Outline

1 Binary Search Tree (BST)

2 Balanced BST, AVL-Tree
Dynamic Data Structure

- Sort a set of given numbers, or search a number:
Sort a set of given numbers, or search a number:

- We can use *Insertion Sort, Merge Sort, Heap Sort, ...* in $\Theta(n \log n)$ time.
Sort a set of given numbers, or search a number:

- We can use *Insertion Sort, Merge Sort, Heap Sort*, ... in $\Theta(n \log n)$ time.
- Then, we can use binary search in $\Theta(\log n)$ time after sorting.

But, before running the algorithms, we require that all the numbers have been already recorded in the memory.

However, in many scenarios:

- Airport: we always have new arrival/departure flights.
- Amazon: we always have new online orders and need to schedule the delivery dates.

So, we need dynamic data structure to augment and store the data in an efficient way.
Dynamic Data Structure

- Sort a set of given numbers, or search a number:
  - We can use *Insertion Sort, Merge Sort, Heap Sort, ...* in $\Theta(n \log n)$ time.
  - Then, we can use binary search in $\Theta(\log n)$ time after sorting.
  - But, before running the algorithms, we require that all the numbers have been already recorded in the memory.
Dynamic Data Structure

Sort a set of given numbers, or search a number:
- We can use **Insertion Sort, Merge Sort, Heap Sort, ...** in $\Theta(n \log n)$ time.
- Then, we can use binary search in $\Theta(\log n)$ time after sorting.
- But, before running the algorithms, we require that all the numbers have been already recorded in the memory.

However, in many scenarios:
Dynamic Data Structure

- Sort a set of given numbers, or search a number:
  - We can use *Insertion Sort, Merge Sort, Heap Sort*, ... in $\Theta(n \log n)$ time.
  - Then, we can use binary search in $\Theta(\log n)$ time after sorting.
  - But, before running the algorithms, we require that all the numbers have been already recorded in the memory.

- However, in many scenarios:
  - Airport: we always have new arrival/departure flights.
Sort a set of given numbers, or search a number:
- We can use Insertion Sort, Merge Sort, Heap Sort, ... in $\Theta(n \log n)$ time.
- Then, we can use binary search in $\Theta(\log n)$ time after sorting.
- But, before running the algorithms, we require that all the numbers have been already recorded in the memory.

However, in many scenarios:
- Airport: we always have new arrival/departure flights.
- Amazon: we always have new online orders and need to schedule the delivery dates.
Dynamic Data Structure

- Sort a set of given numbers, or search a number:
  - We can use *Insertion Sort, Merge Sort, Heap Sort, ...* in $\Theta(n \log n)$ time.
  - Then, we can use binary search in $\Theta(\log n)$ time after sorting.
  - But, before running the algorithms, we require that all the numbers have been already recorded in the memory.

- However, in many scenarios:
  - Airport: we always have new arrival/departure flights.
  - Amazon: we always have new online orders and need to schedule the delivery dates.
  - So, we need dynamic data structure to augment and store the data in an efficient way.
Imagine there is an infinite stream of numbers, and arrives in your memory one by one:
Imagine there is an infinite stream of numbers, and arrives in your memory one by one:

- We want to maintain the arrived numbers structurally. Each time a new number arrives, we need to insert it in an appropriate position.
Imagine there is an infinite stream of numbers, and arrives in your memory one by one:

- We want to maintain the arrived numbers structurally. Each time a new number arrives, we need to insert it in an appropriate position.
- Such that we can answer a search fast, any time.
Imagine there is an infinite stream of numbers, and arrives in your memory one by one:

- We want to maintain the arrived numbers structurally. Each time a new number arrives, we need to insert it in an appropriate position.
- Such that we can answer a search fast, any time.

**Array**: costly for shifting the previous numbers when inserting a new number.
Binary Search Tree (BST)

- Imagine there is an **infinite stream** of numbers, and arrives in your memory one by one:
  - We want to maintain the arrived numbers **structurally**. Each time a new number arrives, we need to insert it in an appropriate position.
  - Such that we can answer a search **fast, any time**.

- **Array**: costly for shifting the previous numbers when inserting a new number.

- **List**: costly for searching.
Imagine there is an infinite stream of numbers, and arrives in your memory one by one:

- We want to maintain the arrived numbers structurally. Each time a new number arrives, we need to insert it in an appropriate position.
- Such that we can answer a search fast, any time.

**Array:** costly for shifting the previous numbers when inserting a new number.

**List:** costly for searching.

**Heap:** costly for searching, too.
Imagine there is an infinite stream of numbers, and arrives in your memory one by one:
- We want to maintain the arrived numbers structurally. Each time a new number arrives, we need to insert it in an appropriate position.
- Such that we can answer a search fast, any time.

**Array:** costly for shifting the previous numbers when inserting a new number.

**List:** costly for searching.

**Heap:** costly for searching, too.

**Binary Search Tree (BST):** solve this problem to certain extent, but not perfectly.
**BST**: A binary tree.
**BST**: A binary tree.

- Each node $x$;
**BST:** A binary tree.
- Each node x;
- Key value key(x);
**BST**: A binary tree.
- Each node x;
- Key value key(x);
- Pointers Left(x), right(x), and parent(x);
Binary Search Tree (BST)

- **BST**: A binary tree.
  - Each node $x$;
  - Key value $key(x)$;
  - Pointers $Left(x)$, $right(x)$, and $parent(x)$;
  - For each node $x$, the key values in its left sub-tree $\leq key(x) \leq$ the key values in its right sub-tree.
Binary Search Tree (BST)

- An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
Binary Search Tree (BST)

- An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
Binary Search Tree (BST)

- An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
Binary Search Tree (BST)

- An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
Binary Search Tree (BST)

- An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ...
Binary Search Tree (BST)

- An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ......
An example: suppose the stream of numbers are 6, 3, 7, 8, 11, 2, 4, 9, ....

Search time is $O(\text{height})$, which could be as large as $O(n)$.
Outline

1. Binary Search Tree (BST)

2. Balanced BST, AVL-Tree
A BST is balanced if height=$\Theta(\log n)$.

AVL-Tree: for each node, the difference between the heights of its left and right sub-trees is at most 1, invented by Georgy Adelson-Velsky and Evgenii Landis.

Let $h$ is the height, $#(h)$ is the minimum number of nodes in an AVL-Tree with height $h$;

$$#(h) \geq #(h-1) + #(h-2) + 1$$

$$\Rightarrow #(h) > 2h/2$$

$$\Rightarrow h = O(\log #(h))$$.

So AVL-Tree is balanced BST.
Balanced BST

- A BST is **balanced** if height=$\Theta(\log n)$.
- **AVL-Tree**: for each node, the difference between the heights of its left and right sub-trees is at most 1, invented by Georgy Adelson-Velsky and Evgenii Landis.
A BST is **balanced** if height=$\Theta(\log n)$.

**AVL-Tree**: for each node, the difference between the heights of its left and right sub-trees is at most 1, invented by Georgy Adelson-Velsky and Evgenii Landis.

- Let $h$ is the height, $\#(h)$ is the minimum number of nodes in a AVL-Tree with height $h$;
A BST is **balanced** if height=$\Theta(\log n)$.

**AVL-Tree:** for each node, the difference between the heights of its left and right sub-trees is at most 1, invented by Georgy Adelson-Velsky and Evgenii Landis.

- Let $h$ is the height, $(h)$ is the minimum number of nodes in a AVL-Tree with height $h$;
- $(h) \geq (h - 1) + (h - 2) + 1$
Balanced BST

- A BST is **balanced** if height=$\Theta(\log n)$.
- **AVL-Tree**: for each node, the difference between the heights of its left and right sub-trees is at most 1, invented by Georgy Adelson-Velsky and Evgenii Landis.
  - Let $h$ is the height, $#(h)$ is the minimum number of nodes in a AVL-Tree with height $h$;
  - $#(h) \geq #(h-1) + #(h-2) + 1$
  - $\implies $#(h) > 2#(h-2) \implies $#(h) > 2^{h/2} \implies h = O(\log #(h))$. 


Balanced BST

- A BST is **balanced** if height $= \Theta(\log n)$.

**AVL-Tree:** for each node, the difference between the heights of its left and right sub-trees is at most 1, invented by Georgy Adelson-Velsky and Evgenii Landis.

- Let $h$ is the height, $(h)$ is the minimum number of nodes in a AVL-Tree with height $h$;
- $(h) \geq (h - 1) + (h - 2) + 1$
- $\implies (h) > 2(h - 2) \implies (h) > 2^{h/2} \implies h = O(\log (h))$.
- So AVL-Tree is balanced BST.
**Basic operations:** Left Rotation (LR), Right Rotation (RR), Right Rotation+Left Rotation (RRLR), Left Rotation+Right Rotation (LRRR).
**Basic operations:** Left Rotation (LR), Right Rotation (RR), Right Rotation+Left Rotation (RRLR), Left Rotation+Right Rotation (LRRR).
**Basic operations:** Left Rotation (LR), Right Rotation (RR), Right Rotation+Left Rotation (RRLR), Left Rotation+Right Rotation (LRRR).
**Basic operations:** Left Rotation (LR), Right Rotation (RR), Right Rotation+Left Rotation (RRLR), Left Rotation+Right Rotation (LRRR).
Basic operations: Left Rotation (LR), Right Rotation (RR), Right Rotation+Left Rotation (RRLR), Left Rotation+Right Rotation (LRRR).
Maintain AVL-Tree each time inserting a new number:
Maintain AVL-Tree each time inserting a new number:

1. Run the simple BST to insert a new node.
Maintain AVL-Tree each time inserting a new number:

1. Run the simple BST to insert a new node.
2. Start from the lowest unbalanced node up to the highest unbalanced node, do one of LR, RR, RRLR, and LRRR.
AVL-Tree

- Maintain AVL-Tree each time inserting a new number:
  1. Run the simple BST to insert a new node.
  2. Start from the lowest unbalanced node up to the highest unbalanced node, do one of LR, RR, RRLR, and LRRR.

- Run time: $O(\log n)$. 