Segunda Lista de Exercícios de Programação I

Data: 25/06/2019
Prof. Flávio Varejão

Aluno:…….....……..

1. Construa um tipo de dados para representar uma árvore binária genérica.
2. Faça uma função chamada *inserir* para inserir um elemento na árvore. Considere que a árvore deve ser uma árvore binária de busca. Isto é, caso o elemento já exista na árvore, a inserção não deve ser realizada. Caso contrário, o elemento deve ser inserido de tal forma que todos os elementos da árvore menores que ele fiquem a sua esquerda e todos os elementos maiores que ele fiquem a sua direita.
3. Faça uma função chamada *criar* para construir uma árvore binária de busca a partir de uma lista de elementos.
4. Faça uma função chamada *buscar* para retornar uma lista ordenada a partir de uma árvore binária de busca. Para retornar uma lista ordenada, basta percorrer a árvore em profundidade.
5. Faça o tipo de dados árvore ser uma instância da classe Functor. Lembre que a definição da classe Functor é a seguinte:

```
class Functor f where
        fmap :: (a → b) → f a → f b
```

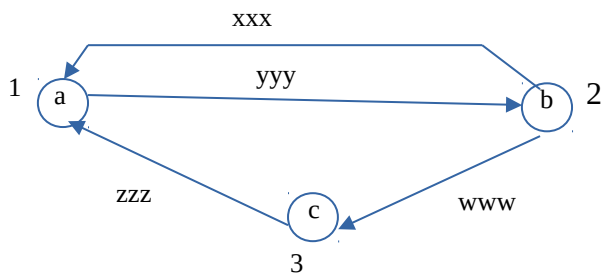
   Para facilitar veja o exemplo de como criar uma instância da classe Functor para o tipo Maybe:

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f Nothing = Nothing
```

6. Considere a seguinte definição de grafo:

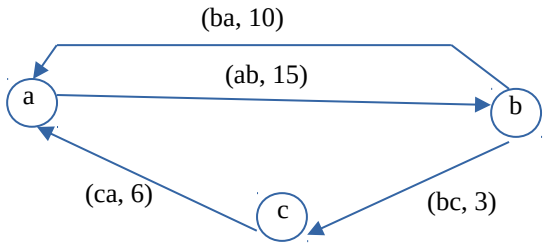
```
type No = Int
type Adj b = [(b, No)]
type Contexto a b = (Adj b, No, a, Adj b)
data Grafo a b = Vazio |
            Conectado (Contexto a b) (Grafo a b)
            deriving Show
```

   Use essas definições para representar o grafo *g*:



7. Construa um tipo de dados para representar uma fila genérica. Use o tipo *Maybe* para garantir que as operações da fila retornem valores consistentes em todas os contextos possíveis.
8. Crie uma função chamada *vazia* para produzir uma fila vazia.
9. Faça uma função chamada *inserir* para inserir um elemento no final da fila.

10. Faça uma função chamada *remover* para remover o primeiro elemento da fila.
11. Faça uma função chamada *pegar* para retornar o primeiro elemento sem removê-lo da fila.
12. Construa um tipo de dados Digrafo para representar um grafo direcionado *G = (V, A)*, tal como o do exemplo abaixo, usando uma representação de lista de adjacências. Note que o grafo *G* do exemplo é composto pelo conjunto de vértices *V = {"a", "b", "c"}* e pelo conjunto de arestas *A = {"ab", "ba", "bc",* "ca"}. Os valores associados as arestas indicam o custo (distância) relacionado àquela aresta. Por exemplo, a aresta *ba* tem custo *10*.



13. Elabore uma função chamada *criar* para construir um digrafo usando a representação definida na primeira questão a partir de uma tupla composta por uma lista de vértices e uma lista de tuplas compostas pelas arestas e seus respectivos custos. Por exemplo, para o digrafo da primeira questão, a entrada da função seria: (["a", "b", "c"], [("ab", 15), ("ba", 10), ("bc", 3), ("ca", 6)]).
14. Faça uma função chamada *buscar* para realizar uma busca em profundidade em um digrafo do tipo definido na primeira questão retornando uma tupla com uma lista indicando a sequência de arestas visitadas e a soma total dos custos das arestas percorridas. Leve em conta que o digrafo pode ser desconexo. A escolha do vértice inicial e da ordem que serão exploradas as arestas que partem de um vértice é deixada em aberto (a sua implementação definirá). Um possível retorno da aplicação dessa função sobre o digrafo da primeira questão é: (["ab", "ba", "bc", "ca"], 34). O algoritmo para realização de busca em profundidade a partir de um nó é:

```
Procedimento PROF(vertice v)
        visitado(v) <- sim
        Para cada vertice w adjacente a v faça
                Se visitado(w) = não então
                        PROF(w)
        fim-para
Fim
```

15. Mostre como o tipo de dados lista é definido como uma instância da classe Monad. Lembre que a definição da classe Monad é a seguinte:

```
class Monad m where
   return :: a -> m a

   (>>=) :: m a -> (a -> m b) -> m b

   (>>) :: m a -> m b -> m b
   x >> y = x >>= \_ -> y

   fail :: String -> m a
   fail msg = error msg
```

Para facilitar veja o exemplo de como criar uma instância da classe Monad para o tipo Maybe:

```
instance Monad Maybe where
      return x = Just x
      Nothing >>= f = Nothing
      Just x >>= f  = f x
      fail _ = Nothing
```