Examples on Dynamic Array and Hashing

Dan (Doris) He
ITEE
University of Queensland
Dynamic Array Problem

Let $S$ be a multi-set of integers that grows with time. At the beginning, $S$ is empty. Over time, the integers of $S$ are added by the following operation:

- **insert($e$):** which adds an integer $e$ into $S$.

At any moment, let $n$ be the number of elements in $S$. We want to store all the elements of $S$ in an array $A$ satisfying:

1. $A$ has length $O(n)$

2. If an integer $x$ was the $i$-th ($i \geq 1$) inserted, then $A[i] = x$ (i.e., $x$ is at the $i$-th position of the array).
Before the first insertion, $A$ is an empty array.

- **insert(41)**: Create an array $A$ with length $2$

\[
\begin{array}{c}
41 \\
\end{array}
\]

- **insert(12)**: $A$ is full. Create another array $A'$ with length $4$, copy the two elements in $A$ to $A'$, destroy $A$, replace it with $A'$.

\[
\begin{array}{cc}
41 & 12 \\
\end{array}
\]

\[
\begin{array}{c}
41 & 12 \\
\end{array}
\]
**Example**

**Input:**

41, 12, 37, 9, 12, 63, 15, 33, 7, 41, 52, 33, 11, 9, 27, 8, 31, ...
Stack-with-Array Problem

Let $S$ be a multi-set of integers that grows with time. At the beginning, $S$ is empty. We must support the following stack operations:

- **push($e$)**: which adds an integer $e$ into $S$.
- **pop**: which removes from $S$ the most recently inserted integer.

At any moment, let $m$ be the number of elements in $S$. We want to store all the elements of $S$ in an array $A$ satisfying:

1. $A$ has length $O(m)$

We will denote by $n$ the number of operations processed so far.
Example

Operations:

push(32), push(15), push(7)
pop, pop
push(18), push(32), push(9), push(12), push(64)
pop, pop, pop, pop,
push(5), push(11)
Example

- At the beginning, create an array $A$ with length 4.
- After push(32), push(15), push(7), $m = 3$.

\[
\begin{array}{ccc}
32 & 15 & 7
\end{array}
\]


\[
\begin{array}{cc}
32 & 15
\end{array}
\]


\[
\begin{array}{c}
32
\end{array}
\]
Example

- push(18), push(32). At this moment, $m = 3$.

```
32 18 32
```

- push(9). At this moment, $m = 4$, and $A$ is full. So we expand $A$ (same as dynamic array insertion)

```
32 18 32 9
```

- push(12), push(64). At this moment, $m = 6$.

```
32 18 32 9 12 64
```
Example

- pop: return $A[3] = 32; m = 2$. At this moment, $A$ becomes sparse. Shrink $A$ as follows: create an array $A'$ with length 4, copy the elements in $A$ to $A'$, destroy $A$ and replace $A$ with $A'$.

After push(5) and push(11), $m = 4$. 

- After push(5) and push(11), $m = 4$. 
Hashing

A Universal Function

- Pick a prime number $p$ such that
  - $p \geq m$.
  - $p \geq$ any possible integer $k$
- Choose a number $\alpha$ uniformly at random from 1, ..., $p - 1$.
- Choose a number $\beta$ uniformly at random from 0, ..., $p - 1$.
- Construct a hash function:

$$h(k) = 1 + ((\alpha k + \beta) \mod p) \mod m$$
Let $S = \{33, 42, 70, 38, 6, 22, 17, 51, 8, 14, 63, 27\}$. Suppose that we choose $m = 10$, $p = 71$, $\alpha = 3$, $\beta = 7$ and $h(k) = 1 + (((3k + 7) \mod 71) \mod 10)$.

How to answer a query with search value 51? How about 52?
Dynamic Array – A New Version

In the following, we will discuss a variant of dynamic arrays with smaller space consumption.

Recall that, while performing insertions to a dynamic array $A$, once $A$ is full, we expand $A$ by doubling the current length. We have proved that each insertion costs $O(1)$ amortized time and that the space consumption is $O(n)$ at any moment, where $n$ is the number of elements in $A$.

In fact, it is not necessary to restrict the expansion to doubling. In the following, we will show that each insertion still costs $O(1)$ amortized time as long as we expand the length of $A$ to $(1 + \epsilon)n$ for any constant $\epsilon > 0$. 
Dynamic Array – A New Version

Suppose that an array expansion occurs when $A$ is full with $n$ elements and that this expansion takes $c \cdot n$ time.

- The previous expansion happened when $A$ was full with $n_{1+\epsilon}$ elements.
- There were $n - \frac{n}{1+\epsilon} = \frac{\epsilon n}{1+\epsilon}$ empty positions right after the previous expansion.
- $\frac{\epsilon n}{1+\epsilon}$ insertions have taken place since the previous expansion.
- Charge the $c \cdot n$ cost over those $\frac{\epsilon n}{1+\epsilon}$ insertions.
- Each of those insertions bears additional $\frac{cn}{1+\epsilon} = \frac{c(1+\epsilon)}{\epsilon} = O(1)$ cost and each of them is charged only once.
Hashing

As discussed in the lecture, to achieve $O(1)$ expected query time in dictionary search, we should choose a $m = \Theta(n)$. Suppose that $m = c \cdot n$, where $c$ is a constant. Next, we will explore the effects of $c$.

- Space consumption at the order of $O(n + m) = O(n + c \cdot n)$.
- Query time $O(1 + n/m) = O(1 + \frac{1}{c})$ in expectation, where:
  - “1” captures the cost for access the hash table to obtain the head of the corresponding linked list.
  - “$\frac{1}{c}$” indicates the expected number of elements in that linked list.

As we can see from the above, for different choices of $c$, there is a tradeoff between the space consumption and the query time.
Hashing

- $m = \frac{1}{2} n$:
  - Space consumption $O(n + m) = O(n + \frac{1}{2} \cdot n)$.
  - Query time $O(1 + n/m) = O(1 + 2)$ in expectation.
Hashing

- $m = 10n$:
  - Space consumption $O(n + m) = O(n + 10 \cdot n)$.
  - Query time $O(1 + n/m) = O(1 + \frac{1}{10})$ in expectation.

Recall that $\frac{1}{10}$ is the expected number of the elements that we need to visit in answering a query. It means that in expectation, in 9 out of 10 queries, we need to access no elements at all.