15-121: Introduction to Data Structures
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 dictionary 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

\[
\begin{align*}
    \text{add}(e) & \quad \text{make } e \text{ a member} \\
    \text{remove}(e) & \quad \text{ensure } e \text{ is not a member} \\
    \text{contains}(e) & \quad \text{check for membership of } e
\end{align*}
\]
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

\[
\begin{align*}
\text{[1, 5, 8, 9]} \\
\text{[_, _, _, _]} \\
\text{[1, 5, 6, 8, 9]}
\end{align*}
\]
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

ex: remove(6) on

\[
\begin{array}{c}
[1, 5, 6, 8, 9] \\
[\_, \_, \_, \_, \_]
\end{array}
\]

\[
[1, 5, 8, 9]
\]
Idea 1: Dynamic Sorted Array Implementation of contains

- contains(e) – binary search the array

ex: contains(1) on [1, 5, 8, 9]
Suppose our dictionary has $N$ elements. What is the cost of:

- **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
  \[ O(N) \]

- **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
  \[ O(N) \]

- **contains($e$)** — binary search the array
  \[ O(\log N) \]
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!!!
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

[ 1 , 5 ] 8 
[ 2 , 4 ]
[ 3 , 6, 7 ]
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

- \([2, 4]\)
- \([1, 5, 8]\)
- \([3, 6, 7]\)
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

\[
\begin{array}{c}
[\emptyset] 4 \\
[1, 5, 8] \\
[3, 6, 7]
\end{array}
\]
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:

- \( \text{add}(e) \) – 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 \((A_1, A_2, \ldots, A_m)\) and the length of array \( A_i \) is \( L_i \). What is the cost of:

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

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

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 ($A_1, A_2, \ldots, A_m$) and the length of array $A_i$ is $L_i$. What is the cost of:

  - `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 \( L_i \) is small

is expensive when \( M \) 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:

- $[3] \quad 2^0 = 1$
- $[1, 4] \quad 2^1 = 2$
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 $N$
    - contains($e$) iff $e$ is in one of the arrays

Let’s call this the “AAD property”
Is this an AAD?

[3]
[1, 6, 7]
Idea 3: AAD

- Is this an AAD?

[3]

[1, 6, 7, 9]

[2, 4, 5, 8]

NO!
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]

NO!
Is this an AAD?

YES!

[ 3 ]

[ 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.

The number of arrays and the size of each array...
Theorem:
The *structure* of an AAD on $N$ elements is unique.

We’ll use this theorem to our advantage. In designing $\text{add}(e)$ and $\text{remove}(e)$, 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 ]
```

Now what do we do???
Recall the theorem we just proved:

“The *structure* of an AAD on $\mathbb{N}$ elements is unique.”

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

We know the structure needs to look like this:

ex: add(2) on

\[
\begin{bmatrix}
3 & \_ \\
1 & 6 & 7 & 9 \\
2 & 4 & 5 & 8 & 9 & 14 & 20 & \_ & \_ & \_ & 25
\end{bmatrix}
\]
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.

\[
\begin{align*}
\text{[ } & 2 \text{ ]} \\
\text{[ } & 3 \text{ ]} \\
\text{[ } & 1, 6, 7, 9 \text{ ]} \\
\text{[ } & 2, 4, 5, 8, 9, 14, 20, 25 \text{ ]}
\end{align*}
\]
Idea 3: AAD add (cont.)

*merging* arrays of the same size until all the arrays have different sizes will enforce the “AAD property”

“mergeDown”
We can merge these guys.
Idea 3: AAD
merging two arrays

Wait, how can we combine two sorted arrays into one sorted array?
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 \( B \)
- \( C\.\text{length} = A\.\text{length} + B\.\text{length} \)
Idea 3: AAD
merging two arrays

Any ideas?

\[ 2, 4, 6, 8 \quad \text{MERGE} \quad [ 1, 3, 5, 7 ] \]

[_, _, _, _, _, _, _, _, _]
**Theorem:**

\( \text{merge}(A, B) \) has a cost of \( O(A.\text{length} + B.\text{length}) \)

**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

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

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

```
[_, e, _, _, _]  [___]
[_, _, _, _, _, _, _, _]  [_, _, _, _, _, _, _, _, _]
```

***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

\[
\begin{array}{c}
[\_ , e , \_ , \_ ] \\
\end{array}
\rightarrow
\begin{array}{c}
[\_ ] \\
[\_ , \_ ]
\end{array}
\]

then just put those arrays in the dictionary
**Case 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?

\[ 3 \]

6, 7, 9

Yes!!!

5, 8, 9, 14, 20, 25

\[ 2^k - 1 = \sum_{i=0}^{k-1} 2^i \]
Cool, we’ve successfully designed the AAD data structure, which solves the dictionary problem.

Let’s prove some stuff about AADs!
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 add($e$) and 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 *frequency*
- \( \text{frequency}(e) = \)
  - The number of elements in the AAD equal to \( e \)
  - *as well as*
  - The number of times we need to perform \( \text{remove}(e) \) before \( \text{contains}(e) \) 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]
We would like to be able to “combine” two dictionaries.

combine(D) – combines the contents of the AAD D

For AADs, we can actually implement combine(D) rather efficiently.
Let’s look at another example:

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

**COMBINE**

\[
[1, 1, 1, 2, 2, 3, 4, 6, 6, 7, 7, 8, 8, 9]
\]
Any ideas?

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

COMBINE

RESULTS IN

MERGEDOWN
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).

\[
N = 2^k - 1
\]

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

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

- **Theorem:**
  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 contains(e) + O(N) in the worst case
- a cost of 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” the remaining N-1 elements in this array

Average-Case Analysis - we can predict the expected structure of an AAD for arbitrary N, and then use that structure to predict the “split ups” that will occur in the remove algorithm
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

- Have a good weekend