PROVA DE RECUPERAÇÃO DE ESTRUTURAS DE DADOS
BCC, 1o. SEMESTRE DE 2012

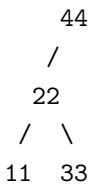
Instruções:

1. Não destaque as folhas do caderno de soluções.
2. A prova pode ser feita a lápis. Cuidado com a legibilidade.
3. Não é permitido o uso de folhas avulsas para rascunho.
4. Não é necessário apagar rascunhos no caderno de soluções.
5. Asserções imprecisas valem pouco. Justifique suas asserções (dentro do razoável).

1. [3 pontos]
   (*i*) Defina precisamente o Problema da Cobertura Exata (PCE) e o Problema da Cobertura Generalizado (PCG); não deixe de descrever exatamente as restrições sobre as colunas primárias e sobre as colunas secundárias.
   (*ii*) Lembre-se de que você resolveu o k-SAT em seu EP2. Defina precisamente o k-SAT.
   (*iii*) Descreva precisamente como você reduziu o k-SAT para um problema de cobertura como em (*i*) acima. Caso você tenha usado um problema de cobertura diferente, não deixe de definir esse problema precisamente.
   (*iv*) O seguinte trecho vem do enunciado do EP2:

   > Fixe k, n e m. Podemos gerar instâncias aleatórias ϕ de k-SAT com n variáveis e m cláusulas escolhendo m das $2^k\binom{n}{k}$ possíveis cláusulas uniformemente a acaso, com reposição. Acredita-se que, para todo k, existe uma constante c_k com a seguinte propriedade: *Seja $\varepsilon > 0$ uma constante e suponha que $n \to \infty$. Se $m \geq (c_k + \varepsilon)n$, então ϕ é quase-certamente não-satisfatível, enquanto que se $m \leq (c_k - \varepsilon)n$, então ϕ é quase-certamente satisfatível.* Você deverá usar seu programa para estimar esta constante c_k (supondo que ela existe). Faça isso para $2 \leq k \leq 5$ e para alguns valores maiores de k (como 10 e outros). Verifique o quão rapidamente seu programa consegue resolver k-SAT para entradas ϕ aleatórias, prestando atenção especialmente nos casos em que m está em torno de $c_k n$.

   Descreva os resultados que você obteve.
2. [3 pontos] Suponha que inserimos, nesta ordem, as 4 chaves `44 11 33 22` em uma ABB aleatória inicialmente vazia. Suponha que a árvore resultante foi a árvore a seguir:



Date: Versão de 26 de julho de 2012.

(*i*) Mostre como se deu essa evolução da ABB vazia até esta ABB resultante (desenhe as várias ABBs intermediárias; dê todas as possibilidades).

(*ii*) Determine a probabilidade de termos essa ABB resultante com base no algoritmo de inserção em ABBs aleatorizadas (as rotinas são dadas a seguir). Mostre como você chegou a seu resultado.

(*iii*) Suponha agora que estamos usando uma ABB *padrão* (*não-aleatorizada*) e que inserimos as 4 chaves `11 22 33 44` em uma tal ABB inicialmente vazia em alguma ordem. Podemos fazer isso de 4! jeitos diferentes, pois há 4! ordenações daquelas chaves. Diga quais dessas 4! ordenações resultam na árvore dada no diagrama acima. Explique sua resposta.

(*iv*) Explique a relação entre suas respostas para (*ii*) e (*iii*) acima.

```
link insertT(link h, Item item)
  { Key v = key(item);
    if (h == z) return NEW(item, z, z, 1);
    if (less(v, key(h->item))) { h->l = insertT(h->l, item); h = rotR(h); }
                          else { h->r = insertT(h->r, item); h = rotL(h); }
    return h;
  }

link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if (rand() < RAND_MAX/(h->N+1)) return insertT(h, item);
    if (less(v, t)) h->l = insertR(h->l, item);
               else h->r = insertR(h->r, item);
    (h->N)++;
    return h;
  }

void STinsert(Item item)
  { head = insertR(head, item); }
```

3. [4 pontos] Esta questão trata de árvores rubro-negra esquerdistas (ARNEs), usadas para implementar tabelas de símbolos. Por simplicidade, suponha que os itens e chaves são inteiros.

(*i*) Suponha que inserimos, nesta ordem, as chaves `22 77 11 88 44 55 99 33 66` em uma ARNE inicialmente vazia. Desenhe as 9 ARNEs que resultam das 9 inserções acima. Não desenhe árvores 2–3; desenhe ARNEs e execute as inserções seguindo rigorosamente a rotina de inserção em ARNEs.

(*ii*) Diga exatamente quantas vezes a comparação entre chaves `less()` (veja o código abaixo) é executada em cada uma das 9 inserção em (*i*).

(*iii*) Suponha que temos duas ARNEs, com raízes `h1` e `h2`. Descreva cuidadosamente uma implementação de uma função de protótipo `void print_inter(link h1, link h2)`

que imprime os *elementos comuns* a `h1` e `h2`, isto é, os objetos de tipo `item` que ocorrem em ambas as ARNEs. Por simplicidade, suponha que os objetos do tipo `item` são simplesmente inteiros `int`. Sua implementação deve ter complexidade de tempo ótima.

(*iv*) Qual é a complexidade de tempo da função `print_inter()` que você descreveu em (*iii*)? Qual é sua complexidade de espaço? Sua resposta deve ser em função do número de elementos N_1 e N_2 nas duas ARNEs envolvidas. Suponha que $N_1 = N_2 = 10^6$ (e os objetos armazenados são `int`s). Dê uma estimativa do espaço *adicional* que sua função usa quando executada (sua resposta deve ser em `bytes`). Como seria sua resposta se a exigência fosse que a complexidade de *espaço* deve ser ótima? Justifique suas respostas cuidadosamente.

As duas rotinas principais envolvidas na inserção em ARNEs são dadas a seguir.

```
link LLRBinsert(link h, Item item)
  { Key v = key(item);

    /* Insert a new node at the bottom*/
    if (h == z) return NEW(item, z, z, 1, 1);

    /* Comparacoes entre chaves */
    if (less(v, key(h->item))) hl = LLRBinsert(hl, item);
                          else hr = LLRBinsert(hr, item);

    /* Enforce left-leaning condition */
    if (hr->red && !hl->red) h = rotL(h);
    if (hl->red && hll->red) h = rotR(h);
    if (hl->red && hr->red) colorFlip(h);

    return fixNr(h);
  }

void STinsert(Item item)
  { head = LLRBinsert(head, item); head->red = 0; }
```