Estruturas de Dados I

Dicionários e Tabelas de Dispersão

Igor Machado Coelho

03/11/2026

1. Dicionários
2. Tabelas de Acesso Direto
3. Tabelas de Dispersão
4. Encadeamento Exterior
5. Endereçamento Aberto
6. Dicionários: Hash $\times$ Árvores
7. Agradecimentos

Section 1

Dicionários

Pré-Requisitos

São requisitos para essa aula:

- Introdução/Fundamentos de Programação (em alguma linguagem de programação)
- Interesse em aprender C/C++
- Noções de tipos de dados e vetores
- Aula de Sacos e Listas (listas encadeadas)
- Noções de análise de complexidade (incluindo custo amortizado)

Agradecimentos especiais ao prof. Fabiano Oliveira, cujo material sobre Tabelas de Dispersão forma a base desses slides.

Dicionários

O Dicionário (do inglês *Dictionary*) ou Mapa (do inglês *Map*) é um Tipo Abstrato de Dado (TAD) que associa **chaves** a **valores**. Também é conhecido como *mapeamento* ou *tabela de símbolos*.

Supondo um dicionário `M` de *caractere para float*, por exemplo:

- Podemos *adicionar* a chave `B` com valor `100.0`, e a chave `C` com valor `150.0`
- Podemos *consultar* a chave `B`, recebendo o valor `100.0`
- Podemos *atualizar* o valor da chave `B` para `120.0` (adicionando novamente!)
- Podemos *remover* a chave `C`

```
M:  B -> 120.0
    C -> 150.0  (removida)
```

Cada chave aparece **no máximo uma vez**: é como um Conjunto de chaves, cada uma com um valor associado.

Dicionários na computação

Dicionários são estruturas fundamentais na própria computação. Algumas linguagens (como Python) oferecem suporte nativo:

```
>>> M = dict()
>>> M['A'] = 100.0
>>> M['A']
100.0
```

- **Python:** `dict` (tabela de dispersão)
- **Java:** `HashMap` (tabela de dispersão) e `TreeMap` (árvore balanceada)
- **C++:** `std::unordered_map` (tabela de dispersão) e `std::map` (árvore balanceada)

Assim como *vetores*, guardam dados de um mesmo tipo. A diferença é que o **índice** (a chave) pode ser de um tipo arbitrário: um caractere, um número grande, um nome...

Operações de um Dicionário

Um Dicionário possui 5 operações básicas:

- `tem(c)`: a chave `c` existe? (do inglês *contains*)
- `consulta(c)`: retorna o valor associado à chave `c`, que **deve existir** (do inglês *at*)
- `adiciona(c, v)`: associa o valor `v` à chave `c`, inserindo **ou atualizando** (do inglês *put*)
- `remove(c)`: remove a chave `c`, se existir, e diz se removeu (do inglês *remove*)
- `tamanho()`: número de chaves

Como no Conjunto, operações que podem não fazer nada devolvem `bool`.

Conceito: Dicionário

O conceito precisa de **dois tipos**: o da chave `K` e o do valor `V`.

```
template <typename D, typename K, typename V>
concept DicionarioTAD =
  requires(D d, K k, V v) {
    { d.tem(k) }      -> std::same_as<bool>;
    { d.consulta(k) } -> std::same_as<V>;
    { d.adiciona(k, v) }; // insere/atualiza
    { d.remove(k) }   -> std::same_as<bool>;
    { d.tamanho() }   -> std::same_as<int>;
};
```

```
// Exemplo de uso
DicionarioTAD<char, float> auto d = ...;
d.cria();
d.adiciona('A', 100.0);
d.adiciona('B', 200.0);
d.adiciona('A', 150.0);   // atualiza!
println("{}", d.consulta('A'));  // 150
println("{}", d.tem('C'));     // false
println("{}", d.remove('B'));     // true
println("{}", d.tamanho());       // 1
d.libera();
```

Cuidado: `consulta(c)` tem uma **pré-condição**: `tem(c)` deve ser verdadeiro. Em C++26: `pre(tem(c))`.

Como implementar um Dicionário?

Já conhecemos algumas opções, com n chaves:

Implementação	`adiciona(c,v)`	`consulta(c)`	`remove(c)`
vetor não ordenado (como o Saco)	$O(n)$ (verifica se existe!)	$O(n)$	$O(n)$
vetor ordenado (como o Conjunto Ordenado)	$O(n)$	$O(\log n)$	$O(n)$
árvore balanceada (próxima aula)	$O(\log n)$	$O(\log n)$	$O(\log n)$
"estrutura dos sonhos"	$O(1)$	$O(1)$	$O(1)$

Existe a "estrutura dos sonhos"? **Quase!** Veremos hoje as **Tabelas de Dispersão** (do inglês *hash tables*), com custo $O(1)$ **em média**.

Section 2

Tabelas de Acesso Direto

Ideia: usar a chave como índice!

Se as chaves são pequenas, por exemplo um `char` (256 valores possíveis), basta um vetor com uma posição **por chave possível**:

```
chave:     ... 'A'   'B'   'C'   'D' ...
valores:   ... 100.0 ----  150.0 ----  ...
ocupado:   ... true  false true  false ...
```

O valor da chave c fica em `valores[c]`. Não há busca: o acesso é **direto**!

Essa é uma **Tabela de Acesso Direto**: todas as operações custam $O(1)$, e o espaço é $O(M)$, onde M é o número de chaves **possíveis** (não as presentes!).

Implementação: Acesso Direto

```
struct DicionarioDireto {
   float valores[256]; // valor de cada chave
   bool  ocupado[256]; // chave presente?
   int N;              // num. de chaves
   void cria() {
      N = 0;
      for (int i = 0; i < 256; i++)
         ocupado[i] = false;
   }
   void libera() { N = 0; }
   int tamanho() { return N; }
   bool tem(char c) {
      return ocupado[c];
   }
   float consulta(char c) {
      return valores[c];  // pre: tem(c)
   }
   void adiciona(char c, float v) {
      if (!ocupado[c]) { ocupado[c] = tr
      valores[c] = v;
   }
   bool remove(char c) {
      if (!ocupado[c]) return false;
      ocupado[c] = false;   N--;
      return true;
   }
};
static_assert(
    DicionarioTAD<DicionarioDireto,
                  char, float>);
```

Desafio: o código **tem erro** com *caracteres negativos*! Explore `unsigned char`.

Quando usar Acesso Direto?

- Quando o número de chaves **possíveis** M é pequeno: `char`, dias do mês, notas de 0 a 10...
- Ou quando o tempo $O(1)$ garantido compensa o desperdício de memória

Exemplo: duas palavras são *anagramas* se uma é um rearranjo das letras da outra. Basta **contar** as letras de cada uma em duas tabelas de acesso direto (`int` para cada `char`) e comparar: tempo $O(n)$!

Quando usar Acesso Direto?

- Quando o número de chaves **possíveis** M é pequeno: `char`, dias do mês, notas de 0 a 10...
- Ou quando o tempo $O(1)$ garantido compensa o desperdício de memória

Exemplo: duas palavras são *anagramas* se uma é um rearranjo das letras da outra. Basta **contar** as letras de cada uma em duas tabelas de acesso direto (`int` para cada `char`) e comparar: tempo $O(n)$!

Mas e se as chaves forem números de 32 bits? Seriam $M = 2^{32}$ posições (16 GB para `float`), mesmo com só 10 chaves presentes!

E se forem *nomes*? O número de chaves possíveis é praticamente infinito...

Solução: usar um vetor pequeno e uma função que **espalhe** as chaves nele: uma *Tabela de Dispersão*!

Section 3

Tabelas de Dispersão

Tabela de Dispersão

Uma *Tabela de Dispersão* (do inglês *Hash Table*) usa um vetor `H` de tamanho M e uma **função de dispersão** h que leva cada chave a uma posição de 0 a $M-1$:

a chave c fica na posição $H[h(c)]$

Exemplo com $M = 10$ e $h(c) = c \bmod 10$:

```
chaves:  100  30  1  49  11  36  644  77  18

H:  | 100,30 | 1,11 |    |    | 644 |    | 36 | 77 |  18 | 49 |
        0       1     2    3    4     5    6    7     8    9
```

Note que $h(100) = h(30) = 0$, e $h(1) = h(11) = 1$: são **colisões**!

Toda tabela de dispersão precisa decidir: (1) qual **função de dispersão** usar e (2) como **tratar as colisões**.

Funções de Dispersão

Uma boa função de dispersão deve ser:

- **Fácil de calcular:** se calcular h for caro, perdemos a vantagem
- **Uniforme:** para conjuntos de chaves "típicos", cada posição recebe cerca de n/M chaves

Método da Divisão: $h(c) = c \bmod M$. É fácil de calcular, mas nem sempre uniforme:

- se $M = 10$, só o **último dígito** importa: chaves terminadas em 0 colidem todas!
- se M é par, chaves pares vão só para posições pares
- na prática, escolher M **primo** (e longe de potências de 2) funciona bem

Para outros tipos de chave (textos, por exemplo), primeiro transformamos a chave em um número. Em C++, `std::hash` faz isso: `std::hash<std::string>{}("ana")`.

Desafio: ver outras técnicas, como *método da dobra*.

Colisões e Fator de Carga

Colisões são **inevitáveis**: se há mais chaves possíveis que posições, alguma posição recebe mais de uma chave (princípio da casa dos pombos).

O **fator de carga** mede quão cheia a tabela está:

$$\alpha = \frac{n}{M} \qquad \text{(chaves por posição, em média)}$$

Duas famílias de tratamento de colisões:

- **Encadeamento (exterior):** cada posição guarda uma *lista* com as chaves que caíram nela. Usado em `std::unordered_map` e no `HashMap` do Java
- **Endereçamento aberto:** em caso de colisão, procura-se **outra posição** no próprio vetor. Usado no `dict` do Python

Section 4

Encadeamento Exterior

Encadeamento Exterior

Cada posição `H[i]` é uma **lista encadeada** com as chaves c tais que $h(c) = i$:

```
H[0] -> [100] -> [30] -> 0
H[1] -> [ 11] -> [ 1] -> 0
H[2] -> 0
...
H[9] -> [ 49] -> 0
```

Todas as operações começam calculando $i = h(c)$ e depois trabalham **só na lista** `H[i]`:

- `tem` e `consulta`: percorre a lista `H[i]` procurando c
- `adiciona`: se c já está na lista, atualiza o valor; senão, insere **no início** da lista (como no Saco!)
- `remove`: procura c na lista e o desliga

Implementação: Encadeamento Exterior (I)

```
struct NoHash {
   int chave;
   float valor;
   NoHash* proximo;
};
constexpr int M = 13;      // primo!
struct DicionarioHash {
   NoHash* H[M];           // uma lista por posição
   int N;                  // num. de chaves
   int h(int c) { return (c % M + M) % M; } }
   void cria() {
      N = 0;
      for (int i = 0; i < M; i++) H[i] = 0; }
   }
   int tamanho() { return N; }
   ...
};
```

```
// busca interna: só na lista de h(c)
NoHash* encontra(int c) {
   for (auto* p = H[h(c)]; p; p = p->pro
      if (p->chave == c)
         return p;
   return 0;   // não encontrou
}
bool tem(int c) {
   return encontra(c) != 0;
float consulta(int c) {   // pre: tem(c)
   return encontra(c)->valor;
```

Implementação: Encadeamento Exterior (II)

```
void adiciona(int c, float v) {
   auto* p = encontra(c);
   if (p) {                    // já existe:
      p->valor = v;            // atualiza
      return;
   }
   int i = h(c);               // insere no início
   H[i] = new NoHash{.chave = c, .valor = v,
                     .proximo = H[i]};
   N++;
}
```

```
bool remove(int c) {
   int i = h(c);
   NoHash* ant = 0;
   for (auto* p = H[i]; p;
        ant = p, p = p->proximo)
      if (p->chave == c) {
         if (ant) ant->proximo = p->prox
         else     H[i] = p->proximo;
         delete p;   N--;
         return true;
      }
   return false;
}
```

O `libera()` percorre as M listas liberando os nós, como no Saco. Exercício!

Análise: Encadeamento Exterior

O custo de cada operação é: calcular $h(c)$, em $O(1)$, mais percorrer **uma lista**.

- **Pior caso:** todas as chaves caem na mesma posição! A lista tem n nós: $O(n)$
- **Caso médio:** com uma função uniforme, cada lista tem em média $\alpha = n/M$ nós: $O(1+\alpha)$

Se mantivermos M proporcional a n (por exemplo, $\alpha \leq 1$), temos $\alpha = O(1)$ e:

todas as operações custam $O(1)$ em média

Atenção: é um custo **médio**, que depende de a função de dispersão espalhar bem as chaves. Uma escolha ruim (ou um adversário que conhece a função!) pode levar ao pior caso.

Redimensionamento (*rehash*)

Se as chaves continuarem chegando, α cresce e as listas ficam longas. A solução é a mesma do Conjunto Dinâmico:

- quando α passar de um limite (ex: $\alpha > 1$), **dobre** M
- crie a nova tabela e **reinsira** todas as chaves, pois $h(c) = c \bmod M$ muda com M!

Reinserir custa $O(n)$, mas, como dobramos o tamanho, isso acontece **raramente**: o custo **amortizado** da inserção continua $O(1)$ (mesmo argumento do vetor dinâmico!).

Na STL, `std::unordered_map` faz isso automaticamente: `max_load_factor()` é 1,0 por padrão.

Desafio: implemente o `DicionarioHash` com `H` alocado dinamicamente e redimensionamento.

Section 5

Endereçamento Aberto

Endereçamento Aberto

E se não quisermos listas (nem ponteiros)? No **endereçamento aberto**, todas as chaves ficam **no próprio vetor**, e $\alpha \leq 1$.

Em caso de colisão, tentamos outras posições, em uma ordem definida por $h(c, k)$, onde $k = 0, 1, 2, \ldots$ é a tentativa:

- **Tentativa linear:** $h(c, k) = (h(c) + k) \bmod M$ (a próxima posição)
- **Tentativa quadrática:** $h(c, k) = (h(c) + c_1 k + c_2 k^2) \bmod M$
- **Dispersão dupla:** $h(c, k) = (h_1(c) + k \cdot h_2(c)) \bmod M$

A tentativa linear é a mais simples, mas forma **agrupamentos**: quanto maior um bloco ocupado, mais chaves caem nele, e mais ele cresce. As outras espalham melhor.

Tentativa Linear

$$h(c, k) = (h(c) + k) \bmod M \qquad k = 0, 1, 2, \ldots$$

Em caso de colisão, tenta a **próxima** posição. Exemplo com $M = 11$, $h(c) = c \bmod 11$, inserindo 22, 33, 44 e 1:

```
 22: h = 0, tenta 0              -> posição 0
 33: h = 0, tenta 0, 1           -> posição 1
 44: h = 0, tenta 0, 1, 2        -> posição 2
  1: h = 1, tenta 1, 2, 3        -> posição 3   (nem colidiu com 22, 33, 44!)

H:  | 22 | 33 | 44 |  1 |    |    |    |    |    |    |    |
      0    1    2    3    4    5    6    7    8    9   10
```

Agrupamento primário: chaves de posições **diferentes** passam a disputar o mesmo bloco. Quanto maior o bloco, maior a chance de uma nova chave cair nele, e mais ele cresce!

Vantagem: visita **todas** as M posições, e percorre memória contígua (ótimo para *cache*)

Tentativa Quadrática e Dispersão Dupla

$$h(c, k) = (h(c) + k^2) \bmod M$$

Com as mesmas chaves, os saltos crescem ($+1$, $+4$, $+9$, ...) e o bloco se desfaz:

```
 33: h = 0, tenta 0, 1          -> posição 1
 44: h = 0, tenta 0, 1, 4       -> posição 4
  1: h = 1, tenta 1, 2          -> posição 2

H:  | 22 | 33 |  1 |    | 44 |    |    |    |    |    |    |
       0    1    2    3    4    5    6    7    8    9   10
```

Cuidado: a sequência pode **não visitar todas** as posições! Com $M = 11$, $k^2 \bmod 11$ só alcança 6 posições. Com M primo, as primeiras $(M + 1)/2$ tentativas são distintas: mantendo $\alpha \leq 1/2$, sempre há lugar.

Ainda resta o **agrupamento secundário**: chaves com o mesmo $h(c)$ fazem **a mesma** sequência de saltos.

Endereçamento Aberto: e a remoção?

Considere $h(a) = h(b) = 2$, com tentativa linear: a ocupa a posição 2, e b a posição 3.

```
inserir a, b:     | . | . | a | b | . |      remover a:   | . | . | ? | b | .
                    0   1   2   3   4                       0   1   2   3   4
```

Se a posição 2 voltar a ficar **vazia**, a busca por b começa em 2, vê uma posição vazia e **para**: b "some"!

Solução: marcar a posição como **removida** (uma *lápide*). Cada posição tem um de três estados:

- `VAZIO`: a busca **para** aqui
- `OCUPADO`: compara a chave
- `REMOVIDO`: a busca **continua**, mas a inserção pode **reaproveitar** a posição

Implementação: Endereçamento Aberto

```
enum Estado { VAZIO, OCUPADO, REMOVIDO };
struct DicionarioAberto {
   int chaves[M];   float valores[M];
   Estado estado[M];
   int N;
   // posição de c, ou -1 se não existe
   int posicao(int c) {
      for (int k = 0; k < M; k++) {
         int i = (h(c) + k) % M;   // linear
         if (estado[i] == VAZIO) return -1;
         if (estado[i] == OCUPADO &&
             chaves[i] == c) return i;
      } // REMOVIDO: continua procurando!
      return -1;
   }
   ...
};
```

```
void adiciona(int c, float v) { // pre:
   int p = posicao(c);
   if (p != -1) { valores[p] = v; return
   for (int k = 0; k < M; k++) {
      int i = (h(c) + k) % M;
      if (estado[i] != OCUPADO) { // rea
         chaves[i] = c;  valores[i] = v;
         estado[i] = OCUPADO;  N++;
         return;
      }
   }
}
bool remove(int c) {
   int p = posicao(c);
   if (p == -1) return false;
   estado[p] = REMOVIDO;  N--;   // lápi
   return true;
}
```

Encadeamento × Endereçamento Aberto

	Encadeamento Exterior	Endereçamento Aberto
Onde ficam as colisões	listas fora do vetor	no próprio vetor
Fator de carga α	pode passar de 1	sempre $\alpha \leq 1$
Memória extra	um ponteiro por chave	nenhuma (mas sobra espaço livre)
Remoção	simples (desliga da lista)	precisa de lápides
Desempenho com α alto	degrada suavemente	degrada rapidamente
Uso de *cache*	ruim (nós espalhados)	ótimo (memória contígua)

Ambos têm custo **médio** $O(1)$ com α limitado, e pior caso $O(n)$.

Curiosidade: existe ainda o *encadeamento interior*, um meio-termo em que as listas usam posições do próprio vetor.

Section 6

Dicionários: Hash × Árvores

Uma Limitação Importante

As tabelas de dispersão **espalham** as chaves de propósito: chaves vizinhas, como 100 e 101, vão para posições sem nenhuma relação.

Por isso, perguntas sobre a **ordem** das chaves ficam caras:

- qual a menor (ou a maior) chave? $O(M + n)$: olha tudo!
- qual a primeira chave ≥ 50? $O(M + n)$
- quais chaves estão entre 10 e 20? $O(M + n)$
- percorrer em ordem crescente? Precisa **ordenar**: $O(n \log n)$

Lembra do Saco Ordenado? Lá, essas perguntas custavam $O(\log n)$, com `buscaMaiorIgual` e `buscaMaior`, mas inserir custava $O(n)$...

Uma Limitação Importante

As tabelas de dispersão **espalham** as chaves de propósito: chaves vizinhas, como 100 e 101, vão para posições sem nenhuma relação.

Por isso, perguntas sobre a **ordem** das chaves ficam caras:

- qual a menor (ou a maior) chave? $O(M + n)$: olha tudo!
- qual a primeira chave ≥ 50? $O(M + n)$
- quais chaves estão entre 10 e 20? $O(M + n)$
- percorrer em ordem crescente? Precisa **ordenar**: $O(n \log n)$

Lembra do Saco Ordenado? Lá, essas perguntas custavam $O(\log n)$, com `buscaMaiorIgual` e `buscaMaior`, mas inserir custava $O(n)$...

Para onde vamos: nas próximas aulas, as **Árvores Balanceadas** darão $O(\log n)$ para **tudo**, e com elas construiremos o **Dicionário Ordenado**, que responde às perguntas de ordem.

Dicionários na Biblioteca Padrão

```
import std;

int main() {
   std::unordered_map<std::string, int> idade;  // tabela de dispersão
   idade["ana"] = 20;                            // adiciona
   idade["bia"] = 22;
   idade["ana"] = 21;                            // atualiza
   println("{}", idade.contains("ana"));         // true
   println("{}", idade.at("ana"));               // 21
   idade.erase("bia");                           // remove
   println("{}", idade.size());                  // 1

   std::map<std::string, int> ordenado;          // árvore balanceada
   // mesmas operações, mas percorre as chaves em ordem!
   return 0;
}
```

Exercícios

(Adaptados do material do prof. Fabiano Oliveira.)

1. **Anagramas:** dadas duas palavras com n letras, determine se são anagramas em tempo $O(n)$, usando tabelas de acesso direto.
2. **Soma K:** dado um vetor com n inteiros e um valor K, determine se existem dois elementos que somam K, em tempo médio $O(n)$.
3. **Repetidos:** dado um vetor com n inteiros, determine se há elementos repetidos em tempo médio $O(n)$.
4. **Frequência de palavras:** leia um texto e conte quantas vezes cada palavra aparece, em tempo médio $O(n)$, onde n é o número de caracteres.
5. **Nota de resgate:** uma nota de resgate é escrita com recortes de um texto fonte. Determine se é possível montar a nota a partir do texto, em tempo médio $O(n_A + n_B)$, quando: (a) podemos recortar letras individuais; (b) podemos recortar apenas palavras inteiras.
6. Implemente o `libera()` do `DicionarioHash`, e uma versão do `DicionarioAberto` com **tentativa quadrática**.

Dicionários e Tabelas de Dispersão: Revisão Geral

- Quais são as operações de um Dicionário? Qual delas tem pré-condição?
- Quando usar uma Tabela de Acesso Direto? Qual o seu custo de espaço?
- O que é uma colisão? Por que elas são inevitáveis?
- O que é o fator de carga α? Por que ele precisa ser limitado?
- Por que o custo $O(1)$ das tabelas de dispersão é apenas **em média**?
- Por que o endereçamento aberto precisa de lápides na remoção?
- Por que tabelas de dispersão não respondem bem a perguntas sobre a **ordem** das chaves?

Bibliografia Recomendada

Além da bibliografia do curso, recomendamos para esse tópico:

- Szwarcfiter, J.L; Markenzon, L. Estruturas de Dados e seus Algoritmos. Rio de Janeiro, LTC, 1994.

Bibliografia Adicional:

- Oliveira, F. Algoritmos e Estruturas de Dados I: Tabelas de Dispersão (slides), IME-UERJ.
- Cerqueira, R.; Celes, W.; Rangel, J.L. Introdução a estruturas de dados: com técnicas de programação em C. Editora, 2004.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Introduction to Algorithms, 3rd ed.. The MIT Press, 2009.
- Sedgewick, R.; Wayne, K. Algorithms, 4th Edition. Addison-Wesley, 2011.
- Knuth, D.E. The Art of Computer Programming - Vol III. 2nd Edition. Addison Wesley, 1998.
- Livro "The C++ Programming Language" de Bjarne Stroustrup
- Dicas e normas C++: https://github.com/isocpp/CppCoreGuidelines

Section 7

Agradecimentos

Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Fabiano Oliveira (IME-UERJ), cujo material de Tabelas de Dispersão forma a base desta aula, e o prof. Jayme Szwarcfiter cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de Pesquisa Operacional, que abriu caminho para verificação prática dessa tecnologia de slides.

Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- groomit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

- github
- gitlab
- microsoft
- google
- ...

Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

- https://igormcoelho.github.io/ilectures-pandoc/

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

Licença: CC-BY 4.0 2020

Igor Machado Coelho

This Slide Is Intentionally Blank (for goomit-mpx)