

18 - bônus 1

- 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”.

- 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	

- then ✓
- a ✗
- a ✗
- NULL
-
- ```
graph TD; Root(()) -- 0 --> L1(()); Root -- 1 --> R1(()); L1 -- 0 --> L2(()); L1 -- 1 --> H((H)); L2 -- 0 --> L3(()); L2 -- 1 --> E((E)); L3 -- 0 --> A((A)); L3 -- 1 --> C((C)); R1 -- 0 --> L4(()); R1 -- 1 --> R2(()); L4 -- 0 --> R3(()); L4 -- 1 --> R4(()); R3 -- 0 --> R5(()); R3 -- 1 --> R6(()); R5 -- 0 --> R7(()); R5 -- 1 --> R8(()); R7 -- 0 --> R9(()); R7 -- 1 --> R10(()); R9 -- 0 --> R11(()); R9 -- 1 --> R12(()); R11 -- 0 --> R13(()); R11 -- 1 --> R14(()); R13 -- 0 --> R15(()); R13 -- 1 --> R16(()); R15 -- 0 --> R17(()); R15 -- 1 --> R18(()); R17 -- 0 --> R19(()); R17 -- 1 --> R20(()); R19 -- 0 --> R21(()); R19 -- 1 --> R22(()); R21 -- 0 --> R23(()); R21 -- 1 --> R24(()); R23 -- 0 --> R25(()); R23 -- 1 --> R26(()); R25 -- 0 --> R27(()); R25 -- 1 --> R28(()); R27 -- 0 --> R29(()); R27 -- 1 --> R30(()); R29 -- 0 --> R31(()); R29 -- 1 --> R32(()); R31 -- 0 --> R33(()); R31 -- 1 --> R34(()); R33 -- 0 --> R35(()); R33 -- 1 --> R36(()); R35 -- 0 --> R37(()); R35 -- 1 --> R38(()); R37 -- 0 --> R39(()); R37 -- 1 --> R40(()); R39 -- 0 --> R41(()); R39 -- 1 --> R42(()); R41 -- 0 --> R43(()); R41 -- 1 --> R44(()); R43 -- 0 --> R45(()); R43 -- 1 --> R46(()); R45 -- 0 --> R47(()); R45 -- 1 --> R48(()); R47 -- 0 --> R49(()); R47 -- 1 --> R50(()); R49 -- 0 --> R51(()); R49 -- 1 --> R52(()); R51 -- 0 --> R53(()); R51 -- 1 --> R54(()); R53 -- 0 --> R55(()); R53 -- 1 --> R56(()); R55 -- 0 --> R57(()); R55 -- 1 --> R58(()); R57 -- 0 --> R59(()); R57 -- 1 --> R60(()); R59 -- 0 --> R61(()); R59 -- 1 --> R62(()); R61 -- 0 --> R63(()); R61 -- 1 --> R64(()); R63 -- 0 --> R65(()); R63 -- 1 --> R66(()); R65 -- 0 --> R67(()); R65 -- 1 --> R68(()); R67 -- 0 --> R69(()); R67 -- 1 --> R70(()); R69 -- 0 --> R71(()); R69 -- 1 --> R72(()); R71 -- 0 --> R73(()); R71 -- 1 --> R74(()); R73 -- 0 --> R75(()); R73 -- 1 --> R76(()); R75 -- 0 --> R77(()); R75 -- 1 --> R78(()); R77 -- 0 --> R79(()); R77 -- 1 --> R80(()); R79 -- 0 --> R81(()); R79 -- 1 --> R82(()); R81 -- 0 --> R83(()); R81 -- 1 --> R84(()); R83 -- 0 --> R85(()); R83 -- 1 --> R86(()); R85 -- 0 --> R87(()); R85 -- 1 --> R88(()); R87 -- 0 --> R89(()); R87 -- 1 --> R90(()); R89 -- 0 --> R91(()); R89 -- 1 --> R92(()); R91 -- 0 --> R93(()); R91 -- 1 --> R94(()); R93 -- 0 --> R95(()); R93 -- 1 --> R96(()); R95 -- 0 --> R97(()); R95 -- 1 --> R98(()); R97 -- 0 --> R99(()); R97 -- 1 --> R100(()); R99 -- 0 --> R101(()); R99 -- 1 --> R102(()); R101 -- 0 --> R103(()); R101 -- 1 --> R104(()); R103 -- 0 --> R105(()); R103 -- 1 --> R106(()); R105 -- 0 --> R107(()); R105 -- 1 --> R108(()); R107 -- 0 --> R109(()); R107 -- 1 --> R110(()); R109 -- 0 --> R111(()); R109 -- 1 --> R112(()); R111 -- 0 --> R113(()); R111 -- 1 --> R114(()); R113 -- 0 --> R115(()); R113 -- 1 --> R116(()); R115 -- 0 --> R117(()); R115 -- 1 --> R118(()); R117 -- 0 --> R119(()); R117 -- 1 --> R120(()); R119 -- 0 --> R121(()); R119 -- 1 --> R122(()); R121 -- 0 --> R123(()); R121 -- 1 --> R124(()); R123 -- 0 --> R125(()); R123 -- 1 --> R126(()); R125 -- 0 --> R127(()); R125 -- 1 --> R128(()); R127 -- 0 --> R129(()); R127 -- 1 --> R130(()); R129 -- 0 --> R131(()); R129 -- 1 --> R132(()); R131 -- 0 --> R133(()); R131 -- 1 --> R134(()); R133 -- 0 --> R135(()); R133 -- 1 --> R136(()); R135 -- 0 --> R137(()); R135 -- 1 --> R138(()); R137 -- 0 --> R139(()); R137 -- 1 --> R140(()); R139 -- 0 --> R141(()); R139 -- 1 --> R142(()); R141 -- 0 --> R143(()); R141 -- 1 --> R144(()); R143 -- 0 --> R145(()); R143 -- 1 --> R146(()); R145 -- 0 --> R147(()); R145 -- 1 --> R148(()); R147 -- 0 --> R149(()); R147 -- 1 --> R150(()); R149 -- 0 --> R151(()); R149 -- 1 --> R152(()); R151 -- 0 --> R153(()); R151 -- 1 --> R154(()); R153 -- 0 --> R155(()); R153 -- 1 --> R156(()); R155 -- 0 --> R157(()); R155 -- 1 --> R158(()); R157 -- 0 --> R159(()); R157 -- 1 --> R160(()); R159 -- 0 --> R161(()); R159 -- 1 --> R162(()); R161 -- 0 --> R163(()); R161 -- 1 --> R164(()); R163 -- 0 --> R165(()); R163 -- 1 --> R166(()); R165 -- 0 --> R167(()); R165 -- 1 --> R168(()); R167 -- 0 --> R169(()); R167 -- 1 --> R170(()); R169 -- 0 --> R171(()); R169 -- 1 --> R172(()); R171 -- 0 --> R173(()); R171 -- 1 --> R174(()); R173 -- 0 --> R175(()); R173 -- 1 --> R176(()); R175 -- 0 --> R177(()); R175 -- 1 --> R178(()); R177 -- 0 --> R179(()); R177 -- 1 --> R180(()); R179 -- 0 --> R181(()); R179 -- 1 --> R182(()); R181 -- 0 --> R183(()); R181 -- 1 --> R184(()); R183 -- 0 --> R185(()); R183 -- 1 --> R186(()); R185 -- 0 --> R187(()); R185 -- 1 --> R188(()); R187 -- 0 --> R189(()); R187 -- 1 --> R190(()); R189 -- 0 --> R191(()); R189 -- 1 --> R192(()); R191 -- 0 --> R193(()); R191 -- 1 --> R194(()); R193 -- 0 --> R195(()); R193 -- 1 --> R196(()); R195 -- 0 --> R197(()); R195 -- 1 --> R198(()); R197 -- 0 --> R199(()); R197 -- 1 --> R200(()); R199 -- 0 --> R201(()); R199 -- 1 --> R202(()); R201 -- 0 --> R203(()); R201 -- 1 --> R204(()); R203 -- 0 --> R205(()); R203 -- 1 --> R206(()); R205 -- 0 --> R207(()); R205 -- 1 --> R208(()); R207 -- 0 --> R209(()); R207 -- 1 --> R210(()); R209 -- 0 --> R211(()); R209 -- 1 --> R212(()); R211 -- 0 --> R213(()); R211 -- 1 --> R214(()); R213 -- 0 --> R215(()); R213 -- 1 --> R216(()); R215 -- 0 --> R217(()); R215 -- 1 --> R218(()); R217 -- 0 --> R219(()); R217 -- 1 --> R220(()); R219 -- 0 --> R221(()); R219 -- 1 --> R222(()); R221 -- 0 --> R223(()); R221 -- 1 --> R224(()); R223 -- 0 --> R225(()); R223 -- 1 --> R226(()); R225 -- 0 --> R227(()); R225 -- 1 --> R228(()); R227 --
```

Uma árvore digital básica

- Algoritmos e Estruturas de Dados 2 - Prof. Mário César San Felice - Departamento de Computação - UFSCar**

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:

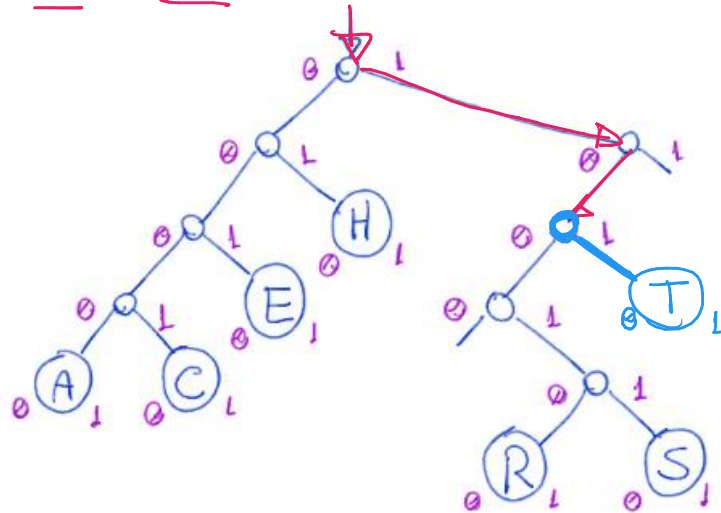
```
Noh *buscaR(Arvore r, Chave chave, int digito, Noh **ppai) {
 if (r == NULL)
 return r; // não encontrou
 if (r->esq == NULL && r->dir == NULL) { // eh uma folha
 if (r->chave == chave)
 return r; // encontrou
 return NULL; // não encontrou
 }
 if (pegaDigito(chave, digito) == 0) { // desce à esquerda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

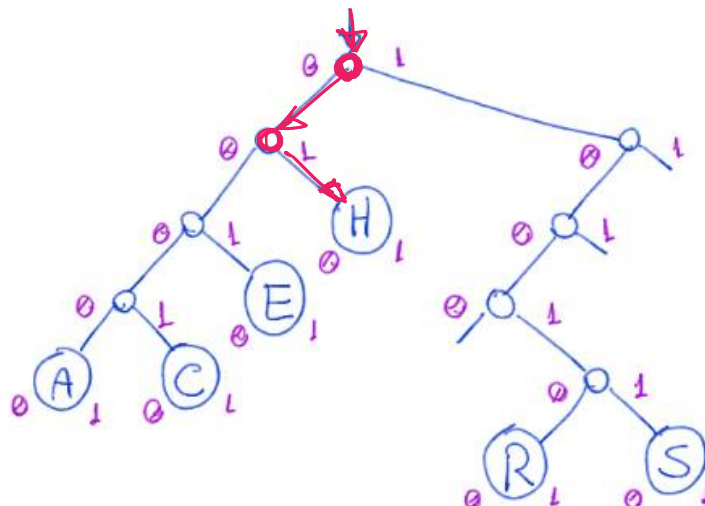
- Primeiro a chave T é buscada.



- 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

nível

nível

0

0

1

1

2

2

3

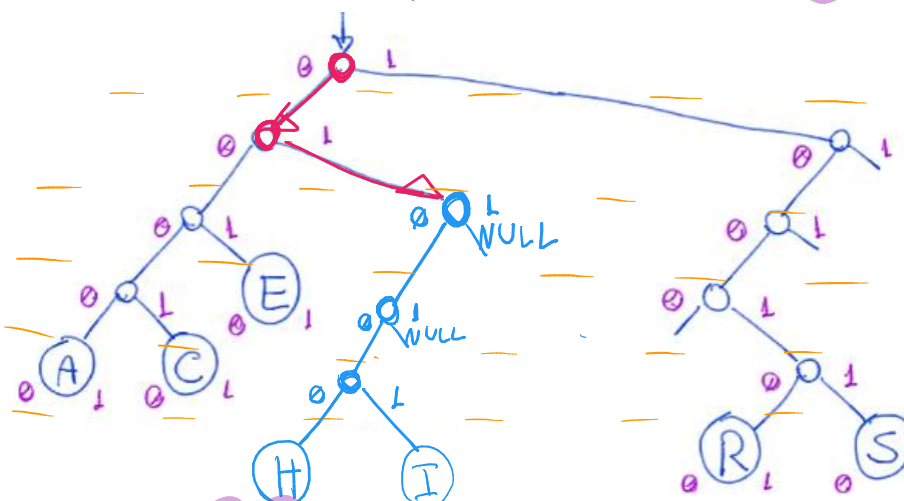
3

4

4

5

5



- 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 5º).

## 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);
}
```

*↑ null/digito*

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;
}
```

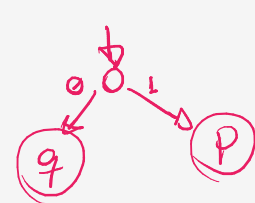
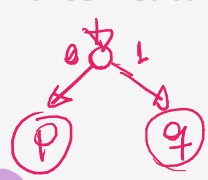
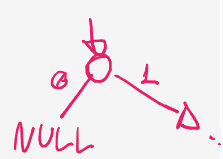
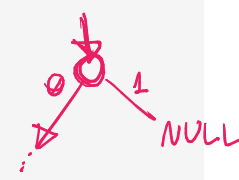
Função que insere recursivamente o novo nó na árvore

```
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.

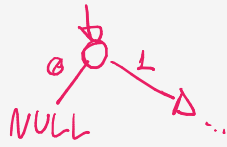
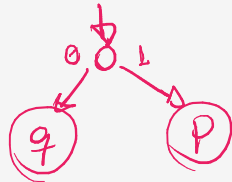
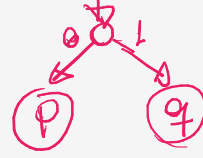
```
Arvore ramifique(Noh *p, Noh *q, int digito) {
 Noh *inter; // apontador para nó intermediário
 inter = (Noh *)malloc(sizeof(Noh));
 inter->chave = -1; // apenas para impressão
 if (pegaDigito(p->chave, digito) == pegaDigito(q->chave,
digito)) { // chaves não diferem no dígito atual
 if (pegaDigito(p->chave, digito) == 0) {
 // desce à esquerda do nó intermediário
 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,

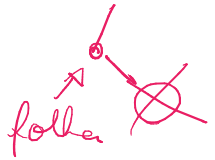
- que usa manipulação de bits e um switch para decidir o que fazer.
  - Inspirado no livro Algorithms in C++, Parts 1-4 de R. Sedgewick.

```
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 intermediário
 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 intermediário
 inter->dir = ramifique2(p, q, digito + 1);
 inter->esq = NULL;
 break;
 }
 return inter;
}
```



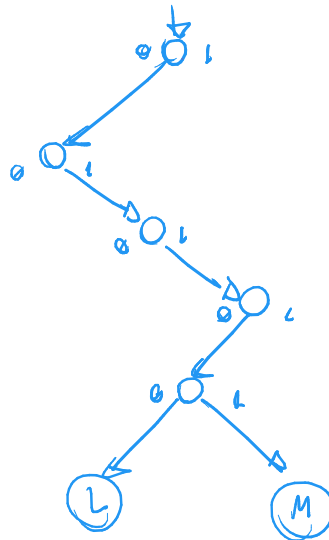
Para a operação de remoção

- 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,<sup>4</sup> ma 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.

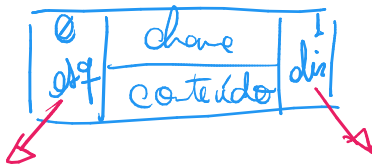
Curiosidade: podemos melhorar um pouco a eficiência de espaço das tries,

- 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.

nó trie binária



nó 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 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:

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

```
typedef int Item;
```

```
typedef char byte;
```

```
typedef byte *Chave;
```

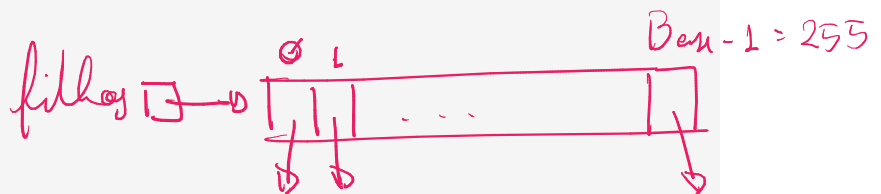
— são strings

```
const int bitsDigito = 8;
```

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

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

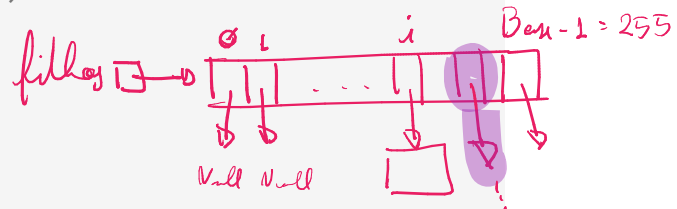
```
typedef Noh *Arvore;
```



```

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

```

```

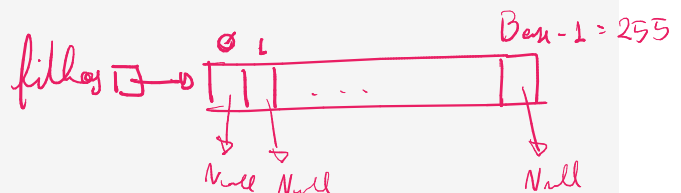
Noh *novoNoh(Chave chave, Item conteudo) {
 int i;
 Noh *novo;
 novo = (Noh *)malloc(sizeof(Noh));
 novo->chave = (char *)malloc((strlen(chave) + 1) *
sizeof(char));
}

```

```

 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;
}

```



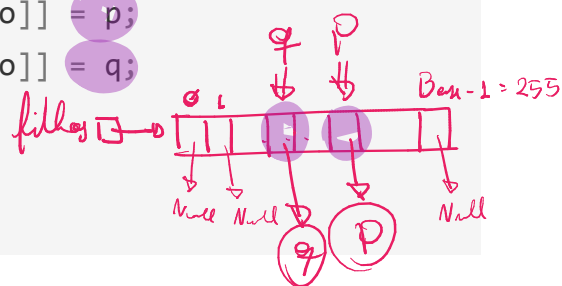
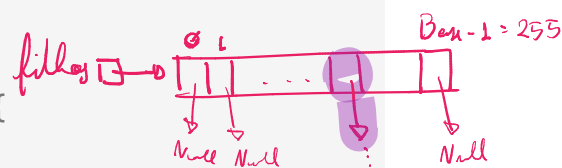
// Quiz4: como melhorar a eficiência dessa função?

```
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;
}
```

é folha?

➡ função sem chaves duplicadas

```
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.