Estruturas de Dados

Luiz Alberto do Carmo Viana

March 3, 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
```

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.

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`.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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.

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.

2.2.1 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:

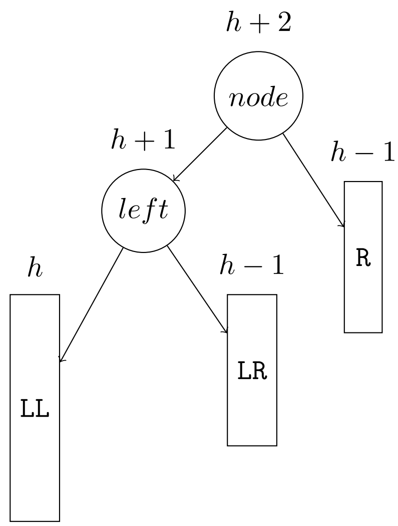
1. $balance(node) = -2$ e $balance(node.left) = -1$;
2. $balance(node) = -2$ e $balance(node.left) \in \{0, 1\}$;
3. $balance(node) = 2$ e $balance(node.right) = 1$
4. $balance(node) = 2$ e $balance(node.right) \in \{-1, 0\}$

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*.

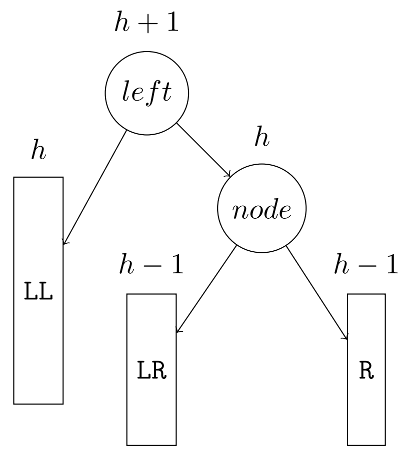
Para resolver o caso 1, usamos uma rotação simples à direita. A Figura 1 ilustra o procedimento dessa rotação. Note que trata-se apenas da atualização do conteúdo de dois nós, e de algumas atualizações de referências. Assim, essa rotação pode ser executada em tempo constante, e após feita, é necessário atualizar as alturas de *node* e $left$.

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 $O(n \log n)$.*



(a) Antes da rotação



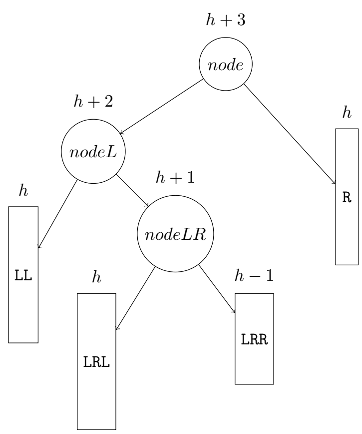
(b) Depois da rotação

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

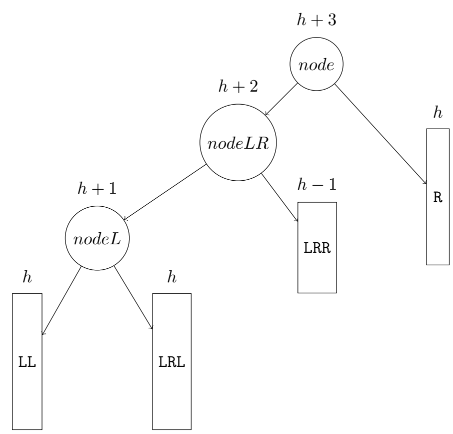
Exercício 8 *Em uma árvore binária, um nó é dito filho único sse ele tem pai e seu nó pai tem exatamente um filho. Vamos definir a* razão de solidão *de uma árvore binária T ($RS(T)$) como o número de filhos únicos em T dividido pelo número de nós de T. Prove que, se T é uma Árvore AVL, então $RS(T)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

2.2.2 Implementação

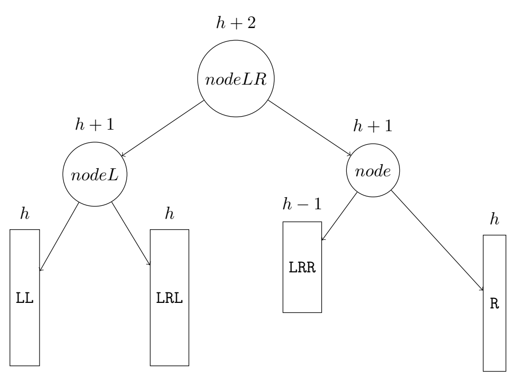
Vamos mostrar uma abordagem de implementação de Árvores AVL utilizando herança a partir de `BSTree`. Para isso, o código de `BSTree` precisou sofrer alguma refatoração, e assim permitir um maior reuso de seus métodos. Vamos explicar a refatoração realizada sempre que for necessário.



(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.

```cpp
#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.

```cpp
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`.

```cpp
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`.

```cpp
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 o mesmo tipo de nó. 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){
      // ...
    }
  };
  // ...
```

3 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.

3.1 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`.