Segundo Trabalho de Programação Funcional – Modalidade AARE

Data de entrega: 06/10/2020 (até 23:55)

Valor: 20 pontos

Obs1: Esse trabalho pode ser desenvolvido em **Dupla** ou **Individual,** mas a arguição na apresentação e a respectiva nota será individual.

Obs2: A nota será dada pela nota de desenvolvimento (de 0 a 20) multiplicada pela nota de arguição (de 0 a 1), onde o aluno deverá demonstrar completo entendimento do trabalho desenvolvido.

PARTE A – Algoritmos de ordenação

Para os exercícios de ordenação, considere as 14 listas a seguir como exemplos para teste dos diversos algoritmos:

```
l1=[1..1000]

l2=[1000,999..1]

l3=l1++[0]

l4=[0]++l2

l5=l1++[0]++l2

l6=l2++[0]++l1

l7=l2++[0]++l2

x1=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

x2=[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]

x3=[11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10]

x4=[10,9,8,7,6,5,4,3,2,1,20,19,18,17,16,15,14,13,12,11]

x5=[11,12,13,14,15,5,4,3,2,1,16,17,18,19,20,10,9,8,7,6]

x6=[1,12,3,14,5,15,4,13,2,11,6,17,8,19,20,10,9,18,7,16]

x7 = [20,8,2,11,13,3,7,18,14,4,16,10,15,1,9,17,19,12,5,6]
```

Exercício 1) Em relação à implementações dos algoritmos seleção e inserção vistas em aula:

a) Refaça a implementação do algoritmo Seleção usando funções genéricas (foldr ou foldr1) .
b) Refaça a implementação do algoritmo Inserção usando funções genéricas (foldr ou foldr1).
c) Refaça a implementação do algoritmo quicksort usando funções genéricas (filter): modifique a função principal do algoritmo (quicksort) para que seja utilizada a função de alta ordem (genérica) **filter** para a obtenção dos elementos maiores e menores do que o pivô a cada iteração.

Realizar execuções comparativas nas listas dadas como exemplo entre os algoritmos das variações e do algoritmo original para avaliar se o resultado está correto (não precisa enviar esses resultados).

Exercício 2) Em relação ao algoritmo de ordenação **Bolha** visto em sala e laboratório:

- Variação 1: parada do algoritmo é antecipada quando uma iteração de trocas é finalizada sem que nenhuma troca efetiva seja realizada na iteração completa.

- Variação 2: a cada iteração de trocas, a avaliação é realizada desconsiderando-se o último elemento cuja posição foi fixada. Ou seja, a lista a ser ordenada é diminuída.

- Variação 3: compões as duas versões anteriores: faz parada antecipada e diminui o tamanho da lista a ser ordenada a cada iteração.

- Repetir as variações anteriores, incluindo um contador de comparações elementares realizadas durante a ordenação.

- Realizar execuções comparativas nas listas dadas como exemplo entre os algoritmos das variações e do algoritmo original para avaliar: 1) o número de comparações em cada execução; 2) o tempo de execução (apenas verifiquem se existe uma mudança aparente de tempo de processamento)

- Eleja a melhor variação do algoritmo Bolha, justificando sua escolha.

Exercício 3) Em relação ao algoritmo de ordenação **Selecao** visto em sala de aula e laboratório:

- Variação 1: apenas modifique a função principal do algoritmo (seleção) para que não seja utilizada uma concatenação de listas a cada iteração, no passo [x] ++ *selecao (remove x xs).* Ao invés disso, utilize o operador de construção de listas ":", como em a:b.

- Variação2: a partir da Variação 1, refazer o código para que a busca pelo menor elemento (função mínimo) e a eliminação desse menor elemento da lista a ser ordenada (função remove) ocorra numa mesma função (remove_menor), sem a necessidade de se percorrer a lista duas vezes a cada iteração (uma para remover e outra para remover o menor elemento).

- Repetir a Variação 2, incluindo um contador de comparações elementares realizadas durante a ordenação.

- Realizar execuções comparativas nas listas dadas como exemplo entre os algoritmos da Variação 2 e do algoritmo original para avaliar: 1) o número de comparações em cada execução; 2) o tempo de execução (apenas verifiquem se existe uma mudança aparente de tempo de processamento).

- Eleja a melhor variação do algoritmo Seleção, justificando sua escolha.

Exercício 4) Em relação ao algoritmo de ordenação **quicksort** visto em sala de aula e laboratório:

- Variação 1: modifique o algoritmo original para que ao invés dos elementos maiores e menores serem encontrados com buscas independentes, que seja elaborada e utilizada a função **divide** que percorre a lista uma única vez, retornando os elementos menores em uma lista e os elementos menores em outra.

EX: > divide 'j' "pindamonhangaba" Resposta: ("idahagaba","pnmonn")

- Variação 2: modifique a variação 1 para que o elemento pivô seja obtido a partir da análise dos 3 primeiros elementos da lista, sendo que o pivô será o elemento mediano entre eles. Exemplo: na lista [3, 9, 4, 7, 8, 1, 2], os elementos 3, 9 e 4 seriam analisados e o pivô escolhido seria 4. Caso a lista a ser analisada tenha menos que 3 elementos, o pivô é sempre o primeiro.

- Repetir as variações 1 e 2, incluindo um contador de comparações elementares realizadas durante a ordenação.

- Realizar execuções comparativas nas listas dadas como exemplo entre os algoritmos da Variação 1, 2 e do algoritmo original para avaliar: 1) o número de comparações em cada execução; 2) o tempo de execução (apenas verifiquem se existe uma mudança aparente de tempo de processamento).

- Eleja a melhor variação do algoritmo QuickSort, justificando sua escolha.

Exercício 5)

- Pesquise e implemente sua própria versão em Haskell dos algoritmos mergesort e *bucketsort*

- Repetir os algoritmos anteriores, incluindo um contador de comparações elementares realizadas durante a ordenação.

- Realizar execuções comparativas nas listas dadas como exemplo entre os algoritmos selecionados como as melhores versões do Bolha, Inserção, Seleção e Quicksort, além do algoritmo Mergesort, para avaliar: 1) o número de comparações em cada execução; 2) o tempo de execução (apenas verifiquem se existe uma mudança aparente de tempo de processamento).

PARTE B – Tipos Algébricos e Ávores

Exercício 6) Dada a definição de tipos abaixo, similar à vista em aula.

```
data Exp a =
  Val a -- um numero
| Add (Exp a) (Exp a) -- soma de duas expressoes
| Sub (Exp a) (Exp a) - subtração

avalia :: Num a => Exp a -> a
avalia (Val x) = x
avalia (Add exp1 exp2) = (avalia exp1) + (avalia exp2)
avalia (Sub exp1 exp2) = (avalia exp1) - (avalia exp2)
```

a) Expanda as definições acima para que além das operações soma e subtração, sejam incluídas expressões usando as operações multiplicação e potenciação.

b) Avalie as expressões abaixo, primeiro declarando-as de acordo com a sintaxe do tipo algébrico e depois executando a função avalia sobre essas declarações:
(3+12)*(15-5)^(1*3) e -((6+8-5+1)*(2+6^2))

Exercício 7) Defina um tipo algébrico Hora para modelar os horários do dia usando a convenção AM (**a**ntes do meio-dia) e PM (a**p**ós o meio-dia), que deve armazenar as horas e os minutos. Os valores do tipo Hora são escritos na forma `(AM x y)` ou `(PM x y)`, sendo `x` e `y` valores do tipo Int.

Ex: (AM 3 23) para o horário 3:23 e (PM 2 48) para o horário 14:28

a) Implemente a função horasDecorridas, que recebe um horário do dia, definido pelo tipo algébrico Hora, e essa função deve retornar a quantidade de horas decorridas no dia até o horário passado como argumento. Exemplo de avaliação:

```
*Main> horasDecorridas (AM 4 34)
4
*Main> horasDecorridas (PM 4 34)
16
```

De forma similar, implemente as funções minutosDecorridos e segundosDecorridos que devem retornar a quantidade de minutos/segundos decorridas no dia até o horário. Ex:

```
*Main> minutosDecorridos (PM 4 34)
994

*Main> minutosDecorridos (AM 4 34)
274

*Main> segundosDecorridos (AM 4 34)
14280

*Main> segundosDecorridos (PM 4 34)
59640
```

b) Modifique as funções do item a, para que sejam rejeitados valores de hora que estejam fora do intervalo de 0 a 11 e valores de minuto fora do intervalo de 0 a 59.

c) Faça as alterações necessárias no código para que os testes abaixo possam ser realizados com sucesso no console do GHC

```
> (AM 10 3) == (AM 10 3)
True
> (AM 10 3) == (AM 10 3)
True
> (PM 10 3) > (PM 5 3)
True
```

```
> (AM 10 3) < (AM 5 3)
False
> (AM 10 3) > (PM 5 3)
False
```

Exercício 8) Defina um Tipo Algébrico para registrar mensagens de texto recebidas, podendo ser provenientes de LinkedIn, WhatsApp ou Facebook. O objetivo é realizar a união destas mensagens de forma a agilizar a interação das pessoas com os seus contatos profissionais. Cada mensagem deve ter: identificador do remetente, curta mensagem (até 100 caracteres), data, hora e proveniência, como nos exemplos:

Contato: nome "Augusto Costa", Msg:"A apresentação de 13h foi cancelada", Data: (13 08 2017), Hora: (AM 10 30), App: WhatsApp

Contato: fone "3232-3232", Msg: "Reunião 14h - Lições para Trabalho em Equipe", Data: (11 08 2017), Hora: (AM 08 50), App: WhatsApp

Contato: nome "Ana Paula Silva", Msg: "Banco de Talentos da USP", Data: (15 08 2017), Hora: (PM 06 57), App: LinkedIn

Contato: nome "Augusto Costa", Msg: "Veja o link aaa.bbb.com", Data: (14 08 2017), Hora: (AM 11 10), App: Facebook

Crie uma pequena base de mensagens com pelo menos 30 ocorrências em dois dias consecutivos de Set/2020 para que seja possível testar as funções definidas abaixo, repetindo contatos em diferentes Apps (como o exemplo do Augusto Costa acima)

Analise os componentes de uma mensagem e defina outros tipos algébricos se necessário.

b) Crie uma estrutura (lista) para armazenar em conjunto os dados das mensagens de texto como as definidas no item a. Crie uma função que ordena a lista de mensagens pelo campo Contato, usando o método de ordenação bolha. Considere que nessa ordenação os números telefônicos devem vir antes de nomes.

c) Crie uma função que ordena a lista de mensagens pela data e hora usando dois o método de ordenação quicksort.

d) Defina uma função para consultar as últimas 2 mensagens de um contato qualquer (se houver), postadas em qualquer das redes em questão. Essa função deve receber

como entrada: o contato e uma lista de mensagens, que pode estar desordenada. A nova função de consulta deve usar a função do item c para ordenar a lista de entrada.

Exercício 9) Considere o tipo algébrico ArvBinInt visto em sala para representar árvores binárias que armazenam números inteiros. Elabore as funções a seguir que manipulam árvores binárias:

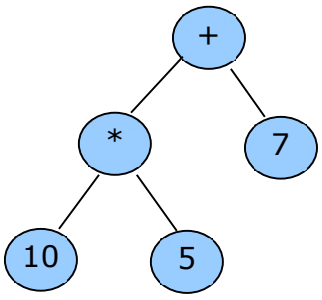
- **a)** *`internos`*: recebe uma árvore binária e devolve uma listagem com todos os nós internos (não folhas) existentes na árvore.
- **b)** *`somaNos`*: somar os valores de todos os elementos da árvore binária
- **c)** *`pertence`*: recebe um valor inteiro e verifica se esse valor é igual a algum dos elementos da árvore binária.

Exercício 10) Uma árvore binária pode ser utilizada para armazenar expressões aritméticas. Para isso definimos o tipo ArvBinEA em que uma árvore pode ser vazia, conter um valor numérico ou conter uma expressão com um operador e outras duas expressões:

```
data ArvBinEA a = Vazia |
                  Folha a |
                  NoEA (Char, ArvBinEA a, ArvBinEA a)
                     deriving (Show)
```

A partir deste novo tipo podemos definir a expressão:

```
ea::ArvBinEA Float
ea = NoEA ('+', NoEA ('*', Folha 10, Folha 5), Folha 7)
```



Faça uma função que receba uma árvore binária de expressão aritmética e retorne o resultado da expressão. Por exemplo, dada a árvore `ea` desenhada acima, a função deve devolver o valor 57.