Amortized Analysis
Master MOSIG - Algorithms and Program Design

Marc Tchiboukdjian - Denis Trystram

25-11-2009

Course

Objectives

Design and analysis of data structures using amortization.

To Remember

1. **Amortized analysis.** The time required to perform a sequence of data-structure operations is averaged over all the operations performed.
2. **No probability involved.** An amortized analysis guarantees the average performance of each operation in the worst case.
3. Three techniques to analyze the cost: aggregate analysis, accounting method, potential method.
4. **Aggregate analysis.** If for all n, a sequence of n operations takes worst-case time $T(n)$ in total, the amortized cost per operation is $T(n)/n$.
5. **Accounting method.** Assign different charges to different operations. The amortized cost $\hat{c}_i$ of operation i is the amount we charge. When $\hat{c}_i$ is more than the actual cost c_i of operation i store the difference $\hat{c}_i - c_i$ on specific objects in the data structure as credit. Use credit later to pay for operations s.t. $c_i > \hat{c}_i$. Credit should never be negative, i.e. for all sequences of n operations $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$.
6. **Potential method.** Think of the credit as potential stored with the entire data structure and not on specific objects. Can release potential to pay for future operations. Most flexible technique.

   If D_i is the data structure after ith operation and $\Phi : D_i \to \mathbb{R}$ the potential function then the amortized cost of operation i is

   $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + \Delta\Phi_i.$$

   If $\forall i, \Phi(D_i) \geq 0$ and if $\Phi(D_0) = 0$ then he amortized cost is

   $$\frac{1}{n}\sum_{i=1}^n c_i = \frac{1}{n}\sum_{i=1}^n (\hat{c}_i - \Delta\Phi_i) = \frac{1}{n}\sum_{i=1}^n \hat{c}_i + \frac{\Phi(D_0) - \Phi(D_n)}{n} \leq \frac{1}{n}\sum_{i=1}^n \hat{c}_i.$$

   Thus the amortized cost is always an upper bound on the actual cost.

Analysis of table expansion using aggregate analysis

We implement a table that can resize itself when an insertion arises and the table is full. A common heuristic is to allocate a new table with a size doubled. If only insertions are performed, the load factor of a table is always at least 1/2, and thus the amount of wasted space never exceeds half the total space.

In the following pseudocode, we assume that T is an object representing the table. The field $table[T]$ contains a pointer to the block of storage representing the table. The field $num[T]$ contains the number of items in the table, and the field $size[T]$ is the total number of slots in the table.

Initially, the table is empty: $num[T] = size[T] = 0$.

Algorithm 1: Table Insert

```
   Input: The table $T$ and the element to be inserted $x$
   Result: $T \leftarrow T \cup x$
 1 if $size[T] = 0$ then
 2     allocate $table[T]$ of size 1
 3     $size[T] \leftarrow 1$
 4 end
 5 if $num[T] = size[T]$ then
 6     allocate $newtable$ of size $2 \times size[T]$
 7     insert all items in $table[T]$ into $newtable$
 8     free $table[T]$
 9     $table[T] \leftarrow newtable$
10     $size[T] \leftarrow 2 \times size[T]$
11 end
12 insert $x$ into $table[T]$
13 $num[T] \leftarrow num[T] + 1$
```

Amortized cost of insertion. Let c_i be the cost of the ith insertion.

$$c_i = \begin{cases} i \text{ if } i-1 \text{ is an exact power of } 2 \\ 1 \text{ otherwise} \end{cases}$$

Thus, the average cost over n insertions is

$$T_{average} = \frac{\sum_{i=1}^{i=n} c_i}{n} \leq \frac{n + \sum_{j=0}^{j=\lfloor \log n \rfloor} 2^j}{n} \leq \frac{n+2n}{n} = \mathrm{O}(1).$$

B-trees

Definition. B-trees of degree $t \geq 2$ are trees having the following properties:

- every node x has the following fields
  - $n[x]$ the number of keys currently stored in node x
  - the keys themselves, stored in non decreasing order so that $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$
  - $leaf[x]$ a boolean value that is TRUE is x is a leaf and FALSE is x is an internal node
- each internal node x also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children.
- the keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$ then $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}$.
- every node other than the root must have at least $t-1$ keys and at most $2t-1$ keys.

Height. The height h of an n-key B-tree of degree t satisfies $h \leq \log_t \frac{n+1}{2}$.

- we have $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$
- which gives $t^h \leq (n+1)/2$
- taking $\log_t$ on both sides gives the result.

B-trees are balanced trees (they guarantee that $h = \mathrm{O}(\log n)$).

Search. Simple generalization of the search in a binary tree. Cost is $\mathrm{O}(\lg t \log_t n) = \mathrm{O}(\log n)$.

Insertion. Insert the key in a leaf node. If leaf node x is full, split the node around its median key $key_t[x]$ into two nodes having $t-1$ keys each. The median key moves up into x's parent y. If y is also full, split again. The need to split full nodes can propagate all the way up the tree. Cost of split is $\mathrm{O}(t)$, height is $h = \mathrm{O}(\log_t n)$, at most one split by node on a leaf to root path thus the worst case insert cost is $\mathrm{O}(t \log_t n)$.

I/O complexity. Choose t so that each node fit in one memory block. SEARCH, INSERT and DELETE cost $\mathrm{O}(h) = \mathrm{O}(\log_t n)$ block transfers. Compared to balanced binary trees $\mathrm{O}(\log n)$ this is a $\log t$ factor improvement. In practice t is big, therefore most databases propose an implementation of the B-tree (or a variation).

Amortized costs of splits in B-tree using the accounting method

Intuition. A leaf node (depth 0) has to be split every t insertions. Similarly a node at depth k can handle t^{k+1} insertions before being split.

Charging scheme. Let $x_0, x_1, \ldots, x_k$ be the path in the B-tree from the leaf x_0 where the element is inserted to the root x_k. We charge the element $\frac{1}{t^k}$ for the node x_i. Total charge is $\hat{c}_i = \sum_{i=0}^{k} \frac{1}{t^k} \leq 2$.

Proof. Let x a node in the B-tree at depth k. Suppose we have just split the node and there is no more credit. Next split will happen after t^{k+1} elements being inserted in the subtree rooted at x. Credit will be $t^{k+1}\frac{1}{t^k} = t$. Cost to split is t. We have just enough to pay for the split.

Conclusion. The amortized cost of splits is $\hat{c}_i = \mathrm{O}(1)$.

Exercises

Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $n_{k-1}, \ldots, n_0$. We have k sorted arrays $A_0, \ldots, A_{k-1}$, where the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefor $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, there is no particular relationship between elements in different arrays.

1. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.
2. Describe how to insert a new element into this data structure. Analyze its worst-case and amortized running times.
3. Discuss how to implement DELETE.

Amortized weight-balanced trees

Consider an ordinary search tree augmented by adding to each node x the field $size[x]$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \leq \alpha < 1$. We say that a given node x is **α-balanced** if

$$size[left[x]] \leq \alpha \times size[x]$$

and

$$size[right[x]] \leq \alpha \times size[x].$$

The tree as a whole is α-**balanced** if every node in the tree is α-balanced.

1. A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes 1/2-balanced. Your algorithm should run in time $O(size[x])$.
   (Hint: you should use a sorted array of size $size[x]$)

2. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.
   Hint: write a recurrence equation.

For the remainder of this problem, assume the constant α is strictly greater that 1/2. Suppose that INSERT and DELETE are implemented as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then the subtree rooted at the highest such node in the tree is rebuilt so that it becomes 1/2-balanced.

3. What is the worst case cost of an insertion (including the rebalancing cost) ?

Instead of using the worst case analysis, we shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define

$$\delta(x) = |size[left[x]] - size[right[x]]|,$$

and we define the potential of T as

$$\Phi(T) = k \sum_{x \in T : \delta(x) \geq 2} \delta(x),$$

where k is a sufficiently large constant that depends on α.

Remember that if c_i is the actual cost of the ith operation and T_i is the tree after the ith operation then the amortized cost of the ith operation is $\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1}) = c_i + \Delta\Phi_i$.

4. Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

5. Suppose that we need to rebalance the subtree rooted at node $x \in T$. Let $m = size[x]$. Show that $\delta(x) \geq (2\alpha - 1)m + 1$.

6. Let Φ_{before} the potential in the tree just before we rebalance the subtree rooted at node x and Φ_{after} the potential in the tree just after. Show that $\Phi_{before} - \Phi_{after} \geq k((2\alpha - 1)m + 1)$.
   Hint: divide the potential in two parts, Φ' which is the potential contained in the subtree rooted at x and Φ'' the remaining potential.

7. Suppose that m units of potential can pay for rebuilding an m-node subtree. When we rebuild the subtree rooted at x, we release potential $\Phi_{before} - \Phi_{after}$. We would like to be able to pay for the rebuild operation with the potential release. How large must k be in terms of α so that the release of potential $\Phi_{before} - \Phi_{after}$ can pay for the rebuild of the subtree rooted at x?

8. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.