AED2 - Aula 19
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 de métodos rápidos
    - para acessar bytes e bits das chaves,
        - 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.

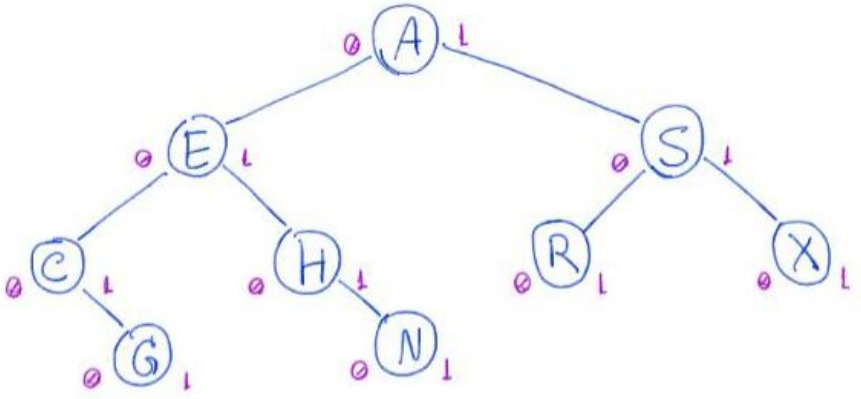
	A 00001	B 00010	C 00011
D 00100	E 00101	F 00110	G 00111

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	Z 11001		

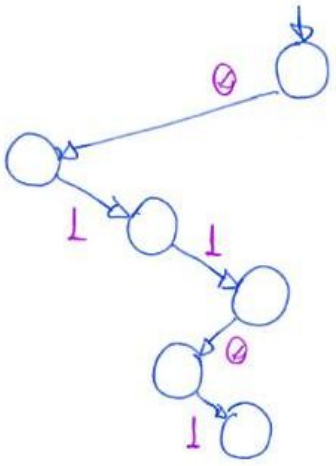
As operações de busca e inserção em árvores de busca digital binárias são

- 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).

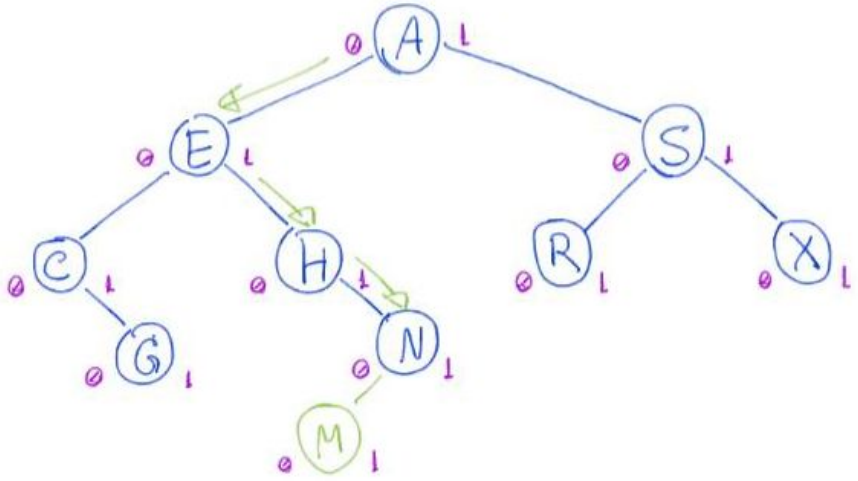


- Para simplificar, não trataremos do caso de chaves repetidas,
    - embora essas possam ser tratadas usando, por exemplo,

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

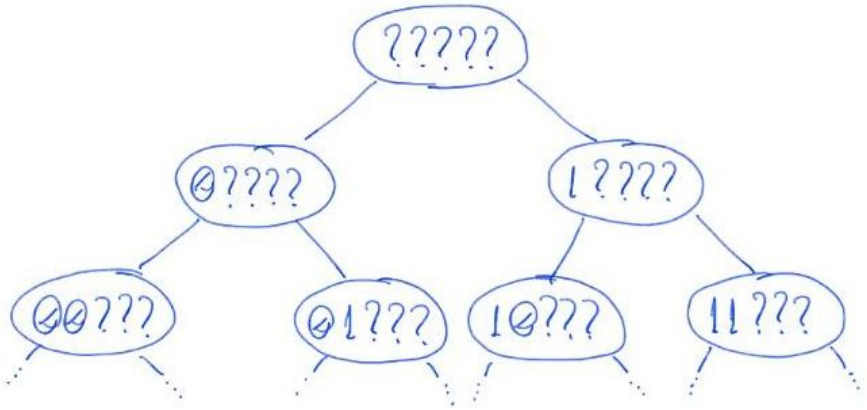
Inserção de M (01101) na árvore anterior:

- 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,

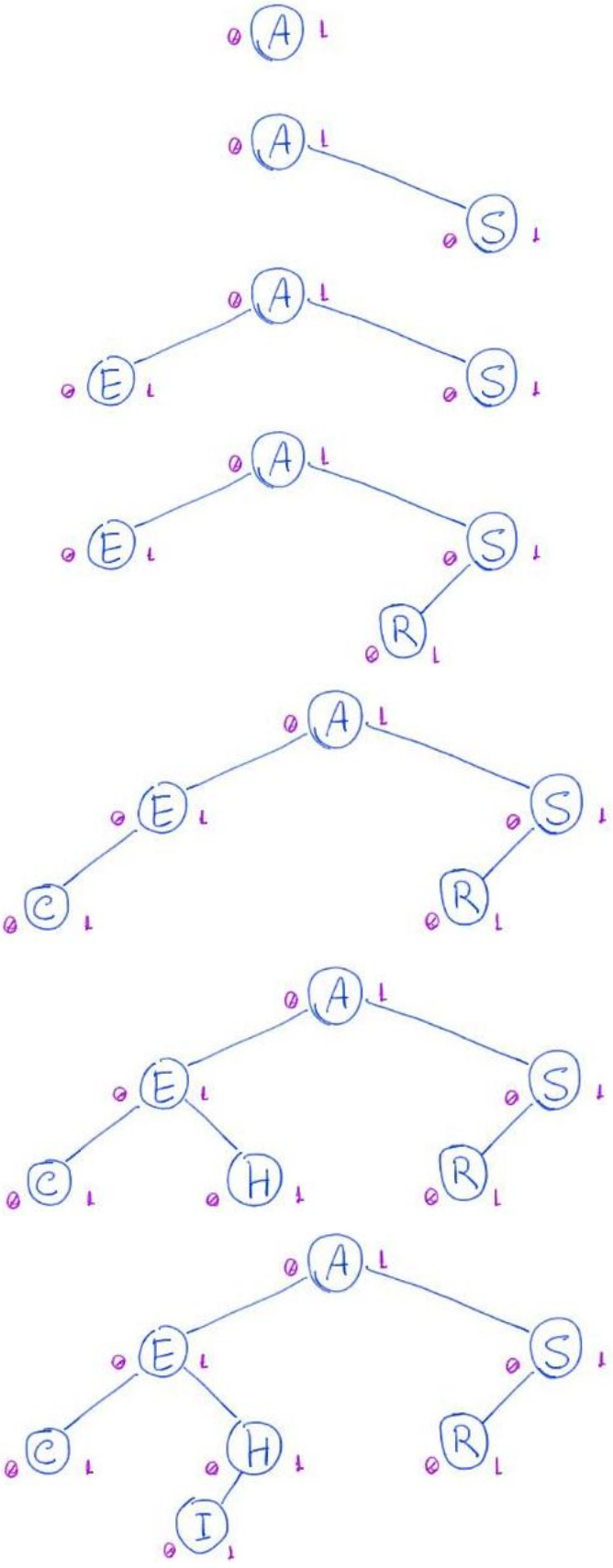
- podemos imaginar a seguinte árvore genérica para chaves de 5 bits,
    - em que ? indica que o bit pode ser 0 ou 1.

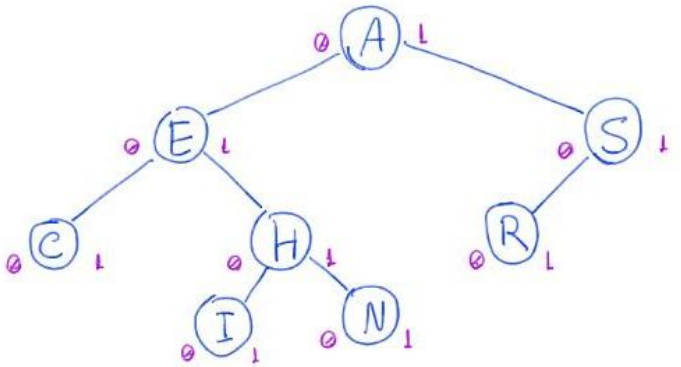


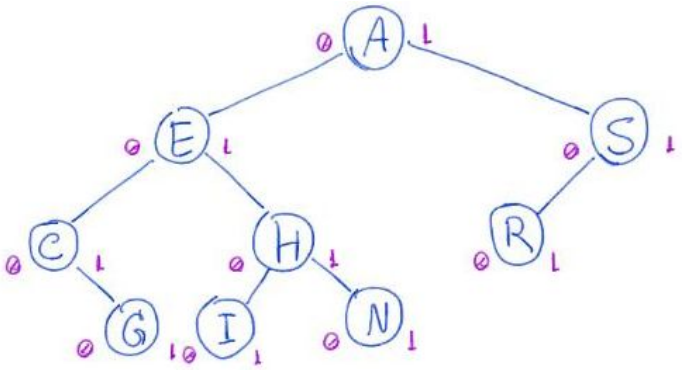
Resultado da construção de uma árvore de busca digital binária,

- 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 sabemos sobre o bit k do nó raiz.







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

- Assim, o número de elementos armazenados n <= 2^W,
    - já que supomos que não existem chaves repetidas.

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

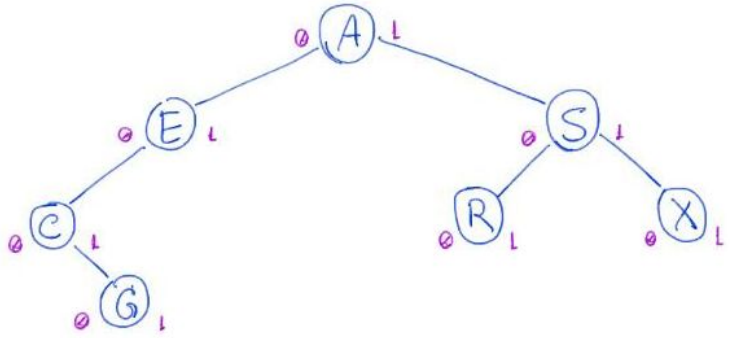
- A altura máxima da árvore de busca digital binária é W,
    - ou seja, o comprimento da chave em bits.
- Isso porque, no caminho de uma chave na árvore
    - descemos um nível da árvore por bit da chave.
- Em geral, isso é muito melhor que o pior caso da árvore binária de busca,
    - que é da ordem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

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

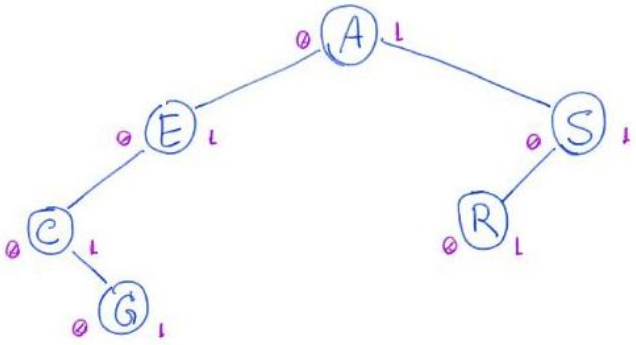
- Por exemplo, se as chaves forem aleatórias
    - a altura é da ordem de lg n.
- A ideia por trás desse resultado é que, como as chaves são aleatórias,
    - quando focamos num bit b qualquer, é esperado que
        - metade das chaves tenha b = 0 e a outra metade tenha b = 1.
    - Assim, a cada nível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

Remoção em uma árvore de busca digital binária:

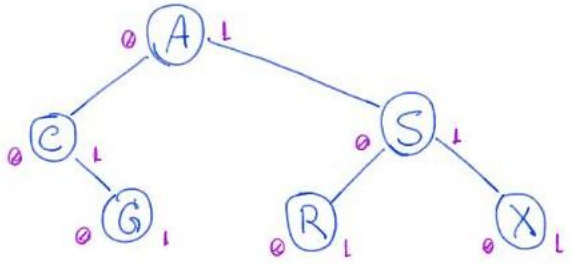
- a princípio, as ideias usadas na árvore binária de busca parecem funcionar.



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

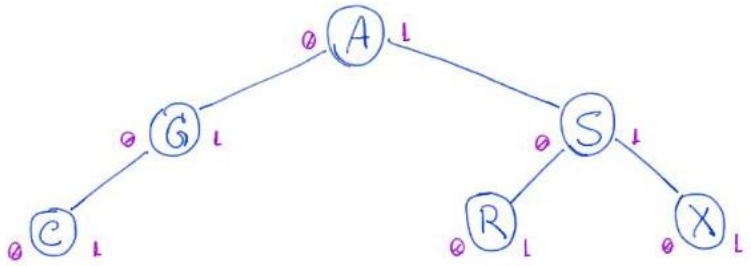


- Agora, se o nó tem apenas um filho, bastaria
    - conectar esse filho ao pai e remover o nó.
    - Ex.: remoção incorreta de E (00101).



    - Note que, a remoção anterior é incorreta, pois

        - o caminho 01 até G não corresponde mais
            - 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).



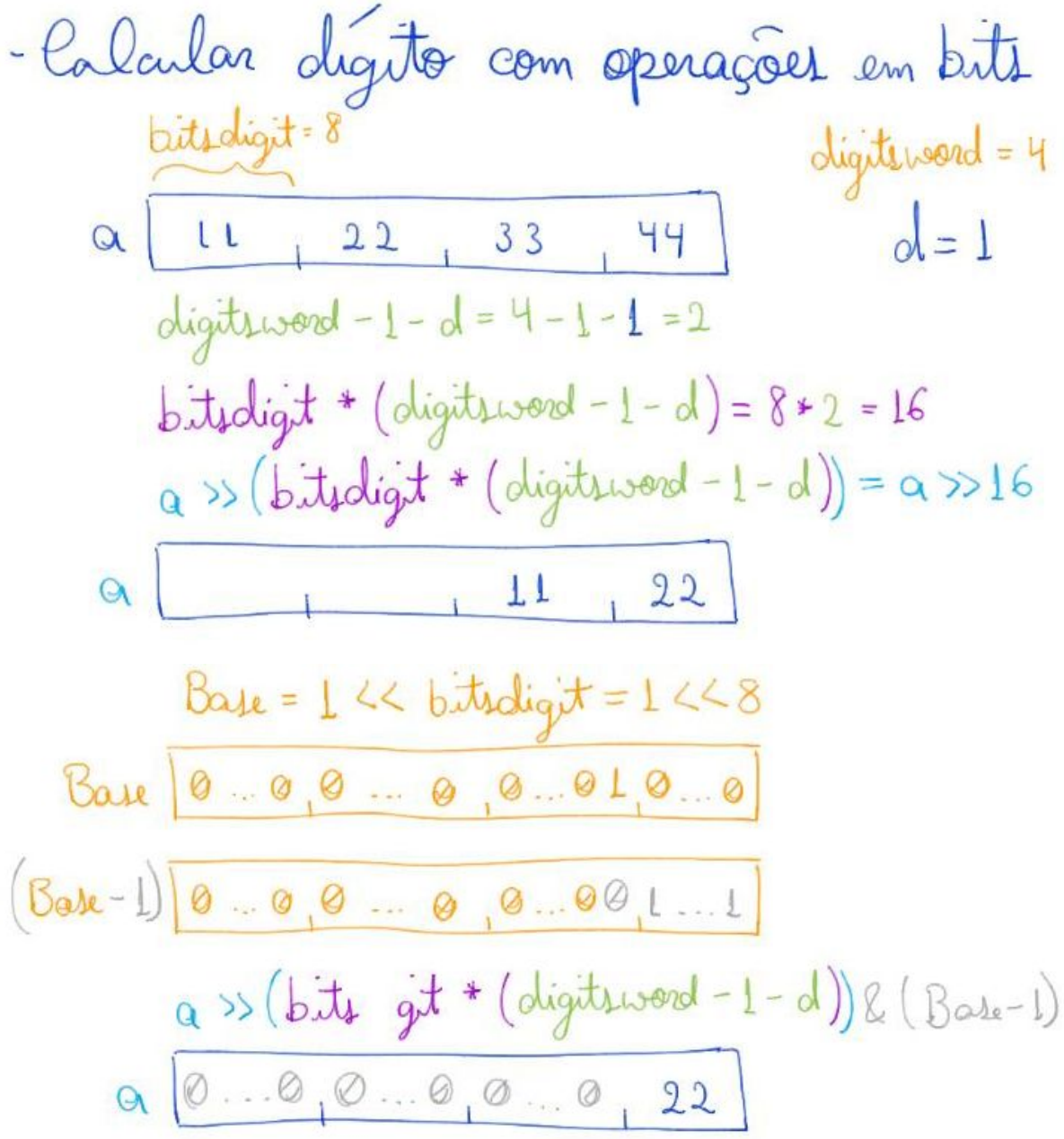
    - Note que, o único caminho alterado foi o de G,
        - que encurtou de 001 para 0, sem deixar de ser
            - um prefixo da representação binária 00111 de G.
    - Portanto, a árvore resultante
        - continua sendo uma árvore de busca digital binária válida.

Códigos para árvore de busca binária digital:

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

typedef int Item;
typedef int Chave;

// #define bitsword 32
// #define bitsdigit 8
// #define digitsword (bitsword / bitsdigit)
// #define Base (1 << bitsdigit)

const int bitsword = 32;
const int bitsdigit = 1;
const int digitsword = bitsword / bitsdigit;
const int Base = 1 << bitsdigit; // Base = 2^bitsdigit

int digit(int a, int d)
{
```

```
   return (int)((a >> (bitsdigit * (digitsword - 1 - d))) & (Base - 1));
}
```

- Calcular dígito com operações em bits




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

typedef Noh *Arvore;

void imprimeSimbolos(int n, char c)
{
```

```
    int i;
    for (i = 0; i < n; i++)
        printf("%c", c);
}

// Percurso pre-ordem destacando altura de cada subárvore
void preOrdemHierarquico(Arvore r, int altura, char t_filho)
{
    if (r != NULL)
    {
        imprimeSimbolos(altura, t_filho);
        printf("%d\n", r->chave);
        preOrdemHierarquico(r->esq, altura + 1, '>');
        preOrdemHierarquico(r->dir, altura + 1, '<');
    }
}

// Percurso in-ordem
void inOrdem(Arvore r)
{
    if (r != NULL)
    {
        inOrdem(r->esq);
        printf("%d ", r->chave);
        inOrdem(r->dir);
    }
}

int altura(Arvore r)
{
    int hesq, hdir;
    if (r == NULL)
        return -1;
    hesq = altura(r->esq);
    hdir = altura(r->dir);
    if (hesq > hdir)
        return hesq + 1;
    return hdir + 1;
}

Noh *buscaR(Arvore r, Chave chave, int d, Noh **pai)
{
    if (r == NULL)
        return r;
    if (r->chave == chave)
        return r;
    if (digit(chave, d) == 0)
    {
```

```
        *pai = r;
        return buscaR(r->esq, chave, d + 1, pai);
    }
    // digit(chave, d) == 1
    *pai = r;
    return buscaR(r->dir, chave, d + 1, pai);
}

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 d)
{
    if (r == NULL)
    {
        return novo;
    }
    if (digit(novo->chave, d) == 0)
    {
        r->esq = insereR(r->esq, novo, d + 1);
    }
    else // digit(novo->chave, d) == 1
    {
        r->dir = insereR(r->dir, novo, d + 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, *p = 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)
    {
        p = aux;
        if (aux->dir != NULL)
            aux = aux->dir;
        else
            aux = aux->esq;
    }
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
    if (p->esq == aux)
        p->esq = removeRaiz(aux);
    else // p->dir == aux
        p->dir = removeRaiz(aux);
    return alvo;
}

Arvore remover(Arvore r, Chave chave)
{
    Noh *alvo, *aux, *p = NULL;
    alvo = buscaR(r, chave, 0, &p);
    if (alvo == NULL)
    {
        // printf("Nao achou\n");
        return r;
    }
    aux = removeRaiz(alvo);
    if (alvo == r) // removeu a raiz da árvore
        return aux;
    if (p->esq == alvo)
        p->esq = aux;
    if (p->dir == alvo)
        p->dir = aux;
    return r;
}

int main(int argc, char *argv[])
{
    int i, n;
    Arvore r = NULL;
    Noh *aux, *pai;
    int chave_aux;
```

```c
int chaves[7];

if (argc != 2)
{
    printf("Numero incorreto de parametros. Ex: arvoreDigitalBinaria 10\n");
    return 0;
}

n = atoi(argv[1]);

// for (int j = 1; j < n; j++)
// {
//     printf("%d\n", j);
//     for (i = 0; i < digitsword; i++)
//         printf("%d ", digit(j, i));
//     printf("\n");
// }
// exit(1);

// inserção em ordem crescente
for (i = 0; i < n; i++)
{
    chave_aux = i;
    r = inserir(r, chave_aux, chave_aux);
    if (i == 0)
        chaves[0] = chave_aux;
    else if (i == n / 2)
        chaves[1] = chave_aux;
    else if (i == n - 1)
        chaves[2] = chave_aux;
}

// inserção em ordem decrescente
// for (i = 0; i < n; i++)
// {
//     chave_aux = n - 1 - i;
//     r = inserir(r, chave_aux, chave_aux);
//     if (i == 0)
//         chaves[0] = chave_aux;
//     else if (i == n / 2)
//         chaves[1] = chave_aux;
//     else if (i == n - 1)
//         chaves[2] = chave_aux;
// }

// inserção aleatoria
// for (i = 0; i < n; i++)
// {
```

```
//     chave_aux = rand();
//     r = inserir(r, chave_aux, chave_aux);
//     if (i == 0)
//         chaves[0] = chave_aux;
//     else if (i == n / 2)
//         chaves[1] = chave_aux;
//     else if (i == n - 1)
//         chaves[2] = chave_aux;
// }

chaves[3] = 0;
chaves[4] = n / 2;
chaves[5] = n - 1;
chaves[6] = n;

preOrdemHierarquico(r, 0, ' ');
inOrdem(r);
printf("\n");

printf("altura = %d\n", altura(r));

for (i = 0; i < 7; i++)
{
    aux = buscaR(r, chaves[i], 0, &pai);
    if (aux != NULL)
        printf("chave de %d = %d\n", chaves[i], aux->chave);
    else
    {
        printf("nao encontrou %d\n", chaves[i]);
        aux = r;
    }
}

for (i = 0; i < n / 2; i++)
{
    // printf("remover %d\n", 2 * i + 1);
    r = remover(r, 2 * i + 1);
}

preOrdemHierarquico(r, 0, ' ');
inOrdem(r);
printf("\n");

printf("altura = %d\n", altura(r));

for (i = 0; i < 7; i++)
{
    aux = buscaR(r, chaves[i], 0, &pai);
```

```
        if (aux != NULL)
            printf("chave de %d = %d\n", chaves[i], aux->chave);
        else
        {
            printf("nao encontrou %d\n", chaves[i]);
            aux = r;
        }
    }

    return 0;
}
```

Códigos para árvore de busca digital geral:

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

typedef int Item;
typedef int Chave;

// #define bitsword 32
// #define bitsdigit 8
// #define digitsword (bitsword / bitsdigit)
// #define Base (1 << bitsdigit)

const int bitsword = 32;
const int bitsdigit = 8;
const int digitsword = bitsword / bitsdigit;
const int Base = 1 << bitsdigit; // Base = 2^bitsdigit

int digit(int a, int d)
{
    return (int)((a >> (bitsdigit * (digitsword - 1 - d))) & (Base - 1));
}

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);
}

// Percurso pre-ordem destacando altura de cada subárvore
void preOrdemHierarquico(Arvore r, int altura, char t_filho)
{
   int i;
   if (r != NULL)
   {
        imprimeSimbolos(altura, t_filho);
        printf("%d\n", r->chave);
        for (i = 0; i < Base; i++)
            preOrdemHierarquico(r->filhos[i], altura + 1, ' ');
   }
}

// Percurso in-ordem
void inOrdem(Arvore r, int d)
{
   int i;
   if (r != NULL)
   {
        i = 0;
        while (i <= digit(r->chave, d))
        {
            inOrdem(r->filhos[i], d + 1);
            i++;
        }
        printf("%d ", r->chave);
        while (i < Base)
        {
            inOrdem(r->filhos[i], d + 1);
            i++;
        }
   }
}

int altura(Arvore r)
{
   int i, h, hmax;
   if (r == NULL)
        return -1;
   hmax = -1;
   for (i = 0; i < Base; i++)
   {
        h = altura(r->filhos[i]);
        if (h > hmax)
            hmax = h;
```

```
    }
    return hmax + 1;
}

Noh *buscaR(Arvore r, Chave chave, int d, Noh **pai)
{
    if (r == NULL)
        return r;
    if (r->chave == chave)
        return r;
    *pai = r;
    return buscaR(r->filhos[digit(chave, d)], chave, d + 1, pai);
}

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(int));
    for (i = 0; i < Base; i++)
        novo->filhos[i] = NULL;
    return novo;
}

Arvore insereR(Arvore r, Noh *novo, int d)
{
    int i;
    if (r == NULL)
    {
        return novo;
    }
    i = digit(novo->chave, d);
    r->filhos[i] = insereR(r->filhos[i], novo, d + 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, *p = 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)
    {
        p = aux;
        aux = aux->filhos[i];
        iaux = i;
        for (i = 0; i < Base; i++)
            if (aux->filhos[i] != NULL)
                break;
    }
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
    p->filhos[iaux] = removeRaiz(aux);
    return alvo;
}

Arvore remover(Arvore r, Chave chave)
{
    Noh *alvo, *aux, *p = NULL;
    int i;
    alvo = buscaR(r, chave, 0, &p);
    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 (p->filhos[i] == alvo)
            break;
    p->filhos[i] = aux;
    return r;
}
```

```c
int main(int argc, char *argv[])
{
   int i, n;
   Arvore r = NULL;
   Noh *aux, *pai;
   int chave_aux;
   int chaves[7];

   if (argc != 2)
   {
       printf("Numero incorreto de parametros. Ex: arvoreDigitalGeral 10\n");
       return 0;
   }

   n = atoi(argv[1]);

   // for (int j = 1; j < n; j++)
   // {
   //     printf("%d\n", j);
   //     for (i = 0; i < digitsword; i++)
   //         printf("%d ", digit(j, i));
   //     printf("\n");
   // }
   // exit(1);

   // inserção em ordem crescente
   // for (i = 0; i < n; i++)
   // {
   //     chave_aux = i;
   //     r = inserir(r, chave_aux, chave_aux);
   //     if (i == 0)
   //         chaves[0] = chave_aux;
   //     else if (i == n / 2)
   //         chaves[1] = chave_aux;
   //     else if (i == n - 1)
   //         chaves[2] = chave_aux;
   // }

   // inserção em ordem decrescente
   for (i = 0; i < n; i++)
   {
       chave_aux = n - 1 - i;
       r = inserir(r, chave_aux, chave_aux);
       if (i == 0)
           chaves[0] = chave_aux;
       else if (i == n / 2)
           chaves[1] = chave_aux;
       else if (i == n - 1)
```

```
        chaves[2] = chave_aux;
}

// inserção aleatoria
// for (i = 0; i < n; i++)
// {
//     chave_aux = rand();
//     r = inserir(r, chave_aux, chave_aux);
//     if (i == 0)
//         chaves[0] = chave_aux;
//     else if (i == n / 2)
//         chaves[1] = chave_aux;
//     else if (i == n - 1)
//         chaves[2] = chave_aux;
// }

chaves[3] = 0;
chaves[4] = n / 2;
chaves[5] = n - 1;
chaves[6] = n;

preOrdemHierarquico(r, 0, ' ');
inOrdem(r, 0);
printf("\n");

printf("altura = %d\n", altura(r));

for (i = 0; i < 7; i++)
{
    aux = buscaR(r, chaves[i], 0, &pai);
    if (aux != NULL)
        printf("chave de %d = %d\n", chaves[i], aux->chave);
    else
    {
        printf("nao encontrou %d\n", chaves[i]);
        aux = r;
    }
}

for (i = 0; i < n / 2; i++)
{
    // printf("remover %d\n", 2 * i + 1);
    r = remover(r, 2 * i + 1);
}

preOrdemHierarquico(r, 0, ' ');
inOrdem(r, 0);
printf("\n");
```

```
    printf("altura = %d\n", altura(r));

    for (i = 0; i < 7; i++)
    {
        aux = buscaR(r, chaves[i], 0, &pai);
        if (aux != NULL)
            printf("chave de %d = %d\n", chaves[i], aux->chave);
        else
        {
            printf("nao encontrou %d\n", chaves[i]);
            aux = r;
        }
    }

    return 0;
}
```

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;

// #define bitsdigit 8
// #define Base (1 << bitsdigit)

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

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);
```

```
}

// Percurso pre-ordem destacando altura de cada subárvore
void preOrdemHierarquico(Arvore r, int altura, char t_filho)
{
   int i;
   if (r != NULL)
   {
       imprimeSimbolos(altura, t_filho);
       printf("%s\n", r->chave);
       for (i = 0; i < Base; i++)
           preOrdemHierarquico(r->filhos[i], altura + 1, ' ');
   }
}

// Percurso in-ordem
void inOrdem(Arvore r, int d)
{
   int i;
   if (r != NULL)
   {
       i = 0;
       while (i <= r->chave[d]) // problema com comprimento das chaves aqui
       {
           inOrdem(r->filhos[i], d + 1);
           i++;
       }
       printf("%s ", r->chave);
       while (i < Base)
       {
           inOrdem(r->filhos[i], d + 1);
           i++;
       }
   }
}

int altura(Arvore r)
{
   int i, h, hmax;
   if (r == NULL)
       return -1;
   hmax = -1;
   for (i = 0; i < Base; i++)
   {
       h = altura(r->filhos[i]);
       if (h > hmax)
           hmax = h;
   }
```

```
    return hmax + 1;
}

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

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

Arvore removeRaiz(Arvore alvo)
{
    Noh *aux = NULL, *p = 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)
    {
        p = aux;
        aux = aux->filhos[i];
        iaux = i;
        for (i = 0; i < Base; i++)
            if (aux->filhos[i] != NULL)
                break;
    }
    strcpy(alvo->chave, aux->chave);
    alvo->conteudo = aux->conteudo;
    p->filhos[iaux] = removeRaiz(aux);
    return alvo;
}

Arvore remover(Arvore r, Chave chave)
{
    Noh *alvo, *aux, *p = NULL;
    int i;
    alvo = buscaR(r, chave, 0, &p);
    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 (p->filhos[i] == alvo)
            break;
    p->filhos[i] = aux;
    return r;
}

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

```c
int main(int argc, char *argv[])
{
    int i, j, n, W;
    byte *string_aux;
    Arvore r = NULL;
    Noh *aux, *pai;
    int chave_aux;
    byte *chaves[7];

    if (argc != 3)
    {
        printf("Numero incorreto de parametros. Ex: arvoreDigitalString 10 5\n");
        return 0;
    }

    n = atoi(argv[1]);
    W = atoi(argv[2]);

    string_aux = (char *)malloc((W + 1) * sizeof(char));

    for (i = 0; i < 7; i++)
        chaves[i] = (byte *)malloc((W + 1) * sizeof(char));

    // inserção aleatoria
    for (i = 0; i < n; i++)
    {
        chave_aux = rand();
        for (j = 0; j < W; j++)
        {
            // string_aux[j] = chave_aux % Base;
            // string_aux[j] = 65 + ((rand() % (90 - 65 + 1)) % Base);
            string_aux[j] = 48 + ((rand() % (57 - 48 + 1)) % Base);
        }
        string_aux[j] = '\0';
        r = inserir(r, string_aux, chave_aux);
        if (i == 0)
            strcpy(chaves[0], string_aux);
        else if (i == n / 2)
            strcpy(chaves[1], string_aux);
        else if (i == n - 1)
            strcpy(chaves[2], string_aux);
    }

    strcpy(chaves[3], "0");

    sprintf(string_aux, "%d", n / 2);
    strcpy(chaves[4], string_aux);
```

```c
sprintf(string_aux, "%d", n - 1);
strcpy(chaves[5], string_aux);

sprintf(string_aux, "%d", n);
strcpy(chaves[6], string_aux);

preOrdemHierarquico(r, 0, ' ');
inOrdem(r, 0);
printf("\n");

printf("altura = %d\n", altura(r));

for (i = 0; i < 7; i++)
{
    aux = buscaR(r, chaves[i], 0, &pai);
    if (aux != NULL)
        printf("chave de %s = %s\n", chaves[i], aux->chave);
    else
    {
        printf("nao encontrou %s\n", chaves[i]);
        aux = r;
    }
}

for (i = 0; i < 7; i++)
{
    // printf("remover %s\n", chaves[i]);
    r = remover(r, chaves[i]);
}

preOrdemHierarquico(r, 0, ' ');
inOrdem(r, 0);
printf("\n");

printf("altura = %d\n", altura(r));

for (i = 0; i < 7; i++)
{
    aux = buscaR(r, chaves[i], 0, &pai);
    if (aux != NULL)
        printf("chave de %s = %s\n", chaves[i], aux->chave);
    else
    {
        printf("nao encontrou %s\n", chaves[i]);
        aux = r;
    }
}
```

```
    free(string_aux);

    for (i = 0; i < 7; i++)
        free(chaves[i]);

    return 0;
}
```

Eficiência 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(W).
- Note que, em geral isso é bastante bom, já que para
    - bitsdigit = 1 (Base = 2) e W = 32, podemos ter
        - n ~= 2^32 ~= 4 bilhões.
- Eficiência esperada das operações caso as chaves sejam aleatórias
    - é proporcional a log_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 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.
                    - 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.