Tutorial 4

Dynamic Set: Amortized Analysis

- Review
  - Binary tree
    - Complete binary tree
    - Full binary tree
  - 2-tree P115
    - Unlike common binary tree, the base case is not an empty tree, but a external node
  - Heap
  - Binary search tree (BST)
  - Balanced BST
    - Red-Black tree
    - …

- In the classes, we learned the following data structures
  - BST
  - Hash Table:
    - Closed address: Linked list
    - Open address: Rehashing
  - Dynamic Set
    - Union-Find
    - wUnion-cFind

Amortized Analysis

- Analyze the cost of a sequence of operations
  - The total cost is averaged over all operations performed
  - Some of the operations might be expensive
  - The average cost is small
- Average performance in the worst case
- Gives an upper bound

- Amortized Analysis vs. Average-case analysis
  - No probability is involved
  - Guarantee the average performance of each operation in the worst case
- Amortized Analysis vs. Adversary Analysis
  - Adversary gives worst case lower bound, Amortized gives average case upper bound
  - Adversary applies for all algorithms theoretically, Amortized only applies in some special algorithms

Example: Stack Operations

- Consider the following two stack operations:
  - Push: push an object onto the stack
  - PopAll: pop all object from the stack
- There are n operations, what is the upper bound of total cost?

- Solution 1: Aggregate method
- We can prove the total cost of n operations in the worst cast is O(n)
  - The total cost of PopAll operations <= the total cost of Push operations
  - The total cost of Push <=n
  - Thus the total cost of n operations <=2n
- So, the average cost is O(n)/n=O(1) in the worst case

- Solution 2: Accounting method
  - Actual cost:
    - Push:1
    - PopAll: k, where k is the number of objects in the stack
  - Accounting cost
    - Push: 1
    - PopAll: -k
  - Amortized cost:
    - Push: 2
    - PopAll: 0
- We can prove the total accounting cost is nonnegative
- Thus, for any sequence of n operations, the total amortized cost is an upper bound, which is 2n

- Solution 3: Potential method
- Define the potential function Φ on a stack to be the number of objects in the stack
- Obviously it satisfies:

  $$\Phi(D_0)=0; \ \Phi(D_i)>=0$$
- Amortized cost:
  - If a Push operation
    - Potential difference is: $\Phi(D_i)-\Phi(D_{i-1})=(k+1)-k=1$
    - So the amortized cost of Push operation is

      $c'_i=c_i+\Phi(D_i)-\Phi(D_{i-1})=1+1=2$
  - If a PopAll operation
    - Potential difference is: $\Phi(D_i)-\Phi(D_{i-1})=0-k=-k$
    - So the amortized cost of PopAll operation is

      $c'_i=c_i+\Phi(D_i)-\Phi(D_{i-1})=k+(-k)=0$
- So, the total amortized cost is an upper bound, which is 2n

Types of Amortized Analysis

- Aggregate method
- Accounting method
- Potential method

(1) Aggregate Method

- Consider a sequence of n operations
- Show that for all n, the n operations takes worse-case time T(n) in total
- Therefore each operation is T(n)/n (amortized cost) in the worst case
- Note that this amortized cost applies to each operation, even there are several types of operations
- Aggregate Method vs. Accounting method and Potential method
    - The accounting method and potential method may assign different amortized cost to different type of operations

(2) Accounting Method

- Assign different charges to different operations
- Some operations are charged more or less than their actually cost
- Amortized equation:

  amortized cost = actual cost + accounting cost

  - The total accounting cost is the difference between the total amortized cost and the total actual cost
  - The total accounting cost must be nonnegative at all times
- Assume
  - Each operation i has an actual cost C_i
  - We charge operation i a fictitious amortized cost C'_i
  - We must ensure that

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} C'_i$$

- Thus, the total amortized costs provide an upper bound on the total actual costs

(3) Potential Method

- Idea: view the bank account as the potential energy of the data structure
- Framework
  - Start with an initial data structure D_0 (n operations are going to perform on D_0)
  - Each operation i has an actual cost C_i , D_i is the data structure that results after applying the ith operation on data structure D_{i-1}
  - Operation i transforms D_{i-1} to D_i
  - Define a potential function Φ: $\{D_i\} \rightarrow R$, such that $\Phi(D_0)=0$ and $\Phi(D_i) \geqslant 0$ for all i.
  - The amortized cost C'_i is defined as

$$C'_i = C_i + \Phi(D_i) - \Phi(D_{i-1})$$

The amortized cost bounded the actual cost

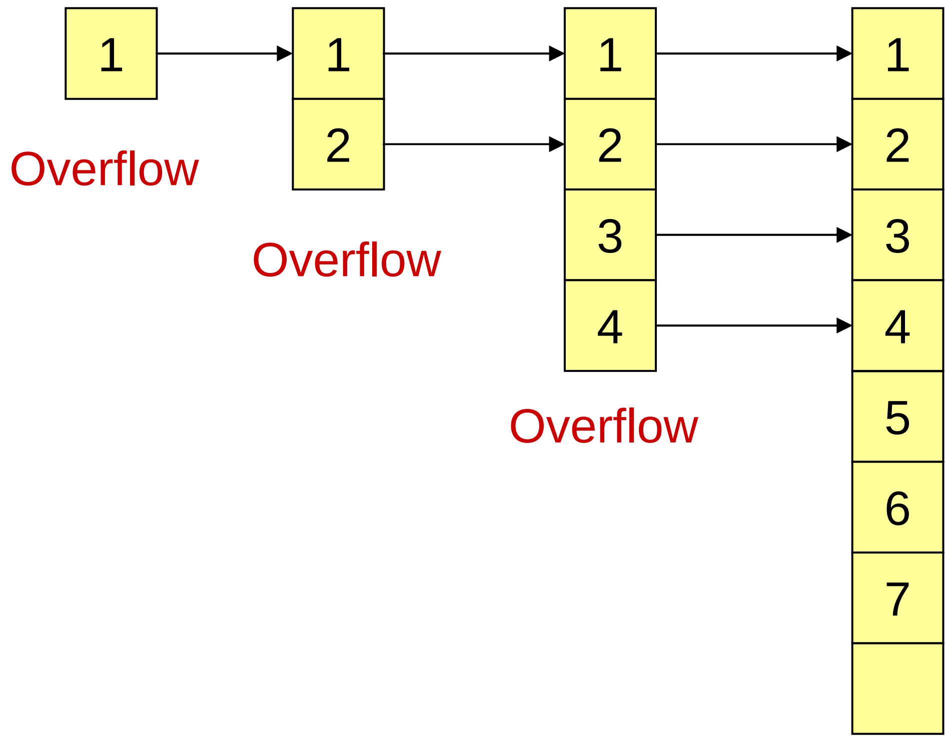
$$\begin{aligned}\sum_{i=1}^{n} C_i' &= \sum_{i=1}^{n}(C_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} C_i + D_n - D_0 \\ &\geq \sum_{i=1}^{n} C_i\end{aligned}$$

Choose an appropriate potential function is important!

Example: Dynamic Table

- Goal: Make a table as small as possible
- Problem: What if we don’t know the proper size in advance?
- Solution: Dynamic tables
  - Each time we insert a new element into the table, we check whether there is an empty unit. If so, the insertion needs only 1 operate.
  - If the table overflows, we “grow” it by allocating a new, larger table (double size of the original one). Move all items from the old table into the new one, insert the new element, and then free the storage for the old table. The total number of operations is: 1+(the size of the old table).

- Insert 1
- Insert 2
- Insert 3
- Insert 4
- Insert 5
- Insert 6
- Insert 7



Worst-case analysis

- Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$?
- WRONG!

- Solution 1: aggregate method
- Let C_i= the cost of the *ith* insertion

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

$$\boxed{=1+2^{\lg(i-1)}}$$

i	1	2	3	4	5	6	7	8	9	10
table size	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1
Overhead	0	1	2	0	4	0	0	0	8	0

- The cost of n insertion is:

$$\sum_{i=1}^{n} C_i = n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \le n + 2n = 3n = \Theta(n)$$

- Solution 2: accounting method
- Actual cost: $C_i = \begin{cases} 1+2^{\lg(i-1)} & \text{if i-1 is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$
- Accounting cost: $a_i = \begin{cases} 2-2^{\lg(i-1)} & \text{if i-1 is an exact power of 2} \\ 2 & \text{otherwise} \end{cases}$
- Amortized cost: C_i'=3
- we can prove $\sum_{i=1}^{n} C_i \le \sum_{i=1}^{n} C_i'$
- Thus, a upper bound of n insertion is: $\sum_{i=1}^{n} C'_i = 3n = \Theta(n)$

i	1	2	3	4	5	6	7	8	9	10
table size	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1
Overhead	0	1	2	0	4	0	0	0	8	0
C'_i	2	3	3	3	3	3	3	3	3	3
Store	1	2	2	2	2	2	2	2	2	2
Consume	0	-1	-2	0	-4	0	0	0	-8	0

- Solution 3: Potential method
- Define a potential function $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$, (assume $2^{\lceil \lg 0 \rceil} = 0$)
- We have:
  - If i-1 is an exact power of 2

$$C'_i = i + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg (i-1) \rceil})$$
$$= i + 2 - (2^{\lceil \lg i \rceil} - 2^{\lceil \lg (i-1) \rceil}) = i + 2 - (2(i-1) - (i-1)) = 3$$

  - If i-1 is not an exact power of 2

$$C'_i = 1 + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg (i-1) \rceil})$$
$$= 1 + 2 - (2^{\lceil \lg i \rceil} - 2^{\lceil \lg (i-1) \rceil}) = 3$$

  - So, we have an upper bound: $\sum_{i=1}^{n} C'_i = 3n = \Theta(n)$

i	1	2	3	4	5	6	7	8	9	10
table size	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1
Overhead	0	1	2	0	4	0	0	0	8	0
C'_i	2	3	3	3	3	3	3	3	3	3
$\Phi(D_i)$	1	2	2	4	2	4	6	8	2	4
$\Delta\ \Phi$	1	1	0	2	-2	2	2	2	-6	2