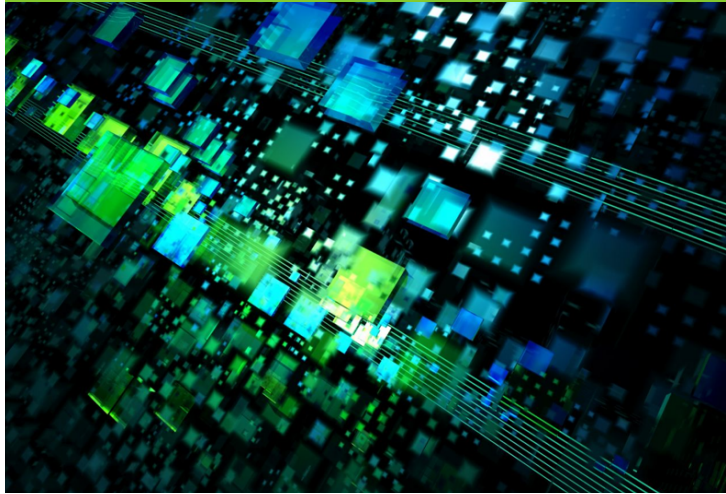


# Programação de Computadores Estruturas de Dados



Alan  
de  
Freitas

## Templates

- São comuns situações em que operações muito parecidas podem ser executadas em diferentes tipos de dados
- Isto é especialmente verdade entre os tipos de dados aritméticos
- Por exemplo, o algoritmo para encontrar o máximo ou mínimo em um conjunto de `ints` é muito similar ao algoritmo para os tipos de dado `double`, `short`, `float` ou `long`

Considere o código abaixo que encontra o maior elemento entre 3 `ints`

```
int maximo(int a, int b, int c){  
    int max = a;  
    if (b > max){  
        max = b;  
    }  
    if (c > max){  
        max = c;  
    }  
    return max;  
}
```

Veja agora o código que faz o mesmo para dados do tipo `double`.

```
int maximo(int a, int b, int c){  
    int max = a;  
    if (b > max){  
        max = b;  
    }  
    if (c > max){  
        max = c;  
    }  
    return max;  
}
```

```
double maximo(double a, double b, double c){  
    double max = a;  
    if (b > max){  
        max = b;  
    }  
    if (c > max){  
        max = c;  
    }  
    return max;  
}
```

Com exceção do tipo de dado, os códigos são idênticos.

Não é difícil perceber que o mesmo código funcionaria para qualquer outro tipo de dado. Precisaríamos apenas alterar o tipo.

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Já vimos neste curso vários algoritmos que poderiam ser generalizados para qualquer tipo de dado.

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

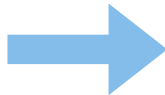
```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Em C++, os templates nos permitem definir uma função que funciona para mais de um tipo de dado como no exemplo abaixo:

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```



```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}

char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}

float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}

int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Definir um template equivale a ter definido a função para qualquer tipo de dado possível.

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```



```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}

char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}

float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}

int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Por exemplo, se a função chamadora tem um comando `maximo(4,2,6)`, o compilador gera uma versão da função `maximo` que aceita `ints`.

`maximo(4,2,6)`

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Por exemplo, se a função chamadora tem um comando `maximo(4,2,6)`, o compilador gera uma versão da função `maximo` que aceita `ints`.

`maximo(4,2,6)`

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

## Templates

- Representam uma coleção de definições
- Definições são geradas sob demanda pelo compilador
- Recurso poderoso de reutilização de código em C++
- Com eles, conseguimos fazer o que chamamos de **programação genérica**
- Retornaremos a este tópico mais vezes neste curso

## Standard Template Library

- Contribuição mais relevante e mais representativa de programação genérica em C++ é a STL (Biblioteca Padrão de Templates)
- Parte do padrão C++ (aprovado em 1997/1998)
- Estende o núcleo de C++ fornecendo componentes gerais

# Standard Template Library

- Como alguns exemplos, a STL fornece
  - O tipo de dado `string`
  - Diferentes estruturas de dados para armazenamento de dados
  - Classes para entrada/saída
  - Algoritmos utilizados frequentemente

## Containers

As estruturas de dados são representadas com a STL através de containers.



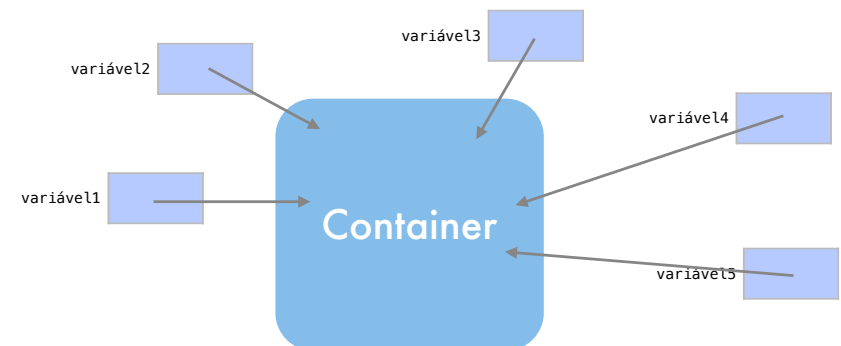
# Standard Template Library

A STL se baseia em três partes fundamentais:

Containers	Gerenciam coleções de objetos
Iteradores	Percorrem elementos das coleções de objetos
Algoritmos	Processam elementos da coleção

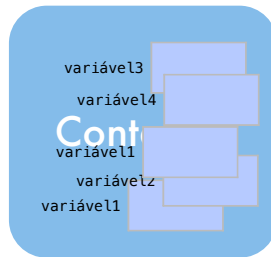
## Containers

Como a analogia sugere, dentro de um container, podemos guardar várias variáveis de qualquer tipo.



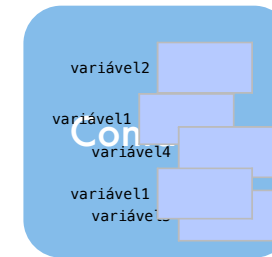
# Containers

Porém, cada container tem suas próprias características pois utiliza internamente diferentes estruturas de dados para guardar estas variáveis.



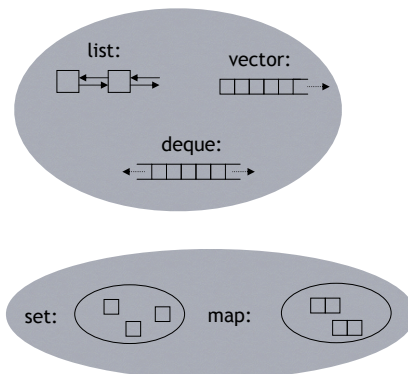
# Containers

Ou seja, 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

- 2 tipos básicos:
  - Containers sequenciais: representa listas onde cada elemento tem uma posição específica
  - Containers associativos: representa conjuntos onde a posição interna de um elemento depende de seu valor



# Containers

- Os containers são representados através de objetos que utilizam templates
- Os templates permitem que o container seja utilizado em qualquer tipo de dado
- Grosso modo, os objetos são recursos similares aos `structs`, porém, além de ter suas próprias variáveis, os objetos possuem suas próprias funções

## Funções-membro comuns aos contêineres

Se está vazio <code>empty()</code>	Tamanho <code>size()</code>	Compara <, >, >=, <=, ==, !=
Troca <code>swap()</code>	Tamanho máximo <code>max_size()</code>	Início <code>begin()</code>
Fim <code>end()</code>	Início (reverse) <code>rbegin()</code>	Fim (reverse) <code>rend()</code>
Apaga elementos <code>erase(i)</code>	Limpa container <code>clear()</code>	

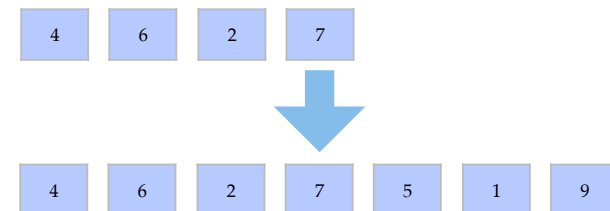
Imagine a sequência de elementos abaixo. Imagine que o vector com esta sequência se chama `c`.



## Vector

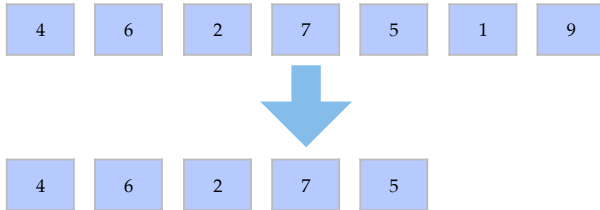
- O vector é talvez o tipo de container de sequência mais utilizado
- Este container tem funções para acessar, remover ou inserir elementos no fim da sequência de elementos de maneira eficiente
- Exige a inclusão do cabeçalho `<vector>`

```
c.push_back(5);  
c.push_back(1);  
c.push_back(9);
```



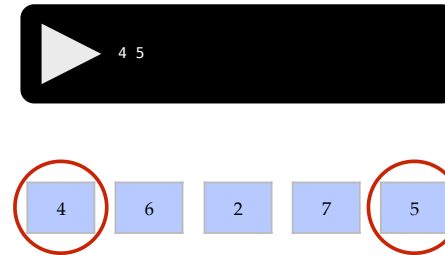
Novos elementos podem ser inseridos ao final de `c` com a função `push_back`.

```
c.pop_back();  
c.pop_back();
```



Elementos podem ser removidos do final de `c` com a função `pop_back`.

```
cout << c.front() << " " << c.back() << endl;
```



O primeiro e último elementos de `c` podem ser acessados com as funções `front` e `back`.

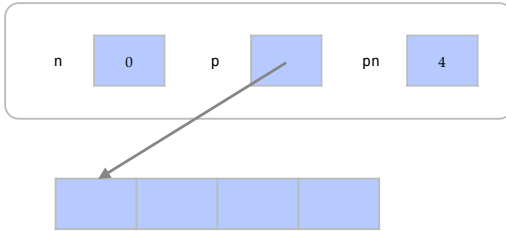
## Vector - Como funciona

- Apesar de todas as simplificações oferecidas pelos templates, os containers são complexos internamente
- Todo container é representado internamente através de recursos que já conhecemos, como ponteiros e arranjos
- Este recursos são organizados para formarem uma estrutura de dados para organizar os elementos

```
vector<int> c;
```

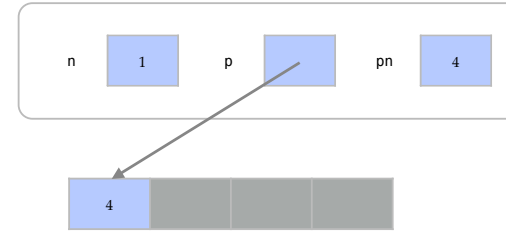
Com este comando, podemos criar um vector vazio. Internamente, o vector tem um ponteiro um dois números inteiros.

```
vector<int> c;
```



Um número inteiro guarda o número de elementos no container enquanto o ponteiro aponta para um arranjo de elementos na memória. O segundo número inteiro guarda o tamanho deste arranjo.

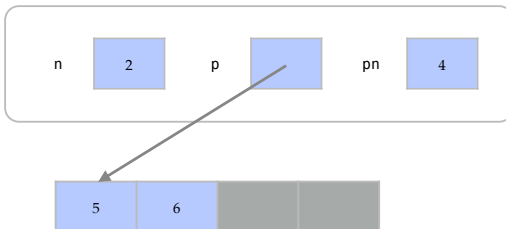
```
vector<int> c;
```



```
c.push_back(4);
```

Quando inserimos um elemento no fim do container, este é inserido na posição `n` do arranjo e `n` é incrementado.

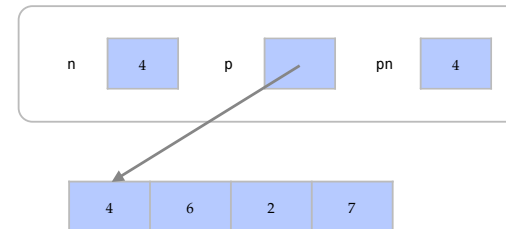
```
vector<int> c;
```



```
c.push_back(6);
```

Como o arranjo já tem espaço para mais que `n` elementos, mais elementos podem ser inseridos sem necessidade de se alocar mais memória.

```
vector<int> c;
```

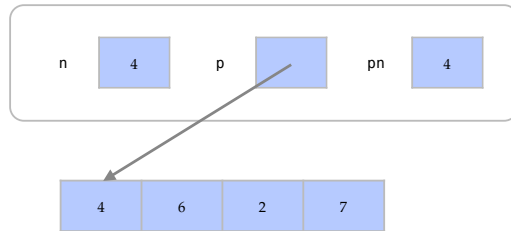


```
c.push_back(2);  
c.push_back(7);
```

Eventualmente, o número de elementos será igual ao tamanho do arranjo. Neste caso, não podemos colocar mais elementos sem alocar mais memória.



```
vector<int> c;
```

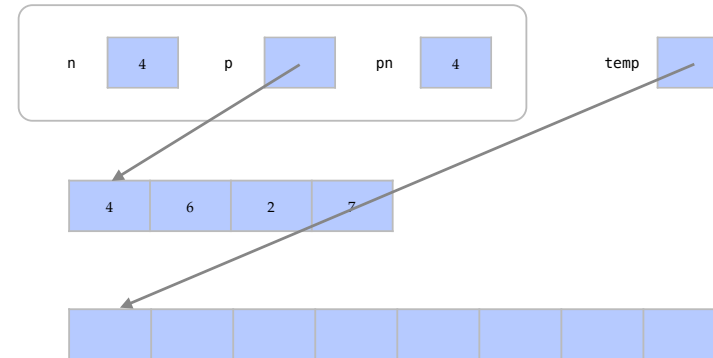


```
c.push_back(5);
```

Queremos agora inserir o elemento 5 ao container.  
Para isto, ocorrem os seguintes passos...

```
vector<int> c;
```

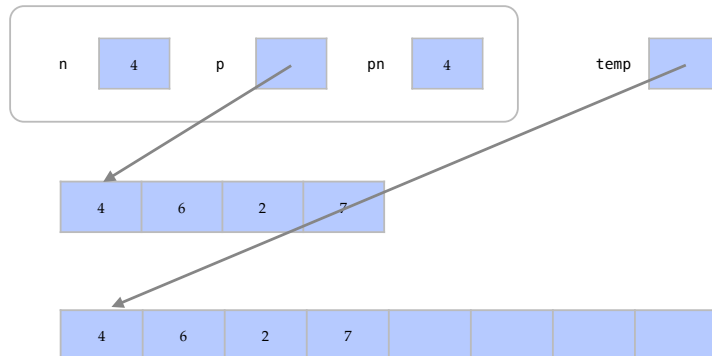
```
c.push_back(5);
```



Alocamos espaço para um arranjo com o dobro de  
elementos e fazemos um ponteiro temporário apontar  
para ele.

```
vector<int> c;
```

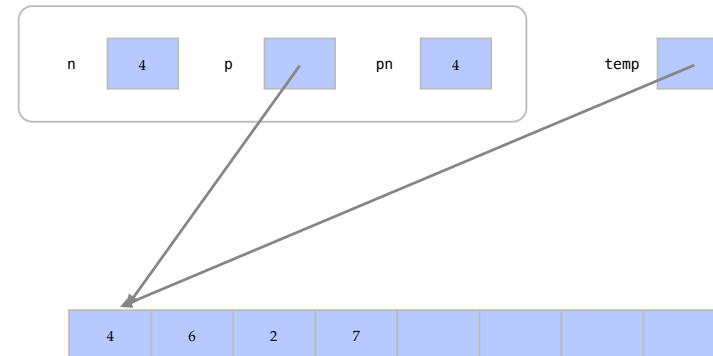
```
c.push_back(5);
```



Copiamos todos os elementos de p para temp. Este  
passo tem um custo  $O(n)$ .

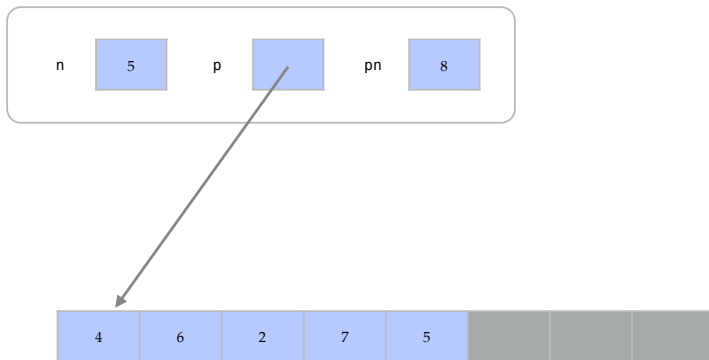
```
vector<int> c;
```

```
c.push_back(5);
```



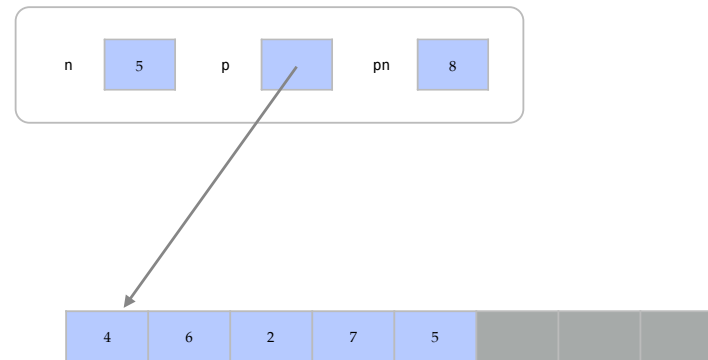
Desalocamos a memória apontada por p e fazemos  
com que p aponte para o mesmo arranjo de temp.

```
vector<int> c;          c.push_back(5);
```



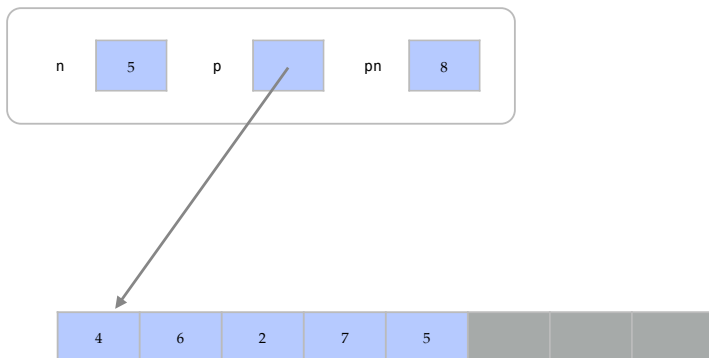
`pn` é dobrado, `n` incrementado, o valor 5 é inserido normalmente com custo  $O(1)$  e `temp` deixa de existir por ter apenas escopo de função.

```
vector<int> c;          c.push_back(5);
```



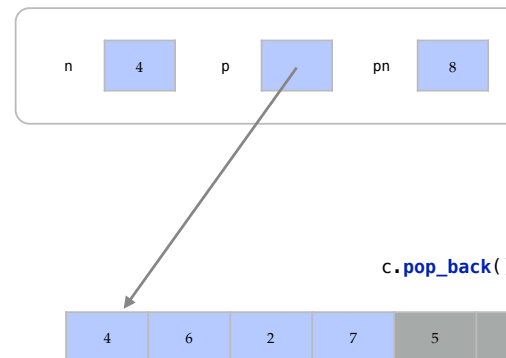
Contudo, toda a operação de inserção teve custo  $O(n)$ . Este custo, porém, se amortizado entre várias inserções tem um custo médio de  $O(1)$ .

```
vector<int> c;          c.push_back(5);
```



Existe inclusive a função `capacity` que retorna quantos elementos ainda podem ser colocados em um vector sem se alocar mais memória.

```
vector<int> c;
```



Com custo  $O(1)$ , a função para remover o último elemento do container é realizada apenas pela atualização do valor de `n`, o fazendo indicar que apenas os 4 primeiros elementos devem ser considerados.

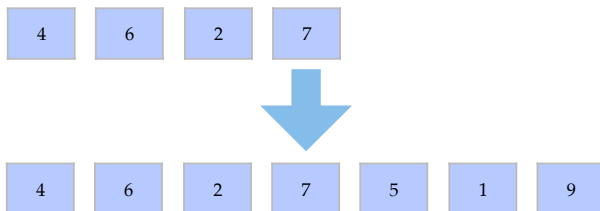
# Deque

- Double ended queue (“fila com duas pontas”)
- Indicado para sequências que crescem nas duas direções
- Inserção de elementos no início e no final da sequência é rápida
- Exige a inclusão do cabeçalho `<deque>`

Imagine a sequência de elementos abaixo. Imagine que o deque que contém esta sequência se chama `c`.

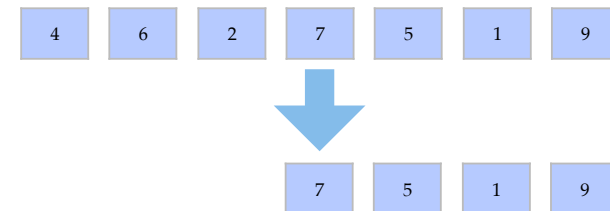


```
c.push_back(5);  
c.push_back(1);  
c.push_back(9);
```



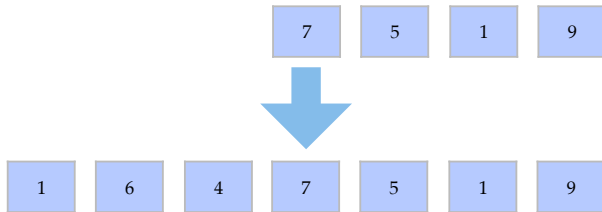
Novos elementos podem ser inseridos ao final de `c` com a função `push_back`.

```
c.pop_front();  
c.pop_front();  
c.pop_front();
```



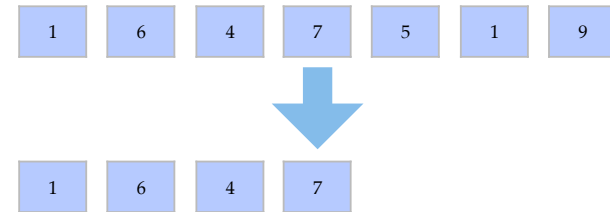
Elementos podem ser removidos do início de `c` com a função `pop_front`.

```
c.push_front(4);  
c.push_front(6);  
c.push_front(1);
```



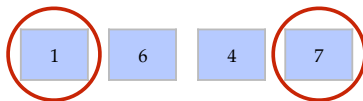
Novos elementos podem ser inseridos no início de c com a função push\_front.

```
c.pop_back();  
c.pop_back();  
c.pop_back();
```



Elementos podem ser removidos no fim de c com a função pop\_back.

```
cout << c.front() << " " << c.back() << endl;
```



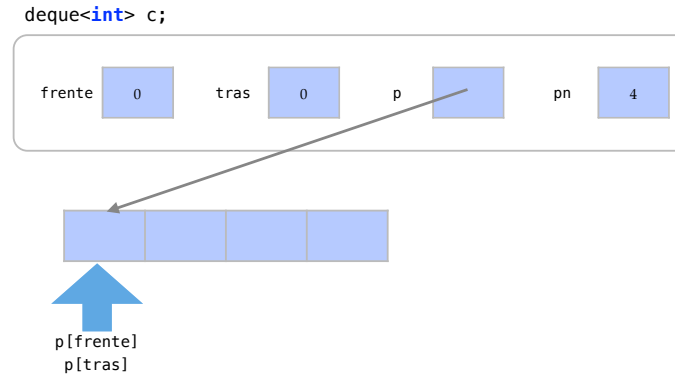
O primeiro e último elementos de c podem ser acessados com as funções front e back.

## Deque - Como funciona

- Os dequeues utilizam uma estrutura de dados um pouco mais complicada que os vectors
- Para que elementos do início sejam removidos eficientemente, dequeues utilizam uma estrutura de arranjo diferente para não precisar deslocar os elementos

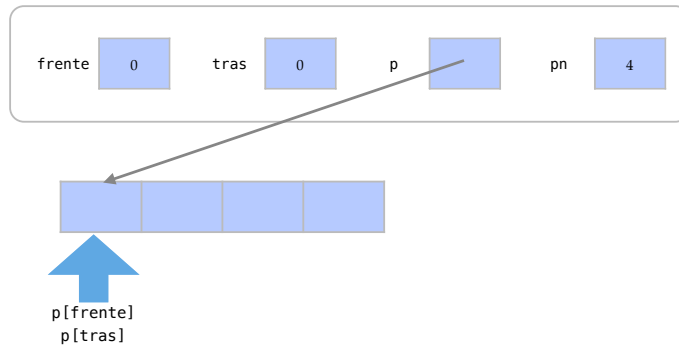
```
deque<int> c;
```

Com este comando, podemos criar um deque vazio. Internamente, o deque tem um ponteiro e três números inteiros.



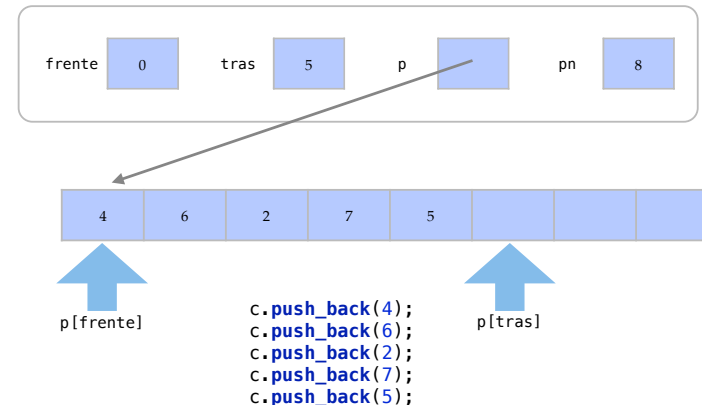
Diferentemente de vector, não temos mais o número *n* de elementos no container e sim dois elementos frente e tras.

```
deque<int> c;
```

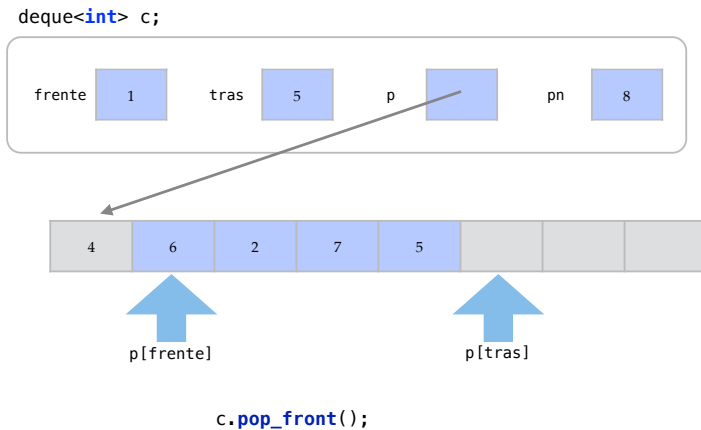


O elemento frente diz onde o primeiro elemento do container se localiza no arranjo. O elemento tras diz onde acaba o arranjo.

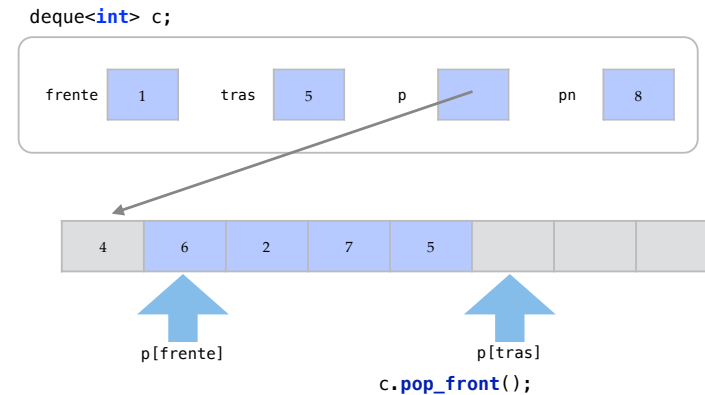
```
deque<int> c;
```



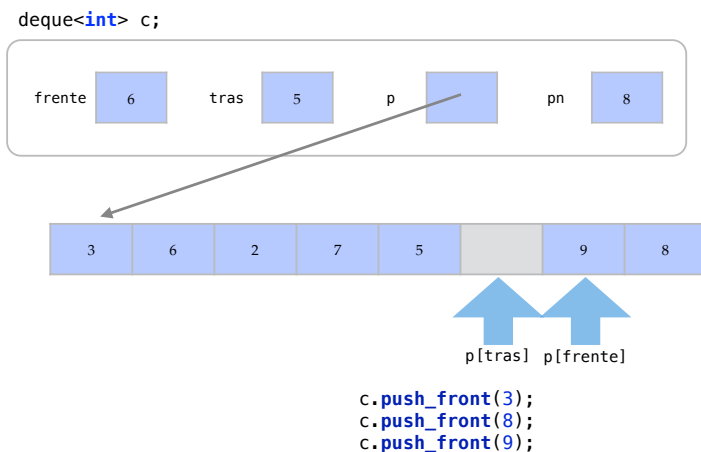
Assim, como no vector, a inserção de vários elementos causa o aumento de memória alocada no arranjo.



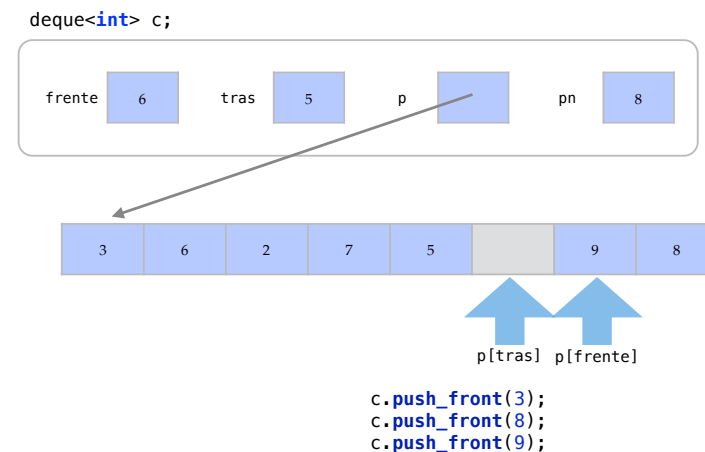
Quando um elemento do início é removido, isto é feito incrementando o valor de frente.



Os valores de frente e tras indicam que devemos considerar apenas valores entre  $p[\text{frente}]$  e  $p[\text{tras}]$  como pertencentes ao container.



Quando muitos elementos são inseridos uma estrutura toroidal é utilizada para representar os elementos do container.



Pode parecer estranho que  $\text{frente} > \text{tras}$ , mas isto indica que os elementos do container vão de  $p[6]$  até  $p[7]$  e depois continuam entre  $p[0]$  e  $p[4]$ , formando uma estrutura circular.

# List

- O container `list` representa listas duplamente encadeadas
- Assim como o `deque`, ele consegue eficientemente inserir e remover elementos das primeiras posições com uma sintaxe similar (`push_front`, `push_back`, `pop_front`, `pop_back`)
- Sua forma de representação, porém, é muito diferente
- Exige a inclusão do cabeçalho `<list>`

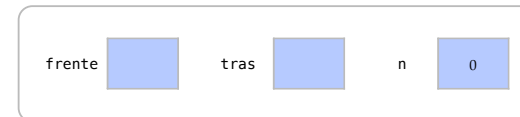
```
list<int> c;
```

Com este comando, podemos criar um `list` vazio. Internamente, o `list` tem dois ponteiros e um número inteiro.

# List - Como funciona

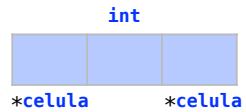
- As listas utilizam estruturas para representar células que contém ponteiros e valores
- Cada célula aponta para uma célula similar para formar a lista
- Assim, quando um novo elemento é inserido, é alocado mais espaço na memória apenas para este elemento

```
list<int> c;
```



Os elementos `frente` e `tras` são ponteiros. Enquanto isso, `n` conta o número de elementos no container.

```
list<int> c;
```



```
c.push_back(4);
```

```
list<int> c;
```



```
c.push_back(4);
```

Quando inserimos um elemento no container, uma estrutura chamada célula é criada para este elemento. Cada célula contém 2 ponteiros e um elemento.

O elemento é inserido na célula e os ponteiros não são utilizados por enquanto.

```
list<int> c;
```



```
c.push_back(4);
```

```
list<int> c;
```



```
c.push_back(5);
```

Como esta é a única célula da lista, os ponteiros frente e tras apontam para ela.

Ao se inserir mais um elemento, é criada para ele mais uma célula.



```
list<int> c;
```



```
c.push_back(5);
```

A memória alocada para esta célula está em um local qualquer da memória.

```
list<int> c;
```



```
c.push_back(5);
```

Um ponteiro da célula aponta para o último elemento da lista e o último elemento da lista aponta para a nova célula.

```
list<int> c;
```



```
c.push_back(5);
```

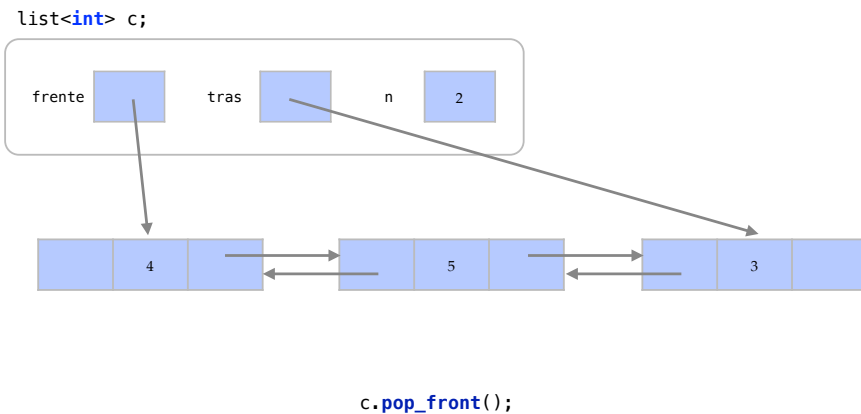
O ponteiro tras passa a apontar para a nova célula, n é incrementado e o novo elemento está inserido.

```
list<int> c;
```

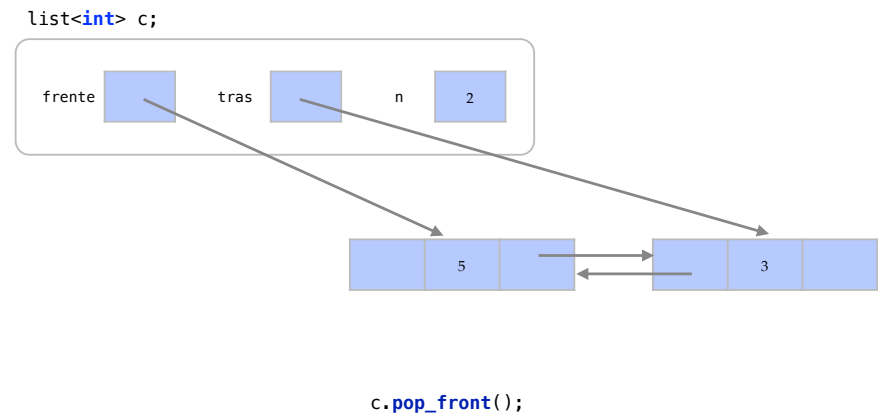


```
c.push_back(3);
```

Podemos assim inserir vários elementos com tempo  $O(1)$ .



Para remover um elemento do início ou do final, um processo muito similar é feito.



frente aponta para o próximo elemento e o anterior é removido, com suas conexões.

## Utilizando subscritos

- Assim como utilizamos os subscritos [ ] para acessar posições de arranjos, podemos utilizar subscritos para acessar elementos dos containers de sequência
- Apenas os containers de sequência chamados de containers **de acesso aleatório** tem este recurso
- O subscrito [i] é utilizado para acessar o (i+1)-ésimo elemento do container.

## Vector - Subscritos

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Este código utiliza subscritos para imprimir os valores do container c.

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Este trecho de código cria um vector chamado c e insere em seu final os elementos i = 1, 2, 3, 4 e 5

```
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Neste trecho de código, o subscrito c[i] é utilizado para imprimir os elementos de c.

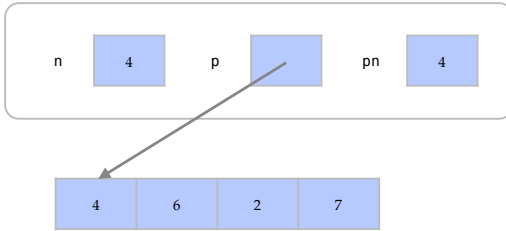
```
#include <iostream>
#include <vector>

using namespace std;

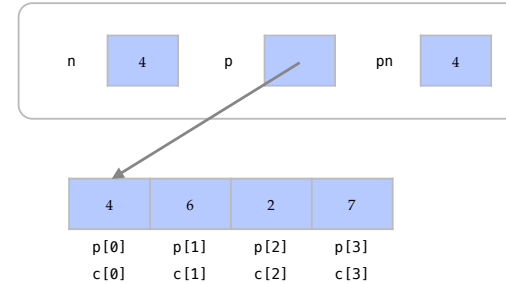
int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```



```
vector<int> c;
```



```
vector<int> c;
```



Internamente, o  $i$ -ésimo elemento do vector pode ser acessado ao se acessar o  $i$ -ésimo elemento do arranjo  $p$  que guarda seus elementos.

Assim, as posições de subscrito do arranjo apontado por  $p$  são as mesmas que devem ser retornadas pelos subscritos do container  $c$ .

## Deque - Subscritos

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> col;
    for (int i=1; i<6; i++) {
        col.push_front(i*1.1);
    }
    for (int i=0; i<col.size(); i++) {
        cout << col[i] << " ";
    }
    cout << endl;
}
```

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Este código utiliza subscritos também para imprimir os valores no deque  $c$ .

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Este bloco de código cria um deque chamado `c` e insere em seu início os elementos  $i = 1.1, 2.2, 3.3, 4.4$  e  $5.5$

```
#include <iostream>
#include <deque>

using namespace std;

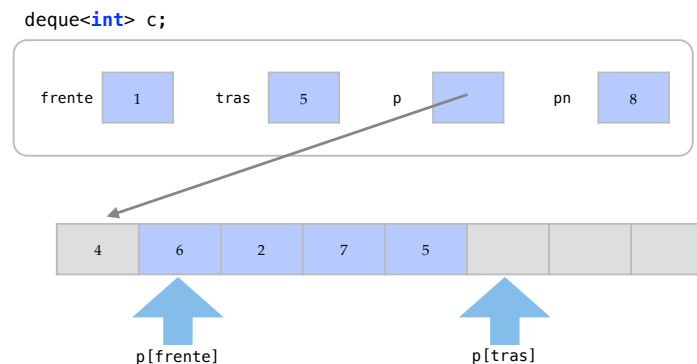
int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Neste trecho de código, o subscrito `c[i]` é utilizado para imprimir os elementos de `c`.

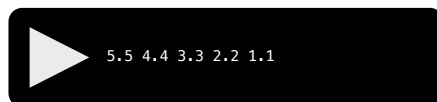
```
#include <iostream>
#include <deque>

using namespace std;

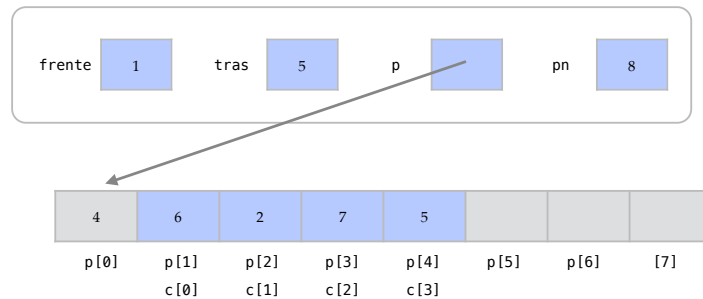
int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```



Internamente o  $i$ -ésimo elemento do deque pode ser acessado ao se acessar o  $i$ -ésimo elemento do arranjo `p` após o elemento `p[frente]`.

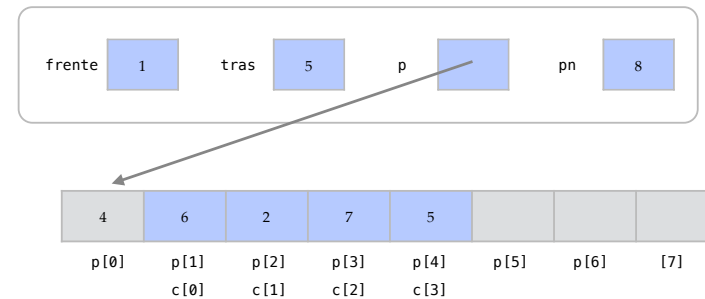


deque<int> c;



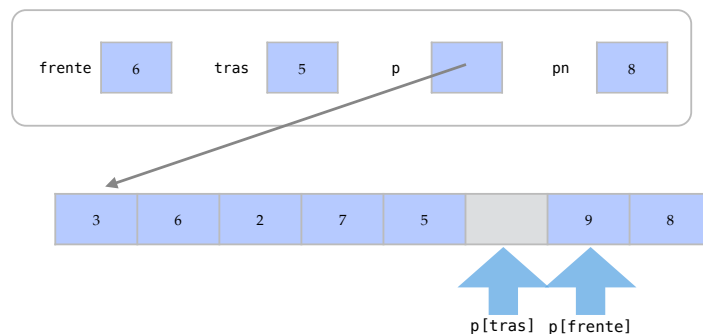
Assim, a localização das posições frente e tras devem ser consideradas ao se procurar o *i*-ésimo elemento do deque `c`.

deque<int> c;



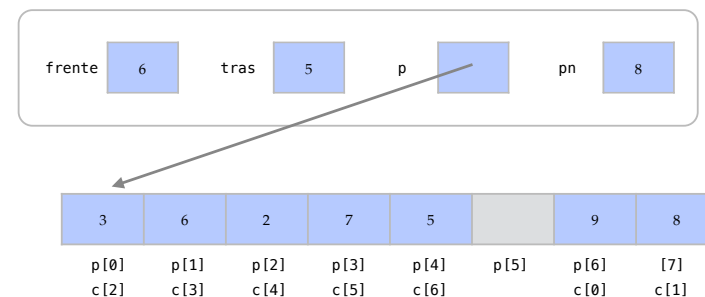
Neste caso, a posição do *i*-ésimo elemento do container pode ser encontrado na posição `p[frente+i]` do arranjo.

deque<int> c;



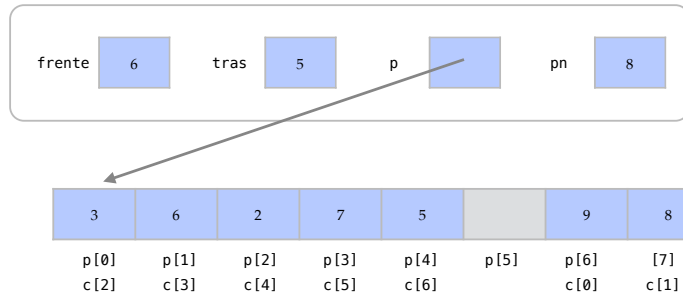
Quando ciclos ocorrem, uma operação de módulo no valor encontrado é necessária para achar a posição correta.

deque<int> c;



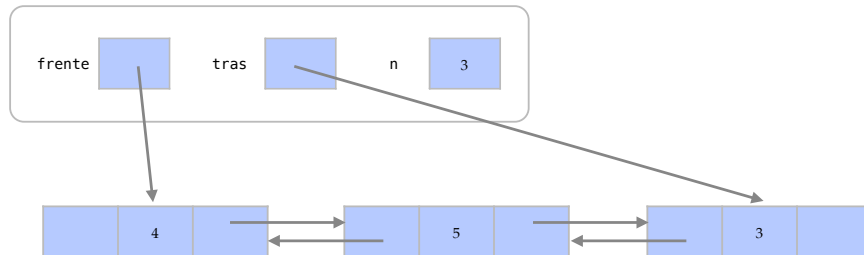
Por exemplo, veja os subscritos do arranjo e os subscritos do container neste caso.

```
deque<int> c;
```



A posição do  $i$ -ésimo elemento do container pode ser encontrado na posição  $p[(frente+i)\%pn]$  do arranjo.

```
list<int> c;
```

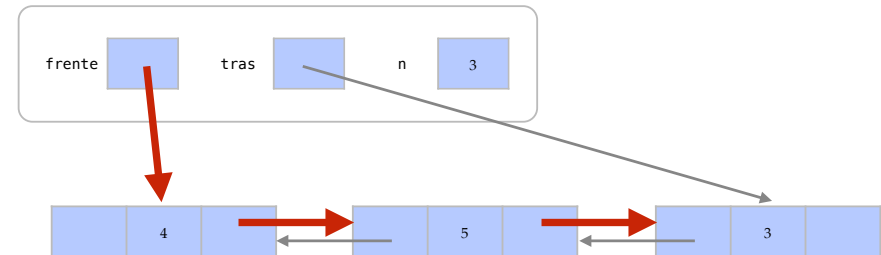


Como os elementos do `list c` estão espalhados na memória, não existe uma operação que calcule diretamente a posição do  $i$ -ésimo elemento do container na memória.

## List - Subscritos

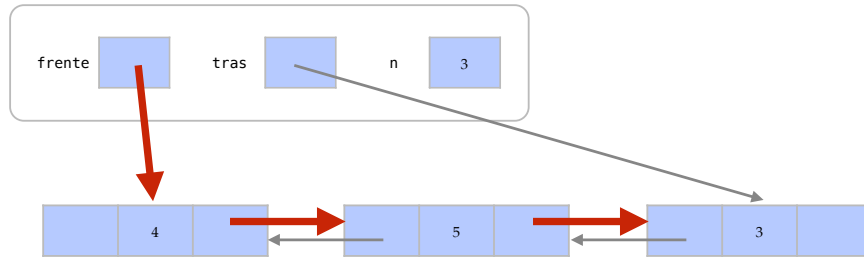
- Já para o container `list`, não é possível encontrar diretamente o  $i$ -ésimo elemento da sequência já que os elementos não são organizados em arranjos nesta estrutura
- Assim, apenas as estruturas `vector` e `deque` são de acesso aleatório por serem baseadas em arranjos.

```
list<int> c;
```



A única maneira de se encontrar o  $i$ -ésimo elemento de `c` é analisando no 1º elemento onde está o 2º, no 2º onde está o 3º, no 3º onde está o 4º, ..., no  $(i-1)$ -ésimo onde está o  $i$ -ésimo.

```
list<int> c;
```



Ou seja, precisamos seguir os ponteiros  $i$  vezes. Assim, achar o  $i$ -ésimo elemento a partir do primeiro tem um custo  $O(i)$ , ou seja,  $O(n)$  no pior caso e caso médio.

## Containers X Arranjos

- A utilização de subscritos é muito útil pois permite até que utilizemos um container de sequência para substituir arranjos
- Com isto ganhamos várias vantagens em relação a arranjos:
  - Containers já controlam seus próprios tamanhos
  - Containers contém em si mesmos algoritmos eficientes para várias tarefas

## Subscritos

- Como vimos, os subscritos são utilizados para acessar diretamente o  $i$ -ésimo elemento do container
- Este é um recurso que encontra o  $i$ -ésimo elemento do container no arranjo alocado para guardar os elementos
- Isto é possível apenas para os containers **vector** e **deque** que usam arranjos, que alocam os elementos em sequência na memória

## Iteradores

- Como vimos, a opção de se acessar elementos de um container através do operador de subscrito é restrita apenas a alguns tipos de container
- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de iteradores.



# Iteradores

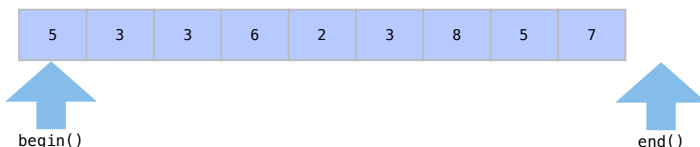
- Os iteradores são objetos que caminham (iteram) sobre elementos
  - Funcionam como um ponteiro especial para elementos de containers
  - Enquanto um ponteiro representa uma posição na memória, um iterador representa uma posição em um container
- São muito importantes pois permitem criar funções genéricas para qualquer tipo de container

# Iteradores

- Funções básicas de qualquer iterador `i` são:
  - `*i` retorna o elemento na posição do iterador
  - `++i` avança o iterador para o próximo elemento
  - `==` e `!=` confere se dois iteradores apontam para mesma posição

## Gerando iteradores

- Suponha um container chamado `c`.
- `c.begin()` retorna um iterador para o primeiro elemento deste container `c`
- `c.end()` retorna um iterador para uma posição após o último elemento do container `c`



## Iteradores - Exemplo

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

Neste exemplo usamos um iterador para percorrer os elementos de um `list`.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

O list criado se chama c e guardará elementos do tipo char.

c	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

Neste trecho de código, colocamos no container c todos os chars entre a e z

c	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

Criamos agora um iterador chamado pos. Repare que indicamos que pos é um iterador para um list de char.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

c	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

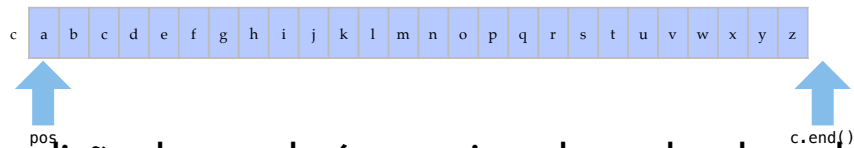
↑  
pos

Inicializamos o for fazendo com que pos seja um iterador apontando para o primeiro elemento de c.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

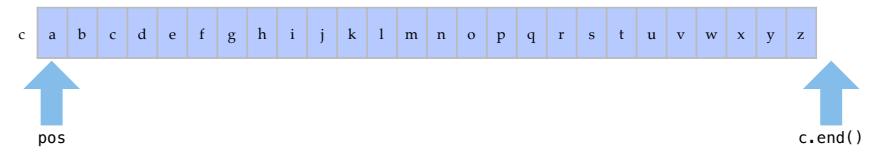


A condição de parada é que o iterador tenha chegado na posição `c.end()`, que é uma posição após o último elemento de `c`.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

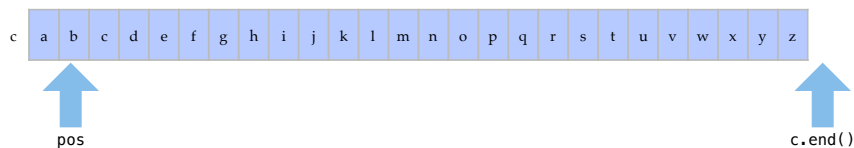


Retornamos então o elemento na posição do iterador com `*pos`.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

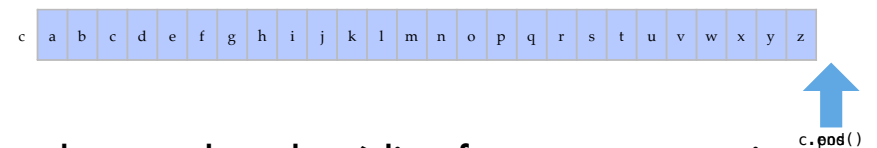


Independente do tipo de container, `++pos` faz com que `pos` aponte para seu próximo elemento.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

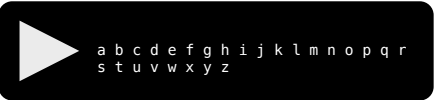


O trecho completo de código faz com que consigamos imprimir todos os elementos de `c` e `pos` finalmente chega à posição `c.end()`.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```



Repare que com iteradores, conseguimos acessar os elementos de um list, que não é um container de acesso aleatório.

# Categorias de iteradores

Unidirecionais	Podem ir apenas para frente	++					
Bidirecionais	Podem ir para frente e para trás	++			--		
De acesso aleatório	Além de ir para frente e para trás, pode-se fazer aritmética sobre eles	++	--	+	-	<	>

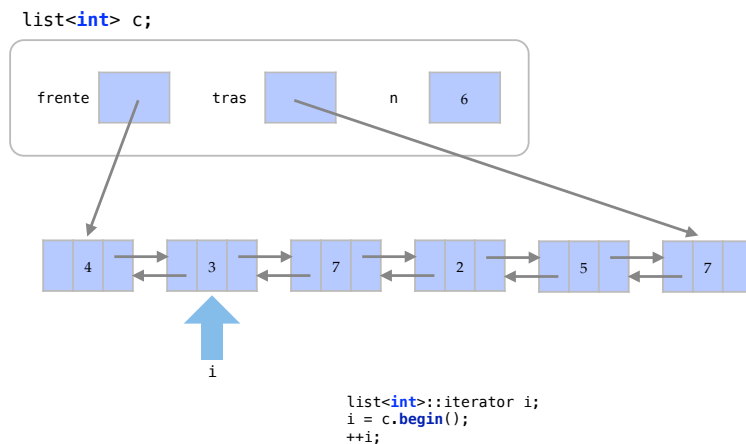
# Categorias de iteradores

- O container list contém iteradores bidirecionais pois a partir de um elemento só temos acesso ao próximo elemento ou o elemento anterior
- Para os containers vector e deque, temos iteradores de acesso aleatório pois a partir de uma posição qualquer, basta uma operação aritmética para encontrar qualquer outro elemento

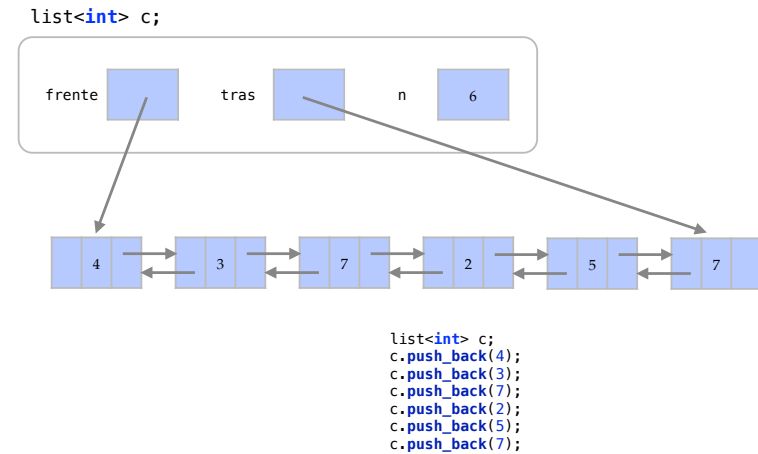
Todos os iteradores		Bidirecionais	
++p	Pré-incrementa	--p	Pré-decrementa
p++	Pós-incrementa	p--	Pós-decrementa
Entrada		Acesso aleatório	
*p	Desreferencia	p += i p -= i	Incrementa ou decrementa i posições
p = p1	Atribuição	p + i p - i	Aritmética com iteradores
p == p1	Compara igualdade de posição	p[i]	Desreferencia i posições após *p
p != p1	Compara desigualdade de posição	p < p1 p > p1 p <= p1 p >= p1	Compara posição no container

# Insert

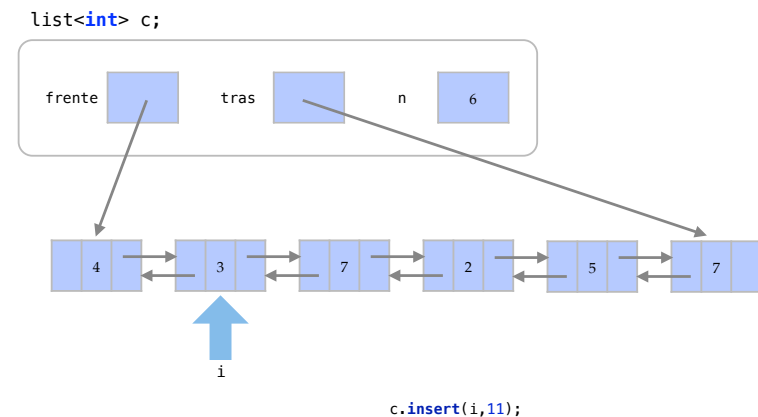
- Uma função dos containers de sequência que precisa de um iterador é a função `insert`.
- Esta função insere um elemento em uma posição qualquer da sequência.
- Para tal, a função precisa de um iterador para a posição onde queremos colocar o elemento e do elemento.



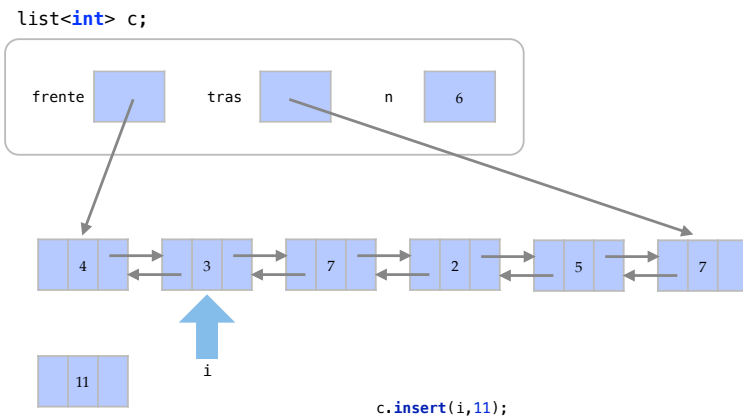
Imagine também que temos um iterador `i` apontando para a segunda posição de `c`.



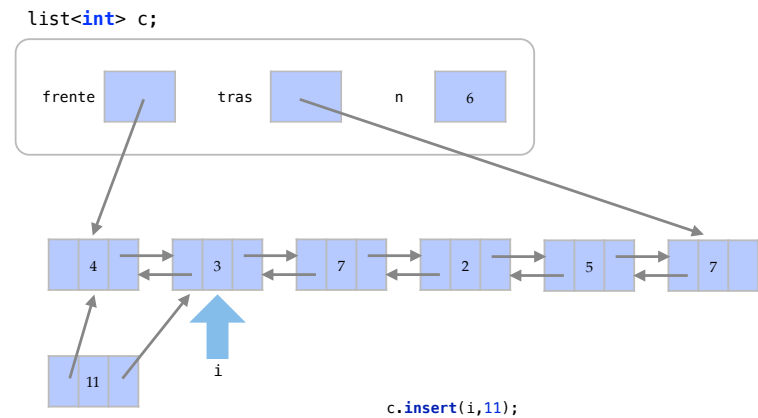
Imagine a estrutura interna de um container do tipo `list` chamado `c`.



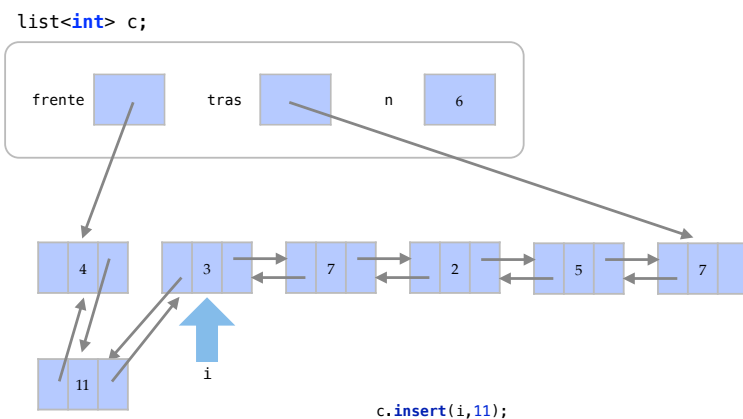
Chamamos deste modo uma função para inserir um elemento 11 na posição `i` do container.



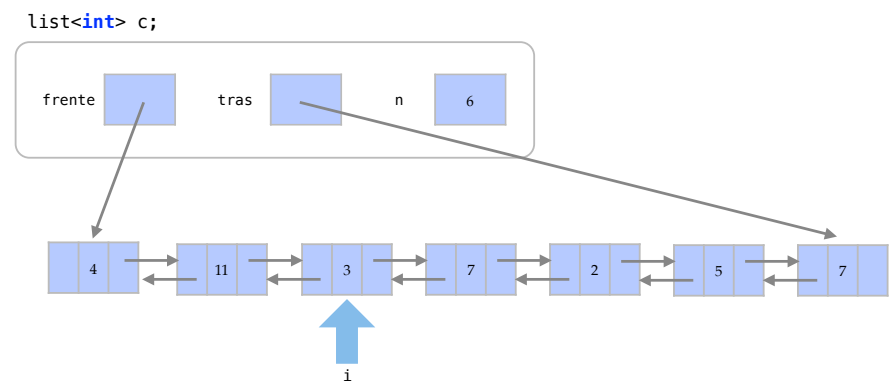
Será alocada na memória uma nova célula para este elemento 11.



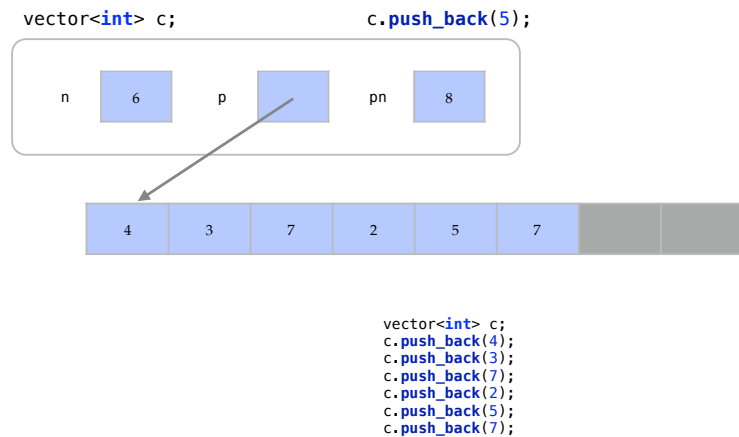
A nova célula aponta para a célula anterior à da posição *i* e para a própria célula da posição *i*.



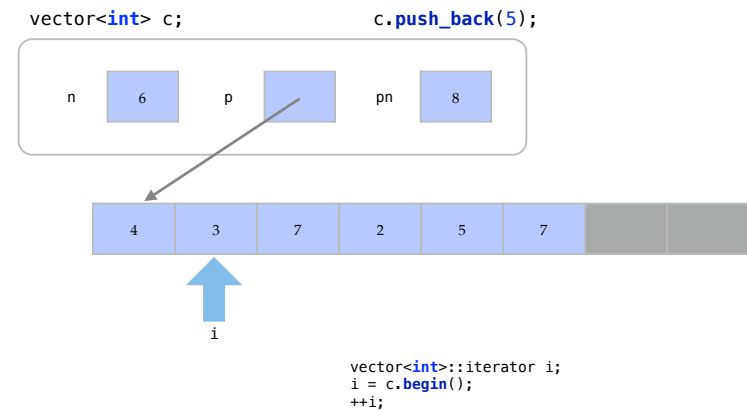
Alteramos então os ponteiros do elemento *i* e do elemento anterior a *i*.



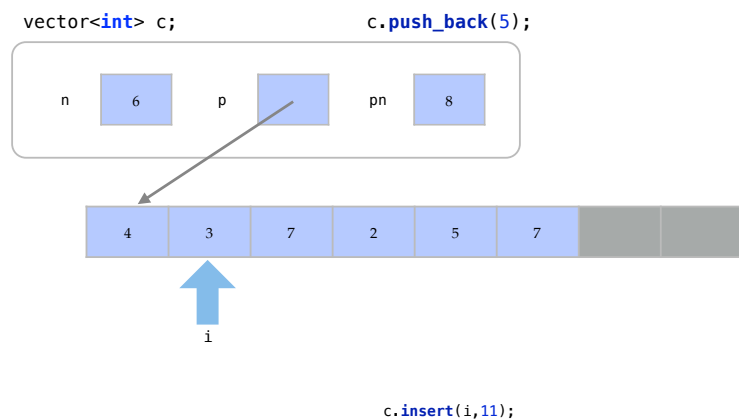
Esta é exatamente a maior vantagem do container `list`. Fazer com que o iterador chegue à posição *i*, porém, pode ter custo  $O(n)$  pois `list` não tem acesso aleatório.



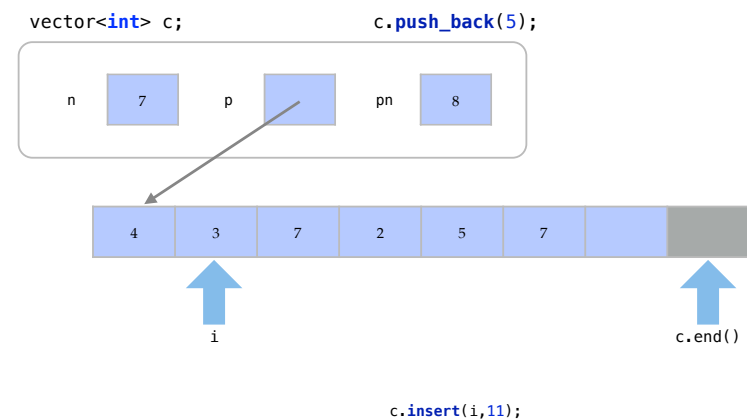
Imagine agora a estrutura interna da mesma sequência representada com um vector.



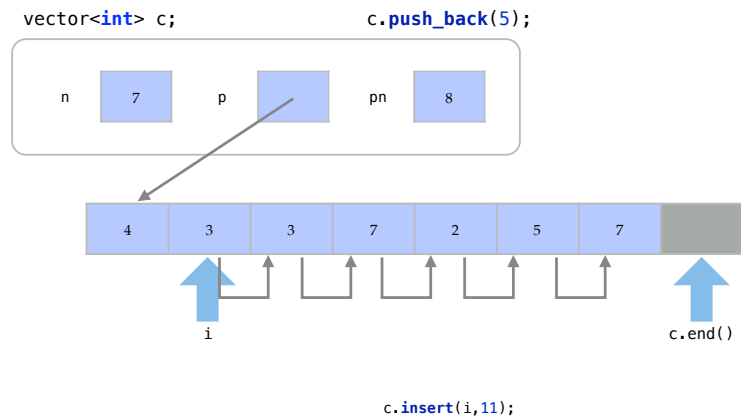
Imagine também um iterador *i* apontando para o segundo elemento da sequência.



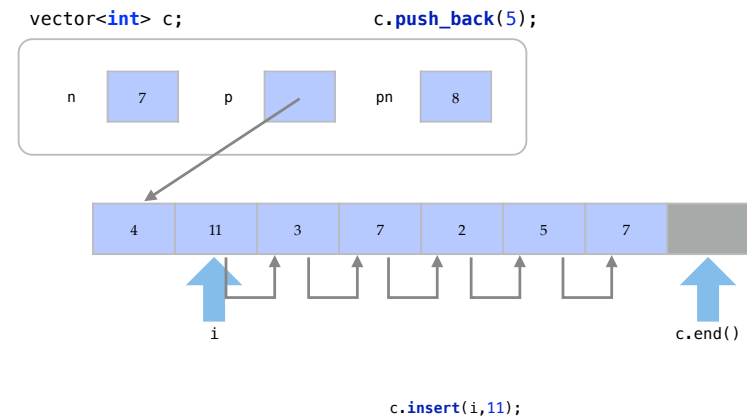
A função que insere o elemento 11 na posição *i* é mais complicada para um vector.



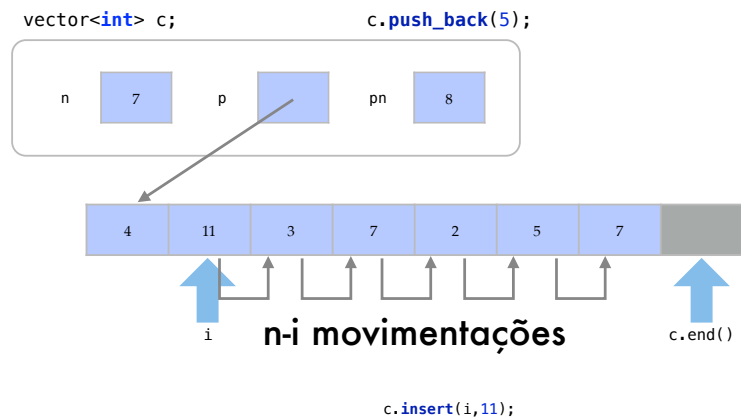
*n* é incrementado, abrindo espaço para mais um elemento. Em algumas situações, pode ser que um novo arranjo precise ser alocado.



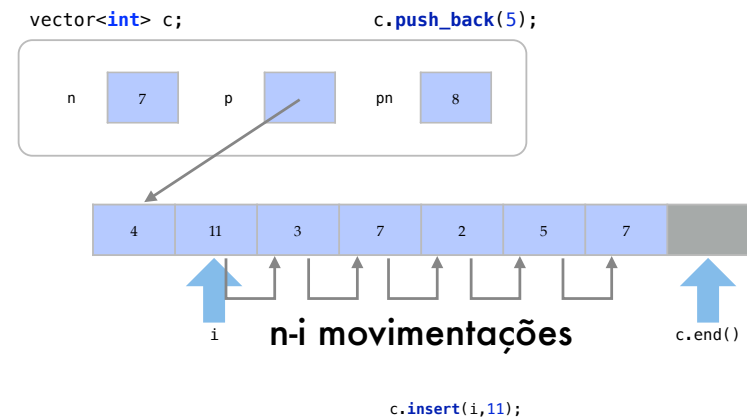
A partir de `c.end() - 2`, todos os elementos entre `i` e `c.end() - 2` precisam ser movidos para a próxima posição do arranjo.



Agora sim o elemento 11 pode ser inserido na posição `i`. Devido a todas as movimentações esta operação de inserção no meio da sequência tem um custo  $O(n)$ .



A grande vantagem dos containers `vector` e `deque` vêm do fato de eles serem baseados em arranjos, o que faz com que os elementos fiquem em sequência na memória e possibilitem acesso aleatório.



Curiosamente, a grande desvantagem destas estruturas também se devem ao fato de utilizarem arranjos pois todos os elementos precisam ser movidos para que um elemento seja inserido no meio da sequência.



## Insert - Erase

- De modo análogo à função `insert(i, elem)`, existe a função `erase(i)` que remove do container o elemento da posição `i`
- A função `insert` pode também receber 2 iteradores em vez de um elemento para inserir vários elementos de uma vez
- A função `erase` pode também receber 2 iteradores em vez de um para remover vários elementos de uma vez

## Construtor

- Com iteradores, um container pode até mesmo já ser construído com elementos de outro container de outro tipo:
  - `vector<int> c2(c.begin(), c.end());`
- Um container também pode ser construído com os elementos de um arranjo, através de seus endereços:
  - `vector<int> c(a, a+n);`

## Funções genéricas

- Vimos no exemplo anterior a função `insert`, que depende de iteradores para definir em qual posição será inserido o elemento.
- Outra utilidade de iteradores é possibilitar funções genéricas, que funcionem para qualquer tipo de container.

## Funções genéricas

```
template <typename T>
void imprime (T const& x)
{
    typename T::const_iterator pos;
    typename T::const_iterator end(x.end());
    for (pos=x.begin(); pos!=end; ++pos) {
        cout << (*pos) << " ";
    }
    cout << endl;
}
```

A função acima imprime todos os elementos de um container. Como ela utiliza templates, ela pode ser utilizada com qualquer container.

```
template <typename T>
void imprime (T const& x)
{
    typename T::const_iterator pos;
    typename T::const_iterator end(x.end());
    for (pos=x.begin(); pos!=end; ++pos) {
        cout << (*pos) << " ";
    }
    cout << endl;
}
```

Nesta função, o tipo de container será referido como T.

```
template <typename T>
void imprime (T const& x)
{
    typename T::const_iterator pos;
    typename T::const_iterator end(x.end());
    for (pos=x.begin(); pos!=end; ++pos) {
        cout << (*pos) << " ";
    }
    cout << endl;
}
```

Criamos um iterador pos que caminhará sobre os itens do container e um iterador end que é inicializado com a posição final do container chamado aqui de x.

```
template <typename T>
void imprime (T const& x)
{
    typename T::const_iterator pos;
    typename T::const_iterator end(x.end());
    for (pos=x.begin(); pos!=end; ++pos) {
        cout << (*pos) << " ";
    }
    cout << endl;
}
```

Utilizamos então o iterador como em qualquer outro código. É importante lembrar que algumas operações não estão disponíveis para todos os tipos de iteradores.

```
for (pos = c.begin(); pos != c.end(); ++pos){
    cout << *pos << " ";
}
```

É preciso tomar cuidado ao tentar criar um código genérico para qualquer tipo de container. O código acima tem capacidade de percorrer os elementos de qualquer tipo de container.

```
for (pos = c.begin(); pos != c.end(); ++pos){
    cout << *pos << " ";
}

for (pos = c.begin(); pos < c.end(); ++pos){
    cout << *pos << " ";
}
```

Já o segundo código não conseguirá percorrer elementos de qualquer tipo de container pois o operador < só está disponível para iteradores de acesso aleatório.

## Análise - Vector

- As maiores vantagens do vector são:
  - Baixo gasto de memória auxiliar para gerenciar sua estrutura
  - É possível acessar qualquer posição com custo  $O(1)$
- Sua maior desvantagem é o custo  $O(n)$  de se inserir um elemento no início ou no meio da sequência.

## Análise - Vector

- Quando acaba a memória, um vector cria um novo arranjo dinamicamente e copia os elementos. Isto tem custo  $O(n)$ .
- Inserção/remoção no fim tem custo  $O(1)$ .
- Por ter seus elementos organizados em uma arranjo (com elementos em sequência na memória), uma inserção/remoção no meio da sequência tem custo  $O(n)$  pois elementos precisam ser deslocados.
- Por outro lado, uma consulta em qualquer posição tem custo  $O(1)$  pois vectors tem acesso aleatório.

## Análise - Deque

- A estrutura do deque permite que elementos sejam inseridos com baixo custo  $O(1)$  tanto no início quanto no fim do arranjo.
- Além desta única diferença para o vector, por ser também baseado em arranjos, deque tem os mesmos resultados assintóticos do vector para:
  - Inserção/remoção no meio da sequência  $O(n)$
  - Realocação de memória  $O(n)$
  - Acesso aleatório  $O(1)$

## Análise - Deque

- Em relação ao `list`, o deque possui as mesmas vantagens e desvantagens do `vector`
  - Baixo custo de memória
  - Acesso aleatório aos elementos
  - Alto custo para inserção no meio da sequência
- Em relação ao `vector`, ele oferece a possibilidade de inserção de elementos no início da sequência em tempo  $O(1)$ .

## Deque X Vector

- Além das vantagens assintóticas do `vector`, um `deque` apresenta inserção no início em tempo  $O(1)$ , o que levaria tempo  $O(n)$  para um `vector`
- Por que é então que usáramos um `vector` em vez de um `deque`?

## Por que usamos vectors

- Nas operações de acesso a elementos onde `deque` e `vector` gastam tempo constante  $O(1)$  o tempo gasto pelo `deque` é sempre maior
- Isso porque ele precisa de operações aritméticas de soma e resto para encontrar seus elementos enquanto o `vector` pode procurar a posição diretamente no arranjo
- Assim, a utilização do `deque` só é recomendada se o recurso de inserção no início da sequência for realmente muito utilizado e importante

## Análise - List

- Vantagens:
  - Sua maior vantagem é a inserção de elementos em qualquer posição com custo constante  $O(1)$
  - Não existe realocação de arranjos
- Desvantagens:
  - Alto gasto de memória pois para cada elemento são guardados dois ponteiros
  - Não tem acesso aleatório e chegar a um elemento pode ter custo  $O(n)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

Qualquer container pode colocar elementos no fim da sequência em tempo  $O(1)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

Apenas um list pode colocar elementos no meio de uma sequência em tempo  $O(1)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

Apenas um vector não pode colocar elementos no início de uma sequência em tempo  $O(1)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

O vector possui então apenas capacidade de colocar elementos no fim da lista em tempo  $O(1)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

Um deque, apesar de ser baseado em arranjos também, pode inserir ou remover elementos no início de uma sequência em tempo  $O(1)$

## Análise

Inserção ou remoção	vector	deque	list
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$

A grande vantagem do list é poder colocar elementos em qualquer posição de uma sequência em tempo  $O(1)$

## Análise

Custo de Acesso	vector	deque	list
Início	$O(1)$	$O(1)$	$O(1)$
Meio	$O(1)$	$O(1)$	$O(n)$
Fim	$O(1)$	$O(1)$	$O(1)$

## Análise

Custo de Acesso	vector	deque	list
Início	$O(1)$	$O(1)$	$O(1)$
Meio	$O(1)$	$O(1)$	$O(n)$
Fim	$O(1)$	$O(1)$	$O(1)$

Em relação a custo de acesso, um vector ou um deque pode acessar qualquer posição tempo  $O(1)$

## Análise

Custo de Acesso	vector	deque	list
Início	$O(1)$	$O(1)$	$O(1)$
Meio	$O(1)$	$O(1)$	$O(n)$
Fim	$O(1)$	$O(1)$	$O(1)$

Já um list, tem um gasto  $O(n)$  para acessar elementos no meio da lista pois ele não tem acesso aleatório.

## Análise

Custo de Acesso	vector	deque	list
Início	$O(1)$	$O(1)$	$O(1)$
Meio	$O(1)$	$O(1)$	$O(n)$
Fim	$O(1)$	$O(1)$	$O(1)$

Assim, uma questão fundamental ao se escolher um container é saber se precisamos do recurso de acesso aleatório para elementos no meio da sequência

## Análise

Custo de Acesso	vector	deque	list
Início	$O(1)$	$O(1)$	$O(1)$
Meio	$O(1)$	$O(1)$	$O(n)$
Fim	$O(1)$	$O(1)$	$O(1)$

Se vamos usar um deque ou um vector, precisamos lembrar que apesar de ser  $O(1)$ , o custo de acesso em um vector é menor que em um deque

# Exercício

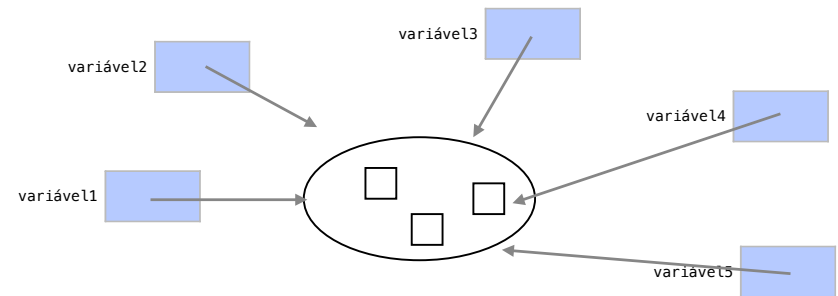
- Desenhe passo a passo como seria a representação interna das seguintes estruturas:

```
vector<int> c;  
c.push_back(4);  
c.push_back(6);  
c.push_back(2);  
c.push_back(7);  
c.push_back(5);  
c.pop_back();  
  
deque<int> c;  
c.push_back(5);  
c.push_back(5);  
c.push_front(1);  
c.push_back(9);  
c.pop_front();  
c.pop_front();  
c.pop_front();  
  
list<int> c;  
c.push_back(4);  
c.push_back(5);  
c.push_back(5);  
c.push_back(3);  
c.pop_front();  
c.pop_back();
```

Suponha que containers baseados em arranjos comecem com capacidade para 4 elementos

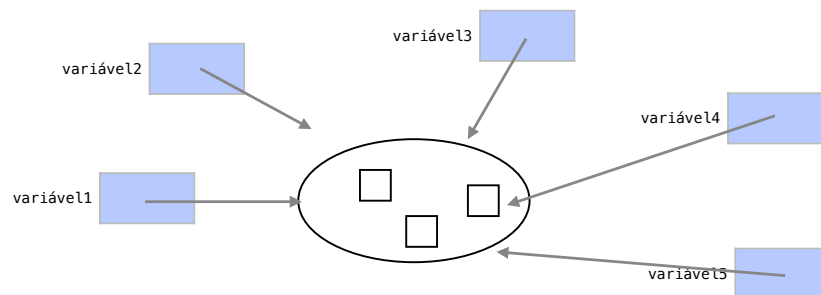
# Contêineres Associativos e Conjuntos

Os containers associativos são utilizados para representar conjuntos.



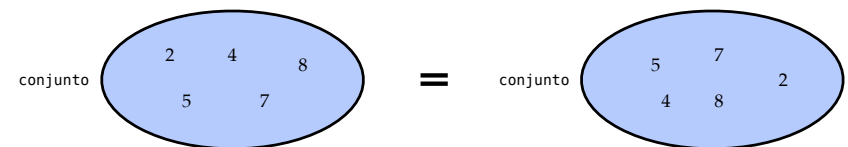
# Contêineres Associativos e Conjuntos

Em conjuntos, é importante saber se um elemento pertence ou não ao conjunto.



# Contêineres Associativos e Conjuntos

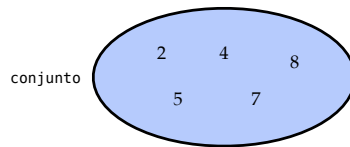
Porém, não queremos saber da posição de um elemento no conjunto. Alterar a posição dos elementos internamente, não altera o conjunto sendo representado. Esta falta de controle na relação de ordem difere estes dos containers de sequência.





## Contêineres Associativos e Conjuntos

- Como a posição dos elementos não é relevante, eles ficam organizados automaticamente
- O critério de organização é especificado em uma função de comparação



## Set

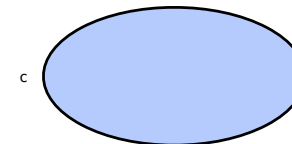
- Set é o container utilizado para representar conjuntos
- Ao acessar os elementos de set um a um, eles são ordenados de acordo com seu valor
- Cada elemento pode ocorrer apenas uma vez em um set

## Contêineres Associativos e Conjuntos

- Elementos são encontrados rapidamente pois a estrutura de dados sabe encontrá-los de acordo com sua chave
- Busca-se elementos pela própria chave e não há uma posição específica de lista para o elemento

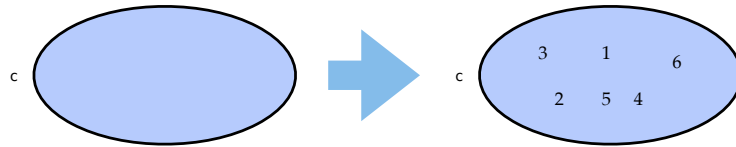
Imagine que o conjunto de números inteiros abaixo está representado por um set chamado c.

```
set<int> c;
```



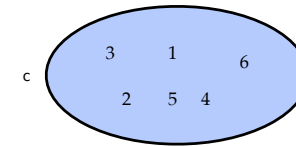
O conjunto criado está inicialmente vazio.

```
c.insert(3);
c.insert(6);
c.insert(4);
c.insert(1);
c.insert(2);
c.insert(5);
```



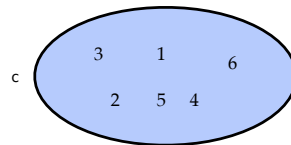
Elementos podem ser inseridos com a função `insert`.

```
set<int>::iterator i;
for (i = c.begin(); i != c.end(); ++i){
    std::cout << *i << " ";
}
cout << endl;
```

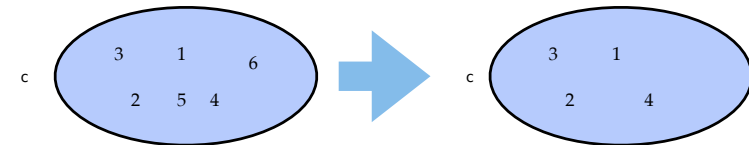


Como em qualquer container, podemos usar iteradores para acessar os elementos do container um a um.

```
set<int>::iterator i;
for (i = c.begin(); i != c.end(); ++i){
    std::cout << *i << " ";
}
cout << endl;
```



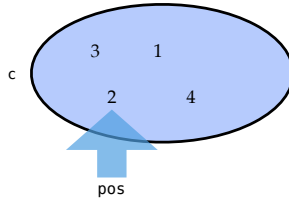
```
c.erase(5);
c.erase(6);
```



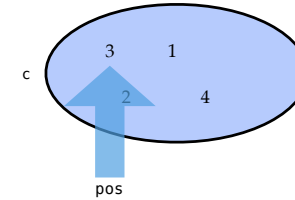
Ao fazer isto, os elementos são impressos em ordem crescente. Isto não depende da ordem em que foram inseridos.

A função `erase` pode ser utilizada para apagar elementos.

```
set<int>::iterator pos;  
pos = c.find(2);
```



```
set<int>::iterator pos;  
pos = c.find(2);  
++pos;
```



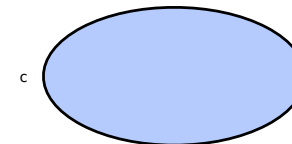
A função `find` procura um elemento e retorna um iterador para ele caso este seja encontrado.

O operador `++` leva o iterador para o próximo elemento do conjunto, considerando os elementos em ordem.

## Multiset

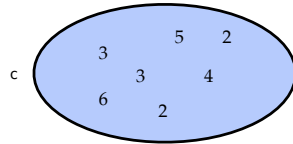
- O multiset é um container muito similar ao set
- A diferença é que ele representa multiconjuntos, ou seja, conjuntos onde um elemento pode ocorrer mais de uma vez

```
multiset<int> c;
```

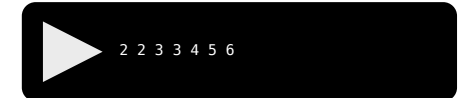


Suponha o multiset acima.

```
c.insert(3);
c.insert(3);
c.insert(6);
c.insert(4);
c.insert(2);
c.insert(2);
c.insert(5);
```



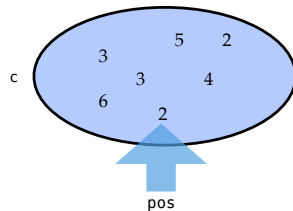
```
multiset<int>::iterator i;
for (i = c.begin(); i != c.end(); ++i){
    cout << *i << " ";
}
cout << endl;
```



Podemos agora ter elementos replicados no conjunto.

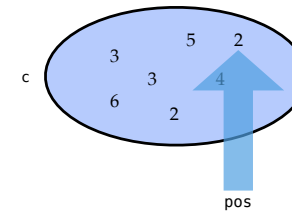
Ao iterar pelos elementos, estes ainda estão ordenados. As repetições de elementos ocorrem em sequência.

```
set<int>::iterator pos;
pos = c.find(2);
```



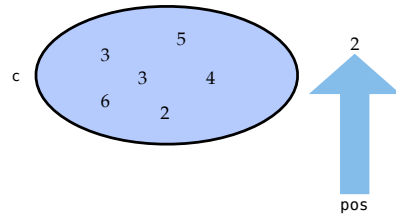
A função `find` procura a primeira ocorrência de um elemento e retorna um iterador para ele caso este seja encontrado.

```
set<int>::iterator pos;
pos = c.find(2);
++pos;
```



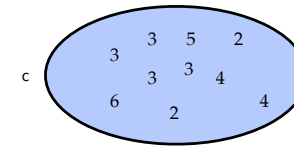
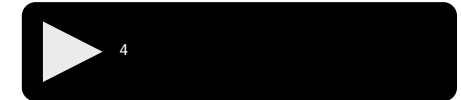
O operador `++` leva o iterador para o próximo elemento do conjunto ou para a próxima repetição do elemento, considerando os elementos em ordem.

```
c.erase(pos);
```



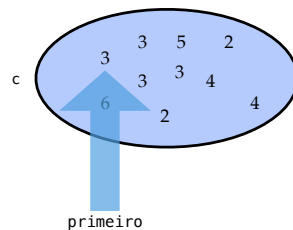
A função `erase` pode receber um iterador para retirar o elemento apontado do conjunto.

```
cout << c.count(3);
```



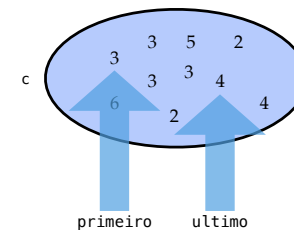
Além das funções usuais, a função `count` pode ser utilizada em um multiset para determinar quantas ocorrências de um elemento existe em um conjunto.

```
set<int>::iterator primeiro = c.lower_bound(3);
```



A função `lower_bound` retorna um iterador para a primeira ocorrência de um elemento no conjunto.

```
set<int>::iterator ultimo = c.upper_bound(3);
```

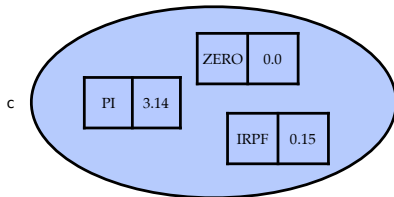


A função `upper_bound` procura a última ocorrência de um elemento no conjunto e retorna um iterador para a próxima posição.

# Map

- Map é um container que representa arranjos associativos
- Em um conjunto, nós temos várias chaves
- Em um map, temos várias chaves e para cada chave temos um registro associado
- A chave e o registro não precisam ser do mesmo tipo

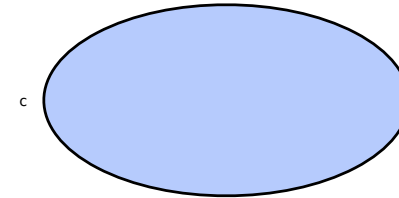
```
c["PI"] = 3.14;  
c["ZERO"] = 0.0;  
c["IRPF"] = 0.15;
```



Podemos inserir elementos facilmente neste conjunto através do operador de subscrito. Colocamos a chave no subscrito e o registro à direita.

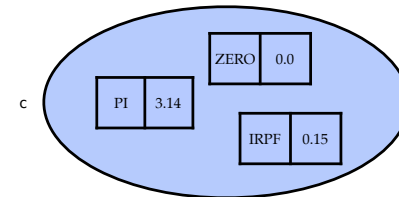
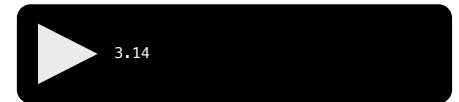
Imagine que o arranjo associativo abaixo representado por um map chamado c.

```
map<string,double> c;
```



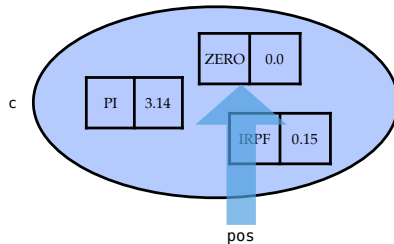
O container terá chaves do tipo `string` e registros do tipo `double`.

```
cout << c["PI"] << endl;
```



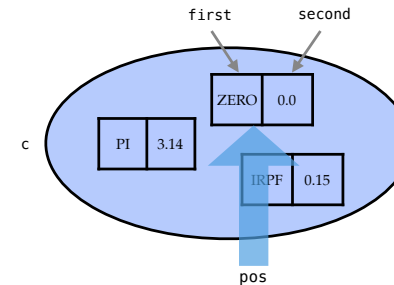
Esta estrutura é frequentemente chamada de arranjo associativo pois quando a utilizamos com o operador de subscrito é como se estivessemos acessando um arranjo onde as posições são as chaves.

```
map<string,double>::iterator pos;
pos = c.find("ZERO");
```



A função `find` é utilizada para encontrar um elemento pela chave. É retornado um iterador para este elemento.

```
map<string,double>::iterator pos;
pos = c.find("ZERO");
```

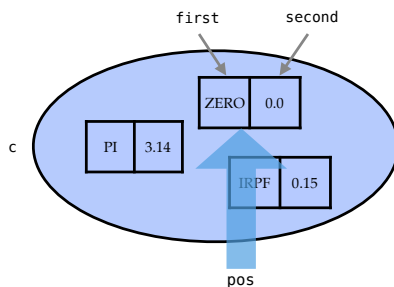
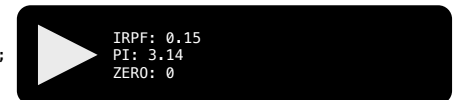


Ao desreferenciar um iterador de um map, é retornada uma estrutura com duas variáveis: `first` e `second`.

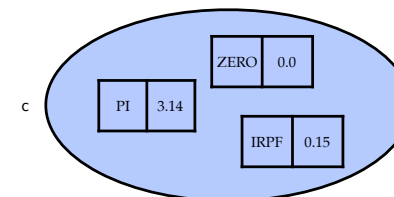
```
cout << pos->first << " : " << pos->second << endl;
ou
cout << (*pos).first << " : " << (*pos).second << endl;
```



```
for (pos = c.begin(); pos != c.end(); ++pos){
    cout << pos->first << " : " << pos->second << endl;
}
```



Podemos guardar toda a estrutura ou utilizar diretamente os campos `first` e `second`.



Assim como em outros containers, ao se iterar pelos elementos, estes ocorrem no conjunto ordenados pela chave.

# Multimap

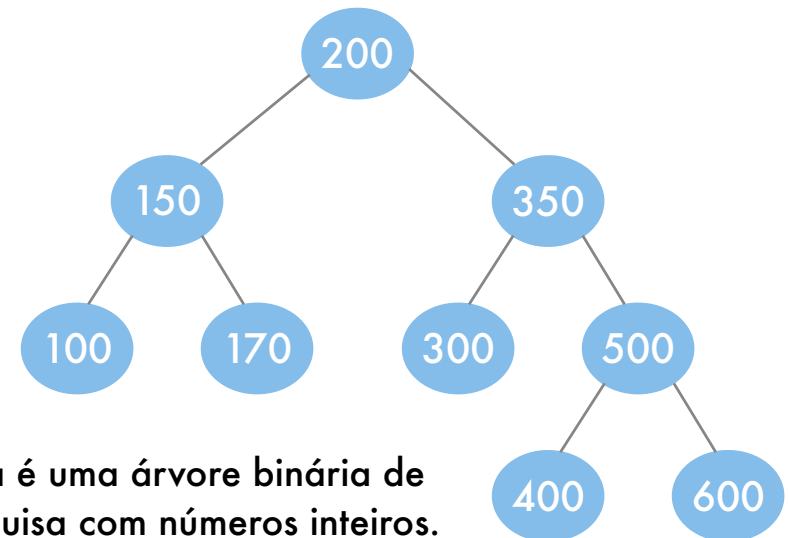
- Assim como os containers `set` e `multiset`, o container `map` também contém um equivalente que aceita entradas repetidas de um elemento
- Todas as funções de `lower_bound`, `upper_bound` e `count` podem ser utilizadas também no `multimap`
- Um recurso interessante em `multimaps` é que réplicas de uma chave podem ter registros diferentes

# Árvore Binária de Pesquisa

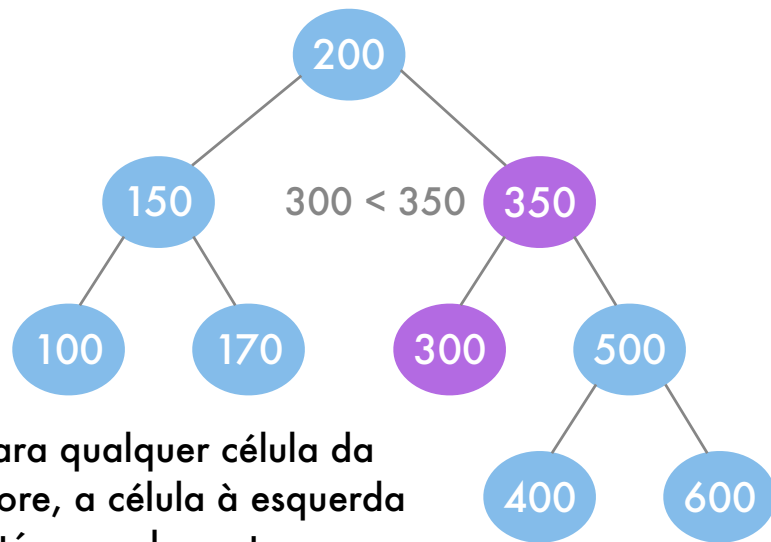
- Estruturas eficientes para armazenar informação
- Acesso direto a elementos eficiente
- Facilidade de inserção e remoção
- Células à esquerda contêm elementos menores e células à direita contêm elementos maiores

## Contêineres Associativos Como funcionam

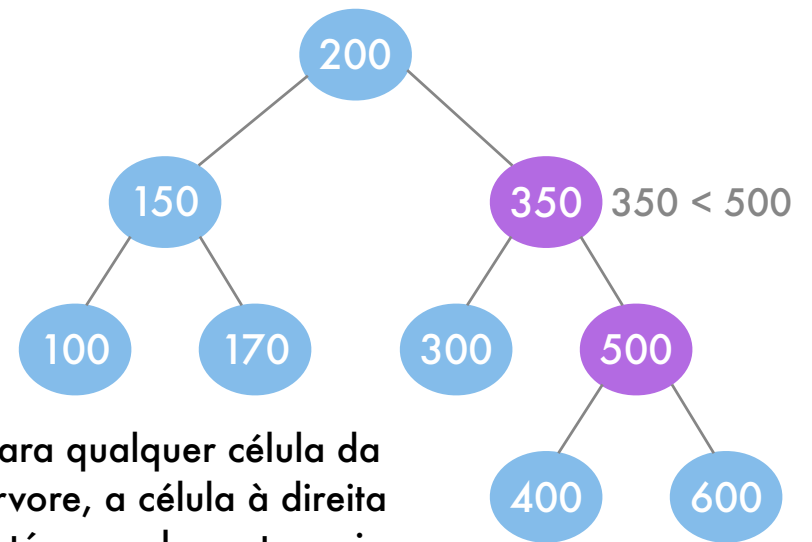
- Apesar de recursos diferentes entre os tipos de containers associativos apresentados, todos eles se baseiam em uma estratégia similar:
- Árvores Binárias de Pesquisa ou BST (*Binary Search Tree*)



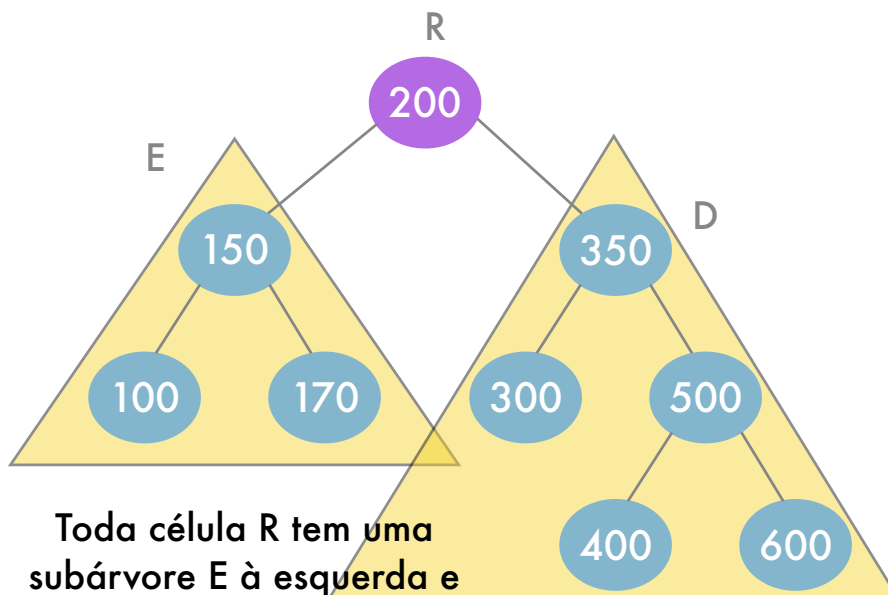




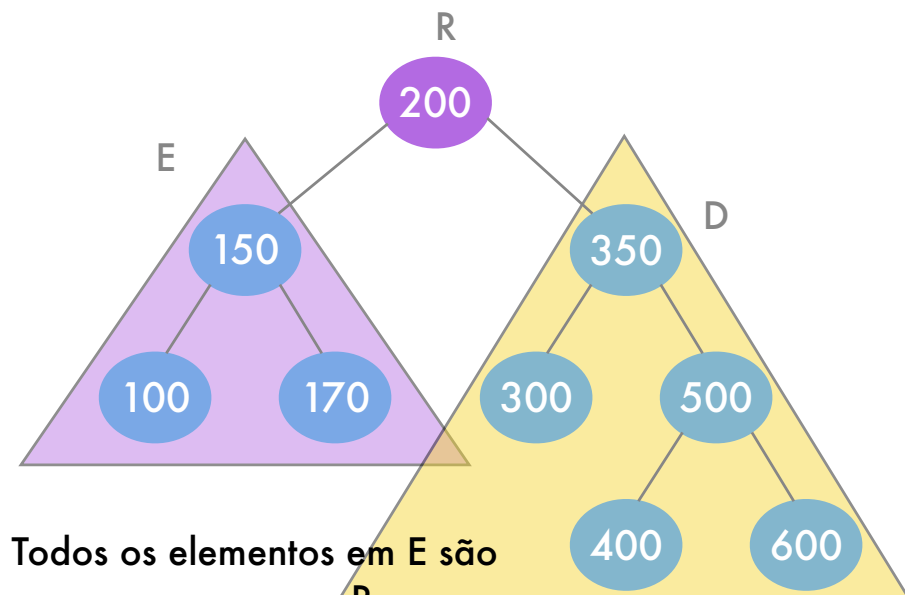
Para qualquer célula da árvore, a célula à esquerda contém um elemento menor.



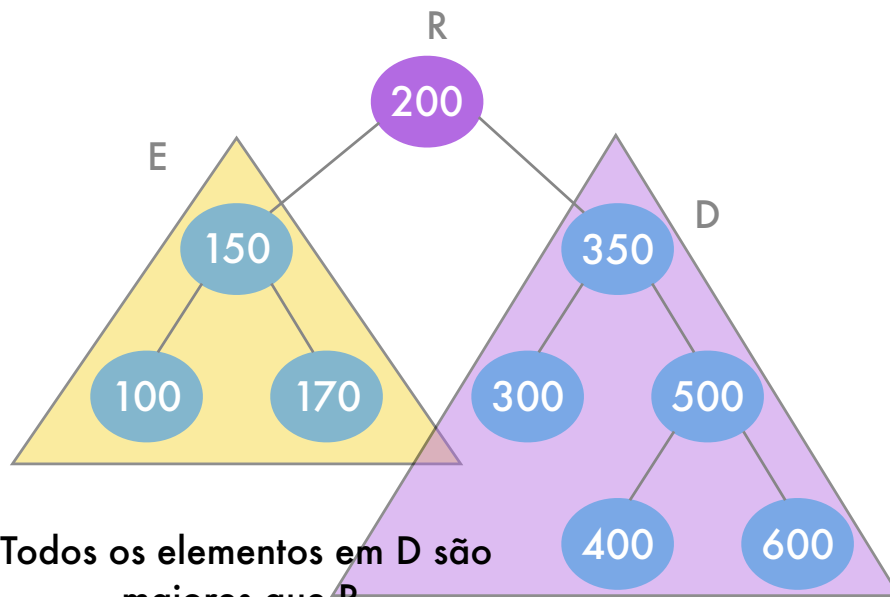
Para qualquer célula da árvore, a célula à direita contém um elemento maior.



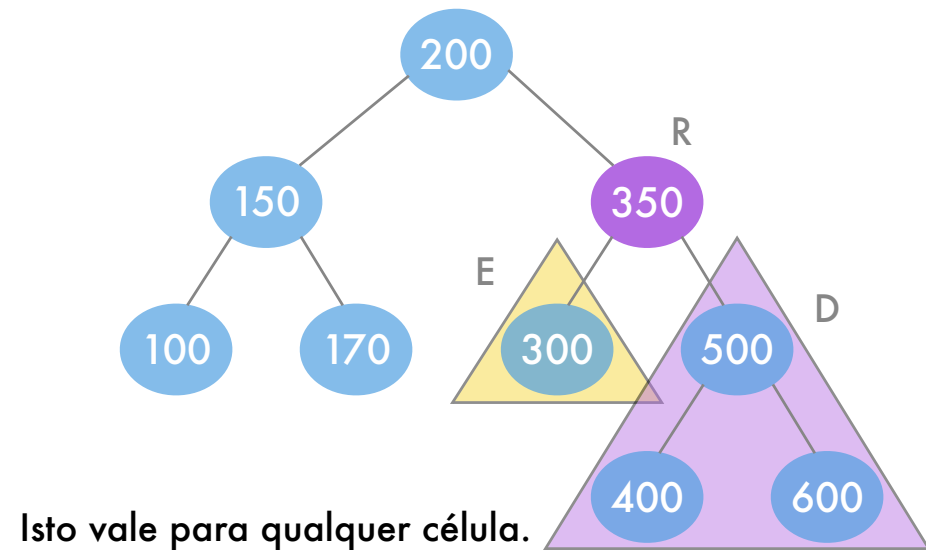
Toda célula R tem uma subárvore E à esquerda e outra subárvore D à direita.



Todos os elementos em E são menores que R.



Em um container C++, a árvore é representada através da alocação de memória para vários structs que representam células.



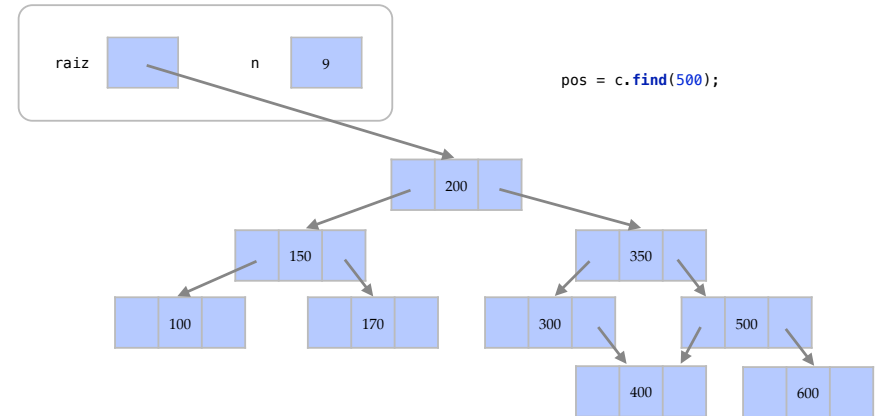
De modo similar às células utilizadas nos containers list, cada célula de uma árvore contém um elemento e dois ponteiros para outras células.

```
set<int> c;
```



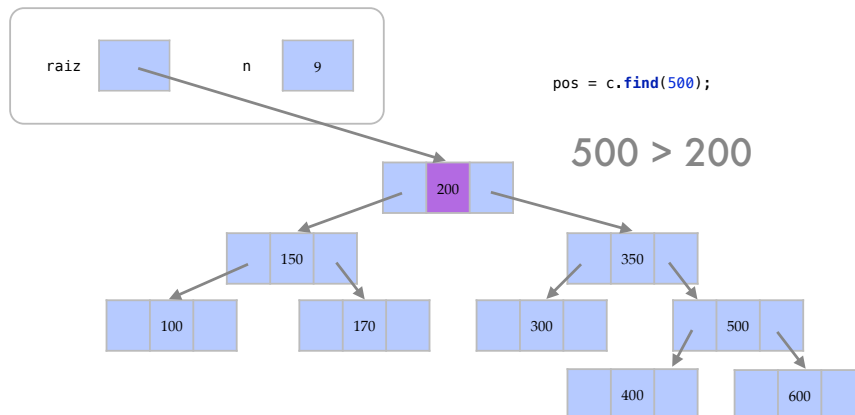
O objeto do tipo set contém um ponteiro para a célula alocada como raiz da árvore e uma variável para contar o número de elementos na estrutura.

```
set<int> c;
```



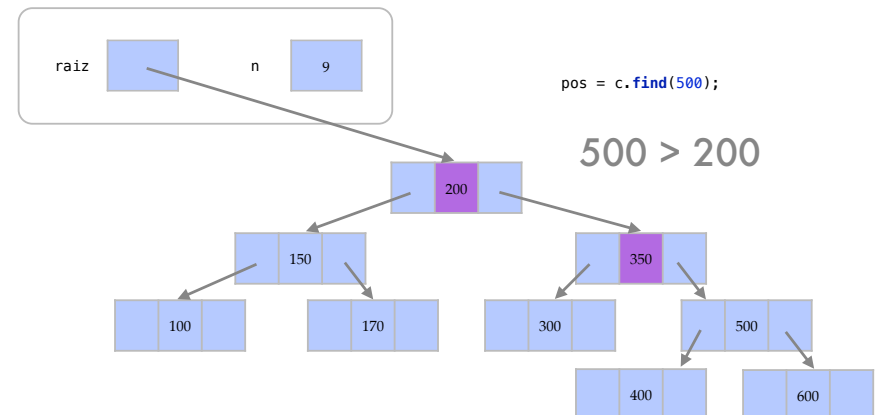
Quando os elementos são inseridos, os ponteiros das células apontam para os elementos raiz das árvores à esquerda e à direita.

```
set<int> c;
```



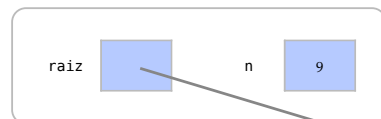
Para procurar um elemento, basta ir à raiz da árvore e fazer uma comparação para saber em qual subárvore está o elemento.

```
set<int> c;
```

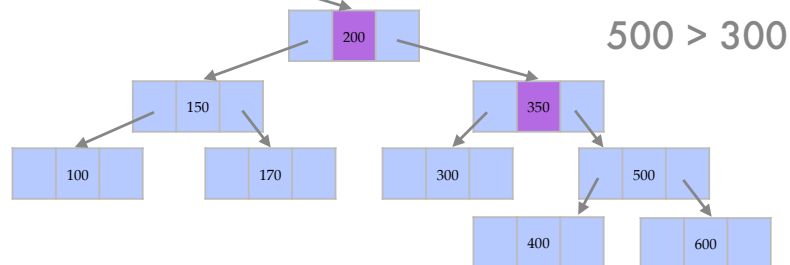


Ao descobrirmos em qual subárvore está o elemento, vamos a esta subárvore e repetimos o processo em sua raiz.

```
set<int> c;
```



```
pos = c.find(500);
```

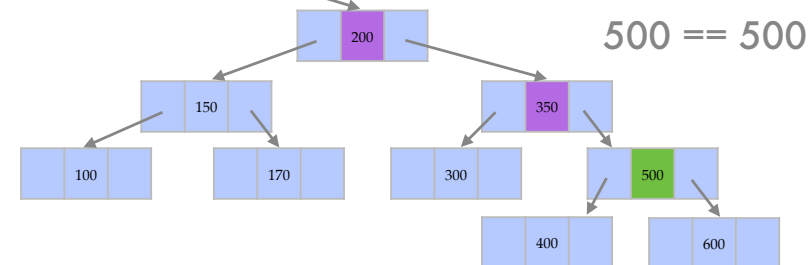


Ao descobrirmos em qual subárvore está o elemento, vamos a esta subárvore e repetimos o processo em sua raiz.

```
set<int> c;
```



```
pos = c.find(500);
```

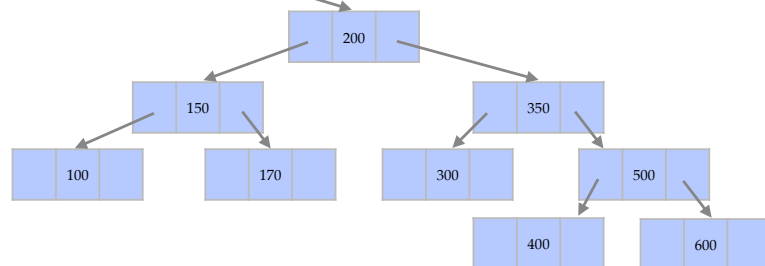


Se a raiz da árvore é o elemento que procuramos, terminamos a busca. Se tal raiz não é encontrada, o elemento não está no conjunto.

```
set<int> c;
```

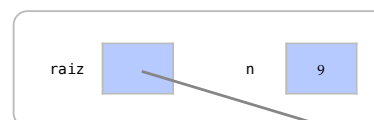


```
c.insert(180);
```

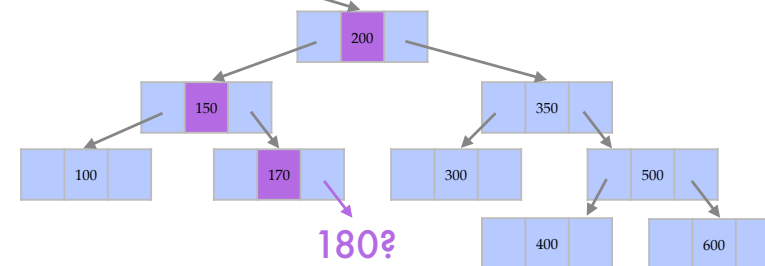


Para inserir um elemento, um processo similar é utilizado. Suponha que queremos inserir o elemento 180.

```
set<int> c;
```

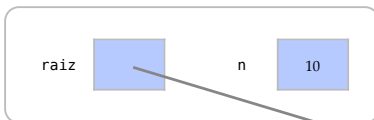


```
c.insert(180);
```

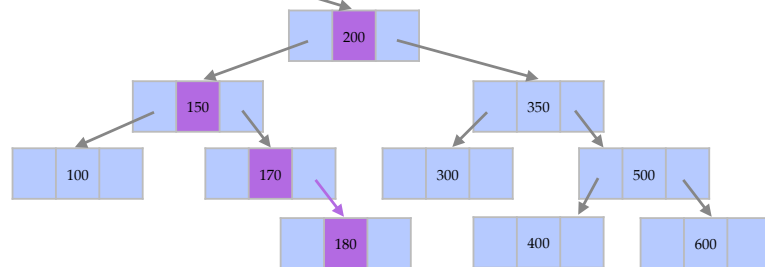


Percorremos as células da árvore até encontrar onde o elemento 180 deveria estar...

```
set<int> c;
```

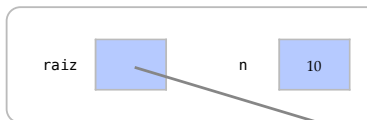


```
c.insert(180);
```

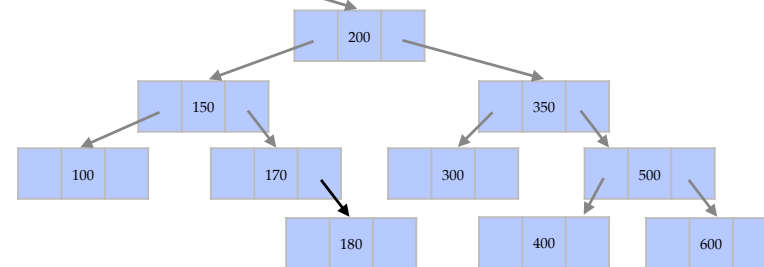


Agora basta criar uma célula para o elemento e inserir nesta posição.

```
set<int> c;
```

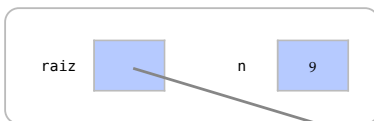


```
c.erase(180);
```

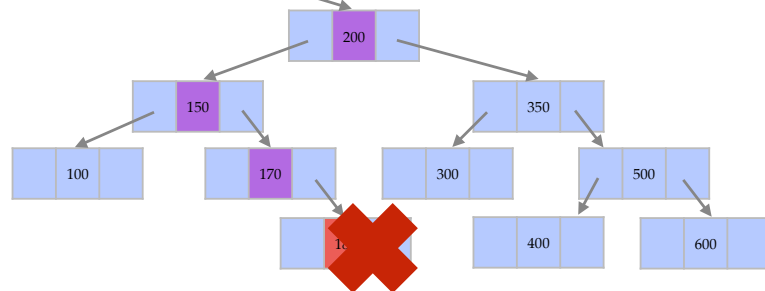


Para remover um elemento, temos um problema um pouco mais complicado, que pode ocorrer em 3 casos.

```
set<int> c;
```

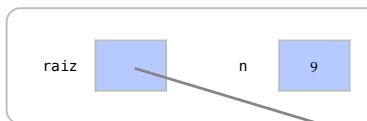


```
c.erase(180);
```

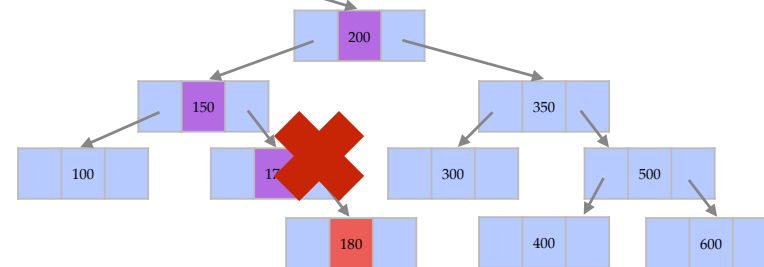


Caso 1) Para remover um nó folha, apenas procuramos por ele e o removemos.

```
set<int> c;
```

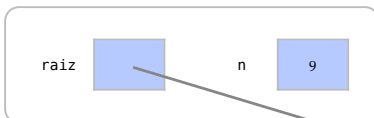


```
c.erase(170);
```

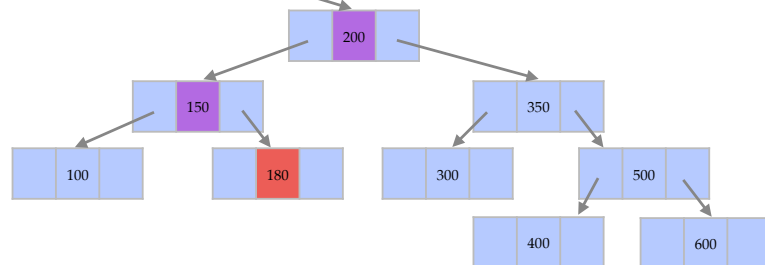


Caso 2) Para remover um nó que tem apenas um filho: primeiro removemos o nó

```
set<int> c;
```



`c.erase(170);`

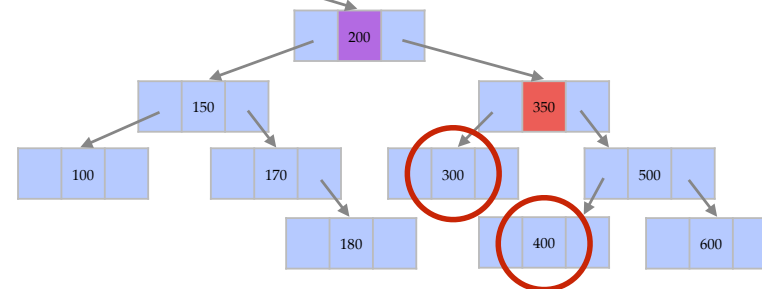


Caso 2) Em seguida substituímos o nó por seu filho.

```
set<int> c;
```

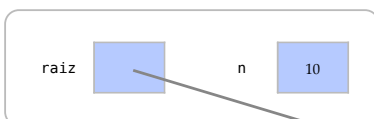


`c.erase(350);`

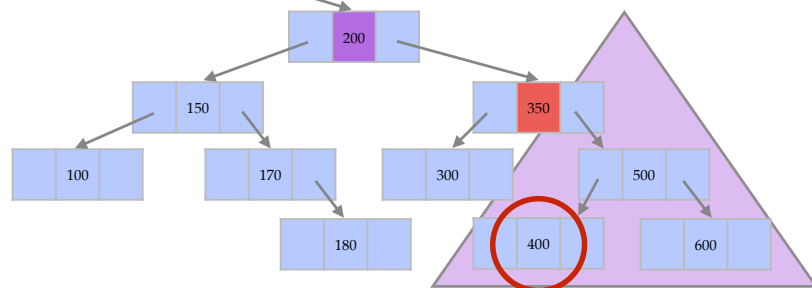


Caso 3) Para remover um nó com 2 filhos: em vez de remover o nó, procuramos seu sucessor ou predecessor de acordo com seu valor

```
set<int> c;
```



`c.erase(350);`

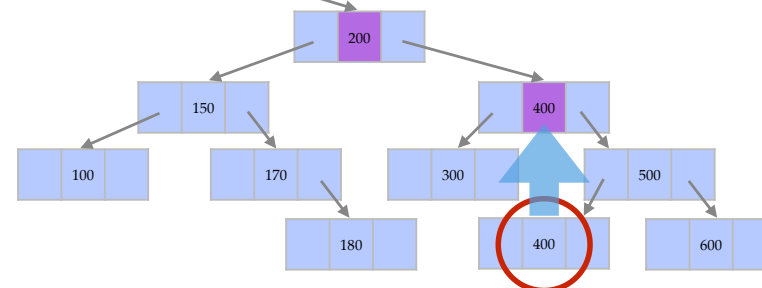


Caso 3) Neste caso, vamos utilizar o sucessor de 350, que é o elemento mais à esquerda da árvore à direita de 350

```
set<int> c;
```

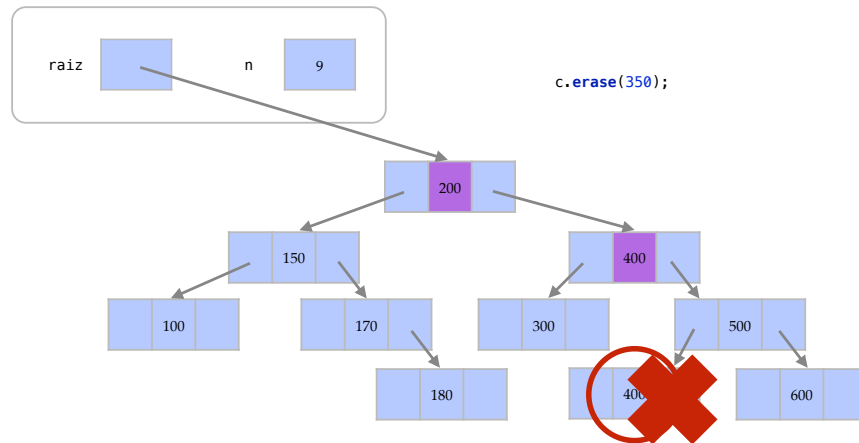


`c.erase(350);`



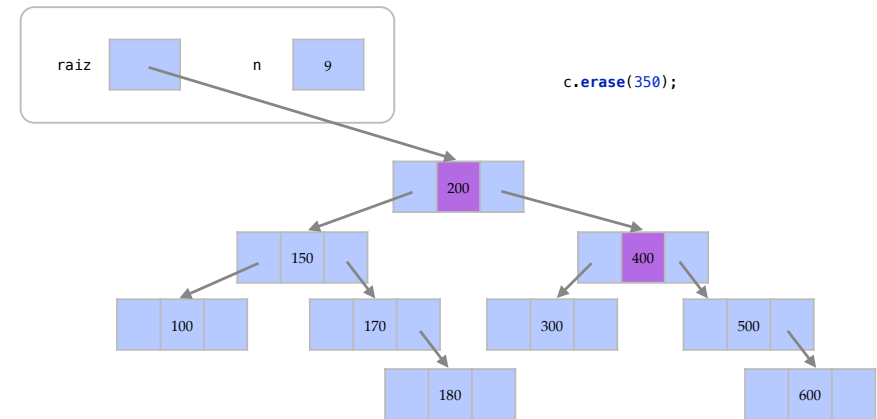
Caso 3) O valor do sucessor é copiado em cima do item que queremos remover.

```
set<int> c;
```



Caso 3) Removemos então o sucessor marcado.

```
set<int> c;
```

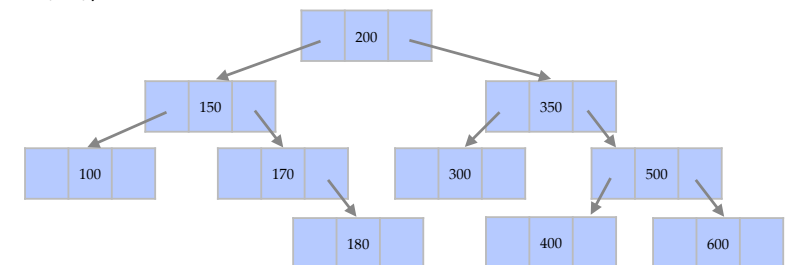


Caso 3) A remoção do sucessor, neste caso, caiu no caso 1. O processo é assim encerrado.

## Balanceamento

- Apesar da eficiência das árvores apresentadas, a ordem de inserção dos elementos pode afetar o desempenho das árvores
- Os elementos podem ser inseridos em uma ordem tal que a árvore final tenha muitos níveis e seja pouco eficiente.

```
c.insert(200);  
c.insert(150);  
c.insert(350);  
c.insert(100);  
c.insert(170);  
c.insert(300);  
c.insert(500);  
c.insert(180);  
c.insert(400);  
c.insert(600);
```

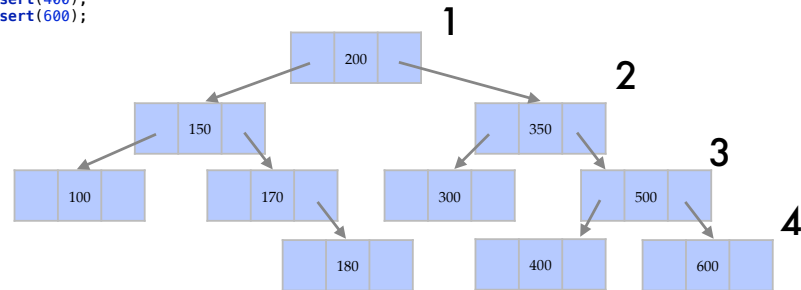


Suponha esta árvore gerada internamente pela inserção dos elementos na ordem apresentada acima.

```

c.insert(200);
c.insert(150);
c.insert(350);
c.insert(100);
c.insert(170);
c.insert(300);
c.insert(500);
c.insert(180);
c.insert(400);
c.insert(600);

```

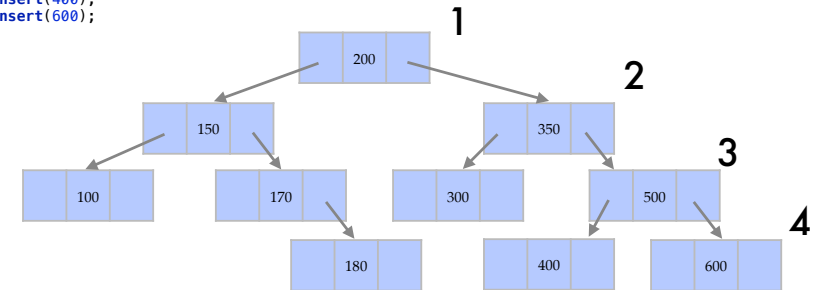


Para encontrar um elemento nesta árvore, no máximo 4 comparações são feitas. Uma em cada nível.

```

c.insert(200);
c.insert(150);
c.insert(350);
c.insert(100);
c.insert(170);
c.insert(300);
c.insert(500);
c.insert(180);
c.insert(400);
c.insert(600);

```

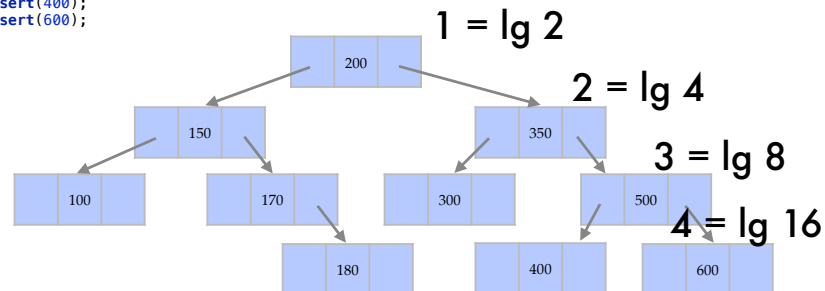


Esta é uma árvore completamente balanceada, onde o número de níveis é o mínimo possível para a quantidade de elementos.

```

c.insert(200);
c.insert(150);
c.insert(350);
c.insert(100);
c.insert(170);
c.insert(300);
c.insert(500);
c.insert(180);
c.insert(400);
c.insert(600);

```

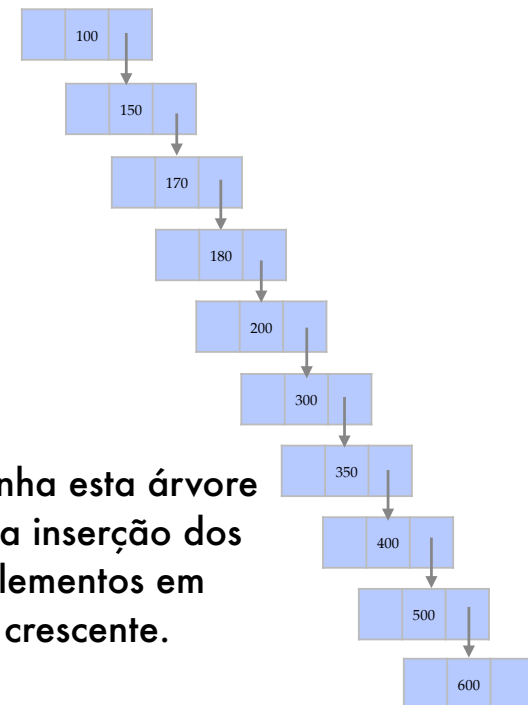


O número de níveis em uma árvore balanceada com  $n$  elementos é  $O(\log n)$ . Sendo o custo de se encontrar um elemento igual ao número de níveis, este custo também é  $O(\log n)$ .

```

c.insert(100);
c.insert(150);
c.insert(170);
c.insert(180);
c.insert(200);
c.insert(300);
c.insert(350);
c.insert(400);
c.insert(500);
c.insert(600);

```



Agora suponha esta árvore gerada pela inserção dos mesmo elementos em ordem crescente.



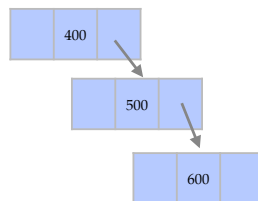
```

c.insert(100);
c.insert(150);
c.insert(170);
c.insert(180);
c.insert(200);
c.insert(300);
c.insert(350);
c.insert(400);
c.insert(500);
c.insert(600);

```



Encontrar um elemento na árvore pode custar até 10 comparações.

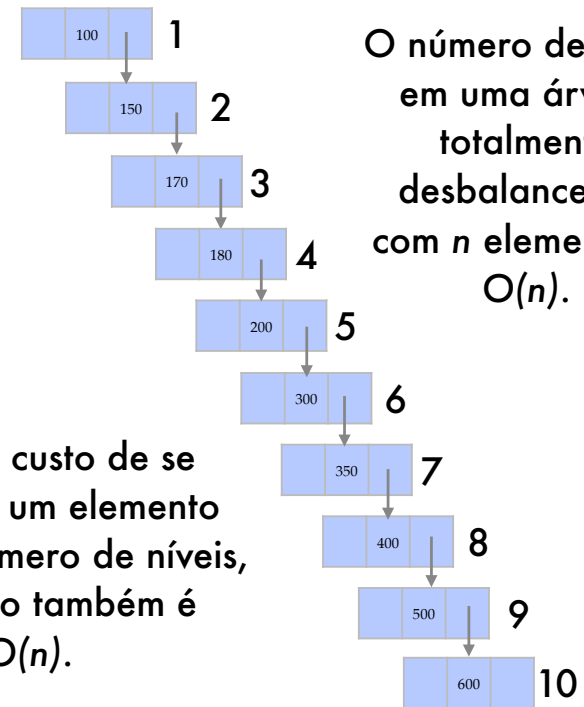


Para que este problema não ocorra, a STL prevê estratégias de balanceamento para que o custo das operações sobre conjunto seja sempre  $O(\log n)$ .

```

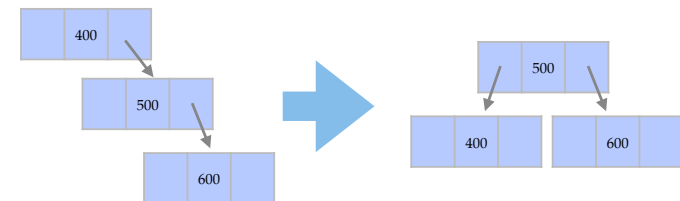
c.insert(100);
c.insert(150);
c.insert(170);
c.insert(180);
c.insert(200);
c.insert(300);
c.insert(350);
c.insert(400);
c.insert(500);
c.insert(600);

```



O número de níveis em uma árvore totalmente desbalanceada com  $n$  elementos é  $O(n)$ .

Sendo o custo de se encontrar um elemento igual ao número de níveis, este custo também é  $O(n)$ .



Estas estratégias usualmente envolvem rotações nos elementos inseridos, porém, variam muito de acordo com o compilador.

## Containers Associativos - Análise

- Em uma árvore binária, cada comparação feita pelo algoritmo que busca um elemento divide as soluções possíveis pela metade, de modo similar a uma busca binária.
- Assim, o custo inserção, remoção e pesquisa em todos os casos é  $O(\log n)$ .

## Containers Associativos - Análise

- Como a STL contém estratégias de balanceamento, o custo das operações continua  $O(\log n)$  mesmo no pior caso.
- Na verdade, mesmo para árvores que não tem estratégias de balanceamento, o custo médio ainda é  $O(\log n)$  para inserções feitas aleatoriamente.
- Seu pior caso, porém, é  $O(n)$ .

## Contêineres Associativos Desordenados

- Até agora, foram apresentados os seguintes containers associativos:
  - `set`
  - `multiset`
  - `map`
  - `multimap`

## Contêineres Associativos Desordenados

- Apresentaremos agora seus seguintes containers equivalentes desordenados:
  - `unordered_set`
  - `unordered_multiset`
  - `unordered_map`
  - `unordered_multimap`

# Contêineres Associativos Desordenados

- A utilização dos containers desordenados se dá de forma idêntica a seus pares ordenados.
- Porém, ao percorrer estes containers com um iterador se perceberá que os elementos não ficam organizados em ordem
- A quebra da restrição de ordenação leva a um ganho em velocidade de busca

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
}
```

Veja estes dois exemplos.

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
}
```

A única diferença é que em um utilizamos um container map e no outro o container unordered\_map.

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
}
```

Nos dois exemplos criamos um arranjo associativo que tem como chave o nome dos meses e como registro o número de dias neste mês.

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
```

```
Setembro -> 30
Abril -> 30
Dezembro -> 31
Fevereiro -> 28
```

```
meses["novembro"] = 30;
meses["dezembro"] = 31;
cout << "Setembro -> ";
cout << meses["setembro"] << endl;
cout << "Abril -> ";
cout << meses["abril"] << endl;
cout << "Dezembro -> ";
cout << meses["dezembro"] << endl;
cout << "Fevereiro -> ";
cout << meses["fevereiro"] << endl;
return 0;
```

```
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
```

```
Setembro -> 30
Abril -> 30
Dezembro -> 31
Fevereiro -> 28
```

```
meses["novembro"] = 30;
meses["dezembro"] = 31;
cout << "Setembro -> ";
cout << meses["setembro"] << endl;
cout << "Abril -> ";
cout << meses["abril"] << endl;
cout << "Dezembro -> ";
cout << meses["dezembro"] << endl;
cout << "Fevereiro -> ";
cout << meses["fevereiro"] << endl;
return 0;
```

```
}
```

Os dois exemplos equivalentes do ponto de vista de resultados.

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
```

```
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    cout << "Setembro -> ";
    cout << meses["setembro"] << endl;
    cout << "Abril -> ";
    cout << meses["abril"] << endl;
    cout << "Dezembro -> ";
    cout << meses["dezembro"] << endl;
    cout << "Fevereiro -> ";
    cout << meses["fevereiro"] << endl;
    return 0;
```

```
}
```

Internamente, porém, sabemos que os dados do unordered\_map não estão ordenados.

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    unordered_map<string, int>::iterator i;
    for (i = meses.begin(); i!=meses.end(); ++i){
        cout << i->first << ": ";
        cout << i->second << endl;
    }
    return 0;
```

```
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    map<string, int>::iterator i;
    for (i = meses.begin(); i!=meses.end(); ++i){
        cout << i->first << ": ";
        cout << i->second << endl;
    }
    return 0;
```

```
}
```

```
#include <iostream>
#include <string>
#include <unordered_map>
```

```
using namespace std;
```

```
int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    unordered_map<string, int>::iterator i;
    for (i = meses.begin(); i!=meses.end(); ++i){
        cout << i->first << ": ";
        cout << i->second << endl;
    }
    return 0;
```

```
}
```

```
#include <iostream>
#include <string>
#include <map>
```

```
using namespace std;
```

```
int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["marco"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;
    map<string, int>::iterator i;
    for (i = meses.begin(); i!=meses.end(); ++i){
        cout << i->first << ": ";
        cout << i->second << endl;
    }
    return 0;
```

```
}
```

Veja agora este exemplo onde os elementos do containers são percorridos. Como os elementos do primeiro não estão em ordem, os resultados devem ser diferentes.

Os elementos do exemplo da direita estão em ordem em relação à chaves. Como a chave é do tipo string, eles estão em ordem alfabética.

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
```

```
    cout << i->second << endl;
    return 0;
}
```

dezembro: 31  
 novembro: 30  
 setembro: 30  
 outubro: 31  
 agosto: 31  
 junho: 30  
 maio: 31  
 fevereiro: 28  
 abril: 30  
 julho: 31  
 março: 31  
 janeiro: 31

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
```

```
    cout << i->second << endl;
    return 0;
}
```

abril: 30  
 agosto: 31  
 dezembro: 31  
 fevereiro: 28  
 janeiro: 31  
 julho: 31  
 junho: 30  
 maio: 31  
 março: 31  
 novembro: 30  
 outubro: 31  
 setembro: 30

Os elementos do unordered\_map estão em ordem arbitrária quando percorremos o container, mas claramente isto não é um problema para o tipo de dado que estamos guardando.

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

int main(){
    unordered_map<string, int> meses;
    meses["janeiro"] = 31;
```

```
    cout << i->second << endl;
    return 0;
}
```

dezembro: 31  
 novembro: 30  
 setembro: 30  
 outubro: 31  
 agosto: 31  
 junho: 30  
 maio: 31  
 fevereiro: 28  
 abril: 30  
 julho: 31  
 março: 31  
 janeiro: 31

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main(){
    map<string, int> meses;
    meses["janeiro"] = 31;
```

```
    cout << i->second << endl;
    return 0;
}
```

abril: 30  
 agosto: 31  
 dezembro: 31  
 fevereiro: 28  
 janeiro: 31  
 julho: 31  
 junho: 30  
 maio: 31  
 março: 31  
 novembro: 30  
 outubro: 31  
 setembro: 30

Ter os meses do ano em ordem alfabética é de pouca utilidade na maioria das aplicações. Por isto, podemos abrir mão da ordenação dos dados caso isto gere um ganho de tempo nas consultas.

## Containers Desordenados - Como funcionam

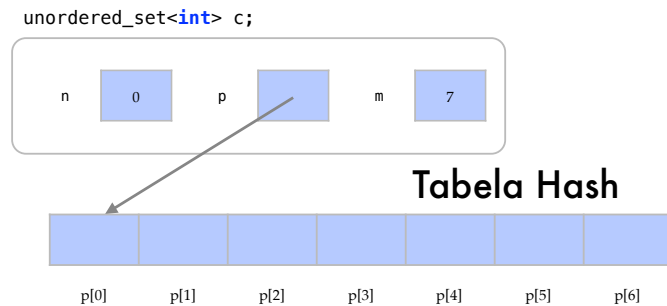
- Assim como os containers ordenados, todos os containers desordenados são também baseados em estruturas de dados similares
- Estes containers desordenados são baseados em estratégias de transformação de chave, o que é chamado de *Hashing*

## Transformação de Chave (Hashing)

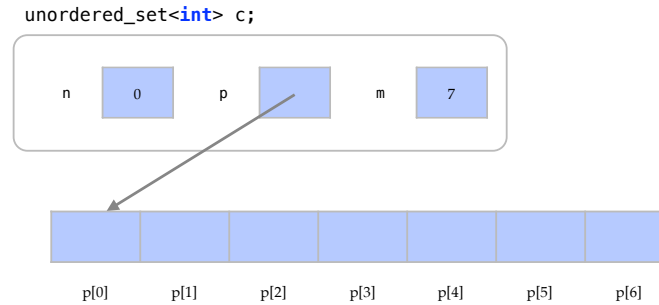
- As estruturas baseadas em transformação de chave utilizam um arranjo para guardar os elementos do conjunto
- Quando uma chave é pesquisada, esta chave é transformada em um número através de uma operação de transformação de chave
- O número retornado representa uma posição em um arranjo, onde o elemento é guardado

```
unordered_set<int> c;
```

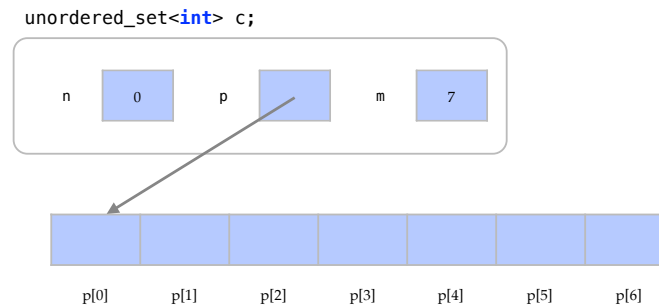
Imagine um `unordered_set` que representa um conjunto desordenado.



Este arranjo alocado na memória é chamado de Tabela Hash. É nele que os elementos serão inseridos.

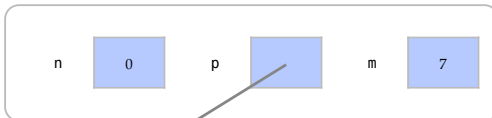


Temos neste objeto uma variável que guarda o número de elementos no container; um ponteiro para um arranjo alocado na memória; e o tamanho deste arranjo.



O tamanho inicial desta tabela depende da implementação. Mas suponha que a tabela hash tenha espaço alocado para 7 elementos como na figura.

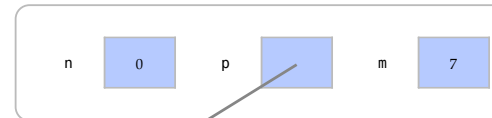
```
unordered_set<int> c;
```



p[0] p[1] p[2] p[3] p[4] p[5] p[6]

```
c.insert(32);
```

```
unordered_set<int> c;
```



p[0] p[1] p[2] p[3] p[4] p[5] p[6]

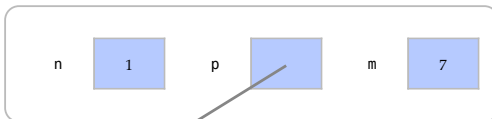
```
c.insert(32);
```

$h(32) = 32 \% 7 = 4$

Suponha agora que queremos inserir na tabela o elemento 32. A função de transformação  $h(x)$  deverá transformar esta chave 32 em uma posição do arranjo.

A função de transformação  $h(x)$  mais comum para números inteiros é  $h(x) = x \% m$ , onde  $m$  é o tamanho da tabela hash e  $x$  é o valor da chave.

```
unordered_set<int> c;
```



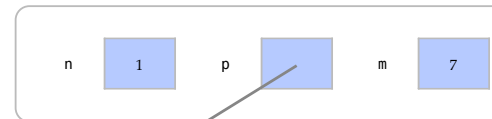
p[0] p[1] p[2] p[3] p[4] p[5] p[6]

```
c.insert(32);
```

$h(32) = 32 \% 7 = 4$

Assim, a função de transformação indica que o elemento deve ser inserido na posição  $p[4]$  da Tabela Hash.

```
unordered_set<int> c;
```



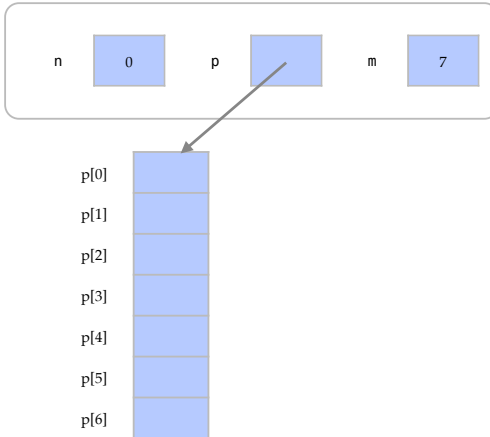
p[0] p[1] p[2] p[3] p[4] p[5] p[6]

```
c.insert(46);
```

$h(46) = 46 \% 7 = 4$

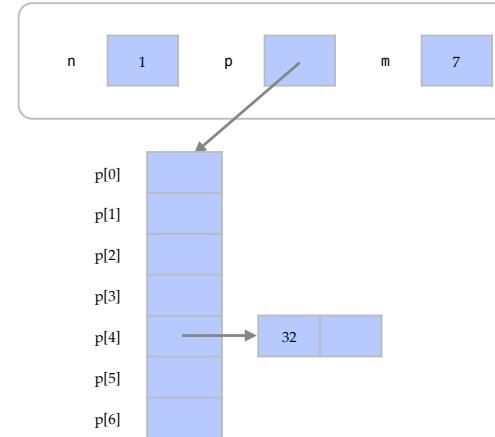
Um problema óbvio com tabelas hash é que a transformação de chave pode querer colocar um outro elemento na mesma posição da tabela.

```
unordered_set<int> c;
```



A estratégia mais comum para tratamento de colisões é transformar este arranjo de elementos em um arranjo ponteiros para listas encadeadas de elementos.

```
unordered_set<int> c;
```

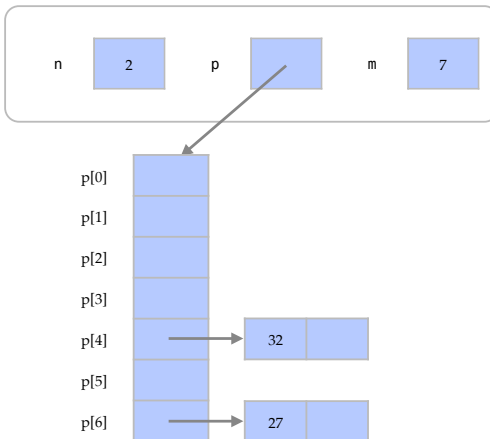


```
c.insert(32);
```

$$h(32) = 32 \% 7 = 4$$

Por exemplo, ao se inserir o elemento 32, ele vai para a posição  $h(32) = 4$  da tabela hash como um elemento em uma célula de uma lista encadeada.

```
unordered_set<int> c;
```

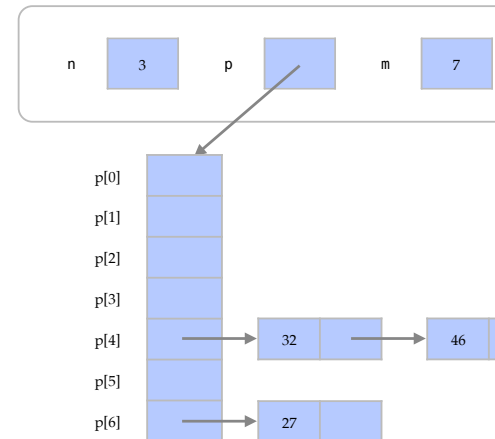


```
c.insert(27);
```

$$h(27) = 27 \% 7 = 6$$

Cada elemento fica em uma célula que é composta de um elemento e um ponteiro para uma possível próxima célula.

```
unordered_set<int> c;
```



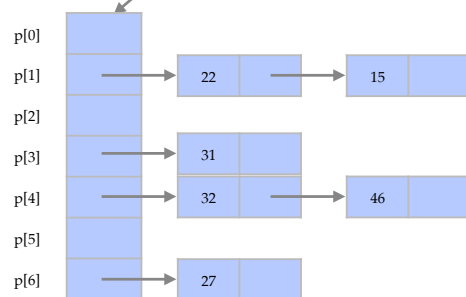
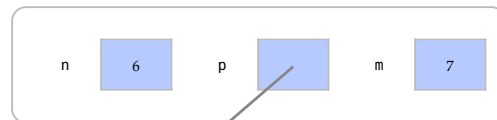
```
c.insert(46);
```

$$h(46) = 46 \% 7 = 4$$

Em caso de colisão, o elemento é simplesmente colocado como último da lista.



```
unordered_set<int> c;
```



```
c.insert(31);  
c.insert(22);  
c.insert(15);
```

Em caso de colisão, o elemento é simplesmente colocado como último da lista.

## Tabelas Hash - Análise

- A maior vantagem de tabelas hash é a sua eficiência em relação a custo médio para pesquisa, inserção e remoção
- Para uma tabela onde os dados estão bem distribuídos o custo de qualquer operação na tabela é  $O(1)$
- O STL já inclui estratégias para que os dados fiquem sempre bem distribuídos na tabela

## Tabelas Hash - Análise

- Para a estratégia de tratamento colisão apresentada, no pior caso, se todos os elementos forem para a mesma posição, o custo de cada operação seria  $O(n)$
- Porém, além da probabilidade deste caso ser desprezível, um tratamento de colisão através de árvores em vez de listas pode levar este pior caso a  $O(\log n)$

## Tabelas Hash - Análise

- Mais ainda, é possível garantir de várias formas uma boa distribuição dos dados na tabela
- Todas as tabelas hash alocam espaço para tabelas maiores quando o número de elementos cresce
- Usualmente uma nova tabela maior é criada quando o número de elementos é maior que 70% do tamanho da tabela
- Esta proporção é chamada de fator de carga

# Tabelas Hash - Análise

- Quando o tamanho da tabela aumenta uma operação  $O(n)$  copia os elementos para a nova tabela
- Para evitar isto, é comum manter duas tabelas
  - Sempre que um elemento novo é inserido na tabela nova, um elemento da tabela antiga é transferido para a tabela nova com custo  $O(1)$
- Porém, nesta estratégia, a busca de um elemento, apesar de ser ainda  $O(1)$ , sempre precisará ser feita em duas tabelas

# Tabelas Hash - Análise

- A maior desvantagem das tabelas hash é que os dados ficam desordenados na tabela
- Para retornar todos os itens em ordem é necessário colocar tudo para uma outra estrutura ordenada ou ordená-los em uma estrutura de sequência.
  - Qualquer opção terá um custo  $O(n \log n)$
- Por isto, esta é uma estrutura a ser utilizada apenas quando os dados em ordem realmente não são necessários

## Containers de Sequência ou Associativos? Qual?



## Qual contêiner associativo?

Container	Mantém itens em ordem	Há um registro associado à chave	Suporta itens repetidos
set			
multiset			
map			
multimap			
unordered_set			
unordered_multiset			
unordered_map			
unordered_multimap			

## Containers - Dicas

- Entenda como é alocado espaço para mais elementos no container.
- Os containers `vector` e `deque` oferecem uma função `reserve(int)` que já aloca um arranjo com o espaço suficiente para qualquer número de elementos

## Containers - Dicas

- Use `c.empty()` para saber se um container está vazio
- Esta função utiliza dados pré-processados para retornar uma resposta
- Não use `c.size() == 0` para saber se um container está vazio
- Fazer isto envolve 2 operações: cálculo do número de elementos e comparação

## Containers - Dicas

- Entenda como os elementos são organizados no container.
- Entender como os dados são organizados no container garante que se entenda o número de operações envolvidas em cada função disponível para o container
- Assim, entendendo a estrutura, não precisamos decorar o custo de cada operação

## Exercício

- Desenhe passo a passo como seria a representação interna das seguintes estruturas:

```
set<int> c;  
c.insert(4);  
c.insert(6);  
c.insert(2);  
c.insert(7);  
c.insert(5);  
c.erase(2);
```

```
unordered_set<int> c;  
c.insert(4);  
c.insert(6);  
c.insert(2);  
c.insert(7);  
c.insert(5);  
c.erase(2);
```

Suponha uma árvore não balanceada para o `set` e tabelas hash de tamanho 5 para o `unordered_set`

# Provas

- Dia 17/junho - Prova 2
- Dia 01/julho - Provas finais

# Adaptadores de Container

- Os containers que vimos até agora são chamados de containers de primeira classe
- Os adaptadores de container não são containers de primeira classe
- Os adaptadores de container são apenas versões mais limitadas dos containers de primeira classe para aplicações específicas

# Adaptadores de Container

- Estas versões limitadas dos containers de sequência contêm apenas funções `push` e `pop`
- As outras opções do container não estão disponíveis no adaptador de container
- Estes containers também não podem ser percorridos com iteradores

# Adaptadores de Container - Stack

- **Stack** (Pilha) permite apenas a inserção ou remoção em uma extremidade do container de sequência
- O adaptador tem apenas as funções de `push` e `pop`, que na verdade chamam as funções `push_back` e `pop_back` dos containers de sequência originais

## Adaptadores de Container - Stack

- Um **stack** pode ser usado para implementar uma pilha com um **vector**, **list**, ou **deque**.
- Nos casos de omissão, um **deque** será utilizado por padrão.
- Neste adaptador, a função **top** obtém o elemento no topo da pilha, equivalente à função **back** dos containers de sequência

## Adaptadores de Contêiner - Stack

Em uma pilha, só temos acesso ao elemento do topo de uma sequência.



## Adaptadores de Contêiner - Stack

`c.push(4)`



A função **push** coloca um elemento no topo da pilha.

## Adaptadores de Contêiner - Stack

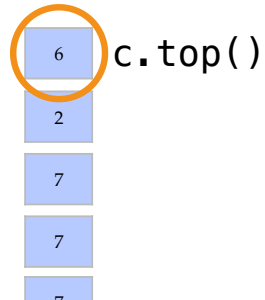
A função **pop** retira um elemento do topo da pilha.

`c.pop()`



# Adaptadores de Contêiner - Stack

A função `top` retorna o valor do elemento no topo da pilha.



# Stack com vector ou deque

`c.push(2);` 2

Imagine o arranjo de um vector ou um deque representando uma pilha com um elemento.

# Stack com vector ou deque

`c.push(2);` 2  
`c.push(3);` 2 3

Quando inserimos mais um elemento no stack, os elementos são copiados para um arranjo com o dobro do tamanho para colocar o novo elemento.

# Stack com vector ou deque

`c.push(2);` 2  
`c.push(3);` 2 3  
`c.push(7);` 2 3 7

Para inserir mais um elemento, um arranjo com o dobro do tamanho precisa novamente ser alocado

# Stack com vector ou deque

c.push(2);	2
c.push(3);	2 3
c.push(7);	2 3 7
c.push(1);	2 3 7 1

Agora é possível inserir mais um elemento no topo da pilha sem aumentar o tamanho do arranjo

# Stack com vector ou deque

c.push(2);	2
c.push(3);	2 3
c.push(7);	2 3 7
c.push(1);	2 3 7 1
c.push(4);	2 3 7 1 4

Para inserir o elemento 4, um arranjo maior precisa ser novamente alocado

# Stack com vector ou deque

c.push(2);	2
c.push(3);	2 3
c.push(7);	2 3 7
c.push(1);	2 3 7 1
c.push(4);	2 3 7 1 4
c.pop();	2 3 7 1

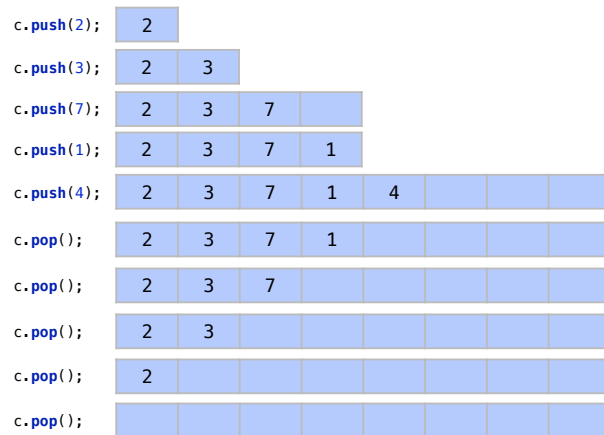
Ao remover elementos do topo do stack, um arranjo menor não é necessariamente criado.

# Stack com vector ou deque

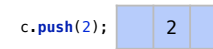
c.push(2);	2
c.push(3);	2 3
c.push(7);	2 3 7
c.push(1);	2 3 7 1
c.push(4);	2 3 7 1 4
c.pop();	2 3 7 1
c.pop();	2 3 7

Ao remover elementos do topo do stack, um arranjo menor não é necessariamente criado.

# Stack com vector ou deque

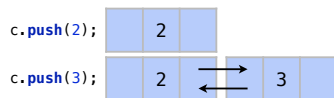


# Stack com list



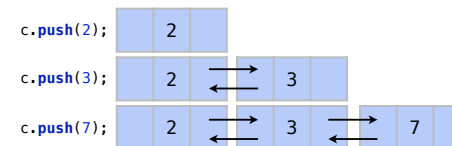
Ao colocar elementos em um stack representado por um list, apenas uma célula para este elemento é alocada

# Stack com list



Sempre que um elemento novo é inserido, o espaço exato para este elemento é alocado.

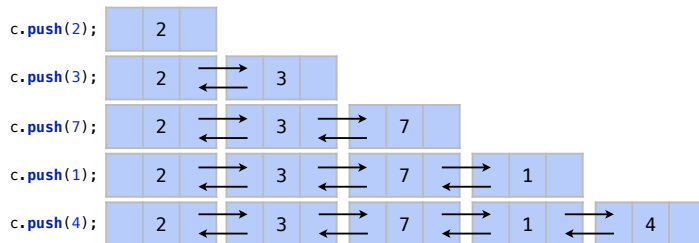
# Stack com list



Sempre que um elemento novo é inserido, o espaço exato para este elemento é alocado.

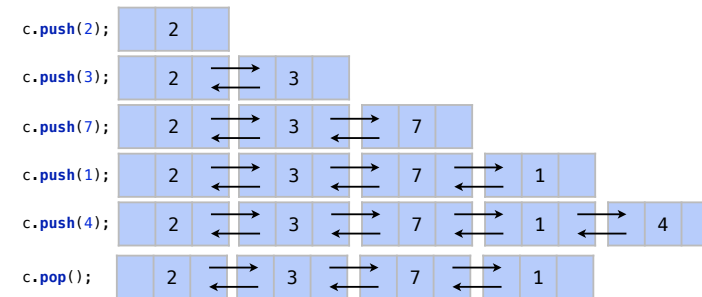


## Stack com list



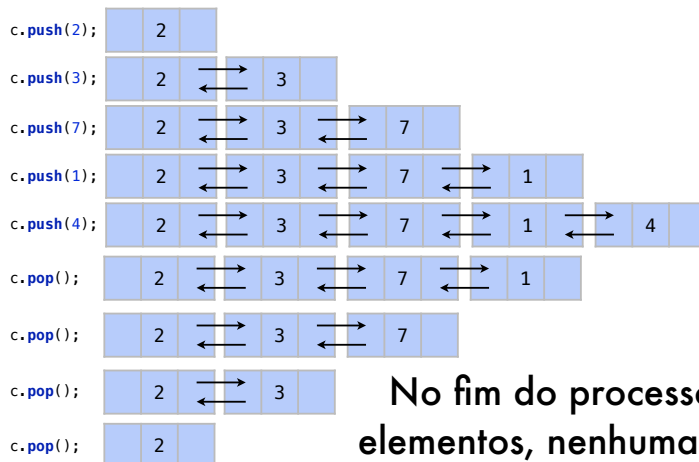
Para cada elemento, porém, são alocados dois ponteiros, que acabam gastando memória extra.

## Stack com list



Quando um elemento é removido, a memória gasta com este elemento é também desalocada.

## Stack com list



No fim do processo, se não há elementos, nenhuma memória está sendo gasta.

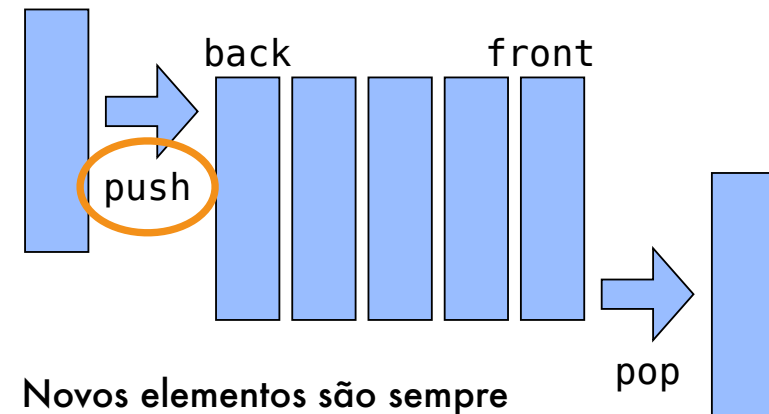
## Comandos

- Para utilizar stacks, o cabeçalho `#include <stack>` precisa ser incluído
- O comando `stack<int> pilha;` cria uma pilha de números inteiros
- Para especificar que queremos a pilha representada com um list, usamos `stack<int, list<int>> pilha;`

## Adaptadores de Contêiner - Queue

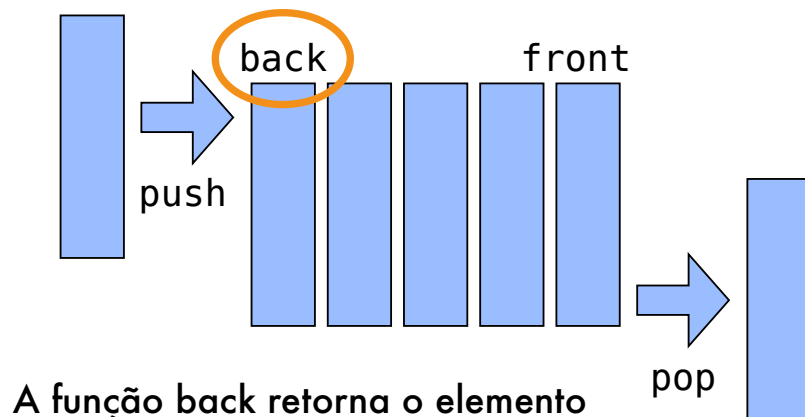
- Queue (Filas) permitem inserções (push) no fim da lista e retiradas do início (pop)
- É melhor implementá-la com `list` ou `deque` (padrão)
- `front` e `back` dão acesso aos primeiro e último elementos

## Adaptadores de Contêiner - Queue



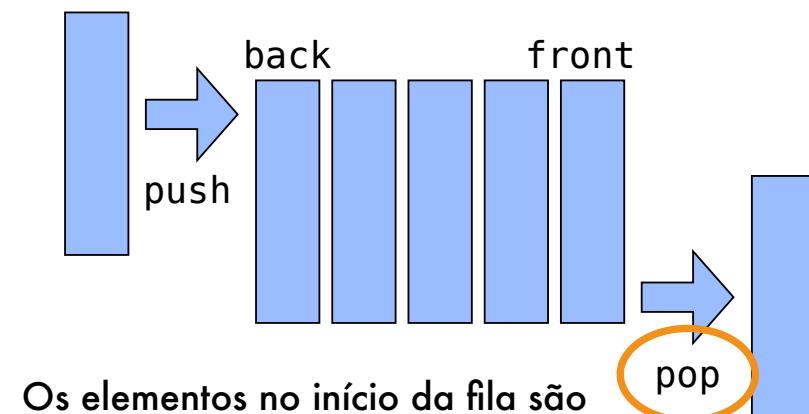
Novos elementos são sempre inseridos no fim da fila.

## Adaptadores de Contêiner - Queue



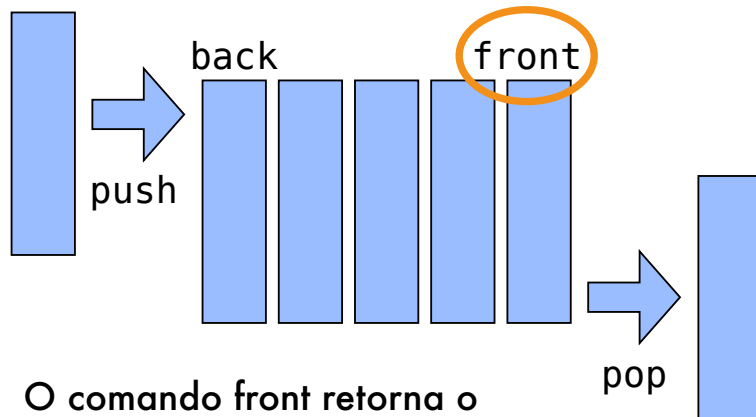
A função `back` retorna o elemento do fim da fila.

## Adaptadores de Contêiner - Queue



Os elementos no início da fila são removidos primeiro.

## Adaptadores de Contêiner - Queue



O comando front retorna o elemento do início da fila.

## Queue com vector

```
c.push(2);
```

2
---

Imagine o arranjo de um vector representando uma fila com um elemento.

## Queue com vector

```
c.push(2);
```

2
---

```
c.push(3);
```

2	3
---	---

Para inserimos um novo elemento, um arranjo com o dobro do tamanho é alocado e os elementos copiados para ele

## Queue com vector

```
c.push(2);
```

2
---

```
c.push(3);
```

2	3
---	---

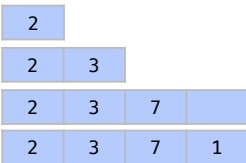
```
c.push(7);
```

2	3	7	
---	---	---	--

Mais uma vez, um arranjo maior precisa ser alocado.

## Queue com vector

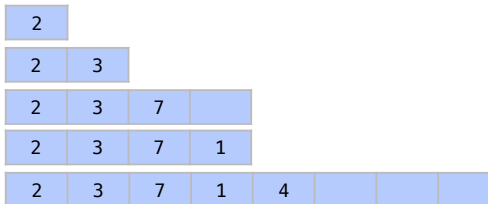
```
c.push(2);  
c.push(3);  
c.push(7);  
c.push(1);
```



Neste passo, conseguimos inserir o elemento 1 sem alocar um novo arranjo

## Queue com vector

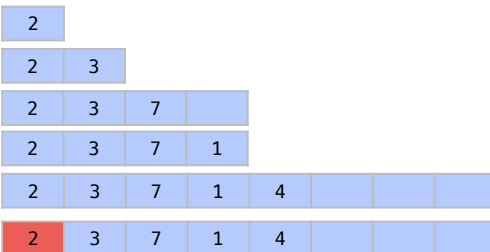
```
c.push(2);  
c.push(3);  
c.push(7);  
c.push(1);  
c.push(4);
```



Para inserir o elemento 4, um novo arranjo com o dobro do tamanho é alocado novamente

## Queue com vector

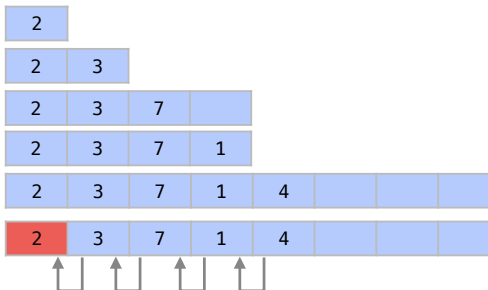
```
c.push(2);  
c.push(3);  
c.push(7);  
c.push(1);  
c.push(4);  
c.pop();
```



Ao se remover um elemento, aquele elemento que está na frente da fila é removido

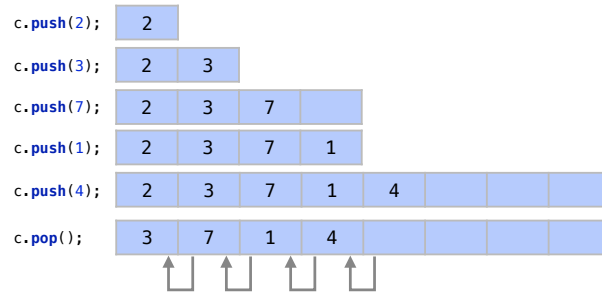
## Queue com vector

```
c.push(2);  
c.push(3);  
c.push(7);  
c.push(1);  
c.push(4);  
c.pop();
```



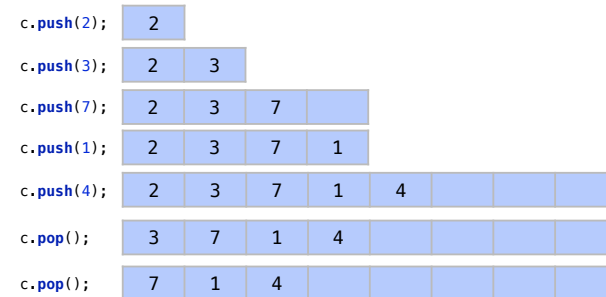
Como o vector não tem uma operação  $O(1)$  para remover o primeiro elemento do container, todos os elementos do container precisam ser deslocados.

## Queue com vector



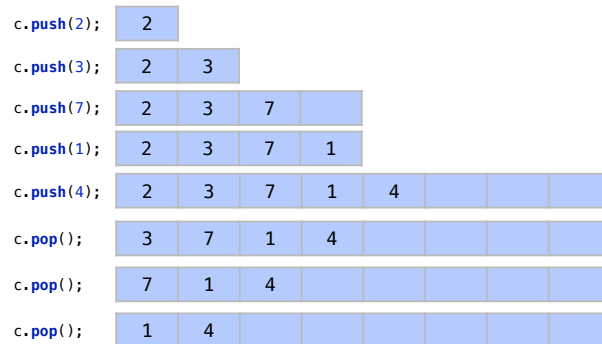
Este deslocamento resulta no primeiro elemento da fila removido, porém isto ocorre com um alto custo  $O(n)$ , o que torna o vector uma estrutura pouco apropriada para um queue.

## Queue com vector



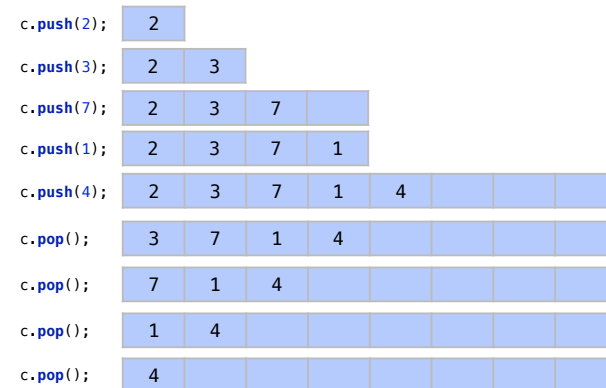
Na remoção do próximo elemento, todos os elementos são deslocados novamente.

## Queue com vector



A cada operação de remoção, um deslocamento  $O(n)$  é executado

## Queue com vector



Deste modo, o vector é uma estrutura muito ineficiente para queue

## Queue com vector

`c.push(2);`

2
---

 $O(1)$   
`c.push(3);`

2	3
---	---

 $O(1)$   
`c.push(7);`

2	3	7	
---	---	---	--

 $O(1)$   
`c.push(1);`

2	3	7	1
---	---	---	---

 $O(1)$   
`c.push(4);`

2	3	7	1	4			
---	---	---	---	---	--	--	--

 $O(1)$   
`c.pop();`

3	7	1	4				
---	---	---	---	--	--	--	--

 $O(n)$   
`c.pop();`

7	1	4					
---	---	---	--	--	--	--	--

 $O(n)$   
`c.pop();`

1	4						
---	---	--	--	--	--	--	--

 $O(n)$   
`c.pop();`

4							
---	--	--	--	--	--	--	--

 $O(n)$   
`c.pop();`

--	--	--	--	--	--	--	--

 $O(n)$

## Queue com deque

`c.push(2);`

2
---

  
`c.push(3);`

2	3
---	---

  
`c.push(7);`

2	3	7	
---	---	---	--

  
`c.push(1);`

2	3	7	1
---	---	---	---

  
`c.push(4);`

2	3	7	1	4			
---	---	---	---	---	--	--	--

Imagine agora o arranjo de um deque que está representando um queue. Ao fim dos passos de inserção, temos um arranjo igual ao do vector.

## Queue com deque

`c.push(2);`

2
---

  
`c.push(3);`

2	3
---	---

  
`c.push(7);`

2	3	7	
---	---	---	--

  
`c.push(1);`

2	3	7	1
---	---	---	---

  
`c.push(4);`

2	3	7	1	4			
---	---	---	---	---	--	--	--

  
`c.pop();`

	3	7	1	4			
--	---	---	---	---	--	--	--

Porém, quando um elemento é removido na frente da fila, os elementos não são deslocados pois no deque apenas indicamos onde começa a fila.

## Queue com deque

`c.push(2);`

2
---

  
`c.push(3);`

2	3
---	---

  
`c.push(7);`

2	3	7	
---	---	---	--

  
`c.push(1);`

2	3	7	1
---	---	---	---

  
`c.push(4);`

2	3	7	1	4			
---	---	---	---	---	--	--	--

  
`c.pop();`

	3	7	1	4			
--	---	---	---	---	--	--	--

  
`c.pop();`

		7	1	4			
--	--	---	---	---	--	--	--

  
`c.pop();`

			1	4			
--	--	--	---	---	--	--	--

  
`c.pop();`

				4			
--	--	--	--	---	--	--	--

  
`c.pop();`

--	--	--	--	--	--	--	--

Isto se repete para as outras remoções.

## Queue com deque

c.push(2);	<table border="1"><tr><td>2</td></tr></table>	2	O(1)				
2							
c.push(3);	<table border="1"><tr><td>2</td><td>3</td></tr></table>	2	3	O(1)			
2	3						
c.push(7);	<table border="1"><tr><td>2</td><td>3</td><td>7</td></tr></table>	2	3	7	O(1)		
2	3	7					
c.push(1);	<table border="1"><tr><td>2</td><td>3</td><td>7</td><td>1</td></tr></table>	2	3	7	1	O(1)	
2	3	7	1				
c.push(4);	<table border="1"><tr><td>2</td><td>3</td><td>7</td><td>1</td><td>4</td></tr></table>	2	3	7	1	4	O(1)
2	3	7	1	4			
c.pop();	<table border="1"><tr><td></td><td>3</td><td>7</td><td>1</td><td>4</td></tr></table>		3	7	1	4	O(1)
	3	7	1	4			
c.pop();	<table border="1"><tr><td></td><td></td><td>7</td><td>1</td><td>4</td></tr></table>			7	1	4	O(1)
		7	1	4			
c.pop();	<table border="1"><tr><td></td><td></td><td></td><td>1</td><td>4</td></tr></table>				1	4	O(1)
			1	4			
c.pop();	<table border="1"><tr><td></td><td></td><td></td><td></td><td>4</td></tr></table>					4	O(1)
				4			
c.pop();	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td></tr></table>						O(1)

Assim, com um deque, todas as operações são  $O(1)$

## Queue com list

c.push(2);	<table border="1"><tr><td>2</td></tr></table>	2		
2				
c.push(3);	<table border="1"><tr><td>2</td><td>→</td><td>3</td></tr></table>	2	→	3
2	→	3		

Sempre que um elemento novo é inserido, o espaço exato para este elemento é alocado.

## Queue com list

c.push(2);	<table border="1"><tr><td>2</td></tr></table>	2
2		

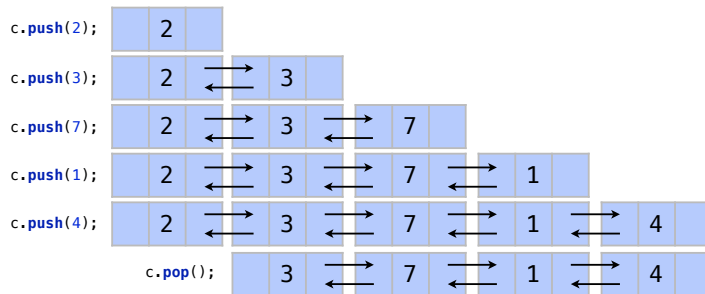
Ao colocar elementos em um stack representado por um list, apenas uma célula para este elemento é alocada

## Queue com list

c.push(2);	<table border="1"><tr><td>2</td></tr></table>	2								
2										
c.push(3);	<table border="1"><tr><td>2</td><td>→</td><td>3</td></tr></table>	2	→	3						
2	→	3								
c.push(7);	<table border="1"><tr><td>2</td><td>→</td><td>3</td><td>→</td><td>7</td></tr></table>	2	→	3	→	7				
2	→	3	→	7						
c.push(1);	<table border="1"><tr><td>2</td><td>→</td><td>3</td><td>→</td><td>7</td><td>→</td><td>1</td></tr></table>	2	→	3	→	7	→	1		
2	→	3	→	7	→	1				
c.push(4);	<table border="1"><tr><td>2</td><td>→</td><td>3</td><td>→</td><td>7</td><td>→</td><td>1</td><td>→</td><td>4</td></tr></table>	2	→	3	→	7	→	1	→	4
2	→	3	→	7	→	1	→	4		

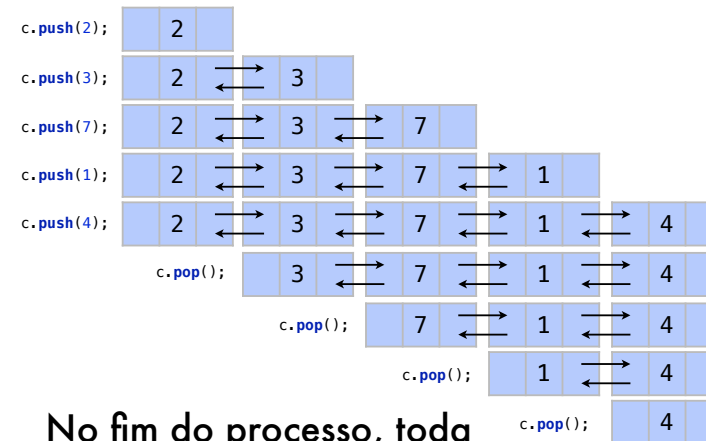
Para cada elemento, porém, são alocados dois ponteiros, que acabam gastando memória extra.

## Queue com list



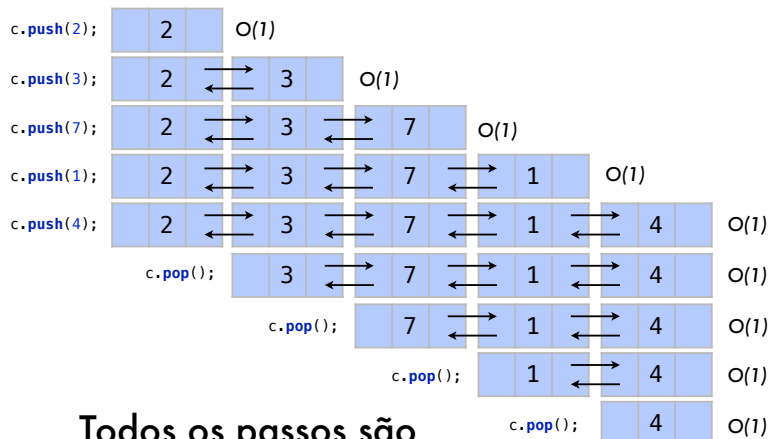
Quando um elemento é removido, ele é removido diretamente do início do list e a memória gasta com este elemento é desalocada.

## Queue com list



No fim do processo, toda a memória é desalocada.

## Queue com list



Todos os passos são também  $O(1)$ .

## Comandos

- Para utilizar um queue, o cabeçalho `#include <queue>` precisa ser incluído
- O comando `queue<int> fila;` cria uma pilha de números inteiros
- Para especificar que queremos a fila representada com um list, usamos `queue<int, list<int>> fila;`



## Adaptadores de Contêiner - priority\_queue

- `priority_queue` (fila de prioridade) permitem inserir elementos de maneira que o maior elemento do container é removido a cada chamada de `pop`
- Pode ser implementado com `deque` ou `vector` (padrão)

## Adaptadores de Contêiner - priority\_queue

- `push` insere um elemento e `pop` retira o maior elemento (chamado de elemento de maior prioridade)
- `top` consulta o elemento de maior prioridade

## Queue com vector

11	10	7	9	5	6	4	8	2	3	1
----	----	---	---	---	---	---	---	---	---	---

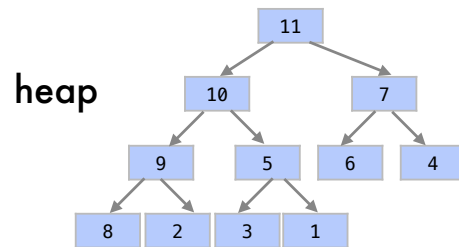
Assim fica organizado o arranjo de um vector que representa um queue

## Queue com vector

11	10	7	9	5	6	4	8	2	3	1
----	----	---	---	---	---	---	---	---	---	---

Apesar de não estar ordenado, o primeiro elemento deste arranjo é sempre o maior elemento do container.

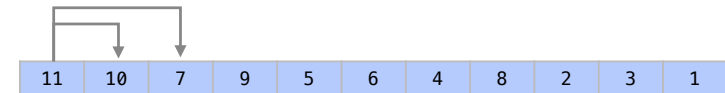
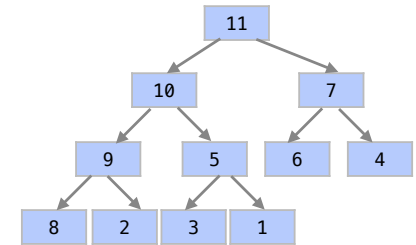
# Queue com vector



11 10 7 9 5 6 4 8 2 3 1

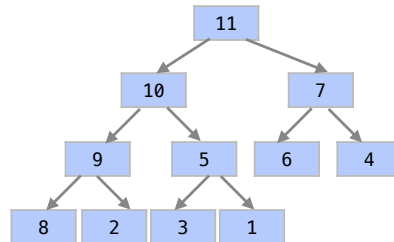
Este arranjo representa uma árvore chamada heap.

# Queue com vector



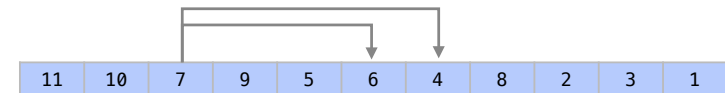
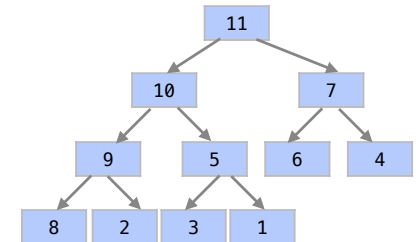
O heap é representado pelo arranjo de modo que os filhos de um elemento na posição  $i$  estejam na posição  $2i+1$  e  $2i+2$

# Queue com vector



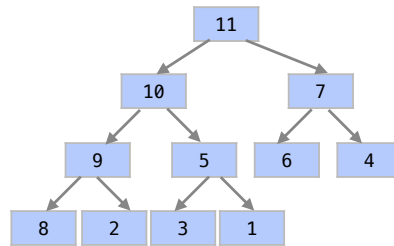
Os filhos do elemento 11 da posição 0 são 10 e 7, das posições 1 e 2.

# Queue com vector



Os filhos do elemento 7, da posição 2, são 6 e 4, das posições 5 e 6.

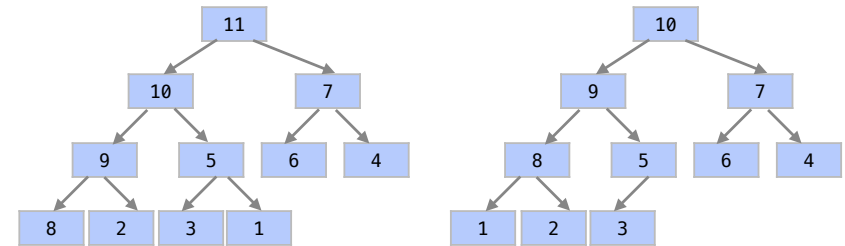
## Queue com vector



11 10 7 9 5 6 4 8 2 3 1

Uma propriedade importante destas árvores é que, apesar dos elementos não estarem ordenados, os filhos de um elemento sempre são menores que o pai

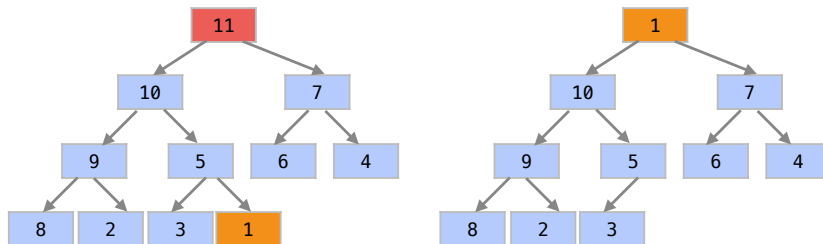
## Queue com vector



10 9 7 8 5 6 4 1 2 3

Assim, caso o elemento de maior prioridade seja removido, o segundo maior elemento toma seu lugar em tempo  $O(\log n)$ .

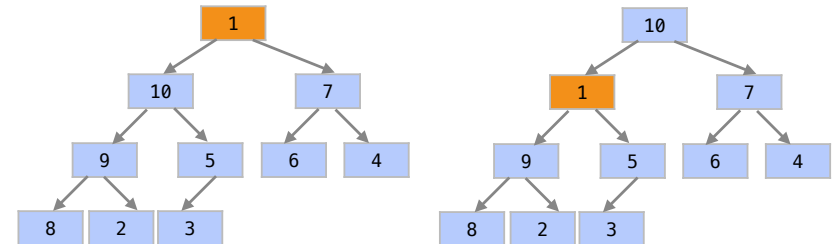
## Queue com vector



1 10 7 9 5 6 4 8 2 3

Para rearranjar o heap após remover o elemento do topo, o último elemento do heap toma o lugar do elemento removido.

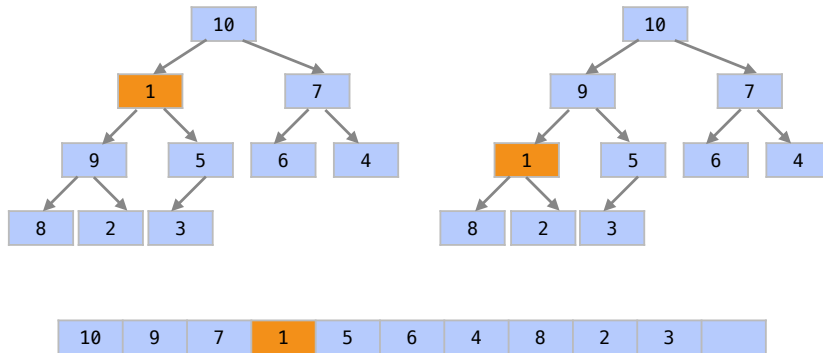
## Queue com vector



10 1 7 9 5 6 4 8 2 3

O elemento em questão é comparado com seus filhos e troca de posição com maior deles.

## Queue com vector

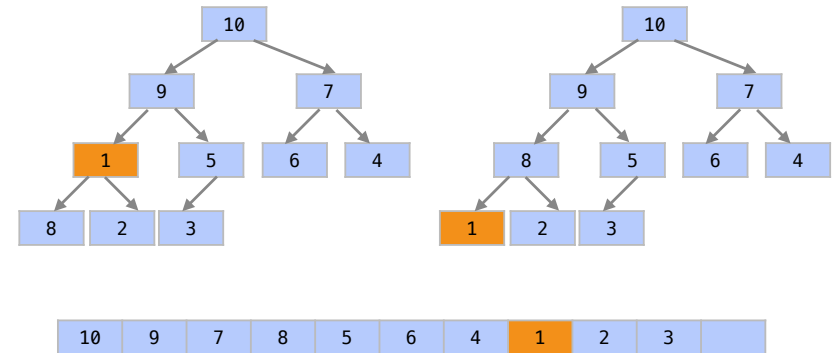


Novamente, o elemento é comparado com seus filhos e troca de posição com maior deles.

## Comandos

- Para utilizar um `priority_queue`, o cabeçalho `#include <queue>` precisa ser incluído
- O comando `priority_queue<int> fila;` cria uma fila de prioridade com números inteiros

## Queue com vector



O processo se repete até que o elemento esteja em uma posição que respeita a condição de ser maior que seus filhos.

## Algoritmos da STL

- Parte fundamental da STL são os algoritmos
- Estes algoritmos podem ser utilizados para executar operações comuns a dados guardados em containers da STL como:
  - Ordenação, busca, cópia, somatório, contagem...

# Algoritmos da STL

- Estes algoritmos são usualmente implementações muito eficientes, além de nos poupar tempo de programação
- Antes de implementar uma operação básica, é sempre bom conferir se a STL já disponibiliza este algoritmo
- O cabeçalho <algorithm> inclui algoritmos comuns para containers
- O cabeçalho <numeric> inclui algoritmos comuns para dados numéricos

## Exemplo - sort

```
#include <algorithm>
...
int nums[] = {32,71,12,45,26,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 12 45 26 80 53 33
// Ordenar elementos nas posições entre begin() e begin()+4
sort(v.begin(), v.begin()+4);
// (12 32 45 71) 26 80 53 33
// Ordenar elementos nas posições entre begin() e end()
sort(v.begin(), v.end());
// 12 26 32 33 45 53 71 80
...
```

## Lista de algoritmos modificadores de sequência da STL

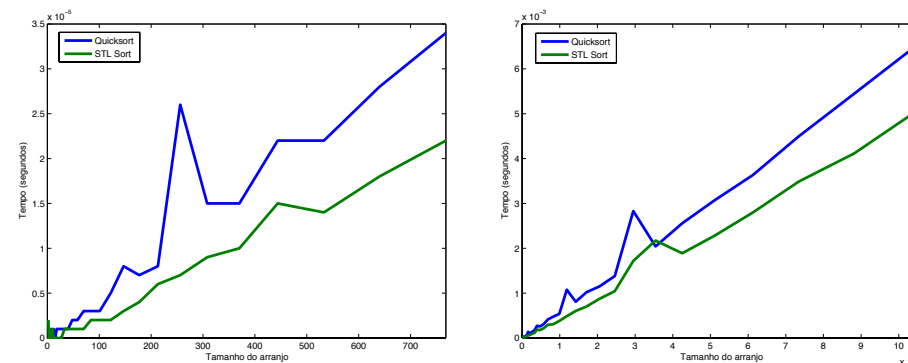
copy	remove	reverse_copy
copy_backward	remove_copy	rotate
fill	remove_copy_if	rotate_copy
fill_n	remove_if	stable_partition
generate	replace	swap
generate_n	replace_copy	swap_ranges
iter_swap	replace_copy_if	transform
partition	replace_if	unique
random_shuffle	reverse	unique_copy

Conferir lista completa em:

<http://www.cplusplus.com/reference/algorithm/>

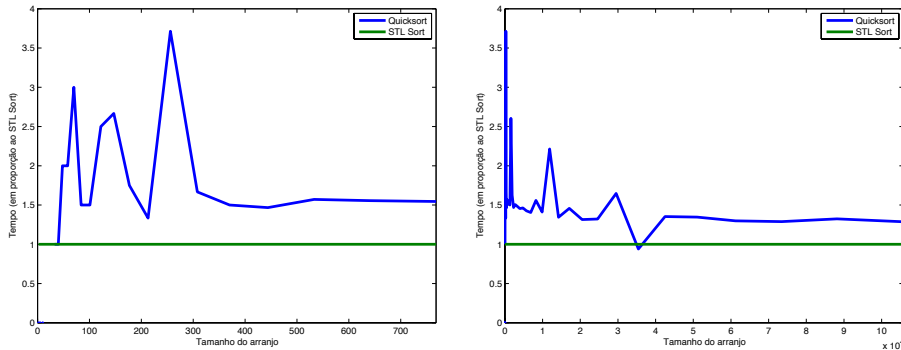
## Teste Real de Tempo

O algoritmo da STL é ainda mais eficiente do que um quicksort.



# Teste Real de Tempo

O quicksort é quase 4 vezes mais lento em arranjos pequenos e quase 50% mais lento em arranjos maiores



# Exemplo - remove

```
#include <algorithm>
...
int nums[] = {32,71,33,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 33 45 33 80 53 33
// Remover as ocorrências de 33 entre begin() e end()
remove(v.begin(), v.end(), 33);
// 32 71 45 80 53
imprimeContainer(v);
...
```

# Exemplo - fill

```
#include <algorithm>
...
vector< char > letras(10);
fill(letras.begin(), letras.end(), 'B');
// B B B B B B B B B B
imprimeContainer(letras);
// Colocar A nos 5 elementos a partir de begin()
fill_n(letras.begin(), 5, 'A');
// A A A A A B B B B B
imprimeContainer(letras);
...
```

# Exemplo - replace

```
#include <algorithm>
...
int nums[] = {32,71,33,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 33 45 33 80 53 33
// Substitui ocorrências de 33 por 100
replace(v.begin(), v.end(), 33, 100);
// 32 71 100 45 100 80 53 100
imprimeContainer(v);
...
```

# Exemplo - random\_shuffle

```
#include <algorithm>
...
int nums[] = {1, 2, 3, 4, 5, 6, 7, 8};
vector<int> v(nums, nums+8);
// 1 2 3 4 5 6 7 8
// Embaralha elementos entre begin() e end()
random_shuffle(v.begin(), v.end());
// 6 4 8 2 3 5 1 7
imprimeContainer(v);
...
```

# Algoritmos não-modificadores de sequência da STL

adjacent_find	find	find_if
count	find_each	mismatch
count_if	find_end	search
equal	find_first_of	search_n

Conferir lista completa em:  
<http://www.cplusplus.com/reference/algorithm/>

# Exemplo - equal

```
#include <algorithm>
...
vector<int> v1(5,10); // 10 10 10 10 10
vector<int> v2(5,10); // 10 10 10 10 10
// compara duas sequências de valores
bool resultado = equal(v1.begin(), v1.end(), v2.begin());
if (resultado){
    cout << "Os vetores têm os mesmos valores" << endl;
} else {
    cout << "Os vetores são diferentes" << endl;
}
...
```

# Exemplo - find

```
#include <algorithm>
...
int nums[] = {32,71,12,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 12 45 33 80 53 33
vector<int>::iterator r;
// Procura elemento 12 e retorna iterador para ele
r = find(v.begin(), v.end(), 12);
// Se o elemento foi encontrado
if (r != v.end()){
    // Elemento: 12
    cout << "Elemento:" << *r << endl;
    // Posição: 2
    cout << "Posição:" << r-v.begin() << endl;
}
...
```

## Exemplo - binary\_search

```
#include <algorithm>
...
int nums[] = {32,71,12,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 12 45 33 80 53 33
// Ordena os elementos entre begin() e end()
sort(v.begin(),v.end());
// 12 32 33 33 45 53 71 80
// Se o elemento foi encontrado
if (binary_search(v.begin(), v.end(), 12)){
    cout << "Elemento encontrado" << endl;
}
...
```

## Exemplo - count

```
#include <algorithm>
...
int nums[] = {32,71,33,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 33 45 33 80 53 33
// Conta quantas ocorrências de 33 entre begin() e end()
int r = count(v.begin(), v.end(), 33);
cout << r << " números 33" << endl;
// 3 números 33
...
```

## Exemplo - min\_element

```
#include <algorithm>
...
int nums[] = {32,71,12,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 12 45 33 80 53 33
// Procura o menor elemento entre begin() e end()
int r = *min_element(v.begin(),v.end());
cout << "Menor elemento:" << r << endl;
// Menor elemento: 12
...
```

## Exemplo - max\_element

```
#include <algorithm>
...
int nums[] = {32,71,12,45,33,80,53,33};
vector<int> v(nums, nums+8);
// 32 71 12 45 33 80 53 33
// Procura o maior elemento entre begin() e end()
int r = *max_element(v.begin(),v.end());
cout << "Maior elemento:" << r << endl;
// Maior elemento: 80
...
```



## Algoritmos numéricos do cabeçalho <numeric>

accumulate	partial_sum
inner_product	adjacent_difference

Conferir lista completa em:  
<http://www.cplusplus.com/reference/numeric/>

## Exercício

- Desenhe passo a passo como seria a representação interna das seguintes estruturas:

```
queue<int> c;  
c.push(2);  
c.push(3);  
c.push(7);  
c.push(1);  
c.push(4);  
c.pop();  
c.pop();  
c.pop();  
c.pop();
```

Suponha que a fila está sendo representada com a) um vector, b) um deque e c) um list.

## Exemplo - accumulate

```
#include <numeric>  
...  
int nums[] = {32,71,12,45,33,80,53,33};  
vector<int> v(nums, nums+8);  
// 32 71 12 45 33 80 53 33  
// Soma todos os elementos do vector iniciando somatório com 0  
int r = accumulate(v.begin(), v.end(), 0);  
cout << "Somatório:" << r << endl;  
// Somatório: 359  
...
```

Programação de Computadores  
Estruturas de Dados  
Alan R R Freitas