AED2 - Aula ~~19~~ - Tries

18 - bônus 1

Tries são árvores de busca digital em que toda chave está numa folha.

- Com isso, as chaves podem ser mantidas em ordem,
  - o que permite implementar de modo eficiente operações como
    - mínimo, máximo, predecessor, sucessor, percurso ordenado.
- As operações máximo e mínimo, relacionadas com ordem das chaves,
  - podem ser implementadas com eficiência nas árvores digitais básicas,
    - ainda que estas árvores não garantam a ordem das chaves.
  - Quiz1: Como? Por que?
- Curiosidade: O nome trie vem de “information reTRIEval”,
  - mas pronunciamos “try” para diferenciar de “tree”.

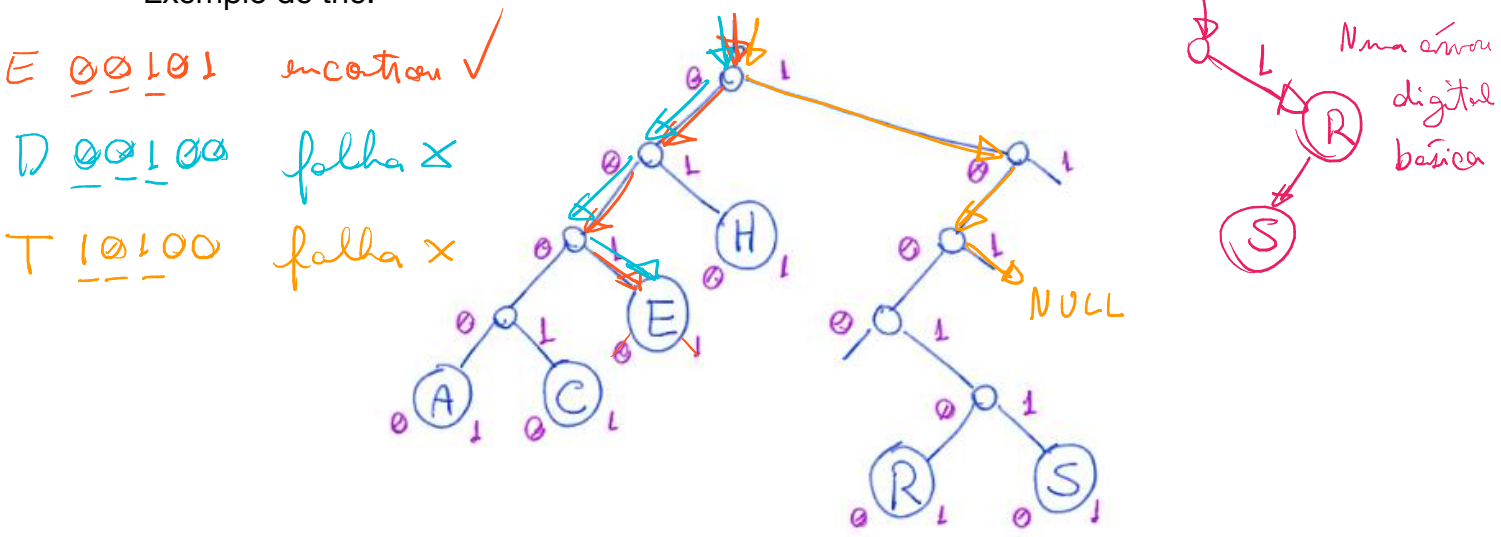
Nos seguintes exemplos consideramos a versão binária das tries,

- e usamos a seguinte representação binária de caracteres:

	A 00001	B 00010	C 00011
D 00100	E 00101	F 00110	G 00111
H 01000	I 01001	J 01010	K 01011
L 01100	M 01101	N 01110	O 01111
P 10000	Q 10001	R 10010	S 10011
T 10100	U 10101	V 10110	W 10111
X 11000	Y 11001	Z 11010	

- Os bits são numerados, a partir do índice 0, da esquerda para a direita.

Exemplo de trie:



- Uma propriedade central da trie é que todos os descendentes de um nó
  - tem prefixo comum com o daquele nó,
    - sendo que a raiz é associada com o prefixo vazio.
- Uma característica única das tries entre as árvores de busca,
  - é que sua estrutura depende apenas das chaves que ela armazena,
    - e não da ordem em que elas foram inseridas.
  - Quiz2: Por que?

Estrutura do nó:

```
typedef struct noh {
    Chave chave;
    Item conteudo;
    struct noh *esq;
    struct noh *dir;
} Noh;
```

Busca em trie:

- Para buscar uma chave, basta percorrer o caminho na árvore
  - seguindo os bits da chave (0 desce à esquerda, 1 à direita).
- Se chegar numa folha, verificar se é a chave procurada.
  - Se for devolve o nó, caso contrário devolve falha da busca.
    - Exemplos na árvore anterior: buscar E (00101) ou D (00100).
- Se chegar num apontador vazio, devolve falha da busca.
  - Exemplo na árvore anterior: buscar T (10100).

Código da busca:

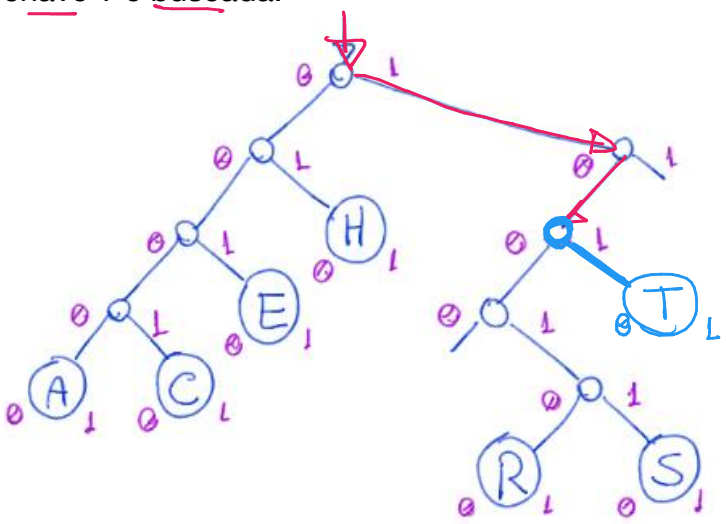
```
Noh *buscaR(Arvore r, Chave chave, int digito, Noh **ppai) {
    if (r == NULL)
        return r;
    if (r->esq == NULL && r->dir == NULL) { // eh uma folha
        if (r->chave == chave)
            return r;
        return NULL;
    }
    if (pegaDigito(chave, digito) == 0) { // desce à esquerda
        *ppai = r;
        return buscaR(r->esq, chave, digito + 1, ppai);
    }
    // pegaDigito(chave, digito) == 1 - desce à direita
    *ppai = r;
    return buscaR(r->dir, chave, digito + 1, ppai);
}
```

- Exemplo de uso

```
        aux = buscaR(r, chaves[i], 0, &pai);
```

Exemplo de inserção do T (10100):

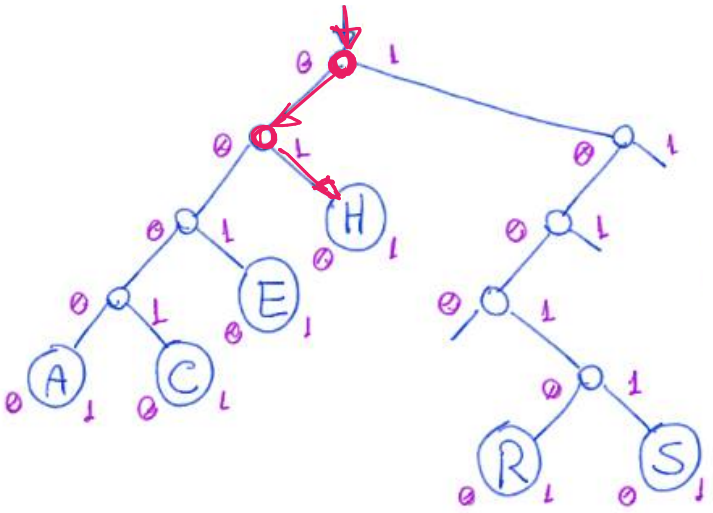
- Primeiro a chave T é buscada.



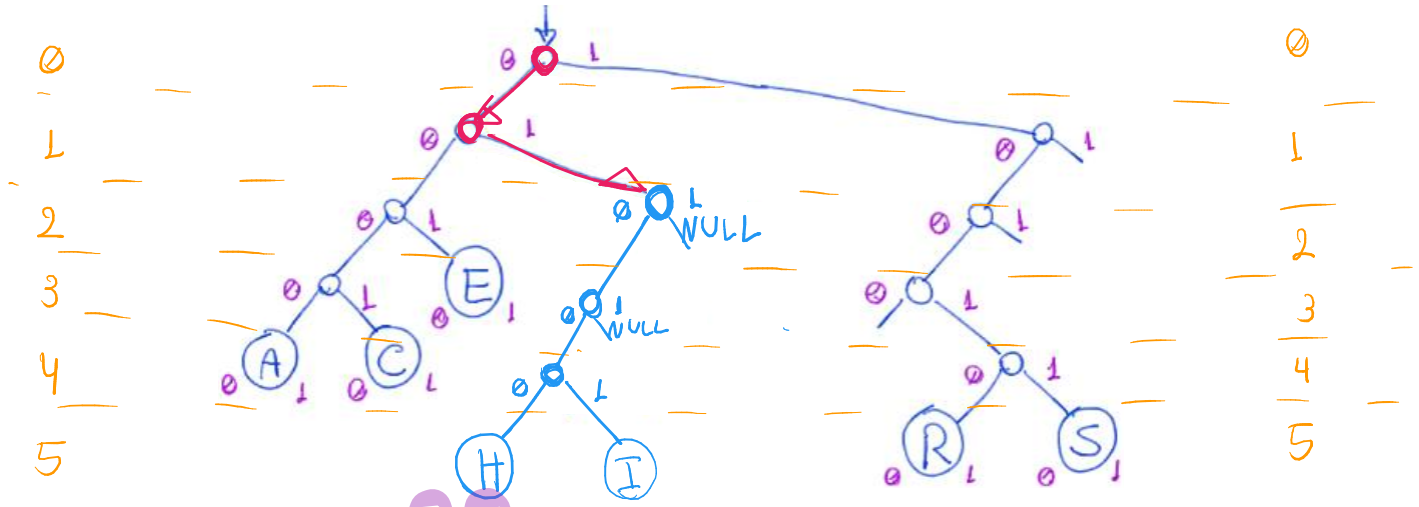
- Como a busca terminou em um apontador nulo de um nó interno,
  - basta substituir tal apontador pelo novo nó.

Exemplo de inserção do I (01001):

- Primeiro a chave I é buscada.



- Como a busca terminou em uma folha diferente de I
  - é necessário ramificar para separar as chaves.
- As chaves H (01000) e I (01001) coincidem nos 4 primeiros dígitos.
  - Por isso, é necessário criar nós internos até o nível 4



- Então inserimos H e I de acordo com o valor de seu próximo bit,
  - que é o primeiro bit em que eles diferem (no caso, é o bit 5°).

Códigos da inserção:

Função que invoca a criação de um novo nó e manda inseri-lo na árvore

```
Arvore inserir(Arvore r, Chave chave, Item conteudo) {
    Noh *novo = novoNoh(chave, conteudo);
    return insereR(r, novo, 0);
}
```

Função que cria um novo nó

```
Noh *novoNoh(Chave chave, Item conteudo) {
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = chave;
    novo->conteudo = conteudo;
    novo->esq = NULL;
    novo->dir = NULL;
    return novo;
}
```

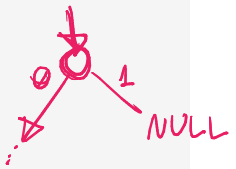
Função que insere recursivamente o novo nó na árvore

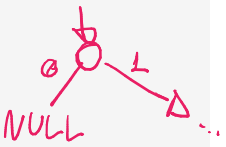
```
Arvore insereR(Arvore r, Noh *novo, int digito) {
    if (r == NULL) // insere folha
        return novo;
    if (r->esq == NULL && r->dir == NULL) // busca terminou em folha
        return ramifique(r, novo, digito);
    if (pegaDigito(novo->chave, digito) == 0) // desce à esquerda
        r->esq = insereR(r->esq, novo, digito + 1);
    else // pegaDigito(novo->chave, digito) == 1 - desce à direita
        r->dir = insereR(r->dir, novo, digito + 1);
    return r;
}
```

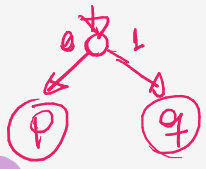
Função que faz a ramificação na árvore, criando novos nós internos,

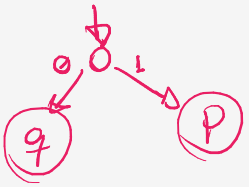
- quando duas folhas p e q compartilham um prefixo.

```
Arvore ramifique(Noh *p, Noh *q, int digito) {
    Noh *inter; // apontador para nó intermediário
    inter = (Noh *)malloc(sizeof(Noh));
    inter->chave = -1; // apenas para impressão
    if (pegaDigito(p->chave, digito) == pegaDigito(q->chave,
digito)) { // chaves não diferem no dígito atual
        if (pegaDigito(p->chave, digito) == 0) {
        // desce à esquerda do nó intermedário
            inter->dir = NULL;
            inter->esq = ramifique(p, q, digito + 1);
        }
        else { // pegaDigito(p->chave, digito) == 1
        // desce à direita do nó intermediário
            inter->esq = NULL;
            inter->dir = ramifique(p, q, digito + 1);
        }
    }
    else { // chaves diferem no dígito atual
        if (pegaDigito(p->chave, digito) == 0) {
        // insere p à esquerda e q à direita do nó intermediário
            inter->esq = p;
            inter->dir = q;
        }
        else { // pegaDigito(p->chave, digito) == 1
        // insere q à esquerda e p à direita do nó intermediário
            inter->esq = q;
            inter->dir = p;
        }
    }
    return inter;
}
```





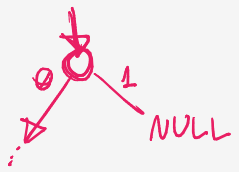


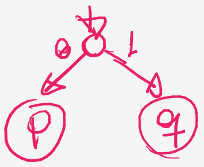


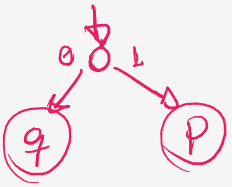
Versão mais elegante da ramificação,

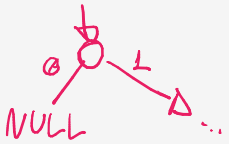
- que usa manipulação de bits e um switch para decidir o que fazer.
    - Inspirado no livro Algorithms in C++, Parts 1-4 de R. Sedgewick.

```
Arvore ramifique2(Noh *p, Noh *q, int digito) {
    Noh *inter; // apontador para nó intermediário
    inter = (Noh *)malloc(sizeof(Noh));
    inter->chave = -1; // apenas para impressão
    switch (pegaDigito(p->chave, digito) * 2 + pegaDigito(q->chave,
digito)) {
    // lembre que em binário 0 = 00, 1 = 01, 2 = 10, 3 = 11
    case 0: // os dígitos das chaves são iguais a 0 - desce à
esquerda do nó intermedário
        inter->esq = ramifique2(p, q, digito + 1);
        inter->dir = NULL;
        break;
    case 1: // dígito de p é 0 e de q é 1 - insere p à esquerda e q
à direita do nó intermediário
        inter->esq = p;
        inter->dir = q;
        break;
    case 2: // dígito de p é 1 e de q é 0 - insere q à esquerda e p
à direita do nó intermediário
        inter->esq = q;
        inter->dir = p;
        break;
    case 3: // os dígitos das chaves são iguais a 1 - desce à
direita do nó intermedário
        inter->dir = ramifique2(p, q, digito + 1);
        inter->esq = NULL;
        break;
    }
    return inter;
}
```



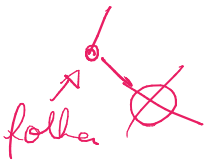






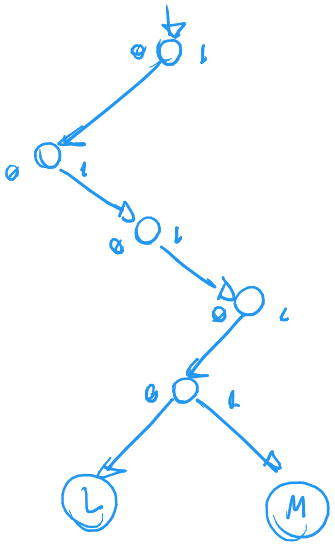
Para a operação de remoção

- podemos usar funções semelhantes àquelas da árvore digital básica,
  - atentando que sempre iremos remover uma folha
- e que ao remover as folhas de um nó intermediário,
  - ele pode se tornar uma folha, que deve ser removida.
- Para fazer isso, no caminho de volta da recursão
  - podemos verificar se cada nó intermediário se tornou folha
    - e aproveitar para eliminá-lo.
- Quiz3: Adapte a função de remoção vista na aula anterior para as Tries.



Quanto à eficiência de tempo das operações, elas continuam sendo

- proporcionais à altura da árvore,
  - que no pior caso corresponde ao comprimento da chave,
    - i.e., ao número de dígitos da mesma.
- Mas este pior caso pode ocorrer com mais facilidade,
  - bastando duas chaves que só diferem no último dígito.
    - Ex.: L (01100) e M (01101)



Quanto à eficiência de espaço, uma trie pode precisar

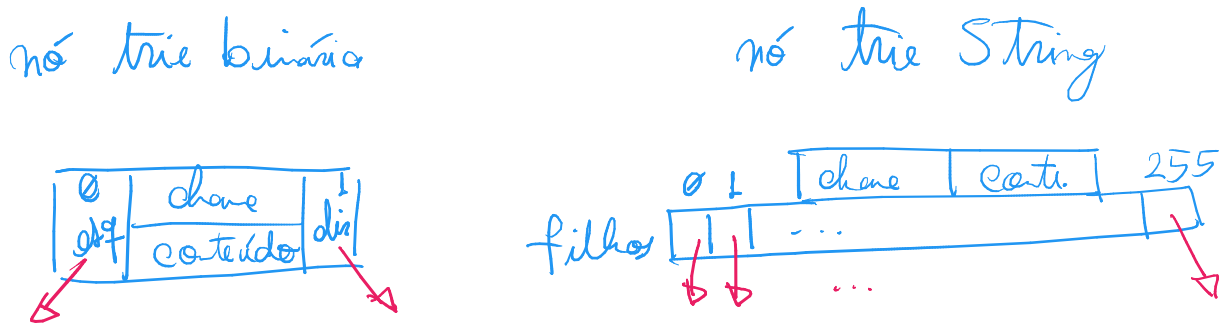
  - de muitos nós internos para armazenar poucas folhas.
- De fato, desperdício de memória é um problema das tries.
  - Embora elas ocupem espaço proporcional ao número de itens,
    - se as chaves forem aleatórias.

Curiosidade: podemos melhorar um pouco a eficiência de espaço das tries,

- declarando dois tipos de estrutura,
  - uma para nós internos (sem os campos chave e conteúdo),
    - e outra para nós folhas (com todos os campos).
- No entanto, isso gera algumas complicações na implementação,
  - pois variáveis do tipo ponteiro vão apontar
    - para células cujas estruturas são diferentes.

Assim como fizemos com as árvores digitais básicas,

- podemos construir tries para tratar chaves
  - que são strings ou que tem dígitos com mais de 1 bit.



Neste caso o gasto de memória por nó cresce, pois

- cada nó terá um vetor de filhos do tamanho do universo de valores
  - que um caractere da string ou dígito da chave pode assumir.
- Por exemplo, se cada caractere da chave tem 8 bits,
  - um único caractere pode indicar $2^8 = 256$ caminhos distintos,
    - i.e., cada nó deve ter um vetor de filhos com 256 apontadores.

Código:

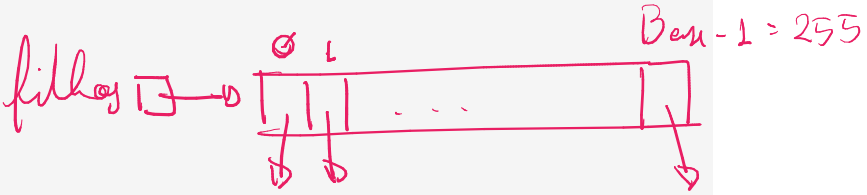
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

typedef int Item;
typedef char byte;
typedef byte *Chave;

const int bitsDigito = 8;
const int Base = 1 << bitsDigito; // Base = 2^bitsDigito

typedef struct noh {
    Chave chave;
    Item conteudo;
    struct noh **filhos;
} Noh;

typedef Noh *Arvore;
```



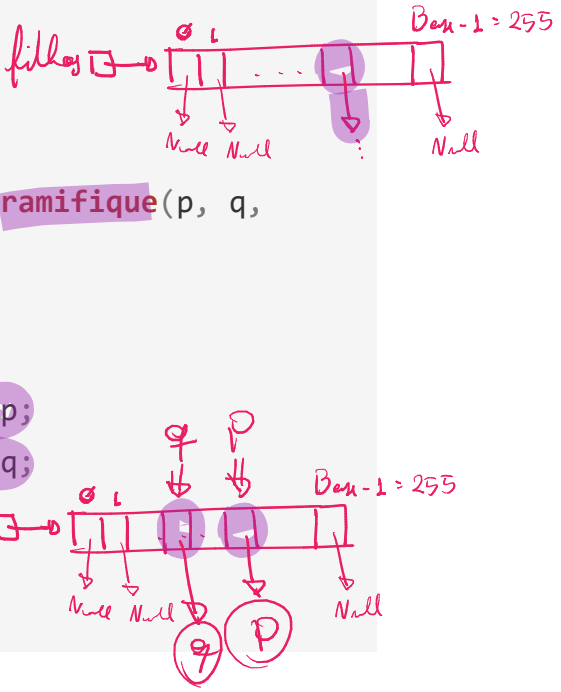
```c
Noh *buscaR(Arvore r, Chave chave, int digito, Noh **ppai) {
    int i;
    if (r == NULL)
        return r;
    for (i = 0; i < Base; i++)
        if (r->filhos[i] != NULL)
            break;
    if (i == Base) { // eh uma folha
        if (strcmp(r->chave, chave) == 0)
            return r;
        return NULL;
    }
    *ppai = r;
    return buscaR(r->filhos[(int)chave[digito]], chave, digito + 1,
ppai);
}

Arvore inserir(Arvore r, Chave chave, Item conteudo) {
    Noh *novo = novoNoh(chave, conteudo);
    return insereR(r, novo, 0);
}

Noh *novoNoh(Chave chave, Item conteudo) {
    int i;
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = (char *)malloc((strlen(chave) + 1) *
sizeof(char));
    strcpy(novo->chave, chave);
    novo->conteudo = conteudo;
    novo->filhos = malloc(Base * sizeof(int));
    for (i = 0; i < Base; i++)
        novo->filhos[i] = NULL;
    return novo;
}
```

```c
// Quiz4: como melhorar a eficiência dessa função?
Arvore insereR(Arvore r, Noh *novo, int digito) {
    int i;
    if (r == NULL) return novo; // insere folha
    for (i = 0; i < Base; i++)
        if (r->filhos[i] != NULL) break;
    if (i == Base) // busca terminou em folha
        return ramifique(r, novo, digito);
    i = (int)(novo->chave[digito]);
    r->filhos[i] = insereR(r->filhos[i], novo, digito + 1);
    return r;
}

Arvore ramifique(Noh *p, Noh *q, int digito) {
    Noh *inter; // apontador para nó intermediário
    inter = (Noh *)malloc(sizeof(Noh));
    inter->chave = (char *)malloc(3 * sizeof(char));
    inter->chave = "-1\0"; // apenas para impressão
    inter->filhos = malloc(Base * sizeof(int));
    for (int i = 0; i < Base; i++)
        inter->filhos[i] = NULL;
    if (p->chave[digito] == q->chave[digito]) {
    // chaves não diferem no dígito atual
        inter->filhos[(int)p->chave[digito]] = ramifique(p, q,
digito + 1);
    }
    else { // chaves diferem no dígito atual
        inter->filhos[(int)p->chave[digito]] = p;
        inter->filhos[(int)q->chave[digito]] = q;
    }
    return inter;
}
```



Uma versão mais sofisticada das tries, chamada Patricia Tries

- evita desperdiçar espaço, tem operações mais eficientes em tempo,
    - e pode ser usada para indexar chaves de tamanho variável.
- O termo PATRICIA é um acrônimo para
    - Practical Algorithm to Retrieve Information Coded in Alphanumeric.