Tries & Huffman Codes

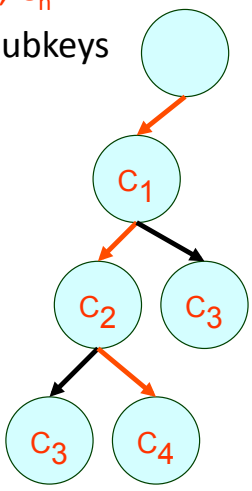
CIS2520 - Data Structures

Fall 2011, LAB 9

Tao Xu

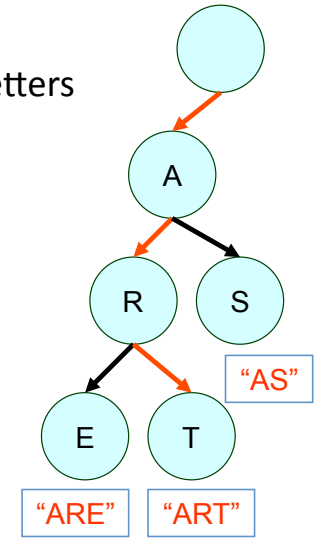
Trie - Indexed Search Tree

- A trie (retrieval) is a special case of tree
- Applicable when
  - Key C can be decomposed into subkeys C_1,…, C_n
  - Redundancy (same prefix) exists between subkeys
- Approach
  - Store subkey at each node
  - Path through trie yields full key



Tries

- Useful for string matching
  - String decomposed into sequence of letters
  - E.g.,
    - "ART" $\Longrightarrow$ "A" "R" "T"
- Can be very fast
  - Less overhead than hashing
- May reduce memory
  - Exploiting redundancy



Types of Tries

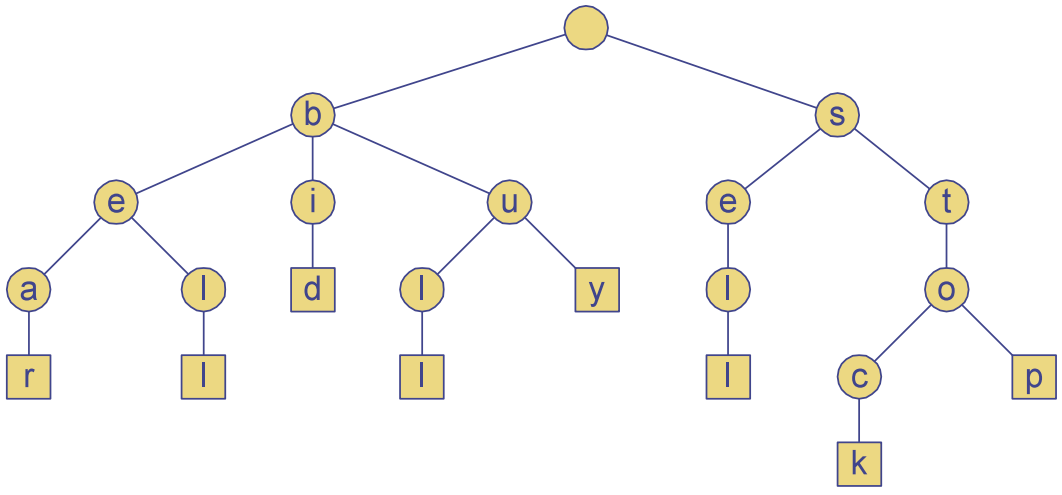
- Standard
  - Single character per node
- Compressed
  - Eliminating chains of nodes
- Compact
  - Stores indices into original string(s)

Standard Tries

- A standard trie for a set of strings $S=\{S_1,S_2,\ldots\}$
  - Each node (except root) is labeled with a character
  - Children of every node are ordered (alphabetically)
  - Paths from root to leaves yield all strings in S

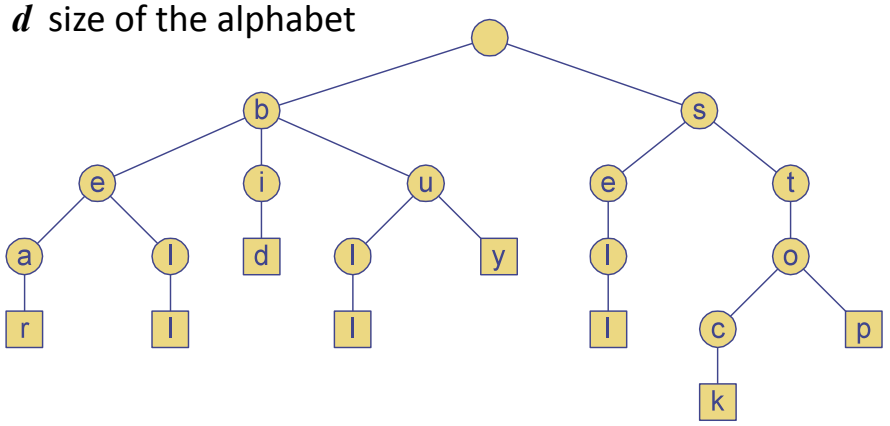
Standard Trie Example

- For strings
  - { bear, bell, bid, bull, buy, sell, stock, stop }



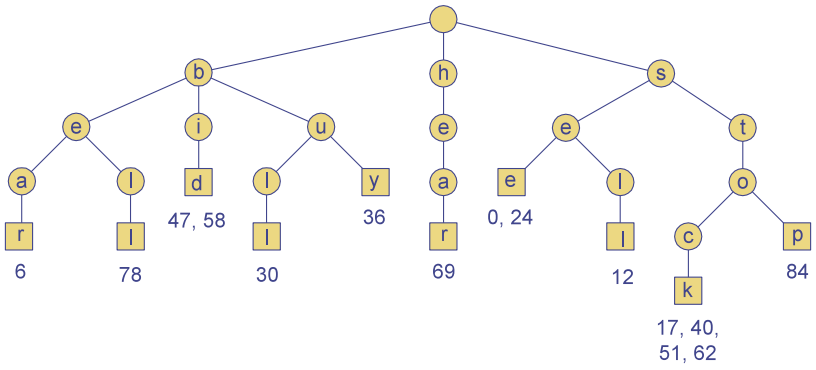
Standard Trie (2)

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  - n total length of all strings in S
  - m size of the string parameter of the operation
  - d size of the alphabet



Word Matching Trie

- Insert words into trie
- Each leaf stores occurrences of word in the text

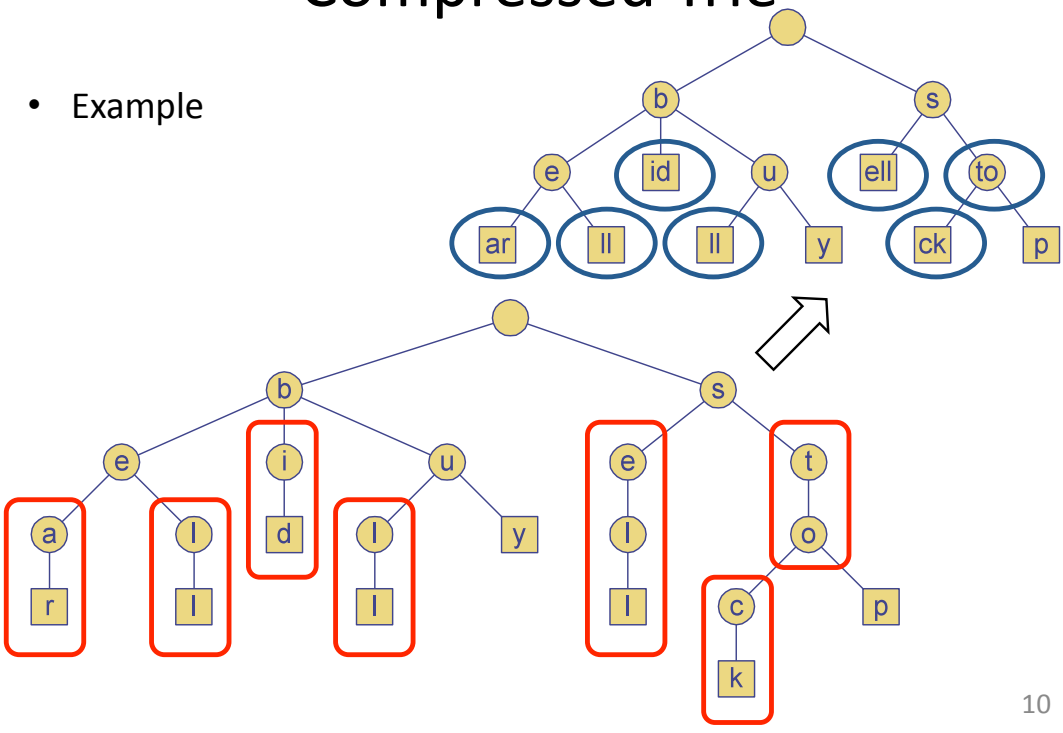


Compressed Trie

- Observation
  - Internal node v of T is redundant if v has one child and is not the root
- Approach
  - A chain of redundant nodes can be compressed
    - Replace chain with single node
    - Include concatenation of labels from chain
- Result
  - Internal nodes have at least 2 children
  - Some nodes have multiple characters

Compressed Trie

- Example

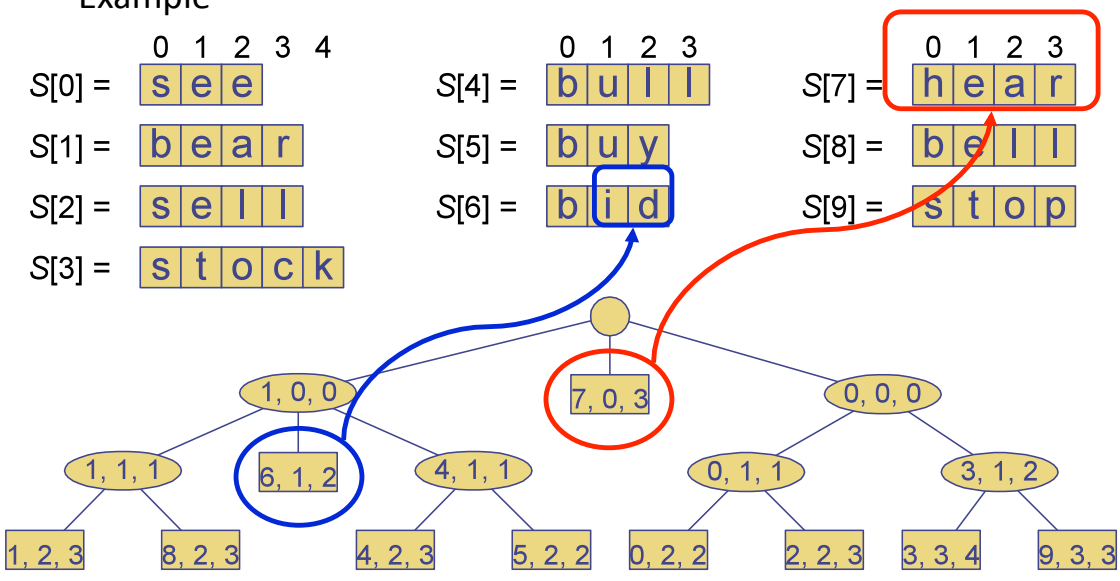


Compact Tries

- Compact representation of a compressed trie
- Approach
  - For an array of strings S = S[0], … S[s-1]
  - Store ranges of indices at each node
    - Instead of substring
  - Represent as a triplet of integers (i, j, k)
    - Such that X = s[i][j..k]
  - Example: S[0] = “abcd”, (0,1,2) = “bc”
- Properties
  - Uses O(s) space, where s = # of strings in the array
  - Serves as an auxiliary index structure

Compact Representation

- Example



Tries and its Applications

- Web search engine indexing
- Spelling check
- Computational biology
- ...

Huffman Encoding

- Compression
  - Typically, in files and messages,
    - Each character requires 1 byte or 8 bits
    - Already wasting 1 bit for most purposes!
- Question
  - What's the minimum number of bits that can be used to store an arbitrary piece of text?
- Idea
  - Find the frequency of occurrence of each character
  - Encode Frequent characters -- short bit strings
  - Rare characters -- long bit strings

Encoding – A Simple Example

- A message of 5 symbols:
  [▸♣♣♠☻▸♣☼▸☻]

- Fix length code (e.g, ASCII)
  5 symbols → at least 3 bits each
  Total Length = 10*3=30 bits

▶	000
♣	001
☻	010
♠	011
☼	100

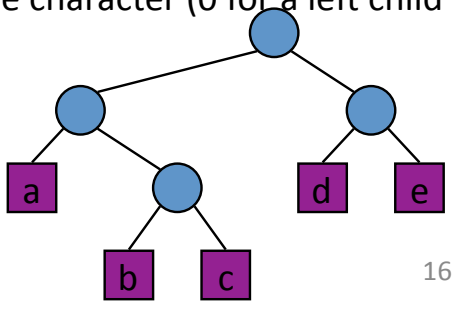
- Huffman code
  Length varies depending on frequency
  Total Length = 3*2 +3*2+2*2+3+3=24bits

Symbol	Freq.	Code
▶	3	00
♣	3	01
☻	2	10
♠	1	110
☼	1	111

Encoding Trie

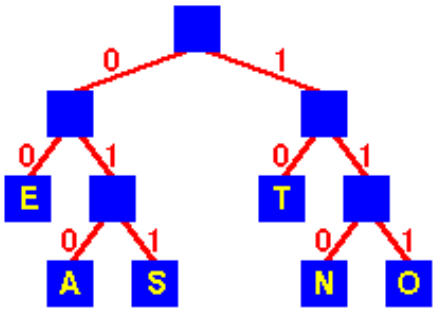
- A code is a mapping each character of an alphabet to a binary code-word
- A prefix code is a binary code such that no code-word is the prefix of another code-word
- An encoding trie represents a prefix code
  - Each leaf stores a character
  - The code word of a character is given by the path from the root to the leaf storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



Huffman Encoding

- Encoding
  - Use a tree
  - Encode by following tree to leaf
  - E.g.,
    - E is 00
    - S is 011
  - Frequent characters
    E, T 2 bit encodings
  - Others
    A, S, N, O 3 bit encodings



Huffman Encoding

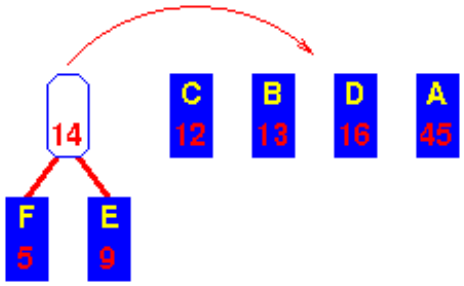
- Greedy Approach
  - Sort characters by frequency
  - Form two lowest weight nodes into a sub-tree
    - Sub-tree weight = sum of weights of nodes
  - Move new tree to correct place
- Bottom-up (optimal)
  - Top-down (Shannon coding, suboptimal)
- Suboptimal to encode block data (e.g., ‘cat’)

Huffman Encoding - Operation

Initial sequence
Sorted by frequency

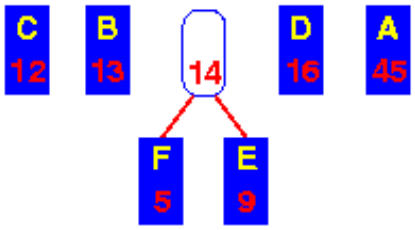
Combine lowest two into sub-tree

Move it to correct place



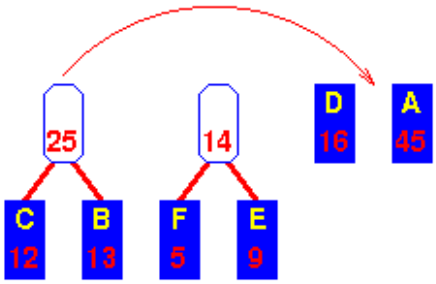
Huffman Encoding - Operation

After shifting sub-tree to its correct place ...



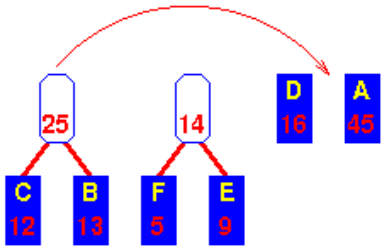
Combine next lowest pair

Move sub-tree to correct place



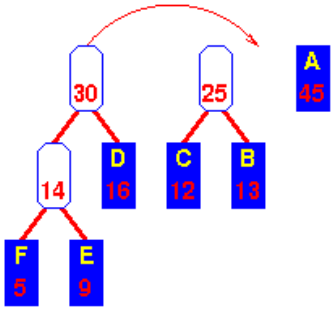
Huffman Encoding - Operation

Move the new tree
to the correct place ...

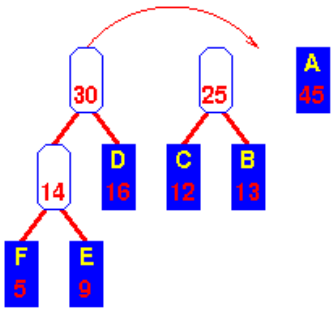


Now the lowest two are the
“14” sub-tree and D

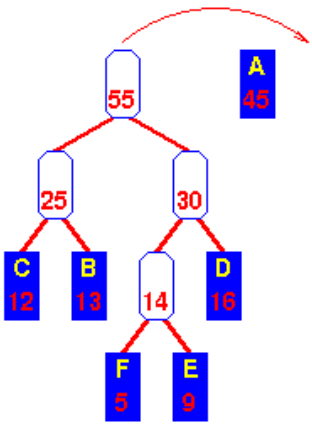
Combine and move to
correct place



Huffman Encoding - Operation



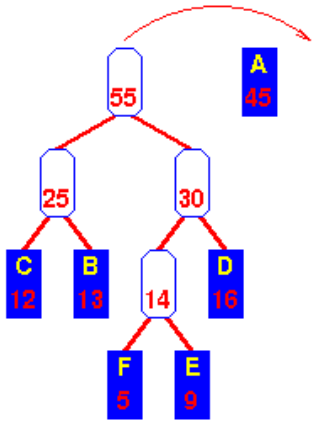
Move the new tree
to the correct place ...



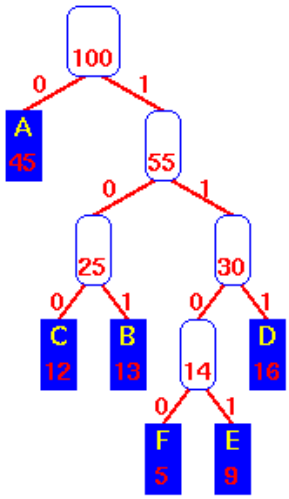
Now the lowest two are the
the “25” and “30” trees

Combine and move to
correct place

Huffman Encoding - Operation



Combine last two trees



Huffman's Algorithm

- Given a string X, Huffman's algorithm construct a prefix code the minimizes the size of the encoding of X
- A heap-based priority queue is used as an auxiliary structure

```
function HuffmanEncoding (X)
    Input: string X of size n
    Output: optimal encoding trie for X
        C = distinctCharacters(X)
        computeFrequencies(C, X)
        Q = new empty heap
        for all c in C
            T = (new single-node tree storing c)
            Q.insert(getFrequency(c), T)
        while Q.size() > 1
            f1 = Q.minKey()
            T1 = Q.removeMin()
            f2 = Q.minKey()
            T2 = Q.removeMin()
            T = join(T1, T2)
            Q.insert(f1 + f2, T);
        return Q.removeMin()
```

Huffman Encoding - Time Complexity

- Sort keys — $O(n \log n)$
- Repeat n times
  - Form new sub-tree — $O(1)$
  - Move sub-tree (binary search) — $O(\log n)$
  - Total — $O(n \log n)$
- Overall — $O(n \log n)$

Huffman Encoding - Decoding