

Programação e Desenvolvimento de Software 2

TADs específicos (STL)

Camila Laranjeira

<http://github.com/flaviovdf/programacao-2>

Introdução

- Existem infinitos TADs
 - Alguns a gente implementa
 - Outros são providos para nós (para problemas genéricos o suficiente)



Introdução

- Nenhum programa é escrito em uma linguagem de programação a partir do zero
- Geralmente
 - Linguagens vêm com bibliotecas
 - Impossível decorar todas
 - Usamos a documentação para entender
- Bibliotecas podem ser vistas como:
 - Conjunto de TADs e funções de uso geral

Exemplos de TADs

- Coleções/Containers
 - Listas, Árvores
- Números
 - Bignum
 - Complexo
- Geometria
 - Ponto

TADs do dia a dia

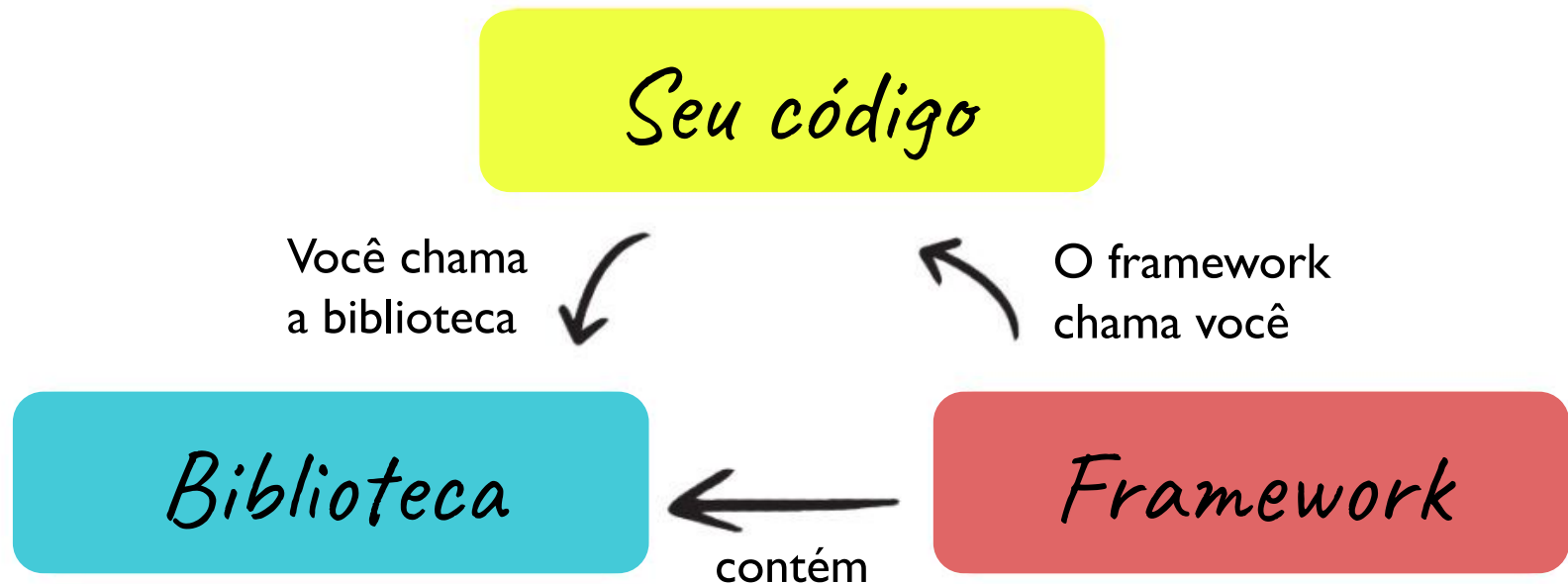
- Existem na biblioteca padrão de C++
- Para PDS2
 - Não precisamos ir muito além da padrão
- No pior dos casos, tente na Boost
 - <https://www.boost.org/>
 - Bignums e números complexos

Bibliotecas v. Frameworks

- Bibliotecas
 - Funcionalidades mais comuns
 - Containers, aritmética, matemática etc
- Frameworks
 - Servem para um propósito maior
 - Serviços web
 - Engines de jogos já prontas

Uma forma de ver

■ Regra geral (há exceções): Inversão de controle



Uma forma de ver

Princípio de Hollywood:

“Don’t call us, we’ll call you”

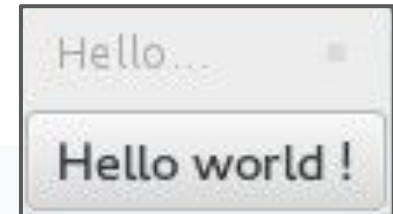


```
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
    QApplication app (argc, argv);

    QPushButton button ("Hello world !");
    button.show();

    return app.exec();
}
```



Uma forma de ver

Princípio de Hollywood:

“Don’t call us, we’ll call you”



```
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
    QApplication app (argc, argv);

    QPushButton button ("Hello world !");
    button.show();

    return app.exec();
}
```

Você preenche as funcionalidades

Mas o framework é que gerencia as chamadas

Componentes da biblioteca padrão

- Strings
 - Suporte para expressões regulares
- Ponteiros inteligentes
 - Gerenciamento de recursos (e.g. `unique_ptr`, `shared_ptr`)
- Containers (e.g. `list` e `map`) e algoritmos (e.g. `sort()`)
 - Convencionalmente chamado STL
(Standard Template Library)

Headers da Biblioteca Padrão

- Qualquer funcionalidade da biblioteca padrão é fornecida através de um header padrão:

```
#include <string>
#include <vector>
```

- A biblioteca padrão é definida em um namespace chamado std. Para usar as funcionalidades, o prefixo std:: é usado:

```
std::string gato = "O gato miou.";
std::vector<std::string> palavras = {"gato", "mia"};
```

Headers da Biblioteca Padrão

- Por simplicidade, podemos evitar o uso de `std::`
- Geralmente não é uma boa prática carregar todos os nomes de um namespace no namespace global

```
#include <string>
using namespace std;
string s = "O gato miou.";
```

- Explícito é melhor!

```
#include <string>
std::string s = "O gato miou.";
```

Strings em C++

■ Funções no nível da biblioteca

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

fx Functions

Convert from strings

stoi <small>C++11</small>	Convert string to integer (function template)
stol <small>C++11</small>	Convert string to long int (function template)
stoul <small>C++11</small>	Convert string to unsigned integer (function template)
stoll <small>C++11</small>	Convert string to long long (function template)
stoull <small>C++11</small>	Convert string to unsigned long long (function template)
stof <small>C++11</small>	Convert string to float (function template)
stod <small>C++11</small>	Convert string to double (function template)
stold <small>C++11</small>	Convert string to long double (function template)

Convert to strings

to_string <small>C++11</small>	Convert numerical value to string (function)
to_wstring <small>C++11</small>	Convert numerical value to wide string (function)

Standard Template Library

Templates indicam um tipo genérico

- Programação Genérica
 - A mesma definição de função atua da mesma forma sobre objetos de diferentes tipos
- Polimorfismo em tempo de compilação
- Templates (C++), Generics (Java)

STL

struct que aceita tipos genéricos

Tipo T usado na struct

Um outro template de struct

Aqui o `node_t<T>` usa o T
localmente definido na lista

```
#ifndef PDS2_LISTAGENERICA_H  
#define PDS2_LISTAGENERICA_H
```

```
template <typename T>  
struct node_t {  
    T elemento;  
    node_t *proximo;  
};
```

```
template <typename T>  
struct Lista {  
    node_t<T> *_inicio;  
    node_t<T> *_fim;  
    int _num_elementos;  
  
    Lista();  
    ~Lista();  
    void inserir_elemento(T elemento);  
    void imprimir();  
};  
#endif
```

STL

- Programação Genérica
- **Polimorfismo** em tempo de compilação

```
#ifndef PDS2_LISTAGENERICA_H
#define PDS2_LISTAGENERICA_H

template <typename T>
struct node_t {
    T elemento;
    node_t *proximo;
};

template <typename T>
struct Lista {
    node_t<T> *_inicio;
    node_t<T> *_fim;
    int _num_elementos;

    Lista();
    ~Lista();
    void inserir_elemento(T elemento);
    void imprimir();
};

#endif
```


STL - usando o template

```
#include <string>

#include "listasimples.h"

int main(void) {
    Lista<int> lista = Lista<int>();
    for (int i = 0; i < 1000; i++)
        lista.inserir_elemento(i);
    lista.imprimir();

    Lista<std::string> lista2 = Lista<std::string>();
    lista2.inserir_elemento("flavio");
    lista2.inserir_elemento("camila");
    lista2.imprimir();
    return 0;
}
```

Standard Template Library

Containers

- Coleções de objetos
- Uso de containers apropriados para uma tarefa e suportá-los com operações fundamentais é crucial
- Containers usam templates por baixo
 - Assim fazemos uso de qualquer tipo
- Nem sempre o mesmo container é o melhor para diferentes problemas

Containers

Elementos em sequência

Sequenciais

- Vector
- Deque
- List

Associação entre
chave e elemento

Associativos

- Set
- Map
- Multiset
- Multimap

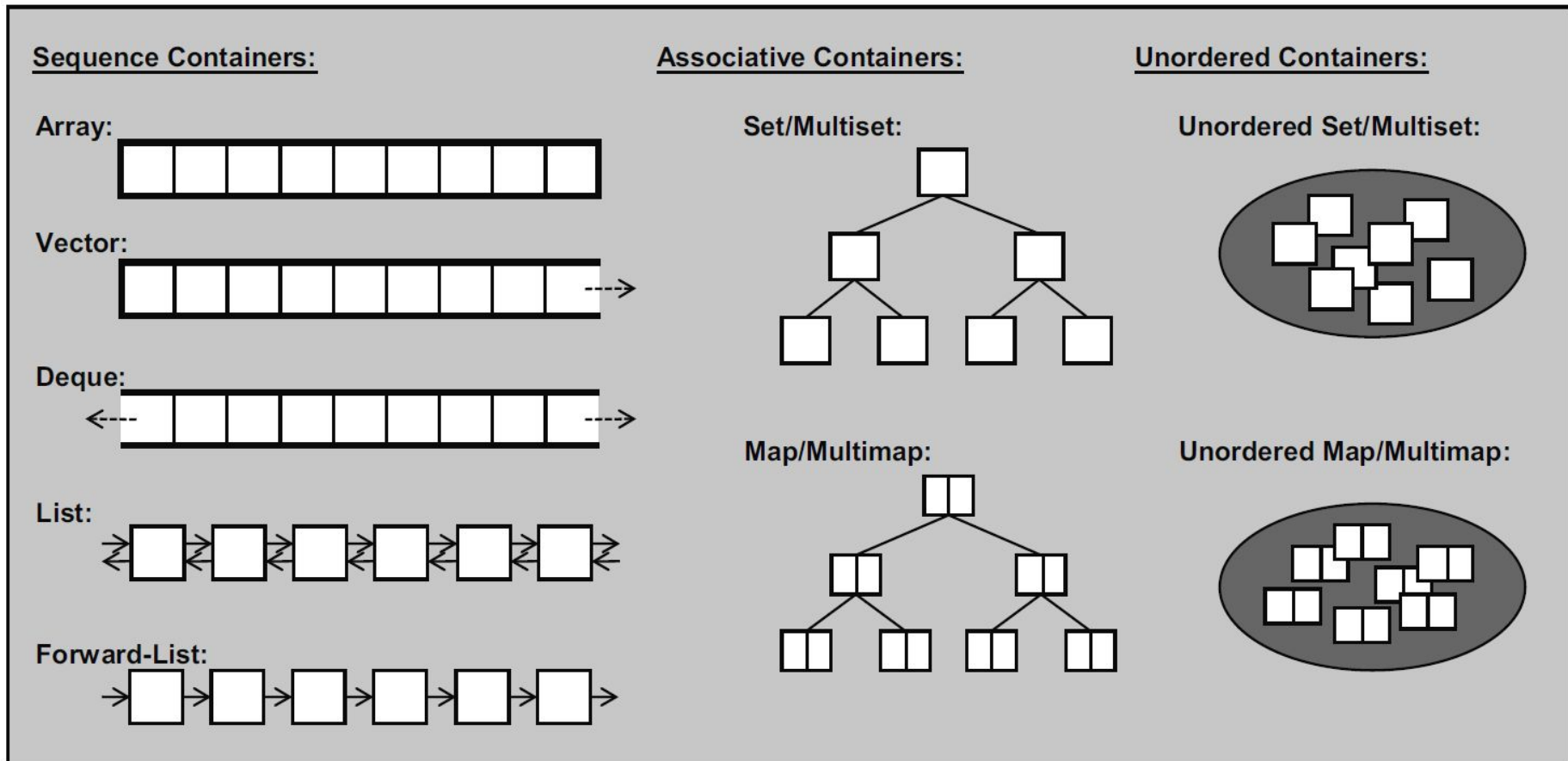
Usam outro container
por baixo e adicionam
restrições.

Adaptadores

- Stack
- Queue
- Priority queue

Containers

■ Fazendo elo com os TADs



Containers sequenciais

- Elementos estão em uma ordem linear
- Gerenciamento automático de memória que permite o tamanho variar dinamicamente
- Operações básicas

Sequenciais

- Vector
- Deque
- List

- `front()`
- `back()`
- `push_back()`
- `pop_back()`
- `size()`
- `empty()`

Vector

- Um dos mais úteis!
- Internamente é um array (armazenado sequencialmente em memória)
- Acesso por índice (e.g. `vec[0]`)
- Adição/remoção de elementos no final
 - `push_back()`, `pop_back()`
- Leia a documentação para a lista completa de operações: <https://cplusplus.com/reference/vector/vector/>

Vector de inteiros

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

int main() {
    std::vector<int> v = {7, 5, 16, 8};
    v.push_back(25);
    v.push_back(13);

    for(int n : v) {
        std::cout << n << std::endl;
    }
    return 0;
}
```

■ Similar aos
nossos TADs

■ push_back
→ Inserção;

■ for each: equivale ao laço abaixo

```
for (int i = 0; i < v.size(); i++) int n = v[i];
```

Vector de classe

Vamos iniciar com uma classe Pessoa

```
#include <string>
```

```
struct Pessoa {  
    const std::string _nome;  
    int _idade;
```

No nosso caso _nome nunca muda, const

```
    // Construtor com lista de  
    inicialização
```

```
    Pessoa(std::string nome, int idade):  
        _nome(nome), _idade(idade) {}
```

Primeira vez que vemos esse construtor.
Funciona igual ao anterior.

```
    std::string get_nome() const {  
        return this->_nome;
```

```
    }  
    int get_idade() const {  
        return this->_idade;
```

Métodos const nunca mudam o objeto. Garantido.

```
    }  
};
```


Vector de classe

```
#include <iostream>
#include <vector>
#include "pessoa.h"

int main() {
    std::vector<Pessoa> pessoas;
    pessoas.push_back(Pessoa("Ana", 18));
    pessoas.push_back(Pessoa("Pedro", 19));

    // Primeira forma de acesso
    std::cout << pessoas[0].get_nome();
    std::cout << pessoas[1].get_nome();

    // Segunda forma, com at
    std::cout << pessoas.at(0).get_nome();
    std::cout << pessoas.at(1).get_nome();
    return 0;
}
```

■ at → Acesso

Vector

- Um vector pode ser copiado:

```
std::vector<Pessoa> lista2 = lista_tel;
```

- Atribuir um vector envolve copiar seus elementos. Após a inicialização de lista2, lista_tel e lista2 têm cópias separadas de cada elemento

- Tal inicialização pode ser cara

- Quando a cópia é indesejável, referências e ponteiros devem ser utilizados

Qual o problema da chamada abaixo?

```
void atualiza_idade(std::vector<Pessoa> pessoas) {  
    for (Pessoa pessoa : pessoas)  
        pessoa.set_idade(pessoa.get_idade() + 1);  
}
```

Qual o problema da chamada abaixo?

```
void atualiza_idade(std::vector<Pessoa> pessoas) {  
    for (Pessoa pessoa : pessoas)  
        pessoa.set_idade(pessoa.get_idade() + 1);  
}
```



- (1) Passagem por cópia

- (2) Mesmo se fosse por referência, for each faz cópia

Qual o problema da chamada abaixo?

```
void atualiza_idade(std::vector<Pessoa>& pessoas) {  
    for (Pessoa& pessoa : pessoas)  
        pessoa.set_idade(pessoa.get_idade() + 1);  
}
```

■ Note o uso da referência durante toda a função

Diferentes laços

■ Laço clássico

```
std::vector<int> dados = {0, 7, 8, 1, 3};  
for (int i = 0; i < dados.size(); i++)  
    std::cout << dados[i];
```

■ Laço compacto

```
for (int x : dados)  
    std::cout << x;
```

■ Laço para a referência

```
for (int &x : dados)  
    x *= 2;
```

Pausa pra um exercício

- Escreva uma função “zera_valores” que recebe um vetor de inteiros, encontra valores menores que seus vizinhos imediatos (esquerda e direita) e os substitui por zero.

Vetor de entrada: 1 4 5 2 3 1 7

Vetor ao final da função: 1 4 5 0 3 0 7

Pausa pra um exercício

```
void zera_valores(std::vector<int>& vec){
    for(int i = 1; i < vec.size(); i++){
        if(vec.at(i) < vec[i-1] && vec.at(i) < vec[i+1]){
            vec.at(i) = 0;
        }
    }
}

int main() {
    std::vector<int> vec = {1, 4, 5, 2, 3, 1, 7 };
    zera_valores(vec);
    for(const int &x : vec)
        std::cout << x << " ";
}
```


Pausa pra um exercício

Agora com iterators só pra exercitar :)

```
void zera_valores(std::vector<int>& vec){
    for(auto i = vec.begin() + 1; i != vec.end(); i++){
        if(*i < *(i - 1) && *i < *(i + 1)){
            (*i) = 0;
        }
    }
}

int main() {
    std::vector<int> vec = {1, 4, 5, 2, 3, 1, 7 };
    zera_valores(vec);
    for(const int &x : vec)
        std::cout << x << " ";
}
```

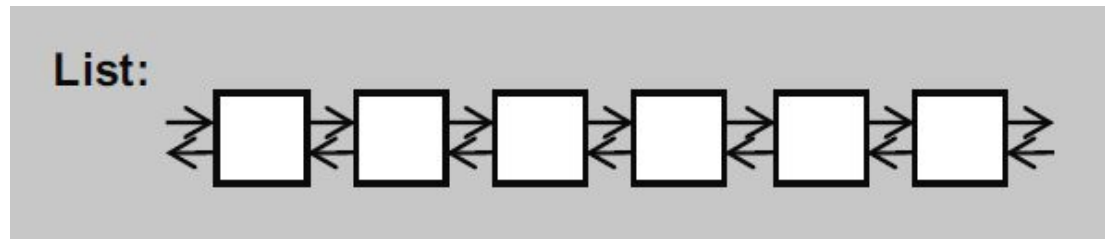
Pausa pra um exercício: Uma rede social

- Crie uma classe Pessoa que tem apenas os atributos nome e um vetor de pessoas amigas.
- Implemente a operação adicionaAmizade() que recebe um objeto Pessoa e modifica os vetores de amizade **de maneira recíproca**, ou seja, de quem adiciona e de quem é adicionado.
- Crie a operação imprime() que imprime o nome do usuário e seu círculo de amigos.
- Na main crie Ana, Pedro, William, e Vitória, onde Ana adiciona Pedro e William, e Pedro adiciona Vitória. Imprima os círculos de amizade.

Qual o problema aqui?

List

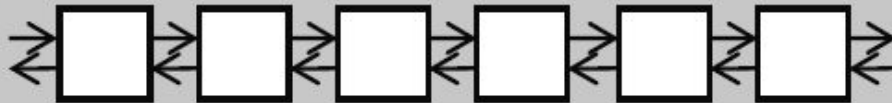
- Lista duplamente encadeada
- Ganhamos funções `push_front()` e `pop_front()`
 - o vector não tinha
- Não temos mais acesso via índice. Motivo?



List

- Não temos mais acesso via índice. Motivo?
 - Não tem um array por baixo como o vector
 - Cada campo guarda os ponteiros dos elementos anterior e próximo
 - Na prática é uma struct

List:



[Clique aqui](#)

```
struct node_t {  
    int valor;  
    node_t *anterior;  
    node_t *próximo;  
}
```

Iteradores

- Iterador para acessar os elementos
- Funciona como ponteiros

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

int main() {
    std::list<int> l = {7, 5, 16, 8};

    l.push_front(25);
    l.push_back(13);

    for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it){
        std::cout << *it << std::endl;
    }
    return 0;
}
```

Iteradores

- Geralmente não acessamos elementos usando índices quando usamos uma lista encadeada
 - É possível, mas é lento
- Quando queremos identificar um elemento em uma list usamos um iterador
- Todo container da biblioteca padrão oferece as funções `begin()` e `end()`, que retorna um iterador pro primeiro e depois do último elemento

Iteradores

São basicamente ponteiros (pelo menos em C++)

- `l.begin()` → primeiro elemento
- `l.begin() + 1` → segundo

```
for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {  
    std::cout << *it << std::endl;  
}
```

- Dado um iterador `it`, `*it` é o elemento que ele se refere, `++it` avança `it` para o próximo elemento
 - ou `next(it)`

Iteradores

São basicamente ponteiros (pelo menos em C++)

As duas chamadas abaixo são equivalentes

```
std::list<int>::iterator ptr = l.begin();  
std::cout << *ptr << std::endl  
ptr = next(it);
```

Aqui usamos aritmética de ponteiros, mas pode fazer com next se achar mais simples

```
std::list<int>::iterator it = l.begin();  
std::cout << *(it++) << std::endl
```


Pequeno Desvio (o auto)

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

int main() {
    std::list<int> l = {7, 5, 16, 8};

    l.push_front(25);
    l.push_back(13);

    auto ptr = l.begin();
    while (ptr != l.end()) {
        std::cout << *ptr << std::endl;
        ptr = next(ptr);
    }
    return 0;
}
```

Que tipo é esse?

Pequeno Desvio (o auto)

- O auto pode ser entendido como:
“Oi C++, tenho preguiça”
- Se for possível, o tipo será inferido a partir do contexto

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

int main() {
    std::list<int> l = {7, 5, 16, 8};

    l.push_front(25);
    l.push_back(13);

    auto ptr = l.begin();
    while (ptr != l.end()) {
        std::cout << *ptr << std::endl;
        ptr = next(ptr);
    }
    return 0;
}
```

Pequeno Desvio (o auto)

A documentação:

<https://docs.microsoft.com/en-us/cpp/cpp/aut-o-cpp?view=msvc-170>

esse begin() sempre vai retornar
`std::list<int>::iterator`

Assim como o next

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

int main() {
    std::list<int> l = {7, 5, 16, 8};

    l.push_front(25);
    l.push_back(13);

    auto ptr = l.begin();
    while (ptr != l.end()) {
        std::cout << *ptr << std::endl;
        ptr = next(ptr);
    }

    return 0;
}
```

Pequeno Desvio (o auto)

■ Sempre que der pra saber o tipo esperado, o auto funciona.

```
auto a = 0;  
auto b = {3.14};  
auto c(5.18);  
  
std::vector<int> dados = {1, 2, 3}  
for (auto& i : dados) funcao(i);
```

Pequeno Desvio (o auto)

■ Sempre que der pra saber o tipo esperado, o auto funciona. Até com retorno de função.

```
double funcao() {  
    return 0.0  
}  
  
int main() {  
    auto var = funcao(); //função retorna int, então var é int  
    return 0;  
}
```

Pequeno Desvio (o auto)

- Mas também não faz milagre...

```
int main(){  
    auto i;  
  
    return 0;  
}
```



list vs vector

- Quando queremos uma sequência de elementos, podemos escolher entre vector e list
- Caso faça deleções e inserções em vários locais, use list
- Mas em geral use vector, ele tem desempenho melhor para percorrer (e.g., `find()`), ordenar e pesquisar (e.g., `sort()`)
- E se precisa remover e inserir apenas nas pontas, a deque é suficiente;

Containers Associativos

(ou Conjuntos sem Repetição)

- Projetados para suportar o acesso direto aos elementos usando chaves pré-determinadas
- Chave
 - Usada internamente para guardar em ordem
- Operações
 - `insert()`
 - `erase()`
 - `find()`
 - `count()`

set

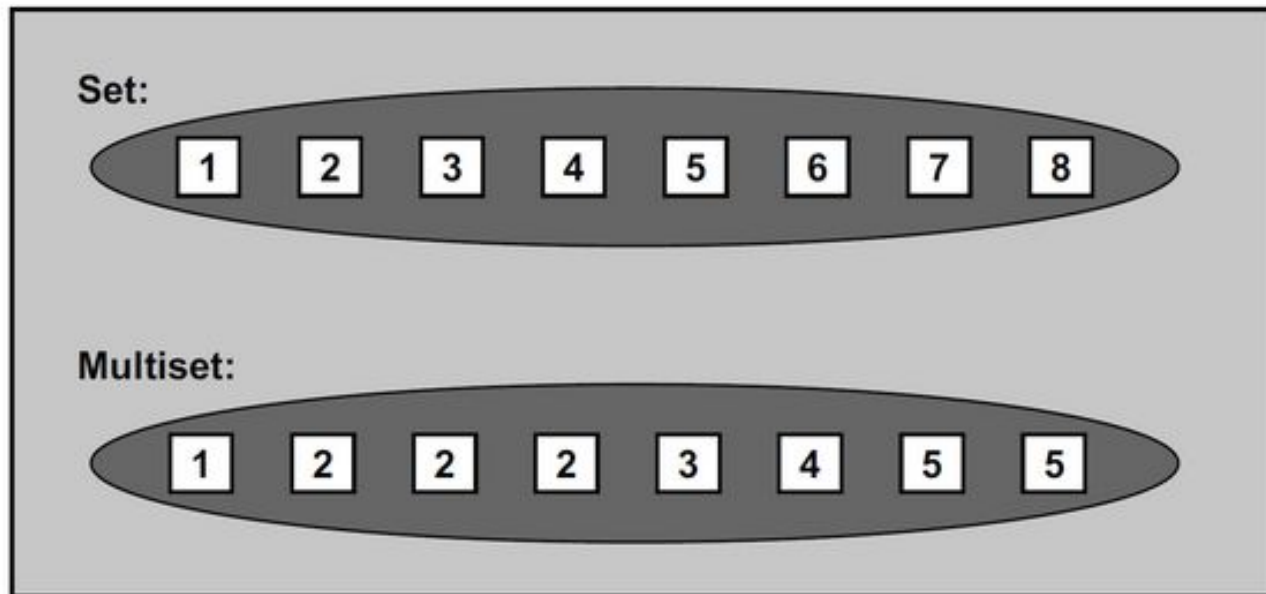
Conjuntos matemáticos

- Armazenado em uma árvore binária
- Comparáveis de acordo com algum critério
 - Números sempre são comparáveis
 - TADs são mais complexos

set

Conjuntos matemáticos

- Tipo `set` armazena valores sem repetição, `multiset` permite repetição



set: qual a saída do código a seguir?

```
#include <iostream>
#include <set>

void imprime(std::set<int> s){
    std::cout << "(" << s.size() << ")" << std::endl;
    for (int e : s) { std::cout << e << " "; }
}

int main() {
    std::set<int> s;
    for(int i = 1; i <= 10; i++)
        s.insert(i);
    imprime(s);

    s.insert(7);
    imprime(s);

    for(int i = 2; i <= 10; i += 2)
        s.erase(i);
    imprime(s);
}
```

set: qual a saída do código a seguir?

```
#include <iostream>
#include <set>

void imprime(std::set<int> s){
    std::cout << "(" << s.size() << ")" << std::endl;
    for (int e : s) { std::cout << e << " "; }
}

int main() {
    std::set<int> s;
    for(int i = 1; i <= 10; i++)
        s.insert(i);
    imprime(s);

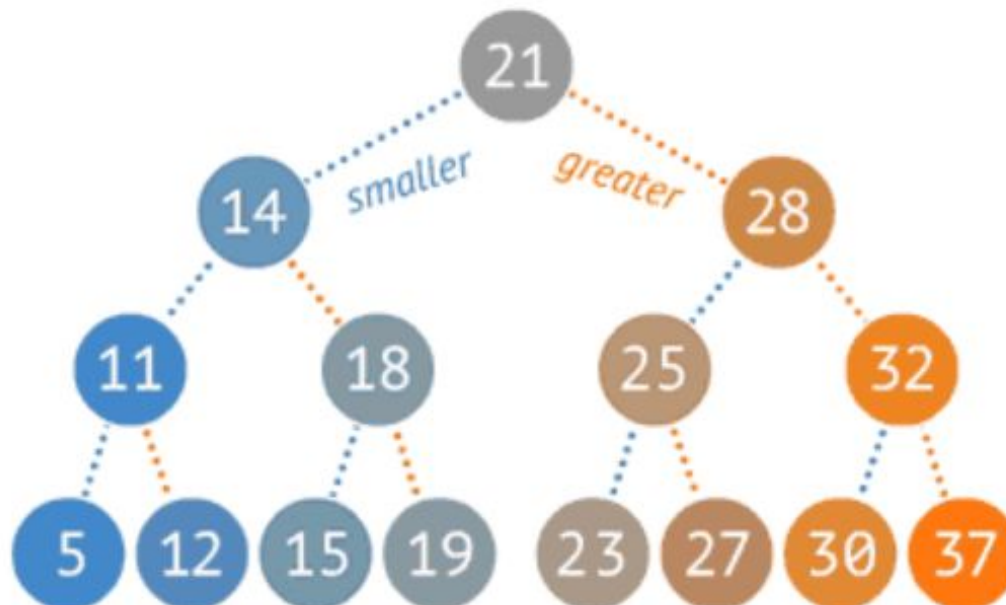
    s.insert(7);
    imprime(s);

    for(int i = 2; i <= 10; i += 2)
        s.erase(i);
    imprime(s);
}
```

```
(10)
1 2 3 4 5 6 7 8 9 10
(10)
1 2 3 4 5 6 7 8 9 10
(5)
1 3 5 7 9
```

Árvore binária

- Cada nó tem dois filhos e invariavelmente temos
 - esquerda < nó < direita
- Consequências: certo nível de previsibilidade ao caminhar na estrutura

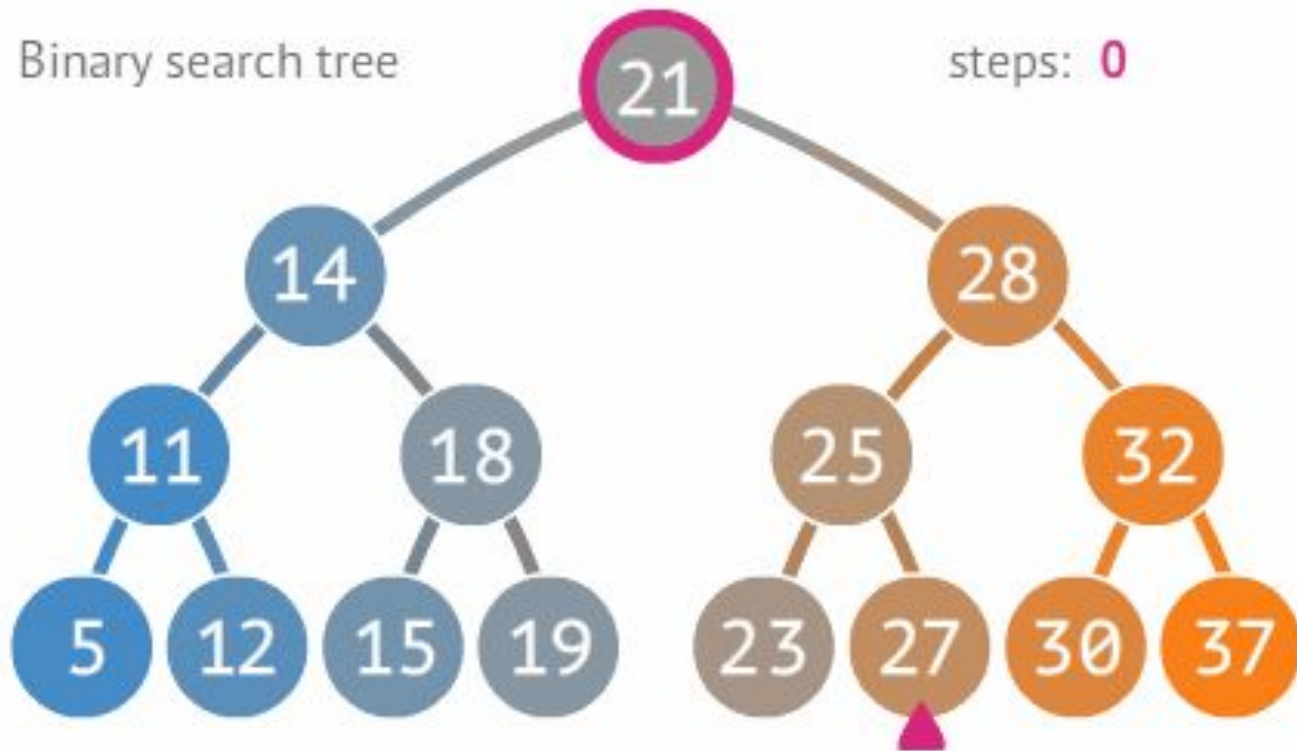


Árvore binária

- Como seria a busca de um elemento no vector?
- Percorre TODOS os elementos até achar
 - complexidade linear $O(n)$
- A árvore é mais rápida! $O(\log n)$
 - Em cada passo eu pulo metade dos elementos
 - $n/2$ várias vezes é $\log(n)$
 - $16 / 2 = 8; 8 / 2 = 4; 4 / 2 = 2; 2 / 2 = 1.$
 - $\log_2(16) = 4$

Árvore binária

- A árvore é mais rápida! $O(\log n)$



Árvore binária

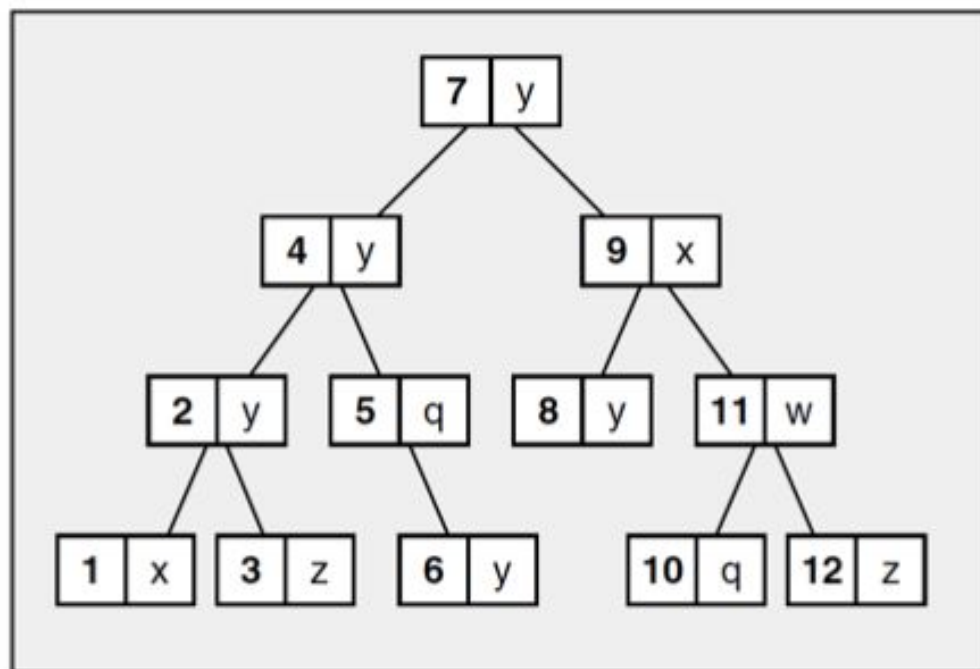
- Permite caminhamento (inserção, deleção, etc.) em tempo logarítmico
 - Não é preciso percorrer todos os elementos

map

- Um mapa armazena pares (chave, valor)
 - Podem ser de qualquer tipo
- A chave é utilizada para achar um elemento rapidamente
 - Uma árvore por baixo ;)
 - Também pode ser tabela hash (assunto de ED)
- Diz-se que um mapa “mapeia chaves para valores”

map

- Cada elemento é um nó
- Guarda uma chave (número neste caso) e um valor (letra neste caso)



map

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

int main() {
    std::map<int, std::string> m;
    m.insert(std::pair<int, std::string>(2017123456, "Joao"));

    m[2016123456] = "Maria";
    m[2018123456] = "Carlos";
    m[2015123456] = "Jose";
    m[2014123456] = "Joana";

    std::map<int, std::string>::iterator it;
    for (it = m.begin(); it != m.end(); it++) {
        std::cout << it->first << ": " << it->second << std::endl;
    }
    return 0;
}
```

map

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

```
int main() {
    std::map<int, std::string> m;
    m.insert(std::pair<int, std::string>(2017123456, "Joao"));
```

```
    m[2016123456] = "Maria";
    m[2018123456] = "Carlos";
    m[2015123456] = "Jose";
    m[2014123456] = "Joana";
```

```
    std::map<int, std::string>::iterator it;
    for (it = m.begin(); it != m.end(); it++)
        std::cout << it->first << ": " << it->second << std::endl;
    }
    return 0;
}
```

Tipo pair constrói tuplas (first, second)

Note a sintaxe de ponteiro no iterador

Criando sets de tipos diferentes

Precisamos saber comparar. Usamos um comparator

■ Suponha a
classe ao lado

```
class Estudante {  
    const std::string _nome;  
    int _matricula;  
  
public:  
    Estudante(std::string nome, int matricula):  
        _nome(nome), _matricula(matricula) {}  
  
    std::string get_nome() const {  
        return this->_nome;  
    }  
    int get_matricula() const {  
        return this->_matricula;  
    }  
};
```

Criando sets de tipos diferentes

Precisamos saber comparar. Usamos um comparator

■ Como podemos evitar a inserção de elementos duplicados?

```
int main() {  
    std::set<Estudante> turma;  
    turma.insert(Estudante("Ana", 20221978));  
    turma.insert(Estudante("Pedro", 20221990));  
    turma.insert(Estudante("Ana", 20221978));  
}
```

■ Erro de compilação

template argument deduction/substitution failed:


Comparator

Precisamos saber comparar. Usamos um comparator

■ O template set aceita um segundo parâmetro que recebe 2 valores a e b e retorna se $a < b$

```
struct compara_estudantes_f {  
    bool operator()(const Estudante& p1, const Estudante& p2) {  
        return p1.get_matricula() < p2.get_matricula();  
    }  
};
```

```
int main() {  
    std::set<Estudante, compara_estudantes_f> turma;  
    turma.insert(Estudante("Ana", 20221978));  
    turma.insert(Estudante("Pedro", 20221990));  
    turma.insert(Estudante("Ana", 20221978));  
}
```



Comparator

Precisamos saber comparar. Usamos um comparator

- O set usa essa função com dois objetivos
 - Saber onde encaixar novos elementos na árvore
 - Detectar duplicatas

$$p1 == p2 \iff !(p1 < p2) \ \&\& \ !(p2 < p1)$$

```
struct compara_estudantes_f {  
    bool operator()(const Estudante& p1, const Estudante& p2) {  
        return p1.get_matricula() < p2.get_matricula();  
    }  
};
```


Comparator

Vale igualmente para o map

std::set

```
template < class T,                                // set::key_type/value_type
    class Compare = less<T>,                       // set::key_compare/value_compare
    class Alloc = allocator<T>                    // set::allocator_type
> class set;
```

2° param →

std::map

```
template < class Key,                                // map::key_type
    class T,                                         // map::mapped_type
    class Compare = less<Key>,                     // map::key_compare
    class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
> class map;
```

3° param →

Comparator

Vale igualmente para o map

■ Note que o map ordena sua estrutura com base na chave. Seu comparator deve ser implementado com isso em mente.

std::map

```
template < class Key,                                // map::key_type
          class T,                                    // map::mapped_type
          class Compare = less<Key>,                 // map::key_compare
          class Alloc = allocator<pair<const Key,T> > // map::allocator_type
        > class map;
```

3° param →

Por fim...

Vimos muitas estruturas, basta saber onde aplicar!

- Vector
 - indexação por inteiros
 - inserção no final
- Deque (double ended queue)
 - indexação por inteiros
 - inserção no início ou no final
- List
 - caminhamento via iterator

Por fim...

Vimos muitas estruturas, basta saber onde aplicar!

- Set
 - recuperação rápida de itens
 - saída ordenada
- Map
 - recuperação rápida de itens
 - associação de tuplas (chave, valor)
 - saída ordenada pela chave

Outras bibliotecas de C++

<code><algorithm></code>	<code>copy()</code> , <code>find()</code> , <code>sort()</code>
<code><cmath></code>	<code>sqrt()</code> , <code>pow()</code>
<code><fstream></code>	<code>fstream</code> , <code>ifstream</code> , <code>ofstream</code>
<code><iostream></code>	<code>istream</code> , <code>ostream</code> , <code>cin</code> , <code>cout</code>
<code><memory></code>	<code>unique_ptr</code> , <code>shared_ptr</code>