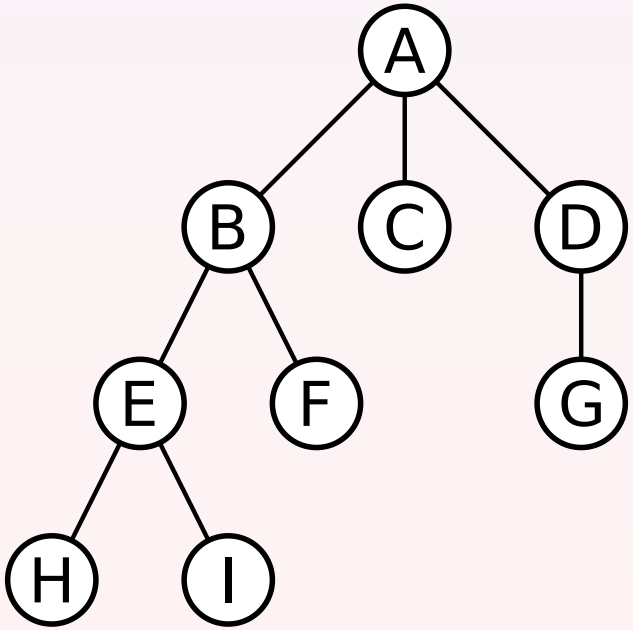
Binary search trees (BST)

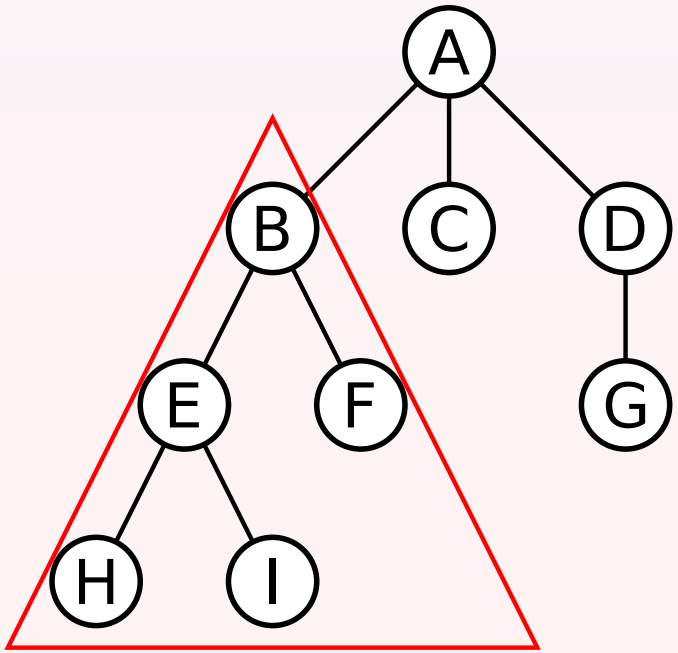
Tree

- A tree is a structure that represents a parent-child relation on a set of object.
- An element of a tree is called a node or vertex.
- The root of a tree is the unique node that does not have a parent (node A in the example below).
- A node is a leaf if it doesn't have children (C,F,G,H,I).
- A node which is not a leaf is called an inner node.

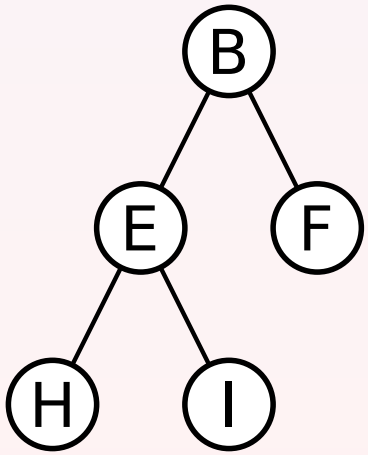


Tree

- A node w is a descendant of a node v if there are nodes $v_1, v_2, \ldots, v_t$, with $v_1 = v$ and $v_t = w$, such that v_{i+1} is a child of v_i for all i. v is called an ancestor of v.
- The subtree rooted at v is a tree containing v and its descendants.

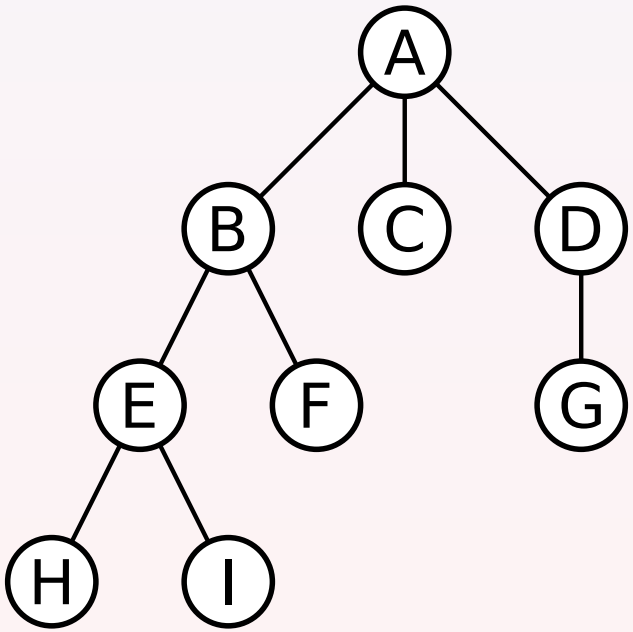


The subtree rooted at B



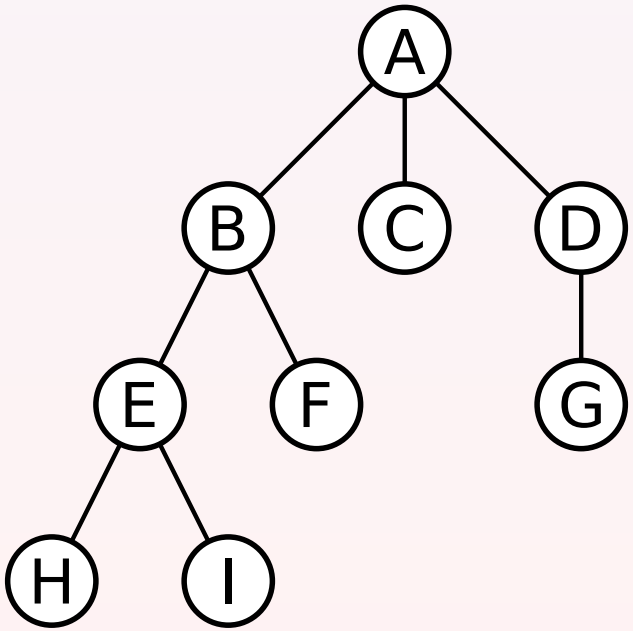
Depth

- The depth of a node v is the number of edges in the path from the root to v.
  In the example below, the depth of B is 1.



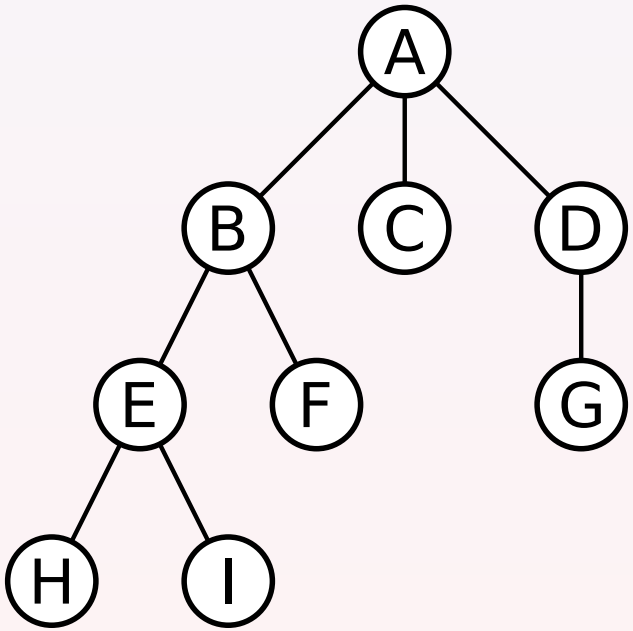
Height

- The height of a node v is the maximum number of edges in a path from v to a descendant of v.
  In the example below, the height of B is 2.
- The height of a tree is the height of its root $=$ the maximum depth of a leaf.
- The height of an empty tree is -1.



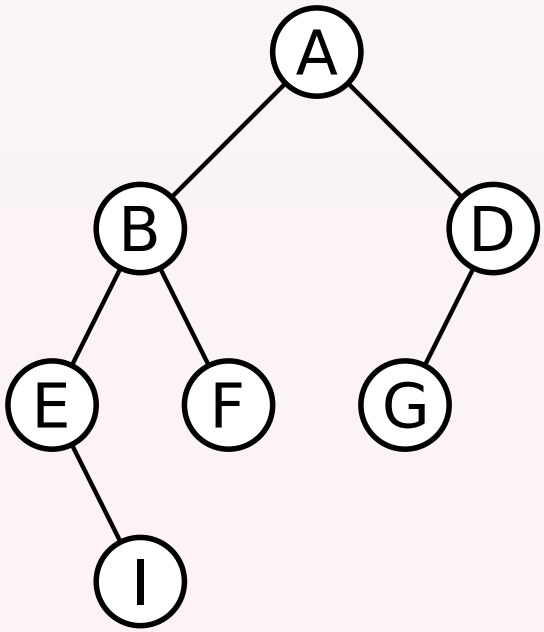
Degree

- The degree of a node v is the number of children of v. For example, degree(A) $= 3$.

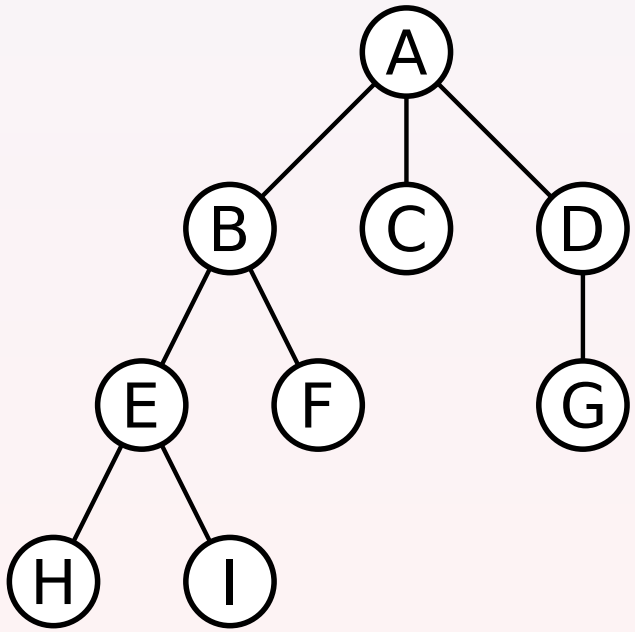


Binary tree

- A binary tree is a tree in which each node has at most two children.
- More precisely, a node can have a left child or not, and have a right child or not.



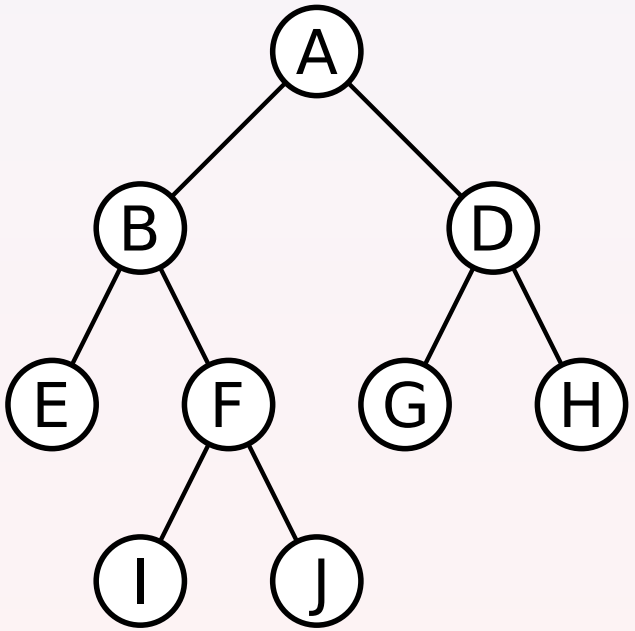
Binary



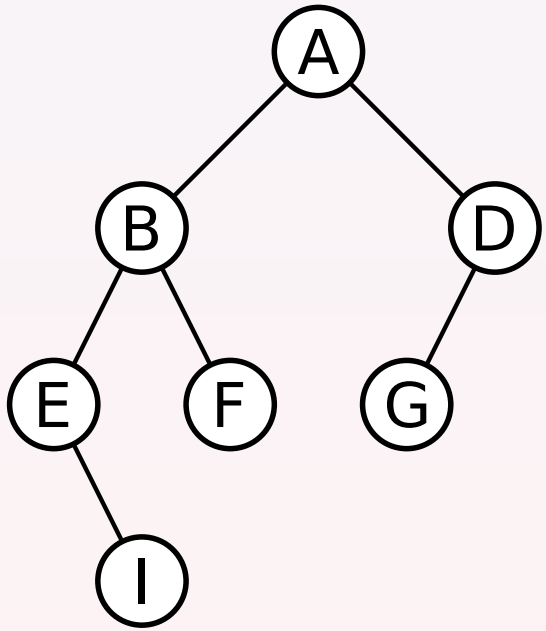
Not binary

Full binary tree

- A full binary tree is a binary tree in which each internal node has exactly two children.



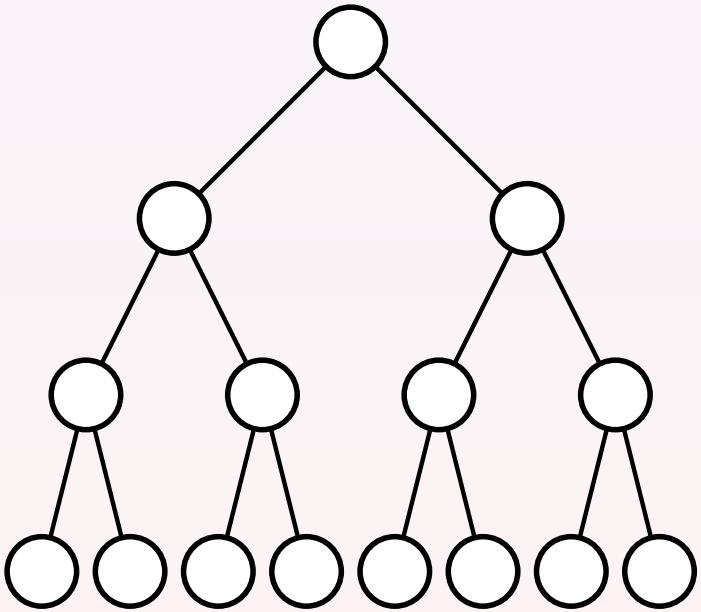
Full



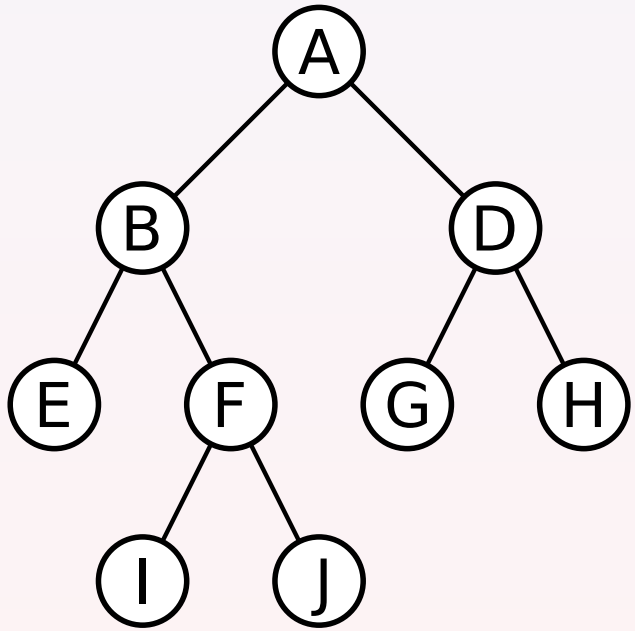
Not full

Perfect binary tree

- A perfect binary tree is a full binary tree in which all leaves have the same depth.



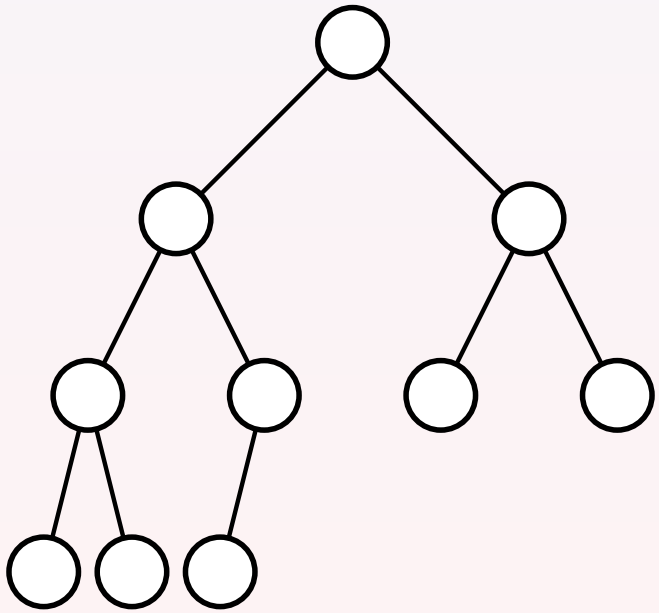
Perfect



Not perfect

Complete binary tree

- A complete binary tree is either a perfect binary tree, or a tree that is obtained from a perfect binary tree by deleting consecutive leaves of the tree from right to left.



Dynamic set ADT

A dynamic set ADT is a structure that stores a set of elements. Each element has a (unique) key and satellite date. The structure supports the following operations.

Search(S, k) Return the element whose key is k.

Insert(S, x) Add x to S.

Delete(S, x) Remove x from S (the operation receives a pointer to x).

Minimum(S) Return the element in S with smallest key.

Maximum(S) Return the element in S with largest key.

Successor(S, x) Return the element in S with smallest key that is larger than $x.\mathrm{key}$.

Predecessor(S, x) Return the element in S with largest key that is smaller than $x.\mathrm{key}$.

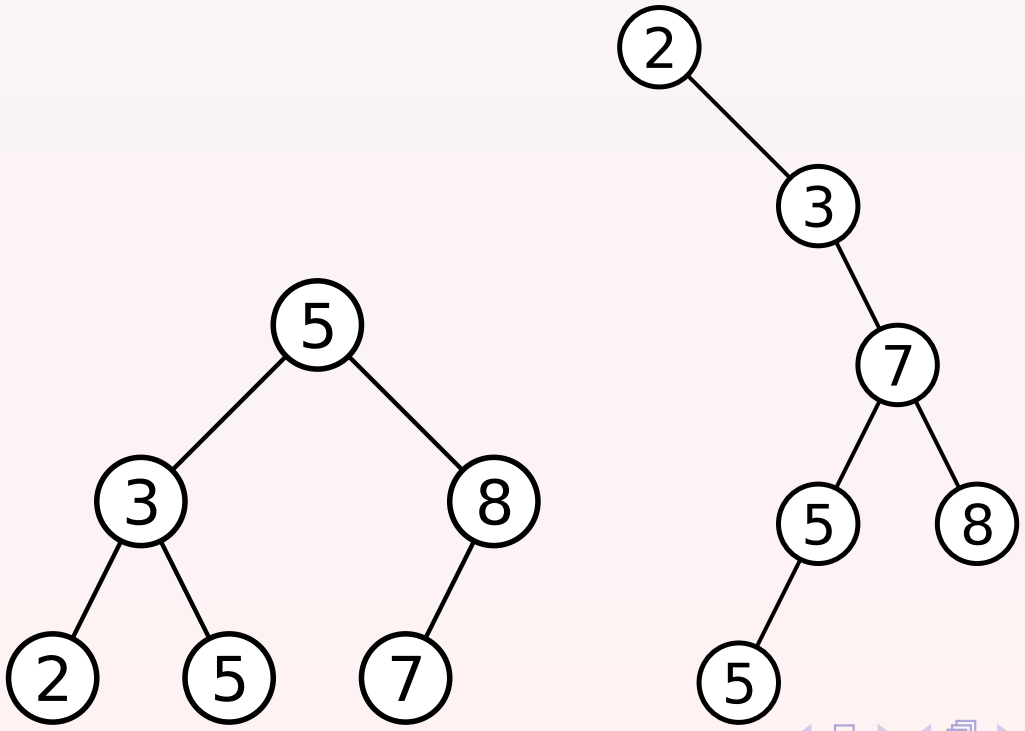
Simple implementation of dynamic set ADT

- The dynamic set ADT can be implemented using linked list.
- Is this implementation good?

Binary Search Tree

A binary search tree (BST) is an implementation of the dynamic set ADT. The elements of the set are stored in the nodes of a binary tree (exactly one element in each node) such that the following property is satisfied for every node x.

- For every node y in the left subtree of x, $y.\mathrm{key} \leq x.\mathrm{key}$.
- For every node z in the right subtree of x, $z.\mathrm{key} > x.\mathrm{key}$.



Binary Search Tree

Each node x in the tree has the following fields.

- key: The key of the element of x.
- Satellite data.
- left: pointer to the left child of x (can be NULL).
- right: pointer to the right child of x (can be NULL)
- p: pointer to the parent of x (NULL for the root).

$T.\mathrm{root}$ is a pointer to the root of tree T.

Tree walk

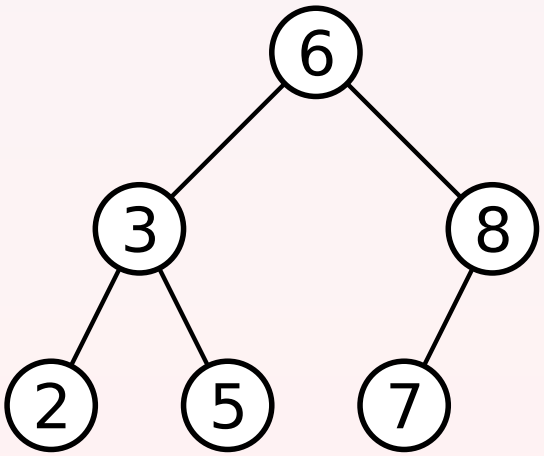
INORDER(x)

(1) **if** $x \neq$ NULL
(2) INORDER(x.left)
(3) **print** x.key
(4) INORDER(x.right)

Time complexity: $\Theta(n)$, where n is the number of nodes.
We also define two additional procedures.

Preorder Print x.key before the two recursive calls.

Postorder Print x.key after the two recursive calls.

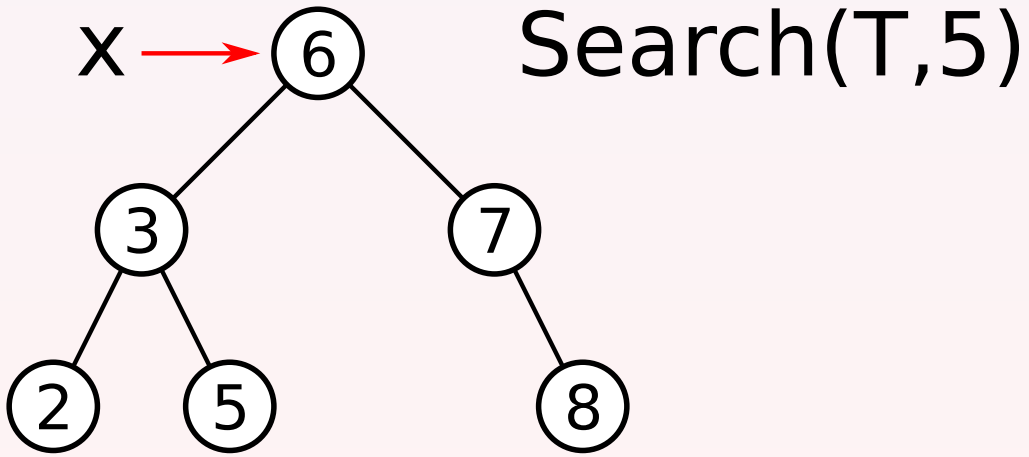


Inorder: 2 3 5 6 7 8
Preorder: 6 3 2 5 8 7
Postorder: 2 5 3 7 8 6

Recursive search

SEARCH(x, k)

(1) **if** $x = \text{NULL}$ OR $k = x.\text{key}$
(2) **return** x
(3) **if** $k < x.\text{key}$
(4) **return** SEARCH$(x.\text{left}, k)$
(5) **else**
(6) **return** SEARCH$(x.\text{right}, k)$

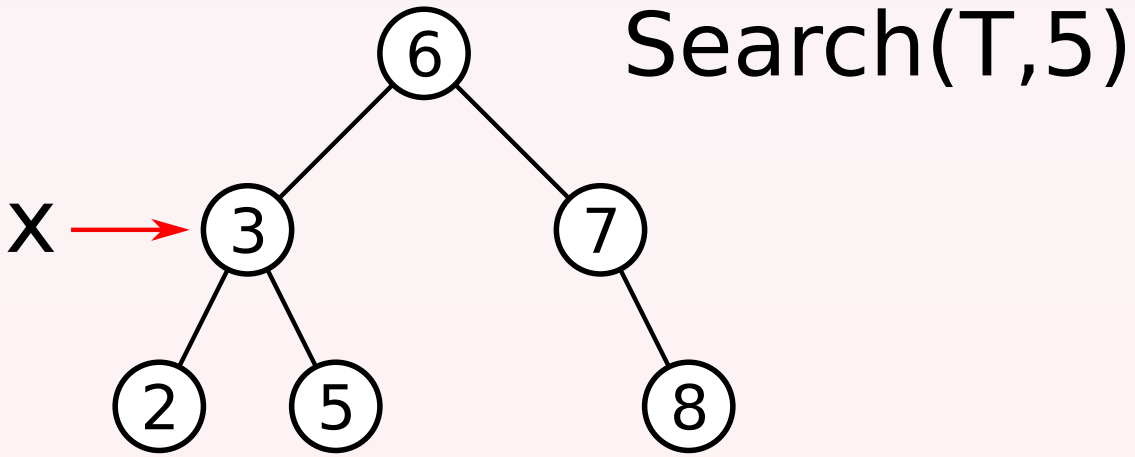


Time complexity: $\Theta(h)$ where h is the height of the tree.

Recursive search

SEARCH(x, k)

(1) **if** $x = \text{NULL}$ OR $k = x.\text{key}$
(2) **return** x
(3) **if** $k < x.\text{key}$
(4) **return** SEARCH$(x.\text{left}, k)$
(5) **else**
(6) **return** SEARCH$(x.\text{right}, k)$

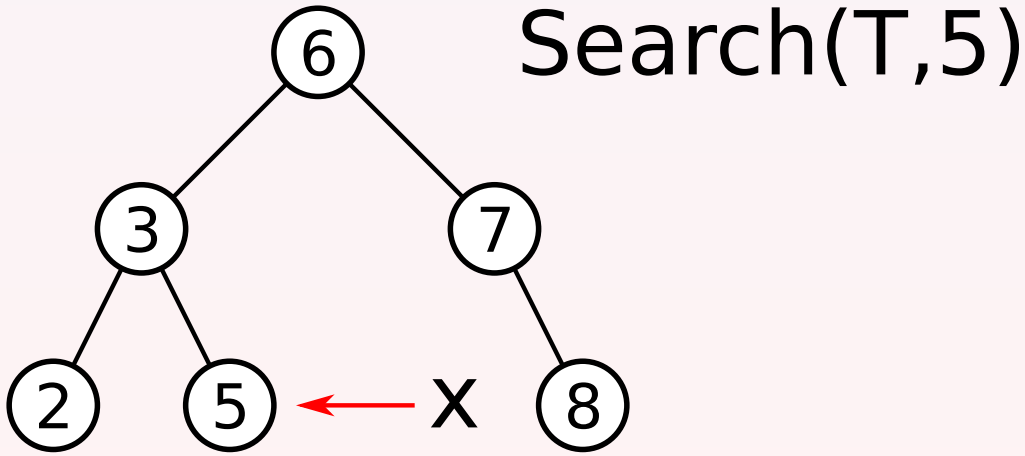


Time complexity: $\Theta(h)$ where h is the height of the tree.

Recursive search

SEARCH(x, k)

(1) **if** $x = \text{NULL}$ OR $k = x.\text{key}$
(2) **return** x
(3) **if** $k < x.\text{key}$
(4) **return** SEARCH$(x.\text{left}, k)$
(5) **else**
(6) **return** SEARCH$(x.\text{right}, k)$



Time complexity: $\Theta(h)$ where h is the height of the tree.

Iterative search

SEARCH(x, k)

```
(1) while x ≠ NULL AND k ≠ x.key
(2)     if k < x.key
(3)         x ← x.left
(4)     else
(5)         x ← x.right
(6) return x
```

Time complexity: $\Theta(h)$.

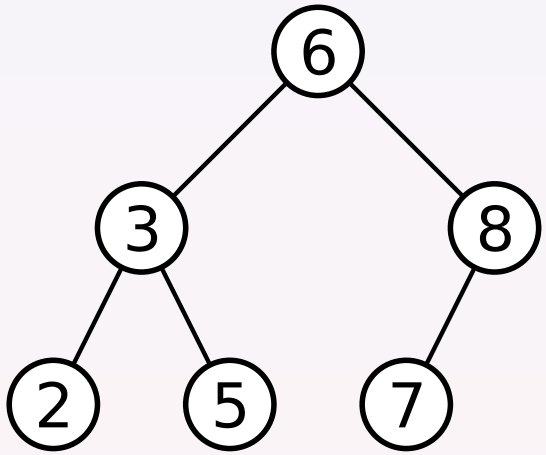
Minimum/Maximum

MINIMUM(x)

(1) **while** x.left $\neq$ NULL
(2) $\quad x \leftarrow x$.left
(3) **return** x

MAXIMUM(x)

(1) **while** x.right $\neq$ NULL
(2) $\quad x \leftarrow x$.right
(3) **return** x



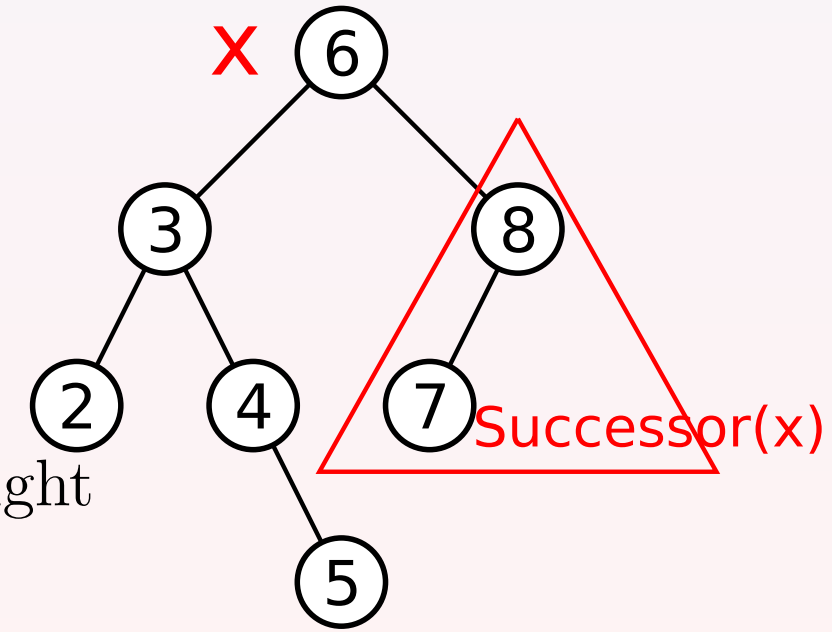
Time complexity: $\Theta(h)$.

Successor

To find the successor of x:

- If x has a right child, the successor is the node with minimum key in the subtree of the right child.
- Otherwise, the successor is the lowest ancestor of x whose left child is also an ancestor of x (if no such ancestor exists, the successor is NULL).

Time complexity: $\Theta(h)$.



SUCCESSOR(x)

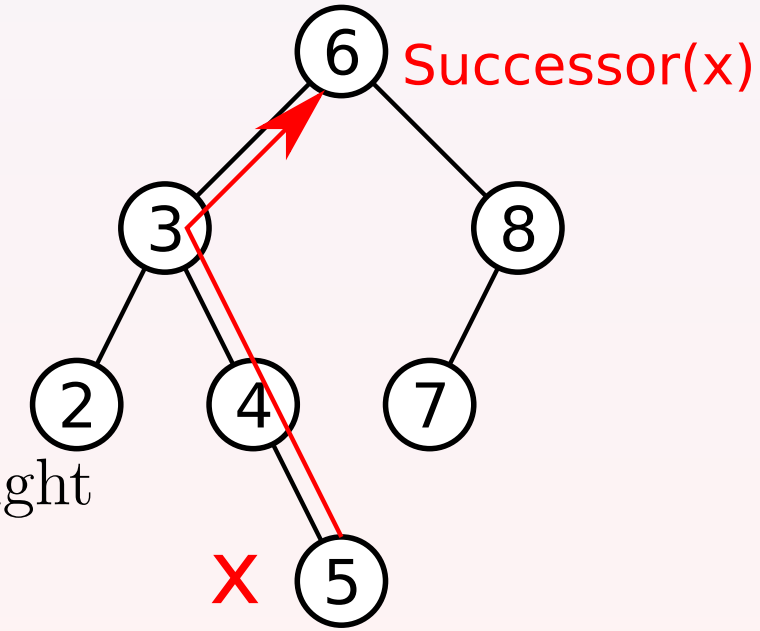
(1) **if** $x.\text{right} \neq \text{NULL}$
(2) **return** MINIMUM($x.\text{right}$)
(3) $y \leftarrow x.p$
(4) **while** $y \neq \text{NULL}$ AND $x = y.\text{right}$
(5) $x \leftarrow y$
(6) $y \leftarrow y.p$
(7) **return** y

Successor

To find the successor of x:

- If x has a right child, the successor is the node with minimum key in the subtree of the right child.
- Otherwise, the successor is the lowest ancestor of x whose left child is also an ancestor of x (if no such ancestor exists, the successor is NULL).

Time complexity: $\Theta(h)$.



SUCCESSOR(x)

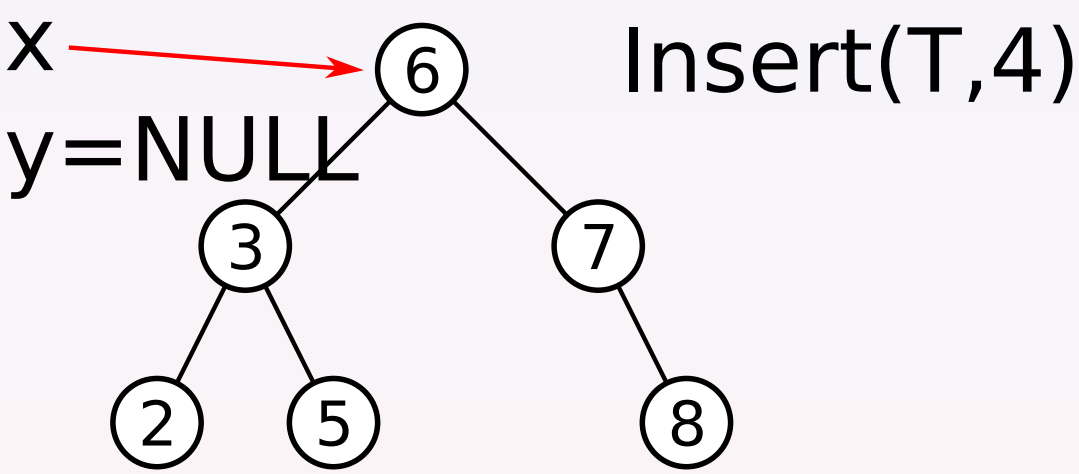
(1) **if** $x.\text{right} \neq \text{NULL}$
(2) **return** MINIMUM($x.\text{right}$)
(3) $y \leftarrow x.p$
(4) **while** $y \neq \text{NULL}$ AND $x = y.\text{right}$
(5) $x \leftarrow y$
(6) $y \leftarrow y.p$
(7) **return** y

Insertion

- When inserting a new element x to the tree, it is inserted as a leaf.
- The location of the new leaf is chosen such that the binary search tree property is maintained.
- This is done by going down from the root using the same algorithm as in the search procedure.
- We need to keep a "trailing pointer", namely a pointer that points to the previous node visited by the algorithm.
- Time complexity: $\Theta(h)$.

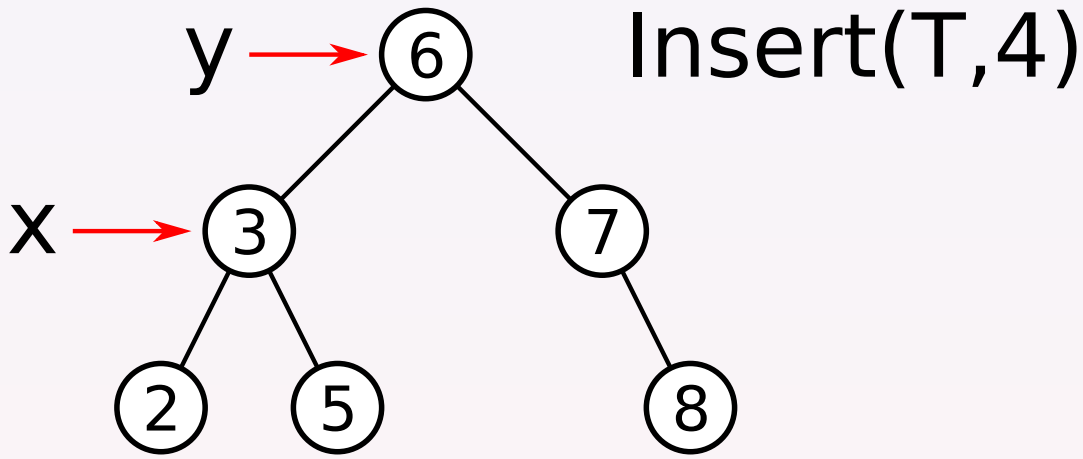
Insertion

```
INSERT(T, z)
y ← NULL
x ← T.root
while x ≠ NULL
    y ← x
    if z.key < x.key
        x ← x.left
    else
        x ← x.right
z.p ← y
if y = NULL
    T.root ← z          // T was empty
else if z.key < y.key
    y.left ← z
else
    y.right ← z
```



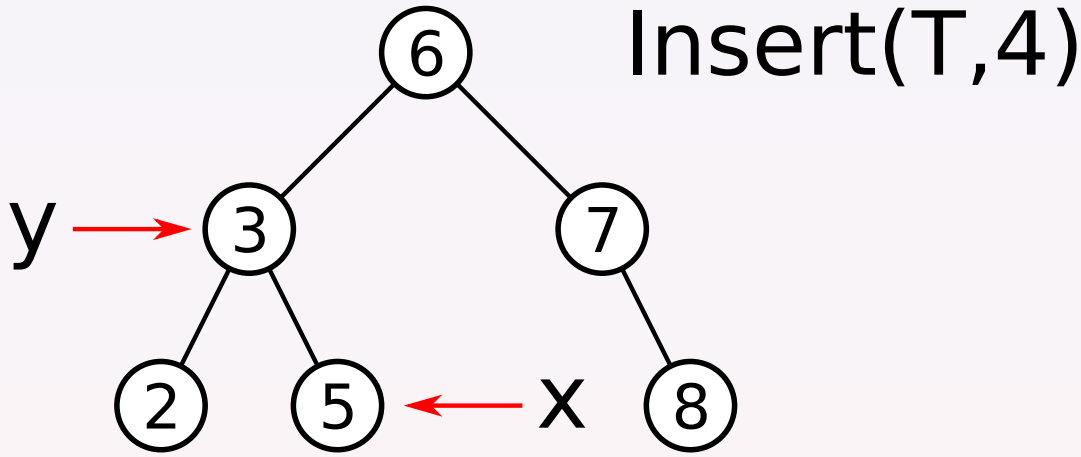
Insertion

```
INSERT(T, z)
y ← NULL
x ← T.root
while x ≠ NULL
    y ← x
    if z.key < x.key
        x ← x.left
    else
        x ← x.right
z.p ← y
if y = NULL
    T.root ← z          // T was empty
else if z.key < y.key
    y.left ← z
else
    y.right ← z
```



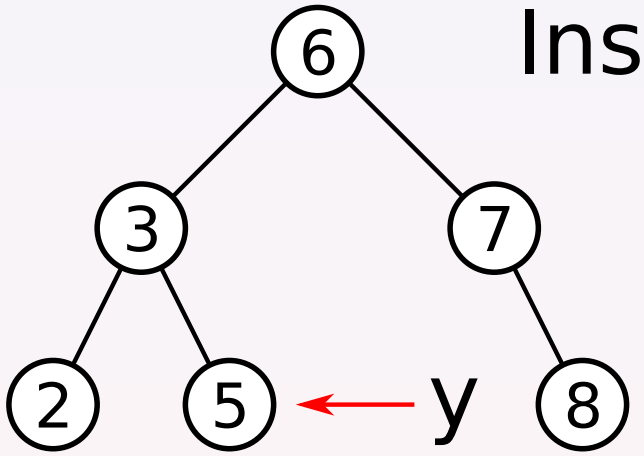
Insertion

```
INSERT(T, z)
y ← NULL
x ← T.root
while x ≠ NULL
    y ← x
    if z.key < x.key
        x ← x.left
    else
        x ← x.right
z.p ← y
if y = NULL
    T.root ← z          // T was empty
else if z.key < y.key
    y.left ← z
else
    y.right ← z
```



Insertion

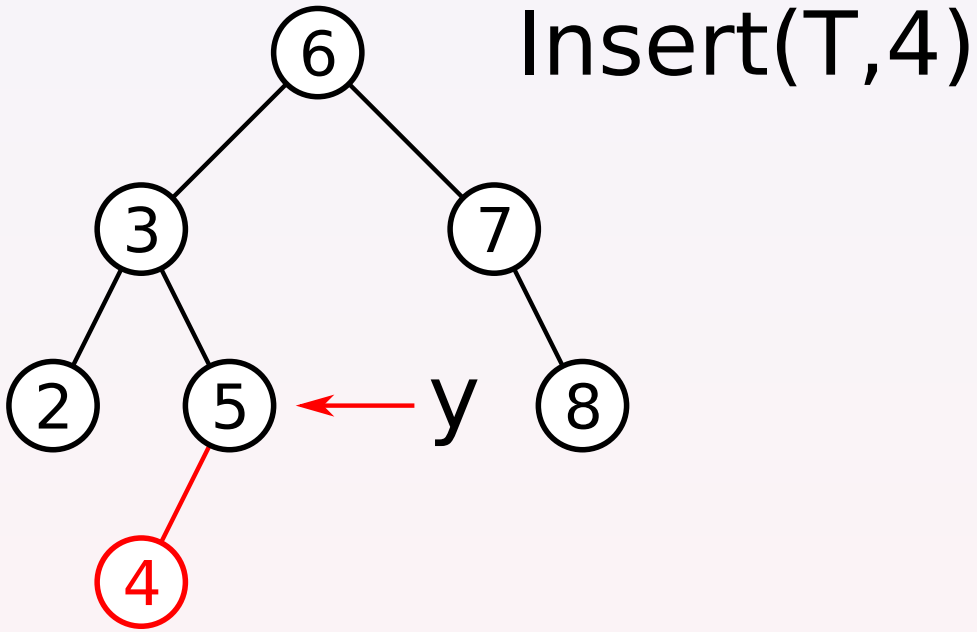
```
INSERT(T, z)
y ← NULL
x ← T.root
while x ≠ NULL
    y ← x
    if z.key < x.key
        x ← x.left
    else
        x ← x.right
z.p ← y
if y = NULL
    T.root ← z          // T was empty
else if z.key < y.key
    y.left ← z
else
    y.right ← z
```



Insert(T,4)

Insertion

```
INSERT(T, z)
y ← NULL
x ← T.root
while x ≠ NULL
    y ← x
    if z.key < x.key
        x ← x.left
    else
        x ← x.right
z.p ← y
if y = NULL
    T.root ← z          // T was empty
else if z.key < y.key
    y.left ← z
else
    y.right ← z
```



Deletion

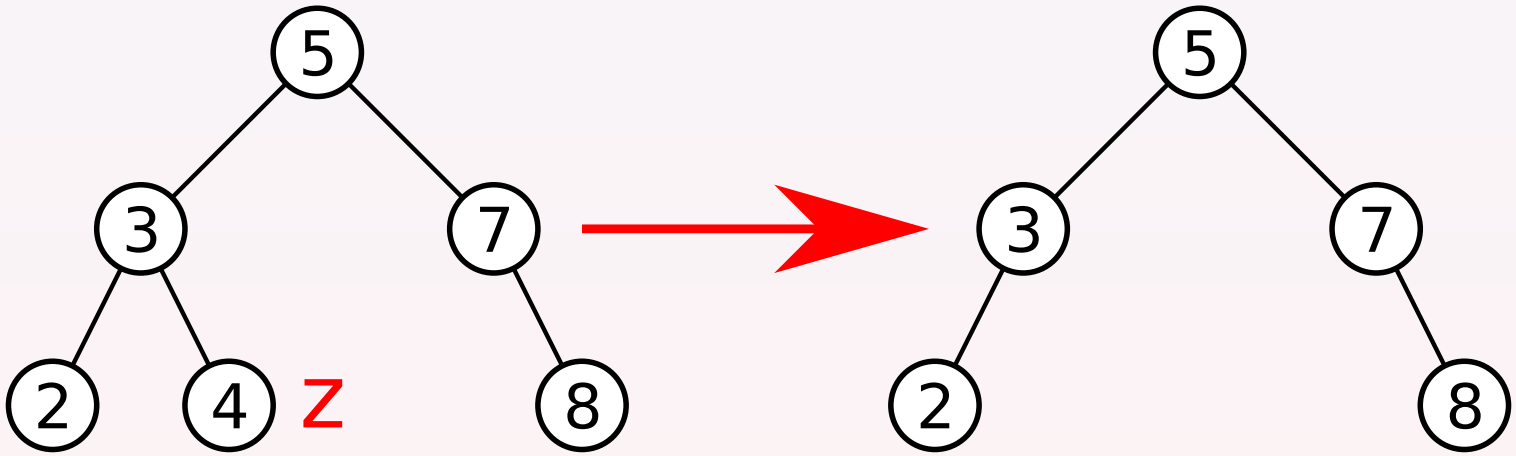
To delete a node z in a binary search tree, there are 3 cases to consider:

- Case 1 z has no children.
- Case 2 z has 1 child.
- Case 3 z has 2 children.

Time complexity: $\Theta(h)$.

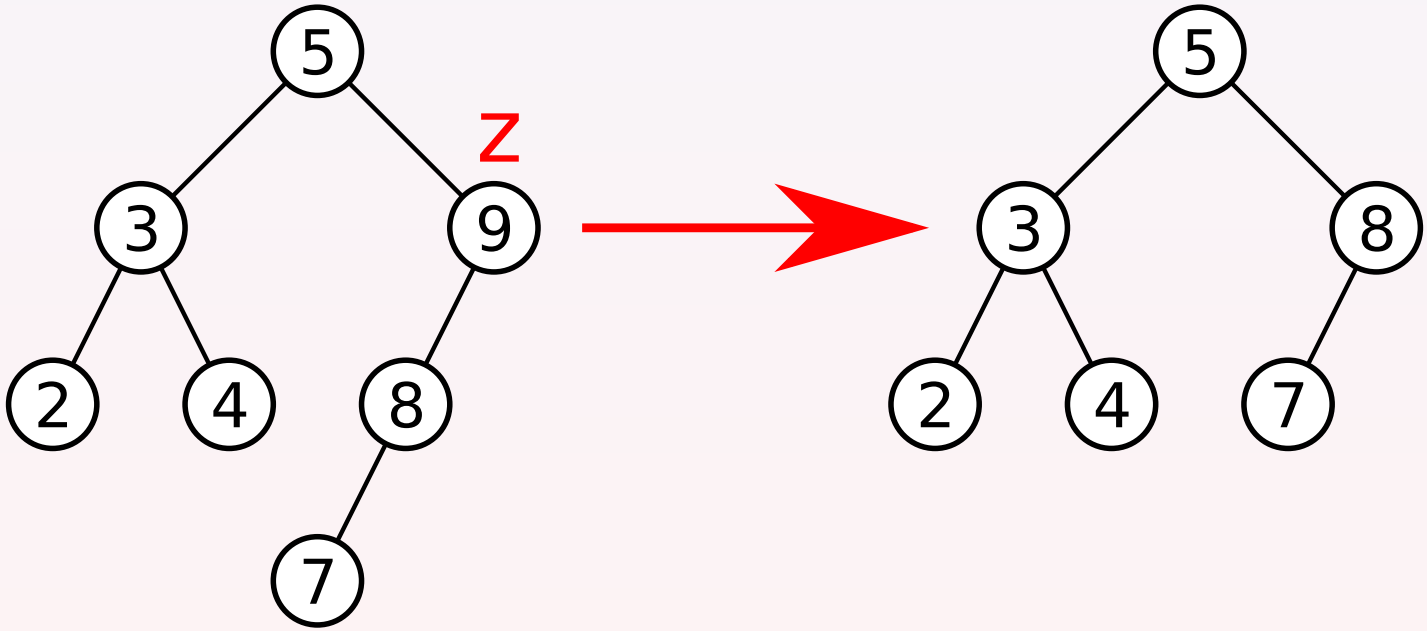
Case 1

If z is a leaf, delete it from the tree.



Case 2

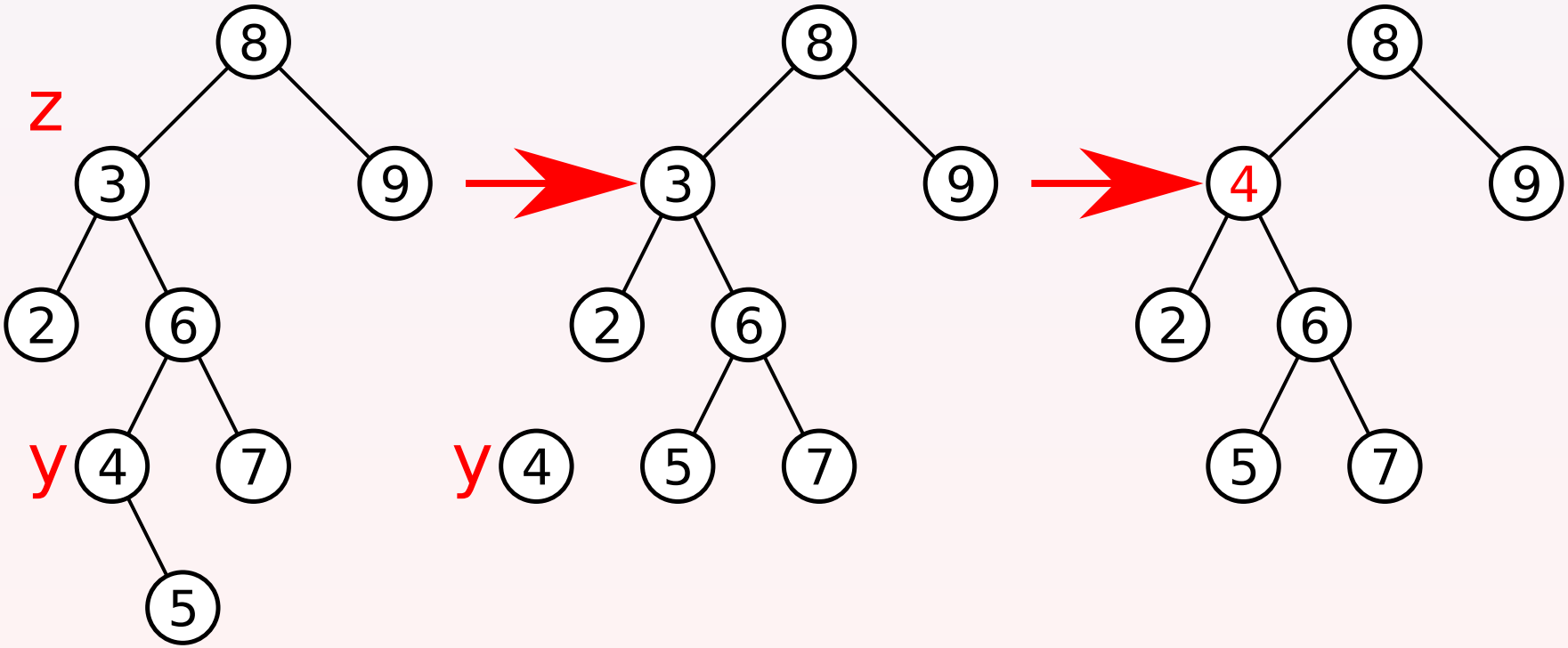
If z has a single child x, delete z from the tree, and make x a child of the parent of z.



Case 3

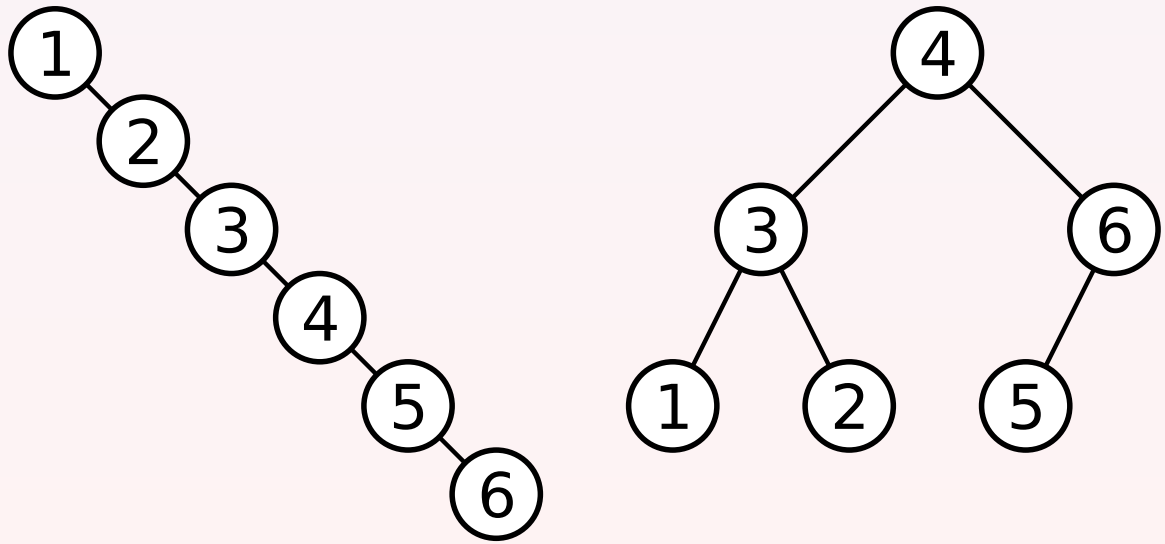
If z has two children:

- $y \leftarrow \textsc{Successor}(z)$
- Delete y from the tree (using Case 1 or 2).
- Make y take the place of z in the tree.



Summary

- All the dynamic set operation take $\Theta(h)$ time (worst case) on a tree of height h.
- In the worst case, $h = n - 1$.
- In the best case, $h = \lfloor \log n \rfloor$ (in an optimal tree, level i of the tree contains exactly 2^i nodes, except perhaps the last level).



Randomly built binary search trees

A randomly built binary search tree is a tree that is built by inserting the elements in random order into an initially empty tree (each of $n!$ permutations of the elements is equally likely).

Theorem

The expected height of a randomly built binary search tree is $\Theta(\log n)$.

Little is known about the average height of a binary search tree when both insertion and deletion are used to create it.