DATA STRUCTURE VISUALIZER

Ana Caroline M. Brito[1], Fernanda Isabel[1], João Pedro Holanda[1], Leonardo Leibovitz[1]

[1] Universidade Federal do Rio Grande do Norte (UFRN)
Instituto Metópole Digital - IMD

a_caroline96@hotmail.com, {feisabel96, jpholanda.prf, leibovitz1}@gmail.com

Abstract. *The Data Structure Visualizer has the didactic finality of making the understanding of the data structures seen in Basic Data Structures I/II easier. This document presents the main points of our chosen approach to this proposal; the main algorithms used to implement and draw the structures are explained in a high level manner. The code was entirely written in Java language.*

Resumo. *O Data Structure Visualizer é um visualizador de estruturas de dados com finalidade didática de facilitar no entendimento do funcionamento das estruturas de dados vistas nas disciplinas de Estruturas de Dados Básicas I/II. Este documento apresenta os principais pontos da abordagem da solução dessa proposta, sendo explicado em alto nível os principais algoritmos utilizados para a implementação das estruturas bem como para o desenho de cada uma delas. Foi utilizada a linguagem Java.*

1. Introdução

O estudo das estruturas de dados tem como maior obstáculo a abstração, a capacidade de enxergar algo palpável a partir de preceitos matemáticos. Professores costumam fazer uso de desenhos em apresentações de *slides.* Normalmente, são os próprios professores que fazem os desenhos de suas estruturas para as aulas. A proposta é que o *Data Structure Visualizer* (*DSV*) possa servir de ferramenta para professores e alunos na hora da construção do conhecimento.

O *DSV* é capaz de representar graficamente uma gama de estruturas de dados de diferentes complexidades de implementação. Ele permite o uso de métodos de típico de cada estrutura e faz a atualização da representação para permitir que o usuário veja a diferença entre o antes e o depois. Também é possível visualizar e controlar diferentes estruturas ao mesmo tempo, por meio de abas. Além disso, o *DSV* também é capaz de salvar estruturas de dados previamente construídas pelo usuário e de carregá-las novamente.

Esse *software* é de interesse tanto do professor quanto do aluno. Ao mesmo tempo que o professor pode usá-lo para ensinar e explicar implementações e funcionalidades, o aluno pode conferir suas respostas e até comparar com seu próprio código funcionando.

2. Descrição do problema abordado

A proposta do *DSV* é fazer um visualizador de estruturas de dados em que o usuário possa inserir, remover e pesquisar e ver a estrutura antes e depois da operação. Dessa forma as duas partes centrais do problema são a implementação das estruturas de dados e o desenho de cada uma delas. Um problema secundário foi o de construir um sistema de comunicação GUI-estruturas-atualização.

3. Descrição das estruturas/abordagem da solução do problema/algoritmos utilizados

A estruturas de dados utilizadas foram: árvore Binária de Busca, árvore AVL (balanceada por altura), árvore Rubro Negra, Conjuntos Disjuntos, *HeapMax* e *HeapMin*. Para cada estrutura foi desenvolvido um algoritmo de desenho diferente.

Para a árvore Binária de Busca, a AVL e a Rubro Negra foi desenvolvido um sistema de desenho em que os nós são acessados em pré ordem, portanto o pai é desenhado para depois os filhos serem desenhados e as ligações (arestas) serem criadas. A indicação do balanceamento da AVL foi feito por meio das cores dos nós, e a árvore rubro negra é desenhada com seus nós pretos e vermelhos. Cada nó tem um campo que guarda a sua cor, portanto foi possível que a mesma função fosse usada para ela e a binária de busca. Porém, no caso da rubro negra houve um acréscimo para que os nós externos aparecessem.

Os métodos de desenho das seguintes estruturas, *UnionFind* (conjuntos disjuntos), *HeapMax*, *HeapMin* foram implementados utilizando algoritmos iterativos que se baseiam em percorrer as estruturas por nível e desenhá-las a partir das informações de x e y (posição).

As estruturas *Deque*, *List*, *Queue*, *Stack* não foram implementadas, pois não estava dentro do escopo de Estruturas de Dados 2, foram implementados apenas sistemas de desenho, baseados na abordagem iterativa de desenhar o nó seguinte baseado na posição do anterior e tomando cuidado para não exceder as extremidades.

Para auxiliar na construção das estruturas graficamente foi utilizada a biblioteca externa JGraph, através dela foi possível estabelecer um padrão de criação de vértices e arestas para facilitar o desenvolvimento dessa tarefa.

A seguir estão as seções com o pseudo código das estruturas implementadas.

3.1. Árvore Binária de Busca

Uma árvore binária de busca (*BST*) consiste em uma árvore onde cada nó possui no máximo dois filhos, há uma ordenação natural entre as chaves dos nós e cada subárvore de um mesmo nó não possui interseção.

3.1.1. Busca e Inserção

A ideia geral da busca em uma árvore binária de busca consiste em comparar a partir da raiz o valor buscado com o valor armazenado no nó, para então decidir para qual dos lados deve continuar a busca, se o valor buscado for maior que a chave do nó atual, então chama recursivamente a busca para o filho direito do nó, caso contrário chama a busca para o filho esquerdo do nó.

Além disso, na nossa implementação há o *bool insert* que funciona possibilitando que a busca funcione como inserção, quando ele é verdadeiro, ao encontrar um nó vazio é

inserido um novo nó com a chave do elemento procurado, caso contrário apenas é retornado o nó procurado. A complexidade da operação busca/inserção é em função da altura da árvore (número de nós que foram acessados), portanto, na melhor situação, será log *n*, e no pior caso será *n*, sendo *n* o número de nós na árvore.

```
buscaPrivada(nó, key, insert)
      se nó = null então
            se árvore vazia && insert então
                  insere nó na raiz
                  retorna raiz
            retorna null
      senão se a chave do nó = key então
            retorna nó
      senão se key do nó > key então
            se filho esquerdo do nó = null && insert então
                  cria novo nó no filho esquerdo do nó atual
                  retorna filho esquerdo
            retorna buscaPrivada(filho esquerdo do nó, key, insert)
      senão
            se filho direito do nó = null && insert
                  cria novo nó no filho direito do nó atual
                  retorna filho direito
            retorna buscaPrivada(filho direito do nó, key, insert)
```

3.1.2. Remoção

A remoção da binária consistem em pesquisar o node que deve ser removido e caso encontrado, este será substituido pelo antecessor. A complexidade da remoção depende da altura da árvore, no melhor caso é log(*n*), sendo *n* o número de nós da árvore.

```
delete(chave)
      nó q = search(chave)
      se q /= null então
            se filho esquerdo de q = null &&
            filho direito de q = null então
                  remove(q, null)
            senão se filho esquerdo de q /= null &&
            filho direito de q = null então
                  remove(q, filho esquerdo q)
            senão se filho esquerdo de q = null &&
            filho direito de q /= null então
                  remove(q, filho direito q)
            senão
                  troca q e antecessor de q
            retorna q


remove(nó, filho)
      se raiz = nó então
            raiz = nó atual
      se nó /= null então
            pai = pai do nó atual
            se pai /= null então
                   se filho esquerdo de pai = nó então
```

```
                        filho esquerdo de pai = filho
                  senão
                        filho direito de pai = filho
            se filho /= null então
                  pai de filho = pai
```

3.1.3. Draw

O algoritmo para desenhar a *BST* é baseado no acesso em pré ordem, isto é, primeiro a ser acessado é o pai e posteriormente os filhos. Dessa forma, o vértice correspondente ao pai é criado e depois, no momento da criação dos filhos, é criada a aresta que liga o pai ao filho. A complexidade é $\Theta(n)$, sendo n o número de nós da árvore, pois todos os nós são percorridos apenas uma vez.

O cálculo da posição x do filho esquerdo é levando em consideração que *x* deve ser o (valor do *x* do pai) - (largura do *panel* /(2^nível do pai + 1)). Já a posição do *x* do filho direito é (valor do *x* do pai) + (largura do *panel* /(2^nível do pai + 1)).

```
preOrdem(mapa com vértices, nó, x, y, cor)
      se raiz /= null então
            criar vértice com coordenada x, y e cor
                  armazenada no nó
            se filho esquerdo da raiz /= null
                  preOrdem(mapa, filho esquerdo, x recalculado,
                              y recalculado, cor do nó)
                  insere aresta de pai para filho
            se filho direito da raiz /= null
                  preOrdem(mapa, filho esquerdo, x recalculado,
                              y recalculado, cor do nó)
                  insere aresta de pai para filho
```

3.2. AVL

Uma árvore AVL ou balanceada por altura é uma árvore binária de busca que mantem sempre a diferença entre as subárvores de um nó com módulo no máximo igual a 1. Por ser binária de busca o algoritmo de busca é o mesmo utilizado pela *BST*. Na implementação, AVL é filha da *BST*.

3.2.1. Inserção

A inserção da AVL percorre a árvore similarmente ao algoritmo de busca da BST, porém além disso, ele aproveita a volta da recursão para verificar o balanço da árvore e se há a necessidade de fazer rebalanceamento. Quando um nó é inserido à esquerda, se o balanço anterior era 1 ou 0 só é necessário atualizar o balanço, porém se for -1 então é necessário fazer operações de rotação.

Primeiro serão definidas as operações de rotação simples à direita e dupla à direita, para que haja compreensão do algoritmo geral da inserção da AVL, não serão feitos pseudos códigos das rotações à esquerda, pois estes são análogos aos à direita. Além disso, esses algoritmos de rotação serão citados outras vezes mais à frente.

3.2.1.1. Rotações à direita

As operações de rotação consistem em alterações na árvore que modificam a altura sem fazer com que a árvore perca suas propriedades.

```
rotacoesDireita(nó)
      nó esq = filho esquerdo do nó atual
      se balanço de esq = -1 então
            faz rotação à direita(nó, esq)
            balanço de nó = 0
            balanço de esq = 0
            se raiz = nó então
                  raiz = esq
      senão
            nó dir = filho direito do nó atual
            faz rotação dupla à direita(nó, esq, dir)
            se balanço de dir = -1 então
                  balanço de nó = 1
            senão
                  balanço de nó = 0
            se balanço de dir = 1 então
                  balanço de esq = -1
            senão
                  balanço de esq = 0
            balanço da dir = 0
            se raiz = nó então
                  raiz = dir


rotacaoDireita(nó, esq)
      pai do esq passa a ser pai do nó atual
      se pai de esq /= null então
            se chave de nó < chave do pai da esq então
                  filho esquerdo do pai de esq passa a ser esq
            senão
                  filho direito do pai de esq passa a ser esq
      filho esquerdo de nó passa a ser filho direito de esq
      se filho esquerdo de nó /= null então
            pai do filho esquerdo de nó passa a ser nó
      filho direito de esq passa a ser nó
      pai de nó passa a ser esq

rotacaoDuplaDireita(nó, esq, dir)
      pai de dir passa a ser o pai de nó
      se pai de dir /= null então
            se chave de nó < chave do pai de dir então
                  filho esquerdo do pai de dir passa a ser dir
            senão
                  filho direito do pai de dir passa a ser dir
      filho direito de esq passa a ser filho esquerdo de dir
      se filho direito de esq /= null então
            pai do filho direito de esq passa a ser esq
      se filho esquerdo de nó /= null então
            pai do filho esquerdo de nó passa a ser nó
      filho esquerdo de dir passar a ser esq
      pai de esq passa a ser dir
      filho direito de dir passa a ser nó
```

```
pai de nó passa a ser dir
```

Agora podemos definir a inserção da AVL. A complexidade dessa operação depende da altura da árvore que é sempre log(*n*), já que esta é quantidade de acessos a nós, um por nível, pois AVLs são balanceadas.

```
insercao(chave, nó, pai, ref_b)
      se nó = null então
            cria novo nó(pai, null, null, chave, 0)
            se raiz = null então
                  raiz = novo nó
            senão se chave < chave do pai então
                  filho esquerdo do pai passa a ser novo nó
            senão
                  filho direito do pai passa a ser novo nó
            ref_b = true
      senão
            se chave = chave do nó então
                  ref_b = falso
                  retorna
            senão chave < chave do nó então
                  insercao(chave, filho esquerdo
                        do nó, nó, ref_b)
                  se ref_b então
                        se balanço do nó = 1 então
                              balanço do nó = 0
                              ref_b = falso
                        senão se balanço de nó = 0
                              balanço de nó = -1
                        senão
                              fazer rotacoesDireita(nó)
                              ref_b = falso
            senão
                  insercao(chave, filho direito do nó,
                              nó, ref_b)
                  se ref_b então
                        se balanço do nó = -1 então
                              balanço do nó = 0
                              ref_b = falso
                        senão se balanço do nó = 0 então
                              balanço do nó = 1
                        senão
                              rotacoesEsquerda(nó)
                              ref_b = falso
```

3.2.2. Remoção

A respeito do algoritmo de remoção da AVL. Esta abordagem de solução usa a pesquisa para retornar o nó que deve ser removido e a partir disso é possível fazer as operações de balanceamento adequadas para que haja manutenção das propriedades da AVL.

É feita a verificação de qual é o caso de remoção e posteriormente é chamada um método de ajuste do balanceamento da estrutura, responsável por atualizar o balanço dos nós e também chamar as rotações adequadas quando necessário. A complexidade está associada a altura da árvore que sempre se mantém balanceada, ou seja, log(n), as rotações representam uma quantidade limitada de operações, sendo n o número de nós da árvore.

```
delete(chave)
      nó = pesquisa(chave)
      se nó /= null então
            se filho esquerdo de nó /= null &&
            filho direito de nó /=null então
                  esq = max(filho esquerdo de nó
                  paiEsquerdo = pega pai de esq
                  replace(nó, esq)
            se paiEsquerdo = nó então
                  balanço de esq = balanço de nó
                  ajusteBalanco(esq, chave de esq - 1 )
            senão
                  balanço de esq = balanço de nó
                  ajusteBalanco(paiEsquerdo, chave de esq)
      senão
            se filho esquerdo de nó = null &&
            filho direito de nó = null então
                  remove(nó, null)
            senão se filho esquerdo de nó /= null &&
            filho direito de nó = null então
                  remove(nó, filho esquerdo de nó)
            senão
                  remove(nó, filho direito de nó)
            ajusteBalanco(pai do nó, chave do nó)
      retorna nó

ajusteBalanco(nó, chave)
      se nó /= null então
            pai = pai do nó
            se chave < chave do nó então
                  balanço do nó = balanço do nó + 1
            senão
                  balanço do nó = balanço do nó + 1
            se balanço do nó /= 1 && balanço do nó /= -1 então
                  se balanço do nó = 2 então
                        rotação à esquerda no nó
                  senão se balanço do nó = -2 então
                        rotação à direita no nó
                  ajusteBalanco(pai, chave do nó)
```

3.3. Rubro Negra

Uma Rubro Negra (RN) consiste em uma árvore que possui uma coloração especial para seus nós, estes podem assumir a cor rubra ou negra (como o próprio nome da estrutura sugere). Também é uma árvore binária de busca, porém é acrescentado o conceito de nó externo. Cada

nó externo sempre é negro e possui altura igual a 0. O objetivo da RN é manter a quantidade de nós negros da raiz para qualquer extremidade sempre igual. Na implementação a RN é filha da *BST*.

Por ser também uma *BST*, a operação de busca é igual para as duas estruturas.

3.3.1. Inserção

A inserção da RN é feita similarmente a da *BST*, apenas com a preocupação adicional de verificar se a coloração está adequada. Nesse algoritmo é utilizado um método auxiliar ajusteCor que faz o ajuste de cores e chama as rotações adequadas, para definir a rotação adequada precisa-se de um método de rotação, similar ao que tem para a árvore AVL, já mostrado anteriormente.

A complexidade da operação depende da altura da árvore

```
rotacao(nó, pai, avo)
      se pai = filho esquerdo do avo então
            se nó = filho esquerdo do pai então
                  rotacaoDireita(avo, pai)
                  se raiz = avo então
                        raiz = pai
                  cor do pai = negro
            senão
                  rotacaoDuplaDireita(avo, pai, nó)
                  se raiz = avo então
                        raiz = nó
                  cor do nó = negro
            cor do avo = rubro
      senão
            se nó = filho do pai então
                  rotacaoEsquerda(avo, pai)
                  se raiz = avo então
                        raiz = pai
                  cor do pai = negro
            senão
                  rotacaoDuplaEsquerda(avo, pai, nó)
                  se raiz = avo então
                        raiz = nó
                  cor do nó = negro
            cor do avo = rubro

ajusteCor(nó, pai, ref_b)
      avo = pai do pai
      tio
      se filho esquerdo do avo = pai então
            tio = filho direito do avo
      senão
            tio = filho esquerdo de avo
      se tio /= null && cor do tio = rubro então
            ref_b = 0
            cor do tio = negro
            cor do pai = negro
            cor do avo = negro
```

```
        senão
                rotacao(nó, pai, avo)
                ref_b = 2
        se cor da raiz = rubro então
                cor da raiz = negro

insercao(chave, nó, pai, ref_b)
        se nó = null então
                nó = cria novo nó(pai, null, null, chave)
                se raiz = null então
                        raiz = nó
                        cor do nó = negro
                senão
                        se chave < chave do pai então
                                filho esquerdo do pai passa a ser nó
                        senão
                                filho direito do pai passa a ser nó
                        retorna nó
        senão
                se chave /= chave do nó então
                        pai = nó
                        se chave < chave do nó então
                                nó = filho esquerdo do nó
                        senão
                                nó = filho direito do nó
                        nó = insercao(chave, nó, pai, ref_b)
                        se cor do pai = negro
                                ref_b <- 2
                        senão se ref_b = 1 então
                                ajusteCor(nó, pai, ref_b)
                        senão se ref_b = 0 então
                                ref_b = 1
                senão
                        ref_b <- 2
                retorna pai
```

3.3.3. Draw

O algoritmo de desenho da *BST* é utilizado para desenhar a RN, exceto que esta estrutura também é representada juntamente com seus nós externos. Portanto, foi utilizada uma ideia similar a da binária de busca porém com um aprimoramento para desenhar os nós especiais. A complexidade deste algoritmo é $\Theta(n)$, sendo n o número de nós.

```
preOrdem(mapa com vértices, x, y, cor)
        se raiz /= null então
                criar vértice
                se filho esquerdo da raiz /= null então
                        preOrdem(mapa, x recalculado, y recalculado,
                        cor do nó)
                        insere aresta do pai para filho
                senão
                        cria vértice externo t
```

```
se filho direito da raiz /= null então
      preOrdem(mapa, x recalculado, y recalculado,
      cor do nó)
      insere aresta do pai para filho
senão
      cria vértice externo
```

3.4. Conjuntos Disjuntos (*UnionFind*)

O *UnionFind* é uma estrutura que segue a ideia de conjuntos disjuntos da matemática, isto é, conjuntos que não possuem elementos em comum. Sua implementação é feita utilizando um vetor para em que dado um índice *i* o elemento armazenado é o representante do elemento *i*. Além disso, também é utilizado um vetor para armazenar a ordem de cada elemento, na posição *i* é encontrada a ordem do elemento *i*.

3.4.1. Gerar

A operação gerar é o momento inicial em que o conjunto disjunto é criado, mediante um valor inicial que indica o tamanho da floresta. Cada elemento é representante de si mesmo. A complexidade dessa operação é $\Theta(n)$. Essa operação se encontra no construtor da classe que representa essa estrutura.

```
para i = 0 até n-1 fazer
      p[x] = x
      ordem[x] = 0
```

3.4.2. *Unite*

Consiste em unir dois conjuntos disjuntos considerando a ordem de cada um deles. A complexidade desse algoritmo é O(1), pois é executada uma quantidade limitada de ações.

```
unite(x, y)
      se ordem[x] > ordem[y]
            p[y] = x
      senão
            p[x] = y
            se ordem[x] = ordem[y] então
                  ordem[y] = ordem[y] + 1
```

3.4.3. *Find*

O algoritmo find não apenas busca pelo representante do conjunto como também faz um operação que modifica a estrutura, fazendo um processo de compressão do caminho. Sua complexidade é O(n), caso o conjunto disjunto possua todos os nós compondo sua altura máxima.

```
find(x)
      se x /= p[x]
            p[x] = busca( p[x] )
      retorna p[x]
```

3.4.4. *Draw*

O algoritmo de desenho da *UnionFind* desenha primeiro os nós pais para depois desenhar os nós filhos de maneira iterativa. Sua complexidade é $O(n^2)$, pois percorre sempre todo o vetor com os elementos dos conjuntos por *n* vezes, sendo n a altura da maior árvore (conjunto disjunto), esta altura no pior caso pode ser *n*, a quantidade de elementos.

```
draw()
```

```
int x = 50, y = 0;
para j = 0 até j = maiorOrdem fazer
      para i = 0 até i < quantidade de elementos
            se level(i) = j então
                  int X, Y
                  se j = 0 então
                        X = x
                        Y = y
                  senão
                        pega posição do pai
                        X = posição x do pai
                        Y = posição y do pai
                        altera posição x do pai para + 50
                  cria vértice com posição X, Y
                  se j = 0 então
                        insere aresta que liga o
                              nó a si mesmo
                  senão
                        insere aresta que liga o
                              elemento i a seu pai
                  verifica se é necessário alterar as
                  coordenadas de x e y para respeitar os limites
```

3.5. *HeapMax/ HeapMin*

A *heap* é uma estrutura que pode ser interpretada como uma árvore binária completa à esquerda, mas que é implementada com um vetor (técnica conhecida como implementação implícita, também aplicada em *UnionFind*). Cada nó deve ter como chave um valor maior ou menor que de seus filhos, de acordo com a comparação usada para cada um dos dois tipos da *heap*.

3.5.1. Inserção

A inserção da *heap* é feita por meio do uso de um método auxiliar de natureza recursiva chamado *goUp*. A complexidade da inserção tem dependência desse método, que é de O(log *n*), sendo *n* o número de nós.

```
insert(int chave)
      se vetor não contém chave então
            imprime chave
            vetor recebe chave na última posição
            se tamanho do vetor > 1 então
                  goUp(última posição)


goUp(pos)
      int aux recebe posição do pai de vetor(pos);
      se aux >= 1 então
            se nó pos superior a nó aux
                  troca nós pos com nó aux
                  se aux > 1 então
                        goUp(aux)
```

3.5.2. Remoção

A remoção da *heap* é feita por meio do uso de um método auxiliar de natureza recursiva chamado *goDown*. A complexidade da remoção tem dependência desse método, que é de O(log *n*), sendo *n* o número de *nós*.

```
remove()
	troca nó 1 com nó último
	deleta último nó
	goDown(1)
fim

goDown(pos)
	int aux = posição do filho esquerdo de pos
	se aux <= nós no vetor então
		se aux+1 <= nós no vetor então
			se nó aux+1 superior a nó aux então
				aux++
		se nó pos superior a nó aux então
			troca nó pos com nó aux
			goDown(aux)
```

3.5.3. Busca

O algoritmo de busca dessa estrutura de dados é o simples percorrimento do vetor por meio de um laço. A complexidade dele é Θ(*n*), pois o vetor é percorrido apenas uma vez, sendo *n* a quantidade de nós.

```
containsKey(chave)
	boolean contains = false
	para i = 1 até i <= quantidade de elementos
		se vetor[i] = chave então
			contains = true
	retorna contains
```

3.5.4. *Draw*

O algoritmo de desenhar a *heapmax* é iterativo e consiste em desenhar por nível (percorrendo o vertor) a estrutura, uma vez desenhado o pai é possível desenhar os filhos. Sua complexidade é Θ(*n*), pois o vetor com os elementos é percorrido apenas uma vez.

```
draw()
	int pai = 0
	para i = 0 até i < tamanho do vetor fazer
		se i % 2 = 0 então
			se i /= 0 então
				pai = vetor.pegar(i/2 -1)
				calcula posição x baseado no pai
			senão
				posição x metade do tamanho padrão do panel
		senão
			pai = vetor.pegar(i/2)
			calcula posição x baseada no pai
```

```
se 2^(nível) = i + 1 então
      nível++
      y+= deltaY
cria vértice de i
se i /= 0 então cria aresta ligando pai a filho
```

3.6. *Deque*, *list*, *queue*, *stack*

3.6.1. *Draw*

Como citado anteriormente, não for feita a implementação das estruturas deste tópico, porém foi feita a função que as desenham usando o modo iterativo, segue apenas um pseudo código, pois todas são baseadas nessa mesma ideia. Complexidade é $\Theta(n)$, sendo n o número de nós, pois a estrutura é percorrida apenas uma vez.

```
draw()
      d = true
      para i = 0 até i < número de elementos
            criar vértice na posição x, y, cor vermelha
            se x + 60 > largura padrão do panel && d então
                  d = falso
                  y+= 60
            senão se x - 60 < 0 && !d então
                  d = verdade
                  y+= 60
            se d então
                  x+= 60
            senão
                  x-= 60
            se i /= 0 então
                  insere aresta de i para i + 1
                  insere aresta de i + 1 para i
```

4. Detalhes do projeto

Para este projeto foram utilizados os padrões *Factory* e *Observer*.

A necessidade da utilização do padrão *Factory* foi decorrente da utilização de muitas estruturas de dados e a busca por lidar de maneira eficiente com elas no momento da criação. O conceito desse padrão de projeto vem a calhar, pois poder direcionar a uma classe a solicitação de objetos e, portanto, separar criação de manipulação tornou melhor o controle da coesão e acoplamento, ou seja, a maneira como as classes se relacionam em nosso projeto.

O padrão de projeto *Observer* foi utilizado em múltiplas instâncias na interface do usuário, pois para cada ação do usuário há algo que precisa ser notificado.

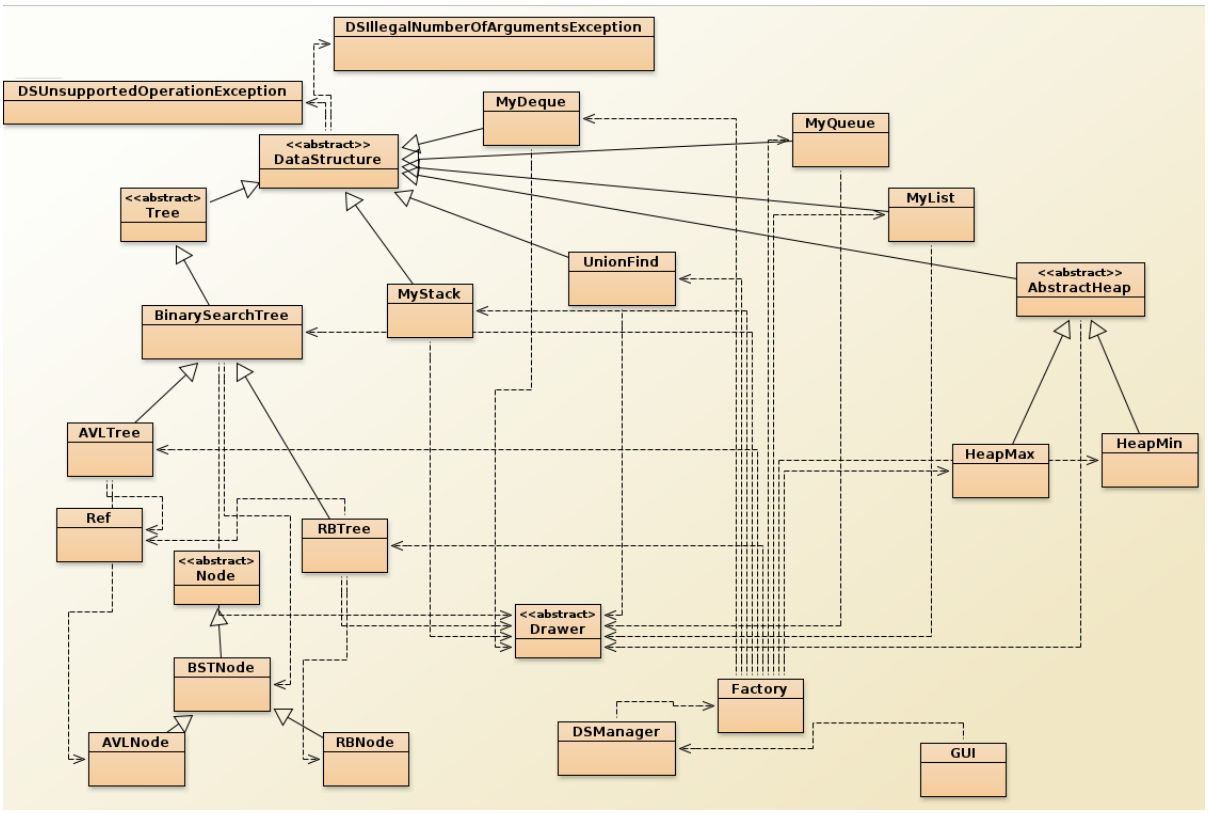

Figura 1. Diagrama de classes do projeto

5. Conclusão

Através da implementação deste projeto foi possível aplicar os conhecimentos adquiridos nas disciplinas de Linguagem de Programação II e Estruturas de Dados Básicas II, para os membros do grupo foi evidente o amadurecimento como programadores, pela primeira vez foram aplicados padrões de projeto e além disso, a visualização das estruturas depois de prontas foi muito realizador.

Entre as principais dificuldades enfrentadas pelo grupo fica a utilização do GIT e conhecer melhor a linguagem Java para modelar os algoritmos que foram vistos em classe em pseudo código similar a C++.