TP1: Banco de Dados com Árvores B+

Alef Henrique de Castro Monteiro

4 de maio de 2016

1 Introdução

As árvores B são estruturas de dados que, diferentemente das árvores binárias, podem ter mais de dois filhos por nó. O número de filhos de uma árvore B varia de acordo com sua ordem m, ou seja, quanto maior é a ordem da árvore, maior é o número de filhos em cada nó da árvore. Um modelo específico de árvore B é a árvore B+. As árvores B+ se diferenciam das árvores B comuns por armazenarem seus registros apenas nos nós folha. Devido a essa característica, as árvores B+ são muito usadas em bancos de dados, uma vez que elas minimizam o número de alterações na árvore durante uma operação.

O Trabalho Prático 1 tem por objetivo a simulação de um banco de dados simplificado, que deve ser implementado com a utilização de uma árvore B+ para indexar os registros. Esse banco de dados deve armazenar todos os registros em memória secundária e deve suportar operações simples de busca e de inserção. Além da implementação do banco de dados, o trabalho discute as vantagens trazidas pela utilização de uma árvore B+ por meio da análise de experimentos.

2 Solução do problema

Nesta seção do trabalho, serão apresentados os detalhes da implementação do banco de dados e das operações que são feitas em cima desse banco de dados. A solução do problema inicia-se com a verificação das informações passados para o programa pelo terminal, antes da execução. Essas informações contém a ordem(M) da árvore B+, o número de campos fornecidos pelos registros que serão armazenados na arvore e, ainda, qual desses campos deve ser utilizado como chave para indexar os registros na árvore. Para a implementação deste, foi estipulado que o número máximo de chaves permitidas em cada nó da árvore é (M-1) e o número mínimo de chaves permitidas é (teto(M/2)- 1). Ou seja, cada nó interno terá entre M/2 e M filhos.

Ao ser executado, o programa verifica cada linha do arquivo de entrada e realiza a operação solicitada. Após a verificação da operação, o fluxo do programa pode tomar três caminhos diferentes, dependendo da operação solicitada: inserção, busca ou impressão. É importante ressaltar que, entre cada uma das operações, somente o conteúdo do nó raiz da árvore é mantido na memória primária, os demais nós da árvore ficam armazenados em memória secundária (disco). Para armazenar o conteúdo dos nós em disco, cada nó é serializado em bytes em um arquivo. Quando o programa necessita de alguma informação do nó, ele busca o nó no disco e realiza o processo inverso, ou seja, desserializa os bytes do arquivo. Cada uma das operações realizadas pelo programa (inserção, busca e impressão) serão detalhadas a seguir.

2.1 Inserção

Para realizar a inserção de um novo registro na árvore, inicialmente o programa verifica qual é a chave do novo registro e armazena o registro no disco. A partir de então, o objetivo é inserir a chave nova na árvore. O programa utiliza uma função que percorre a árvore, nível por nível, até encontrar o nó folha ideal para o armazenamento da chave e do registro. Ao encontrar esse nó folha, o programa armazena a chave e um ponteiro para o novo registro. Todo esse funcionamento pode ser ilustrado com pseudocódigo a seguir[1].

Código 1:]InsereRegistro(Novo Registro, Arvore T, ordemArvore m)- Adaptado de [Cormen, 2009]

```
1
2  r= T.raiz
3  Serializa(NovoRegistro)          //Armazena novo registro no disco.
4  if r.numeroChaves == m-1         //Se a raiz estiver cheia, aumenta a arvore e cria nova raiz.
5                  s= Aloca Memoria()
6                  T.raiz= s
7                  s.folha= false
8                  s.numeroChaves= 0
9                  s.numeroRegistros= 0
10                 s.ponteirofilhos[1]= r
11                 ReparteNodo(s, 1)
12                 InsereArvoreNaoCheia(s, NovoRegistro)
13 else                             //Se a raiz nao estiver cheia, insere o registro.
14         InsereArvoreNaoCheia(r, NovoRegistro)
```

Código 2:]InsereArvoreNaoCheia(Nodo n, Novo Registro)- Adaptado de [Cormen, 2009]

```
1
2  i= n.numeroChaves
3  if n.folha==true                         //Se chegou em um no folha, insere o registro.
4                  while i>=1 e NovoRegistro.chave < n.chave[i]
5                          n.chave[i+1] = n.chave[i]
6                          n.ponteiroRegistros[i+1] = n.ponteiroRegistros[i]
7                          i--
8                  n.chave[i+1] = NovoRegistro.chave
9                  n.ponteiroRegistros[i+1] = NovoRegistro.posicaoNoDisco
10                 if n != T.raiz
11                         Serializa(n)    //Altera os valores do nodo no disco.
12 else                                    //Se ainda nao e um no folha, continua caminhando.
13                 while i>=1 e NovoRegistro.chave < n.chave[i]
14                         i--
15                 i++
16                 Deserializa(n.filhos[i])        //Traz o filho do nodo para memoria principal.
17                 if n.filhos[i].numeroChaves == m-1      //Se o filho estiver cheio, reparte.
18                                 ReparteNodo(n, i)
19                                 if k > n.chave[i]
20                                         i++
21                                 if n != T.raiz
22                                         Serializa(n)
23                                 Deserializa(n.filhos[i])
24                 InsereArvoreNaoCheia(n.filhos[i], NovoRegistro) //Chama a funcao recursivamente para
                        o nodo filho
```

Código 3:]ReparteNodo(Nodo n, Indice i) - Adaptado de [Cormen, 2009]

```
1  parte2 = Aloca memoria()         //Cria o nodo que recebe a parte da direita da reparticao.
2  Deserializa(n.filhos[i])         //Traz o filho cheio que sera repartido para a memoria.
3  parte1 = n.filhos[i]             //Parte1 e o filho cheio.
4  parte2.folha = parte1.folha
5  if parte2.folha == true          //Se for nodo folha, parte2 deve conter uma copia da chave que foi
       para o nodo pai.
6          parte2.numeroChaves = minimoPermitido + 1
7  else
8          parte2.numeroChaves = minimoPermitido
9  if parte2.folha == false
10         parte2 recebe a primeira metade dos filhos de parte1
11 parte1.numeroChaves = minimoPermitido   //O filho que estava cheio agora tem a metade das chaves.
12 todos filhos de n a partir da posicao i pulam uma casa para a direita
13 todas chaves de n a partir da posicao i pulam uma casa para a direita
14 n.filhos[i+1] = parte2                  //O nodo pai recebe seu novo filho.
15 n.chave[i+1] = parte1.chave[m/2]        //O nodo pai recebe a chave do meio do nodo filho cheio.
16 n.numeroChaves++
17 n.numeroRegistros = 0
18 if n!=T.raiz
19         Serializa(n)                    //Armazena no disco o nodo modificado.
20 Serializa(parte1)                       //Armazena no disco o nodo modificado.
21 Serializa(parte2)                       //Armazena no disco o novo nodo.
```

2.2 Busca

Ao realizar uma busca, primeiramente o programa verifica qual campo do registro é utilizado como chave. A partir daí, o programa percorre a árvore, nível por nível, até um nó folha. Caso o nó folha contenha a chave, o

programa verifica a posição onde o registro foi armazenado no disco, acessa o disco, obtém o registro e imprime no arquivo de saída o registro buscado. Caso a busca seja sem sucesso, o programa imprime "null"no arquivo de saída. O funcionamento detalhado da operação de busca é demonstrado a partir do pseudocódigo a seguir[1].

Código 4:]Busca(Nodo n, Registro reg) - Adaptado de [Cormen, 2009]

```
1  i=1
2  while i<=n.numeroChaves e reg.chave > n.chave[i]   //Compara com todas as chaves do nodo, para ver
       em qual filho a busca segue.
3          i++
4  if n.folha==true  //Se chegou em um no folha, ou o registro esta no nodo ou ele nao esta na arvore.
5          if <= n.numeroChaves e reg.chave == n.chave[i]  //Se achou o registro.
6                  DesserializaRegistro(Reg)
7                  imprimeRegistro(Reg)    //imprime o registro encontrado.
8                  return 1
9          else                            //Se nao achou, termina a busca.
10                 return 0
11 else                                    //Se o nodo e interno, segue a busca no nodo filho.
12         Desserialize(n->filhos[i])
13         filho = n->filhos[i]
14         Busca(filho, reg)
```

2.3 Impressão

A operação imprime todas as chaves contidas na árvore, a partir raiz até a folhas, da esquerda para a direita. O programa imprime as chaves separadas por vírgulas, em várias linhas, onde cada linha representa um nó da árvore. Para realizar essa operação, o programa utiliza o algoritmo de busca em largura, que é demonstrado no pseudocódigo a seguir[1].

Código 5:]ImprimeArvore(Arvore T) - Adaptado de [Cormen, 2009]

```
1  Criar Fila
2  Enfileira(T.raiz)
3  while Fila nao estiver vazia
4          n= Desenfileira(Fila)
5          for cada filho i de n
6                  Desserializa(n.filhos[i])
7                  Enfileira(n.filhos[i])
8          for cada chave j de n
9                  imprime(n.chave[j])
10         if n!=T.raiz
11                 libera memoria(n)
```

3 Análise de Complexidade

O trabalho é composto do programa principal "tp1.c"é mais 4 tipos abstratos de dados (TAD's): "Operação", "Árvore B+", "Fila"e "Serialização". Nesta seção, cada um dos TAD's será analisado separadamente e, por fim, será analisado o programa principal. Para as análises a seguir, será utilizada a seguinte notação: M é a ordem da árvore, N é o número de campos do registro, H é a altura da árvore e Q é a quantidade de registros presentes na árvore.

3.1 TAD Fila

- int filaVazia(fila *queue) - A função tem apenas operações de complexidade $O(1)$. Portanto, a função tem complexidade $O(1)$.
- void novaFila(fila *queue) - A função tem apenas operações de complexidade $O(1)$. Portanto, a função tem complexidade $O(1)$.
- void enfileira(fila *queue, No* new-item) - A função tem apenas operações de complexidade $O(1)$. Portanto, a função tem complexidade $O(1)$.
- No* desenfileira(fila *queue) - A função tem apenas operações de complexidade $O(1)$. Portanto, a função tem complexidade $O(1)$.

3.2 TAD Serialização

- void Serialize-string(char *string, long int tam-string, FILE *disco, long int offset) - A função serializa uma string no disco, em forma de bytes. Essa função é composta de um loop do tipo "for"e outras operações $O(1)$. Dentro do loop há apenas uma operação $O(1)$, e o número de iterações do loop varia de acordo com o tamanho da string. No entanto, todas as vezes que a função é utilizada no programa, o tamanho da string é 30 e , portanto, o loop "for"pode ser considerado $O(1)$. Portanto, a complexidade da função é $O(1)$.

- void Deserialize-string(char *string, long int tam-string, FILE *disco, long int offset) - A função desserializa uma sequência de bytes do disco em uma string. De maneira análoga à função anterior, esta função contém apenas um loop, que sempre é executado 30 vezes. Dentro do loop, existe apenas uma operação $O(1)$. Logo, o loop é $O(1)$ e, portanto, a função também é $O(1)$.

- void Serialize-vetor(unsigned long long int *vetor, unsigned long long int tam-vetor, FILE *disco, long int offset) - A função serializa um vetor de inteiros no disco. No escopo da função existe apenas um loop "for", que é executado de acordo com o tamanho do vetor. Durante o programa, essa função é utilizada para serializar os vetores que representam os filhos, as chaves e os registros dos nodos. Como o tamanho máximo desses vetores é a ordem da árvore, pode-se dizer que o loop é $O(M)$. Portanto, a função é $O(M)$.

- void Deserialize-vetor(unsigned long long int *vetor, unsigned long long int tam-vetor, FILE *disco, long int offset) - A função desserializa uma sequência de bytes do disco em um vetor de inteiros. De maneira análoga à função anterior, esta função contém apenas um loop, que é executado no máximo "ordem"vezes. Logo, o loop é $O(M)$ e, portanto, a função também é $O(M)$.

- void Serialize (No *nodo , FILE *disco, long int offset, unsigned long long int ordem) - A função realiza a serialização, em bytes, de todo o conteúdo de um nó. Além de operações $O(1)$, para cumprir a tarefa, a função faz 3 chamadas à função "Serialize-vetor- $O(M)$. Portanto, a complexidade final da função é $O(M)$.

- void Deserialize (No *nodo, FILE *disco, long int offset, unsigned long long int ordem) - A função tem funcionamento semelhante ao da função anterior. Entretanto, essa função tem a tarefa de transformar os bytes em conteúdo do nó. Essa função tem diversas operações $O(1)$ e faz 3 chamadas à função "Deserialize-vetor- $O(M)$. Portanto, a função tem complexidade $O(M)$.

- void Serialize-registro(Registro k, FILE *disco, long int offset) - A função realiza a serialização do conteúdo de um registro. Além das operações de complexidade constante $O(1)$, a função tem um loop "for". Dentro do "for", há apenas uma operação, que chama a função "Serialize-string- $O(1)$. Como o conteudo do registro é armazenado na forma de uma matriz do tipo "char", o loop é executado para cada uma das linhas da matriz, ou seja, para cada campo do registro, o loop é executado uma vez. Então, o loop "for"é $O(N)$ e, portanto, a função também é $O(N)$.

- void Deserialize-registro(Registro *k, FILE *disco, long int offset) - Essa função realiza a desserialização de uma sequência de bytes para obter o conteúdo de um registro. Assim como a função anterior, essa função tem apenas um loop "for", as demais operações são $O(1)$. Dentro deste loop, a única operação é uma chamada à função "Deserialize-string- $O(1)$. Como o loop é executado para cada campo do registro, pode-se dizer que o loop é $O(N)$. Portanto, a função também é $O(N)$.

3.3 TAD Árvore B+

- int teto(int x, int y) - A função é utilizada para calcular o teto da divisão (x/y). Todas operações da função são $O(1)$ e, portanto, a função é $O(1)$.

- void alocaMemoriaNodo(No *nodo, unsigned long long int ordem) - A função realiza a alocação dinâmica de memória para todos os ponteiros da estrutura Nó, além de inicializar todos os valores do nó. A função contém diversas operações $O(1)$. Porém, para inicializar os valores de alguns vetores, a função utiliza um loop "for", que é executado M vezes. Portanto, a complexidade da função é $O(M)$.

- void desalocaMemoriaNodo(No *nodo, unsigned long long int ordem) - A função libera a memória utilizada pelo nó. Essa função realiza apenas operações $O(1)$ e, portanto, tem complexidade $O(1)$.

- void criaArvore(Arvore *T, FILE *disco) - A função tem apenas operações de complexidade O(1) e uma chamada à função "alocaMemoriaNodo- $O(M)$. Portanto, a função tem complexidade $O(M)$.

- void imprimeArvore(Arvore *T, No *n, FILE *output, FILE *disco) - A função foi demonstrada pelo pseudocódigo 5 (Seção 2.3). Nessa função existe uma chamada à função "novaFila-$O(1)$ e uma chamada à função "enfileira- $O(1)$. Além disso, existe um loop "while"(linha 3), que é executado uma vez para cada nó existente na árvore. Como a árvore cheia pode possuir até M^H registros, o while é executado até MˆH vezes, no pior caso. Dentro do while, existem operações $O(1)$, uma chamada à função "desenfileira- $O(1)$, um loop "for"(linha 8) e, no pior caso, um loop(linha 5) que é executado M vezes, uma vez para cada filho do nó. No pior caso, o loop for é executado é executado $(M-1)$ vezes. Porém, esse loop chama a função "alocaMemoriaNodo- $O(M)$. Portanto, a função tem um "while"com complexidade $O(M*(M-1)*M) = O(M^3)$. Esse while é executado uma vez para cada nó existente na árvore. Como a árvore tem até $M^1+M^2+...+M^H = M^{H*(H+1)/2}$ nós, o while é executado $M^{H*(H+1)/2}$ vezes e,portanto, a complexidade da função é $O(M^{3+H*(H+1)/2}) = O(M^3 + M^{(H^2)/2} + M^{H/2}) = O(M^3 + M^{H^2} + M^H) = O(M^{H^2})$, para $H >= 2$. Como a altura H pode ter tamanho máximo igual a $log_M(Q)$[1],a complexidade da função é $O(2Q) = O(Q)$.

- void reparteNodoOrdemImpar(No* nodo, No* parte1, No* parte2, unsigned long long int ordem) - A função tem diversas operações $O(1)$. No pior caso, que acontece quando o nó a ser repartido é um nó interno, a função apresenta também 2 loops "for". Os dois loops são executados $M/2$ vezes. Dessa forma, a complexidade da função é $O(2 * M/2) = O(M)$.

- void reparteNodoOrdemPar(No* nodo, No* parte1, No* parte2, unsigned long long int ordem) - A análise de complexidade desta função é idêntica à da função anterior. Portanto, a complexidade da função é $O(M)$.

- void reparteNodo(No *nodo, unsigned long long int indice, unsigned long long int ordem, FILE *disco) - A função pode ser visualizada no pseudocódigo 3(Seção 2.1). Além das operações $O(1)$, a função faz chamada às funções "alocaMemoriaNodo- $O(M)$, "Serialize- $O(M)$, "Deserialize- $O(M)$ e a uma das funções: "reparteNodoImpar- $O(M)$ ou "repartenodoPar- $O(M)$. Além disso, a função conta ainda com mais 2 loops, que são executados no pior caso M vezes (número máximo de chaves menos um). Portanto a função tem complexidade $O(6 * M) = O(M)$.

- void insereArvoreNaoCheia(Arvore *T, No *nodo, Registro k, unsigned long long int ordem, FILE *disco) - A função insere um registro na árvore e é representada no pseudocódigo 2(Seção 2.1). Para isso, a função percorre cada um dos níveis da árvore de forma recursiva e só para de ser executada quando atinge um nó folha. Assim, a função é executada H vezes. Dentro da função existem operações $O(1)$ e também chamadas às funções "alocaMemoriaNodo- $O(M)$, "Deserialize- $O(M)$. No pior caso, a função também chama "reparteNodo- $O(M)$, "Serialize- $O(M)$ e "Deserialize- $O(M)$. Portanto, a complexidade da função é $O(H * (5 * M))$. Como a altura H pode ter tamanho máximo igual a $log_M(Q)$[1],a complexidade da função é $O(M * log_M(Q))$.

- void insereRegistro(Arvore *T, Registro k, FILE *disco) - A função foi representada no pseudocódigo 1 (Seção 2.1) e pior caso dessa função ocorre quando a raiz da árvore está cheia. Neste caso, a função realiza operações $O(1)$ e faz chamada às funções "Serialize- $O(M)$, "alocaMemoriaNodo- $O(M)$, "reparteNodo- $O(M)$ e "insereArvoreNaoCheia- $O(M * log_M(Q))$. Portanto, a função tem complexidade $O(M * (3 + log_M(Q))) = O(M * log_M(Q))$.

- void imprimeRegistro(Registro reg, FILE *output, FILE *disco) - A função é utilizada para imprimir o conteúdo de um registro. Para cada campo do registro, a função imprime a string correspondente. Portanto, o loop da função realiza N iterações. Como a função tem apenas uma operação $O(1)$ além do loop, a função tem complexidade $O(N)$.

- int busca(No *nodo, unsigned long long int chave-buscada, FILE *output, FILE *disco, unsigned long long int ordem, unsigned long long int num-campos, Arvore *T) - A função, represntada pelo pseudocódigo 4(Seção 2.2), é uma função recursiva. A recursividade da função ocorre para cada nível da árvore atingido pela busca, até que a função chegue em um nó folha. Dessa forma, a função é executada H vezes. A função possui um loop "while"(linha 2), que é executado até M vezes, caso o nó esteja cheio e o registro esteja no último filho do nó. Além disso, a função possui um "if"(linha 4) que só é executado em uma das H vezes em que a função é executada. Dentro desse "if", existem 2 loops aninhados, com complexidade $O(N)$, um outro loop também $O(N)$ e chamadas às funções "DeserializeRegistro- $O(N)$ e "imprimeRegistro- $O(N)$. Além do "while- $O(M)$ e do "if- $O(N)$, a função ainda chama as funcões "alocaMemoriaNodo- $O(M)$ e "Deserialize- $O(M)$. Portanto, a complexidade final da função é $O(H * 3 * M + N)$. Como a altura H pode ter tamanho máximo igual a $log_M(Q)$[1], a complexidade da busca é $O(log_M(Q) * 3 * M + N) = O(M * log_M(Q) + N)$.

3.4 TAD Operação

- void operacaoInsere(char *aux, Arvore *T, unsigned long long int num-campos, unsigned long long int id-campo, FILE *disco, unsigned long long int posicao) - A função tem 3 "for"que são executados N vezes. No entanto, um deles tem dois loops dentro. Um desses loops dentro do "for"é executado apenas uma vez, pois serve apenas para ignorar a tabulação existente no arquivo de entrada. O outro loop é executado sempre 30 vezes, que é o tamanho de cada campo do registro. Portanto, os 3 "for"tem complexidade $O(N)$ cada. Além disso, a função faz chamada às funções "SerializeRegistro-$O(M)$ e "insereRegistro-$O(M * log_M(Q))$. Portanto, a complexidade final é $O(3 * N + M + M * log_M(Q)) = O(M * log\_M(Q) + N)$.

- void operacaoBusca(char *aux, Arvore *T, FILE *output, unsigned long long int num-campos, FILE *disco, unsigned long long int ordem, unsigned long long int posicao) - A função tem 3 loops que executam um número constante de vezes e faz chamada à função "busca-$O(M * log_M(Q) + N)$. Portanto, a complexidade final é $O(M * log_M(Q) + N)$.

- void executaOperacao(char *aux, char *operacao, Arvore *T, FILE *output, unsigned long long int num-campos, unsigned long long int id-campo, FILE *disco, unsigned long long int ordem) - Além das operações $O(1)$, a função tem um loop que será executado no máximo 6 vezes(tamanho da palavra "search"). Além disso, a função pode chamar uma das 3 funções: "operacaoInsere- $O(M * log\_M(Q) + N)$, "operacaoBusca- $O(M * log_M(Q) + N)$ ou "imprimeArvore- $O(Q)$. Caso a operação solicitada pelo arquivo seja "add", a complexidade da funçaão é $O(M * log_M(Q) + N)$. Se a operação for "search", a complexidade é $O(M * log_M(Q) + N)$. Por fim, se a operação for "dump", a complexidade da função será $O(Q)$.

3.5 Main

Seja I o número de operações de inserção presentes no arquivo de entrada, B o número de operações de busca e X o número de operações de impressão. O programa principal é composto de operações O(1) e de um loop "while", que é executado $(I + B + X)$ vezes. Esse loop tem apenas operações O(1) e uma chamada à função "executaOperacao". Portanto, a complexidade final do programa é $O(I * (M * log_M(Q) + N) + B * (M * log_M(Q) + N) + X * Q) = O((I + B) * (M * log_M(Q) + N) + X * Q)$.

Na operação de inserção, a complexidade é $O(M * log_M(Q) + N)$. Essa complexidade pode ser ilustrada com o pior caso, onde a chave inserida deve ficar na última chave da última folha de uma árvore. Neste caso, a função teria que percorrer toda a altura H da árvore e, em cada nível da árvore, a função teria que comparar a chave buscada até M vezes, com todas as chaves do nó. O número N de campos do registro aparece na análise porque a função tem que inserir todos os campos do registro no disco.

No caso da busca, a complexidade é $O(M * log_M(Q) + N)$ e a análise é semelhante à da inserção. No entanto, em vez de inserir o registro na última chave do último nó folha, a função iria encontrar a chave buscada. Neste caso, o número N de campos do registro aparece na análise porque a função deve imprimir todo o registro, caso o encontre.

Na operação de impressão dos dados, a complexidade é $O(Q)$. Isso mostra que o custo da impressão das chaves de uma árvore é proporcional ao número de registro da árvore. Essa conclusão é justificada pelo fato de a impressão depender do número de nós da árvore inseridos na fila para imprimir.

3.6 Análise de Complexidade do Espaço

Quando o programa executa uma operação de inserção, é necessário alocar memória para o registro. Essa alocação depende do número N de campos do registro e, então, é $O(N)$. Além disso, o programa aloca memória para todos os nós que são lidos até encontrar a poisção onde o registro deve ser inserido. Ao alocar memória para um nó, o tamanho da memória alocada para o nó depende somente da ordem M da árvore e tem complexidade $O(M)$. Portanto, a alocação dos nós depende da altura H da árvore e da ordem M. Logo, essa alocação é O(H*M), já que a cada nível da árvore é alocado um nó. Por fim, uma inserção tem complexiade espacial $O(H * M + N)$. Caso a operação seja de busca, o processo é igual ao da inserção, ou seja, memória nó é alocada em cada nível da árvore. Além disso, o programa precisa alocar memória para que o registro que foi encontrado seja impresso. Dessa forma, o programa aloca memória com complexidade $O(H * M)$ para os nós e aloca memória com complexidade $O(N)$ para os registros. Assim, a busca tem complexidade espacial $O(H * M + N)$. Na impressão, o programa aloca memória para todos os nós da árvore. Como a árvore tem $M^{H*(H+1)/2}$ nós, a complexidade

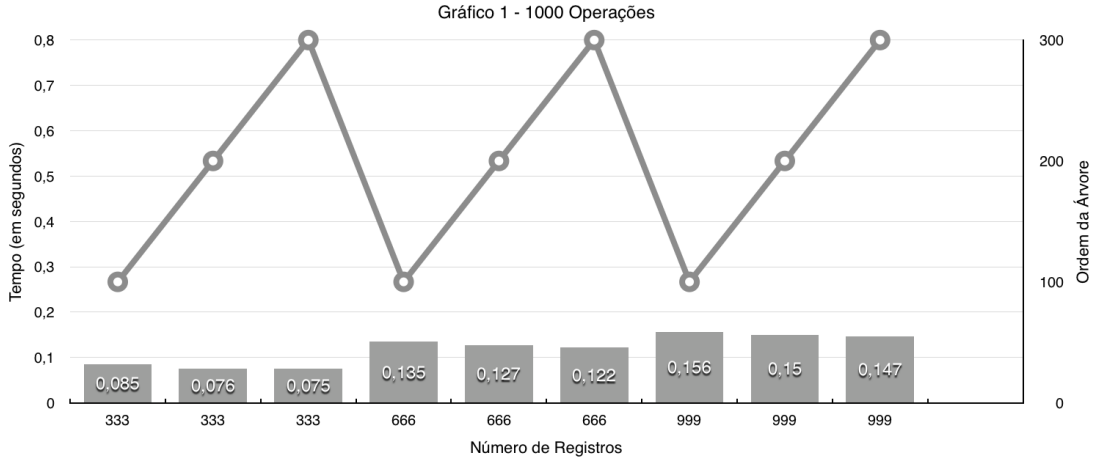
espacial da operação de impressão é $O(M * M^{H*(H+1)/2}) = O(M^{1+H*(H+1)/2}) = O(M^1 + M^{(H^2)/2} + M^{H/2}) = O(M^1 + M^{H^2} + M^H) = O(M^{H^2})$, para $H >= 2$. Como a altura H pode ter tamanho máximo igual a $log_M(Q)$[1],a complexidade espacial da operação é $O(2Q) = O(Q)$. Portanto, a complexidade espacial do programa é $O(I*(M*log_M(Q)+N)+B*(M*log_M(Q)+N)+X*Q) = O((I+B)*(M*log_M(Q)+N)+X*Q)$.

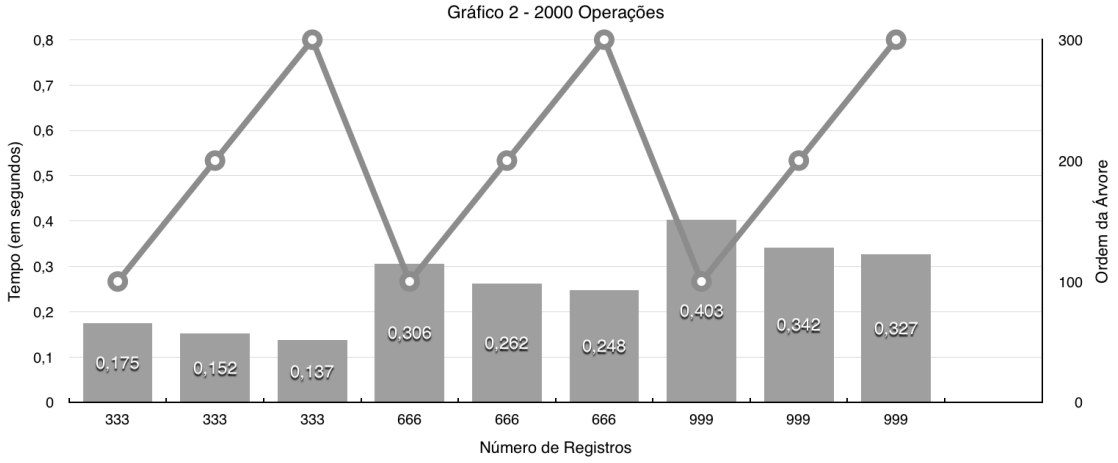
4 Análise de Experimentos

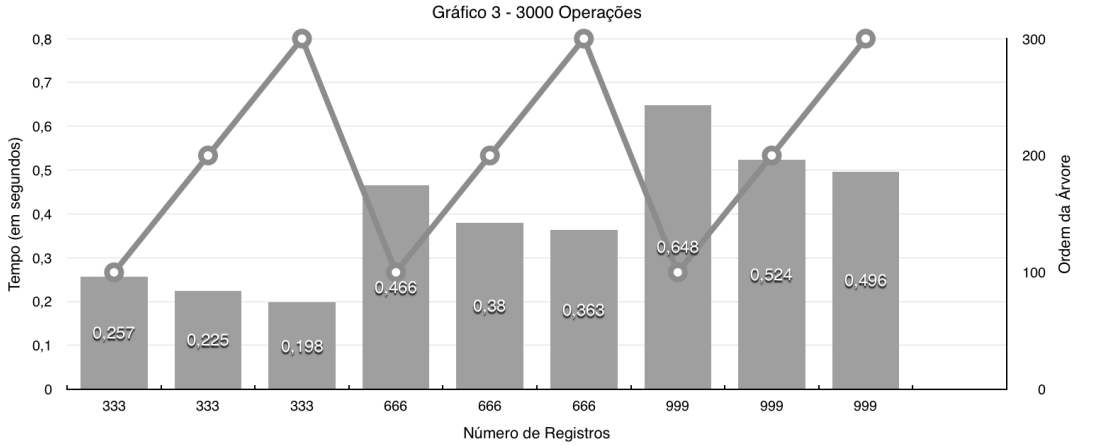
Para realizar um estudo experimental com base no número de registros de um banco de dados e com base na ordem da árvore B+ utilizada como estrutura de dados, foram realizados 27 testes. Cada um dos testes utilizou um arquivo de texto diferente como entrada de dados. Os testes foram divididos em 3 grupos de 9 arquivos: um com arquivos de 1000 linhas, um com arquivos de 2000 linhas e outro com arquivos de 3000 linhas, onde cada linha contém uma operação de inserção, busca ou impressão. Dentro de cada grupo, os testes contém valores diferentes para a ordem da árvore e para o número de registros inseridos na árvore. Os testes foram realizados em um MacBook Pro, processador 2,7 GHz Intel Core i5, memória 8 GB 1867 MHz DDR3, SSD 128GB. Para os testes apresentados nesta seção, a compilação foi feita utilizando o GCC com as flags (-O3 -g -Wall) e a execução foi única, ou seja, foram tomados como base o tempo de uma só execução. Os detalhes do teste e os resultados estão apresentados na tabela e nos gráficos a seguir.

Tabela 1

	Teste	Número de operações (P)	Número de operações "ADD" (Número de Registros)	Ordem da Árvore (M)	Tempo de execução (em segundos)
ARQUIVO 1.1	Teste 1.1.1	1000	333	100	0,085
	Teste 1.1.2	1000	333	200	0,076
	Teste 1.1.3	1000	333	300	0,075
ARQUIVO 1.2	Teste 1.2.1	1000	666	100	0,135
	Teste 1.2.2	1000	666	200	0,127
	Teste 1.2.3	1000	666	300	0,122
ARQUIVO 1.3	Teste 1.3.1	1000	999	100	0,156
	Teste 1.3.2	1000	999	200	0,150
	Teste 1.3.3	1000	999	300	0,147
ARQUIVO 2.1	Teste 2.1.1	2000	333	100	0,175
	Teste 2.1.2	2000	333	200	0,152
	Teste 2.1.3	2000	333	300	0,137
ARQUIVO 2.2	Teste 2.2.1	2000	666	100	0,306
	Teste 2.2.2	2000	666	200	0,262
	Teste 2.2.3	2000	666	300	0,248
ARQUIVO 2.3	Teste 2.3.1	2000	999	100	0,403
	Teste 2.3.2	2000	999	200	0,342
	Teste 2.3.3	2000	999	300	0,327
ARQUIVO 3.1	Teste 3.1.1	3000	333	100	0,257
	Teste 3.1.2	3000	333	200	0,225
	Teste 3.1.3	3000	333	300	0,198
ARQUIVO 3.2	Teste 3.2.1	3000	666	100	0,466
	Teste 3.2.2	3000	666	200	0,380
	Teste 3.2.3	3000	666	300	0,363
ARQUIVO 3.3	Teste 3.3.1	3000	999	100	0,648
	Teste 3.3.2	3000	999	200	0,524
	Teste 3.3.3	3000	999	300	0,496







Depois de obtido os tempos de execução dos experimentos, pode-se realizar um estudo comparativo entre a análise da complexidade do programa e o resultado dos testes. No experimento, quando o dobramos o número de linhas do arquivo de entrada e mantivemos os outros valores (gráficos 1 e 2), praticamente todos os tempos de execução tiveram um aumento de pouco mais de 100 por cento, ou seja, os tempos mais que dobraram. Quando o número de linhas foi triplicado (gráficos 1 e 3), o tempo de execução dos arquivos também aumentou. Porém, o tempo final agora foi pouco mais de 3 vezes maior que o tempo inicial. Dessa forma, os experimentos apontam um crescimento linear do tempo de execução em relação ao número de linha do arquivo de entrada, ou seja, há uma proporção direta entre o número P de operações realizadas e o tempo de execução do programa. Esse resultado foi similar ao da análise de complexidade feita na seção 3, onde a complexidade do programa foi $O(P * (M * log_M(Q) + N))$, ou seja, o tempo também era proporcional a P.

Ao modificar apenas o número de registros inseridos, também foram obtidos resultados interessantes. Quando dobramos o número de registros inseridos(colunas 1 a 3 e colunas 4 a 6 dos gráficos), verifica-se que o tempo quase dobra em todos os testes. Quando o número de registros inseridos foi triplicado(colunas 1 a 3 e

colunas 7 a 9 dos gráficos), verifica-se que o tempo fica bem próximo do triplo do tempo inicial. Dessa maneira, os testes mostram que o tempo de execução cresce quando há aumento no número Q de registros inseridos na árvore. No entanto, os testes mostram que esse crescimento não chega a ser linear. Na seção 3, a análise de complexidade mostrou que quanto mais operações de inserção e de busca, mais a complexidade da função se aproxima de $O(P * (M * log_M(Q) + N))$, onde P é o número total de operações realizadas, Q é o número de registros da árvore e N é o número de campos do registro. Portanto, essa análise mostra que o tempo de execução do programa deve aumentar com o aumento do número Q de registros na árvore, mas o aumento é logaritmico. Esse resultado foi comprovado pelos experimentos.

Com relação à ordem da Árvore, o comportamento do tempo é diferente. Quando dobramos a ordem da árvore de 100 para 200, os tempos de execução reduzem. Essa redução é muito pequena nos testes realizados. Ao triplicar a ordem da árvore de 100 para 300, o resultado é semelhante ao anterior. Neste caso, também é verificada uma redução nos tempos de execução. Ao verificar o percentual de redução dos tempos entre as colunas 1 e 2 do gráfico 3, por exemplo, é possível perceber que a redução é de 13%, aproximadamente. Já a redução do tempo entre as colunas 1 e 3 é de 23%, aproximadamente. Portanto, apesar da redução dos tempos serem pequenas, os tempos tendem a diminuir cada vez mais quando a ordem aumenta. Dessa forma, pode-se verificar que o aumento da ordem da árvore diminui o tempo de execução do programa, mas essa redução no tempo será considerável somente quando o número de operações for muito grande ou quando o aumento da ordem da árvore for muito grande. Esse resultado é similar ao da análise de complexidade realizado na Seção 3.

Na análise, a complexidade do programa foi $O(P * (M * log_M(Q) + N))$, ou seja, quanto maior a ordem M, menor é o valor de $log_M(Q)$. Vale ressaltar que o M que multiplica $log_M(Q)$ representa a busca do registro em um nó, que no pior caso percorre todos os nós até o final para adicionar ou buscar uma chave. Na prática, na grande parte das vezes o programa não precisa percorrer todo o nó. Portanto, a grande influência da ordem M está no valor de $log_M(Q)$. Dessa forma, tanto a análise de complexidade quanto os experimentos mostram o mesmo resultado: a ordem diminui logaritmicamente o tempo de execução.

Por fim, a utilização de árvores B+ como banco de dados mostra-se bastante eficiente. Em comparação à uma lista, por exemplo, a lista teria complexidade $O(Q)$ para buscar um registro. Já a árvore B+, teve complexidade $O(log_M(Q))$ para buscar um nó e $O(M)$ para buscar o registro no nó, ou seja, complexidade final $O(M * log_M(Q))$. Essa complexidade é bem melhor que a da lista, já que a quantidade de registros Q é muito maior que a ordem M nos bancos de dados reais. Já uma árvore binária, por exemplo, tem complexidade $O(log_2(Q))$ para encontrar um nó na árvore. Portanto, a árvore B+ também se mostra mais eficiente que as árvores binárias de busca, pois a ordem M costuma ser do tamanho de uma página da memória, ou seja, M é muito maior que 2. Por ser bem maior que 2, a ordem M torna $log_M(Q)$ muito menor que $log_2(Q)$, ou seja, torna a busca muito mais rápida na prática.

5 Conclusão

O Trabalho Prático 1 abordou a implementação de uma árvore B+ como banco de dados. O grande desafio do trabalho foi a implementação do Tipo Abstrato de Dados(TAD) da árvore B+. Esse TAD envolve operações complicadas, principalmente nas funções que inserem um registro na árvore. Outro ponto importante do trabalho foi estabelecer relações entre a análise de complexidade e os experimentos realizados. Nestas relações, os testes executados comprovaram a complexidade do programa pré-estabelecida pela análise de complexidade.

Os experimentos realizados utilizaram valores menores que o tamanho da página do sistema para a ordem M da árvore. Caso o valor de M fosse maior que $4Kb$(tamanho da página), não seria possível alocar memória primária para todo o conteúdo do nó. Neste caso, a solução seria dividir o conteúdo do nó em nós menores, o que aumentaria os acessos ao disco e tornaria a busca muito mais lenta. Portanto, o funcionamento do programa e a eficiência da árvore estão limitam o valor da ordem M da árvore a ser no máximo igual ao tamanho da página do sistema de memória virtual.

Finalmente, o uso de uma árvore B+ como banco de dados é bastante eficiente. Entretanto, o programa implementado poderia ter um melhor funcionamento para operações de inserção e de busca. Nessas operações, o programa compara, em todos os níveis da árvore, a chave com todas as chaves do nó, até encontrar a chave ou o filho correspondente. Quando o número de chaves no nó é grande, essa operação pode se tornar muito custosa. Dessa forma, uma melhoria no algoritmo seria a utilização de Hash para indexar as chaves dentro de um nó. Assim, o processo de comparação de chaves dentro de cada nó passaria de linear para logaritmico. Assim, haveria grande melhora na eficiência do programa para uma grande quantidade de registros.

Referências

[1] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.