Relatório do Trabalho 1 **Katielen, Luiz Cubas Jr.**
Estruturas de Dados 2 RAs: 1507575, 1441965
Prof. Dr. Rodrigo Minetto Data: 08/05/16

Problema

Implementar a função responsável por inserir elementos em uma árvore b.

Descrição

A codificação da estrutura mostrou-se desafiadora quando achava-se que seria uma simples tradução dos algoritmos já descritos no livro texto da disciplina (*Algoritmos: Teoria e Prática. Autor: Cormen et al.)* o que não contávamos eram com as manipulações que teríamos que realizar nos índices dos vetores dos nós.

A função responsável por inserir elementos em uma árvore b é dividida em duas subfunções. São elas:

- `dividir_no`

Primeiramente, implementamos a função `dividir_no`, a qual divide o vetor do nó a partir da mediana. No caso a mediana "subiria" para o nó superior e um novo nó seria criado, resultando em um nodo com elementos menores que a mediana e um novo nó com elementos maiores que a mediana.

Inicialmente a função `dividir_no` recebe como parâmetro uma `Arvore *x`, um `int i` e uma `Arvore *y`. Uma `Arvore *x` é o nó pai, uma `Arvore *y` é o nó filho deste pai e o `int i` é a posição da mediana do nó cheio. Em seguida um nó `z` é criado e o vetor deste nó recebe os elementos maiores que a mediana do nodo completo.

Por fim, o tamanho do vetor do nó `y` é atualizado, uma vez que ele perdeu alguns dos seus elementos para o vetor do nó `z`, os elementos, ou chaves, do vetor do nodo `x` são organizados em ordem crescente e seus filhos são postos nas posições corretas, isto apenas se a quantidade de chaves do nó `x` for maior que 1. O nodo `z` ainda é "linkado" ao nó pai, a mediana do nó anteriormente cheio é adiciona ao nó pai e a quantidade de elementos do nó pai é atualizada em 1 unidade. Ainda existe uma instrução nesta função que aborda elementos que não são folhas da árvore b, mas estão cheios. Nesta instrução os nós folhas são "linkados" na árvore/nodo `z` para para posterior ordenação e acoplamento na árvore b.

Código:

```
/*Descricao: Divide o no caso o vetor
que abriga as chaves esteja cheio. */
Arvore* dividir_no (Arvore *x, int i, Arvore *y) {

   int j;
```

```
    /* Cria e preenche o novo no (para o filho i + 1). */
    Arvore *z = criar();
    z->folha = y->folha;
    z->n = T - 1;

    /* Move as chaves maiores que a mediana.
    De y (filho i) para z (novo filho i + 1). */
    for(j = 0; j < T - 1; j++)
            z->chaves[j] = y->chaves[j + T];

    /* Move T filhos para cima, caso necessario. */
    if(!y->folha){
            for(j = 0; j <= T - 1; j++)
                    z->filhos[j] = y->filhos[j + T];
    }

    /* Atualiza! */
    y->n = T - 1; //Novo tamanho da antiga raiz.
    for(j = x->n + 1; j > i + 1; j--){ //Arruma filhos e chaves.
            x->filhos[j] = x->filhos[j - 1];
            x->chaves[j - 1] = x->chaves[j - 2];
    }
    x->filhos[i + 1] = z; //Insere o novo filho em x.
    x->chaves[i] = y->chaves[T - 1]; //Mediana sobe.
    x->n = x->n + 1; //Novo tamanho de x.

    return x;
}
```

- `inserir_arvore_nao_cheia`

Em seguida implementamos a função `inserir_arvore_nao_cheia`, a qual insere um elemento na posição correta do vetor do nó correto e se o vetor estiver cheio, invoca a função anteriormente descrita, a `dividir_no`.

No começo, a função recebe como parâmetro uma `Arvore *x` e o elemento que quer se adicionar, `TIPO k`, para este trabalho `TIPO` pode ser um número inteiro ou um caractere. `Arvore *x` aqui é o nó pai.

Primeiramente é verificado se o nodo `x` é um nó folha, caso seja, os elemento do vetor são colocados em ordem crescente e o item que se quer inserir é posto na posição correta. A quantidade de elementos em `x` é atualizada. Entretanto, se o nodo `x` não for folha, o novo elemento deve ser posto em um dos filhos, ou nos filhos dos filhos, utilizando recursão e obedecendo a ordem crescente em que os elementos devem ser postos, "chamando" a função aqui descrita, até alcançar os nós folha. Ainda é verificado se o nó seguinte da fila de recursão não está cheio. Logo, o nosso algoritmo, sempre que encontrar um nó cheio, "quebra-o" a partir da mediana utilizando a função `dividir_no`.

Código:

```
/*Descricao: Insere um elemento k em um no nao cheio. */
Arvore* inserir_arvore_nao_cheia (Arvore *x, TIPO k) {

    int i = x->n;

    if(x->folha){
            /* coloca os elementos em ordem crescente. */
            while(i >= 1 && k < x->chaves[i - 1]){
                    i--;
                    x->chaves[i + 1] = x->chaves[i];
            }
            x->chaves[i] = k; //Adiciona k.
            x->n = x->n + 1; //Atualiza n.
    }else{ //Adiciona nos filhos.
            /* Seleciona a chave correta. */
            while(i >= 1 && k < x->chaves[i - 1]){
                    i--;
            }
            if(x->filhos[i]->n == 2*T - 1){ //Filho cheio! Divide.
                    dividir_no(x, i, x->filhos[i]);
                    if(k > x->chaves[i]) //Seleciona o filho correto.
                            i++;
            }
            inserir_arvore_nao_cheia(x->filhos[i], k);
    }
    return x;
}
```

Conclusão

A princípio parecia ser um exercício simples de ser realizado, implementar uma estrutura de inserção de elementos em uma árvore b, uma vez que boa parte do código já havia sido implementado pelo professor e as estruturas necessárias para conclusão do código, `dividir_no` e `inserir_arvore_nao_cheia` estavam explicadas e implementadas (em Pascal?) no livro *Algoritmos: Teoria e Prática* do Cormen.

Contudo, Houve um grande dispêndio de tempo nos testes de mesa, para entender o código e corrigir os índices dos vetores, que no algoritmo implementado no livro iniciam-se em 1 e em linguagem C, necessária para a realização do trabalho, iniciam-se em 0. Houveram também confusões com o valor da variável T, responsável por definir o tamanho dos vetores e a quantidade de filhos que esse vetores poderão ter. No teste 1 o valor padrão já definido pelo professor, T = 2, funciona perfeitamente, mas já no teste 2, o T precisa ter um valor de 3 para a saída ficar equivalente a do professor. Esta correção só foi

descoberta mediante a um e-mail enviado pelo professor Minetto informando a correção do valor de T para o teste 2.

Por fim, não ficou claro para a equipe o motivo de T necessitar ser 3 no teste 2, uma vez que na saída do código nenhum nó tem mais de 3 chaves, nem mesmo os nós folhas. Um T igual a 3 resultaria em nós de até 5 chaves, o que não ocorre no teste 2. Talvez seja a entrada do teste 2 que não encha seus nodos no resultado final das inserções.

Esta dúvida fica mais evidente com o teste 3, um teste feito pelos membros deste trabalho. Este teste foi baseado de uma aula teórica sobre árvores b encontrada na internet (Árvores B), logo não havia saída de código para comparação. Entretanto havia uma árvore b desenhada, com chaves equivalentes ao do teste 3, mas com resultados distintos. Apesar disso, a árvore gerada pelo algoritmo não deixa de estar correta, uma vez que os elementos estão em posições corretas, não há divergências de maior e menor elemento, e não há perda de informação.

Testes

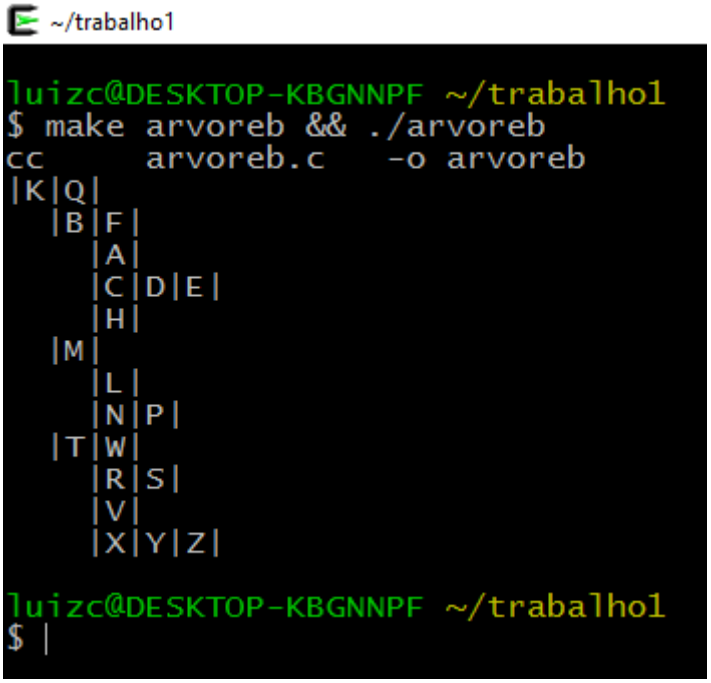

Figura 1: Teste 1 - Entrada: F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E - T = 2

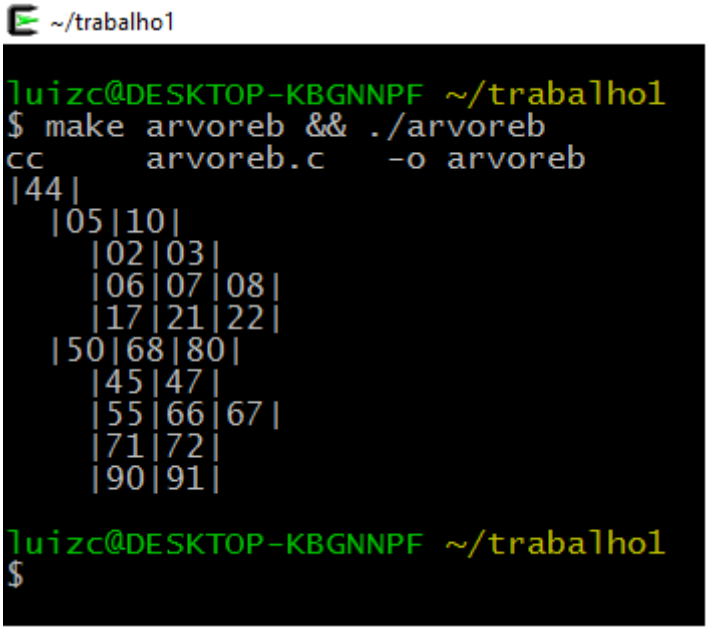

Figura 2: Teste 2 - Entrada: 91, 90, 80, 71, 72, 50, 45, 47, 10, 8, 7, 5, 2, 3, 22, 44, 55, 66, 68, 17, 6, 21, 67 - T = 3

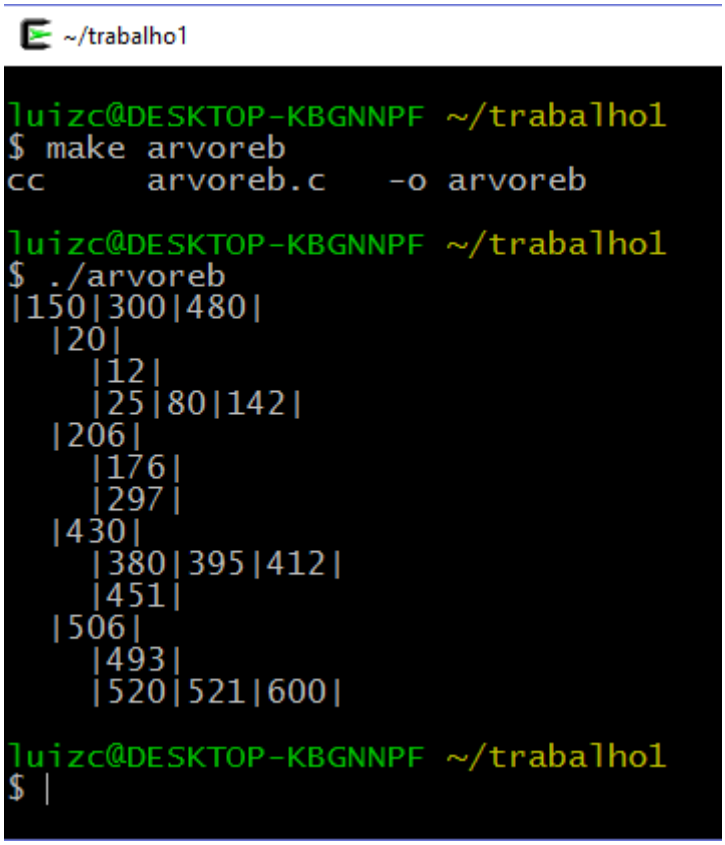

Figura 3: Teste 3 - Entrada: 300, 20, 150, 430, 480, 520, 12, 25, 80, 142, 176, 206, 297, 380, 395, 412, 451, 493, 506, 521, 600 - T = 2