Relatório segundo trabalho ED 2

O presente trabalho consiste em encontrar elementos pertencentes a dois conjuntos e realizar operações de inserção e remoção caso o elementos esteja em um conjunto e não em outro, etc.
Para isso deve ser implementadas as estruturas de dados:

- Lista simplesmente encadeada
- Hash table
- Árvore binaria de busca
- Árvore AVL
- Árvore Rubro Negra

Arquivos

O arquivos fornecidos possuem chaves numéricas com 1000, 5000, 10000, 50000, 100000, 500000 e 1000000 chaves. As chaves devem ser armazenas nas estruturas de dados especificadas.
O conjunto com menor quantidade de chaves deve ser armazenada na lista simplesmente encadeada, o segundo conjunto deve ser armazenada conforme a escolha do usuário.

Estruturas Implementadas

Para o presente trabalho foram implementadas as seguintes estruturas de dados:

Lista Simplesmente Encadeada

A lista simplesmente encadeada foi implementa sob duas classes, a classe Node, e a classe List
A classe node possui um ponteiro para o objeto Node.data definido no construtor da classe, e um ponteiro Node.next que aponta para o próximo Nó, mas é inicializado com o valor None (equivalente ao NULL da linguagem C).

```
class Node(object):
 def __init__(self, data):
  self.data = data
  self.next = None
```

A classe List implementa a estrutura da lista e seus principais métodos, seus atributos são List.first que é inicializado com None, e List.len que é inicializado com 0.

```
class List(object):
 def __init__(self):
  self.first = None
  self.len = 0
```

Seus métodos implementados foram o **append ** que insere um elemento no inicio da lista, o **get** que retorna o elemento por uma chave e recebe por argumento uma função de comparação, caso a função não seja definida é usada uma função de comparação padrão. e a função **remove** que remove um elemento

dada um chave.
Também foram implementadas métodos de interface da linguagem python como a função *def __iter__(self):* que implementa um iterador a lista.

Hash Table

A estrutura **Hash Table** foi implementada em duas classes a **HashItem** que guarda o dado e possui as flags de controle.

```
class HashItem:
 def  __init__(self):
  self.key = None
  self.data = None
  self.stat = "None"
```

O atributo stat é a flag de controle inicializada com “None” para itens vazios, “insert” para itens ocupados, e “remove” para itens removidos.
A classe **HashTable** implementa a estrutura da tabela hash, ela possui um lista de *HashItens*, um tamanho (Quantidade de HashItens Alocados), e um atributo len com a quantidade de elementos inseridos na estrutura.
O tamanho escolhido foi 37, por ser um numero primo e por não ser par, Quando a tabela chegar a 70% de sua capacidade é alocada uma nova tabela com capacidade 37^x sendo x a quantidade de vezes que esta operação ja foi realizada.

```
class HashTable(object):
 def __init__(self, len=37):
  self.len = len
  self.numItens = 0
  self.lista = []
  for i in range(self.len):
   self.lista.append(HashItem())
```

Para esta estrutura foram implementados os métodos **insert**, **get** e **remove**

Árvore Binaria de Pesquisa

A estrutura foi implementada em duas classes, a classe TNode, com um ponteiro para os dados, inicializados no construtor da classe, e dois ponteiros para suas sub-árvores, inicializados com valor None.

```
class TNode(object):

 def __init__(self, data):
  self.data = data
  self.left = None
  self.right = None
```

E a classe BTree, que implementa a estrutura da árvore e seus principais métodos. Seus atributos são a raiz, que é inicializada por None, e duas funções de comparação, Uma para comparar dois objetos do mesmo tipo, para a inserção, e uma para comparar a chave com os dados da árvore para a remoção e busca. Caso essas funções não sejam definidas, é usada um função padrão de comparação implementada na própria classe.

```
class BTree(object):
 def __init__(self, CMPFunc = None, CMPKey = None):
  self.root = None
  if CMPFunc == None:
   self.cmp = self.defautCMP
  else:
   self.cmp = CMPFunc
  if CMPKey == None:
   self.cmpKey = self.defautCMP
  else:
   self.cmpKey = CMPKey
```

Foram implementados os métodos **insert**, **get** e **remove**.

Árvore AVL

A árvore AVL foi implementada recursivamente utilizando duas classes, a **TNode**, e a **AvlTree**. A classe **TNode** possui um ponteiro para o dado, e dois ponteiros para as sub-árvores, além de um atributo para a altura do nó.

```
class TNode(object):
 def __init__(self, data):
  self.data = data
  self.left = None
  self.right = None
  self.h = 0
```

A classe **AvlTree** implementa a estrutura e os métodos básicos do TAD, seus atributos são a o Nó raiz inicializado com None, e as duas funções de comparação, uma para comparação de chaves, e outras comparação de objetos para inserção. Caso as funções não forem definidas há uma função de comparação padrão na classe.

```
class  AvlTree(object):
 def __init__(self, funCMP=None, funCMPKey = None):
  self.root = None
  if funCMP is None:
   self.cmp = self.defautCMP
  else:
   self.cmp = funCMP
  if funCMPKey is None:
   self.cmpKey = self.defautCMP
  else:
   self.cmpKey = funCMPKey
```

Para essa estrutura foram implementados os métodos **insert** e **get**

Árvore Rubro Negra

A Árvore Rubro Negra foi implementada de forma iterativa tendo em vista as operações em que é necessário o Nó avô, ou o Nó tio. para isso incluso na classe **TNode** o atributo *parent*, que aponta para o Nó pai.

```
class TNode(object):
 def __init__(self, data=None, left=None, right=None, parent=None, color=True):
  self.data = data
  self.left = left
  self.right = right
  self.parent = parent
  self.color = color #true is red; false is black
```

Também foram acrescentados métodos a classe estrutural da árvore **RedBlack**, como *getColor*. A classe RedBlack possui como atributos o Nó raiz, e as duas funções de comparação.

```
class RedBlack(object):
 def __init__(self, funCMP=None, funCMPKey = None):
  self.__root = None
  if funCMP is None:
   self.cmp = self.defautCMP
  else:
   self.cmp = funCMP
  if funCMPKey is None:
   self.cmpKey = self.defautCMP
  else:
   self.cmpKey = funCMPKey
```

Para essa estrutura foram implementados os métodos **insert** e **get**

Estrutura do Trabalho

O trabalho foi estruturado em dois arquivos, o **main**.**py** onde são definidos os parâmetros por linha de comando e o método **main**, e o arquivo **utils**.**py** onde são definidas funções utilitárias como a leitura dos arquivos e o carregamento dos dados nas estruturas.
O arquivo de saída é nomeado como “out.txt”

Linha de Comando

deve ser passado por linha de comando os arquivos, a estrutura onde será carregado o segundo arquivo, e o tipo de operação.

```
python3 main.py [conjunto a] [conjunto b] [estrutura] [op]
```

- [conjunto a] arquivo menor que será carregado na lista
- [conjunto b] arquivo maior que será carregado na estrutura definida
- [estrutura] estrutura utilizada para carregar o conjunto b
  – lista
  – hash
  – binary_tree
  – avl
  – red_black
- [op] operação realizada
  – 1 : interseção dos elementos de A e B
  – 2 : inserir em B, os elementos de A que não estão em B

– 3 : remover os elementos de A que estão em B

Exemplo:

```
python3 main.py data/files1000.txt data/files5000.txt hash 3
```

Carregamento dos Arquivos

O carregamento de arquivos foi realizado através de uma função definida na utils, ela ler a unica linha do arquivo, trata a string removendo as “[]”, e posteriormente separa por ", " para converter os números para o tipo **int**, e retorna uma lista da biblioteca padrão do python com os inteiros.

```
def readAchive(arq):
 file = open(arq, "r")
 string = file.read()
 string = string.replace('[', '')
 string = string.replace(']', '')
 lista = string.split(", ")
 listNum = []
 for item in lista:
  listNum.append(int(item))
 return listNum
```

Carregamento das Estruturas

As estruturas são carregadas a partir das lista retornadas da função anterior. A estrutura é instanciada, e posteriormente é inserido item a item da lista na estrutura.

```
def loadAVL(lista):
 if not lista is None:
  MyAVL = AvlTree()
  for item in lista:
   MyAVL.insert(item)
  return MyAVL
 return None
```

Operação 1

A operação 1 é a interseção do conjunto A com o conjunto B. Primeiramente é alocado uma lista para guardar os elementos que estão em A e em B, depois para cada elemento de A, usa-se a função de busca em B. caso retorne um valor valido significa que este elemento está presente nos dois conjuntos, e é inserido na lista. Ao fim a lista com a interseção é retornada.

```
def OP1BTree(lista, BTree):
 if not (lista is None or BTree is None):
  inter = []
  for item in lista:
   aux = BTree.get(item)
   if aux == item:
    inter.append(item)
  return inter
```

O arquivo de saída contem a interseção dos dois conjuntos.

Operação 2

A operação 2 é inserir no Conjunto B, os elementos de A que não estão em B, para isso é necessário percorrer a lista com o conjunto A, e para cada elemento de A é feita uma busca no conjunto B, caso essa busca retorne um valor invalido, inserimos o elemento em B. e por fim é retornado a string da estrutura B (equivalente ao toString do java).

```
def OP2RB(lista, RB):
 if not (lista is None or RB is None):
  for item in lista:
   aux = RB.get(item)
   if aux == None:
    RB.insert(item)
  return str(RB)
```

O arquivo de saída possui os itens do segundo conjunto.

Operação 3

A operação 3 é remover os elementos do conjunto A que também pertencem ao conjunto B. Com isso para cada item de A é feita uma busca em B, caso essa busca retorne um valor valido, o item é removido de A. Ao fim se retorna uma string com todos os elementos de A.

```
def OP3Hash(lista, Hash):
 if not (lista is None or Hash is None):
  for item in lista:
   aux = Hash.get(item)
   if aux != None:
    lista.remove(item)
  return str(lista)
```

O arquivo de saída possui os elementos do conjunto A.

Resultados

Os resultados foram conseguidos através do intervalo de tempo em que as estruturas eram carregadas e a operação era realizada, não contabilizando o tempo de leitura dos arquivos.

Os primeiros testes foram realizados testes com o conjunto A com 1.000 chaves e o conjunto B com 5.000, 10.000, 100.000 e 1.000.000 chaves.

Conjunto A	Conjunto B
1.000	5.000
1.000	100.000

Conjunto A	Conjunto B
1.000	500.000
50.000	100.000
50.000	500.000
100.000	500.000

Primeiro Caso

Conjunto A com 1000 e conjunto B com 5000

Estrutura	OP1	OP2	OP3
lista	6,02s	6,20s	6,34s
hash	0,45s	0,47s	0,62s
Av. Binaria	18,56s	19,87s	19,23s
Av. AVL	1,01s	1,66s	1,22s
Av. RN	0,47s	0,62s	0,74s

Segundo Caso

Conjunto A com 1.000 e conjunto B com 100.000

Estrutura	OP1	OP2	OP3
lista	79,46s	67,68s	71,61s
hash	7,01s	9,90s	7,86s
Av. Binaria	ERROR	ERROR	ERROR
Av. AVL	9,10s	10,82s	9,05s
Av. RN	3,59s	5,98s	4,58s

Neste teste a arvore binaria de pesquisa apresentou o error:
MemoryError: stack overflow

Terceiro Caso

Conjunto A com 1.000 e conjunto B com 500.000

Estrutura	OP1	OP2	OP3

Estrutura	OP1	OP2	OP3
lista	–	–	–
hash	8,09s	15,7s	7,42s
Av. Binaria	–	–	–
Av. AVL	52,38s	69,41s	74,37s
Av. RN	28,22s	42,84s	20,99s

Quarto Caso

Conjunto A com 50.000 e conjunto B com 100.000

Estrutura	OP1	OP2	OP3
lista	–	–	–
hash	7,09s	9,31s	13,92s
Av. Binaria	–	–	–
Av. AVL	12,37s	17,86s	18,29s
Av. RN	5,82s	10,48s	14,67s

Quinto Caso

Conjunto A com 50.000 e conjunto B com 500.000

Estrutura	OP1	OP2	OP3
lista	–	–	–
hash	7,01s	16,29s	45,14s
Av. Binaria	–	–	–
Av. AVL	49,11s	81,50s	144,62
Av. RN	22,11s	38,42s	111,94s

Sexto Caso

Conjunto A com 100.000 e conjunto B com 500.000

Estrutura	OP1	OP2	OP3
lista	–	–	–
hash	7,75s	17,20s	–

Estrutura	OP1	OP2	OP3
Av. Binaria	–	–	–
Av. AVL	122,81s	110,01s	–
Av. RN	22,84s	38,97s	–

Conclusão

Na primeira Operação A arvore rubro negra e o hashtable possuem desempenho parecido com arquivos pequenos, mas para arquivos grandes como no terceiro, quinto e sexto caso as tabelas hash são consideravelmente melhores.
As listas não foram utilizadas a partir do terceiro caso pois o tempo utilizado estava muito grande, superando 30 minutos no terceiro caso, dessa forma o processo foi interrompido, e não foi utilizado nos outros testes, tendo em vista que o processamento seria pior.
As arvores binarias de pesquisas não foram muito boas no primeiro caso, talvez pelo custo do processo recursivo, no segundo caso a pilha recursiva da arvore binaria de pesquisa ultrapassou o limite de 1000 recursões definidas por padrão no python sendo necessário mudar o limite através da biblioteca **sys**, isso significa que uma sub-árvore estava com altura maior que 1000, evidenciando o desbalanceamento da árvore.
No terceiro caso, a arvore binaria retornou o erro de *stack overflow*, significa que a pilha recursiva ultrapassou a memoria reservada pela linguagem, a partir deste teste não foi mais utilizado nos testes posteriores.
As árvores AVL, não possuíram os mesmos problemas da árvore binaria de pesquisa, porém a arvore rubro negra teve um desempenho superior, talvez pela implementação recursiva da avl.
No sexto caso não foi realizado a terceira operação pelo mesmo fato, de não utilizar as listas a partir do terceiro caso, o custo de remoção na lista simplesmente encadeada é maior que nas outras estruturas.