Universidade de São Paulo

EACH – Escola de Artes, Ciências e Humanidades

Relatório referente ao Exercício Programa II

Algoritmos e Estruturas de Dados II

Felipe Brigatto	7972602
Igor Oliveira Borges	8122442
José Roberto Ventaja	7972290
Marcelo Gaioso Werneck	8061963

São Paulo

2014

Sumário

Introdução

Este trabalho consiste em implementar a busca em registros de três maneiras distintas: busca linear, busca binária e busca por índices. Assim, será realizado uma análise dos benefícios obtidos com cada técnica.

Os registros possuem tamanho fixo de 128 *bytes*, sendo 4 *bytes* para o identificador e 124 para conteúdo. As implementações de leitura e escrita foram feitas em blocos de 4096 *bytes*, mantendo-se no máximo 2000 blocos em memória física.

Na busca linear, os registros são varridos sequencialmente até que o registro desejado seja encontrado, portanto sua complexidade é $O(n)$.

Para a busca binária é necessária a ordenação dos registros, que foi feita através do k-way merge, onde os registros são divididos em k blocos na memória secundária para que cada parte caiba na memória e possibilite o *merge sort* das partes.

A busca binária se baseia em divisão e conquista, consiste em dividir a totalidade dos registros no meio, se o registro do meio entre as duas metades tiver o identificador menor que o procurado pega-se a metade maior, se o identificador for maior que o procurado pega-se a metade menor, esse procedimento se repete até que o registro do meio tenha o identificador procurado. A busca binária possui complexidade $O(\log n)$.

A busca indexada é realizada a partir de acesso direto e para isto se faz necessário uma estrutura auxiliar que guardará os índices para que o acesso posteriormente seja em tempo constante, são gerados dois índices um em relação ao registro (denso) e o outro relativo ao bloco do registro (esparso).

Descrição dos Algoritmos

Para a implementação dos algoritmos de buscas, um levantamento bibliográfico foi necessário para entender a lógica e então dar sequência ao trabalho. A seguir resumos das buscas utilizando k-way merge, linear, binária e por índice.

K-Way Merge Sort

Para os algoritmos de busca binária e busca por índice, se faz necessário que os registros estejam ordenados, tal ordenação foi feita pelo K-Way Merge Sort. A técnica de ordenação é interessante porque há os limites impostos pela memória que pode não comportar todos os registros e isso é levado em conta pelo algoritmo. Os registros são divididos em “K” blocos, sendo esse número “K” aquele que resulta “n/K” no maior número possível de registros que cabem em memória, onde “n” é o número de registros. Em cada bloco é realizado um *merge sort*, que consiste em um método de divisão e conquista no qual os registros são divididos em unidades menores de forma recursiva até que a ordenação seja trivial e no caminho inverso junta todos os pedaços ordenando-os.

Busca Linear

Na busca linear, todos os registros são varridos de forma sequencial até que a chave de identificação procurada seja encontrada. Considerando que “n” seja o número de registros armazenados, a complexidade deste algoritmo de busca é $O(n)$, o que significa que, no pior caso (se o registro desejado for o último a ser encontrado), serão necessárias “n” comparações para que a busca seja concluída.

Busca Binária

Na busca binária, um método de divisão e conquista, precisamos de um conjunto de registros ordenados. Verificamos se o registro do meio desse conjunto tem identificador maior ou menor do que estamos procurando, se for maior pegamos a primeira metade dos registros, se for menor pegamos a segunda metade dos registros. Com esse novo conjunto de registros realizamos o mesmo procedimento, até que o esse "registro do meio" seja o registro que procuramos. Considerando que "n" seja o número de registros armazenados, a complexidade desse algoritmo é *O*(log n), o que significa que, no pior caso (se estiver em uma das pontas), serão necessárias "log n" comparações para que a busca seja concluída.

Busca por Índice

A busca por índices é feita a partir de um índice com dois níveis, "base" e "topo", no nível "base" existe uma entrada para cada registro de arquivo no sistema, já no nível "topo" existem entradas apenas para a primeira entrada de cada bloco do índice "base". Como as entradas de índice possuem 8 bytes, o número de entradas "topo" é 512 vezes menor que "base", já que os blocos são de 4096 bytes.

Para a geração dos índices, é criado um arquivo com entradas relativas ao índice base, uma para cada registro, esse arquivo é ordenado e posteriormente é gerado o índice topo, percorre-se cada bloco do índice base e gera uma nova entrada no índice topo para cada entrada no bloco com o *id* e *offset* do índice base.

Para a leitura, primeiramente é feita a verificação no índice topo afim de encontrar o bloco do índice base que deve ser verificado. Então é feita a verificação no índice base, se o registro existir obtém seu *offset*. Com o *offset* é realizada a leitura do arquivo de registro.

Descrição do Experimento

Começamos a codificação do arquivo “principal.c, que é responsável pela leitura de “config.txt” automaticamente (sem precisar como parâmetro na execução pela linha de comando no terminal) e criação da estruturas especificadas em “estruturas.h”.

Dado este inicio, o relatório começou a ser escrito, deixando pronto introdução e preparando os itens para resultado e conclusões.

O primeiro passo para a realização do trabalho foi a ‘preparação’ do método para sobre a entrada dos parâmetros, sendo implementado no main() a leitura do arquivo -- usado como “config.txt” -- e armazenando suas variáveis para suas devidas finalidades. Arquivos de entrada, saida, id’s a serem buscados, estão incluídos nesse trabalho.

Com tudo isso, começando pelo mais fácil, passamos a implementar a busca sequencial, consequentemente a aprender a realizar leitura do arquivo em blocos. De inicio foi um pouco difícil para captar a ideia, e entender o conceito junto com o algoritmo por trás. Mas nada que muito esforço fosse incapaz de finalizar essa tarefa.

Uma vez que a busca sequencial estava pronta, e funcionando, partimos a realizar a busca binária porém, para ela poder rodar, deve-se primeiro ordenar o arquivo de registros, então na verdade fomos pesquisar, e implementá-lo.

O problema, principal problema do programa, foi entender e implementá-lo de forma correta. Foi uma tarefa extremamente dificultosa, que levou muito tempo. Quando vimos que essa tarefa estava demorando muito, dividimos para fazer paralelamente a busca por índices, ou melhor, gerar os índices para realizar a busca.

Porém, mesmo quando parte da geração de índices foi feita, ou talvez até ela totalmente, ainda era necessário a parte de ordenação para ordenar o índice denso, afim de testar a busca por índices. E novamente ficamos entalados nessa tarefa. Até um certo momento em que a data de entrega do trabalho estava próxima, e para ter pelo menos a busca binaria garantida, parou-se um pouco a ordenação e foi feita a busca binaria.

Para os testes da busca binaria, foi utilizado um arquivo de registros ordenado emprestado de um colega de classe. Uma vez que achou corretamente os valores,

voltamos para a ordenação. O triste disso é que a data de entrega já estava muito próxima. Para nossa sorte, da noite para o dia da entrega, conseguimos finalizar a ordenação.

O problema final que surgiu foi ter que adaptá-la para não ordenar mais registros, mas sim índices também. Podemos colocar um booleano como parâmetro do método junto com vários if's, porem iria ficar complicado e mais difícil de achar os devidos erros. Sendo assim, copiamos em outros métodos parecidos com o nome de variáveis diferentes, e utilizando os métodos de índices ao invés de registros.

O tempo e a dificuldade de um elemento principal foi o bastante para atrapalhar o resto do trabalho, como resultado. A busca em largura foi implementada, porém não pode ser testada na hora, apenas quando estava já no final do prazo de entrega. E infelizmente ela não funcionou de primeira, mas por algum errinho (do qual todos achamos que seja) não foi possível rodar essa parte do trabalho, e completar a comparação das buscas.

Falta de tempo, falta de experiência, baixa cooperação do grupo entre nós mesmos, dificuldade do trabalho, ou mesmo falta de persistência, não sabemos exatamente o que foi nos fez levar a essa pequena falha (ou falta) do trabalho geral, mas sabemos que foi com muito esforço que chegamos com o que temos, muitas horas sem dormir na frente dos algoritmos, e principalmente, muita dedicação.

As especificações de cada arquivo-fonte está a baixo.

Código e Entrada

Nosso projeto é composto dos seguintes arquivos:

- "create_data.h" / "create_data.c": contém as propriedades e métodos de criação das estruturas a serem armazenadas tanto registro quanto índice;
- "buscas.h" / "buscas.c": implementa as três buscas solicitadas: Linear, Binária e Indexada e define o tipo registro de acordo com o enunciado.
- "principal.c": contém o método principal que chamará a execução tendo como fonte o arquivo "config.txt" que deve ser previamente alterado, antes da execução.

- “estrutura.h”: contém as estruturas necessárias para a execução do programa, definindo os tipos “TItem”, “TChave” e “TDicionario”.
- “config.txt”: contém as instruções a serem executadas pelo programa no formato proposto pelo enunciado.

Testes e Resultados

Os testes foram realizados com 2000000, 4000000 e 8000000 registros, e são detalhados a seguir:

Gerando aleatoriamente 2000000 registros com a ferramenta disponibilizada e buscando 9 ids (753763427, 1298938, 1692741001, 2201, 28877013, 1792702375, 666, 38209744 e 478070300) nas buscas:

Linear:

```
<753763427>    <SIM>    <(753763427)
nascljsqjdhvtnhhxizecnlnctyhxcgmdyrqjlgsqnnjdwqaerhgfswjoutnyz
c>    <1791>    <0.000204s>
<1298938>    <NAO>    < >    <1792000>    <0.112877s>
<1692741001>    <SIM>    <(1692741001)
ofaugvlbsdfrombcbkvubfakkfooxwmlbphjksn>    <35071>
<0.001984s>
<2201>    <NAO>    < >    <1792000>    <0.104239s>
<28877013>    <SIM>    <(28877013)
gtdusizrohwpylucbvnxmccuzvnbyzsgswcmebgujckhoelrbzondqhdlxglwy
truvfazlxiohrenfxogndjdkmr>    <5767>    <0.000320s>
<1792702375>    <NAO>    < >    <1792000>    <0.103463s>
<666>    <NAO>    < >    <1792000>    <0.105692s>
<38209744>    <SIM>    <(38209744)
jwwwpuyrvleegabyqugoaoeignucwxkhvifkcfcxqieyihyabfrdvvmclggjfs
```

saaxnfcrcuziukpvmsadwxzkbmskvxcpzf> <17278> <0.000955s>
<478070300> <SIM> <(478070300)
ymduohftiocdupgascuagvouphclnrxngbkukppsgsycj> <9959>
<0.000485s>

Binária:

<753763427> <SIM> <(753763427)
nascljsqjdhvtnhhxizecnlnctyhxcgmdyrqjlgsqnnjdwqaerhgfswjoutnyz
c> <1337> <0.000115s>
<1298938> <NAO> < > <864> <0.000061s>
<1692741001> <SIM> <(1692741001)
ofaugvlbsdfrombcbkvubfakkfooxwmlbphjksn> <2074>
<0.000144s>
<2201> <NAO> < > <512> <0.000034s>
<28877013> <SIM> <(28877013)
gtdusizrohwpylucbvnxmccuzvnbyzsgswcmebgujckhoelrbzondqhdlxglwy
truvfazlxiohrenfxogndjdkmr> <678> <0.000044s>
<1792702375> <NAO> < > <2656> <0.000170s>
<666> <NAO> < > <512> <0.000031s>
<38209744> <SIM> <(38209744)
jwwwpuyrvleegabyqugoaoeignucwxkhvifkcfcxqieyihyabfrdvvmclggjfs
saaxnfcrcuziukpvmsadwxzkbmskvxcpzf> <1168> <0.000069s>
<478070300> <SIM> <(478070300)
ymduohftiocdupgascuagvouphclnrxngbkukppsgsycj> <1569>
<0.035288s>

Os valores mínimo, máximo e médio do tempo de execução e tamanho médio dos blocos para as buscas foram:

BUSCA	Tempo Mínimo	Tempo Médio	Tempo Máximo	Média de

				Blocos Lidos
Linear	0,000208s	0,0478021111 s	0,105692s	804207,33
Binária	0,000031s	0,0039951111 s	0,035288s	1263,33
Indexada	não realizado	não realizado	não realizado	não realizado

Para 4000000 registros, buscou 9 ids (753763427, 1298938, 1692741001, 2201, 28877013, 1792702375, 666, 38209744 e 478070300).

Linear

```
<753763427>    <SIM>     <(753763427)
nascljsqjdhvtnhhxizecnlnctyhxcgmdyrqjlgsqnnjdwqaerhgfswjoutnyz
c>   <1791>    <0.000128s>
<1298938> <NAO>     < >  <128000000>    <9.076764s>
<1692741001>   <SIM>     <(1692741001)
ofaugvlbsdfrombcbkvubfakkfooxwmlbphjksn>     <35071>
<0.002154s>
<2201>    <NAO>     < >  <128000000>    <10.448487s>
<28877013>     <SIM>     <(28877013)
gtdusizrohwpylucbvnxmccuzvnbyzsgswcmebgujckhoelrbzondqhdlxglwy
truvfazlxiohrenfxogndjdkmr>   <5767>    <0.000346s>
<1792702375>   <NAO>     < >  <128000000>    <12.330650s>
<666>     <NAO>     < >  <128000000>    <11.971433s>
<38209744>     <SIM>     <(38209744)
jwwwpuyrvleegabyqugoaoeignucwxkhvifkcfcxqieyihyabfrdvvmclggjfs
saaxnfcrcuziukpvmsadwxzkbmskvxcpzf>     <17278>   <0.001056s>
```

<478070300> <SIM> <(478070300)
ymduohftiocdupgascuagvouphclnrxngbkukppsgsycj> <9959>
<0.000589s>

Binária

<753763427> <SIM> <(753763427)
nascljsqjdhvtnhhxizecnlnctyhxcgmdyrqjlgsqnnjdwqaerhgfswjoutnyz
c> <1445> <0.117565s>
<1298938> <NAO> < > <864> <0.000110s>
<1692741001> <SIM> <(1692741001)
ofaugvlbsdfrombcbkvubfakkfooxwmlbphjksn> <2912>
<0.023953s>
<2201> <NAO> < > <544> <0.000113s>
<28877013> <SIM> <(28877013)
gtdusizrohwpylucbvnxmccuzvnbyzsgswcmebgujckhoelrbzondqhdlxglwy
truvfazlxiohrenfxogndjdkmr> <1072> <0.000140s>
<1792702375> <SIM> <(1792702375)
grnmlqxpmsldiqdjvtfgahcqejvawbjcswpfmouagheozjawefceoewsottkuc
ppaewpurpcattcetyiybnpf> <2561> <0.000335s>
<666> <NAO> < > <544> <0.000068s>
<38209744> <SIM> <(38209744)
jwwwpuyrvleegabyqugoaoeignucwxkhvifkcfcxqieyihyabfrdvvmclggjfs
saaxnfcrcuziukpvmsadwxzkbmskvxcpzf> <917> <0.000123s>
<478070300> <SIM> <(478070300)
ymduohftiocdupgascuagvouphclnrxngbkukppsgsycj> <2038>
<0.000265s>

Os valores mínimo, máximo e médio do tempo de execução e tamanho médio dos blocos para as buscas foram:

BUSCA	Tempo Mínimo	Tempo Médio	Tempo Máximo	Média de Blocos Lidos
Linear	0,000128s	4,8701786s	12,330650s	56896651,77
Binária	0,000068	0,0158524	0,117565	1433
Indexada	não realizado	não realizado	não realizado	não realizado

*Não ordenado

Para 8000000 registros, buscou 9 ids (753763427, 1298938, 1692741001, 2201, 28877013, 1792702375, 666, 38209744 e 478070300).

Linear

<753763427> <SIM> <(753763427) nascljsqjdhvtnhhxizecnlnctyhxcgmdyrqjlgsqnnjdwqaerhgfswjoutnyzc> <1791> <0.000127s>

<1298938> <NAO> < > <256000000> <19.922090s>

<1692741001> <SIM> <(1692741001) ofaugvlbsdfrombcbkvubfakkfooxwmlbphjksn> <35071> <0.002153s>

<2201> <NAO> < > <256000000> <23.331283s>

<28877013> <SIM> <(28877013) gtdusizrohwpylucbvnxmccuzvnbyzsgswcmebgujckhoelrbzondqhdlxglwytruvfazlxiohrenfxogndjdkmr> <5767> <0.000539s>

<1792702375> <NAO> < > <256000000> <23.756701s>

<666> <NAO> < > <256000000> <25.712954s>

<38209744> <SIM> <(38209744) jwwwpuyrvleegabyqugoaoeignucwxkhvifkcfcxqieyihyabfrdvvmclggjfs

```
saaxnfcrcuziukpvmsadwxzkbmskvxcpzf>     <17278>   <0.001602s>
<478070300>    <SIM>     <(478070300)
ymduohftiocdupgascuagvouphclnrxngbkukppsgsycj>    <9959>
<0.000906s>
```

Binária

```
<753763427>    <NAO>     < >  <1696>    <0.000235s>
<1298938> <NAO>     < >  <736>     <0.000090s>
<1692741001>   <NAO>     < >  <4512>    <0.000544s>
<2201>    <NAO>     < >  <736>     <0.000085s>
<28877013>     <NAO>     < >  <736>     <0.000078s>
<1792702375>   <NAO>     < >  <6560>    <0.000770s>
<666>     <NAO>     < >  <736>     <0.000149s>
<38209744>     <NAO>     < >  <736>     <0.000054s>
<478070300>    <NAO>     < >  <1632>    <0.000119s>
```

Os valores mínimo, máximo e médio do tempo de execução e tamanho médio dos blocos para as buscas foram:

BUSCA	Tempo Mínimo	Tempo Médio	Tempo Máximo	Média de Blocos Lidos
Linear	0,000127s	10,3031506s	25,712954s	113785540,66
Binária	*	*	*	*
Indexada	não realizado	não realizado	não realizado	não realizado

*Não ordenado

Conclusão

Dos três algoritmos de busca implementados como esperado o mais simples foi o Linear, que no entanto conseguiu o menor desempenho pois no pior caso ele precisa varrer todos os registros então o tempo médio para encontrar um registro é bem alto como constatado.

Já os algoritmos de Busca Binária e de Busca Indexada são mais sofisticados e por isto precisam que certas condições sejam atendidas para o seu correto funcionamento e o tempo de busca caia drasticamente. Para a execução da busca binária o arquivo de registros da entrada precisa estar ordenado, garantindo que a busca localize rapidamente o menor bloco a se percorrer para tentar encontrar o arquivo, resolvendo o problema da busca linear que necessita percorrer a coleção de registros inteira para confirmar a ausência do arquivo pois garantidamente ele deveria estar lá. E para executar a busca indexada faz se necessário que existam índices ordenados de acesso, que foram divididos em denso e esparso. O denso contém um índice para todo registro ordenado, e o esparso contém o índice do primeiro elemento de cada bloco, permitindo que a consulta precise buscar no máximo um bloco e que em casos que o índice confira com o do esparso o tempo fica constante para este acesso

Com base nos experimentos, pudemos concluir que a busca binária em arquivos é mais rápida devido ao baixo número de blocos lidos comparados a busca linear, conseguindo assim menor tempo médio em todos os casos.