Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030

◁ 1ª Avaliação ▷

11 de abril de 2017

A avaliação consiste de 5 **funções genéricas** que precisam ser desenvolvidas e testadas. É fornecido junto com a prova um programa `prova.cpp` que *testa apenas as 2 primeiras funções solicitadas*. Portanto, você deve desenvolver testes para as últimas 3 funções.

As funções que você precisa desenvolver estão codificadas com *stubs*[1] As partes que precisam ser alteradas estão marcadas com `// TODO`.

2 pts

1. Desenvolva uma função `eh_particionada` que retorna `true` se todos os elementos do intervalo [`first`, `last`) que satisfazem o predicado `p` aparecem **antes** dos elementos que não satisfazem o predicado `p`. A função também retorna `true` se o intervalo [`first`, `last`) for vazio. O protótipo da função pode ser:

```
using UnaryPredicate = bool ( const void * );
bool eh_particionada( const void *first, const void *last, size_t size,
                      UnaryPredicate *p );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para examinar.
- `size` - tamanho de um elemento do intervalo em bytes.
- `p` - predicado unário que retorna `true` para o elemento desejado. A assinatura da função predicado deve ser equivalente a

```
bool pred( const void * a );
```

4 pts

2. A função `limite_inferior` recebe como parâmetros o intervalo [`first`, `last`) cujos elementos estão em ordem não-decrescente, um valor `value` e retorna um ponteiro `void` para o primeiro elemento do intervalo que **não é menor** do que `value`, ou retorna um ponteiro para `last` se tal elemento não for encontrado.

Por exemplo, se chamarmos `limite_inferior` sobre o intervalo `[begin(A), end(A))`, com `value` $=4$, considerando o vetor abaixo, a função retornaria um ponteiro para posição $A[7]$ que contém o elemento 4, que é o primeiro elemento do intervalo que **não é menor** do que 4.

A:	1	1	2	3	3	3	3	4	4	4	5	5	6
	0	1	2	3	4	5	6	7	8	9	10	11	12

Analogamente, a função `limite_superior` recebe os mesmos parâmetros que `limite_inferior` mas retorna um ponteiro para o primeiro elemento do intervalo fechado-aberto [`first`; `last`) **maior** do que `value`, ou um ponteiro para `last` se tal elemento não for encontrado.

[1] Um código temporário que não resolve o problema mas está lá apenas para permitir que o programa de testes seja compilado e executado.

Se aplicada ao exemplo anterior com os mesmo parâmetros, a função `limite_superior` retornaria um apontador para $A[10]$ que contém o elemento 5, que é o primeiro elemento do intervalo que é **maior** que 4.

Implemente as duas funções **com complexidade de pior caso** $O(\log n)$. Os protótipos das funções podem ser:

```
using Compare = int( const void *, const void * );
void * limite_inferior( const void *first, const void *last, size_t size,
                        const void *value, Compare *cmp );
void * limite_superior( const void *first, const void *last, size_t size,
                        const void *value, Compare *cmp );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para examinar.
- `size` - tamanho de um elemento do intervalo em bytes.
- `value` - valor usado para definir o limite.
- `cmp` - função de comparação que retorna -1 se $a < b$, 0 se $a = b$ e 1 se $a > b$. A assinatura da função de comparação deve ser equivalente a

  ```
  int cmp( const void * a, const void *b );
  ```

Dica: utilize a estratégia da busca binária iterativa para estruturar seu algoritmo.

2 pts 3. Desenvolva uma função `remove` que remove todos os elementos no intervalo [`r_first`, `r_last`) definido dentro de um intervalo possivelmente maior, [`first`, `last`), e retorna um ponteiro para o elemento imediatamente seguinte ao novo fim lógico do intervalo resultante. A remoção envolve o deslocamento dos elementos no intervalo de tal maneira que os elementos que devem ser preservados aparecem no início do intervalo. A ordem relativa dos elementos que permanecem é preservada e o tamanho físico do vetor não é modificado.

Por exemplo, se chamarmos `remove(begin(A), end(A), begin(A)+2, begin(A)+5)` sobre o vetor abaixo,

A:	1	2	3	4	5	6	7	8	9	10
	0	1	2	3	4	5	6	7	8	9

o resultado seria a remoção dos elementos $A[2]$ até $A[4]$ (inclusive), resultando no vetor abaixo e retornaria um ponteiro para $A[7]$, que é a posição logo após o novo fim lógico do vetor.

A:	1	2	6	7	8	9	10	?	?	?
	0	1	2	3	4	5	6	7	8	9

O protótipo da função `remove` pode ser:

```
void * remove( const void *first, const void *last, const void *first_r,
               const void *last_r, size_t size );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para examinar.
- `first_r`, `last_r` - ponteiros que definem o sub-intervalo de elementos para remover.
- `size` - tamanho de um elemento do intervalo em bytes.

2 pts 4. Desenvolva uma função `contem` que retorna `true` se todos os elementos do intervalo ordenado [`first2`, `last2`) estão presentes no intervalo ordenado [`first1`, `last1`), ou `false` em caso contrário. A verificação de **pertinência de conjunto** deve ser feita com complexidade linear com relação a soma dos comprimentos dos intervalos. O protótipo da função `contem` pode ser:

```
using Compare = int( const void *, const void * );
bool contem( const void *first1, const void *last1, const void *first2,
             const void *last2, size_t size, Compare *cmp );
```

- `first1`, `last1` - ponteiros que definem o intervalo de elementos que representam o conjunto principal a ser examinado.
- `first2`, `last2` - ponteiros que definem o possível sub-conjunto que desejamos verificar se está contido no primeiro conjunto.
- `size` - tamanho de um elemento do intervalo em bytes.
- `cmp` - função de comparação que retorna -1 se $a < b$, 0 se $a = b$ e 1 se $a > b$. A assinatura da função de comparação deve ser equivalente a

  ```
  int cmp( const void * a, const void *b );
  ```

2 pts 5. Desenvolva uma função `remove_repetidos` que remove todos os elementos iguais a `value` no intervalo [`first`, `last`) e retorna um ponteiro após-o-final para o novo fim lógico do intervalo resultante. Desenvolva esta função da forma mais eficiente possível. Assuma que os elementos do intervalo estarão em uma ordem qualquer. O protótipo da função `remove_repetidos` pode ser:

```
using Compare = int( const void *, const void * );
void * remove_repetidos( const void *first, const void *last, const void *value,
                         size_t size, Compare *cmp );
```

- `first`, `last` - ponteiros que definem o intervalo de elementos para examinar.
- `value` - valor a ser removido do intervalo.
- `size` - tamanho de um elemento do intervalo em bytes.
- `cmp` - função de comparação que retorna -1 se $a < b$, 0 se $a = b$ e 1 se $a > b$. A assinatura da função de comparação deve ser equivalente a

  ```
  int cmp( const void * a, const void *b );
  ```

Dica #1: utilize as funções desenvolvidas anteriormente.
Dica #2: se o intervalo estiver ordenado, torna-se mais fácil solucionar o problema.

∽ FIM ∽