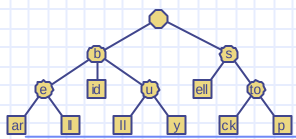
Tries

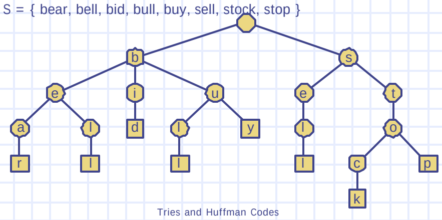


Preprocessing Strings

- A trie (retrieval) is a special kind information access tree
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - A tries supports pattern matching queries in time proportional to the pattern size

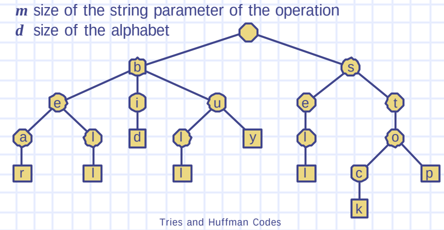
Standard Trie (1)

- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings
  S = { 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

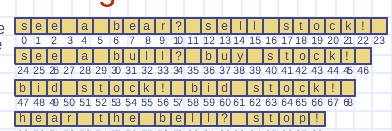


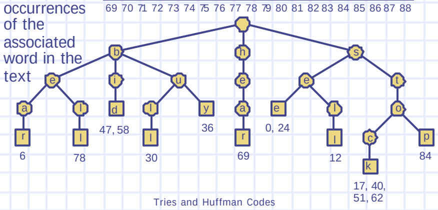
Application:

- use a trie to perform a special type of pattern matching: word matching.
- differ from standard pattern matching since the pattern can not match with an arbitrary substring of the text, but only one of its words.
- suitable for applications where a series of queries is performed on a fixed text.

Word Matching with a Trie

- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text



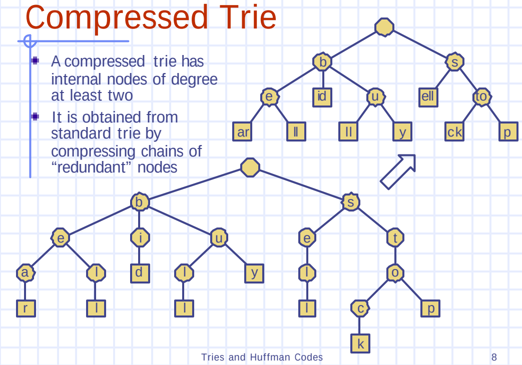


Compressed Tries:

- an internal node v of T is redundant if v has one child and is not the root.
- a chain of redundant nodes can be compressed by replacing the chain with a single node with the concatenation of the labels of nodes in the chain.

Compressed Trie

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of "redundant" nodes



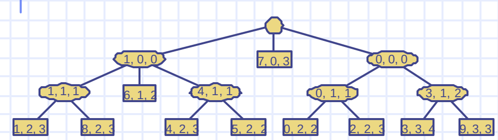
Compact Representation

- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(s)$ space, where s is the number of strings in the array
  - Serves as an auxiliary index structure
  - S is an array of strings S[0], ... S[s-1]
  - Instead of storing a node label X explicitly, we represent it implicitly by a triplet of integers (i, j, k), such that X = s[i][j..k].

	0	1	2	3	4
S[0] =	s	e	e		
S[1] =	b	e	a	r	
S[2] =	s	e	l	l	
S[3] =	s	t	o	c	k
S[4] =	b	u	l	l	
S[5] =	b	u	y		
S[6] =	b	i	d		
S[7] =	h	e	a	r	
S[8] =	b	e	l	l	
S[9] =	s	t	o	p	

Compact Representation

	0	1	2	3	4
S[0] =	s	e	e		
S[1] =	b	e	a	r	
S[2] =	s	e	l	l	
S[3] =	s	t	o	c	k
S[4] =	b	u	l	l	
S[5] =	b	u	y		
S[6] =	b	i	d		
S[7] =	h	e	a	r	
S[8] =	b	e	l	l	
S[9] =	s	t	o	p	



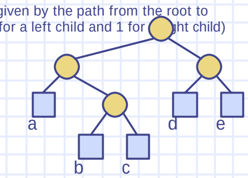
Huffman Encoding

- 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 smallest 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
  - Rarer characters longer bit strings

Encoding Trie (1)

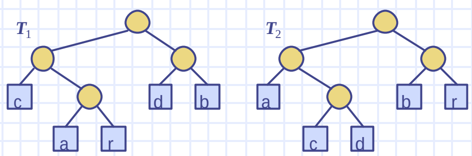
- A code is a mapping of each character of an alphabet to a binary code-word
- A prefix code is a binary code such that no code-word is the prefix of another code-word
- An encoding trie represents a prefix code
  - Each leaf stores a character
  - The code word of a character is given by the path from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

00	010	011	10	11
a	b	c	d	e



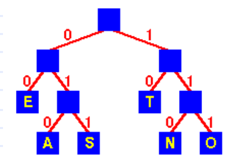
Encoding Trie (2)

- Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X
- Example
  - X = abracadabra
  - T_1 encodes X into 29 bits
  - T_2 encodes X into 24 bits



Huffman Encoding

- Encoding
  - Use a tree
  - Encode by following tree to leaf
  - eg
    - 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

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

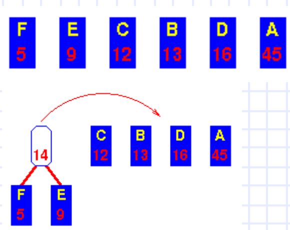
```
function HuffmanEncoding(X)
  Input string X of size n
  Output optimal encoding trie for X
  C = distinctCharacters(X);
  computeFrequencies(C, X);
  Q = new empty heap ;
  for (all c ∈ C) {
     T = (new single-node tree storing c);
     Q.insert(getFrequency(c), T); }
  while (Q.size() > 1){
     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 - Operation

Initial sequence Sorted by frequency

Combine lowest two into sub-tree

Move it to correct place

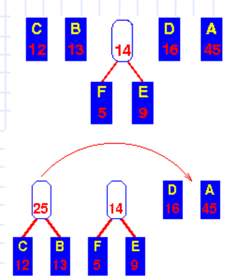


Huffman Encoding - Operation

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

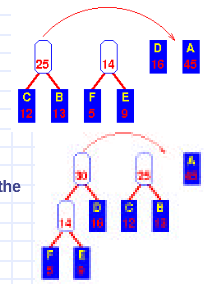
Combine next lowest pair

Move sub-tree to correct place



Huffman Encoding - Operation

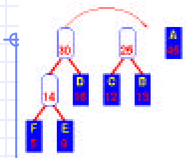
Move the new tree to the correct place ...



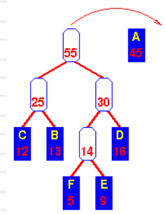
Now the lowest two are the “14” sub-tree and D

Combine and move to correct place

Huffman Encoding - Operation



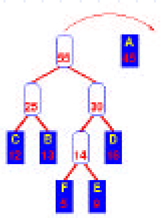
Move the new tree to the correct place ...



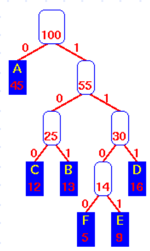
Now the lowest two are the the “25” and “30” trees

Combine and move to correct place

Huffman Encoding - Operation



Combine last two trees

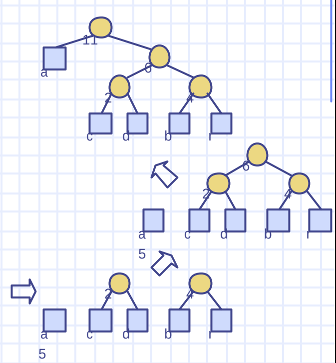


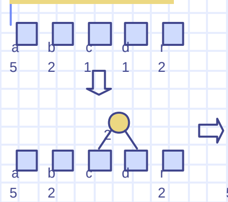
Example

X = abracadabra

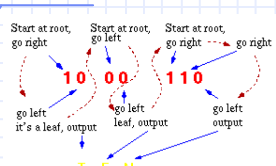
Frequencies

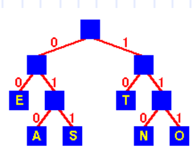
a	b	c	d	r
5	2	1	1	2





Huffman Encoding - Decoding





Huffman Encoding - Time Complexity

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