Lista 5 - PAA - Fábio Alves de Freitas

Anotações Heap de Fibonacci

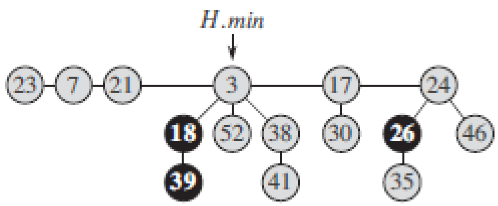
↳ *1 célula oculta*

1º) (Ref. 1522) Suponha que uma raiz x em um Heap de Fibonacci está marcada. Explique como x veio a ser uma raiz marcada. Argumente que não importa para a análise que x esteja marcado, mesmo que este não seja uma raiz que foi primeiro unida a outro nodo e então tenha perdido um filho.

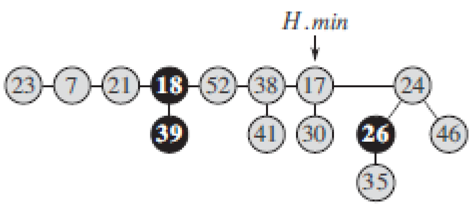
Resposta:

Digamos que em dado momento o nodo x está marcado e é um dos filhos de $H.min$. Se $H.min$ for removido pela função FIB-HEAP-EXTRACT-MIN, então todos os seus filhos se tornarão raizes e adiconalmente nodo x será uma raiz marcada.

> Nodo 18 é um filho do $H.min$



> Nodo 3, que é o $H.min$ atual, foi removido pela função FIB-HEAP-EXTRACT-MIN, tornando o nodo 18 uma raiz marcada



Um Nodo marcado se tornando uma raiz não implica em mais processamento para o algoritmo, pois o único momento que essa marcação é verificada é na linha 3 da função CASCADING-CUT, e que só é executada caso o nodo atual não for uma raiz (verificação da linha 2 do CASCADING-CUT).

2º) (Ref. 2526) Suponha que a regra CASCADING-CUT tenha sido generalizada para cortar um nodo x a partir de seu pai tão breve quanto este perca seu k-ésimo filho,

para algum inteiro constante k. Para que valores de k tem-se $D(n) = O(log_2 n)$?

Resposta:

Sabendo que x possa assumir qualquer nó, se um $heap$ de Fibonacci, $x.degree = m$, e x tem filhos $y_1, y_2, \ldots, y_m$, então $y_1.degree$ ≥ 0 e $y_i.degree \geq i - k$. Logo, se s_m apontar o menor número possível de nós em um nó de grau m, então temos $s_0 = 1$, $s_1 = 2, \ldots, s_{k-1} = k$ e em geral, $s_m = k + \sum_{i=0}^{m-k} s_i$. Assim, a diferença entre s_m e s_{m-1} é s_{m-k}.

Seja fm a sequência tal que $f_m = m + 1$ para $0 \leq m < k$ e $fm = f_{m-1} + f_{m-k}$ com $m \geq k$.

Se $F(x)$ é a função geradora para f_m, então temos $F(x) = \frac{1-x^k}{(1-x)(1-x-x^k)}$. Seja α alpha uma raiz de $x^k = x^{k-1} + 1$. Mostraremos por indução que $f_{m+k} \geq \alpha^m$. Para os casos básicos:

$$f_k = k + 1 \geq 1 = \alpha^U.$$

$$f_{k+1} = k + 3 \geq 1 = \alpha^1.$$

$$f_{k+1} = k + \frac{(k+1)(k+2)}{2} = k + k + 1 + \frac{k(k+1)}{2} \geq 2k + 1 + \alpha^{k-1} \geq \alpha^k.$$

Em geral, temos;

$$f_{m+k} = f_{m+k-1} + f_m \geq \alpha^{m-1} + \alpha^{m-k} = \alpha^{m-k}(\alpha^{k-1} + 1) = \alpha^m.$$

A seguir, mostramos que $f_{m+k} = k + \sum_{i=0}^{m} f_i$. O caso base é claro, uma vez que $f_k = f_0 + k = k + 1$. Para a etapa de indução, temos;

$$f_{m+k} = f_{m-1-k} + f_m = k \sum_{i=0}^{m-1} f_i + f_m = k + \sum_{i=0}^{m} f_i.$$

Observe que $s_i \geq f_i + k$ para $0 \leq i \leq k$. Novamente, por indução, para $m \geq k$ temos;

$$s_m = k + \sum_{i=0}^{m-k} s_i \geq k + \sum_{i=0}^{m-k} f_i + k \geq k + \sum_{i=0}^{m} f_i = f_{m+k}.$$

Então, em geral, $s_m \geq f_{m+k}$. Juntando tudo, temos;

$$tamanho(x) \geq s_m$$

$\geq k + \sum_{i=k}^{m} s_{i-k}$

$\geq k + \sum_{i=k}^{m} f_i$

$\geq f_{m+k}$

$\geq \alpha^m.$

Pegando logs em ambos os lados, temos;

$$log_\alpha n \geq m.$$

Em outras palavras, desde que α seja uma constante, temos um limite logarítmico no grau máximo.

3º) (Ref. 4556) O que acontece se a função vEB-TREE-INSERT for invocada com um elemento que já exista na árvore vEB? O que acontece se a função vEB-TREE-DELETE for invocada com um elemento que não exista na árvore vEB? Explique porque os procedimentos exibem os seus respectivos comportamentos. Mostre como modificar uma árvore vEB e sua operações de tal forma a seja possível checar em tempo constante se um dado elemento está presente na árvore.

Resposta:

Suponha que x já esteja em V e chamemos INSERT. Logo as linhas 1, 3, 6 ou 10 não serão satisfeitas, de modo a entrar no *else* na linha 9 todas as vezes até chegarmos ao caso base. Se x já estiver na árvore do caso-base, não mudaremos nada. Se x estiver armazenado em um atributo min de uma árvore vEB que não é o caso-base, entretanto, inseriremos uma duplicata dele em alguma árvore do caso-base. Agora suponha que chamemos DELETE quando x não estiver em V . Se houver apenas um único elemento no V , as linhas 1 a 3 o excluirão, independentemente de qual elemento seja. Para inserir o *else if* da linha 4, x não pode ser igual a 0 ou 1 e a árvore vEB deve ser de tamanho 2. Nesse caso, excluímos o elemento max, independentemente de qual seja. Como a chamada recursiva sempre nos coloca neste caso, sempre excluímos um elemento que não deveríamos. Para evitar esses problemas, mantenha e atualize o A da matriz auxiliar com os elementos de u. Defina A[i] = 0, se i não estiver na árvore e 1 se estiver. Como podemos realizar atualizações de tempo constantes neste array, isso não afetará o tempo de execução de nenhuma de nossas operações. Ao inserir x, verifique primeiro se A[x] = 0. Se não estiver, basta retornar. Se for, defina A[x] = 1 e prosseguir com a inserção conforme boa prática. Ao excluir x, verifique se A[x] = 1. Se não for, basta retornar. Se for, defina A[x] = 0 e prossiga com a exclusão conforme boa prática.

4º) (Ref. 5556) Suponha que ao invés de $\sqrt[\uparrow]{u}$ u clusters, cada qual com uni verso de tamanho $\sqrt[\downarrow]{u}$ u, seja construída uma árvore vEB que tenha $u^{1/k}$ clusters, cada qual com universo de tamanho $u^{1-1/k}$, com $k > 1$ como uma constante. Dado que as operações foram modificadas apropriada mente, qual o custo em tempo? Assuma que $u^{1/k}$ e $u^{1-1/k}$ sejam sempre inteiros.

Resposta:

De modo similar a análise (20.4), nós analizaremos:

$$T(u) \leq T(u^{1-1/k}) + T(u^{1/k}) + O(1)$$

Esta é uma boa alternativa para a análise, pois para muitas operações nós primeiro verificamos o *summary vEB tree*, que terá tamanho $u^{1/k}$ (o segundo termo). E então será possível ter de verificar uma árvore vEB no cluster, que terá tamanho u^{1-1/k} (primeiro termo). Nós definidos $T(2^m) = S(m)$, para que a equação torne-se

$$S(m) \leq S(m(1-1/k)) + S(m(1/k)) + O(1)$$

Se $k > 2$, então o primeiro termo domina, então pelo teorema mestre, teremos que $S(m) = O(log_2 m)$, isso significa que T será $O(log_2(log_2 u))$ da mesma forma que o caso original, onde temos raizes quadradas.

5º) (Ref. 2572) Escreva uma versão não recursiva da função FIND-SET com compressão de percurso (Conjuntos disjuntos).

Resposta:

Para implementar FIND-SET não recursivo, seja x o elemento que chamamos na execução dessa função. Crie uma lista encadeada A, que contem um ponteiro para x. Cada vez que colocamos mais um elemento na árvore, insere um ponteiro para esse elemento em A. Uma vez que a raiz r foi encontrada, use a lista encadeada para encontrar cada nodo no caminho da raiz até x e atualize seu pai para r.

6º) (Ref. 5572) Mostre que qualquer sequência de m operações de MAKE SET, FIND-SET e LINK, onde todas as operações de LINK aparecem antes das operações de FIND-SET, toma apenas um custo em tempo de $O(m)$ se for utilizado conjuntamente a compressão de percurso e a união por rank. O que acontece na mesma situação se apenas a heurística de compressão de percurso for aplicada?

Resposta:

Claramente cada operação MAKE-SET e LINK demandam apenas $O(1)$, então, suponha que n é o número de operações FIND-SET que ocorrem após o "make" e "link", então nós precisamo mostrar qu todos as operações FIND-SET demandam apenas $O(n)$.

Para fazer isto, nós vamos amortizar alguns custos de operação do FIND-SET nos custos de operação do MAKE-SET. Então, quando fizermos uma operação FIND-SET(x), teremos três possibilidades:

- Primeiro, nós poderíamos ter que x é o representante de seu próprio conjnto. Neste caso, ele claramente demanda um tempo constante para executar.
- Segundo, nós poderíamos ter que o caminho de x até o representante de seu conjunto já está comprimido, logo isso apenas demanda um único passo para encontrar seu representando. Neste cenário, o tempo requirido é constante.
- Terceiro, nós poderíamos ter que x não é o representante a seu caminho não está comprimido. Então, suponha que existem k nodos entre x e seu representante. O tempo para a operação deste FIND-SET será $O(k)$, mas consequentemente ele termina comprimindo os caminhos dos k nodos, logo nós utilizamos esse tempo extra de processamento de operação do MAKE-SET para esses k nodos cujos caminhos foram comprimidos. Qualquer chamada subsequente do FIND-SET demandará um tempo constante, logo nunca tentaríamos usar o trabalho desse montante de amortização duas vezes para um determinado nó.

✓ 0s conclusão: 19:02