Binary search trees

Goal: A data structure which

(a) can find min, max

(b) can insert, delete

(c) can find pred, successor

(d) can "find"

efficiently

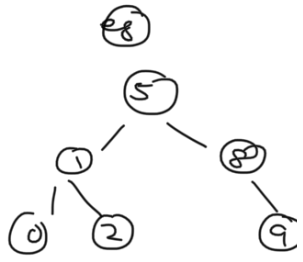
Binary tree :

↝ each node has 0, 1, or 2 children

↝ each node links to its parent, also links to its children

↝ store values at each node

Value property imposed on binary tree :

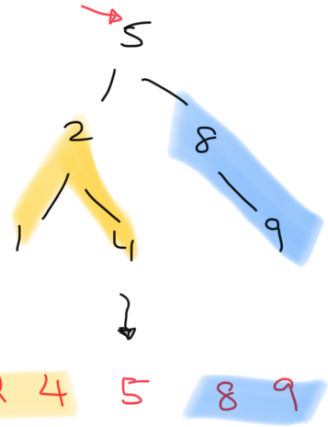


- no duplicate values
- at each node v,

(a) values in left subtree of v $<$ value at v

(b) values in right subtree of v $>$ value at v

In order traversal



→ walk through left subtree

→ then walk through node

→ then walk through right subtree

Recursive implementation

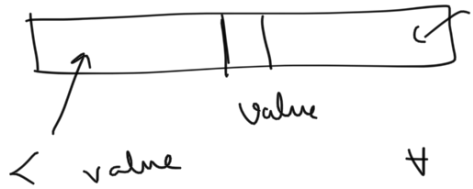
```
inorder traversal ( vertex v):

    if v is not None :

        inorder traversal ( v . left child)
        print ( v's value)
        inorder traversal ( v . right child)
```

Inorder traversal : outputs sorted list

because



Find value v in binary search tree T

Recursive

```
def find (v, root node) :

    if root node is None :        ← empty
        return false
```

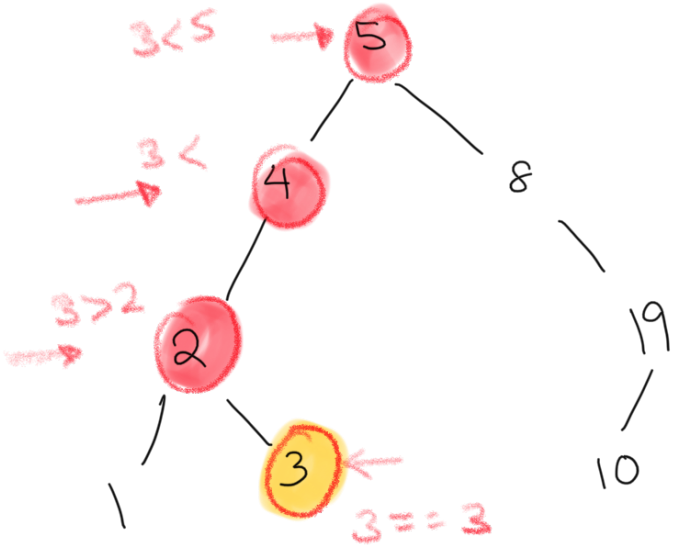
```
If      root node value == v :
            return true
    If      v < root node value
            return find ( v, root node . left child )
    If      v > root node value
            return find ( v, root node . right child )
```

Subtree with root node = root node . left child

Subtree with root node = root node . right child

Iterative

```
find ( v, rootnode ):
    while ( root node not none ):
        If      root node's value == v :
                return true
        If      v < root node's value :
                root node = root node . left child
        If      v > root node's value :
                root node = root node . right child
    return False
```



find (3)

iterative version

- (walk down path...)

---

min (binary search tree) ····> left most node's value

Assume tree not empty.

recursive

```
def minval (rootnode):
    if rootnode.leftchild is none:
        return rootnode's value
    else:
        return minval (rootnode.leftchild)
```

iterative

(keep going left until you can no longer go left)

Assume tree not empty.

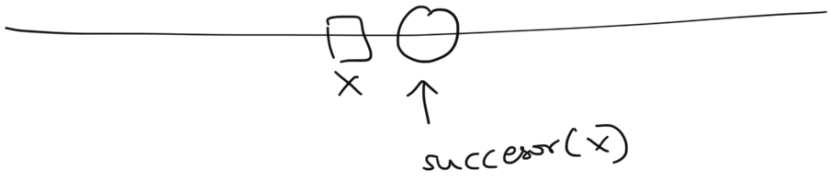
```
def minval (rootnode):
    while (rootnode.leftchild is not none):
        root node = rootnode.leftchild
    return rootnode's value
```

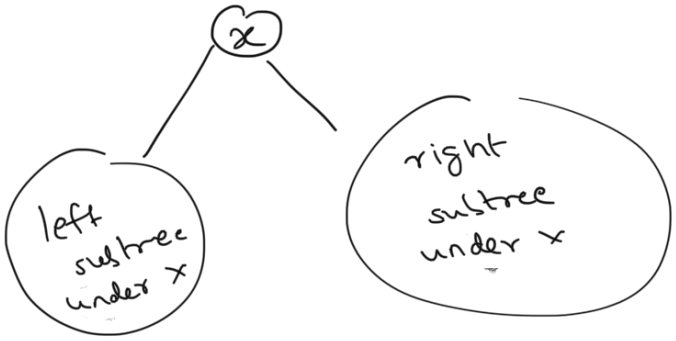
Similarly max value (binary search tree)

→ value of right most node

Finding Successor (value = x) :

Inorder traversal:

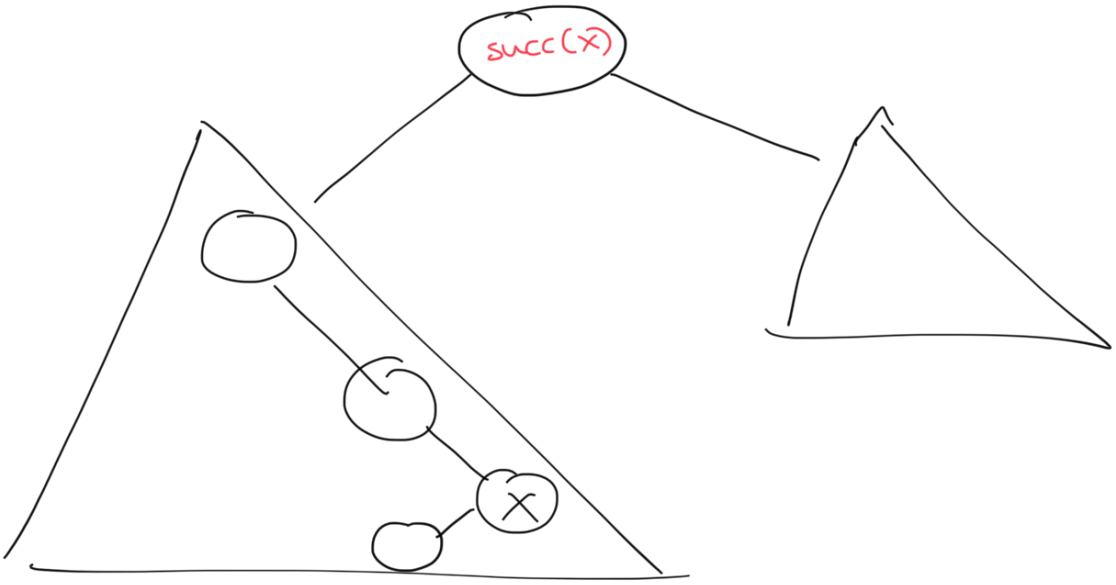




Successor (x)

: min value in (right subtree under x)

what if x has no right subtree under it ??



So x is max of subtree it belongs to

succ(x) ... → keeping going up (ie) you should be right child of your parent

until you are left child of your parent

→ then that parent is succ(x)

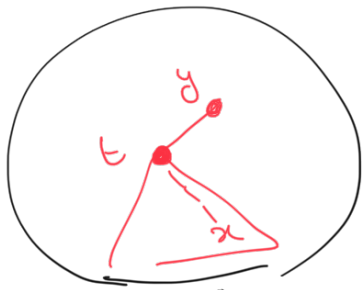
Code

```
def succ (node t):
    if t.rightchild is not None:
        return minval (t.rightchild)
```

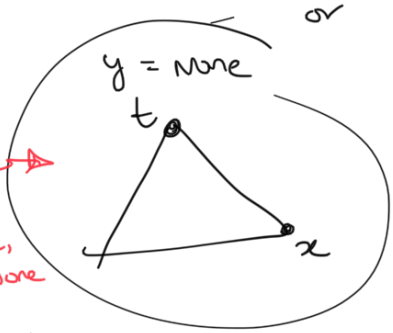
min of right subtree with node = t.rightchild

keeping going up until you are not or you are the right child of your parent

```
y = t.parent
while (y != None and t == y.rightchild)
    t = y
    y = t.parent
return y
```

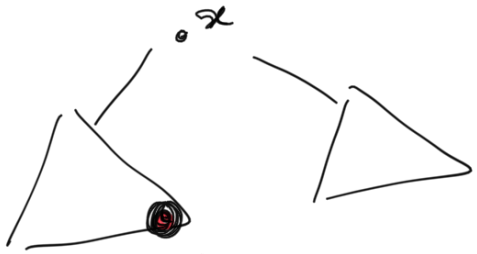


or



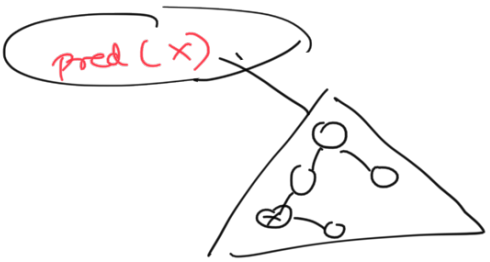
if x is max value of search tree, returns None as no successor

predec.

- 

pred(x) = max(left subtree under x)

- if no left child of x



keep walking up until you turn left - .

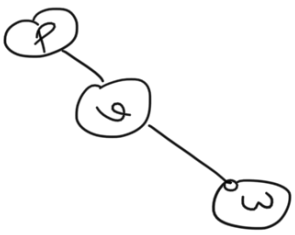
insert(v):

- Try to find v
- insert it where search fails..

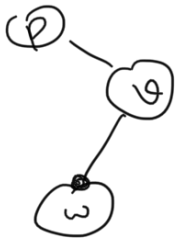
Delete(v)

- Find v

Case I if leaf, delete it



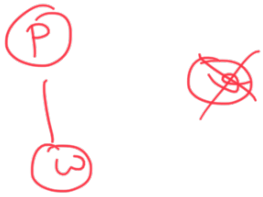
Case II



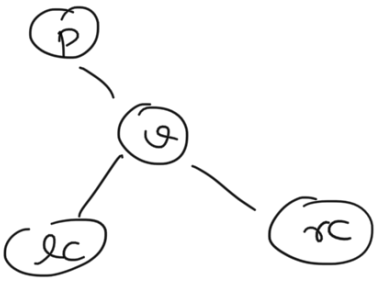
or

only 1 child

update link to p directly to w



Case III



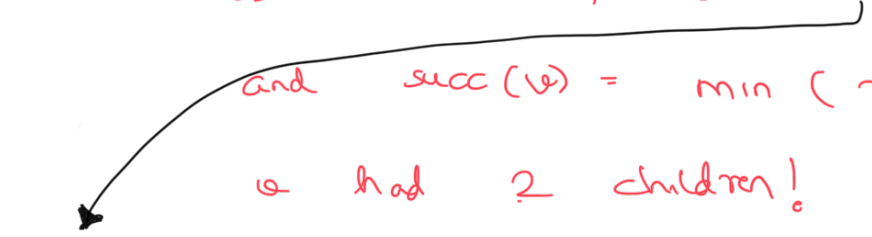
- Replace v with pred(v) (or succ(v))
- Delete pred(v) or succ(v)

Note that these will be leaf / have only 1 child

as here pred(v) = max(left subtree)

and succ(v) = min (right subtree) as

v had 2 children!



In particular, don't have to walk up tree for finding pred for case when no left subtree. -]

All operations are O (height of tree)