MAC323 – Algoritmos e estruturas de dados II

BCC – Primeiro semestre de 2015

Prova 1 – 16/4/2015

Nome do aluno: _____ Turma: _____

Assinatura: _____

Nº USP: _____ Curso: _____

Instruções:

1. Não destaque as folhas deste caderno.
2. A prova pode ser feita a lápis. Cuidado com a legibilidade.
3. Há 4 questões na prova. Verifique antes de começar a prova se o seu caderno de questões está completo.
4. Não é permitido o uso de aparelhos eletrônicos (computadores, tablets, celulares, etc).
5. Não é permitido o uso de folhas avulsas para rascunho.
6. Não é necessário apagar rascunhos no caderno de soluções.
7. Não é permitido consultar anotações ou colegas.
8. **Duração da prova:** 1 hora e 50 minutos

Não escreva nesta parte da folha

Questão	Nota
1	
2	
3	
4	
Total	

BOA SORTE!

Questão 1 (valor: 3.0)

(i) Descreva, precisamente, um algoritmo para transformar expressões aritméticas infixas envolvendo inteiros e as operações binárias `+` e `*` em expressões equivalentes pós-fixas. Por exemplo, seu algoritmo deve transformar `( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )` na expressão `1 2 3 + 4 5 * * +`. Neste item, você deve supor que a expressão de entrada é completamente 'parentizada'.

(ii) Repita o item (i), mas agora supondo que a expressão não é necessariamente totalmente parentizada. Você deve levar em conta que `*` tem precedência sobre `+`.

Você pode achar interessante considerar o seguinte programa, visto em sala:

```
public class Evaluate {
    public static void main(String[] args) {
        Stack<String> ops  = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();

        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if      (s.equals("("))               ;
            else if (s.equals("+"))    ops.push(s);
            else if (s.equals("*"))    ops.push(s);
            else if (s.equals(")")) {
                String op = ops.pop();
                double v = vals.pop();
                if      (op.equals("+"))    v = vals.pop() + v;
                else if (op.equals("*"))    v = vals.pop() * v;
                vals.push(v);
            }
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

Questão 2 (valor: 3.0)

Descreva, precisamente, como podemos implementar a seguinte API, de forma que as operações tenham complexidade de tempo constante:

```
public class Stack<Item extends Comparable<Item>> implements Iterable<Item>
---------------------------------------------------------------------------
        Stack()                 create an empty stack
   void push(Item item)         add an item
   Item pop()                   remove the most recently added item
boolean isEmpty()               is the stack empty?
    int size()                  number of items in the stack
   Item min()                   returns the minimum in the stack
```

[*Sugestão.* Use duas pilhas. Na segunda pilha, armazene itens m_k, onde m_k é o item mínimo dos k itens no fundo da pilha.]

Questão 3 (valor: 3.0)

Considere os seguintes trechos de código, da implementação de `ResizingArrayStack`:

```
// resize the underlying array holding the elements
private void resize(int capacity) {
   Item[] temp = (Item[]) new Object[capacity];
   for (int i = 0; i < N; i++) {
       temp[i] = a[i];
   }
   a = temp;
}

public Item pop() {
   Item item = a[--N];
   a[N] = null;                              // to avoid loitering
   // shrink size of array if necessary
   if (N > 0 && N == a.length/4) resize(a.length/2);
   return item;
}
```

(i) Suponha que, em um dado momento, um `ResizingArrayStack st` é tal que `st.length` é 4 e que $N = 4$. Suponha que os elementos de `st` são removidos sucessivamente até que `st` fique vazio (isto é, executamos `st.pop()` $N = 4$ vezes). Diga quantos acessos a vetor ocorrem nesse processo. [*Observação.* Devem ser considerados tanto o vetor `a[]` como o `temp[]`.] Diga em detalhe como você chegou em sua resposta.

(ii) Repita o item (i) para $N = 16$ (supondo também que `st.length` é igual a 16). Como podemos generalizar para valores de N da forma 4^k (k inteiro positivo)?

(iii) O que se entende pela afirmação 'as operações em um `ResizingArrayStack` têm custo amortizado $O(1)$'?

Questão 4 (valor: 3.0)

Considere o seguinte fragmento da implementação de `QuickUnionUF`:

```
public class QuickUnionUF {
    private int[] id;    // id[i] = parent of i

    public QuickUnionUF(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
    }

    public int find(int p) {
        while (p != id[p]) p = id[p];
        return p;
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;
        id[rootP] = rootQ;
        count--;
    }
}
```

(i) Suponha que $N = 10$ e que executamos, sucessivamente, a união dos pares (p, q) abaixo, na ordem dada, usando `QuickUnionUF`: $(9, 0)$, $(3, 4)$, $(5, 8)$, $(7, 2)$, $(2, 1)$, $(5, 7)$, $(0, 3)$, $(4, 2)$. Diga qual é o conteúdo inicial de `id[]` e qual é o conteúdo desse vetor após cada execução de `union()` (você deverá dizer o conteúdo desse vetor 9 vezes).

(ii) Faça um diagrama da floresta representada pelo vetor `id[]` *final* do item (i).

(iii) Considere o seguinte vetor `id[]`:

```
  i     0 1 2 3 4 5 6 7 8 9
---------------------------
id[i]   1 1 3 1 5 6 1 3 4 5
```

Faça um diagrama da floresta representada por este vetor `id[]`.

(iv) Dê uma seqüência de pares (p, q) (como no item (i)) que gera a floresta pedida em (iii). Se tal seqüência não existir, você deve provar tal fato.

(v) (valor: 1 ponto de bônus) Repita (iv), supondo que estamos usando `WeightedQuickUnionUF`.