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

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

Geometric series
\[
\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1} = \theta(c^n)
\]

Arithmetic series:
\[
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2} = \theta(n^2)
\]
Merge Two Sorted Lists

• Consider the problem of merging two n/2 size sorted lists into a new combined sorted list

• Can be done in O(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 $n$?

• $n = 2^k - 1$
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, \( n=2^k - 1 \), so \( k = \log_2(n+1) \)
• 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 stopping at location $k$ requires $1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^{k+1}-1 = O(2^{k+1})$ merge steps.

```
insert(19)
```

List 0 is full so merge two arrays of size 1

List 1 is full so merge two arrays of size 2

Merge
Insert Examples

<table>
<thead>
<tr>
<th>Insert</th>
<th>DB State</th>
<th>Cost</th>
<th>Stop @</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(4)</td>
<td>0 → 1 → NULL</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>insert(2)</td>
<td>0 → 1 → NULL</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>insert(5)</td>
<td>0 → 1 → NULL</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>insert(19)</td>
<td>0 → 1 → NULL</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>insert(8)</td>
<td>0 → 1 → NULL</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>insert(7)</td>
<td>0 → 1 → NULL</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>insert(12)</td>
<td>0 → 1 → NULL</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
• **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 sorted arrays of size \(m/2\) 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
  \[1+2+4+\ldots+2^{k-1}+2^k = 2^{k+1}-1 = O(2^{k+1})\] 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:
    \[2^1+2^2+2^1+2^3+2^1+2^2+2^1+2^4+2^1+2^2+2^1+2^3+2^1+2^2+2^1+2^5\]
  - \[=2^1 \times n/2 + 2^2 \times n/4 + 2^3 \times n/8 + 2^4 \times n/16 + 2^5 \times 1\]
  - \[= n + n + n + n + 2 \times n\]
  - \[= n \times \log_2(n) + 2n\]

- 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 0th position will be free with probability $\frac{1}{2}$ (p=0.5).
  - We will stop at the 1st position with probability $\frac{1}{4}$ (p=0.25).
  - We will stop at the 2nd 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 $O(2^{k+1})$ 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
• More reading:
  – http://www.benstopford.com/2015/02/14/log-structured-merge-trees/