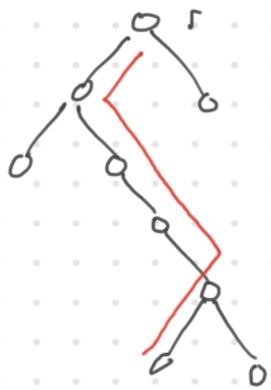
Binary search trees

Dynamic data structure

Def binary tree is a rooted tree where root has deg ≤ 2 & every other vertex has deg ≤ 3

height $= 5$



Def height max distance of a leaf from the root.

Binary search tree information is stored as keys assigned to individual nodes

Information for a node x:

x. key   x. parent   x. left , x. right

value stored at that node   parent in the tree   left & right children in the tree

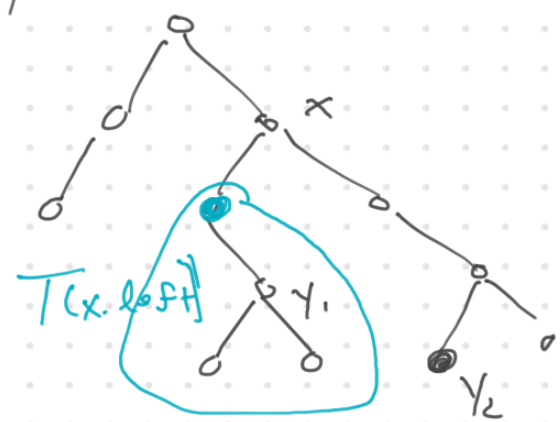
Note if x has no left child x. left = NIL

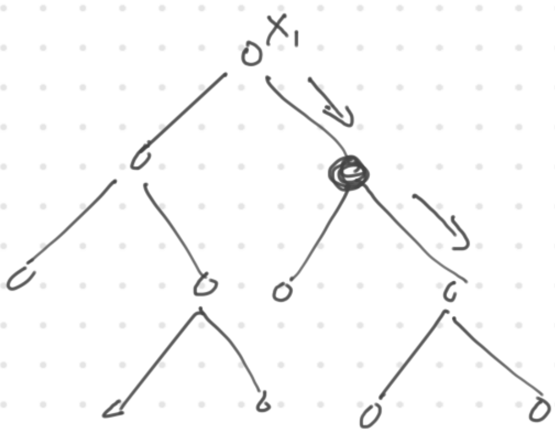
Def binary search tree property:

x is a node of a BST & y_1 is a node of the subtree of T with root $x.left \Rightarrow y_1.key < x.key$

if y_2 is a note of subtree of T w/ root $x.right \Rightarrow y_2.key > x.key$

Notation: $T(x.left)$ = subtree of T w/ root $x.left$





This recursive structure on the keys gives an easy algorithm to order the keys

```
in_order_treewalk (x)
    if x ≠ NIL
        in_order_treewalk (x.left)
        add x,
        in_order_treewalk (x.right)
```

Complexity of in-order_treewalk $O(n)$

Thm in-order_treewalk takes $\Theta(n)$ runtime

pf let $S(n)$ denote the amount of time we need run alg on a tree w/ n nodes

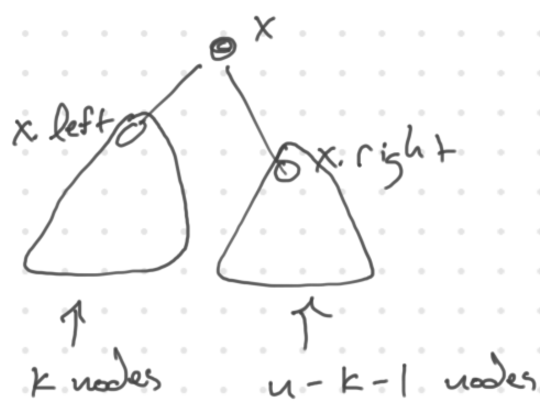
let c be the constant # of steps to evaluate "if $x \neq$ NIL" when instead the tree is empty

let d be the constant # of steps needed to invoke the recursive calls (not counting the actual work done in those calls, though)

+ the constant work to add x

Cl The alg needs $(c+d)n + c$ operations on a tree w/ n nodes.

pf induction on n base case $n=0$ ✓



~~by induction,~~ ~~The~~

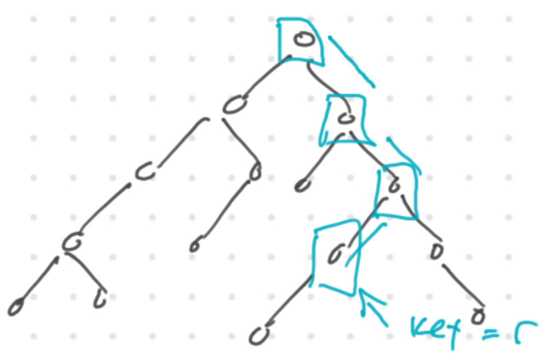
$$S(n) = S(k) + S(n-k-1) + d$$

by induction, this ⇒

$$S(n) \leq (c+d)k + c + (c+d)(n-k-1) + c + d$$
$$= (c+d)n - (c+d) + 2c + d$$
$$= (c+d)n + c \quad \blacksquare$$

+ claim immediately implies the Theorem.

given a value r, we want to test whether r is the key of some node.




```
Tree_search (x, r)
    if x = NIL or x.key = r
        return x
    if r < x.key
        Tree_search (x.left, r)
    else Tree_search (x.right, r)
```

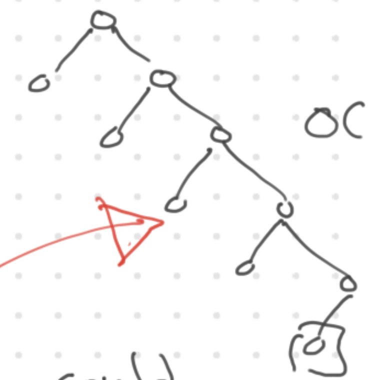
root of a BST

key value we're searching for

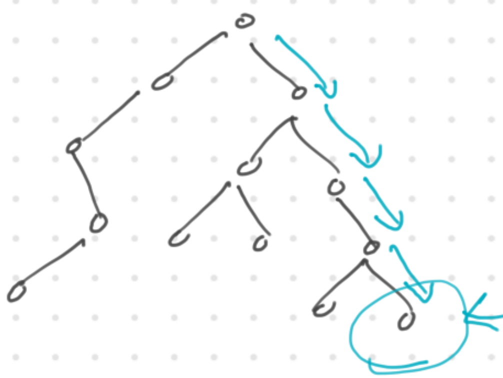
written w/ while loops

```
while x ≠ NIL + r ≠ x.key
      if  r < x.key
            x = x.left
      else  x = x.right
return x
```

Complexity here?

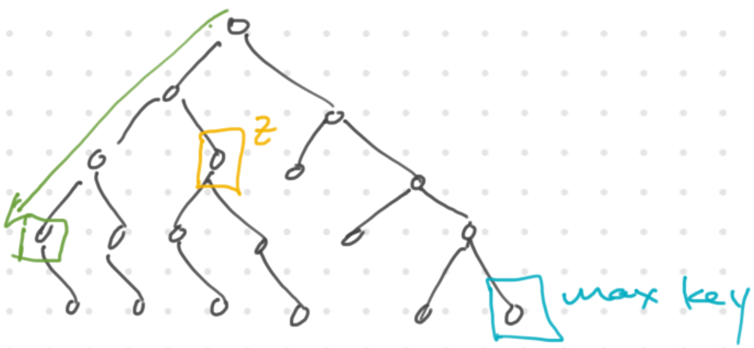


because The tree isn't balanced, worse case could be $O(n)$, In general, it is O(height of The tree)



BST

max key value — walking from The root, The right child always has larger key value, keep going until This terminates in The bottom



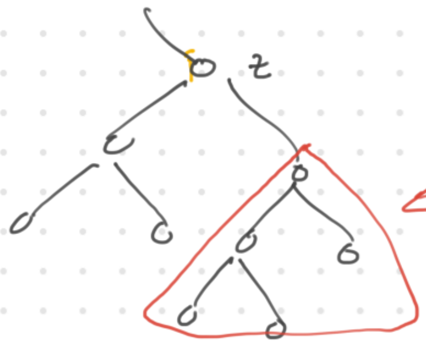
in the bottom right corner

instead, the min key - more accurately, walking from root, first node w/ .left = NIL

Tree_min (x)
  while x.left ≠ NIL
    x = x.left
  return x (x.key if we want value)

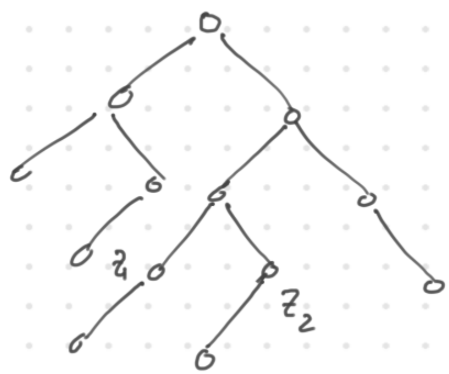
Tree_max (x)
  while x.right ≠ NIL
    x = x.right
  return x

Problem given a node z of the tree, find the successor to z in the tree_order (without calculating out the full order)



if $z.right \neq$ NIL
we take min in this subtree Tree_min (z. right)

but what happens when z. right = NIL

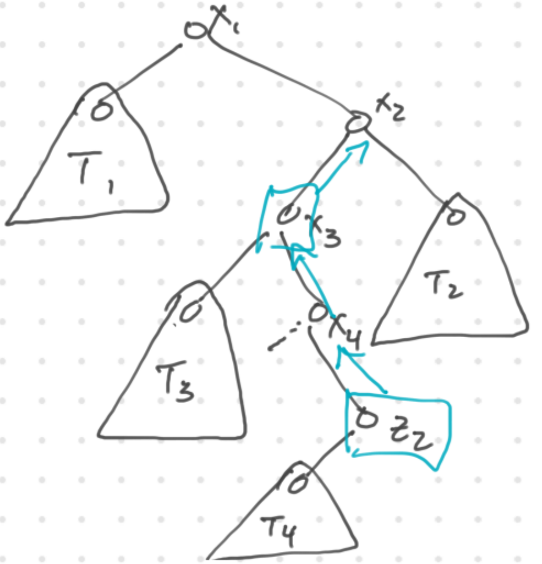


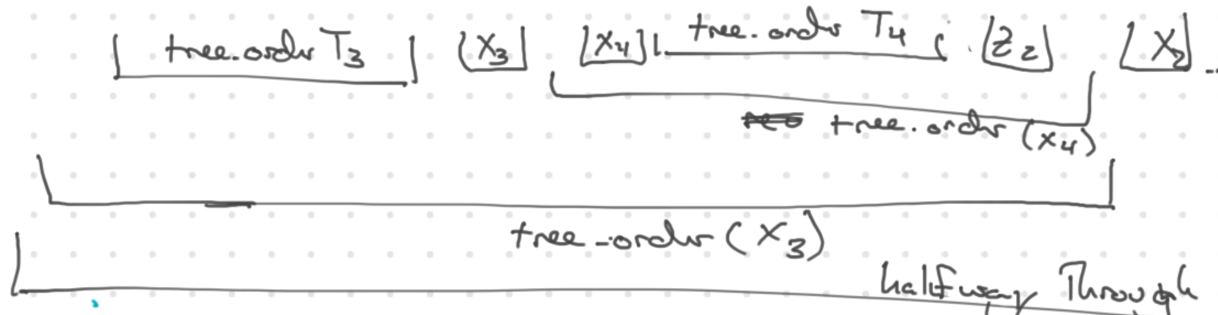
z. right = NIL still divides further into two subcases

Case 2a z_1. right = NIL + z_1 = z_1. parent. left

→ z_1. parent is the successor

Case 2b z_2. right = NIL + z_2 = z_2. parent. right





tree-order (x_2)

successor of z_2 = x_2

```
Tree-successor (z)
    if z.right ≠ NIL
        return Tree.min (z.right)
    y = z.parent
    while y ≠ NIL & z = y.right
        z = y
        y = y.parent
    return y
```

How do we reach x_2 from z_2

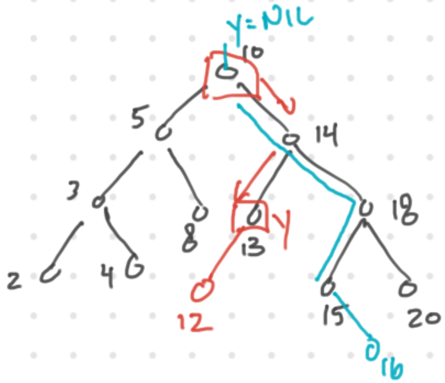
walk back in the tree until first time we get to a node which is the left child of it's parent.

Tree-predecessor is the same

runtime = $O(\text{height}(T))$

Insertion & deletion

of the two, insertion is easy



insert 12

Starting at the root, walk down finding which subtree should contain the new node until we find an empty child to insert the value.

Tree-insert (root, z)

```
y = NIL
x = root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
~~if y = nil~~
```

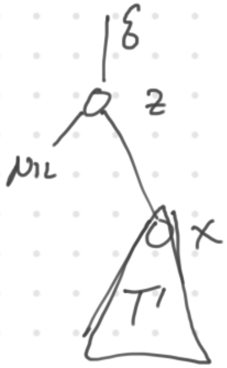
if y = NIL ⇒ tree empty & z is the root of the tree

```
else if z.key < y.key
    y.left = z
else y.right = z
```

Deletion

we are trying to delete a node z from the tree

Cases

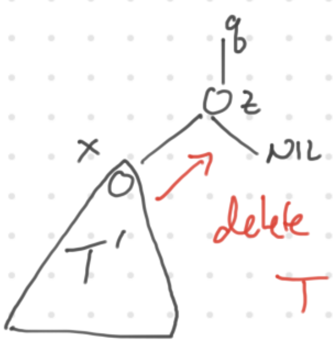


No matter whether z is left or right child of q

→

delete z + move T' up.

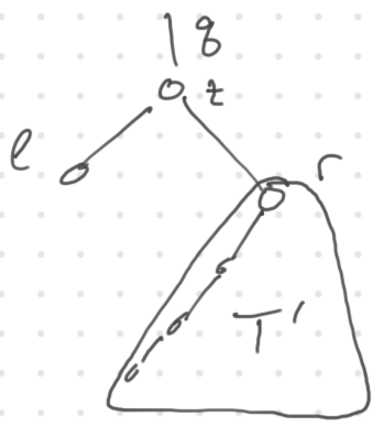
but the case where z.right = NIL is the same



delete z + slide T' up into place

→

What about when both $z.right$ & $z.left \neq$ NIL

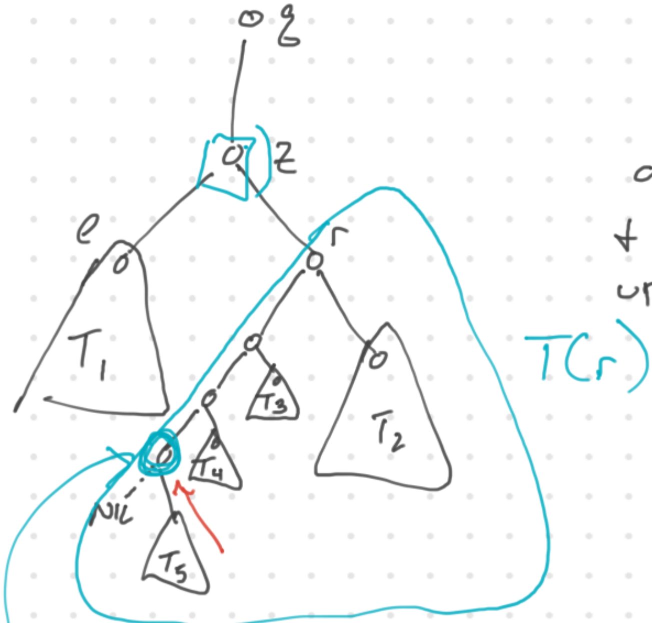


we want to delete z & add a new node in its place

We want to replace it with its successor in the tree order

an easy ~~case~~ subcase is when $z.right.left =$ NIL

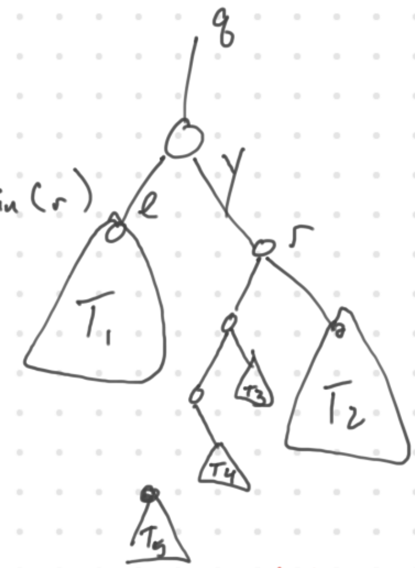
in This case we slide up r to The position of z



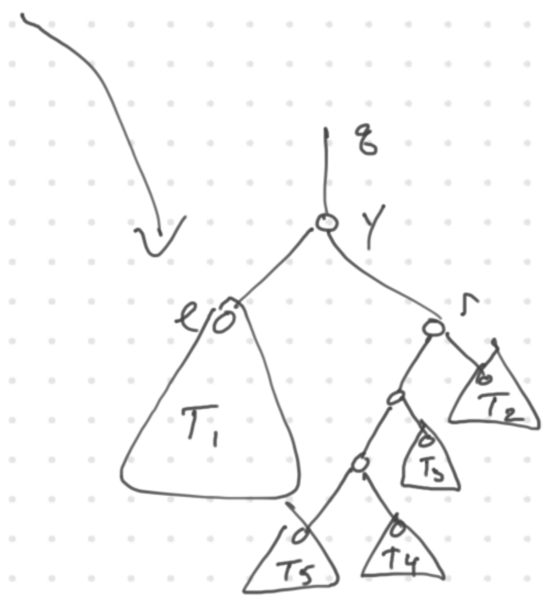
T(r) tree_min(r) = y

delete z

+ move tree_min(r) up to z



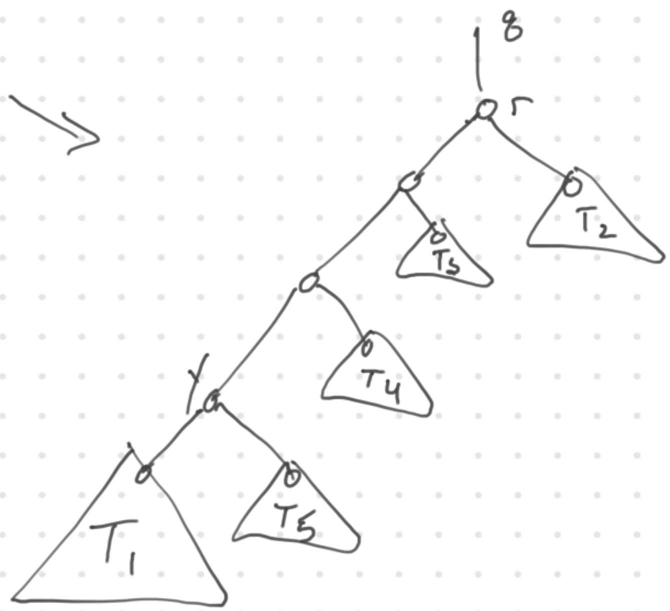
T_5 is no longer attached - slide it up into y's original position



an alternate way to delete

$y = \text{tree\_min}(r)$

we could move r to position of z
+ attach T_1 to the left child of y



preserves binary search tree property

But has effect of

height now (in the worst case) is equal to $\text{height}(T) - 1 + \text{height}(T_1)$ ie $\sim 2 \cdot \text{height}(T)$