Binary Search Trees

CMPT 225

ADTs related to Sets

- Set: unordered collection of values/objects
- Operations:
  - insert(x) // add x to set
  - member(x) // check if x in set. a.k.a. find(x), search(x), lookup(x) ...
  - remove(x) // remove x from set
  - size() // get size of set
  - empty() // is set empty?
  - clear() // remove all elements (i.e. make set empty).
- We call the values we store keys,
- We assume the keys are from some ordered set S

  ie, for any two keys $x, y \in S$, we have exactly
  one of $x<y$, $x=y$, $y<x$

- Want implementations where all operations are efficient/fast

Q: What will count as "fast"?

ADTs related to Sets

- Consider time complexity of operations for simple list + array implementations:

	insert	find	remove
un-ordered array	$O(1)$	$O(n)$	$O(n)$
ordered array	$O(n)$	$\boxed{O(\log n)}$	$O(n)$
un-ordered linked list	$O(1)$	$O(n)$	$O(n)$
ordered linked list	$O(n)$	$O(n)$	$O(n)$

Q: What will count as "fast"?

A: Time $O(\log n)$ // n is size of set

Some Related Container ADTs

- Multiset: like set, but with multiplicities (aka <u>bag</u>)
  - count(x)

- Map: unordered collection of <key, value> pairs, associating at most one value with each key. (e.g. partial function Keys $\rightarrow$ Values).
  - put(key, val) // in place of insert x
  - get(key) // returns value associated with key

- Dictionary: like map, but associates a collection of values with each key.

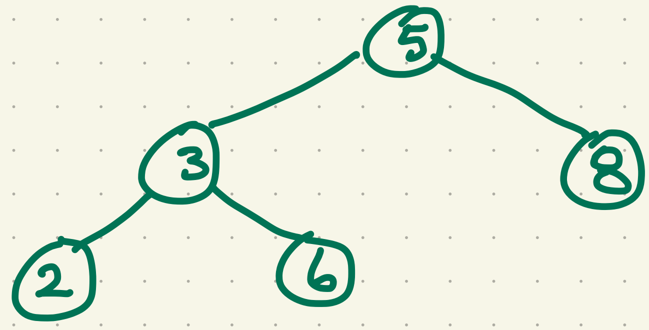
Implementations of these are simple extentions to implementations of <u>sets</u>, which we focus on.

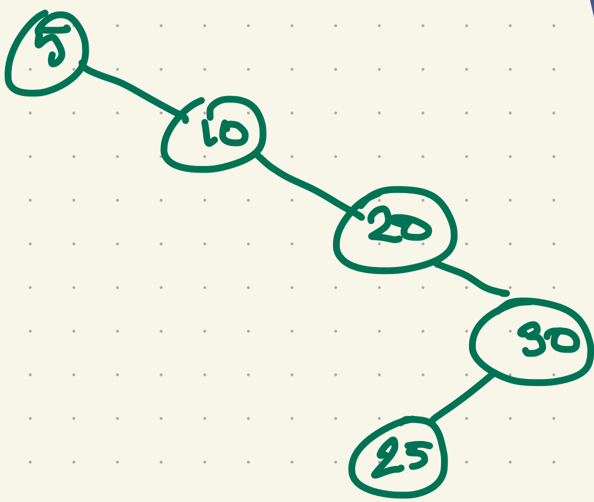
Binary Search Trees (BSTs)

A BST is
- a binary tree // a structure invariant
- with nodes labelled by keys
- satisfying the following order invariant.

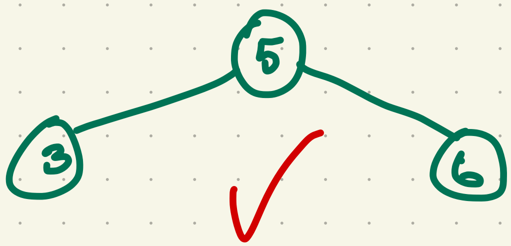
for every two nodes u, v:
- if u is in the left subtree of v
  then $key(u) < key(v)$
- if u is in the right subtree of v,
  then $key(u) > key(v)$

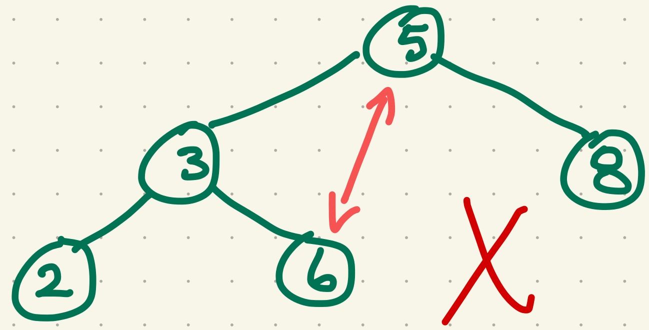
Ex

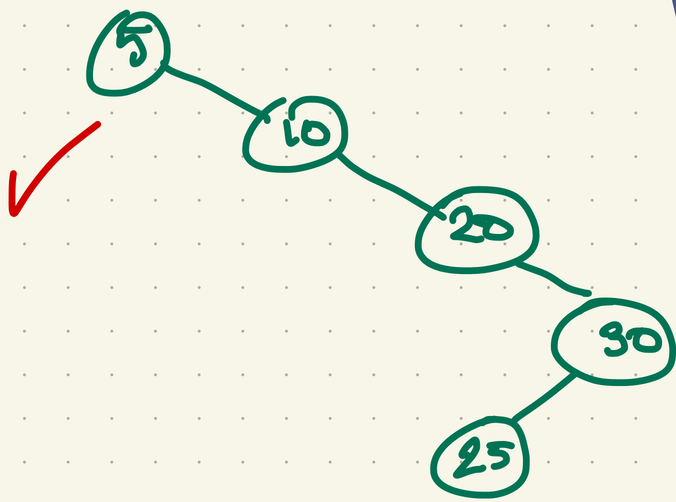




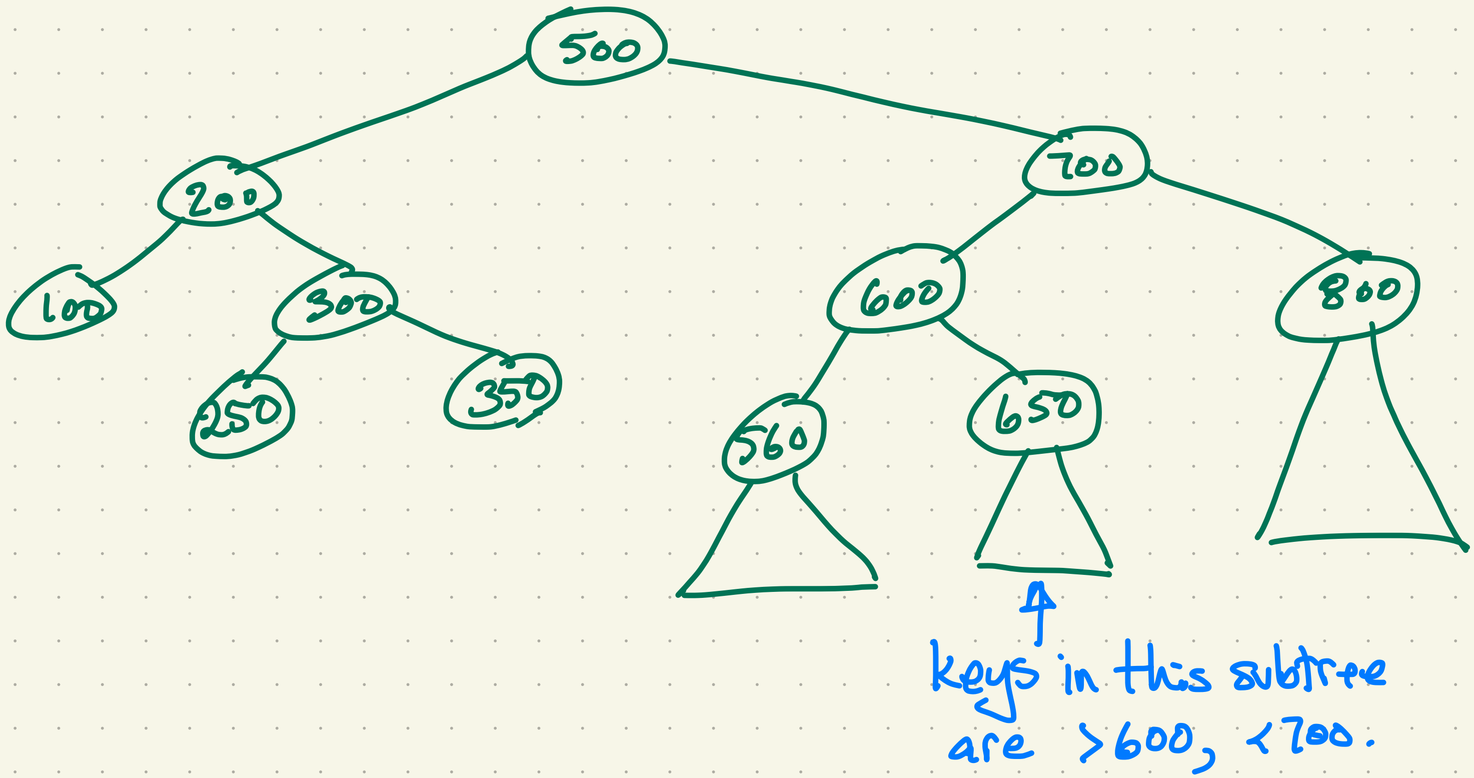
Ex







Every subtree of a BST is a BST.



This makes recursive algorithms very natural.

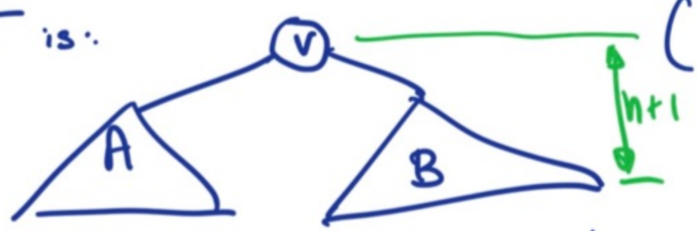
Fact: In-order traversal of a BST visits keys in non-decreasing order.

Proof Sketch:

Basis: $h=0$, so one node, ✓

I.H.: The claim holds for trees of height $\le h$.

I.S.: T is:



(A, B may be empty)

we: 1) traverse A, visiting key in sequence $a_1, a_2, \dots a_k$.

2) visit v

3) traverse B, visiting keys in sequence $b_1, b_2, \dots b_m$

Overall, we visit:

$$a_1\, a_2 \dots a_k\, v\, b_1\, b_2 \cdots b_m$$

By I.H. $a_1 \le a_2 \le \dots \le a_k$

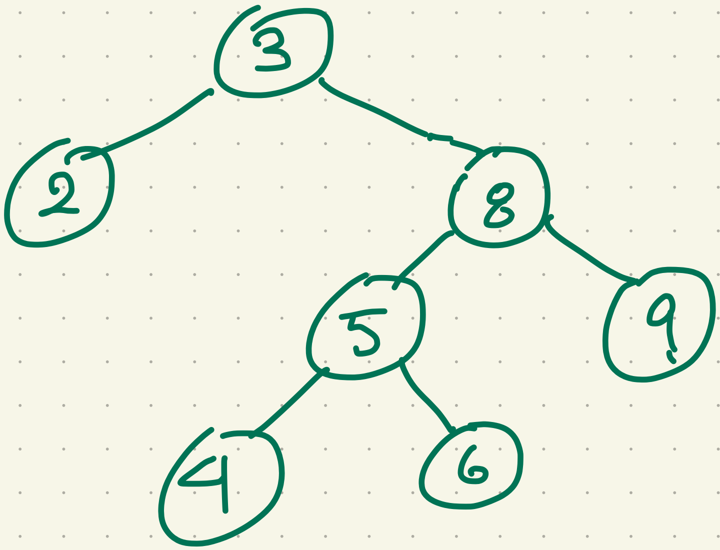
$b_1 \le b_2 \le \dots \le b_m$

Because T is a BST, so $a_k \le key(v) < b_1$

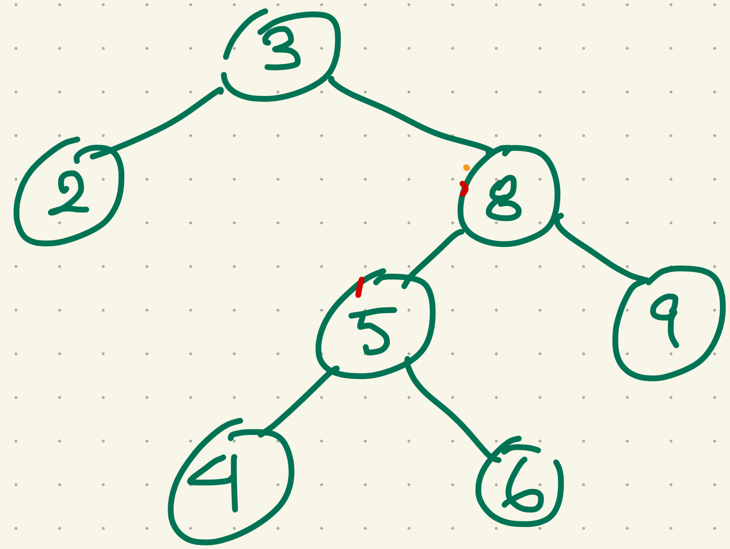
$\therefore$ $a_1 \le a_2 \le \dots \le a_k \le key(v) \le b_1 \le b_2 \cdots \le b_m$.

BST Find / Search : examples

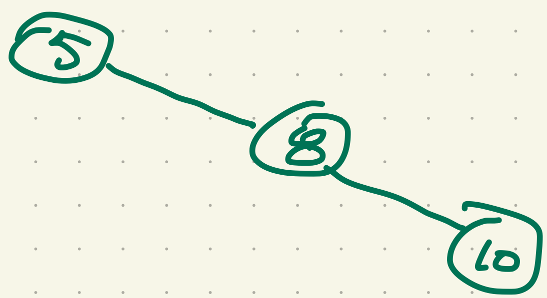
find(5)



find(7)

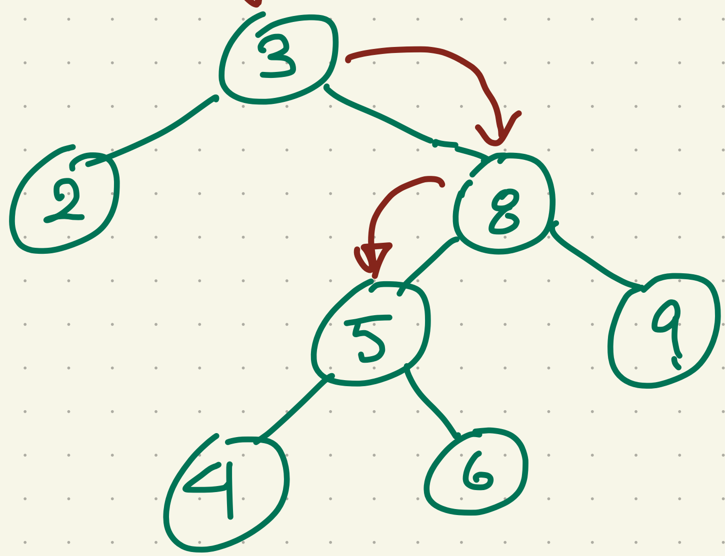


find 6

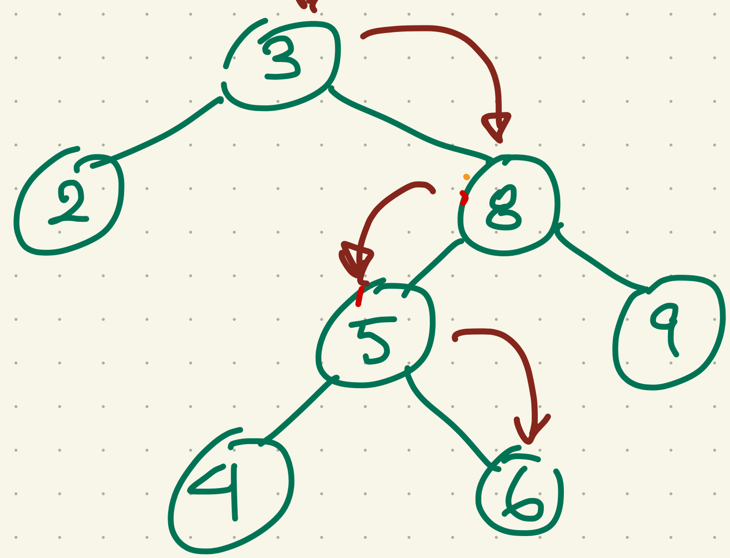


BST Find: Chooses sub-trees

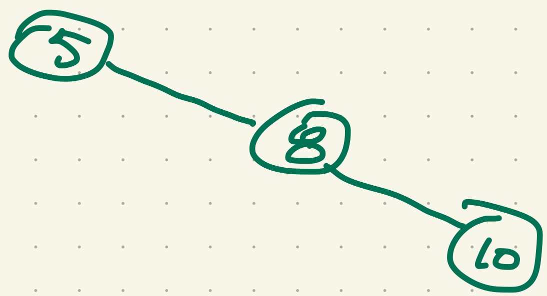
find(5)



find(7)

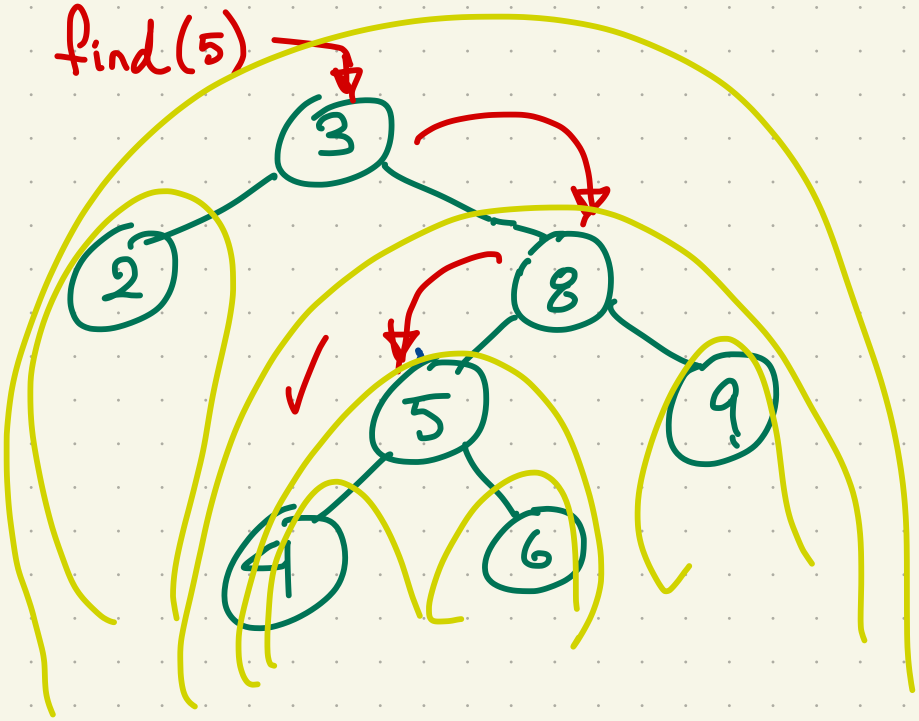


find 6

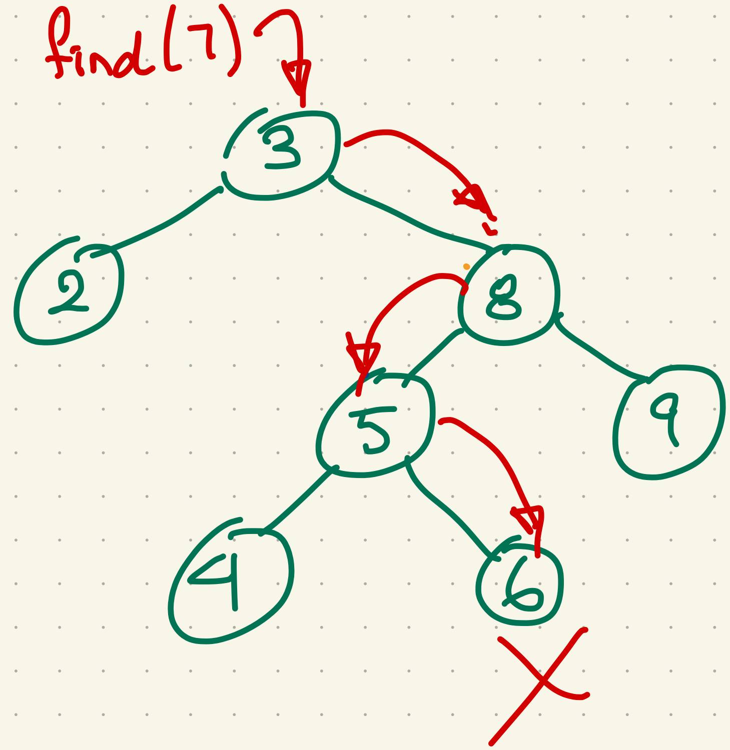


BST member/find: examples

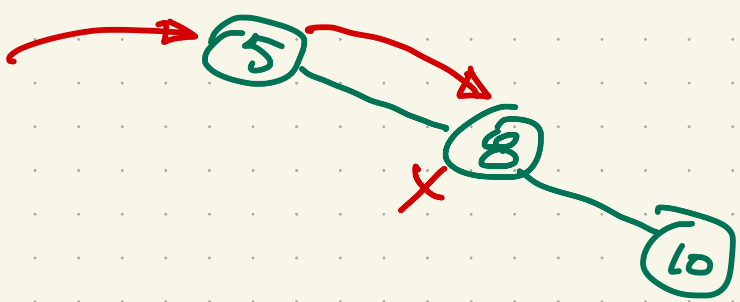
find(5)



find(7)



find 6



Some notation

Suppose v is a node of a BST. We write:

$\text{left}(v)$ = left child of v

$\text{right}(v)$ = right child of v

$\text{key}(v)$ = key labelling v

$\text{node}(x)$ = node v s.t. $\text{key}(v) = x$.

BST find(x) Pseudo-code

```
find (t) { // return true iff t is in the tree.
    return find(t, root)
}


find (t, v) // return true if t appears in
{           // subtree rooted at v.

    if  t < key(v)  & v has a left subtree
        return  find(t, left(v))
    if  t > key(v)  &  v has a right subtree
        return  find(t, right(v))

    if  key(v) = t
        return true
    reteern false // v is a leaf, does not have t
}
```

BST find(t,v) pseudo-code – alternate version

```
find (t, v) // return true if t appears in
{           // subtree rooted at v.

   if key(v) = t
       return true
   if t < key(v) & v has a left subtree
       return find(t, left(v))
   if t > key(v) & v has a right subtree
       return find(t, right(v))
   retearn false
}
```

Q: Which version is better?

A: key(v)=t will almost always be false, so the first version should do fewer comparisons, and usually be faster.

BST insert(x) Pseudo-code

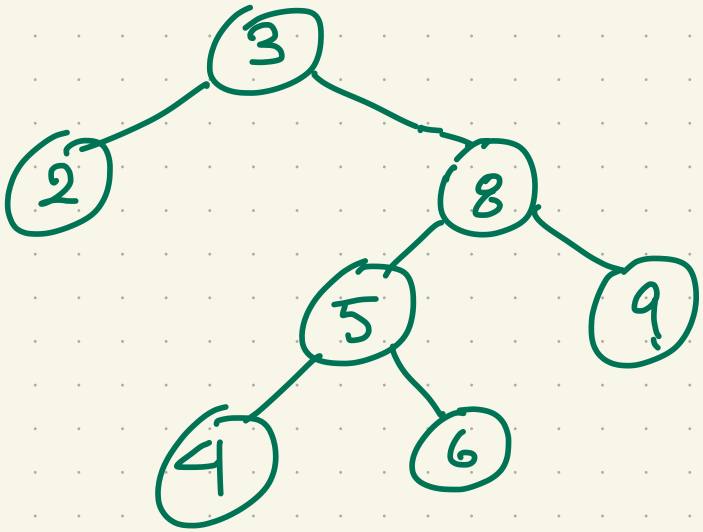
```
insert(t){
    // adds t to the tree
    // assumes t is not in the tree already*
    u ← node at which find(t,root) terminates **
    if  t < key(u)
        give u a new left child with key t
    else
        give u a new right child with key t.
}
```

* Exercise: Write the version that does not make this assumption.

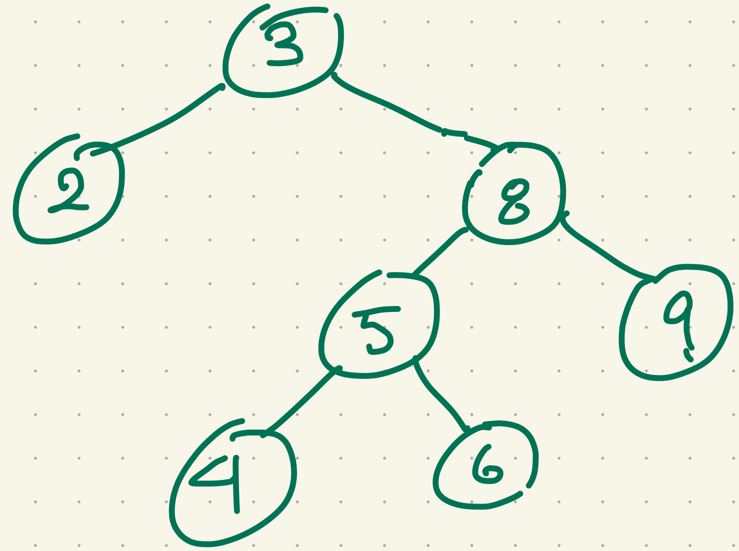
** Exercise: Write the version where the search is explicit.

BST Insert Examples

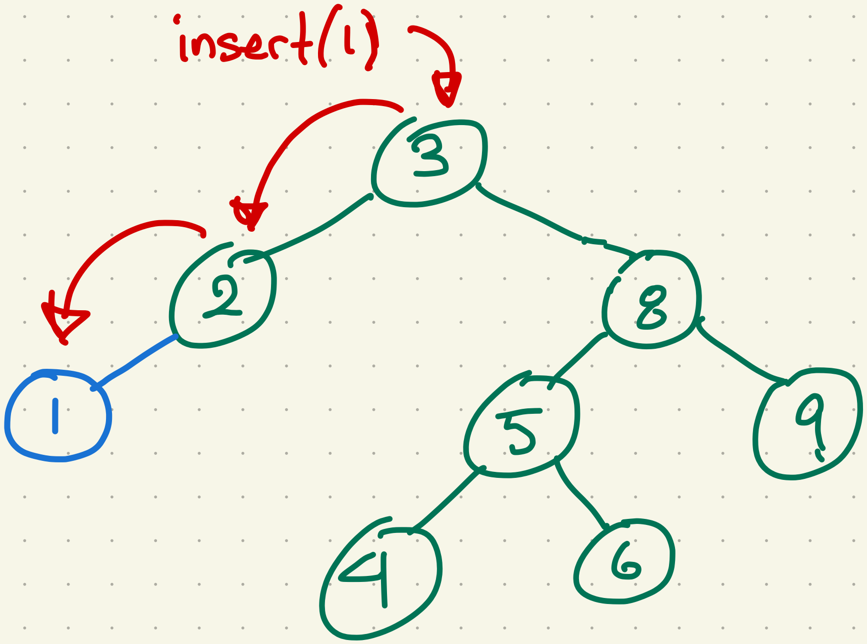
insert(1)

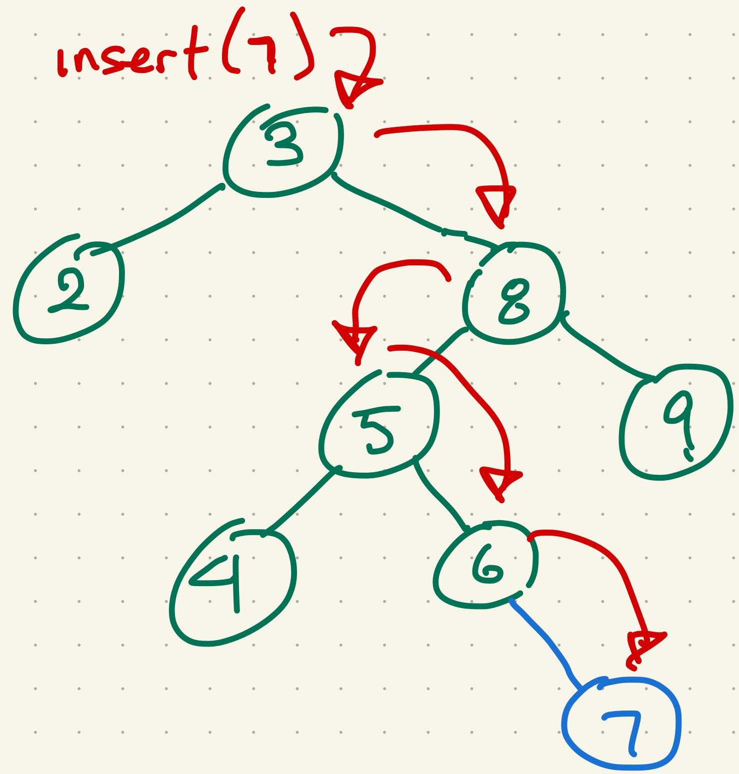


insert(7)



BST Insert Examples





BST insert(x) Pseudo-code - explicit search version.

```
insert(t){ //adds t to the tree, if it is not already there.
    insert(t,root)
}

insert(t,v)// insert t in the subtree rooted at v, if it is not there.
{
    if  t < key(v)  & v has a left subtree
        insert(t, left(v))

    if  t > key(v)  &  v has a right subtree
        insert(t, right(v))
    if  t < key(v) // here v has no left child
        give v a new left child with key t
    if  t > key(v) // here v has no right child
        give v a new right child with key t.

    //if we reach here, t = key(v), so do nothing.
}
```

Insertion Order for BSTs: Examples.

1). start with an empty BST
 - insert 5, 2, 3, 7, 8, 1, 6 in the given order

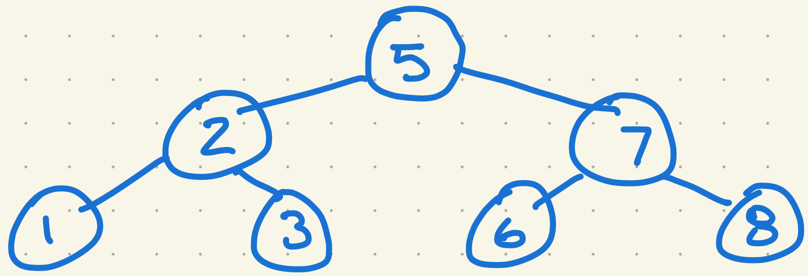
2). start with an empty BST
 - insert 1, 2, 3, 5, 6, 7, 8 in the order given

* Insertion order affects the shape of a BST
* Removal order can too.

Insertion Order for BSTs: Examples.

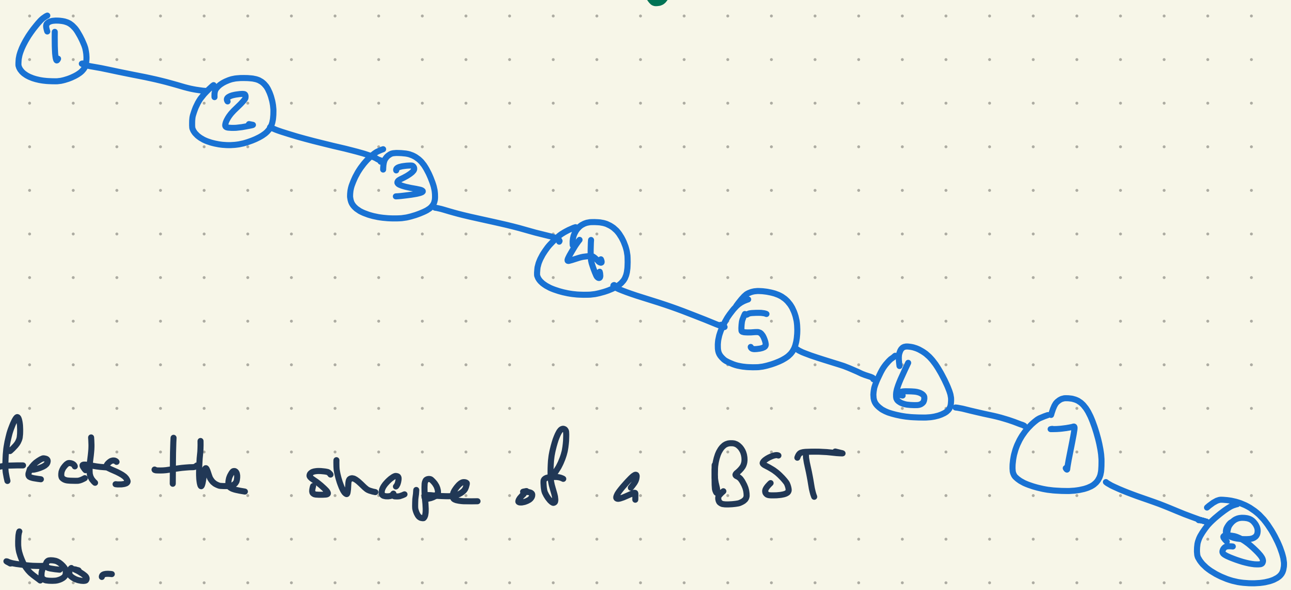
1). start with an empty BST

- insert 5, 2, 3, 7, 8, 1, 6 in the given order



2). start with an empty BST

- insert 1, 2, 3, 5, 6, 7, 8 in the order given



* Insertion order affects the shape of a BST

* Removal order can too.

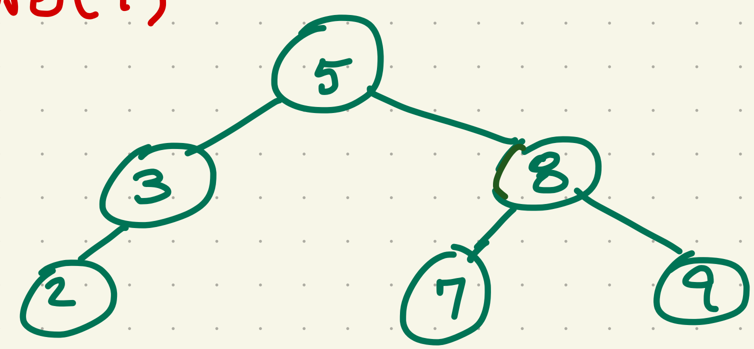
BST remove(t)

- We consider 3 cases, of increasing difficulty.

- Case 1: t is at a leaf

> i) find the node v with $key(v) = t$
> ii) delete v

Ex: remove(7)



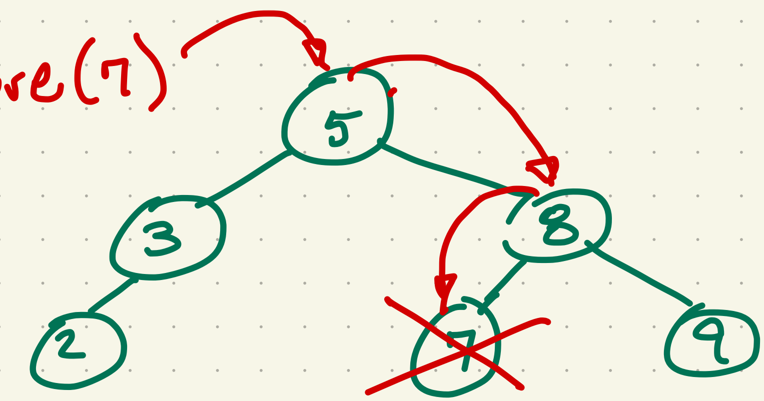
BST remove(t)

- We consider 3 cases, of increasing difficulty.

- Case 1: t is at a leaf

i) find the node v with key(v) = t ii) delete v

Ex: remove(7)

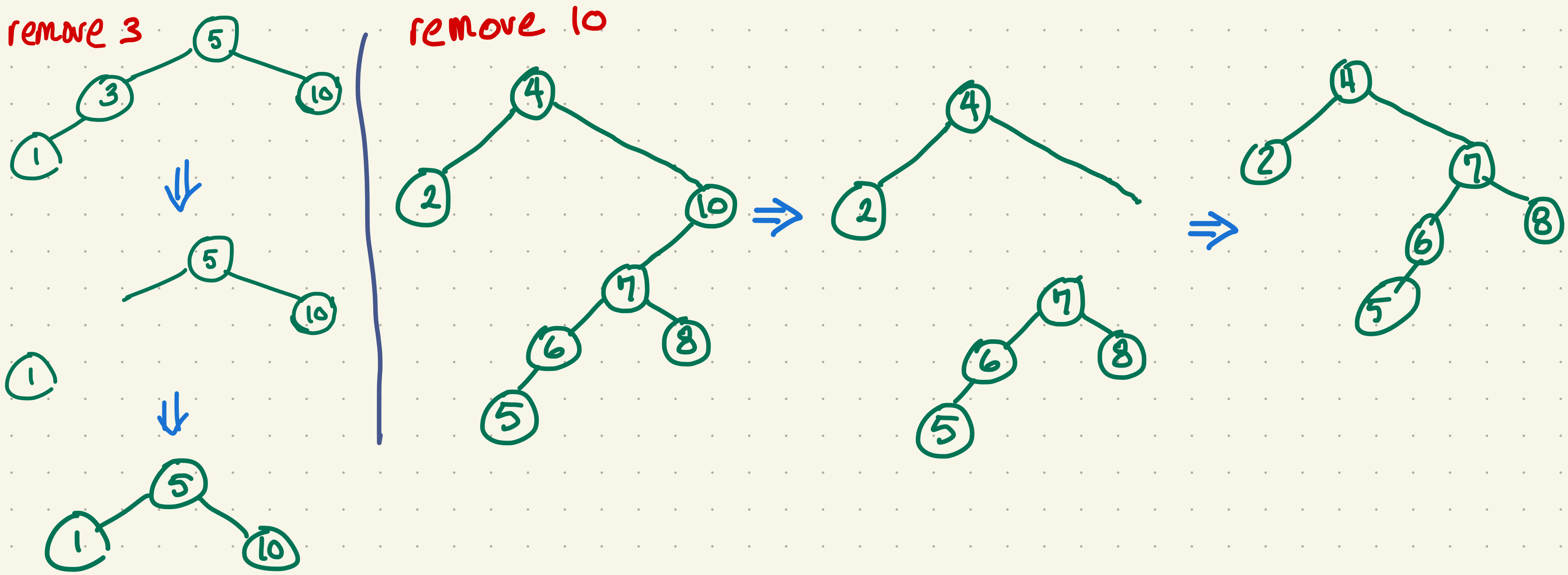


BST remove(t)

Case 2: t is at a node with 1 child

i) find the node v with $key(v) = t$
ii) let u be the child of v
iii) replace v with the subtree rooted at u.

Examples:

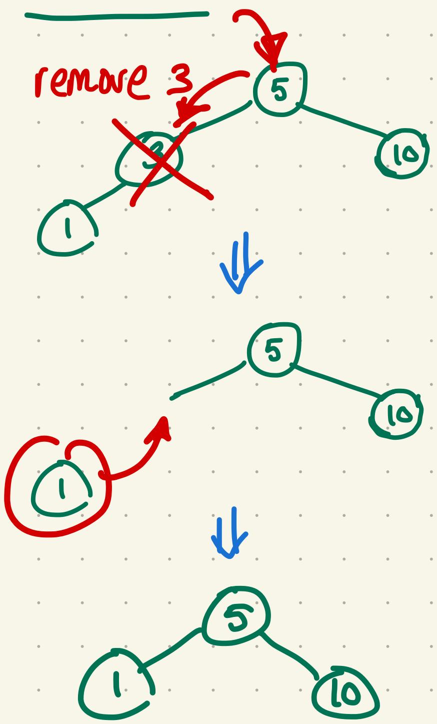


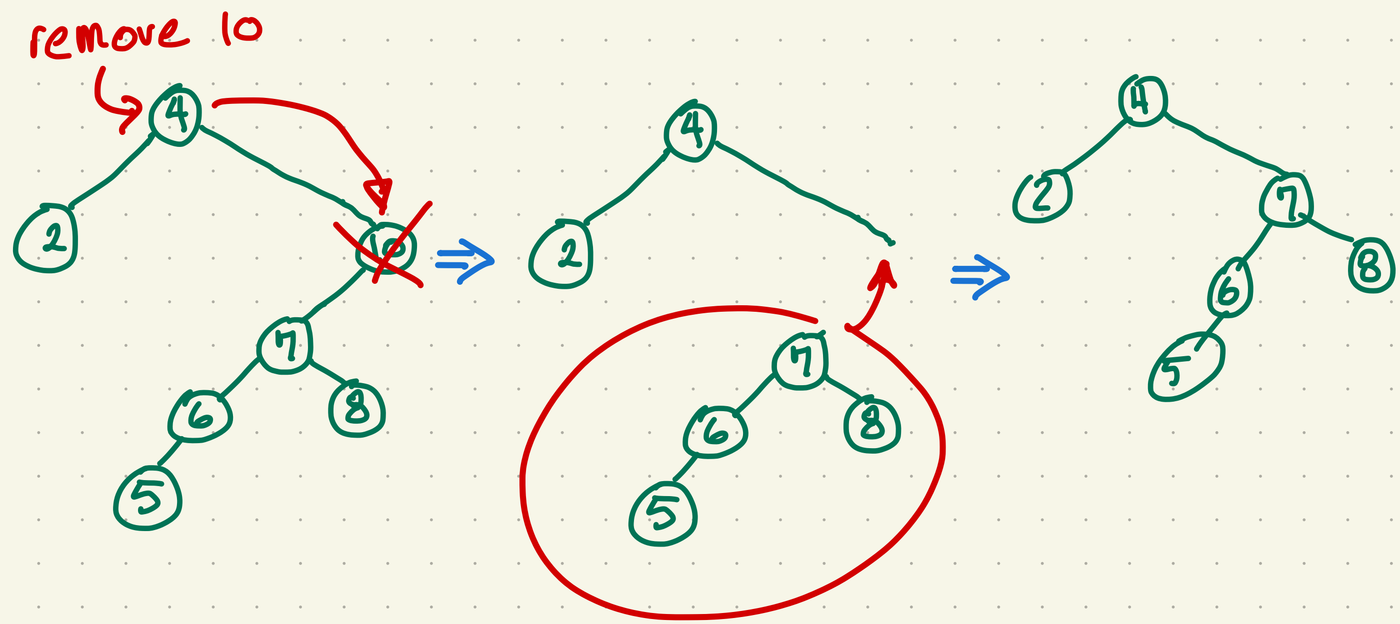
BST remove(t)

Case 2: t is at a node with 1 child

i) find the node v with key(v) = t
ii) let u be the child of v
iii) replace v with the subtree rooted at u.

Examples:





BST remove: Case 3 Preparation: Successors

- In an ordered collection $X = \langle \ldots s_{i-1}, s_i, s_{i+1}, s_{i+2} \ldots \rangle$

  s_{i-1} is the predecessor of s_i

  s_{i+1} is the successor of s_i

  Write $succ_X(s_i) = s_{i+1}$

- Let $V = \langle v_1, \ldots v_n \rangle$ be the nodes of the tree ordered as per an in-order traversal.

- Let $K = \langle k_1, \ldots, k_n \rangle$ be the keys, in non-decreasing order.

- Then: $y = key(u) \Rightarrow succ_K(y) = key(succ_V(u))$

  i.e., the next node has the next key.

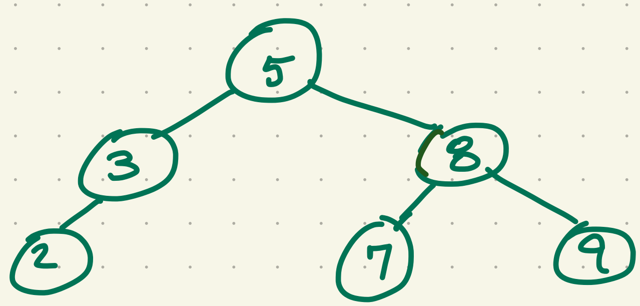
BST remove: Case 3 Preparation: Successors in BSTs

- If S is a set of keys, and $x \in S$, then the successor of x in S is the smallest value $y \in S$ s.t. $x < y$.

Ex: $S = \{19, 27, 8, 3, 12\}$, $succ(8) = 12$, $succ(12) = 19$, ...

$(S = \{3, 8, 12, 19, 27\})$

- In a BST, in-order traversal visits keys in order.
  Let S be the set of keys in BST T.
  If v is a node of T, and $key(v) = x$, then $succ(x)$, the successor of x in S, is $key(u)$ where u is the node of T that an in-order traversal of T visits next after v.



BST remove: Case 3 Preparation: Successors in BSTs

- If v is a node of BST T, we can say the successor of v in T is the node of T visited just after v by an in-order traversal of T.

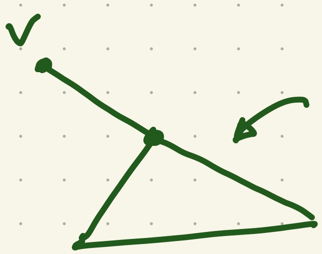
  Then: $$succ(x) = key(succ(node(x)))$$

- Or: If $key(v) = x$, we can find the successor of x by finding the successor node of v, and getting it's key:

$$\underline{succ(key(v)) = key(succ(v))}$$

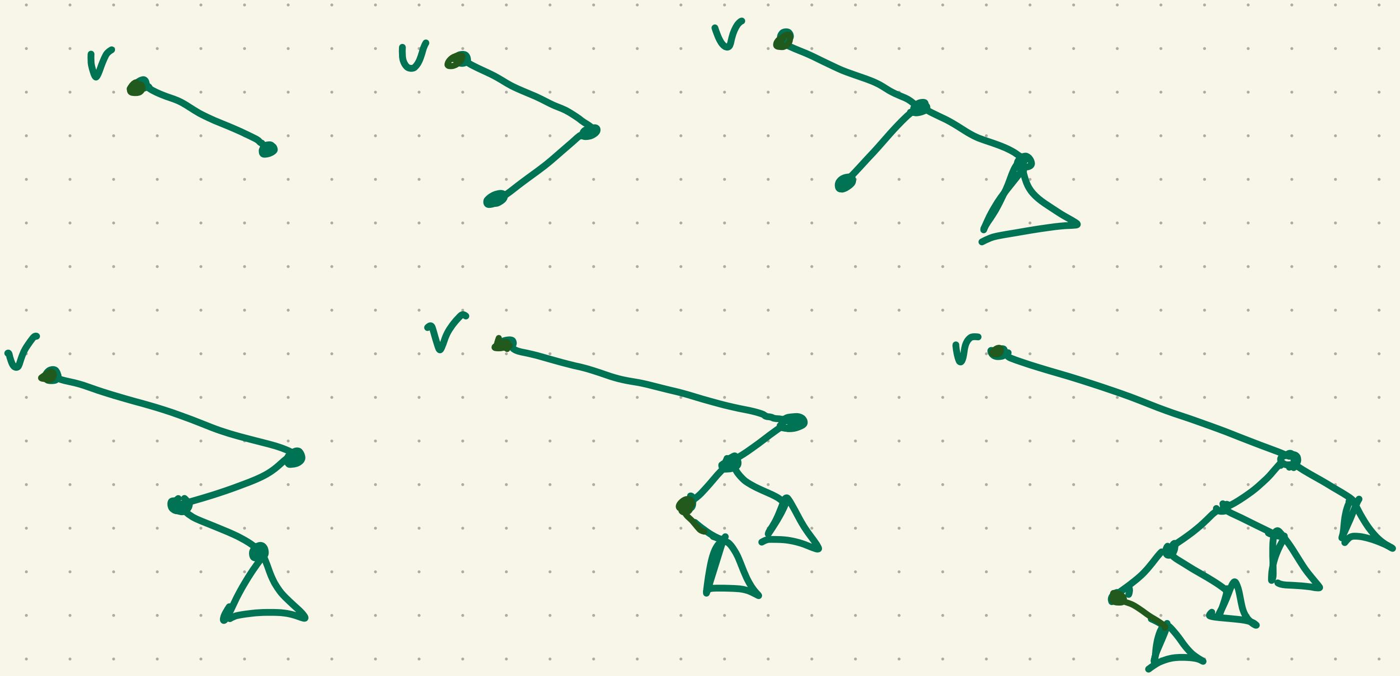
BST remove: Case 3 Preparation: Successors.

- If node v has a right child, it is easy to find its successor:



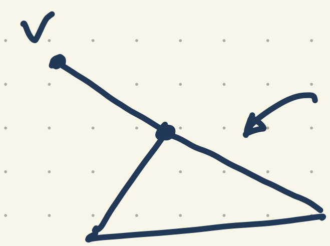
succ(v) is the first node visited by an in-order traversal of the right subtree of v

Ex:



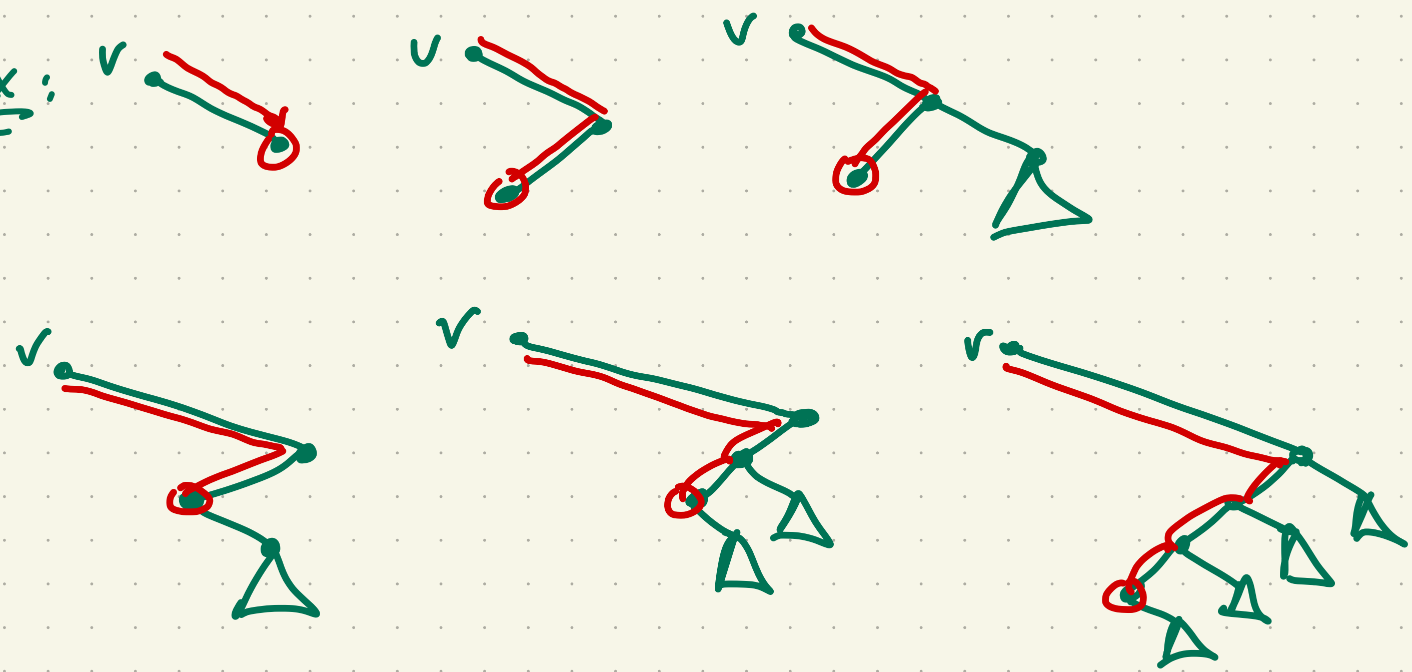
BST remove: Case 3 Preparation: Successors.

- If node v has a right child, it is easy to find its successor:



succ(v) is the first node visited by an in-order traversal of the right subtree of v.

Ex:



BST remove: Case 3 preparation: Successors

- To find the successor of node v that has a right child, use:

```
succ(v){
    u ← right(v)
    while(left(u) exists){
        u ← left(u)
    }
    return u
}
```

BST remove(t)

Case 3: t is at a node with 2 children

i) find the node v with $key(v) = t$
ii) find the successor of v – call it u.
iii) $key(v) \leftarrow key(u)$ // replace t with succ(t) at v.
iv) delete u:

a) if u is a leaf, delete it.
b) if u is not a leaf, it has one child w, replace u with the subtree rooted at w.

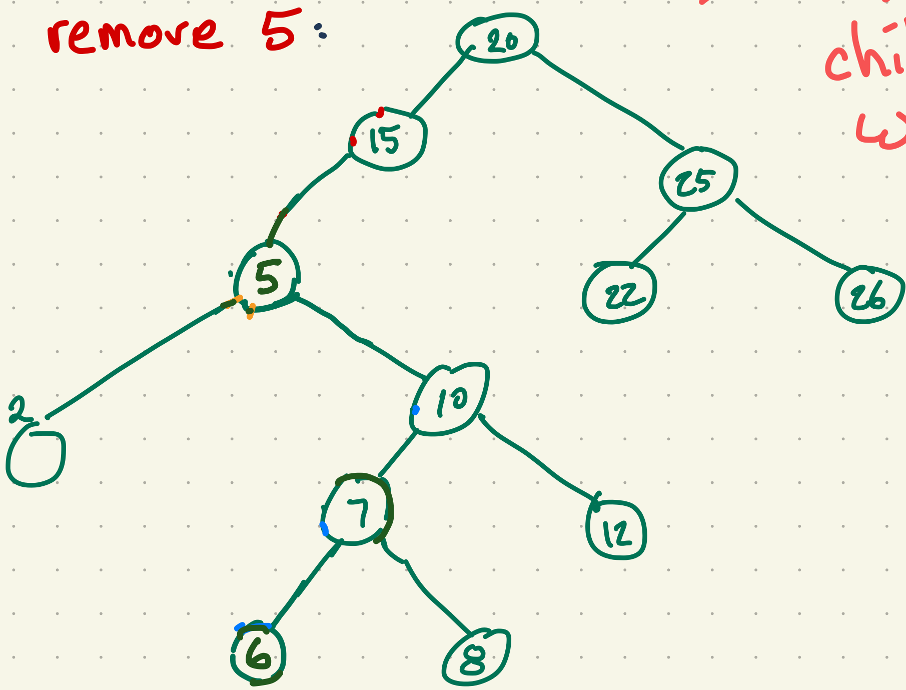
Notice: iv(a) is like case 1
iv (b) is like case 2

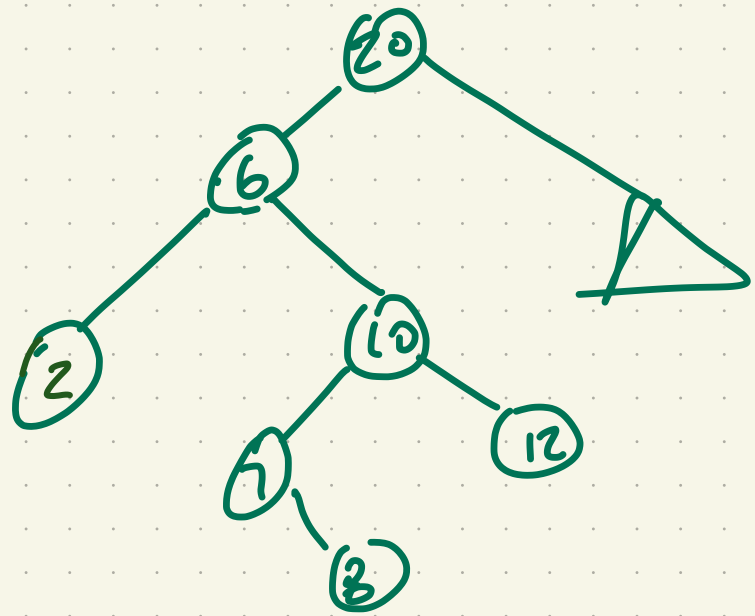
BST remove(k) when node(k) has 2 children

Ex. To remove 5:

1) Find 5
2) Find successor of 5
3) Replace 5 with its succ.
4) In this example, succ(5) has no children so just delete the node where it was.

remove 5:



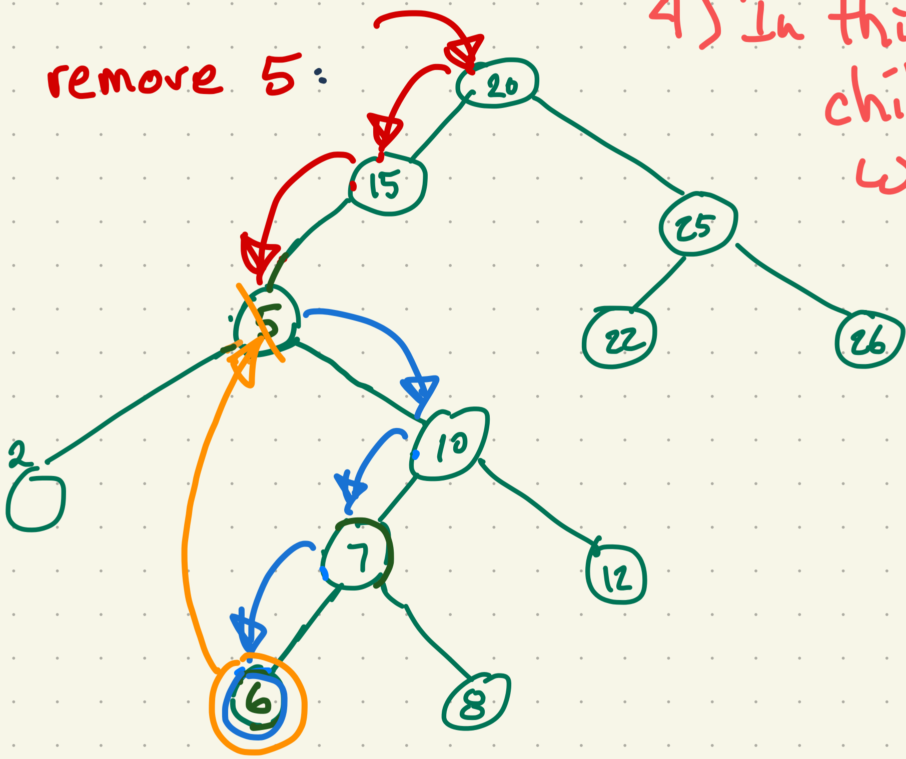


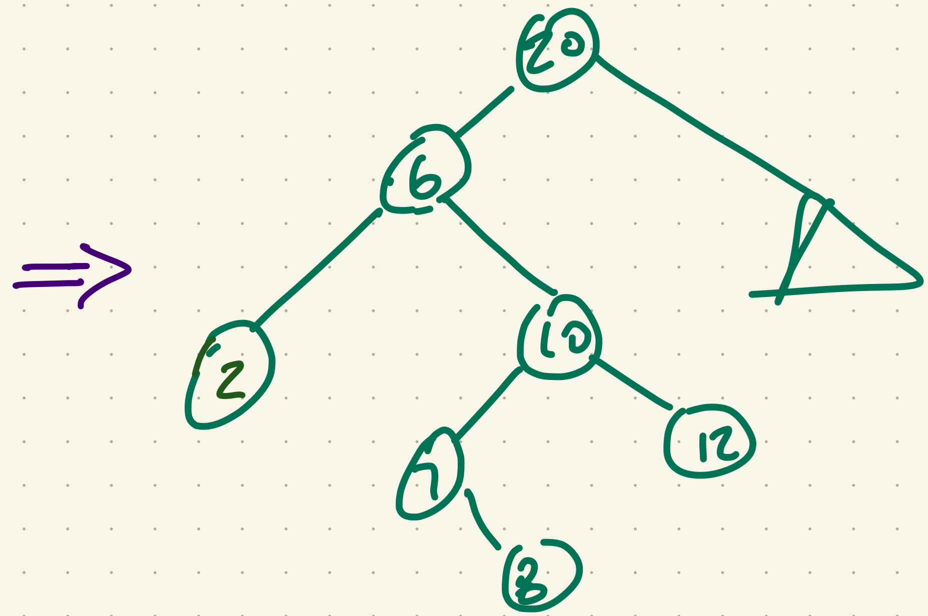
BST remove(k) when node(k) has 2 children

Ex. To remove 5:

1) Find 5
2) Find successor of 5
3) Replace 5 with its succ.
4) In this example, succ(5) has no children so just delete the node where it was.

remove 5:

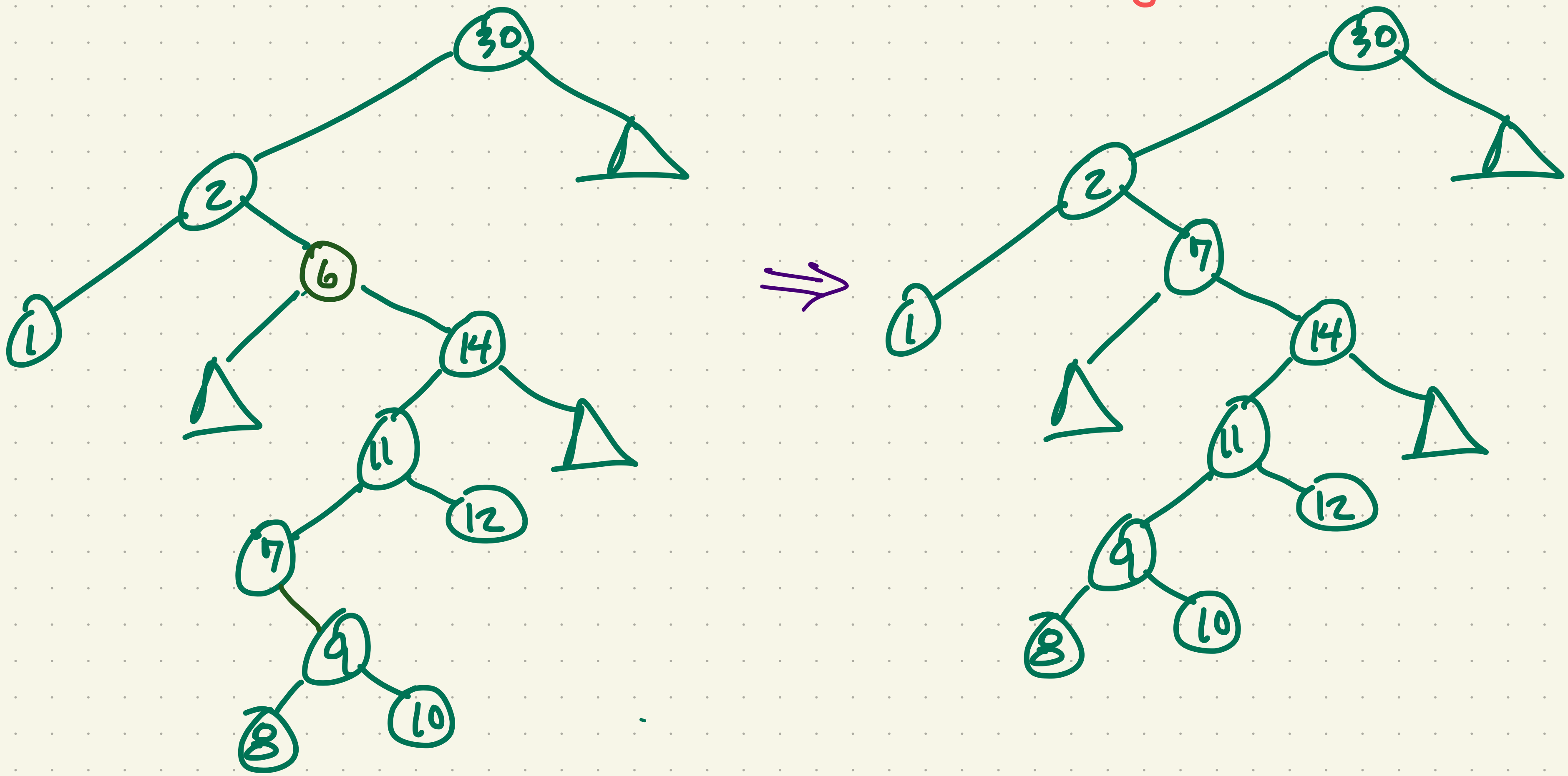




BST remove(k) when node(k) has 2 children

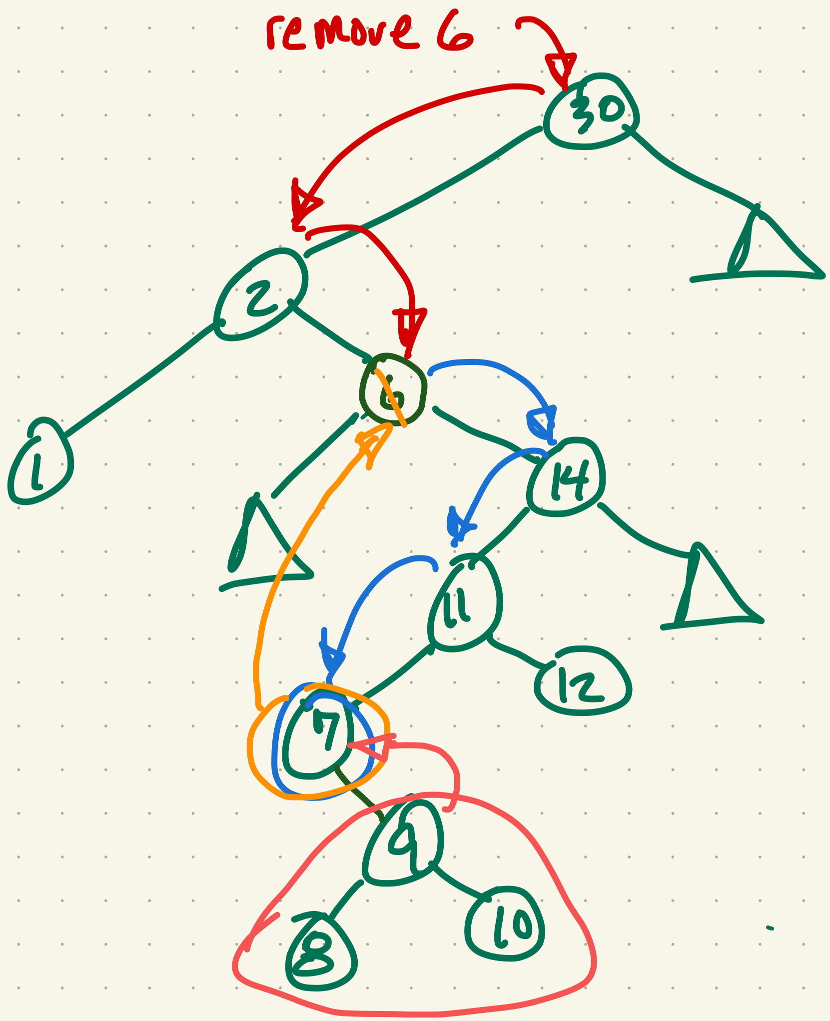
To remove 6:
1) Find 6
2) Find successor of 6
3) Replace 6 with its successor
4) Replace succ(6) with its non-empty subtree.

remove 6

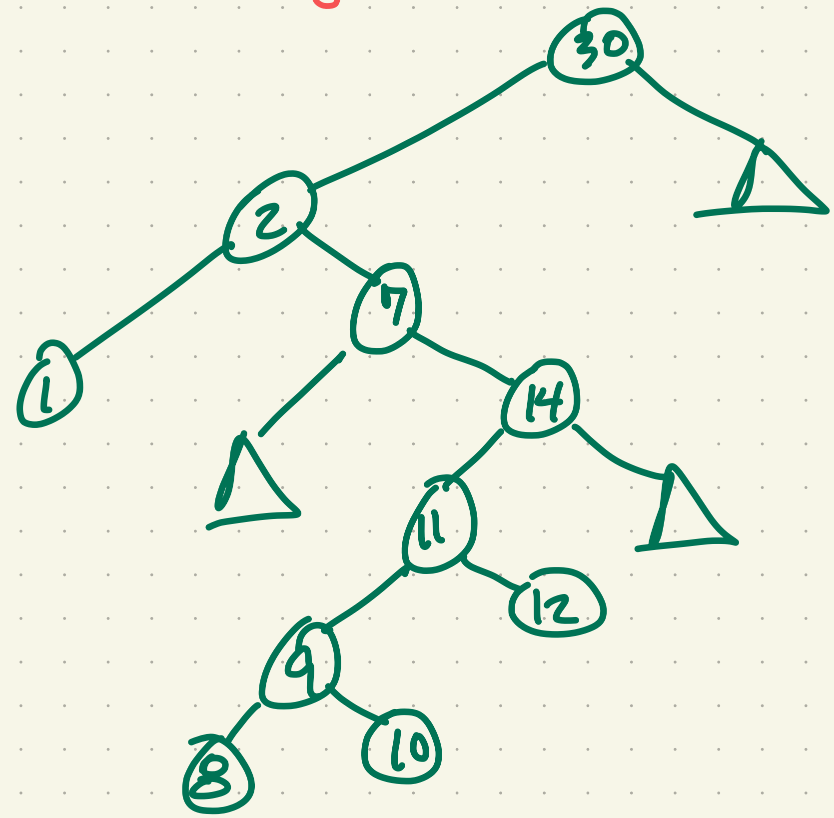


BST remove(k) when node(k) has 2 children

To remove 6:
1) Find 6
2) Find successor of 6
3) Replace 6 with its successor
4) Replace succ(6) with its non-empty subtree.



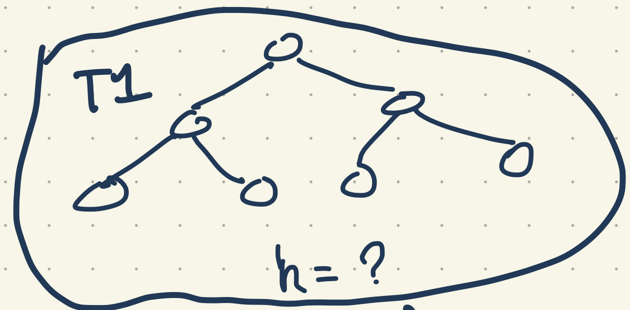
⇒



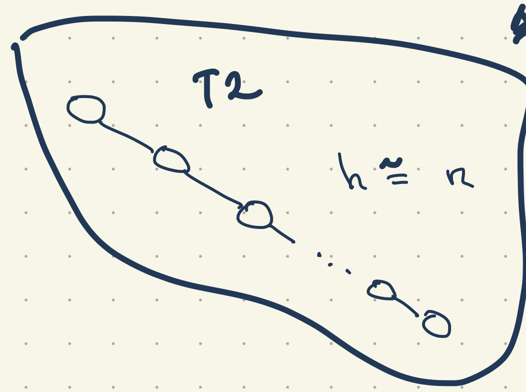
Complexity of BST Operations

\# keys

- Measure as a function of: height (h) or size (n).
- All operations essentially involve traversing a path from the root to a node v, where in the worst case v is a leaf of maximum depth.
- So: find: $O(h)$, $O(n)$
  insert: $O(h)$, $O(n)$
  remove: $O(h)$, $O(n)$



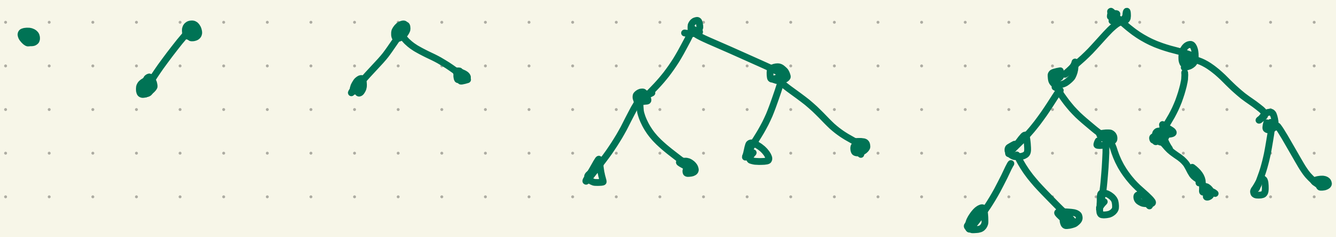
- For "short bushy" trees (e.g. T1) h is small relative to n.
- For "tall skinny" trees (e.g. T2) h is proportional to n.

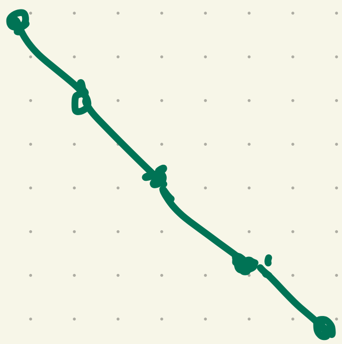


Q: Can we always have short bushy BSTs?

Perfect Binary Trees

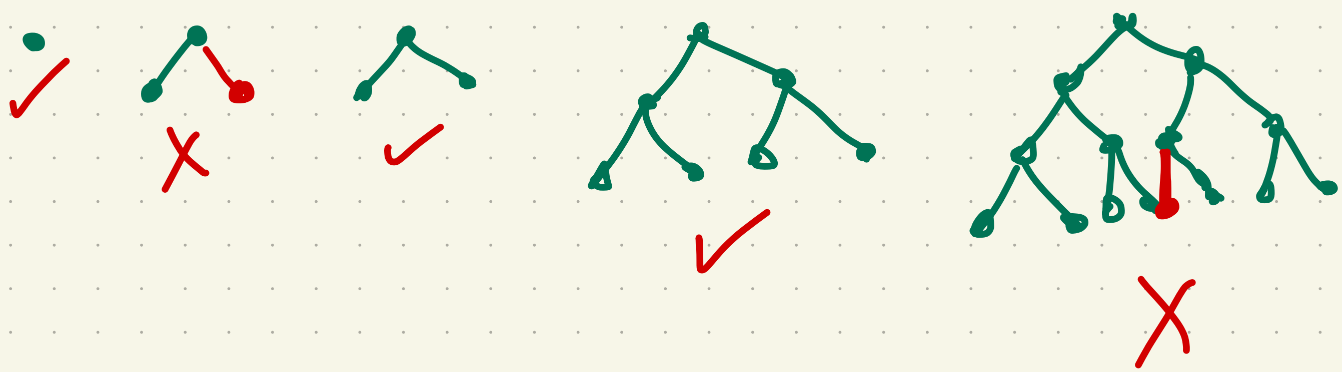
- A <u>perfect binary tree</u> of height h is a binary tree of height h with the max. number of nodes:

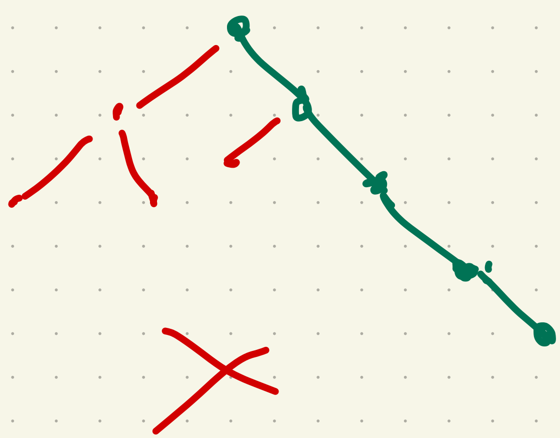




Perfect Binary Trees

- A perfect binary tree of height h is a binary tree of height h with the max. number of nodes:





Claim: Every perfect binary tree of height h has $2^{h+1}-1$ nodes.

Pf: By induction on h, or on the structure of the tree.

Basis: If $h=0$, there is one node (the root).

We have $2^{h+1}-1 = 2^1-1 = 1$ as required.

I.H.: Let $k \geq 0$, and assume that every perfect binary tree of height k has $2^{k+1}-1$ nodes.

I.S.: (Need to show a p.b.t. of height $k+1$ has $2^{(k+1)+1}-1$ nodes)

A perfect binary tree of height $k+1$ is constructed as:

where A, B are perfect binary trees of height k.

By I.H. they have $2^{k+1}-1$ nodes.

So, the tree has $2^{k+1}-1 + 2^{k+1}-1+1$

$$= 2 \cdot 2^{k+1} - 1$$

$$= 2^{(k+1)+1} - 1,$$ as required.

Existence of Optimal BSTs

Claim: For every set S of n keys, there exists a BST for S with height at most $1+\log_2 n$

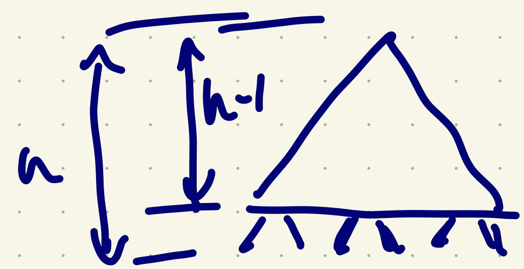
Proof: Let h be the smallest integer s.t. $2^h \geq n$, and let $m=2^h$.

So: $2^h \geq n > 2^{h-1}$

$$\log_2 2^h \geq \log_2 n > \log_2 2^{h-1}$$

$$h \geq \underbrace{\log_2 n > h-1}$$

$$h < 1+\log_2 n$$



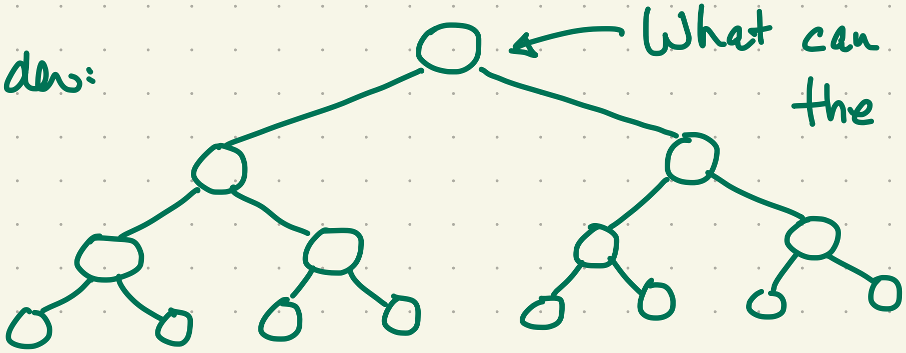
- let T be the perfect binary tree of height h
- label the first n nodes of T (as visited by an in-order traversal) with the keys of S, and delete the remaining nodes (to get T').
- T' is a BST for S with height $h < 1+\log_2 n$

- So, there is always a BST with height $O(\log n)$.

Optimal BST Insertion Order

Given a set of keys, we can insert them so as to get a minimum height BST:

Consider:



What can we say about the key here?

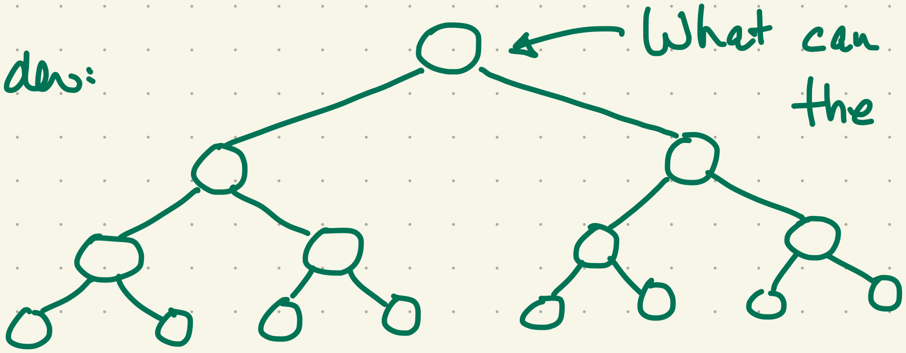
Observe: The first key inserted into a BST is at the root forever (unless we remove it from the BST)

$\Rightarrow$

Optimal BST Insertion Order

Given a set of keys, we can insert them so as to get a minimum height BST:

Consider:



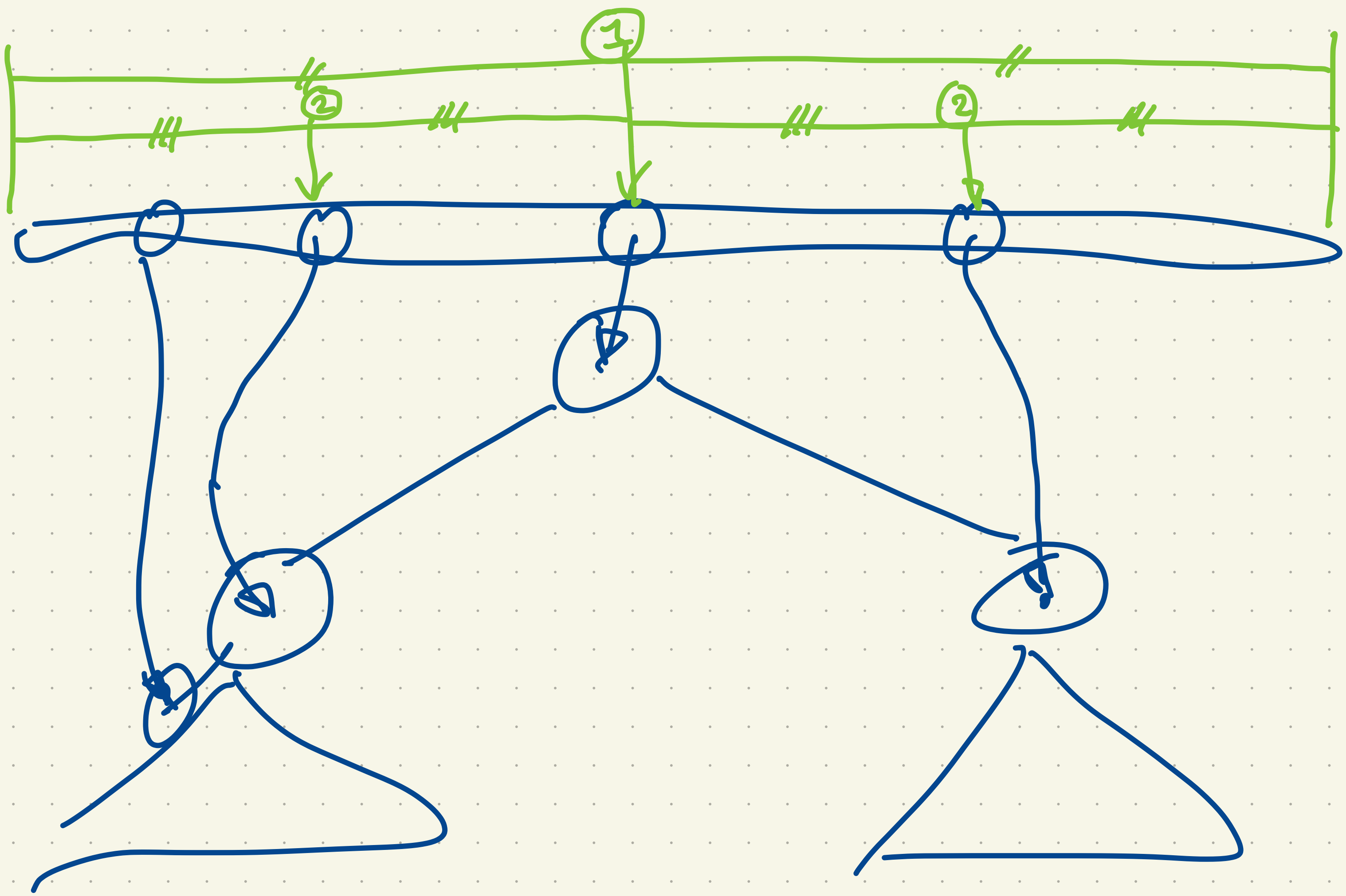
What can we say about the key here?

(It is the median key)

Observe: The first key inserted into a BST is at the root forever (unless we remove it from the BST)

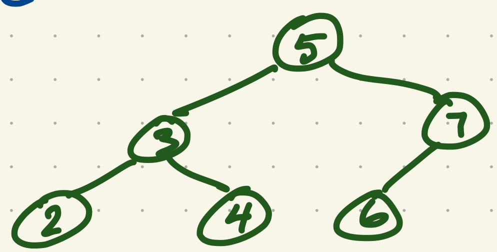
$\Longrightarrow$

Optimal BST Insertion Order:



* Apply the "root is the median key" principle to each sub-tree.

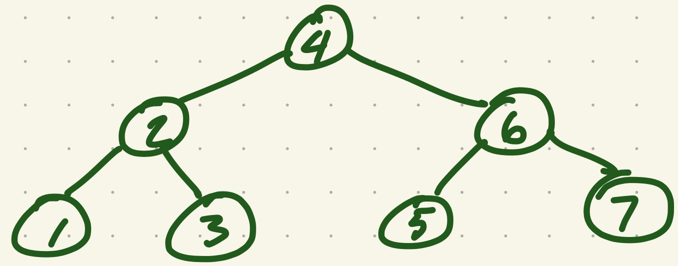
- So, there is always a BST with height $\sim \log n$
- Can we <u>maintain</u> min. height with $O(\log n)$ as we insert & remove keys?
- Consider:



A

<u>insert 1</u>

$\Rightarrow$



B

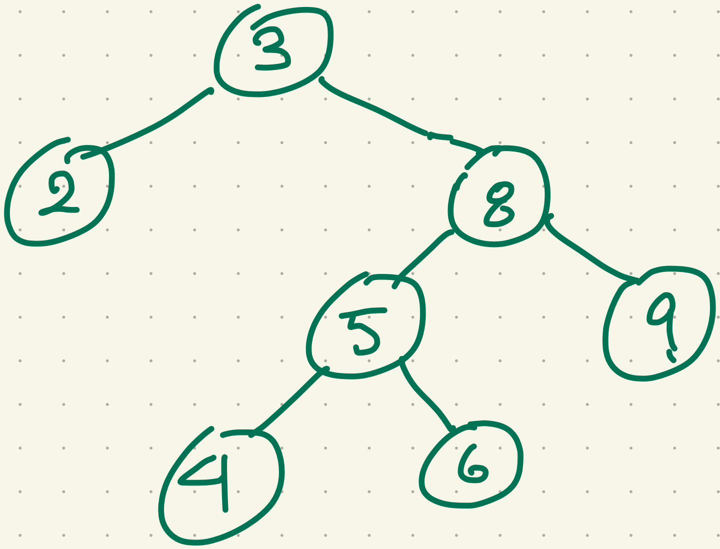
- B is the <u>only</u> min height BST for 1..7.
- $A \to B$ required "moving every node"
- To get $O(\log n)$ operations, we need another kind of search tree, other than plain BSTs.

- To get efficient search trees, give up at least one of:
  - binary
  - min. height
- Next: Self-balancing search trees.

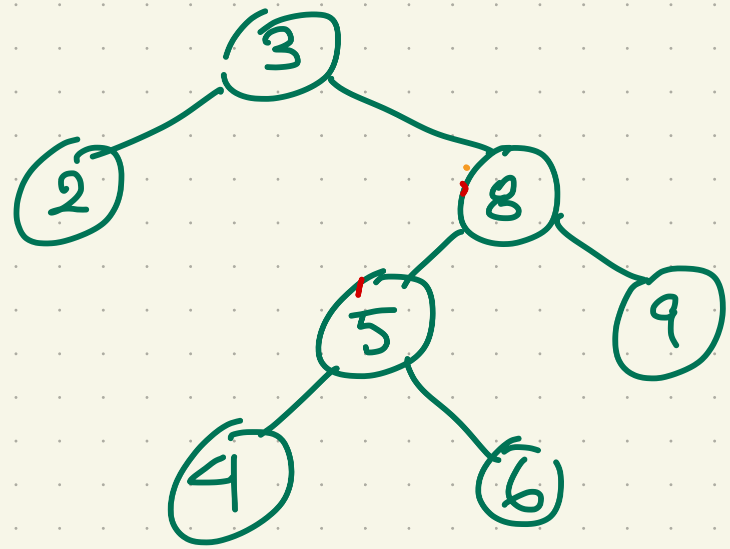
End

BST member/find: examples

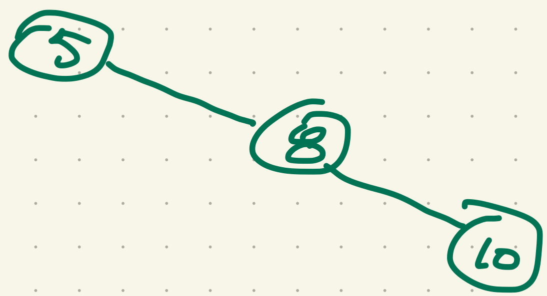
find(5)



find(7)

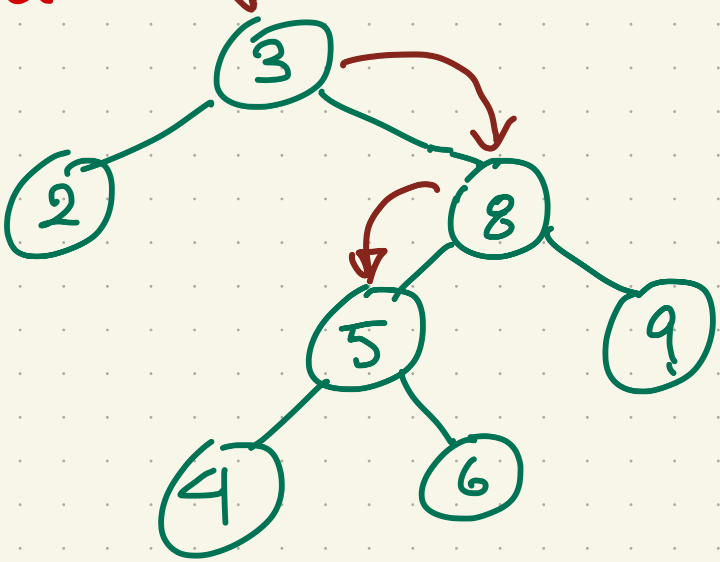


find 6

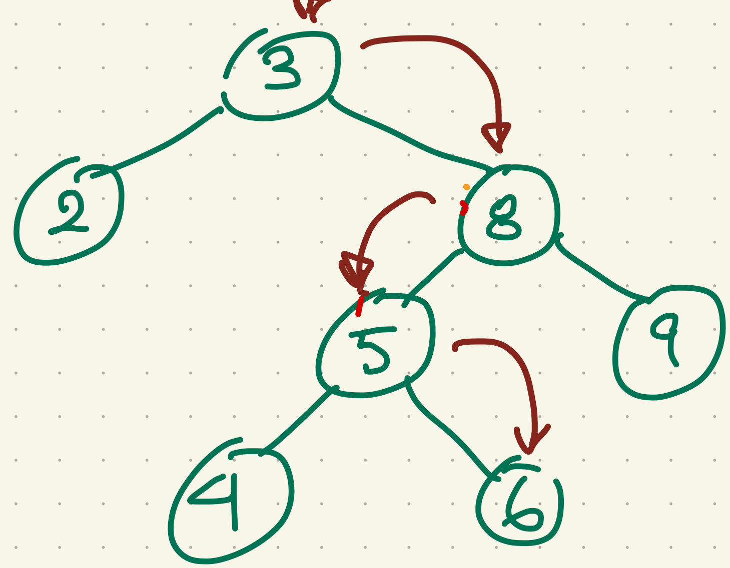


BST Find: Chooses sub-trees

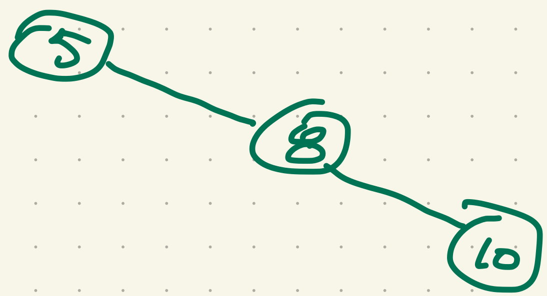
find(5)



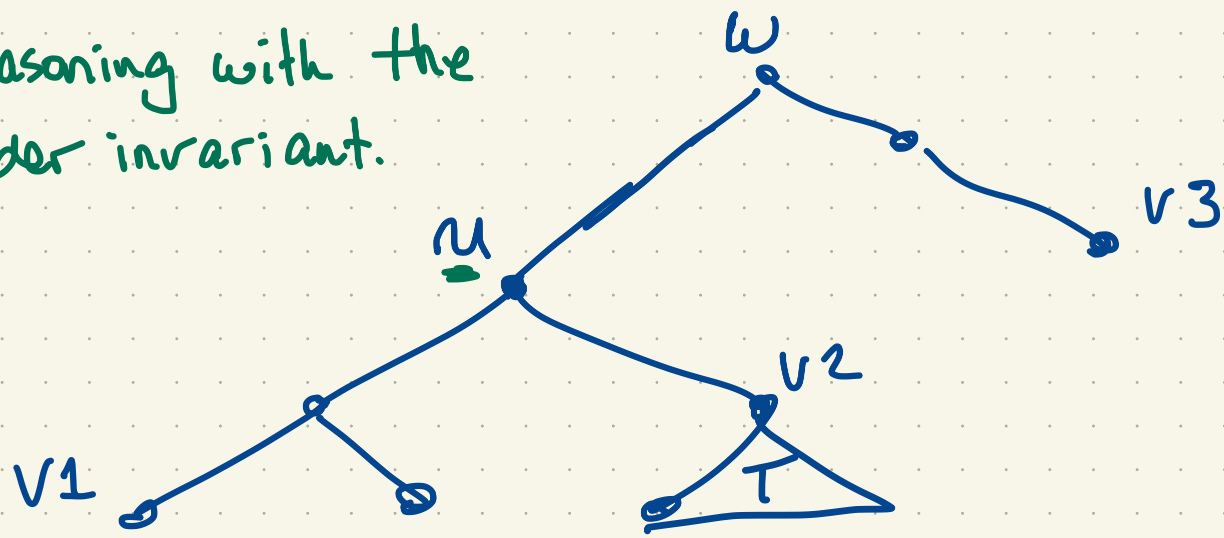
find(7)



find 6



Ex: Reasoning with the order invariant.



$$key(v1) < key(u)$$

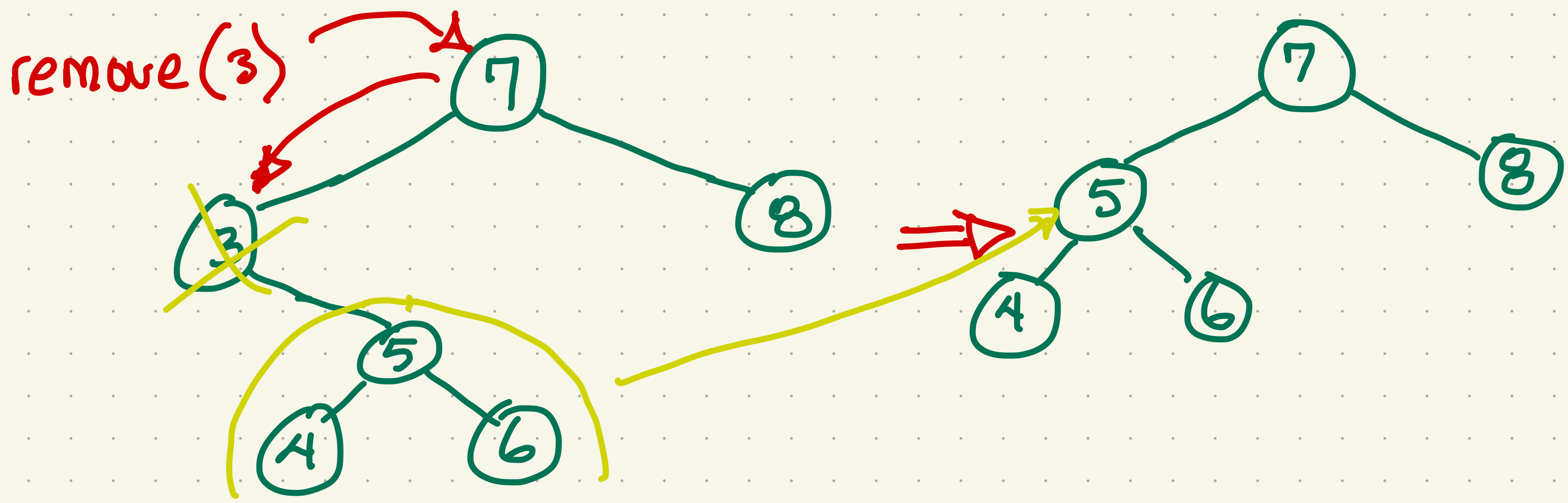
$$key(v2) > key(u)$$

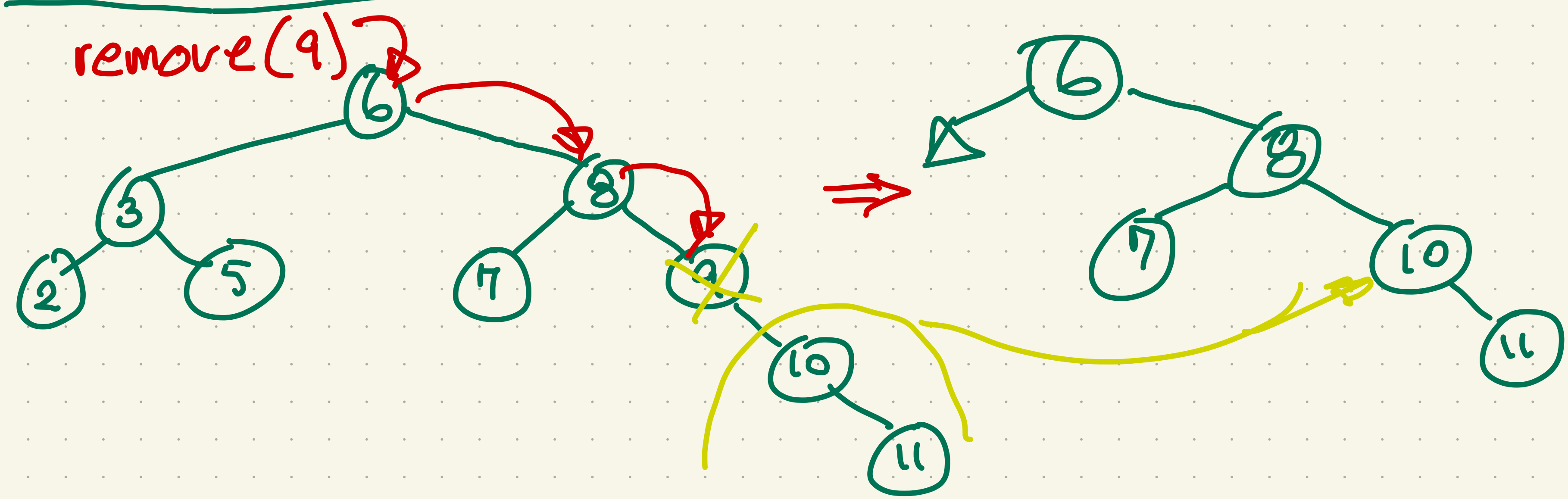
$$key(v3) \ ? \ key(u)$$

$$key(u) < key(w) < key(v3)$$

$$\Rightarrow key(u) < key(v3)$$

Example: BST remove(t) where node(t) has 1 child





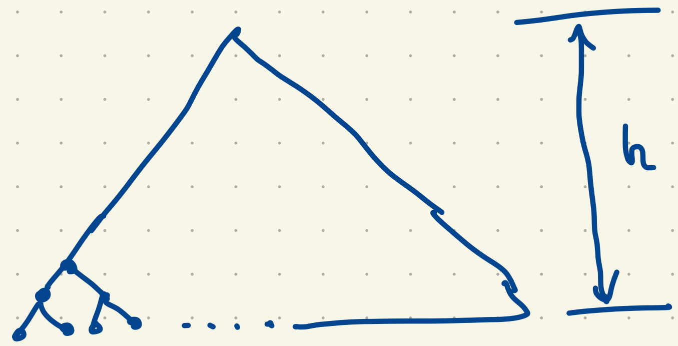
Notice:

Because a perfect binary tree of height h is like this:

$2^{h+1}-1$ nodes

$2^h - 1$ internal nodes

2^h leaves



$$2^h + 2^h - 1 = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$