

Relational Database System Implementation

CS122 – Lecture 12

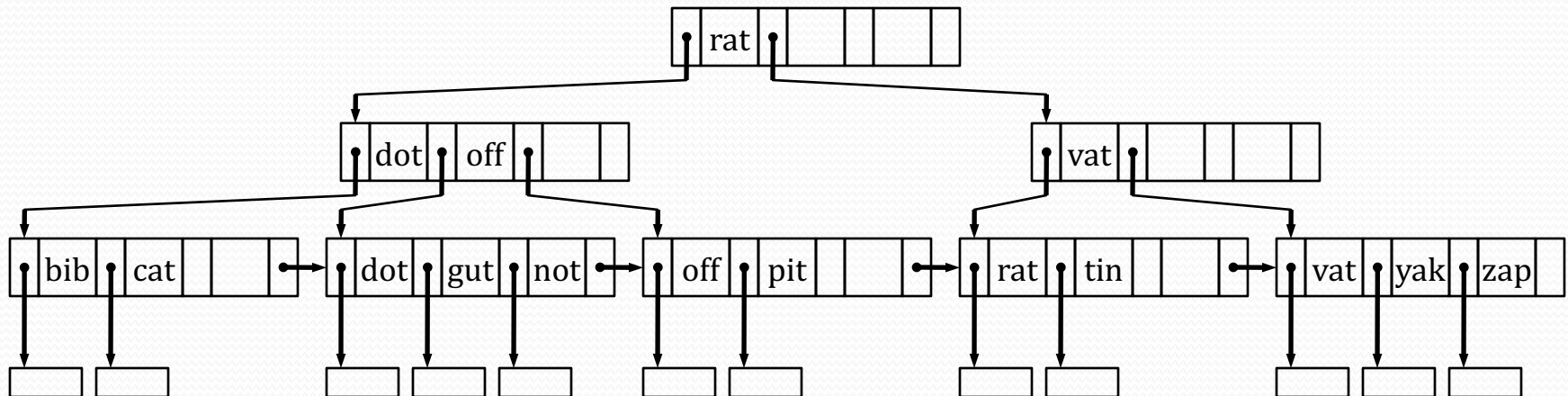
Winter Term, 2018-2019

Last Time: B⁺-Tree Insertion

- Last time, discussed insertion into B⁺-trees:
 - If inserting into a full node, must split the node into two
 - Need to add new node into parent-node's pointer-list
 - May require the parent to be split as well
 - Can even increase tree-depth, if root node is split
- General principle:
 - When a node is split into two, need to promote second node's first key-value up to the parent-node's table
 - e.g. if splitting node N into N and N' , promote $N'.K_1$ up to $\text{parent}(N')$
 - N and N' have the same parent, of course
 - This may also result in the parent node being split

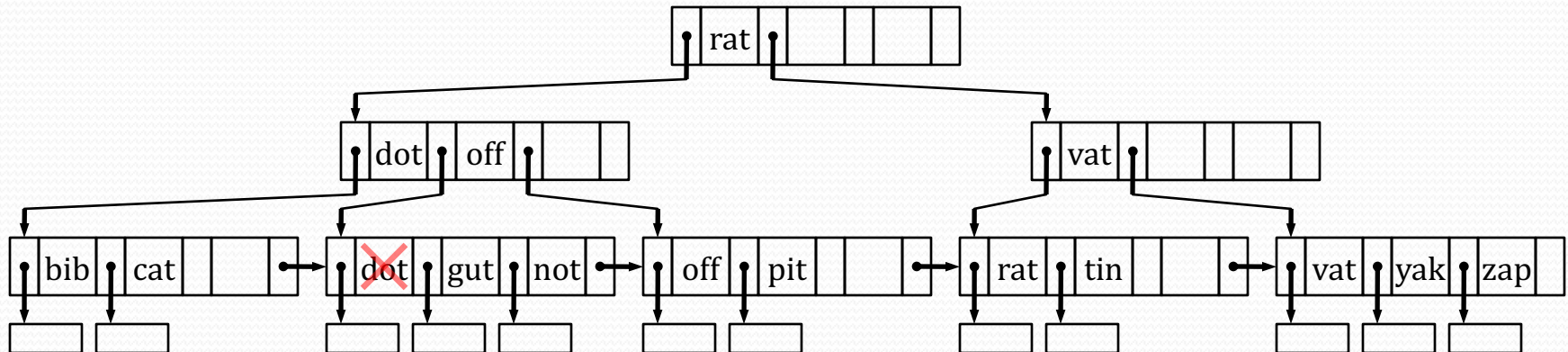
B⁺-Trees: Deletion

- Deletion is much more complicated than insertion
- (Non-root) nodes must always be at least 50% full
- For our tree with $n = 4$:
 - Non-leaf nodes must have at least 2 pointers and 1 key
 - Leaf nodes must have at least 2 pointers and 2 keys
- Often we won't hit the node-size constraints
 - In these cases, deletion is easy

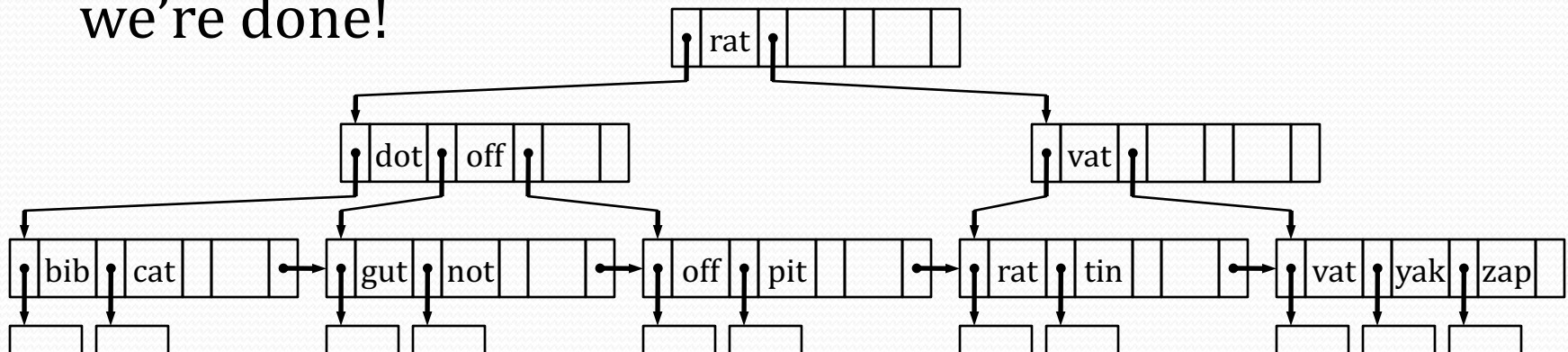


B⁺-Trees: Deletion (2)

- Example: delete “dot” from the index
 - Find leaf-node containing “dot” and remove the record

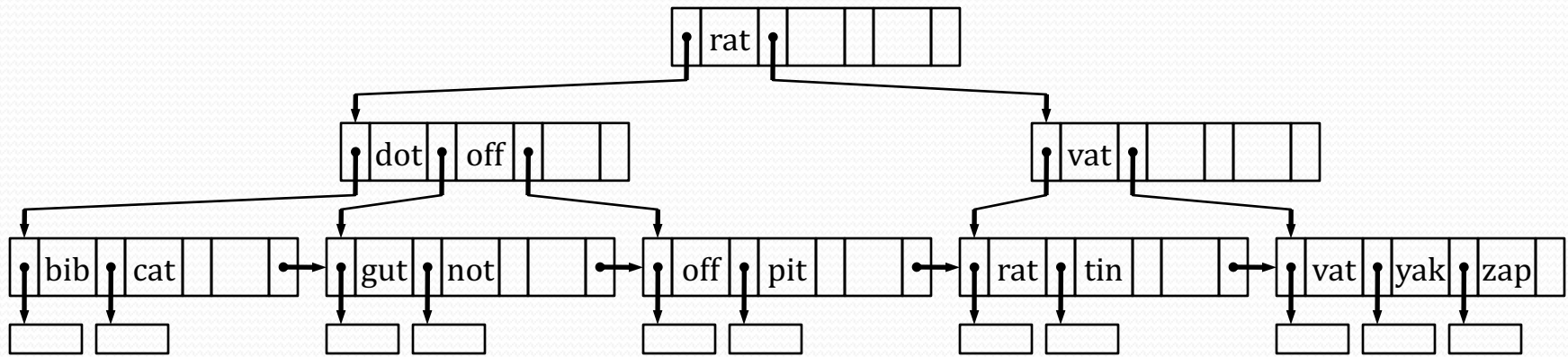


- Removing “dot” doesn’t cause node to be under-full, so we’re done!



B⁺-Trees: Deletion (3)

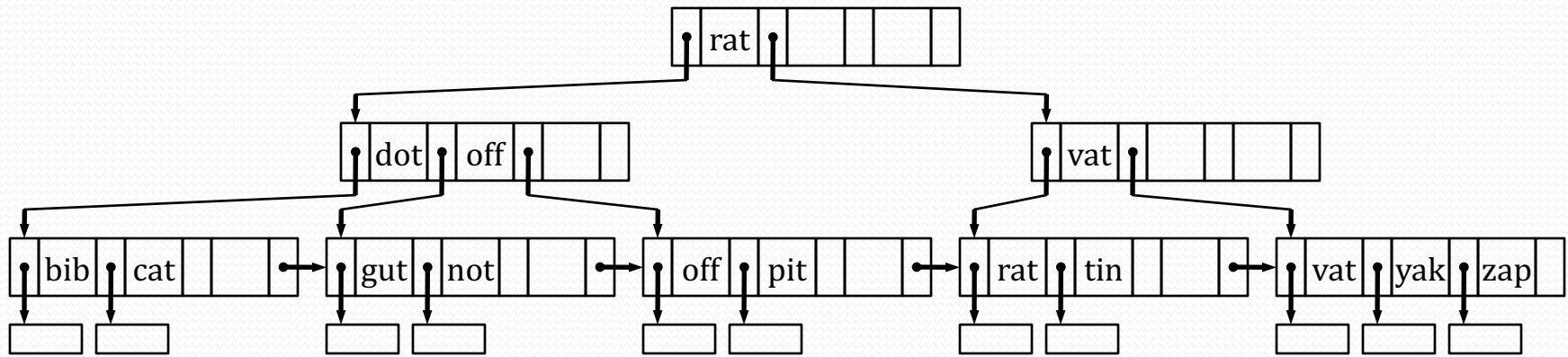
- Our B⁺-tree index now contains a curious situation:



- Value “dot” is no longer in the leaf nodes, but still appears in the non-leaf nodes
- We don’t care about this, as long as our node-fullness requirements are satisfied
 - Doesn’t affect lookups at all

B⁺-Trees: Deletion (4)

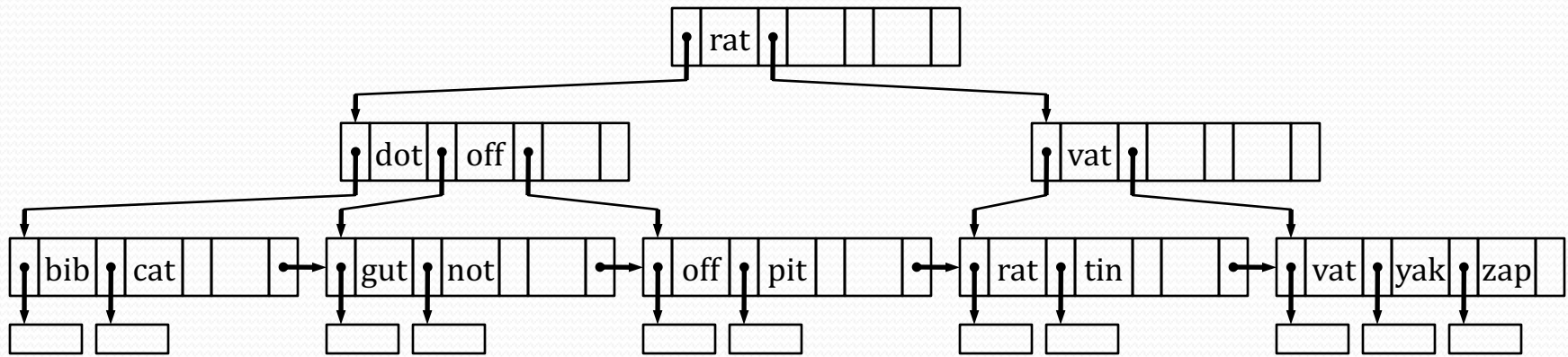
- If a node becomes too empty, we have several choices



- If a node's sibling has extra values, *redistribute* values across both nodes to satisfy space requirements
 - (Sibling nodes must share the same parent node.)
 - e.g. if we delete "tin", can move "vat" left to ensure both nodes have enough entries

B⁺-Trees: Deletion (5)

- If a node becomes too empty, we have several choices



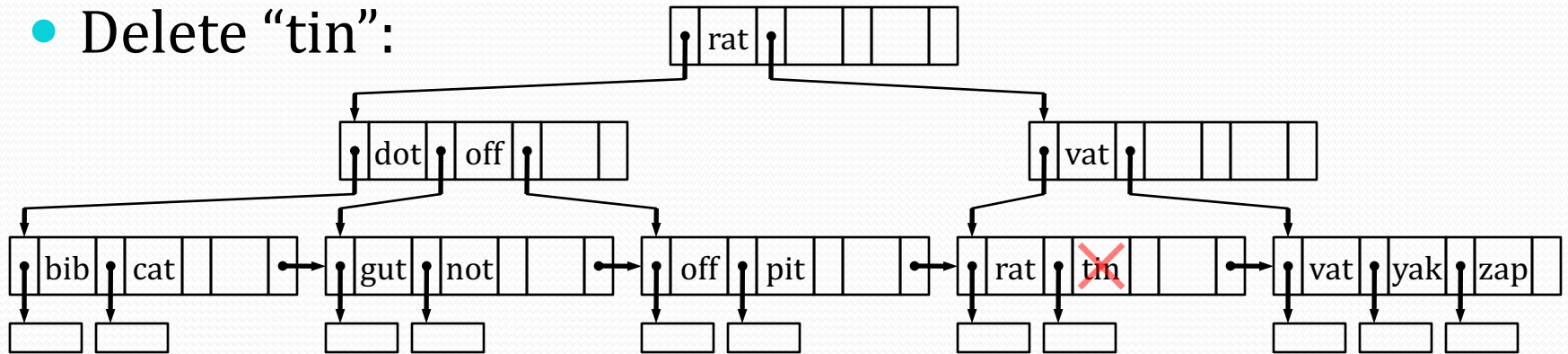
- If a node's sibling is also half-full, could *coalesce* the two nodes together into a single node
 - (Again, sibling nodes must share the same parent node.)
 - e.g. if we delete "gut", can coalesce the leaf-node together with either sibling to produce a single node

B⁺-Trees: Deletion (6)

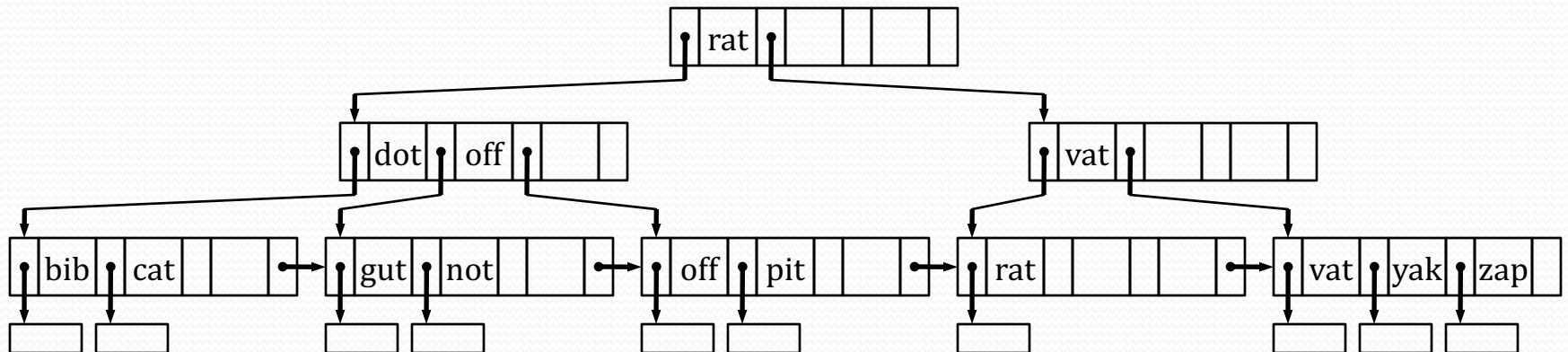
- When we redistribute values between two nodes, or when two nodes are coalesced, parent node(s) are clearly affected!
- Unfortunately, these behaviors are rather complex
 - Due to differences between leaf and non-leaf nodes
 - When deleting/rearranging leaf nodes, updates to parent nodes are more straightforward
 - When deleting/rearranging non-leaf nodes, updates are more involved
- Will examine leaf-node behaviors first, then non-leaf node behaviors

Delete at Leaf: Redistribute

- Delete “tin”:

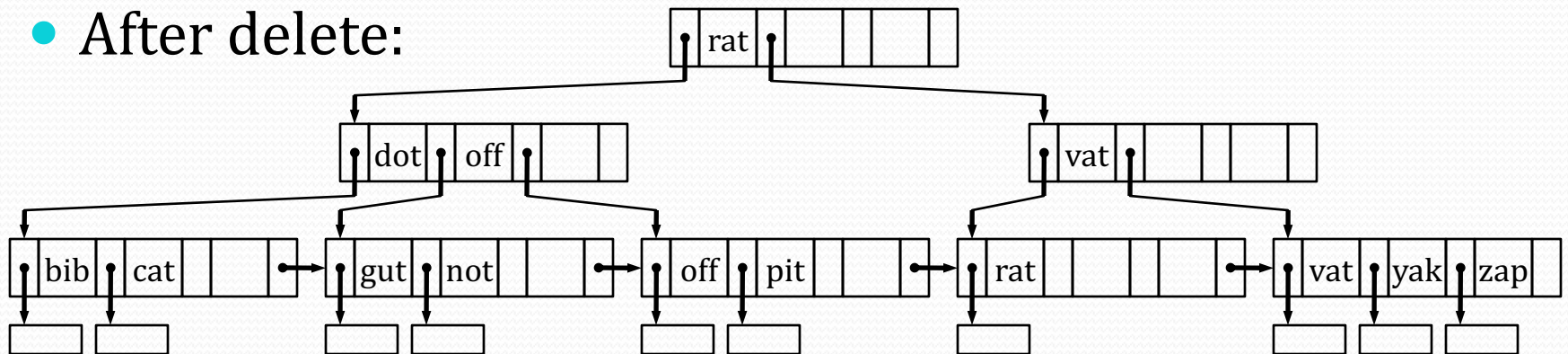


- The leaf-node is now under-full!
 - Can't coalesce with sibling since sibling is completely full

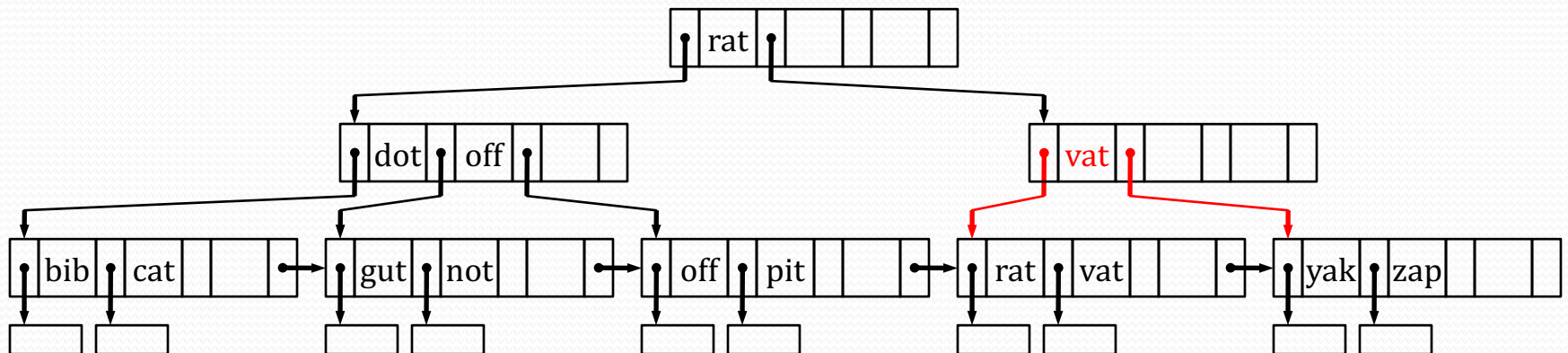


Delete at Leaf: Redistribute (2)

- After delete:

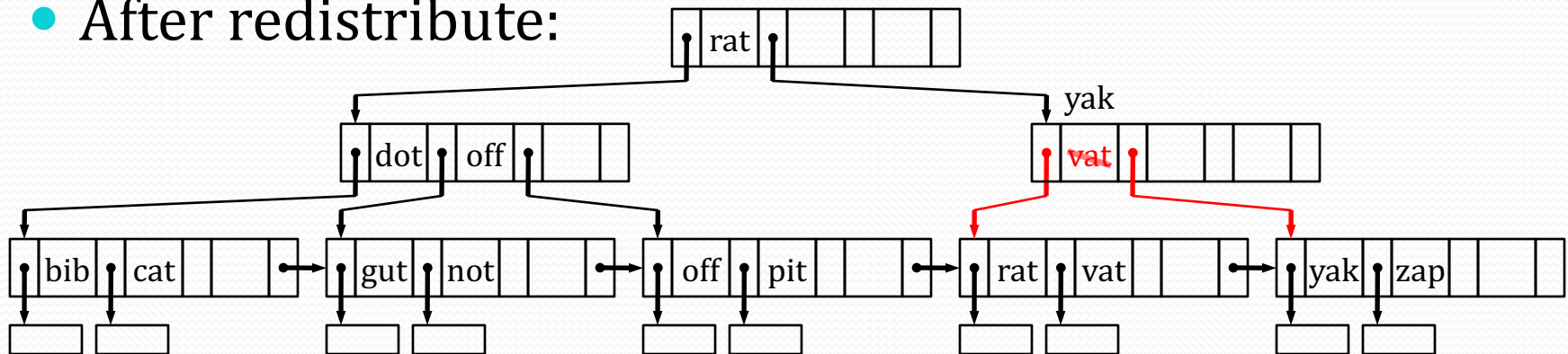


- Move an entry from sibling into the under-full node:



Delete at Leaf: Redistribute (3)

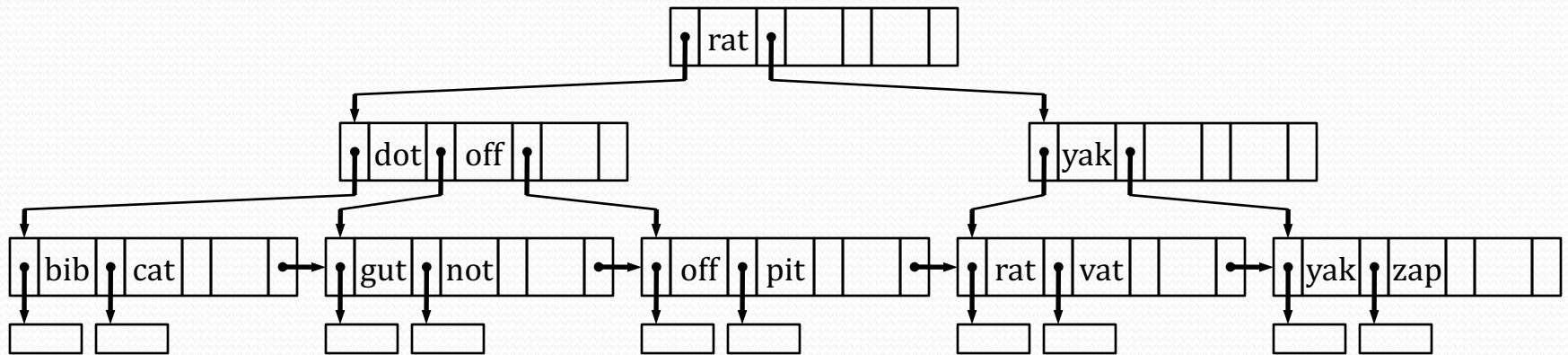
- After redistribute:



- Parent-node entry is clearly wrong now
- Given a pair of sibling leaf-nodes N and N' :
 - N is the immediate predecessor to N'
 - Redistributing values between N and N'
 - Either moving a value from N to N' , or from N' to N
 - In parent, replace key between N and N' with $N'.K_1$

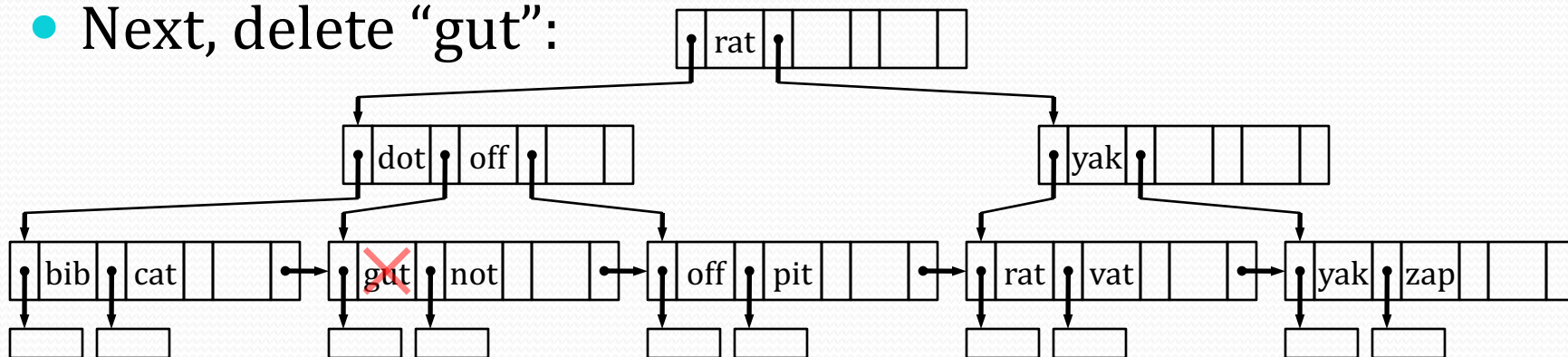
Delete at Leaf: Redistribute (4)

- After redistribute and fix-up of parent node:

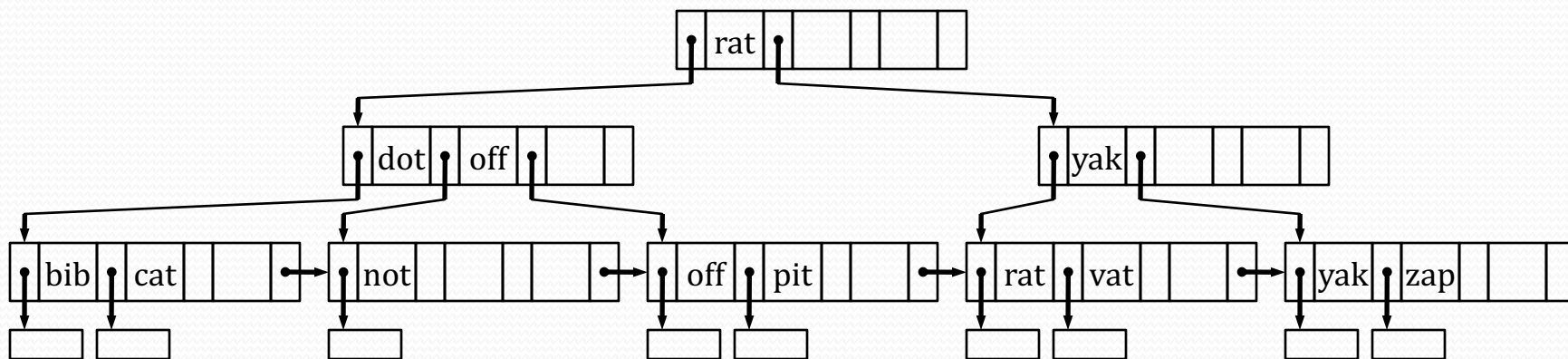


Delete at Leaf: Coalesce

- Next, delete "gut":

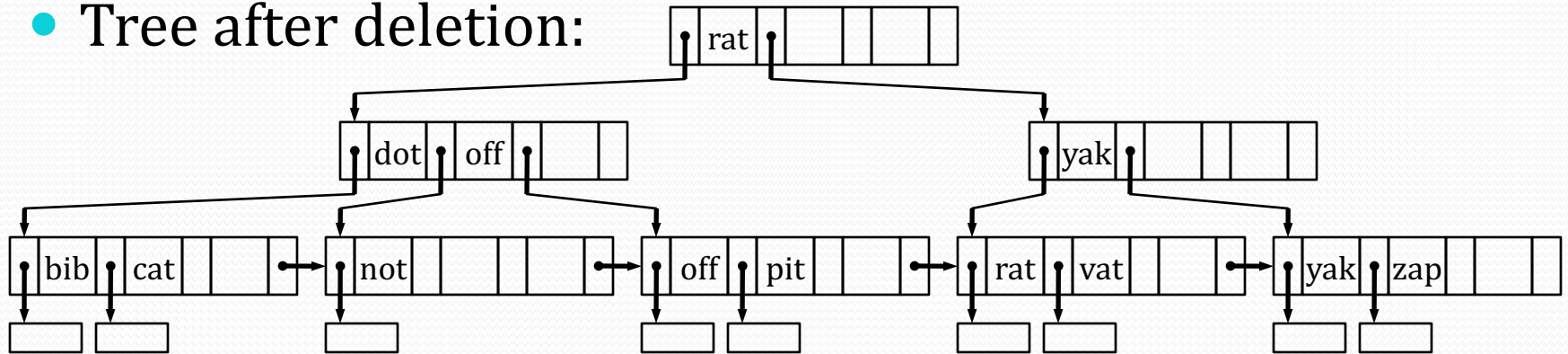


- The leaf-node is clearly now under-full:

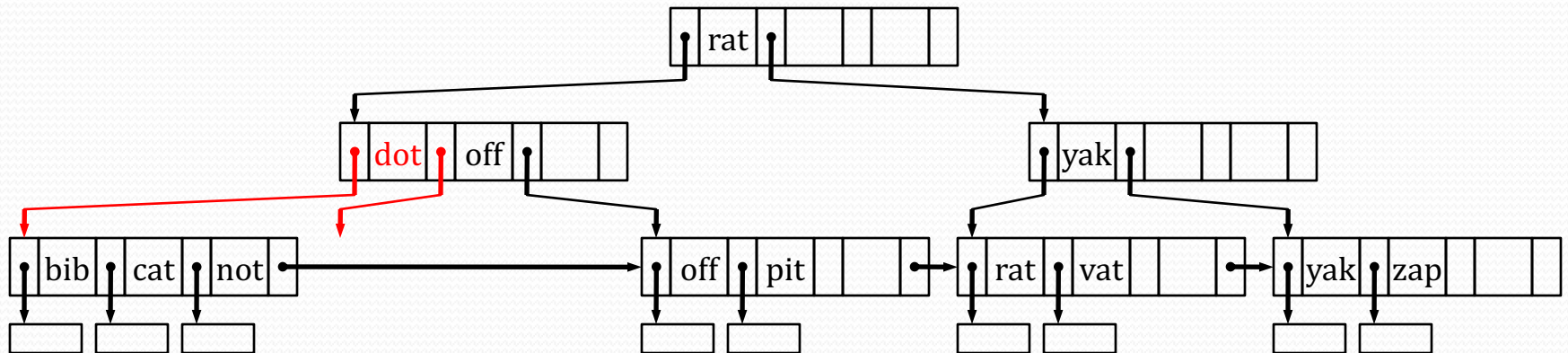


Delete at Leaf: Coalesce Left

- Tree after deletion:

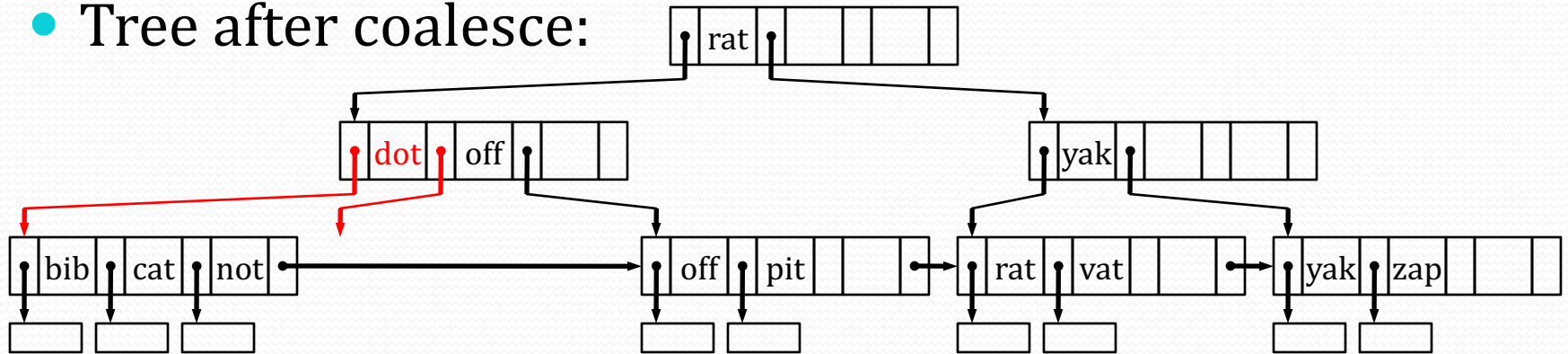


- Coalesce leaf-node with its left sibling:

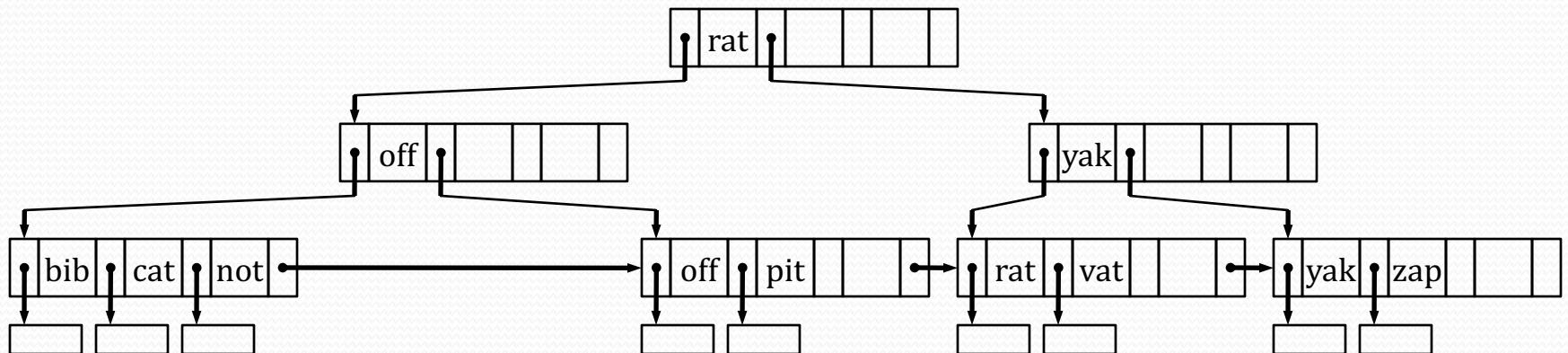


Delete at Leaf: Coalesce Left (2)

- Tree after coalesce:

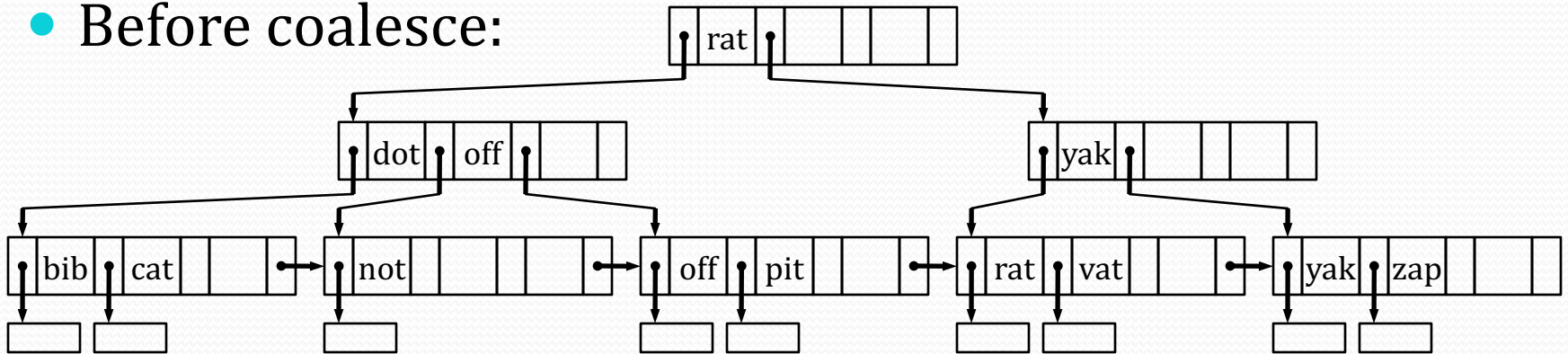


- Clearly need to modify parent-node contents
 - No longer need “dot” entry, or pointer to deleted node

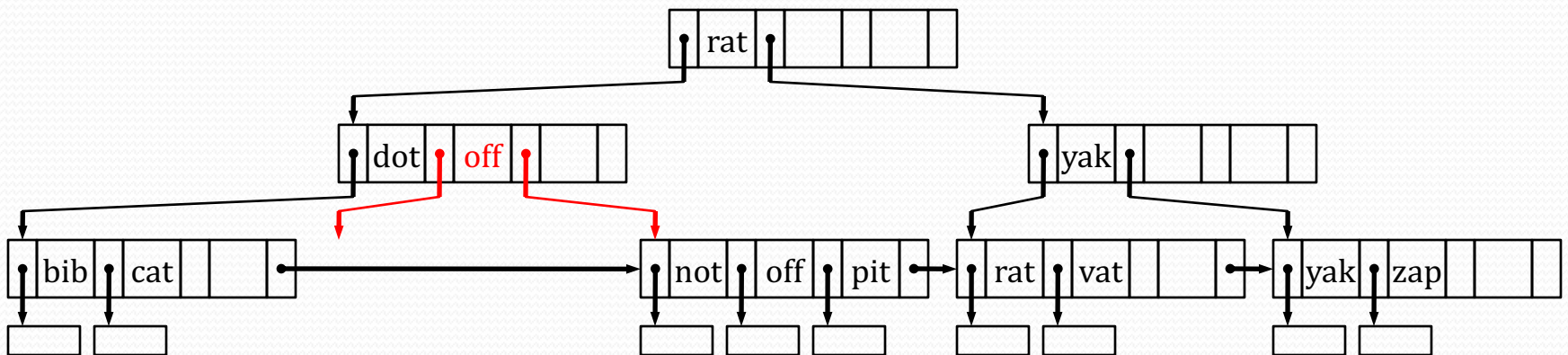


Delete at Leaf: Coalesce Right

- Before coalesce:

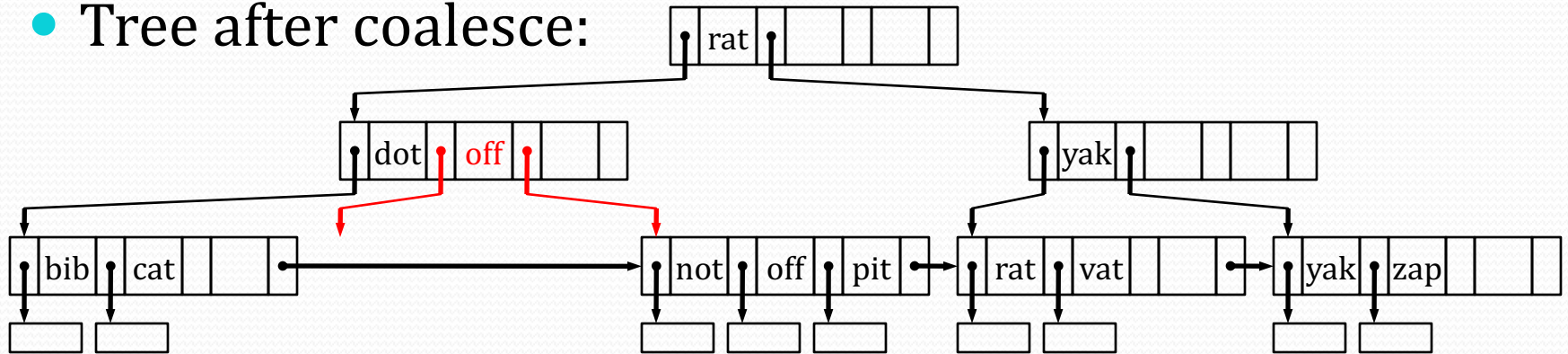


- This time, coalesce leaf-node with its *right* sibling:

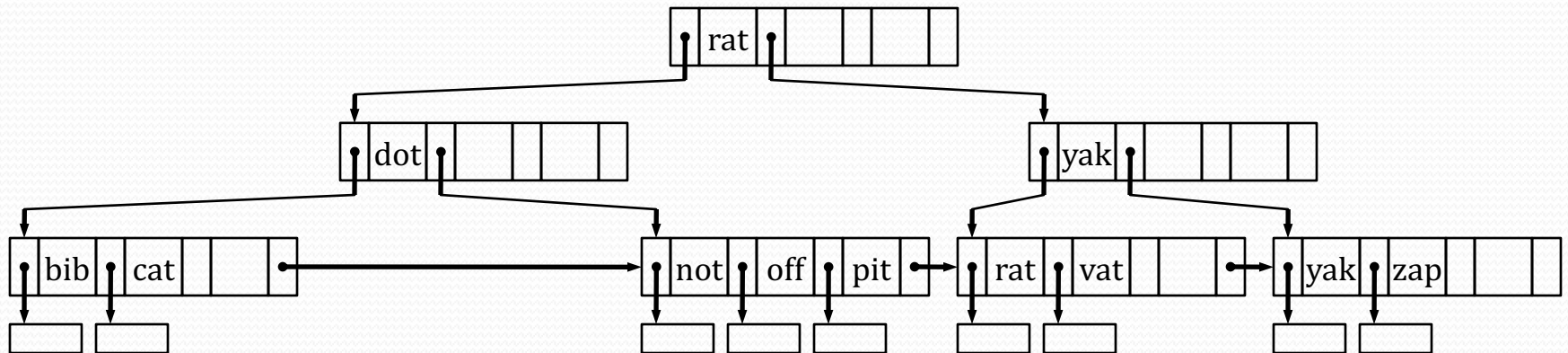


Delete at Leaf: Coalesce Right (2)

- Tree after coalesce:

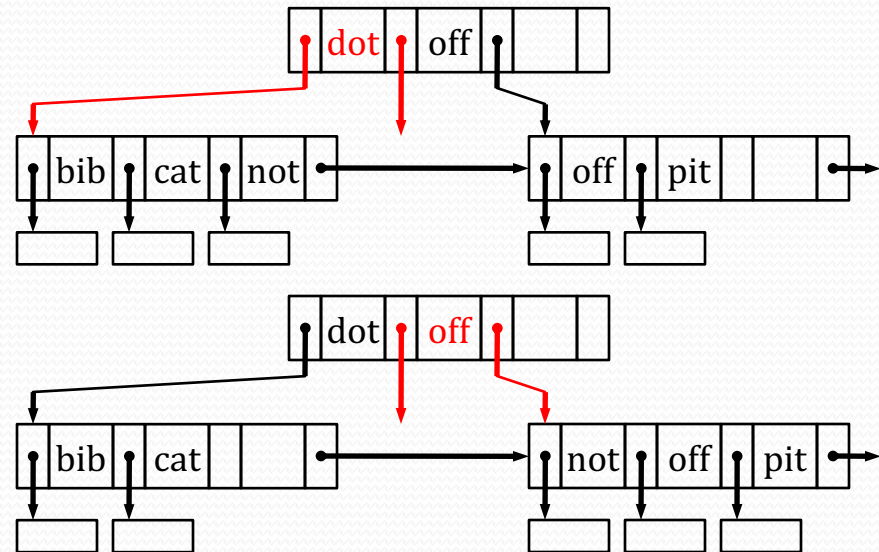


- This time, need to delete “off” from parent, not “dot”



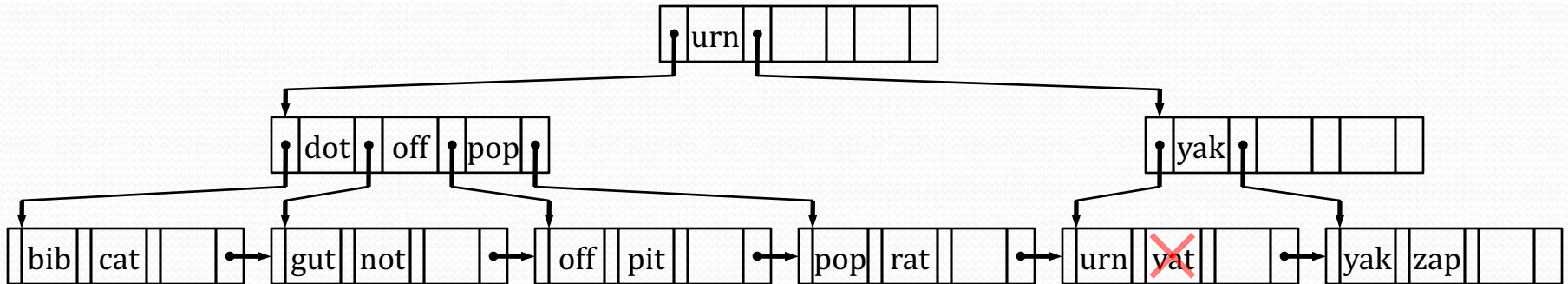
Delete at Leaf: Coalesce Nodes

- When coalescing two sibling leaf-nodes N and N' :
 - N is the immediate predecessor to N'
- The two siblings will be separated by a key-value in their shared parent-node
 - Coalesce the two sibling nodes into one, then remove the key in their parent that separated these two nodes
 - (along with the pointer to the now-deleted node)



Deletes at Internal Nodes

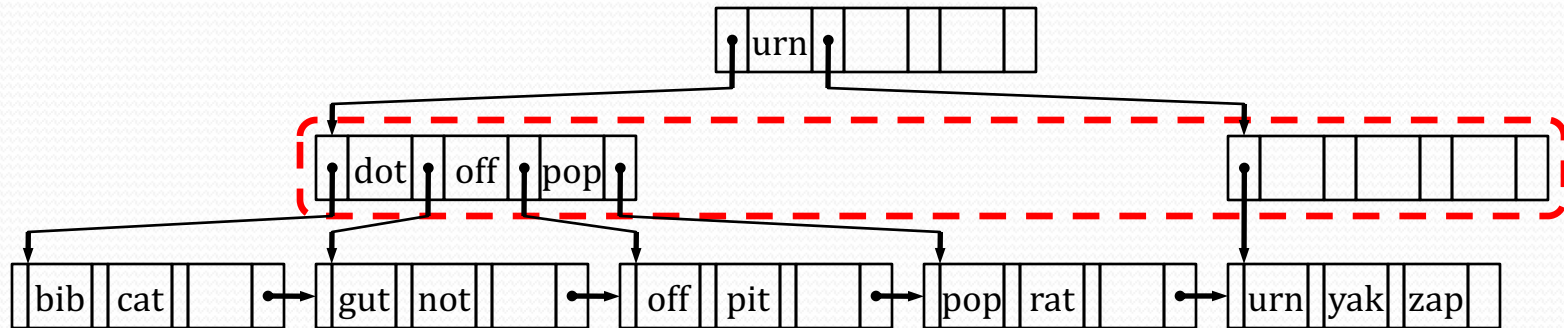
- Another B⁺-tree with more nodes:



- Won't show record pointers, etc. due to space limitations
- Delete "vat" from the index
 - Leaf node becomes too empty, but it has a sibling
 - Can't redistribute values: sibling doesn't have enough
 - Coalesce node with its sibling (we know how to do that)

Deletes at Internal Nodes

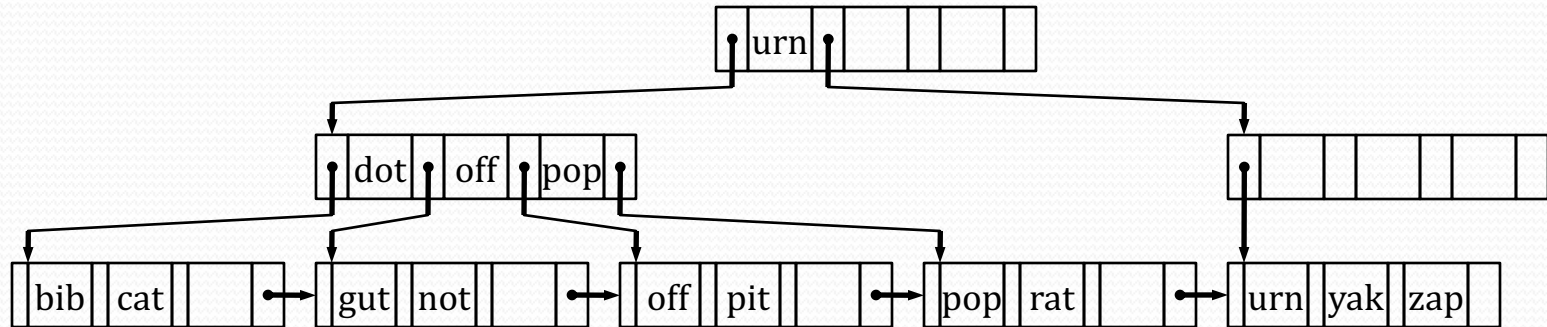
- After deleting “vat”, coalescing leaf-nodes, and removing intervening key and pointer:



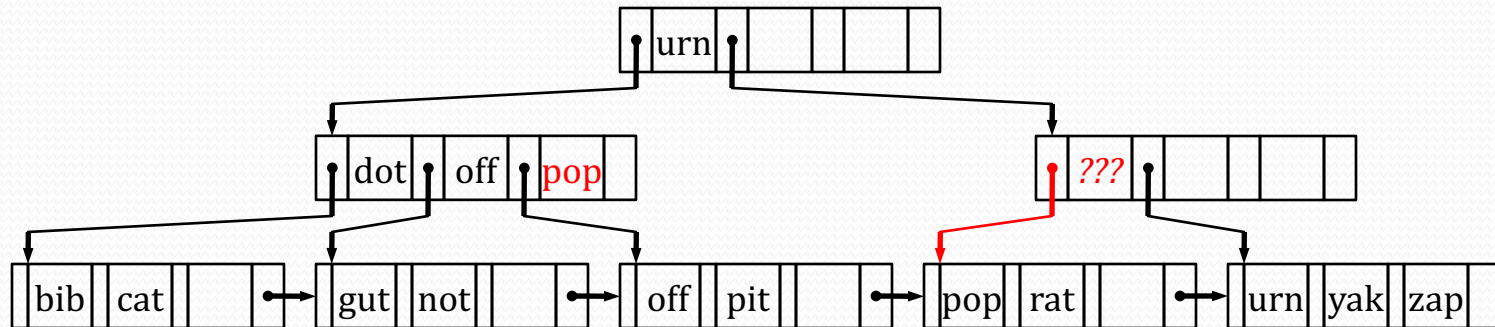
- Problem: now internal node is too empty
 - For $n = 4$, internal nodes must have at least 2 pointers
- Can't coalesce in this situation:
 - Left sibling already has 4 pointers
 - Can only redistribute values

Deletes at Internal Nodes (2)

- Need to redistribute key/pointer values:

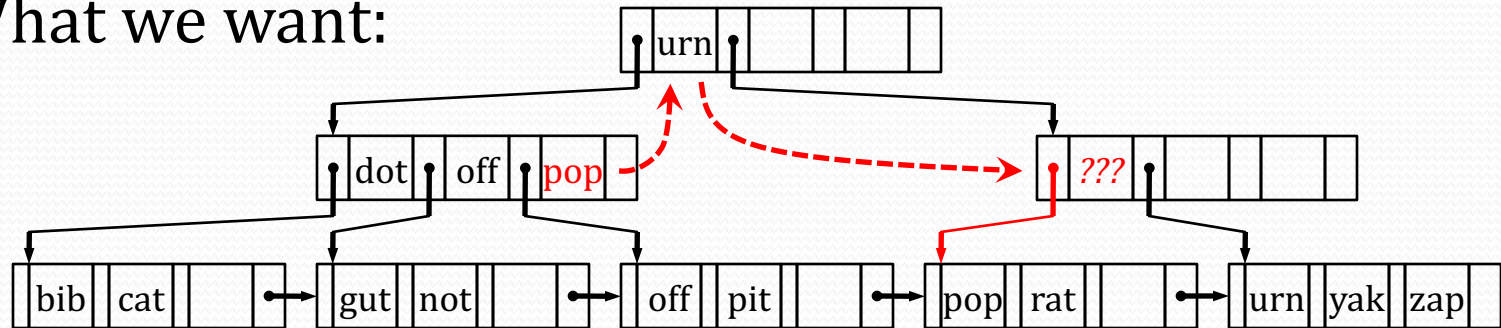


- In this situation, would like right sibling to point to both “pop/rat” leaf, and “urn/yak/zap” leaf
 - Can move rightmost pointer in left node to right node



Deletes at Internal Nodes (3)

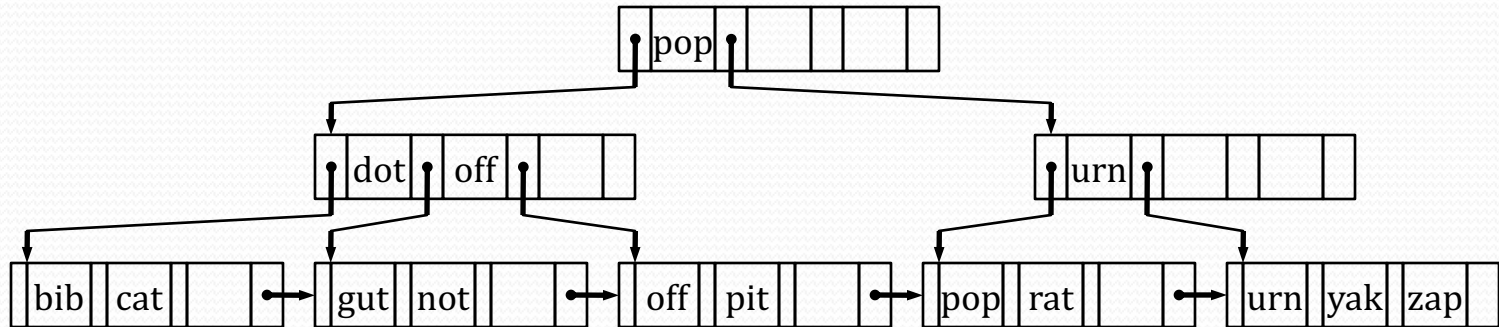
- What we want:



- Where do we move the search-key values?
- Can't move "pop" straight across to right sibling!
 - Right sibling should get "urn" as its key
- Move "pop" to parent node, "urn" to right sibling
- General principle:
 - When redistributing pointers between internal nodes, keys must be rotated through the parent node

Deletes at Internal Nodes (4)

- Final result:

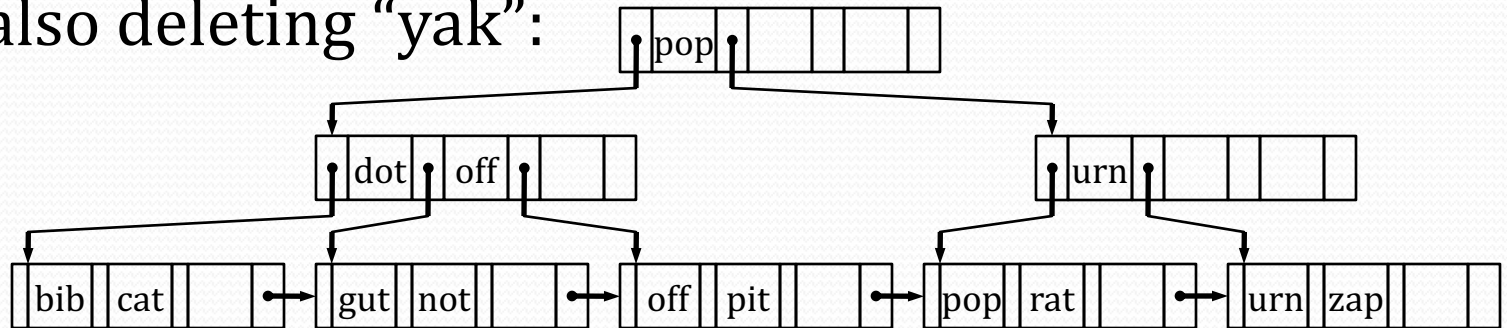


Redistribute across Internal Nodes

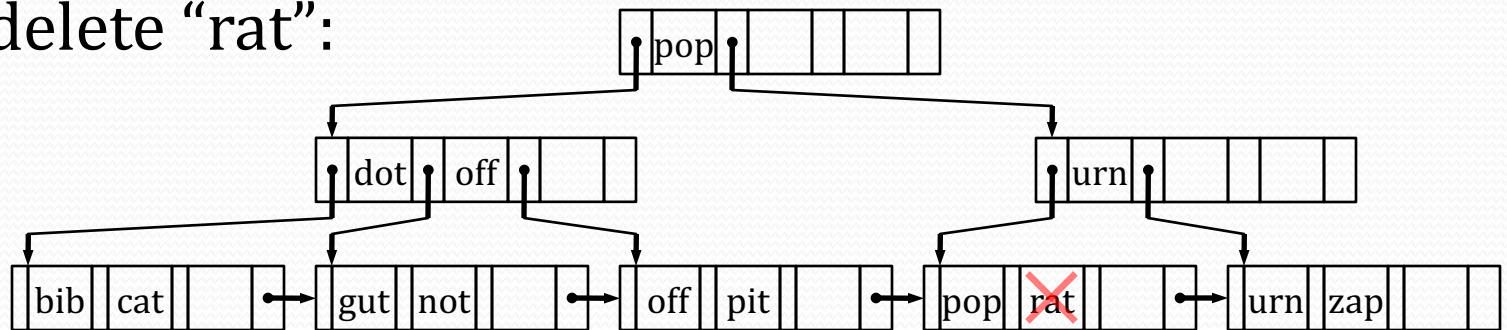
- Redistributing pointers between sibling internal nodes:
 - As with leaf nodes, siblings are separated by a single key in their shared parent-node
- Let N and N' be sibling internal nodes
 - N is immediate predecessor to N'
 - K' is the search-key value between N and N' pointers in parent
- If moving pointer $N.P_m$ to N' (insert before $N'.P_1$):
 - $N'.K_1$ is set to K'
 - $N.K_{m-1}$ replaces K' in parent node
 - Both $N.K_{m-1}$ and $N.P_m$ are removed from N
- If moving pointer $N'.P_1$ to N (append after $N.P_m$), same idea

Coalesce at Internal Nodes

- After also deleting “yak”:



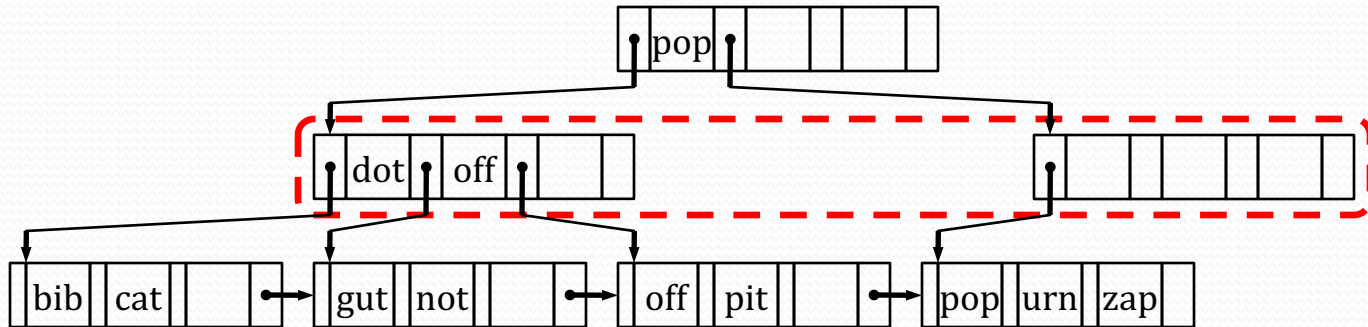
- Next, delete “rat”:



- Causes leaf node to become too empty...
- Need to coalesce leaf nodes; handle as usual

Coalesce at Internal Nodes (2)

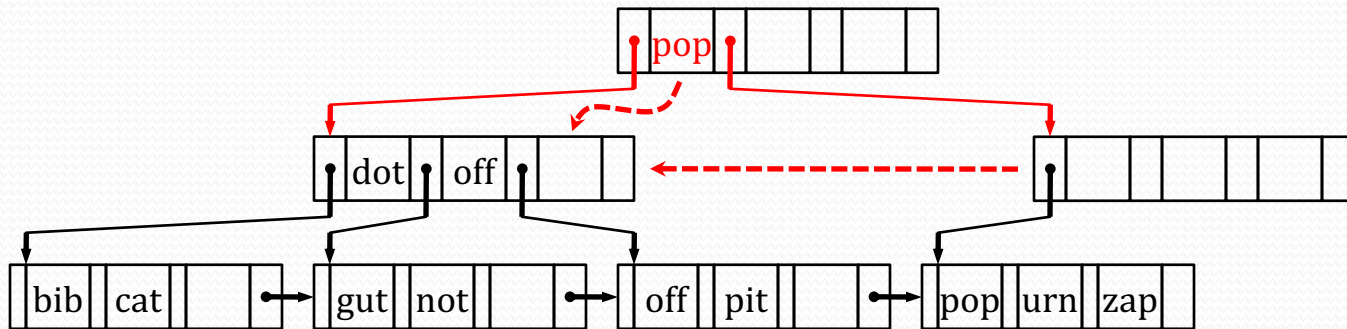
- After delete and coalesce:



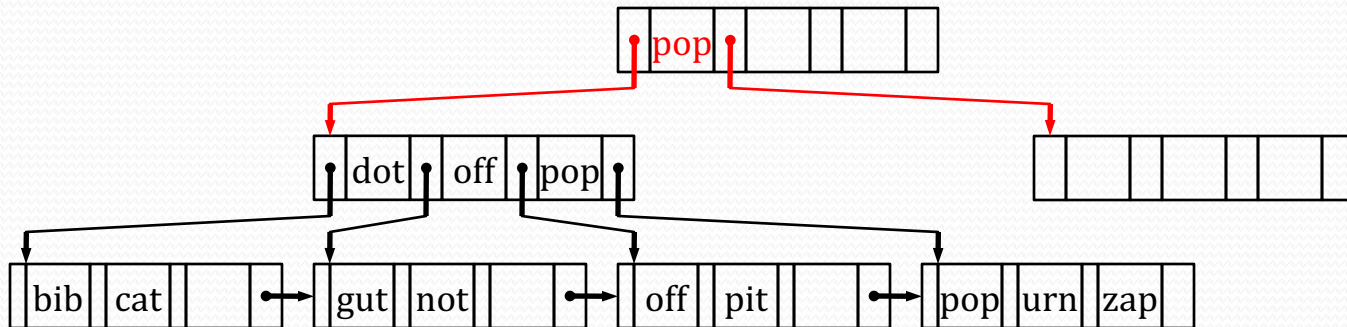
- Could redistribute from left sibling as before, but this time we can coalesce the two internal nodes together

Coalesce at Internal Nodes (3)

- As before, the two internal nodes being coalesced are separated by a key in the parent node

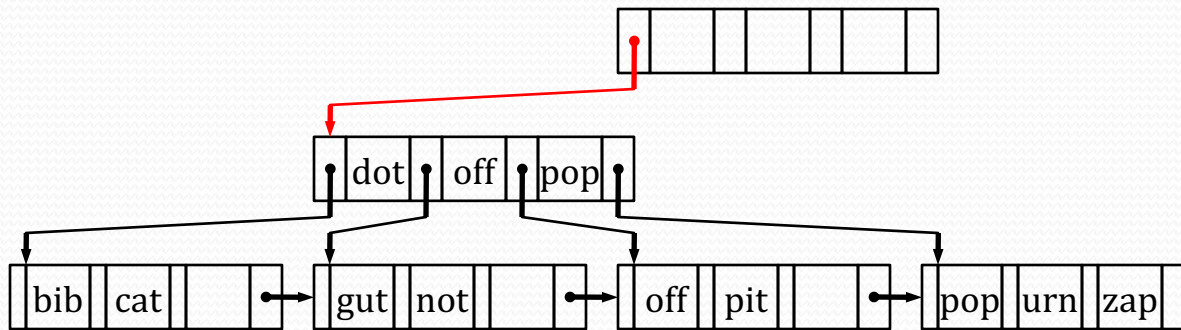


- When coalescing internal nodes' contents, use key from parent to separate the combined contents

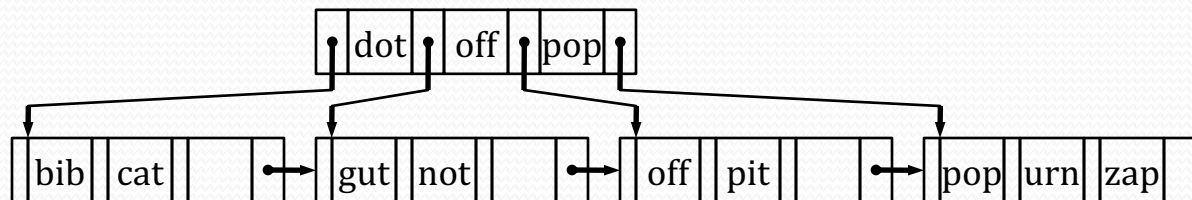


Coalesce at Internal Nodes (4)

- Also as before, remove pointer to deleted node, and also remove the key that separated them:



- Finally, if the root node only has one pointer, we don't need it anymore
 - Node pointed to by old root's lone pointer becomes the new root



Sketch of Delete Algorithm (1)

delete(value K , pointer P):

find leaf node L containing (K, P)

delete_entry(L, K, P)

delete_entry(node N , value K , pointer P):

find and remove (K, P) from N

if N is root and has only one
child-pointer:

make child the new root

delete N

else if N isn't full enough:

try to coalesce N with

either sibling of N

else, redistribute N 's contents
with either sibling of N

coalesce(N, N'):

K' = key that separates N and N'
in parent(N)

/* Details of coalesce depend on
* whether leaves or internal nodes
* are being combined; e.g. coalesce
* will use K' for internal nodes.
*/

combine contents of N and N'

/* Assuming N' was the node
* that ends up empty...

*/
delete_entry(parent(N'), K', N')
delete node N'

Sketch of Delete Algorithm (2)

delete(value K , pointer P):

find leaf node L containing (K, P)

delete_entry(L, K, P)

delete_entry(node N , value K , pointer P):

find and remove (K, P) from N

if N is root and has only one
child-pointer:

make child the new root

delete N

else if N isn't full enough:

try to coalesce N with

either sibling of N

else, redistribute N 's contents
with either sibling of N

redistribute(N, N'):

K' = key that separates N and N'
in parent(N)

/* Details of redistribute depend on
* whether leaves or internal nodes
* are involved; e.g. use K' for nonleaf.
* Also, may move ptr from N to N' ,
* or from N' to N ... *ugh...*
*/

move a pointer/key pair
from fuller node to emptier node

replace K' in parent(N) with
appropriate new key-value

B⁺-Tree Delete Algorithm

- Glossed over *many* details in sketch of algorithm
 - Mainly boring bookkeeping details, not hard to figure out, but quite tedious!
- Delete has a lot of similar but slightly different cases:
 - Can coalesce with either left or right sibling (if it exists!)
 - Can redistribute values with either left or right sibling – value may move in either direction
- Captured general principles in sketch and in examples

Deletes and Duplicate Values

- B⁺-tree deletion removes a specific record from index
 - `delete(value K , pointer P)`
 - We know the record we want to remove (P), and the search-key value it contains (K)
- Simplified examples by disallowing duplicate values
- Main change when duplicate keys are allowed:
 - When looking for a specific (K, P) pair in leaf nodes, we may have multiple index-entries to examine
 - If K appears *many* times, may have to traverse multiple leaf-nodes to find the specific value of P that was given

Deletes and Duplicate Values (2)

- A simple solution to this issue:
 - Add a *uniquifier* attribute to the search-key that always guarantees search-key values will be unique in the index
- Example: record-pointer would be a good uniquifier
 - Can easily compare and order record-pointers
 - Is readily available when inserting or deleting records
- When inserting or deleting on the index:
 - Include the uniquifier attribute when making placement decisions, when splitting/coalescing nodes, etc.
 - Specifically with deletion, use uniquifier to navigate to leaf directly; avoids any searching through leaf nodes

Deletes and Duplicate Values (3)

- When searching on the index:
 - Usually, uniquifier will not be specified for searches
- Can write search logic to ignore uniquifier component of index's search-key values
- Can pad input search-key V with “min-value” for the uniquifier
 - e.g. smallest possible value for a record-pointer
- This technique will reduce branching factor of non-leaf nodes
 - But, will improve performance when search-key values are repeated many times

B⁺-Trees and String Keys

- B⁺-tree search key could also include large strings:
 - Large key values will negatively impact branching factor of each node
 - Large keys can also greatly hamper tree restructuring
- Can use a *prefix compression* technique
 - Non-leaf nodes only store a prefix of the search string
 - Size of prefix must be large enough to distinguish reasonably well between values in each subtree
- Similarly, many databases allow user to specify how much of string index-components to use:

```
CREATE INDEX idx_customer_name  
ON customers (last_name(4), first_name(4));
```