Text Processing Algo

算法的本质：
在某些特定的 assumptions 下，我们可以得到比 brute-force 更快的表现。

String

A ***string*** is a sequence of characters.

An ***alphabet*** Σ is the set of possible characters in strings.

Notation:

- $length(P)$ … #characters in P
- λ … empty string ($length(\lambda) = 0$)
- Σ^m … set of all strings of length m over alphabet Σ
- Σ^* … set of all strings over alphabet Σ

Substring of P: any string Q such that P = vQω, for some $\nu, \omega \in \Sigma^*$

- ***prefix*** of P: any string Q such that P = Qω, for some $\omega \in \Sigma^*$
- ***suffix*** of P: any string Q such that P = ωQ, for some $\omega \in \Sigma^*$
- Note: substring can be empty string or P

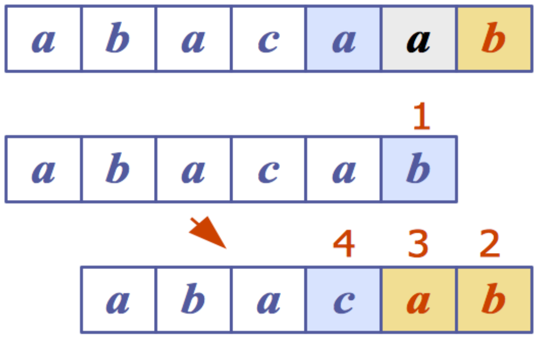
In C a string is an array of chars containing ASCII codes.

- these arrays have an extra element containing a 0
- the extra 0 can also be written '\0' (null character or null-terminator)

- convenient because don't have to track the length of the string

Pattern Matching

Normally, we have pattern checked *backwards.



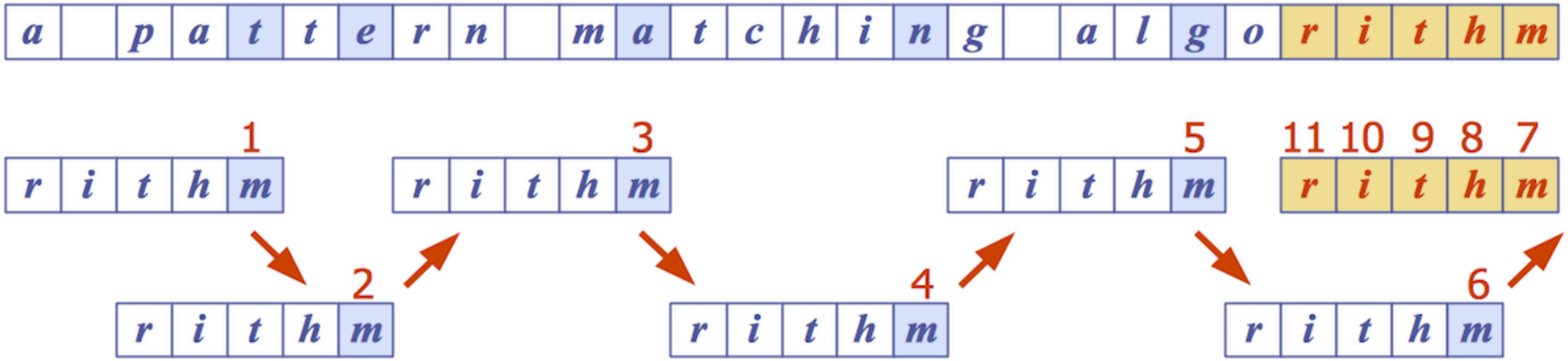
Performance of brute-force algo: $O(m \times n)$

Boyer-Moore Algorithm

The Boyer-Moore pattern matching algorithm is based on two heuristics:

- **Looking-glass heuristic**: Compare `P` with subsequence of `T` moving *backwards*
- **Character-jump heuristic**: When a mismatch occurs at `T[i] = c`
  - **Small jump**: if `P` contains `c`
    - shift `P` so as to align the last occurrence of `c` in `P` with `T[i]`
    - or move forward 1 character, if this shift need a backward move
  - **Big jump**: otherwise shift `P` so as to align `P[0]` with `T[i+1]`

Example:



1. Compare `m` and `t`, `m != t` and `rithm` contains `t`, small jump to make the `t` align with the position `t` in `rithm`
2. Compare `m` and `e`, `m != e` and P not contains `e`, big jump
3. ...

Last-occurrence function

Boyer-Moore algorithm preprocesses pattern P and alphabet Σ to build the ***last-occurrence function*** L. This functions map a character to its last occurred index in the pattern.

L maps Σ to integers such that $L(c)$ is defined as

- the largest index i such that $P[i] = c$, or
- -1 if no such index exists

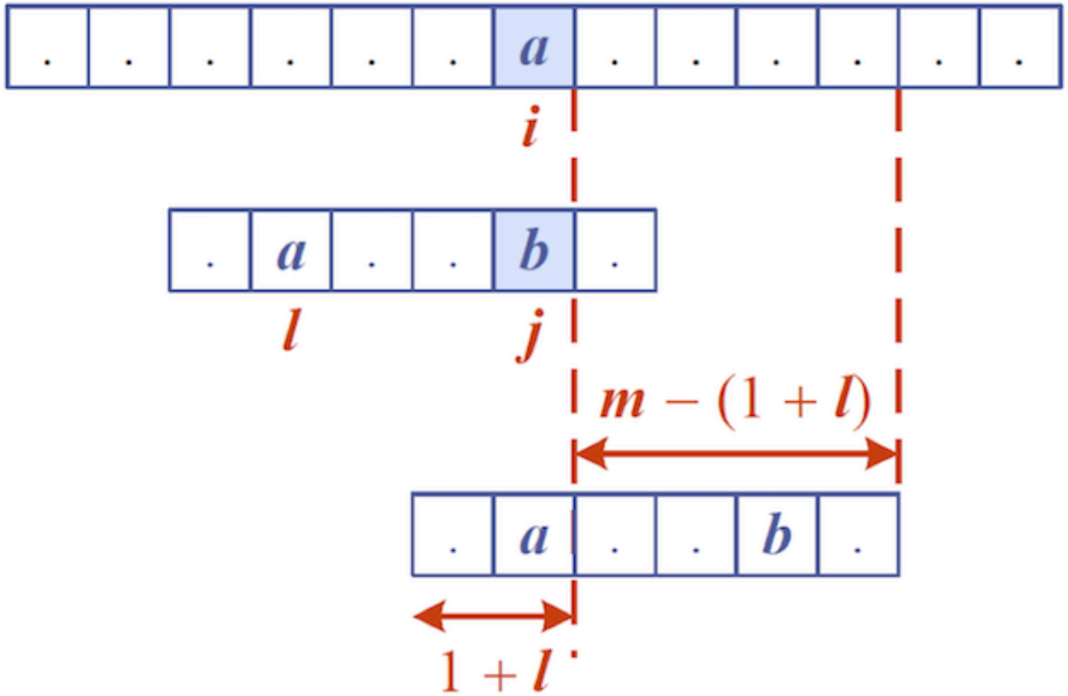
Example: Σ = {**a**, **b**, **c**, **d**}, *P* = **acab**

c	a	b	c	d
L(c)	2	3	1	-1

Suppose we have a mismatch at `T[i]` and `P[j]` (i.e. `T[i]` != `P[j]`)
if `L(T[i]) = m` :

- if `m` != -1, align `P[m]` with `T[i]`
- if `m` == -1, big jump

```
BoyerMooreMatch(T,P,Σ):
|  Input  text T of length n, pattern P of length m, alphabet Σ
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  L=lastOccurenceFunction(P,Σ)
|  i=m-1, j=m-1                     // start at end of pattern
|  repeat
|  |  if T[i]=P[j] then
|  |     if j=0 then
|  |        return i                // match found at i
|  |     else
|  |        i=i-1, j=j-1
|  |     end if
|  |  else                          // character-jump
|  |     i=i+m-min(j,1+L[T[i]])     // choose the smaller distance to jump
|  |     j=m-1
|  |  end if
|  until i≥n
|  return -1                        // no match
```

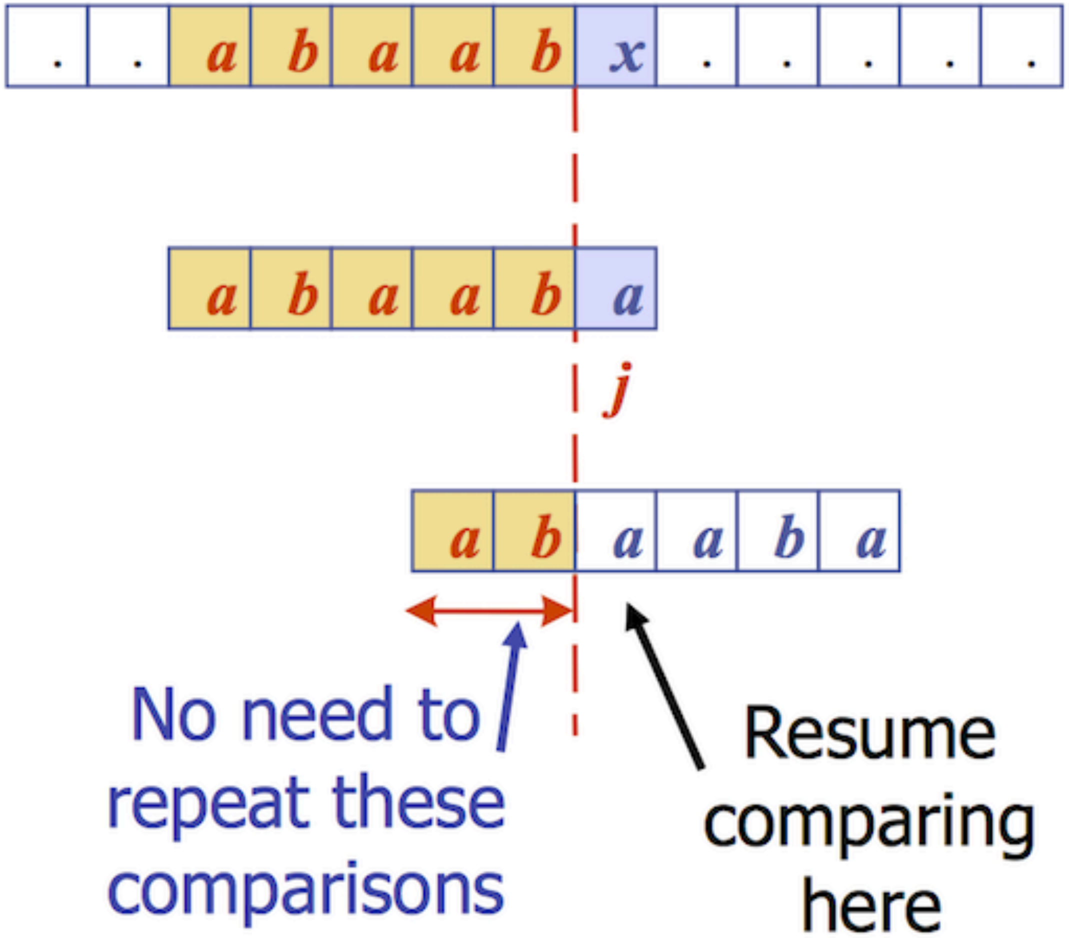


Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm

- compares the pattern to the text left-to-right
- but shifts the pattern more intelligently than the brute-force algorithm

This is useful for text with small alphabet Σ (with repeated characters and suffix).



KMP preprocesses the pattern to find matches of its prefixes with itself

- ***Failure function*** $F(j)$ defined as the size/length of the largest prefix of `P[0..j]` that is also a suffix of `P[1..j]`
- if mismatch occurs at Pj ⇒ advance j to F(j-1)

Difference between Last-occurrence function and Failure function:

1. Last-occurrence function map characters to an **index**, while Failure function map characters to a **length**.

2. Last-occurrence function has `-1`, while Failure function has `0`.

But these 2 functions have same purpose: tell the pattern how to move its pointer to and align the pointer with the `T[i]`.

If we mismatch `T[i]` and `P[j]`,

- if `j > 0`, then we calculate `F(P[j-1]) = m` (`j-1` is the last matched character), align `P[m]` with `T[i]`.
- else move forward on the string one character

Example: P = **abaaba**

j	0	1	2	3	4	5
P_j	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

```
KMPMatch(T,P):
|  Input  text T of length n, pattern P of length m
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  F=failureFunction(P)
|  i=0, j=0                        // start from left
|  while i<n do
|  |  if T[i]=P[j] then
|  |     if j=m-1 then
|  |        return i-j             // match found at i-j
|  |     else
```

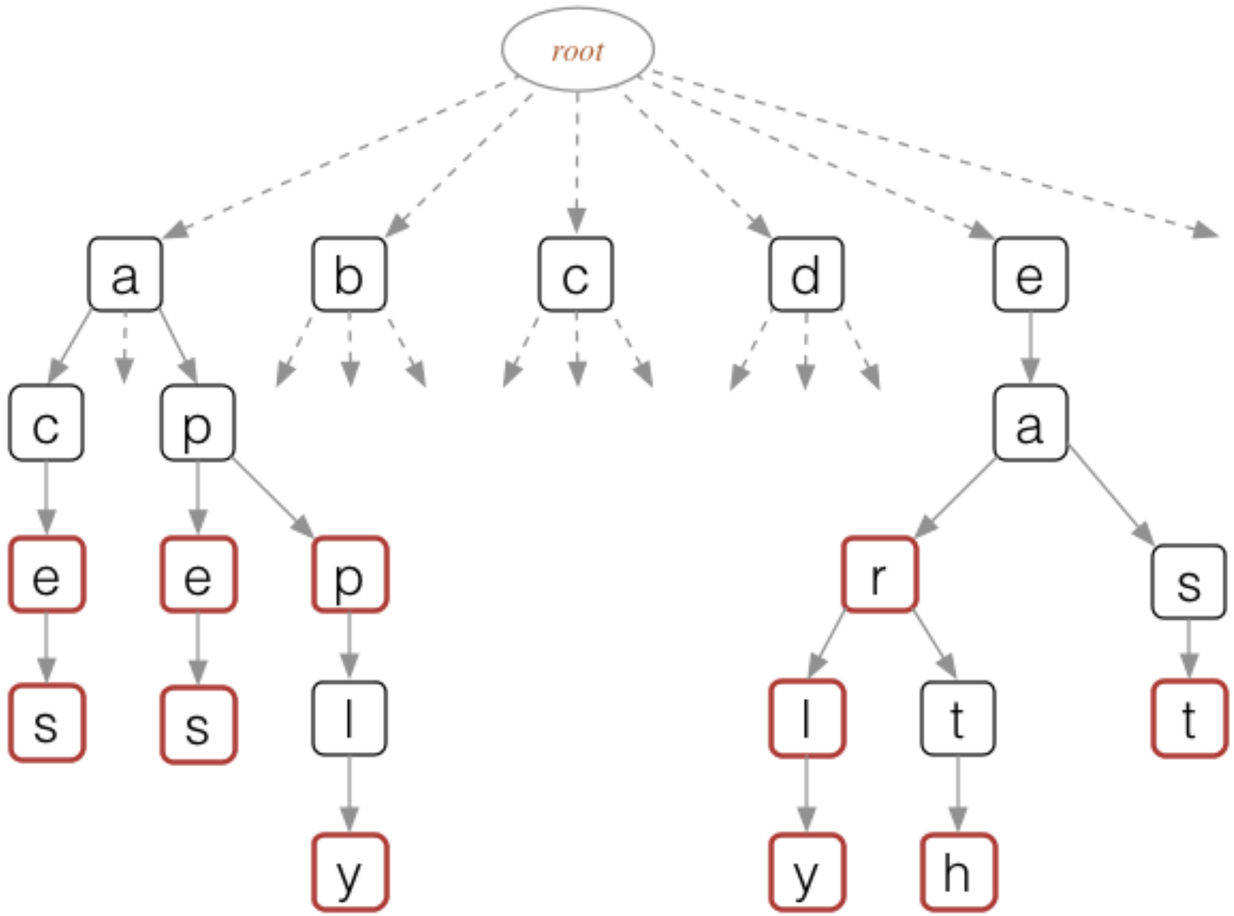
```
|  |       i=i+1, j=j+1
|  |     end if
|  |  else                     // mismatch at P[j]
|  |     if j>0 then
|  |        j=F[j-1]           // resume comparing P at F[j-1]
|  |     else
|  |        i=i+1
|  |     end if
|  |  end if
|  end while
|  return -1                   // no match

// failure function: get the position of last previous match for every character
failureFunction(P):
|  Input  pattern P of length m
|  Output failure function for P
|
|  F[0]=0
|  i=1, j=0
|  while i<m do
|  |  if P[i]=P[j] then   // we have matched j+1 characters, j+1 would be the length of
 already matched characters
|  |     F[i]=j+1
|  |     i=i+1, j=j+1
|  |  else if j>0 then    // use failure function to shift P
|  |     j=F[j-1]
|  |  else
|  |     F[i]=0           // no match
|  |     i=i+1
|  |  end if
|  end while
|  return F
```

Performance: $O(m + n)$

Trie

A trie is a compact data structure for representing a set of strings, which supports pattern matching queries in time proportional to the pattern size.

Tries are trees organised using parts of keys (rather than whole keys).



Each node in a trie

- contains one part of a key
  - Compressed tries: one character
  - Suffix tries: one suffix
- may have up to 26 children
- may be tagged as ***a "finishing" node*** (red nodes in the example diagram)
- but even "finishing" nodes may have children
  Depth d of trie = length of longest key value
  Cost of searching $O(d)$ (independent of n)

Basic operations on tries:

1. search for a key
2. insert a key

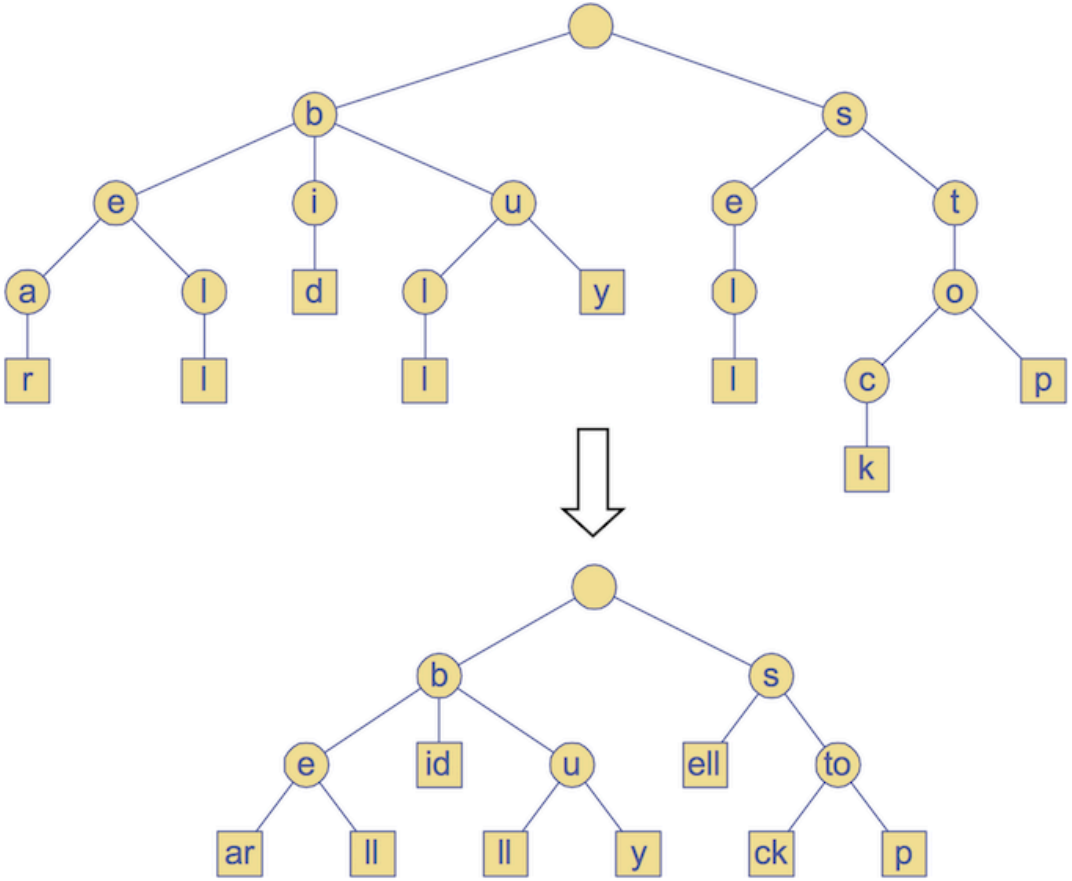
Performance of standard tries:

- $O(n)$ space
- insertion and search in time $O(d \cdot m)$

Compressed Tries

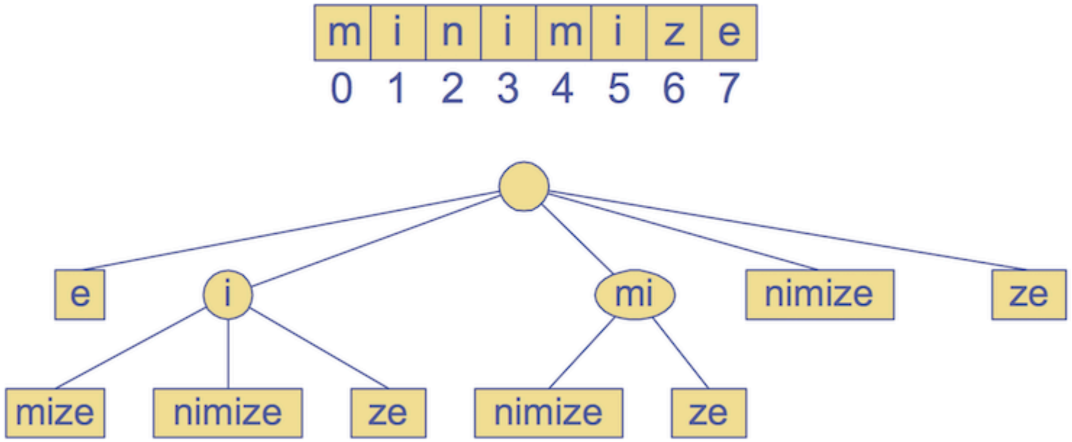
Compressed tries

- have internal nodes of degree ≥ 2
- are obtained from standard tries by compressing "redundant" chains of nodes (which have only 1 child)



Suffix Tries

The suffix trie of a text `T` is the compressed trie of all the suffixes of `T`.



Text Compression

Problem: Efficiently encode a given string X by a smaller string Y

Huffman's algorithm

- computes frequency f(c) for each character c
- encodes high-frequency characters with short code
- no code word is a prefix of another code word
- uses optimal encoding tree to determine the code words

Code: mapping of each character to a binary code word

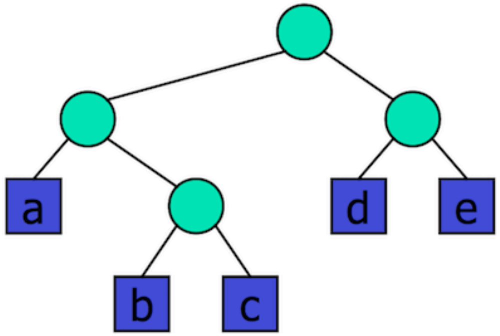
Prefix code: binary code such that no code word is prefix of another code word. (otherwise, it is impossible to uncode a string)

Encoding tree:

- represents a prefix code

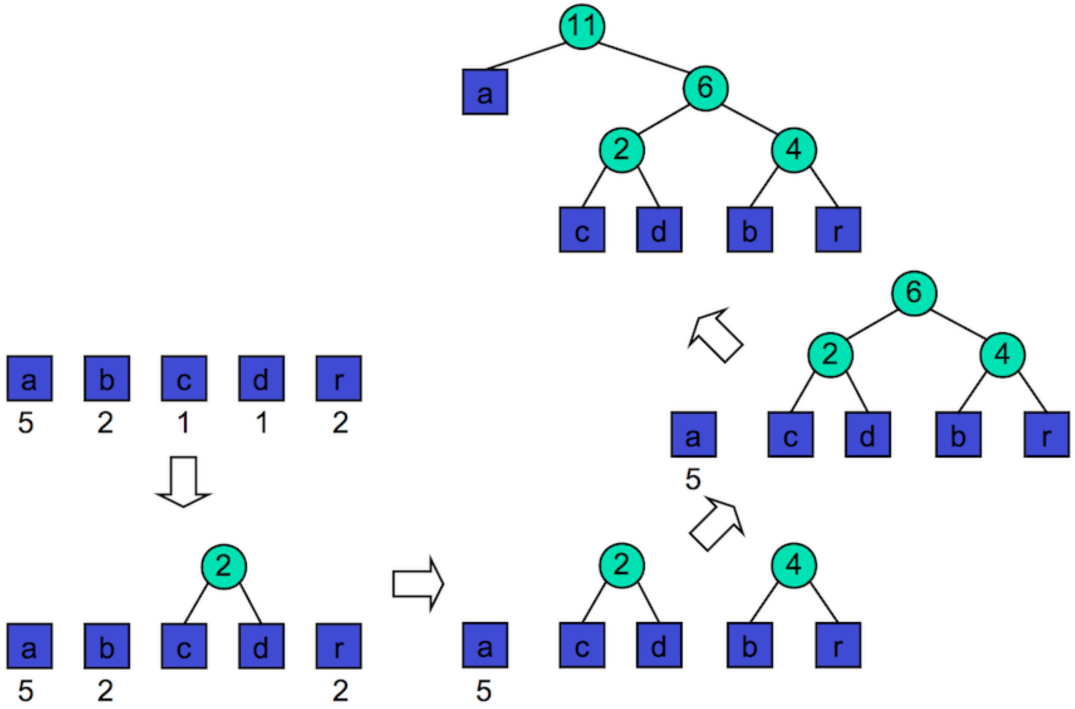
- each leaf stores a character
- code word given by the path from the root to the leaf (`0` for **left child**, `1` for **right child**)

00	010	011	10	11
a	b	c	d	e



How to build an encoding tree:

- computes **frequency** f(c) for each character
- successively combines pairs of lowest-frequency characters to build encoding tree "bottom-up"
  (make sure more frequent characters have less depth, which is equal to the length of its code)



- The numbers in the green nodes stand for the total frequency of their children
- All the leaves stand for a character. And parent nodes cannot be a character (*Prefix code*).

Huffman's algorithm for build an encoding tree by using **priority queue**:

```
HuffmanCode(T):
|  Input  string T of size n
|  Output optimal encoding tree for T
|
|  compute frequency array
|  Q=new priority queue
|  for all characters c do
|     T=new single-node tree storing c
|     join(Q,T) with frequency(c) as key
|  end for
|  while |Q|≥2 do
```

```
|     f1=Q.minKey(), T1=leave(Q)
|     f2=Q.minKey(), T2=leave(Q)
|     T=new tree node with subtrees T1 and T2
|     join(Q,T) with f1+f2 as key
|  end while
|  return leave(Q)
```