

## AED2 - Aula 18

### Árvores de Busca Digital

Nós já estudamos diversos métodos de ordenação, em particular,

- vimos vários métodos baseados na comparação entre chaves,
- e o radix sort, baseado na análise de pedaços das chaves dos elementos.

Também já estudamos árvores de busca cujas operações

- são baseadas na comparação entre as chaves de seus elementos.

Agora vamos estudar árvores de busca digital, cujas operações

- são baseadas na análise de pedaços das chaves dos elementos.

Outros nomes usados para se referir às árvores digitais são

- árvore radix, árvore de prefixos e trie,
  - embora este último costume ser usado
    - para um tipo específico de árvore digital,
  - sendo que nesta aula iremos focar nas árvores digitais mais básicas.

A vantagem dessas árvores é que elas combinam

- implementação mais simples que das árvores balanceadas de busca,
- com tempo de acesso razoável no pior caso
  - e bastante eficiente na prática,
- sendo competitivos tanto com árvores balanceadas quanto com hash tables.

As desvantagens envolvem

- uso excessivo de memória,
  - o que pode ser contornado,
- e performance dependente do comprimento das chaves
  - e de métodos rápidos para acessar bytes e bits destas,
    - como acontece com o radix sort.

Exemplos de aplicação são roteadores e firewalls, que lidam com IPs.

Primeiro vamos estudar as árvores de busca digital binárias mais simples.

- Para tanto, usaremos a seguinte representação binária de caracteres.
- Consideramos que os bits são numerados, a partir do 0,
  - incrementalmente da esquerda para a direita.

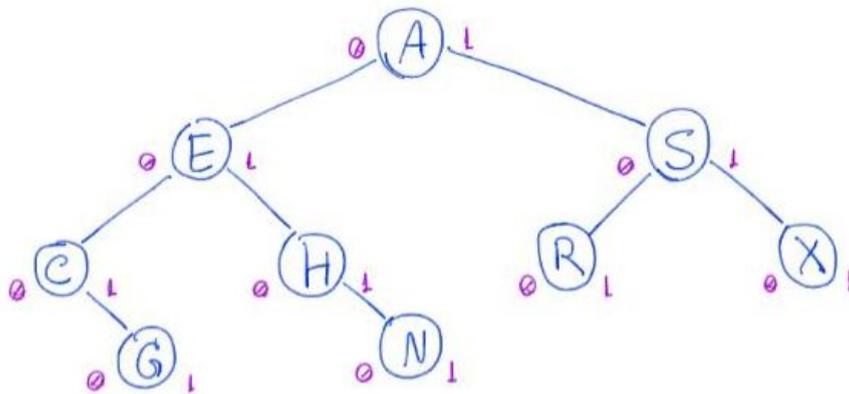
	A 00001	B 00010	C 00011
D 00100	E 00101	F 00110	G 00111

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	

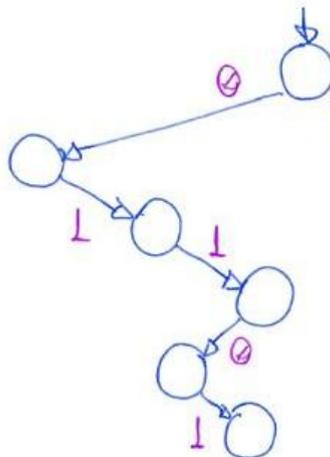
As operações de busca e inserção em árvores de busca digital binárias são

- muito parecidas com essas operações nas árvores binárias de busca.
- A diferença é que a escolha do lado para descer na árvore
  - não se dá pela comparação da chave buscada com a chave do nó,
    - mas pela análise de um bit da chave buscada.
- No caso, o bit considerado depende do nível de profundidade na árvore.

Exemplo de uma árvore de busca digital binária:



- Árvores de busca digital binárias são caracterizadas pela **propriedade**:
  - toda chave está em algum ponto do caminho definido pelos seus bits.
    - Exemplo: considere o caminho definido pela chave M (01101).

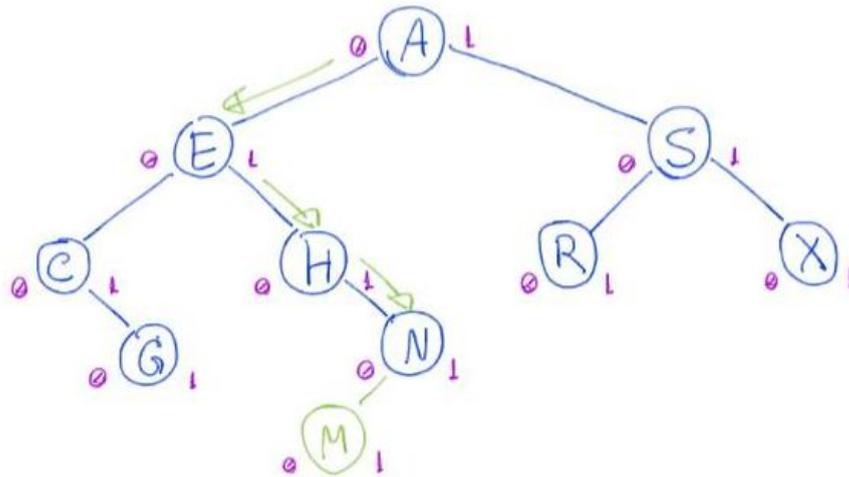


- Para simplificar, não trataremos do caso de chaves repetidas,
  - embora essas possam ser tratadas usando, por exemplo,

- listas encadeadas nos nós, como fazemos em Hash Tables.

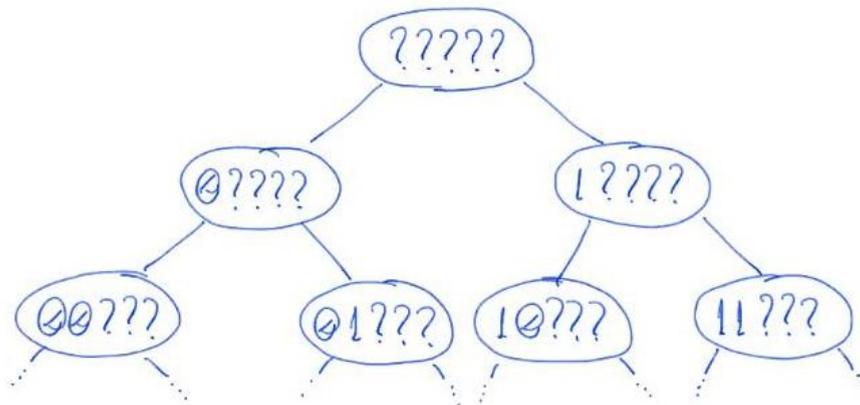
Inserção de M (01101) na árvore anterior:

- A seguir vemos a inserção de M, que ocupará
  - a primeira posição que estiver vazia na árvore,
    - ao percorrer seu caminho característico.



Como os caminhos são definidos pelos bits das chaves,

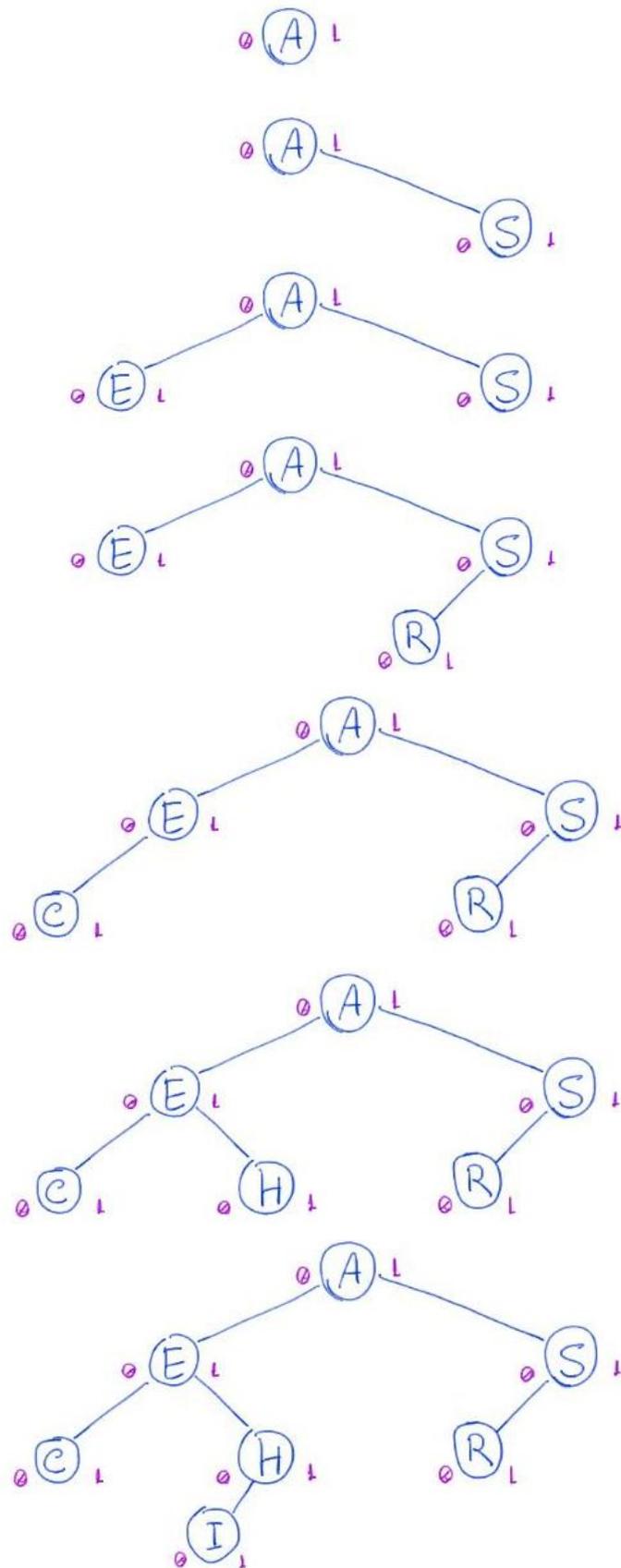
- podemos imaginar a seguinte árvore genérica para chaves de 5 bits,
  - em que ? indica que o bit pode ser 0 ou 1.

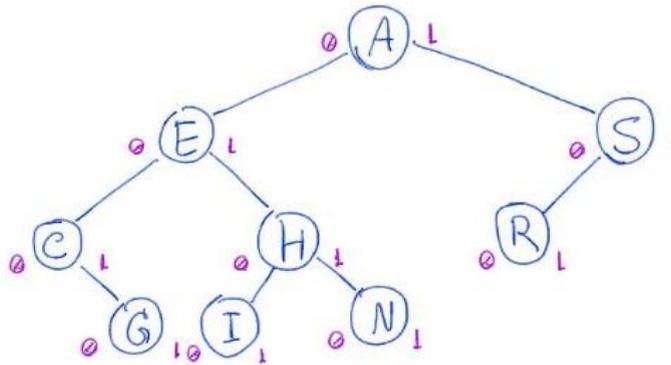
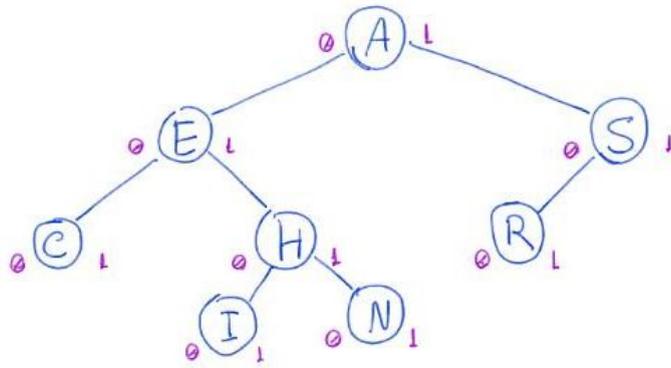


Resultado da construção de uma árvore de busca digital binária,

- pela inserção das chaves A S E R C H I N G em uma árvore vazia.
- Note que a árvore não mantém as chaves em ordem, ou seja,
  - não necessariamente chaves à esquerda do nó são menores que ele
    - ou as chaves à direita do nó são maiores que ele.
- Apesar disso, é curioso observar que chaves à esquerda de um nó
  - são menores que chaves à direita dele.
- Isso porque, toda chave em uma subárvore no nível k
  - tem os mesmo k bits iniciais.
- No entanto, chaves à esquerda do nó raiz da subárvore tem bit  $k = 0$ ,

- e chaves à direita tem bit  $k = 1$ , mas nada sabemos do bit  $k$  do nó raiz.





Destacamos que, nossas chaves tem comprimento constante de bitsPalavra bits.

- Assim, o número de elementos armazenados  $n \leq 2^{\text{bitsPalavra}}$ ,
  - já que supomos que não existem chaves repetidas.

Pior caso para a altura de uma árvore de busca digital binária:

- A altura máxima da árvore de busca digital binária é bitsPalavra,
  - ou seja, o comprimento da chave em bits.
- Isso porque, no caminho de uma chave na árvore
  - descemos um nível da árvore por bit da chave.
- Em geral, isso é muito melhor que o pior caso da árvore binária de busca,
  - que é da ordem do número de elementos  $n$ .

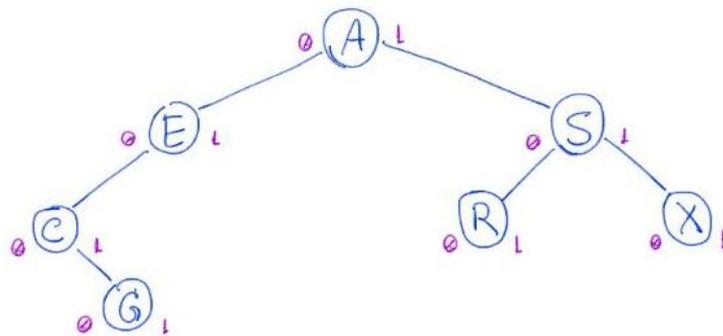


Em muitas situações, a altura da árvore é ainda menor.

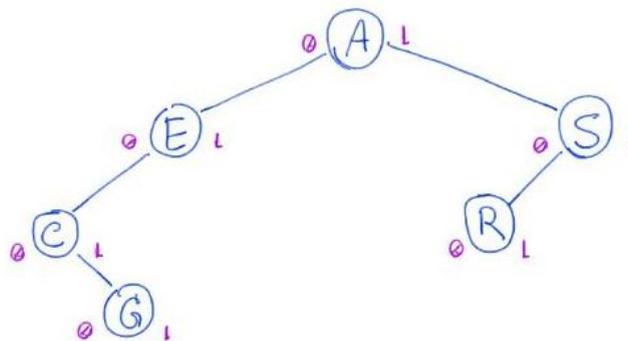
- Por exemplo, se as chaves forem aleatórias
  - a altura é da ordem de  $\lg n$ .
- A ideia por trás desse resultado é que, como as chaves são aleatórias,
  - quando focamos num bit  $b$  qualquer, é esperado que
    - metade das chaves tenha  $b = 0$  e a outra metade tenha  $b = 1$ .
  - Assim, a cada nível que descemos na árvore, esperamos
    - dividir por dois o número de chaves na subárvore corrente.
- Note que, como  $n \leq 2^{\text{bitsPalavra}}$  temos  $\lg n \leq \text{bitsPalavra}$ .

Remoção em uma árvore de busca digital binária:

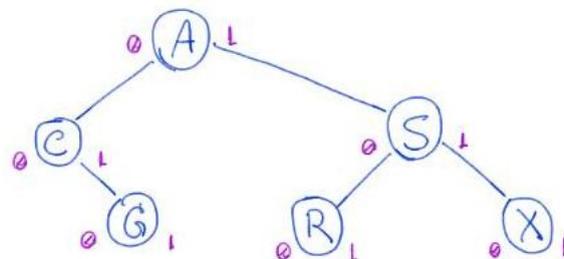
- A princípio, as ideias usadas na árvore binária de busca parecem funcionar.



- Isto é, se o nó é uma folha, basta removê-lo.
  - Ex.: remoção do X (11000).



- Agora, se o nó tem apenas um filho, bastaria
  - conectar esse filho ao pai e remover o nó.
  - Ex.: remoção incorreta de E (00101).



- Note que, a remoção anterior é incorreta, pois



```
return (int)((chave >>
    (bitsDigito * (digitosPalavra - 1 - digito))) & (Base - 1));
}
```

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

```
typedef Noh *Arvore;
```

```
Noh *buscaR(Arvore r, Chave chave, int digito, Noh **ppai)
{
    if (r == NULL)
        return r;
    if (r->chave == chave)
        return r;
    if (pegaDigito(chave, digito) == 0)
    {
        *ppai = r;
        return buscaR(r->esq, chave, digito + 1, ppai);
    }
    // pegaDigito(chave, digito) == 1
    *ppai = r;
    return buscaR(r->dir, chave, digito + 1, ppai);
}
```

```
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;
}

```

Arvore **insereR**(Arvore r, Noh \*novo, int digito)

```

{
    if (r == NULL)
    {
        return novo;
    }
    if (pegaDigito(novo->chave, digito) == 0)
    {
        r->esq = insereR(r->esq, novo, digito + 1);
    }
    else // pegaDigito(novo->chave, digito) == 1
    {
        r->dir = insereR(r->dir, novo, digito + 1);
    }
    return r;
}

```

Arvore **inserir**(Arvore r, Chave chave, Item conteudo)

```

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

```

*// remove alvo e devolve a nova raiz da subárvore*

Arvore **removeRaiz**(Arvore alvo)

```

{
    Noh *aux = NULL, *pai = NULL;
    // se eh folha
    if (alvo->esq == NULL && alvo->dir == NULL)

```

```

{
    free(alvo);
    return NULL;
}
// se nao eh folha
aux = alvo;
while (aux->dir != NULL || aux->esq != NULL)
{
    pai = aux;
    if (aux->dir != NULL)
        aux = aux->dir;
    else
        aux = aux->esq;
}
// aux chegou numa folha
alvo->chave = aux->chave;
alvo->conteudo = aux->conteudo;
if (pai->esq == aux)
    pai->esq = removeRaiz(aux);
else // pai->dir == aux
    pai->dir = removeRaiz(aux);
return alvo;
}

```

Arvore **remover**(Arvore r, Chave chave)

```

{
    Noh *alvo, *aux, *pai = NULL;
    alvo = buscaR(r, chave, 0, &pai);
    if (alvo == NULL)
    {
        // printf("Nao achou\n");
        return r;
    }
    aux = removeRaiz(alvo);
    if (alvo == r) // removeu a raiz da arvore

```

```

    return aux;
if (pai->esq == alvo)
    pai->esq = aux;
if (pai->dir == alvo)
    pai->dir = aux;
return r;
}

```

Códigos para árvore de busca digital geral:

```

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

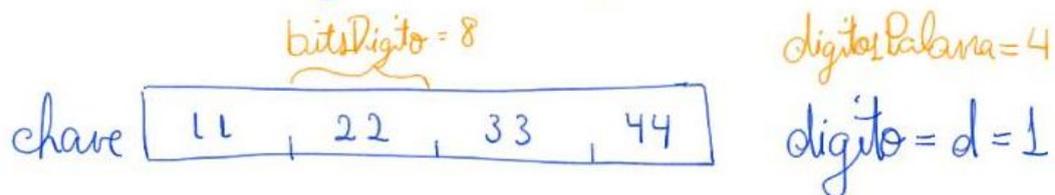
typedef int Item;
typedef int Chave;

const int bitsPalavra = 32;
const int bitsDigito = 8;
const int digitosPalavra = bitsPalavra / bitsDigito;
const int Base = 1 << bitsDigito; // Base = 2^bitsDigito

int pegaDigito(int chave, int digito)
{
    return (int)((chave >>
(bitsDigito * (digitosPalavra - 1 - digito))) & (Base - 1));
}

```

- Calcular dígito com operações em bits



$$\text{digitosPalavra} - 1 - d = 4 - 1 - 1 = 2$$

$$\text{bitsDigito} * (\text{digitosPalavra} - 1 - d) = 8 * 2 = 16$$

$$\text{chave} \gg (\text{bitsDigito} * (\text{digitosPalavra} - 1 - d)) = \text{chave} \gg 16$$



$$\text{Base} = 1 \ll \text{bitsDigito} = 1 \ll 8$$



$$\text{chave} \gg (\text{bitsDigito} * (\text{digitosPalavra} - 1 - d)) \& (\text{Base} - 1)$$



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

```
typedef Noh *Arvore;
```

```
Noh *buscaR(Arvore r, Chave chave, int digito, Noh **ppai)
{
    if (r == NULL)
        return r;
    if (r->chave == chave)
```

```

        return r;
    *ppai = r;
    return buscaR(r->filhos[pegaDigito(chave, digito)], chave,
digito + 1, ppai);
}

```

Noh **\*novoNoh**(Chave **chave**, Item **conteudo**)

```

{
    int i;
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = chave;
    novo->conteudo = conteudo;
    novo->filhos = malloc(Base * sizeof(Noh *));
    for (i = 0; i < Base; i++)
        novo->filhos[i] = NULL;
    return novo;
}

```

Arvore **insereR**(Arvore **r**, Noh **\*novo**, int **digito**)

```

{
    int i;
    if (r == NULL)
    {
        return novo;
    }
    i = pegaDigito(novo->chave, digito);
    r->filhos[i] = insereR(r->filhos[i], novo, digito + 1);
    return r;
}

```

Arvore **inserir**(Arvore **r**, Chave **chave**, Item **conteudo**)

```

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

```

```

}

// remove alvo e devolve a nova raiz da subárvore
Arvore removeRaiz(Arvore alvo)
{
    Noh *aux = NULL, *pai = NULL;
    int i, iaux;
    for (i = 0; i < Base; i++)
        if (alvo->filhos[i] != NULL)
            break;
    // se eh folha
    if (i == Base)
    {
        free(alvo->filhos);
        free(alvo);
        return NULL;
    }
    // se nao eh folha
    aux = alvo;
    while (i < Base)
    {
        pai = aux;
        aux = pai->filhos[i];
        iaux = i;
        for (i = 0; i < Base; i++)
            if (aux->filhos[i] != NULL)
                break;
    }
    // aux chegou numa folha
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
    pai->filhos[iaux] = removeRaiz(aux);
    return alvo;
}

```

```

Arvore remove(Arvore r, Chave chave)
{
    Noh *alvo, *aux, *pai = NULL;
    int i;
    alvo = buscaR(r, chave, 0, &pai);
    if (alvo == NULL)
    {
        // printf("Nao achou\n");
        return r;
    }
    aux = removeRaiz(alvo);
    if (alvo == r) // removeu a raiz da árvore
        return aux;
    for (i = 0; i < Base; i++)
        if (pai->filhos[i] == alvo)
            break;
    pai->filhos[i] = aux;
    return r;
}

```

Códigos para árvore de busca digital strings:

```

#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;

void imprimeSimbolos(int n, char c)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%c", c);
}

Noh *buscaR(Arvore r, Chave chave, int digito, Noh **ppai)
{
    if (r == NULL)
        return r;
    if (strcmp(r->chave, chave) == 0)
        return r;
    *ppai = r;
    return buscaR(r->filhos[(int)chave[digito]], chave, digito + 1,
ppai);
}

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;
}

Arvore insereR(Arvore r, Noh *novo, int digito)
{
    int i;
    if (r == NULL)
    {
        return novo;
    }
    i = (int)(novo->chave[digito]);
    r->filhos[i] = insereR(r->filhos[i], novo, digito + 1);
    return r;
}

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

// remove alvo e devolve a nova raiz da subárvore
Arvore removeRaiz(Arvore alvo)
{
    Noh *aux = NULL, *pai = NULL;
    int i,iaux;
    for (i = 0; i < Base; i++)
        if (alvo->filhos[i] != NULL)
            break;
    // se eh folha
    if (i == Base)
    {
        free(alvo->chave);
        free(alvo->filhos);
    }
}

```

```

    free(alvo);
    return NULL;
}
// se nao eh folha
aux = alvo;
while (i < Base)
{
    pai = aux;
    aux = pai->filhos[i];
   iaux = i;
    for (i = 0; i < Base; i++)
        if (aux->filhos[i] != NULL)
            break;
}
// aux chegou numa folha
strcpy(alvo->chave, aux->chave);
alvo->conteudo = aux->conteudo;
pai->filhos[iaux] = removeRaiz(aux);
return alvo;
}

```

Arvore **remover**(Arvore r, Chave chave)

```

{
    Noh *alvo, *aux, *pai = NULL;
    int i;
    alvo = buscaR(r, chave, 0, &pai);
    if (alvo == NULL)
    {
        // printf("Nao achou\n");
        return r;
    }
    aux = removeRaiz(alvo);
    if (alvo == r) // removeu a raiz da arvore
        return aux;
    for (i = 0; i < Base; i++)

```

```

    if (pai->filhos[i] == alvo)
        break;
    pai->filhos[i] = aux;
    return r;
}

```

Eficiência de tempo das operações:

- Lembre que a eficiência é proporcional à altura da árvore.
- No pior caso é da ordem do comprimento/número de dígitos da chave,
  - i.e.,  $O(\text{digitosPalavra}) = O(\text{bitsPalavra} / \text{bitsDigito})$ .
- Note que, em geral isso é bastante bom, já que para
  - bitsDigito = 1 (Base = 2) e bitsPalavra = 32, temos
    - digitosPalavra = 32
  - Ou ainda, para bitsDigito = 8 (Base = 256) e digitosPalavra = 64,
    - temos digitosPalavra =  $64 / 8 = 8$
- Eficiência esperada das operações caso as chaves sejam aleatórias
  - é proporcional a  $\log_{\text{Base}}$  do número de elementos, i.e.,  $O(\log n)$ .

Conclusões:

- Árvores de busca digital são interessantes em muitas aplicações,
  - por combinarem eficiência (e balanceamento)
    - comparável a árvores balanceadas, i.e., AVL, rubro negras,
  - com implementações muito mais simples.
- No entanto, elas não têm algumas propriedades de árvores de busca.
  - Por exemplo, elas não mantêm as chaves em ordem, o que complica operações como sucessor, predecessor, percurso ordenado, etc.
  - Curiosamente, as operações máximo e mínimo,
    - que também estão relacionadas com ordem das chaves,
      - podem ser implementadas eficientemente.
        - Como? Por que?
- Além disso, elas tem problema ao lidar com chaves de comprimento variável.
  - Por que?
- Uma curiosidade, as árvores de busca digital também funcionam
  - considerando os dígitos dos menos significativos para os mais,
    - i.e., da direita para a esquerda.
- Isto pode ser vantajoso se as chaves diferem
  - principalmente nos dígitos menos significativos.

Na próxima aula veremos as Tries,

- um tipo mais sofisticado de árvore digital de busca,
  - que visa superar algumas dessas desvantagens.