Lista de Exercícios 3 - CAP-241 2017

Prof. Dr. Gilberto Ribeiro de Queiroz.

Aluno: Paulo Henrique Barchi[1a]

17 de abril de 2017

[1]paulobarchi@gmail.com

[a] Laboratório Associado de Computação e Matemática Aplicada (LAC)
Coordenação de Laboratórios Associados (CTE)
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos, SP - Brasil.

Exercício 01. A partir da implementação de lista duplamente encadeadas com templates e iteradores da wiki do curso, foram implementadas as seguintes operações:

- `front()`: Retorna o primeiro elemento da lista. É uma operação de custo *O(1)* pois apenas retorna o próximo do sentinela.
- `back()`: Retorna o último elemento da lista. É uma operação de custo *O(1)* pois apenas retorna o anterior ao sentinela.
- `push_front(v)`: Insere o valor v na cabeça da lista. É uma operação de custo *O(1)*. Chama operação `insert(begin(), v)`, passando a posição do começo da lista).
- `push_back(v)`: Insere o valor v no final da lista. É uma operação de custo *O(1)*. Chama operação `insert(end(), v)`, passando a posição do fim da lista).
- `pop_front()`: Remove o primeiro elemento da lista. É uma operação de custo *O(1)*. Copia o valor do primeiro elemento da fila para retorná-lo, e então chama operação `erase(begin())`, passando a posição do começo da lista.
- `pop_back()`: Remove o último elemento da lista. É uma operação de custo *O(1)*. Copia o valor do último elemento da fila para retorná-lo, e, passando a posição do anterior ao sentinela da lista, chama operação `erase(iterator(sentinel_.prev))`. É necessário usar `sentinel_.prev` porque `end()` é a posição do próximo ao último elemento da lista, para inserção no fim da lista.
- `splice(`L_2`)`: Funde os elementos da lista L_2 ao final da lista operada. Considerando o tamanho da lista L_2 como n, é uma operação de custo *O(n)*. Para cada elemento da lista L_2, chama a operação `insert(end(), v)` para inserção do valor atual no fim da lista operada.
- `reverse()`: Reverte os elementos da lista. É uma operação de custo *O(n)*. Para realizar a operação, são utilizados os iteradores `pprev`, `pnext` (iniciados com `end()`) e `current` (iniciado com `begin()`). Enquanto `current` não chegar ao final da lista, os links de próximo e anterior de `current` são trocados, com as atribuições: `pnext.item_` recebe `current.item_->next`, `current.item_->next` recebe `pprev.item_`, `pprev.item_` recebe `current.item_` e `current.item_` recebe `pnext.item_`. Ao chegar ao fim da lista, define o começo da lista com `sentinel_.next = pprev.item_`.
- `merge(`L_2`)`: Junta duas listas ordenadas de forma ascendente em uma só, também ordenada. A Lista L_2 ficará vazia ao final. Esta é a operação mais custosa. Considerando n o tamanho da lista operada e m o tamanho da lista L_2, podemos afirmar que o custo desta operação é $O(n \times m)$. Para tamanhos de listas muito grandes, é uma operação de custo $O(n^2)$. Para realizar a operação, são utilizados os iteradores `current_l1`, iniciado com o começo da lista operada, e, `current_l2` com o começo da lista L_2. Enquanto `current_l2` não chegar ao final da lista L_2, e, enquanto `current_l1` não chegar ao final da lista operada (laços aninhandos), caso o valor do item em `current_l2` seja menor que o valor do item em `current_l1`, chama `insert(current_l1, current_l2.item_->value)` para inserir na lista operada o valor do item de `current_l2` na posição de `current_l1` e quebra o laço mais interno, para seguirmos para o próximo elemento da lista L_2. Os iteradores `current_l1` e `current_l2` são incrementados em seus respectivos laços para percorrerem as listas por completo. Depois de realizados os laços aninhados, a lista L_2 é esvaziada com a operação `clear()`.

O código referente a este programa está nos arquivos `list.hpp` (implementação das operações) e `exercicio01.cpp` (`main()` e testes), compactados juntos com este documento e demais arquivos.

Exercício 02. A implementação destes algoritmos está no arquivo `exercicio01.cpp`. A função `push(v)` do arquivo `stack.hpp` estava estava gerando *Segmentation fault* na realocação da pilha. Por isso, foi feita uma alteração do tamanho da nova alocação de $(2 * size\_)$ para $(5 + size\_)$, e, desta forma, os testes realizados obtiveram sucesso no processamentos e resultados. Estes arquivos (`exercicio01.cpp` e `stack.hpp`) estão compactados juntos com este documento e demais arquivos.