

Estruturas de Dados 1

481440

Maio/2018

Mario Liziér
lazier@ufscar.br

Matrizes

- Estrutura retangular (bidimensional) de valores, organizados em linhas e colunas
 - Generalização para dimensões superiores (*arrays*)
- Muito útil em diversas aplicações
- Matrizes especiais:
 - **Quadrada**
 - Diagonal, Tri-diagonal, Simétrica, Triangular inferior/superior
 - **Esparsa**
- Representações computacionais

Matrizes (densas)

- Retangular/quadrada densas
 - Acesso aos elementos: $O(1)$
 - Consumo de memória: $O(n^2)$

```
double M[5][5];
```

```
M[2][3] = 4;
```

A 5x5 matrix M is shown with row index i and column index j . The matrix contains the following values:

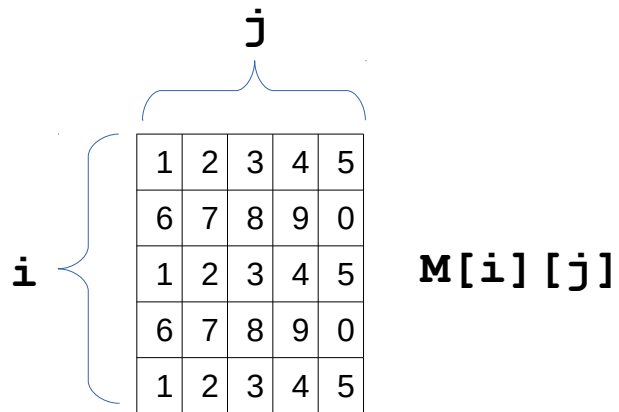
| | | | | |
|---|---|---|---|---|
| 1 | 4 | 0 | 6 | 7 |
| 2 | 3 | 5 | 0 | 7 |
| 8 | 5 | 4 | 4 | 8 |
| 5 | 6 | 4 | 8 | 1 |
| 5 | 5 | 0 | 7 | 2 |

The matrix is labeled $M[i][j]$ on the right.

- Transparente na linguagem C
- Agora, como utilizaríamos vetor unidimensional para armazenar uma matriz?

Matrizes em vetores

- Vetor armazenando uma matriz:

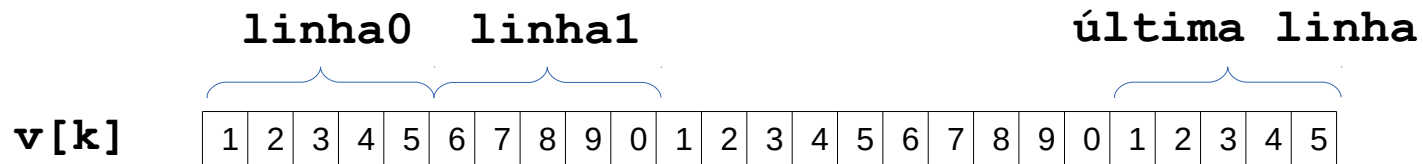


```
double M[nlinhas][ncolunas];
```

```
double v[nlinhas*ncolunas];

double get(int i, int j) {
    // ???
}

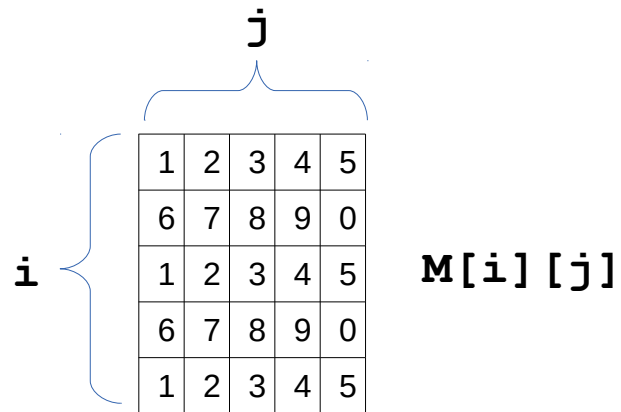
void set(int i, int j, double d) {
    // ???
}
```



```
double v[nlinhas*ncolunas];
```

Matrizes em vetores

- Vetor armazenando uma matriz:

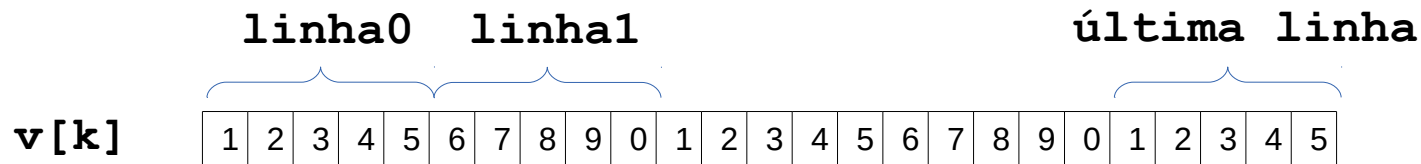


```
double M[nlinhas][ncolunas];
```

```
double v[nlinhas*ncolunas];

double get(int i, int j) {
    return v[i*ncolunas+j];
}

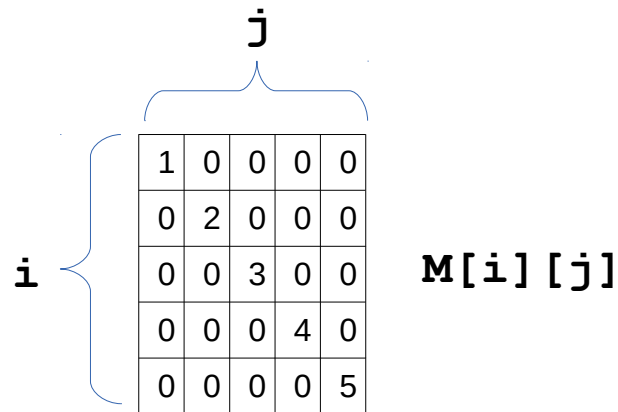
void set(int i, int j, double d) {
    v[i*ncolunas+j] = d;
}
```



```
double v[nlinhas*ncolunas];
```

Matrizes especiais quadradas

- Diagonal: $M[i][j] = 0$ para $i \neq j$



| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | j |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| i | 1 | 0 | 0 | 0 | 0 |
| | 0 | 2 | 0 | 0 | 0 |
| | 0 | 0 | 3 | 0 | 0 |
| | 0 | 0 | 0 | 4 | 0 |
| | 0 | 0 | 0 | 0 | 5 |
| | | | | | |

$M[i][j]$

```
double M[n][n];
```

```
v[n] 1 2 3 4 5
```

```
double v[n];
```

```
double v[dimensao];

double get(int i, int j) {
    // ?????
}

void set(int i, int j, double d) {
    // ?????
}
```

Matrizes especiais quadradas

- Diagonal: $M[i][j] = 0$ para $i \neq j$

| | | | | | |
|---|---|---|---|---|---|
| | | | | | j |
| | 1 | 0 | 0 | 0 | 0 |
| | 0 | 2 | 0 | 0 | 0 |
| | 0 | 0 | 3 | 0 | 0 |
| | 0 | 0 | 0 | 4 | 0 |
| i | 0 | 0 | 0 | 0 | 5 |

$M[i][j]$

```
double M[n][n];
```

```
v[n] 1 2 3 4 5
```

```
double v[n];
```

```
double v[dimensao];

double get(int i, int j) {
    if( i == j) return v[i];
    else return 0;
}

void set(int i, int j, double d) {
    if( i == j) v[i] = d;
}
```

Matrizes especiais quadradas

- Tridiagonal: $M[i][j] = 0$ para $|i-j| > 1$

| | | | | |
|---|---|---|---|---|
| 1 | 6 | 0 | 0 | 0 |
| 6 | 2 | 7 | 0 | 0 |
| 0 | 7 | 3 | 8 | 0 |
| 0 | 0 | 8 | 4 | 9 |
| 0 | 0 | 0 | 9 | 5 |

M[i][j]

```
double M[n][n];
```

```
double v[3*n-2];
```

```
double get(int i, int j) {
    // ????
```

```
void set(int i, int j, double d) {
    // ???
}
```

The diagram shows a 14x14 matrix $\mathbf{v}[\mathbf{k}]$. The columns are indexed 1 through 14. Above the matrix, three groups of columns are identified with blue curly braces:

- inf.** (inferior): columns 6, 7, 8, 9
- diag.** (diagonal): columns 1, 2, 3, 4, 5
- sup.** (superior): columns 10, 11, 12, 13

Columns 14 and 15 are not present in this specific diagram.

```
double v[3*n-2];
```


Matrizes especiais quadradas

- Tridiagonal: $M[i][j] = 0$ para $|i-j| > 1$

| | | | | |
|---|---|---|---|---|
| 1 | 6 | 0 | 0 | 0 |
| 6 | 2 | 7 | 0 | 0 |
| 0 | 7 | 3 | 8 | 0 |
| 0 | 0 | 8 | 4 | 9 |
| 0 | 0 | 0 | 9 | 5 |

$M[i][j]$

`double M[n][n];`

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

`double v[3*n-2];`

```
double v[3*n-2];

double get(int i, int j) {
    switch(i-j) {
        case 1: // inf.
            return v[i-1];
        case 0: // diag.
            return v[n+i-1];
        case -1: // sup.
            return v[2*n+i-1];
        default:
            return 0;
    }
}

void set(int i, int j, double d) {
    switch(i-j) {
        case 1: // inf.
            v[i-1] = d;
        case 0: // diag.
            v[n+i-1] = d;
        case -1: // sup.
            v[2*n+i-1] = d;
    }
}
```

Matrizes especiais quadradas

- Triangular:
 - Inferior: $\mathbf{M}[\mathbf{i}][\mathbf{j}] = 0$ para $\mathbf{i} < \mathbf{j}$

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 6 | 2 | 0 | 0 | 0 |
| 1 | 7 | 3 | 0 | 0 |
| 4 | 2 | 8 | 4 | 0 |
| 6 | 5 | 3 | 9 | 5 |

M[i][j]

```
double M[n][n];
```

```
double v[n*(n+1)/2];

double get(int i, int j) {
    // ????
}

void set(int i, int j, double d) {
    // ????
}
```

Diagram illustrating the mapping of indices to values in the array $v[k]$:

| | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $v[k]$ | 1 | 6 | 2 | 1 | 7 | 3 | 4 | 2 | 8 | 4 | 6 | 5 | 3 | 9 | 5 |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Indices 1-5 are grouped under **linha3**.
 Indices 6-10 are grouped under **linha3**.
 Indices 11-15 are grouped under **linha4**.

```
double v[n*(n+1)/2];
```

Matrizes especiais quadradas

- Triangular:
 - Inferior: $M[i][j] = 0$ para $i < j$

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 6 | 2 | 0 | 0 | 0 |
| 1 | 7 | 3 | 0 | 0 |
| 4 | 2 | 8 | 4 | 0 |
| 6 | 5 | 3 | 9 | 5 |

$M[i][j]$

`double M[n][n];`

```
double v[n*(n+1)/2];
```

```
double get(int i, int j) {  
    if(i >= j) return v[i*(i+1)/2+j];  
    else return 0;  
}
```

```
void set(int i, int j, double d) {  
    if(i >= j) v[i*(i+1)/2+j] = d;  
}
```

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 2 | 1 | 7 | 3 | 4 | 2 | 8 | 4 | 6 | 5 | 3 | 9 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$v[k]$

`double v[n*(n+1)/2];`

Matrizes especiais quadradas

- Triangular:
 - Superior: $\mathbf{M}[\mathbf{i}][\mathbf{j}] = 0$ para $\mathbf{i} > \mathbf{j}$

A 5x5 matrix M is shown with row index i and column index j . The matrix contains the following values:

| | | | | |
|---|---|---|---|---|
| 1 | 6 | 1 | 4 | 6 |
| 0 | 2 | 7 | 2 | 5 |
| 0 | 0 | 3 | 8 | 3 |
| 0 | 0 | 0 | 4 | 9 |
| 0 | 0 | 0 | 0 | 5 |

The notation $M[i][j]$ is shown to the right of the matrix.

```
double M[n][n];
```

```
double v[n*(n+1)/2];

double get(int i, int j) {
    // ???
}

void set(int i, int j, double d) {
    // ???
}
```

v[k]

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 2 | 1 | 7 | 3 | 4 | 2 | 8 | 4 | 6 | 5 | 3 | 9 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

coluna3 coluna4

```
double v[n*(n+1)/2];
```

Matrizes especiais quadradas

- Triangular:
 - Superior: $M[i][j] = 0$ para $i > j$

Diagram illustrating a 5x5 matrix $M[i][j]$ with indices i and j shown. The matrix is upper triangular, meaning $M[i][j] = 0$ for $i > j$.

| | | | | |
|--|--|--|--|--|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Matrix values (row by row):

| | | | | |
|---|---|---|---|---|
| 1 | 6 | 1 | 4 | 6 |
| 0 | 2 | 7 | 2 | 5 |
| 0 | 0 | 3 | 8 | 3 |
| 0 | 0 | 0 | 4 | 9 |
| 0 | 0 | 0 | 0 | 5 |

`double M[n][n];`

```
double v[n*(n+1)/2];

double get(int i, int j) {
    if(i <= j) return v[j*(j+1)/2+i];
    else return 0;
}

void set(int i, int j, double d) {
    if(i <= j) v[j*(j+1)/2+i] = d;
}
```

Diagram illustrating the storage of matrix elements in a 1D array $v[k]$. The array is divided into segments corresponding to columns of the matrix. The first segment is labeled "coluna3" and the second segment is labeled "coluna4".

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 2 | 1 | 7 | 3 | 4 | 2 | 8 | 4 | 6 | 5 | 3 | 9 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

`double v[n*(n+1)/2];`

Matrizes especiais quadradas

- Simétrica:
 - $M[i][j] = M[j][i]$ para todo i e j

$M[i][j]$

| | | | | | |
|---|---|---|---|---|---|
| | j | | | | |
| i | 1 | 6 | 1 | 4 | 6 |
| | 6 | 2 | 7 | 2 | 5 |
| | 1 | 7 | 3 | 8 | 3 |
| | 4 | 2 | 8 | 4 | 9 |
| | 6 | 5 | 3 | 9 | 5 |

```
double M[n][n];
```

```
double v[n*(n+1)/2];
```

```
double get(int i, int j) {  
    // ????
```

```
}  
  
void set(int i, int j, double d) {  
    // ????
```

```
}
```

linha3 linha4

| | | | | | | | | | | | | | | |
|------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| v[k] | | | | | | | | | | | | | | |
| 1 | 6 | 2 | 1 | 7 | 3 | 4 | 2 | 8 | 4 | 6 | 5 | 3 | 9 | 5 |

```
double v[n*(n+1)/2];
```

Matrizes especiais quadradas

- Simétrica:
 - $M[i][j] = M[j][i]$ para todo i e j

$M[i][j]$

| | | | | | | |
|---|--|---|---|---|---|---|
| | | j | | | | |
| i | | 1 | 6 | 1 | 4 | 6 |
| | | 6 | 2 | 7 | 2 | 5 |
| | | 1 | 7 | 3 | 8 | 3 |
| | | 4 | 2 | 8 | 4 | 9 |
| | | 6 | 5 | 3 | 9 | 5 |

```
double M[n][n];
```

```
double v[n*(n+1)/2];
```

```
double get(int i, int j) {  
    if(i >= j) return v[i*(i+1)/2+j];  
    else return v[j*(j+1)/2+i];  
}  
void set(int i, int j, double d) {  
    if(i >= j) v[i*(i+1)/2+j] = d;  
    else v[j*(j+1)/2+i] = d;  
}
```

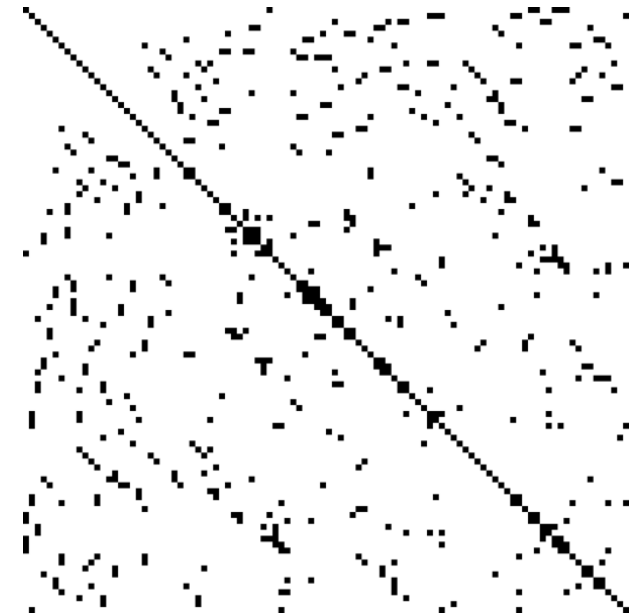
linha3 linha4

| | | | | | | | | | | | | | | | |
|------|--------|---|---|---|---|--------|---|---|---|---|---|---|---|---|---|
| v[k] | 1 | 6 | 2 | 1 | 7 | 3 | 4 | 2 | 8 | 4 | 6 | 5 | 3 | 9 | 5 |
| | linha3 | | | | | linha4 | | | | | | | | | |

```
double v[n*(n+1)/2];
```

Matrizes esparsas

- Grande maioria de valores nulos
 - Não existe um limiar fixo, precisamos analisar cada caso
- Para matrizes pequenas, armazenar n^2 elementos não é problemático
- Mas para matrizes grandes, alocar n^2 elementos pode ser proibitivo!
 - Se a grande maioria dos valores forem nulos, não precisamos alocar todos eles!
- Representação computacional:
 - Vetores
 - Listas



Matrizes esparsas

- Alocação por vetores:

0 0 0 2 0 0 1 0
0 6 0 0 7 0 0 3
0 0 0 9 0 8 0 0
0 4 5 0 0 0 0 0

(a) A 4×8 matrix

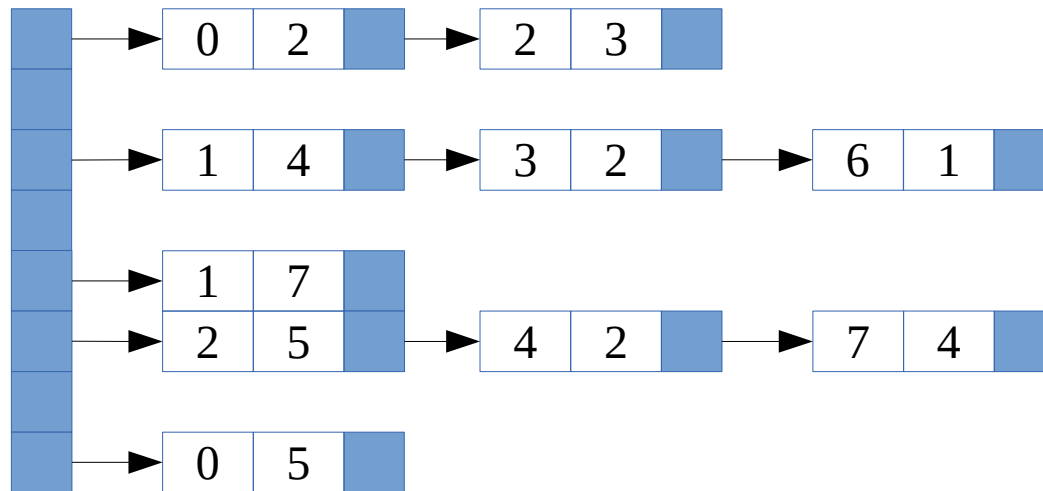
| | | | | | | | | | | |
|-------|--|---|---|---|---|---|---|---|---|---|
| a[] | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| row | | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4 |
| col | | 4 | 7 | 2 | 5 | 8 | 4 | 6 | 2 | 3 |
| value | | 2 | 1 | 6 | 7 | 3 | 9 | 8 | 4 | 5 |

(b) Its representation

- Qual o custo das operações:
 - Construir
 - Acessar uma posição
 - Atribuir nulo a uma posição
 - Atribuir não nulo a uma posição

Matrizes esparsas

- Alocação por listas:
 - Indexação por linha ou coluna

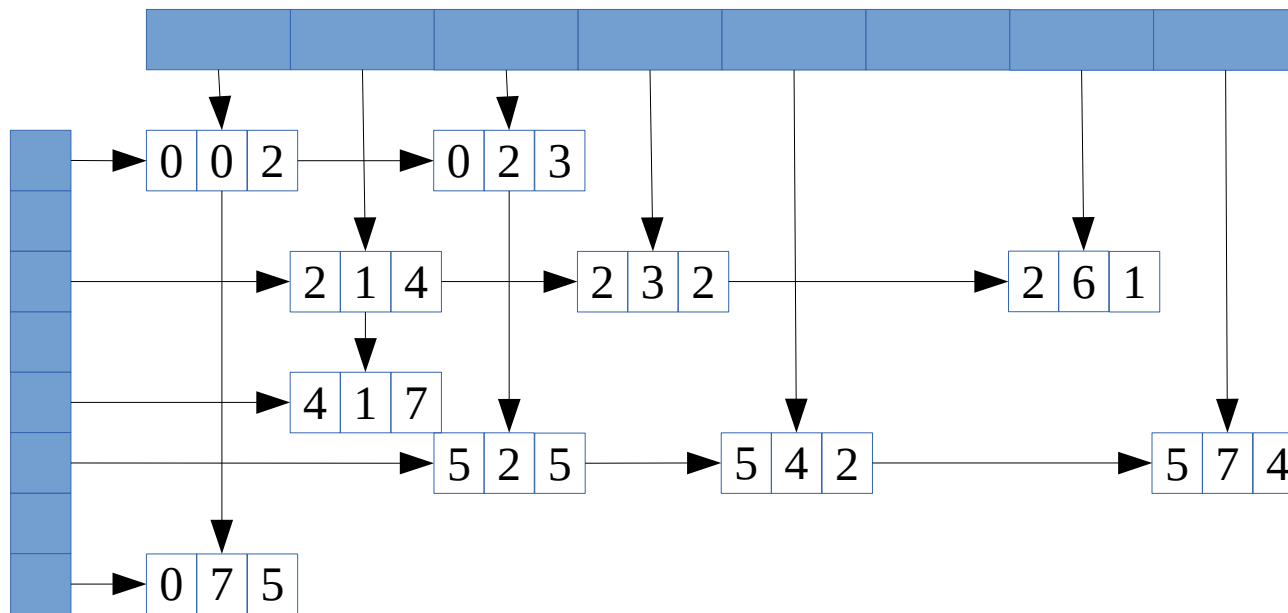


- Qual o custo das operações:
 - Construir
 - Acessar uma posição
 - Atribuir nulo a uma posição
 - Atribuir não nulo a uma posição

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 4 | 0 | 2 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 5 | 0 | 2 | 0 | 0 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Matrizes esparsas

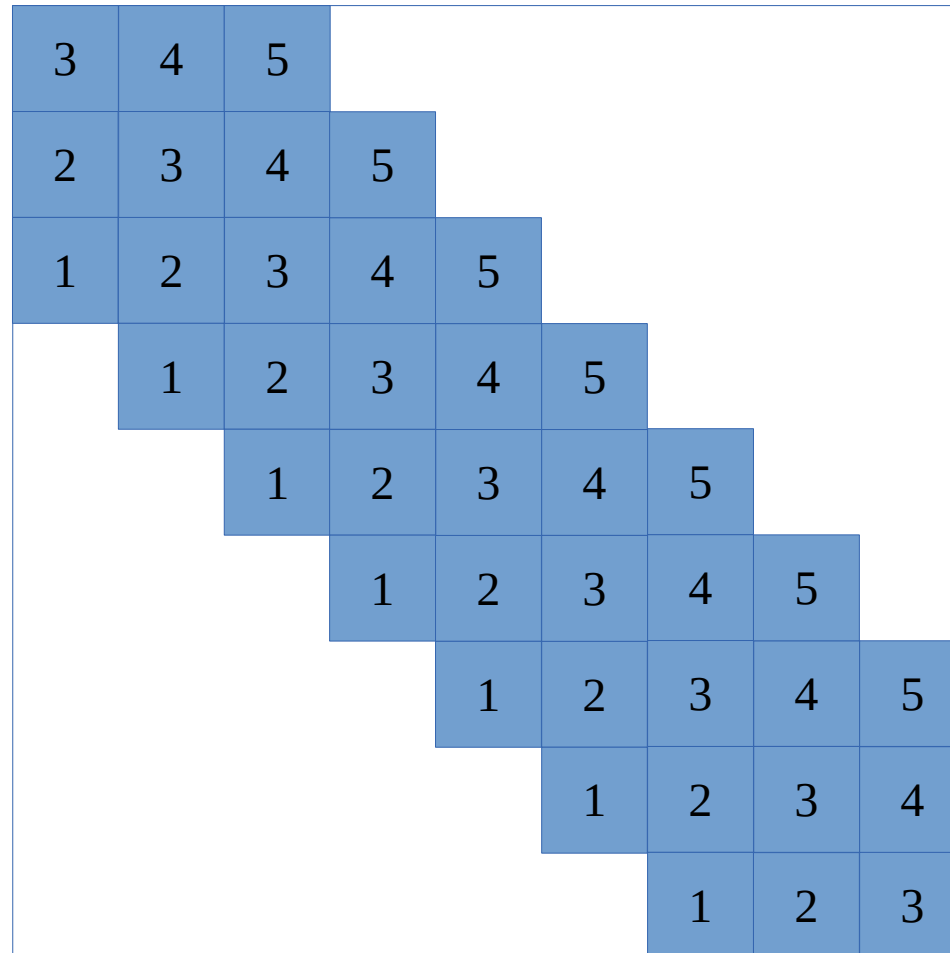
- Alocação por listas:
 - Indexação por linha e coluna
- Qual o custo das operações:
 - Construir
 - Acessar uma posição
 - Atribuir nulo a uma posição
 - Atribuir não nulo a uma posição



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 4 | 0 | 2 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 5 | 0 | 2 | 0 | 0 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Exercício

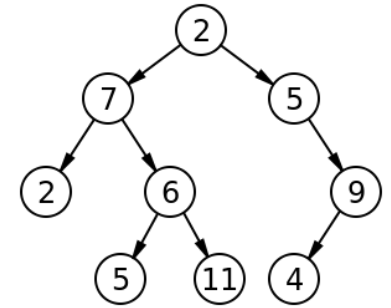
- Implemente eficientemente o armazenamento de uma matriz com 5 diagonais paralelas?



The diagram illustrates a matrix structure with 5 parallel diagonals. The matrix is represented as a grid of blue cells, each containing a number. The numbers are arranged in a descending staircase pattern from top-left to bottom-right. The first row contains 3, 4, 5. The second row contains 2, 3, 4, 5. The third row contains 1, 2, 3, 4, 5. The fourth row contains 1, 2, 3, 4, 5. The fifth row contains 1, 2, 3, 4, 5. The sixth row contains 1, 2, 3, 4, 5. The seventh row contains 1, 2, 3, 4. The eighth row contains 1, 2, 3.

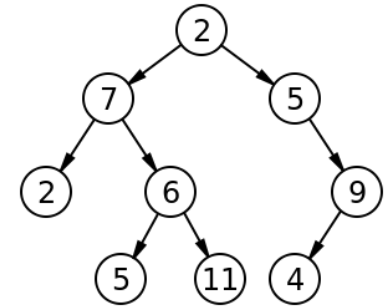
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|--|
| 3 | 4 | 5 | | | | | | | |
| 2 | 3 | 4 | 5 | | | | | | |
| 1 | 2 | 3 | 4 | 5 | | | | | |
| | 1 | 2 | 3 | 4 | 5 | | | | |
| | | 1 | 2 | 3 | 4 | 5 | | | |
| | | | 1 | 2 | 3 | 4 | 5 | | |
| | | | | 1 | 2 | 3 | 4 | 5 | |
| | | | | | 1 | 2 | 3 | 4 | |
| | | | | | | 1 | 2 | 3 | |

Estruturas de dados



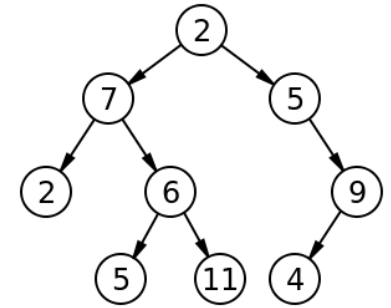
- Por que “estruturar” os dados?
 - Principal operação: BUSCAS
 - Se vamos armazenar algo, precisamos recuperar esses dados
 - Como estruturar um conjunto de dados para obter o melhor desempenho nas operações sobre esses dados?
- Preocupações constantes:
 - Tempo:
 - Custo para buscar ou manter a estrutura (inserir/remover/construir)
 - Não queremos comparar (nem tocar) muitos dados
 - Espaço:
 - Como representar esta estrutura (espaço adicional ao já ocupados pelos dados)

Estruturas de dados



- Estruturas estáticas:
 - Organizam um conjunto de dados já completo
 - Operações de inserir/remover inexistentes ou muito ineficientes
 - Questões relevantes:
 - Custo para construir a estrutura de dados
 - Custo das operações de busca ou acesso aos dados
 - Espaço de memória extra (além do ocupado pelos dados)
 - Tipos de dados considerados
 - Características dos dados

Estruturas de dados



- Estruturas dinâmicas:
 - Organizam um conjunto de dados que pode se alterar ao longo do tempo
 - Questões relevantes:
 - Todas das estruturas estáticas
 - Custo das operações de inserção ou remoção
 - Algumas estruturas permitem apenas inserir novos elementos
 - Outras permitem inserir e remover
- Em ambos o casos, precisamos analisar a viabilidade de cada uma para cada aplicação!
 - Permitir uma operação não significa que devemos usar sempre!
 - Complexidade

Estruturas lineares

- “Um elemento após o outro”
 - Arrays (alocação sequencial dos dados)
 - Busca sequencial: $O(N)$
 - Busca binária: $O(\log N)$ ← Isso é bom!
 - Inserção/Remoção: $O(N)$
 - Listas ligadas
 - Busca/Inserção/Remoção: $O(N)$
- Positivo: simplicidade nas implementações
- Negativo:
 - Quando N é muito grande: manutenção muito ineficiente
 - Buscas em dados dinâmicos ruim
 - A própria estruturação impõe uma restrição: única sequência

Estruturas lineares

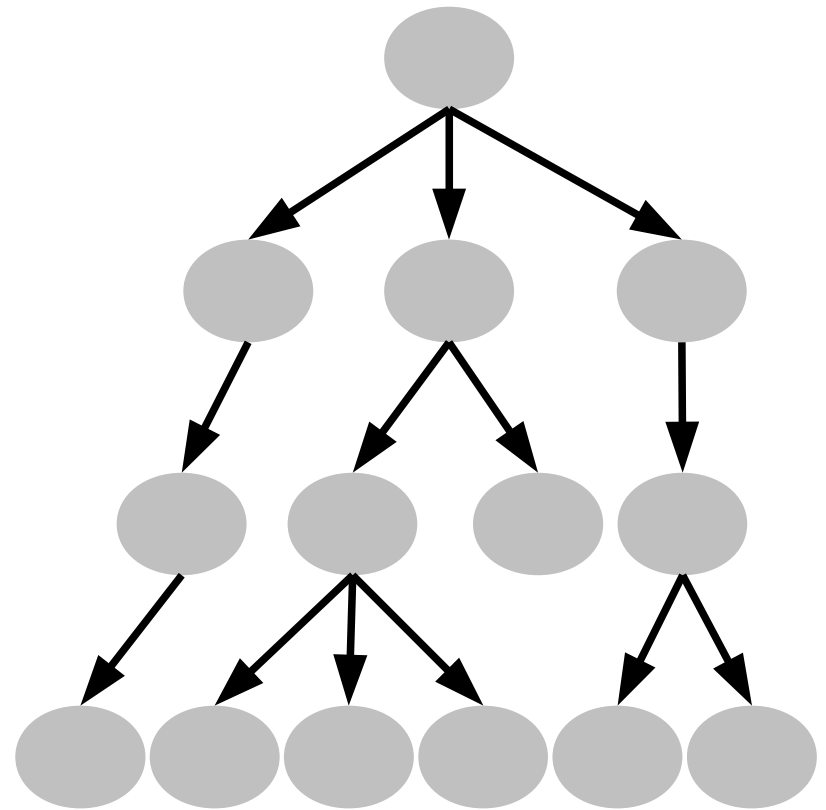
- Porque a busca binária é boa?
 - $O(\log N)$
- Onde está o “poder” da busca binária?

```
int bsequencial( T vetor[], T key, int n )
{
    int i = 0;
    while( (i < n) && (vetor[i] < key) )
        i++;
    if ( (i < n) && (vetor[i] == key) )
        return i;
    else
        return -1;
}
```

```
int bbinaria( T vetor[], T key, int n )
{
    int imax = n-1;
    int imin = 0;
    while( imax >= imin )
    {
        int imid = imin + ((imax - imin) / 2);
        if( key > vetor[imid] )
            imin = imid + 1;
        else if( key < vetor[imid] )
            imax = imid - 1;
        else
            return imid;
    }
    return -1;
}
```

Árvores

- Estrutura hierárquica e não-linear
- Definição recursiva:
 - Uma árvore pode ser vazia ou ser um elemento com duas (ou mais) árvores (chamadas de sub-árvores)
- Hierárquica:
 - Sub-árvores disjuntas
- Não-linear:
 - Um elemento pode possuir mais de um “próximo” imediato (diferente da lista)



Árvores

- “Poder” computacional:
 - Divisão de um conjunto em partes menores
 - Permite resolver problemas muito grandes
- Estruturas famosa:
 - Árvore *Heap* ← aula passada
 - Árvore binária de busca (ABB ou BST) ← próxima aula
 - Árvore 2-3
 - Árvore AVL
 - Árvore *Red-Black* (Rublo-Negra, Vermelho-Preta)
 - Árvore B
 - ...

Implementações

```
struct node {  
    T data;  
    Lista filhos;  
    struct node *pai; // opcional  
};  
  
struct node {  
    T data;  
    struct node *esq, *dir;  
    struct node *pai; // opcional  
};  
  
struct node {  
    T data;  
    int nfilhos;  
    struct node *filhos[GRAU_MAX];  
    struct node *pai; // opcional  
};
```

Profundidade

- Como calculamos a profundidade de um nó?
 - Número de “pais” do nó em questão até a raiz
 - Distância do nó até a raiz

Profundidade de um nó

- Implementação utilizando o exemplo anterior:
 - Iterativa:

```
int profundidade( struct node *p ) {  
    int i = 0;  
    while( p->pai ) {  
        p = p->pai;  
        i = i + 1;  
    }  
    return i;  
}
```

Código feito em aula:

```
int profundidade( struct node *no) {  
    int contador = -1;  
    while( no ) {  
        contador++;  
        no = no->pai;  
    }  
    return contador;  
}
```

Profundidade de um nó

- Implementação utilizando o exemplo anterior:
 - Recursiva:

```
int profundidade( struct node *p ) {  
    if( p->pai )  
        return 1 + profundidade( p->pai );  
    else  
        return 0;  
}
```

Código feito em aula:

```
int profundidade( struct node *no) {  
    if(!no)  
        return -1;  
    return 1 + profundidade(no->pai);  
}
```

Altura

- Como calculamos a altura de uma árvore (ou sub-árvore)?
 - Número de níveis hierárquicos
 - A raiz está no nível 0

Altura de sub-árvore

- Implementação utilizando o exemplo anterior:
 - Recursiva:

```
int altura( struct node *p ) {  
    if(vazia(p->filhos))  
        return 0;  
    int h = 0;  
    for(Iterador it = primeiro(p->filhos); !acabou(it); proximo(it)) {  
        int temp = altura(elemento(it));  
        if( temp > h )  
            h = temp;  
    }  
    return 1 + h;  
}
```

```
int altura( struct node *p ) {  
    if(p->nfilhos == 0)  
        return 0;  
    int h = 0;  
    for(int i = 0; i < p->nfilhos; i++) {  
        int temp = altura(p->filhos[i]);  
        if( temp > h )  
            h = temp;  
    }  
    return 1 + h;  
}
```

Código feito em aula:

```
int altura(struct node* no) {  
    if( !no )  
        return -1;  
    int maior = 0;  
    for(int i=0; i<no->nfilhos; i++) {  
        int temp = altura(no->filhos[i]);  
        if( temp > maior )  
            maior = temp;  
    }  
    return maior + 1;  
}
```

Número de elementos

- Como podemos contar o número de elementos de uma árvore (ou sub-árvore)?
 - Versão recursiva
 - Versão iterativa

Número de elementos

- Implementação utilizando o exemplo anterior:
 - Recursiva:

```
int size(struct node* p) {  
    if (p==NULL)  
        return 0;  
    else {  
        int acc = 0;  
        for(int i = 0; i < p->nfilhos; i++) {  
            acc += size(p->filhos[i]);  
        }  
        return 1 + acc;  
    }  
}
```

Número de elementos

- Implementação utilizando o exemplo anterior:
 - Iterativa:

```
int size(struct node* p) {  
    if (p==NULL)  
        return 0;  
    else {  
        int count = 0;  
        Fila f;  
        push(f, p);  
        while( !empty(f) ) {  
            p = pop(f);  
  
            count++;  
  
            for(int i = 0; i < p->nfilhos; i++)  
                push(f, p->filhos[i]);  
        }  
        return count;  
    }  
}
```

Referências:

- Livro do Drozdek
- Livro do Cormen
- Livro do Robert Sedgewick
 - <https://algs4.cs.princeton.edu>