CSCI 104
Log Structured Merge Trees

Mark Redekopp
Series Summation Review

• Let $n = 1 + 2 + 4 + \ldots + 2^k = \sum_{i=0}^{k} 2^i$. What is $n$?
  - $n = 2^{k+1} - 1$

• What is $\log_2(1) + \log_2(2) + \log_2(4) + \log_2(8) + \ldots + \log_2(2^k)$
  $= 0 + 1 + 2 + 3 + \ldots + k = \sum_{i=0}^{k} i$
  - $O(k^2)$

• So then what if $k = \log(n)$ as in:
  $\log_2(1) + \log_2(2) + \log_2(4) + \log_2(8) + \ldots + \log_2(2^{\log(n)})$
  - $O(\log^2 n)$

Arithmetic series:
$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$$

Geometric series
$$\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1} = \theta(c^n)$$
Merge Trees Overview

- Consider a list of (pointers to) arrays with the following constraints
  - Each array is sorted *though no ordering constraints exist between arrays*
  - The array at list index k is of exactly size $2^k$ or empty

Note: These are the keys for a set (or key,value pairs for a map)
Merge Trees Size

• Define...
  – n as the # of keys in the entire structure
  – k as the size of the list (i.e. positions in the list)
• Given list of size k, how many total values, n, may be stored?
  – Let $n = 1 + 2 + 4 + \ldots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i$. What is n?
• $n=2^k-1$

An array at list location k can be of size $2^k$ or empty

Note: These are the keys for a set (or key,value pairs for a map)
Merge Trees Find Operation

- To find an element (or check if it exists)
- Iterate through the arrays in order (i.e. start with array at list position 0, then the array at list position 1, etc.)
  - In each array perform a binary search
- If you reach the end of the list of arrays without finding the value it does not exist in the set/map

Note: These are the keys for a set (or key,value pairs for a map)
Find Runtime

• What is the worst case runtime of find?
  - When the item is not present which requires, a binary search is performed on each list
• \( T(n) = \log_2(1) + \log_2(2) + \ldots + \log_2(2^{k-1}) \)
  - \( = 0 + 1 + 2 + \ldots + k-1 = \sum_{i=0}^{k-1} i \)
  - \( = O(k^2) \)
• But let's put that in terms of the number of elements in the structure (i.e. \( n \))
  - Recall \( k = \log_2(n+1) \)
• So find is \( O(\log_2(n)^2) \)
Improving Find's Runtime

• While we might be okay with \([\log(n)]^2\), how might we improve the find runtime in the general case?
  – Hint: I would be willing to pay \(O(1)\) to know if a key is not in a particular array without having to perform find

• A Bloom filter could be maintained alongside each array and allow us to skip performing a binary search in an array
Insertion Algorithm

- Let $j$ be the smallest integer such that array $j$ is empty (first empty slot in the list of arrays)
- An insertion will cause
  - Location $j$'s array to become filled
  - Locations 0 through $j-1$ to become empty
Insertion Algorithm

- Starting at array 0, iteratively merge the previously merged array with the next, stopping when an empty location is encountered.
Insert Examples

insert(4)
Cost = 1 / Stop @ 0

insert(2)
Cost = 2 / Stop @ 1

insert(5)
Cost = 1 / Stop @ 0

insert(19)
Cost = 4 / Stop @ 2

insert(8)
Cost = 1 / Stop @ 0

insert(7)
Cost = 2 / Stop @ 1

insert(12)
Cost = 1 / Stop @ 0
Insertion Runtime: First Look

- **Best case?**
  - First list is empty and allows direct insertion in $O(1)$
- **Worst case?**
  - All list entries (arrays) are full so we have to merge at each location
  - In this case we will end with an array of size $n=2^k$ in position $k$
  - Also recall merging two arrays of size $m$ is $\Theta(m)$
  - So the total cost of all the merges is $1 + 2 + 4 + 8 + \ldots + 2^k = \Theta(2^{k+1}) = \Theta(n)$
- But if the worst case occurs how soon can it occur again?
  - It seems the costs vary from one insert to the next
  - This is a good place to use amortized analysis
Total Cost for N insertions

• Reminder: Insert stopping at location $k$ requires $O(2^{k+1})$ merge steps

• Total cost of $n=16$ insertions:
  - Stop at: 0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,4
  - Cost: $2^1 + 2^2 + 2^1 + 2^3 + 2^1 + 2^2 + 2^1 + 2^4 + 2^1 + 2^2 + 2^1 + 2^3 + 2^1 + 2^2 + 2^1 + 2^5$

  \[= 2^1 \cdot n/2 + 2^2 \cdot n/4 + 2^3 \cdot n/8 + 2^4 \cdot n/16 + 2^5 \cdot 1\]

  \[= n + n + n + n + 2 \cdot n\]

  \[= n \cdot \log_2(n) + 2n\]

• Amortized cost = Total cost / $n$ operations
  - $\log_2(n) + 2 = O(\log_2(n))$
Amortized Analysis of Insert

- We have said when you end (place an array) in position k you have to do $O(2^{k+1})$ work for all the merges.
- How often do we end in position $k$:
  - The $0^{th}$ position will be free with probability $\frac{1}{2}$ ($p=0.5$).
  - We will stop at the $1^{st}$ position with probability $\frac{1}{4}$ ($p=0.25$).
  - We will stop at the $2^{nd}$ position with probability $\frac{1}{8}$ ($p=0.125$).
  - We will stop at the $k^{th}$ position with probability $\frac{1}{2^{k+1}} = 2^{-(k+1)}$.
- So we pay $2^{k+1}$ with probability $2^{-(k+1)}$.
- Suppose we have $n$ items in the structure (i.e. max $k$ is $\log_2 n$) what is the expected cost of inserting a new element:
  - $\sum_{k=0}^{\log(n)} 2^{k+1}2^{-(k+1)} = \sum_{k=0}^{\log(n)} 1 = \log(n)$.
Summary

• Variants of log structured merge trees have found popular usage in industry
  – Starting array size might be fairly large (size of memory of a single server)
  – Large arrays (from merging) are stored on disk

• Pros:
  – Ease of implementation
  – Sequential access of arrays helps lower its constant factors

• Operations:
  – Find = $\log^2(n)$
  – Insert = Amortized $\log(n)$
  – Remove = often not considered/supported