# Binary Search Trees

Search trees are data structures that support many dynamic set operations including:

- SEARCH
- MINIMUM
- MAXIMUM
- PREDECESSOR
- SUCCESSOR
- INSERT
- DELETE

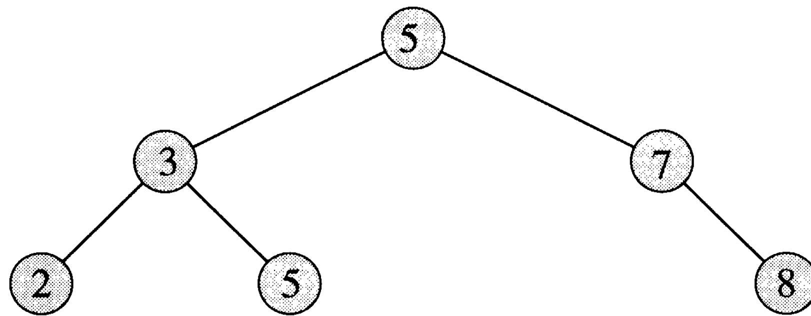
# Binary Search Trees

The keys in a binary search tree are always stored in such a way as to satisfy the **binary search tree property:**

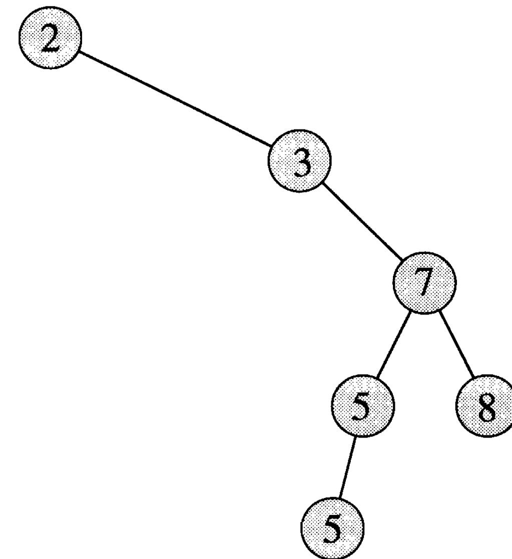
Let $x$ be an internal node in a binary search tree:

- If $y$ is a node in the left subtree of $x$, then $key[y] \leq key[x]$.
- If $y$ is a node in the right subtree of $x$, then $key[x] < key[y]$.

# Binary Search Trees




(a)




(b)

# Traversing binary search trees

A binary search tree can be traversed in different ways:

- **In-order tree walk:** visit the left subtree; then the root; then the right subtree.
- **Pre-order tree walk:** visit the root; then the left subtree; then the right subtree.
- **Post-order tree walk:** visit the left subtree; then the right subtree; then the root.

# Traversing Binary Search Trees

## Inorder tree walk

INORDER_TREE_WALK($x$)

| | | |
|---|---|---|
| 1 | if | $x \neq NIL$ |
| 2 | then | INORDER_TREE_WALK(left[x]) |
| 3 | | print $key[x]$ |
| 4 | | INORDER_TREE_WALK(right[x]) |

# Traversing Binary Search Trees

## Preorder tree walk

PREORDER_TREE_WALK($x$)

| | | |
|---|---|---|
| 1 | if | $x \neq NIL$ |
| 2 | then | print $key[x]$ |
| 3 | | PREORDER_TREE_WALK(left[x]) |
| 4 | | PREORDER_TREE_WALK(right[x]) |

# Traversing Binary Search Trees

## Postorder tree walk

POSTORDER_TREE_WALK($x$)

| | | |
|---|---|---|
| 1 | if | $x \neq NIL$ |
| 2 | then | POSTORDER_TREE_WALK(left[x]) |
| 3 | | POSTORDER_TREE_WALK(right[x]) |
| 4 | | print $key[x]$ |

# Querying a binary search tree

## Searching

TREE_SEARCH($x, k$)

| | | |
|---|---|---|
| 1 | if | $x = NIL$ or $k = key[x]$ |
| 2 | | then return $x$ |
| 3 | if | $k \leq key[x]$ |
| 4 | | then TREE_SEARCH($left[x], k$) |
| 5 | | else TREE_SEARCH($right[x], k$) |

# Querying a binary search tree

## Minimum and Maximum

TREE_MINIMUM($x$)

1 while $left[x] \neq NIL$

2 do $x \leftarrow left[x]$

3 return $x$

TREE_MAXIMUM($x$)

1 while $right[x] \neq NIL$

2 do $x \leftarrow right[x]$

3 return $x$

# Querying a binary search tree

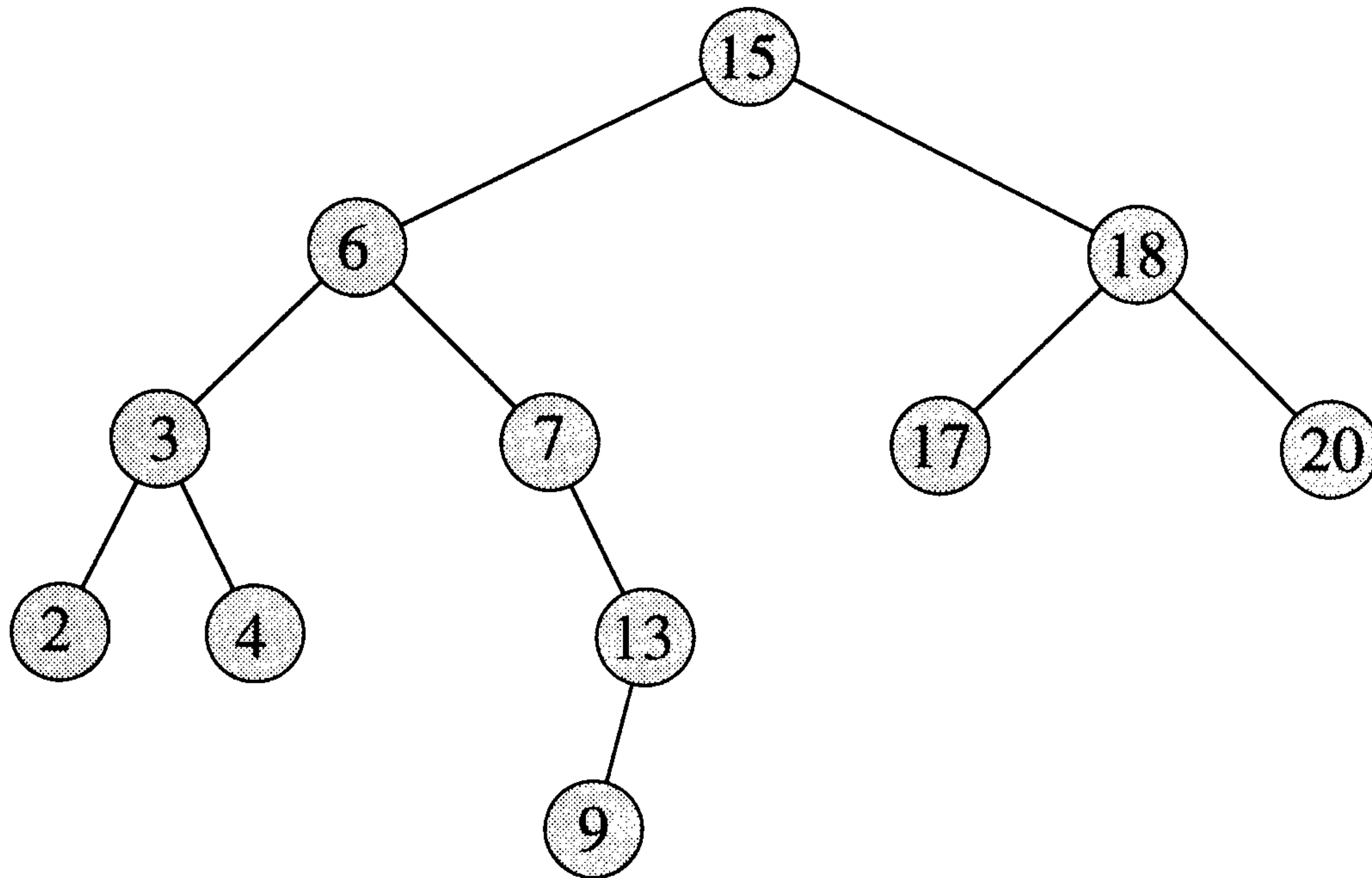
## Successor and Predecessor

If all keys are distinct, the successor of a node $x$ is the node with the smallest key greater than $key[x]$.

TREE_SUCCESSOR($x$)

| | | | |
|---|---|---|---|
| 1 | if | $right[x] \neq NIL$ | |
| 2 | | then | return TREE_MINIMUM(right[$x$]) |
| 3 | $y \leftarrow p[x]$ | | |
| 4 | while | $y \neq NIL$ and $x = right[y]$ | |
| 5 | | $x \leftarrow y$ | |
| 6 | | $y \leftarrow p[y]$ | |
| 7 | return $y$. | | |

# Binary Search Trees



# Insertion and Deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.

The data structure must be modified to reflect this change, but in such a way that the binary search tree property continues to hold.

# Insertion

TREE_INSERT($T, z$)

```
1    y ← NIL
2    x ← root[T]
3    while  x ≠ NIL
4           do    y ← x
5                 if    key[z] ≤ key[x]
6                       then x ← left[x]
7                       else x ← right[x]
8    p[z] ← y
9    if     y = NIL
10          then  root[T] ← z
11          else  if    key[z] ≤ key[y]
12                           left[y] ← z
13                           right[y] ← z
```

See Fig 12.3 (page 262)

# Deletion

```
TREE_DELETE(T, z)
  if left[z] = NIL or right[z] = NIL then y ← z
                                      else y ← TREE_SUCCESSOR(z)

  if left[y] ≠ NIL                    then x ← left[y]
                                      else x ← right[y]

  if x ≠ NIL                          then p[x] ← p[y]

  if p[y] = NIL                       then root[T] ← x
                                      else if y = left[p[y]] then left[p[y]] ← x
                                                             else right[p[y]] ← x

  if y ≠ z                            then key[z] ← key[y]
  return y
```

# Binary Search Trees