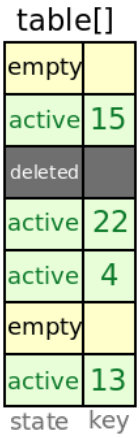
8ªaula prática - Tabelas de Dispersão

Instruções

- Faça download do ficheiro ***aed2122_p08.zip*** da página da disciplina e descomprima-o (contém a pasta ***lib,*** a pasta ***Tests*** com os ficheiros ***funWithHashTables.cpp, hashTable.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- Se desejar pode criar métodos e/ou classes auxiliares para resolver estes exercícios

1. Uma implementação simplificada de Tabelas de Dispersão (*hash tables*).



As tabelas de dispersão são uma estrutura de dados muito útil e eficiente para implementar conjuntos ou dicionários de elementos, permitindo tempo médio constante - $\mathcal{O}(1)$ - para operações como procurar, inserir ou remover elementos. Neste exercício tens acesso a uma implementação base (parcial) para representar conjuntos que usa endereçamento aberto (*open addressing*) com procura linear (*linear probing*).

Em baixo podes ver a declaração da classe `HashTable` (disponível em ***hashtable.h***) e na imagem à esquerda podes ver um possível preenchimento representando um conjunto com 4 elementos ({4,13,15,22}) em posições ativas, 2 posições vazias (nunca foram preenchidas) e 1 posição apagada (onde já chegou a estar um elemento).

```
template <class KeyType>
class HashTable {
  enum EntryType {EMPTY, ACTIVE, DELETED}; // Type of table entry
  struct HashEntry { // One table entry
    EntryType state;
    KeyType key;
  };

  int numActive; // Number of active table entries (with keys)
  int numEmpty;  // Number of empty table entries
  vector<HashEntry> table; // The hash table itself
  function<unsigned(const KeyType&)> hash; // Hash function: key -> unsigned

  int findPos(const KeyType& k); // Linear probing to find suitable position

 public:
  HashTable(int n, function<unsigned(const KeyType&)> h);
  bool contains(const KeyType& k); // Does it contain key k?
  void clear();                    // Clear hash table
  int getNumActive();              // Number of active table entries
  int getNumEmpty();               // Number of empty table entries

  // ----- Functions to implement in this exercise -----
  bool insert(const KeyType& k);   // Insert key k (true if sucessfull)
  bool remove(const KeyType& k);   // Remove key k (true if sucessfull)
  bool rehash(int newSize);        // Resize to n and rehash (true if suc.)
};
```

A tua primeira tarefa é espreitar a implementação (ficheiros ***hashtable.h*** e ***hashtable.cpp)*** e garantir que percebes na sua essência o que está a fazer cada pedaço de código. Nas alíneas seguintes (que deverás fazer por ordem) irás completar a implementação para chegar a um código funcional de uma tabela de dispersão simplificada.

a) Inserindo elementos. Implementa a seguinte função no ficheiro ***hashTable.h***:

```
template <class KeyType>
bool HashTable<KeyType>::insert(const KeyType& k)
```

Complexidade temporal esperada: $\mathcal{O}(1)$ em média, $\mathcal{O}(n)$ no pior caso
(onde *n* é o número de chaves na tabela de dispersão)

Deve inserir a chave *k* na tabela. Use o método `findPos` para procurar a posição onde inserir (que já usa a função de hash e faz pesquisa linear). A função deve devolver *true* se conseguir inserir ou *false* caso contrário (nomeadamente se a chave já existia ou se ao inserir a tabela fica sem posições vazias: queremos que a tabela tenha sempre pelo menos uma posição *empty* em qualquer altura).

b) Removendo elementos. Implementa a seguinte função no ficheiro ***hashTable.h***:

```
template <class KeyType>
bool HashTable<KeyType>::remove(const KeyType& k)
```

Complexidade temporal esperada: $\mathcal{O}(1)$ em média, $\mathcal{O}(n)$ no pior caso
(onde *n* é o número de chaves na tabela de dispersão)

Deve remover a chave *k* da tabela. Use o método `findPos` para procurar a posição onde ela pode estar (que já usa a função de hash e faz pesquisa linear). A função deve devolver *true* se conseguir remover (e nesse caso deve colocar a posição como *deleted* – espreita os slides para perceber porque não pode simplesmente colocar como *empty*) ou *false* caso contrário (nomeadamente se a chave não existia).

c) Refazendo a tabela. Implementa a seguinte função no ficheiro ***hashTable.h***:

```
template <class KeyType>
bool HashTable<KeyType>::rehash(int newSize)
```

Complexidade temporal esperada: $\mathcal{O}(n)$ em média, $\mathcal{O}(n^2)$ no pior caso
(onde *n* é o número de chaves na tabela de dispersão)

Deve refazer a tabela. Para isso deve redimensionar a tabela para o tamanho *n* e voltar a inserir todos os chaves na tabela (note que ao mudar o tamanho, as novas posições têm de voltar ser calculadas). A função deve devolver *true* se conseguir fazer o *rehash*, ou *false* caso contrário (nomeadamente se a nova tabela não tiver espaço para conter todas as chaves e pelo menos uma posição livre).

[vamos agora agora ver uma pequena aplicação da estrutura de dados que foi implementada]

d) Contando diferentes somas de pares. Implementa a seguinte função no ficheiro ***funWithHashTables.cpp***

```
int FunWithHashTables::sumPairs(const vector<int>& numbers)
```

Complexidade temporal esperada: $\mathcal{O}(n^2)$ [em média]
(onde *n* é o tamanho do vector *numbers*)

Dada um vector *numbers* com números inteiros distintos, a função deve devolver a quantidade de inteiros diferentes que se podem obter como uma soma de um par de números de *numbers*.

Exemplo de chamada e output esperado:

```
cout << FunWithHashTables::sumPairs({2,3,6,7,8}) << endl;
```

```
8
```

Explicação: Pode obter-se 8 diferentes números, nomeadamente: **5** (2+3), **8** (2+6), **9** (2+7 ou 3+6), **10** (2+8 ou 3+7), **11** (3+8), **13** (6+7), **14** (6+8) e **15** (7+8)

Sugestão: Gerar todos os pares e guardar cada uma das suas somas numa `HashTable<int>`. No final basta ver o quantas chaves ficaram guardadas na tabela (como função de hash usar a função `hashInt` já disponibilizada).

Os três exercícios seguintes têm como objetivo dar-lhe a oportunidade de perceber e usar as classes de C++ que implementam tabelas de dispersão (`unordered_set`, `unordered_map`, `unordered_multiset` e `unordered_multimap`).

2. Um padrão no DNA. Função a implementar:

```
int FunWithHashTables::dnaMotifs(string dna, int k, unordered_set<string>& motifs)
```

Complexidade temporal esperada: $\mathcal{O}(k \times n)$ [em média]
(onde *n* é o tamanho da string dna)

Uma sequência de DNA pode ser modelada como uma string formada apenas pelas letras A, T, C e G. Um "motivo" é um padrão recorrente que se pensa poder ter alguma função biológica, onde por padrão queremos dizer uma *substring* (subsequência de carateres consecutivos). Neste exercício queremos perceber quais os motivos de comprimento *k* mais frequentes (onde por frequência queremos dizer o número de ocorrências). A função `dnaMotifs` recebe uma sequência dada na string *dna* e um inteiro *k*, e deve devolver a maior frequência de uma substring de tamanho *k,* colocando também em *motifs* as substrings desse tamanho que têm essa frequência máxima (pode existir mais do que uma). As várias ocorrências de um motivo podem ter sobreposição.

Exemplo de chamada e output esperado:

```
unordered_set<string> motifs;
cout << FunWithHashTables::dnaMotifs("GTATAGCGTAATAGTAG", 3, motifs) << endl;
for (auto s : motifs) cout << s << endl;
```

```
3
GTA
TAG
```

Explicação: Na string `"GTATAGCGTAATAGTAG"` os padrões de tamanho 3 mais frequentes são GTA e TAG, aparecendo 3 vezes cada um (`"`**`GTA`**`TAGC`**`GTA`**`ATA`**`GTA`**`G"` e `"GTA`**`TAG`**`CGTAA`**`TAGTAG`**`"`)

Sugestão: Use a função `substr` para ir extraindo todas as substrings de tamanho *k*, e mantenha as frequências num `unordered_map<string, int>` que associa a cada padrão a sua frequência. No final basta ver qual a maior frequência e colocar em *motifs* os padrões que têm essa frequência.

3. Torre de Babel. Função a implementar:

```
void FunWithHashTables::findLanguage(string text,
                                     const unordered_map<string, vector<string>>& dict,
                                     unordered_map<string, int>& answer)
```

Complexidade temporal esperada: $\mathcal{O}(n+m)$ [em média]
(onde *n* é o comprimento de text e *m* o número total de palavras em *dict*)

Dado um pedaço de texto, será que consegues descobrir em que língua foi escrito? Uma metodologia possível é ver quantas palavras do texto pertencem a cada linguagem conhecida. Para te ajudar, além de uma string *text* descrevendo o texto a analisar neste problema recebes também um dicionário *dict* do tipo `unordered_map<string, vector<string>>` que associa a cada língua um vetor de palavras escritas nessa língua (por exemplo, podiamos associar a `"pt"`, ou seja português, as palavras `"algoritmos"`, `"estruturas"` e `"dados"`). A tua tarefa neste exercício é devolver em *answer* a quantidade de palavras de cada língua. (se uma palavra aparecer 2x no texto, conta também 2x na resposta)

As palavras de *dict* vêm garantidamente em minúsculas e são constituídas apenas por letras , mas em *text* podem vir outros caracteres que não letras e as palavras do texto podem conter letras maiúsculas. É garantido que não existem caracteres acentuados e podes considerar que uma palavra é uma sequência contígua de letras separadas por caracteres que não são letras.

Exemplo de chamada e output esperado:

```
unordered_map<string,vector<string>> dict;
dict["pt"] = {"algoritmos", "estruturas", "dados", "tabela","dispersao"};
dict["es"] = {"algoritmos", "estructuras", "datos", "table", "dispersion"};
dict["en"] = {"algorithms", "data", "structures", "hash", "table"};

unordered_map<string, int> answer;
FunWithHashTables::findLanguage("Algoritmos e Estruturas de Dados: uma UC de LEIC",
                                dict, answer);
for (auto i : answer) cout << i.first << ": " << i.second << endl;
```

```
pt: 3
es: 1
en: 0
```

Explicação: Na texto `"Algoritmos e Estruturas de Dados: uma UC de LEIC"` aparecem 3 palavras na língua `"pt"` (`"algoritmos"`, `"estruturas"` e `"dados"`), 1 palavra na língua `"es"` (`"algoritmos"`) e 0 palavras na língua `"en"`.

Sugestão: Parta o problema em 3 partes:

- No que toca ao texto faça uma função separada para fazer o *split*: receber uma string e devolver um vetor de strings com as palavras que surgem no texto. Pode por exemplo fazer um ciclo por todas os caracteres do texto e cada vez que recebe uma letra vai concatenado a sua versão minúscula (operador + nas strings) com a string atual. Não se esqueça de verificar se este se método está a funcionar corretamente! Para verificar se um caracter é uma letra pode por exemplo fazer `#include <cctype>` e usar depois a função `isalpha`. Para converter uma letra para a sua versão minúscula pode usar a função `tolower` da mesma biblioteca.

- Para processar o dicionário dado em dict, crie um outro dicionário onde a chave é uma palavra e o conteúdo são as línguas onde aparece (para o exemplo do enunciado, a palavra `"algoritmos"` estaria associada a `"pt"` e a `"es"`. Para isso crie um `unordered_multimap<string, string>`.

- Finalmente, basta percorrer todas as palavras descobertas com o *split* e para cada uma delas incrementar a frequência das línguas a ela associadas em *answer*, procurando no mapa feito no ponto anterior. Para descobrir todos os valores associados a um dada chave pode usar o método `equal_range`. Não se esqueça de garantir que todas as línguas aparecem em *answer*, mesmo que a quantidade de palavras respetiva seja zero.

4. *(exercício extra)* **Palavras Alienígenas.** Função a implementar:

```
void FunWithHashTables::wordIndex(const vector<string> &words,
                                  vector<int>& answer)
```

Complexidade temporal esperada: $\mathcal{O}(n)$ [em média]
(onde *n* é a quantidade de palavras em *words*)

Num planeta distante todas as palavras possíveis são constituídas por um máximo de cinco letras minúsculas do alfabeto inglês: {a,b,c,…,z}. Para uma palavra ser considerada válida, as suas letras têm de vir em ordem alfabética estritamente crescente, ou seja, uma letra só pode aparecer depois de outra letra se aparecer também depois no alfabeto.

Por exemplo, as seguintes três palavras são válidas: `abc aep gwz`

Já as três palavras seguintes não o são: `aab are cat`

A cada palavra válida é atribuído um inteiro (o índice) correspondendo à sua posição numa lista de todas palavras válidas ordenada alfabeticamente:

```
a -> 1
b -> 2
...
z -> 26
ab -> 27
ac -> 28
...
az -> 51
bc -> 52
...
vwxyz -> 83681
```

A tua tarefa calcular a posição (os indíces) de cada palavra do vetor *words* dado como input. As posições devem ser colocadas no vector *answer*, na mesma ordem em que as palavras aparecem no input.

Exemplo de chamada e output esperado:

```
vector<int> answer;
FunWithHashTables::wordIndex({"ade","a", "ac", "vwxyz"}, answer);
for (auto i : answer) cout << i << endl;
```

```
399
1
28
83681
```

Explicação: A palavra "`ade`" está na posição 399, a palavra "`a`" está na posição 1, a palavra "`ac`" está na posição 28 e palavra "`vwxyz`" está na posição 83681.

Sugestão: Começa por gerar (por ordem) todas as palavras da linguagem e coloque-as num dicionário (`unordered_map<string,int>`) onde a cada palavra associa logo a sua posição. Resolver o problema depois é simplesmente percorrer as palavras dadas em words, ir ver ao dicionário qual a sua posição e ir colocando as respostas em answer. A maior dificuldade é gerar as palavras, mas como este é um exercício extra temos de deixar algo para pensares por ti, certo? (se quiseres ajuda ou dicas fora da aula, podes e deves usar o Slack de AED).

(exercícios extra) **Mais ideias para exploração:**

Se te sentires com vontade de continuar a explorar este tema, ficam aqui duas sugestões possíveis:

- Usar classes customizadas com os containers *unordered* da STL:

No contexto desta aula apenas usamos tipos padrão (ex: *int* e *string*) que já têm implementadas funções de hash em C++. E se o tipo que queremos colocar num *unordered_set*, por exemplo, for uma classe feita por nós? Como podemos fazer a nossa própria função de hash e igualdade e depois passar isso à STL?

Explore esta possibilidade tentando criar um *unordered_set* ou *unordered_map* onde a chave é do tipo de uma classe feita por ti.

Sugestões de leitura:
- std::hash no cpp reference (ver exemplo de código)
- How to create an unordered_map of user defined class in C++? (GeeksforGeeks)
- Combining Hash Values (Boost C++ Libraries)

- Completar a classe simplificada de tabelas de dispersão dada na aula:

Ainda falta muitas coisas para que a classe que completou no primeiro exercício seja "usável". Aqui ficam algumas funcionalidades que podes adicionar:

- Ao invés de não inserir quando o espaço para novas chaves terminar, aumentar o tamanho da tabela e fazer rehash. *(para que tamanho fará sentido aumentar? e se a tabela ficar demasiado vazia depois de remover?)*

- Reaproveitar os espaços *deleted* ao inserir novos elementos

- Implementar outros tipos de pesquisa como os dados na teórica (*quadratic probing* e *double hashing*)