Tabelas de dispersão

Algoritmos e Estruturas de Dados

2005/2006

Tabela de dispersão

- Uma tabela de dispersão é um vector de tamanho fixo em que os elementos são colocados em na posição determinada por uma função denominada **função de dispersão**.
- A função de dispersão deve:
  - ser fácil de calcular
  - distribuir os objectos uniformemente pela tabela
- Vários objectos podem ser mapeados em uma mesma posição : **colisão**
- O comportamento das tabelas de dispersão é caracterizado 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

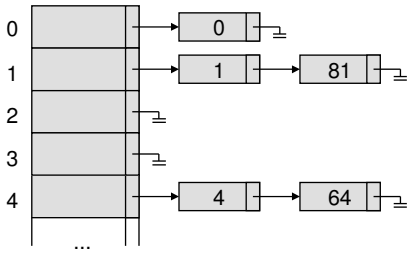
```
int hash (const string & key, int tableSize) {
    int hashVal = 0;
    for ( int i = 0; i < key ; i++ )
        hashVal = 37*hashVal + key[i];
    hashVal %= tableSize;
    if (hashVal < 0 ) hashVal += tableSize;
    return hashVal;
}

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

A qualidade da função de dispersão depende 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

Tabelas de dispersão

- Resolução de colisões por listas

O desempenho pode ser medido pelo número de sondagens efectuadas. Este depende do factor de carga λ

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

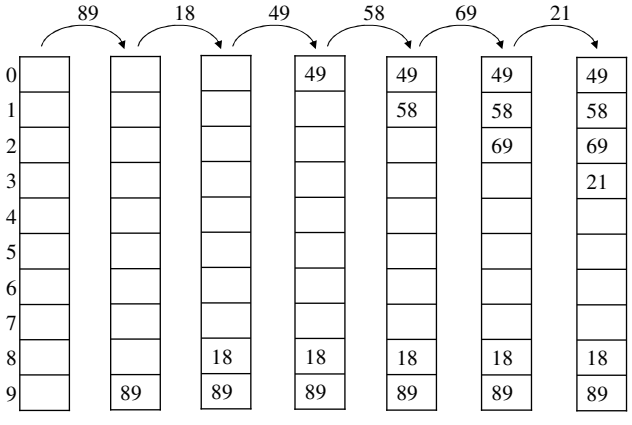
- Comprimento médio de cada lista é λ
- Tempo médio de pesquisa (número de sondagens)
  - Pesquisa sem sucesso: λ
  - Pesquisa com sucesso : 1 + λ/2

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



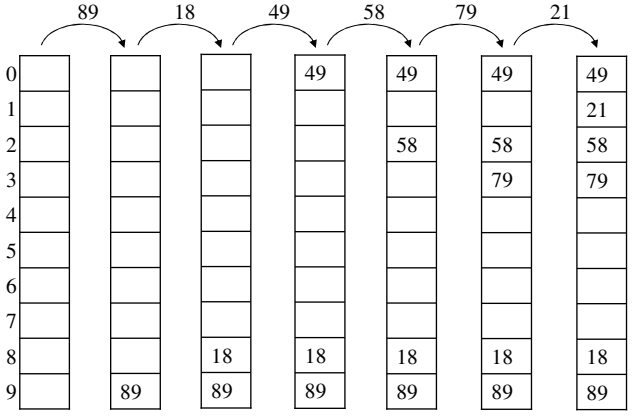
hash(x) = x % 10 ; H(x) = (hash(x) + i) % 10

Tabelas de dispersão

- Sondagem linear
  - Utiliza toda a tabela : $0 \leq \lambda \leq 1$
  - Susceptivel ao fenómeno de *agregação primária*
  - Número médio de sondagens
    - inserção / pesquisa sem sucesso: $\frac{1}{2}\left(1 + 1 / (1-\lambda)^2\right)$
    - pesquisa com sucesso : $\frac{1}{2}\left(1 + 1 / (1-\lambda)\right)$
  - 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) % 10

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 : $H_i = (H_{i-1} + 2i - 1) \bmod TableSize$

Tabelas de dispersão: implementação encadeada

- Declaração da classe ***HashTable***

```
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);

private:
    vector<LList<T> > theLists;
    const T ITEM_NOT_FOUND;
};
```

Tabelas de dispersão: implementação encadeada

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

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

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

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

Tabelas de dispersão: implementação encadeada

- classe ***HashTable*** : inserção, remoção, pesquisa

```
template <class T> void HashTable<T>:: insert(const T & x)
{
    LList<T> & lista1 = theLists[ hash(x, theLists.size()) ];
    LListItr<T> itr = lista1.find(x);
    if (itr.isPastEnd()) lista1.insert(x, lista1.beforeStart());
}

template <class T> void HashTable<T>:: remove(const T & x)
{  theLists[ hash(x, theLists.size()) ].remove(x);  }

template <class T> const T & HashTable<T>:: find(const T & x) const
{
    LListItr<T> itr;
    itr = theLists[ hash(x, theLists.size()) ].find(x);
    if (itr.isPastEnd()) return ITEM_NOT_FOUND;
    else return itr.retrieve();
}
```

Tabelas de dispersão: endereçamento aberto

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

```
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();
}

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;
}

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

Tabelas de Dispersão (Standard Template Library - STL)

- class ***hash_set***

  hash_set<T, HashFunc, EqualFunc>

- Alguns métodos:
  - void insert(const T & x)
  - iterator erase(iterator it)
  - iterator find(const T & x) const
  - iterator begin()
  - iterator end()
  - bool empty() const
  - void clear()

(http://www.sgi.com/tech/stl/hash_set.html)

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.

  - 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

```
int diff = 0;
string NOT_FOUND("");
HashTable<string> tab(NOT_FOUND);

palavra1 = getPalavra();
while (palavra1!="") {
        string s = tab.find(palavra1);
        if ( s == NOT_FOUND) {
                diff ++;
                tab.insert(palavra1);
        }
        palavra1 = getPalavra();
 }
 cout << "número de palavras diferentes : " << diff << endl;
```

Tabelas de dispersão: aplicação

```
// uso de tabela de dispersão implementada através da classe hash_set (STL)
struct eqstr {
      bool operator() (string s1, const string s2) const   { return s1==s2; }
};
struct hstr {
      int operator() (string s1) const  {
            int v = 0;
            for ( int i=0; i< s1.size(); i++ )  v = 37*v + s1[i];
            return v;
     }
};

 typedef hash_set<string, hstr, eqstr> hashStr;
int diff = 0;
hashStr tab;
palavra1 = getPalavra();
while (palavra1!="") {
        hashStr::iterator it = tab.find(palavra1);
        if ( it == tab.end() {
                diff ++;
                tab.insert(palavra1);
        }
        palavra1 = getPalavra();
 }
 cout << "número de palavras diferentes : " << diff << endl;
```