Tabelas de dispersão

Algoritmos e Estruturas de Dados

2020/2021

FEUP

Tabela de dispersão

- Uma tabela de dispersão é um vetor de tamanho fixo em que os elementos são colocados em uma posição determinada por uma função denominada **função de dispersão**.
- A função de dispersão deve:
  - ser fácil de calcular
  - distribuir os objetos uniformemente pela tabela
- Vários objetos podem ser mapeados numa mesma posição: **colisão**
- O comportamento das tabelas de dispersão é caraterizado por:
  - função de dispersão
  - técnica de resolução de colisões
- As tabelas de dispersão asseguram tempo médio constante para inserção, remoção e pesquisa

Tabelas de dispersão

Ilustração do conceito

Função de dispersão:

F(x) = comprimento(x)%10;

Nome	*F(Nome)*
Carlos	6
Rodrigo	7
Artur	5
Ana	3
Miguel	6
Clementina	0
Aristófanes	1

0	Clementina
1	Aristófanes
2	
3	Ana
4	
5	Artur
6	Carlos / Miguel
7	Rodrigo
8	
9	

É uma má função de dispersão, porque tem grandes probabilidades de levar a muitas colisões

Tabelas de dispersão

- Função de dispersão

A função de dispersão envolve o comprimento da tabela para assegurar que os resultados estão dentro da gama pretendida

```
int hash (const char* key, int tableSize) {
   int hashVal = 0;
   for ( int i = 0; i < key ; i++ )
      hashVal = 37*hashVal + key[i];
   hashVal %= tableSize;
   if (hashVal < 0 ) hashVal += tableSize;
   return hashVal;
}
```

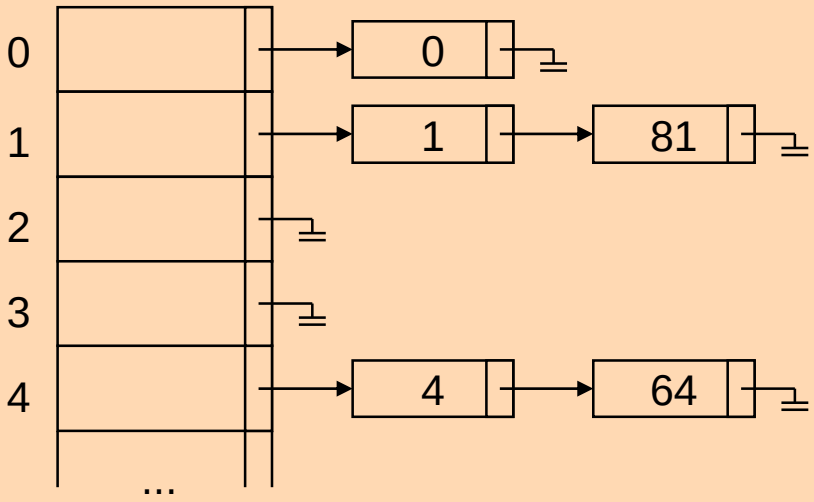
```
int hash (int key, int tableSize) {
   if ( key < 0 ) key = -key;
   return key%tableSize;
}
```

A qualidade da função de dispersão depende também do tamanho da tabela: tamanhos primos são os melhores.

Tabelas de dispersão

- Resolução de colisões por listas

Os elementos mapeados na mesma posição são guardados numa lista ligada



hash(x) = x % 10

Nota: no exemplo, tamanho tabela =10
apenas para simplificação de cálculos (pois este deve ser um nº primo)

Tabelas de dispersão

- Resolução de colisões por listas

O desempenho pode ser medido pelo número de sondagens efetuadas. Este depende do fator de carga λ

λ = número de elementos presentes na tabela / tamanho da tabela

  - Comprimento médio de cada lista é λ
  - Tempo médio de pesquisa (número de sondagens)
    - Pesquisa sem sucesso: λ
    - Pesquisa com sucesso : 1 + λ/2

Tabelas de dispersão

- Resolução de colisões com dispersão aberta
  - Quando ocorre uma colisão procura-se uma célula alternativa, sondando sequencialmente as posições $H_1(x)$, $H_2(x)$, …, até se encontrar uma posição livre.
  - $H_i(x) = (hash(x) + f(i)) \bmod tableSize$
  - **Sondagem linear** : $f(i) = i$

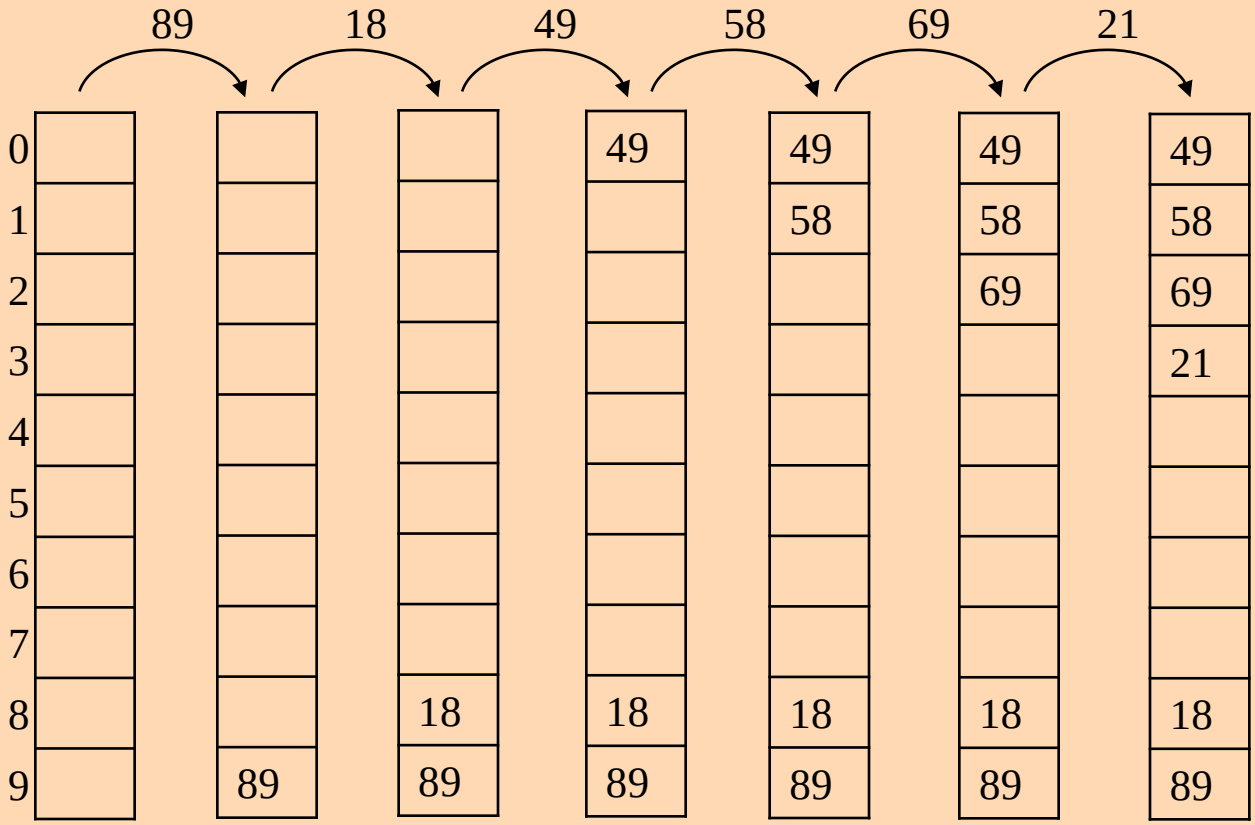
    Garante a utilização completa da tabela
  - **Sondagem quadrática** : $f(i) = i^2$

    Pode ser impossível inserir um elemento numa tabela com espaço

    Evita o fenómeno da agregação primária

Tabelas de dispersão

- Sondagem linear



hash(x) = x % 10 ; H(x) = (hash(x) + i) % 10

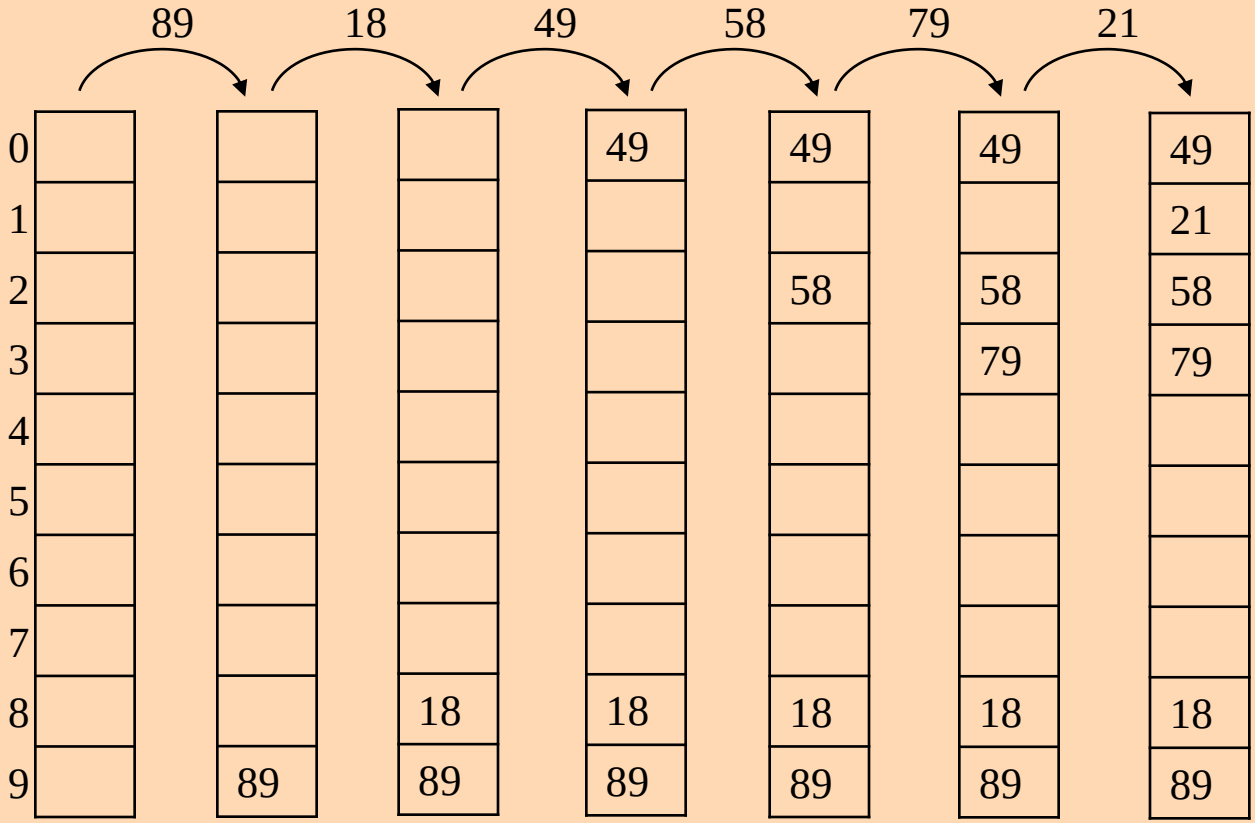
Nota: no exemplo, tamanho tabela =10
apenas para simplificação de cálculos (pois este deve ser um nº primo)

Tabelas de dispersão

- Sondagem linear
  - Utiliza toda a tabela : $0 \leq \lambda \leq 1$
  - Suscetível ao fenómeno de *agregação primária*
  - Número médio de sondagens
    - inserção / pesquisa sem sucesso: $\frac{1}{2} (1 + 1 / (1- \lambda)^2)$
    - pesquisa com sucesso : $\frac{1}{2} (1 + 1 / (1- \lambda))$
  - Número médio de sondagens no caso ideal (sem agregação)
    - inserção / pesquisa sem sucesso: $1 (1- \lambda)$
    - pesquisa com sucesso : $1/\lambda \ \ln(1/(1- \lambda))$
  - Sondagem quadrática elimina o fenómeno de agregação primária

Tabelas de dispersão

- Sondagem quadrática



hash(x) = x % 10 ; H(x) = (hash(x) + i^2) % 10

Nota: no exemplo, tamanho tabela =10
apenas para simplificação de cálculos (pois este deve ser um nº primo)

Tabelas de dispersão

- Sondagem quadrática
  - Não garante que se encontre sempre uma posição livre para um dado elemento

    > Quando o tamanho da tabela é primo, e se usa sondagem quadrática, é sempre possível inserir um elemento se a tabela não estiver preenchida a mais de 50%

  - Desempenho aproxima-se do caso ideal sem agregação
  - A geração de posições alternativas na sondagem quadrática pode ser realizada com apenas uma multiplicação : $H_i = (H_{i-1} + 2i - 1) \mod TableSize$

Tabelas de dispersão: endereçamento aberto

- Declaração da classe ***HashTable*** (secção pública)

  (uma implementação)

```
template <class T> class HashTable
{
public:
   explicit HashTable (const T & notFound, int size = 101);
   HashTable(const HashTable & ht) ;
   const T & find (const T & x) const;
   void makeEmpty();
   void insert(const T & x);
   void remove(const T & x);
   const HashTable & operator= (const HashTable & ht);
   enum EntryType { ACTIVE, EMPTY, DELETED };

private:
   // ...
};
```

Tabelas de dispersão: endereçamento aberto

- Declaração da classe ***HashTable*** (secção privada)

```
template <class T> class HashTable
{  // ...
private:
   struct HashEntry {
      T element;
      EntryType info;
      HashEntry(const T &e = T(), EntryType i = EMPTY):
                                 element(e), info(i) {}
   };

   vector<HashEntry> array;
   int currentSize;
   const T ITEM_NOT_FOUND;

   bool is Active(int currentPos) const;
   int findPos(const T & x) const;
   void rehash();
};
```

Tabelas de dispersão: endereçamento aberto

- classe ***HashTable*** : construtores, esvaziar

```
template <class T>
HashTable<T>::HashTable(const T & notFound, int size):
      ITEM_NOT_FOUND(notFound), array(nextPrime(size))
{}
```

```
template <class T>
HashTable<T>::HashTable(const HashTable & ht):
          ITEM_NOT_FOUND(ht.ITEM_NOT_FOUND),array(ht.array),
          currentSize(ht.currentSize)
{}
```

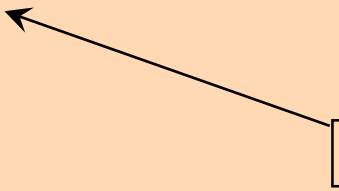
```
template <class T>
void HashTable<T>::makeEmpty()
{
   currentSize = 0;
   for ( int i = 0; i < array.size(); i++ )
      array[i].info = EMPTY;
}
```

Tabelas de dispersão: endereçamento aberto

- classe ***HashTable*** : pesquisa

```
template <class T> const T & HashTable<T>::find(const T & x) const
{
   int currentPos = findPos(x);
   if ( isActive(currentPos) )
      return array[currentPos].element;
   else return ITEM_NOT_FOUND;
}
```

```
template <class T> int HashTable<T>::findPos(const T & x) const
{
   int collisionNum = 0;
   int currentPos = hash(x, array.size());
   while( array[currentPos].info != EMPTY &&
                        array[currentPos].element != x ) {
      currentPos += 2 * ++collisionNum - 1;
      if ( currentPos >= array.size() )
         currentPos -= array.size();
   }
   return currentPos;
}
```



Tabelas de dispersão: endereçamento aberto

- classe ***HashTable*** : inserção

```
template <class T> void HashTable<T>::insert(const T & x)
{
   int currentPos = findPos(x);
   if ( isActive(currentPos) ) return;
   array[currentPos] = HashEntry(x, ACTIVE);
   if ( ++currentSize > array.size()/2 ) rehash();
}
```

pode ser necessário rehash

```
template <class T> void HashTable<T>::rehash()
{
   vector<HashEntry> oldArray = array;
   array.resize(nextPrime(2 * oldArray.size()));
   for( int j = 0; j < array.size(); j++ )
      array[j].info = EMPTY;
   currentSize = 0;
   for( int i = 0; i < oldArray.size(); i++ )
      if ( oldArray[i].info == ACTIVE )
         insert(oldArray[i].element);
}
```

Tabelas de dispersão: endereçamento aberto

- classe ***HashTable*** : remoção

```
template <class T>
void HashTable<T>::remove(const T & x)
{
   int currentPos = findPos(x);
   if ( isActive(currentPos) )
      array[currentPos].info = DELETED;
}
```

não "retira" elemento da tabela

```
template <class T>
bool HashTable<T>:: isActive(int currentPos) const
{
   return ( array[currentPos].info == ACTIVE );
}
```

Tabelas de Dispersão (Standard Template Library - STL)

- class ***unordered_set***

  unordered_set<T, HashFunc, EqualFunc>

- Alguns métodos:
  - pair<iterator,bool> insert(const T & x)
    - valor de retorno:
      - iterador para o elemento adicionado (ou o elemento que não permitiu a inserção)
      - bool: se inserção foi efetuada ou não
  - iterator erase(iterator it)
    - valor de retorno: iterador para o elemento a seguir ao removido
  - iterator find(const T & x) const
  - iterator begin()
  - iterator end()
  - bool empty() const
  - void clear()

Tabelas de dispersão: aplicação

- Contagem de palavras diferentes

Pretende-se escrever um programa que leia um ficheiro de texto e indique o número de palavras diferentes nele existentes e quais são essas palavras.

- Usar uma tabela de dispersão, onde são guardadas as palavras diferentes que vão sendo encontradas.
- Para cada palavra, verificar se já existe na tabela; se não existir, inseri-la e incrementar um contador (conta o número de palavras diferentes).

Tabelas de dispersão: aplicação

// uso de tabela de dispersão implementada através da classe unordered_set (STL)

função de igualdade

```
struct eqstr {
   bool operator() (const string &s1, const string &s2) const {
      return s1==s2;
   }
};
```

função de hash

```
struct hstr {
   int operator() (const string &s1) const  {
      int v = 0;
      for ( unsigned int i=0; i< s1.size(); i++ )
         v = 37*v + s1[i];
      return v;
   }
};
```

Tabelas de dispersão: aplicação

```
#include <unordered_set>

int main() {
   try {
      ifstream fich("texto.txt");
      int diff = 0;
      unordered_set<string, hstr, eqstr> tab1;
      while (!fich.eof()) {
         string palavra1; fich >> palavra1;
         pair<unordered_set<string,hstr,eqstr>::iterator, bool>
                              res = tab1.insert(palavra1);
         if ( res.second == true )    //inseriu, não existia
            diff ++;
      }

      cout << "numero de palavras diferentes : " << diff << endl;
     // continua ...
```

Tabelas de dispersão: aplicação

```
  // continuação

      cout << "palavras diferentes:" << endl;
      unordered_set<string,hstr,eqstr>::iterator it =
                                                  tab1.begin();
      while (it!=tab1.end() {
         cout << *it << endl;
         it++;
      }
    }

    catch (FicheiroNaoExiste e) {
       cout << "ficheiro nao existe"; return -1; }
}
```

Tabelas de dispersão: aplicação 2

- Contagem de ocorrência de palavras

Pretende-se escrever um programa que leia um ficheiro de texto e apresente uma listagem das palavras nele existentes e o respectivo número de ocorrências.

- Usar uma tabela de dispersão, onde são guardadas as palavras diferentes que vão sendo encontradas e contadores associados
- Para cada palavra, verificar se já existe na tabela
  - se não existir, inseri-la com contador = 1
  - se existir, incrementar um contador para contagem do número de palavras diferentes (necessário eliminar o elemento da tabela, e depois inseri-lo com contagem atualizada).

Tabelas de dispersão: aplicação 2

```
class Texto
{
    ifstream f;
public:
    Texto(string nomefich);
    string getPalavra();
    bool fimTexto();
    ~Texto() { f.close(); }
};
```

```
bool Texto::fimTexto() {
    return f.eof();
}
```

```
Texto::Texto(string nomefich) {
    f.open(nomefich.c_str());
    if (!f) throw FicheiroNaoExiste();
}
```

```
string Texto::getPalavra(){
    string pal="";
    if (!f.eof()) f>>pal;
    return pal;
}
```

Tabelas de dispersão: aplicação 2

```
class PalavraFreq
{
   string palavra;
   int frequencia;
public:
   PalavraFreq() : palavra(""), frequencia(0) {};
   PalavraFreq(string p) : palavra(p), frequencia(1) {};
   string getPalavra() const { return palavra; }
   friend ostream & operator << (ostream &out, const PalavraFreq &p);
   void incFrequencia() { frequencia ++; }
};
```

```
ostream & operator << (ostream & out, const PalavraFreq & p) {
   out << p.palavra << " : " << p.frequencia << endl;
   return out;
}
```

Tabelas de dispersão: aplicação 2

função de igualdade

```
struct eqPalF {
   bool operator() (const PalavraFreq &pf1,
                    const PalavraFreq &pf2) const {
      return pf1.getPalavra()==pf2.getPalavra();
   }
};
```

função de hash

```
struct hPalF {
   int operator() (const PalavraFreq &pf1) const  {
      string s1=pf1.getPalavra();
      int v = 0;
      for ( unsigned int i=0; i< s1.size(); i++ )
            v = 37*v + s1[i];
      return v;
   }
};
```

Tabelas de dispersão: aplicação 2

```
typedef
unordered_set<PalavraFreq,hPalF,eqPalF>::iterator iteratorH;
typedef unordered_set<PalavraFreq,hPalF,eqPalF> tabH;

int main()  {
  try {
     Texto tx("texto1.txt");
     tabH tab1;
     while (!tx.fimTexto()) {
       PalavraFreq palavraf1 = PalavraFreq(tx.getPalavra());
       pair<iteratorH, bool> res = tab1.insert(palavraf1);
       if ( res.second ==false) { //não inseriu, já existia
         iteratorH it= res.first;
         PalavraFreq palf=*it;
         tab1.erase(it);
         palf.incFrequencia();
         tab1.insert(palf);
       }
    }
   // continua
```

Tabelas de dispersão: aplicação 2

```
// continuação ...
    cout << "palavras encontradas:" << tab1.size() << endl;

    iteratorH it = tab1.begin();
    while (it!=tab1.end()){
       cout << *it;
       it++;
    }

  }
  catch (FicheiroNaoExiste e) {
    cout << "ficheiro nao existe"; return -1; }

}
```