A **data structure** has two functionalities:

- store a set of elements;
- supports certain operations on those elements.

The only data structure in our discussion so far is the **array**.

In this lecture, we will first discuss a new data structure, the **linked list**, and then utilize it to design two other structures: the **stack** and the **queue**.
A **linked list** is a sequence of **nodes** where:

- each node is an array;
- the node’s **address** is defined as its array’s starting memory address;
- the node stores in its array
  - a **back-pointer** to its preceding node (if it exists);
  - a **next-pointer** to its succeeding node (if it exists).

Recall that a “pointer” is a memory address.

In a linked list, the first node is called the **head** and the last node is called the **tail**.
The figure below illustrates a linked list of three nodes $u_1, u_2,$ and $u_3$, whose addresses are $a, b,$ and $c$, respectively.

The back-pointer of node $u_1$ (the head) is nil, denoted by $\bot$. The next-pointer of $u_3$ (the tail) is also nil.
**Example:**

A linked list storing a set of integers \( \{14, 65, 78, 33, 82\} \):

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th></th>
<th>a</th>
<th></th>
<th>d</th>
<th></th>
<th>e</th>
<th></th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>a</td>
<td>c</td>
<td>65</td>
<td>⊥</td>
<td>b</td>
<td>82</td>
<td>c</td>
<td>e</td>
<td>14</td>
</tr>
</tbody>
</table>

Conceptually, we can think of the sequence \( (65, 78, 33, 82, 14) \) in the linked list as:

\[ 65 \quad \longrightarrow \quad 78 \quad \longrightarrow \quad 33 \quad \longrightarrow \quad 82 \quad \longrightarrow \quad 14 \]
Two (Simple) Facts

Suppose that we use a linked list to store a set $S$ of $n$ integers (one node per integer).

**Fact 1:** The linked list uses $O(n)$ space, namely, $O(n)$ memory cells.

**Fact 2:** Starting from the head node, we can enumerate all the integers in $S$ in $O(n)$ time.
A linked list storing a set $S$ supports updates:

- **insertion**: add a new element to $S$;
- **deletion**: remove an existing element from $S$. 

Insertion in a Linked List

To insert a new element $e$, append $e$ to the linked list:

1. Identify the tail node $u$.
2. Create a new node $u_{\text{new}}$ to store $e$.
3. Set the next-pointer of $u$ to the address of $u_{\text{new}}$.
4. Set the back-pointer of $u_{\text{new}}$ to the address of $u$.

$O(1)$ time.
Example

After inserting 57:

Yufei Tao

Linked Lists, Stacks, and Queues
Deletion from a Linked List

Given a pointer to a node \( u \) in the linked list, we can delete the node as follows:

1. Identify the preceding node \( u_{\text{prec}} \) of \( u \).
2. Identify the succeeding node \( u_{\text{succ}} \) of \( u \).
3. Set the next-pointer of \( u_{\text{prec}} \) to the address of \( u_{\text{succ}} \).
4. Set the back-pointer of \( u_{\text{succ}} \) to the address of \( u_{\text{prec}} \).
5. Free up the memory of \( u \).

\( O(1) \) time
Example

<table>
<thead>
<tr>
<th>f</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>e</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>e</td>
<td>⊥</td>
<td>78</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>65</td>
<td>⊥</td>
<td>b</td>
<td>82</td>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>14</td>
<td>d</td>
<td>f</td>
<td>33</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

After deleting 78:

<table>
<thead>
<tr>
<th>f</th>
<th></th>
<th>a</th>
<th>d</th>
<th>e</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>e</td>
<td>⊥</td>
<td>65</td>
<td>c</td>
<td>82</td>
</tr>
<tr>
<td>14</td>
<td>d</td>
<td>f</td>
<td>33</td>
<td>a</td>
<td>d</td>
</tr>
</tbody>
</table>

65 ←→ 78 ←→ 33 ←→ 82 ←→ 14 ←→ 57

65 ←→ 33 ←→ 82 ←→ 14 ←→ 57
Next, we will deploy the linked list to implement two data structures: stack and queue.
A stack manages a set $S$ of elements and supports two operations:

- **push**($e$): insert a new element $e$ into $S$.
- **pop**: remove the *most recently inserted* element from $S$ and returns it.

First-In-Last-Out (FILO).
Example

Consider the following sequence of operations on an empty stack:

- **Push(35):** $S = \{ 35 \}$.
- **Push(23):** $S = \{ 35, 23 \}$.
- **Push(79):** $S = \{ 35, 23, 79 \}$.
- **Pop:** return 79 after removing it from $S$. Now $S = \{ 35, 23 \}$.
- **Pop:** return 23 after removing it from $S$. Now $S = \{ 35 \}$.
- **Push(47):** $S = \{ 35, 47 \}$.
- **Pop:** return 47 after removing it from $S$. Now $S = \{ 35 \}$.
Linked-List implementation of a Stack

Store the elements of $S$ in a linked list $L$.

**Push** ($e$): insert $e$ at the end of $L$.
**Pop**: delete the tail node of $L$ and return the element therein.

At all times, keep track of a pointer to the tail node.

**Guarantees:**

- $O(n)$ space where $n = |S|$ (assuming that each element in $S$ occupies $O(1)$ memory).
- Push in $O(1)$ time.
- Pop in $O(1)$ time.
A queue stores a set $S$ of elements and supports two operations:

- $\text{en-queue}(e)$: inserts an element $e$ into $S$.
- $\text{de-queue}$: removes the least recently inserted element from $S$ and returns it.

First-In-First-Out (FIFO).
Example

Consider the following sequence of operations on an initially empty queue:

- **En-queue(35):** $S = \{35\}$.
- **En-queue(23):** $S = \{35, 23\}$.
- **En-queue(79):** $S = \{35, 23, 79\}$.
- **De-queue:** return 35 after removing it from $S$. Now $S = \{23, 79\}$.
- **De-queue:** return 23 after removing it from $S$. Now $S = \{79\}$.
- **En-queue(47):** $S = \{79, 47\}$.
- **De-queue:** return 79 after removing it from $S$. Now $S = \{47\}$.
Linked-List Implementation of a Queue

Store the elements of $S$ in a linked list $L$.

**En-queue**($e$): insert $e$ at the end of $L$.
**De-queue**: delete the head node of $L$ and return the element therein.

At all times, keep track of the addresses of the head and the tail.

**Guarantees:**
- $O(n)$ space, where $n = |S|$ (assuming each element in $S$ occupies $O(1)$ memory).
- En-queue in $O(1)$ time.
- De-queue in $O(1)$ time.