7ªaula prática - Árvores binárias de pesquisa equilibradas

Instruções

- Faça download do ficheiro ***aed2122_p07.zip*** da página da disciplina e descomprima-o (contém a pasta ***lib,*** a pasta ***Tests*** com os ficheiros ***funWithBSTs.cpp, funWithBSTs.h, bst.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- Se desejar pode criar métodos e/ou classes auxiliares para resolver estes exercícios

Os exercícios base desta aula envolvem a criação de métodos estáticos da classe **`FunWithBSTs`** e têm como objetivo dar-lhe a oprtunidade de perceber e usar as classes de C++ que implementam árvores binárias de pesquisa equilibradas (`set`, `map`, `multiset` e `multimap`).

1. Coleção de Cromos. Método a implementar:

```
int FunWithBSTs::newBag(const vector<int>& collection, const vector<int>& bag)
```

Complexidade temporal esperada: $\mathcal{O}(n \log n)$
(onde *n* é a soma dos tamanho dos vectores `collection` e `bag`)

A Alice está a começar uma nova coleção de cromos com as figuras mais importantes no mundo dos Algoritmos e das Estruturas de Dados. Cada cromo é identificado por um número inteiro positivo.

A Alice regista religiosamente todos os cromos que tem na sua coleção (incluindo os repetidos) e cada vez que compra uma nova saqueta quer saber quantos cromos novos passou a ter. Será que podes ajudá-la a fazer esta tarefa de uma forma eficiente?

A lista de cromos da coleção da Alice é dada pelo vector de inteiros `collection` (indicando os seus identificadores) e a lista de cromos na nova saqueta é dada pelo vector `bag`. A tua tarefa é devolver o número de diferentes cromos novos.

Exemplo de chamada e output esperado:

```
vector<int> collection = {42, 124, 34, 12, 42, 24, 71};
vector<int> bag = {112, 42, 31, 31, 34, 62};
cout << FunWithBSTs::newBag(collection, bag) << endl;
```

```
3
```

Explicação: 3 novos (diferentes) cromos: o 112, o 31 e o 62. Os cromos 42 e 34 são repetidos.

Sugestão: Use um `set` para representar a coleção. Ao inserir elementos o que acontece se forem repetidos?

2. Batalha de Pokemons. Método a implementar:

```
int FunWithBSTs::battle(const vector<int>& alice, const vector<int>& bruno)
```

Complexidade temporal esperada: $\mathcal{O}(n \log n)$
(onde *n* é a soma dos tamanho dos vectores `alice` e `bruno`)

A Alice decidiu agora colecionar cartas de Pokemon e está ansiosa por batalhar contra o Bruno. Cada carta de Pokemon é identificada por um inteiro positivo indicando a sua "energia" e inicialmente a Alice e o Bruno dispõe de um baralho. A batalha decorre por rondas da seguinte maneira:

- No início da ronda a Alice escolhe a carta com maior energia que ainda resta no seu baralho (seja essa carta chamada de *A*). O Bruno faz o mesmo e escolhe a carta *B* com mais energia do seu baralho.
- Existe uma luta entre as cartas *A* e *B* e ganha a carta com maior energia:
  . Se a *energia(A)* > *energia(B)*, *B* "morre" e sai da batalha e *A* é devolvida ao baralho da Alice e a sua energia diminui, passando a ser *energia(A)* - *energia(B)*
  . Se *energia(A)* < *energia(B)*, *A* "morre", acontecendo algo similar (mas simétrico) ao caso anterior
  . Se *energia(A)* = *energia(B)* ambas as cartas *A* e *B* morrem e saem da batalha
- O jogo termina quando uma das crianças fica sem cartas no seu baralho
- O vencedor é aquele que fica ainda com cartas, sendo que pode acontecer um empate se a última batalha resultar num caso onde ambos ficam sem cartas no baralho.

A Alice não quer nada perder com o Bruno e precisa da tua ajuda para simular o jogo e perceber quem irá ganhar e com quantas cartas ficará.

O baralho da Alice é dado pelo vector de inteiros `alice` (indicando as suas energias) e o vector `bruno` com o baralho do Bruno. A tua tarefa é simular a batalha e devolver o resultado da seguinte forma:

- devolver um inteiro positivo *a* se a Alice ganhar e ficar no final com *a* cartas
- devolver um inteiro negativo *-b* se o Bruno ganhar e ficar no final com *b* cartas
- devolver zero se houver um empate (ambos ficam sem cartas)

Exemplo de chamada e output esperado:

```
vector<int> a = {400,300,700};
vector<int> b = {700,200,500,200};
cout << FunWithBSTs::battle(a, b) << endl;
```

```
-2
```

Explicação: Ganha o Bruno e sobram no final duas cartas no seu baralho:
. 1ª ronda: Lutam duas cartas de 700 e ambas "morrem" ficando Alice={400,300} e Bruno={200,500,200}
. 2ª ronda: 400 da Alice luta com 500 do Bruno; ganha o Bruno e fica Alice={300} e Bruno={200,100,200}
. 3ª ronda: 300 da Alice luta com um 200 do Bruno; ganha a Alice e fica Alice={100} e Bruno={100,200}
. 4ª ronda: 100 da Alice luta com 200 do Bruno; ganha o Bruno e termina Alice={} e Bruno={100,100}

Sugestão: Use um `multiset` para representar um baralho. Como ir buscar o máximo para ir simulando a batalha?

3. Reviews Cinematográficas.

Quem gosta de cinema conhece sites como o *IMDB* ou o *Rotten Tomatoes*, que contêm muitas informações sobre filmes, com particular destaque para as reviews de filmes feitas por utilizadores ou críticos cinematográficos.

Neste problema em particular iremos querer poder processar reviews. Cada review é dada como um par *(nomeFilme, pontuacao)*, onde a pontuação é um inteiro entre 1 e 10, e em todas as alíneas receberá as reviews num vector de pares de strings e ints. O seguinte pedaço de código ilustra um possível vector contendo 6 avaliações (este mesmo vector será usado depois em todos os exemplos de chamadas a funções deste exercício).

```
vector<pair<string, int>> reviews = {
    {"FreeGuy", 8},
    {"Eternals", 6},
    {"Dune", 10},
    {"Eternals", 5},
    {"NoTimeToDie", 7},
    {"Dune", 9}
};
```

a) Contando Filmes. Método a implementar:

```
int FunWithBSTs::numberMovies(const vector<pair<string, int>>& reviews)
```

Complexidade temporal esperada: $\mathcal{O}(n \log n)$
(onde *n* é o tamanho do vector `reviews`)

Para começar e para "aquecer os motores" e garantir que percebe como funcionam os pares, tem apenas de contar quantos filmes existem. A tua tarefa é por isso devolver um inteiro contendo o número de nomes diferentes de filmes.

Exemplo de chamada e output esperado (usando o vector `reviews` dado anteriormente):

```
cout << FunWithBSTs::numberMovies(reviews) << endl;
```

```
4
```

Explicação: Existem 4 filmes: *Eternals*, *Dune*, *FreeGuy* e *NoTimeToDie*.

Sugestão: Ainda precisa de ajuda depois dos exercícios anteriores? Vamos apenas dizer que um `pair` contém dois atributos `first` e `second`.

b) O filme com mais reviews. Método a implementar:

```
void FunWithBSTs::moreReviews(const vector<pair<string, int>>& reviews, int&m, int&n)
```

Complexidade temporal esperada: $\mathcal{O}(n \log n)$
(onde *n* é o tamanho do vector `reviews`)

A tua tarefa é devolver em *m* o número máximo de reviews feitas a um só filme e em *n* o número de filmes que têm *m* reviews.

Exemplo de chamada e output esperado (usando o vector `reviews` dado anteriormente):

```
int m, n;
FunWithBSTs::moreReviews(reviews, m, n);
cout << m << " " << n << endl;
```

```
2 2
```

Explicação: *m*=2, o maior número de reviews de um filme, e existem *n=2* filmes com 2 reviews (*Dune* e *Eternals*).

Sugestão: Use um `map` para associar a cada filme o número de reviews feitas.

c) Os melhores filmes. Método a implementar:

```
vector<string> FunWithBSTs::topMovies(const vector<pair<string, int>>& reviews, double k)
```

Complexidade temporal esperada: $\mathcal{O}(n \log n)$
(onde *n* é o tamanho do vector `reviews`)

A tua tarefa é devolver um vector contendo os nomes de todos os filmes cuja média aritmética das reviews seja maior ou igual a *k*. Os nomes devem vir por ordem lexicográfica (uma string *a* deve vir antes da string *b* se $a < b$).

Exemplo de chamada e output esperado (usando o vector `reviews` dado anteriormente):

```
vector<string> ans = FunWithBSTs::topMovies(reviews, 7.5);
for (auto s : ans) cout << s << endl;
```

```
Dune
FreeGuy
```

Explicação: São devolvidos 2 filmes: *Dune* (com média 9.5 nas reviews) e *FreeGuy* (com média 8.0).

Sugestão: já chega de dicas, não acham?

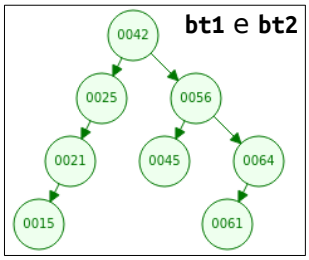
4. *(exercício extra)* **Brincando com árvores.**

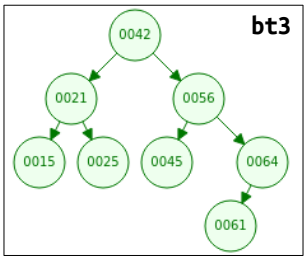
Nas aulas teóricas foram descritas várias árvores binárias de pesquisa equilibradas. Implementar completamente uma das estruturas de dados dadas está fora do escopo de uma única aula prática, mas vamos sugerir alguns exercícios que lhe vão dar alguma compreensão do tipo de operações necessárias. Como são exercícios extras não vamos dar sugestões de resolução no enunciado.

A classe usada é a `BST` dada na aula prática anterior com apenas dois novos métodos: uma versão de `insert` que permite inserir todos os elementos de um vector pela ordem em que aparecem (para facilitar a criação de árvores) e um overload do operador `==` na classe BST que permite comprar se duas árvores são exatamente iguais (têm a mesma estrutura e os mesmos elementos nos mesmos sítios).

O seguinte excerto de código ilustra o uso dos métodos adicionados com a criação de 3 diferentes árvores binárias de pesquisa e sua comparação.

```
BST<int> bt1(-1), bt2(-1), bt3(-1);
bt1.insert({42,25,56,21,45,64,15,61});
bt2.insert({42,25,21,15,56,64,61,45});
bt3.insert({42,21,25,15,56,64,61,45});
cout << (bt1 == bt2) << endl; // escreve 1
cout << (bt1 == bt3) << endl; // escreve 0
```





Pode visualizar BSTs no url dado nas teóricas: https://www.cs.usfca.edu/~galles/visualization/BST.html

a) Desequilíbrio de um nó. Método a implementar:

```
int BST<Comparable>::balance(const Comparable& x) const
```

Complexidade temporal esperada: $\mathcal{O}(n)$
(onde *n* é o tamanho da árvore)

A tua tarefa é devolver um inteiro contendo o desequilíbrio do nó contendo o valor *x*. O desequilíbrio é expresso como sendo a altura do ramo direito menos a altura do ramo esquerdo. Se o valor *x* não existir, o método deve devolver zero.

Exemplo de chamada e output esperado (usando as árvores dadas anteriormente):

```
cout << bt1.balance(42) << endl;
cout << bt1.balance(25) << endl;
cout << bt1.balance(56) << endl;
```

```
0
-2
1
```

Explicação: - nó 42 está equilibrado
- ramo esquerdo do nó 25 mais profundo 2 níveis do que ramo direito
- ramo direito do nó 56 mais profundo 1 nível do que ramo esquerdo

b) É uma árvore AVL? Método a implementar:

```
bool BST<Comparable>::isAVL() const
```

Complexidade temporal esperada: $\mathcal{O}(n^2)$ *[bónus: é possível fazer em $\mathcal{O}(n)$, consegue ver como?]*
(onde *n* é o tamanho da árvore)

A tua tarefa é devolver *true* se a BST for uma árvore AVL ou *false* caso contrário. Recorda que ser árvore AVL todos os seus nós têm de ser equilibrados (em todos eles a diferença entre as alturas dos dois ramos é no máximo 1).

Exemplo de chamada e output esperado (usando as árvores dadas anteriormente):

```
cout << bt1.isAVL() << endl;
cout << bt3.isAVL() << endl;
```

```
0
1
```

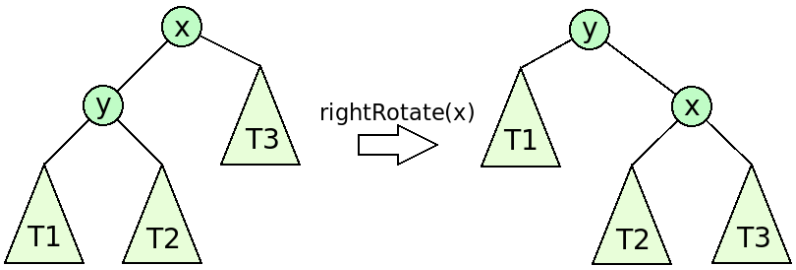
Explicação: - `bt1` não é uma árvore AVL (nó 25 tem desequilíbrio de valor absoluto 2)
- `bt3` é uma árvore AVL

c) Rotação à direita. Método a implementar:

```
void BST<Comparable>::rightRotate(const Comparable& x)
```

Complexidade temporal esperada: $\mathcal{O}(h)$
(onde *h* é o tamanho da árvore)

A tua tarefa é fazer uma rotação à direita no nó com o elemento *x* como indicado na figura. Se não for possível fazer a rotação (o elemento não existe ou não tem filho esquerdo) o método não deve fazer nada.



Exemplo de chamada e output esperado (usando as árvores dadas anteriormente):

```
cout << (bt1 == bt2) << " " << (bt1 == bt3) << endl;
bt1.rightRotate(25);
cout << (bt1 == bt2) << " " << (bt1 == bt3) << endl;
```

```
1 0
0 1
```

Explicação: - inicialmente `bt1` é igual a `bt2` mas diferente de `bt3`
- depois de rodar à direita no 25, `bt1` fica igual a `bt3`

Árvores AVL.
Se estiver a sentir-se com vontade de experimentar mais pode continuar e implementar realmente árvores AVL. Comece pela inserção (já tem "quase" todas as peças básicas e até pode testar se consegue manter o invariante chamando o método `isAVL`). Note que para uma implementação realmente eficiente as operações de inserção, remoção e pesquisa têm de ser sempre em tempo $\mathcal{O}(\log n)$