UNIVERSIDADE FEDERAL DE MINAS GERAIS
ALGORITMOS E ESTRUTURA DE DADOS II

Trabalho Prático 2 – Tabela do Brasileirão

Deiziane Natani da Silva – TA1

1. Introdução

O trabalho tem como objetivo praticar os conceitos de ordenação, árvores e busca. O TP consiste em criar uma tabela do campeonato brasileira que é ordenada a cada rodada pelos métodos Quicksort e Shellsort.

Além disso, também é criada uma tabela hash para armazenar os jogos para eventuais buscas. Os problemas de colisão da tabela são resolvidos através de uma árvore SBB e listas.

2. Modelagem e Funcionamento

Na ordenação, foram criadas várias funções em que uma chama a outra e retorna o valor até a primeira função chamada. Os resultados são ordenados utilizando-se vários critérios. O primeiro deles é pelos pontos obtidos pelo time. A cada vitória, um time ganha 3 pontos, em empates cada um dos times ganha 1 ponto, e em derrotas o time perdedor não ganha nenhum ponto. O segundo critério de ordenação, caso haja empate por pontos, é por número de vitórias. O terceiro, em caso de empate por vitórias, é por saldo de gols. O quarto é por número de gols pró. Se após passarem por todos ainda houver empates, os times empatados são ordenados por ordem alfabética **crescente**.

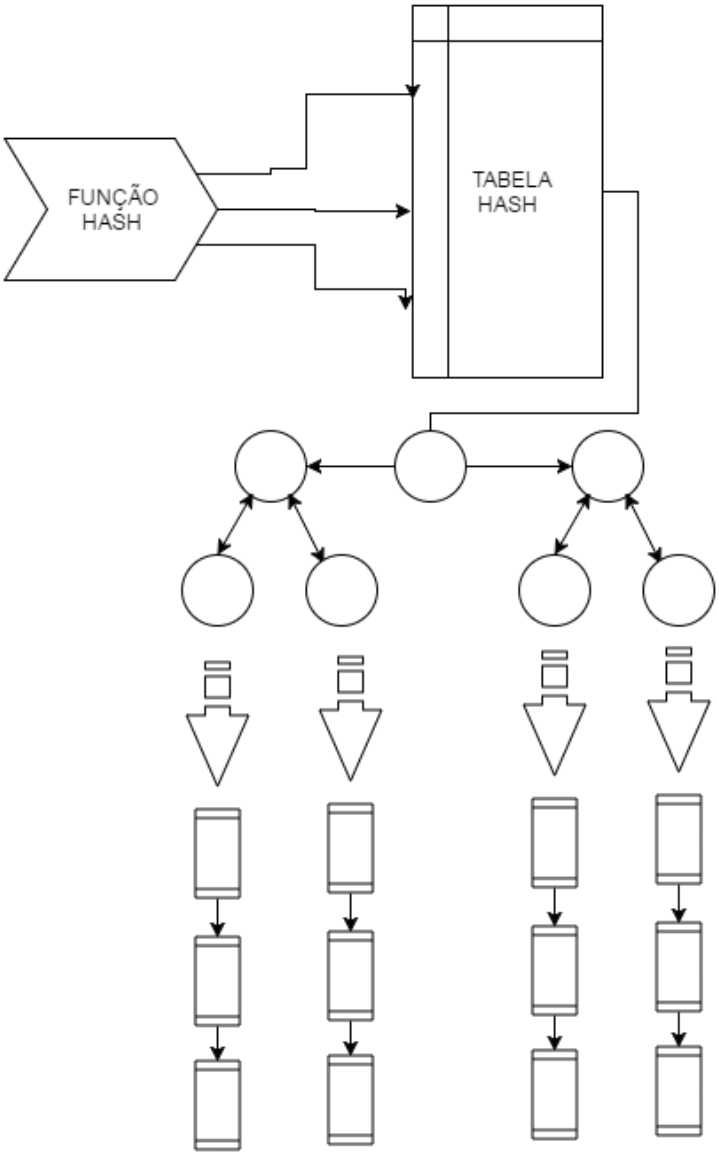
Os resultados são armazenados na tabela hash através de uma função que converte a data do jogo em um número, para em seguida fazer a operação mod o tamanho da tabela. Se houver colisões, é criada uma árvore SBB, em que cada nó possui um ponteiro para uma lista encadeada, que resolve o problema em que vários jogos são realizados no mesmo dia.

3. Implementação

Estrutura de Dados

A tabela hash implementada funciona da seguinte maneira:

A função é chamada, e através dela é gerada uma chave que indica a posição em que o elemento deve ficar na tabela. Caso mais de um item receba a mesma posição, é criada uma arvore SBB que armazena esses valores. Como a chave é a data e podem ocorrer mais de um jogo no mesmo dia, vão haver casos em que a posição da árvore já está ocupada. Nesse caso, é criada uma lista encadeada em cada nó em que haja conflitos. A estrutura está representada pelo diagrama abaixo:



Também é criada uma tabela de classificação em que os itens são ordenados segundo os critérios descritos no item 2.

Funções e Procedimentos

Para a tabela de classificação, foram criadas 5 funções auxiliares:

```
Table* auxiliarP(Table* t, int i, int j, int sizeT, int ord);
Table* auxiliarV(Table* t, int i, int j, int sizeT, int ord);
Table* auxiliarS(Table* t, int i, int j, int sizeT, int ord);
Table* auxiliarGP(Table* t, int i, int j, int sizeT, int ord);
Table* auxiliarA(Table* t, int i, int j, int sizeT, int c, int ord);
```

Essas funções têm como objetivo chamar a função de ordenação definida no arquivo de entrada. A primeira chama a ordenação por pontos, após feita, é verificado se existem empates. Caso existam, a próxima função auxiliar é chamada. Esta ordena por número de vitórias e, da mesma forma, é verificado se existem empates. Caso existem, a próxima

função auxiliar é chamada, e assim sucessivamente até a última função auxiliar que ordena por ordem alfabética. Essa função, diferentemente das outras auxiliares, sempre chama a ordenação por ordem alfabética repetidas vezes até que não haja empates, verificando caractere por caractere da palavra.

As funções que verificam empates são as seguintes:

int verificaEmpatesP(Table* t, int i, int j, int sizeT);*
int verificaEmpatesV(Table* t, int i, int j, int sizeT);*
int verificaEmpatesS(Table* t, int i, int j, int sizeT);*
int verificaEmpatesGP(Table* t, int i, int j, int sizeT);*
int verificaEmpatesA(Table* t, int i, int j, int sizeT, int c);*

Estas, ao encontrarem mais de um item com o mesmo valor, seja de pontos, vitorias, saldo de gols etc, retornam um vetor de inteiros com os índices da tabela que estão empatados. Caso não haja empates, o vetor retornado pelas funções tem -1 como valor de todas as posições. Dessa forma, nas funções auxiliares é feita a verificação dos valores desse vetor, e caso sejam diferentes de -1, é identificado que há empates.

As funções que realmente fazem o processo de ordenação são as seguintes:

Quicksort
Table ordenaQuickP(Table* t, int i, int j, int sizeT);*
int particionaQuickP(Table t, int i, int j);*
Table ordenaQuickV(Table* t, int i, int j, int sizeT);*
int particionaQuickV(Table t, int i, int j);*
Table ordenaQuickS(Table* t, int i, int j, int sizeT);*
int particionaQuickS(Table t, int i, int j);*
Table ordenaQuickGP(Table* t, int i, int j, int sizeT);*
int particionaQuickGP(Table t, int i, int j);*
Table ordenaQuickA(Table* t, int i, int j, int sizeT, int c);*
int particionaQuickA(Table t, int i, int j, int c);*

Shellsort
Table ordenaShellP(Table* tb, int sizeT);*
Table ordenaShellV(Table* tb, int sizeT);*
Table ordenaShellS(Table* tb, int sizeT);*
Table ordenaShellGP(Table* tb, int sizeT);*
Table ordenaShellA(Table* tb, int sizeT, int c);*

Essas funções são chamadas por suas respectivas funções auxiliares. E retornam a tabela ordenada de acordo com cada critério.

A função que insere os itens na tabela, é a seguinte:

void insere2(Table t, char* time1, char* time2, int p1, int p2, int ord, int size);*

Essa função, recebe uma tabela já criada e todos os dados de cada jogo da rodada. Como a tabela já foi preenchida com os times na main, é feita uma busca na tabela, e ao encontrar o time, os dados sobre ele são atualizados.

As funções citadas acima se encontram nos arquivos TabelaClassificacao.c e TabelaClassificacao.h

Para a tabela hash foram implementadas as seguintes funções:

*HashTable *criaHash(int hash_size);*
int funcaoHash(int chave, int hash_size);
void insereHash(HashTable h, Jogos s, SBBTree *raiz);*
void buscaHash (HashTable h, int chave, SBBTree *raiz, char* data);*
*int transformaChave(char *str);*

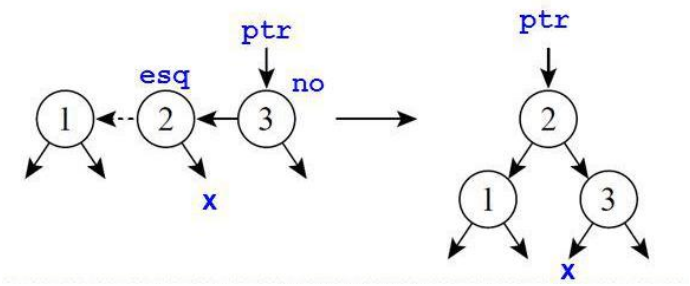
A primeira cria a tabela com o tamanho definido pelo arquivo de entrada. A tabela é feita utilizando ponteiro para ponteiro e todos os itens da tabela são inicializados com NULL. A função transformaChave, recebe a string data e transforma o valor de seus caracteres em um número inteiro. Após ser transformado, o valor é enviado para a funçãoHash, e está realiza uma operação que determina a posição em que um item vai ficar na tabela. A função utilizada foi escolhida pela simplicidade de implementação. Ela usa o método da congruência linear que calcula o resto da divisão do valor da chave pelo tamanho da tabela. A operação & bit a bit com o valor 0x7FFFFFFF elimina o bit de sinal da chave e com isso evita o risco de ocorrer um overflow ao obter um numero negativo.

A função insereHash insere o item na tabela em sua posição, caso a posição já esteja ocupada, o item é inserido na árvore. A buscaHash é utilizada na pesquisa de itens.
As funções citadas acima estão implementadas nos arquivos TabelaHash.c e TabelaHash.h.

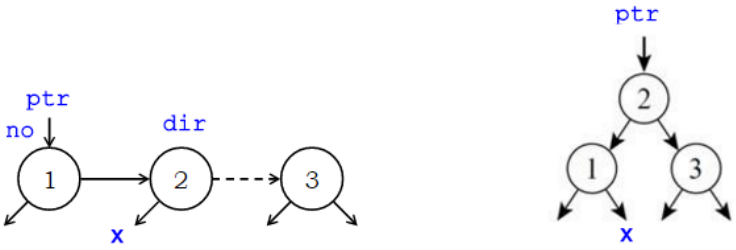
A árvore SBB utiliza as seguintes funções:

*void iinsere(Dados d, SBBTree *ptr, int *incli, int *fim);*
*void iinsere_aqui(Dados d, SBBTree *ptr, int *incli, int *fim);*
*void insere (Dados d, SBBTree *raiz);*
*void pesquisaTree(SBBTree *raiz, int chave, char* data);*
*void ee(SBBTree *ptr);*
*void ed(SBBTree *ptr);*
*void dd(SBBTree *ptr);*
*void de(SBBTree *ptr);*
*void percorreArv(SBBTree *raiz);*

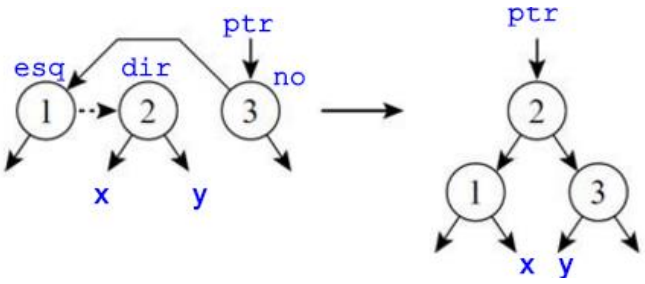
As três primeiras funções são utilizadas para inserir um item na arvore. A função ee resolve casos em que existem dois apontadores horizontais seguidos a esquerda.



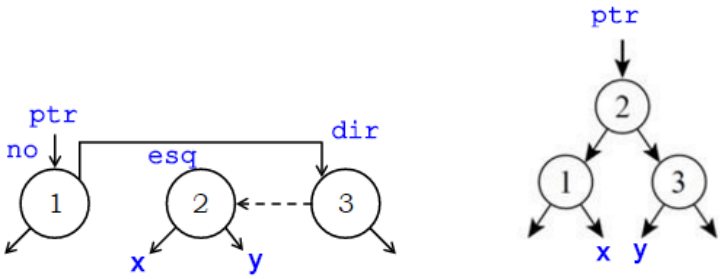
A função dd resolve os casos em que são ponteiros horizontais a direita e esquerda



A função ed resolve casos em que são ponteiro horizontais a esquerda e a direita



A função de resolve casos em que são ponteiros horizontais a direita e a esquerda



As funções citadas acima estão implementadas nos arquivos ArvoreSBB.c e arvoreSBB.h.

Na estrutura de dados Lista, foram utilizadas as seguintes funções:

Lista criaLista();*
*void criaNode(Lista *l, Data d); //cria no da lista*
void imprimeLista(Lista l, char* data); //imprime lista (utilizada apenas para testes)*
bool vazia(Lista l);*

A primeira cria uma lista vazia, a segunda cria um nó na lista. A imprimeLista escreve os itens procurados no arquivo de saída. A última verifica se a lista está vazia.

Programa Principal

O programa principal main recebe um arquivo txt como entrada. São lidas todas as informações referente ao TP, como o número de times, rodadas, ordenação etc. Nele também é criada a tabela hash, tabela da classificação e arvore SBB. Após ler as informações de cada rodada, elas são colocadas na tabela de classificação e são devidamente ordenadas de acordo com a ordenação escolhida. Além de também armazenar os jogos na tabela hash de pesquisa. Também são lidas as datas de pesquisa.

Organização do Código, Decisões de Implementação e Detalhes Técnicos

O programa está dividido em 9 arquivos: *main.c, TabelaHash.c, TabelaHash.h, TabelaClassificacao.c, TabelaClassificacao.h, ArvoreSBB.c, ArvoreSBB.h, Lista.c e Lista.h.*

Na minha implementação foi utilizada apenas uma arvore SBB e cada nó dessa árvore possui um ponteiro para uma lista encadeada. A IDE utilizada foi o CodeBlocks 16.01, no sistema operacional Windows 7.

4. **Análise de Complexidade**

A análise de complexidade será feita em função do número de rodadas e número de times.

Função insere2(): São feitos dois loops, o primeiro percorre a tabela de classificação até encontrar o time 1 do jogo em questão. Como essa tabela tem o tamanho do número de times existentes, esse loop pode ser classificado como O(n). Em seguida existe outro loop que também percorre a tabela até encontrar o time 2. Esse loop também pode ser classificado como O(n). Logo, a função será O(n)+O(n) = **O(n).**

Funções verificaEmpatesP, verificaEmpatesV, verificaEmpatesS, verificaEmpatesGP, verificaEmpatesA: As funções possuem dois loops aninhados que executam em função do tamanho da tabela de classificação, ou seja, o número de times. Sendo assim, se nenhum item estiver empatado, no pior caso a complexidade será **$O(n^2)$.**

Funções Quicksort: A complexidade dos algoritmos de ordenação quicksort são no caso médio **O(n log n).** No pior caso esse valor pode ser quadrático **$O(n^2)$.**

Funções Shellsort: A complexidade dos algoritmos de ordenação shellsort pode ser definida como **$O(n^{1.25})$ ou $O(n\ (\ln n)^2)$.**

Função buscaHash(): Como a busca é feita numa tabela hash, sua complexidade pode ser definida como **O(1).**

Função pesquisaTree(): A complexidade do pior caso é igual à altura da árvore. O pior caso pode ser O(n), onde n é o número de nós da árvore. Quando a árvore está balanceada, a busca é eficiente e o pior caso é **O(log n).**

*Função iinsere():*A inserção precisa localizar o local para fazer a inserção. Como a árvore SBB é quase balanceada, a complexidade da inserção é **O(lg n).**

5. **Testes e Resultados**

Foram feitos testes utilizando os casos de testes disponibilizados no moodle.

Percebi, ao executar todos os testes utilizando tanto o Quick quanto o Shell, que geralmente o Shellsort levava menos tempo para fazer a ordenação. A diferença, porém, é bem sutil. Isso pode ter se dado porque o tamanho dos arquivos são relativamente pequenos, um caso onde o Shellsort é bem eficiente.

Nos casos 7 e 8, como existem muito mais times (400 e 1000), é possível perceber uma pequena diferença entre os dois métodos. Dessa vez, o Quicksort aparenta ser mais rápido que o Shellsort ao fazer a ordenação.

6. Conclusão

O objetivo foi atingido e o algoritmo implementado realiza todas as funções que deveria. A maior dificuldade encontrada por mim ao desenvolver o TP diz respeito a ordenação dos itens. Necessitei de fazer várias funções, praticamente idênticas, que mudavam apenas alguns parâmetros e isso acabou tornando o código muito grande e cansativo para debug em caso de erros. Em relação a tabela hash, não tive nenhum problema em relação a sua implementação. Já na arvore SBB, tive um pouco de dificuldade em entender como ela funciona. O material disponibilizado pelo professor ajuda bastante, porém alguns pontos ainda se tornam um pouco difíceis de serem compreendidos, e o material sobre esse tipo de arvore na internet é bem escasso. A lista utilizada por mim neste TP já foi utilizada também no TP anterior, por isso também não tive nenhum problema na sua implementação.

7. **Referências**

Na implementação da tabela hash, utilizei a estrutura mostrada nesse vídeo:

Linguagem C Programação Descomplicada. Tabela Hash. Disponível em: <https://www.youtube.com/playlist?list=PL8iN9FQ7_jt7GZiYfxGIb7sZJOQhw3Xr7>. Acesso em: 15 nov 2016.

O algoritmo do Quicksort foi retirado de:

Linguagem C Programação Descomplicada. Ordenação - QuickSort. Disponível em: < https://www.youtube.com/watch?v=spywQ2ix_Co>. Acesso em: 19 nov 2016.

A algoritmo da arvore SBB foi retirado de:

<http://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/sbbs.pdf>. Acesso em: 20 nov 2016.