Fundação Getulio Vargas
Estruturas de Dados e Algoritmos
Prof. Jorge Poco

Homework 2
Due:: August 13, 2019

Aluno: Franklin Alves de Oliveira

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:

```
int main() {
  // this constructor must call the function insert multiple times
  // respecting the order
  RBTree tree(41, 38, 31, 12, 19, 8);
  tree.print();

  // testing the remove function
  tree.remove(8);
  tree.print();
}
```

OBS: Não consegui fazer a tempo.

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.

```python
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]
```

Solução:

Vide o arquivo q2-RadixSort.ipynb. Nele consta uma implementação do Radix Sort, com Counting Sort como *stable sorting algorithm*.

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.

Solução: Counting Sort. Tem complexidade $O(n)$ e é um algoritmo *stable*.

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

Solução: Quick Sort, com *Partition*. Por exemplo, podemos selecionar um elemento $x = 0$ como pivô. Então, dividimos o vetor em duas partições: Uma com elementos ≤ 0 e outra com elementos > 0 (todos os 1's). Então, cada um dos vetores é ordenado.

Esse algoritmo é *inplace* e tem complexidade $O(n)$, conforme requerido.

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

Solução: Insertion Sort. Como sabemos (Rerefência: Cormen, T. H.; *Introduction to Algorithms*, 3rd Ed.), esse algoritmo é *stable* e *inplace*. Além disso, não usa nenhum espaço adicional de memória em relação ao vetor original.

(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.

Solução: Sim. Poderíamos usar o Counting Sort com esse propósito. Sabemos que esse algoritmo tem complexidade $O(n)$. Para valores com chave $b-$bits, com cada *bit* variando entre 0 e 1, podemos ordenar o vetor em um tempo de ordem $O(b(n+2)) = O(bn)$.

(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 $k = 3$?)

Solução: O algoritmo Counting Sort Modificado para esse item será dado a seguir (em Python).

```
1   def Modified_Counting_Sort(A, k):
2       B = [0]*k  # inicializa um novo vetor B
3
4       # Aumenta o elemento B[A[i]] em 1
5       for i in range(len(A)):
6           B[A[i]] = B[A[i]] + 1
7
8       y = 0
9       for i in range(k):
10          for j in range(1,C[i]):
11              y += 1
12              A[y] = i   # Armazena os elementos cujos valores sao i
```

O algoritmo modificado é *inplace*, porém não é *stable*. Além disso, sua complexidade é de $O(n+k)$.

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 $E[X_i]$?

Solução:

Considerando que o pivô é um elemento do vetor escolhido ao acaso, se o vetor tem n elementos e a chance de qualquer elemento ser selecionado é a mesma para os demais (isto é, nenhum elemento tem maior probabilidade de ser escolhido em relação aos demais), temos:

Se p_i é a probabilidade de se escolher o elemento i do vetor, temos que:

- $p_i \geq 0, \ \forall \ i.$
- $\sum_{i=1}^{n} p_i = 1$

Do segundo ponto, como todos os elementos têm a mesma chance de ser escolhido, vamos considerar $p_i = \overline{p}$. Assim, $\sum_{i=1}^{n} p_i = n \cdot \overline{p} = 1 \Rightarrow \overline{p} = \frac{1}{n}$.

Ainda, considerando a variável aleatória $X_i = I\{i-\text{ésimo menor elemento é escolhido como pivô}\}$, como cada elemento é equiprovável, segue que:

$$E(X_i) = p_i \cdot I_i = p_i \cdot 1 = \overline{p} = \frac{1}{n}$$

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

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

Solução:

Tome o seguinte pseudocódigo do Quicksort:

```
1  def quicksort(A,d,k):
2        if d < k:
3            q = partition(A,d,k)
4            quicksort(A,d,q-1)
5            quicksort(A,q+1,k)
```

Nesse algoritmo, é definido um pivô (que estamos chamando de p no código acima), que será tomado como referência para particionar o vetor original, isto é, dividimos o vetor original em dois novos vetores (um com $q-1$ elementos menores que q e outro com $n-q$ elementos maiores que q). Então, chamamos o quicksort recursivamente em cada um desses novos vetores. O algoritmo de partição (*partition*), tem complexidade $\Theta(n)$. Assim, o tempo de execução do *quicksort* é dado por:

$$T(n) = T(q-1) + T(n-q) + \Theta(n)$$

Logo, considerando a variável aleatória X_i definida no item anterior e aplicando o operador esperança, temos:

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

(c) Show that equation 1 simplifies to

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

Solução:

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

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

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

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

(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 $k = \lceil n/2 \rceil, \ldots, n-1$.)

Solução:

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{\lceil n/2 \rceil}^{n-1} k \log k \leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k =$$

$$= \log\left(\frac{n}{2}\right) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n + \sum_{\lceil n/2 \rceil}^{n-1} k = \log n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{\lceil n/2 \rceil}^{n-1} k =$$

$$= \log n \cdot \left(\frac{n(n-1)}{2}\right) - \left(\frac{\lceil n/2 \rceil (\lceil n/2 \rceil - 1)}{2}\right) \leq \log n \cdot \left(\frac{n^2 - 2n}{2}\right) - \left(\frac{n^2 - 1}{2}\right) \leq$$

$$\leq \frac{n^2 \log n}{2} - \frac{n^2}{8}$$

Se tomarmos $N \in \mathbb{N}$ grande o suficiente, a desigualdade acima é válida para todo $n \geq N$. Nesse caso, a desigualdade é válida para todo $n \geq 2$.

(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.)

Solução:

Suponha, por indução, que $T(q) \leq q \lg(q) + \Theta(n)$. Das equações 2 e 3, temos:

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

Aplicando o método da substituição...

$$\mathrm{E}[T(n)] \leq \frac{2}{n} \sum_{q=2}^{n-1} q \lg q + \frac{2}{n} \Theta(n) + \Theta(n) \leq \frac{2}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2\right) + \Theta(n)$$

Da equação 3, temos:

$$\mathrm{E}[T(n)] \leq n \lg n - \frac{1}{4} n + \Theta(n) = n \lg n + \Theta(n)$$

Como queríamos demonstrar.