**UNIVERSIDADE FEDERAL DE OURO PRETO**

**DECOM**

**Arthur Henrique Santos Celestino 21.1.4019**

**Caio silas de Araujo Amaro 21.1.4111**

**Mateus Viana Maschietto 21.1.4020**

**Henrique Dantas Pighini 21.1.4025**

**Trabalho Prático 1 - Pesquisa Externa**

**Ouro Preto - MG**

**2022**

**Sumário**

[1) Introdução: 3](#_Toc113313719)

[2) Análise Dos Quesitos: 5](#_Toc113313720)

[3) Análise dos Resultados: 17](#_Toc113313721)

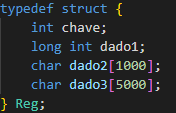
[4) Conclusão: 22](#_Toc113313722)

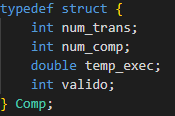
# Introdução:

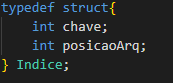
Quando a memória principal não suporta os dados, surge a necessidade do uso da memória secundária. Portanto, foi necessário a criação de métodos utilizados para pesquisa em memória externa, tais como, acesso sequencial indexado, árvore binária, árvore B e árvore B\*. Essa necessidade surge, pois, normalmente se trabalha com um volume de dados muito grande, e apenas ler os dados dos arquivos de maneira simples deixa o processo muito lento, pois o acesso a disco é mais demorado que um acesso a memória interna. Cada um dos métodos citados acima, tem uma estratégia diferente para ler e pesquisar dados em arquivos, visando tornar o processo mais rápido.

A partir disso, com o objetivo de estudar a complexidade dos algoritmos acima, implementamos um código que contém esses.

A criação desses arquivos contou com a utilização das seguintes TADs:

****

****

****

Além disso, como proposto pelo professor e doutor Guilherme Tavares de Assis, o programa foi implementado de maneira que sua execução ocorre através da seguinte linha de comando.



* <método> representa o método de pesquisa externa a ser executado, podendo ser um número inteiro de 1 a 4, de acordo com a ordem dos métodos mencionados;
* <quantidade> representa a quantidade de registros do arquivo considerado;
* <situação> representa a situação de ordem do arquivo, podendo ser 1 (arquivo ordenado ascendentemente), 2 (arquivo ordenado descendentemente) ou 3 (arquivo desordenado aleatoriamente);
* <chave> representa a chave a ser pesquisada no arquivo considerado;
* [-P] representa um argumento opcional que deve ser colocado quando se deseja que as chaves de pesquisa dos registros do arquivo considerado sejam apresentadas na tela.

Com o objetivo de analisar os algoritmos, consideramos os seguintes quesitos:

* Número de transferências (leitura) entre as memórias interna e externa;
* Número de comparações entre as chaves de pesquisa;
* Tempo de execução do código.

Para compilar nosso código, usamos as seguintes diretivas de compilação (usando o terminal do Linux, pois alguns comandos e execução do código apresentam leves erros no Windows):

***$ gcc main.c acessoSequencialIndexado.c arq\_binary.c arvoreB.c arvoreBE.c -o programa -Wall***

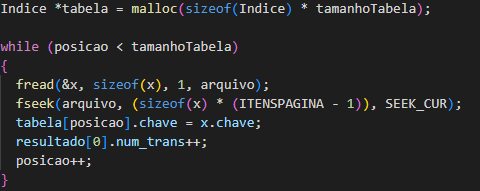
***$ ./programa <metodo> <quantidade> <situação> <chave>***

Para geração dos arquivos, usamos um programa gerador criador por nós mesmos, que se encontra presente na pasta GeraArquivo, simplesmente executando o arquivo .c presente na mesma. Esse arquivo são 3 mains, cada um gerando um dos tipos de arquivo, onde apenas comentamos os 2 que não estavam em uso no momento e executávamos apenas um. Após gerado, jogamos todos os arquivos na pagina do main do programa principal.

# Análise dos Quesitos:

2.1 Acesso sequencial indexado

Este método está ligado a pesquisa sequencial, em que a menor chave de cada página é armazenada no vetor.



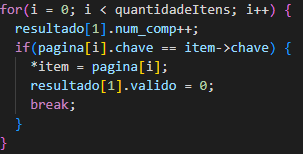
Agora, utilizando o vetor acima, nós somos capazes de encontrar em qual página o item está contido. Essa pesquisa é feita verificando se a chave do item é maior que a chave de menor valor da página. Assim, no momento em que o valor for menor ou igual a pagina que ele está contido terá sido encontrado:



Ademais, com a página achada, nós navegamos até a parte do arquivo que faz referência a essa página e lemos os itens dela (página):



Por fim, passamos por essa página e procuramos o item dentro dela, caso o registro seja achado, o mesmo será retornado por referência pela função:

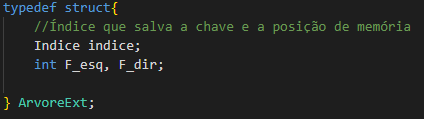


Se não, o valor de valido não será alterado e retornará para o main dizendo que não foi possível encontrar o item.

2.2 Árvore Binária

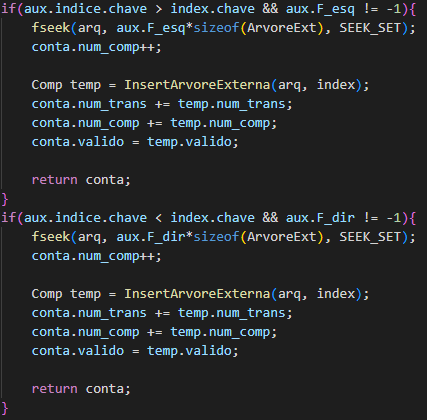
A Árvore de pesquisa binária é um método que irá pegar a ideia de uma árvore binária normal e modificará sua base para ser manipulável em um arquivo.

Para a criação desse método de pesquisa vamos precisar de uma maneira de representar uma árvore em um arquivo binário, e para isto, foi criado uma TAD com elementos necessários para realizar tal necessidade:



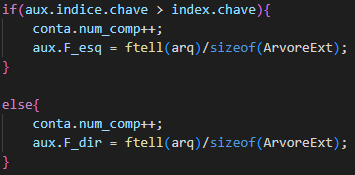
Ao montar a árvore os valores de F\_esq e F\_dir são inicializados com -1 e em seguida é conferido se a chave é maior ou menor para fazer uma chamada recursiva para inserir o índice, se a chave for igual ao índice isso quer dizer que o elemento já foi inserido na árvore.

Para fazer a chamada recursiva utilizamos a função fseek para acessarmos a posição do elemento atual no arquivo binário criado e pegar a posição do filho:



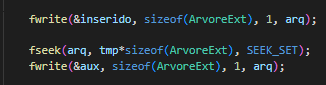
Após a chamada recursiva chegar em um nó em que o filho está inválido (-1) a função não entrará na recursão e vai para a parte que realiza a inserção do elemento no arquivo que representa nossa árvore binária.

Para realizar o que foi dito acima, vamos precisar atualizar o valor do filho do pai como o fim do arquivo (já que a inserção no arquivo é sempre e estritamente no final).

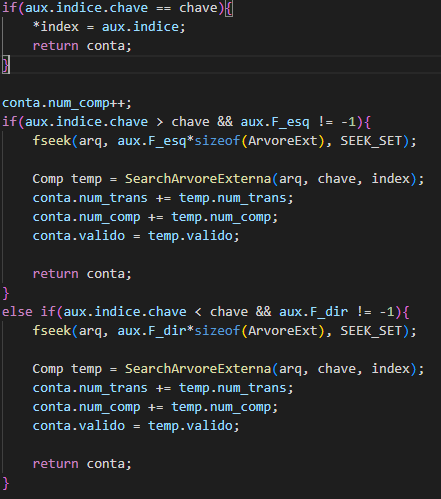


Após alterar os valores internamente, é necessário registrar o novo filho e a posição dele no pai dentro do arquivo.

Usando uma posição previamente registrada do pai, é inserido no fim do arquivo o filho e após darmos um fseek para a posição do pai sobrescrevemos o valor de seus filhos mantendo assim a árvore binária externa correta:



Na função de Search Árvore Externa é feita uma procura pela chave, se a chave procurada for igual ao índice lido da árvore externa, o índice é atualizado e é retornado, se não, é checado se a chave e maior ou menor e o ponteiro do arquivo é atualizado de acordo com isso através do fseek utilizando o valor dos filhos como parâmetro, caso essa chave não seja encontrada significa que ela não está presente na árvore.



2.3 Árvore B

A árvore B é uma árvore de ordem M, que possui como propriedades:

- A página raiz contém entre 1 a 2M itens;

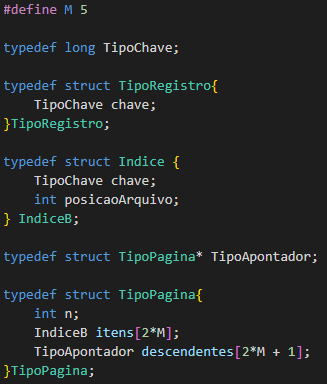
- As demais páginas possuem, no mínimo, M itens e M+1 descendentes e, no máximo, 2M itens e 2M+1 descendentes;

- As páginas folhas estão todas no mesmo nível;

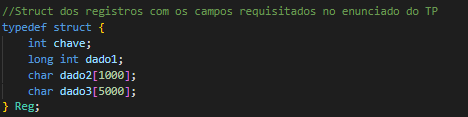
- Os registros estão ordenados de forma crescente dentro das páginas.

O código da árvore B já nos foi disponibilizado nos slides da matéria de Estruturas de Dados II, porém, foram necessárias algumas mudanças para que fosse possível realizar as pesquisas condizentes com o enunciado do trabalho.

A TAD criada para representar a árvore possui um vetor de registros do tipo Índice e um vetor de apontadores do tipo Apontador, além de um contador para guardar a quantidade de registros presentes na página da árvore.



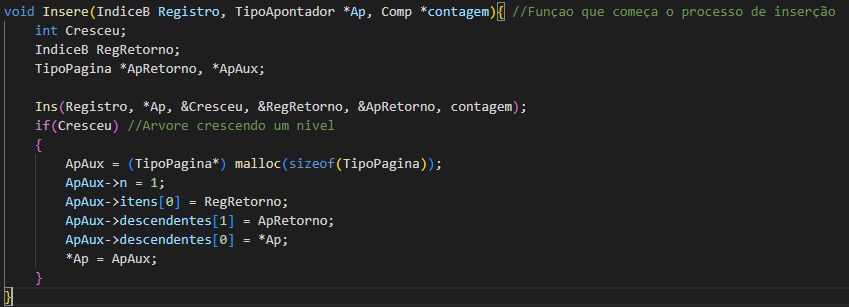
E em vez de utilizarmos o TipoRegistro tivemos que utilizar o tipo Reg que possui os campos requisitados no TP 1 de pesquisa externa:



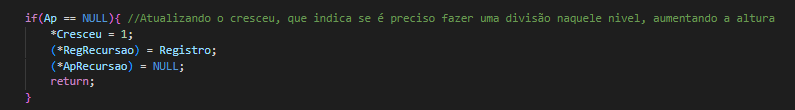
O método de inserção foi dividido em três funções.



A inserção tem início na função insere, que recebe o registro a ser inserido na árvore e o apontador para a página raiz da mesma. Em seguida, há a chamada função Ins que recursivamente acessa a árvore B. Após essa chamada é verificado se a raiz cresceu e se é necessário criar uma nova página:



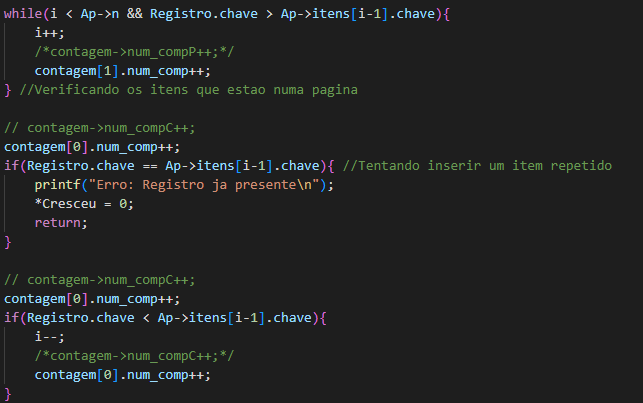
A função Ins procura a posição onde deverá ser inserido o novo registro. Se a árvore for nova e não ter elementos na primeira vez que rodar irá parar neste if e será criado a primeira página da árvore após retornar para a função insere com o cresceu = 1



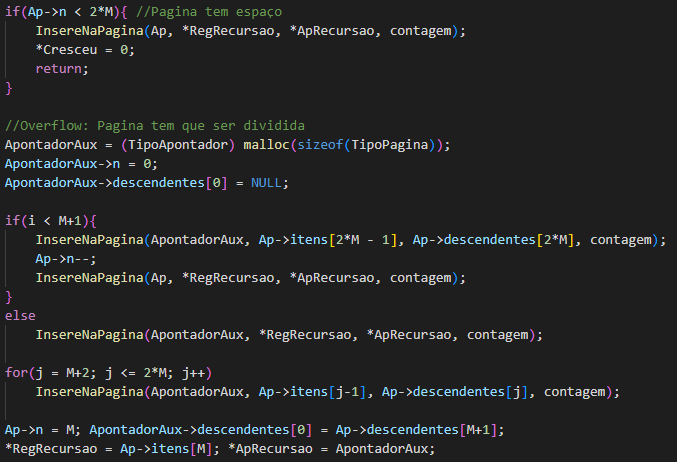
Se tiver mais elementos na árvore será necessário encontrar a posição que vamos inserir o registro.

No código abaixo, a variável i armazena a posição a se inserir o novo registro, ou a página a ser apontada para que se encontre a posição para inserção.

E após encontrarmos tal i vamos verificar se o elemento já existe e se não existir vamos fazer alterações no valor encontrado para direcionarmos o nosso ponteiro.



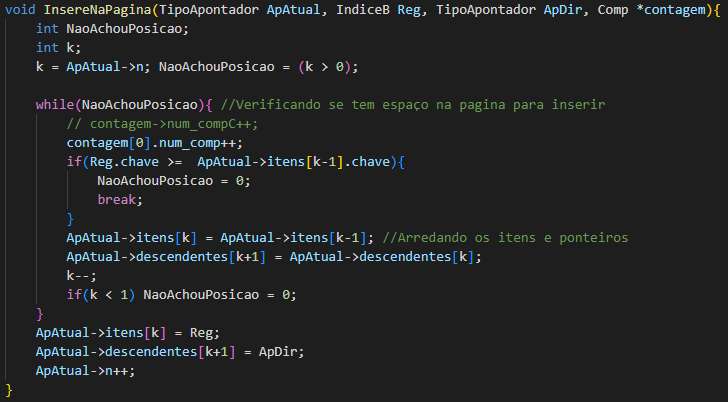
Em seguida, a função realiza chamadas recursivas para encontrar a página folha correta para a inserção do novo registro. Depois, a função verifica se há espaço na página para armazenar o registro. Caso haja espaço na página, o novo registro é inserido nela, chamando a função InsereNaPagina. Caso a página onde deverá ser inscrito o registro estiver cheia, cria-se uma nova página e metade dos registros presentes na página cheia são transferidos para esta nova e o registro do meio é transferido para a página pai das duas páginas:



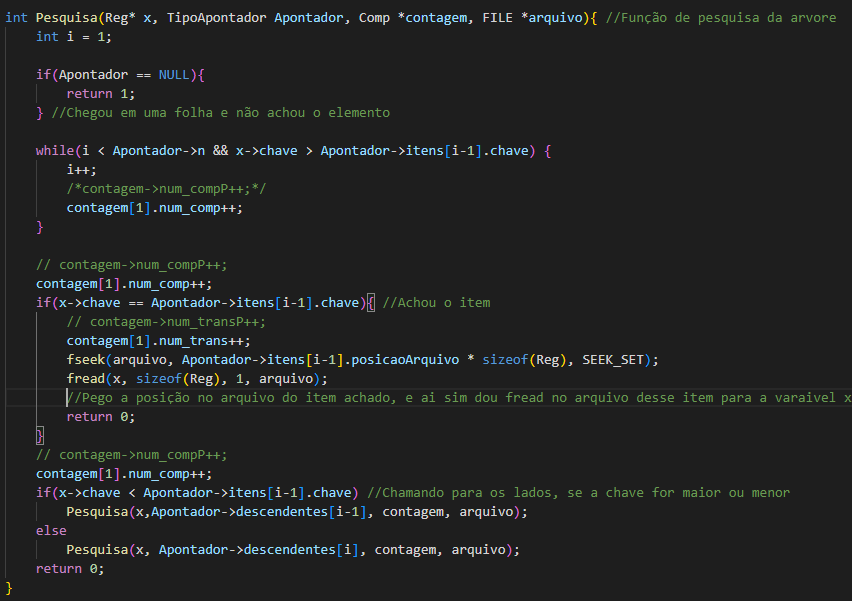
Após ser realizado a inserção os valores de RegRecursao e ApRecusrsao são alterados para que o elemento que vai subir para ser inserido seja facilmente adicionado nos níveis acima.

A função InsereNaPagina insere o novo registro na página enviada como referência.

Essa inserção é realizada de maneira iterativa de maneira que até que seja encontrada a posição que será inserido o novo elemento, vamos empurrando os registros e os ponteiros assim mantendo a propriedade da árvore B e não perdendo os ponteiros



A função pesquisa vai procurar recursivamente o elemento de maneira similar à inserção e vai retornar válido e o elemento se for encontrado ou invalido se não for possível encontrar:

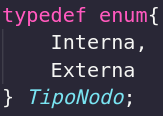


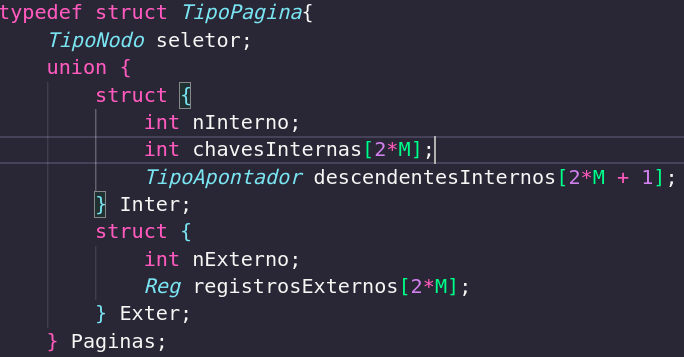
2.4 ÁRVORE B\*

O método árvore B\* é uma melhoria da árvore B, com algumas características diferentes. Será mantido a ideia do limite de valores em uma página e descendentes, porém, teremos páginas internas e externas na nossa árvore.

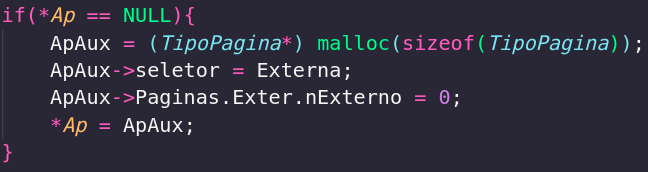
As páginas internas de uma árvore B\* funcionam exatamente como as páginas de uma árvore B, entretanto, é nas páginas externas que o diferencial desse método de pesquisa externa será relevante. Os itens úteis de uma B\* serão todos encontrados em suas folhas que serão externas, isso significa que todos os itens que você deseja procurar estarão em uma página externa e que sua busca só irá parar quando for encontrado a mesma. Desta forma, a árvore B\* possuirá todos os registros do arquivo em memória interna, porém, será fácil acessar a pesquisa já que a mesma possuirá a estrutura de uma árvore B.

Devido a esses diferenças, tivemos que implementar novas funções e refatorar outra TAD, elas estão abaixo, respectivamente:

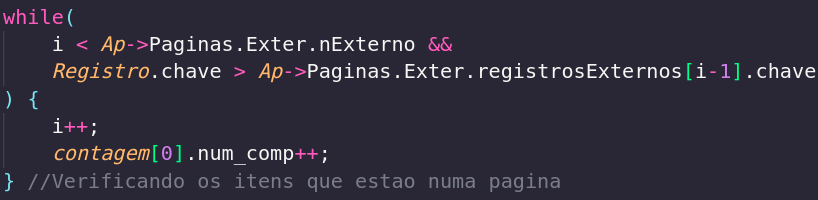




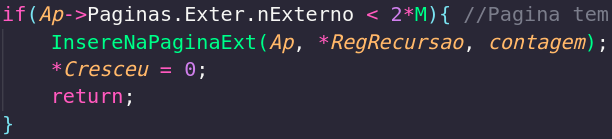
A função de inserção (InsereBE), caso a árvore esteja vazia ele fará primeiro a criação de uma página externa.



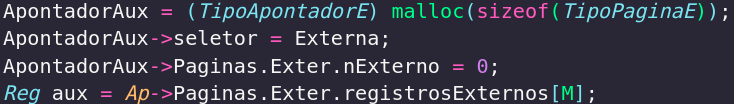
Após isso, é importante lembrar da propriedade das páginas da árvore B\*. Caso ela seja interna seu funcionamento será o mesmo que vimos anteriormente, porém, caso ela seja externa, seu funcionamento será diferente. Nesse sentido, começamos procurando pela posição que o elemento será inserido ou o ponteiro que será acessado na árvore:



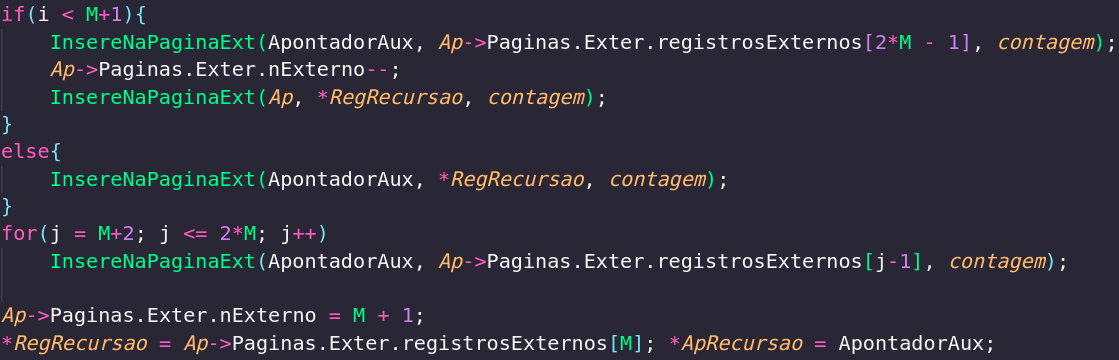
Após chegarmos em uma página externa, verificamos se existe espaço livre para a inserção e se for possível inserir este será inserido sem necessidade de manutenção da árvore:



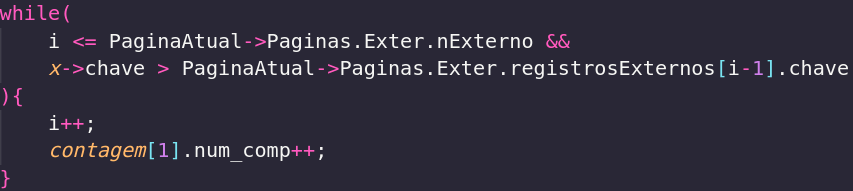
Caso contrário, criaremos uma nova página externa.



E a partir do valor da posição que foi calculado anteriormente será possível identificar se o novo valor será inserido na nova página ou se continuará na mesma. Ademais, os valores que necessitam de ser transferidos para a nova página vão ser inseridos no “for” e os ponteiros que retornaram na recursão para a manutenção da árvore.



Voltando a função principal da árvore B\*, nós chamamos a função que pesquisa dentro da árvore B\*(pesquisaBE). No início ela funciona igual a pesquisaB, até chegar à página folha, onde ela funciona de maneira sequencial.

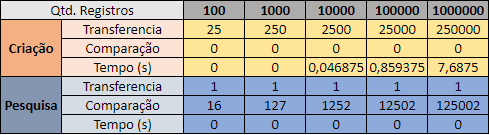


# Análise dos Resultados:

Foram realizados alguns testes para determinar o desempenho de cada método, de forma a conseguir demonstrar maior eficiência de um método em relação a outro:

Pesquisa Sequencial Indexada

A pesquisa sequencial indexada só é possível quando o arquivo está ordenado, assim, só foi possível realizar testes concretos do arquivo em ordem crescente:

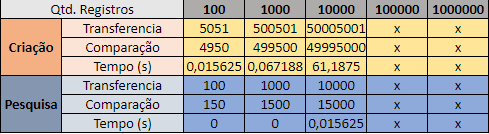


A partir dos dados analisados, percebemos que o tempo de criação cresce rapidamente, considerando que nos primeiros casos o processo foi tão rápido que foi imperceptível.

Além disso, aumenta exponencialmente pois com o aumento do número de páginas, surge o aumento do número de comparações das chaves. Isso durante a pesquisa.

Como o acesso sequencial indexado é limitado à organização ascendente do arquivo, ele se torna menos eficiente, já que ele não funciona com qualquer tipo de arquivo, necessitando de uma organização posterior que vai aumentar o custo da pesquisa.

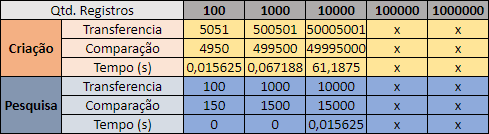
Árvore Binária Com Arquivo Ascendente



Como a árvore binária não consegue se manter balanceada, ela se torna menos benéfica em arquivos que foram ordenados de modo crescente ou decrescente, já que ao pesquisar por um valor ela faz uma pesquisa quadrática, o que não é a intuição ao usar uma árvore binária.

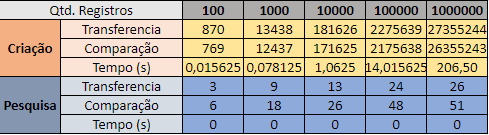
Para concluir, é possível notar que nos casos de entrada extensos não foi possível concluir o código, já que ele demandou mais que a máquina que o estava compilando era capaz de processar.

Árvore Binária Com Arquivo Descendente



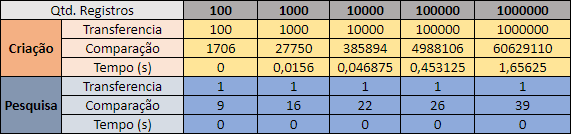
De maneira análoga a árvore binária com arquivo ascendente, ele mantém os mesmos defeitos tanto que as tabelas de ambos são iguais

Árvore Binária Com Arquivo Aleatório



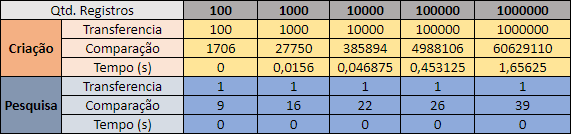
No caso em que a entrada contém valores não ordenados, o método é muito mais eficiente, já que, mesmo que sem o balanceamento, ela é uma árvore de pesquisa. Nesse sentido, esse método propõe um menor número de comparações e tempo de pesquisa.

Árvore B Com Arquivo Ascendente



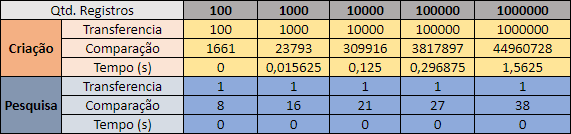
A quantidade de transferências durante a criação dos índices é igual ao número de chaves e, a porcentagem entre o número de chaves e a quantidade de comparações do índice vai aumentando à medida que a quantidade de chave aumenta. Deste modo, para casos maiores fica inviável a reprodução. Com relação às comparações na pesquisa, em todos os casos com árvore B e B\* que virão abaixo, temos uma quantidade baixa pois como a árvore sempre é bem balanceada, vão sendo a cada comparação, cortados uma grande parte dos elementos da mesma.

Árvore B Com Arquivo Descendente



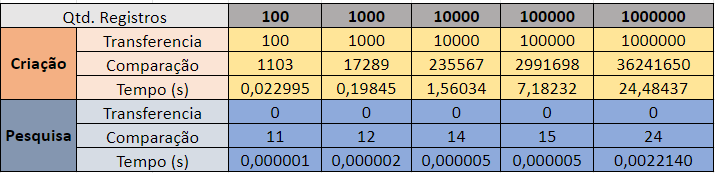
Pôr a árvore B ser balanceada o método ascendente e descendente do código apresenta o mesmo resultado.

Árvore B Com Arquivo Aleatório



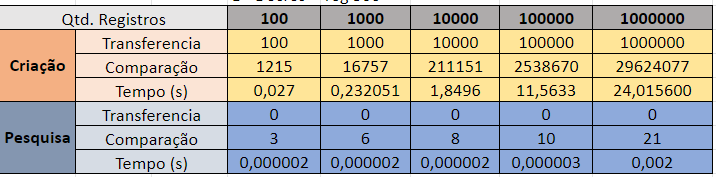
A quantidade de transferências e comparações para a criação da árvore B se mantém alta, mas a pesquisa continua sendo feita de forma eficiente e rápida. Com isso podemos concluir que a natureza do arquivo em questão não é muito relevante para a execução da árvore B, já que o fator de maior influência é a quantidade de registros considerados.

Árvore B\* Com Arquivo Ascendente



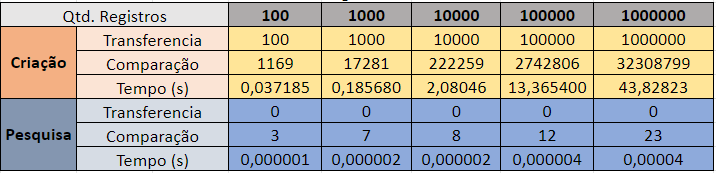
Para a árvore B\*, os resultados foram relativamente parecidos com o da árvore B normal, pelo menos em número de comparações, porém como estamos fazendo a árvore em memória interna, e nesse tipo de árvore os registros tem que ficar nas páginas mais externas, a criação da árvore ficou lenta, já que são registros grandes.

Árvore B\* Com Arquivo Descendente



Para os registros de forma descendente, conseguimos resultados muito semelhantes com relação ao tempo se comparado a aplicação no arquivo ascendente. Na pesquisa tivemos menos comparações porque como nesse tipo de algoritmo, a chave do elemento de uma página externa pode ser a mesma de uma chave da árvore interna, temos algumas comparações a mais de acordo com a implementação escolhida (se o elemento repetido vai para o filho a esquerda ou direita).

Árvore B\* Com Arquivo Aleatório



Por fim, no arquivo aleatório, temos no geral uma média de tempos e comparações razoável, mas que não foge muito dos casos anteriores. Ou seja, na árvore B e B\*, como são balanceadas, a condição do arquivo não influencia tanto quanto nos outros métodos apresentados.

# Conclusão:

Neste trabalho, foi criada uma aplicação capaz de executar 4 métodos diferentes de pesquisa externa, juntamente com 5 quantidades de chaves diferentes (100, 1.000, 10.000, 100.000 e 1.000.000) e 3 métodos de ordenação (crescente, decrescente e aleatório).

No entanto, devido ao grande uso da memória em alguns casos, não foi possível com que o grupo rodasse todos os testes em tempo hábil, tendo alguns membros não conseguindo rodar nenhum dos maiores testes por erro de memória.

Deste modo, podemos tirar conclusões da importância dos métodos de organização e os seus desempenhos. Generalizando, nossa árvore B foi a melhor para as três situações de arquivos com muitos registros, pois ela é sempre balanceada e otimizada, desse modo, sempre vão sendo eliminados metade dos registros (daí sua complexidade logarítmica), gerando assim um tempo total muito pequeno. Tivemos uma discrepância grande entre ela e a B\* pois fizemos a B utilizando apenas os índices, acessando essa posição do item pesquisado no arquivo depois, não precisando inserir o registro inteiro, economizando muita memória e processamento. Para entradas de poucos dados, a árvore binária também se torna bem eficiente, tanto em número de comparações quanto em tempo, porém se torna muito custosa em entradas grandes, já que a cada elemento, precisa ler a árvore toda novamente

Tivemos como principais dificuldades, a implementação da B\*, principalmente quando estávamos fazendo as divisões das páginas externas e passando isso para um outro tipo de página (no caso a interna). Diversas vezes a árvore estava sendo dividida errada, gerando erros de busca e falhas. Também tivemos problemas com a árvore binária, pois como é um método totalmente baseado em memória externa, foi necessário mexer com uma grande quantidade de ponteiros dentro do arquivo, que combinado a uma recursão, nos deixou perdido na leitura e análise do algoritmo.