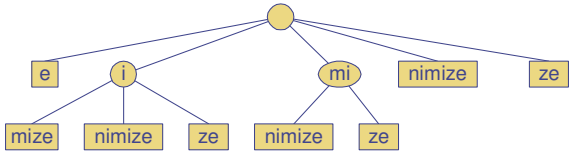
Structures en arbre pour le traitement de textes

Tries



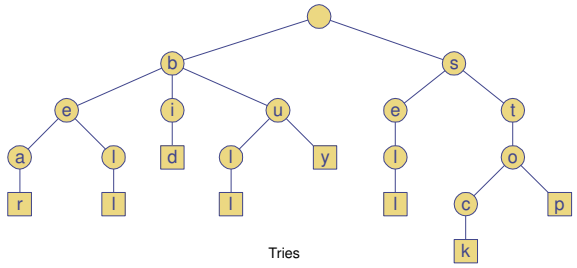
Trie: du mot anglais "*retrieval*"

Preprocessing Strings

- Un trie est une structure de données pour représenter un ensemble de chaines de caractères (par exemple, les mots dans un texte)
- Utile pour la recherche rapide
- Aussi pour comprimer un texte.

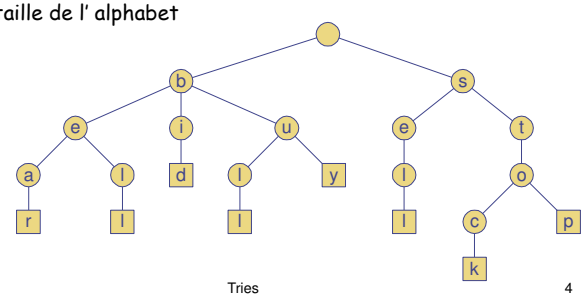
Standard Trie (1)

- Le trie de base est un ensemble de mots dans un arbre tel que:
  - Chaque noeud sauf la racine est un caractère
  - Les enfants sont en ordre alphabétique
  - Les différents chemins de la racine aux feuilles mènent à un mot de S
- Exemple:
  S = { bear, bell, bid, bull, buy, sell, stock, stop }



Trie de base (2)

- Un trie utilise $O(n)$ espace et requiert une complexité de $O(dm)$ pour l'insertion, la recherche et le retrait.
  - n nombre de mots dans S
  - m longueur d'un mot
  - d taille de l' alphabet



Trie de base (3)

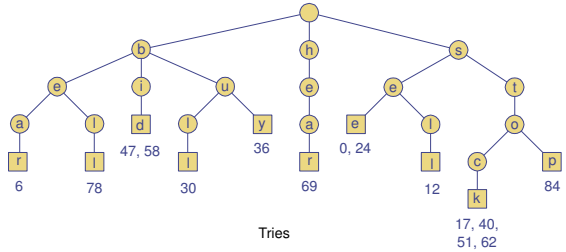
Pourquoi O(dm) pour la recherche?

par exemple "ZZZZZZZ"

A chaque niveau, il faut parcourir tout l'alphabet, donc 7*26 comparaisons!

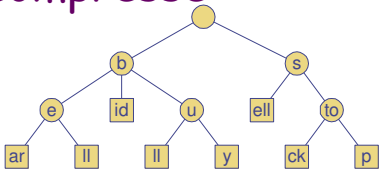
Recherche de mots avec un Trie

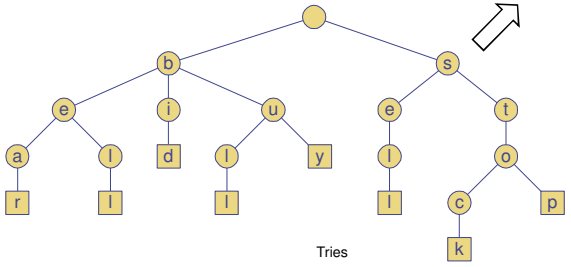
- Chaque feuille enregistre les positions d'occurences de chaque mot d'un texte



Trie compressé

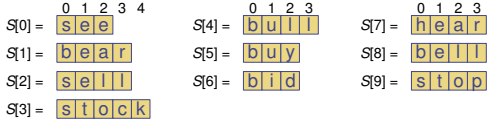
- Dans un trie compressé, chaque noeud doit avoir un degré d'au moins 2
- On élimine les branches redondantes

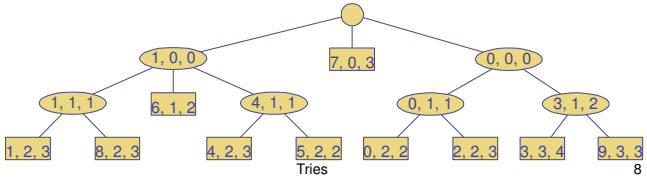




Représentation compacte

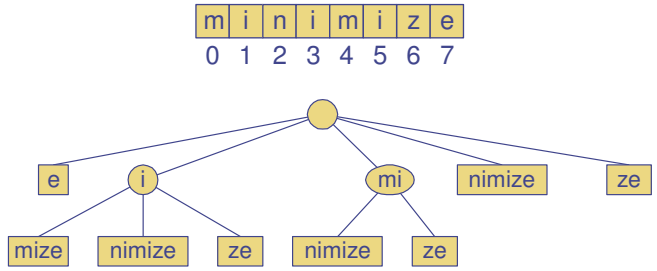
- Chaque noeud stocke le triplet (i,l,k) avec i le mot, l et k Sont les indices, e.g. (6,1,2) correspond à S[6] de 1 à 2 ou "id".





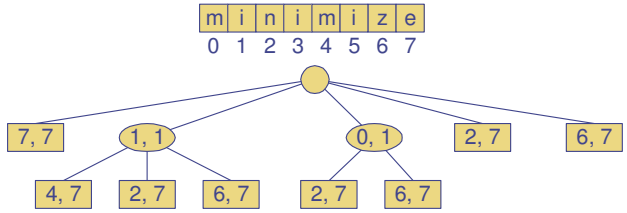
Trie Suffixe (1)

- Un trie suffixe d'un mot X est le trie compressé de tout ses suffixes



Trie Suffixe (2)

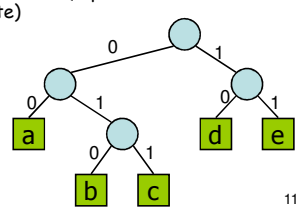
- La représentation compacte du trie suffixe d'un mot X de longueur n à partir d'un alphabet de taille d
  - Utilise un espace $O(n)$
  - Supporte une recherche de motifs avec une complexité $O(dm)$ ou e m est la longueur du motif



Codage avec Trie (1)

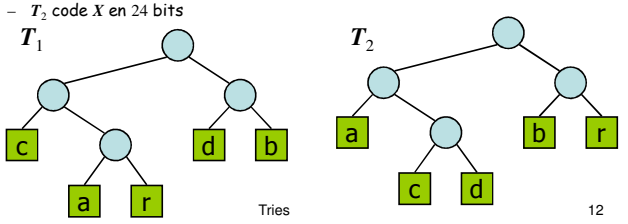
- Un code fait correspondre chaque caractère d'un alphabet avec un mot binaire
- Un code préfixe est un code binaire tel que aucun mot de code est le préfixe d'un autre (par exemple le code Morse)
- Un codage avec Trie produit un code préfixe
  - Chaque feuille est un caractère (un symbole)
  - On doit donc produire un arbre binaire ayant n feuilles
  - Le mot de code associé à un caractère est obtenu en suivant le chemin de la racine jusqu'à ce caractère (0 pour l'enfant de gauche et 1 pour l'enfant de droite)

00	010	011	10	11
a	b	c	d	e



Codage avec Trie (2)

- Soit une chaine X, nous voulons trouver un code préfixe pour les caractère de X donnant lieu à un codage compact
  - Les caractères les plus fréquents devraient avoir des mots de codes très courts
  - Les caractères rares peuvent avoir des codes plus long
- Example
  - X = abracadabra
  - T_1 code X en 29 bits
  - T_2 code X en 24 bits



Algorithme de Huffman

- Soit une chaine de caractères (ou de symboles) X, l'algorithme de Huffman construit un code préfixe qui minimise la taille du code pour X
- Il s'exécute en temps $O(n + d \log d)$, avec n la longueur de X et d le nombre de caractères distinct de X
- Une file à priorité est utilisée comme structure auxiliaire

Algorithme de Huffman

• Insérer chacun des caractères distincts de la chaine X dans une file à priorité en utilisant la fréquence d'apparition de chaque caractères comme clé; le calcul des fréquences est O(n).

•Retirer 2 noeuds de la file à priorité; O(log$d$).

•Combiner ces 2 noeuds et créer un arbre binaire dont la racine est la somme des deux clés; O(1).

•Insérer ce nouvel arbre dans la file à priorité (insérer un arbre correspond à insérer sa racine); O(log$d$).

•Répéter jusqu'à ce qu'il y ait un seul arbre dans la file; O(d).

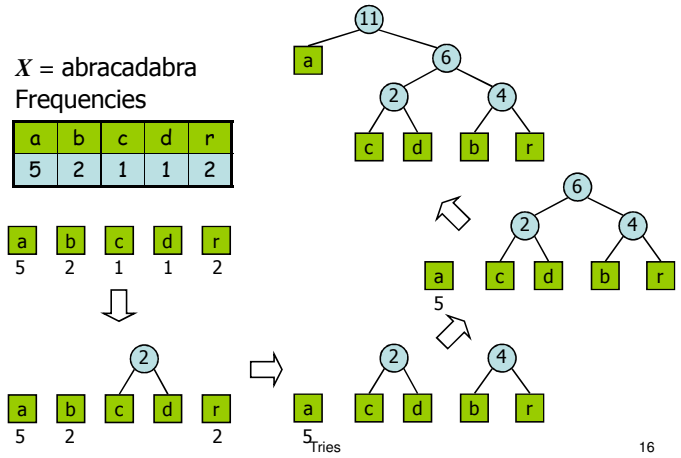
Huffman's Algorithm

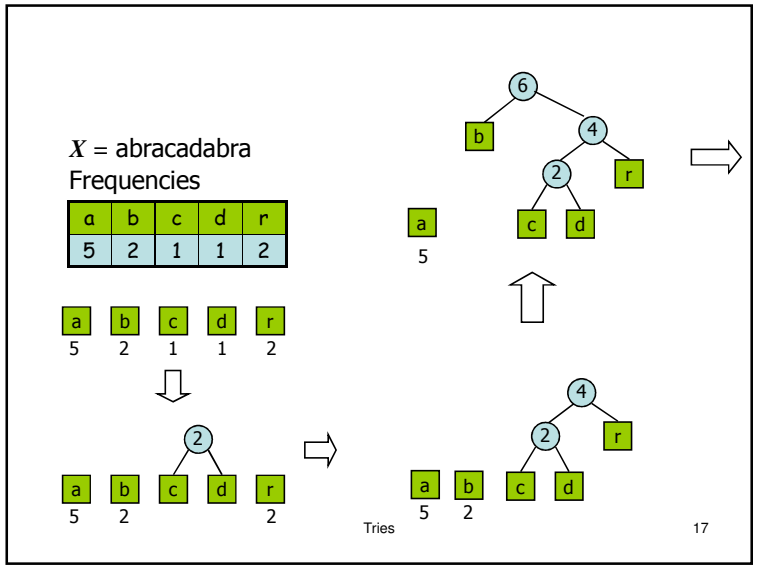
```
Algorithm HuffmanEncoding(X)
  Input string X of size n
  Output optimal encoding trie for X
  C ← distinctCharacters(X)
  computeFrequencies(C, X)
  Q ← new empty heap
  for all c ∈ C
    T ← new single-node tree storing c
    Q.insert(getFrequency(c), T)
  while Q.size() > 1
    f1 ← Q.minKey()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

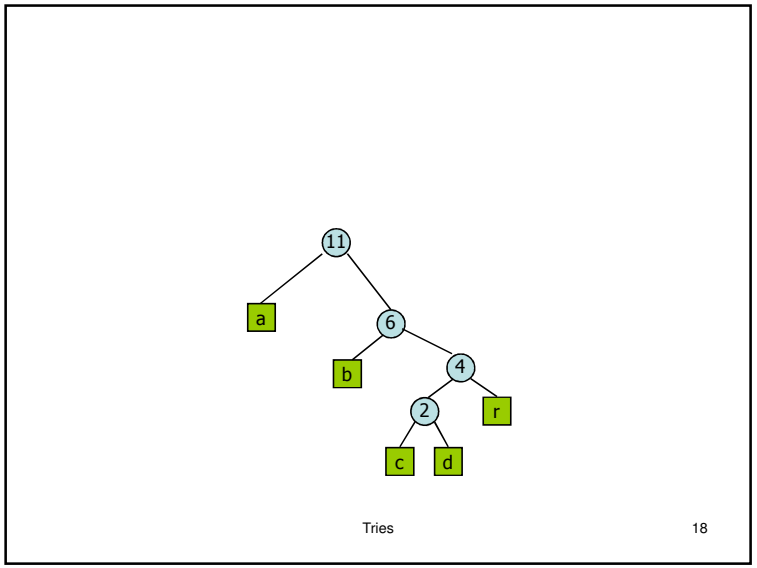

Exemple

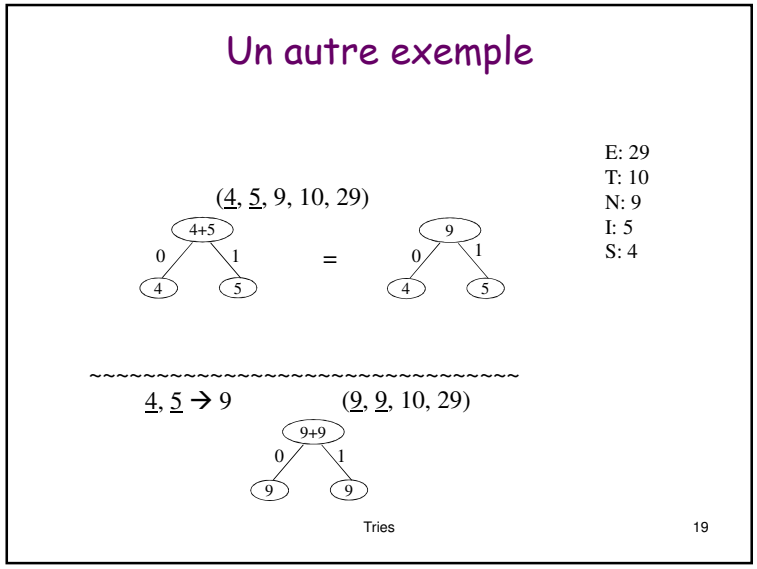
X = abracadabra
Frequencies

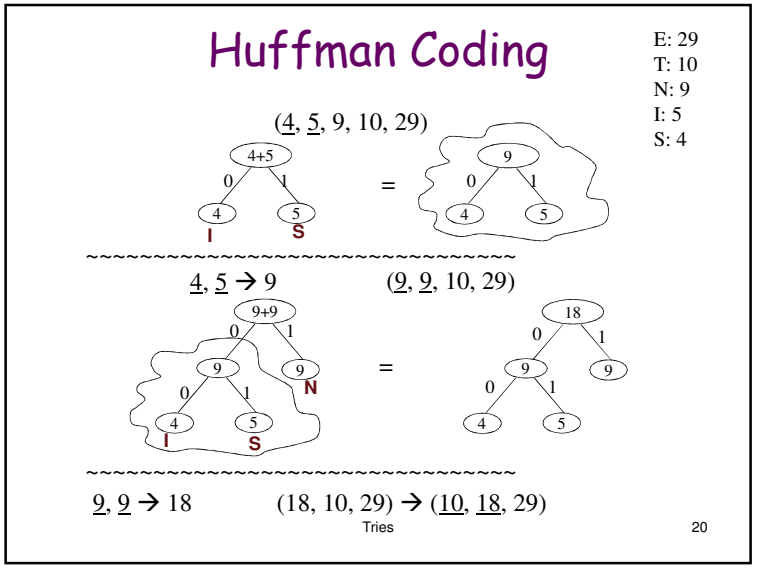
a	b	c	d	r
5	2	1	1	2

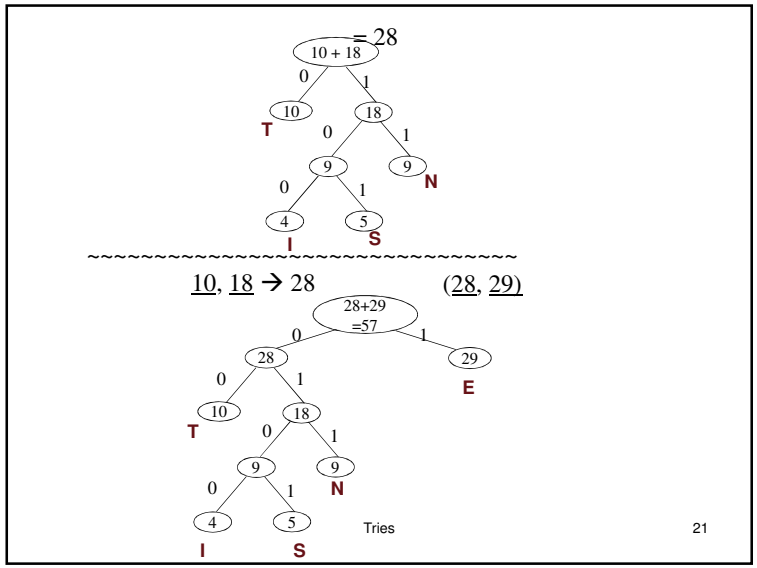


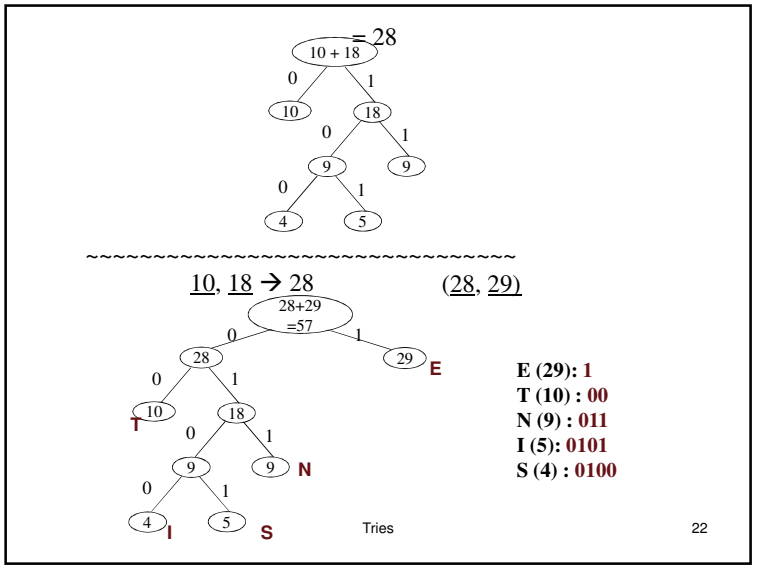

X = abracadabra
Frequencies
a b c d r
5 2 1 1 2
a b c d r
5 2 1 1 2
2
a b c d r
5 2 2
6
b 4
2 r
c d
a
5
4
2 r
a b c d
5 2
Tries
17


11
a 6
b 4
2 r
c d
Tries
18


Un autre exemple
(4, 5, 9, 10, 29)
4+5
0 1
4 5
=
9
0 1
4 5
E: 29
T: 10
N: 9
I: 5
S: 4
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
4, 5 → 9 (9, 9, 10, 29)
9+9
0 1
9 9
Tries
19


Huffman Coding
E: 29
T: 10
N: 9
I: 5
S: 4
(4, 5, 9, 10, 29)
4+5
0 1
4 5
I S
=
9
0 1
4 5
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
4, 5 → 9 (9, 9, 10, 29)
9+9
0 1
9 9
0 1
4 5
I S
N
=
18
0 1
9 9
0 1
4 5
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
9, 9 → 18 (18, 10, 29) → (10, 18, 29)
Tries
20
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


= 28
10 + 18
0
1
10
18
T
0
1
9
9
N
0
1
4
5
I
S
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
10, 18 → 28
(28, 29)
28+29
=57
0
1
28
29
E
0
1
10
18
T
0
1
9
9
N
0
1
4
5
I
S
Tries
21


= 28
10 + 18
0
1
10
18
0
1
9
9
0
1
4
5
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
10, 18 → 28
(28, 29)
28+29
=57
0
1
28
29
E
0
1
10
18
T
0
1
9
9
N
0
1
4
5
I
S
E (29): 1
T (10) : 00
N (9) : 011
I (5): 0101
S (4) : 0100
Tries
22