

Lecture 14:

B+Tree



Logistics

- Programming assignment 3 (B+Tree) released
- Exam 1 scores released
- Extra-credit project (**+5%**)

Extra-Credit Project

- LLM-based question answering over textbook using local models on your laptop
- **Find ways to better and faster answer questions from students**
- <https://github.com/georgia-tech-db/TokenSmith>
- “What is atomicity?”, “How is OLAP different from OLTP?”
- Around 20 students from both sections will earn **+5%**
- Subjective evaluation only possible

Extra-Credit Project

- Publish code on public GitHub
- Iterate over the code and add features incrementally
- Focus on evaluating latency and accuracy metrics
- One-page project update report due on **Oct 30**
- Two-page project final report due on **Nov 18**
- In-class presentations from selected students on **Nov 20**

Recap

- Range Query
- Ordered Index
- B+Tree Template

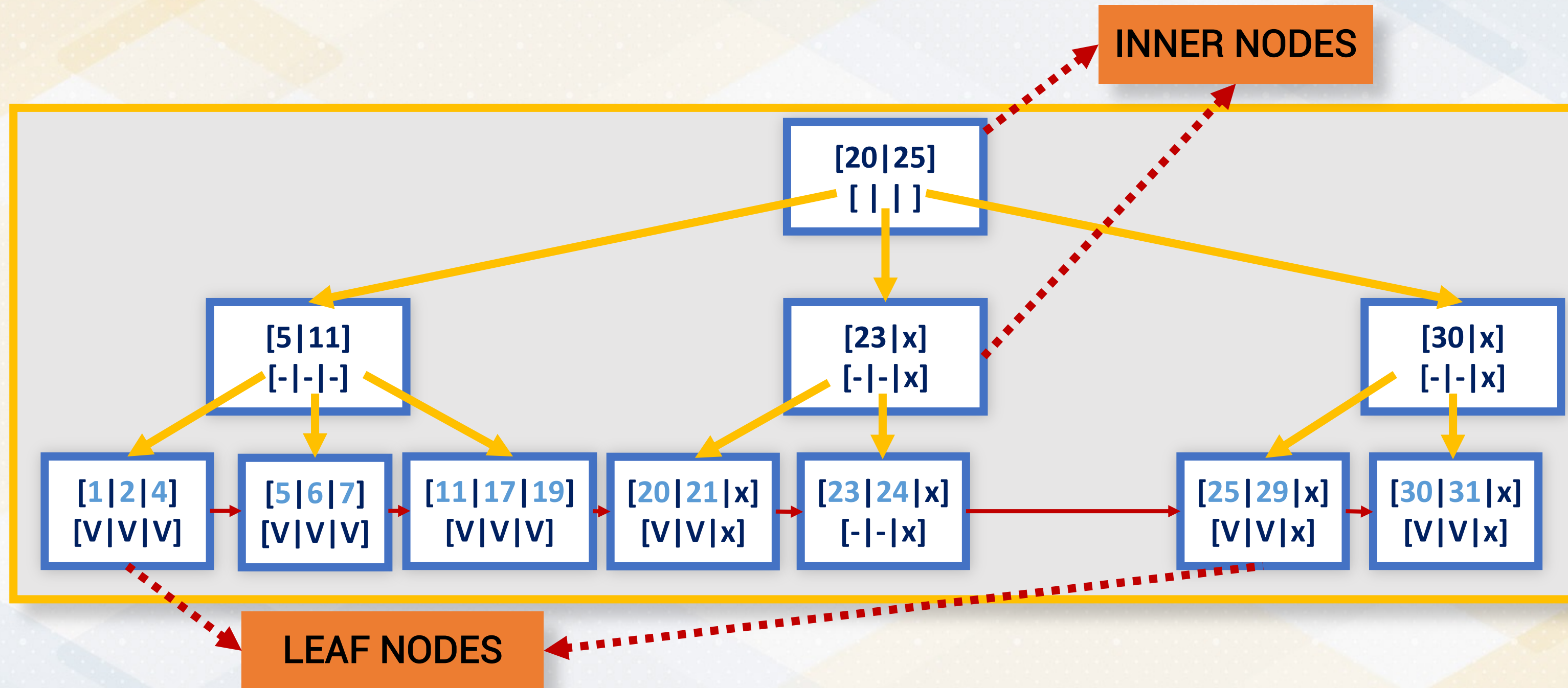
Lecture Overview

- B+Tree Structure
- Range Query Processing
- Node Split
- On-disk B+Tree

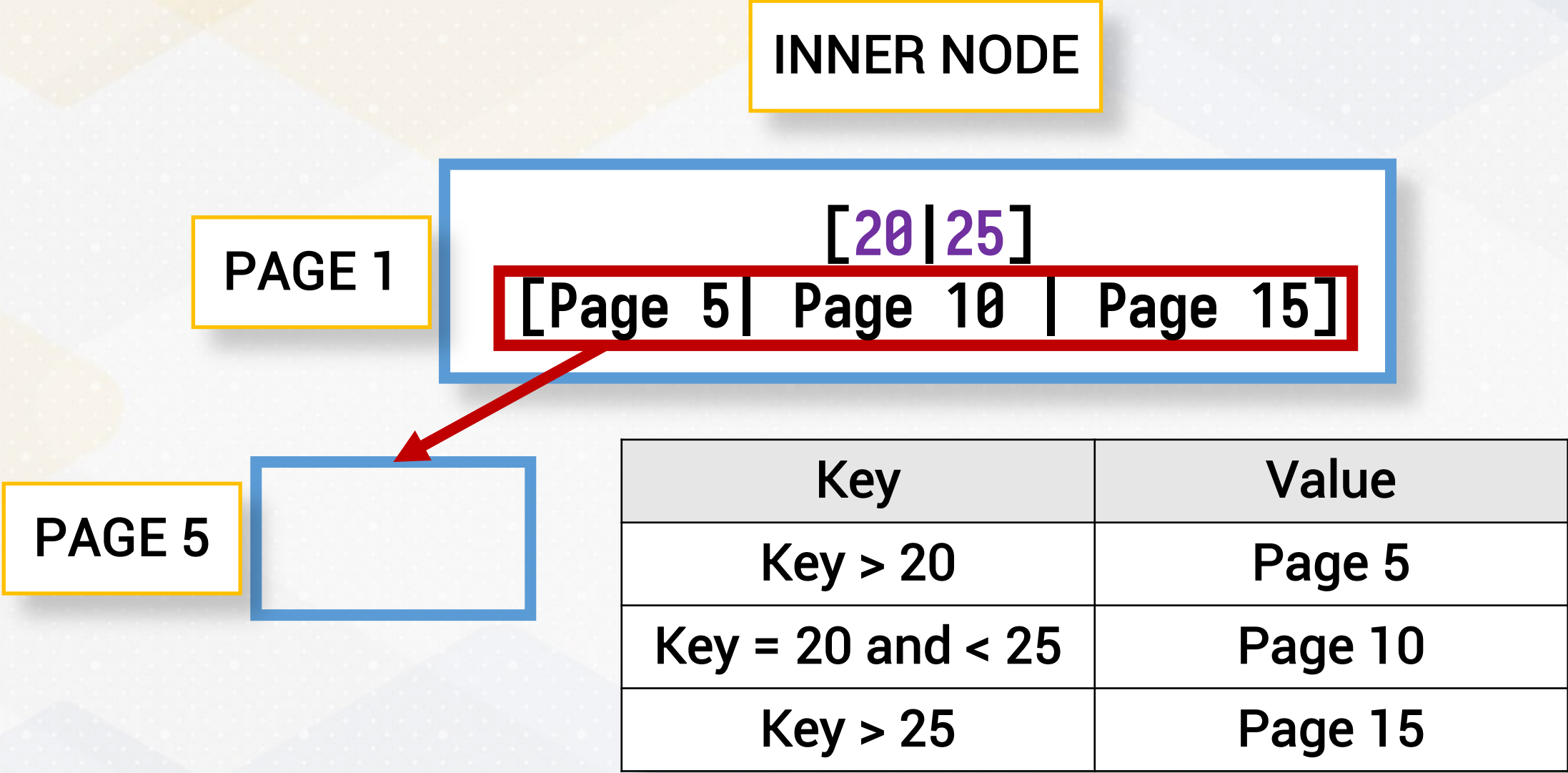
B+Tree Structure



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?

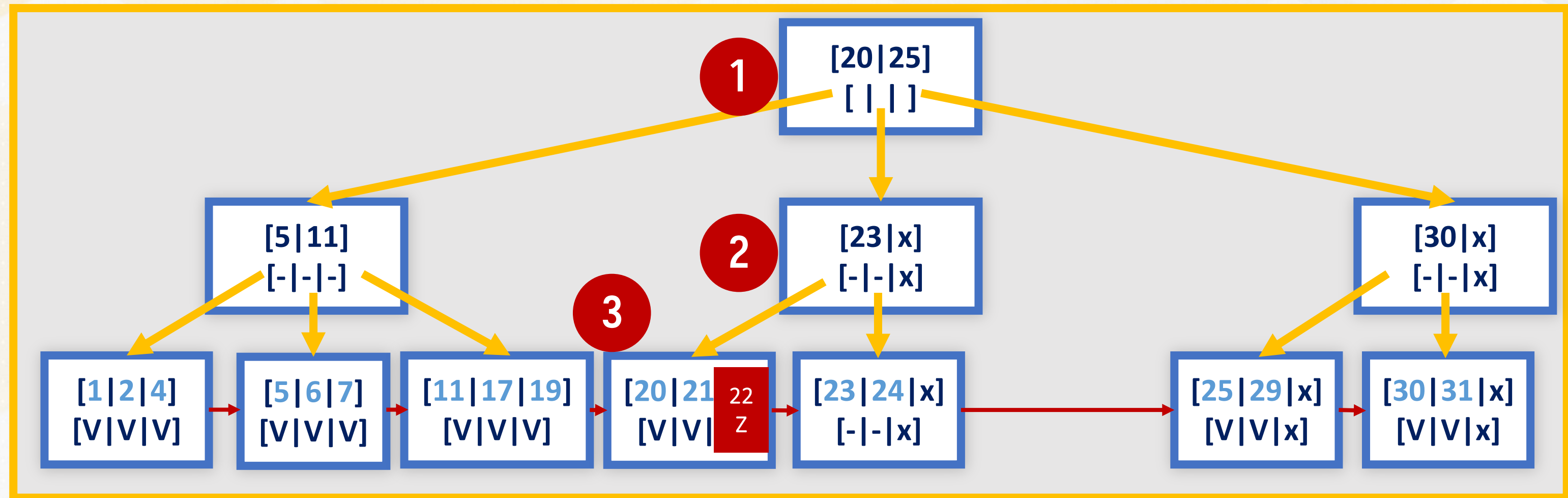
Vector of Entries

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

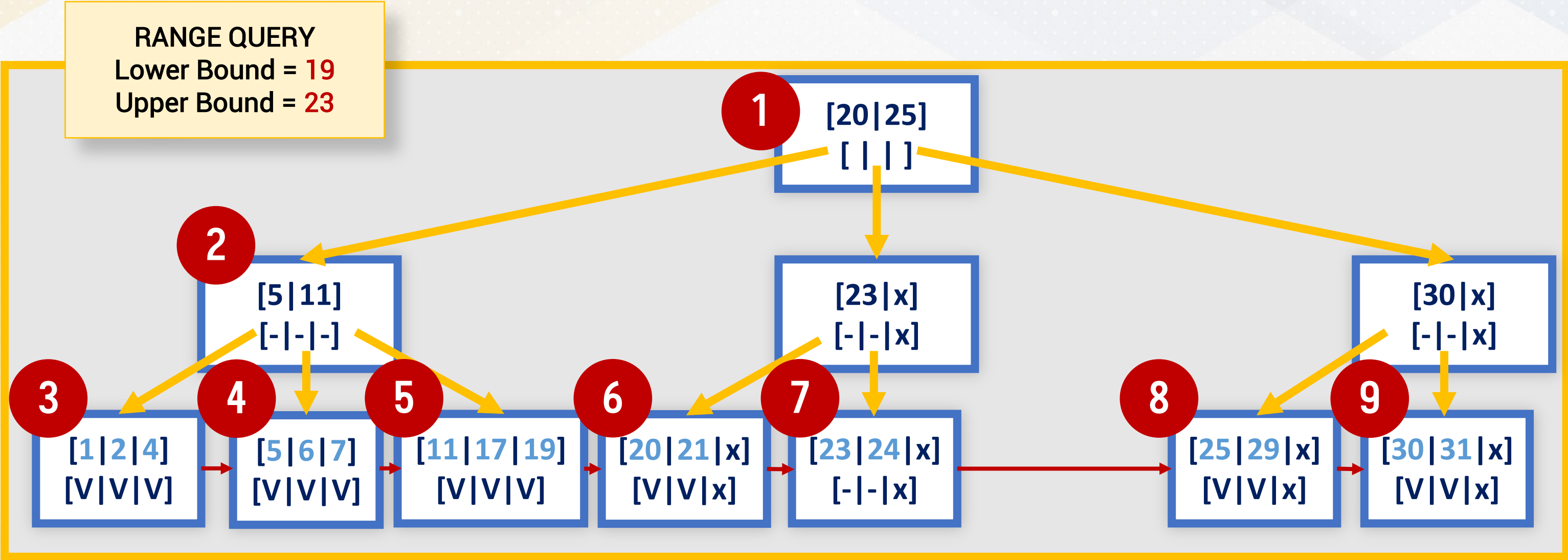
Sorted Order

```
// 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));
```


Range Query Processing



Inefficient Range Query Processing

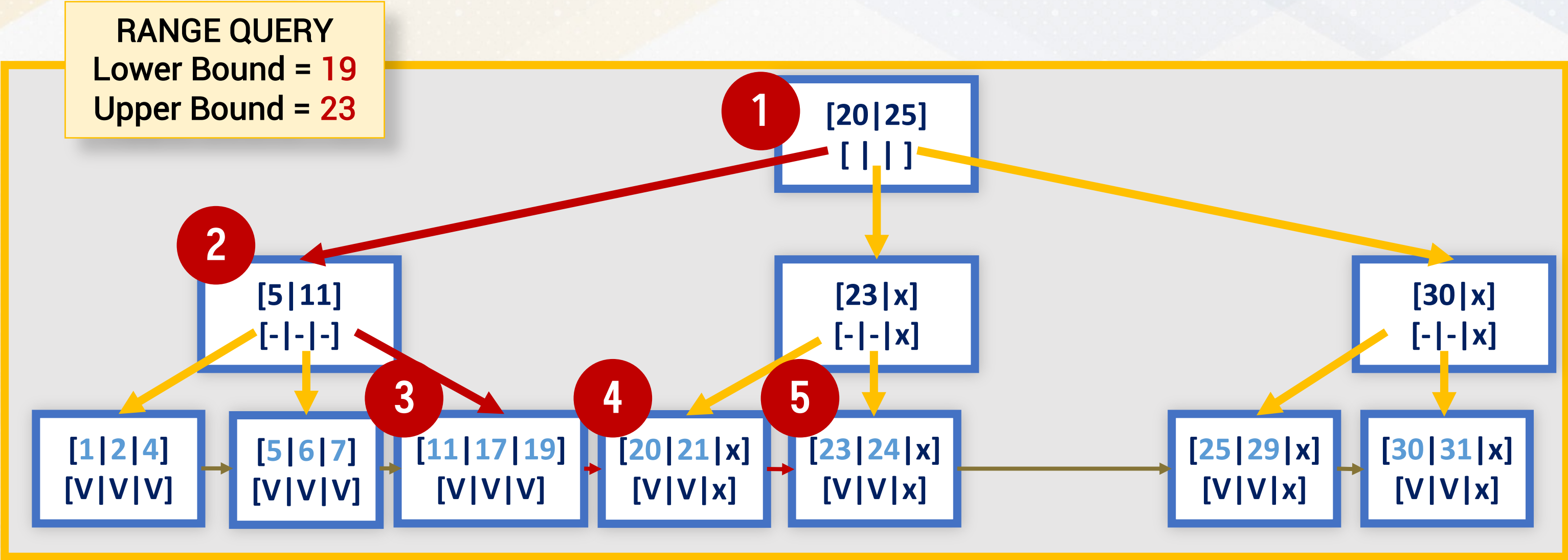


Inefficient Range Query Processing

The rangeQuery method begins at the root and traverses down to the left-most leaf, then continues horizontally through the leaves collecting all qualifying values.

```
std::vector<Value> rangeQuery(const Key &lowerBound,
                             const Key &upperBound) const {
    std::vector<Value> result;
    // Traverse to leftmost leaf
    while (!currentNode->isLeaf) {
        currentNode = currentNode->entries.front().next.get();
    }
    // Traverse all the leaf nodes
    while (currentNode != nullptr) {
        // Add relevant entries within lower and upper bounds
        currentNode = currentNode->next.get();
    }
}
```

Efficient Range Query Processing

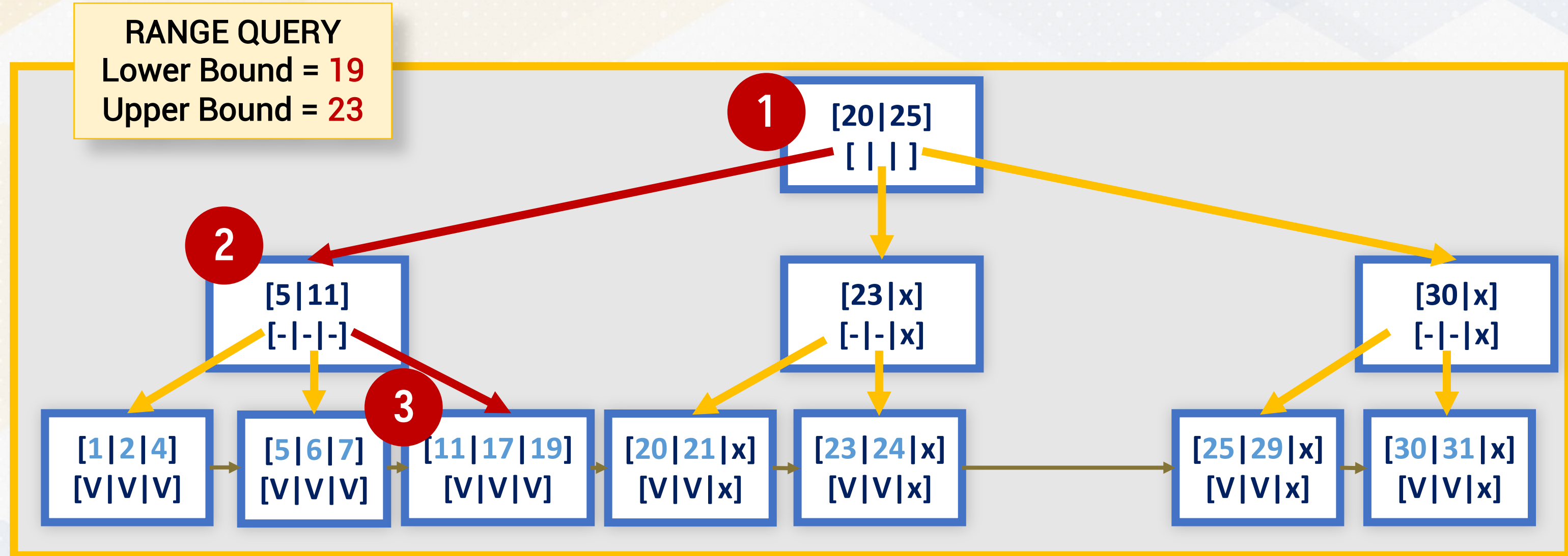


Vertical Traversal to Leaf Node

Move to appropriate child until a leaf node is reached by comparing lowerBound with node key, ensuring the starting point is as close as possible to lowerBound.

```
std::vector<Value> rangeQuery(const Key &lowerBound,
                             const Key &upperBound) const {
    auto node = root;
    while (node && !node->isLeaf) {
        size_t i = 0;
        while (i < node->keys.size() && lowerBound > node->keys[i]) {
            ++i;
        }
        node = node->children[i];
    }
    ...
}
```

Vertical Traversal to Leaf Node

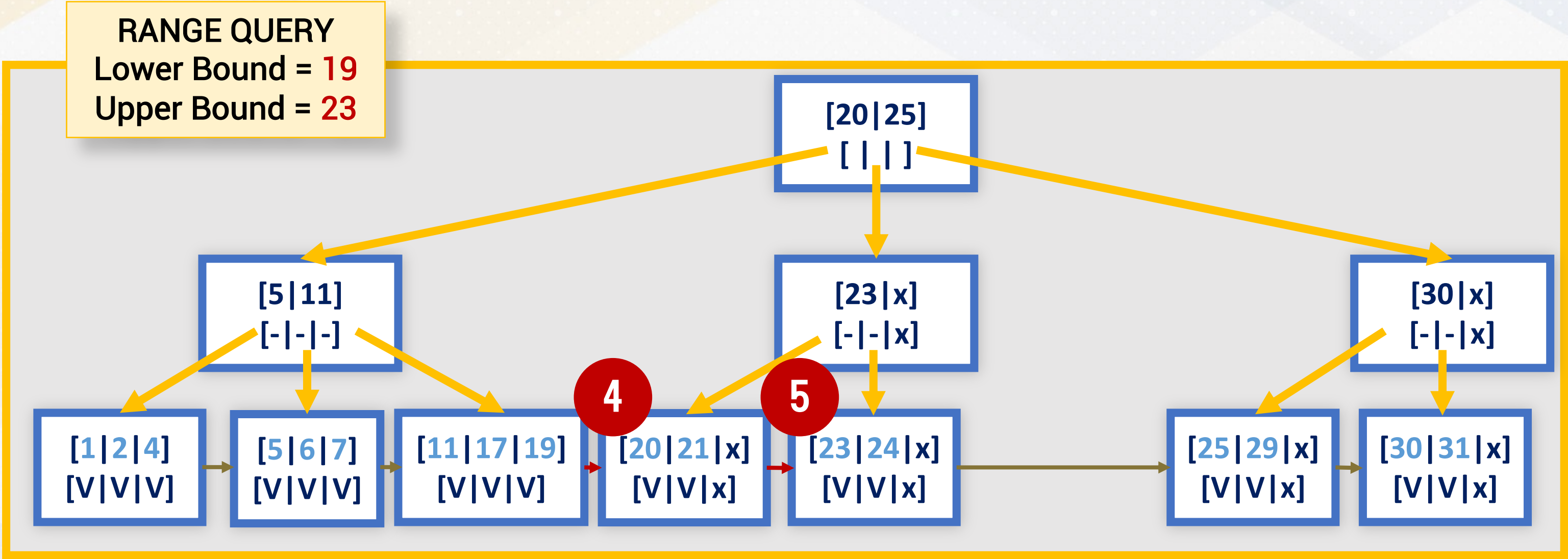


Horizontal Traversal Across Leaf Nodes

Once the appropriate leaf node is reached, the function iterates only through relevant nodes using the linked nature of leaf nodes in a B+Tree.

```
while (node) {  
    for (size_t i = 0; i < node->keys.size(); ++i) {  
        if (node->keys[i] > upperBound)  
            return result;  
        if (node->keys[i] >= lowerBound) {  
            result.push_back(node->values[i]);  
        }  
    }  
    node = node->next;  
}
```

Horizontal Traversal Across Leaf Nodes



Node Split

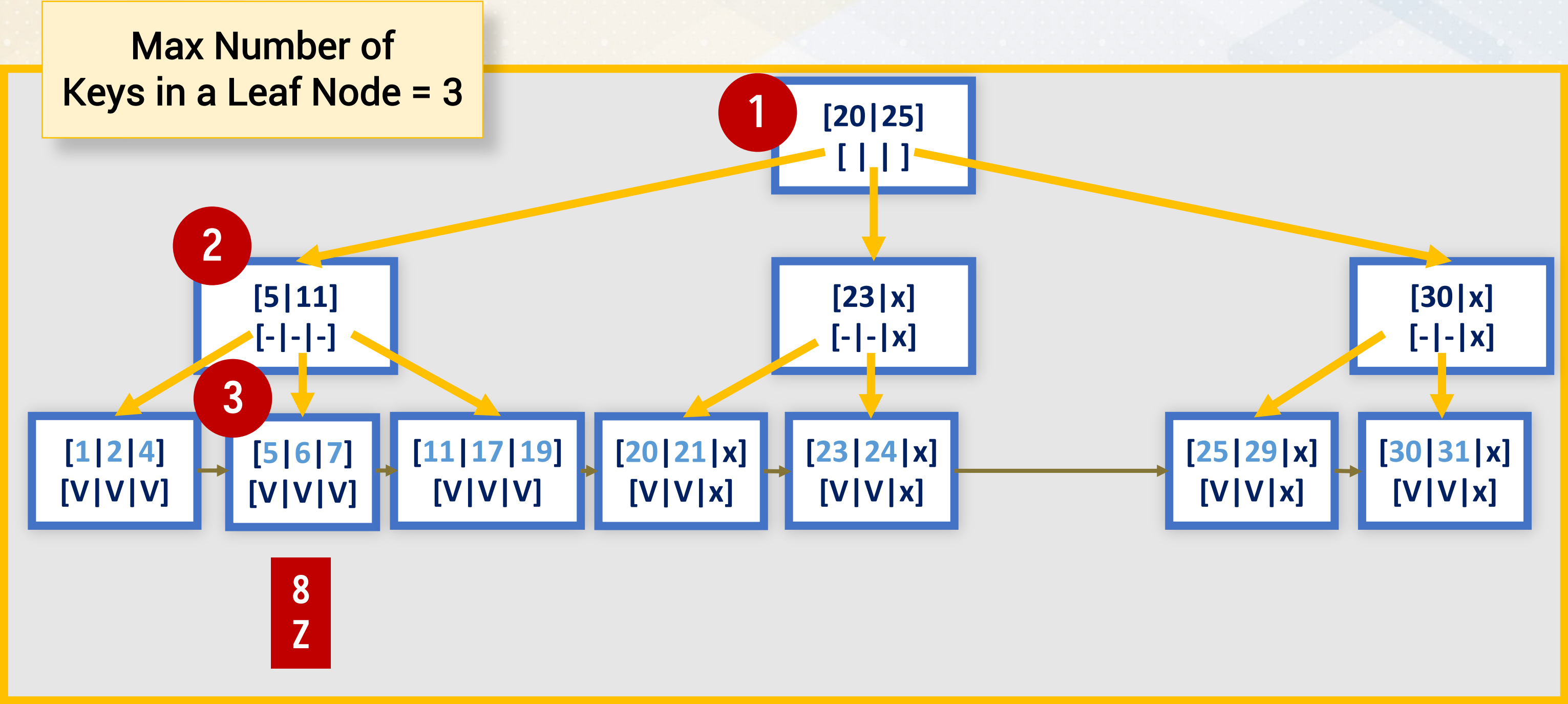


Separate Vectors for Keys and Values

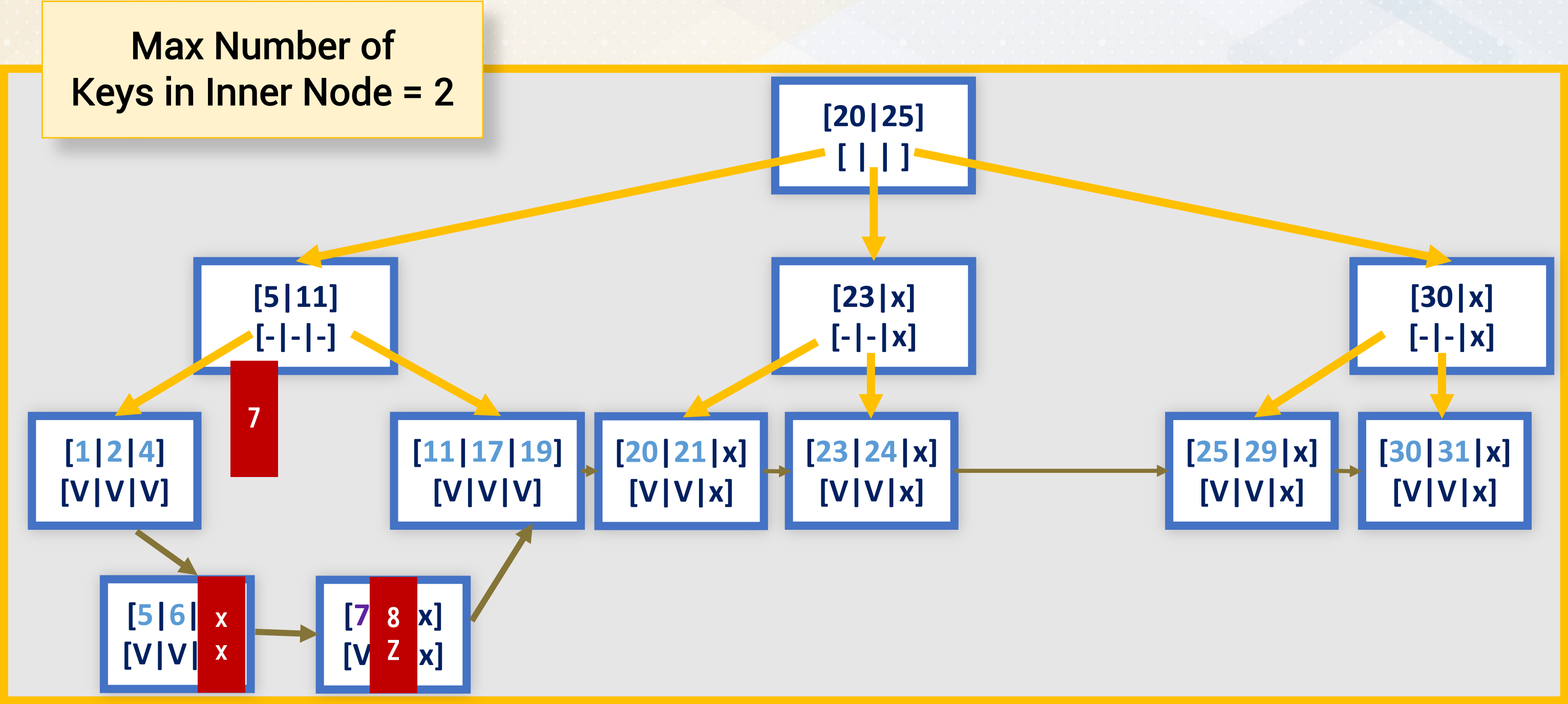
Node contains keys and values in separate vectors.

```
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]

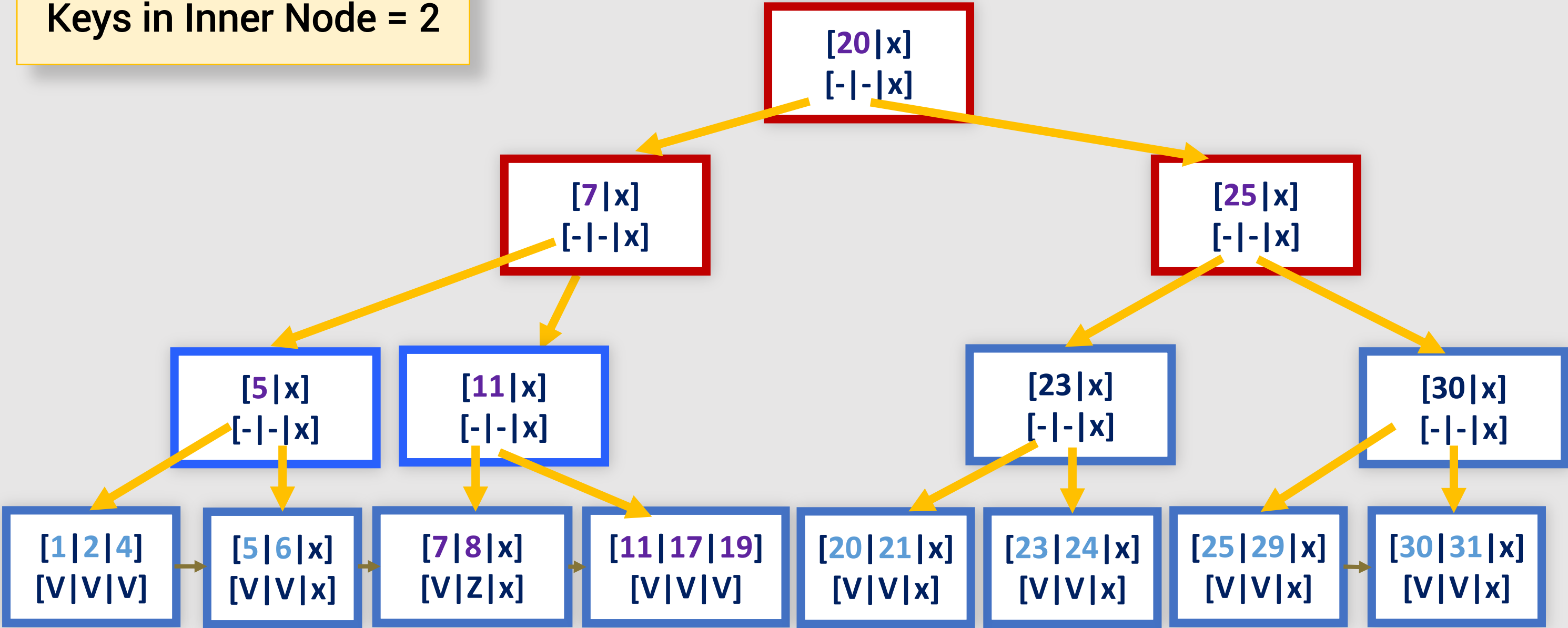


Node Split: Inserting [8, Z]



Node Split: Inserting [8, Z]

Max Number of
Keys in Inner Node = 2



Node Split in C++

buzzDB

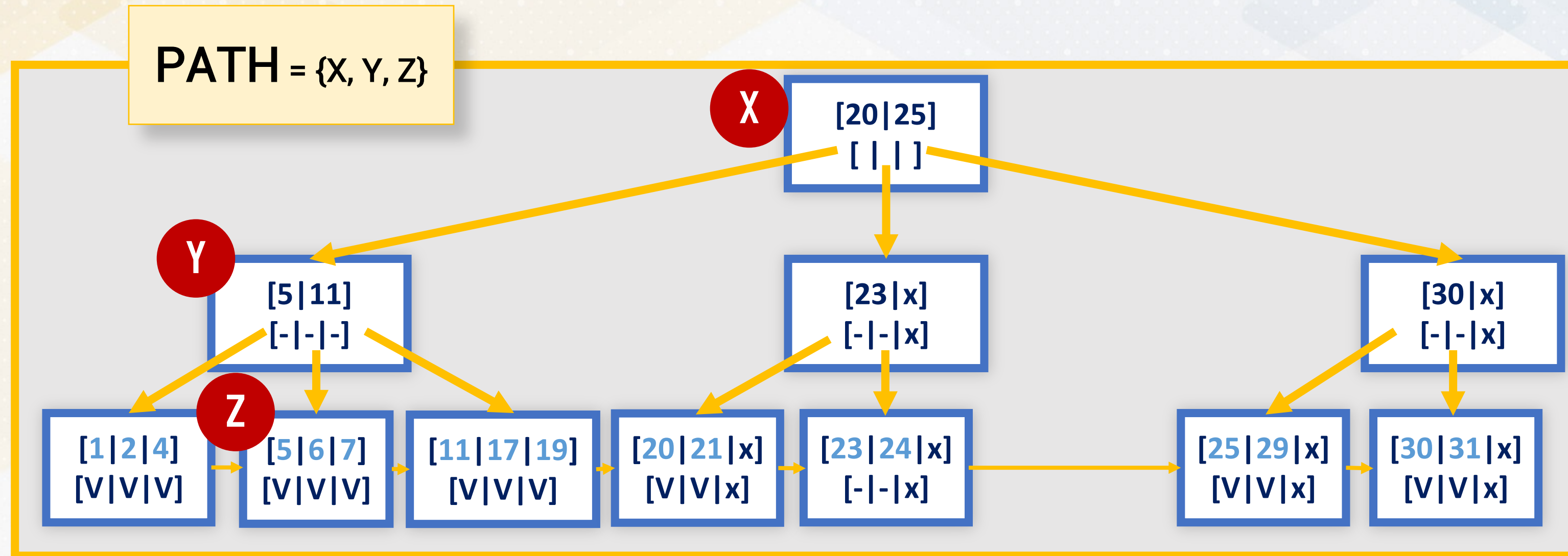
Node Exceeds Capacity → Split Node

```
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;  
}
```


On-Disk B+Tree



Need for On-Disk B+Tree



Scalability

Due to Disk
Capacity



Durability

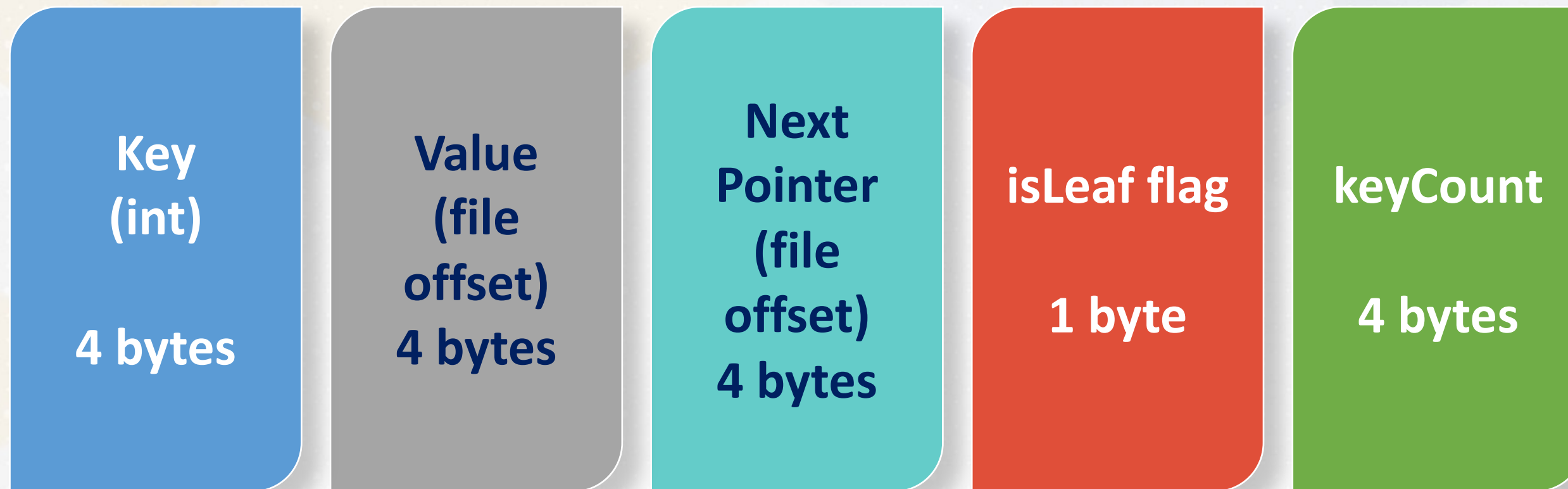
Due to Disk
Durability

Adaptation for Disk



```
struct Node {  
    Key keys[MAX_KEYS];           // Array to store keys  
    uint32_t children[MAX_KEYS + 1]; // File offsets for child nodes  
    uint32_t next;                // File offset for the next leaf node  
    bool isLeaf;  
    int keyCount;  
};  
  
std::vector<uint8_t> serializeNodeToBytes(const DiskNode &node);  
void deserializeNodeFromBytes(const std::vector<uint8_t> &buffer,  
                               DiskNode &node);
```


Calculating Size of Node Struct



Size of Node **struct** with **m** keys =
 $m \times (\text{size of key} + \text{size of value}) + \text{size of an extra value} +$
 $\text{size of next pointer} + \text{size of isLeaf} + \text{size of keyCount} =$
 $m \times (4 + 4) + 4 + 4 + 1 + 4 = m \times 8 + 13$

Aligning Node Struct with Disk Page Size

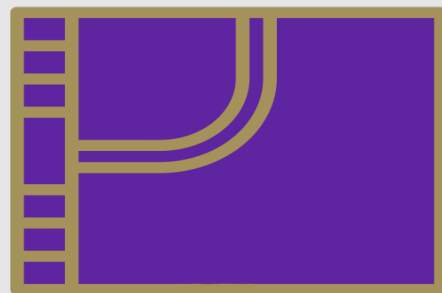


Page

4 kB Page

M = 510 keys

```
// 4 kB disk page size  
m × 8 + 13 ≤ 4096 bytes  
m ≈ 510 keys
```



SSD

16 kB Page

M = 2046 keys

```
// 16 kB disk page size  
m × 8 + 13 ≤ 16384 bytes  
m ≈ 2046 keys
```


Capacity of a B+Tree

Really Wide High Fan Out

Page Size

4 kB

Fan Out

510

Levels

3

Leaf Pages

$$510 ^ { (LEVELS - 1) } = 510 ^ { (3 - 1) } = 510 ^ { 2 } \text{ PAGES}$$

Max Keys

$$510 ^ { (LEVELS) } = 510 ^ { 3 } = 132 \text{ Million KEYS}$$

B+Tree File Size

$$510 ^ { 2 } \text{ PAGES} * 4 \text{ KB} / \text{PAGE} = 1 \text{ GB}$$

Capacity of a B+Tree (4 Levels)

Page Size

4 kB

Fan Out

510

Levels

4

Leaf Pages

$$510 ^ { (LEVELS - 1) } = 510 ^ { (4 - 1) } = 510 ^ { 3 } \text{ PAGES}$$

Max Keys

$$510 ^ { (LEVELS) } = 510 ^ { 4 } = 67 \text{ Billion KEYS}$$

B+Tree File Size

$$510 ^ { 3 } \text{ PAGES} * 4 \text{ KB} / \text{PAGE} = 0.5 \text{ TB}$$

Serialization & Deserialization



Serialization

Convert an in-memory DiskNode structure to a byte stream suitable for disk

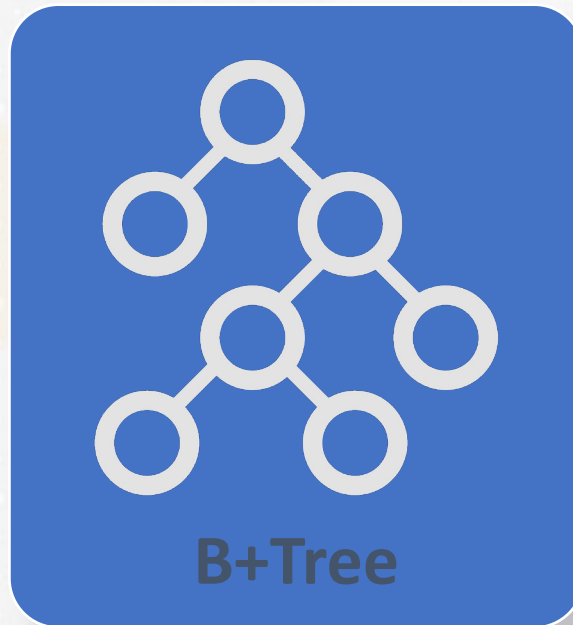
```
std::vector<uint8_t> serializeNodeToBytes(const DiskNode &node) {  
    std::vector<uint8_t> buffer;  
    // Assuming each key and offset is of fixed size, e.g., 4 bytes for int keys  
    for (const auto &key : node.keys) {  
        auto bytes = reinterpret_cast<const uint8_t *>(&key);  
        buffer.insert(buffer.end(), bytes, bytes + sizeof(Key));  
    }  
    ...  
    return buffer;  
}
```


Deserialization

Convert a byte stream read from disk back to an in-memory Node

```
void deserializeNodeFromBytes(const std::vector<uint8_t> &buffer, DiskNode &node) {  
    size_t offset = 0;  
    // Assuming we know the number of keys and that Key is of a fixed size  
    for (int i = 0; i < node.keyCount; ++i) {  
        Key key;  
        std::memcpy(&key, buffer.data() + offset, sizeof(Key));  
        node.keys[i] = key;  
        offset += sizeof(Key);  
    }  
    ...  
}
```

Node Storage in On-Disk File

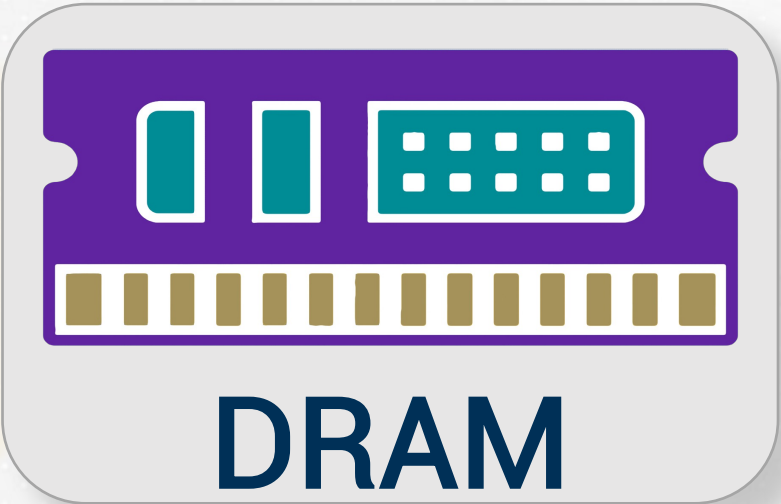


FILE OFFSET	
0	NODE 1
4 KB	NODE 2
12 KB	NODE 3
16 KB	NODE 4
20 KB	NODE 5
24 KB	NODE 6
30 KB	NODE 7
...	...

Minimizing Disk Reads

Reduce Disk
Operations

DOT Node
Cached



N 1	N 5	N 2
N 9	N 6	N 10

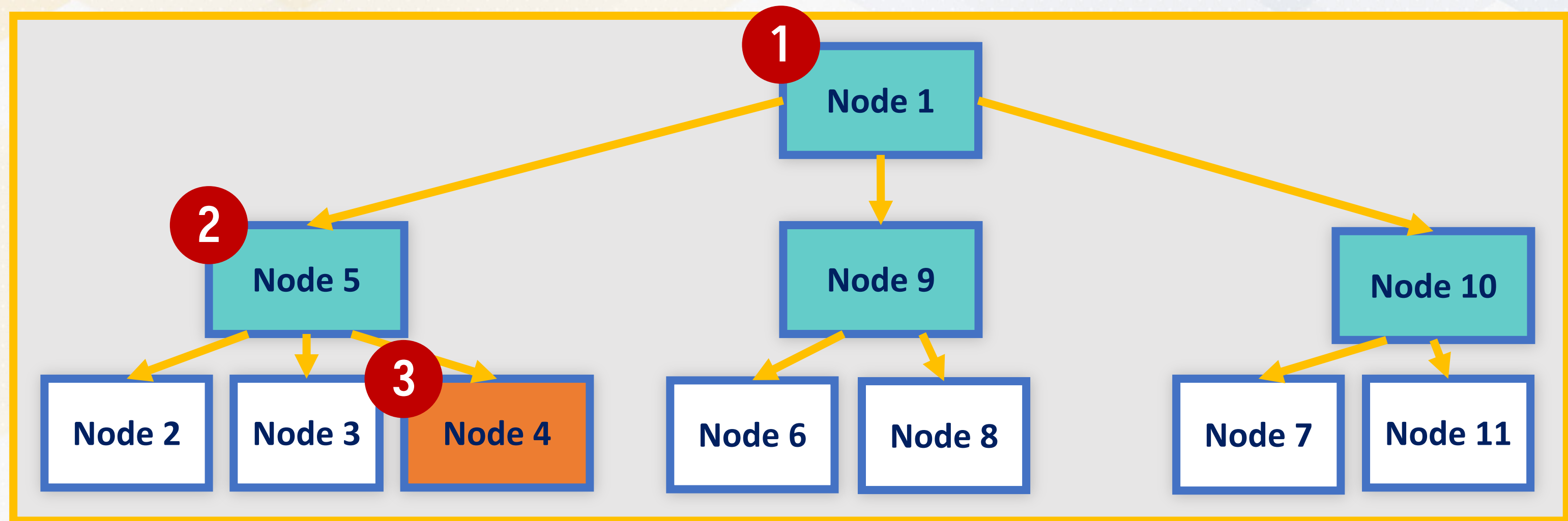
*Cached
Nodes*



N 1	N 2	N 3	N 4
N 5	N 6	N 7	N 8
N 9	N 10	N 11	

B+Tree

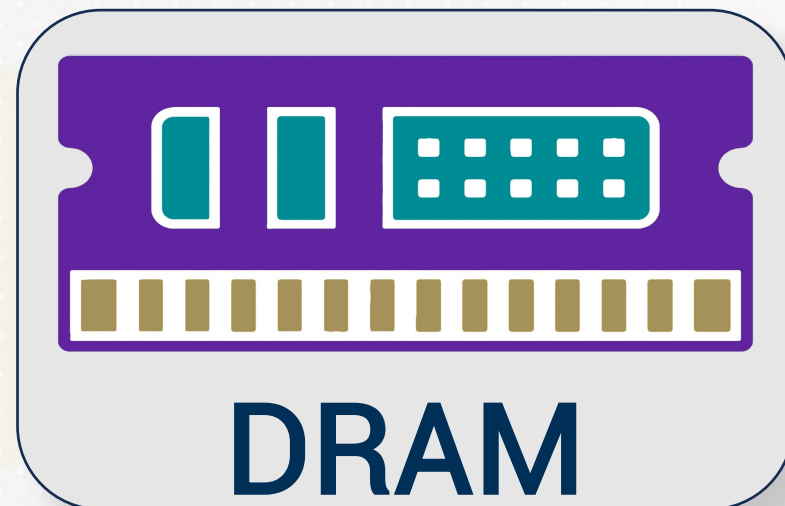
Minimizing Disk Reads: Finding a Key in Node 4



Minimizing Disk Writes

Node Eviction

Node evicted from buffer pool when it is full



N 1	N 5	N 2
N 9	N 6	N 10

*Cached
Nodes*

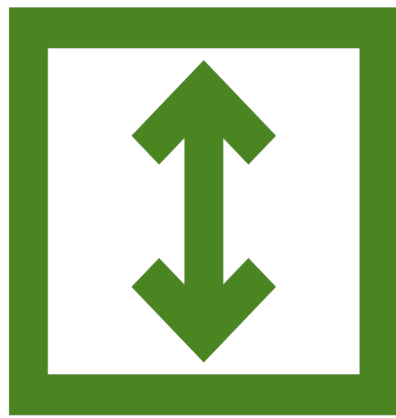


N 1	N 2	N 3	N 4
N 5	N 6	N 7	N 8
N 9	N 10	N 11	

B+Tree

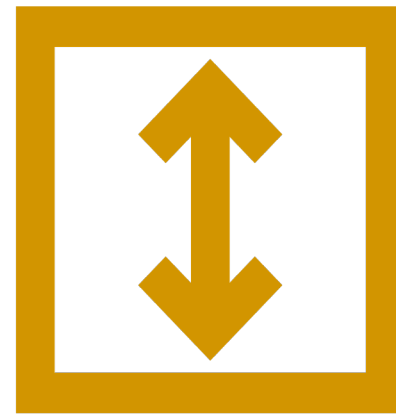
Advantages of B+Trees on Disk

B+Trees are designed for storage on disk



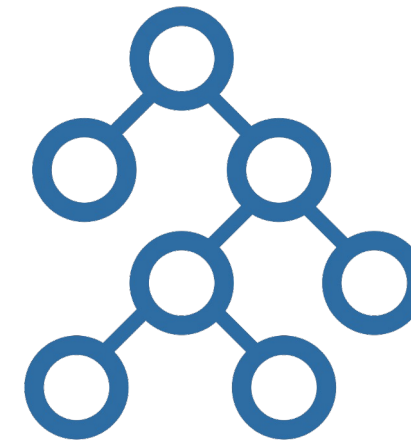
Uniform Height

All operations
need same
number of disk
reads/writes



Short Height

Higher-level
inner nodes are
buffered leading
to fewer page
misses



Linked Leaves

Range queries
can be
processed
efficiently

Conclusion

- Range Query
- Ordered Index
- B+Tree Template