%WHOAMI

- Hui Han Chin
- Raising Senior, Mathematical Sciences/ Computer Science
- Office Hours: MR 430-630
- Location: GHC 7110
ACTUAL COURSE GOALS

- Get a “good” grade
- Appear to know CS algorithm
- Able to implement simple algorithms
- Be motivated to do future work on algorithm
WHO ARE YOU?

- Who here is a ___________________?
  - Not registered
  - Repeat offender
  - Sop / Jnr / Snr / Master
  - CS / Math / Others

- Motivation
  - Fulfill requirements
  - Fulfill graduation requirements
  - For the interest
Recitations structure

- Administrivia
  - Boring stuff that I am required to say

- Topic review
  - What you should have known
  - What is on the lab

- Practice Problem(s)
  - “What might be in the quiz”

- Tips and Tricks
  - What might be useful in the final
RECITATION GROUND RULES

- Homework Questions will not be entertain during class time.
  - Come to office hours instead
- Leave interesting questions to the end of class
- Sleeping is welcome but not snoring. And please be discreet.
- Attendance is not compulsory
  - But quizzes are!
  - But don’t blame me if I give vital clues and you missed that recitations
Office Hours Ground Rules

- I will not tell you the answers
  - Come only after reading the questions
  - Come prepared
- I will not lead you in the wrong direction
  - If I pose you a question, it is usually a hint
- I more than willing to answer interesting questions
  - Homework problem suggestions are welcome!
HOMEWORK HANDIN

- Theory is handed in class
- Hand in folder
- FTP using putty/ SSH
- IDE / Eclipse / Vanilla Java
- Junit Testing.
ALGORITHMS

- Just a CS cooking recipe
- 1. To design algorithms for new problems.
- 2. To analysis the performance of algorithms.
- 3. To decide whether a problem is tractable.

Question

- How long does it takes to sort $10^9$ bytes (8 bit data struct)
**Computational Model**

- "A computational model is a mathematical model in computational science that requires extensive computational resources to study the behavior of a complex system by computer simulation" - wiki definitions.

- A model where
  - The basic computation step is determined.
  - The limits of computation are well defined.

- Infinite RAM model
ASYMPTOTIC ANALYSIS

- Big-O, Theta, Big-Omega
- Definitions
- Common Pitfalls
  - 1. Not check the computational model
  - 2. Using the wrong input size or type
  - 3. Assuming a loose bound.
  - 4. Ignoring special cases.
- Example
Solving Recurrences

- Solving Recurrences VS Theta bound
  - Solving requires finding the constants
  - Matters a lot in real world implementation
- 5 ways of solving recurrences
  - Induction
  - Loop unrolling
  - Tree
  - Masters Theorem?
  - Mathematica!
15-211: LAB 1: AAD

Slides stolen from Nathan Harmata (15121 / s09)
TODAY’S PLAN

- Develop a new data structure based on everything we’ve learned so far
- Use this data structure to solve the Dictionary Problem
- Analyze this data structure with respect to efficiency
BEFORE WE BEGIN...

- Recall the definition of the **Dictionary Problem**:
  - Design a way to:
    - Store stuff
    - Remove stuff
    - Check if stuff has been stored
THE DICTIONARY PROBLEM

More formally –

- Design a *data structure* that supports the following operations:
  - `add(e)` – make `e` a member
  - `remove(e)` – ensure `e` is not a member
  - `contains(e)` – check for membership of `e`
THE DICTIONARY PROBLEM

Question:

- add(e) – make e a member
- remove(e) – ensure e is not a member
- contains(e) – check for membership of e

Is a ... a solution to the **Dictionary Problem**?
BY THE WAY...

- We’ve already encountered at least two explicit solutions to the Dictionary Problem:
  - FastLinkedLists – aka “Skip Lists”
    - insert, delete, contains
  - HashSets
    - add, remove, contains

add(e) – make e a member
remove(e) – ensure e is not a member
contains(e) – check for membership of e
LET’S GET MOTIVATED

- Arrays are pretty cool, so let’s try to solve the Dictionary Problem by maintaining a sorted dynamic array structure

[1,5,8,9]  
Hey look, it’s sorted!
IDEA 1: DYNAMIC SORTED ARRAY IMPLEMENTATION OF ADD

- add(e) – make a new array that is one size bigger, and copy e and all the elements into it so that the new array is sorted

ex: add(6) on

\[
\begin{bmatrix}
1, 5, 8, 9 \\
_, _, _, _, _ \\
1, 5, 6, 8, 9
\end{bmatrix}
\]
**Idea 1: Dynamic Sorted Array**

**Implementation of remove**

- **remove(e)** – make a new array that is one size smaller, and copy all the elements except for `e` into so the new array is sorted

Example: remove(6) on

```
[ 1 , 5 , 6 , 8 , 9 ]
[ _ , _ , _ , _ , _ ]
[ 1 , 5 , 8 , 9 ]
```
Idea 1: Dynamic Sorted Array
Implementation of contains

- contains(e) – binary search the array

ex: contains(1) on 
[ 1 , 5 , 8 , 9 ]

[ 1 , 5 , 8 , 9 ]

[ 1 , 5 , 8 , 9 ]

[ 1 , 5 , 8 , 9 ]
**Idea 1: Dynamic Sorted Array Efficiency Analysis?**

- Suppose our dictionary has $N$ elements. What is the cost of:

  - $\text{add}(e)$ – make a new array that is one size bigger, and copy $e$ and all the elements into it so the new array is sorted

  - $\text{remove}(e)$ – make a new array that is one size smaller, and copy all the elements except for $e$ into so the new array is sorted

  - $\text{contains}(e)$ – binary search the array
AN OBSERVATION

For large $N$, add(e) and remove(e) are pretty expensive.

That’s because $O(n)$ is an increasing polynomial!
In general, would you rather do all that stuff (like binary search and array copying) on small arrays or big arrays?

Small arrays are ez!!!
Idea 2: A bunch of small dynamic sorted arrays

Let’s just maintain a bunch of sorted arrays. Whenever we do something, we try to do it with the smallest array first (because that would be the least expensive).

[ 1 , 5 ]  
[ 2 , 4 ]  
[ 3 , 6 , 7 ]
**IDEA 2: A BUNCH OF SMALL DYNAMIC SORTED ARRAYS**

- **add(e)** - insert e in the smallest array

ex: **add(8)** on

\[
\begin{array}{c}
[1, 5] \\
[2, 4] \\
[3, 6, 7]
\end{array}
\]
IDEA 2: A BUNCH OF SMALL DYNAMIC SORTED ARRAYS

- contains(e) - look for e in each of the arrays, starting with the smallest array

ex: contains(7) on

\[ \begin{array}{c}
\text{[ 2, 4 ]} \\
\text{[ 1, 5, 8 ]} \\
\text{[ 3, 6, 7 ]}
\end{array} \]
IDEA 2: A BUNCH OF SMALL DYNAMIC SORTED ARRAYS

- remove(e) - look for e, starting with the smallest array. If we find it, we replace that array with a new one that doesn’t contain e

ex: remove(2) on

\[
\begin{align*}
&\begin{bmatrix} 2 \end{bmatrix} 4 \\
&\begin{bmatrix} 1, 5, 8 \end{bmatrix} \\
&\begin{bmatrix} 3, 6, 7 \end{bmatrix}
\end{align*}
\]
IDEA 2: A BUNCH OF DYNAMIC SORTED ARRAYS
EFFICIENCY ANALYSIS?

- Suppose our dictionary has \( N \) elements, in \( M \) arrays \((A_1, A_2, \ldots, A_m)\) and the length of array \( A_i \) is \( L_i \). What is the cost of:

- add(e) – insert \( e \) in the smallest array

\[ O(L_{\text{smallest array}}) \]
IDEA 2: A BUNCH OF DYNAMIC SORTED ARRAYS

EFFICIENCY ANALYSIS?

Suppose our dictionary has \( N \) elements, in \( M \) arrays \((A_1, A_2, ..., A_m)\) and the length of array \( A_i \) is \( L_i \). What is the cost of:

- \text{contains(}e\text{)} — look for \( e \), starting with the smallest array

\[
O(\log(L_1)) + O(\log(L_2)) + ... = O(\sum_{i=1}^{M} \log(L_i)) = O(\log(\prod_{i=1}^{M} L_i))
\]

We need to binary search each array
IDEA 2: A BUNCH OF DYNAMIC SORTED ARRAYS

EFFICIENCY ANALYSIS?

Suppose our dictionary has $N$ elements, in $M$ arrays $(A_1,A_2,...,A_m)$ and the length of array $A_i$ is $L_i$. What is the cost of:

- $\text{remove}(e)$ – look for $e$, starting with the smallest array. If we find it, we replace that array with a new one that doesn’t contain $e$

$$O(\log(\prod_{i}^{M} L_i)) + O(L_k)$$

We need to search for $e$

Once we find it (in $A_k$) we need to remove it
TWO OBSERVATIONS

\[ O(\log(\prod_{i=1}^{m} L_i)) \]

is expensive when \( M \) is big

is expensive when \( L_{\text{smallest array}} \) is big
So,

- for a dictionary on $N$ elements, in $M$ arrays $(A_1, A_2, \ldots, A_m)$ and the length of array $A_i$ is $L_i$,

... it would be nice if we could keep both $M$ and $L_{\text{smallest array}}$ small...
QUESTION

In general, would you rather do all that stuff (maintaining a bunch of sorted arrays) with a lot of arrays or a few arrays?

A few arrays plz!!!
IDEA 3

- With these observations in mind, let’s try to do better
IDEA 3: AMORTIZED ARRAY-BASED DICTIONARY (AAD)

- Basically the same as our previous idea, except:
  - All of the arrays have different sizes
  - Each array has a size of the form $2^k$, for some $k$

ex:

- $[3]$  \[ 2^0 = 1 \]
- $[1, 4]$  \[ 2^1 = 2 \]
**IDEA 3: AAD**

- **Formal definition:**
  - An AAD on $N$ elements:
    - Consists of sorted arrays
    - Each array has a different length
    - Each array has a length that is a power of 2
    - The sum of the lengths of the arrays is $N$
    - $\text{contains}(e)$ iff $e$ is in one of the arrays

Let’s call this the “AAD property”
IDEA 3: AAD

- Is this an AAD?

\[
\begin{bmatrix}
3 \\
1, 6, 7
\end{bmatrix}
\]
Idea 3: AAD

- Is this an AAD?

\[ [3] \]
\[ [1, 6, 7, 9] \]
\[ [2, 4, 5, 8] \]
IDEA 3: AAD

- Is this an AAD?

By our definition, this is THE WAY to represent a dictionary with no elements!
IDEA 3: AAD

- Is this an AAD?

[3]
[7, 1, 9, 6]
IDEA 3: AAD

- Is this an AAD?

\[ [ 1, 6, 7, 9 ] \]
\[ [ 2, 4, 5, 8, 9, 14, 20, 25 ] \]
Idea 3: AAD

- **Theorem:**
  The *structure* of an AAD on N elements is unique

- **Proof:**
  The *structure* of such an AAD is related to the binary representation of N, which is unique.
IDEA 3: AAD

- **Theorem:**
  The *structure* of an AAD on N elements is unique

- We’ll use this theorem to our advantage. In designing add(e) and remove(e), we’ll try to think of the simplest and most efficient algorithms that get the job done.
IDEA 3: AAD

ADD

- add(e) – include [e], and then enforce the “AAD property”

ex: add(2) on

[ 2 ]
[ 3 ]
[ 1, 6, 7, 9 ]
[ 2, 4, 5, 8, 9, 14, 20, 25 ]
IDEA 3: AAD ADD (CONT.)

- Recall the theorem we just proved: “The *structure* of an AAD on $N$ elements is unique”

- We just added an element to an AAD on 13 elements, so now we have 14 elements

ex: $\text{add}(2)$ on

$$\begin{bmatrix} 3 & \_ \\ 1, 6, 7, 9 \end{bmatrix}$$

$$\begin{bmatrix} 2, 4, 5, 8, 9, 14, 20, 25 \end{bmatrix}$$
A really simple (and efficient) idea is to just *merge* the arrays of the same size (starting with the smallest arrays) until they all have different sizes.

\[
\begin{align*}
[2] \\
[3] \\
[1, 6, 7, 9] \\
[2, 4, 5, 8, 9, 14, 20, 25]
\end{align*}
\]
**Idea 3: AAD**

ADD (CONT.)

*merging* arrays of the same size until all the arrays have different sizes will enforce the “AAD property”
Idea 3: AAD
ADD (CONT.)

\[
\begin{align*}
[2] \\
[2 \ 3] \\
[1, 6, 7, 9] \\
[2, 4, 5, 8, 9, 14, 20, 25]
\end{align*}
\]

We can merge these guys
IDEA 3: AAD
MERGING TWO ARRAYS

Wait, how can we combine two sorted arrays into one sorted array?
IDEA 3: AAD
MERGING TWO ARRAYS

- We would like to design the function `merge` with the following specification:

  when A and B are sorted arrays,

  \[ \text{merge}(A,B) = C \]

  such that:

  - C contains, in sorted order, the contents of A and B
  - C.length = A.length + B.length
Idea 3: AAD
Merging Two Arrays

Any ideas?

\[
\begin{align*}
[2, 4, 6, 8] & \quad \text{MERGE} \quad [1, 3, 5, 7] \\
[_ , _ , _ , _ , _ , _ , _ , _ , _ , _ ]
\end{align*}
\]
IDEA 3: AAD

- **Theorem:**
  \[ \text{merge}(A,B) \text{ has a cost of } O(A.length + B.length) \]

- **Proof:**
  This follows directly from the intelligent way to implement merge – taking advantage of the fact that \( A \) and \( B \) are sorted!
IDEA 3: AAD
ADD (CONT.)

ex: add(8) on

[8]
MERGE
[2, 8]
MERGE
[2, 3, 4, 8]
MERGE
[1, 2, 3, 4, 5, 6, 7, 8]
Idea 3: AAD
MERGING TWO ARRAYS

This only works if we merge the smallest arrays first!
**Idea 3: AAD CONTAINS**

- `contains(e)` - look for `e` in each of the arrays, starting with the smallest array
- (exactly the same as with Idea 2)

**Example:**

`contains(14)` on

```
[2, 3]
[1, 6, 7, 9]
[2, 4, 5, 8, 9, 14, 20, 25]
```
Idea 3: AAD

REMOVE

- `remove(e)` – there are three cases:
  - **Case 1** – `e` is not in the dictionary
  - **Case 2** – `e` is in the dictionary, and it’s in the smallest array
  - **Case 3** – `e` is in the dictionary, and it’s not in the smallest array
IDEA 3: AAD
REMOVE — CASE 1

**Case 1** — e is not in the dictionary

We’re done!!!
IDEA 3: AAD
REMOVE — CASE 2

Case 2 — e is in the dictionary, and it’s in the smallest array

\[
\begin{align*}
\text{[ }_\text{, e } \text{, } _\text{, } _\text{, } _\text{, } _\text{] } & \quad \rightarrow \quad \text{[ }_\text{, } _\text{, } _\text{] } \\
\text{[ }_\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{] } & \quad \rightarrow \quad \text{[ }_\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{, } _\text{] }
\end{align*}
\]

***The rest of the dictionary didn’t change***
Idea 3: AAD
A cool idea for REMOVE(e)

Idea: remove e from the smallest array, and then split it up into a bunch of smaller arrays

\[
\begin{align*}
\left[ \_, e, \_, \_ \right] & \quad \rightarrow \quad \left[ \_, \_ \right] \\
\text{then just put those arrays in the dictionary}
\end{align*}
\]
Idea 3: AAD
Remove — Case 3

Case 3 — e is in the dictionary, and it’s not in the smallest array

Idea:

- find the array that contains e
- remove e from that array
- steal the biggest element from the smallest array and insert it
- then, simply split up the smallest array
Idea 3: AAD
Remove – Case 3 (cont.)

- Does this idea of using “split up” work?

\[
\begin{align*}
&\frac{1}{3} + \frac{1}{6} = \frac{1}{2} \\
&\frac{1}{9} + \frac{1}{18} = \frac{1}{9} \\
&\frac{1}{2} + \frac{1}{4} + \frac{1}{5} + \frac{1}{8} + \frac{1}{9} + \frac{1}{14} + \frac{1}{20} + \frac{1}{25} = \frac{128}{258} > 1/2
\end{align*}
\]

Yes!!!

\[2^k - 1 = \sum_{i=0}^{k-1} 2^i\]
IDEA 3: AAD

Cool, we’ve successfully designed the AAD data structure, which solves the dictionary problem.

Let’s prove some stuff about AADs!
IDEA 3: AAD

- **Theorem:**
  The *specific structure* of an AAD on $N$ elements is uniquely determined by the operations which created it.

- **Proof:**
  The empty AAD is unique.
  
  Both $\text{add}(e)$ and $\text{remove}(e)$ have predictable structural behavior, given the structure of the AAD.
IDEA 3: AAD
AN IMPORTANT OBSERVATION

We DEFINITELY want to permit duplicates in an AAD!!! Otherwise, add(e) becomes more complicated.
IDEA 3: AAD
FREQUENCY

- So, we introduce the notion of \textit{frequency}
- \texttt{frequency}(e) =
  - The number of elements in the AAD equal to \texttt{e}
  \textit{as well as}
  - The number of times we need to perform \texttt{remove}(e) before \texttt{contains}(e) is false
Idea 3: AAD

Frequency

- frequency(e) – search for e and count how many times we find it

ex: frequency(9) on

\[
[ 2, 3 ] \\
[ 1, 9, 9, 9 ] \\
[ 2, 4, 5, 8, 9, 14, 20, 25 ]
\]
IDEA 3: AAD

COMBINE

- We would like to be able to “combine” two dictionaries.
- \( \text{combine}(D) \) – combines the contents of the AAD \( D \)

\[
\begin{align*}
[3] & \quad \text{COMBINE} \quad [1, 3] \\
[1, 6] & \quad \rightarrow \quad [3] \\
[1, 1, 3, 6] & \quad \rightarrow \quad [1, 1, 3, 6]
\end{align*}
\]

- For AADs, we can actually implement \( \text{combine}(D) \) rather efficiently.
IDEA 3: AAD

Let's look at another example:

\[
\begin{align*}
[2, 3] & \quad [7, 8] \\
[1, 6, 7, 9] & \quad [1, 1, 4, 8]
\end{align*}
\]

RESULTS IN

\[
[1, 1, 1, 2, 2, 3, 4, 6, 6, 7, 7, 8, 8, 9]
\]
IDEA 3: AAD

Any ideas?

Let’s just combine the two AAD’s structurally, and then mergeDown
IDEA 3: AAD

COMBINE

[ _, _ ]

COMBINE

[ _, _, _, _ ]

RESULTS IN

[ _ ]

[ _ ]

MERGEDOWN

[ _, _, _, _, _, _, _, _, _ ]
**Idea 3: AAD**

- **Theorem:**
  
  contains(e) on an AAD on N elements is $O((\log N)^2)$

- **Proof:**
  
  In the worst case, the AAD *does not* contain e and it has log N arrays (so we need to search through each of them).

\[
O(\log(L_1)) + O(\log(L_2)) + ... = O(\sum_{i}^{\log N} \log(L_i)) = O(\sum_{k=0}^{\log N-1} \log(2^k))
\]

\[
= O(\sum_{k=0}^{\log N-1} k) = O\left(\frac{(\log N)(\log N - 1)}{2}\right) = O((\log N)^2)
\]
**Idea 3: AAD**

- **Theorem:**
  
  \( \text{add}(e) \) on an AAD on \( N \) elements has a cost of \( O(\log N) \) in the average case.

- **Proof** (the general idea):
  
  We can predict the expected structure of an AAD for arbitrary \( N \), and then use that structure to predict the merges will occur in the add algorithm (and we know the cost of each merge).
**Idea 3: AAD**

- **Theorem:**
  - remove(e) on an AAD on N elements has:
    - a cost of \(\text{contains}(e) + O(N)\) in the worst case
    - a cost of \(\text{contains}(e) + O(N')\) in the average case,
    where \(N'\) is a really small fraction of \(N\)

- **Proof** (the general idea):
  - (in both cases, we need to find the array that contains \(e\))
    - **Worst-Case Analysis** - the worst case for removal is that \(N\) is a power of 2 (so there is only 1 array). In this case, we need to "split up" remaining \(N-1\) elements in this array.
    - **Average-Case Analysis** - we can predict the expected structure of an AAD for arbitrary \(N\), predict the remove
**IDEA 3: AAD**

Suppose $e$ has a frequency of $F$

- **Theorem:**
  
  $\text{frequency}(e)$ on an AAD on $N$ elements has a cost of $\text{contains}(e) + O(F)$

- **Proof:**
  
  This follows directly from our algorithm for $\text{frequency}(e)$
THAT’S ALL

- OH 430-630 @ GHC 7110