Listas Lineares com Alocação Sequencial

Estrutura de Dados — QXD0010

Roberto Cabral
rbcabral@ufc.br

Universidade Federal do Ceará

2º semestre/2022

Introdução

Introdução

- Uma estrutura de dados armazena dados na memória do computador a fim de permitir o acesso eficiente dos mesmos.
  - Permite a manipulação eficiente, em tempo e em espaço, dos dados armazenados através de operações específicas.

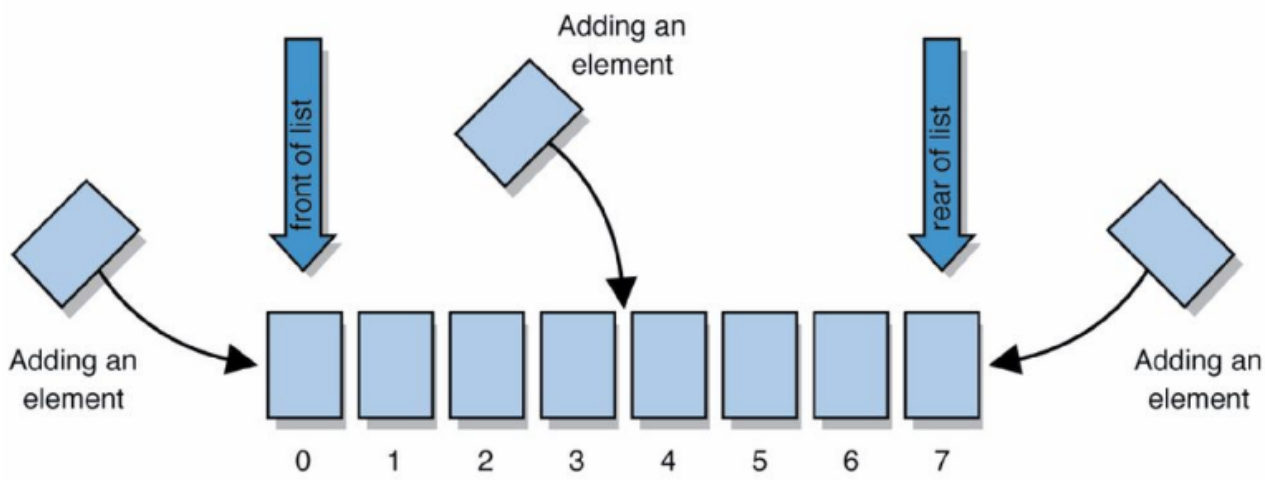
Introdução

- Uma estrutura de dados armazena dados na memória do computador a fim de permitir o acesso eficiente dos mesmos.
  - Permite a manipulação eficiente, em tempo e em espaço, dos dados armazenados através de operações específicas.

- A maioria das estruturas de dados usam como recurso principal a memória primária (RAM) como listas, pilhas, filas, árvores binárias de busca, árvores AVL e árvores rubro-negras.

Lista linear — Definição

- Uma lista linear L é um conjunto de $n \geq 0$ nós (ou células) $L_0, L_1, \ldots, L_{n-1}$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:
  - Se $n > 0$, L_0 é o primeiro nó,
  - Para $0 < k \leq n - 1$, o nó L_k é precedido por L_{k-1}.



Lista linear

- Os elementos de uma lista linear armazenam informações referentes a um conjunto de elementos que se relacionam entre si.
  - Informações sobre os funcionários de uma empresa.
  - Notas de alunos
  - Itens de estoque, etc.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
- Combinar duas ou mais listas lineares em uma só.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
- Combinar duas ou mais listas lineares em uma só.
- Quebrar uma lista linear em duas ou mais.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
- Combinar duas ou mais listas lineares em uma só.
- Quebrar uma lista linear em duas ou mais.
- Copiar uma lista linear em um outro espaço.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao elemento L_k, para k qualquer.
(2) inserir ou remover elementos em qualquer posição da lista linear.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao elemento L_k, para k qualquer.
(2) inserir ou remover elementos em qualquer posição da lista linear.

A operação (1) fica eficiente se a lista é implementada em um vetor (array) em alocação sequencial na memória.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao elemento L_k, para k qualquer.
(2) inserir ou remover elementos em qualquer posição da lista linear.

A operação (1) fica eficiente se a lista é implementada em um vetor (array) em alocação sequencial na memória.

Para a operação (2) é mais adequada a alocação encadeada, com o uso de ponteiros.

Tipos de alocação

O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa na memória de dois nós consecutivos na lista.

- **Alocação sequencial:** dois nós consecutivos na lista estão em posições contíguas de memória.

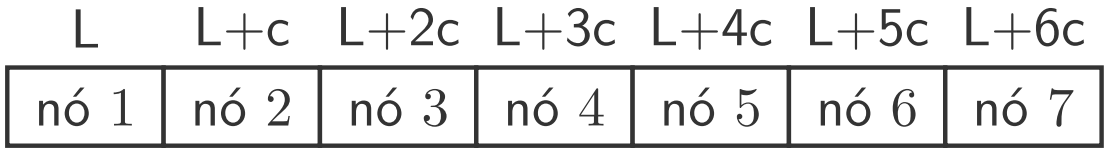
- **Alocação encadeada:** dois nós consecutivos na lista podem estar em posições não contíguas da memória.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Listas Sequenciais

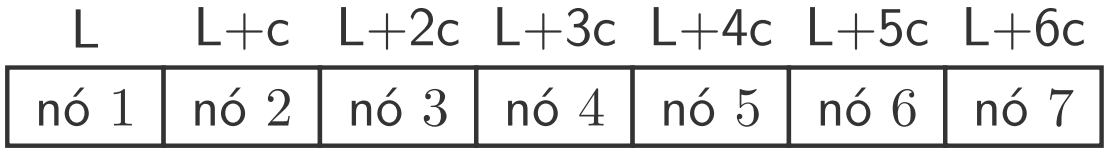
Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.



Listas sequenciais (Vetores)

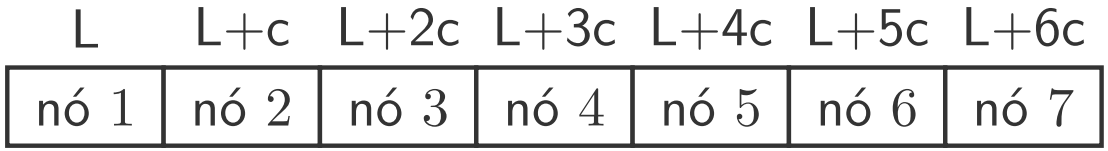
- Nós em posições contíguas da memória.



- Neste caso, o endereço real do $(j+1)$-ésimo nó da lista se encontra c unidades adiante daquele correspondente ao j-ésimo. A constante c é o número de bytes que cada nó ocupa.

Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.



- Neste caso, o endereço real do $(j+1)$-ésimo nó da lista se encontra c unidades adiante daquele correspondente ao j-ésimo. A constante c é o número de bytes que cada nó ocupa.

- A correspondência entre o índice da lista e o endereço real é feita automaticamente pela linguagem de programação quando da compilação do programa.

TAD Lista Sequencial

- O TAD Lista Sequencial tem os seguintes **atributos**:
  - a lista de elementos (implementada como um vetor).
  - a quantidade de elementos atualmente na lista.
  - a capacidade total da lista.
    - Vamos, inicialmente, implementar a lista com uma capacidade fixa.

TAD Lista Sequencial

- O TAD Lista Sequencial tem os seguintes **atributos**:
  - a lista de elementos (implementada como um vetor).
  - a quantidade de elementos atualmente na lista.
  - a capacidade total da lista.
    - Vamos, inicialmente, implementar a lista com uma capacidade fixa.

- O TAD pode ter as seguintes **operações**:
  - Criar lista vazia.
  - Criar lista a partir de uma lista prévia.
  - Liberar lista.
  - Retornar número de elementos.
  - Consultar o tamanho atual da lista.
  - Saber se lista está cheia; ou se ela está vazia.
  - Retornar uma referência para o elemento na posição k.
  - Inserir elemento em uma dada posição da lista.
  - Remover elemento da lista, e remover elemento em certa posição.
  - Buscar primeira ocorrência de um elemento e retorna índice.
  - Limpar a lista, deixando-a vazia.
  - Retornar uma sublista da lista.

Implementação em C++

Arquivo SeqList.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #ifndef SEQLIST_H
2  #define SEQLIST_H
3
4  class SeqList {
5  private:
6      int *vec;          // ponteiro para o array de inteiros
7      int size_vec;      // número de elementos na lista
8      int capacity_vec; // capacidade total da lista
9
10 public:
11     // Construtor: recebe como argumento a capacidade n
12     // Se n <= 0, o construtor seta o valor default 10
13     SeqList(int n);
14
15     // Construtor de cópia
16     SeqList(const SeqList& list);
17
18     // Destrutor: libera memoria alocada
19     ~SeqList();
```

Arquivo SeqList.h (cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
20      // Recebe um inteiro x como argumento e o adiciona
21      // logo apos o ultimo elemento da lista.
22      // Retorna 'true' se for bem sucedido, ou 'false'
23      // caso contrario. Nenhum elemento deve ser adicionado
24      // se a lista estiver cheia.
25      bool add(int x);
26
27      // Remove o primeiro valor x que estiver na lista
28      // Se nenhum valor for removido, retorna false;
29      // caso contrario retorna true
30      bool remove(int x);
31
32      // Busca um elemento x e retorna seu indice se
33      // ele existir; ou -1 caso contrario
34      int search(int x);
35
36      // Retorna uma referência para o elemento no indice k.
37      // Se o índice k for inválido, lança uma exceção
38      int& at(int k);
```

Arquivo SeqList.h (cont.)


```
39      // Retorna o numero de elementos na lista
40      int size();
41
42      // Retorna 'true' se lista estiver cheia;
43      // ou 'false' caso contrario
44      bool isFull();
45
46      // Retorna 'true' se lista estiver vazia;
47      // ou 'false' caso contrario
48      bool isEmpty();
49
50      // Deixa a lista vazia, com size() == 0
51      void clear();
52
53      // Retorna a lista como uma std::string
54      // Ex.: se a lista tiver elementos 2,5,7,8 entao a
55      // string "[2,5,7,8]" eh retornada. Se a lista
56      // estiver vazia, deve ser retornada a string "[]"
57      std::string toString();
```

Arquivo SeqList.h (cont.)


```
58      // Retorna uma sublista da lista original
59      // contendo os elementos no intervalo de
60      // índices [from,...,to].
61      // Se os índices forem inválidos, lança uma exceção.
62      SeqList sublist(int from, int to);
63
64  };
65
66  #endif
```

Programa cliente `main.cpp`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream>
2  #include "SeqList.h"
3  using namespace std;
4
5  int main() {
6    SeqList list1(15), list2(10);
7
8    for(int i = 0; i < 10; ++i) {
9      list1.add(i);
10     list2.add(i * 2);
11   }
12   // imprime [0,1,2,3,4,5,6,7,8,9]
13   cout << list1.toString() << endl;
14   // imprime [0,2,4,6,8,10,12,14,16,18]
15   cout << list2.toString() << endl;
16
17   SeqList list3 = list2.sublist(3,7);
18   // imprime // [6,8,10,12,14]
19   cout << list3.toString() << endl;
20 }
```

Exercícios

Exercício

Exercício: Implementar as funções-membro da classe `SeqList`. Você pode implementar as funções-membro dentro da própria classe ou pode implementá-las em um arquivo-fonte separado chamado `SeqList.cpp`.

Exercício

Implemente as seguintes operações adicionais na Lista Sequencial:

- `void replaceAt(int x, int k)`: Troca o elemento no índice k pelo elemento x (somente se $0 \leq k \leq size\_vec - 1$)

- `void removeAt(int k)`: Remove o elemento com índice k na lista. Deve-se ter $0 \leq$ `k` $\leq$ `size_vec-1`; caso contrario, a remoção não é realizada.

- `bool insertAt(int x, int k)`: Adiciona o elemento x no índice k (somente se $0 \leq k \leq size\_vec$ e $size\_vec < capacity\_vec$). Antes de fazer a inserção, todos os elementos do índice k em diante são deslocados uma posição para a direita.

- `void removeAll(int x)`: Remove todas as ocorrências do elemento x na lista.

Lista Sequencial Redimensionável

Lista Sequencial Redimensionável

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Atividade: Implementar uma lista sequencial sem limite de capacidade.
A lista deve aumentar de tamanho sempre size() atingir a capacidade total.

Lista Sequencial Redimensionável

Atividade: Implementar uma lista sequencial sem limite de capacidade.
A lista deve aumentar de tamanho sempre size() atingir a capacidade total.

Implemente as seguintes funções-membros públicas na classe `SeqList`:

- `void reserve(int n)`
  Solicita que a capacidade do vetor seja maior ou igual a n.

  Se n for maior que a capacidade atual do vetor, a função faz com que a lista realoque seu armazenamento aumentando sua capacidade.
  Em todos os outros casos, a chamada da função não causa uma realocação e a capacidade do vetor não é afetada.

- `void push_back(int elemento)`
  Adiciona um elemento ao final da lista.

- `void insert(int elemento, int i)`
  Insere um elemento na posição i da lista, somente se $0 \leq i \leq size()$.

Introdução à STL

Standard Template Library

- A STL é uma parte do padrão C++ aprovada em 1997/1998 e estende o núcleo do C++ fornecendo componentes gerais.
  - A STL fornece o tipo de dado `std::string`, diferentes estruturas para armazenamento de dados, classes para entrada/saída e algoritmos utilizados frequentemente por programadores.

Parte lógica	**Descrição**
Containers	Gerenciam coleções de objetos
Iteradores	Percorrem elementos das coleções de objetos
Algoritmos	Processam elementos da coleção

Containers

- Containers são como a STL chama suas estruturas de dados. Dentro dos containers podemos guardar vários dados de um mesmo tipo.
  - Cotainers armazenam qualquer tipo de dado válido.

Containers

- Containers são como a STL chama suas estruturas de dados. Dentro dos containers podemos guardar vários dados de um mesmo tipo.
  - Cotainers armazenam qualquer tipo de dado válido.

- Cada tipo de container possui uma estratégia diferente para organização interna de seus dados.
  - Por isso, cada container possui vantagens e desvantagens em diferentes situações.

Containers

- Containers são como a STL chama suas estruturas de dados. Dentro dos containers podemos guardar vários dados de um mesmo tipo.
  - Cotainers armazenam qualquer tipo de dado válido.

- Cada tipo de container possui uma estratégia diferente para organização interna de seus dados.
  - Por isso, cada container possui vantagens e desvantagens em diferentes situações.

- Containers sequenciais: são aqueles utilizados para representar sequências de elementos. Em uma sequência de elementos, **cada elemento deve ter uma posição específica**.
  Exemplos: `vector`, `list`, `deque`, `forward_list`

std::vector

std::vector

- `std::vector` é um container sequencial implementado como um array redimensionável.
  - O `vector` permite acesso randômico aos seus elementos individuais e pode aumentar dinamicamente. O gerenciamento de memória é feito automaticamente pelo container.

- Definido na biblioteca: <vector>
  - `#include <vector>`

Construindo um std::vector

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <iostream> // vector01.cpp
#include <vector>
using namespace std;

int main() {
    // cria um vector vazio
    vector<int> primeiro;

  for(int i = 1; i <= 9; i++)
    primeiro.push_back(i);

  vector<int>::iterator it;
  for(it = primeiro.begin(); it != primeiro.end(); it++) {
    if(*it == 5) {
      primeiro.erase(it);
      break;
    }
  }

  for(int e : primeiro)
    cout << e << " ";
  cout << endl;

}
```

Inserindo e removendo no final

- `void push_back(const value_type& val)`
  Adiciona um novo elemento ao final do `vector`, depois do seu último elemento atual.

```
vector<int> myVector; // um vetor vazio
myVector.push_back(30);
myVector.push_back(25);
myVector.push_back(80);
```

Inserindo e removendo no final

- `void push_back(const value_type& val)`
  Adiciona um novo elemento ao final do `vector`, depois do seu último elemento atual.

```
vector<int> myVector; // um vetor vazio
myVector.push_back(30);
myVector.push_back(25);
myVector.push_back(80);
```

- `void pop_back()`
  Remove o último elemento no `vector`, decrementando seu tamanho em 1.

```
myVector.pop_back();
myVector.pop_back();
```

size e resize

- `size_type size()`
  Retorna o número de elementos no vector.
  Exemplo: `myVector.size()`

- `void resize(size_type n)`
  Modifica o vector de modo que ele contenha `n` elementos.
  - Se `n` for menor que o `size()` atual, o conteúdo é reduzido aos primeiros `n` elementos, removendo os demais.
  - Se `n` for maior que o `size()` atual, o conteúdo é expandido inserindo quantos elementos forem necessários até atingir o tamanho `n`. O elemento a ser inserido pode ser especificado na função (`resize(n,val)`), caso contrário ele é um valor *default*.

resize — exemplo


```
1  #include <iostream> // vector02.cpp
2  #include <vector>
3  using namespace std;
4
5  void print(vector<int>& vec) {
6      for(int e : vec) cout << e << " ";
7      cout << endl;
8  }
9
10 int main () {
11     vector<int> myVector;
12
13     for(int i = 1; i <= 8; i++)
14         myVector.push_back(i);
15
16     myVector.resize(10);
17     print(myVector); // 1 2 3 4 5 6 7 8 0 0
18     myVector.resize(5);
19     print(myVector); // 1 2 3 4 5
20     myVector.resize(8,100); // 1 2 3 4 5 100 100 100
21     print(myVector);
22 }
```

Acesso randômico aos elementos

- O `operator[]` permite acesso randômico do elemento na posição i do vector (`0` $\leq$ `i` $\leq$ `size()-1`).
  Ele devolve uma referência para o elemento requerido.
  - Se o índice i requisitado estiver fora do intervalo válido, o comportamento será indefinido. **Nunca faça isso**.

- `value_type& at(size_type i)`
  Retorna uma referência para o elemento na posição i do vector. Esta função checa se i está dentro do intervalo `0..size()-1` e, caso não esteja, lança uma exceção.

Acesso randômico aos elementos


```
1  #include <iostream> // vector03.cpp
2  #include <vector>
3  using namespace std;
4
5  int main () {
6      vector<int> myVector(7);
7
8      for(int i = 0; i < 7; ++i)
9          myVector[i] = i+1;
10
11     for(size_t i = 0; i < myVector.size(); ++i)
12         cout << myVector[i] << " ";
13     cout << endl;
14 }
```

Iteradores

Iteradores

- A opção de se acessar elementos de um container através do `operator[]` é restrita apenas a containers de acesso aleatório, como o `vector`.

- Os containers `list` e forward_list, por exemplo, não possuem o `operator[]`. Como são implementados com listas, seus dados não podem ser acessados randomicamente.

- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de iteradores.

Iteradores

- A opção de se acessar elementos de um container através do `operator[]` é restrita apenas a containers de acesso aleatório, como o `vector`.

- Os containers `list` e forward_list, por exemplo, não possuem o `operator[]`. Como são implementados com listas, seus dados não podem ser acessados randomicamente.

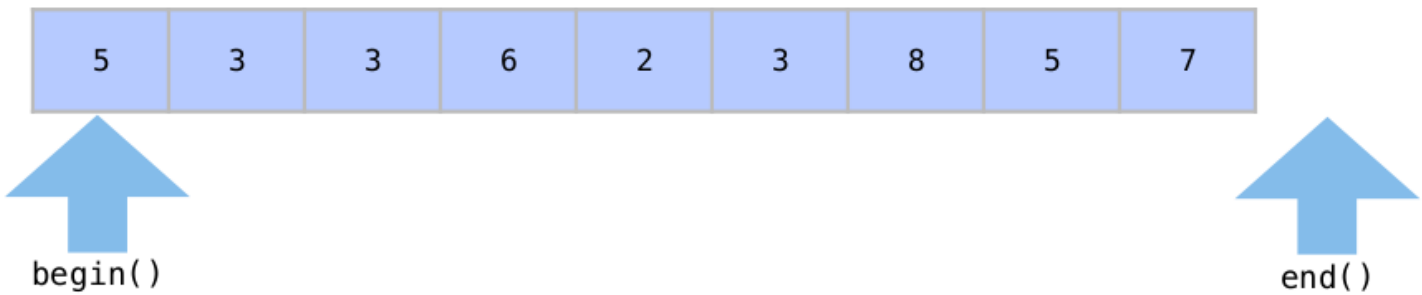
- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de iteradores.

Definição: Os iteradores são objetos que caminham (iteram) sobre elementos de containers.

- Eles funcionam como um ponteiro especial para elementos de containers: enquanto um ponteiro representa uma posição na memória, um iterador representa uma posição em um container.

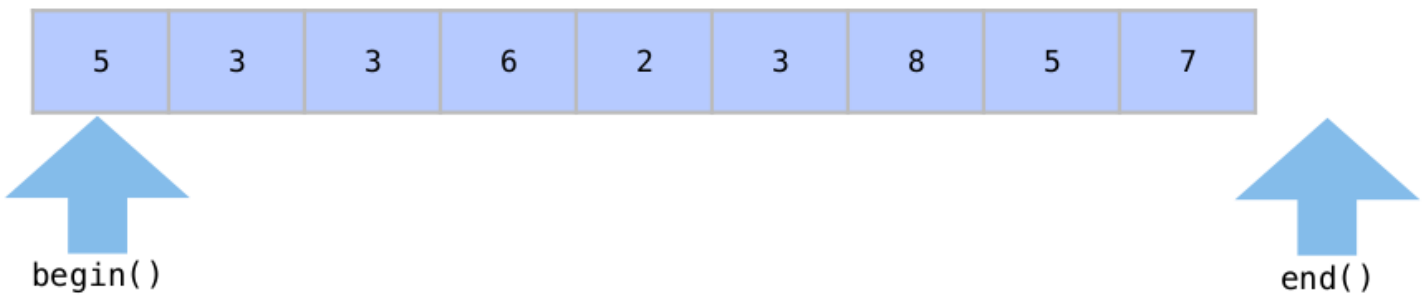
Iteradores

- Os iteradores podem ser gerados através de funções de um container.
  - Suponha um container chamado `i`
  - O comando `i.begin()` retorna um iterador para o primeiro elemento deste container `c`.
  - O comando `i.end()` retorna um iterador para uma posição após o último elemento do container `i`.



Iteradores

- Os iteradores podem ser gerados através de funções de um container.
  - Suponha um container chamado `i`
  - O comando `i.begin()` retorna um iterador para o primeiro elemento deste container `c`.
  - O comando `i.end()` retorna um iterador para uma posição após o último elemento do container `i`.



- Por exemplo, o container `vector` possui funções `begin()` e `end()` que retornam iteradores, para o primeiro elemento e para uma posição após o último elemento, respectivamente.

Operações comuns com iteradores

- Supondo um iterador chamado `i`, a tabela abaixo apresenta algumas funções que são comuns a iteradores.

Função	Retorna
`*i`	Retorna o elemento na posição do iterador
`++i`	Avança o iterador para o próximo elemento
`==`	Confere se dois iteradores apontam para mesma posição
`!=`	Confere se dois iteradores apontam para posições diferentes

Exemplo de uso de iteradores – vector

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // vector04.cpp
2  #include <vector>
3  using namespace std;
4
5  int main () {
6      vector<int> vec(7);
7
8      vector<int>::iterator it; // definição de um iterador
9      int val = 1;
10
11     for(it = vec.begin(); it != vec.end(); ++it) {
12         *it = val;
13         val++;
14     }
15
16     for(it = vec.begin(); it != vec.end(); ++it)
17         cout << *it << " ";
18
19     cout << endl;
20 }
```

Exemplo de uso de iteradores – vector

Uso da palavra-chave auto

```
1  #include <iostream> // vector05.cpp
2  #include <vector>
3  using namespace std;
4
5  int main () {
6      vector<int> vec(7);
7
8      int val = 1;
9
10     for(auto it = vec.begin(); it != vec.end(); ++it) {
11         *it = val;
12         val++;
13     }
14
15     for(auto it = vec.begin(); it != vec.end(); ++it)
16         cout << *it << " ";
17
18     cout << endl;
19 }
```

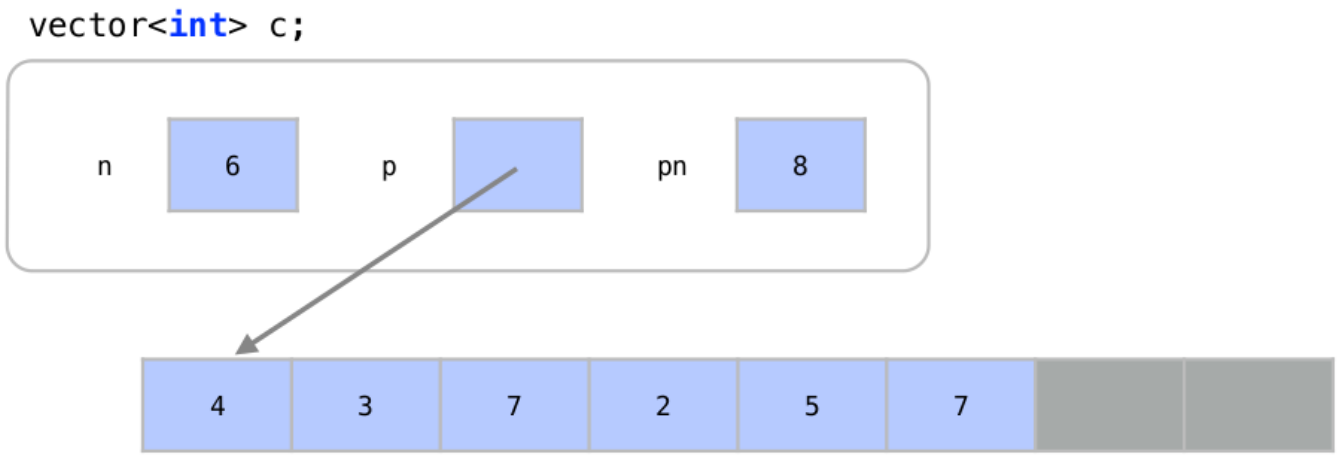
Funções de `vector` que usam iteradores

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

`insert`

- Considere um `vector` gerado pelo código abaixo.

```
1  vector<int> c;
2  c.push_back(4);
3  c.push_back(3);
4  c.push_back(7);
5  c.push_back(2);
6  c.push_back(5);
7  c.push_back(7);
```

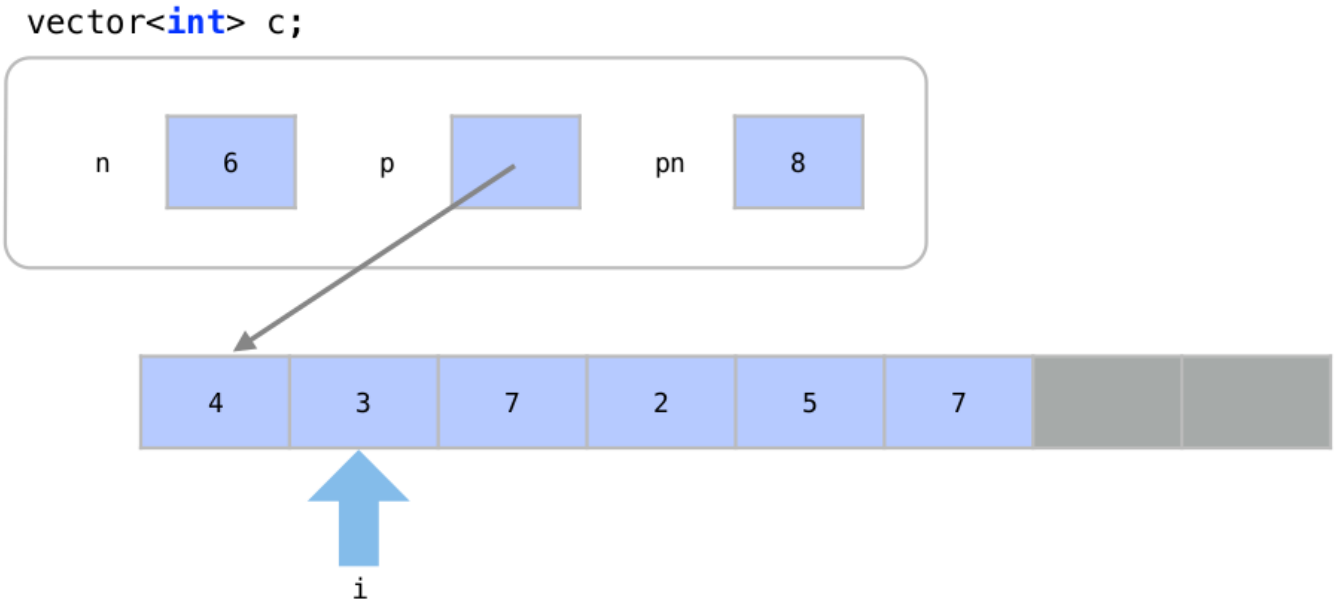


Funções de `vector` que usam iteradores

`insert`

- Considere também que geramos um iterador i apontando para o segundo elemento da sequência.

```
1  vector<int>::iterator i;
2  i = c.begin();
3  ++i;
```



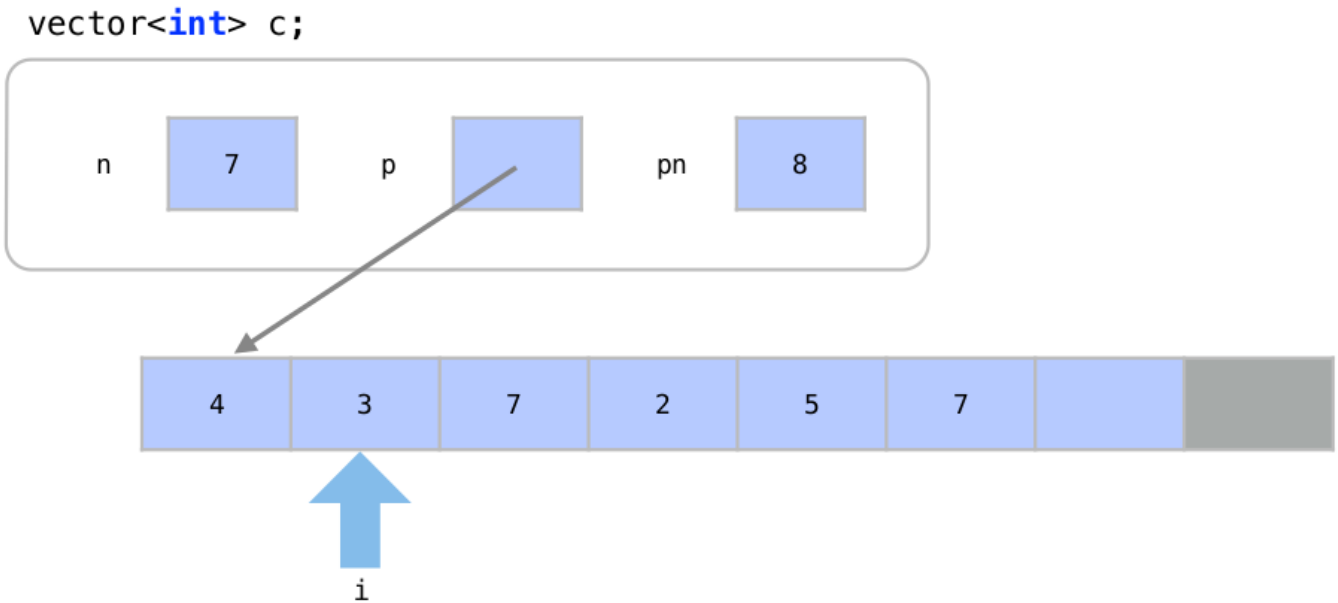
Funções de `vector` que usam iteradores

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

`insert`

- Queremos inserir 11 na posição 2 do vector. Uma sequência de passos deve ser executada.

1. Primeiro, a variável n é incrementada. Em alguns casos, pode ser que um novo array precise ser alocado, levando a um custo $O(n)$.

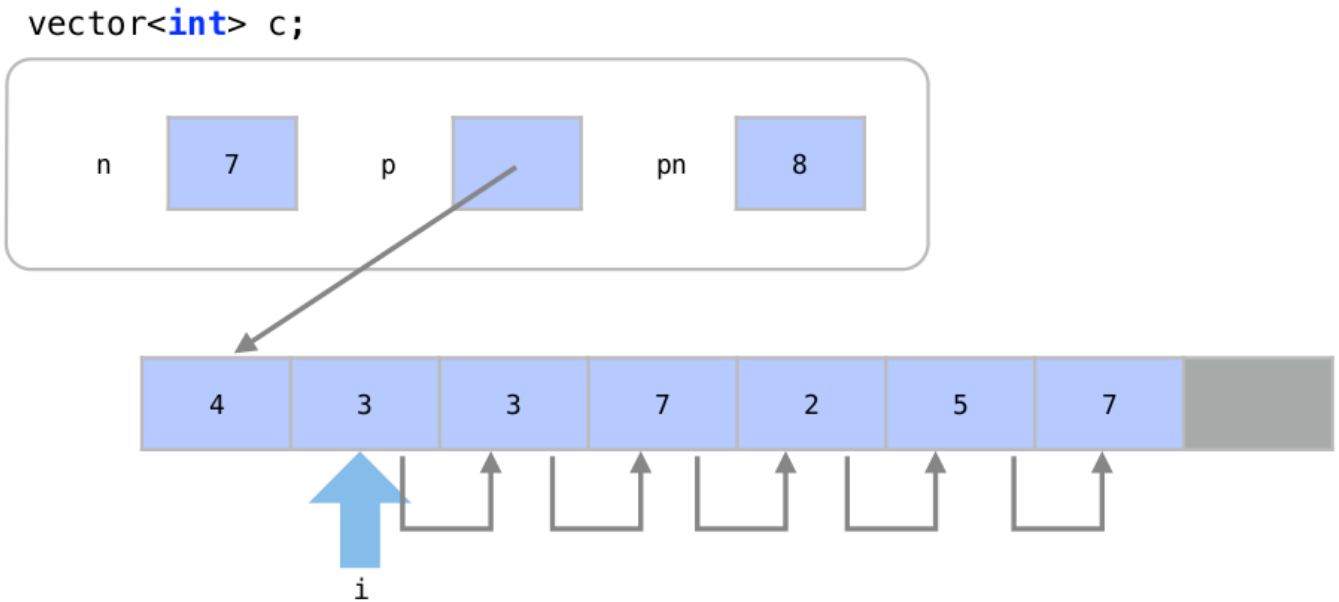


Funções de `vector` que usam iteradores

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

`insert`

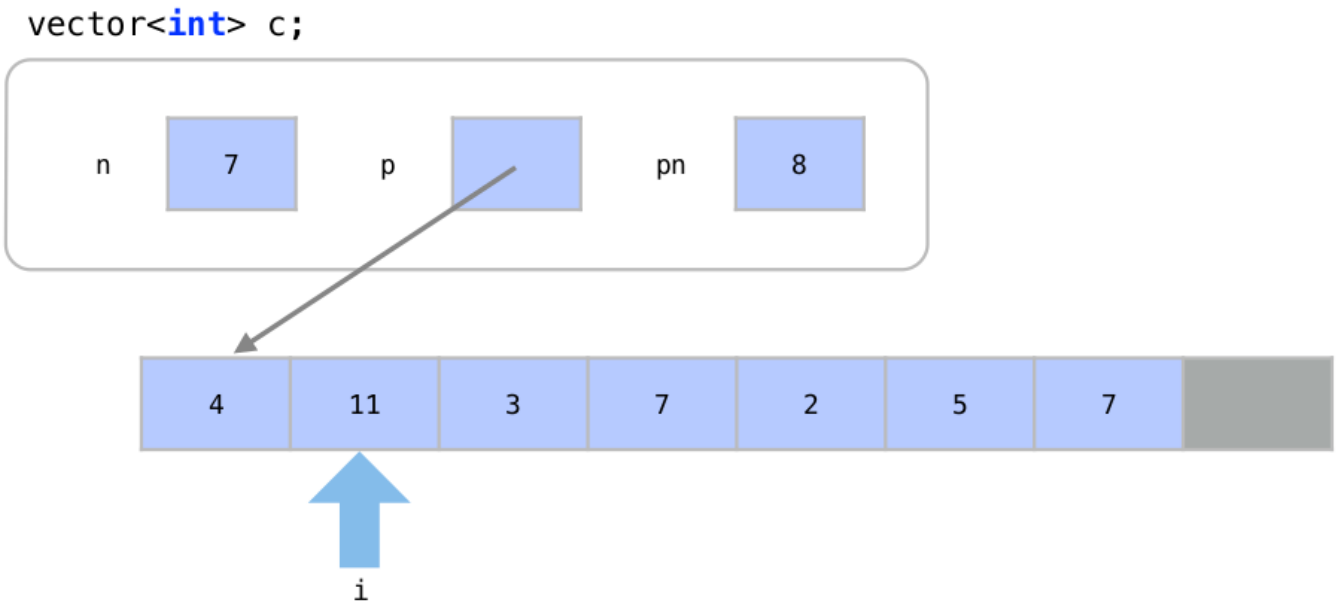
2. Todos os elementos entre i e o penúltimo elemento precisam ser deslocados para a próxima posição do arranjo. Esta operação tem um custo $O(n - i)$. Se i for o último elemento, temos um melhor caso $O(1)$. Se i for o primeiro elemento, temos um pior caso $O(n)$.



Funções de `vector` que usam iteradores

`insert`

3. O elemento 11 pode ser inserido na posição i, com custo $O(1)$.



Devido a todas as movimentações, esta operação completa de inserção no meio da sequência tem um custo $O(n)$.

Funções de `vector` que usam iteradores

`insert`

- `iterator insert(iterator it, const value_type& val)`
  Insere o valor `val` na posição indicada pelo iterador `it`.
  Esta função retorna um iterador apontando para o objeto recém inserido.

Funções de `vector` que usam iteradores

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

`insert`

- `iterator insert(iterator it, const value_type& val)`
  Insere o valor `val` na posição indicada pelo iterador `it`.
  Esta função retorna um iterador apontando para o objeto recém inserido.

- `iterator insert(iterator it, int n, const value_type& val)`
  Insere o valor `val` um total de `n` vezes na posição indicada pelo iterador `it`. Esta função retorna um iterador apontando para o objeto recém inserido.

Funções de `vector` que usam iteradores

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

`insert`

- `iterator insert(iterator it, const value_type& val)`
  Insere o valor `val` na posição indicada pelo iterador `it`.
  Esta função retorna um iterador apontando para o objeto recém inserido.

- `iterator insert(iterator it, int n, const value_type& val)`
  Insere o valor `val` um total de `n` vezes na posição indicada pelo iterador `it`. Esta função retorna um iterador apontando para o objeto recém inserido.

- `iterator insert(iterator it, InputIterator first, InputIterator last)`
  Essa operação insere todos os elementos entre o iterador `first` (inclusive) e o iterador `last` (exclusive) no vector, a partir da posição indicada pelo iterator `it`. Os iteradores `first` e `last` pertencem a um outro container.

Funções de vector que usam iteradores

insert

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6      vector<int> vec(3, 10); //vec: 10,10,10
7      vector<int>::iterator it;
8
9      it = vec.begin();
10     it = vec.insert(it, 20); //vec: 20,10,10,10
11
12     vec.insert(it, 2, 30); //vec: 30,30,20,10,10,10
13
14     // "it" no longer valid, get a new one:
15     it = vec.begin();
16
17     vector<int> z(2, 40); //z: 40,40
18     vec.insert(it+2, z.begin(), z.end());
19     //vec: 30,30,40,40,20,10,10,10
20 }
```

Funções de `vector` que usam iteradores

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

`erase`

- `iterator erase (iterator position)`
  Remove do `vector` um único elemento, que é o elemento apontado pelo iterador `position`.
  - Essa função decrementa o tamanho do vector em 1 unidade.
  - Essa função devolve um iterador apontando para a nova localização do elemento que seguia o último elemento apagado pela chamada de função. Este elemento é o `end()` se a operação apagou o último elemento na sequência.

Funções de vector que usam iteradores

erase

```
1  #include <iostream> // vector07.cpp
2  #include <vector>
3  using namespace std;
4
5  int main() {
6      vector<int> vec;
7
8      // set some values (from 1 to 10)
9      for(int i = 1; i <= 10; i++)
10         vec.push_back(i);
11
12     // erase the 6th element
13     auto it = vec.erase(vec.begin() + 5);
14     vec.erase(it); // erase the number 7
15
16     cout << "vec contains: ";
17     for(size_t i = 0; i < vec.size(); ++i)
18         cout << " " << vec[i];
19     cout << endl;
20 }
```

Mais informações

Sobre vector e outros containers:

- https://www.cplusplus.com/reference/vector/vector/
- https://www.geeksforgeeks.org/vector-in-cpp-stl/

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM