

**Trabalho prático 02**

**Curso Ciência da Computação**

**Ano 2017 – UFV – Campus Florestal**

|  |  |
| --- | --- |
| **Professora:** | Gláucia Braga e Silva |
| **Disciplina:** | Algoritmos e Estrutura de Dados II |
| **Aluno:** | Matheus da Silva Alves - EF02649 |
| **Aluno:** | Ruan Formigoni - EF02661 |

**Florestal, 27 de Junho de 2017**

**ÍNDICE**

**1. Introdução**

**2. Objetivo**

**3. Desenvolvimento**

**4. Conclusões**

**1. Introdução**

A dupla teve como guia uma descrição do trabalho prático 02 (TP2), o qual foi disponibilizada pela professora da disciplina. Nesta descrição continha o requerimento da criação de um programa que cria uma árvore B que armazena diversos tipos de dados como estrutura, esses dados deveriam ser inseridos, removidos e pesquisados dentro da árvore respeitando as regras da estrutura de árvore B. Os algoritmos básicos para existencia da arvore, após serem criados deveriam passar por uma fase de teste que consistia em receber como entrada em um arquivo “Entrada.txt”, assim, relatar os resultados obtidos.

**2. Objetivo**

O objetivo do TP2 consiste em analisar o desempenho da arvore B em diferentes cenários. Esta analise constitui-se na comparação das diferentes operações levando em consideração três métricas de ordem divergentes para calcular o desempenho de tal. Essas três métricas são respectivamente: 2, 20 e 40, para as operações de cálculo de altura, o número de acessos a disco, o número de comparações nas operações de inserção, pesquisa e retirada em cada ordem diferente. O cenários é criado a partir da criação da árvore B inserção, retirada, pesquisa, numero de acessos a disco e altura para ordem 2 com 10.000 registros, após essa primeira manda de dados formados, há uma mudança de ordem na árvore B, dessa forma repete-se as operações e é gerado novos dados para serem comparados com os primeiros dados gerados, com fito de gerar uma conclusão de qual ordem é melhor para o melhor desempenho da árvore B.

**3. Desenvolvimento**

Primeiramente foi necessário a criação das funções de ordenação de vetores. Foi criado um arquivo.h e um arquivo.c para cada quicksort pedido. Os arquivos .c e .h foram denominados com um padrão, “quickSort” concatenando com a ordem do pedido na descrição do TP3, ou seja o primeiro quicksort na descrição do TP3, é o quicksort recursivo visto em sala de aula, dessa forma o arquivo .c e .h referente a esse quicksort e o “quickSort1”, pois e padrão o nome “quickSort” concatenado com a ordem dos quicksort's pedidos na descrição do TP3.

-Variações do quickSort

***void FQS1 (int\* A, int n, int\* M, int\* C){***

***Ordena1 (0, n-1, A, M, C);***

***}***

a função “FQS1” é a função que é chamada pelo main. Por parâmetros e passa um vetor de inteiros A, um inteiro n que é o tamanho do vetor, um inteiro que e modificado dentro da função M para saber o numero de movimentações que ocorre, um inteiro que pode ser modificado dentro da função C para saber o numero de comparações que ocorre. Esse modelo é um padrão adotado até para os outros quickSort's pois e os parâmetros que são necessários em todos eles, alguns possuem mais parâmetros pois eles possuem algo característico que dá um diferencial.

***void FQS3 (int\* A, int n, int m, int\* M, int\* C){***

***Ordena3 (0, n-1, A, m-1, M, C);***

***}***

A função “FQS3” é a função que é chamada pelo main. Os parâmetros da função são padrões com um diferencial da variável m, um inteiro que define quando a função “Inserção” será chamada, isso ocorre quando a partição no quicksort tem um tamanho menor ou igual a variável m. Essa variavel pode variar em 10 ou 100.

***void FQS2 (int\* A, int n, int k, int\* M, int\* C){***

***Ordena2 (0, n-1, A, k, M, C);***

***}***

A função FQS2 é a função que é chamada pelo main. Os parâmetros da função são padrões com um diferencial da variável k, um inteiro que controla quantos números aleatórios serão escolhido pra fazer a mediana. K pode variar entre 3 e 5.

***void FQS4 (int\* A, int n, int\* M, int\* C){***

***Ordena4 (0, n-1, A, M, C);***

***}***

A função FQS4 é a função que é chamada pelo main. Os parâmetros da função são padrões. O diferencial da função é que ela sempre ordena a menor partição. Na descrição do TP3 esse quicksort e denominado como “Empilha inteligente”.

***void FQS5 (int\* v, int n, int\* M, int\* C){***

***Ordena5 (0, n-1, v, M, C);***

***}***

A função FQS5 é a função que é chamada pelo main. Os parâmetros da função são padrões. O diferencial da função é que ela não é recursiva. Na descrição do TP3 esse quicksort e denominado como “Interativo”.

***void FQS6 (int\* v, int n, int\* M, int\* C){***

***Ordena6 (0, n-1, v, M, C);***

***}***

A função FQS5 é a função que é chamada pelo main. Os parâmetros da função são padrões. O diferencial da função é que ela não é recursiva e sempre ordena as menores partições primeiro. Na descrição do TP3 esse quicksort e denominado como “Interativo empilha inteligente”.

-Variações das funções internas

***void Particao1(int Esq, int Dir, int \*i, int \*j, int \*A, int\* M, int\* C){***

***int pivo, aux;***

***\*i = Esq; \*j = Dir;***

***pivo = A[(\*i + \*j)/2]; /\* obtem o pivo x \*/***

***do{***

***while (pivo > A[\*i]){***

***(\*i)++;***

***(\*C)++;***

***}***

***while (pivo < A[\*j]){***

***(\*j)--;***

***(\*C)++;***

***}***

***if (\*i <= \*j){***

***aux = A[\*i]; A[\*i] = A[\*j]; A[\*j] = aux;***

***(\*M) = (\*M)+3;***

***(\*i)++; (\*j)--;***

***}***

***} while (\*i <= \*j);***

***}***

Pode-se definir que essa função denominada de “Partição1” é uma base para todas as outras. O pivô e definido pela elemento médio do vetor.

***void Particao2(int Esq, int Dir, int \*i, int \*j, int \*A, int k, int\* M, int\* C){***

***int pivo, aux, vetAux[k], cont;;***

***\*i = Esq; \*j = Dir;***

***for(cont=0;cont<k;cont++){***

***vetAux[cont] = A[(rand()%(Dir-Esq))+Esq];***

***}***

***FQS1(vetAux, k, M, C);***

***pivo = vetAux[k/2]; /\* obtem o pivo x \*/***

***do{***

***while (pivo > A[\*i]){***

***(\*i)++;***

***(\*C)++;***

***}***

***while (pivo < A[\*j]){***

***(\*j)--;***

***(\*C)++;***

***}***

***if (\*i <= \*j){***

***aux = A[\*i]; A[\*i] = A[\*j]; A[\*j] = aux;***

***(\*M) = (\*M)+3;***

***(\*i)++; (\*j)--;***

***}***

***} while (\*i <= \*j);***

***}***

Pode-se definir que essa função denominada de “Partição2” recebe como base a função “Partição1”, mas com um diferencial, como é necessário montar uma mediana com elementos do vetor, foi criado como uma variável auxiliar da função “vetAux[k]”, o qual e definido um vetor com tamanho k e adicionado k elementos aleatórios que fazem parte do vetor principal dentro desse vetor secundário, após isso a função “FQS1(vetAux, k, M, C);” é chamada para ordena esse vetor secundário afim de obter a mediana desses elementos e obter esse elemento como pivô.

***void Particao3 (int Esq, int Dir, int \*i, int \*j, int \*A, int m, int\* M, int\* C){***

***int pivo, aux;***

***\*i = Esq; \*j = Dir;***

***if(Dir - Esq <= m){***

***Insercao(A, Esq, Dir, M, C);***

***}***

***pivo = A[(\*i + \*j)/2];/\* obtem o pivo x \*/***

***do{***

***while (pivo > A[\*i]){***

***(\*i)++;***

***(\*C)++;***

***}***

***while (pivo < A[\*j]){***

***(\*j)--;***

***(\*C)++;***

***}***

***if (\*i <= \*j){***

***aux = A[\*i]; A[\*i] = A[\*j]; A[\*j] = aux;***

***(\*M) = (\*M)+3;***

***(\*i)++; (\*j)--;***

***}***

***} while (\*i <= \*j);***

***}***

Pode-se definir que essa função denominada de “Partição2” recebe como base a função “Partição1”, mas com um diferencial, como é necessário saber quando a partição é menor ou igual a m é inserido esta condição “if(Dir - Esq <= m)”, caso a condição seja verdadeira então a função “Insercao(A, Esq, Dir, M, C);” é chamada para ordenar pelo método “InsertSort”, assim como pedido na descrição do TP3.

Até o momento essas 3 variações de quicksort possuem uma função que comanda a recursividade que seria um padrão, e como as mudanças necessárias não necessitaram uma mudança em tal função, ela é identica em todas as tres variações, e pode também servir de base para outras variações dessa função.

***void Ordena”X”(int Esq, int Dir, int \*A, int m, int\* M, int\* C){***

***int i,j;***

***Particao”X”(Esq, Dir, &i, &j, A, m, M, C);***

***if (Esq < j) Ordena”X”(Esq, j, A, m, M, C);***

***if (i < Dir) Ordena”X”(i, Dir, A, m, M, C);***

***}***

Assim, a única variação ocorrida entre as funções “Partição” já citadas é o numero após a palavra “Ordena”, que foi representado acima como “X”. Como já foi mencionado acima de acordo com o a ordem pedida na descrição do TP3 e concatenado com a ordem pedida.

***void Particao4(int Esq, int Dir, int \*i, int \*j, int \*A, int\* M, int\* C, int\* p1, int\* p2){***

***int pivo, aux;***

***\*i = Esq; \*j = Dir;***

***pivo = A[(\*i + \*j)/2]; /\* obtem o pivo x \*/***

***do{***

***while (pivo > A[\*i]){***

***(\*i)++;***

***(\*C)++;***

***(\*p1)++;***

***}***

***while (pivo < A[\*j]){***

***(\*j)--;***

***(\*C)++;***

***(\*p2)++;***

***}***

***if (\*i <= \*j){***

***aux = A[\*i]; A[\*i] = A[\*j]; A[\*j] = aux;***

***(\*M) = (\*M)+3;***

***(\*i)++; (\*j)--;***

***}***

***} while (\*i <= \*j);***

***}***

Pode-se perceber que a “Partição4” é quase igual a “Partição1”, pois a função que é necessária alterar para poder “empilhar inteligente” é a função “Ordena”, pois é lá que define qual partição será menor, a única diferença na função “Partição” é a inserção de duas variáveis para contar qual qual partição é menor, dessa forma pode-se concluir que essas variáveis que foram adicionada são respectivamente: “p1”, “p2”

***void Ordena4 (int Esq, int Dir, int \*A, int\* M, int\* C){***

***int i,j,p2, p1;***

***p1 =0;***

***p2 =0;***

***Particao4(Esq, Dir, &i, &j, A, M, C, &p1, &p2);***

***if(p2>p1){***

***if (Esq < j) Ordena4(Esq, j, A, M, C);***

***if (i < Dir) Ordena4(i, Dir, A, M, C);***

***}else{***

***if (i < Dir) Ordena4(i, Dir, A, M, C);***

***if (Esq < j) Ordena4(Esq, j, A, M, C);***

***}***

***}***

Dessa forma, é fácil visualizar a alteração e porque ela foi necessária.

***void Particao5(int Esq, int Dir, int \*i, int \*j, int \*A, int\* M, int\* C){***

***int pivo, aux;***

***\*i = Esq; \*j = Dir;***

***pivo = A[((\*i) + (\*j))/2];***

***do{***

***while (pivo > A[\*i]){***

***(\*i)++;***

***(\*C)++;***

***}***

***while (pivo < A[\*j]){***

***(\*j)--;***

***(\*C)++;***

***}***

***if (\*i <= \*j){***

***aux = A[\*i]; A[\*i] = A[\*j]; A[\*j] = aux;***

***(\*M) = (\*M)+3;***

***(\*i)++; (\*j)--;***

***}***

***} while (\*i <= \*j);***

***}***

A função “Partição5” é idêntica a função “Partição1”, pois a diferença do quicksort interativo dos outros é que ele não é recursivo dessa forma o que realmente difere no algoritmo do quicksort interativo com o recursivo é na função ordena, que é controladora de quantas vezes a função é chamada e sua condição de parada.

***void Ordena5 (int Esq, int Dir, int \*Vet, int\* M, int\* C){***

***int j,i,t;***

***int \*pilhaEsq, \*pilhaDir;***

***pilhaEsq =(int\*) malloc((Dir-Esq+1)\*sizeof (int));***

***pilhaDir = (int\*) malloc((Dir-Esq+1)\*sizeof (int));***

***pilhaEsq[0] = Esq;***

***pilhaDir[0] = Dir;***

***t = 0;***

***while (t >= 0) {***

***Esq = pilhaEsq[t];***

***Dir = pilhaDir[t];***

***t--;***

***if (Esq < Dir) {***

***Particao5(Esq, Dir, &i, &j,Vet, M, C);***

***t++; pilhaEsq[t] = Esq; pilhaDir[t] = j;***

***t++; pilhaEsq[t] = i; pilhaDir[t] = Dir;***

***}***

***}***

***}***

Na função interativa a variável de controle é t que por sua vez para de receber incrementações positivas quando fez todas as partições possíveis, assim essa variável de controle começa a receber apenas incrementações negativas até sair do laço de repetição.

***void Particao6(int Esq, int Dir, int \*i, int \*j, int \*A, int\* M, int\* C){***

***int pivo, aux;***

***\*i = Esq; \*j = Dir;***

***pivo = A[((\*i) + (\*j))/2];***

***do{***

***while (pivo > A[\*i]){***

***(\*i)++;***

***(\*C)++;***

***}***

***while (pivo < A[\*j]){***

***(\*j)--;***

***(\*C)++;***

***}***

***if (\*i <= \*j){***

***aux = A[\*i]; A[\*i] = A[\*j]; A[\*j] = aux;***

***(\*M) = (\*M)+3;***

***(\*i)++; (\*j)--;***

***}***

***} while (\*i <= \*j);***

***}***

Pode-se perceber que a função “Partição6” é idêntica as funções “Partição5” e “Partição1”, pois o quicksort interativo empilha inteligente, não necessita de contadores como o quicksort recursivo empilha inteligente. Para saber se uma partição é menor que a outra eu posso utilizar o valor dentro do vetor auxiliar na posição t, fazendo pilhaDir[t] – pilhaEsq[t] comparando com os mesmos vetores na posição t-1 na função “Ordena6”. Como exemplo no código abaixo.

***void Ordena6(int Esq, int Dir, int \*Vet, int\* M, int\* C){***

***int j,i,t;***

***int \*pilhaEsq, \*pilhaDir;***

***pilhaEsq =(int\*) malloc((Dir-Esq+1)\*sizeof (int));***

***pilhaDir = (int\*) malloc((Dir-Esq+1)\*sizeof (int));***

***pilhaEsq[0] = Esq;***

***pilhaDir[0] = Dir;***

***t = 0;***

***while (t >= 0){***

***if(t==0){***

***Esq = pilhaEsq[t];***

***Dir = pilhaDir[t];***

***t--;***

***if (Esq < Dir){***

***Particao6(Esq, Dir, &i, &j, Vet, M, C);***

***t++; pilhaEsq[t] = Esq; pilhaDir[t] = j;***

***t++; pilhaEsq[t] = i; pilhaDir[t] = Dir;***

***}***

***}else{***

***if(pilhaDir[t]-pilhaEsq[t]>pilhaDir[t-1]-pilhaEsq[t-1]){***

***Esq = pilhaEsq[t];***

***Dir = pilhaDir[t];***

***t--;***

***if (Esq < Dir){***

***Particao6(Esq, Dir, &i, &j, Vet, M, C);***

***t++; pilhaEsq[t] = Esq; pilhaDir[t] = j;***

***t++; pilhaEsq[t] = i; pilhaDir[t] = Dir;***

***}***

***}else{***

***Esq = pilhaEsq[t];***

***Dir = pilhaDir[t];***

***t--;***

***if (Esq < Dir){***

***Particao6(Esq, Dir, &i, &j, Vet, M, C);***

***t++; pilhaEsq[t] = i; pilhaDir[t] = Dir;***

***t++; pilhaEsq[t] = Esq; pilhaDir[t] = j;***

***}***

***}***

***}***

***}***

***}***

**4. Conclusões**

Foi concluído pela dupla que o trabalho prática foi de bastante valia para os conhecimentos da matéria pois a parte prática ajuda a entender melhor a parte teórica e suas dificuldades. Assim, foi possível perceber as características, vantagens e desvantagens de cada variação do quicksort, como não foi requisitado no trabalho prático quantificar a alocação de memória de cada quicksort então não pode comparar numericamente a alocação de memória de cada um, mas foi possível concluir que alguns quicksort's pelo código e maneira de ordenar o vetor alguns possuem perda em tempo de execução mas possuem um certo ganho em alocação de memória, como exemplo seria o quicksort empilha inteligente que libera mais rápido a memória das menores partições, ou seja as menores partições desalocam mais rapidamente uma quantidade significativa de memória. Da mesma forma o quicksort recursivo tem um desempenho menor pois é recursivo, ou seja, devido ao tempo para gerenciamento das chamadas, assim ,tem um menor desempenho na execução, enquanto o quicksort interativo já não utiliza recursão tem uma vantagem maior em relação a execução.

No gráfico o eixo X representa o tamanho do vetor, o eixo Y representa o valor quantitativo, e na legenda os números que representam as cores é a semente.

No próximo gráfico podemos observar que foi invertido o eixo X e o eixo Y.