ISEL | DEETC

Algoritmos e Estruturas de Dados

2ª Série de Exercícios

Semestre de Inverno de 2009/2010

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

Indície

Enunciado.....3
Resolução.....4
Exercício 1.....4
Alínea 1.1.....4
Alínea 1.2.....5
Alínea 1.3.....6
Conclusão:.....7
Exercício 2:.....8
Exercício 3:.....9
Alínea 3.1.....9
Alínea 3.2.....10
Exercício 4.....12
Bibliografia.....14

Enunciado

1. Realize a classe `BinaryHeaps` contendo os seguintes métodos estáticos.

   1.1. `public static void printByDepth(int[] v, int count);`
   Este método apresenta na consola os inteiros presentes no amontoado binário representado pelos primeiros `count` inteiros do *array* `v`. Esta apresentação deve ter a seguinte organização:
   - Apresentar na mesma linha da consola os elementos com a mesma profundidade (distância à raíz).
   - Apresentar primeiro a linha com a raíz e por último a linha com os elementos de maior profundidade.
   - Em cada linha, os elementos são apresentados da esquerda para a direita.

   1.2. `public int countMaxInMaxHeap(int[] v, int count)`
   Este método, dado o *max-heap* representado pelos primeiros `count` inteiros do *array* `v`, retorna o número de ocorrências do maior elemento presente nesse *max-heap*.

   1.3. `public static int largestSubArrayThatIsAMaxHeap(int[] v);`
   Este método retorna a dimensão do maior *sub-array* de v, com início no índice 0, que representa um *max-heap*.

2. Realize a classe `AedLinkedList<E>` com a mesma funcionalidade da classe `java.util.LinkedList<E>`. A classe realizada pode extender a classe `java.util.AbstractCollection<E>`.

3. Realize a classe `Iterables` contendo os seguintes métodos estáticos:

   3.1. `public static <E> Iterable<E> concat(Iterable<E> iter1, Iterable<E> iter2)`
   que retorna um objecto com a interface `Iterable<E>`, representando a concatenação das sequências `iter1` e `iter2`. Não é necessário implementar o método `remove`, pertencente à interface genérica `Iterator<E>`.
   A implementação deste método deve minimizar o espaço ocupado pelo iterador.

   3.2. `public static <E> Iterable<E> takeWhile(Iterable<E> iter, Predicate<T> pred){`
   que retorna um objecto com a interface `Iterable<E>`, representando o maior prefixo da sequência `iter` em que todos os elementos satisfazem o predicado `pred`. Não é necessário implementar o método `remove`, pertencente à interface genérica `Iterator<E>`
   A implementação deste método deve minimizar o espaço ocupado pelo iterador.
   Use a seguinte definição da interface `pred`.

   ```
   public interface Predicate<E> {
       boolean eval(E e);
   }
   ```

4. Considere a representação de redes rodoviárias através de *grafos não dirigidos com pesos* (ver capítulo 22 e apêndice B.4 do livro de referência):
   - Cada intersecção entre ligações da rede corresponde a um nó do grafo.
   - Cada ligação na rede corresponde a um arco do grafo. Este arco tem associado a distância da ligação.

   Considere também os ficheiros com redes rodoviárias presentes em

   `http://www.dis.uniroma1.it/~challenge9/download.shtml`

   Os formatos destes ficheiros estão descritos em

   `http://www.dis.uniroma1.it/~challenge9/format.shtml`

   Realize um programa que, dado um ficheiro com a definição dum grafo, produza outro ficheiro com a definição do mesmo grafo mas com os arcos ordenados por ordem crescente do seu custo.

   Assuma que o conteúdo do grafo excede a dimensão de memória disponível na máquina virtual. Utilize o algoritmo de *ordenação externa* descrito na secção 11.4 do livro R. Sedgewick, "Algorithms in Java", 3ª edição, Addison-Wesley, 2003.

Resolução

Exercício 1

Alínea 1.1

Na analise desta alínea, concluímos que não seria problemática a sua implementação, uma vez que o essencial para poder resolver o proposto:

- o Heap a ser impresso
- o a quantidade de elementos que o heap contém

eram fornecidos pelos argumentos.

Assim, numa breve análise a estrutura de um heap, inferimos que o numero de elementos em de cada nível é equivalente a ter $2^n, n \in N_0$. Sendo este facto verdade, optámos por controlar uma variável auxiliar que irá controlar quando é que chega ao fim de cada linha de impressão.

```
public static void printByDepth(int[] v, int count){
    int level=0;
    int len=(int)Math.pow(2, level);
    for (int i=0,j=0; i<count ;i++){
        System.out.printf("[ %5d ]\t",v[i]);
        ++j;
        if(j == len){
            System.out.printf("\n");
            level++;
            j=0;
            len=(int)Math.pow(2, level);
        }
    }
}
```

Alínea 1.2

Uma das condições para a realização deste exercício, provém de no enunciado indicarem que nos parâmetros fornecidos já estão em max-heap, tornando um pouco a resolução do problema mais simples.

Sendo um max-heap uma estrutura em árvore binária, tal que o ***parent[i] >= childs[i]*** então sem parâmetros adicionais basta verificar desde a raiz até a condição de que são iguais falhe.

```
public static int countMaxInMaxHeap(int[] v, int count){
   return countMaxinHeap(v, 0, count,v[0]);
}
public static int countMaxinHeap(int[] v, int i, int count ,int max){
   if (2*i+1>=count) return 0;
   if (2*i+2>=count) return 0;
   if (v[i] != max) return 0;
   int total =0;
   if(v[2*i+1] == max) total+=countMaxinHeap(v, 2*i+1, count,max);
   if(v[2*i+2] == max) total+=countMaxinHeap(v, 2*i+2, count,max);
   return total+1;
}
```

Decidimos separar o método por duas razões:

- podermos chamar recursivamente
- ter um método mais generalista que nos permita efectuar a mesma operação para qualquer nó do heap.

Alínea 1.3

Esta alínea foi mais trabalhosa, quer na análise quer na implementação uma vez que foi complicado criar uma camada de abstracção para poder resolver o heap, seja qual for o seu tamanho.

Partimos a análise do heap a partir do seu último parent, verificamos se está em max-heap ou não. Caso não esteja em max-heap, por definição, todos os seus antecessores também não estão em max-heap.

Esta analise é registada num array auxiliar de inteiros onde é indicado com o valor -1 que esse nó não está em max-heap, para posteriormente ser efectuada a contagem de elementos em max-heap.

```
public static int largestSubArrayThatIsAMaxHeap(int[] v){
   return checkMaxHeap(v);
}

public static int checkMaxHeap(int[] v){
   int lastParent=((v.length-1)-1)/2;
   int[] parents=new int[v.length];
     do {
      if (parents[lastParent] != -1){
        if(isMaxHeap(v, lastParent)){
          parents[lastParent]=1;
        }else{
          setIsNotMaxHeap(parents, lastParent);
        }
      }
      --lastParent;
   }while(lastParent>=0);
  return getMaxInMaxHeap(parents, v.length);
}
  public static int getMaxInMaxHeap(int[] v, int dim){
      int max=0;
      for (int i=0;i<dim;++i){
          if (v[i] == 1){
              return getLen(v, i, dim);
          }
      }
      return max;
  }
```

```
    private static int getLen(int[] v, int i,int dim){
        if (i>=dim) return 0;
        int total=0;
        total+= getLen(v, 2*i+1, dim);
        total+= getLen(v, 2*i+2, dim);
        return total+1;
    }

    public static void setIsNotMaxHeap(int v[], int idx){
        while (idx>0){
            v[idx]=-1;
            idx=(idx-1)/2;
        }
        v[idx]=-1;
    }
    public static boolean isMaxHeap(int[] v,int idx){
             return  (v[2*idx+1] <= v[idx]   && ( ((2*idx+2)< v.length) ? v[2*idx+2] <=
v[idx]:true) );
    }
```

Como se pode verificar, este problema foi dividido em sub-problemas tendo métodos para resolver cada situação de forma singular tornando o código legível e elegante.

O retorno do numero de elementos tem em conta as propriedades do max-heap, executando de forma eficiente a analise do heap.

Conclusão:

Na elaboração desta primeira fase do trabalho, verificamos que o estudo e analise dos heap não é complicado, mas deve o ser feito com precaução, pois um analise incorrecta leva a implementações desastrosas.

Enquanto as duas primeiras alíneas são de grau de dificuldade simples, no terceiro já não podemos afirmar o mesmo, como se pode verificar da implementação que seguimos, que existiram muitos pormenores em ter em conta.

No fim deste exercício achamos que o que foi proposto foi eficaz na implementação dos nossos conhecimentos.

Exercício 2:

Este exercício foi complicado de ser inicializado uma vez que ao estendermos somente de java.util.AbstractCollection<E> não teríamos todos os métodos necessários para que a nossa classe tivesse a mesma funcionalidade de java.util.LinkedList.

Para que tal acontece-se foi necessário implementar as interfaces List<E> e Deque<E> que fez com que implementássemos quase todos os métodos que LinkedList<E> tivesse, dai ser o propósito secundário deste exercício, um vez que o principal seria como é que as listas funcionariam como estrutura de dados, fazendo a implementação que garantisse essa estrutura.

A implementação desta classe foi facilitada pela leitura da documentação de LinkedList e das interfaces que ela implementa, tornado a implementação de muitos métodos bastante rápidos, uma vez que se conseguiam à custa de outros.

Os métodos que requereram mais atenção e tempo foram os que implicavam a adição, de remoção e de indexação de elementos da estrutura, uma vez que era necessário compreender as situações em que as mesma aconteciam.

Dada a generalidade da classe, não colocamos a sua implementação aqui no relatório, mas será disponibilizada como anexo, no qual tem comentários javadoc em cada método, que tipificando, será semelhante ao que existe para a LinkedList.

Exercício 3:

Este exercício serviu para demonstrar uma (óptima) maneira de implementar filtros a estruturas de dados. Esta forma de entendimento e de utilização dos iteráveis, foram úteis na implementação do exercício seguinte.

Ambos os exercícios são de execução simples mas de grande eficácia, permitindo a construção de classes anónimas para alcançar o objectivo pretendido sem estar comprometido com nenhuma implementação, nem comprometendo futuras utilizações destes métodos.

Somente foi necessário respeitar a assinatura dos métodos assim como a implementação das respectivas interfaces, realizando posteriormente os filtros necessários para a efectuar o solicitado.

Alínea 3.1

```
public static <E> Iterable<E> concat(final Iterable<E> iter1, final Iterable<E> iter2) {
    return new Iterable<E>() {

        public Iterator<E> iterator() {
            return new Iterator<E>() {
                private Iterator<E> i1 = iter1.iterator();
                private Iterator<E> i2 = null;
                E element =null;

                public boolean hasNext() {
                    if (i1.hasNext()){
                        element=i1.next();
                        return true;
                    }else{
                        if(i2 == null)i2 = iter2.iterator();
                        if (i2.hasNext()){
                            element=i2.next();
                            return true;
                        }else{
                            element=null;
                            return false;
                        }
                    }
                }

                public E next() {
                    if (element == null && !hasNext()) {
                        throw new NoSuchElementException();
                    }
                    E e = element;
```

```
                    element=null;
                    return e;
                }

                public void remove() {
                    throw new UnsupportedOperationException("Not supported yet.");
                }
            };
        }
    };
}
```

Na implementação desta solução seguimos a sugestão de *"minimizar o espaço do iterador", como se pode analisar no método hasNext(), contudo para minimizar o espaço ocupado pelo iterador, teremos mais processamento a ser realizado pela máquina virtual, uma vez que enquanto o segundo iterador não tiver terminado será avaliada avaliada a instrução* `if(i2 == null)i2 = iter2.iterator();`

Alínea 3.2

```
public static <E> Iterable<E> takeWhile(final Iterable<E> iter, final Predicate<E> pred) {
  return new Iterable<E>() {

    public Iterator<E> iterator() {
      return new Iterator<E>() {
        private E element=null;
        private Iterator<E> it=iter.iterator();
        public boolean hasNext() {
          while (it.hasNext()){
            if (pred.eval(element=it.next())) {
              return true;
            }
          }
          element=null;
          return false;
        }
        public E next() {
          if (element==null && !hasNext()){
            throw new NoSuchElementException();
          }
          E e=element;
          element=null;
          return e;
        }
```

```
            public void remove() {
               throw new UnsupportedOperationException("Not supported yet.");
            }
         };
      }
   };
}

public interface Predicate<E> {

   boolean eval(E e);
}
```

Exercício 4

O algoritmo de ordenação externa, nasceu nos anos 50 com o intuito de ordenar grandes quantidades de informação, não havendo memória física para proceder à sua ordenação. Numa altura que os recursos eram de custos elevados, esta solução teve grandes adeptos ao longo dos anos seguintes.

Nos dias que correm, esta solução tornou-se um pouco obsoleta devido à redução drástica do preço das memórias e da velocidade dos processadores. No entanto, é uma óptima maneira de testar algoritmos assumindo que se tem pouca memória para poder processar toda a informação disponível.

Foi interessante analisar o modo de funcionamento deste tipo de ordenação e implementa-lo, uma vez que levando uma série de variáveis para além do próprio algoritmo, como seja:

- **Hardware**
  - Velocidade dos Discos
  - Velocidade das Memórias
  - Tipo de Sistema de Ficheiros
- **Software**
  - Sistema Operativo que está a correr
  - Se está a correr software de Privacy Protection

Nesta nossa implementação tivemos testes muito promissores, uma vez que estávamos a obter valores na ordem dos 3 segundos ao executa-lo contra o ficheiro ***USA-road-d.NY.gr*** que contem pouco mais de 700.000 registos e cerca de 30 minutos ao executar no ficheiro ***USA-road-d.USA.gr*** que contém cerca de 58.000.000 de registos.

No entanto, ao seguir a sugestão de utilizar o método String.split em deterioramento do StringTokenizer, uma vez que é desaconselhado o uso pela documentação, perdemos performance na execução do programa, ficando o programa a demorar, no ficheiro mais pequeno, cerca de 12 segundos. Este resultado impulsionou-nos a tomar a decisão de voltar ao uso do StringTokenizer, que nos reduziu o tempo de execução para 7 segundos. Efectuamos testes, no intuito de utilizar a classe Pattern e expressões regulares, como forma de tornar o '*parse*' das strings mais rápidas, mas os testes não demonstraram ganhos na execução. Apesar da melhoria de performance, não nos foi possível voltar a ter tempos na ordem dos 3 segundos.

Devemos referir que nos nossos testes admitimos que só teríamos memória disponível para 100.000 registos, pelo que mesmo no ficheiro mais pequeno gera alguns ficheiros temporários.

Após alguns testes, este algoritmo teve uma consequência um pouco desagradável: a fragmentação do disco. Devemos referir que após a execução sobre o ficheiro ***USA-road-d.USA*****.gr,** que ocupa cerca de 1.3GB

de espaço, o disco onde foram efectuados os testes apresentou uma taxa de fragmentação de 54%, o que é deveras elevado mas que deveríamos estar à espera, dado o algoritmo que é.

O nosso programa tem dois algoritmos de ordenação. Implementámos o quicksort para ordena o array na altura que separa o ficheiro original em ***n*** ficheiros mais pequenos e um mergesort alterado, na altura que se concatena um par de ficheiros mais pequenos num maior.

No nosso método para concatenar os ficheiros foi implementado um iterador para iterar sobre os ficheiros, retornando o próximo elemento a ser colocado no array para posteriormente ser realizado um dump num ficheiro temporário. Este iterador, foi implementado seguindo a ideia do exercício anterior, que se tornou uma grande mais valia para o nosso código tornando-o eficaz e elegante.

A implementação do quicksort tem em conta alguns problemas de performance, tais como:

- a execução da condição de paragem cada vez que é executada o método quicksort.
- Executando o quicksort da parte do array mais pequeno para tornar mais rápida e minimizar o uso do stack
- utilizando a mediana para redução dos piores casos na execução do método partition().

Data a extensão do código, o mesmo não será colocado neste relatório mas constará em anexo, com comentários javadoc para complementar informações constantes no relatório.

Bibliografia

- Folhas de Apoio da Professora Cátia Vaz, nomeadamente os capítulos que dizem respeito ao Heap e ao QuickSort.
- http://ww0.java4.datastructures.net/
- http://en.wikipedia.org/wiki/Linked_list
- http://jga.sourceforge.net/docs/javadoc/net/sf/jga/util/Iterables.html
- http://java.sun.com/javase/6/docs/api/
- http://www.javaperformancetuning.com/news/roundup032.shtml