Binary Search Tree (BST)

Cormen Sects 12.1, 12.2, 12.3

Predecessor Problem

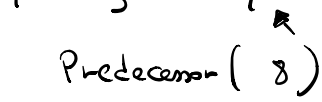
Given a set $S \subseteq U$ of n keys and we would like to build a data structure supporting the following operations:

Dictionary Problems:

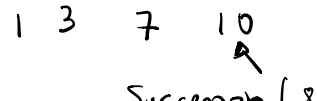
- Insert (S, x) adds x to S
- Delete (S, x) removes x from S
- Search (S, x) returns True if $x \in S$, False otherwise

Also:

- Min (S) / Max (S) returns min/max in S
- Predecessor (S, x) returns the predecessor of x in S i.e. the largest key in S which is smaller than x (no need for x to be in S)



- Successor (S, x) returns the successor of x in S, i.e. the smallest key in S which is larger than x



Solution for this problem are often called ordered dictionary / map

Binary Search solves the static version in $O(\log n)$ time

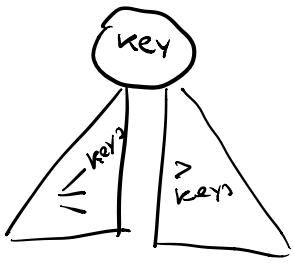
(B alanced) Binary Search tree ((B)BST) solves all operations in $\Theta(\log n)$ time

Keys of S are stored in a binary tree T satisfying the binary-search-tree property

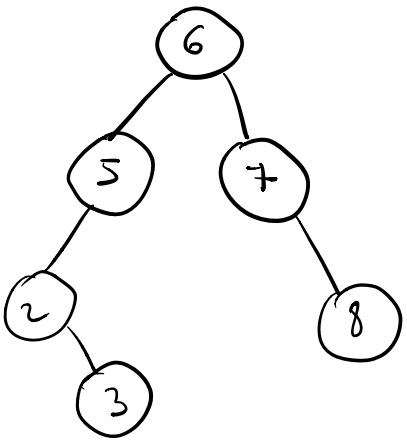
Let x be a node in BST.

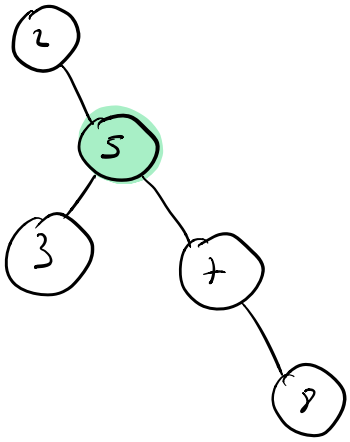
if y is a node in left subtree of x,
then $y.key \leq x.key$

if y is a node in right subtree of x,
then $x.key < y.key$

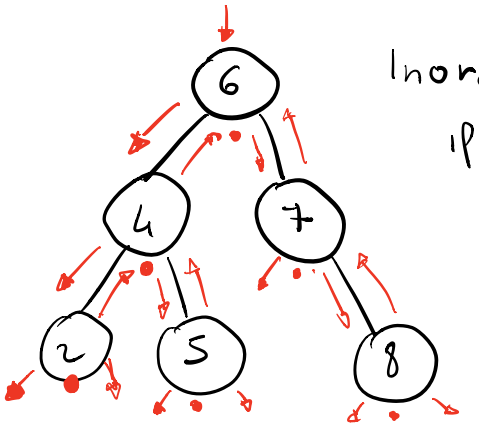


Several different BSTs satisfying the property





Important property: if we print keys by traversing the tree inorder, then we get S sorted




```
Inorder-Visit(x)
  if x ≠ NIL
    Inorder-visit(x.left)
    Print x.key
    Inorder-visit(x.right)
```

preorder visit

postorder visit

2 4 5 6 7 8

Complexity $\Theta(n)$ time

x.key key stored in x

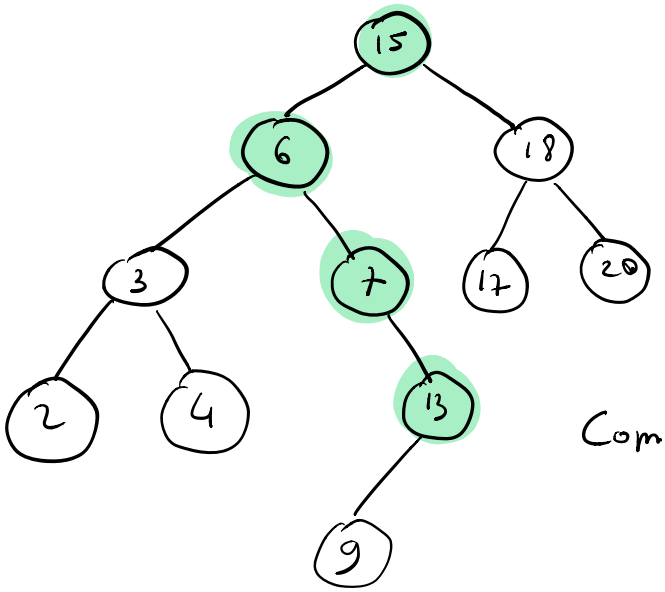
x.left

x.right

are left and right children

NIL if the child does not exist

Searching



Search (13)

Complexity $\Theta(h)$ time

```
Search (x, k)
1  if x == NIL  or  k == x.key
2        return x
3  if k < x.key
4        return Search (x.left, k)
5  else
6        return Search (x.right, k)
```

Minimum and Maximum

```
Min(x)
1 while x.left ≠ NIL
2       x = x.left
3 return x
```

```
Max(x)
1 while x.right ≠ NIL
2       x = x.right
3 return x
```

Complexity $\Theta(h)$ time

Min(T.root) Max(T.root)

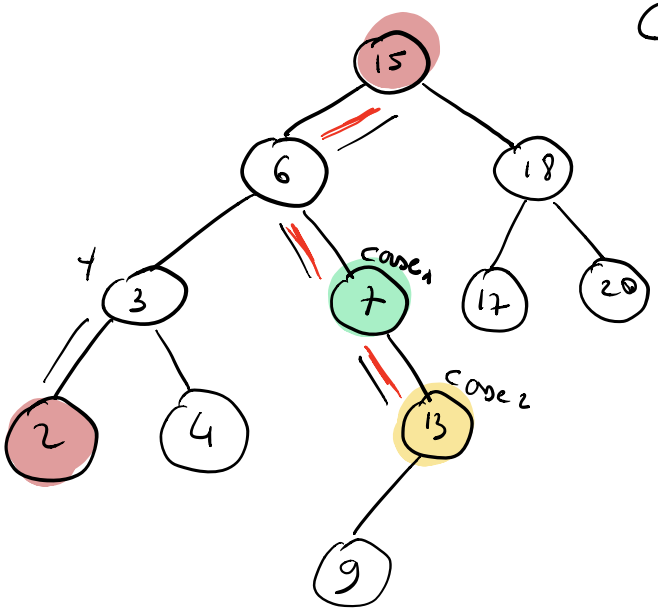
Successor and Predecessor

Report successor of a node x

there are two possible cases

- if x has the right subtree, then
  successor of x is minimum in right subtree
- if x's right subtree is empty and x has a successor y,
  y is the lowest ancestor of x whose
  left child is also ancestor of x (or x itself)

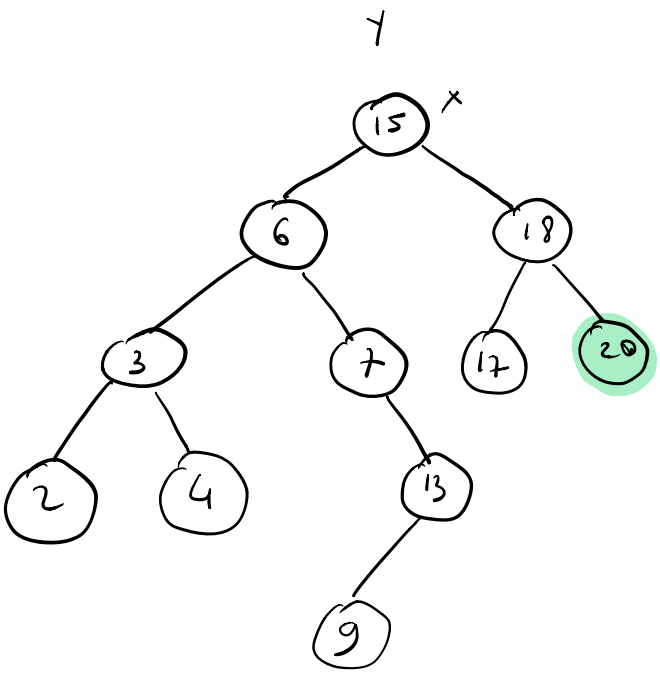
Complexity $\Theta(h)$ time



Successor (x)

$x.p$ points to the parent of x

```
1 if x.right ≠ NIL
2       return min(x.right)
3 y = x.p
4 while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7 return y
```

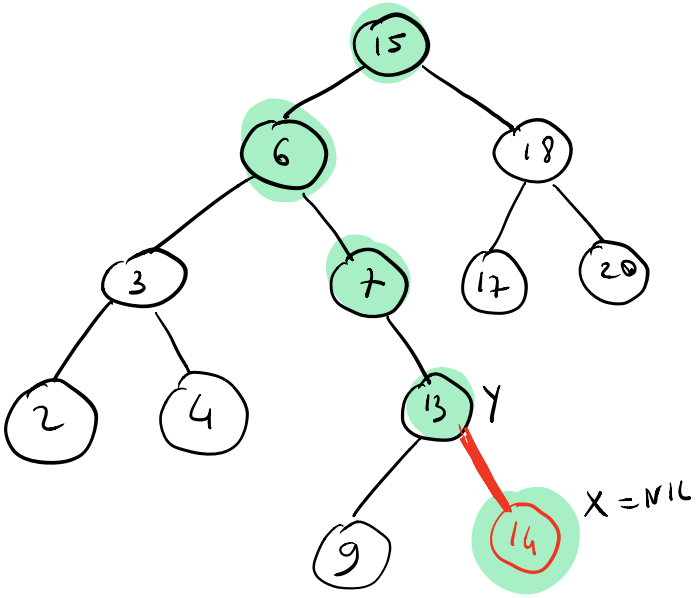



```
Successor (x)
1 if x.right ≠ NIL
2       return min(x.right)
3 y = x.p
4 while y ≠ NIL and x == y.right
5         x = y
6         y = y.p
7 return y
```

$x == y.right$

Insertion

Insert (14)



Complexity $\theta(h)$ time

z.key = v
z.left = NIL
z.right = NIL

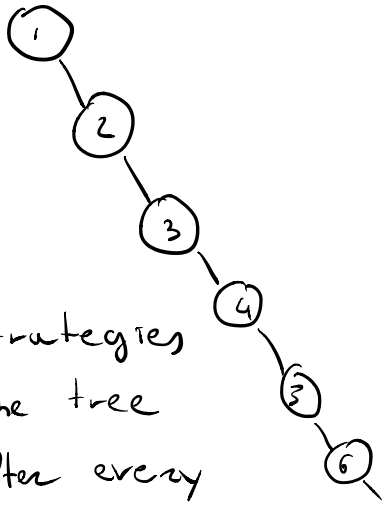
```
Insert (T, z)        // z a new node
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4         y = x
5         if z.key < x.key
6               x = x.left
7         else
8               x = x.right
9   z.p = y
10  if y == NIL      // tree was empty
11        T.root = z
12  elseif z.key ≤ y.key
```

```
13          y.left = z
14      else
15          y.right = z
```

Tree could become very unbalanced

What's a worst case insertion sequence?

1, 2, 3, 4, 5, 6, ...

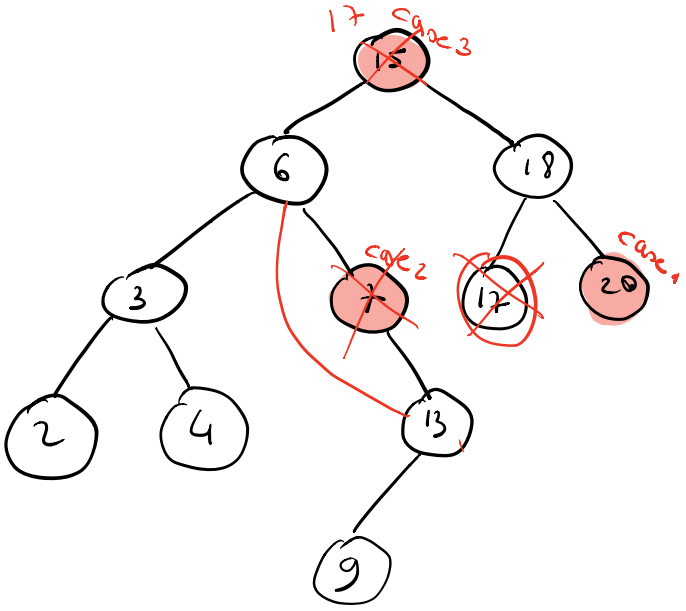


$h = \Theta(n)$

There are strategies to keep the tree balanced after every insert so that $h = \Theta(\log n)$

e.g. AVL-trees, a-b trees, Red Black trees

Deletion



There are 3 different cases:

- x is a leaf, just remove it
- x has just one child, just remove x and connect x's parent with x's child
- x has both children. replace x with its successor (or predecessor) y
  y is the minimum in x's right subtree
  so, y is easy to remove because it has either 0 or 1 child.

Complexity is $\Theta(h)$ time