Métodos de Programação II

LESI / LMCC

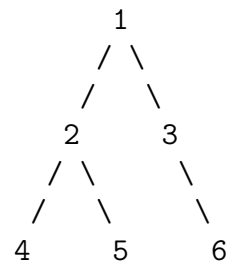
28 de Junho 2004

Questão 1 Considere duas sequência de chaves armazenadas em dois arrays globais, `pre` e `in`, que correspondem, respectivamente, ao resultado das travessias *preorder* e *inorder* de uma árvore binária. Assumindo que não há chaves repetidas, podemos afirmar que as duas sequências determinam univocamente a árvore binária. O primeiro elemento em `pre` é a raiz, e as duas sub-árvores podem ser determinadas usando o facto de que a raiz parte o array `in` de forma que todos os elementos da sub-árvore esquerda estão em `in` antes da raiz e todos os elementos da sub-árvore direita estão em `in` depois da raiz. Considere as seguintes declarações:

```
typedef struct node {
  int chave;
  struct node *esq;
  struct node *dir;
} Node;

int pre[MAX], in[MAX];
```

1. (a) Desenhe a árvore definida pelas seguintes sequências de chaves:
   `pre` = 1, 2, 4, 5, 3, 6 e `in` = 4, 2, 5, 1, 3, 6.
   Resposta:



   (b) Complete a definição da função `buildtree` que constroi a árvore binária a partir das sequências `in` e `pre`, de forma a que se a árvore binária tem n elementos, `buildtree(0, n-1, 0, n-1)` devolve o apontador para a raiz da árvore.

```
Node * buildtree(int i1, int i2, int p1, int p2)
{ Node *t=NULL;
  int k;
                      // assume-se que o input está correcto
  if (i1>i2 && p1>p2)
     ...A... ;
  else {
        t = malloc(sizeof(Node));
        t->chave = ...B... ;
        for( k=i1; ...C... ; k++);
        t->esq = ...D... ;
        t->dir = ...E... ;
        return t;
```

Nome:_____ **Número:**_____ **Curso:**_____

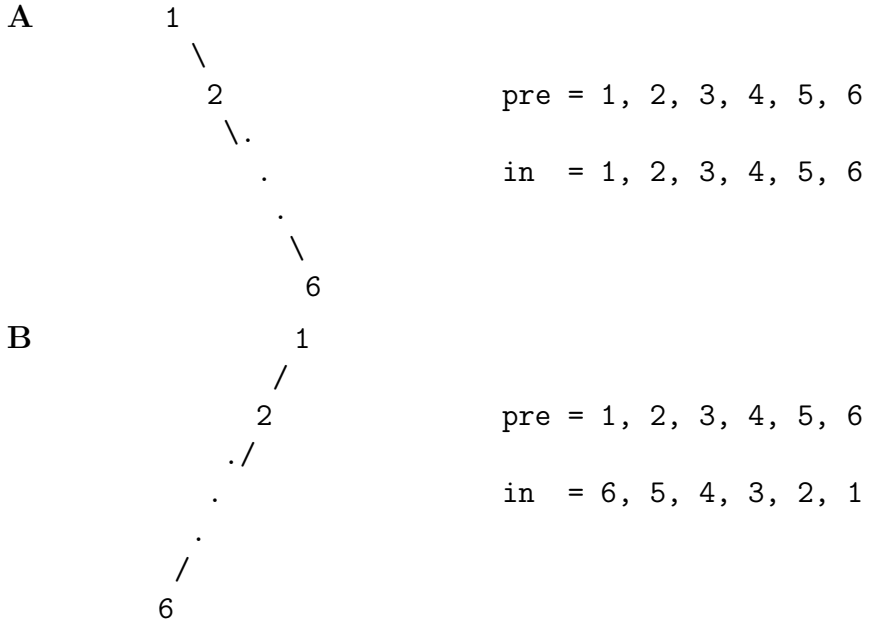
```
        }
    }
```

Resposta:

```
...A... = return t
...B... = pre[p1]
...C... = pre[p1] != in[k]
...D... = buildtree(i1,k-1,p1+1,p1+k)
...E... = buildtree(k+1,i2,p1+k+1,p2)
```

2. Escreva o conteúdo dos arrays `pre` e `in` para os dois casos extremos que correspondem a árvores de altura máxima com as chaves 1, 2, 3, 4, 5 e 6. Estas situações correspondem ao melhor e ao pior caso de execução da função `buildtree`. Justifique esta afirmação.

   Resposta:
   Um exemplo dos dois casos extremos pode ser:



   O tempo de execução da função `buildtree` depende do tempo que demora a encontrar o valor `pre[p1]` no array `in` (ver ciclo `for`).

   Na situação **A**, `pre[p1]` vai estar sempre no início do array `in` (tempo execução constante). **A** é portanto o *melhor caso*.

   Na situação **B**, `pre[p1]` vai estar sempre no fim do array `in` (tempo execução linear em relação ao tamanho do array). **B** é portanto o *pior caso*.

3. Escreva equações de recorrência que exprimem o tempo de execução de `buildtree` no caso genérico, no melhor caso, e no pior caso. Analise o tempo de execução para cada caso.

Nome:_____ **Número:**_____ **Curso:**_____

Resposta:
A medida do tamanho do input do algoritmo `buildtree` é dada pelo número de elementos da árvore (= tamanho de qualquer dos arrays).

Equações de recorrência para o caso genérico:

$$T(n) = \begin{cases} \Theta(1) & \text{se n} = 0 \\ \Theta(e+1) + T(e) + T(n-e-1) & \text{se } n > 0 \end{cases}$$

com $0 \leq e < n$, representando e o número de elementos da sub-árvore esquerda.

No *melhor caso* (caso **A**) as sub-árvores esquerdas são sempre vazias ($e = 0$). As equações de recorrência para o melhor caso ficam então:

$$T(n) = \begin{cases} \Theta(1) & \text{se n} = 0 \\ \Theta(1) + T(0) + T(n-1) & \text{se } n > 0 \end{cases}$$

com solução $T(n) = \Theta(n)$.

No *pior caso* (caso **B**) as sub-árvores direitas são sempre vazias ($e = n-1$). As equações de recorrência para o pior caso ficam então:

$$T(n) = \begin{cases} \Theta(1) & \text{se n} = 0 \\ \Theta(n) + T(n-1) + T(0) & \text{se } n > 0 \end{cases}$$

com solução $T(n) = \Theta(n^2)$.

Nome:_____ **Número:**_____ **Curso:**_____

Métodos de Programação II

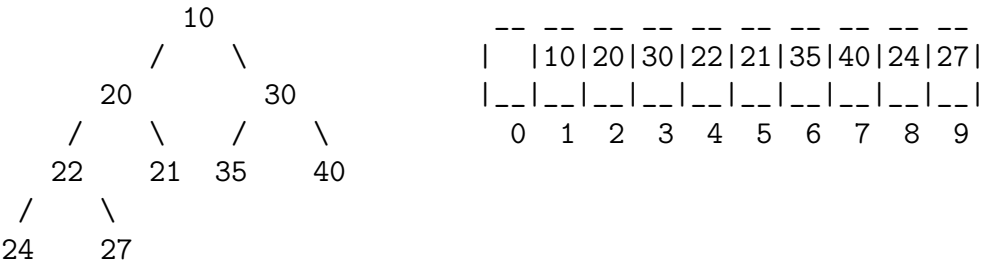
LESI / LMCC

28 de Junho 2004

Questão 2 [*Estruturas de Dados*] Uma *heap* é uma árvore binária que verifica a seguinte propriedade: o conteúdo de cada nó é menor ou igual que o conteúdo dos seus descendentes (não havendo, no entanto, qualquer relação de ordem entre os conteúdos das duas sub-árvores de um mesmo nó).

Uma implementação possível destas estruturas, vantajosa em diversas situações, recorre apenas a um vector, contendo, a partir do índice 1, os elementos de uma *heap* dispostos por níveis, da esquerda para a direita. As heaps assim representadas têm uma forma particular: o nível $k+1$ só pode ter elementos a partir do momento em que o nível k está totalmente preenchido, e cada nível começa a ser preenchido da esquerda para a direita, não havendo nós vazios pelo meio.

Exemplo de uma *heap* e sua representação em vector:



1. Se se acrescentar um elemento no fim do vector, isso corresponderá a acrescentar um folha à árvore, o que pode não resultar numa *heap* uma vez que o conteúdo dessa folha não é necessariamente superior ou igual ao dos seus ascendentes.

   A seguinte função recebe um vector contendo uma heap entre as posições 1 e $n-1$ e um elemento qualquer na posição n. O vector contém, no fim da sua execução, uma *heap* entre as posições 1 e n.

```
void insere_ultimo(int h[], int n)
{
  int i = n, p;
  while (i!=1) {
    p = ...A...;
    if (...B...) {
      swap(h, p, i);
    }
    ...C...;
  }
}
```

   (`swap` troca o conteúdo de duas posições de um vector).

   Complete a definição da função com as expressões A, B, e C. Sugestão: comece por estabelecer uma relação entre os índices do vector correspondentes a um nó e aos seus dois filhos,

Nome:_____ **Número:**_____ **Curso:**_____

Resposta:

```
void insere_ultimo(int h[], int n)
{
  int i = n, p;
  while (i!=1) {
    p = i/2                              /* N.B. divisão inteira... */;
    if (h[p] > h[i]) {
      swap(h, p, i);
    }
    i = p;
  }
}
```

2. Efectue a análise assimptótica de melhor e pior caso da função da alínea anterior. Se não tiver resolvido essa alínea, assuma que A, B, C contêm apenas operações de tempo constante.

   Resposta:
   Dentro do ciclo há apenas operações de tempo constante, pelo que resta contar o número de iterações deste ciclo, que é naturalmente logarítmico em N, uma vez que a variável i, inicializada com N, é em cada iteração dividida por 2. Se $N = 10$, i tomará sucessivamente os valores 10, 5, 2, e 1 (paragem). Outra forma de entender isto é a seguinte: i indexa sempre um elemento colocado num nível da árvore, subindo de nível em cada iteração. Ora, uma destas árvores com N elementos tem necessariamente $\lg N$ níveis.

   Temos assim, para o pior e o melhor caso, $T(N) = \Theta(\lg N)$. A execução ou não da instrução `swap` não altera o tempo assimptoticamente.

3. Estabeleça um invariante para o ciclo `while` da função e prove a correcção do algoritmo.

   Resposta:
   É possível estabelecer o invariante em termos do vector, ou em termos da *heap* por ele representada:

   "No início da iteração i, o conteúdo de cada nó da árvore é superior ou igual ao conteúdo do seu pai, excepto possivelmente para o nó indexado por i".

   ou

   "No início da iteração i, o conteúdo de cada posição k do vector é superior ou igual ao conteúdo da posição $k/2$, excepto possivelmente para $k = i$".

   Inicialização $i = N$; trivial

   Preservação a instrução `swap` repõe a ordem correcta entre os conteúdos de i e $i/2$, possivelmente violando a ordem entre $i/2$ e $i/4$. Sendo depois i actualizado para $i/2$, é reposto o invariante.

   Terminação O ciclo termina necessariamente, sendo o valor de i em cada iteração menor do que na anterior. No fim da execução tem-se $i = 1$, correspondente à raíz da árvore, que não tem pai. Sendo assim, o invariante garante que neste ponto o vector representa uma *heap*.

Nome:_____ **Número:**_____ **Curso:**_____

4. Altere a função anterior por forma a melhorar o funcionamento do algoritmo no melhor caso. Repita a análise assimptótica do tempo de execução para esta nova versão do algoritmo.

   Resposta:
   Quando não é necessário trocar um elemento com o seu ascendente na árvore, então seguramente não será também necessário continuar esse processo nos níveis superiores, que há partida contituíam uma *heap*. Sendo assim, pode-se optimizar a função, saindo do ciclo atempadamente. Por exemplo:

```
void insere_ultimo(int h[], int n)
{
  stop = 0;
  int i = n, p;
  while (i!=1 && !stop) {
    p = i/2                          /* N.B. divisão inteira... */;
    if (h[p] > h[i]) {
      swap(h, p, i);
    }
    else stop = 1;
    i = p;
  }
}
```

Métodos de Programação II

LESI / LMCC

28 de Junho 2004

Questão 3 [*Algoritmos de Grafos*] Considere o algoritmo de Warshall que calcula o fecho transitivo sobre um grafo não pesado.

```
void fechoTrans(int A[][],int F[][], int n) {
    int i,j,k;

    for (i=1;i<=n;i++)
       for(j=1;j<=n;j++)
          F[i][j]=A[i][j];

    for (k=1 ; k<=n; k++)
        for (i=1 ; i<=n ; i++)
            for (j=1 ; j<=n ; j++)
                if (F[i][k] && F[k][j]) F[i][j] = 1;
 }
```

1. (a) Explique o funcionamento do algoritmo.
   (b) Enuncie (sem provar) um invariante de ciclo para o ciclo mais exterior.
   (c) Identifique e explique a representação do grafo utilizada neste algoritmo.
   (d) Explique por que razão esta representação é utilizada neste caso particular.

   A resposta a esta pergunta está nos apontamentos da disciplina, Parte II

2. Apresente uma adaptação deste algoritmo que calcule os caminhos mais curtos entre todos os pares de nós num grafo pesado. Admita que o grafo é fornecido utilizando o mesmo tipo de representação, com o valor `MAXINT` a representar a inexistência de um arco. A matriz resultante deverá conter a distância correspondente ao caminho mais curto entre cada par de nós, caso exista, e `MAXINT` caso não exista.

   ```
   typedef int Weight;

   void distancias(Weight weights[][], Weight dists[][], int n);
   ```

   Diga ainda, justificando, se este algoritmo apresenta vantagens em relação à aplicação exaustiva do algoritmo de Dijkstra, para o mesmo fim.

   A resposta à primeira parte desta pergunta está nos apontamentos da disciplina, Parte II.

   Quanto aos benefícios em termos de eficiência, este algoritmo executa em $\Theta(|V|^2)$.

   O algoritmo de Dijkstra executa em $O(|V|^2 + |E|)$. A aplicação deste algoritmo a todos os pares de nós, implicaria $\Theta(|V|^2)$ invocações.

   Assim, o ganho em eficiência é significativo. De uma forma simplificada podemos dizer que se reduz a taxa de crescimento do tempo de execução de $O(|V|^4)$ para $O(|V|^3)$.

Nome:_____ **Número:**_____ **Curso:**_____

Métodos de Programação II

LESI / LMCC

28 de Junho 2004

Questão 4 [*Problemas NP-completos*] Considere o seguinte problema de decisão:

- É dado um conjunto O de n objectos;
- cada objecto i tem dimensão s_i (um número inteiro positivo);
- é dado ainda um número inteiro positivo K;
- pretende-se decidir se existe ou não um sub-conjunto de objectos de O cuja soma de dimensões seja exactamente igual a K.

Prove que se trata de um problema NP, isto é, prove que existe um algoritmo não-determinístico, limitado polinomialmente, que o resolve.

Resposta
Um algoritmo não-determinístico é constituído por duas partes distintas: a geração de uma solução proposta, e a verificação dessa solução proposta.
Antes do mais, é necessário definir o que é uma solução proposta no caso deste problema. Consideremos uma solução proposta como uma sequência de, no máximo, n números inteiros, com o mesmo tamanho de representação de n, ou seja, lista de bits de tamanho $\Theta(\log n * n)$.
A primeira fase do nosso algoritmo não determinístico, consiste em gerar uma tal sequência de bits aleatória.
A verificação da validade da solução consiste nos seguintes passos:

1. Verificar que a lista de bits contém a representação de l números inteiros positivos, com valores entre 1 e n.
2. Verificar se a soma das dimensões s_{l_i} dos objectos apontados por esses l números inteiros é exactamente igual a K.
3. Em caso afirmativo devolver "Sim"; caso contrário, devolver "Não".

Claramente, todas as fases deste algoritmo executam em tempo polinomial, donde este é um algoritmo não determinístico limitado polinomialmente. A simples existência deste algoritmo comprova que este problema de decisão pertence à Classe NP.

Nome:_____ **Número:**_____ **Curso:**_____