Lecture 15

- Radix search
- Digital search trees
- Multiway tries
- Ternary tries

Whole-key search

- Consider the usual find operation in a linked list
  - Nodes in the list hold keys
  - The key you are looking for is compared to the key in each node in sequence, until found or until reaching the end of the list
- Consider the usual find operation in a binary search tree
  - Nodes in the tree hold keys
  - The key you are looking for is compared to the key in the root node; depending on the result of the comparison, the search continues recursively in the left or right subtree, until found or reaching a null reference
- In each case, when a comparison is done, the entire search key is compared to the key in the current node. (This comparison is usually assumed to take O(1) time...)
  - For example, if keys are Strings, all the characters in the search key String may be compared to all the characters in the current node key String
  - ... or if keys are double precision floating point variables, all 64 bits in both keys are used for the comparisons
- Another approach is possible: each comparison only uses a small piece of the keys
- This leads to the idea of *radix search*

Radix search

- A key can be considered as a sequence of smaller segments
- Call each segment a *digit*
- Each digit can take on values from some set
- The size of this set of possible digit values is called the *base* or *radix*
- Examples:
  - An int can be considered as a sequence of 32 1-bit digits
    - The radix is 2: each digit can have one of two values
  - An int can be considered as a sequence of 8 4-bit digits
    - The radix is 16: each digit can have one of 16 values
  - A Java String can be considered as a sequence of 16-bit chars
    - The radix is 65536: each digit can have one of 65536 values

- The idea of radix search is: search is guided by comparisons that involve only one digit of the keys at a time
- This will have some advantages, but requires somewhat different data structures and algorithms: digital search trees, multiway tries, ternary tries

Digital search trees

- Digital search trees (DST's) are binary trees
- In a DST, both internal nodes and leaf nodes hold keys
- But, unlike a regular binary search tree, branching when searching a DST is determined by comparing just one bit of the keys at a time (i.e., radix search with radix 2)
  - however: whole-key comparisons are also done
- The basic search algorithm to find `key` in a DST:

```
1.  Set currentNode = root, i = 0.
2.  If currentNode is null, return "not found."
3.  Compare key to currentNode.key.  If equal, return "found".
4.  Look at the value of the ith bit in key.  If 0,
      set currentNode = currentNode.left; else
      set currentNode = currentNode.right
5.  Set i = i+1 and Go to 2.
```

- The basic insert algorithm in DST's follows the search algorithm, except:

```
2.  If currentNode is null, create a new node newNode and
      set newNode.key = key.  Splice newNode into the tree as a new
      leaf in place of the null reference. Return.
```

Digital search tree: example

- Consider the following 6 5-bit keys with values as shown

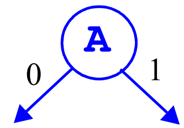
```
A  00001
S  10011
E  00101
R  10010
C  00011
H  10100
```

- We will insert them in that order into an initially empty DST

- (In this example, the 0*th* bit is the leftmost bit)

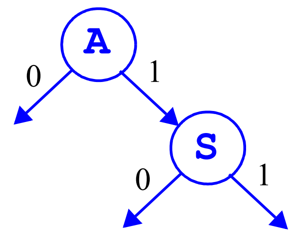
Digital search tree: example

```
A   00001
S   10011
E   00101
R   10010
C   00011
H   10100
```



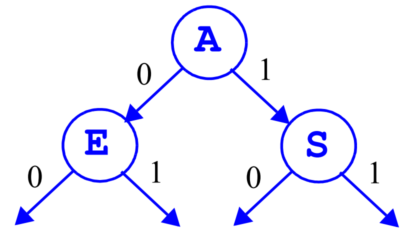
Digital search tree: example

```
A  00001
S  10011
E  00101
R  10010
C  00011
H  10100
```



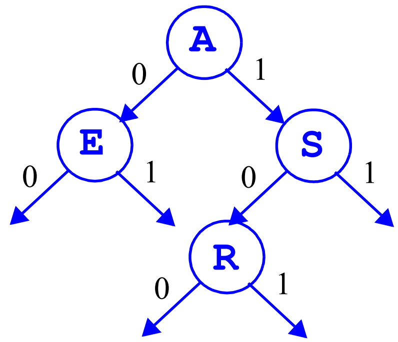
Digital search tree: example

```
A  00001
S  10011
E  00101
R  10010
C  00011
H  10100
```



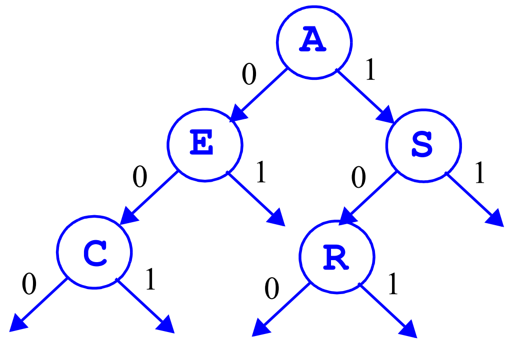
Digital search tree: example

A	00001
S	10011
E	00101
R	10010
C	00011
H	10100



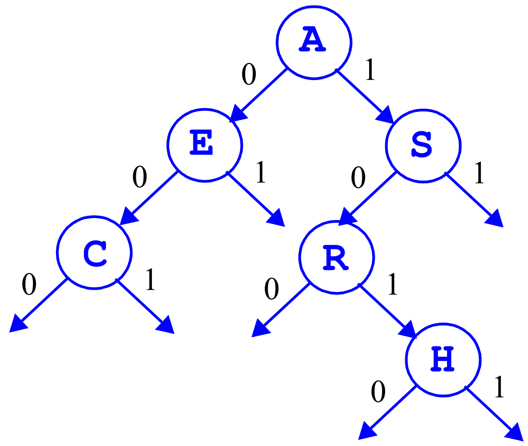
Digital search tree: example

A	00001
S	10011
E	00101
R	10010
C	00011
H	10100



Digital search tree: example

A	00001
S	10011
E	00101
R	10010
C	00011
H	10100



Digital search tree properties

- In a DST, each key is somewhere along the path specified by the bits in the key (this guarantees that the find and insert algorithms will work)
- Suppose keys each contain no more than B bits. Then:
  - The worst case height of a DST containing N keys is B
  - Compare: worst case height in a regular BST containing N keys is N, which with B-bit keys can be as much as 2^B
- So, when N is large and B is comparable to $\log_2 N$, DST's give worst-case guarantees comparable to balanced BST's, and are much easier to implement

- However, DST's do not have a strong key ordering property:
  - The key in a node X can be larger or smaller than keys in either of its subtrees
  - So, an ordinary traversal of a DST is not guaranteed to visit keys in sorted order

- So consider *tries*:
  - Give strong key ordering property
  - Preserve the nice worst-case properties of DST's
  - Do true radix search, avoiding repeated full-key comparisons

Binary tries

- Binary tries are binary trees
- Unlike a BST or DST, in a binary trie, only leaf nodes hold keys
- However, like a DST, search in a binary trie is guided by comparing keys one bit at a time (radix search with radix 2)
- The basic search algorithm to find `key` in a binary trie:

```
1.  Set currentNode = root, i = 0.
2.  If currentNode is null, or i > # of bits in key,
      return "not found".
3.  If currentNode is a leaf, and i == # of bits in key,
      return "found".
4.  Look at the value of the ith bit in key.  If 0,
      set currentNode = currentNode.left; else
      set currentNode = currentNode.right
5.  Set i = i+1 and Go to 2.
```

- The basic insert algorithm in binary tries is straightforward. (Use find algorithm to find where the key must go, if it is to be found later! And put it there.)

Binary trie: example

- Consider the following 6 keys with values as shown

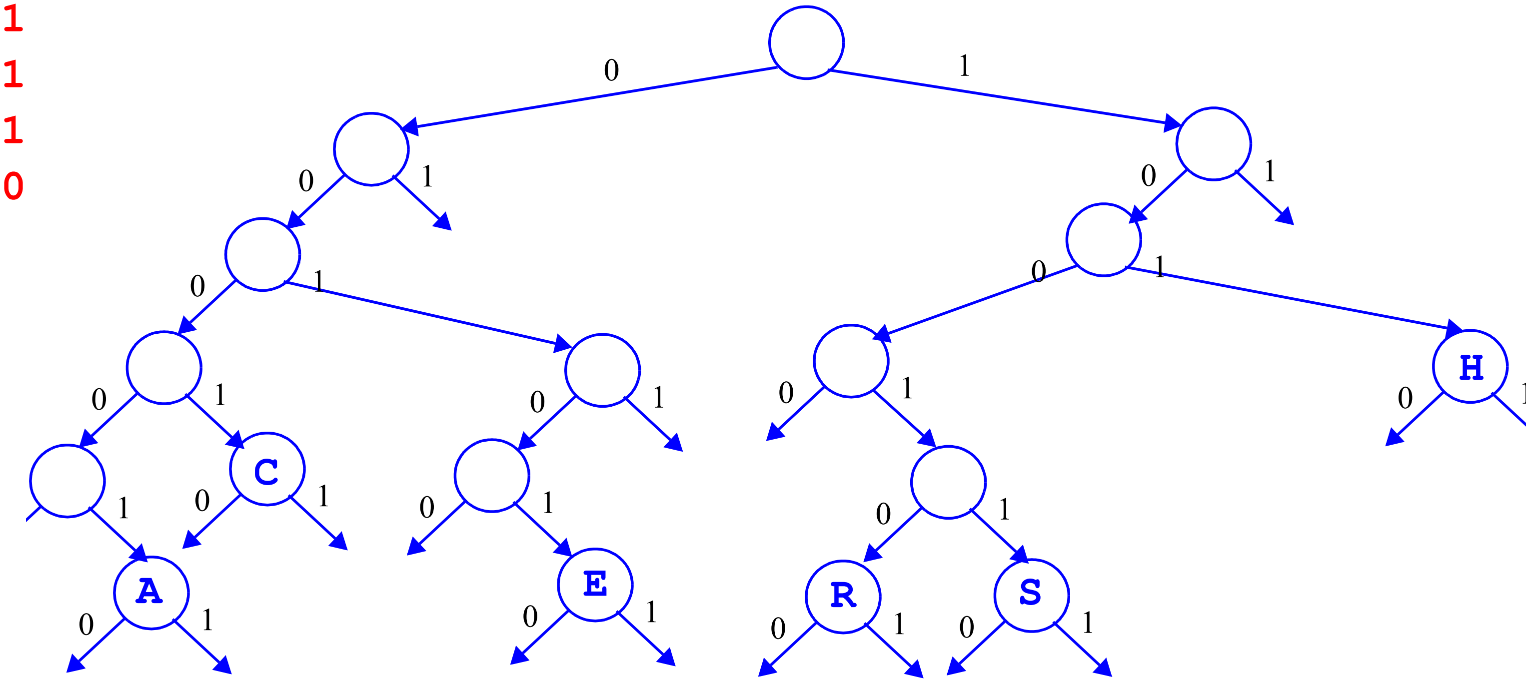
```
A  00001
S  10011
E  00101
R  10010
C  0001
H  101
```

- We will insert them in that order into an initially empty binary trie

- (In this example, the 0*th* bit is the leftmost bit)

Binary trie: example

```
A   00001
S   10011
E   00101
R   10010
C   0001
H   101
```



Binary trie properties

- The structure of a binary trie depends only on the keys in it, not on the order in which they were inserted
- Binary tries do have a strong key ordering property: At a node X, all keys in X's left subtree are smaller (by lexicographic ordering) than keys in X's right subtree
  - So, an ordinary traversal of a binary trie visits keys in sorted order
- Suppose keys contain at most B bits. Then:
  - The worst case height of a binary trie containing N keys is B
  - Compare: worst case height in a regular BST containing N keys is N, which with B-bit keys can be as much as 2^B
- So when N is large and B is comparable to $\log_2 N$, binary tries give worst-case guarantees comparable to balanced BST's, and are much easier to implement

- However, a binary trie as shown cannot contain two keys, one of whose binary representation is a prefix of the other's
  - (If all keys contain the same number of bits, this condition is certainly satisfied, but it may not be satisfied if keys have different lengths)
- This problem can be solved by having a boolean "`end`" bit that can be set in a node, indicating that this node ends a bit sequence that represents a stored key

Binary tries with "end" bits

- Suppose each binary trie node contains an "`end`" bit that is true if this node represents a key
- Then internal nodes, as well as leaf nodes, can hold keys
- The basic search algorithm to find `key` in a binary trie with end bits:

```
1.  Set currentNode = root, i = 0.
2.  If currentNode is null, or i > # of bits in key,
      return "not found".
3.  If currentNode.end is true, and i == # of bits in key,
      return "found".
4.  Look at the value of the ith bit in key.  If 0,
      set currentNode = currentNode.left; else
      set currentNode = currentNode.right
5.  Set i = i+1 and Go to 2.
```

- The basic insert algorithm in binary tries with end bits is also straightforward...

Binary trie with end bits: example

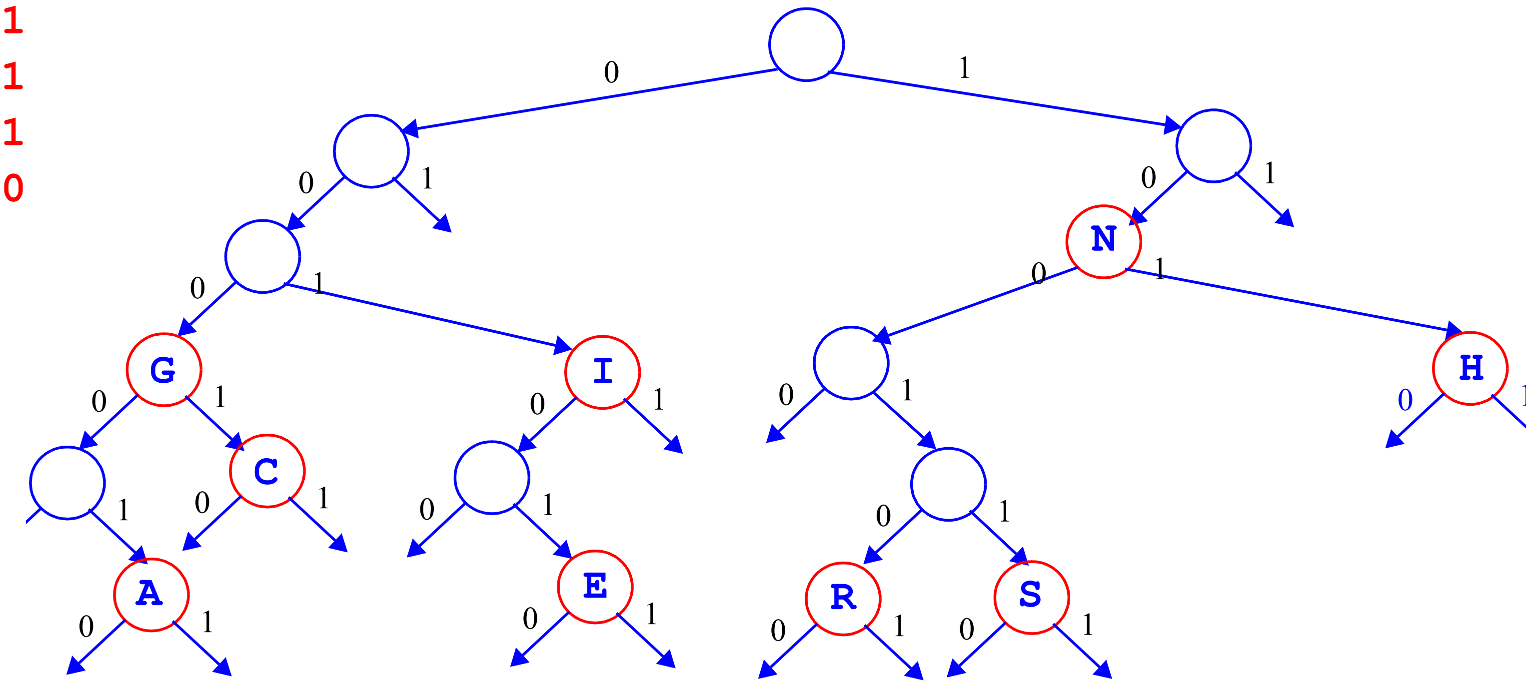
- Consider the following 8 keys with values as shown

```
A  00001
S  10011
E  00101
R  10010
C  0001
H  101
I   001
N   10
G   000
```

- We will insert them in that order into an initially empty binary trie with end bits
- (In this example, the 0*th* bit is the leftmost bit; and nodes with `end` bit true are shown in red)

Binary trie with end bits: example

A 00001
S 10011
E 00101
R 10010
C 0001
H 101
I 001
N 10
G 000



Binary tries with end bits properties

- Binary tries with end bits are able to store keys whose binary representations are prefixes of each other
- And in addition, they have all the other nice properties of binary tries
  - (to visit keys in lexicographic order, use a pre-order traversal)
- In particular, if keys have at most B bits, the worst-case number of comparisons for a search is B

- But what if we could use radix search with radix > 2?
  - If each digit in a key has r bits, the radix is $R = 2^r$
  - ... and if keys have at most B bits, the worst-case number of comparisons would be only B/r

- This leads to the idea of *multiway tries*

Multiway tries

- A binary trie uses radix search with radix 2; a multiway trie uses radix search with radix R > 2
  - multiway tries are sometimes called R-ary tries
- If each digit in a key has r bits, the radix is $R = 2^r$, and if keys have at most B bits, the worst-case number of comparisons would be only B/r
- However, to implement this idea, a node in the trie must be able to have as many as R children
- Examples:
  - Keys are words made up of lower-case letters in English. There are 26 different lower-case letters in English, so a R-ary trie with R=26 could hold these keys. (This specific variant is sometimes called an "alphabet trie")
  - Keys are decimal integers made up of decimal digits. There are 10 different decimal digits, so a R-ary trie with R=10 could hold these keys
  - Keys are 128-bit IEEE high precision floating point numbers. Consider each as made up of 32 4-bit nybbles. There are $2^4 = 16$ different nybble values, so a R-ary trie with R=16 could hold these keys (note that lexicographic ordering of such keys is not the same as their numeric ordering)

Multiway tries with "end" bits

- A multiway trie with radix R uses nodes each with an adjacency list (or adjacency array, or adjacency table) of R child pointers, indexed 0,...,R-1
- Also suppose each node contains an "`end`" bit that is true if this node represents a key
  - Then internal nodes, as well as leaf nodes, can correspond to keys
- The basic search algorithm to find `key` in a multiway trie with end bits:

```
1.  Set currentNode = root, i = 0.
2.  If currentNode is null, or i > # of digits in key,
      return "not found".
3.  If currentNode.end is true, and i == # of digits in key,
      return "found",
4.  Look at the value of the ith digit in key; let this digit
      value be d.
      set currentNode = currentNode.children[d]
5.  Set i = i+1 and Go to 2.
```

- The basic insert algorithm in multiway tries with end bits is straightforward

Multiway trie with end bits: example

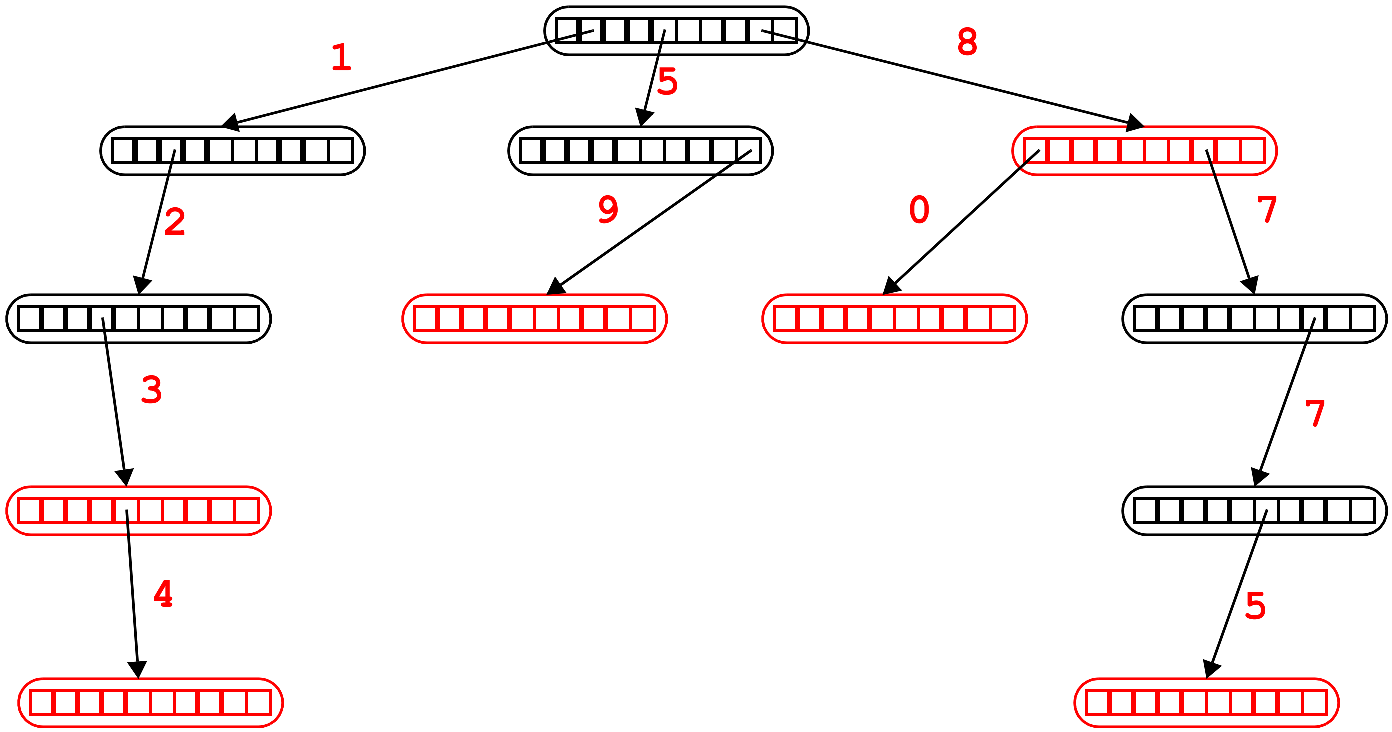
- Consider the following 6 decimal integer keys with values as shown

```
8
1234
59
123
8775
80
```

- We will insert them in that order into an initially empty 10-way trie
- (In this example, the 0*th* digit is the leftmost digit; and nodes with **`end`** bit true are shown in red)

Multiway trie with end bits: example

8
1234
59
123
8775
80



Multiway trie properties

- The structure of a multiway trie depends only on the keys in it, not on the order in which they were inserted
- Multiway tries have a strong key ordering property: At a node X, all keys in X's leftmost subtree are smaller than keys in X's next-to-leftmost subtree, etc. (according to lexicographic ordering)
  - So, a preorder traversal of a multiway trie visits keys in sorted order
  - Also, after following a sequence of digits to get to a node X, all keys in the trie that have that sequence as prefix are in the subtree rooted at X
- Suppose there are *r* bits per digit (so radix $R = 2^r$), and keys contain at most B bits. Then:
  - The worst case height of a R-ary trie containing N keys is B/r
  - Compare: worst case height in a regular BST containing N keys is N, which with B-bit keys can be as much as 2^B
- When N is large and B is comparable to $\log_R N$, DST's give worst-case time cost guarantees better than balanced BST's, and are much easier to implement
- However there is a space cost disadvantage of multiway trees: Each node must store R child pointers, and for a typical tree many of these will be null
- This problem can be addressed with *ternary tries*

Why ternary tries?

- In a multiway trie, each node has number of child pointers equal to the radix R
- This can waste a lot of space
  - consider Java Strings as keys, with each 16-bit char as a digit: radix R = 65536
  - even ASCII strings with 7-bit characters will have radix R = 128
  - ... unless there are very many strings stored in the R-ary trie and the strings are very short, almost all of these child pointers will be null
- Ternary tries avoid this space cost
- (The tradeoff is ternary tries lose the nice worst-case guarantees of multiway tries... though their average case performance is still very good)

Ternary tries

- In a ternary trie, each node contains
  - a key `digit`, for radix search comparison
  - 3 child pointers `left`, `middle`, and `right`, corresponding to keys whose digit being considered is less than, greater than, or equal to the node's `digit`
  - an `end` bit, to indicate that this node contains a key
- The basic search algorithm to find `key` in a ternary trie with end bits:

```
1. Set currentNode = root, i = 0.
2. If currentNode is null, or i >= # of digits in key,
      return "not found".
3. Set d = the ith digit in key.
4. If currentNode.end is true, and i == # of digits in key,
      and currentNode.digit == d, return "found",
5. If d<currentNode.digit, set currentNode =currentNode.left;
   else if d>currentNode.digit, set currentNode =currentNode.right;
   else set i = i + 1, and currentNode = currentNode.middle
6.  Go to 2
```

- The insert algorithm in ternary tries is not quite as simple as for multiway tries...

Ternary trie: example

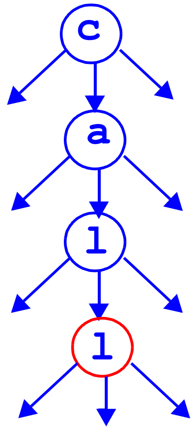
- Consider the following 9 strings

```
call
me
how
mind
not
no
money
milk
note
```

- We will insert them in that order into an initially empty ternary trie

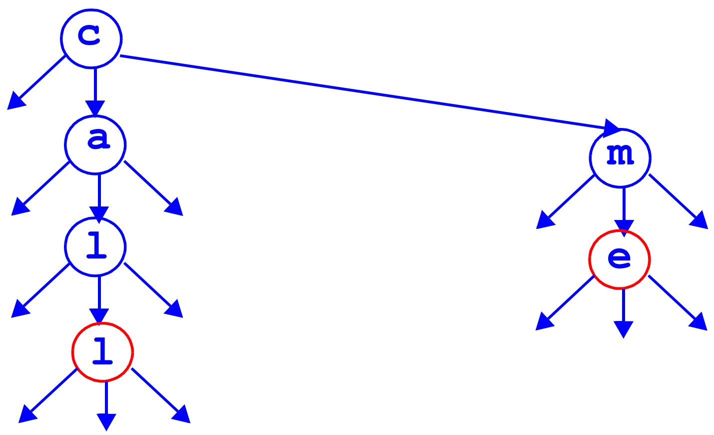
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



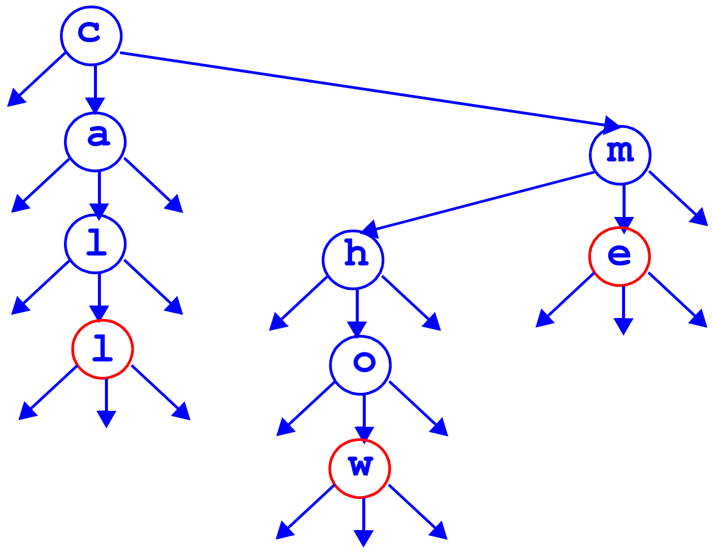
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



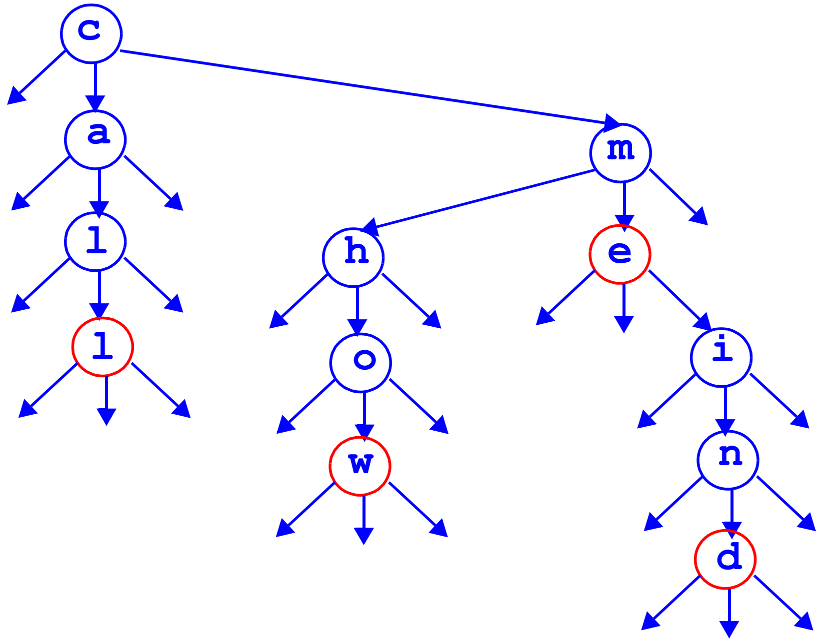
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



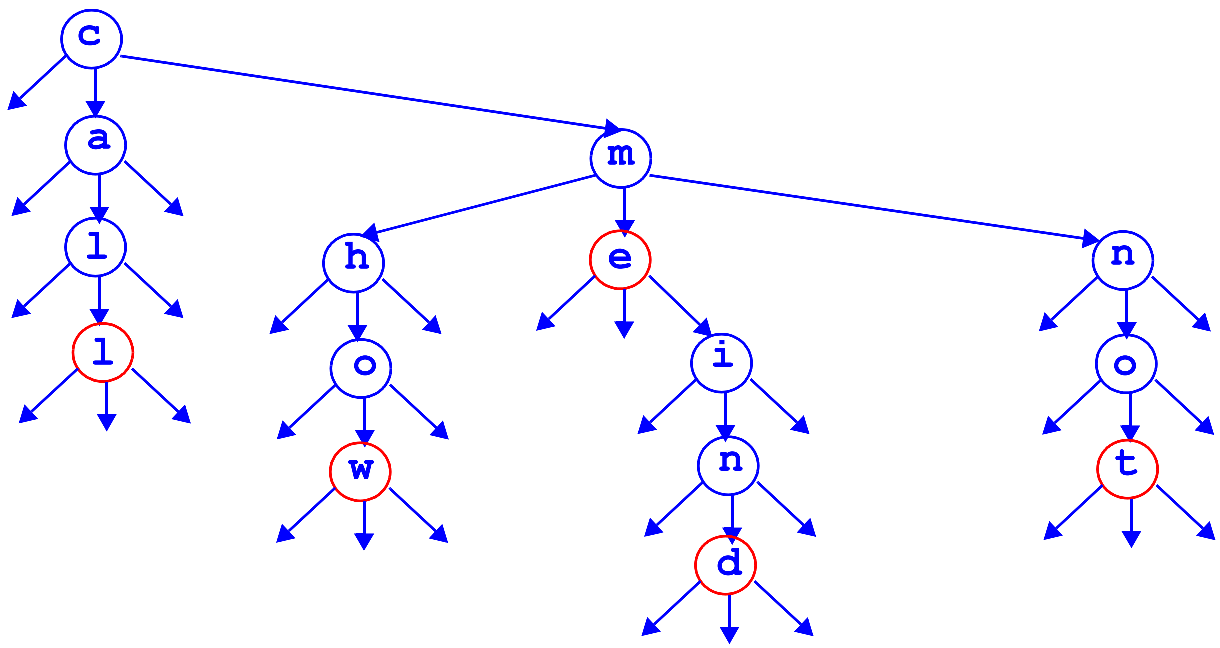
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



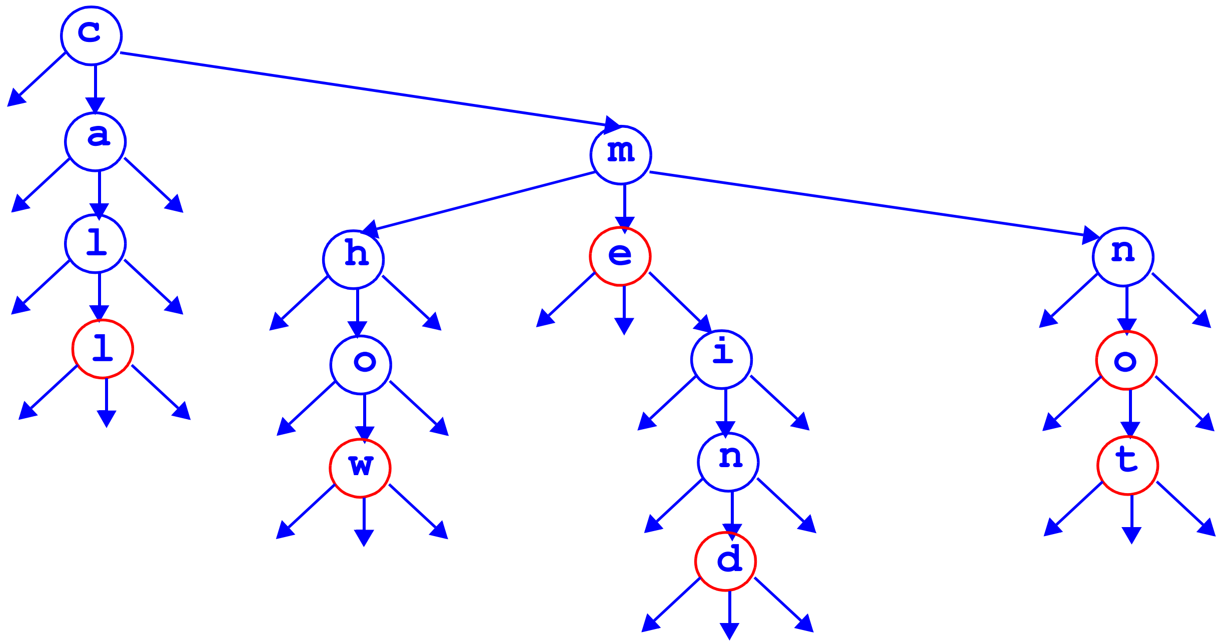
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



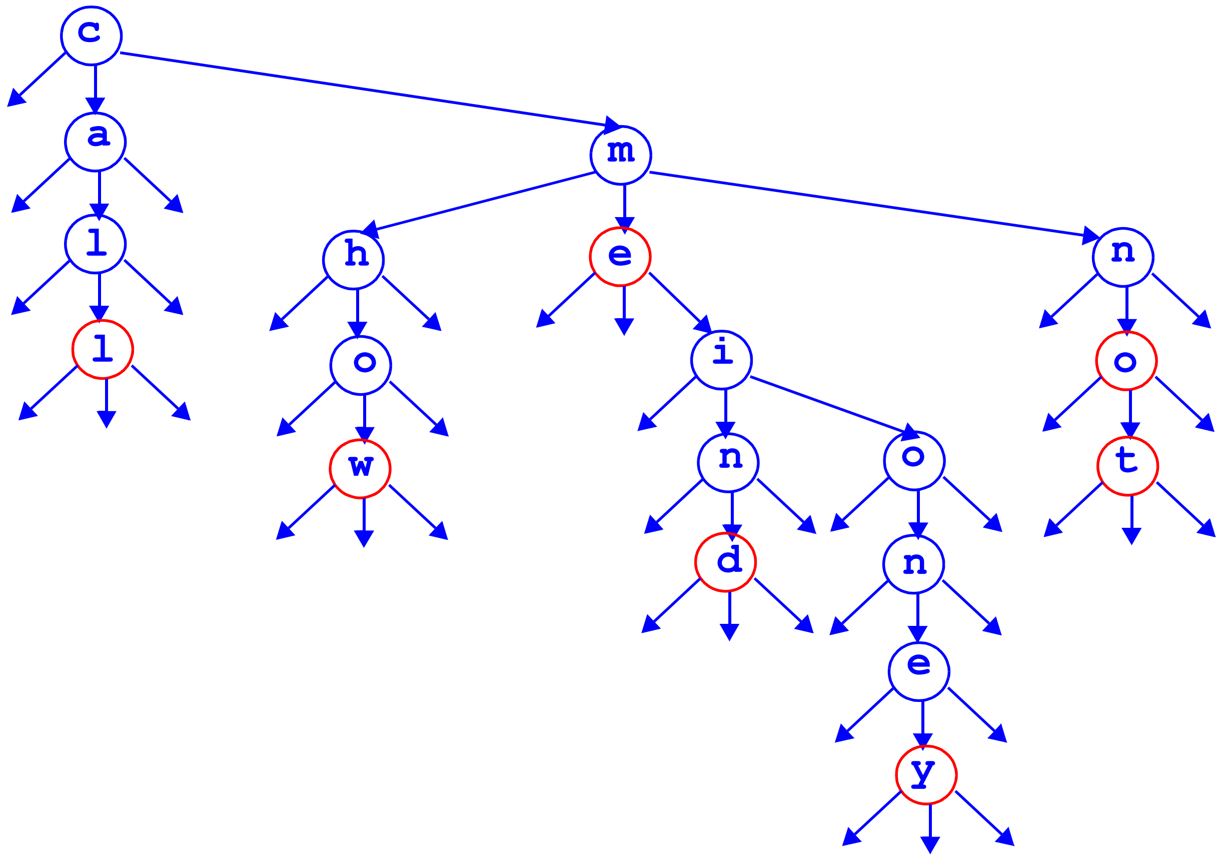
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



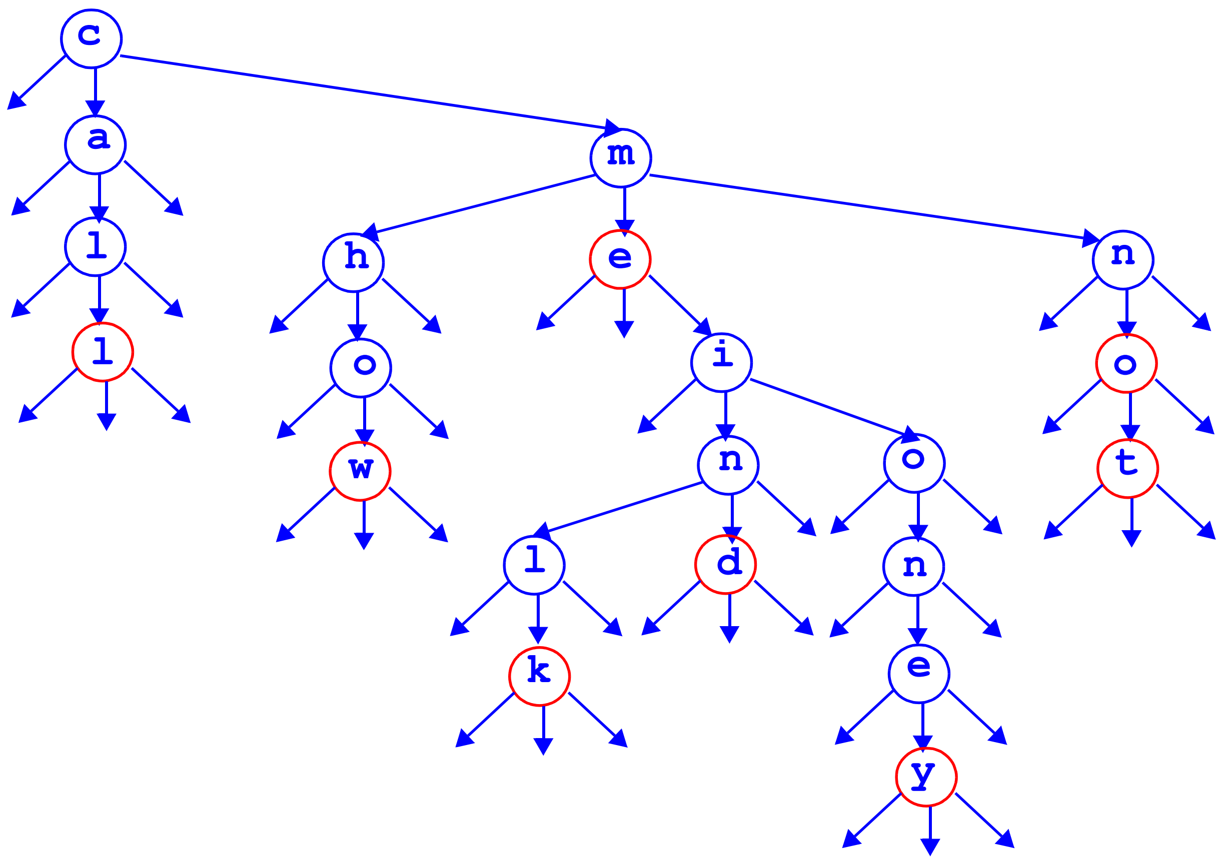
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



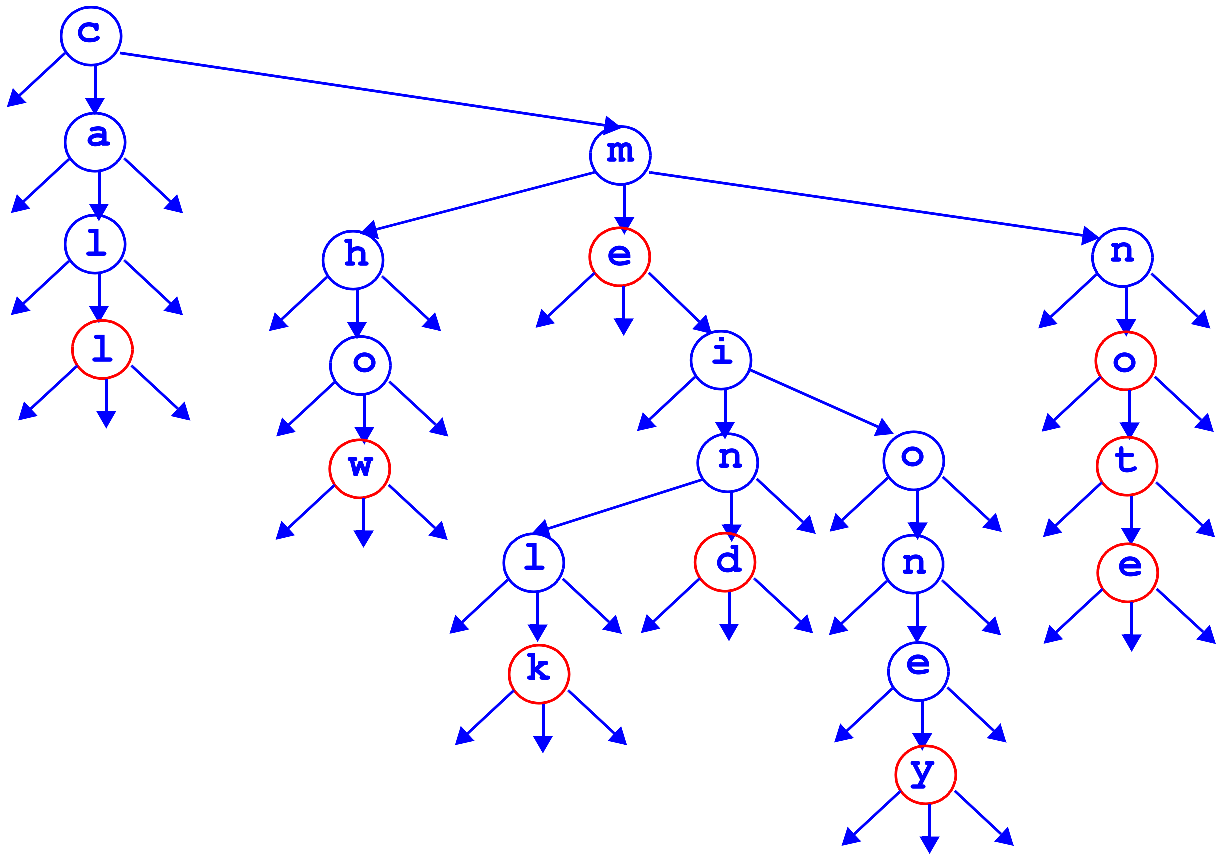
Ternary trie: example

call
me
how
mind
not
no
money
milk
note



Ternary trie: example

```
call
me
how
mind
not
no
money
milk
note
```



Ternary trie properties

- The structure of a ternary trie depends on the order in which keys were inserted
- Ternary tries have a strong key ordering property: At a node X, all keys in X's left subtree are smaller than keys in X's middle subtree, which are smaller than keys in X's right subtree according to lexicographic ordering
  - So, a preorder traversal of a ternary trie visits keys in lexicographic sorted order
  - Also, after following a sequence of digits to get to a node X, all keys in the trie that have that sequence as prefix are in X itself or the subtree rooted at the middle child of X (not the subtrees rooted at the left or right child of X!)
- Ternary trees are space efficient: to hold keys containing a total of D digits requires at most D nodes and 3*D pointers
- Ternary trees have poor worst-case time costs; they can be as bad as linked lists (if keys share no prefixes and are inserted in lexicographic order, for example)
- However in practice with typical keys and key insertion sequences, their performance is quite good, O(logN) average case

Next time

- Hashing
- Hash table and hash function design
- Hash functions for integers and strings
- Some collision resolution strategies: linear probing, double hashing, random hashing
- Hash table cost functions

Reading: Weiss Ch. 5