Splay Trees

Recall: **Asymptotic analysis** examines how an algorithm will perform in worst case.

Amortized analysis examines how an algorithm will perform in practice or on average.

The **90–10 rule** states that **90%** of the accesses are to **10%** of the data items.

However, balanced search trees do not take advantage of this rule.

- The **90–10** rule has been used for many years in **disk I/O systems**.
- A **cache** stores in main memory the contents of some of the disk blocks. The hope is that when a disk access is requested, the block can be found in the main memory cache and thus save the cost of an expensive disk access.
- **Browsers** make use of the same idea: A cache stores locally the previously visited Web pages.

Splay Trees:

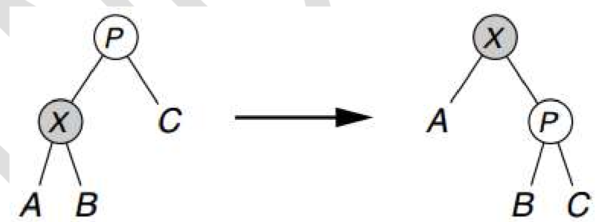
- Uses the standard binary search tree property.
- After any operation on a node, make that node the new root of the tree.

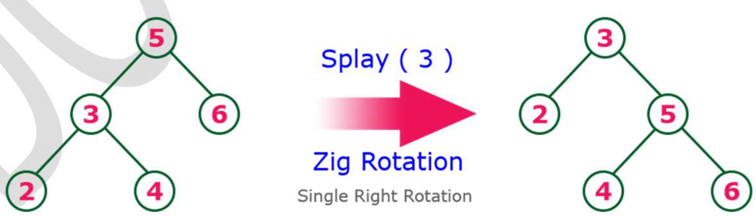
The basic bottom-up splay tree

Splaying cases:

- **The zig case** (normal single right rotation)

  If **X** is a non-root node on the access path on which we are rotating and the parent of **X** is the root of the tree, we merely rotate **X** and the root, as shown:





In zig rotation, every node moves one position to the right from its current position.

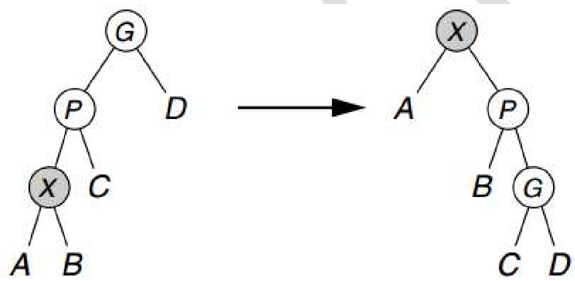
- **The zag case** (normal single left rotation)

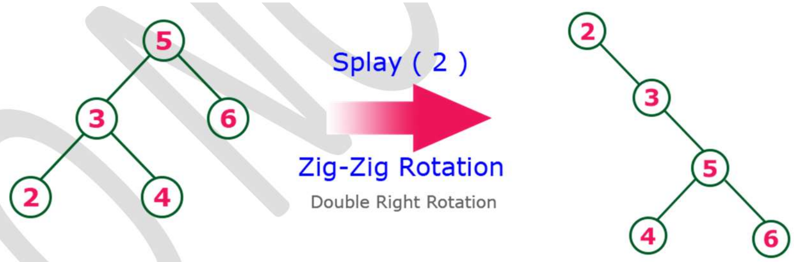
If **X** is a non-root node on the access path on which we are rotating and the parent of **X** is the root of the tree, we merely rotate **X** and the root, as shown:

In zag rotation, every node moves one position to the left from its current position.

Otherwise, **X** has both a parent **P** and a grandparent **G**, and we must consider two cases and symmetries.

- **zig-zig case:**
  - The left outside case for **AVL** trees.
  - Here, **X** and **P** are both left children.
  - In this case, we transform the left-hand tree to the right-hand tree.
  - Note that this method differs from the AVL **rotate-to-root strategy**.
    - The **zig-zig** splay rotates between <u>**P** and **G** and then **X** and **P**</u>, whereas the AVL **rotate-to-root strategy** rotates between <u>**X** and **P** and then between **X** and **G**</u>.



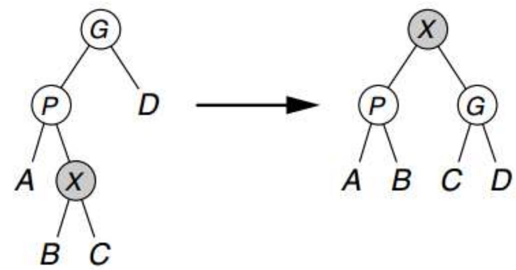


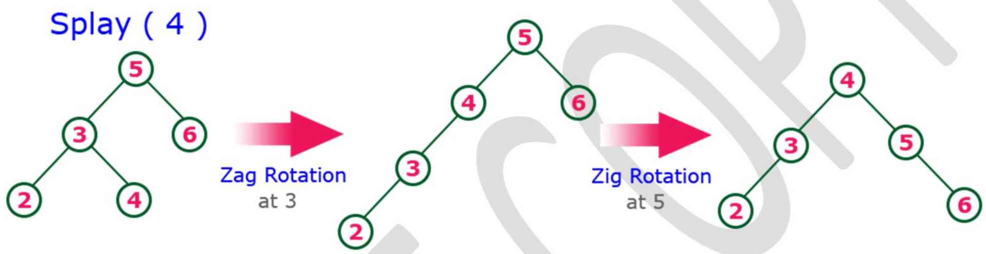
Every node moves two positions to the right from its current position.

- **zag-zag case:**
  - The right outside case for **AVL** trees.
  - Here, **X** and **P** are both right children.
  - In this case, we transform the right-hand tree to the left-hand tree.

In zag-zag rotation, every node moves two positions to the left from its current position.

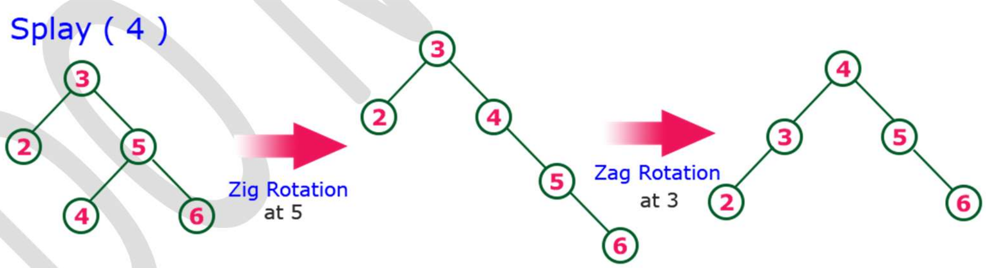
- **zag-zig case**:
  - This corresponds to the inside right-left case for **AVL** trees.
  - Here **X** is a right child and **P** is a left child.
  - We perform a **double rotation** exactly like an **AVL** double rotation, as shown:





In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.

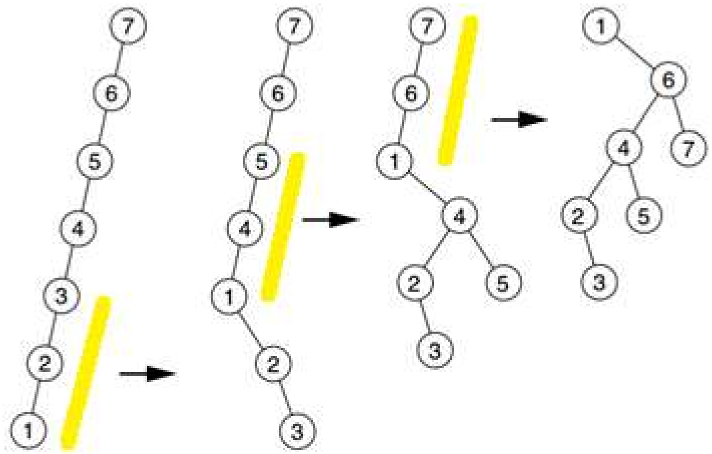
- **zig-zag case**:
  - This corresponds to the inside left-right case for **AVL** trees.
  - Here **X** is a left child and **P** is a right child).
  - We perform a **double rotation** exactly like an **AVL** double rotation, as shown:



In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position.

Splaying has the effect of roughly **halving** the depth of most nodes on the access path and increasing by at most **two levels** the depth of a few other nodes.

Example: Result of splaying at node **1** (three zig-zigs)



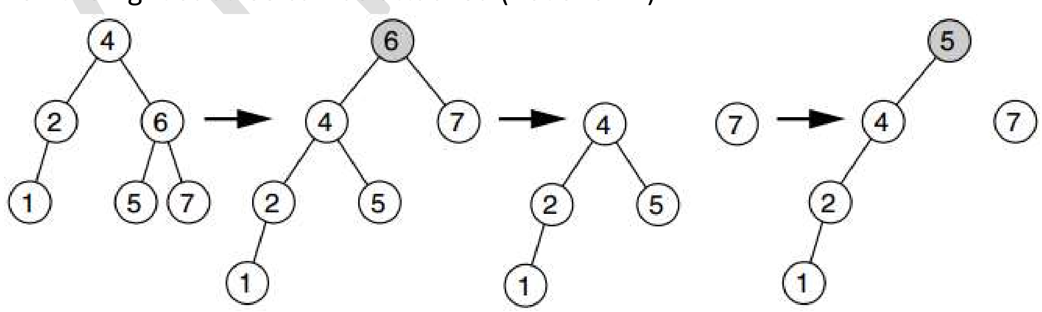
Exercise: perform rotate-to-root strategy

Basic splay tree operations

A splay operation is performed after each access:

- After an item has been inserted as a leaf, it is **splayed** to the root.
- All searching operations incorporate a **splay**. (**find, findMin** and **findMax**)
- To perform deletion, we access the node to be deleted, which puts the node at the root. If it is deleted, we get two subtrees, **L** and **R** (left and right). If we find the largest element in **L**, using a **findMax** operation, its largest element is rotated to **L**'s root and **L**'s root has no right child. We finish the remove operation by making **R** the right child of **L**'s root. An example of the remove operation is shown below:

  Example: The remove operation applied to node **6**:
  - First, **6** is splayed to the root, leaving two subtrees;
  - A **findMax** is performed on the left subtree, raising **5** to the root of the left subtree;
  - Then the right subtree can be attached (not shown).



- The cost of the remove operation is **two splays**.