Relatório EP2 - MAC 0323

Algoritmos e Estrutura de Dados II

Odair Gonçalves de Oliveira - 13671581

Índice

Introdução

Sobre o EP2:

O EP2 de MAC 0323 - Algoritmos e Estrutura de Dados consistia na implementação de 5 diferentes estruturas de dados para armazenar palavras de um texto dado como entrada e permitir a realização de algumas consultas sobre os dados armazenados de cada chave inserida na estrutura.

As estruturas de dados a serem implementadas eram:

- [VO] - Vetor Dinâmico Ordenado;
- [ABB] - Árvore de Busca Binária;
- [TR] - Treaps;
- [A23] - Árvores 2-3;
- [ARN] - Árvores Rubro Negras;

E para cada estrutura era necessário a implementação das seguintes funções:

- [void add(Key key, Item val)] - Função que adiciona o par (key, val) na estrutura;
- [Item value(Key key)] - Função que verifica se existe uma chave key na estrutura e devolve o valor associado a essa chave caso ela esteja presente na estrutura;

Em relação ao campo ‘value’ de cada Item adicionado, a estrutura escolhida foi a seguinte:

- [int tam] - Quantidade de letras presente na palavra chave;
- [int vog] - Quantidade de vogais sem repetição na palavra, caso a palavra apresente vogais repetidas, seu vog recebe o valor de 0;
- [int repet] - Quantidade de vezes que a palavra se repetiu dentro do texto dado como entrada;

Já as consultas a serem realizadas estão descritas abaixo:

- [F] - Palavras mais frequentes no texto;
- [O ‘palavra’] - Dada uma palavra, devolve o número de vezes que ele ocorreu no texto;

- [L] - Lista das palavras mais longas presentes no texto;
- [SR] - Lista das maiores palavras que não repetem letras;
- [VD] - Lista das menores palavras com mais vogais sem repetição;

Explicações Iniciais:

Para esse EP implementei 2 modos de uso, que aparecem como uma opção que pode ser escolhida assim que ele estiver rodando. Para compilar o programa, usei o seguinte comando:

```
g++ estruturas.cpp EP2.cpp -o EP2
```

Ao rodar o comando o [`g++ -Wall estruturas.cpp EP2.cpp -o EP2`] notei a presença de alguns ‘warning’s sobre a comparação de uma variável do tipo int com variáveis que são do tipo ‘long unsigned int’ e ‘size_type’, porém, em nenhum dos testes executado essa comparação deu algum problema.

Os modos do EP são:

- [1] - Modo EP2 Enunciado - Que é a versão solicitada no enunciado do EP2;
- [2] - Modo EP2 Testes - A versão que utilizei para realizar os testes expostos mais adiante;

Vale também destacar que a ordenação das palavras, em todas as estruturas, modos e arquivos foi feita utilizando a função strcmp(palavra1, palavra2). Ou seja, a ordem considerada é a seguinte: (A < B < C < ... < Z < a < b < c ... < z), já que strcmp compara o valor do caractere na tabela ASCII.

Apesar de ser possível implementar uma função própria de comparação para utilizar uma outra ordenação de letras, como não foi especificado que era necessário, além de não ser o intuito principal desse EP, decidi utilizar a função já pronta mesmo, sem nenhuma adaptação. Dessa forma, é considerado, por exemplo, que a palavra ‘Cavalo’ é diferente de ‘cavalo’ que é diferente de ‘cAvalo’ e assim por diante.

Outra questão com a leitura das palavras que não foi abordada é a separação de palavras com hífen. No caso de uma palavra que esteja separada por um hífen numa quebra de linha, as sílabas que ficaram em uma linha serão consideradas uma palavra e as que ficaram na outra, serão consideradas outra palavra. Novamente, como achei que não era necessário nenhum cuidado sobre esses casos, a implementação realizada não considera essa possibilidade.

Por fim, nas leituras, foram utilizadas funções que já realizam o recorte das palavras, retirando determinados símbolos e pontuações. Números foram considerados palavras também.

MODO 1 >>

O Modo 1 funciona exatamente como o enunciado propõe. Ao começo deve-se escolher a estrutura que deseja testar, o número de palavras a serem testadas e por fim deve-se inserir as palavras:

```
Escolha a estrutura a ser utilizada:
  [ VO  ] - Vetor Dinâmico Ordenado
  [ ABB ] - Árvore de Busca Binária
  [ TR  ] - Treaps
  [ A23 ] - Árvores 2-3
  [ ARN ] - Árvores Rubro-Negras

   >> [ Digitar o código da estrutura escolhida ]
Digite o número de palavras: [ Número de palavras ]

Digite as palavras: [ Inserir palavras aqui ]
```

A leitura é feita linha por linha, sendo o número máximo de caracteres por linha igual a 100.000 (parâmetro determinado pela constante global CHAR_MAX_LINHA no arquivo EP.cpp). Para a parte das consultas, deve-se primeiro digitar o número de consultas que deseja fazer e logo após digitar o código em caracteres associado a cada tipo de consulta. Veja um exemplo:

```
Hora das Consultas:
 [ F  ] - Palavras mais frequente;
 [ O  ] 'termo' - Quantas vezes 'termo' aparece no texto;
```

```
[ L  ]- Palavras mais longas;
[ SR ] - Maiores palavras sem repetição;
[ VD ] - Menores palavras com mais vogais sem repetição;

Digite o número de consultas que deseja fazer:
   >> [ Inserir número de consultas (Valor máximo = 5) ]

Digite as consultas:

   >> [ Colocar os códigos de cada consulta desejada ]
```

Para a consulta O em específico, deve-se digitar ‘O’ + espaço + palavra que deseja buscar. Após terminar de digitar as consultas, basta pressionar ‘enter’ para ver o resultado delas e encerrar o programa.

A implementação de cada uma das estruturas será abordada posteriormente, já em relação ao processamento dos dados para a realização das consultas, como o professor autorizou no enunciado pré processar os dados e não necessariamente realizar as consultas após a inserção nas estruturas, criei um vetor de chaves e algumas variáveis globais para cada uma das consultas a serem realizadas:

- `[ vector<char *> f_words ]` - Vetor para palavras mais frequentes;
- `[ long long int max_freq = 1 ]` - Valor para determinar qual foi a maior frequência. Se a frequência da palavra que acabou de ser inserida for igual a esse valor, eu coloco ela no vetor. Caso seja maior, eu limpo o vetor e coloco ela. Caso seja menor, não faço nada. Essa mesma ideia é repetida de forma análoga nas análises abaixo:

- `[ vector<char *> sr_words ]` - Vetor para MAIORES palavras sem repetição de letras;
- `[ long long int max_tam_sr = 0 ]` - Valor para determinar qual foi o maior tamanho de palavra sem repetição de letra;

- `[ vector<char *> srv_words ]` - Vetor para MENORES palavras com mais vogais sem repetição;

- [long long int max_vog = 0] - Valor para maior número de vogais sem repetição entre todas as palavras;
- [long long int min_tam_srv = 101] - Valor para menor tamanho de palavra com vog = max_vog;

- [vector<char *> max_words] - Vetor para guardar as palavras com maior tamanho;
- [long long int max_size = 1] - Valor para o maior tamanho de palavra que já foi inserida na estrutura;

Assim, a cada nova inserção é checado se devo colocar a palavra em algum dos vetores a partir dos valores passados como variáveis globais.

Além disso, para o objeto da classe Item, está definido uma espécie de objeto 'nullptr' artificial. Como a função de busca retorna diretamente um valor do tipo Item, no caso em que a palavra buscada não está presente na estrutura, é criado um item artificial com os caracteres "##!PALAVRAERRO!##" que tem -1 como o valor de repet. Assim, caso não haja a palavra buscada, é devolvido um item com valor de repet = -1 e, a partir disso, é possível concluir a inexistência da palavra no texto.

MODO 2 >>

O Modo 2 foi construído para a execução dos testes para comparação das estruturas. Ao invés de fazer a leitura do texto diretamente pelo terminal, a leitura é executada em um dos 3 arquivos enviados em conjunto com os códigos. São 4 arquivos no total com as seguintes características:

- teste_oc.txt - Uma lista com 256.000 palavras distintas ordenadas em ordem crescente. Considerando (A < B < C < ... < Z < a < b < c ... < z).
- teste_od.txt - Uma lista com 256.000 palavras distintas ordenadas em ordem decrescente. Considerando (A < B < C < ... < Z < a < b < c ... < z).
- teste_a.txt - Uma lista com 256.000 palavras com repetições e sem nenhuma ordenação.

- teste_640k.txt - Uma lista com 640.000 palavras com repetições e sem nenhuma ordenação.

Para os testes, é recomendável não testar com mais de 87k palavras para as versões ordenadas, pois, pelo fato de estar calculando o número de comparações, alturas das árvores, etc., esses valores podem estourar o long long int utilizado e resultar em uma falha de segmentação. Para as versões aleatórias, tudo rodou bem, já que existiam várias palavras repetidas. Ou seja, pode testar com o máximo de 640k palavras.

Ao rodar o programa e escolher o modo de testes, deve-se escolher se deseja fazer o teste com as palavras ordenadas de forma crescente, ordenadas de forma decrescente ou com elas em ordem aleatória. Após escolher, basta digitar o número de palavras que deseja testar. O exemplo abaixo indica em vermelho as entradas. O número em "???" indica o valor máximo recomendável para o teste.

```
Escolha modo:
 1 - EP2 Versão Enunciado
 2 - EP2 Versão Testes
 >> 2
 ----------- EP2 - Versão de Testes ------------

 >> Digite o tipo de teste que deseja realizar:

 1 - Lista de palavras em ordem crescente;
 2 - Lista de palavras em ordem decrescente;
 3 - Lista de palavras em ordem aleatória;
 >> [ Digitar a opção que indica com qual arquivo deseja
realizar o teste ]

 >> Digite o número de palavras que deseja testar:
[TAM_MAX = ???]
 >> [ Digite número de palavras ]
```

O código então irá inserir as palavras em todas as estruturas criadas, então, dependendo do número de palavras, pode-se demorar um pouco. Ao finalizar a inserção, será mostrado o menu abaixo, onde você pode escolher visualizar a impressão dos dados em cada uma das estruturas para verificar se deu tudo certo.

```
Digite os códigos para impressão dos dados da estrutura:

 1 - Vetor Ordenado - Palavras Ordenadas
 2 - Árvore de Busca Binária - In Order
 3 - Árvore de Busca Binária - Pre Order
 4 - Treap - In Order
 5 - Treap - Pre Order
 6 - Árvores Rubro Negras - In Order
 7 - Árvores Rubro Negras - Pre Order
 8 - Árvores 2-3 - In Order
 9 - Árvores 2-3 - Pre Order
 0 - Sair
```

Ao pressionar 0 serão exibidos os resultados dos testes no seguinte formato:

```
>>> Resultados para testes com 256000 palavras!

 1 - VETOR ORDENADO:
Número de Comparações Inserção.......3154712
Número de Trocas.....................73025160

 2 - ÁRVORE DE BUSCA BINÁRIA:
Número de Comparações Inserção.......3074320
Altura...............................55

 3 - TREAPS:
Número de Comparações Inserção.......4135076
Altura...............................34
Número de Rotações...................33187

 4 - ÁRVORES RUBRO NEGRAS:
Número de Comparações Inserção.......2542272
Altura...............................18
Número de Rotações...................20232

 5 - ÁRVORE 2-3:
Número de Comparações Inserção.......2874564
Altura...............................11
Número de Quebras....................12756
```

A existência dessa versão é puramente para verificar o funcionamento das estruturas e realizar as comparações que serão expostas a seguir.

Estruturas Implementadas

Vetor Dinâmico Ordenado (VO):

Todas as implementações das estruturas foram realizadas utilizando as classes do C++ para maior praticidade no cálculo de alguns parâmetros utilizados nas comparações das estruturas e para aprofundar meu conhecimento sobre o uso das classes.

A estrutura do vetor dinâmico ordenado utilizada foi a seguinte:

```
typedef struct {
   Key key;
   Item val;
} vo;

class VO {
   public:
       long long int fim;
       vo * vetor;

       long long int n_comp_insercao;
       long long int n_comp_busca;
       long long int n_trocas;
       VO(long long int n);
       void add(Key key, Item val);
       Item value(Key key);
       long long int busca(Key key);
       void printa();
```

```
};
```

Como o número de palavras a serem inseridas já era dado na entrada, já aloquei o tamanho necessário para caber todas as palavras.

A função para adicionar já insere as chaves de forma ordenada. É realizada uma busca binária para descobrir em qual posição essa palavra deve ser inserida. Se estiver vazia, apenas coloca e atualiza-se o parâmetro ‘fim’ que guarda o índice do último elemento do vetor. Caso não esteja vazia, todas as palavras a partir daquela posição em diante serão empurradas uma posição para dar espaço para a nova palavra inserida.

Por tal fato, o vetor dinâmico realiza muitas trocas entre os elementos quando temos uma lista de palavras ordenadas de forma decrescente ou aleatória. Quando está ordenado, a inserção é sempre feita no final, então não ocorrem trocas.

Os parâmetros da class VO, alguns comuns também para outras classes, são os seguintes:

- `[ fim ]` - Guarda o índice do último elemento inserido, é utilizado para realizar a busca binária;
- `[ n_comp_insercao ]` - Guarda o número de comparações realizado na inserção dos elementos (COMUM COM OUTRAS ESTRUTURAS);
- `[ n_comp_busca ]` - Dada uma determinada palavra buscada, guarda o número de comparações realizada ao se buscar a palavra (COMUM COM OUTRAS ESTRUTURAS);
- `[ n_trocas ]` - Guarda o número de trocas realizadas quando é necessário empurrar os elementos uma casa para frente;

Destaca-se que a função ‘busca’ retorna um valor inteiro. Caso esse inteiro seja -1, isso indica que o elemento não encontra-se no vetor.

No início tive alguns problemas com falha de segmentação no momento de empurrar os elementos para as posições da frente, possivelmente por estar acessando posições que não eram válidas, mas após alguns testes e ajustes deu tudo certo nesses índices deu tudo certo.

Árvore de Busca Binária (ABB):

Para ABB utilizei uma classe do C++ e um struct para os nós. Sua estrutura é a seguinte:

```
typedef struct cel_abb {
   Item val;
   Key key;
   struct cel_abb * esq;
   struct cel_abb * dir;
} abb;

class ABB {
   public:
       abb * arvore;
       long long int n_comp_busca;
       long long int n_comp_insercao;
       long long int altura;

       // Cria um objeto do tipo ABB - Árvore de Busca Binária
       ABB();
       // Adiciona um item na ABB.arvore;
       void add(Key key, Item val);
       abb * put(Key key, Item val, abb * raiz, long long int n);
       void print_in_order(abb * raiz);
       void print_pre_order(abb * raiz);
       Item value(Key key);
       abb * busca_aux(Key key, abb * raiz);
};
```

Cada nó tem um ponteiro para a esquerda e direita, uma chave Key e um item val. O único parâmetro diferente nesse caso é a altura, que já é calculada/atualizada na inserção de cada novo elemento.

Em relação às funções, a função ‘put’ é recursiva e a inserção é sempre feita em uma folha nullptr. Já as funções de ‘print’ servem apenas para verificar se a estrutura está funcionando corretamente. Enquanto o ‘in order’ permite verificar a correta ordenação dos elementos, o ‘pre order’ permite reconstruir a árvore manualmente,

inserindo os elementos na ordem que é impressa para verificar como ela ficou ao final. Essas duas funções são comuns para todas as outras árvores.

Sobre a função de ‘busca’, ela retorna um ponteiro para o nó que contém a chave procurada e caso a chave não esteja lá, ela devolve nullptr e então é criado o Item com a palavra “##!PALAVRAERRO!##" como foi descrito anteriormente.

Para essa estrutura não ocorreram muitos problemas e ela comportou-se como esperava-se nos testes.

TREAP (TR):

TREAPs são uma mistura de árvores de busca binária com heaps. Sua propriedade é a seguinte: Para cada nó da árvore temos que a subárvore esquerda contém descendentes com valores de chave menores e na subárvore direita contém os descendentes com valores de chave maiores e nenhum desses descendentes tem prioridade maior que o nó ancestral.

Assim, a estrutura da TREAP implementada foi a seguinte:

```
typedef struct cel_treap {
   Item val;
   Key key;
   cel_treap * esq;
   cel_treap * dir;
   long long int prioridade;
} tree_heap;

class TREAP {
   public:
       tree_heap * treap;
       long long int n_comp_busca;
       long long int n_comp_insercao;
       long long int n_rotacoes;
       long long int valor_max_prioridade;
```

```
        TREAP(long long int n);
        void add(Key key, Item val);
        tree_heap * put(Key key, Item val, tree_heap * raiz);
        tree_heap * rotaciona(tree_heap * p, char lado);
        Item value(Key key);
        tree_heap * busca(Key key, tree_heap * raiz);
        void print_in_order(tree_heap * raiz);
        void print_pre_order(tree_heap * raiz);
        long long int calcula_altura(tree_heap * raiz);
};
```

Cada nó tem ponteiros para a esquerda e direita e um valor inteiro de prioridade. Ao ser inserido um novo nó é sorteado utilizando a função rand() um inteiro de 0 até 2 * N (Com N sendo o número de palavras indicado na entrada). Esse valor é salvo no parâmetro 'valor_max_prioridade'. A escolha de tal valor foi realizada de forma arbitrária de acordo com os testes. Importante destacar que é possível ocorrer de o nó novo ter prioridade exatamente igual a de seu pai, nesse caso, não ocorre rotação.

Ao inserir um novo nó e ser sorteado a prioridade, é verificado se a propriedade do heap foi abalada, isso é, se esse novo nó tem prioridade menor do que seus ancestrais. Caso tenha maior, ocorrem rotações para corrigir tal problema. Essa correção ocorre de maneira recursiva, após cada chamada recursiva da função de inserção é verificado se o filho (esquerdo ou direito) tem prioridade maior que o nó atual. Se tiver, rotaciona, e tal ação é executada até tudo estar certo.

Não temos o parâmetro altura aqui, pois seria mais complexo calcular o mesmo a cada nova inserção já que ocorrem rotações, por isso temos uma função que calcula a altura da árvore.

A função de busca é análoga à função de busca da ABB.

Estava tendo problemas para os casos em que a rotação chegava recursivamente até a raiz, pois estava mantendo um ponteiro para o pai de cada nó e quando chegava na raiz, que tem pai nullptr, ocorreram alguns problemas de falha de segmentação. Porém, ao excluir o ponteiro para o pai e realizar a rotação mandando diretamente o pai como parâmetro, tudo deu certo. Assim, a função de

rotação ficou análoga à função e rotação da Rubro Negra, explicada a seguir.

Árvore Rubro Negra (ARN):

Para a implementação dessa árvore segui um tutorial explicativo disponível no site do IME:

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-redblack.html

O site utiliza como referência o livro do Sedgewick e Wayne - que é o livro recomendado pelo professor para estudar a matéria.

A estrutura da árvore rubro negra implementada foi a seguinte:

```
typedef struct cel_arn {
   Item val;
   struct cel_arn * esq;
   struct cel_arn * dir;
   char cor;
   Key key;
} arn;

class ARN {
   public:
       arn * arvore;
       long long int n_comp_busca;
       long long int n_comp_insercao;
       long long int n_rotacoes;
       ARN();
       void add(Key key, Item val);
       arn * put(Key key, Item val, arn * raiz);
       bool eh_vermelho(arn * no);
       arn * rotaciona(arn * p, char lado);
       void print_in_order(arn * raiz);
       void print_pre_order(arn * raiz);
       Item value(Key key);
       arn * busca(Key key, arn * raiz);
       long long int calcula_altura(arn * raiz);
};
```

Para cada nó, temos ponteiros para o nó esquerdo e direito, e um char indicando a cor do nó, que pode ser vermelha ou preta. A propriedade de uma árvore rubro negra é que todo caminho da raiz até uma folha nullptr precisa ter o mesmo número de nós pretos e nós vermelhos não podem ter filhos vermelhos. Desta forma, a inserção ocorre sempre em uma folha, que é pintada de vermelho e a correção, caso seja necessária, ocorre de forma recursiva por meio de rotações.

Assim como a TREAP, pela questão das rotações, a altura só é calculada ao final e não durante as inserções. A função de busca é análoga à função de busca da árvore de busca binária.

De função diferente, temos a função que verifica se o nó é vermelho ou nullptr e as funções de print imprimem, juntamente com a chave, a cor do nó.

Com o tutorial não houveram muitos problemas na implementação e foi interessante verificar como os casos diferentes, que foram estudados em aula, foram tratados de maneira mais concisa e genérica. A correção da propriedade foi um tanto análoga à correção realizada na TREAP e ao final, tudo funcionou como esperado.

Árvore 2-3 (A23):

A A23 foi a mais complexa de se fazer pela quantidade de casos diferentes a serem considerados. Implementei ela baseada nas aulas do professor, nas quais ele passou alguns dos códigos e deixou o resto para a gente completar. Apesar da pluralidade de casos diferentes a serem considerados, ao construir o código de forma cuidadosa para cada um deles, tudo acabou dando certo no final.

A estrutura implementada foi a seguinte:

```
typedef struct cel_a23 {
   cel_a23 * p1;
   cel_a23 * p2;
```

```
   cel_a23 * p3;
   Item val1;
   Item val2;
   Key key1;
   Key key2;
   bool eh_2no;
} arv23;

class A23 {
   public:
       arv23 * arvore;
       long long int n_comp_busca;
       long long int n_comp_insercao;
       long long int quebras;
       long long int altura;

       A23();
       void add(Key key, Item val);
       arv23 * put(Key key, Item val, arv23 * raiz, bool &cresceu);
       void print_in_order(arv23 * raiz);
       void print_pre_order(arv23 * raiz);
       Item value(Key key);
       arv23 * busca(Key key, arv23 * raiz);
       bool eh_folha(arv23 * no);
       long long int calcula_altura(arv23 * raiz);
};
```

Para cada nó temos 3 ponteiros, uma variável do tipo bool que indica se o nó é 2-nó ou 3-nó e as duas chaves. Na inserção podem ocorrer quebras de 3-nós, quando isso ocorre, o contador de ‘quebras’ é atualizado, marcando no final da execução da inserção, o número de quebras que ocorreram no total.

Para a função de altura, como todas as folhas ficam na mesma altura na A23, só foi necessário descer recursivamente pelo ponteiro p1 partindo da raiz até chegar em um ponteiro nullptr. Já para a busca, houveram algumas adaptações também, pois se o nó fosse 3-nó, seria necessário comparar com as duas chaves.

Nas função de print, a ‘pre order’ imprime um 2 caso o nó seja um 2 nó, o que permite verificar como ficou a versão final da árvore.

Não houveram muitos problemas à princípio, mas levou um tempo para escrever tudo com calma e verificar que tudo estava certo. Tive alguns problemas com as trocas de chaves dentro de um mesmo nó, quando ocorria uma quebra, mas isso foi resolvido facilmente utilizando variáveis auxiliares para salvar os valores antes de realizar a troca e não deu problemas depois.

Resultados

>> Verificar arquivos com os resultados dos testes feitos.

Estrutura dos Testes:

Os testes forem realizados utilizando os arquivos:

- teste_oc.txt - Um arquivo com 256k palavras ordenadas em ordem crescente;
- teste_od.txt - Um arquivo com 256k palavras ordenadas em ordem decrescente (a maioria distinta, existem algumas com repetição , fato que verifiquei nos testes);
- teste_a.txt - Um arquivo com 256k palavras em ordem aleatória com repetições;
- teste_640k.txt - Um arquivo com um texto lorem com um total de 640 mil palavras com repetições;

Os testes dos parâmetros da estrutura na inserção foram realizados para cada um dos seguintes arquivos: teste_oc.txt, teste_od.txt e teste_a.txt começando em 125 palavras e dobrando esse valor até chegarmos em 64.000 palavras. Para valores maiores que 87.000 palavras distintas (ou seja, testes com os arquivos teste_oc.txt e teste_od.txt) o programa deu falha de segmentação, ocorrida provavelmente pelo cálculo dos parâmetros de número de comparações, trocas, etc.

Para os testes das buscas, foi feito o seguinte:

Com os arquivos teste_oc.txt e teste_od.txt:

- Inseri 87 mil palavras na estrutura;
- Busquei a primeira palavra do arquivo;
- Busquei a última palavra do arquivo;
- Busquei uma palavra menor do que a menor palavra do arquivo;
- Busquei uma palavra maior do que a maior palavra do arquivo;

Com o arquivo teste_a.txt:

- Inseri 256 mil palavras na estrutura;
- Busquei uma palavra presente na estrutura;
- Busquei uma palavra ausente na estrutura;

Conclusão Resultados:

Vetor Dinâmico Ordenado (VO):

Para os testes com as palavras ordenadas de forma crescente, o vetor ordenado foi excepcionalmente bem , já que por inserir sempre ao final, nenhuma posição foi empurrada, ou seja, não houveram trocas. A busca obedeceu um padrão constante nos testes, que era esperado já que é uma busca binária. O pior caso, com o arquivo ordenado de forma decrescente, o número de trocas foi extremamente grande enquanto as comparações mantiveram um padrão.

Árvore de Busca Binária (ABB):

Os piores casos da ABB foram os casos em que a lista de palavras apresentava alguma ordenação, pois a altura da árvore era exatamente o número de palavras inseridas menos um. O número de comparações realizadas nesses casos foi maior do que qualquer outra estrutura, pois em toda inserção, era necessário realizar a comparação com todos os elementos já presentes na estrutura. Dessa forma, na busca para esses casos, o número máximo de comparações realizadas até encontrar a palavra ou dizer que ela não estava na estrutura era igual a altura da árvore. Para os testes com a lista aleatória de palavras, o desempenho foi melhor, um tanto parecido com o vetor ordenado.

Treaps (TR):

Para as Treaps, os resultados não foram muito alterados pela característica de ordenação do arquivo utilizado. Entre as quatro árvores, a Treap manteve a 2º maior altura (após a ABB). As rotações feitas auxiliaram na altura, mas pela questão da prioridade ser um valor aleatório, o balanceamento não era tão preciso, mas já era melhor do que a ABB. Para a busca, que é uma busca binária, o número de comparações realizadas não foi tão grande e ficou próximo das outras que realizavam buscas binárias. Foi melhor do que a ABB, já que a árvore estava melhor balanceada.

Árvores Rubro Negras (ARN):

Para as Rubro-Negras, pela questão das rotações e do balanceamento ser realizado pelas propriedades de cores, a altura ficou melhor do que a altura das Treap. O número de rotações também foi menor e as comparações de inserção e busca, por conta desse fator, ficaram melhores também.

Árvore 2-3 (A23):

A A23 manteve a melhor altura entre todas as árvores, mas sua inserção ‘custa’ mais do que as outras, pela questão de realizar mais comparações, quebras dos nós e trocas de chaves e valores de lugar. Entretanto, como a altura é bem menor, a busca fica melhorada, já que ela é muito bem balanceada. Mas, como dito, esse balanceamento tem um custo alto na inserção. Com 87 mil palavras distintas, a altura ficou 15, enquanto a segunda menor (a da Rubro-Negra) ficou em 24, o que ilustra essa melhora. Entretanto, para buscar na prática, como alguns nós tem 2 chaves, pode-se observar um número maior de comparações em comparação com outras árvores.Porém, o número máximo de comparações é 2 vezes a altura, e como a altura dela é melhor que as outras, para uma quantidade grande de dados, ela é bem melhor na busca.

Comparação:

Em conclusão, para inserir dados ordenados de forma crescente, o melhor é o vetor. Em relação às árvores, a altura é um dos parâmetros mais importantes, assim, como a ABB não apresenta nenhum balanceamento, ela fica refém de como os dados serão inseridos, o que torna ela não muito confiável. Para as outras árvores, o melhor balanceamento é da A23, porém ‘custa’ mais inserir nela. E entre a

ARN e a TR, o balanceamento da ARN é melhor pois depende de algo menos aleatório, que é o caso do sorteio de prioridade da TR.