

**Algoritmos Avançados**

2.º Semestre 2015/2016 Alameda

Prof. Alexandre Francisco

Prof. Luís Russo

## Relatório do Projeto

Search Trees e Longest Substring

**69439** Marta Nascimento

**76346** José Amaral

Parte 1 – Search Trees

# Introdução

A primeira parte do projeto tem como objetivo a implementação das operações **tsearch**, **tfind** e **tdelete** para as estruturas de dados Treap. Nesta estrutura cada chave fica associada a uma prioridade gerada aleatoriamente. Esta prioridade é usada para manter a estrutura da árvore uma vez que o seu valor num nó tem que ser superior às prioridades dos seus nós filhos.

Esta estrutura foi implementada nos seguintes passos:

1. Implementou-se as 3 operações para uma estrutura de dados mais simples, ou seja, para árvores de procura binária sem prioridades:
   1. tsearch – procura por um elemento com uma chave específica na árvore. Se não encontrar cria e adiciona um novo elemento com a referida chave na árvore e retorna-o. Caso contrário, retorna logo o elemento encontrado.
   2. tfind – idêntico ao tsearch mas caso não encontre o elemento retorna NULL. Ou seja, não são criados novos elementos. Caso encontre o elemento, retorna-o.
   3. tdelete – remove um elemento da árvore, caso exista, e retorna o seu pai.
2. Acrescentaram-se as prioridades aleatórias à estrutura.
3. Implementaram-se as rotações para a esquerda e direita de acordo com o valor das prioridades para que se mantenha a estrutura da árvore.

# Resultados Experimentais

Foram realizadas 12 experiências para cada função em que o resultado apresentado representa a média em 500 iterações. Cada experiência é realizada para N elementos, com chaves e prioridades obtidas aleatoriamente.

Todos os testes foram realizados numa máquina com processador 2.7 GHz Intel Core i5.

### Análise Temporal

### **Função tsearch**

|  |  |  |
| --- | --- | --- |
| N | Project Time [ms] | Glibc Time [ms] |
| 500 | 0.000352 | 0.000299 |
| 1000 | 0.000375 | 0.000327 |
| 5000 | 0.000469 | 0.000425 |
| 10000 | 0.000547 | 0.000495 |
| 15000 | 0.000593 | 0.000548 |
| 20000 | 0.000633 | 0.000586 |
| 50000 | 0.000794 | 0.000837 |
| 100000 | 0.001021 | 0.001318 |
| 200000 | 0.001355 | 0.002007 |
| 500000 | 0.001983 | 0.002423 |
| 750000 | 0.002482 | 0.002623 |
| 1000000 | 0.002308 | 0.002672 |

### **Função tdelete**

Foram primeiramente adicionados aleatoriamente 2N elementos à árvore e posteriormente procedeu-se à remoção da primeira metade (e.g. adicionaram-se 500 elementos e removeram-se 250).

|  |  |  |
| --- | --- | --- |
| N | Project Time [ms] | Glibc Time [ms] |
| 250 | 0.000735 | 0.000345 |
| 500 | 0.000802 | 0.000399 |
| 2500 | 0.000954 | 0.000547 |
| 5000 | 0.001062 | 0.000651 |
| 7500 | 0.001142 | 0.000768 |
| 10000 | 0.001191 | 0.000949 |
| 25000 | 0.001426 | 0.001169 |
| 50000 | 0.001954 | 0.001847 |
| 100000 | 0.003573 | 0.002539 |
| 250000 | 0.003866 | 0.003005 |
| 375000 | 0.003853 | 0.003084 |
| 500000 | 0.003793 | 0.003081 |

### **Função tfind**

Foram primeiramente adicionados aleatoriamente N elementos à árvore e posteriormente procedeu-se à procura desses mesmos elementos.

|  |  |  |
| --- | --- | --- |
| N | Project Time [ms] | Glibc Time [ms] |
| 500 | 0.000234 | 0.000294 |
| 1000 | 0.000266 | 0.000331 |
| 5000 | 0.000373 | 0.000459 |
| 10000 | 0.00045 | 0.000572 |
| 15000 | 0.00053 | 0.000666 |
| 20000 | 0.000575 | 0.000782 |
| 50000 | 0.000724 | 0.001137 |
| 100000 | 0.001123 | 0.001461 |
| 200000 | 0.001389 | 0.001888 |
| 500000 | 0.002763 | 0.002381 |
| 750000 | 0.002858 | 0.002518 |
| 1000000 | 0.002869 | 0.002553 |

### Conclusões

Com base nas experiências realizadas é possível constatar que as três operações implementadas, tsearch, tfind e tdelete, têm o comportamento esperado e executam em tempo aproximado *O*(log N).

Para o número médio de rotações, para as funções de tsearch e tdelete, obtiveram-se os valores médios de 1.9723 e 1.9149 respetivamente, comprovando assim o valor teórico esperado de duas rotações.

Parte 2 – Longest substrings

# Introdução

A segunda parte do projeto tem como objetivo implementar um algoritmo que calcula o tamanho das maiores substrings que existem em pelo menos d strings diferentes, onde d varia entre 2 e k, sendo k o número total de strings.

Este algoritmo é composto por duas fases:

* 1ª Fase – Construção da árvore de sufixos generalizada para as k strings. Isto implica implementar o algoritmo de Ukkonen, que permite realizar esta construção em tempo linear.
* 2ª Fase – Percorrer a árvore de sufixos criada na primeira fase para determinar as maiores substrings existentes. Isto é realizado com recurso a uma DFS para realizar a união das listas dos identificadores das strings presentes em cada elemento da árvore. Durante a DFS é também guardado num array de tamanho k – 1 o maior tamanho, na posição d. No final este array poderá estar inconsistente e por isso é necessário percorrê-lo (do fim para o início) para atualizar as substrings maiores que existam em k menores. Ou seja, se uma substring de tamanho 4 existe em três strings (d = 3) diferentes, então existe também para valores de d menores (d = 2 e d = 1). Logo, se o maior tamanho em duas strings for inferior ao das três strings, então terá que ser atualizado.

# Resultados Experimentais

Foram realizadas 16 experiências para o algoritmo de Ukkonen em que o resultado apresentado representa a média em 1000 iterações. Cada experiência é realizada para K = 5 Strings com tamanho N.

Todos os testes foram realizados numa máquina com processador 2.7 GHz Intel Core i5.

### Análise Temporal

|  |  |  |
| --- | --- | --- |
| N | Project Time [ms] | Expected Time *O* (KN) |
| 50 | 0.039469 | 0.25 |
| 100 | 0.082651 | 0.5 |
| 150 | 0.123587 | 0.75 |
| 200 | 0.169557 | 1 |
| 250 | 0.21067 | 1.25 |
| 300 | 0.259377 | 1.5 |
| 350 | 0.306394 | 1.75 |
| 400 | 0.353895 | 2 |
| 450 | 0.398577 | 2.25 |
| 500 | 0.449596 | 2.5 |
| 1000 | 1.020192 | 5 |
| 2500 | 2.957694 | 12.5 |
| 5000 | 6.652066 | 25 |
| 7500 | 10.896436 | 37.5 |
| 10000 | 15.937874 | 50 |
| 20000 | 45.36639 | 100 |

### Análise Espacial

|  |  |  |
| --- | --- | --- |
| N | Project Space [KB] | Expected Space *O* (N) |
| 50 | 29.125 | 50 |
| 100 | 58.046875 | 100 |
| 150 | 86.6875 | 150 |
| 200 | 114.9765625 | 200 |
| 250 | 143.8984375 | 250 |
| 300 | 172.8203125 | 300 |
| 350 | 200.546875 | 350 |
| 400 | 230.0390625 | 400 |
| 450 | 258.890625 | 450 |
| 500 | 288.515625 | 500 |
| 1000 | 572.46875 | 1000 |
| 2500 | 1436.78125 | 2500 |
| 5000 | 2875.890625 | 5000 |
| 7500 | 4308.898438 | 7500 |
| 10000 | 5752.570313 | 10000 |
| 20000 | 11486.375 | 20000 |

### ../Downloads/snapshot2.png

### Figura 1 – Gráfico representativo da utilização da memória durante a execução do algoritmo Ukkonen para N = 20000, obtido através da ferramenta *massif* do *valgrind*.

### Conclusões

Com base na análise temporal é possível constatar que até um certo ponto (7500 caracteres por cada string), o tempo necessário para aplicar o algoritmo de Ukkonen cresce linearmente. A partir daí as partes do algoritmo que até então poderiam ser amortizadas passam a produzir algum atraso significativo.

Com base na análise espacial é possível constatar que independentemente do tamanho da string generalizada (soma do tamanho de todas as strings) o espaço ocupado cresce linearmente. Devido ao tamanho do gráfico só é apresentado um dos exemplos obtidos com a ferramenta *massif* do *valgrind* mas todas as experiências realizadas tinham o mesmo comportamento).