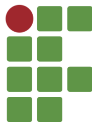
Listas

Introdução à Programação Competitiva



Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga

Sumário

1. Introdução
2. Listas
3. Outras operações
4. Exemplo
5. Referências

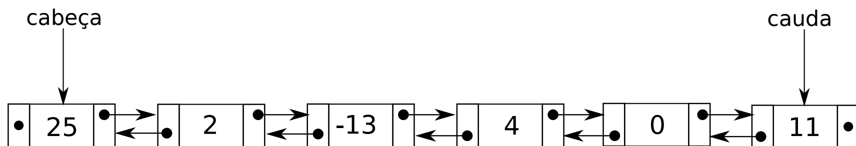
Sumário

1 Introdução

Introdução

- Listas são **tipos abstratos de dados** que representam uma sequência de elementos.
- Idealmente devem suportar inserção e remoção eficiente na primeira e última posições.
- Na `STL` do `C++`, são implementadas através de uma sequência autorreferenciada duplamente encadeada.
- Os nós das listas possuem ponteiros para o elementos anterior e próximo.
- Acesso especial aos nós das extremidades.

Introdução



Introdução

- Por conta dos nós não se encontrarem, necessariamente, em posições consecutivas na memória, perdemos o acesso direto a cada elemento.
- Para chegar no 3° elemento, por exemplo, devemos começar do primeiro e seguir as referências até chegar ao terceiro elemento.
- Contudo, conseguimos **inserir e remover** elementos no **meio** da lista se tivermos a referência para a posição a ser inserida ou removida.

Sumário

2 Listas

Sumário

2 Listas
- Declaração e Inicialização
- Inserção
- Remoção
- Acesso
- Limpeza
- Métodos auxiliares

Declaração

- Para declarar uma lista em C++, basta utilizarmos o tipo `std::list<T> nome_variavel;` em que T corresponde ao tipo desejado.
- Exemplos:
  - `std::list<int> lista;`
  - `std::list<vector<int>> lista_de_vetores;`
  - `std::list<pair<string,double>> lista_pares;`

Inicialização

- Existem vários construtores para lista:

```
1  int main() {
2      // C++11 initializer list syntax:
3      std::list<std::string> words1{"the", "frogurt", "is",
   ↪   "also", "cursed"};
4
5      // words2 == words1
6      std::list<std::string> words2(words1.begin(),
   ↪   words1.end());
7
8      // words3 == words1
9      std::list<std::string> words3(words1);
10
11     // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
12     std::list<std::string> words4(5, "Mo");
13 }
```

Sumário

2 Listas
- Declaração e Inicialização
- Inserção
- Remoção
- Acesso
- Limpeza
- Métodos auxiliares

Inserção: extremidades

- Para inserir nas extremidades da lista, podemos utilizar os métodos `push_front` e `push_back`. O primeiro insere no início da lista (cabeça) e o último no final da lista (cauda).
- Eles possuem custo $\Theta(1)$.
- As variantes `emplace_front` e `emplace_back` fazem o mesmo, mas *in-place*.

Inserção: extremidades

```
1   #include <iostream>
2   #include <list>
3   #include <string>
4
5   using namespace std;
6
7   int main() {
8       list<string> l;
9       l.push_back("a roupa");
10      l.push_front("roeu");
11      l.emplace_front("o rato");
12      l.emplace_back("do rei de roma");
13      for (const auto &s : l) {
14          cout << s << '\n';
15      }
16      return 0;
17  }
```

Inserção: posições arbtirárias

- Também é possível inserir em uma posição qualquer da lista utilizando o método `insert`.
- Ela recebe um iterador para uma determinada posição da lista.
- O(s) elemento(s) serão inseridos antes desta posição.
- O custo total depende do número de elementos inseridos.
- Dependendo da forma como a qual os itens são inseridos, precisa-se posicionar o iterador na posição correta, o que leva, no pior caso, tempo $\Theta(n)$.

Inserção: posições arbitrárias

```
#include <iostream>
#include <iterator>
#include <list>
#include <vector>

using namespace std;

void print(int id, const list<int> &container) {
    std::cout << id << ". ";
    for (const int x : container)
        std::cout << x << ' ';
    std::cout << '\n';
}

```

Inserção: posições arbitrárias

```
15 int main() {
16     // 100 100 100
17     list<int> c1(3, 100);
18     print(1, c1);
19
20     // 200 100 100 100
21     auto it = c1.begin();
22     it = c1.insert(it, 200);
23     print(2, c1);
24
25     // 300 300 200 100 100 100
26     c1.insert(it, 2, 300);
27     print(3, c1);
28
29     // reset `it` to the begin:
30     it = c1.begin();
31
```

Inserção: posições arbitrárias

```
32      // 300 300 400 400 200 100 100 100
33      list<int> c2(2, 400);
34      c1.insert(std::next(it, 2), c2.begin(), c2.end());
35      print(4, c1);
36
37      // 501 502 503 300 300 400 400 200 100 100 100
38      vector<int> arr = {501, 502, 503};
39      c1.insert(c1.begin(), arr.cbegin(), arr.cend());
40      print(5, c1);
41
42      // 501 502 503 300 300 400 400 200 100 100 100 601 602 603
43      c1.insert(c1.end(), {601, 602, 603});
44      print(6, c1);
45      return 0;
46  }
```

Sumário

2 Listas
- Declaração e Inicialização
- Inserção
- Remoção
- Acesso
- Limpeza
- Métodos auxiliares

Remoção: extremidades

- Para remover nas extremidades da lista, podemos utilizar os métodos `pop_front` e `pop_back`. O primeiro remove a cabeça da lista e o segundo a cauda da lista.
- Eles possuem custo $\Theta(1)$.

Remoção: cabeça

```
1   #include <iostream>
2   #include <list>
3
4   using namespace std;
5
6   int main() {
7       list<char> chars{'A', 'B', 'C', 'D'};
8       for (; !chars.empty(); chars.pop_front()) {
9           std::cout << "chars.front(): '" << chars.front() << "'\n";
10      }
11      return 0;
12  }
```

- Será impresso `A`, `B`, `C`, `D`.

Remoção: cauda

```
1   #include <iostream>
2   #include <list>
3
4   using namespace std;
5
6   int main() {
7       list<char> chars{'A', 'B', 'C', 'D'};
8       for (; !chars.empty(); chars.pop_back()) {
9           std::cout << "chars.back(): '" << chars.back() << "'\n";
10      }
11      return 0;
12  }
```

- Será impresso `D`, `C`, `B`, `A`.

Remoção: posições arbitrárias

- Para remover itens em posições arbitrárias, basta utilizar o método `erase`.
- Ele recebe um iterador para o elemento a ser removido e o remove.
- Também é possível receber um intervalo a ser removido através de iteradores de início e fim.
- Custo total é proporcional ao número de elementos a serem removidos.
- Dependendo da forma como a qual os itens são inseridos, precisa-se posicionar o iterador na posição correta, o que leva, no pior caso, tempo $\Theta(n)$.
- **Importante**: o método retorna um iterador para o elemento que estava à direita do removido.

Remoção: posições arbitrárias

```
1  #include <iostream>
2  #include <iterator>
3  #include <list>
4
5  using namespace std;
6
7  void print_list(list<int> &l) {
8      for (int i : l)
9          std::cout << i << " ";
10     std::cout << '\n';
11 }
12
13 int main() {
14     std::list<int> l{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
15     print_list(l);
```

Remoção: posições arbitrárias

```
16
17      // 1 2 3 4 5 6 7 8 9
18      l.erase(l.begin());
19      print_list(l);
20
21      auto range_begin = l.begin();
22      auto range_end = l.begin();
23      // iterador range_begin está sob o terceiro elemento
24      std::advance(range_begin, 2);
25      // iterador range_end está sob o sexto elemento
26      std::advance(range_end, 5);
27
28      /** Apagamos todos os inteiros nas posições [2,5) da lista **/
29      l.erase(range_begin, range_end);
30      // 1 2 6 7 8 9
31      print_list(l);
```

Remoção: posições arbitrárias

```
32
33      // apaga-se todos os pares
34      for (auto it = l.begin(); it != l.end();) {
35          if (*it % 2 == 0)
36              it = l.erase(it);
37          else
38              ++it;
39      }
40      // 1 7 9
41      print_list(l);
42  }
```

Sumário

2 Listas
- Declaração e Inicialização
- Inserção
- Remoção
- Acesso
- Limpeza
- Métodos auxiliares

Acesso: extremidades

- Para acessar o dado da cabeça, basta utilizar o método `front`.
- Para acessar o dado da cauda, usamos o método `back`.
- Ambos levam tempo constante.
- Qualquer outra informação deve ser acessar percorrendo a lista até a posição desejada, o que leva, no pior caso, tempo $\Theta(n)$.

Acesso: extremidades

```
1  #include <iostream>
2  #include <list>
3
4  int main() {
5      std::list<char> letters{'o', 'm', 'g', 'w', 't', 'f'};
6      if (!letters.empty()) {
7          std::cout << "Primeiro caractere: " << letters.front() <<
   ↪  '\n';
8          std::cout << "Último caractere: " << letters.back() <<
   ↪  '\n';
9      }
10     return 0;
11 }
```

Sumário

2 Listas
- Declaração e Inicialização
- Inserção
- Remoção
- Acesso
- Limpeza
- Métodos auxiliares

Limpeza

- Para remover todos os elementos da lista, utilizamos o método `void clear();`.
- Exemplo: `l.clear();`

Sumário

2 Listas
- Declaração e Inicialização
- Inserção
- Remoção
- Acesso
- Limpeza
- Métodos auxiliares

Métodos auxiliares

- `bool empty() const;`: retorna verdadeiro se e somente se a lista está vazia.
- `size_t size() const;`: retorna o número de elemento da lista;
- Exemplos:
  - `l.empty();`
  - `l.size();`

Sumário

3 Outras operações

Sumário

3 Outras operações
- remove e remove_if
- sort
- merge
- unique

remove e remove_if

- O método `remove` recebe um valor e remove todos os elementos de um dado valor.
- Já o método `remove_if` recebe um predicado e remove todos os elementos que possuem aquele predicado.
- Ambos retornam o número de elementos removidos.
- **Importante:** disponíveis apenas no C++20.

remove e remove_if

```
1   #include <iostream>
2   #include <list>
3
4   int main() {
5       list<int> l = {1, 100, 2, 3, 10, 1, 11, -1, 12};
6       auto count1 = l.remove(1);
7       auto count2 = l.remove_if([](int n) -> bool { return n > 10; });
8       cout << count2 << " elements greater than 10 were removed\n";
9       cout << "Finally, the list contains: ";
10      for (int n : l) {
11          cout << n << ' ';
12      }
13      cout << '\n';
14      return 0;
15  }
```

Sumário

3 Outras operações
- remove e remove_if
- sort
- merge
- unique

sort

- O método `sort` ordena uma lista.
- Opcionalmente ele pode receber uma função de comparação.
- Complexidade: $\Theta(n \lg n)$.

sort

```
1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  int main() {
7      // 8 7 5 9 0 1 3 2 6 4
8      list<int> list = {8, 7, 5, 9, 0, 1, 3, 2, 6, 4};
9      // 0 1 2 3 4 5 6 7 8 9
10     list.sort();
11     // 9 8 7 6 5 4 3 2 1 0
12     list.sort(greater<int>());
13     return 0;
14 }
```

Sumário

3 Outras operações
- remove e remove_if
- sort
- merge
- unique

merge

- O método `merge` faz a junção de duas listas **ordenadas**.
- Usar o método sem que as listas estejam ordenadas é um erro de lógica.
- Complexidade: $\Theta(n)$.

merge

```
1  #include <iostream>
2  #include <list>
3
4  using namespace std;;
5
6  int main() {
7      list<int> list1 = {5, 9, 1, 3, 3};
8      list<int> list2 = {8, 7, 2, 3, 4, 4};
9
10     list1.sort(); // list1: 1 3 3 5 9
11     list2.sort(); // list2: 2 3 4 4 7 8
12     list1.merge(list2); // list1: 1 2 3 3 3 4 4 5 7 8 9
13     return 0;
14 }
```

Sumário

3 Outras operações
- remove e remove_if
- sort
- merge
- unique

unique

- O método `unique` remove os elementos repetidos.
- A partir do `C++20` ele retorna o número de elementos removidos.
- Opcionalmente ele pode receber uma função que compara dois elementos e retorna verdadeiro se eles são iguais. Neste caso, `unique` removerá todos os elementos que são iguais de acordo com esta função.
- Complexidade: $\Theta(n)$.

unique

```cpp
#include <iostream>
#include <list>

using namespace std;

int main() {
    std::list<int> l = {1, 2, 2, 3, 3, 2, 1, 1, 2};
    const auto count1 = l.unique();
    l = {1, 2, 12, 23, 3, 2, 51, 1, 2, 2};
    const auto count2 =
        l.unique([mod = 10](int x, int y) { return (x % mod) == (y %
    mod); });
    cout << count1 << ' ' << count2 << endl;
    return 0;
}
```

Sumário

4 Exemplo

Exemplo

José e a eliminação

José e os seus amigos acharam uma nota de 100 reais na rua. Eles combinaram de utilizar um método para decidir quem ficaria com o dinheiro. Para isto, eles se organizam em uma roda e em conjunto escolhem um número x. Começando de João, uma contagem é realizada no sentido horário para determinar qual será a pessoa eliminada e esta contagem finaliza quando o número escolhido entre eles é alcançado. Por exemplo, se o número escolhido for 2, a pessoa ao lado de João no sentido horário é eliminada. Caso o número escolhido seja 1, o próprio João é eliminado.

Exemplo

José e a eliminação

Após cada eliminação uma outra contagem, considerando apenas as pessoas que não foram eliminadas e partindo da pessoa próxima da última eliminada na rodada anterior considerando o sentido horário, uma nova contagem utilizando o mesmo número x é realizada para determinar o próximo a ser eliminado.

Exemplo

José e a eliminação

A pessoa que ficará com o dinheiro é aquela que restou após as sucessivas eliminações. Note que, dependendo do número escolhido, o processo de contagem pode dar várias voltas na roda.

Exemplo

José e a eliminação

Para exemplificar este processo, suponha que existam 5 pessoas; João, José, Maria, Pedro e Helena; organizadas no sentido horário e que o número escolhido seja $x = 2$.

- Na primeira rodada, a contagem inicia de João e José é eliminado.
- Na segunda rodada, a contagem inicia de Maria e Pedro é eliminado.
- Na terceira rodada, a contagem inicia de Helena e João é eliminado.
- Na quarta e última rodada, a contagem inicia de Maria e Helena é eliminada. Portanto, Maria é a detentora da nota de 100.

Exemplo

Entrada

A primeira linha da entrada contém dois inteiros separados por um espaço: n $(1 \leq n \leq 100)$, indicando o número de pessoas a participarem do processo de eliminação, e x $(1 \leq x \leq 1000)$, oo número escolhido pelos participantes.

As próximas n linhas contém o nome de cada pessoa e representam a ordem delas em sentido horário. Os nomes são distintos, possuem apenas letras maiúsculas e estão limitados à 30 caracteres. É garantido que João é a primeira pessoa listada.

Exemplo

Saída

Seu programa deverá determinar o nome da pessoa restante após os processos de eliminação.

Exemplo

Entrada/Saída esperada

Entrada:

```
5 2
JOAO
JOSE
MARIA
PEDRO
HELENA
```

Saída:

```
MARIA
```

Exemplo

Entrada/Saída esperada

Entrada:

```
5 7
JOAO
JOSE
MARIA
PEDRO
HELENA
```

Saída:

```
PEDRO
```

Resolução

- Para resolver este problema basta construir a lista com os nomes dados e simular o processo de eliminação.
- O único cuidado é, ao iterar sobre a lista, caso chegue-se no fim, devemos voltar ao início.
- Uma otimização é pegar o número de passos necessários para eliminar uma pessoa e tirar o resto pelo tamanho da lista, para evitar múltiplas voltas.

Resolução

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int n, x;
6  list<string> l;
```

Resolução

```
8  void read_input() {
9      cin >> n >> x;
10     for (int i = 0; i < n; i++) {
11         string s;
12         cin >> s;
13         l.emplace_back(s);
14     }
15 }
```

Resolução

```
17  void solve() {
18      auto it = l.begin();
19      while (l.size() > 1) {
20          int steps = (x % l.size() == 0) ? l.size() : x % l.size();
21          auto aux = it;
22          for (int i = 1; i < steps; i++) {
23              aux++;
24              if (aux == l.end()) {
25                  aux = l.begin();
26              }
27          }
28          it = l.erase(aux);
29          if (it == l.end())
30              it = l.begin();
31      }
32      cout << l.front() << endl;
33  }
```

Resolução

```
34
35 int main() {
36     read_input();
37     solve();
38     return 0;
39 }
```

Sumário

5 Referências

Referências

cppreference, *cppreference.com*, `https://en.cppreference.com/`, Acessado em 11/2022.