AED2 - Aula ~~18~~16
Árvores de Busca Digital

Nós já estudamos diversos métodos de ordenação. Em particular,
- vimos vários métodos baseados na comparação entre chaves,
- e outros baseados no valor 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 no valor de pedaços das chaves dos elementos,
    - lembrando a abordagem do radixSort.

Outros nomes usados para se referir às árvores digitais são
- árvore radix, árvore de prefixos e trie,
    - embora o último seja usado para um tipo específico de árvore digital.
- Nesta aula vamos 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 radixSort.

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

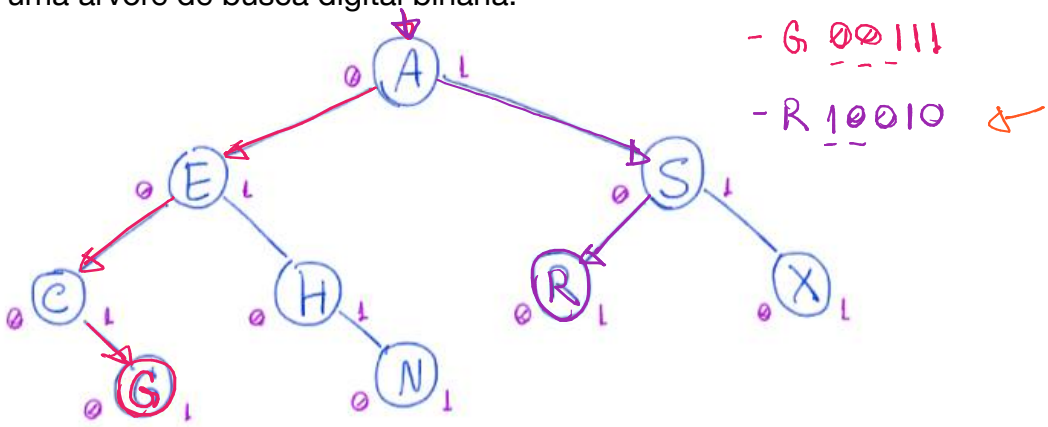
Primeiro vamos estudar as árvores de busca digital binárias mais simples.
- Para tanto, usaremos a seguinte representação binária de caracteres.
- Consideramos que os bits são numerados, a partir do 0,
    - incrementalmente da esquerda para a direita.

	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	

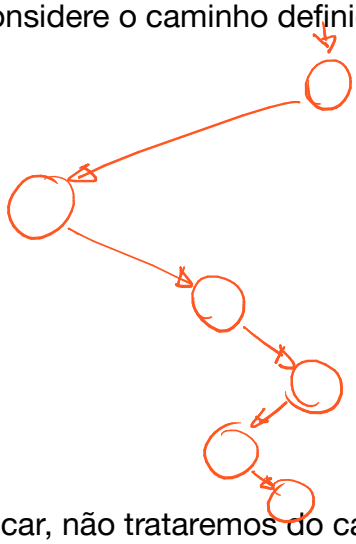
As operações de busca e inserção em árvores de busca digital binárias

- 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:



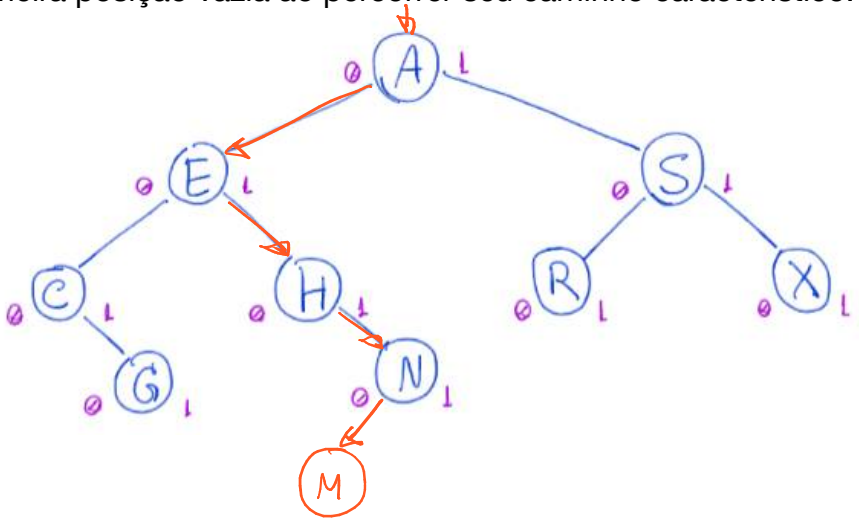
- Árvores de busca digital binárias são caracterizadas pela **propriedade**:
    - toda chave está em algum ponto do caminho definido pelos seus bits.
- Exemplo: considere o caminho definido pela chave M (01101).



- 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.

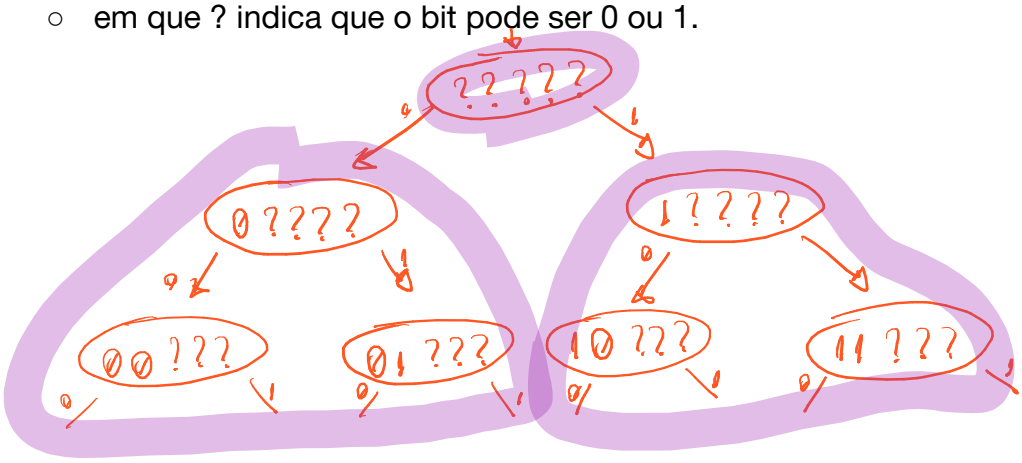
Inserção de M (01101) na árvore anterior: A inserção de M ocupará

- a primeira posição vazia ao percorrer seu caminho característico.



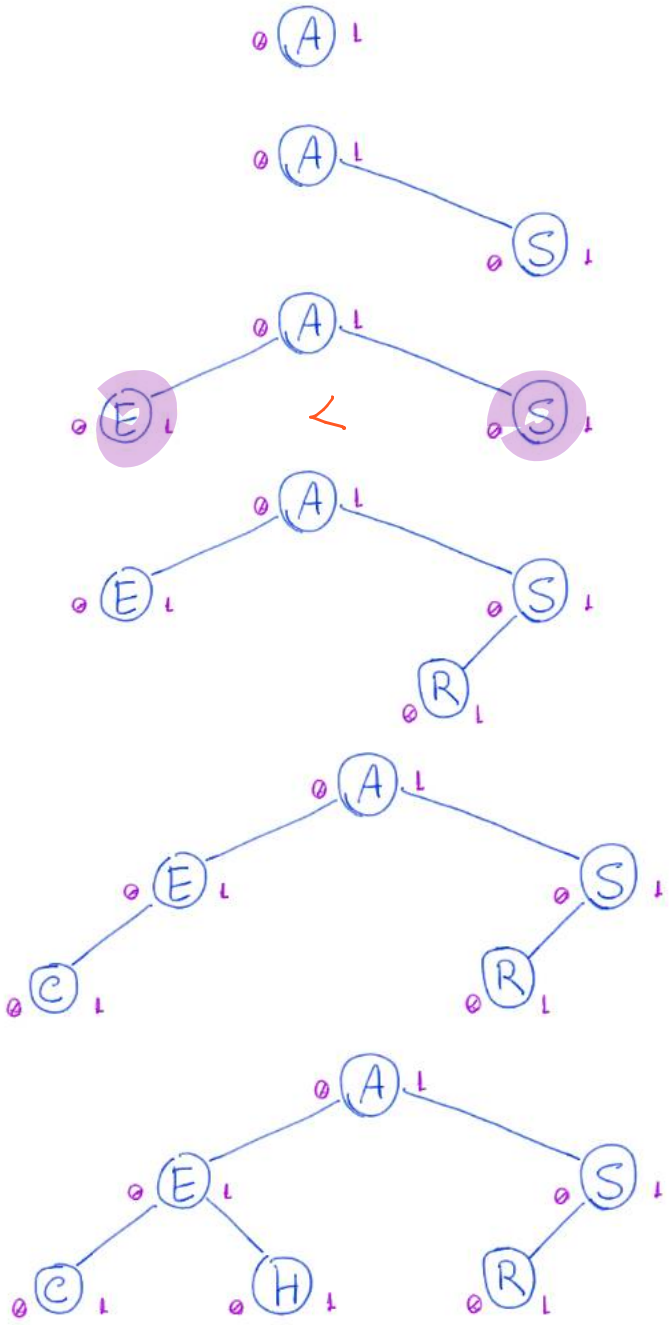
Como os caminhos são definidos pelos bits das chaves,

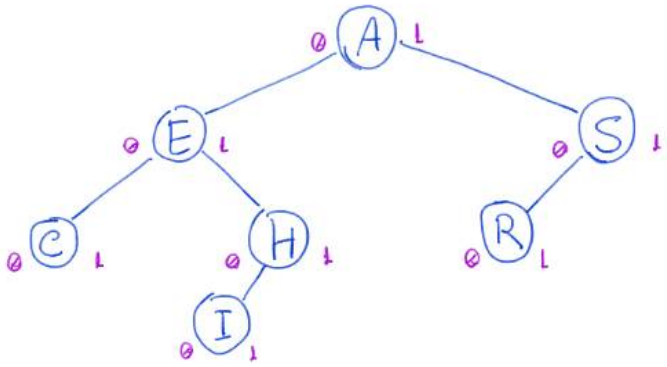
- podemos imaginar a seguinte árvore genérica para chaves de 5 bits,
    - em que ? indica que o bit pode ser 0 ou 1.

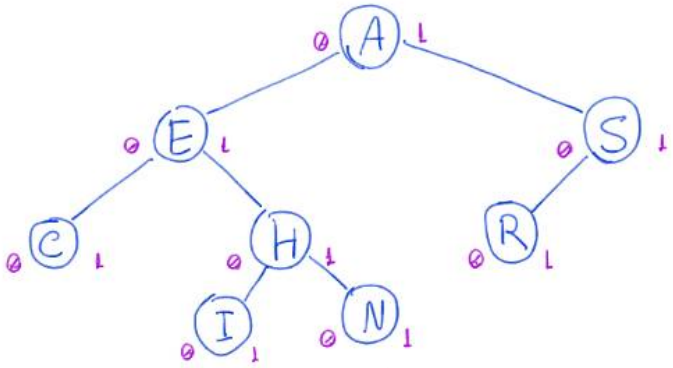


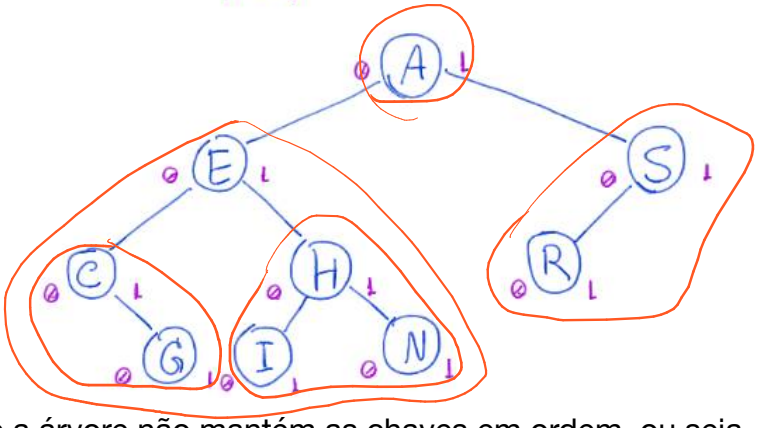
A seguir, temos o resultado da construção de uma árvore de busca digital binária,

- pela inserção das chaves A S E R C H I N G em uma árvore vazia.

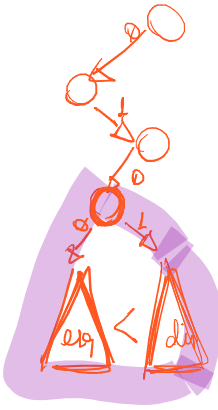








- Note que a árvore não mantém as chaves em ordem, ou seja,
    - não necessariamente chaves à esquerda do nó são menores que ele
        - ou as chaves à direita do nó são maiores que ele.
- Apesar disso, é curioso observar que chaves à esquerda de um nó
    - são menores que chaves à direita dele.
- Isso porque, toda chave em uma subárvore no nível k
    - tem os mesmo k bits iniciais.
- Além disso, 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.

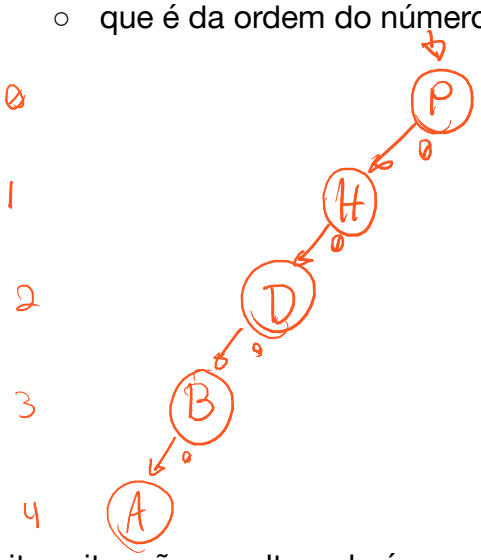


Destacamos que, nossas chaves tem comprimento constante de bitsPalavra bits.

- Assim, o número de elementos armazenados n <= 2^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.



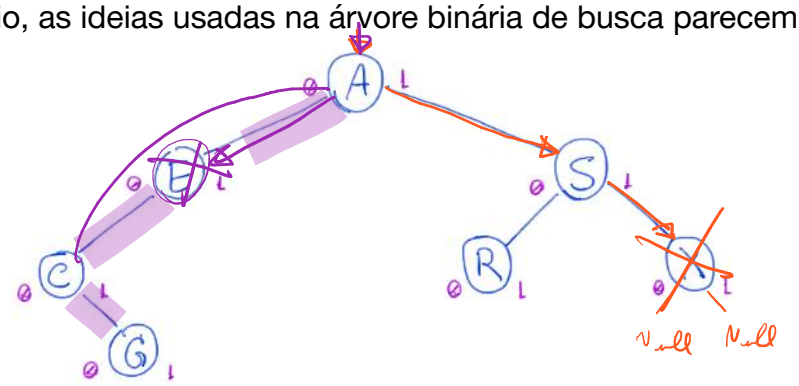
P 10000
H 01000
D 00100
B 00010
A 00001

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

- Por exemplo, se as chaves forem aleatórias
    - a altura esperada 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 <= 2^bitsPalavra temos $\lg n \leq$ bitsPalavra

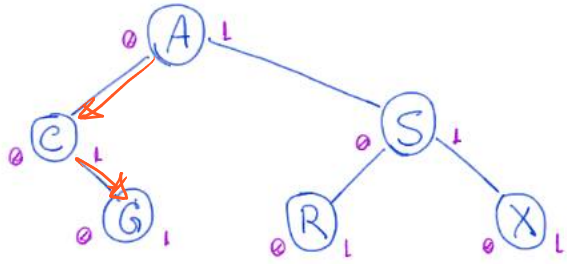
Remoção em uma árvore de busca digital binária:

- 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).

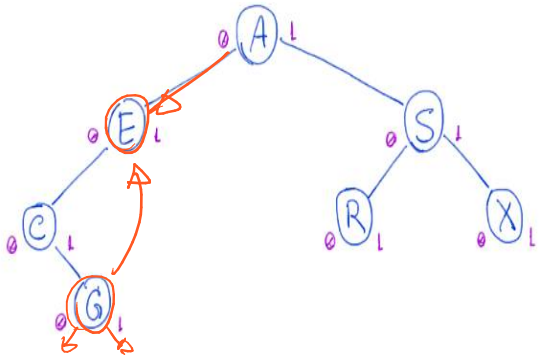
G 00111

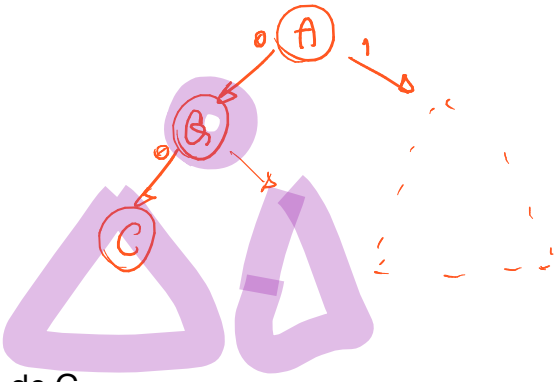


- Note que, a remoção anterior é incorreta, pois o caminho 01 até G
  - não corresponde a um prefixo da representação binária 00111 de G.
- Por isso, a árvore resultante não é uma árvore de busca digital binária válida.

Para corrigir a remoção em árvores de busca digital binárias,

- no caso em que o nó a ser removido tem algum filho,
  - este deve ser substituído por algum de seus descendentes
    - que seja uma folha,
  - e então essa folha deve ser removida.
- Essa operação é segura porque todo descendente do nó
  - tem uma chave com o mesmo prefixo que a chave dele.
- Ex.: remoção correta de E (00101).





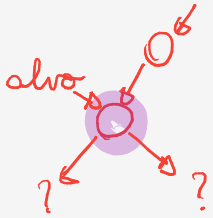
- Note que, o único caminho alterado foi o de G,
  - que encurtou de 001 para 0, sem deixar de ser
    - um prefixo da representação binária 00111 de G.
- Portanto, a árvore resultante
  - continua sendo uma árvore de busca digital binária válida.

Códigos para árvore de busca binária digital:

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

typedef int Item;
typedef int Chave;

const int bitsPalavra = 32;
const int bitsDigito = 1;
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));
}

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);
}
```

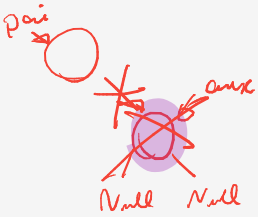
```c
Noh *novoNoh(Chave chave, Item conteudo) {
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = chave;
    novo->conteudo = conteudo;
    novo->esq = NULL;
    novo->dir = NULL;
    return novo;
}

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);
}
```

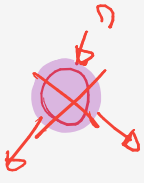
```c
// remove alvo e devolve a nova raiz da subárvore
Arvore removeRaiz(Arvore alvo) {
    Noh *aux = NULL, *pai = NULL;
    if (alvo->esq == NULL && alvo->dir == NULL) { // se eh folha
        free(alvo);
        return NULL;
    }
    // se nao eh folha
    aux = alvo;
    while (aux->dir != NULL || 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) // não achou
        return r;
    aux = removeRaiz(alvo);
    if (alvo == r) // removeu a raiz da árvore
        return aux;
    if (pai->esq == alvo)
        pai->esq = aux;
    if (pai->dir == alvo)
        pai->dir = aux;
    return r;
}
```



desce na árvore enquanto aux não for folha

substitui alvo por aux

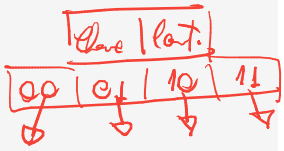




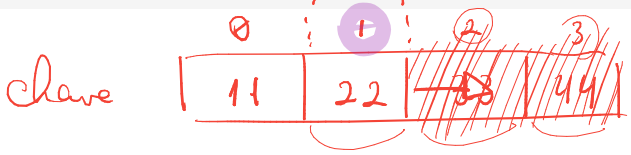
Códigos para árvore de busca digital geral:

```c
#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));
}
```

bitsDigito = 2 ⇒ Base = $2^2 = 4$



bitsDigito = 8



digito = d = 1

(digitosPalavra - 1 - d) = 4 - 1 - 1 = 2

bitsDigito * (digitosPalavra - 1 - d) = 8 . 2 = 16

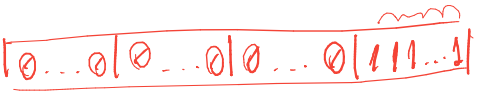
chave >> (bitsDigito * (digitosPalavra - 1 - d)) = chave >> 16

⇒

Base = 1 << bitsDigito = 1 << 8

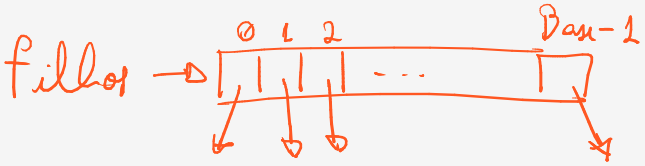
Base

⇒ (Base - 1)



(chave >> (bitsDigito * (digitosPalavra - 1 - d))) & (Base - 1)

```c
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);
}
```

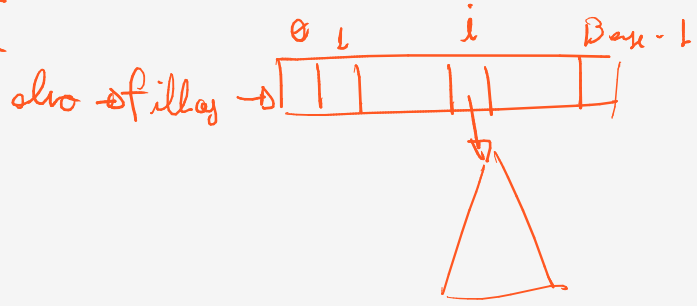


aloco o vetor de filhos

inicializando todos os filhos com NULL

```c
// 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;
    if (i == Base) { // se eh folha
        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 remover(Arvore r, Chave chave) {
    Noh *alvo, *aux, *pai = NULL;
    int i;
    alvo = buscaR(r, chave, 0, &pai);
    if (alvo == NULL) // não achou
        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;
}
```

verifica se alvo tem filho



verifica se aux é folha

substitui alvo por aux

Códigos para árvore de busca digital strings:

```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;

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;
}
```

```c
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);
}

Arvore removeRaiz(Arvore alvo) {
    Noh *aux = NULL, *pai = NULL;
    int i, iaux;
    for (i = 0; i < Base; i++)
        if (alvo->filhos[i] != NULL) break;
    if (i == Base) { // se eh folha
        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) return r; // não achou
    aux = removeRaiz(alvo);
    if (alvo == r) return aux; // removeu a raiz da árvore
    for (i = 0; i < Base; i++)
        if (pai->filhos[i] == alvo) break;
    pai->filhos[i] = aux;
    return r;
}
```

Eficiência de tempo das operações: a eficiência é proporcional à altura da árvore.

- No pior caso ela é da ordem do número de dígitos da chave,
  - i.e., O(digitosPalavra) = O(bitsPalavra / 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_Base do número de elementos, i.e., O(log n).
- No entanto, quando bitsDigito cresce, Base cresce exponencialmente
  - bem como o tamanho dos nós e o consumo de memória.
- Quiz1: se um ponteiro usa 4 bytes e bitsDigito = 8, quanto ocupa um nó?

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 mantém 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.
  - Quiz2: como implementá-las em árvores de busca digital?
- Árvores digitais tem problema ao lidar com chaves de comprimento variável.
  - Quiz3: por que? Dica: pense no caminho característico das chaves.
- 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.
- Isso pode ser vantajoso se as chaves diferem principalmente nos dígitos menos significativos.

Outros tipos de árvores de busca digital, como Tries e PATRICIA Tries,

- buscam superar algumas dessas limitações.