Course Notes for

CS 1501
Algorithm Implementation

By

John C. Ramirez

Department of Computer Science

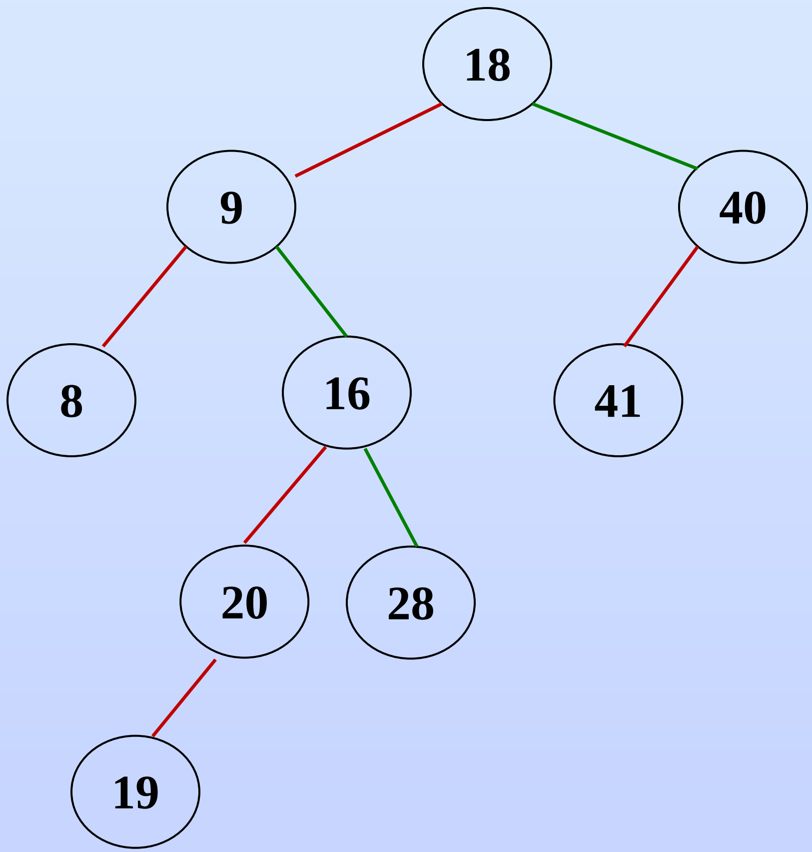
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Consider BST search for key K:
  - For each node T in the tree we have 4 possible results:
    1) T is empty (or sentinel node) indicating item not found.
    2) K matches T.key and item is found.
    3) K < T.key and we go to left child.
    4) K > T.key and we go to right child.
  - Consider now the same basic technique, but proceeding left or right based on the current bit within the key:
    - We still have a tree and 1) and 2) are the same
    - But 3) and 4) will be different.

- Call this tree a Digital Search Tree (DST).
- DST search for key K:
  - For each node T in the tree we have 4 possible results:
    1) T is empty (or a sentinel node) indicating item not found.
    2) K matches T.key and item is found.
    3) Current bit of K is a 0 and we go to left child.
    4) Current bit of K is a 1 and we go to right child.
    - Each time we move down in the tree we go to the next bit in the key.
  - Look at example on next page. →

Digital Search Trees



010010 = 18
001001 = 9
001000 = 8
101000 = 40
101001 = 41
010000 = 16
010100 = 20
011100 = 28
010011 = 19

Note:
Go left on 0 bit
Go right on 1 bit

Digital Search Trees

- Run-times?
  - Given N random keys, the height of a DST should average $O(\log_2 N)$.
    - Think of it this way – if the keys are random, at each branch it should be equally likely that a key will have a 0 bit or a 1 bit.
      - Thus the tree should be well-balanced.
  - In the worst case, we are bound by the number of bits in the key (say it is b number of bits).
    - So in a sense we can say that this tree has a constant run-time, if the number of bits in the key is a constant.
      - This is an improvement over the BST.

- But DSTs have drawbacks:
  1) Data is not sorted.
     - If we want sorted data, we would need to extract all of the data from the tree and sort it.
  2) Bitwise operations are not always easy.
     - Some languages do not provide for them at all, and for others it is costly.
  3) What about keys of different lengths?
     - What if a key is a prefix of another key that is already present?
       - Where would we put it?

4) In addition to our bitwise comparisons, we are still doing comparisons of the entire key.

- Note that we use the bits to branch, but we use the whole key for the equality test.
- May do b key comparisons in the worst case!
- If a key is long and comparisons are costly, this can be inefficient.
- BST does key comparisons as well, but does not have the additional bit comparison.

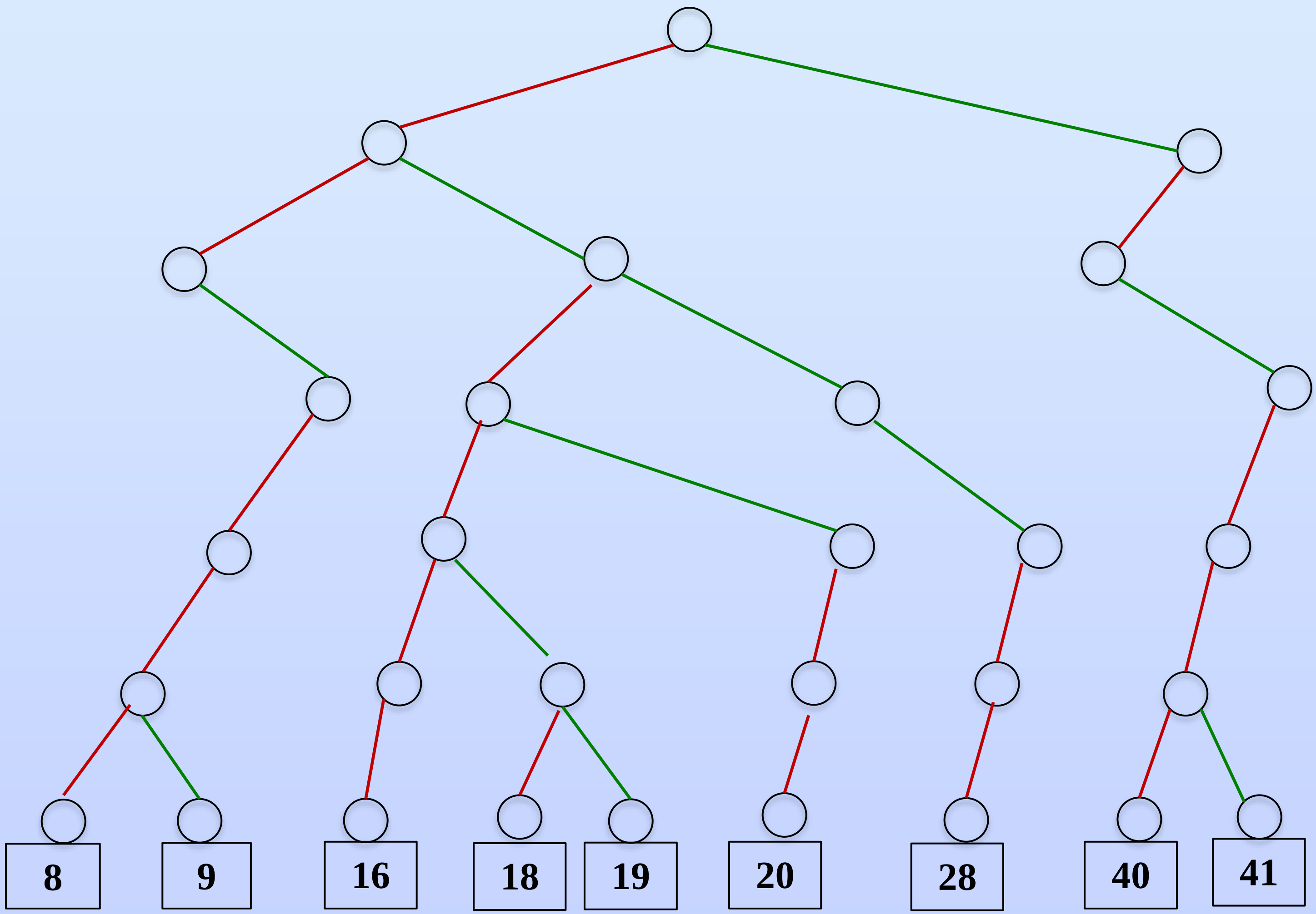
- All of these issues make DSTs less than desirable as an implementation of a symbol table.
  - But maybe we can improve them…

- Let's first address the last problem (4):
  - How to reduce / eliminate the number of comparisons (of the entire key)?
    - This will also resolve (1) and (3).
    - We will resolve (2) soon.
  - We'll modify our tree slightly.
    - All values will be in exterior nodes at leaves in the tree.
    - Interior nodes will not contain keys, but will just direct us down the tree toward the leaves, based on the current bit value.
    - In fact, we do not need to store the keys themselves at all.

- Each path from the root to a leaf represents a unique key.
  - Thus the keys are “stored” within the trie, but implicitly via the paths to the leaves rather than explicitly.

- This gives us a Radix Search Trie:
  - Trie is from reTRIEval (see text).
  - See example with same data as DST on next slide. →
  - Note that the tree is taller because all bits in the keys are used.
  - Note also that the keys are being stored in the leaves.
    - This is assuming the symbol table is storing (key, key) pairs.
    - Normally, the symbol table will have (key, value) pairs where the value differs from the key.

Radix Search Tries



010010 = 18
001001 = 9
001000 = 8
101000 = 40
101001 = 41
010000 = 16
010100 = 20
011100 = 28
010011 = 19

- Benefit of simple Radix Search Tries
  - No comparisons of the **entire key**
    - Thus solves problem (4) of DSTs
  - Do not require keys to be stored
    - Paths from root implicitly define the keys
  - Can handle keys with common prefixes
    - Simply allow interior nodes to have values as well
      - Put value in next node after last bit of key to indicate the termination of a key
      - Thus solves problem (3) of DSTs

- An "inorder" traversal of the tree will give the data in "sorted" order (problem (1)).
  - Look again at the tree in slide 11 to see this.

- Drawbacks:
  - The tree will have more overall nodes than a DST.
    - Bits to each key are fully elaborated as branches in the tree.
    - Thus, we need to traverse b+1 levels of the tree to find a key of b bits, regardless of the number of keys in the tree.
  - Many nodes will have an unused value reference.
    - Every node has this reference, but most will not store a value.
  - We are still doing bit operations (problem 2).

- Asymptotic run-time is similar to DST
  - Since all bits are elaborated, every successful search requires b bit comparisons where b is the number of bits in the key.
    - This was the worst case for the DST
    - **However, we require no comparisons of the entire key**
  - So, again, a **benefit to RST is that the entire key does not need to be stored or compared**

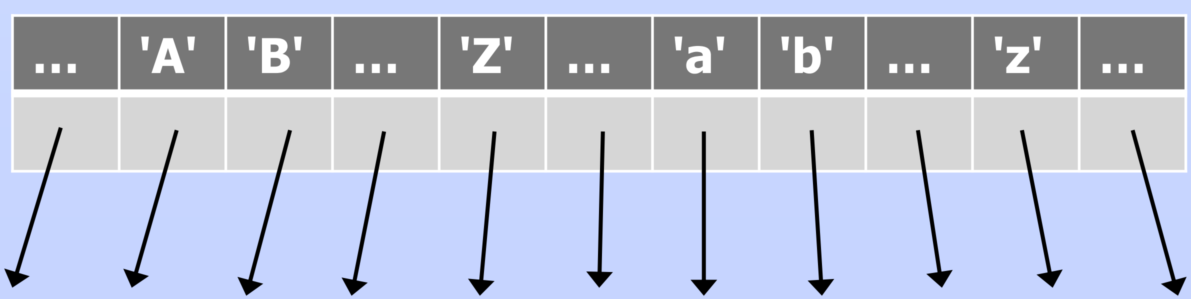
- How can we improve tries?
  - If we can reduce the heights we will shorten the path length to the leaves
  - If we can process multiple bits at a time we can avoid bit operations
- We will examine a two variations that improve over the basic trie
  - Multiway Trie (in text as R-way Trie)
    - Allow more than 2 branches per node
  - de la Brandeis Tree (DLB)
    - Maintain fast lookups while reducing memory

- RST that we have seen considers the key 1 bit at a time.
  - This causes a maximum height in the tree of b, where b is the bits in the longest key stored.
  - A search miss, as explained in Proposition H on p. 743 of Sedgewick, will require on average $\lg_2 N$ nodes to be examined (similar to BST).
- If we considered m bits at a time, then we could reduce the tree height.
  - Maximum height is now b/m since m bits are consumed at each level.
  - Let $R = 2^m$
    - Average nodes on search miss is now $O(\log_R N)$, since we branch in R directions at each node.

- Let's look at an example
  - Consider 2^{20} (1 meg) keys of length 40 bits
    - Simple RST will have
      - Worst Case height = 40
      - Ave. nodes per miss = $O(\log_2[2^{20}]) \approx 20$
    - Multiway Trie using 8 bits would have
      - Worst Case height = 40/8 = 5
      - Ave. nodes per miss = $O(\log_{256}[2^{20}]) \approx 2.5$
- This is a considerable improvement
- Let's look at an example using character data (ex: a String)

- Consider a String: DATUM
  - This is 40 bits, 8 bits per character
    - If we considered it 1 bit at a time, our tree would have a height of 40
    - However, if we can consider it 1 character (8 bits) at a time we can reduce the height to 40/8 = 5
  - How can we do this?
    - Each bit has 2 possible values (1 or 0)
    - Each character has $2^8 = 256$ possible values (ASCII)
    - Thus we need to somehow allow 256-way branching at each node in our tree
    - How can we do this?
      - Interesting question!

- Let's define a node in our tree to be an array of 256 pointers (references)
  - Each reference will point to a child node on the next level of the tree
  - Our array will be indexed on all of the possible character values in the ASCII set
    - Recall that ASCII values are simply integers from 0-255 which are interpreted as character values
    - We can use these values to index our array, giving us:



- Now the value of each character in the String will determine the index and branch to take
- Branching based on characters reduces the height greatly
  - If a string with K characters has n bits, it will have n/8 (= K) characters and therefore at most K levels
- Character / byte comps are simple, so, we eliminate the bit comparison problem mentioned previously as item (2)
- Thus, given N strings in a multiway trie, we have
  - Theta(1) for a successful search or insert
    - Theta(K) in terms of the key length
  - Theta(lg_RN) node accesses for unsuccessful search

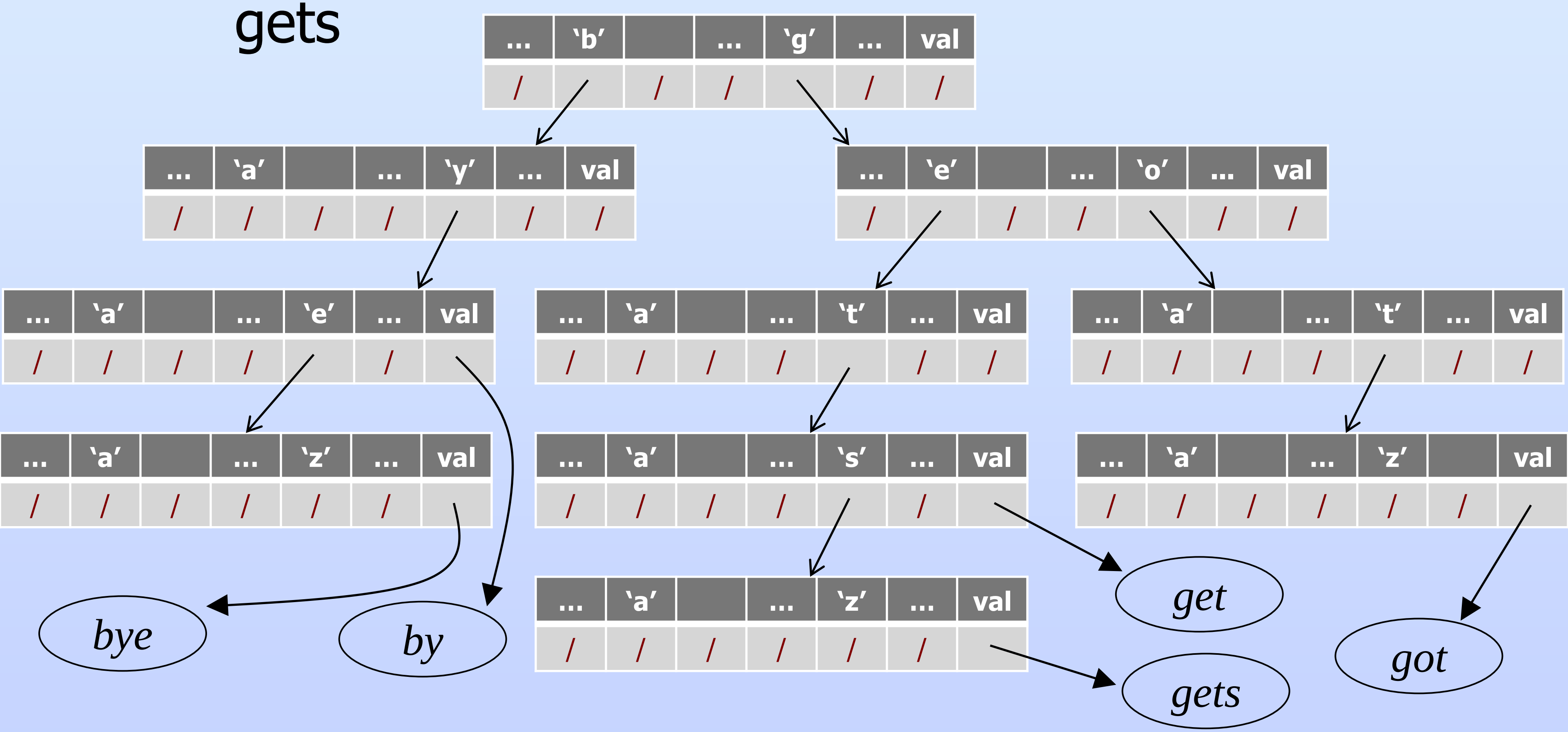
- So what is the catch (or cost)?
  - **Memory**
    - Multiway Tries use considerably more memory than simple tries
  - Each node in the multiway trie contains **R (= 2^m) pointers/references**
    - In example with ASCII characters, R = 256
    - With simple words, R could be 26
  - Many of these are unused, especially
    - During **common paths** (prefixes), where there is no branching (or "one-way" branching)
      - > Ex: through and throughout
    - At the **lower levels** of the tree, where previous branching has likely separated keys already
      - > Unique suffixes of words

- Multiway Trie Structure
  - As mentioned, a node represents the possible characters at a given position in the key
    - This is represented by an array of references
    - Indices on array are the characters in the character set
    - A non-null value at an index indicates that a key exists with that char at that point in the key
    - Branching occurs when more than one of the pointers is non-null
  - Each new key added will have some prefix represented by nodes already in the trie, followed by a suffix represented by new nodes

- Keys of different lengths are handled as follows:
  - In addition to the array of pointers, each node has a field to indicate if a key terminates there
    - It is set to true to indicate that the string up to that point is a key
    - It is set to false to indicate that the string up to that point is not a key
      - Rather it is only a prefix to a key
- Note that keys are not directly stored in the trie at all
  - Rather each path from the root to a terminator node represents an individual key in the trie

- Alternatively, if we make our trie symbol table, for each node we can have a reference to store the associated value
  - Recall that a symbol table maps keys to values
  - For a given node, if the value reference is non-null, then that node terminates a key, and the key (expressed as a path from the root to that node) is thus in the trie
    - Note that we need to proceed one level down from that last match
  - For a given node, if the value reference is null, then that node is within a key but does not terminate it, so the path from the root to that node is a prefix of a key only

- Ex: Consider strings: bye, by, get, got and gets



- See TrieST.java and DictTest.java

- Let's review again the 4 issues with DSTs:
  1) Data is not sorted – As with the simple RST a traversal here (done correctly) will produce the data in sorted order (look at figure in previous slide)
  2) Bitwise operations are not easy – comparing 8 bits (a char) at a time IS easy and efficient
  3) What about keys of different lengths? We just saw how to handle this with a termination field or value reference
  4) Comparisons of the entire key – these are not done at all

- Multiway tries solve these issues nicely
  - But with a cost of a lot of memory
  - This is especially evident when the symbol table is not very dense
    - i.e. most of the possible string keys do not actually exist in the symbol table
      - Yet we have references to all possible children in each node
    - Think about a dictionary of real words vs. the number of possible permutations of letters
  - Is there a way to keep (mostly) the "good" of a multiway trie while reducing memory?