* **Acesso Sequencial Indexado**

1. Definição de itens por página

Com a intenção de otimizar o método em geral e ilustrar um contexto mais próximo do real, optamos por dinamizar o tamanho das páginas de acordo com o tamanho da entrada. Como o número de registros está bem definido no enunciado do trabalho, não nos preocupamos com grandezas diferentes, haja vista que a verificação de parâmetros ao início da execução impede que esses casos vão adiante. No caso de um arquivo de 1.000.000 registros, o consumo máximo de memória do programa será de aproximadamente 2.9MB(vetor de 500 registros), um valor bastante razoável se comparado à dimensão do arquivo de entrada.

**switch** **(**n\_Registros**)**

**{**

**case** 100**:**

**return** 5**;**

**case** 1000**:**

**return** 50**;**

**case** 10000**:**

**return** 100**;**

**case** 100000**:**

**return** 200**;**

**case** 1000000**:**

**return** 500**;**

**}**

1. Definição da tabela e preenchimento

O código de exemplo fornecido lê cada registro e utiliza uma variável auxiliar para realizar um somatório e selecionar as chaves a serem adicionadas ao índice da tabela. Alteramos essa função de forma que apenas o inteiro referente ao índice daquela página seja lido no arquivo.

int aux**;**

**for** **(**int i **=** 0**;** fread**(&**aux**,** **sizeof(**int**),** 1**,** arquivo\_Binario**);** i**++)**

**{**

tabela\_Indice**[**i**]** **=** aux**;**

fseek**(**arquivo\_Binario**,** **(**itens\_Pagina **\*** **sizeof(**Registro**))** **-** **sizeof(**int**),** SEEK\_CUR**);**

**}**

rewind**(**arquivo\_Binario**);**

Após a leitura do inteiro, o ponteiro, na posição seguinte, é deslocado na quantidade de 1 página – 1 inteiro, posicionando-o sobre a primeira chave da página seguinte. Ao final do processo, o ponteiro é reiniciado na posição inicial do arquivo.

1. Busca pelo índice

Iniciamos tratando casos onde o registro obviamente não se encontra no arquivo. Em um arquivo crescente, por exemplo, se o registro for menor que a primeira chave, então a busca pode ser interrompida imediatamente. Como o enunciado também inclui a possibilidade de arquivos decrescentes, utilizamos a mesma lógica para excluir esse tipo de caso.

**switch** **(**n\_Situacao**)**

**{**

**case** 1**:**

**if** **(**tabela\_Indice**[**0**]** **>** n\_Chave**)**

**return** **-**1**;**

**for** **(**aux **=** 0**;** aux **<** tam\_Tabela**;** aux**++)**

**{**

**if** **(**tabela\_Indice**[**aux**]** **==** n\_Chave**)**

**return** aux**;**

**if** **(**tabela\_Indice**[**aux**]** **>** n\_Chave**)**

**return** aux **-** 1**;**

**}**

**return** aux **-** 1**;**

**break;**

**case** 2**:**

**if** **(**tabela\_Indice**[**0**]** **<** n\_Chave**)**

**return** **-**1**;**

**for** **(**aux **=** 0**;** aux **<** tam\_Tabela**;** aux**++)**

**{**

**if** **(**tabela\_Indice**[**aux**]** **==** n\_Chave**)**

**return** aux**;**

**if** **(**tabela\_Indice**[**aux**]** **<** n\_Chave**)**

**return** aux **-** 1**;**

**}**

**return** aux **-** 1**;**

**break;**

**}**

Em seguida, percorremos a tabela sequencialmente comparando os índices com a chave pesquisada. No caso de um arquivo decrescente, se a chave for maior do que o índice atualmente comparado, então ela só pode estar localizada na página anterior. Se a iteração for concluída, retornamos o índice da última página independentemente, pois se a chave existir no arquivo, então ela só pode pertencer àquela página.

1. Leitura da página e busca em memória principal

Uma vez encontrado o índice da página, calculamos o deslocamento, posicionamos o ponteiro do arquivo de acordo e realizamos a leitura dos dados. Em memória principal, outra pesquisa sequencial é realizada e a iteração é interrompida se a chave pesquisada for encontrada. Note que não há preocupação quanto à possibilidade de página incompleta, pois a dinamização do tamanho da página impede que isso aconteça para as possibilidades de entradas enunciadas no trabalho.

Registro aux**[**itens\_Pagina**];**

long int desloc\_Ponteiro **=** itens\_Pagina **\*** indice\_Pagina **\*** **sizeof(**Registro**);**

fseek**(**arquivo\_Binario**,** desloc\_Ponteiro**,** SEEK\_SET**);**

fread**(**aux**,** **sizeof(**Registro**),** itens\_Pagina**,** arquivo\_Binario**);**

**for** **(**int i **=** 0**;** i **<** itens\_Pagina**;** i**++)**

**{**

**if** **(**aux**[**i**].**chave **==** n\_Chave**)**

**{**

printf**(**"Registro encontrado!\n"**);**

imprimir\_Registro**(**aux**[**i**]);**

**return** 1**;**

**}**

**}**

1. Análise experimental

Para a análise experimental do método, utilizamos a função *clock* da biblioteca *time.h* para realizar o cálculo de tempo de execução dos métodos de pesquisa e variáveis inteiras para somar as comparações e leituras. Tanto as etapas de criação dos índices quanto a própria pesquisa foram contabilizadas, e as chaves pesquisadas foram determinadas por ‘*1 + rand () % (nro\_registros – 1)*’. Observamos que o número de comparações é variável, haja vista que depende do quão distante o índice da página e a própria chave estão em relação ao início dos vetores que as armazenam. Já o número de transferências é constante para o mesmo número de registros, pois a obtenção da tabela de índices efetua (*nro\_registros / tam\_tabela*) transferências de inteiros (chaves) e a pesquisa sequencial requer a transferência de uma página da grandeza de (*itens\_pagina*). Os outputs utilizados para a construção dos gráficos a seguir estão localizados no diretório “análise experimental” do repositório.

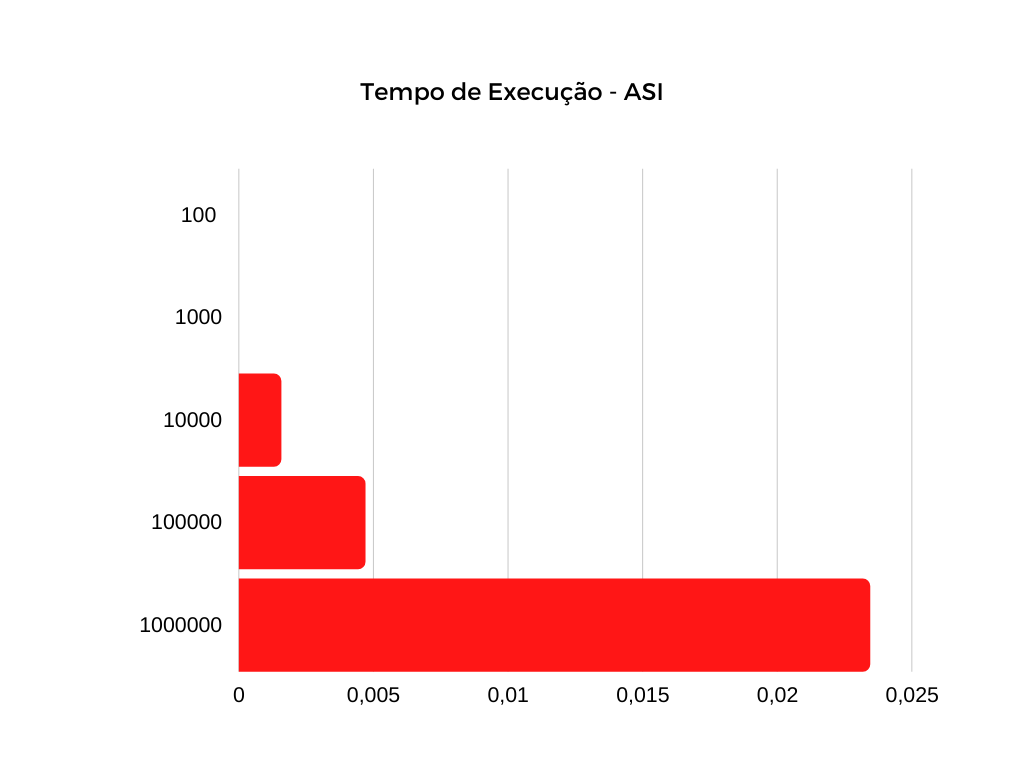


Figura 1 - Número de registros e tempo de execução (em milissegundos)

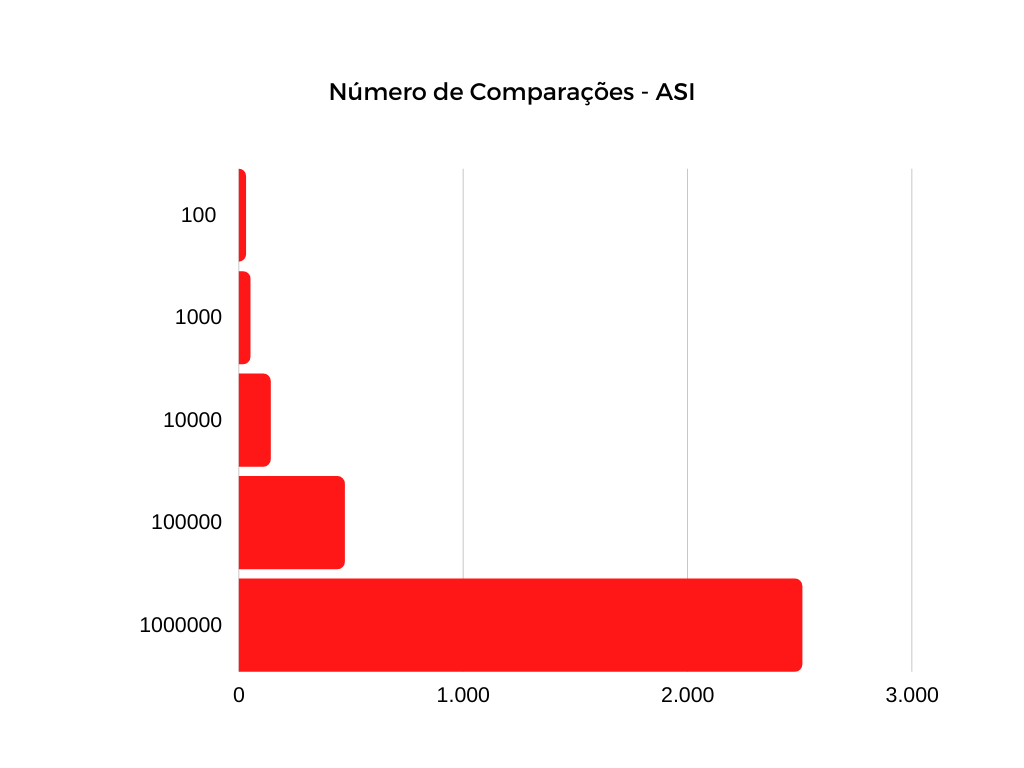


Figura 2 - Número de registros e número de comparações realizadas

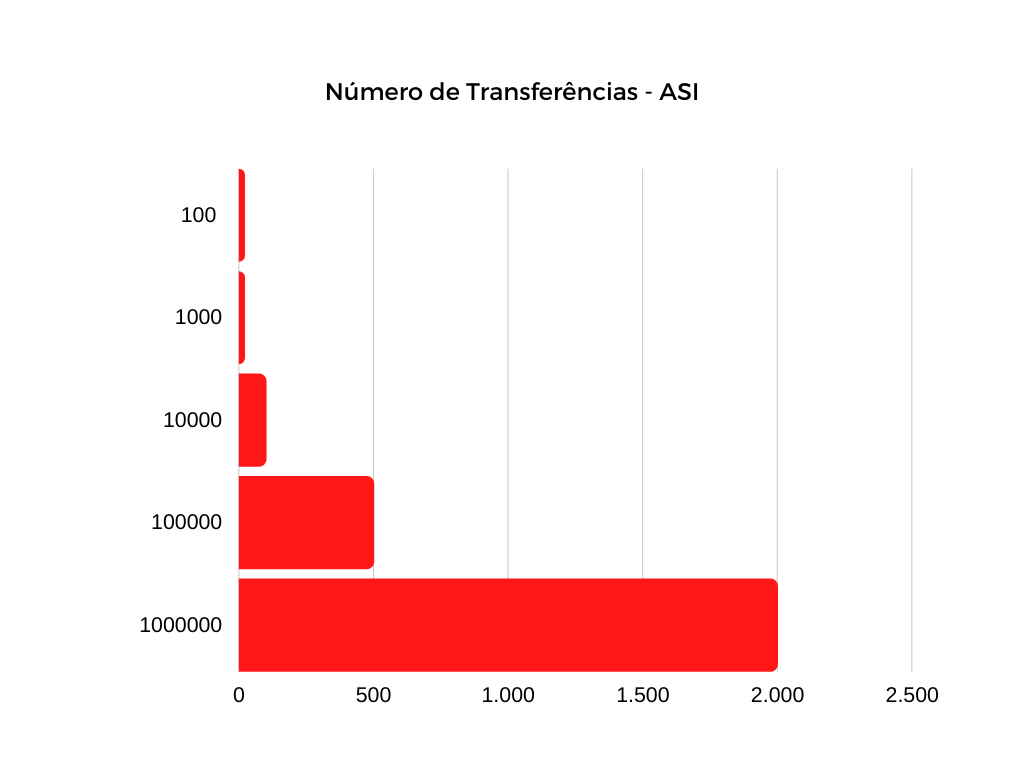


Figura 3 - Número de registros e número de transferências realizadas

* **Árvore B**

1. Pesquisa na Árvore B

Para realizar a pesquisa na Árvore B, primeiro nós comparamos a chave com as outras chaves que estão na página raiz até que a chave seja encontrada ou o intervalo no qual ela se encaixa. Caso a chave não fosse localizada, o apontador iria para a subárvore do intervalo encontrado. Em seguida, o processo era repetido recursivamente até achar a chave ou atingir uma página folha (apontador nulo).

int Pesquisa **(**Registro **\***x**,** Apontador Ap**)** **{**

long i **=** 1**;**

ab\_comparacoes**++;**

**if** **(**Ap **==** **NULL){**

**return** 0**;**

**}**

**while** **(**i **<** Ap**->**n **&&** x**->**chave **>** Ap**->**r**[**i**-**1**].**chave**)** i**++;**

ab\_comparacoes**++;**

**if** **(**x**->**chave **==** Ap**->**r**[**i**-**1**].**chave**){**

**\***x **=** Ap**->**r**[**i**-**1**];**

**return** 1**;**

**}**

ab\_comparacoes**++;**

**if** **(**x**->**chave **<** Ap**->**r**[**i**-**1**].**chave**)** **{**

**return** Pesquisa **(**x**,** Ap**->**p**[**i**-**1**]);**

**}**

**else** **{**

**return** Pesquisa **(**x**,** Ap**->**p**[**i**]);**

**}**

**return** 0**;**

**}**

1. Caminhamento em Árvore B

Realizamos o caminhamento de maneira recursiva, passando o apontador inicial da árvore, caso o apontador seja nulo, nada será impresso. Assim o loop será acionado, e enquanto a condição do while for aceita, cada item da primeira página vai ser selecionado, e em seguida, imprimindo seus filhos à esquerda, então todo processo é por ele (caminhamento à esquerda). Se o valor de i for diferente da quantidade de itens, então será impressa a chave daquela posição.

void Imprime**(**Apontador arvore**)** **{**

int i **=** 0**;**

ab\_comparacoes**++;**

**if** **(**arvore **==** **NULL)**

**return;**

**while** **(**i **<=** arvore**->**n**)** **{**

Imprime**(**arvore**->**p**[**i**]);**

ab\_comparacoes**++;**

**if** **(**i **!=** arvore**->**n**)**

printf**(**"'%d' \n"**,** arvore**->**r**[**i**].**chave**);**

i**++;**

**}**

printf**(**"\n"**);**

**}**

1. Inserção em Árvore B

Para o item ser inserido, primeiro nós localizamos a página onde ele deve ser inserido, caso encontre uma página válida (menos de **2m** itens), ele é inserido nela. Caso encontre uma página que esteja cheia, outra página é criada para a divisão de itens, caso a página pai também esteja cheia, a divisão se propaga.

// A página tem espaço.

ab\_comparacoes**++;**

**if** **(**Ap**->**n **<** MM**){**

InsereNaPagina **(**Ap**,** **\***RegRetorno**,** **\***ApRetorno**);**

**\***Cresceu **=** 0**;**

**return;**

**}**

// A página tem que ser dividida (Overflow)

ApTemp **=** **(**Apontador**)**malloc**(sizeof(**Pagina**));**

ApTemp**->**n **=** 0**;**

ApTemp**->**p**[**0**]** **=** **NULL;**

ab\_comparacoes**++;**

**if** **(**i **<** M**+**1**){**

InsereNaPagina**(**ApTemp**,** Ap**->**r**[**MM**-**1**],** Ap**->**p**[**MM**]);**

Ap**->**n **--;**

InsereNaPagina**(**Ap**,** **\***RegRetorno**,** **\***ApRetorno**);**

**}** **else**

InsereNaPagina**(**ApTemp**,** **\***RegRetorno**,** **\***ApRetorno**);**

**for** **(**j **=** M **+** 2**;** j **<=** MM**;** j**++)** **{**

InsereNaPagina**(**ApTemp**,** Ap**->**r**[**j**-**1**],** Ap**->**p**[**j**]);**

**}**

Ap**->**n **=** M**;**

ApTemp**->**p**[**0**]** **=** Ap**->**p**[**M**+**1**];**

**\***RegRetorno **=** Ap**->**r**[**M**];**

**\***ApRetorno **=** ApTemp**;**

Depois, no pior caso, o processo de divisão pode propagar-se até a raiz da árvore B e, assim, sua altura aumenta. (Única forma de aumentar a altura de uma árvore B: divisão da raiz).

void Ins **(**Registro Reg**,** Apontador Ap**,** short **\***Cresceu**,** Registro **\***RegRetorno**,** Apontador **\***ApRetorno**){**

long i **=** 1**;** long j**;**

Apontador ApTemp**;**

ab\_comparacoes**++;**

**if** **(**Ap **==** **NULL)** **{**

**\***Cresceu **=** 1**;**

**(\***RegRetorno**)** **=** Reg**;**

**(\***ApRetorno**)** **=** **NULL;**

**return;**

**}**

Ins**(**Reg**,** **\***Ap**,** **&**Cresceu**,** **&**RegRetorno**,** **&**ApRetorno**);**

**if** **(**Cresceu **==** 1**)** **{** // Caso a árvore tenha crescido

ApTemp **=** **(**TipoPagina **\*)** malloc **(sizeof(**TipoPagina**));**

ApTemp**->**n **=** 1**;**

ApTemp**->**r**[**0**]** **=** RegRetorno**;**

ApTemp**->**p**[**1**]** **=** ApRetorno**;**

ApTemp**->**p**[**0**]** **=** **\***Ap**;**

**\***Ap **=** ApTemp**;**

**}**

1. Execução da Árvore B

Primeiramente, fizemos um loop - que se encerra ao final do arquivo criado -, que tem a função de contar a quantidade de transferências realizadas, além de inserir os registros na árvore.

**while** **(!**feof**(**arquivo\_binario**))** **{**

ab\_transferencias**++;**

fread**(**reg**,** **sizeof(**Registro**),** 1**,** arquivo\_binario**);**

Insere**(\***reg**,** **&**arv**);**

**}**

Depois, nós fazemos a pesquisa na Árvore B e criamos uma condição para quando o registro for encontrado, o usuário será avisado imprimindo uma mensagem, além da quantidade de transferências e comparações que foram realizadas durante todo o método de execução. Caso a condição não for ativada, ou seja, o registro não ser encontrado, é retornado o valor de 0.

ab\_comparacoes**++;**

**if** **(**Pesquisa**(**pesquisa**,** arv**))** **{**

printf**(**"Registro encontrado!\n"**);**

printf**(**"Nº de transferências: %d\n"**,** ab\_transferencias**);**

printf**(**"Nº de comparações: %d\n"**,** ab\_comparacoes**);**

**if** **(**print\_registro**)**

imprimir\_registro**(\***pesquisa**);**

**return** 1**;**

**}**

**return** 0**;**

1. Análise experimental

Para a análise experimental do método, utilizamos a função *clock* da biblioteca *time.h* para realizar o cálculo de tempo de execução dos métodos de pesquisa e variáveis inteiras para somar as comparações e leituras. Tanto as etapas de criação dos índices quanto a própria pesquisa foram contabilizadas, e as chaves pesquisadas foram determinadas por ‘*1 + rand () % (nro\_registros – 1)*’.

Os testes foram feitos alternando o registro e os arquivos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente). Os outputs utilizados para a construção dos gráficos a seguir estão localizados no diretório “análise experimental” do repositório. Infelizmente, não foi possível realizar os testes de 1.000.000 de registros, pois o computador não rodou em nenhuma situação.

1. Tempo de execução

O tempo de execução aumenta consideravelmente, de forma proporcional à quantidade de registros dentro de um arquivo, entretanto, por se tratar de uma Árvore B, ela é sempre balanceada. Então, independentemente da maneira em que os registros estiverem ordenados, o tempo praticamente não se altera, considerando a mesma quantidade de registros nos 3 casos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente).

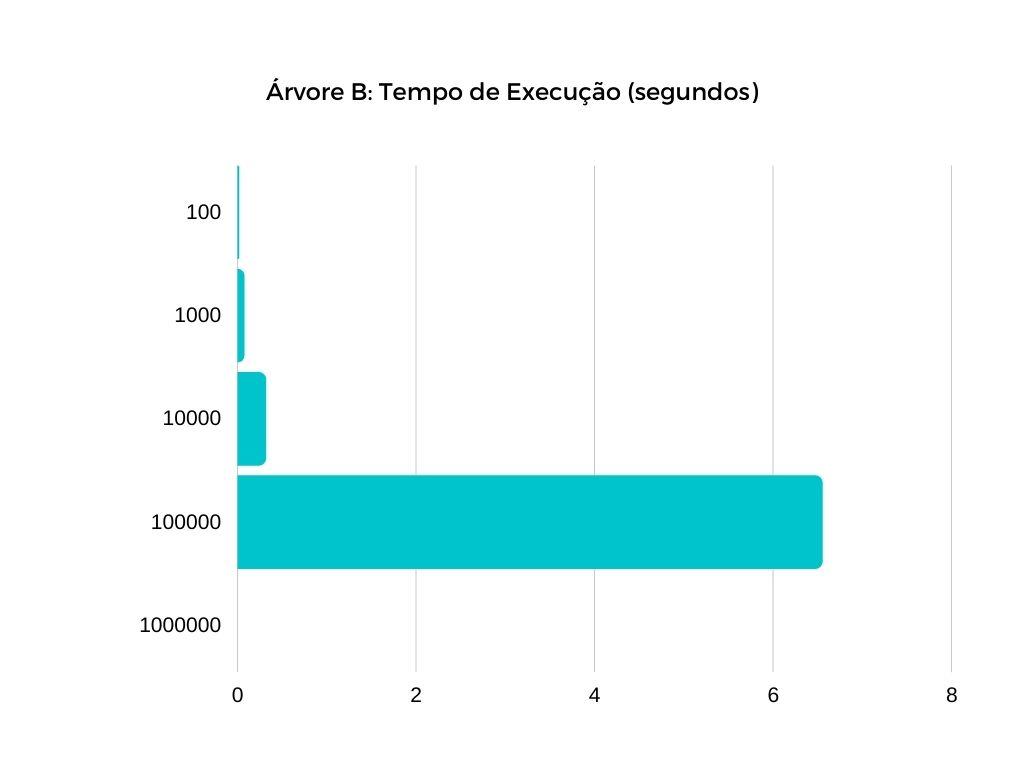


Figura 4 - Número de registros e tempo de execução (em segundos)

1. Quantidade de Comparações

Assim como o tempo de execução, a quantidade de comparações aumenta consideravelmente, de forma proporcional à quantidade de registros dentro de um arquivo, entretanto, por se tratar de uma Árvore B, ela é sempre balanceada. Então, independentemente da maneira em que os registros estiverem ordenados, o número de comparações praticamente não se altera, considerando a mesma quantidade de registros nos 3 casos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente).

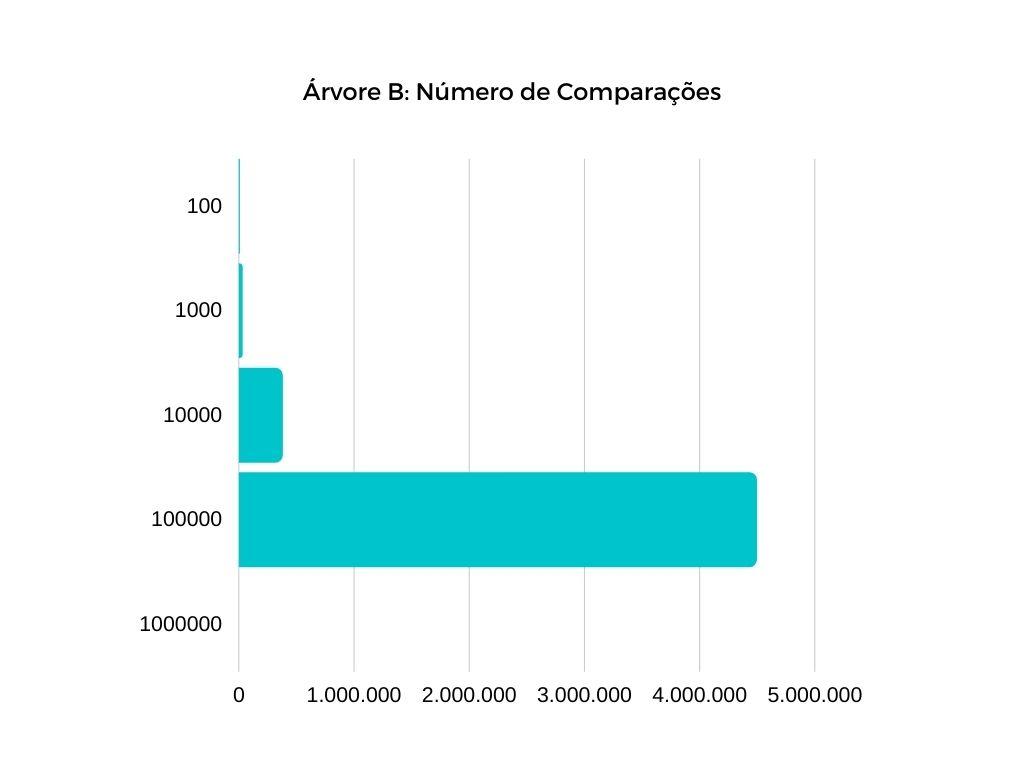


Figura 5 - Número de registros e comparações realizadas

1. Quantidade de Transferências

O número de transferências é sempre constante, independentemente da maneira do teste, ela sempre será igual a quantidade de registros na Árvore B.

