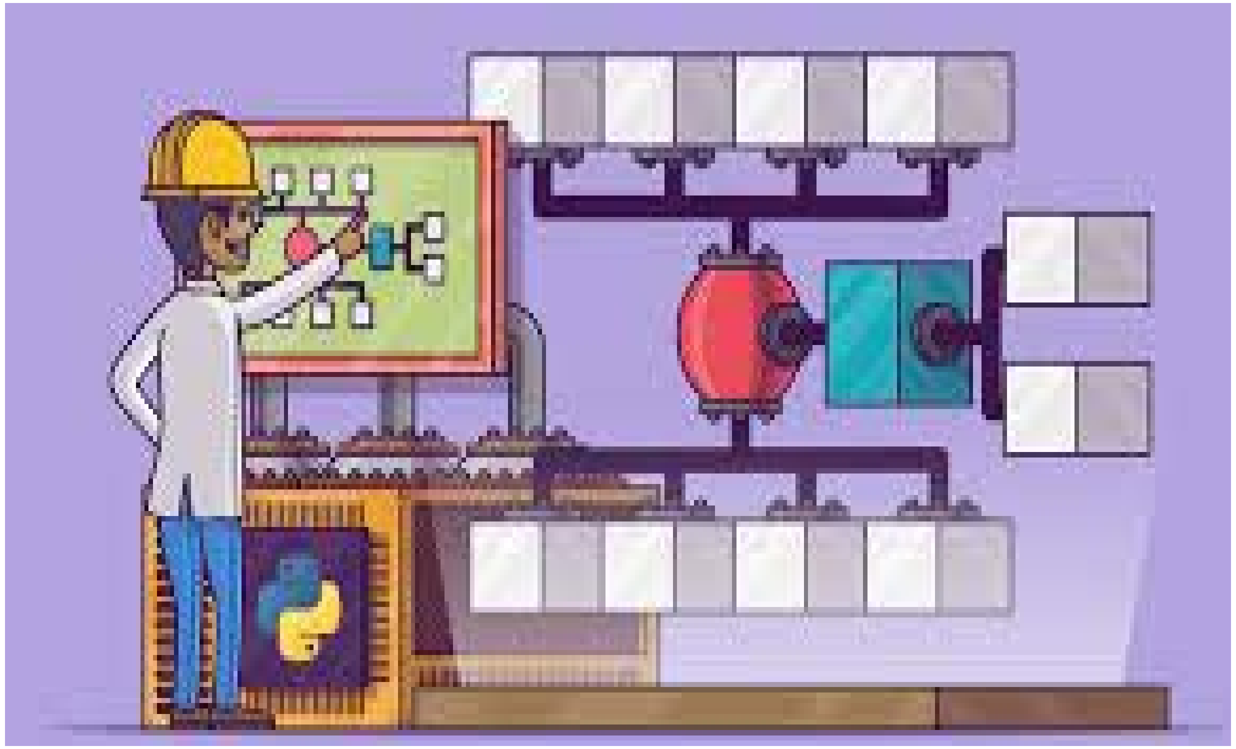
ساختمان داده ها



مدرس:
سمانه حسینی سمنانی

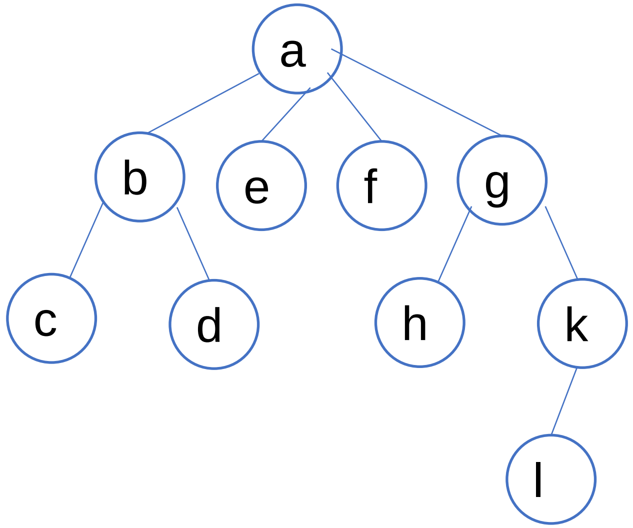
دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

پیاده سازی درخت

سمانه حسینی سمنانی
هیات علمی دانشکده برق و کامپیوتر- دانشگاه صنعتی اصفهان

پیاده سازی درخت

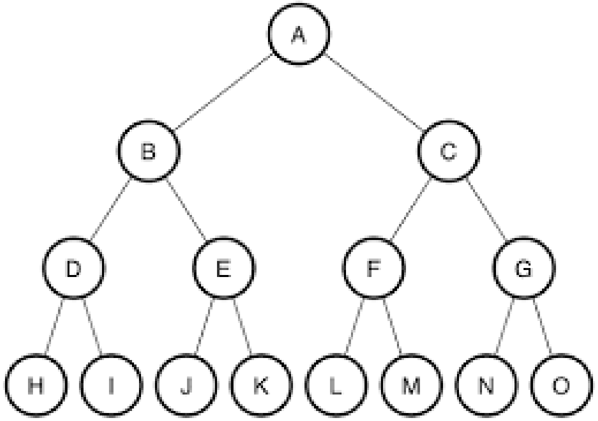
- پیاده سازی با آرایه
- معمولا اولین عنصر آرایه ریشه است.
- مزیت : دسترسی راحت هر گره به پدر.
- عیب : دسترسی به فرزندان مشکل است.



a	b	d	f	…
-1	0	1	0	…

پیاده سازی درخت

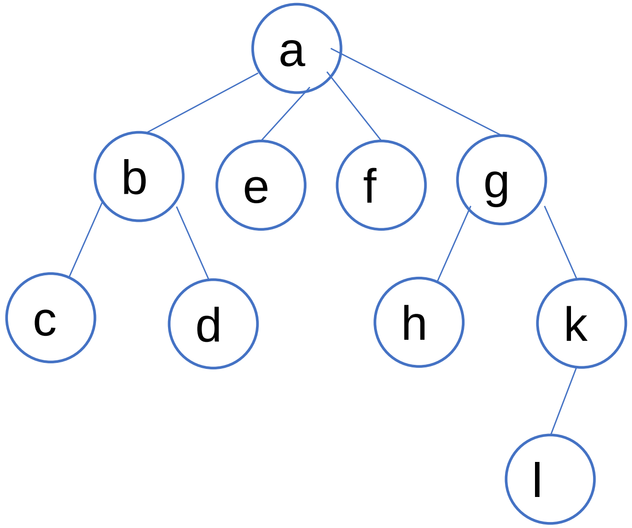
- پیاده سازی با آرایه
- فرض: هر گره حداکثر دو فرزند دارد .

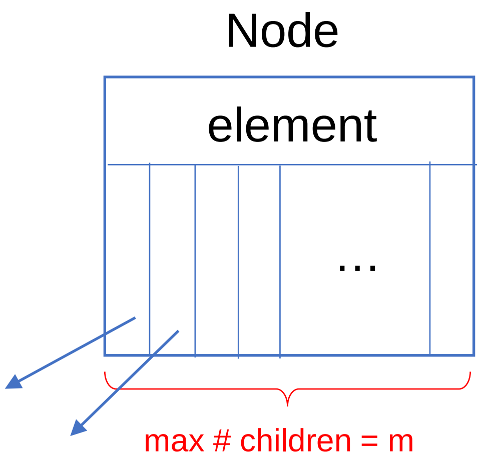


A	B	C	D	…
-1	0	0	1	…
1	3	5	…	
2	4	6	…	

پیاده سازی درخت

- پیاده سازی با اشاره گر ها
- فرض: هر گره حداکثر m فرزند دارد .

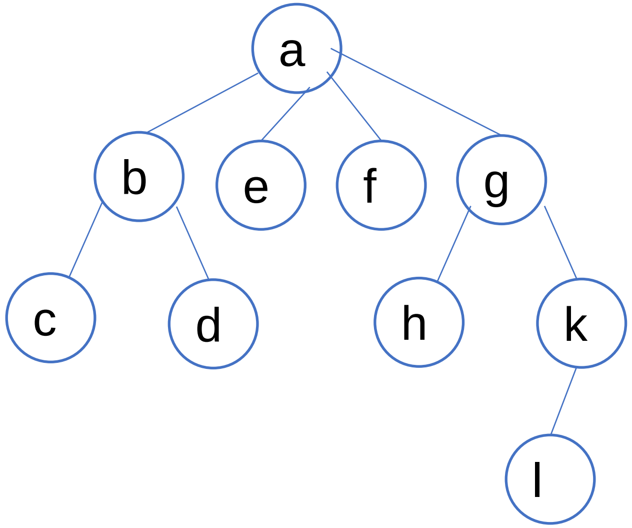


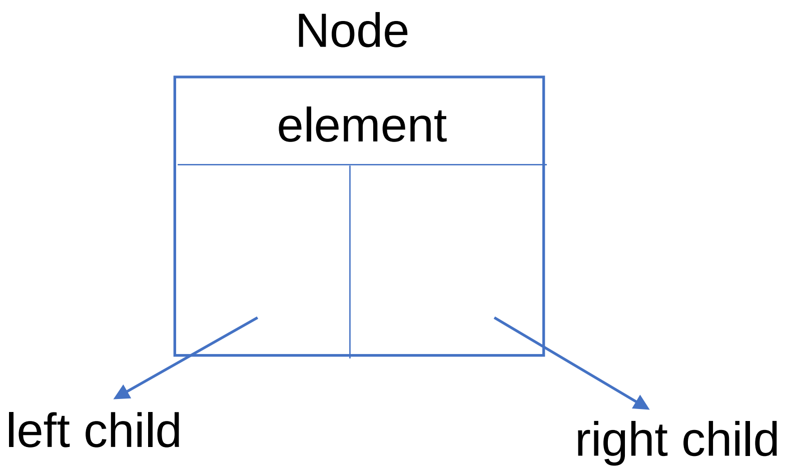


- کدام پیاده سازی از لحاظ مکان بهتر است؟
  - مقدار حافظه یکسان
  - مزیت روش پوینتر به آرایه در عدم تخصیص فضای اولیه و عدم وجود مشکلات پر شدن آرایه اولیه

پیاده سازی درخت

- پیاده سازی درخت دودویی با اشاره گر ها

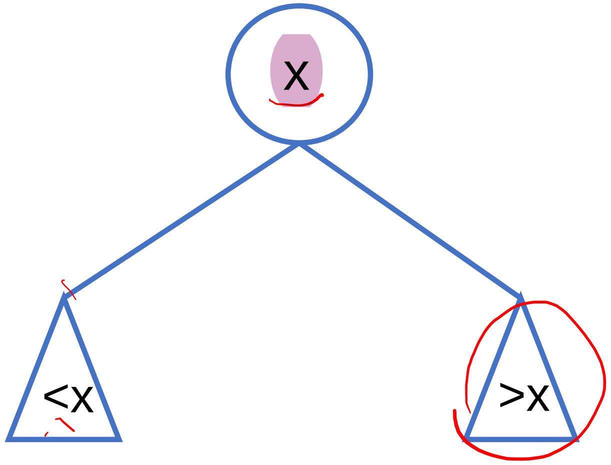




- درخت دودویی معادل
- بعضی خواص مشترک با درخت اصلی

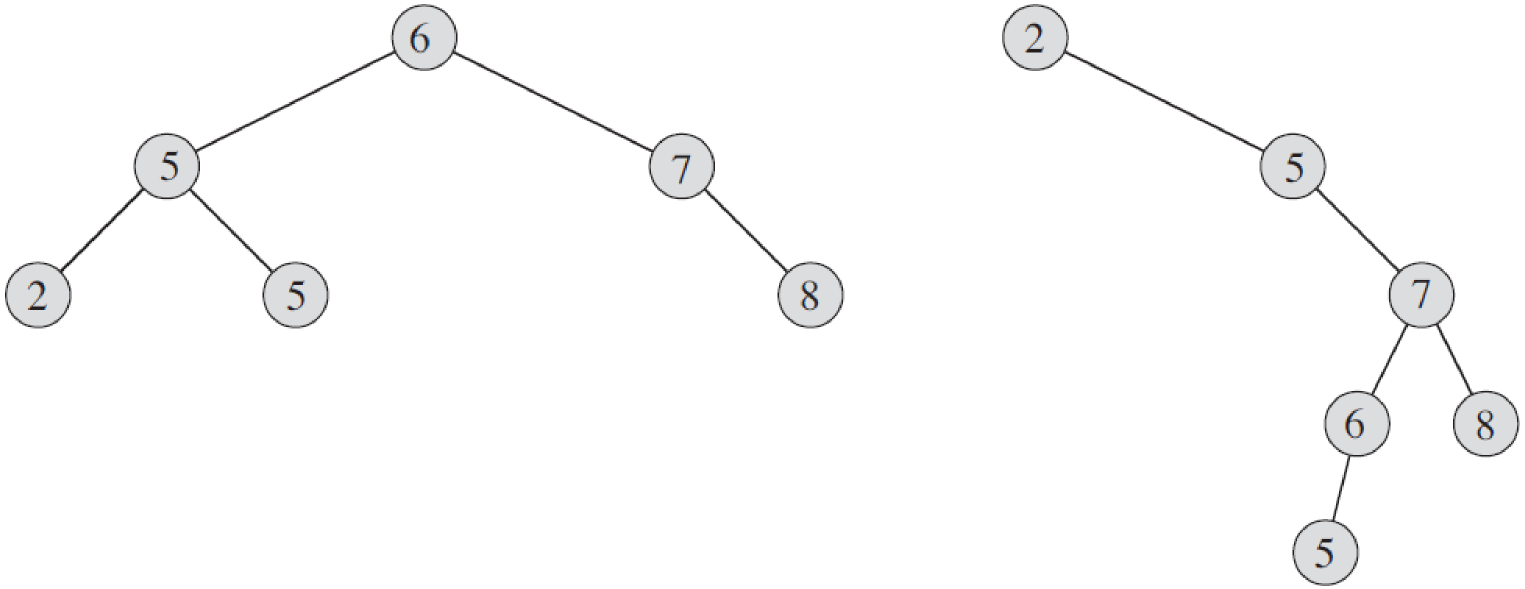
درخت جستجوی دودویی

- Binary search tree



درخت جستجوی دودویی

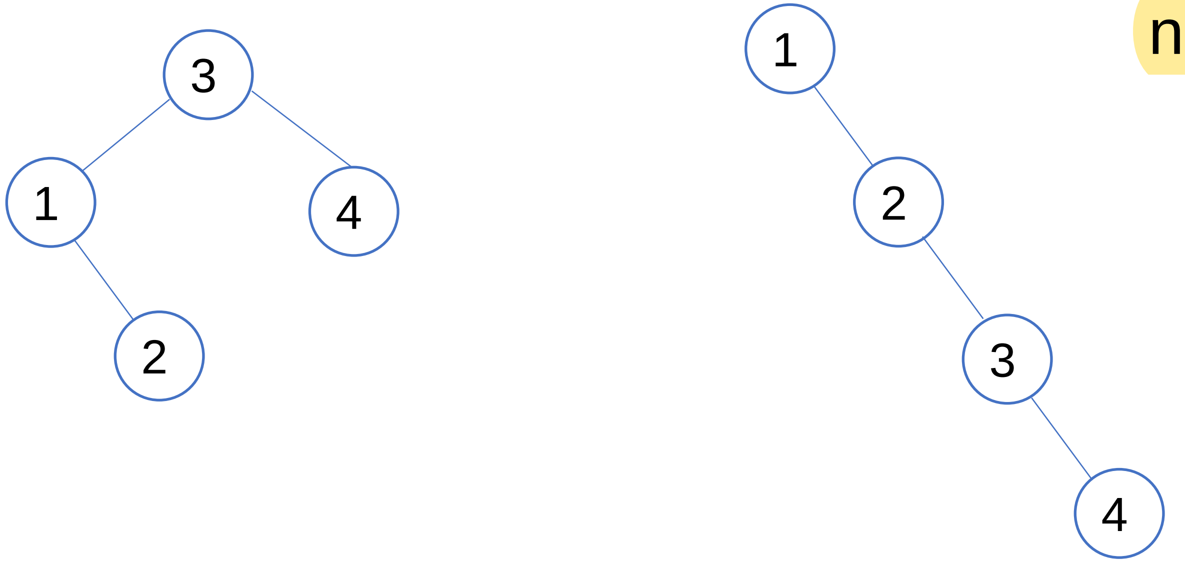
- Different binary search trees can represent the same set of values.



Different binary search trees

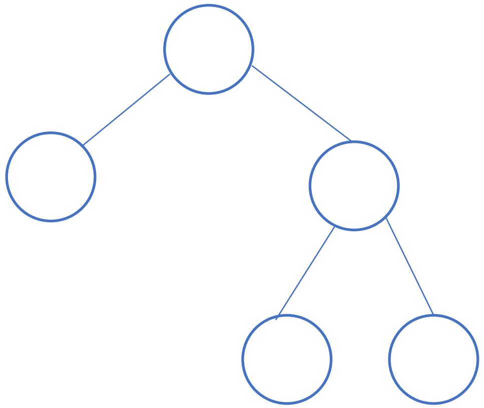
- 1, 2, 3, 4

- چند درخت با ارتفاع n-1



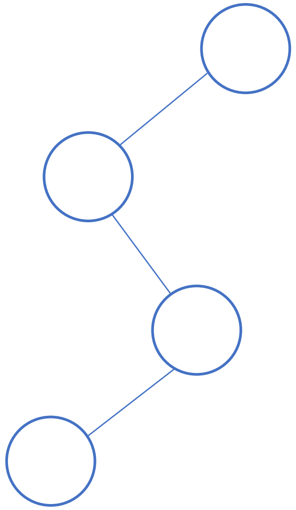
Different binary search trees

- 1, 2, 3, 4, 5



Different binary search trees

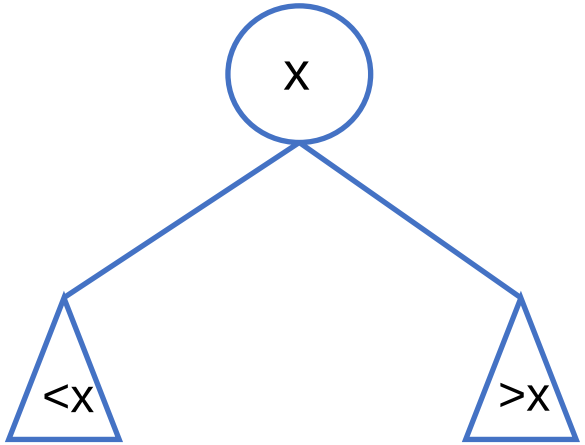
- 1, 2, 3, 4



- چند درخت با ارتفاع n-1 ؟ 2^{n-1}
- حداکثر ارتفاع؟ $n-1$
- حداقل ارتفاع؟ $\log n$

درخت جستجوی دودویی

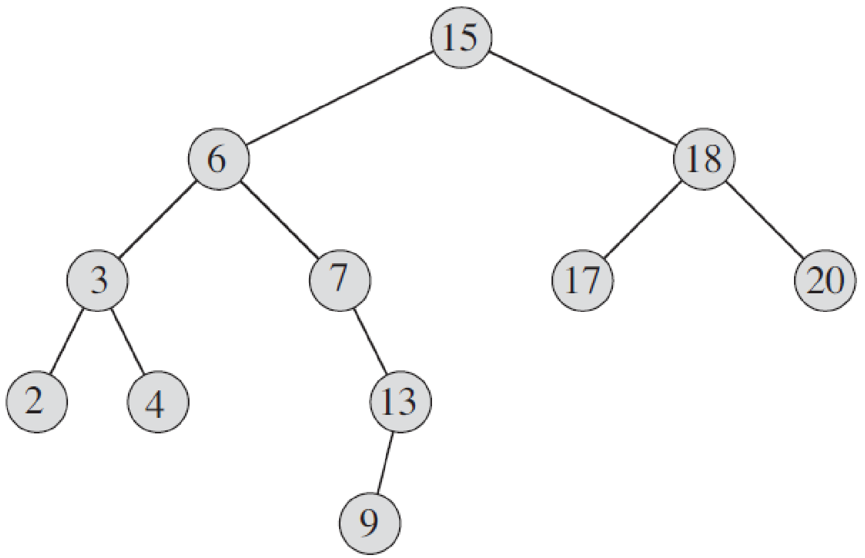
- The most important operation in binary search tree:
  - SEARCH
  - MINIMUM
  - MAXIMUM
  - PREDECESSOR
  - SUCCESSOR
  - INSERT
  - DELETE



Search

- SEARCH: Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

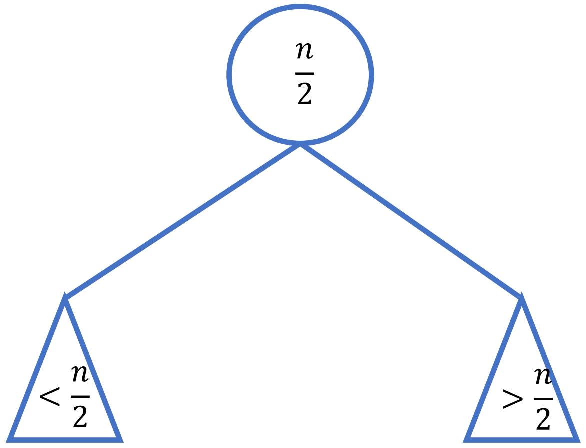
```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```



- To search for the key 13 in the tree, we follow 15 -> 6 -> 7 -> 13 from the root.

Search

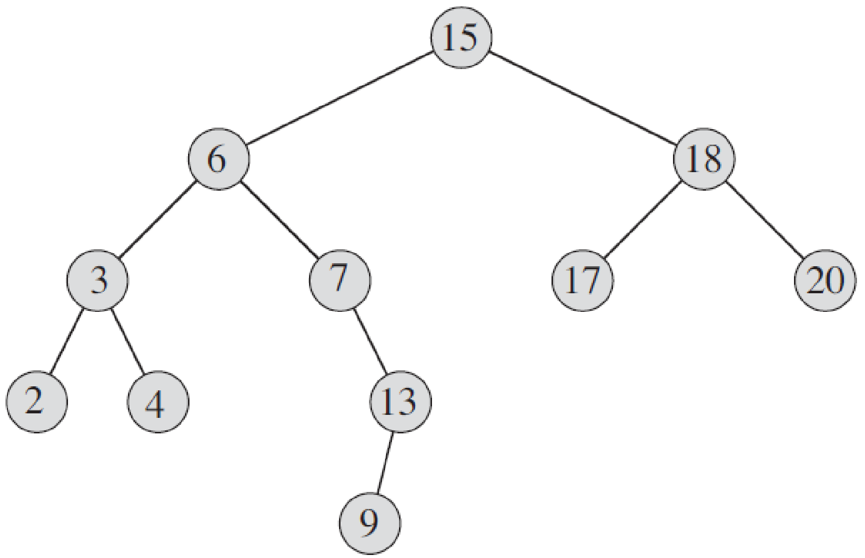
- Search operation in binary search tree take time proportional to the height of the tree
- We prefer to have binary search tree with minimum height
- For a complete binary tree with n nodes
- Worst case: $\theta(logn)$
- 1, 2, 3, …..n
- We don’t have all numbers from the beginning
- Various binary search trees:
  - Red- black tree
  - B-tree
  - , …



Search

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

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```



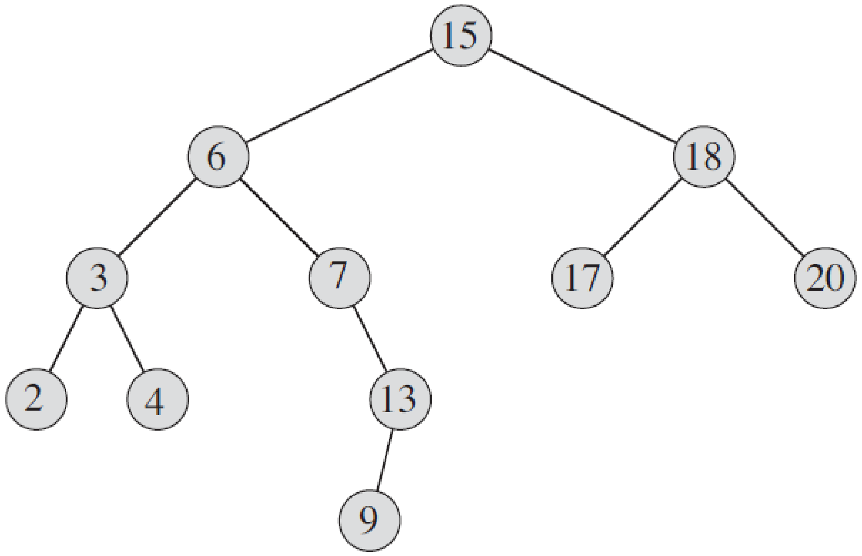
On most computers iterative version is more efficient.

Minimum and maximum

- We can always find an element in a binary search tree whose key is a minimum by following left child pointers from the root until we encounter a NIL.
- The binary-search-tree property guarantees that TREE-MINIMUM is correct.

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

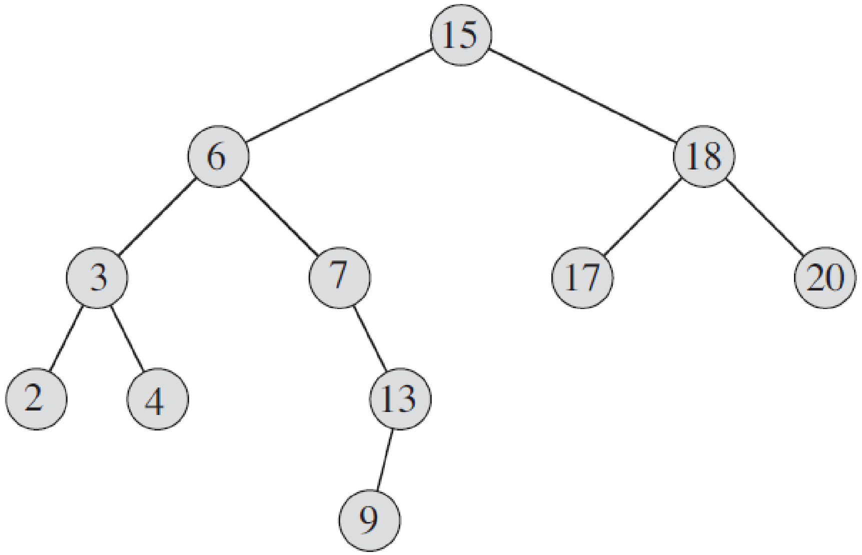


The minimum key in the tree is 2

Minimum and maximum

- Both of these procedures run in O(h) time on a tree of height h.

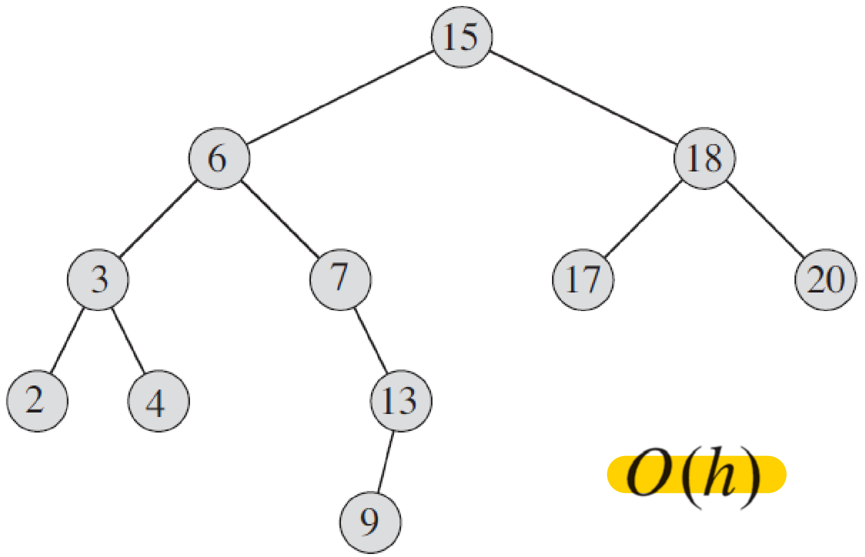
```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```



Successor and predecessor

- successor of a node x is the node with the smallest key greater than x:key.

```
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
```



$O(h)$

the successor of the node with key 15 is the node with key 17

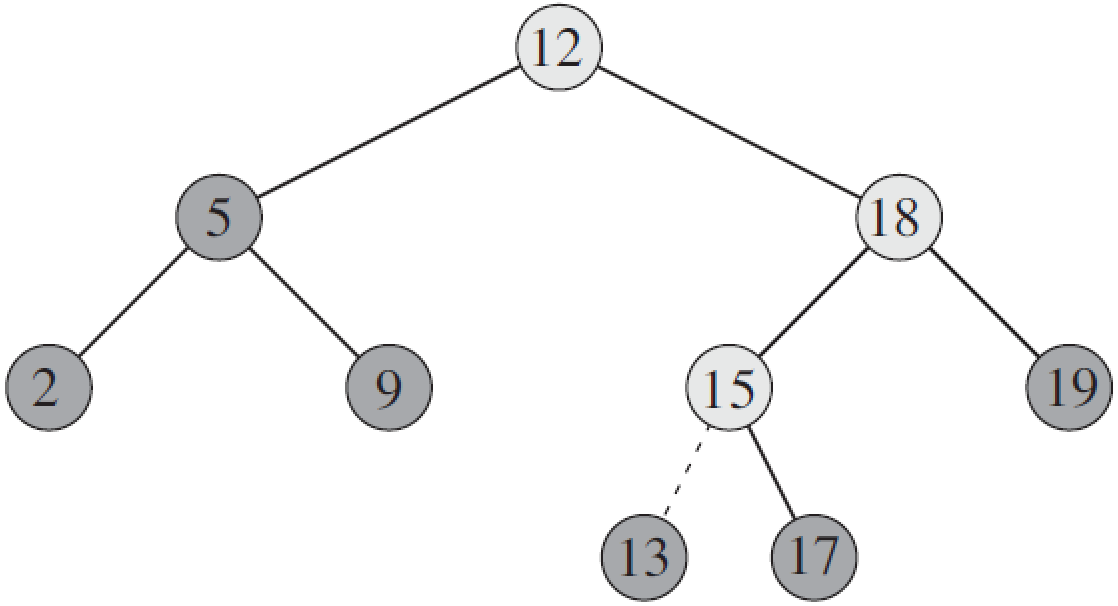
the successor of the node with key 13 is the node with key 15.

if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Insertion of node Z

TREE-INSERT(T, z)

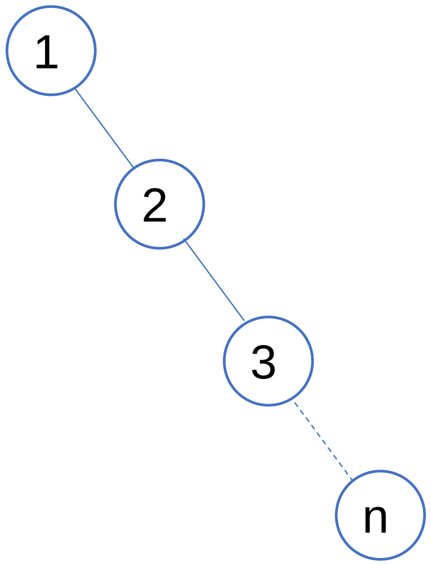
```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



$O(h)$

Insert

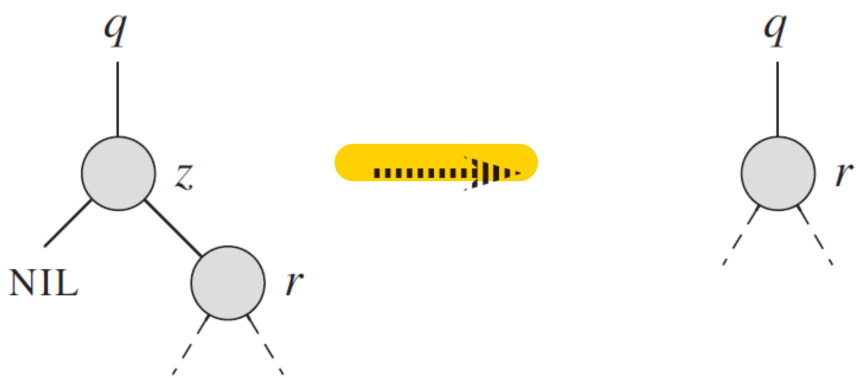
- This method can greatly increase the height of tree
- 1, 2, … n
- It is not good for search
- AVL tree tries to improve this process



Deletion of mode Z

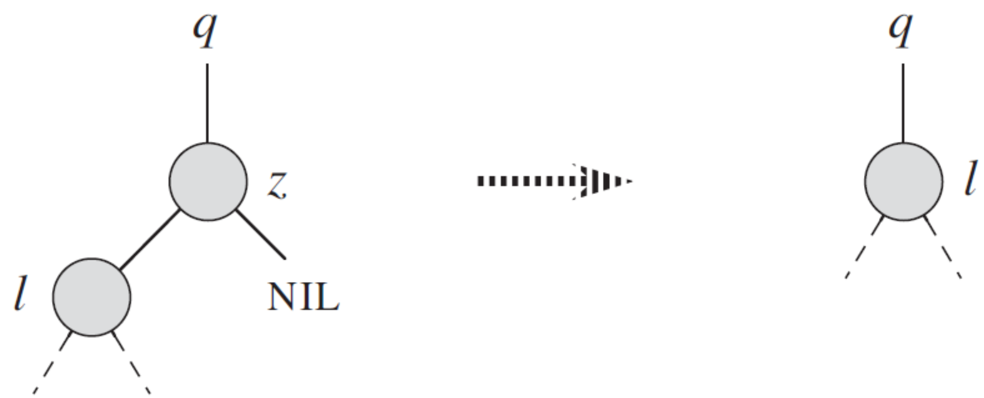
- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, then we find z's successor y—which must be in z's right subtree—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. This case is the tricky one because, as we shall see, it matters whether y is z's right child.

Deletion of mode Z



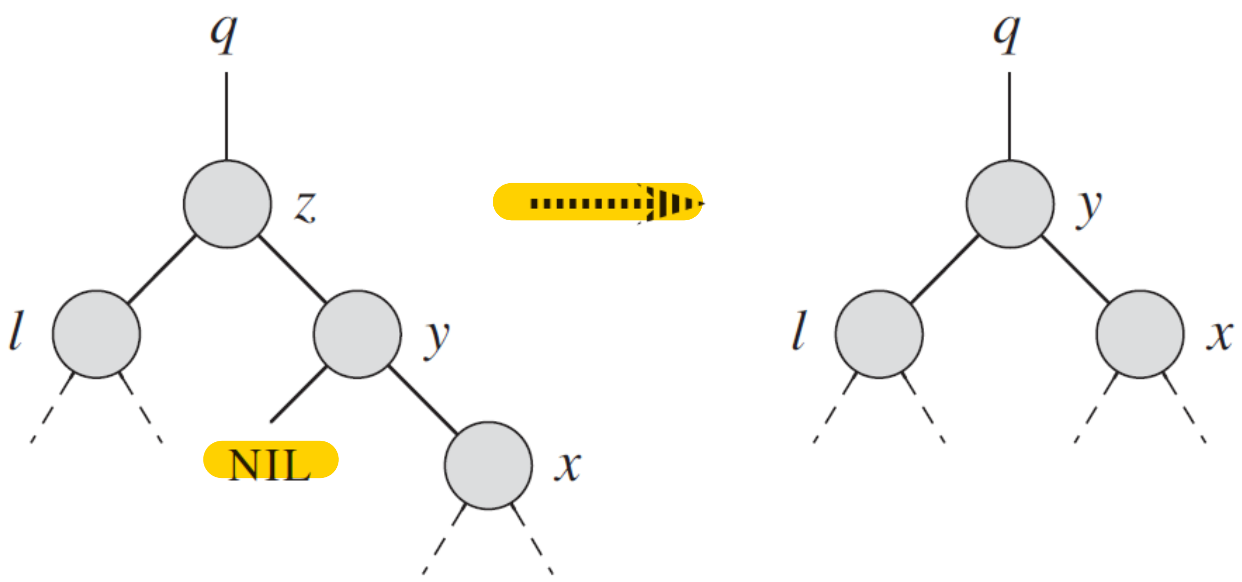
Node Z has no left child. We replace Z by its right child r, which may or may not be NIL

Deletion



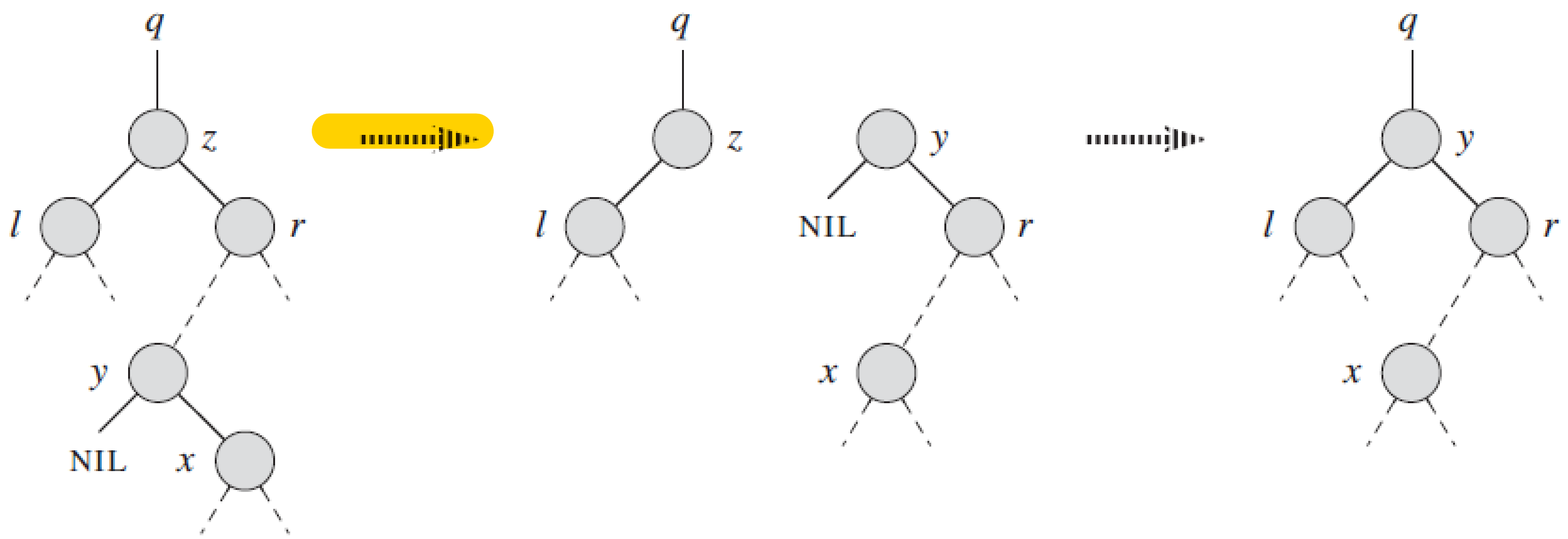
Node Z has a left child l but no right child. We replace Z by l

Deletion



Node z has two children;
It’s left child is node l , it’s right child is its successor y, and y’s right child is node x.
We replace z by y, updating y’s left child to become l, but leaving x as y’s right child.

Deletion



Node z has two children (left child l and right child r), and its successor y≠r lies within the subtree rooted at r.
We replace y by its own right child x, and we set y to be r’s parent.
Then, we set y to be q’s child and the parent of l .

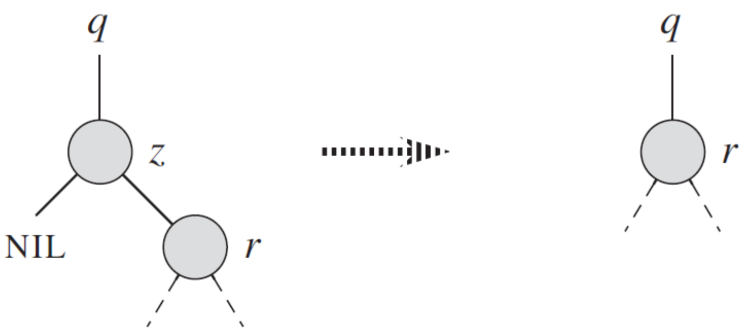
Deletion

Replaces one subtree as a child of its parent with another subtree.

```
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
```

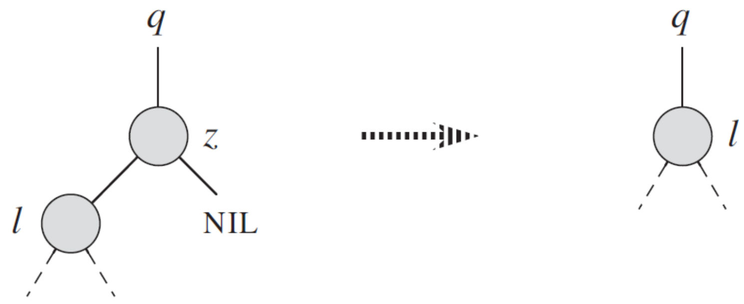
Deletion

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```



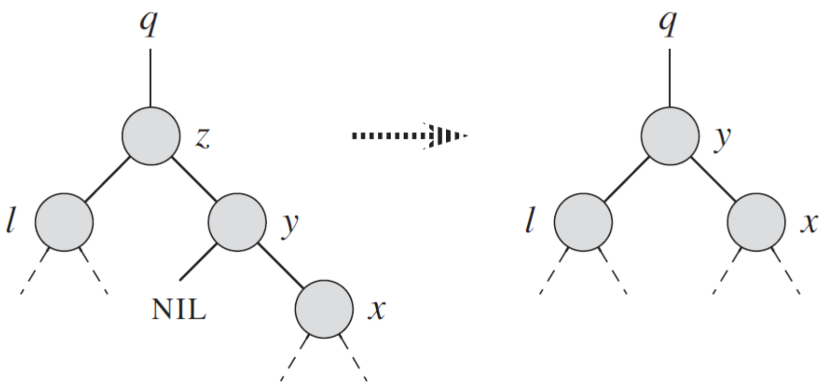
Deletion

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```



Deletion

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```



Deletion

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```