Fundação Getulio Vargas
Estruturas de Dados e Algoritmos
Aluno: Marcelo Bianchi Barata Ribeiro
Prof. Jorge Poco

Homework 2

Due:: August 13, 2019

Homework 2

1 Red-Black Trees

I am attaching a binary tree source code (`bst-0.0.cpp`) with the methods insert, delete and print. Your job would be to implement a Red-Black Tree with the functions insert, remove and print.
To test your code you can follow the examples described in the document `anexo1.pdf`. In addition, you might be interested in the document `anexo2.pdf` for a more detailed description of this tree, there is also some Java code that might be useful.
Note, your code must be implemented in C++ and based in the BST class I'm providing you. Grading would be as follow:

(a) **(2.5pts)** insert

(b) **(2.5pts)** remove

An example of the main function is:

```
1  int main() {
2    // this constructor must call the function insert multiple times
3    // respecting the order
4    RBTree tree(41, 38, 31, 12, 19, 8);
5    tree.print();
6
7    // testing the remove function
8    tree.remove(8);
9    tree.print();
10 }
```

Resposta:
Ver arquivo **rbt.cpp**.

Como referências, usei:

- Livro "Introduction to Algorithms", capítulo 13 (CORMEN, Thomas et al), terceira edição.
- Slide L3 do curso de Standford que está sendo utilizado pelo professor.
  - Link: http://web.stanford.edu/class/archive/cs/cs161/cs161.1178/

Para os exemplos de teste, o algoritmo funcionou corretamente.

2 Radix Sort

(2pts) Your job is to implement the radix sort algorithm in Python. The following code is going to be used to test your implementation. You have to submit a notebook with your code.

```
def radix_sort(A, d, k):
  # A consists of n d-digit ints, with digits ranging 0 -> k-1
  #
  # implement your code here
  # return A_sorted


# Testing your function
A = [201, 10, 3, 100]
A_sorted = radix_sort(A, 3, 10)
print(A_sorted)
# output: [3, 10, 100, 201]
```

Resposta:

Ver arquivo: **radix_sort.ipynb**.

Como referências, usei:

- Livro "Introduction to Algorithms", capítulo 8.4 (CORMEN, Thomas et al).
- Slide L3 do curso de Standford que está sendo utilizado pelo professor.
  - Link: http://web.stanford.edu/class/archive/cs/cs161/cs161.1178/

Além do exemplo de teste, fiz mais alguns testes com números naturais. O radix sort faz recursão em cada dígito do conjunto de elementos da lista e chama bucket sort, que por sua vez cria uma chave-valor e usa o dígito como chave, chamando em seguida um stable sort em cada bucket gerado. Escolhi o insert sort dado que o foco não está na escolha do stable sort por algum critério específico. Nesse caso, o insertion sort ocupa menos linhas, tornando o código mais conciso de modo a visualizar mais rápido as outras funções.

3 Sorting in Place in Linear Time

(1.5pts) Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

(a) Give an algorithm that satisfies criteria 1 and 2 above.

(b) Give an algorithm that satisfies criteria 1 and 3 above.

(c) Give an algorithm that satisfies criteria 2 and 3 above.

(d) Can any of your sorting algorithms from parts(a)–(c) be used to sort n records with b-bit keys using radix sort in $O(bn)$ time? Explain how or why not.

(e) Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that the records can be sorted in place in $O(n+k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (Hint: How would you do it for $k = 3$?)

Resposta:

a)
Nesse caso, podemos aplicar **Counting sort**, dado que sabemos o número distinto de valores. Counting sort tem complexidade **O(n)** e é um stable sort, tal como demonstrado no livro.

b)
Nesse caso específico, **Quicksort** pode ser aplicado ao escolhermos x = 0 como pivô e usar **partition** em cima desse valor, dado que isso vai separar a lista, deixando todos os 0s do lado esquerdo (elementos i tal que $i \leq 0$) e todos os 1s do lado direito (elementos i tal que $i > 0$). Quicksort é sempre inplace. Sua complexidade será **O(n)** neste caso, dada a especificidade do problema, apesar de ser em geral O(nlogn).

c)
Insertion sort é inplace e também é conhecido como um stable sort. O algoritmo faz contínua comparação entre A[i] (cur_value) e A[j]. Consideremos $i < j$ e A[i] = A[j]. Nesse caso, A[i] virá primeiro e será encaixado iterando em ordem decrescente a j (chamemos cada elemento dessa iteração de j') até que $j' \leq i$, alterando A[i] para A[j'+1] (chamemos essa posição de A[k]). Quando for a vez de j passar por esse mesmo processo, será também deslocado para a esquerda, mas não poderá ultrapassar a nova posição de i (A[k]), dado que a condição para mudar a posição do algoritmo é:
while $j' \geq 0$ and $A[j'] > cur\_value$ (lembrando que agora cur_value = A[j]).

Como essa comparação com o A[k] não é satiseita, saímos do loop e A[j] estará à direita de A[k] (novo A[i]), garantindo assim que é um stable sort.

d)
Nesse caso, **Counting sort**. Sabemos que Counting sort possui running time de O(n). Para chaves b-bit com valores variando somente entre 0 e 1, teremos um running time de:
O(b(n+2)) = cb(n+2) (para algum $c > 0$)
= cbn + cb
= O(bn).

e)
O algoritmo inicia parecido com o Counting sort original (ver CORMEN, Thomas et al pg 195). Dessa vez não será um stable sort, mas será **inplace**.
Escrevendo em pseudo-código semelhante a python:

```
k = max(A)

def COUNTING_SORT_INPLACE(A,k):
1 B = [0] * k                    #cria array de 0s de tamanho k
2 for i in range(length(A)):
3 —B[A[i]] = B[A[i]] + 1         # B[i] agora contém o número de elementos iguais a i.
4 count = 0                      # a partir daqui, é diferente de counting sort original
5 for i in range(0,k):
6 — for j in range(1,B[i]):
7 ——— A[count] = i
8 ——— count += 1
```

A complexidade do novo algoritmo será **O(n+k)**. O ideal será usá-lo quando k = O(n), pois nesse caso, a complexidade do algoritmo seria Θ(n) (ver CORMEN, Thomas et al pg 196).

4 Alternative Quicksort Analysis

(1.5pts) An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to QUICKSORT, rather than on the number of comparisons performed.

(a) Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = I\{i\text{th smallest element is chosen as the pivot}\}$. What is $E[X_i]$?

(b) Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] \tag{1}$$

(c) Show that equation 1 simplifies to

$$E[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1} E[T(q)] + \Theta(n) \tag{2}$$

(d) Show that

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \tag{3}$$

(Hint: Split the summation into two parts, one for $k = 1, 2, \ldots, \lceil n/2 \rceil - 1$ and one for $k = \lceil n/2 \rceil, \ldots, n-1$.)

(e) Using the bound from equation 3, show that the recurrence in equation 2 has the solution $E[T(n)] = \Theta(n \lg n)$. (Hint: Show, by substitution, that $E[T(n)] \leq an \log n - bn$ for some positive constants a and b.)

Respostas:

a)
Tal como explicado no livro "Introduction to Algorithms", capítulo 5 (CORMEN, Thomas et al, pgs 118, 179), dado um espaço amostral S e um evento A, a Indicadora da variável aleatória I{A} associada ao evento A será definida por:

$$I\{A\} = \begin{cases} 1 & \text{se A ocorrer} \\ 0 & \text{se A não ocorrer} \end{cases}$$

Neste caso, queremos encontrar o valor esperado do *i-ésimo menor elemento a ser selecionado como pivô* (A_i). O espaço amostral é dado por S = $\{A_1, A_2, ..., A_n\}$, sendo a probabilidade de $A_i = \frac{1}{n}, \forall 1 \leq i \leq n$, pois as probabilidades de cada evento são todas iguais e, somadas, totalizam 1. Podemos definir uma variável aleatória Xi associada ao evento A_i. Essa variável conta o número de vezes que o pivô 'i' será selecionado ao fazer a seleção, que será 1 em caso afirmativo, ou 0 caso contrário. Escrevemos então:

$$X_i = \begin{cases} 1 & \text{se } A_i \text{ ocorrer} \\ 0 & \text{se } A_i \text{ não ocorrer} \end{cases}$$

O número esperado de vezes que A_i será selecionado como pivô é simplesmente o valor esperado da variável indicadora Xi, dado que cada elemento tem a mesma probabilidade de acontecer:

$$\begin{aligned} E[X_i] &= E[I\{A_i\}] \\ &= 1 \cdot Pr\{A_i\} + 0 \cdot Pr\{A_i\} \\ &= 1 \cdot \frac{1}{n} + 0 \cdot \frac{n-1}{n} \\ &= \frac{1}{n} \end{aligned}$$

Portanto, o número esperado de vezes que A_i será selecionado como pivô é $\frac{1}{n}$

b)
Primeiro, definindo T(n):
Relembrando o algoritmo quicksort:
QUICKSORT(A, p, r)
1. if p < r:
2. —- q = PARTITION(A, p, r)
3. —- QUICKSORT(A, p, q-1)
4. —- QUICKSORT(A, q+1, r)
Sendo que a chamada inicial da função é: QUICKSORT(A,1,A.length).
Não precisamos entrar em muitos detalhes sobre o PARTITION. Essencialmente, será definido um pivô "q" por meio do qual o array A será dividido entre os elementos menores do que 'q' (elementos q-1) e os maiores do que 'q' (elementos n-q), formando duas sublistas onde se dará nova recursão em cada uma delas. Tal tarefa de particionamento tem custo linear, $\Theta(n)$
Portanto, a recursão de quicksort é:

$$T(n) = T(q-1) + T(n-q) + \Theta(n) \tag{4}$$

A partir daqui, sabemos uma das principais propriedades da expectância, a de linearidade. Basta aplicar sobre todos os eventos X_i (definido na proposição 'a') para chegarmos ao resultado:

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] \tag{5}$$

c)

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right]$$

$$= \sum_{q=1}^{n} E[X_q(T(q-1) + T(n-q) + \Theta(n))]$$

$$= \sum_{q=1}^{n} \frac{(T(q-1) + T(n-q) + \Theta(n))}{n} \text{ Usando o que foi demonstrado na letra 'a'}$$

$$= \Theta(n) + \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-1))$$

$$= \Theta(n) + \frac{1}{n}\left(\sum_{q=1}^{n} T(q-1) + \sum_{q=1}^{n} T(n-q)\right)$$

$$= \Theta(n) + \frac{1}{n}\left(\sum_{q=1}^{n} T(q-1) + \sum_{q=1}^{n} T(q-1)\right)$$

$$= \Theta(n) + \frac{1}{n}\sum_{q=1}^{n} 2 \cdot T(q-1)$$

$$= \Theta(n) + \frac{2}{n}\sum_{q=1}^{n} T(q-1)$$

$$= \Theta(n) + \frac{2}{n}\sum_{q=0}^{n-1} T(q).$$

d) Resolvendo por integração, ao considerarmos $f(k) = k \lg k$ como função contínua, temos que $f'(k) = \lg k + 1$. Ao fazer integração por partes, temos a função F cuja derivação leva a $k \lg k$:

$$\frac{1}{\lg 2}(\frac{k^2}{2} \ln k - \frac{k^2}{4}).$$

Usando os limites e fazendo subtração, obtemos $\frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} - 1$. Dado o intervalo {2,n-1}, f é uma derivada positiva sobre todo esse intervalo. Como estamos comparando com uma função contínua, nosso resultado nunca poderá ser maior do que a aproximação. Logo:

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} - 1$$

$$\leq \frac{n^2 \lg n}{2} - \frac{n^2}{8}, \text{dado que } \ln 2 > 1/2$$

e)
Utilizando o método de substituição, estipulamos que $T(q) \leq q \lg(q) - bq$ e assumimos por indução que a propriedade se mantém. Pela proposição "c", temos que:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \frac{2}{n}\sum_{q=0}^{n-1}\mathrm{E}[T(q)] + \Theta(n) \\
&\leq \frac{2}{n}\sum_{q=0}^{n-1} aq\lg q - bq + \Theta(n) \quad \text{Aplicando substituição.} \\
&= \frac{2}{n}\sum_{q=1}^{n-1} aq\lg q - bq + \Theta(n) \\
&\leq \frac{2}{n}(\frac{1}{2}an^2\lg n - \frac{a}{8}n^2 - \frac{bn^2}{2}) + \Theta(n) \text{ Aplicando o resultado de "d".} \\
&= an\lg n - \frac{an}{4} - \frac{2}{n}\frac{bn^2}{2} + \Theta(n) \\
&= an\lg n - bn - \frac{an}{4} + \Theta(n) \\
&\leq an\lg n - bn, \ (\exists \text{a}{>}0 \text{ e b}{>}0 \text{ que satisfazem a desigualdade, tal como sugerido pelo enunciado})
\end{aligned}
$$

Logo, o passo indutivo está comprovado e podemos dizer que:
$E[T(n)] = \Theta(n\lg n)$