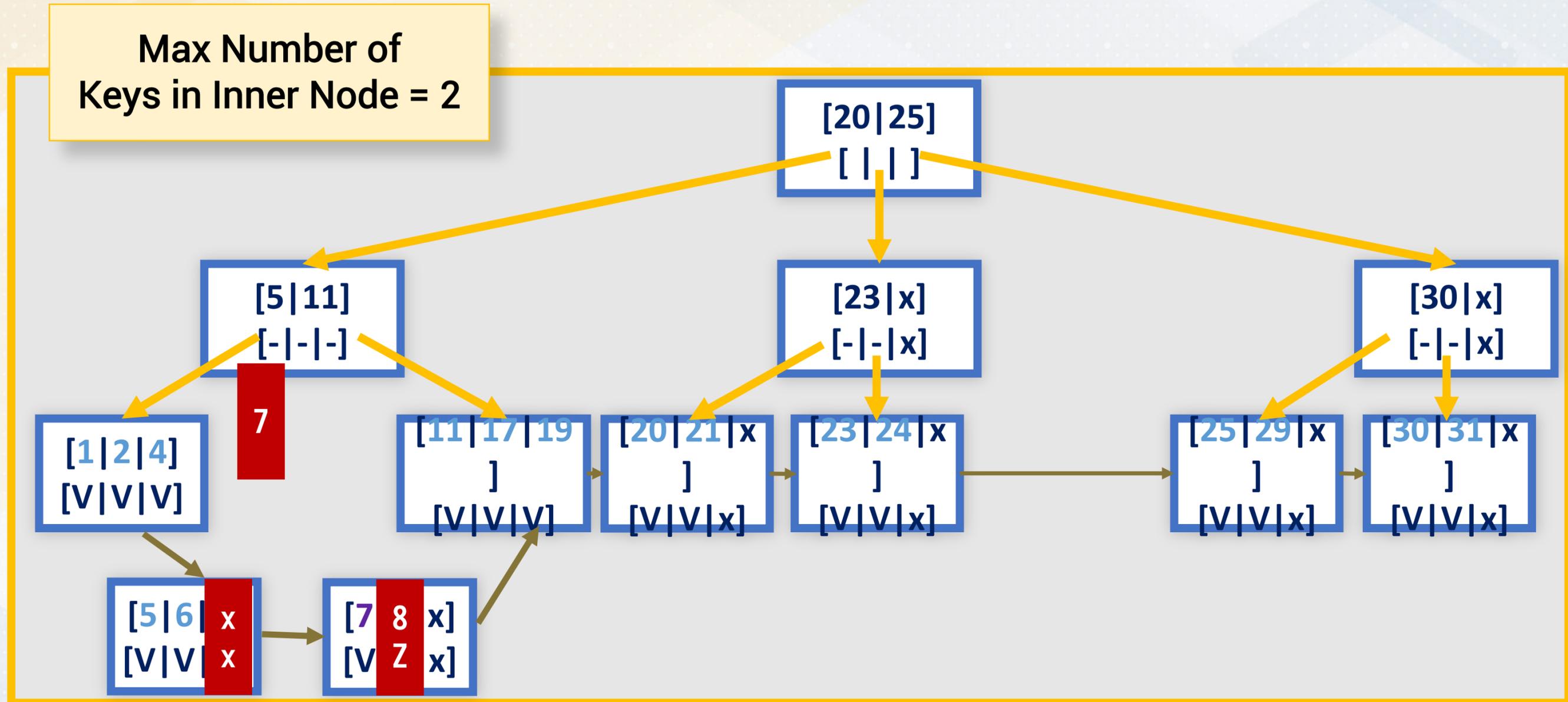
```
template <typename Key, typename Value> class BPlusTree {
    struct Node {
        std::vector<Key> keys;
        std::vector<Value> values; // Only used in leaf nodes
        std::vector<std::shared_ptr<Node>> children; // Only used in internal nodes
        std::shared_ptr<Node> next = nullptr; // Next leaf node
        bool isLeaf = false;
        Node(bool leaf) : isLeaf(leaf) {}
    };
    size_t maxKeys; // Order of the tree
    std::shared_ptr<Node> root;
};
```

Node Split: Inserting [8, Z]

Max Number of
Keys in a Leaf Node = 3

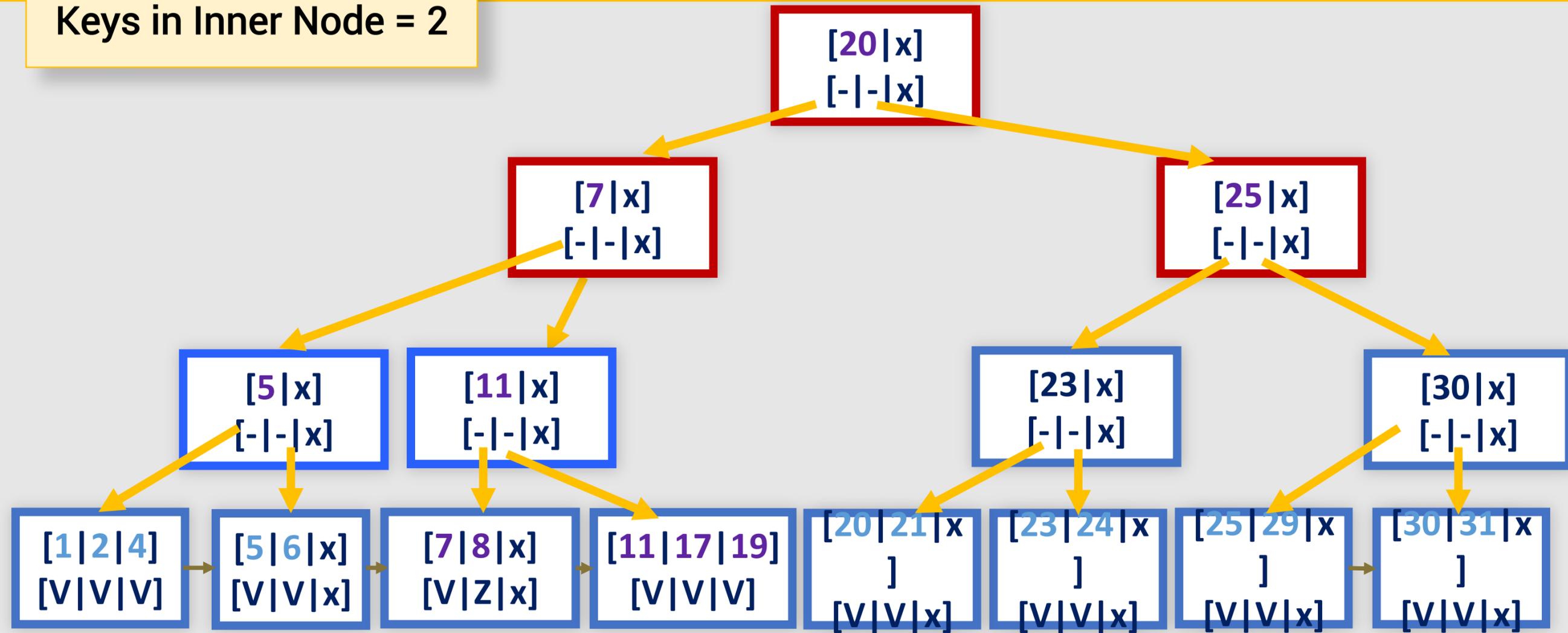


Node Split: Inserting [8, Z]



Node Split: Inserting [8, Z]

Max Number of
Keys in Inner Node = 2



Node Split in C++

buzzDB

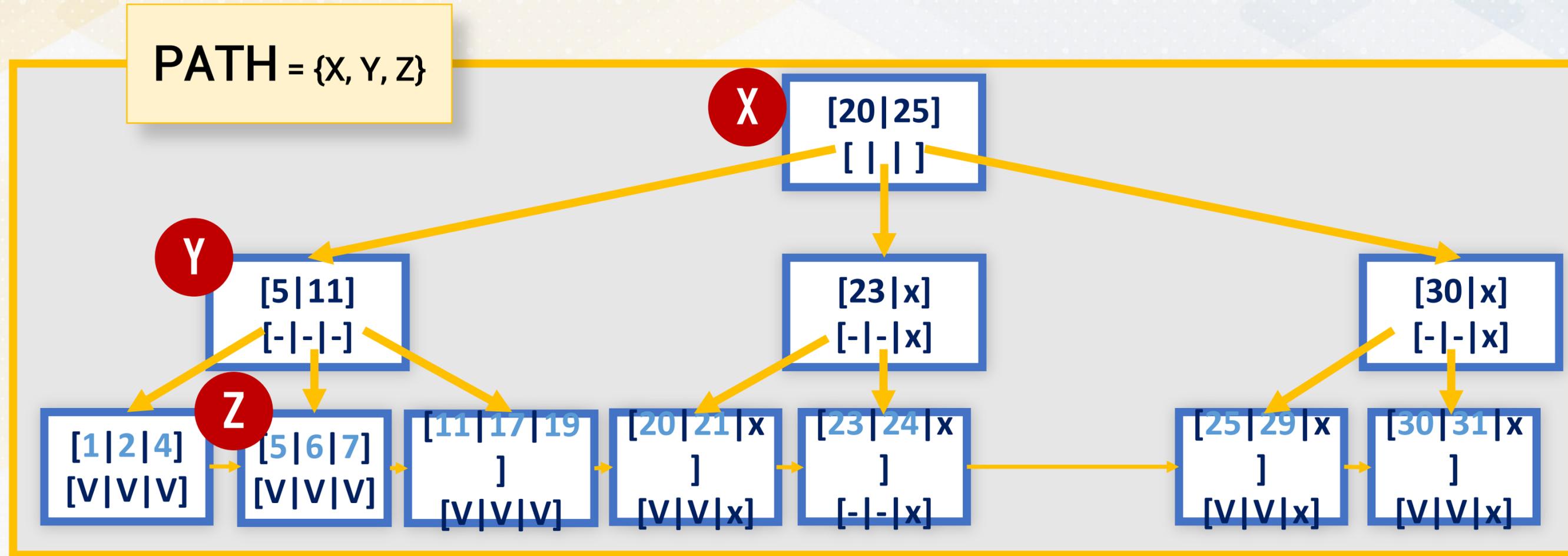
Node Exceeds Capacity → Split Node

```
if (node->keys.size() > maxKeys) {  
    splitNode(path, node);  
}
```

Insertion

```
auto node = root;
std::vector<std::shared_ptr<Node>> path; // Track the path for backtracking
while (!node->isLeaf) {
    path.push_back(node);
    auto it = std::upper_bound(node->keys.begin(), node->keys.end(), key);
    size_t index = it - node->keys.begin();
    node = node->children[index];
}
```

Path Tracking and Backtracking



Insertion in Leaf Node

```
auto it = std::lower_bound(node->keys.begin(), node->keys.end(), key);
if (it != node->keys.end() && *it == key) {
    size_t pos = std::distance(node->keys.begin(), it);
    node->values[pos] += value; // Update existing key's value
}
else {
    size_t pos = it - node->keys.begin();
    node->keys.insert(node->keys.begin() + pos, key);
    node->values.insert(node->values.begin() + pos, value);
}
```

Node Splitting in Leaf Node

```
if (node->isLeaf) {  
    auto newNode = std::make_shared<Node>(true); // Create a new leaf node  
    size_t mid = node->keys.size() / 2; // Calculate the middle index  
    // Move the second half of keys and values to the new node  
    std::move(node->keys.begin() + mid, node->keys.end(),  
              std::back_inserter(newNode->keys));  
    std::move(node->values.begin() + mid, node->values.end(),  
              std::back_inserter(newNode->values));  
    newNode->next = node->next;  
    node->next = newNode;  
}
```

Node Splitting in Inner Node

```
else { // Handling internal nodes
    auto newNode = std::make_shared<Node>(false); // Create a new internal node
    size_t mid = node->keys.size() / 2; // Middle index
    Key midKey = node->keys[mid]; // Key that will move to the parent
    // Move keys and children to the new node
    std::move(node->keys.begin() + mid + 1, node->keys.end(),
              std::back_inserter(newNode->keys));
    std::move(node->children.begin() + mid + 1, node->children.end(),
              std::back_inserter(newNode->children));
}
```

Backtracking to Parent Node

Path Vector

Determines
Parent Node

Median Key
Insertion

Parent Node May
Exceed Capacity

Recursion

Can Travel Up to
Root Node

```
else {  
    auto parent = path.back(); // Get the parent node  
    path.pop_back();          // Remove the last tracked node  
    size_t pos = std::distance(  
        parent->keys.begin(),  
        std::lower_bound(parent->keys.begin(), parent->keys.end(), midKey));  
    parent->keys.insert(parent->keys.begin() + pos, midKey); // Insert median key  
    parent->children.insert(parent->children.begin() + pos + 1, newNode); // Insert  
}
```

Backtracking to Root Node

Backtracking

Empty Path Vector

New Root Node

Two Children

Tree Height += 1

```
if (path.empty()) {  
    // New root for the tree  
    auto newRoot = std::make_shared<Node>(false);  
    newRoot->keys.push_back(midKey);           // Add median key  
    newRoot->children.push_back(node);         // Left child  
    newRoot->children.push_back(newNode);     // Right child  
    // Update root pointer  
    root = newRoot;  
}
```

Trie

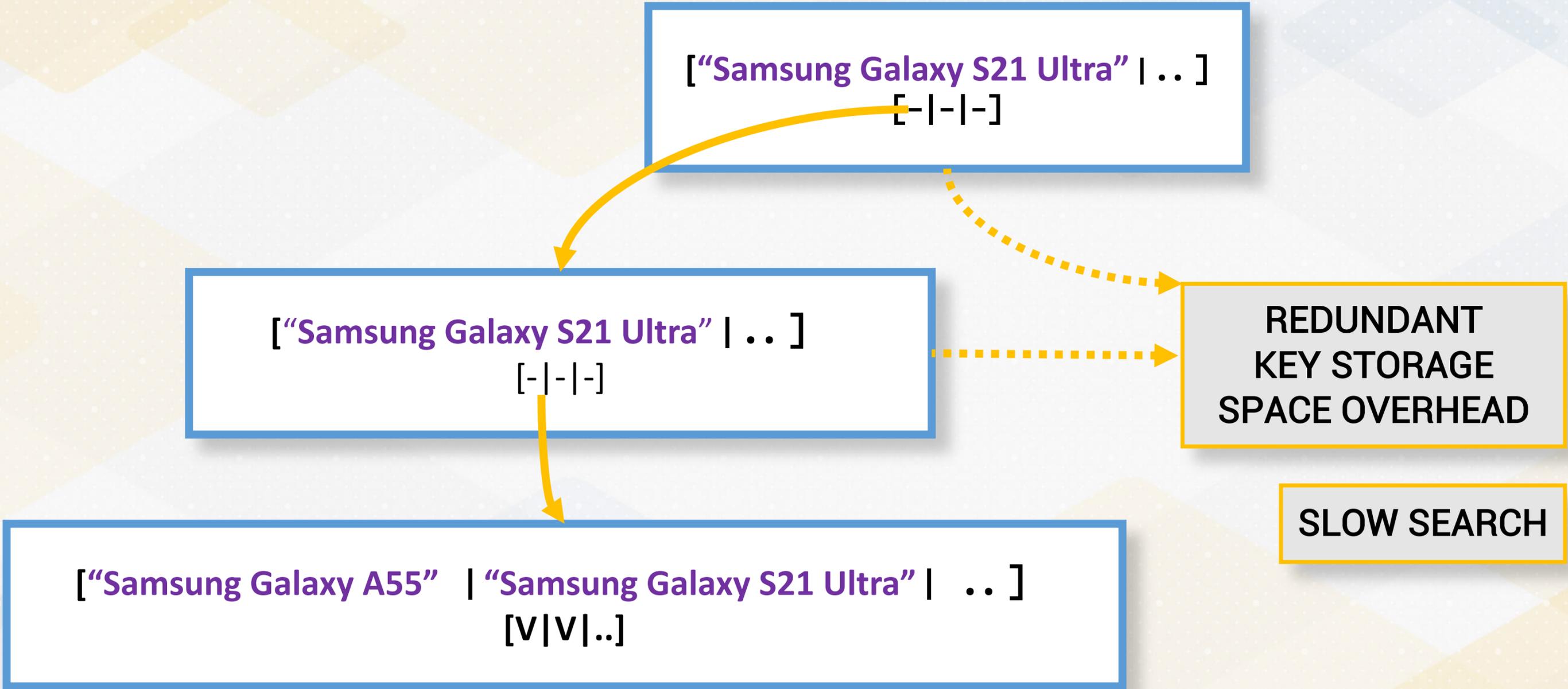


Product Catalog Search

Product Name	Relevant or not
Samsung Galaxy S21 Ultra	Yes
Apple iPhone XS Max	No
Samsung Galaxy A55	Yes
Apple iPhone 12	No
Samsung Galaxy S27	Yes

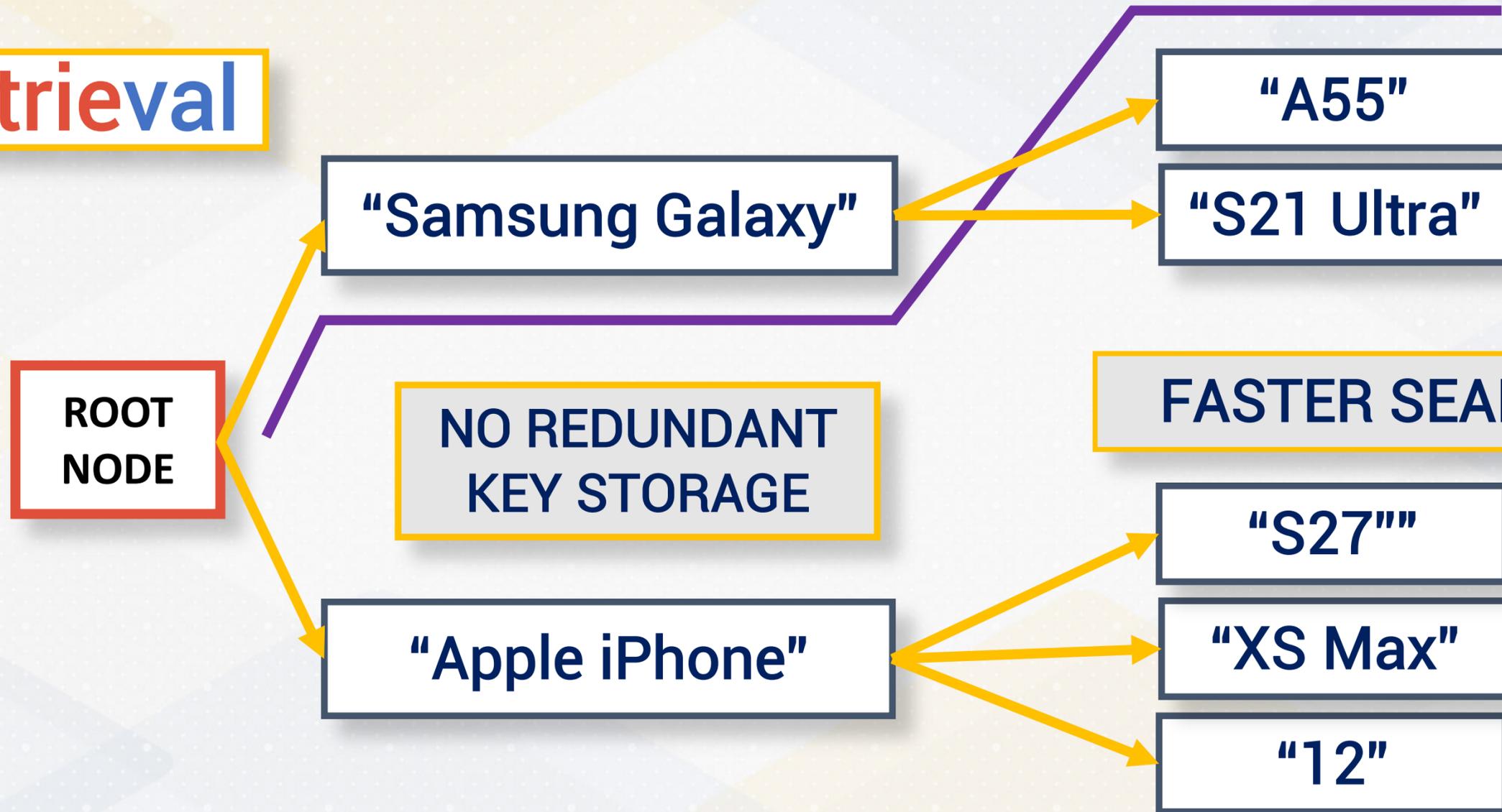
```
// Find all Samsung Galaxy phones in the catalog
std::vector<std::string> galaxyPhones =
    productCatalog.startsWith("Samsung Galaxy");
```

Limitations of B+Tree



Trie

Retrieval



Patricia Trie

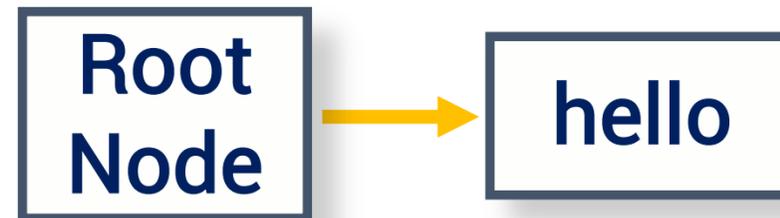


Basic Trie

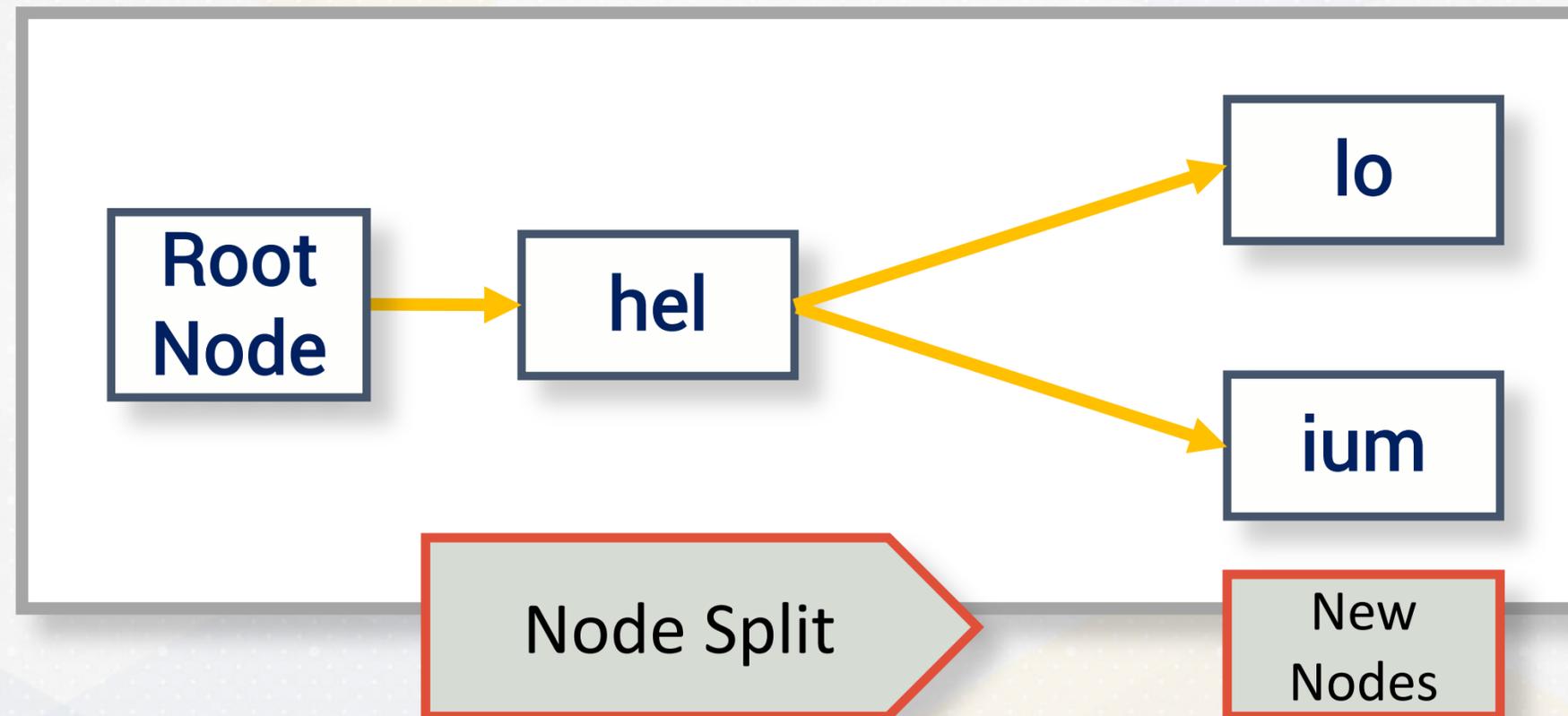
Patricia Trie



Insertion Example: "hello"



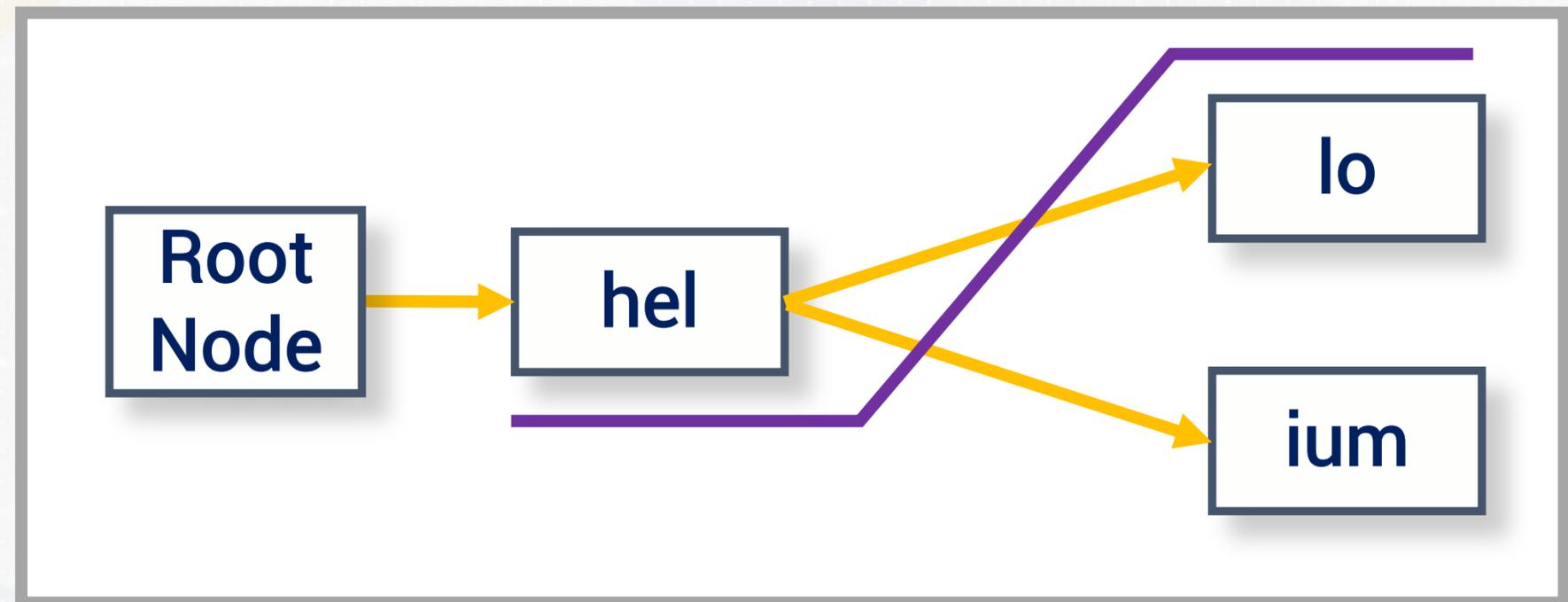
Insertion Example: "helium"



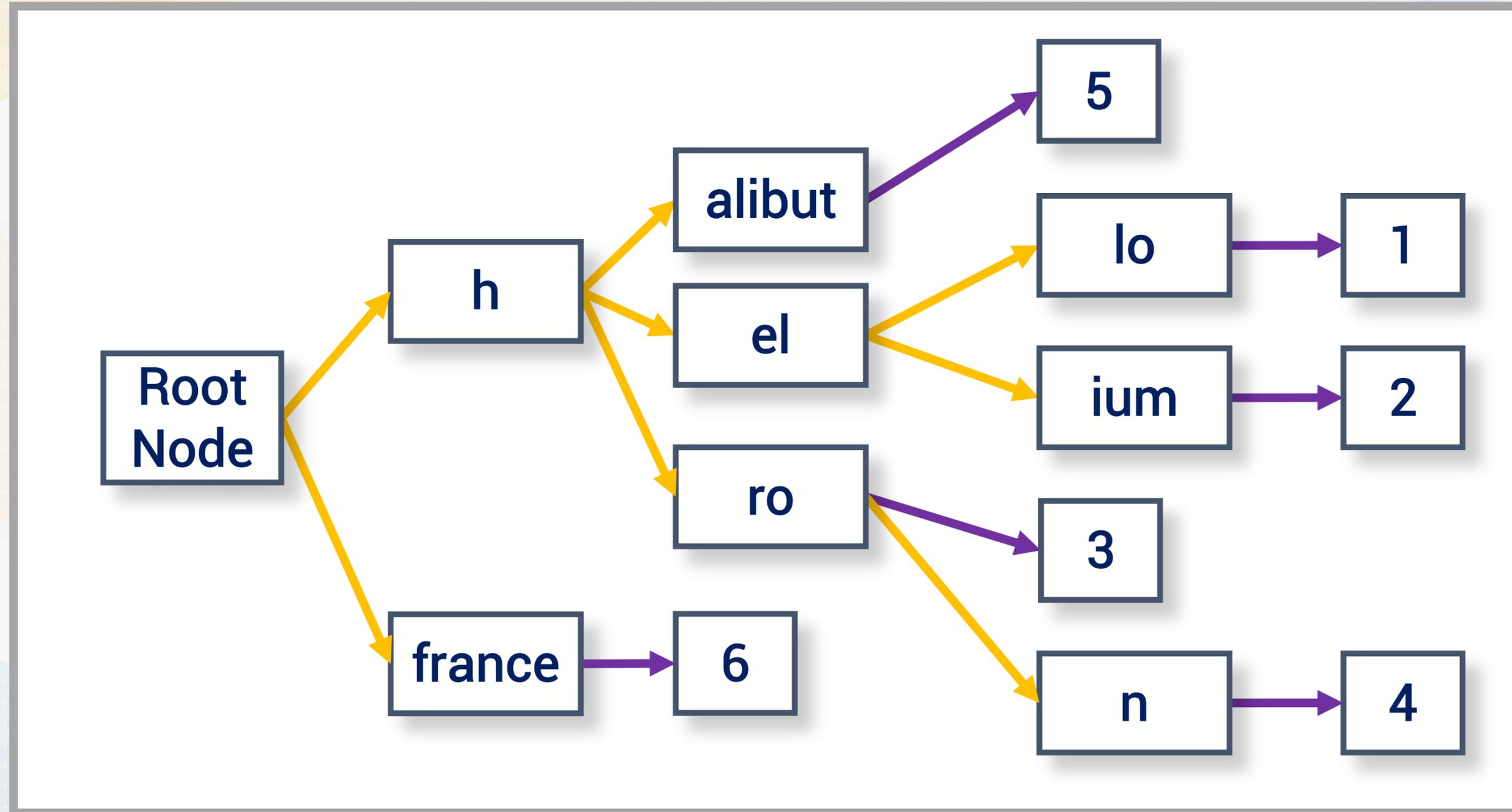
Retrieval: “hello” and “helicopter”

Navigate Root Node to Relevant Leaf Node

Each Character Narrows Down Search



More Complex Trie with Key-Value Pairs



Inverted Index



Inverted Index

Map words to the positions in which they appear in a document.

Word	Positions
russia	10, 20
napolean	25

Proximity Search

distance("russia", "napoleon") less than ten words

Word	Positions
russia	10, 20
napoleon	25

Library Search

Map words to the documents in which they appear.

Word	Document ID, Position
russia	(1, 10), (1, 20)
napoleon	(1, 25), (2, 30), (3, 50)

Why Inverted Index?

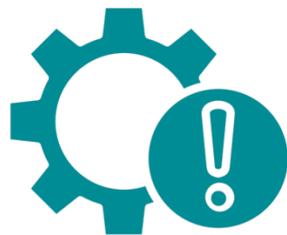
Inverted Index



Text
Data

Proximity
Search

Internals



Hash Table
Foundation

Values are
Doc IDs,
Positions

Structure of an Inverted Index

Key

"russia"

Value

{1: {5, 20}, 2: {10}}

```
std::unordered_map<  
    std::string,  
    std::unordered_map<int, std::vector<int>>  
>
```

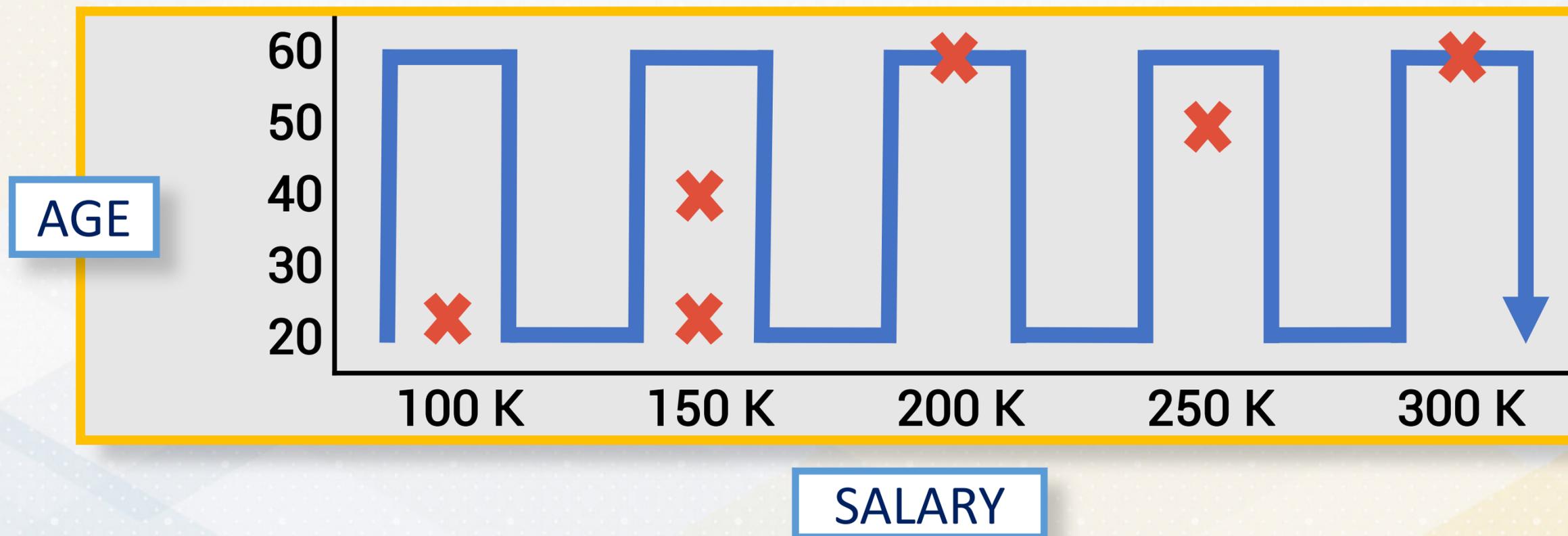
R Tree



Limitations of B+Tree

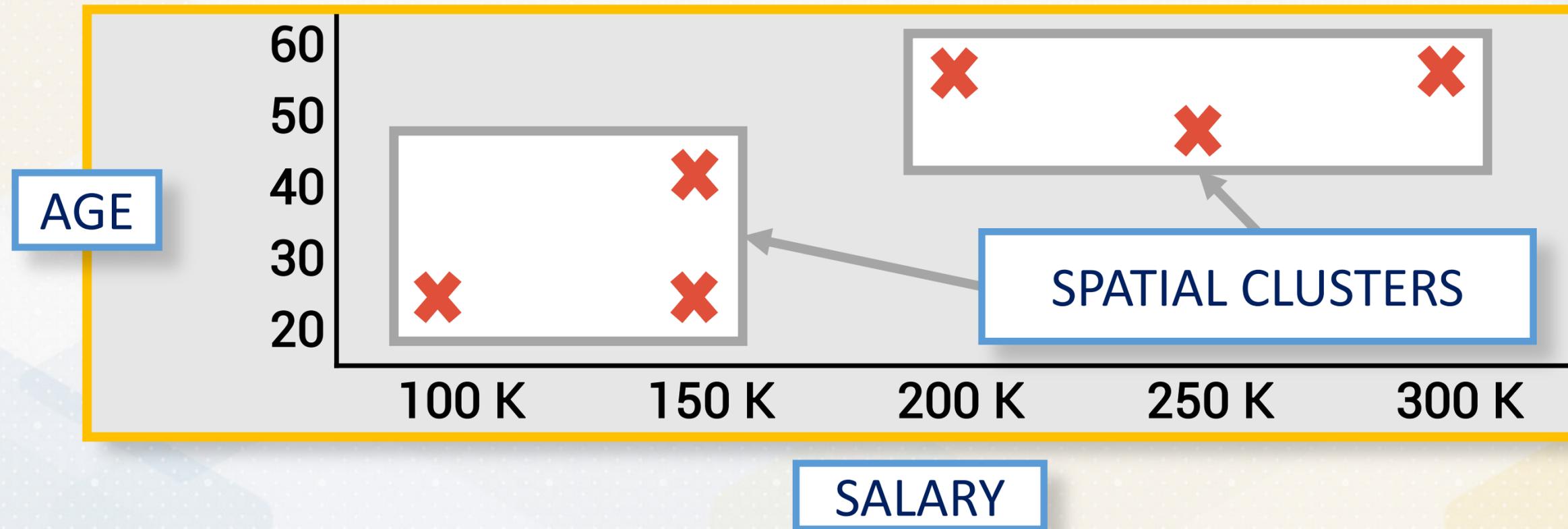


Designed for Single-Dimensional Indexing



R-Tree: A Multidimensional Index

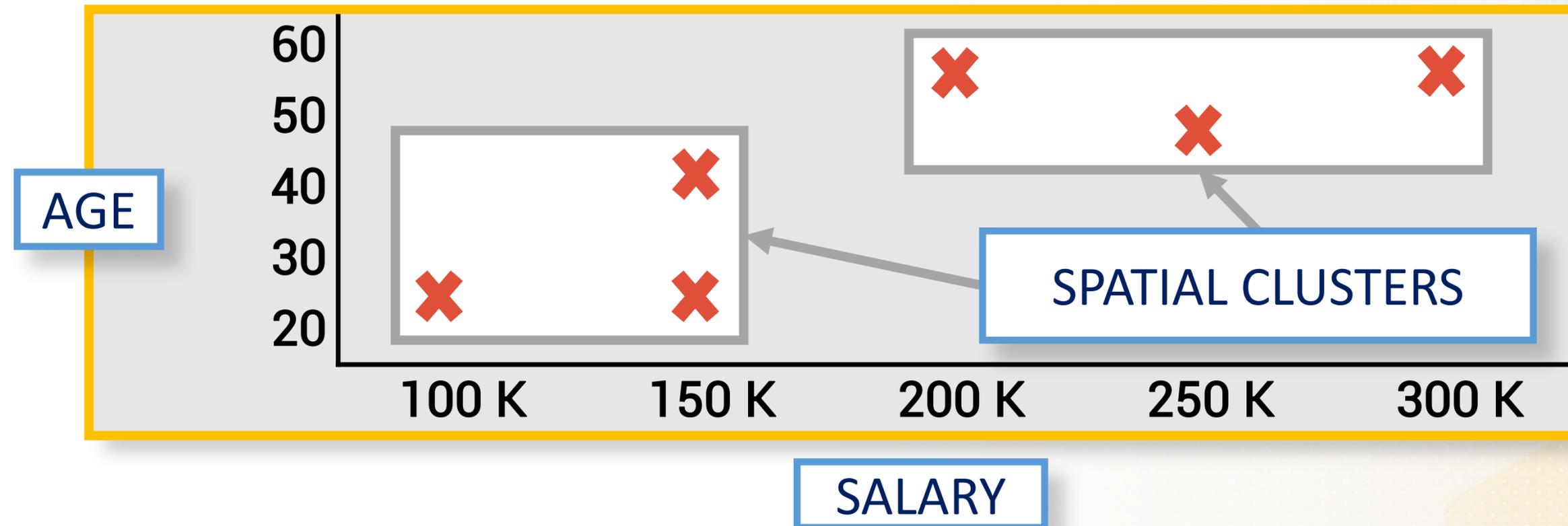
R-Tree groups multi-dimensional keys based on spatial proximity



R-Tree: A Multidimensional Index

(Salary, Age): Point

Clusters: Bounding Rectangles

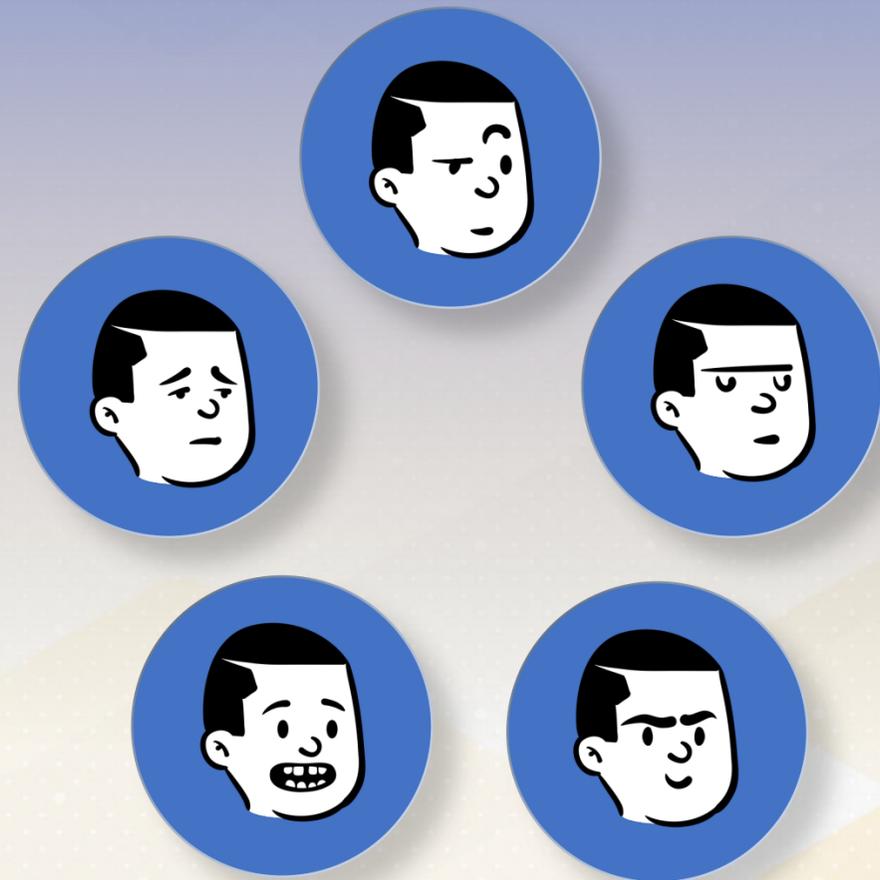


Supported Queries

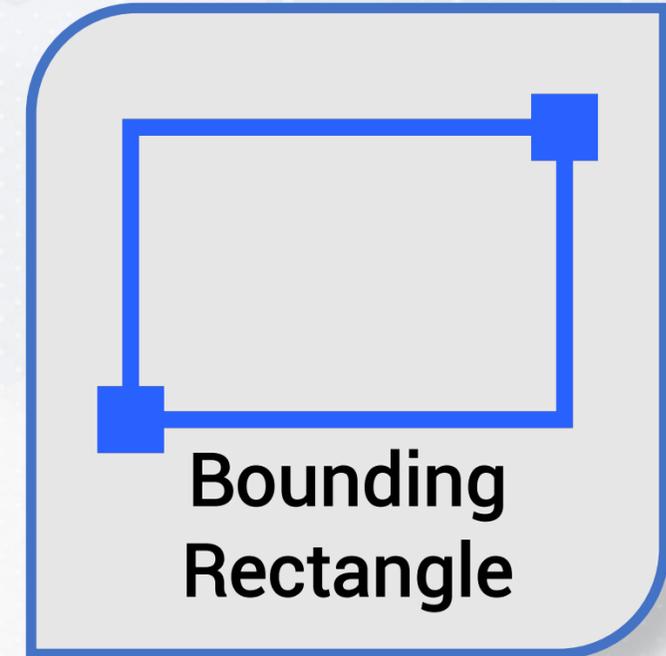
Spatial Queries



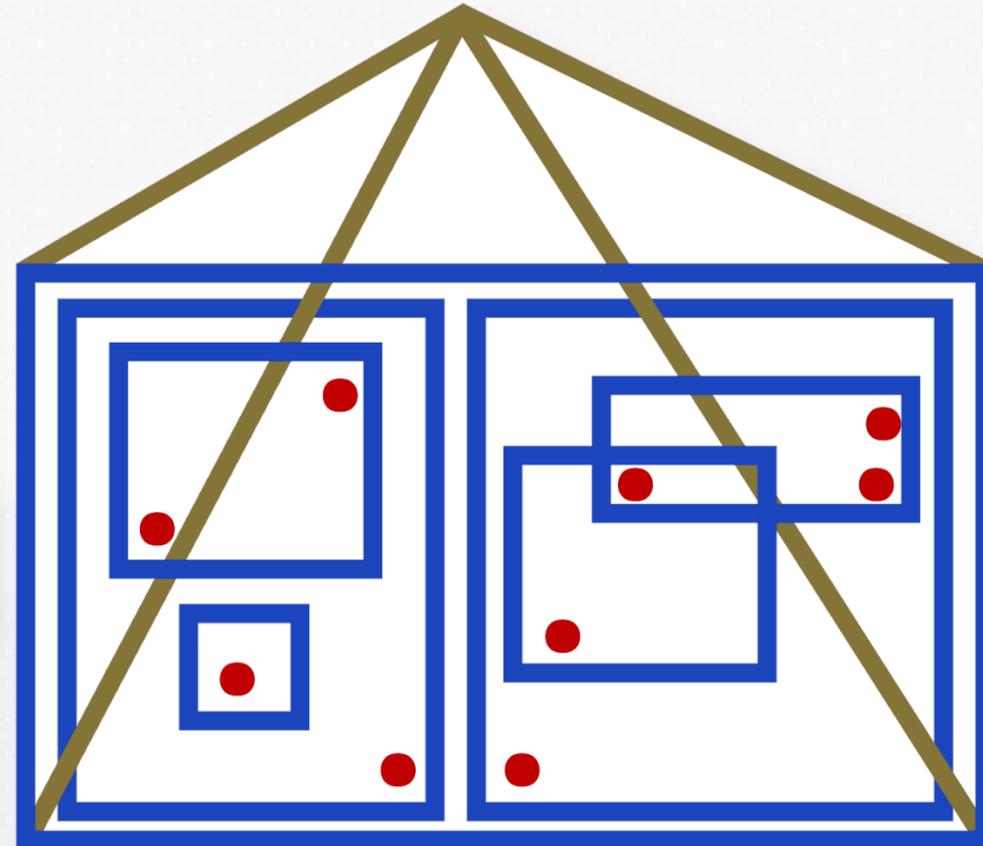
Nearest Neighbor Queries



Key Characteristics



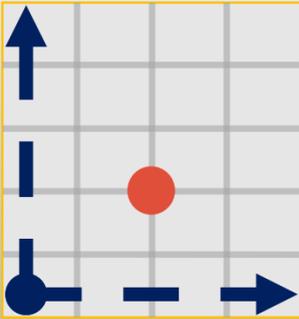
AGE

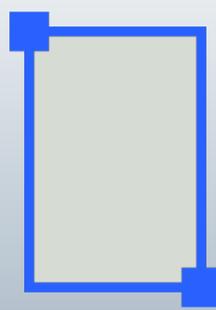


ROOT

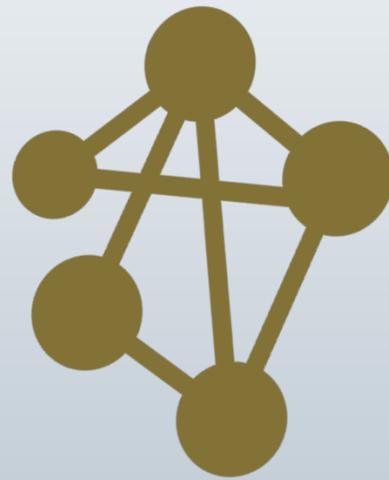
SALARY

R-Tree Data Types

	Point	Specific location in space
---	--------------	----------------------------

	Rectangle	loses points or other rectangles
---	------------------	----------------------------------

R-Tree Nodes



**R-Tree
Node**

Leaf Node

Contains
Points; Lowest
Level of Tree

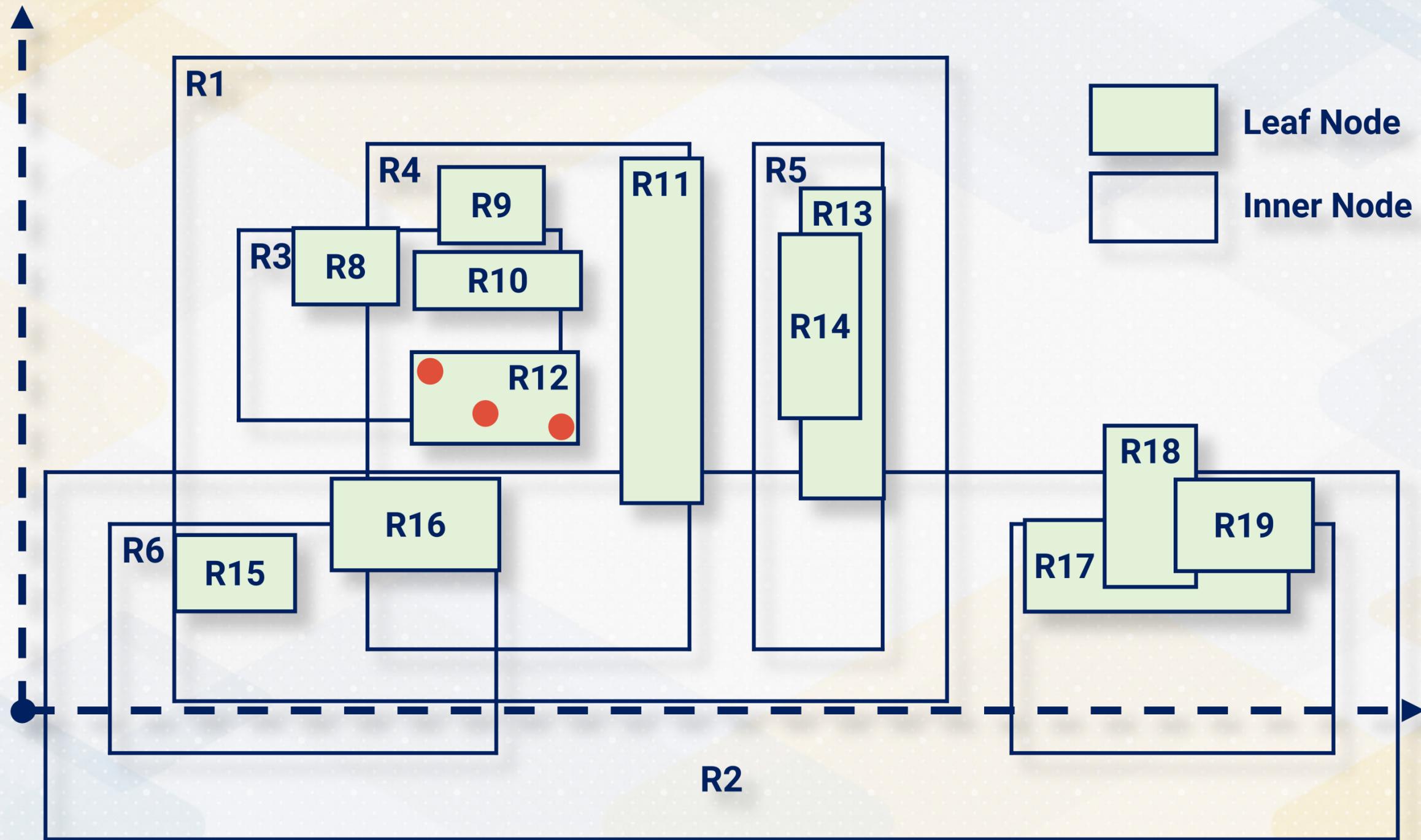
Inner Node

Contains Child
Nodes and
their Bounding
Rectangles

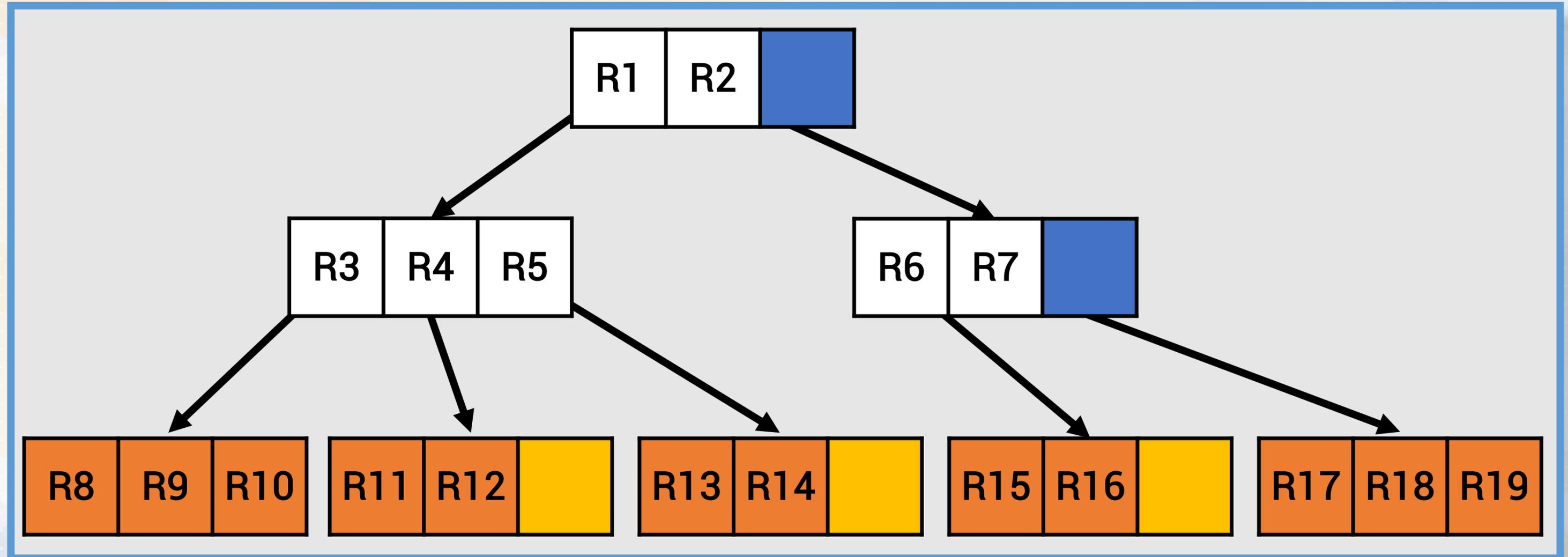
Root Node

Top-Most Inner
Node;
Entry
Point

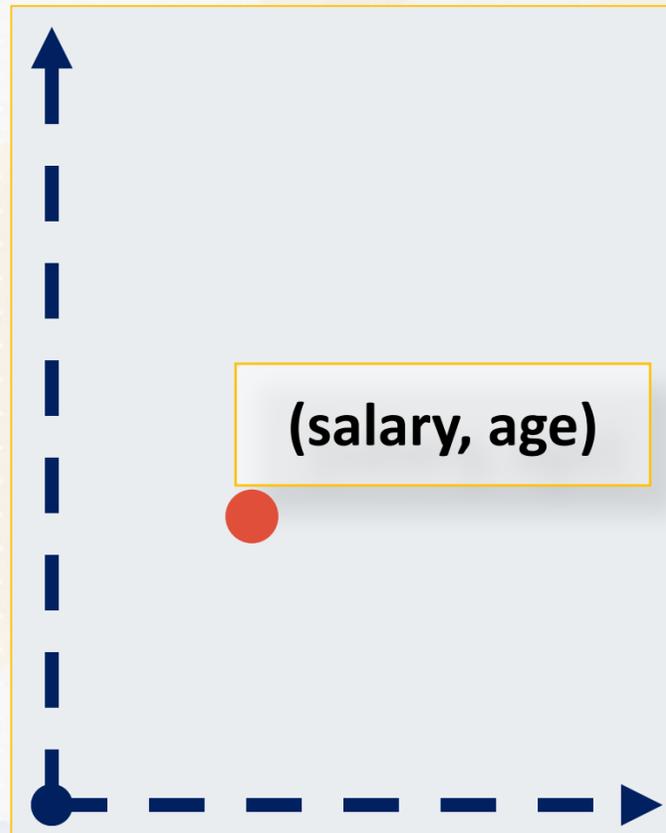
R-Tree Structure



R-Tree Structure



Point

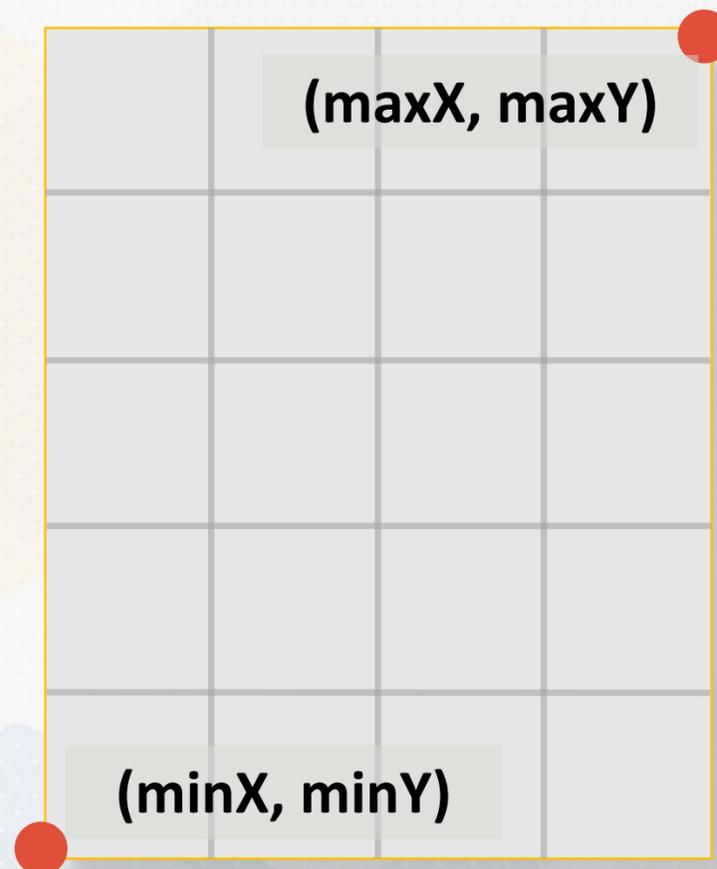


$(x, y) = (\text{salary}, \text{age})$

```
struct Point {  
    float x, y;  
    Point(float x, float y) : x(x), y(y) {}  
};
```

Rectangle

Defines a bounding box using its minimum and maximum coordinates



```
struct Rectangle {  
    float minX, minY, maxX, maxY;  
    Rectangle(float minX, float minY, float maxX, float maxY)  
        : minX(minX), minY(minY), maxX(maxX), maxY(maxY) {}  
  
    bool contains(const Point &p);  
    bool intersects(const Rectangle &other) const;  
};
```

Conclusion

- Indexing
- Hash Table
- Double Hashing
- Range Query
- B+Tree
- Trie
- RTree