Amortization*

Simple Examples

*Mostly taken from CLR book (Chapter 17)

Amortization – Why and What?

- As an everyday example think of paying off a debt, e.g., mortgage, car loan etc., by smaller payments made over time.
- In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over an arbitrary sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average (over an arbitrary sequence of operations) performance of each operation in the worst case.
- Word of Caution: In applications where cost of individual operations is required to have low and/or comparable costs, we may need an algorithm with a worse amortized cost but a better worst-case per operation bound.
- Another way of looking at performance of an algorithm: Amortized analysis exploits the fact that some "expensive" operations usually pay for future operations by limiting the number or cost of future such operations. Examples: splay trees, B-trees, disjoint set unions, Fibonacci heaps, security, database, distributed computing applications etc.
- "Amortized analysis is closely related to competitive analysis, which involves comparing the worst case performance of an online algorithm to the performance of an optimal offline algorithm on the same data. Amortization is useful because competitive analysis's performance bounds must hold regardless of the particular input."

Amortization – How?

- There are 3 principal approaches:
  - **Aggregate analysis**: we determine an upper bound T(n) on the total cost of a sequence of n operations. The average cost per operation is then T(n)/n. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.
  - **Accounting method**: [Banker's view] we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.
  - **Potential method**: Similar to the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.
- The charges assigned during an amortized analysis are for analysis purposes only. They need do not appear in the code.
- When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design.

Aggregate Analysis

- In aggregate analysis, we show that for all n, a sequence of n operations takes worst-case time T(n) in total. In the worst case, the average cost, or amortized cost, per operation is therefore T(n)/n. **Note** that this amortized cost applies to each operation, even when there are several types of operations in the sequence; this is one difference with the accounting method and the potential method, that may assign different amortized costs to different types of operations.
- **Example 1**: Consider a fixed size (say n) stack S with 2 operations Push (S, x) and Pop (S, x). Assuming there are overflow and underflow checks for errors, each operation takes O(1) time; consider the cost of each is 1; then total cost of any arbitrary sequence of n operations T(n) = $\Theta(n) \Rightarrow$ amortized cost of each operation is O(1).
  - Now, add a multi-pop operation MP (S, k) that pops k elements sequentially; obviously, total cost of MP (S, k) operation is min (s, k) [this many Pops]. Worst case behavior of multi-pop operation MP (S, k) is then O(n). Does that mean that amortized cost of an arbitrary sequence of 3 operations is O(n)? NO.
  - Note that we can pop each object from the stack at most once for each time we have pushed it onto the stack. The number of times that POP can be called on a nonempty stack, including calls within multi-pop is at most the number of Push operations, which is at most n. For any value of n, any sequence of n Push, Pop and multiop operations takes a total of O(n) time. The average cost of an operation is O(n)/n = O(1).
  - Note that we did not use probabilistic reasoning

Aggregate Analysis

- **Example 2:** [Increment a binary counter] Consider a k-bit counter A[0 .. k – 1] that is initially 0. An integer stored in A is $x = \sum_{0}^{k-1} A[i].2^i$
  - To add 1 (modulo 2^k) to the counter, we use **{**i=0; while (i<k && A[i]=1) {A[i]=0; i++} if i<k A[i]=1**}**
- A single execution of increment takes time O(k) in the worst case, when A contains all 1s. Thus, a sequence of n such operations on an initially zero counter takes time O(nk) in the worst case. Correct? Yes, but …
- Observe that not all bits flip each time increment is called. A[0] does flip each time increment is called. The next bit up, A[1], flips only every other time and so on.
- The total number of flips in the sequence of n increment operation is $\sum_{0}^{k-1} \left\lfloor n/2^i \right\rfloor < n \sum_{0}^{\infty} 1/2^i = 2n$
- The worst-case time for a sequence of n increment operations on an initially zero counter is therefore O(n). The average cost of each operation, and therefore the amortized cost per operation, O(n)/n = O(1).
- **Exercise**: Show that if a decrement operation is included, a sequence of n operations will cost Θ(nk) time in the worst case.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Accounting Method

- In the accounting method, we assign differing charges to different operations, with some operations charged more or less than they actually cost (c). We call the amount we charge an operation its amortized cost (ac). When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit. Credit can help pay for later operations whose amortized cost is less than their actual cost. Different operations may have different amortized costs – thus, this method differs from aggregate analysis, in which all operations have the same amortized cost.
- To show that in the worst case the average cost per operation is small by analyzing with amortized costs, we must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, this relationship must hold for all sequences of operations.
- If we denote the actual cost of the i^{th} operation by c_i and the amortized cost of the i^{th} operation by ac_i , we require $\sum_{i=1}^{n} ac_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations.
- **Example 1**: Consider the stack example again. Recall that the actual costs of the operations were c(Push) = 1, c(Pop) = 1, c(multi-pop) = min (s, k). Let us assign the amortized costs of those be 2, 0, 0 respectively. The rest is obvious.
- Note that the amortized cost of multi-pop is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

Accounting Method

- **Example 2:** [Incrementing a binary counter] We again we analyze the Increment operation on a binary counter that starts at zero.
  - The running time of this operation is proportional to the number of bits flipped; we use this as our cost.
  - We charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 (out of the 2 charged) to pay for the actual setting of the bit, and we place the other 1 on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a credit 1 on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the 1 already credited on the bit.
  - Now we determine the amortized cost. The cost of resetting the bits within the while loop is paid for by the credits on the bits that are reset. The Increment procedure sets at most one bit, and therefore the amortized cost of an Increment operation is at most 2. The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n Increment operations, the total amortized cost is O(n) which bounds the total actual cost.
- **Exercise:** Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Potential Method

- The potential method of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure.
- We perform n operations, starting with an initial data structure D_0. For each i = 1, 2, … n, let c_i be the actual cost of the i^{th} operation and D_i be the data structure that results after applying the i^{th} operation to data structure D_{i-1}. A **potential function Φ** maps each D_i to a real number $\Phi(D_i)$. The **amortized cost** ac_i of the i^{th} operation is given by $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, i.e., amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. Thus, the **total amortized cost of n operations is $\sum ac_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$.**
- If we can choose a suitable Φ such that $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost $\sum ac_i$ will be an upper bound on total actual cost $\sum c_i$, i.e., $\Phi(D_i) \geq \Phi(D_0)$ for all i, since n is arbitrary. Usually we choose $\Phi(D_0) = 0$ and then show that $\Phi(D_i) \geq 0$ for all i [This is by no means necessary].
- The amortized costs, as defined, depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. We often find trade-offs that we can make in choosing a potential function; the best potential function to use depends on the desired time bounds.

Potential Method

- Example 1: Stack Operations with Pop, Push and multi-pop:
  - Define Φ to be the number of elements in the stack, $\Phi(D_0) = 0$ (stack is initially empty).
  - Potential after i operations, $\Phi(D_i) \geq 0 = \Phi(D_0)$ (the stack never has negative number of elements); total amortized cost of n operations with respect to Φ represents an upper bound on the actual cost.
  - If the i^{th} operation on a stack with x elements is a Push, $\Phi(D_i) - \Phi(D_{i-1}) = (x+1) - x = 1$. So, amortized cost $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
  - If the i^{th} operation on a stack with x elements is a multipop (S, k), causing $y = \min(x,k)$ elements to be popped from the stack S, then actual cost is y and potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = y - y = 0$. Similarly, the amortized cost of an ordinary Pop operation is 0.
  - The amortized cost of each of the three operations is O(1), and thus the total amortized cost of a sequence of n operations is O(n). Since $\Phi(D_i) \geq 0 = \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore O(n).

Potential Method

- **Example 2:** Incrementing a binary counter. We define the potential of the counter after the i^{th} Increment operation to be b_i, *the number of 1s in the counter after* the i^{th} operation.
  - Assume the i^{th} operation resets t_i bits and sets at most one bit to 1 (why?). The actual cost c_i is at most t_i+1.
    - If $b_i = 0$, then the i^{th} operation resets all k bits, and so $b_{i-1} = t_i = k$.
    - If $b_i > 0$, $b_i = b_{i-1} - t_i + 1$
  - In either case, $b_i \leq b_{i-1} - t_i + 1$ and so, the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$, or amortized cost $\mathbf{ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2}$.
  - If counter starts at zero, then $\Phi(D_0) = 0$ and since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n Increment operations is an upper bound on the total actual cost, and so the worst-case cost of n Increment operations is O(n).
  - **Brain Teaser**: Now, suppose the counter does not start at 0, but at some (arbitrary) positive value. Can we compute the amortized cost? Is it still O(n)? May be, under some condition?
- **Exercises:**
  1. Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in O(log n) worst-case time. Give a potential function Φ such that the amortized cost of INSERT is O(log n) and the amortized cost of EXTRACT-MIN is O(1) and show that it works.
  2. What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

Solution of Brain Teaser

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method gives us an easy way to analyze the counter even when it does not start at zero. The counter starts with b_0 1s, and after n INCREMENT operations it has b_n 1s, where $0 \leq b_0, b_n \leq k$. (Recall that k is the number of bits in the counter.) We can rewrite equation (17.3) as

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

We have $\hat{c}_i \leq 2$ for all $1 \leq i \leq n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

Note in particular that since $b_0 \leq k$, as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Dynamic Tables

- We allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table. Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size. Using amortized analysis, we show that the amortized cost of insertion and deletion is only O(1), even though the actual cost of an operation is large when it triggers an expansion or a contraction.
- The details of data structures used (e.g., arrays, stack, heap, or hash table) is not necessary for this analysis – we dynamically expand the table when we attempt an insertion in an already full table (T) and we contract the table when the table becomes mostly empty.
- We need a measure when to trigger contraction. We use a similar concept used in hashing – the load factor of a table T, α(T), number of items stored in the table divided by the size (number of slots) of the table. We assign an empty table (one with no items) size 0, and we define its load factor to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.
- Note that the problem is different from dynamic insert/delete in a linked list. We will start with table expansion only and then will add the deletion.

Dynamic Table Expansion

- Assume that storage for a table T is allocated as a contiguous array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1; assuming integer elements, int *T = (int *) malloc (size_of_T * sizeof(int)); use integers Tsize and Tnum to denote the size of the table and the actual number of elements inserted in T. Note that only operation on the table is an insert (T, x).
- To insert an item into a full table, we expand the table by allocating a new table with more slots than the old table had; we allocate a new array for the larger table and then copy items from the old table into the new table.
- A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the **load factor** of the table is always at least 1/2, and thus the amount of wasted space never exceeds half the total space in the table.
- Initially, the table is empty, i.e., Tsize = 0 and Tnum = 0.

  Insert (T, x)

  if (Tsize == 0) {allocate a table of size 1; Tsize = 1; insert x; Tnum = 1};

  if (Tnum == Tsize) {allocate a table newT of size 2*Tsize; copy elements; free T; T = newT; Tsize = 2*Tsize; insert x; Tnum ++;}
- What is the cost of ith operation? If the table is not full, $c_i = 1$, else $c_i = i$ (1 + cost of copying earlier i – 1 elements).

$$c_i = \begin{cases} i & \text{if } i\text{-}1 \text{ is a power of 2} \\ 1 & \end{cases}, \text{ or } \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\log n} 2^j < n + 2n = 3n$$

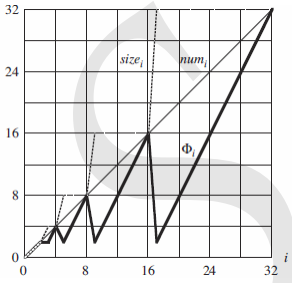
Dynamic Table Expansion

- Using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3. Intuitively, each item pays for 3 elementary insertions: (1) inserting itself into the current table, (2) moving itself when the table expands, and (3) moving another item that has already been moved once when the table expands.
- Consider the case when the table size is m immediately after expansion – the table has m/2 items (m is a power of 2) and has no credit at this point. We charge 3 dollars for each insertion. The elementary insertion that occurs immediately costs 1 dollar. We place another dollar as credit on the item inserted (for its eventual copy into a bigger table). We place the third dollar as credit on one of the m/2 items already in the table (for its eventual copy). The table will not fill again until we have inserted another m/2 – 1 items, and thus, by the time the table contains m items and is full, we will have placed a dollar on each item to pay to reinsert it during the expansion.
- We can use the potential method to analyze a sequence of n table-insertions. Consider a potential function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential. Consider the function **Φ(T) = 2Tnum – Tsize**. We observe
  - Φ(T) = 0; Immediately after an expansion, we have Tnum = Tsize/2, so Φ(T) = 0, as desired.
  - Since table is always at least half full, Tnum ≥ Tsize/2 ⇒ Φ(T) is always +ve.
- Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.
- We still need to compute the amortized cost of the i^{th} insert operation.

Dynamic Table Expansion

- Let num_i, $size_i$, Φ_i denote number of items in the table, size of the table and potential function respectively after the i-th operation. Initially $num_0 = size_0 = \Phi_0 = 0$. we need to consider two cases:
  - i^{th} operation does not trigger an expansion: then $size_i = size_{i-1}$ and the amortized cost of the operation is $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) = 3$
  - If the i^{th} operation does trigger an expansion, then we have $size_i = 2size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, which implies that $size_i = 2(num_i - 1)$. Thus, the amortized cost of the operation is $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3$

- The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i .
- Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table.
- Afterwards, the potential drops to 0, but it is immediately increased by 2 upon inserting the item that caused the expansion.



Dynamic Table Expansion & Contraction

- Simple deletion is easily done by Tnum --. We also want to contract the table when the load factor becomes too small; such contraction is somewhat analogous to expansion – when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. As before, we want
  - Load factor is bounded below by a positive constant
  - Amortized cost of a single operation is bounded above by a constant.
- To mirror the strategy we used for table expansion, a possible strategy for contraction could be to halve the size when a deleting an item would cause the table to become less than half full, guaranteeing that the load factor of the table never drops below ½.
- This strategy looks simple, but does not satisfy the second requirement. Consider a scenario: we perform n operations on an initially empty table where n is a power of 2. The first n/2 operations are insertions, with a total cost of $\Theta(n)$ and Tnum = Tsize = n/2. The next n/2 operations is given by the sequence, insert, delete, delete, insert, insert, delete, delete, insert, insert, ...
- The first insertion causes the table to expand to size n. The next two deletions cause the table to contract back to size n/2. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$ and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of a single operation $\Theta(n)$.
- The downside of this strategy is obvious: after expanding the table, we do not delete enough items to pay for a contraction. Likewise, after contracting the table, we do not insert enough items to pay for an expansion.

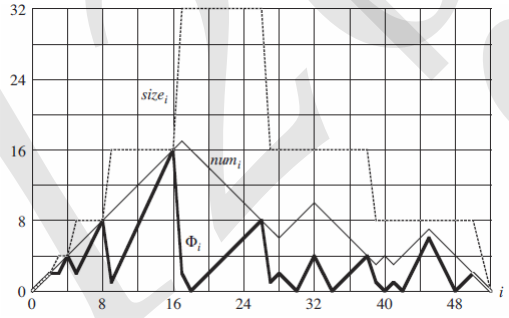
Dynamic Table Expansion & Contraction

- New strategy: We allow the load factor of the table to drop below ½. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than ¼ full, rather than ½ full as before. The load factor of the table is therefore bounded below by the constant ¼. The deletion algorithm is analogous to the insertion one.
- *Why does it solve the previous problem*? The problem was that no credit was available at the time of contraction, hence amortized cost was not constant. The new scheme solves the problem. The idea is at the beginning, after both expansion and contraction the potential ideally should be zero. Thus, we will need a potential function that has grown to Tnum by the time that the load factor has either increased to 1 or decreased to ¼. After either expanding or contracting the table, the load factor goes back to ½ and the table's potential reduces back to 0. Note: inventing an appropriate potential function for an application is the most important job.
- Remember α(T) = Tnum/Tsize; for an empty table Tnum = Tsize = 0 and α(T) = 1. Here is one possible potential function we use.

$$\Phi(T) = \begin{cases} 2Tnum - Tsize, & \text{if } \alpha(T) \geq 1/2 \\ Tsize/2 - Tnum, & \text{if } \alpha(T) < 1/2 \end{cases}$$

- **Observe** that the potential of an empty table is 0 and that the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

Dynamic Table Expansion & Contraction



- When the load factor is ½, the potential is 0.
- When the load factor is 1, Tsize = Tnum, or Φ(T) is Tnum and potential is enough to pay for an expansion.
- When the load factor is ¼, Tsize = 4Tnum or Φ(T) = Tnum; thus potential can pay for contraction if the next operation is a deletion.

- The effect of a sequence of n Insert and delete operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential Φ_i each measured after i^{th} operation.
- The thin line shows num_i, the dashed line shows $size_i$, and the thick line shows Φ_i

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2, \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2, \end{cases}$$

Dynamic Table Expansion & Contraction

- Now, consider a sequence of n insert/delete operations. Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$ and $\Phi_0 = 0$.
- Consider the case when i^{th} operation is an insert; $num_i = 1 + num_{i-1}$
  - If $\boldsymbol{\alpha_{i-1} \geq ½}$, the analysis is exactly as before; whether the table expands or not, the amortized cost ac_i of the operation is at most 3.
  - If $\boldsymbol{\alpha_{i-1} < ½}$, the table cannot expand as the result of i^{th} operation; the table expands only when $\alpha_{i-1} = 1$. There are 2 subcases.
    - $\boldsymbol{\alpha_i < ½}$: $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0$.
    - $\boldsymbol{\alpha_i \geq ½}$: $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) = 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) = 3num_{i-1} - (3/2)size_{i-1} + 3 = 3\alpha_{i-1}size_{i-1} - (3/2)size_{i-1} + 3 < (3/2)size_{i-1} - (3/2)size_{i-1} + 3 = 3$.

  Thus, the amortized cost of an insert operation is at most 3.
- Consider the case when i^{th} operation is a delete; $num_i = num_{i-1} - 1$.
  - $\boldsymbol{\alpha_{i-1} < ½}$**:** There are two possibilities:
    - Table does not contract after i^{th} operation: $size_i = size_{i-1}$; $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 2$.
    - Table does contract after i^{th} operation: since copying is involved, actual cost $c_i = num_i + 1$, $size_i = size_{i-1}/2$, $num_{i-1} = size_{i-1}/4$. : $ac_i = c_i + \Phi_i - \Phi_{i-1} = (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 1$.
  - $\boldsymbol{\alpha_{i-1} \geq ½}$**:** No table contraction; $size_i = size_{i-1}$; $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2numi - sizei) - (2num_{i-1} - size_{i-1}) = 3$. Thus, the amortized cost of a delete operation is at most 3.

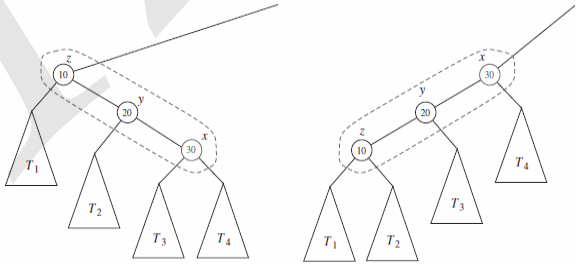
Splay Trees & Amortized Analysis

Observations

- There are many variations of balanced binary trees. The prominent among them are Red Black trees, B-trees (again of many kinds), weak AVL trees etc.; they all share the same property that (1) height of the tree is $O(\log_2 n)$, where n is the number of nodes and (2) the worst case time for each of the operations of search, insert, delete is $O(\log_2 n)$.
- Limitations of Balanced Search Trees: Balanced search trees require storing an extra piece of information per node. Their worst-case, average-case, and best-case performance are essentially identical. We do not win when easy inputs occur – would be nice if the second access to the same piece of data was cheaper than the first. The 90-10 rule – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items; it would be nice to get easy wins for the 90% case.
- The structure of **Splay Tree** is conceptually different from that of AVL or balanced binary search trees. It applies a certain **move-to-root** operation, called **splaying** after every access, in order to keep the search tree balanced in an amortized sense. The splaying operation is performed at the bottommost node x reached during an insertion, deletion, or even a search.
  - The structure of a splay tree is simply a binary search tree T; there are no additional height, balance, or color labels that we associate with the nodes of this tree.
  - Each operation, search, insert, delete, has an amortized cost of $O(\log_2 n)$.
  - In the worst case, an individual operation may be as bad as O(n) and the height of the tree can also be O(n) in the worst case.

Splaying

- Given an internal node x of a binary search tree T, we splay x by moving x to the root of T through a sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and (if it exists) x's grandparent z. There are three cases that we consider.
  - **zig-zig**: The node x and its parent y are both left or right children. We replace z by x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.



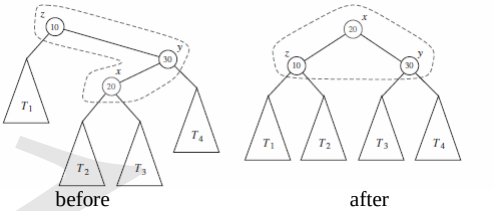
before after

Note: There is another *symmetric configuration* where x and y are left children [Draw before and after diagrams yourself]. Remember what we did for AVL trees.

Goodrich+Tamasia Book

Splaying

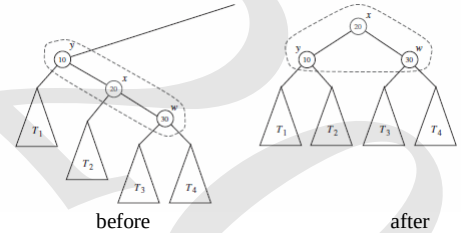
- **zig-zag**: One of x and y is a left child and the other is a right child. In this case, we replace z by x and make x have as its children the nodes y and z, while maintaining the inorder relationships of the nodes in T.



before after

Note: There is another *symmetric configuration* where x is a right child and y is a left child. Remember what we did for AVL trees.

- **zig:** x does not have a grandparent (or we are not considering x's grandparent for some reason). In this case, we rotate x over y, making x's children be the node y and one of x's former children w, so as to maintain the relative inorder relationships of the nodes in T.



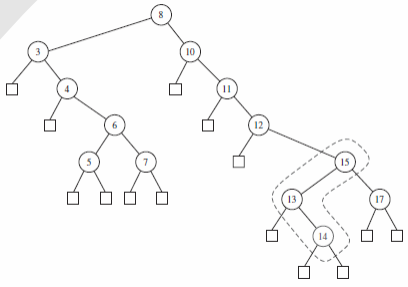
before after

Note: There is another *symmetric configuration* where x and w are left children. Remember what we did for AVL trees.

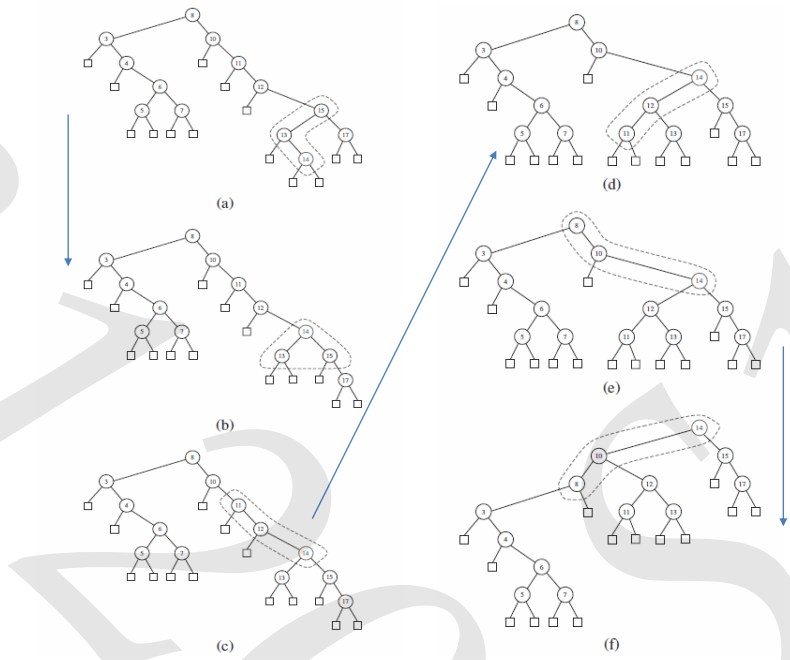
Note: Both zig-zig and zig-zag operations move a node x up 2 levels. The zig operation moves a node x up 1 level.

Guidelines for Splaying

- We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent.
- A splaying step consists of repeating these restructurings at x until x becomes the root of T. [**Caution**: this is not the same as a sequence of simple rotations that brings x to the root.]
- We will splay the node storing 14; what do we do first? zig-zag; why?



Note: x = 14, y = 13, z =15;
x has a parent and a grandparent.

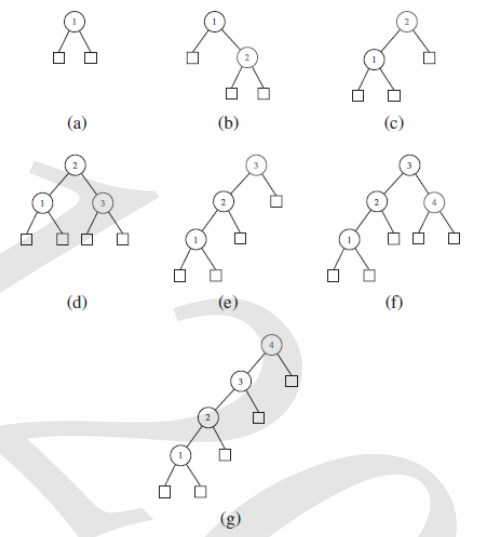


(a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step is a zig-zig; (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig.

Observations

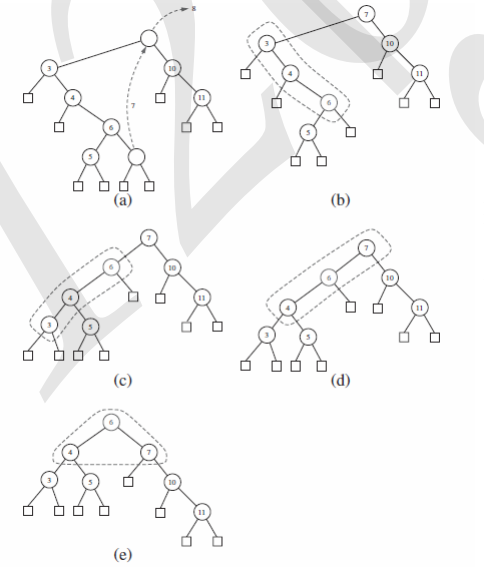
- After a zig-zig or zig-zag, the depth of x decreases by two, and after a zig the depth of x decreases by one. Thus, if x has depth d, splaying x consists of a sequence of $\lfloor d/2 \rfloor$ zig-zigs and/or zig-zags, plus one final zig if d is odd.
  - Since a single zig-zig, zig-zag, or zig affects a constant number of nodes, it can be done in O(1) time. Thus, splaying a node x in a binary search tree T takes time O(d), where d is the depth of x in T.
  - In other words, the time for performing a splaying step for a node x is asymptotically the same as the time needed just to reach that node in a top-down search from the root of T.
- **When to splay:**
  - When searching for key k, if k is found at a node x, we splay x, else we splay the parent of the external node at which the search terminates unsuccessfully.
  - When inserting key k, we splay the newly created internal node where k gets inserted.
  - When deleting a key k, we splay the parent of the node w that gets removed, that is, w is either the node storing k or one of its descendants. (Recall the deletion algorithm for binary search trees)
- In the **worst case**, the overall running time of a search, insertion, or deletion in a splay tree of height h is O(h), since the node we splay might be the deepest node in the tree. Moreover, it is possible for h to be $\Omega(n)$. Thus, from a worst-case point of view, a splay tree is not an attractive data structure.

Insertion Example



(a) 1 is inserted; (b) 2 is inserted; (c) after splaying; (d) after inserting 3; (e) after splaying; (f) after inserting 4; (g) after splaying.

Deletion Example



(a) the deletion of 8 from node r is performed by moving to r the key of the right-most internal node v, in the left subtree of r, deleting v, and splaying the parent u of v; (b) splaying u starts with a zig-zig; (c) after the zig-zig; (d) the next step is a zig; (e) after the zig.

Amortized Cost

- In an arbitrary sequence of intermixed searches, insertions, and deletions, each operation in a splay treetakes on average logarithmic time. We note that the time for performing a search, insertion, or deletion is proportional to the time for the associated splaying; hence, in our analysis, we consider only the splaying time.
- Let T be a splay tree with n keys, and let v be a node of T. We define the **size n(v) of v** as the number of nodes in the subtree rooted at v. Note that the size of an internal node is one more than the sum of the sizes of its two children. We define **the rank r(v) as r(v) = $\log_2 n(v)$.** [if e is a leaf node, n(e) = 1 and r(e) =0]
- Clearly, the root of T has the maximum size 2n + 1 and the maximum rank, log (2n + 1), while each external node has size 1 and rank 0.
- We use a method similar to the potential function. Say, we pay for the work in splaying a node x in T, and we assume that **one cyber-dollar pays for a zig**, while **two cyber-dollars pay for a zig-zig or a zig-zag.** Hence, the cost of splaying a node at depth *d* is *d* cyber-dollars.
- We keep a virtual (imaginary, not actually stored in the data structure) account, storing cyber-dollars, at each internal node of *T*. We make a payment for each operation.
- We have 3 cases to consider:
  - If the payment is equal to the splaying work, then we use it all to pay for splaying.
  - If the payment is greater than the splaying work, we deposit the excess in the accounts of several nodes.
  - If the payment is less than the splaying work, we make withdrawals from the accounts of several nodes to cover the deficiency.

Amortized Cost (contd.)

- We pay O(log n) cyber-dollars per operation to keep the system working, that is, to ensure that each node keeps a nonnegative account balance. We use a scheme in which transfers are made between the accounts of the nodes to ensure that there will always be enough cyber-dollars to withdraw for paying for splaying work when needed. We also maintain the following invariant:

  Before and after a splaying, each node v of T has r(v) cyber-dollars

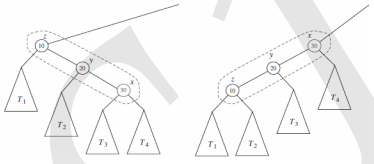
- Note that the we do not require us to endow an empty tree with any cyber-dollars (really?)
- **Notations:**
  - **(1)** Let **r(T) be the sum of the ranks of nodes of T**. To preserve the invariant after a splaying, we must make a payment equal to the splaying work plus the total change in r(T). We refer to a single zig, zig-zig, or zig-zag operation in a splaying as a **splaying sub step**.
  - **(2)** we denote the rank of a node *v* of *T* before and after a splaying sub step with $r(v)$ and $q(v)$ [we will need to compute the change]
  - **(3)** Let **δ** be the variation of $r(T)$ caused by a single splaying sub step (a zig, zig-zig, or zig-zag) for a node *x* in a splay tree *T*. Also, let **Δ** be the total variation of r(T) caused by splaying a node x at depth d.
- We need to use the formula **If $a > 0$, $b > 0$, and $c > a + b$, then $\log_2 a + \log_2 b \leq 2 \log_2 c - 2$.** (Can you show why it is true? Try!)
- What is the upper bound of **δ?**

$$(a-b)^2 \geq 0 \Rightarrow (a+b)^2/4 \geq ab \Rightarrow 2^{2\log_2(a+b)-2} \geq 2^{\log_2 a + \log_2 b}$$

Amortized Cost (contd.)

- **Claim:** Let **δ** be the variation of $r(T)$ caused by a single splaying sub step (a zig, zig-zig, or zig-zag) for a node x in a splay tree T, at depth d. Then
  - $\delta \leq 3(q(x) - r(x)) - 2$ if the substep is a zig-zig or zig-zag.
  - $\delta \leq 3(r(x) - r(x))$ if the substep is a zig.
- We consider each type of splaying substep. Recall that the size of each node is one more than the size of its two children, note that only the ranks of x, y, and z change in a zig-zig operation, where y is the parent of x and z is the parent of y.

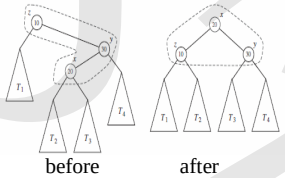
zig-zig: Note q(x) = r(z), q(y) ≤ q(x), r(y) ≥ r(x) [Check why!]
Then, δ = q(x) + q(y) + q(z) − r(x) − r(y) − r(z)
= q(x) + q(z) − r(x) − r(y)
≤ q(x) + q(z) − 2r(x)
Also, n(x) + n′(z) ≤ n′(x) ⇒ r(x) + q(z) ≤ 2q(x) – 2
⇒ q(z) ≤ 2q(x) – r(x) – 2.
Combining this with the previous inequality we get,
δ ≤ q(x) + (2q(x) – r(x) – 2) – 2r(x)
≤ 3(q(x) − r(x)) − 2



before after

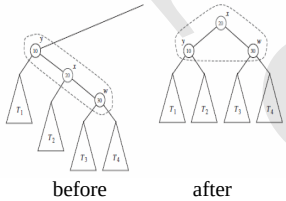
Amortized Cost (contd.)

zig-zag: By the definition of size and rank, only the ranks of x, y, and z change, where y denotes the parent of x and z denotes the parent of y. Also, q(x) = r(z) and r(x) ≤ r(y). Thus,
δ = q(x) + q(y) + q(z) − r(x) − r(y) − r(z)
≤ q(x) + q(z) − r(x) − r(y)
≤ q(x) + q(z) − 2r(x)
Again, n′(y) + n′(z) ≤ n′(x) or q(y) + q(z) ≤ 2q(x) – 2.
So, we get for δ
δ ≤ 2q(x) – 2 – 2 – 2r(x) ≤ 3(q(x) – r(x)) – 2.



before after

zig: In this case, only the ranks of x and y change, where y denotes the parent of x. Also, q(y) ≤ r(y) and q(x) ≥ r(x). Thus,
δ = q(y) + q(x) − r(y) − r(x)
≤ q(x) − r(x)
≤ 3(q(x) − r(x)).



before after

Bound the total variation of r(T) caused by splaying a node x

- δ is the variation of r(T) [with root t] caused by a single splaying sub step (a zig, zig-zig, or zig-zag) for a node *x* in a splay tree T. Also, Δ is the total variation of r(T) caused by splaying a node x at depth d. We show **$\Delta \leq 3(r(t) - r(x)) - d + 2$**.
  - Splaying node x consists of $p = \lceil d/2 \rceil$ splaying substeps, each of which is a zig-zig or a zig-zag, except possibly the last one, which is a zig if d is odd. Let $r_0(x) = r(x)$ be the initial rank of x, and for i = 1, . . . , p, let $r_i(x)$ be the rank of x after the i^{th} substep and δ_i be the variation of r(T) caused by the i^{th} substep.

$$\Delta = \sum_{i=1}^{p} \delta_i \leq \sum_{i=1}^{p} (3(r_i(x) - r_{i-1}(x)) - 2) + 2 = 3(r_p(x) - r_0(x)) - 2p + 2 \leq 3(r(t) - r(x)) - d + 2$$

  - If we make a payment of 3(r(t) − r(x)) + 2 cyber-dollars toward the splaying of node x, we have enough cyber-dollars to maintain the invariant, keeping r(v) cyber-dollars at each node v in T, and pay for the entire splaying work, which costs d dollars. Since the size of the root t is 2n+1, its rank r(t) = log(2n + 1). In addition, we have r(x) < r(t). Thus, the payment to be made for splaying is O(log n) cyber-dollars.
  - To complete our analysis, we have to compute the cost for maintaining the invariant when a node is inserted or deleted.

Bound . . . (contd.)

- When inserting a new node v into a splay tree with n keys, the ranks of all the ancestors of v are increased. Namely, let $v_0, v_i, \ldots, v_d$ be the ancestors of v, where $v_0 = v$, vi is the parent of v_{i-1}, and v_d is the root. For i = 1, . . . , d, let $n1(v_i)$ and $n(v_i)$ be the size of v_i before and after insertion and $r1(v_i)$ and $r(v_i)$ be the rank of v_i before and after insertion. We have $n1(v_i) = n(v_i) + 1$
- Also, since $n(v_i)+1 \leq n(v_{i+1})$, for i = 0, 1, . . . , d−1, we have the following for each i in this range: $r1(v_i) = \log(n1(v_i)) = \log(n(v_i) + 1) \leq \log(n(v_{i+1})) = r(v_{i+1})$.
- Thus, the total variation of r(T) caused by the insertion is

$$\sum_{i=1}^{d} (r1(v_i) - r(v_i)) \leq r1(v_d) + \sum_{i=1}^{d-1} r(v_{i+1}) - r(v_i)) = r1(v_d) - r(v_0) \leq \log(2n+1)$$

- Thus, a payment of O(log n) cyber-dollars is sufficient to maintain the invariant when a new node is inserted.
- When *deleting* a node v from a splay tree with n keys, the ranks of all the ancestors of v are decreased. Thus, the total variation of r(T) caused by the deletion is negative, and we do not need to make any payment to maintain the invariant.

Bound … (contd.)

- Theorem: Consider a sequence of m operations on a splay tree, each a search, insertion, or deletion, starting from an empty splay tree with zero keys. Also, let n_i be the number of keys in the tree after operation i, and n be the total number of insertions. The total running time for performing the sequence of operations is $O(m + \Sigma_{i=1 \ldots m} \log n_i)$
- Thus, the amortized running time of performing a search, insertion, or deletion in a splay tree is O(log n), where n is the size of the splay tree at the time. Thus, a splay tree can achieve logarithmic time amortized performance for searching and updating. This amortized performance matches the worst-case performance of AVL trees, red-black trees, and B trees, but it does so using a simple binary tree that does not need any extra balance information stored at each of its nodes. In addition, splay trees have a number of other interesting properties that are not shared by these other balanced search trees.
- Consider a sequence of m operations on a splay tree, each a search, insertion, or deletion, starting from a tree T with no keys. Also, let f(i) denote the number of times the item i is accessed in the splay tree, that is, its frequency, and let n be total number of items. Assuming that each item is accessed at least once, then the total running time for performing the sequence of operations is **$O(m + \Sigma_{i=1 \ldots n} \log (m/f(i)))$**
- The remarkable thing about this theorem is that it states that the amortized running time of accessing an item i is O(log (m/f(i))). For example, if a sequence of operations accesses some item i as many as m/4 times, then the amortized running time of each of these accesses is O(1) when we use a splay tree. Thus, an additional nice property of splay trees is that they can "adapt" to the ways in which items are being accessed so as to achieve faster running times for the frequently accessed items.