1 Wrapping up on hashing

Recall from last lecture that we showed that when using hashing with chaining of \( n \) items into \( m \) bins where \( m = \Theta(n) \) and hash function \( h : [u] \rightarrow m \) that if \( h \) is from \( C \log n \) \( \log \log n \) wise family, then no linked list has size \( > \frac{C \log n}{\log \log n} \).

1.1 Using two hash functions

Azar, Broder, Karlin, and Upfal '99 [1] then used two hash functions \( h, g : [u] \rightarrow [m] \) that are fully random. Then ball \( i \) is inserted in the least loaded of the 2 bins \( h(i), g(i) \), and ties are broken randomly.

**Theorem:** Max load is \( \leq \frac{\log \log n}{\log d} + O(1) \) with high probability. When using \( d \) instead of 2 hash functions we get max load \( \leq \frac{\log \log n}{\log d} + O(1) \) with high probability.

1.2 Breaking \( n \) into \( \frac{n}{d} \) slots

Vöcking '03 [2] then considered breaking our \( n \) buckets into \( \frac{n}{d} \) slots, and having \( d \) hash functions \( h_1, h_2, \ldots, h_d \) that hash into their corresponding slot of size \( \frac{n}{d} \). Then insert(i), loads i in the least loaded of the \( h(i) \) bins, and breaks tie by putting i in the left most bin.

This improves the performance to Max Load \( \leq \frac{\log \log n}{d\phi_d} \) where \( 1.618 = \phi_2 < \phi_3 \ldots \leq 2. \)

1.3 Survey: Mitzenmacher, Richa, Sitaraman [3]

**Idea:** Let \( B_i \) denote the number of bins with \( \geq i \) balls.

Let \( H(b) = \) height of ball \( b \) in its bin.

Then in order for a bin to grow to height \( i+1 \), both bins that ball \( b \) hashed to had to be at least height \( i \), so we get \( \mathbb{P}(H(b) \geq i + 1)) \leq \left( \frac{B_i}{n} \right)^2. \)

This gives us \( \mathbb{E}(B_{i+1}) \leq \mathbb{E}(\text{Number of balls with height } \geq i + 1) \leq \frac{B_i^2}{n} \), so \( \frac{B_{i+1}}{n} \leq \left( \frac{B_i}{n} \right)^2. \)

As a base case, since there are only \( n \) balls we know that at most \( \frac{n}{2} \) bins can have a height of 2 balls, giving us \( \frac{B_2}{n} \leq \frac{1}{2}. \) Then using our recursive definition from above we get \( \frac{B_3}{n} \leq \frac{1}{2^2}, \frac{B_4}{n} \leq \frac{1}{2^3}, \ldots \), and can get the rule \( \frac{B_{i+1}}{n} \leq \frac{1}{2^i}. \)
We then need to show that $P_i$, that doesn't matter. We then pick a uniform random variable $U$ in $[0,1]$. If $U$ is in $[0 \leq X \leq 1]$, we draw a number of heads when flipping $n$ $p$-biased coins.

Proof: Let $t_i = e^{2^i}$, and $\alpha_{i+1} = e \cdot \frac{\alpha_i^2}{n}$ for $i \geq 6$. Let $E_i$ be the event that $B_i \leq \alpha_i$.

We then need to show that $P(\forall i \geq 6, E_i) \geq 1 - \frac{1}{\poly(n)}$.

Claim 1: $P((B_{i+1} > \alpha_{i+1}) \cap j=1 E_j) \leq P(E_0)P(E_7|E_6)P(E_8|E_7, E_6)\ldots$

$$P_{\forall i \leq i} E_{ij} \cap E_j = \frac{P_{\forall i \leq i} E_{ij} \cap E_j}{P_{\forall i \leq i} E_{ij}} \leq \frac{P_{\forall i \leq i} E_{ij}}{P_{\forall i \leq i} E_{ij}}$$

The denominator here is close to one, and the numerator is upper bounded by $P(Bin(n, (\alpha_i/n)^2) > \alpha_{i+1})$.

Consider the following experiment. We insert the balls one by one into the bins. Number the balls $1, \ldots, n$ and assign them to bins in increasing order of ball number. In our experiment we maintain two variables: $X$ and $Y$. $X$ is a Boolean "flag" that starts as TRUE and becomes FALSE if any of $E_1, \ldots, E_i$ ever fail to hold. $Y$ is a counter: it counts the number of balls of height at least $i+1$. We also increment $Z$ iff $U$ we drew (and potentially set $Y$ to 0 if $X$ got set to FALSE).

Otherwise, if $X$ is true, then we know there are exactly $P(E_i > \alpha_i)$ random variable $Z$ starting at 0 and gradually being incremented as we insert our balls, and $Z$ and our $X,Y$ will be defined on the same probability space so that $A$ and $B$ are dependent and easy to compare, but marginally $A$ and $B$ have the correct probabilities of occurring). When we assign ball $j$, there are two cases. Either $X$ is already FALSE, in which case we just set $Y$ to 0 (we already know A failed) and we increment $Z$ with probability $p = \frac{\alpha_i^2}{n}$. Otherwise, if $X$ is true, then we know there are exactly $t \leq \alpha_i$ bins of load $\geq i$ for some $t$. We label these bins $1, \ldots, t$ in this time step (and the other bins are labeled $t+1, \ldots, n$ for some ordering that doesn’t matter). We then pick a uniform random variable $U$ in $[0,1]$. If $U$ is in $[0, \frac{1}{n^2}]$ then we imagine the two bins $j$ got hashed to are 1.1. If it’s in $[\frac{1}{n^2}, \frac{2}{n^2}]$, then we imagine it got hashed to bins 1 and 2. Etc. We slice up $[0,1]$ into $n^2$ buckets of size exactly $\frac{1}{n^2}$ each such that the first $t^2$ buckets all correspond to the $t^2$ different ways we can hash to two heavy bins (i.e. bins with load at least $i$). The remaining buckets of size $\frac{1}{n}$ in $[0,1]$ then correspond to all the other $t^2$ different hash possibilities for ball $j$. We then adjust $X$ as needed if some condition failed based on the $U$ we drew (and potentially set $Y$ to 0 if $X$ got set to FALSE).

We also increment $Z$ iff $U \leq p$. ***This is where the coupling happens***, since $Z$ and $X,Y$ are now defined on the same probability space based on these $U$ random variables over our iterations! AND THE KEY THING: event "A" can only happen if "B" also happened, since the way we coupled $Y$ and $Z$ maintains the invariant that $Y \leq Z$ always (note we hold $Y$ at 0 if $X$ is ever set to FALSE)!

Thus $P(A) \leq P(B)$.

Claim 2: $P(Bin(n, (\alpha_i/n)^2) > \alpha_{i+1})$. For those not aware Bin(n,p) refers to the distribution of the number of heads when flipping $n$ $p$-biased coins.

Proof: Let $X_i$ be an indicator for $C_i$ being heads. Then $E(X_i) = \frac{\alpha_i^2}{n^2}$. Then using the Chernoff
bound we get $\mathbb{P}(\sum X_i > e \times n \times p) \leq e^{-c n^2 / n}$

## 2 Amortization

### 2.1 Definition

If a data structure supports operations $A_1, A_2, ..., A_r$, we say that the amortized cost of $A_j$ is $t_j$ if $\forall$ sequences of operations with $n_i$ operations of type $A_i$ the total runtime is $\leq \sum_j n_j t_j$.

### 2.2 Potential function:

Let $\Phi$ map the states of the data structures to $\mathbb{R}_{>0}$ (the non-negative real numbers), and $\Phi$ of the empty data structure is defined to be 0. Then the valid amortized cost of an operation $t_{\text{actual}} + \Delta \Phi = t_{\text{actual}} + \Phi(\text{after op}) - \Phi(\text{before op})$

Then the total amortized cost of a sequence of ops $= \sum_{j=1}^{R} t_{\text{actual}(j)} + \Phi(\text{time } j) - \Phi(\text{time } j - 1) = \sum_{j} (\text{actual time for jth operation}) + \Phi(\text{final}) - \Phi(0)$. Since $\Phi(0) = 0$ and $\Phi(\text{final}) \geq 0$ then our equation above for total amortized cost is $\geq$ the total actual time.

### 2.3 Y-fast Tries

Recall from the first lecture that a Y-fast trie consisted of an x-fast trie of $\frac{n}{\theta(\log w)}$ groups each pointing to a BJT of $[\frac{w}{2}, 2w)$ elements, and we said this supports query of $\theta(\log w)$ and an amortized insert time of $\theta(\log w)$.

From now on, let $t$ refer to actual time and $\tilde{t}$ refers to amortized time.

Now lets analyze the Y-fast trie insertion using our potential function. We have $\Phi(\text{State of the structure}) = \sum_{\text{grps}} ((\text{Size of group } j) - w)$

For the query operation we get $\tilde{t}_{\text{query}} = t_{\text{query}} + \Delta \Phi = \tilde{t}_{\text{query}} = t_{\text{query}} + 0 = \Theta(\log w)$.

For $\tilde{t}_{\text{insert}} = t_{\text{insert}} + \Delta \Phi$. Now consider the two cases (either we have to split a group of 2w into 2 groups of size w or we don’t). In the first case $t_{\text{insert}} = \Theta(\log w) + w$ and $\Delta \Phi = -w$ giving $\tilde{t}_{\text{insert}}$ is $\Theta(\log w)$. In the second case $t_{\text{insert}} = \Theta(\log w)$ and $\Delta \Phi = 1$ giving $\tilde{t}_{\text{insert}} = \Theta(\log w)$.

## 3 Heaps:

### 3.1 Definition:

Heaps maintain keys with comparable values subject to the follow operations:
1. deleteMin(): report the smallest item and delete it from the heap
2. decKey(*P, v'): reduce value of p to V’
3. insert(K, V): insert key K with value V to the heap

Binary Heaps: John Williams ’64 [4] published the binary heap which supports \( t_I \) (insertion time) = \( t_D \) (delete min) = \( t_K \) (dec key) in \( O(\log n) \).

Binomial Heaps: Vuillemin ’78 [5] published the Binomial Heap which supports \( \tilde{t}_I = O(1) \) and \( t_D = t_K = O(\log n) \).

Fibonacci Heaps: Fredman and Tarjan ’87 [6] published the Fibonacci Heap \( \tilde{T}_I = \tilde{T}_K = O(1) \), and \( t_D = O(\log n) \).

Strict Fibonacci Heaps: Brodal, Lagogiannis, and Tarjan ’12 [7] published the Strict Fibonacci Heap \( T_I = T_K = O(1) \), and \( t_D = O(\log n) \).

Then Thorup ’07 [8] showed that in Word RAM sorting in \( nS(n) \Rightarrow \) there exists a heap with \( O(1) \) find min (note this is not the same as delete min, and \( O(S(n)) \) insert/delete time. As pset 1 showed, there exist sorting algorithms where \( S(n) = O(\log \log n) \) or \( O(\sqrt{\log \log n}) \).

\[ 3.2 \text{ Binomial Heaps} \]

A binomial heap stores its items in a forest. Each tree is given a "rank" equal to the number of children of the root, and has the requirement that a rank k tree has k subtrees of ranks 0,1,2,...,k-1. Each tree is also in heap order (meaning the parent’s value \( \leq \) all of it’s children’s values. The binomial heap is also subject to the invariant that there are \( \leq 1 \) rank-k tree in our forest for each k.

A heap of rank 0 is just a single element. A heap of rank 1 is just a parent and child node. A heap of rank 2 has one child leaf node, and one child that also has a child. And a heap of rank 3 has 3 children (one that’s a tree of rank 2, one a tree of rank 1, and one a tree of rank 0).

The heap then performs the following operations:

- **Insert**: Create a new rank 0 tree, and then keep merging to preserve our invariant.
- **Dec key**: Bubble up the node to maintain heap order.
- **Delete min**: Compare roots, cut the smallest root, and put its children as new top level trees then merge.

**Claim**: A rank k tree has \( 2^k \) nodes.

**Proof**: Induction.

This implies that the largest rank in the forest is \( \leq \log n \).

To show our runtime, let \( \Phi = \) number of trees in our forest.

\[
\tilde{t}_I = t_I + \Delta \Phi = 1 + (\text{number of trees merged}) + 1 - (\text{number of trees merged}) = O(1).
\]
\( \tilde{t}_k = t_k + \Delta \Phi = t_k + 0 = O(\log n). \)

\( \tilde{t}_d = t_d + \Delta \Phi = k + \text{(number of trees)} + \text{(at most number of trees)} = O(\log n + \log n + \log n) = O(\log n). \)

References


