UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

EDIMAR JACOB BAUER

ÁRVORE BINÁRIA DE PESQUISA OCULTA COM CRESCIMENTO DINÂMICO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2018

EDIMAR JACOB BAUER

ÁRVORE BINÁRIA DE PESQUISA OCULTA COM CRESCIMENTO DINÂMICO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. MSc. Saulo Jorge Beltrão de Queiroz

PONTA GROSSA

2018

RESUMO

BAUER, Edimar. **Árvore Binária de Pesquisa Oculta com Crescimento Dinâmico**. 2018. 69 f. Trabalho de Conclusão de Curso Bacharelado em Ciência da Computação - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

A Árvore Binária de Pesquisa Oculta (do inglês, *Hidden Binary Search Tree*, HBST) é uma estrutura de dados que propõe uma definição alternativa à propriedade de pesquisa das árvores de busca. Na HBST, o caminho de busca é determinado pela média do intervalo de chaves possíveis. Assim, se B representa a mínima quantidade de bits necessários para representar todos os valores chaves, o maior intervalo é $[0, 2^B[$, resultando em uma altura $O(B)$. A HBST não garante elementos em ordem pelo valor chave, logo não se conhece método para realizar a operação de travessia em tempo linear. Este trabalho apresenta a Árvore Binária de Pesquisa Oculta Ordenada (do inglês, *Sorted* HBST, SHBST), que deixa a árvore Oculta em ordem tanto pelo valor oculto quanto pelo valor chave. Apresenta ainda o método de Propagação Estendida que diminui a quantidade máxima de níveis da HBST em uma unidade. E como objetivo principal, o trabalho discute diferentes métodos de crescimento dinâmico da HBST visando uma melhor distribuição dos nós na estrutura e conclui tal discussão apresentando a Árvore Binária de Pesquisa Oculta Dinâmica (DHBST). Por fim, é realizada uma pesquisa empírica de desempenho entre as Árvores AVL, HBST, SHBST e DHBST. Os resultados indicam que as estruturas propostas apresentam a mesma eficiência assintótica de árvores binárias de pesquisa auto balanceadas.

Palavras-chave: Árvore Oculta. HBST. O(B). Propagação. Crescimento Dinâmico.

ABSTRACT

BAUER, Edimar. **Árvore Binária de Pesquisa Oculta com Crescimento Dinâmico**. 2018. 69 p. Work of Conclusion Course in Bachelor of Science in Computer Science - Federal Technology University - Paraná. Ponta Grossa, 2018.

The hidden binary search tree (HBST) is a data structure that proposes an alternative definition for the search property of binary search trees. In the HBST, the search path is determined by the mean value of the keys interval. If B represents the minimum amount of bits to uniquely represent every possible key, the largest interval is $[0.2^B[$, which leads to an $O(B)$ height. However, HSBT does not support linear-time in-order traversal. In this work we present the Sorted HBST (SHBST), a data structure that satisfies not only the hidden search property but also the traditional binary search tree property. This works also presents a procedure (named Enhanced Propagation) to improve the height of HSBT by one unit. Also, the work discusses different methods to enable the dynamic growth of HBST and presents the Dynamic HSBT. All discussed structures were evaluated along with the AVL search tree. The results suggest that all studied structures present the same asymptotic efficiency.

Keywords: Hidden Tree. HBST. O(B). Propagation. Dinamic growth.

LISTA DE ILUSTRAÇÕES

LISTA DE GRÁFICOS

SUMÁRIO

1 INTRODUÇÃO

A Árvore Binária de Pesquisa Oculta, do inglês *Hidden Binary Search Tree* (HBST), é uma estrutura de dados desenvolvida em 2017 pelo professor Saulo Queiroz (2017), que utiliza princípios da divisão e conquista para armazenar dados em forma de árvore. Sua principal característica é o fato de não existir rotações entre seus nós para manter balanceamento e mesmo assim manter uma altura máxima da árvore igual a B, onde B representa a quantidade mínima de bits necessários para representar todos os possíveis valores chaves a serem inseridos. Logo a altura máxima da árvore Oculta é proporcional a quantidade de bits do maior valor chave que pode ser inserido na estrutura. Com isso, a estrutura garante um limite constante de degeneração e ganha em simplicidade de código, por não precisar realizar rotações entre os nós, podendo em algumas circunstâncias realizar as operações de inserção, busca e remoção mais rápido que árvores auto balanceadas como a AVL ou a Red-Black.

A HBST possui altura máxima igual a B, atingindo esta altura quando tiver pelo menos B elementos inseridos, no pior caso. Neste momento, diz-se que a HBST está pseudodesbalanceada, pois a árvore não está balanceada (altura equilibrada entre os ramos) e também não piora mais. Logo, seu desbalanceamento é limitado a B.

Em algumas instâncias de entrada, onde ocorrem os piores casos da árvore Oculta, pode-se aplicar métodos para minimizar os pseudodesbalanceamentos causados por estas instâncias ruins, realizando assim uma melhor distribuição dos nós na árvore, diminuindo a altura média destes nós e consequentemente, melhorando seu tempo de execução para as operações de inserção, busca e remoção.

Este trabalho tem por objetivo principal o estudo de métodos para minimizar os pseudodesbalanceamentos causados na HBST por algumas instâncias de entrada. Serão apresentados três métodos de crescimento dinâmico *in-place* (usando a própria estrutura): Crescimento por Memória, Crescimento por Degrau e Crescimento por Salto. Será apresentado também um método de Crescimento Dinâmico com Estrutura Externa e por fim, definido e apresentado a Árvore Binária de Pesquisa Oculta Dinâmica, do inglês *Dinamic Hidden Binary Search Tree* (DHBST).

Este trabalho também apresenta o método de Propagação. A Propagação permite que a HBST fique ordenada tanto pelo valor Oculto quanto pelo seu valor chave, podendo-se realizar a operação de travessia da HBST agora em $O(n)$. A esta HBST ordenada pelo valor chave dá-se o nome de Árvore Binária de Pesquisa Oculta Ordenada, do inglês *Sorted Hidden Binary Search Tree* (SHBST).

Também é apresentado o método de Propagação Estendida que em algumas situações diminui um nível da altura da HBST. E por fim, é feita uma análise empírica de desempenho entre as principais estruturas abordadas: AVL, HBST, SHBST e DHBST.

Para maior esclarecimento da organização deste trabalho, segue descrição dos próximos Capítulos, sendo que o Capítulo 2 e a Seção 3 são revisões bibliográficas e a partir da Seção 3.1 são contribuições originais deste trabalho.

No Capítulo 2 é realizada uma revisão bibliográfica sobre tópicos e algoritmos pertinentes a este trabalho. No Capítulo 3 é apresentada a Árvore Binária de Pesquisa Oculta. No Capítulo 4 é apresentada a Árvore Binária de Pesquisa Oculta Ordenada. No Capítulo 5 são discutidos diferentes métodos de crescimento dinâmico que realizam uma melhor distribuição dos nós na árvore Oculta para certos tipos de instâncias de entrada. O Capítulo 6 apresenta a Árvore Binária de Pesquisa Oculta Dinâmica. O Capítulo 7 faz uma análise empírica de desempenho entre a AVL, HBST, SHBST e DHBST. E no Capítulo 8, tem-se a conclusão deste trabalho.

2 REVISÃO BIBLIOGRÁFICA

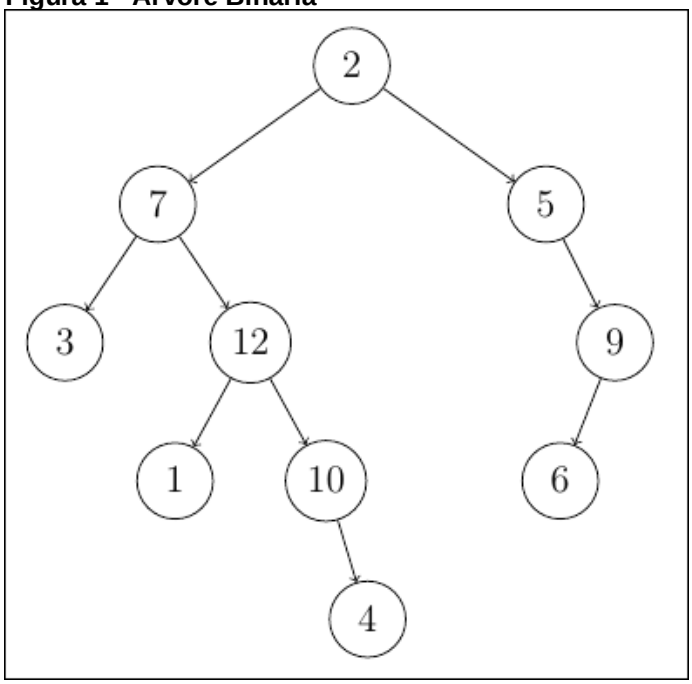
Neste capítulo é feita uma revisão bibliográfica sobre estruturas e algoritmos pertinentes a este trabalho. Serão revisados: tipos de árvores, árvore AVL, tipos de travessia, Heap Binária e Tabela Hash.

2.1 ÁRVORES BINÁRIAS

Árvores fazem parte de um subconjunto de grafos. Todo grafo conexo que não possui ciclo é uma árvore.

Árvores binárias são árvores caracterizadas por possuir 2 ponteiros, um para a subárvore da esquerda e outro para a subárvore da direita, podendo estes serem nulos ou não. Elas são representadas por nós que armazenam dados relevantes ao usuário e pelas ligações entre estes nós, feitas através dos ponteiros. Árvores não possuem ciclo, logo existe um único caminho que parte da raiz até qualquer outro nó. A Figura 1 ilustra o formato de uma árvore binária.

Figura 1 - Árvore Binária



Fonte: Autoria própria (2018)

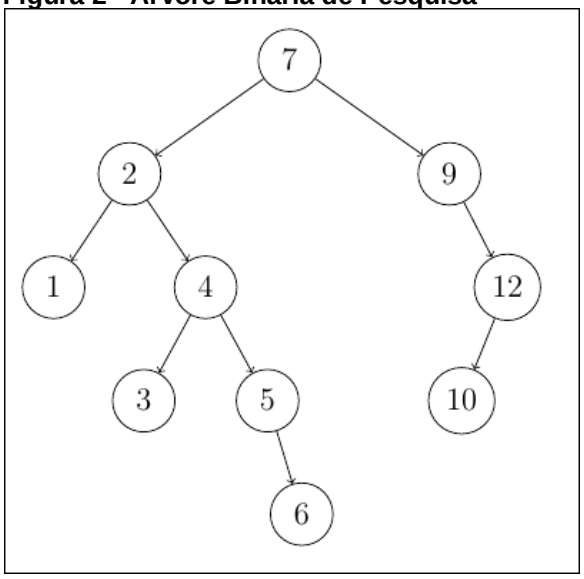
Em estrutura de dados, árvores binárias são caracterizadas também por serem dinâmicas, ou seja, utilizam apenas a quantidade de memória suficiente para

armazenar os dados que a compõem, diferentemente de uma Tabela Hash por exemplo, que utiliza mais memória que o necessário, mas que em geral encontra qualquer chave em $O(1)$ (CORMEN et al, 2009). Portanto, a quantidade de memória gasta em uma árvore binária é proporcional a quantidade de elementos inseridos na estrutura.

2.1.1 Árvores Binárias de Pesquisa

Árvores binárias de pesquisa (ABP) são uma subclasse de árvores binárias. Segundo Chang e Iyangar (1984, p.1), “uma árvore binária de pesquisa é organizada tal que para qualquer nó, todas as chaves na subárvore da esquerda sejam menores e todas as chaves da subárvore da direita sejam maiores que o valor chave deste nó”. Logo, tem-se que, em uma ABP, qualquer elemento na subárvore à esquerda será menor que qualquer outro elemento na subárvore à sua direita. Sendo assim, é possível encontrar qualquer valor chave (x) na árvore a partir do nó raiz verificando se o valor chave do nó atual (key) é igual ao valor procurado x, se não for, verifica-se se $x < key$, seguindo recursivamente para a subárvore da esquerda em caso afirmativo do condicional ou para a subárvore da direita caso contrário, até encontrar o elemento, ou não ter mais nós a seguir, caso que acontece quando o valor x não está inserido na árvore. A Figura 2 ilustra os mesmos elementos da Figura 1 porém aplicado a propriedade de pesquisa pelo valor da chave.

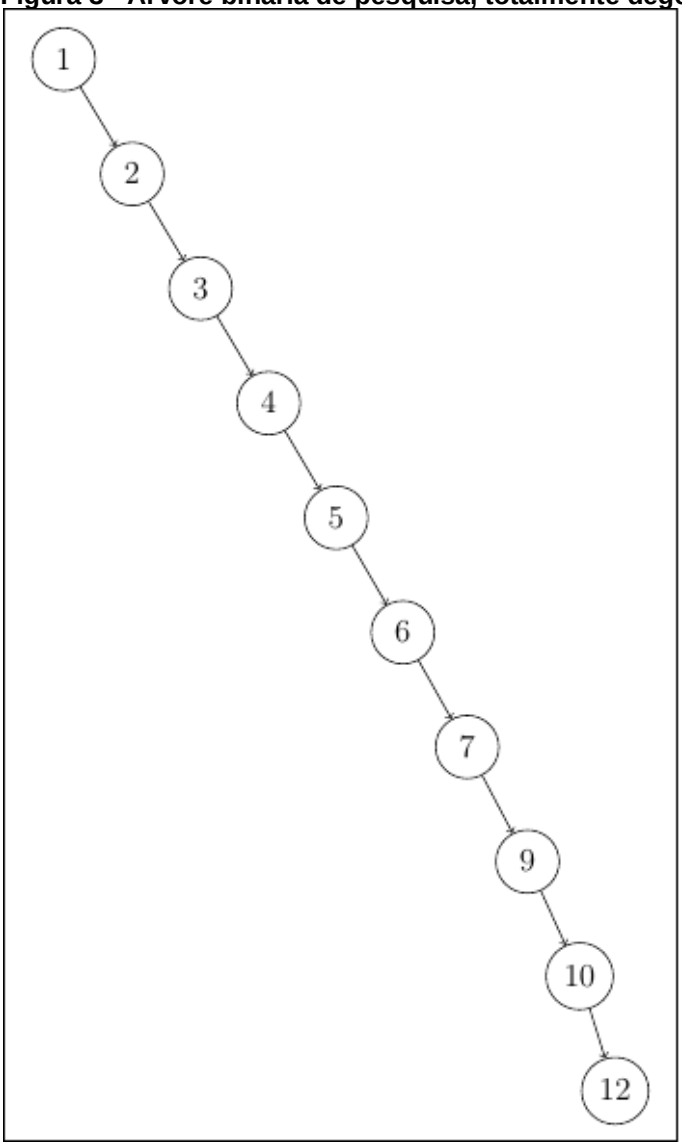
Figura 2 - Árvore Binária de Pesquisa



Fonte: Autoria própria (2018)

Árvores binárias de pesquisa tendem a manter altura logarítmica em função da quantidade de chaves inseridas na árvore (n). Em distribuições uniformes, Knuth (1998) mostra que ABPs requerem apenas $(2 \ln n) \approx (1.386 \log_2 n)$ comparações para realizar qualquer busca na árvore se as chaves forem inseridas em uma ordem aleatória. Logo, elas são ótimas nessas circunstâncias, porém em circunstâncias adversas, elas podem degenerar drasticamente podendo se tornar uma lista encadeada no pior caso, cuja complexidade de inserção, busca e remoção é $O(n)$. Um exemplo de pior caso é mostrado na Figura 3 após inserir em ordem crescente os valores chaves do 1 ao 12 com exceção das chaves de valor 8 e 11. Logo a complexidade de pior caso da inserção, busca e remoção na ABP também é $O(n)$.

Figura 3 - Árvore binária de pesquisa, totalmente degenerada

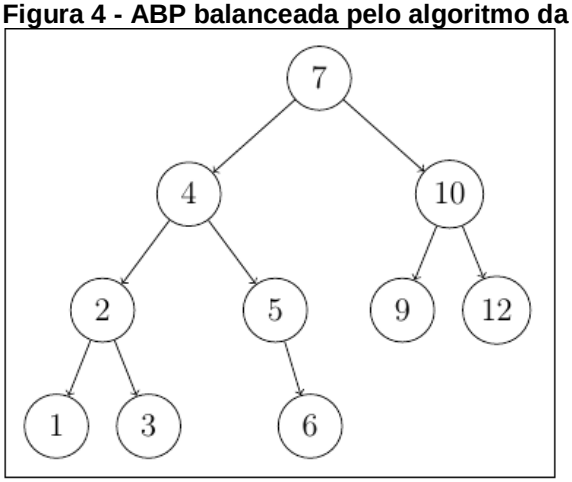


Fonte: Autoria própria (2018)

2.1.1.1Árvores binárias de pesquisa auto balanceadas

Para garantir altura logarítmica para árvores binárias de pesquisa em todos os casos, foram desenvolvidos algoritmos que fazem rotações entre os nós sempre que um ramo estiver consideravelmente mais alto do que outro. Estruturas de Dados como a árvore AVL e a árvore Rubro-Negra são exemplos de estruturas que aplicam essas rotações. Esses algoritmos alteram os nós de posição realizando uma melhor distribuição deles na árvore sem quebrar as propriedades de pesquisa, com isso garantem que a árvore sempre terá altura logarítmica. Como a complexidade das operações de inserção, busca e remoção destas estruturas está diretamente relacionada com a altura da árvore, estes algoritmos possuem, portanto, complexidade $O(\log_2 n)$ (e.g. AVL), (ADELSON-VELSKY; LANDIS, 1962).

Figura 4 - ABP balanceada pelo algoritmo da AVL



Fonte: Autoria própria (2018)

Para realizar essas rotações são necessárias uma série de desvios que verificam e atualizam as posições de cada nó. Logo o código torna-se complexo quando comparado com a ABP. Além disso, é necessário um marcador em cada nó que sinalize quando for necessário rotacionar. Este marcador deve ser atualizado sempre que uma mudança de altura acontecer em alguma das subárvores de um nó, podendo ocorrer no pior caso, $\log_2 n$ iterações para inserir ou remover um elemento mais $\log_2 n$ iterações para atualizar o sinalizador dos nós pais e mais algumas rotações para balancear a árvore, logo tem-se que a complexidade desses algoritmos torna-se $O(2\ \log_2 n + k)$, onde k representa uma constante de operações relacionada às rotações feitas para balancear a árvore. Contudo, pelo princípio da notação assintótica $O(2\ \log_2 n + k)\ = O(\log_2 n)$ (CORMEN et al, 2009). A Figura 4

ilustra os mesmos elementos das Figuras 1, 2 e 3 devidamente balanceada pelo algoritmo da árvore AVL.

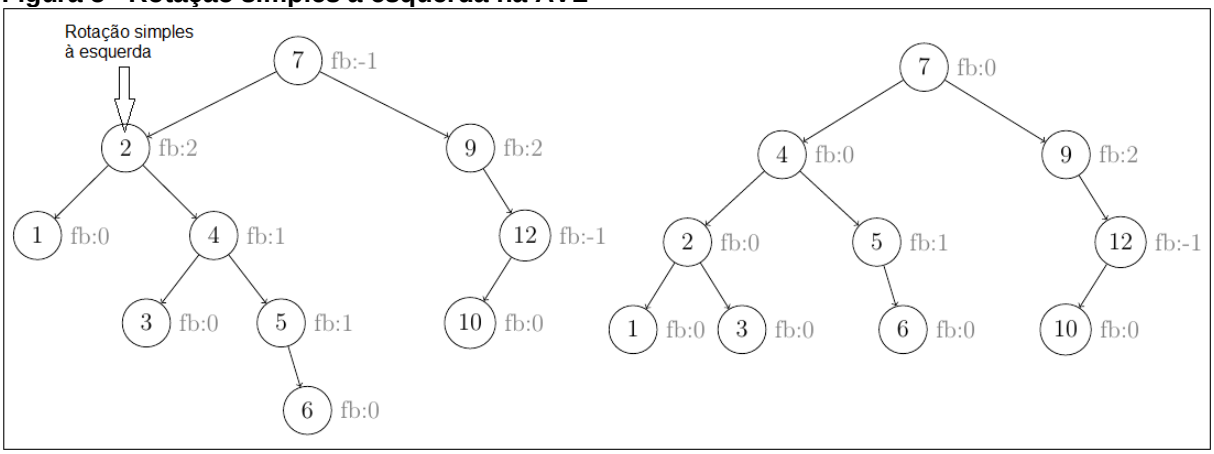
2.2 ÁRVORE AVL

A árvore AVL é uma árvore binária de pesquisa auto balanceada (ADELSON-VELSKY; LANDIS, 1962) que realiza rotação entre as subárvores de um nó sempre que uma subárvore tiver altura h e a outra subárvore tiver altura $h + 2$. Para identificar esta diferença de altura é atribuído a cada nó uma variável chamada fator de balanceamento (fb) que indicará quando for necessário realizar rotações. Todo nó folha possui $fb = 0$. E sempre que a subárvore esquerda de um nó y aumenta um nível, em caso de inserção, diminui-se uma unidade do fb de y, e sempre que a subárvore direita de y aumenta um nível aumenta-se uma unidade do fb de y. Quando alguma subárvore de y diminui um nível, em caso de exclusão de um nó, atualiza-se fb de y também.

Existem 4 tipos de rotação da AVL: rotação simples à esquerda, rotação simples à direita, rotação dupla à esquerda e rotação dupla à direita. Cada vez que um novo elemento é inserido ou removido na AVL, deve-se atualizar o fb de seus antecessores, pois um novo nível foi gerado. Quando $fb = 0$ significa que as duas sub árvores do nó possuem mesma altura. Quando $fb = 1$ significa que a sub árvore da direita possui um nível a mais que a sub árvore da esquerda. Quando $fb = -1$ significa que a sub árvore da esquerda possui um nível a mais que a sub árvore da direita. Quando $fb = 2$ significa que a subárvore da direita possui 2 níveis a mais que a sub árvore da esquerda, neste caso deve-se fazer rotação (simples ou dupla) à esquerda para equilibrar os ramos. Quando $fb = -2$ significa que a sub árvore da esquerda possui 2 níveis a mais que a sub árvore da direita, neste caso deve-se fazer rotação (simples ou dupla) à direita para equilibrar os ramos.

A Figura 5 mostra o funcionamento de uma rotação simples à esquerda. Nesta situação, é realizada rotação sobre o nó com valor chave 2, pois seu $fb = 2$. A subárvore direita da chave 2 passa a ocupar seu lugar, o nó que contém a chave 2 passa a ser a subárvore esquerda do nó de chave 4 e a subárvore direita do nó de chave 2 passa a ser a antiga subárvore esquerda do nó de chave 4.

Figura 5 - Rotação simples à esquerda na AVL



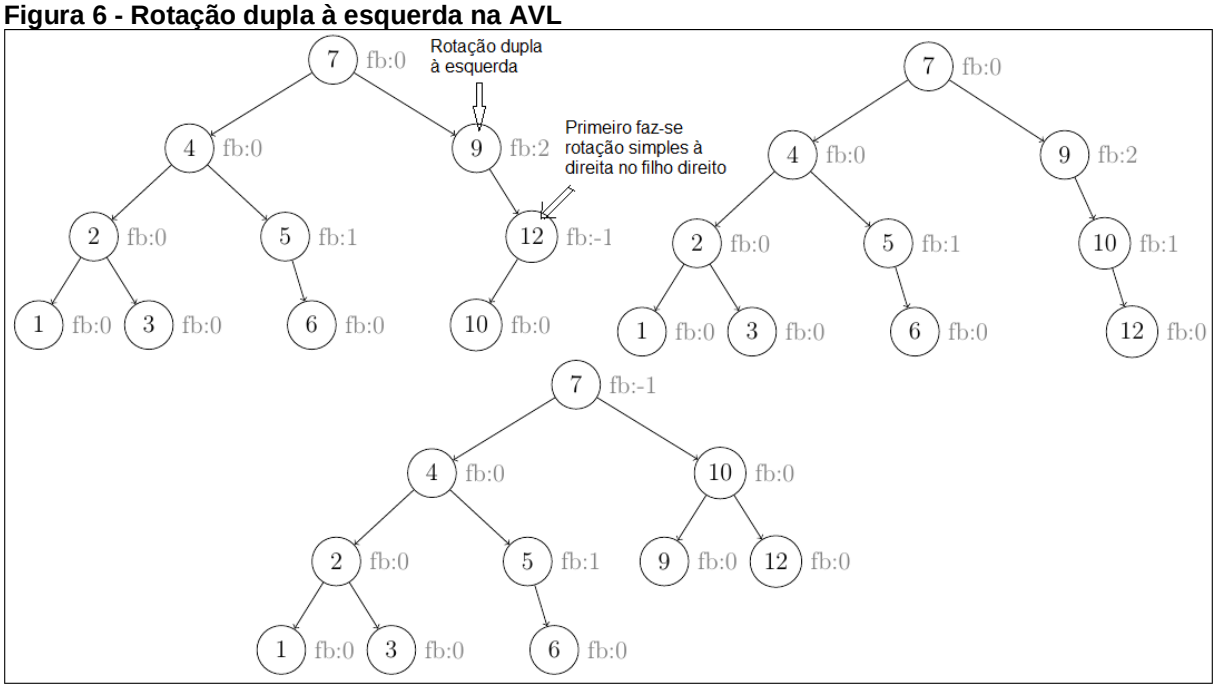
Fonte: Autoria própria (2018)

Uma rotação dupla ocorre quando é necessário fazer uma rotação para uma direção em um nó y e a subárvore de y do lado oposto à rotação, estiver mais pesado na mesma direção da rotação. Diz-se que um ramo está mais pesado do que outro se este ramo tiver um nível a mais do que o outro. Por exemplo, se necessário realizar rotação no nó y à esquerda e a subárvore direita de y estiver mais pesado para a esquerda. Uma rotação dupla à esquerda significa primeiro rotacionar a subárvore direita do nó y com uma rotação simples à direita e depois realizar uma rotação simples à esquerda em y. Uma rotação dupla à direita significa primeiro rotacionar a subárvore esquerda do nó y com uma rotação simples à esquerda e depois realizar uma rotação simples à direita em y.

Na inserção, sempre deve-se atualizar o fb dos antecessores do nó recém inserido. Contudo, quando durante a atualização algum fb tornar-se 0, não será mais necessário continuar atualizando os fb dos demais antecessores, pois quando algum fb torna-se 0 significa que este nó ficou equilibrado e que para seus antecessores nada mudará. Outra situação semelhante ocorre durante a remoção de um nó, quando removido um nó de uma subárvore que possui mesma altura de seu irmão deixando esta subárvore com um nível a menos, atualiza-se o fb do nó pai e pode-se terminar a atualização dos fb dos próximos antecessores, pois a altura destes outros antecessores também nada mudará.

A Figura 6 ilustra três desenhos de ABPs que representam respectivamente, uma ABP desbalanceada no nó de valor 9, a mesma ABP após aplicada uma rotação simples a direita no nó de valor chave 12 e o resultado final desta ABP após aplicado mais uma rotação simples a esquerda no nó de valor chave 9, concretizando uma rotação dupla a esquerda.

Figura 6 - Rotação dupla à esquerda na AVL



Fonte: Autoria própria (2018)

A AVL é uma estrutura de dados que realiza uma excelente distribuição dos elementos, mantendo a altura da maior sub árvore mínima. Sua altura máxima é aproximadamente $1.44\log_2(n+2) - 0.328$, e sua complexidade para as operações de inserção, busca, remoção, sucessor e antecessor é de $O(\log_2 n)$ (ADELSON-VELSKY; LANDIS, 1962)..

2.3 ALGORITMOS DE TRAVESSIA

Algoritmos de travessia ou percurso são utilizados para percorrer todos os elementos de uma estrutura uma única vez. Em geral utilizados para imprimir os valores de cada um dos nós. Esta Seção trata das três formas de realizar a travessia em uma árvore binária de pesquisa, passando por todos os nós e imprimindo seus valores chaves. São elas: pré ordem, em ordem e pós ordem.

No método de travessia pré ordem, imprime-se o valor chave do nó no momento em que se alcança aquele nó. No método de travessia em ordem, imprime-se o valor chave do nó após voltar da recursão da subárvore da esquerda. No método de travessia pós ordem, imprime-se o valor chave do nó após voltar da recursão da subárvore da esquerda e da recursão da subárvore da direita. O método de travessia em ordem, é o método utilizado para fazer a impressão dos valores

chaves em ordem crescente de valores em uma árvore binária de pesquisa tradicional, que mantém valores chaves menores à esquerda e valores chaves maiores à direita.

Como uma ABP possui exatamente duas subárvores, não é necessário laço para percorrer todos os vértices adjacentes de um nó para realizar a travessia, basta seguir recursivamente para a subárvore da esquerda e depois seguir recursivamente para a subárvore da direita. Por se tratar de uma árvore, a estrutura não possui ciclo, portanto também não é necessário um marcador para identificar os nós que já foram passados. Sendo assim, o código torna-se bem simples para qualquer um dos tipos de travessia.

O Algoritmo 1 apresenta a função de travessia programada em linguagem de programação C++ para melhor entendimento.

Algoritmo 1 - Travessia em uma árvore binária de pesquisa

```
//Considere Tree a struct que representa os nós da ABP
void print(Tree* tree){
        if (tree == NULL) return;
        //printf("%d ", tree->key);      //pré ordem
        print(tree->left);
        printf("%d ", tree->key);      //em ordem
        print(tree->right);
        //printf("%d ", tree->key);      //pós ordem
}
```

Fonte: Autoria própria (2018)

Percebe-se pelo Algoritmo 1 que as três formas de travessia se diferenciam unicamente pela posição do comando *printf()* na função. E independente de qual forma for escolhida, cada nó será passado uma única vez pela função, logo a complexidade da travessia é $O(n)$, onde n representa a quantidade de nós da árvore.

2.4 HEAP BINÁRIA

Uma fila de prioridades, em estrutura de dados, é uma fila que mantém sempre o elemento de maior ou menor valor na primeira posição da fila independente da ordem de chegada (CORMEN et al, 2009).

A Heap Binária é uma Árvore Binária que implementa o conceito de fila de prioridades. Logo, esta estrutura é utilizada para tal. A propriedade base para seu funcionamento é sempre manter o valor do nó pai sendo maior ou menor que os dois nós nas suas subárvores, sendo Heap de Mínimo ou Heap de Máximo. A Heap de Mínimo é aquela que mantém sempre o valor do nó pai menor que de suas duas subárvores e a Heap de Máximo faz o oposto. Logo, o nó raiz sempre conterá o menor ou maior valor da árvore. A segunda propriedade mais importante de uma Heap Binária é o fato dela ser completamente balanceada devido aos métodos de inserção e remoção aplicados a ela.

Uma inserção na Heap Binária, sempre acontece na primeira posição vazia da árvore. A primeira posição vazia de uma árvore binária é a posição com o menor nível e mais à esquerda que não esteja ocupada por outro nó. Após ser inserida a chave x na primeira posição vazia da Heap Binária, deve-se ir subindo x na árvore (trocando os valores de x com o valor chave de seu pai) enquanto o nó que contém x tiver pai e este pai possuir valor chave menor (se for Heap de Mínimo) ou maior (se for Heap de Máximo) que x.

Uma remoção sempre remove o valor chave da raiz da Heap Binária e uma outra chave deve preencher a raiz vazia, se existir tal chave. A chave z escolhida é a que está contida na última posição preenchida da Heap Binária. Depois de preencher z na raiz e excluir seu antigo repositório (pois não faz sentido ter dois nós representando o mesmo elemento), deve-se descer z na árvore enquanto existir subárvore não nula abaixo de z e o valor chave desta subárvore for menor (se for Heap de Mínimo) ou maior (se for Heap de Máximo) que z. Se z possuir 2 subárvores válidas, escolhe-se a de menor valor chave, se for Heap de Minimo, para realizar a troca e escolhe-se a de maior valor chave, se for Heap de Máximo.

Tanto as operações de inserção quanto de remoção percorrem no pior caso todos os níveis da Heap Binária realizando trocas de valores. Como a estrutura é plenamente balanceada, pois só insere na primeira posição vazia e exclui o último nó válido, ela mantém sempre altura logarítmica, logo a complexidade das operações de inserção e remoção são $O(\log_2 n)$ (CORMEN et al, 2009). Acessar o elemento do topo, o qual contém o menor valor da árvore, se Heap de Mínimo, ou contém o maior valor da árvore, se Hep de Máximo, é $O(1)$. A operação de busca na Heap Binária não pode ser realizada em tempo logarítmico pois não existe um critério de pesquisa para a busca.

2.5 TABELA HASH

Uma Tabela Hash ou também chamada Tabela de Espalhamento é uma estrutura de dados que associa chaves de pesquisa a valores (CORMEN et al, 2009). Seu objetivo é a partir de um valor chave encontrar rapidamente sua posição de armazenamento. Usa-se uma tabela Hash quando o universo de valores chaves é maior do que o universo de armazenamento, porém sabe-se que não se necessita armazenar mais valores chaves do que o tamanho do universo de armazenamento criado.

Para atribuir uma chave de pesquisa a um valor, é necessário o uso de uma função hash. Esta função irá aplicar cálculos sobre a chave e converte-la em uma posição de armazenamento na tabela. Como existem mais chaves do que posições, algumas chaves são convertidas para a mesma posição gerando colisão. Existem técnicas para tratar colisões e pode-se continuar encontrando qualquer chave na tabela, porém o desempenho diminui quanto maior for o número de colisões. Logo, para uma boa função hash, é necessário que ela distribua as chaves na tabela de forma a minimizar essas colisões e consequentemente, obter a posição de armazenamento da chave em $O(1)$ (CORMEN et al, 2009).

Uma tabela Hash deve ser pré alocada, logo a quantidade de memória gasta para a TABELA é constante, contudo, a quantidade total de memória utilizada para armazenamento dos dados pode ser ainda maior, dependendo da técnica de tratamento de colisão escolhida, como por exemplo, tratamento de colisão com lista encadeada que pode aumentar ainda mais a quantidade de memória total utilizada.

2.6 CONCLUSÃO DA REVISÃO BIBLIOGRÁFICA

A revisão bibliográfica teve por objetivo relembrar o leitor de algoritmos e técnicas pertinentes a este trabalho. A árvore AVL será utilizada em comparação com a árvore Oculta e algumas de suas variantes, pois seus comportamentos e utilidades são muito semelhantes. A operação de travessia, quando aplicada na HBST possui comportamento diferente devido a organização dos valores chaves desta inovadora estrutura, e a Heap Binária será utilizada como auxílio para realizar esta travessia. Por último, a Tabela Hash foi revisada pois ela possui semelhanças

com a árvore Oculta por determinar previamente onde cada valor chave deve ser inserido, porém a Tabela Hash aloca previamente uma quantidade m de memória enquanto que a HBST é dinâmica com relação ao uso de memória.

3 ÁRVORE BINÁRIA DE PESQUISA OCULTA

Queiroz (2017) adotou um novo conceito de pesquisa para ABPs permitindo outros valores de referência que não a chave dos nós. Na Árvore Binária de Pesquisa Oculta, desenvolvida pelo autor, o valor de referência (ref) é dado pela média do intervalo do nó. Este intervalo diz respeito ao menor ($lower$) e ao maior ($upper$) valor chave que podem ser inseridos naquele nó. Por definição, o $upper$ de cada nó não está incluso no intervalo, logo o intervalo de um nó é dado por $[lower, upper[$. E o ref de cada nó é dado pela equação $ref = (lower + upper) \div 2$.

Quando visualizando imagens da HBST, representa-se o $lower$, o ref e o $upper$ de um nó ao seu lado, estando o $lower$ incluso e o $upper$ excluso do intervalo, e o ref sublinhado (e.g. $[lower, \underline{ref}, upper[$). Esta representação aparece a partir da Figura 7 até a Figura 22, com exceção da Figura 10 que apresenta seu limite inferior aberto devido a aplicação do método de Propagação Estendida que será visto na Seção 4.2.

Durante uma operação de inserção, busca ou remoção, calcula-se ref do nó raiz através da média entre $lower$ e $upper$ do raiz, que devem estar armazenados em algum lugar. No caso de uma busca, por exemplo, onde o elemento buscado (x) não está na raiz, para decidir qual subárvore seguir deve-se comparar se $x < ref$, se for segue-se para a subárvore da esquerda e atualiza-se $upper$ com o valor de ref, caso contrário segue-se para a subárvore da direita e atualiza-se $lower$ com o valor de ref. O Algoritmo 2 mostra a função de busca na HBST, implementado em linguagem de programação C++.

Apenas o $lower$ e $upper$ do nó raiz são definidos e salvos na HBST. As duas subárvores da raiz herdam metade do intervalo de seu pai, onde a subárvore esquerda possuirá o intervalo $[lower, ref[$ e a subárvore direita possuirá o intervalo $[ref, upper[$. Esta ideia segue recursivamente para todas as demais subárvores, onde cada nó filho herda metade do intervalo de seu respectivo pai, como pode ser visto na Figura 8. Dessa forma tem-se que a cada iteração diminui-se pela metade o intervalo de valores chaves original. Logo, o $lower, ref$ e $upper$ não precisam ser armazenados em cada nó, com exceção do $lower$ e o $upper$ do nó raiz. E embora não sejam armazenados os limites de cada nó, cada nó possuirá intervalo único e imutável, não podendo este ser alterado, logo uma vez definido o intervalo do nó raiz, este intervalo torna-se constante.

Algoritmo 2 - Busca na HBST em linguagem C++

```
//considere Tree a struct que representa cada nó
//considere Lower a Upper as variáveis globais que guardam os limites do nó raiz
Tree* search(Tree* tree, int x){
        int ref, lower = Lower, upper = Upper;
        while(tree){
                if (tree->key == x){
                        return tree;
                }
                ref = (lower + upper) / 2;
                if (x < ref){
                        tree = tree->left; upper = ref;
                }else{
                        Tree = tree->right; lower = ref;
                }
        }
        return NULL;
}
```

Fonte: Autoria própria (2018)

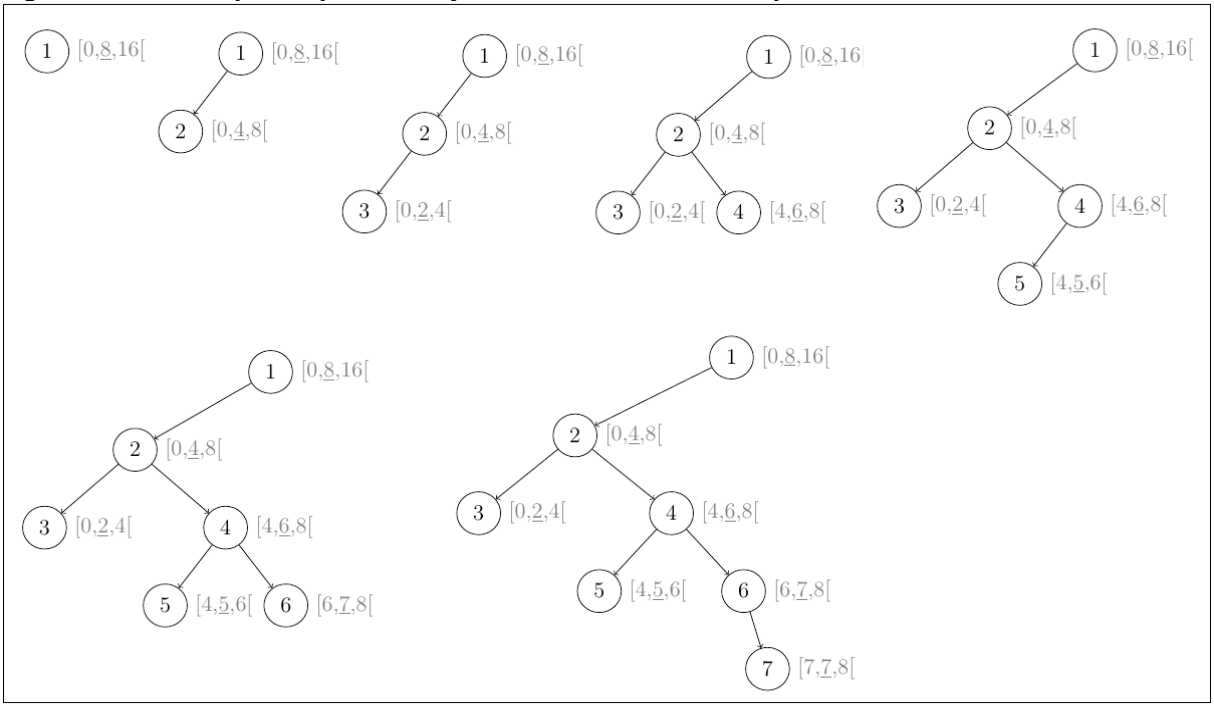
A altura máxima da árvore Oculta é dada por B, onde B representa a mínima quantidade de bits necessários para representar todos os possíveis valores chaves. Como o tempo de execução das operações de inserção, busca e remoção são diretamente proporcionais à altura máxima da HBST, tem-se que a complexidade destas operações é $O(B)$. E a quantidade de elementos que podem ser inseridos na HBST é de 2^B (QUEIROZ; BAUER, 2018).

A árvore Oculta não aplica rotações entre seus nós. Em casos ruins, pode pseudodesbalancear e uma de suas subárvores pode ser vazia enquanto a outra ter altura igual a B, porém a maior subárvore não terá mais do que B níveis.

Para qualquer operação na HBST, pesquisa-se sempre pelo ref do nó e não pelo seu valor chave. Em consequência disso, é possível que ocorra uma inserção na subárvore da esquerda com valor chave maior que seu pai, ou que ocorra uma inserção na sub árvore da direita com valor chave menor que seu pai, mas qualquer subárvore à esquerda sempre terá valor chave menor que qualquer sub árvore à direita, propriedade garantida pelo princípio da divisão do intervalo (metade menor à esquerda, metade maior à direita). Logo a árvore pode não permanecer ordenada entre raiz e subárvore, mas sempre estará ordenada entre subárvores de mesmo pai, da esquerda para a direita.

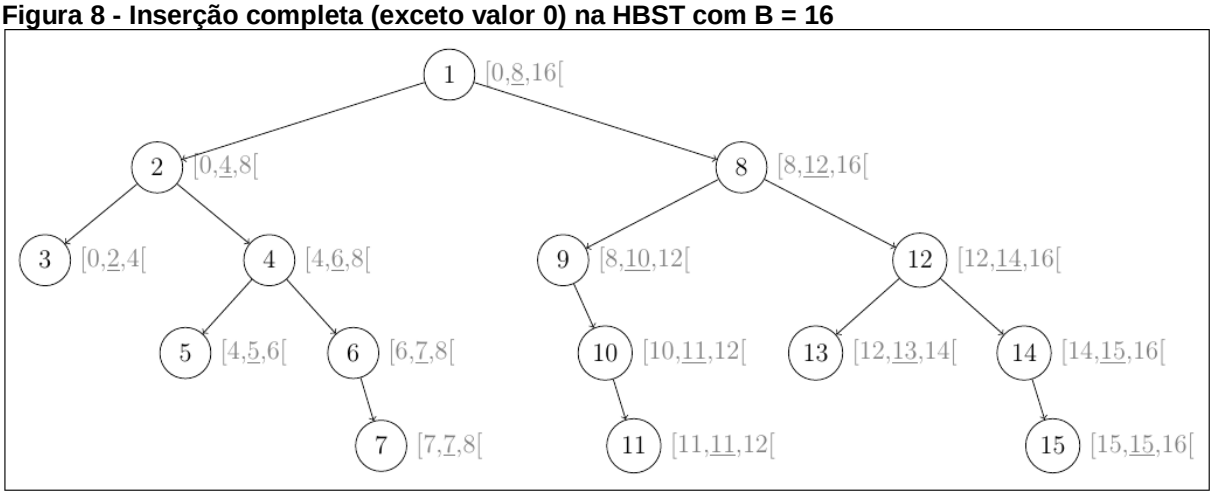
A Figura 7 monstra a inserção passo a passo de valores chaves crescentes a partir de 1 até 7 na HBST, reforçando que para decidir qual subárvore seguir deve-se comparar a chave (x) que está sendo inserida com o ref do nó atual. A Figura 8 mostra a mesma HBST da Figura 7 após aplicada as inserções sequenciais de 8 até 15. Os valores ao lado de cada nó representam respectivamente $lower$, ref e $upper$ de cada nó, sendo que o $lower$ está contido no intervalo e o $upper$ não.

Figura 7 - Passo a passo para inserção crescente na HBST a partir do valor 1



Fonte: Autoria própria (2018)

Figura 8 - Inserção completa (exceto valor 0) na HBST com B = 16



Fonte: Autoria própria (2018)

Como cada nó possui ref único, não é possível fazer a remoção de um nó y que possua apenas uma subárvore não nula conectando esta subárvore ao pai de y

e então excluir y, como é feito na árvore AVL ou na árvore Rubro-Negra, pois poderia haver quebra da propriedade do intervalo nas subárvores abaixo de y. Nesse caso, deve-se fazer a substituição da chave de y por qualquer chave na folha de uma de suas subárvores e então excluir esta subárvore folha. Uma subárvore folha é um nó que aponta para subárvores vazias. Uma subárvore vazia não possui qualquer elemento.

A HBST é ótima para entradas com distribuição uniforme, pois não realiza balanceamento, tampouco necessita controlar o balanceamento, da mesma forma que a ABP, porém diferentemente da ABP, ela garante um limite máximo de degeneração igual a B, considerado pequeno assintoticamente. Exemplificadamente, quando trabalhado com números inteiros (números inteiros em linguagens de programação comumente possuem 32 bits e podem representar mais de 4 bilhões de valores distintos), B será igual a 32, pois um inteiro possui 32 bits, logo, sua altura máxima será de 32, independentemente da quantidade n de elementos inseridos, ao contrário da ABP onde sua altura máxima poderá ser igual a n, neste caso, maior que 4 bilhões.

Em termos de memória utilizada, a HBST é superior a árvore AVL e a árvore Rubro-Negra por não possuir variável de controle de altura, enquanto a árvore AVL possui o fator de balanceamento (fb) e a árvore Rubro-Negra possui um bit sinalizador de cor (CORMEN et al, 2009). Em termos de codificação, seu algoritmo é quase tão simples quanto a ABP, sendo de fácil manutenção. E em termos de tempo de processamento, ela é equivalente a AVL ou a Rubro-Negra, podendo ser mais rápida ou mais demorada, porém em fator constante.

A altura máxima da HBST é B e a quantidade de elementos que a árvore suporta é 2^B. Logo, tanto um intervalo de tamanho 17 quanto um intervalo de tamanho 32 terão altura máxima igual a 5, portanto pode-se definir o tamanho do intervalo do nó raiz sempre em potências de 2, pois a cada novo nível da árvore dobra-se a quantidade de elementos que podem ser inseridos. Dessa forma, o cálculo para encontrar ref sempre se dará em uma divisão inteira e isso torna mais claro a visualização do intervalo de cada nó.

3.1 CÁLCULO ALTERNATIVO DO VALOR OCULTO

O ref é o valor que guia a pesquisa na HBST e seu valor é calculado pela média entre $lower$ e $upper$. Uma maneira alternativa de calcular o ref de cada nó é guardar ao invés de $lower$ e $upper$ da raiz, guardar o ref e uma outra variável que represente o tamanho da metade do intervalo do nó raiz ($universe$). Quando for iterar pela árvore, o ref da próxima subárvore será igual a $ref + (universe \div 2)$, se for seguir para a sub árvore da direita, e será igual a $ref - (universe \div 2)$, se for seguir para a sub árvore da esquerda. E a variável $universe$ da próxima iteração será igual a $universe \div 2$. Esta, é apenas uma outra maneira de iterar pela HBST e suas variantes. Nada muda estruturalmente ou assintoticamente.

3.2 TRAVESSIA

Pelo fato da árvore Oculta não garantir ordenação, não é possível realizar a travessia em $O(n)$. Porém é possível realizá-la em $O(n \log_2 B)$, através do auxílio de uma Heap Binária de Mínimo ($fila$) que vai armazenando os valores dos nós por onde passa enquanto descendo na árvore pelo algoritmo de travessia, e vai extraindo os valores da $fila$ enquanto retorna da recursão, passando por todos os nós da esquerda para a direita. Antes de retornar de cada iteração, pode-se extrair valores da $fila$ enquanto ela não estiver vazia e enquanto o valor a ser extraído for $\leq key$. Logo, o máximo de elementos que a $fila$ conterá será igual ao maior nível da HBST que é dado por B.

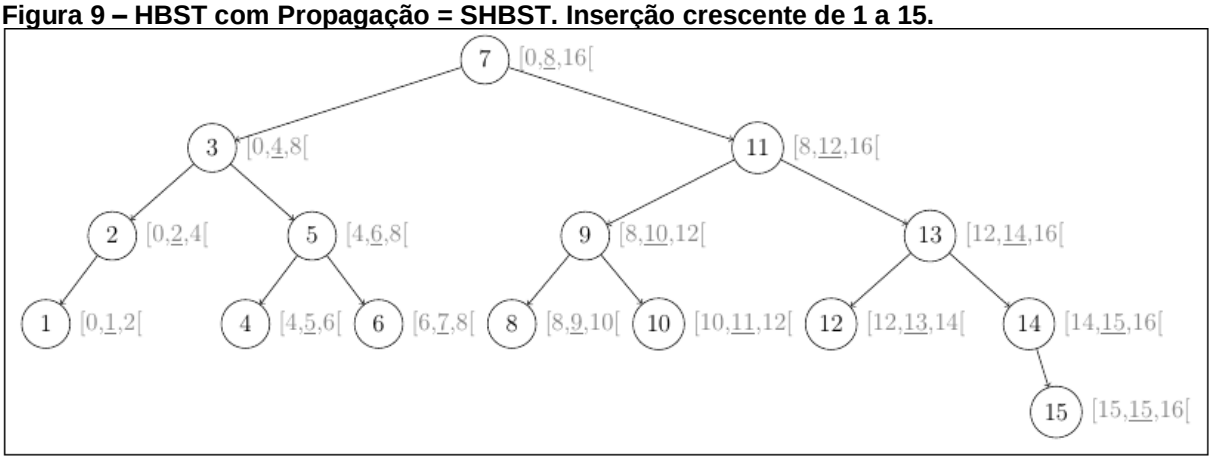
Este método de travessia funciona, pois embora não se saiba sobre o valor key de um nó pai (se ele é maior ou menor que qualquer key de suas subárvores (e por esta razão insere-se valores chaves na $fila$ enquanto descendo em qualquer subárvore)), sabe-se que uma subárvore da esquerda sempre terá key menor que qualquer key da subárvore da direita, logo, pode-se extrair da $fila$ todos os nós da subárvore da esquerda antes de prosseguir para a subárvore da direita. Assim, a $fila$ terá no máximo B elementos inseridos nela, pois B também representa a altura máxima da HBST, e só se insere elementos na $fila$ enquanto descendo na HBST e sempre se retira pelo menos 1 elemento da $fila$ enquanto retorna (exceto quando o referido elemento já foi retirado anteriormente na iteração de alguma de suas

subárvores, situação que ocorre quando o valor chave do nó pai é menor que o valor chave desta subárvore), e como a $fila$ possui complexidade $O(\log_2 n)$ para as operações de inserção e extração, e este $n = B$ no pior caso, tem-se que a complexidade de inserção e de extração na $fila$ é $O(\log_2 B)$ nesta situação. Através do algoritmo de travessia, passa-se por n nós e em cada nó é feita uma inserção e uma extração da $fila$, logo tem-se a complexidade da travessia para a HBST igual a $O(n \times 2 \times \log_2 B) = O(n \log_2 B)$.

4 ÁRVORE BINÁRIA DE PESQUISA OCULTA ORDENADA

É possível realizar uma adaptação na operação de inserção da HBST fazendo com que os valores chaves dela sempre fiquem em ordem, garantindo a travessia em $O(n)$. Essa adaptação é dada através de um desvio condicional, que após identificada a direção da pesquisa, compara x com key e troca os valores caso necessário. Durante uma pesquisa, se $x < ref$, significa que deve-se seguir a esquerda, nesse momento verifica se $x > key$, em caso afirmativo significa que key também pertence ao intervalo da esquerda e que key é menor do que x. Como ambos pertencem ao intervalo da esquerda, pode-se fazer a troca entre x e key para garantir que o filho da esquerda seja menor do que a chave do nó atual. Usa-se a lógica inversa para a direita, garantindo sempre chaves maiores a direita. Realizando estes condicionais para cada subárvore, garante-se a ordem pelo valor chave dos elementos na árvore. Este procedimento recebe o nome de **Propagação**.

Figura 9 – HBST com Propagação = SHBST. Inserção crescente de 1 a 15.



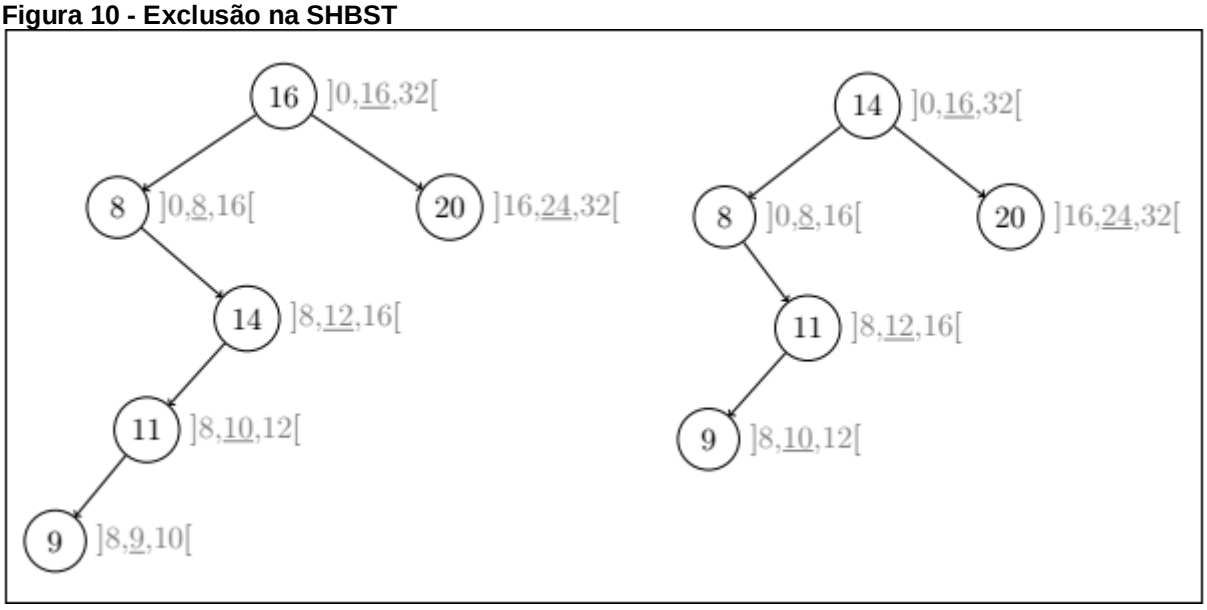
Fonte: Autoria própria (2018)

Nestas circunstâncias, a HBST torna-se ordenada da esquerda para a direita, podendo agora ser realizada a pesquisa através de ref ou de key, como mostra a Figura 9. Esta nova forma da HBST recebe o nome de **SHBST** (*Sorted-Hidden Binary Search Tree*) em português, Árvore Binária de Pesquisa Oculta Ordenada. Sua complexidade permanece a mesma nas operações de busca, inserção e remoção, e passa a ser agora $O(n)$ para a operação de travessia.

4.1 EXCLUSÃO NA SHBST

A SHBST possui um tempo maior para realização da exclusão em termos de constantes, pois sua exclusão é realizada por substituição inclusive quando o nó (y) a ser removido possui uma única subárvore não vazia, da mesma forma que a HBST, porém a HBST pode fazer a substituição por qualquer nó folha que seja subárvore de y, enquanto que a SHBST atua como no modelo de substituição da AVL, selecionando o nó de maior valor chave na subárvore esquerda de y ou selecionando o nó de menor valor chave na subárvore direita de y, para poder manter a ordem pelo valor chave. Porém a SHBST, assim como a HBST, não pode remover um nó não folha sem ser por substituição, pois causaria quebra do intervalo, logo se o nó de maior valor chave na subárvore da esquerda de y ou se o nó de menor valor chave na subárvore direita de y, não forem folhas, estes também devem ser substituídos. No pior caso, podem ocorrer B substituições até chegar ao nó folha, acarretando uma constante maior de desempenho em função do tempo. Contudo a operação de exclusão continua sendo $O(B)$.

Figura 10 - Exclusão na SHBST



Fonte: Autoria própria (2018)

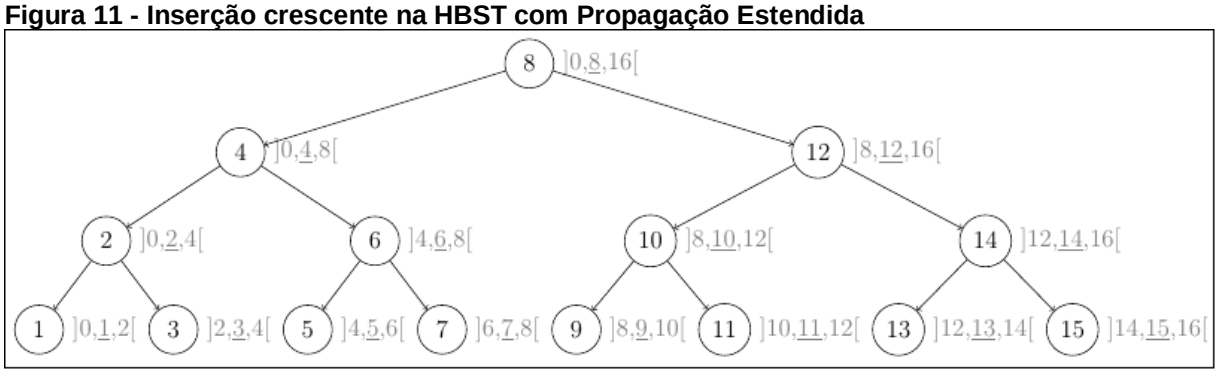
A Figura 10 mostra na imagem à esquerda uma árvore SHBST e na imagem à direita a mesma árvore após ser removido a chave 16. Note que foi selecionado o maior valor chave (14) na subárvore direita do nó de valor chave 16 para ser seu substituto, e como o nó que contém 14 possui subárvore esquerda, este também

deve ser substituído. Logo o 14 foi substituído pelo 11, o 11 foi substituído pelo 9 e então excluído o nó folha de valor chave 9.

4.2 PROPAGAÇÃO ESTENDIDA

A **Propagação Estendida** trata-se de um desvio condicional na inserção de um novo valor x. Quando na inserção, se $x = ref$, deve-se fazer a troca dos valores (x e key) e então continuar a pesquisa normalmente. Esta modificação mantém todo $key = ref$ ou key acima do seu ref na árvore. Em alguns casos, isso pode melhorar a distribuição dos nós acarretando em um nível a menos na altura no pior caso, pois nenhum key ficará abaixo de seu ref. Portanto, a representação do intervalo agora será de colchetes abertos em ambos os lados como mostra a Figura 11.

Figura 11 - Inserção crescente na HBST com Propagação Estendida



Fonte: Autoria própria

No próximo capítulo serão discutidos diferentes métodos de crescimento dinâmico que possuem o objetivo de minimizar a altura média dos nós na HBST.

5 CRESCIMENTO DINÂMICO DA HBST

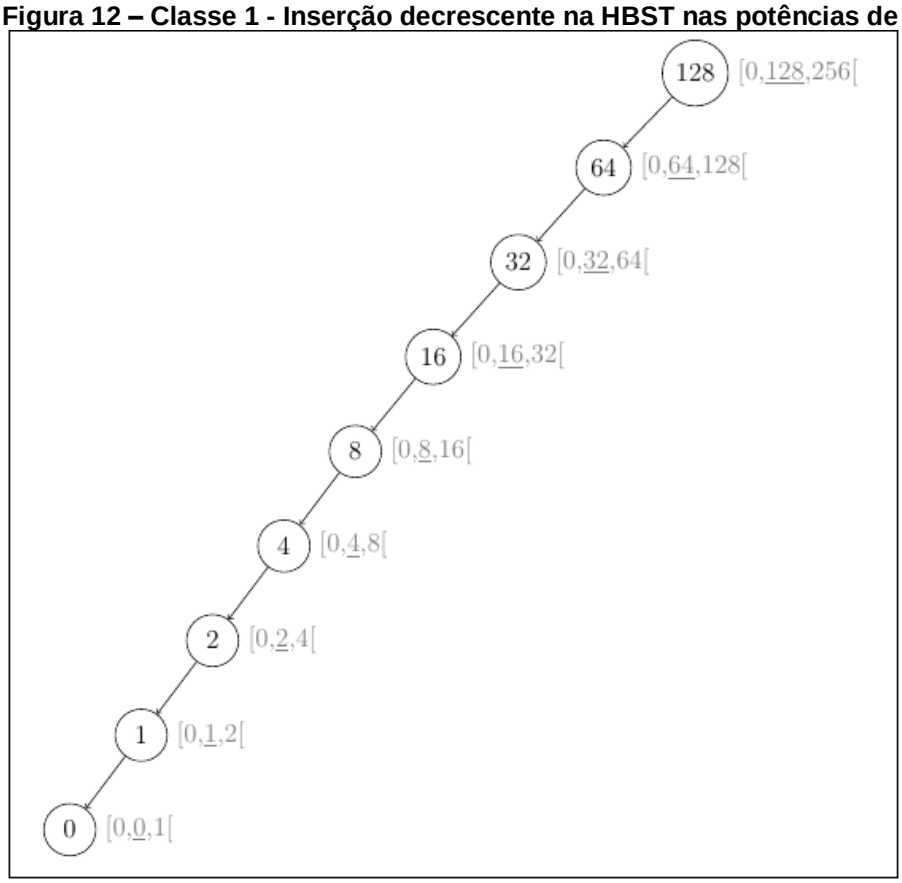
A HBST possui algumas instâncias que podem causar o pseudodesbalanceamento, ou seja, deixar a árvore desbalanceada em função de n, onde n representa a quantidade de nós inseridos na árvore, porém este desbalanceamento não será maior do que B, por esta razão chamado pseudodesbalanceamento.

Para tentar evitar este pseudodesbalanceamento propõem-se uma metodologia de crescimento dinâmico da árvore. Esta metodologia, porém, só melhora algumas instâncias de entrada. Para poder exemplificar quais instâncias são resolvidas pela metodologia de crescimento dinâmico, estas foram divididas em classes.

As instâncias ruins podem ser caracterizadas em duas super classes:

1. Quando a HBST possui chaves de diferentes níveis que seguem o mesmo caminho em profundidade na árvore, elementos em potência de 2 por exemplo, como mostra a Figura 12.

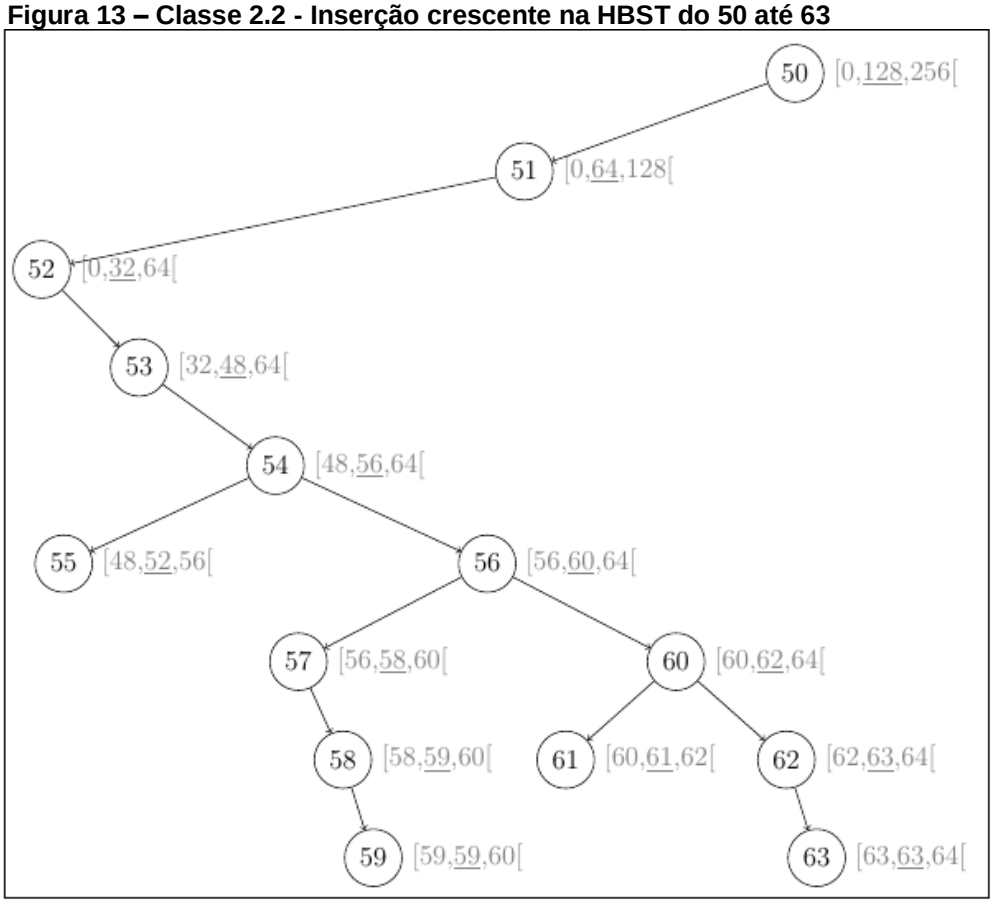
Figura 12 – Classe 1 - Inserção decrescente na HBST nas potências de 2



Fonte: Autoria própria (2018)

2. Quando a entrada compreende um ou mais intervalos específicos, cujos valores das chaves inseridos são muito próximos uns dos outros, como mostra a Figura 13;

Figura 13 – Classe 2.2 - Inserção crescente na HBST do 50 até 63



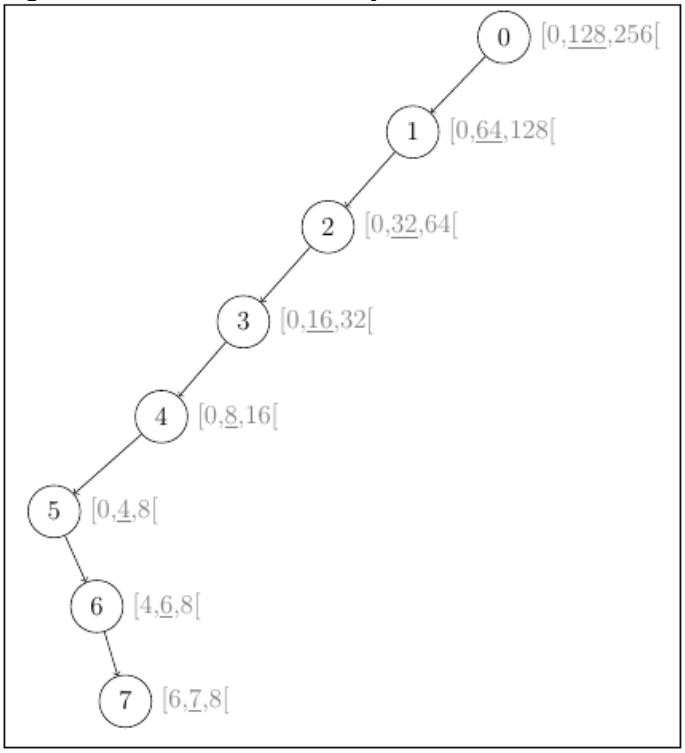
Fonte: Autoria própria (2018)

A classe 2 pode ser subdividida em outras três subclasses:

1. Inserção dos elementos em ordem crescente com diferença de 1 unidade no valor da próxima chave, como mostra a Figura 14;
2. Inserção em um intervalo específico que não seja da classe 2.1, como mostra a Figura 13;
3. Inserção em mais de um intervalo específico.

Crescer dinamicamente significa ter um intervalo mínimo para o nó raiz e ir aumentando este intervalo conforme a necessidade para manter uma melhor distribuição dos nós na árvore. Serão apresentadas em seguida duas formas de crescimento dinâmico, um *in-place* na Seção 5.1 e outro externo na Seção 5.2, que utiliza uma estrutura auxiliar.

Figura 14 – Classe 2.1 - Inserção crescente na HBST de 0 até 7

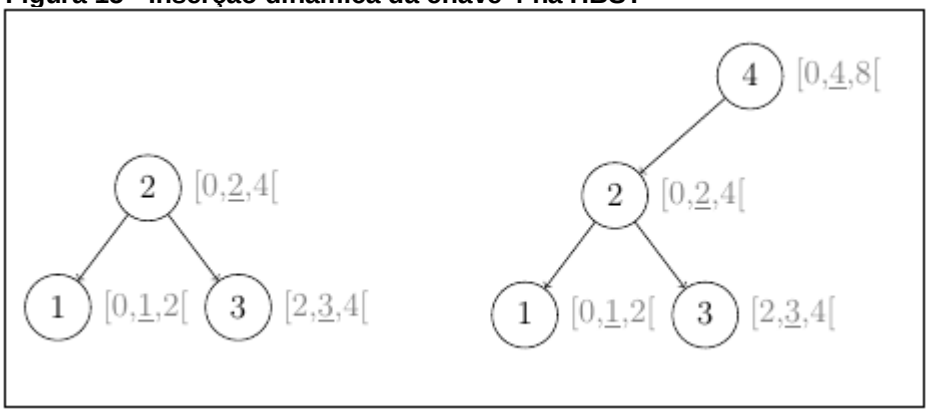


Fonte: Autoria própria (2018)

5.1 CRESCIMENTO DINÂMICO *IN-PLACE*

Como já visto anteriormente no Capítulo 3, não se pode alterar o intervalo de um nó, pois perde-se a propriedade do intervalo da HBST, logo uma vez definido o intervalo do nó raiz não se pode aumentar este intervalo. O que se pode fazer é adicionar um nó pai ao nó raiz, com intervalo igual ao dobro da raiz atual. Este nó pai recém-criado será o novo nó raiz e seu intervalo terá o dobro do tamanho.

Figura 15 - Inserção dinâmica da chave 4 na HBST



Fonte: Autoria própria (2018)

Conforme ilustrado na Figura 15, pode-se criar um nó pai fazendo com que a árvore passe a abranger o dobro do intervalo atual sem alterar as propriedades da HBST. Pode-se perceber também, que é possível aumentar o limite inferior ao invés do limite superior na hora da criação do nó pai, simplesmente adicionando a antiga raiz ao ponteiro direito do nó pai recém-criado.

Para a forma *in-place*, serão abordadas duas formas diferentes de crescimento dinâmico: crescimento dinâmico em apenas um dos limites ($lower$ ou $upper$) e crescimento dinâmico em ambas as direções.

5.1.1 Crescimento Dinâmico Em Uma Direção

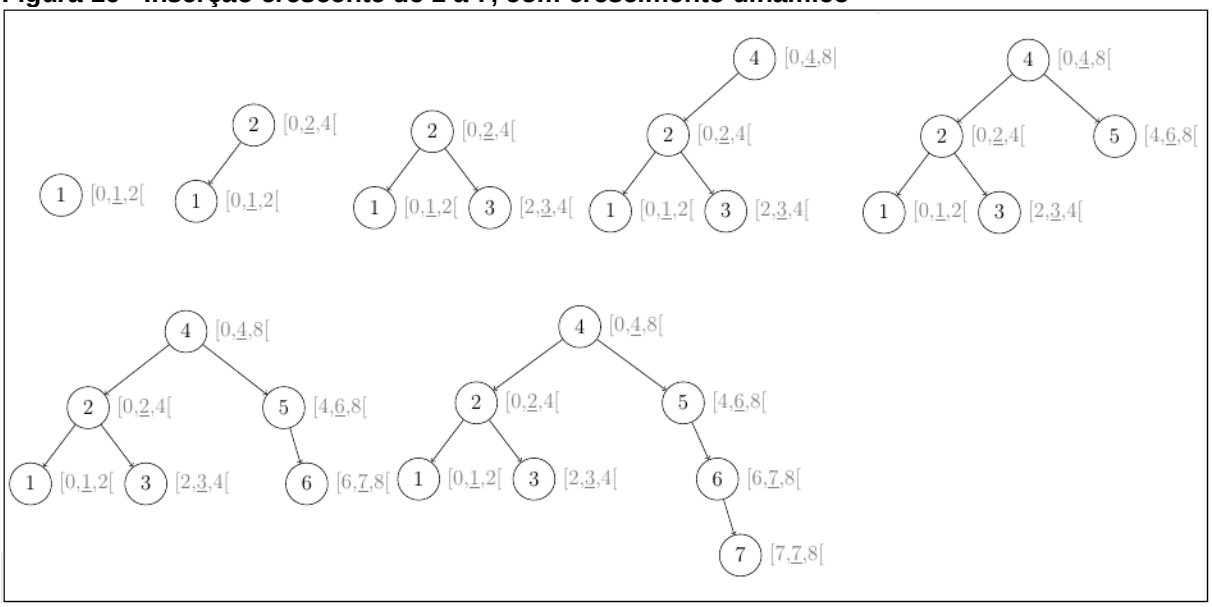
O crescimento dinâmico em apenas um dos limites ou em uma direção, pode ser realizado com o $lower$ fixo e o $upper$ dinâmico ou vice-versa. Para maior esclarecimento, será trabalhado aqui com o $lower$ fixo em zero e o $upper$ dinâmico, porém $lower$ poderia estar fixo em qualquer outro valor. Assim, nosso $lower$ da raiz será sempre zero e o $upper$ da raiz será sempre o menor valor possível de modo que a chave key contida na raiz seja menor que este $upper$. Assim, o $upper$ da raiz será sempre uma potência de 2 que dobra a cada novo nó pai criado.

Perceba que se $lower$ estivesse fixo em outro valor que não 0, poderia ocorrer do upper ser um valor diferente de um valor potência de 2, porém o que permanece em potência de 2 é o tamanho do intervalo ($upper - lower$) e este é que dobra a cada novo nó pai criado. Perceba também que não há restrição alguma quanto ao uso de números negativos, pois o que define a estrutura e a altura máxima da HBST é o tamanho do intervalo do nó raiz, independente de onde comece ou termine seu intervalo.

A classe 2.1 é resolvida com esta abordagem inicial, pois $upper$ da raiz iniciará mínimo, sendo a chave 1 a primeira a ser inserida (considerando iniciar em 1) e a altura máxima da árvore só irá aumentar quando for criado um novo nó pai. Para esta classe, só é acrescentado um novo nó pai depois que todos os valores do intervalo da raiz forem preenchidos. E quando todos os elementos de um intervalo estão inseridos em uma subárvore da HBST, então esta subárvore estará completa, logo, com seus elementos bem distribuídos e com altura máxima igual a quantidade de bits necessária para representar seu $upper - 1$, que é igual a $\log_2 upper$. Como a árvore esta balanceada, a complexidade de inserção e busca dela passa a ser

$O(\log_2 n)$, como pode ser visto na Figura 16, melhorando seu desempenho em função do tempo.

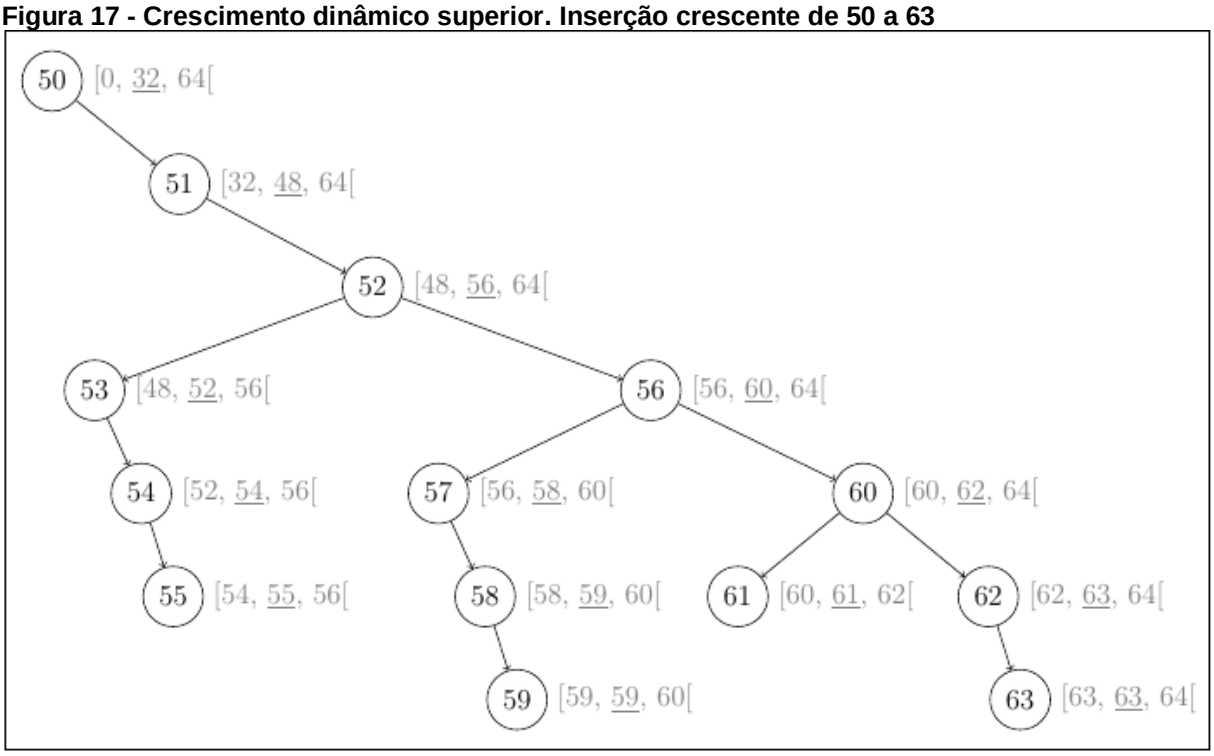
Figura 16 - Inserção crescente de 1 a 7, com crescimento dinâmico



Fonte: Autoria própria (2018)

Para as classes 2.2 e 2.3, resolve-se o problema parcialmente, pois embora esta técnica diminua a altura máxima da HBST, pelo fato de encurtar o limite superior do nó raiz, ela não garante um balanceamento tão bom. Porém, ela melhora o balanceamento da HBST nesta classe, como mostra a Figura 17, onde é possível visualizar como ficaria a HBST com crescimento dinâmico no limite superior após inserir os mesmos elementos que foram inseridos na Figura 13 na mesma ordem de inserção. Observando ainda que a Figura 13 possui intervalo entre 0 e 256, porém se este intervalo fosse maior, o desbalanceamento na Figura 13 também seria maior, enquanto na HBST com crescimento dinâmico o resultado continuaria sendo o mesmo da Figura 17. Este melhor balanceamento, é proporcional ao intervalo de valores inseridos, quanto maior for os valores chaves do intervalo, menor será a eficácia do crescimento dinâmico em uma direção, sendo que, se for utilizado esta técnica para um intervalo cujos valores chaves possuam todos B bits, ela não melhorará em nada a altura média dos nós na árvore.

Figura 17 - Crescimento dinâmico superior. Inserção crescente de 50 a 63



Fonte: Autoria própria (2018)

A HBST no pior caso pode ter $B + 1$ elementos inseridos e ter altura igual a B. Quando estes elementos que causaram o pior caso na HBST pertencem a um intervalo específico, por exemplo, se $B = 31$ e os elementos inseridos compreendem o intervalo de 1 a 31 e forem inseridos em ordem crescente de valor, esta HBST terá 31 níveis. Quando aplicado o crescimento dinâmico mencionado nesta Seção, para esta mesma entrada, adicionado do método de Propagação Estendida, esta mesma HBST terá apenas 5 níveis, pois só cria-se nó pai quando a subárvore da raiz atual estiver cheia e como em 5 níveis é possível armazenar $2^5 - 1 = 31$ elementos, 5 níveis são suficientes para armazenar os 31 elementos. Sendo assim, a aplicação do método de crescimento dinâmico pode reduzir até $B - \lceil \log_2 B \rceil$ níveis da HBST.

As Figuras 14 e 16 mostram um comparativo sem e com a aplicação do crescimento dinâmico na HBST para $B = 8$. Note que o objetivo da Figura 16 não é de fazer este comparativo, porém seu resultado é conveniente. Note que a Figura 16 não implementa Propagação Estendida, mas se aplicasse teria um nível a menos com o rearranjo dos valores chaves 5, 6 e 7 na imagem de última inserção. Note também que a chave de valor 0 não consta na Figura 16 e que a Figura 14 contém 8 níveis totalizando altura 7.

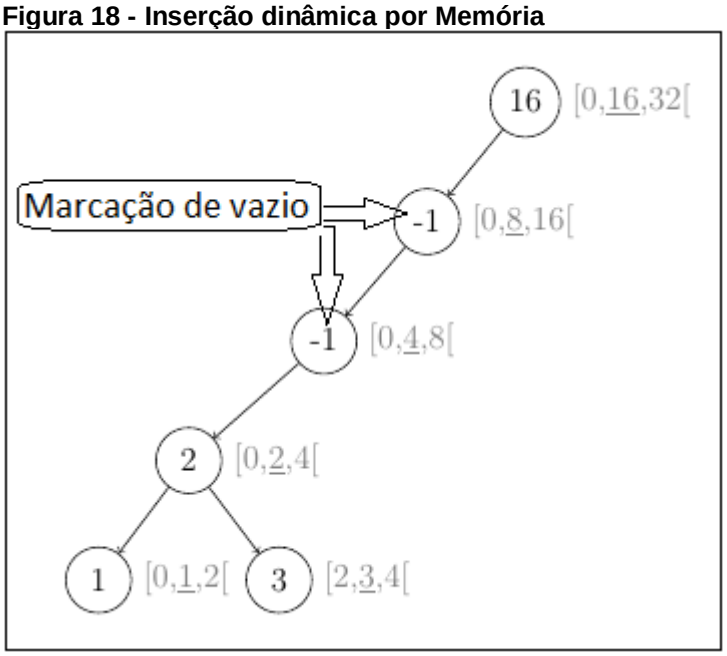
Um dos problemas do crescimento dinâmico surge quando se precisa de um intervalo maior do que apenas o dobro do intervalo atual, pois precisar-se-ia criar mais de um nó pai para atingir o intervalo correspondente dessa nova chave. Dá-se o nome deste problema de **Problema do Salto na HBST**. Lembre-se, a chave a ser inserida precisa estar contida no intervalo do nó. Dessa forma, é preciso utilizar alguma técnica para fazer o crescimento dinâmico mantendo o mínimo de desbalanceamento.

A seguir, serão abordadas três técnicas para resolver o Problema do Salto na HBST: Crescimento Dinâmico por Memória, Crescimento Dinâmico por Salto, e Crescimento Dinâmico por Degrau.

5.1.1.1 Crescimento Dinâmico por Memória

Uma abordagem simples de se pensar é ir criando nós pais um em cima do outro até que o intervalo do último pai criado satisfaça o valor da chave e ela possa ser inserida nele, a essa metodologia dá-se o nome de Crescimento Dinâmico por Memória. Cada um desses nós criados, exceto o último, seriam marcados como vazios, para que pudessem ser preenchidos assim que uma nova inserção correspondente àquele intervalo surgir, como mostra a Figura 18.

Figura 18 - Inserção dinâmica por Memória



Fonte: Autoria própria (2018)

Essa abordagem traz desvantagens, pois utiliza mais memória do que o necessário no momento por criar nós vazios intermediários e utilizar de um marcador para nó vazio, e aumenta o desbalanceamento por criar estes nós intermediários vazios, embora, reforçando, o desbalanceamento seja limitado a B, mais precisamente, o desbalanceamento máximo, quando aplicado em alguma técnica de crescimento dinâmico, é igual ao $\lceil \log_2(upper - lower) \rceil$ da raiz. No caso do Crescimento por Memória, a altura da HBST pode ser inclusive maior que a quantidade de elementos inseridos.

5.1.1.2 Crescimento dinâmico por Salto

Outra alternativa é criar um único nó pai com um intervalo suficientemente grande para cobrir o valor da nova chave a ser inserida (x), a esta metodologia dá-se o nome de Crescimento Dinâmico por Salto. Porém ficarão faltando nós intermediários entre a antiga raiz e a nova. Estes nós faltantes deverão ser criados assim que uma chave correspondente ao seu intervalo aparecer.

Diz-se que uma chave pertence a determinado intervalo se ela for maior ou igual ao ref do nó (quando utilizado crescimento dinâmico no limite superior), desta forma as chaves 2 e 3 pertencem ao intervalo $[0, 2, 4[$, enquanto que as chaves 4, 5, 6 e 7 pertencem ao intervalo $[0, 4, 8[$ (quando $lower$ fixo em 0). Note também que as chaves de um mesmo intervalo utilizam a mesma quantidade de bits para representá-las. Tendo essa informação, para saber se existe um nó intermediário faltando na hora de inserir uma chave x é só verificar a qual intervalo pertence a chave (key) da subárvore da esquerda e a qual intervalo pertence x. Se x pertencer a um intervalo maior que o intervalo de key, significa que existe um nó intermediário faltando, logo cria-se este nó e insere-se x ali mesmo. Note também que só existirão nós intermediários faltantes entre os nós mais à esquerda.

O Algoritmo 3 de inserção, implementado em linguagem de programação C++ trata o Crescimento Dinâmico por Salto no limite superior e com $lower$ fixo em zero.

Algoritmo 3 - Inserção na HBST com Crescimento Dinâmico por Salto em *upper*

```
//Tree é a struct que representa cada nó da árvore
//Upper representa uma variável global que guarda o limite superior da raiz
//se a árvore estiver vazia, Upper será igual a 1
void inserirNode(Tree *&root, int x){
        if (root == NULL || x >= Upper){
                //função makeHead() cria nó pai, conecta a antiga raiz e atualiza Upper
                root = makeHead(root, x);
                return;
        }
        bool right = 0;  //indica que não seguiu nenhuma vez para alguma sub árvore da
direita
        Tree *tree = root; int lower = 0, upper = Upper;
        while(tree->key != x){
                int ref = (lower + upper) / 2;
                if (x < ref){
                        if (tree->left == NULL){
                                tree->left = newNode(x);
                                return;
                        }
                        if (!right){
                                do{
                                        upper = ref; ref = (lower + upper) / 2;
                                while(tree->left->key < ref)
                                if (x >= upper){
                                        criaNoIntermediario(tree, x);
                                        return;
                                }
                        }else{
                                upper = ref;
                        }
                        tree = tree->left;
                }else{
                        if (tree->right == NULL){
                                tree->right = newNode(x);
                                return;
                        }
                        right = 1; tree = tree->right; lower = ref;
                }
        }
}
```

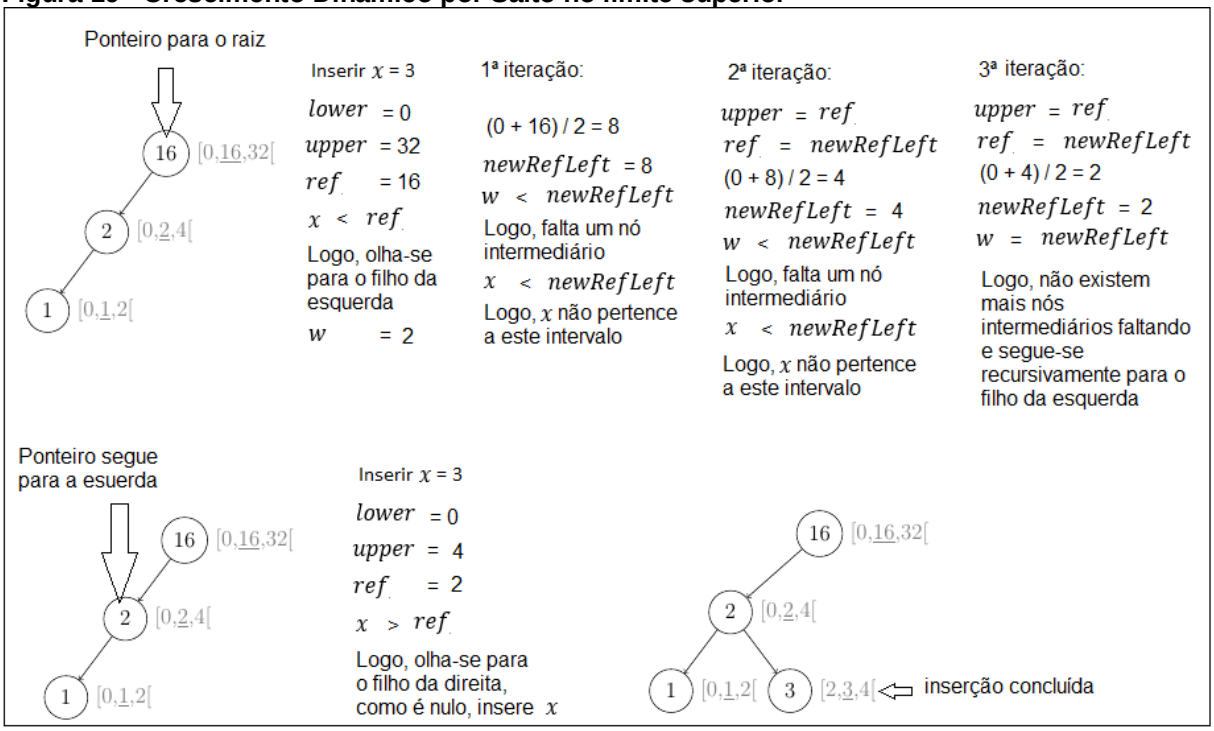
Fonte: Autoria própria (2018)

O cálculo usado para verificar se existe nó faltante na inserção também deve ser usado nos métodos de busca e exclusão para manter as propriedades de intervalo de cada nó, a menos que o método de Propagação esteja aplicado, pois se estiver as chaves estarão em ordem de valor da esquerda para a direita e a pesquisa pode ser feita pelo valor da chave, logo para o método de Crescimento

Dinâmico por Salto, a Propagação é muito conveniente pois evita iterar pelos nós intermediários faltantes nas operações de busca e remoção.

A complexidade não se altera com o Crescimento Dinâmico por Salto, o que muda são as constantes, aumenta-se algumas operações nos métodos de inserção, busca e remoção e diminui-se a altura da árvore em alguns casos. A Figura 19 mostra as iterações ocorridas no processo de inserção da chave 3 para a HBST ilustrada. Com w representando o valor chave da subárvore esquerda.

Figura 19 - Crescimento Dinâmico por Salto no limite superior



Fonte: Autoria própria (2018)

Se durante uma exclusão for detectado que o nó a ser excluído é um nó mais à esquerda da HBST e que este nó possui sub árvore vazia na direita, então pode-se fazer a remoção “tradicional” deste nó e conectar sua sub árvore esquerda ao seu pai. Se este nó excluído for o raiz, então deve-se também atualizar o $upper$ do raiz. Esta é a única situação em que pode ser feita uma exclusão “tradicional” dentre os métodos de crescimento dinâmico apresentados neste Capítulo, pois a metodologia de Crescimento Dinâmico por Salto permite que existam nós intermediários faltantes mais à esquerda e ainda assim conseguir encontrar o ref correspondente do próximo nó. Em todos os outros cenários, deve ser feita a substituição do valor chave do nó a ser excluído pela chave de alguma folha das subárvores e então excluir esta subárvore folha.

5.1.1.3 Crescimento Dinâmico por Degrau

A ideia deste método é fazer o crescimento do intervalo contínuo a partir de um $upper$ inicial. Fazer o crescimento de $upper$ ser o dobro a cada novo nó pai criado, mesmo que o valor chave desta nova raiz seja maior que o intervalo atribuído a ela. Este é o método mais complicado de implementar, pois existem várias situações possíveis.

Neste método poderá haver valores chaves maiores que seu respectivo intervalo, além da raiz. Mas estas irregularidades só poderão ocorrer nos nós mais à esquerda. Este é um método que deve ser utilizado somente se aplicada a técnica de Propagação que deixa a HBST ordenada. Como será usada a Propagação as operações de busca e remoção poderão ser realizadas através da pesquisa convencional, que é pelo valor da chave, logo, o fato de existir valores chaves fora de seus respectivos intervalos não afeta as operações que se baseiam em pesquisa, desde que aplicada a pesquisa pelo valor da chave.

Para que este método funcione é preciso adequar os valores chaves que estão fora de posição assim que possível. Logo, estas chaves irregulares serão substituídas assim que um nó correspondente àquele intervalo vier a ser inserido, e então estas chaves irregulares deverão ser reinseridas a partir da raiz. Além disso, é preciso fazer a **Propagação de Chave** do x recém-inserido no lugar da chave irregular. Fazer a Propagação de Chave deste x significa ir trocando os valores de x e do key da subárvore da esquerda enquanto $x > key$. Mas antes de realizar estas operações (de Propagação de Chave e reinserção), é necessário certificar-se de que x já não está inserido em alguma destas subárvores mais à esquerda.

Quando na inserção, olhando para a raiz, cria-se um nó pai se $x \geq upper$ ou se $key \geq upper$ (considerando ainda a premissa de $lower$ estar fixo em zero). Se criado nó pai, insere-se x e faz-se a Propagação de Chave em x para adequação dos valores.

O método por Degrau faz a reinserção de valores chaves irregulares. No pior caso, pode acontecer de todos os valores chaves inseridos estarem à esquerda e serem irregulares com exceção da raiz. Assim será aplicada a reinserção de todas estas chaves irregulares fazendo com que a complexidade deste método se torne $O(B^2)$, pois seriam $B - 1$ reinserções.

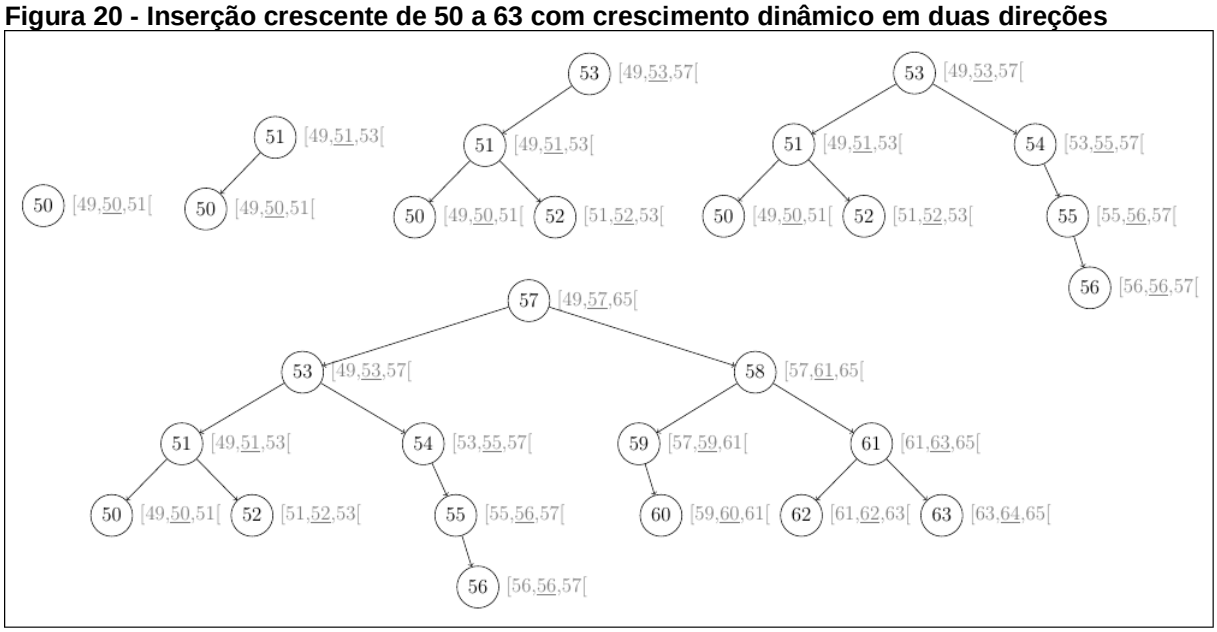
5.1.2 Crescimento Dinâmico Nas Duas Direções

Quando o crescimento dinâmico ocorre nas duas direções, a distribuição dos nós melhora um pouco mais nas instâncias de classe 2.2, pois independentemente do valor chave inserido, o intervalo será adequado para ele o mais justo possível, tanto no limite inferior quanto no limite superior. Logo não se mantém a premissa de que o balanceamento será melhor quanto menor forem os valores chaves de um determinado intervalo, como no caso do crescimento dinâmico em uma direção.

Para crescer dinamicamente em $upper$ basta criar um nó pai a direita do antigo nó raiz e para crescer dinamicamente em $lower$ basta criar um nó pai a esquerda do antigo nó raiz, podendo, neste caso, $lower$ atingir valores negativos. A cada novo nó pai criado, aumenta-se o dobro do intervalo ou mais. Logo, não existem limites fixos e os valores de $lower$ e $upper$ não estarão presos a valores em potência de 2, porém o intervalo de qualquer nó manterá tamanho em potência de 2.

A Figura 20 mostra o resultado deste método aplicado a mesma entrada usada nas Figuras 13 e 17.

Figura 20 - Inserção crescente de 50 a 63 com crescimento dinâmico em duas direções



Fonte: Autoria própria (2018)

Pelo fato de $lower$ e $upper$ não estarem presos aos valores das potências de 2, eles podem atingir seus extremos em diferentes unidades. Em termos de programação de código, pode ocorrer do último $lower$ a ser aumentado ser menor do que o valor da menor chave que pode ser inserida, e o $upper$ ser maior do que o

valor da maior chave que pode ser inserida, logo a HBST poderá ter um nível a mais no pior caso em consequência disso. Observando ainda, em termos de implementação, o cuidado com os extremos do intervalo, para não estourar o limite das variáveis.

Segue a aplicação do crescimento dinâmico nas duas direções implementada nos 3 métodos citados anteriormente.

5.1.2.1 Crescimento Dinâmico por Memória nas duas direções

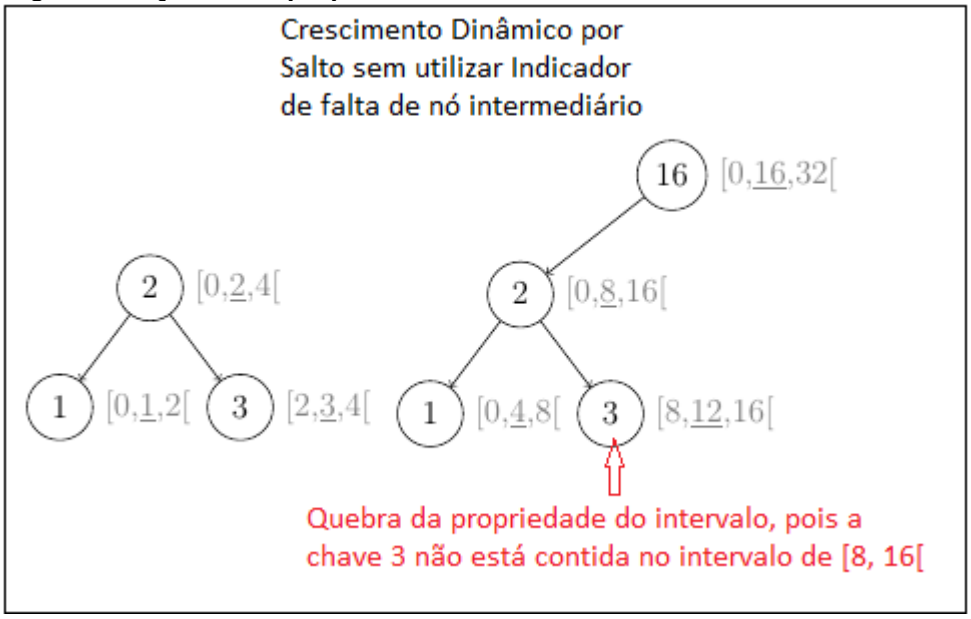
O Crescimento Dinâmico por Memória funciona perfeitamente quando aplicado nas duas direções. Do mesmo modo que foi feito no crescimento em uma direção, aplica-se agora nas duas direções, inserindo nós pais vazios para manter as propriedades de intervalo da HBST até encontrar um intervalo suficiente para compor a chave a ser inserida. Sua complexidade em nada se altera e da mesma forma que no Crescimento Dinâmico por Memória em uma direção, neste método pode acontecer de ter uma altura maior que a quantidade de elementos inseridos.

5.1.2.2 Crescimento Dinâmico por Salto nas duas direções

Quando não há limites fixos, não é possível calcular se existem nós intermediários faltantes olhando-se apenas para o valor chave da subárvore da esquerda ou direita. Para tanto, faz-se necessário um indicador de que existem nós intermediários faltando entre determinados nós.

Lembre-se, o $lower, ref$ e $upper$ de cada nó não são armazenados, com exceção do nó raiz que mantém salvo seu $lower$ e $upper$. Também não se pode alterar o limite do antigo nó raiz e subsequentemente de suas subárvores, pois infringirá, ocasionalmente, a regra do intervalo, causando perda de dados, como mostra a Figura 21. Logo, o indicador torna-se necessário para guardar a quantidade de nós intermediários faltantes. Neste caso, perde-se a vantagem em relação as árvores AVL e Rubro-Negra na economia de memória.

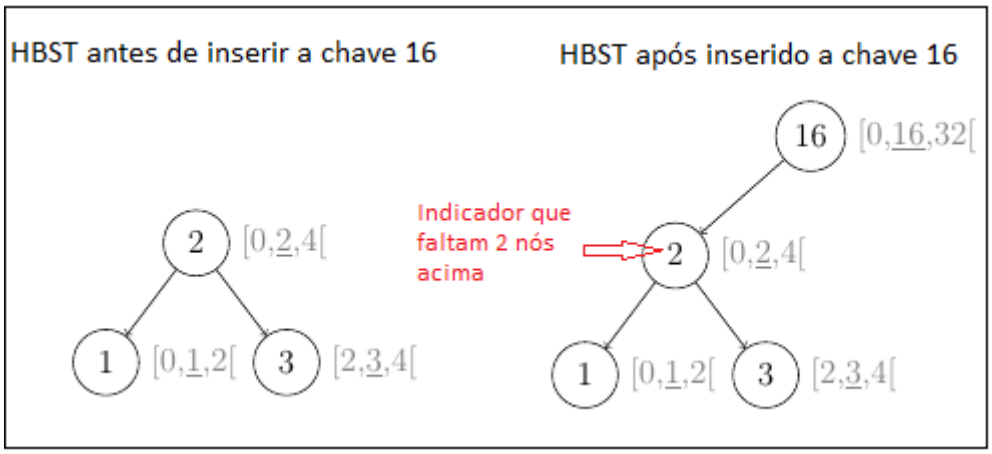
Figura 21 - Quebra da propriedade do intervalo



Fonte: Autoria própria (2018)

A quantidade de nós intermediários faltantes é igual a quantidade de vezes que o intervalo do antigo nó raiz teve que dobrar de tamanho, em $lower$ ou em $upper$, até que x coubesse nesse intervalo resultante, menos 1, correspondente ao intervalo do novo nó raiz.

Figura 22 - Inserção por Salto



Fonte: Autoria própria (2018)

Na criação de um novo nó pai, se existir intervalo faltante, a quantidade de nós faltantes deve ser indicada no antigo nó raiz, como mostra a Figura 22. Quando for inserido um nó intermediário, deve-se manter o devido controle de indicadores atualizados, tanto no nó recém-criado quanto no nó que indicou o intervalo faltante.

Estas adequações mantém o devido controle de nós intermediários faltantes entre dois nós, consequentemente o devido controle sobre a estrutura.

Este método, assim como os outros, não altera a complexidade da HBST, mas melhora a distribuição dos nós na árvore para a classe 2.2, de forma melhor que o crescimento dinâmico em uma única direção, porém sua implementação é mais complexa e com constantes mais caras em função do tempo e de memória.

5.1.2.3 Crescimento Dinâmico por Degrau nas duas direções

Seguindo o modelo deste método para uma direção, quando aplicado o Crescimento Dinâmico por Degrau nas duas direções e for necessário fazer uma Propagação de Chave, esta Propagação de Chave pode percorrer não só o caminho da esquerda ou só o caminho da direita, mas a combinação destes caminhos. Além disso, pode haver nós fora de seus respectivos intervalos no meio da árvore, não sendo este um nó mais à esquerda ou um nó mais à direita, o que torna mais minucioso seu comportamento e sua adequação.

Este tipo de crescimento dinâmico é muito complexo e envolve várias situações possíveis, tal como ter a menor chave (k) da árvore armazenada em um nó intermediário qualquer e este estar fora de seu devido intervalo, necessitando ser adequado assim que um novo valor chave x com valor menor que k ou menor igual que $lower$ do raiz surgir. E assim como no Crescimento por Degrau em uma direção, podem haver mais do que um nó fora de seu devido intervalo, porém aqui, tal situação, ocorre nas duas direções.

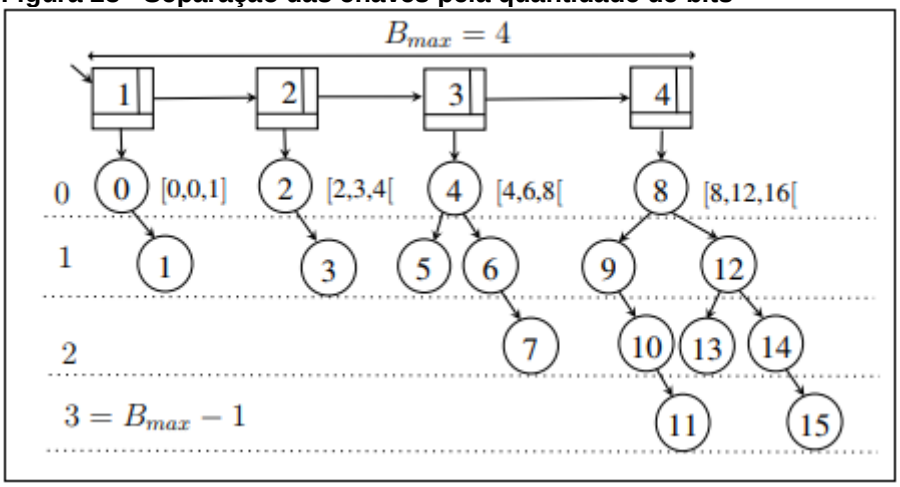
Contudo o algoritmo pode ser implementado. Mas como o código se torna tão grande e confuso quanto o código de uma AVL e seu desempenho em função do tempo na operação de inserção é pior no pior caso, sua utilidade e aplicação se tornam irrelevantes. Assim como no crescimento dinâmico em uma direção, a complexidade assintótica deste método é $O(B^2)$. Por todas estas razões, não serão exemplificados os detalhes deste método.

5.2 CRESCIMENTO DINÂMICO COM ESTRUTURA AUXILIAR

Uma das técnicas de crescimento dinâmico é com o auxílio de uma estrutura auxiliar (lista encadeada por exemplo), que irá separar as chaves pela mínima

quantidade de bits necessário para representá-las. Para cada quantidade de bits mínimos para representar uma chave, cria-se um nó na lista encadeada ($balde$), que possuirá uma HBST independente. A HBST de cada $balde$ irá armazenar apenas as chaves que possuam a mesma quantidade de bits do $balde$. Um exemplo da distribuição dos valores chaves por $balde$ pode ser visto na Figura 22.

Figura 23 - Separação das chaves pela quantidade de bits



Fonte: Queiroz e Bauer, 2018, p.4

Desta forma, serão criados apenas os $balde$'s que possuírem pelo menos uma chave armazenada, obedecendo ao crescimento dinâmico e resolvendo o Problema do Salto na HBST.

Este modelo, assim como o crescimento dinâmico em uma direção, resolve a classe 2.1 e parcialmente as classes 2.2 e 2.3. Na realidade, este método é praticamente idêntico ao do Crescimento Dinâmico por Salto no limite superior e com limite inferior fixo em zero, pois separa os valores chaves pelo seu intervalo correspondente.

Esta técnica foi publicada por Queiroz (Queiroz e Bauer, 2018) e tem como vantagem a capacidade de usar tipos de dados específicos para cada HBST de cada $balde$, utilizando um mínimo de memória para representar cada chave. E traz como desvantagens, a necessidade de utilizar uma estrutura externa como apoio para separar as chaves pela quantidade de bits que as compõem, o que é mais complexo de implementar, e o tempo para encontrar o respectivo $balde$ de cada x. Este tempo para encontrar o respectivo $balde$ é de no máximo $2 \times \log_2 B$ iterações para encontrar o respectivo $balde$, sendo $\log_2 B$ iterações para descobrir a quantidade de bits que compõe uma chave e $\log_2 B$ iterações para encontrar o $balde$

correspondente, mais b iterações para inserir ou buscar ou remover o nó na respectiva HBST, onde b representa a quantidade de bits úteis (mínima quantidade de bits para representar tal valor chave) de x. Em termos de complexidade o método é $O(B)$ para as operações de inserção, busca e remoção.

Vale destacar, que o fato deste tipo de técnica economizar memória não significa ganho exclusivo para a HBST, pois é possível visualizar que a mesma técnica pode ser aplicada a outras estruturas como a AVL, por exemplo, para que a mesma separe os elementos pelo tamanho da chave (tipo de dado) e assim possa fazer a colocação da respectiva chave na árvore AVL que comporte aquele tipo de dado e ter como consequência a diminuição do uso de memória.

5.3 DISCUSSÃO DOS MODELOS DE CRESCIMENTO DINÂMICO

Todos os métodos de crescimento dinâmico vistos resolvem perfeitamente a classe 2.1. Os métodos de Crescimento Dinâmico por Salto nas duas direções e Crescimento Dinâmico por Degrau nas duas direções resolvem bem a classe 2.2, enquanto que os métodos de Crescimento por Salto e Crescimento por Degrau em uma direção e o Crescimento com Estrutura Auxiliar melhoram a distribuição dos nós. O método de Crescimento por Memória pode por vezes ter uma distribuição dos nós pior que a HBST tradicional.

Com exceção do método de Crescimento Dinâmico por Degrau que possui complexidade de inserção no pior caso $O(B^2)$, todos os demais possuem complexidade de inserção igual a $O(B)$, se tornando $O(\log_2 n)$ para entradas da classe 2.1. Em relação à classe 1, nenhum dos métodos apresentados melhora a altura máxima da HBST neste cenário.

O Crescimento Dinâmico por Memória seja em uma direção ou em duas, é o mais simples de implementar, porém como consequência há um gasto de memória maior em comparação com os demais métodos de crescimento dinâmico. E pode ainda possuir uma altura maior do que a quantidade de elementos inseridos, porém esta altura estará limitada a B.

O Crescimento Dinâmico por Salto em apenas uma direção não gasta nada de memória adicional, porém os nós faltantes são iterados mesmo não estando presentes, para poder encontrar o ref correspondente da subárvore esquerda ou direita, logo seu tempo de processamento na inserção é equivalente ao Crescimento

Dinâmico por Memória. Para os métodos de busca e remoção, se aplicado a Propagação, não é mais necessário realizar a pesquisa pelo ref, podendo então realizar-se a pesquisa pelo valor da chave, assim, a constante de tempo deste modelo de crescimento dinâmico para as operações de busca e remoção, melhoram em comparação ao Crescimento Dinâmico por Memória, pois este mantém uma altura média dos nós menor. Este modelo é o único, dentre os mencionados até agora, que permite a exclusão de um nó com uma subárvore vazia sem ser por substituição, porém em apenas alguns casos específicos.

O Crescimento Dinâmico por Salto nas duas direções necessita de um indicador para cada nó da HBST que indique quais possuem nós intermediários faltantes acima deles e a respectiva quantidade de nós faltantes. Este método gasta um pouco a mais de memória por isso, mas é o método que realiza a melhor distribuição dos nós na classe 2.2.

O Crescimento Dinâmico por Degrau em uma direção ou em ambas direções possui a pior complexidade para a inserção sendo $O(B^2)$. É também o mais complexo de implementar.

O Crescimento Dinâmico com Estrutura Auxiliar faz uso de estruturas externas e trabalha com diferentes tipos de dados, tornando sua implementação mais complexa, porém economizando memória.

6 CRESCIMENTO DINÂMICO PLENO, DHBST

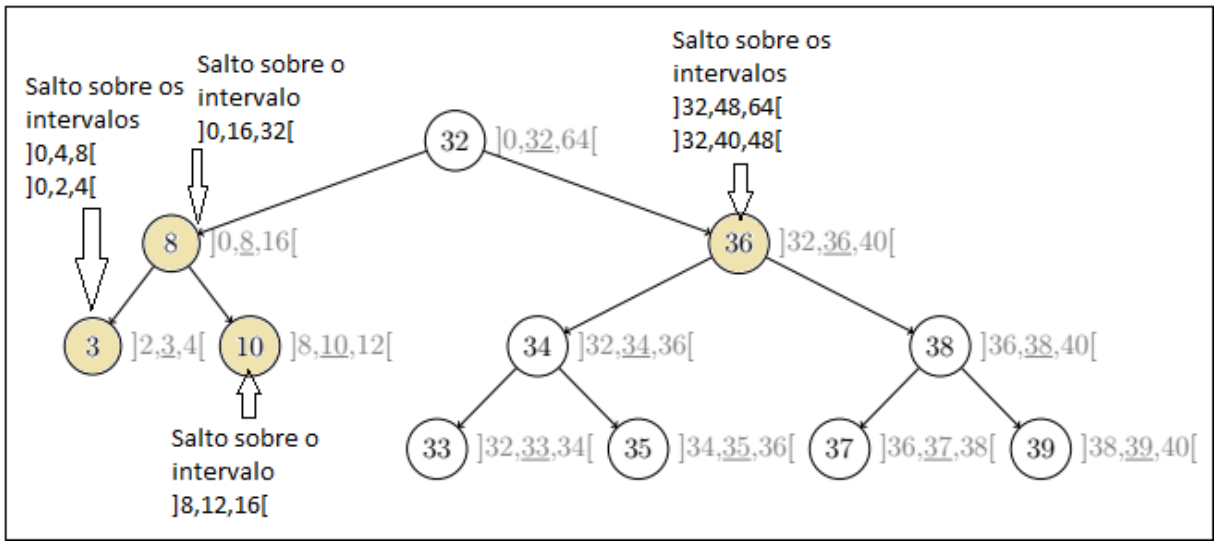
O método de Crescimento Dinâmico por Salto em duas direções permite saltar intervalos quando um novo nó raiz é criado. Para isso, este método mantém salvo em cada nó (y) a quantidade ($miss$) de nós que foram saltados (quantidade de nós intermediários faltantes) para chegar em y. Isso permite diminuir a altura média dos nós em algumas instâncias de entrada, pois alguns nós intermediários não precisam ser criados. Porém este modelo só aplica o salto de intervalo quando criado um novo nó raiz. Se o salto de intervalo fosse aplicado para todos os nós, a distribuição dos nós melhoraria ainda mais, em especial para instâncias de entrada da classe 2.3. Além disso, seria possível inserir nós intermediários em qualquer parte da HBST e também excluir nós que possuam apenas uma subárvore válida conectando esta subárvore válida ao seu nó pai.

Para que isso ocorra, é necessário calcular o menor intervalo possível para todo nó (y) recém-criado e então salvar seu $miss$ e atualizar também o $miss$ da subárvore logo abaixo (quando houver). O menor intervalo possível para y será calculado a partir do intervalo de seu nó pai. Segue-se reduzindo pela metade o intervalo do nó pai até encontrar o intervalo correspondente de y, onde cada vez que se reduz o intervalo do nó pai pela metade e não se determina o intervalo de y corresponde a um nó intermediário faltante.

O menor intervalo possível para um nó folha é aquele cujo ref seja igual ao próprio valor da chave. E o menor intervalo possível para um nó intermediário é o menor intervalo que comporte a própria chave a ser inserida e a chave da subárvore que ficará abaixo do nó recém-criado. Percebe-se que para calcular o intervalo correspondente de um nó, é possível intercalar entre aumentar o valor de $lower$ e diminuir o valor de $upper$ sem perder o devido controle de intervalo. Aumenta-se $lower$ quando x é maior que o ref atualmente calculado, e diminui-se $upper$ quando x é menor que o ref atualmente calculado.

A Figura 24 mostra a diferença de intervalo entre os nós de valores chaves 32 e 36, mostrando que faltam dois nós intermediários entre estes, consequentemente diminuindo a quantidade de níveis da árvore. Também evidência níveis faltantes entre os nós de valor chave 8 e 32, entre 3 e 8, e entre 8 e 10.

Figura 24 - Salto de Intervalo no meio da árvore



Fonte: Autoria própria (2018)

O intervalo de um nó, assim como a quantidade de nós intermediários faltantes acima dele são definidos no momento da inserção do respectivo nó. A cada iteração da operação de inserção deve-se verificar se existe nó intermediário faltante, se sim, deve-se ir iterando o ref na mesma quantidade de nós intermediários faltantes ou parar antes, se detectado que x deve ser inserido como nó intermediário. Uma chave x será inserida como nó intermediário se x estiver fora do intervalo da próxima subárvore a seguir na recursão.

O Algoritmo 4 mostra uma função implementada em linguagem de programação C++ para atualizar o intervalo em cada iteração. Considera-se, neste caso, uma iteração como sendo cada nova subárvore acessada.

Calcular o salto de intervalo para todos os nós da árvore a torna plenamente dinâmica, pois cada nó será inserido com intervalo o mais próximo possível de seu próprio ref, mantendo uma melhor distribuição dos nós na árvore, permitindo inserções no meio da árvore e podendo ser realizada a operação de exclusão de um nó que possua apenas uma subárvore válida conectando essa subárvore ao seu pai, como é feito na AVL, por exemplo. A esta variação da HBST dá-se o nome Árvore Binária de Pesquisa Oculta Dinâmica, em inglês Dinamic Hidden Binary Search Tree (**DHBST**).

A DHBST implementa a Propagação Estendida, sendo esta implementação fundamental para o funcionamento do código, pois o cálculo para determinar o $miss$

de um nó folha se baseia no fato de que ref do nó será igual a sua chave, logo uma chave não pode ficar abaixo de seu respectivo ref.

Algoritmo 4 - Atualização do ref na DHBST

```
//retornar true significa que x foi inserido como nó intermediário
//a sintaxe de & na variável parâmetro da função representa passagem por referência
bool updateRef(Tree *tree, int x, int &lower, int &upper){
        int cont, key, ref = (lower + upper) / 2;
        if (x < ref){       //verifica se está indo para a esquerda ou para a direita e
                            //atribui os devidos valores
                cont = tree->left->miss;
                key = tree->left->key;
        }else{
                cont = tree->right->miss;
                key = tree->right->key;
        }
        do{
                //atualiza o intervalo
                if (x < ref) upper = ref;
                else lower = ref;

                //verifica se ainda existem nós intermediários para continuar atualizando o intervalo
                //se cont for 0 não existem mais nós intermediários faltantes para atualizar o intervalo
                if (cont-- == 0) {
                        return false;
                }
                ref = (lower + upper) / 2;
        }while((x < ref && key < ref) || (x > ref && key > ref));
        //enquanto as variáveis x e key estiverem do mesmo lado de ref
        ///(ambas maiores que ref ou ambas menores que ref),
        //significa que estarão no mesmo intervalo na próxima iteração, logo continue

        createMiddleNode(tree, x, cont);  //esta função também atualiza o miss de cada nó envolvido
        return true;
}
```

Fonte: Autoria própria (2018)

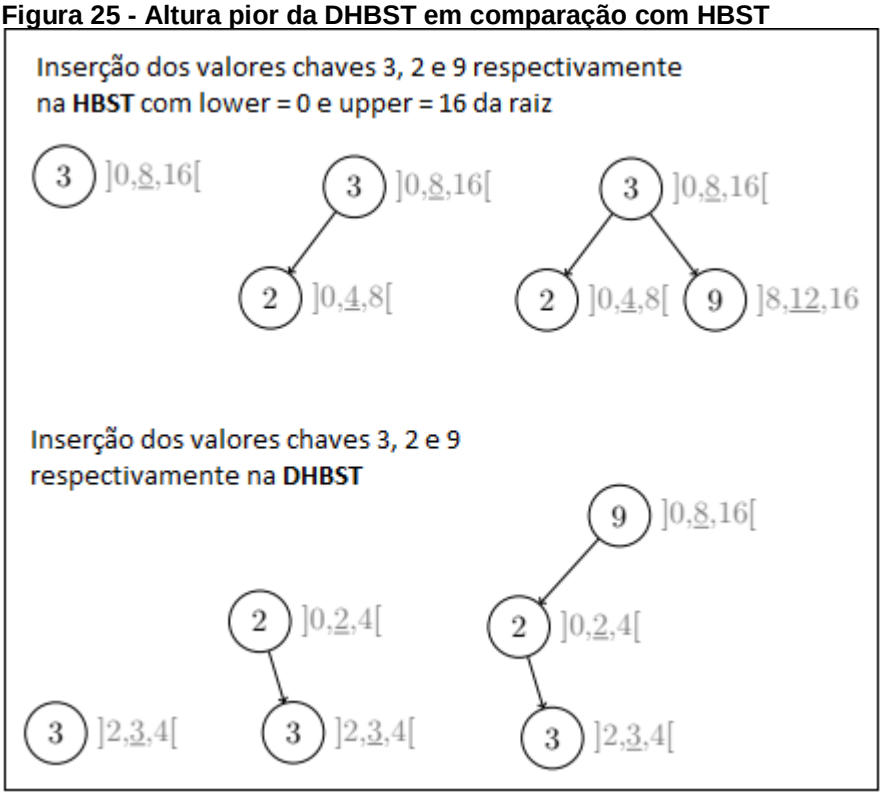
6.1 ALTURA DOS NÓS NA DHBST

A DHBST elimina todos os intervalos entre dois nós y e z, sendo y o nó pai de z, que não precisam existir para manter a propriedade do intervalo (todos os valores chaves inseridos em um nó ou em qualquer de suas subárvores devem estar dentro do intervalo deste nó). Eliminar um intervalo significa eliminar um nível de uma subárvore. Porém o elemento que seria inserido no nível eliminado, será inserido em algum outro nível mais abaixo, podendo esta inserção acarretar ou não

um nível a mais para o nó y. Logo a DHBST sempre terá uma altura média dos nós menor ou igual a HBST.

Porém existe uma exceção, quando na escolha do valor chave de um nó pai. Quando um nó pai é criado, seja criado como nó pai da raiz, vindo a se tornar o novo nó raiz, ou seja na criação de um nó intermediário, este nó pai (y) recém-criado terá um único filho (apenas uma subárvore não vazia) (z) que já estava inserida previamente na árvore. Se y foi criado é porque seu valor chave não entra no intervalo de z, logo cria-se nó pai. Porém se o valor chave que for inserido em y for diferente do ref de y, significa que este valor chave inserido em y pertence a subárvore vazia de y, vindo a ser realocado para esta subárvore posteriormente, quando o condicional da Propagação Estendida for satisfeito em y.

Figura 25 - Altura pior da DHBST em comparação com HBST



Fonte: Autoria própria(2018)

Logo qualquer valor chave, seja da subárvore esquerda ou seja da subárvore direita de y, pode ser inserido em y, sendo que se o valor chave inserido em y vier da subárvore esquerda, então a subárvore esquerda terá um nó a menos, se o valor chave vier da subárvore direita, então a subárvore direita terá um nó a menos. Portanto, seria vantagem que fosse escolhido o valor chave da subárvore com mais elementos para ocupar o valor chave de y para manter um melhor

equilíbrio entre os ramos. Porém na DHBST, sempre é escolhido o valor chave da subárvore vazia para preencher y, pois este valor chave é que está sendo inserido no momento. Enquanto que na HBST, por ela usar uma metodologia de inserção diferente, esta escolha acaba, por vezes, ocasionalmente, sendo a "correta" (inserindo em y o valor chave correspondente ao intervalo da subárvore mais pesada (com maior quantidade de elementos inseridos)), acarretando uma melhor distribuição dos nós, como pode ser visto na Figura 25.

Para resolver este problema, poder-se-ia fazer um método para que sempre que um novo nó pai fosse criado, fosse feito uma busca em z para encontrar o valor chave adequado (valor chave mais próximo do ref de y, para poder manter a propriedade de ordem pela chave) para inserir em y, então inserir o valor chave que seria inserido em y na subárvore vazia de y, melhorando a distribuição dos nós no momento. Porém este método não teria um custo-benefício claro, parecendo ser até pior. Lembrando que se está discutindo tempo de inserção em função de constantes, a complexidade de inserção continuaria sendo $O(B)$ independente se aplicado tal método ou não. Além disso, mesmo se aplicado tal método, depois de algumas inserções, o ramo mais pesado poderia se inverter e passar a ser o mais leve, e então o valor chave de y estaria "errado" novamente. Portanto, tal método não é aplicado na DHBST e existirá casos em que a DHBST possuirá um nível a mais que a HBST ou a SHBST em consequência desta exceção, porém limitado ao máximo de um nível a mais (situação semelhante ao que acontece com a Propagação Estendida), enquanto que por outro lado, poderá ter até $B - \lceil \log_2 B \rceil$ níveis a menos para qualquer subárvore da DHBST, dependendo sempre das instâncias de entrada.

Ou seja, quanto maior a quantidade de bits utilizada pela estrutura, maior será o potencial de ganho da DHBST, pois a DHBST pode transformar uma altura B em $\lceil \log_2 B \rceil$ no melhor caso. Porém, para dados do tipo inteiro, a DHBST não terá uma eficácia tão expressiva, pois o tipo *int* possui apenas 32 bits (na maioria das arquiteturas), logo a DHBST mesmo que diminua alguns níveis para este tipo de dado, sua diferença não será tão perceptível. Porém, para tipos de dados que utilizam muitos bits como valor chave, como *string,* por exemplo, a diferença usando DHBST em vez de SHBST, pode ser bem expressiva.

Sua complexidade é a mesma da SHBST em todas as principais operações: inserção, remoção, busca e travessia. E dentre os métodos de crescimento dinâmico

apresentados, esta é a que realiza a melhor distribuição dos nós, sendo ótima para todas as instâncias da classe 2.

Para a classe 1, nenhum dos métodos propostos neste trabalho obtém qualquer ganho.

7 ANÁLISE DE DESEMPENHO

Neste Capítulo é feita uma análise empírica de desempenho entre as árvores:

- AVL
- HBST
- SHBST
- DHBST

Serão realizados dois tipos de análise: 1) valores chaves gerados aleatoriamente pela função *rand()* da linguagem de programação C++. 2) valores chaves controlados.

Como a função *rand()* do C++ repete a mesma aleatoriedade cada vez que reinicia, todas as inserções e remoções analisadas ocorrem pelos mesmos valores chaves e na mesma ordem de entrada.

Foram analisadas 5 quantidades diferentes de elementos inseridos em cada árvore: 2^{20}, 2^{21}, 2^{22}, 2^{23} e 2^{24} elementos distintos inseridos. Todos os valores chaves gerados compreendem o intervalo de 0 a $2^{30} - 1$.

As análises foram realizadas em um notebook Lenovo, de processador intel core i3 e memória RAM de 4 GB. Não foi feito uso de nenhum software específico para testes ou realizado qualquer tipo de procedimento para dedicação exclusiva de CPU para os algoritmos testados, logo as análises possuem uma margem de erro.

7.1 VALORES ALEATÓRIOS

Para esta análise foram avaliadas as operações de inserção e remoção para as quatro árvores apresentadas e a operação de travessia entre HBST e SHBST somente, pois a travessia para qualquer uma das árvores que não a HBST possui mesmo tempo de execução, já que todas passam uma única vez por cada elemento. Também é feita a análise de altura média dos nós para todas as quatro árvores.

Os resultados são apresentados nas Seções a seguir.

7.1.1 Inserção

A operação de inserção foi a operação mais discutida neste trabalho. A AVL é a árvore auto balanceada que realiza rotações entre seus nós através da variável de controle fator de balanceamento (fb) que é atualizado nos ancestrais do nó recém inserido e realizado rotação nestes ancestrais caso necessário.

A HBST possui um algoritmo de inserção tão simples quanto uma árvore binária de pesquisa comum, necessitando realizar a mais, apenas um cálculo de média e a atualização de um dos limites a cada iteração.

A SHBST possui dois condicionais de desvio a mais sobre o algoritmo da HBST o qual, realiza a troca entre o valor chave da iteração atual e o valor chave que está sendo inserido, sendo um destes condicionais representando a Propagação, para manter a propriedade de ordem pelo valor chave e o outro condicional a Propagação Estendida que pode melhorar a distribuição dos nós em um nível.

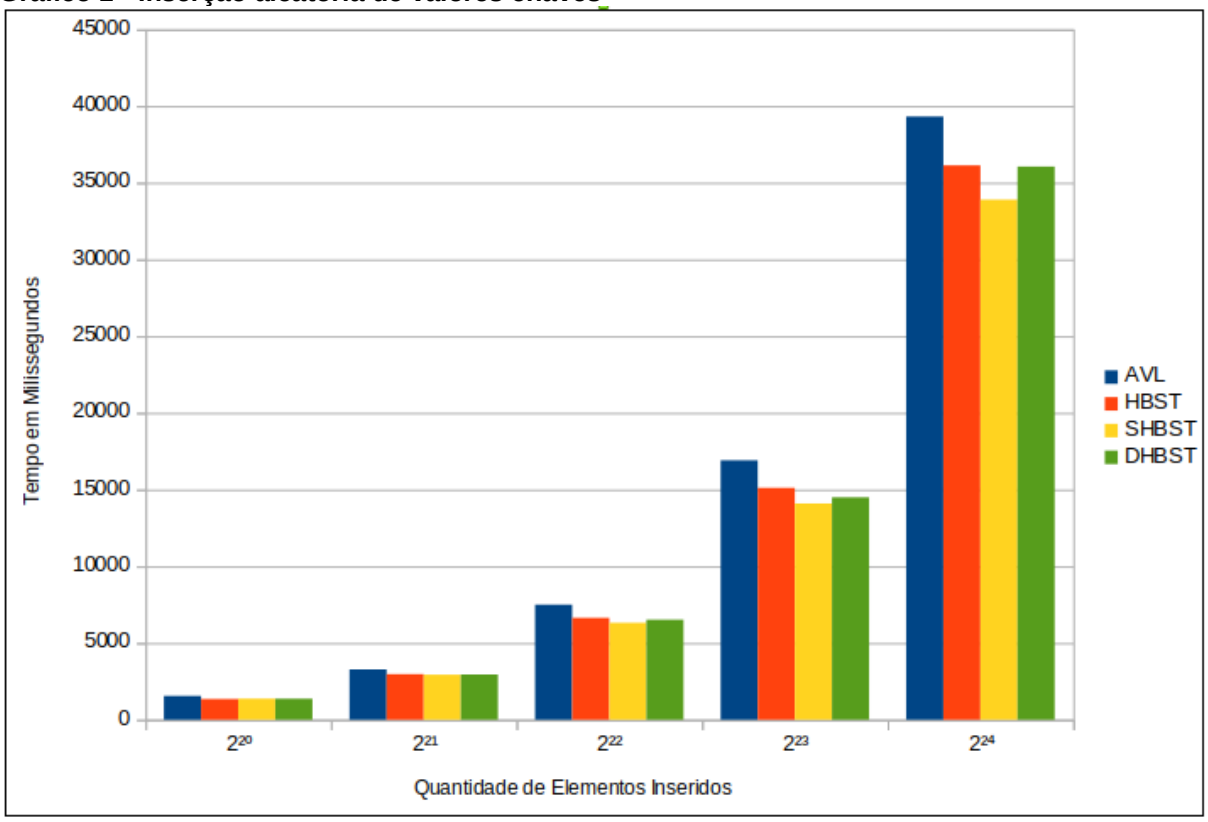
A DHBST usa todo o algoritmo de inserção da SHBST mais um condicional de desvio para criação de nó pai para o nó raiz e consequentemente atualização do $miss$ do antigo nó raiz e atualização do $lower$ ou $upper$ globais (que guardam os limites do intervalo do nó raiz), dependendo se foi criado pai esquerdo ou direito, mais a função de atualização do ref mostrada no Algoritmo 3, mais função de criação de nó intermediário, mais função para atualização do miss do nó folha recém inserido. Sua complexidade de implementação da operação de inserção é comparável a implementação da operação de inserção de uma AVL.

A DHBST é dentre as variações da HBST a que possui mais condicionais de desvio, podendo levar a percepção de ter constantes mais caras, porém quando um nó pai é criado para a raiz, esta inserção nem precisa iterar pela árvore e quando é criado nó intermediário a função termina antes de chegar ao nó folha, ao contrário da HBST e SHBST que sempre inserem novos nós como folhas. Logo esta percepção pode ser verdadeira ou não a depender da instância de entrada.

A HBST e SHBST utilizam a mesma quantidade de memória para cada nó. Enquanto que a DHBST e a AVL possuem ambas um tipo de dado *char* a mais em cada nó, para controle do $miss$ e controle do fb, respectivamente. Além disso, a HBST e suas variantes possuem duas constantes a mais para guardar $lower$ e $upper$ do nó raiz.

O Gráfico 1 apresenta os resultados da análise para a operação de inserção de valores aleatórios.

Gráfico 1 - Inserção aleatória de valores chaves



Fonte: Autoria própria (2018)

Para valores aleatórios, o tempo de execução entre as quatro árvores analisadas é praticamente idêntico, sendo a AVL a estrutura que obteve o pior desempenho em função do tempo nesta análise pelo uso de constantes mais caras para controle de balanceamento e realização de rotações quando necessário.

A SHBST obteve um desempenho de tempo melhor que a HBST, mesmo utilizando dois condicionais a mais na inserção. Este resultado pode ser explicado pela utilização da Propagação Estendida que diminui um nível na altura da árvore por vezes.

7.1.2 Remoção

A AVL precisa atualizar fb dos nós ancestrais ao nó excluído e usar rotação caso necessário. A AVL só aplica substituição quando o nó a ser excluído possui duas subárvores válidas.

A HBST sempre precisa excluir nó folha, logo a HBST aplica substituição tanto para nós com duas subárvores válidas quanto para nós com uma subárvore válida. Para substituir o nó a ser excluído, pode-se usar qualquer valor chave folha de qualquer subárvore do nó a ser excluído.

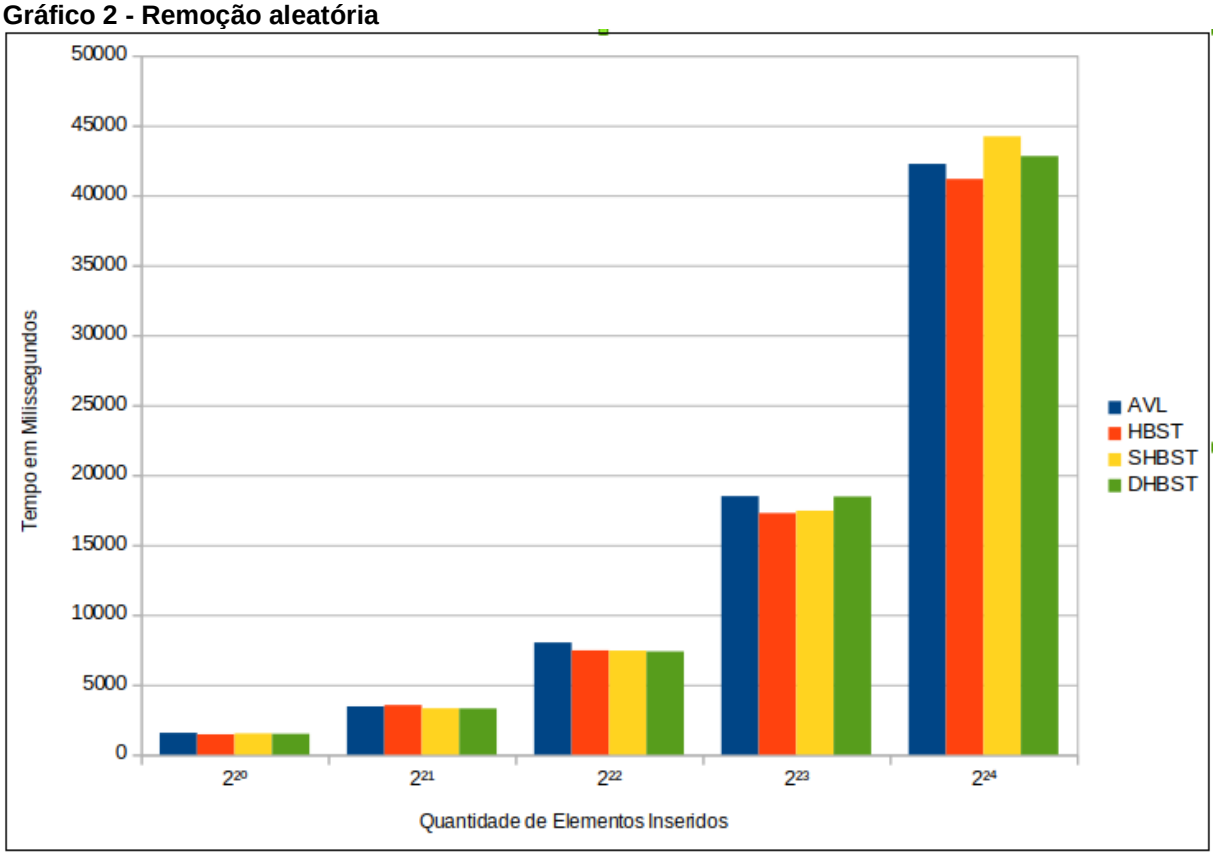
A SHBST sempre precisa excluir nó folha. A SHBST deve substituir o nó pelo maior valor chave da subárvore esquerda ou pelo menor valor chave da subárvore direita. Esta situação de substituição pode-se repetir B vezes no pior caso.

A DHBST, para quisquer outras operações que não a operação de inserção, baseia sua pesquisa pelo valor chave, logo não precisa iterar pelos nós intermediários faltantes como é feito na operação de inserção. A DHBST pode remover um nó (y) que possui apenas uma subávore válida, conectando esta subárvore ao nó pai de y e adicionando o $miss$ de y à subárvore válida. Quando excluindo nó raiz e este raiz possuir apenas uma subárvore válida (z), deve-se atualizar $lower$ ou $upper$ (globais), dependendo se z é subárvore direita ou esquerda do nó raiz respectivamente, correspondente ao tamanho da variável $miss$ de z.

A operação de remoção foi realizada sobre os valores inseridos no teste anterior. A ordem das exclusões foi decidida também pela função *rand()* do C++ e todos os elementos que estavam contidos em cada árvore foram excluídos.

O Gráfico 2 apresenta os resultados.

Gráfico 2 - Remoção aleatória



Fonte: Autoria própria (2018)

Neste cenário a diferença média de tempo de execução é ainda menor que na análise anterior, havendo uma alternância entre as árvores que obtiveram piores resultados em cada uma das quantidades amostrais. Logo observa-se, que todas as estruturas possuem tempo de exclusão equivalentes, e que uma pode ser melhor do que outra dependendo da instância de valores inseridos em cada árvore e sua respectiva ordem.

A SHBST deveria ser sempre pior que a HBST na operação de exclusão em função do tempo de execução, pois ela pode precisar aplicar a substituição mais de uma vez em cada exclusão, enquanto que a HBST aplica uma única vez, porém na análise com 2^{21} elementos inseridos o resultado foi o oposto. A explicação pode ser dada pela escolha do caminho a seguir na subárvore até encontrar um nó folha, pois o algoritmo da HBST está implementado para seguir sempre a esquerda enquanto puder, enquanto a SHBST precisa encontrar o maior valor chave da subárvore esquerda ou o menor valor chave da subárvore direita, logo são caminhos diferentes e podem possuir alturas diferentes.

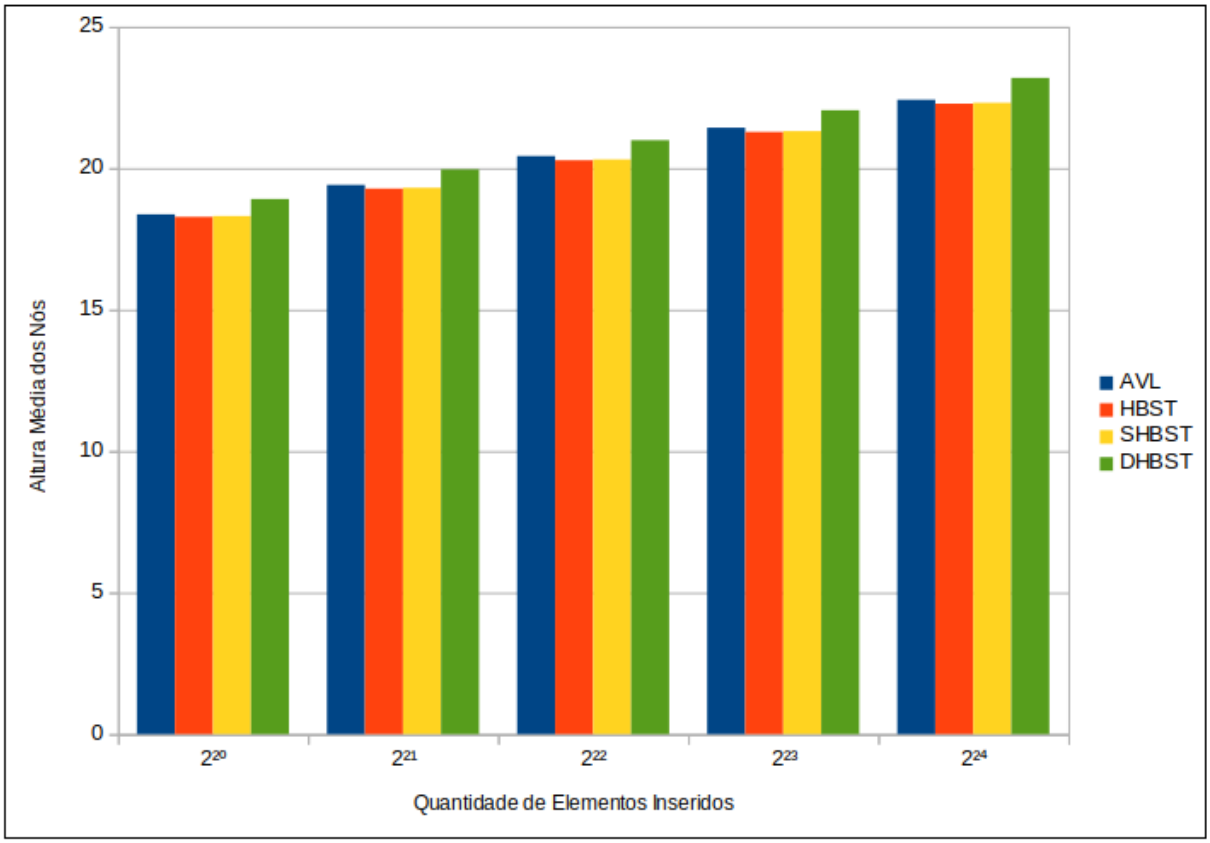
A DHBST deveria ser sempre melhor que a HBST e SHBST na operação de exclusão em função do tempo, pois a DHBST não precisa aplicar substituição quando possui apenas uma subárvore válida, podendo realizar a conexão direta entre nó pai e filho como é feito na AVL. Porém a DHBST obteve um pior resultado nas análise de 2^{23} e de 2^{24} elementos inseridos comparado com a HBST. A explicação pode ser dada pelo fato da DHBST apresentar uma altura média dos nós maior para as mesmas quantidades de elementos inseridos, como pode ser visto no Gráfico 3.

7.1.3 Altura Média dos Nós

A altura média dos nós em cada árvore foi calculada com base na inserção realizada na primeira análise. A altura média foi calculada apenas depois que todos os elementos já estavam inseridos.

O Gráfico 3 apresenta os resultados.

Gráfico 3 - Altura média dos nós



Fonte: Autoria própria (2018)

A DHBST, embora tenha como proposta diminuir a altura média dos nós, obteve o pior desempenho nesta análise. Porém seus resultados são aceitáveis dado que dependendo da ordem de inserção dos valores chaves ela pode obter uma altura média dos nós maior, como descrito na Seção 6.1 e visto na Figura 25.

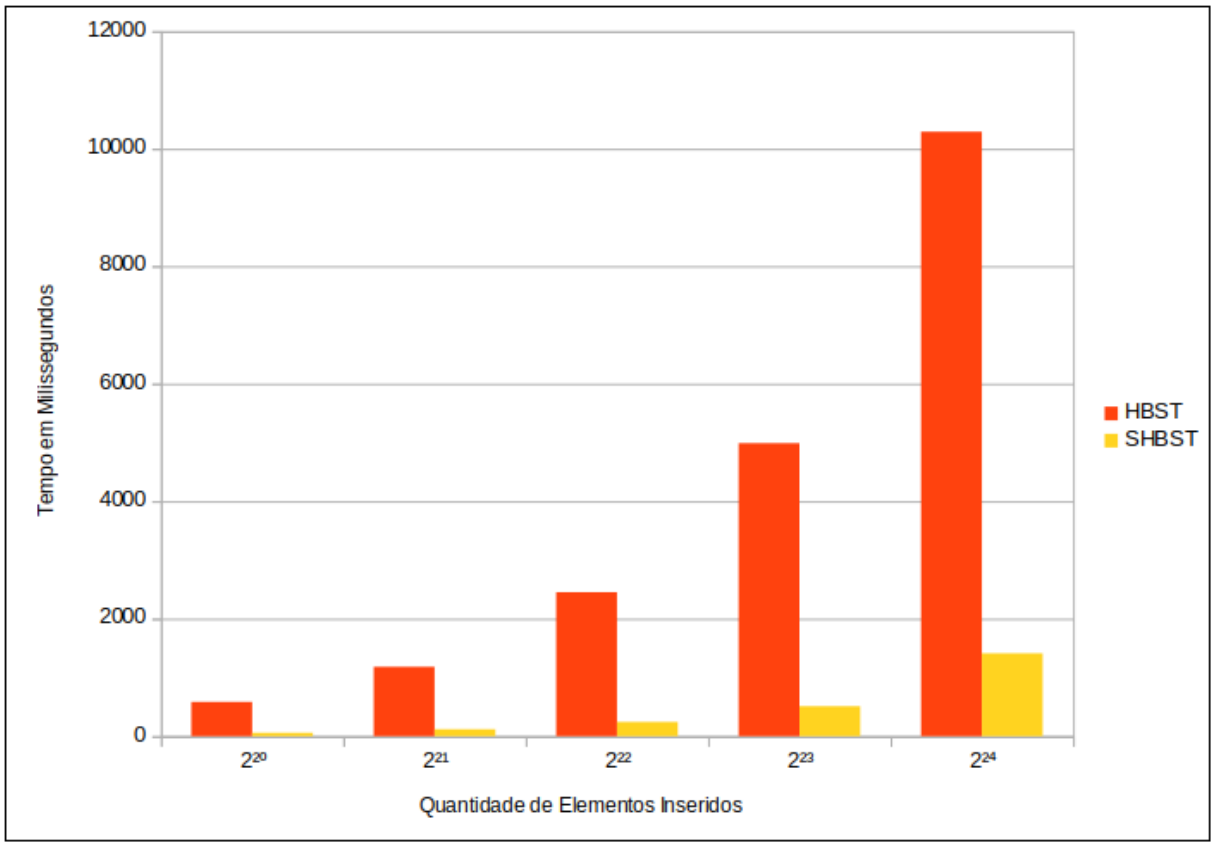
A HBST e SHBST mesmo não possuindo variáveis de controle para balanceamento da árvore, obtiveram uma distribuição uniforme dos nós para valores gerados aleatoriamente, inclusive obtendo altura média dos nós melhor que a árvore AVL, que possui um algoritmo excelente para controle de altura dos nós.

7.1.4 Travessia

A análise de travessia tem por objetivo analisar o tempo de travessia da HBST em relação às demais árvores. Como todas as demais árvores possuem tempo equivalente, dado que todas passam exatamente uma única vez por cada nó, apenas a SHBST foi comparada.

O Gráfico 4 apresenta os resultados.

Gráfico 4 - Travessia entre HBST e SHBST



Fonte: Autoria própria (2018)

Percebe-se que a diferença de tempo da HBST para a SHBST é de aproximadamente 8 vezes no cenário analisado. A SHBST ordena a HBST e por isso consegue realizar a travessia em $\Theta(n)$ enquanto a HBST implementa o algoritmo descrito na Seção 3.2 que possui complexidade $O(n \log_2 B)$.

7.2 VALORES CONTROLADOS

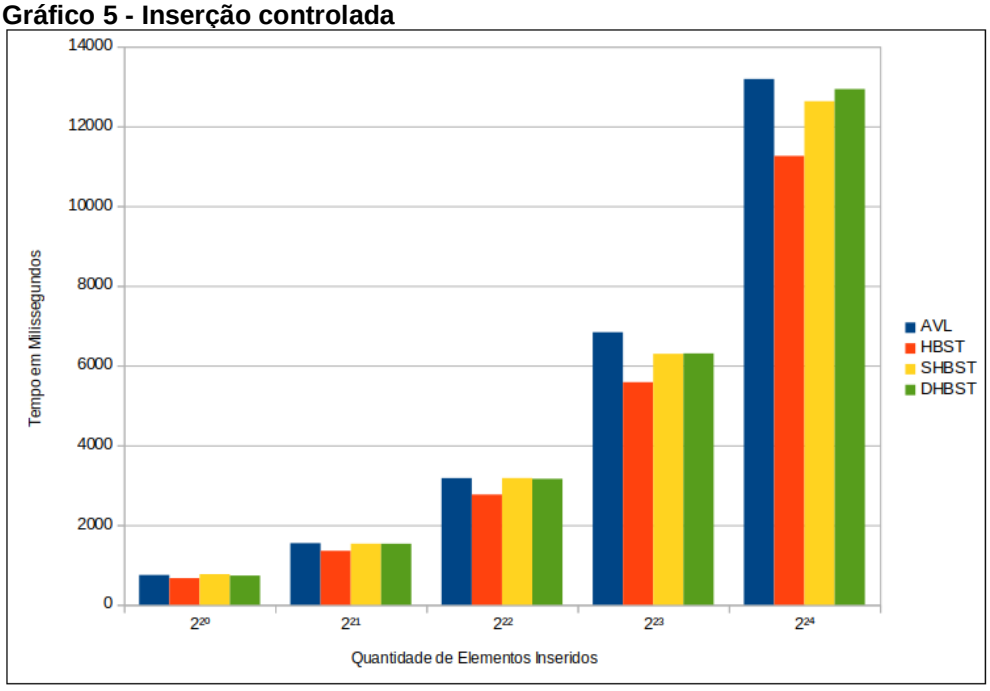
Esta análise visa buscar instâncias que possam promover uma melhor resposta da DHBST. Para isso, buscou-se criar instâncias pertinentes à classe 2.3, com vários intervalos específicos.

7.2.1 Inserção

O algoritmo utilizado para gerar os valores controlados seleciona um número aleatoriamente pela função *rand()* do C++ e a partir deste valor são inseridos 1000 valores chaves sequenciais em ordem crescente com diferença de 1 unidade entre eles. Se algum dos valores já estiver inserido, o intervalo é interrompido e escolhe-se um novo número aleatório para continuar inserindo.

O Gráfico 5 apresenta os resultados para a operação de inserção.

Gráfico 5 - Inserção controlada



Fonte: Autoria própria (2018)

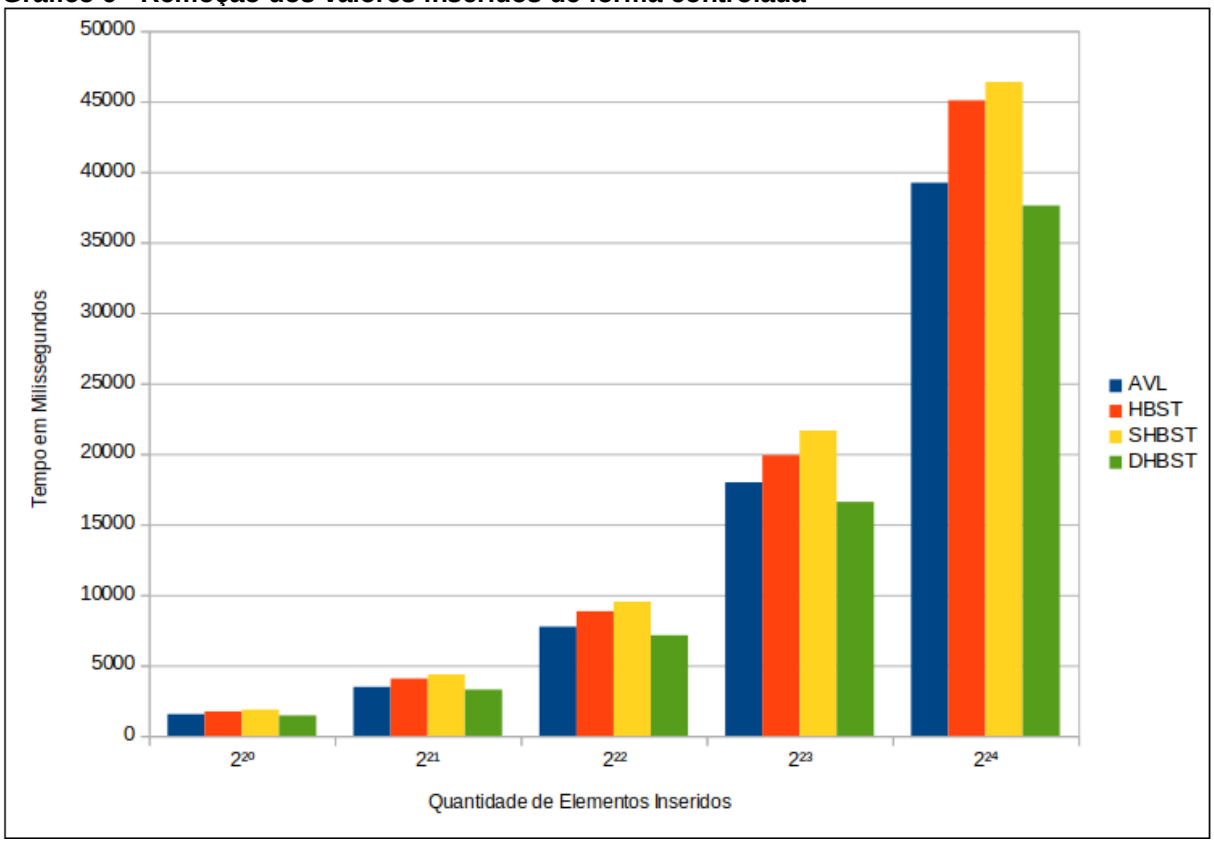
A HBST foi a árvore que apresentou o melhor desempenho em todas as quantidades amostrais mesmo obtendo a pior altura média dos nós, como mostra o Gráfico 7. É provável que a resposta para este resultado se deva ao uso acentuado de memória cache, pois as inserções controladas na HBST ocorrem com frequência seguindo pelo mesmo caminho na árvore e como a HBST não aplica rotações como a AVL nem aplica condicionais de desvio extras como a SHBST e a DHBST, sua performance se sobressaiu.

7.2.2 Remoção

A operação de remoção ocorre seguindo os mesmos critérios aplicados na outra análise de remoção, ou seja, por aleatoriedade, porém removendo os valores que foram inseridos na análise de inserção controlada.

O Gráfico 6 apresenta os resultados.

Gráfico 6 - Remoção dos valores inseridos de forma controlada



Fonte: Autoria própria (2018)

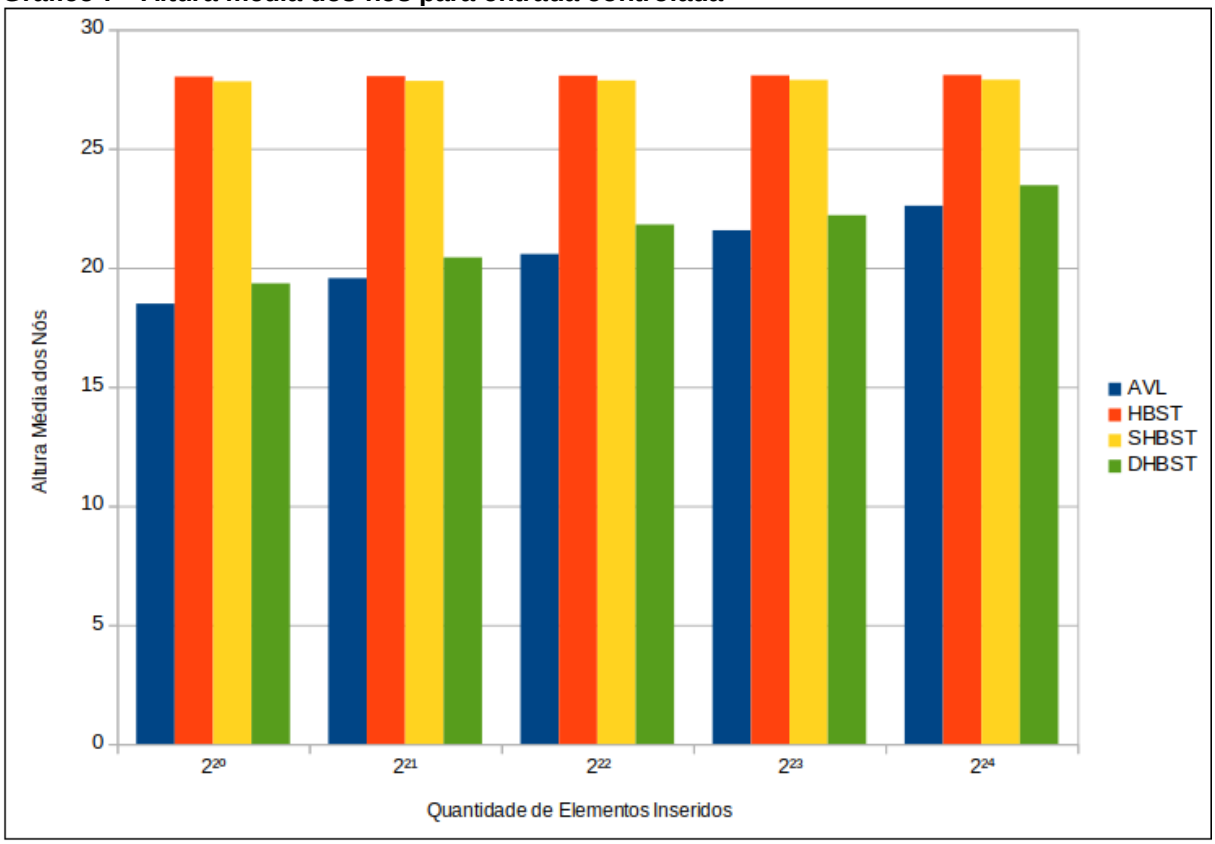
Neste cenário a DHBST apresentou o melhor desempenho em função do tempo seguido pela árvore AVL, pois ambas estruturas obtiveram uma melhor altura média dos nós, como mostra o Gráfico 7. Embora a DHBST tenha obtido uma altura média dos nós maior que a AVL, a DHBST não necessita fazer controle de balanceamento dos nós, por isso obteve um melhor resultado na análise de remoção desta Seção.

7.2.3 Altura Média dos Nós

Altura média dos nós calculada para a inserção controlada somente após todos os elementos estarem inseridos.

O Gráfico 7 apresenta os resultados.

Gráfico 7 - Altura média dos nós para entrada controlada



Fonte: Autoria própria (2018)

Neste cenário a HBST e a SHBST obtiveram seus piores resultados enquanto que a DHBST se mantém competitiva junto a AVL. Percebe-se no Gráfico

7 que a SHBST mantém uma pequena vantagem em relação à HBST devido a aplicação da Propagação Estendida aplicada na SHBST.

Para quantidades amostrais menores, tanto a HBST quanto a SHBST tendem a manter altura próxima do máximo enquanto que a DHBST e a AVL mantém altura logarítmica.

7.3 RESULTADOS

Embora a HBST e a SHBST tenham obtido as piores alturas médias dos nós na análise com valores controlados, seus tempos para as operações de inserção e remoção para os mesmos valores são equivalentes aos da DHBST e da AVL.

Fato que a HBST e suas variantes embora possuam complexidade assintótica $O(B)$ que é teoricamente pior do que $O(\log_2 n)$ como a AVL, na prática estas estruturas são equivalentes em tempo de execução para tipos de dados inteiros. Porém o código da HBST é muito mais simples de implementar do que o código de uma árvore AVL.

A SHBST mostrou-se muito superior a HBST na operação de travessia mantendo mesmo desempenho nas demais operações.

A DHBST mantém uma melhor distribuição média dos nós na árvore em relação à HBST e a SHBST para instâncias da classe 2. Embora a DHBST tenha perdido na análise de números aleatórios para a HBST e SHBST no quesito altura média dos nós, Seção 7.1.3, essa “derrota” possui como diferença uma constante pequena que não consegue aumentar, enquanto que nos casos que a DHBST é superior, a diferença média de altura pode ser até logarítmica.

8 CONCLUSÃO

A HBST representa uma nova classe de árvores binárias de pesquisa. Seu uso prático foi pouco explorado devido ao fato de ter sido pouco divulgada até então. Ainda há muitos campos inexplorados de utilidade da HBST como na área de memória externa, por exemplo, que necessita de um mínimo de movimentações (rotações) dos dados devido à baixa velocidade de processamento destes dispositivos.

A operação de travessia da HBST possui complexidade $O(n \log_2 B)$. Para melhorar este tempo, foi proposto o método de Propagação permitindo que a HBST ficasse ordenada também pelo seu valor chave, criando uma nova árvore denominada *Sorted Hidden Binary Search Tree* (SHBST), logo a pesquisa na estrutura pode ser feita tanto pelo seu ref quanto pelo seu valor chave, e a operação de travessia agora pode ser realizada em $O(n)$.

O método de Propagação Estendida diminui um nível a menos no pior caso da HBST, pois se nenhum valor chave estiver abaixo de algum ref com mesmo valor e a quantidade de nós em uma determinada altura é igual ou maior que a quantidade de possíveis valores chaves, tem-se um ref para cada valor chave até àquela altura.

O uso dos métodos de crescimento dinâmicos abordados não traz impactos significativos. Foram discutidos 6 métodos de Crescimento Dinâmico *in-place* que resolvem o Problema do Salto na HBST: crescimento dinâmico por Memória em 1 e 2 direções, crescimento dinâmico por Salto em 1 e 2 direções e crescimento dinâmico por Degrau em 1 e 2 direções; e um método externo que também resolve o Problema do Salto na HBST e que traz vantagens quanto a economia de memória utilizada.

O método de crescimento por Memória não é viável pois utiliza mais memória que o necessário e pode tornar a altura da árvore maior do que a quantidade de elementos inseridos.

O método de crescimento por Salto não é viável, pois aplica o crescimento dinâmico apenas sobre alguns nós da árvore enquanto que a DHBST aplica sobre todos os nós.

O método de crescimento dinâmico por Degrau não é viável, pois além de não aplicar o crescimento dinâmico sobre todos os nós da árvore, sua complexidade de inserção é $O(B^2)$ e sua implementação é complexa.

O método de crescimento dinâmico com memória externa é viável quando se busca economia de memória, pois este método separa as chaves inseridas pela quantidade de bits que possuem, adequando a chave ao tipo de dado suficiente para comportá-la. E seu uso pode ir além da HBST, podendo ser aplicada tal técnica em outras árvores inclusive.

A DHBST foi introduzida e apresentada como sendo o melhor método de crescimento dinâmico para a HBST, pois permite que nós sejam criados em qualquer parte da HBST, dando margem a exclusão de um nó com uma única subárvore válida sem ser por substituição, aplicando o crescimento dinâmico em todos os nós da árvore e realizando, consequentemente, uma melhor distribuição dos nós na árvore para instâncias da classe 2.

Para instâncias da classe 2.1, é melhor usar crescimento dinâmico “simples”, sem precisar de controle para o Problema do Salto na HBST, o que torna o código simples de implementar e mais eficiente que a AVL, pois mantém altura logarítmica sem precisar balancear os nós. Um exemplo deste tipo de classe ocorre na criação de chaves primárias sequenciais em banco de dados.

A análise empírica entre HBST, SHBST, DHBST e AVL demonstrou que todas as estruturas possuem tempo de execução muito semelhante para as operações de inserção e remoção para chaves do tipo inteiro, e que por isso o uso da HBST ou SHBST se torna mais vantajoso devido a sua facilidade de implementação e menor uso de memória em comparação com a AVL e DHBST. E como HBST e SHBST também obtiveram resultados equivalentes nas operações de inserção e remoção e também na altura média dos nós, o uso da SHBST se torna mais vantajoso devido ao fato da SHBST permitir a pesquisa pelo valor chave, acarretando a realização da operação de travessia em $O(n)$, enquanto que a HBST possui complexidade $O(n \log_2 B)$ para a operação de travessia.

A DHBST utiliza mesma quantidade de memória que a árvore AVL e sua implementação é tão complexa quanto. Seus resultados na análise empírica também foram semelhantes, logo conclui-se que ambas são equivalentes para tipos de dados inteiros.

O uso da DHBST pode ter impacto mais significativo quando aplicado a tipos de dados que possuem grande quantidade de bits. Porém isto ainda precisa ser estudado.

O trabalho teve por objetivo discutir o funcionamento de métodos de crescimento dinâmico da HBST e apresentar a SHBST e a DHBST, variações da HBST que promovem benefícios de tempo de execução para a operação de travessia e de altura dos nós, respectivamente.

REFERÊNCIAS

ADELSON-VELSKY, G. M. and LANDIS E. M. An algorithm for the organization of information. In Proceedings of the USSR Academy of Sciences, v. 146, p. 263–266, 1962.

CHANG, His; IYANGAR, SITHARAMA. Efficient Algorithms to Globally Balance a Binary Search Tree**. Communications of the ACM 27.7** (1984): 695-702.

CORMEN, T. H; LEISERSON, C. E; RIVEST, R. L; STEIN, C. **Introduction to Algorithms, Third Edition**. The MIT Press, 3ª edição, 2009.

QUEIROZ, Saulo. The Hidden Binary Search Tree: A Balanced Rotation-Free Search Tree in the AVL RAM Model. arXiv preprint arXiv*:1711.07746* (2017).

QUEIROZ, Saulo; BAUER, Edimar. The Hidden Binary Search Tree. **Encontro de Teoria da Computação (ETC_CSBC)**, v. 3, n. 1/2018, july 2018.

KNUTH, Donald. **The Art of Computer Programming**, v. 3: (2Nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.