Algoritmos e Estruturas de Dados I

http://ctp.di.fct.unl.pt/lei/aed1/

Segundo semestre, 2001-2002
Licenciatura em Engenharia Informática

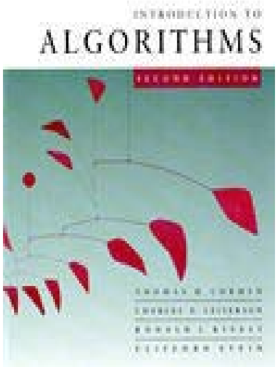
por
Pedro Guerreiro
pg@di.fct.unl.pt, http://ctp.di.fct.unl.pt/~pg
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal

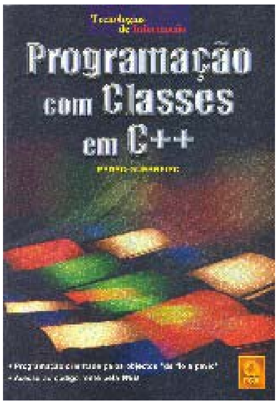
Plano

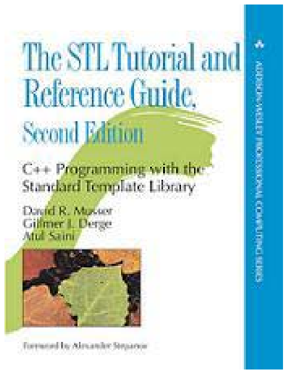
- Estruturas de dados fundamentais: listas, pilhas, filas, tabelas de dispersão, árvores, iteradores: 4 semanas.
- Ordenação e estatísticas de ordem: 2 semanas.
- Análise de algoritmos: 1 semana.
- Grafos: 2 semanas.
- Encontro de cadeias: 1 semana.
- Estratégias algorítmicas: programação dinâmica, algoritmos gananciosos, etc.: 1 semana.
- Operações com matrizes: 1 semana.
- Algoritmos numéricos: o resto do tempo...

Bibliografia Principal

- *Introduction to Algorithms* (2nd edition), Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein, 2001.
- Programação com Classes em C++, …, 2000.
- *STL Tutorial and Reference Guide*, David Musser, Gillmer Derge, Atul Saini, 2001.







Ingredientes

- Programação.
- Algoritmos.
- Estruturas de dados
- Programação orientada pelos objectos.
- C++.
- Programação genérica.
- Bibliotecas de classes.
- STL.
- Visual C++.
- Engenharia de Software.

Listas

- "Uma lista é uma sequência de um ou mais elementos na qual se podem *acrescentar elementos ou retirar elementos em qualquer posição*."
- "Uma lista é uma estrutura *sequencial* finita, na qual existe um primeiro elemento e um último elemento. Ser sequencial significa que pode ser atravessada do primeiro até ao último elemento."
- As listas são a estrutura ideal quando há muitas inserções e remoções em posições interiores e quando o acesso aos elementos é quase preferencialmente sequencial (por oposição a aleatório).

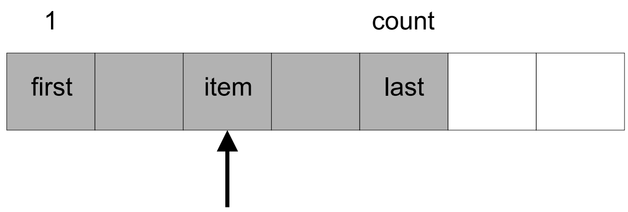
Variantes

- ARRAYED_LIST
- LINKED_LIST
- TWO_WAY_LIST
- MULTI_ARRAY_LIST
- SORTED_LIST
- SORTED_TWO_WAY_LIST

Classificação Eiffel.

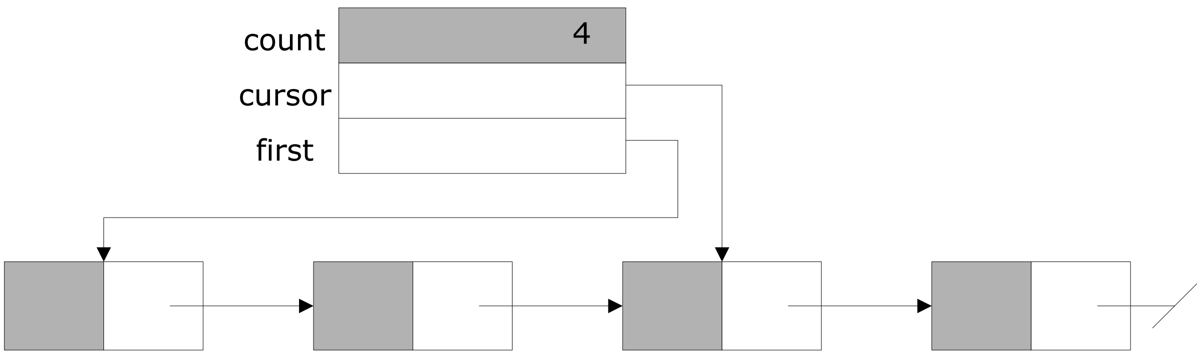
ARRAYED_LIST

- Implementadas por meio de um vector redimensionável.
- Boas se as inserções e remoções são no fim e se não é preciso crescer muitas vezes.



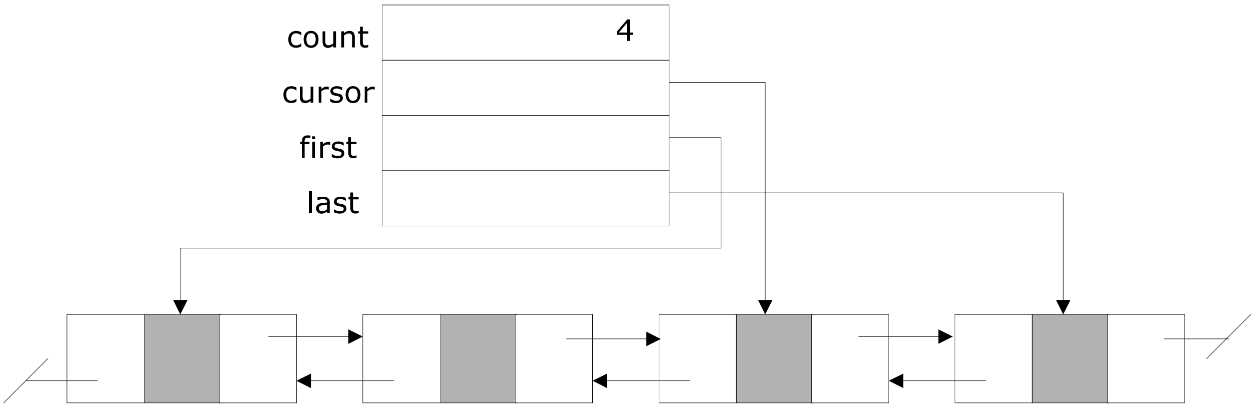
LINKED_LIST

- Implementadas por meio de listas ligadas, com apontadores e memória dinâmica.
- Muito boas se as inserções e remoções são à cabeça. Boas para inserir ou remover em qualquer posição



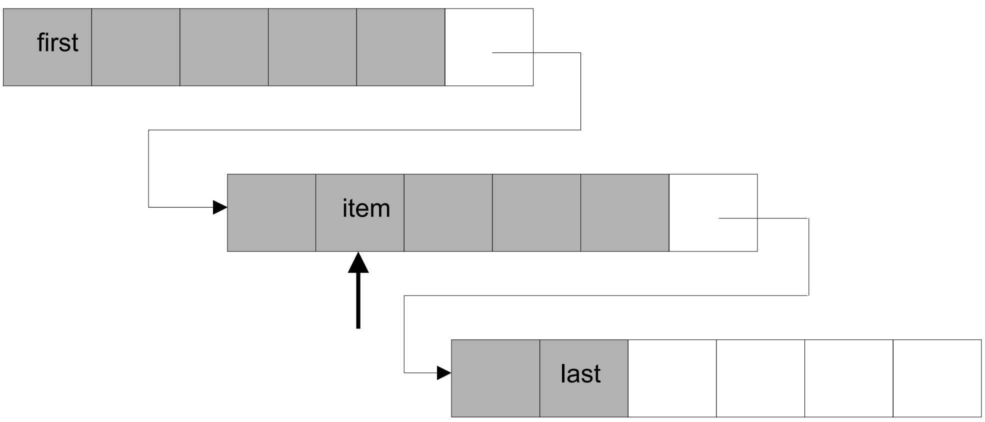
TWO_WAY_LIST

- Cada nó tem dois apontadores, um para o elemento seguinte outro para o elemento precedente.
- Boas quando é preciso percorrer a lista nos dois sentidos.

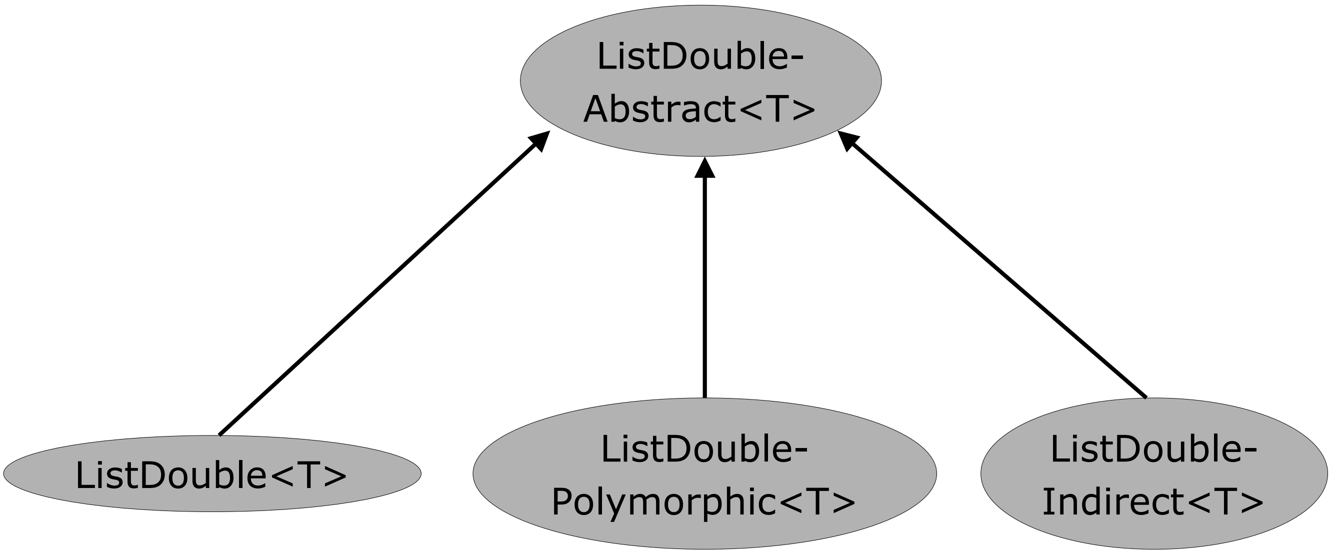


MULTI_ARRAY_LIST

- Ao crescer, não tem de duplicar a estrutura toda.



Listas Duplas Genéricas



Classe ListDoubleAbstract<T> (1)

```
template <class T>
class ListDoubleAbstract: public Clonable,
              public virtual Dispenser<T>,
              public virtual Collection<T>,
              public virtual Bilinear<T> {
// ...

//From Container<T>
  virtual void Put(const T& x);
  virtual int Count() const;
  virtual void Clear();

//From Dispenser<T>
  virtual const T& Item() const;
  virtual T& Item();
  virtual void Remove();

//From Collection<T>
  virtual bool Has(const T& x) const;
  virtual int CountIf(const T& x) const;
  virtual void Prune(const T& x);
  virtual void PruneAll(const T& x);
```

Classe ListDoubleAbstract<T> (2)

```
//From Linear<T>
  virtual void Reset();
  virtual void Start();
  virtual void Forth();
  virtual bool Off() const;

//From Bilinear<T>
  virtual void Finish();
  virtual void Back();

// Iterators
 virtual IteratorSmart<T> Items() const;
 virtual IteratorSmart<T> ItemsReverse() const;
```

Iteradores

Classe ListDoubleAbstract<T> (3)

```
//Declared now
  virtual const T& First() const;    // pre: !Empty();
  virtual const T& Last() const;     // pre: !Empty();
  virtual void RemoveFirst();        // pre: !Empty();
  virtual void RemoveLast();         // pre: !Empty();
  virtual void PutFirst(const T& x);
  virtual void PutLast(const T& x);
  virtual bool AtFirst() const;      // pre: !Off();
  virtual bool AtLast() const;       // pre: !Off();
  virtual void Search(const T& x);
  virtual void SearchForward(const T& x);
  virtual void SearchLast(const T& x);
  virtual void SearchBackward(const T& x);
  virtual void BringToFront();       // pre: !Off();
  virtual void SendToBack();         // pre: !Off();
  virtual void SwapWithNext();       // pre: !Off() && !AtLast();
  virtual void SwapWithPrevious();   // pre: !Off() && !AtFirst();
  virtual void Head();
  virtual void Tail();
  virtual void Replace(const T& x); // pre: !Off();
```

Classe ListDoubleAbstract<T> (4)

```
private:
  void Unlink(NodeDoubleAbstract<T>* p);
  void Link(NodeDoubleAbstract<T>* p1, NodeDoubleAbstract<T>* p2);

private:  // Factory method.
  virtual NodeDoubleAbstract<T>* NewNode(const T& x) const = 0;

// ...
};
```

Método de fábrica

Listas duplas polimórficas

```
template <class T>
class ListDoublePolymorphic: public ListDoubleAbstract<T> {
public:
  ListDoublePolymorphic();
  ListDoublePolymorphic(const ListDoublePolymorphic& other);
  virtual ~ListDoublePolymorphic();

  virtual Clonable* Clone() const;
  virtual void Copy(const ListDoublePolymorphic<T>& other);
  virtual const ListDoublePolymorphic& operator = (const ListDoublePolymorphic<T>& other);

private:
  virtual NodeDoubleAbstract<T>* NewNode(const T& x) const; // Factory method.
};
```

O Método de Fábrica

Cada classe derivada de ListDoubleAbstract<T> tem o seu:

```
template <class T>
NodeDoubleAbstract<T>* ListDoublePolymorphic<T>::NewNode(const T& x) const
{
  return new NodeDoublePolymorphic<T>(x);
}
```

É usado na função Put, que acrescenta um elemento à lista, na posição do cursor:

```
template <class T>
void ListDoubleAbstract<T>::Put(const T& x)
{
  Link(NewNode(x), cursor);
  count++;
}
```


Função de teste

Eis uma função main, que testa a classe genérica:

```
int main()
{
  std::cout << "Aqui vou eu..." << std::endl;
  ListDoublePolymorphic<int> list;
  list.Put(2);
  list.Put(3);
  list.Put(5);
  list.WriteLine();
  ListDoublePolymorphic<StringBasic> ls;
  ls.Put("aaa");
  ls.Put("bbb");
  ls.Put("ccc");
  ls.Put("ddd");
  ls.WriteLine();
  return 0;
}
```

Projectos com classes genéricas

Temos de juntar ao projecto os ficheiros de instanciação explícita:

```
#pragma warning (disable: 4661)

#include "CellAbstract.cpp"
template class CellAbstract<int>;
//...
#include "ListDouble.cpp"
template class ListDouble<int>;

#include "ListDoublePolymorphic.cpp"
template class ListDoublePolymorphic<int>;
```

Isto é parte do ficheiro Instantiations_ListDouble_int_.cpp, fornecido. Também faz falta o ficheiro Instantiations_Container_int_.cpp.

Os ficheiros das classes genéricas não precisam de estar no projecto. No entanto, aqueles sobre que estamos a trabalhar devem estar, para ficarem mais à mão.

Pilhas genéricas

Pilhas são distribuidores em que o elemento distinto é o que foi metido mais recentemente. (Disciplina LIFO: *last in, first out.*)

Há pilhas limitadas, com capacidade fixa (e finita), e pilhas ilimitadas, com capacidade "infinita".

```
template <class T>
class StackUnboundedPolymorphic: public Dispenser<T> {

// from Container<T>


// from Dispenser<T>


};
```

As pilhas limitadas implementam-se com vectores e as ilimitadas com listas.

StackUnboundedPolymorphic<T>

```
template <class T>
class StackUnboundedPolymorphic: public Dispenser<T> {
private:
  ListDoublePolymorphic<T> items;
public:
  StackUnboundedPolymorphic();
  virtual ~StackUnboundedPolymorphic();

// from Container<T>
  virtual void Put(const T& x);
  virtual int Count() const;
  virtual void Clear();

// from Dispenser<T>
  virtual const T& Item() const;
  virtual T& Item();
  virtual void Remove();

private:
  StackUnboundedPolymorphic<T>(const StackUnboundedPolymorphic<T>&){};
  void operator = (const StackUnboundedPolymorphic<T>&){};
};
```

Apenas aparecem as funções dos distribuidores: Put, Item e Remove (além das funções burocráticas: Clear, Count, Capacity (herdada).

O construtor de cópia e o operador de afectação estão bloqueados.

Implementação das pilhas genéricas

Vejamos as funções Put, Item e Remove:

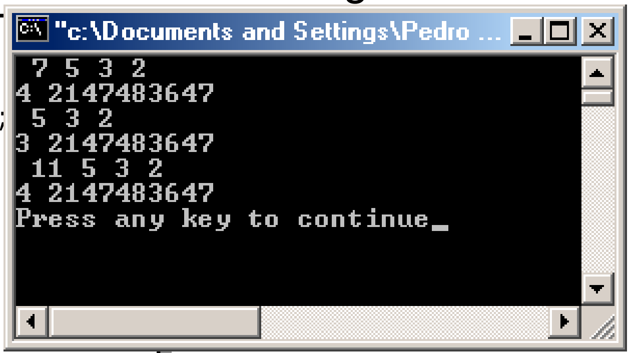
```
template <class T>
void StackUnboundedPolymorphic<T>::Put(const T& x)
{
  items.PutFirst(x);
}

template <class T>
const T& StackUnboundedPolymorphic<T>::Item() const
{
  return items.Item();
}

template <class T>
T& StackUnboundedPolymorphic<T>::Item()
{
  return items.Item();
}

template <class T>
void StackUnboundedPolymorphic<T>::Remove()
{
  items.RemoveFirst();
}
```

No folclore da Programação, estas funções têm o nome Push, Top e Pop. No entanto, os nomes Put, Item e Remove são mais apropriados.

Funções de teste (1)

Eis uma função de teste, para experimentar a classe genérica.

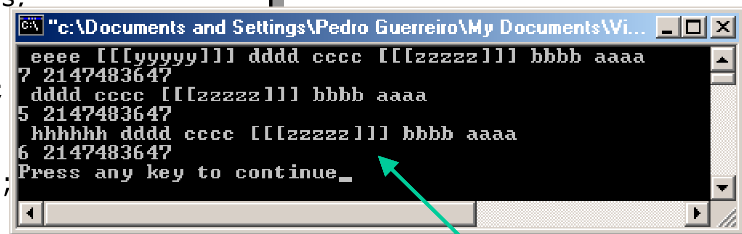
```
void TestStackint()
{
  ListDoublePolymorphic<int>::SetPrefixSuffix(" ", "");
  StackUnboundedPolymorphic<int> s;
  s.Put(2);
  s.Put(3);
  s.Put(5);
  s.Put(7);
  s.WriteLine();
  std::cout << s.Count() << " " << s.Capacity() << std::endl;
  s.Remove();
  s.WriteLine();
  std::cout << s.Count() << " " << s.Capacity() << std::endl;
  s.Put(11);
  s.WriteLine();
  std::cout << s.Count() << " " << s.Capacity() << std::endl;
}
```



Funções de teste (2)

E outra, agora para uma pilha de strings:

```
void TestStackString()
{
  ListDoublePolymorphic<StringBasic>::SetPrefixSuffix(" ", "");
  StackUnboundedPolymorphic<StringBasic> s;
  s.Put("aaaa");
  s.Put("bbbb");
  s.Put(StringBracketed("[[[", "zzzzz", "]]]"));
  s.Put("cccc");
  s.Put("dddd");
  s.Put(StringBracketed("[[[", "yyyyy", "]]]"));
  s.Put("eeee");
  s.WriteLine();
  std::cout << s.Count() << " " << s.Capacity() << std::endl;
  s.Remove();
  s.Remove();
  s.WriteLine();
  std::cout << s.Count() << " " << s.Capacity() << std::endl;
  s.Put("hhhhhh");
  s.WriteLine();
  std::cout << s.Count() << " " << s.Capacity() << std::endl;
}
```



Este exemplo exibe o polimorfismo

Filas genéricas

Filas são distribuidores em que o elemento distinto é o que foi metido há mais tempo. (Disciplina FIFO: *first in, first out.*)

Há filas limitadas e ilimitadas. Eis uma classe para filas limitadas não-polimórficas:

```
template <class T>
class QueueBounded: public Dispenser<T>{
private:
  T* items;
  int capacity;
  int count;
  int in;
  int out;
public:
  explicit QueueBounded(int capacity);
  ~QueueBounded();

// from Container<T>

// from Dispenser<T>
  virtual const T& Item() const;
};
```


Implementação das filas genéricas

Vejamos as funções Put, Item e Remove:

```
template <class T>
void QueueBounded<T>::Put(const T& s)
{
  count++;
  items[in] = s;
//  in = (in + 1) % capacity;
  ++in %= capacity;
}

template <class T>
void QueueBounded<T>::Remove()
{
  count--;
//  out = (out + 1) % capacity;
  ++out %= capacity;
}
```

```
template <class T>
const T& QueueBounded<T>::Item() const
{
  return items[out];
}

template <class T>
T& QueueBounded<T>::Item()
{
  return items[out];
}
```

!

Os elementos são acrescentados ao vector por afectação. Logo não há polimorfismo,

Filas com prioridade

Filas com prioridade são distribuidores em que o elemento distinto é o que tem maior *prioridade*. Tipicamente, a prioridade é uma função inteira dos elementos da fila.

```
template <class T>
class PriorityQueue: public Dispenser<T> {
// …
public:
  explicit PriorityQueue(int capacity);
  virtual ~PriorityQueue();

// from Container<T>
  virtual void Put(const T& x);
  virtual int Count() const;
  virtual void Clear();

// from Dispenser<T>
  virtual const T& Item() const;
  virtual T& Item();
  virtual void Remove();

// declared now
  virtual void Add(const T& s, int x = std::numeric_limits<int>::min());
};
```

A função Add é a única nova. O segundo argumento representa a prioridade do novo elemento. Por defeito, é mínima.

Implementação ingénua

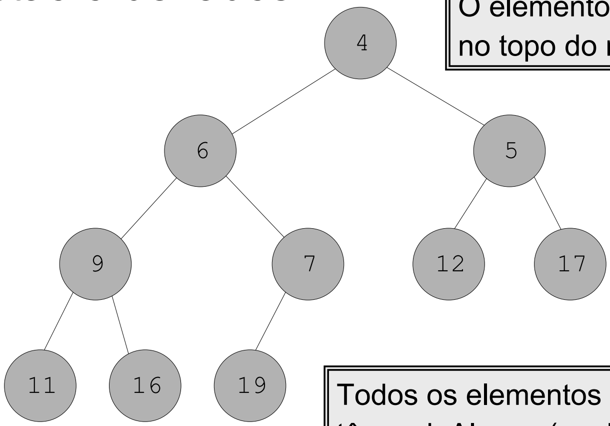
Uma implementação ingénua usaria uma lista para os elementos, tal como a classe StackUnbounded-Polymorphic<T>:

```
template <class T>
class PriorityQueue: public Dispenser<T> {
private:
  ListDoublePolymorphic<T> items;
public:
  // …
};
```

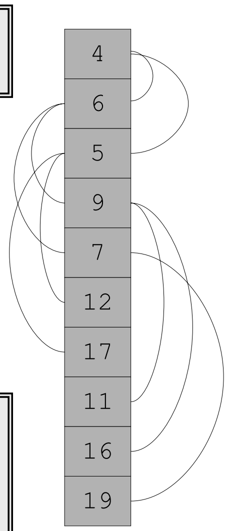
Das duas uma: ou ao acrescentar um novo elemento se insere ordenadamente, de maneira a que o primeiro é sempre o mais prioritário, ou se põe no fim da lista e então para aceder ao elemento mais prioritário é preciso determiná-lo. Inserir ordenadamente é uma operação de tempo linear e determinar o elemento mais prioritário numa lista não ordenada também.

Montes (isto é, *heaps*)

Um monte (em inglês *heap*) é um distribuidor implementado com um vector que representa uma árvore binária quase completa, tal que todos os caminhos da raiz até uma folha estão ordenados:



O elemento mínimo está no topo do monte.



Todos os elementos excepto o topo têm pai. Alguns (quais?) tem filho sinistro e alguns (quais?) além de filho sinistro têm filho destro.

Classe HeapPolymorphic<T>

```
template <class T>
class HeapPolymorphic: public Dispenser<T> {
private:
  VectorPolymorphic<T> items;
public:
  explicit HeapPolymorphic(int capacity);
  virtual ~HeapPolymorphic();

// from Container<T>
  virtual void Put(const T& x);
  virtual int Count() const;
  virtual void Clear();

// from Dispenser<T>
  virtual const T& Item() const;
  virtual T& Item();
  virtual void Remove();

private:
  virtual int Left(int i) const;
  virtual int Right(int i) const;
  virtual int Parent(int i) const;
};
```

Um monte tem uma certa capacidade inicial, mas pode crescer. O elemento distinto é o que está no topo. É esse que a função Item devolve e é esse que a função Remove remove.

Estas funções privadas implementam a relação dos índices no vector de um elemento e o seu pai, filho sinistro e filho destro.

Pai e Filhos

As funções Parent, Left e Right fazem umas contas simples:

```
template <class T>
int HeapPolymorphic<T>::Left(int i) const
{
  return 2*(i+1) - 1;
}

template <class T>
int HeapPolymorphic<T>::Right(int i) const
{
  return 2*(i+1);
}

template <class T>
int HeapPolymorphic<T>::Parent(int i) const
{
  return (i-1) / 2;
}
```

Antes de usar o resultado destas funções é preciso garantir que representa um índice válido no vector items.

Pôr no monte

Para acrescentar um elemento ao monte, primeiro acrescenta-se ao vector e depois faz-se esse elemento subir no monte até à sua posição, trocando com o pai, sempre que ele e pai estiverem fora de ordem. É a função privada Increase:

```
template <class T>
class HeapPolymorphic: public Dispenser<T> {
// ...

private:
// ...
  virtual void Increase(int x);
};
```

```
template <class T>
void HeapPolymorphic<T>::Put(const T& x)
{
  items.Extend(x);
  Increase(items.Count() - 1);
}
```

Subir o monte

Um elemento sobe o monte trocando de posição com o seu pai, quando for menor do que o pai.

```
template <class T>
void HeapPolymorphic<T>::Increase(int x)
{
  while (x > 0 && items[x] <= items[Parent(x)])
  {
    items.Swap(x, Parent(x));
    x = Parent(x);
  }
}
```

Recorde que a função Swap apenas troca os apontadores para os elementos no vector items. O vector items é implementado por um vector de apontadores. Por isso a função Swap não envolve a criação de novos objectos.

O número de trocas é da ordem do logaritmo na base 2 do número de elementos no monte.

No topo

O elemento distinto é o que está no topo, isto é, na posição de índice zero no vector:

```
template <class T>
const T& HeapPolymorphic<T>::Item() const
{
  return items[0];
}

template <class T>
T& HeapPolymorphic<T>::Item()
{
  return items[0];
}
```

Ao remover, vai ser preciso colocar na posição zero o novo elemento mínimo. Queremos fazer isso logaritmicamente também.

Tirar do monte

Para tirar do monte, a técnica é copiar para a primeira posição o último elemento do vector, eliminando-o da última posição e fazê-lo descer o monte até à sua posição de “equilíbrio”, trocando-o sucessivamente com o menor dos seus filhos, se for maior do que ele. É a função privada Decrease:

```
template <class T>
class HeapPolymorphic: public Dispenser<T> {
// ...

private:
// ...
  virtual void Decrease(int x);
};
```

```
template <class T>
void HeapPolymorphic<T>::Remove()
{
  items.Swap(0, Count()-1);
  items.Remove();
  Decrease(0);
}
```

Colocamos o último elemento na primeira posição por troca.

Descer o monte

A função Decrease é a mais subtil de todas:

```
template <class T>
void HeapPolymorphic<T>::Decrease(int x)
{
  int left = Left(x);
  int right = Right(x);
  int smallest = x;
  if (left < Count() && items[left] <= items[smallest])
    smallest = left;
  if (right < Count() && items[right] <= items[smallest])
    smallest = right;
  if (smallest != x)
  {
    items.Swap(x, smallest);
    Decrease(smallest);
  }
}
```

Chamada recursiva, para continuar a descer.

O número de trocas é da ordem do logaritmo do número de elementos no monte, tal como na subida.

Prioritização

Para usar montes na implementação filas com prioridade de elementos de tipo T temos de *prioritizar* o tipo T.

```
template <class T>
class Prioritized: public T {
private:
  int priority;
public:
  Prioritized();
  Prioritized(const T& x, int value);
  Prioritized(const Prioritized<T>& other);
  virtual ~Prioritized();

  virtual Clonable* Clone() const;

  virtual void SetPriority(int priority);
  virtual int Priority() const;
  virtual bool operator <= (const Prioritized<T>& other) const;
};
```

A prioridade é uma função inteira.

Esta é a relação de ordem que vai ser usada no monte. O mínimo de acordo com esta relação é o mais prioritário.

Mais prioritário do que

Um elemento prioritizado é *menor* no monte se a sua prioridade for maior:

```
template <class T>
bool Prioritized<T>::operator <= (const Prioritized<T>& other) const
{
  return priority >= other.priority;
}
```

Filas prioritizadas

A estrutura que implementa uma fila com prioridade de elementos de tipo T é um monte de elementos de tipo Prioritized<T>:

```
template <class T>
class PriorityQueue: public Dispenser<T> {
private:
  HeapPolymorphic<Prioritized<T> > items;
public:
  explicit PriorityQueue(int capacity);
  virtual ~PriorityQueue();

// from Container<T>

// from Dispenser<T>

// declared now
  virtual void Add(const T& s, int x = std::numeric_limits<int>::min());
};
```

Recorde que o espaço entre os dois sinais > é indispensável.

Implementação da fila com prioridade

É deveras simples:

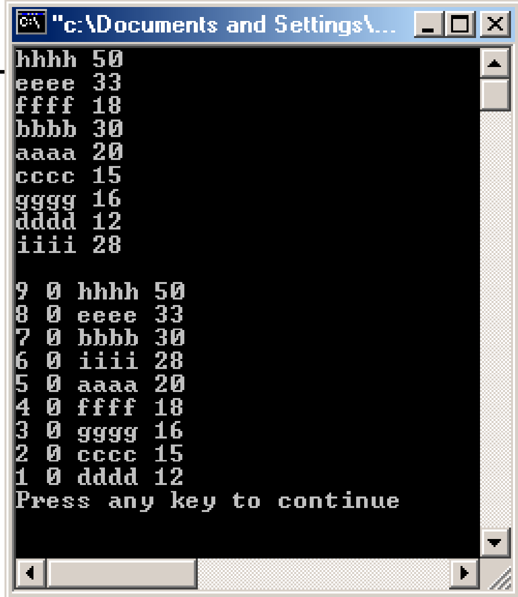
```
template <class T>
void PriorityQueue<T>::Put(const T& s)
{
  items.Put(Prioritized<T>(s, std::numeric_limits<int>::min()));
}

template <class T>
void PriorityQueue<T>::Add(const T& s, int x)
{
  items.Put(Prioritized<T>(s, x));
}
```

```
template <class T>
const T& PriorityQueue<T>::Item() const
{
  return items.Item();
}

template <class T>
T& PriorityQueue<T>::Item()
{
  return items.Item();
}

template <class T>
void PriorityQueue<T>::Remove()
{
  items.Remove();
}
```

Tudo se faz automaticamente através do monte.

Testando com Strings

Eis a função de teste:




```
void TestPriorityQueueGeneric()
{
  PriorityQueue<StringBasic> q(20);
  q.Add("aaaa", 20);
  q.Add("bbbb", 30);
  q.Add("cccc", 15);
  q.Add("dddd", 12);
  q.Add("eeee", 33);
  q.Add("ffff", 18);
  q.Add("gggg", 16);
  q.Add("hhhh", 50);
  q.Add("iiii", 28);
  q.WriteLine();
  while (!q.Empty())
  {
    std::cout << q.Count() << " "<< q.Empty() << " " << q.Item() << std::endl;
    q.Remove();
  }
}
```


Testando com Pessoas

Uma pessoa tem nome e idade. A importância de uma pessoa é o produto da idade pelo comprimento do nome. Queremos uma fila onde a prioridade é a importância. Eis a classe Person:

```
class Person: public Clonable {
private:
  StringBasic name;
  int age;
public:
  Person(const StringBasic& name, int age);
  Person(const Person& other);
  virtual ~Person();

  virtual Clonable* Clone() const;

  virtual const StringBasic& Name() const;
  virtual int Age() const;
  virtual int Importance() const;

  // ...
```

```
int Person::Importance() const
{
  return age * name.Count();
}
```

Especializando a função Put

Para ter uma fila com prioridade usando a função Importance como critério, basta especializar a função Put no ficheiro de instanciação genérica para classe PriorityQueue<Person>:

```
#include <iostream>
#include "Clonable.h"
#include "StringBasic.h"
#include "Person.h"

#include "PriorityQueue.cpp"
template class PriorityQueue<Person>;

void PriorityQueue<Person>::Put(const Person& x)
{
  Add(x, x.Importance());
}
```


Fila de pessoas por importância

Eis a função de teste:

```
void TestPriorityQueuePerson()
{
  PriorityQueue<Person> q(3);
  q.Put(Person("Filipa", 26));
  q.Put(Person("Vera", 21));
  q.Put(Person("Pedro", 49));
  q.Put(Person("Teresa", 43));
  q.Put(Person("Rui", 41));
  q.Put(Person("Diana", 17));
  q.Put(Person("Manuela", 55));
  q.Put(Person("Catarina", 15));
  q.Put(Person("Miguel", 47));
  while (!q.Empty())
  {
    std::cout << q.Count() << " " << q.Item() << std::endl;
    q.Remove();
  }
}
```

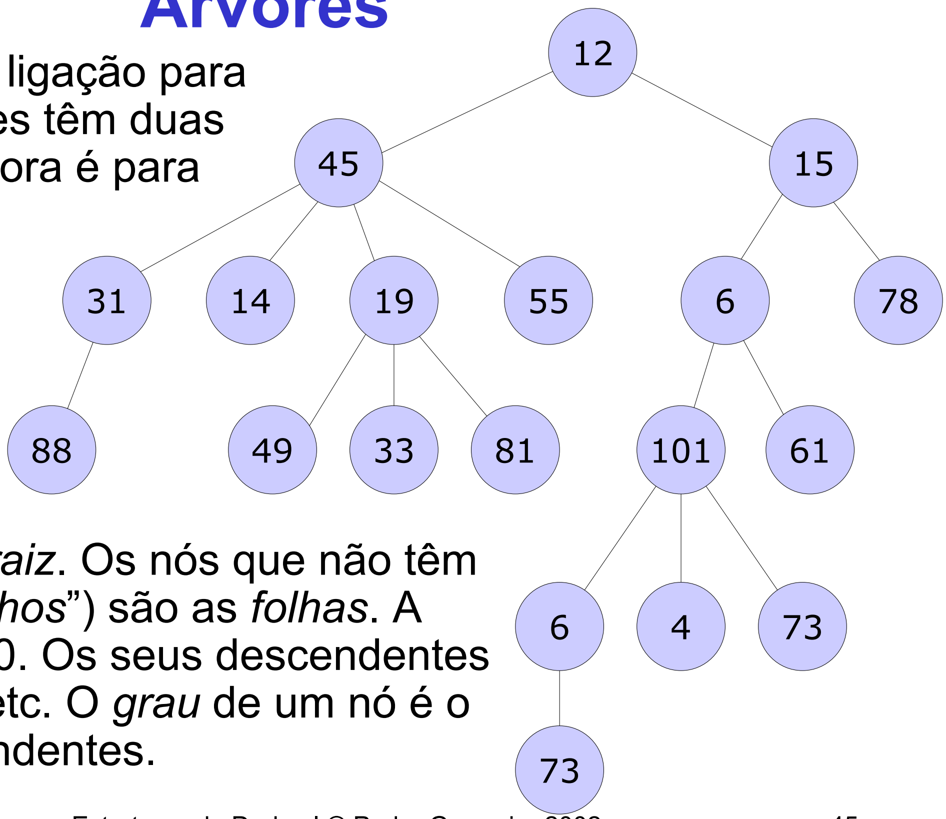
A capacidade inicial é 3, para teste. Assim a fila tem de crescer algumas vezes.

```
"c:\Documents and Settings\...
9 Manuela 55 385
8 Miguel 47 282
7 Teresa 43 258
6 Pedro 49 245
5 Filipa 26 156
4 Rui 41 123
3 Catarina 15 120
2 Diana 17 85
1 Vera 21 84
Press any key to continue
```

Árvores

As listas têm uma ligação para a frente. As árvores têm duas (ou mais), mas agora é para baixo (!).

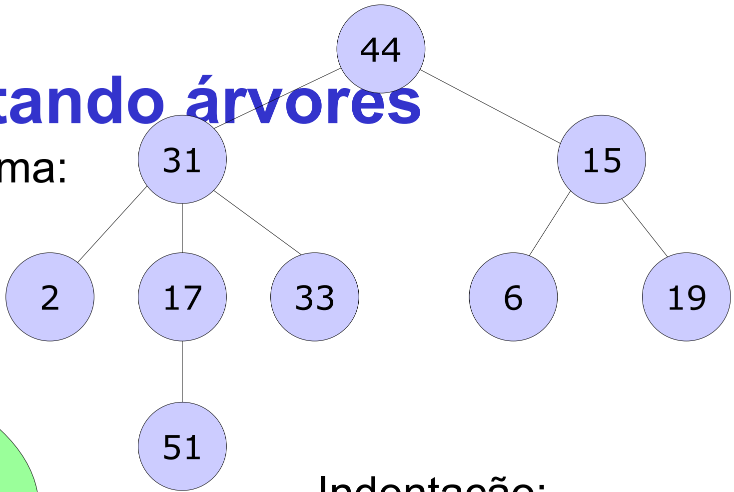
> *I think that I shall never see*
> *A poem lovely as a tree.*
>
> Joyce Kilmer (1913), citado em Knuth, *Fundamental Algorithms*.



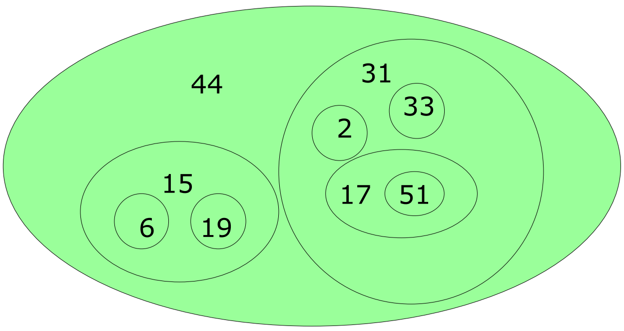
O nó com 12 é a *raiz*. Os nós que não têm descendentes (“*filhos*”) são as *folhas*. A raiz está no nível 0. Os seus descendentes estão no nível 1, etc. O *grau* de um nó é o número de descendentes.

Representando árvores

Diagrama:



Conjuntos encaixados:



Indentação:

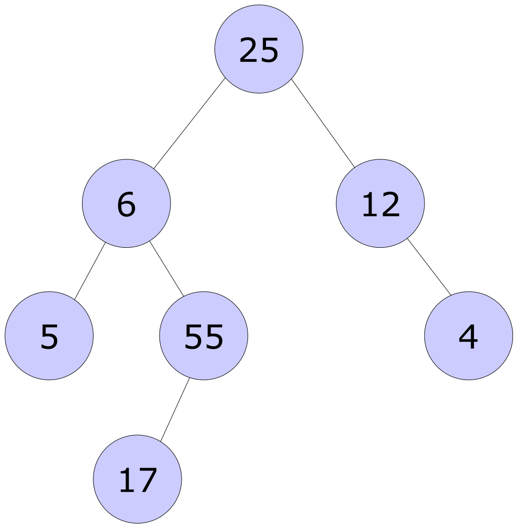
```
44
  31
    2
    17
      51
    33
  15
    6
    19
```

Expressão parentética:

(44(31(2 17(51) 33) 15(6 19))

Árvores binárias

Nas árvores binárias, cada nó tem um filho esquerdo e um filho direito, cada um dos quais ou os dois podem faltar.



O nó 12 só tem filho direito e o nó 55 só tem filho esquerdo.

Definições formais

Uma *árvore* é um conjunto finito *T* não vazio de *nós*, tal que:

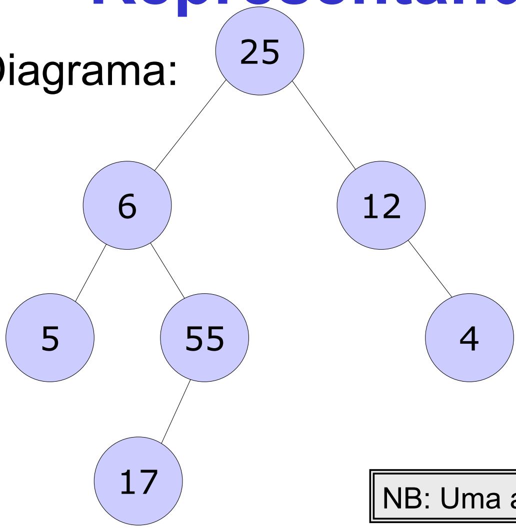
- Há um nó distinto chamado a *raiz* de *T*.
- Os restantes nós estão particionados em m>0 conjuntos disjuntos, T_1, T_2, ..., T_m, cada um dos quais é uma árvore. As árvores T_1, T_2, ..., T_m são as *subárvores* da raiz.

Uma *árvore binária* um conjunto finito de *nós* que ou é vazio ou é formado por um nó distinto chamado *raiz* e por duas árvores binárias, chamadas *subárvores* da raiz.

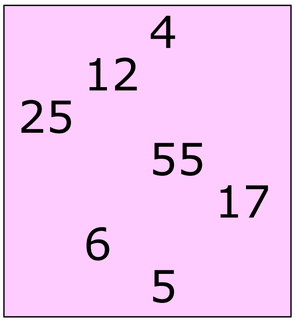
Logo, as árvores binárias não são árvores !!??.

Representando árvores binárias

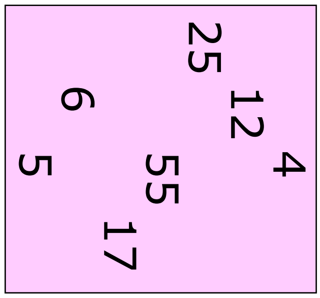
Diagrama:



Indentação:



Indentação, rodado:



NB: Uma árvore binária ou é vazia ou tem raiz e duas subárvores

Expressão parentética:

Representando as subárvores vazias por *...

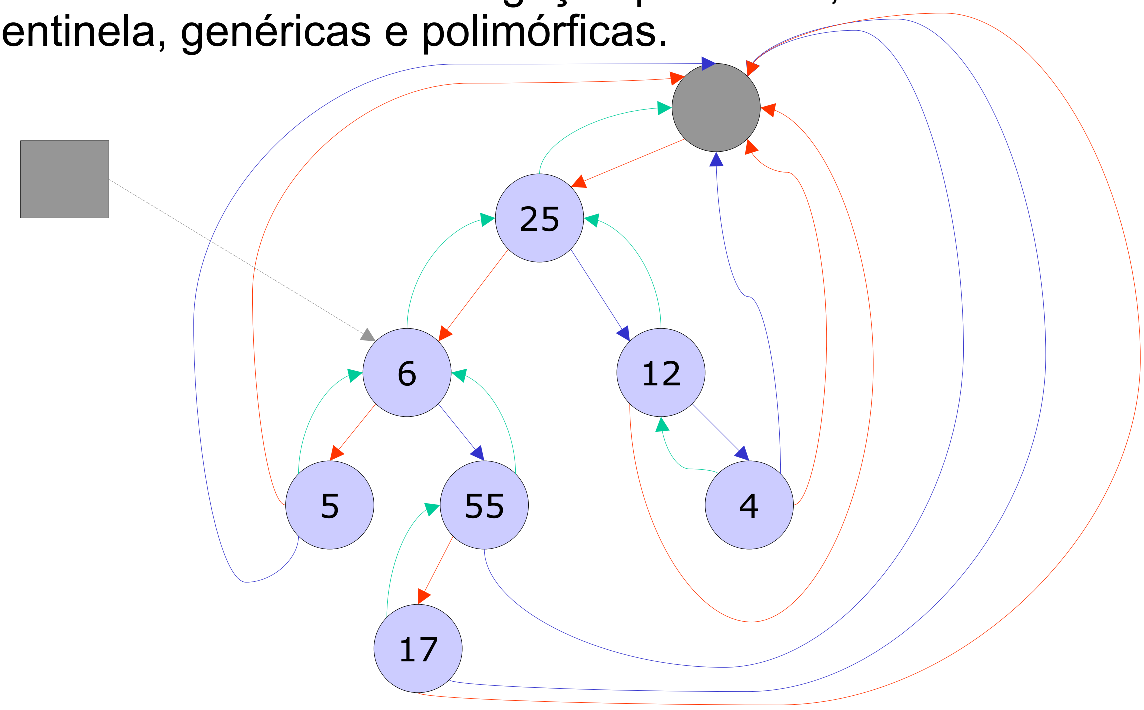
(25(6(5(**)55(17(**)*))12(*4(**))))

... e representado-as por ().

(25(6(5(()())55(17(()())()))12(()4(()()))))

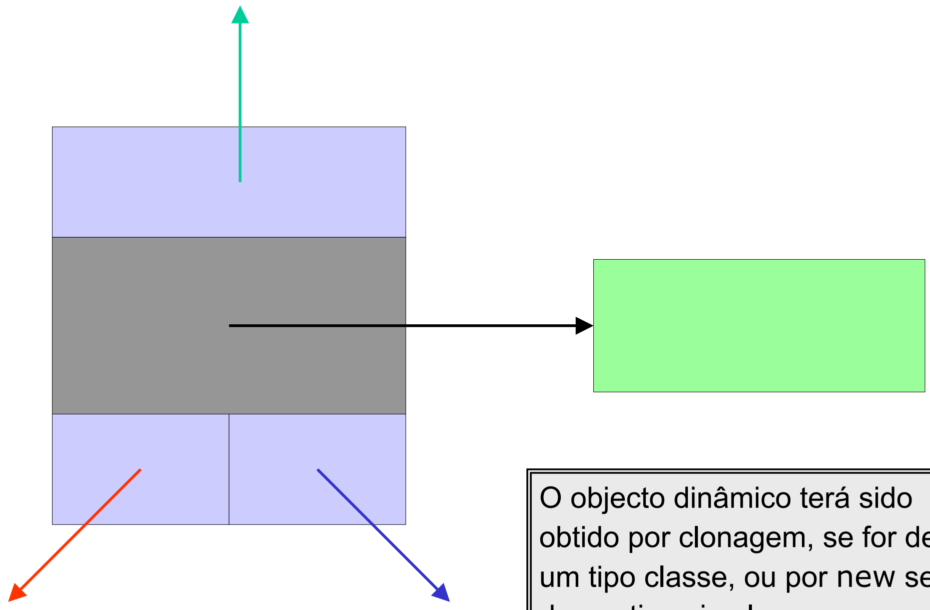
Implementação das árvores binárias

Vamos usar árvores com ligação para cima, com cursor e com sentinela, genéricas e polimórficas.



Nós arborescentes polimórficos

Têm dois apontadores para baixo, um para cima e um para o objecto dinâmico:



O objecto dinâmico terá sido obtido por clonagem, se for de um tipo classe, ou por new se for de um tipo simples.

Classe TreeNodePolymorphic<T> (1)

```
template <class T>
class TreeNodePolymorphic {
private:
  T *item;
  TreeNodePolymorphic<T>* child[2];
  TreeNodePolymorphic<T>* parent;
public:
  explicit TreeNodePolymorphic<T>(const T& other = T());
  TreeNodePolymorphic(const TreeNodePolymorphic<T>& other);
  virtual ~TreeNodePolymorphic<T>();

  virtual void Put(const T& item);
  virtual T& Item();
  virtual const T& Item() const;

  virtual bool operator == (const TreeNodePolymorphic<T>& other) const;

// ...
};
```

Sim, um vector de dois apontadores.

O construtor de cópia é necessário por causa da genericidade. Mais tarde vamos ter vectores de nós, e os vectores usam uma classe genérica Equality<T> para a igualdade no tipo T. Esta classe necessita do construtor de cópia do tipo T.

A igualdade de nós é necessária pela mesma razão.

Classe TreeNodePolymorphic<T> (2)

```
  // ...
  virtual TreeNodePolymorphic<T>* Child(bool right) const;
  virtual TreeNodePolymorphic<T>* Left() const;
  virtual TreeNodePolymorphic<T>* Right() const;
  virtual TreeNodePolymorphic<T>* Parent() const;
  virtual void SetChild(bool right, TreeNodePolymorphic<T>* other);
  virtual void SetLeft(TreeNodePolymorphic<T>* other);
  virtual void SetRight(TreeNodePolymorphic<T>* other);
  virtual void SetParent(TreeNodePolymorphic<T>* other);

  static void SwapItems(TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2);

private:
  virtual void Put(T* item);  // pre this->item != item;

private: // blocked
  virtual void operator = (const TreeNodePolymorphic<T>& other){};
};
```

O filho falso é o esquerdo e o verdadeiro é o direito.

Para trocar os itens de dois nós polimórficos basta trocar os apontadores.

Tal e qual como nas listas...

Implementação TreeNode...<T> (1)

```
template <class T>
TreeNodePolymorphic<T>::TreeNodePolymorphic(const T& s):
  item(dynamic_cast<StringBasic *>(s.Clone())),
  parent(0)
{
  child[0] = child[1] = 0;
}

template <class T>
TreeNodePolymorphic<T>::~TreeNodePol
{
  delete item;
}
```

Um nó é criado com os três apontadores a zero e o item a apontar para um clone do argumento. Esta função terá de ser especializada para tipos simples.

```
template <class T>
void TreeNodePolymorphic<T>::Put(const T& item)
{
  Put(dynamic_cast<StringBasic *>(item.Clone()));
}

template <class T>
void TreeNodePolymorphic<T>::Put(T* item)
{
  delete this->item;
  this->item = item;
}
```

NB: ao redefinir o item, temos de apagar o objecto corrente.

Implementação TreeNode...<T> (2)

```
template <class T>
T& TreeNodePolymorphic<T>::Item()
{
  return *item;
}

template <class T>
const T& TreeNodePolymorphic<T>::Item() const
{
  return *item;
}
```

Versão const e versão não const.

```
template <class T>
void TreeNodePolymorphic<T>::SwapItems
                    (TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2)
{
  T* m = p1->item;
  p1->item = p2->item;
  p2->item = m;
}
```

Troca de apontadores.

Esta função é estática.

Implementação TreeNode...<T> (3)

```
template <class T>
TreeNodePolymorphic<T>* TreeNodePolymorphic<T>::Child(bool right) const
{
  return child[right];
}

template <class T>
TreeNodePolymorphic<T>* TreeNodePolymorphic<T>::Parent() const
{
  return parent;
}

template <class T>
void TreeNodePolymorphic<T>::SetChild(bool right, TreeNodePolymorphic<T>* other)
{
  child[right] = other;
}

template <class T>
void TreeNodePolymorphic<T>::SetParent(TreeNodePolymorphic<T>* other)
{
  parent = other;
}
```

Estas são todas muito simples. Ainda há as Left, Right, SetLeft,SetRight, que são parecidas.

Classe TreePolymorphic<T> (1)

```
template <class T>
class TreePolymorphic: public Container<T>,  public Bilinear<T>{
private:
  TreeNodePolymorphic<T> sentinel;
  TreeNodePolymorphic<T>* cursor;
  int count;
public:
  TreePolymorphic<T>();
  virtual ~TreePolymorphic<T>();

// from Container<T>
  virtual void Put(const T& x);
          // post "The cursor points to the newly inserted node.";
  virtual int Count() const;
  virtual void Clear();

// ...

private:  // blocked
  TreePolymorphic(const TreePolymorphic<T>& other){};
  virtual void operator = (const TreePolymorphic<T>& other){};
};
```

O filho esquerdo da sentinela aponta para a raiz da árvore.

Uma árvore é um contentor. Podemos lá pôr objectos de tipo T (por clonagem, já sabemos...). É uma estrutura bilinear: logo pode ser percorrida do princípio para o fim, e do fim para o princípio. Mas como?

Estas são as redefinidas. Ainda há as herdadas (Empty, etc.)

O construtor de cópia e o operador de afectação estão bloqueados.

Classe TreePolymorphic<T> (2)

```
// from Linear<T>
  virtual void Reset();  // post Off();
  virtual void Start();
  virtual void Forth();
  virtual bool Off() const;

// from Bilinear<T>
  virtual void Finish();
  virtual void Back();
// ...
```

Com estas funções podemos visitar todos os nós da árvores. Mas por que ordem?

Percurso directo:

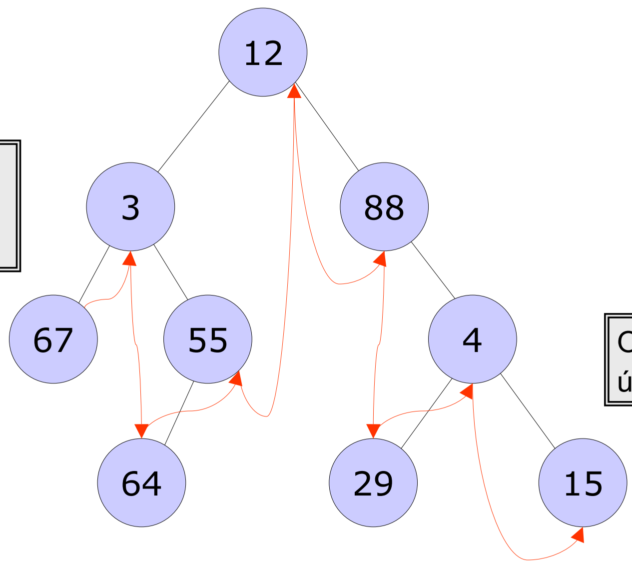
```
TreePolymorphic<StringBasic> t;
// ...
for (t.Start(); !t.Off(); t.Forth())
  std::cout << " " << t.Item();
std::cout << std::endl;
```

Percurso reverso:

```
TreePolymorphic<StringBasic> t;
// ...
for (t.Finish(); !t.Off(); t.Back())
  std::cout << " " << t.Item();
std::cout << std::endl;
```

Percorrendo a árvore

No percurso directo, percorre-se recursivamente a subárvore esquerda, depois visita-se a raiz e depois percorre-se recursivamente a subárvore direita:



O 67 é o primeiro nó.

Percurso directo:

67 3 64 55 12 88 29 4 15

Percurso reverso:

15 4 29 88 12 55 64 3 67

O 12 é o último nó

O percurso directo também é chamado percurso prefixo e o percurso reverso também é chamado percurso posfixo.

Classe TreePolymorphic<T> (3)

```
// …
virtual const T& Item() const;
virtual T& Item();

virtual const T& Root() const;   // pre !Empty();
virtual T& Root();   // pre !Empty();

virtual void Insert(const T& x, bool right1, bool right2 = false);
        // inserts a new node with item x as the right1 child of the node pointed by the cursor
        // linking the current right1 child as the right2 child of the new node.
        // post "The cursor points to the newly inserted node.";
virtual void Remove(); // pre !Off() && CountChildren() <= 1;
                              // post "the cursor points to the parent of the removed node.";
// …
```

A função Item devolve o elemento que está no nó apontado pelo cursor.

Ao inserir, é preciso indicar de que lado se insere e de que lado do nó inserido se pendura a subárvore que ficou “solta”. Não se pode apagar um nó com dois filhos, pois uma das subárvores ficaria “solta”.

Classe TreePolymorphic<T> (4)

Estas são simples:

```
// ...
virtual bool Has(const T& x) const;
virtual int CountIf(const T& x) const;

virtual void Search(const T& x);
virtual void Replace(const T& x); // pre !Off();

virtual int CountChildren() const;
virtual bool HasChild(bool right) const;
virtual bool HasLeft() const;
virtual bool HasRight() const;
virtual bool AtRoot() const;
virtual bool AtFirstLevel() const;  // pre: !AtRoot();
virtual bool Side() const;
virtual bool SideParent() const;  // pre: !AtRoot();
// ...
```

A função Side dá true se o nó apontado pelo cursor for um filho direito. A função SideParent dá true se o pai do nó apontado pelo cursor for um filho direito

Classe TreePolymorphic<T> (5)

Mais funções para mover o cursor:

```
// ...
virtual void Up(); // pre !Off();
virtual void Left();
virtual void Right();
virtual void Down(bool right);
// ...
```

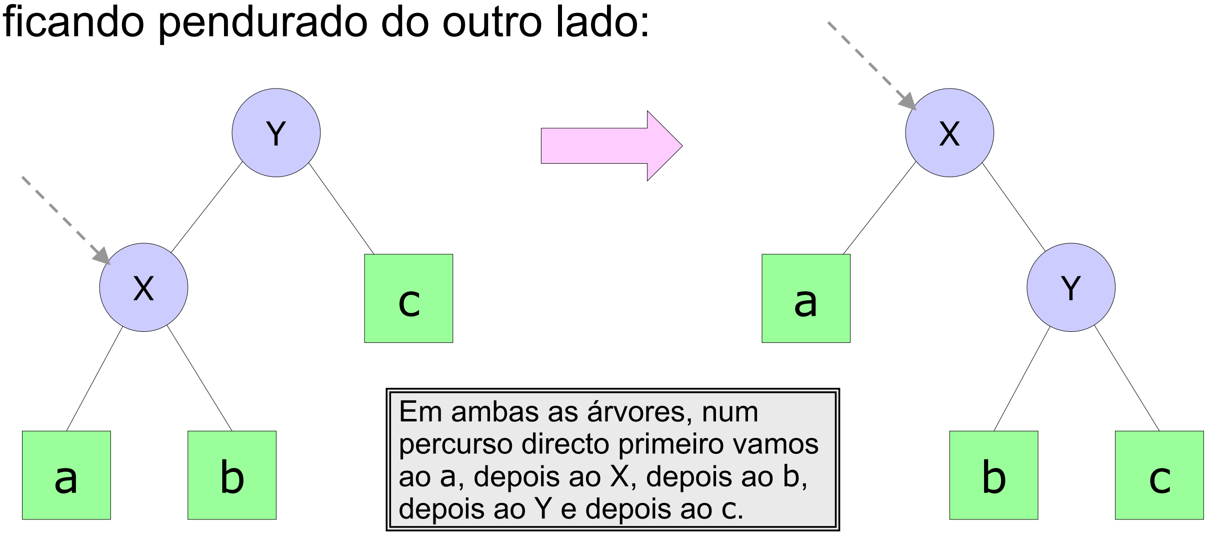
Promover um nó é subi-lo de nível, fazendo descer o pai pelo lado contrário:

```
// ...
virtual void Promote();
// ...
```

A função Promote promove o nó apontado pelo cursor.

Promoção

O nó apontado pelo cursor sobe um nível e o seu pai desce, ficando pendurado do outro lado:



Em ambas as árvores, num percurso directo primeiro vamos ao a, depois ao X, depois ao b, depois ao Y e depois ao c.

Neste caso, há uma rotação para a direita Y. Se, na figura do lado direito, o cursor estivesse a apontar para Y, a promoção daria a figura da esquerda, através de uma rotação para a esquerda. (Quem roda é a ligação XY.)

Classe TreePolymorphic<T> (6)

Funções de escrita:

```
  // ...
  virtual void Write(std::ostream& output = std::cout) const;
  virtual void WriteLine(std::ostream& output = std::cout) const;
  friend std::ostream& operator << (std::ostream& output, const TreePolymorphic<T>& w);
  virtual void WriteIndented(const StringBasic& indent = "\t", std::ostream& output = std::cout)
    const;

public: // static
  static void SetPrefixSuffix(const StringBasic& newPrefix, const StringBasic& newSuffix);
  static const StringBasic& Prefix();
  static const StringBasic& Suffix();

private: // static
  static StringBasic prefix;
  static StringBasic suffix;
  // ...
```

Este esquema dos prefixos e dos sufixos já é conhecido: cada elemento da árvore é escrito entre o prefixo e o sufixo.

Escrevendo a árvore

```
template <class T>
void TreePolymorphic<T>::Write(std::ostream& output) const
{
  WriteHere(output, sentinel.Left());
}

template <class T>
void TreePolymorphic<T>::WriteHere
                    (std::ostream& output, const TreeNodePolymorphic<T>* p) const
{
  output << prefix;
  if (p != &sentinel)
  {
    output << p->Item();
    WriteHere(output, p->Left());
    WriteHere(output, p->Right());
  }
  output << suffix;
}

template <class T>
void TreePolymorphic<T>::WriteLine(std::ostream& output) const
{
  Write(output);
  output << std::endl;
}
```

Função WriteHere, auxiliar (privada), recursiva. Primeiro escreve o prefixo, depois, se não for uma subárvore vazia, a raiz, depois recursivamente a subárvore esquerda e depois recursivamente a subárvore direita e por fim o sufixo.

Escrevendo indentadamente

```
template <class T>
void TreePolymorphic<T>::WriteIndented(const StringBasic& indent, std::ostream& output) const
{
  WriteIndentedHere(output, sentinel.Left(), "", indent);
}

template <class T>
void TreePolymorphic<T>::WriteIndentedHere
        (std::ostream& output, const TreeNodePolymorphic<T>* p,
          const StringBasic& margin, const StringBasic& indent) const
{
  if (p != &sentinel)
  {
    WriteIndentedHere(output, p->Right(), margin + indent, indent);
    output << margin << p->Item() << std::endl;
    WriteIndentedHere(output, p->Left(), margin + indent, indent);
  }
}
```

Primeiro escreve-se a subárvore direita, com a indentação, depois a raiz, depois a subárvore esquerda. Assim, quando rodarmos 90º no sentido do ponteiro dos relógios, a subárvore direita fica à direita.

Mais funções recursivas (1)

Existe na árvore?

```
template <class T>
bool TreePolymorphic<T>::Has(const T& x) const
{
  return HasHere(sentinel.Left(), x);
}

template <class T>
bool TreePolymorphic<T>::HasHere(const TreeNodePolymorphic<T>* p, const T& x) const
{
  if (p == &sentinel)
    return false;
  else
    return p->Item() == x || HasHere(p->Left(), x) || HasHere(p->Right(), x);
}
```

Um valor existe na árvore se for igual à raiz ou se existir na subárvore esquerda ou se existir na subárvore direita.

Mais funções recursivas (2)

Contando recursivamente os nós com um certo valor:

```
template <class T>
int TreePolymorphic<T>::CountIf(const T& x) const
{
  return CountIfHere(sentinel.Left(), x);
}

template <class T>
int TreePolymorphic<T>::CountIfHere(const TreeNodePolymorphic<T>* p, const T& x) const
{
  if (p == &sentinel)
    return 0;
  else
    return (p->Item() == x) + CountIfHere(p->Left(), x) + CountIfHere(p->Right(), x);
}
```

Palavras para quê?

Procurando recursivamente

Procura-se na raiz e, não encontrando, depois nas subárvores:

```
template <class T>
void TreePolymorphic<T>::Search(const T& x)
{
  cursor = &sentinel;
  SearchHere(sentinel.Left(), x);
}

template <class T>
void TreePolymorphic<T>::SearchHere(TreeNodePolymorphic<T>* p, const T& x)
{
  if (p == &sentinel)
    return;
  if (p->Item() == x)
    cursor = p;
  if (Off())
    SearchHere(p->Left(), x);
  if (Off())
    SearchHere(p->Right(), x);
}
```

Ao encontrar, posicionamos o cursor.

Só continuamos a procurar se não tivermos encontrado.

Apagando recursivamente

Apagam-se as subárvores e depois a raiz:

```
template <class T>
void TreePolymorphic<T>::Clear()
{
  ClearHere(sentinel.Left());
  sentinel.SetLeft(&sentinel);
  sentinel.SetRight(&sentinel);
  sentinel.SetParent(&sentinel);
  cursor = &sentinel;
  count = 0;
}

template <class T>
void TreePolymorphic<T>::ClearHere(TreeNodePolymorphic<T>* p)
{
  if (p == &sentinel)
    return;
  ClearHere(p->Left());
  ClearHere(p->Right());
  delete p;
}
```

Apagamos os nós todos, e depois acertamos a sentinela, o cursor e o contador.

Todas estas funções recursivas são privadas. Elas trabalham com apontadores e não queremos apontadores na interface da classe.

Apagamos as subárvores e depois a raiz (por esta ordem, claro).

Operações sobre o cursor

As que não movem o cursor são deveras elementares:

```
template <class T>
int TreePolymorphic<T>::CountChildren() const
{
  return CountChildren(cursor);
}

template <class T>
bool TreePolymorphic<T>::HasChild(bool right) const
{
  return cursor->Child(right) != &sentinel;
}

template <class T>
bool TreePolymorphic<T>::HasLeft() const
{
  return HasChild(0);
}

template <class T>
bool TreePolymorphic<T>::HasRight() const
{
  return HasChild(1);
}
```

```
template <class T>
bool TreePolymorphic<T>::AtRoot() const
{
  return cursor->Parent() == &sentinel;
}

template <class T>
bool TreePolymorphic<T>::AtFirstLevel() const
{
  return cursor->Parent()->Parent() == &sentinel;
}

template <class T>
bool TreePolymorphic<T>::Side() const
{
  return Side(cursor);
}

template <class T>
bool TreePolymorphic<T>::SideParent() const
{
  return Side(cursor->Parent());
}
```

A função Side interna tem um argumento de tipo TreeNodePoly...<T>*.

Movimentação simples do cursor

Para cima, para a esquerda, para a direita, para baixo:

```
template <class T>
void TreePolymorphic<T>::Up()
{
  cursor = cursor->Parent();
}

template <class T>
void TreePolymorphic<T>::Left()
{
  cursor = cursor->Left();
}

template <class T>
void TreePolymorphic<T>::Right()
{
  cursor = cursor->Right();
}

template <class T>
void TreePolymorphic<T>::Down(bool right)
{
  cursor = cursor->Child(right);
}
```

Estar Off, fazer o Reset:

```
template <class T>
bool TreePolymorphic<T>::Off() const
{
  return cursor == &sentinel;
}

template <class T>
void TreePolymorphic<T>::Reset()
{
  cursor = &sentinel;
}
```

Ir para o primeiro nó, ir para o último

Para chegar ao primeiro nó num percurso directo, vira-se sempre à esquerda, enquanto for possível:

```
template <class T>
void TreePolymorphic<T>::Start()
{
  Reset();
  while (HasLeft())
    Left();
}
```

Repare no primeiro Left na função Finish. É necessário porque a raiz está pendurada do lado esquerdo da sentinela.

Para ir para o último nó do percurso directo, que é o primeiro do percurso reverso, vira-se sempre à direita a partir da raiz, enquanto for possível:

```
template <class T>
void TreePolymorphic<T>::Finish()
{
  Reset();
  Left();  // the root is left linked to the sentinel.
  while (HasRight())
    Right();
}
```


Avançar, recuar

Estas afinal também são simples, pois passam o trabalho a outras:

```
template <class T>
void TreePolymorphic<T>::Forth()
{
  cursor = Next(cursor);
}

template <class T>
void TreePolymorphic<T>::Back()
{
  cursor = Previous(cursor);
}
```

Estas funções Next e Previous têm argumentos de tipo TreeNodePolymorphic<T>*. Não podem ser funções públicas, pois não queremos apontadores na interface da classe. Serão privadas?

Funções protegidas (1)

Vamos declarar as funções que manipulam apontadores TreeNodePolymorphic<T>* como protegidas, para não serem públicas mas poderem ser usadas por classes derivadas.

```
template <class T>
class TreePolymorphic: public Container<T>,  public Bilinear<T>{
// ...
protected:
  virtual TreeNodePolymorphic<T>* Next();
  virtual TreeNodePolymorphic<T>* Previous();
  virtual TreeNodePolymorphic<T>* Parent();
  virtual void Goto(TreeNodePolymorphic<T>* p);
  virtual void SwapWith(TreeNodePolymorphic<T>* p);

  virtual bool IsLeaf(const TreeNodePolymorphic<T>* p) const;
  virtual bool Side(const TreeNodePolymorphic<T>* p) const;
  virtual bool HasLeft(const TreeNodePolymorphic<T>* p) const;
  virtual bool HasRight(const TreeNodePolymorphic<T>* p) const;
  virtual bool HasChild(const TreeNodePolymorphic<T>* p, bool right) const;
  virtual int CountChildren(const TreeNodePolymorphic<T>* p) const;
// ...
```

O nó a seguir ao cursor (num percurso directo), o nó antes do cursor e o pai do cursor.

Mandar o cursor apontar para p, trocar o valor do cursor com o valor do nó apontado por p.

Os nomes são auto-explicativos, não são?.

Funções protegidas (2)

```
 // ...
virtual TreeNodePolymorphic<T>* Offspring(const TreeNodePolymorphic<T>* p);
      // post "returns one the the children, ie, if there are two, returns one of them,
      // else if there is one returns it,
      // else if there is none returns a pointer to the sentinel."
  virtual bool Off(const TreeNodePolymorphic<T>* p) const;

  virtual void Promote(TreeNodePolymorphic<T>* p);

  virtual void SwapItems(TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2);

  virtual const TreeNodePolymorphic<T>* RootNode() const;
  virtual TreeNodePolymorphic<T>* RootNode();

  virtual const TreeNodePolymorphic<T>*
                    Next(const TreeNodePolymorphic<T>* x, bool right = true) const;
  virtual TreeNodePolymorphic<T>* Next(TreeNodePolymorphic<T>* x, bool right = true);
  virtual const TreeNodePolymorphic<T>* Previous(const TreeNodePolymorphic<T>* x) const;
  virtual TreeNodePolymorphic<T>* Previous(TreeNodePolymorphic<T>* x);
};
```

Off(p) dá true se p não apontar para nenhum dos nós da árvore, isto é, se apontar para sentinela.

Promover o nó apontado por p.

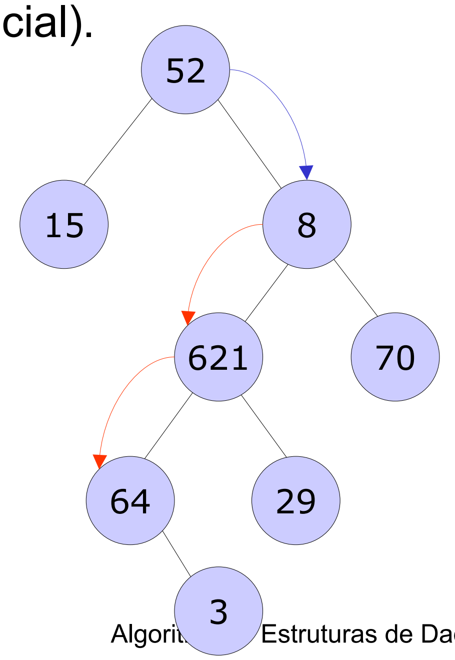
Trocar os valores dos nós apontados por p1 e p2.

O nó da raiz.

Next(p) dá o nó a seguir a p, num percurso directo; Previous(p) dá o nó antes.

Qual é o próximo? (1)

Como passamos de um nó ao próximo? Se tiver filho direito, viramos à direita e depois sempre à esquerda, enquanto for possível (assim apanhando o primeiro nó da subárvore direita do nó inicial).

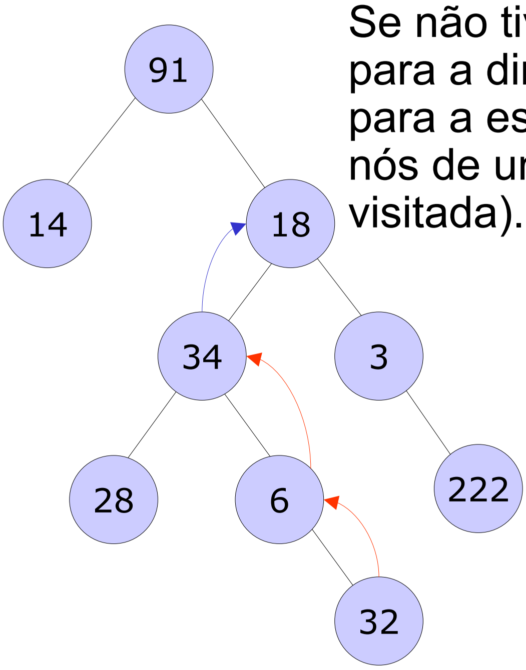


Do 52 vira-se à direita para o 8, do 8 vira-se à esquerda para o 621 e do 621 vira-se à esquerda para o 64. Como do 64 não se pode virar à esquerda, o 64 é o sucessor do 52.

```
if (x->Right() != &sentinel)
{
  x = x->Right();
  while (x->Left() != &sentinel)
    x = x->Left();
}
else
// ...
```


Qual é o próximo? (2)

Se não tiver filho direito, subimos até subirmos para a direita (pois enquanto a subida tiver sido para a esquerda, teremos estado a apanhar nós de uma subárvore já complemente visitada).



Do 32 sobe-se para o 6, do 6 sobe-se para o 34, do 34 sobe-se para o 18. A primeira subida é para a esquerda e a segunda também. A terceira é para a direita, e por isso não se sobe mais. Logo, o sucessor do 32 é o 18.

```
// ...
else
{
  const TreeNodePolymorphic<T>* p = x;
  x = x->Parent();
  while (x != &sentinel && p != x->Left())
  {
    p = p->Parent();
    x = x->Parent();
  }
}
```

TreePolymorphic<T>::Next

Assim se passa para o próximo:

```
template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Next(TreeNodePolymorphic<T>* x)
{
  if (x->Right() != &sentinel)
  {
    x = x->Right();
    while (x->Left() != &sentinel)
      x = x->Left();
  }
  else
  {
    const TreeNodePolymorphic<T>* p = x;
    x = x->Parent();
    while (x != &sentinel && p != x->Left())
    {
      p = p->Parent();
      x = x->Parent();
    }
  }
  return x;
}
```

Esta é a função mais complicada das classe TreePolymorphic<T>: um if-else, dois whiles (um em cada ramo do if)!

Generalizando

Andar para trás tem de ser simétrico de andar para a frente. Podemos generalizar com um parâmetro booleano para dizer se vamos para a frente (para a direita) ou para trás:

```
template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Next(TreeNodePolymorphic<T>* x, bool right)
{
  if (x->Child(right) != &sentinel)
  {
    x = x->Child(right);
    while (x->Child(!right) != &sentinel)
      x = x->Child(!right);
  }
  else
  {
    const TreeNodePolymorphic<T>* p = x;
    x = x->Parent();
    while (x != &sentinel && p != x->Child(!right))
    {
      p = p->Parent();
      x = x->Parent();
    }
  }
  return x;
}
```

Qual é o anterior?

O anterior é o próximo quando de anda para trás:

```
template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Previous(TreeNodePolymorphic<T>* x)
{
  return Next(x, false);
}
```

A função Next também se pode usar sem o segundo argumento, pois esse argumento tem um valor por defeito:

```
  // ...
  virtual TreeNodePolymorphic<T>* Next(TreeNodePolymorphic<T>* x, bool right = true);
  virtual TreeNodePolymorphic<T>* Previous(TreeNodePolymorphic<T>* x);
  // ...
```

Note bem: ambas as funções são não const pois devolvem um apontador através do qual se pode modificar o objecto (isto é, a árvore). Por isso, não podem ser usadas com árvores constantes.

Andando em árvores constantes

Eis as funções Next e Previous para árvores constantes:

```
  // ...
  virtual const TreeNodePolymorphic<T>*
    Next(const TreeNodePolymorphic<T>* x, bool right = true) const;
  virtual const TreeNodePolymorphic<T>* Previous(const TreeNodePolymorphic<T>* x) const;
  // ...
```

Para não programar tudo de novo, usamos dois const_casts.

```
template <class T>
const TreeNodePolymorphic<T>* TreePolymorphic<T>::
  Next(const TreeNodePolymorphic<T>* x, bool right) const
{
  return
    const_cast<TreePolymorphic<T>*>(this)->Next(const_cast<TreeNodePolymorphic<T>*>(x), right);
}
```

const_cast

O operador const_cast usa-se para remover a constância de um objecto de uma classe, assim permitindo invocar para esse objecto constante funções não const. Só deve ser usado se essas funções, mesmo não tendo sido declaradas const, não mudarem o valor do objecto.

Observe uma maneira alternativa, mais pausada, de programar a função Next, versão const:

```
template <class T>
const TreeNodePolymorphic<T>* TreePolymorphic<T>::
  Next(const TreeNodePolymorphic<T>* x, bool right) const
{
  TreePolymorphic<T>* that = const_cast<TreePolymorphic<T>*>(this);
  TreeNodePolymorphic<T>* y = const_cast<TreeNodePolymorphic<T>*>(x);
  return that->Next(y, right);
}
```

A função Next chamada é a versão não const: o objecto that é um apontador não const para TreePolymorphic<T> e o argumento y é um apontador não const para TreeNodePolymorphic<T>.

As outras funções protegidas (1)

São todas simples:

```
template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Next()
{
  return Next(cursor);
}

template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Previous()
{
  return Previous(cursor);
}

template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Parent()
{
  return cursor->Parent();
}

template <class T>
void TreePolymorphic<T>::Goto(TreeNodePolymorphic<T>* p)
{
  cursor = p;
}

template <class T>
void TreePolymorphic<T>::SwapWith(TreeNodePolymorphic<T>* p)
{
  TreeNodePolymorphic<T>::SwapItems(cursor, p);
  cursor = p;
}
```

As outras funções protegidas (2)

```
template <class T>
bool TreePolymorphic<T>::IsLeaf(const TreeNodePolymorphic<T>* p) const
{
  return p->Left() == &sentinel && p->Right() == &sentinel;
}

template <class T>
bool TreePolymorphic<T>::Side(const TreeNodePolymorphic<T>* p) const
{
  return p->Parent()->Child(1) == p;
}

template <class T>
bool TreePolymorphic<T>::HasLeft(const TreeNodePolymorphic<T>* p) const
{
  return HasChild(p, 0);
}

template <class T>
bool TreePolymorphic<T>::HasRight(const TreeNodePolymorphic<T>* p) const
{
  return HasChild(p, 1);
}

template <class T>
bool TreePolymorphic<T>::HasChild(const TreeNodePolymorphic<T>* p, bool right) const
{
  return p->Child(right) != &sentinel;
}

template <class T>
int TreePolymorphic<T>::CountChildren(const TreeNodePolymorphic<T>* p) const
{
  return HasLeft(p) + HasRight(p);
}
```

As outras funções protegidas (3)

```
template <class T>
bool TreePolymorphic<T>::Off(const TreeNodePolymorphic<T>* p) const
{
  return p == &sentinel;
}

template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::Offspring(const TreeNodePolymorphic<T>* p)
{
  return p->Child(Off(p->Child(0)));
}

template <class T>
void TreePolymorphic<T>::
  SwapItems(TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2)
{
  TreeNodePolymorphic<T>::SwapItems(p1, p2);
}

template <class T>
TreeNodePolymorphic<T>* TreePolymorphic<T>::RootNode()
{
  return sentinel.Left();
}

template <class T>
const TreeNodePolymorphic<T>* TreePolymorphic<T>::RootNode() const
{
  return sentinel.Left();
}
```

Esta é engraçada: se houver filho esquerdo, retorna um apontador para ele; se não houver retorna um apontador para o filho direito, que será o apontador para a sentinela se não houver filho direito (nem filho esquerdo, portanto).

A função SwapItems chamada é a função estática da classe TreeNodePolymorphic<T>.

Promovendo

Só falta a função para promover um nó referenciado por um apontador. Temos de manipular os apontadores dos nós envolvidos com cuidado:

```
template <class T>
void TreePolymorphic<T>::Promote(TreeNodePolymorphic<T>* x)
{
  bool side = Side(x);
  TreeNodePolymorphic<T>* p = x->Parent();
  x->SetParent(p->Parent());
  x->Parent()->SetChild(Side(p), x);
  p->SetChild(side, x->Child(!side));
  p->Child(side)->SetParent(p);
  x->SetChild(!side, p);
  p->SetParent(x);
}
```

Depois disto, fica simples promover o cursor:

```
template <class T>
void TreePolymorphic<T>::Promote()
{
  Promote(cursor);
}
```


Pondo, contando

Redefinamos as funções herdadas de Container<T>:

Pôr é inserir do lado esquerdo, pendurando a subárvore esquerda do lado esquerdo:

```
template <class T>
void TreePolymorphic<T>::Put(const T& s)
{
  Insert(s, false, false);
}
```

Não é preciso contar, porque já está contado:

```
template <class T>
int TreePolymorphic<T>::Count() const
{
  return count;
}
```

Inserir é isto:

```
template <class T>
void TreePolymorphic<T>::Insert(const T& s, bool right1, bool right2)
{
  Link(new TreeNodePolymorphic<T>(s), cursor, right2, right1);
  Down(right1);
  count++;
}
```

Isto é: liga-se um novo nó do lado indicado, pendurando a subárvore solta do lado indicado, reposiciona-se o cursor e incrementa-se o contador.

Ligando um nó numa árvore

O comentário exprime o significado da função:

```
template <class T>
void TreePolymorphic<T>::
  Link(TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2, bool right1, bool right2)
{
  // links *p1 to the right2 of *p2, relinking the right2 of *p2 to the right1 of *p1
  p1->SetParent(p2);
  p1->SetChild(right1, p2->Child(right2));
  p1->SetChild(!right1, &sentinel);
  p2->Child(right2)->SetParent(p1);
  p2->SetChild(right2, p1);
}
```

Compare com a função Link da classe ListDoublePolymorphic<T>.

Esta função é privada:

```
private:
  virtual void Link0(TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2);
  virtual void Link(TreeNodePolymorphic<T>* p1, TreeNodePolymorphic<T>* p2,
                         bool right1 = false, bool right2 = false);
      // links *p1 to the right2 of *p2, relinking the right2 of *p2 to the right1 of *p1
  virtual void Unlink(TreeNodePolymorphic<T>* p1); // pre CountChildren(p) <= 1;
```

Claro que também há uma função para desligar um nó.

Removendo

As árvores não são distribuidores, mas têm Remove e Item:

```
template <class T>
void TreePolymorphic<T>::Remove()
{
  TreeNodePolymorphic<T>* p = cursor;
  Up();
  Unlink(p);
  delete p;
  count--;
}
```

Antes de o nó ser apagado, o cursor sobe.

Para remover, desliga-se o nó apontado pelo cursor, apaga-se e desconta-se.

O item é o item:

```
template <class T>
const T& TreePolymorphic<T>::Item() const
{
  return cursor->Item();
}

template <class T>
T& TreePolymorphic<T>::Item()
{
  return cursor->Item();
}
```

A raiz está à esquerda da sentinela:

```
template <class T>
const T& TreePolymorphic<T>::Root() const
{
  return sentinel.Left()->Item();
}

template <class T>
T& TreePolymorphic<T>::Root()
{
  return sentinel.Left()->Item();
}
```

Desligando um nó

O comentário recorda a precondição:

```
template <class T>
void TreePolymorphic<T>::Unlink(TreeNodePolymorphic<T>* p1)
{
  // unlinks *p1 which has zero or one child;
  p1->Parent()->SetChild(Side(p1), Offspring(p1));
  Offspring(p1)->SetParent(p1->Parent());
}
```

Compare com a função Unlink da classe ListDoublePolymorphic<T>.

A precondição da função Unlink é garantida pela precondição da função Remove:

```
virtual void Remove(); // pre !Off() && CountChildren() <= 1;
                      // post "the cursor points to the parent of the removed node.";
```

```
virtual void Unlink(TreeNodePolymorphic<T>* p1); // pre CountChildren(p) <= 1;
```

Tecnicamente, as árvores não são distribuidores porque a precondição da função Remove em TreePolymorphic<T> é mais forte do que em Dispenser<T>. Em Dispenser<T> para apagar basta que não esteja Off. Aqui é preciso isso e ainda que o nó não tenha dois filhos.

Construtores e destrutor

O construtor por defeito constrói uma árvore vazia:

```
template <class T>
TreePolymorphic<T>::TreePolymorphic():
  sentinel(),
  cursor(&sentinel),
  count(0)
{
  sentinel.SetLeft(&sentinel);
  sentinel.SetRight(&sentinel);
  sentinel.SetParent(&sentinel);
}
```

Todos os apontadores da sentinela apontam para a sentinela.

Antes de destruir a árvore, eliminam-se todos os nós:

```
template <class T>
TreePolymorphic<T>::~TreePolymorphic()
{
  Clear();
}
```

Juntando iteradores

Já que as árvores são estruturas bilineares, podemos juntar à classe funções que devolvem iteradores para percorrer as árvores de várias maneiras.

```
template <class T>
class TreePolymorphic: public Container<T>, public Bilinear<T>{
private:
  // ...
public:
  // ...
public: // iterator functions
  virtual IteratorSmart<T> Items() const;
  virtual IteratorSmart<T> ItemsReverse() const;
  virtual IteratorSmart<T> ItemsInOrder() const;
  virtual IteratorSmart<T> ItemsPreOrder() const;
  virtual IteratorSmart<T> ItemsLevelOrder() const;
  // ...
};
```

Percurso directo.

Percurso reverso.

Percurso infixo: primeiro a raiz, depois a subárvore esquerda, depois a subárvore direita.

Percurso prefixo: primeiro a subárvore esquerda, depois a raiz, depois a subárvore direita. (Equivalente ao percurso directo.)

Percurso por níveis: primeiro o pai, depois os filhos, depois os netos...

Iteradores

Iterator<T> é uma classe abstracta genérica que representa sequências de objectos de tipo T. Os elementos de cada sequência são debitados um a um pelo iterador.

```
template <class T>
class Iterator {
public:
  virtual ~Iterator();
  virtual Iterator<T>* Clone() const = 0;

  virtual const T& Current() const = 0; // pre ! IsDone();
  virtual bool IsDone() const = 0;
  virtual void Get() = 0; // pre ! IsDone();

  virtual const T& operator *() const;   // pre ! IsDone();
  virtual const T* operator ->() const;  // pre ! IsDone();
  virtual operator bool() const;
  virtual void operator ++(int);  // pre ! IsDone();
};
```

Novidade: os iteradores são clonáveis. No entanto, nós não vamos usar essa facilidade na classe TreePolymorphic<T>.

Interface alternativa.

Implementação da interface alternativa

```
template <class T>
void Iterator<T>::operator ++(int)
{
  Get();
}

template <class T>
const T& Iterator<T>::operator *() const
{
  return Current();
}

template <class T>
const T* Iterator<T>::operator ->() const
{
  return &Current();
}

template <class T>
Iterator<T>::operator bool() const
{
  return !IsDone();
}
```

Precisamos dos dois operadores * e ->. Com efeito, para um iterador i, escreveremos *i para aceder ao elemento corrente. Logo, para lhe aplicar uma função F, quereremos escrever i->F() em vez do mais verboso (*i).F().

Para mais informação, veja no livro.

Iteradores *smart* (1)

Iterator<T> é uma classe abstracta. Não há objectos directamente de tipo Iterator<T>, mas apenas de tipos derivados de Iterator<T>. Por isso, as funções não podem retornar Iterator<T>.

Em vez disso, retornam IteratorSmart<T>. Esta é uma classe não abstracta que herda de Iterator<T>. Tem um membro de dados de tipo Iterator<T>* que é libertado automaticamente pelo destrutor.

```
template <class T>
class IteratorSmart: public Iterator<T> {
private:
  Iterator<T>* i;
public:
  // ...
};
```

Iteradores *smart* (2)

A versão actual é uma evolução da classe do livro.

```
template <class T>
class IteratorSmart: public Iterator<T> {
private:
  Iterator<T>* i;
  const Predicate<T> *p;
public:
  explicit IteratorSmart(Iterator<T>* i, const Predicate<T>& p = Predicate<T>());
  explicit IteratorSmart(const Iterator<T>& i, const Predicate<T>& p = Predicate<T>());
  IteratorSmart(const IteratorSmart<T>& other);
  virtual ~IteratorSmart();
  virtual Iterator<T>* Clone() const;

  virtual const T& Current() const;
  virtual bool IsDone() const;
  virtual void Get();
private:
  virtual void Advance();
};
```

Por enquanto, podemos ignorar este segundo membro de dados, de tipo Predicate<T>.

Para mais informação sobre iteradores *smart*, genéricos e não genéricos, veja no livro.

Classes iterador na classe Tree…<T>

Dentro da classe TreePolymorphic<T> há as classes iterador que servirão para criar os objectos IteratorSmart<T> retornados pelas funções. Observemos o iterador directo:

```
template <class T>
class TreePolymorphic: public Container<T>,  public Bilinear<T>{
// …
public: // iterator classes
  class TreeIteratorDirect: public Iterator<T> {
  private:
    const TreePolymorphic<T>& tree;
    const TreeNodePolymorphic<T>* current;
  public:
    explicit TreeIteratorDirect(const TreePolymorphic<T>& tree);
    virtual ~TreeIteratorDirect();
    virtual Iterator<T>* Clone() const;
    virtual const T& Current() const;
    virtual bool IsDone() const;
    virtual void Get();
  };
  // …
};
```

Uma referência const para a árvore que está a ser iterada.

Um apontador para o nó corrente.

Normalmente, as definições destas funções apareceriam no ficheiro TreePolymorphic.cpp, juntamente com as outras funções da classe TreePolymorphic<T>. No entanto...

Tecnologia ☹

O compilador VC++ 7.0 não suporta a definição separada de funções de classes internas de classe genéricas.

Erro C3206:

> **'*function*' : member functions of nested classes of a template class cannot be defined outside the class**
>
> For inner nested classes inside a template, you must define functions inside the class. Such functions automatically become inline functions.
>
> This error is generated for code allowed by the C++ language, however, not yet supported by Visual C++.

Exemplo de função que dá o erro C3206:

```
template <class T>
void TreePolymorphic<T>::TreeIteratorDirect::Get()
{
  current = tree.Next(current);
}
```

> Note bem: isto é válido em C++, mas o VC++ 7.0 ainda não aceita isto.
>
> Temos de dar a volta ao problema.

O iterador directo (1)

Como não podemos definir separadamente, definimos internamente, logo na declaração da classe, assim:

```
template <class T>
class TreePolymorphic: public Container<T>,  public Bilinear<T>{
// ...
public: // iterator classes
 class TreeIteratorDirect: public Iterator<T> {
  private:
    const TreePolymorphic<T>& tree;
    const TreeNodePolymorphic<T>* current;
  public:
    explicit TreeIteratorDirect(const TreePolymorphic<T>& tree):
      tree(tree),
      current(&tree.sentinel)
    {
      while (tree.HasLeft(current))
        current = current->Left();
    };

    virtual ~TreeIteratorDirect()
    {
    };

// ...
};
```

> Repare bem nesta técnica. Em teoria, poderíamos programar sempre assim, em todas as classes, mas não é costume e, além disso, o desenvolvimento ficava mais difícil.

O iterador directo (2)

Eis o resto da classe interna:

```
  // ...
  virtual Iterator<T>* Clone() const {return 0;};

  virtual const T& Current() const
  {
    return current->Item();
  };

  virtual bool IsDone() const
  {
    return tree.Off(current);
  };

  virtual void Get()
  {
    current = tree.Next(current);
  };
 };
 // ...
};
```

Como não vamos precisar de clonar iteradores, por enquanto, deixamos aqui um *stub*.

O iterador reverso

O iterador reverso é semelhante:

```
class TreeIteratorReverse: public Iterator<T> {
 private:
  const TreePolymorphic<T>& tree;
  const TreeNodePolymorphic<T>* current;
 public:
  explicit TreeIteratorReverse(const TreePolymorphic<T>& tree):
    tree(tree),
    current(&tree.sentinel)
  {
    current = current->Left();
    while (tree.HasRight(current))
      current = current->Right();
  };

  // ...

  virtual void Get()
  {
    current = tree.Previous(current);
  };
 };
};
```

Recorde que esta é uma classe interna.

Agora o Get faz Previous, em vez de Next.

O iterador infixo (1)

Ao iterar infixamente, primeiro visita-se a raiz, depois a subárvore esquerda e depois a subárvore direita. Isso implementa-se com uma pilha. Ao visitar um nó, colocam-se os seu filhos na pilha. Em cada passo da iteração, visita-se o topo da pilha.

Como não queremos duplicar os nós e como sabemos quantos nós há, usamos uma pilha limitada indirecta:

```
class TreeIteratorInOrder: public Iterator<T> {
 private:
   const TreePolymorphic<T>& tree;
   StackBoundedIndirect<TreeNodePolymorphic<T> > stack;
 public:
   explicit TreeIteratorInOrder(const TreePolymorphic<T>& tree):
     tree(tree),
     stack(tree.Count())
   {
     if (!tree.Empty())
       stack.Put(*tree.RootNode());
   };
```

Recorde que esta é uma classe interna.

A pilha é criada com a capacidade certa.

O construtor coloca a raiz na pilha. O elemento corrente será o topo da pilha, neste caso a raiz.

O iterador infixo (2)

```
   virtual const T& Current() const
   {
     return stack.Item().Item();
   }

   virtual bool IsDone() const
   {
     return stack.Empty();
   };

   virtual void Get()
   {
     const TreeNodePolymorphic<T>& temp = stack.Item();
     stack.Remove();
     if (tree.HasRight(&temp))
       stack.Put(*temp.Right());
     if (tree.HasLeft(&temp))
       stack.Put(*temp.Left());
   };
 };
```

Atenção ao Get!

Primeiro empilha-se o filho direito para depois desempilhar primeiro o filho esquerdo...

Omitimos o destrutor, que não faz nada e o Clone, que é um *stub*...

O iterador prefixo (1)

O iterador prefixo também usa uma pilha, mas de maneira diferente. Ao visitar um nó, empilhamos a linhagem esquerda do seu filho direito (se houver). Assim a subárvore esquerda de um nó é visitada antes do pai do nó (e da respectiva subárvore direita), tal como desejamos.

```
class TreeIteratorPreOrder: public Iterator<T> {
 private:
   const TreePolymorphic<T>& tree;
   StackBoundedIndirect<TreeNodePolymorphic<T> > stack;
 public:
   explicit TreeIteratorPreOrder(const TreePolymorphic<T>& tree):
     tree(tree),
     stack(tree.Count())
   {
     const TreeNodePolymorphic<T> *p = tree.RootNode();
     while (!tree.Off(p))
     {
       stack.Put(*p);
       p = p->Left();
     }
   };
```

Recorde que esta é uma classe interna.

A pilha é criada com a capacidade certa.

O construtor empilha logo a linhagem esquerda da raiz.

Recorde que como a pilha é indirecta, não há duplicação de objectos.

O iterador prefixo (2)

```
   virtual const T& Current() const
   {
     return stack.Item().Item();
   };

   virtual bool IsDone() const
   {
     return stack.Empty();
   };

   virtual void Get()
   {
     const TreeNodePolymorphic<T>& temp = stack.Item();
     stack.Remove();
     TreeNodePolymorphic<T> *p= temp.Right();
     while (!tree.Off(p))
     {
       stack.Put(*p);
       p = p->Left();
     }
   };
 };
```

Aqui se empilha a linhagem esquerda do nó temp.

Omitimos o destrutor, que não faz nada e o Clone, que é um *stub*...

O iterador por níveis (1)

Para iterar por níveis usamos uma fila, em vez de uma pilha. Ao visitar um nó, colocam-se na fila os seus filhos. Em cada passo da iteração, retira-se da fila o próximo elemento a visitar.

```
class TreeIteratorLevelOrder: public Iterator<T> {
 private:
  const TreePolymorphic<T>& tree;
  QueueBoundedIndirect<TreeNodePolymorphic<T> > queue;
 public:
  explicit TreeIteratorLevelOrder(const TreePolymorphic<T>& tree):
    tree(tree),
    queue(tree.Count())
  {
    if (!tree.Empty())
      queue.Put(*tree.RootNode());
  };
```

Recorde que esta é uma classe interna.

A fila é criada com a capacidade certa.

O construtor mete a raiz na fila

Recorde que como a fila é indirecta, não há duplicação de objectos.

O iterador por níveis (2)

```
  virtual const T& Current() const
  {
    return queue.Item().Item();
  };

  virtual bool IsDone() const
  {
    return queue.Empty();
  }

  virtual void Get()
  {
    const TreeNodePolymorphic<T>& temp = queue.Item();
    queue.Remove();
    if (tree.HasLeft(&temp))
      queue.Put(*temp.Left());
    if (tree.HasRight(&temp))
      queue.Put(*temp.Right());
  }
};
```

Primeiro entra o filho esquerdo e depois o filho direito, para mais tarde saírem pela mesma ordem.

Omitimos o destrutor, que não faz nada e o Clone, que é um *stub*...

Devolvendo os iteradores

Usa-se sempre a mesma técnica: invocar o construtor da classe IteratorSmart<T> passando um iterador criado dinamicamente do tipo desejado inicializado com a árvore a iterar:

```
template <class T>
IteratorSmart<T> TreePolymorphic<T>::Items() const
{
  return IteratorSmart<T>(new TreePolymorphic<T>::TreeIteratorDirect(*this));
}

template <class T>
IteratorSmart<T> TreePolymorphic<T>::ItemsReverse() const
{
  return IteratorSmart<T>(new TreePolymorphic<T>::TreeIteratorReverse(*this));
}

template <class T>
IteratorSmart<T> TreePolymorphic<T>::ItemsInOrder() const
{
  return IteratorSmart<T>(new TreePolymorphic<T>::TreeIteratorInOrder(*this));
}

template <class T>
IteratorSmart<T> TreePolymorphic<T>::ItemsPreOrder() const
{
  return IteratorSmart<T>(new TreePolymorphic<T>::TreeIteratorPreOrder(*this));
}

template <class T>
IteratorSmart<T> TreePolymorphic<T>::ItemsLevelOrder() const
{
  return IteratorSmart<T>(new TreePolymorphic<T>::TreeIteratorLevelOrder(*this));
}
```

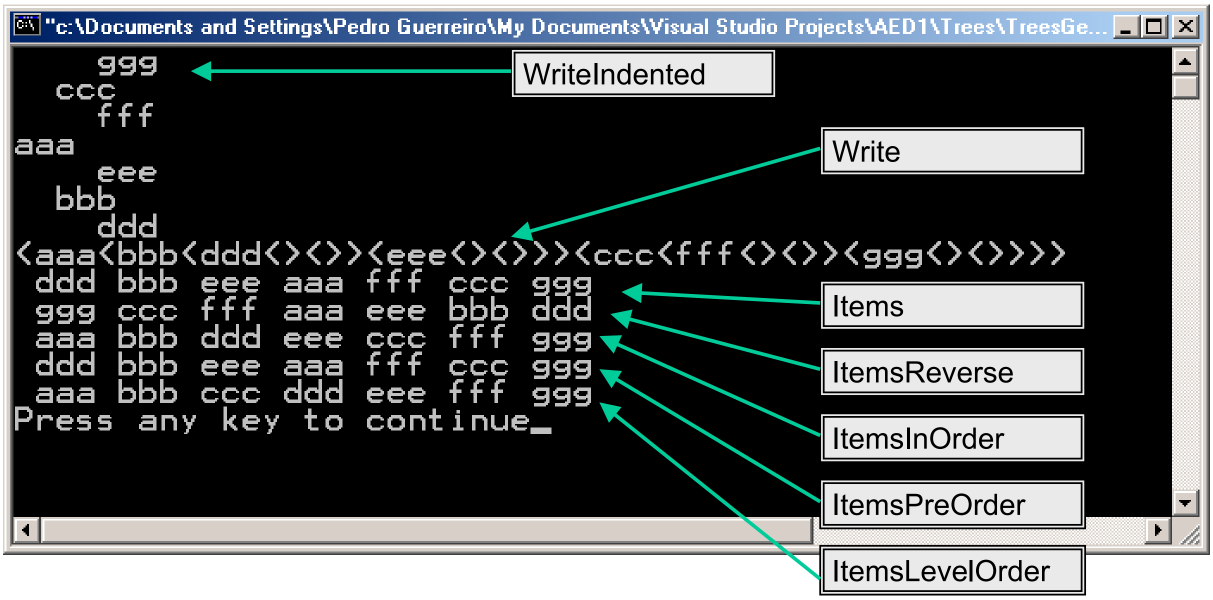

Utilizando os iteradores

Eis uma função para testar os cinco iteradores:

```
void TestIterators()
{
  TreePolymorphic<StringBasic>::SetPrefixSuffix("<", ">");
  TreePolymorphic<StringBasic> t;
  t.Put("aaa");
  t.Put("bbb");
  t.Up();
  t.Insert("ccc", true);
  t.Search("bbb");
  t.Insert("ddd", false);
  t.Up();
  t.Insert("eee", true);
  t.Search("ccc");
  t.Insert("fff", false);
  t.Up();
  t.Insert("ggg", true);

  t.WriteIndented("  ");
  t.WriteLine();
  // ...
```

```
  // ...
  for (IteratorSmart<StringBasic>& i = t.Items(); i; i++)
    std::cout << " " << *i;
  std::cout << std::endl;
  for (IteratorSmart<StringBasic>& i = t.ItemsReverse(); i; i++)
    std::cout << " " << *i;
  std::cout << std::endl;
  for (IteratorSmart<StringBasic>& i = t.ItemsInOrder(); i; i++)
    std::cout << " " << *i;
  std::cout << std::endl;
  for (IteratorSmart<StringBasic>& i = t.ItemsPreOrder(); i; i++)
    std::cout << " " << *i;
  std::cout << std::endl;
  for (IteratorSmart<StringBasic>& i = t.ItemsLevelOrder(); i; i++)
    std::cout << " " << *i;
  std::cout << std::endl;
}
```

Resultado do teste



Observamos que WriteIndented escreve os nós pela mesma ordem que ItemsReverse e que Write escreve pela mesma ordem que ItemsInOrder.

Pilha, fila limitada indirecta

Usa-se um vector indirecto:

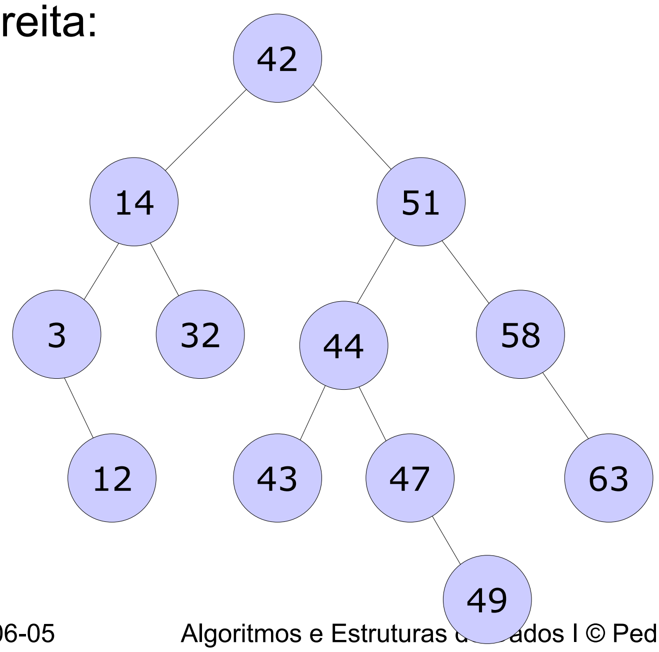
```
template <class T>
class StackBoundedIndirect: public Dispenser<T> {
private:
  VectorIndirect<T> items;
public:
  StackBoundedIndirect(int capacity);
  virtual ~StackBoundedIndirect();

  // ...
};
```

```
template <class T>
class QueueBoundedIndirect: public Dispenser<T> {
private:
  VectorIndirect<T> items;
  int count;
  int in;
  int out;
public:
  // ...
};
```

E aqui termina a apresentação da classe TreePolymorphic<T>.

Árvores binárias de busca

Ou *binary search trees*, BST: para cada subárvore, todos os elementos da subárvore da esquerda são menores do que a raiz e a raiz é menor do que todos os elementos da subárvore da direita:

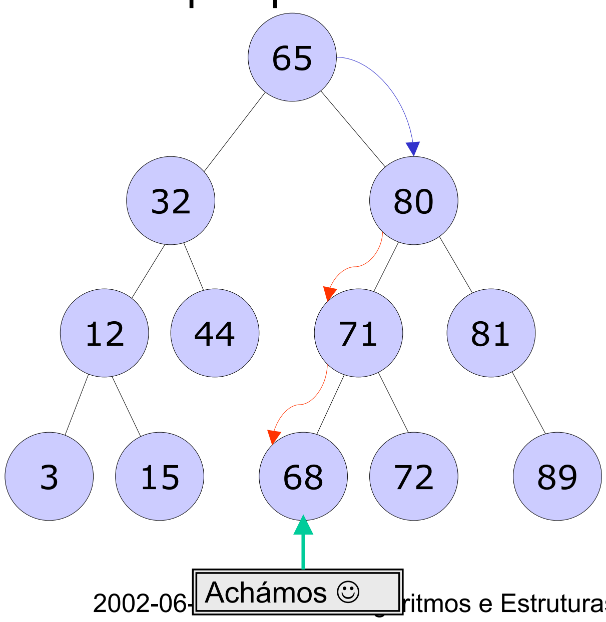


Sendo assim, ao fazer um percurso directo da árvore, apanhamos os elementos por ordem crescente: 3, 12, 14, 32, 42, 43, 44, 47, 49, 51, 58, 63.

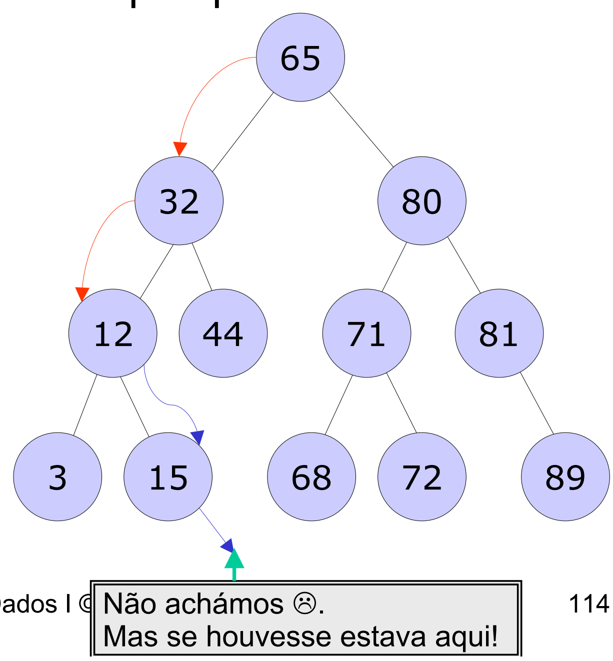
Procurando dicotomicamente

Se o elemento procurado é igual à raiz, já achámos; se não, se é menor, procuramos na subárvore esquerda; se não, procuramos na subárvore direita:

Exemplo: procurando 68:



Exemplo: procurando 20:



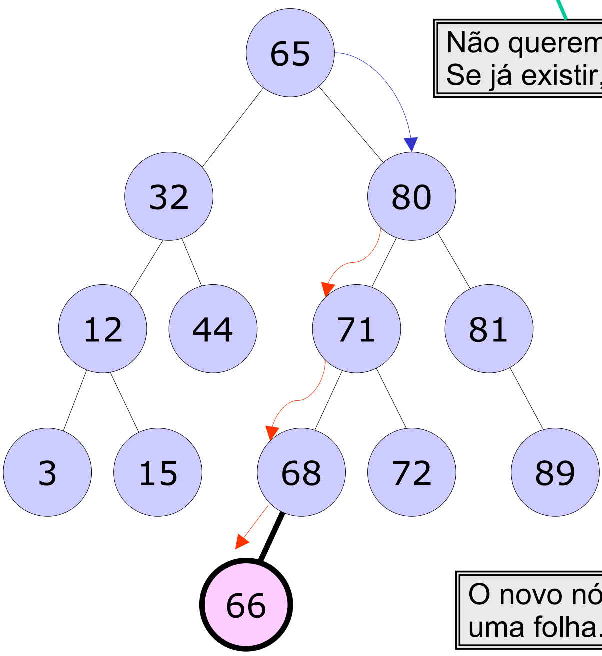
Pondo na árvore de busca

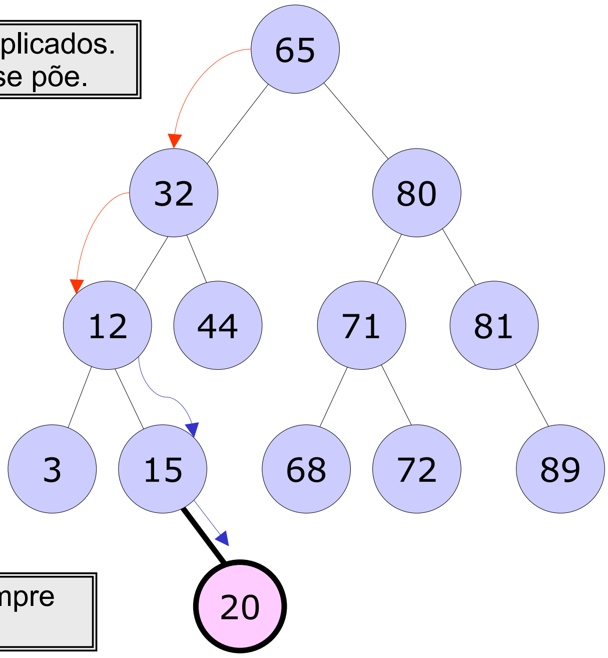
Procura-se e, não havendo, pendura-se do lado certo:

Exemplo: pondo 66:

Exemplo: pondo 20:

Não queremos duplicados.
Se já existir, não se põe.

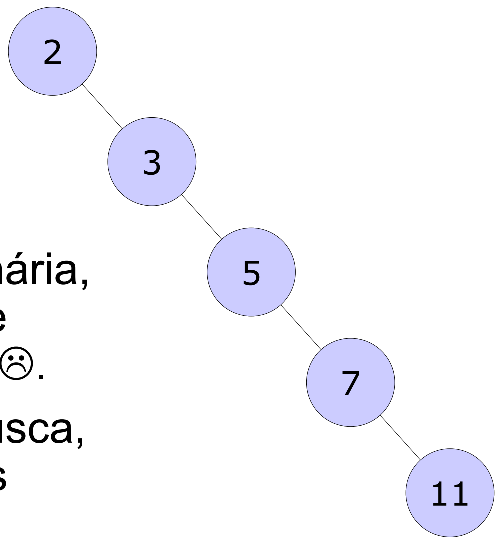




O novo nó é sempre uma folha.

Enviesando...

Se pomos por ordem, enviesamos a árvore. Exemplo: pôr numa árvore de busca os primeiros cinco números primos, gerados sucessivamente:



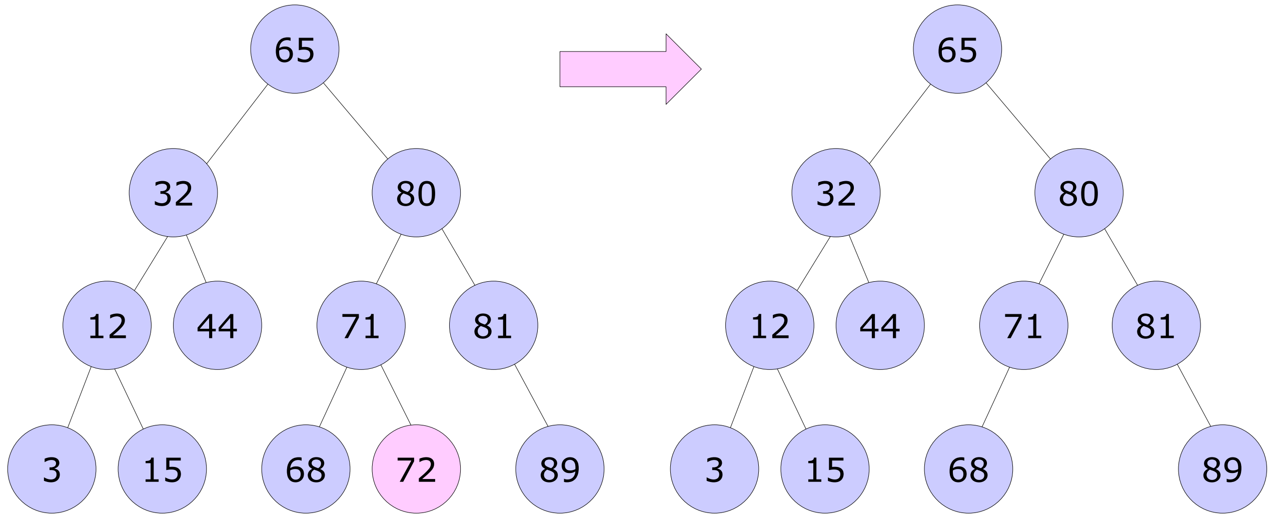
Sim, isto é uma árvore binária, mas tão enviesada que se comporta como uma lista ☹.

Ao pôr numa árvore de busca, convém que os elementos surjam aleatoriamente.

Apagando (1)

Apagar uma folha é simplicíssimo.

Exemplo: apagar o 72:



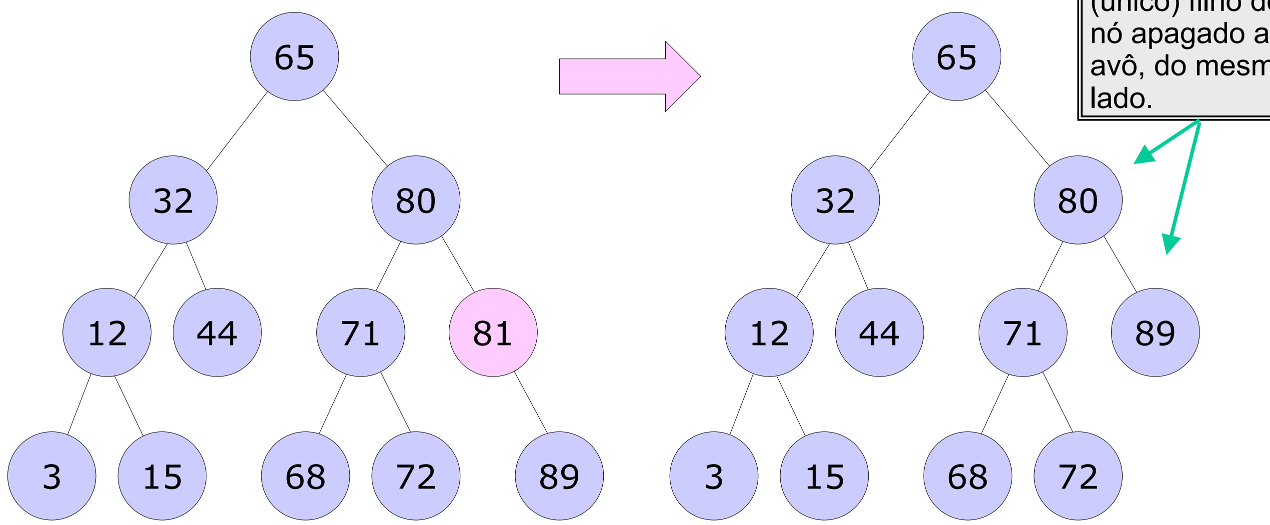
A árvore continua a ser uma árvore de busca, claro.

Apagando (2)

Apagar um nó que só tem um filho também não é complicado:

Exemplo: apagar o 81:

Liga-se o (único) filho do nó apagado ao avô, do mesmo lado.

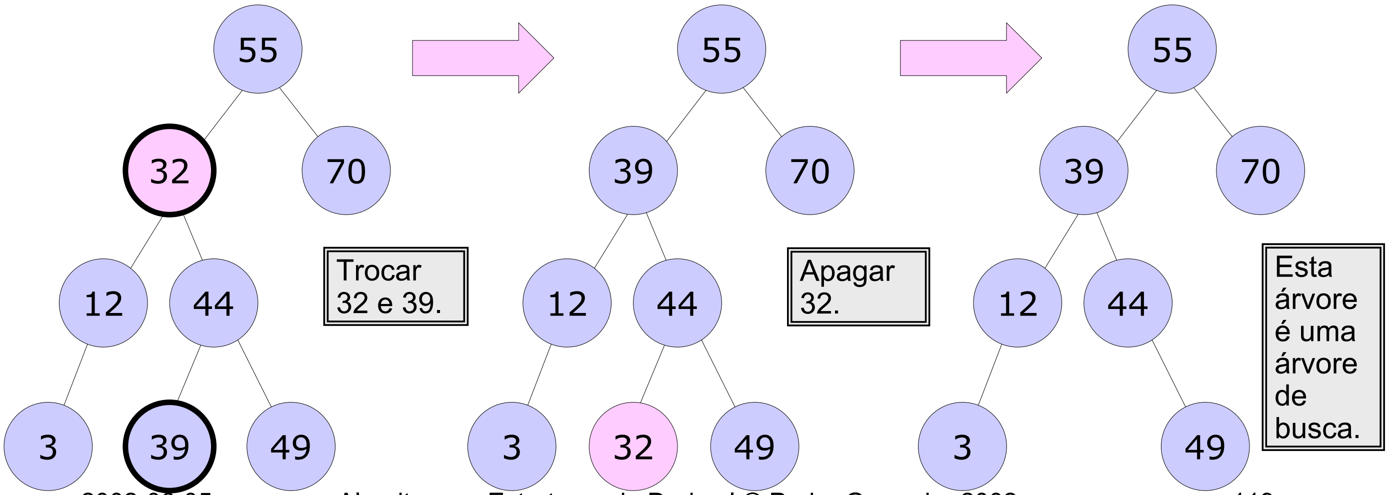


A árvore continua a ser uma árvore de busca, claro.

Apagando (3)

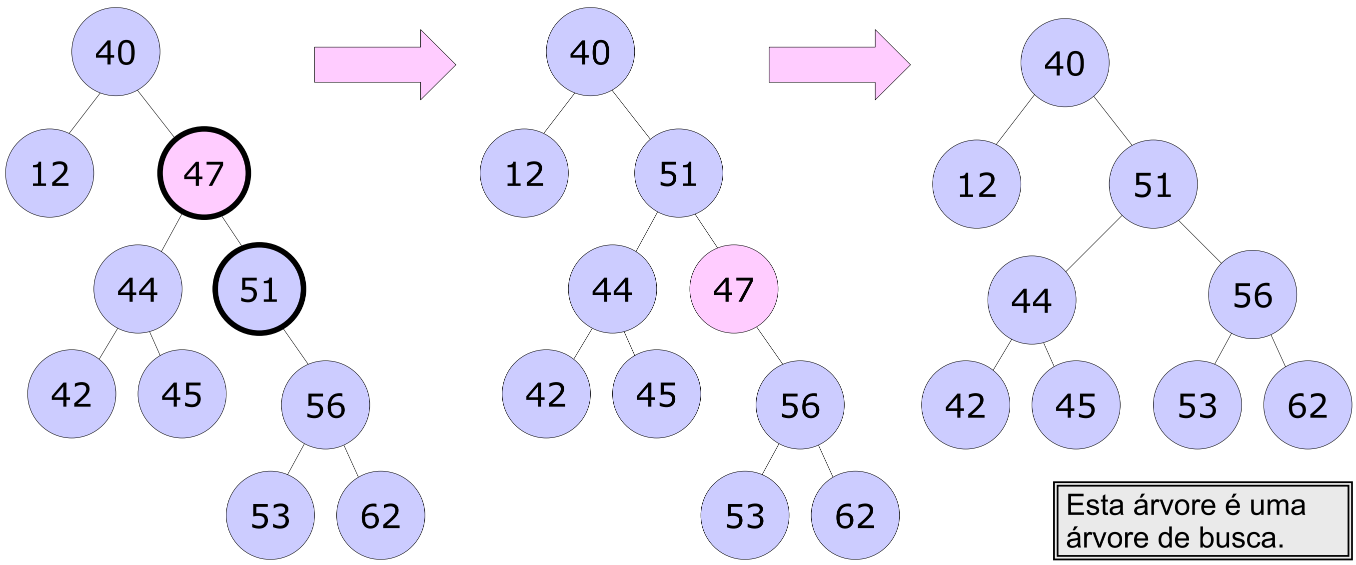
Para apagar um nó com dois filhos troca-se o valor do nó com o valor do nó seguinte e depois apaga-se o nó seguinte. O nó seguinte de um nó com dois filhos ou é uma folha ou só tem um filho. Logo, pode apagar-se com a técnica da página anterior.

Exemplo: apagar o 32 (cujo seguinte é o 39):



Apagando (4)

Eis um exemplo em que o seguinte tem um filho: apagar o 47:



Ao trocar um valor com o seguinte, a árvore fica momentaneamente irregular, mas como logo a seguir apagamos um dos elementos trocados fica tudo bem outra vez.

Classe BinarySearchTreePoly...<T>

Deriva de TreePolymorphic<T> e redefine Put, Remove e Search:

```
template <class T>
class BinarySearchTreePolymorphic: public TreePolymorphic<T> {
public:
  BinarySearchTreePolymorphic();
  virtual ~BinarySearchTreePolymorphic();

  virtual void Put(const T& s);
  virtual void Remove();
  virtual void Search(const T& x);
};
```

Tudo o resto se herda directamente!

Procurar, pôr

Procurar é muito simples:

```
template <class T>
void BinarySearchTreePolymorphic<T>::Search(const T& x)
{
  BinarySearch(x);
  if (Item() != x)
    Reset();
}
```

Para pôr, primeiro procura-se binariamente e depois, não tendo encontrado, pendura-se do lado certo:

```
void BinarySearchTreePolymorphic<T>::Put(const T& x)
{
  if (Empty())
    TreePolymorphic<T>::Put(x);
  else
  {
    BinarySearch(x);
    if (Item() != x)
      Insert(x, Item() <= x);
  }
}
```

Não queremos duplicados numa árvore binária de busca.

Busca binária

A busca binária, que desce pela subárvore do lado certo até achar ou até não haver mais por onde procurar, é implementada numa função privada:

```
template <class T>
void BinarySearchTreePolymorphic<T>::BinarySearch(const T& x)
{
  Reset();
  if (Empty())
    return;
  Down(0);  // start at root
  while (Item() != x)
  {
    bool side = Item() <= x;
    if (!HasChild(side))
      break;
    Down(side);
  }
}
```

```
template <class T>
class BinarySearchTreePolymorphic: public TreePolymorphic<T> {
// …
protected:
  void BinarySearch(const T& x);
};
```


Remover

Se o nó não tiver filhos ou tiver um filho, usamos a função herdada; se tiver dois filhos trocamos com o seguinte, após o que o nó apontado pelo cursor (com o valor que queremos remover) não terá filhos ou terá um filho, pelo que podemos usar então a função herdada:

```
template <class T>
void BinarySearchTreePolymorphic<T>::Remove()
{
  TreeNodePolymorphic<T>* temp = Parent();
  if (CountChildren() <= 1)
    TreePolymorphic<T>::Remove();
  else
  {
    SwapWith(Next());
    TreePolymorphic<T>::Remove();
  }
  Goto(temp);
}
```

No final, o cursor aponta para o pai do nó removido.

Comportamento das árvores de busca

Procurar, pôr e apagar numa árvore binária de busca demora um tempo proporcional à altura da árvore.

Numa árvore *completa* com *n* nós, altura é *ln*(*n*).

Numa árvore completamante enviesada com *n* nós, a altura é *n*.

Se os *n* nós da árvore forem postos por ordem aleatória, em média a altura será *ln*(*n*).

Nem sempre podemos garantir que os nós chegam por ordem aleatória

Convém que as árvores binárias de busca estejam *equilibradas*, claro.

Árvores chanfradas (*splay trees*)

As árvores chanfradas (*splay trees*) são árvores binárias de busca auto-ajustáveis. Depois de uma operação de acesso, o elemento acedido é promovido até à raiz.

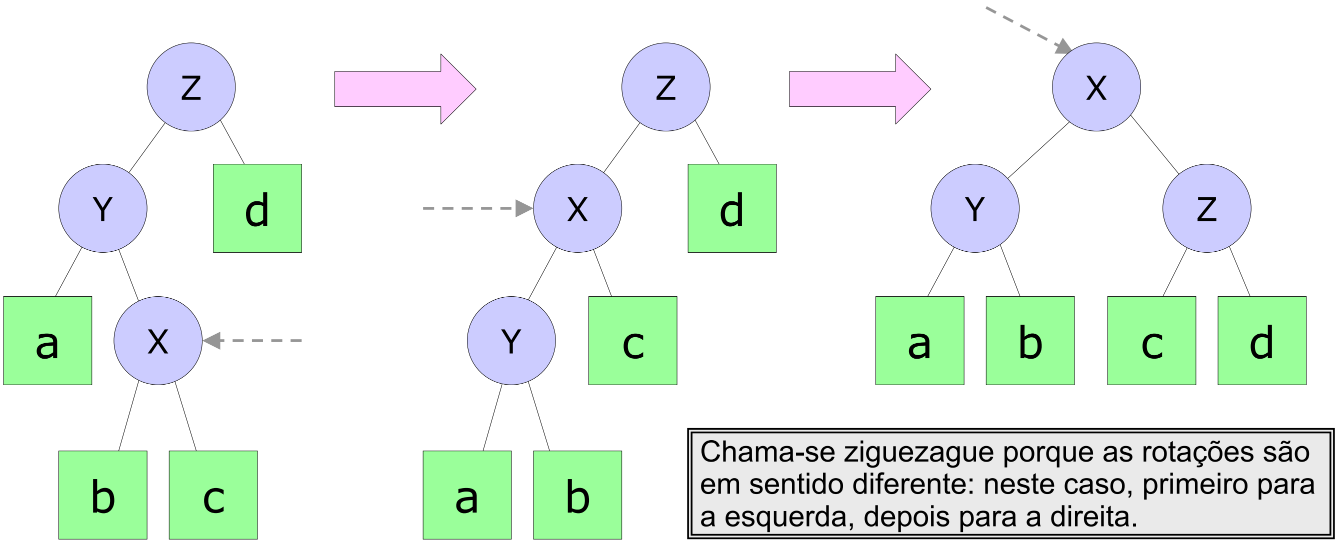
A promoção é feita de dois em dois níveis, excepto quando o nó está no primeiro nível.

Foram inventadas por Sleator e Tarjan em 1985. São uma das mais usadas estruturas de dados inventadas no últimos 20 anos. Sleator, professor na Universidade Carnagie-Mellon, e Tarjan, professor na Universidade de Princeton, receberam em o prémio Paris Kanellakis da Teoria e Prática em Informática (ACM) de 1999 pela sua invenção.

O prémio Kanellakis distingue contribuições teóricas que tiveram repercussão notável na prática.

Ziguezague

Quando filho e pai estão de lados diferentes, promove-se em ziguezague:



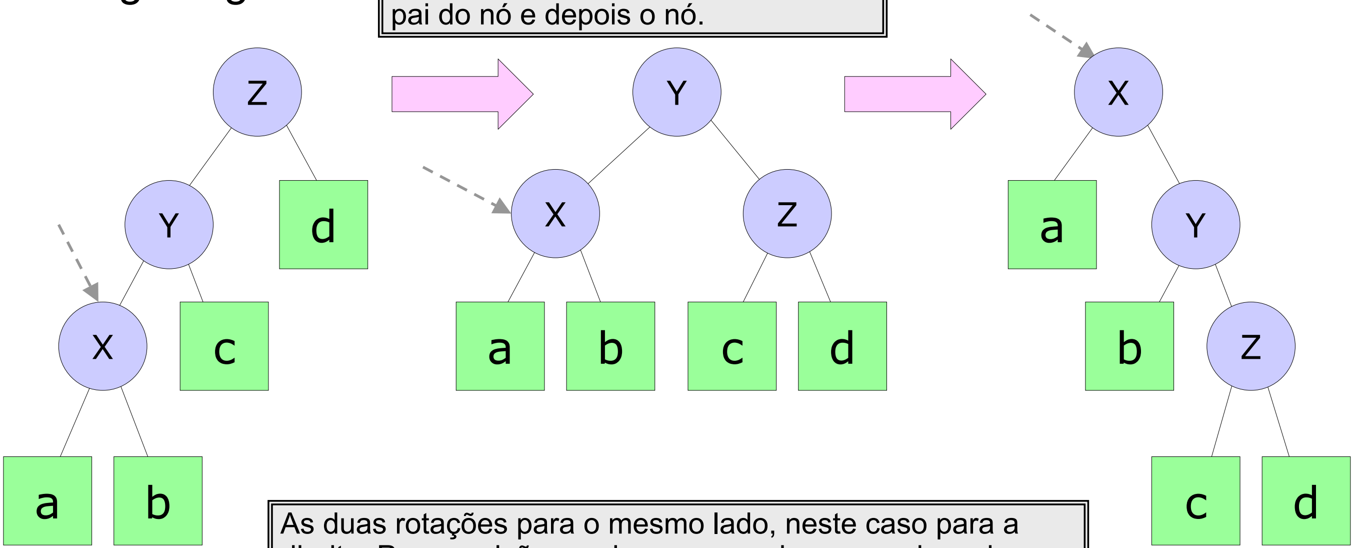
Chama-se ziguezague porque as rotações são em sentido diferente: neste caso, primeiro para a esquerda, depois para a direita.

Depois do ziguezague, a árvore está localmente equilibrada no nó promovido.

Ziguezigue

Quando filho e pai estão de lados diferentes, promove-se em "ziguezigue":

Note bem: primeiro promove-se o pai do nó e depois o nó.



As duas rotações para o mesmo lado, neste caso para a direita. Por oposição ao ziguezague, chama-se ziguezigue.

Depois do ziguezigue, as subárvores do nó promovido estão mais próximas da raiz.

Classe SplayTreePolymorphic<T>

Deriva de BinarySearchTreePolymorphic<T> e redefine Put, Remove e Search:

```
template <class T>
class SplayTreePolymorphic: public BinarySearchTreePolymorphic<T> {
public:
  SplayTreePolymorphic();
  virtual ~SplayTreePolymorphic();

  virtual void Put(const T& s);
  virtual void Remove();
  virtual void Search(const T& x);
};
```

Tudo o resto se herda de TreePolymorphic<T>!

Chanfrando

Chanfrar (*to splay*) um nó é promovê-lo até à raiz em ziguezague e em ziguezigue, excepto eventualmente a última promoção que poderá ser simples se o nível inicial era ímpar.

```
template <class T>
void SplayTreePolymorphic<T>::Splay()
{
  while (!AtRoot() && !AtFirstLevel())
    if (Side() == SideParent())
    {
      bool side = Side();
      Up();
      Promote();
      Down(side);
      Promote();
    }
    else
    {
      Promote();
      Promote();
    }
  if (AtFirstLevel())
    Promote();
}
```

Ziguezigue

Ziguezague

```
template <class T>
class SplayTreePolymorphic: public BinarySearchTreePolymorphic<T> {
public:
  // ...
private:
  virtual void Splay();
};
```

Splay é uma função privada.

Procurar e chanfrar

Ao procurar, se encontramos, chanframos o nó; se não encontramos, chanframos o pai “virtual” (isto é, o nó que seria o pai do nó encontrado se o valor procurado existisse):

```
template <class T>
void SplayTreePolymorphic<T>::Search(const T& x)
{
  BinarySearch(x);
  if (!Off())
    Splay();
  if (Root() != x)
    Reset();
}
```

BinarySearch é uma função protegida da classe BinarySearchTreePolymorphic<T>. Coloca o cursor no nó com o valor procurado ou deixa-o no pai “virtual”, se não houver.

Depois de BinarySearch a árvore só estará Off se for vazia. Por isso, se não tivermos encontrado, colocamos a árvore Off.

Pôr, remover e chanfrar

Depois de pôr, chanframos o nó, que assim fica na raiz:

```
template <class T>
void SplayTreePolymorphic<T>::Put(const T& x)
{
  BinarySearchTreePolymorphic<T>::Put(x);
  Splay();
}
```

Depois de remover, chanframos o pai do nó removido, se a árvore não tiver ficado vazia:

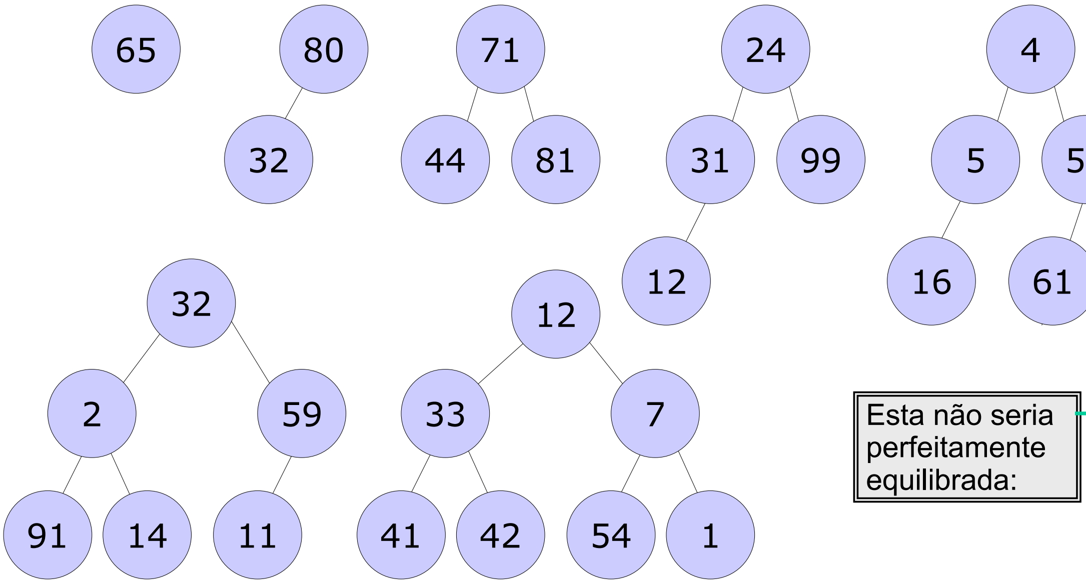
```
template <class T>
void SplayTreePolymorphic<T>::Remove()
{
  BinarySearchTreePolymorphic<T>::Remove();
  if (!Empty())
    Splay();
}
```

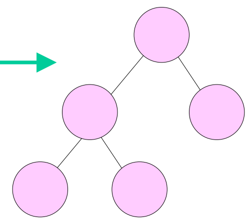
Ao remover na classe de base, o cursor fica a apontar para o pai do nó removido.

Árvores perfeitamente equilibradas

Uma árvore diz-se *perfeitamente equilibrada* se para cada nó o número de nós da suas subárvores diferir de 1 no máximo.

Exemplos:



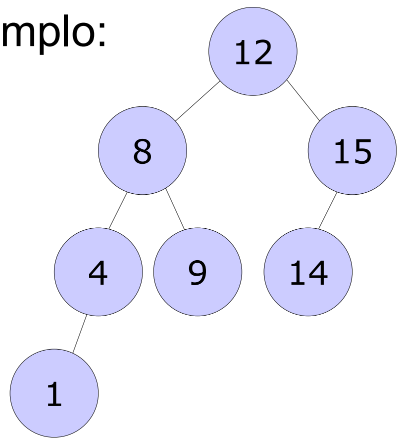


Árvores equilibradas, árvores AVL

Uma árvore diz-se *equilibrada* se para cada nó a altura da suas subárvores diferir de 1 no máximo.

Uma árvore AVL é uma árvore de busca equilibrada. “AVL” são as iniciais dos inventores, os matemáticos russos Adelson-Velskii e Landis (1962).

Exemplo:



Manter uma árvore equilibrada é menos complicado do que mantê-la perfeitamente equilibrada, ao pôr e ao remover.

Os nós das árvores AVL têm um membro que guarda a altura da subárvore cuja raiz é esse nó (ou a diferença da altura com a subárvore “irmã”).

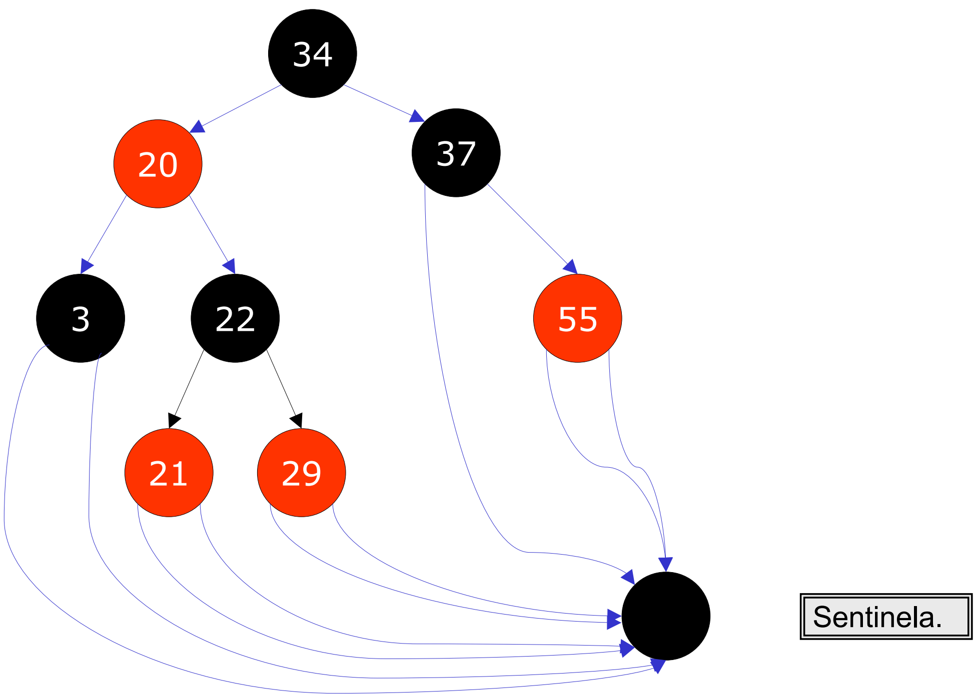
Árvores rubinegras (*red-black*)

Um árvore rubinegra (*red-black tree*) é uma árvore de busca binária em que cada nó tem uma de duas cores: vermelho ou preto. Restringindo as maneiras de colorir os nós ao longo dos caminhos da raiz até às folhas, garante-se que nenhum caminho tem mais do dobro dos nós do que qualquer outro. Assim, a árvore fica "aproximadamente" equilibrada.

As árvores rubinegras verificam as seguintes propriedades:

1. Cada nó ou é vermelho ou é preto.
2. A raiz é preta.
3. A sentinela é preta.
4. Se um nó é vermelho, os seus dois filhos são pretos.
5. Todos os caminhos desde um nó até à sentinela têm o mesmo número de nós pretos.

Árvores rubinegras, exemplo

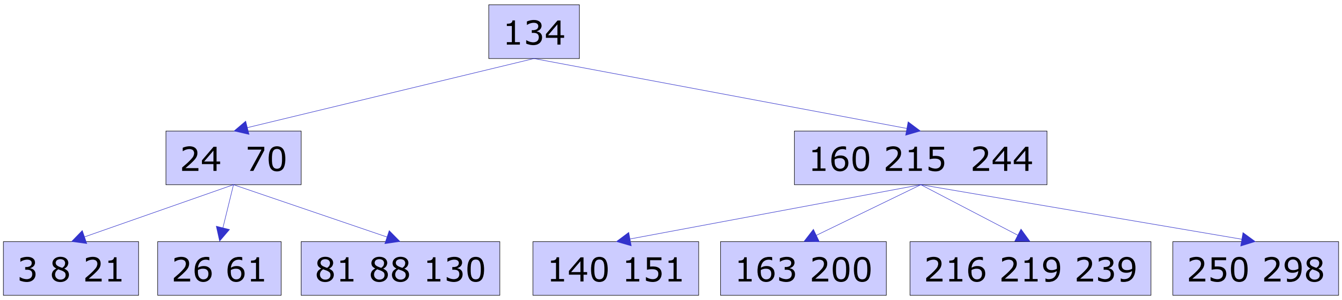


Bárvores (*B-trees*)

Uma bárvore (*B-tree*) é uma árvore de busca equilibrada, desenhada para ter um bom desempenho em dispositivos de memória secundária de acesso directo.

Os nós das bárvores podem ter muitos filhos (e não apenas dois).

As bárvores são uma generalização das árvores binárias de busca, como mostra o exemplo:



Bárvores, definição

Numa bárvore, cada nó tem vários valores e vários filhos. Os nós que são folhas não têm filhos. Os nós internos que tenham n valores têm n+1 filhos.

O número mínimo de filhos num nó interno de uma árvore é o grau mínimo da árvore. Se grau mínimo for t cada nó tem pelo menos t-1 valores. Numa bárvore de grau t, o número máximo de valores num nó é 2t-1 e, portanto, o número máximo de filhos é 2t.

Os valores num nó estão ordenados e separam os valores das subárvores correspondentes.

Ao pôr e ao remover, é preciso garantir que as propriedades da bárvore se mantêm.

Dicionários

A classe abstracta Dictionary<K, T> representa distribuidores cujos elementos de tipo T são identificados por *chaves* de tipo K.

```
template <class K, class T>
class Dictionary: public Dispenser<T> {
public:
  virtual ~Dictionary();

// from Container<T>
  virtual void Put(const T& x);

// declared now
  virtual void PutAt(const T& value, const K& key) = 0;  // pre ValidKey(k);
  virtual void RemoveAt(const K& key) = 0;  // pre ValidKey(k);
  virtual void Search(const K& key) = 0;   // pre ValidKey(k);
  virtual void Search(const K& key, const T& item) = 0;   // pre ValidKey(k);
  virtual bool Inserted() const = 0;
  virtual bool Removed() const = 0;
  virtual K Key(const T& x) const;
  virtual bool ValidKey(const K& key) const;
};
```

Algumas funções têm implementações por defeito.

Implementando Dictionary<K, T>

A chave é válida se não for igual ao valor por defeito do tipo K:

```
template <class K, class T>
bool Dictionary<K, T>::ValidKey(const K& key) const
{
  return !(key == K());
}
```

Por defeito, a chave obtém-se usando um construtor da classe K:

```
template <class K, class T>
K Dictionary<K, T>::Key(const T& x) const
{
  return K(x);
}
```

Se estes construtores não existirem na classe K, as funções terão de ser especializadas no ficheiro de instanciação.

Ao pôr, usa-se a chave:

```
template <class K, class T>
void Dictionary<K, T>::Put(const T& x)
{
  PutAt(x, Key(x));
}
```

Tabelas de dispersão, *hash tables*

Uma tabela de dispersão é um dicionário em que os elementos são guardados num vector, numa posição que é função da chave. As chaves são guardadas num outro vector, em posições paralelas. O elemento distinto (as tabelas de dispersão são distribuidores) é referenciado por um cursor:

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
private:
  VectorPolymorphic<K> keys;
  VectorPolymorphic<T> items;
  int count;
  int cursor;
  // ...
};
```

Nesta tabela simples, não vamos admitir duplicados, isto é, elementos distintos com a mesma chave.

Função de dispersão

Quando prevemos que objectos de uma classe vão ser chaves de outros numa tabela de dispersão, equipamos essa classe com uma função de dispersão. Exemplo: classe StringBasic:

```
class StringBasic: public Clonable {
// ...
public:
  // ...
 virtual int Hash() const;
  // ...
};
```

```
int StringBasic::Hash() const
{
  int result = 0;
  int i = 0;
  int count = MyUtil::Min(Count(), 127);
  while (i < count)
  {
    int p = 0;
    for (int j = 0; j < 3 && i < count; j++)
      p += 256 * p + static_cast<unsigned char>(At(i++));
    result += p;
  }
  return result;
}
```

Agrupam-se os caracteres 3 a 3 a partir da direita. Cada grupo dá origem a um número calculado a partir do valor numérico dos caracteres com peso 1, 256 e 65536. Soma-se tudo.

Dispersando a chave

Se a tabela tem capacidade N, a chave k dispersa para o índice k.Hash()%N.

Verbo "dispersar": eu disperso, tu dispersas, ele ou ela dispersa,...

Esta operação é realizada por uma função privada na classe Hashtable<K, T>, a qual posiciona o cursor na posição calculada:

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
// ...
private:
  virtual void Hash(const K& key);
};
```

```
template <class K, class T>
void HashtableSimple<K, T>::Hash(const K& key)
{
  // if class K does not have a Hash function, redefine
  // this function in the instantiation file.
  cursor = key.Hash() % Capacity();
}
```


Colisões

Duas chaves diferentes podem dispersar para a mesma posição, claro. Isso chama-se uma *colisão*.

Como se resolvem os conflitos causados pelas colisões?

1. Usando a técnica dos encadeamentos separados (*separate chaining*): os elementos com chaves colididas ficam na mesma lista. O vector dos elementos passa a ser um vector de listas, assim, por exemplo:

```
template <class K, class T>
class HashtableSeparate: public Dictionary<K, T>{
private:
  VectorPolymorphic<K> keys;
  VectorPolymorphic<ListDoublePolymorphic<T> > items;
  // ...
};
```

É esta técnica que vamos ilustrar com a nossa classe HashtableSimple<K, T>.

2. Usando a técnica do endereçamento aberto (*open addressing*): redispersa-se para uma outra posição livre.

Sondagem linear

Quando uma chave dispersa para uma posição já ocupada, escolhe-se a posição livre imediata.

Esta operação é realizada pela função Search:

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
// ...
public:
  virtual void Search(const K& key);
};
```

Inicialmente, todas as posições de um vector polimórfico estão indefinidas, isto é, contêm o apontador zero.

```
template <class K, class T>
void HashtableSimple<K, T>::Search(const K& key)
{
  Hash(key);
  while (keys.DefinedIndex(cursor) && keys[cursor] != key)
    ++cursor %= Capacity();
}
```

Se houver pelo menos uma posição livre (isto é, com o apontador zero), este ciclo tem de terminar.

Pôr na tabela

Para pôr na tabela, primeiro procura-se a posição de dispersão; se for uma posição livre, põe-se; se não for, é porque a chave já existe, e nesse caso não se põe:

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
// ...
public:
  virtual void PutAt(const T& value, const K& key);
  virtual bool Inserted() const;
  //...
};
```

```
template <class K, class T>
void HashtableSimple<K, T>::PutAt(const T& x, const K& key)
{
  inserted = false;
  Search(key);
  if (Off())
  {
    count++;
    keys.PutAt(key, cursor);
    items.PutAt(x, cursor);
    inserted = true;
  }
}
```

A tabela está Off quando o cursor está numa posição vazia.

```
template <class K, class T>
bool HashtableSimple<K, T>::Inserted() const
{
  return inserted;
}
```

O membro de dados inserted regista se o elemento foi inserido.

Note bem: se a chave já existir, a tabela não insere, pois não estará Off. (Recorde que queremos uma tabela sem duplicados.) Depois de inserir a tabela fica !Off.

Factor de carga

Quanto menos preenchida estiver a tabela, mais perto da posição de dispersão inicial ficará a chave.

O grau de preenchimento é medido pelo factor de carga.

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
// ...
public:
  virtual double LoadFactor() const;
  //...
};
```

```
template <class K, class T>
double HashtableSimple<K, T>::LoadFactor() const
{
  return static_cast<double>(count) / Capacity();
}
```

O static_cast é necessário para que o numerador seja um número real e assim termos a divisão exacta.

Factor de carga máximo

O factor de carga não deve ultrapassar um certo limite, ou a tabela deixa de ter um funcionamento interessante.

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
private:
  // ...
  double maxLoadFactor;
public:
  HashtableSimple(int capacity, double maxLoadFactor = 0.8);
};
```

O construtor indica a capacidade e o factor de carga máximo. Nesta tabela simples, estes valores são fixos.

Usemos 80% como valor máximo para o factor de carga, por defeito.

Consideramos que a tabela está cheia se o factor de carga máximo tiver sido atingido:

```
template <class K, class T>
bool HashtableSimple<K, T>::Full() const
{
  return count >= Capacity() * maxLoadFactor;
}
```

Simple tables do not remove

Remover numa tabela simples é complicado. Não basta remover a chave, pois ficaria um buraco numa sequência de chaves colididas que estragaria a busca. O melhor é não remover:

```
template <class K, class T>
void HashtableSimple<K, T>::Remove()
{
  // these simple tables do not remove
}

template <class K, class T>
void HashtableSimple<K, T>::RemoveAt(const K& key)
{
  // these simple tables do not remove
}
```

```
template <class K, class T>
bool HashtableSimple<K, T>::Removed() const
{
  return false;
}
```

Note bem: para a tabela ser um dicionário, tem de ter a função Remove (a qual terá como precondição !Off( ) ou uma condição mais fraca). Por outro lado, o efeito da função Remove é arbitrário, pois na classe Dispenser<T> a função não tem nenhuma pós-condição. No caso desta classe, não faz nada

Construtor, destrutor, Clear

O construtor inicializa os vectores, etc.:

```
template <class K, class T>
HashtableSimple<K, T>::
HashtableSimple(int capacity, double maxLoadFactor):
  keys(capacity),
  items(capacity),
  count(0),
  cursor(0),
  inserted(false),
  maxLoadFactor(maxLoadFactor)
{
}
```

O destrutor apenas remete automaticamente para os destrutores dos vectores:

```
template <class K, class T>
HashtableSimple<K, T>::~HashtableSimple()
{
}
```

A função Clear esvazia a tabela.

```
template <class K, class T>
void HashtableSimple<K, T>::Clear()
{
  keys.Clear();
  items.Clear();
  count = 0;
  cursor = 0;
  inserted = false;
}
```

Item, Off, Reset

Estas são todas muito simples:

```
template <class K, class T>
const T& HashtableSimple<K, T>::Item() const
{
  return items[cursor];
}

template <class K, class T>
T& HashtableSimple<K, T>::Item()
{
  return items[cursor];
}
```

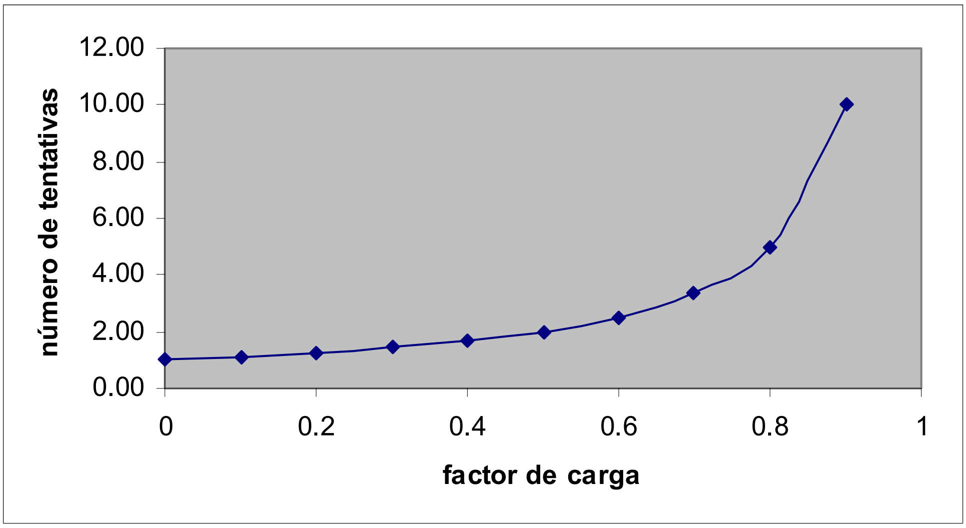
```
template <class K, class T>
bool HashtableSimple<K, T>::Off() const
{
  return !keys.DefinedIndex(cursor);
}
```

```
template <class K, class T>
void HashtableSimple<K, T>::Reset()
{
  while (keys.DefinedIndex(cursor))
    ++cursor %= Capacity();
}
```

A função Reset procura circularmente uma posição livre, que encontrará.

Desempenho das tabelas de dispersão

O número de passos de uma busca numa tabela de dispersão de endereçamento aberto com sondagem linear depende apenas do factor de carga. Se o factor de carga for *f*, esse número é 1/(1-*f*).



Aglomeração primária

Considere, como exemplo, uma tabela com capacidade 7, inicialmente vazia. Ao chegar a primeira chave, a probabilidade de ela vir a preencher a posição 3, por exemplo, é igual à de ela preencher qualquer outra posição, ou seja é 1/7. Suponhamos que ela preenche a posição 3.

Agora chega a segunda chave. A probabilidade de esta vir a preencher a posição 3 é zero, porque a posição 3 já está ocupada. Mas a probabilidade de vir a preencher a posição 4 é 2/7, a soma da probabilidade de a posição de dispersão calculada ser 3 com a probabilidade de ser 4.

Quer dizer, a probabilidade de uma nova chave ficar a seguir a uma posição ocupada é maior do que a probabilidade média. Logo, há uma certa tendência indesejável para as chaves se aglomerarem no vector das chaves, em vez de terem uma distribuição uniforme.

Dupla dispersão

Evita-se a aglomeração primária usando a dupla dispersão: em vez de tentar apanhar uma posição livre linearmente na tabela incrementa-se de um valor obtido por uma segunda função de dispersão.

```
template <class K, class T>
class HashtableSimple: public Dictionary<K, T>{
private:
  // ...
  int hash2;
  // ...
};
```

```
template <class K, class T>
void HashtableSimple<K, T>::Search(const K& key)
{
  Hash(key);
  while (keys.DefinedIndex(cursor) && keys[cursor] != key)
    cursor = (cursor + hash2) % Capacity();
}
```

O resultado da função de dispersão secundária é guardado em hash2.

O valor de hash2 é calculado na função Hash.

A capacidade deve ser prima

Para a função Search tentar sempre posições diferentes, é preciso que hash2 e Capacity() sejam primos entre si. Isso consegue-se fazendo hash2 menor do que Capacity() e escolhendo um número primo para a capacidade.

Eis a nova função Hash:

```
template <class K, class T>
void HashtableSimple<K, T>::Hash(const K& key)
{
  int temp = key.Hash();  // if K does not have function Hash, redefine in instantiation file.
  cursor = temp % Capacity();
  hash2 = Capacity() - 2 - temp % (Capacity() - 2);
}
```

Esta é a função sugerida por Sedgewick, no livro *Algorithms* (ISBN 0-201-06672-6), página 207.

Especializando a função de dispersão

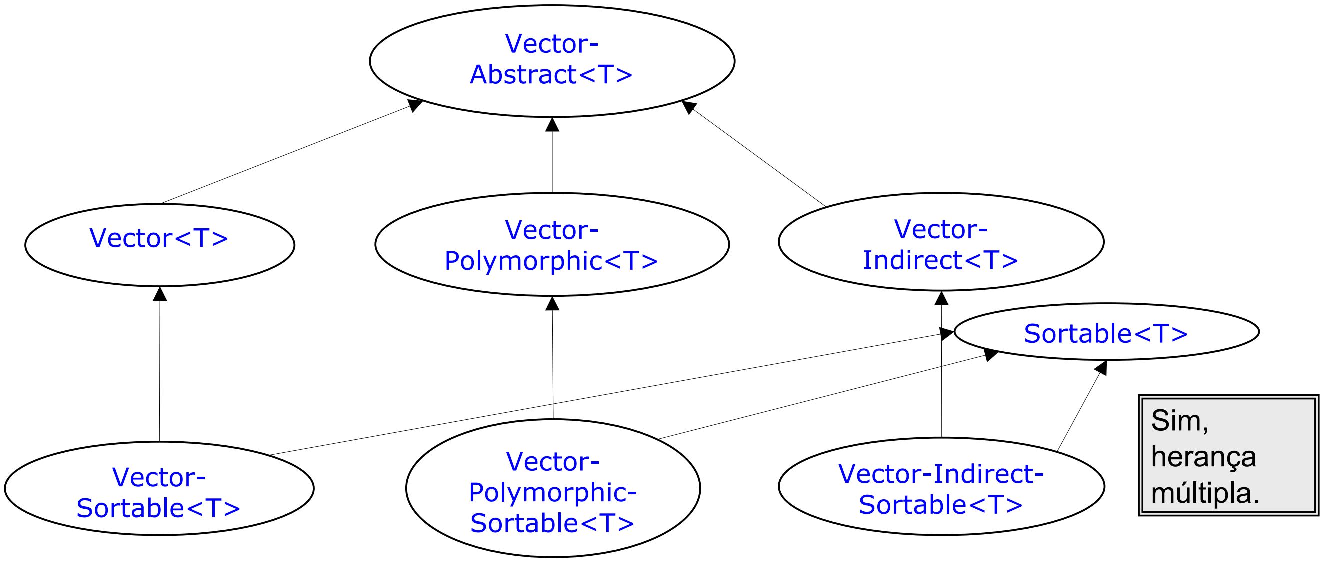
Se tivermos uma tabela com chaves inteiras, temos de especializar a função Hash no ficheiro de instanciação:

```
#include <iostream>

#include "Clonable.h"
#include "StringBasic.h"

#include "Dictionary.cpp"
template class Dictionary<int, StringBasic>;

#include "HashtableSimple.cpp"
template class HashtableSimple<int, StringBasic>;

int Dictionary<int, StringBasic>::Key(const StringBasic& x) const
{
  return static_cast<int>(x.First());
}

void HashtableSimple<int, StringBasic>::Hash(const int& key)
{
  cursor = key % Capacity();
  hash2 = Capacity() - 2 - key % (Capacity() - 2);
}
```

Neste exemplo, as chaves são inteiros e os elementos são strings. A chave é o valor numérico da primeira letra. (Logo, não haverá nesta tabela duas strings que comecem pela mesma letra.)

Ordenação

As operações de ordenação são agrupadas num classe abstracta Sortable<T>. Quem precisar de ordenar, herda de Sortable<T>:



Classe Sortable<T>

A classe Sortable<T> tem os algoritmos de ordenação básicos — bubblesort e quicksort — programados genericamente e geralmente. Estes algoritmos são acessíveis através das funções SortStable e Sort.

```
template <class T>
class Sortable: public Container<T> {
public:
  virtual ~Sortable();

  virtual void Sort();   // post IsSorted();
  virtual void SortGeneral(const Order<T>& newOrder);   // post IsSorted();
  virtual void SortStable();   // post IsSorted();
  virtual void SortStableGeneral(const Order<T>& newOrder);   // post IsSorted();
  virtual bool IsSorted() const;
private:
  virtual void Bubblesort(int lowerBound, int upperBound);
  virtual void Quicksort(int lowerBound, int upperBound);
};
```

Ordenação geral

Os algoritmos de ordenação são independentes da função de ordenação. Esta é um membro privado. As funções Sort e SortStable usam-no directamente. As funções SortGeneral e SortStableGeneral redefinem-no e depois chamam as outras:

```
template <class T>
class Sortable: public Container
private:
  Order<T>* order;
public:
  // ...
};
```

```
template <class T>
void Sortable<T>::SortGeneral(const Order<T>& newOrder)
{
  if (order != &newOrder)
  {
    delete order;
    order = dynamic_cast<Order<T>*>(newOrder.Clone());
  }
  Sort();
}
```

```
template <class T>
void Sortable<T>::SortStableGeneral(const Order<T>& newOrder)
{
  if (order != &newOrder)
  {
    delete order;
    order = dynamic_cast<Order<T>*>(newOrder.Clone());
  }
  SortStable();
}
```


Ordenação estável e não estável

Um algoritmo de ordenação é estável se elementos “iguais” de acordo com a função de ordem mantêm as suas posições relativas (isto é, não são trocados).

O bubblesort é estável, o quicksort não é estável:

```
template <class T>
void Sortable<T>::SortStable()
{
  if (Count() > 1)
    Bubblesort(0, Count() - 1);
}
```

```
template <class T>
void Sortable<T>::Sort()
{
  if (Count() > 1)
    Quicksort(0, Count() - 1);
}
```

Classe Order<T>

A função de ordenação é representada por um objecto de uma classe derivada da classe abstracta Order<T>. Tais objectos representam funções: são *objectos funcionais*.

```
template <class T>
class Order: public Clonable {
public:
  virtual ~Order();

  virtual bool Equal(const T& x, const T& y) const;
  virtual bool NotEqual(const T& x, const T& y) const;
  virtual bool LessThan(const T& x, const T& y) const;
  virtual bool LessThanStrict(const T& x, const T& y) const;
  virtual bool GreaterThan(const T& x, const T& y) const;
  virtual bool GreaterThanStrict(const T& x, const T& y) const;

  virtual bool operator ()(const T& x, const T& y) const;
};
```

Todas estas funções têm implementação. A classe fica abstracta porque não define a função Clone, herdada.

Implementando Order<T> **(1)**

A igualdade e a menoridade são programadas em termos dos operadores == e <= da classe T. As restantes são ficam em termos dessas:

```
template <class T>
bool Order<T>::Equal(const T& x, const T& y) const
{
  return x == y;
}

template <class T>
bool Order<T>::LessThan(const T& x, const T& y) const
{
  return x <= y;
}
```

Assim, nas classes derivadas basta redefinir estas duas. As outras herdam-se e o polimorfismo faz o resto.

Implementando Order<T> (2)

```
template <class T>
bool Order<T>::NotEqual(const T& x, const T& y) const
{
  return !Equal(x, y);
}

template <class T>
bool Order<T>::LessThanStrict(const T& x, const T& y) const
{
  return LessThan(x, y) && NotEqual(x, y);
}

template <class T>
bool Order<T>::GreaterThan(const T& x, const T& y) const
{
  return LessThan(y, x);
}

template <class T>
bool Order<T>::GreaterThanStrict(const T& x, const T& y)
     const
{
  return !LessThan(x, y);
}

template <class T>
bool Order<T>::operator ()(const T& x, const T& y) const
{
  return LessThan(x, y);
}
```

O operador () é equivalente à função LessThan.

Ordem simples

A ordem simples, OrderSimple<T>, é a ordem não abstracta associada automaticamente a um tipo que tenha os operadores == e <=. Deriva de Order<T>:

```
template <class T>
class OrderSimple: public Order<T> {
public:
  virtual ~OrderSimple();
  virtual Clonable* Clone() const;
};
```

Para não ficar abstracta, basta implementar a função Clone. Tudo resto é herdado:

```
template <class T>
OrderSimple<T>::~OrderSimple()
{
}

template <class T>
Clonable* OrderSimple<T>::Clone() const
{
  return new OrderSimple<T>(*this);
}
```

Nestas classes simples, que não têm membros de dados, não nos damos ao trabalho de programar os construtores (por defeito, de cópia), confiando nos que o C++ gera automaticamente.

Exemplo: ordenando números inteiros

```
void TestSortRandomIntegers()
{
  Vector<int>::SetPrefixSuffix(" ", "");
  Random::Reset();
  VectorPolymorphicSortable<int> v(100);

  v.Vector<int>::PutItems(Random(1000, 16));
  v.WriteLine();
  v.Sort();
  v.WriteLine();
}
```

16 números aleatórios entre 0 e 1000.

A classe Random é um iterador de inteiros:

```
class Random: public Iterator<int> {
private:
  // …
public:
  explicit Random(int max = RAND_MAX, int size = std::numeric_limits<int>::max());
public: // static
  static void Reset();
  // …
};
```

Reinicializa a semente aleatoriamente.

Ordenando descartavelmente

Por vezes usamos uma classe descartável para representar a função de ordem que queremos usar. Exemplo: ordenar um vector de números pela soma dos algarismos:

```
class ByDigits: public Order<int>{
public:
  virtual bool LessThan(const int& x, const int& y) const
  {
    return Sum(x) < Sum(y) || Sum(x) == Sum(y) && x <= y;
  }
  virtual Clonable* Clone() const
  {
    return new ByDigits(*this);
  }
private:
  static int Sum(int x)
  {
    return x == 0 ? 0 : x % 10 + Sum(x / 10);
  }
};
```

Esta classe poderia aparecer no ficheiro onde vai ser usada, em vez de constituir uma classe autónoma, com o seu ficheiro ponto-agá.

Baste redefinir a função LessThan. Note que LessThan(x, x) vale true, sempre.

Repare que as funções são definidas na classe.

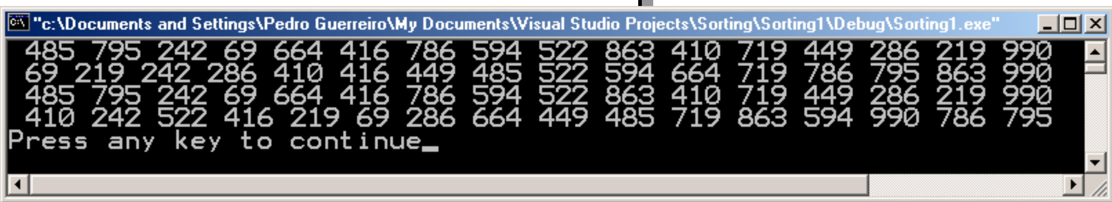
Usando a função de ordenação

Eis a ordenação pela soma dos dígitos, usando a função de ordenação:

```
void TestSortRandomIntegers()
{
  VectorPolymorphicSortable<int>::SetPrefixSuffix(" ", "");
  Random::Reset();
  VectorPolymorphicSortable<int> v(100);

  v.VectorPolymorphic<int>::PutItems(Random(1000, 16));
  VectorPolymorphicSortable<int> v2(v);
  v.WriteLine();
  v.Sort();
  v.WriteLine();
  v = v2;
  v.WriteLine();
  v.SortGeneral(ByDigits());
  v.WriteLine();
}
```

Repare no construtor de cópia e na operação de afectação na classe VectorPolymorphic<T>.

Repare na forma da chamada da função PutItems, para desambiguar, por causa da herança múltipla.



Um exemplo com cadeias

```
void TestSortStrings()
{
  VectorPolymorphicSortable<StringBasic>::SetPrefixSuffix(" ", "");
  VectorPolymorphicSortable<StringBasic> v(100);

  v.Put("kkkkkf");
  v.Put("zzza");
  v.Put("bbbbbbbbc");
  v.Put("fffe");
  v.Put("xxd");
  v.Put("aaaah");
  v.Put("nnnnb");
  VectorPolymorphicSortable<StringBasic> v2(v);

  v.WriteLine();
  v.Sort();
  v.WriteLine();
  v = v2;
  v.WriteLine();
  v.SortGeneral(ByLength());
  v.WriteLine();
}
```

Ordenação não estável (quicksort) simples.

Ordenação por comprimento.

Ordenação por comprimento

Usamos uma classe descartável para representar a função de ordenação por comprimento:

```
class ByLength: public Order<StringBasic>{
public:
  virtual bool LessThan(const StringBasic& x, const StringBasic& y) const
  {
    return x.Count() < y.Count() || x.Count() == y.Count() && x <= y;
  }
  virtual Clonable* Clone() const
  {
    return new ByLength(*this);
  }
};
```

Herdamos sempre de Order<T>, e não de OrderSimple<T> para não corrermos o risco de nos esquecermos de programar a função Clone. (Se nos esquecermos, a classe fica abstracta.)

Resultado:

```
"c:\Documents and Settings\Pedro Guerreiro\My Documents\Visual Studio Project...
 kkkkkf zzza bbbbbbbbbc fffe xxd aaaah nnnnb
 aaaah bbbbbbbbbc fffe kkkkkf nnnnb xxd zzza
 kkkkkf zzza bbbbbbbbbc fffe xxd aaaah nnnnb
 xxd fffe zzza aaaah nnnnb kkkkkf bbbbbbbbbc
Press any key to continue
```


O bubblesort genérico geral

Ei-lo:

```
template <class T>
void Sortable<T>::Bubblesort(int lowerBound, int upperBound)
{
  for (int i = 1; i < Count(); i++)
  {
    bool swapped = false;
    for (int j = Count() - 1; j >= i; j--)
      if (!order->LessThan(At(j-1), At(j)))
      {
        Swap(j-1, j);
        swapped = true;
      }
    if (!swapped)
      return;
  }
}
```

Repare na maneira de usar o objecto funcional que representa a função de ordenação.

Usamos a técnica da variável booleana para antecipar a saída, logo que não haja trocas numa passagem.

Análise do bubblesort

Se o vector tiver *N* elementos, na primeira passagem há *N*-1 comparações, na segunda há *N*-2, etc., e na última há 1. Ao todo há (*N*-1) + (*N*-2) + ... + 1, ou seja *N**(*N*-1)/2 comparações.

A expressão *N**(*N*-1)/2 é equivalente a $N^2/2 - N/2$. Para valores de *N* grandes, o segundo termo é desprezável face ao primeiro.

Dizemos que o bubblesort é um algoritmo quadrático, pois o tempo de execução é proporcional ao quadrado do tamanho do vector.

Se o tamanho aumentar para o dobro, o tempo de execução aumenta para o quádruplo; se o tamanho aumentar 10 vezes, o tempo aumenta 100 vezes. (Experimente para ver se é mesmo assim.)

O quicksort genérico geral

O quicksort é uma das obras-primas da programação. Esta é a versão original de Hoare, tal como publicada por Wirth em *Algorithms + Data Structures = Programs*, adaptada para C++, genérico e geral:

Se precisar, veja a sebenta de Programação I, página 93.

```
template <class T>
void Sortable<T>::Quicksort(int lowerBound, int upperBound)
{
  int i = lowerBound;
  int j = upperBound;
  const T p(At((i+j)/2));
  do
  {
    while (order->LessThanStrict(At(i), p))
      i++;
    while (order->GreaterThanStrict(At(j), p))
      j--;
    if (i <= j)
      Swap(i++, j--);
  } while (i <= j);
  if (lowerBound < j)
    Quicksort(lowerBound, j);
  if (i < upperBound)
    Quicksort(i, upperBound);
}
```

Note que o pivô é inicializado pelo construtor de cópia do tipo T, e não por clonagem. A função de ordenação usada é a guardada no vector, e não o operador <= no tipo T.

Repare na utilização de outras funções da classe Order<T>.

Descrição do quicksort

O ciclo do-while corresponde à fase de partição. O vector é "partido" em duas partes, de maneira a que todos os elementos da primeira parte sejam menores que todos os elementos da segunda parte. Depois aplica-se o mesmo esquema a cada uma das partes, enquanto tiverem mais do que um elemento.

Esta função Partition (não programada) representa a fase de partição do quicksort.

```
template <class T>
void Sortable<T>::Quicksort(int lowerBound, int upperBound)
{
  int i;
  int j;
  Partition(lowerBound, upperBound, i, j);
  if (lowerBound < j)
    Quicksort(lowerBound, j);
  if (i < upperBound)
    Quicksort(i, upperBound);
}
```

Cuidado: i e j são parâmetros de saída.

Aqui, após a partição, temos j < i e, para quaisquer x e y, com x in [lowerBound, j] e y in [i, upperBound], é verdade que x <= y.

Análise do quicksort

O desempenho do quicksort depende de a partição ser equilibrada.

Seja um vector com *N* elementos.

Na primeira partição são visitados os *N* elementos, seguindo-se duas chamadas recursivas de primeiro nível, as quais em conjunto visitam todos os *N* elementos. Cada uma delas dá origem a duas chamadas recursivas de segundo nível. Há, portanto, quatro chamadas recursivas de segundo nível, as quais em conjunto visitam os *N* elementos. E assim por diante. Em cada nível são visitados os *N* elementos, até que, por os intervalos se tornarem vazios, algumas das chamadas recursivas deixem de se fazer. Nessa altura, em cada nível visitam-se menos de *N* elementos.

Quicksort com sorte

Num quicksort com sorte, a partição é perfeitamente equilibrada: em cada nível o intervalo é dividido ao meio. Todas as chamadas recursivas terminam ao mesmo nível (mais ou menos 1).

Nesse caso, o número de níveis de chamadas recursivas é $\log_2 N$ aproximadamente.

Logo, o procedimento exige $N \log N$ acessos aos elementos do vector.

> $\log N$ significa $\log_2 N$.

Esta é a melhor situação possível. O tempo de execução é proporcional a $N \log N$. ☺

$N \log N$ é melhor do que N^2 mas é menos bom do que N.

> Um vector já ordenado com pivô central dá origem a uma partição perfeitamente equilibrada.

Quicksort com azar

Num quicksort com azar, a partição é completamante desequilibrada: em cada nível um dos subintervalos tem 1 elemento e o outro N-1.

Neste caso há N níveis de recursividade, ainda que em cada nível haja apenas uma chamada recursiva, para a parte não unitária.

No nível k são visitados N-k elementos.

Ao todo, o procedimento desequilibrado exige $N + (N-1) + ... + 1$ acessos. Isto é um número da ordem de $N^2/2$.

Este é o caso mais desfavorável para o quicksort.

Aqui o desempenho é quadrático. ☹

> Um vector já ordenado com pivô inicial ou final dá origem a uma partição completamente desequilibrada. Por isso é que preferimos o pivô central. Seria muito triste gastar tempo quadrático para não fazer nada.

Partição equilibrada

Suponhamos que em cada partição 1/3 dos elementos fica numa parte e 2/3 noutra parte. Por exemplo, se *N* for 1000 a parte maior fica com 667, 444, 296, 197, 131, ..., 1, elementos. A parte menor fica com 333, 111, 37, ..., 1. A parte menor esgota-se a fim de $\log_3 N$ níveis. A parte maior esgota-se ao fim de $\log_{3/2} N$ níveis. Em cada nível até $\log_3 N$ há *N* acessos. A partir daí há menos de *N*. Logo ao todo há menos de $N \log_{3/2} N$ acessos, ou seja menos de $\log_{3/2}2 * N \log N$.

$\log_a x = \log_a b * \log_b x$

Logo, continuamos com um funcionamento proporcional a *N* log *N*, ainda que com uma constante de proporcionalidade maior.

Em geral, se pudermos garantir que a partição é equilibrada, o quicksort é *N* log *N*, seja qual for o factor de equilíbrio.

Melhorando o quicksort

Que há de errado no vector <2 4 6 7 1 5 3>?

Nada, mas se usarmos o quicksort com pivô central, vamos ter sempre partições completamente desequilibradas.

Como evitar estes casos patológicos?

1. Usar um pivô aleatório.
   Gera-se um número aleatório k no intervalo `[lowerBound..upperBound]`, troca-se `At(k)` com o elemento central e a partir daí é igual.

   Pouco efectivo.

2. Usar a mediana de três.
   O pivô ideal é a mediana. Não sendo prático obtê-la, contentamo-nos com a mediana dos elementos inicial, central e final. (Assim nunca teremos um desequilíbrio completo.)

   Ganho marginal.

Ordenação algorítmica

Se quisermos incorporar estas variantes no quicksort, ou programar outros algoritmos de ordenação, temos de mexer na classe para acrescentar novas funções Sort?

Não, se a classe estiver preparada com uma função de ordenação algorítmica, em que o argumento é um objecto representando o algoritmo de ordenação.

```
template <class T>
class Sortable: public Container<T> {
private:
  Order<T>* order;
public:
// …
  virtual void SortAlgoritmically(const SortingAlgorithm<T>& a);
};
```

Cada algoritmo será representado por uma classe derivada de SortingAlgorithm<T>.

Classe SortingAlgorithm<T>

A classe SortingAlgorithm<T> é uma classe de base com uma função Sort virtual que, por escolha nossa, implementa o quicksort:

```
template <class T>
class SortingAlgorithm {
public:
  virtual ~SortingAlgorithm();
  virtual void Sort(Sortable<T>& v) const;
};
```

```
template <class T>
SortingAlgorithm<T>::~SortingAlgorithm()
{
}

template <class T>
void SortingAlgorithm<T>::Sort(Sortable<T>& v) const
{
  v.Sort();
}
```

Exemplos já a seguir...

Claro que não repetimos o texto do quicksort. Apenas chamamos a função Sort da classe Sortable<T>.

Ordenação por selecção directa

Este é um algoritmo elementar: percorre-se o vector calculando o menor elemento; troca-se com o primeiro. Depois percorre-se novamente, mas a partir da segunda posição calculando o menor elemento; troca-se com o segundo. E assim por diante:

```
template <class T>
class SelectionSort: public SortingAlgorithm<T> {
public:
  virtual ~SelectionSort();
  virtual void Sort(Sortable<T>& v) const;
};
```

A função IndexOfMin calcula o índice do elemento mínimo no intervalo dos índices, em relação à função de ordenação do objecto.

```
template <class T>
void SelectionSort<T>::Sort(Sortable<T>& v) const
{
  for (int i = 0; i < v.Count(); i++)
    v.Swap(i, v.IndexOfMin(i, v.Count() - 1));
}
```


Selecção directa, exemplo

Eis uma função de teste que usa a ordenação directa num vector de StringBasic, com várias funções de ordenação:

```
void TestSelectionSort()
{
  VectorPolymorphicSortable<StringBasic>::SetPrefixSuffix(" ", "");
  VectorPolymorphicSortable<StringBasic> v(1000);
  v.Put("kkkkkf");
  v.Put("zzza");
  v.Put("bbbbbbbbc");
  v.Put("fffe");
  v.Put("xxd");
  v.Put("aaaah");
  v.Put("nnnnb");
  v.WriteLine();
  v.SortAlgorithmically(SelectionSort<StringBasic>());
  v.WriteLine();
  v.SetOrder(ByLength());
  v.SortAlgorithmically(SelectionSort<StringBasic>());
  v.WriteLine();
  v.SetOrder(ByReverse());
  v.SortAlgorithmically(SelectionSort<StringBasic>());
  v.WriteLine();
  v.SetOrder(OrderSimple<StringBasic>());
  v.SortAlgorithmically(SelectionSort<StringBasic>());
  v.WriteLine();
}
```

```
"c:\Documents and Settings\Pedro Guerreiro\My Documents\Visual Studio Proje...
 kkkkkf zzza bbbbbbbbc fffe xxd aaaah nnnnb
 aaaah bbbbbbbbc fffe kkkkkf nnnnb xxd zzza
 xxd fffe zzza aaaah nnnnb kkkkkf bbbbbbbbc
 zzza nnnnb bbbbbbbbc xxd fffe kkkkkf aaaah
 aaaah bbbbbbbbc fffe kkkkkf nnnnb xxd zzza
Press any key to continue_
```

Classe ByReverse: ver página seguinte.

Função SetOrder: ver página seguinte + 1.

Ordenação reversa

A classe ByReverse representa a ordenação reversa de StringBasic, isto é, as cadeias comparam-se do fim para o princípio:

```
class ByReverse: public Order<StringBasic>{
public:
  virtual bool LessThan(const StringBasic& x, const StringBasic& y) const
  {
    StringBasic x1(x);
    StringBasic y1(y);
    x1.Reverse();
    y1.Reverse();
    return x1 <= y1;
  }
  virtual Clonable* Clone() const
  {
    return new ByReverse(*this);
  }
};
```


Funções GetOrder, SetOrder

A classe Sortable<T> oferece as funções GetOrder e Set-Order para aceder à ordem privada:

```
template <class T>
class Sortable: public Container<T> {
private:
  Order<T>* order;
public:
  // ...
  virtual const Order<T>& GetOrder() const;
  virtual void SetOrder(const Order<T>& newOrder);
};
```

```
template <class T>
const Order<T>& Sortable<T>::GetOrder() const
{
  return *order;
}
```

```
template <class T>
void Sortable<T>::SetOrder(const Order<T>& newOrder)
{
  if (order != &newOrder)
  {
    delete order;
    order = dynamic_cast<Order<T>*>(newOrder.Clone());
  }
}
```

Repare, na página anterior, que v.GetOrder()(v.At(j), v.At(i)) é o mesmo que v.GetOrder().LessThan(v.At(j), v.At(i)).

Mínimos e máximos (1)

A classe Sortable<T> vem equipada com funções para obter o índice do mínimo e do máximo, de maneira geral, e também o mínimo e o máximo.

```
template <class T>
class Sortable: public Container<T> {
public:
  // ...
  virtual int IndexOfMin() const; // pre !Empty();
  virtual int IndexOfMax() const; // pre !Empty();
  virtual const T& Min() const;  // pre !Empty();
  virtual const T& Max() const;  // pre !Empty();

  virtual int IndexOfMin(int startPos, int endPos) const;  // pre ValidRange(startPos, endPos);
  virtual int IndexOfMax(int startPos, int endPos) const; // pre ValidRange(startPos, endPos);
  virtual const T& Min(int startPos, int endPos) const;    // pre ValidRange(startPos, endPos);
  virtual const T& Max(int startPos, int endPos) const;   // pre ValidRange(startPos, endPos);
```

Estas funções calculam usando a função de ordenação do objecto.

Mínimos e máximos (2)

Existe um outro grupo de funções em que a função de ordenação usada é passa por argumento, num objecto de tipo Order<T>. São as funções gerais:

```
template <class T>
class Sortable: public Container<T> {
public:
  // ...
  virtual int IndexOfMinGeneral(const Order<T>& order) const; // pre !Empty();
  virtual int IndexOfMaxGeneral(const Order<T>& order) const; // pre !Empty();
  virtual const T& MinGeneral(const Order<T>& order) const;  // pre !Empty();
  virtual const T& MaxGeneral(const Order<T>& order) const;  // pre !Empty();

  virtual int IndexOfMinGeneral(const Order<T>& order, int startPos, int endPos) const;
                                                // pre ValidRange(startPos, endPos);
  virtual int IndexOfMaxGeneral(const Order<T>& order, int startPos, int endPos) const;
                                                // pre ValidRange(startPos, endPos);
  virtual const T& MinGeneral(const Order<T>& order, int startPos, int endPos) const;
                                                // pre ValidRange(startPos, endPos);
  virtual const T& MaxGeneral(const Order<T>& order, int startPos, int endPos) const;
                                                // pre ValidRange(startPos, endPos);
```

Só estas duas é que trabalham realmente...

Índice do mínimo, do máximo, geral

Calculamos o índice do mínimo e o índice do máximo com a parametrização completa, usando os algoritmos costumeiros:

```
template <class T>
int Sortable<T>::IndexOfMinGeneral(const Order<T>& order, int startPos, int endPos) const
{
  int result = startPos;
  for (int i = startPos + 1; i <= endPos; i++)
    if (order.LessThanStrict(At(i), At(result)))
      result = i;
  return result;
}
```

```
template <class T>
int Sortable<T>::IndexOfMaxGeneral(const Order<T>& order, int startPos, int endPos) const
{
  int result = startPos;
  for (int i = startPos + 1; i <= endPos; i++)
    if (order.GreaterThanStrict(At(i), At(result)))
      result = i;
  return result;
}
```

As restantes funções baseiam-se nestas, directamente ou indirectamente.

Outros mínimos (1)

Índice do mínimo, com ordem paramétrica no vector todo:

```
template <class T>
int Sortable<T>::IndexOfMinGeneral(const Order<T>& order) const
{
  return IndexOfMinGeneral(order, 0, Count() - 1);
}
```

Índice do mínimo, com ordem privada num subvector:

```
template <class T>
int Sortable<T>::IndexOfMin(int startPos, int endPos) const
{
  return IndexOfMinGeneral(*order, startPos, endPos);
}
```

Índice do mínimo, com ordem privada no vector todo:

```
template <class T>
int Sortable<T>::IndexOfMin() const
{
  return IndexOfMinGeneral(*order);
}
```

Outros mínimos (2)

Para os máximos é a mesma coisa.

Mínimo com ordem paramétrica, num subvector:

```
template <class T>
const T& Sortable<T>::MinGeneral(const Order<T>& order, int startPos, int endPos) const
{
  return At(IndexOfMinGeneral(order, startPos, endPos));
}
```

Mínimo com ordem paramétrica, no vector todo:

```
template <class T>
const T& Sortable<T>::MinGeneral(const Order<T>& order) const
{
  return At(IndexOfMinGeneral(order));
}
```

Mínimo com ordem privada, num subvector:

```
template <class T>
const T& Sortable<T>::Min(int startPos, int endPos) const
{
  return At(IndexOfMin(startPos, endPos));
}
```

Mínimo com ordem privada, no vector todo:

```
template <class T>
const T& Sortable<T>::Min() const
{
  return At(IndexOfMin());
}
```


Ordenação por inserção directa

É o que usamos quando jogamos às cartas, para ordenar a nossa mão: a parte esquerda está ordenada e em cada passo retiramos um elemento da parte direita (não ordenada) e inserimo-lo ordenadamente na parte esquerda, que assim cresce, enquanto a parte direita diminui:

Para muitos, este é o algoritmo mais intuitivo.

```
template <class T>
void InsertionSort<T>::Sort(Sortable<T>& v) const
{
  for (int i = 1; i < v.Count(); i++)
  {
    int j = i - 1;
    while (j >= 0 && !(v.GetOrder()(v.At(j), v.At(i))))
      j--;
    v.RotateUp(j + 1, i);
  }
}
```

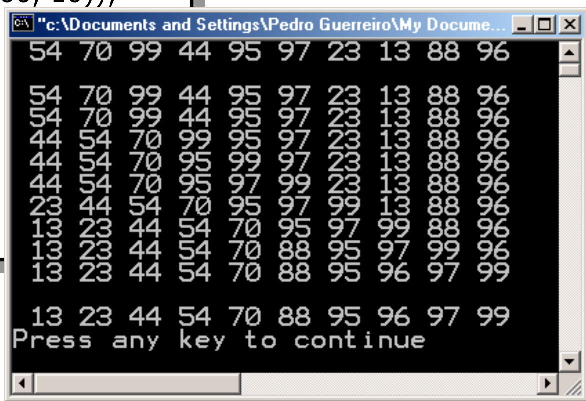
Percorremos o vector da direita para a esquerda e paramos quando encontramos um elemento fora de ordem com aquele de onde partimos.

A função RotateUp é herdada de Sortable<T> (onde é virtual pura). A chamada v.RotateUp(x, y) "roda" para cima os elementos do vector v entre os índices x e y. Por exemplo, se o v for <2 3 5 7 11 13 17>, depois de v.RotateUp(2, 5), v vale <2 3 13 5 7 11 17>.

Inserção directa, exemplo

Eis uma função de teste simples, que ordena um vector de inteiros. A função InsertionSort<T>::Sort foi instrumentada para mostrar o estado do vector após cada rotação:

```
void TestInsertionSort_int1()
{
  VectorPolymorphicSortable<int>::SetPrefixSuffix(" ", "");
  Random::Reset();
  VectorPolymorphicSortable<int> v(100);

  v.VectorPolymorphic<int>::PutItems(Random(100, 10));
  v.WriteLine();
  std::cout << std::endl;

  v.SortAlgorithmically(InsertionSort<int>());

  std::cout << std::endl;
  v.WriteLine();
}
```



Ordenação por troca directa

Este é o bubblesort. Em rigor, não precisávamos de o programar, pois ele já existe na classe Sortable<T>. Mas façamo-lo, para ficarmos com a colecção completa dos algoritmos de ordenação directa. Claro que o vamos buscar à classe Sortable<T>:

```
template <class T>
class Bubblesort: public SortingAlgorithm<T> {
public:
  virtual ~Bubblesort();
  virtual void Sort(Sortable<T>& v) const;
};
```

Não repetimos o código do bubblesort. Apenas o vamos buscar à classe Sortable<T>, onde é representado pela função SortStable:

```
template <class T>
Bubblesort<T>::~Bubblesort()
{
}

template <class T>
void Bubblesort<T>::Sort(Sortable<T>& v) const
{
  v.SortStable();
}
```

Comparação das ordenações directas

São todas quadráticas. No entanto, há diferenças no desempenho. Seja um vector com *N* elementos. O tempo de execução depende essencialmente do número de comparações *C* e do número de afectações no vector *A*. O seguinte quadro resume a situação, para o caso mais favorável, médio e para o caso mais favorável.

		Mínimo	Médio	Máximo
Inserção	C =	N-1	$(N^2+N-2)/4$	$(N^2-N)/2-1$
	A =	$2(N-1)$	$(N^2-9N-10)/4$	$(N^2+3N-4)/2$
Selecção	C =	$(N^2-N)/2$	$(N^2-N)/2$	$(N^2-N)/2$
	A =	$3(N-1)$	$N(\mathrm{Ln}\ N+0.57)$	$(N^2/4+3(N-1))$
Troca (bubblesort)	C =	$(N^2-N)/2$	$(N^2-N)/2$	$(N^2-N)/2$
	A =	0	$(N^2-N)*0.75$	$(N^2-N)*1.5$

Retirado de Wirth, A+DS=P.

O InsertionSort parece ser o melhor dos três.

Quicksort algorítmico

Já que temos o bubblesort, temos de ter o quicksort enquanto classe:

```
template <class T>
class Quicksort: public SortingAlgorithm<T> {
public:
  virtual ~Quicksort();
};
```

```
template <class T>
Quicksort<T>::~Quicksort()
{
}
```

Isto é mesmo só para marcar posição, porque a classe SortingAlgorithm<T> já implementa o quicksort...

Quicksort com partição de Lomuto

O que distingue o quicksort é a partição. O livro de *Introduction to Algorithms* usa uma partição diferente da de Hoare, muito interessante também.

```
template <class T>
class QuicksortLomuto: public SortingAlgorithm<T> {
public:
  virtual ~QuicksortLomuto();
  virtual void Sort(Sortable<T>& v) const;
private:
  void Sort(Sortable<T>& v, int lowerBound, int upperBound) const;
protected:
  int Partition(Sortable<T>& v, int lowerBound, int upperBound) const;
};
```

```
template <class T>
void QuicksortLomuto<T>::Sort(Sortable<T>& v, int lowerBound, int upperBound) const
{
  if (lowerBound < upperBound)
  {
    int separation = Partition(v, lowerBound, upperBound);
    Sort(v, lowerBound, separation - 1);
    Sort(v, separation + 1, upperBound);
  }
}
```

```
template <class T>
void QuicksortLomuto<T>::Sort(Sortable<T>& v) const
{
  Sort(v, 0, v.Count() -1);
}
```


Partição de Lomuto

Programa-se assim:

```
template <class T>
int QuicksortLomuto<T>::Partition(Sortable<T>& v, int lowerBound, int upperBound) const
{
  const T& x = v.At(upperBound);
  int i = lowerBound;
  for (int j = lowerBound; j < upperBound; j++)
    if (v.GetOrder()(v.At(j), x))
      v.Swap(i++, j);
  v.Swap(i, upperBound);
  return i;
}
```

Percebido? Não? Então veja na página seguinte.

Descrição da partição de Lomuto

O pivô (a cinzento) é o último elemento (está na posição upperBound). Em cada momento, os elementos nas posições [lowerBound..i[(a amarelo) são menores ou iguais ao pivô e os nas posições [i..j[(a verde) são maiores do que o pivô. Os outros (nas posições [j..upperBound[) não sabemos. No final (depois do ciclo) troca-se o elemento na posição j com o pivô, o que garante que o pivô já não encontrou a sua posição definitiva, e que o vector está partido.

3	7	2	9	5	8	1	7	4	5
3	7	2	9	5	8	1	7	4	5
3	7	2	9	5	8	1	7	4	5
3	2	7	9	5	8	1	7	4	5
3	2	7	9	5	8	1	7	4	5
3	2	5	9	7	8	1	7	4	5
3	2	5	9	7	8	1	7	4	5
3	2	5	1	7	8	9	7	4	5
3	2	5	1	7	8	9	7	4	5
3	2	5	1	4	8	9	7	7	5
3	2	5	1	4	5	9	7	7	8

Quicksort com pivô aleatório

Para evitar azares com o pivô, escolhe-se aleatoriamente. Na verdade basta redefinir a partição. Usemos como base a partição de Lomuto:

Repare.

```
template <class T>
class QuicksortRandomized: public QuicksortLomuto<T> {
private:
  mutable Random random;
public:
  QuicksortRandomized();
  virtual ~QuicksortRandomized();
protected:
  virtual int Partition(Sortable<T>& v, int lowerBound, int upperBound) const;
};
```

A classe Random é um iterador de números aleatórios. (Veja página 165.)

Um membro de dados classificado mutable pode ser modificado numa função const.

Podemos antecipar que esta função randomiza o pivô e depois chama a função herdada.

Randomizando o pivô

“Randomizar” é um anglicismo que significa “tornar aleatório”.

Claro que não vamos repetir o código da partição de Lomuto: vamos buscá-lo à classe de base:

```
template <class T>
int QuicksortRandomized<T>::Partition(Sortable<T>& v, int lowerBound, int upperBound) const
{
  random.SetMax(upperBound - lowerBound);
  int i = lowerBound + random.Current();
  random.Get();
  v.Swap(i, upperBound);
  return QuicksortLomuto<T>::Partition(v, lowerBound, upperBound);
}
```

O iterador Random gera números aleatórios entre 0 e um certo limite máximo. O modificador SetMax muda esse limite máximo durante a iteração.

Repare.

O construtor inicializa o iterador:

```
template <class T>
QuicksortRandomized<T>::QuicksortRandomized():
  random()
{
  Random::Reset();
}
```

Note bem: a função Partition é um selector da classe QuicksortRandomized<T>. No entanto, ela modifica o iterador. Por isso é que ele tem de ser mutável.

Quicksort com partição de Hoare

Se temos um quicksort para o Lomuto, temos de ter um para o Hoare, evidenciando a partição:

```
template <class T>
class QuicksortHoare: public SortingAlgorithm<T> {
public:
  virtual ~QuicksortHoare();
  virtual void Sort(Sortable<T>& v) const;
private:
  virtual void Sort(Sortable<T>& v, int lowerBound, int upperBound) const;
                                          // pre lowerBound < upperBound;
protected:
  virtual void Partition(Sortable<T>& v, int lowerBound, int upperBound, int& ii, int& jj) const;
};
```

Repare na precondição. A função recursiva só é chamada se o intervalo a ordenar tiver mais do que um elemento.

```
template <class T>
void QuicksortHoare<T>::Sort(Sortable<T>& v) const
{
  if (v.Count() > 1)
    Sort(v, 0, v.Count() -1);
}
```

Aqui a situação é menos límpida, pois precisamos de dois argumentos de saída para os limites direito da partição esquerda e para o limite esquerdo da partição direita.

Partição de Hoare

Já a conhecemos muito bem:

```
template <class T>
void QuicksortHoare<T>::Partition(Sortable<T>& v,
     int lowerBound, int upperBound, int& ii, int& jj) const
{
  int i = lowerBound;
  int j = upperBound;
  const T p(v.At((i+j)/2));
  do
  {
    while (v.GetOrder().LessThanStrict(v.At(i), p))
      i++;
    while (v.GetOrder().GreaterThanStrict(v.At(j), p))
      j--;
    if (i <= j)
      v.Swap(i++, j--);
  } while (i <= j);
  ii = i;
  jj = j;
}
```

Preferimos não usar os parâmetros de saída como variáveis.

Usa-se assim:

```
template <class T>
void QuicksortHoare<T>::Sort(Sortable<T>& v,
       int lowerBound, int upperBound) const
{
  int i;
  int j;
  Partition(v, lowerBound, upperBound, i, j);
  if (lowerBound < j)
    Sort(v, lowerBound, j);
  if (i < upperBound)
    Sort(v, i, upperBound);
}
```


Quicksort com pivô mediano

Em alternativa ao pivô aleatório temos o pivô mediana de três. Este adapta-se especialmente bem à partição de Hoare:

```
template <class T>
class QuicksortMedianOfThree: public QuicksortHoare<T> {
public:
  virtual ~QuicksortMedianOfThree();
  virtual void Sort(Sortable<T>& v) const;
private:
  virtual void Sort(Sortable<T>& v, int lowerBound, int upperBound) const;
};
```

Usamos a partição de Hoare, herdada. Apenas a função Sort recursiva é redefinida.

Podíamos ter feito o mesmo desenvolvimento a partir da partição de Lomuto, claro.

Calculando o pivô mediano

Basta ordenar os elementos inicial, central e final do subvector. Depois, como o primeiro elemento já está do lado certo e o último também, escusamos de nos preocupar com eles na partição.

```
template <class T>
void QuicksortMedianOfThree<T>::Sort(Sortable<T>& v, int lowerBound, int upperBound) const
{
  int i;
  int j;
  v.SortThree(lowerBound, (lowerBound + upperBound) / 2, upperBound);
  if (upperBound - lowerBound > 2)
  {
    Partition(v, lowerBound + 1, upperBound - 1, i, j);
    if (lowerBound < j)
      Sort(v, lowerBound, j);
    if (i < upperBound)
      Sort(v, i, upperBound);
  }
}
```

A função SortThree vem da classe Sortable<T>.

Se houver três ou menos elementos, já estão ordenados.

Os elementos nas posições lowerBound e upperBound já estão nas suas partições.

Ordenando só três

A função SortThree ordena três elementos de um contentor Sortable<T> referenciados pelos seus índices:

```
template <class T>
class Sortable: public Container<T> {
private:
  Order<T>* order;
public:
  Sortable();
  // …
  virtual void SortThree(int i, int j, int k);
};
```

É uma espécie de bubblesort: troca-se o terceiro e o segundo, se estiverem fora de ordem; depois o segundo e o primeiro, se idem; depois o terceiro e o segundo, se idem.

```
template <class T>
void Sortable<T>::SortThree(int i, int j, int k)
{
  if (!order->LessThan(At(j), At(k)))
    Swap(j, k);
  if (!order->LessThan(At(i), At(j)))
    Swap(i, j);
  if (!order->LessThan(At(j), At(k)))
    Swap(j, k);
}
```

Quicksort com *cutoff*

A ideia é não fazer as chamadas recursivas quando os subvectores tiverem menos do que um certo número de elementos, o *cutoff*. O vector ficará “quase ordenado”, mas haverá alguns elementos localmente fora de ordem. A seguir entra o InsertionSort, pois é mesmo desse tipo de vectores que ele gosta mais.

```
template <class T>
class QuicksortWithCutoff: public QuicksortHoare<T> {
private:
  int cutoff;
public:
  QuicksortWithCutoff(int cutoff = 3); // pre cutoff >= 3;
  virtual ~QuicksortWithCutoff();
  virtual void Sort(Sortable<T>& v) const;
private:
  virtual void Sort(Sortable<T>& v, int lowerBound, int upperBound) const;
};
```

Deriva de QuicksortHoare, mas usa a técnica do pivô mediana de três.

Implementando o *cutoff*

O construtor inicializa o *cutoff*:

```
QuicksortWithCutoff<T>::QuicksortWithCutoff(int cutoff):
  cutoff(cutoff)
{
}
```

No fim faz-se um *insertion sort*:

```
template <class T>
void QuicksortWithCutoff<T>::Sort(Sortable<T>& v) const
{
  if (v.Count() >= cutoff)
    Sort(v, 0, v.Count() - 1);
  v.SortAlgorithmically(InsertionSort<T>());
}
```

Observe o *cutoff*:

```
template <class T>
void QuicksortWithCutoff<T>::Sort(Sortable<T>& v, int lowerBound, int upperBound) const
{
  int i;
  int j;
  v.SortThree(lowerBound, (lowerBound + upperBound) / 2, upperBound);
  Partition(v, lowerBound + 1, upperBound - 1, i, j);
  if (j - lowerBound + 1 >= cutoff)
    Sort(v, lowerBound, j);
  if (upperBound - i + 1 >= cutoff)
    Sort(v, i, upperBound);
}
```

Aqui haverá sempre 3 ou mais elementos para ordenar.

Assim, as muitas chamadas recursivas nos níveis finais são substituídas por um único *insertion sort*.

Outros algoritmos (1): Mergesort

A ideia é simples: divide-se o vector ao meio; ordena-se cada metade; fundem-se as duas metades já ordenadas. Para ordenar cada metade, usamos o MergeSort recursivamente.

```
template <class T>
class MergeSort: public SortingAlgorithm<T> {
public:
  virtual ~MergeSort();
  virtual void Sort(Sortable<T>& v);
private:
  virtual void Sort(Sortable<T>& v, int x, int y);
  virtual void Merge(Sortable<T>& v, int x, int m, int y);
};
```

```
template <class T>
void MergeSort<T>::Sort(Sortable<T>& v)
{
  Sort(v, 0, v.Count() - 1);
}
```

```
template <class T>
void MergeSort<T>::Sort(Sortable<T>& v, int x, int y)
{
  if (x < y)
  {
    int m = (x + y) / 2;
    Sort(v, x, m);
    Sort(v, m+1, y);
    Merge(v, x, m, y);
  }
}
```

Só falta a fusão: veja na página seguinte.

Fusão de subvectores

Primeiro copiam-se os dois vectores para vectores auxiliares. Depois percorrem-se esses dois vectores, recolhendo em cada passo o menor dos elementos correntes. Finalmente, junta-se o que sobrar.

```
template <class T>
void MergeSort<T>::Merge(Sortable<T>& v, int x, int m, int y)
{
  Sortable<T>* p1 = v.Prototype(m - x + 1);
  Sortable<T>* p2 = v.Prototype(y - m);
  for (int i = 0; i <= m - x; i++)
    p1->Put(v.At(x + i));
  for (int i = 0 ; i < y - m; i++)
    p2->Put(v.At(m + 1 + i));
  int i1 = 0;
  int i2 = 0;
  int k = x;
  while (i1 < p1->Capacity() && i2 < p2->Capacity())
    if (v.GetOrder()(p1->At(i1), p2->At(i2)))
      v.PutAt(p1->At(i1++), k++);
    else
      v.PutAt(p2->At(i2++), k++);
  while (i1 < p1->Capacity())
    v.PutAt(p1->At(i1++), k++);
  while (i2 < p2->Capacity())
    v.PutAt(p2->At(i2++), k++);
  delete p1;
  delete p2;
}
```

Protótipos.

Ciclo de fusão principal: transfere-se o menor dos elementos correntes e avança-se

Ciclos residuais: só um deles é que funciona.

Atenção!

Protótipos de vectores

A classe Sortable<T> tem uma função Prototype que aloca um objecto do mesmo tipo, com a capacidade indicada:

```
template <class T>
class Sortable: public Container<T> {
private:
  Order<T>* order;
public:
  // …
  virtual Sortable<T>* Prototype(int capacity) const = 0;
};
```

Cada classe derivada implementa a sua, por exemplo:

```
template <class T>
Sortable<T>* VectorPolymorphicSortable<T>::Prototype(int capacity) const
{
  return new VectorPolymorphicSortable<T>(capacity);
}
```


Outros algoritmos (2): Heapsort

Um algoritmo fantástico. Fica como exercício.

```
template <class T>
class HeapSort: public SortingAlgorithm<T> {
public:
  // …
private:
  // …
};
```

Sim, isto tem muito a ver com os montes (*heaps*) que nós estudámos logo no início.

Mínimo e máximo simultâneos

Se precisarmos do mínimo e do máximo é melhor calculá-los em simultâneo do que um de cada vez: podemos poupar 25% das comparações.

Por outro lado, havendo dois valores a retornar, o melhor é usar parâmetros de saída:

```
template <class T>
class Sortable: public Container<T> {
private:
  Order<T>* order;
public:
  // ...
  virtual void IndicesOfMinAndMaxGeneral(const Order<T>& order,
          int startPos, int endPos, int& xMin, int& xMax);    // pre ValidRange(startPos, endPos);
  virtual void IndicesOfMinAndMaxGeneral(const Order<T>& order, int& xMin, int& xMax);
                                                             // pre ValidRange(startPos, endPos);
  virtual void IndicesOfMinAndMax(int startPos, int endPos, int& xMin, int& xMax);
                                                             // pre ValidRange(startPos, endPos);
  virtual void IndicesOfMinAndMax(int& xMin, int& xMax); // pre ValidRange(startPos, endPos);
};
```

Parâmetros de saída, passados por referência: só para especialistas. ☺

Poupando nas comparações

```
template <class T>
void Sortable<T>::IndicesOfMinAndMaxGeneral(const Order<T>& order,
          int startPos, int endPos, int& xMin, int& xMax)
{
  int iMin;
  int iMax;
  if ((endPos - startPos) % 2 == 0)  // odd number of elements
    iMin = iMax = startPos;
  else
    if (order.LessThan(At(startPos), At(startPos + 1)))
      iMax = (iMin = startPos) + 1;
    else
      iMin = (iMax = startPos) + 1;
  for (int i = startPos + 2 - (iMin == iMax); i <= endPos; i += 2)
    if (order.LessThan(At(i), At(i + 1)))
    {
      if (order.LessThanStrict(At(i), At(iMin)))
        iMin = i;
      if (order.GreaterThanStrict(At(i + 1), At(iMax)))
        iMax = i + 1;
    }
    else
    {
      if (order.LessThanStrict(At(i+1), At(iMin)))
        iMin = i+1;
      if (order.GreaterThanStrict(At(i), At(iMax)))
        iMax = i;
    }
  xMax = iMax;
  xMin = iMin;
}
```

Aqui inicializamos. Depende da paridade do número de elementos.

Aqui comparamos 2 elementos.

Aqui comparamos com o mínimo e máximo correntes.

Poupamos em comparações, mas gastamos em texto ☹.

Avançamos 2 a 2. Antes de comparar com o mínimo e o máximo correntes, comparamos os dois elementos entre si. Assim, por cada 2 elementos fazemos 3 comparações, em vez de 4.

O *i*-ésimo elemento na ordem

Problema: achar o *i*-ésimo elemento de um vector de acordo com a ordem.

Atenção: o primeiro é o zeroésimo.

Solução: adaptemos a classe QuicksortRandomized<T> com uma função para isso, baseada na partição de Lomuto:

```
template <class T>
class QuicksortRandomized: public QuicksortLomuto<T> {
private:
  mutable Random random;
public:
  QuicksortRandomized();
  virtual ~QuicksortRandomized();
  virtual const T& Select(Sortable<T>& v, int i) const;
protected:
  virtual int Partition(Sortable<T>& v, int lowerBound, int upperBound) const;
  virtual const T& Select(Sortable<T>& v, int lowerBound, int upperBound, int i) const;
};
```

Esta é a função nova. É implementada através da função privada.

Função QuicksortRand...<T>::Select

É como o quicksort, mas basta seleccionar na partição do lado que contém o *i*-ésimo elemento:

```
template <class T>
const T& QuicksortRandomized<T>::Select(Sortable<T>& v, int i) const
{
  return Select(v, 0, v.Count() - 1, i);
}
```

```
template <class T>
const T& QuicksortRandomized<T>::Select(Sortable<T>& v,
                  int lowerBound, int upperBound, int i) const
{
  if (lowerBound == upperBound)
    return v.At(lowerBound);
  int separation = Partition(v, lowerBound, upperBound);
  int k = separation - lowerBound;
  if (i == k)
    return v.At(separation);
  else if (i < k)
    return Select(v, lowerBound, separation - 1, i);
  else
    return Select(v, separation + 1, upperBound, i - k - 1);
}
```

Se o intervalo só tem um elemento, tem de ser esse.

A variável k representa o número de elementos da partição esquerda.

Se i < k então o *i*-ésimo elemento está na partição da esquerda; se não é o (*i*-*k*-1)-ésimo elemento da partição da direita.

Análise do Select

Com azar, é quadrático, como o quicksort: se o pivô for sempre o maior elemento e estivermos à procura zeroésimo, faremos *N*-1 partições; as partições são lineares (o tempo de execução é proporcional a *N*) e, portanto, com azar o tempo de execução do Select é proporcional a N^2.

Com sorte, a partição parte sempre ao meio. Da primeira vez percorre *N* elementos, da segunda *N*/2, da terceira *N*/4, etc. Tudo somado dá menos de 2**N*. Logo, com sorte, o Select é linear.

A análise matemática do algoritmo (cf. *Introduction to Algorithms*) confirma que em média o Select é linear.

> Repare que, ao contrário do quicksort, a recursividade é simples: só prossegue por um dos lados. Por isso, não é difícil encontrar um algoritmo iterativo equivalente. (Fica como exercício.)

Busca binária

Quando um vector está ordenado, as buscas podem ser binárias. A classe Sortable<T> tem três funções de busca binária:

```
template <class T>
class Sortable: public Container<T> {
private:
  Order<T>* order;
public:
  // ...
  virtual int Upper(const T& s) const;  // pre IsSorted();
                        // returns the greatest r s.t. At(r) <= s, or -1 if At(0) > s;
  virtual int Lower(const T& s) const;  // pre IsSorted();
                        // returns the least r s.t. s <= At(r), or Count() if At(Count() - 1) < s;
  virtual bool HasBinary(const T& s) const;   // pre IsSorted();
};
```

> Note bem: a função Upper dá o índice do último elemento menor ou igual ao argumento, se houver ou –1, se não; a função Lower dá o índice do primeiro elemento maior ou igual ao argumento, se houver, ou Count() se não houver. Portanto, o resultado da função Lower é a posição de inserção, isto é, a posição em que deveríamos inserir o argumento de modo a que o vector continuasse ordenado.

> s.t. = *such that*

> A função HasBinary dá true se o argumento existir no objecto.

Funções Lower **e** Upper

Ei-las, para referência:

```
template <class T>
int Sortable<T>::Lower(const T& s) const
{
  int i = 0;
  int j = Count();
  while (i < j)
  {
    int m = (i + j) / 2;
    if (order->LessThanStrict(At(m), s))
      i = m + 1;
    else
      j = m;
  }
  return j;
}
```

Ambas as funções usam busca dicotómica e têm comportamento logarítmico.

```
template <class T>
int Sortable<T>::Upper(const T& s) const
{
  int i = -1;
  int j = Count() - 1;
  while (i < j)
  {
    int m = (i + j + 1) / 2;
    if (order->LessThan(At(m), s))
      i = m;
    else
      j = m - 1;
  }
  return i;
}
```

Se houver vários elementos iguais a s, o primeiro está na posição Lower(s) e o último na posição Upper(s).

Convença-se de que elas funcionam como anunciado. Se precisar de ajuda, veja no livro PCCC++, páginas 375, 376, 377, 660.

Função HasBinary

As funções Lower e Upper, sendo de busca, não indicam logo a presença ou ausência do argumento. Se precisarmos disso, usamos um teste suplementar, ou recorremos logo à função booleana HasBinary:

```
template <class T>
bool Sortable<T>::HasBinary(const T& s) const
{
  int x = Upper(s);
  return x != -1 && At(x) == s;
}
```

Pares, classe Pair<T, U> (1)

Os pares são uma estrutura de dados básica. A sua algoritmia é rudimentar:

```
template <class T, class U = T>
class Pair: public Clonable{
  T first;
  U second;
public:
  Pair();
  Pair(const T& first, const U& second);
  Pair(const Pair& other);
  virtual ~Pair();

  virtual void Copy(const Pair& other);
  virtual const Pair& operator = (const Pair& other);
  virtual Clonable* Clone() const;

  virtual const T& First() const;
  virtual T& First();
  virtual const U& Second() const;
  virtual U& Second();
  virtual void Set(const T& first, const U& second);
  virtual void SetFirst(const T& first);
  virtual void SetSecond(const U& second);

  //...
```

Por defeito, o segundo tipo paramétrico é o mesmo que o primeiro.

Construtores, destrutor.

Copy, =, Clone.

Selectores, modificadores, para o primeiro elemento, para o segundo, ...

Pares, classe Pair<T, U> (2)

```
  // ...
  virtual bool operator == (const Pair& other) const;
  virtual bool operator <= (const Pair& other) const;

  virtual bool LessThanByFirst (const Pair& other) const;
  virtual bool LessThanBySecond (const Pair& other) const;
  virtual bool EqualsByFirst (const Pair& other) const;
  virtual bool EqualsBySecond (const Pair& other) const;

  virtual void Read(std::istream& input = std::cin);
  friend std::istream& operator >> (std::istream& input, Pair<T, U>& x)
    {x.Read(input); return input;};

  virtual void Write(std::ostream& output = std::cout) const;
  virtual void WriteLine(std::ostream& output = std::cout) const;
  friend std::ostream& operator << (std::ostream& output, const Pair<T, U>& x)
    {x.Write(output); return output;};
};
```

Comparação.

Mais comparação.

Leitura.

Escrita.

As funções friend são definidas com a classe para facilitar a instanciação.

Pares, implementação

Não há surpresas. Vejamos as funções, de comparação, como exemplo:

```
template <class T, class U>
bool Pair<T, U>::operator == (const Pair& other) const
{
  return first == other.first && second == other.second;
}

template <class T, class U>
bool Pair<T, U>::operator <= (const Pair& other) const
{
  return LessThanByFirst(other);
}

template <class T, class U>
bool Pair<T, U>::EqualsByFirst(const Pair& other) const
{
  return first == other.first;
}

template <class T, class U>
bool Pair<T, U>::EqualsBySecond(const Pair& other) const
{
  return second == other.second;
}
```

Dá true se os primeiros forem iguais.

Dá true se os segundos forem iguais.

```
template <class T, class U>
bool Pair<T, U>::LessThanByFirst(const Pair& other) const
{
  return first <= other.first && !(first == other.first)
    || first == other.first && second <= other.second;
}

template <class T, class U>
bool Pair<T, U>::LessThanBySecond(const Pair& other) const
{
  return second <= other.second && !(second == other.second)
    || second == other.second && first <= other.first;
}
```


Ordenando vectores de pares

Problema: ordenar um vector de pares <StringBasic, int> pelo primeiro campo.

```
void TestSortPairs()
{
  VectorPolymorphicSortable<Pair<StringBasic, int> >::SetPrefixSuffix("", ";");
  VectorPolymorphicSortable<Pair<StringBasic, int> > v(100);

  v.Put(Pair<StringBasic, int>("kkkkkf", 27));
  v.Put(Pair<StringBasic, int>("zzza", 36));
  v.Put(Pair<StringBasic, int>("bbbbbbbc", 566));
  v.Put(Pair<StringBasic, int>("fffe", 12));
  v.Put(Pair<StringBasic, int>("xxd", 15));
  v.Put(Pair<StringBasic, int>("nnnnnb", 80));
  v.Put(Pair<StringBasic, int>("cccccck", 20));
  v.Put(Pair<StringBasic, int>("hhhhd", 19));
  v.WriteLine();
  v.SortGeneral(OrderSimple<Pair<StringBasic, int> >());
  v.WriteLine();
}
```

Recorde que a ordem simples é a ordem associada ao operador <= no tipo paramétrico.

E se quiséssemos ordenar pelo segundo campo?

Ordem adaptada

Para ordenar pelo segundo campo, podíamos usar uma classe descartável para a função de ordem, mas é melhor recorrer a uma ordem adaptada:

```
template <class T>
class OrderAdapted: public Order<T>{
private:
  typedef bool (T::*F)(const T&) const;
  F f;
public:
  OrderAdapted(F f);
  virtual ~OrderAdapted();
  virtual bool LessThan(const T& x, const T& y) const;
  virtual Clonable* Clone() const;
};
```

O tipo definido F é o tipo dos selectores booleanos da classe T que têm um argumento de tipo T passado por referência const. (É o caso dos operadores de comparação.)

Sim, a classe tem um membro de dados f de tipo F. (Como é que se usará isto?)

No construtor passamos a função. (Como se trata de um apontador, passamos por valor.)

A função LessThan redefine a função herdada, e função Clone implementa a função virtual pura herdada, como de costume.

Ordem adaptada, implementação

O construtor apenas inicializa o membro de dados que representa a função:

```
template <class T>
OrderAdapted<T>::OrderAdapted(F f):
  f(f)
{
}
```

O Clone é como de costume:

```
template <class T>
Clonable* OrderAdapted<T>::Clone() const
{
  return new OrderAdapted<T>(*this);
}
```

A função LessThan retorna o valor da função f chamada para o objecto x e com o argumento y:

```
template <class T>
bool OrderAdapted<T>::LessThan(const T& x, const T& y) const
{
  return (x.*f)(y);
}
```

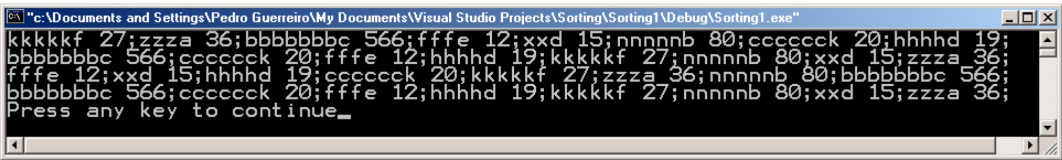
Repare: (x.*f)(y). Tem de se escrever mesmo assim.

Usando a ordem adaptada

Usemos a ordem adaptada para ordenar o vector de pares pelo segundo campo e depois outra vez pelo primeiro:

```
void TestSortPairs()
{
  VectorPolymorphicSortable<Pair<StringBasic, int> >::SetPrefixSuffix("", ";");
  VectorPolymorphicSortable<Pair<StringBasic, int> > v(100);

  v.Put(Pair<StringBasic, int>("kkkkkf", 27));
  // ...
  v.Put(Pair<StringBasic, int>("hhhhd", 19));
  v.WriteLine();
  v.SortGeneral(OrderSimple<Pair<StringBasic, int> >());
  v.WriteLine();
  v.SortGeneral(OrderAdapted<Pair<StringBasic, int> >(Pair<StringBasic, int>::LessThanBySecond));
  v.WriteLine();
  v.SortGeneral(OrderAdapted<Pair<StringBasic, int> >(Pair<StringBasic, int>::LessThanByFirst));
  v.WriteLine();
}
```

Letra mais pequena ☹ para caber na linha...



Outras ordens: a ordem pesada

Por vezes queremos ordenar pelo valor de uma função inteira. Por exemplo, ordenar um vector de StringBasic pelo comprimento da cadeia.

Em vez de criar uma classe descartável (a classe ByLength da página 169), usamos a classe OrderWeighted<T>:

```
template <class T>
class OrderWeighted: public Order<T>{
private:
  typedef int (T::*F)() const;
  F f;
  Order<T>* tieBreak;
public:
  OrderWeighted(F f);
  OrderWeighted(F f, const Order<T>& tieBreak);
  OrderWeighted(const OrderWeighted<T>& other);
  virtual ~OrderWeighted();
  virtual bool LessThan(const T& x, const T& y) const;
  virtual Clonable* Clone() const;
};
```

Aqui o tipo F é o tipo da função do peso, um selector com resultado int, sem argumentos.

O membro tieBreak representa o função de ordem usada para desempate. Por defeito, será a ordem simples.

Ordem pesada, implementação (1)

Construtores:

```
template <class T>
OrderWeighted<T>::OrderWeighted(F f):
  f(f),
  tieBreak(new OrderSimple<T>)
{
}
```

Repare na inicialização por defeito com a ordem simples.

```
template <class T>
OrderWeighted<T>::OrderWeighted(F f, const Order<T>& tieBreak):
  f(f),
  tieBreak(dynamic_cast<Order<T>*>(tieBreak.Clone()))
{
}
```

Aqui o argumento é clonado.

```
template <class T>
OrderWeighted<T>::OrderWeighted(const OrderWeighted<T>& other):
  f(other.f),
  tieBreak(dynamic_cast<Order<T>*>(other.tieBreak->Clone()))
{
}
```

Destrutor:

```
template <class T>
OrderWeighted<T>::~OrderWeighted()
{
  delete tieBreak;
}
```

Clone:

```
template <class T>
Clonable* OrderWeighted<T>::Clone() const
{
  return new OrderWeighted<T>(*this);
}
```


Ordem pesada, implementação (2)

O mais interessante é a função LessThan:

```
template <class T>
bool OrderWeighted<T>::LessThan(const T& x, const T& y) const
{
  int x1 = (x.*f)();
  int y1 = (y.*f)();
  return x1 < y1 || x1 == y1 && tieBreak->LessThan(x, y);
}
```

Simples, não é?

Usando a ordem pesada (1)

Usemos a ordem pesada para ordenar um vector de StringBasic, primeiro desempatando com a ordem simples, depois desempatando com a inversa da ordem simples inversa, isto é, a ordem associada ao operador >=.

```
void TestSortStringsWeighted()
{
  VectorPolymorphicSortable<StringBasic>::SetPrefixSuffix(" ", "");
  VectorPolymorphicSortable<StringBasic> v(100);

  v.Put("kkkkkf");
  v.Put("zzza");
  v.Put("bbbbbbbbc");
  // ...
  v.Put("uud");
  v.Put("ggggo");
  v.Put("ssssz");
  VectorPolymorphicSortable<StringBasic> v2(v);
  // ...
}
```


Usando a ordem pesada (2)

Aqui se vê a utilização das ordens:

```
void TestSortPairs()
{
  // ...
  v.WriteLine();
  v.SortGeneral(OrderWeighted<StringBasic>(StringBasic::Count));
  v.WriteLine();
  v = v2;
  v.WriteLine();
  Order<StringBasic>& secondary = OrderAdapted<StringBasic>(StringBasic::operator >=);
  Order<StringBasic>& order = OrderWeighted<StringBasic>(StringBasic::Count, secondary);
  v.SortGeneral(order);
  v.WriteLine();
}
```

Aqui desempatamos pela ordem simples...

Usamos referências auxiliares para as ordens para evitar expressões muito longas

... e aqui pela inversa da ordem simples.

"c:\Documents and Settings\Pedro Guerreiro\My Documents\Visual Studio Projects\Sorting\Sorting1\Debug\Sorting1.exe"

```
 kkkkkf zzza bbbbbbbbc fffe xxd aaaah nnnnb jjk uv ccccm uud ggggo ssssz
 uv jjk uud xxd fffe zzza aaaah ccccm ggggo nnnnb ssssz kkkkkf bbbbbbbbc
 kkkkkf zzza bbbbbbbbc fffe xxd aaaah nnnnb jjk uv ccccm uud ggggo ssssz
 uv xxd uud jjk zzza fffe ssssz nnnnb ggggo ccccm aaaah kkkkkf bbbbbbbbc
Press any key to continue
```

Colecções com funções gerais

As colecções têm funções gerais cujos argumentos são predicados:

```
template <class T>
class Collection: public virtual Container<T> {
public:
  virtual ~Collection<T>();
  virtual bool Has(const T& x) const;
  virtual int CountIf(const T& x) const;
  virtual void Prune(const T& x);
  virtual void PruneAll(const T& x);  // post !Has(x);

  virtual bool HasGeneral(const Predicate<T>& p) const;
  virtual int CountIfGeneral(const Predicate<T>& p) const = 0;
  virtual void PruneGeneral(const Predicate<T>& p) = 0;
  virtual void PruneAllGeneral(const Predicate<T>& p);  // post !HasGeneral(p);

  virtual IteratorSmart<T> Items() const;
  virtual IteratorSmart<T> ItemsGeneral(const Predicate<T>& p) const = 0;
};
```

Tens cá algum elemento que verifique o predicado p?

Quantos tens que verifiquem o predicado p?

Deita fora um dos que verificam o predicado p.

Deita fora os que verificam o predicado p.

Passa para cá os que verificam o predicado p.

Predicados

Um predicado é um objecto de uma classe derivada de Predicate<T>. Representa uma função booleana com um argumento de tipo T:

```
template <class T>
class Predicate {
public:
  virtual ~Predicate();
  virtual const Predicate<T>* Clone() const;

  virtual bool Good(const T& x) const;
  virtual bool Bad(const T& x) const ;
  virtual bool operator ()(const T& x) const;
};
```

Repare que a classe Predicate<T> tem uma função Clone, mas não herda de Clonable. Assim, Predicate<T> é a classe de base e a função Clone retorna um apontador para Predicate<T> e não para Clonable. Isto facilita as operações porque dispensa a conversão dinâmica (dynamic_cast).

A classe de base Predicate<T> representa a função que dá sempre true. (Veja a implementação na página seguinte). É usada por defeito em algumas ocasiões.

Os bons, os maus e os clones

Por defeito, todos somos bons:

```
template <class T>
bool Predicate<T>::Good(const T& x) const
{
  return true;
}
```

Os maus são os que não são bons:

```
<class T>
bool Predicate<T>::Bad(const T& x) const
{
  return !Good(x);
}
```

As classes derivadas redefinem o Good, mas não o Bad.

O operador () equivale à função Good:

```
template <class T>
bool Predicate<T>::operator ()(const T& x) const
{
  return Good(x);
}
```

Os clones são cópias dinâmicas polimórficas.

```
template <class T>
const Predicate<T>* Predicate<T>::Clone() const
{
  return new Predicate<T>(*this);
}
```


Exemplo com listas duplas

As listas são colecções:

```
template <class T>
class ListDoubleAbstract: public Clonable,
                 public virtual Dispenser<T>,
                 public virtual Collection<T>,
                 public virtual Bilinear<T> {
// ...
public:
// ...
//From Collection<T>
// ...
  virtual bool HasGeneral(const Predicate<T>& p) const;
  virtual int CountIfGeneral(const Predicate<T>& p) const;
  virtual void PruneGeneral(const Predicate<T>& p);
  virtual void PruneAllGeneral(const Predicate<T>& p);
// ...
};
```

Esta é a classe ListDoubleAbstract<T>, da qual derivam ListDouble<T>, ListPolymorphic<T> e ListIndirect<T>.

Funções HasGeneral, CountIfGeneral

Observe com calma:

```
template <class T>
bool ListDoubleAbstract<T>::HasGeneral(const Predicate<T>& p) const
{
  for (Iterator<T>& i = Items(); i; i++)
    if (p(*i))
      return true;
  return false;
}
```

Repare que p(*i) é o mesmo que p.Good(*i).

```
template <class T>
int ListDoubleAbstract<T>::CountIfGeneral(const Predicate<T>& p) const
{
  int result = 0;
  for (Iterator<T>& i = Items(); i; i++)
    result += p(*i);
  return result;
}
```


Funções PruneGeneral

Observe, ainda com mais calma:

```
template <class T>
void ListDoubleAbstract<T>::PruneGeneral(const Predicate<T>& p)
{
  SearchGeneral(p);
  if (!Off())
    Remove();
}
```

```
template <class T>
void ListDoubleAbstract<T>::PruneAllGeneral(const Predicate<T>& p)
{
  for(SearchGeneral(p); !Off(); SearchGeneralForward(p))
    Remove();
}
```

Estude as funções de busca geral na página seguinte.

Busca geral

As listas têm funções de busca geral:

```
template <class T>
class ListDoubleAbstract: public … {
// …
public:
// …
  virtual void SearchGeneral(const Predicate<T>& p);
  virtual void SearchGeneralForward(const Predicate<T>& p);
};
```

Também há funções de busca geral para trás. Veja no livro, página 680 e seguintes.

A primeira programa-se em termos da segunda:

```
template <class T>
void ListDoubleAbstract<T>::SearchGeneral(const Predicate<T>& p)
{
  Start();
  SearchGeneralForward(p);
}
```

```
template <class T>
void ListDoubleAbstract<T>::SearchGeneralForward(const Predicate<T>& p)
{
  while (!Off() && p.Bad(Item()))
    Forth();
}
```

Já sabemos que p.Bad(...) é o mesmo que !p.Good(...).

Exemplo: classe Multiple

Problema: eliminar de uma lista de inteiros os múltiplos de um dado número.

Começamos pelo predicado Multiple, que representa a propriedade "ser múltiplo de um número x". Este número x é indicado no construtor:

```
class Multiple: public Predicate<int>{
private:
  int divisor;
public:
  explicit Multiple(int divisor);
  virtual bool Good(const int& x) const;
  virtual const Predicate<int>* Clone() const;
};
```

```
Multiple::Multiple(int divisor):
  divisor(divisor)
{
}
```

Ser bom é dar resto zero:

```
bool Multiple::Good(const int& x) const
{
  return x % divisor == 0;
}
```

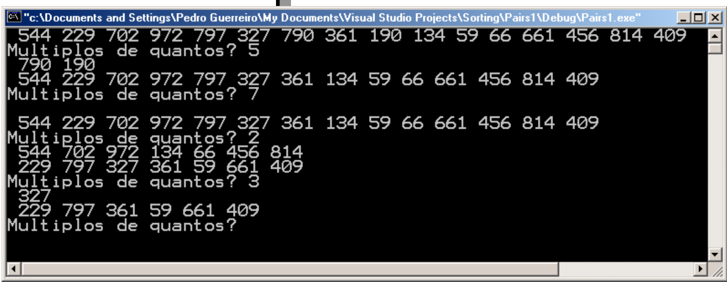
Para o Clone confiamos no construtor de cópia gerado automaticamente:

```
const Predicate<int>* Multiple::Clone() const
{
  return new Multiple(*this);
}
```

Removendo os múltiplos

Eis uma função de teste com uma lista de números aleatórios. Antes de remover, mostra os elementos que vão ser removidos:

```
void TestMultiples()
{
  ListDouble<int>::SetPrefixSuffix(" ", "");
  ListDouble<int> w;
  Random::Reset();
  w.PutItems(Random(1000, 16));
  w.WriteLine();
  for (;;)
  {
    int x;
    std::cout << "Multiplos de quantos? ";
    std::cin >> x;
    Output<int>(" ", "").PutItems(w.ItemsGeneral(Multiple(x)));
    w.PruneAllGeneral(Multiple(x));
    w.WriteLine();
  }
}
```



A classe Output<int> é um contentor que representa uma *stream* onde pôr um int é escrevê-lo.

Contentor Output<T>

A classe Output<T> é um contentor genérico que representa uma *stream* onde escrevemos elementos de tipo T com a função Put.

```
template <class T>
class Output: public Container<T> {
private:
  std::ostream& output;
  StringBasic prefix;
  StringBasic suffix;
  int count;
public:
  explicit Output(std::ostream& output = std::cout,
    const StringBasic& prefix = StringBasic(), const StringBasic& suffix = "\n");
  Output(const StringBasic& prefix, const StringBasic& suffix);
  virtual ~Output();
  virtual void Put(const T& x);
  virtual int Count() const;
  virtual void Clear();
  virtual void SetPrefixSuffix(const StringBasic& prefix, const StringBasic& suffix);
};
```

Ao escrever um elemento com Put, precedemo-lo pelo prefixo e sucedemo-lo pelo sufixo.

Classe Output<T>, implementação

A classe Output<T> é um contentor genérico que representa uma *stream* onde escrevemos elementos de tipo T com a função Put.

```
template <class T>
Output<T>::Output(std::ostream& output, const StringBasic& prefix, const StringBasic& suffix):
  output(output),
  prefix(prefix),
  suffix(suffix),
  count(0)
{
}
```

O construtor inicializa a *stream*, o prefixo e o sufixo.

Mais informações no livro, página 554 e seguintes.

O destrutor acrescenta um fim de linha, se for preciso:

```
template <class T>
Output<T>::~Output()
{
  if (!suffix.EndsBy("\n"))
    output << std::endl;
}
```

Para pôr, usa-se o operador <<:

```
template <class T>
void Output<T>::Put(const T& x)
{
  output << prefix << x << suffix;
  count++;
}
```

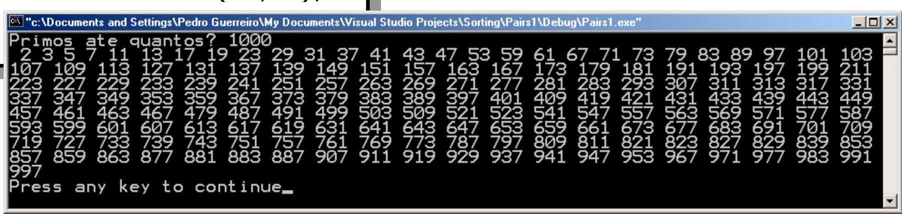

Crivo de Eratóstenes, de novo

O crivo de Eratóstenes é o algoritmo clássico para construir uma tabela de números primos. Ei-lo:

```
void TestEratosthenes()
{
  ListDouble<int> w;
  ListDouble<int> z;
  std::cout << "Primos ate quantos? ";
  int x;
  std::cin >> x;
  w.PutItems(FromDownto(x, 2));
  while (w.First() * w.First() <= x)
  {
    z.Put(w.First());
    w.PruneAllGeneral(Multiple(w.First()));
  }
  w.PutItems(z.Items());
  ListDouble<int>::SetPrefixSuffix(" ", "");
  w.WriteLine();
}
```

FromDownto é um iterador de números inteiros que, neste caso dá os números desde x até 2, em sucessão descendente.

Note que vamos recolhendo os números primos na lista z para no final os acrescentarmos à cabeça da lista objecto.



Iteradores FromTo, FromDownto

Servem para gerar sequências de números:

```
class FromTo: public Iterator<int> {
private:
  int current;
  int to;
  int step;
public:
  explicit FromTo(int from = 0, int to = std::numeric_limits<int>::max(), int step =  1);
                                                              // pre step >= 1;
  FromTo(const FromTo& other);
  virtual ~FromTo();
  virtual Iterator<int>* Clone() const;

  virtual const int& Current() const;
  virtual bool IsDone() const;
  virtual void Get();
};
```

Sequências ascendentes.

```
class FromDownto: public Iterator<int> {
private:
  int current;
  int downto;
  int step;
public:
  FromDownto(int from, int downto, int step =  1);
                                  // pre step >= 1;
  FromDownto(const FromDownto& other);
  virtual ~FromDownto();
  virtual Iterator<int>* Clone() const;

  virtual const int& Current() const;
  virtual bool IsDone() const;
  virtual void Get();
};
```

Sequência descendentes.

Observe os valores dos argumentos por defeito nos construtores.

Experiência com estringues 😐

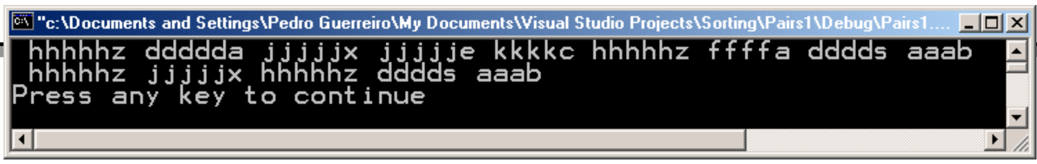
Vejamos agora um exemplo com listas de StringBasic.
Problema: listar todas as palavras portuguesas que são cadeias ordenadas.

Experimentemos com um pequeno ficheiro, lendo para uma lista e escrevendo as palavras que interessam:

```
void TestSortedWords1()
{
  ListDoublePolymorphic<StringBasic>::SetPrefixSuffix(" ", "");
  ListDoublePolymorphic<StringBasic> w;
  w.PutItems(Input<StringBasic>(std::ifstream("../dic.txt")));
  w.WriteLine();
  Output<StringBasic>
                    (" ", "").PutItems(w.ItemsGeneral(Logical<StringBasic>(StringBasic::IsSorted)));
}
```

Input<T> é um iterador que lê objectos de tipo T linha a linha a partir de uma *stream*.

Logical<T> é um predicado construído a partir de um selector booleano na classe T.



Iterador Input<T>

A classe Output<T> é um contentor. A classe Input<T> é um iterador.

```
template <class T>
class Input: public Iterator<T> {
private:
  std::istream& input;
  T current;
  int count;
public:
  explicit Input(std::istream& input = std::cin);
  virtual ~Input();

  virtual Iterator<T>* Clone() const;

  virtual void Get();
  virtual const T& Current() const;
  virtual bool IsDone() const;

  virtual int Count() const;
};
```

```
template <class T>
Input<T>::Input(std::istream& input):
  input(input),
  count(0)
{
  Get();
}
```

O construtor disponibiliza o primeiro elemento, se houver.

```
template <class T>
void Input<T>::Get()
{
  input >> current;
  if (input)
    count++;
}
```

Lê-se com o operador >>.

```
template <class T>
bool Input<T>::IsDone() const
{
  return !input;
}
```

Termina-se no fim do ficheiro.

Predicado Logical<T>

Serve para obtermos um predicado a partir de uma função booleana da classe:

```
template <class T>
class Logical: public Predicate<T>{
private:
  typedef bool (T::*F)() const;
  F f;
public:
  Logical(F f);
  virtual ~Logical();
  virtual const Predicate<T>* Clone() const;
  virtual bool Good(const T& x) const;
};
```

F é o tipo dos selectores booleanos da classe T.

```
template <class T>
Logical<T>::Logical(F f):
  f(f)
{
}
```

O elemento x é bom se quando lhe aplicamos o selector guardado em f o resultado for true.

```
template <class T>
bool Logical<T>::Good(const T& x) const
{
  return (x.*f)();
}
```

Cf. classe OrderAdapted<T>, na página 223.

Variante: remover as palavras más

Para isto, precisamos da negação do predicado Logical<StringBasic>(StringBasic::IsSorted). Faz-se assim:

```
void TestSortedWords2()
{
  ListDoublePolymorphic<StringBasic>::SetPrefixSuffix(" ", "");
  ListDoublePolymorphic<StringBasic> w;
  w.PutItems(Input<StringBasic>(std::ifstream("../dic.txt")));
  w.WriteLine();
  w.PruneAllGeneral(Negation<StringBasic>(Logical<StringBasic>(StringBasic::IsSorted)));
  w.WriteLine();
}
```

Assim se exprime a negação do predicado Logical<StringBasic>(StringBasic::IsSorted).

Negação

A classe Negation<T> é um predicado cujo construtor tem um argumento de tipo Predicate<T>. O objecto construído representa a negação desse:

```
template <class T>
class Negation: public Predicate<T>{
private:
  const Predicate<T>* p;
public:
  Negation(const Predicate<T>& p = Predicate<T>());
  Negation(const Negation<T>& other);
  virtual ~Negation();
  virtual const Predicate<T>* Clone() const;
  virtual bool Good(const T& x) const;
};
```

O predicado privado é inicializado por clonagem:

```
template <class T>
Negation<T>::Negation(const Predicate<T>& p):
  p(p.Clone())
{
}
```

A negação é boa quando o original é mau:

```
template <class T>
bool Negation<T>::Good(const T& x) const
{
  return p->Bad(x);
}
```

Palavras crescentes

Processamos o ficheiro “pt.txt” que contém quase 30000 palavras portuguesas, uma em cada linha:

```
void TestSortedWords()
{
  StringBasic fileName("C:/Documents and Settings/.../pt.txt");
  ListDoublePolymorphic<StringBasic> w;
  w.PutItems(Input<StringBasic>(std::ifstream(fileName.Image())));
  w.PruneAllGeneral(Negation<StringBasic>(Logical<StringBasic>(StringBasic::IsSorted)));
  Output<StringBasic>(" ", "").PutItems(w.ItemsReverse());
}
```

Eis o resultado:

```
"c:\Documents and Settings\Pedro Guerreiro\My Documents\Visual Studio Projects\Sorting\Pairs1\Debug\Pairs1.exe"
 a acem aceno aco acor afim agil agio agir ai aio amo amor ano ao ar as beijo be
lo bem bis ce cego cem ceu chino cimo cio cor cos cru cruz cu de demo dez do dor
 e eis elo em eu fim fino fio flor for foz giz hino ir km luz mo mor no nos noz
nu o ou tu
Press any key to continue
```


Palavras crescentes, com vector

Usemos um vector para as palavras, em vez de uma lista. É praticamente igual, porque os vectores também são colecções:

```
void TestSortedWordsWithVector()
{
  StringBasic fileName("C:/Documents and Settings/.../pt.txt");
  VectorPolymorphic<StringBasic> w(30000);
  w.PutItems(Input<StringBasic>(std::ifstream(fileName.Image())));
  w.PruneAllGeneral(Negation<StringBasic>(Logical<StringBasic>(StringBasic::IsSorted)));
  Output<StringBasic>(" ", "").PutItems(w.Items());
}
```

É preciso dimensionar o vector.

É igual, mas é bem mais rápido, pois não é precisa de alocar memória palavra a palavra.

Experiência com listas de pares

Mais difícil ainda: temos uma lista de pares Pair<StringBasic, int>. A string é a chave, o int é o valor. Queremos eliminar todos os pares com uma dada chave.

```
void TestNamesAndNumbers1()
{
  ListDoublePolymorphic<Pair<StringBasic, int> >::SetPrefixSuffix("", "/");
  ListDoublePolymorphic<Pair<StringBasic, int> > w;
  w.Put(Pair<StringBasic, int>("rosa", 218494455));
  // ...
  w.Put(Pair<StringBasic, int>("helena", 281301304));
  w.WriteLine();
  for (;;)
  {
    StringBasic s;
    s.Accept("Nome? ");
    w.PruneAllGeneral
           (Attribute<Pair<StringBasic, int>, StringBasic>(Pair<StringBasic, int>::First, s));
    w.WriteLine();
  }
}
```

Eliminam-se todos os pares cujo primeiro elemento vale s.

Predicado Attribute<T, U>. Ver página seguinte.

Predicado Attribute<T, U>

Representa a situação em que um objecto da classe T tem um atributo com um dado valor na classe U, acessível através de um selector sem argumentos, por referência const.

```
template <class T, class U>
class Attribute: public Predicate<T>{
private:
  typedef const U& (T::*F)() const;
  F f;
  U u;
public:
  Attribute(F f, const U& u);
  virtual ~Attribute();
  virtual const Predicate<T>* Clone() const;
  virtual bool Good(const T& x) const;
};
```

F é o tipo dos selectores sem argumenosda classe T com resultado de tipo U, por referência const.

```
template <class T, class U>
Attribute<T, U>::Attribute(F f, const U& u):
  f(f),
  u(u)
{
}
```

O elemento x é bom se quando usado para objecto da função f o resultado é u.

```
template <class T, class U>
bool Attribute<T, U>::Good(const T& x) const
{
  return (x.*f)() == u;
}
```

Seleccionando por valor

Queremos saber quem tem um dado número de telefone. É simples:

Usamos vectores, para variar.

```
void TestNamesAndNumbers2()
{
  VectorPolymorphic<Pair<StringBasic, int> >::SetPrefixSuffix("", "/");
  VectorPolymorphic<Pair<StringBasic, int> > w(100);
  w.Put(Pair<StringBasic, int>("rosa", 218494455));
  // ...
  w.Put(Pair<StringBasic, int>("helena", 281301304));
  w.WriteLine();
  for (;;)
  {
    int x;
    std::cout << "Telefone? ";
    std::cin >> x;
    Output<Pair<StringBasic, int> >("", "/").
        PutItems(w.ItemsGeneral
                  (Attribute<Pair<StringBasic, int>, int>(Pair<StringBasic, int>::Second, x)));
  }
}
```

Aqui vemos se o segundo do par tem valor x.

Aprenda a ler estas coisas...

Correndo os programas

Eis o resultado da função da página 251:

```
"c:\Documents and Settings\Pedro Guerreiro\My Documents\Visual Studio Projects\Sorting\Pairs1\Debug\Pairs1.exe"
helena 281301304/clara 214837710/joana 962676563/isabel 913455109/sofia 26580715
6/joana 214109091/clara 963488222/camila 281301304/rosa 218494455/
Nome? isabel
helena 281301304/clara 214837710/joana 962676563/sofia 265807156/joana 214109091
/clara 963488222/camila 281301304/rosa 218494455/
Nome? rosa
helena 281301304/clara 214837710/joana 962676563/sofia 265807156/joana 214109091
/clara 963488222/camila 281301304/
Nome? helena
clara 214837710/joana 962676563/sofia 265807156/joana 214109091/clara 963488222/
camila 281301304/
Nome? sandra
clara 214837710/joana 962676563/sofia 265807156/joana 214109091/clara 963488222/
camila 281301304/
Nome? joana
clara 214837710/sofia 265807156/clara 963488222/camila 281301304/
Nome?
```

E eis o da função da página anterior:

```
"c:\Documents and Settings\Pedro Guerreiro\My Documents\Visual Studio Projects\Sorting\Pairs1\Debug\Pairs1.exe"
rosa 218494455/camila 281301304/clara 963488222/joana 214109091/sofia 265807156/
isabel 913455109/joana 962676563/clara 214837710/helena 281301304/
Telefone? 214109091
joana 214109091/
Telefone? 281301304
camila 281301304/helena 281301304/
Telefone? 217812345

Telefone? 214837710
clara 214837710/
Telefone?
```

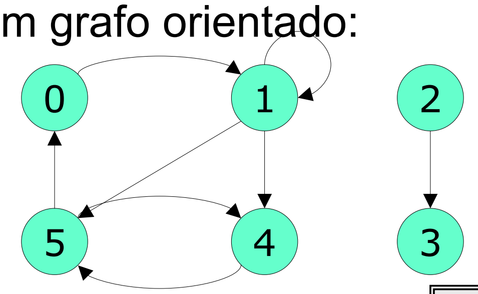
Grafos

Um grafo G é um par (V, E) onde V é um conjunto e E é uma relação binária definida em V.

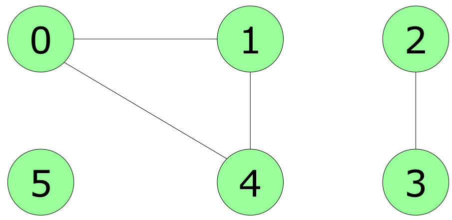
O elementos de V são os *vértices* e os de E são as *arestas*.

Nos grafos *não-orientados* a relação E é simétrica e anti-reflexiva, isto é, se (x, y) ∈ E então (y, x) ∈ E e não existem pares da forma (x, x).

Um grafo orientado:



Um grafo não-orientado:



Nos nossos exercícios, os vértices são um intervalo de números inteiros entre 0 e countVertices − 1.

Classe Graph

Começamos com uma classe abstracta Graph:

```
class Graph
{
public:
  virtual ~Graph();

  virtual Graph* Clone() const = 0;

  virtual int CountVertices() const = 0;
  virtual bool ValidVertex(int x) const = 0;
  virtual void Link(int x, int y) = 0;  // pre ValidVertex(x) && ValidVertex(y);
  virtual void LinkUndirected(int x, int y);  // pre ValidVertex(x) && ValidVertex(y) && x != y;
  virtual bool IsLinked(int x, int y) const = 0;
  virtual bool HasSuccessors(int x) const = 0;
  virtual int CountEdges() const = 0;
  virtual void Clear() = 0;
  virtual void Transpose() = 0;

  virtual IteratorSmart<int> Successors(int x) const = 0; // pre ValidVertex(x);
};
```

Esta é a única que não é virtual pura:

```
void Graph::LinkUndirected(int x, int y)
{
  Link(x, y);
  Link(y, x);
}
```

Os sucessores de um vértice x são todos os vértices y tais que (x, y) ∈ E.

Representação dos grafos

Há duas maneiras padrão de representar grafos:

- Com um vector de listas de vértices.
- Com uma matriz de adjacências.

No vector de listas, a lista na posição i do vector contém os sucessores do vértice i.

Na matriz de adjacências, o elemento na posição (i, j) valerá true se (i, j) for uma aresta. (Cf. Programação I, ☺)

A representação por listas é preferível para grafos rarefeitos. A representação por matrizes usa-se sobretudo com grafos densos.

Ambas dão para grafos não-orientados e para grafos orientados.

Vamos exprimir os algoritmos em termos das classes abstractas e depois experimentamo-los usando a representação com listas.

Classe GraphUsingList

Usamos listas duplas de números inteiros:

```
class GraphUsingList: public Graph
{
private:
  Vector<ListDouble<int> > successors;
public:
  GraphUsingList(int capacity);
  GraphUsingList(const GraphUsingList& other);
  virtual ~GraphUsingList();

  virtual Graph* Clone() const;

  virtual int CountVertices() const;
  virtual bool ValidVertex(int x) const;
  virtual void Link(int x, int y);
  virtual bool IsLinked(int x, int y) const;
  virtual bool HasSuccessors(int x) const;
  virtual int CountEdges() const;
  virtual void Clear();
  virtual void Transpose();

  virtual IteratorSmart<int> Successors(int x) const;
};
```

Exercício: programe uma classe GraphUsingMatrix.

Transpor um grafo é trocar o sentido das setas.

Classe GraphUsingList, implementação

É relativamente simples:

```
GraphUsingList::GraphUsingList(int countVertices):
  successors(countVertices)
{
  successors.Fill();
}

GraphUsingList::GraphUsingList(const GraphUsingList& other):
  successors(other.successors)
{
}

GraphUsingList::~GraphUsingList()
{
}

Graph* GraphUsingList::Clone() const
{
  return new GraphUsingList(*this);
}

bool GraphUsingList::ValidVertex(int x) const
{
  return successors.ValidIndex(x);
}
```

Inicializamos todos os elementos do vector com listas vazias.

O trabalho é realizado pelo construtor de cópia do vector de listas.

A função Clone é necessária para permitir o polimorfismo, mais adiante.

A função ValidVertex é usada nas precondições.

Class GraphUsingList, impl... (2)

Continuação:

```
void GraphUsingList::Link(int x, int y)
{
  if (!successors[x].Has(y))
    successors[x].PutLast(y);
}

bool GraphUsingList::IsLinked(int x, int y) const
{
  return successors[x].Has(y);
}

bool GraphUsingList::HasSuccessors(int x) const
{
  return !successors[x].Empty();
}

int GraphUsingList::CountVertices() const
{
  return successors.Capacity();
}

int GraphUsingList::CountEdges() const
{
  int result = 0;
  for (int i = 0; i < CountVertices(); i++)
    result += successors[i].Count();
  return result;
}
```

Pomos no fim da lista, mas a ordem não é significativa.

O número de vértices é dado pela capacidade do vector, estabelecida na construção.

Class GraphUsingList, impl... (3)

Continuação:

```
void GraphUsingList::Clear()
{
  successors.Clear();
  successors.Fill();
}

void GraphUsingList::Transpose()
{
  GraphUsingList temp(*this);
  Clear();
  for (int i = 0; i < temp.CountVertices(); i++)
    for (Iterator<int>& j = temp.Successors(i); j; j++)
      Link(*j, i);
}

IteratorSmart<int> GraphUsingList::Successors(int x) const
{
  return successors[x].Items();
}
```

Ao esvaziar um grafo, devemos logo a seguir colocar listas vazias em todas as posições do vector.

Para transpor o grafo, precisamos de um grafo auxiliar.

O iterador dos sucessores do nó x é o iterador da lista successors[x].

Busca larga

O problema é determinar todos os vértices alcançáveis a partir de um vértice de partida, calculando a distância (expressa em número de arestas) desde o vértice de partida até cada um dos vértices alcançados.

Os vértices alcançados formam uma árvore *larga* cuja raiz é o vértice de partida. Para cada vértice na árvore, o caminho desde a raiz corresponde a um caminho mínimo no grafo desde o vértice de partida.

Na busca larga, todos os vértices à distância k da raiz são alcançados, ou *descobertos*, antes dos vértices à distância k+1.

Brancos, cinzentos e pretos

Inicialmente todos os vértices são brancos. Quando um vértice é descoberto, fica cinzento. Quando todos os vértices adjacentes a um vértice cinzento ficam cinzentos, esse vértice fica preto.

Os vértices cinzentos ficam guardados numa fila.

Quando um vértice sai da fila, todos os seus sucessores brancos vão para a fila, mas pintados de cinzento, porque já foram descobertos. Assim, esse vértice que saiu da fila (e que era cinzento) pode passar a preto.

A disciplina da fila garante que vértices entram na fila por ordem crescente da distância ao vértice de partida.

Classe BreadthFirstSearch

Representamos o algoritmo de busca larga por uma classe:

```
class BreadthFirstSearch
{
private:
  const Graph& graph;
  enum ColorType {WHITE = 0, GRAY = 1, BLACK = 2};
  Vector<int> color;
  Vector<int> distance;
  Vector<int> predecessor;
  int start;
public:
  BreadthFirstSearch(const Graph& graph);
  virtual ~BreadthFirstSearch();

  virtual void Compute(int start);  // post Computed();
  virtual bool Computed() const;

  virtual int Predecessor(int x) const; // pre Computed();
  virtual int DistanceTo(int x) const; // pre Computed();
  virtual int Start() const; // pre Computed();

  virtual ListDouble<int> ShortestPathTo(int x) const; // pre Computed();
};
```

Referência constante para o grafo.

As cores.

distance[i] representa no fim a distância do vértice inicial até ao vértice i.

Representamos a árvore larga por meio deste vector predecessor.

Classe BreadthFirstSearch, impl...

Construtor, destrutor:

```
BreadthFirstSearch::BreadthFirstSearch(const Graph& graph):
  graph(graph),
  color(graph.CountVertices()),
  distance(graph.CountVertices()),
  predecessor(graph.CountVertices()),
  start(-1)
{
}
```

```
BreadthFirstSearch::~BreadthFirstSearch()
{
}
```

Se o vértice de partida ainda valer –1, o algoritmo ainda não calculou:

```
bool BreadthFirstSearch::Computed() const
{
  return start != -1;
}
```

```
int BreadthFirstSearch::Start() const
{
  return start;
}
```


Função da busca larga

```
void BreadthFirstSearch::Compute(int start)
{
  this->start = start;
  color.Fill(WHITE);
  distance.Fill(std::numeric_limits<int>::infinity());
  predecessor.Fill(-1);

  color[start] = GRAY;
  distance[start] = 0;

  QueueBounded<int> queue(graph.CountVertices());
  queue.Put(start);
  while (!queue.Empty())
  {
    int x = queue.Item();
    queue.Remove();
    for (Iterator<int>& i = graph.Successors(x); i; i++)
      if (color[*i] == WHITE)
      {
        color[*i] = GRAY;
        distance[*i] = distance[x] + 1;
        predecessor[*i] = x;
        queue.Put(*i);
      }
    color[x] = BLACK;
  }
}
```

Inicialmente todas as distâncias são infinitas, excepto a do vértice de partida, que é zero.

Palavras, para quê?

Caminho mais curto

O caminho desde o vértice inicial até a um dado vértice descoberto é devolvido na forma de uma lista de vértices, construída do fim para o princípio, através do vector predecessor:

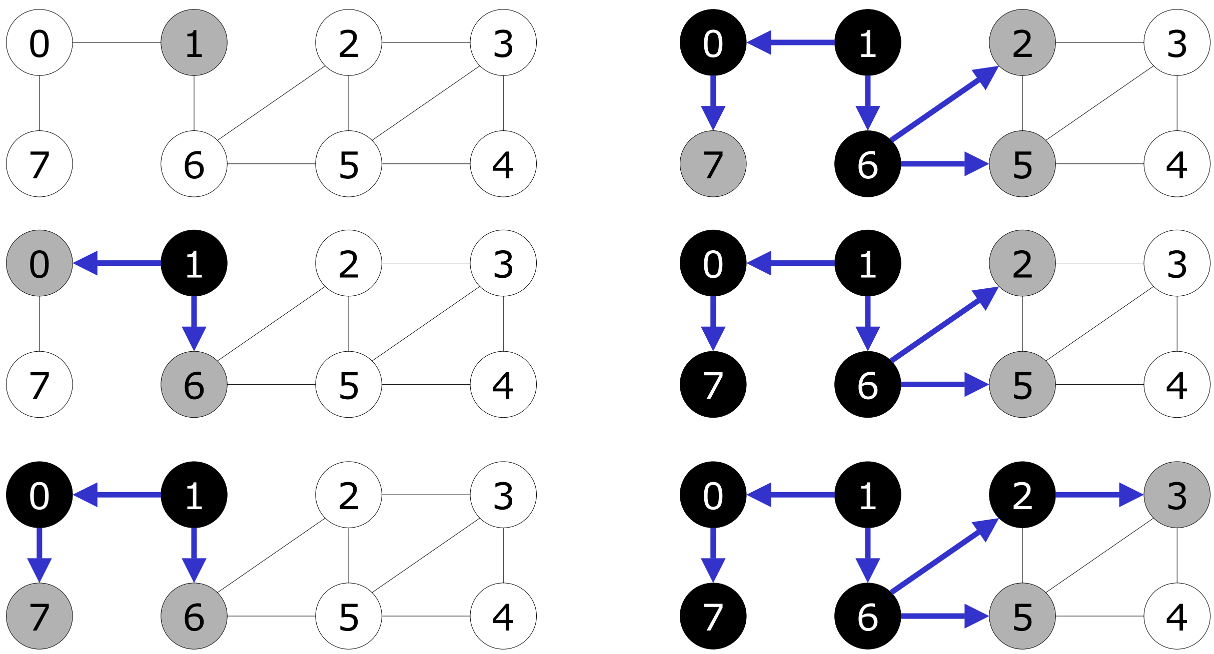
```
ListDouble<int> BreadthFirstSearch::ShortestPathTo(int x) const
{
  ListDouble<int> result;
  for (int i = x; i != -1; i = predecessor[i])
    result.PutFirst(i);
  return result;
}
```

```
int BreadthFirstSearch::Predecessor(int x) const
{
  return predecessor[x];
}
```

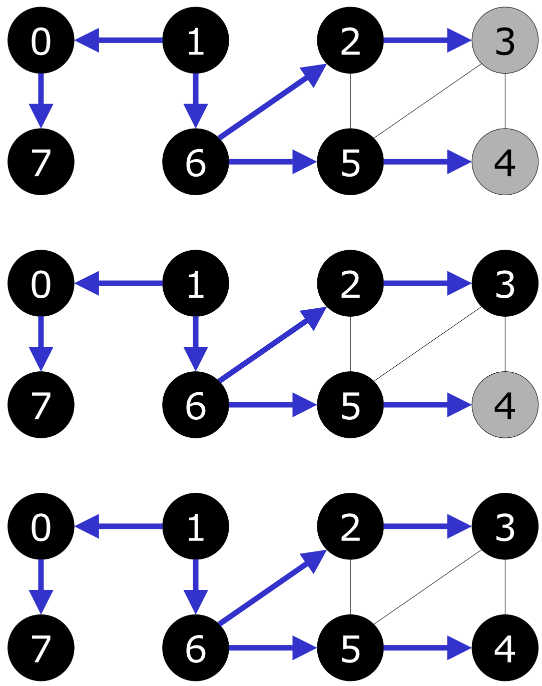
A distância já está calculada no vector distance:

```
int BreadthFirstSearch::DistanceTo(int x) const
{
  return distance[x];
}
```


Filme dos acontecimentos (1)



Filme dos acontecimentos (2)



Estado final dos vectores:

vértice	predecessor	distance
0	1	1
1	-1	0
2	6	2
3	2	3
4	5	3
5	6	2
6	1	1
7	0	2

Tudo preto.
Fim!

Análise da busca larga

Após a inicialização, nenhum vértice é pintado de branco. Portanto, o teste if `(color[*i] == WHITE)` garante que nenhum vértice entra na fila duas vezes. Pôr na fila usa tempo constante e portanto o tempo total para pôr na fila é proporcional ao número de vértices.

Por outro lado, admitindo a representação com listas de sucessores, o tempo do iterador é proporcional ao tamanho da lista. Ora o iterador é percorrido uma vez para cada nó. Ao todo, o tempo gasto nesse processamento é proporcional ao número de arestas.

O tempo de inicialização é proporcional ao número de vértices.

Juntando tudo, concluímos que o tempo é proporcional à soma do número de vértices com o número de arestas.

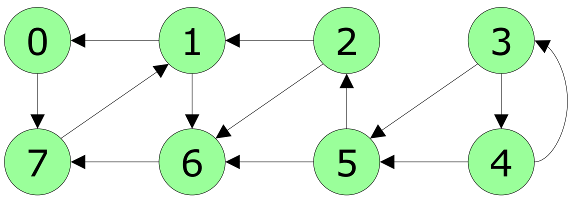
Por outras palavras, a busca larga usa tempo linear com o tamanho da representação do grafo em vértices mais arestas.

Busca profunda

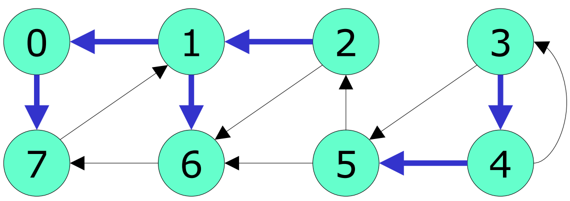
O problema é alcançar todos os vértices, buscando sempre segundo uma aresta que parte do vértice mais recentemente descoberto em direcção a um vértice ainda não descoberto. Quando atinge um vértice de onde todas as arestas levam a vértices já descobertos, a busca retrocede até um outro vértice de onde parta uma aresta em direcção a um vértice não descoberto e segue por essa. O processamento continua até não haver mais vértices alcançáveis a partir do vértice inicial. Se nesta altura ainda houver vértices por descobrir (não alcançados), o processamento repete-se a partir de um deles, até todos os vértices terem sido descobertos.

Floresta profunda

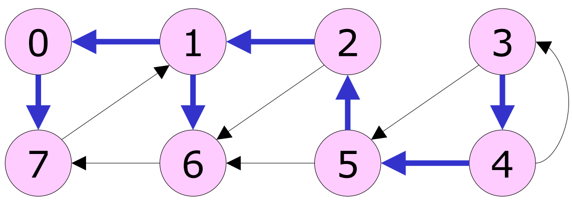
O subgrafo das predecessores é uma floresta. As árvores de precedências encontradas dependem da escolha dos vértices iniciais. Por exemplo, seja o grafo:



Começando primeiro no 2 e depois no 3, duas árvores:



Começando primeiro no 3, uma árvore:



E se começarmos no 7 e depois no 5 e depois no 3 ficamos com três árvores. Etc.

Classificando as arestas

Dado um grafo e a uma floresta profunda, há quatro conjuntos de arestas:

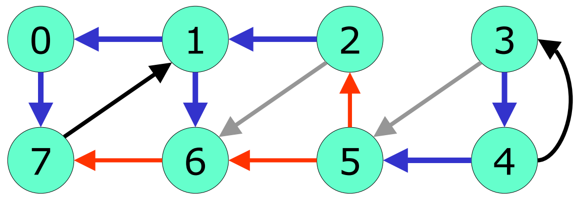
- Arestas ramosas
- Arestas progressivas
- Arestas retrógradas
- Arestas cruzadas

Pertencem a uma das árvores.

Ligam um vértice a outro alcançável a partir dele na árvore, mas não pertencem à árvore.

Ligam um vértice a outro a partir do qual o primeiro é alcançável na árvore.

Todos os outros. Podem ligar dois vértices de uma árvore, nenhum dos quais é alcançável a partir do outro na árvore, ou ligar vértices de árvores distintas.



Pintando e carimbando

Usamos um esquema de cores análogo ao da busca larga: brancos enquanto não são descobertos, cinzentos enquanto, tendo sido descobertos, têm descendentes ainda não descobertos e pretos quando já foram descobertos e já não têm descendentes por descobrir.

Além disso, carimbamos os vértices com a hora de descoberta (momento em que ficam cinzentos) e com a hora de fim (momento em que ficam pretos).

Também contamos as arestas retrógradas, para detectar grafos acíclicos.

E também inserimos os vértices à cabeça de uma lista à medida que chegam ao fim, para fins de ordenação topológica.

Classe DepthFirstSearch **(1)**

A busca profunda é uma classe, claro:

```
class DepthFirstSearch
{
private:
  const Graph& graph;
  enum ColorType {WHITE = 0, GRAY = 1, BLACK = 2};
  Vector<int> color;
  Vector<int> discoveryTime;
  Vector<int> finishTime;
  Vector<int> predecessor;
  ListDouble<int> finished;
  ListDouble<int> roots;
  int time;
  int countBackEdges;

public:
  DepthFirstSearch(const Graph& graph);
  virtual ~DepthFirstSearch();
// ...
};
```

Referência constante para o grafo.

Hora da descoberta.

Hora do fim.

Lista dos vértices por ordem inversa de hora de fim.

Raízes das árvores da floresta profunda.

Variável global para o marcar a passagem do tempo.

Classe DepthFirstSearch **(2)**

Continuação:

```
class DepthFirstSearch
{
// ...
  virtual void Compute();  // post Computed();
  virtual void Compute(Iterator<int>& i);  // post Computed();
  virtual bool Computed() const;

  virtual int Predecessor(int x) const; // pre Computed();
  virtual int DiscoveryTime(int x) const; // pre Computed();
  virtual int FinishTime(int x) const; // pre Computed();

  virtual IteratorSmart<int> Descendants(int x) const;  // pre Computed();
  virtual const ListDouble<int>& Topological() const;  // pre Computed();
  virtual bool IsAcyclic() const; // pre Computed();
  virtual const ListDouble<int>& Roots() const; // pre Computed();

private:
  virtual void Visit(int x);
// ...
};
```

Sem argumentos calcula pela ordem dos vértices: 0, 1, ..., countVertices – 1.

O iterador especifica a sequência por que visitamos os vértices.

Descendentes de x na floresta profunda.

Ordem topológica, se o grafo for acíclico.

Um grafo acíclico é um grafo sem ciclos.

Lista das raízes das árvores da floresta profunda.

DepthFirstSearch::IsPredicateOf

Predicado para caracterizar os descendentes de um vértice na floresta profunda:

```
class DepthFirstSearch
{
// ...
public: // classes
  class IsDescendantOf: public Predicate<int> {
  private:
    const DepthFirstSearch& dfs;
    int vertex;
  public:
    IsDescendantOf(const DepthFirstSearch& dfs, int vertex);
    virtual ~IsDescendantOf();
    virtual const Predicate<int>* Clone() const;
    virtual bool Good(const int& x) const;
  };
};
```

Sim, é uma classe interna da classe DepthFirstSearch.

```
IteratorSmart<int> DepthFirstSearch::Descendants(int x) const
{
  return IteratorSmart<int>(new FromTo(0, graph.CountVertices()), IsDescendantOf(*this, x));
}
```

Claro!

Construindo a busca profunda

A construção é simples:

```
DepthFirstSearch::DepthFirstSearch(const Graph& graph):
  graph(graph),
  color(graph.CountVertices()),
  predecessor(graph.CountVertices()),
  discoveryTime(graph.CountVertices()),
  finishTime(graph.CountVertices()),
  finished(),
  roots(),
  time(0),
  countBackEdges(0)
{
}

DepthFirstSearch::~DepthFirstSearch()
{
}
```

Vectores.

Listas.

Números inteiros.

Calculando profundamente

Visitamos os vértices brancos, pela sequência indicada no iterador:

```
void DepthFirstSearch::Compute(Iterator<int>& i)
{
  color.Fill(WHITE);
  discoveryTime.Fill(-1);
  finishTime.Fill(-1);
  predecessor.Fill(-1);
  roots.Clear();
  finished.Clear();
  time = 0;
  countBackEdges = 0;
  for (; i; i++)
    if (color[*i] == WHITE)
    {
      Visit(*i);
      roots.Put(*i);
    }
}
```

Por defeito, visitamos pela ordem natural:

```
void DepthFirstSearch::Compute()
{
  Compute(FromTo(0, graph.CountVertices() - 1));
}
```


Visitando recursivamente

Quem faz o trabalho todo é a função Visit:

Cf. Programação I.

```
void DepthFirstSearch::Visit(int x)
{
  color[x] = GRAY;
  discoveryTime[x] = time++;
  for (Iterator<int>& i = graph.Successors(x); i; i++)
    if (color[*i] == WHITE)
    {
      predecessor[*i] = x;
      Visit(*i);
    }
    else if (color[*i] == GRAY)
      countBackEdges++;
  color[x] = BLACK;
  finishTime[x] = time++;
  finished.PutFirst(x);
}
```

Note bem: em cada momento, os vértices cinzentos formam o conjunto dos ascendentes do vértice visitado na árvore profunda.

Aqui está a busca profunda: se encontramos um vértice branco, seguimos para ele.

Se um vértice cinzento tem uma ligação para outro vértice cinzento, só pode ser uma aresta retrógrada.

Quando acabamos de processar um vértice, colocamo-lo à cabeça da lista finished.

Teorema parentético e seu corolário

Introduction to Algorithms, página 543.

Teorema

Na busca profunda de um grafo, dados dois vértices x e y, verifica-se uma e uma só das seguintes três condições:

- Os intervalos [discovery(x)..finish(x)] e [discovery(y)..finish(y)] são disjuntos e nem x é descendente de y, nem y é descendente de x na floresta profunda.
- O intervalo [discovery(x)..finish(x)] está contido no intervalo [discovery(y)..finish(y)] e x é um descendente de y na floresta profunda.
- O intervalo [discovery(y)..finish(y)] está contido no intervalo [discovery(x)..finish(x)] e y é um descendente de x na floresta profunda.

Introduction to Algorithms, página 545.

Corolário

Um vértice x é descendente de um vértice y (em sentido estrito) se e só se discovery(y) < discovery(x) < finish(x) < finish(y).

Descendentes na floresta

Eis as funções da classe predicado IsDescendentOf:

Repare que se trata de uma classe interna.

```
DepthFirstSearch::IsDescendantOf::IsDescendantOf(const DepthFirstSearch& dfs, int vertex):
  dfs(dfs),
  vertex(vertex)
{
}

DepthFirstSearch::IsDescendantOf::~IsDescendantOf()
{
}

const Predicate<int>* DepthFirstSearch::IsDescendantOf::Clone() const
{
  return new IsDescendantOf(*this);
}

bool DepthFirstSearch::IsDescendantOf::Good(const int& x) const
{
  return MyUtil::Sorted(dfs.discoveryTime[vertex], dfs.discoveryTime[x], dfs.finishTime[vertex]);
}
```

Por aplicação directa do corolário.

Ordenação topológica

Uma ordenação topológica de um grafo orientado acíclico é uma sequência de vértices tal que se o grafo contém uma aresta (x, y), então x aparece antes de y na sequência.

Por exemplo: pense no grafo das precedências das cadeiras de um curso: Análise 2 tem precedência de Análise 1; AED1 tem precedência de Programação 2; etc. Uma ordem topológica neste grafo corresponde a uma sequência possível para fazer as cadeiras. Em geral, num grafo cíclico, haverá muitas ordens topológicas.

Num grafo orientado acíclico, se há uma aresta de x para y então x termina (fica preto) depois de y. Logo:

```
const ListDouble<int>& DepthFirstSearch::Topological() const
{
  return finished;
}
```

Recorde que a lista finished tem os vértices por ordem inversa da hora em que ficaram pretos.

Grafos orientados acíclicos

Lema

Um grafo orientado é acíclico se e só se a busca profunda não encontrar arestas retrógradas.

Introduction to Algorithms, página 550.

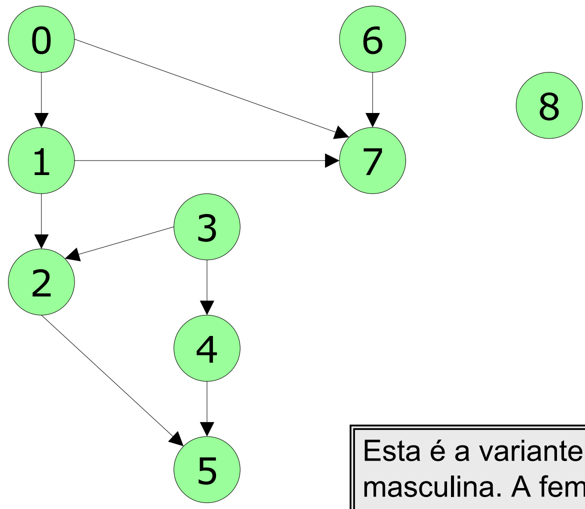
Logo:

```
bool DepthFirstSearch::IsAcyclic() const
{
  return countBackEdges == 0;
}
```

Afinal não é complicado verificar se um grafo orientado tem ou não tem ciclos.

Vestindo-se de manhã

0: cuecas
1: calças
2: cinto
3: camisa
4: gravata
5: casaco
6: meias
7: sapatos
8: relógio



Esta é a variante masculina. A feminina é ligeiramente diferente. ☺

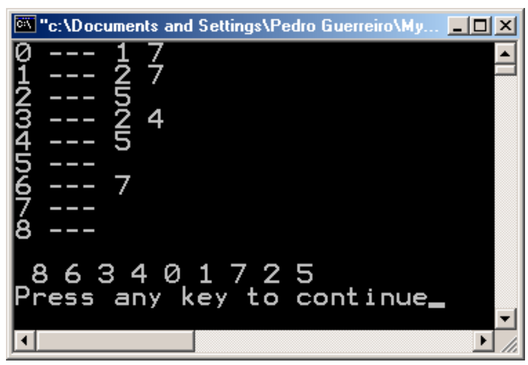
Problema: calcular uma ordem topológica para isto.

Ordem topológica matinal

Eis a função de teste para o problema " vestindo-se de manhã":

```
void TestGettingDressed()
{
  GraphUsingList g(9);
  g.Link(0, 1);
  g.Link(0, 7);
  g.Link(1, 2);
  g.Link(1, 7);
  g.Link(2, 5);
  g.Link(3, 2);
  g.Link(3, 4);
  g.Link(4, 5);
  g.Link(6, 7);
  g.Write();
  DepthFirstSearch dfs(g);
  dfs.Compute();
  std::cout << std::endl;
  ListDouble<int>::SetPrefixSuffix(" ", "");
  dfs.Topological().WriteLine();
}
```

Resultado:



Ou seja: relógio, meias, camisa, gravata, cuecas, calças, sapatos, cinto e, finalmente, casaco.

Análise da busca profunda

A função Visit é chamada uma vez para cada vértice, uma vez que só é chamada para vértices brancos, os quais passam imediatamente a cinzento.

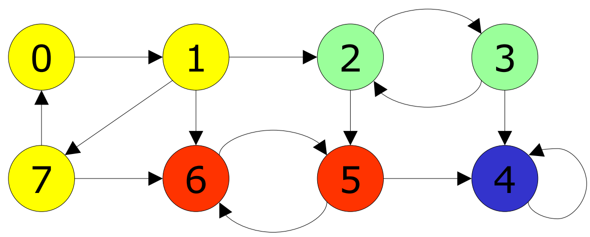
Por outro lado, em cada chamada de Visit, o iterador gera tantos elementos quantos os sucessores do vértice visitado. Admitindo a implementação com listas, em que o tempo de iteração é proporcional ao número de nós da lista, somando para todos os vértices, dá o número de arestas.

A inicialização consiste no preenchimento linear de vectores que têm tantos elementos quantos os vértices

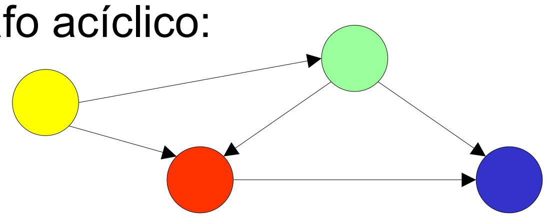
Juntando tudo, e tal como na busca larga, concluímos que o tempo é proporcional à soma do número de vértices com o número de arestas.

Componentes fortemente conexas

Uma componente fortemente conexa de um grafo é um conjunto maximal de vértices tal que cada dois elementos desse conjunto são alcançáveis a partir um do outro.



Contraindo as arestas que ligam as componentes fortemente conexas e guardando apenas um vértice de cada componente, obtemos um grafo acíclico:



Calculando-as

Faz-se assim:

Primeiro, carimbam-se os vértices, usando a busca profunda.

A seguir, faz-se uma busca profunda no grafo transposto, visitando os vértices por ordem decrescente de carimbo final.

As árvores da floresta calculada na segunda busca profunda são as componentes fortemente conexas. ☺

Introduction to Algorithms, página 554.

Classe StronglyConnectedComponents

Talvez assim:

```
class StronglyConnectedComponents: public DepthFirstSearch {
private:
  Graph* graph;
public:
  StronglyConnectedComponents(const Graph& graph);
  virtual ~StronglyConnectedComponents();

  virtual void Compute();
};
```

Só?

Classe Strongly..., implementação

Construtor, destrutor:

```
StronglyConnectedComponents::StronglyConnectedComponents(const Graph& graph):
  DepthFirstSearch(graph),
  graph(graph.Clone())
{
}

StronglyConnectedComponents::~StronglyConnectedComponents()
{
  delete graph;
}
```

Cálculo:

```
void StronglyConnectedComponents::Compute()
{
  graph->Transpose();
  DepthFirstSearch dfs(*graph);
  dfs.Compute();
  DepthFirstSearch::Compute(dfs.Topological().Items());
}
```

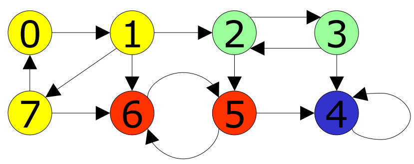
Na verdade, estamos a carimbar os vértices do grafo transposto e a fazer a segunda busca no grafo original, o que, não sendo exactamente o esquema indicado em *Introduction to Algorithms*, é equivalente, porque um grafo e o seu transposto têm as mesmas componentes fortemente conexas, claro.

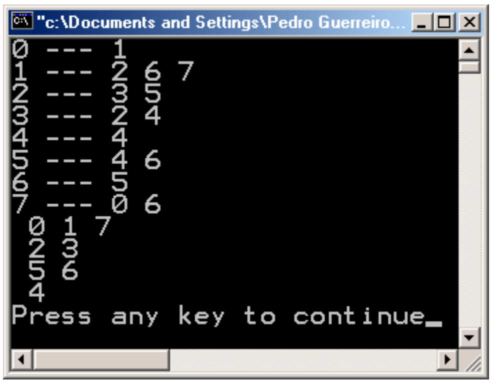
Recorde que dfs.Topological() é uma lista que regista os vértices por ordem inversa de carimbo final.

Experimentando

Função de teste:

```
void TestStronglyConnectedComponents()
{
  GraphUsingList g(8);
  g.Link(0, 1);
  g.Link(1, 2);
  g.Link(1, 6);
  g.Link(1, 7);
  g.Link(2, 3);
  g.Link(2, 5);
  g.Link(3, 2);
  g.Link(3, 4);
  g.Link(4, 4);
  g.Link(5, 4);
  g.Link(5, 6);
  g.Link(6, 5);
  g.Link(7, 0);
  g.Link(7, 6);
  g.Write();
  StronglyConnectedComponents scc(g);
  scc.Compute();
  for (Iterator<int>& i = scc.Roots().Items(); i; i++)
    Output<int>(" ", "").PutItems(scc.Descendants(*i));
}
```

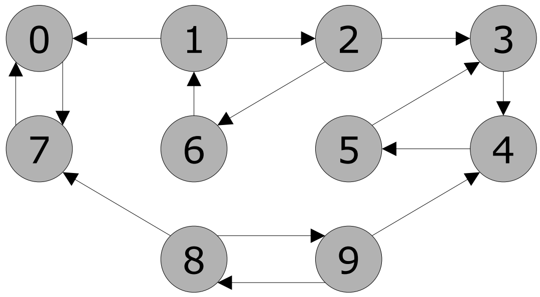




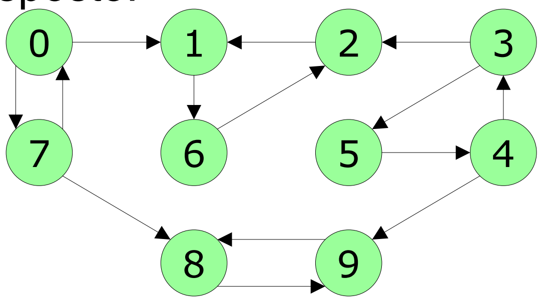
Completamente espectacular!

Percebendo as componentes conexas

Analisemos o funcionamento do algoritmo que calcula as componentes fortemente conexas, com um exemplo:

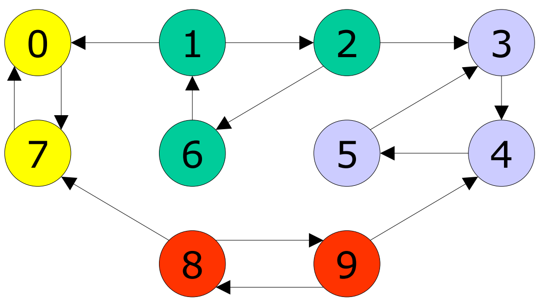


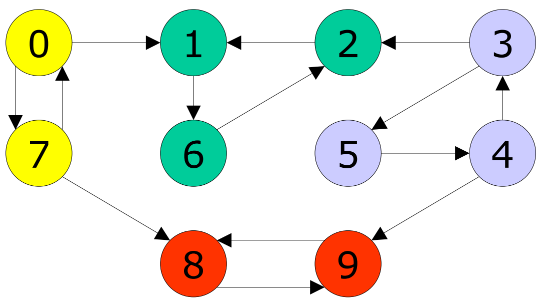
Este é o grafo transposto:



Observações (1)

Primeira observação: as componentes fortemente conexas de um grafo e do seu transposto são as mesmas.

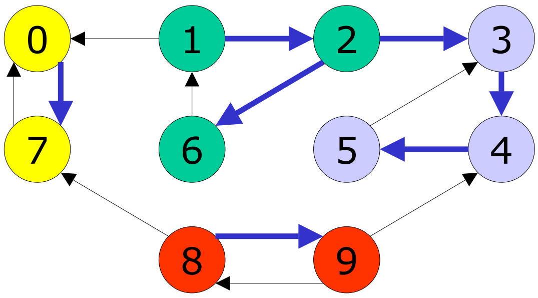




Logo, tanto faz começarmos pelo grafo ou pelo seu transposto.

Observações (2)

Segunda observação: cada árvore profunda contém a reunião de algumas componentes fortemente conexas.



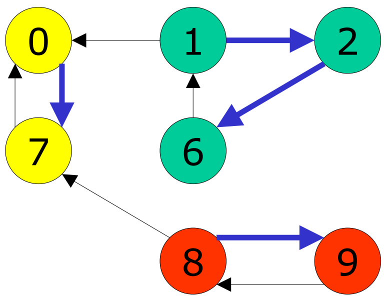
Temos aqui três árvores, com raízes 0, 1 e 8. A segunda abarca duas componentes fortemente conexas, as outras uma cada. Note bem: nunca pode haver uma componente que fique dividida entre duas árvores. Isto é evidente.

Logo, se começarmos por fazer uma busca profunda, podemos depois calcular as componentes dentro de cada árvore profunda, árvore a árvore. Mas começando por qual?

Observações (3)

Terceira observação: se retirarmos de um grafo todos os vértices de uma componente fortemente conexa e todas as arestas que deles saem ou neles entram, as restantes componentes fortemente conexas ficam na mesma.

Isto é uma consequência directa da própria definição.

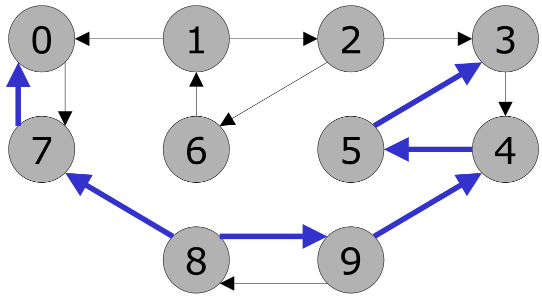


Retirámos a componente cinzenta. As outras três mantêm-se.

Logo, depois de detectarmos uma componente fortemente conexa, podemos retirá-la e prosseguir como se ele nunca tivesse existido.

Começando pela última

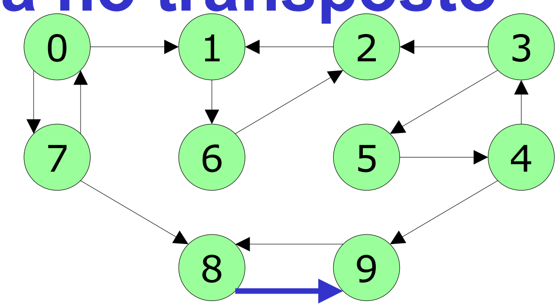
Façamos uma busca profunda começando pela última árvore profunda, com raiz no vértice 8. Obtemos o seguinte:



Claramente, não é isto que queremos, pois esta árvore abarca três das componentes fortemente conexas. ☹

Começando pela última no transposto

Se começarmos no vértice 8 mas no grafo transposto, já não saímos da componente fortemente conexa:



O vértice 8 é a raiz da última árvore encontrada na busca profunda do grafo original.

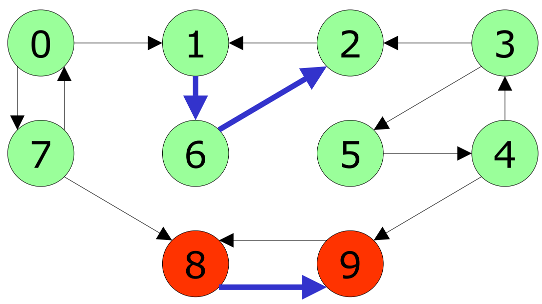
Se pudéssemos sair da componente fortemente conexa a que pertence o vértice 8 no grafo transposto, isso quereria dizer que havia no grafo original uma aresta incidente num dos vértices desta componente vinda de fora da componente. Então, das duas uma: essa aresta viria ou de outra árvore de busca ou da árvore cuja raiz é 8. O primeiro caso é impossível, porque então 8 não seria a raiz da última árvore de busca, já que era alcançável a partir de uma outra árvore, a qual teria sido visitada anteriormente. O segundo caso também é impossível, pois o vértice de onde partiria a aresta faria parte da componente, já que seria alcançável a partir do 8 (e, portanto, de todos os outros vértices da componente), o que é uma contradição, pois partimos do princípio de que esse vértice não pertencia à componente.

Generalizando

Generalizando o raciocínio anterior, concluímos que os vértices da árvore de busca no grafo transposto cuja raiz é a raiz da última árvore de busca no grafo original formam uma componente fortemente conexa.

Continuando (1)

No caso do exemplo, a árvore cuja raiz é 8 no grafo transposto coincide com a árvore no grafo original e corresponde a uma componente fortemente conexa. Podemos, por isso, retirar conceptualmente dos grafos (original e transposto) essa componente e prosseguir, com o mesmo raciocínio, para as restantes, isto é, fazendo uma busca no grafo transposto a partir do vértice 1, como se a componente (8, 9) nunca tivesse existido.



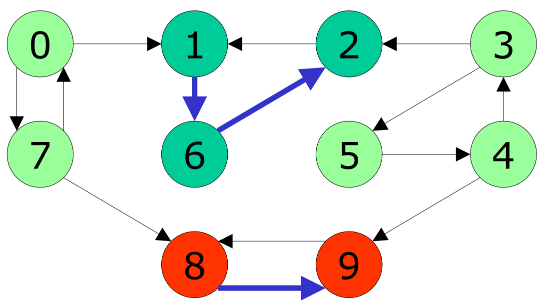
A árvore cuja raiz é 1 representa outra componente, mas não coincide com a árvore correspondente no grafo original.

Retirando as componentes

Conceptualmente, estamos sempre a processar no grafo transposto os vértices da última árvore de busca no grafo original, pois, de cada vez que esgotamos uma árvore de busca, retiramo-la das nossas preocupações (com as componentes já calculadas) e continuamos com a árvore precedente, que é agora a última (uma vez que as componentes já foram retiradas), repetindo o processamento e usando o mesmo raciocínio.

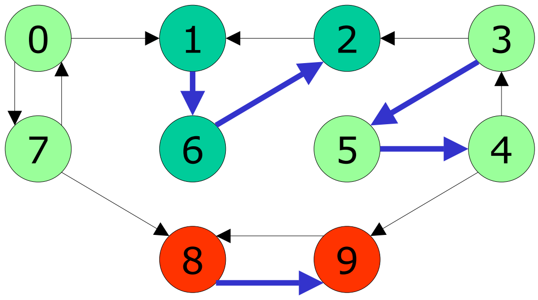
Continuando (2)

Se detectamos uma componente antes de esgotar a árvore, podemos conceptualmente retirá-la da árvore, tal como no caso em que a árvore inclui uma só componente, pois essa componente não interfere nos processamentos subsequentes, e podemos prosseguir como se ela nunca tivesse existido. Logo, após ter detectado a componente (1, 2, 6), continuamos a partir do vértice 3.

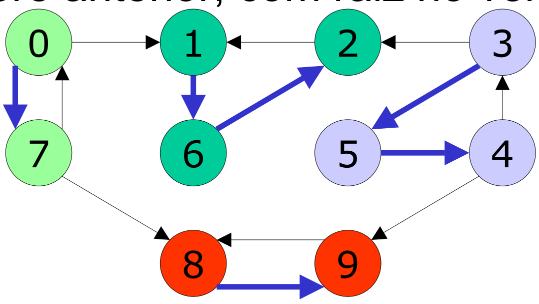


Etc.

Agora fazemos uma busca a partir do vértice 3:

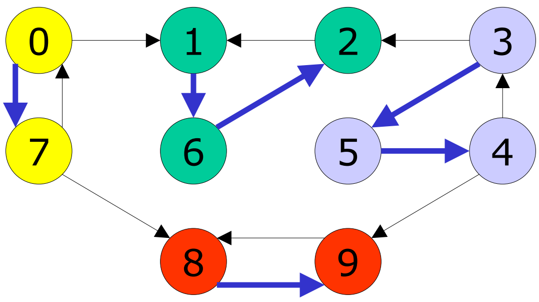


Isto esgota a árvore que no grafo original tinha a raiz no vértice 1. Passamos à árvore anterior, com raiz no vértice 0:



Concluindo

Isto leva-nos à configuração final, com as quatro componentes fortemente conexas devidamente detectadas:



Como observámos na página 291, o nosso programa carimba o grafo transposto e faz a segunda busca no grafo original, mas o efeito é o mesmo.

O processamento que descrevemos corresponde ao algoritmo da página 289: primeiro, carimbam-se os vértices, usando a busca profunda; a seguir, faz-se uma busca profunda no grafo transposto, visitando os vértices por ordem decrescente de carimbo final; as árvores da floresta calculada na segunda busca profunda são as componentes fortemente conexas.

Grafos pesados

Num grafo pesado, as arestas pesam. O peso é um número. Mais correctamente, o peso é uma função das arestas nos números reais. Representamos os grafos pesados com a classe abstracta GraphWeighted:

```
class GraphWeighted: public Graph {
public:
  virtual ~GraphWeighted();

  virtual Graph* Clone() const = 0;
  virtual double Weight(int x, int y) const = 0;
  virtual void SetWeight(int x, int y, double w) = 0;
  virtual void Link(int x, int y);
  virtual void Link(int x, int y, double w) = 0;
  virtual void LinkUndirected(int x, int y);
  virtual void LinkUndirected(int x, int y, double w);

  virtual void Write(std::ostream& = std::cout) const;
};
```

Sendo abstracta, esta classe não se compromete com a maneira de representar os pesos das arestas.

As três funções Link que não são virtuais puras programam-se em termos da outra.

Classe GraphWeighted, impl...

Por defeito, o peso é 1:

```
void GraphWeighted::Link(int x, int y)
{
  Link(x, y, 1.0);
}
```

```
void GraphWeighted::LinkUndirected(int x, int y)
{
  LinkUndirected(x, y, 1.0);
}
```

Por defeito, o peso é o mesmo nos dois sentidos:

```
void GraphWeighted::LinkUndirected(int x, int y, double w)
{
  Link(x, y, w);
  Link(y, x, w);
}
```

A função Write é usada para *debug*. Mostramos o peso entre parêntesis, a seguir ao vértice.

```
void GraphWeighted::Write(std::ostream& output) const
{
  for (int i = 0; i < CountVertices(); i++)
  {
    output << i << " ---";
    for (Iterator<int>& j = Successors(i); j; j++)
      output << " " << *j << "(" << Weight(i, *j) << ")";
    output << std::endl;
  }
}
```

Grafos pesados, com listas (1)

Usaremos uma classe GraphWeightedUsingList, em que o grafo é representado por um vector de listas, como antes, e os pesos por uma matriz de números reais:

```
class GraphWeightedUsingList: public GraphUsingList, public GraphWeighted {
private:
  Matrix<double> weight;
public:
  GraphWeightedUsingList(int countVertices);
  GraphWeightedUsingList(const GraphWeightedUsingList& other);
  virtual ~GraphWeightedUsingList();

  virtual Graph* Clone() const;

  virtual double Weight(int x, int y) const;
  virtual void SetWeight(int x, int y, double w);
  virtual void Link(int x, int y, double w);

  // ...
};
```

A classe Matrix é uma nova classe. Ver um pouco mais à frente.

Este é um caso de herança múltipla em losango, pois as duas classes de base derivam de Graph. Não precisamos de herança virtual porque a classe Graph não tem membros de dados.

Grafos pesados, com listas (2)

As funções virtuais puras herdadas da classe Graph têm de ser redefinidas, mesmo se já estão implementadas na classe GraphUsingList:

```
class GraphWeightedUsingList: public GraphUsingList, public GraphWeighted {
// ...
  virtual int CountVertices() const;
  virtual bool ValidVertex(int x) const;
  virtual bool IsLinked(int x, int y) const;
  virtual bool HasSuccessors(int x) const;
  virtual int CountEdges() const;
  virtual void Clear();
  virtual void Transpose();

  virtual IteratorSmart<int> Successors(int x) const;

  virtual void Write(std::ostream& output = std::cout) const;
};
```

Matrizes

Matrizes são vectores organizados em linhas e colunas. Todas as linhas têm o mesmo número de elementos, estabelecido na construção:

```
template <class T>
class Matrix: public Vector<T> {
private:
  int countRows;
  int countColumns;
public:
  Matrix(int countRows, int countColumns);
          // pre countRows > 0 && countColumns > 0;
  Matrix(const Matrix<T>& other);
  virtual ~Matrix();

  virtual void Copy(const Matrix<T>& other);
  virtual Matrix<T>& operator = (const Matrix<T>& other);

  virtual int CountRows() const;
  virtual int CountColumns() const;
  virtual bool ValidIndices(int i, int j) const;
  virtual const T& At2(int i, int j) const;
  virtual T& At2(int i, int j);
  virtual void PutAt2(const T& x, int i, int j);

  virtual void Transpose();
  virtual void SwapOut(Matrix<T>& other);

  virtual void Write(std::ostream& output = std::cout) const;

  virtual int Index(int i, int j) const;
};
```

Repare nas funções At2 e PutAt2, com dois índices.

A função SwapOut troca o objecto com o argumento, internamente (isto é, funcionando ao nível da representação)

Índice no vector correspondente ao par de índices (i, j) na matriz.

Classe Matrix<T>, implementação (1)

Construtores, destrutor, Copy, afectação:

```
template <class T>
Matrix<T>::Matrix(int countRows, int countColumns):
  Vector<T>(countRows * countColumns),
  countRows(countRows),
  countColumns(countColumns)
{
  Fill();
}

template <class T>
Matrix<T>::Matrix(const Matrix<T>& other):
  Vector<T>(other),
  countRows(other.countRows),
  countColumns(other.countColumns)
{
}

template <class T>
Matrix<T>::~Matrix()
{
}
```

Nova técnica para fazer a cópia: cria-se um temporário com uma cópia do argumente e troca-se com o objecto. O objecto velho é destruído por via do temporário, quando este desaparece.

```
template <class T>
void Matrix<T>::Copy(const Matrix<T>& other)
{
  Matrix temp(other);
  SwapOut(temp);
}

template <class T>
Matrix<T>& Matrix<T>::operator = (const Matrix<T>& other)
{
  Copy(other);
  return *this;
}
```

Função SwapOut

Eis a função SwapOut da classe Matrix<T>:

```
template <class T>
void Matrix<T>::SwapOut(Matrix<T>& other)
{
  Vector<T>::SwapOut(other);
  std::swap(countRows, other.countRows);
  std::swap(countColumns, other.countColumns);
}
```

A função std::swap é uma função genérica da biblioteca do C++ que troca os valores dos argumentos (de qualquer tipo com afectação).

Ela baseia-se na função SwapOut da classe Vector<T>:

```
template <class T>
class Vector: public VectorAbstract<T> {
// ...
 virtual void SwapOut(Vector<T>& other);
}
```

```
template <class T>
void Vector<T>::SwapOut(Vector<T>& other)
{
  std::swap(items, other.items);
  std::swap(capacity, other.capacity);
  std::swap(count, other.count);
  std::swap(growFactor, other.growFactor);
}
```


Classe Matrix<T>, implementação (2)

Quantas linhas?

```
template <class T>
int Matrix<T>::CountRows() const
{
  return countRows;
}
```

Quantas colunas?

```
template <class T>
int Matrix<T>::CountColumns() const
{
  return countColumns;
}
```

Acesso por par de índices, const e não const:

```
template <class T>
const T& Matrix<T>::At2(int i, int j) const
{
  return At(Index(i, j));
}
```

```
template <class T>
T& Matrix<T>::At2(int i, int j)
{
  return At(Index(i, j));
}
```

Modificação por par de índices:

```
template <class T>
void Matrix<T>::PutAt2(const T& x, int i, int j)
{
  PutAt(x, Index(i, j));
}
```

Índice no vector

```
template <class T>
int Matrix<T>::Index(int i, int j) const
{
  return i * countColumns + j;
}
```

Validação do par de índices:

```
template <class T>
bool Matrix<T>::ValidIndices(int i, int j) const
{
  return 0 <= i && i < countRows && 0 <=j && j < countColumns;
}
```

Transpondo a matriz

Usamos uma matriz auxiliar e depois trocamos:

```
template <class T>
void Matrix<T>::Transpose()
{
  Matrix temp(countColumns, countRows);
  for (int i = 0; i < countRows; i++)
    for (int j = 0; j < countColumns; j++)
      temp.PutAt2(At2(i, j), j, i);
  SwapOut(temp);
}
```

Mais um exemplo da técnica do SwapOut. É com certeza muito melhor do que fazer uma cópia.

Classe GraphWeightedUsingList (1)

Construtores, destrutor, Clone:

```
GraphWeightedUsingList::GraphWeightedUsingList(int countVertices):
  GraphUsingList(countVertices),
  weight(countVertices, countVertices)
{
}

GraphWeightedUsingList::GraphWeightedUsingList(const GraphWeightedUsingList& other):
  GraphUsingList(other),
  weight(other.weight)
{
}

GraphWeightedUsingList::~GraphWeightedUsingList()
{
}

Graph* GraphWeightedUsingList::Clone() const
{
  return static_cast<GraphWeighted*>(new GraphWeightedUsingList(*this));
}
```

A matriz dos pesos é quadrada.

Este static_cast é apenas para eliminar a ambiguidade, pois a classe deriva de Graph por duas vias.

Classe GraphWeightedUsingList (2)

As outras funções recorrem às funções homónimas herdadas de GraphUsingList, com pequenas adaptações, quando afectam a matriz dos pesos.

```
void GraphWeightedUsingList::Link(int x, int y, double w)
{
   GraphUsingList::Link(x, y);
   SetWeight(x, y, w);
}

int GraphWeightedUsingList::CountVertices() const
{
  return GraphUsingList::CountVertices();
}

bool GraphWeightedUsingList::ValidVertex(int x) const
{
  return GraphUsingList::ValidVertex(x);
}

bool GraphWeightedUsingList::IsLinked(int x, int y) const
{
  return GraphUsingList::IsLinked(x, y);
}

bool GraphWeightedUsingList::HasSuccessors(int x) const
{
  return GraphUsingList::HasSuccessors(x);
}
```

Sim, isto é um bocado monótono.

Classe GraphWeightedUsingList (3)

O resto:

```
int GraphWeightedUsingList::CountEdges() const
{
  return GraphUsingList::CountEdges();
}

void GraphWeightedUsingList::Clear()
{
  GraphUsingList::Clear();
  weight.Clear();
}

void GraphWeightedUsingList::Transpose()
{
  GraphUsingList::Transpose();
  weight.Transpose();
}

IteratorSmart<int> GraphWeightedUsingList::Successors(int x) const
{
  return GraphUsingList::Successors(x);
}

void GraphWeightedUsingList::Write(std::ostream& output) const
{
  GraphWeighted::Write(output);
}
```

Árvores de cobertura

Problema: suponha que temos *N* computadores numa sala e queremos ligá-los usando *N*-1 cabos. Como fazer de maneira a que o comprimento total do cabo seja mínimo?

Problema geral: dado um grafo não orientado pesado com *N* vértices escolher as *N*-1 arestas que permitem ligar todos os vértices, de maneira que o peso total das arestas escolhidas seja mínimo.

Em inglês, *spanning tree*.

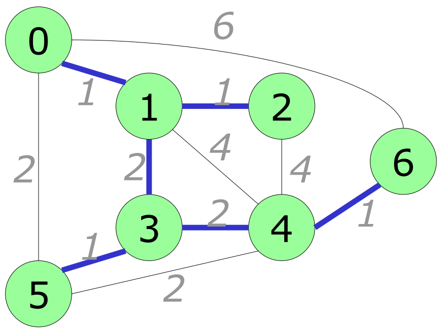
Num grafo não orientado, uma *árvore* é um subgrafo que não contém ciclos. Uma *árvore de cobertura* é uma árvore que contém todos os vértice

Em inglês, *minimum spanning tree*.

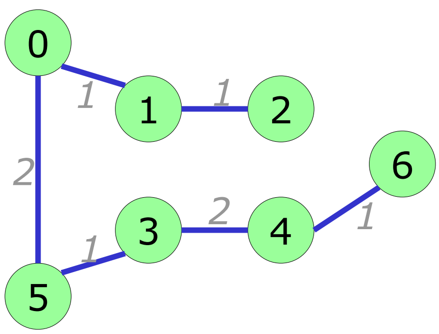
Uma *árvore de cobertura mínima* é uma árvore de cobertura tal que a soma dos pesos das arestas é menor do que a soma dos pesos das arestas de qualquer outro conjunto de arestas que ligam todos os vértices do grafo.

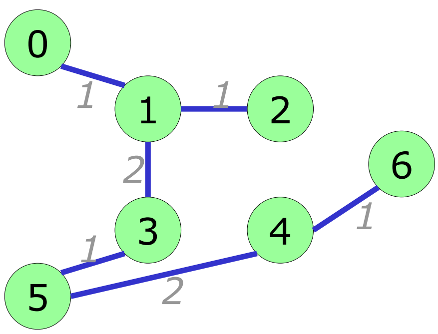
Árvores de cobertura, exemplos

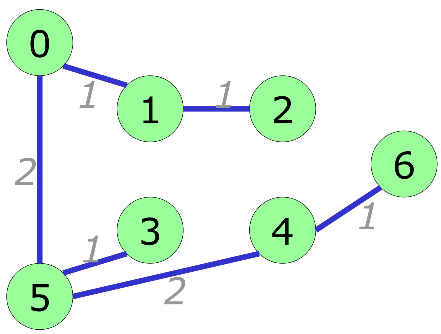
Eis um grafo pesado, com uma árvore de cobertura mínima assinalada a azul:



Pode haver várias árvores de cobertura mínima. Eis mais três:



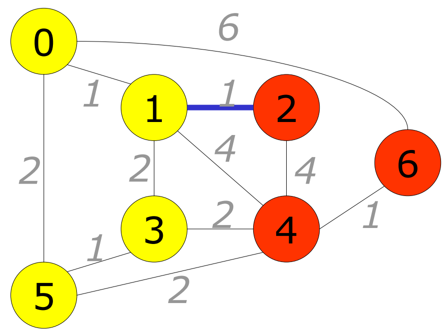




Propriedade fundamental das árvores de cobertura mínima

Qualquer que seja a partição do conjunto de vértices de um grafo em dois conjuntos, a árvore de cobertura mínima contém a aresta menos pesada das que ligam um vértice de um conjunto a um vértice de outro.

Por exemplo, neste grafo a aresta (1, 2) tem de pertencer à arvore de cobertura mínima, pois é a menos pesada das que ligam os vértices da esquerda (amarelos) aos da direita (vermelhos):



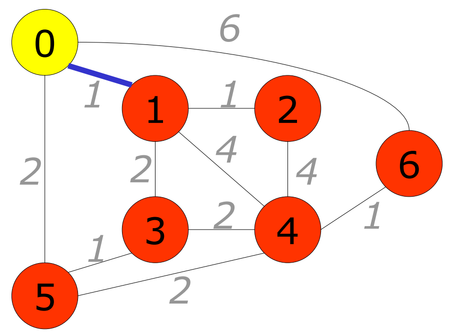
Suponha que a aresta (1, 2) não pertencia à arvore de cobertura mínima. Então haveria uma outra aresta entre os amarelos e os vermelhos nessa suposta árvore de cobertura mínima. Juntemos a aresta (1, 2) a esta árvore. O subgrafo assim obtido tem um ciclo, e esse ciclo deve ter uma outra aresta ligando os amarelos aos vermelhos. Se retirarmos essa outra aresta voltamos a ter uma árvore de cobertura menos pesada do que a inicial. Logo a árvore inicial não era a árvore de cobertura mínima.

Algoritmo de Prim

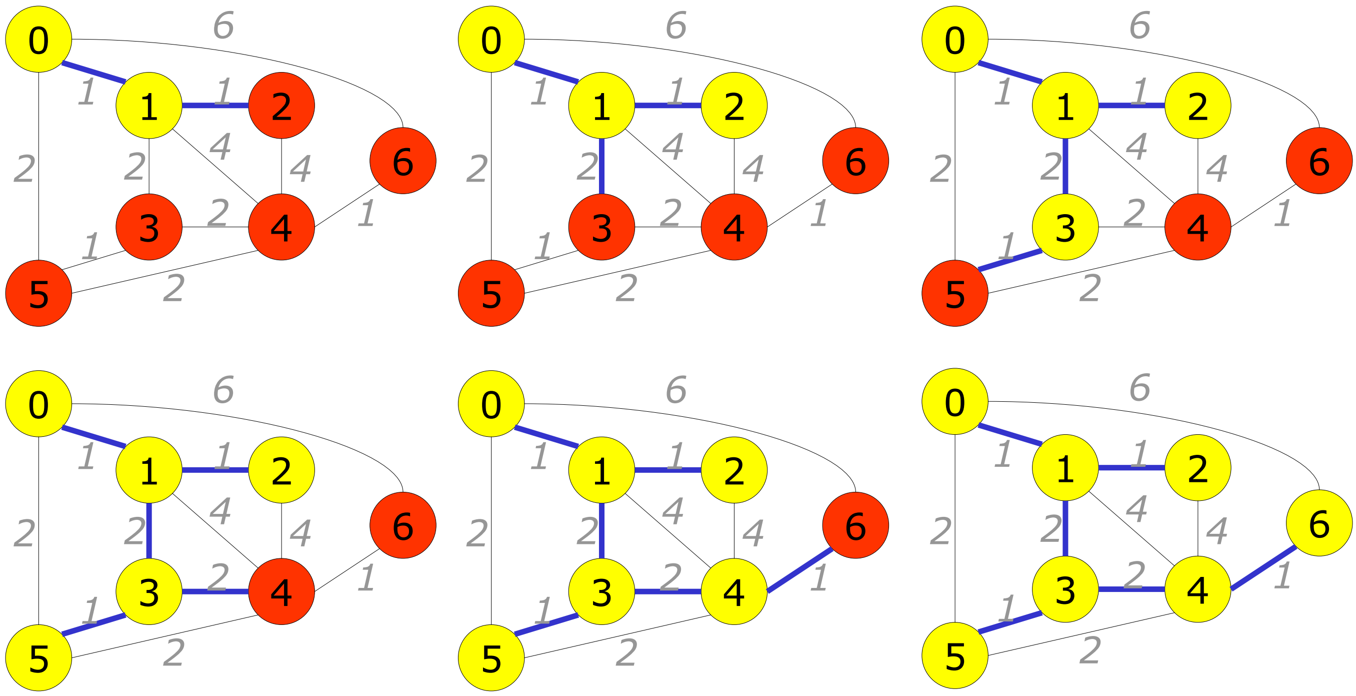
É uma aplicação da propriedade fundamental. Começa-se com um vértice qualquer e em cada passo junta-se à árvore a aresta menos pesada das que ligam o conjunto dos vértices que já estão na árvore ao conjunto dos vértices que ainda não estão na árvore.

É um algoritmo *ganancioso*, porque em cada passo toma a decisão mais favorável no momento.

Vejamos o filme dos acontecimentos, começando pelo vértice zero. Os vértices que já estão na árvore ficam amarelos, os outros vermelhos. De todas as arestas que ligam amarelos a vermelhos nesta fase, a menos pesada é a aresta (0, 1):



Filme do algoritmo de Prim (cont.)



Usando uma fila com prioridades

Para programar o algoritmo de Prim com eficiência, convém ter os vértices num fila com prioridades, de maneira a que o mais prioritário seja aquele cuja distância a um dos vértices já na árvore é a menor.

Após um vértice entrar na árvore, temos de actualizar as prioridades de todos os seus sucessores que ainda não estão na árvore.

Isto obriga a retocar a classe PriorityQueue<T>, de maneira a poder consultar e mudar a prioridade de um elemento que está na fila.

E isso envolve aumentar a classe HeapPolymorphic<T>, com uma função para mudar um elemento no monte, fazendo--o subir ou descer, consoante o seu valor diminua ou aumente.

Classe HeapPolymorphic<T>, **bis**

Novas funções, para substituir um elemento por outro, para obter a posição de um elemento e para obter o valor de um elemento dada a posição:

```
template <class T>
class HeapPolymorphic: public Dispenser<T> {
private:
  VectorPolymorphic<T> items;
public:
  //...
  virtual void Replace(const T& x, const T& y);
  virtual IndexOf(const T& x) const;
  virtual const T& At(int x) const;
};
```

Definição:

```
template <class T>
void HeapPolymorphic<T>::Replace(const T& x, const T& y)
{
  int k = IndexOf(x);
  if (k != -1)
  {
    items[k] = y;
    Increase(k);
    Decrease(k);
  }
}
```

```
template <class T>
int HeapPolymorphic<T>::IndexOf(const T& x) const
{
  return items.IndexOf(x);
}
```

```
template <class T>
const T& HeapPolymorphic<T>::At(int x) const
{
  return items.At(x);
}
```


Nova classe PriorityQueue<T>

A prioridade é um número real:

```
template <class T>
class PriorityQueue: public Dispenser<T> {
private:
  HeapPolymorphic<Prioritized<T> > items;
public:
  //...
// from Container<T>
  //...
// from Dispenser<T>
  //...
// declared now
  virtual double Priority() const;  // pre !Empty();
  virtual void Add(const T& s, double x);
  virtual void Raise(const T& s, double x);
  virtual bool Has(const T& s) const;
  virtual double PriorityOf(const T& s) const;   // pre Has(x);

  virtual int IndexOf(const T& s) const;
  virtual double PriorityAt(int x) const;
};
```

Prioridade do elemento mais prioritário.

Acrescentar à fila um elemento com a prioridade indicada.

Aumentar (ou diminuir) a prioridade.

Prioridade do elemento indicado.

Prioridade do elemento na posição indicada.

Nova classe PriorityQueue<T>, **impl...**

A função Put acrescenta um elemento com prioridade mínima:

```
template <class T>
void PriorityQueue<T>::Put(const T& s)
{
  items.Put(Prioritized<T>(s));
}
```

A prioridade é mínima por defeito, na classe Prioritized<T>. (V. adiante.)

A prioridade é a do primeiro elemento:

```
template <class T>
double PriorityQueue<T>::Priority() const
{
  return items.Item().Priority();
}
```

Para aumentar a prioridade, substitui-se o elemento na fila por um novo elemento com o mesmo valor mas com a nova prioridade.

```
template <class T>
void PriorityQueue<T>::Raise(const T& s, double x)
{
  items.Replace(Prioritized<T>(s), Prioritized<T>(s, x));
}
```


Restantes funções

As restantes funções novas são auxiliares:

```
template <class T>
bool PriorityQueue<T>::Has(const T& x) const
{
  return IndexOf(x) != -1;
}

template <class T>
double PriorityQueue<T>::PriorityOf(const T& x) const
{
  return PriorityAt(IndexOf(x));
}

template <class T>
int PriorityQueue<T>::IndexOf(const T& x) const
{
  return items.IndexOf(Prioritized<T>(x, 0));
}

template <class T>
double PriorityQueue<T>::PriorityAt(int x) const
{
  return items.At(x).Priority();
}
```

A função IndexOf realiza uma busca linear (na classe VectorAbstract<T>). Nalguns algoritmos podemos querer esquemas mais eficientes.

Nova classe Prioritized<T>

```
template <class T>
class Prioritized: public Clonable {
private:
  Pair<T, double> item;
public:
  Prioritized();
  Prioritized(const T& x, double priority = -std::numeric_limits<double>::infinity());
  Prioritized(const Prioritized<T>& other);
  virtual ~Prioritized();

  virtual Clonable* Clone() const;

  virtual void SetPriority(double priority);
  virtual const T& Item() const;
  virtual T& Item();
  virtual double Priority() const;
  virtual bool operator <= (const Prioritized<T>& other) const;
  virtual bool operator == (const Prioritized<T>& other) const;

  virtual void Write(std::ostream& output = std::cout) const;
  friend std::ostream& operator << (std::ostream& output, const Prioritized<T>& x)
    {x.item.Write(output); return output;};
};
```

Esta classe substitui a anterior, que fica obsoleta.

Pair<T, double>, pois a prioridade é um número real.

Por defeito, a prioridade é mínima (neste caso, menos infinito).

Classe Prioritized<T>**, impl... (1)**

Construtores, destrutor, Clone, tudo como de costume:

```
template <class T>
Prioritized<T>::Prioritized():
  item(T(), -std::numeric_limits<double>::infinity())
{
}
```

```
template <class T>
Prioritized<T>::Prioritized(const T& x, double priority):
  item(x, priority)
{
}
```

```
template <class T>
Prioritized<T>::Prioritized(const Prioritized<T>& other):
  item(other.item)
{
}
```

```
template <class T>
Prioritized<T>::Prioritized(const Prioritized<T>& other):
  item(other.item)
{
}
```

```
template <class T>
Prioritized<T>::~Prioritized()
{
}
```

```
template <class T>
Clonable* Prioritized<T>::Clone() const
{
  return new Prioritized<T>(*this);
}
```

Classe Prioritized<T>, impl... (2)

O elemento cuja prioridade registamos (const e não const):

```
template <class T>
const T& Prioritized<T>::Item() const
{
  return item.First();
}
```

```
template <class T>
T& Prioritized<T>::Item()
{
  return item.First();
}
```

A prioridade é o segundo elemento do par:

```
template <class T>
double Prioritized<T>::Priority() const
{
  return item.Second();
}
```

Atenção a isto: a igualdade == é a dos elementos, no tipo T, a menoridade <= é a das prioridades, no tipo double.

```
template <class T>
bool Prioritized<T>::operator <= (const Prioritized<T>& other) const
{
  return other.item.LessThanBySecond(item);
}
```

```
template <class T>
bool Prioritized<T>::operator == (const Prioritized<T>& other) const
{
  return item.EqualsByFirst(other.item);
}
```


Prim, Prim, Prim

Temos tudo para programar o algoritmo de Prim. Vem numa classe, como de costume:

```
class MinimumSpanningTree_Prim
{
private:
  const GraphWeighted& graph;
  Vector<int> predecessor;
  int start;
public:
  MinimumSpanningTree_Prim(const GraphWeighted& graph);
  virtual ~MinimumSpanningTree_Prim();

  virtual void Compute(int start);  // post Computed();
  virtual bool Computed() const;

  virtual const Vector<int>& Predecessors() const; // pre Computed();
  virtual int Start() const; // pre Computed();
  virtual double MinimumWeight() const; // pre Computed();
};
```

A árvore de cobertura mínima vem no vector predecessors.

Peso da árvore de cobertura mínima.

Classe MinimumSpanningTree_Prim

Construtores, destrutor, etc.:

```
MinimumSpanningTree_Prim::MinimumSpanningTree_Prim(const GraphWeighted& graph):
  graph(graph),
  predecessor(graph.CountVertices()),
  start(-1)
{
}

MinimumSpanningTree_Prim::~MinimumSpanningTree_Prim()
{
}
```

```
bool MinimumSpanningTree_Prim::Computed() const
{
  return start != -1;
}

const Vector<int>& MinimumSpanningTree_Prim::Predecessors() const
{
  return predecessor;
}

int MinimumSpanningTree_Prim::Start() const
{
  return start;
}
```

O algoritmo propriamente dito vem na página seguinte.

Função do algoritmo de Prim

Ei-la:

```
void MinimumSpanningTree_Prim::Compute(int start)
{
  predecessor.Fill(-1);
  PriorityQueue<int> queue(graph.CountVertices());
  queue.PutItems(FromTo(0, graph.CountVertices() - 1));
  queue.Raise(start, 0);
  while (!queue.Empty())
  {
    int x = queue.Item();
    queue.Remove();
    for (Iterator<int>& i = graph.Successors(x); i; i++)
      if (queue.Has(*i) && graph.Weight(x, *i) < -queue.PriorityOf(*i))
      {
        predecessor[*i] = x;
        queue.Raise(*i, -graph.Weight(x, *i));
      }
  }
}
```

Fila de prioridade para os vértices que ainda não estão na árvore.

Todos os vértices vão para a fila, com prioridade mínima.

Sobe-se a prioridade do vértice de partida. Assim ele será o primeiro a sair.

Atenção: a prioridade é um número negativo: o simétrico do peso da aresta. Isto é assim porque o próximo elemento a sair da fila é o de maior prioridade. Ao usar os simétricos, o elemento mais prioritário é o corresponde à aresta de menor peso, como pretendemos.

Peso da árvore de cobertura

É calculado pela função MinimumWeight:

```
double MinimumSpanningTree_Prim::MinimumWeight() const
{
  double result = 0;
  for (int i = 0; i < graph.CountVertices(); i++)
    result += (predecessor[i] == -1 ? 0 : graph.Weight(predecessor[i], i));
  return result;
}
```

Pormenor sintáctico: os parêntesis à volta da expressão condicional são indispensáveis pois a afectação tem precedência em relação à expressão à expressão condicional. Sem os parêntesis, a expressão seria sintacticamente válida, mas o programa estaria errado.

Análise do algoritmo de Prim

O desempenho do algoritmo de Prim depende do da fila de espera. Se aumentar a prioridade de um vértice envolver procurá-lo linearmente na fila e depois movê-lo linearmente para a sua nova posição, o algoritmo é E^*V^2 (V é o número de vértices, E é o número de arestas) no pior caso, pois o corpo da instrução if é executado no máximo para cada uma das arestas do grafo.

Se usarmos uma fila baseada num monte, então aumentar a prioridade é Log V, pelo que, procurando o vértice linearmente, o algoritmo fica E^*V^*Log V.

Com um "pequeno" esforço suplementar (que envolveria especializar a nossas classes Heap<T> e PriorityQueue<T>) podemos encontrar o vértice em tempo constante, tornando o algoritmo E^*Log V.

Exercício. Faça esse pequeno esforço...

Algoritmo de Kruskal

O algoritmo de Kruskal constrói a árvore de cobertura ganaciosamente, acrescentando em cada passo a aresta menos pesada que não forma um ciclo.

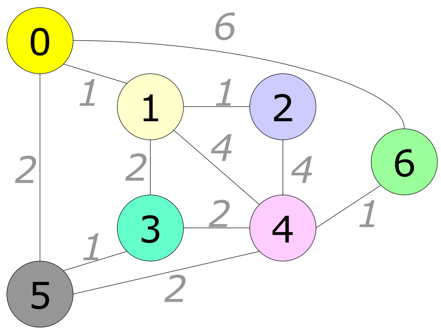
Em vez de usar apenas dois conjuntos de vértices – os que já estão na árvore e os que ainda não estão – como o algoritmo de Prim, usa muitos conjuntos disjuntos de vértices. Cada nova aresta seleccionada liga dois desses conjuntos, até aqui disjuntos, mas que agora é temos de reunir.

Precisamos de ordenar as arestas por peso. Usaremos um vector de pares <aresta, peso>, isto é, `VectorPolymorphic-Sortable<Pair<Pair<int, int>, double> >`.

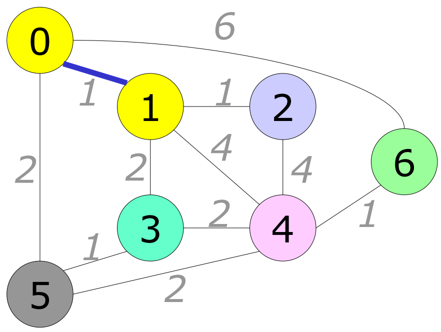
Para os conjuntos disjuntos, desenvolveremos uma nova classe, `DisjointSets`.

Filme de Kruskal

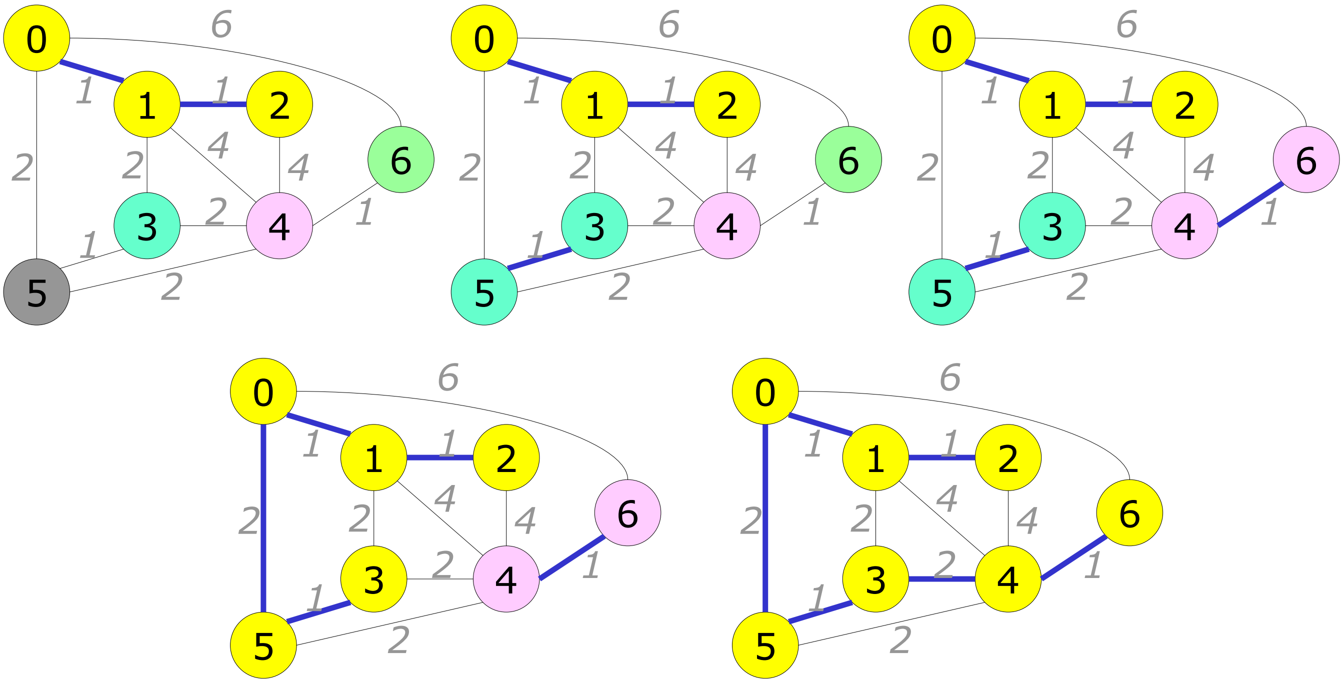
Inicialmente, cada vértice pertence a um dos conjuntos disjuntos:



Escolhe-se a aresta menos pesada entre dois conjuntos disjuntos e reúnem-se esses conjuntos:



Filme de Kruskal (continuação)



Conjuntos disjuntos

Queremos uma classe para representar um conjunto de conjuntos disjuntos no intervalo [0..*N*-1], para um dado *N*.

Inicialmente (isto é, na construção), haverá *N* conjuntos unitários.

Queremos uma operação para saber se dois elementos pertencem ao mesmo conjunto: Find.

Queremos uma operação para reunir dois conjuntos, identificados cada um por um dos seus elementos: Union.

```
class DisjointSets {
private:
  // ...
public:
  DisjointSets(int capacity);
  ~DisjointSets();

  virtual void Union(int x, int y);
  virtual bool Find(int x, int y) const;
};
```

> Estes nomes – Union e Find – são consagrados. Fala-se frequentemente nos algoritmos Union-Find.

Conjuntos disjuntos, implementação

Usaremos um vector de inteiros, items. Se items[x] valer y, isso significa que x pertence ao mesmo conjunto que y.

Inicialmente, todos os elementos do vector valem –1. Logo, nessa altura, nenhum elemento pertence ao mesmo conjunto que outro.

Os elementos de um conjunto formarão uma árvore de predecessores, de tal forma que cada conjunto pode ser representado internamente pela raiz dessa árvore.

Logo, para ver se dois elementos pertencem ao mesmo conjunto basta ver se os seus representantes são iguais.

E para reunir dois conjuntos, basta fazer o representante de um deles "apontar" (no vector items) para o representante do outro.

Classe DisjointSets

Juntamos às funções Union e Find algumas operações administrativas e para *debug*:

```
class DisjointSets {
private:
  Vector<int> items;
public:
  DisjointSets(int capacity);
  ~DisjointSets();

  virtual int Capacity() const;
  virtual int Count() const;

  virtual bool ValidElement(int x) const;

  virtual void Union(int x, int y);  // pre ValidElement(x) && ValidElement(y);
  virtual bool Find(int x, int y) const;   // pre ValidElement(x) && ValidElement(y);
                                           // returns true if x and y are in the same set.
private:
  virtual int Representative(int x) const;
public:  // for debugging
  virtual void WriteLine() const;
  virtual void WriteItems() const;
};
```

Classe DisjointSets, impl... (1)

Construtor:

```
DisjointSets::DisjointSets(int capacity):
  items(capacity)
{
  items.Fill(-1);
}
```

Destrutor:

```
DisjointSets::~DisjointSets()
{
}
```

Capacidade:

```
int DisjointSets::Capacity() const
{
  return items.Capacity();
}
```

Tamanho:

```
int DisjointSets::Count() const
{
  return items.CountIf(-1);
}
```

Repare: o número de conjuntos disjuntos é igual ao número de ocorrências do valor –1 no vector.

Elemento válido:

```
bool DisjointSets::ValidElement(int x) const
{
  return 0 <= x && x < Capacity();
}
```


Classe DisjointSets, impl... (2)

Para encontrar o representante, sobe-se na árvore de predecessores até à raiz:

```
int DisjointSets::Representative(int x) const
{
  int result = x;
  while (items[result] != -1)
    result = items[result];
  return result;
}
```

Dois elementos estarão no mesmo conjunto se tiverem o mesmo representante:

```
bool DisjointSets::Find(int x, int y) const
{
  return Representative(x) == Representative(y);
}
```

Para reunir dois conjuntos, fazemos o representante de um deles apontar para o representante do outro:

```
void DisjointSets::Union(int x, int y)
{
  int xr = Representative(x);
  int yr = Representative(y);
  if (xr != yr)
    items[yr] = xr;
}
```

Classe DisjointSets, **impl... (3)**

Eis as duas funções Write… que são usadas para *debug*. A primeira escreve cada conjunto entre chavetas:

```
void DisjointSets::WriteLine() const
{
  for (int i = 0; i < items.Capacity(); i++)
  {
    int count = 0;
    for (int j = 0; j < items.Capacity(); j++)
      if (Representative(j) == i)
        std::cout << (count++ == 0 ? "{" : " ") << j;
    if (count > 0)
      std::cout << "}";
  }
  std::cout << std::endl;
}
```

É uma função com desempenho quadrático (faz dois ciclos imbricados), mas toleramos isso, pois apenas a usamos para *debug*. Mesmo assim, observe a programação, que é interessante.

A segunda recorre à função da classe Vector<int>:

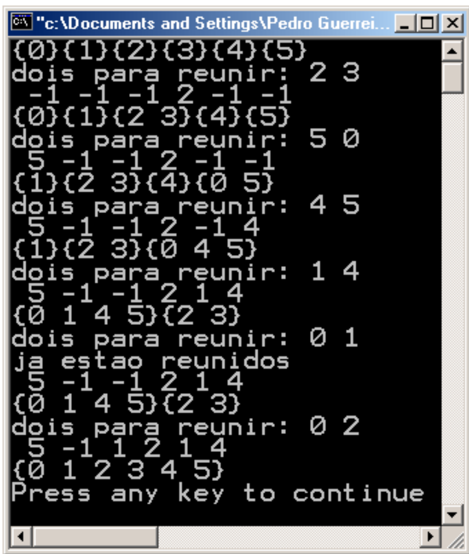
```
void DisjointSets::WriteItems() const
{
  items.WriteLine();
}
```


Testando os conjuntos disjuntos

Eis a função de teste:

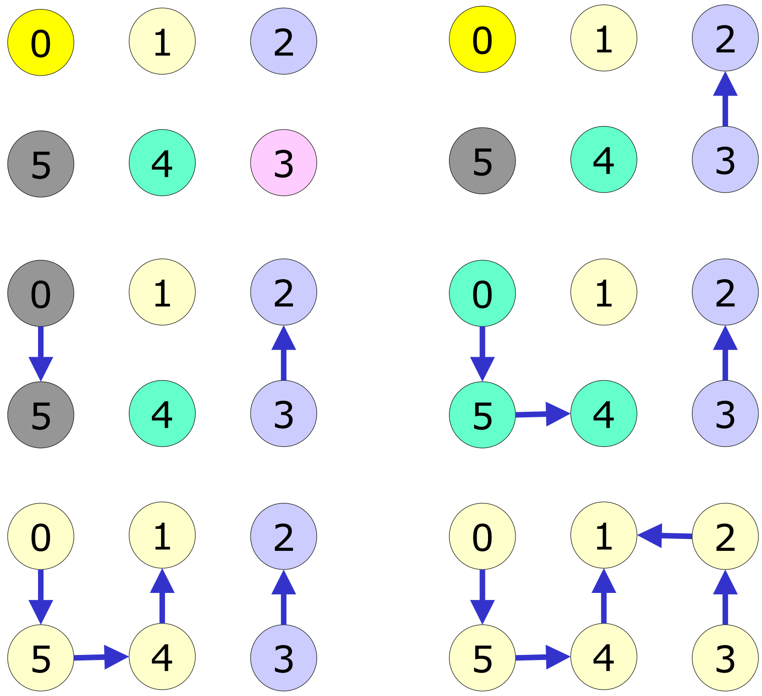
```
void TestDisjointSets1()
{
  Vector<int>::SetPrefixSuffix(" ", "");
  DisjointSets ds(6);
  ds.WriteLine();
  while (ds.Count() > 1)
  {
    std::cout << "dois para reunir: ";
    int x;
    int y;
    std::cin >> x >> y;
    if (ds.Find(x, y))
      std::cout << "ja estao reunidos" << std::endl;
    else
      ds.Union(x, y);
    ds.WriteItems();
    ds.WriteLine();
  }
}
```

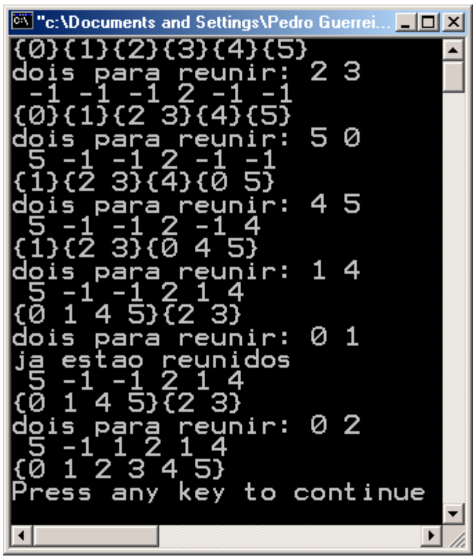
Neste teste entramos sucessivamente os pares (2, 3), (5, 0), (4, 5), (1, 4), (0, 1) e (0, 2).



Filme do teste

Eis o filme das operações:





Duas heurísticas

Pode acontecer que ao longo do funcionamento se formem percursos muito compridos (como ilustra o caso <0, 5, 4, 1> no teste). Era bom evitar isso.

Usaremos duas heurísticas. Primeiro, sempre que calcularmos o representante de um vértice, ligaremos esse vértice, e já agora, todos os seus predecessores, directamente ao representante. Assim, encurtamos o caminho dos vértices até ao representante. Isto é a técnica da compressão do caminho (em inglês, *path compression*).

Segundo, ao unir dois conjuntos ligaremos o representante do conjunto com menos elementos ao do conjunto com mais elementos (e nunca o contrário). Assim, as árvores ficarão menos profundas. Isto é a técnica do equilibrismo dos pesos (em inglês, *weight balancing*).

Equilibrismo, *s. m.* Acto de equilibrar.

Conjuntos disjuntos, com heurísticas

Substituímos a anterior classe DisjointSets por uma nova, que implementa as heurísticas.

Guardamos o número de elementos de cada conjunto no vector, na posição do representante, convencionalmente pelo simétrico desse número. (Até calha bem, porque na inicialização ficam todos a valer –1, que é precisamente o que convém).

Além disso, mudamos a semântica da operação Union. Normalmente, antes de fazer uma reunião, testamos se os dois elementos pertencem aos mesmo conjunto com a função Find. Internamente, isso implica calcular os representantes. Depois, na função Union, os representantes são calculados de novo, o que é um desperdício. Usaremos então a técnica de fazer a reunião logo, sem testar, perguntando depois, com uma função booleana United, se os dois conjuntos foram mesmo reunidos, o que terá acontecido se e só se eles eram disjuntos.

Nova classe DisjointSets

Eis a nova classe, que substitui a anterior:

```
class DisjointSets {
private:
  Vector<int> items;
  bool united;
public:
  DisjointSets(int capacity);
  ~DisjointSets();

  virtual int Capacity() const;
  virtual int Count() const;

  virtual bool ValidElement(int x) const;

  virtual void Union(int x, int y); // pre ValidElement(x) && ValidElement(y);
  virtual bool Find(int x, int y) const; // pre ValidElement(x) && ValidElement(y);
                                         // returns true if x and y are in the same set.
  virtual bool United() const;
private:
  virtual int RepresentativeBasic(int x) const;
  virtual int Representative(int x) const;
  virtual void PathCompress(int x, int r); // compress to root r
public:  // for debugging
  virtual void WriteLine() const;
  virtual void WriteItems() const;
};
```

A função RepresentativeBasic corresponde à função Representative da classe anterior.

Nova classe DisjointSets, **impl...**

As funções "de apoio" são parecidas ou iguais:

```
DisjointSets::DisjointSets(int capacity):
  items(capacity),
  united(false)
{
  items.Fill(-1);
}
```

```
DisjointSets::~DisjointSets()
{
}
```

```
int DisjointSets::Capacity() const
{
  return items.Capacity();
}
```

Esta é diferente:

```
int DisjointSets::Count() const
{
  return items.CountIfGeneral(LessThan<int>(-1));
}
```

Repare na utilização da função geral CountIfGeneral, da classe VectorAbstract<T>.

Esta é nova:

```
bool DisjointSets::ValidElement(int x) const
{
  return 0 <= x && x < Capacity();
}
```

```
bool DisjointSets::United() const
{
  return united;
}
```

```
void DisjointSets::WriteLine() const
{
  // ...
}
```

```
void DisjointSets::WriteItems() const
{
  items.WriteLine();
}
```


Predicado LessThan<T>

Para referência, eis o predicado LessThan<T>, usado na função DisjointSets::Count:

```
template <class T>
class LessThan: public Predicate<T>{
private:
  T x;
public:
  LessThan(const T& x);
  virtual ~LessThan();
  virtual bool Good(const T& x) const;
  virtual const Predicate<T>* Clone() const;
};
```

Implementação:

```
template <class T>
LessThan<T>::LessThan(const T& x):
  x(x)
{
}
```

```
template <class T>
LessThan<T>::~LessThan()
{
}
```

```
template <class T>
const Predicate<T>* LessThan<T>::Clone() const
{
  return new LessThan(*this);
}
```

```
template <class T>
bool LessThan<T>::Good(const T& x) const
{
  return x <= this->x;
}
```

Também há os predicados GreaterThan<T> e Equality<T> análogos a este.

Compressão do caminho

A função PathCompress(int x, int r) sobe de x até r, ligando a r todos os elementos visitados:

```
void DisjointSets::PathCompress(int x, int r)
{
  if (x == r)
    return;
  int i = x;
  while (items[i] != r)
  {
    int temp = i;
    i = items[i];
    items[temp] = r;
  }
}
```

A função Representative-Basic é como antes:

```
int DisjointSets::RepresentativeBasic(int x) const
{
  int result = x;
  while (items[result] >= 0)
    result = items[result];
  return result;
}
```

A compressão é feita na nova função Representative:

```
int DisjointSets::Representative(int x) const
{
  int representative = RepresentativeBasic(x);
  const_cast<DisjointSets*>(this)->PathCompress(x, representative);
  return representative;
}
```

Repare no const_cast do objecto da função, necessário porque a função é const mas a função PathCompress não é.

Equilibrismo dos pesos

Os pesos são equilibrados na função Union, a quem agora também compete posicionar a variável booleana united:

```
void DisjointSets::Union(int x, int y)
{
  int xr = Representative(x);
  int yr = Representative(y);
  if (united = (xr != yr))
    if (items[yr] < items[xr]) // subtree yr is smaller
    {
      items[yr] += items[xr];
      items[xr] = yr;
    }
    else
    {
      items[xr] += items[yr];
      items[yr] = xr;
    }
}
```

Agora, sempre que chamamos a função Representative, comprimimos o caminho.

Repare na técnica, pouco ortodoxa, de afectar a variável booleana na condição do if. 😐

A função Find é programada como antes, mas é menos usada, porque fazemos o serviço com as função Union e United.

```
bool DisjointSets::Find(int x, int y) const
{
  return Representative(x) == Representative(y);
}
```

Testando os novos conjuntos disjuntos

Eis a nova função de teste:

```
void TestDisjointSets2()
{
  Vector<int>::SetPrefixSuffix(" ", "");
  DisjointSets ds(6);
  ds.WriteLine();
  while (ds.Count() > 1)
  {
    std::cout << "dois para reunir: ";
    int x;
    int y;
    std::cin >> x >> y;
    ds.Union(x, y);
    if (!ds.United())
      std::cout << "ja estao reunidos" << std::endl;
    ds.WriteItems();
    ds.WriteLine();
  }
}
```

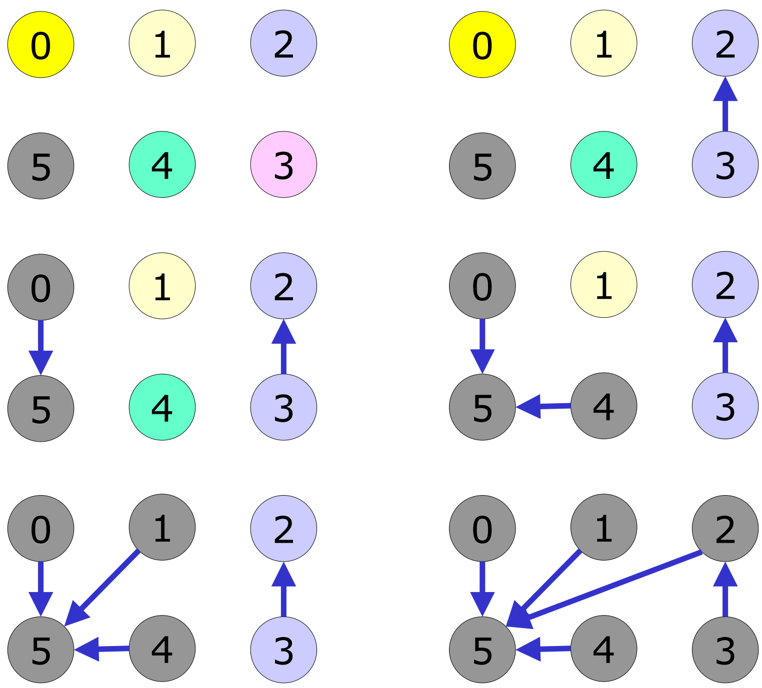
Repara na utilização conjunta das funções Union, para reunir, e United, para verificar se houve mesmo reunião.

Usamos a mesma sequência da dados de anteriormente, para comparar.

```
"c:\Documents and Settings\Pedro Guerrei...
{0}{1}{2}{3}{4}{5}
dois para reunir: 2 3
 -1 -1 -2 2 -1 -1
{0}{1}{2 3}{4}{5}
dois para reunir: 5 0
 5 -1 -2 2 -1 -2
{1}{2 3}{4}{0 5}
dois para reunir: 4 5
 5 -1 -2 2 5 -3
{1}{2 3}{0 4 5}
dois para reunir: 1 4
 5 5 -2 2 5 -4
{2 3}{0 1 4 5}
dois para reunir: 0 1
ja estao reunidos
 5 5 -2 2 5 -4
{2 3}{0 1 4 5}
dois para reunir: 0 2
 5 5 5 2 5 -6
{0 1 2 3 4 5}
Press any key to continue
```


Novo filme do teste

Eis o filme das operações com compressão e equilibrismo:



```
"c:\Documents and Settings\Pedro Guerrei...
{0}{1}{2}{3}{4}{5}
dois para reunir: 2 3
 -1 -1 -2 2 -1 -1
{0}{1}{2 3}{4}{5}
dois para reunir: 5 0
 5 -1 -2 2 -1 -2
{1}{2 3}{4}{0 5}
dois para reunir: 4 5
 5 -1 -2 2 5 -3
{1}{2 3}{0 4 5}
dois para reunir: 1 4
 5 5 -2 2 5 -4
{2 3}{0 1 4 5}
dois para reunir: 0 1
ja estao reunidos
 5 5 -2 2 5 -4
{2 3}{0 1 4 5}
dois para reunir: 0 2
 5 5 5 2 5 -6
{0 1 2 3 4 5}
Press any key to continue
```

Esta árvore final é menos profunda do que a da implementação original.

Classe MinimumSpanningTree_Kruskal

É muito parecida com a classe MinimumSpanningTree_Prim:

```
class MinimumSpanningTree_Kruskal {
private:
  const GraphWeighted& graph;
  Vector<Pair<int, int> > tree;
  VectorPolymorphicSortable<Pair<Pair<int, int>, double> > edges;
public:
  MinimumSpanningTree_Kruskal(const GraphWeighted& graph);
  virtual ~MinimumSpanningTree_Kruskal();

  virtual void Compute();  // post Computed();
  virtual bool Computed() const;

  virtual const Vector<Pair<int, int> >& Tree() const;
  virtual double MinimumWeight() const;  // pre Computed();

public: // for debugging
  virtual const VectorAbstract<Pair<Pair<int, int>, double> >& Edges() const;
};
```

Esta é o vector de arestas que representa a árvore de cobertura calculada.

Este é o vector das arestas com peso, para ordenação.

O algoritmo de Prim tinha um ponto de partida para os cálculos, este não.

Função para devolver o resultado.

Construção Kruskal

Na construção inicializa-se logo o vector da arestas pesadas e ordena-se:

```
MinimumSpanningTree_Kruskal::MinimumSpanningTree_Kruskal(const GraphWeighted& graph):
  graph(graph),
  tree(graph.CountVertices() - 1),
  edges(graph.CountEdges())
{
  for (int i = 0; i < graph.CountVertices(); i++)
    for (Iterator<int>& j = graph.Successors(i); j; j++)
      edges.Put(Pair<Pair<int, int>, double>(Pair<int, int>(i, *j), graph.Weight(i, *j)));
   edges.SortGeneral(OrderAdapted<Pair<Pair<int, int>, double> >
                                   (Pair<Pair<int, int>, double>::LessThanBySecond));
}
```

```
MinimumSpanningTree_Kruskal::~MinimumSpanningTree_Kruskal()
{
}
```

Nada a fazer no destrutor.

Ordenamos usando a ordem adaptada gerada pela função de comparação LessThanBySecond na classe dos elementos do vector, Pair<Pair<int, int>, double>.

Função do algoritmo de Kruskal

É simplicíssima e claríssima:

```
void MinimumSpanningTree_Kruskal::Compute()
{
  tree.Clear();
  DisjointSets sets(graph.CountVertices());
  for (int i = 0; i < edges.Count(); i++)
  {
    int v1 = edges[i].First().First();
    int v2 = edges[i].First().Second();
    sets.Union(v1, v2);
    if (sets.United())
      tree.Put(Pair<int, int>(v1, v2));
  }
}
```


Pseudo-código

No livro *Introduction to Algorithms*, o algoritmo de Kruskal é descrito assim, em pseudo-código:

```
MST-Kruskal(G, w)
    A ← ∅
    for each vertex v ∈ V[G]
        do Make-Set(v)
    sort the edges of E into nondecreasing order by weight w
    for each edge (u, v) ∈ E, taken in nondecreasing order by weight
        do if Find-Set(u) ≠ Find-Set(v)
              then A ← A ∪ {(u, v)}
                   Union(u, v)
    return A
```

No nosso caso, a ordenação é feita na construção.

As funções Make-Set, Find-Set e Union são as dos conjuntos disjuntos. O conjunto dos conjuntos disjuntos não é nomeado explicitamente neste pseudo-código.

Restantes funções da classe

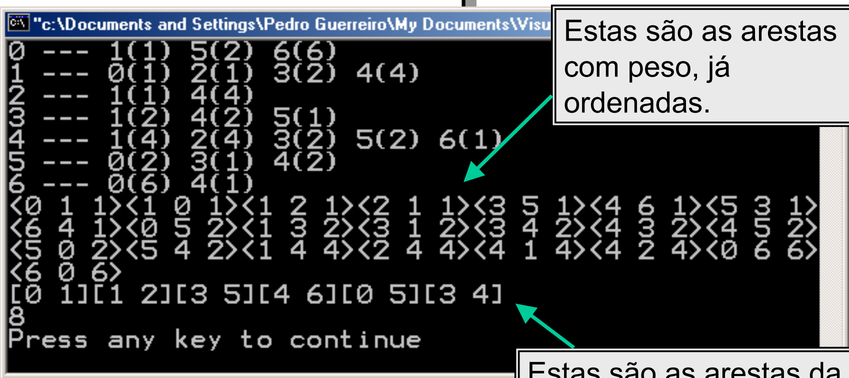
Também são simples. Ei-las para referência:

```
bool MinimumSpanningTree_Kruskal::Computed() const
{
  return tree.Count() > 0;
}

const Vector<Pair<int, int> >& MinimumSpanningTree_Kruskal::Tree() const
{
  return tree;
}

double MinimumSpanningTree_Kruskal::MinimumWeight() const
{
  double result = 0;
  for (int i = 0; i < tree.Count(); i++)
    result += graph.Weight(tree[i].First(), tree[i].Second());
  return result;
}

const VectorAbstract<Pair<Pair<int, int>, double> >&
MinimumSpanningTree_Kruskal::Edges() const
{
  return edges;
}
```


Função de teste, algoritmo de Kruskal

Eis a função de teste que corresponde ao exemplo usado na

```
void TestMinimumSpanningTree_Kruskal()
{
  Vector<Pair<int, int> >::SetPrefixSuffix("[", "]");
  Vector<Pair<Pair<int, int>, double> >::SetPrefixSuffix("<", ">");
  GraphWeightedUsingList g(7);
  g.LinkUndirected(0, 1, 1);
  g.LinkUndirected(0, 5, 2);
  g.LinkUndirected(0, 6, 6);
  g.LinkUndirected(1, 2, 1);
  g.LinkUndirected(1, 3, 2);
  g.LinkUndirected(1, 4, 4);
  g.LinkUndirected(2, 4, 4);
  g.LinkUndirected(3, 4, 2);
  g.LinkUndirected(3, 5, 1);
  g.LinkUndirected(4, 5, 2);
  g.LinkUndirected(4, 6, 1);

  g.Write();
  MinimumSpanningTree_Kruskal mst(g);
  mst.Edges().WriteLine();
  mst.Compute();

  mst.Tree().WriteLine();
  std::cout << mst.MinimumWeight() << std::endl;
}
```



Estas são as arestas com peso, já ordenadas.

Estas são as arestas da árvore de cobertura, pela ordem por que foram calculadas.

Análise do algoritmo de Kruskal

A eficiência do algoritmo de Kruskal depende da implementação dos conjuntos disjuntos. Admitindo que as técnicas de compressão e equilibrismo mantêm as árvores razoavelmente equilibradas (o que poderíamos comprovar com uma análise mais refinada) a operação de reunião é logarítmica no número de vértices. Ora faz-se uma operação de reunião para cada aresta. Logo, representando por *V* o número de vértices e por *E* o número de arestas, podemos concluir que o algoritmo de Kruskal é *E**Log *V* (tal como o algoritmo de Prim, aliás).

Estamos a excluir desta análise o tempo de ordenação das arestas, que, usando o quicksort, será *E**Log *E*.

Caminho mais curto

Trata-se de encontrar um caminho de peso mínimo entre dois vértices num grafo orientado pesado. Pode haver vários.

O peso de um caminho é a soma dos pesos das arestas que o compõem.

Pode haver arestas com peso negativo desde que não haja ciclos com peso negativo.

Os caminhos mais curtos têm subestrutura optimal:

> Se $<v_1, v_2, \ldots v_N>$ for um caminho mais curto entre v_1 e v_N, então para todo o *i* e *j* entre 1 e *N* tal que $i <= j$, o caminho $<v_i, v_{i+1}, \ldots, v_{j-1}, v_j>$ é um caminho mais curto entre v_i e v_j.

Os caminhos mais curtos não têm ciclos. Podem ser representados num vector de predecessores.

Origem única

Interessa-nos o problema quando é dado um único vértice de partida: "caminho mais curto com origem única".

Começamos com uma classe de base abstracta que reúne os membros de dados para o vector de predecessores e ainda para as distâncias já calculadas:

```
class ShortestPathSingleSource
{
private:
  const GraphWeighted& graph;
  Vector<int> predecessor;
  Vector<double> distance;
  int start;
public:
  // ...
};
```

Na verdade, o vector predecessor guardará todos os caminhos mínimos a partir do vértice start e o vector distance guardará a distância deste o vértice start até cada um dos outros, segundo o respectivo caminho mínimo.

Classe ShortestPathSingleSource

O único modificador público é a função Compute, que é uma função abstracta, pois a sua definição fica ao cuidado das classes derivadas;

```
class ShortestPathSingleSource {
private:
  const GraphWeighted& graph;
  Vector<int> predecessor;
  Vector<double> distance;
  int start;
public:
  ShortestPathSingleSource(const GraphWeighted& graph);
  virtual ~ShortestPathSingleSource();

  virtual void Compute(int start) = 0; post Computed();
  virtual bool Computed() const;

  virtual const GraphWeighted& GetGraph() const;
  virtual const Vector<int>& Predecessors() const; // pre Computed();
  virtual const Vector<double>& Distances() const; // pre Computed();
  virtual int Start() const; // pre Computed();
  virtual double DistanceTo(int x) const; // pre Computed();
  virtual const ListDouble<int> Path(int finish) const;  // pre   Computed();
};
```

Se a classe tem membros privados e um único modificador virtual puro, terá de fornecer alguns modificadores protegidos em benefício das classes derivadas, claro.

Selectores para o grafo, para os predecessores, para as distâncias, para o ponto de partida, para a distância de um vértice e para o caminho calculado.

Construtor, etc., para o caminho

Tudo como se esperaria:

```
ShortestPathSingleSource::ShortestPathSingleSource(const GraphWeighted& graph):
  graph(graph),
  predecessor(graph.CountVertices()),
  distance(graph.CountVertices()),
  start(-1)
{
}
```

```
ShortestPathSingleSource::~ShortestPathSingleSource()
{
}
```

```
bool ShortestPathSingleSource::Computed() const
{
  return start != -1;
}
```

```
const Vector<int>& ShortestPathSingleSource::Predecessors() const
{
  return predecessor;
}
```

```
const Vector<double>& ShortestPathSingleSource::Distances() const
{
  return distance;
}
```

```
const GraphWeighted& ShortestPathSingleSource::GetGraph() const
{
  return graph;
}
```

```
int ShortestPathSingleSource::Start() const
{
  return start;
}
```


A lista com o caminho

Calcula-se a partir do vector dos predecessores:

```
const ListDouble<int> ShortestPathSingleSource::Path(int finish) const
{
  ListDouble<int> result;
  int x = finish;
  for(;;)
  {
    result.PutFirst(x);
    if (x == start)
      break;
    x = predecessor[x];
    if (x == -1)
    {
      result.Clear();
      break;
    }
  }
  return result;
}
```

Calcula e devolve uma lista com o caminho desde o vértice start (que está registado num membro de dados) e o vértice finish, que vem no argumento. Se não houver caminho, porque o vértice finish não é acessível a partir do vértice start, devolve a lista vazia.

Inicialização

O vector distance contém em cada momento uma estimativa por excesso da distância do vértice respectivo (em relação ao vértice de partida. O vector predecessor mantém os predecessores já calculados. Estas duas estruturas são inicializadas na função InitializeComputation:

```
class ShortestPathSingleSource
{
//...
protected:
  virtual void InitializeComputation(int start);
};
```

```
void ShortestPathSingleSource::InitializeComputation(int start)
{
  this->start = start;
  predecessor.Fill(-1);
  distance.Fill(std::numeric_limits<double>::infinity());
  distance[start] = 0.0;
}
```

Colocamos –1 em todas as posições do vector predecessors, pois de início nenhuns predecessores estão calculados.

A estimativa inicial da distância é $+\infty$ para todos os vértices, excepto para o vértice de partida.

Relaxação

Relaxar é melhorar a estimativa e acertar o predecessor, quando de obtém informação que permita fazer isso:

```
class ShortestPathSingleSource
{
//...
protected:
  virtual void InitializeComputation(int start);
  virtual void Relax(int x, int y);
};
```

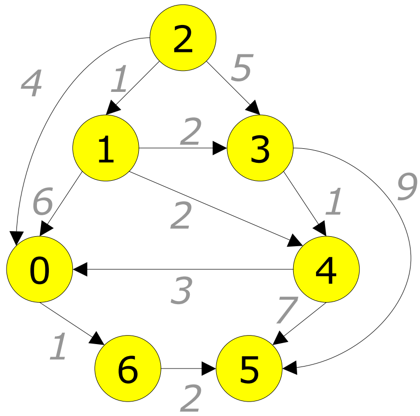
```
void ShortestPathSingleSource::Relax(int x, int y)
{
  if (distance[y] > distance[x] + graph.Weight(x, y))
  {
    distance[y] = distance[x] + graph.Weight(x, y);
    predecessor[y] = x;
  }
}
```

Isto é: se verificarmos que a estimativa da distância até *y* é maior do que a soma da estimativa da distância até *x* com o peso da aresta (*x*, *y*), então melhoramos a estimativa com o valor dessa soma, registando que no caminho "actual" se chega a *y* vindo de *x*.

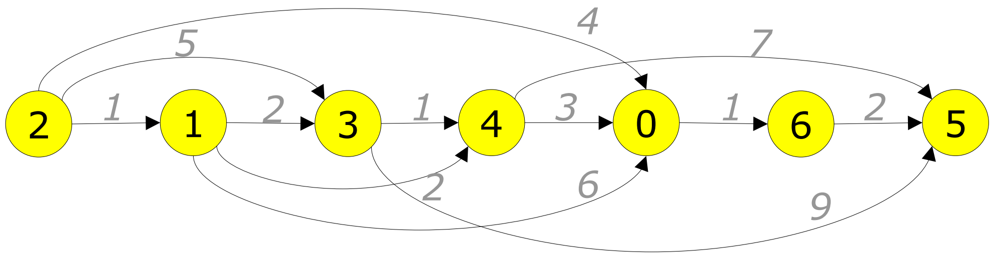
Em grafos orientados acíclicos

Para achar o caminho mais curto num grafo orientado acíclico basta relaxar por ordem topológica.

Exemplo: qual é o caminho mais curto entre os vértices 1 e 5 no seguinte grafo?



Ordenando topologicamente, dá, em esquema:



ShortestPathDirectedAcyclicGraph

Deriva de ShortestPathSingleSource<T> e tem um membro privado para a busca profunda que calcula a ordem topológica:

```
class ShortestPathDirectedAcyclicGraph: public ShortestPathSingleSource {
private:
  DepthFirstSearch dfs;
public:
  ShortestPathDirectedAcyclicGraph(const GraphWeighted& graph);
  virtual ~ShortestPathDirectedAcyclicGraph();

  virtual void Compute(int start);
  virtual void Compute();
  virtual const ListDouble<int>& Topological() const;
};
```

A função Compute sem argumentos calcula a partir do primeiro vértice da ordem topológica.

```
ShortestPathDirectedAcyclicGraph::ShortestPathDirectedAcyclicGraph(const GraphWeighted& graph):
  ShortestPathSingleSource(graph),
  dfs(graph)
{
  dfs.Compute();
}
```

```
ShortestPathDirectedAcyclicGraph::~ShortestPathDirectedAcyclicGraph()
{
}
```

A busca profunda faz-se logo na construção.

Calculando o caminho mais curto

Primeiro procura-se o vértice de partida na ordem; depois relaxa-se por ordem topológica:

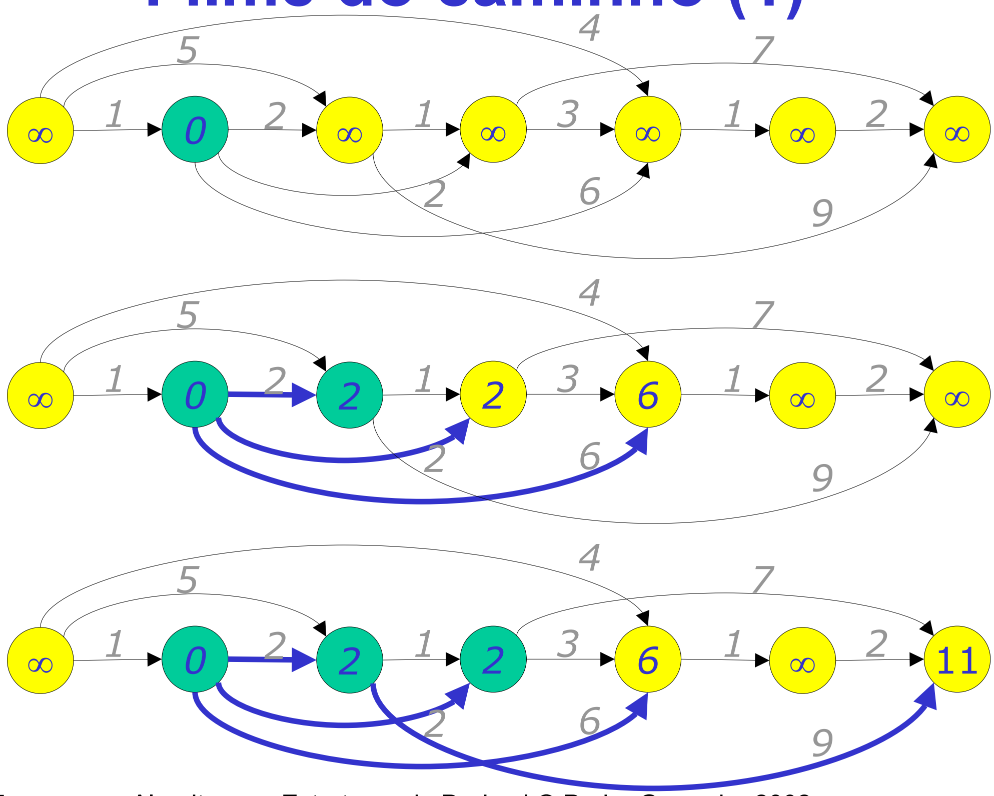
```
void ShortestPathDirectedAcyclicGraph::Compute(int start)
{
  InitializeComputation(start);
  Iterator<int>& i = dfs.Topological().Items();
  while (*i != start)
    i++;
  for (; i; i++)
    for (Iterator<int>& j = GetGraph().Successors(*i); j; j++)
      Relax(*i, *j);
}
```

Note que este esquema funciona com pesos positivos ou negativos.

```
void ShortestPathDirectedAcyclicGraph::Compute()
{
  Compute(Topological().First());
}
```

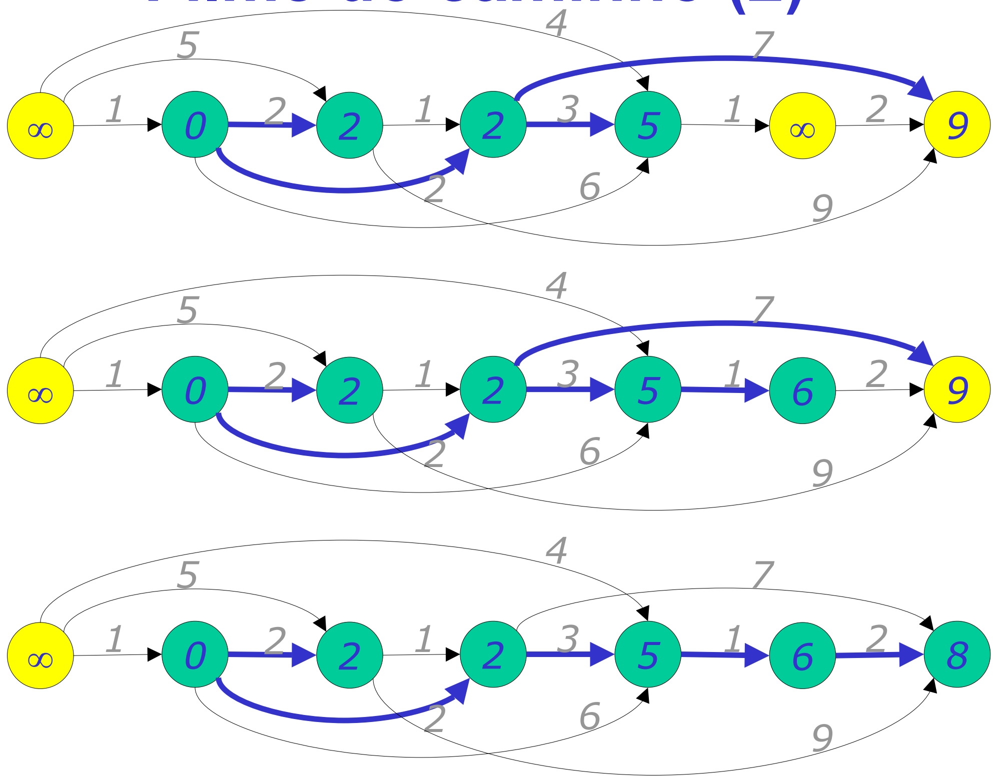

Filme do caminho (1)

Os vértices são, da esquerda para a direita: 2 1 3 4 0 6 5.
O número que aparece na bola é a distância estimada.



Filme do caminho (2)

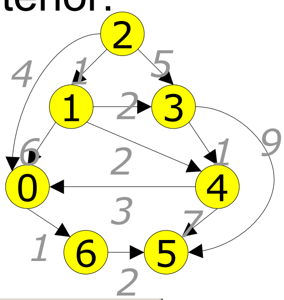
Os vértices são, da esquerda para a direita: 2 1 3 4 0 6 5.
O número que aparece na bola é a distância estimada.



Função de teste

Eis a função de teste que corresponde ao filme anterior:

```
void TestShortestPathSingleDirectedAcyclicGraph()
{
  Vector<int>::SetPrefixSuffix(" ", "");
  Vector<double>::SetPrefixSuffix(" ", "");
  ListDouble<int>::SetPrefixSuffix(" ", "");
  GraphWeightedUsingList g(7);
  g.Link(0, 6, 1);
  g.Link(1, 0, 6);
  g.Link(1, 3, 2);
  g.Link(1, 4, 2);
  g.Link(2, 0, 4);
  g.Link(2, 1, 1);
  g.Link(2, 3, 5);
  g.Link(3, 4, 1);
  g.Link(3, 5, 9);
  g.Link(4, 0, 3);
  g.Link(4, 5, 7);
  g.Link(6, 5, 2);
  g.Write();
  ShortestPathDirectedAcyclicGraph spdag(g);
  spdag.Topological().WriteLine();
  spdag.Compute(1);
  spdag.Path(5).WriteLine();
  spdag.Predecessors().WriteLine();
  spdag.Distances().WriteLine();
}
```

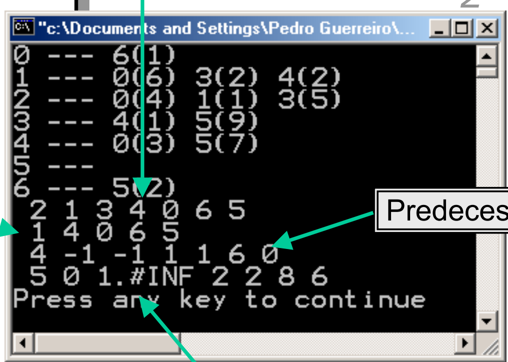


Ordem topológica.

Caminho.

Predecessores.

Distâncias



Análise deste caminho mais curto

O tempo de cálculo da ordem topológica é o tempo da busca profunda, que é proporcional a *V*+*E* (*V* é o número de vértices, *E* é o número de arestas). O tempo de inicialização é proporcional a *V*, claro. O ciclo while e o ciclo for em conjunto, na pior das hipóteses visitam cada vértice e cada aresta. Portanto, o seu tempo de execução é proporcional a *V*+*E* igualmente.

Reunindo tudo, concluímos que o cálculo do caminho mais curto num grafo orientado acíclico é proporcional ao tamanho da representação do grafo usando listas.

O caminho crítico

Se usarmos um grafo orientado acíclico para representar um conjunto de tarefas relacionadas, de tal maneira que terminar uma aresta *e*1 num vértice e começar outra aresta *e*2 nesse vértice significa que a tarefa representada por *e*2 só pode começar depois da tarefa representada por *e*1 ter terminado, então o caminho mais longo no grafo é o chamado *caminho crítico*. O peso de uma aresta representa a duração da tarefa respectiva, bem entendido.

A duração do caminho crítico representa a duração planeada para a realização do conjunto de tarefas. Um atraso numa das tarefas do caminho crítico acarreta um atraso no projecto.

Como calcular o caminho mais longo num grafo orientado acíclico?

Negar os pesos e calcular o caminho mais curto!

Podemos fazer isto, porque o grafo não tem ciclos.

Preparando o caminho crítico

Temos de enriquecer a classe GraphWeighted com uma função para negar os pesos:

```
class GraphWeighted: public Graph {
public:
  // ...
  virtual void Negate();
};
```

```
void GraphWeighted::Negate()
{
  for (int i = 0; i < CountVertices(); i++)
    for (Iterator<int>& j = Successors(i); j; j++)
      SetWeight(i, *j, -Weight(i, *j));
}
```

As distâncias calculadas serão ficticiamente negativas, pelo que, no fim, devemos negá-las:

```
class ShortestPathSingleSource
{
// ...
protected:
  // ...
  virtual void NegateDistances();
};
```

```
void ShortestPathSingleSource::NegateDistances()
{
  for (int i = 0; i < distance.Count(); i++)
    if (distance[i] != std::numeric_limits<double>::infinity())
      distance[i] *= -1;
}
```

Só negamos as distâncias que não forem $+\infty$, claro.

Calculando o caminho crítico

Mais duas funções para a classe:

```
class ShortestPathDirectedAcyclicGraph: public ShortestPathSingleSource {
// ...
  virtual void ComputeCriticalPath(int start);
  virtual void ComputeCriticalPath();
};
```

A versão sem argumentos calcula a partir do primeiro vértice da ordem topológica.

Queremos negar os pesos no grafo, mas o grafo é const. Temos de fazer batota ☹, usando const_cast:

```
void ShortestPathDirectedAcyclicGraph::CriticalPath(int start)
{
  const_cast<GraphWeighted &>(GetGraph()).Negate();
  Compute(start);
  NegateDistances();
  const_cast<GraphWeighted &>(GetGraph()).Negate();
}
```

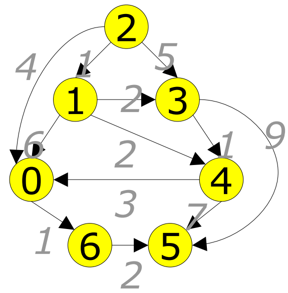
No fim negamos outra vez, para deixar as coisas como estavam...

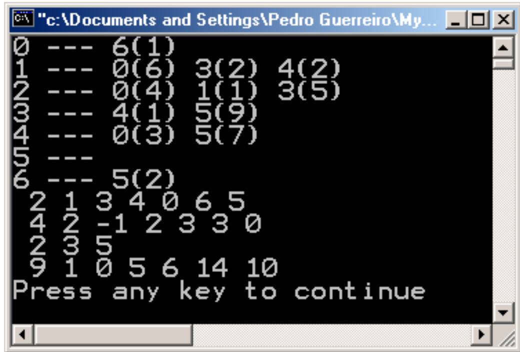
```
void ShortestPathDirectedAcyclicGraph::ComputeCriticalPath()
{
  ComputeCriticalPath(Topological().First());
}
```

Testando o caminho crítico

Qual o caminho crítico no grafo do exemplo, começando no primeiro vértice da ordem topológica?

```
void TestCriticalPath()
{
  Vector<int>::SetPrefixSuffix(" ", "");
  Vector<double>::SetPrefixSuffix(" ", "");
  ListDouble<int>::SetPrefixSuffix(" ", "");
  GraphWeightedUsingList g(7);
  g.Link(0, 6, 1);
  // …
  g.Link(6, 5, 2);
  g.Write();
  ShortestPathDirectedAcyclicGraph spdag(g);
  spdag.Topological().WriteLine();
  spdag.ComputeCriticalPath();
  spdag.Predecessors().WriteLine();
  spdag.Path(5).WriteLine();
  spdag.Distances().WriteLine();
}
```






```
0 --- 6(1)
1 --- 0(6) 3(2) 4(2)
2 --- 0(4) 1(1) 3(5)
3 --- 4(1) 5(9)
4 --- 0(3) 5(7)
5 ---
6 --- 5(2)
 2 1 3 4 0 6 5
 4 2 -1 2 3 3 0
 2 3 5
 9 1 0 5 6 14 10
Press any key to continue
```


O algoritmo de Dijkstra

Calcula o caminho mais curto num grafo orientado pesado onde todos os pesos são não-negativos.

A estratégia é usar uma fila com prioridades para os vértices para os quais o caminho mais curto ainda não foi calculado. Nessa fila, a prioridade é o simétrico da estimativa da distância ao vértice de partida.

Assim, em cada passo, o vértice mais prioritário, isto é, aquele que menos dista do vértice de partida, é seleccionado e os seus sucessores são relaxados (de maneira a actualizar as estimativas da distância e os predecessores provisórios).

O algoritmo de Dijkstra é um algoritmo ganancioso, que em cada passo faz a escolha mais favorável no momento. (Nem sempre uma escolha gananciosa é acertada, mas neste caso é.)

Classe ShortestPath_Dijkstra

Deriva de ShortestPathSingleSource:

```
class ShortestPath_Dijkstra: public ShortestPathSingleSource {
public:
  ShortestPath_Dijkstra(const GraphWeighted& graph);
  virtual ~ShortestPath_Dijkstra();

  virtual void Compute(int start);
};
```

Tudo muito simples.

Construtor, destrutor:

```
ShortestPath_Dijkstra::ShortestPath_Dijkstra(const GraphWeighted& graph):
  ShortestPathSingleSource(graph)
{
}
```

```
ShortestPath_Dijkstra::~ShortestPath_Dijkstra()
{
}
```


Calculando com o algoritmo de Dijkstra

```
void ShortestPath_Dijkstra::Compute(int start)
{
  InitializeComputation(start);
  PriorityQueue<int> queue(GetGraph().CountVertices());
  queue.PutItems(FromTo(0, GetGraph().CountVertices() - 1));
  queue.Raise(start, 0);
  while (!queue.Empty())
  {
    int x = queue.Item();
    queue.Remove();
    for (Iterator<int>& i = GetGraph().Successors(x); i; i++)
    {
      Relax(x, *i);
      queue.Raise(*i, -DistanceTo(*i));
    }
  }
}
```

Inicialização (função herdada).

Pomos todos os vértices na fila, com prioridade mínima, excepto o vértice de partida cuja prioridade é zero (pois a distância dele a ele próprio é zero).

Retiramos da fila o vértice mais prioritário...

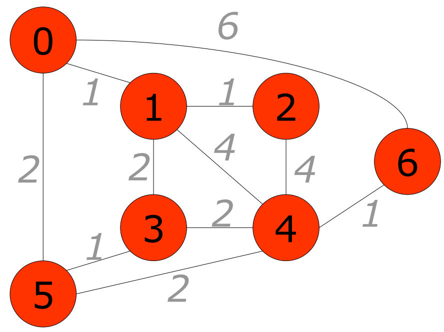
... e relaxamos os seus sucessores.

Como o mais prioritário é o menos distante, usamos o simétrico.

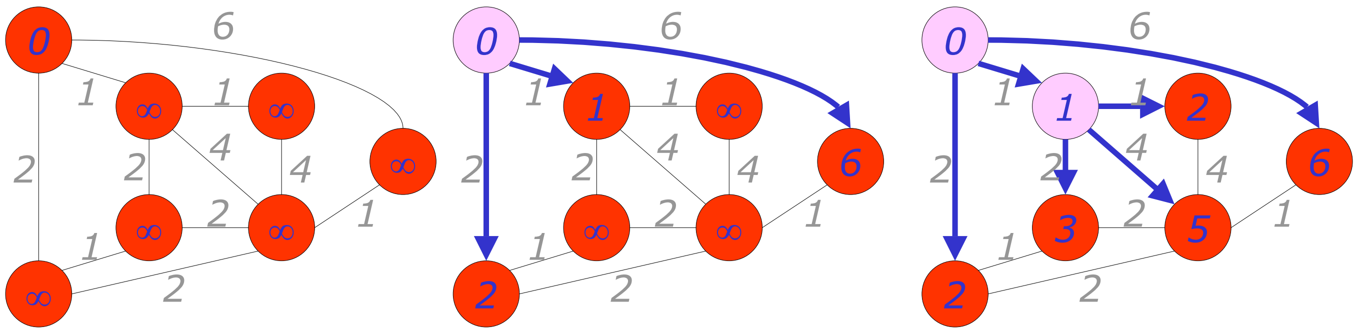
É parecido com o algoritmo de Prim, mas veja as diferenças!

Filme do algoritmo de Dijkstra

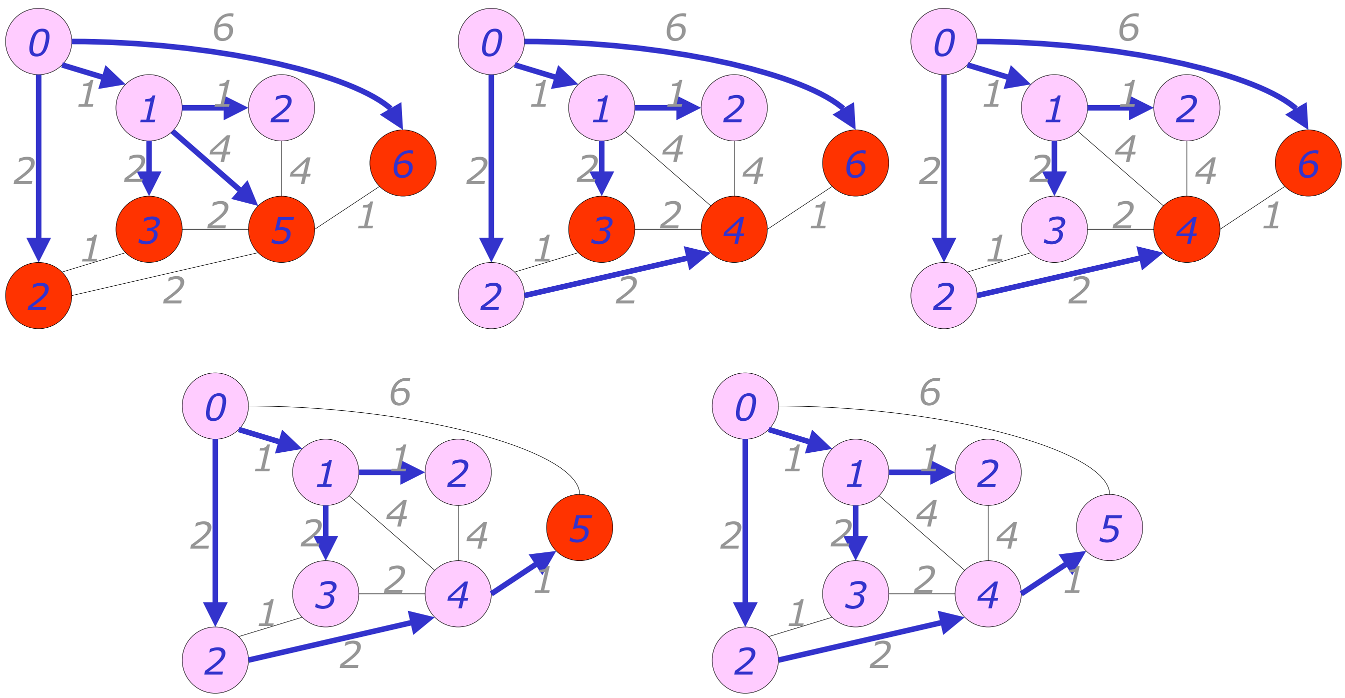
Vejamos a operação do algoritmo de Dijkstra para calcular a os caminhos mais curtos a partir do vértice 0 do seguinte grafo:



Nas figuras seguintes, o número em cada bola é a distância estimada.



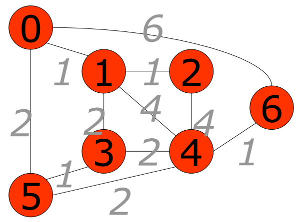
Filme do algoritmo de Dijkstra (cont.)

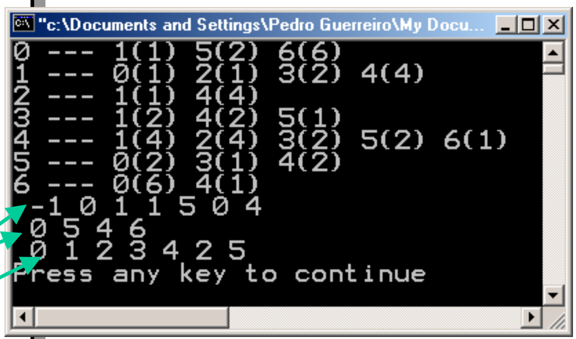


Testando o algoritmo de Dijkstra

Eis a função de teste que corresponde ao exemplo usado:

```
void TestShortestPath_Dijkstra()
{
  Vector<int>::SetPrefixSuffix(" ", "");
  Vector<double>::SetPrefixSuffix(" ", "");
  ListDouble<int>::SetPrefixSuffix(" ", "");
  GraphWeightedUsingList g(7);
  g.LinkUndirected(0, 1, 1);
  g.LinkUndirected(0, 5, 2);
  g.LinkUndirected(0, 6, 6);
  g.LinkUndirected(1, 2, 1);
  g.LinkUndirected(1, 3, 2);
  g.LinkUndirected(1, 4, 4);
  g.LinkUndirected(2, 4, 4);
  g.LinkUndirected(3, 4, 2);
  g.LinkUndirected(3, 5, 1);
  g.LinkUndirected(4, 5, 2);
  g.LinkUndirected(4, 6, 1);
  g.Write();
  ShortestPath_Dijkstra spd(g);
  spd.Compute(0);
  spd.Predecessors().WriteLine();
  spd.Path(6).WriteLine();
  spd.Distances().WriteLine();
}
```




"c:\Documents and Settings\Pedro Guerreiro\My Docu...

```
0 --- 1(1) 5(2) 6(6)
1 --- 0(1) 2(1) 3(2) 4(4)
2 --- 1(1) 4(4)
3 --- 1(2) 4(2) 5(1)
4 --- 1(4) 2(4) 3(2) 5(2) 6(1)
5 --- 0(2) 3(1) 4(2)
6 --- 0(6) 4(1)
-1 0 1 1 5 0 4
0 5 4 6
0 1 2 3 4 2 5
Press any key to continue
```


Análise do algoritmo de Dijkstra

Tal como o algoritmo de Prim, a eficiência do algoritmo de Dijkstra depende da implementação da fila com prioridade. Numa fila com monte, tirar o elemento da fila é uma é Log *V* (há no máximo *V* vértices na fila), e faz-se *V* vezes, o que dá *V**Log *V* ao todo.

Por outro lado, se procurar um vértice na fila for realizado em tempo constante, aumentar a prioridade de um vértice dado é Log *V*. Esta operação faz-se no máximo uma vez por cada aresta, o que dá *E**Log *V*. (A variável *E* representa o número de arestas.)

Portanto, o algoritmo de Dijkstra é (*V*+*E*)*Log *V*.