AED2 - Aula ~~20~~ - PATRICIA Tries

18 - bônus 2

Nos seguintes exemplos consideramos a versão binária das tries,

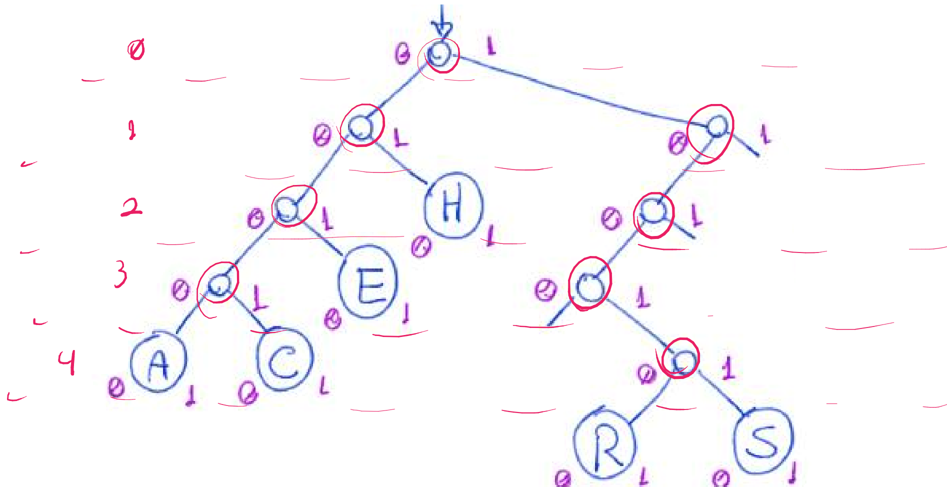
- e usamos a seguinte representação binária de caracteres:

	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.

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 diversas operações.



- Uma propriedade central da trie é que todos os descendentes de um nó
    - tem prefixo comum com o daquele nó.

Problemas das tries:

- Não armazenar dados nos nós internos desperdiça memória.
- Longos caminhos podem surgir para diferenciar duas chaves
    - cujos bits mais significativos são iguais.
- Isso porque alguns nós intermediários não correspondem a bifurcações.

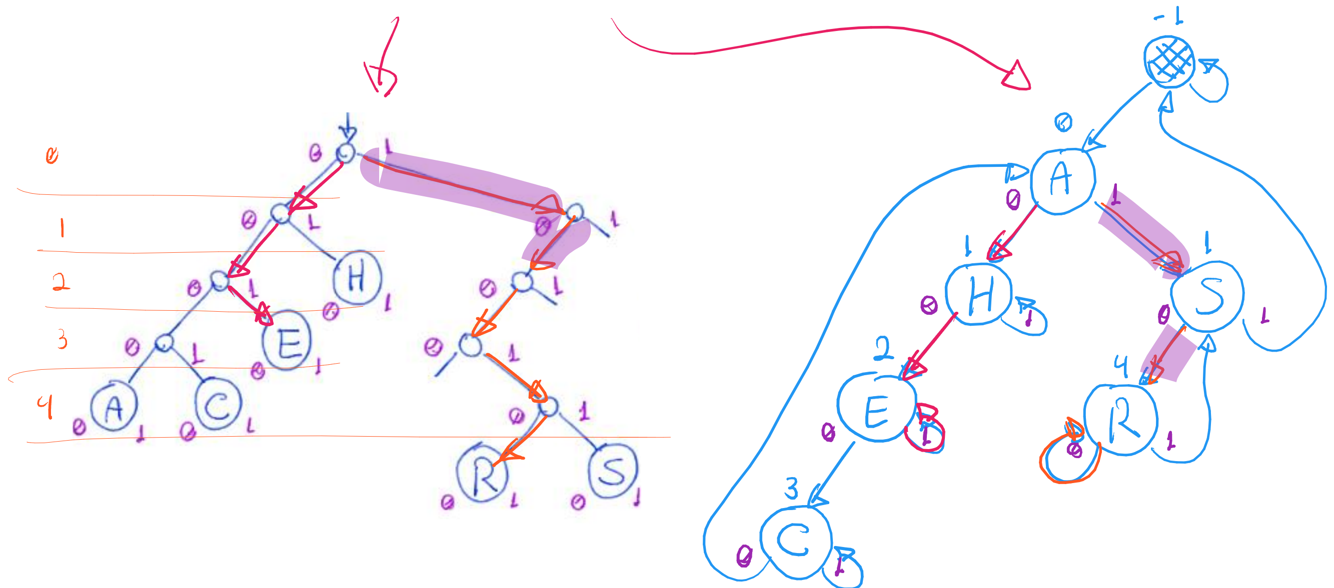
Para contornar estes problemas podemos usar as PATRICIA tries:

- Practical Algorithm To Retrieve Information Coded In Alphanumeric

Nas PATRICIA tries:

- Nós internos são aproveitados para armazenar chaves,
    - embora essas não sejam consideradas durante a descida na árvore.
- Longos caminhos são evitados,
    - olhando em cada nó apenas para o dígito que importa.

Exemplo comparando trie com PATRICIA trie:



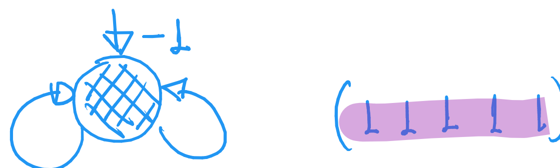
Estrutura do nó:

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

typedef Noh *Arvore;
```

Nas PATRICIA tries não temos apontadores NULL.

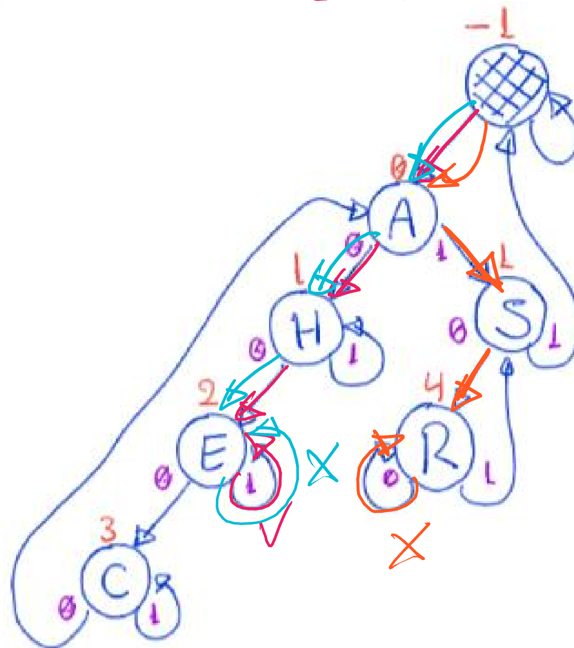
- Por isso, mesmo a árvore vazia tem de ser inicializada com uma raiz dummy
    - que tem uma chave proibida, preenchida de bits 1s, e digito -1.



```
Arvore inicializa() {
    Noh *raiz;
    raiz = (Noh *)malloc(sizeof(Noh));
    raiz->chave = __UINT32_MAX__; // chave proibida
    raiz->esq = raiz;
    raiz->dir = raiz;
    raiz->digito = -1;
    return raiz;
}
```

Busca na PATRICIA trie:

- Para buscar uma chave, basta percorrer o caminho na árvore
    - seguindo os bits da chave (0 desce à esquerda, 1 à direita),
        - lembrando de olhar em cada nó para o dígito indicado por este.
- Nas PATRICIA tries é importante diferenciar
    - o caminho conceitual do caminho real,
        - até porque as PATRICIA tries só são árvores conceituais.
- Quando "subirmos" na árvore, ou seja,
    - formos para um nó com dígito ≤ ao dígito do nó anterior
        - sabemos que chegamos numa "folha".
- Ao chegar numa "folha", verificamos se é a chave procurada.
    - Se for devolve o nó, caso contrário devolve falha da busca.
- Exemplos na árvore anterior: buscar E (00101), D (00100) e T (10100).



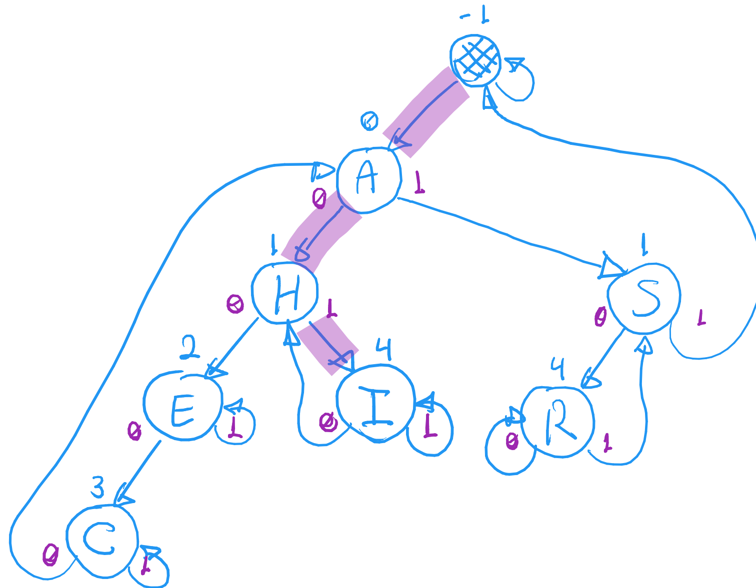
Código da busca:

```
Noh *busca(Arvore r, Chave chave) {
    Noh *alvo;
    alvo = buscaR(r->esq, chave, -1);
    return alvo->chave == chave ? alvo : NULL;
}

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

Exemplo de inserção do I (01001):

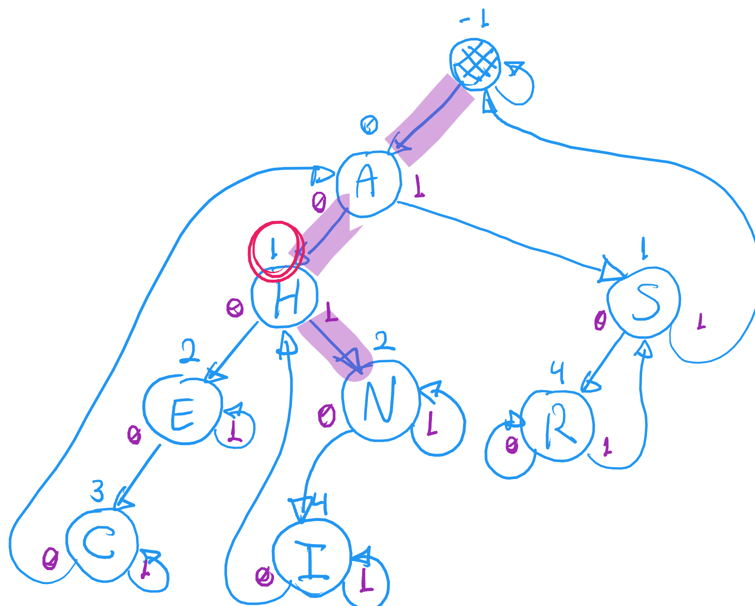
- Primeiro a chave I é buscada.



- Como a busca terminou em H (01000),
  - verificamos o primeiro dígito em que I (01001) e H (01000) diferem.
    - Neste caso, é o 4
- Então criamos um novo nó à direita de H.

Exemplo de inserção do N (01110):

- Primeiro a chave N é buscada.

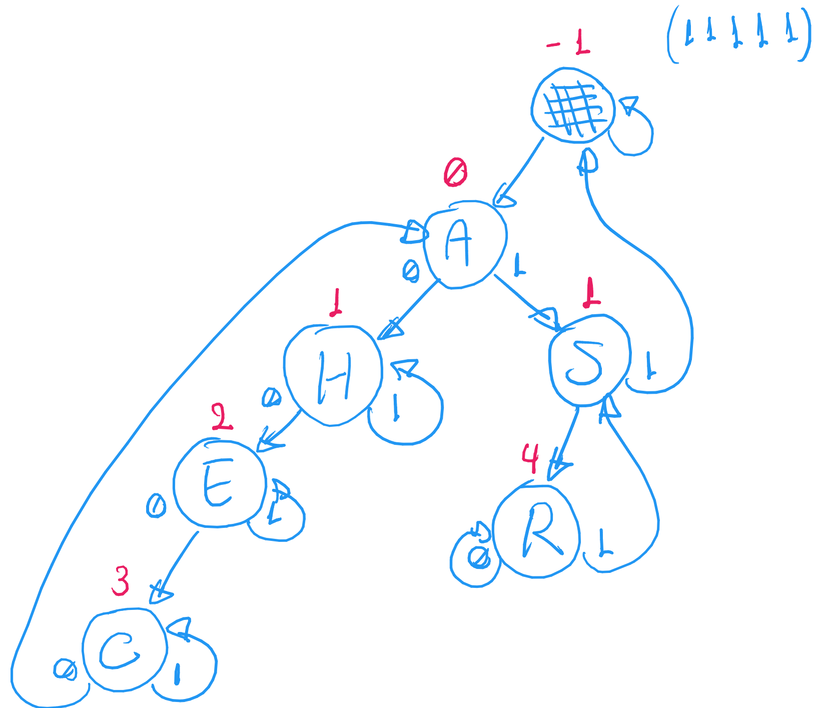


- Como a busca terminou em H (01000),
  - verificamos o primeiro dígito em que N (01110) e H (01000) diferem.
    - Neste caso, é o 2
- Então criamos um novo nó entre H e I.

Resultado da construção de uma PATRICIA trie binária, pela inserção

- das chaves A S E R C H em uma árvore vazia, exceto pela raiz dummy.

	0	1	2	3	4
	1	1	1	1	1
A	0	0	0	0	1
S	1	0	0	1	1
E	0	0	1	0	1
R	1	0	0	1	0
C	0	0	0	1	1
H	0	1	0	0	0



Códigos da inserção:

Função que invoca a busca, detecta o primeiro dígito distinto entre as chaves,

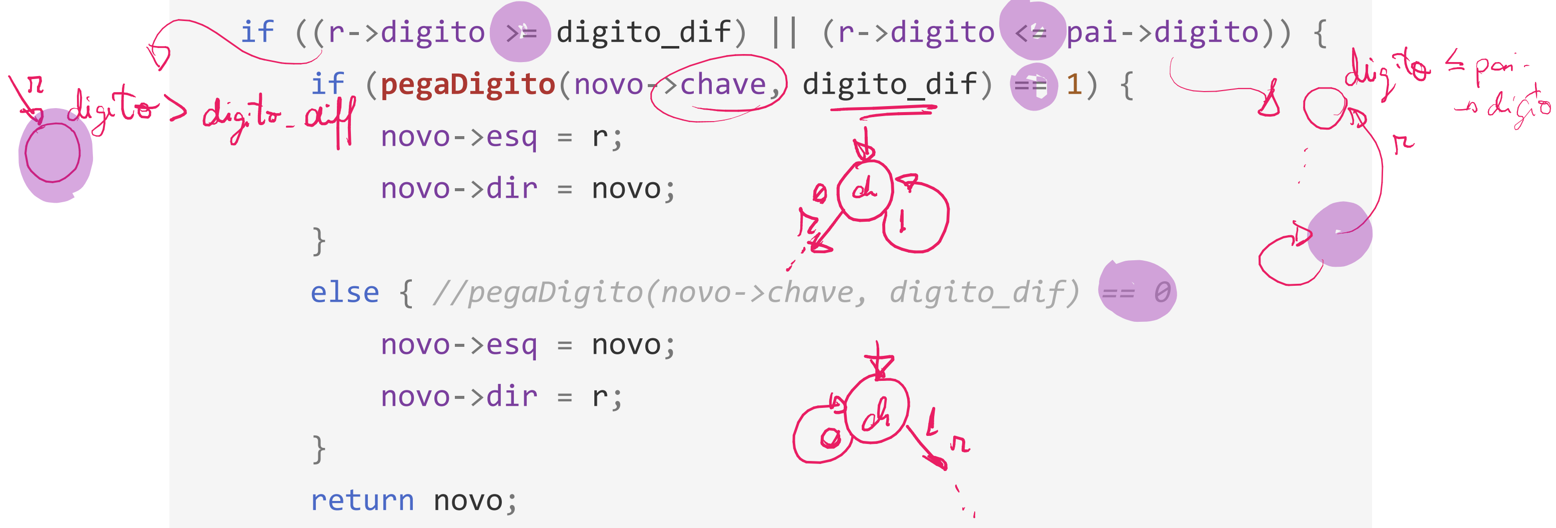
- cria um novo nó e manda inseri-lo na árvore.

```
void inserir(Arvore r, Chave chave, Item conteudo) {
    int i;
    Noh *aux = buscaR(r->esq, chave, -1);
    if (aux->chave == chave)
        return; // não inserimos duplicatas
    for (i = 0; pegaDigito(chave, i) == pegaDigito(aux->chave, i);
i++)
        ; // descobre qual o primeiro dígito diferente nas chaves
    Noh *novo = novoNoh(chave, conteudo, i);
    r->esq = insereR(r->esq, novo, i, r);
}
```

Função que cria um novo nó.

```
Noh *novoNoh(Chave chave, Item conteudo, int digito) {
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = chave;
    novo->conteudo = conteudo;
    novo->digito = digito;
    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_dif, Noh *pai) {
    // se encontrei o ponto de quebra ou cheguei numa folha
    if ((r->digito >= digito_dif) || (r->digito <= pai->digito)) {
        if (pegaDigito(novo->chave, digito_dif) == 1) {
            novo->esq = r;
            novo->dir = novo;
        }
        else { //pegaDigito(novo->chave, digito_dif) == 0
            novo->esq = novo;
            novo->dir = r;
        }
        return novo;
    }
```

```
    if (pegaDigito(novo->chave, r->digito) == 0) // inserir descendo à esquerda
        r->esq = insereR(r->esq, novo, digito_dif, r);
    else // pegaDigito(novo->chave, r->digito) == 1 - inserir descendo à direita
        r->dir = insereR(r->dir, novo, digito_dif, r);
    return r;
}
```

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.
- Em muitas situações, a altura da árvore é ainda menor.
    - Por exemplo, se as chaves forem aleatórias
        - a altura é da ordem de $\lg n$
    - já que a cada decida na árvore, como as chaves são aleatórias,
        - esperamos dividir por 2
            - o número de chaves na subárvore corrente.

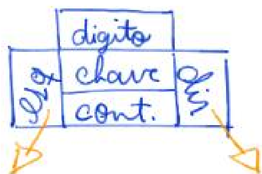
Quanto à eficiência de espaço, as PATRICIA tries

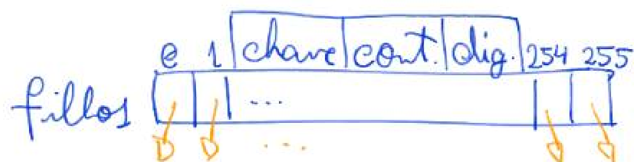
- gastam apenas um nó por elemento armazenado,
    - não tendo o problema com nós internos das tries.

Assim como fizemos com as árvores digitais básicas,

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

nó PATRICIA trie binária VS. nó PATRICIA trie String





- Neste caso o gasto de memória por nó cresce, pois cada nó terá
    - um vetor de filhos do tamanho do universo de valores que
        - os caracteres da string ou dígitos da chave podem assumir.
- Por exemplo, se cada caracter 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.