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 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

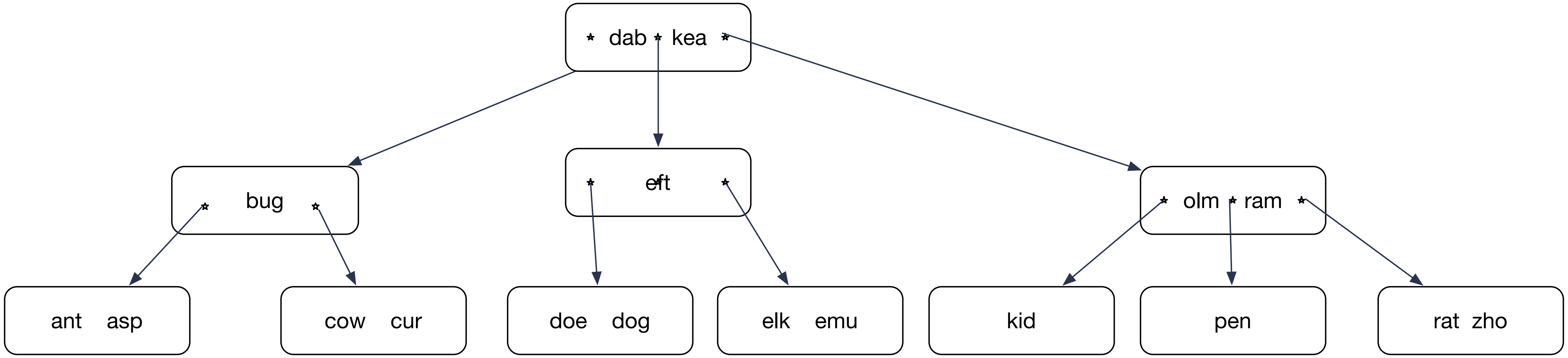
- Log:
  - Append, Read

B-Trees

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

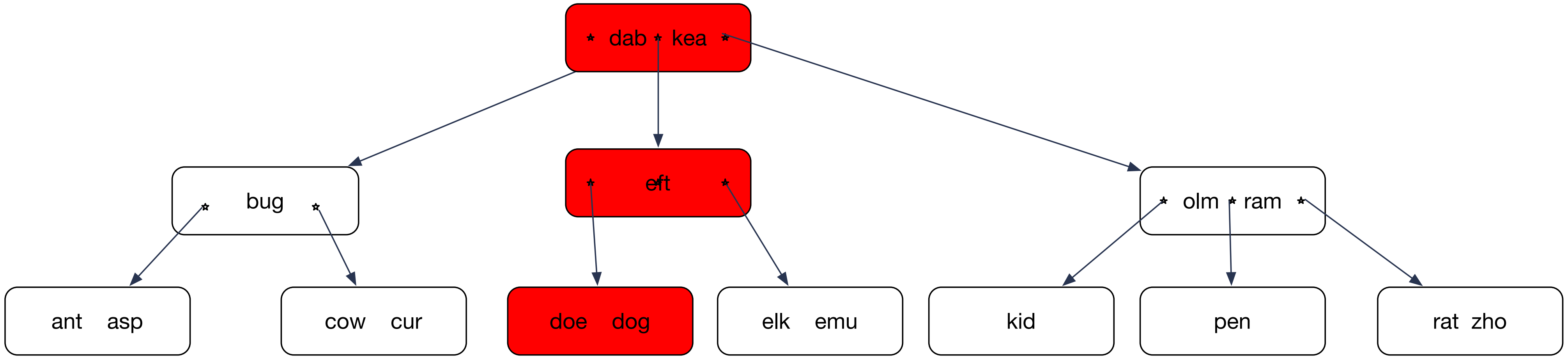
- Example: 2-3 tree: Each node has two or three children



B-Trees

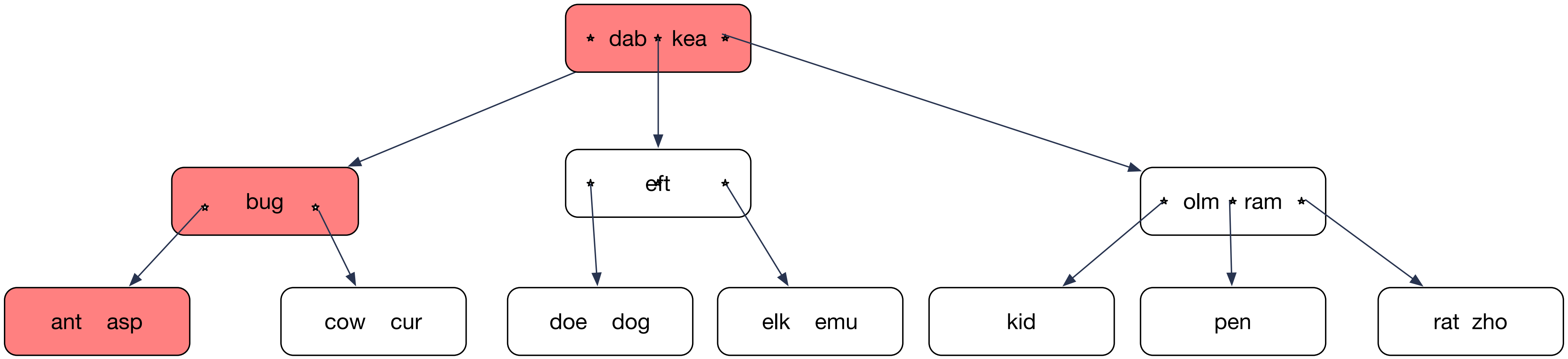
- Read dog:
    - Load root, determine location of dog in relation to the keys
    - Follow middle pointer
    - Follow pointer to the left
    - Find “dog”

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” :



dab kea

bug

eft

olm ram

ant asp

cow cur

doe dog

elk emu

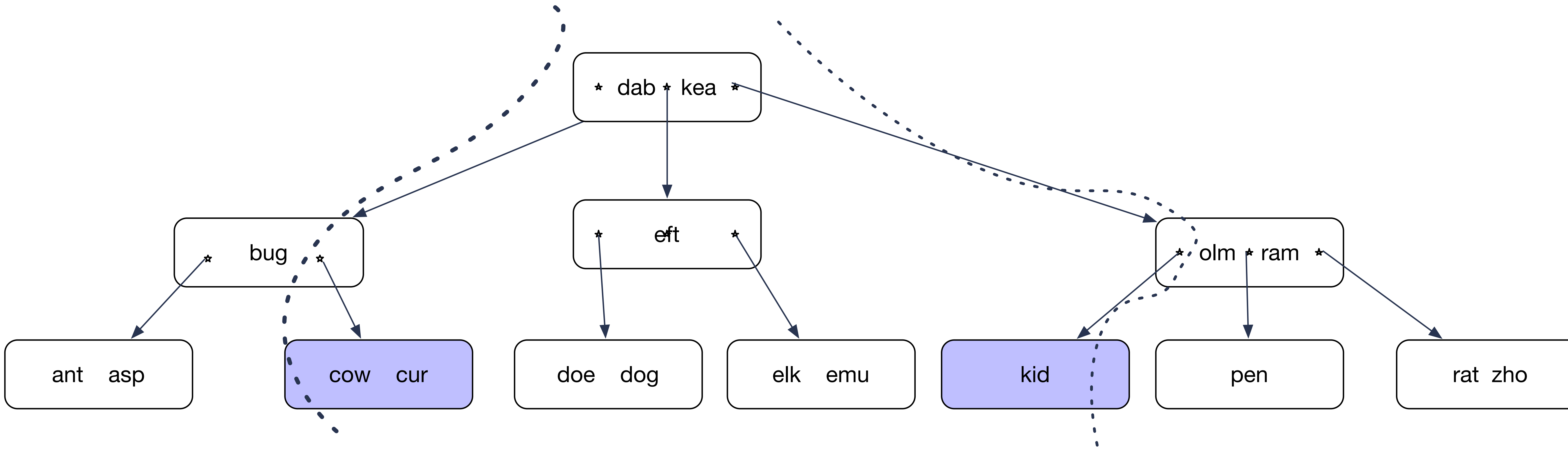
kid

pen

rat zho

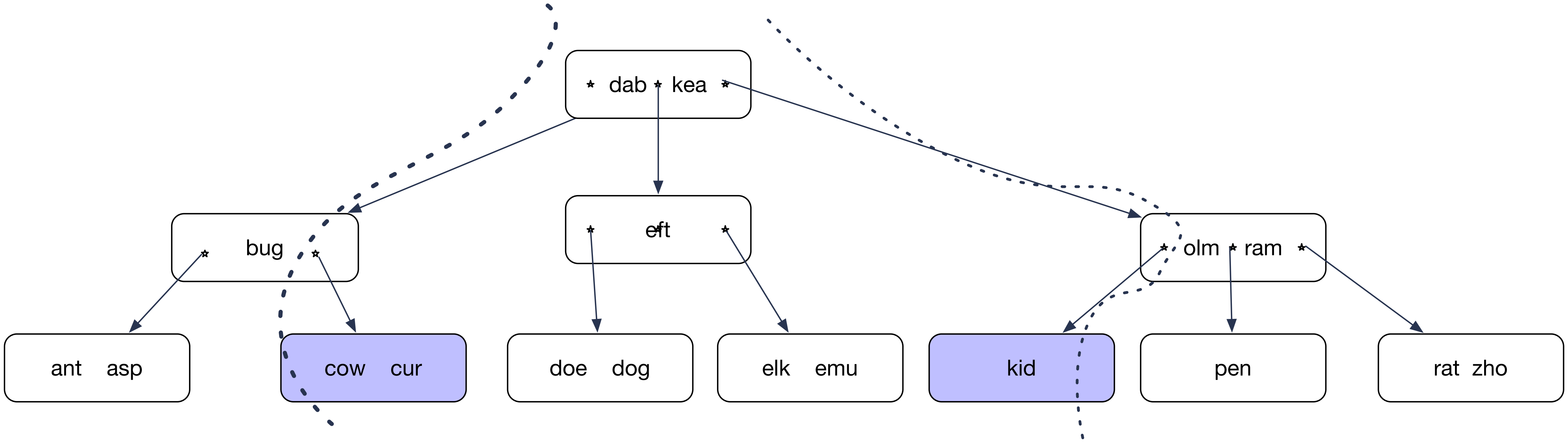
B-Trees

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



B-Trees

- Recursively enumerate all nodes between the lines starting with root



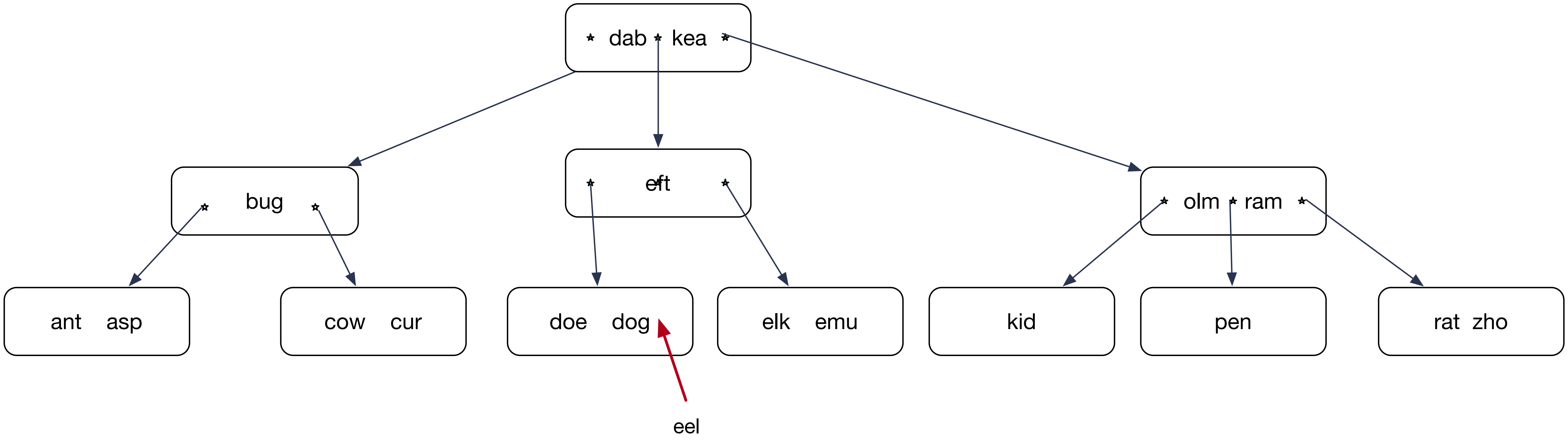
B-trees

- 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

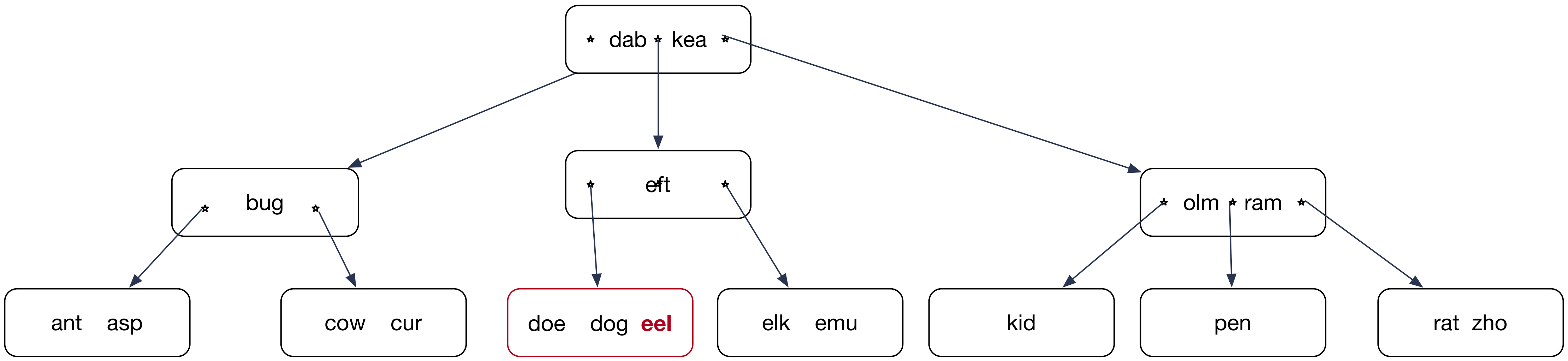
B-trees

- 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”

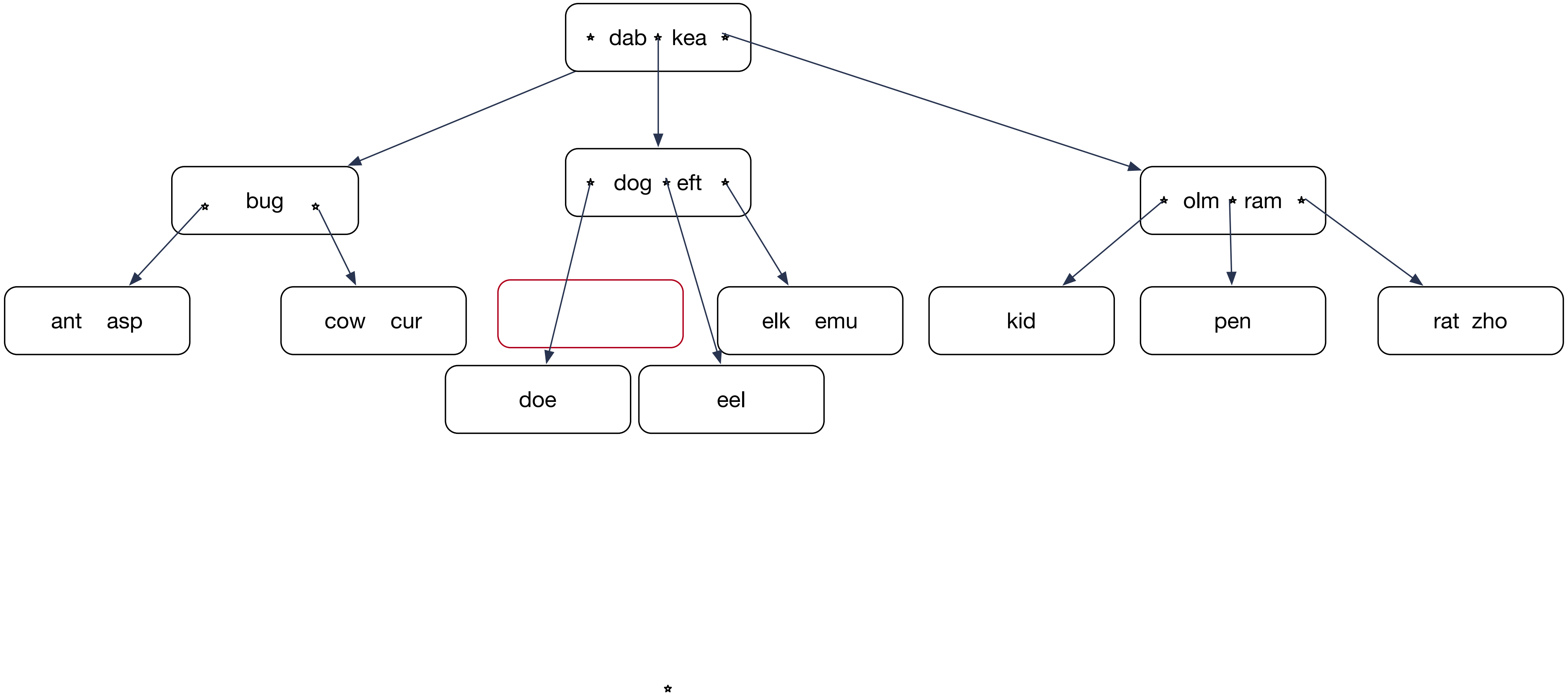
B-trees


⋆ 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

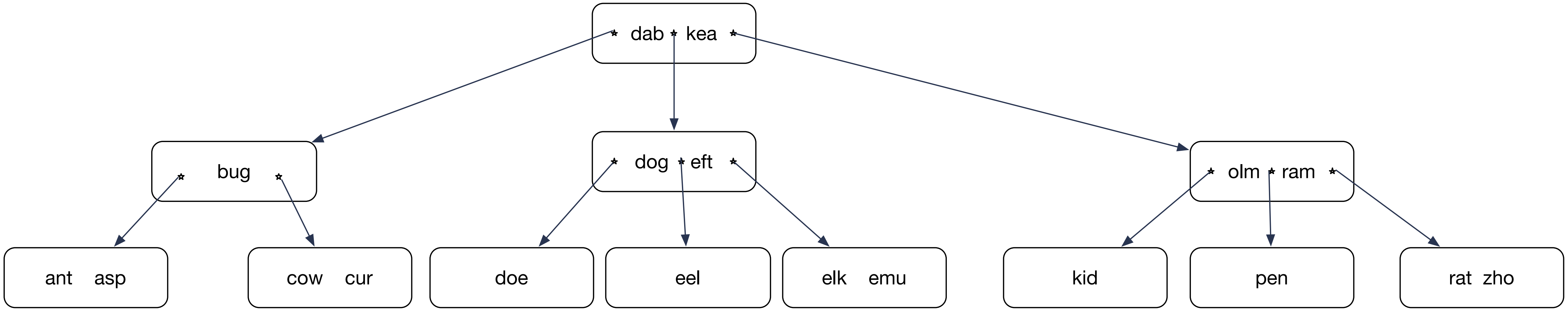
- Insert: Lock all nodes from root on down so that only one process can operate on the nodes
- Tree only grows a new level by splitting the root

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

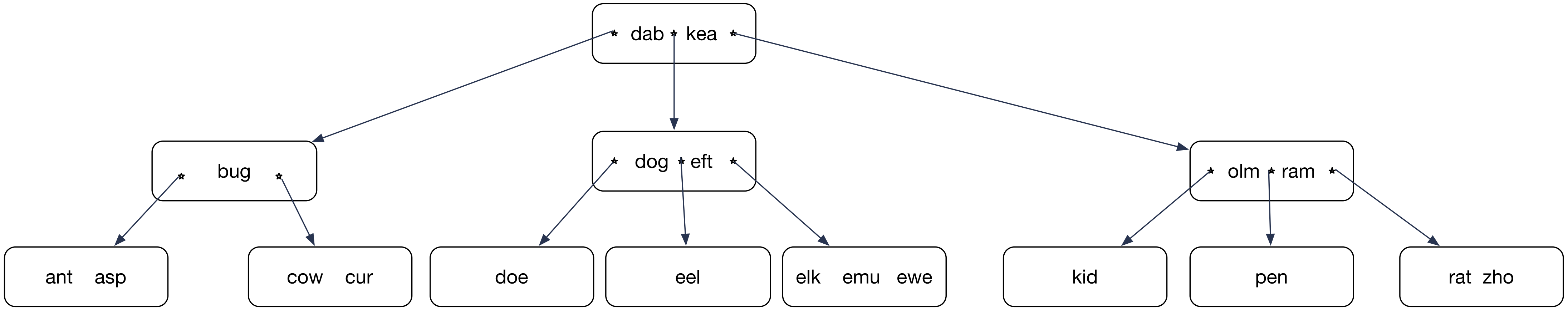
B-tree

- Insert ewe into



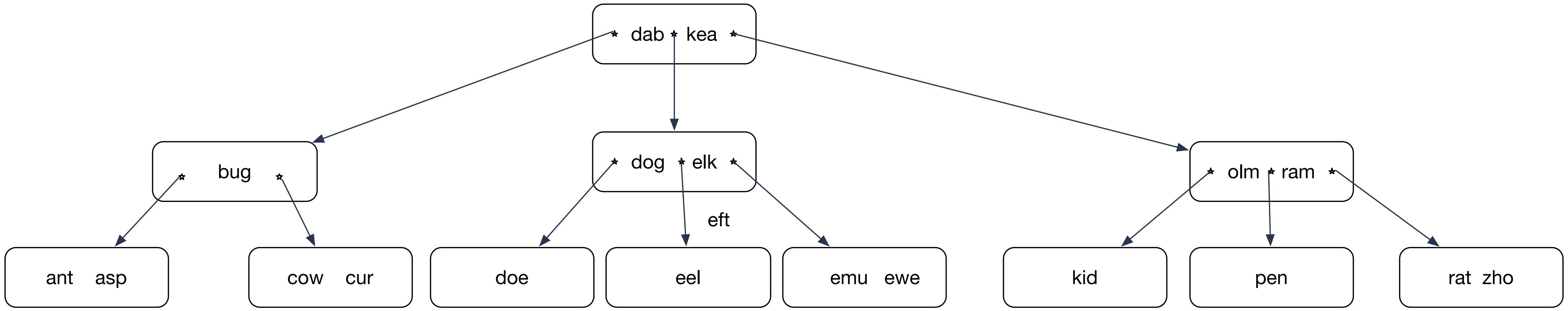
B-tree

- Insert ewe



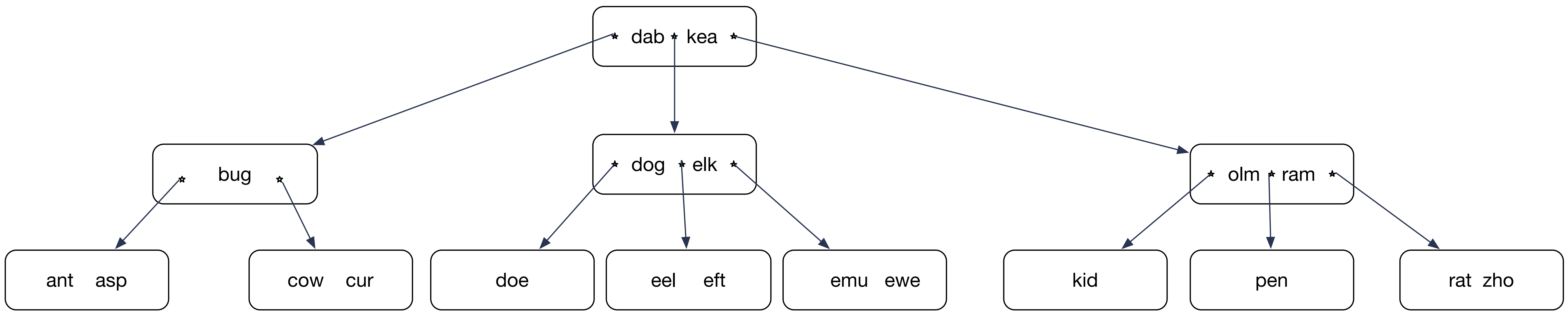
B-tree

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



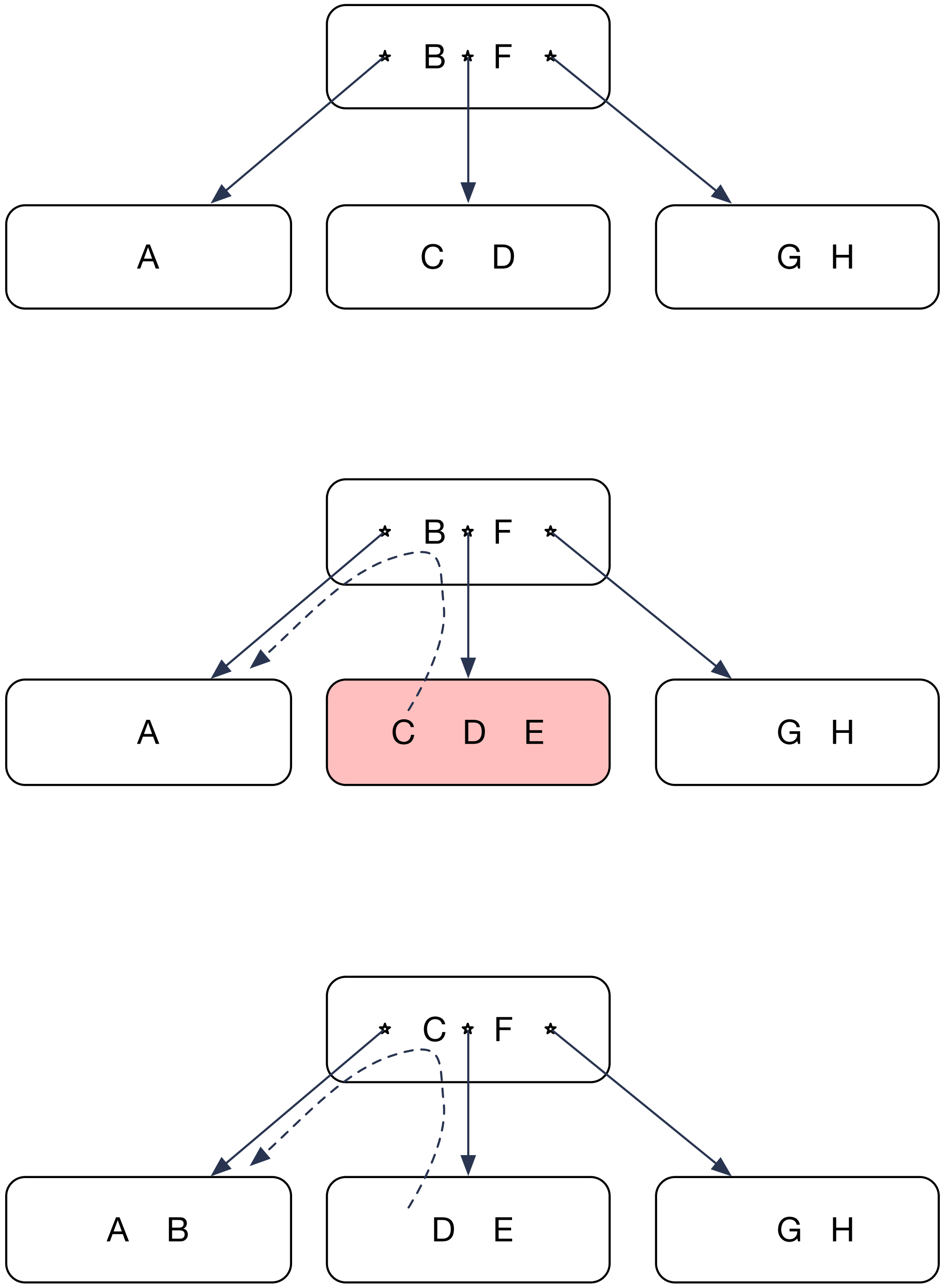
B-tree

- Insert eft into the leaf node

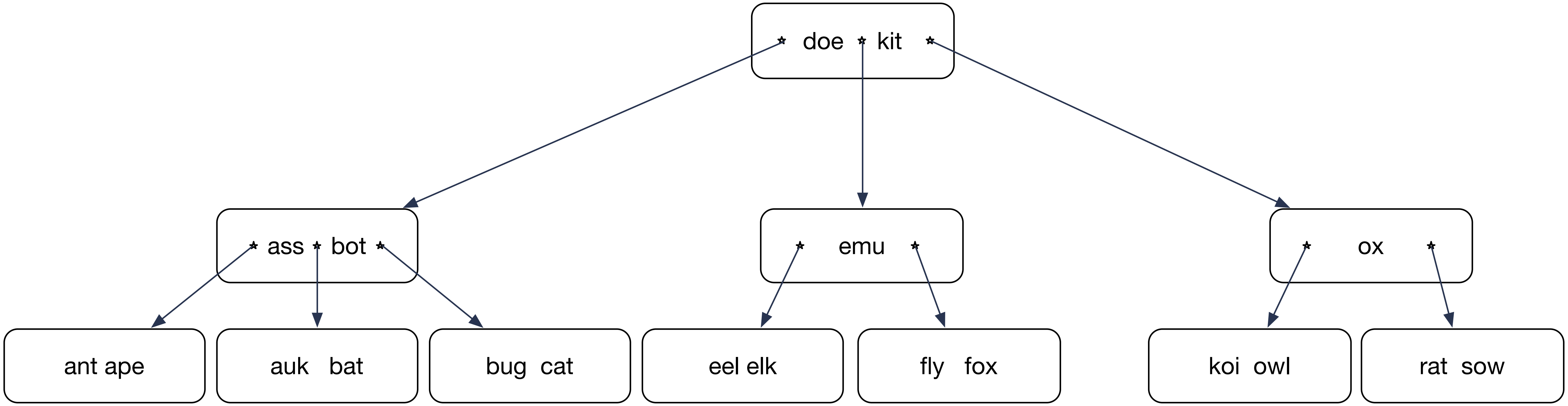


B-tree

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



B-tree



doe kit

ass bot

emu

ox

ant ape

auk bat

bug cat

eel elk

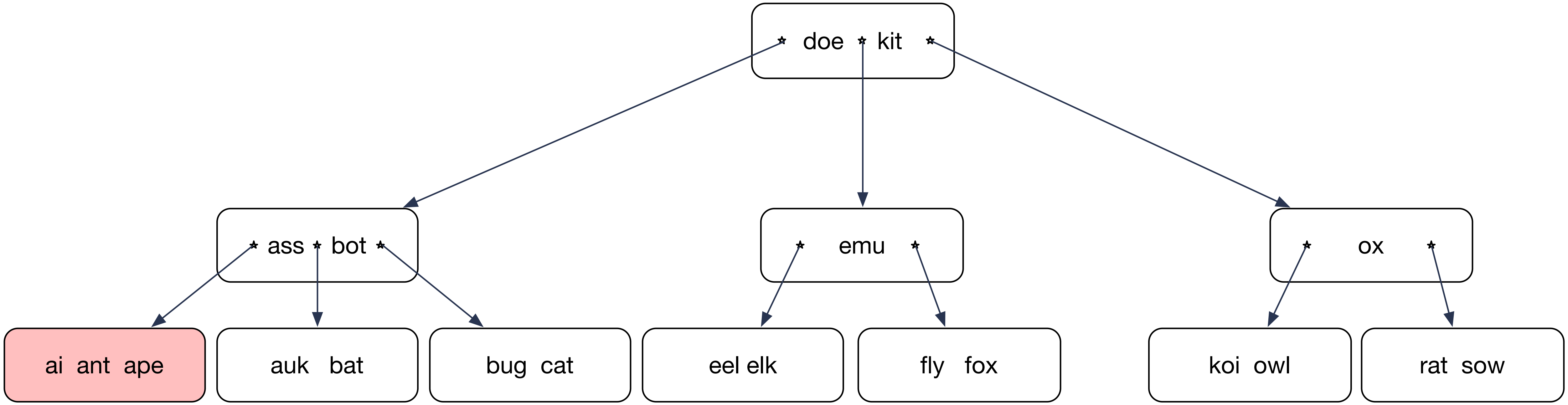
fly fox

koi owl

rat sow

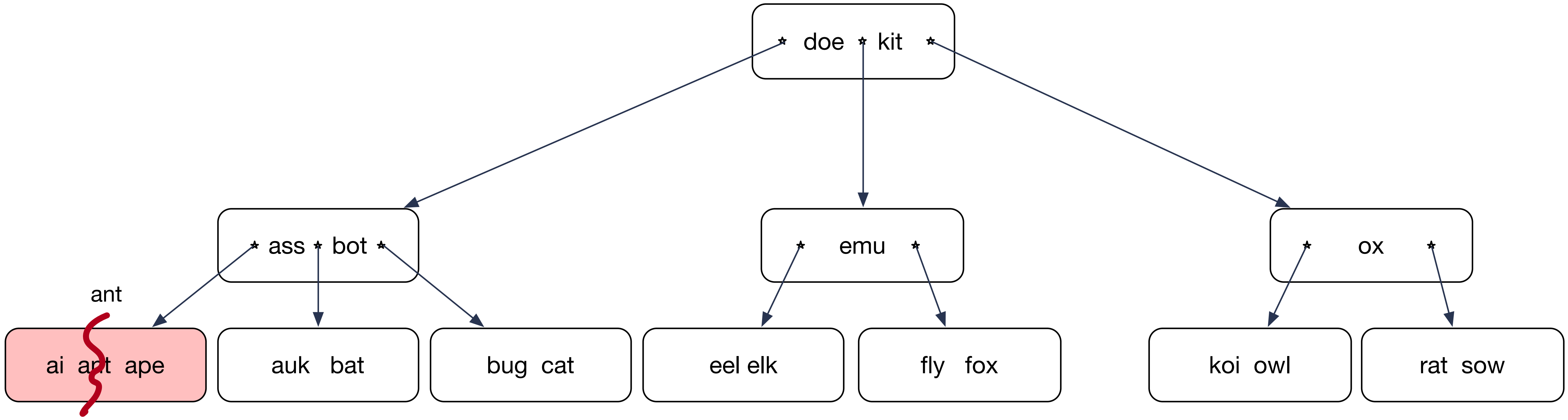
Now insert “ai”

B-tree



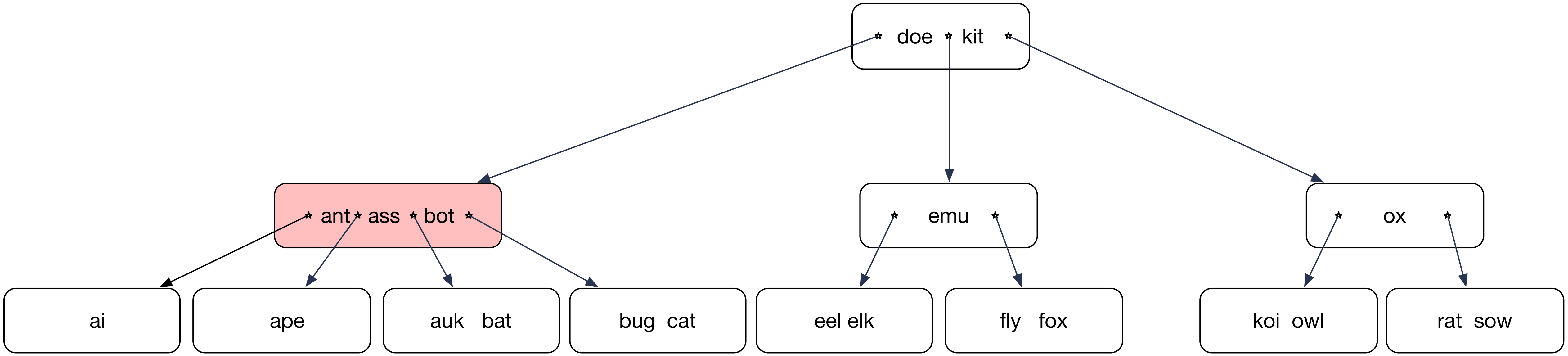
Insert creates an overflowing node
Only one neighboring sibling, but that one is full
Split!

B-tree



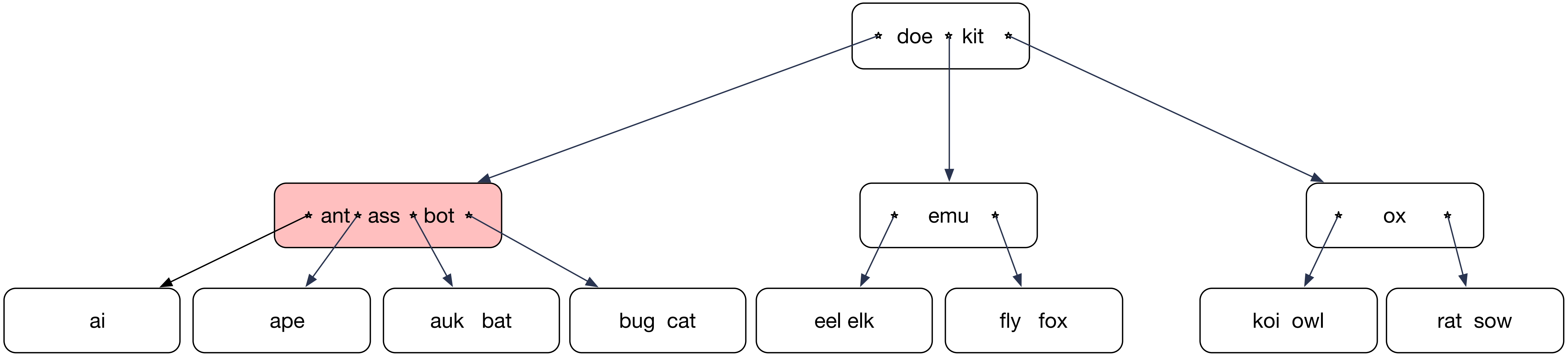
Middle key moves up

B-tree



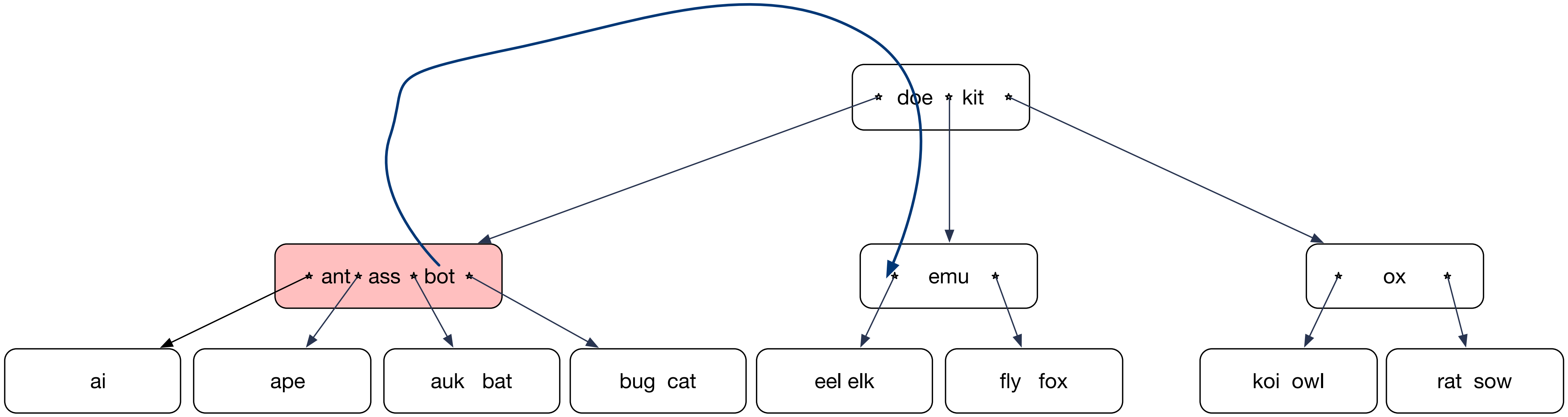
Unfortunately, this gives another overflow
But this node has a right sibling not at full capacity

B-tree



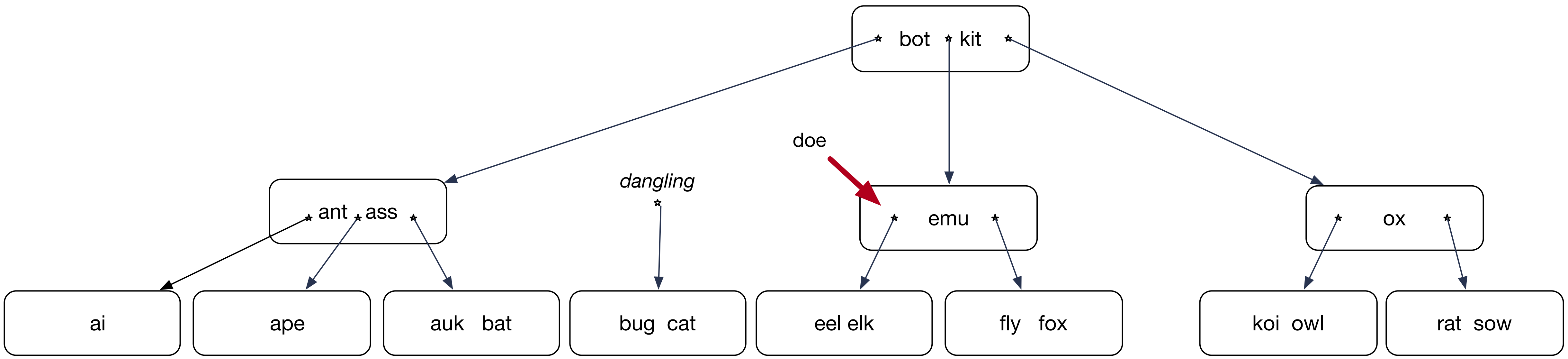
Right rotate:
Move “bot” up
Move “doe” down
Reattach nodes

B-tree



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

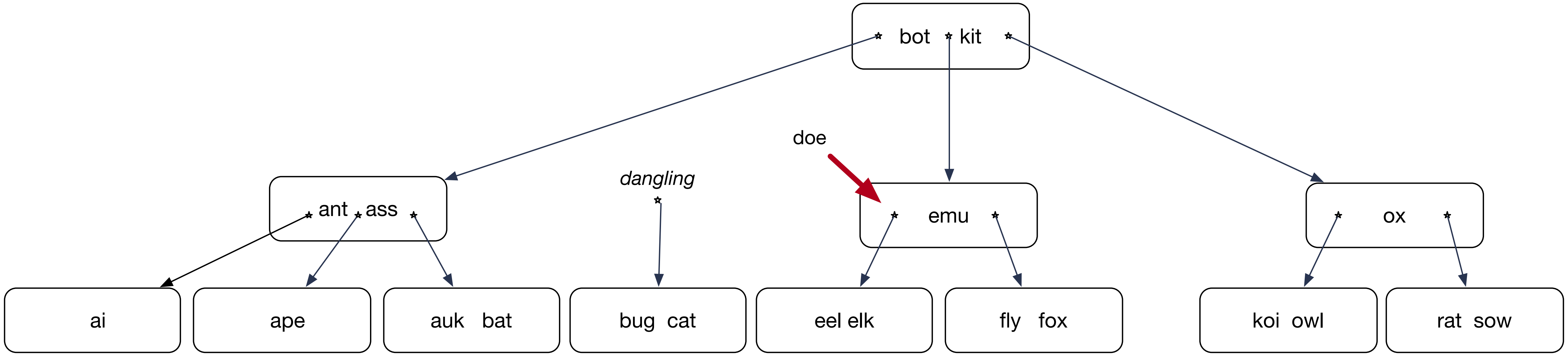
B-tree



“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

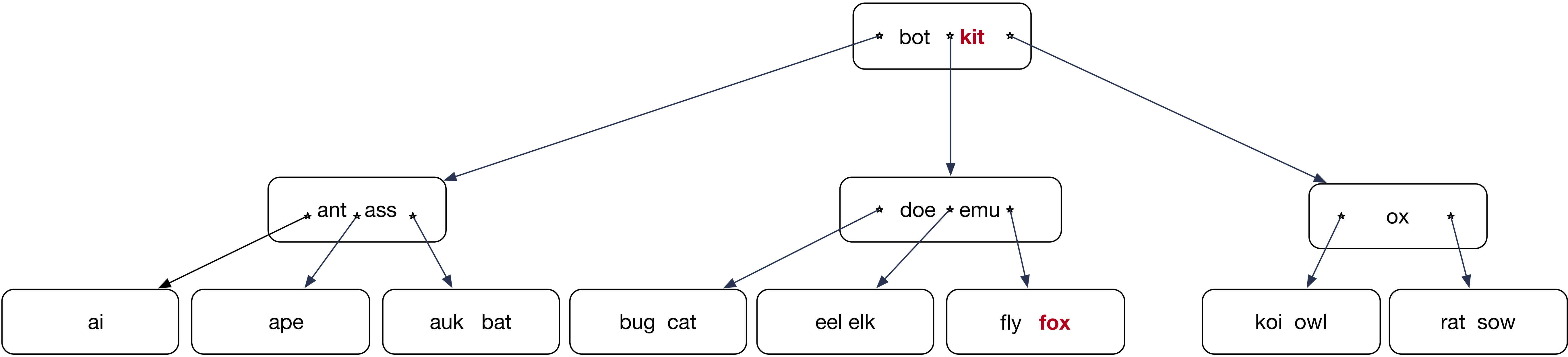
- Deletes
  - Usually restructuring not done because there is no need
  - Underflowing nodes will fill up with new inserts

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

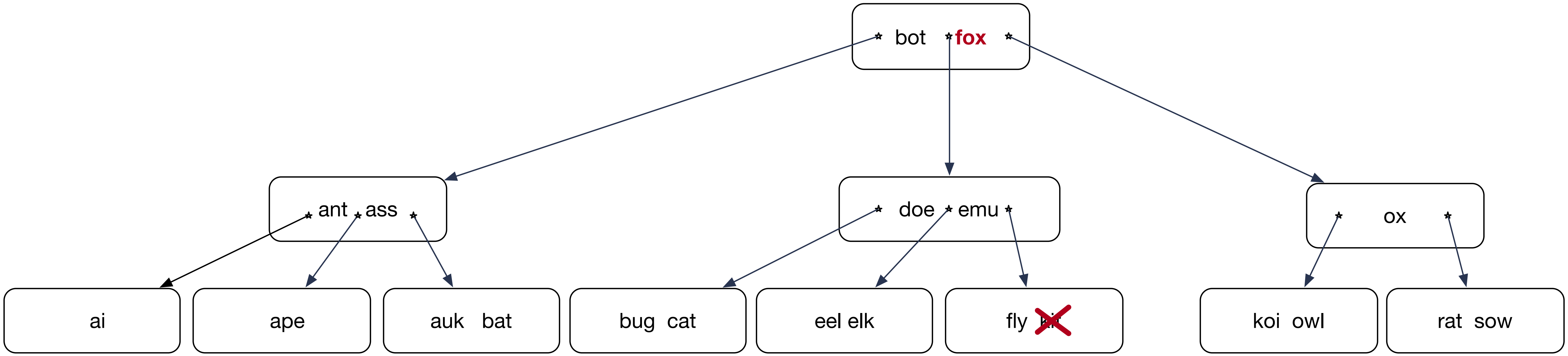
B-tree

Delete “kit”



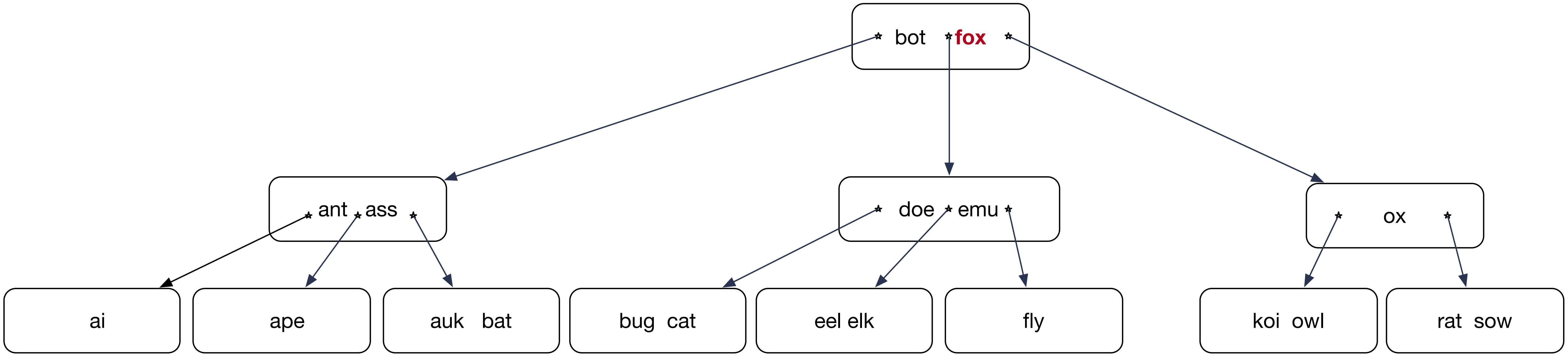
Delete “kit”
“kit” is in an interior node.
Exchange it with the key in the leave immediately before
“fox”

B-tree



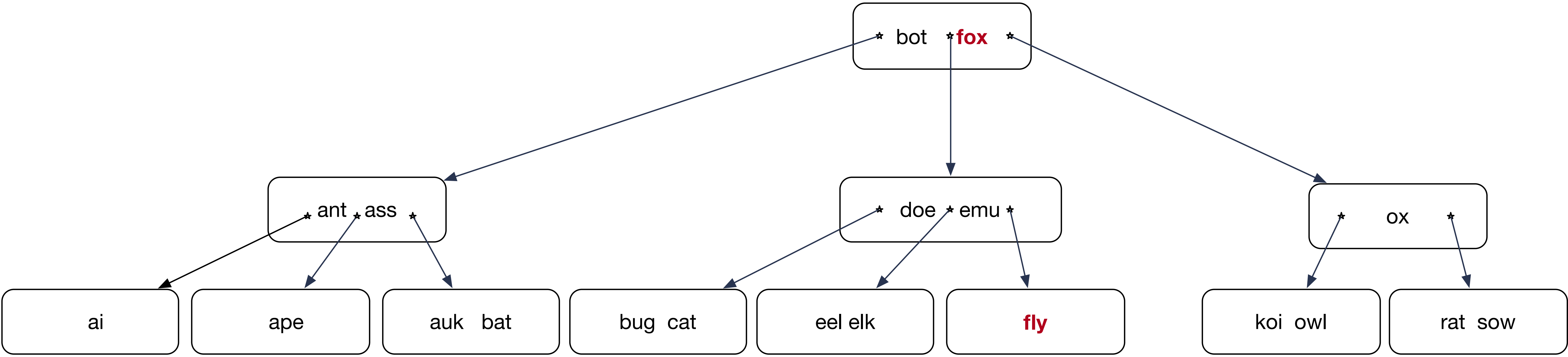
After interchanging “fox” and “kit”, can delete “kit”

B-tree



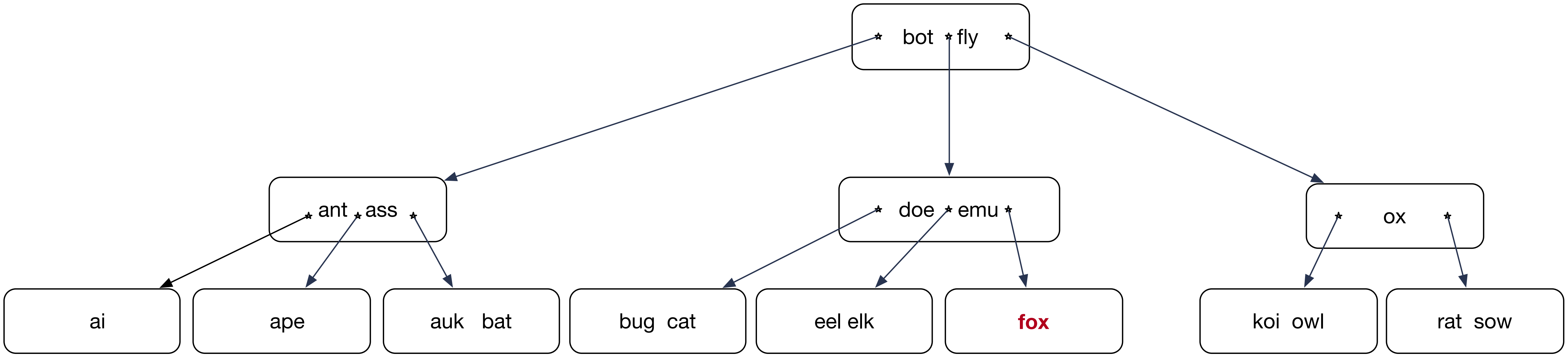
Now delete “fox”

B-tree



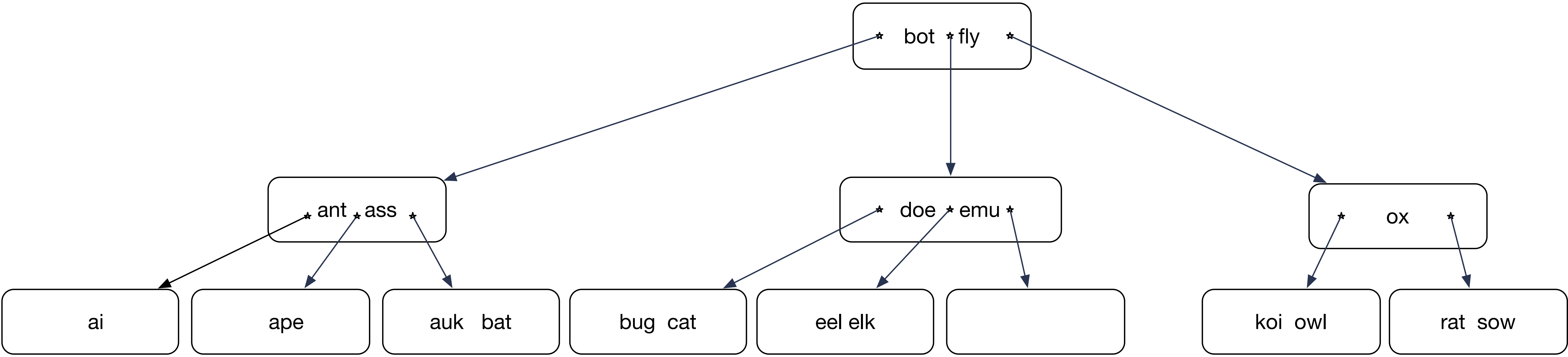
Step 1: Find the key. If it is not in a leaf
Step 2: Determine the key just before it, necessarily in a leaf
Step 3: Interchange the two keys

B-tree



Step 4: Remove the key now from a leaf

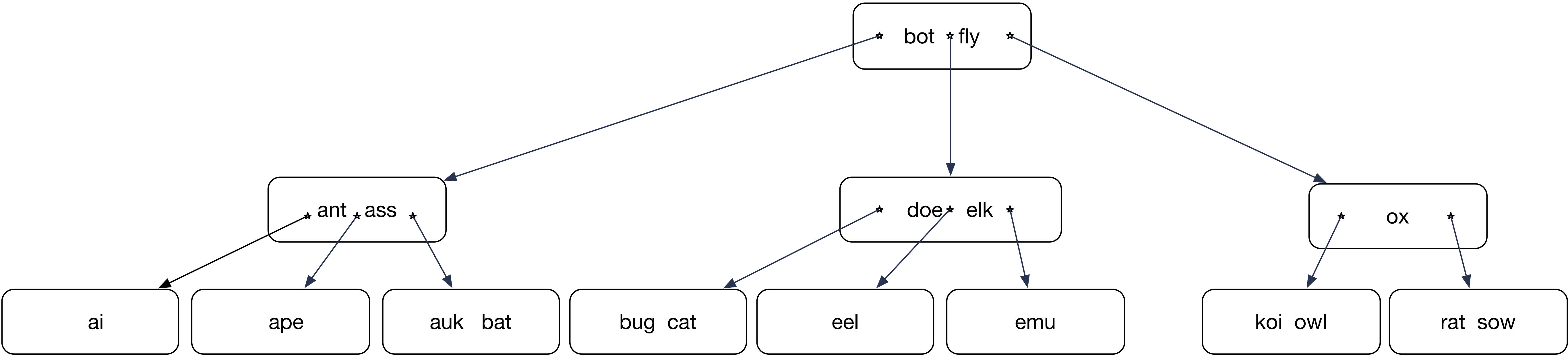
B-tree



This causes an underflow

Remedy the underflow by right rotating from the sibling

B-tree



bot fly

ant ass

doe elk

ox

ai

ape

auk bat

bug cat

eel

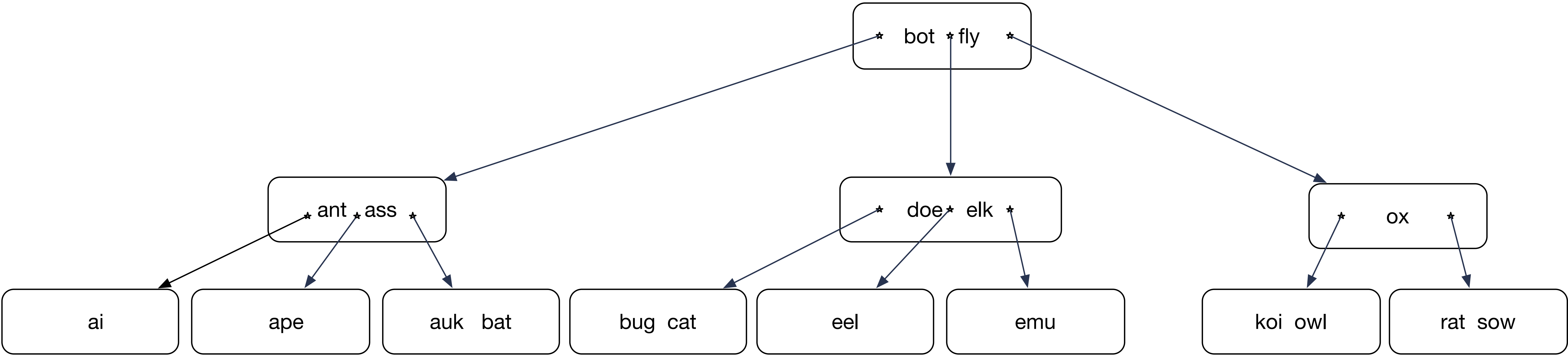
emu

koi owl

rat sow

Everything is now in order

B-tree


bot fly

ant ass

doe elk

ox

ai

ape

auk bat

bug cat

eel

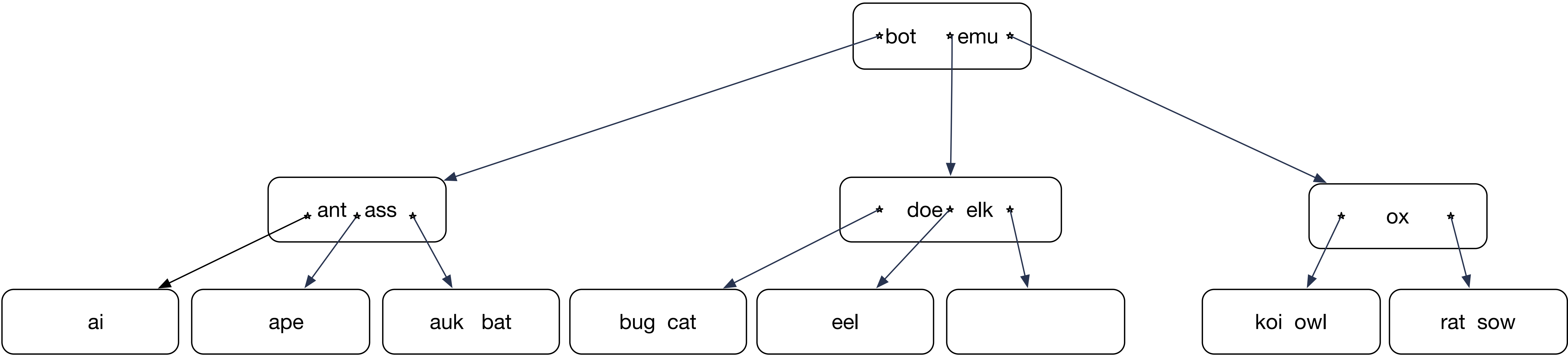
emu

koi owl

rat sow

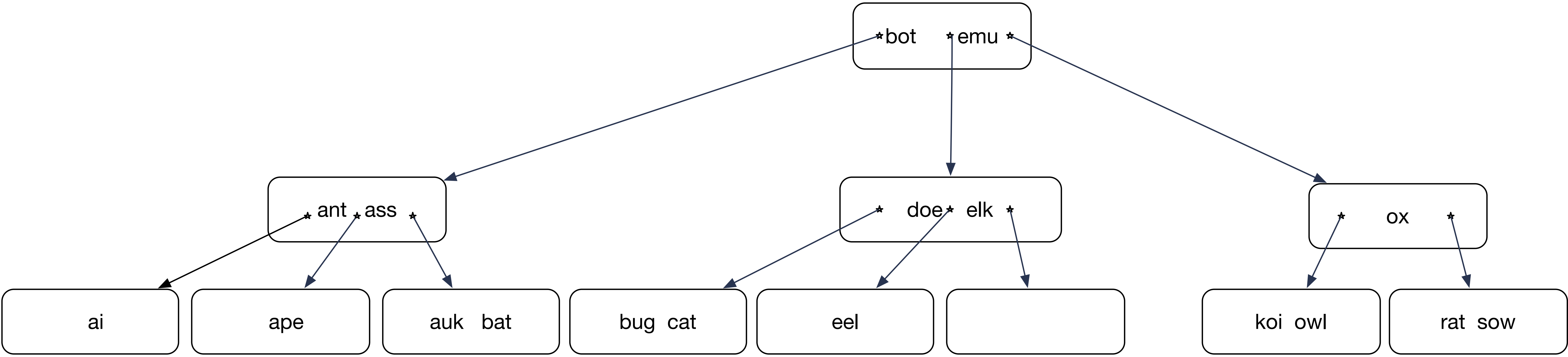
Now delete fly

B-tree



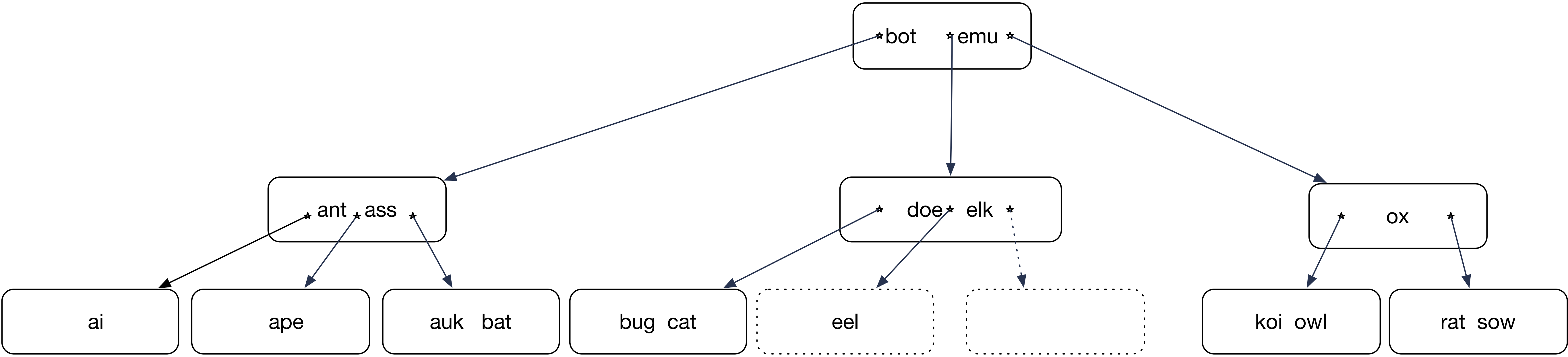
Switch “fly” with “emu”
remove “fly” from the leaf
Again: underflow

B-tree



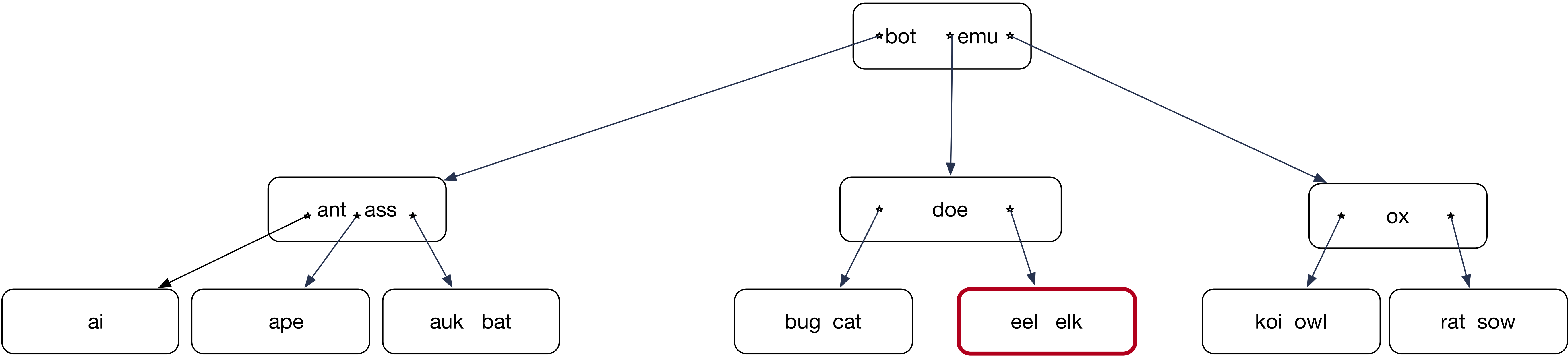
Cannot left-rotate: There is no left sibling
Cannot right-rotate: The right sibling has only one key
Need to merge: Combine the two nodes by bringing down “elk”

B-tree

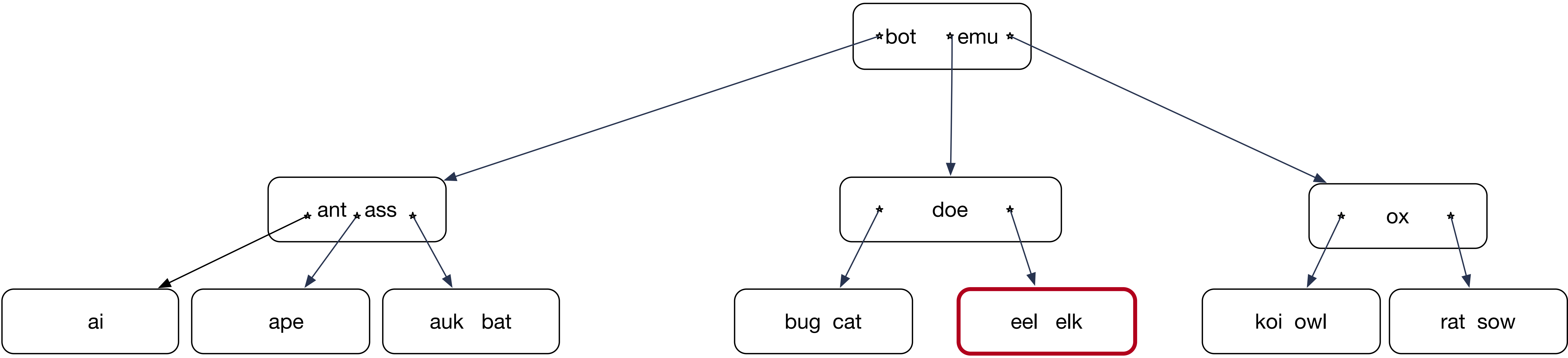


We can merge the two nodes because the number of keys combined is less than 2 *k*

B-tree

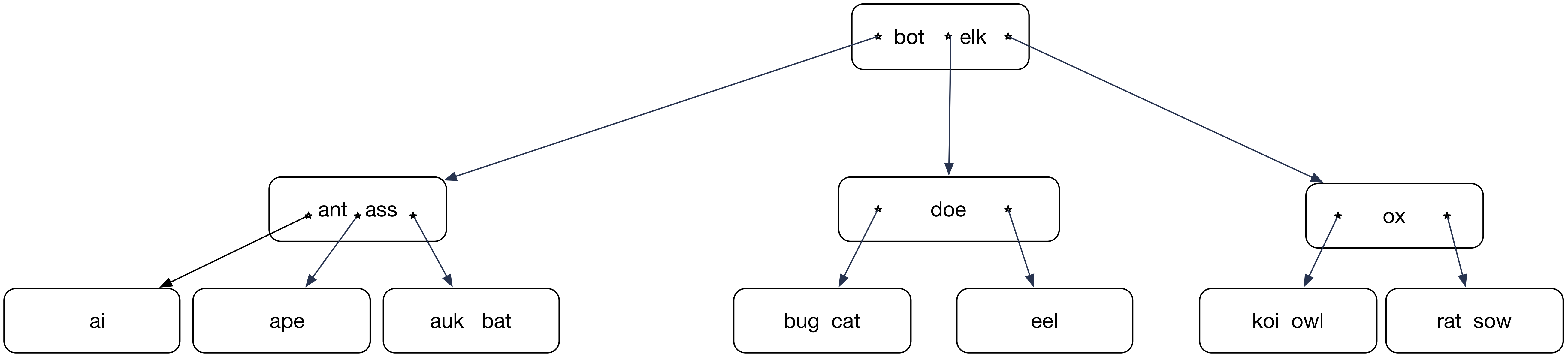

bot emu
ant ass
doe
ox
ai
ape
auk bat
bug cat
eel elk
koi owl
rat sow

B-tree



Delete “emu”

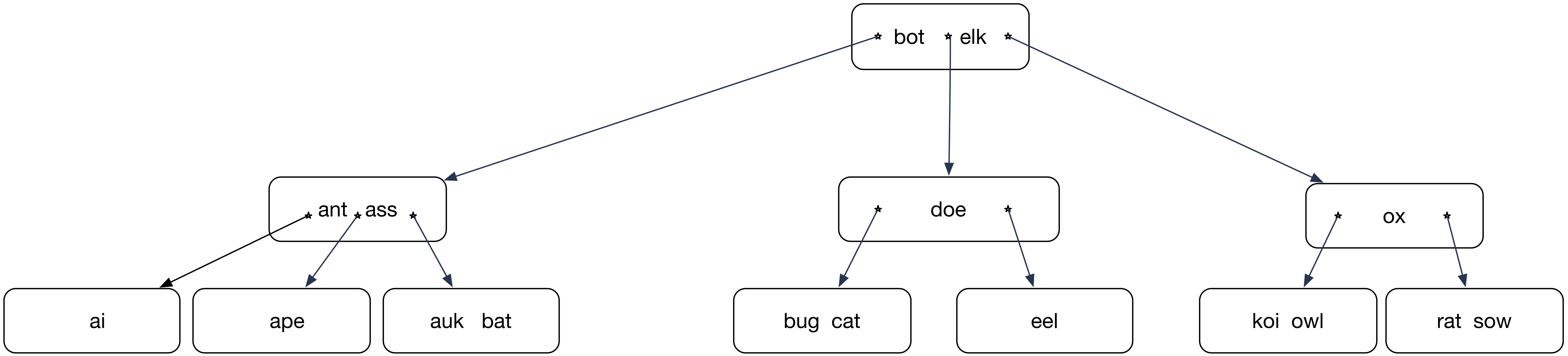
B-tree



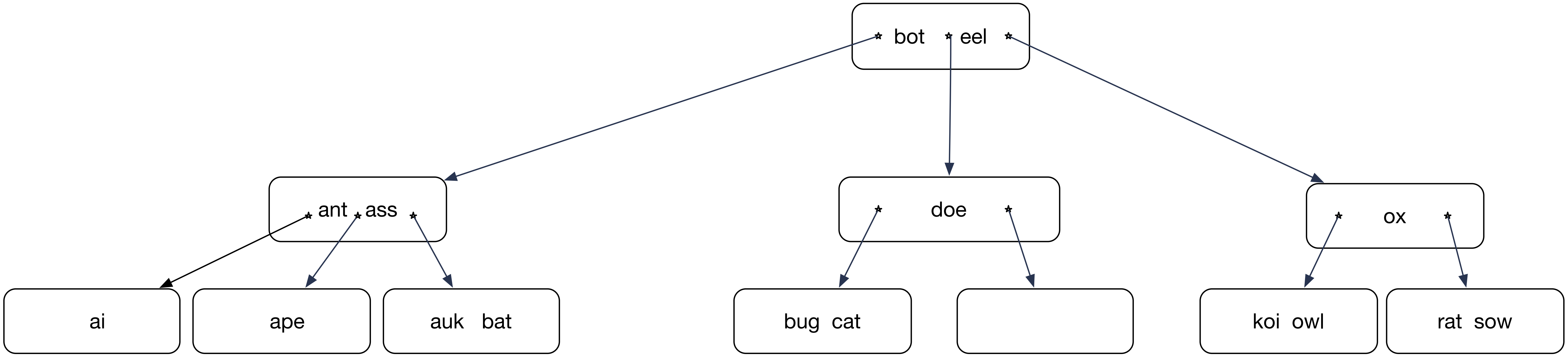
Switch predecessor, then delete from node

B-tree

Now delete “elk”

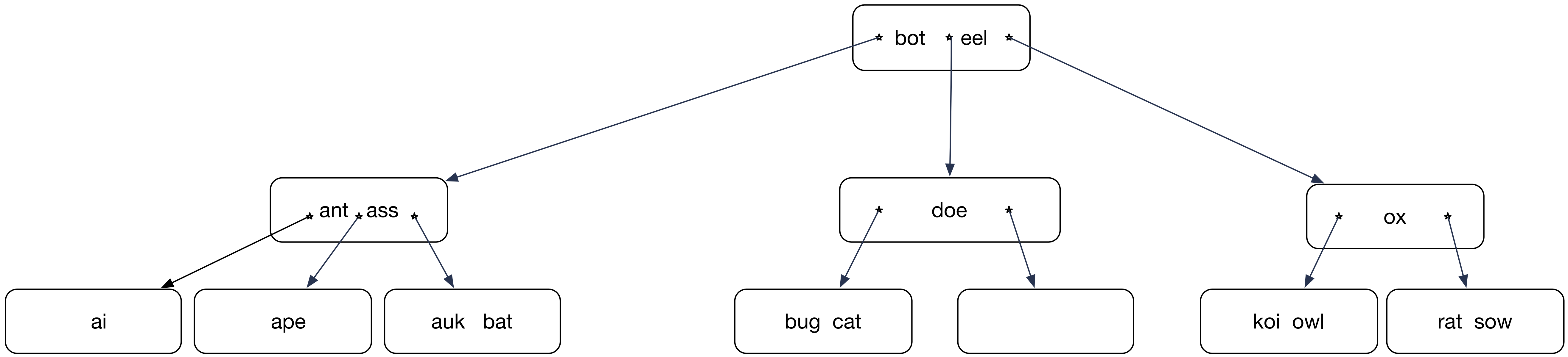


B-tree



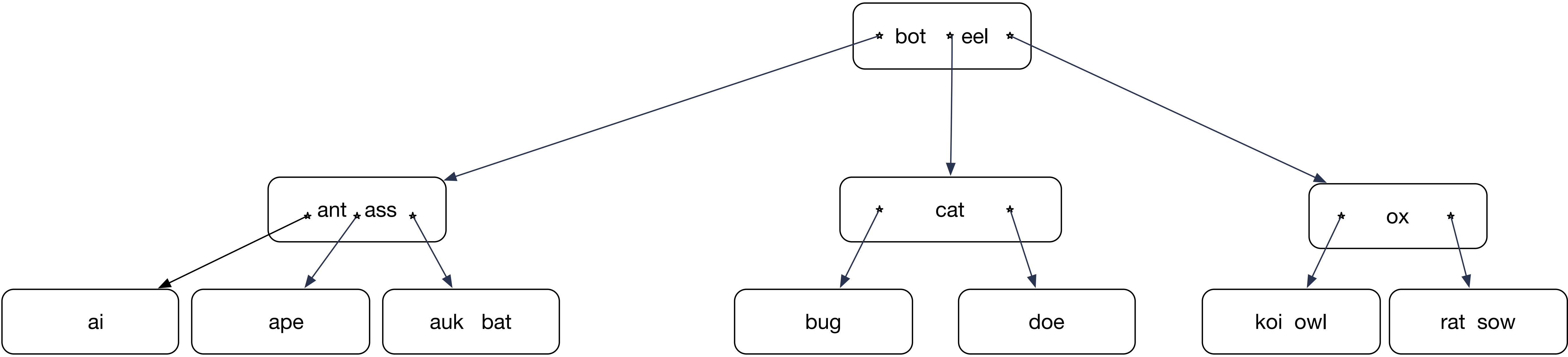
Results in an underflow

B-tree



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

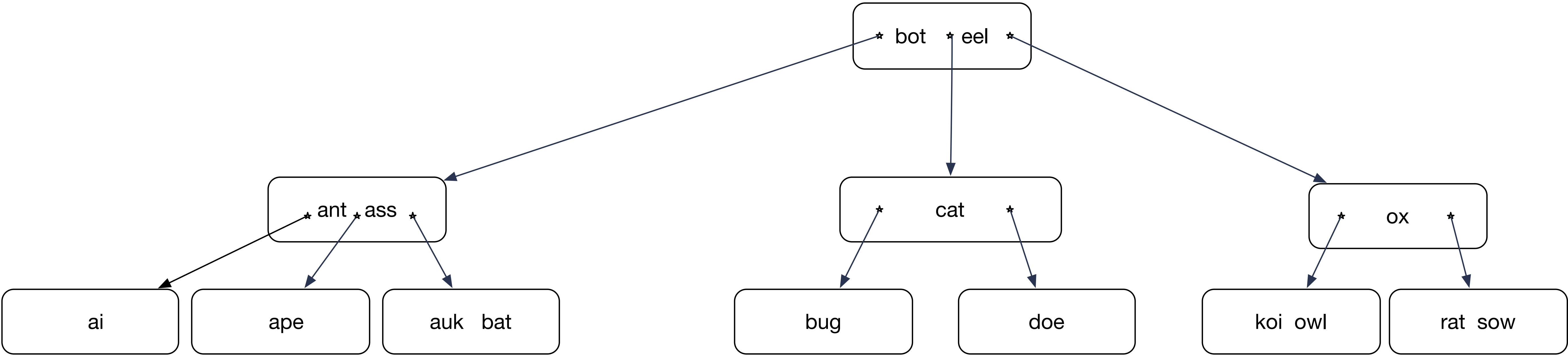
B-tree



Result after left-rotation

B-tree

“Now delete “eel”



bot eel

ant ass

cat

ox

ai

ape

auk bat

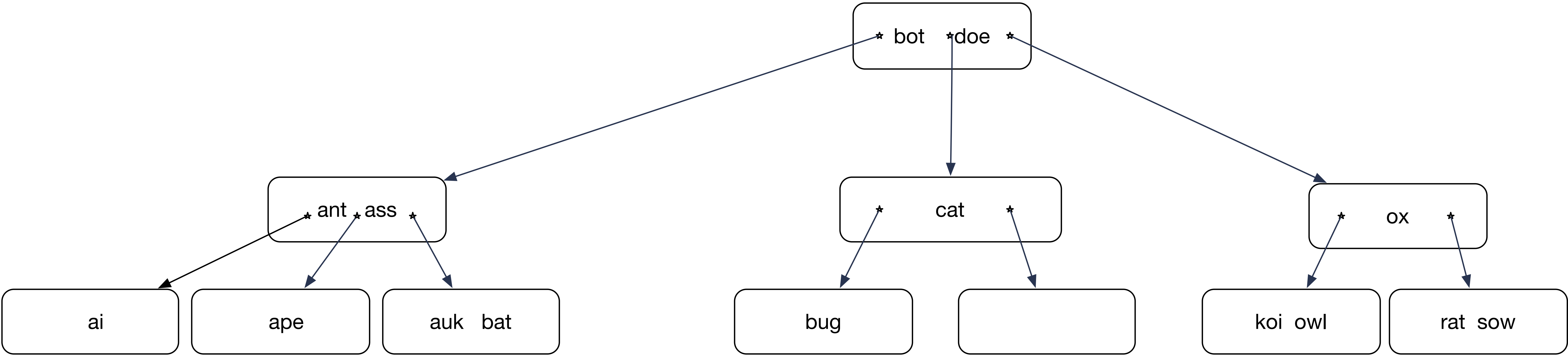
bug

doe

koi owl

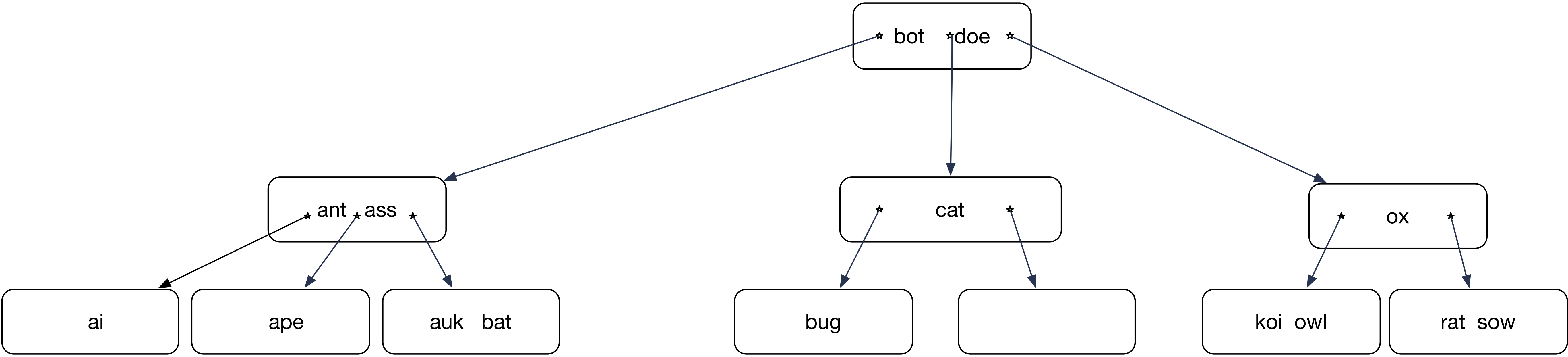
rat sow

B-tree



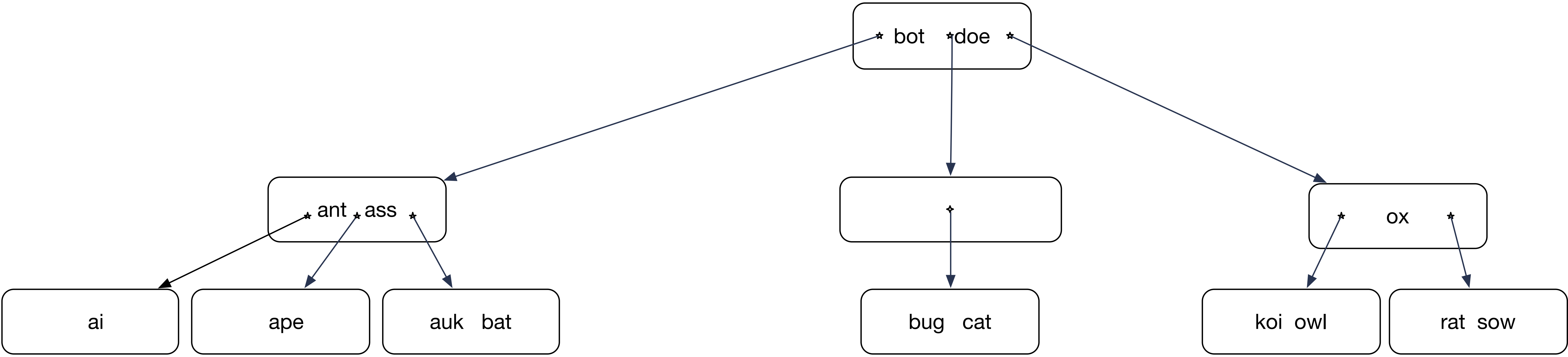
Interchange "eel" with its predecessor
Delete "eel" from leaf:
Underflow

B-tree



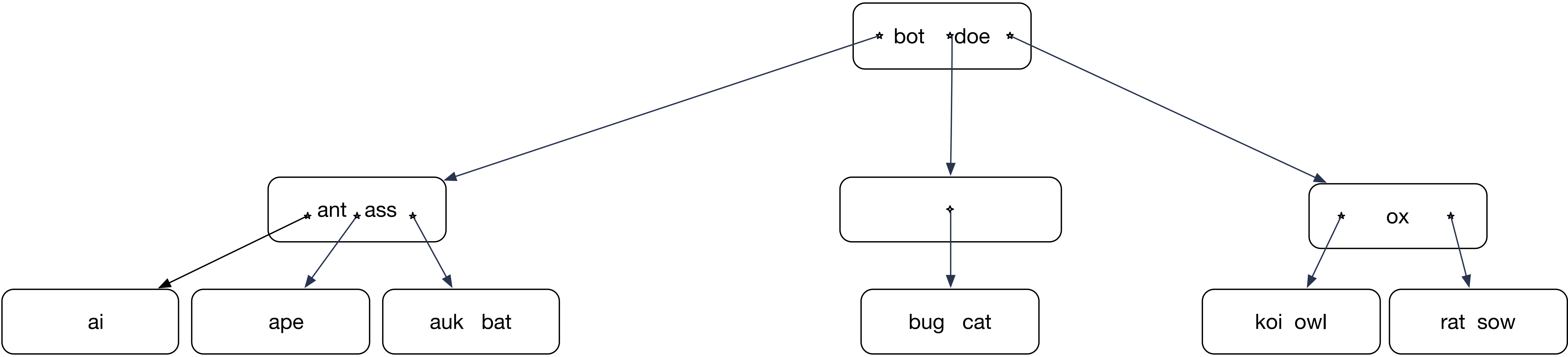
Need to merge

B-tree



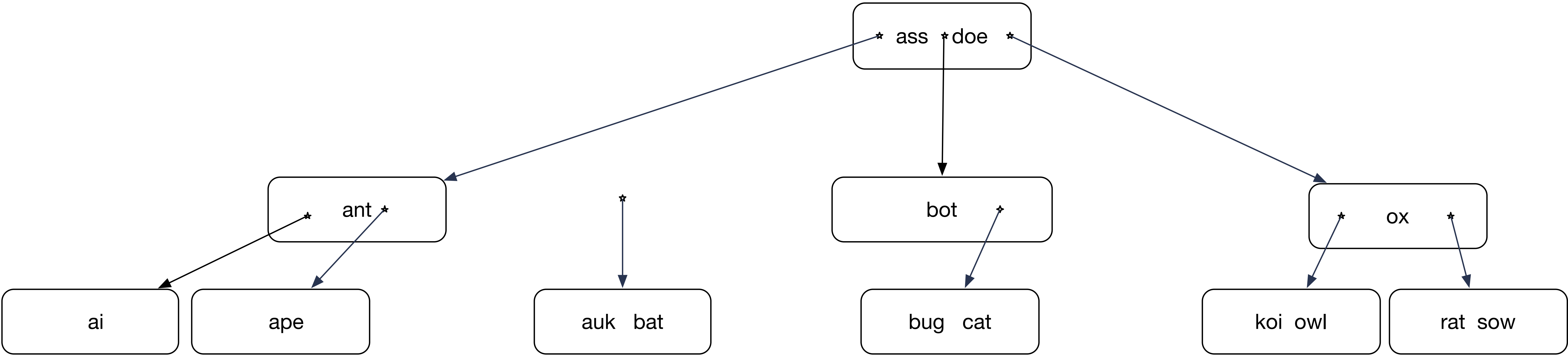
Merge results in another underflow
Use right rotate
(though merge with right sibling
is possible)

B-tree



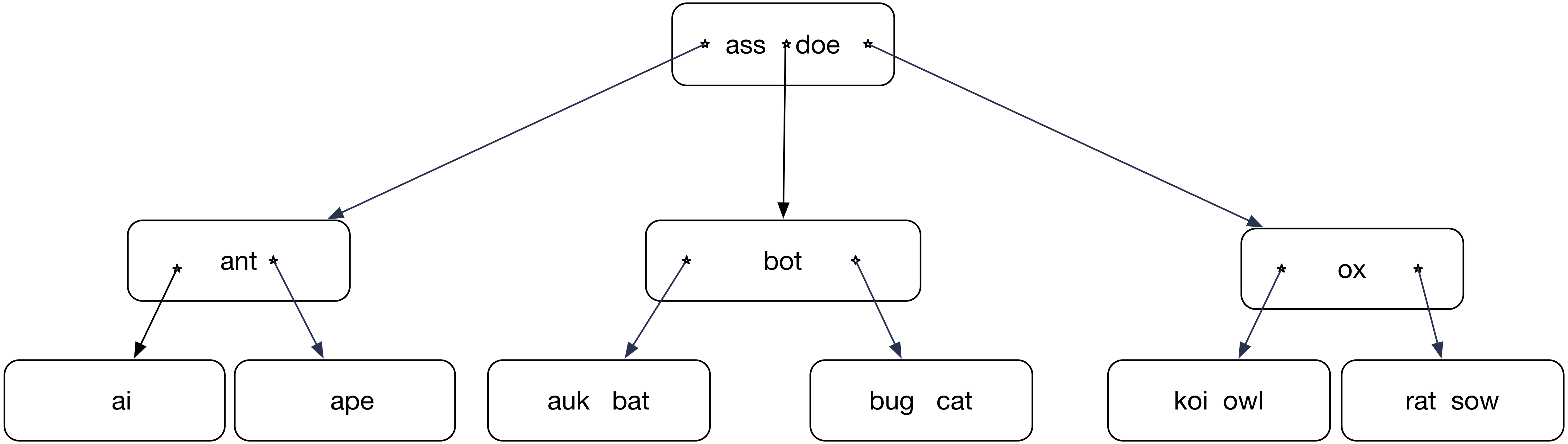
“ass” goes up, “bot” goes down
One node is reattached

B-tree



Reattach node

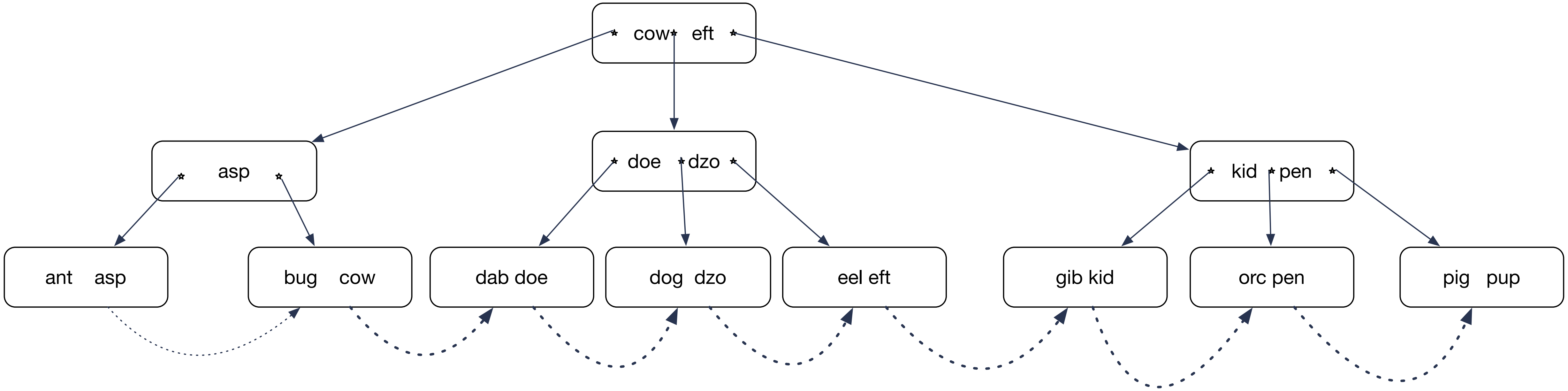
B-tree



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

Linear 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

Linear 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

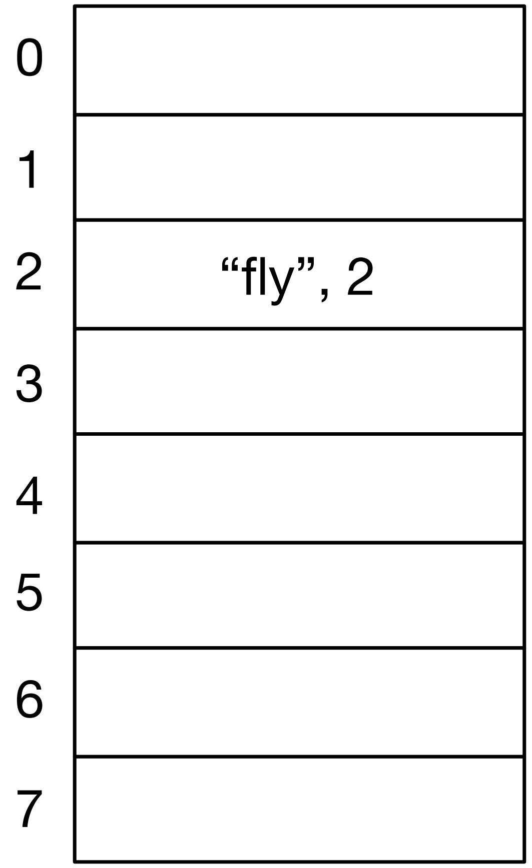
Linear Hashing

- 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”



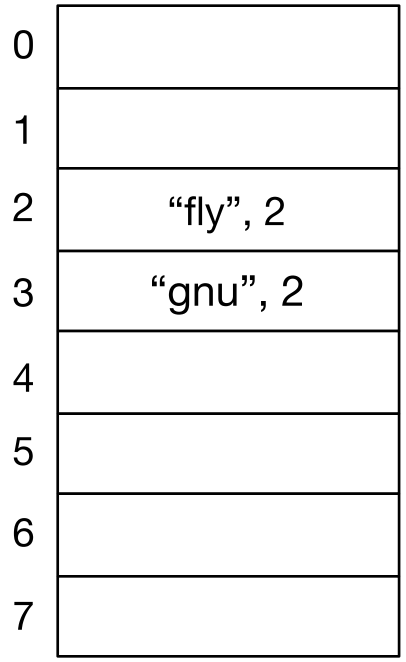
0	
1	
2	“fly”, 2
3	
4	
5	
6	
7	

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”

Hashing Example

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

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

Looking for “gnu”

`hash(“gnu”) —> 2`

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

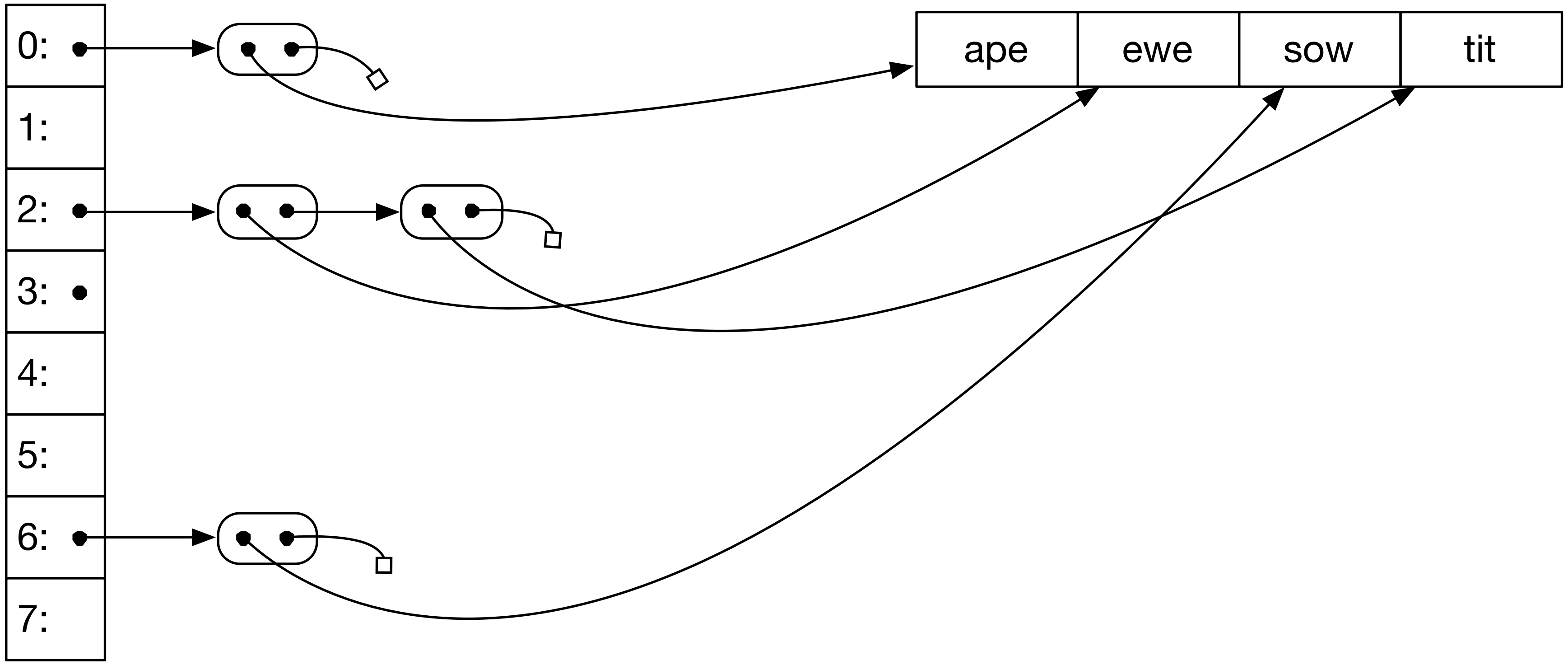
Hashing

- 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

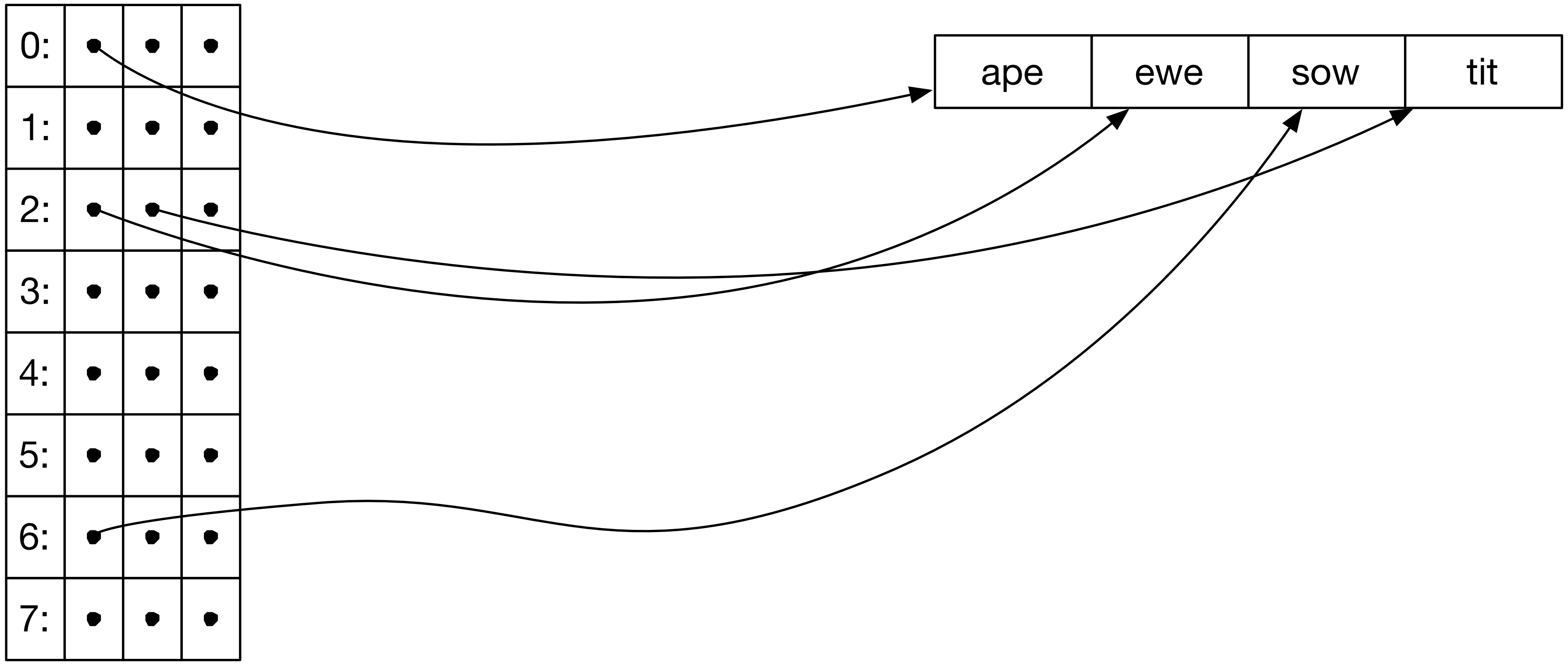
- Chaining
  - Keep data mapped to a location in a “bucket”
    - Can implement the bucket in several ways
      - Linked List

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 α = Space Used / Space Provided
  - Load factor determines performance
  - Idea of extensible hashing:
    - Gracefully add more capacity to a growing hash table

Hashing

Hashing Example

Hashing Example

Hashing Example