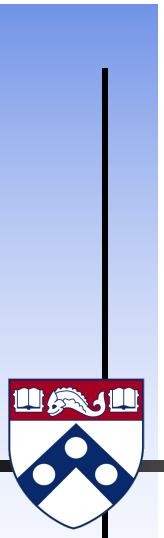


CIS 455/555: Internet and Web Systems

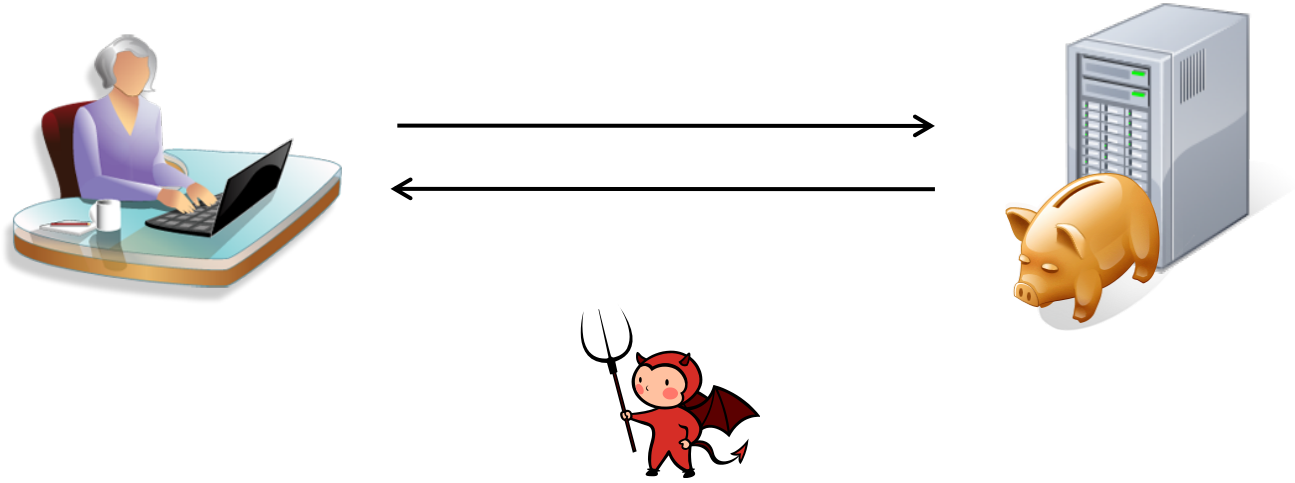
Indexing

September 27, 2021

Plan for today

- Naming ✓
  - Flat naming ✓
  - Attribute-based naming; LDAP ✓
  - The Domain Name System (DNS) ✓
  - Attacks on DNS ← NEXT
  - DNSSEC
- Inverted indices
- B+ trees

Attacking DNS

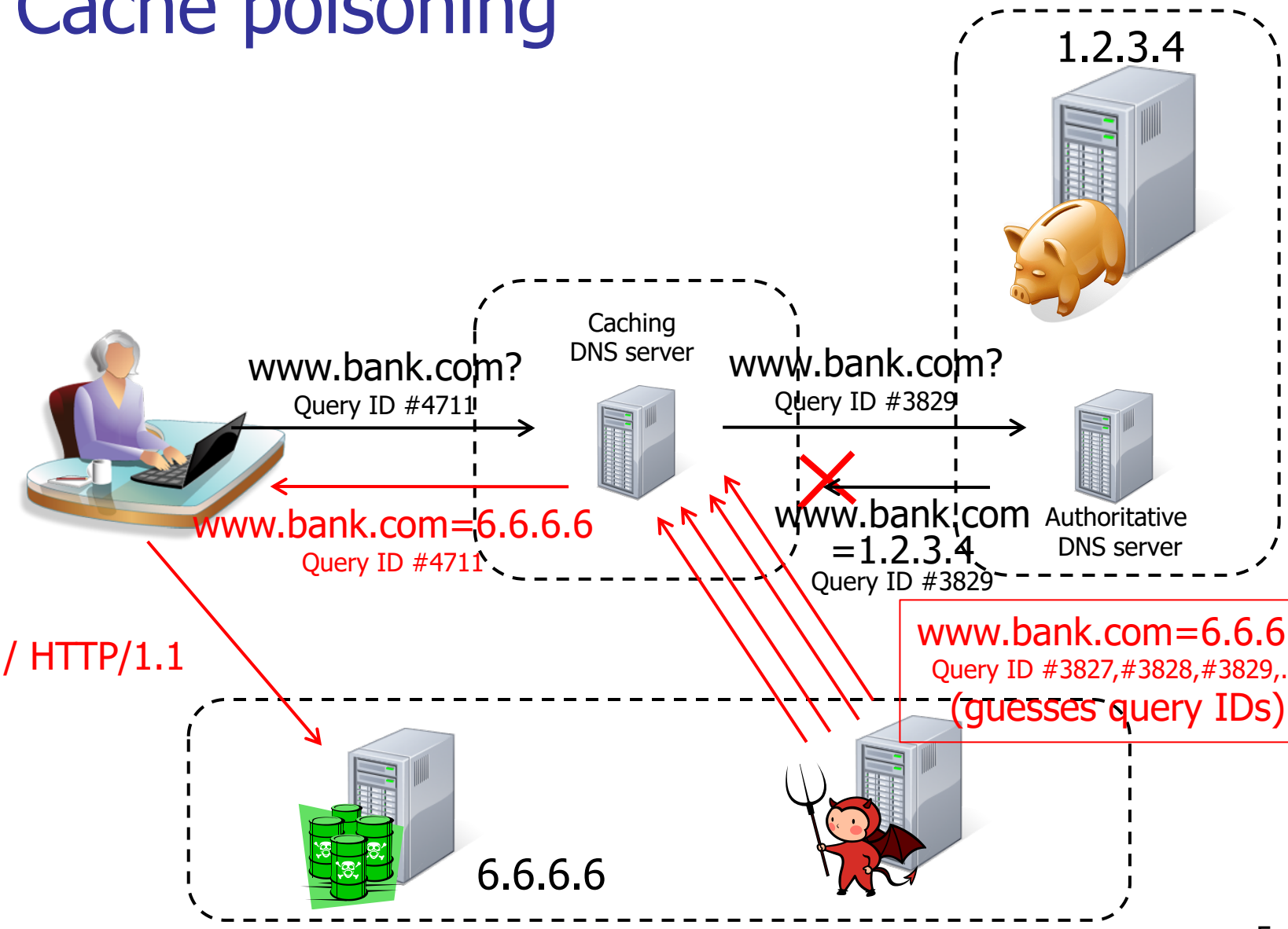


- Suppose an adversary wants to learn Alice's banking password
  - The adversary wants to redirect Alice's HTTP requests to his own server
- How can the adversary accomplish this?

Simple attacks

- Break into the DNS server
  - Can modify the zone file directly
- Phishing / Spearphishing
- Spoof the responses

Cache poisoning

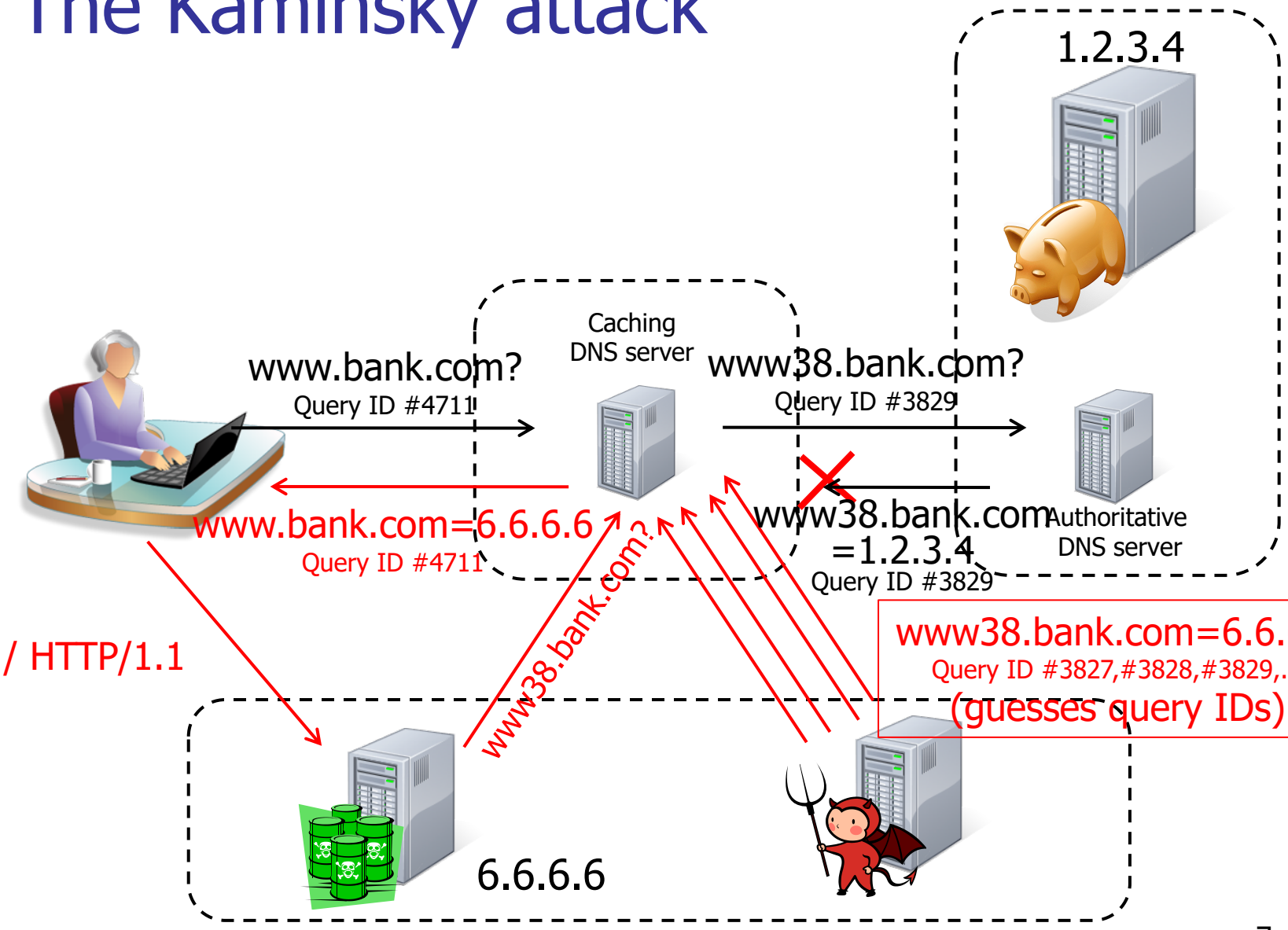


When does this work?

- Name can't already be in the cache
  - Otherwise no external queries will occur
- Adversary has to guess the query ID
  - If the name server doesn't properly choose the query IDs (e.g., counts up by one), this is easy
- Adversary has to be faster than the real name server

- Countermeasures?
  - Randomize query IDs?
  - Set long TTLs?

The Kaminsky attack



When does this work?

- Name can't already be in the cache
  - Otherwise no external queries will occur
- Adversary has to guess the query ID
  - If the name server doesn't properly choose the query IDs (e.g., counts up by one), this is easy
- Adversary has to be faster than the real name server

- Countermeasures?
  - Randomize query IDs?
  - Set long TTLs?
  - Randomize source ports?

Plan for today

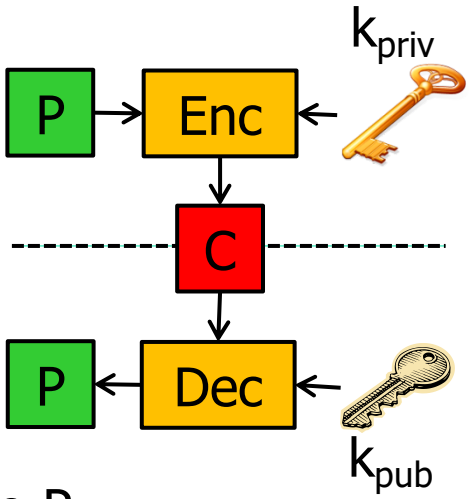
- Naming ✓
  - Flat naming ✓
  - Attribute-based naming; LDAP ✓
  - The Domain Name System (DNS) ✓
  - Attacks on DNS ✓
  - DNSSEC NEXT
- Inverted indices
- B+ trees

What is DNSSEC?

- DNS Security Extensions
  - A change to the DNS specification to make it more secure while maintaining backwards compatibility
- What does it do?
  - Protect DNS against spoofing and corruption
  - Cryptographically signs DNS responses

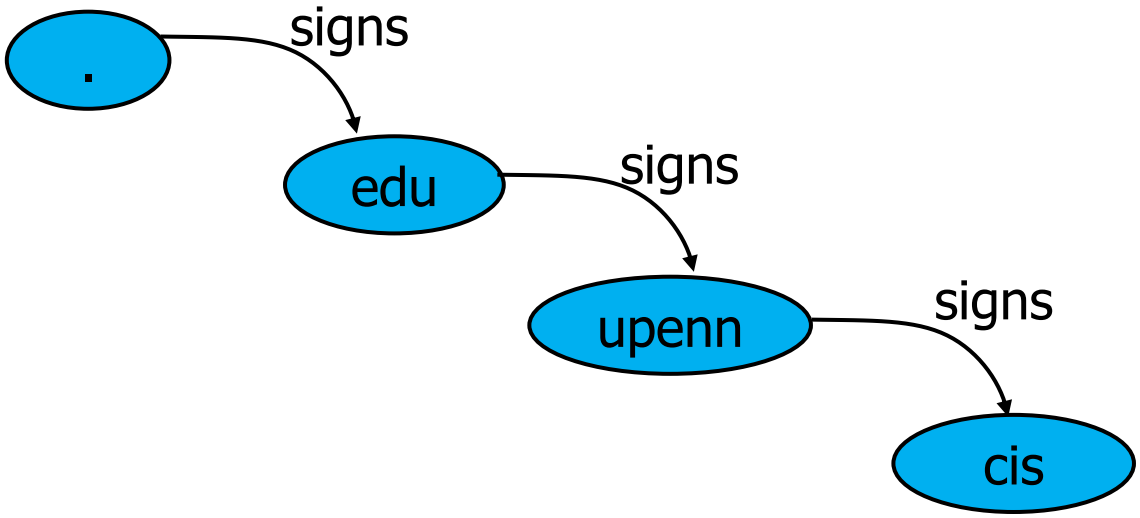
Digital signature basics

- We can generate a keypair (k_{pub},k_{priv}) and implement two operations:
  - Enc(k_{priv}, P) → C: encrypt plain text P with the private key; result is a ciphertext C
  - Dec(k_{pub}, C) → P: decrypt ciphertext C with the public key; result is plaintext P
  - We can make it very difficult to
    1) compute k_{priv} from k_{pub}, or to
    2) compute, without k_{priv}, a C that decrypts to P



- Idea: Alice publishes k_{pub}, but k_{priv} is secret
- When Alice wants to sign some statement S, she publishes (S, Enc(k_{priv},S))
  - Anyone can verify by decrypting Enc(k_{priv},S) with k_{pub}

How DNSSEC works



- Each domain has a keypair (public/private key)
  - Controlled by the entity to whom the domain is delegated
- Parent domains sign keys of child domains
  - Key of the root domain is well-known
  - NOTE: Parent does NOT sign the child domain itself!
  - NOTE: The data is signed, not just the messages!

DNSSEC deployment

- DNSSEC is now finally deployed
  - Signed root zone available since July 15, 2010
  - But DNSSEC has been under discussion since at least 1993!
- Why has it taken so long?
  - Attacks are relatively rare
  - DNSSEC is not free (e.g., key management)
  - Bootstrapping problem: Value is limited until it is widely deployed → Initial lack of incentives to deploy
    - A common problem with deploying new technologies
    - Other classic examples: S-BGP, IPv6

Recap: Attacks on DNS; DNSSEC

- The original DNS is vulnerable to a variety of attacks
  - Spoofing
  - Cache poisoning
  - Kaminsky attack
- DNSSEC solves many of these issues
  - DNS records are signed cryptographically
- It has taken a long time to deploy DNSSEC
  - Under discussion since at least 1993
  - Reasons are more economic than technical (incentives!)
  - Bootstrapping problem

Finding the 'right' architecture

- Should we use names or addresses?
- If names, what kind of names?
  - Flat names (Gnutella, ARP), hierarchical names (LDAP, DNS)
- How are names assigned?
  - Choose-your-own (Gnutella), explicit registration (DNS)
  - Delegation in DNS
- How do we resolve names?
  - Flooding (Gnutella), centralized directory (Napster), hierarchical directory (DNS, LDAP), decentralized directory
  - Caching in DNS, dynamic mapping in Akamai
- What about security?
  - Flooding/poisoning/Kaminsky → Anycast, DNSSEC

Plan for today

- Naming ✓
  - Flat naming ✓
  - Attribute-based naming; LDAP ✓
  - The Domain Name System (DNS) ✓
  - Attacks on DNS ✓
  - DNSSEC ✓
- Inverted indices NEXT
- B+ trees

Finding data by content

- We've seen two approaches to search:
  - Flood the network with requests (example: Gnutella), and do all the work at the data stores
  - Have a directory based on names (example: LDAP)
    - Which of these is the 'best'?
- An alternative, two-step process:
  - Build a content index over what's out there
    - An index is a key→value map
  - Typically limited in what kinds of queries can be supported
  - Most common instance: an index of document keywords
  - Example: Incidence matrix.
    - Is this a good idea?

A common model for search

- Index the words in every document
- "Forward index": document (ID) → list of words
- "Inverted index": word → document (ID)

Inverted indices

- A conceptually very simple map-multiset data structure: <keyword, {list of occurrences}>
- In its simplest form, each occurrence includes a document pointer (e.g., URI), perhaps a count and/or position
  - What might a count be useful for? A position?
- Requires two components, an indexer and a retrieval system
- We'll consider the cost of building the index, plus searching the index using a single keyword
  - Storage efficiency is also a concern

How do we lay out an inverted index?

- Which operations do we need to support?
  - insert
  - delete
  - find
  - next
- Which data structures could we use?
  - Hash table
  - Unordered list (e.g., a log)
  - Ordered list
  - Tree
  - …
- Which properties are we looking for?

Unordered, ordered, and linked lists

- Assume that we have list of entries such as:
  <keyword, #items, {occurrences}>
- What does ordering buy us?
- How do we insert items?

Tree-based indices

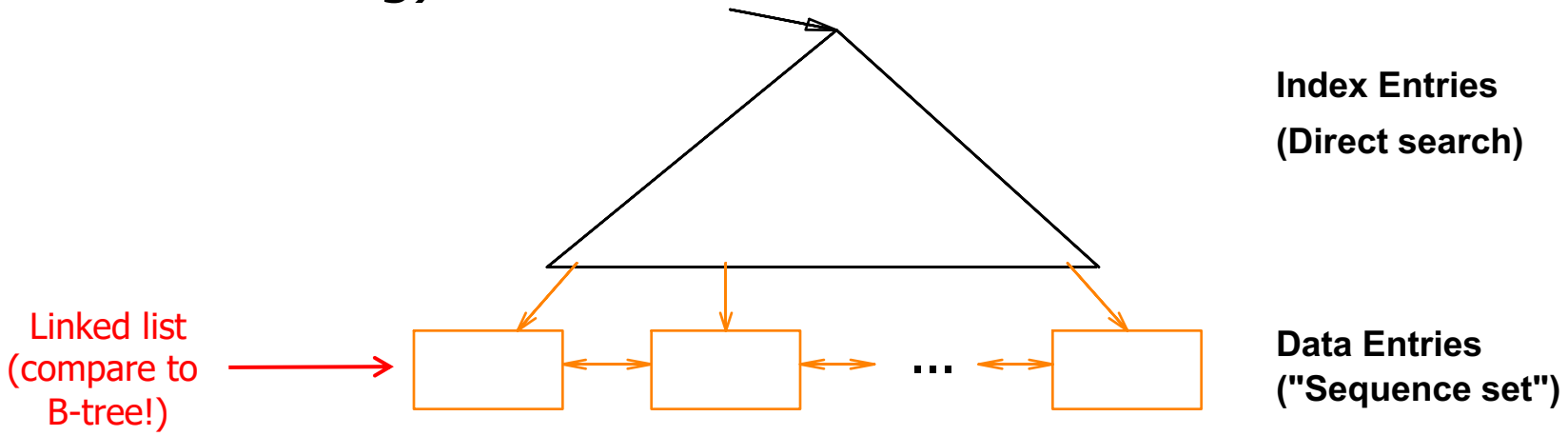
- Trees have several benefits over lists:
  - Potentially logarithmic search time, as with a well-designed sorted list
    - if it is balanced!
  - Ability to handle variable-length lists
- We've already seen how trees might make a natural way of distributing data, as well
- How does a binary search tree fare?
  - Cost of building?
  - Cost of finding an item in it?

Plan for today

- Naming ✓
  - Flat naming ✓
  - Attribute-based naming; LDAP ✓
  - The Domain Name System (DNS) ✓
  - Attacks on DNS ✓
  - DNSSEC ✓
- Inverted indices ✓
- B+ trees NEXT

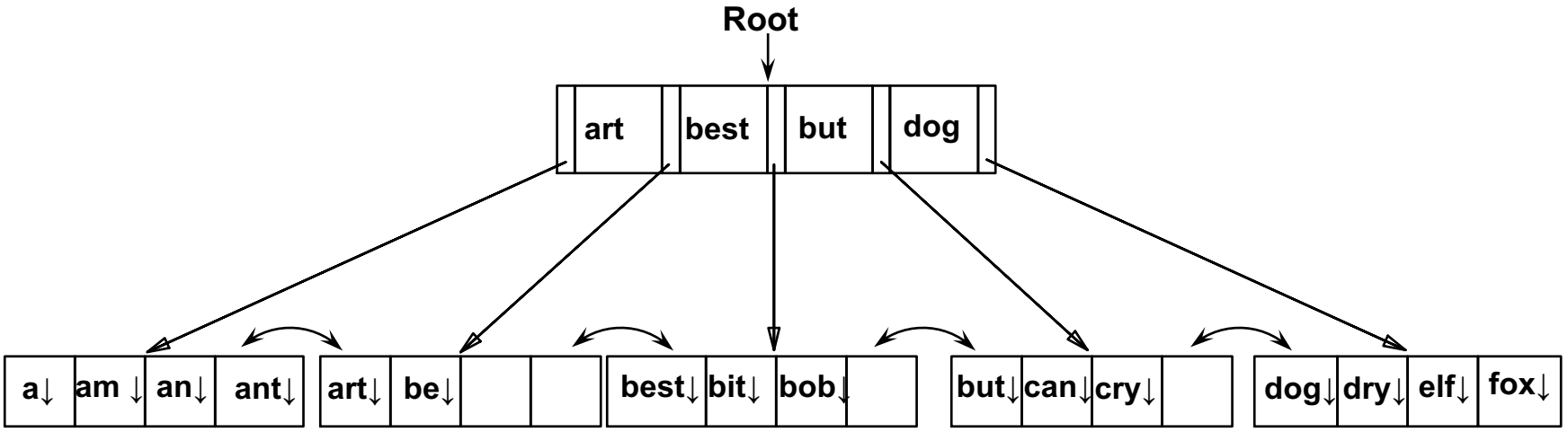
The B+ tree

- A flexible, height-balanced, high-fanout tree
- Insert/delete at $\log_F N$ cost (F = fanout, N = # leaf pages)
  - Need to keep tree height-balanced
- Minimum 50% occupancy (except for root)
  - Each node contains **d** <= *m* <= 2**d** entries
  - Inner nodes contain up to 2**d**+1 pointers
  - **d** is called the order of the tree
- Can search efficiently based on equality (or also rang)



Example B+ Tree

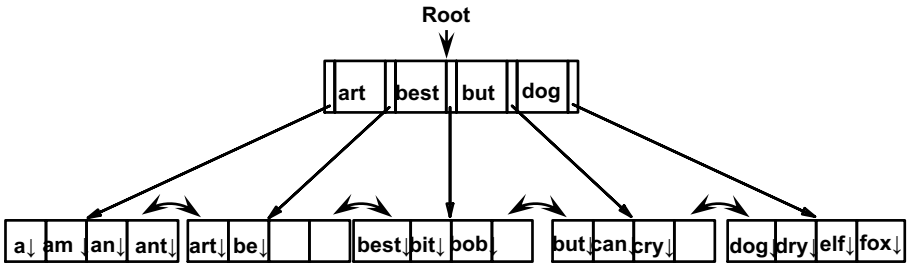
- Data (inverted list pointers) is at the leaves; intermediate nodes have copies of search keys
  - Why is that a good thing?
- Search begins at root, and key comparisons direct it to a leaf
- Search for be↓, bobcat↓ ...



➢ *Based on the search for bobcat*, we know it is not in the tree!*

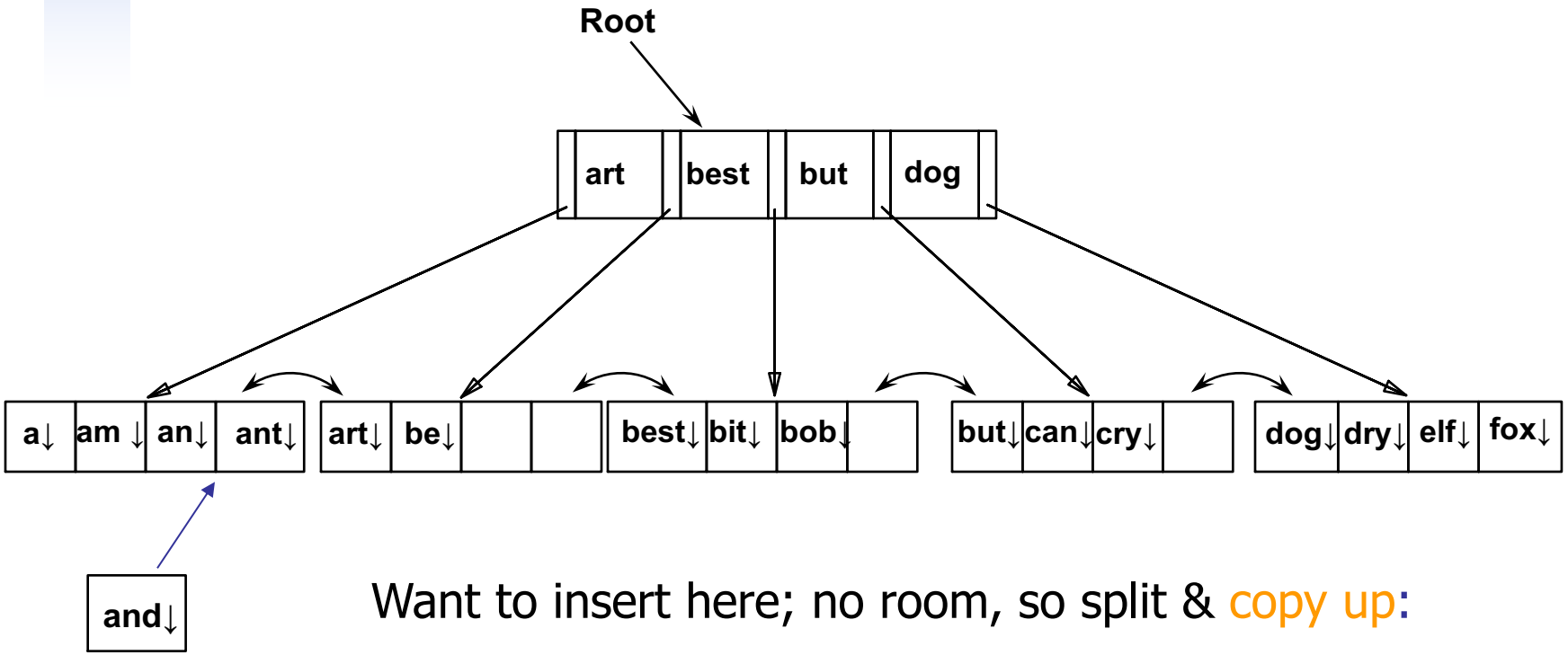
Inserting data into a B+ Tree

- Find correct leaf L
- Put data entry onto L
  - If L has enough space we are, done!

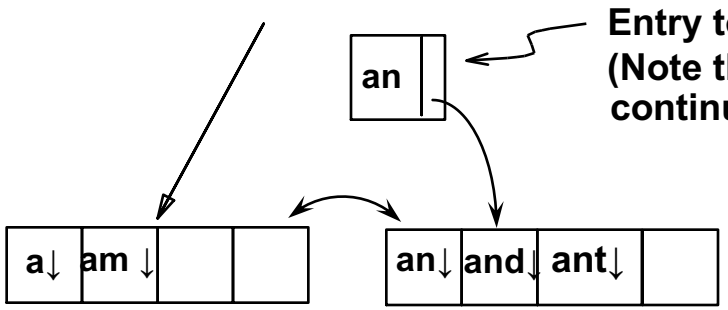


  - Else, must split leaf node L (into L and a new node L2)
    - Redistribute entries evenly, copy up middle key
    - Insert index entry pointing to L2 into parent of L
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height
  - Tree growth: gets wider or one level taller at the top

Inserting "and↓" Example: Copy up



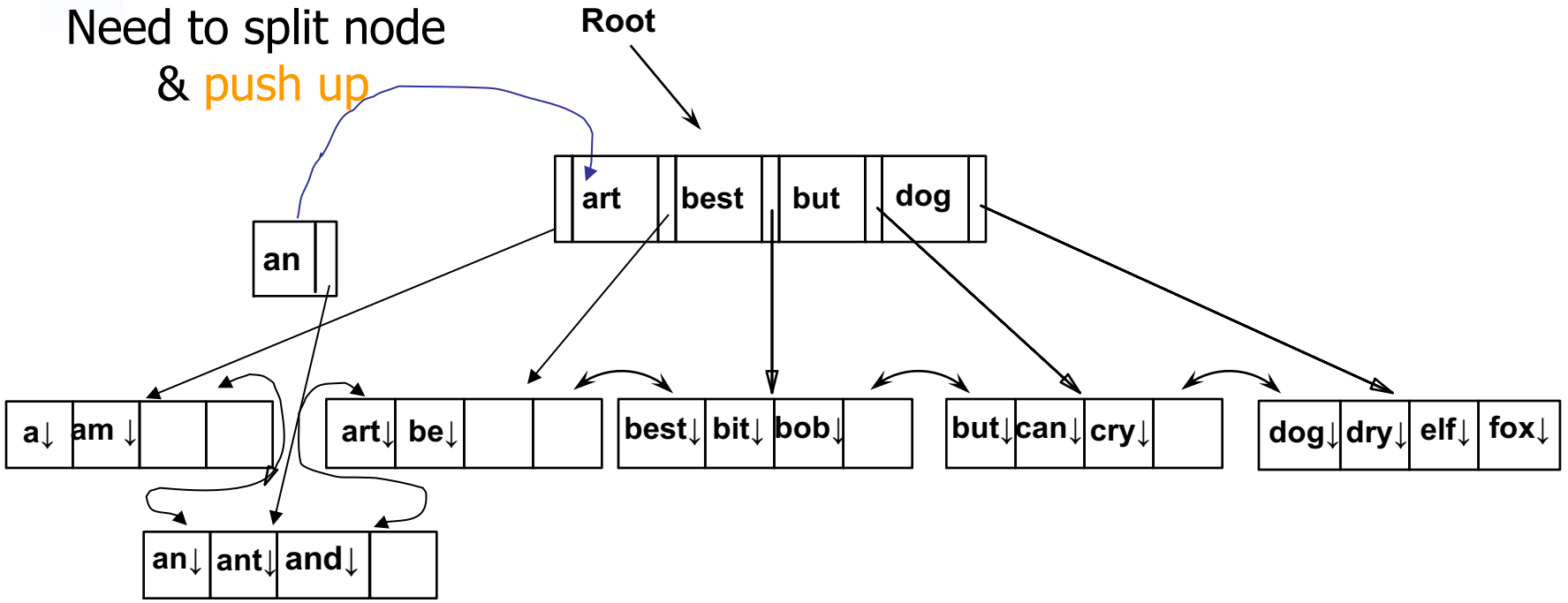
Want to insert here; no room, so split & copy up:

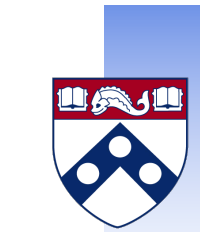


Entry to be inserted in parent node. (Note that key "an" is copied up and continues to appear in the leaf.)

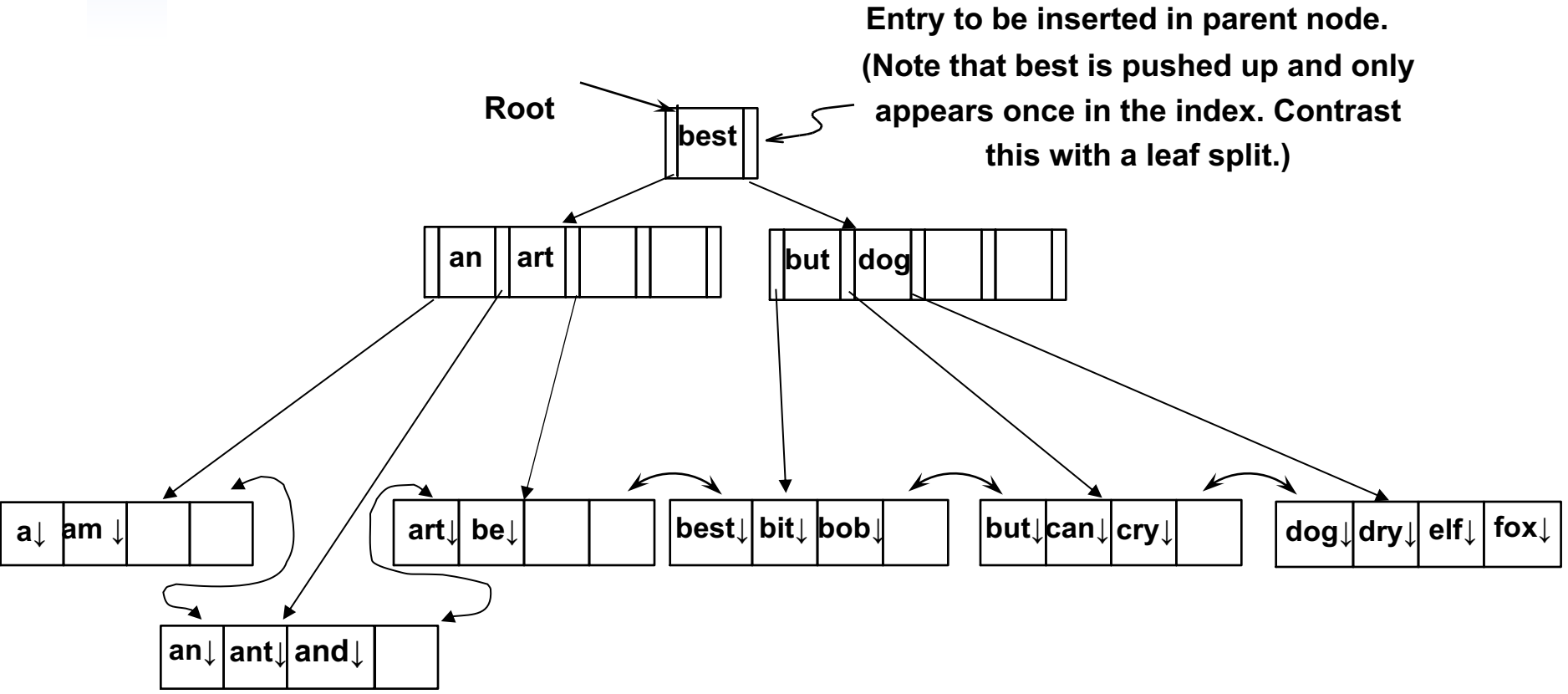
But where? Parent node is already "full"!

Inserting "and↓" Example: Push up 1/2





Inserting "and↓" Example: Push up 2/2



Summary: Copying vs. splitting

- Every keyword (search key) appears in at most one intermediate node
  - Hence, in splitting an intermediate node, we push up
- Every inverted list entry must appear in a leaf
  - We may also need it in an intermediate node to define a partition point in the tree
  - We must copy up the key of this entry
- Note that B+ trees easily accommodate multiple occurrences of a keyword