Splay Trees

By: S. Hassan Adelyar

Splay Trees

Splay Trees

- **Support all** of the **BST operations** but **does not guarantee O (Log n) worst-case** performance.
- Its **bound** is **amortized**, meaning, although **individual operations** can be **expensive**, any **sequence** of **operations** is guaranteed to be **logarithmic**.
- Because this is a **weaker guarantee** than that provided by **balanced BST**, **only the data** and two **references** per node are required for each item and the **operations** are somewhat **simpler**.

Splay Trees

- Although **balanced BST** provide **logarithmic** worst-case running-time per operation, they have several **limitations**:
  - Require **storing** an extra **balancing information**
  - They are **complicated** to implement. As a result, **insertions** and **deletions** are **expensive** and potentially **error-prone**.
  - We don’t win when **easy inputs** occur.

Splay Trees

- The **performance** of a **balance BST is improvable**. That is, there **worst-case, average-case, and best-case** performance are essentially **identical**.
- An **example** is a **find operation** for some item **X**. It is reasonable to expect not only that the **cost** of the **find** will be **logarithmic**, but also that if we perform an **immediate second find for X**, the second access will be cheaper than the first. In a **red-black** trees this is **not true**.
- We would also expect that if we perform an **access of X, Y, and Z**, then a **second set of accesses** for the same sequence would be **easy**.
- **90-10 rule**.

Splay Trees

- The **90 -10 rule** has been used for many years in **disk I/O** system.
- A **cache** stores in main memory the contents of some of the **disk blocks**.
- **Browsers** use the same idea: a **cache** stores locally the **previously** visited **Web Pages**.

Amortized Time Bounds

Splay Trees

- There is, however, a reasonable **compromise**: **O(N) time** for a **single** access may be **acceptable** as long as it does not happen **too often**. In particular any **M operations** take a total of **O (MLog N) worst-case** time, then the fact that **some operations** are **expensive** might be **inconsequential**.

Splay Trees

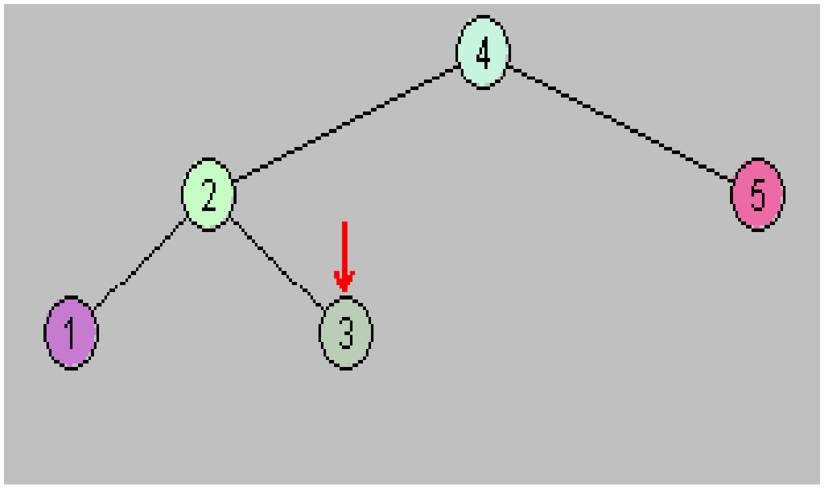
- When we can show a **worst-case** bound for a **sequence of operations** that is **better** than the corresponding bound obtained by considering **each operation separately**, the running time is said to be **amortized**.
- Some **single operations** may take **more** than logarithmic time.
- However, **amortized** bounds are **not always acceptable**. Specifically, if a **single bad operation** is too **time-consuming**, then we really need **worst-case bounds** rather than amortized bounds.

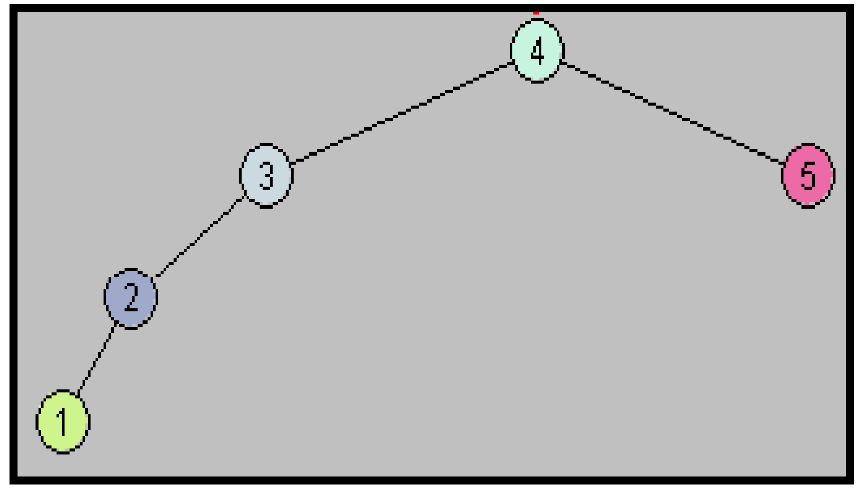
A simple self-adjusting strategy (that does not work)

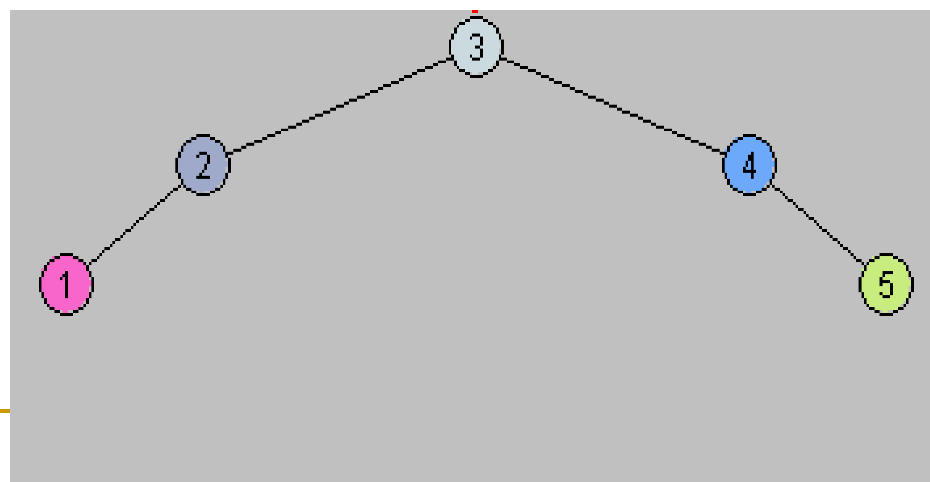
Splay Trees

- The **easiest** way to **move** an item toward the **root** is to **rotate** it continually **with its parent** until it becomes a root node.
- Then, if the item is **accessed** a **second time**, the second access is **cheap**.
- Even if a few other operations intervene before the item is **re-accessed**, that item will remain **close** to the **root** and thus will be quickly found.
- This process is called **rotate-to-root** strategy.

Splay Trees







Splay Trees

- Future **access** to **node 3** is cheaper. But node **4** and **5** each move **down** a level.
- This means that if **access patterns** do not follow the **90-10** rule, it is possible for a **long sequence** of **bad accesses** to occur.
- As a result, the **rotate-to-root** rule will not have **logarithmic amortized** behavior; this will be **unacceptable**.

Basic Bottom-up Splay

Splay Trees

- Achieving **logarithmic amortized cost** seems **impossible** because when we move an item to **root** via **rotations**, other items are pushed deeper.
- It means there would **always** be some **very depth nodes**, if **no balancing information** is maintained.
- There is a simple **fix** to the **rotate-to-root** strategy that allows the **logarithmic amortized bound** to be obtained. The resulting rotate-to-root strategy is called **splaying**.

Splay Trees

- Let **X** be a **non-root** node on the access **path** on which we are **rotating**.
- If the **parent** of **X** is the **root** of the **tree**, we merely rotate **X** and the root as shown in figure 21.4.
- This is the **last rotation** along the access path, and it **places X** at the **root**.
- This is a **zig** case.

Splay Trees

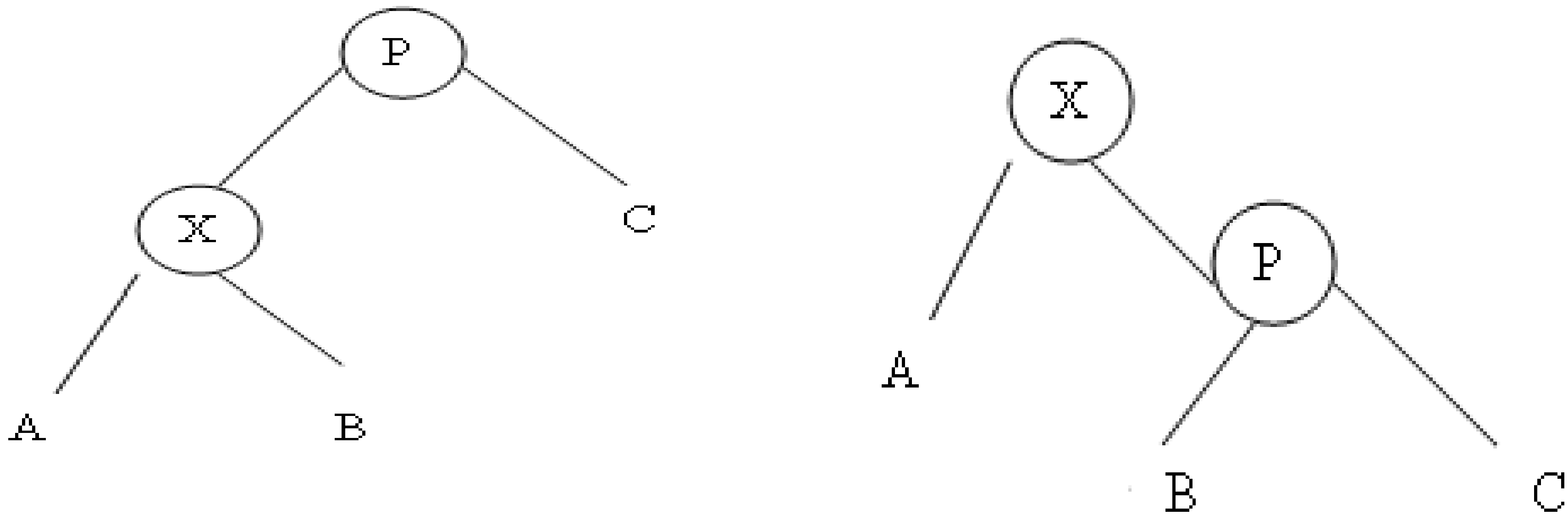


Figure 21.4 Zig case (Normal single rotation)

Splay Trees

- Otherwise, **X** has both a **parent P** and a **grandparent G**, and there are **two cases** plus **symmetries** to consider.
- **Zig-zag** case, which corresponds to the **inside** case for **AVL** trees. Here **X** is a **right child** and **P** is a **left child** (or **vice versa**). We perform a **double rotation**, exactly like an **AVL** double rotation, as shown in figure 21.5.
- In **figure 21.1**, the splay at node **3** is a single **zig-zag** rotation.

Splay Trees

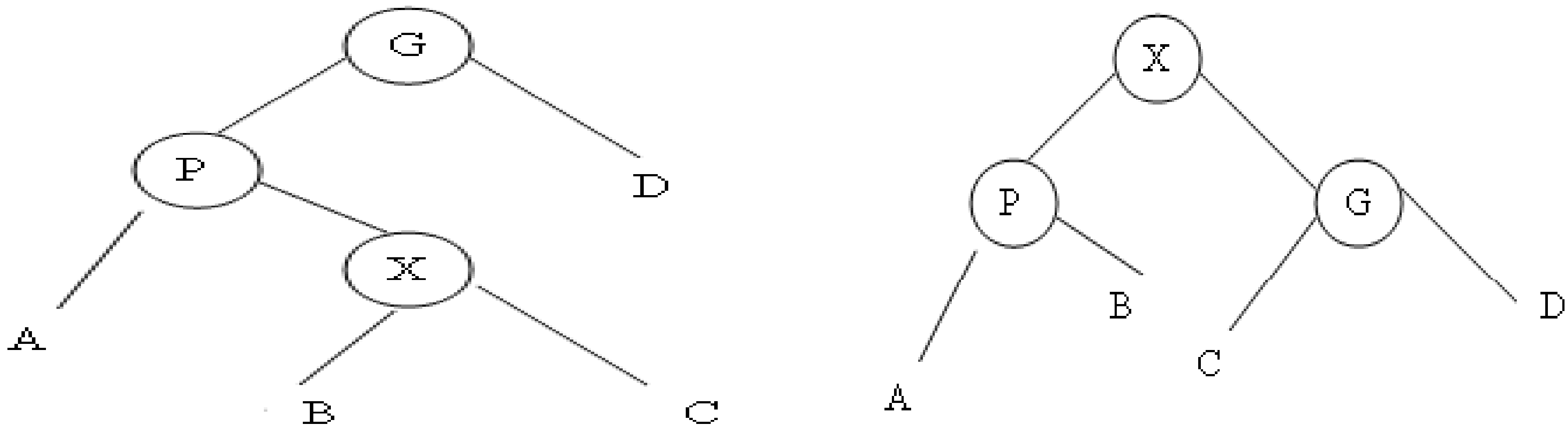


Figure 21.5 Zig-zag case (some as a double rotation); the symmetric case has been omited.

Splay Trees

- **Zig-zig** case, which is the **outside** case for **AVL** trees. Here, **X** and **P** are either **both left** children or **both right children**. In this case, we **transform** the **left-hand** tree of figure 21.6 to the **right-hand** tree.
- This **zig-zig** splay rotates between **P and G** and then **X and P**.

Splay Trees

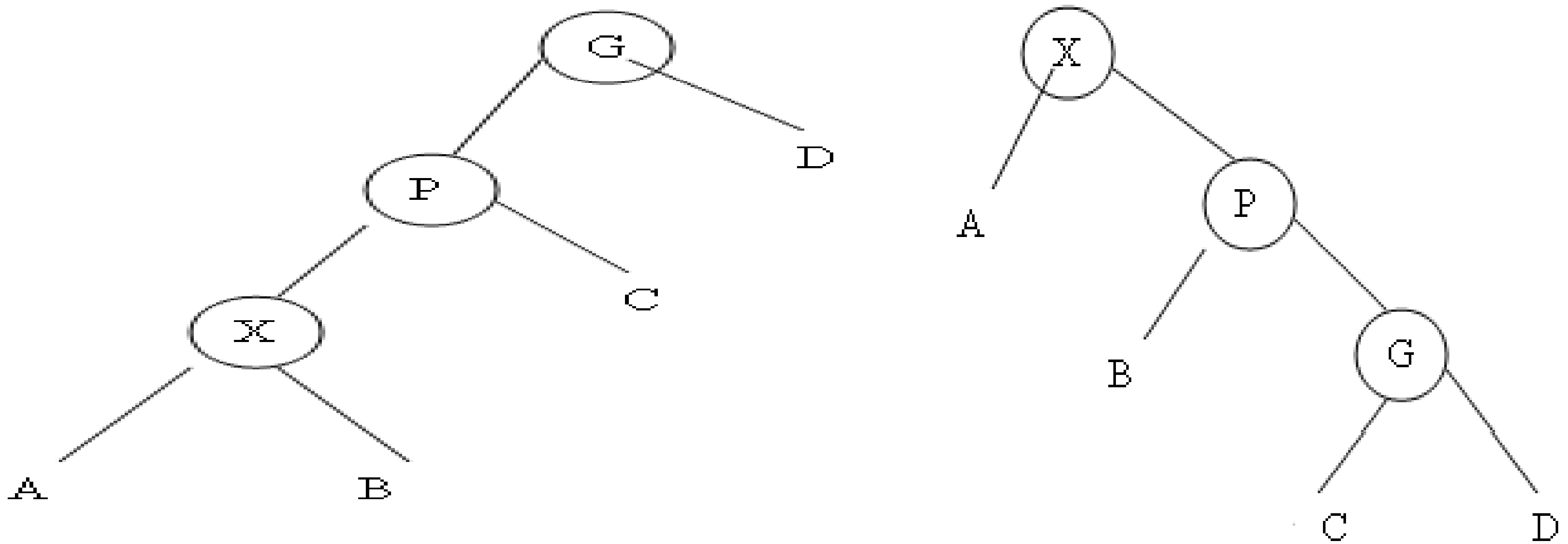


Figure 21.6 Zig-zig case (this is unique to the splay tree); the symmetric case has been omited

Splay Trees

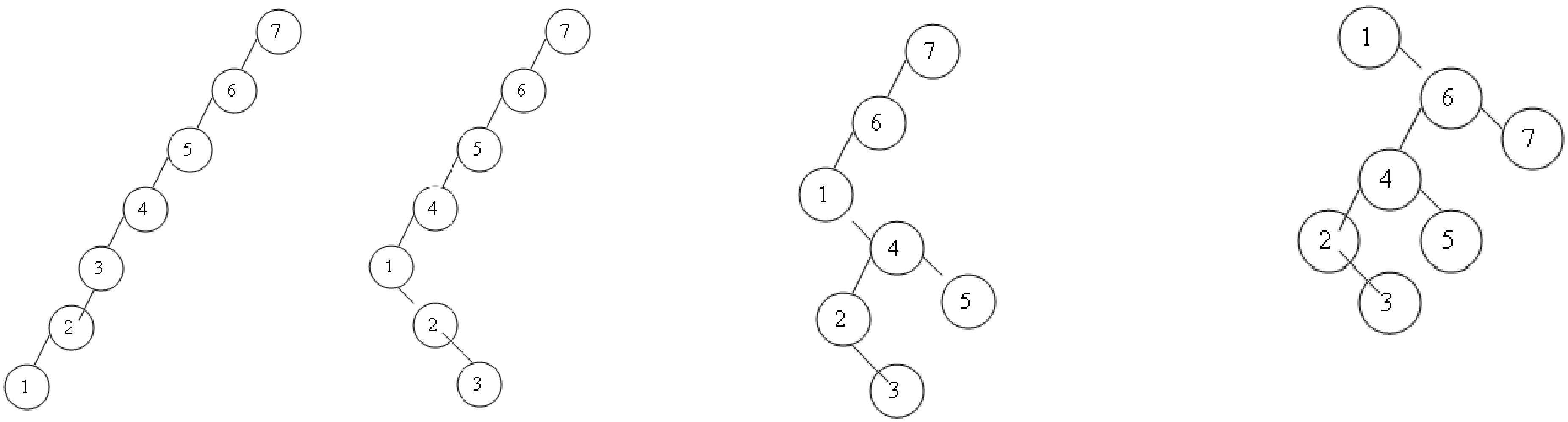
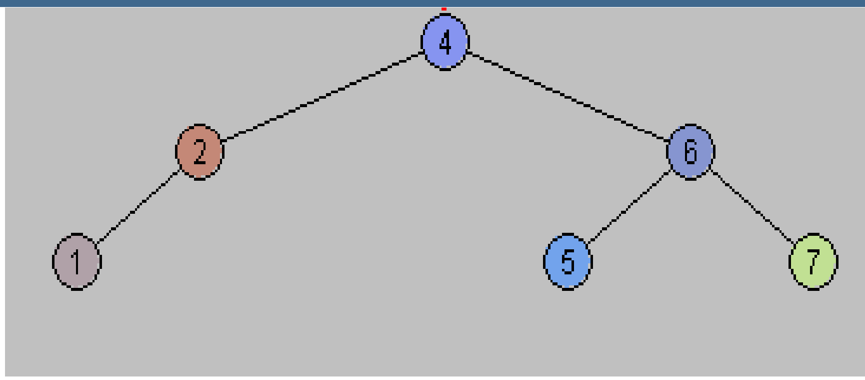


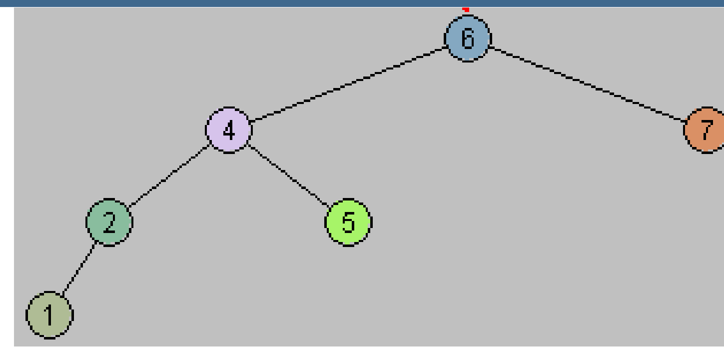
Figure 21.7 Result of splaying at node 1 (three zig-zigs and a zig)

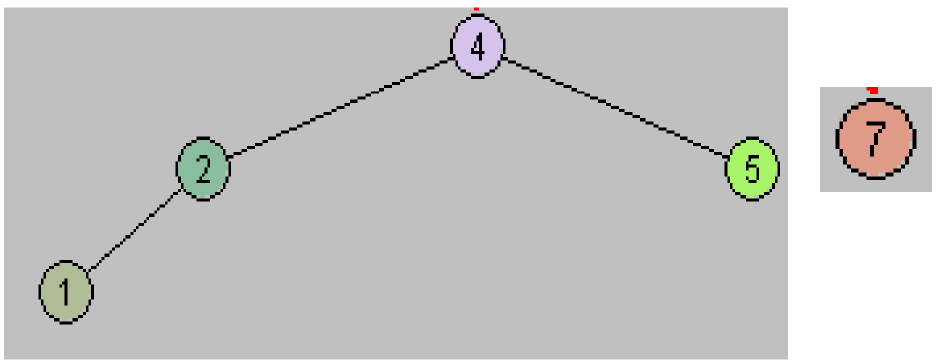
Splay Trees

- **Splaying** not only moves the accessed node to the **root**. It also roughly **halves** the **depth** of most nodes on the access path.

Splay Trees







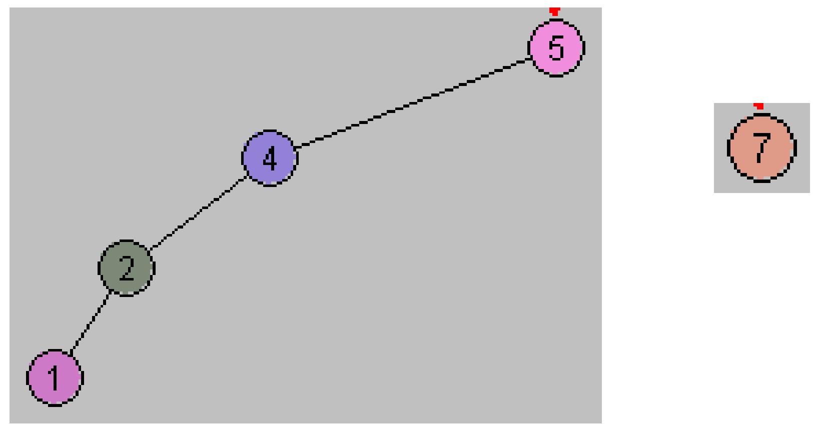


Figure 21.8 The remove operation applied to node 6: first 6 is splayed to the root, thus leaving two sub-trees; a findMax on the left sub-tree is performed, raising 5 to the root of the left sub-tree; then the right sub-tree can be attached (not shown)

Analysis of Bottom-up Splaying

- The **analysis** of splay tree algorithm is **complicated** because **each splay vary** from a **few rotations** to **O (N)** rotations. Furthermore, unlike with balanced search trees, **each splay changes** the **structure** of the **tree**. This section proves that the **amortized** cost of a splay is at most **3log N+1 single rotations**. The splay tree's amortized bound guarantees that any **sequence of M splays** will use at most **3Mlog N+M tree rotations**, and consequently any sequence of **M operations** starting from an empty tree will take a total of at most **O (M log N) time**.
- To **prove** this bound, we introduce an **accounting function** called the **potential function**. The potential function is not maintained by the algorithm. Rather it is merely an **accounting device** that aids in establishing the required time bound. Its choice is not obvious and is the result of a large amount of trial and error. See pages 624 – 630.

Top-down Splay Trees

Splay Trees

- **Bottom-up splay** require **two pass**. This can be done either by maintaining **parent references**, by **storing** the **access path** on a **stack**, or by using a **clever trick** to store the path using the available references in the accessed nodes.
- Unfortunately, all of these methods require a substantial amount of **overhead**, and we must handle many **special cases**. This section describes a **top-down splay tree** that maintains the **logarithmic amortized bound**. The top-down procedure is **faster** in practice and uses only constant extra space. It is the method recommended by the **inventors** of splay tree.

Data Structures and Algorithms

Splay Trees

- As we **descend** the tree in our search for some **node X**, we must take the **nodes** that are on the **access path** and **move** them and their **sub-trees** out of the way. We must also **perform** some **tree rotations** to guarantee the **amortized time bound**. At **any point** in the middle of the splay, there is a **current node X** that is the **root** of its **sub-tree**; this is represented in the diagrams as the **middle tree**. Tree **L** stores nodes that are **less than X**; similarly, tree **R** stores nodes that are **larger than X**. Initially, **X** is the **root** of **T**, and **L** and **R** are empty.

Splay Trees

- Descending the tree **two levels** at a time, we encounter a **pair of nodes**. Depending on whether these nodes are **smaller** than X or **larger** than **X**, they are placed in **L** or **R** along with **sub-trees** that are not on the access path to **X**. Thus the **current node** on the search path is always the **root** of the **middle tree**. When we finally **reach X**, we can then **attach L** and **R** to the **bottom** of the of the **middle tree**. As a result, **X** will have been **moved** to the **root**. The issue then is how nodes are placed into **L and R** and how the reattachment is performed at the end. This is what the tree in figure **21.9** are illustrating.

Splay Trees

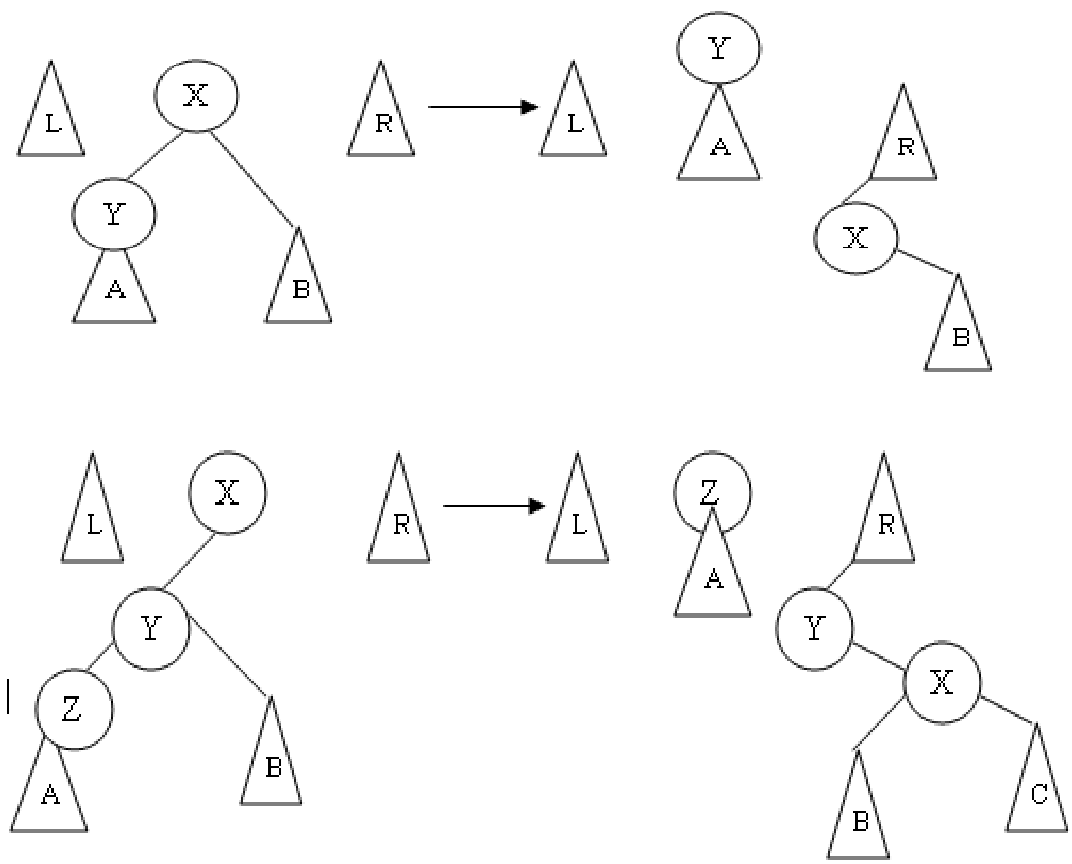


Figure 21.9 continue (see next page)

Splay Trees

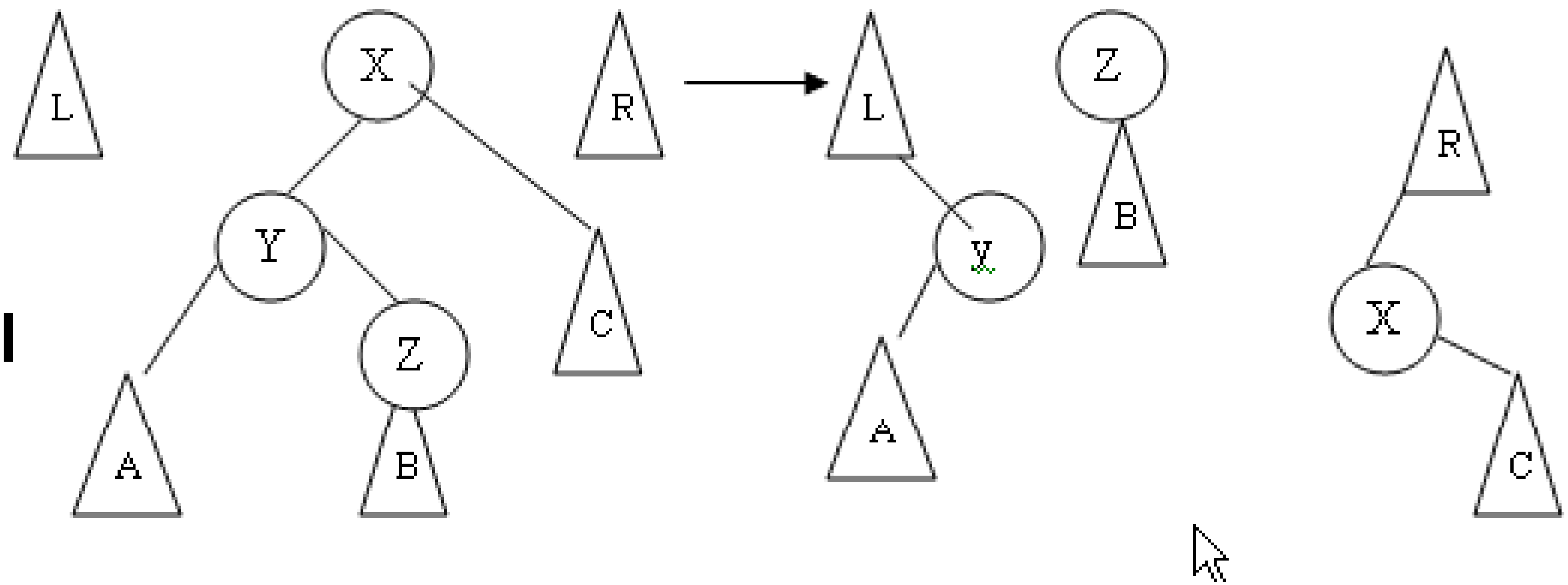


Figure 21.9 Top-down splay rotations; zig(top), zig-zig (middle), and zig-zag (bottom)

Splay Trees

- In all the **pictures**, **X** is the **current node**, **Y** is its **child**, and **Z** is a **grandchild**.
- If the **rotation** should be a **zig**, then the tree **rooted** at **Y** becomes the **new root** of the **middle tree**. **X** and **sub-tree B** are attached as a **left child** of the smallest item in **R**; **X's left child** is logically made null. As a result, **X** is the new smallest element in **R**, thus making future attachment easy.
- Notice carefully that **Y** does not have to be a **leaf** for the **zig case** to apply. If the item sought is found in **Y**, a **zig case** will apply even if **Y** has children. A **zig case** also applies if the item sought is **smaller** than **Y** and **Y** has no **left child**, even if **Y** has a **right child**, and also for the symmetric case.

Splay Trees

- A **similar** discussion applies to the **zig-zig case**. The crucial point is that a **rotation between X and Y** is performed. The **zig-zag** case brings the bottom node **Z** to the **top** of the **middle tree** and attaches **sub-trees X and Y** to **R** and **L**, respectively. Note that **Y** is attached to, and then becomes, the largest item in **L.**
- The **zig-zag** step can be simplified somewhat because no rotations are performed. Instead of making **Z** the **root** of the **middle tree**, we make **Y** the root. This is shown in **figure 21.10**. This simplifies the coding because the action for the **zig-zag** case becomes identical to the **zig case**. This would seem advantages, since testing for a host of cases is time-consuming. The **disadvantages** is that a descent of only one level results in more iterations in the splaying procedure.
- Once we performed the **final splaying step**, then **L, R**, and the **middle tree** are arranged to form a **single tree,** as shown in **figure 21.11**. Notice carefully that the result is different from that obtained with **bottom-up** splaying. The crucial fact is that the **O (log N) amortized bound** is preserved.

Splay Trees

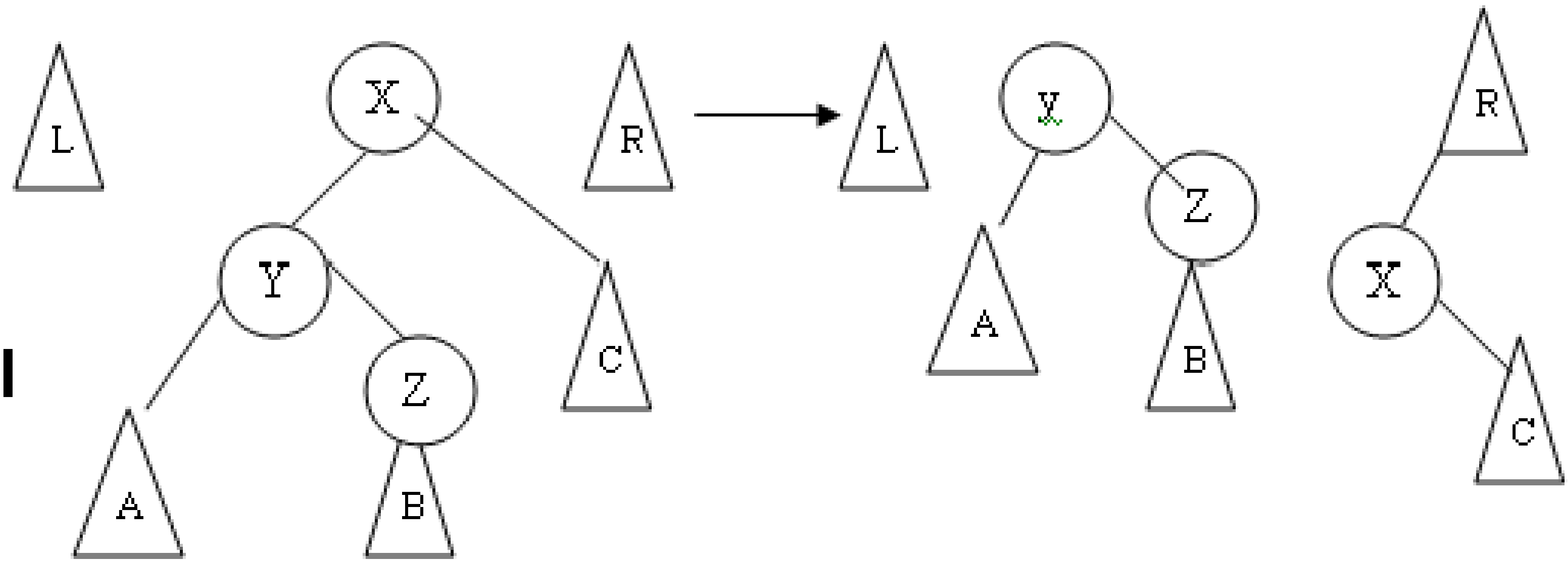


Figure 21.10: Simplified top-down zig-zag

Splay Trees

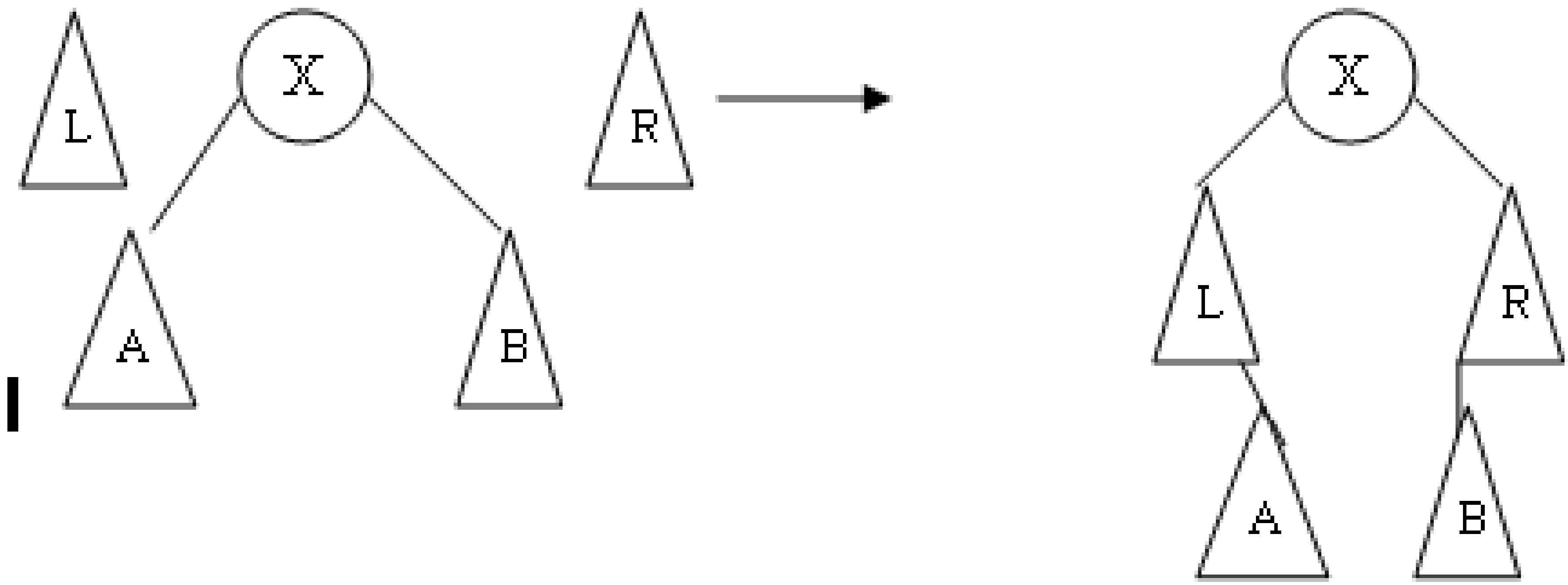


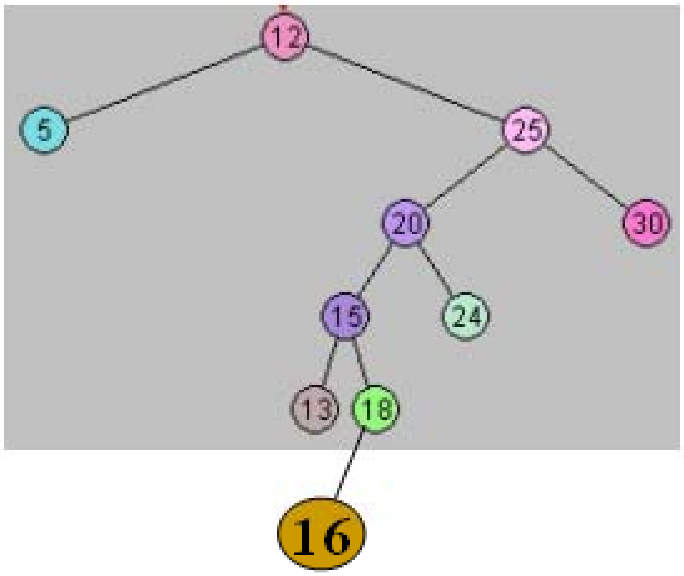
Figure 21.11: Final arrangement for top-down splaying

Splay Trees

- An example of the **simplified top-down splaying** algorithm is shown in **figure 21.12**. We attempt to access 19 in the tree. The first step is a **zig-zag**. In accordance with a symmetric version of figure 21.10, we bring the **sub-tree rooted** at **25** to the **root** of the **middle** tree and attach **12** and its **left sub-tree to L**. Next, we have a **zig-zig**: **15** is elevated to the **root** of the **middle tree**, and a rotation between **20** and **25** is performed, with the resulting **sub-tree** being attached to **R**. The search for **19** then results in a terminal **zig**. The middle's new root is **18,** and **15** and its **left sub-tree** are attached as a **right child** of **L's largest node**. The reassembly, in accordance with figure **21.11**, terminates the splay step.

Splay Trees

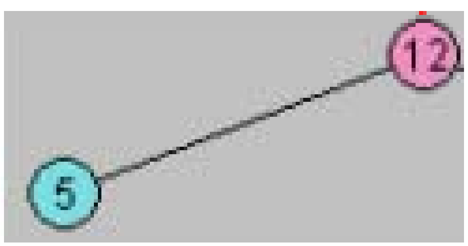
Empty

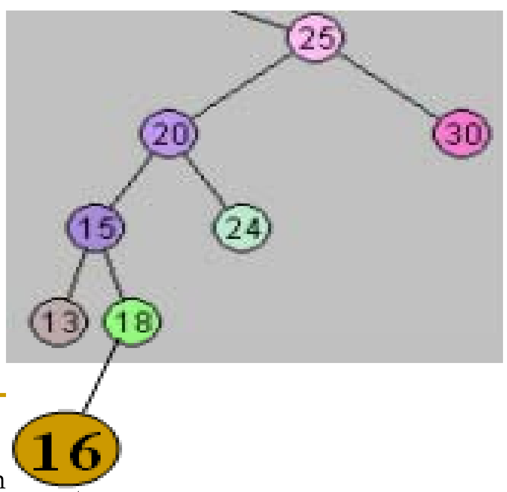


Empty

Empty

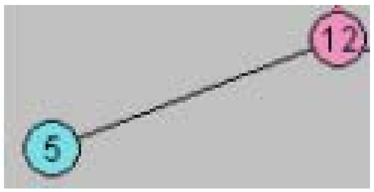
Simplified zig-zag

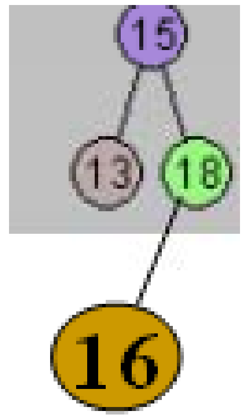


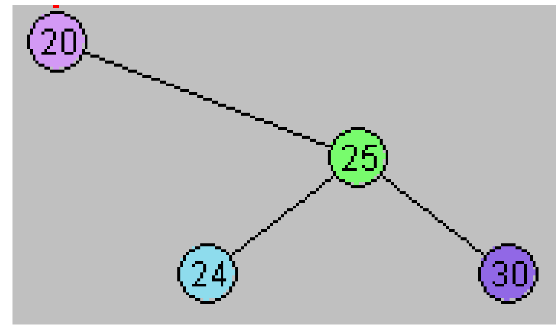


Splay Trees

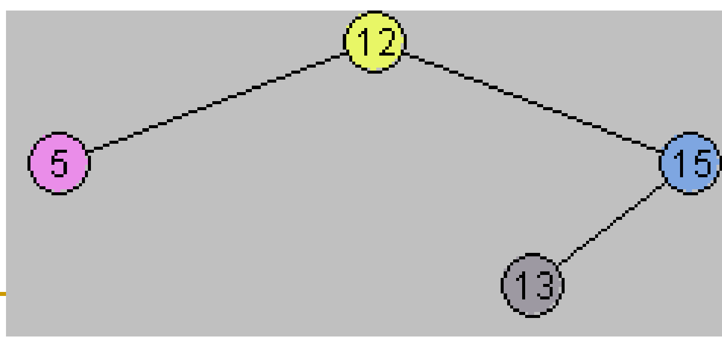
Zig-zig

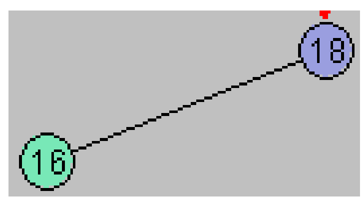


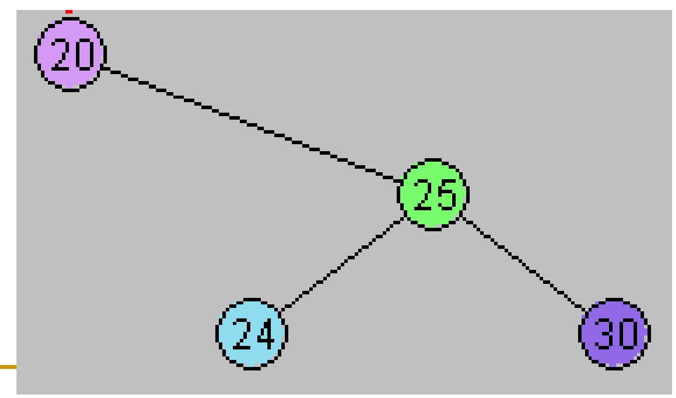




Zig







Splay Trees

Reassemble

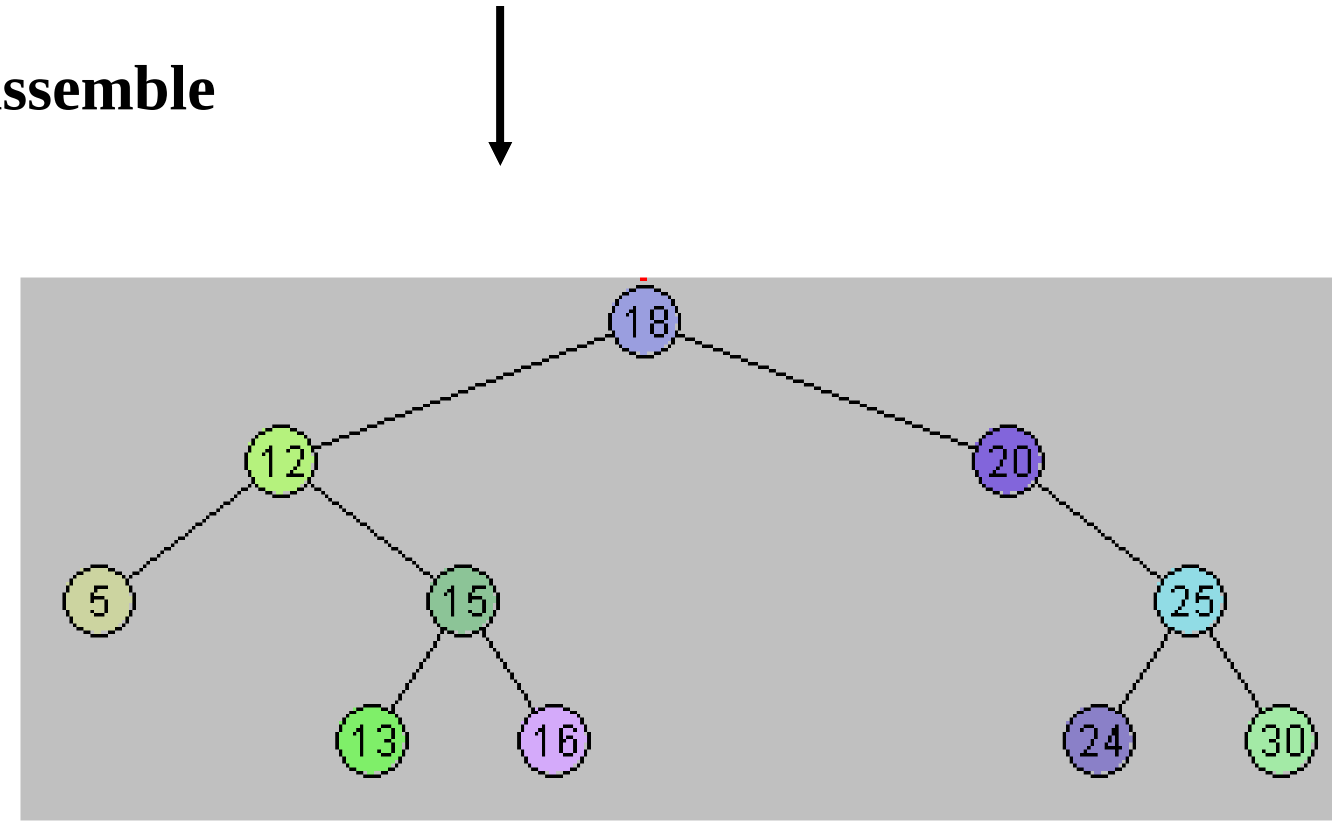


Figure 21.12: Steps in a top-down splay (accessing 19 in top tree)