Tutorial 3: Index Structures

Implementation of Databases (DBS2)

Arik Ermshaus

Tutorial appointments

Week	Topic
16.10 - 20.10	-
23.10 - 27.10	Organisation, Exercise Sheet 1
30.10 - 03.11	Q&A
06.11 - 10.11	Q&A
13.11 - 17.11	Exercise Sheet 2
20.11 - 24.11	Q&A
27.11 - 01.12	Q&A
04.12 - 08.12	**Exercise Sheet 3**
11.12 - 15.12	Q&A
18.12 - 22.12	Q&A
25.12 - 29.12	-
01.01 - 05.01	-
08.01 - 12.01	Exercise Sheet 4
15.01 - 19.01	Q&A
22.01 - 26.01	Q&A
29.01 - 02.02	Exercise Sheet 5
05.02 - 09.02	Q&A
12.02 - 16.02	Exam preparation

Cancelled Q&A sessions:
- 14.12.2023
- 19.12.2023

Disclaimer: Timetable is provisional, and will (probably) change!

Table of Contents

- **Solutions of Exercise Sheet 2**
- Exercise Sheet 3
- B+ Trees

Task 1: Fixed and Variable-length Records

- DB stores collection of books
- Attributes: 5 required (64 bytes each), 15 optional (16 bytes each)
- Optional attributes are present with probability *p*

(a) Calculate size of “fixed-length” records in bytes. *Hint:* Empty optional fields are filled with NULL values.

- $r_{fixed} = 5 \cdot 2^6 B + 15 \cdot 2^4 B = 560B$

Task 1: Fixed and Variable-length Records

- DB stores collection of books
- Attributes: 5 required (64 bytes each), 15 optional (16 bytes each)
- Optional attributes are present with probability *p*

(b) Calculate expected size of “variable-length” records in bytes.
End of attribute is terminated with tag of 2 bytes.

- $r_{variable} = 5 \cdot 2^6 B + 15 \cdot p \cdot 2^4 B + 20 \cdot 2B$
- $r_{variable} = 360B + p \cdot 240B$

Task 1: Fixed and Variable-length Records

- DB stores collection of books
- Attributes: 5 required (64 bytes each), 15 optional (16 bytes each)
- Optional attributes are present with probability *p*

(c) For which range of probabilities *p* should one favour fixed-length records?

- $r_{fest} < r_{variable} \leftrightarrow 560B < 360B + p \cdot 240B$
- $\dfrac{200B}{240B} < p \leftrightarrow \dfrac{5}{6} < p\ (\approx 0.83)$

- For probabilities $p > \dfrac{5}{6}$, one should favour fixed-length records
- Expected to be smaller than variable-sized ones

Task 2: Storing Relations in Blocks

- DB stores 2 relations; authors and books, associated by foreign key
- 5k authors (512 bytes each), 20k books (128 bytes each)
- Blocks: 4096 bytes, 64 bytes reserved for header information

(a) How many blocks are required, if authors and books are stored separately?

- $$b_{authors} = \frac{5000}{\left\lfloor \frac{4096B - 64B}{512B} \right\rfloor} = \frac{5000}{7} \approx 715$$
- $$b_{books} = \frac{20000}{\left\lfloor \frac{4096B - 64B}{128B} \right\rfloor} = \frac{20000}{31} \approx 646$$

- In total, one needs $b_{authors} + b_{books} = 715 + 646 = 1361$ blocks

Task 2: Storing Relations in Blocks

- DB stores 2 relations; authors and books, associated by foreign key
- 5k authors (512 bytes each), 20k books (128 bytes each)
- Blocks: 4096 bytes, 64 bytes reserved for header information

(b) How many blocks are required, if authors are stored with their books? *Hint:* Books are distributed equally over authors.

- $$b_{all} = \frac{5000}{\left\lfloor \frac{4096B - 64B}{512B + 4 \cdot 128B} \right\rfloor} = \frac{5000}{3} \approx 1667$$

- In total, one needs $b_{all} = 1667$ blocks

Task 2: Storing Relations in Blocks

- DB stores 2 relations; authors and books, associated by foreign key
- 5k authors (512 bytes each), 20k books (128 bytes each)
- Blocks: 4096 bytes, 64 bytes reserved for header information

(c) Which scenario (subtask a or b) is preferable, if query outputs entire author relation? Justify.

- Scenario (a) is preferable, because DB needs to read 715 instead of 1667 blocks

Task 3: Buffer Manager

Complete the provided buffer manager data structure with LRUn replacement strategy.

```cpp
std::shared_ptr<Block> BufferManager::fix_block(std::string const& block_id)
{
    // check if the block is already in cache
    auto it = cache.find(block_id);

    if (it != cache.end()) {
        // increase reference count and return
        it->second.reference_count++;
        return it->second.block;
    }

    // if cache is full and no block can be evicted, throw error
    if (cache.size() >= n_blocks && unfixed_blocks.empty()) {
        throw std::runtime_error("Cannot fix block. Cache is already full.");
    }

    // evict the least recently unfixed block if necessary
    if (cache.size() >= n_blocks) {
        // get ID of the least recently unfixed block
        std::string block_id_to_evict = unfixed_blocks.front();
        unfixed_blocks.pop_front();

        // evict block and write to disk if it's dirty
        std::shared_ptr<Block> block_to_evict = cache[block_id_to_evict].block;
        if (block_to_evict->is_dirty()) {
            block_to_evict->write_data();
        }

        // remove block from cache
        cache.erase(block_id_to_evict);
    }

    // load block into cache
    std::shared_ptr<Block> block = std::make_shared<Block>(block_id);
    // set reference count to 1
    cache[block_id] = {block, 1};

    return block;
}
```

```cpp
bool Block::write_data()
{
    // open file handle
    std::ofstream file(BLOCK_DIR + get_block_id(), std::ios::binary | std::ios::out);

    if (!file.is_open())
        return false;

    // write data
    char* buffer = static_cast<char*>(data.get());
    file.write(buffer, BLOCK_SIZE);

    // close file handle and reset dirty flag
    file.close();
    dirty = false;

    return true;
}
```

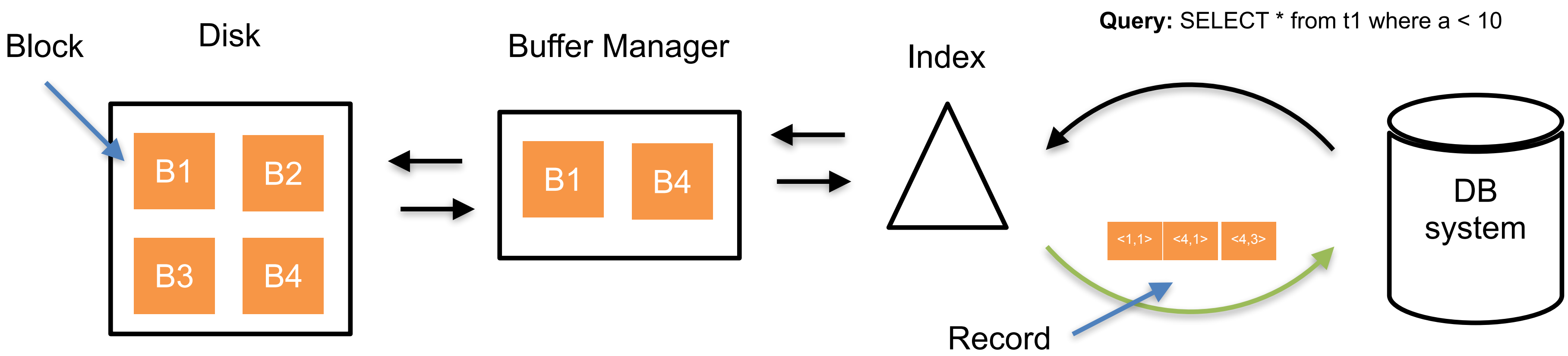
```cpp
bool BufferManager::unfix_block(std::string const& block_id)
{
    auto it = cache.find(block_id);

    if (it == cache.end())
        throw std::invalid_argument("Cannot unfix block that is not in cache.");

    if (it->second.reference_count == 0)
        throw std::invalid_argument("Cannot unfix a block without fixes.");

    // Decrease the reference count
    it->second.reference_count--;

    // If the reference count is zero, add to the list of unfixed blocks
    if (it->second.reference_count == 0 && std::find(unfixed_blocks.begin(), unfixed_blocks.end(),
block_id) == unfixed_blocks.end()) {
        unfixed_blocks.push_back(block_id);
    }

    return true;
}
```

Table of Contents

- Solutions of Exercise Sheet 2
- **Exercise Sheet 3**
- B+ Trees

Table of Contents

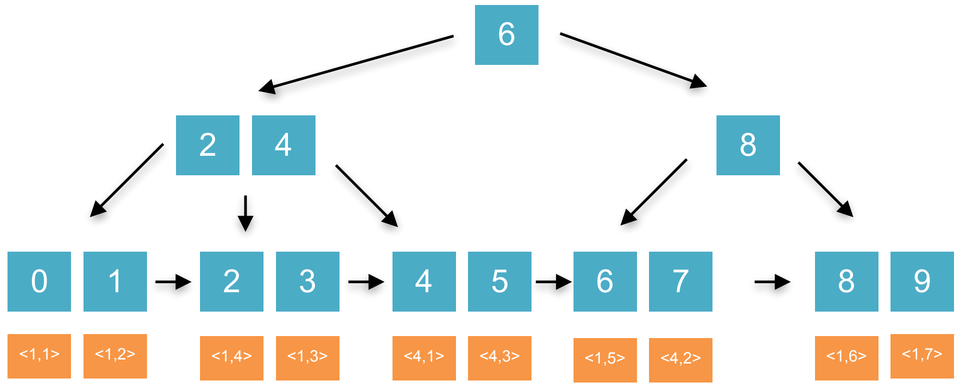
Recap: From Attributes to Record IDs



Question: How do you retrieve the relevant tuples?

- Problem: Queries ask for tuples based on data model; **not** record IDs
- Task: Retrieve all relevant tuples based on attributes
- Typical solutions: Scanning, hash files, index structures
- Challenges: selectivity of searches, IO access overhead, keeping data structures updated, degeneration

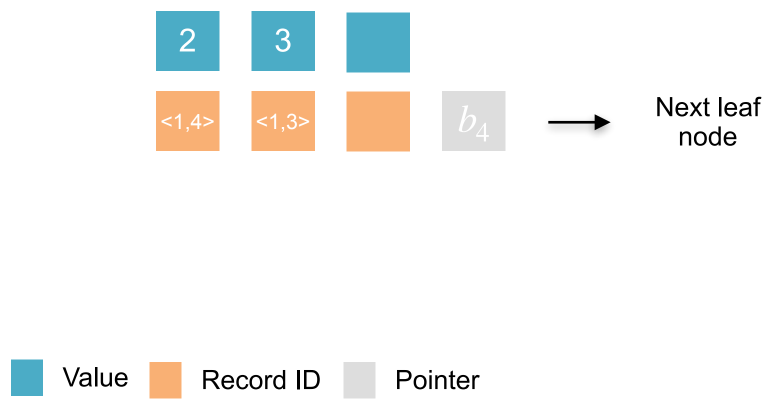
Recap: B+ Tree



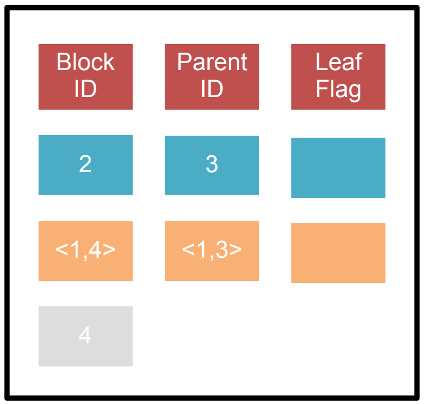
- *m*-ary tree structure with (typically) large number of children (per node)
- Structure: Root, internal nodes and leaves
  - Internal nodes: Signposts to leaf nodes
  - Leaves: (attribute, tuple id) pairs
- Block-oriented nodes with high fanout
- All CRUD operations in $\mathcal{O}(\log_k b)$ IO accesses
  - with *k* values per node and *b* blocks in total

Example: Leaf nodes

Data Structure



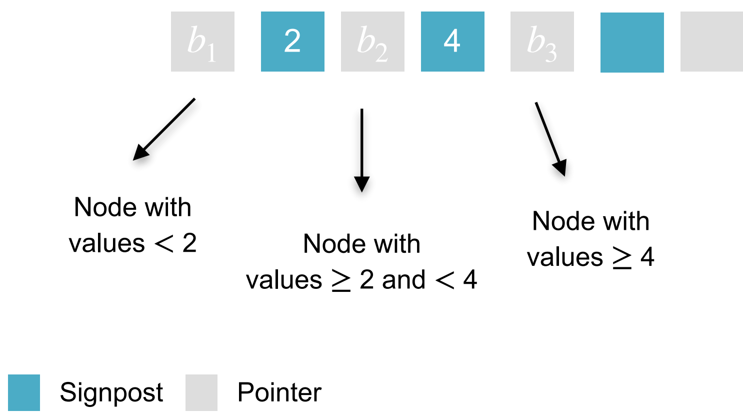
Block Layout



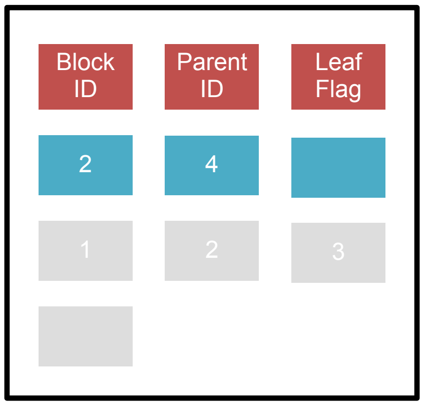
- Number of values / record IDs: $\lceil \frac{k}{2} \rceil$ to k (e.g. k=3)
- 1 pointer that links the subsequent leaf
- Values are ordered

Example: Internal nodes

Data Structure

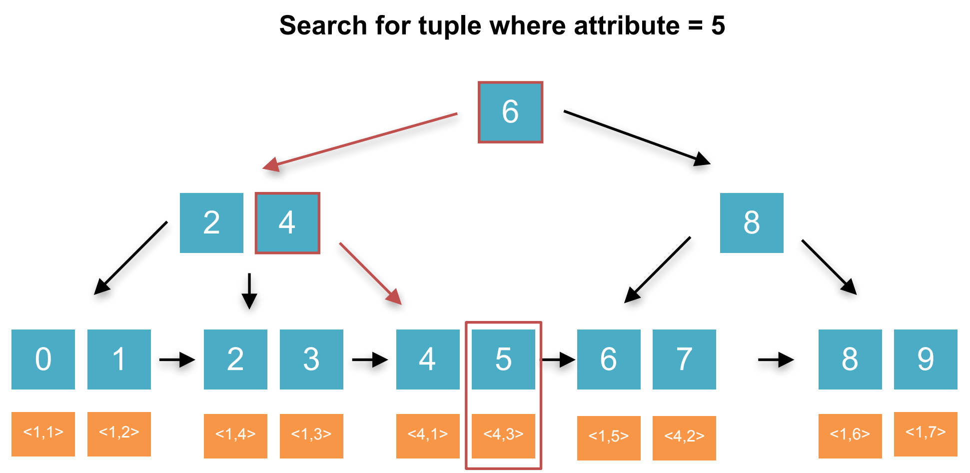


Block Layout



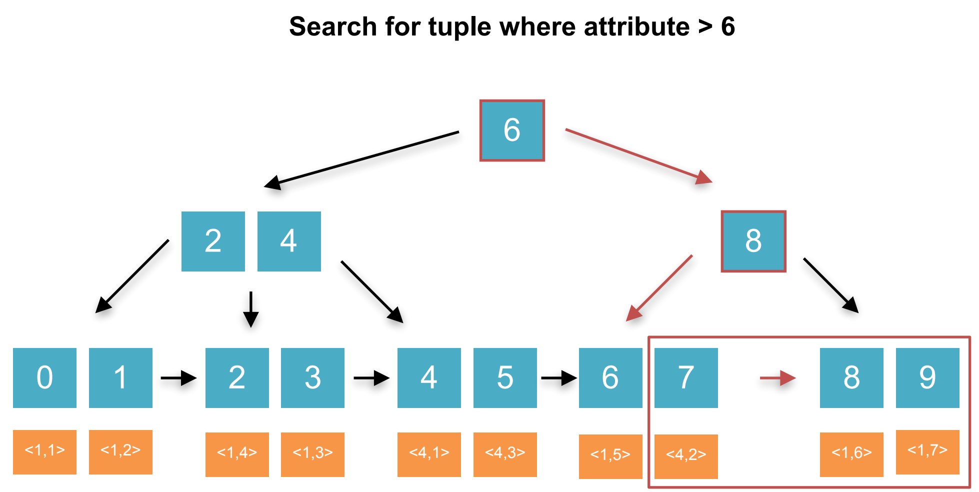
- Number of values: $\lfloor \frac{k}{2} \rfloor$ to k (e.g. k=3)
- Number of pointers: $\lfloor \frac{k}{2} \rfloor + 1$ to $k + 1$ (e.g. k=3)
- Values are ordered, pointers link to respective subtrees

Example: Search



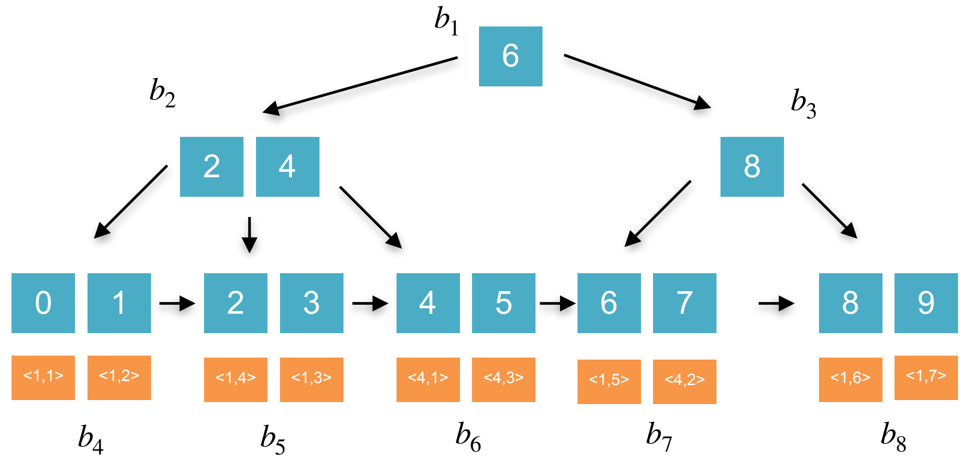
1. Find correct leaf node for search attribute v
   - Starting at the root node …
   - Traverse to pointer where $v <$ signpost
   - … or traverse to last pointer where $v \geq$ signpost
   - Repeat with new node until leaf is found
2. Search v in leaf node, return tuple ID
   - Duplicates: Traverse leaf node(s) to the right until new value
   - Ranges: Search smallest value, traverse leaf node(s) right to largest value

Example: Range Search



1. Find correct leaf node for search attribute v
   - Starting at the root node …
   - Traverse to pointer where $v <$ signpost
   - … or traverse to last pointer where $v \geq$ signpost
   - Repeat with new node until leaf is found
2. Search v in leaf node, return tuple ID
   - Duplicates: Traverse leaf node(s) to the right until new value
   - Ranges: Search smallest value, traverse leaf node(s) right to largest value

Task 1: B+ Tree Search



- **Question**: Assume a DB system can permanently fix 3 of the 8 index blocks. Which ones would be optimal to reduce IO access?

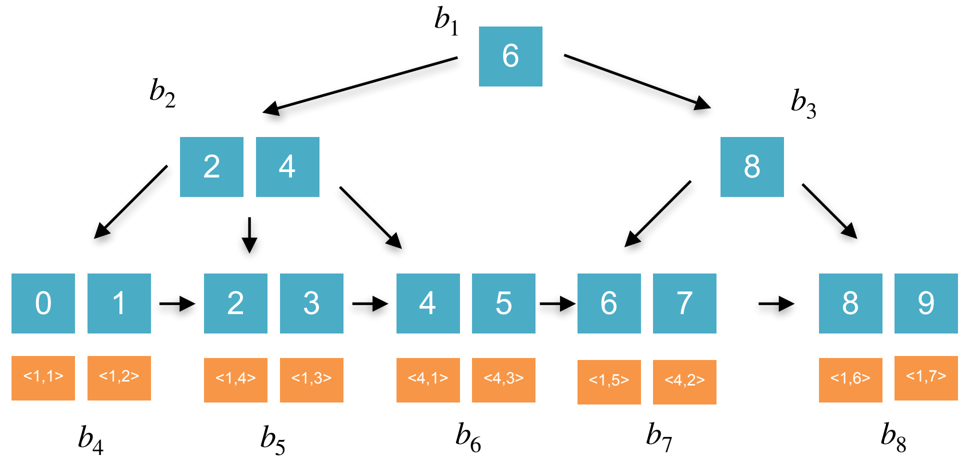
(A) b_1,b_2,b_4

(B) b_4,b_5,b_6

(C) b_1,b_2,b_3

(D) b_1,b_3,b_8

Task 1: B+ Tree Search



- **Question**: Assume a DB system can permanently fix 3 of the 8 index blocks. Which ones would be optimal to reduce IO access?

(A) b_1, b_2, b_4

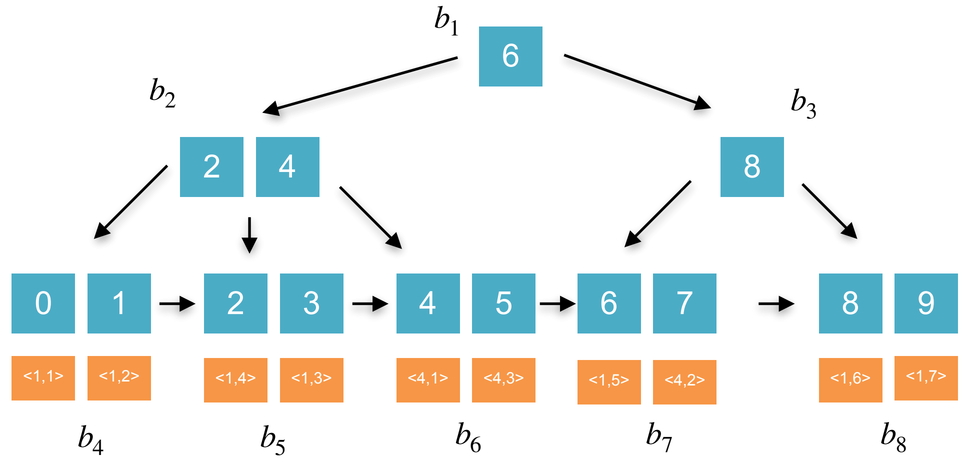
(B) b_4, b_5, b_6

(C) b_1, b_2, b_3

(D) b_1, b_3, b_8

Internal nodes are accessed more frequently than leaves

Task 2: B+ Tree Search



- **Question**: How many index blocks must be accessed to search for the attribute 3?

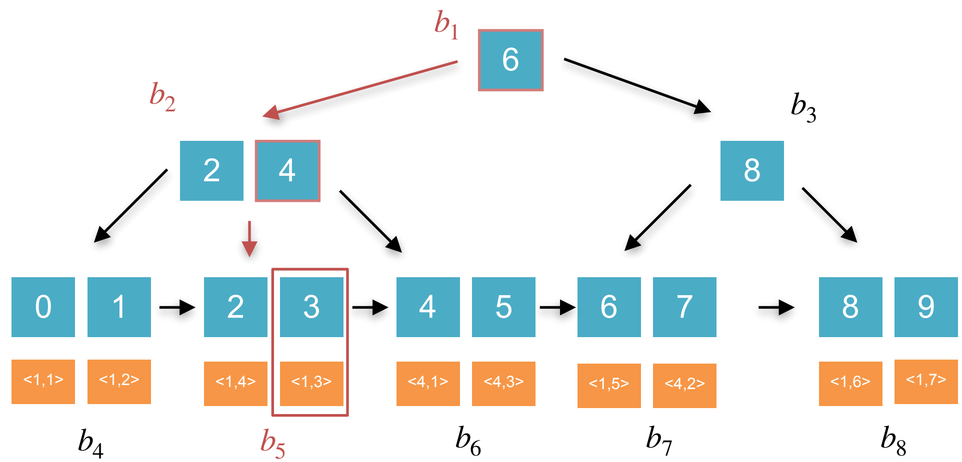
(A) 1

(B) 2

(C) 3

(D) 4

Task 2: B+ Tree Search

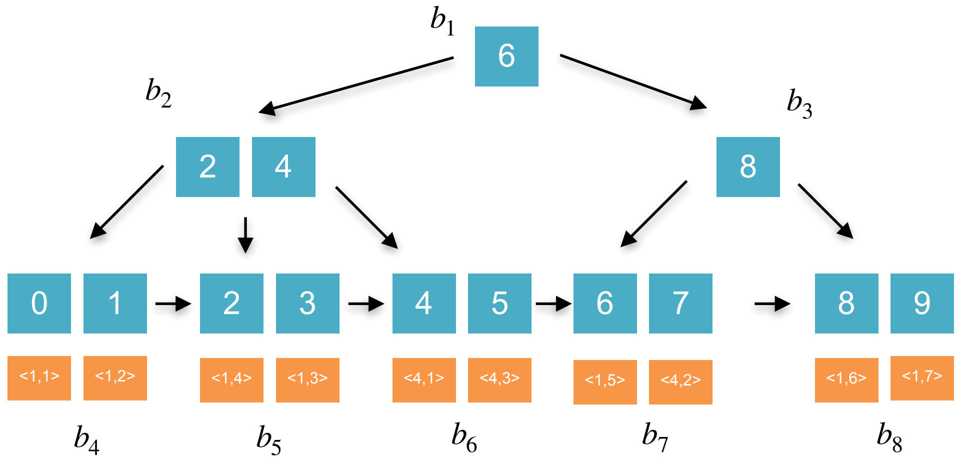


- **Question**: How many index blocks must be accessed to search for the attribute 3?

(A) 1	(B) 2	(C) 3	(D) 4

We access b_1, b_2, b_5

Task 3: B+ Tree Search



- **Question**: How many index blocks must be accessed to search for the attributes < 6?

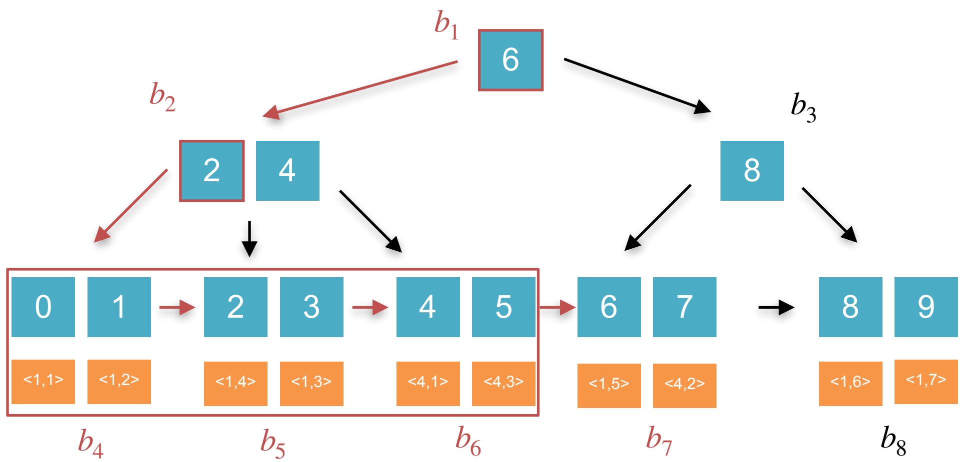
(A) 7

(B) 6

(C) 5

(D) 3

Task 3: B+ Tree Search



- **Question**: How many index blocks must be accessed to search for the attributes < 6?

(A) 7	(B) 6	(C) 5	(D) 3

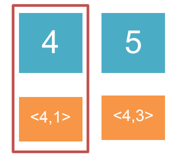
We access b_1, b_2, b_4, b_5, b_6, b_7

Example: Insert

Insert: (5, <4,3>); Steps: 1,2

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

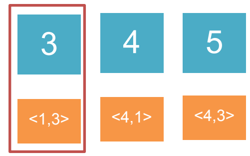
Example: Insert



Insert: (4, <4,1>); Steps: 1,2

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

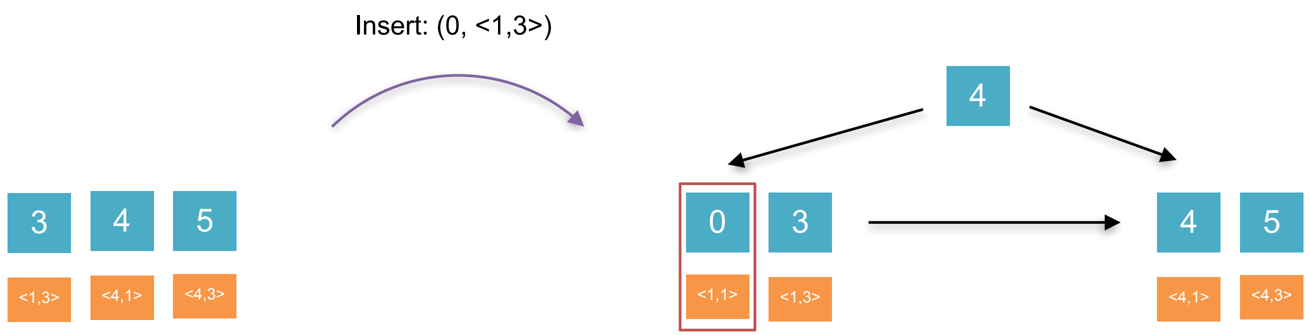
Example: Insert



Insert: (3, <1,3>); Steps: 1,2

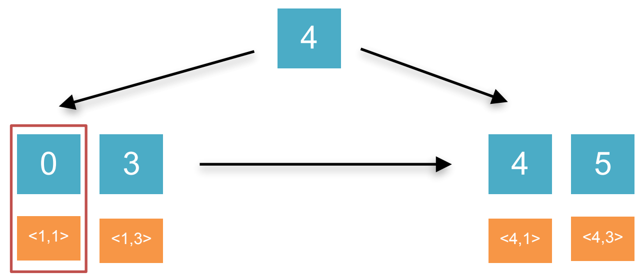
1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

Example: Leaf Splitting



1. Insert (v, t_{id}) at correct position and split node in two halves
   - Original leaf node has $\lceil \frac{k+1}{2} \rceil$ and new one has $\lfloor \frac{k+1}{2} \rfloor$ values (e.g. k=3)
2. **Copy** left-most value of new node to parent, adjust pointers
3. Set new node's parent ID accordingly

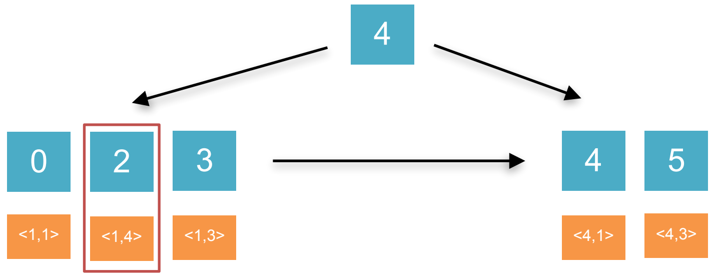
Example: Insert



Insert: (0, <1,3>); Steps: 1,3,4

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
    - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
    - If root splits, create new root as parent and execute step 4 and 5

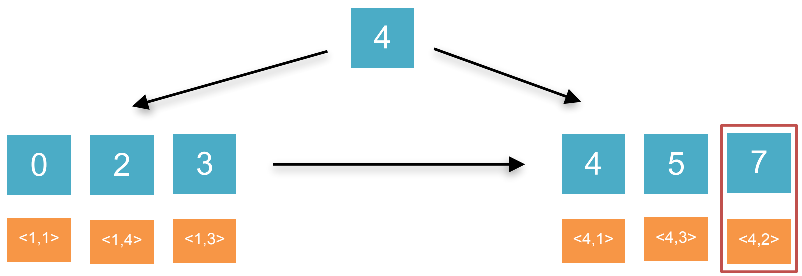
Example: Insert



Insert: (2, <1,4>); Steps: 1,2

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

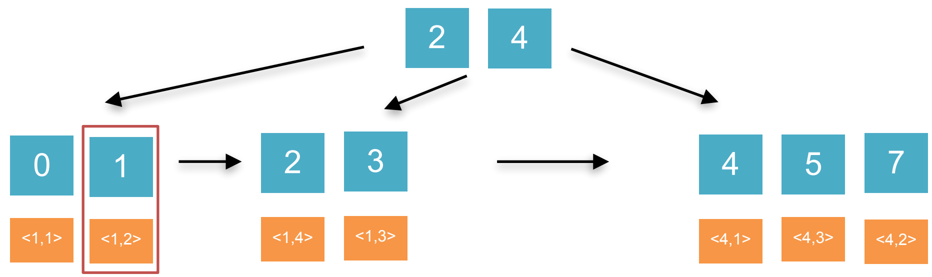
Example: Insert



Insert: (7, <4,2>); Steps: 1,2

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
    - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
    - If root splits, create new root as parent and execute step 4 and 5

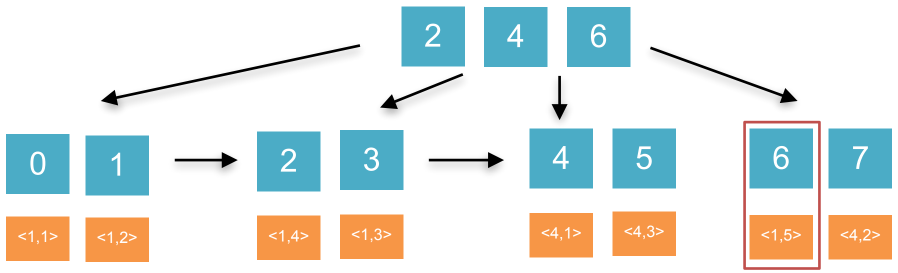
Example: Insert



Insert: (1, <1,2>); Steps: 1,3,4

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

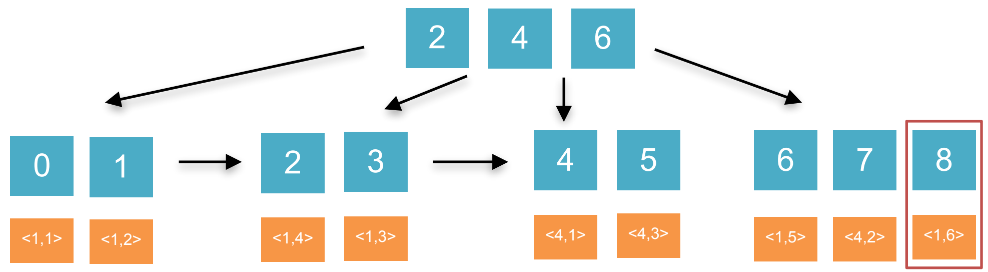
Example: Insert



Insert: (6, <1,5>); Steps: 1,3,4

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

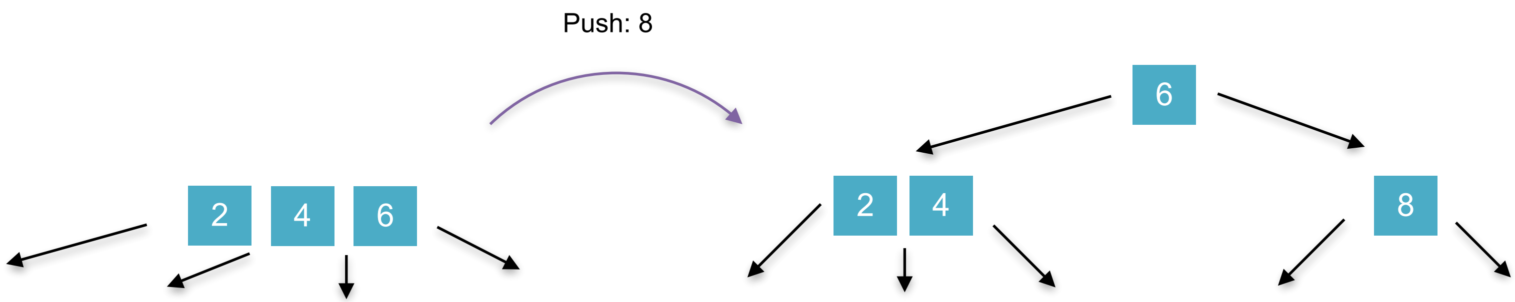
Example: Insert



Insert: (8, <1,6>); Steps: 1,2

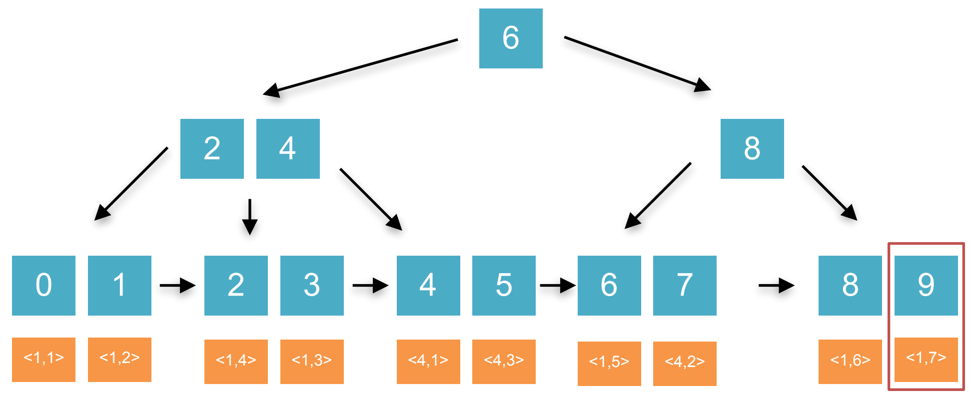
1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
   - If root splits, create new root as parent and execute step 4 and 5

Example: Internal Node Splitting



1. Insert v at correct position and split node in two halves
   - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values (e.g. k=3)
2. **Push** left-most value of new node to parent, adjust pointers
3. Update parent IDs accordingly

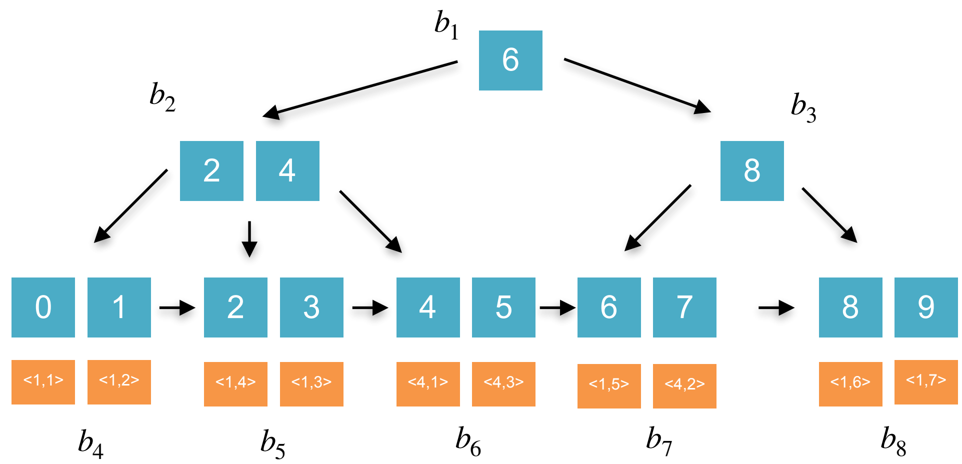
Example: Insert



Insert: (9, <1,7>); Steps: 1,3,4,3,4

1. Search correct leaf node to insert attribute tuple ID pair (v, t_{id})
2. If leaf is not full, insert (v, t_{id}) and stop
3. Otherwise, insert (v, t_{id}) and split node into two nodes
    - Original internal node has $\lceil \frac{k}{2} \rceil$ and new one has $\lfloor \frac{k}{2} \rfloor$ values
4. Copy/Push left-most value of new node to parent, adjust pointers
5. Repeat until parent found that needs no split
    - If root splits, create new root as parent and execute step 4 and 5

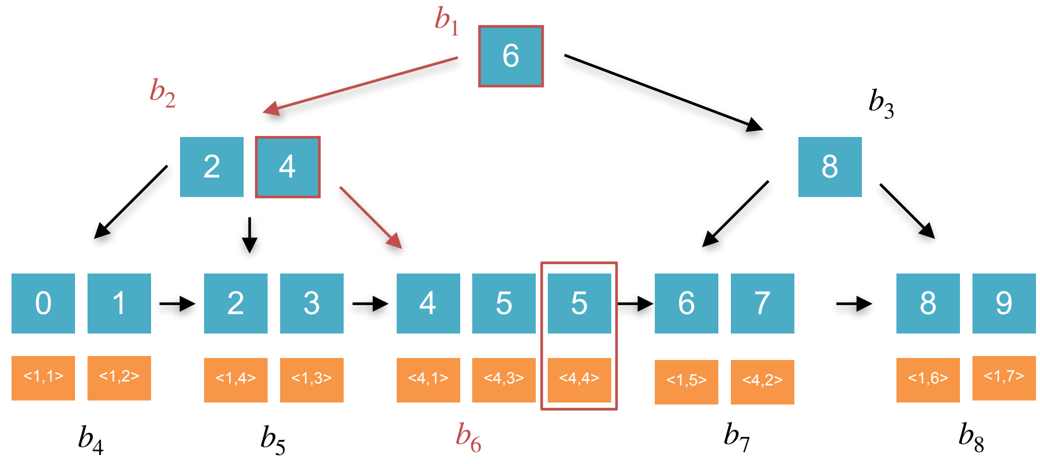
Task 4: B+ Tree Insert



- **Question**: In which block will the entry (5, <4,4>) be inserted?

(A) b_2 | (B) b_3 | (C) b_5 | (D) b_6

Task 4: B+ Tree Insert



- **Question**: In which block will the entry (5, <4,4>) be inserted?

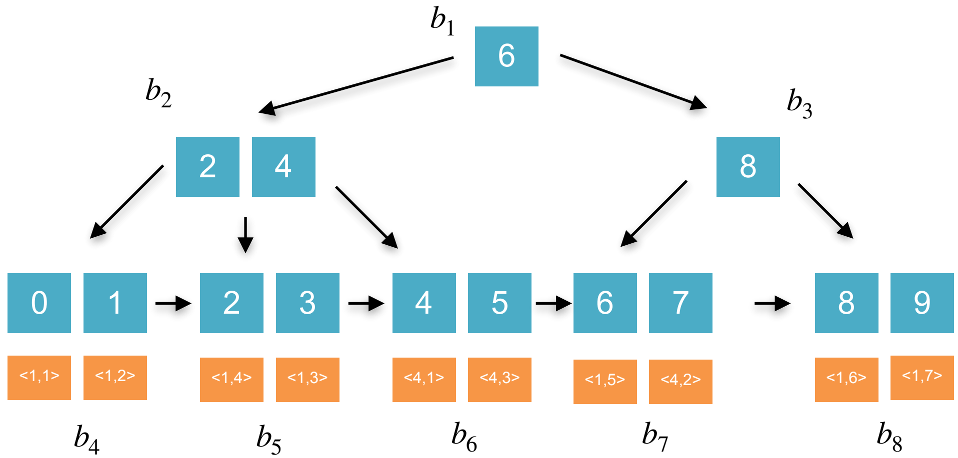
(A) b_2

(B) b_3

(C) b_5

(D) b_6

Task 5: B+ Tree Insert



- **Question**: Which of the following aggregates can be used to compute signposts in the node splitting process?

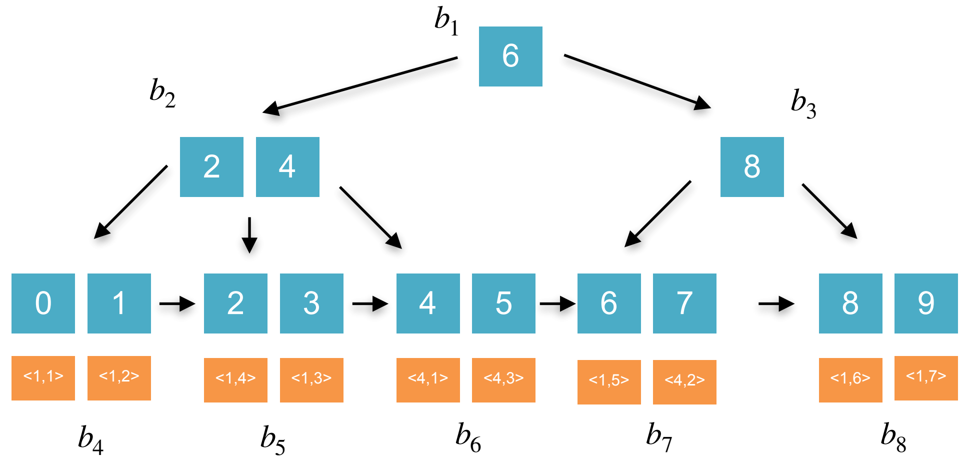
(A) Median

(B) Max

(C) Min

(D) Mode

Task 5: B+ Tree Insert



- **Question**: Which of the following aggregates can be used to compute signposts in the node splitting process?

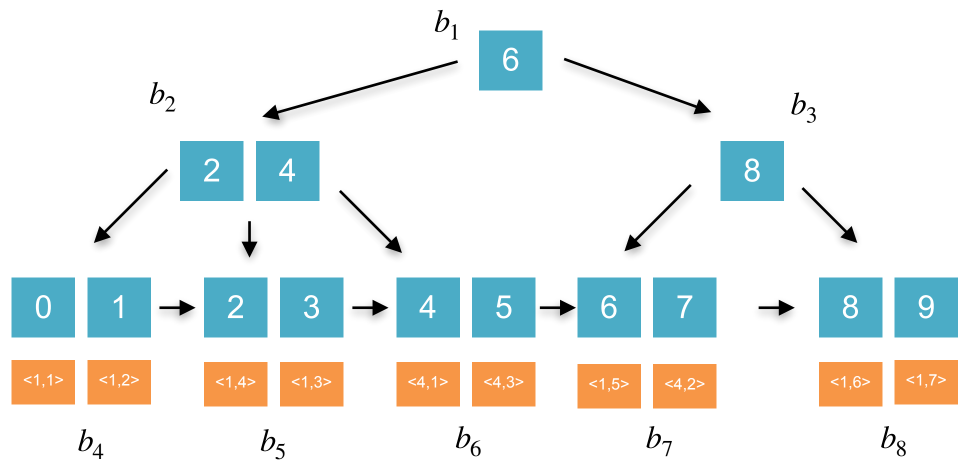
(A) Median

(B) Max

(C) Min

(D) Mode

Task 5: B+ Tree Insert

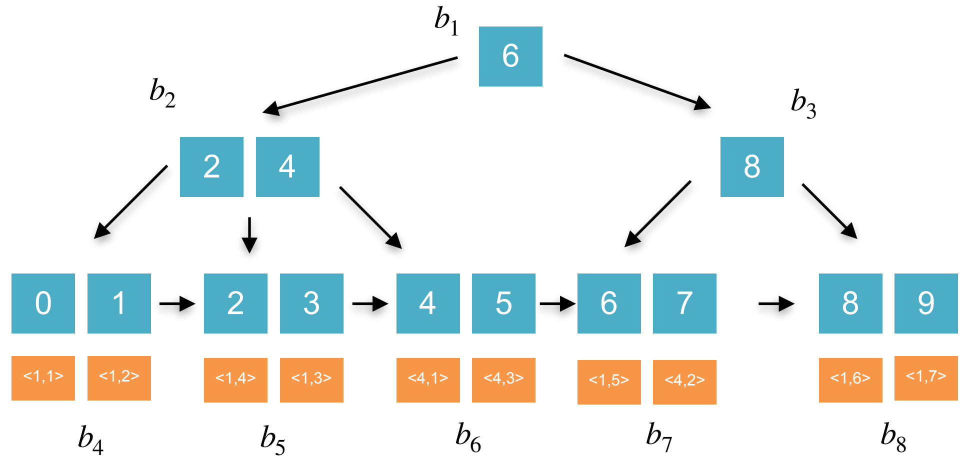


- **Question**: In which direction does a B+ tree grow?

(A) Downwards

(B) Upwards

Task 5: B+ Tree Insert



- **Question**: In which direction does a B+ tree grow?

(A) Downwards

(B) Upwards

Parent nodes are created while splitting