Análise Empírica a AVL Trees e Treaps

Armando Manuel Martins (201504230), Diogo Ribeiro (201504115)

Faculdade de Ciências da Universidade do Porto,
Mestrado de Ciência de Computadores,
Tópicos Avançados em Algoritmos

1. Introdução

Este relatório refere-se ao primeiro trabalho de Tópicos Avançados em Algoritmos e é inserido na primeira parte Unidade Curricular onde foram discutidas várias estruturas de dados. O objetivo é decidir de entre dois grupos de estruturas dadas (determinísticas e probabilísticas) pelo menos duas para as estudar e comparar. Escolhemos explorar as AVL Trees e Treaps.

2. AVL Tree

Uma AVL Tree na sua essência uma árvore de pesquisa binária tendo no entanto a particularidade de garantir que, para cada nó, as alturas das suas sub-árvores diferente por, no máximo, uma unidade (invariante de altura). Esta propriedade é sempre mantida mesmo quando se executam operações de inserção ou remoção.

2.1 Operações

As operações que foram implementadas para avaliação foram Inserção (insert), Remoção(delete) e procura(search).

2.1.1 Inserção

Numa árvore AVL, a inserção é idêntica à inserção a qualquer outra árvore binária tendo no entanto de garantir que se mantém o invariante de altura. Para isso sempre que se insere um novo valor é necessário verificar se a propriedade se mantém e caso isso não aconteça são realizadas operações de rotação para a repor. Corrigir uma árvore mais alta à esquerda ou à direita é simétrico sendo que quando numa se realiza uma rotação à esquerda na outra se realiza uma rotação à direita. Consideremos os seguintes casos em que o lado esquerdo é mais pesado:

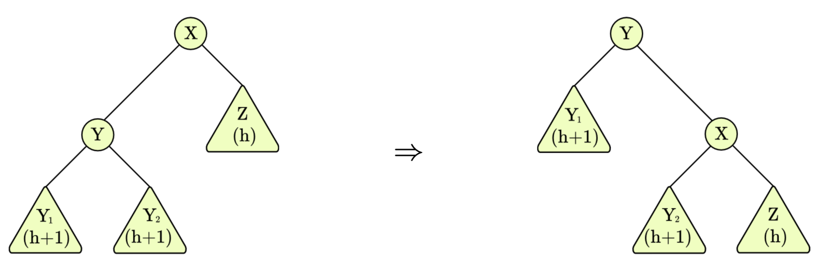

Fig. 1. Lado esquerdo é demasiado pesado por isso fazemos uma rotação à direita em X. [1]

Ao realizar esta rotação podemos ver que a árvore fica equilibrada sendo que mesmo que a altura de Y_2 seja apenas h isto se verifica.

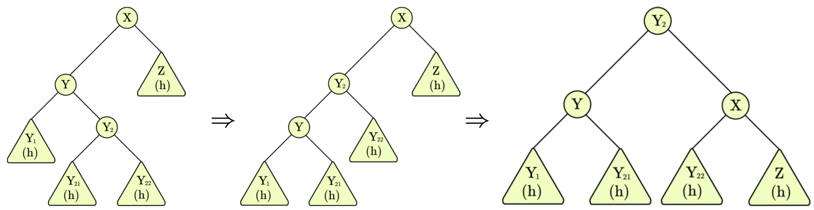

Fig. 2. Lado esquerdo é demasiado pesado mas também não é equilibrado. Aqui realizamos uma rotação à esquerda em Y seguida de uma rotação à direita em X. [1]

Neste caso o lado esquerdo é demasiado pesado, no entanto não equilibrado em relação a ele próprio uma vez que o seu lado direito é mais pesado. Isto significa que temos duas fontes de desequilíbrio, X e Y. Para corrigir isto realizamos uma rotação à esquerda em Y seguida de uma rotação à direita em X.

Estes dois exemplos notam um desequilíbrio derivado de um maior peso do lado esquerdo da árvore, no entanto, podemos também verificar desequilíbrio por termos maior peso no lado direito. A correção para este desequilíbrio é idêntica sendo que os passos são o inverso, ou seja, onde antes fazíamos uma rotação à esquerda agora fazemos à direita e vice-versa.

Tanto no melhor caso como no pior temos de percorrer a árvore até ao último nível para encontrar o local onde deveremos inserir o novo valor. Como a árvore tem altura $\log n$ esta operação tem complexidade assintótica de $O(\log n)$.

2.1.2 Procura

Esta operação consiste em verificar se um dado elemento consta na árvore retornando True em caso afirmativo e False caso contrário. Esta pesquisa é feita exatamente da mesma forma que numa árvore de pesquisa binária, chamamos a função dando a raiz da árvore e se o valor for superior ao valor da raiz chamamos a mesma função para a sub-árvore à direita, se for menor chamamos para a sub-árvore à esquerda e se for igual significa que encontramos o valor.

No melhor caso a pesquisa tem complexidade assintótica de $O(1)$ uma vez que este se verifica quando o valor que procuramos é logo o primeiro (raiz da árvore). No pior caso teremos de percorrer toda a árvore até ao seu último nível tendo assim complexidade assintótica de $O(\log n)$ sendo este caso verificado quando o valor que procuramos não se encontra na árvore ou quando se encontra no último nível.

2.1.3 Remoção

Esta operação segue o mesmo raciocínio da anterior. Procura o valor que queremos remover e após a sua remoção deixamos o nó de menor valor da sua sub-árvore esquerda no seu lugar e verificamos o balanceamento da árvore. Caso a árvore não esteja balanceada utilizamos as operações descritas na parte de inserção para tornar a árvore equilibrada.

No melhor caso a remoção acontece na raiz da árvore e por isso segue uma complexidade constante, $O(1)$. Já no pior caso temos de percorrer a árvore uma vez até uma das folhas o que representa um complexidade de $O(\log n)$.

3. Treap

Esta estrutura tal como a anterior segue a linha de uma árvore binária mas com uma mistura de outra estrutura também muito conhecida, a heap. A Treap, Tree + Heap, toma partido das propriedades destas duas estruturas.

Propriedade Heap:

- O nó com a maior prioridade deve ser a raiz da árvore

Propriedade Tree:

- Qualquer nó com $key(u) < key(v)$ deve estar na sub-árvore à esquerda
- Qualquer nó com $key(w) \geq key(v)$ deve estar na sub-árvore à direita

Seguindo estas propriedades a Treap é uma árvore binária de pesquisa para as chaves dos nós e uma heap de máximo para as prioridades.

3.1 Operações

Tal como na AVL, as operações que foram implementadas para avaliação foram Inserção, Remoção e Procura.

3.1.1 Inserção (Insert)

A operação de inserção é a responsável pela criação da árvore. Para inserirmos um nó novo é gerado um número aleatório para a prioridade. Após isso segue pela árvore comparando as chaves dos nós e preservando a propriedade Tree. Quando chega a uma folha compara a prioridade dele com a do nó pai, se a dele for menor que a do pai deixa-se estar como folha, senão realizam-se rotateLeft ou rotateRight, dependendo se o nó que estamos a analisar é o filho à direita ou à esquerda respetivamente, até encontrarmos um nó pai cuja prioridade seja maior do que o nó que estamos a ”elevar” em direção ao topo da árvore. No caso de não encontrarmos um pai que tenha prioridade maior significa que esse nó é a raiz da árvore.

```python
def insertTreap(node, data):
  if node == None:
    return TreapNode(data)

  elif data < node.data:
    node.left = insertTreap(node.left, data)
    if node.left != None and node.left.priority >= node.priority:
      node = rotateRight(node)

  else:
    node.right = insertTreap(node.right,data)
    if node.right != None and node.right.priority >= node.priority:
      node = rotateLeft(node)

  return(node)
```

Listing 1. Código Python - função insert

3.1.2 Procura (Search)

A operação de procura segue a ideia geral de uma BST (Binary Search Tree). Segue recursivamente na árvore até encontrar o nó pretendido.

3.1.3 Apagar (Delete)

Esta função é idêntica à procura mas quando encontra o nó pretendido apaga-o. No caso desse nó ter filho à esquerda e direita compara as prioridades deles e escolhe como pai o que tem maior prioridade para contemplar a propriedade heap.

3.2 Complexidade Assintótica

Como esta estrutura não "obriga" a que a árvore seja equilibrada pode acontecer que a inserção dos elementos seja feita em linha, isto é, todos os elementos estão por ordem decrescente ou crescente e as suas prioridades estão por ordem decrescente. Por isso, ao inserirmos um novo nó ou estivermos à procura de um que pode estar na folha da árvore (search ou delete) temos de percorrer os n nós inseridos. Por esta razão no pior caso segue $O(n)$ para as 3 operações implementadas. O caso médio das três operações segue $O(\log n)$ porque a probabilidade do pior caso acontecer é extremamente baixa e por isso, em princípio, o tamanho da árvore não ultrapassará $\log n$. No melhor caso, a inserção continua $O(\log n)$ uma vez que este terá de seguir até ao último nível sempre para inserir. A procura e remoção no melhor caso, tal como nas AVL Trees, o nó pretendido é a raiz da árvore e por isso segue complexidade constante, $O(1)$.

4. Resultados

Para testar as duas estruturas em causa elaboramos listas de números que foram convertidas em árvores de forma a testar os tempos de inserção e remoção. Não foi testado o tempo de pesquisa visto que esta é muito semelhante à remoção.

4.1 Datasets

As listas de inteiros são geradas aleatoriamente sendo compostas por diferentes valores de 1 a n sendo n o tamanho da lista. Para os testes corridos geramos 5 árvores de tamanho 100, 1000, 10000, 1000000 e 10000000 respetivamente.

Para testarmos a remoção geramos sempre aleatoriamente uma lista com 20% dos elementos da árvore sendo esses os elementos a serem removidos.

4.2 Comparação

Table 1. Operação - Inserção

	AVL Tree Time	AVL Tree Profundidade	Treap	Treap Tree Profundidade
100 elementos	0.0012011528015136719	8	0.00045108795166015625	14
1000 elementos	0.012511014938354492	12	0.00640416145324707	24
10000 elementos	0.17727994918823242	16	0.08694577217102051	32
1000000 elementos	37.66878414154053	24	26.463167905807495	51
10000000 elementos	483.2004110813141	28	353.75063014030457	61

Table 2. Operação - Remoção

	AVL Tree	Treap
100 elementos	0.0001900196075439453	$6.985664367675781e-05$
1000 elementos	0.002916097640991211	0.0009980201721191406
10000 elementos	0.038461923599243164	0.014424800872802734
1000000 elementos	6.5350189208984375	2.909256935119629
10000000 elementos	99.97793507575989	41.4301438331604

5. Conclusão

Verificamos que a Treap é mais rápida em ambas as operações para os casos testados. No entanto, verificamos também que a profundidade das árvores criadas utilizando a estrutura AVL é menor o que significa que, em casos extremos, as operações de pesquisa e remoção serão mais rápidas. Desta forma concluímos que a aleatoriedade da Treap ao gerar as prioridades resulta numa melhor performance para o exemplo utilizado e apesar de ser pior em casos extremos acaba por ser melhor em média.

6. Referências

[1] Ribeiro P (2020/2021) Balanced binary search trees. Ultimo acesso 07/05/2021 Available at https://www.dcc.fc.up.pt/~pribeiro/aulas/taa2021/balancedsearchtrees.pdf.