ISEL | DEETC

Algoritmos e Estruturas de Dados

3ª Serie de Exercícios

Semestre de Inverno de 2009/2010

Autores:
30896 – Ricardo Canto
31401 – Nuno Cancelo
33595 – Nuno Sousa

Indície

Enunciado

1. Realize a classe genérica `AedHashTable<E>` para a representação e manipulação de tabelas de dispersão, contendo os seguintes métodos de instância públicos:

   - `void put(E e)`, que insere o elemento e na tabela.
   - `boolean contains(E e)`, que retorna `true` se e só se a tabela contiver um elemento igual a e.
   - `void removeAllDuplicates()`, que remove todos os elementos duplicados, de forma a que a instância só contenha elementos distintos.
   - `E putUnique(E e)`, que insere o elemento e na tabela se e só se a tabela não contiver outro elemento igual. Caso já exista um elemento igual, retorna esse elemento. Caso contrário, retorna `null`.

   Esta classe deve implementar a interface `Iterable<E>`. O iterador retornado pelo método `iterator()` deve suportar remoção.

   Utilize o método `equals`, declarado na classe`Object`, para verificar se dois elementos são iguais.

2. Realize a classe contendo os seguintes métodos estáticos para a manipulação de árvores. Assuma que cada objecto do tipo `Node<E>` tem 3 campos: um `value` do tipo `E` e duas referências, `left` e `right`, para os descendentes respectivos.

   2.1. `public static <E> int copyToArray(Node<E> root, E[] v);`

   que copia para o *array* `v` os elementos da árvore binária de pesquisa referenciada por `root`. Os elementos devem ficar organizados no *array* por ordem decrescente, com o maior elemento na primeira posição. Caso a dimensão da árvore exceda a dimensão do *array*, apenas são copiados os últimos `v.length` elementos.

   2.2. `public static Node<Integer> createBSTFromArithmeticProgression(int a, int d, int n)`

   que retorna a referência para o nó raíz duma árvore binária de pesquisa contendo os inteiros da progressão aritmética com início em a, dimensão n e factor d (diferença entre dois elementos consecutivos $d = a_i - a_{i-1}$). Assuma que o factor d é maior do que zero. A árvore resultante deve estar balanceada.

   2.3. `public static <E extends Comparable<E>> int rangeCount(Node<E> root, E l, E r)`

   que retorna o número de elementos da árvore binária de pesquisa com raíz `root` pertencentes ao intervalo $[l, r]$. Considere que o critério de organização da árvore binária de pesquisa é a *ordenação natural* do tipo `E`.

3. Acrescente, à classe `Iterables` realizada na segunda série de exercícios, o método

   ```
   public static <E> Iterable<E> distinct(Iterable<E> iter)
   ```

   que retorna um objecto com a interface `Iterable<E>`, representando a sequência `iter` sem os elementos duplicados.

   Não é necessário implementar o método `remove`, pertencente à interface genérica `Iterator<E>`.

   A implementação deste método deve minimizar o espaço ocupado pelo iterador.

4. Esquematize a inserção da seguinte sequência de elementos numa *B-Tree* com $M = 2T - 1 = 5$ (número máximo de chaves por nó/página):

   $\{10, 20, 30, 40, 50, 4, 5, 6, 7, 31, 32, 33, 34\}$.

Exercício 1

Neste exercício analisamos todo o contexto do exercício de forma a podermos realizar uma introspecção eficiente sobre a forma de implementação.

Uma vez que nos é dito que a classe deve ser genérica, assumimos que o numero de elementos da tabela de dispersão é infinita e assim sendo optamos pela solução de Dispersão por separação em listas, que torna a resolução de colisões mais eficaz, tornando-se mais eficiente a analise.

De acordo com os exercícios solicitados, concluímos que para a sua execução seria bastante eficiente se os objectos E fossem comparáveis e para tal seria necessário obter um comparador. Assim optamos (com o acordo da nossa Docente) que o nosso construtor recebesse como parâmetro o Comparador, retornando uma excepção (NullPointException) caso o comparador fosse null.

Para a tabela de dispersão em listas procedemos à implementação de duas classes do tipo Node. A classe Node que contem os apontadores para os Node anterior e posterior e o elemento E, e a classe ExtNode que estende de Node e inclui um campo indicador do numero de elementos que a lista contém e alguns métodos para obtenção, incremento e decremento desse mesmo campo.

Desta forma, apresentamos uma forma simples de controlar se a tabela de dispersão necessita ou não de ser feito um rearranjo dos seus elementos. Assumimos o valor indicativo de 70% de preenchimento de cada lista sobre o tamanho da tabela. Desta forma, analisamos o pior caso de todos os elementos serem remetidos para a mesma lista e sendo esse o caso testamos se lista está a ocupar os 70% antes de proceder à sua inserção, caso contrário aumentamos a tabela de dispersão para o triplo do seu tamanho e procedemos à re-dispersão de todos os elementos.

Para procedermos à navegação pelas listas, implementámos um objecto Iterable (como era solicitado) mas que procede à iteração dos elementos da lista e não sobre a tabela.

Ao realizarmos os métodos solicitados, houve necessidade da nossa parte implementar métodos auxiliares para simplificar a interacção por entre os métodos, tais como:

- void put(E e)
  - private int getIndex(E e)

    Obtém o indicie para o elemento 'e'.
  - private void add(ExtNode<E> ex, Node<E> n)

    insere de forma ordenada o novo nó 'n' na lista. Tem um custo linear na inserção, mas torna os restante métodos bastante mais eficientes.
  - private void addHere(Node<E> head, Node<E> newNode)

    insere o novo nó na posição de head.

- boolean contains(E e)
  - private Node<E> contain(E e)

    Este método foi implementado para tornar mais eficaz os dois métodos seguintes, reduzindo o numero de vezes a iteração por entre as listas.
- void removeAllDuplicates()
  - private void removeDuplicates(ExtNode<E> n)

Navega sobre uma lista removendo os elementos duplicados. Foi implementado o método, para tornar o código mais legivel e separar o iterar sobre a tabela e iterar a lista.

- public E putUnique(E e)

  Utiliza o contain(e) para poder inserir de forma única os elementos.

Mais informações sobre os métodos podem ser encontrados na implementação de AEDHashTree.java.

Exercício 2

A primeira alínea foi de fácil implementação uma vez que nos foi indicado que o "Os elementos devem ficar organizados no array por ordem decrescente, com o maior elemento na primeira posição.", assim bastou percorrer a árvore binário para a direita e de forma infixa(right , current, left).

Na segunda alínea a implementação foi bastante lenta, apesar da solução evidente. Isto porque não nos foi possível durante muito tempo obter a localização dos indicies correctos, uma vez que para quando houve-se um numero elevado de elementos os indicies ficavam de alguma forma trocados. Apesar da valiosa ajuda da nossa Docente, quando os numero de elementos era pequeno o algoritmo funcionava, mas o caso mudava de figura quando criava-mos árvores de três níveis e que tinha mais elementos para um lado do que para o outro.

Numa última tentativa e forma a analisar os últimos comentários que nos foram lançados, foi possível encontrar uma solução funcional para o exercício.

A implementação de um método para calcular o valor da progressão aritmética foi bastante simples, sendo mesmo o primeiro método implementado.

A construção da solução para a última alínea foi bastante rápida, pois era evidente o que era solicitado e o que se tinha que fazer, sendo necessário ter em atenção os casos de paragem.

Exercício 3

Para a realização deste exercício procedemos como solicitado, contudo não acrescentamos à mesma classe da segunda série, por acharmos que apesar fazer sentido que métodos com funções semelhantes deverem estar agrupados na mesma classe genérica, no contexto desta série não haveria necessidade de acrescentar métodos que não trazem mais-valia ao trabalho.

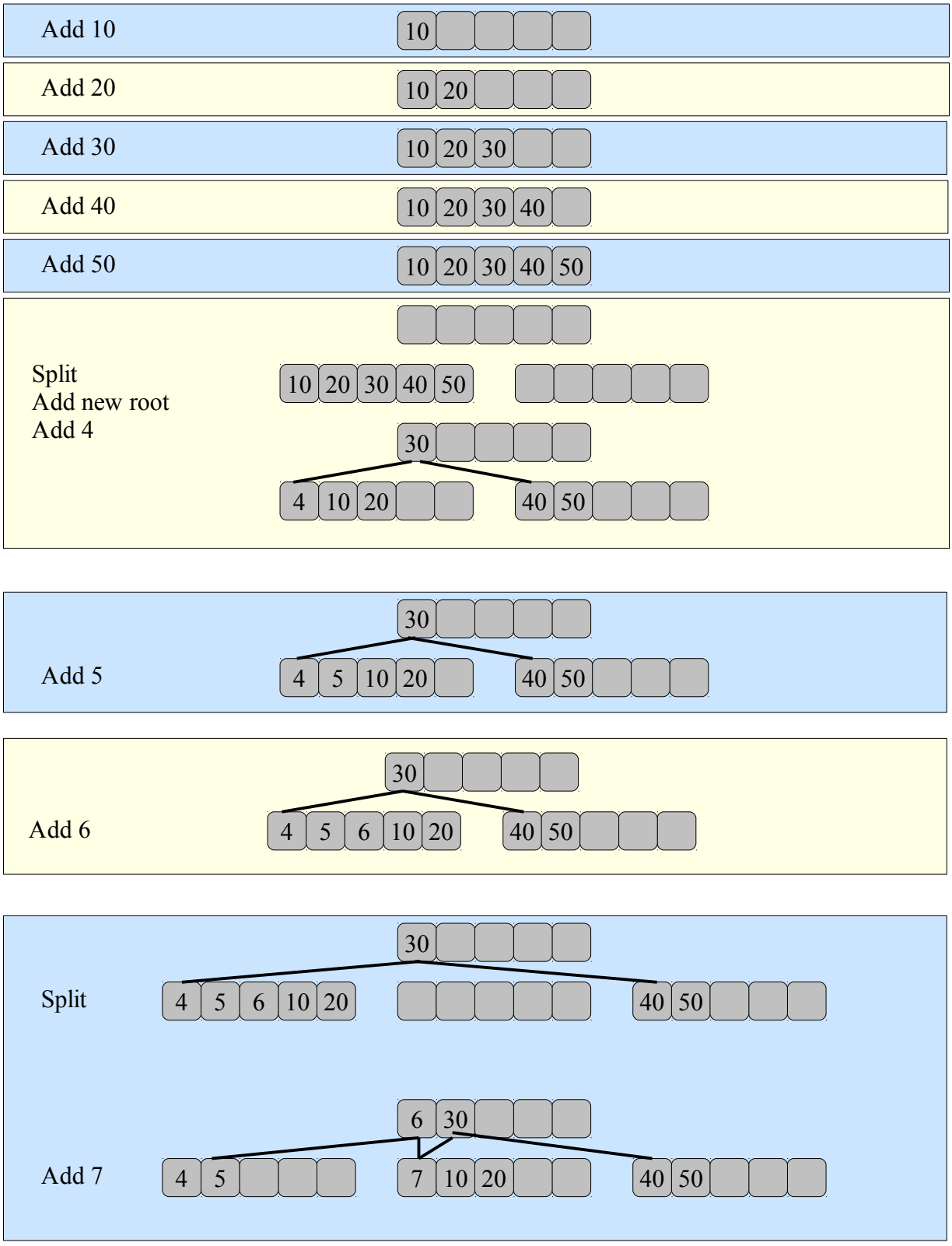
De forma a executar o solicitado o nosso método distinct utiliza uma estrutura de dados em lista para manter o "histórico" de elementos que já foram retornados, de forma a poder responder ao solicitado.

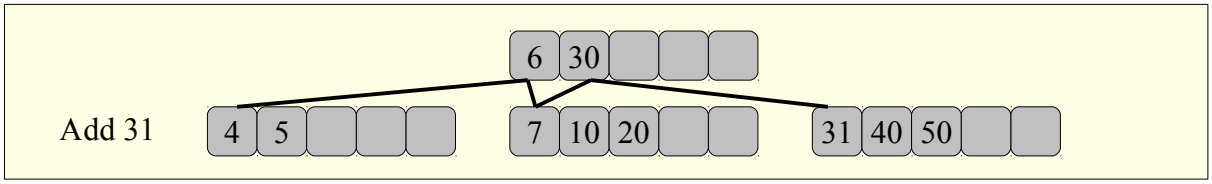
Poderíamos ter implementado uma estrutura em forma de árvore binária para reduzir o tempo de pesquisa. Para tal teríamos que alterar o elemento E para ser comparável e introduzir os elementos ordenado.

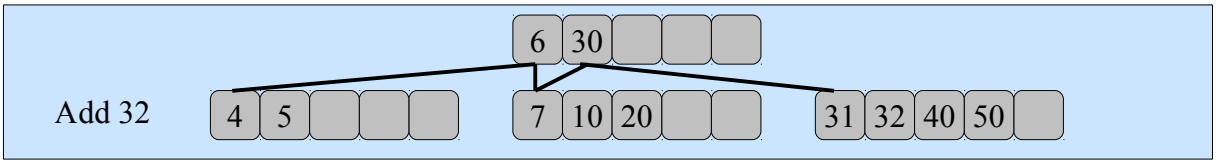
Poderíamos também utilizar a nossa estrutura do primeiro exercício como forma de manter os elementos do iterável.

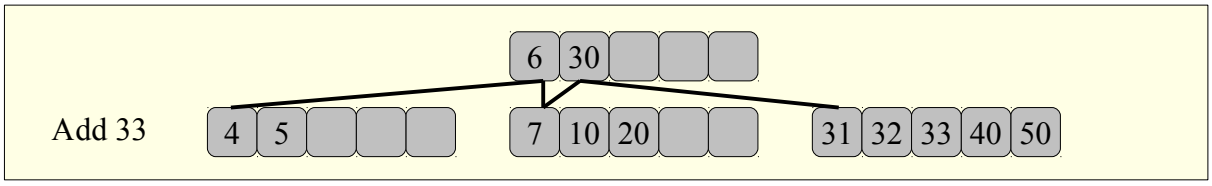
Nós temos consciência que esta alteração seria benéfica para a implementação e achamos que o facto de darmos a conhecer que sabemos da solução, apesar de não a implementarmos na altura, não tira a valorização pedagógica que o exercício tem. Aprendemos qual seria a melhor estrutura, mas não achamos necessário investir mais tempo na sua implementação.

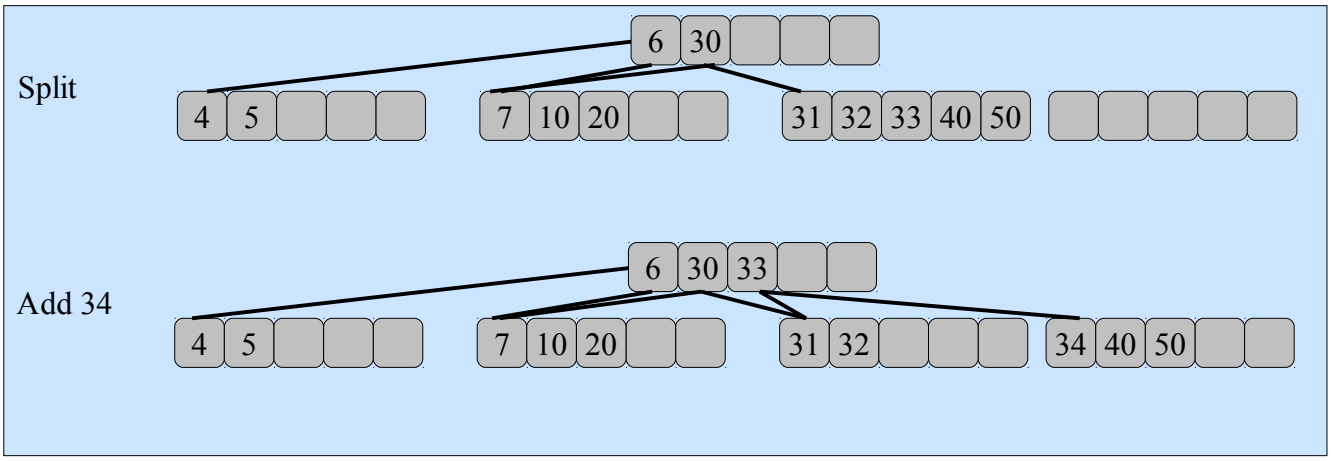
Exercício 4


Add 10
10
Add 20
10 20
Add 30
10 20 30
Add 40
10 20 30 40
Add 50
10 20 30 40 50
Split
Add new root
Add 4
10 20 30 40 50
30
4 10 20
40 50
Add 5
30
4 5 10 20
40 50
Add 6
30
4 5 6 10 20
40 50
Split
30
4 5 6 10 20
40 50
Add 7
6 30
4 5
7 10 20
40 50


Add 31
6 30
4 5
7 10 20
31 40 50


Add 32
6 30
4 5
7 10 20
31 32 40 50


Add 33
6 30
4 5
7 10 20
31 32 33 40 50


Split
6 30
4 5
7 10 20
31 32 33 40 50
Add 34
6 30 33
4 5
7 10 20
31 32
34 40 50

Conclusão

Nesta série foi possível analisar forma de navegação sobre as árvores binárias e que as mesmas são bastante eficientes na analise dos custos, mantendo o tempo de execução em tempo logaritmo.

Tornou-se também evidente alguns pontos discutidos na aula na implementação das soluções, sendo necessário alguma atenção na análise do problema para que a solução seja “simple”.

Queremos agradecer à nossa Docente pelo tempo dispensado durante este curto período de descanso para nos esclarecer e ajudar em alguns pontos que nos estava a atrapalhar.