Estruturas de Dados

Luiz Alberto do Carmo Viana

August 24, 2020

Abstract

Notas de aula para as disciplinas de Estrutura de Dados e Estrutura de Dados Avançada do curso de Ciência da Computação do campus da UFC em Crateús.

Contents

1 Introdução

Logo após os conceitos básicos de programação, é necessário aprender a escrever programas que façam um bom uso dos recursos do computador. Tais recursos consistem, dentre outros, das unidades de processamento e das unidades de armazenamento.

Em se tratando de armazenamento, ou mais precisamente da memória do computador, é sabido que boa parte da execução de um programa consiste em obter dados que devem ser, de alguma forma, processados. Nota-se, portanto, que o custo de execução de um programa não depende apenas do tempo de processamento dos dados, mas também do tempo de acesso a eles.

Nestas notas de aula, vamos lidar com boas práticas para a organização e gerenciamento dos dados na memória do computador. Com elas, nossos programas podem armazenar, e portanto acessar, de forma eficiente os dados que produzem ou recebem de seus usuários.

Para descrever e implementar nossas estruturas de dados, vamos utilizar a linguagem `C++`, em seu padrão `C++17`. Construiremos nossos programas com as ferramentas de compilação do projeto GNU, em particular o compilador `g++` e o *build manager* `make`.

1.1 Ferramentas

Primeiro, descrevemos como vamos utilizar o programa `make`. Para isso, vamos explicar o que é um arquivo `makefile` e, aos poucos, o conteúdo do nosso.

O arquivo `makefile` descreve como um projeto deve ser construído, e costuma situar-se no diretório raiz de seu projeto. Seu conteúdo consiste de variáveis e regras. Cada regra determina, por meio de uma "receita", como um alvo deve ser construído a partir de seus pré-requisitos. Vale notar que um alvo pode ser pré-requisito de outros alvos, o que permite uma abordagem hierárquica para a construção de um projeto.

Vamos enumerar o conteúdo de nosso `makefile`, a começar por suas variáveis.

```
# compiler and c++ standard variables
COMPILER = g++
STD      = c++17
```

Começamos criando duas variáveis, com o propósito de determinar, de maneira flexível, o compilador que vamos utilizar junto com o padrão da linguagem. Vamos fazer uso dessas variáveis para a construção de comandos. Observamos também que linhas comentadas devem ser iniciadas com `#`.

```
# directories to look for .h and .hpp files (preceded by -I parameter)
INCLUDE_DIRS = -I.
```

Com a variável `INCLUDE_DIRS`, listamos os diretórios em que vamos buscar por arquivos-cabeçalho. Como vamos utilizar essa informação em chamadas do compilador, precedemos o nome de cada diretório por `-I`. No presente momento, indicamos que apenas o diretório raiz do projeto (`.`) contém arquivos-cabeçalho de nosso interesse.

```
# compiler parameters for compiling and linking
COMPILING_OPTIONS = -c -g -std=$(STD) $(INCLUDE_DIRS) -Wall -Wextra
LINKING_OPTIONS   = -g -std=$(STD) -Wall -Wextra
```

Definimos a variável `COMPILING_OPTIONS` com os parâmetros a serem utilizados na fase de compilação. Observe a utilização das variáveis `STD` e `INCLUDE_DIRS`, definidas anteriormente. Para a fase de *linking*, usamos os parâmetros definidos em `LINKING_OPTIONS`.

```
# list of project headers
HEADERS = bstree.hpp avltree.hpp rbtree.hpp
```

A variável `HEADERS` lista os arquivos-cabeçalho que utilizaremos nestas notas de aula. Cada um desses arquivos deve conter as definições de exatamente uma estrutura de dados. Utilizaremos a extensão `.hpp`, uma vez que nossas classes terão seus métodos definidos diretamente em sua declaração.

```
# these variables define one tester program for each header
TESTERS_SRC = $(HEADERS:.hpp=_test.cpp)
TESTERS_OBJ = $(TESTERS_SRC:.cpp=.o)
TESTERS     = $(TESTERS_SRC:.cpp=)
```

Como é boa prática em programação, vamos definir um programa testador para cada uma de nossas estruturas de dados. Com a decisão de conter exatamente uma estrutura de dados por arquivo `.hpp`, basta haver um testador para cada cabeçalho.

A variável `TESTERS_SRC` determina justamente isso: para cada entrada em `HEADERS`, trocamos o sufixo `.hpp` por `_test.cpp`. De forma similar, a variável `TESTERS_OBJ` traz os nomes dos arquivos-objeto de nossos programas testadores, assim como `TESTERS` indica o nome de seus arquivos executáveis. Note o truque empregado em `TESTERS` para excluir a extensão `.cpp` dos nomes em `TESTERS_SRC`. Agora, vamos começar a descrever algumas "receitas".

```
# $< refers to prerequisite and $@ refers to target name

# instructions for creating object files from cpp files
%.o : %.cpp
	$(COMPILER) $(COMPILING_OPTIONS) $< -o $@
```

Em nossa primeira "receita", determinamos como deve ser obtido um arquivo-objeto qualquer (`%.o`) a partir de seu arquivo `.cpp` correspondente (`%.cpp`). A primeira linha significativa de nossa "receita" é da forma `target : prerequisites`, estabelecendo a relação de dependência entre arquivos `.o` e arquivos `.cpp`. As demais linhas, que devem ser indentadas com `TAB`, constituem os comandos necessários para a criação do alvo. Nesse caso, temos apenas um comando, criado a partir de variáveis que definimos anteriormente. Também fazemos uso de duas variáveis especiais: `\$@` refere-se ao nome do alvo, e `\$<` ao nome do primeiro pré-requisito (para os nomes de todos os pré-requisitos, podemos usar `\$\^`).

```
# instructions for creating tester programs and running them
%_test : %_test.o %.hpp
	$(COMPILER) $(LINKING_OPTIONS) $< -o $@
	./$@
```

Agora apresentamos a "receita" responsável por criar os executáveis testadores e executá-los. Dessa vez, utilizamos dois comandos: criamos o executável, que em seguida é utilizado.

```
# this indicates that these rules do not correspond to files
.PHONY : test clean
```

Esse trecho do `makfile` serve para indicar que as "receitas" `test` e `clean` não correspondem a nomes de arquivos (*phony* significa falso). Isso é útil para definirmos as formas como vamos invocar o programa `make`.

```
# instructions on how to clean the project (deleting some stuff)
clean :
	rm -f *.o
	rm -f $(TESTERS)
```

A "receita" `clean` tem a finalidade de remover arquivos intermediários ou auxiliares. Note que ela não tem pré-requisitos e usa dois comandos, deletando arquivos-objeto e programas testadores.

```
# this runs the test suite
test : $(TESTERS)
```

Por último, a "receita" `testers` executa todos os programas testadores que foram modificados desde sua última execução. Observe que, como seu propósito é apenas agrupar a execução de vários programas em uma única instrução, não é necessário que `test` tenha comandos próprios.

2 Estruturas de Dados Arbóreas

Vamos começar nosso estudo com Árvores Binárias de Busca simples, sem balanceamento. Em seguida, vamos apresentar técnicas que garantem uma boa distribuição de altura entre sub-árvores. Por fim, analisamos árvores que permitem mais de dois filhos por nó.

2.1 Árvores Binárias de Busca

2.1.1 Introdução

TODO escrever isso apenas ao lecionar Estruturas de Dados.

2.1.2 Implementação

Vamos criar o arquivo `bstree.hpp` para conter a declaração de uma classe que implementa, de forma genérica, o conceito de Árvore Binária de Busca. Junto a ela, estarão as definições de seus métodos, cada um correspondendo a uma operação simples: busca, inserção, e remoção.

```
// each compilation session must consider this file at most once
#pragma once

// we are going to use smart pointer facilities
#include <memory>
// this type is helpful to represent optional value returning
#include <optional>
```

Começamos `bstree.hpp` com algumas diretivas de pré-processamento. A declaração `#pragma once` determina que o código-fonte contido no arquivo deve ser avaliado uma única vez em cada sessão de compilação. Em seguida, temos dois `#include`: `<memory>` vai nos dar acesso aos ponteiros inteligentes de `C++`, automatizando o gerenciamento de memória; `<optional>` define um tipo genérico contendo zero ou um elementos de um certo tipo.

```
// generic types
template<typename Key, typename Val>
class BSTree{
  // ...
};
```

Definimos a classe `BSTree`, com algo que pode ser novo para o leitor. A linha de `template` cria dois símbolos, `Key` e `Val`, que serão usados, dentro de `BSTree`, para representar os tipos de chave e valor, respectivamente, a serem armazenados (em pares) nos nós de nossa `BSTree`. Isso nos permite declarar, por exemplo, uma `BSTree` que mapeia valores inteiros a strings com `BSTree<int, std::string>`. Vamos agora preencher o conteúdo dessa classe.

```
// ...
private:
  // node of BSTree
  class BSTreeNode{
  // ...
  };

  // root node of BSTree
  std::unique_ptr<BSTreeNode> root;
// ...
```

Primeiro, falamos dos membros privados de `BSTree`. A classe aninhada `BSTreeNode` (a ser definida futuramente) é responsável por representar um nó de nossa `BSTree`, junto com algumas operações. O outro membro privado é um ponteiro inteligente cuja função é referenciar o nó raiz de nossa `BSTree`.

Um ponteiro `unique_ptr` traz a garantia de ser o único ponteiro inteligente que referencia um certo endereço de memória. Assim, vemos que é razoável `root` ser do tipo `std::unique_ptr<BSTreeNode>`, afinal, para preservar as propriedades de instâncias de nossa estrutura de dados, é saudável que apenas elas tenham acesso a suas respectivos raizes.

```
// ...
public:
  // constructor to create an empty BSTree
  BSTree() : root{nullptr}
  {}

  // constructor to create a BSTree rooted by Key and Val
  BSTree(Key key, Val val) : root{std::make_unique<BSTreeNode>(key, val)}
  {}

  // ...
```

Como nossos primeiros membros públicos, apresentamos dois construtores para `BSTree`. Em `C++`, construtores têm, além de um corpo (a sequência de instruções delimitadas por chaves), uma lista de inicialização para as variáveis-membro de sua classe.

O primeiro construtor não recebe argumentos e cria uma `BSTree` vazia, com `root` assumindo o valor `nullptr`. Já o segundo construtor, que recebe valores de tipos `Key` e `Val`, deve criar o nó raiz de sua instância, e para isso utiliza a função `make_unique`. Observe que ambos têm um corpo vazio.

```
// ...
public:
  // ...

  bool isEmpty(){
    return root == nullptr;
  }

  // returns Key Val pair whose Val corresponds to the maximum BSTree
  // Key
  std::optional<std::pair<Key, Val>> maxKey(){
    if (root){
      return root->maxKey();
    }
    else{
      return {};
    }
  }

  // returns Key Val pair whose Val corresponds to the minimum BSTree
  // Key
  std::optional<std::pair<Key, Val>> ninKey(){
    if (root){
      return root->minKey();
    }
    else{
      return {};
    }
  }

  // ...
```

Aqui temos mais três métodos públicos. Não há tanta necessidade de explicar `isEmpty`, dada sua simplicidade.

Note que `maxKey` e `minKey` retornam valores de mesmo tipo. Utilizamos `std::pair<Key, Val>` para declarar um par cuja primeira (segunda) componente tem um valor de tipo `Key` (`Val`). Além disso, fazemos uso do tipo `std::optional<std::pair<Key, Val»` para indicar que os métodos retornam um objeto que pode conter um par ou nada. Caso a `BSTree` não esteja vazia, ambos delegam seu retorno para métodos homônimos de `BSTreeNode`.

```
// ...
public:
  // ...

  // searches for Key, returning the corresponding Value or nothing
  std::optional<Val> search(Key key){
    if (root){
      return root->search(key);
    }
    else{
      return {};
    }
  }

  // inserts Val attached to Key in case Key is not present
  // yet. Return value indicates whether insertion really took place
  bool insert(Key key, Val val){
    if (root){
      return root->insert(key, val);
    }
    else{
      root = std::make_unique<BSTreeNode>(key, val);
      return true;
    }
  }

  // removes Key and corresponding attached Val. Return value
  // indicates whether removal really took place
  bool remove(Key key){
    // ...
  }
```

Agora temos as três operações principais em `BSTree`. Por hora, vamos definir `search` e `insert`, ambos delegando sua operação para métodos homônimos de `BSTreeNode` caso `BSTree` não esteja vazia.

É digno de nota que `search` e `insert` usam seus retornos para indicar se a operação foi bem-sucedida ou não. Em `insert`, é retornado `false` sse a chave a ser inserida já está presente na `BSTree`. Já em `search`, usa-se `std::optional<Val>` para permitir que se retorne nada caso a chave buscada não esteja presente na `BSTree`. Definimos agora a classe `BSTreeNode`.

```
class BSTreeNode{
public:
  // since this is an internal private class, there is no need to
  // private members
  Key key;
  Val val;
  // smart pointers to left and right subtrees: these guys deal with
  // memory deallocation by themselves
  std::unique_ptr<BSTreeNode> left;
  std::unique_ptr<BSTreeNode> right;

  // ...
};
```

Como se trata de uma classe aninhada privada, não existe a necessidade de declarar seus membros como privados. Como variáveis, `BSTreeNode` armazena um par de chave e valor, além de ponteiros inteligentes para suas sub-árvores esquerda e diretia. Como descrevem os comentários, ponteiros inteligentes são capazes de lidar com a desalocação de memória.

```
// ...
public:
  // ...

  // constructor: notice that member initialization is done outside
  // of the constructor body
  BSTreeNode(Key k, Val v) : key{k},
                             val{v},
                             left{nullptr},
                             right{nullptr}
  {}

  // returns Key Val pair whose Key is maximum
  std::pair<Key, Val> maxKey(){
    if (right){
      return right->maxKey();
    }
    else{
      return std::make_pair(key, val);
    }
  }

  // returns Key Val pair whose Key is minimum
  std::pair<Key, Val> minKey(){
    if (left){
      return left->minKey();
    }
    else{
      return std::make_pair(key, val);
    }
  }

  // ...
```

Não há nada novo no único construtor de `BSTreeNode`. Já os métodos `minKey` e `maxKey` valem-se das invariantes de `BSTree` para retornar apropriadamente.

Como a chave de um nó de `BSTree` é menor que a de qualquer nó em sua sub-árvore direita, ela é mínima sse sua sub-árvore esquerda é vazia. Em caso negativo, buscamos a chave mínima da sub-árvore esquerda. Isso justifica a corretude de `minKey`, e um argumento análogo se aplica a `maxKey`.

```
// ...
public:
  // ...

  // searches for Key, returning a Val or nothing
  std::optional<Val> search(Key k){
    // current node contains requested key
    if (k == key){
      return val;
    }
    // left subtree is not empty and requested key may be at it
    else if (left && k < key){
      return left->search(k);
    }
    // the same in regard of right subtree
    else if (right && k > key){
      return right->search(k);
    }
    // if requested key cannot be found, return nothing
    return {};
  }

  // ...
```

O método `search` recebe como argumento um valor do tipo `Key` e talvez retorne um valor do tipo `Val`, encapsulado em `optional`. Em `search`, vemos o uso das invariantes de `BSTree` na condução da busca por `k`: caso `k` não seja a raiz do nó atual, deve-se buscar por `k` na sub-árvore direita ou esquerda, a depender de como `k` se compara com `key` e se a devida sub-árvore é não-vazia.

```
// ...
public:
  // ...

  // inserts Val attached to Key in case Key is not present. Return
  // value indicates whether insertion really happened
  bool insert(Key k, Val v){
    // current node already contains Key, so does nothing
    if (k == key){
      return false;
    }
    // insertion may occur at left subtree
    else if (k < key){
      // if left subtree is not empty, recursively inserts into it
      if (left){
        return left->insert(k, v);
      }
      // if left subtree is empty, insertion will occur
      else{
        left = std::make_unique<BSTreeNode>(k, v);
      }
    }
    // same idea but applied to right subtree
    else if (k > key){
      if (right){
        return right->insert(k, v);
      }
      else{
        right = std::make_unique<BSTreeNode>(k, v);
      }
    }
    // if execution reaches this line, insertion indeed has occured,
    // so returns accordingly
    return true;
  }
```

No método `insert`, mais uma vez as invariantes de `BSTree` ficam evidentes. Caso a inserção seja conduzida para uma sub-árvore vazia, ela de fato ocorre, criando o primeiro nó daquela sub-árvore. Caso a sub-árvore não esteja vazia, a inserção é tentada recursivamente. Enfim definimos `remove`.

```
bool remove(Key key){
  // if BSTree is not empty, we may have something to delete
  if (root){
    // ...
  }
  // if BSTree is empty, we simply indicate nothing was deleted
  else{
    return false;
  }
}
```

A primeira coisa que `remove` verifica é se a árvore está vazia. Em caso afirmativo, nada vai ser removido e o retorno indica isso.

```
if (root){
  // raw pointers to perform a traversal
  BSTreeNode* currentNode = root.get();
  // since currentNode starts pointing towards root, it has no
  // parent node
  BSTreeNode* parentNode  = nullptr;
  // tries to find requested key. This may end up in an empty
  // subtree
  while (currentNode && key != currentNode->key){
    // updates parentNode
    parentNode = currentNode;
    // goes either left or right accordingly
    if (key < currentNode->key){
      currentNode = currentNode->left.get();
    }
    else if (key > currentNode->key){
      currentNode = currentNode->right.get();
    }
  }

  // ...
}
```

Caso a árvore não esteja vazia, fazemos uma "descida" em sua estrutura, buscando pela chave a ser removida. Para isso, usamos ponteiros simples `currentNode` e `parentNode` cujo propósito é indicar, respectivamente, o nó atual e seu pai. Como nossa busca começa pela raiz, `parentNode` é inicialmente nulo.

Vale observar que, apesar de estarmos usando o método `get` de um ponteiro inteligente para ter acesso ao ponteiro simples que ele encapsula, não seria boa prática desalocar a região referenciada pelo ponteiro simples (com `delete` ou `free`), uma vez que essa responsabilidade é atribuída ao ponteiro inteligente. Isso quer dizer que, no momento de destruição do ponteiro inteligente, a região de memória por ele referenciada será invariavelmente desalocada, e caso já o tenha sido, teremos um *double free error*. Assim, usamos ponteiros simples apenas para "passear" pela estrutura, e nenhum gerenciamento de memória os compete.

O laço `while` apresentado faz `currentNode` "descer" apropriadamente na estrutura da árvore sempre que a chave buscada é diferente de sua chave. Além disso, `parentNode` mantém o valor anterior de `currentNode`.

```
if (root){
  // ...

  // now we must verify why we exited the while loop
  // if currentNode is not nullptr, we exited the while loop
  // because key == currentNode->key, so currentNode content must
  // be deleted
  if (currentNode){
    // ...
  }
  // if currentNode is nullptr, then the only subtree that could
  // contain key is empty, so no deletion is performed
  else{
    return false;
  }
}
```

Uma vez fora do `while`, é preciso verificar que parte de sua condição foi violada. Caso a saída tenha ocorrido por conta de `currentNode` assumir um valor nulo, então a "descida" em busca da chave a ser removida nos levou a uma sub-árvore vazia, o que indica que a chave que buscávamos não existia na árvore.

```
if (currentNode){
  // currentNode has no subtrees, so it can safely be deleted
  if (currentNode->left == nullptr && currentNode->right == nullptr){
    // ...
  }
  // currentNode has both subtrees not empty
  else if (currentNode->left && currentNode->right){
    // ...
  }
  // currentNode has exactly one subtree not empty
  else{
    // ...
  }

  return true;
}
```

Agora que vamos de fato lidar com a remoção de um nó da árvore, nos deparamos com três cenários possíveis: o nó não tem sub-árvores significativas, e portanto pode ser simplesmente excluído; o nó tem ambas as suas sub-árvores não-vazias; o nó tem exatamente uma sub-árvore não vazia, e esta vai ocupar o seu lugar. Vamos tratar cada um desses casos.

```
if (currentNode->left == nullptr && currentNode->right == nullptr){
  // currentNode is not root
  if (parentNode){
    // ...
  }
  // currentNode is root, so we simply deallocate BSTreeNode
  // at root
  else{
    root = nullptr;
  }
}
```

Caso o nó não tenha sub-árvores significativas, podemos removê-lo. Contudo, precisamos verificar se ele é a raiz da árvore, pois nesse caso é preciso atualizar `root`. Note ainda que, como `root` é um `unique_ptr`, `root` não perde sua referência sem antes desalocá-la (isso pode ser feito de forma segura, dado que um `unique_ptr` mantém uma referência de forma exclusiva). Assim, atribuir `nullptr` a `root` é o suficiente.

```
if (parentNode){
  // currentNode is parentNode's left child
  if (currentNode->key < parentNode->key){
    // this assignment is enough to deallocate currentNode
    parentNode->left = nullptr;
  }
  // currentNode is parentNode's right child
  else if (currentNode->key >  parentNode->key){
    parentNode->right = nullptr;
  }
}
```

Quando o nó não é a raiz da árvore, precisamos verificar como ele se relaciona com seu pai. Assim, podemos determinar qual filho de `parentNode` deve ser removido. Note ainda que `parentNode->left` e `parentNode->right` são do tipo `unique_ptr`. Vamos para o próximo cenário de remoção.

```
else if (currentNode->left && currentNode->right){
  // we could also have taken currentNode->right->minKey
  auto[leftMaxKey, leftMaxVal] = currentNode->left->maxKey();

  remove(leftMaxKey);

  currentNode->key = leftMaxKey;
  currentNode->val = leftMaxVal;
}
```

No caso em que o nó a ser deletado tem ambas as sub-árvores não-vazias, curiosamente não é ele quem é removido. Em vez disso, tomamos o conteúdo do nó com a maior chave em sua sub-árvore esquerda e “copiamos” esse conteúdo no nó que deveria ser removido. Isso faz com que o conteúdo que deveria ser deletado de fato desapareça da árvore, e nos permite remover um nó com ao menos uma sub-árvore vazia.

Exercício 1. *O que aconteceria caso* `remove(leftMaxkey);` *fosse posta como a última instrução em seu bloco?*

Exercício 2. *Por que o nó de* `leftMaxkey` *tem ao menos uma sub-árvore vazia?*

Precisamos ainda destacar uma novidade sintática. Como o método `maxKey` de `BSTreeNode` retorna um valor do tipo `std::pair<Key, Val>`, podemos receber esse retorno de forma desestruturada, isto é, atribuindo separadamente os valores de tipos `Key` e `Val` a variáveis recém-criadas. É precisamente esse o propósito da sintaxe utilizada na linha 3 da última listagem.

```
else{
  // currentNode is not root
  if (parentNode){
    // ..
  }
  // currentNode is root, so we update root to be its only
  // nonempty subtree
  else{
    if (currentNode->left){
      root = std::move(currentNode->left);
    }
    else if (currentNode->right){
      root = std::move(currentNode->right);
    }
  }
}
```

Agora lidamos com o caso em que o nó a ser removido tem exatamente uma sub-árvore não-vazia. Nesse cenário, o nó pode ser desalocado, com sua única sub-árvore não-vazia agora referenciada por seu pai.

Mais uma vez, devemos fazer a distinção se o nó a ser removido é a raiz da árvore ou não. Caso seja, apenas sobrescrevemos `root` com a referência de sua única sub-árvore não-vazia.

Destacamos o uso de `std::move`. Em `C++` moderno, podemos tanto "copiar" como "recortar" variáveis. Para "copiar" uma variável, basta uma operação de atribuição. Já para "cortar", passamos a variável a ser "cortada" como argumento para `std::move` durante a operação de atribuição. Assim, após `a = std::move(b)`, o conteúdo de `b` deve estar em `a`, e acessar `b` pode ter comportamento indefinido, portanto não é recomendado.

Exercício 3. *Pesquise sobre* move semantics *em* `C++`.

Posto isso, perceba que não faz muito sentido "copiar" o conteúdo de um `unique_ptr`, já que haveriam ao menos dois deles apontando para um mesmo endereço. Inclusive, tentar fazer isso resultaria em um erro de compilação. No nosso caso, de fato queremos "cortar" a sub-árvore não-vazia de `root` e atribuí-la a `root`.

```
if (parentNode){
  // currentNode is parentNode's left child
  if (currentNode->key < parentNode->key){
    if (currentNode->left){
      // notice how we don't copy the unique_ptr. We move it
      // instead
      parentNode->left = std::move(currentNode->left);
    }
    else if (currentNode->right){
      parentNode->left = std::move(currentNode->right);
    }
  }
  // currentNode is parentNode's right child
  else if (currentNode->key > parentNode->key){
    if (currentNode->left){
      parentNode->right = std::move(currentNode->left);
    }
    else if (currentNode->right){
      parentNode->right = std::move(currentNode->right);
    }
  }
}
```

Caso o nó a ser removido não seja a raiz da árvore, atualizamos a sub-árvore apropriada de `parentNode`. Mais uma vez, precisamos determinar qual a única sub-árvore não-vazia de `currentNode`.

2.1.3 Análise de Complexidade

TODO escrever isso apenas ao lecionar Estruturas de Dados.

Exercício 4. *Prove que, se um nó em uma Árvore Binária de Busca tem dois filhos, então seu sucessor não tem filho esquerdo e seu antecessor não tem filho direito.*

2.2 Árvores AVL

Sabendo que as operações de busca, inserção e remoção de uma Árvore Binária de Busca têm complexidade $O(h)$, onde h é a altura da árvore, devemos fazer suas operações de modificação de forma a minimizar a altura da árvore resultante. Com esse objetivo, vamos definir o conceito de Árvore AVL, uma estrutura de dados autoajustável.

2.2.1 Definição

Uma Árvore AVL é uma Árvore Binária de Busca onde cada nó, além de seus campos usuais, registra também a altura da sub-árvore nele enraizada. Com essa informação, podemos determinar invariantes que, uma vez obedecidas, garantem que uma Árvore AVL de n nós tem altura $O(\log n)$. Posto isso, as invariantes de uma Árvore AVL assseguram que suas operações básicas têm complexidade $O(\log n)$.

Antes de prosseguirmos, vale avisar que tratamos de forma indistinta nós e sub-árvores. Não há prejuízo em cometer esse abuso, posto que cada nó é raiz de exatamente uma sub-árvore. As sub-árvores vazias, que não têm raiz, fogem disso e serão tratadas explicitamente.

Primeiramente, convencionamos que uma sub-árvore vazia (enraizada por nenhum nó, portanto) tem altura -1. Uma sub-árvore sem filhos tem altura 0 e, de forma geral, a altura de um *node* é definida por

$$height(node) = \max(height(node.left), height(node.right)) + 1$$

onde $node.left$ e $node.right$ são seus dois filhos. Essa definição se relaciona com o número máximo de nós que uma árvore binária de altura h pode ter.

Exercício 5. *Prove que uma árvore binária de altura h tem no máximo $2^{h+1} - 1$ nós. Dica: tente usar indução.*

Além da altura, definimos o conceito de fator de balanceamento. O fator de balanceamento de um nó é definido por

$$balance(node) = height(node.right) - height(node.left)$$

Como invariante, cada *node* de uma Árvore AVL deve obedecer $balance(node) \in \{-1, 0, 1\}$. Em palavras, um nó não pode permitir que suas sub-árvores tenham uma diferença de altura maior que um. Antes de entender como manteremos essa invariante, vamos explicar por que ela fornece uma boa altura para a árvore.

Vamos denotar por $n(h)$ o número mínimo de nós que uma Árvore AVL de altura h deve ter. É certo que $n(0) = 1$. De forma geral, $n(h) = 1 + n(h_L) + n(h_R)$ (somamos esse 1 para a raiz), onde h_L e h_R são as alturas das sub-árvores esquerda e direita, respectivamente.

Uma Árvore AVL de altura h deve ter ao menos uma de suas sub-árvores com altura $h-1$. Pela invariante, a outra sub-árvore pode ter altura $h-1$ ou $h-2$. Como estamos interessados no número mínimo de nós, vamos tomar a outra sub-árvore como tendo altura $h-2$. Assim, sem perda de generalidade, podemos assumir que $h_L = h-1$ e $h_R = h-2$. Isso nos dá a seguinte relação de recorrência.

$$n(0) = 1 \tag{1}$$
$$n(1) = 2 \tag{2}$$
$$n(h) = n(h-1) + n(h-2) \tag{3}$$

Como essa relação de recorrência é muito similar à recorrência de Fibonacci, é possível argumentar que (TODO da próxima vez que lecionar EDA, escrever o argumento)

$$n(h) \approx \left(\frac{1+\sqrt{5}}{2}\right)^h$$

Assim, sabemos que $n(h) \approx \phi^h$, onde ϕ é a *proporção áurea*. Isso nos permite concluir que $h \approx \log_\phi n(h)$, e portanto uma Árvore AVL de n nós tem altura $h \in O(\log n)$ (já que a mudança de base de um logaritmo é apenas a multiplicação por uma constante). Agora descrevemos como manter a invariante de uma Árvore AVL após uma modificação.

2.2.2 Rotações

Após uma operação de inserção ou remoção, é possível que haja algum *node* na árvore com $balance(node) \in \{-2, 2\}$. Sob essa condição, dizemos que um nó está *desbalanceado*. Portanto, logo após a operação modificadora, devemos garantir que não haja nós desbalanceados.

Exercício 6. *Explique por que não é possível, mesmo após uma inserção ou remoção, haver um node com* $balance(node) \notin \{-2, -1, 0, 1, 2\}$.

No caso de uma inserção, perceba que só pode haver nós desbalanceados no caminho da raiz até o nó recém-inserido. Dessa forma, devemos nos preocupar em manter a invariante desses nós, já que os demais não são afetados pela operação.

Já na remoção, apenas pode haver nós desbalanceados no caminho da raiz até o pai que teve um filho removido. Dado que esse cenário é muito parecido com o da inserção, vamos tirar vantagem disso em nossa implementação.

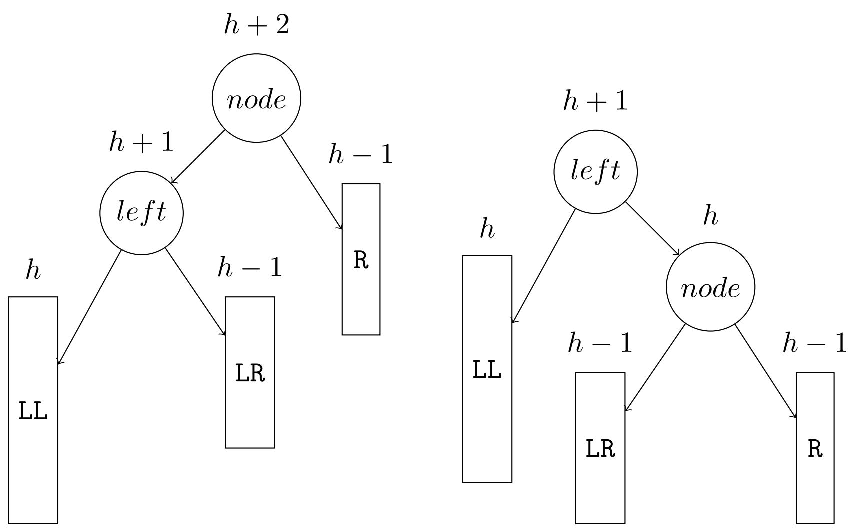
Vale destacar que em ambos os casos, queremos tratar os nós começando pelo mais distante da raiz, indo em direção à raiz. Essa sequência nos garante que as operações que fazemos nesses nós não criam novos desbalanceamentos e também simplifica o número de casos que devemos considerar.

Para rebalancear um nó, uma única operação local de tempo constante é necessária. Chamamos esse tipo de operação de *rotação* pelo fato de alguns nós mais "baixos" parecerem estar "subindo" e outros mais altos parecerem estar "descendo", sempre respeitando um sentido "direita-esquerda" ou "esquerda-direita".

Há quatro cenários de desbalanceamento em um *node* que devemos considerar:

1. $balance(node) = -2$ e $balance(node.left) = -1$;
2. $balance(node) = -2$ e $balance(node.left) \in \{0, 1\}$;
3. $balance(node) = 2$ e $balance(node.right) = 1$
4. $balance(node) = 2$ e $balance(node.right) \in \{-1, 0\}$

Primeiro, é certo que há uma simetria entre os casos 1 e 2 e os casos 3 e 4. Nos casos 1 e 2, dizemos que *node* está *left-heavy*. Já nos casos 3 e 4, *node* se encontra *right-heavy*. Vamos ilustrar os casos *left-heavy*.

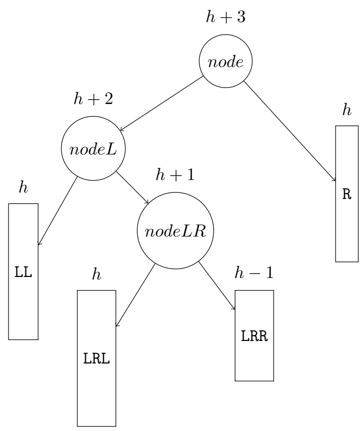


(a) Antes da rotação (b) Depois da rotação

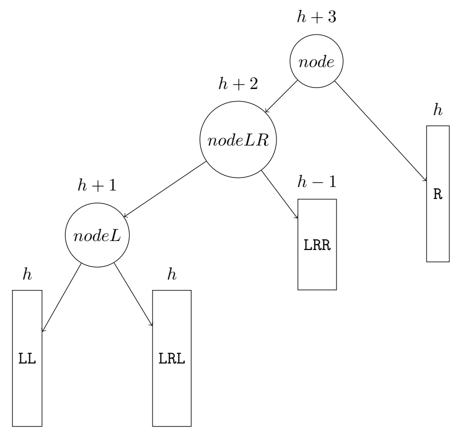
Figure 1: Rotação simples à direita. Nós representados como círculos e sub-árvores como retângulos. Acima dos nós e sub-árvores, representamos sua altura.

Para resolver o caso 1, usamos uma rotação simples à direita. A Figura 1 ilustra o procedimento dessa rotação. Note que trata-se apenas da atualização do conteúdo de dois nós, e de algumas atualizações de referências.

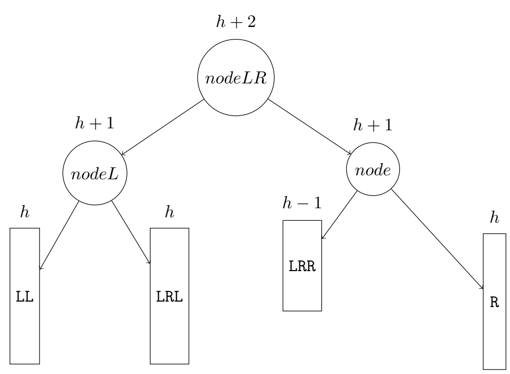
Assim, essa rotação pode ser executada em tempo constante, e após feita, é necessário atualizar as alturas de $node$ e $left$.



(a) Antes da rotação



(b) Depois da primeira rotação



(c) Depois da segunda rotação

Figure 2: Rotação dupla: à esquerda em um nível mais baixo, e à direita em um nível mais alto. Nós são círculos e sub-árvores são retângulos. Acima de cada nó e sub-árvore está representada sua altura.

No caso 2, é necessário fazer uma rotação dupla. A Figura 2 traz uma ilustração de como tal rotação é feita. Primeiro, faz-se uma rotação simples à esquerda no filho esquerdo, e em seguida uma rotação simples à direita no nó. Novamente, essa rotação tem um custo constante, e quando realizada, devem ser atualizadas as alturas de $node$, $nodeL$ e $nodeLR$. Como os casos *right-heavy* são meros “espelhos” dos casos aqui ilustrados, rotações análogas às apresentadas são suficientes para corrigí-los.

Exercício 7. *Descreva uma forma de converter uma Árvore Binária de Busca com n nós em uma Árvore AVL em tempo $O(n \log n)$.*

Exercício 8. *Em uma árvore binária, um nó é dito filho único sse ele tem pai e seu nó pai tem exatamente um filho. Vamos definir a* razão de solidão *de uma árvore binária T ($RS(T)$) como o número de filhos únicos em T dividido pelo número de nós de T. Prove que, se T é uma Árvore AVL, então $RS(T) \leq \frac{1}{2}$. Dica: quais nós de uma Árvore AVL podem ser filhos únicos?*

2.2.3 Implementação

Vamos mostrar uma abordagem de implementação de Árvores AVL utilizando herança a partir de `BSTree`. Para isso, o código de `BSTree` precisou sofrer alguma refatoração, e assim permitir um maior reuso de seus métodos. Vamos explicar a refatoração realizada sempre que for necessário.

```
#pragma once
// for max function
#include <algorithm>
// smart pointers
#include <memory>
// optional type
#include <optional>
// stack type
#include <stack>
// we are going to inherit from BSTree
#include <bstree.hpp>
```

Não há muita novidade nos cabeçalhos. Importamos `algorithm` para usar a função `std::max` e `stack` para utilizar pilhas como uma representação de caminhos da raiz até um certo nó da árvore. Além disso, utilizaremos `bstree.hpp` para fazer a herança.

```
template<typename Key, typename Val>
class AVLTree : public BSTree<Key, Val>{
  // ...
}
```

Descrevemos a classe `AVLTree` como uma subclasse de `BSTree`, sob os mesmos parâmetros de chave e valor. É certo que não faria muito sentido, por exemplo, que `AVLTree<int, char>` fosse subclasse de `BSTree<std::string, int>`. Vamos descrever os membros de `AVLTree`.

```
private:
  // type alias for saving us from some typing
  using BST = BSTree<Key, Val>;
  // the node of AVLTree
  struct AVLTreeNode{
    // ...
  };
  // returns heights of left and right subtrees
  static std::pair<int, int> childrenHeights(const AVLTreeNode* node){
    // ...
  }
  // calculates balance factor of a given node
  static int balanceFactor(const AVLTreeNode* node){
    // ...
  }
  // updates height of node based on the heights of its children
  static void updateHeight(AVLTreeNode* node){
    // ...
  }
  // ...
```

Começando a descrição dos membros privados de `AVLTree`, temos uma `struct` para representar o nó de uma `AVLTree`, `AVLTreeNode`. Vale lembrar que, com a refatoração, também transformamos `BSTreeNode` em uma `struct`, e minimizamos a quantidade de operações delegadas aos nós. Agora, apenas delegamos `minKey` e `maxKey`.

Em `C++`, não há muita diferença entre `struct` e `class`. A única distinção é que, por padrão, membros de `struct` são públicos e membros de `class` são privados. No entanto, é boa prática designar como `struct` entidades mais simples (como os nós, que passaram a ser após a refatoração) e como `class` entidades mais complexas.

Mantendo a simplicidade de `AVLTreeNode`, três funções auxiliares são implementadas como `static`, e não como métodos de `AVLTreeNode`. A função `childrenHeights` retorna um par de `int`, cada um sendo a altura de um filho. A função `balanceFactor` retorna o *balance* de um nó passado como argumento. Por fim, a função `updateHeight` atualiza a altura de um nó, baseando-se apenas na altura de seus filhos. Vamos descrever em detalhes `AVLTreeNode`.

```
struct AVLTreeNode{
  Key key;
  Val val;
  // smart pointers
  std::unique_ptr<AVLTreeNode> left;
  std::unique_ptr<AVLTreeNode> right;
  // height of node (no negative value makes sense)
  unsigned int height;
  // initializes AVLTreeNode.  Since it has no children, it has
  // height zero
  AVLTreeNode(Key k, Val v) : key{k},
                              val{v},
                              left{nullptr},
                              right{nullptr},
                              height{0}
  {}
  // returns Key Val pair whose Key is maximum.  We need this
  // information on every subtree for the removal algorithm
  std::pair<Key, Val> maxKey() const {
    // ...
  }
  // returns Key Val pair whose Key is minimum
  std::pair<Key, Val> minKey() const {
    // ...
  }
};
```

Em `AVLTreeNode`, temos os mesmos campos de `BSTreeNode`, mais um inteiro não-negativo para representar a altura do nó. Os métodos `minKey` e `maxKey` têm a mesma implementação de `BSTreeNode`, e portanto não precisam ser apresentados. Inclusive, esse é o único reuso que não faremos.

O motivo por trás da refatoração são os ponteiros `left` e `right`. Note que, em `BSTreeNode`, eles apontam para tipos diferentes de nós, e portanto não poderíamos utilizá-los em `AVLTreeNode` caso `AVLTreeNode` fosse subclasse de `BSTreeNode` (até poderíamos, mas eu não gosto de fazer *casting*). Como não podemos ter herança entre os nós, boa parte do código de `BSTreeNode` foi passado para `BSTree`, a fim de maximizar o reuso. Os métodos `minKey` e `maxKey` permanecem nos nós porque precisamos dessa informação, em cada sub-árvore, para o procedimento de remoção.

```
// returns heights of left and right subtrees
static std::pair<int, int> childrenHeights(const AVLTreeNode* node){
  // get height of left and right subtrees
  int leftHeight  = node->left  ? node->left->height  : -1;
  int rightHeight = node->right ? node->right->height : -1;

  return std::make_pair(leftHeight, rightHeight);
}
```

A implementação de `childrenHeights` é conforme a definição de altura, inclusive quanto à convenção adotada para sub-árvores vazias. Como `childrenHeights` não deve alterar seu argumento, ele é recebido como `const`. Perceba ainda que, apesar da altura de um nó ser `unsigned int`, a altura de uma sub-árvore vazia precisa ser `int`.

```
// calculates balance factor of a given node
static int balanceFactor(const AVLTreeNode* node){
  auto[leftHeight, rightHeight] = childrenHeights(node);

  return rightHeight - leftHeight;
}
```

A função `balanceFactor` não exige grandes explicações. Vale notar, de qualquer maneira, que seu argumento é recebido como `const`.

```
// updates height of node based on the heights of its children
static void updateHeight(AVLTreeNode* node){
  // get height of left and right subtrees
  auto[leftHeight, rightHeight] = childrenHeights(node);
  // calculates new height
  unsigned int newHeight = std::max(leftHeight, rightHeight) + 1;
  node->height = newHeight;
}
```

A implementação de `updateHeight` é bastante intuitiva. Note o uso de `std::max` para calcular a maior altura entre os filhos de `node`.

```
private:
  // ...
  // implementation of AVLTree
  template<typename Node>
  class AVLTreeWithNode : public BST::template BSTreeWithNode<Node>{
    // ...
  };
```

Uma parte importante da refatoração é que a implementação de `BSTree` passou para a nova classe `BSTreeWithNode`. Em `BSTreeWithNode`, o tipo de nó a ser utilizado é recebido como um parâmetro. Dessa forma, `AVLTreeWithNode` pode ser subclasse de `BSTreeWithNode` utilizando como nó `AVLTreeNode`. No caso, quando formos criar uma instância de `AVLTreeWithNode`, usaremos `AVLTreeWithNode<AVLTreeNode>`, e isso fará com que o código de `BSTreeWithNode` seja definido para `AVLTreeNode`. Para instanciar `BSTreeWithNode`, basta `BSTreeWithNode<BSTreeNode>`.

```
private:
  // ...
  template<typename Node>
  class AVLTreeWithNode : public BST::template BSTreeWithNode<Node>{
  private:
    // since BST is not instantiated yet, we need to tell the compiler
    // that BSTreeWithNode is a template and a type name when instantiated
    using BSTWithNode = typename BST::template BSTreeWithNode<Node>;
  public:
    // builds an empty AVLTree. Basically delegates all the work to BSTreeWithNode
    AVLTreeWithNode() : BSTWithNode{}
    {}
    // Creates an AVLTree with a nonempty root
    AVLTreeWithNode(Key key, Val val) : BSTWithNode{key, val}
    {}
    // inserts a Key Val pair in case Key is not present.  Return
    // indicates whether insertion occurred
    bool insert(Key key, Val val){
      // ...
    }
    // removes key in case it is present.  Return value indicates
    // whether removal has occurred
    bool remove(Key key){
      // ...
    }
  };
  // ...
```

Com a descrição de `AVLTreeWithNode`, notamos que há o aproveitamento completo dos métodos `isEmpty`, `sesrch`, `maxKey` e `minKey` de `BSTreeWithNode`. Os métodos que modificam a estrutura, `insert` e `remove`, farão um reuso parcial das funcionalidades correspondentes de `BSTreeWithNode`. Isso se deve ao fato de que, após fazerem sua operação usual, eles precisam manter a invariante de balanceamento de `AVLTreeWithNode`. Vamos descrever protótipos desses métodos (que deverão ser completado em laboratório).

```
// inserts a Key Val pair in case Key is not present.  Return
// indicates whether insertion occurred
bool insert(Key key, Val val){
  // the actual insertion is made by BSTReeWithNode
  bool hasInserted = BSTWithNode::insert(key, val);
  // if insertion really happened ...
  if (hasInserted){
    // ... do avl stuff
  }

  return hasInserted;
}
```

Como podemos ver, o método `insert` delega a operação de inserção para `BSTreeWithNode`. Caso a inserção tenha de fato ocorrido, então é necessário verificar e manter o balanceamento de `AVLTreeWithNode`. Essa parte deve ser escrita em laboratório.

```
// removes key in case it is present.  Return value indicates
// whether removal has occurred
bool remove(Key key){
  // first we verify if key is present
  bool hasKey = BSTWithNode::search(key);
  // in case it is, removal will occurr
  if (hasKey){
    // performs removal, and returns parent key in case root was not deleted
    std::optional<Key> parentKey = BSTWithNode::removeExistingKey(key);
    // in case a node other than root has been deleted ...
    if (parentKey){
      // ... do avl stuff
    }
    return true;
  }
  // otherwise, we simply indicate that removal has not occurred
  else{
    return false;
  }
}
```

O método `remove` apresenta mais um produto da refatoração de `BSTree`, que é o método `protected` `removeExistingKey`. A finalidade de `removeExistingKey` é encapsular toda a lógica de remoção de uma chave existente na árvore, e deixar para o `remove` de `BSTreeWithNode` apenas o trabalho de verificar se a chave a ser removido existe e, em caso afirmativo, chamar `removeExistingKey`. A ideia por trás disso é que `remove` continua retornando `bool`, que é significativo para o usuário, mas `removeExistingKey` retorna um `optional` contendo a chave do pai do nó que foi deletado (caso ele tenha um pai, daí o `optional`). Dessa forma, o retorno de `removeExistingKey` facilita a implementação de `AVLTreeWithNode`.

Em sua lógica, `remove` se assemelha a `insert`. É verificada a existência da chave a ser removida. Caso ela esteja presente na árvore, a remoção é delegada para `BSTreeWithNode`. Com a informação fornecida por `removeExistingKey`, é possível manter o balanceamento de `AVLTreeWithNode`, caso seja necessário. Essa parte da implementação deve ser feita em laboratório.

Para facilitar a implementação do balanceamento de `AVLTreeWithNode`, existem alguns métodos `static` que auxiliam certas tarefas. Vamos descrevê-los a seguir.

```
// rebalances a node
static void rebalanceNode(AVLTreeNode* node){
  // calculates balance factor
  int nodeBalanceFactor = balanceFactor(node);
  // node is left-heavy
  if (nodeBalanceFactor <= -2){
    int leftBalanceFactor = balanceFactor(node->left.get());
    if (leftBalanceFactor <= -1){
      // which rotation?
    }
    else{
      // which rotation?
    }
  }
  // node is right-heavy
  else if (nodeBalanceFactor >= 2){
    int rightBalanceFactor = balanceFactor(node->right.get());
    if (rightBalanceFactor >= 1){
      // which rotation?
    }
    else{
      // which rotation?
    }
  }
}
```

O método estático `rebalanceNode` utiliza `balanceFactor` para determinar a lógica de balanceamento, e assim decidir qual rotação deve ser aplicada. Vamos descrever uma rotação simples à direita, para ilustrar como deve ser realizado esse tipo de operação.

```
// performs a simple right rotation on node
static void rotateR(AVLTreeNode* node){
  // first we set aside all the moving subtrees
  std::unique_ptr<AVLTreeNode> subtreeLL = std::move(node->left->left);
  std::unique_ptr<AVLTreeNode> subtreeLR = std::move(node->left->right);
  std::unique_ptr<AVLTreeNode> subtreeR  = std::move(node->right);
  // then we save the contents of the moving nodes
  std::pair<Key, Val> nodeContent      = std::make_pair(node->key, node->val);
  std::pair<Key, Val> leftChildContent = std::make_pair(node->left->key, node->left->val);
  // left child becomes node
  node->key = leftChildContent.first;
  node->val = leftChildContent.second;
  // node becomes the right child
  node->right = std::make_unique<AVLTreeNode>(nodeContent.first, nodeContent.second);
  // finally we rearrange the moving subtrees ...
  node->left         = std::move(subtreeLL);
  node->right->left  = std::move(subtreeLR);
  node->right->right = std::move(subtreeR);
  // ... and update heights on the affected nodes
  updateHeight(node->right.get());
  updateHeight(node);
}
```

Os comentários de `rotateR` descrevem bem seu funcionamento. É deixado como exercício prático implementar as demais rotações.

Para manter o balanceamento de `AVLTreeWithNode`, é necessário atualizar a altura de cada nó no caminho afetado pela modificação, "de baixo para cima". Após isso, percorre-se o mesmo caminho de nós, na mesma direção, efetuando os rebalanceamentos. A função `pathToExistingKey` de `BSTreeWithNode`, se usada adequadamente, nos permite determinar tal caminho a ser corrigido, representando-o como uma pilha de `AVLTreeNode*`.

```
// the compiler will deduce what Function is.  Applies func to each
// node on path
template<typename Function>
static void applyOnPath(Function func, std::stack<AVLTreeNode*> path){
  // node to have function applied on
  AVLTreeNode* currentNode = nullptr;
  // while there are nodes to be visited
  while (!path.empty()){
    // gets a new node
    currentNode = path.top();
    // apply function
    func(currentNode);
    // then discards its reference
    path.pop();
  }
}
```

A função `applyOnPath` executa uma operação em todas as referências de nó armazenadas numa pilha, em sua ordem de remoção. Essa é a base lógica tanto para atualizarmos as alturas de nós em um caminho afetado por modificações, quanto para realizarmos os devidos balanceamentos nos mesmos nós. As funções responsáveis por essas duas tarefas são descritas a seguir.

```
// does the necessary height updates for nodes on path
static void updateHeightsOnPath(std::stack<AVLTreeNode*> path){
  // notice how we do not need to specify Function
  applyOnPath(updateHeight, path);
}
// rebalances each node on path
static void rebalanceNodesOnPath(std::stack<AVLTreeNode*> path){
  applyOnPath(rebalanceNode, path);
}
```

Essencialmente, cada uma dessas funções é uma aplicação de `applyOnPath` com uma função diferente. Esse tipo de função, que recebe outras funções como argumentos, é chamado de *função de alta ordem*.

```
private:
  // ...
  // AVLTree with proper node type
  AVLTreeWithNode<AVLTreeNode> avlt;
// ...
```

Por fim, temos o último membro privado de `AVLTree`, que é justamente uma instância de `AVLTreeWithNode` utilizando `AVLTreeNode` como seu tipo de nó. Dessa forma, os métodos públicos de `AVLTree` apenas precisam ser redirecionados para os métodos públicos de `avlt`.

```
public:
  // builds an empty AVLTree
  AVLTree() : avlt{}
  {}
  // Creates an AVLTree with a nonempty root
  AVLTree(Key key, Val val) : avlt{key, val}
  {}
  // inserts a Key Val pair in case Key is not present.  Return
  // indicates whether insertion occurred
  bool insert(Key key, Val val){
    return avlt.insert(key, val);
  }
  // removes key in case it is present.  Return value indicates
  // whether removal has occurred
  bool remove(Key key){
    return avlt.remove(key);
  }
  // returns Val attached to Key.  In case Key is not present, returns
  // nothing
  std::optional<Val> search(Key key) const {
    return avlt.search(key);
  }
  // returns Key Val pair whose Val corresponds to the maximum BSTree
  // Key.  In case tree is empty, returns nothing
  std::optional<std::pair<Key, Val>> maxKey() const {
    return avlt.maxKey();
  }
  // returns Key Val pair whose Val corresponds to the minimum BSTree
  // Key.  In case tree is empty, returns nothing
  std::optional<std::pair<Key, Val>> ninKey() const {
    return avlt.ninKey();
  }
```

2.2.4 Análise de Complexidade

Como sabemos, uma Árvore Binária de Busca com altura h tem as suas operações com custo $O(h)$. Uma Árvore Binária de Busca com n nós tem, no melhor caso, $h \in O(\log n)$ e, no pior caso, $h \in O(n)$.

Em uma Árvore AVL com n nós, o custo das operações é o mesmo de uma Árvore Binária de Busca de altura $O(\log n)$, acrescido do custo de manutenção do balanceamento. Como atualização de altura e rotações são operações de tempo constante, e uma Árvore AVL precisa realizar cada uma dessas operações em um caminho de nós até a raiz, esse custo adicional é $O(\log n)$, já que sua altura é $O(\log n)$. Assim, o custo total das operações de uma Árvore AVL é $O(\log n)$.

2.3 Árvores Rubro-Negras

Continuando com o tópico de estruturas de dados autoajustáveis, apresentamos mais um tipo de Árvore Binária de Busca com uma boa altura. Dessa vez, usaremos um critério diferente para determinar um bom balanceamento.

2.3.1 Definição

Uma Árvore Rubro-Negra é uma Árvore Binária de Busca cujos nós, além das informações usuais, trazem consigo um atributo de cor que pode assumir exatamente um de dois valores: ou vermelho ou preto. Com esse novo atributo, Árvores Rubro-Negras conseguem estabelecer a propriedade de que nenhum caminho da raiz a uma folha tem o dobro de nós de outro tal caminho, o que garante um balanceamento razoável, certamente não tão bom quanto o de uma Árvore AVL.

As seguintes propriedades devem ser satisfeitas por uma Árovre Rubro-Negra:

1. Sua raiz é um nó preto;
2. Toda folha é um nó preto;
3. Se um nó é vermelho, então seus filhos são nós pretos;
4. Para cada nó, todos os caminhos a partir dele até uma folha contêm o mesmo número de nós pretos.

Vamos convencionar que, em vez de sub-árvores vazias, nós podem ter folhas pretas que não contém dados significativos. Dessa forma, todo nó significativo da árvore é um nó interno, e trataremos o tamanho de uma Árvore Rubro-Negra pelo número de seus nós internos (nós que não são folhas).

Exercício 9. *Prove que um nó vermelho tem exatamente dois filhos.*

Exercício 10. *Prove que um nó vermelho não pode ter seu pai com cor vermelha.*

Definimos $bh(node)$ (onde bh denota *black height*) como o número de nós pretos em um caminho qualquer a partir de $node$ (mas sem incluir $node$) até uma folha. A propriedade 4 garante que bh está bem definido para qualquer nó.

Teorema 1. *Uma Árvore Rubro-Negra com n nós internos tem altura no máximo $2\log(n+1)$.*

Proof. Inicialmente, vamos mostrar que a sub-árvore enraizada em $node$ contém no mínimo $2^{bh(node)} - 1$ nós internos. Vamos provar isso por indução na altura de $node$. Se $node$ tem altura 0, então $node$ é uma folha e portanto sua sub-árvore tem 0 nós internos, o que está de acordo com o mínimo de $2^{hb(node)} - 1 = 2^0 - 1 = 0$ nós internos. Agora tome $node$ com altura positiva. Dessa forma, $node$ é um nó interno e, conforme a nossa convenção, $node$ tem dois filhos. Cada filho de $node$ tem seu bh igual a ou $bh(node)$ (quando o filho é vermelho) ou $bh(node) - 1$ (quando o filho é preto). Com isso, sabemos que o bh de um filho de $node$ é pelo menos $bh(node) - 1$. Como os filhos de $node$ têm altura menor que a de $node$, podemos concluir, por hipótese, que a sub-árvore de cada filho de $node$ tem no mínimo $2^{bh(node)-1} - 1$ nós internos. Com isso, a sub-árvore de $node$ contém ao menos $2(2^{bh(node)-1} - 1) + 1 = 2^{bh(node)} - 2 + 1 = 2^{bh(node)} - 1$ nós internos.

Seja h a altura da árvore. De acordo com a propriedade 3, um caminho a partir da raiz até uma folha (mas sem incluir a raiz) tem no mínimo metade de seus nós pretos (Exercício 11), o que implica que o bh da raiz é no mínimo $\frac{h}{2}$. Como a árvore tem n nós internos, sabemos que

$$n \geq 2^{\frac{h}{2}} - 1 \tag{4}$$

$$n + 1 \geq 2^{\frac{h}{2}} \tag{5}$$

$$\log(n+1) \geq \log(2^{\frac{h}{2}}) \tag{6}$$

$$\log(n+1) \geq \frac{h}{2} \tag{7}$$

$$h \leq 2\log(n+1) \tag{8}$$

□

Exercício 11. *Prove que um caminho a partir da raiz até uma folha (mas sem incluir a raiz) tem no mínimo metade de seus nós pretos.*

Como consequência do Teorema 1, sabemos que as operações em uma Árvore Rubro-Negra tem custo $O(\log n)$. No entanto, é preciso assegurar que as operações modificadoras (inserção e remoção) mantenham as propriedades esperadas de uma Árvore Rubro-Negra.

Exercício 12. *Prove que, em qualquer sub-árvore de uma Árvore Rubro-Negra, o caminho mais longo de sua raiz a uma folha tem no máximo o dobro do número de nós do caminho mais curto de sua raiz até uma folha.*

2.3.2 Manutenção

Diferente de Árvores AVL, Árvores Rubro-Negras precisam apenas de rotações simples para a manutenção de suas propriedades. No entanto, pode ser preciso também modificar a cor de certos nós. Dessa forma, como rotações simples já foram ilustradas na Subseção 2.2, vamos discutir principalmente as mudanças de cor necessárias para as operações de inserção e remoção.

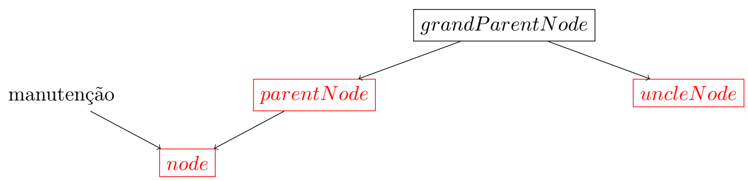
A operação de inserção é realizada da mesma forma que em uma Árvore Binária de Busca, e insere o novo nó (que chamaremos de *node*) como um nó interno, pai de duas folhas pretas. Para não perturbar a propriedade 4, *node* deve ter cor vermelha. Porém, é possível que *node* tenha pai, e esse ($parentNode$) também seja vermelho, o que violaria a propriedade 3. Caso *node* não tenha pai, *node* é a raiz e estamos violando a propriedade 1.

Se *node* for a raiz, podemos colorir *node* de preto sem prejuízo para as propriedades. No caso de *node* não ser raiz, vamos olhar para seu tio ($uncleNode$, irmão de $parentNode$) em três casos, e em cada um deles vamos fazer *node* respeitar a propriedade 3 sem violar as demais:

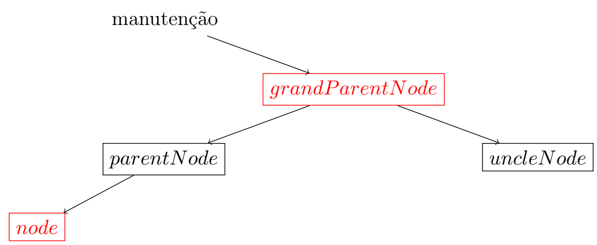
1. $uncleNode$ é vermelho: como $parentNode$ é vermelho e respeitava as propriedades antes da inserção de *node*, $parentNode$ não é raiz e seu pai ($grandParentNode$) é preto; assim, podemos "descer" a cor preta de $grandParentNode$, colorindo-o de vermelho e tornando pretos seus filhos, $parentNode$ e $uncleNode$; isso corrige *node*, mas agora é preciso fazer a manutenção de $grandParentNode$; ilustramos esse procedimento na Figura 3.

2. $uncleNode$ é preto e *node* é filho direito: aplicamos uma rotação simples à esquerda em $parentNode$, fazendo com que $parentNode$ torne-se filho esquerdo de *node*, e portanto precisamos fazer a manutenção de $parentNode$, o que nos leva ao caso 3; note que isso não viola a propriedade 4, já que os dois nós movidos são ambos vermelhos; a Figura 4 ilustra esse caso.

3. $uncleNode$ é preto e *node* é filho esquerdo: assim como argumentado no caso 1, $grandParentNode$ existe e é preto; "descemos" a



(a) Antes da operação



(b) Depois da operação

Figure 3: Ilustração do caso 1 da inserção.

cor preta de $grandParentNode$ para $parentNode$, o que pode violar a propriedade 4 para nós acima de $grandParentNode$ ou a propriedade 1, caso $grandParentNode$ seja raiz; para mitigar isso, aplicamos uma rotação simples à direita em $grandParentNode$; note que, como $grandParentNode$ (vermelho) passa a ser filho direito de $parentNode$ (preto), não há mais violações da propriedade 3, e portanto nenhum nó que precise de manutenção; a Figura 5 sintetiza essa operação.

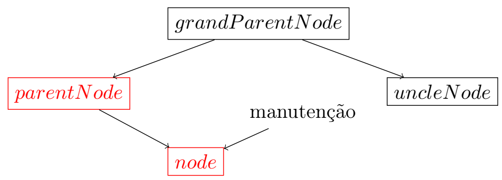
Exercício 13. *Argumente que, após uma inserção, é preciso fazer no máximo duas rotações simples para manter as propriedades de uma Árvore Rubro-Negra.*

Exercício 14. *Considere uma Árvore Rubro-Negra com n nós internos. Argumente que, se $n \geq 2$, então a árvore tem nós vermelhos.*

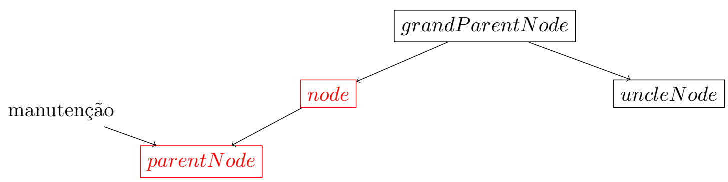
TODO Da próxima vez que lecionar a disciplina, tratar da remoção em Árvores Rubro-Negras.

2.3.3 Implementação

Estruturalmente, a implementação de `RBTree` é análoga à implementação de `AVLTree`, uma vez que ambas são subclasses de `BSTree`. Posto isso, e como iremos realizar a implementação de `RBTree` em laboratório, inclusive de suas funções auxiliares, não é necessária a apresentação de código-fonte.

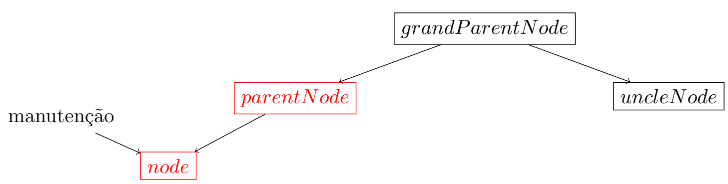


(a) Antes da operação

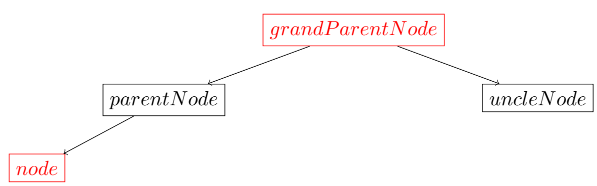


(b) Depois da operação

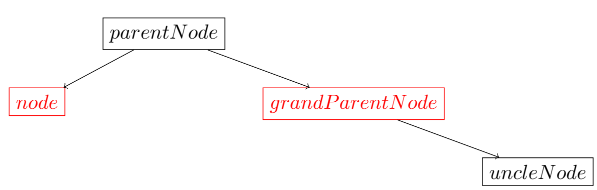
Figure 4: Ilustração do caso 2 da inserção.



(a) Antes da operação



(b) Depois da mudança de cores



(c) Depois da rotação

Figure 5: Ilustração do caso 3 da inserção.

2.3.4 Análise de Complexidade

Como sabemos, a complexidade das operações de uma Árvore Binária de Busca é $O(h)$, onde h é a altura da árvore. Uma Árvore Rubro-Negra com n nós internos tem altura $h \in O(\log n)$. Dessa forma, o custo de suas operações consiste do custo usual de $O(\log n)$ mais o custo de manutenção de suas propriedades, após cada operação de modificação.

Sabemos que o custo de manutenção após uma operação de modificação pode chegar a $O(\log n)$. Assim, o custo total das operações de uma Árvore Rubro-Negra é $O(\log n)$, ou seja, assintoticamente o custo de melhor caso para modificações em Árvores Binárias de Busca.

2.4 Árvores B

Árvores B são árvores de busca balanceadas, usadas principalmente em armazenamento secundário, como por exemplo sistemas de arquivos de discos e bancos de dados. São essencialmente uma generalização de Árvores Binárias de Busca, em que é permitido a um nó ter mais de uma chave.

Com um fator de ramificação bf (de *branching factor*) possivelmente maior que o de uma Árvore Binária de Busca ($bf \geq 2$) e usualmente determinado pelo tamanho de uma página de disco, uma Árvore B com n nós deve ter altura $O(\log_{bf} n)$. Isso permite que suas operações sejam realizadas em tempo $O(\log n)$.

2.4.1 Definição

Seja T uma Árvore B. As seguintes propriedades devem ser satisfeitas por T:

- cada *node* de T tem os seguintes atributos:
  - $node.n$, o número de chaves armazenadas em *node*;
  - as chaves $node.key_1, node.key_2, \ldots, node.key_n$, armazenadas de forma que $node.key_1 \leq node.key_2 \leq \cdots \leq node.key_n$;
  - Um campo booleano $node.leaf$, indicando se *node* é uma folha.
- cada *node* com $node.leaf$ falso deve ter $node.n{+}1$ sub-árvores $node.subtree_1, node.subtree_2, \ldots, node.subtree_{n+1}$;
- para cada *node* com $node.leaf$ falso, e para qualquer chave k_i da sub-árvore $node.subtree_i$, para $1 \leq i \leq node.n{+}1$, temos $k_1 \leq node.key_1 \leq k_2 \leq node.key_2 \leq \cdots \leq k_n \leq node.key_n \leq k_{n+1}$;

- todo *node* com $node.leaf$ verdadeiro está no mesmo nível, e esse nível é justamente a altura de T;
- cada *node* tem $node.n$ limitado, tanto inferior como superiormente. Isso significa que, dado $t \geq 2$:
  - cada *node*, a menos da raiz, deve ter $node.n \geq t-1$, e portanto deve ter no mínimo t sub-árvores;
  - cada *node* deve ter $node.n \leq 2t-1$, e portanto deve ter no máximo $2t$ sub-árvores. Se *node* tem $node.n = 2t-1$, dizemos que *node* está cheio.

Teorema 2. *Tome uma Árvore B com n chaves, $t \geq 2$ e altura h. Temos*

$$h \leq \log_t \left(\frac{n+1}{2}\right)$$

Proof. A raiz de uma Árvore B contém no mínimo uma chave, enquanto os demais nós contém no mínimo $t-1$ chaves. Assim, haveria a raiz no nível 0, ao menos dois nós no nível 1, ao menos $2t$ nós no nível 2, ao menos $2t^2$ nós no nível 3, e assim por diante, até termos ao menos $2t^{h-1}$ nós no nível h. Com isso, temos

$$\begin{aligned}
n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} && (9)\\
&= 1 + (t-1)\left(2\left(\frac{t^h-1}{t-1}\right)\right) && (10)\\
&= 1 + 2(t^h-1) && (11)\\
&= 1 + 2t^h - 2 && (12)\\
&= 2t^h - 1 && (13)
\end{aligned}$$

A partir de $n \geq 2t^h - 1$, temos $t^h \leq \frac{n+1}{2}$. Basta tomar $\log_t$ de ambos os lados dessa inequação para concluir a prova. □

Exercício 15. *Em função de h e t, qual o número máximo de chaves que uma Árvore B de altura h e parâmetro t pode conter? E o mínimo?*

2.4.2 Operações

Uma Árvore B suporta as operações usuais de busca, inserção e remoção. Dessas três, trataremos da remoção como uma atividade de laboratório.

Dessa forma, definimos aqui a implementação de uma Árvore B com as operações de busca e inserção, além de algumas funções auxiliares. Vamos apresnetar e descrever o conteúdo de `btree.hpp`.

```
#pragma once

// array type
#include <array>
// smart pointers
#include <memory>
// optional type
#include <optional>

// generic b tree
template<typename Key, typename Val, unsigned int t>
class BTree{
  // ...
};
```

Em `btree.hpp`, o único cabeçalho novo é `array`, que diferente de `vector`, implementa um array de tamanho estático. A classe `BTree` será nossa implementação de Árvore B, e ela tem três parâmetros: os tipos `Key` e `Val`, com seu significado usual; o inteiro não-negativo `t`, fazendo jus à definição de que uma página de `BTree` (a menos da raiz) deve ter entre `t - 1` e `2t - 1` chaves.

```
private:
  // represents a page of BTree
  struct Page{
    // indicates whether page is a leaf
    bool leaf;
    // number of keys currently stored
    unsigned int numberKeys;
    // arrays of keys and vals
    std::array<Key, 2*t - 1> key;
    std::array<Val, 2*t - 1> val;
    // children array
    std::array<std::unique_ptr<Page>, 2*t> child;
    // constructor
    Page() : leaf{false},
             numberKeys{0},
             key{},
             val{},
             child{}
    {}
    // returns whether page is full
    bool isFull(){
      return numberKeys == 2*t - 1;
    }
    // searches for Key. In case it is not present, returns nothing
    std::optional<Val> search(Key k){
      // ...
    }
    // splits a full child of this node, if this node is nonfull.  In
    // case child index is invald or the former conditions are not
    // met, returns false
    bool splitChild(unsigned int childIndex){
      // ...
    }
  };
// ...
```

Como primeiro membro privado de `BTree`, temos `Page`, representando uma página de uma árvore B. Seu membro `leaf` indica se a página é uma folha, enquanto `numberKeys` é o número de chaves atualmente nela armazenadas. Os membros `key` e `val` armazenam chaves e valores correspondentes em posições de mesmo índice. Caso a página não seja uma folha, `child` deve armazenar ponteiros para suas páginas filhas. Note que as dimensões de `key`, `val` e `child` estão de acordo com a definição de uma Árvore B.

Quanto aos métodos de `Page`, o construtor não inspira muitos comentários. A função `isFull` determina se a página está cheia, e isso ocorre quando o número de chaves nela armazenada é exatamente o número de posições de `key`. O método `search` é autodescritivo em sua finalidade, e `splitChild` divide um filho cheio de uma página não-cheia em dois.

```
std::optional<Val> search(Key k){
  unsigned int i = 0;
  // we go right searching for k
  while (i < numberKeys && k > key[i]){
    i++;
  }
  // now either key[i - 1] < k <= key[i] or i > numberKeys - 1
  // if k was found, we return the corresponding value
  if (i < numberKeys && k == key[i]){
    return val[i];
  }
  // now either key[i - 1] < k < key[i] or i > numberKeys - 1.  In
  // either case, we need to either go down or report an failed
  // search
  // if this page is a leaf, there is nowhere to go down, and the
  // search fails
  else if (leaf){
    return {};
  }
  // otherwise, we recurse into the appropriate child
  else{
    return child[i]->search(k);
  }
}
```

A função `search` admite que as chaves estão armazenadas em ordem crescente, e assim faz `i` assumir o menor índice tal que `k <= key[i]`. Caso `k == key[i]`, a chave buscada é encontrada, e `val[i]` é retornado. Caso contrário, sabe-se que `k < key[i]`, e portanto a chave pode se encontrar no filho "esquerdo" de `key[i]`, `child[i]`, onde a busca é feita recursivamente.

Há ainda as situações em que a página é uma folha e em que `k` é maior que todas as chaves da página. No primeiro cenário, se `k` não estiver presente na página, a busca encerra retornando nada. No segundo, note que `i == numberKeys`, e `k` pode se encontrar no filho "direito" de `key[numberKeys - 1]`, `child[numberKeys]`.

Exercício 16. *Desenvolva algoritmos para encontrar as chaves mínima e máxima de sub-árvores de uma Árvore B. Desenvovla também algoritmos para encontrar as chaves sucessora e predecessora de uma certa chave.*

```
bool splitChild(unsigned int childIndex){
  if (!isFull() && childIndex <= numberKeys && child[childIndex]->isFull()){
    // ...
  }
  else{
    return false;
  }
}
```

Antes de realizar sua operação, `splitChild` verifica se a página não está cheia, se `childIndex` é índice de algum filho da página e, por fim, se o referido filho se encontra cheio. Caso uma dessas condições não seja atendida, `splitChild` apenas sinaliza que sua operação não ocorreu.

```
if (!isFull() && childIndex <= numberKeys && child[childIndex]->isFull()){
  // a reference to the child to be split
  auto& splittingChild = child[childIndex];
  // its new "right" sibling
  std::unique_ptr<Page> newSibling = std::make_unique<Page>();
  // newSibling is a leaf iff splittingChild is a leaf
  newSibling->leaf = splittingChild->leaf;
  newSibling->numberKeys = t - 1;
  for (unsigned int i = 0; i < t - 1; i++){
    newSibling->key[i] = splittingChild->key[t + i];
    newSibling->val[i] = splittingChild->val[t + i];
  }
  // if newSibling is not a leaf, it also receives some
  // children from splittingChild
  if (!newSibling->leaf){
    for (unsigned int i = 0; i < t; i++){
      newSibling->child[i] = std::move(splittingChild->child[t + i]);
    }
  }
  splittingChild->numberKeys = t - 1;
  // we make room for newSibling at position childIndex + 1 ...
  for (unsigned int i = numberKeys; i >= childIndex + 1; i--){
    child[i + 1] = std::move(child[i]);
  }
  // ... and put it right there
  child[childIndex + 1] = std::move(newSibling);
  // now we make room for the median key (and its val) of splittingChild at
  // position childIndex of this node ...
  if (numberKeys > 0){
    for(unsigned int i = numberKeys - 1; i >= childIndex; i--){
      key[i + 1] = key[i];
      val[i + 1] = val[i];
    }
  }
  // ... and put it right there
  key[childIndex] = splittingChild->key[t - 1];
  val[childIndex] = splittingChild->val[t - 1];
  // and this page just gained a new key
  numberKeys++;
  return true;
}
```

Caso suas condições sejam atendidas, `splitChild` cria o novo irmão "direito" de `splittingChild`, `newSibling`. Como ficarão no mesmo nível, `newSibling` é folha sse `splittingChild` é folha.

Após isso, as `2*t - 1` chaves de `splittingChild` (que está cheio) são remanejadas: as `t - 1` menores chaves permanecem em `splittingChild`; as `t - 1` maiores ficam com `newSibling`; a chave mediana fica com seu pai, justamente para fazer a "separação" entre `splittingChild` e `newSibling`. Caso `splittingChild` tenha filhos, os filhos correspondentes às chaves movidas para `newSibling` também são dados a `newSibling`.

Em relação ao pai, note que é preciso posicionar adequadamente tanto a chave mediana recebida de `splittingChild` quanto `newSibling` (entre seus filhos). Para isso, chaves e filhos sofrem um *shift* para a direita (possível porque o pai não está cheio), de forma que a chave mediana de `splittingChild` possa se tornar `key[childIndex]` (`splittingChild` torna-se filho "esquerdo" de sua antiga chave mediana) e `newSibling` possa se tornar `child[childIndex + 1]` (`newSibling` torna-se filho "direito" da antiga chave mediana de `splittingChild`).

```
private:
  // ...
  // root pointer
  std::unique_ptr<Page> root;
public:
  // constructor
  BTree() : root{nullptr}
  {}
  bool isEmpty(){
    return root == nullptr;
  }
  // search method
  std::optional<Val> search(Key key){
    if (root){
      return root->search(key);
    }
    else{
      return {};
    }
  }
  // ...
```

Por enquanto, estamos omitindo um membro privado, mas listamos `root`, o ponteiro para a página raiz de `BTree`. Os membros públicos dessa listagem são bastante triviais.

```
// ...
public:
  // ...
  // insert method
  bool insert(Key key, Val val){
    // if key is present we just signal insertion did not occur
    if (search(key)){
      return false;
    }
    // otherwise we do the insertion
    else{
      // if tree is not empty
      if (root){
        // if root is full ...
        if (root->isFull()){
          // ... we create a new root, ...
          std::unique_ptr<Page> newRoot = std::make_unique<Page>();
          // ... make it the parent of the old root, ...
          newRoot->child[0] = std::move(root);
          // ... update root pointer, ...
          root = std::move(newRoot);
          // and split the old root
          root->splitChild(0);
        }
        // here root is certain to be nonfull, so we make the insertion
        insertOnNonfullPage(root.get(), key, val);
      }
      // if there is no root
      else{
        // we make a new one ...
        root = std::make_unique<Page>();
        // ... which is surely a leaf ...
        root->leaf = true;
        // ... and then we perform the insertion
        insertOnNonfullPage(root.get(), key, val);
      }
      // in either case, insertion took place
      return true;
    }
  }
```

O método `insert`, antes de tudo, verifica se a árvore já contém a chave a ser inserida. Em caso afirmativo, a inserção não ocorre e isso é sinalizado. Caso contrário, verifica-se se a raiz é nula. Nessa situação, a inserção vai criar a página raiz, que certamente não está cheia, tratá-la como folha e usar `insertOnNonfullPage` (a ser apresentada) para realizar a inserção.

Caso já haja uma raiz, verificamos se ela está cheia. Caso esteja, criamos uma nova página como pai da raiz, e a dividimos com o `splitChild` da nova página, que passa a ser nossa nova raiz. Como a nova raiz tem exatamente uma página e `t >= 2`, a nova raiz não está cheia, e após isso fazemos a inserção com `insertOnNonfullPage` na raiz da árvore. Agora apresentamos `insertOnNonfullPage`.

Exercício 17. *Explique por que uma Árvore B tem sua altura aumentada apenas quando sua raiz é dividida.*

```
// inserts key val pair on nonfull page
static void insertOnNonfullPage(Page* page, Key key, Val val){
  // index of "rightest" key
  int i = page->numberKeys - 1;
  // if page is a nonfull leaf, we do the insertion
  if (page->leaf){
    // while we search for the appropriate place to put key and val,
    // we also make room for them
    while (i >= 0 && key < page->key[i]){
      page->key[i + 1] = page->key[i];
      page->val[i + 1] = page->val[i];
      i--;
    }
    // we put key and val into their appropriate places ...
    page->key[i + 1] = key;
    page->val[i + 1] = val;
    // ... and update numberKeys
    page->numberKeys++;
  }
  // if page is not a leaf, we recurse to its appropriate child
  else{
    // looking for the child index to recurse
    while (i >= 0 && key < page->key[i]){
      i--;
    }
    // when we exit the while loop,
    // page->key[i] <= key < page->key[i+1], so insertion will
    // recurse into page->child[i + 1]
    i++;
    // before going down, we verify whether page->child[i] is full.
    // In case it is, we split it ...
    if (page->child[i]->isFull()){
      page->splitChild(i);
      // ... and see if the insertion should take place in the newly
      // created sibling, that is, we see if key is greater than the
      // median key that just went up from page->child[i] to page
      if (key > page->key[i]){
        i++;
      }
    }
    // finally, we perform the recursive insertion
    insertOnNonfullPage(page->child[i].get(), key, val);
  }
}
```

A função `insertOnNonfullPage` trata de forma distinta folhas e não-folhas. Quando recebe uma folha, e essa deve ser não-vazia, apenas é realizado um *shift* para a direita para que a chave a ser inserida seja posta em seu devido lugar, mantendo as chaves da página em ordem crescente.

Quando se trata da inserção em uma não-folha, a operação é feita recursivamente em um de seus filhos. Após determinado em que filho deve ser realizada a inserção, é preciso verificar se ele está cheio. Caso não esteja, a operação recursiva é realizada imediatamente. Mas se estiver cheio, esse filho precisa ser dividido em dois filhos não-cheios. Após isso, é preciso comparar a chave a ser inserida com a chave mediana que acabou de "subir" para o pai, e baseado nessa comparação, decide-se em qual dos dois novos filhos deve ser feita a inserção.

Com a definição de `insertOnNonfullPage`, notamos que novas chaves são sempre inseridas em folhas. As páginas que não são folhas ganham novas chaves apenas quando a chave mediana de um de seus filhos "sobe". É possível observar também que, no caminho da raiz até a folha em que a inserção irá ocorrer, `insertOnNonfullPage` divide todas as páginas cheias que encontra.

Exercício 18. *Qual a complexidade do procedimento de inserção para uma Árvore B de altura h e parâmetro t?*

Agora lidamos com a operação de remoção. Como esta vai ser passada como atividade de laboratório, vamos apenas descrever seu funcionamento, sem a apresentação de código-fonte.

Baseando-se na invariante de que `page` sempre terá mais que `t - 1` chaves (a menos que seja a raiz), podemos descrever o procedimento de remoção com os seguintes casos:

1. estamos removendo `key` de uma `page` não-folha:
   (a) que não contém `key`: considerando `0 <= j < numberKeys`, se `key` é maior que toda `key[j]`, fazemos `i = numberKeys`; se `key` é menor que toda `key[j]`, fazemos `i = 0`; do contrário, fazemos `i` ser tal que `key[i - 1] < key < key[i]`; uma vez definido `i`, a sub-árvore enraizada em `child[i]` pode conter `key`:
      i. `child[i]` tem mais que `t - 1` chaves: recursivamente, removemos `key` de `child[i]`;
      ii. `child[i]` tem `t - 1` chaves:
         A. `child[i]` tem um irmão adjacente (`child[i - 1]` ou `child[i + 1]`) com mais que `t - 1` chaves: caso esse seja `child[i - 1]`, movemos sua maior chave para o lugar de `key[i - 1]`, que se torna a menor chave de `child[i]` (após um *shift* para a "direita" nas chaves e nos filhos de `child[i]`); o filho mais "à direita" de `child[i - 1]` torna-se o filho de índice `0` de `child[i]`; temos um procedimento análogo para `child[i + 1]`;

B. `child[i]` não tem irmãos adjacentes com mais que `t - 1` chaves: vamos supor que `child[i - 1]` é um filho válido de `page`; como `child[i - 1]` e `child[i]` têm `t - 1` chaves cada, fazemos o *merge* de `child[i - 1]`, `key[i - 1]` e `child[i]`, e colocamos a página resultante (com `2t - 1` chaves) em `child[i - 1]`; fazemos um *shift* para a "esquerda" nas chaves de `page`, de `key[i]` em diante, e nos filhos de `page`, de `child[i + 1]` em diante; há um procedimento análogo para `child[i + 1]`, caso `child[i - 1]` não seja um filho válido;

(b) que contém `key` como `key[i]`:

   i. caso `child[i]` tenha mais que `t - 1` chaves: encontramos o predecessor de `key` em `child[i]`, digamos `predKey`; sobrescrevemos `key` com `predKey` e deletamos, recursivamente, `predKey` de `child[i]`;

   ii. caso `child[i + 1]` tenha mais que `t - 1` chaves: análogo ao caso anterior, mas em relação ao sucessor de `key` em `child[i + 1]`;

   iii. caso ambos `child[i]` e `child[i + 1]` tenham `t - 1` chaves: os números de chaves e de filhos de `page` diminuem em 1 quando fazemos o *merge* de `child[i]`, `key[i]` e `child[i + 1]`, criando um novo filho de `page` com `2t - 1` chaves que será `child[i]`; as chaves de `key[i + 1]` em diante sofrem um *shift* para a "esquerda"; agora podemos remover `key` de `child[i]`;

2. estamos removendo `key` de uma `page` folha:

   (a) que não contém `key`: neste caso, sabemos que a árvore não contém `key`, e portanto nenhuma chave é removida;

   (b) que contém `key`: removemos `key` de `page`; caso `page` não seja raiz, a invariante garante que `page` tem agora pelo menos `t - 1` chaves.

Exercício 19. *Qual a complexidade do procedimento de remoção para uma Árvore B de altura h e parâmetro t?*

3 Heaps

Heaps são estruturas de dados que permitem acesso em tempo constante a um certo elemento, considerado o mais prioritário, dentre os elementos armazenados. Nesta seção, descrevemos dois tipos de heaps: as heaps binárias, e as heaps de Fibonacci.

3.1 Heaps Binárias

Heaps binárias são estruturas arbóreas em que cada nó tem no máximo dois filhos. Para cada nó, deve-se obedecer a propriedade de que seu elemento é mais prioritário que os elementos de seus filhos. Com isso, é sabido que o elemento mais prioritário deve se encontrar na raiz da heap.

Exercício 20. *Argumente que o elemento mais prioritário de uma subárvore de heap binária se encontra em sua raiz. Dica: indução na altura.*

Os filhos de um certo nó não precisam obedecer uma relação entre si, mas apenas cada um com seu pai. Assim, podemos representar a heap de forma linear, abstraindo-a como uma árvore bem balanceada, com suas folhas ocorrendo em seus dois últimos níveis.

3.1.1 Implementação

Apresentamos nossa implementação de heap binária, que se encontra em `binary_heap.hpp`.

```
// ensures file is read at most once per compilation unit
#pragma once
// for comparation functions
#include <functional>
// for contiguous memory management
#include <vector>
// A binary max heap assgning a priority to each Element
template<typename Element,
         typename priority_type = unsigned long,
         typename Comparator    = std::greater<priority_type>>
class BinaryMaxHeap{
  // ...
};
// analogous data structure where a lower priority value means higher
// priority
template<typename Element, typename priority_type = unsigned long>
using BinaryMinHeap = BinaryMaxHeap<Element, priority_type, std::less<priority_type>>;
```

Por padrão, nossas heaps vão usar `unsigned long` para representar valores de prioridades. Observe que a diferença entre `BinaryMaxHeap` e `BinaryMinHeap` está em `Comparator`: `BinaryMaxHeap` usa `std::greater` para considerar mais prioritários os elementos com maior valor de prioridade, enquanto `BinaryMinHeap` usa `std::less` e põe seus elementos com menos valor de prioridade como os mais prioritários.

```
class BinaryMaxHeap{
private:
  // static comparator used to determine which elements have more
  // priority
  static constexpr const Comparator cmp_ {};
  // this helps to associate elements to priorities and to write the
  // algorithms
  struct Node{
    // ...
  };
  // type aliases for easy refactoring (if necessary)
  using Data       = std::vector<Node>;
  using index_type = typename Data::size_type;
  // this will hold elements and their priorities
  Data data_;
  // current number of stored elements
  index_type size_;
  // ...
};
```

A classe `BinaryMaxHeap` tem um membro privado estático `cmp_`, um objeto `Comparator` responsável por determinar a ordem de prioridade entre os elementos armazenados em instâncias de `BinaryMaxHeap`. Note o uso de `constexpr` para podermos inicializar um membro estático em tempo de compilação.

Ainda em suas definições privadas, `BinaryMaxHeap` traz `Node`, que ajuda a associar elementos às suas prioridades e encapsula o uso de `cmp_`. Usamos `data_` como a estrutura que armazena objetos `Node` de forma contígua, e tomamos nota do número de elementos armazenados em `size_`.

```
struct Node{
  priority_type priority;
  Element       element;
  // very simple constructor
  Node(priority_type p, Element e) : priority{p}, element{e}
  {}
  // this allows to compare Node values in terms of the instantiated
  // Comparator
  bool operator>(const Node& n){
    return cmp_(priority, n.priority);
  }
};
```

Na definição de `Node`, chama a atenção `operator>`. Tal método vai nos permitir esquecer que `cmp_` existe enquanto implementamos as definições restantes.

```
private:
  // ...
  // calculates parent index of node whose index is i
  static index_type parent_index_(index_type i){
    return (i - 1) / 2;
  }
  // calculates left child index of node whose index is i
  static index_type left_child_index_(index_type i){
    return 2*i + 1;
  }
  // calculates right child index of node whose index is i
  static index_type right_child_index_(index_type i){
    return 2*i + 2;
  }
  // ...
```

Utilizamos os métodos `parent_index_`, `left_child_index_` e `right_child_index_` para abstrair o cálculo dos índices do pai, filho esquerdo e filho direito de um nó, respectivamente. Note que os filhos de um nó são armazenados em posições consecutivas, mas o pai de índice `i` se encontra a uma distância `i` de seu primeiro filho.

Exercício 21. *Argumente que `BinaryMaxHeap` armazena uma árvore binária em `data_`, de forma que nós de um mesmo nível são armazenados consecutivamente, e que níveis são armazenados do mais alto (nível da raiz) para o mais baixo.*

```
private:
  // ...
  void heapify_down_(index_type i){
    // this will hold the index of node with highest priority among i
    // and its children
    index_type largest {i};
    // indices of i children
    index_type left  {left_child_index_(i)};
    index_type right {right_child_index_(i)};
    // determines which of the three nodes has highest priority
    if (left < size_ && data_[left] > data_[largest]){
      largest = left;
    }
    if (right < size_ && data_[right] > data_[largest]){
      largest = right;
    }
    // if a child has more priority than i, swap them
    if (largest != i){
      std::swap(data_[largest], data_[i]);
      // then adjust position of the moved parent, now located at
      // largest, further down
      heapify_down_(largest);
    }
  }
// ...
```

Exercício 22. *Escreve uma versão iterativa de `heapify_down_`.*

O método `heapify_down_` tem o propósito de assegurar que um certo nó de índice `i` é mais prioritário que seus descendentes, dado que seus dois filhos já obedecem essa propriedade. Para isso, sua prioridade é comparada com a de seus filhos, e caso um de seus filhos seja mais prioritário, o pai troca de lugar com o filho mais prioritário e, recursivamente, é comparado com seus novos filhos. Note que, da forma como está escrito, o critério de parada de `heapify_down_` consiste da validade dos índices dos filhos, isto é, se o nó de índice `i` de fato tem filhos, ou se esse é mais prioritário que seus dois filhos.

```
private:
  // ...
  // adjusts position of node at index i according to a comparison
  // with its parent. If it is moved, its position is adjusted
  // iteratively
  void heapify_up_(index_type i){
    // while i is an (upwards) valid index and it has more priority
    // than its parent, move it up
    while (i > 0 && !(data_[parent_index_(i)] > data_[i])){
      std::swap(data_[i], data_[parent_index_(i)]);

      i = parent_index_(i);
    }
  }
```

No sentido contrário ao de `heapify_down_`, `heapify_up_` garante que um nó de índice `i` não é mais prioritário que seus ancestrais, dado que seu pai já obedece essa propriedade. Isso é feito trocando filho e pai de lugar, caso o pai não seja mais prioritário que o filho, e comparando, de forma iterativa, o recém-deslocado filho com seu novo pai. O critério de parada de `heapify_up_` é mais explícito que o de `heapify_down_`.

```
class BinaryMaxHeap{
  // ...
public:
  // basic constructor
  BinaryMaxHeap() : data_{}, size_{0}
  {}
  // returns whether heap is empty
  bool empty(){
    return size_ == 0;
  }
  // ...
};
```

Agora descrevemos a interface pública de `BinaryMaxHeap`. Temos um construtor bem simples, sem argumentos, que inicializa `data_` e `size_` de forma a indicar que a estrutura ainda não possui elementos. O método `empty` indica se a estrutura está vazia ou não.

```
public:
  // ...
  // gets the element with highest priority. Returns nothing in case
  // heap is empty
  std::optional<Element> priority_element(){
    if (empty()){
      return {};
    }
    else{
      return data_[0].element;
    }
  }
  // ...
```

O método `priority_element` retorna o elemento mais prioritário da estrutura. Como esse se encontra na raiz da heap, retornamos o elemento armazenado na posição de índice `0`. Caso a heap esteja vazia, retornamos um `optional` vazio.

```
public:
  // ...
  // extracts the element with highest priority. Does nothing in case
  // heap is empty
  std::optional<Element> extract(){
    if (empty()){
      return {};
    }
    else{
      // moves element at position 0, since it is about to be
      // overwritten
      Element moved {std::move(data_[0].element)};
      // overwrites heap root with the leaf of maximum valid index
      data_[0] = std::move(data_[size_ - 1]);
      // accounts for removal of one element
      --size_;
      // adjusts position of new root
      heapify_down_(0);
      // returns extracted element
      return moved;
    }
  }
  // ...
```

Em `extract`, removemos e retornamos o elemento mais prioritário. Para tanto, salvamos o elemento mais prioritário e, em seguida, sobrescrevemos a raiz da heap com a folha de maior índice. Para garantir as propriedades da heap, ajustamos a posição da nova raiz com `heapify_down_`.

```
public:
  // ...
  // inserts element with priority
  void insert(priority_type priority, const Element& element){
    // data_ might be using all its allocated space, so use push_back
    // to ensure reallocation (if needed)
    if (size_ == data_.size()){
      data_.push_back({priority, element});
    }
    // otherwise simply put element and its priority as the leaf with
    // maximum valid index
    else{
      data_[size_] = {priority, element};
    }
    // accounts for insertion of element
    ++size_;
    // adjusts position of newly inserted element
    heapify_up_(size_ - 1);
  }
```

Durante a inserção, precisamos levar em conta se será necessário realocar a região de memória que armazena nossos elementos. Isso pode ser necessário apenas quando `size_` é igual a `data_.size()`, e nesse caso delegamos a inserção a `push_back`, que é capaz de realocar `data_` caso seja necessário. Uma vez inserido o novo elemento e sua prioridade como a folha de maior índice, usamos `heapify_up_` para assegurar as propriedades de heap.

Exercício 23. *É possível implementar uma fila em termos de uma heap? E quanto a uma pilha?*

3.1.2 Complexidade

As operações de inserção e remoção têm sua complexidade dominada por `heapify_up_` e `heapify_down_`, respectivamente. Como essas funções fazem uma quantidade constante de comparações e `swap` por iteração ou chamada, sua complexidade é proporcional ao número de iterações ou chamadas que podem ser realizadas, e esse é limitado pela altura da árvore binária que representa nossa heap. Como organizamos os nós linearmente de forma que a árvore esteja sempre bem balanceada, inserção e remoção em uma heap de n elementos custam $O(\log n)$. O acesso ao elemento mais prioritário, que tem sua localização bem determinada pela disposição dos nós, é feito em $O(1)$.

3.2 Heaps de Fibonacci

TODO escrever na próxima versão da disciplina.

4 Conjuntos Disjuntos

Em certos contextos, é desejável representar uma relação entre um grupo de elementos de forma que eles fiquem particionados em conjuntos disjuntos. Nesses contextos, duas operações costumam ser importantes: determinar a que conjunto disjunto pertence um certo elemento, e unir dois conjuntos em um novo conjunto disjunto de forma eficiente.

Vamos implementar o agrupamento de n elementos em conjuntos disjuntos como uma associação indireta entre cada elemento e seu único representante. Dessa forma, os elementos de um conjunto disjunto podem ser caracterizados por terem um mesmo representante.

Se um elemento está associado a si mesmo, então ele é seu próprio representante. Caso contrário, ele e seu elemento associado têm o mesmo representante. Inicialmente, teremos n conjuntos disjuntos unitários, ou seja, cada elemento será representado por si mesmo.

4.1 Implementação

Vamos criar nossa implementação de conjuntos disjuntos em um arquivo chamado `disjoint_sets.hpp`. Seu conteúdo é descrito a seguir.

```
#pragma once
// we will use a vector to create an association between elements
#include <vector>
// A disjoint set implementation: each of n elements is uniquely
// assigned to a index between 0 and n - 1, and is initially
// represented by itself. Elements with a common representative can
// therefore be seen as disjoint from other groups of elements. This
// way, each element starts in a unitary disjoint set
class DisjointSet{
public:
  // alias used as index type
  using size_type = unsigned long;
private:
  // alias for our data representation: each representative is given
  // by its index, so size_type seems a reasonable choice
  using Data = std::vector<size_type>;
  // this stores representative of element i in position of index i
  Data representative_;
public:
  // ...
};
```

Em nossa classe `DisjointSet`, cada um dos `n` elementos é identificado por um índice entre `0` e `n - 1`. Dada essa escolha, usamos `unsigned long` para armazenar os índices de nossos elementos, e `Data` como um alias para um `vector` de `unsigned long`, que é utilizado para armazenar, em sua posição de índice `0 <= i <= n`, o índice do elemento associado a `i`.

```
public:
  // constructor of DisjointSet with number_elements initial elements
  DisjointSet(size_type number_elements) : representative_{}
  {
    // allocates space for number_elements representative indices
    representative_.resize(number_elements);
    // initially, each element is represented by itself
    for (size_type i {0}; i < number_elements; ++i){
      representative_[i] = i;
    }
  }
  // returns number of elements currently being represented by DisjointSet
  size_type size() const{
    return representative_.size();
  }
  // ...
```

O construtor de `DisjointSet` tem como argumento o número inicial de elementos, e é responsável por alocar memória para tais elementos e fazer com que sejam representados por si mesmos. Já o método `size` apenas retorna o número de elementos que a estrutura está representando.

```
// returns index of representative of element
size_type representative(size_type element) const{
  // if element is represented by itself, returns its own index
  if (representative_[element] == element){
    return element;
  }
  // otherwise, goes recursively in search of a representative
  else{
    return representative(representative_[element]);
  }
}
```

O método `representative` retorna o índice do representante de `element`, e sua implementação segue nossa definição. Se `element` está associado a si mesmo, ele é seu próprio representante e portanto é retornado. Caso contrário, ele e seu elemento associado têm o mesmo representate, logo devemos retornar, recursivamente, o representante de `representative_[element]`.

```
// adds a new element index to DisjointSet
size_type new_element(){
  // gets number of elements currently being represented by
  // DisjointSet
  size_type number_elements {representative_.size()};
  // adds a new element (with number_elements as index) being
  // represented by itself
  representative_.push_back(number_elements);
  // returns index of newly represented element
  return number_elements;
}
```

A finalidade de `new_element` é fazer com que nossa estrutura passe a representar um novo elemento. Para isso, criamos uma posição em `representative_` cujo índice é exatamente igual ao número de elementos anteriormente representados, e fazemos com que seu conteúdo seja seu próprio índice. Tal índice é retornado, permitindo ao usuário da classe se referir ao novo elemento por seu índice.

```
// joins two distinct groups of elements: if element_a and element_b
// are represented by distinct representatives, representative of
// element_b starts to represent all elements previously represented
// by the representative of element_a. Returns true if a join has
// occured
bool join(size_type element_a, size_type element_b){
  // gets the representative of each element
  size_type representative_a {representative(element_a)};
  size_type representative_b {representative(element_b)};
  // if they are the same, no join needs to occur, and this is
  // signaled to the caller
  if (representative_a == representative_b){
    return false;
  }
  // otherwise, representative_a becomes represented by
  // representative_b, and as a consequence, each element formerly
  // represented by representative_a is now represneted by
  // representative_b
  else{
    representative_[representative_a] = representative_b;
    // a join has occured, so returns true
    return true;
  }
}
```

Em `join`, temos o objetivo de fazer com que os conjuntos disjuntos de `element_a` e `element_b` sejam unidos, caso não sejam o mesmo. Primeiro, tomamos os representantes de `element_a` e `element_b`. Caso sejam o mesmo, nada é feito e isso é sinalizado no retorno. Caso contrário, fazemos com que `representative_a` passe a ser represnetado por `representative_b`, e portanto os elementos dos conjuntos de `element_a` e `element_b` passam a ter o mesno representante, e estão agora no mesmo conjunto, ainda disjunto dos demais.

4.2 Complexidade

Em nossa estrutura de conjuntos disjuntos, determinar o representante de um elemento depende de passear por uma série de associações. Como um elemento é indiretamente associado a seu representante, pode ser que todos os n elementos tenham o mesmo representante e estejam associados de forma linear. Nesse cenário, que é o de pior caso, determinar o representante de um certo elemento custa $O(n)$.

A complexidade de nossa outra operação não é diferente. Note que, apesar da união de dois conjuntos disjuntos poder ser feita com uma única mudança em uma associação, o que é feito em tempo constante, precisamos primeiro determinar os representantes de dois elementos, e isso tem um custo linear no número de elementos.

No entanto, a implementação aqui apresentada não é ótima. Nas atividades de laboratório referentes a conjuntos disjuntos, iremos implementar técnicas que melhoram significativamente os custos de nossas operações: união por *rank* e compressão de caminhos.

5 Grafos

Grafos são construções matemáticas que expressam relações entre pares de elementos de um certo conjunto. Formalmente, o grafo $G = (V, E)$ é determinado por um par de conjuntos: V é um conjunto finito contendo os vértices de G, enquanto $E \subseteq \binom{V}{2}$ é um conjunto de pares não ordenados de elementos de V, e cada um desses pares é dito uma aresta de G. As arestas de um grafo expressam relações entre seus vértices. Por exemplo, os vértices podem representar pessoas e uma aresta entre dois vértices pode indicar que as pessoas correspondentes se conhecem.

No parágrafo anterior, é utilizada a notação $\binom{V}{2}$, que representa todos os subconjuntos de V com exatamente dois elementos. Formalmente, $\binom{V}{2} = \{S \subseteq V : |S| = 2\}$ e, de maneira geral, $\binom{C}{k} = \{S \subseteq C : |S| = k\}$.

É possível que as relações entre os vértices sejam representadas por pares ordenados. Nesse caso, estamos lidando com grafos direcionados, ou digrafos, e os pares ordenados são chamados de arcos, tendo um vértice como origem e um vértice como destino. Em escrita formal, um digrafo $D = (V, A)$ tem um conjunto finito de vértices V e um conjunto de arcos $A \subseteq V \times V = V^2$.

Aqui, vamos implementar digrafos e grafos, nessa ordem. Para tanto, vamos entender como representá-los.

5.1 Representação

Um digrafo $D = (V, A)$ pode ser representado de duas formas principais: utilizando listas de adjacência ou uma matriz de adjacência. Em uma representação por listas de adjacência, a cada vértice $v \in V$ é atribuida uma lista encadeada simples contendo os vértices atingíveis a partir de v, ou seja, os vértices do conjunto $N^+(v) = \{u \in V : (v, u) \in A\}$. Já em uma representação por matriz de adjacência, o conjunto A é representado por uma matriz $M \in \{0, 1\}^{|V| \times |V|}$, em que $M_{uv} = 1$ se e somente se $(u, v) \in A$. Note que, na segunda representação, admitimos que $V = \{1 \ldots n\}$. Com pouca adaptação, tais representações também podem ser usadas para grafos.

Listas encadeadas são eficientes, em termos de espaço, para representar grafos (ou digrafos) esparsos. Por outro lado, matrizes de adjacência permitem verificar, em tempo constante, se um arco (ou aresta) existe em um digrafo (ou grafo).

Nestas notas de aula, vamos fazer nossa implementação de digrafos baseada em matrizes de adjacência. Além disso, vamos fazer seu reuso para nossa implementação de grafos.

5.2 Implementação

6 Laboratórios

Nesta seção, descrevemos atividades práticas que devem ser desenvolvidas em aulas de laboratório. Idealmente, deve haver uma subseção para cada estrutura de dados descrita nestas notas de aula.

6.1 Árvores Binárias de Busca

Neste laboratório, vamos desenvolver atividades majoritariamente relacionadas com passeios em Árvores Binárias de Busca.

1. Implemente os seguintes métodos na classe `BSTree` como públicos:

   (a) `static std::vector<std::pair<Key, Val>> inOrder(const BSTree<Key, Val>& bst)`

   (b) `static std::vector<std::pair<Key, Val>> preOrder(const BSTree<Key, Val>& bst)`

   (c) `static std::vector<std::pair<Key, Val>> postOrder(const BSTree<Key, Val>& bst)`

   Lembramos que, como esses métodos não devem alterar a `BSTree` passada como argumento, todos eles receben uma referência `const` para `BSTree`. Caso tivéssemos, por exemplo, um objeto `bst` de tipo `BSTree<int, char>`, faríamos a chamada `BSTree<int, char>::inOrder(bst)`.

2. Implemente o método `bool update(Key key, Val newVal)` como público em `BSTree`. O método dever retornar `false` sse `key` não está presente na árvore.

3. Escreva testes em `bstree_test.cpp` para assegurar que a implementação de seus métodos está correta.

4. Escreva um programa em `lab1.cpp` que, dado o nome de um arquivo, imprime no terminal o número de ocorrências de palavras no arquivo (se a palavra "gato" ocorre três vezes, o programa deve imprimir a linha `gato:  3` ou algo próximo disso). O programa deve imprimir as palavras em ordem alfabética. Dica: use uma `BSTree<std::string, int>`.

5. Por fim, crie uma "receita" em `makefile` de forma que o executável `lab1` possa ser construído com o comando `make lab1`.

6.2 Árvores AVL

Neste laboratório, nosso objetivo é completar a implementação de `AVLTree`, conforme as indicações feitas na Subseção 2.2. Além disso, vamos escrever testes para garantir que nossa implementação está correta.

1. Existe uma rotação já implementada, `rotateR` (rotação simples à direita). Implemente as outras três rotações (`rotateL`, `rotateLR` e `rotateRL`), também como métodos estáticos privados de `AVLTree`. As rotações duplas poderiam ser definidas em termos das rotações simples?

2. Complete a implementação de `rebalanceNode`, utilizando chamadas para as rotações nos casos adequados.

3. Complete a implementação de `insert`, utilizando `pathToExistingKey` (herdado de `BSTreeWithNode`), `updateHeightsOnPath` e `rebalanceNodesOnPath`. Lembre-se que os nós que precisam ser verificados formam um caminho da raiz até o nó recém-inserido.

4. Faça o mesmo para `remove`. Dessa vez, o caminho de nós que precisa de manutenção vai da raiz até o pai do nó excluído.

5. Acrescente um método público `isBalanced` a `AVLTreeWithNode`, que existe apenas quando `checkBalance` está definido. Para isso, use as diretivas de pré-processamento `#ifdef` e `#endif`. O objetivo de `isBalanced` é verificar se cada *node* satisfaz $balance(node) \in \{-1, 0, 1\}$. Não se esqueça de criar um método público `isBalanced` em `AVLTree` (que existe apenas com `checkBalance` definido) que redireciona sua chamada para o `isBalanced` de `avlt`.

6. Agora que todos os métodos estão completos e que temos ferramentas para teste, escreva mais testes em `avltree_test.cpp`, de forma a garantir que sua implementação de fato mantém a árvore bem balanceada. Use `#define checkBalance` (antes de incluir `avltree.hpp`) em `avltree_test.cpp` para tornar `isBalanced` disponível, e chame-o em um `assert` após cada operação de modificação em seus testes.

6.3 Árvores Rubro-Negras

O objetivo deste laboratório é completar o código-fonte de `RBTree`, seguindo as indicações feitas em `rbtree.hpp`. Algumas funções auxiliares serão sugeridas, tanto para facilitar a implementação como para a realização de testes.

1. Vamos tratar `nullptr` como nossas folhas pretas sem dados significativos. Em `RBTree`, implemente o método privado `static Color color(const RBTree* node)`, que deve retornar `Color::black` quando `node` é `nullptr`, e `node->color` caso contrário.

2. Implemente as funções de rotação simples como métodos privados de `RBTRee`:

   - `static void rotateR(RBTreeNode* node);`
   - `static void rotateL(RBTreeNode* node);`

   É possível basear-se nas rotações simples de `AVLTree`.

3. Implemente a função `static void insertionMaintenance(RBTreeNode* node)` como método privado de `RBTreeWithNode`. Essa função deve sempre ser chamada com nós vermelhos, já que a manutenção é sempre feita para nós vermelhos. O propósito dessa função é fazer as trocas de cor e rotações necessárias para manter as propriedades de um nó de `RBTree` após uma inserção. O método `pathToExistingKey` pode ser útil para essa função.

4. Complete a implementação de `insert` com a operação de manutenção para o nó recém-inserido. O que deve ser feito para tomar um ponteiro para esse nó?

5. Acrescente o método público `bool obeysProperties()` a `RBTreeWithNode` (e também a `RBTree`), que deve existir apenas quando `checkProperties` estiver definido. O objetivo de `obeysProperties` é verficar se `RBTree` satisfaz as propriedades de uma Árvore Rubro-Negra.

6. Acrescente mais testes a `rbtree_test.cpp`, de forma a garantir que sua implementação de `RBTree` está correta. Use `#define checkProperties` (antes de incluir `rbtree.hpp`) de forma a tornar `obeysProperties` disponível. Após cada operação de modificação em seus testes, chame `obeysProperties` em um `assert`.

6.4 Árvores B

A finalidade deste laboratório é completar o código-fonte de `BTree`, seguindo as indicações feitas em `btree.hpp`. A principal atividade aqui proposta é a implementação do método `remove` de `BTree`, a ser seguida pela realização de testes que assegurem o funcionamento correto de `BTree`.

1. Implemente os métodos `rotateKeysL` e `rotateKeysR` de `Page`. Esses métodos estão relacionados com o caso 1(a)iiA da remoção.

2. Implemente o método `mergeSiblingsWithKey` de `Page`. Note que esse método é utilizado nos casos 1(a)iiB e 1(b)iii da remoção.

3. Implemente o método `remove` de `BTree`. As funções dos pontos anteriores podem ser úteis nessa implementação.

4. Em `btree_test.cpp`, acrescente chamadas de inserção, busca e remoção. Após cada operação de modificação, verifique (com `assert`) se as chaves contidas na árvore estão associadas aos seus devidos valores.

6.5 Heaps Binárias

Neste laboratório, vamos utilizar `BinaryMaxHeap` e `BinaryMinHeap` para construir `BinaryMedianHeap`, uma heap que permite acesso em tempo constante, dentre seus elementos, ao elemento mediano. Lembramos que, dados n elementos ordenados e indexados de 1 a n, o elemento mediano é aquele de índice $\lceil \frac{n}{2} \rceil$, ou seja, é menor que cerca de metade dos outros elementos, e maior que a outra metade. Implemente `BinaryMedianHeap` em `binary_median_heap.hpp`.

1. Implemente um método público `index_type size()` em `BinaryMaxHeap`, que retorna o número de elementos armazenados na heap.

2. Em `BinaryMaxHeap`, adapte `extract` e `priority_element` para retornar `std::optional<std::pair<priority_type, Element»`, isto é, retornar o elemento mais prioritário junto com seu valor de prioridade.

3. Crie a classe `BinaryMedianHeap`, com `typename Element` e `typename priority_type` como seus parâmetros de template. O parâmetro `priority_type` deve ser, por padrão, o tipo `unsigned long`.

4. Crie dois membros privados em `BinaryMedianHeap`, uma `BinaryMaxHeap` e uma `BinaryMinHeap`, ambas utilizando os mesmos parâmetros de `BinaryMedianHeap`, cada uma utilizando seu parâmetro `Comparator` padrão.

5. Crie um construtor para inicializar `BinaryMedianHeap` com nenhum elemento. Lembre-se que o construtor deve inicializar os membros privados da classe.

6. Crie um método `std::optional<Element> median_element()`, que deve dar acesso ao elemento mediano, de acordo com os valores de prioridade. Vamos criar os métodos de inserção e remoção de forma que o elemento mediano seja sempre o elemento mais prioritário de `BinaryMaxHeap`.

7. Crie o método de inserção com as seguintes propriedades:

   - a inserção sempre é feita em `BinaryMaxHeap`;
   - `BinaryMaxHeap` tem no máximo um elemento a mais que `BinaryMinHeap`.

8. Crie o método de remoção com as seguintes propriedades:

   - a remoção sempre é feita de `BinaryMaxHeap`;
   - `BinaryMaxHeap` não tem menos elementos que `BinaryMinHeap`.

9. Crie um arquivo `binary_median_heap_test.cpp`, e nele escreva testes para garantir que sua implementação de `BinaryMedianHeap` está correta.

6.6 Heaps de Fibonacci

TODO escrever na próxima versão da disciplina.

6.7 Conjuntos Disjuntos

Neste laboratório, vamos aprimorar nossa implementação de conjuntos disjuntos com duas técnicas: compressão de caminhos e união por rank. Para tanto, resolva as seguintes atividades.

1. Crie um novo membro em `DisjointSet` para armazenar os ranks dos elementos. Inicialmente, todo elemento tem rank zero. Observe que isso implica em fazer modificações no construtor de `DisjointSet` e no método `new_element`.

2. Observe que o método `representative` poderia atualizar o elemento associado a `element` antes de retornar, o que faria com que `element` ficasse diretamente associado a seu elemento representante. Essa técnica é conhecida como compressão de caminho. Implemente a compressão de caminho em `representative`: basta salvar o retorno da chamada recursiva de `representative` em uma varável, usá-la para atualizar o elemento associado a element, e em seguida retorná-la.

3. Por fim, vamos implementar união por rank. Primeiro, observe que o único método que altera os conjuntos é `join`, que sempre prefere fazer `representative_a` passar a ser representado por `representative_b`. Vamos fazer `join` utilizar o rank como um critério para determinar se `representative_a` deve passar a ser representado por `representative_b` ou o contrário: dentre os dois, o elemento de maior rank deve passar a ser o representante do elemento de menor rank; caso ambos tenham o mesmo rank, não importa qual dos dois tornar-se-á representante do outro, mas esse deve ter seu rank aumentado em uma unidade.