aula 10: Tabelas de Dispersão

Complexidade das estruturas usadas:

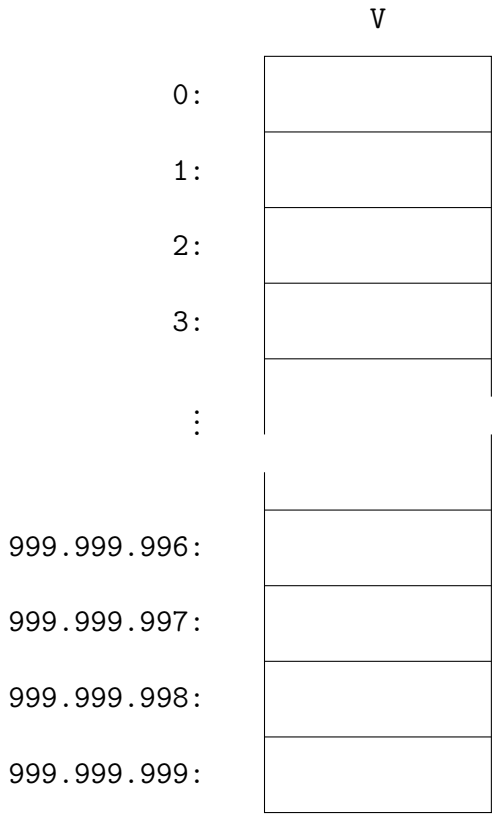
pior caso	inserção	busca	remoção
vetor não-ordenado	$O(n)$	$O(n)$	$O(n)$
vetor ordenado	$O(n)$	$O(\log_2 n)$	$O(n)$
lista encadeada	$O(1)$	$O(n)$	$O(n)$
lista encadeada ordenada	$O(n)$	$O(n)$	$O(n)$

Endereçamento direto

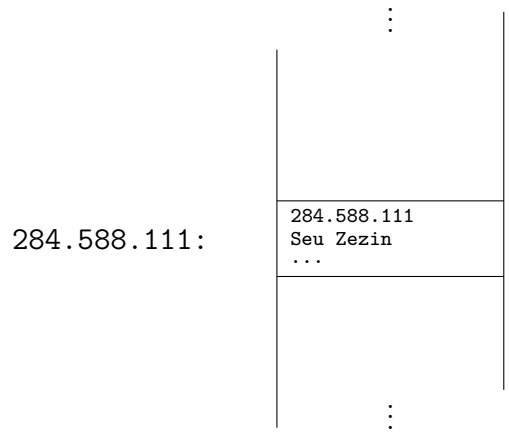
```
class Registro {
        public:
        int chave;
        char nome[100];
        ...
};
```

Imagine que queiramos manter um conjunto de registros. Imagine ainda que o valor de chave pode variar entre 0 e 999.999.999.

Uma possibilidade seria criar um imenso vetor de tamanho 1.000.000.000 e utilizar a própria chave do registro como a posição onde o registro deve ser colocado.



Por exemplo, um registro cuja chave é 284.588.111 será colocado na posição 284.588.111:



Essa estratégia é chamada de **endereçamento direto**. Neste esquema, a inserção, busca e remoção podem ser implementadas das seguintes formas:

```
void insercao(Registro reg){
    int pos = reg.chave;  //posição onde o registro deve ficar no vetor V
    V[pos] = reg;
}
```

```
Registro busca(int chave){
    return V[chave];
}
```

```
void remocao(int chave){
    int pos = reg.chave;  //posição onde o registro deve estar no vetor V
    V[pos].chave = -1;  //valor -1 indica que a posição não está ocupada
}
```

Fazendo dessa forma, as três operações podem ser realizadas em tempo $O(1)$:

	pior caso	inserção	busca	remoção
endereçamento direto		$O(1)$	$O(1)$	$O(1)$

A velocidade com que as operações são realizadas é muito boa, no entanto, esse método tem uma clara desvantagem, já que uma quantidade enorme de espaço precisa ser previamente alocado. Imagine que a quantidade de valores de chaves possíveis é tão grande que o tamanho do vetor não coubesse na sua memória disponível. Ou, mesmo que coubesse, que a sua aplicação precisasse manter a cada momento apenas uma quantidade pequena de registros (digamos 1000, por exemplo).

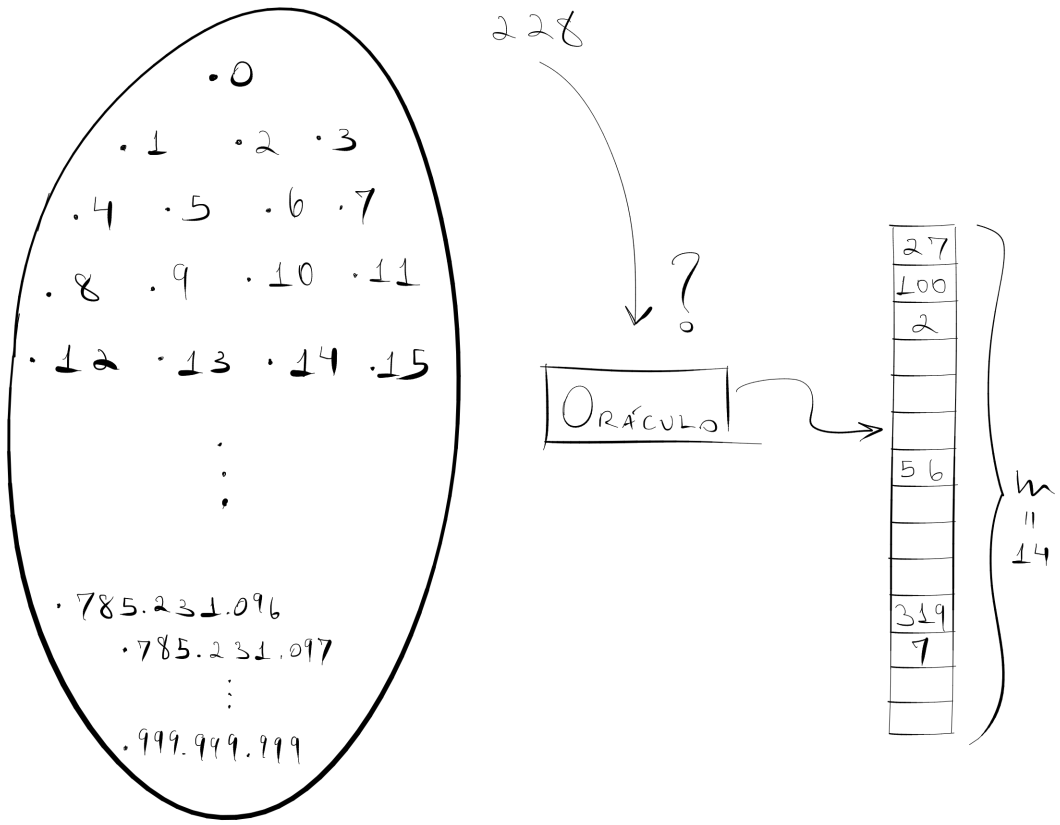
Nesse cenário, poderíamos usar alguma das estruturas que já vimos antes (vetores ou listas encadeadas). No entanto, a diferença entre os tempos de execução é bastante relevante.

A questão que se impõe é: não seria possível manter a ótima complexidade do endereçamento direto sem ter que reservar tanto espaço?

A primeira observação é que o que permite a ótima complexidade do endereçamento direto é que a posição exata em que o registro deve ficar já é conhecida, pois corresponde à chave.

Mas, mesmo que a posição onde um registro deveria ficar não fosse exatamente o mesmo número de sua chave, se houvesse uma maneira de descobrirmos a posição onde ele deveria ficar, poderíamos realizar essas operações em tempo constante.

Imagine então a situação onde o universo de chaves possíveis é extremamente grande, mas a quantidade de registros mantidos a cada momento é bem menor. Seja m a quantidade máxima de elementos que manteremos a cada momento.



Se usarmos um vetor para armazenar registros, quando queremos inserir um registro novo o principal esforço realizado é para encontrar uma posição livre para armazenaro novo registro.

Se houvesse uma maneira (um oráculo?) que nos informasse rapidamente uma posição livre, poderíamos simplesmente copiar o registro para esta posição. Os algoritmos de inserção, remoção e busca ficariam assim:

```
void inserir(Registro reg){
    int pos = Oraculo(reg.chave);

    tabelaHash[pos] = reg;
}
```

```
void remover(Registro reg){
    int pos = Oraculo(reg.chave);

    tabelaHash[pos].chave = -2;  //chave igual a -2 indicará que a
                                 // posição está vaga
}
```

```
int buscar(Registro reg){    //retorna a posição do registro na tabela
    return Oraculo(reg.chave);
}
```

Bem, não dispomos de um oráculo em nossa linguagem de programação. Então poderíamos tentar advinhar de outra forma.

As tabelas de dispersão (hash tables) substituem o oráculo na figura acima por uma função de dispersão (hash function) que tenta advinhar uma posição livre. Vejamos um exemplo de função de dispersão.

— *Método da divisão*

Um dos primeiros problemas que uma função de dispersão deve resolver é como mapear as chaves possíveis em um intervalo $\{0, 1, \ldots, m-1\}$.

Uma forma simples e intuitiva é dividir o conjunto de chaves de acordo como resto da divisão por m.

```
int metodo_divisao(int chave, int m){
    return chave % m;
}
```

Basta então substituir o nosso "oráculo" pela nossa função de dispersão:

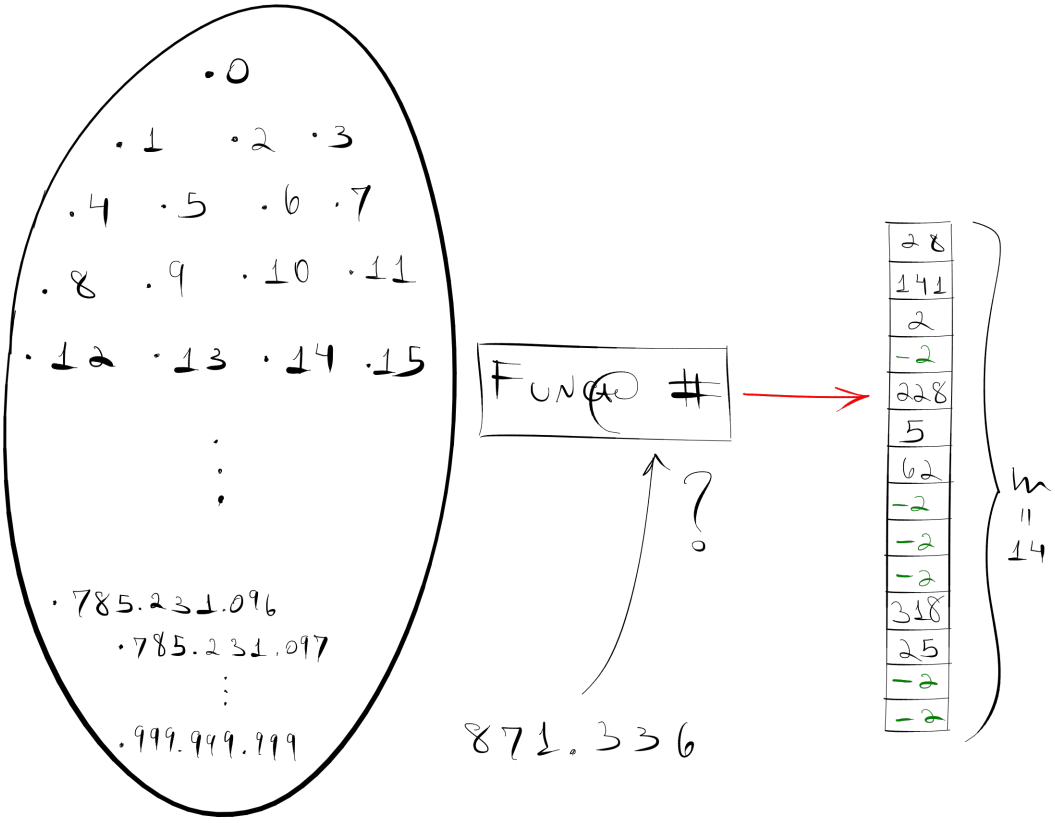
```
void inserir(Registro reg){
    int pos = metodo_divisao(reg.chave);

    tabelaHash[pos] = reg;
}
```

```
void remover(Registro reg){
    int pos = metodo_divisao(reg.chave);

    tabelaHash[pos].chave = -2;  //chave igual a -2 indicará que a
                                 // posição está vaga
}
```

```
int buscar(Registro reg){    //retorna a posição do registro na tabela
    return metodo_divisao(reg.chave);
}
```

Como uma mera função não é mágica, às vezes ela "falha", isto é, diz para inserirmos um registro em uma posição que já está ocupada.

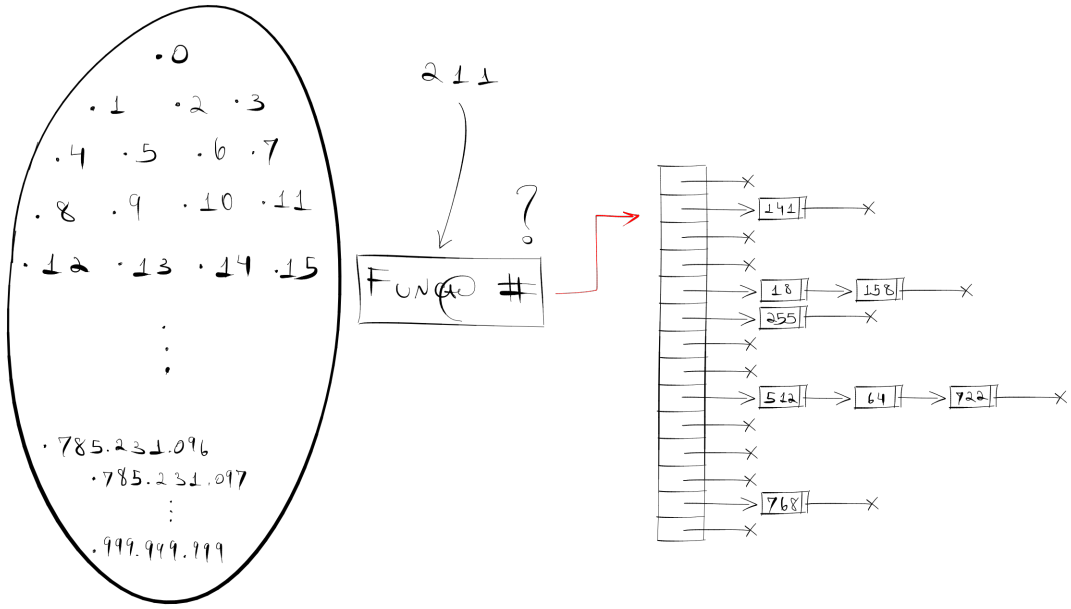
Na verdade, a função hash, ou função de dispersão, tenta espalhar as chaves pelas m posições da tabela de forma a minimizar a chance de colisão. Mesmo assim, podem haver colisões e temos que tratá-las.



Existem duas formas amplamente usadas de se tratar as colisões: *encadeamento externo* e *endereçamento aberto*.

Colisões: encadeamento externo

No encadeamento externo, nós associamos a cada posição da tabela não um registro, mas uma lista encadeada de registros. Nesse caso, a tabela não será mais uma tabela de registros, mas uma tabela de ponteiros para listas encadeadas cujos nós são registros.



Em caso de colisão, isto é, caso a função de dispersão indique uma posição ocupada, nós iserimos o novo registro na lista encadeada. Afim de realizar a inserção da forma mais rápida possível, iserimos o registro no início da lista encadeada.

Como os registros serão armazendos em lista encadeada, modelaremos nossos registros como nós de lista encadeada:

```
class Registro {
        public:
        int chave;
        ⋮
        Registro* prox;
};
```

Uma lista encadeada é um ponteiro para o primeiro nó da lista. Como cada posição da tabela será na realidade uma lista encadeada, nossa tabela de dispersão será então um vetor de ponteiros para registros.

O seguinte código cria a tabela:

```
int m;   //tamanho da tabela
...
Registro** tabela = new Registro*[m];

for (int i = 0; i < m; i++)
    tabela[i] = NULL;
```

A inserção na tabela fica assim:

```
int inserir(Registro* reg) {   //retorna a posição na tabela
    int pos = metodo_divisao(reg.chave, m);

    if (tabela[pos] == NULL) {   //posição está livre
        tabela[pos] = reg;
    }
    else {
        reg->prox = tabela[pos];
        tabela[pos] = reg;
    }

    return pos;
}
```

No caso da busca, temos que usar a função de dispersão para ter acesso à lista de elementos que ficam em uma certa posição. Daí em diante, temos que realizar uma busca em lista encadeada.

```
Registro* busca(int c) {   //retorna ponteiro para o registro buscado
    int pos = metodo_divisao(c, m);

    Registro* aux = tabela[pos];
    while (aux != NULL) {
        if (aux->chave == c) {
            return aux;
        }

        aux = aux->prox;
    }

    return NULL;
}
```

A remoção fica assim:

```
void remocao(int c) {
    int pos = metodo_divisao(c, m);

    if (tabela[pos] == NULL) {
        return NULL;
    }
    else {
        if (tabela[pos]->chave == c){
            Registro* h = tabela[pos];
            tabela[pos] = tabela[pos]->prox;
            delete h;
        }
        else {
            Registro* aux = tabela[pos];
            while (aux->prox != NULL){
                if (aux->prox->chave == c) {
                    Registro* h = aux->prox;
                    aux->prox = aux->prox->prox;
                    delete h;
                    break;
                }

                aux = aux->prox;
            }
        }
```

```
    }
}
```

Outra função de dispersão

— *Método da Multiplicação*

Para o método da multiplicação, mapeamos a chave no intervalo $[0, 1)$ e então multiplicamos por m. O mapeamento é feito multiplicando a chave por um número entre 0 e 1, e então tomando a parte fracionária.

```
int metodo_multiplicacao(int chave, int m){
    int partint;
    int partfrac;

    float A = 0.6180339887;

    partfrac = modf(chave * A, &partint);
    modf(m*partfrac, &partint);   //reaproveitando a variável partint...

    return partint;
}
```

Colisões: Endereçamento aberto

Imagine que haja uma colisão. Nesse caso, nosso "oráculo" falhou e temos que dar um jeito nós mesmos de inserir o elemento. Uma possibilidade seria simplesmente sair percorrendo a tabela em alguma direção até encontrar uma posição livre e colocar nosso registro. Esse método é chamado de *tentativa linear*.

Assim, o método de inserção ficaria assim:

```
void inserir(Registro reg){
    int pos = metodo_divisao(reg.chave);

    if (tabelaHash[pos].chave < 0){   //nesse caso a posição está livre
        tabelaHash[pos] = reg;
    }
    else {
        for (int i =  1; i < m; i++){
            if (tabelaHash[(i + pos) % m].chave < 0){   //encontrou posição livre
                tabelaHash[(i + pos) % m] = reg;
                break;
            }
        }
    }
}
```

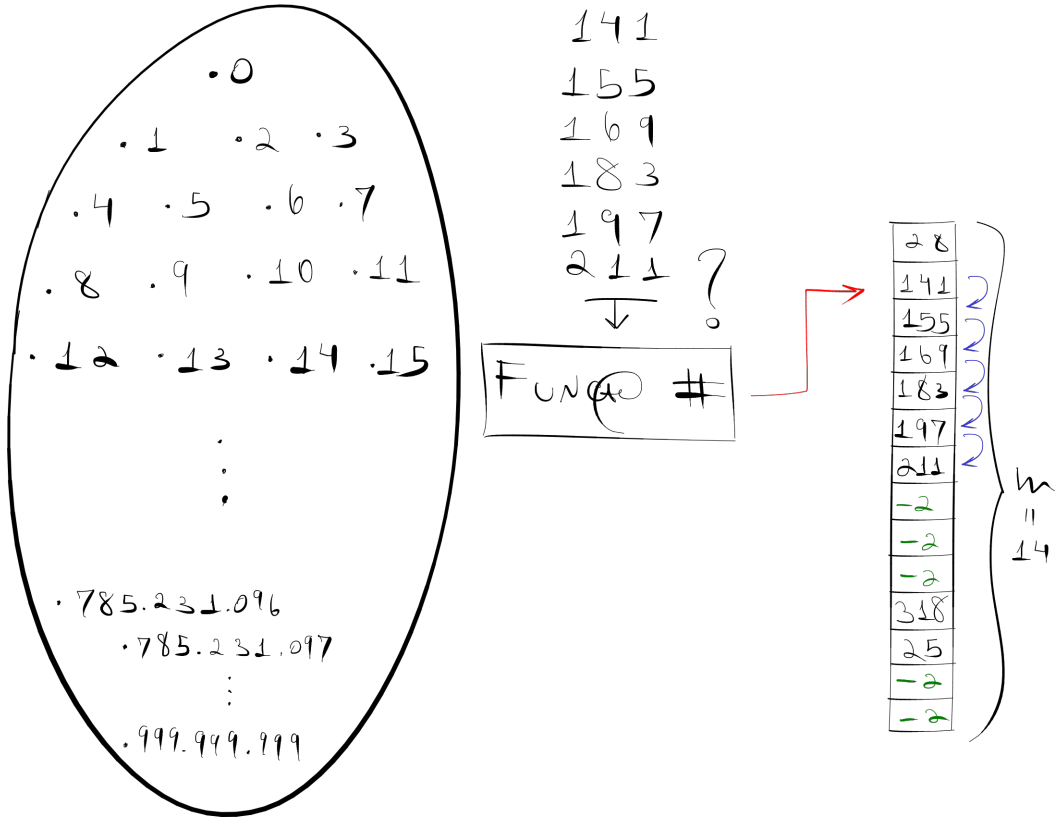
As colisões também afetam as buscas. Se um registro sofreu colisão no momento da inserção, então ele não estará na posição onde deveria estar. Nesse caso, temos que procurá-lo. Como nossa estratégia de inserção é percorrer o restante da lista até encontrar uma posição vazia, podemos repetir esse trajeto a fim de encontar o registro desejado.

Note no entanto que nem sempre precisamos percorrer a tabela inteira. Nós convencionamos que posições vazias são posições onde a chave do registro é -2. Nesse caso, assim que encontrarmos uma posição vazia podemos parar a busca.

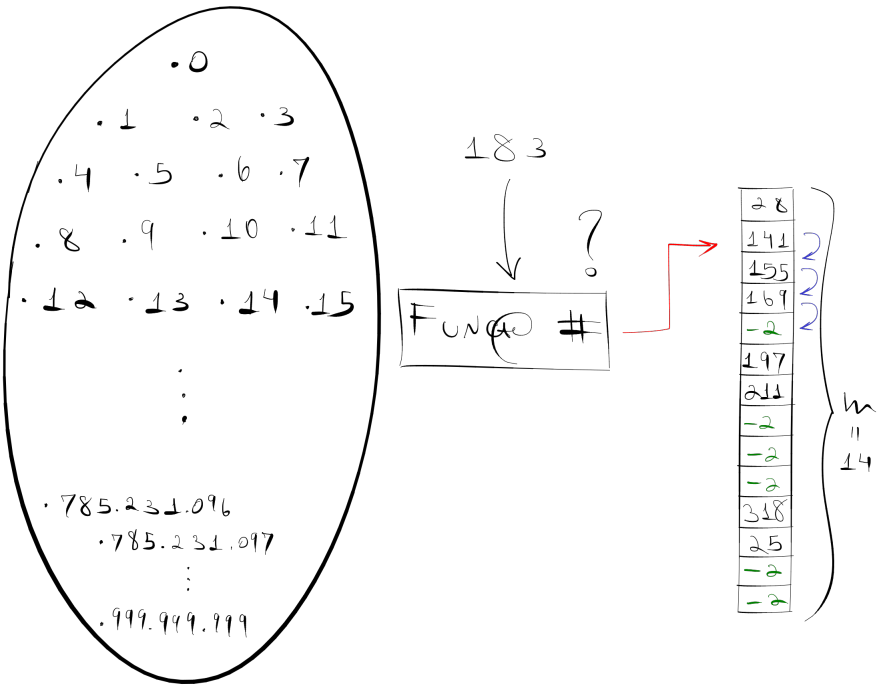
```
int busca(int chave){    //retorna a posição do registro na tabela
    int pos = metodo_divisao(reg.chave);

    if (tabelaHash[pos].chave == chave){   //nesse caso a posição está livre
        return pos;
    }
    else {
        for (int i = 1; i < m; i++){
            if (tabelaHash[(i + pos) % m].chave == chave){   //encontrou a chave
                return i;
            }
            else if (tabelaHash[(i + pos) % m].chave == -2){   //significa que o
                                                    //elemento não está
                                                    //na tabela
                return -1;
            }
        }
    }
}
```

Já o caso da deleção é semelhante à busca, mas há um detalhe. Imagine uma situação em que inserimos vários registros que a função de dispersão levou para a mesma posição. No nosso exemplo, se tentássemos inserir registros com as chaves $141, 155, 169, 183, 197, 211$, o método da divisão nos indica a posição 1. Se resolvermos as colisões usando tentativa linear, colocaríamos,

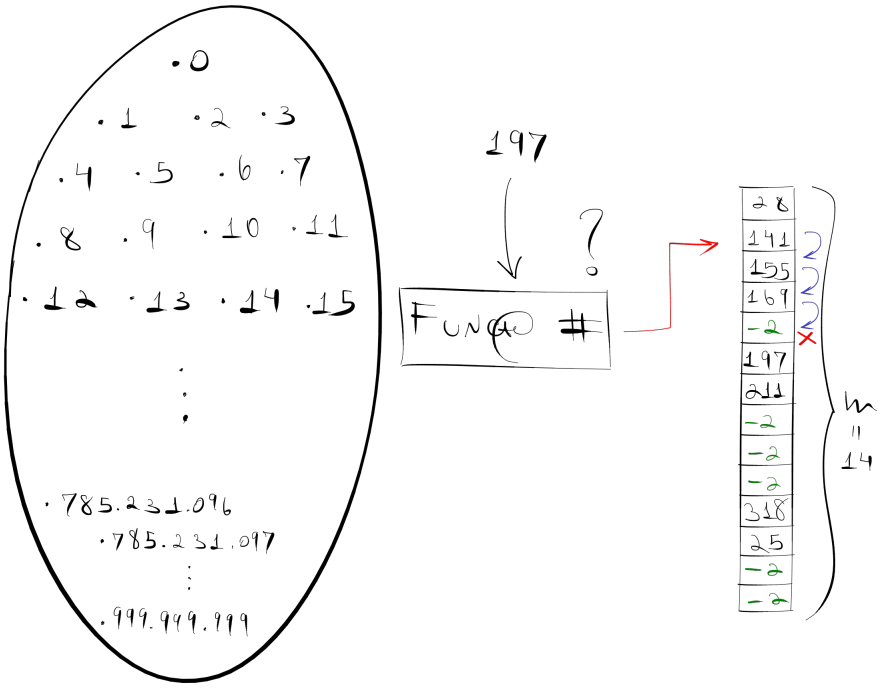
na medida do possível, os registros nas posições subsequentes.



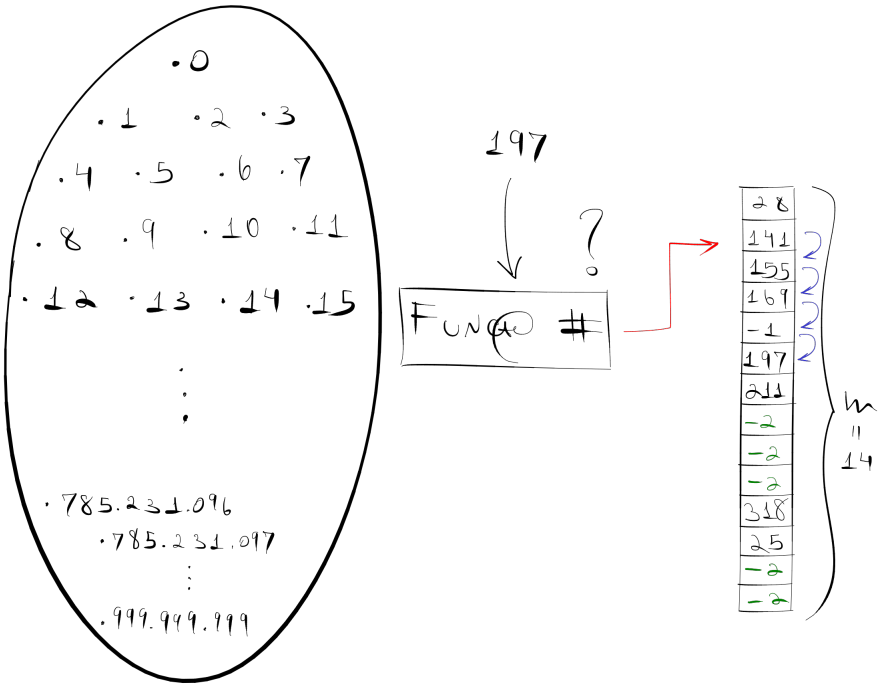
Imagine agora que queiramos remover o registro de chave 183. Nesse caso a função retornaria a posição 1 e, havendo outro registro no lugar, passaríamos a percorrer o restante da tabela sequencialmente. Como a sinalização de posição vaga se dá pelo valor de chave -2, assim que achássemos esse registro, alteraríamos a sua chave para -2. Ficaríamos assim:



Agora imagine que logo em seguida fizéssemos uma busca pelo registro de chave 197. A função de dispersão nos retornaria a posição 1, e percorreríamos as posições subsequentes para procurar a chave desejada. No entanto, ao chegar na posição 4, verificaríamos que a posição está marcada com chave -2 e interromperíamos a busca sem remover o registro 197.



Uma maneira de resolver esse problema é encontrar uma forma de indicar a possibilidade de haver mais registros logo em seguida. Por exemplo, podemos usar o valor de chave -1 para indicar que naquela posição houve um registro em algum momento mas que foi apagado e portanto em caso de busca ainda é necessário continuar procurando.



O método de remoção fica assim:

```
void remocao(int chave){
    int pos = metodo_divisao(reg.chave);

    if (tabelaHash[pos].chave < 0){   //nesse caso a posição está livre
        tabelaHash[pos].chave = -1;
    }
    else {
        for (int i = pos + 1; i % m != pos % m; i++){
            if (tabelaHash[i].chave = chave){   //encontrou o registro
                tabelaHash[i].chave = -1;    //marca como apagado
                break;
            }
            else if (tabelaHash[i].chave = -2){   //nesse caso, não houve
                                                  //mais inserções neste
                                                  //trajeto
                break;
            }
        }
    }
}
```

Tente outra vez...

Em caso de colisão, nós precisamos procurar uma posição vaga para colocar o elemento no caso de inserção. Nós vimos o caso da *tentativa linear*. Vejamos outros:

— *Tentativa por Salto Primo*

Para o método do salto primo, escolhemos um número p primo (ou de forma que p e m sejam primos entre si) e vasculhamos a tabela pulando de p-em-p posições, vejamos como fica a inserção

```
void inserir(Registro reg){
    int pos = metodo_divisao(reg.chave);

    if (tabelaHash[pos].chave < 0){   //nesse caso a posição está livre
        tabelaHash[pos] = reg;
    }
    else {
        for (int i =  1; i < m; i++){
            if (tabelaHash[(i*p + pos) % m].chave < 0){   //posição livre
                tabelaHash[(i*p + pos) % m] = reg;
                break;
            }
        }
    }
}
```

— *Tentativa quadrática*

No método da tentativa quadrática, saltamos, a partir da posição inicial dada pela função hash, usando uma função quadrática do número de tentativas.

Ou seja, inicialmente, tentamos inserir na posição dada pela função de dispersão. Se a posição estiver ocupada faremos mais tentativas até encontrar uma posição livre. A posição visitada na i-ésima tentativa é

$$(pos + c_1 * i + c_2 * i^2) \textbf{ mod } m$$

onde *pos* é a posição inicial determinada pela função de dispersão. A insersão ficaria assim, para o caso de $c_1 = c_2 = \frac{1}{2}$:

```
void inserir(Registro reg){
    int pos = metodo_divisao(reg.chave);

    if (tabelaHash[pos].chave < 0){   //nesse caso a posição está livre
        tabelaHash[pos] = reg;
    }
    else {
        for (int i =  1; i < m; i++){
            if (tabelaHash[(pos + 0.5*i + 0.5*i*i) % m].chave < 0){   //posição livre
                tabelaHash[(pos + 0.5*i + 0.5*i*i) % m] = reg;
                break;
            }
        }
    }
}
```