UNIVERSIDADE ESTADUAL DE MONTES CLAROS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS – CCET

CORREÇÃO DE PROVAS DE CONCURSOS

Montes Claros, Minas Gerais
Junho de 2012

Patrick Pierre
Rodrigo Basílio

CORREÇÃO DE PROVAS DE CONCURSOS

Trabalho apresentado ao professor Renato Cota, como parte das exigências para avaliação da disciplina de Algoritmos e Estruturas de Dados II, 2º período.

Montes Claros – MG
Junho/2011

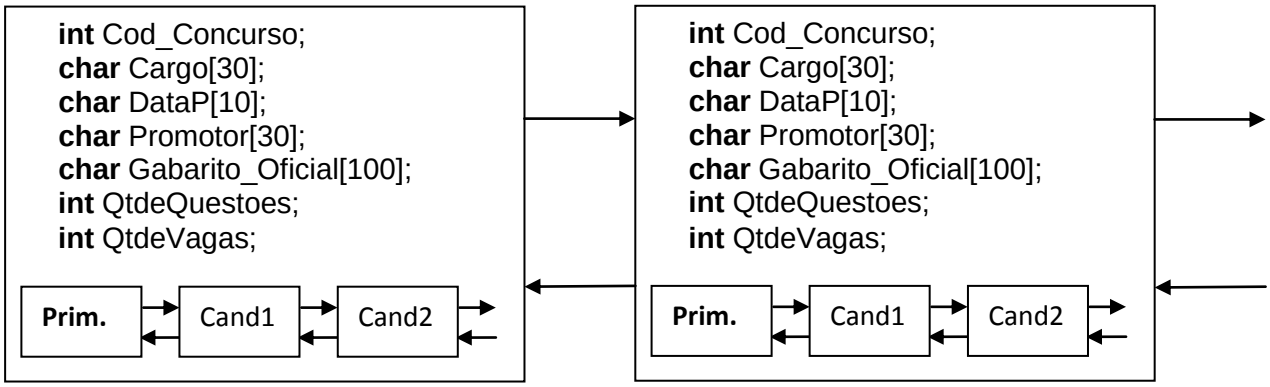
Introdução

O diretor da COTEC – Comissão Técnica de Concursos, órgão normalmente responsável pela execução de concursos em Montes Claros e região, está necessitando de um software para cadastro de concursos, candidatos e correção das provas. Neste caso, pede-se que o programa a ser desenvolvido tenha um modo para cadastrar os concursos, informando cargo, quantidade de questões da prova e o número de vagas disponíveis, salve o gabarito oficial do concurso (contendo as alternativas “A, B, C, D, E”), permita a inserção dos candidatos, calculando automaticamente a nota do candidato, e que possibilite também a pesquisa por nome e por código do candidato (número de inscrição).

Basicamente, o programa funciona da seguinte forma: foram criados dois tipos de listas, uma que poderíamos categorizar como uma “lista interna”, a lista de candidatos, e outra que poderíamos chamar de “lista externa”, a lista de concursos. Cada Tipo Concurso possui uma lista de candidatos em sua formação. Foram utilizados também uma arvore binária e uma tabela *hash* para pesquisa dos dados.

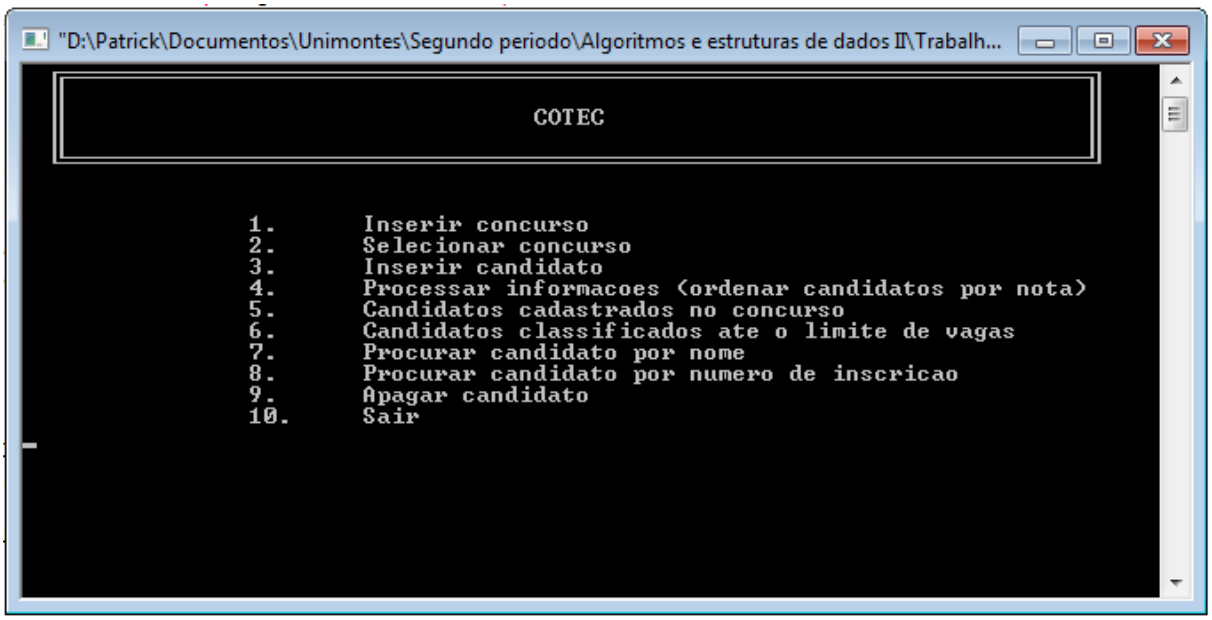
Implementação

Como dito anteriormente, foram utilizados duas listas para a solução do problema proposto, possibilitando o salvamento de múltiplos concursos em um mesmo programa sem que um alterasse em nada o outro. Desse modo, foram criadas duas listas, uma externa, a lista de concursos (*TipoLista_Concursos*) sendo que, cada *TipoConcurso* incluído na lista de concursos era constituída das variáveis inteiras *Cod_Concurso* (código de cadastro do concurso), *QtdeQuestoes* (quantidade de questões da prova a ser corrigida) e *QtdeVagas* (quantidade de vagas) e as strings Cargo (cargo concorrido), *DataP* (data da prova), Promotor (referente ao órgão para o qual o concurso está sendo realizado) e *GabaritoOficial*. Cada *TipoConcurso* possui também um *TipoLista_Candidatos*, onde foi salvo a lista de candidatos participantes do concurso, como a ideia mostrada no diagrama abaixo:



E cada *TipoCandidato* (Cand1, Cand2 no diagrama acima) possuem um código do candidato, gerado automaticamente pelo programa, uma variável duas strings (uma para salvar o nome e outra para o gabarito do candidato) e uma outra variável inteira Nota, onde vai ser atribuída a nota final do candidato após a inserção dos dados. A inserção dos dados na *Tabela Hash* por nomes e na *Arvore binária balanceada* acontece no mesmo momento em que são inseridos na lista de candidatos.

No programa, para os candidatos serem inseridos é necessário primeiro que se insira ao menos um *TipoConcurso* na lista de concursos, esse concurso é selecionado para que os dados possam ser inseridos nele. No menu inicial existe a opção para selecionar concurso, caso o usuário do software deseje mudar para um concurso cadastrado anteriormente.



Podemos ver na imagem acima as demais alternativas oferecidas para o menu principal e que serão explicadas no decorrer do desenvolvimento dessa documentação. A inserção dos concursos, assim como dos candidatos, são feitas em listas duplamente encadeadas, para dinamizar o processo de acesso dos dados. Como foi dito anteriormente, as notas dos candidatos é calculada no momento da inserção dos mesmos da seguinte maneira:

```
void CorrigeProva (TipoConcurso Conc, TipoCandidato *Cand){
    int x;
    Cand->Nota = 0;
    for (x = 0; x < Conc.QtdeQuestoes; x++){
        if (Cand->GabaritoCand[x] == Conc.Gabarito_Oficial[x]) (Cand->Nota)+=3;
        if (Cand->GabaritoCand[x] != Conc.Gabarito_Oficial[x] && Cand->GabaritoCand[x] != 'n') (Cand->Nota)--;
    }
    if (Cand->Nota < 0) Cand->Nota = 0;
} //Avalia nota do candidato;
```

```
printf ("\n\tEntre com os dados do candidato a ser inserido\n\n");
Cand.Cod_Candidato = ContaCand;
printf ("\tCodigo de cadastro: %d\n", Cand.Cod_Candidato);
printf ("\tNome: ");
fflush (stdin);
gets (Cand.Nome);
printf ("\tEntre com o gabarito do candidato: \n");
printf ("Obs.:\tQuestões para as questões que não possuirem respostas, entre com o valor -n-;");
printf ("\n\tSomente somente letras minusculas para o cadastro das alternativas são permitidos.");
for (x = 0; x < Conc.QtdeQuestoes; x++){
    printf ("\n\t%d. ", x+1);
    fflush(stdin);
    scanf ("%c", &conf);
    if (conf != 'a' && conf != 'b' && conf != 'c' && conf != 'd' && conf != 'e' && conf != 'n'){
        printf ("\n\tCaractere invalido!");
        x--;
    } else {
        Cand.GabaritoCand[x] = conf;
    }
}
CorrigeProva (Conc, &Cand);
InsereCandidato (&Conc.Lista, Cand);
posicao = CalculoPosicao (Cand);
InsereCandidato (&HashCand[posicao], Cand);
InsereSBB (Cand, &Arvore);
```

A opção “Processar informações” irá ordenar, utilizando o algoritmo de ordenação Quicksort, os dados por ordem de nota, em ordem decrescente. Para a utilização do Quicksort em uma lista duplamente encadeada foi necessário a criação de apontadores para percorrer a lista juntamente com Esq, Dir e o Pivot.

Seguindo a ordem, vem agora as opções para impressão na tela dos candidatos, a primeira irá exibir todos os candidatos cadastrados no concurso, a segunda irá exibir todos os candidatos classificados até o limite de vagas. Utilizamos para isso duas funções básicas de impressão, a primeira imprimindo todos os valores até que um apontador *aux* fosse igual a *NULL*, e a segunda percorrendo somente até que um contador alcançasse o número limite de candidatos.

A próxima opção é a primeira opção para busca, a opção de busca por nome do candidato. Para essa opção foi implementada o conceito de *Tabela Hash*, utilizando uma tabela de tamanho 13, método da divisão e listas encadeadas para o tratamento de colisões. Para calcular a posição das strings na *Hash* foi necessário a utilização de uma função presente na biblioteca *stdlib.h*, chamada *atoi* (ASCII to integer), cuja função é “converter” uma *string* em um valor numérico. Dessa forma podemos inserir ou encontrar dados na tabela *Hash* simplesmente fazendo a operação “*posição = (atoi(Cand.Nome))%13*”.

```
//====================== TABELA HASH ======================
void PesquisaHash (TipoCandidato *Cand, TipoLista_Candidatos *HashCand){
    Apontador Aux;
    int i;
    if (VaziaLista(*HashCand)){
        printf ("\n\t\tNao existem itens cadastrados!\n");
        getch();
        return;
    }
    system("cls");
    Logo();
    Aux = HashCand->Primeiro->Prox;
    while (Aux != NULL){
        if (strcmpi(Aux->Candidato.Nome, (*Cand).Nome) == 0){
            printf ("\n\t\tCadastro:\t%d", Aux->Candidato.Cod_Candidato);
            printf ("\n\t\tNome:\t%s", Aux->Candidato.Nome);
            printf ("\n\t\tNota:\t%d", Aux->Candidato.Nota);
            getch();
            return;
        }
        Aux = Aux->Prox;
    }
} //Pesquisa valor na Tabela Hash

int CalculoPosicao (TipoCandidato Cand){
    int i;
    i = atoi(Cand.Nome);
    return (i%MAX);
} //Calcula a posição do item na Hash
```

Tipos criados para a Tabela Hash:

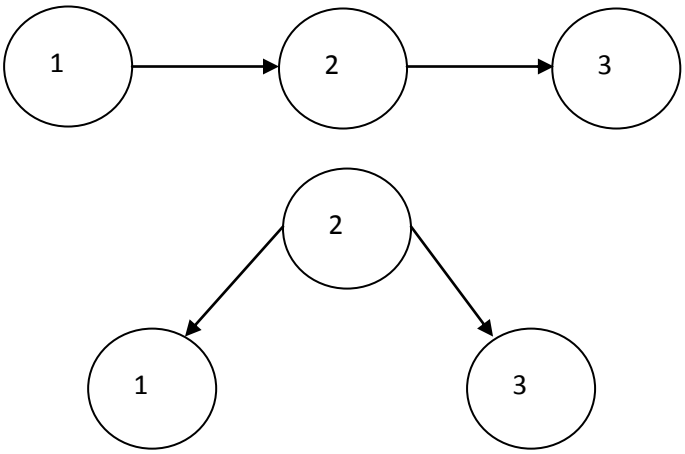
```
//====================== TABELA HASH ======================
typedef struct Cel *Indicate;

typedef struct Cel {
    TipoCandidato Cand;
    Indicate Prox;
}Celula_Hash;
```

A opção seguinte é a para busca por número, onde foi implementado o conceito de Árvore Binária Balanceada. Um fator importante a ser lembrado é que, apesar de o programa cadastrar concursos diferentes e os candidatos nos seus respectivos concursos, a árvore SBB e a tabela *Hash* recebem todos os candidatos cadastrados em todos os concursos. Diferente da Hash, a SBB apresenta alta complexidade de implementação. Inicialmente foi criado um tipo *ApontadorNo* que seria o nosso "tipo árvore", onde seriam inseridos os dados posteriormente. Também se fez necessário a criação de ponteiros para definir a inclinação (horizontal ou vertical) e a posição dos "filhos" (direita ou esquerda), da seguinte maneira:

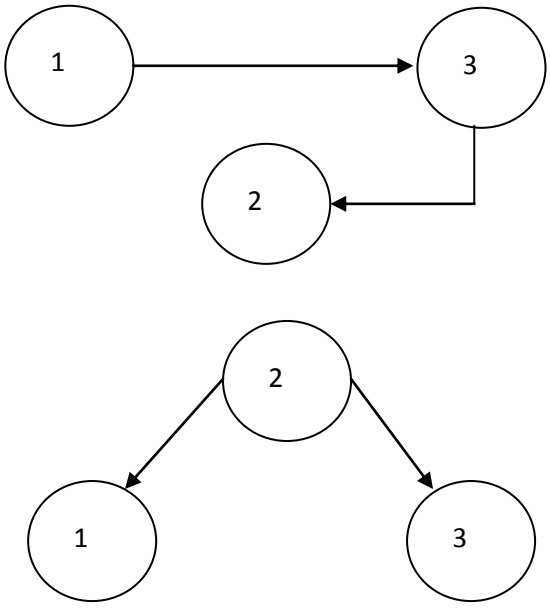
```
//======================== ÁRVORE SBB ========================
typedef enum {
    Vertical, Horizontal
} Inclinacao;

typedef struct no* ApontadorNo;

typedef struct no {
    TipoCandidato Cand;
    ApontadorNo Esq, Dir;
    Inclinacao BitE, BitD;
} No;
```

As funções de inserção de elementos na Árvore SBB vão possuir características básicas e regras que devem ser seguidas. A primeira regra importante é que todos os dados inseridos na tabela menores que o chamado nó raiz (ponto inicial da árvore) são inseridos a sua esquerda, e os maiores a sua direita. As demais regras também vão influenciar no posicionamento dos dados em relação ao nó "pai" em alguns casos específicos que serão ilustrados abaixo:

Dois apontadores na mesma direção e sentido, sequencialmente (direita-direita / esquerda-esquerda com a ordem de inserção 1, 2 e 3):



Dois apontadores sequenciais, mas com sentidos diferentes (direita-esquerda / esquerda-direita, com a ordem de inserção 1, 3 e 2):



O processo de busca funciona de uma maneira bem prática: como podemos ver, o nó pai liga-se sempre a dois nós filhos, um maior e um menor que ele. Assim é verificado se a chave do item procurado é igual, menor ou maior que a chave do nó em questão, sendo menor, passa-se a esquerda dessa árvore, maior, a direita. Uma coisa importante que não pode ser visualizado no diagrama acima é que, cada nó possui duas ligações. Mesmo os nós filhos das classes mais abaixo na árvore apontam para “NULL”. Deste modo, quando o apontador utilizado para percorrer a arvore for igual a NULL, saberemos que o item procurado não está posicionado na árvore.

Conclusão

Os métodos de pesquisa por Hashing e Árvore SBB se tornam estritamente necessários, principalmente quando lidamos com quantidades de dados muito grandes. Acessar os dados cadastrados em um banco de dados muito grande por uso de pesquisa sequencial (o modo mais simples de pesquisa), tornaria o processo de busca extremamente lento, sendo assim, inutilizável.A *Hash* e SBB possibilitam ao utilizador rapidez na obtenção de informações gravadas na memória.

A maior dificuldade de implementação foi em relação às funções da árvore SBB, pois esta possui uma lógica mais complexa e que, às vezes, pode gerar certa confusão e o método para implementar a pesquisa por nome de forma mais rápida, a função *atoi*, que permite a “transformação” de strings em valores, que nos era desconhecida.

O *software* foi desenvolvido para simular cadastros de concursos, candidatos e correção das provas, em que mediante os testes de mesa e execuções apresentou desempenho satisfatório,atendendo ao modelo pedido, servindo sua criação para fixar a pratica de programação e fazer com que os criadores exercitem e tenha o domínio da linguagem.

BIBLIOGRAFIA

ASCENCIO, Ana F.G; CAMPOS, Edilene A. V. Fundamentos da Programação de Computadores: Algoritmos, Pascal, C/C++ e Java. 2. Ed. São Paulo, SP: Pearson Prentice Hall, 2007

ZIVIANI,Nivio.Projeto de Algoritmos com Implementações em Pascal e C.Ed.Pioneira.São Paulo,SP:Cengacge Learning,2007.