

## Trabalho Prático 2 - Manipulação e Operações de um Banco de Dados em Memória Secundária

Lucas Afonso Pereira Chagas - 22050316  
Maria Vitória Costa do Nascimento - 22053592  
Rodrigo Santos Corrêa - 22251139

<sup>1</sup>Instituto de Ciências Exatas – Universidade Federal do Amazonas (UFAM)  
Av. General Rodrigo Octavio Jordão Ramos, 1200 – Coroado I, Manaus – AM – Brasil  
CEP 69080-900

{lucas.chagas, vitoria.nascimento, rodrigo.correa}@icomp.ufam.edu.br

### 1. Decisões de Projeto

Para que pudéssemos começar o nosso trabalho, teríamos de definir as maneiras como faríamos o armazenamento dos dados no arquivo. Dessa forma, calculamos e definimos um conjunto de valores que nos guiarão pelo trabalho.

#### 1.1. Sobre os Registros

No que tange aos registros e sua implementação, tomamos as seguintes decisões, seguindo os materiais que pertenciam à aula 9 - Representação de Dados:

- Formato fixo
- Tamanho variável
- Alocação não espalhada
- Referência totalmente indireta (Cabeçalho apontando para a posição de cada registro)

Temos que a média dos registros foi de aproximadamente 519 bytes, como diz a imagem abaixo:



A media em bytes de um registro eh de: 518.91

Figura 1. Média dos registros

Dividindo o tamanho livre em um bloco (4060 bytes), considerando a alocação em modo não espalhada e os valores dos alfas disponibilizados na especificação, podemos inserir 7 registros por bloco, ou 2 registros no pior caso (quando o snippet alcança o alfa 1024).

#### 1.2. Sobre os Blocos

O tamanho do bloco foi estabelecido em 4096 bytes, uma decisão baseada no slide 09 apresentado. Pela escolha de fazer referência indireta para os registros, o bloco foi dividido em cabeçalho e área para registros. No cabeçalho temos informações gerais do bloco: o tamanho disponível, a quantidade de registros inseridos, e as posições dos mesmo.

Sobre o cabeçalho, o tamanho disponível e a quantidade de registros são representados por inteiros, ocupando um total de 8 bytes. Além disso, as posições dos registros são

indicadas por uma lista de inteiros, cujo tamanho é proporcional à quantidade de registros. No nosso melhor caso, para 7 registros por bloco, a lista ocupa 28 bytes.

Assim, temos que o tamanho total do cabeçalho é de 36 bytes, deixando 4060 bytes para a área de registros.

### **1.3. Sobre os Buckets**

Com 7 registros por bloco para a quantidade de registros de entrada precisaríamos de 146.000 blocos, entretanto, esse número de blocos divididos em buckets não funciona quando usamos ele exatamente, pois, por causa do tamanho variável dos registros e pela alocação não espalhada acabamos precisando de mais blocos, no final a quantidade necessária para a quantidade de registros de entrada a fim de não termos overflow foi 225.000 blocos, estes foram divididos em 15.000 buckets com 15 blocos por bucket.

Escolhemos fazer uma divisão de mais buckets do que blocos para diminuir ainda mais as chances de overflow, pois assim temos um range maior da chave de hash, os blocos foram então divididos em partes iguais para a quantidade de buckets estabelecida.

### **1.4. Sobre a função Hash**

- Como função para hash, usamos duas ideias do livro:
  - Fazer a divisão inteira entre o id e um número primo;
  - Fazer uma operação com o número de buckets;

### **1.5. Sobre entradas fora do padrão**

No arquivo de entrada, encontramos casos de exceção, isto é, entradas que não continham os dados requeridos, entradas sem título, citações, entre os demais campos. Decimos então por não inserir essas entradas no arquivo de dados e consequentemente nos arquivos de índice.

Pensamos que esses dados poderiam comprometer a integridade dos nossos arquivos, o que não seria aceito de forma prática em um banco de dados.

## **2. Arquivos fontes para a formação de cada programa**

Nessa seção estão os arquivos fontes auxiliares, primários e o extra, adicionado para facilitar a correção e de uso opcional.

Arquivos fontes auxiliares, usadas para criar as estruturas que serão utilizadas para as operações requisitadas estão distribuídos em pastas, por sua semântica, são essas:

- bptree
- estruturas
- hash

Arquivos auxiliares:

- bptreeSerializada.cpp
- bptreeStringSerializada.cpp
- definicoes.hpp
- estruturaBlocoRegistro.hpp
- parser.cpp

- hashTable.cpp

Arquivos fontes principais, como requisitados na especificação

- upload.cpp
- findrec.cpp
- seek1.cpp
- seek2.cpp

## 2.1. Arquivo extra - Make

Decidimos criar um Makefile que executa o comando "g++ -std=c++11 -o <nome\_executável> <nome\_arquivo\_fonte>", para facilitar a correção. Para usar, basta usar o comando: make -B.

Inserimos -std=c++11 para indicar o compilador a versão que gostaríamos de utilizar, pois no código utilizamos o ponteiro \*auto, que não é suportado por versões de compiladores anteriores a essa.

## 3. Funções presentes em cada arquivo fonte

### 3.1. Arquivo parser.cpp

#### Listing 1. Função remover\_aspas

```
void remover_aspas(std::istream& stream, std::string&
    resultado)
```

Remove as aspas de uma entrada do arquivo "artigo.csv". Isso se viu necessário pois há entradas com múltiplas aspas aninhadas, que não é a regra para a maioria das entradas. Por exemplo na entrada: "262150";"two - 4 - six;...

(Feita por: Maria Vitória).

#### Listing 2. Função remover\_aspas\_input

```
string remover_aspas_input(const std::string& entrada)
```

Remove as aspas do input realizado no arquivo seek2, no qual o título deve ser passado como parâmetro. Por exemplo, no terminal é executado ./seek2 "título", mas apenas título é usado para as operações.

(Feita por: Maria Vitória).

#### Listing 3. Função normalizar\_string

```
std::string normalizar_string(std::string str)
```

Padroniza uma string, essencial para diversos caracteres que podiam afetar o trabalho, estes são:

- Caracteres ASCII acima de 127, que estão em um padrão ASCII estendido;
- \n e aspas duplas;
- Espaços extras;

(Feita por: Maria Vitória).

### 3.2. Arquivo bptreeSerializada.cpp

Class BPlusTree Construtor - BPlusTree(std::size\_t grau): recebe grau da árvore, ou seja, a quantidade máxima de filhos por nó

struct No: implementa o nó da árvore, contendo o grau(número máximo de filhos no nó), a ocupação atual, o vetor interno de registros, um ponteiro para o nó ancestral imediato e um ponteiro de ponteiro para acessar os seus descendentes.

#### Listing 4. Função get\_raiz

```
No<RegistroBPT>* get_raiz()
```

Retorna a raiz da árvore, muito útil para operações recursivas.

(Feita por: Rodrigo Santos)

#### Listing 5. Função busca\_chave\_bptree

```
No<RegistroBPT>* busca_arvore (No<RegistroBPT>* node, int  
    chave_busca)
```

Busca na árvore uma chave passada como parâmetro, indo até a lista encadeada simbolizada pelas folhas e vasculha se a chave buscada está presente.

(Feita por: Rodrigo Santos)

#### Listing 6. Função posiciona\_cursor\_bptree

```
No<RegistroBPT>* busca_arvore_alcance (No<RegistroBPT>* node  
    , RegistroBPT chave_busca)
```

Posiciona o cursor em um nó folha que corresponda a um nó buscado, passado como parâmetro. Usado para operações na árvore, com destaque para inserções e o balanceamento.

(Feita por: Rodrigo Santos)

#### Listing 7. Função busca\_index

```
int busca_index(RegistroBPT* registros, RegistroBPT  
    registro_buscado, int tamanho)
```

Percorre um vetor de registros de um nó, comparando se a chave é a mesma do registro passado, incrementando um valor index retornado. Usado como função auxiliar de inserção.

(Feita por: Rodrigo Santos)

#### Listing 8. Função inserir\_registro

```
RegistroBPT* inserir_registro(RegistroBPT* registros,  
    RegistroBPT registro_insercao, int tamanho)
```

Insere um registro em um vetor de registros internos do nó.

(Feita por: Rodrigo Santos)

#### Listing 9. Função inserir\_filho

```
No<RegistroBPT>** inserir_filho (No<RegistroBPT>** filhos,  
    No<RegistroBPT>* filho, int tamanho, int index):
```

Funcionalidade: Insere um nó (\*filho) no vetor de filhos de um nó.

(Feita por: Rodrigo Santos)

**Listing 10. Função inserir\_registro\_filho**

```
No<RegistroBPT>** inserir_filho (No<RegistroBPT>** filhos,  
    No<RegistroBPT>* filho, int tamanho, int index):
```

Funcionalidade: insere um registro em um nó filho, que é passado como parâmetro. Utilizado na parte de balanceamento da árvore.

(Feita por: Rodrigo Santos)

**Listing 11. Função inserir\_ancestral**

```
void inserir_ancestral (No<RegistroBPT>* ancestral, No<  
    RegistroBPT>* filho, RegistroBPT registro_insercao)
```

Insere um registro no nó ancestral de um nó. Essencial para operações de balanceamento da árvore.

(Feita por: Rodrigo Santos)

**Listing 12. Função inserir\_arvore**

```
void inserir_arvore (RegistroBPT* registro_insercao)
```

Função principal de inserção, insere na árvore um registro passado como parâmetro. Utiliza as outras funções de inserção como auxiliaadoras.

(Feita por: Rodrigo Santos)

**Listing 13. Função deletar**

```
void deletar (No<RegistroBPT>* cursor)
```

Função que recebe como parâmetro a raiz da árvore e deleta recursivamente os nós. Usada no destruidor, como referido anteriormente.

(Feita por: Rodrigo Santos)

**Listing 14. Função imprimir\_arvore**

```
void imprimir_arvore ()
```

Imprime a árvore chamando recursivamente a função que imprime os nós.

(Feita por: Rodrigo Santos)

**Listing 15. Função imprimir\_registro**

```
void imprimir_registro (RegistroBPT registro)
```

Função que imprime um registro da árvore.

(Feita por: Rodrigo Santos)

**Listing 16. Função imprimir\_no**

```
void imprimir_no (No<RegistroBPT>* cursor)
```

Imprime um nó passado como parâmetro. Utilizado de forma recursiva, por isso o nome de seu parâmetro é denominado cursor.

(Feita por: Rodrigo Santos)

**Listing 17. Função desserializar\_no**

```
No<RegistroBPT>* desserializar_no(istream& arquivo, No<
    RegistroBPT>* ancestral, size_t grau)
```

Traz um nó do arquivo de dados para a memória, caso o mesmo não seja uma folha desserializa recursivamente seus filhos.

(Feita por: Rodrigo Santos)

**Listing 18. Função destruir\_no**

```
void destruir_no (No<RegistroBPT>* no)
```

Deleta os dados de um nó. É aplicada recursivamente caso o nó não seja folha.

(Feita por: Rodrigo Santos)

**Listing 19. Função desserializar\_arvore**

```
BPlusTree desserializar_arvore(const string& nome_arquivo)
```

Usa a função desserializar\_no recursivamente para trazer toda a árvore do arquivo para memória.

(Feita por: Rodrigo Santos)

**Listing 20. Função contar\_nos**

```
int contar_nos (No<RegistroBPT>* no)
```

Função para contabilizar os nós totais do arquivo.

(Feita por: Rodrigo Santos)

**Listing 21. Função serializar\_no**

```
void serializar_no(ofstream& arquivo, const No<RegistroBPT
    >* no)
```

Escreve os registros do no no arquivo de indexação de forma recursiva.

(Feita por: Rodrigo Santos)

**Listing 22. Função buscar\_registro\_bpt**

```
Registro* buscar_registro_bpt(istream& arquivo_dados, int
    id_busca)
```

Procura registro no arquivo de index e o lê para memória.

(Feita por: Rodrigo Santos)

**Listing 23. Função serializar\_arvore**

```
void serializar_arvore(const BPlusTree& arvore, const
    string& nome_arquivo)
```

Escreve a árvore recebida em um arquivo binário com o nome passado. Utiliza recursivamente a função `serializar_no`.

(Feita por: Rodrigo Santos)

### 3.3. Arquivo `bptreeStringSerializada.cpp`

Esse arquivo foi desenvolvido para a implementação da árvore B+ voltada ao índice secundário, para isso foram feitas modificações para que trabalhássemos com strings. As funções abaixo diferem em nome por motivos semânticos mas realizam as mesmas operações do outro arquivo `bptreeSerializada.cpp`:

#### Listing 24. Função `get_raiz()`

```
NoS<RegistroString>* busca_arvore_s (NoS<RegistroString>* no
, string chave_busca)
```

(Feita por: Lucas Afonso).

#### Listing 25. Função `busca_arvore_s()`

```
NoS<RegistroString>* busca_arvore_s (NoS<RegistroString>* no
, string chave_busca)
```

(Feita por: Lucas Afonso).

#### Listing 26. Função `busca_arvore_alcance_s()`

```
NoS<RegistroString>* busca_arvore_alcance_s (NoS<
RegistroString>* no, RegistroString chave_busca)
```

(Feita por: Lucas Afonso).

#### Listing 27. Função `busca_index_s()`

```
int busca_index_s (RegistroString* registros, RegistroString
registro_buscado, int tamanho)
```

(Feita por: Lucas Afonso).

#### Listing 28. Função `inserir_registro_s`

```
RegistroString* inserir_registro_s (RegistroString*
registros, RegistroString registro_insercao, int tamanho
)
```

(Feita por: Lucas Afonso).

#### Listing 29. Função `inserir_filho_s`

```
NoS<RegistroString>** inserir_filho_s (NoS<RegistroString>**
filhos, NoS<RegistroString>*filho, int tamanho, int index
)
```

(Feita por: Lucas Afonso).

#### Listing 30. Função `inserir_registro_filho_s`

```
NoS<RegistroString>* inserir_registro_filho_s (NoS<
RegistroString>* no_insercao, RegistroString
registro_insercao, NoS<RegistroString>* filho)
```

(Feita por: Lucas Afonso).

**Listing 31. Função inserir\_ancestral\_s**

```
void inserir_ancestral_s(NoS<RegistroString>* ancestral, NoS<RegistroString>* filho, RegistroString registro_insercao)
```

(Feita por: Lucas Afonso).

**Listing 32. Função inserir\_arvore\_s**

```
void inserir_arvore_s(RegistroString* registro_insercao)
```

(Feita por: Lucas Afonso).

**Listing 33. Função deletar\_s**

```
void deletar_s(NoS<RegistroString>* cursor)
```

(Feita por: Lucas Afonso).

**Listing 34. Função imprimir\_arvore\_s**

```
void imprimir_arvore_s()
```

(Feita por: Lucas Afonso).

**Listing 35. Função imprimir\_registro\_s**

```
void imprime_registro_s(const RegistroString& registro,  
    const std::string& prefixo)
```

(Feita por: Lucas Afonso).

**Listing 36. Função imprimir\_no\_s**

```
void imprimir_no_s(NoS<RegistroString>* cursor, int nivel,  
    const std::string& prefixo)
```

(Feita por: Lucas Afonso).

**Listing 37. Função desserializar\_no\_s**

```
NoS<RegistroString>* desserializar_no_s(ifstream& arquivo,  
    NoS<RegistroString>* ancestral, size_t grau)
```

(Feita por: Lucas Afonso).

**Listing 38. Função destruir\_no\_s**

```
void destruir_no_s(NoS<RegistroString>* no)
```

(Feita por: Lucas Afonso).

**Listing 39. Função desserializar\_arvore\_s**

```
BPlusTreeString desserializar_arvore_s(const string&  
    nome_arquivo)
```

(Feita por: Lucas Afonso).



**Listing 40. Função contar\_nos\_s**

```
int contar_nos_s(NoS<RegistroString>* NoS)
```

(Feita por: Lucas Afonso).

**Listing 41. Função serializar\_arvore\_s**

```
void serializar_arvore_s(const BPlusTreeString& arvore,  
    const string& nome_arquivo)
```

(Feita por: Lucas Afonso).

**Listing 42. Função serializar\_no\_s**

```
void serializar_no_s(ofstream& arquivo, const NoS<  
    RegistroString>* no)
```

(Feita por: Lucas Afonso).

**Listing 43. Função buscar\_registro\_s**

```
Registro* buscar_registro_s(ifstream& arquivo_index, string  
    id_busca)
```

(Feita por: Lucas Afonso).

### 3.4. Arquivo hashTable.hpp

Abaixo estarão as funções para criação, uso e operações que envolvam a tabela Hash.

**Listing 44. Função media\_registros**

```
void HashTable::media_registros()
```

Calcula a média de tamanho em bytes dos registros. Útil para gerar as informações necessárias para otimizar o número de buckets, blocos e a quantidade de registros por bloco.

(Feita por: Maria Vitória)

**Listing 45. Função deletar\_bucket**

```
void deletar_bucket(Bucket* bucket)
```

Deleta um bucket da memória.

(Feita por: Maria Vitória)

**Listing 46. Função inicializar\_bucket**

```
void inicializar_bucket(ofstream& arquivoBin)
```

Inicializa um bucket, criando os blocos que estarão sobre seu domínio e escrevendo-o no arquivo de dados.

(Feita por: Maria Vitória)

**Listing 47. Função HashTable**

```
HashTable::HashTable(string nomeArquivo, bool sobrescrever  
    = true)
```

Inicializador da class `HashTable`, cria o arquivo de dados, abre o arquivo de entrada e inicializa os buckets.

(Feita por: Maria Vitória)

**Listing 48. Função `funcao_hash`**

```
size_t HashTable::funcao_hash(size_t id)
```

Retorna o índice de um bucket, utilizando a técnica de espalhamento máximo.

(Feita por: Maria Vitória)

**Listing 49. Função `inserir_registro_bucket`**

```
size_t HashTable::inserir_registro_bucket(Registro*  
    registro)
```

Insere o registro em um bucket, utilizando a função `Hash` e invocando o a função que insere o registro em um bloco. O bloco então é copiado para um buffer e em seguida escrito no arquivo de saída.

(Feita por: Maria Vitória)

**Listing 50. Função `busca_registro_hashtable`**

```
Registro* HashTable::busca_registro_hashtable(int id)
```

Ao aplicar a função `hash`, obtemos o índice do bucket no qual devemos inserir um registro desejado. Percorre-se então os blocos do bucket, trazendo-os para a memória em loop, comparando se o `id` do registro com o `id` buscado. Se sim, o registro é preenchido, devolvido e o bloco é apagado da memória.

(Feita por: Maria Vitória)

### 3.5. Arquivo `estruturaBlocoRegistro.hpp`

Abaixo, estarão as funções responsáveis pelas operações que envolvem diretamente os blocos e os registros.

**Listing 51. Função `criar_registro`**

```
Registro* criar_registro(int id, string title, int year,  
    string authors, int citations, string update, string  
    snippet)
```

Função que cria um registro nos moldes das entradas do arquivo `csv`.

(Feita por: Rodrigo Santos).

**Listing 52. Função `imprimir_registro`**

```
void imprimir_registro(Registro* registro)
```

Imprime no terminal um registro passado como parâmetro. (Feita por: Rodrigo Santos).

**Listing 53. Função `preencher_registro`**

```
Registro* preencher_registro(Registro* registro, Bloco*  
    bloco, int cursor)
```

Utilizado como função auxiliar para a busca de um registro na hashTable. Preenche um registro com as informações vindas do disco restantes (com exceção do id, que já foi lido no arquivo de hash).

(Feita por: Rodrigo Santos).

**Listing 54. Função criar\_cabecalho\_bloco**

```
Cabecalho_Bloco* criar_cabecalho_bloco()
```

Cria um cabeçalho de bloco, responsável por conter informações dos blocos.

(Feita por: Rodrigo Santos).

**Listing 55. Função criar\_bloco**

```
Bloco* criar_bloco()
```

Cria um bloco, inicializando todos os seus dados com 0.

(Feita por: Rodrigo Santos).

**Listing 56. Função inserir\_registro\_bloco**

```
Bloco* inserir_registro_bloco(Bloco* bloco, Registro*  
    registro)
```

Insere um registro em um bloco, utilizado no arquivo hashTable.cpp.

(Feita por: Rodrigo Santos).

**Listing 57. Função deletar\_bloco**

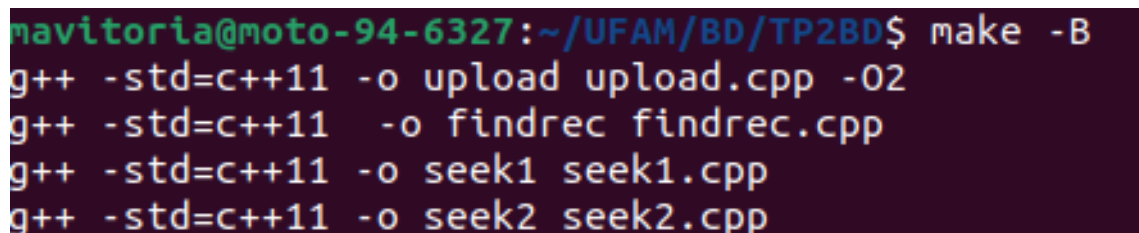
```
void deletar_bloco(Bloco* bloco)
```

Deleta um bloco, utilizado para liberar memória.

(Feita por: Rodrigo Santos).

## 4. Apêndice

Apenas como um extra, disponibilizamos imagens dos programas em funcionamento.

A terminal window with a dark background and light-colored text. The prompt is 'navitoria@moto-94-6327:~/UFAM/BD/TP2BD\$'. The command 'make -B' has been executed, resulting in four lines of output: 'g++ -std=c++11 -o upload upload.cpp -O2', 'g++ -std=c++11 -o findrec findrec.cpp', 'g++ -std=c++11 -o seek1 seek1.cpp', and 'g++ -std=c++11 -o seek2 seek2.cpp'.

```
navitoria@moto-94-6327:~/UFAM/BD/TP2BD$ make -B  
g++ -std=c++11 -o upload upload.cpp -O2  
g++ -std=c++11 -o findrec findrec.cpp  
g++ -std=c++11 -o seek1 seek1.cpp  
g++ -std=c++11 -o seek2 seek2.cpp
```

**Figura 2. Comando de Makefile**

```

● mavitoria@moto-94-6327:~/UFAM/BD/TP2BD$ ./upload artigo.csv
Iniciando arquivo Hash.
Iniciando árvores.
Lendo arquivo CSV.
Adicionando registros no arquivo Hash.
Criando arquivo de índice primário
Criando arquivo de índice secundário
A media em bytes de um registro eh de: 518.91

```

Figura 3. Upload

```

● mavitoria@moto-94-6327:~/UFAM/BD/TP2BD$ ./findrec 9747
Quantidade de Blocos Lidos: 1
Quantidade de Blocos Totais: 225000
Id: 9747
Titulo: GPSO: An improved search algorithm for resource allocation in cloud databases.| Ano: 2013
Autores: Radhya Sahal|Sherif M. Khattab|Fatma A. Omara
Citacoes: 3
Atualizacao: 2016-09-14 12:45:41
Snippet: GPSO: An improved search algorithm for resource allocation in cloud databases. R Sahal, SM Khattab, FA Omara
- Computer Systems and , 2013 - ieexplore.ieee.org. Abstract The Virtual Design Advisor (VDA) has addressed the probl
em of optimizing the performance of Database Management System (DBMS) instances running on virtual machines that share
a common physical machine pool. In this work, the search algorithm ..

```

Figura 4. Findrec

```

● mavitoria@moto-94-6327:~/UFAM/BD/TP2BD$ ./seek1 874234
Quantidade total de blocos do arquivo de índice primário: 766046
Id: 874234
Titulo: Comparison of Spectrum Sharing Techniques for IMT-A Systems in Local Area Networks.| Ano: 2009
Autores: Luis Gacia|Yuanye Wang|Simone Frattasi|Nicola Marchetti|Preben E. Mogensen|Klaus I. Pedersen
Citacoes: 8
Atualizacao: 2016-11-05 06:34:11
Snippet: Comparison of spectrum sharing techniques for IMT-A systems in local area networks. L Gacia, Y Wang, S Fratta
si, N Marchetti - Technology , 2009 - ieexplore.ieee.org. Abstract The explosive growth of mobile communications on o
ne hand and the overly crowded and expensive spectrum on the other hand have fueled hot debates on spectrum sharing te
chniques, anticipating fundamental changes in spectrum regulation. A key point ..

```

Figura 5. Seek1

```

● mavitoria@moto-94-6327:~/UFAM/BD/TP2BD$ ./seek2 "Secrecy outage of a two-user slow fading broadcast channel."
Quantidade total de blocos do arquivo de índice secundário: 498526
Id: 76253
Titulo: Secrecy outage of a two-user slow fading broadcast channel.| Ano: 2014
Autores: Bo Wang|Pengcheng Mu|Huiming Wang|Qinye Yin
Citacoes: 2
Atualizacao: 2016-11-01 22:01:39
Snippet: Secrecy outage of a two-user slow fading broadcast channel. B Wang, P Mu, HM Wang, Q Yin - 2014 IEEE Global ,
2014 - ieexplore.ieee.org. Abstract The secrecy outage of a two-user slow fading broad-cast channel is studied in th
is paper. We consider the scenario where independent and confidential messages are transmitted to each user using supe
rposition coding in the presence of an external ..

```

Figura 6. Seek2

## **5. Agradecimento**

Agradecemos pelos adiamentos concedidos professor, graças a eles conseguimos refinar ainda mais nosso projeto. Obrigado!