Dynamic Set

- Sets that can change over time
- Element
  - Key – a value identifying the element
  - Satellite data – additional data belonging to the element
  - Element keys may or may not be unique (set vs. multiset)
  - In our book, elements are objects and the dynamic sets store pointers to elements

- Operations
  - Queries - return information about the set
  - Modifying operations - change the set

- Typical operations
  - Search(S, k) - A query that, given a set S and a key k, returns a pointer x to an element in S such that $x.key == k$ or NIL if no such element is in S

- Insert(S, x) - a modifying operation that augments the set S with the element pointed to by x
- Delete(S, x) - delete x from S, x is a pointer not a key
- minimum(S) - returns the element with the smallest key
- maximum(S)
- Successor(S, x) - given an element pointer x, returns a pointer to the next larger element in S, or NIL if x is the maximum

- predecessor(S, x) - like successor, but returns the next smaller

- The time complexity of an operation is usually in terms of the size of the set

- Stacks

    - Operations

        push(S, x)

        pop(S)

Binary Tree

- Tree where each node has at most 2 children
- Can be represented by a linked structure
    - Elements (or nodes) have these attributes:

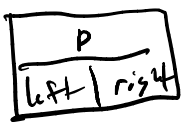
        p - pointer to the parent, or NIL for the root

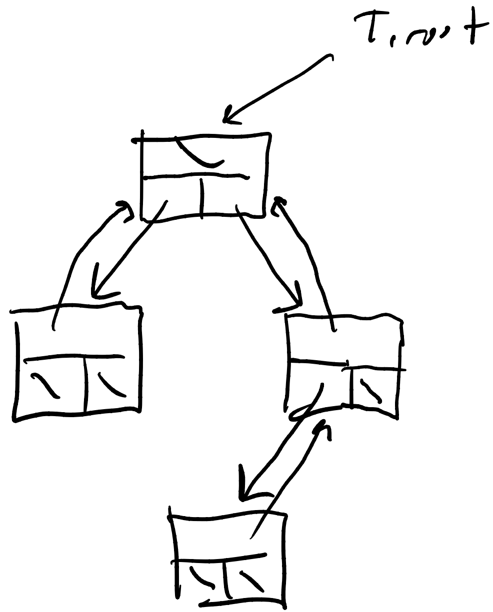
        left - pointer to left child, or NIL

        right - pointer to right child, or NIL

        key - the key value

- Tree T has attribute T.root, a pointer to the root node, or NIL for an empty tree





- left and right subtrees of x - subtrees rooted at x's children

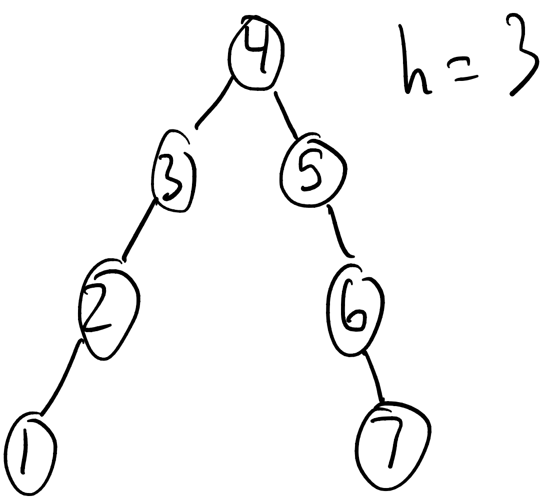
Binary Search Tree

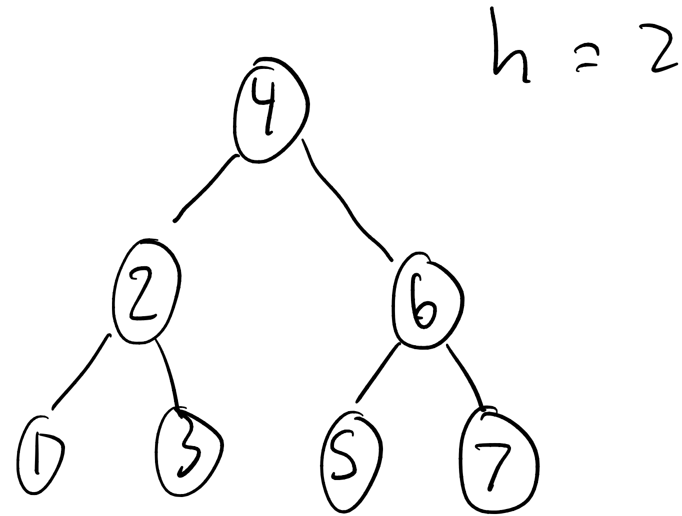
- Supports search, insert, minimum, maximum, successor, predecessor, and delete

- Binary search tree property
  - Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$

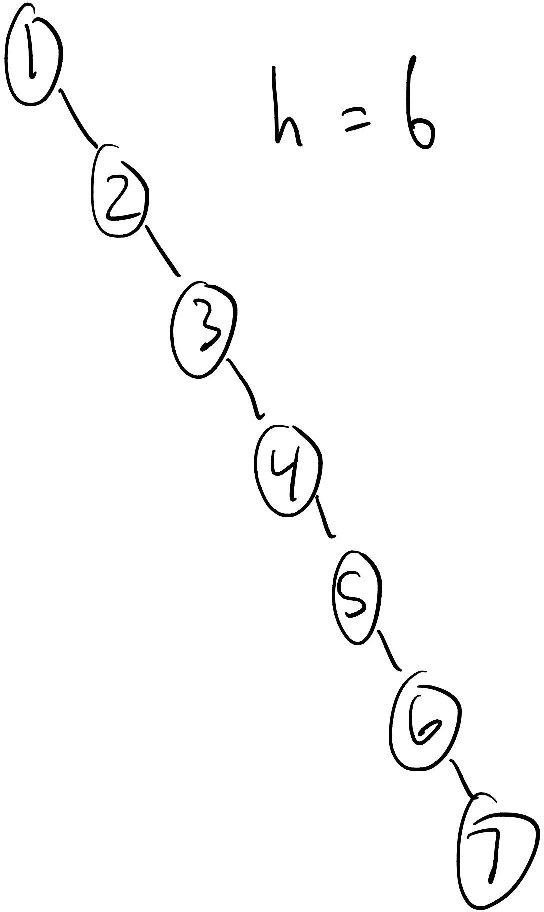
    If y is a node in the right subtree of x, $y.key \geq x.key$

Represent $\{1, 2, 3, 4, 5, 6, 7\}$





Minimum is always leftmost element, maximum is rightmost



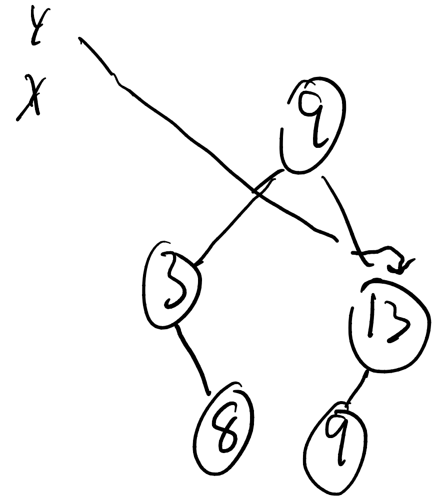
$h = 6$

In general, height is $O(n)$

For a balanced tree, height is $O(\lg n)$

- Modifying operations
    - Insert a new node z
        - Adds a new leaf
        - Follow a path until a NIL pointer that can be replaced by z

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```



insert 8

insert 9

$O(h)$

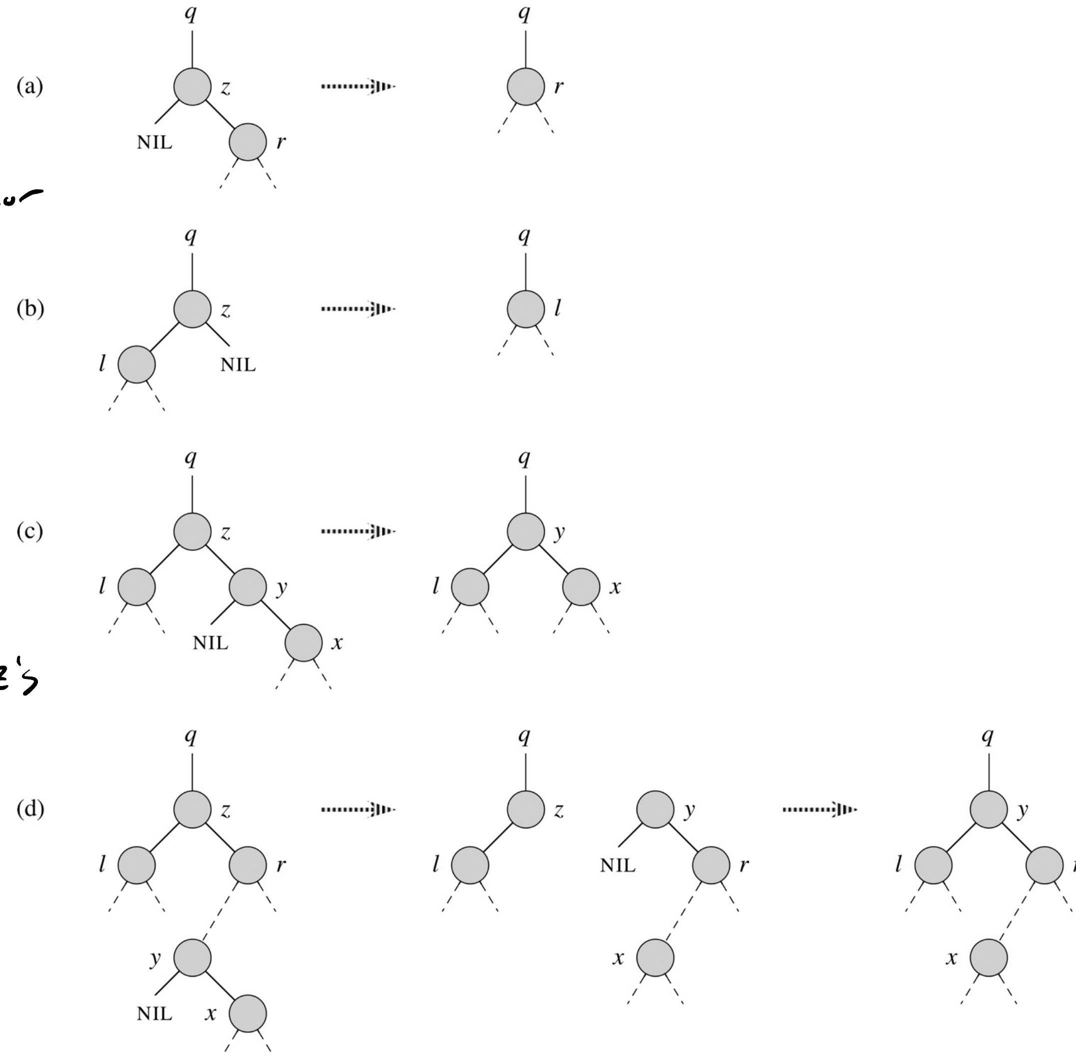
Best case $\Theta(\lg n)$ for a balanced tree

Worst case $\Theta(n)$

- Delete a node z
  - Cases
    - If z has no children, modify its parent to replace z with NIL as its child
    - If z has one child, elevate that child to take z's position

- if z has 2 children find z's successor y and have y take z's position in the tree.

The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree



- Helper function transplant(T, u, v)
  - Replaces the subtree rooted at u with the subtree rooted at v
  - Node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child

```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
        return x
  if k < x.key
        return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

TREE-MINIMUM(x)

while $x.left \neq$ NIL

$\quad x = x.left$

return x

TREE-MAXIMUM(x)

while $x.right \neq$ NIL

$\quad x = x.right$

return x

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```