Outline

• Dynamic array vs. linked list

• Dynamic array: space and update cost tradeoff

• An application of the stack
Dynamic Array vs Linked List

A linked list ensures $O(1)$ insertion cost. A dynamic array guarantees $O(1)$ insertion cost after amortization.

However, a dynamic array provides constant-time access to any position, which a linked list cannot achieve.
Dynamic Array vs Linked List

Question:
Design a data structure of $O(n)$ space to store a set $S$ of $n$ integers to satisfy the following requirements:

• An integer can be inserted in $O(1)$ time.
• We can enumerate all integers in $O(n)$ time.

Answer: Linked list.
Dynamic Array vs Linked List

Question:

Design a data structure of $O(n)$ space to store a set $S$ of $n$ integers to satisfy the following requirements:

• An integer can be inserted in $O(1)$ amortized time.
• We can enumerate all integers in $O(n)$ time.
• For each $i \in [1, n]$, we can access the $i$-th inserted integer in $O(1)$ time.

Answer: Dynamic array
Outline

• Dynamic array vs. linked list

• Dynamic array: space and update cost tradeoff

• An application of the stack
Space-Update Tradeoff of the Dynamic Array

In the lecture, we expand the array from size $n$ to $2n$ when it is full.

What if we expand the array size to $3n$ instead?
Space-Update Tradeoff of the Dynamic Array

• Initially, size 3 (define $s_1 = 3$)
• 1st expansion: size from $s_1$ to $s_2 = 3s_1 = 9$.
• 2nd expansion: from $s_2$ to $s_3 = 3s_2 = 27$.
  ...
• $i$-th expansion: from $s_i$ to $s_{i+1} = 3s_i$.

We have $s_i = 3^i$. 
Space-Update Tradeoff of the Dynamic Array

• The total cost of \( n \) insertions is bounded by:

\[
\left( \sum_{i=1}^{n} O(1) \right) + \sum_{i=1}^{h} O(3^{i+1}) = O(n + 3^{h+1})
\]

where \( h \) is the number of expansions.

It must hold that \( n \geq s_h \geq 3^h \) (the \( h \)-th expansion happened because the array of size \( s_h \) was full).

Hence, the total cost is \( O(n) \).
Space-Update Tradeoff of the Dynamic Array

• Consider what happens in general. When the array is full, expand its size from $n$ to $\alpha n$, for some constant $\alpha > 1$. 
Space-Update Tradeoff of the Dynamic Array

- Initially, size 2 (define $s_1 = 2$)
- 1\textsuperscript{st} expansion: size from $s_1$ to $s_2 = \lceil \alpha s_1 \rceil$.
- 2\textsuperscript{nd} expansion: from $s_2$ to $s_3 = \lceil \alpha s_2 \rceil$.
  
  ...

- $i$-th expansion: from $s_i$ to $s_{i+1} = \lceil \alpha s_i \rceil$.

We can prove: $s_i = O\left(\frac{\alpha^i}{\alpha - 1}\right)$ and $s_i \geq \alpha^i$. 
Space-Update Tradeoff of the Dynamic Array

The total cost of $n$ insertions is bounded by:

$$\left( \sum_{i=1}^{n} O(1) \right) + \sum_{i=1}^{h} O\left( \frac{\alpha^{i+1}}{\alpha - 1} \right) = O\left( n + \frac{\alpha^{h+2}}{(\alpha - 1)^2} \right)$$

where $h$ is the number of expansions.

It must hold that $n \geq s_h \geq \alpha^h$ (the $h$-th expansion happened because the array of size $s_h$ was full).

Hence, the total cost is $O\left( n + \frac{\alpha^2}{(\alpha - 1)^2} n \right)$, namely, amortized cost $= O\left( 1 + \frac{\alpha^2}{(\alpha - 1)^2} \right)$.
Space-Update Tradeoff of the Dynamic Array

Amortized cost = $O \left( 1 + \frac{\alpha^2}{(\alpha-1)^2} \right)$.

When $\alpha$ increases, the space consumption goes up, but the insertion cost goes down.
Outline

• Dynamic array vs. linked list

• Dynamic array: space and update cost tradeoff

• An application of the stack
Input: A sentence stored in a sequence of $n$ cells. Each cell contains a word or one of the following pairing characters: 

```
" ", ( ), { }, <, >
```

Please design an algorithm to determine whether the pairing characters have been matched correctly (in the way we are used to in English).

The following input is a correct sentence:

```
I say " I like ( red ) apple "
```

while the one below is not:

```
I say " I like ( red apple " )
```

Your algorithm should finish in $O(n)$ time.
Using a Stack

The key idea is to use a stack to check whether all the ", (, {, < are closed properly. We will discuss the ideas on the following two examples:

```
{ < < " " } > ( ) > }
```

```
{ < < " { } > ( ) > }
```
The Algorithm

Sequentially scan the input sentences.
At reading a "", (, <, or {, push it into the stack.
At reading a ",", ), >, or }, check whether the top of the stack
matches the character just read. If so, pop the stack and
continue; otherwise, report “incorrect”.

After reading all the cells, check whether the stack is empty. If
so, report “correct”; otherwise, report “incorrect”.

The running time is clearly O(n).