Algoritmos e Complexidade – 5 de Fevereiro de 2015 – Duração: 2h30m

Parte A

1. Considere o programa para calcular o quociente `q` entre inteiros `b` e `a`

   - Apresente as condições de verificação necessárias à prova da sua correcção parcial utilizando o símbolo `I` no lugar do invariante de ciclo.
   - Anote o programa com um invariante e um variante de ciclo que lhe permitam provar a correcção total do algoritmo.

```
// a = a0 > 0 && b = b0 >= 0
int q, m;
q=0; m=0;
while (b>=m+a) {
    // I
    m = m+a; q=q+1
}
// q*a0 + (b0-m) = b0 && 0 <= m <= b0
```

2. Considere o tipo definido à direita para representar uma árvore binária de inteiros.

```
typedef struct node {
   int value;
   struct node *left, *right;
} Node, *BTree;
```

   Uma heap é uma árvore binária em que a raiz é menor ou igual a todos os outros nodos e as subárvores são ainda heaps.

   (a) Defina uma função `int heapOK (BTree a)` que testa se uma dada árvore é uma heap (retornando `1` em caso afirmativo e `0` caso não seja).

   Note que se pretende apenas testar a relação entre os valores armazenados (e não a forma da árvore).

   Assegure-se que a função que definiu executa, no pior caso em tempo linear no número de elementos da árvore, e no melhor caso em tempo constante.

   (b) Considere as seguintes funções que preenchem um array com os elementos de uma heap por ordem crescente, retornando o número de elementos preenchidos.

```
int toArray1 (BTree b, int v[]){
   int i=0;
   while (b!=NULL) {
      v[i++] = b->value;
      b = removeRoot (b);
   }
   return i;
}
```

```
int toArray2 (Btree b, int v[]) {
  int l=0,r=0;
  if (b!=NULL) {
     v[0] = b->value;
     l = toArray2 (b->left,v+1);
     r = toArray2 (b->right,v+l+1);
     merge (v+1,l,r);
     }
  return (1+l+r);
  }
}
```

   Admita que as árvores em causa estão balanceadas.

   i. Determine a complexidade assimptótica da função `toArray1`, assumindo que a função `removeRoot` tem uma complexidade de `log N` sempre que é invocada com uma árvore com `N` elementos.

   ii. Apresente uma recorrência que traduza a complexidade da função `toArray2` assumindo que a função `merge` tem uma complexidade `N` quando invocada com um array de tamanho `N = l+r`. Apresente ainda uma solução para essa recorrência.

3. Considere o seguinte tipo para armazenar as arestas de um grafo pesado com `N` vértices.

```
typedef struct aresta {
    int destino;
    int peso;
    struct aresta *prox;
} *Grafo [N];
```

Defina em C uma a função `int pesoC (Grafo g, int V[], int k)` que calcula o custo do caminho com `k` vértices (armazenados no vector `V`) no grafo `g`. Assuma que os vértices do vector correspondem realmente a um caminho, i.e., que existe uma aresta entre cada par de valores consecutivos do array.

4. Relembre o algoritmo de Dijkstra para o cálculo de caminhos de menor peso em grafos pesado. Apresente a evolução desse algoritmo quando é invocado, a partir do vértice 2, sobre o grafo com 5 vértices representado na seguinte matriz. O elemento da linha x coluna y representa o peso da aresta com origem x e destino y. Um peso `0` marca a inexistência de aresta. Na sua resposta deve apresentar os vários estados da orla bem como dos vectores de antecessores e pesos.

	0	**1**	**2**	**3**	**4**
0	0	0	1	2	0
1	7	0	0	0	2
2	9	1	0	0	0
3	1	1	5	0	0
4	3	0	0	1	0

Parte B

1. Relembre o algoritmo de procura numa arvore binaria de procura.

```
Node *search (BTree a, int x){
   while ((a!=NULL) && (a->value != x))
      if (a->value > x) a = a->right;
         else a = a->left;
      return a;
}
```

Assumindo que se trata de uma arvore balanceada, determine o numero medio de nodos consultados numa arvore (i.e., o numero de itreracoes do ciclo while) com N nodos. Assuma que, caso o elemento a procurar exista na arvore, ele pode estar com igual probabilidade em qualquer ponto da arvore.

2. Considere que se implementa uma tabela de Hash com tratamento de colisões por open addressing e usando arrays dinâmicos.

Considere ainda que as inserções e consultas de uma nova chave executam em tempo constante (`1`), desde que não haja realocação do array.

Esta assumpção só é válida quando o factor de carga da tabela não é superior a 50%. Por isso, quando esse factor atinge esse valor, i.e., a tabela está 50% ocupada, o array é duplicado e essa duplicação tem um custo adicional igual ao tamanho do array.

Mostre que o custo amortizado da inserção é constante.