

Data Structures

Algorithms

Types of Data Structures

- Organize data to make access / processing fast
 - Speed depends on the internal organization
 - Internal organization allows different types of accesses
- Problems:
 - Large data is nowadays distributed over several data centers
 - Need to take advantage of storage devices

Types of Data Structures

- Internal Memory
 - DRAM: fast access, byte addressable
- Storage
 - Hard Disk Drives
 - Data in blocks
 - Decent for streaming (consecutive blocks)
 - Bad for random access (~10 msec per access)
 - Solid State Disks
 - Data in blocks (called pages)
 - Decent access times (~1msec per access)

Types of Data Structures

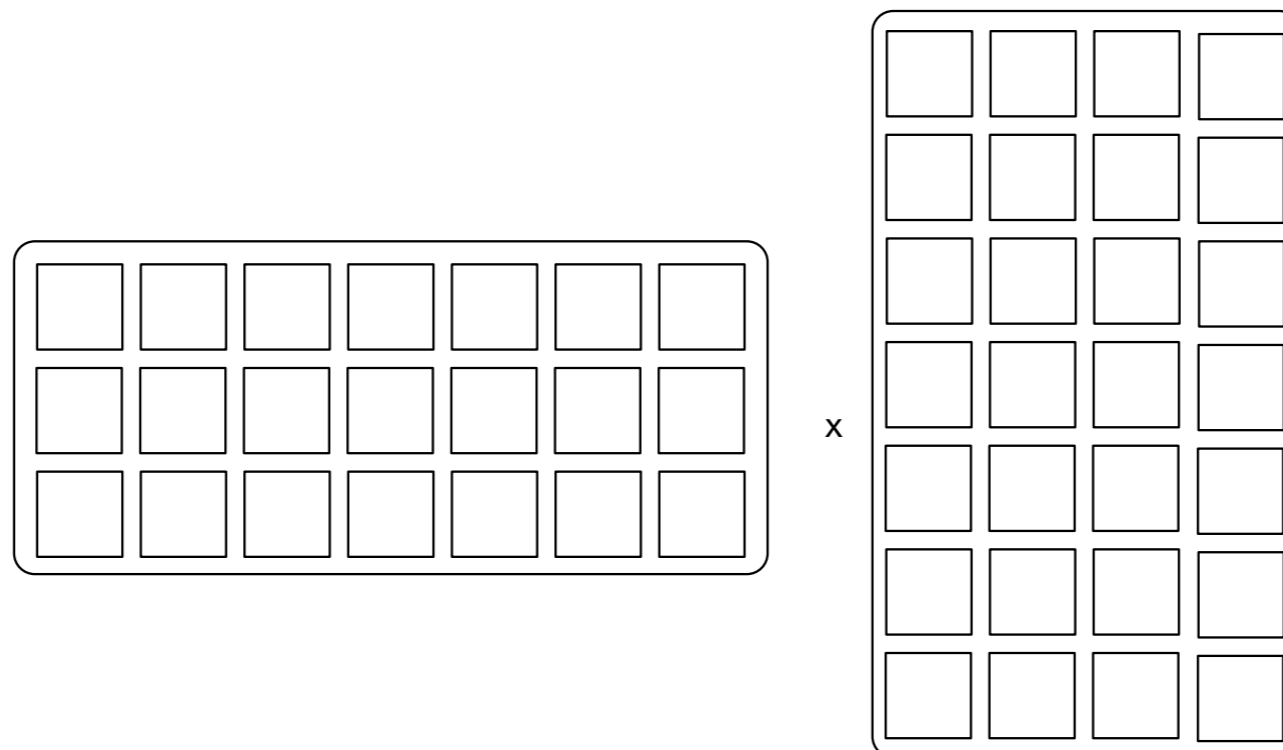
- Thread safe:
 - Several threads can safely access data structure
 - Need collaboration between threads
 - Implemented with locks
 - Implemented without locks
 - Difficult to do
 - Needs atomic instructions: Hardware and compiler support

Types of Data Structures

- Caches can make big performance differences
 - Cache aware algorithms
 - Get the parameter of the caches
 - Cache oblivious algorithms
 - Work well for all cache sizes
- Dumb algorithms
 - Do not pay attention to caches at all
 - Frequent surprises with bad performance

Example

- Multiplying two big, non-dense matrices
 - Cache aware:
 - Break matrices into subsquares
 - Three subsquares fit comfortably into cache



Example

- Cache Oblivious
 - Use a Divide and Conquer Algorithm that subdivides the sub-squares repeatedly
 - Only **cold** cache misses when a new sub-square needs to be loaded into cache.

Types of Data Structure

- Dictionary — Key - Value Store
 - CRUD operations: create, read, update, delete
 - Solutions differ regarding read and write speeds

Types of Data Structure

- Range Queries (Big Table, RP)
 - CRUD and range operation

Types of Data Structure

- Priority queue:
 - Insert, retrieve minimum and delete it

Types of Data Structure

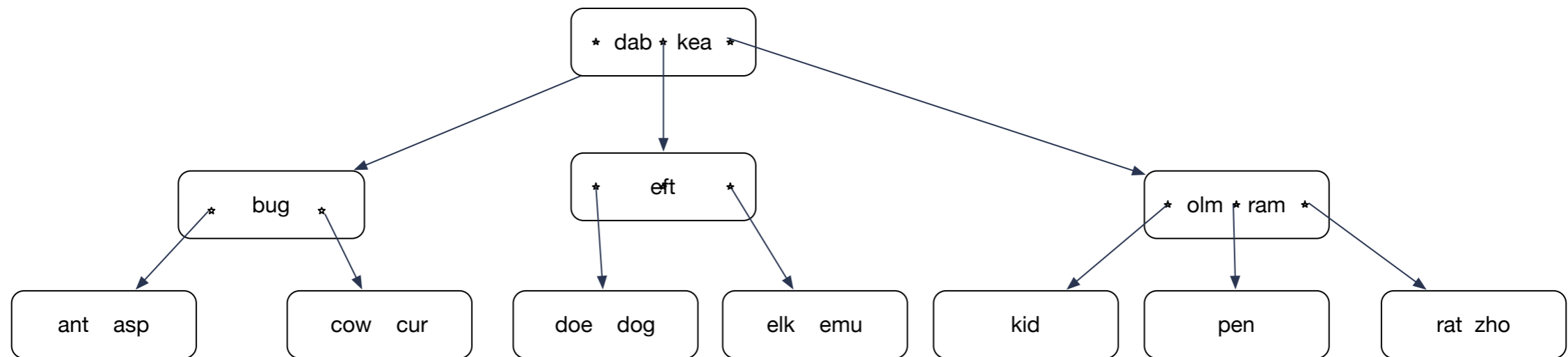
- Log:
 - Append, Read

B-Trees

- B-trees: In-memory data structure for CRUD and range queries
 - Balanced Tree
 - Each node can have between d and $2d$ 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

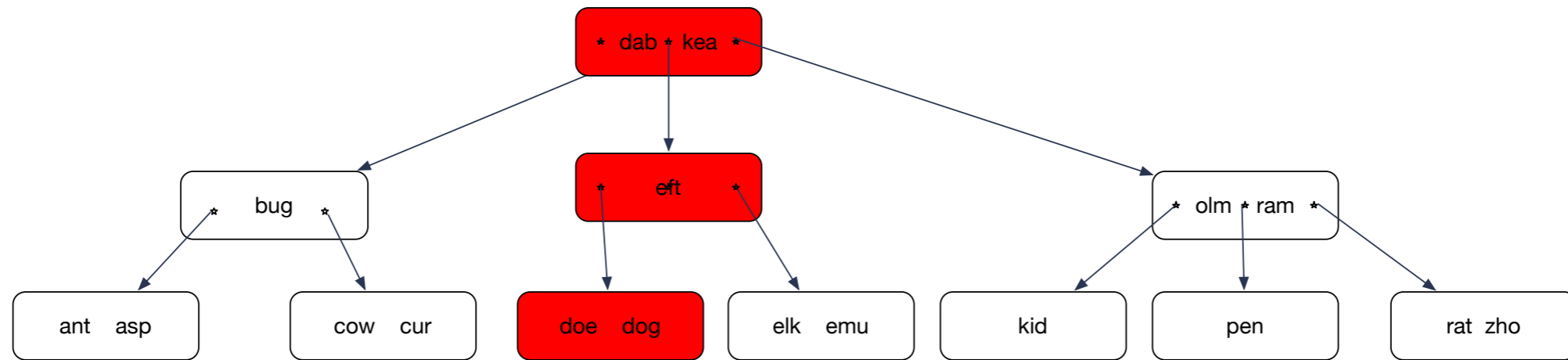
- Example: 2-3 tree: Each node has two or three children



B-Trees

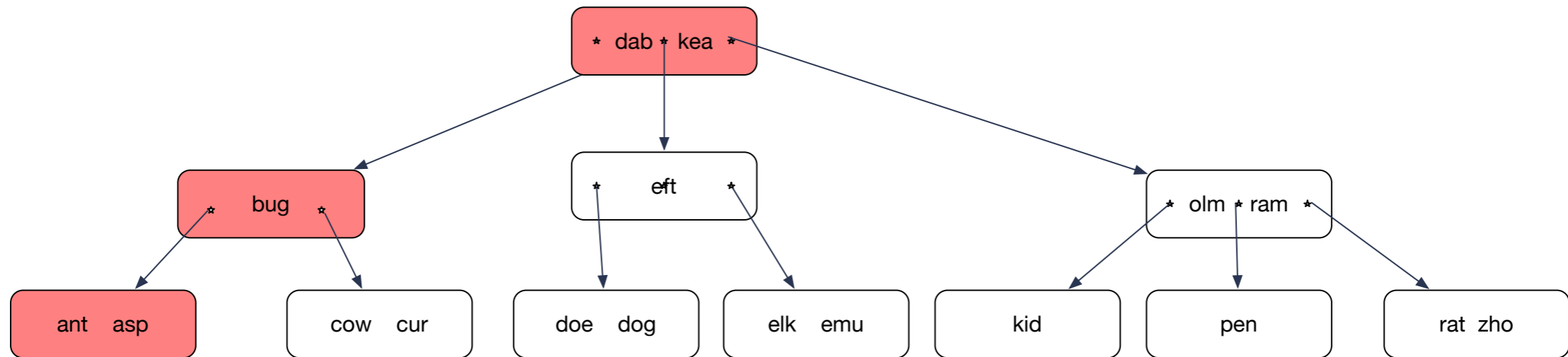
- Read dog:
 - Load root, determine location of dog in relation to the keys
 - Follow middle pointer
 - Follow pointer to the left
 - Find “dog”

B-Trees



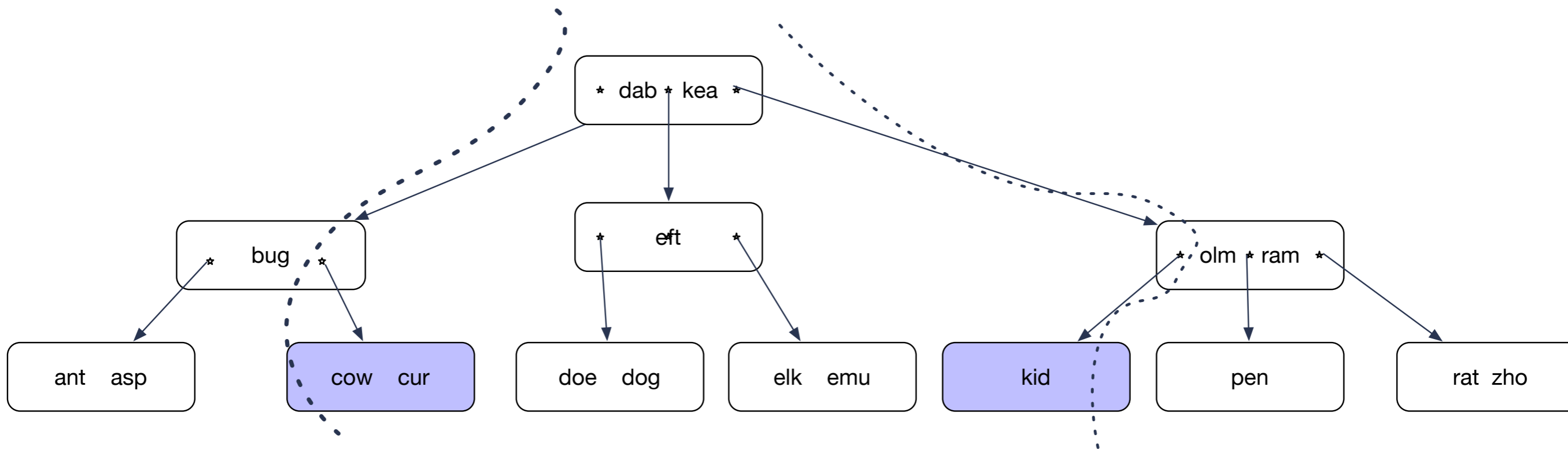
B-Trees

- Search for “auk” :



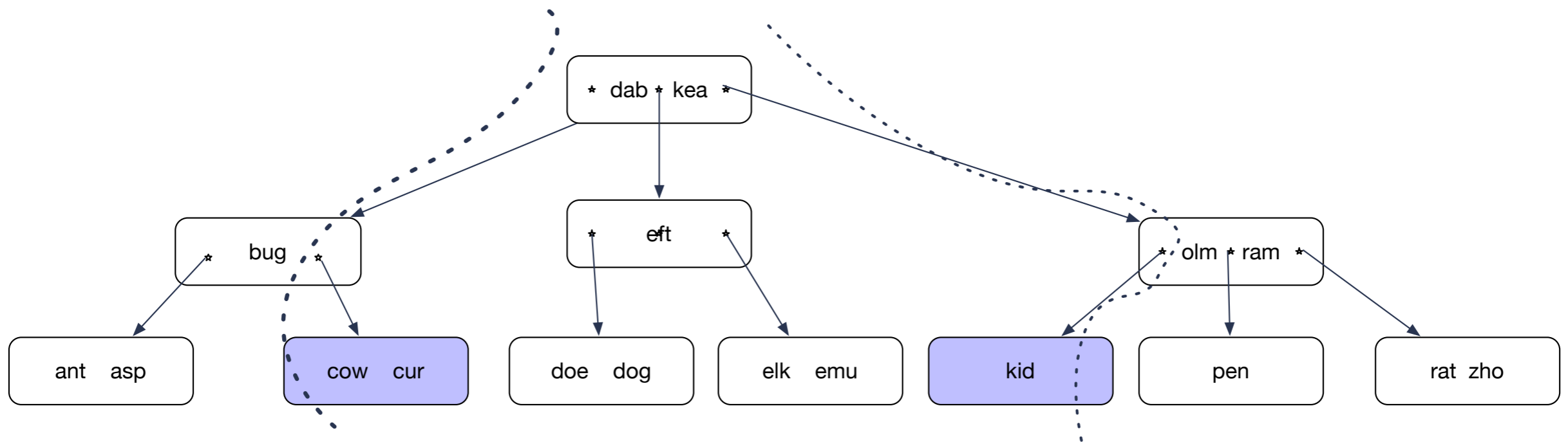
B-Trees

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



B-Trees

- Recursively enumerate all nodes between the lines starting with root



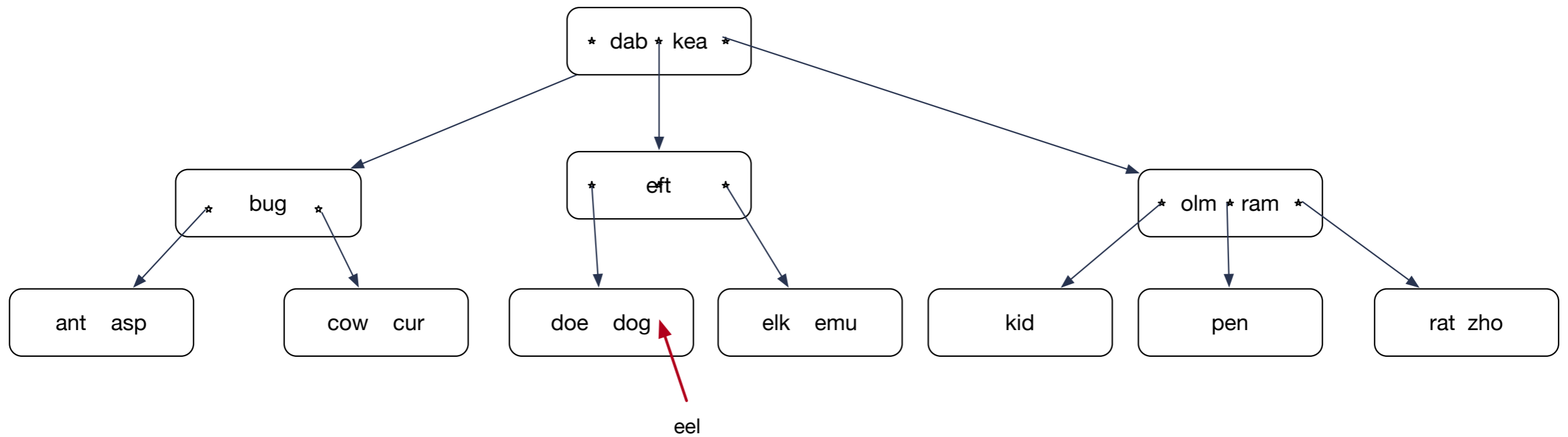
B-trees

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

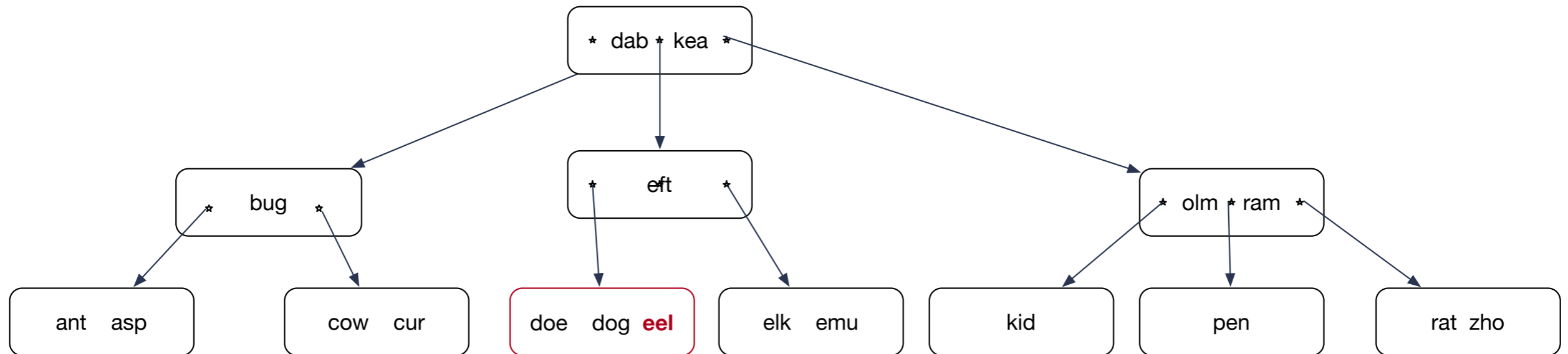
B-trees

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

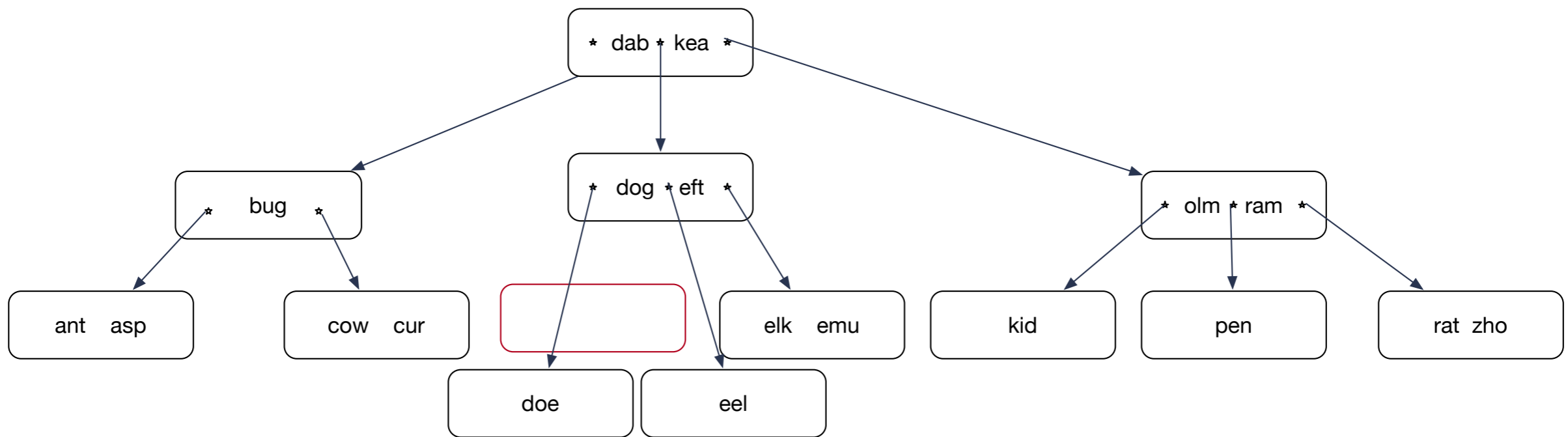
B-trees



B-trees



B-trees



*

B-trees

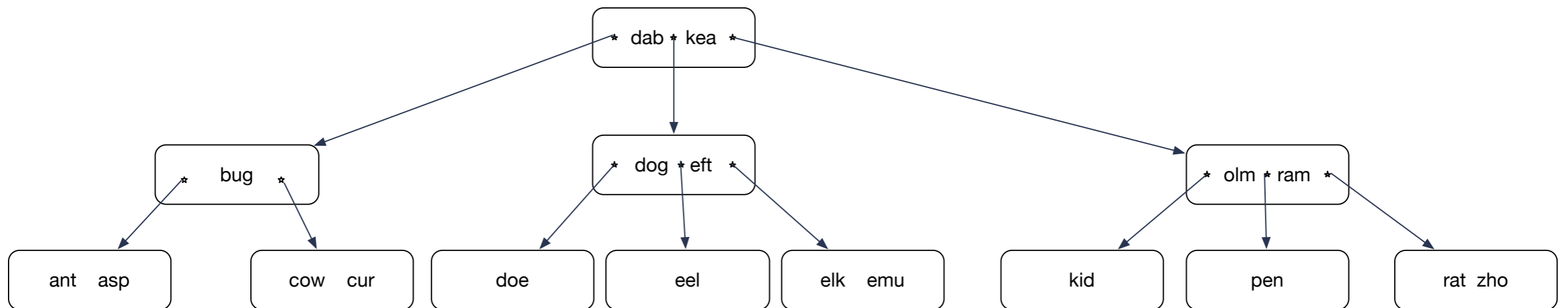
- Insert: Lock all nodes from root on down so that only one process can operate on the nodes
- Tree only grows a new level by splitting the root

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

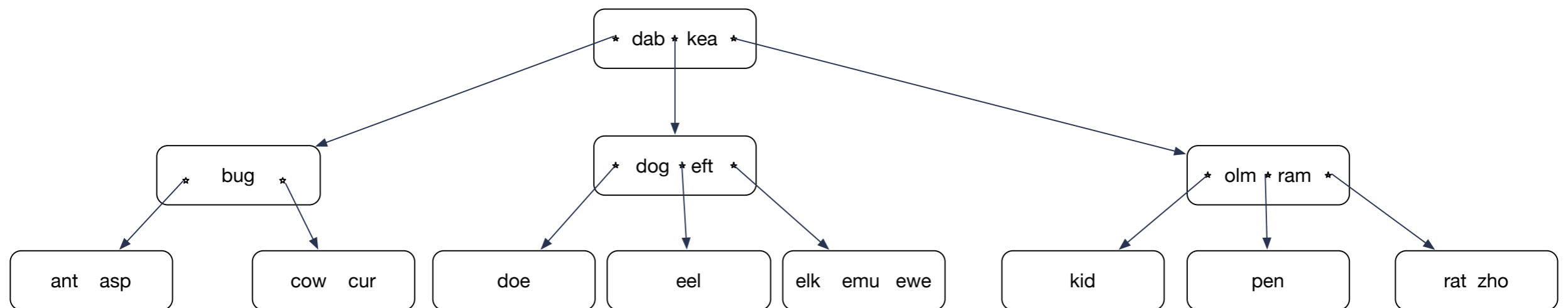
B-tree

- Insert ewe into



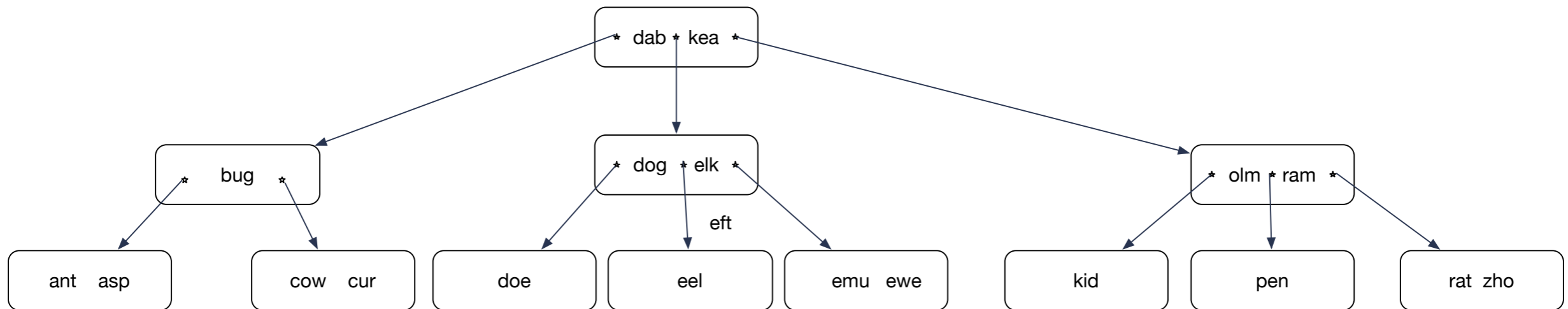
B-tree

- Insert ewe



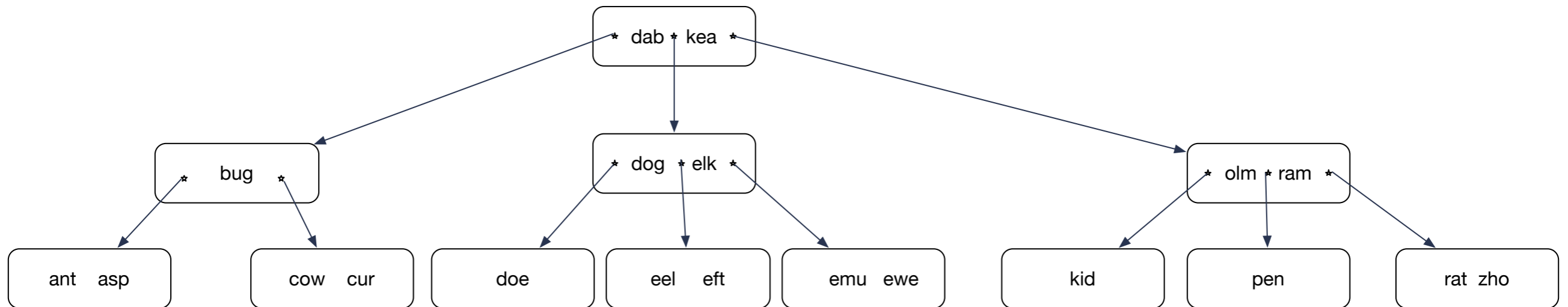
B-tree

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



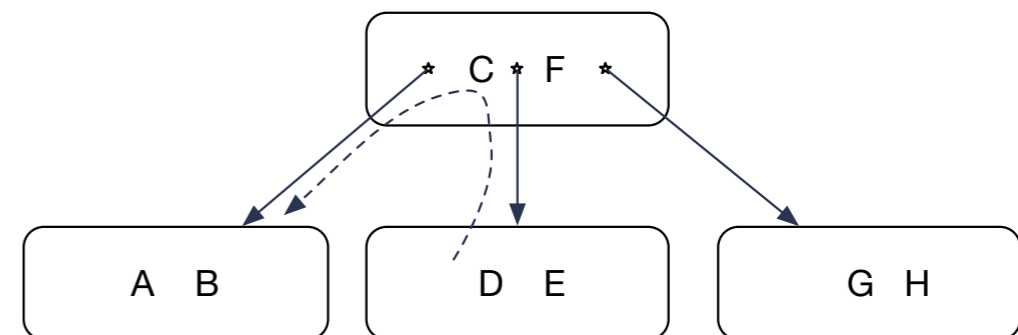
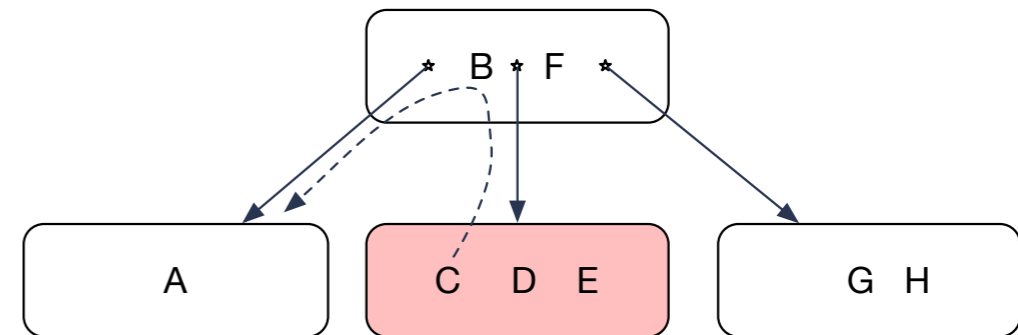
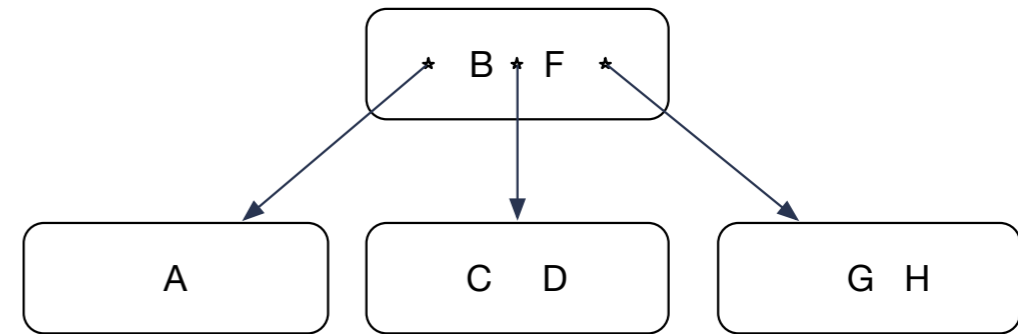
B-tree

- Insert eft into the leaf node

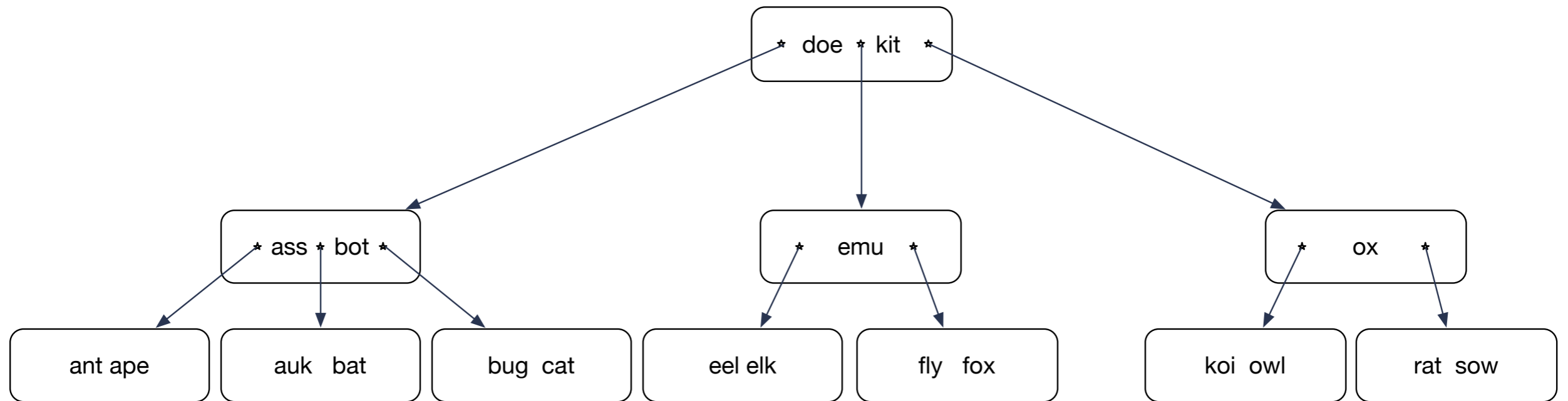


B-tree

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

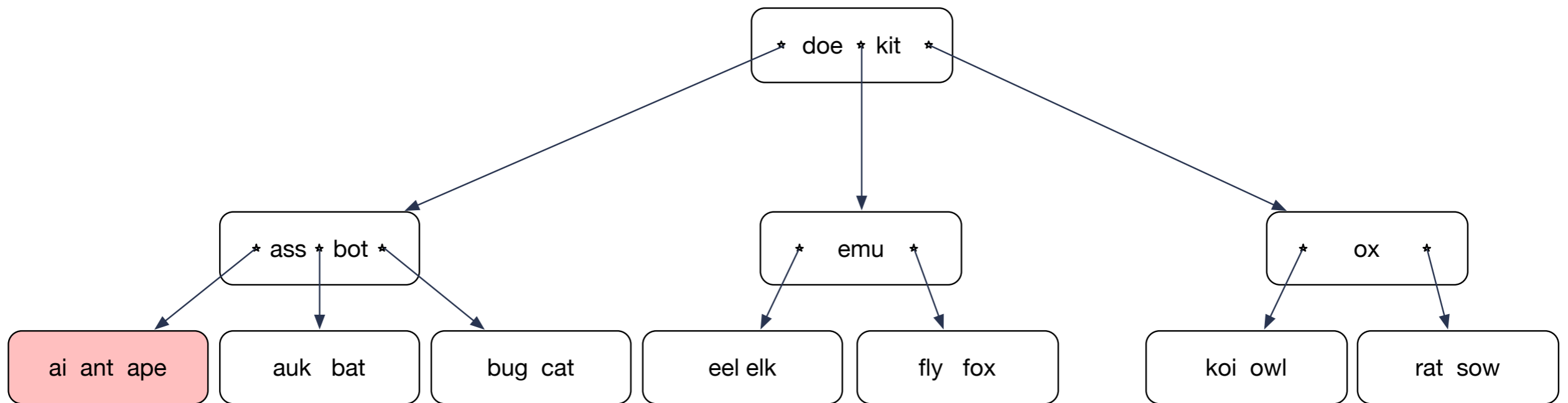


B-tree



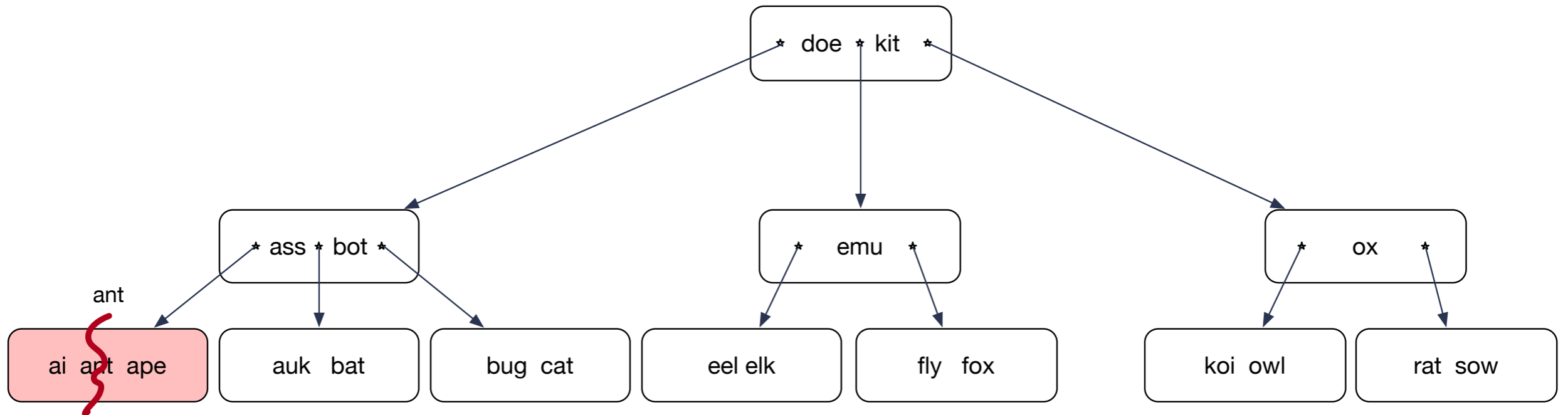
Now insert "ai"

B-tree



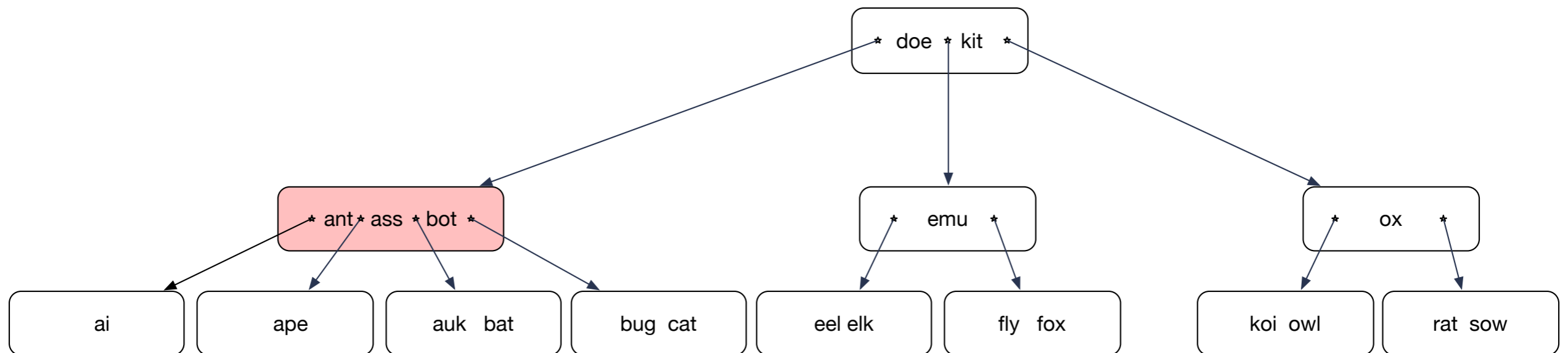
Insert creates an overflowing node
Only one neighboring sibling, but that one is full
Split!

B-tree



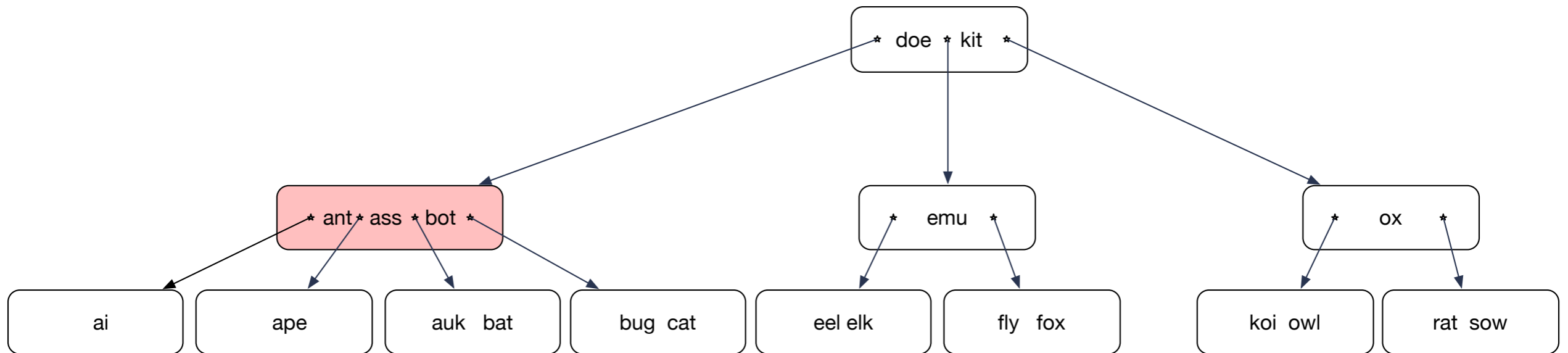
Middle key moves up

B-tree



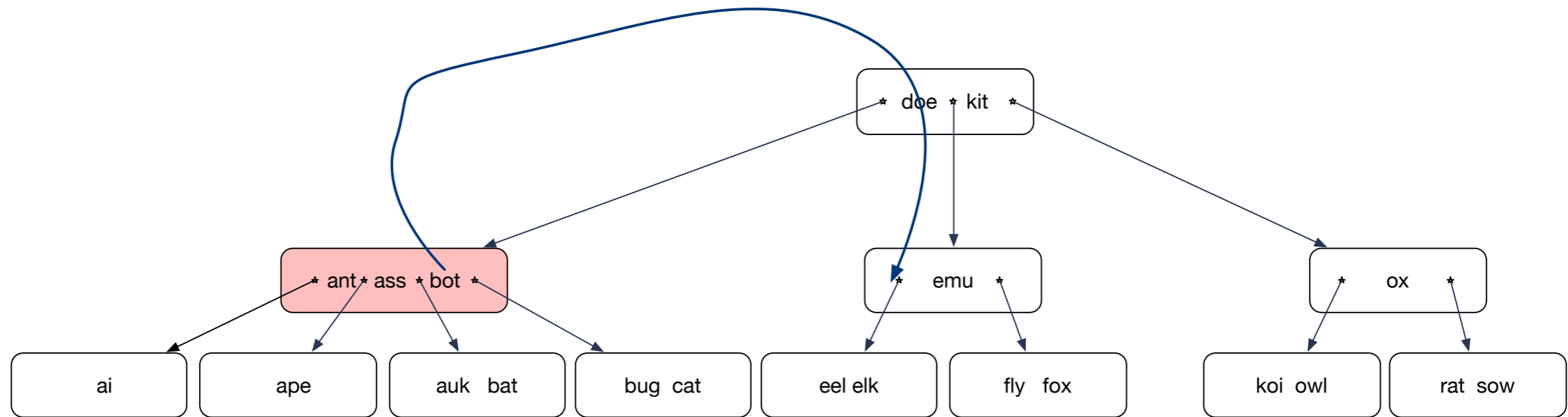
**Unfortunately, this gives another overflow
But this node has a right sibling not at full capacity**

B-tree



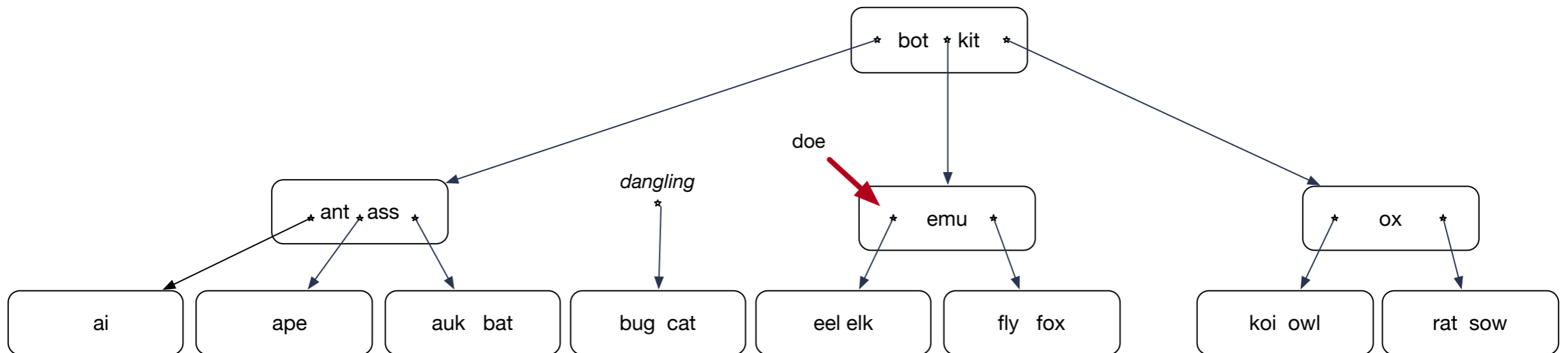
Right rotate:
Move “bot” up
Move “doe” down
Reattach nodes

B-tree



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

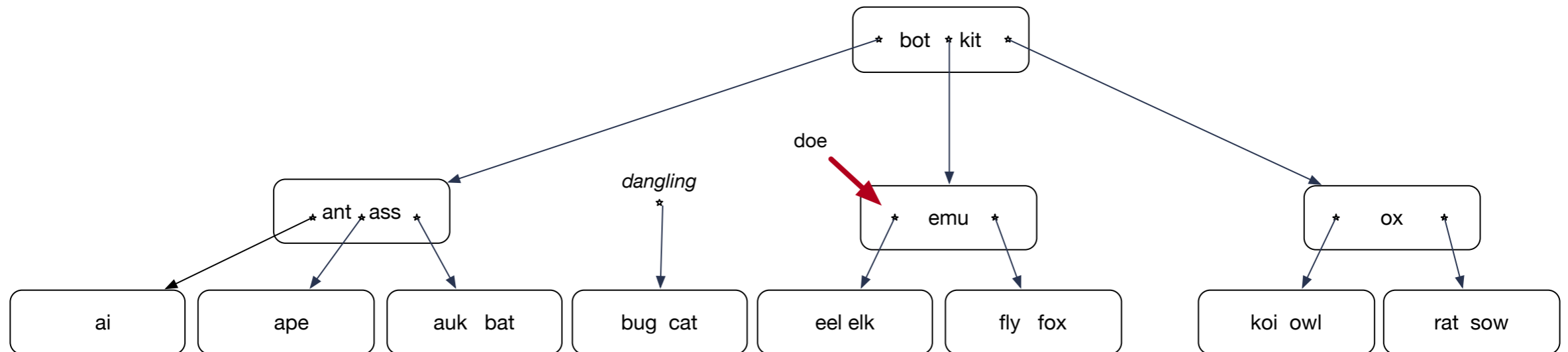
B-tree



**“bot” had moved up
and replaced doe**

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

B-tree



B-tree

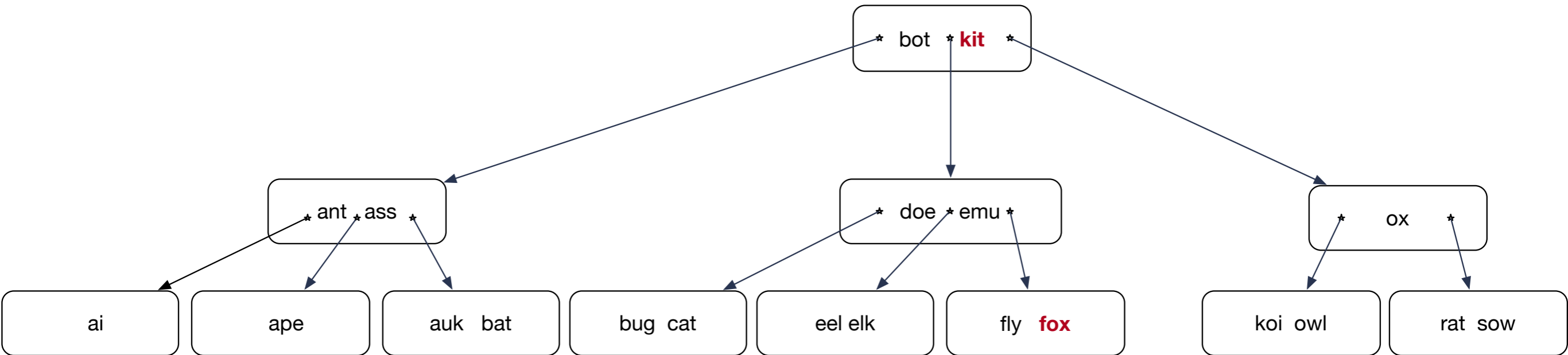
- Deletes
 - Usually restructuring not done because there is no need
 - Underflowing nodes will fill up with new inserts

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”

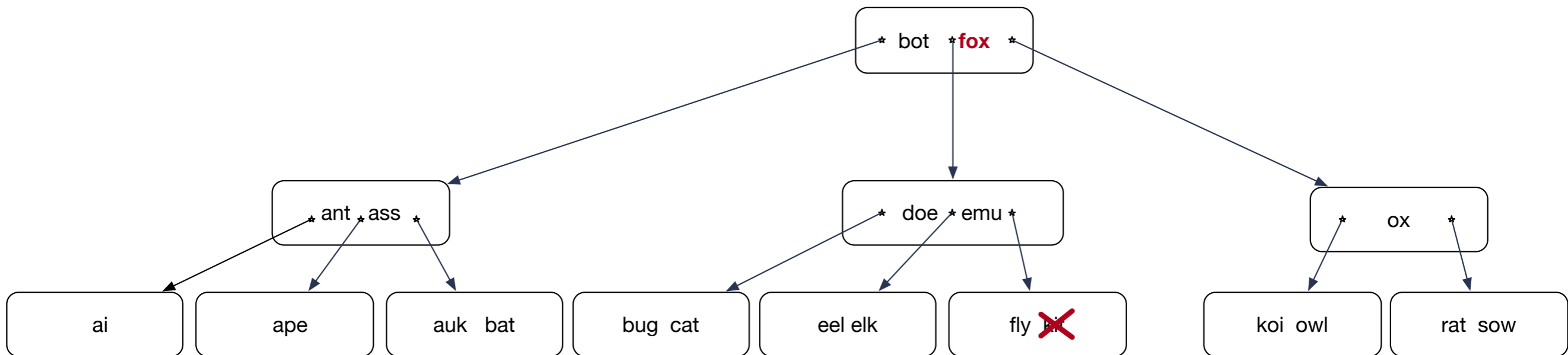


Delete “kit”

“kit” is in an interior node.

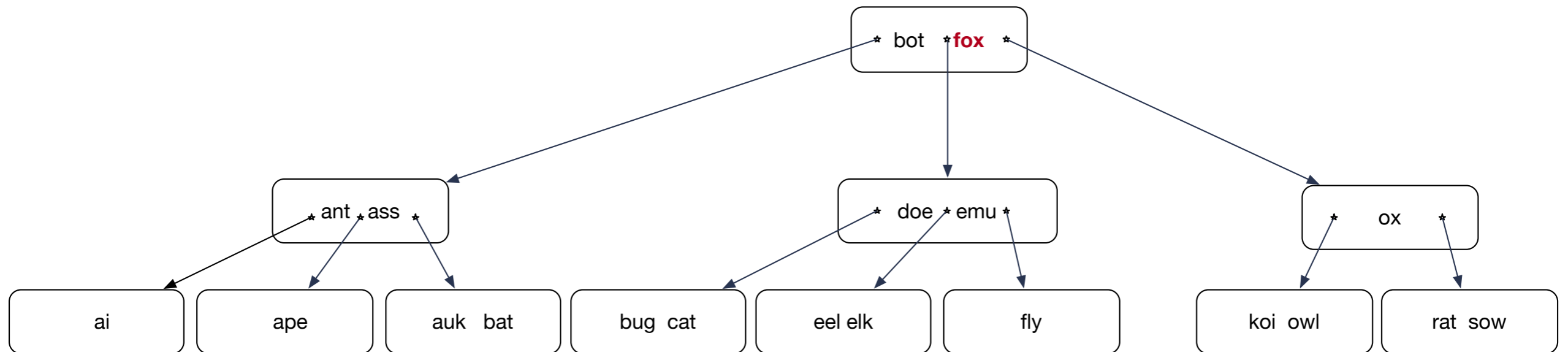
Exchange it with the key in the leaf immediately before “fox”

B-tree



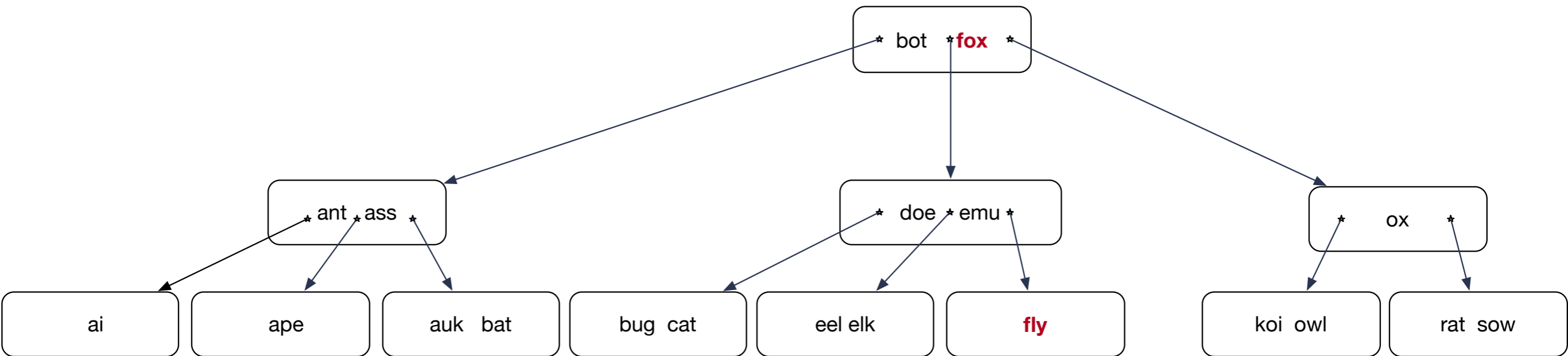
After interchanging “fox” and “kit”, can delete “kit”

B-tree



Now delete "fox"

B-tree

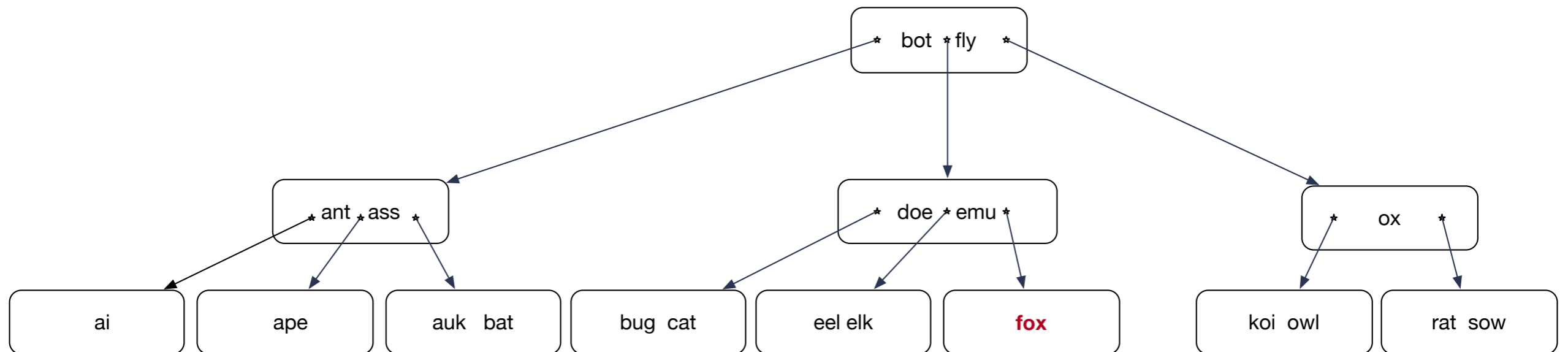


Step 1: Find the key. If it is not in a leaf

Step 2: Determine the key just before it, necessarily in a leaf

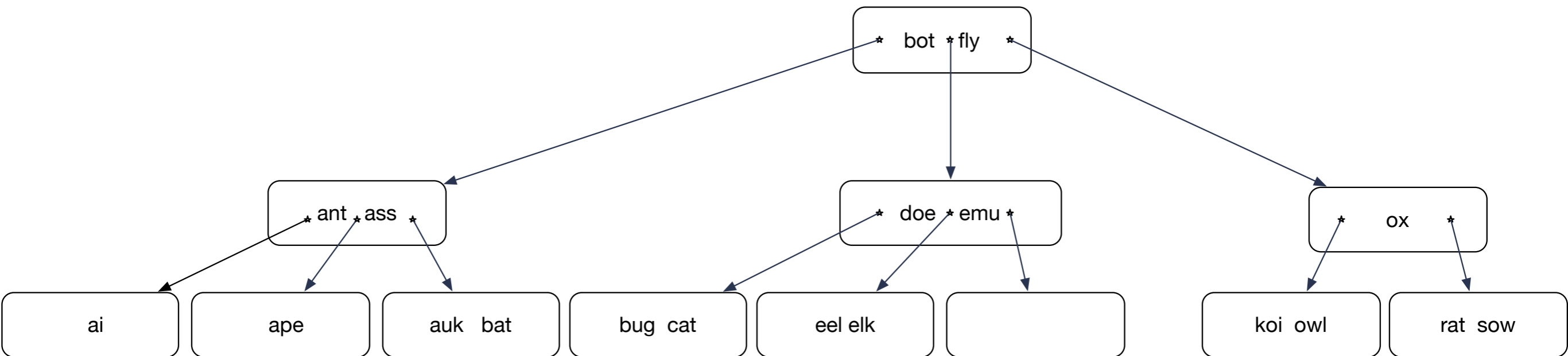
Step 3: Interchange the two keys

B-tree



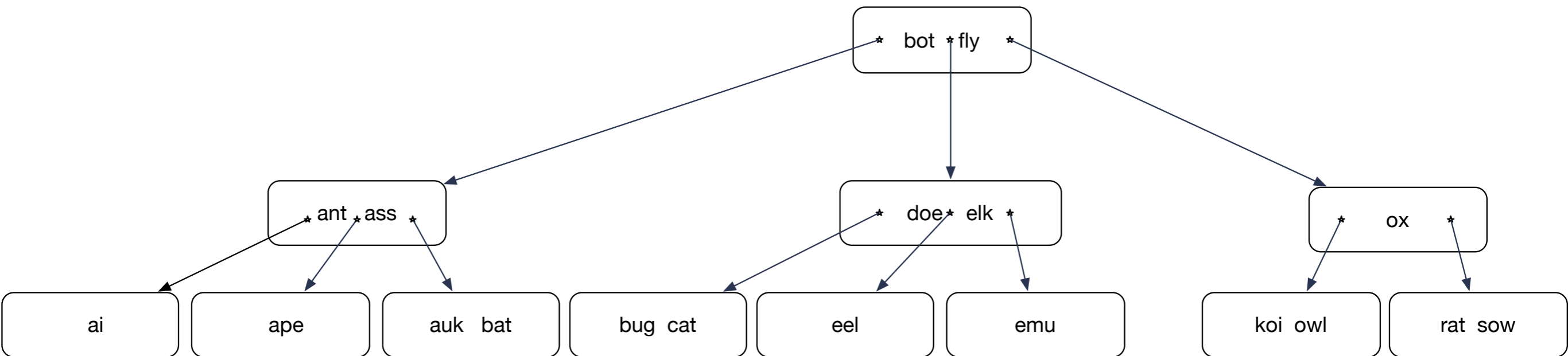
Step 4: Remove the key now from a leaf

B-tree



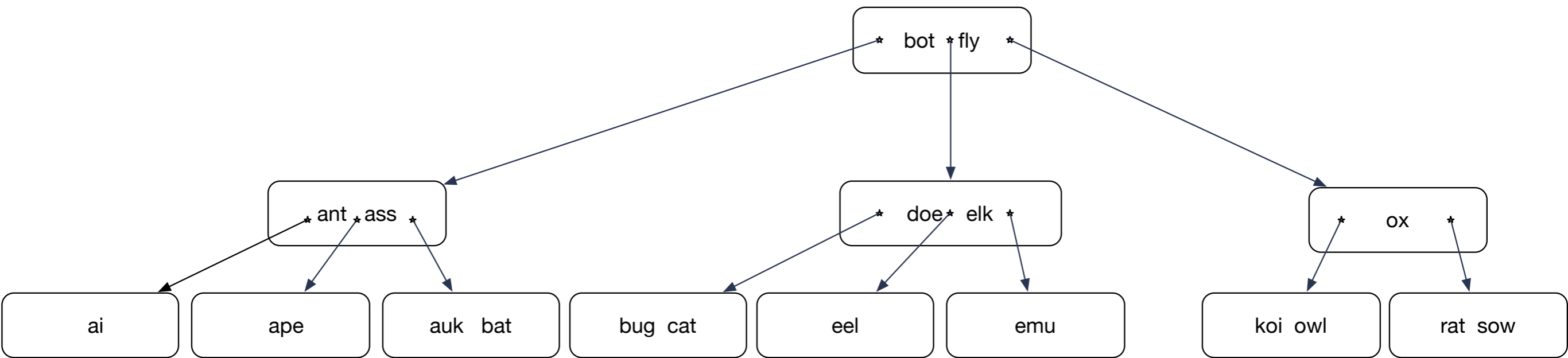
This causes an underflow
Remedy the underflow by right rotating from the sibling

B-tree



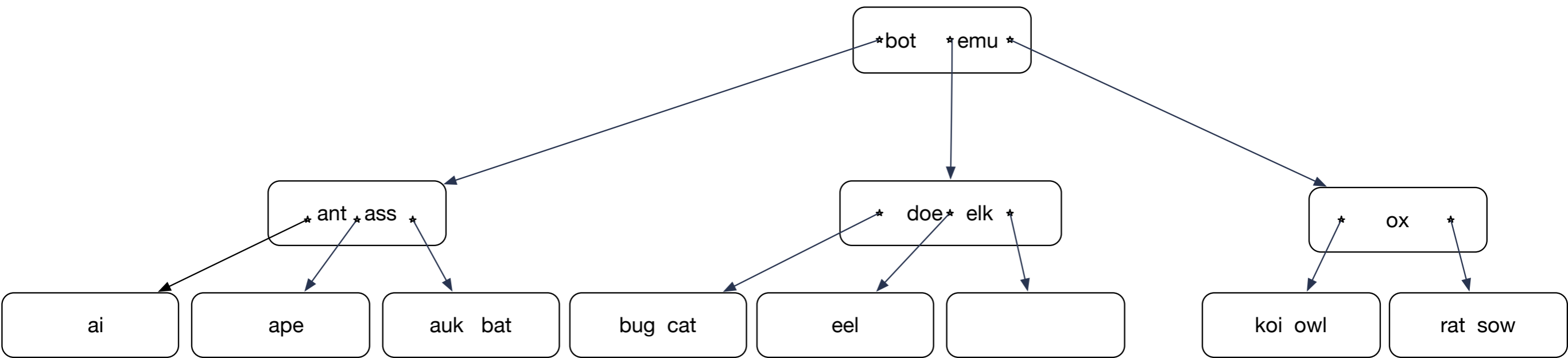
Everything is now in order

B-tree



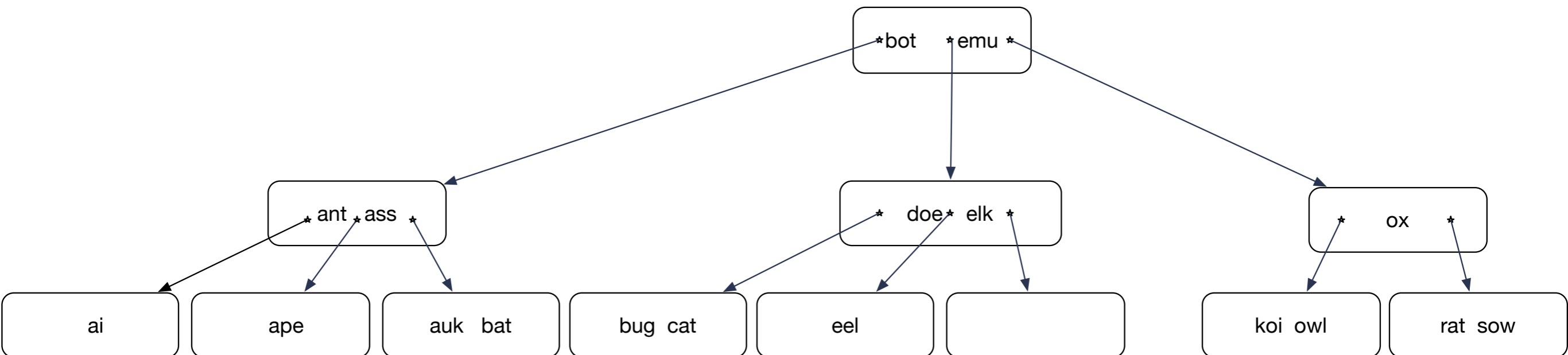
Now delete fly

B-tree



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

B-tree

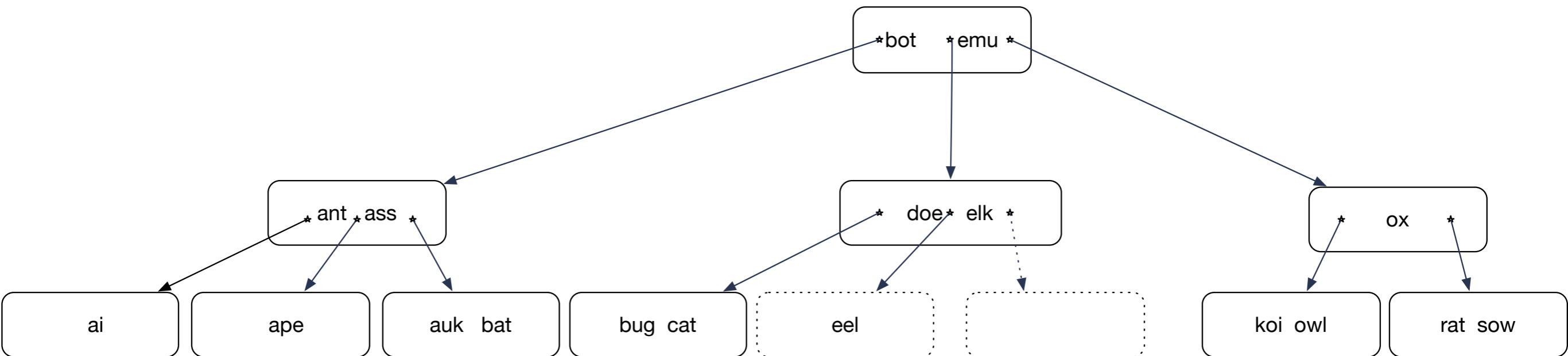


Cannot left-rotate: There is no left sibling

Cannot right-rotate: The right sibling has only one key

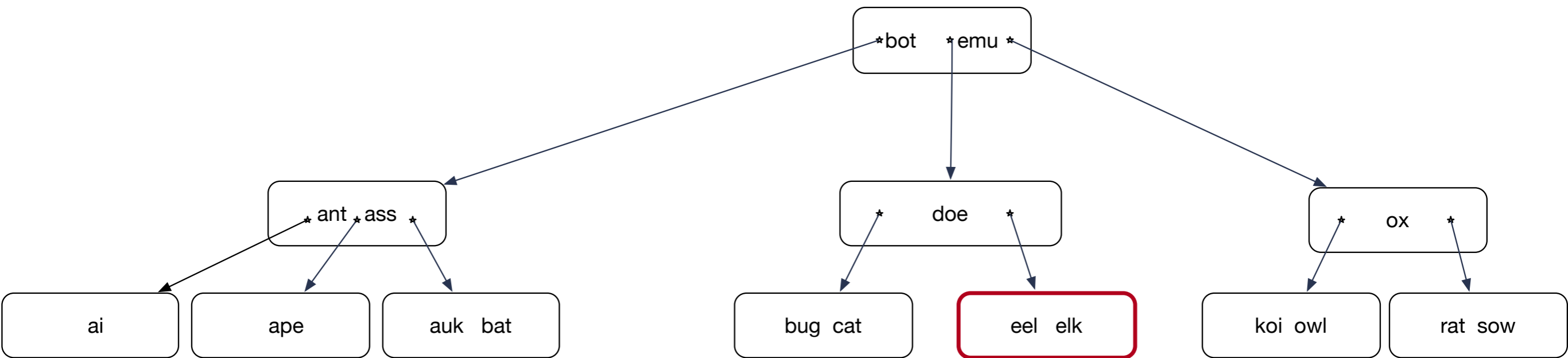
Need to merge: Combine the two nodes by bringing down “elk”

B-tree

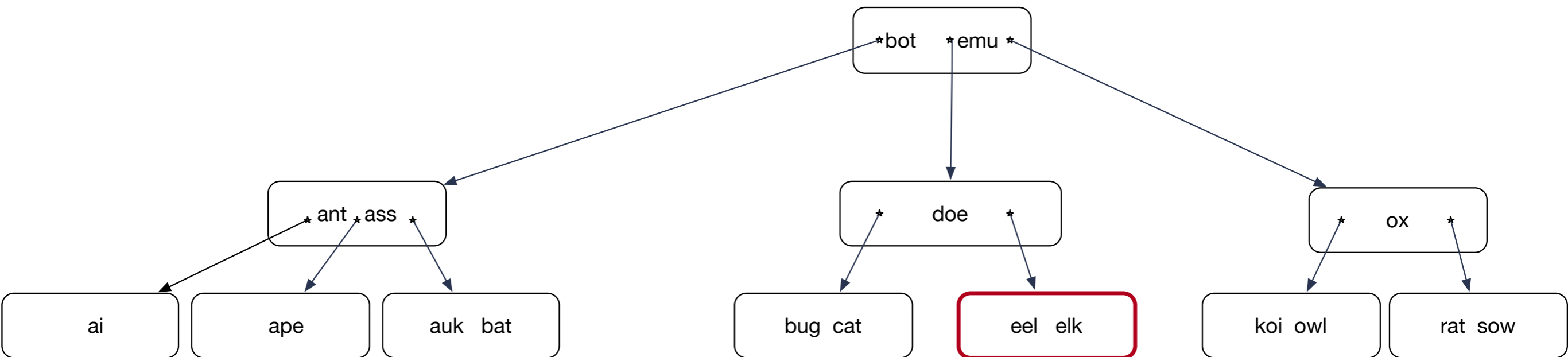


**We can merge the two nodes because
the number of keys combined is less than $2k$**

B-tree

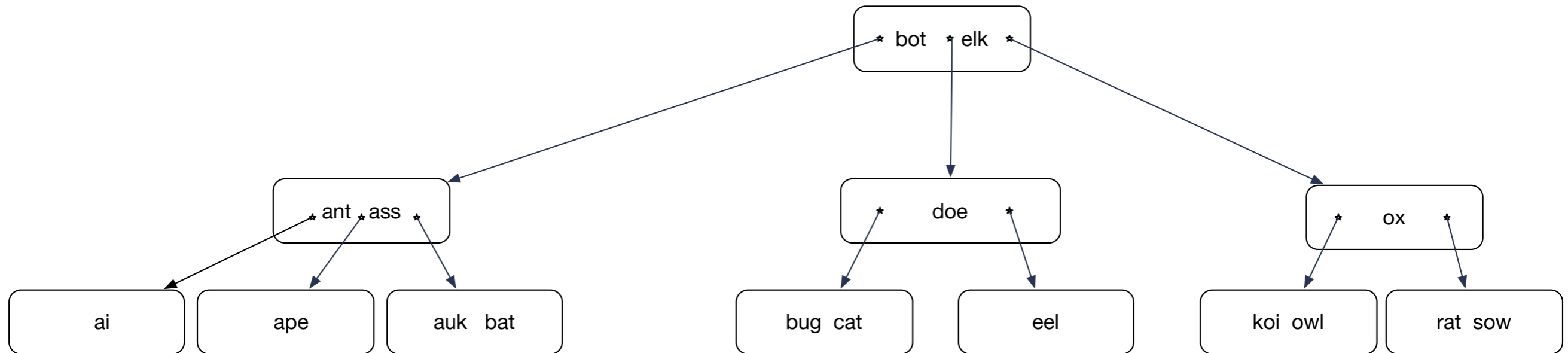


B-tree



Delete “emu”

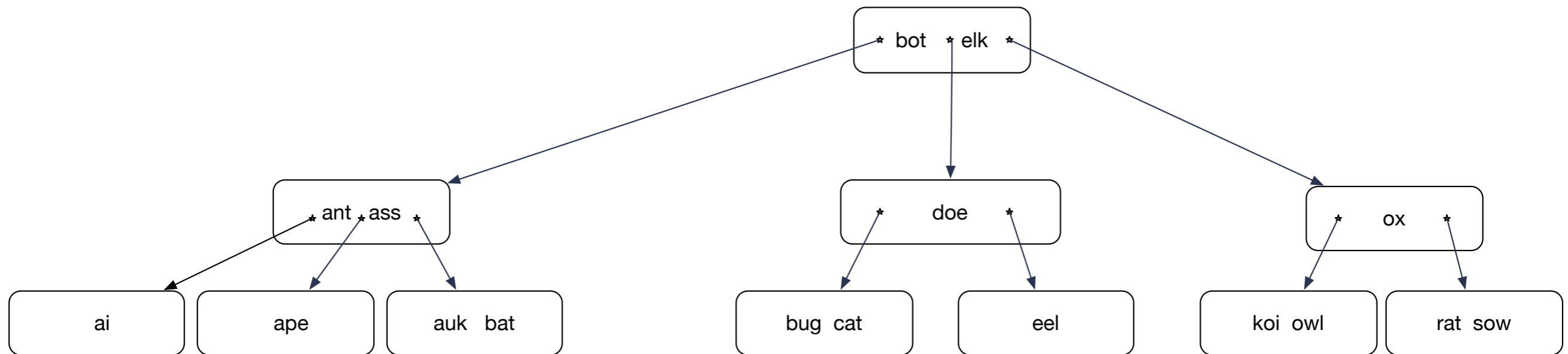
B-tree



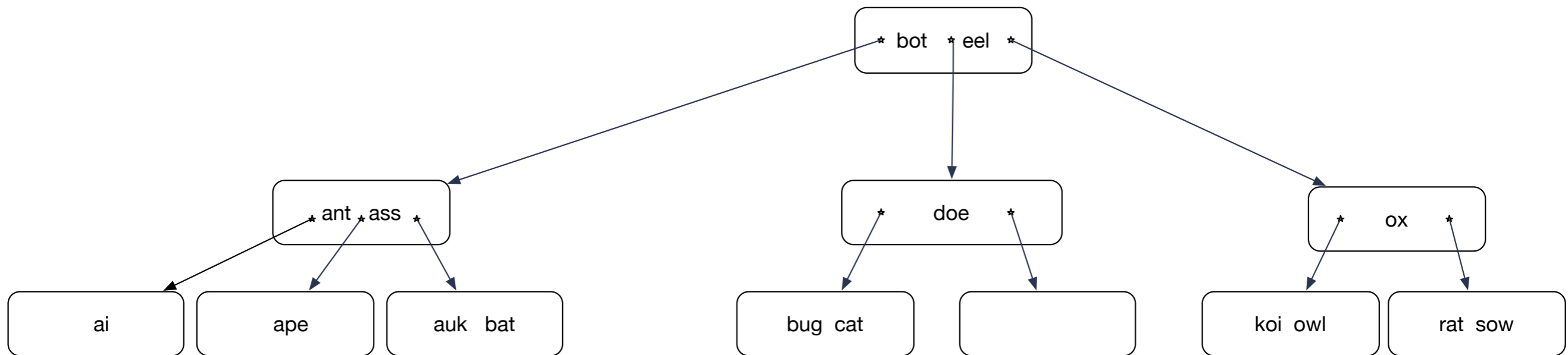
Switch predecessor, then delete from node

B-tree

Now delete "elk"

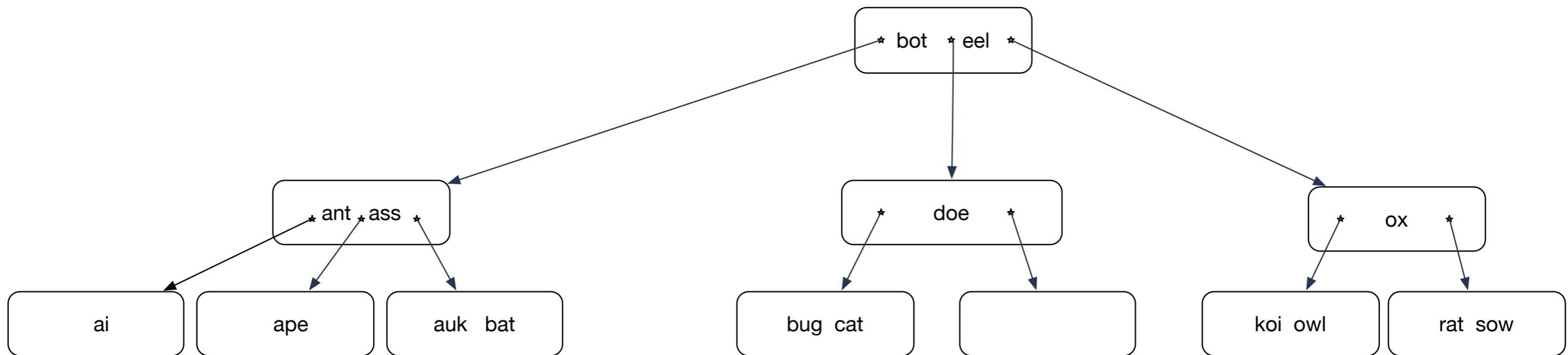


B-tree



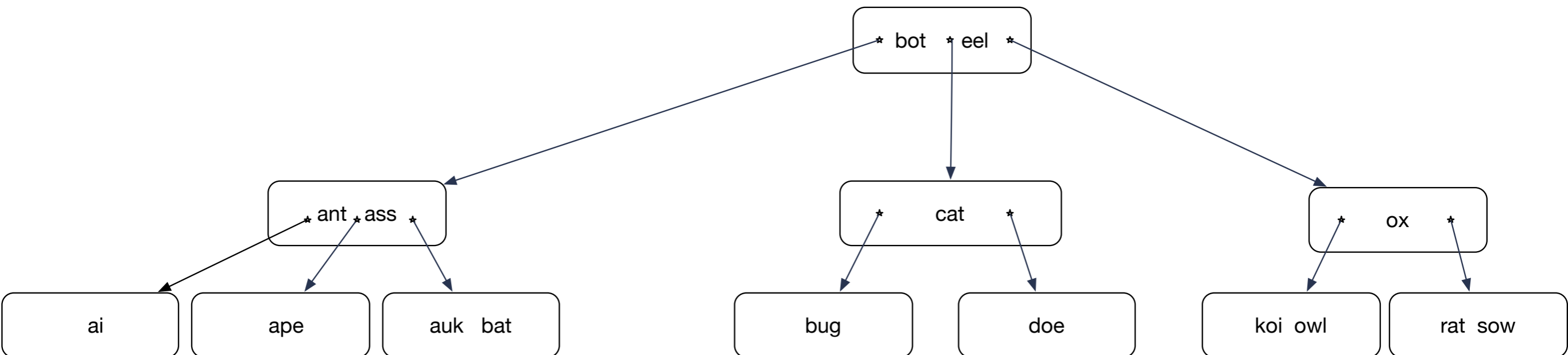
Results in an underflow

B-tree



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

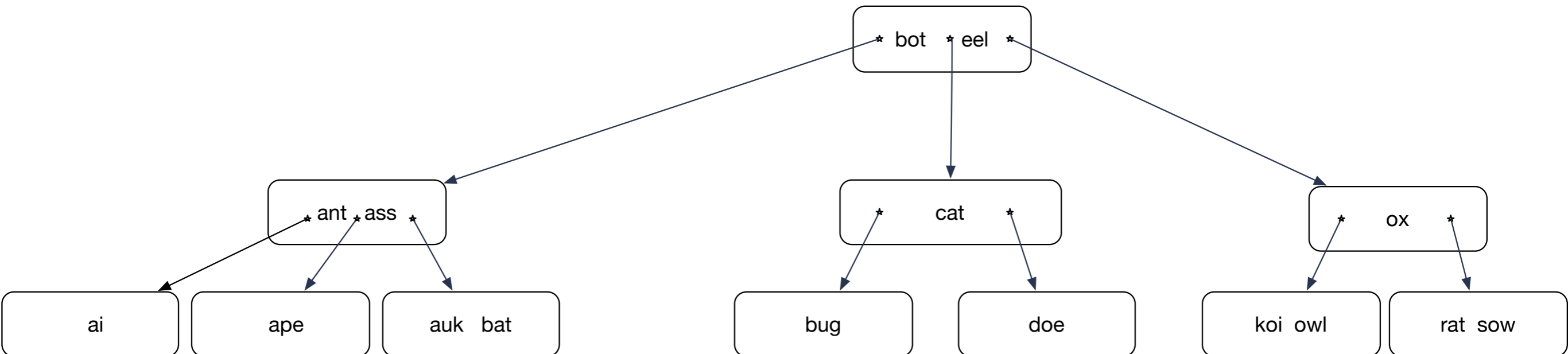
B-tree



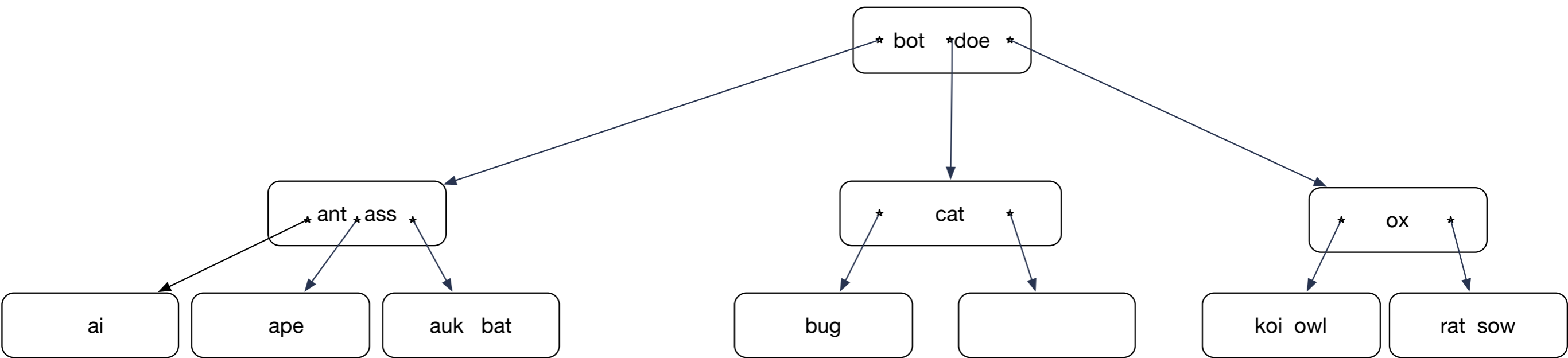
Result after left-rotation

B-tree

“Now delete “eel”

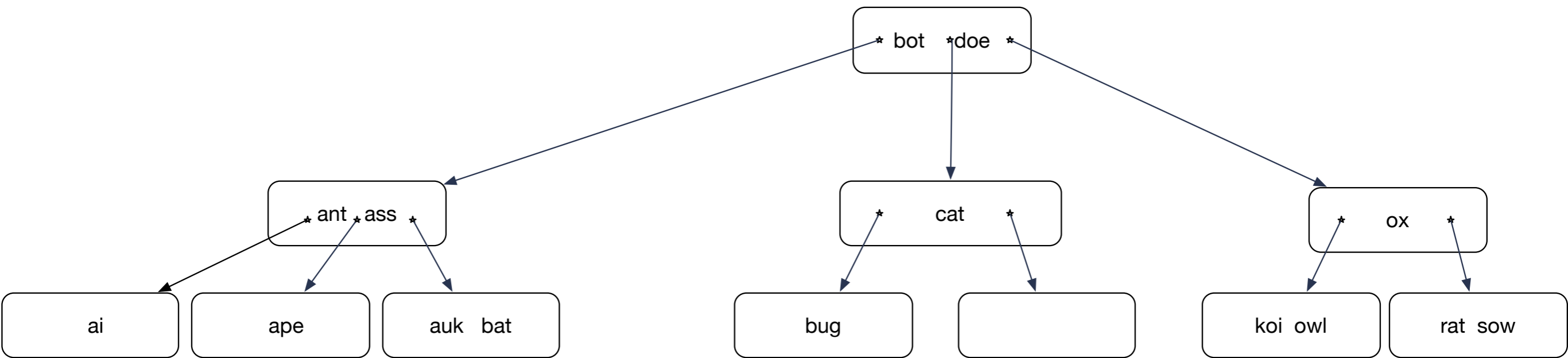


B-tree



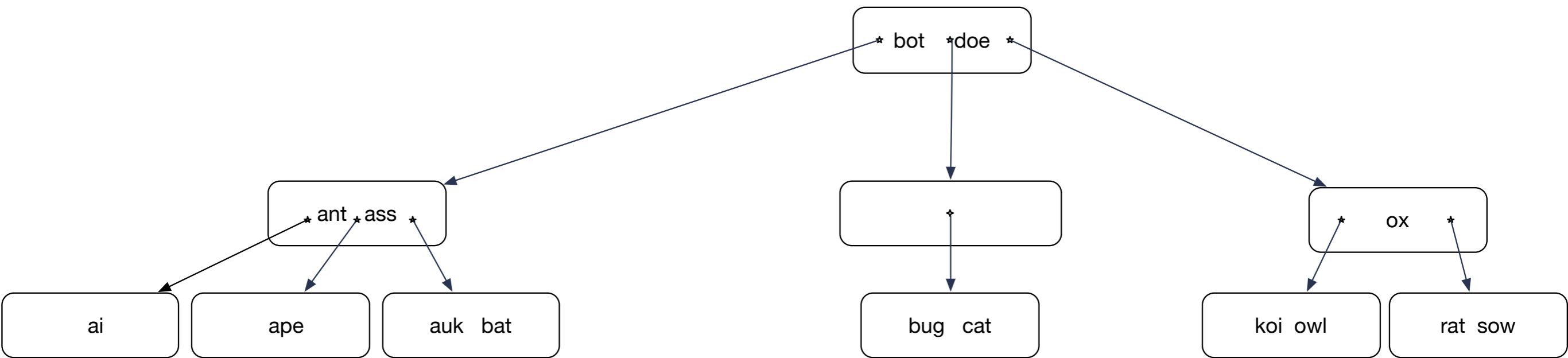
**Interchange “eel” with its predecessor
Delete “eel” from leaf:
Underflow**

B-tree



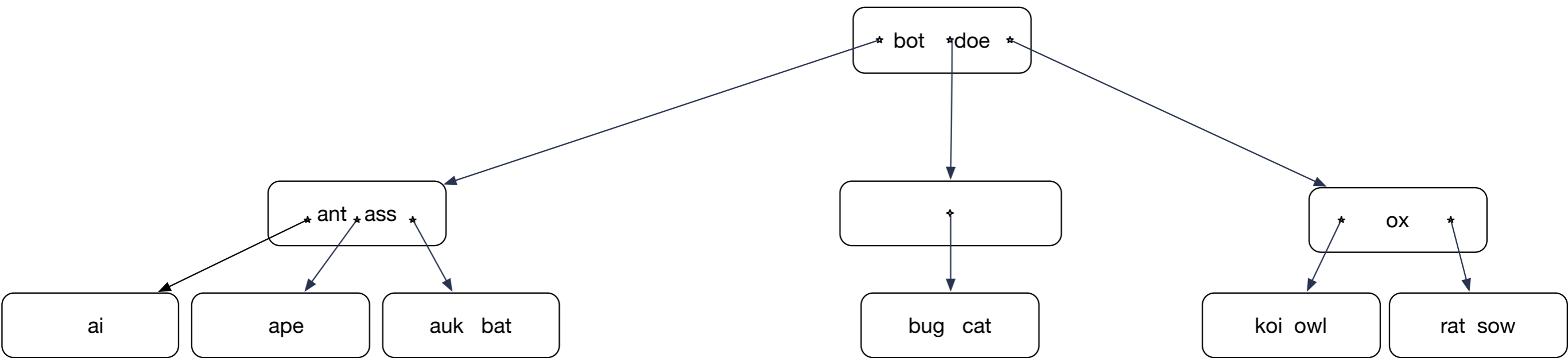
Need to merge

B-tree



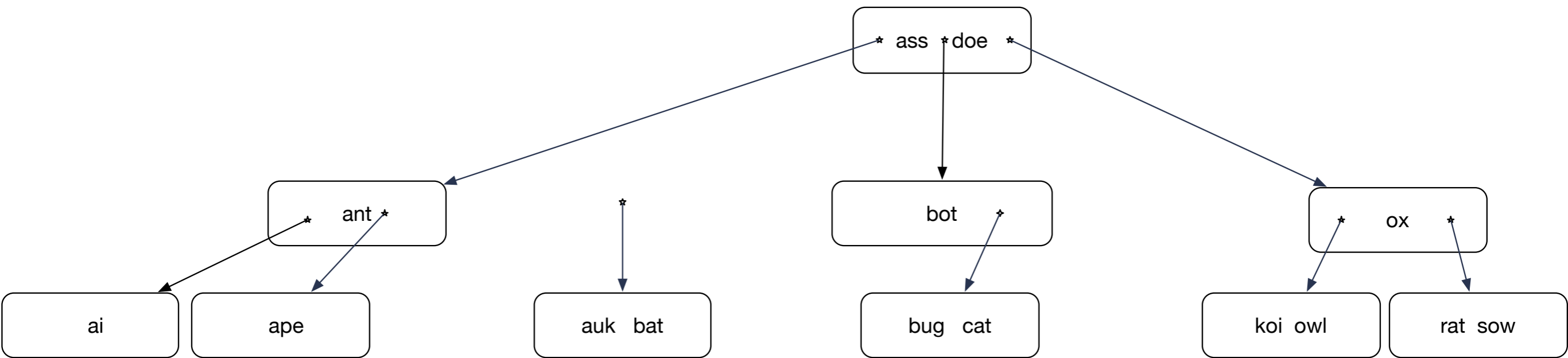
**Merge results in another underflow
Use right rotate
(though merge with right sibling
is possible)**

B-tree



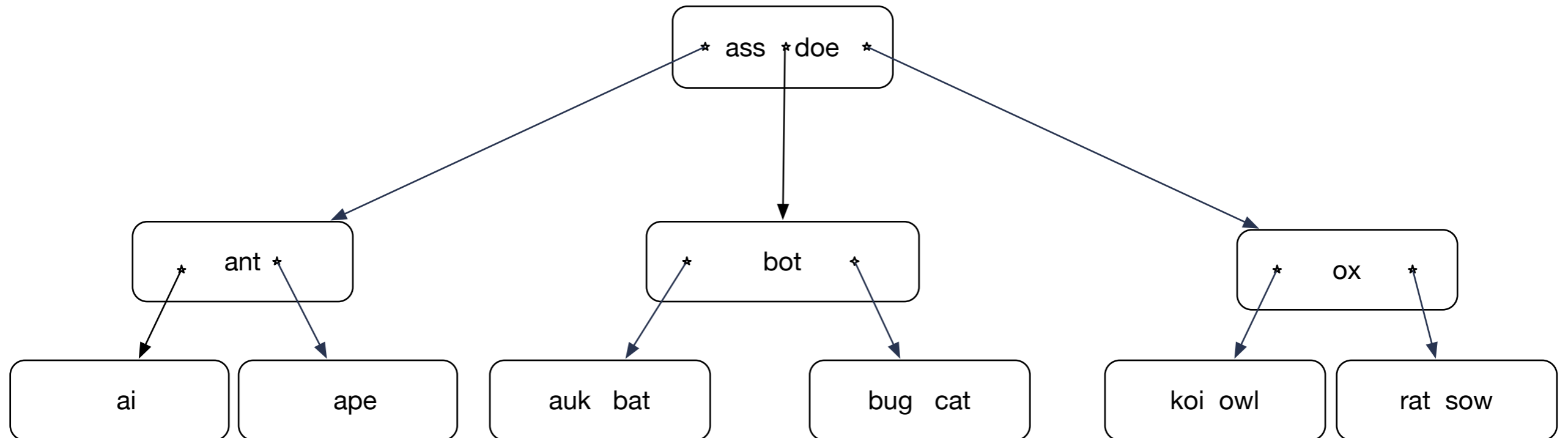
**“ass” goes up, “bot” goes down
One node is reattached**

B-tree

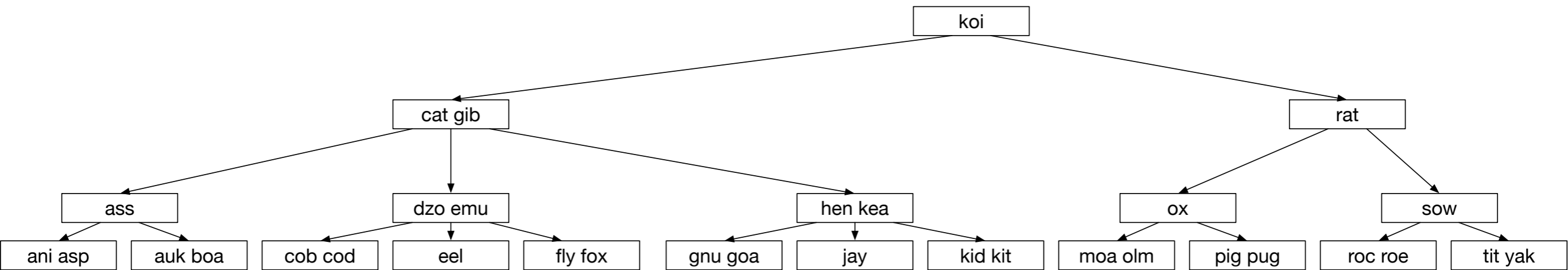


Reattach node

B-tree

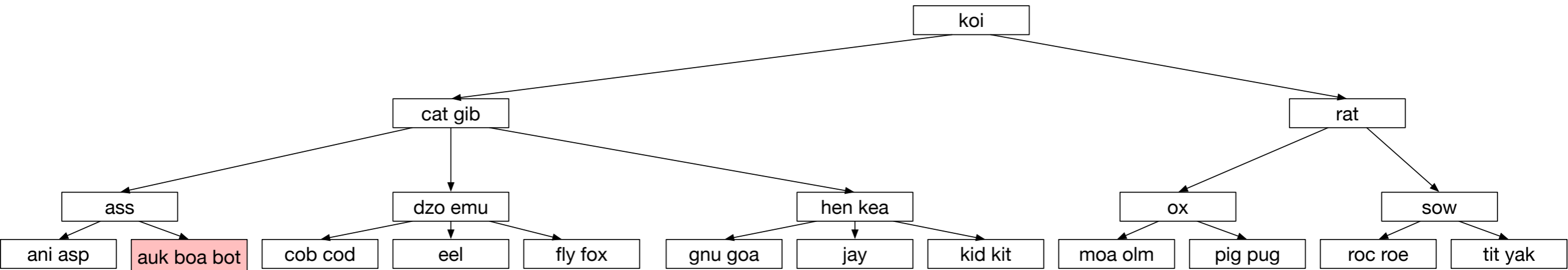


Activities



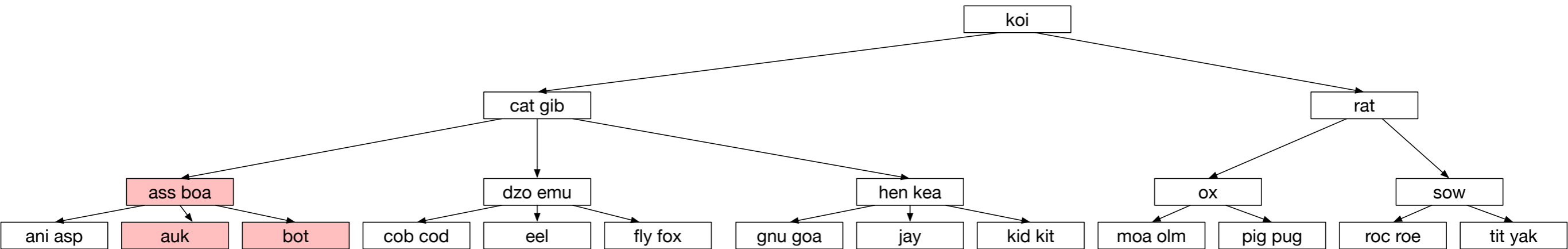
- Insert bot

Activities



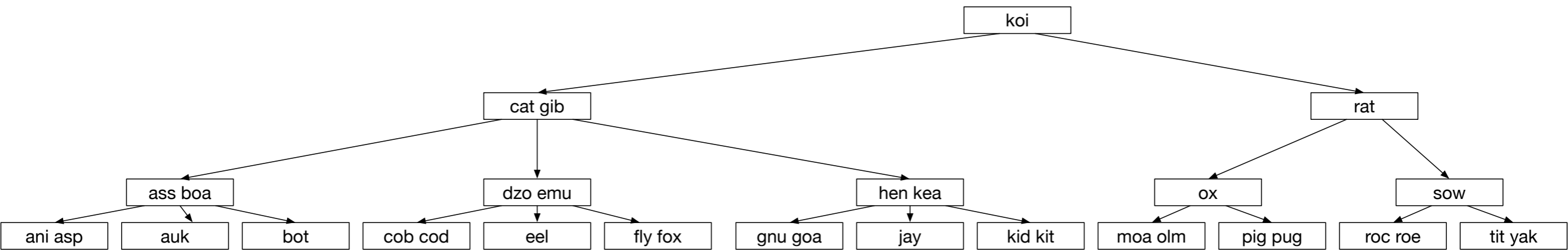
- Inserting bot in the leaf
 - Leads to an overflow

Activities



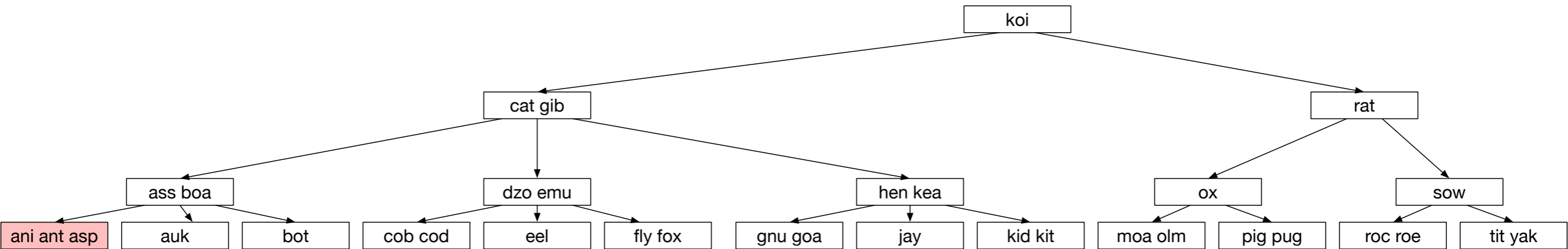
- Since rotates are not possible:
- Split: Middle node goes up, left and right become their own nodes

Activities



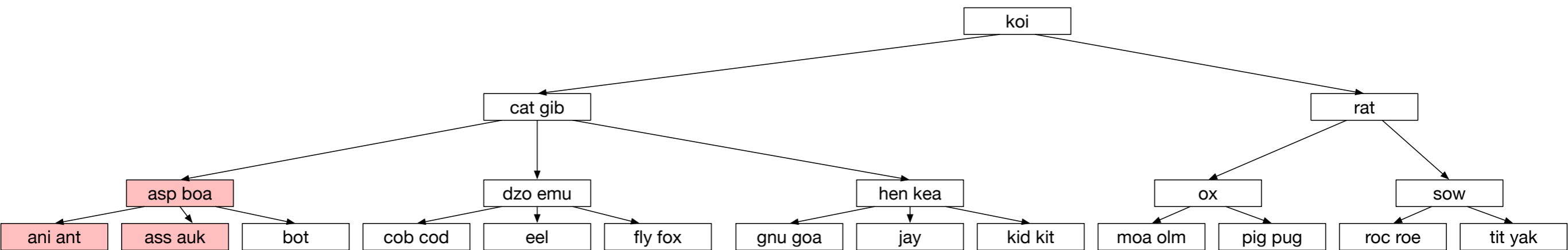
- Insert ant

Activities



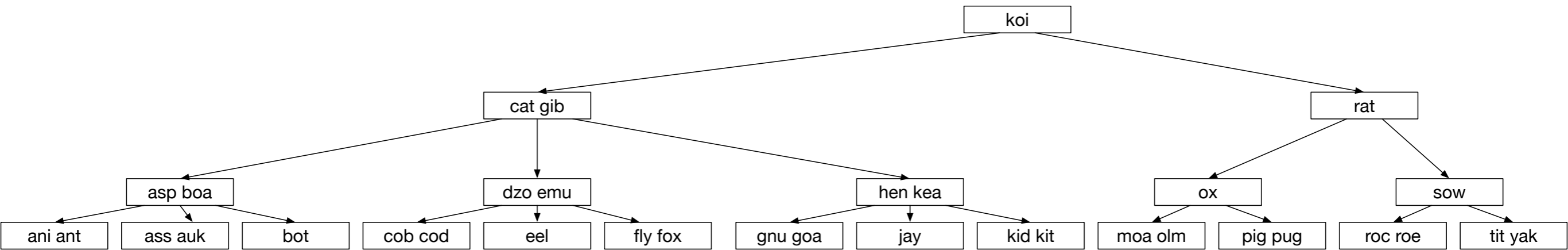
- Insert ant into leaf: Leaf is now at overflow

Activities



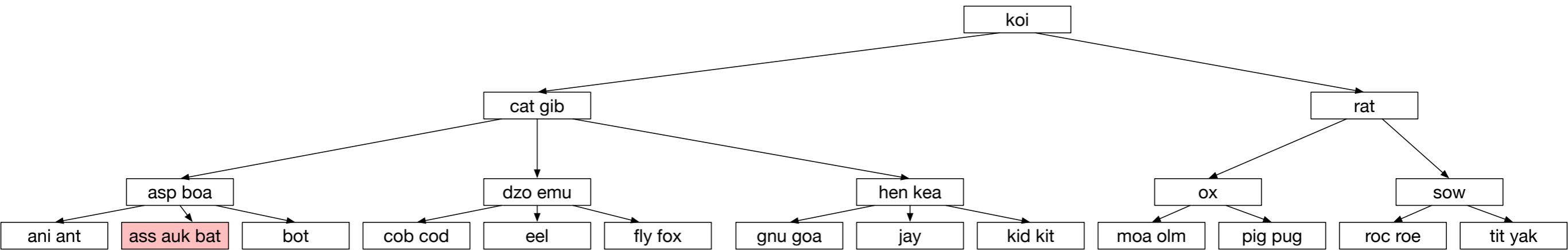
- Right rotate: asp goes up, ass goes down

Activities



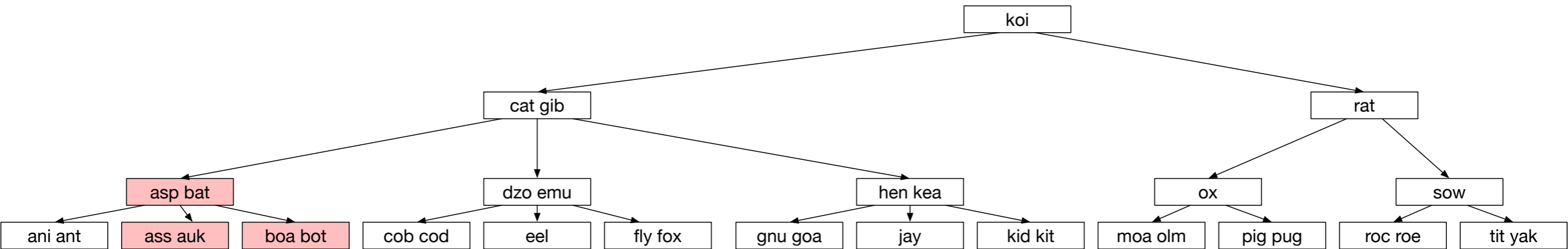
- Insert bat

Activities



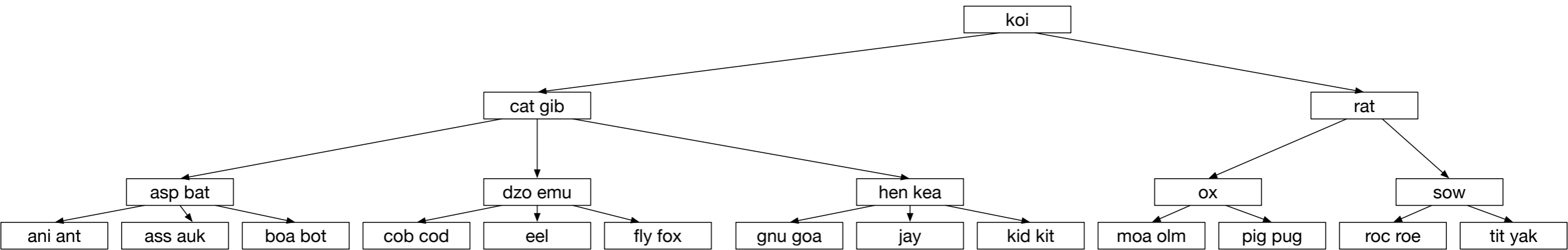
- Insert into leaf: overflow

Activities



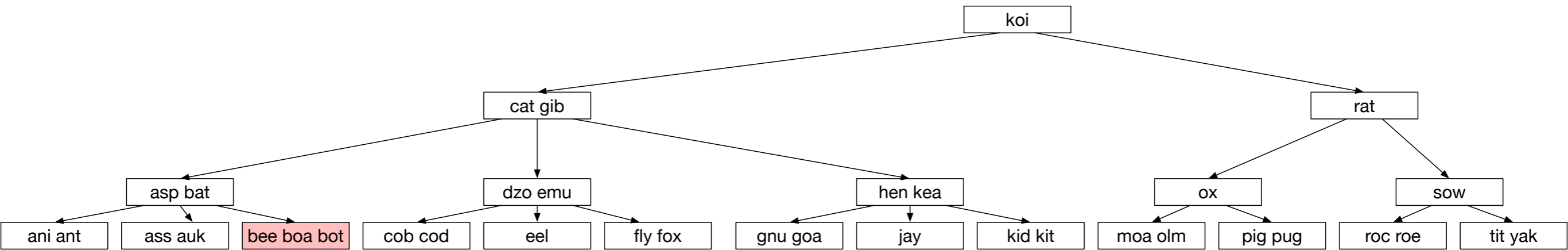
- Right rotate: bat goes up, boa goes down

Activities



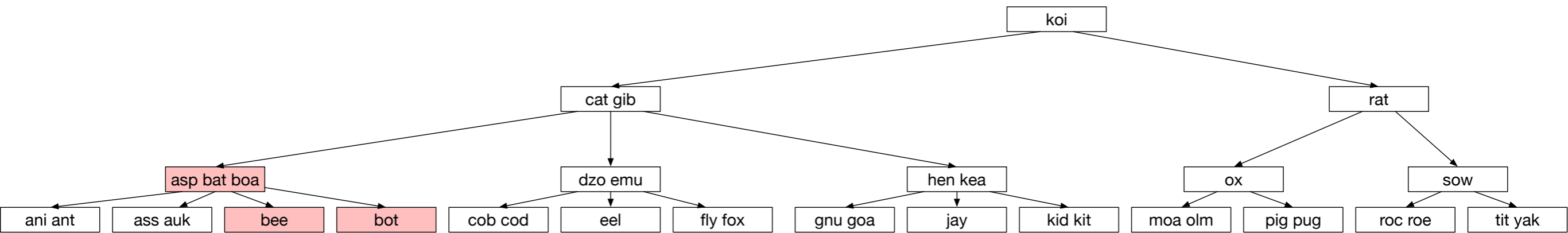
- Insert bee

Activities



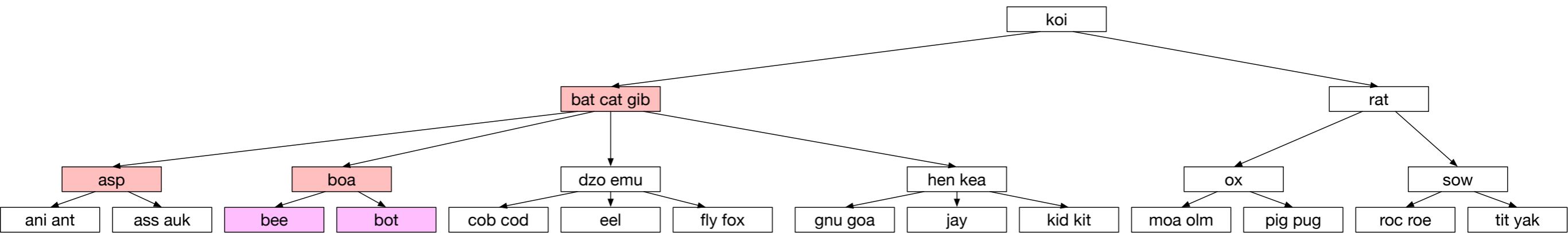
- Insert into leaf

Activities



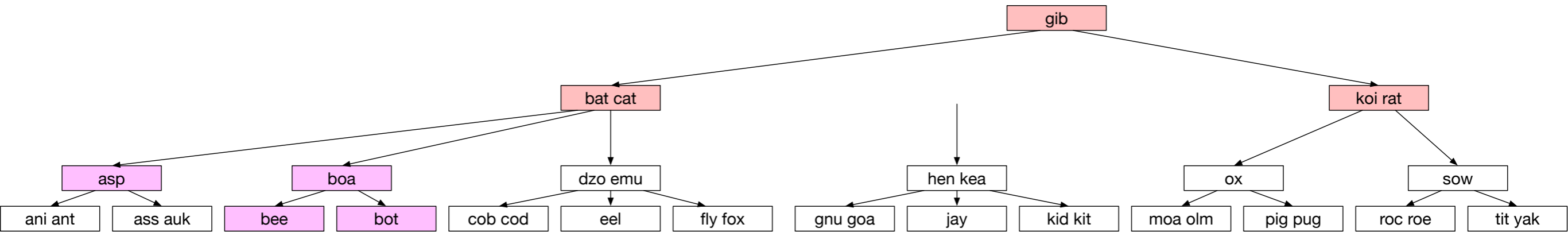
- Need to split: boa goes up, bee and bot get their own nodes
- Now there is an overflow at a higher level
- Rotate is not possible

Activities



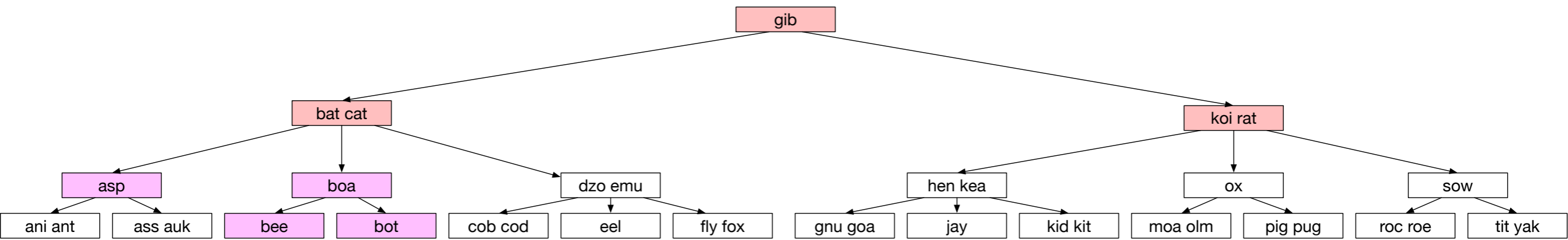
- Now we have yet another overflow
- Need to do a right rotate: gib goes up and koi goes down

Activities



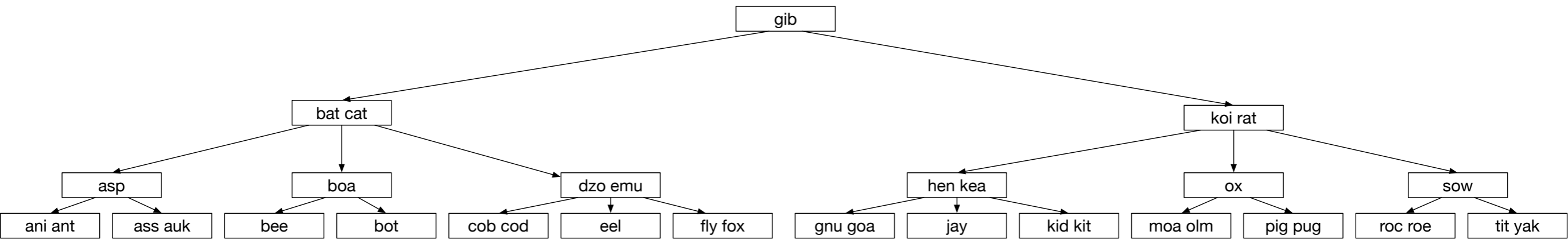
- Reattach the dangling pointer

Activities



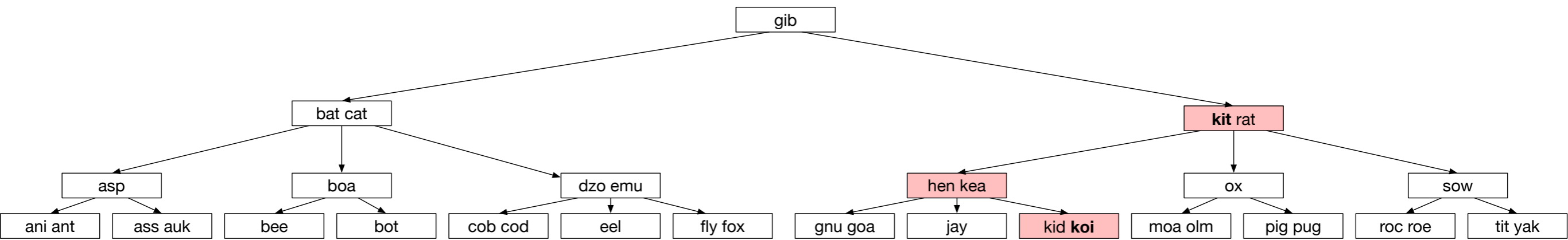
- Final result

Activities



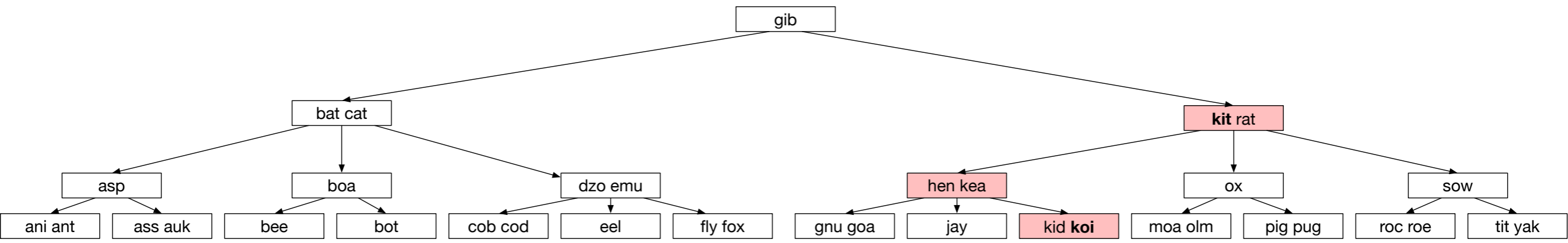
- Delete koi

Activities



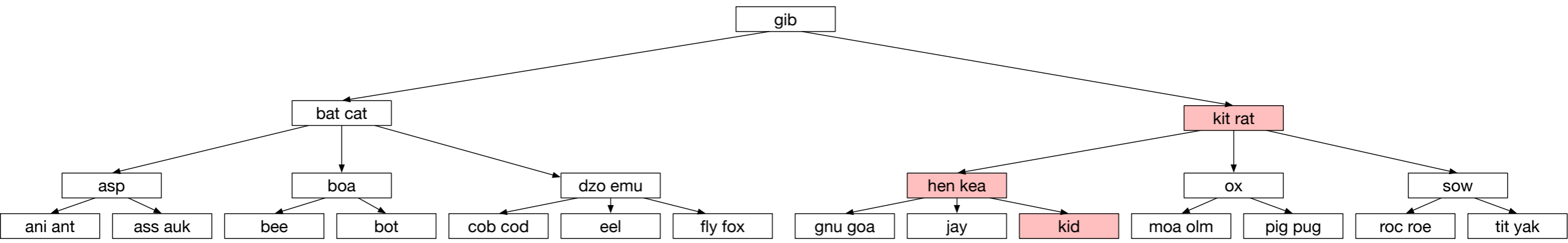
- Switch with predecessor

Activities



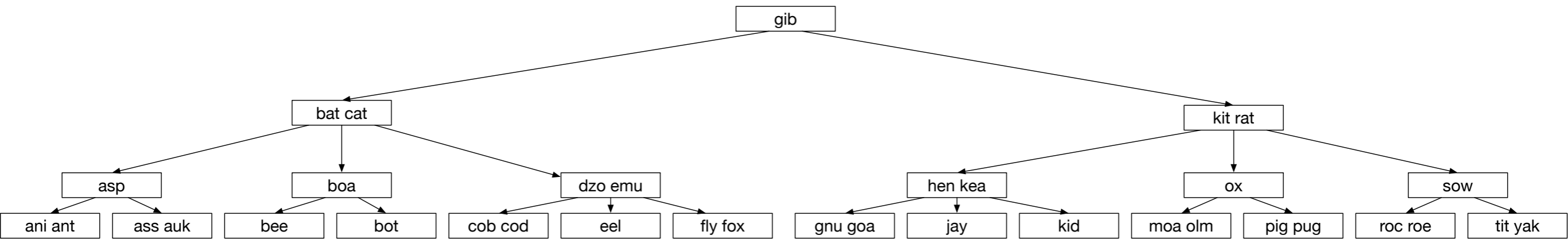
- Delete koi from leave

Activities



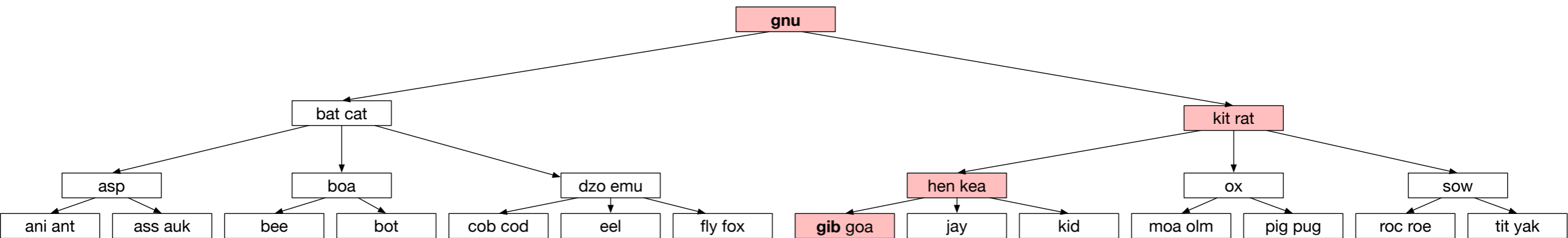
- Done

Activities



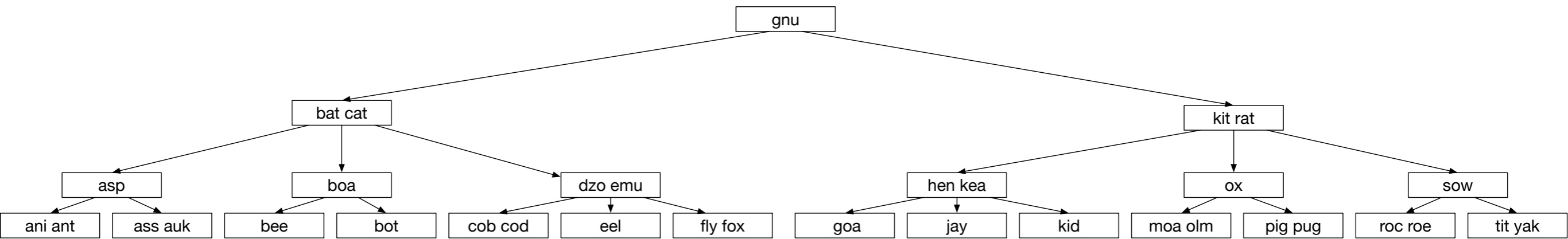
- Delete gib using successor

Activities



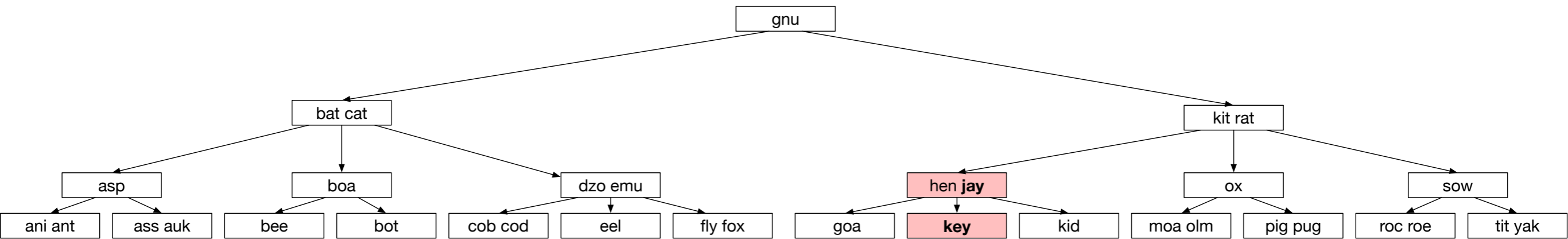
- After swap, delete gib from leaf

Activities



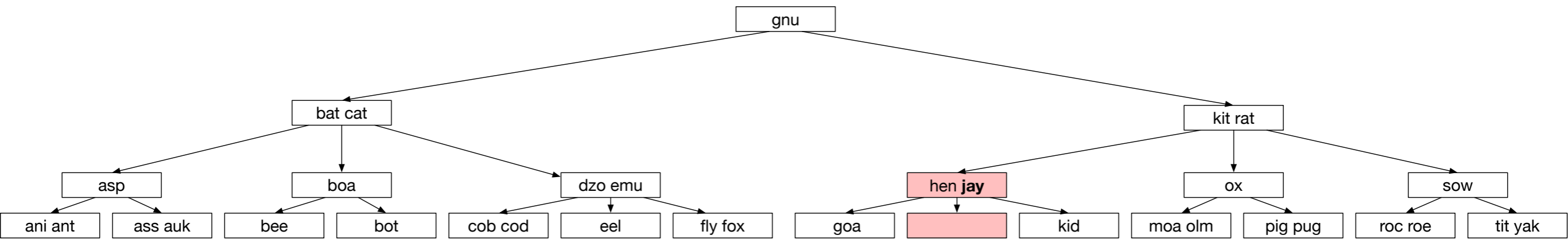
- Delete kea with predecessor

Activities



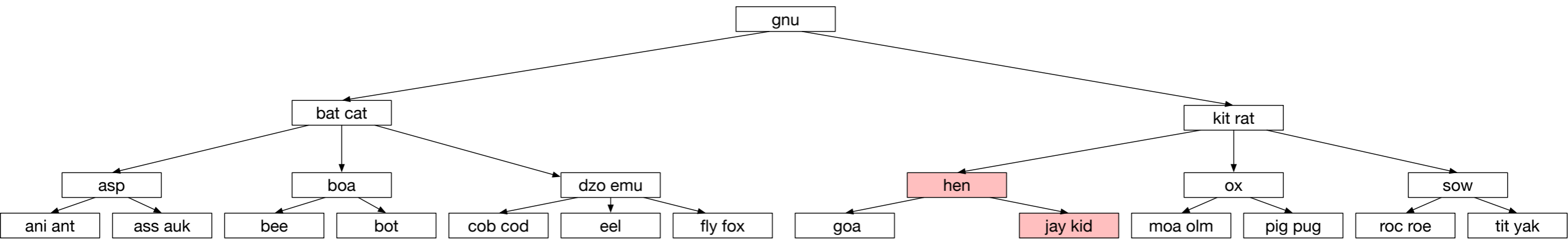
- Delete from leaf

Activities

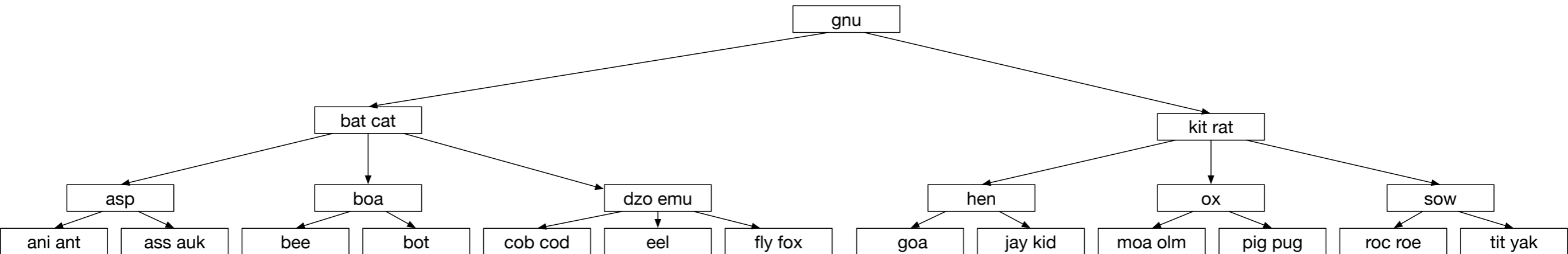


- Underflow:
- Rotate is not possible
- Merge: Two candidates, merge right

Activities

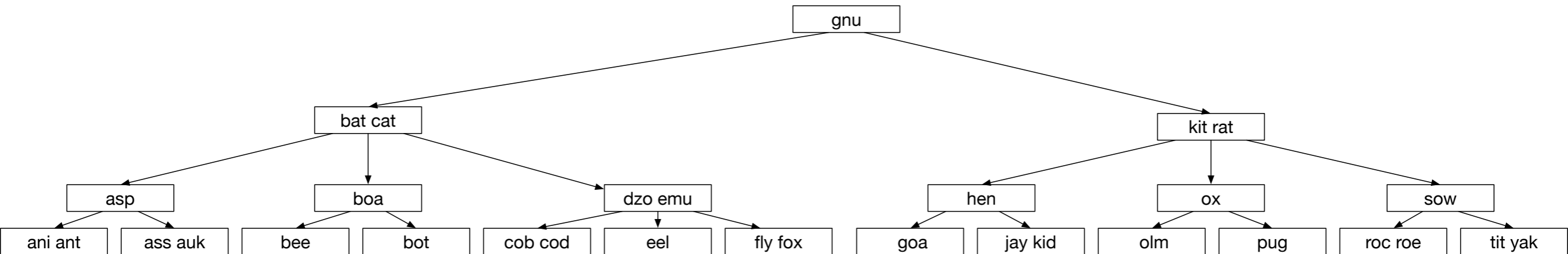


Activities



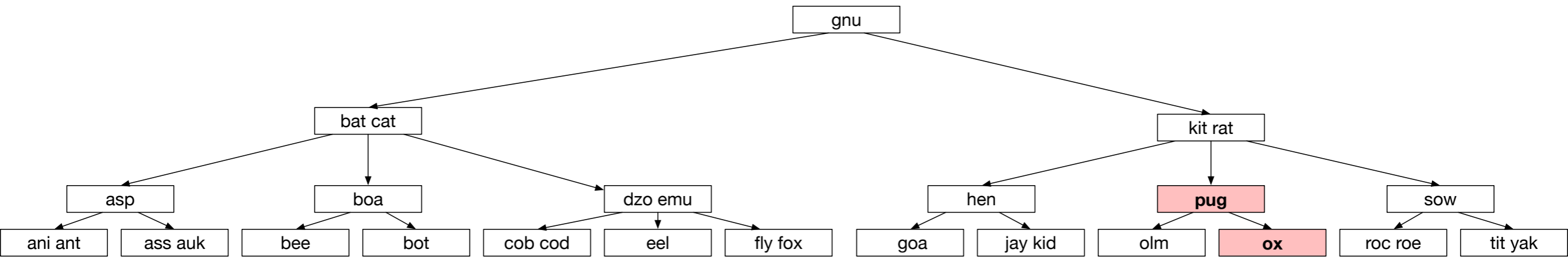
- Delete moa and pig

Activities



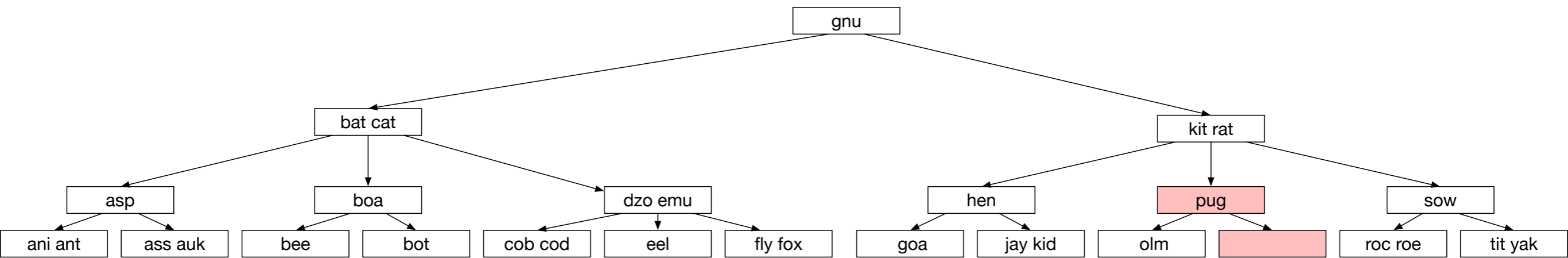
- Delete ox with successor

Activities



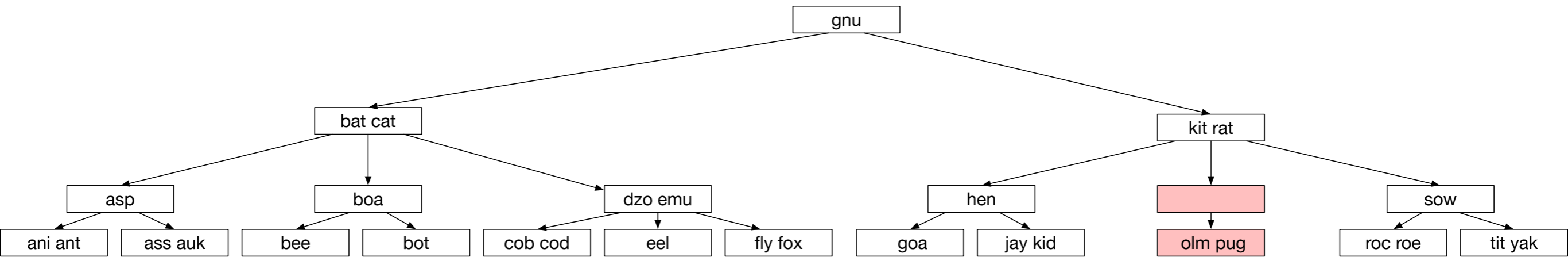
- Now delete from leaf

Activities



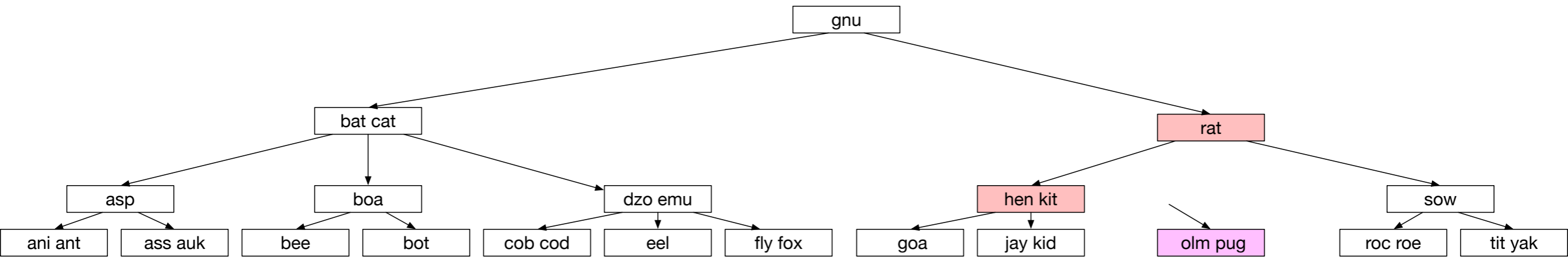
- Underflow: need to do a merge

Activities



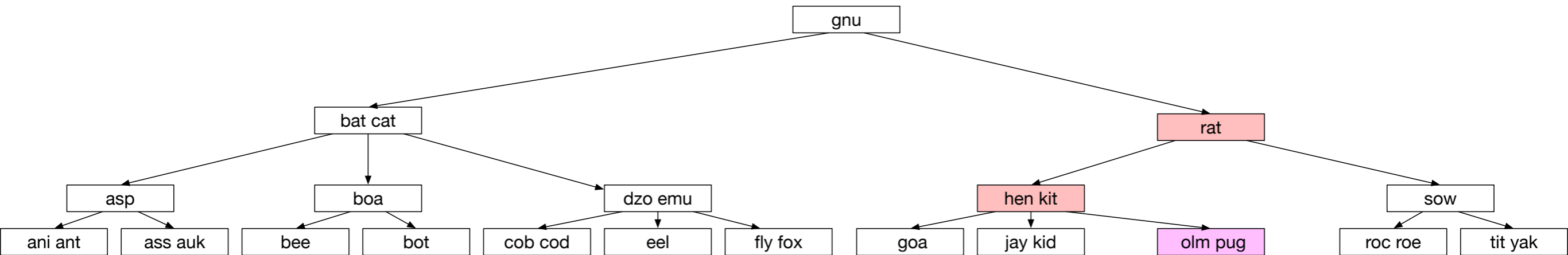
- Now the leaf is fine, but the parent has an underflow
- Cannot rotate, so we need to do another merge
- This time we merge left

Activities

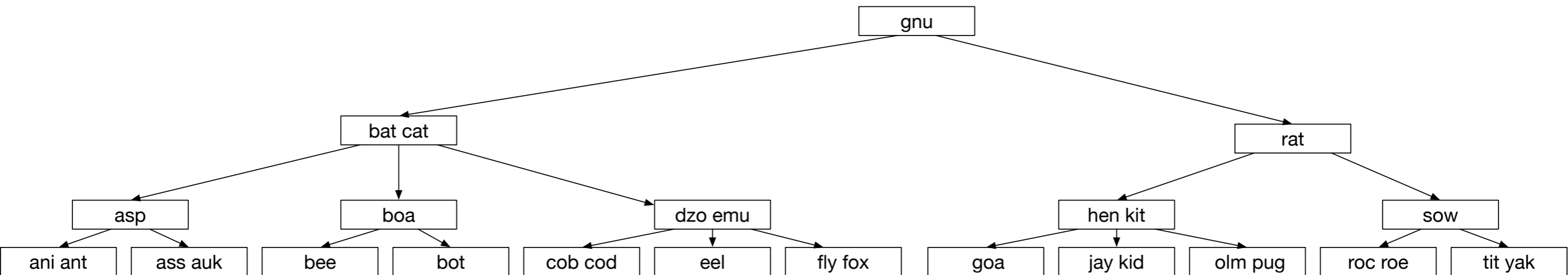


- Need to re-attach dangling node pointer

Activities

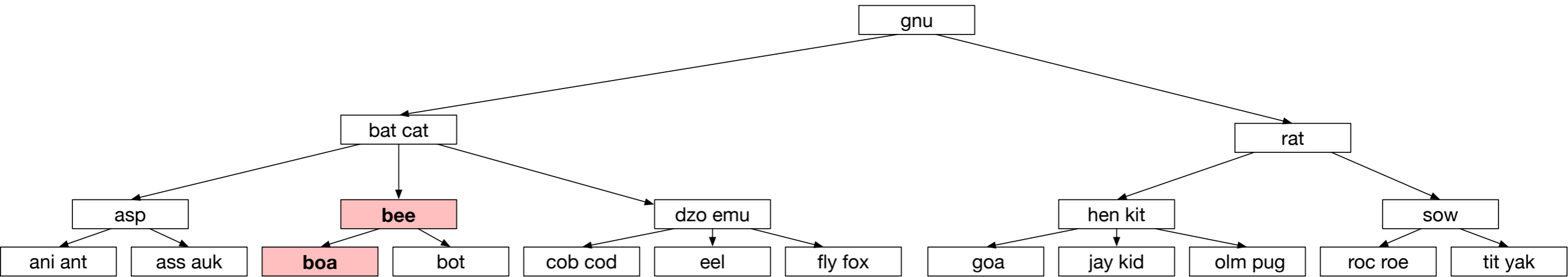


Activities



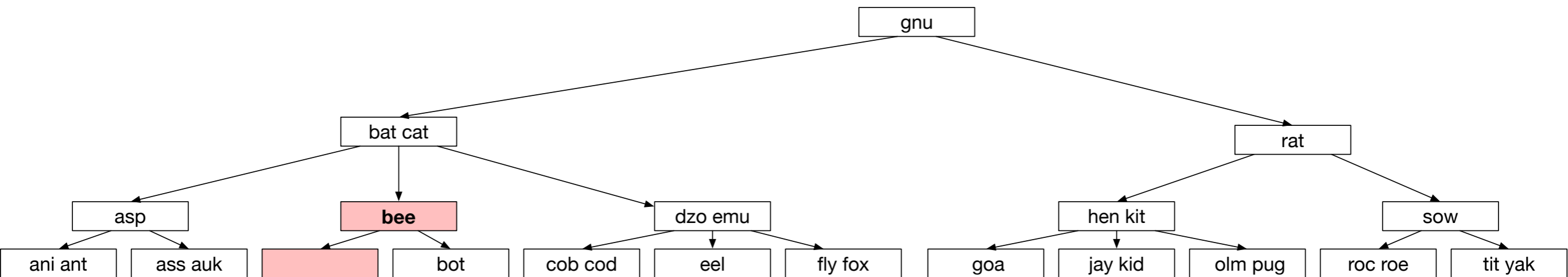
- Delete boa with predecessor

Activities



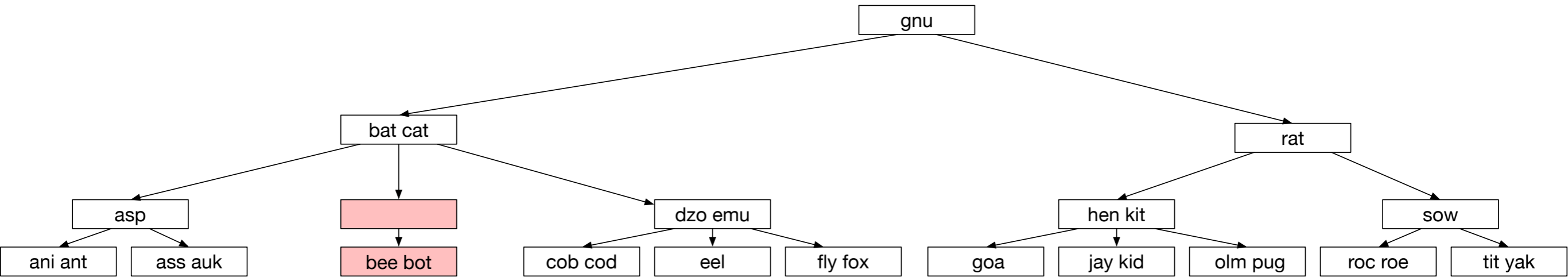
- Swap

Activities



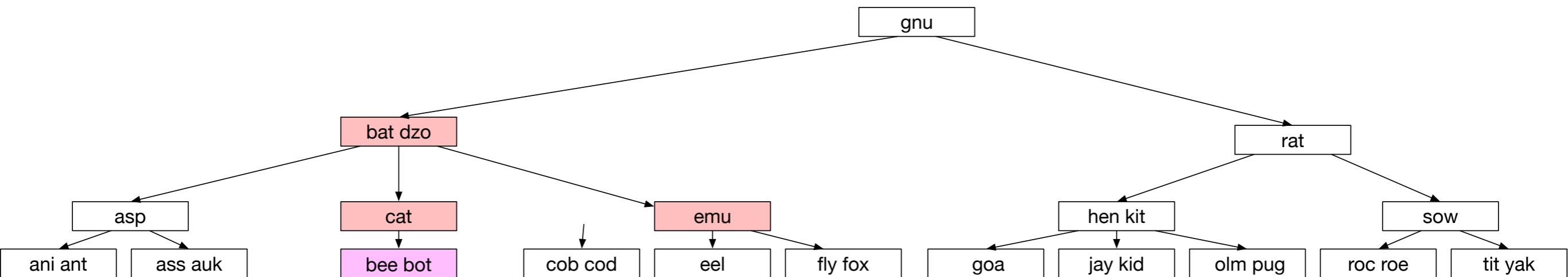
- Delete
- Deal with underflow
- Only merge is possible

Activities



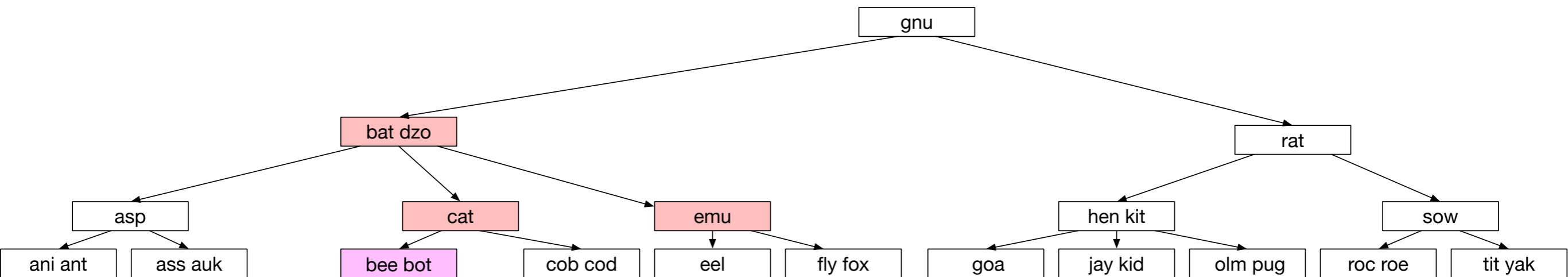
- This leaves the parent at an underflow
- Left sibling is at minimum capacity
- Right sibling is not
- Rotate Left: dzo goes up, cat goes down

Activities



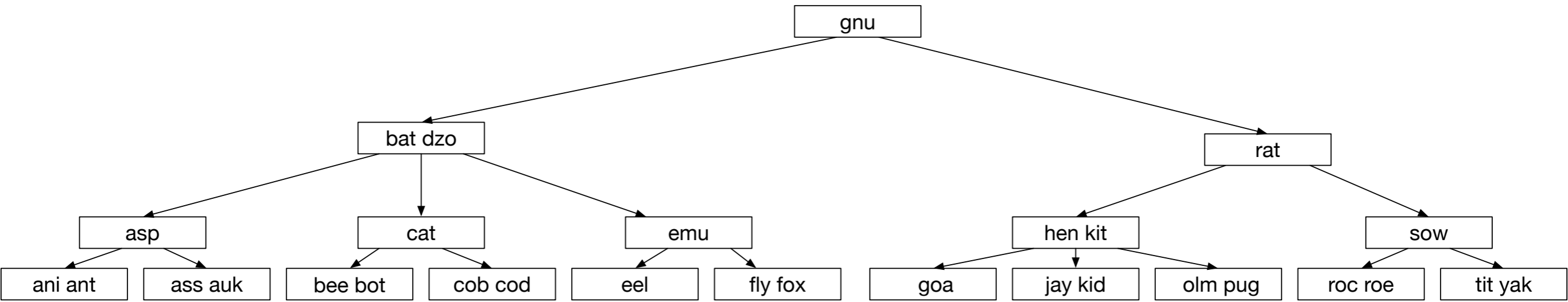
- Need to re-attach dangling node pointer

Activities



- And pretty-print

Activities

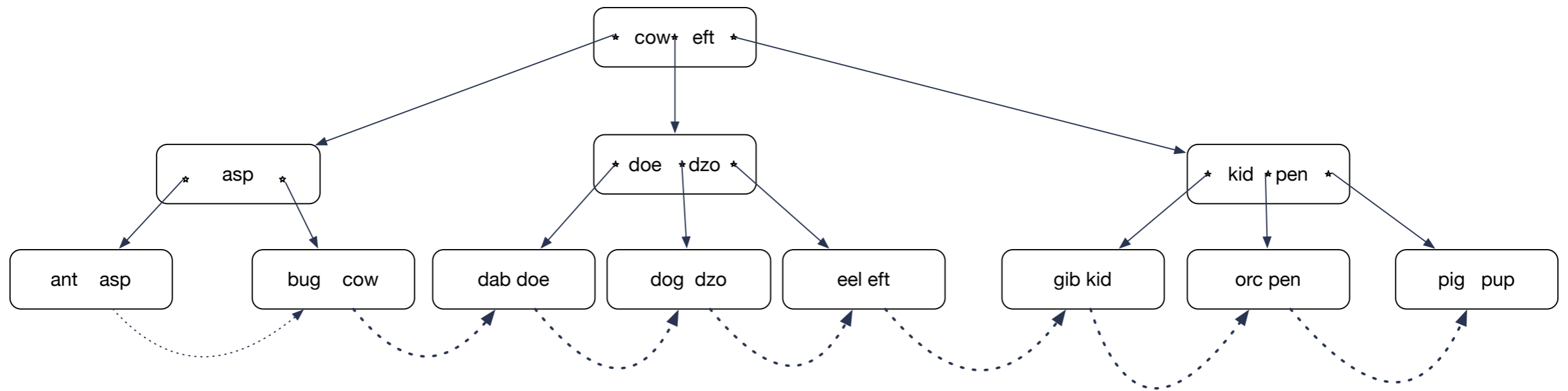


- The End

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



B+ Tree

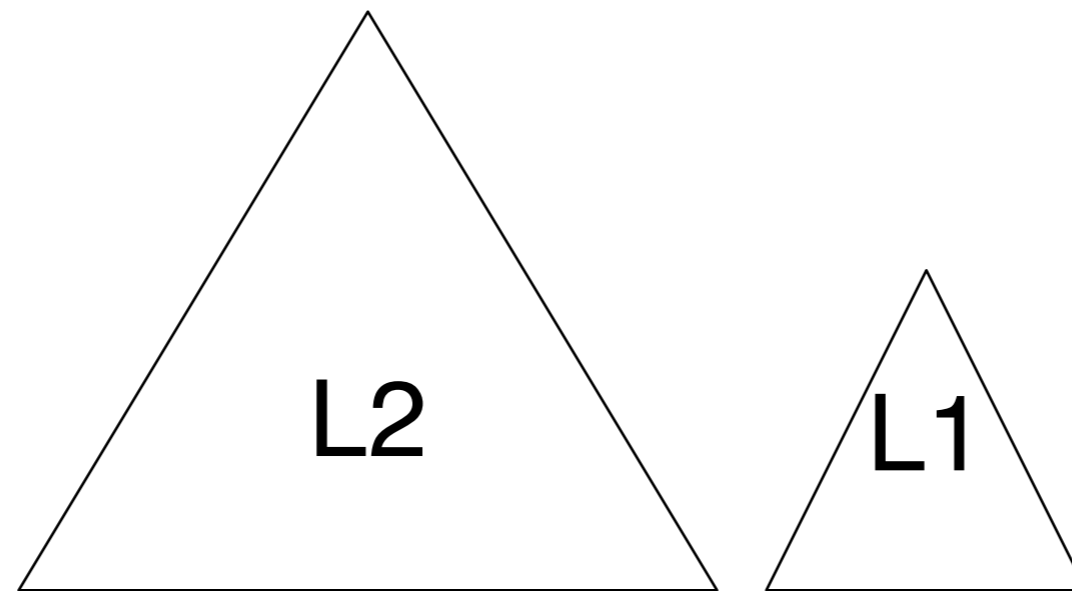
- Real life B+ trees:
 - Interior nodes have many more keys (e.g. 100)
 - Leaf nodes have as much data as they can keep
 - Need few levels:
 - Fast lookup

Combating Fringe Behavior

- Restructuring often happens with (almost) consecutive inserts
 - “Waves of misery” in running databases
- Use a way to buffer inserts

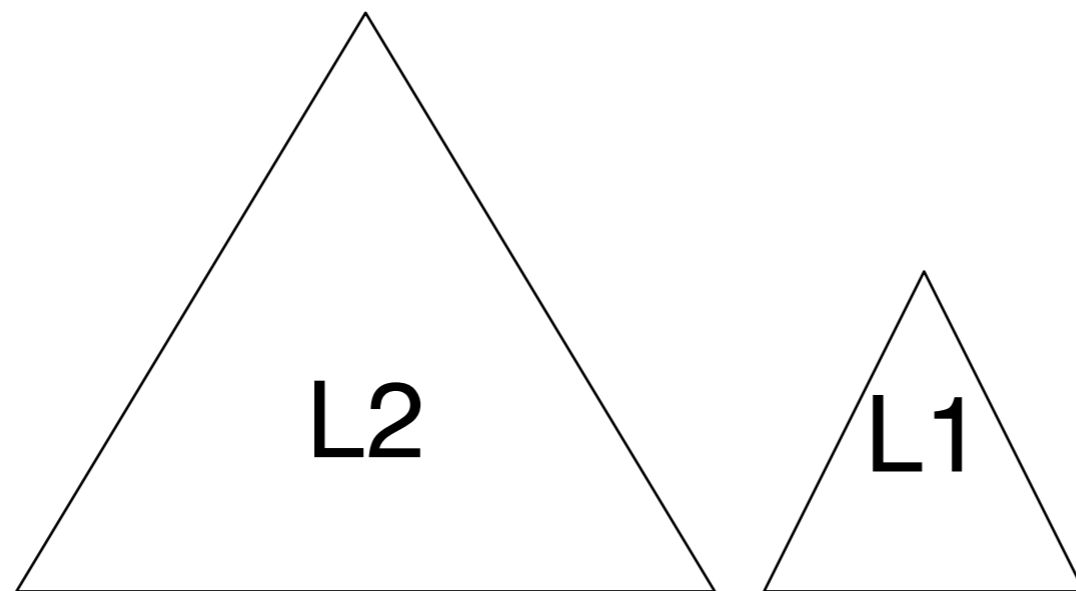
Combating Fringe Behavior

- Log-structured Merge Trees
 - Two related B-trees
 - Can optimize merge of L1 into L2
 - Allow merges to run in parallel with CRUD



Combating Fringe Behavior

- Log-structured Merge Trees
 - Keep L1-tree in main memory
 - Keep L2-tree in storage
 - Use Bloom filters in order to check whether keys are in either



Hashing

- Central idea of hashing:
 - Calculate the location of the record from the key
 - Hash functions:
 - Can be made indistinguishable from random function
 - SH3, MD5, ...
 - Often simpler
 - ID modulo slots

Hashing

- Can lead to collisions:
 - Two different keys map into the same address
 - Two ways to resolve:
 - **Open Addressing**
 - Have a rule for a secondary address, etc.
 - **Chaining**
 - Can store more than one datum at an address

Hashing

- Open addressing example:
 - Linear probing: Try the next slot

Hashing Example

```
def hash(a_string):  
    accu = 0  
    i = 1  
    for letter in a_string:  
        accu += ord(letter)*i  
        i+=1  
    return accu % 8
```

Insert "fly"

0	
1	
2	"fly", 2
3	
4	
5	
6	
7	

Hashing Example

```
def hash(a_string):  
    accu = 0  
    i = 1  
    for letter in a_string:  
        accu += ord(letter)*i  
        i+=1  
    return accu % 8
```

Insert "gnu"

hash("gnu") → 2

0	
1	
2	"fly", 2
3	"gnu", 2
4	
5	
6	
7	

Since spot 2 is taken, move to the next spot

Hashing Example

```
def hash(a_string):  
    accu = 0  
    i = 1  
    for letter in a_string:  
        accu += ord(letter)*i  
        i+=1  
    return accu % 8
```

Insert "hog"

hash("hog") -> 3

0	
1	
2	"fly", 2
3	"gnu", 2
4	"hog", 3
5	
6	
7	

Since spot is taken, move to the next

Hashing Example

```
def hash(a_string):  
    accu = 0  
    i = 1  
    for letter in a_string:  
        accu += ord(letter)*i  
        i+=1  
    return accu % 8
```

Looking for "gnu"

hash("gnu") -> 2

0	
1	
2	"fly", 2
3	"gnu", 2
4	"hog", 3
5	
6	
7	"pig", 7

Try out location 2. Occupied, but not by "gnu"



www.shutterstock.com • 762496525

Hashing Example



www.shutterstock.com • 762496525

```
def hash(a_string):  
    accu = 0  
    i = 1  
    for letter in a_string:  
        accu += ord(letter)*i  
        i+=1  
    return accu % 8
```

Looking for "gnu"

hash("gnu") -> 2

0	
1	
2	"fly", 2
3	"gnu", 2
4	"hog", 3
5	
6	
7	"pig", 7

Try out location 3. Find "gnu"

Hashing Example



```
def hash(a_string):  
    accu = 0  
    i = 1  
    for letter in a_string:  
        accu += ord(letter)*i  
        i+=1  
    return accu % 8
```

Looking for "ram"

hash("ram") -> 3

0	
1	
2	"fly", 2
3	"gnu", 2
4	"hog", 3
5	
6	
7	"pig", 7

Look at location 3: someone else is there

Look at location 4: someone else is there

Look at location 5: nobody is there, so if it were in the dictionary, it would be there

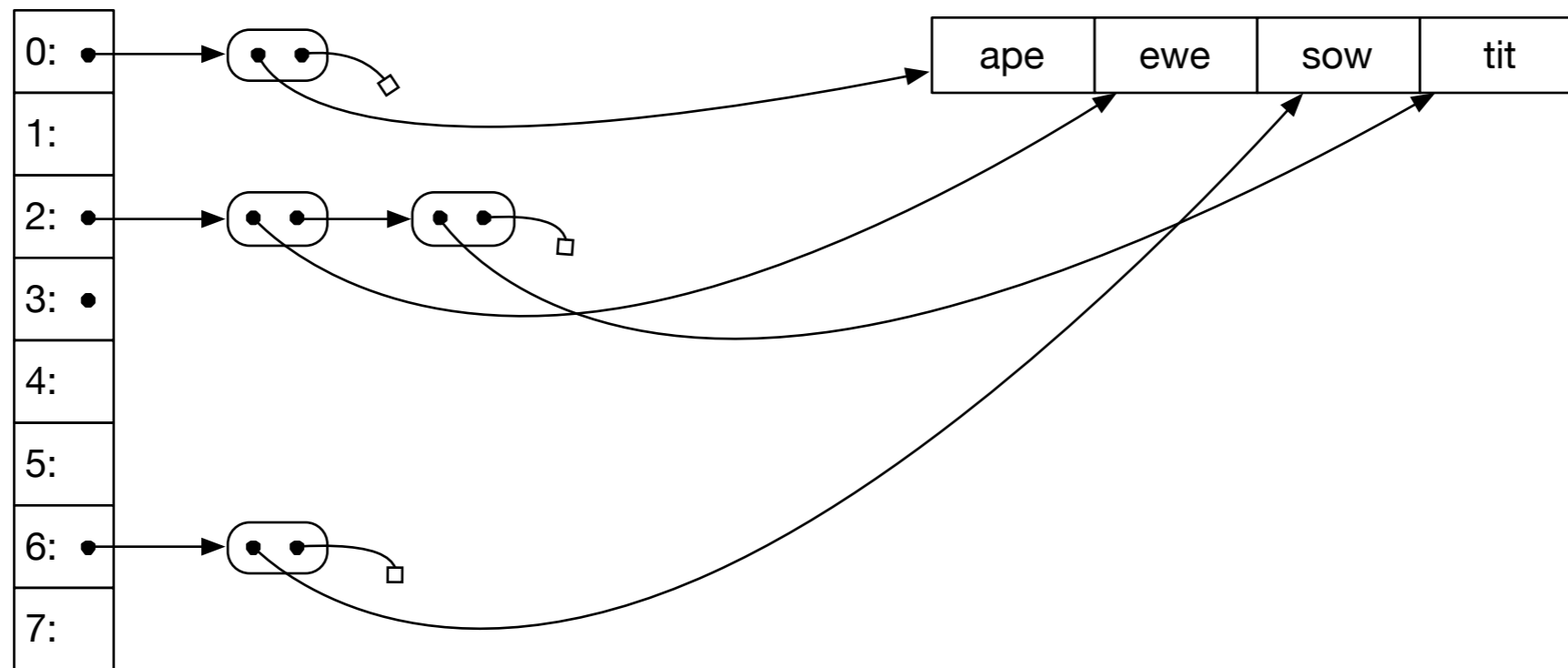
Hashing

- Linear probing leads to convoys:
 - Occupied cells tend to coalesce
- Quadratic probing is better, but might perform worse with long cache lines
- Large number of better versions are used:
 - Passbits
 - Cuckoo hashing
 - Uses two hash functions
 - Robin Hood hashing ...

Hashing

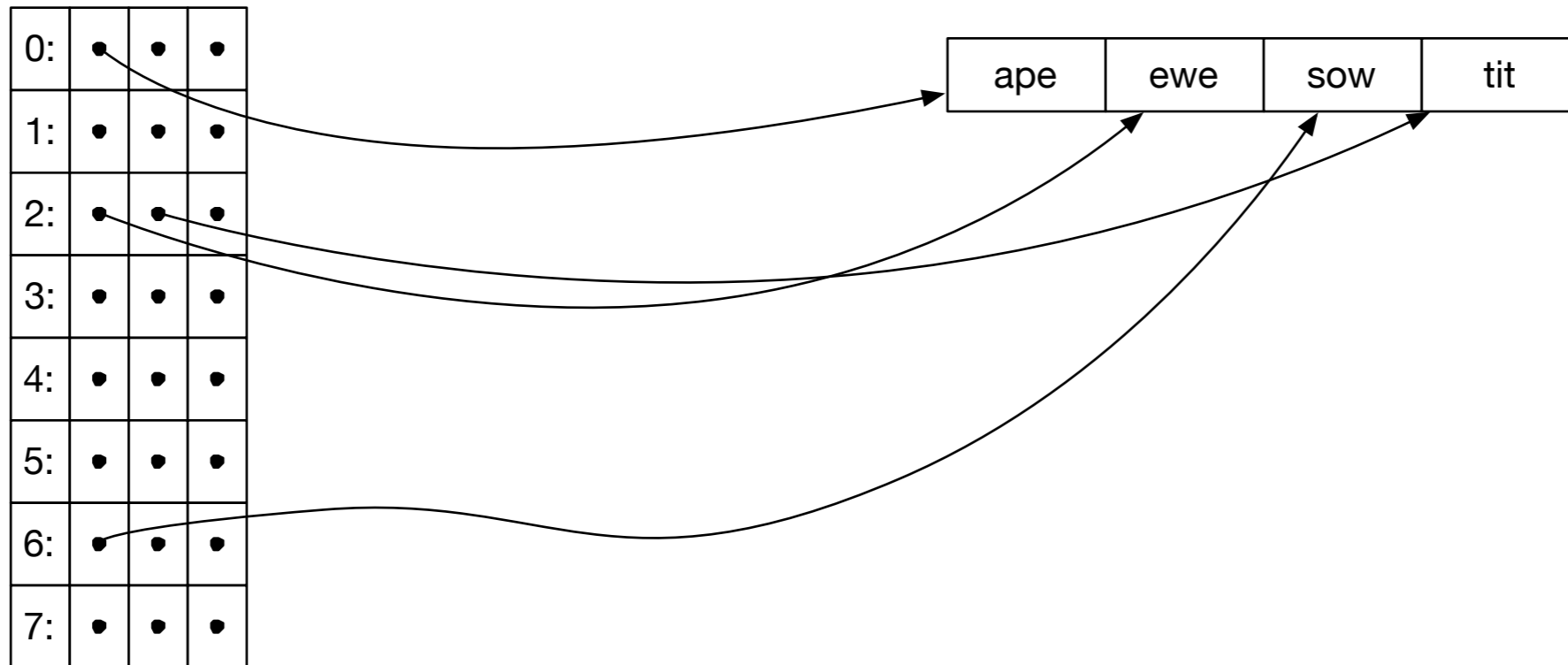
- Chaining
 - Keep data mapped to a location in a “bucket”
 - Can implement the bucket in several ways
 - Linked List

Hashing



Chaining Example with linked lists

Hashing Example



**Chaining Example with an array of pointers
(with overflow pointer if necessary)**

Hashing Example

0:	ape	null	null
1:	null	null	null
2:	ewe	tit	null
3:	null	null	null
4:	null	null	null
5:	null	null	null
6:	sow	null	null
7:	null	null	null

Chaining with fixed buckets
Each bucket has two slots and a pointer
to an overflow bucket

Hashing

- Extensible Hashing:
 - Load factor $\alpha = \text{Space Used} / \text{Space Provided}$
 - Load factor determines performance
 - Idea of extensible hashing:
 - Gracefully add more capacity to a growing hash table

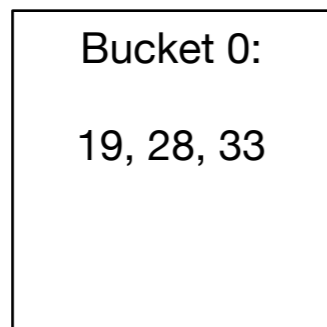
Linear Hashing

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:

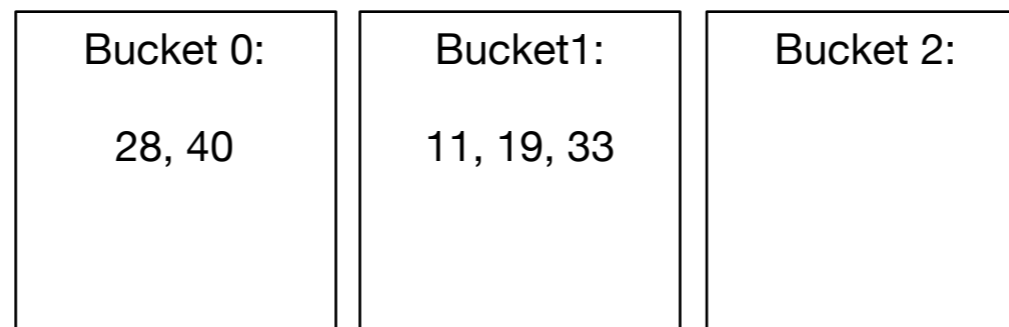
Bucket 0: 28	Bucket1: 19, 33
-----------------	--------------------

- After more insertions, the load factor again exceeds 2

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

Linear Hashing

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



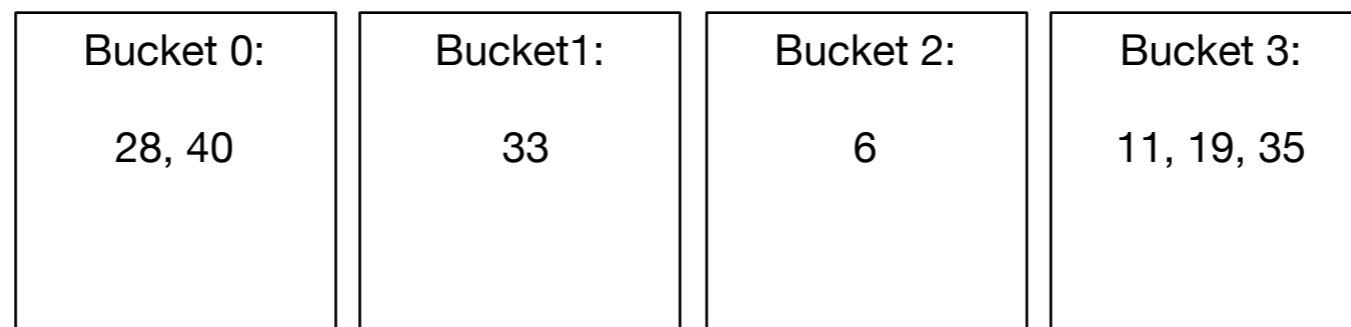
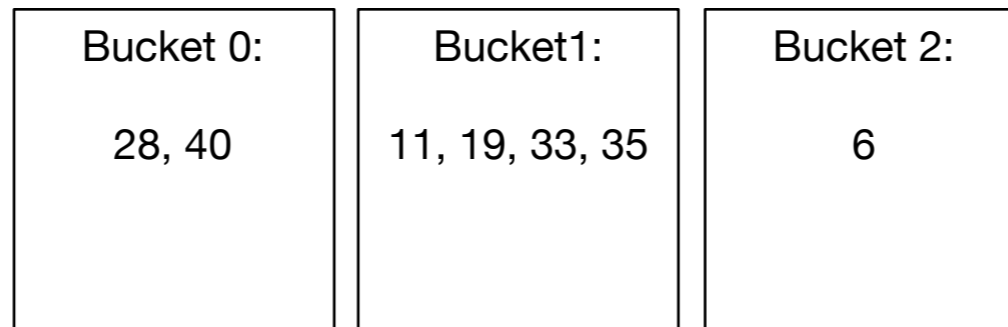
- For the rehashing, we now use two bits, i.e.

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

- But only for those items in Bucket 0

Linear Hashing

- After some more insertions, Bucket 1 will split



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

Bucket 0:

19, 28, 33

Linear Hashtable Evolution

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

Number of buckets: 2

Split pointer: 0

Level: 1

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

Bucket 0: 28	Bucket1: 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**1  
    if a < s:  
        a = hash(c) % 2**(1+1)  
    return a
```

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

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

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

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

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

Add items 6, 35

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

Because of overflow, we split
Bucket 1

Linear Hashtable Evolution

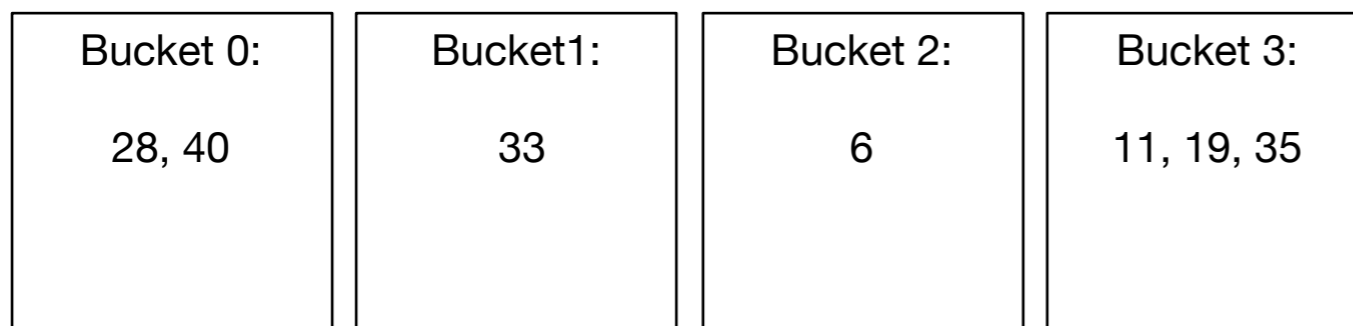
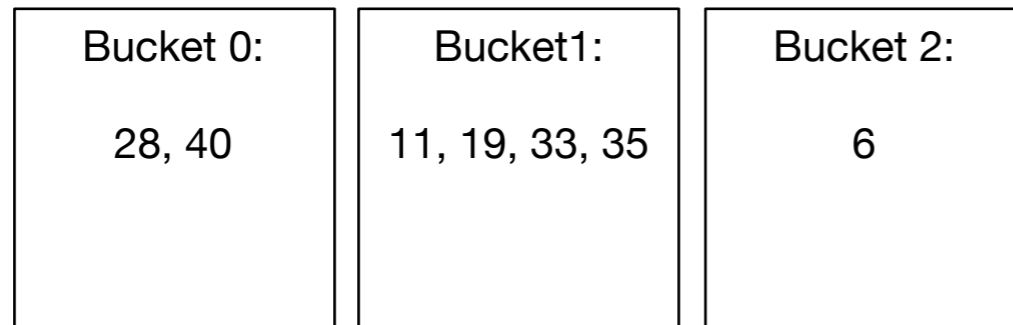
$$N = 4 = 2^2 + 0$$

Number of buckets: 4

Split pointer: 0

Level: 2

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



Linear Hashtable Evolution

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

Number of buckets: 4

Split pointer: 0

Level: 2

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

Bucket 0: 28, 40	Bucket 1: 33	Bucket 2: 6	Bucket 3: 11, 19, 35
---------------------	-----------------	----------------	-------------------------

Now add keys 8, 49

Bucket 0: 28, 40, 8	Bucket 1: 33, 49	Bucket 2: 6	Bucket 3: 11, 19, 35
------------------------	---------------------	----------------	-------------------------

Creates an overflow!
Need to split!

Linear Hashtable Evolution

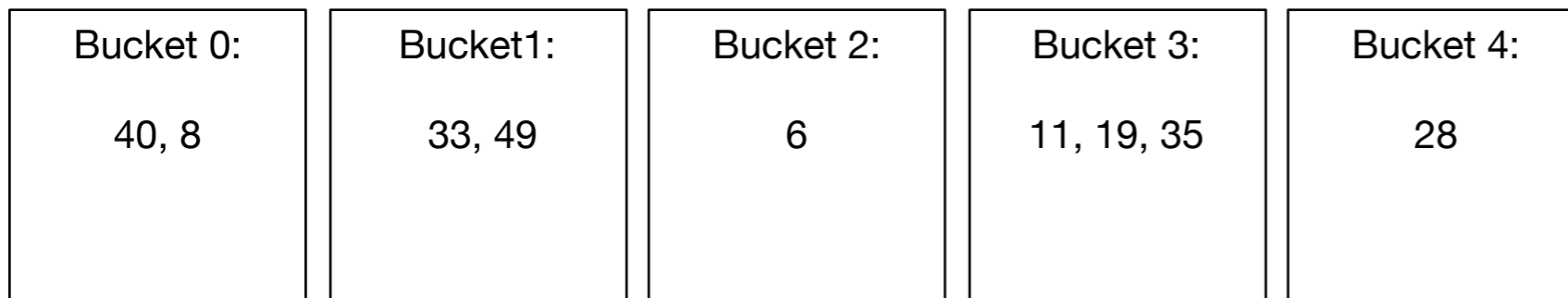
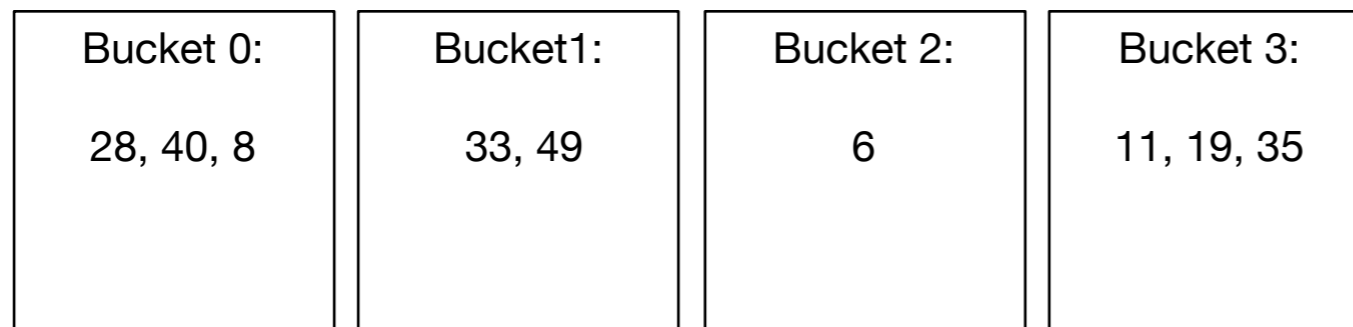
$$N = 5 = 2^2 + 1$$

Number of buckets: 1

Split pointer: 1

Level: 2

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



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**1  
    if a < s:  
        a = hash(c) % 2**(1+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

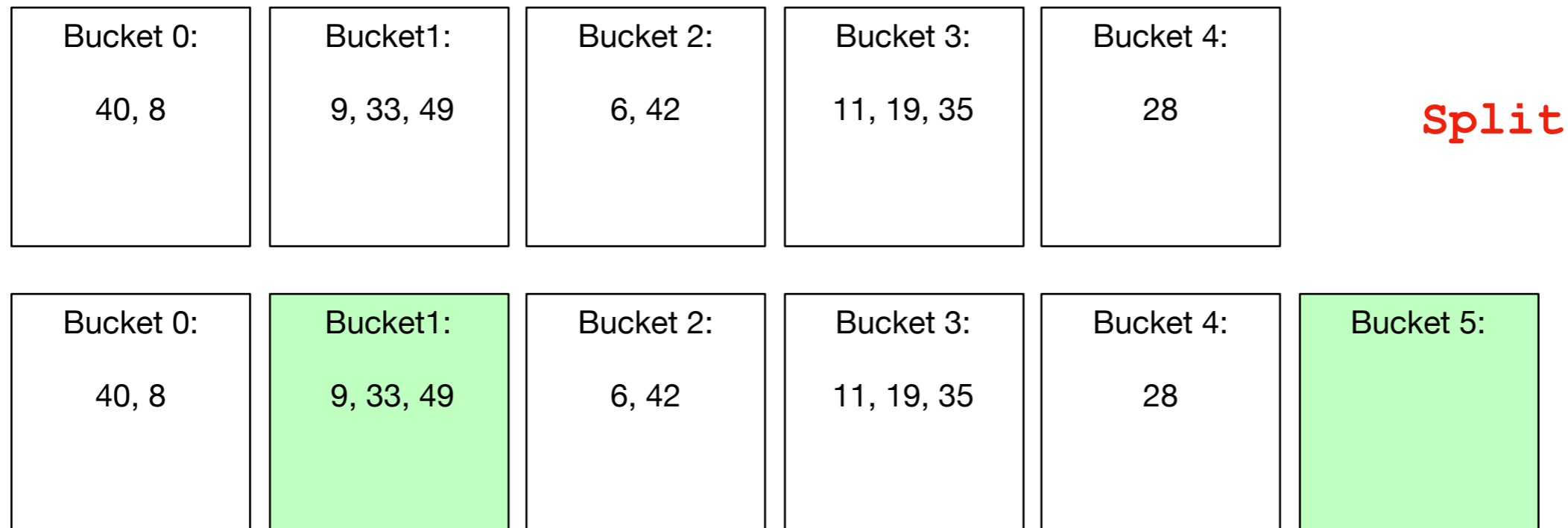
$$N = 6 = 2^2 + 2$$

Number of buckets: 1

Split pointer: 2

Level: 2

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



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

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

Linear Hashtable Evolution

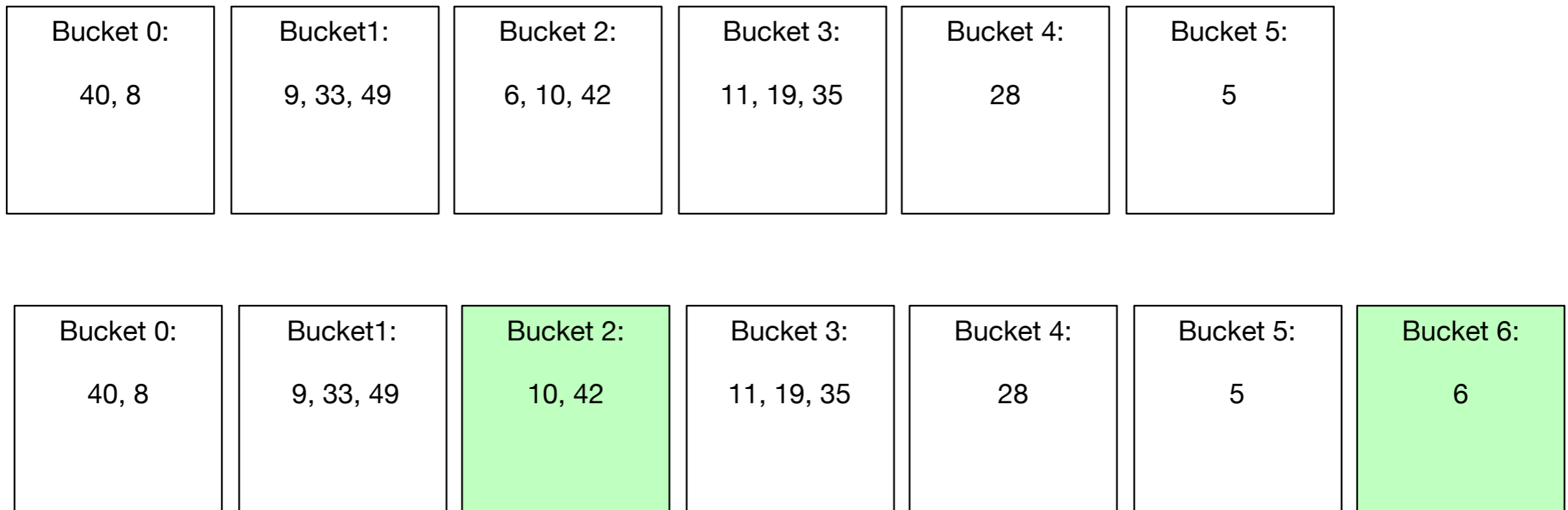
$$N = 7 = 2^2 + 3$$

Number of buckets: 7

Split pointer: 3

Level: 2

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



Linear Hashtable Evolution

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

Number of buckets: 7

Split pointer: 3

Level: 2

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

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

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

Linear Hashtable Evolution

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

Number of buckets: 8

Split pointer: 0

Level: 3

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

Bucket 0:	Bucket 1:	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	

add 13, 54

Bucket 0:	Bucket 1:	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**1  
    if a < s:  
        a = hash(c) % 2**(1+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:	Bucket 1: 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	add 1, 81
-----------	------------------------	-------------------------	-------------------------	---------------------	--------------------	--------------------	-----------	--------------------	-----------

Bucket 0:	Bucket 1: 1, 9, 33, 49, 81	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

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

Number of buckets: 10

Split pointer: 2

Level: 3

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

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