Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Data Structures

Particular ways of storing data *to support special operations*.

Binary Search Trees (BSTs)

Storing fully-dynamic data to be searched fast.

Search Trees

- Search trees are data structures that support the following dynamic-set operations for a set A.
  - **Search, Predecessor, Successor** – to search for an element, or to report the predecessor / successor of the element.
  - **Minimum, Maximum**
    – to report the minimum / maximum element in a set.
  - **Insert, and Delete** – to insert or delete a given element.

A search tree can be used as a ***dictionary*** or a ***priority queue***.

Binary Search Tree (BST)

- A binary search tree (BST) is a binary tree with the following property that

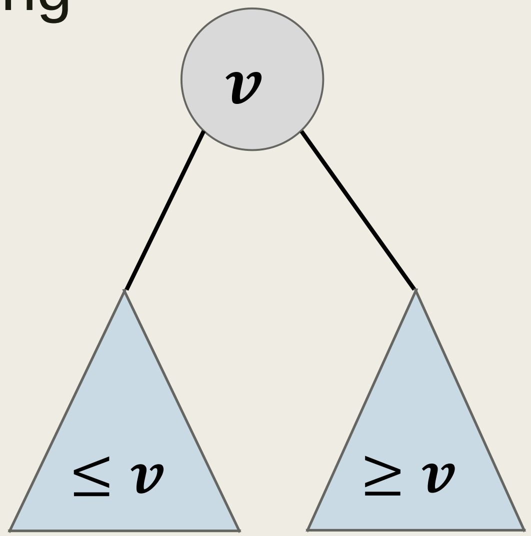
  - The *nodes* in the tree ***are comparable*** to each other.

  - (*BST property*)

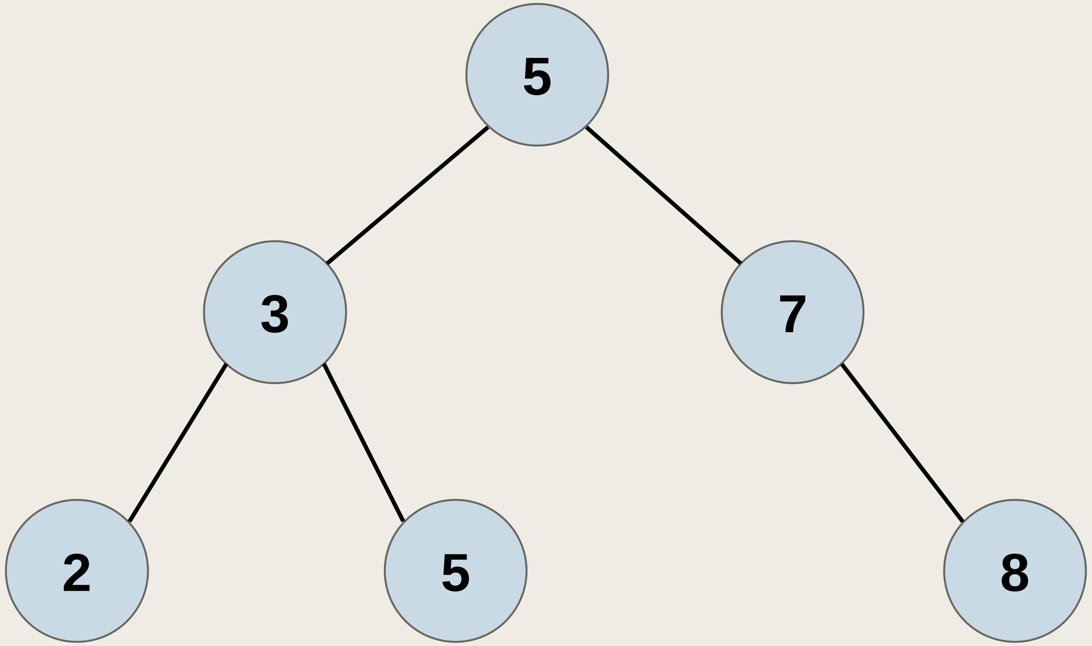
    For any node v in the tree and any node y,

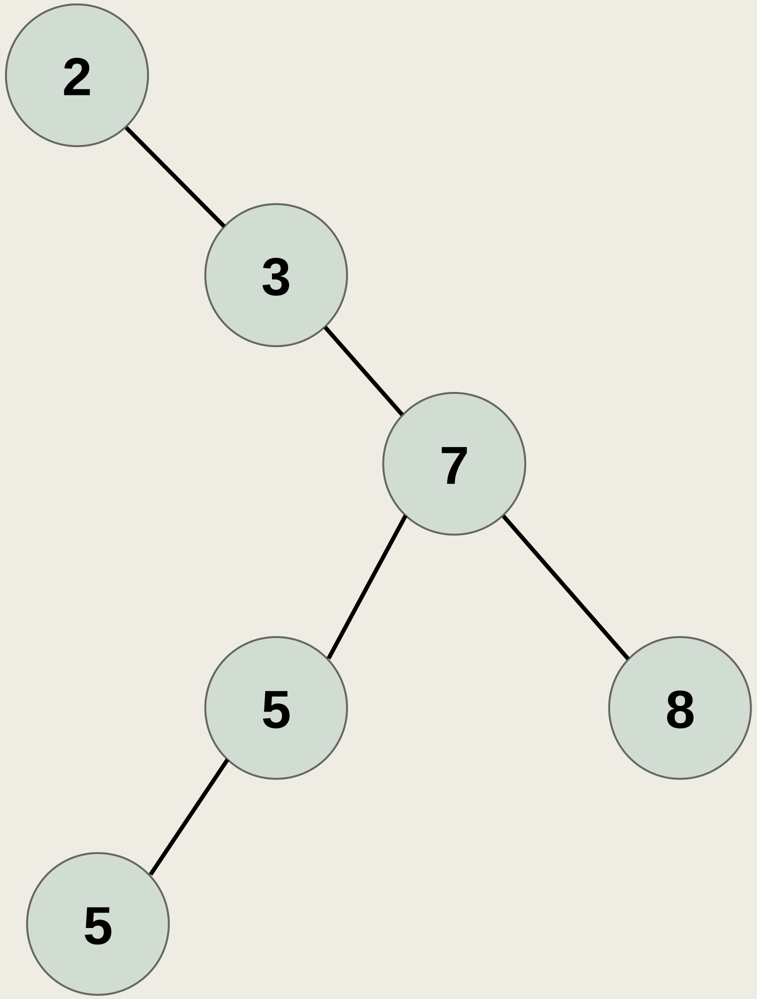
    - If y is a node in the ***left-subtree of v***, then $y \leq v$.

    - If z is a node in the ***right-subtree of v***, then $v \leq z$.



Binary Search Trees

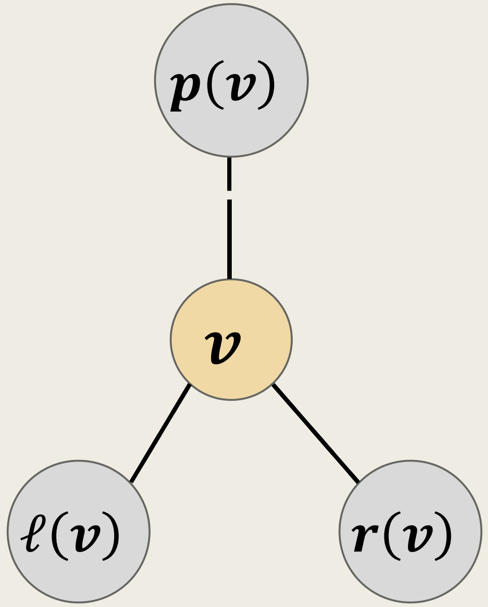




Two different ways for constructing a BST for $\{2, 3, 5, 5, 7, 8\}$.

Storing the Structure of a BST

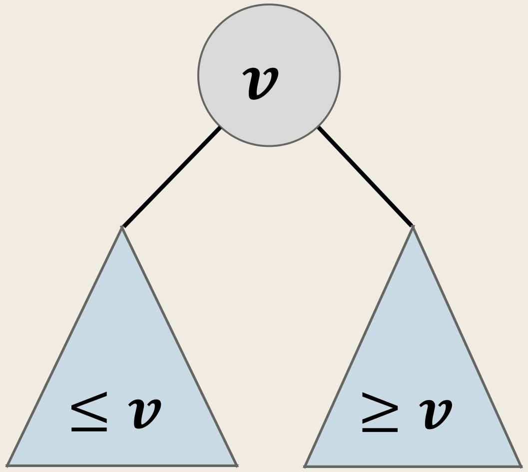
- Recall that, to store the structure of a binary tree T, we store the following information for each node v in the tree.
  1. The *parent node* of v, denoted $p(v)$.
  2. The *left-* and *right- children nodes* of v, denoted $\ell(v)$ and $r(v)$, respectively.
  3. The ***key value*** of v.
  4. Other *auxiliary information* (if needed).



- In addition, we need to record the root node $\mathrm{root}[T]$ of T.

Basic Operations for BSTs

Extracting the Sorted Order



- Given a BST T, the *sorted order* of the elements can be obtained via an ***in-order tree walk*** on T.

- In-order-Tree-Walk(v)

  -- to Print the sorted order of the elements in the BST rooted at v.

A. If $v = NIL$, then return.

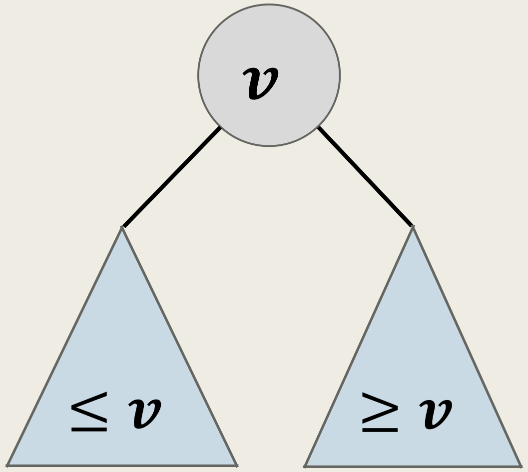
B. In-order-Tree-Walk(v.left).

   Print v.

   In-order-Tree-Walk(v.right).

This process takes $\Theta(n)$ time.

Searching for an Element



- As the name suggests,
  BSTs are *meant for searching*.
  - This process takes $O(h)$ **time**,
    where ***h* is the height of the BST**.

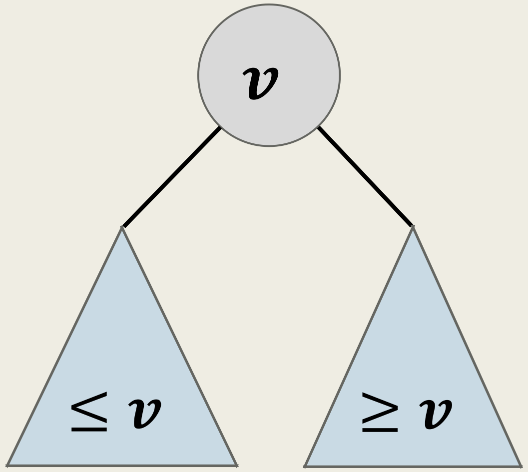
- Tree-Search(v, k) -- to research for an element k in the BST rooted at v.

A. If $v = NIL$ or $v = k$, then return v.

B. If $k < v$, then return Tree-Search($v.\text{left},\ k$).
Otherwise, return Tree-Search($v.\text{right},\ k$).

Can be unfolded
to a simple while loop.

Minimum / Maximum Element



- Finding the extremum element is straightforward.
  - This process takes $O(h)$ **time**, where h is the height of the BST.

- Tree-Minimum(v) - to return the minimum element in the BST rooted at v.

  A. if $v.\text{left} \neq NIL$, return Tree-Minimum($v \leftarrow v.\text{left}$). Otherwise, return v.

- Tree-Maximum(v) - to return the minimum element in the BST rooted at v.

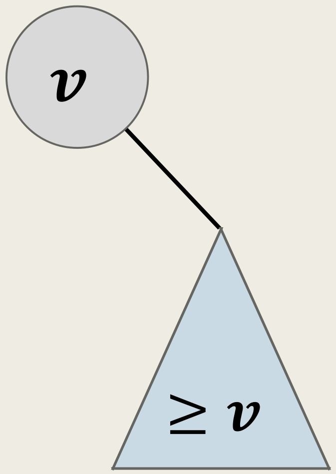
  A. if $v.\text{right} \neq NIL$, return Tree-Maximum($v \leftarrow v.\text{right}$). Otherwise, return v.

Can be unfolded
to a simple while loop.

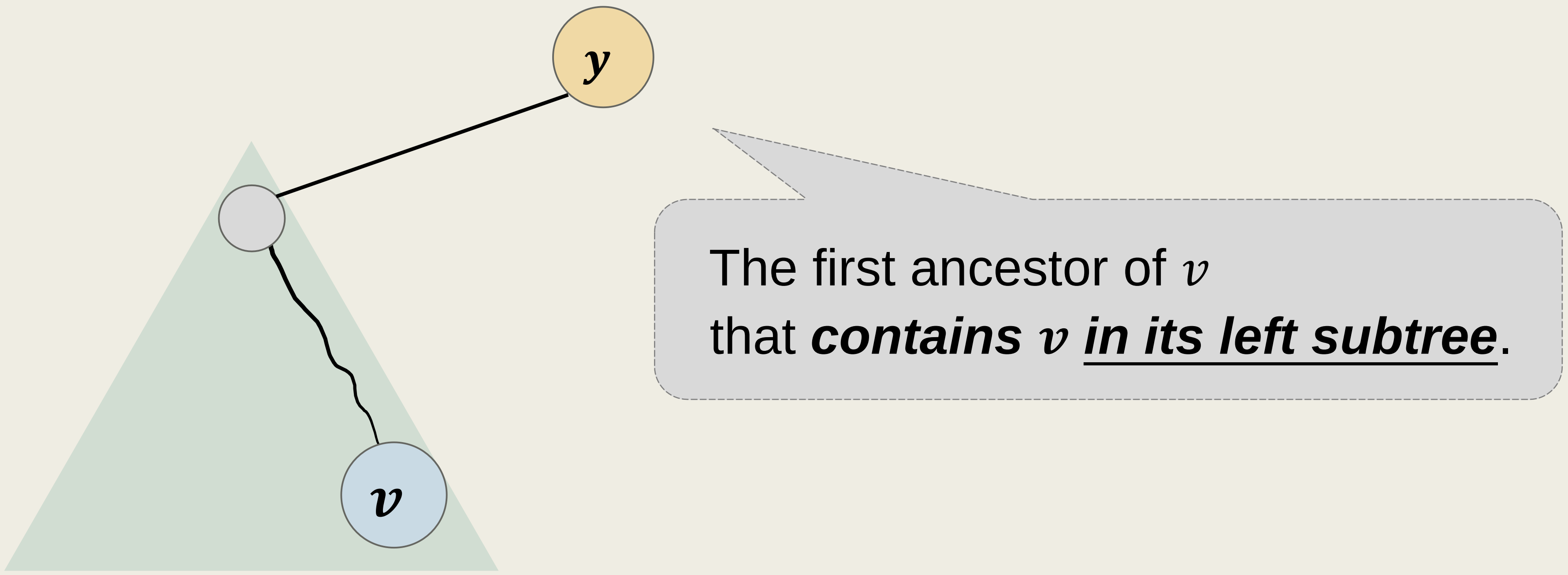
Finding the Successor of v

- To find ***the successor*** of a node v in the sorted order given by the ***in-order tree walk***.

- If v has a *non-empty right-subtree*,
  then its successor is ***the minimum*** in the right-subtree.



- To find ***the successor*** of a node v in the sorted order given by the ***in-order tree walk***.
- If v has no right-subtree, then its successor is "**the first element** ***to the right of*** v".
  - That is, the first ancestor that have v in its left-subtree.



- If v has a *non-empty right-subtree*,
  then its successor is ***the minimum*** in the right-subtree.

- If v has no right-subtree,
  then its successor is ***the first ancestor*** that have v *in its left-subtree*.

- Tree-Successor(v) -- to find the successor of v in the in-order walk order.

A. If $v.\text{right} \neq NIL$, then return Tree-Minimum($v.\text{right}$).

B. Otherwise, let $y \leftarrow v.\text{parent}$.

C. While $y \neq NIL$ and $v \neq y.\text{left}$, do
   set $v \leftarrow y$ and $y \leftarrow v.\text{parent}$.

D. Return y.

This process takes $O(h)$ time.

This procedure *mimics* ***the in-order walk*** after v.

Finding the Predecessor of a Node v

- Finding the predecessor is symmetric to finding the successor.
  - This process takes $O(h)$ time.

- Tree-Predecessor(v) -- to find the predecessor of v.

A. If $v.\text{left} \neq NIL$, then return Tree-Maximum($v.\text{left}$).

B. Otherwise, let $y \leftarrow v.\text{parent}$.

C. While $y \neq NIL$ and $v \neq y.\text{right}$, do
set $v \leftarrow y$ and $y \leftarrow v.\text{parent}$.

D. Return y.

This procedure mimics the in-order walk before v.

Insertion / Deletion for BSTs

Modifying a BST – Insertion & Deletion

Let T be a BST with root node r.

- In the following, we introduce procedures that can be used to
  - ***Insert** a new node* v into T.
  - ***Remove** an existing node* v from T.

- Since $\mathrm{root}[T]$ may change after these operations, *the procedures* are designed so that they ***return the new root*** of the tree T.

Inserting a Node v

Let T be a BST and v be a ***new node*** we wish to insert into T.

- Let r be the root of T.

  We ***search*** in T ***for a (leaf) position*** for v to reside.

  – If $v < r$, then inserting v in the left-subtree of r satisfies the BST-property.

    If $r.\mathrm{left} = NIL$, we can insert v at $r.\mathrm{left}$ directly.
    Otherwise, we have a recursive problem (to insert v into $r.\mathrm{left}$).

  – The argument for the case with $v \geq r$ is similar.

This process takes $\boldsymbol{O(h)}$ **time**.

- Tree-Insert(r, v)

  -- to insert v into the BST rooted at r and return the new root.

A. If $r = NIL$, then return v. // T was empty, so v is the new root.

B. If $v < r$ and $r.\mathrm{left} = NIL$,
   then set $r.\mathrm{left} \leftarrow v$ and $v.\mathrm{parent} \leftarrow r$, and return r.

C. If $v \geq r$ and $r.\mathrm{right} = NIL$,
   then set $r.\mathrm{right} \leftarrow v$ and $v.\mathrm{parent} \leftarrow r$, and return r.

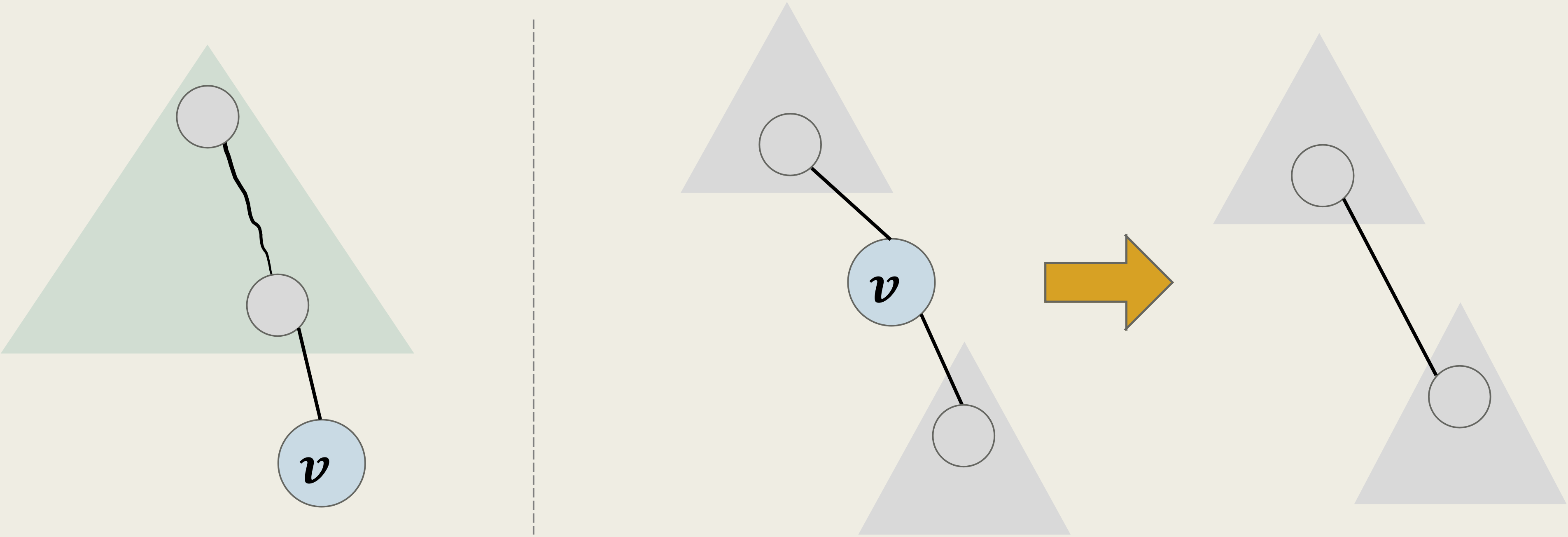
D. If $v < r$, then call Tree-Insert$(r.\mathrm{left}, v)$.
   Otherwise, call Tree-Insert$(r.\mathrm{right}, v)$.

E. Return r.

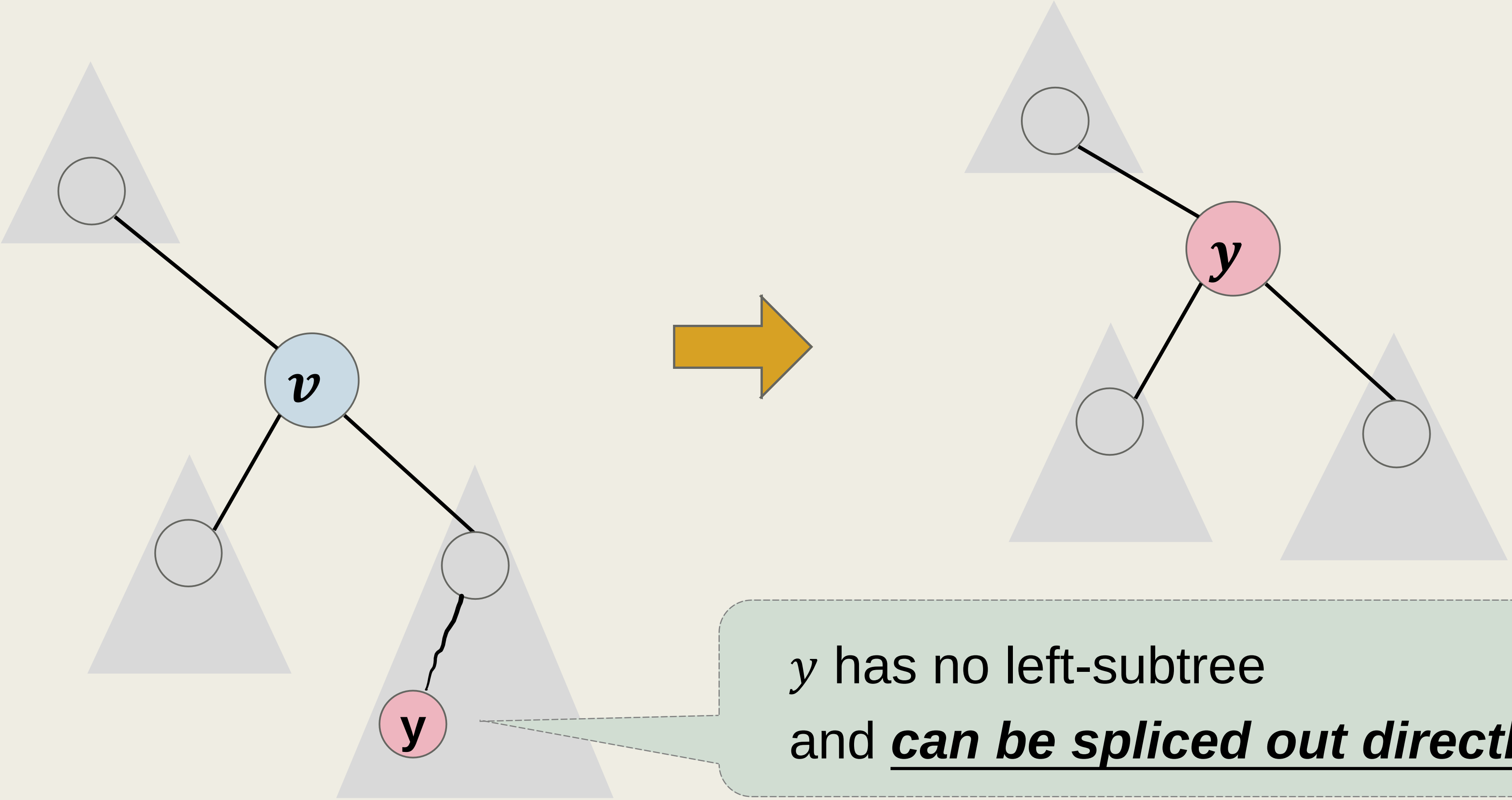
This process can be unfolded to a simple while loop.

Deleting a Node v

- Let T be a BST and $v \in T$ be a node we wish to remove.
- For this, we consider two different cases.
  - If v has ***<u>at most one child</u>***, then v can be ***<u>spliced out directly</u>***.



- For this, we consider two different cases.
  - If v has *two children*, then we find the successor y of v.
  - We **splice** $\boldsymbol{y}$ from the right-subtree of v and **replace** $\boldsymbol{v}$ **with** $\boldsymbol{y}$.



y has no left-subtree
and ***can be spliced out directly***.

- Tree-Delete(r, v)

  -- to delete v from the BST T_r rooted at r and return the new root.

A. If $v.\text{left} = NIL$ and $v.\text{right} = NIL$, then

   - If $v.\text{parent} = NIL$,   // v is the only node in T_r

     then return NIL.

   - If $v = v.\text{parent}.\text{left}$, then set $v.\text{parent}.\text{left} \leftarrow NIL$.

     Otherwise, set $v.\text{parent}.\text{right} \leftarrow NIL$.

   - If $v = r$, then return NIL.   // v is the root of T_r

     Otherwise, return r.

- Tree-Delete(r, v)

  -- to delete v from the BST rooted at r and return the new root.

A. Handle the case for $v.\text{left} = NIL$ and $v.\text{right} = NIL$.

B. If $v.\text{left} = NIL$ or $v.\text{right} = NIL$, then

   - Set $y \leftarrow v.\text{left}$ if $v.\text{left} \neq NIL$. // y is the child of v
     Otherwise, set $y \leftarrow v.\text{right}$.
   - Set $y.\text{parent} \leftarrow v.\text{parent}$.
   - If $v.\text{parent} = NIL$, then return y. // v is the root of T_r
   - If $v = v.\text{parent}.\text{left}$, then set $v.\text{parent}.\text{left} \leftarrow y$.
     Otherwise, set $v.\text{parent}.\text{right} \leftarrow y$.
   - Return r.

- Tree-Delete(r, v)

  -- to delete v from the BST rooted at r and return the new root.

A. Handle the case for $v.\text{left} = NIL$ and $v.\text{right} = NIL$.

B. Handle the case for $v.\text{left} = NIL$ or $v.\text{right} = NIL$.

C. // v has two children

   Let y ←Tree-Successor(v). // y has no left-subtree

   - Tree-Delete(r, y).
   - Replace v with y by copying the key and auxiliary information of y to v.
   - Return r.

This process takes **$O(h)$ time**.

Binary Search Trees – Summary

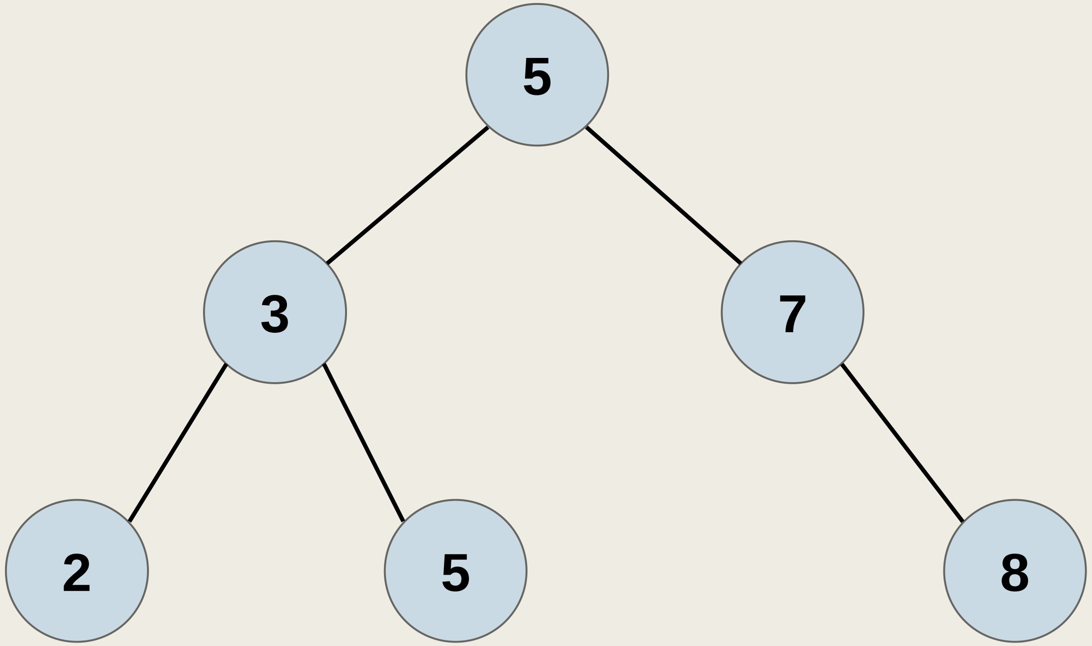
- Binary search trees support all of the following operations in $\boldsymbol{O(h)}$ **time**, where h is the height of the tree.
  - **Search, Predecessor, Successor** – to search for an element, or to report the predecessor / successor of the element.
  - **Minimum, Maximum**
    – to report the minimum / maximum element in a set.
  - **Insert, and Delete** – to insert or delete a given element.

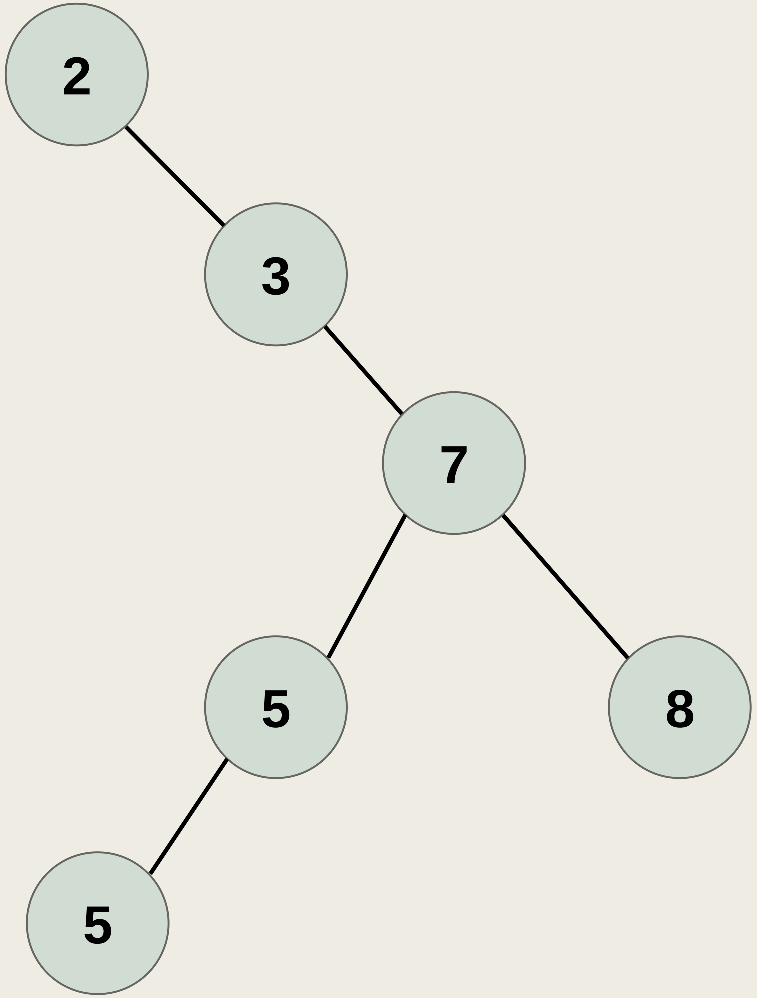
Binary Search Trees – Summary

- Binary search trees support all of the following operations in $O(h)$ time, where h is the height of the tree.

- As we have seen, the efficiency of BST operations is determined by the height of the tree.
  - Furthermore, the height of a BST for a set of n elements can range from $\Theta(\log n)$ to $\Theta(n)$, depending on the construction.

Binary Search Trees





Two different ways for constructing a BST for $\{2, 3, 5, 5, 7, 8\}$.

The Efficiency of BSTs

- As we have seen, the efficiency of BST operations is determined by the height of the tree.
  - Furthermore, the height of a BST for a set of n elements can range from $\Theta(\log n)$ to $\Theta(n)$, depending on the construction.

- Is it possible to guarantee an $O(\log n)$-height of the resulting BST?

  We will see two different approaches.

  1. ***Treap*** - Expected $O(\log n)$ height ***in the average case***.
  2. ***Red-Black Tree*** - $O(\log n)$ height ***in the worst-case***.

Balancing the Height of a BST

-- Tree Rotations

Fundamental Operations to ***Adjust the Height*** of a BST.

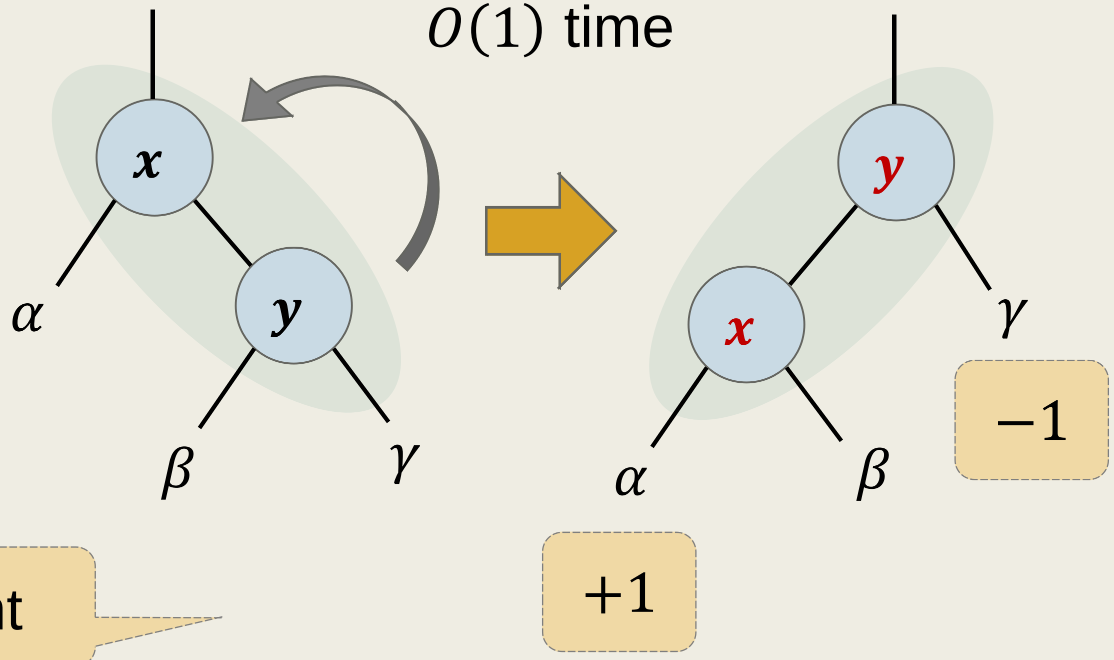
Tree Rotations

- Tree rotation is a fundamental operation that can be used to *adjust the height* of a BST *while **maintaining the BST property***.

- We have two types of tree rotations.
  - **Left rotation**.

    Works on ***a node*** $\boldsymbol{x}$ and ***its right-child*** $\boldsymbol{y}$.

$O(1)$ time



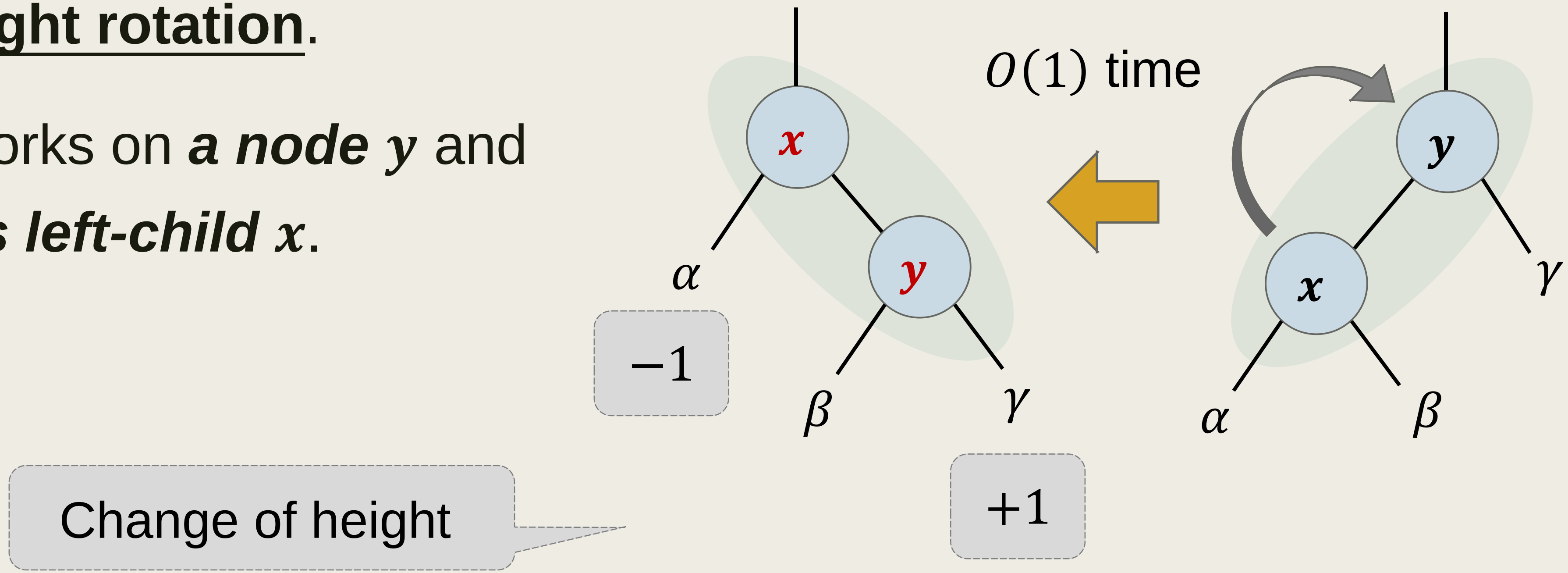
Change of height

Tree Rotations

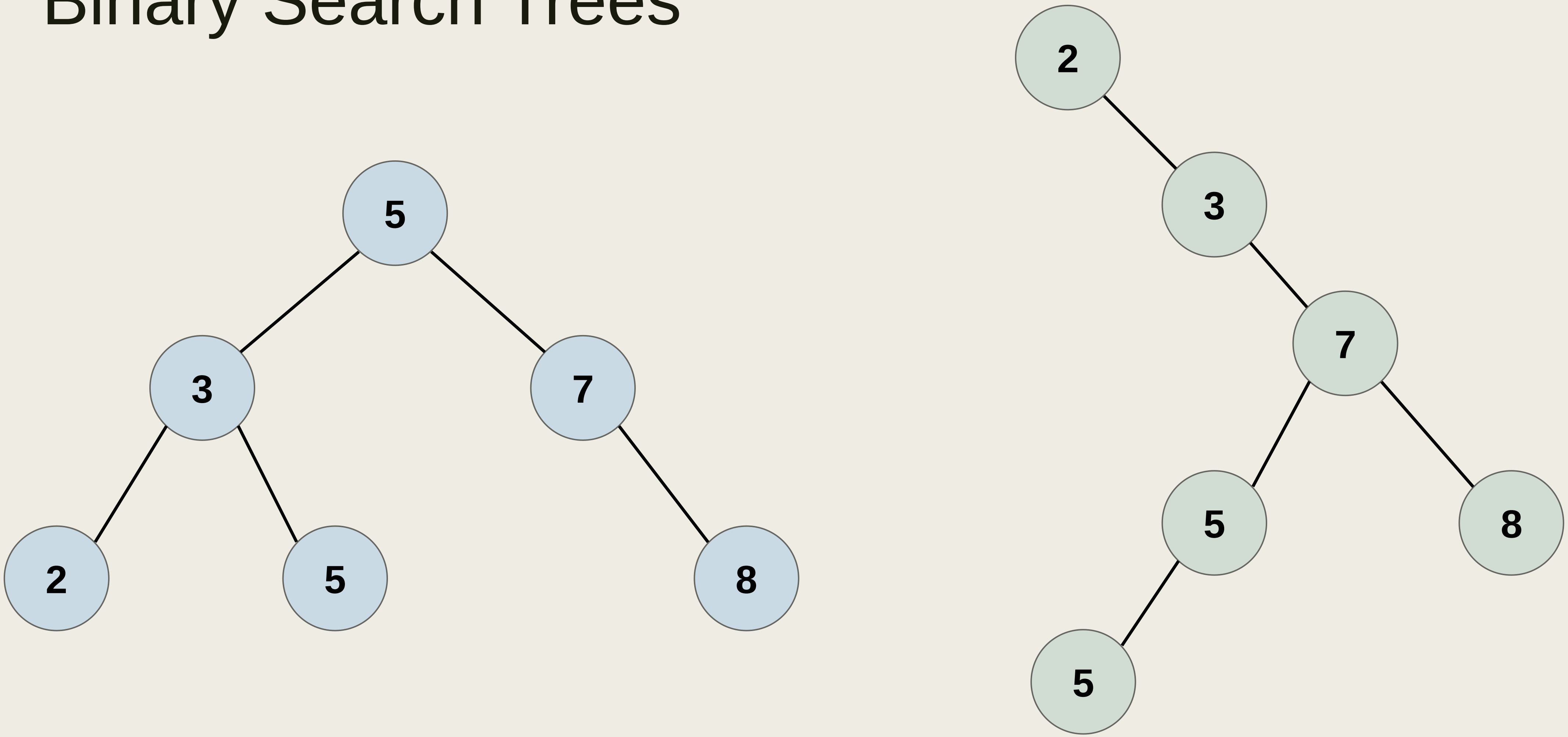
- Tree rotation is a fundamental operation that can be used to adjust the height of a BST *while maintaining the BST property*.

- We have two types of tree rotations.
  - **Right rotation**.

    Works on ***a node y*** and ***its left-child x***.

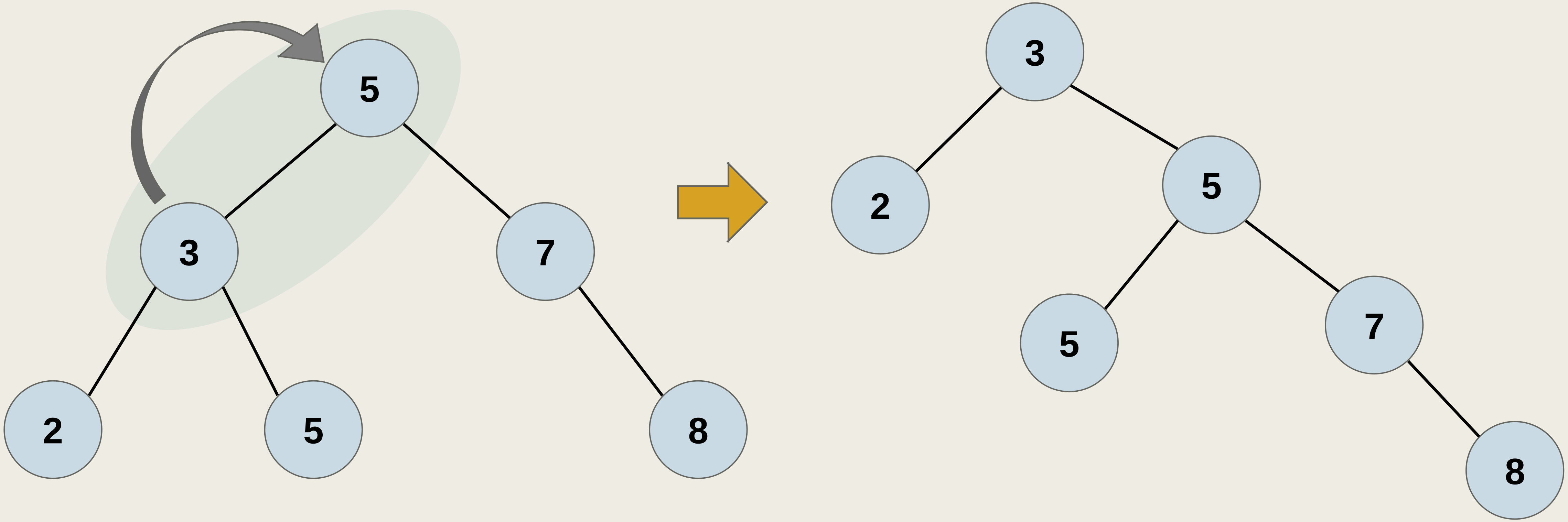


Binary Search Trees

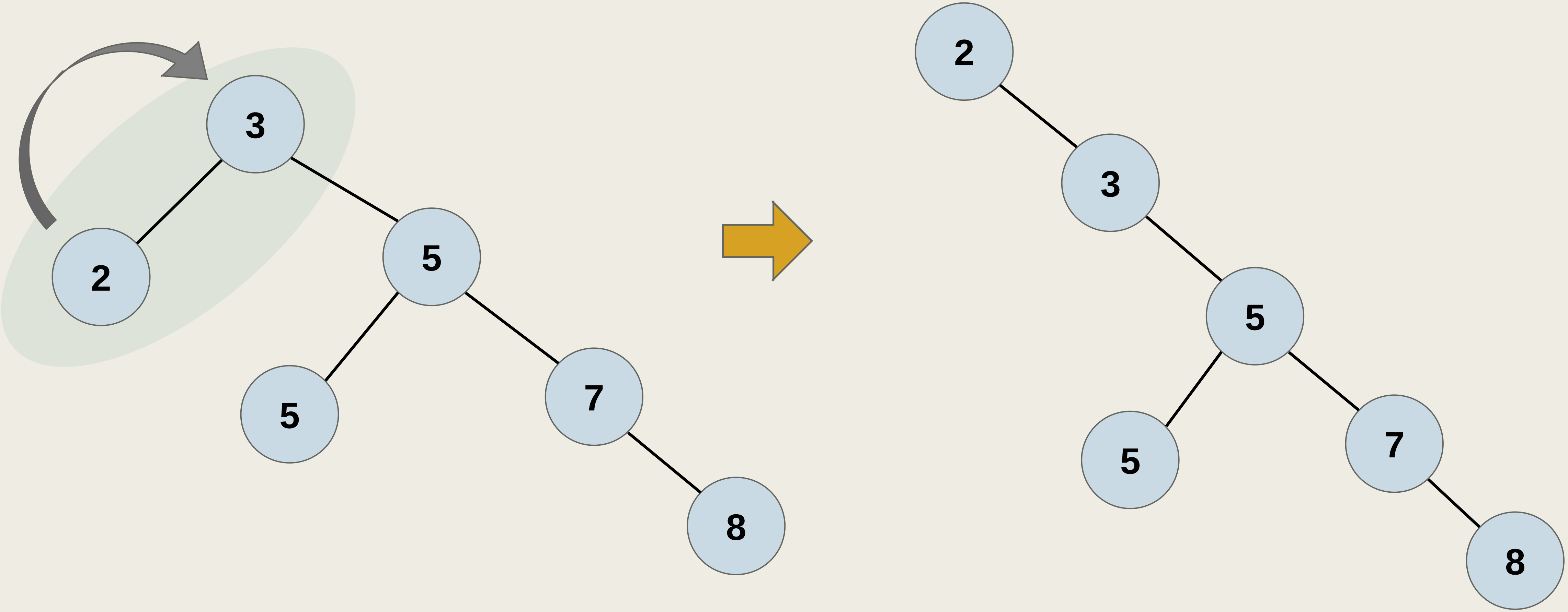


Two different ways for constructing a BST for $\{2, 3, 5, 5, 7, 8\}$.

Rotating From One to Another



Rotating From One to Another



Rotating From One to Another