CSE221 Data Structures
Lecture 24
String Algorithms II

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

December 8, 2021

Introduction

- Final exam is on Wednesday 15 December, 20:00–22:00.
- Similar format as midterm.
- Emphasis will be on the second part of the semester, i.e. Lectures 11–24.
- Assignment 4 due on Friday. I posted the last 3 instances yesterday.
- This is a second lecture on algorithms for *strings*.
- Reference for this lecture: Textbook Chapter 13.

Huffman Coding

- Consider the string:

$$X = \text{"a fast runner need never be afraid of the dark"}$$

- The number of occurrences of each character is given in the table below.

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

- We want to find an encoding of each character such that the encoding of the whole string X is as small as possible.
- This task is called *text compression*. We present one approach to it, called *Huffman coding*.

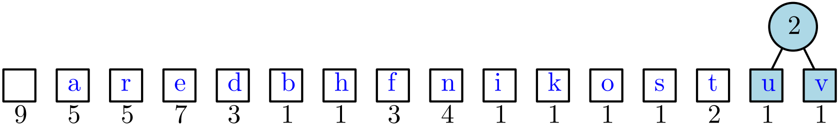
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

	a	r	e	d	b	h	f	n	i	k	o	s	t	u	v
9	5	5	7	3	1	1	3	4	1	1	1	1	2	1	1

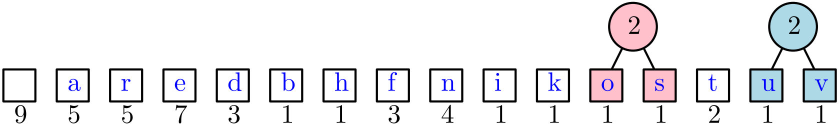
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



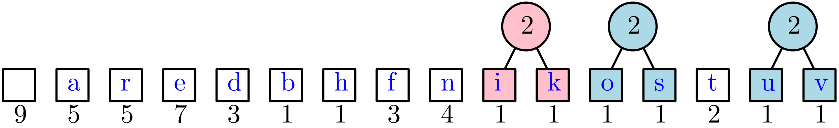
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



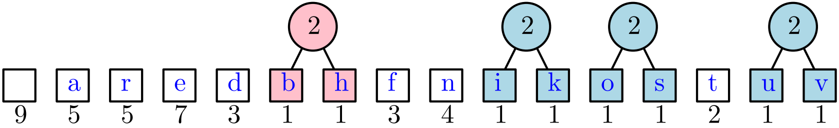
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



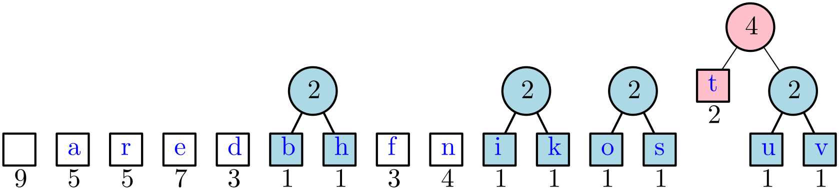
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



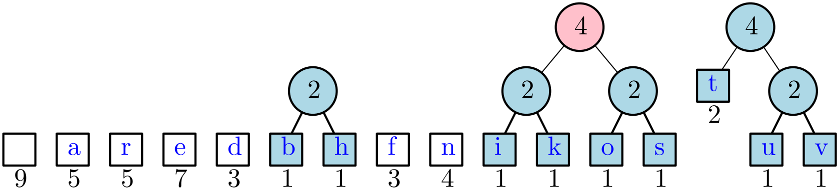
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



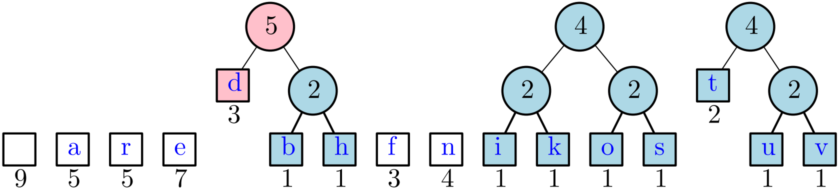
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



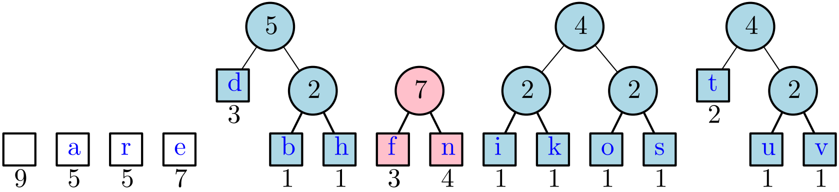
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



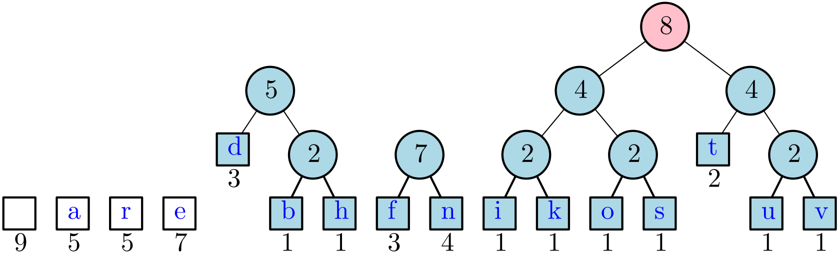
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



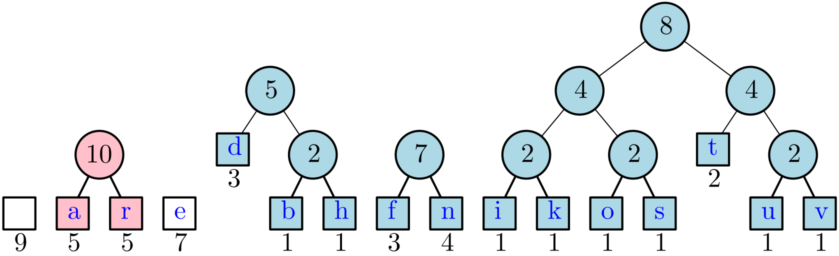
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



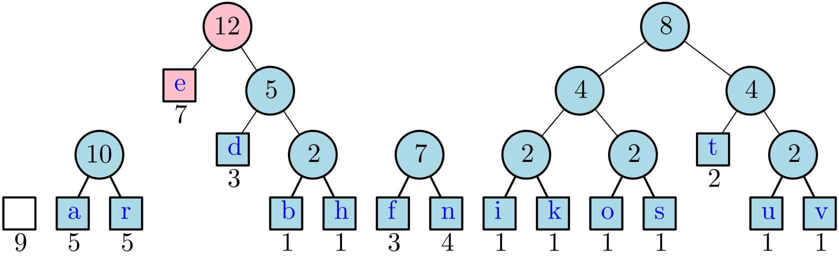
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



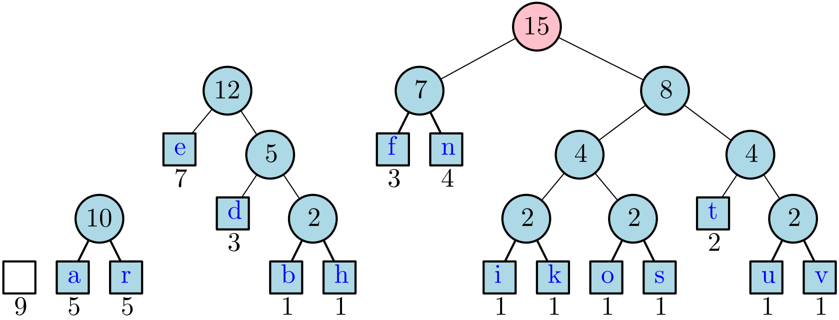
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



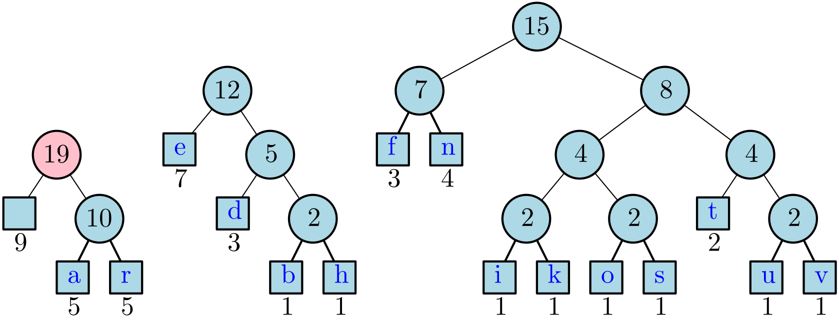
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



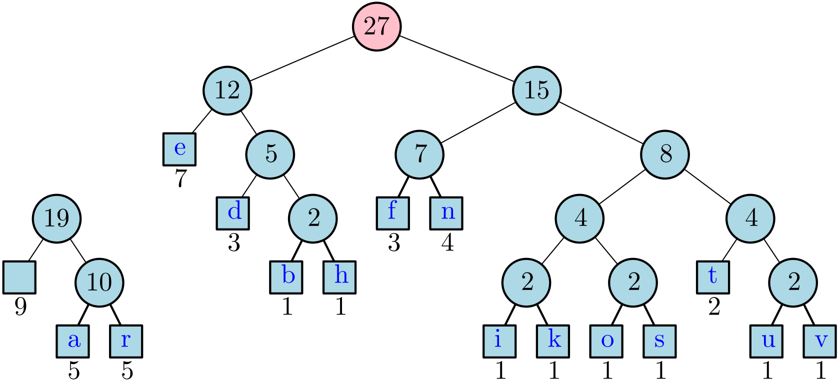
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



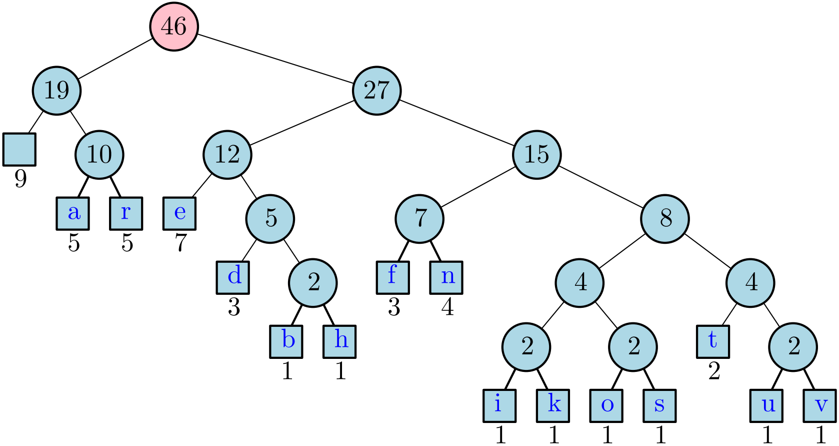
Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

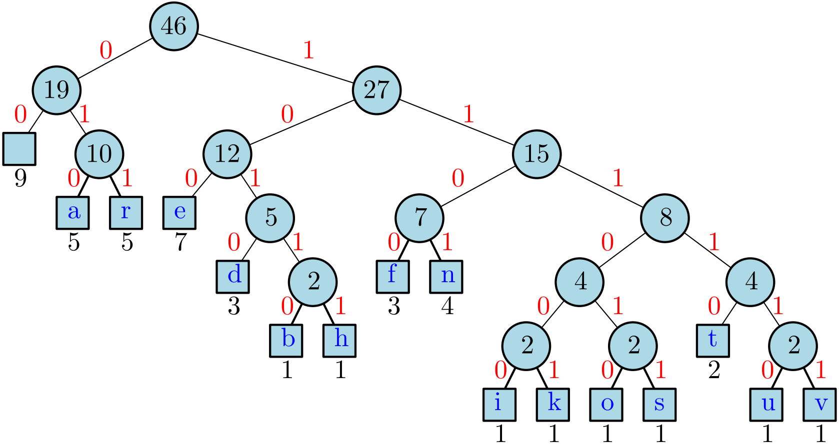


Huffman Coding

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Huffman Coding



- The encoding of 'a' is 010 and the encoding of 'f' is 1100.

Huffman-Coding Algorithm

Pseudocode

```
procedure HUFFMAN(X)
    Compute the frequency f(c) of each character c of X.
    Q ← empty priority queue.
    for each character c in X do
        Create a single-node binary tree T storing c.
        Insert T into Q with key f(c).
    while Q.size() > 1 do
        f1 ← Q.min()
        T1 ← Q.removeMin()
        f2 ← Q.min()
        T2 ← Q.removeMin()
        T ← new binary tree with left subtree T1 and right subtree T2.
        Insert T into Q with key f1 + f2.
    return return tree Q.removeMin()
```

Huffman-Coding

Proposition

Let d be the number of distinct characters in X. Then the algorithm above runs in time $O(n + d \log d)$.

- Huffman coding allows us to encode a string into a binary sequence in an unambiguous way, thanks to the property below.

Proposition

Huffman coding is a prefix code: no codeword is a prefix of another codeword.

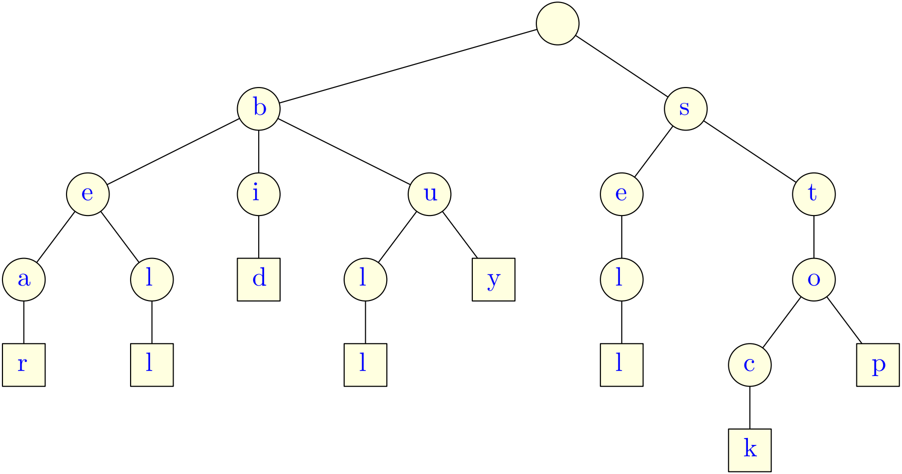
Example

0101100 encodes “af” without ambiguity.

Greedy Algorithms

- Huffman-coding is an example of a *greedy algorithm*.
- It means that at each step of the algorithm, we make the choice that achieves the best cost improvement *for this step*.
- In other words, the algorithm *only looks one step ahead*.
- More examples of greedy algorithms are given in CSE331: Introduction to Algorithms.
- Greedy algorithms often do not give an optimal result, but they may provide a reasonable approximation.
- It can be shown that Huffman coding is optimal in the sense that it minimizes the length of the encoding (not covered in this course or in the textbook).

Tries



- Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

Tries

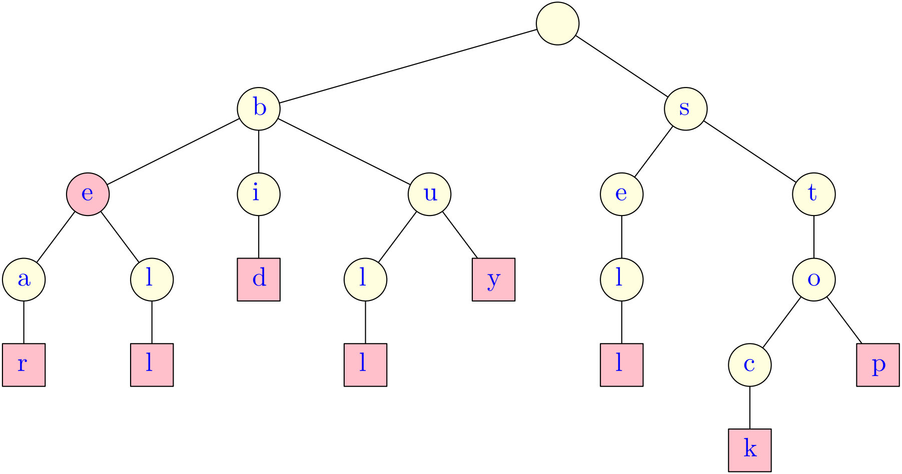
Definition

Let S be a set of s strings from alphabet Σ such that no string in S is a prefix of another string. A *standard trie* for S is an ordered tree T with the following properties:

- Each node of T, except the root, is labeled with a character of Σ.
- The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ.
- T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf node v of T yields the string of S associated with v.

- What if some strings in S are prefixes of other strings?
- We can mark some nodes as terminal. See next slide.

Tries



- Standard trie for the strings {be, bear, bell, bid, bull, buy, sell, stock, stop}.

Tries

Proposition

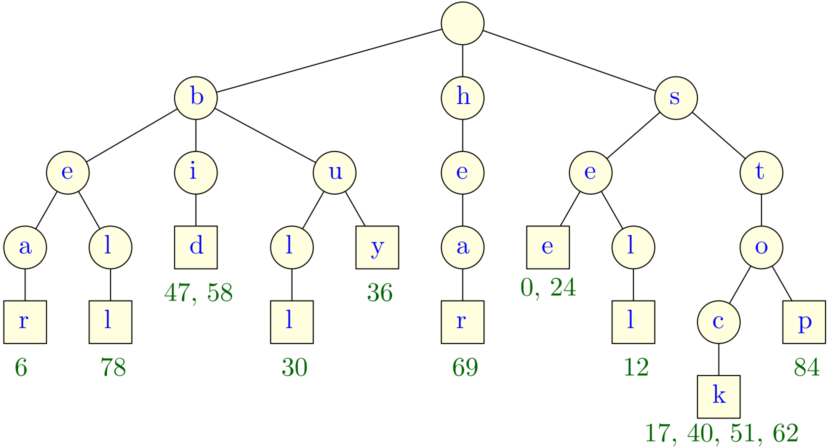
A standard trie storing a collection S of s strings of total length n from an alphabet of size d has the following properties:

- *Every internal node of T has at most d children*
- *T has s leaves*
- *The height of T is equal to the length of the longest string in S*
- *The number of nodes of T is $O(n)$*

Tries

- A word w can be inserted in time $O(dm)$ where $d = |\Sigma|$ and $m = |w|$ is the number of characters in the word.
- So constructing a trie for a set S can be done in $O(dn)$ time where n is the total number of characters.
- A trie allows us to perform *word matching* queries: finding a word w in a set S of strings. It can be done in $O(dm)$ time.
- It also allows to do *prefix matching*: Finding all the strings in S that have w as a prefix.
- Next slide shows how to augment the tree with the positions of each word so that, after finding a word in the trie, we can find its position in the text in constant time.

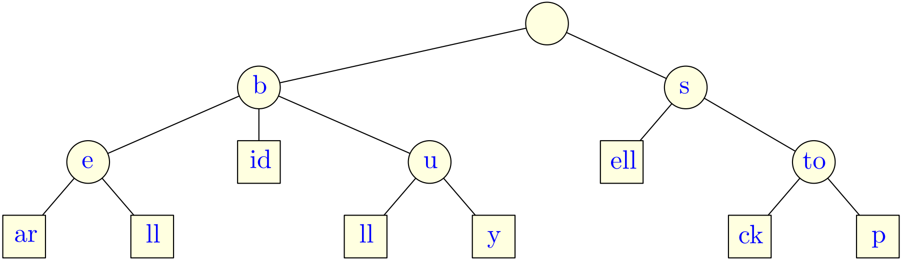
Tries



see a bear? sell stock! see a bull? buy stock!
0 10 20 30 40

bid stock! bid stock! hear the bell? stop
50 60 70 80

Compressed Trie



compressed version of the trie from Slide 25

- A *compressed trie* is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges.
- It allows us to save space if the strings stored in the node are represented by their indices in the set S of strings that it indexes. (See textbook.) Then the compressed trie takes $O(s)$ space, where s is the number of strings in S.

Conclusion

- This was the last lecture of CSE221.
- We studied data structures such as arrays, linked lists, stacks, queues, heaps, hash tables, graphs, tries ...
- We implemented some of them in C++.
- We studied algorithms design approaches such as divide and conquer, dynamic programming, the greedy approach, backtracking.
- We studied algorithm analysis and made some proofs of correctness.
- To study further in this direction, you can take CSE331: Introduction to Algorithm.