Database Implementation

Thomas Schwarz, SJ

Memory Hierarchy

- Almost stable since the 1970s
  - Cache — Fast, Expensive, Volatile
  - DRAM memory — Fast, Expensive, Volatile
  - Storage: HDD, SDD — Slow, Cheap, Non-volatile
  - Tertiary Storage: offline HDD, Tape — Very slow, very cheap, non-volatile

Memory Hierarchy

- Database data is stored:
  - Usually in storage
    - Accessed not using the file system, but by direct block access
  - Sometimes in virtual memory (Main Memory Databases)

Memory Hierarchy

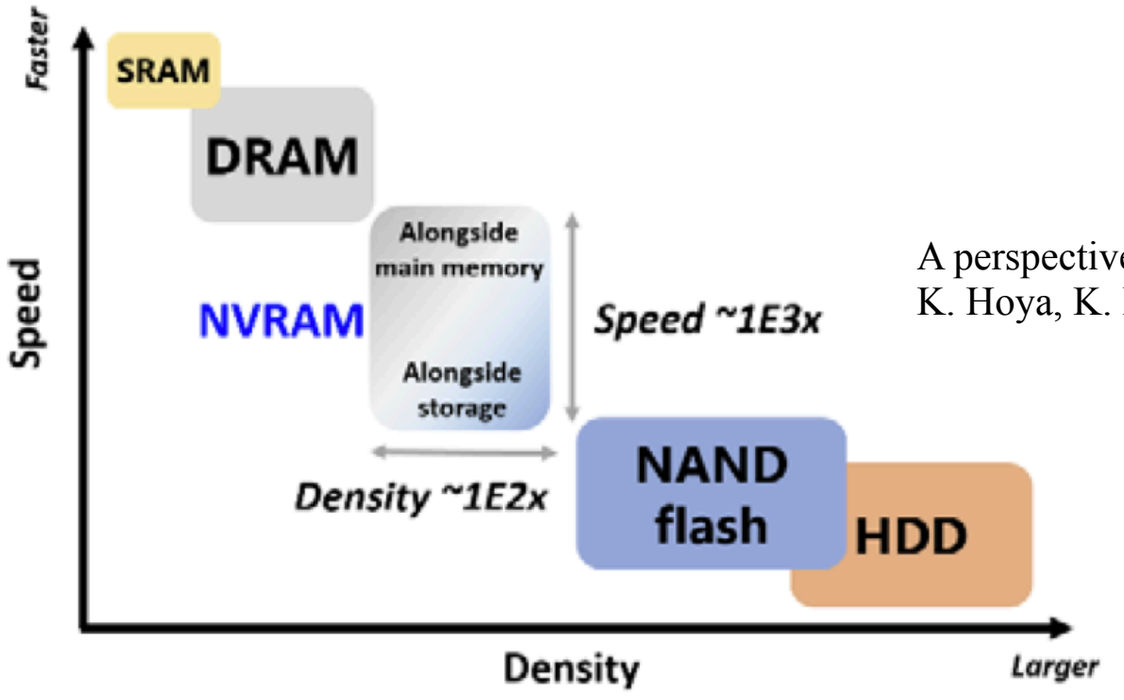
- Volatility implies:
  - All changes to a database need to be logged
    - If there is a loss of power:
      - Database needs to be in a consistent state
        - Use the log to rebuild the consistent state from a snapshot and a log
- Data needs to be transferred between storage and CPU

Memory Hierarchy

- Durability with fallible components
  - All storage systems are fallible
    - Use mirroring or redundant storage

Memory Hierarchy

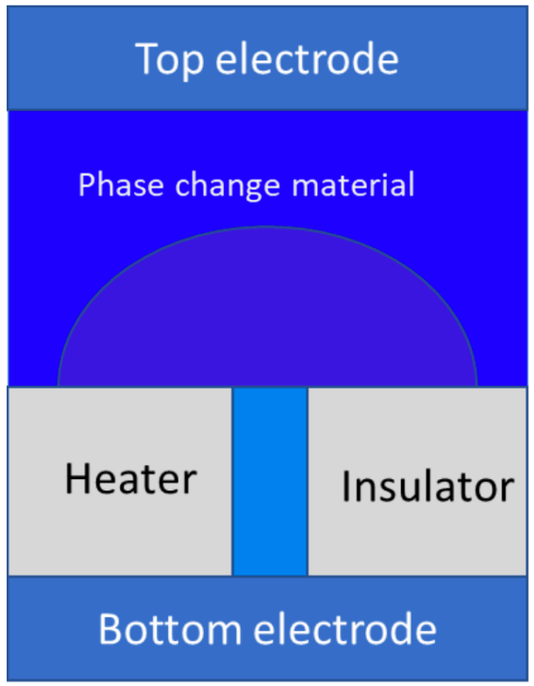
- Memory hierarchy might be in for a big change
  - Non-volatile memories



A perspective on NVRAM technology for future computing system, K. Hoya, K. Hatsadu,K. Tsuchida, Y. Watanabe, Y. Shirota, T. Kanai

Phase Change Memories

- Phase change material is in crystalline (low resistance) or amorphous (high resistance) state
- To change phase:
  - Short burst of current leads to amorphous state
  - Longer, but lower burst of current leads to crystalline state



Phase Change Memories

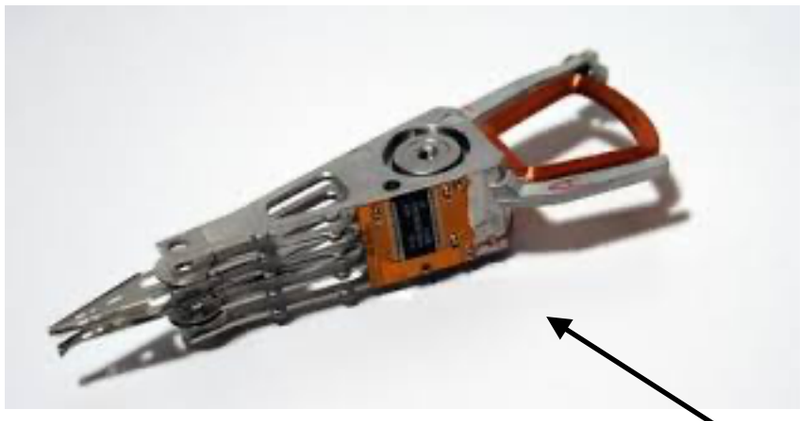
- Almost as fast as RAM
- Denser than Flash —> Will become cheaper
- On the market as Intel Optane

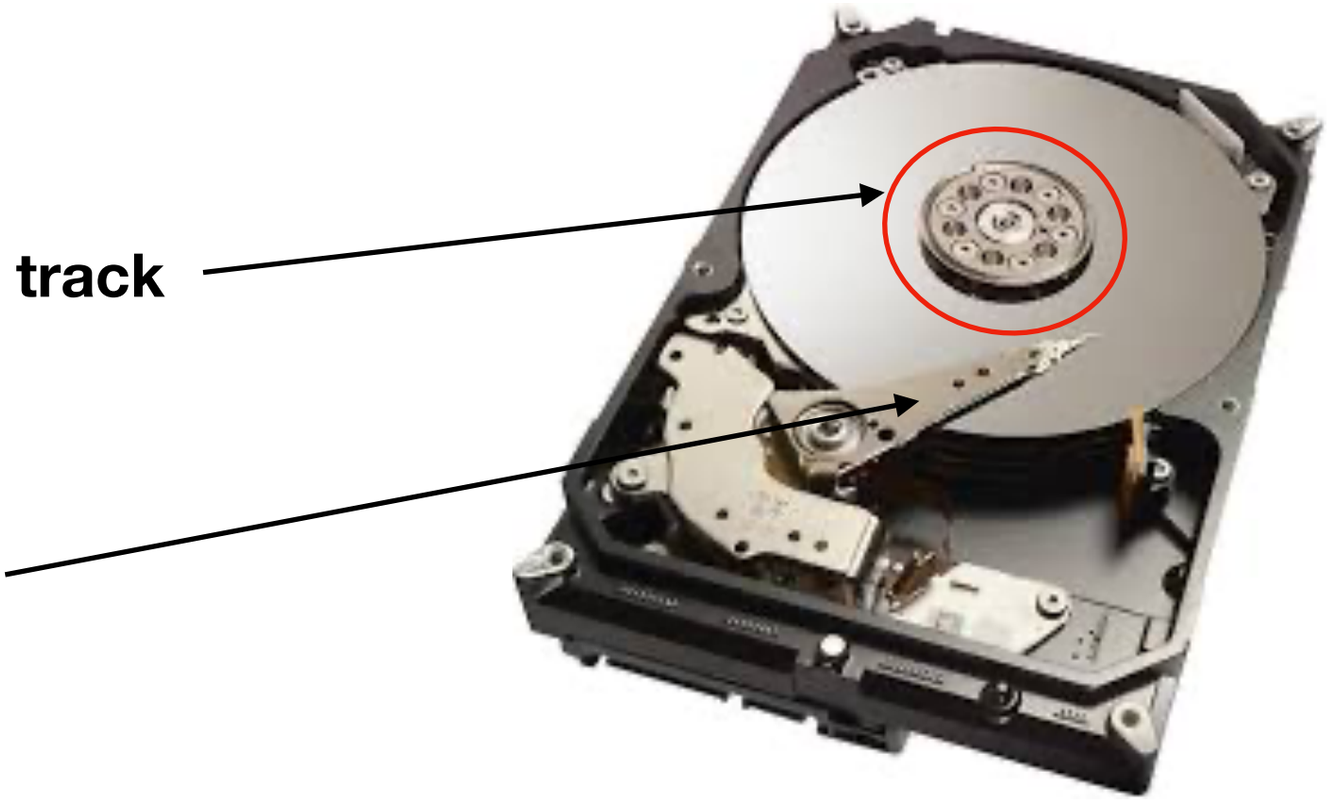
Memory Hierarchy

- With new technologies
  - Non-volatile memories might be:
    - Replacing /supplementing memory
    - Replacing / supplementing storage
    - Replace both with a single layer
      - with large implications for OS design and database design

Optimizing disk access

- Disk access:
  - Move actuator over track — *seek time*
  - Wait for block to appear under track — rotation time
  - Transfer data == transfer time





Optimizing disk access

- Avoid seek times:
  - Keeping table blocks together on the track
    - Streaming bandwidth ~ 200 MB/sec
    - Store most frequently access data in the middle tracks of the platter
    - Access data track by track (elevator algorithm)
- Hide access times:
  - Prefetching and large scale buffering
    - Example: Writes can be delayed and ordered

Optimizing disk access

- SSD:
  - No mechanical parts —> no seek times
  - Can use parallelism
  - But: need to use wear leveling because flash can only be safely overwritten 5000 times
    - Fortunately, this is handled internally

Database Implementation

- Access to data in storage:
  - Blocks (HDD) and pages (SSD) have logical block address
  - Translation between logical block address and physical block address is done within the device

Database Implementation

- To access storage:
  - Allocate space for storage blocks in memory
  - Prefetch for reading
  - Have a write-back queue

Index Structures

Index Data Structures

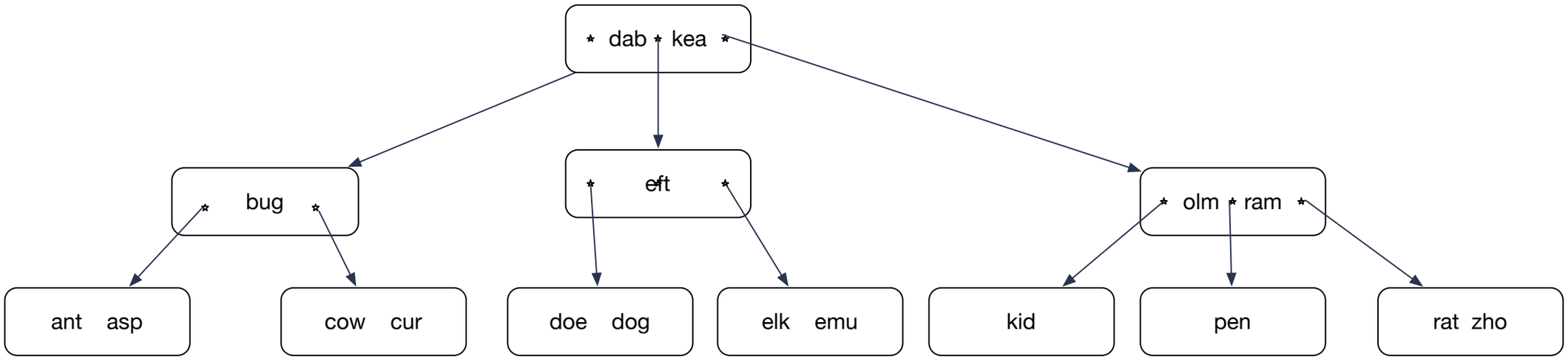
- Structures for one-dimensional data
  - Linear or Extensible Hashing
  - B - tree / B+ - tree for ordered indices
- Structures for multi-dimensional data

B-Trees

- B-trees: In memory data structure for CRUD and range queries
  - Balanced Tree
  - Each node can have between *d* and 2*d* keys with the exception of the root
  - Each node consists of a sequence of node pointer, key, node pointer, key, …, key, node pointer
  - Tree is ordered.
    - All keys in a child are between the keys adjacent to the node pointer

B-Trees

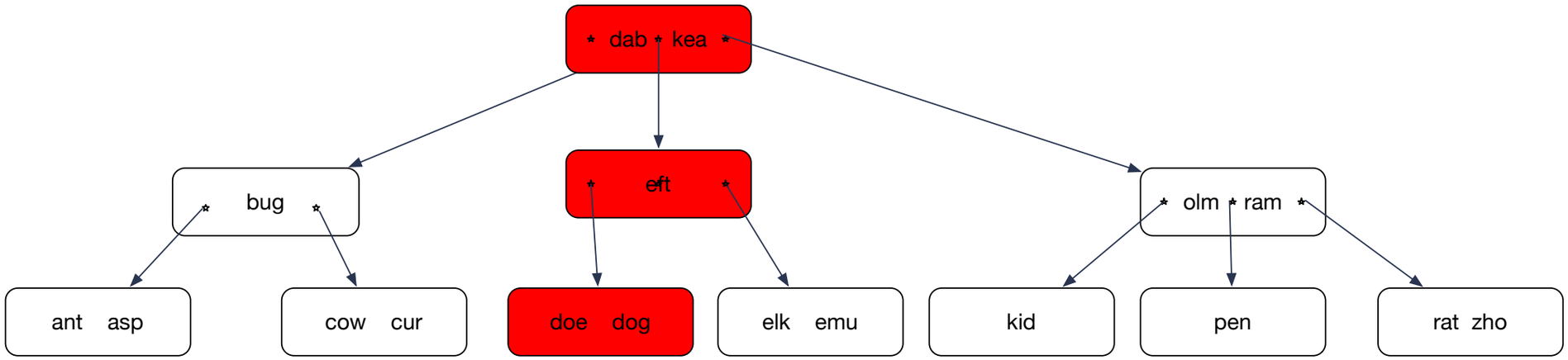
- Example: 2-3 tree: Each node has two or three children



B-Trees

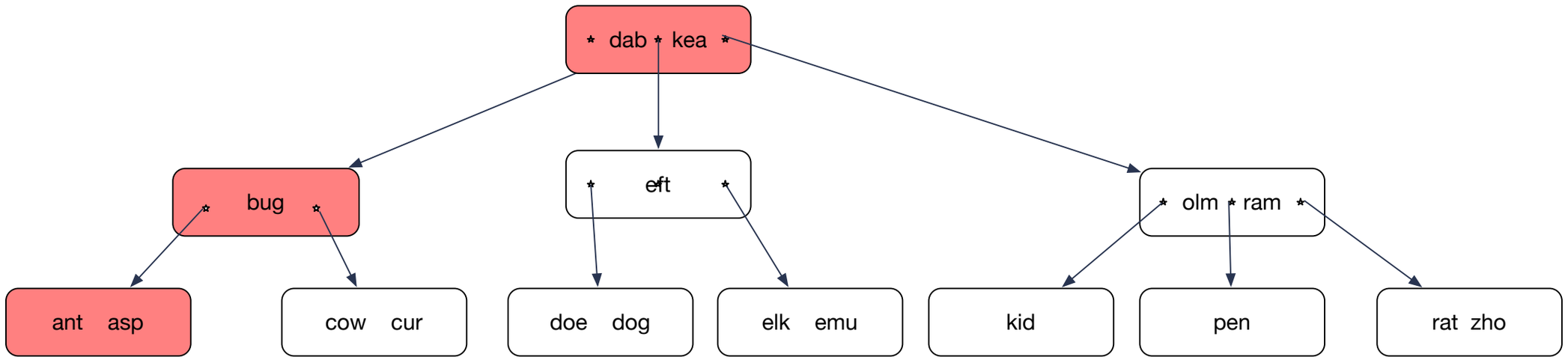
- Read dog:
  - Load root, determine location of dog in relation to the keys
  - Follow middle pointer
  - Follow pointer to the left
  - Find “dog”

B-Trees


dab kea
bug
eft
olm ram
ant asp
cow cur
doe dog
elk emu
kid
pen
rat zho

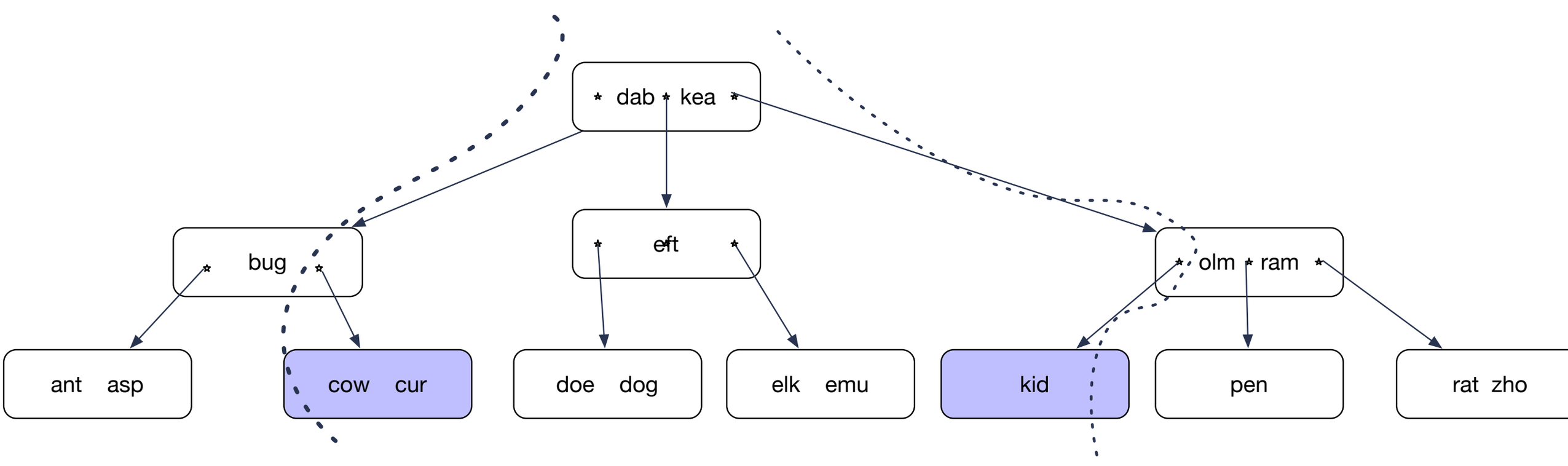
B-Trees

- Search for “auk” :



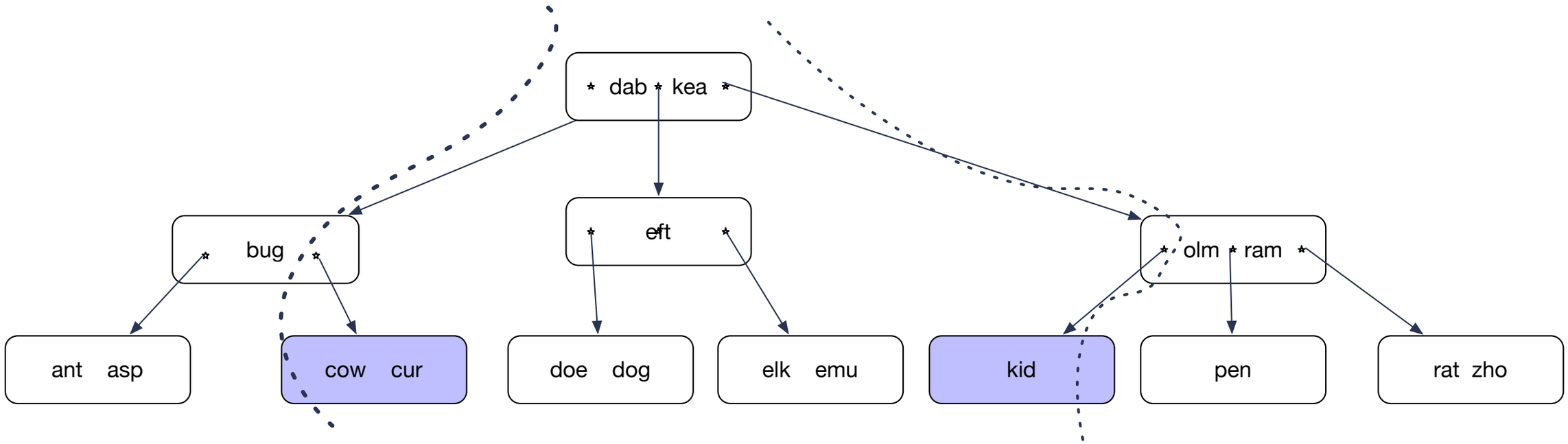
B-Trees

- Range Query c - l
  - Determine location of c and l



B-Trees

- Recursively enumerate all nodes between the lines starting with root



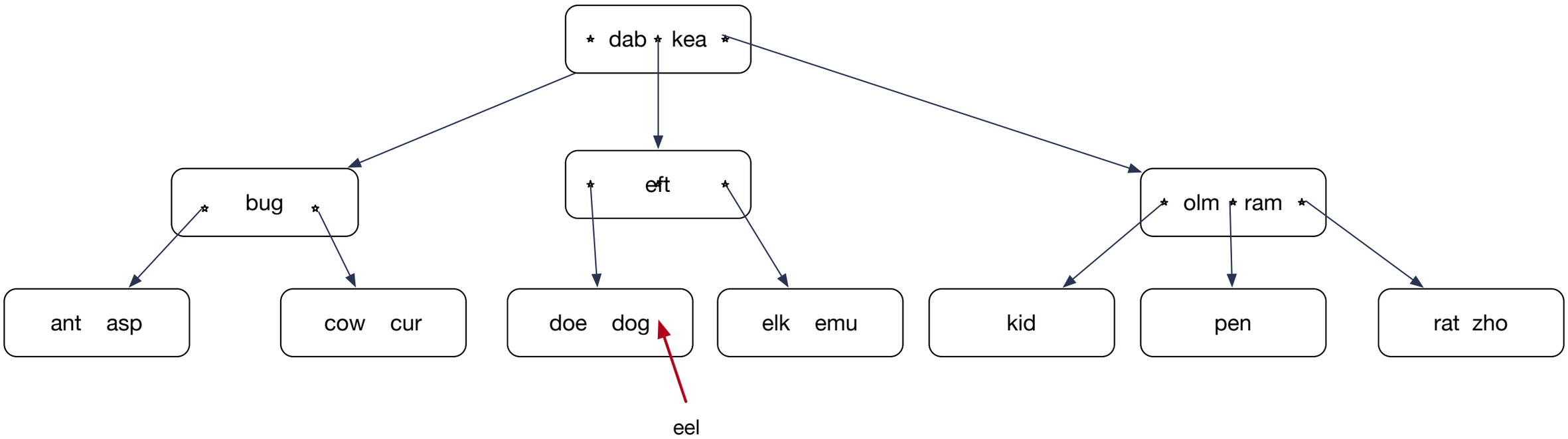
B-trees

- Capacity: With *l* levels, minimum of $1 + 2 + 2^2 + \ldots + 2^l$ nodes:
  - $1(2^{l+1} - 1)$ keys
- Maximum of $1 + 3 + 3^2 + \ldots + 3^l$ nodes
  - $\frac{2}{2}(3^{l+1} - 1)$ keys

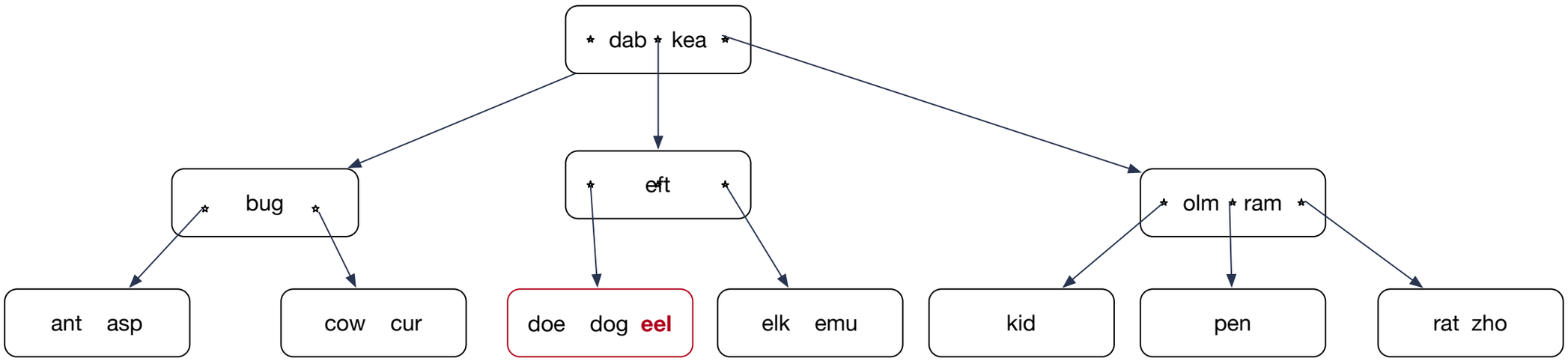
B-trees

- Inserts:
  - Determine where the key should be located in a leaf
  - Insert into leaf node
  - Leaf node can now have too many nodes
  - Take middle node and elevate it to the next higher level
  - Which can cause more “splits”

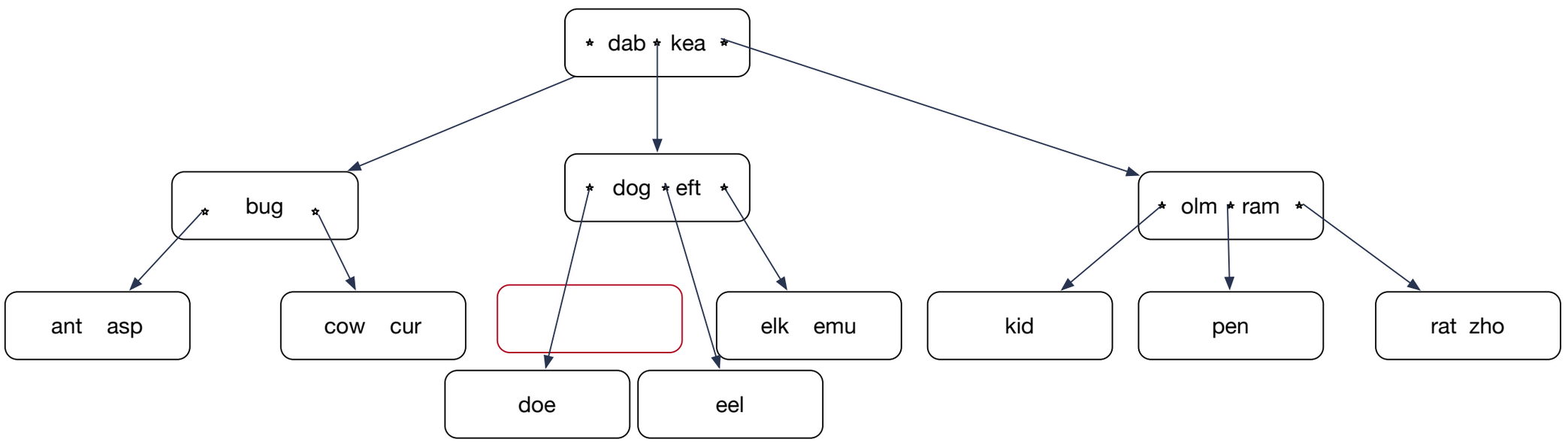
B-trees


dab kea
bug
eft
olm ram
ant asp
cow cur
doe dog
elk emu
kid
pen
rat zho
eel

B-trees


dab kea
bug
eft
olm ram
ant asp
cow cur
doe dog eel
elk emu
kid
pen
rat zho

B-trees


dab kea
bug
dog eft
olm ram
ant asp
cow cur
elk emu
kid
pen
rat zho
doe
eel

B-trees

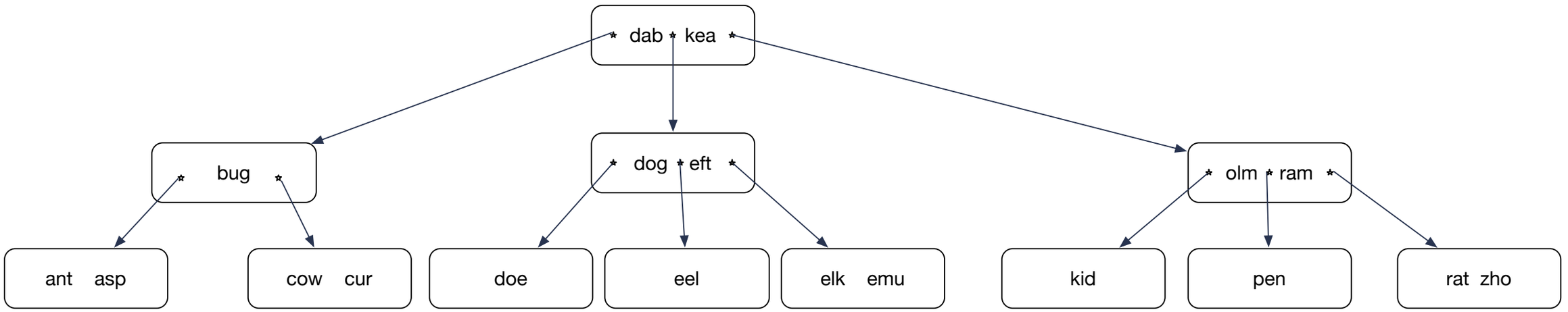
- Insert: Lock all nodes from root on down so that only one process can operate on the nodes
- Tree only grows a new level by splitting the root

B-Trees

- Using only splits leads to skinny trees
  - Better to make use of potential room in adjacent nodes
  - Insert “ewe”.
    - Node elk-emu only has one true neighbor.
      - Node kid does not count, it is a cousin, not a sibling

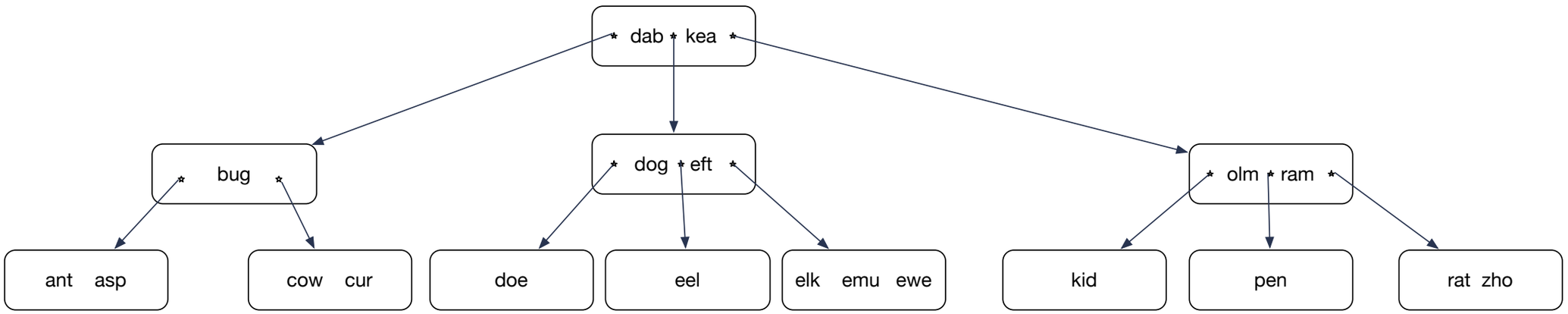
B-tree

- Insert ewe into



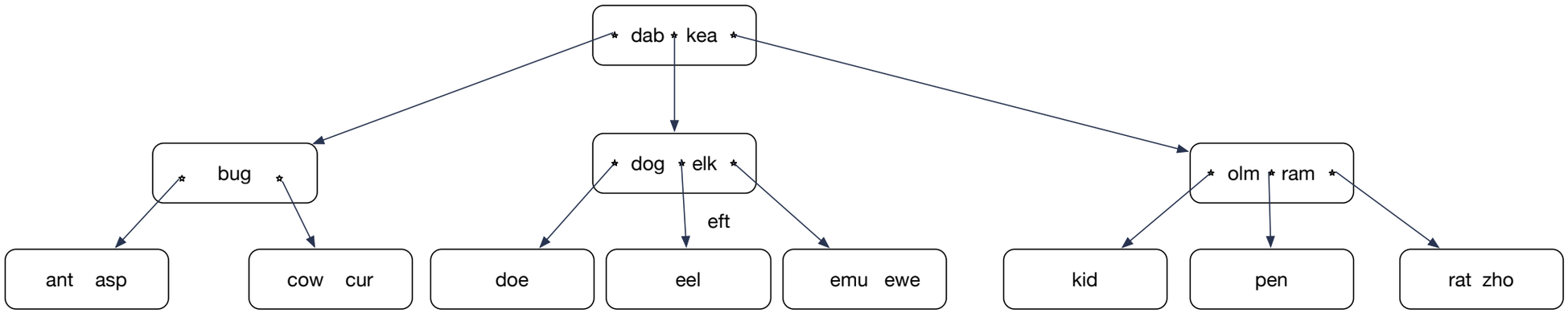
B-tree

- Insert ewe



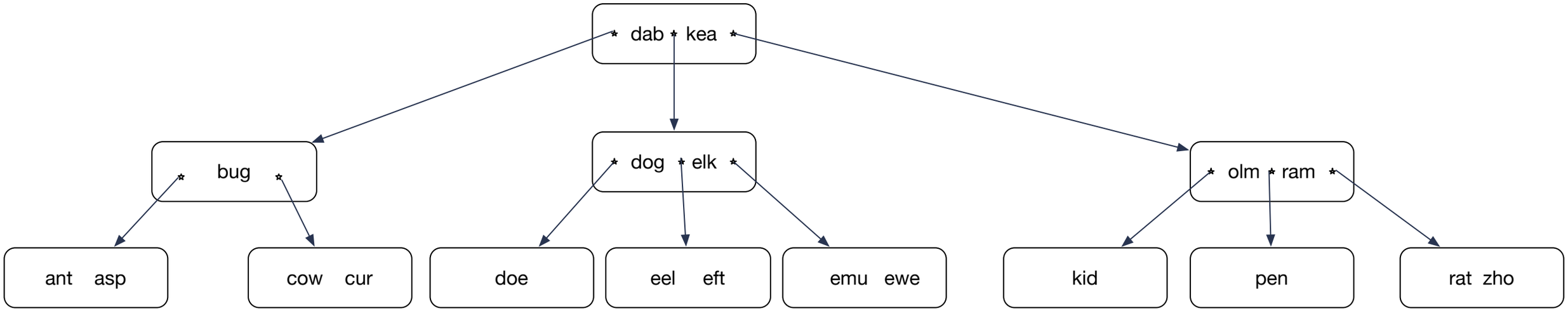
B-tree

- Promote elk. elk is guaranteed to come right after eft.
- Demote eft



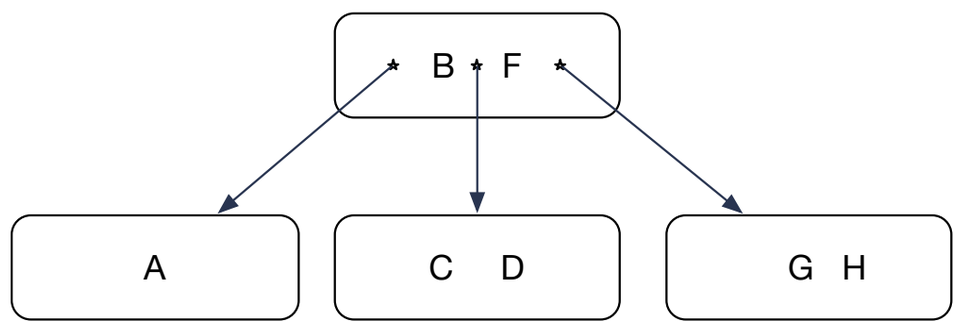
B-tree

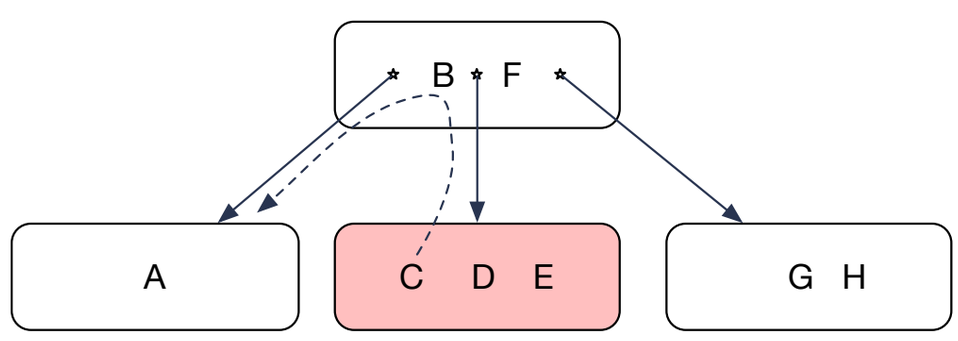
- Insert eft into the leaf node

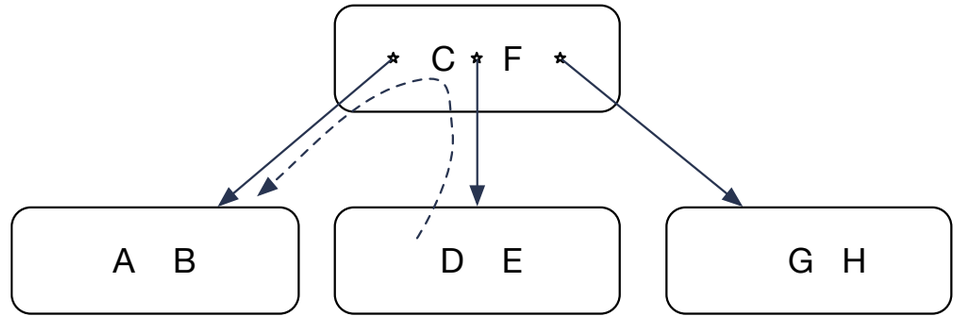


B-tree

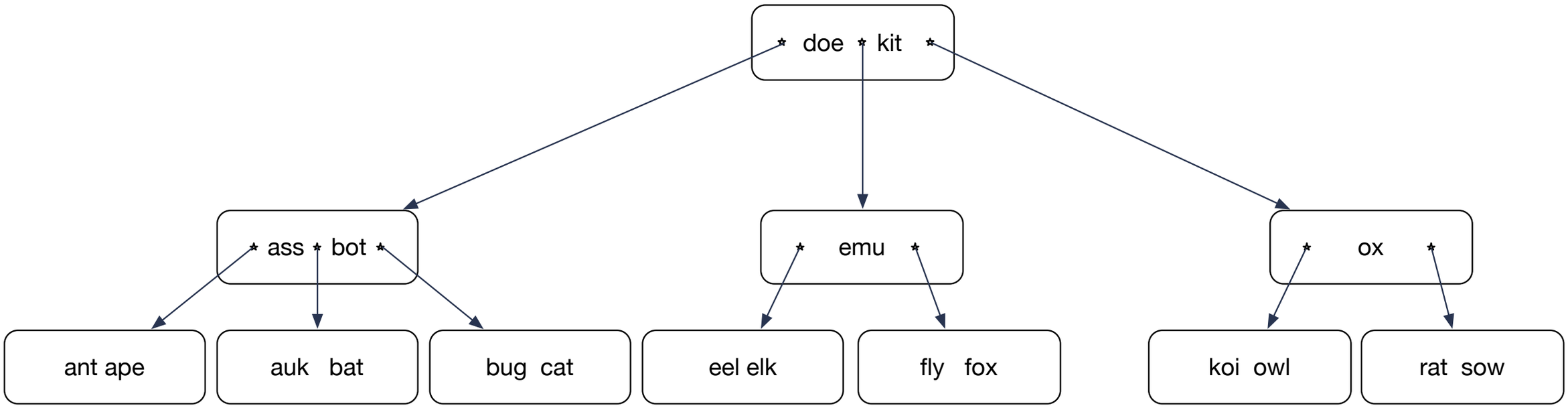
- Left rotate
  - Overflowing node has a sibling to the left with space
  - Move left-most key up
  - Lower left-most key





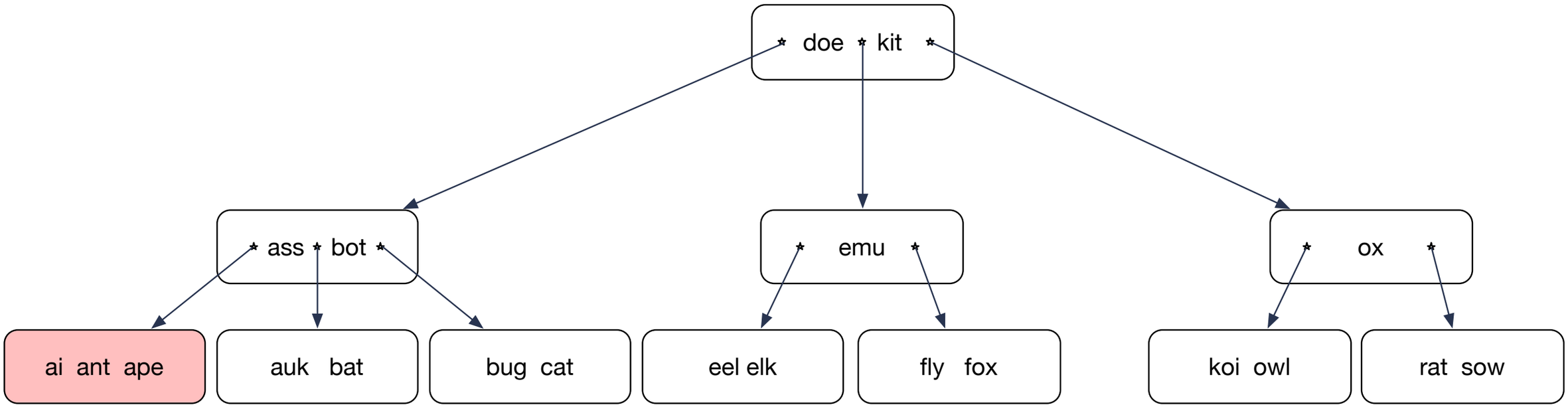


B-tree



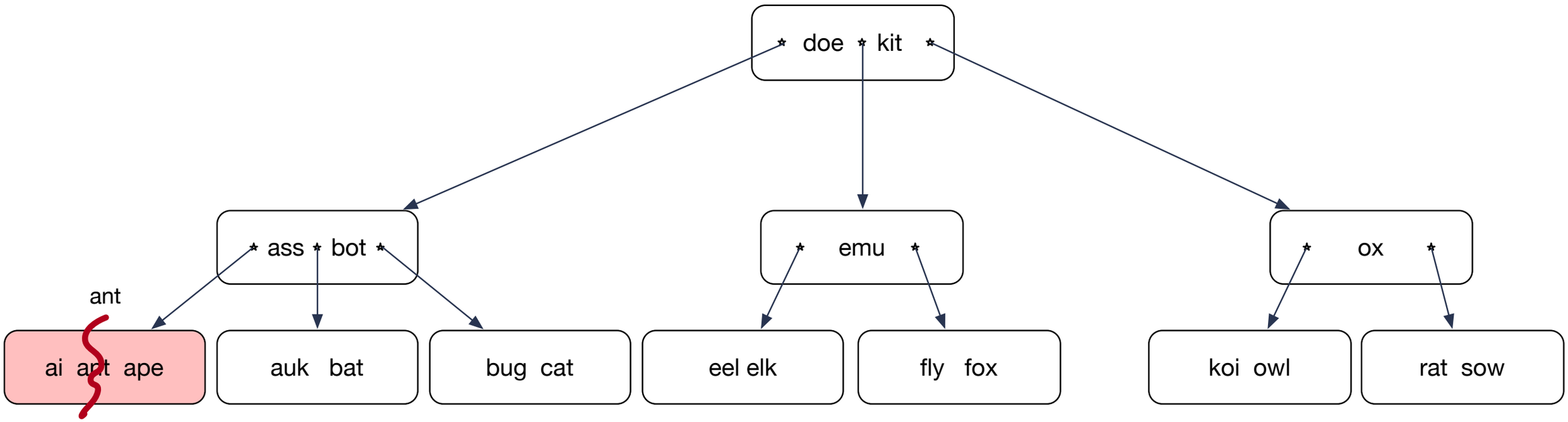
Now insert “ai”

B-tree



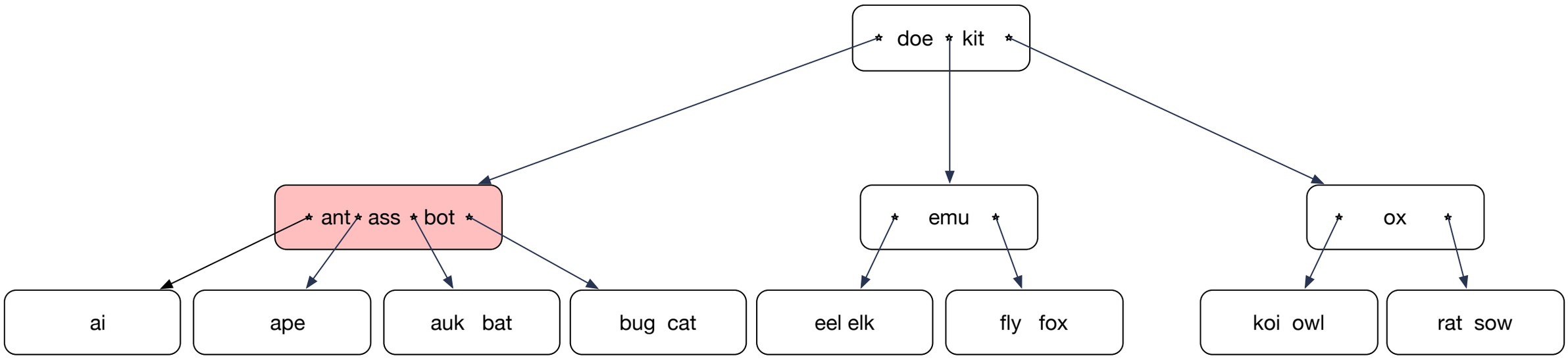
Insert creates an overflowing node
Only one neighboring sibling, but that one is full
Split!

B-tree



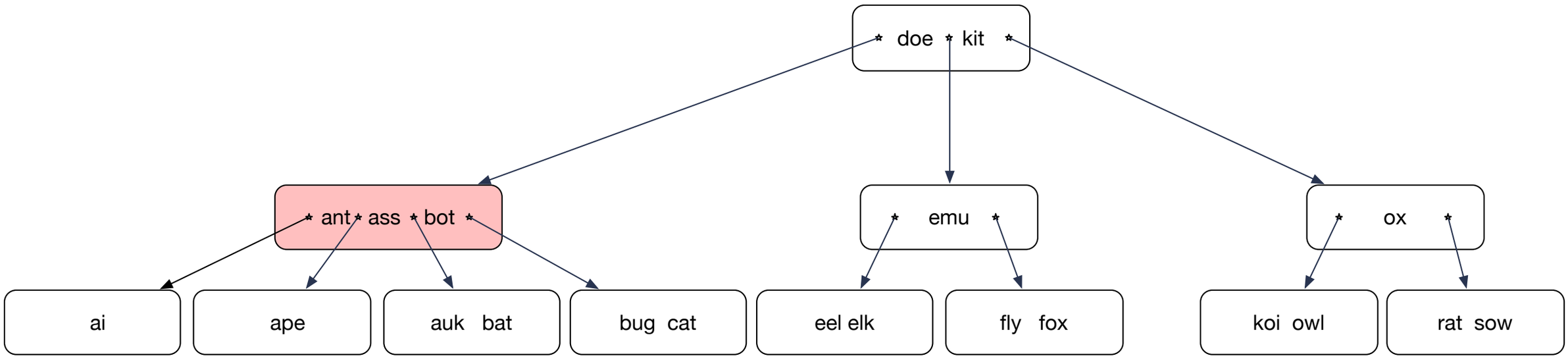
Middle key moves up

B-tree



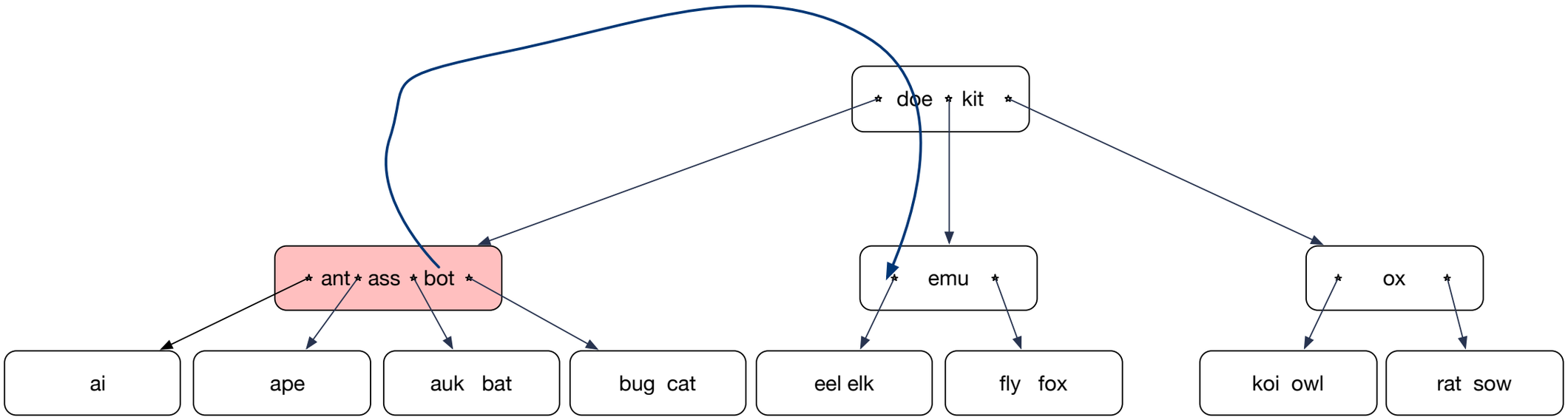
Unfortunately, this gives another overflow
But this node has a right sibling not at full capacity

B-tree



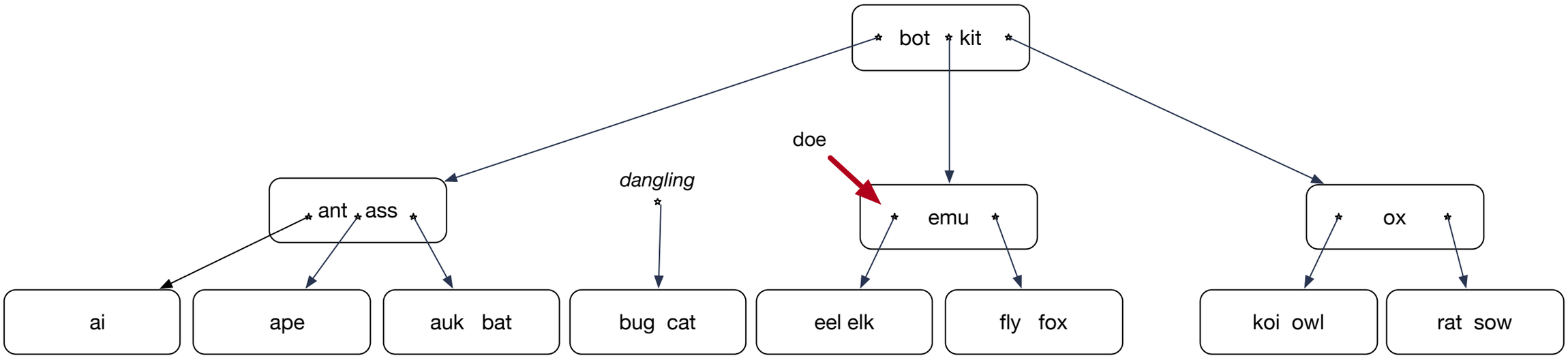
Right rotate:
Move “bot” up
Move “doe” down
Reattach nodes

B-tree



Move “bot” up
Move “doe” down
Reattach the dangling node

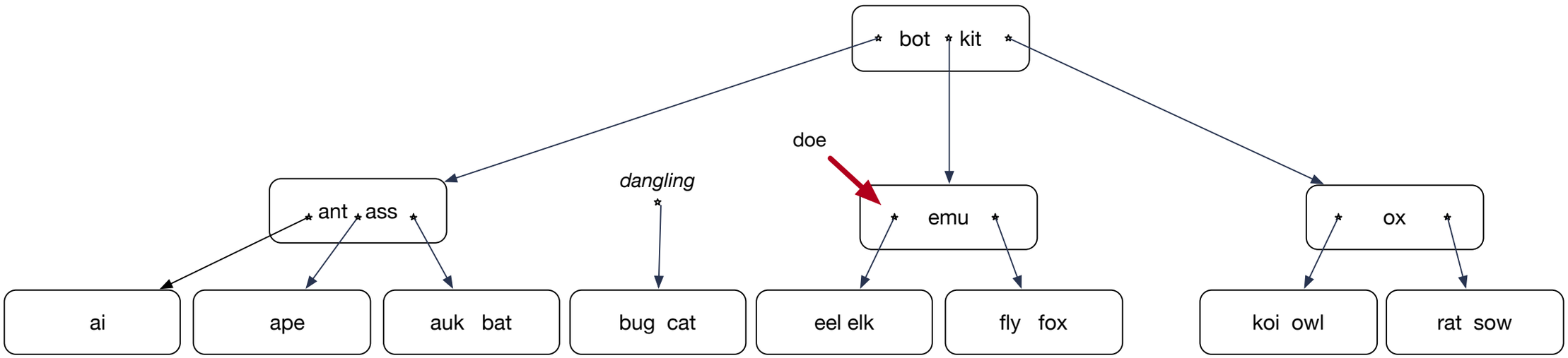
B-tree



“bot” had moved up and replaced doe

The “emu” node needs to receive one key and one pointer

B-tree


bot kit
doe
dangling
ant ass
emu
ox
ai
ape
auk bat
bug cat
eel elk
fly fox
koi owl
rat sow

B-tree

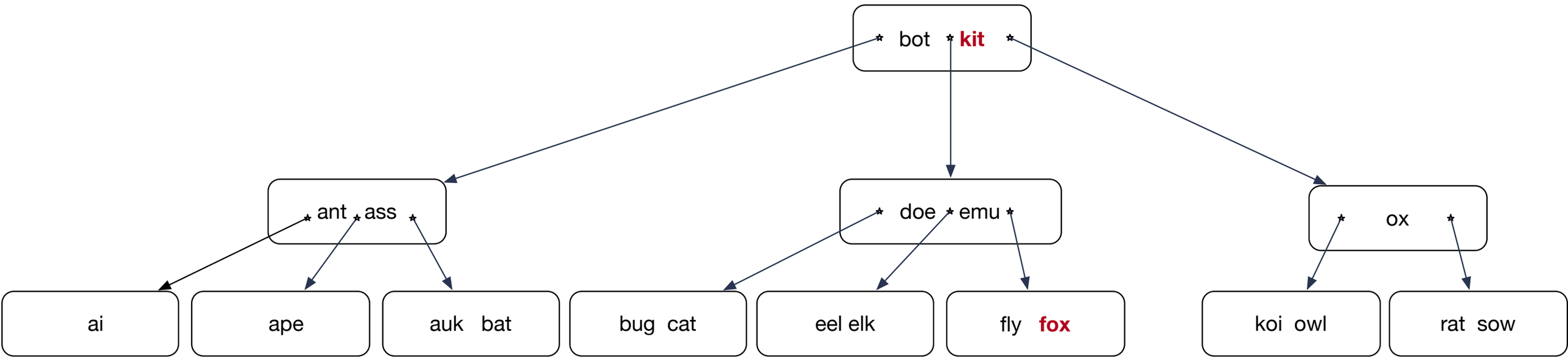
- Deletes
  - Usually restructuring not done because there is no need
  - Underflowing nodes will fill up with new inserts

B-tree

- Implementing deletion anyway:
  - Can only remove keys from leaves
  - If a delete causes an underflow, try a rotate into the underflowing node
  - If this is not possible, then merge with a sibling
    - A merge is the opposite of a split
  - This can create an underflow in the parent node
    - Again, first try rotate, then do a merge

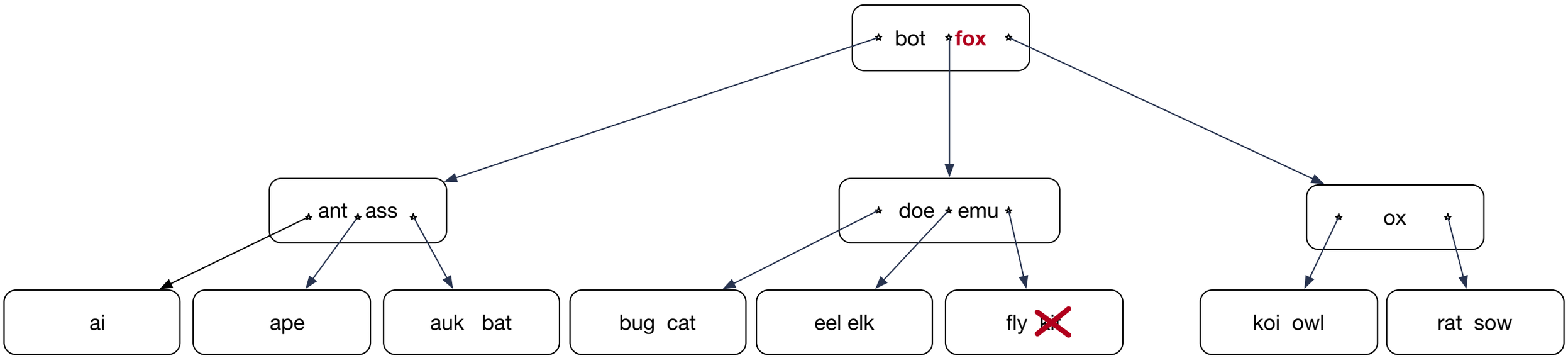
B-tree

Delete “kit”



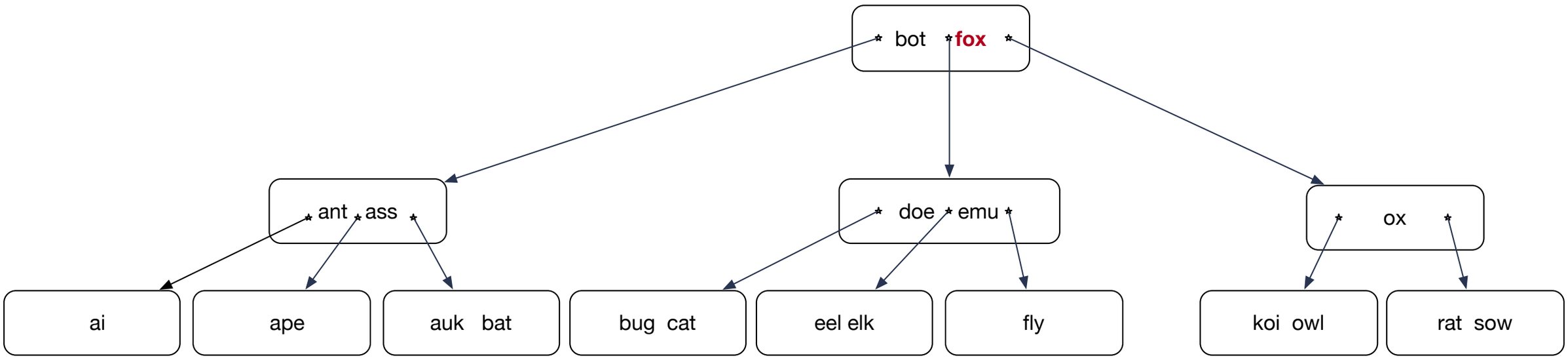
Delete “kit”
“kit” is in an interior node.
Exchange it with the key in the leave immediately before
“fox”

B-tree



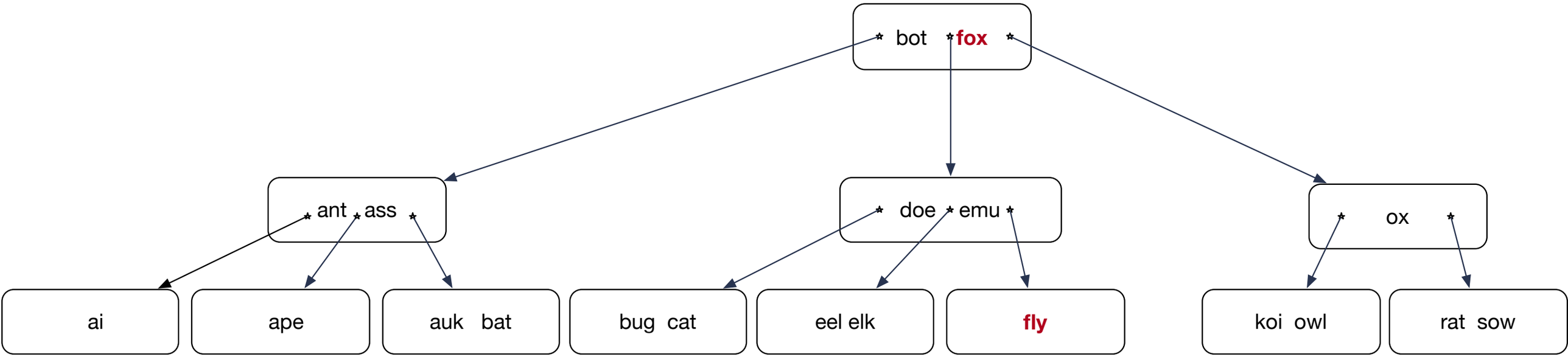
After interchanging “fox” and “kit”, can delete “kit”

B-tree



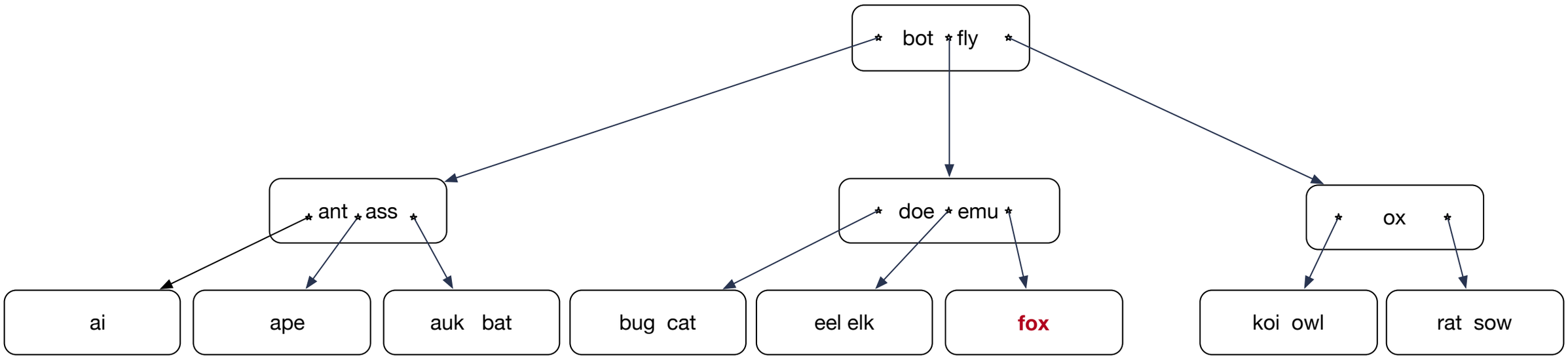
Now delete “fox”

B-tree



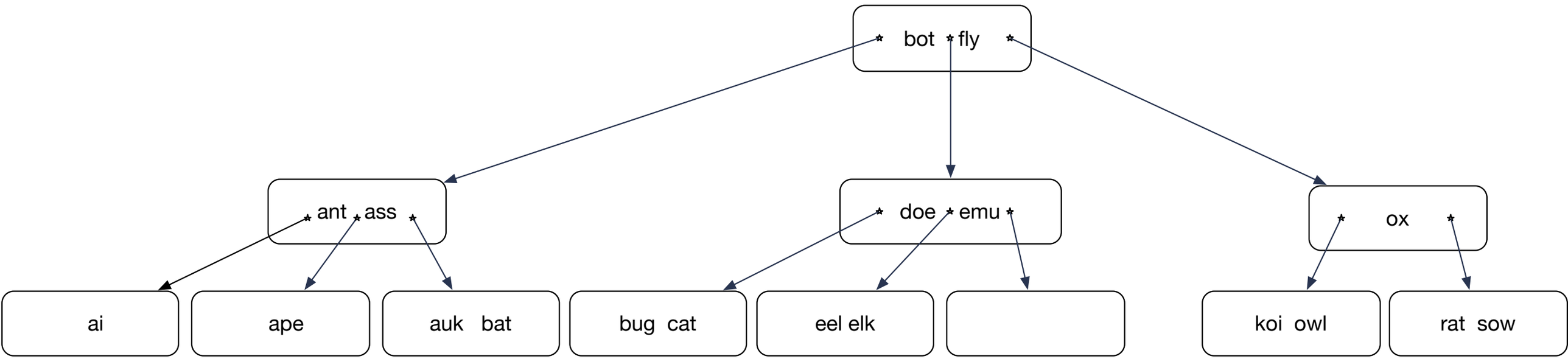
Step 1: Find the key. If it is not in a leaf
Step 2: Determine the key just before it, necessarily in a leaf
Step 3: Interchange the two keys

B-tree



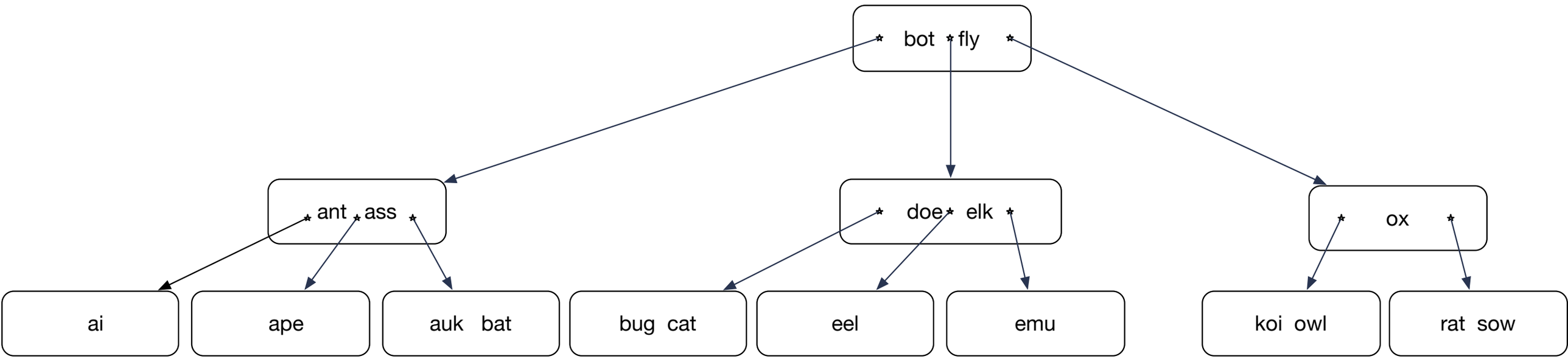
Step 4: Remove the key now from a leaf

B-tree



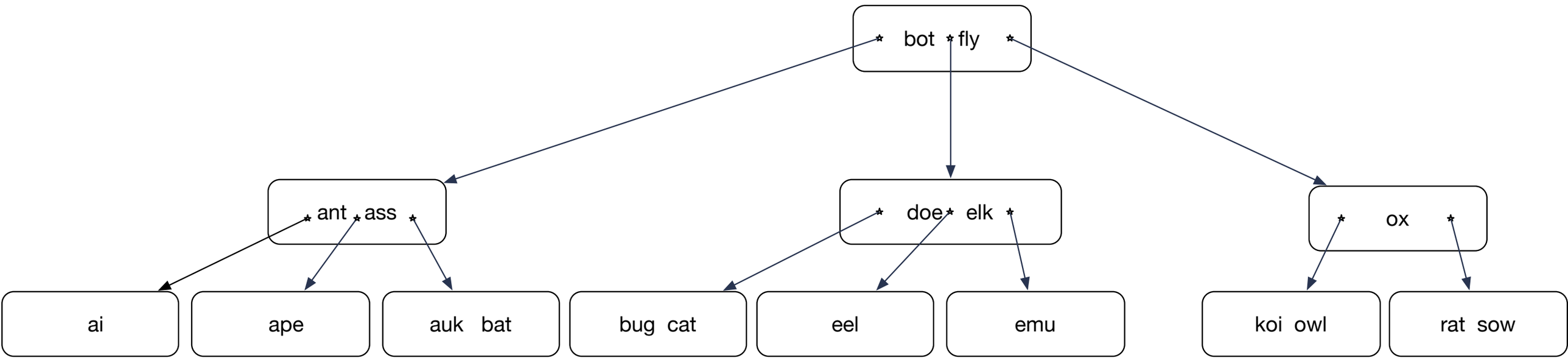
This causes an underflow
Remedy the underflow by right rotating from the sibling

B-tree



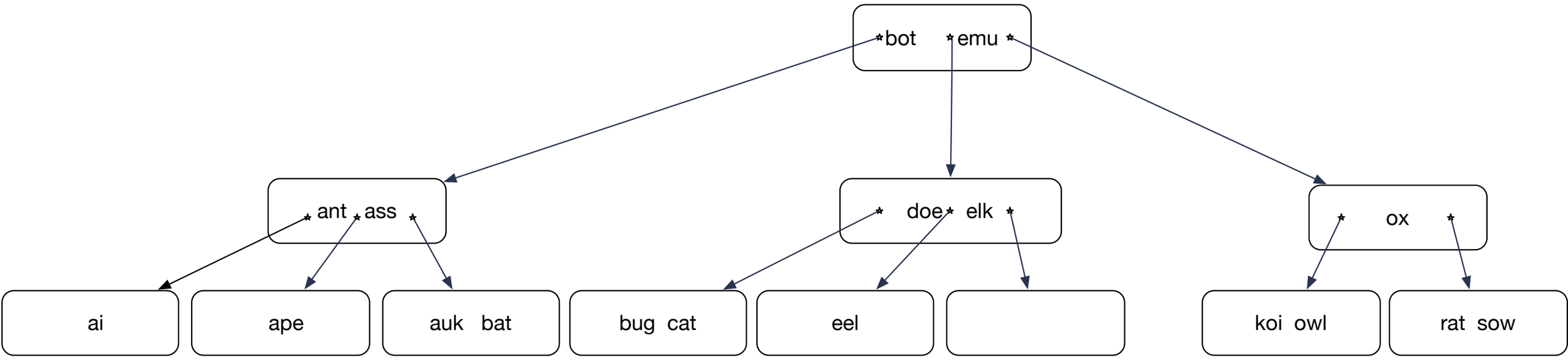
Everything is now in order

B-tree



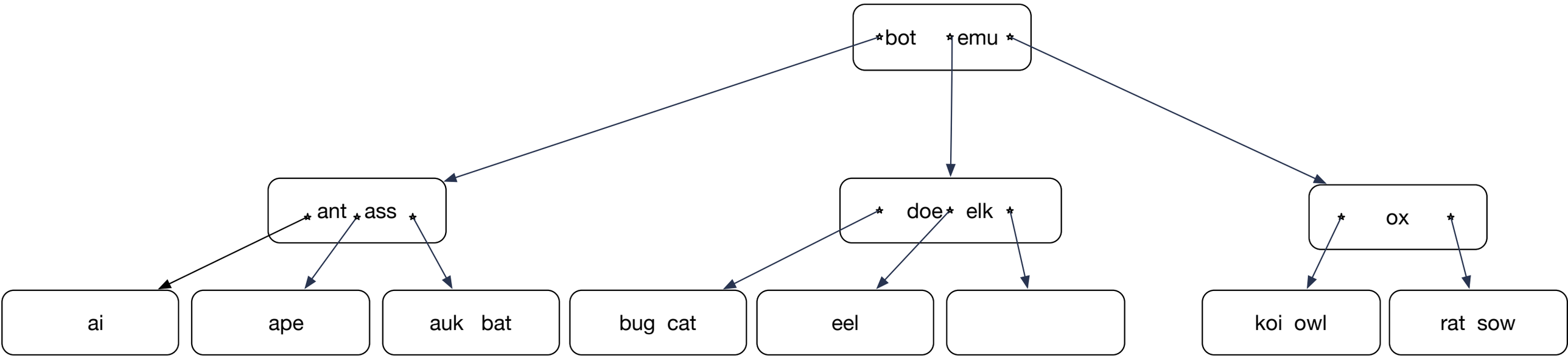
Now delete fly

B-tree



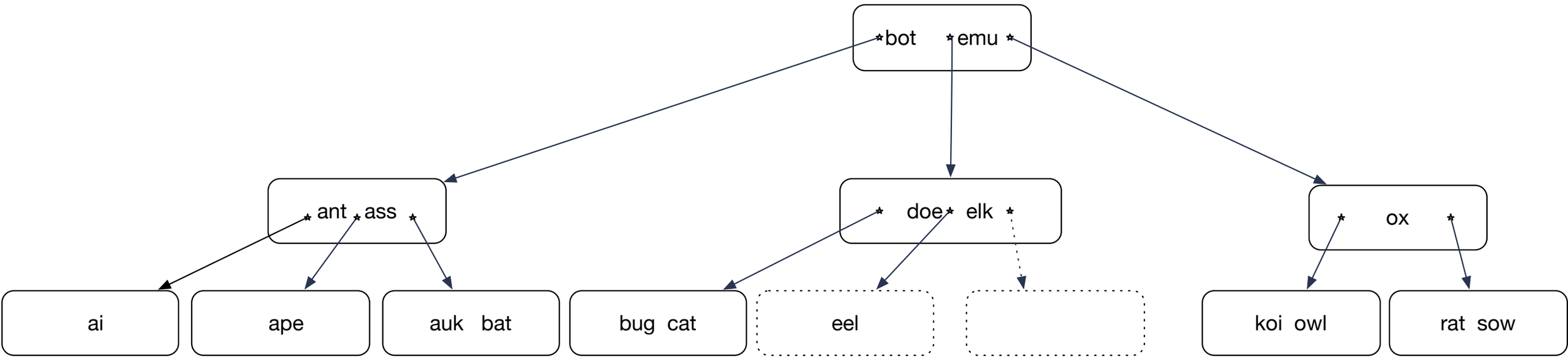
Switch “fly” with “emu”
remove “fly” from the leaf
Again: underflow

B-tree



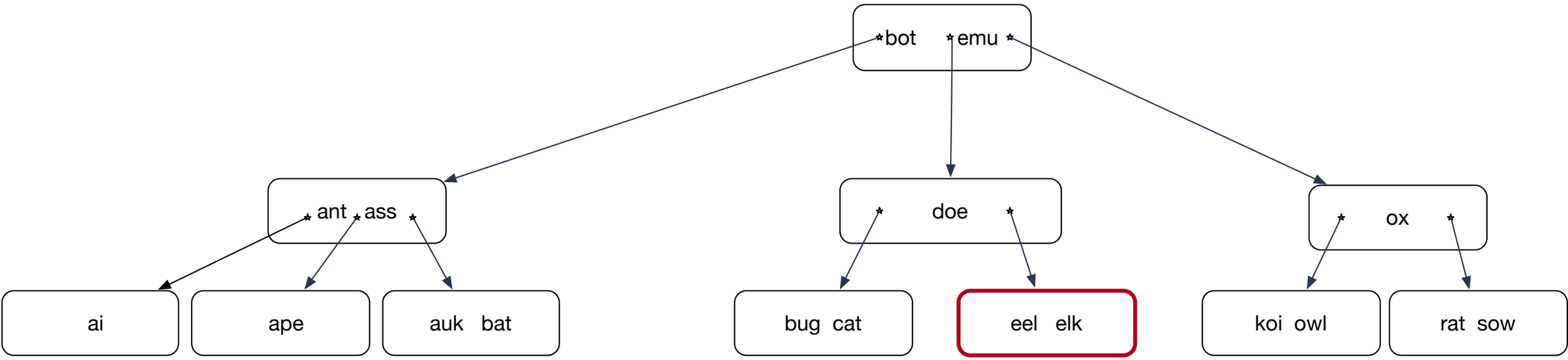
Cannot left-rotate: There is no left sibling
Cannot right-rotate: The right sibling has only one key
Need to merge: Combine the two nodes by bringing down “elk”

B-tree

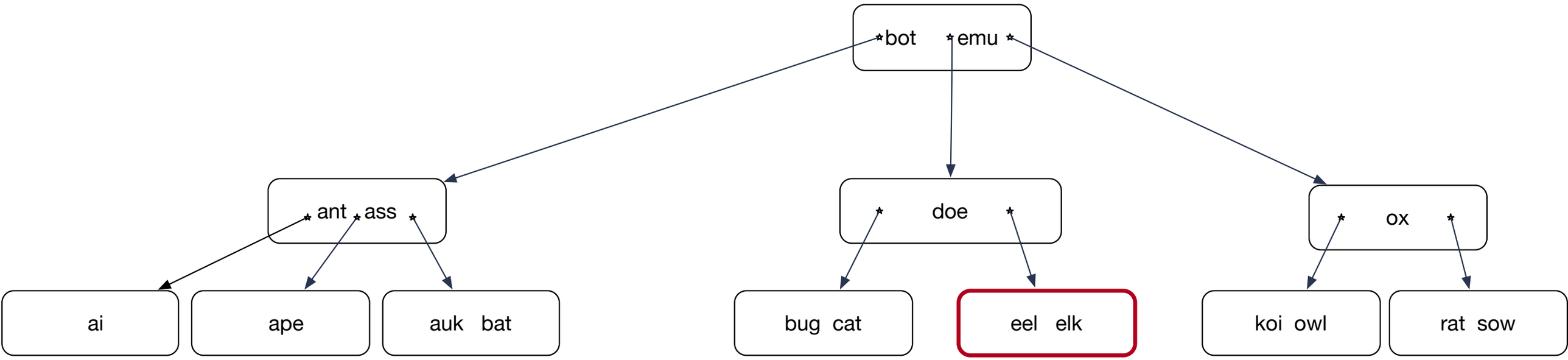


We can merge the two nodes because the number of keys combined is less than 2 *k*

B-tree

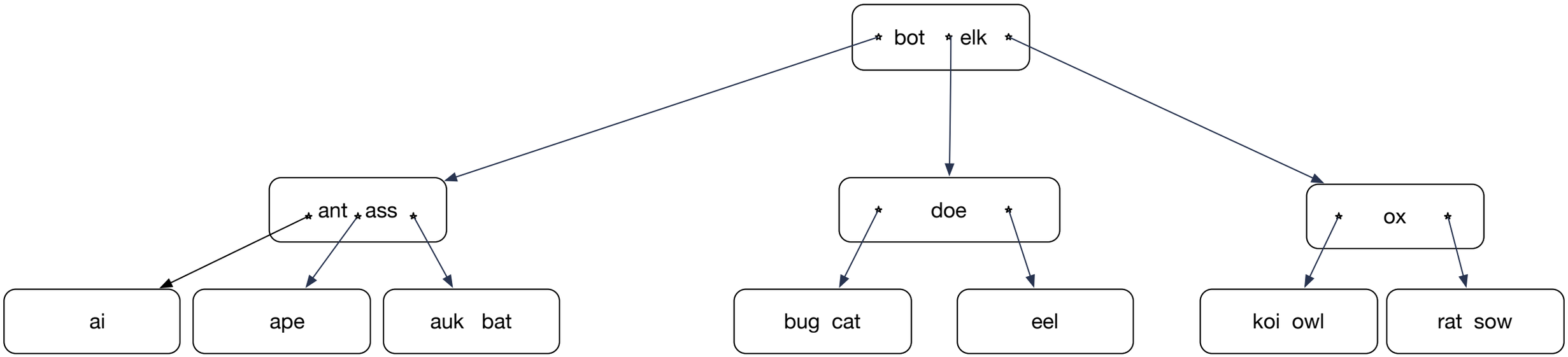

bot
emu
ant
ass
doe
ox
ai
ape
auk
bat
bug
cat
eel
elk
koi
owl
rat
sow

B-tree



Delete “emu”

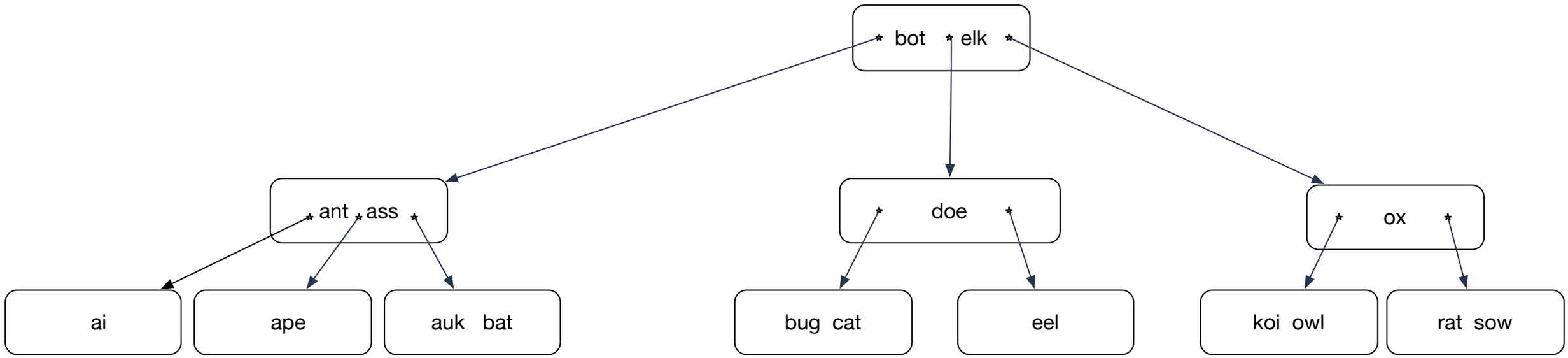
B-tree



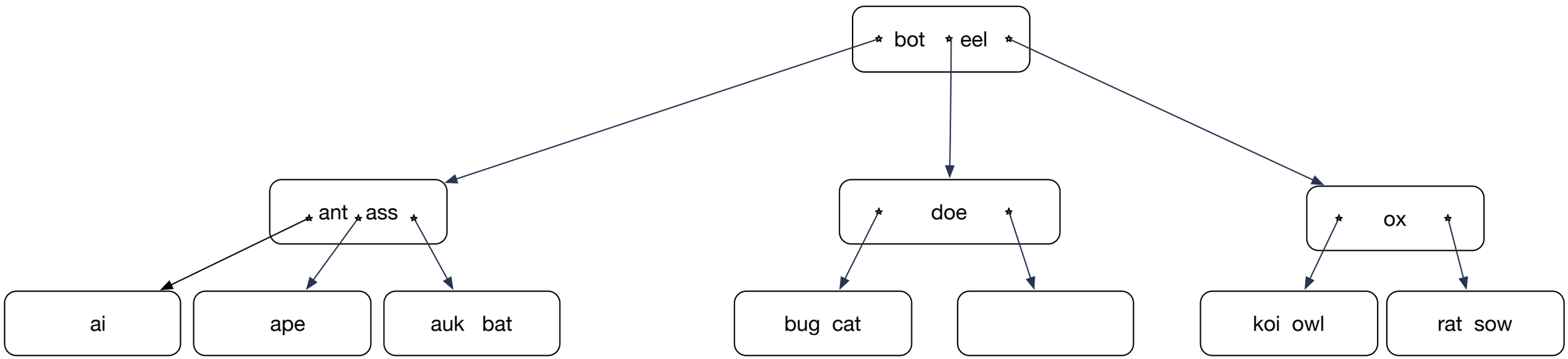
Switch predecessor, then delete from node

B-tree

Now delete “elk”

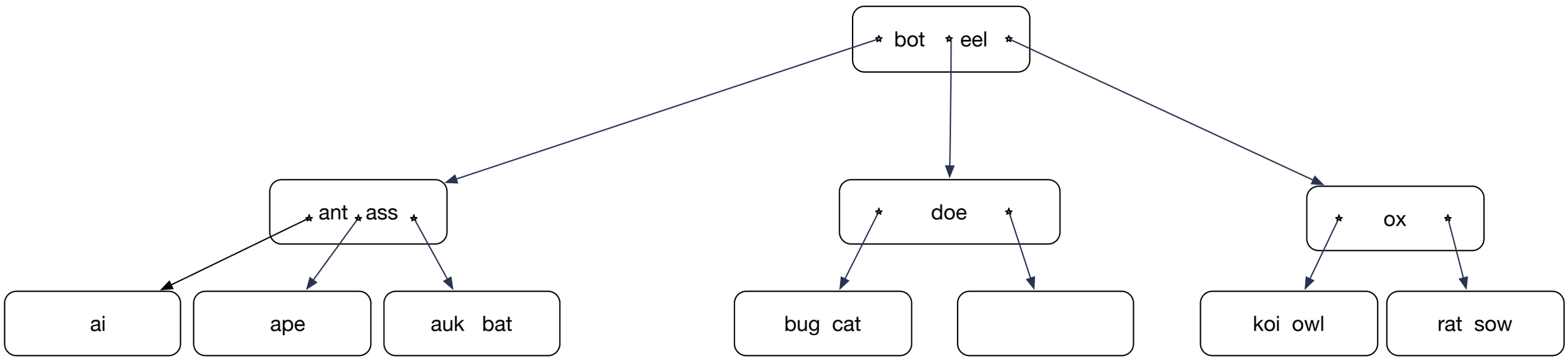


B-tree



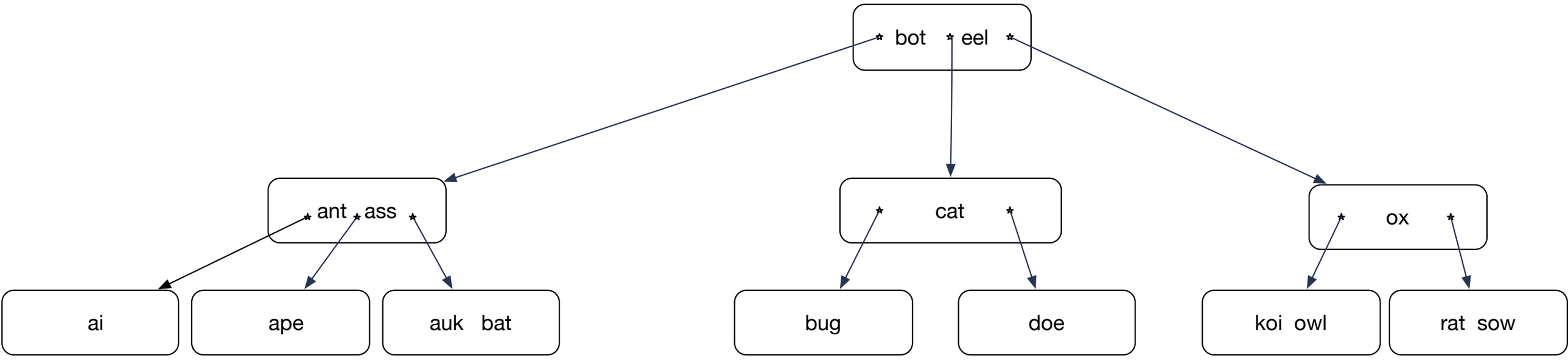
Results in an underflow

B-tree



Results in an underflow
But can rotate a key into the underflowing node

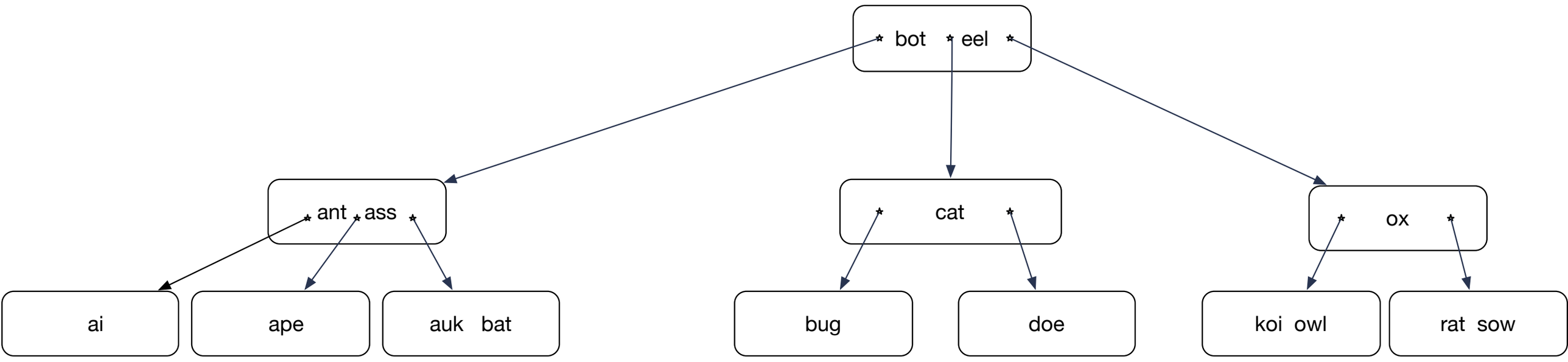
B-tree



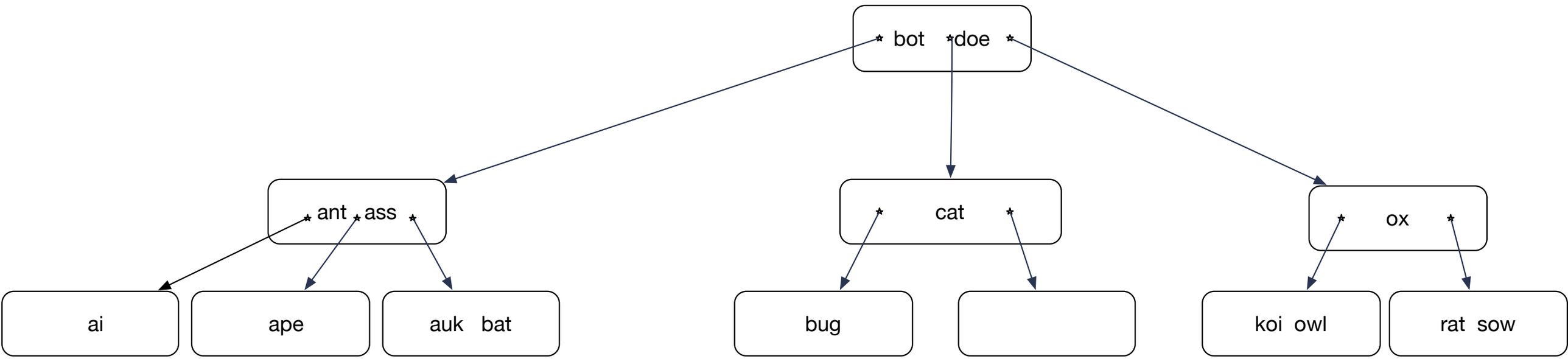
Result after left-rotation

B-tree

“Now delete “eel”

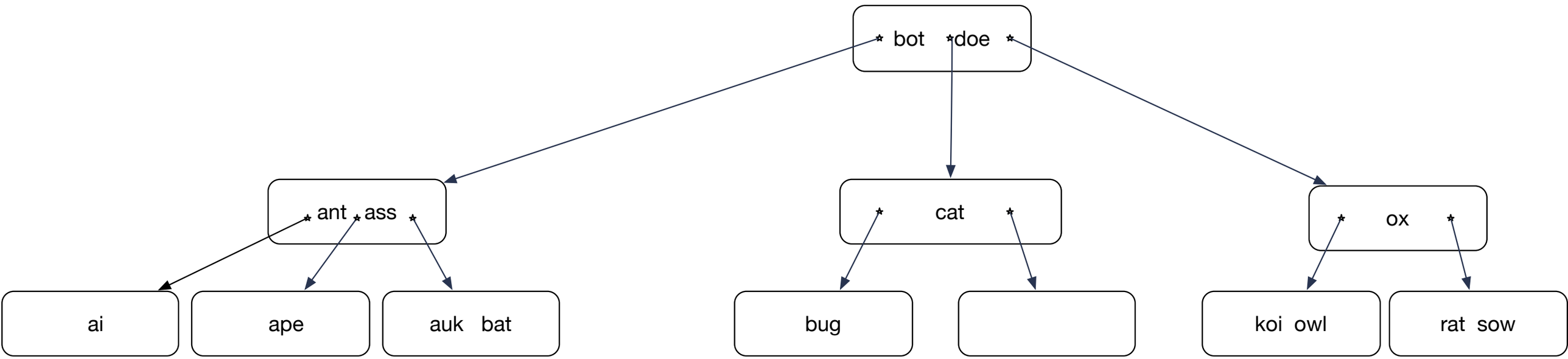


B-tree



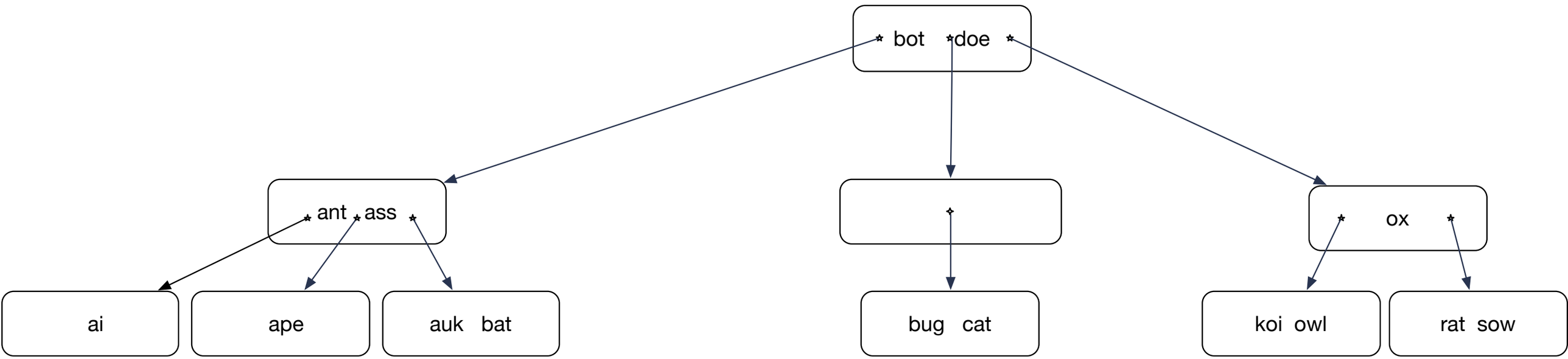
Interchange “eel” with its predecessor
Delete “eel” from leaf:
Underflow

B-tree



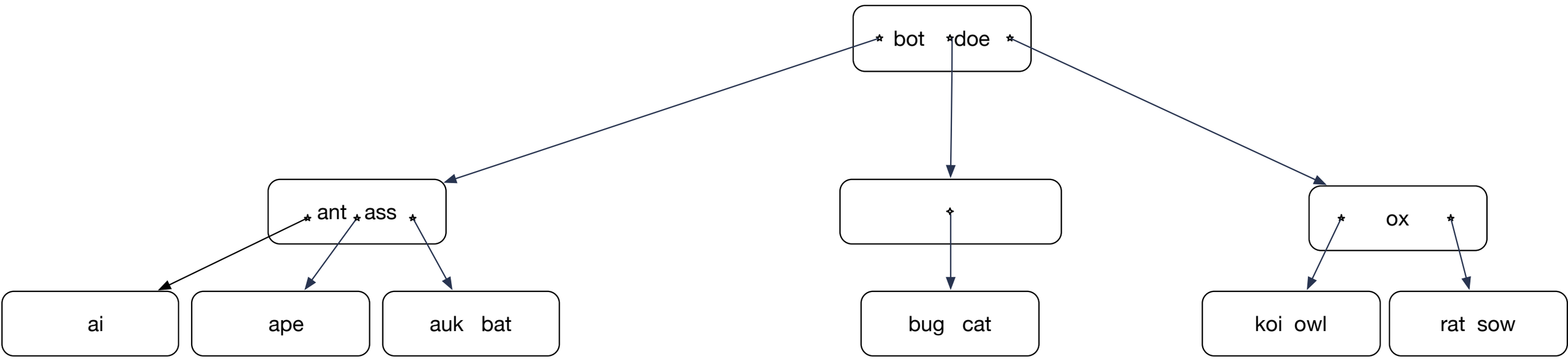
Need to merge

B-tree



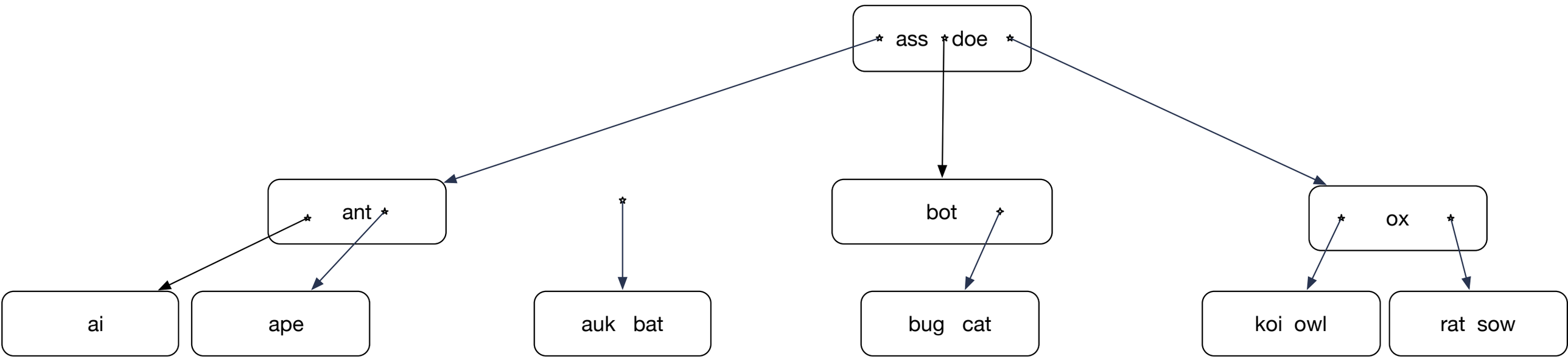
Merge results in another underflow
Use right rotate
(though merge with right sibling
is possible)

B-tree



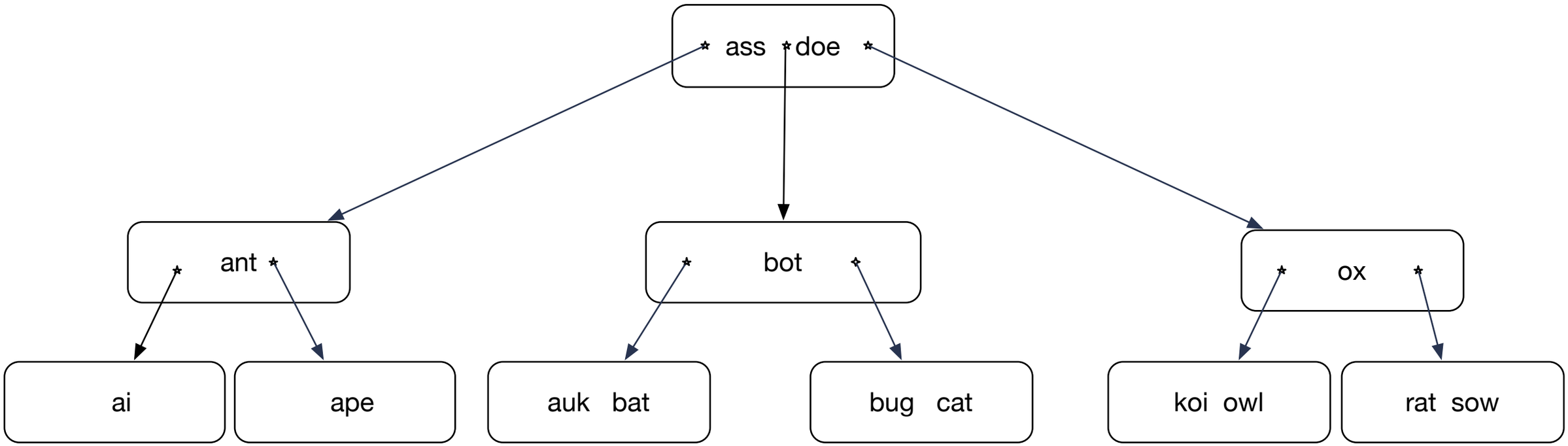
“ass” goes up, “bot” goes down
One node is reattached

B-tree



Reattach node

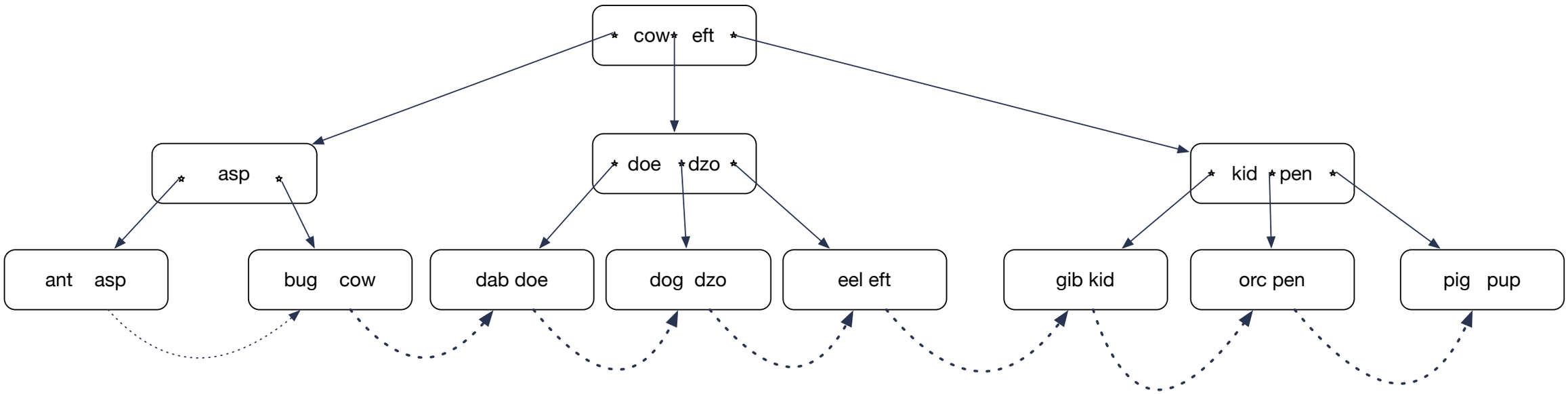
B-tree



In real life

- Use B+ tree for better access with block storage
  - Data pointers / data are only in the leaf nodes
  - Interior nodes only have keys as signals
  - Link leaf nodes for faster range queries.

B+ Tree


cow eft
asp
doe dzo
kid pen
ant asp
bug cow
dab doe
dog dzo
eel eft
gib kid
orc pen
pig pup

B+ Tree

- Real life B+ trees:
  - Interior nodes have many more keys (e.g. 100)
  - Leaf nodes have as much data as they can keep
  - Need few levels:
    - Fast lookup

Hashing

- Basic idea: Place records in a bucket defined by the hash value of the key
  - Hash value: Pseudo-random number
- Extensible hashing:
  - Number of buckets increases with number of records through bucket splits
    - Extendible hashing (Fegan): Update a data structure in order to find out which buckets are split
    - Linear Hashing (Litwin): Buckets split in order

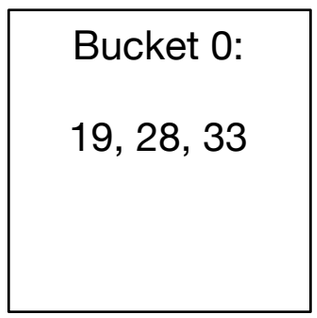
Linear Hashing

Linear Hashing

- Extensible Hashing:
  - Uses a lot of metadata to reflect history of splitting
    - But only splits buckets when they are needed
  - Linear Hashing
    - Splits buckets in a predefined order
    - Minimal meta-data
    - Sounds like a horrible idea, but …

Linear Hashing

- Assume a hash function that creates a large string of bits
  - We start using these bits as we extend the address space
- Start out with a single bucket, Bucket 0
- All items are located in Bucket 0



Items with keys 19, 28, 33

Linear Hashing

- Eventually, this bucket will overflow
  - E.g. if the load factor is more than 2
  - Bucket 0 splits
    - All items in Bucket 0 are rehashed:
      - Use the last bit in order to determine whether the item goes into Bucket 0 or Bucket 1
      - Address is $h_1(c) = c \pmod 2$

Linear Hashing

- After the split, the hash table has two buckets:

Bucket 0:	Bucket1:
28	19, 33

- After more insertions, the load factor again exceeds 2

Bucket 0:	Bucket1:
28, 40	11, 19, 33

Linear Hashing

- Again, the bucket splits.
  - But it has to be Bucket 0

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33	

- For the rehashing, we now use two bits, i.e.

$$h_2(c) = c \pmod 4$$

  - But only for those items in Bucket 0

Linear Hashing

- After some more insertions, Bucket 1 will split

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33, 35	6

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40	33	6	11, 19, 35

Linear Hashing

- The state of a linear hash table is described by the number N of buckets
  - The level l is the number of bits that are being used to calculate the hash
  - The split pointer s points to the next bucket to be split
  - The relationship is

$$N = 2^l + s$$

    - This is unique, since always $s < 2^l$

Linear Hashing

- Addressing function
  - The address of an item with key c is calculated by

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

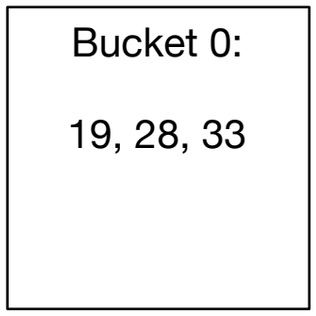
  - This reflects the fact that we use more bits for buckets that are already split

Linear Hashtable Evolution

$$N = 1 = 2^0 + 0$$

```
Number of buckets: 1
Split pointer: 0
Level: 0
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```



Linear Hashtable Evolution

$$N = 2 = 2^1 + 0$$

```
Number of buckets: 2
Split pointer: 0
Level: 1
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:
28	19, 33

```
Add items with hashes 40 and 11
This gives an overflow and we split Bucket 0
```

Linear Hashtable Evolution

$$N = 3 = 2^1 + 1$$

```
Number of buckets: 3
Split pointer: 1
Level: 1
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:
28, 40	11, 19, 33

```
split Bucket 0
Create Bucket 2
Use new hash function on items in Bucket 0
```

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33	

```
No items were moved
```

Linear Hashtable Evolution

$N = 3 = 2^1 + 1$

```
Number of buckets: 3
Split pointer: 1
Level: 1
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33	

Add items 6, 35

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33, 35	6

Because of overflow, we split Bucket 1

Linear Hashtable Evolution

$$N = 4 = 2^2 + 0$$

```
Number of buckets: 4
Split pointer: 0
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33, 35	6

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40	33	6	11, 19, 35

Linear Hashtable Evolution

$N = 4 = 2^2 + 0$

```
Number of buckets: 4
Split pointer: 0
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40	33	6	11, 19, 35

Now add keys 8, 49

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40, 8	33, 49	6	11, 19, 35

Creates an overflow!
Need to split!

Linear Hashtable Evolution

$$N = 5 = 2^2 + 1$$

```
Number of buckets: 1
Split pointer: 1
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40, 8	33, 49	6	11, 19, 35

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	33, 49	6	11, 19, 35	28

```
Create Bucket 4.
Rehash Bucket 0.
```

Linear Hashtable Evolution

$$N = 5 = 2^2 + 1$$

```
Number of buckets: 5
Split pointer: 1
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	33, 49	6	11, 19, 35	28

```
Add keys 9, 42
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	9, 33, 49	6, 42	11, 19, 35	28

```
Creates an overflow!
Need to split!
```

Linear Hashtable Evolution

$$N = 6 = 2^2 + 2$$

```
Number of buckets: 1
Split pointer: 2
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	9, 33, 49	6, 42	11, 19, 35	28

Split

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 42	11, 19, 35	28	

No item actually moved, but average load factor is now again under 2.

Linear Hashtable Evolution

$$N = 6 = 2^2 + 2$$

```
Number of buckets: 6
Split pointer: 2
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 42	11, 19, 35	28	

```
add 5,10
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 10, 42	11, 19, 35	28	5

Linear Hashtable Evolution

$$N = 7 = 2^2 + 3$$

```
Number of buckets: 7
Split pointer: 3
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 10, 42	11, 19, 35	28	5

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42	11, 19, 35	28	5	6

Linear Hashtable Evolution

$$N = 7 = 2^2 + 3$$

```
Number of buckets: 7
Split pointer: 3
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42	11, 19, 35	28	5	6

```
add 92, 74
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6

Linear Hashtable Evolution

$$N = 8 = 2^3 + 0$$

```
Number of buckets: 8
Split pointer: 0
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6	

Linear Hashtable Evolution

$$N = 8 = 2^3 + 0$$

```
Number of buckets: 8
Split pointer: 0
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

```
add 13, 54
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6	

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:
	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54	

Linear Hashtable Evolution

$$N = 9 = 2^3 + 1$$

```
Number of buckets: 9
Split pointer: 1
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1: 9, 33, 49	Bucket 2: 10, 42, 74	Bucket 3: 11, 19, 35	Bucket 4: 28, 92	Bucket 5: 5, 13	Bucket 6: 6, 54	Bucket 7:

Bucket 0:	Bucket1: 9, 33, 49	Bucket 2: 10, 42, 74	Bucket 3: 11, 19, 35	Bucket 4: 28, 92	Bucket 5: 5, 13	Bucket 6: 6, 54	Bucket 7:	Bucket 8: 40, 8

Linear Hashtable Evolution

$N = 9 = 2^3 + 1$

```
Number of buckets: 9
Split pointer: 1
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:
	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54		40, 8

```
add 1, 81
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:
	1, 9, 33, 49, 81	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54		40, 8

Linear Hashtable Evolution

$$N = 10 = 2^3 + 2$$

```
Number of buckets: 10
Split pointer: 2
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:	Bucket 9:
	1, 33, 49, 81	10, 42, 74	11, 19, 35, 67, 99	28, 92	5, 13	6, 54	39	40, 8	9

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:	Bucket 9:	Bucket 10:
	1, 33, 49, 81		11, 19, 35, 67, 99	28, 92	5, 13	6, 54	39	40, 8	9	10, 42, 74

Linear Hashing

- Observations:
  - Buckets split in fixed order
    - 0, 0,1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, …, 15, 0, …
  - Address calculation is modulo 2^l, i.e. the *l* least significant bits
  - Buckets 0, 1, …, s-1 and 2**l*, 2**l*+1, … *N*-1 are already split, they have on average half the size of the buckets *s*, *s*+1, …, 2**l*.

Linear Hashing

- Observations:
  - An overflowing bucket is not necessarily split immediately
    - Sometimes, a split leaves all keys in the splitting bucket or moves them all to the new bucket
- On average, a bucket will have α items in them

Grid Files

Grid Files

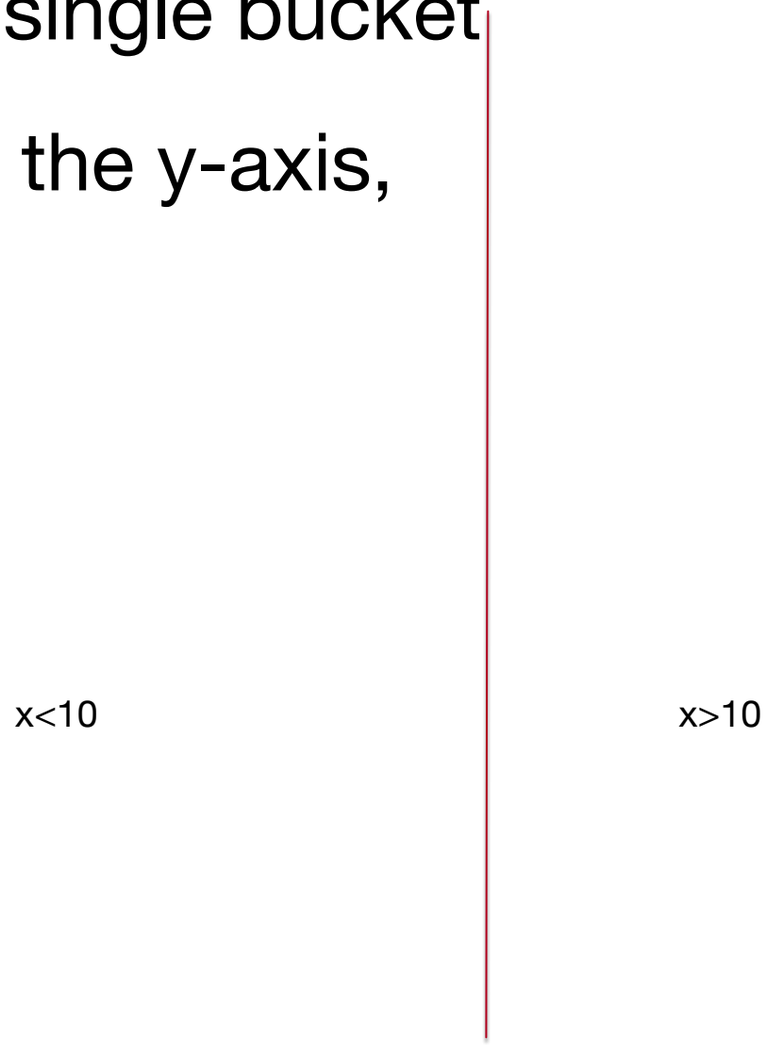
- Many data structures support multi-dimensional indexing
  - Works up to moderate dimensions
    - Geometry of large dimensional spaces is weird

KD Trees

- KD-trees are binary trees where each node is a point in a k-dimensional space
  - Each point divides the space according to a hyperplane

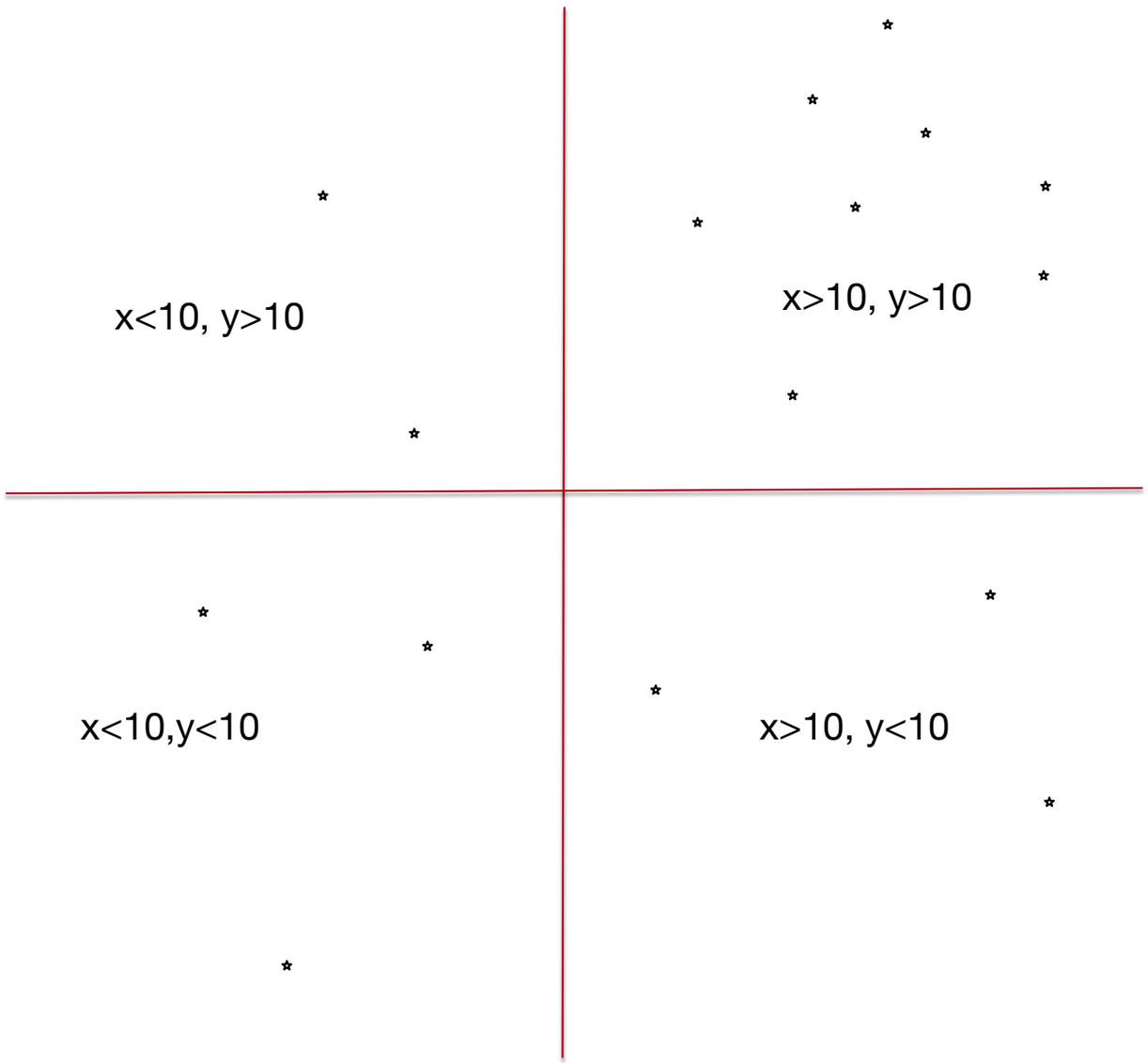
KD Trees

- Two-dimensional example
  - Initially, the whole plane is in a single bucket
  - Then we split the bucket along the y-axis,
  - creating two halves



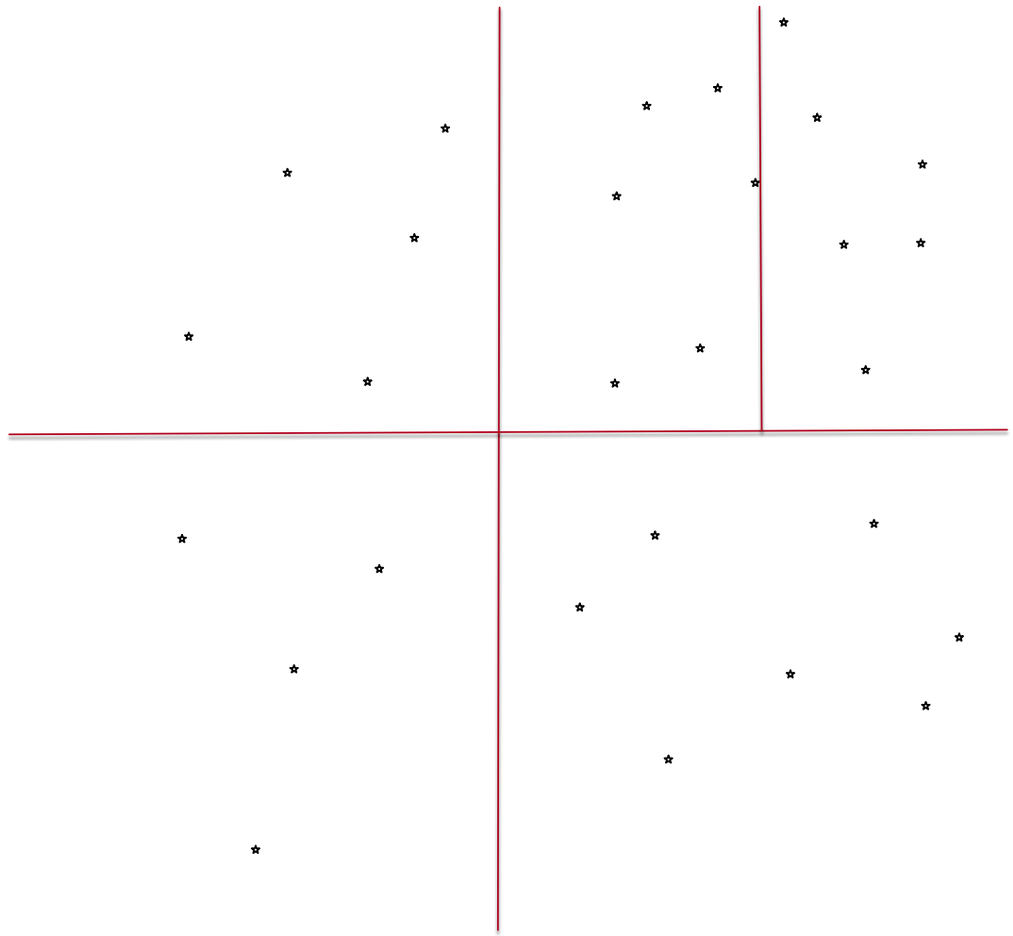
KD Trees

- As we insert more records, we want to subdivide the buckets



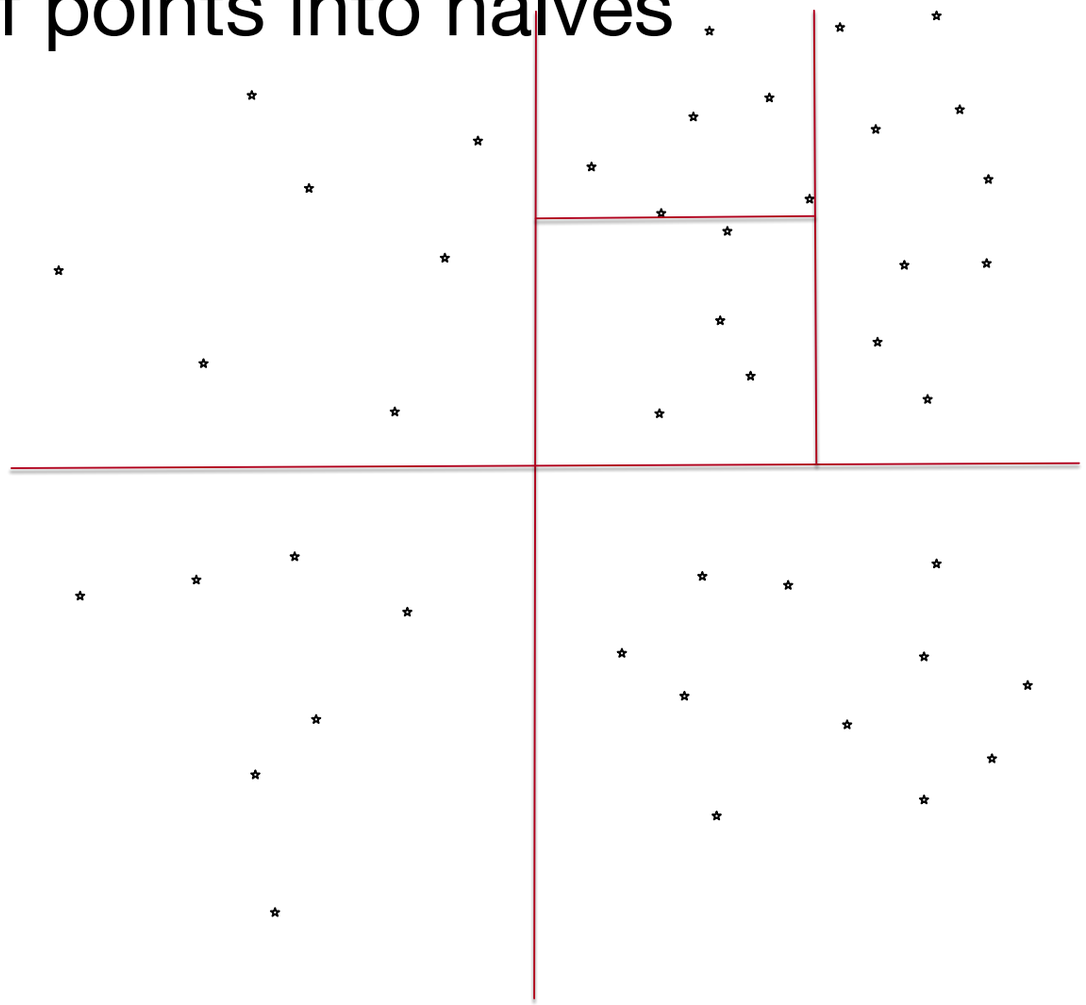
KD-Tree

- The separating axis switches from vertical to horizontal

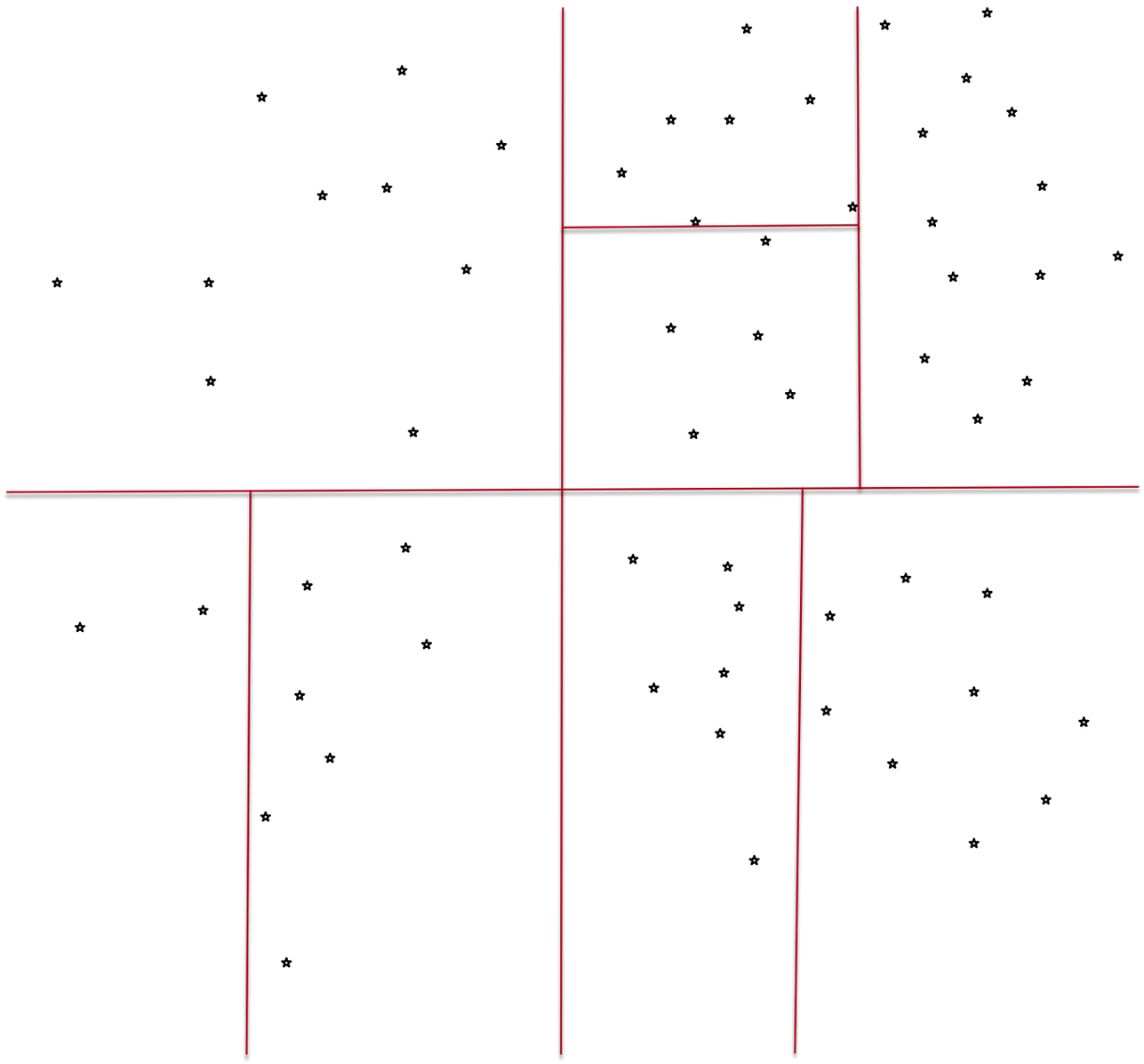


KD-Tree

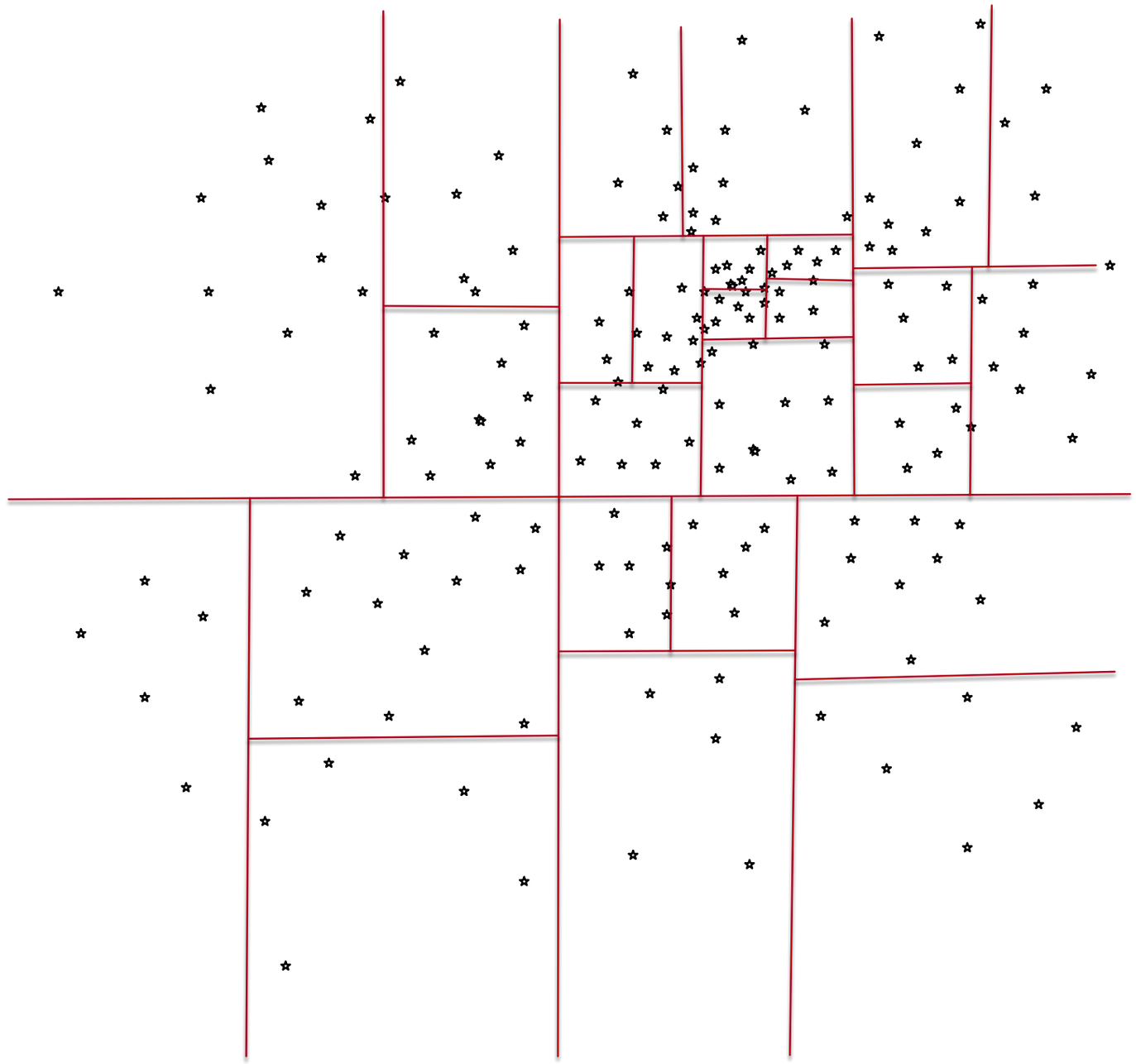
- We select the splitting plane such that it divides the current set of points into halves



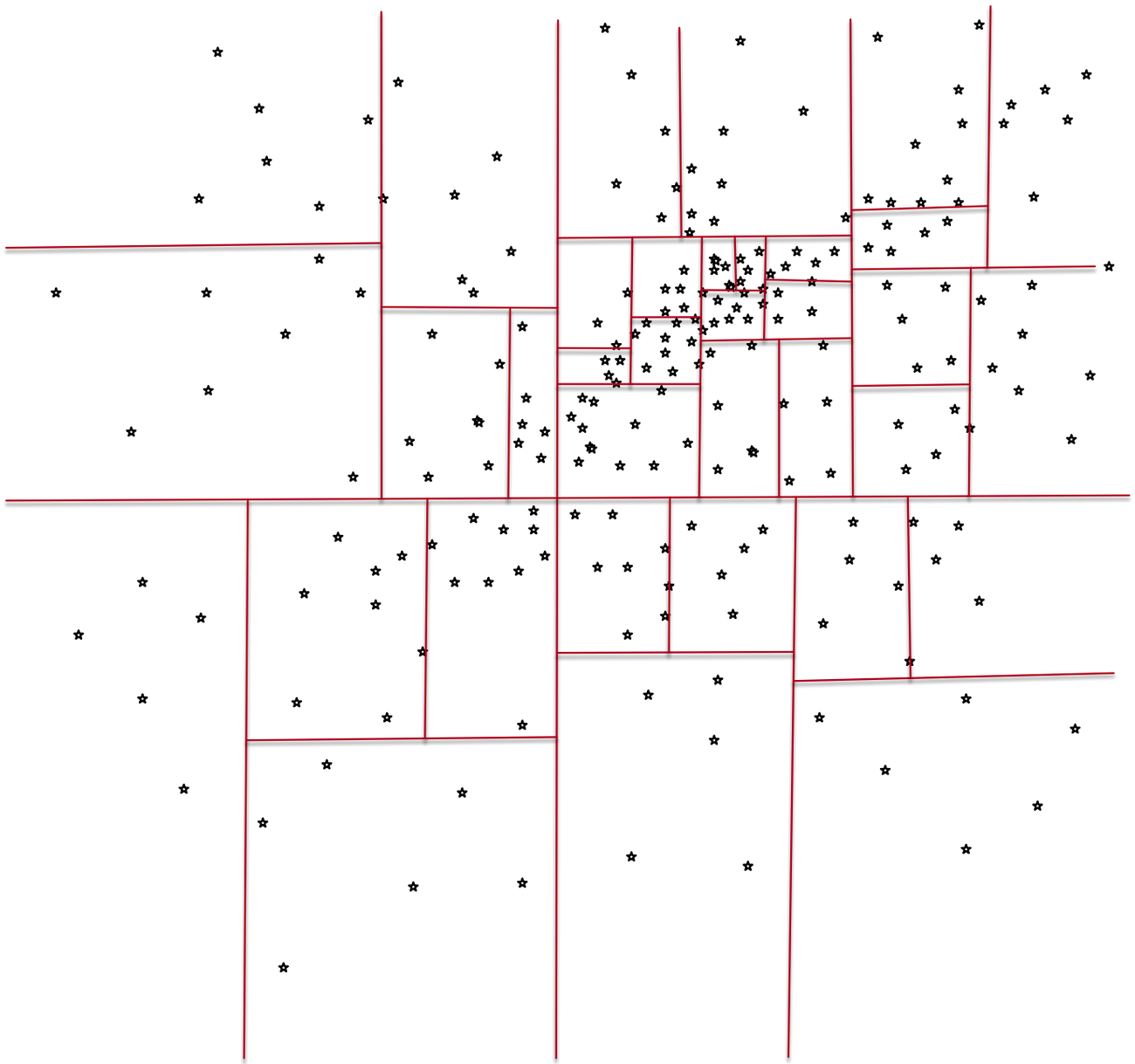
KD-Trees



KD-Trees



KD-Trees

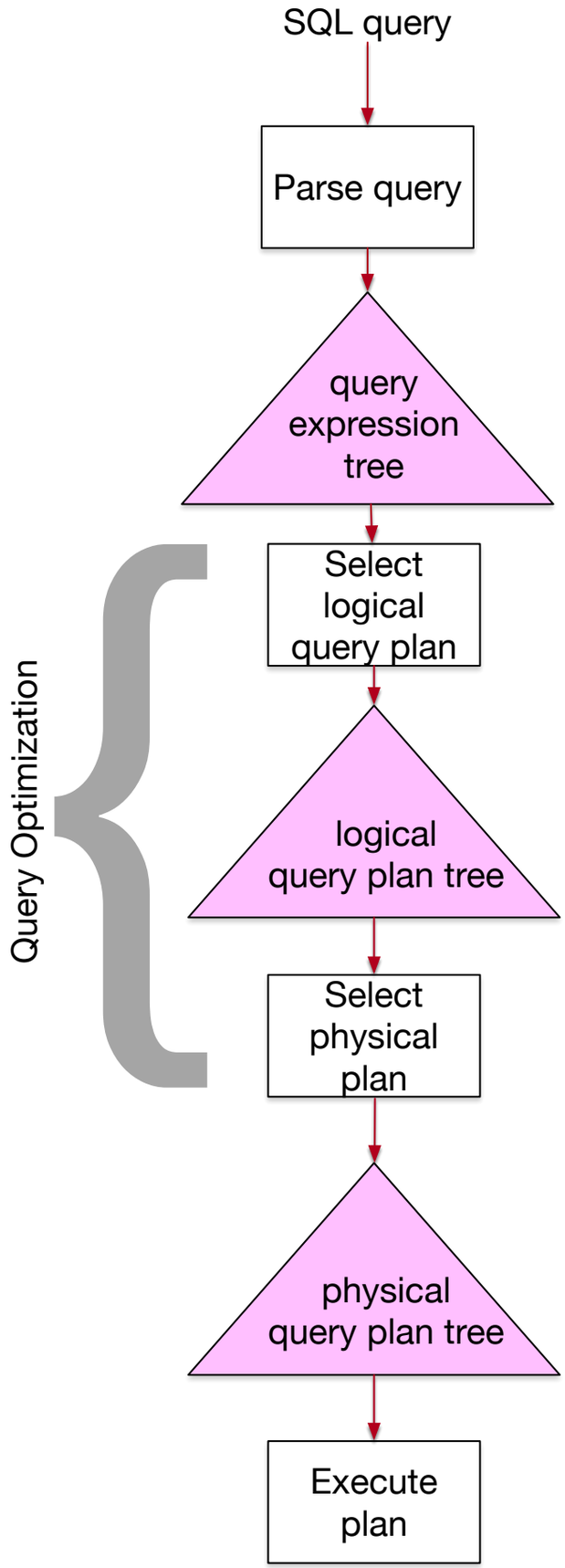


KD-Trees

- This is only one example of many multi-dimensional data structures

Physical Query Plan Execution

Overview


SQL query
Parse query
query expression tree
Select logical query plan
logical query plan tree
Select physical plan
physical query plan tree
Execute plan
Query Optimization

Physical Query Plan Costs

- Usually completely dominated by the cost of I/O
  - Will change with changes in the memory hierarchy

Scanning Tables

- Table-scan
  - Relation is stored in an area of storage
  - Fetch blocks one by one (using prefetching)
- Index-scan
  - If the relation has an index
    - Index has LBAs of all blocks storing tuples (using prefetching)

Scanning Tables

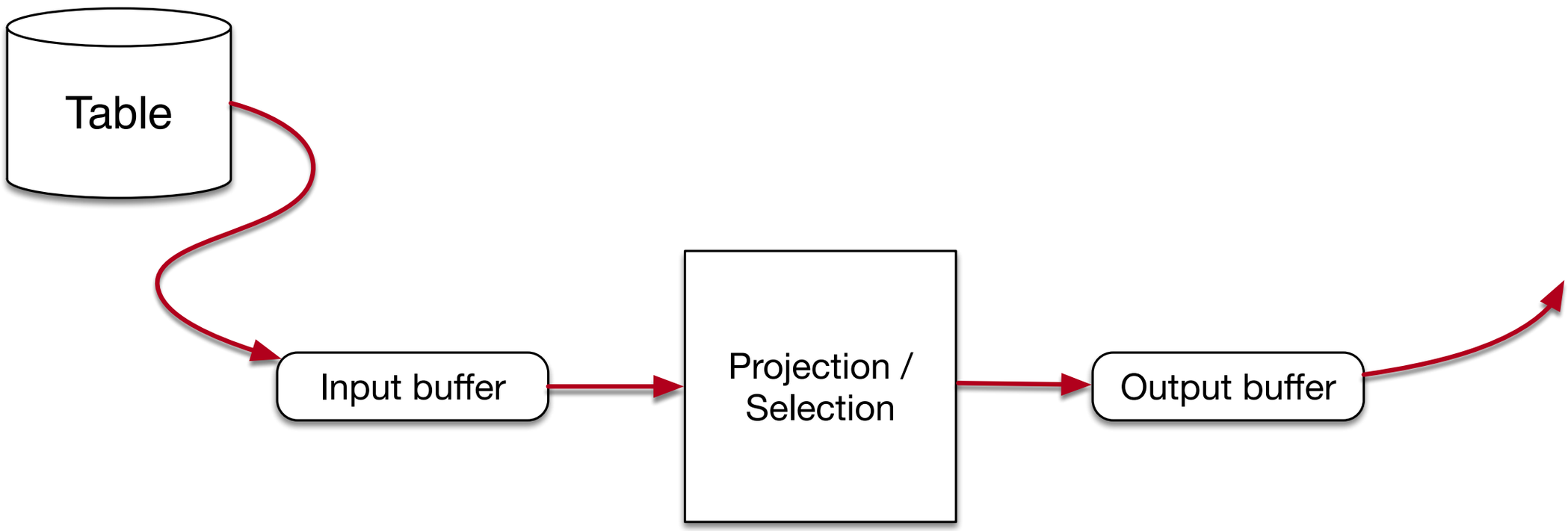
- Sort-scan:
  - As we read a table into main memory:
    - Can sort it
      - Using various data structures
- Sometimes a by-product of the table scan
  - E.g.: Index scan using a B-tree

Iterators

- Records can be clustered or scattered
  - This can make performance prediction difficult
- Use iterators to implement table scanning:
  - Open, Next, Close operations

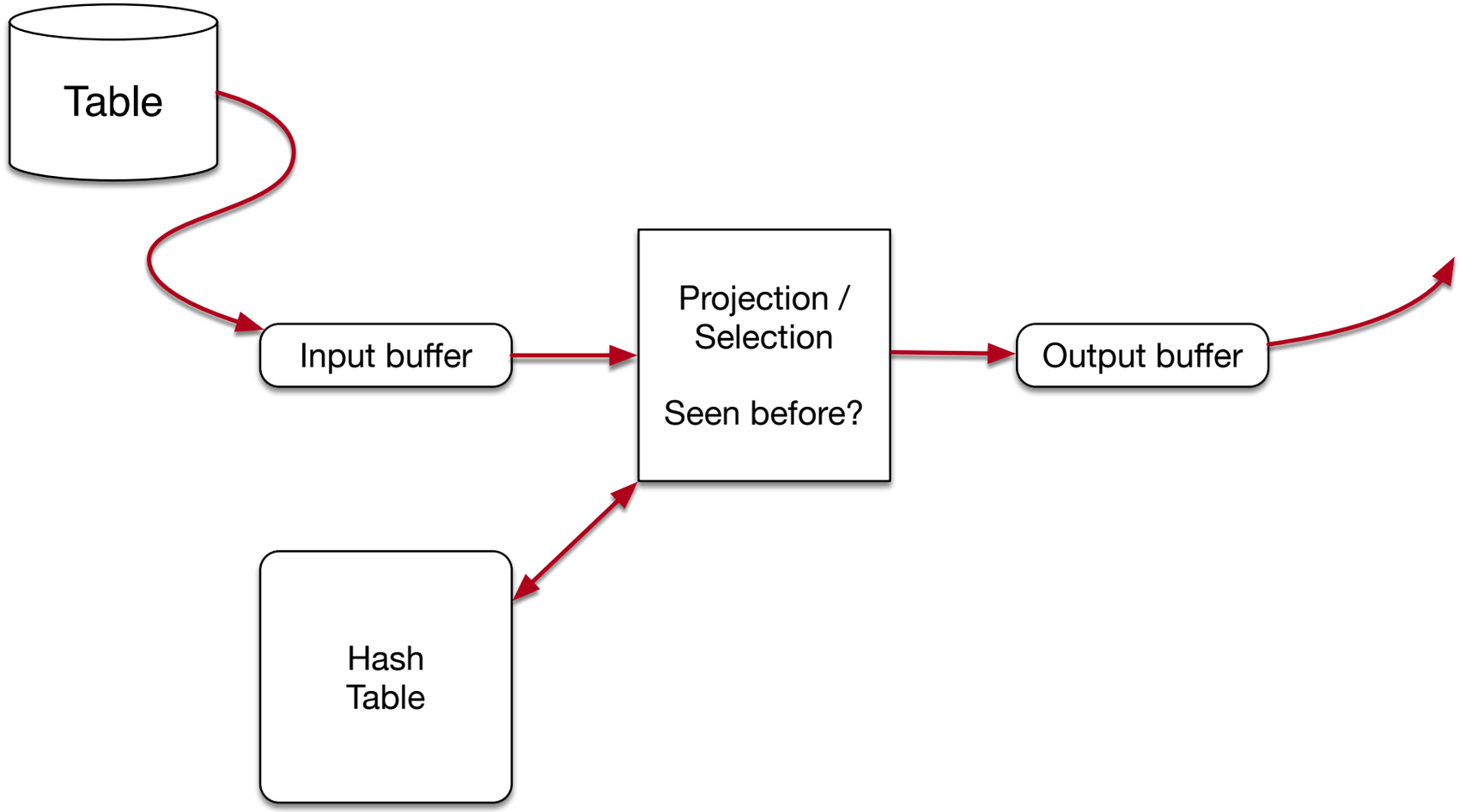
One Pass Operations

- Selection and projection just need to look at one record at a time



One Pass Operations

- Can often be combined with full or partial duplicate elimination
  - Keep seen records in efficient data structure

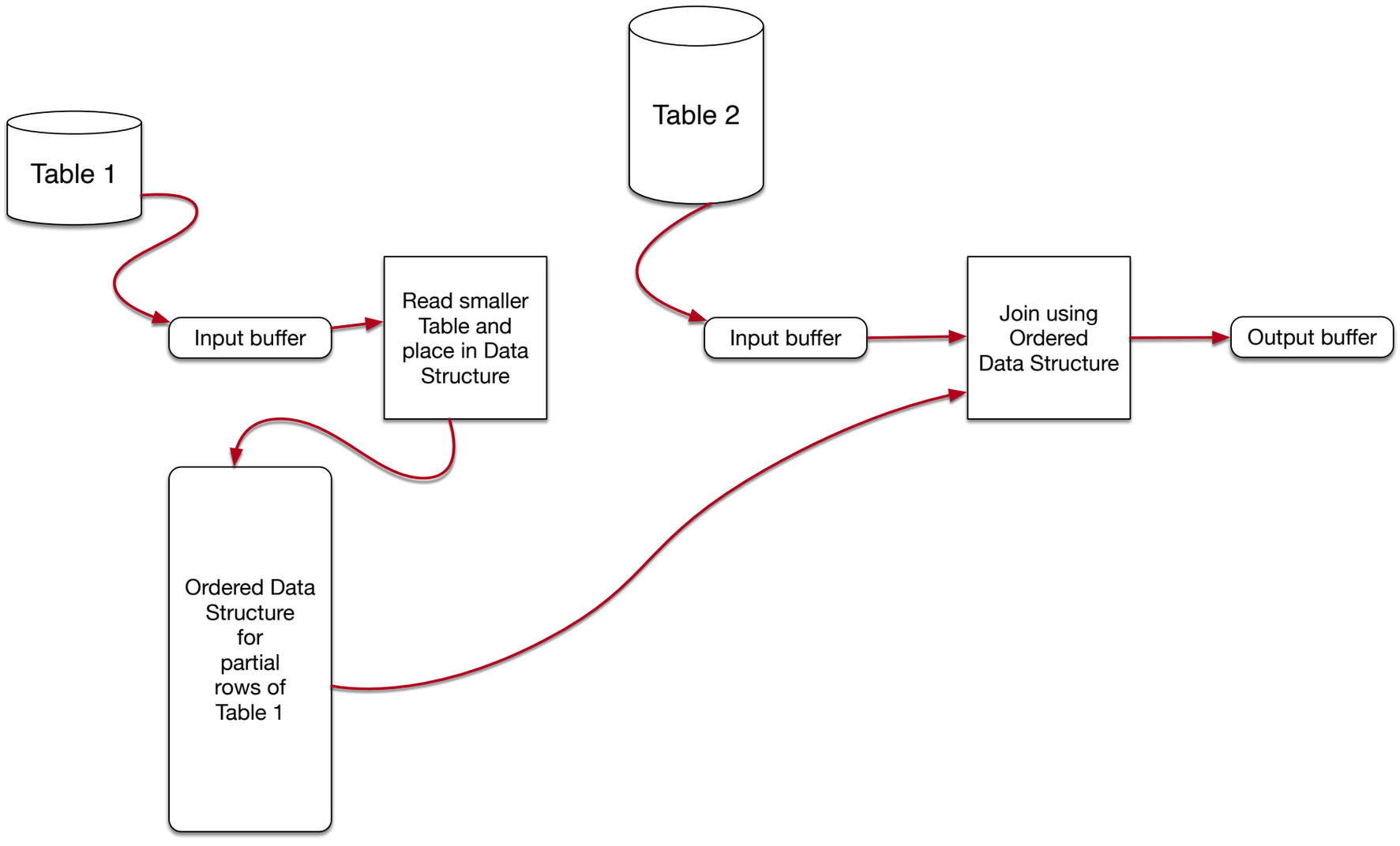


Join Implementation

- Join (on equality) is the most important database operation for performance purposes

One Pass Join Implementation

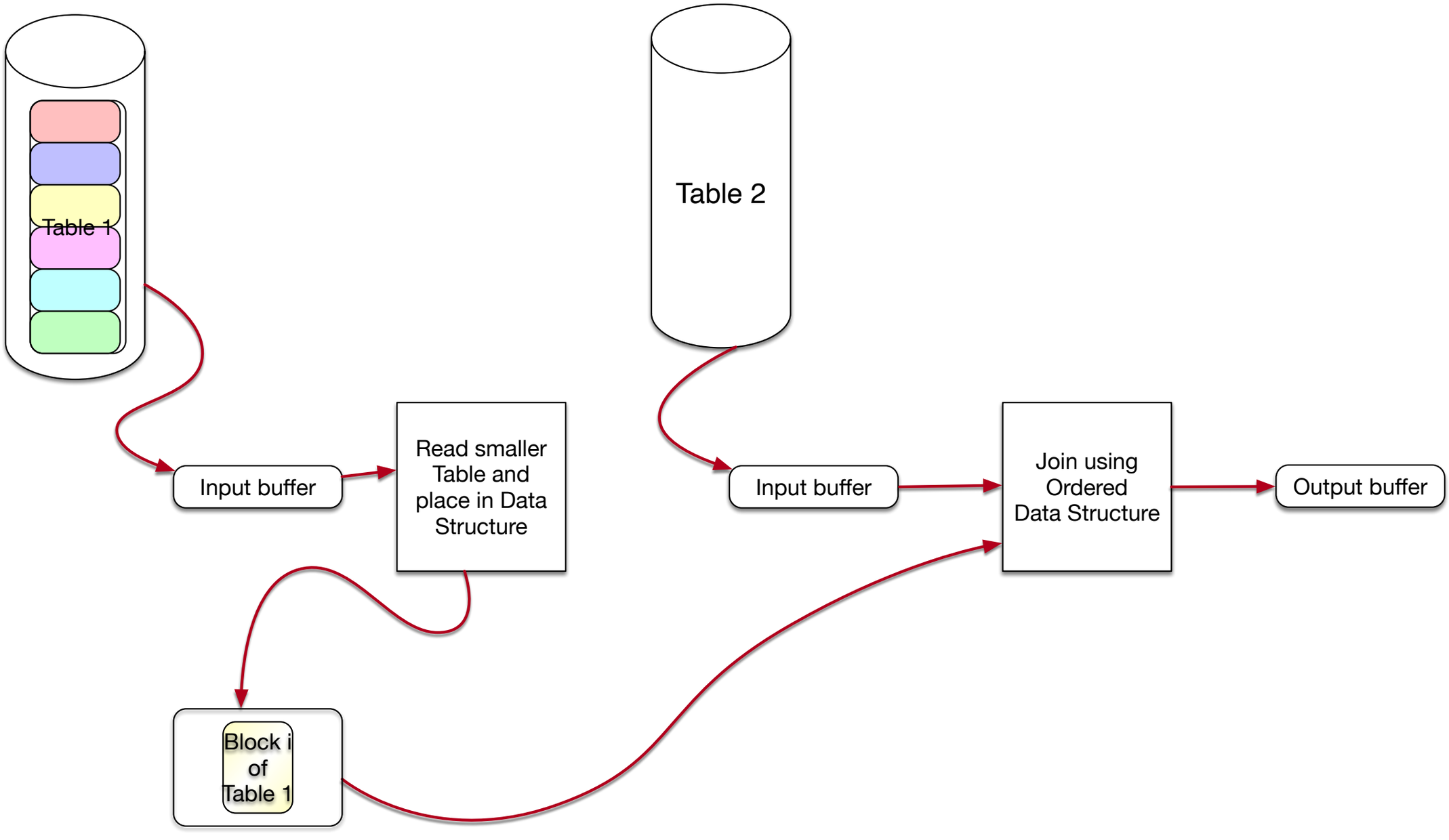
- Load smaller table into a data structure
- Read larger table row by row, compare with data structure



One and a half pass join implementation

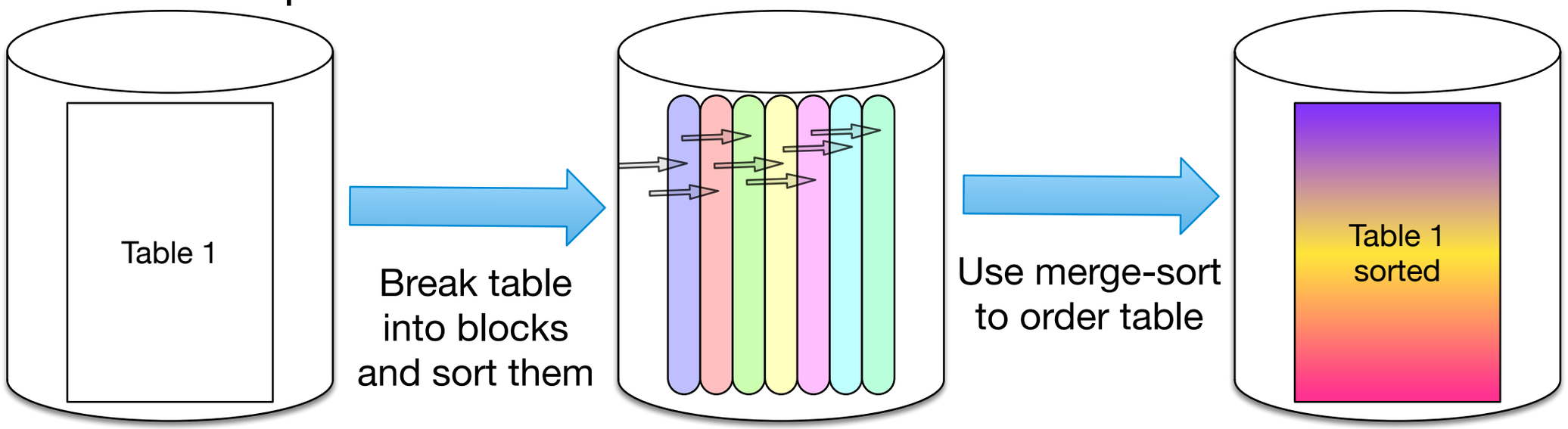
- Nested Loop Joins
  - Idea: If both tables are too big for a one pass, break one table into smaller pieces
    - Can be done tuple by tuple
    - Or block by block

One and a half pass join implementation



Sorting in Two Passes

- Break big table into blocks
- Sort each block
- Merge-sort to generate single ordered table:
  - Point to beginning tuple in each block
  - Write the smallest tuple pointed to into the sorted table
  - Advance pointer



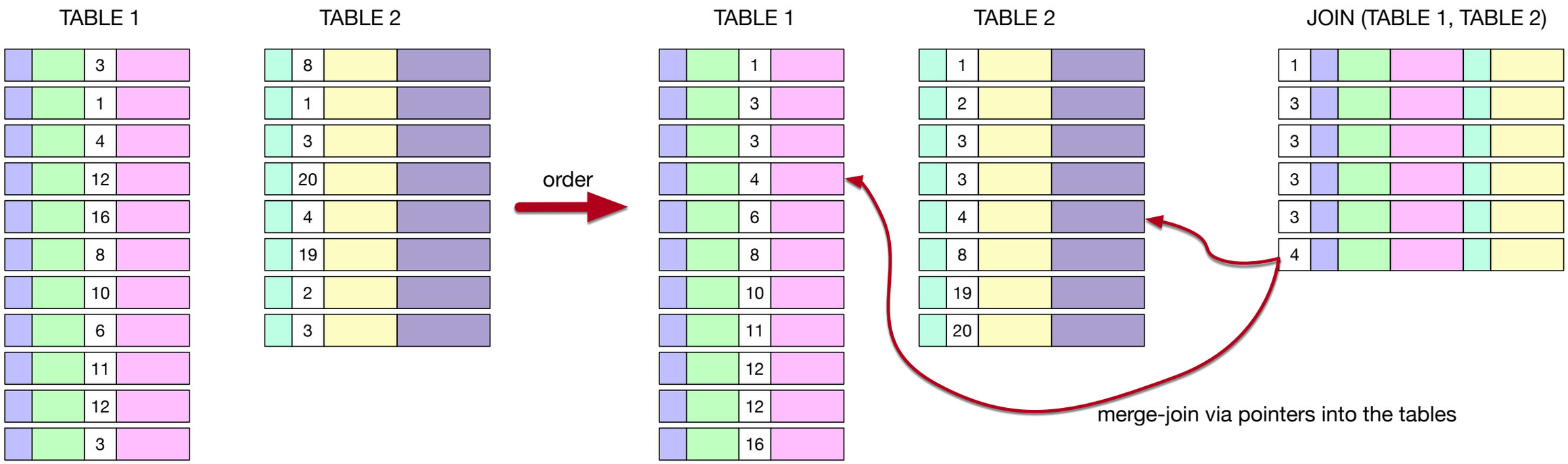
Two Pass Operations

- Same trick can be used for:
  - duplicate elimination
  - grouping and aggregation

Join Implementations

- Sort-merge join
  - Sort both tables by the joining predicate
  - Create two cursors (pointers) into each table
  - Advance the cursors step by step:
    - emitting a new tuple whenever the cursors point to a row with the same value for the join attribute

Join Implementations



We join on the white attribute

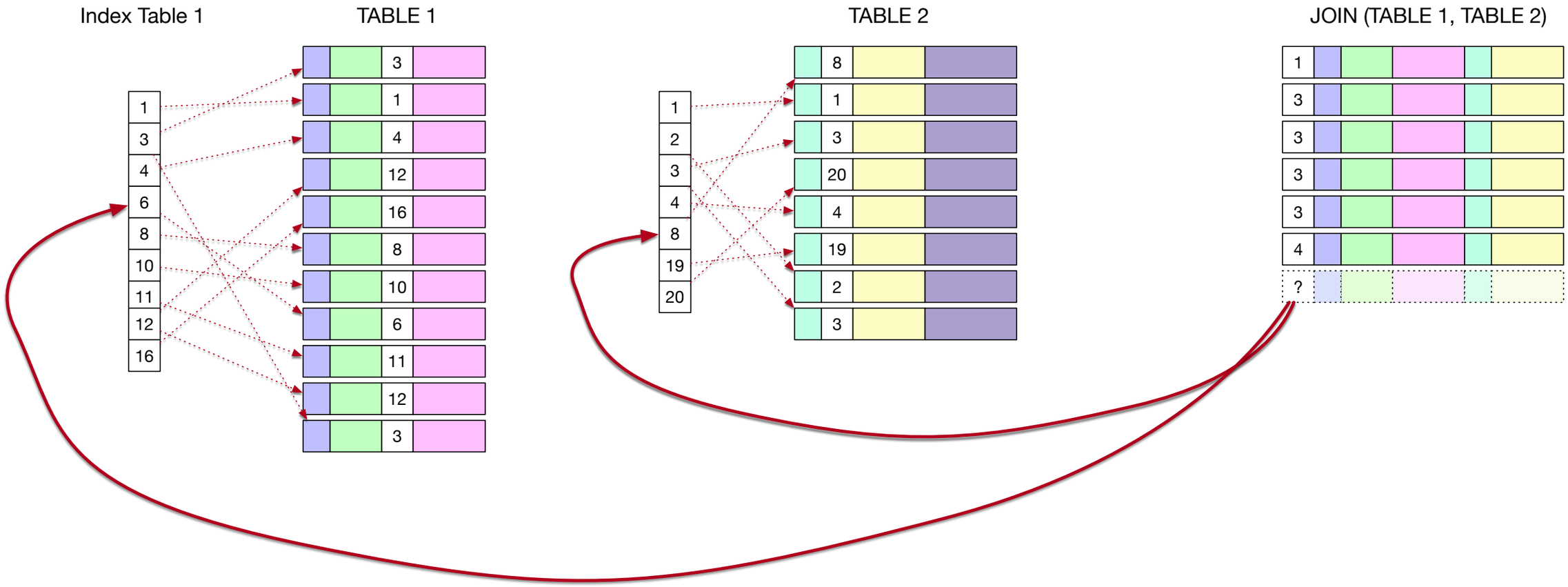
We sort both tables on the white attribute

Create two cursors into the sorted tables. Whenever the cursor points to rows with the same white attribute value, emit a new row. Careful if values repeat.

Join Implementations

- Index join
  - Can create an index on both tables
    - Unless index already exists
  - Use the index as the target for each cursor

Join Implementations



Join Implementations

- Hash Join
  - Select the smaller table as the build side
    - Hash the (addresses of the) tuples in the build side
    - Go through the other (the probe) table row by row
      - For each value, look up the hash bucket of the attribute value and see whether you need to create a row of the join

Join Implementations

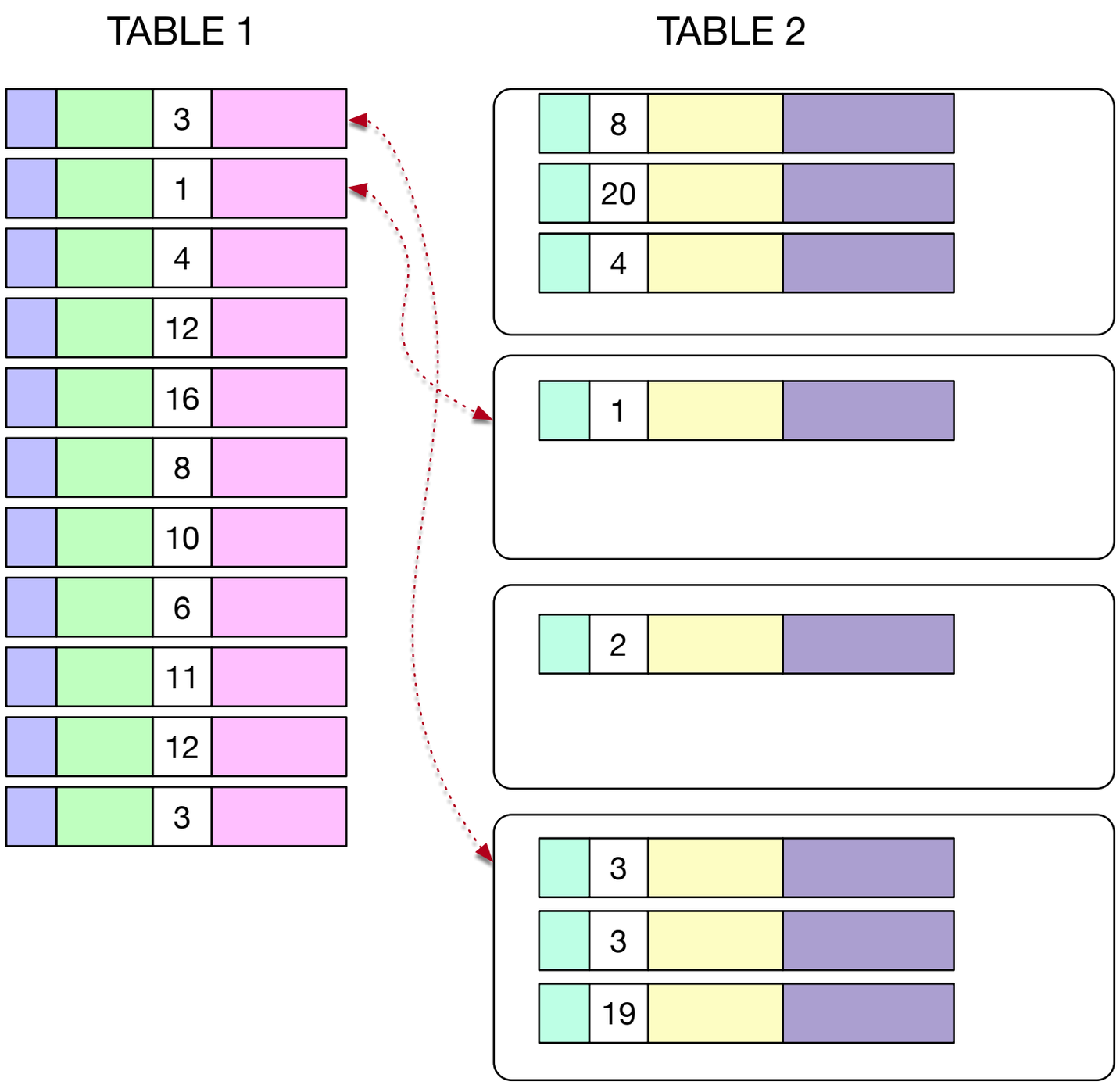


Table 2 is hashed

For each row in Table 1, we join with the rows in the corresponding hash bucket.

Here, the hash function is attribute value modulo 4.

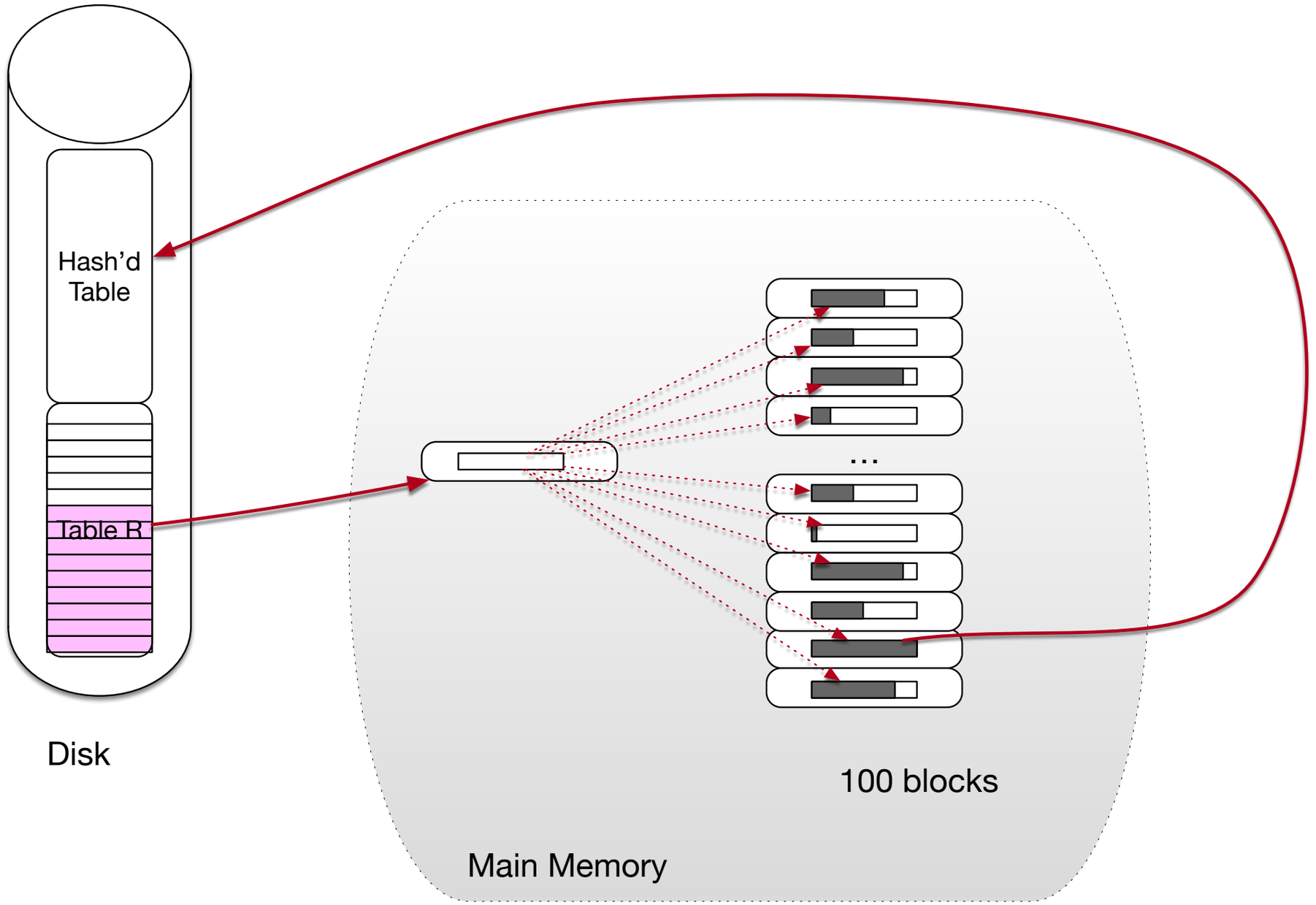
Join Implementations

- If neither table fits into memory
  - Need a two pass algorithm
    - Example: R, S have 5,000 and 10,000 blocks each
    - We have a ridiculously low number of 101 blocks in memory

Join Implementation

- Idea:
  - We create hash buckets for each table independently
  - We read a block from the table
  - Each record in the table is put in one bucket
    - For each bucket, we keep one block in memory
      - If block is full, we move the block to disk and open up a fresh one

Join Implementation



One Pass Hashing for a Relational Table

Join Implementation

- The number of buckets is limited by the number of blocks in main memory
- This method reads all blocks of a table
  - and writes the same number of blocks back
    - unless there is a projection or selection executed simultaneously

Join Implementation

- To join R and S:
  - Create hash table with 100 buckets each
    - Costs $2 \times 5000 + 2 \times 10000 = 30000$ IO
      - A bucket of R uses approximately 50 blocks
  - Bring in one block of the corresponding bucket from S
    - Calculate the join
    - This needs only space for 52 blocks
    - Adds 15000 IO for reading the hashed versions of R and S
    - Needs also write the result

Combined Operations

Pipelining vs. Materialization

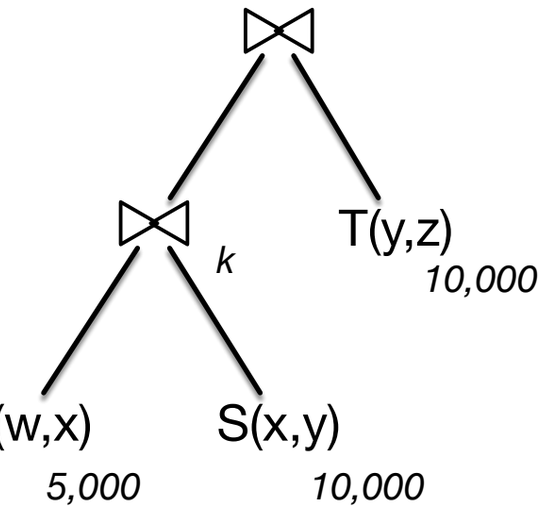
- Combination: Can do some tuple based operations together: e.g. selection and projection for a single table
- Materialization: Write results of an operation into an intermediate table
- Pipelining: Moving results from one operation to inputs of another operation

Pipelining vs. Materialization

- Example for pipelining: Calculate $\left(R(w,x) \bowtie S(x,y)\right) \bowtie T(y,z)$
  - Assume: R has 5,000 blocks, S and T 10,000 blocks
  - Both joins will be implemented as hash joins
    - one pass or two pass
  - We only have 102 block buffers available
    - This is a very low number

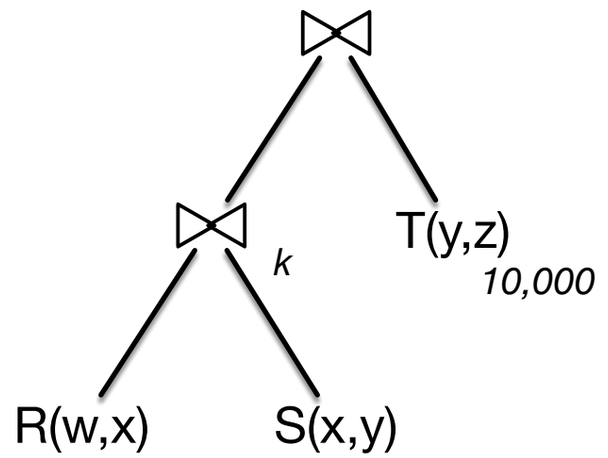
Pipelining vs. Materialization

- To join R and S:
  - Neither table fits into main memory, so we need a two-pass hash-join
    - Join R and S in $45000 + k$ I/O
    - If k is small (<45 blocks), keep result in MM
    - Materialization: use a 1-pass hash join with T: $k + 10000$ I/O
    - Pipeline (because the second pass of the join of R and S does not need all the space) we save $2k$ I/O



Pipelining vs. Materialization

- If the output of the join of $R \bowtie S$ does not fit in MM, we can still pipeline with a trick
  - The second phase of the join of $R \bowtie S$ leaves 50 blocks of MM space unused
  - We use this space to hash $R \bowtie S$ into 50 buckets as before
  - We assume that we already hashed T into 50 buckets.
  - We then hash-join $R \bowtie S$ and T
    - This will work if a bucket of $R \bowtie S$ fits into 100 blocks
    - Thus, $R \bowtie S$ should fit into 5000 blocks



Costs:
45000 IO for $R \bowtie S$
k IO to store hashed version of $R \bowtie S$
20000 IO to generate hashed version of U
k + 10000 to join U with $R \bowtie S$
Total: $75000 + 2k$

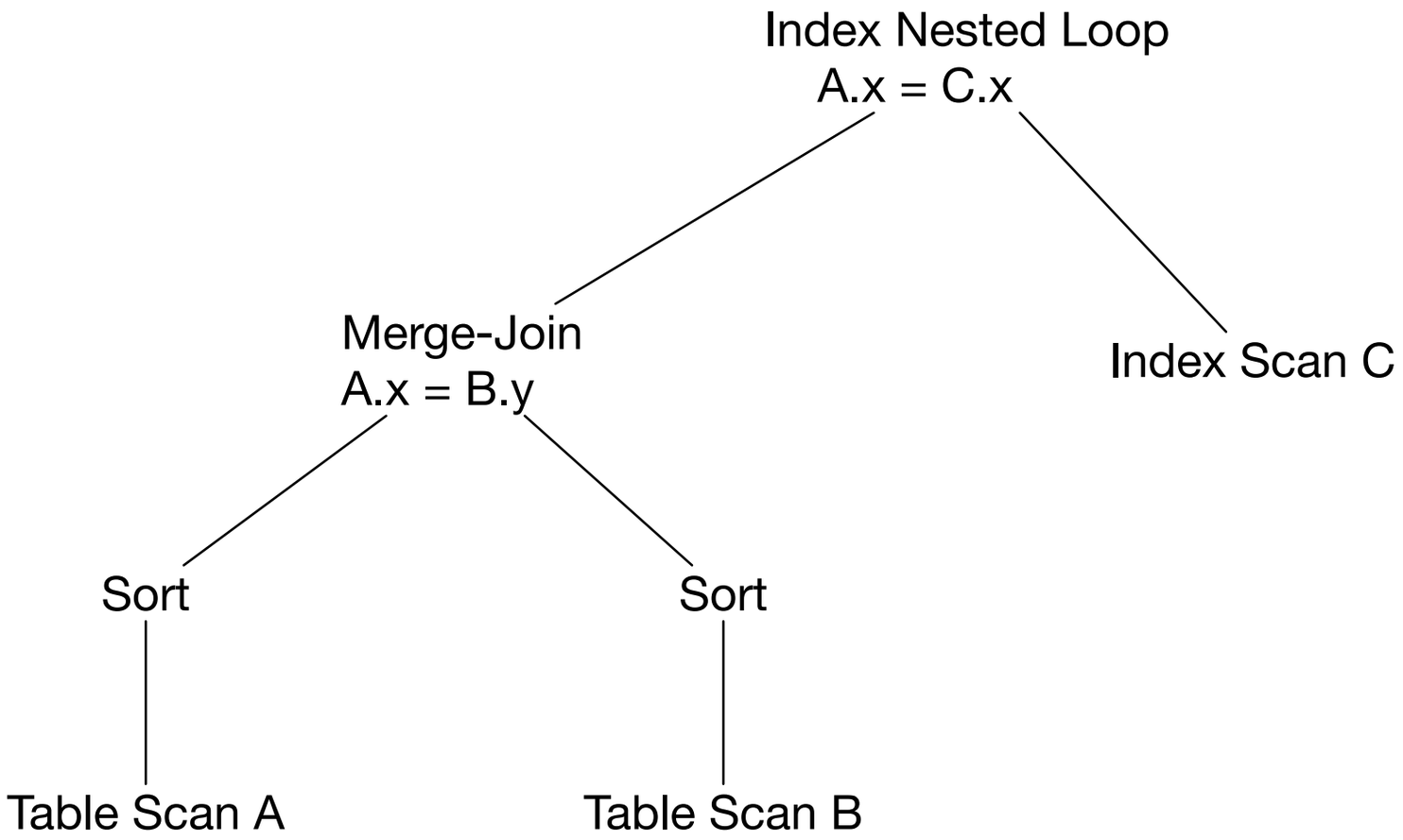
Pipelining vs. Materialization

- What happens if $R \bowtie S$ has $k > 5000$ blocks?
  - In this case, pipelining can be replaced with materialization
  - Calculate $R \bowtie S$ using $45000 + k$ I/O, storing the result on the disk
  - Use T as the build table, using 100 buckets, costing 20000 I/O
  - Now join with $R \bowtie S$, costing another $10000 + 2k$ I/Os

Query Optimization

Basics

- Queries can be expressed as an operator tree



Basics

- Any query can be expressed as many equivalent algebraic representation
- Any algebraic representation can be represented by many operator trees
- The throughput / response time of any execution of an operator tree can differ widely from timings for other operator trees
- Each operator tree is a different "*query plan*"

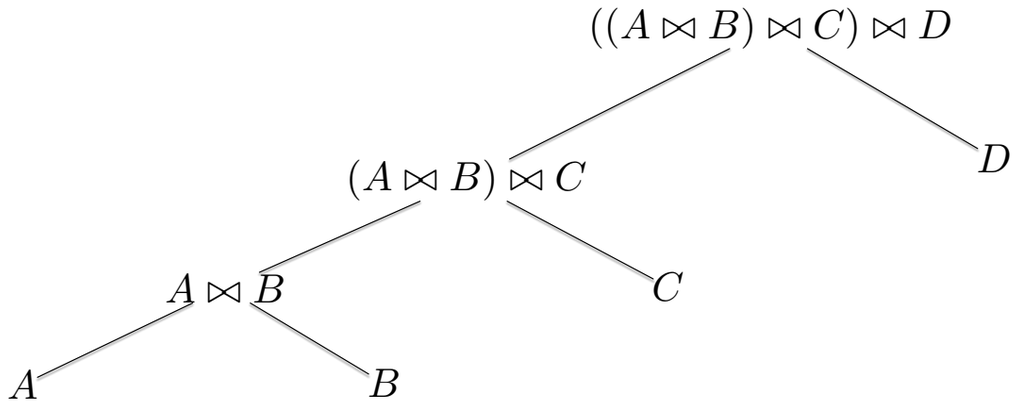
Basics

- Query optimization needs
  - A "search space"
    - Set of equivalent query plans
  - An enumeration algorithm to search the search space
  - A cost estimation technique
    - Needs to deal with possibility for parallelism
      - This is more important for SSD than for HDD

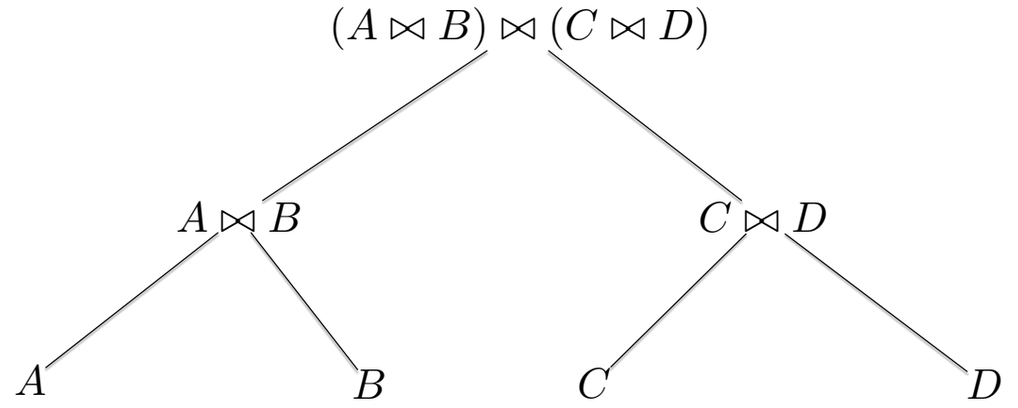
Basics

- SPJ (Select Project Join) queries
  - Standard SQL query consisting of select, project, and join operations

Basics



Linear Join

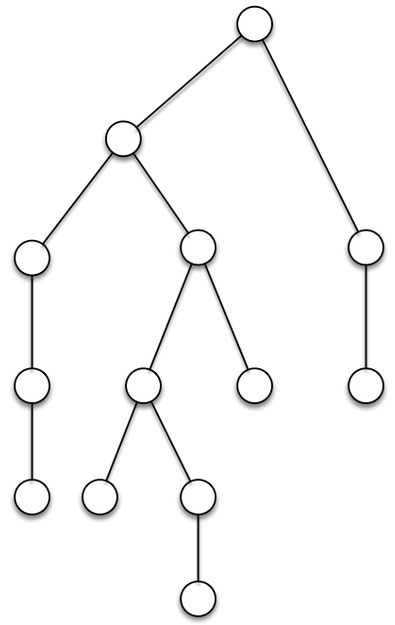


Bushy Join

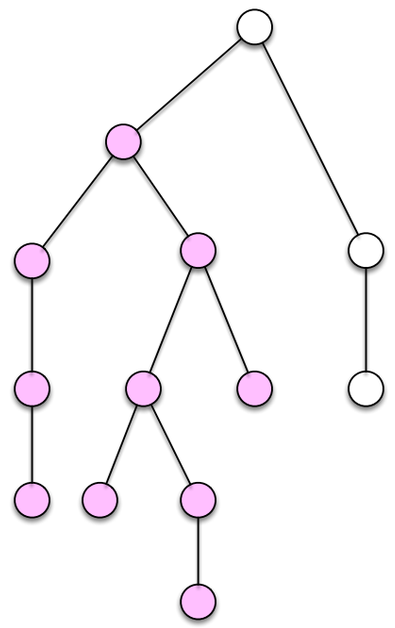
- The resulting queries are equivalent
  - Execution times can vary considerably
  - Dynamic Programming approach
    - *Principle of Optimality:* If a subexpression is optimal, it is also best for the calculation of the whole expression

Basics

- Principle of optimality
  - If the execution of the whole tree is optimal



- then the execution of a subtree is also optimal



Query Cost Evaluation

- Query plan costs depend on the size of the intermediate results
  - For cost-estimation:
    - Use heuristics
    - Make assumptions on probability distributions
    - Use statistics by using samples