BK-TREE: EFFICIENT RETRIEVAL OF SIMILAR STRINGS

Session 5

David Buchaca Prats
2023

DEALING WITH WORDS OUT OF THE VOCABULARY

- Based on two notions that we have already seen: The edit distance and a Language model, we can build an spellchecker.
- Let us consider that we want to correct misspelled words:
  - That is: words that are not in the vocabulary
- Base algorithm: Let x be a sentence and V the vocabulary.

```
For w in x:
 If w not in V:
   Find the closest words to w (candidate search)
   Evaluate each of the candidate words (candidate evaluation)
   Return the most probable candidate
```

FINDING THE CLOSEST ITEMS TO A QUERY

- This is an extremely relevant problem for many Data Science and ML problems.
- Sklearn implements the kdtree for dense vectors.
- What if we have strings…?

```
n_examples = 3_000_000
n_features = 25

X,y = sklearn.datasets.make_blobs(n_examples, n_features, centers=30, random_state=123)
x = X[0:1]
```

```
kdtree = sklearn.neighbors.KDTree(X)
kdtree
```

```
<sklearn.neighbors._kd_tree.KDTree at 0x7fa1778e9810>
```

```
%timeit kdtree.query(x, return_distance=False)
```

```
33.1 µs ± 220 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%timeit closest_match = np.argpartition(np.sum((X - x)**2,axis=1),1)[0]
```

```
301 ms ± 9.49 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
kdtree.query(x, return_distance=False)
```

```
array([[0]])
```

```
np.argpartition(np.sum((X - x)**2,axis=1),1)[0]
```

```
0
```

FINDING THE CLOSEST ITEMS TO A QUERY

- Efficient search of similar values in a dataset is a very challenging problem. In particular, computing distances between a word and a huge vocabulary can be computationally expensive.
- Let w be a string that is out of the vocabulary.
- Let us consider $W_k(w; X) = \{w \,|\, w \in V, d(w, w_j) < k\}$
- Finding $W_k(w; X)$ can be done in two different ways:

  I) compute $d(w, w_j)$ for all w_j then select the elements that are at distance at most k

  II) Use a data structure to avoid computing $d(w, w_j)$ for all w_j in X.

TREE INTUITION

- We can build a tree to do efficient search of similar words. This will allow us to prune a lot of the search space, with the objective of avoiding many distance computations on a big part of the vocabulary.
- Example: consider w = pleistation

ana	d(pleistation, ana)
playstation	d(pleistation, playstation)
house	d(pleistation, house)

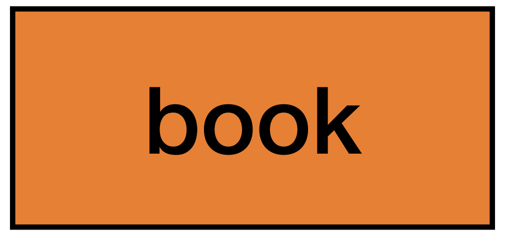
- If pleistation has 11 characters and we want all candidates to bet at most at distance k=3, is there any need to compute d(pleistation, ana) ?
  - Ana has 3 characters!

BK-TREE

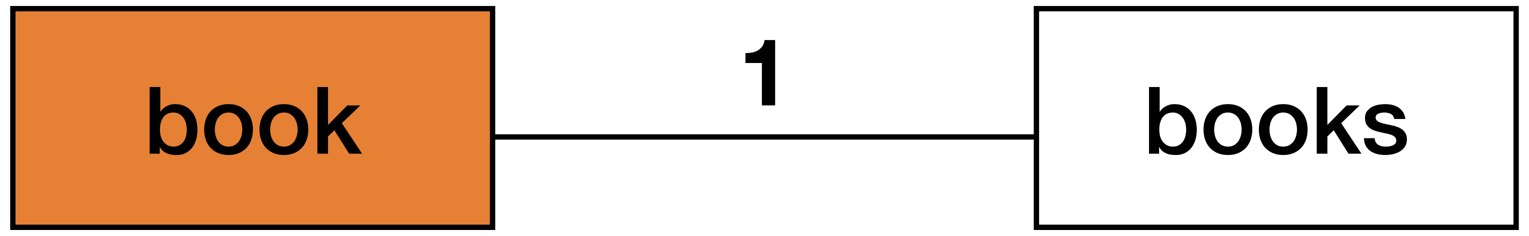
- To create a BK-tree we will follow this approach.
  - Select any word from the vocabulary and use it in as the root node.
  - Keep adding words until all vocabulary is the tree.
    - Each time we add a word the distance between the word and the root node is computed, let us assume this distance is d.
    - If no node from the root node is at distance d we add a new leave as a descendant of the rood node with edge value equal to d.
    - If there exist another node at distance d then… we repeat this process redefining the root node as the node that produced the collisiond(pleistation, ana)

BK TREE CONSTRUCTION EXAMPLE

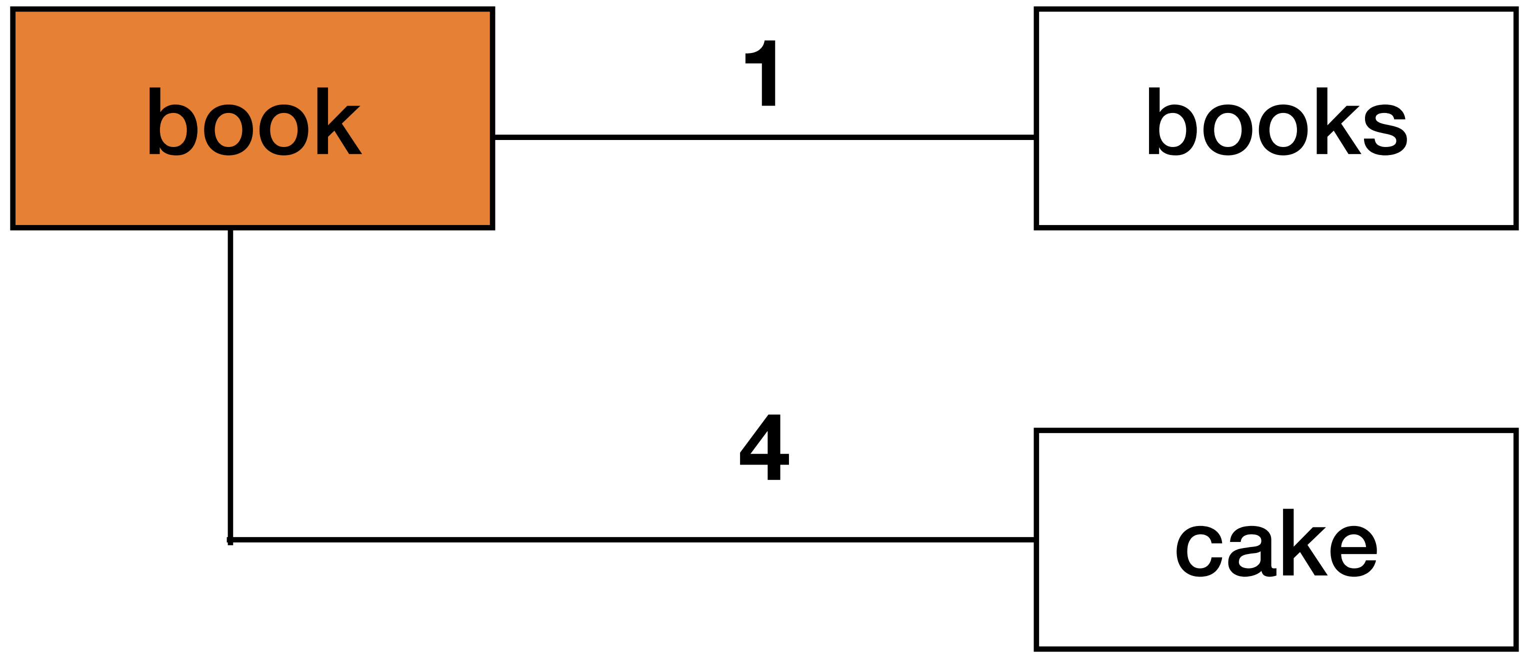
- Let us consider the data [book, books, cake], we start from **book (which becomes root node)**



- [book, books, cake]: **books** comes and we compute d(book, books)=1

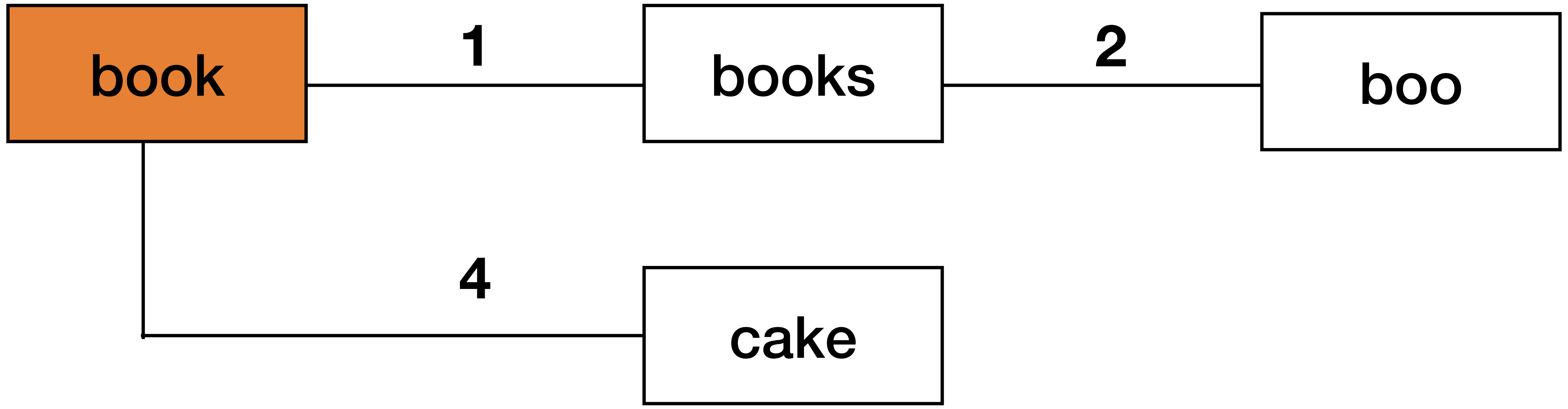


- [book, books, cake]: **cake** comes and we compute d(book, cake)=4



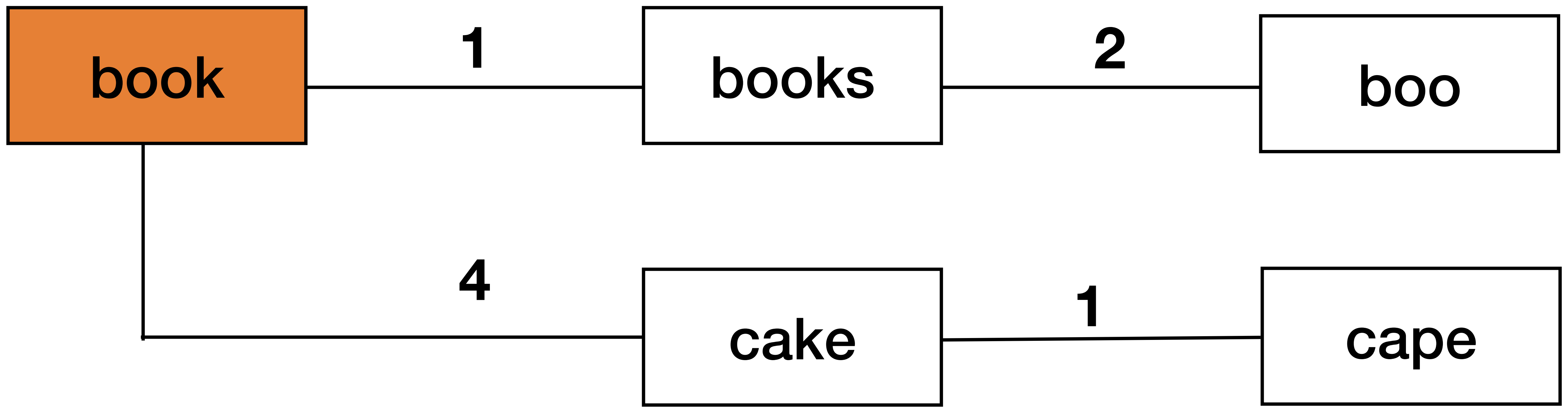
BK-TREE EXAMPLE: [BOOK, BOOKS, CAKE, BOO, CAPE, CART, BOON, COOK]

- [book, books, cake, boo]: **boo** comes and we compute d(book, boo)=1. Note that there is already **books** at distance 1.
- The BK tree has to respect that every node have all children with different distances, **since there is already a word at the same edit distance 1** we go to the branch of words at distance 1.
- **If there is a collision** (like we have now) **the new word must become a children of the collisioned word**. In this case, a children of book.
- The new weight from books to boo has to be distance(books, boo)=2.



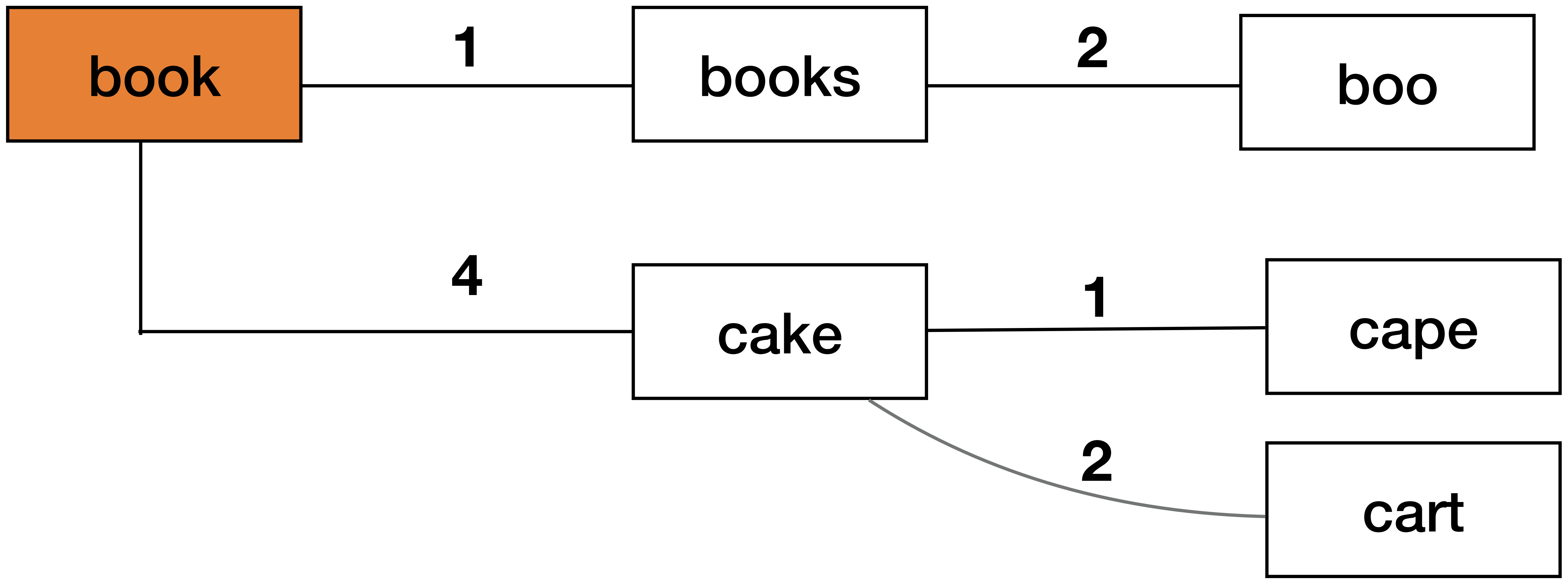
BK-TREE EXAMPLE: [BOOK, BOOKS, CAKE, BOO, CAPE, CART, BOON, COOK]

- Root=book: Compute d(book, cape)=4
  - Collision! There is already cake at distance 4 from book
  - Root node is now cake
  - Root=cake: Compute d(cake, cape)=1
  - There is no descendant from cake at distance 1 => we can add it



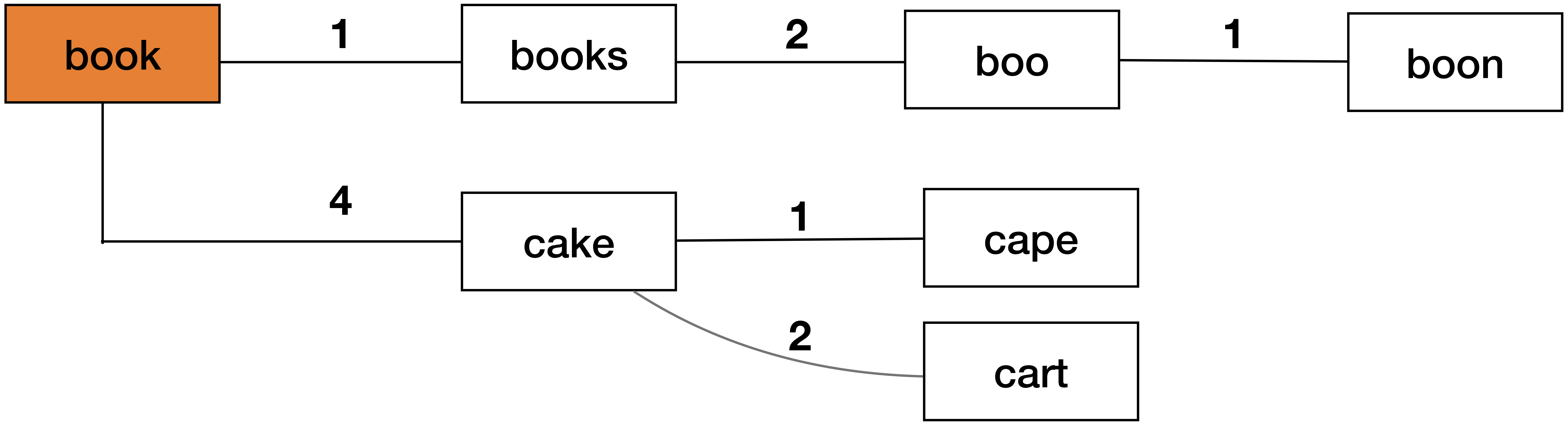
BK-TREE EXAMPLE: [BOOK, BOOKS, CAKE, BOO, CAPE, CART, BOON, COOK]

- Root=book: Compute d(book, cart)=4
  - Collision! There is already cake at distance 4 from book
  - Root node is now cake
  - Root=cake: Compute d(cake, cart)=2
  - There is no descendant from cake at distance 2 => we can add it



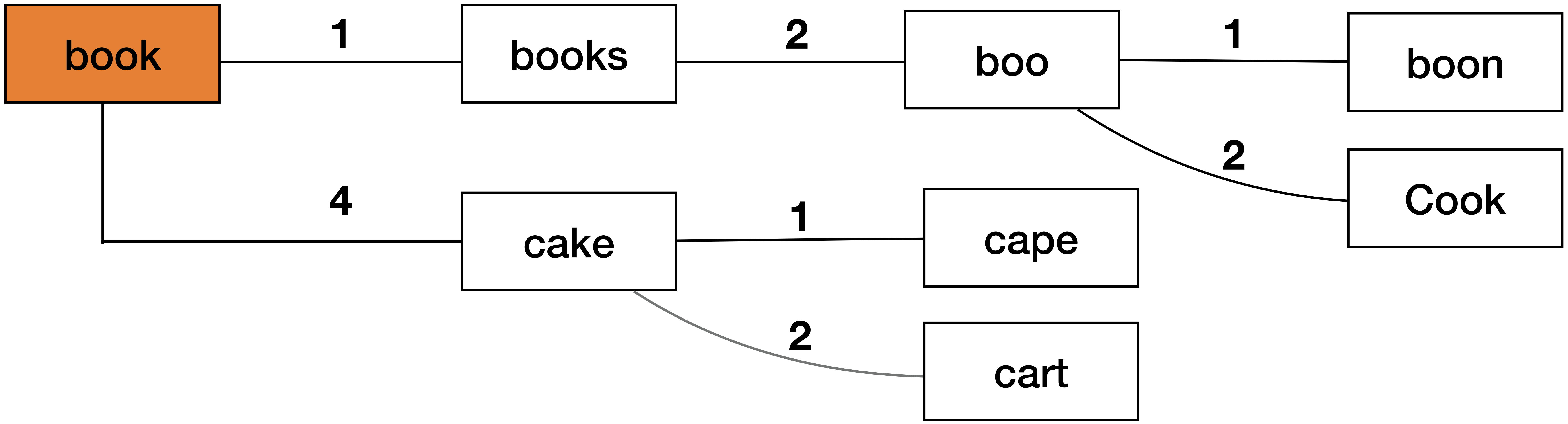
BK-TREE EXAMPLE: [BOOK, BOOKS, CAKE, BOO, CAPE, CART, BOON, COOK]

- Root=book: Compute d(book, boon)=1
  - Collision! There is already books at distance 1 from book
  - Root node is now books
  - Root=books: Compute d(books, boon)=2
  - Collision! There is already boo at distance 2 from books
  - Root=boo: Compute d(books, boon)=1
  - There is no descendant from boo at distance 1 => we can add it

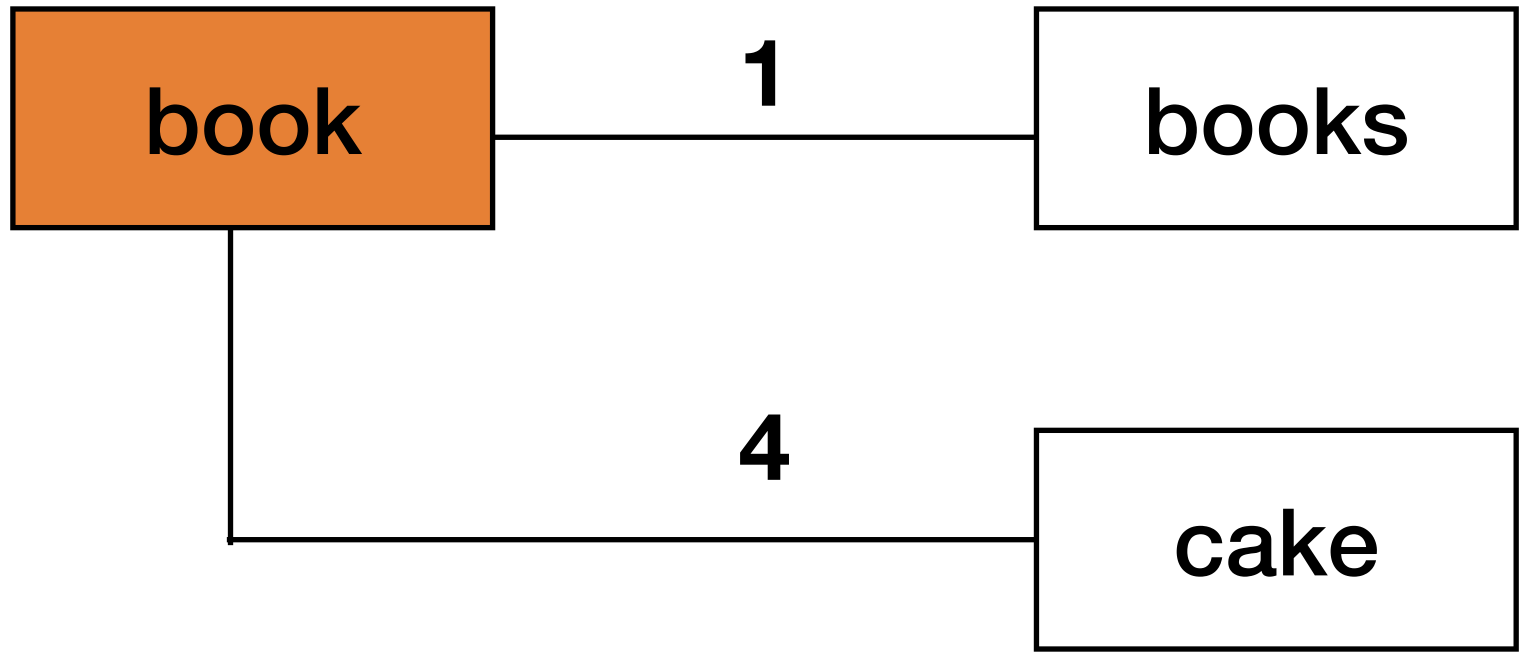


BK-TREE EXAMPLE: [BOOK, BOOKS, CAKE, BOO, CAPE, CART, BOON, COOK]

- Root=book: Compute d(book, cook)=1
  - Collision! There is already books at distance 1 from book
  - Root node is now books
  - Root=books: Compute d(books, cook)=2
  - Collision! There is already boo at distance 2 from books
  - Root=boo: Compute d(boo, cook)=2
  - There is no descendant from boo at distance 2 => we can add it

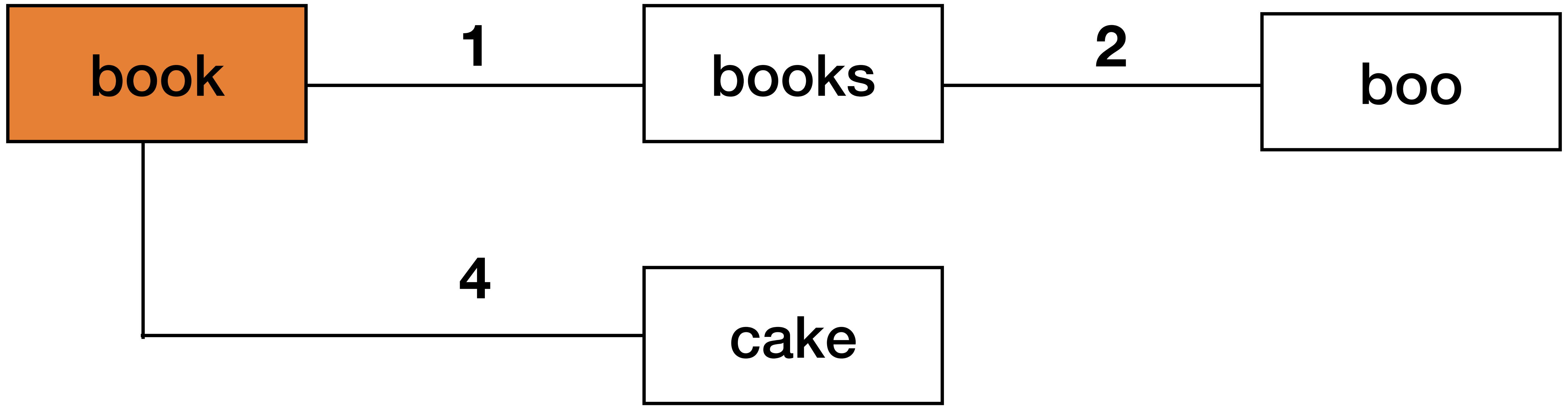


BK TREE: STORAGE IN MEMORY

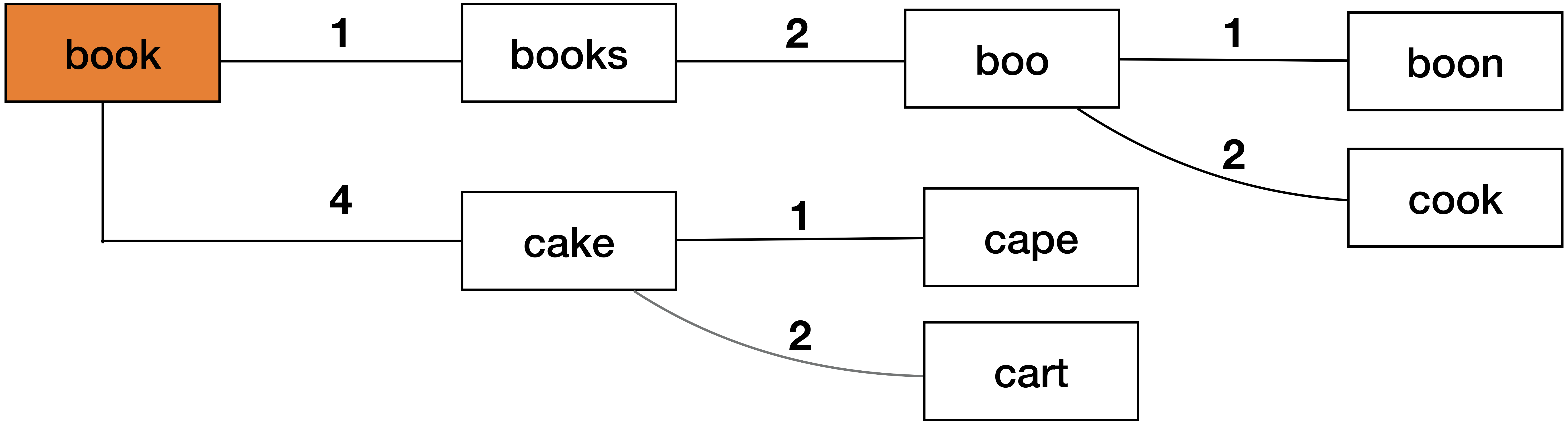


- Let us consider the following tree
- What would be a reasonable way to store the tree in memory?
  - A sensible way could be a tuple
  - The first element is the word assigned to the node
  - The second element is the subtree that spawns from that node
    - A subtree can be represented as a `Dict[Int, Tuple]`
      - keys are the distances to the root node
      - Values are tuples which represent subtree
  - In other words: `('book', {1: ('books', {}), 4: ('cake', {})})`

BK TREE: STORAGE IN MEMORY



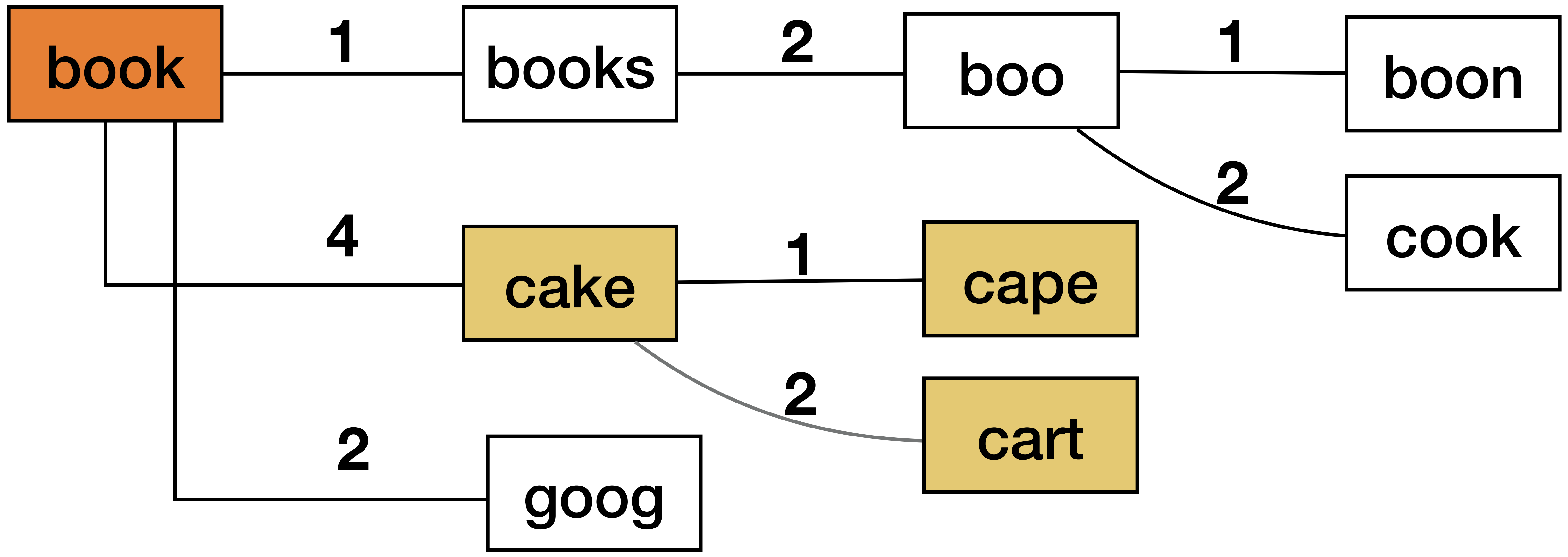
- ('book',
  {1: ('books', {2: ('boo', {})}),
   4: ('cake', {})})



- ('book',
  {1: ('books', {2: ('boo', {1: ('boon', {}), 2: ('cook', {})})}),
   4: ('cake', {1: ('cape', {}), 2: ('cart', {})})})

SEARCHING IN A BK-TREE

- Problem: Search all words that appear at distance less or equal than a tolerance T form a query word q .
- Bad solution: Compute all edit distances between q and w for w in the Vocabulary.
- Key idea: Visit all words w that are at distance [d(w,q)-T, d(w,q)+T].
- Example:



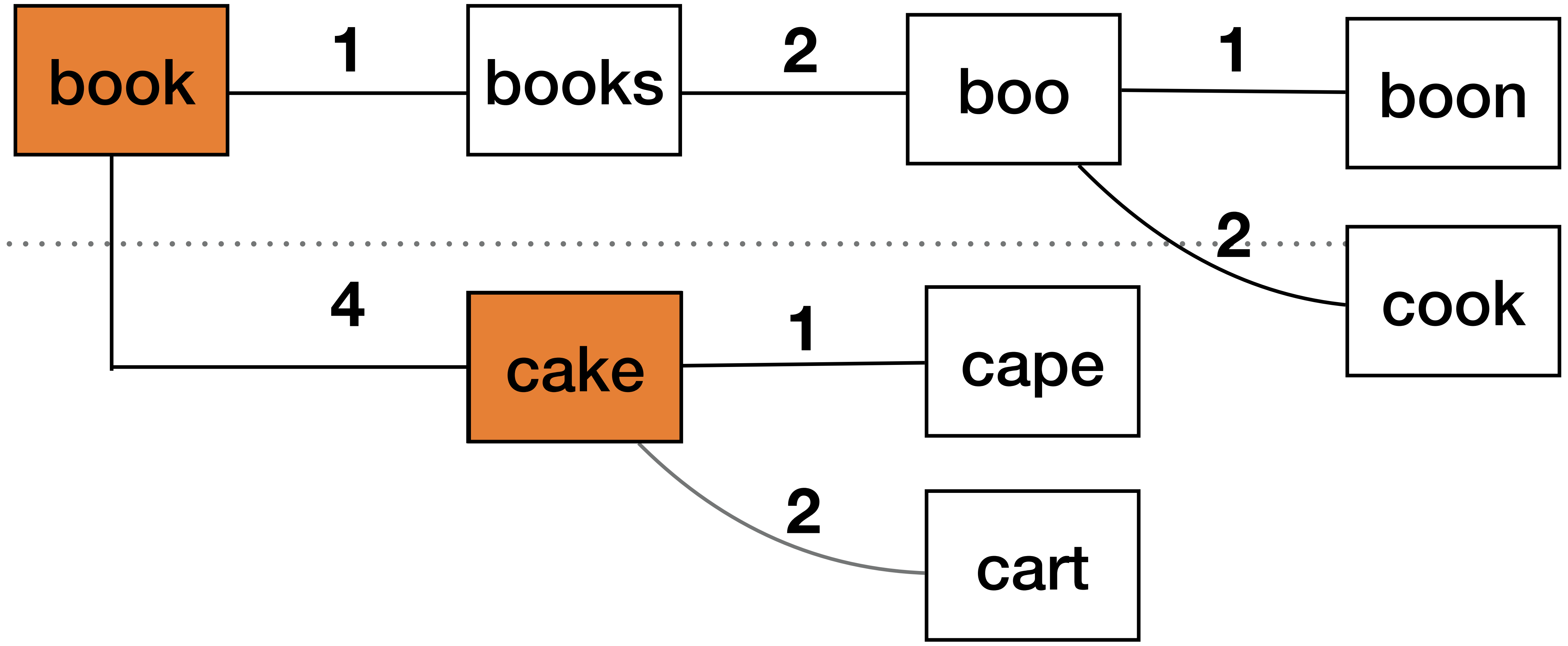
  - q = vook, T = 2
  - d(vook,book)=1
  - Consider w form key 4 from book (yellow)
    - By construction all words in yellow subtree are at a distance 4 from book
    - d(vook,w) <= d(vook, book) + d(book, w) = 5

This is 1

This is 4

No need to search for w in the yellow subtree: we want d(vook,w) <= 2

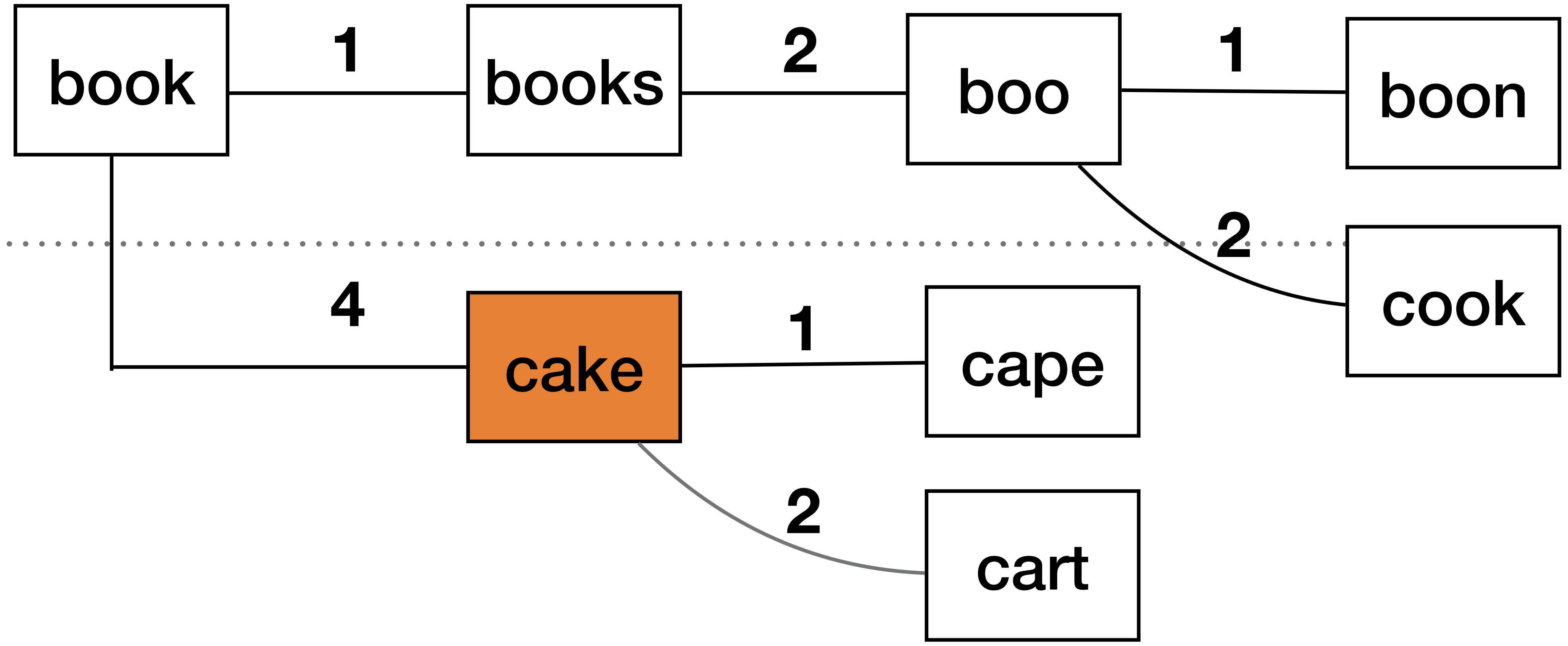
BK-TREE EXAMPLE: SEARCHING



- Let us consider
- q=caqe, T=1, candidates = []

- Select candidate **book** from search=[book]
  - d(book, caqe) = 4 => candidates is not updated
    - Crawl all children of book at distance I=[4-1,4+1]=[3,5]
    - Only node cake is connected to book and with distance in I=[3,5]
    - search = [book, cake]\book = [cake]

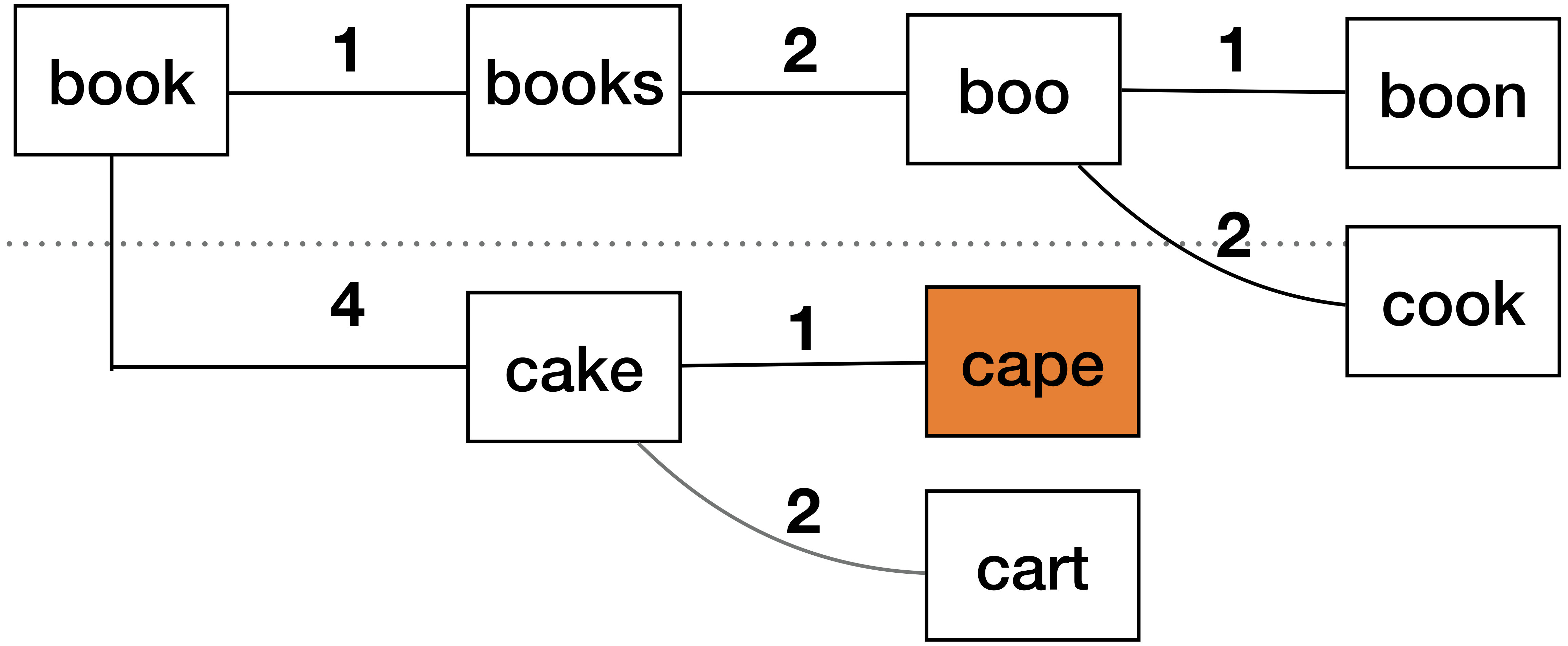
BK-TREE EXAMPLE: SEARCHING



- Let us consider
- q=caqe, T=1, candidates = []

- Select candidate **cake** from search=[cake]
  - d(cake, caqe) = 1 => candidates += [cake]
    - Crawl all children of cake at distance I=[1-1,1+1]=[0,2]
    - Only are 2 possible nodes, search=[cape, cart]

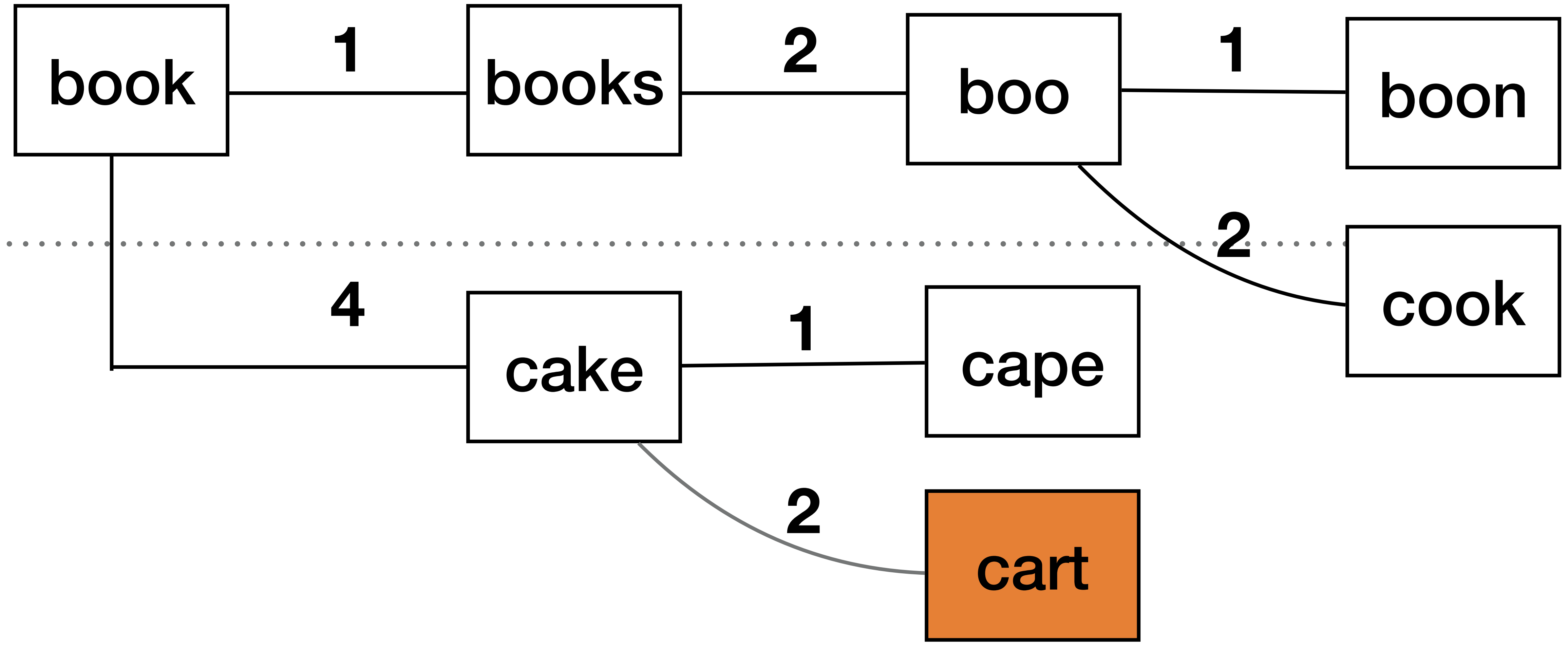
BK-TREE EXAMPLE: SEARCHING



- Let us consider
- q=caqe, T=1, candidates = [cake]

- Select candidate **cape** from search=[cape, cart]
  - d(cape, caqe) = 1 => candidates += [cape]
    - Crawl all children of cape at distance I=[1-1,1+1]=[0,2]
    - cape has no children
    - search = [cape, cart]\cape = [cart]

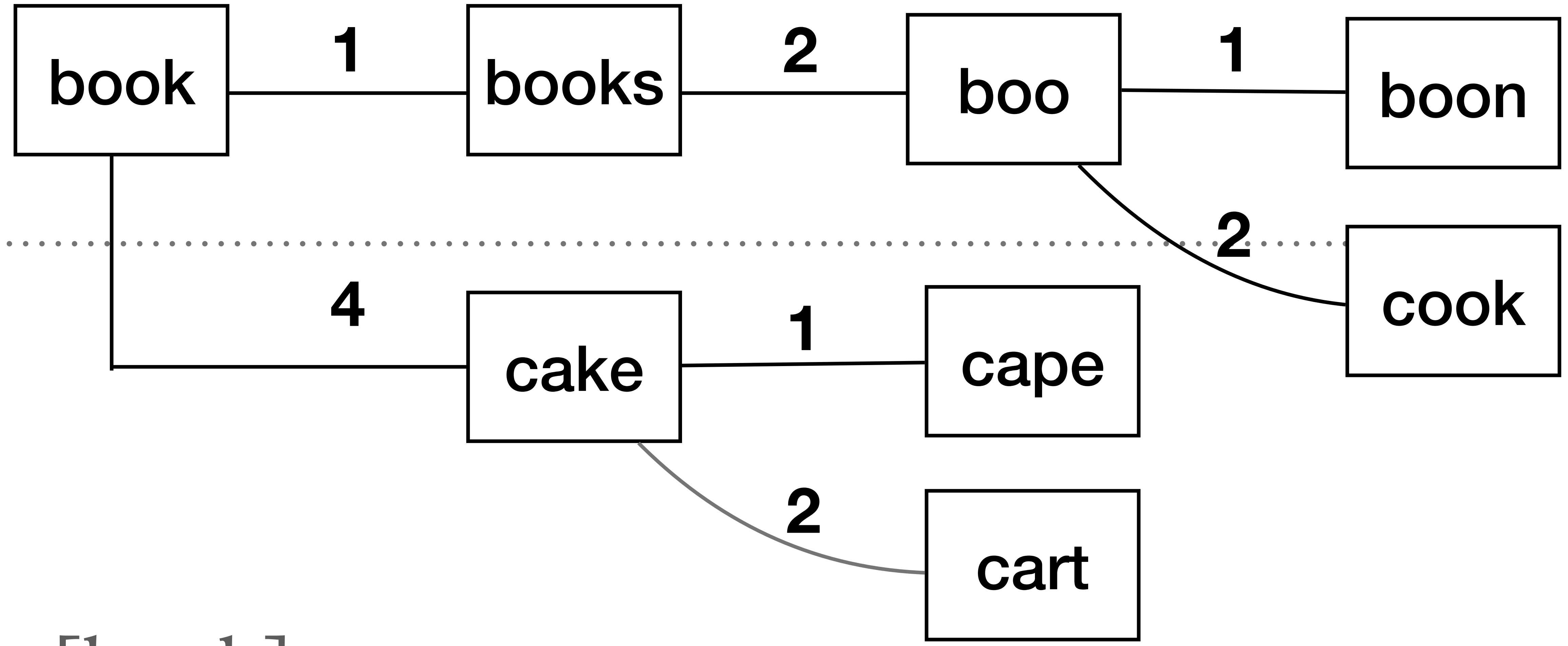
BK-TREE EXAMPLE: SEARCHING



- Let us consider
- q=caqe, T=1, candidates = [cake, cape]

- Select candidate **cart** from search=[cart]
  - d(cart,caqe) = 2 => candidates is not updated
    - Crawl all children of cape at distance I=[1-1,1+1]=[0,2]
    - Caqe has no children
    - search = [cart]\cart = [] =>Search space is empty, stop search
- The resulting set of possible candidates at distance 1 are: [cape,cake]

BK-TREE EXAMPLE: SEARCHING



- To sum up:
  - We started from:
    - q=caqe, T=1, candidates = [], search=[book]
  - After searching in the BK-Tree we know
    - The set of possible candidates at distance 1 are: [cape, cake].
- Observation:we ended up computing 4 edit distances yet we have 8 nodes.

BK- TREE SPEEDUP

- In the case that the search space is drastically pruned, the speedup can be massive:

```
word = "anthropomorphologicaly"
max_dist = 2
sort_candidates=False

%timeit candidates_ext = get_candidates_exhaustive(word,max_dist,words)
```

```
404 ms ± 5.26 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
candidates_ext = get_candidates_exhaustive(word,max_dist,words)
candidates_ext
```

```
[(1, 'anthropomorphological'), (1, 'anthropomorphologically')]
```

```
word = "anthropomorphologicaly"

%timeit candidates_ext = t.query(word, 2)
```

```
214 µs ± 1.77 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
candidates_ext = t.query(word, 2)
candidates_ext
```

```
[(1, 'anthropomorphological'), (1, 'anthropomorphologically')]
```

BK- TREE SPEEDUP

- If the pruned search space still contains a huge amount of words the speedup might note be that huge:

```
word = "astrologi"
max_dist = 2
sort_candidates=False

%timeit candidates_ext = get_candidates_exhaustive(word, max_dist, words)
```

```
319 ms ± 1.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
word = "astrologi"

%timeit t.query(word, 2)
```

```
96.2 ms ± 737 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```