i206: Lecture 17:
Data Structures for Disk Access; Advanced Trees

Tapan Parikh

Spring 2013

Some slides courtesy Marti Hearst, John Chuang and others

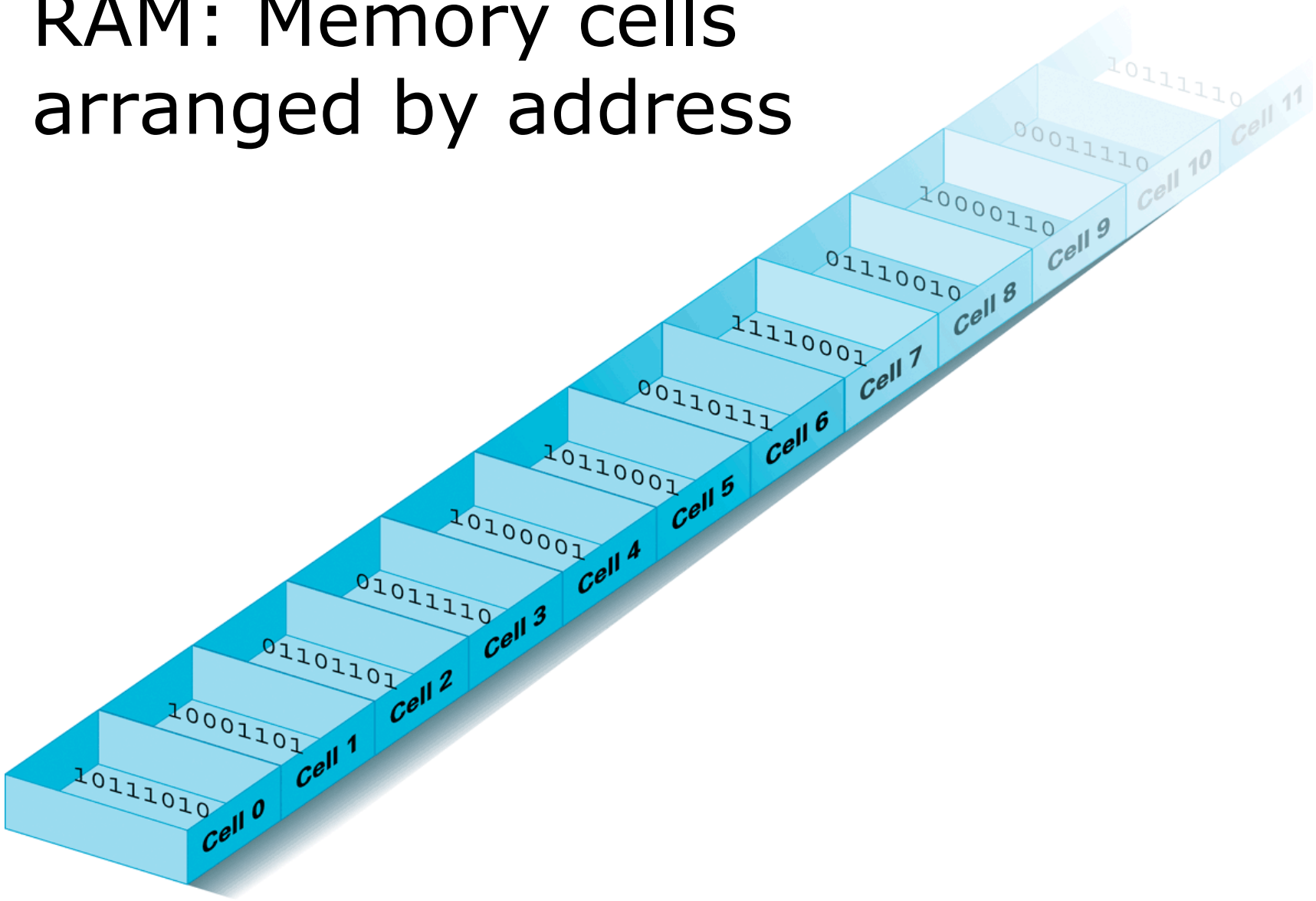
Data Structures & Memory

- So far we've seen data structures stored in main memory
- What happens when you have a very large set of data?
  - Too slow to load it into memory
  - Might not fit into memory

Disk vs. RAM

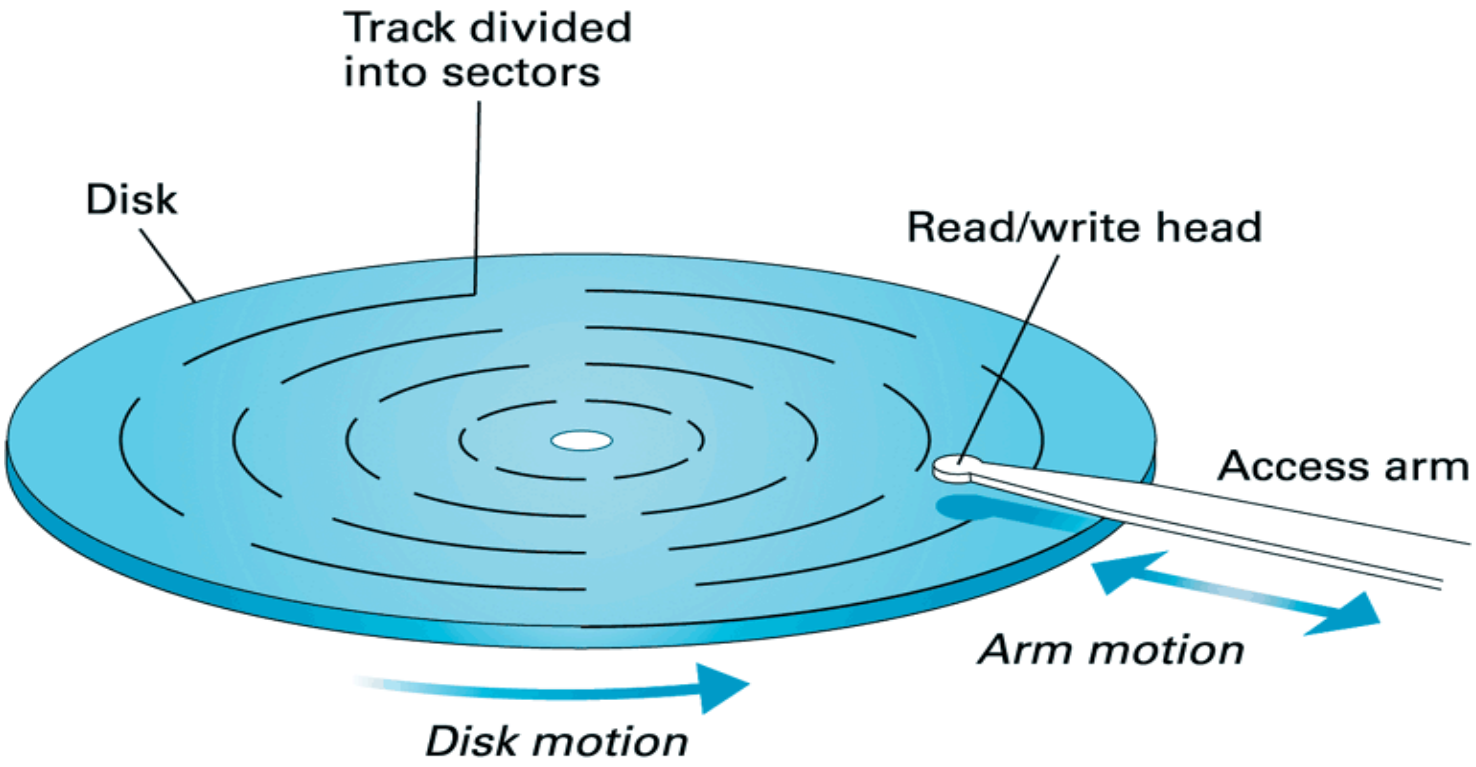
- Disk has much larger capacity than RAM
- Disk is much slower to access than RAM

RAM: Memory cells arranged by address



Images copyright 2003 Pearson Education

Disk: Memory cells arranged by address



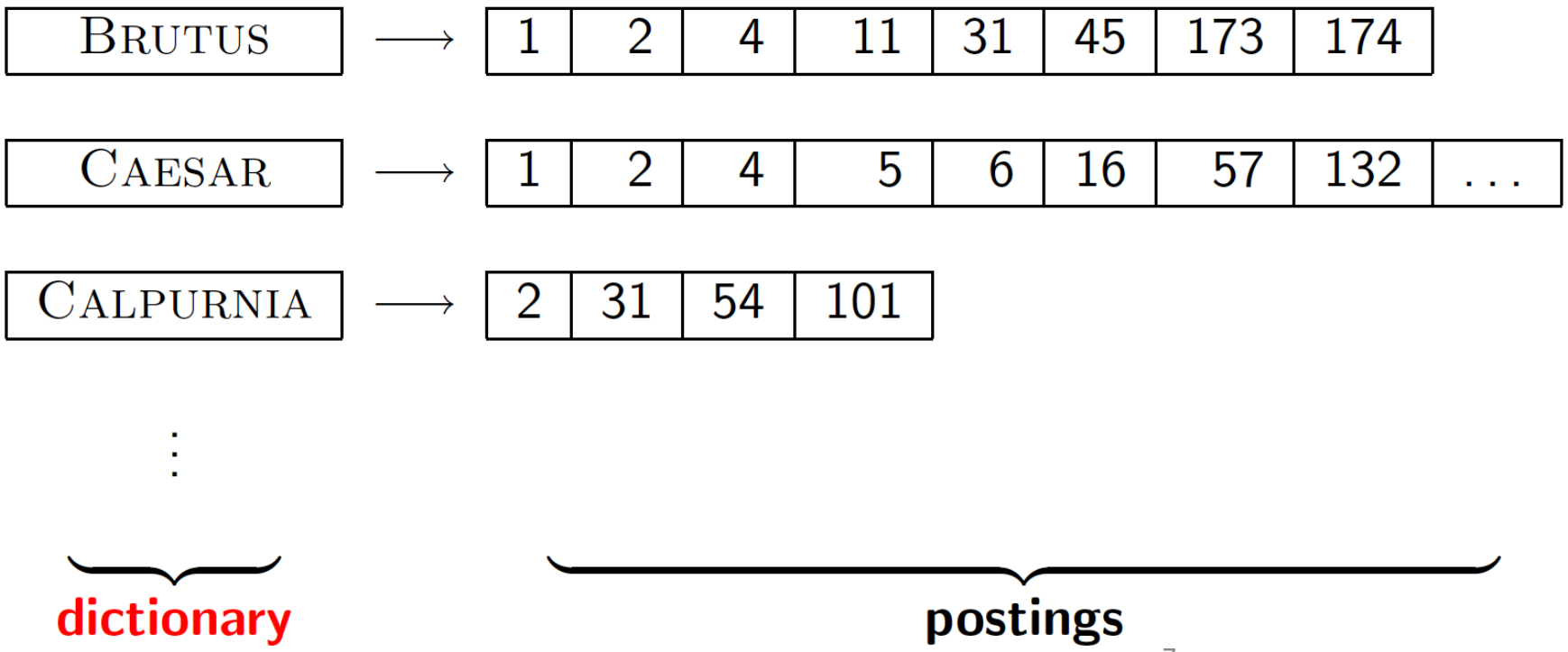
Images copyright 2003 Pearson Education

Data Structures on Disk

- For very large sets of information, we often need to keep most of it on disk
- Examples:
  - Information retrieval systems
  - Database systems
- To handle this efficiently:
  - Keep an index in memory
  - Keep the data on disk
  - The index contains pointers to the data on the disk
- Two most common techniques:
  - Hash tables and B-trees

Dictionaries for inverted indexes

- The dictionary data structure stores the term vocabulary, pointers to each postings list… in what data structure?



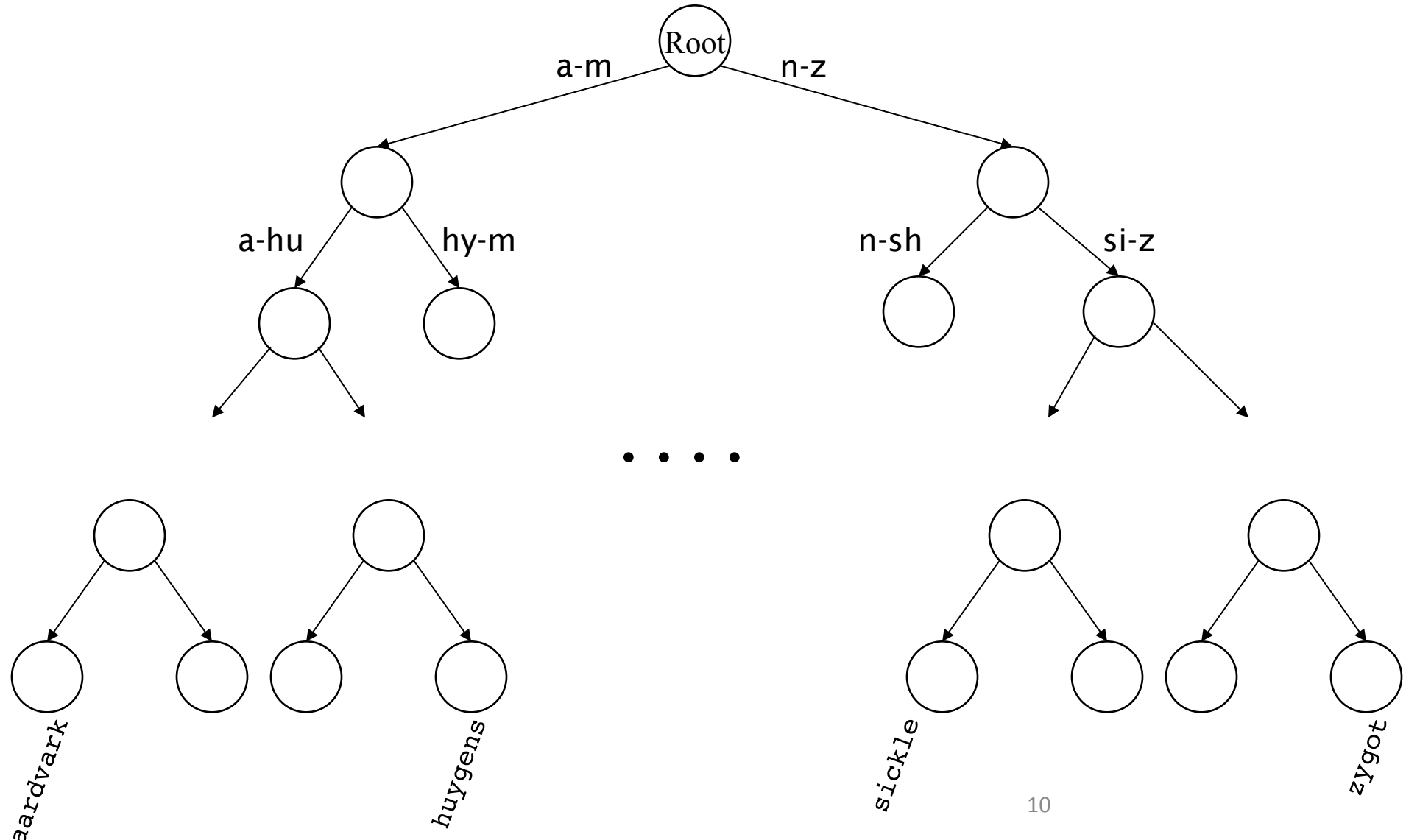
Hashtables for inverted indexes

- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

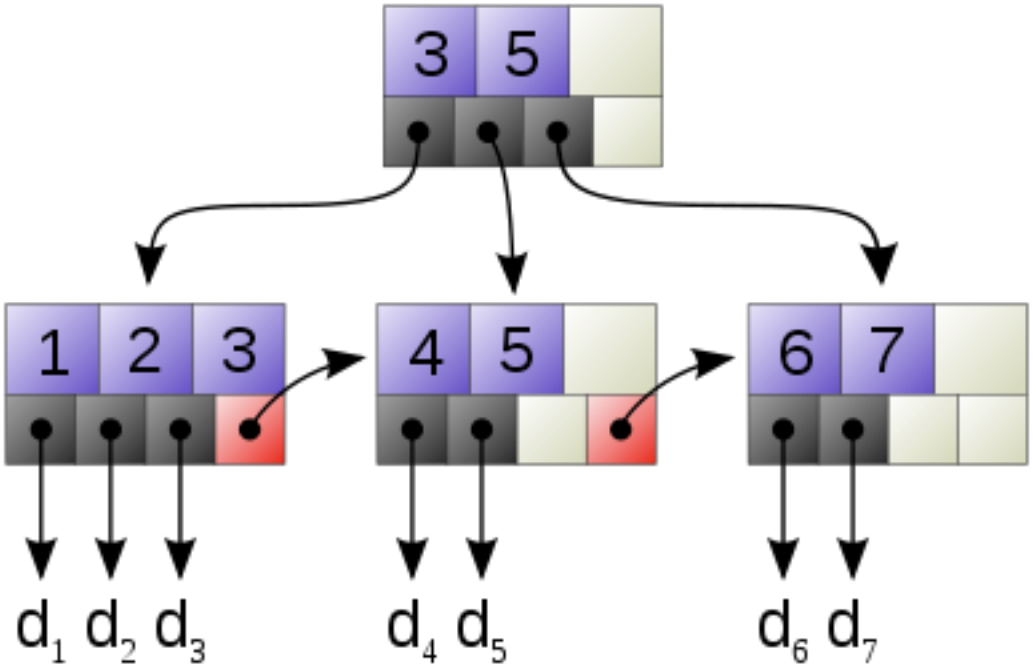
Trees

- Simplest: binary tree
- More common for disk-based storage: B-trees
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *N*) lookup [for *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

Binary tree



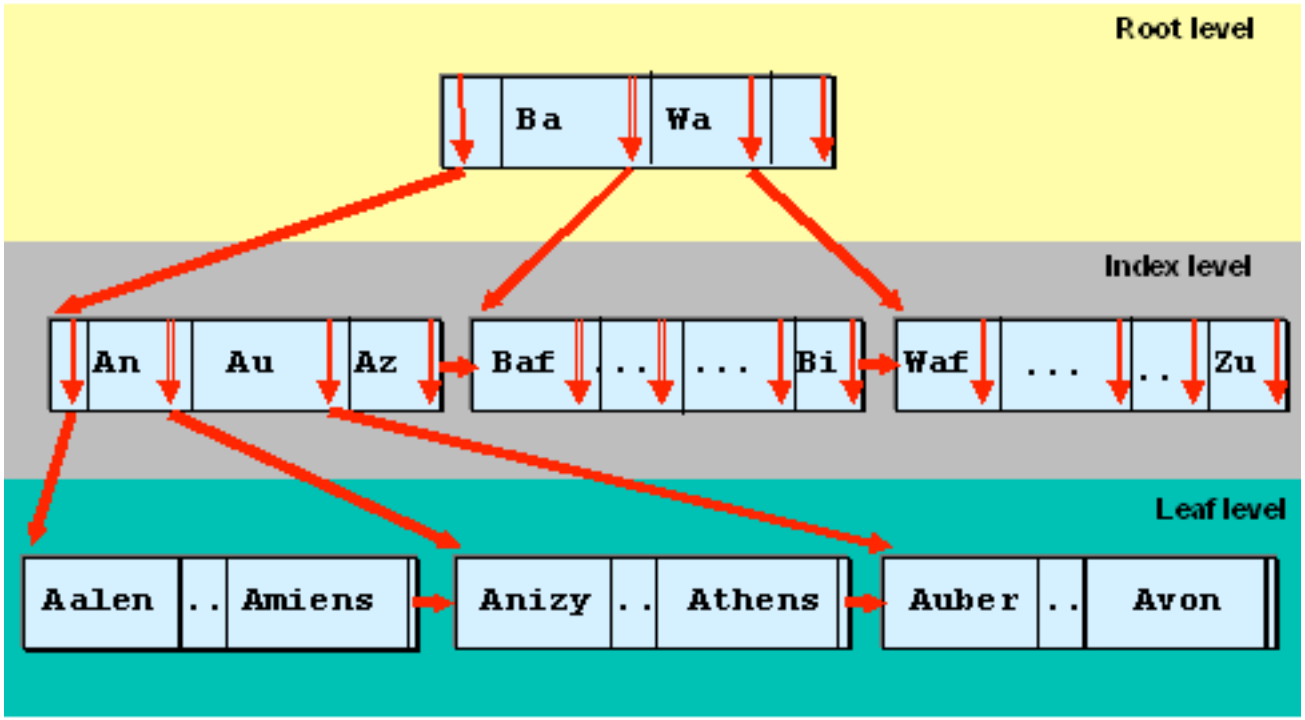
B+tree: Can have more then 2 children



A simple B+ tree example linking the keys 1–7 to data values d1-d7. Note the linked list (red) allowing rapid in-order traversal.

http://www.thefullwiki.org/B%2B_tree

Tree: B*-tree



http://www.sapdb.org/htmhelp/bd/91973b7889a326e10000000a114084/content.htm

Hash Tables vs. B-Trees

- Hash tables great for selecting individual items
  - Fast search and insert
  - O(1) if the table size and hash function are well-chosen
- BUT
  - Hash tables inefficient for finding sets of information with similar keys or for doing range searches
    - (e.g., All documents published in a date range)
  - We often need this in text and DBMS applications
- Search trees are better for this
- B-Trees are a popular kind of disk-based search tree
  - Can keep lists of children in contiguous blocks, disk pages, cached in RAM, etc.

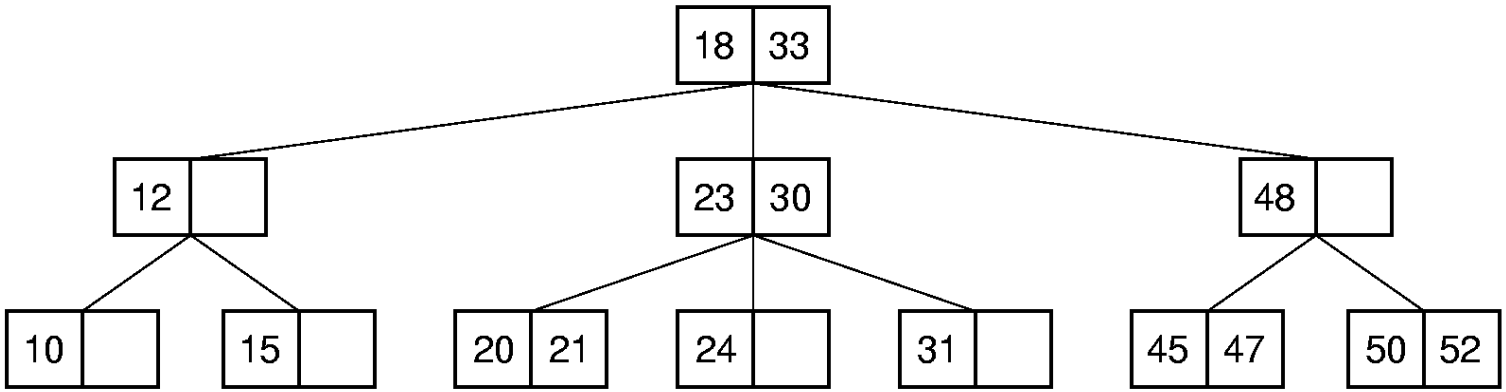
2-3 Tree (Simplest B-Tree)

- The 2-3 Tree has a search tree property analogous to the Binary Search Tree.

- A 2-3 Tree has the following properties:
  - A node contains one or two keys
  - Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
  - All leaves are at the same level in the tree, so the tree is always height balanced.

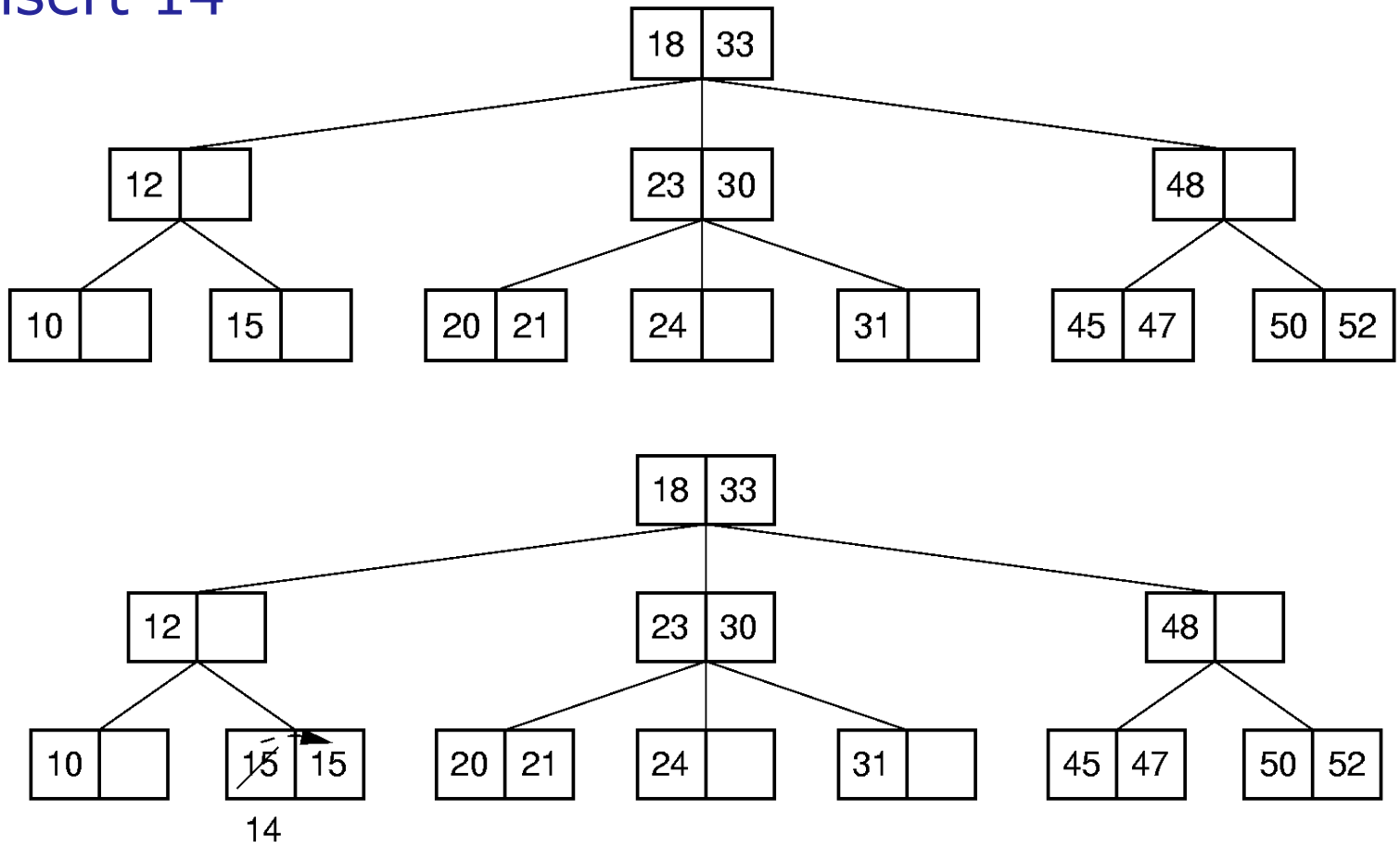
2-3 Tree

One advantage of the 2-3 Tree over the Binary Search Tree is that it can be updated at low cost.



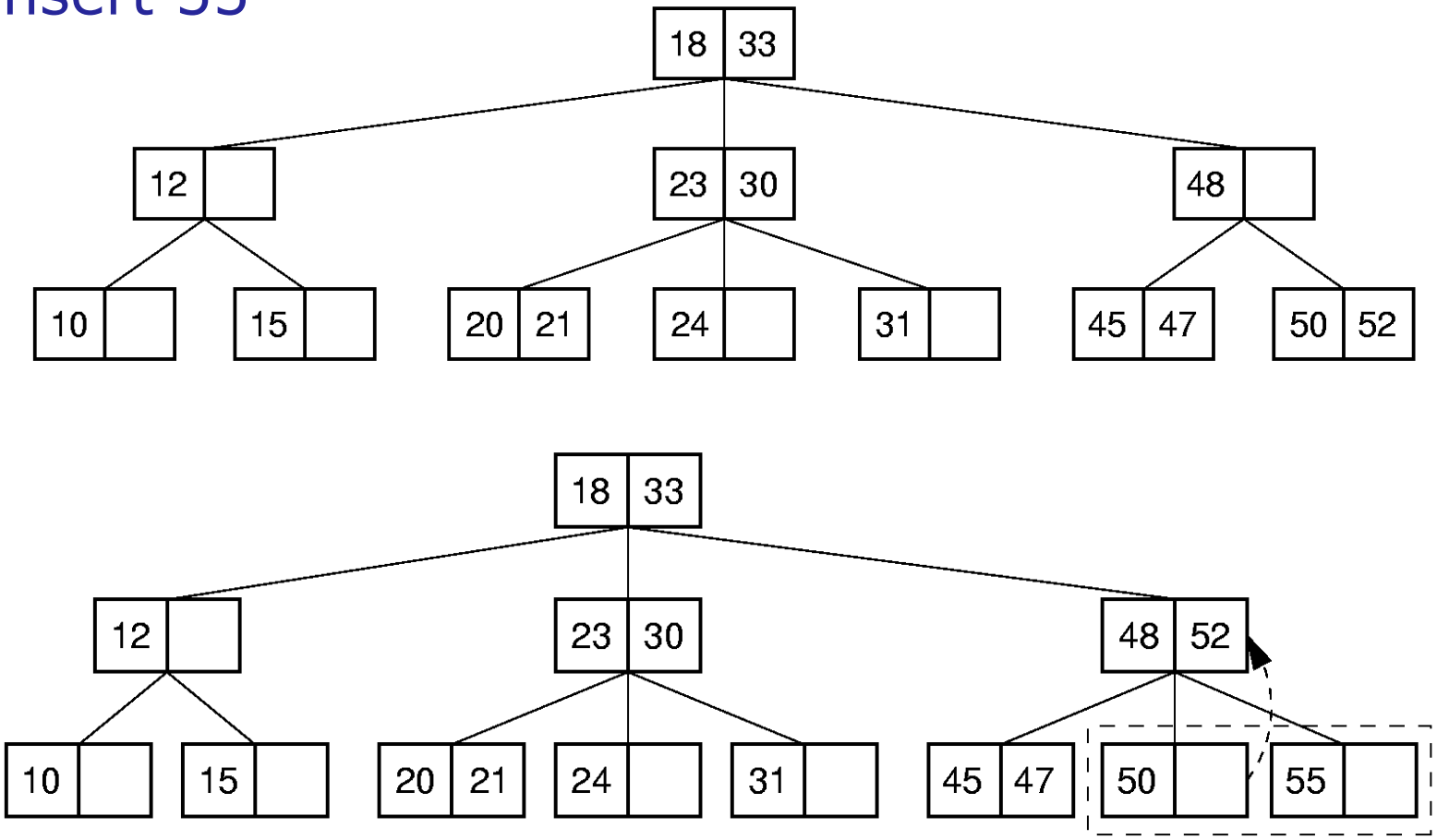
2-3 Tree Insertion

Insert 14



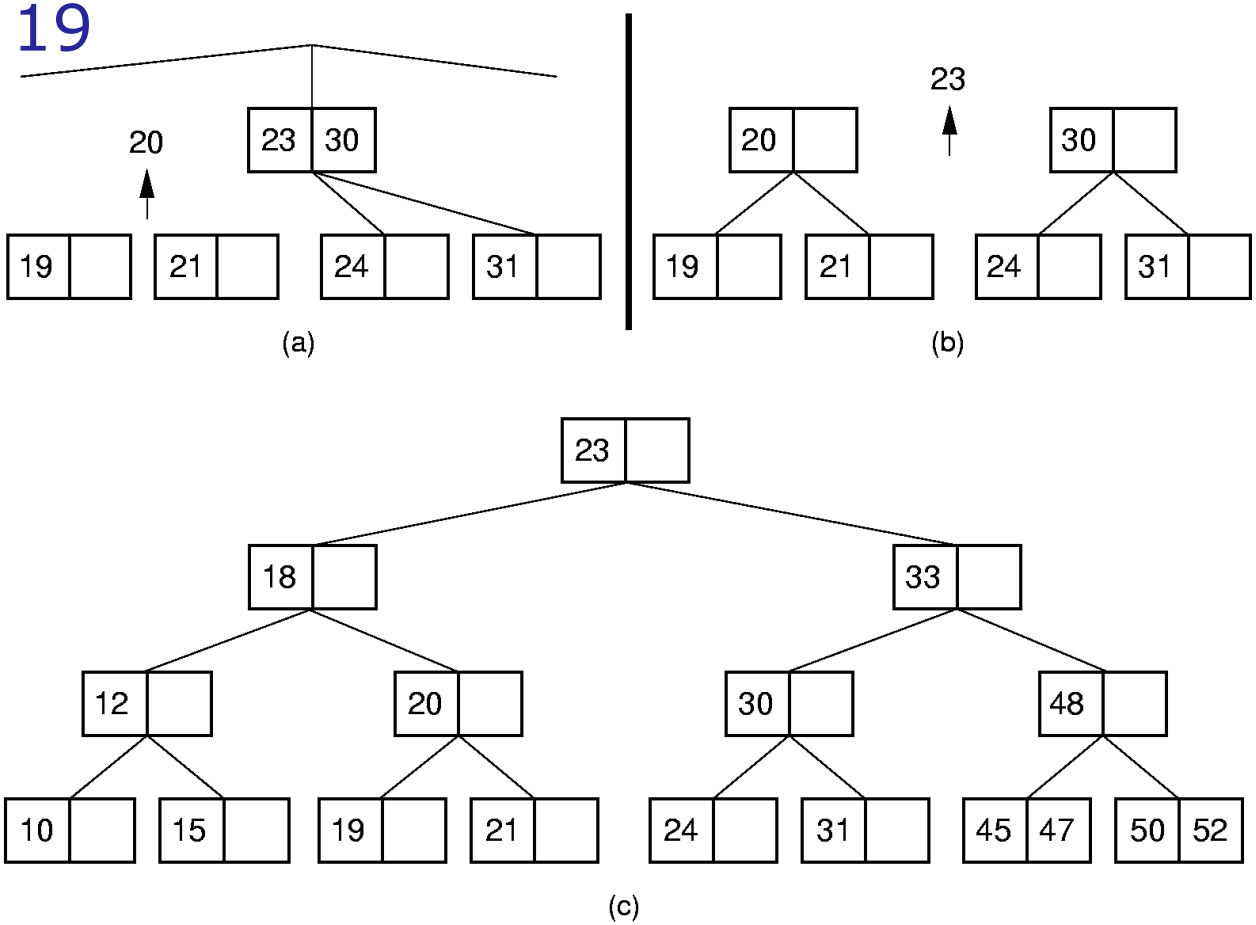
2-3 Tree Insertion

Insert 55



2-3 Tree Insertion (3)

Insert 19



B-Trees

- The B-Tree is an extension of the 2-3 Tree.
- The B-Tree is a standard file organization for applications requiring insertion, deletion, and key range searches.

B-Trees

- Goal: efficient access to information
- Balanced Structure
- Sorted Keys
- Each node has many children
- Each node contains many data items
  - These are stored in an array in sorted order
- B-Tree is defined in terms of rules:

B-Trees

- B-Trees are always balanced.
- B-Trees keep similar-valued records together, which takes advantage of locality of reference.
- B-Trees guarantee that every node in the tree will be full at least to a certain minimum percentage.
  - This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

B-Tree Definition

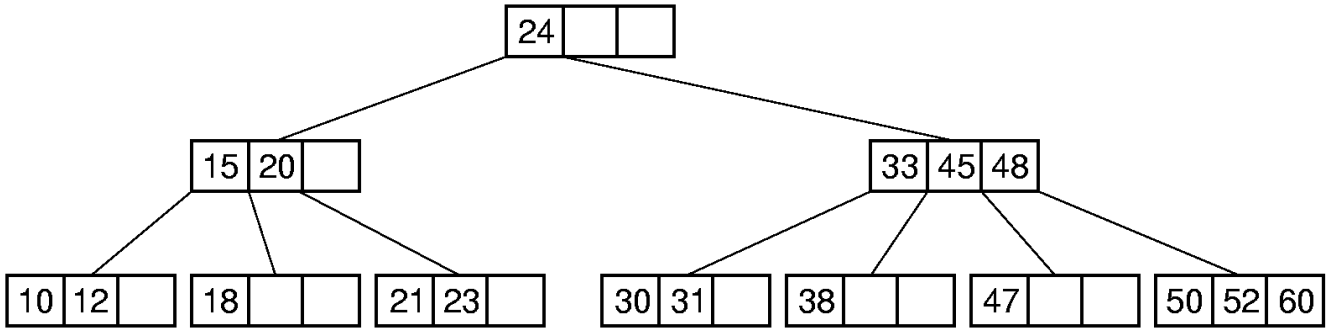
A B-Tree of order *m* has these properties:

- The root is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and *m* children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

A B-Tree's parameters are usually selected to match the size of a disk block.

B-Tree Search

- Search in a B-Tree is a generalization of search in a 2-3 Tree.
  - Do binary search on keys in current node.
    - If search key is found, then return record.
    - If current node is a leaf node and key is not found, then report an unsuccessful search.
  - Otherwise, follow the proper branch and repeat the process.



B^+-Trees

One most commonly implemented form of the B-Tree is the B^+-Tree.

Internal nodes of the B^+-Tree do not store records -- only key values to guide the search.

Leaf nodes store records or pointers to records.

A leaf node may store more or fewer records than an internal node stores keys.

Leaf nodes connected in a linked list for iteration

B+-Tree Space Analysis

B+-Trees nodes are always at least half full.

Asymptotic cost of search, insertion, and deletion of nodes from B-Trees is O(log *n*).

- Base of the log is the (average) branching factor of the tree.

Ways to reduce the number of disk fetches:

- Keep the upper levels in memory.
- Manage B+-Tree pages with a buffer pool.

B+Trees

- Differences from B-Tree
  - Assumes the actual data is in a separate file on disk
  - Internal nodes store keys only
    - Each node may contain many keys (“branchy” or “bushy”)
    - Designed to have shallow height
    - Has a limit on the number of keys per node
      - This way each node can be read in a few operations, stored in RAM, cache etc.
  - Only leaves store data records
    - The leaf nodes refer to memory locations on disk
    - Each leaf is linked to an adjacent leaf

Slide adapted from cis.stvincent.edu

B+Tree and Disk Reads

- Goal:
  - Optimize the B+tree structure so that a minimum number of disk blocks need to be read
- If the number of keys is not too large, keep all of the B+tree in memory
- Otherwise,
  - Keep the root and first levels of nodes in memory
  - Organize the tree so that each node fits within a block to reduce the number of disk reads

Tradeoffs:

- B-trees have faster lookup than B+trees
- But B+Tree is faster lookup if reading from disk using fixed-sized blocks
- In B-tree, deletion more complicated

➨ B+trees often preferred

Slide adapted from lecture by Hector Garcia-Molina

Example Use of B+Trees

- Recall that an inverted index is composed of
  - *A Dictionary* file

    and
  - A *Postings* file
- Use a B+Tree for the dictionary
  - The keys are the words
  - The values stored on disk are the postings

Inverted Index

Dictionary

Term	N docs	Tot Freq
a	1	1
aid	1	1
all	1	1
and	1	1
come	1	1
country	2	2
dark	1	1
for	1	1
good	1	1
in	1	1
is	1	1
it	1	1
manor	1	1
men	1	1
midnight	1	1
night	1	1
now	1	1
of	1	1
past	1	1
stormy	1	1
the	2	4
their	1	1
time	2	2
to	1	2
was	1	2

Postings

Doc #	Freq
2	1
1	1
1	1
2	1
1	1
1	1
2	1
2	1
1	1
1	1
2	1
1	1
2	1
2	1
1	1
2	1
2	1
1	1
1	1
2	1
2	1
1	2
2	2
1	1
1	1
2	1
1	2
2	2

Using B+Trees for Inverted Index

- Use it to store the dictionary
- More efficient for searches on words with the same prefix
  - count* matches count, counter, counts, countess
  - Can store the postings for these terms near one another
  - Minimum disk accesses is needed to get to these

Other Tree Types

- Splay tree
  - A binary search tree with the additional property that recently accessed elements are quick to access again.
- Red-black tree
  - A binary search tree that inserts and deletes in such a way that the tree is always reasonably balanced; worst case running time is O(log n)
- Trie / prefix tree
  - Good for accessing strings, faster for looking up keys than a binary search tree, good for matching prefixes (first sequence of characters).

Problem: String Search

- Determine if, and where, a substring occurs within a string

Approaches/Algorithms:

- Brute Force
- Rabin-Karp
- Tries
- Dynamic Programming

“Brute Force” Algorithm

- The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

```
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
---------------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
 ROADS
---------------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
  ROADS
---------------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
   ROADS
---------------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
    ROADS
---------------------------------------
TWO ROADS DIVERGED IN A YELLOW WOOD
    ROADS
---------------------------------------
```

  - Compared characters are italicized.
  - Correct matches are in boldface type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.

Worst-case Complexity

- Given a pattern M characters in length, and a text N characters in length...
- **Worst case**: compares pattern to each substring of text of length M. For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparisons made**
2) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparisons made**
3) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparisons made**
4) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparisons made**
5) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparisons made**

....

N) AAAAAAAAAAAAAAAAAAAAA*AAAAH*
5 comparisons made *AAAAH*

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Best-case Complexity, String Found

- Given a pattern M characters in length, and a text N characters in length...
- **Best case if pattern found**: Finds pattern in first M positions of text. For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAA **5 comparisons made**

- Total number of comparisons: M
- Best case time complexity: O(M)

Best-case Complexity, String Not Found

- Given a pattern M characters in length, and a text N characters in length...
- **Best case if pattern not found**: Always mismatch on first character. For example, M=5.

1) *A*AAAAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
2) A*A*AAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
3) AA*A*AAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
4) AAA*A*AAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
5) AAAA*A*AAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**

...

N) AAAAAAAAAAAAAAAAAAAAAAA*A*AAAH
1 comparison made *O*OOOH

- Total number of comparisons: N
- Best case time complexity: O(N)

Rabin-Karp Algorithm

- Calculate a hash value for
  - The pattern being searched for (length M), and
  - Each M-character subsequence in the text
- Start with the first M-character sequence
  - Hash it
  - Compare the hashed search term against it
  - If they match, then look at the letters directly
    - Why do we need this step?
  - Else go to the next M-character sequence

(Note 1: Karp is a Turing-award winning prof. in CS here!)
(Note 2: CS theory is a good field to be in because they name things after you!)

Karp-Rabin: Looking for 31415

31415 mod 13 = 7

Thus compute each 5-char substring mod 13 looking for 7

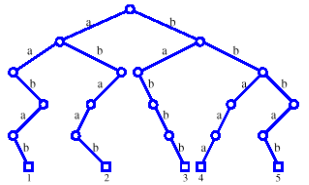
```
2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
-----------
     8

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
  -----------
       9

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
    -----------
        3

2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
            -----------
                7
```

Found 7! Now check the digits

Rabin-Karp Algorithm

- Worst case time?
  - N is length of the string
  - O(N) if the hash function is chosen well

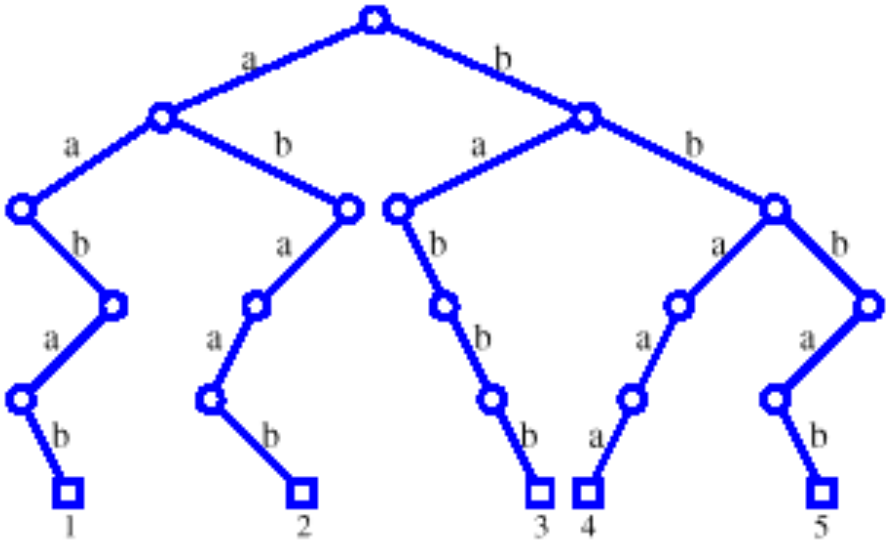
Tries



- A tree-based data structure for storing strings in order to make pattern matching faster
- Main idea:
  - Store all the strings from the document, one letter at a time, in a tree structure
  - Two strings with the same prefix are in the same subtree
- Useful for IR prefix queries
  - Search for the longest prefix of a query string Q that matches a prefix of some string in the trie
  - The name comes from Information Retrieval

Trie Example

The standard trie over the alphabet {a,b} for the set {aabab, abaab, babbb, bbaaa, bbbab}

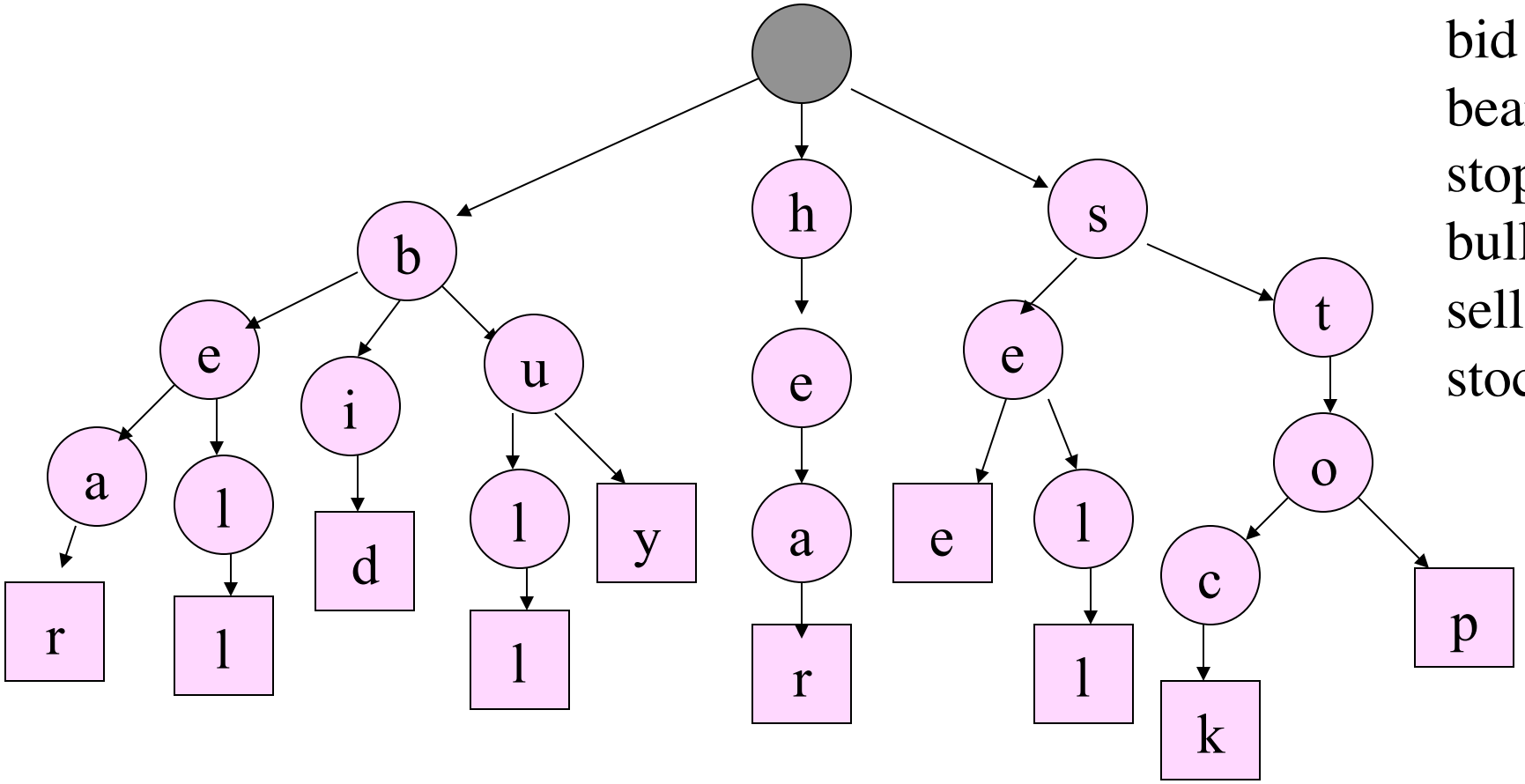


A Simple Incremental Algorithm

- To build the trie, simple add one string at a time
- Check to see if the current character matches the current node.
- If so, move to the next character
- If not, make a new branch labeled with the mismatched character, and then move to the next character
- Repeat

Trie-growing Algorithm

buy
bell
hear
see
bid
bear
stop
bull
sell
stock



Trie, running time

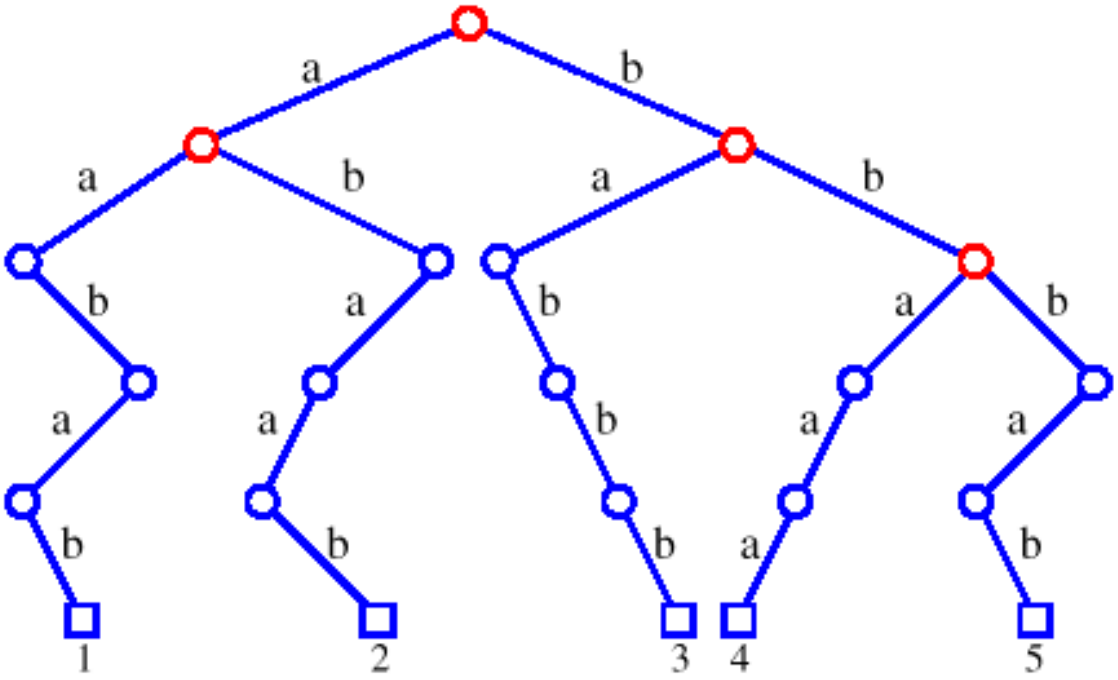
- The height of the tree is the length of the longest string
- If there are S unique strings, there are S leaf nodes
- Looking up a string of length M is O(M)

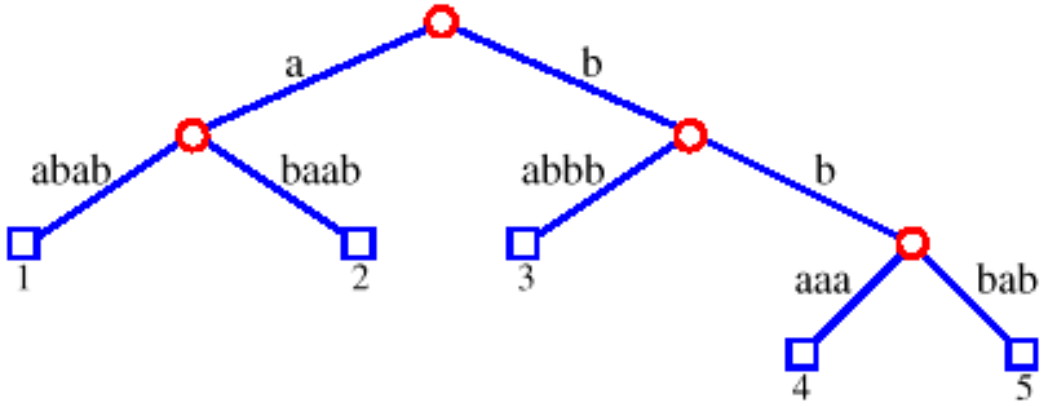
Compressed Tries

- Also known as PATRICIA Trie
  - Practical Algorithm To Retrieve Information Coded In Alphanumeric
  - D. Morrison, Journal of the ACM 15 (1968).
- Improves a space inefficiency of Tries
- Tries to remove nodes with only one child
- The number of nodes is proportional to the number of strings, not to their total length
  - But this just makes the node labels longer
  - So this only helps if an auxiliary data structure is used to actually store the strings

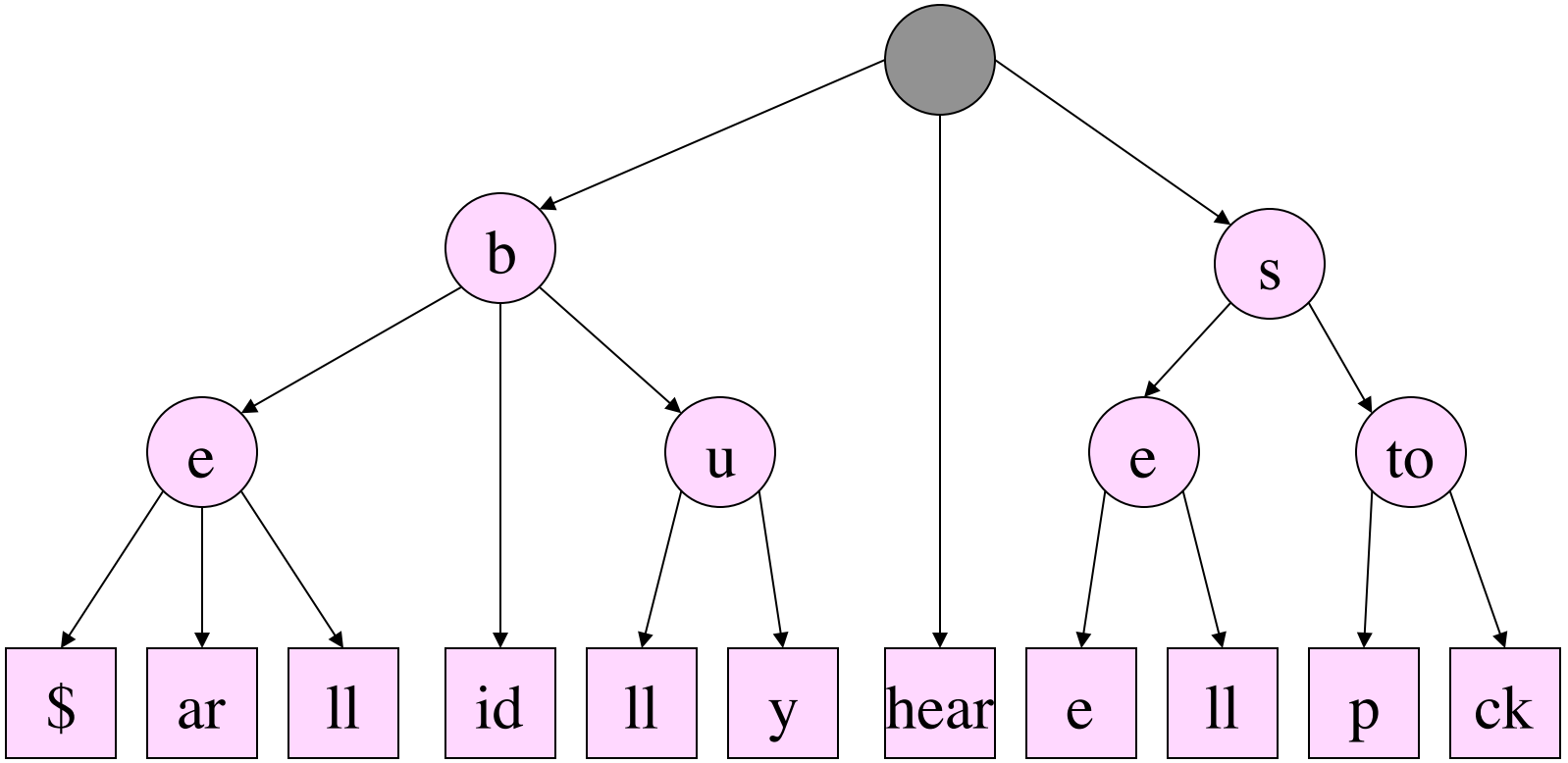
Compressed Tries

Compression is done after the trie has been built up; can't add more items.



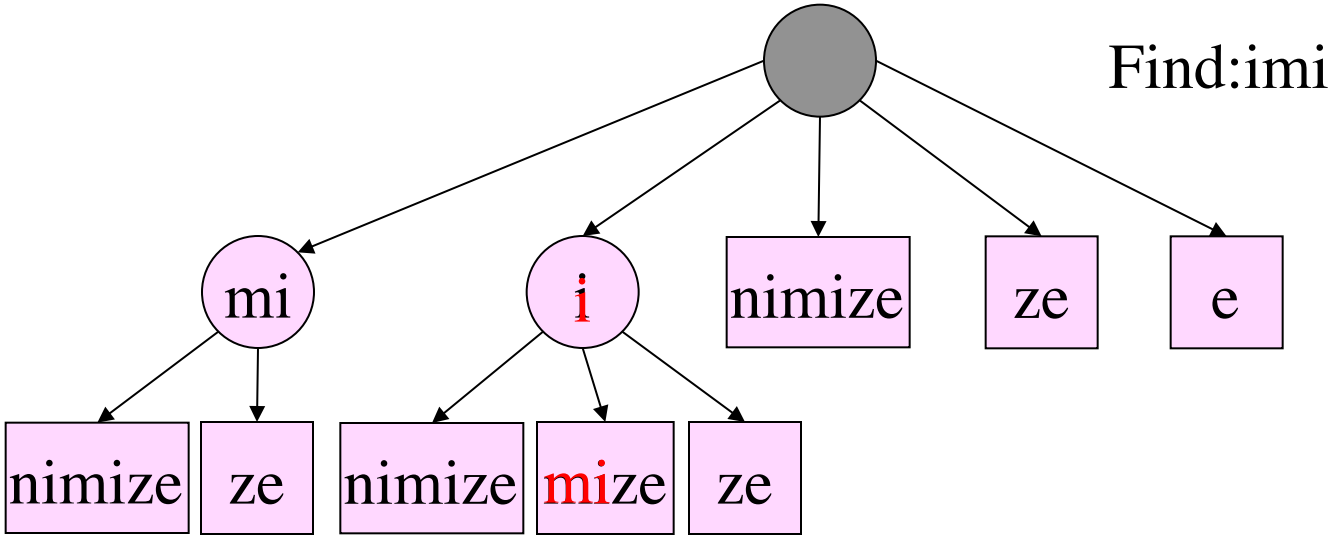


Compressed Trie



Suffix Tries

- Regular tries can only be used to find whole words.
- What if we want to search on suffixes?
  - build*, mini*
- Solution: use suffix tries where each possible suffix is stored in the trie
- Example: minimize



Choosing Data Structures

- Given a huge number of points representing geographic location positions, what data structure would you use for the following:
  - Return the number of points that are within distance D of a given point.

Choosing Data Structures

- Assume you have a large set of data, say 1,000,000 elements. Each data element has a unique string associated with it.
  - What data structure has the optimal algorithmic access time to any element when the only thing you know about the element is the string associated with it?

Choosing Data Structures

- What data structure would you use to get behavior with the following properties:
  - Optimal random access time to any element knowing only the unique string associated with it?
  - Keeps track of the order of insertion into the data structure, so you can acquire the elements back in the same order?
  - Optimal access time to any element knowing only the index of insertion?