- SEARCH
- INSERT
- DELETE

The 3 main operations that we perform on data structures

How fast can we do these?

	Array	Linked List	Hash Table expected / worst-case
SEARCH	$O(n)$ by key $\rightarrow$ $O(\log n)$ if sorted $O(1)$ by index	$O(n)$	$O(1) \rightarrow O(n)$
INSERT	$O(n)$	$O(1) \rightarrow O(n)$ if sorted	$O(1) \rightarrow O(n)$
DELETE	$O(n)$	$O(n)$	$O(1) \rightarrow O(n)$

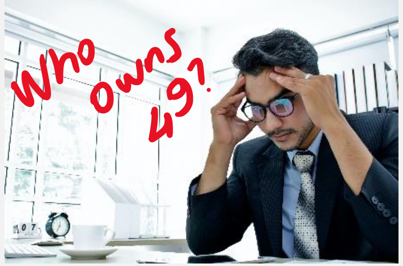
Is Hash Table always used?

Example: Let's store memory address intevals in a hash table and find corresponding users.

intervals		userids
(0,4)	→	ff11
(6,7)	→	0e54
(10,16)	→	fr22
(20,22)	→	jrg94
(30,36)	→	rf33
(42,45)	→	adw58
(48,50)	→	rr44

Hash it up!

keys: intervals	values: userids
(0,4)	ff11
(10,16)	fr22
(6,7)	0e54
(48,50)	rr44
(30,36)	rf33
(20,22)	jrg94
(42,45)	adw58

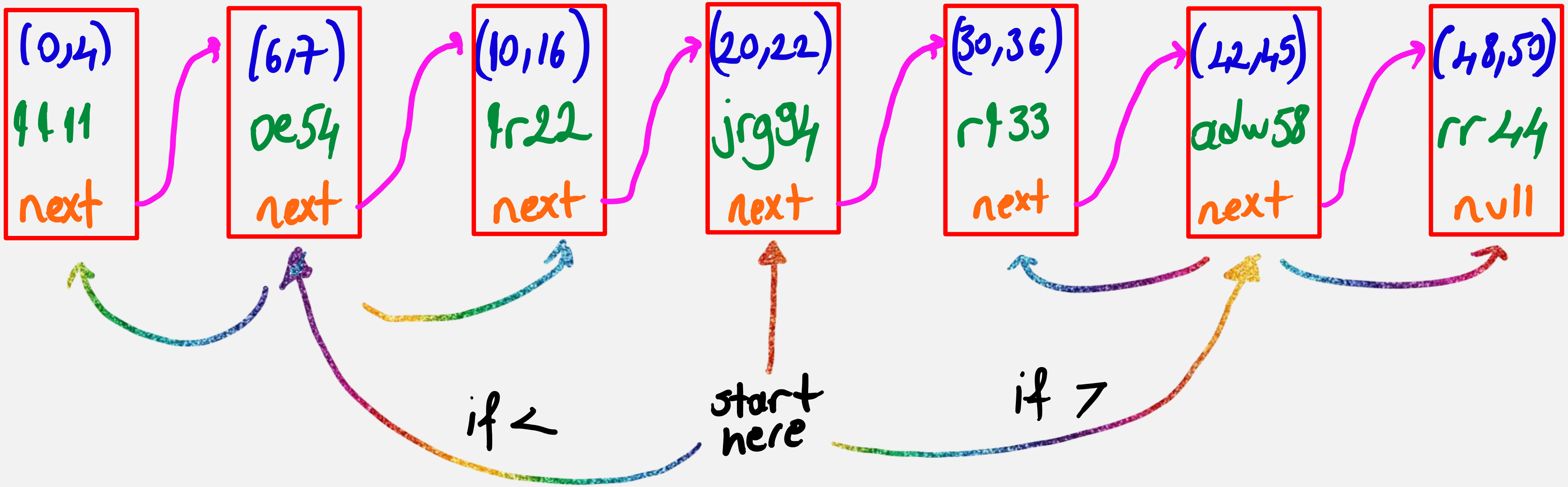


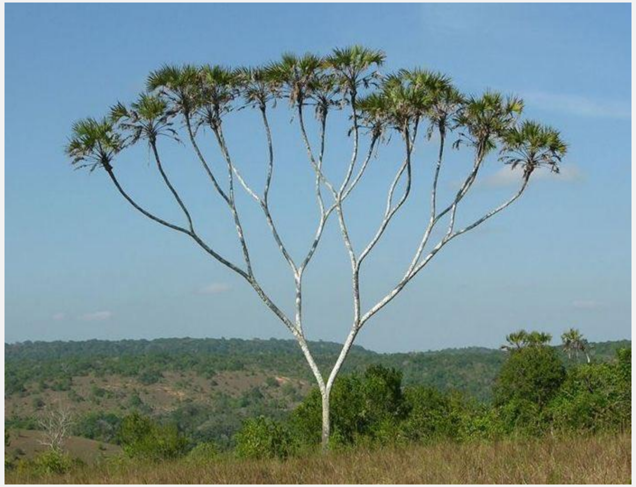
- SEARCH
- INSERT
- DELETE

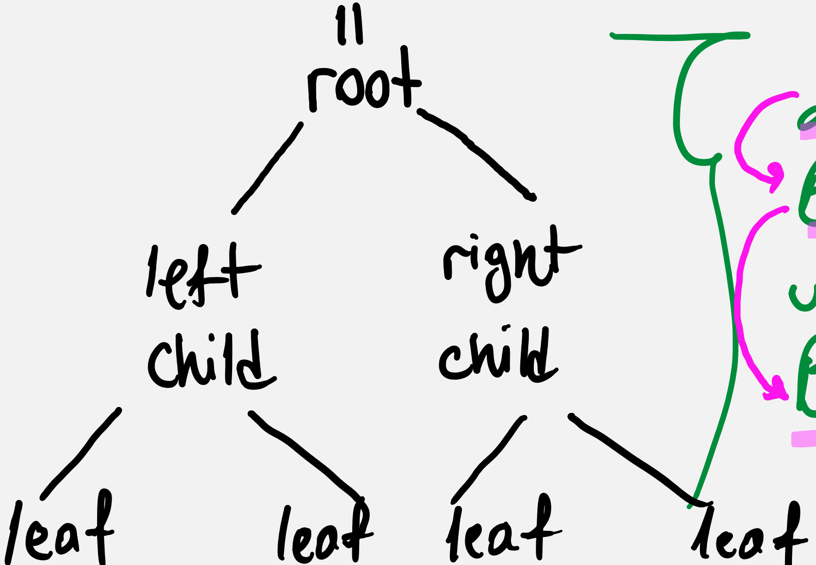
The 3 main operations that we perform on data structures

How fast can we do these?

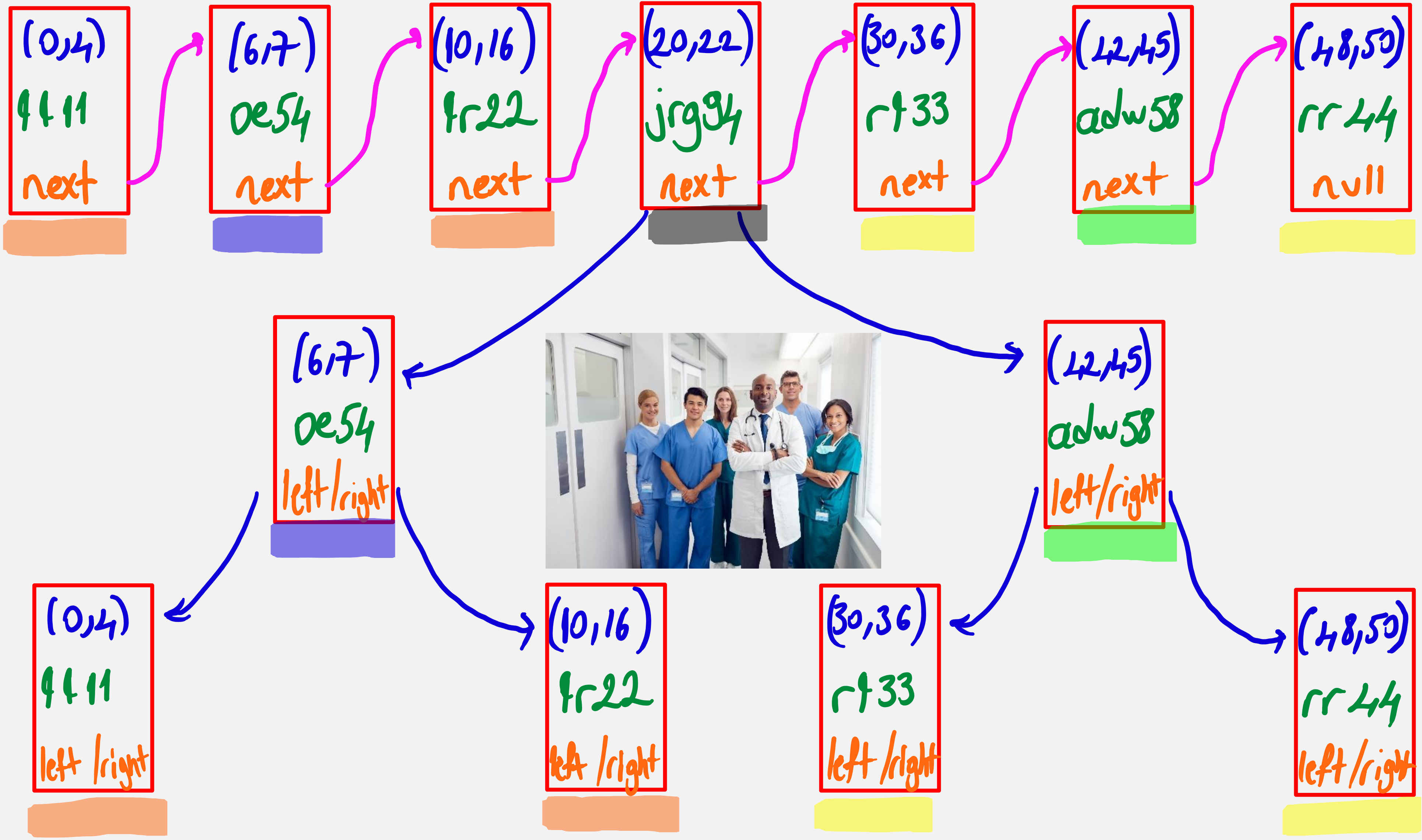
	Array	Linked List	Hash Table expected / worst-case
• SEARCH	$O(n) \rightarrow O(\log n)$ by key; if sorted $O(1)$ by index	$O(n)$	$O(1) \rightarrow O(n)$
• INSERT	$O(n)$	$O(1) \rightarrow O(n)$ if sorted	$O(1) \rightarrow O(n)$
• DELETE	$O(n)$	$O(n)$	$O(1) \rightarrow O(n)$



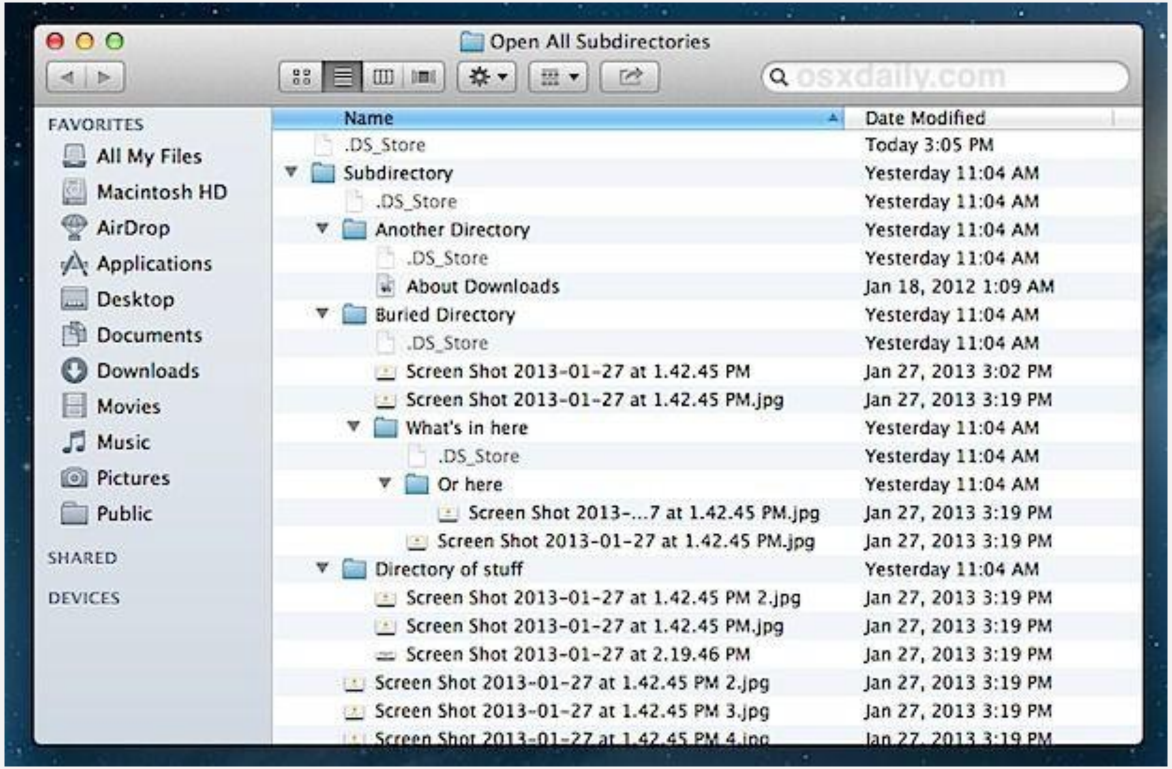




Linked List in a Binary Tree form which does Binary Search


(0,4)
4411
next
(6,7)
oe54
next
(10,16)
fr22
next
(20,22)
jrg94
next
(30,36)
rf33
next
(42,45)
adw58
next
(48,50)
rr44
null
(6,7)
oe54
left/right
(42,45)
adw58
left/right
(0,4)
4411
left/right
(10,16)
fr22
left/right
(30,36)
rf33
left/right
(48,50)
rr44
left/right

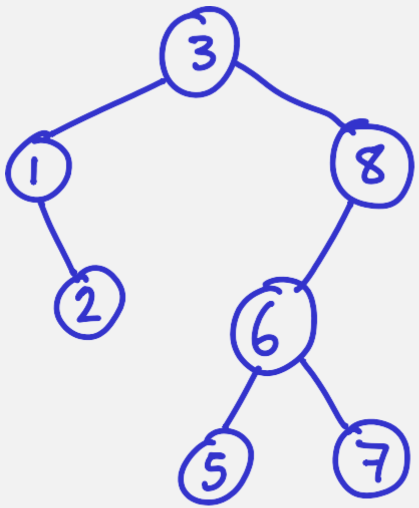
Trees: Filesystem

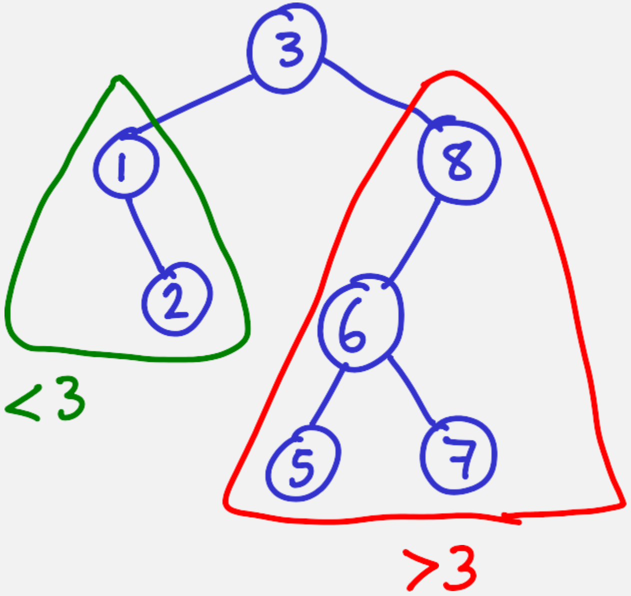


BINARY SEARCH TREES

BST data structure

- extension of BinaryTree
- invariant:
  - nodes in LEFT subtree are less than root
  - nodes in RIGHT subtree are greater than or equal to root

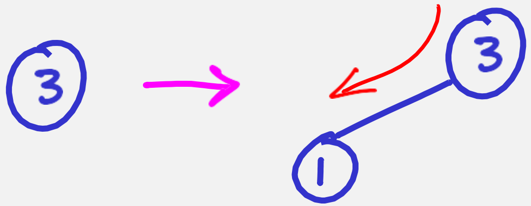




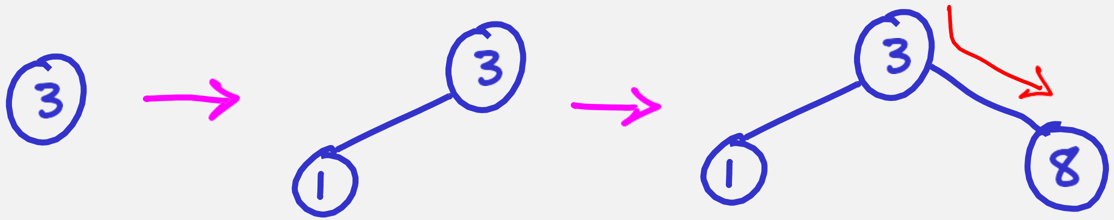
Given array of elements : 3 1 8 2 6 7 5

3

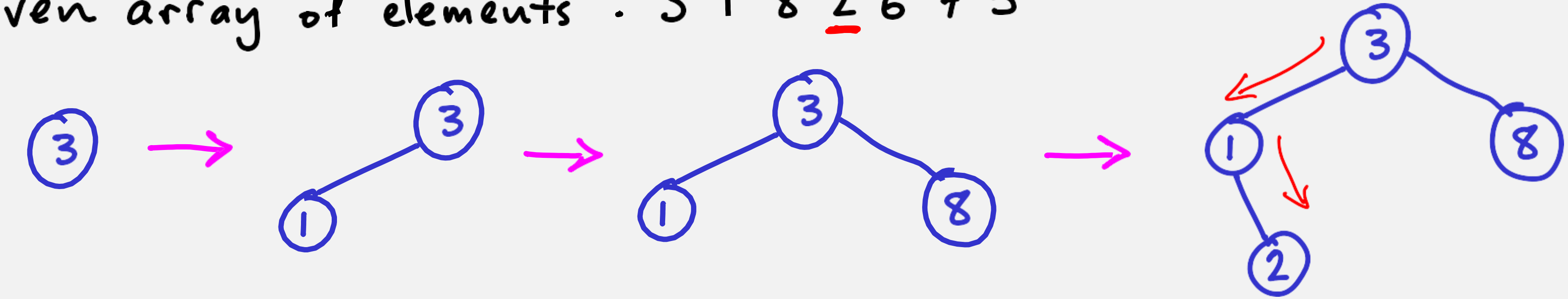
Given array of elements : 3 1 8 2 6 7 5



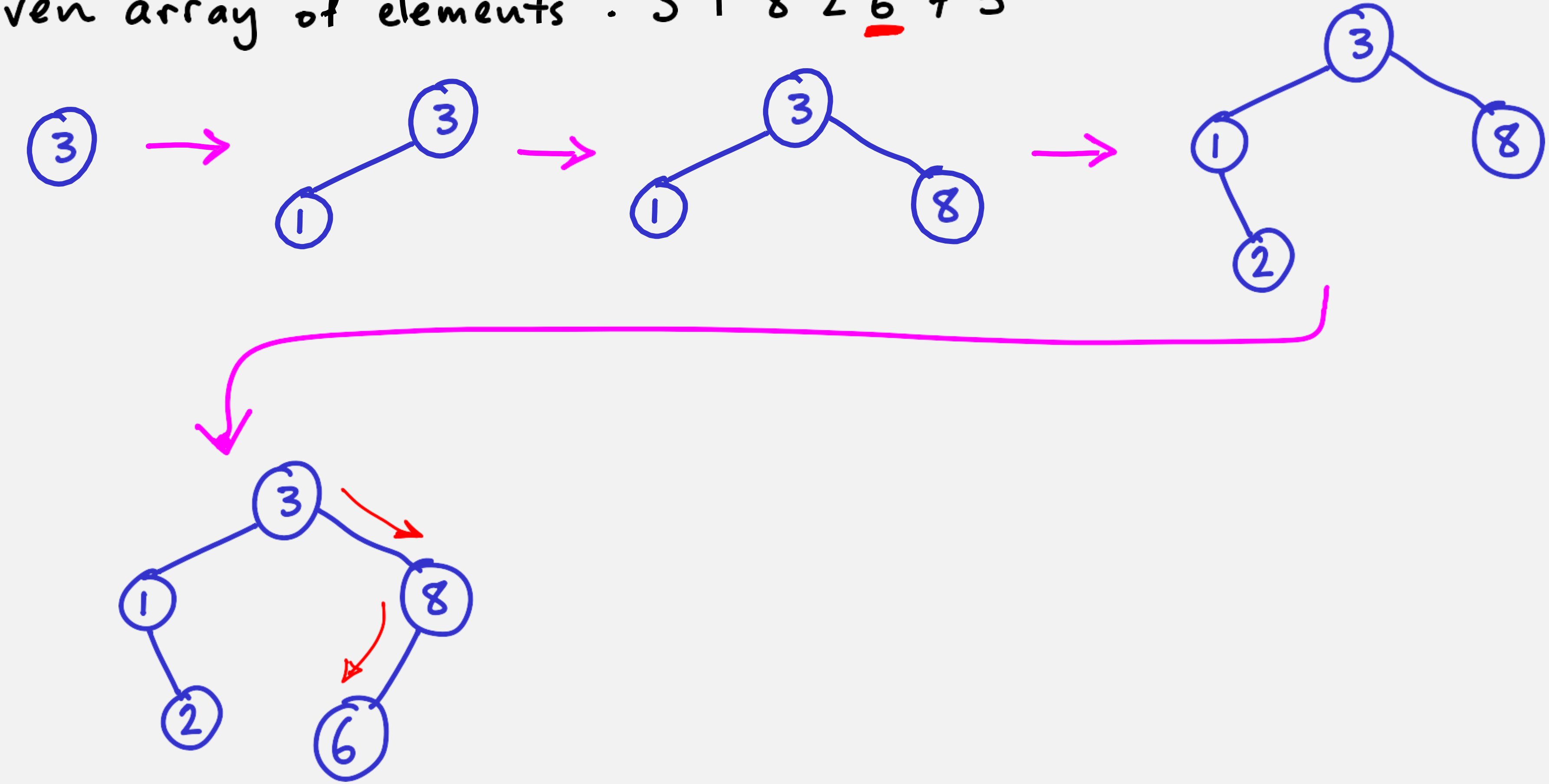
Given array of elements : 3 1 8 2 6 7 5



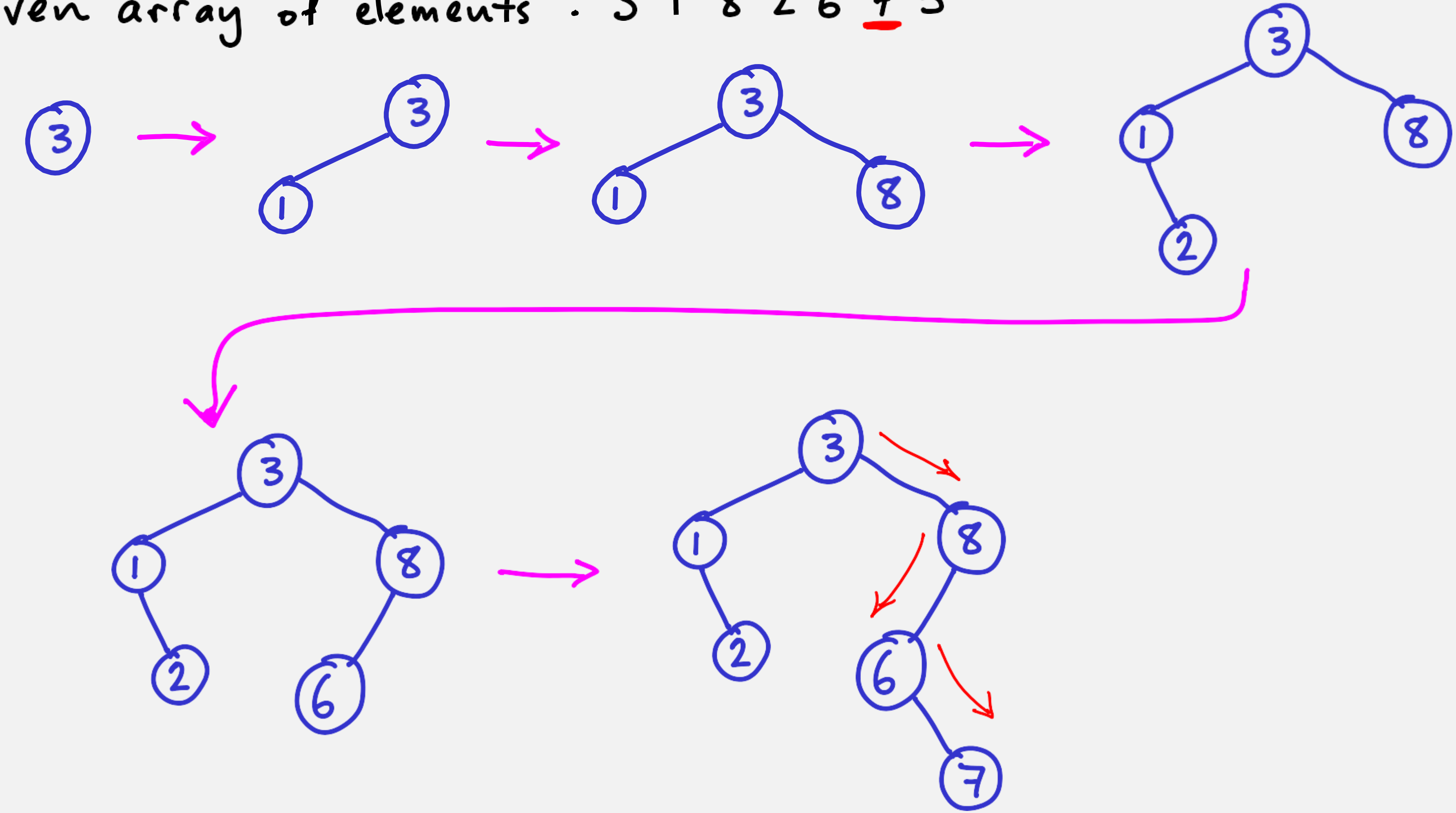
Given array of elements : 3 1 8 2 6 7 5



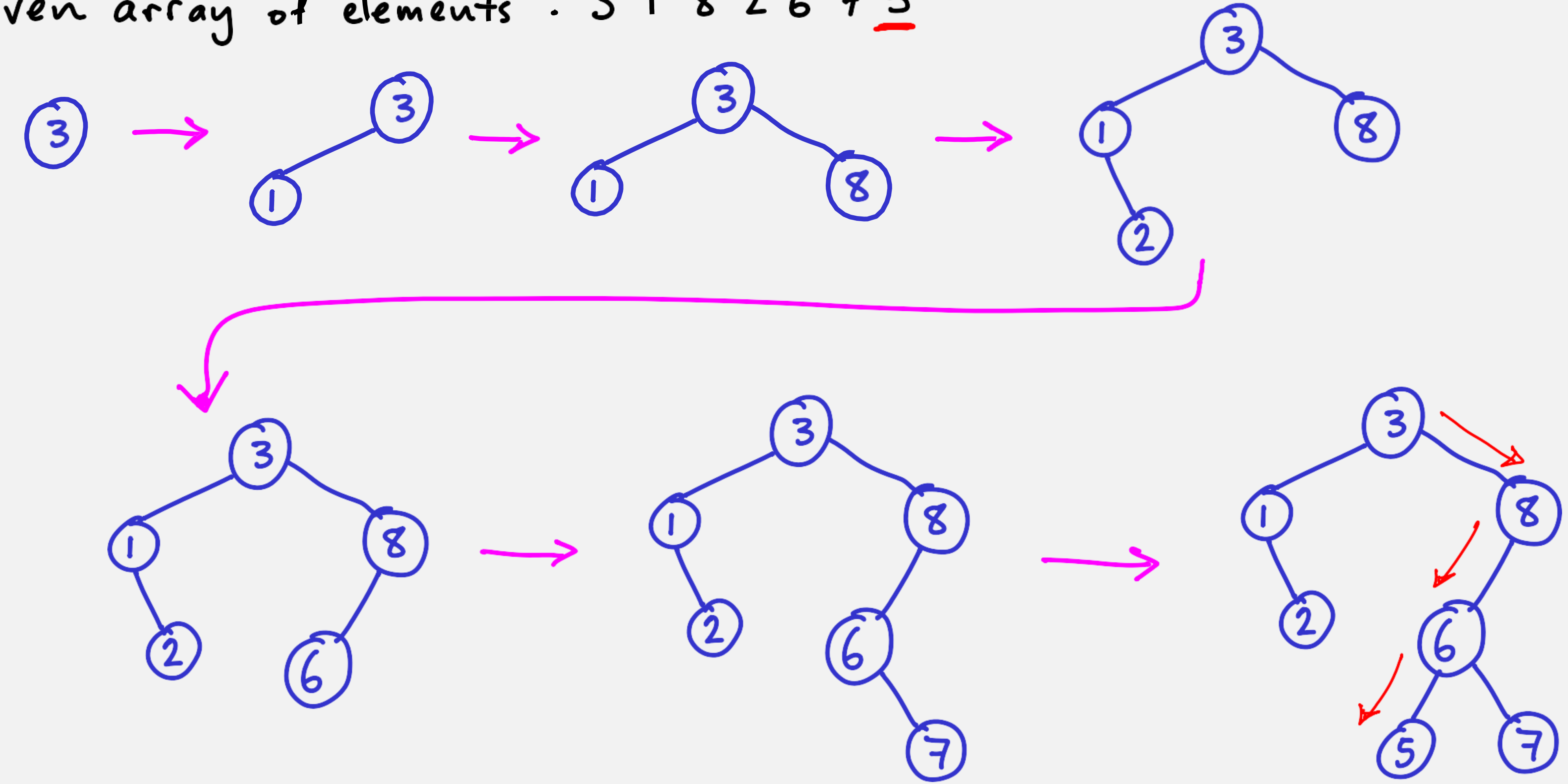
Given array of elements : 3 1 8 2 6 7 5



Given array of elements : 3 1 8 2 6 7 5

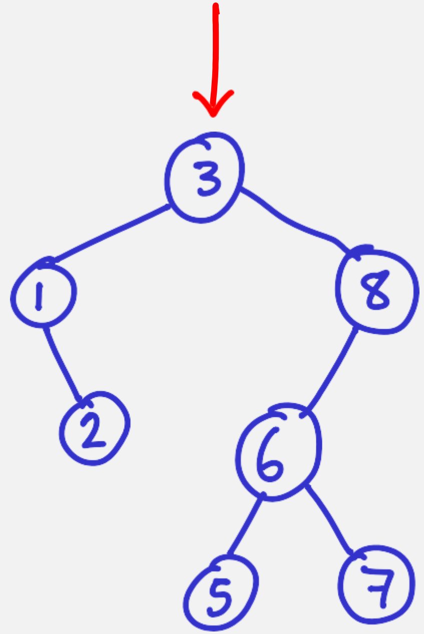


Given array of elements : 3 1 8 2 6 7 5



BINARY SEARCH TREES

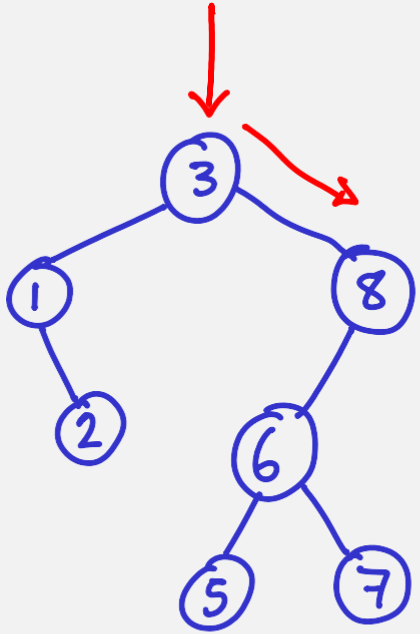
(binary) search for 5




```c
struct node {
int key;

struct node *left; /* left child */
struct node *right; /* right child */
};

#define NUM_CHILDREN (2)
struct node {
int key;
struct node *child[NUM_CHILDREN];
};
```

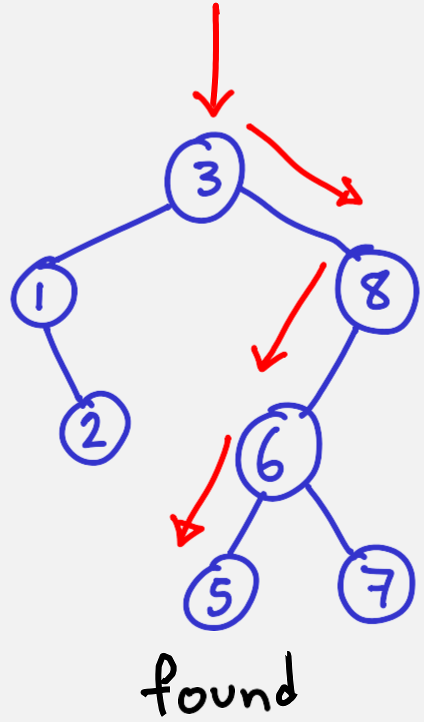
BINARY SEARCH TREES

(binary) search for 5



Binary search trees

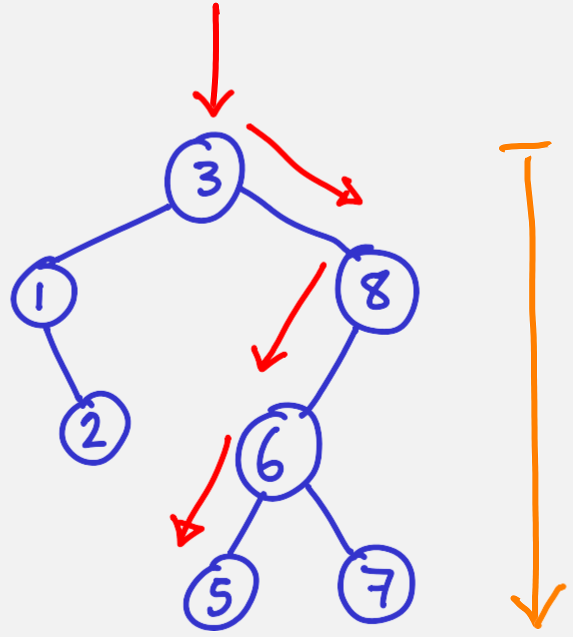
(binary) search for 5




```
/* returns pointer to node with given target key */
/* or 0 if no such node exists */
struct node *
treeSearch(struct node *root, int target)
{
        if(root == 0 || root->key == target) {
                return root;
        } else if(root->key > target) {
                return treeSearch(root->left, target);
        } else {
                return treeSearch(root->right, target);
        }
}
```

BINARY SEARCH TREES

(binary) search for 5



found

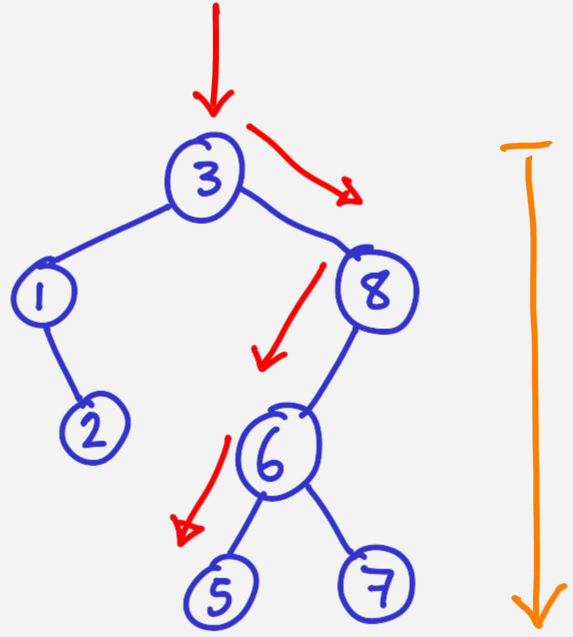
time:

$O(\text{depth})$

$O(\log n)$

BINARY SEARCH TREES

(binary) search for 5



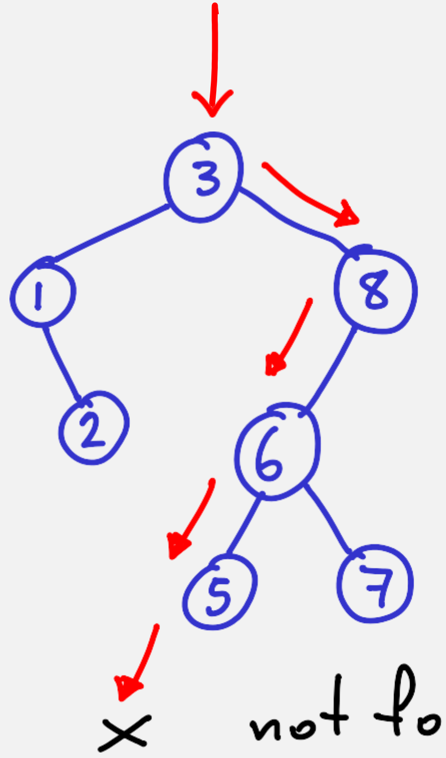
found

time:

$O(\text{depth})$

$O(\log n)$

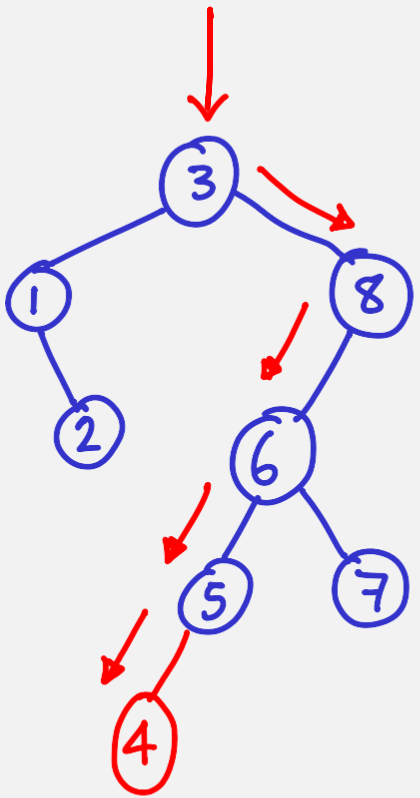
(binary) search for 4



not found

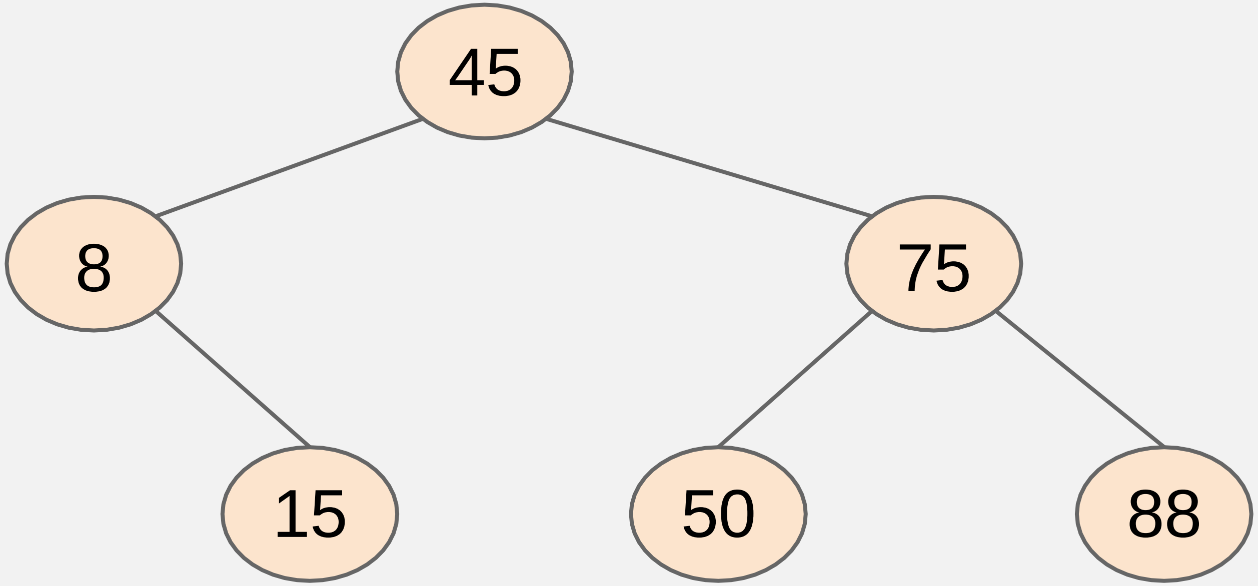
BINARY SEARCH TREES

insert (4) ∼ search



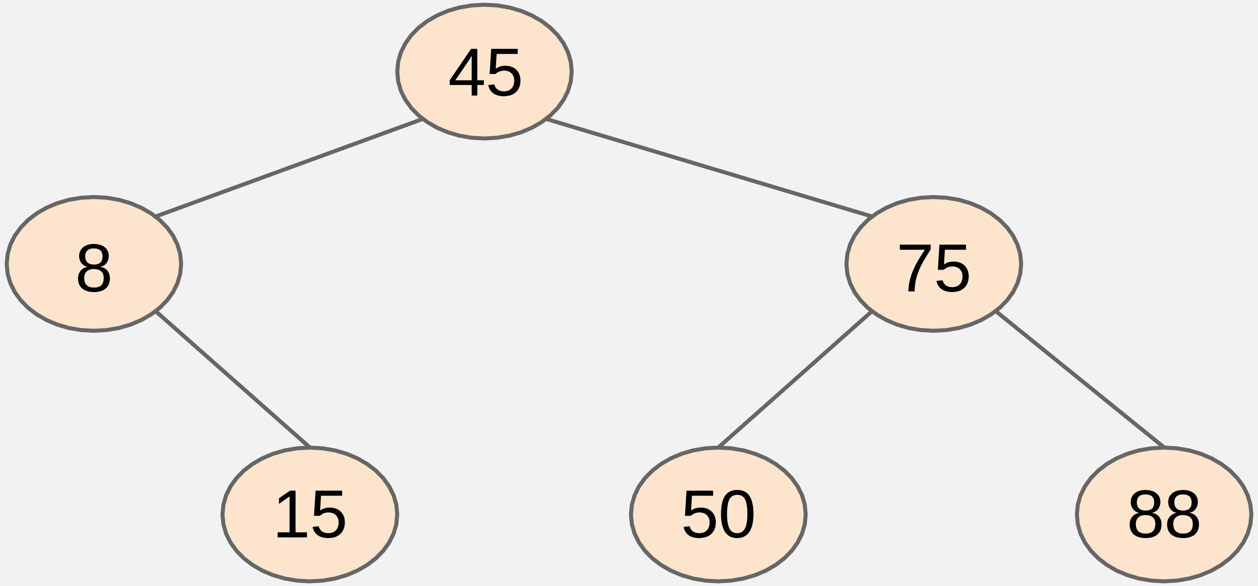
Insertion

<u>True or False</u>: nodes that are inserted will always become leaves in the tree



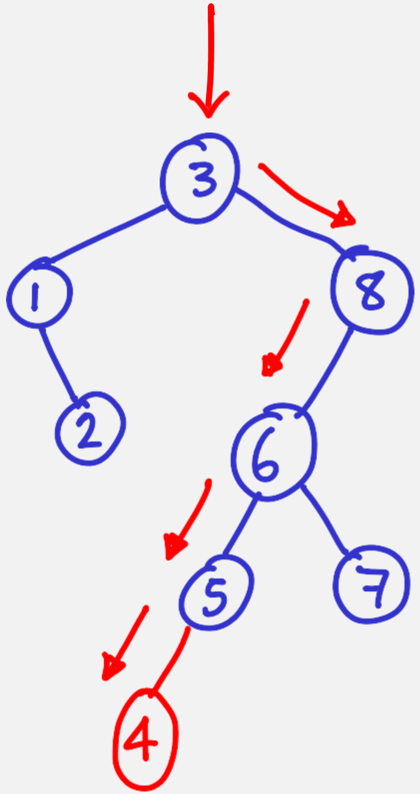
Insertion

True or False: nodes that are inserted will always become leaves in the tree

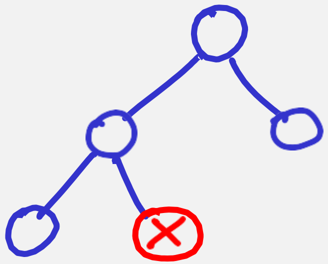


Binary search trees

insert (4) ~ search



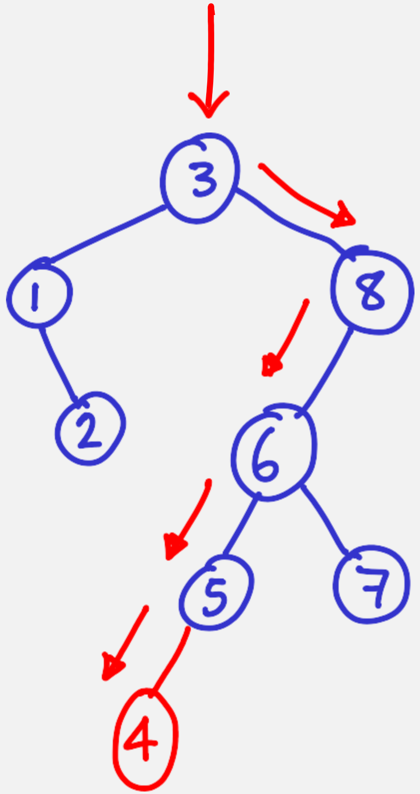
instant delete (x)



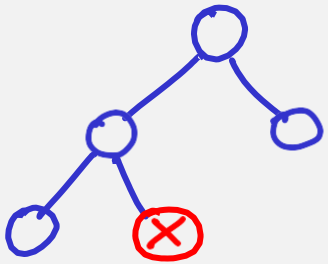
x : leaf

BINARY SEARCH TREES

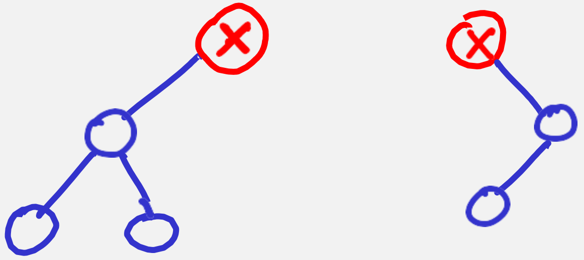
insert (4) ~ search



instant delete (x)



x : leaf

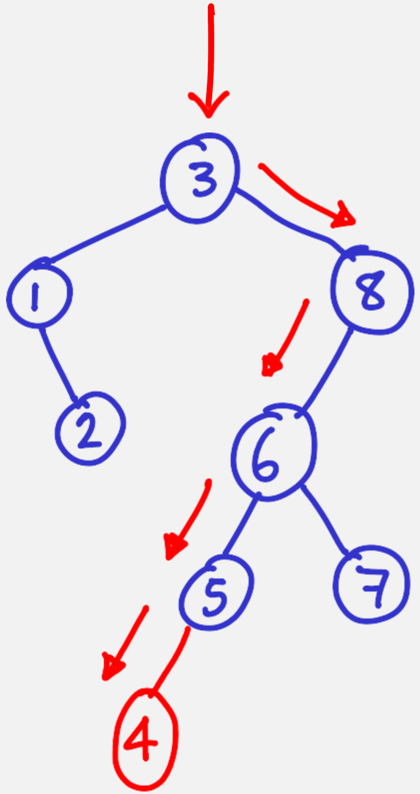


x : root w/ 1 child

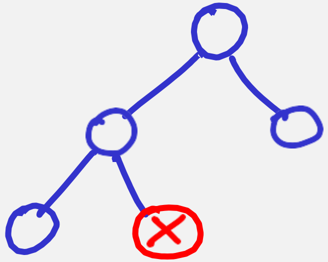
↳ x = MIN OR MAX

BINARY SEARCH TREES

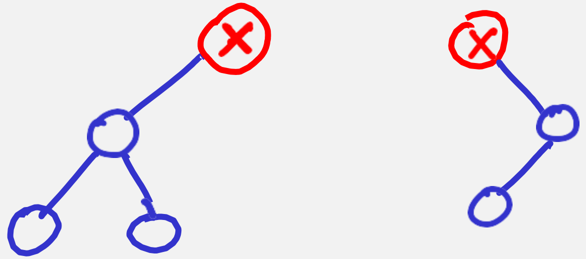
insert (4) ~ search



instant delete (x)

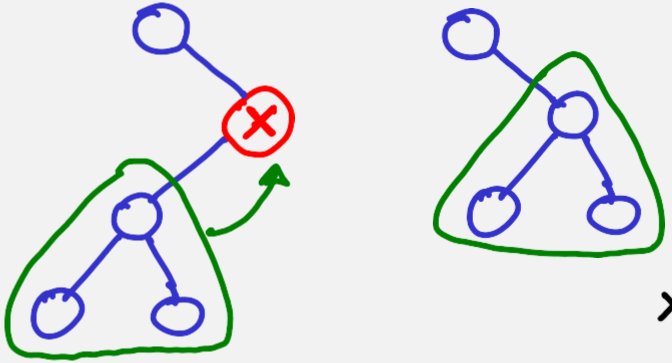


x : leaf



x : root w/ 1 child

↳ X = MIN OR MAX

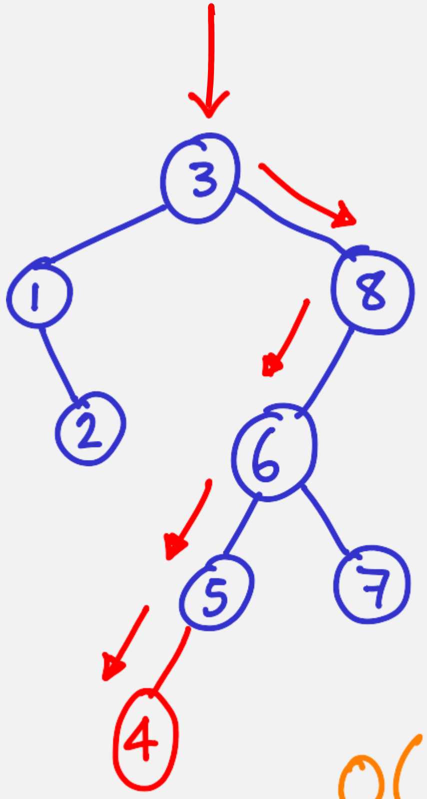


x : any node w/ 1 child

BINARY SEARCH TREES

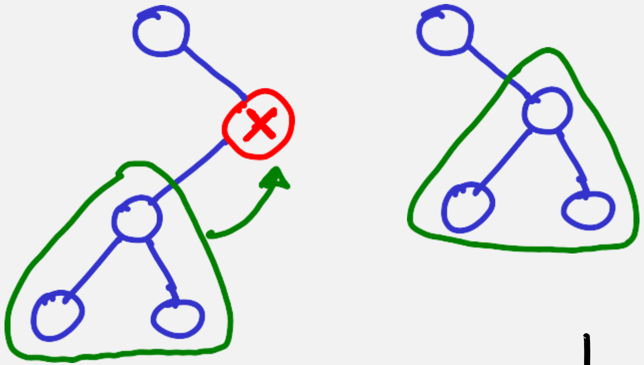
insert (4) ~ search

instant delete(x)



O(depth)

O(log n)



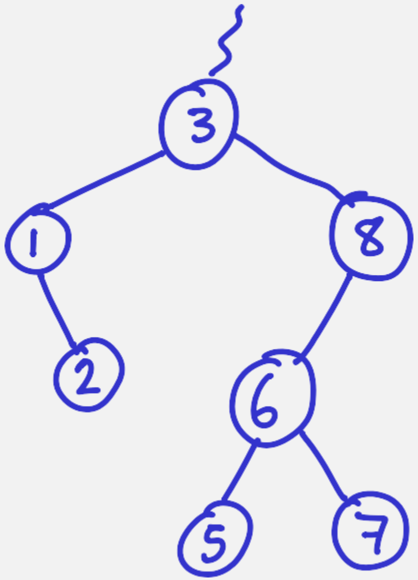
x: any node w/ < 2 children

If (one) subtree exists, promote it.

O(1)

BINARY SEARCH TREES

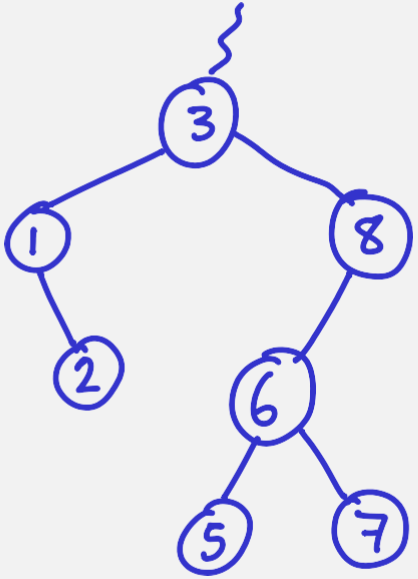
delete(3)

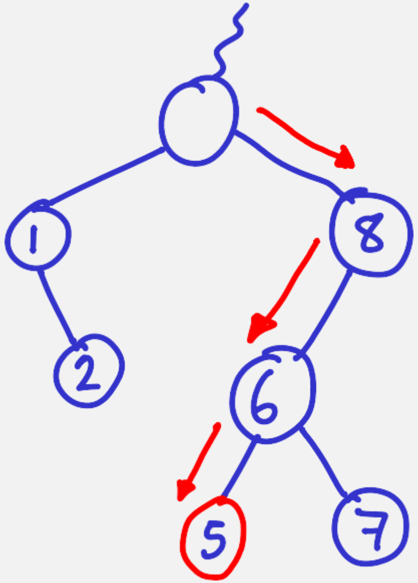


BINARY SEARCH TREES

non-instant

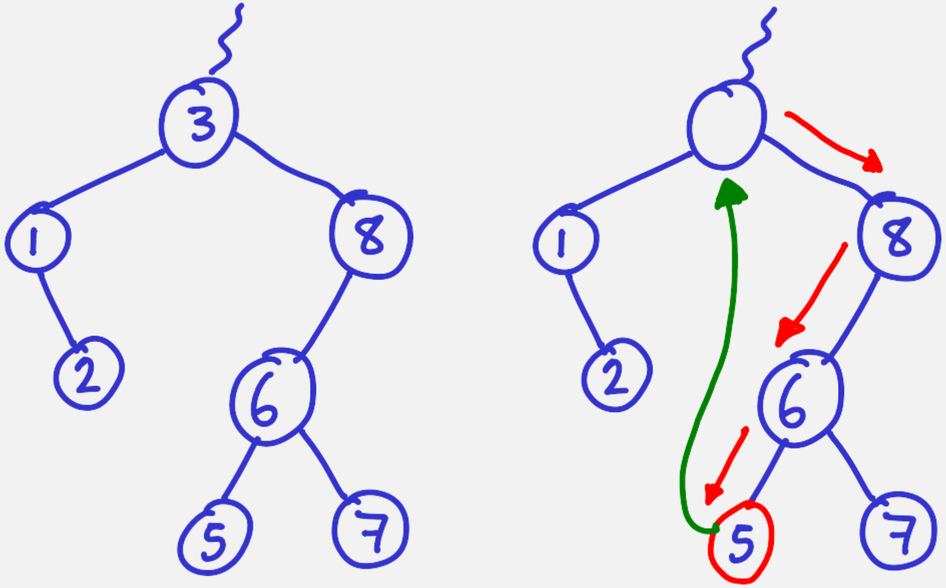
delete(3) ⟶ find successor : smallest element greater than 3
(which exists because : 2 children)





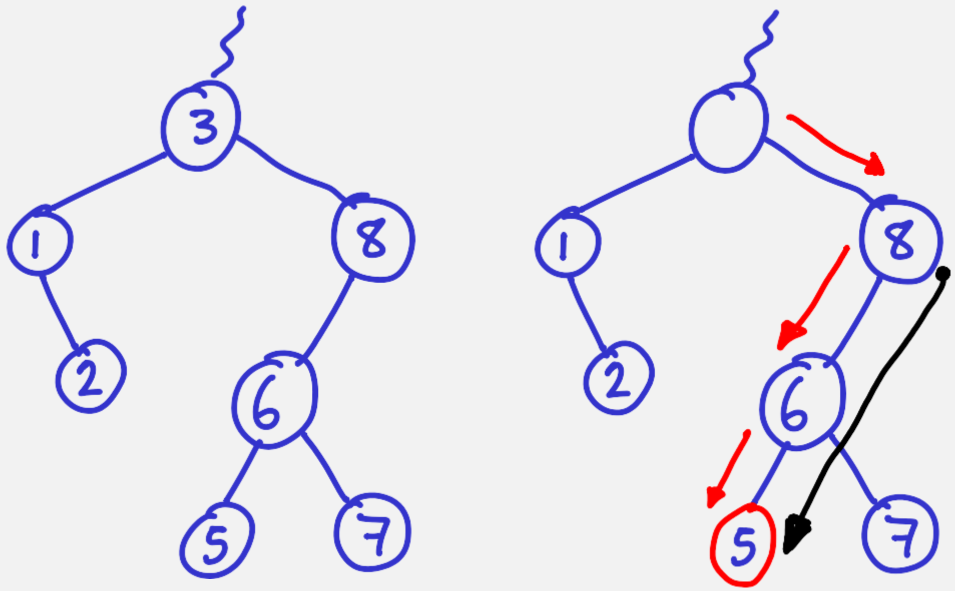
BINARY SEARCH TREES

delete(3) ⟶ find successor
& replace



BINARY SEARCH TREES

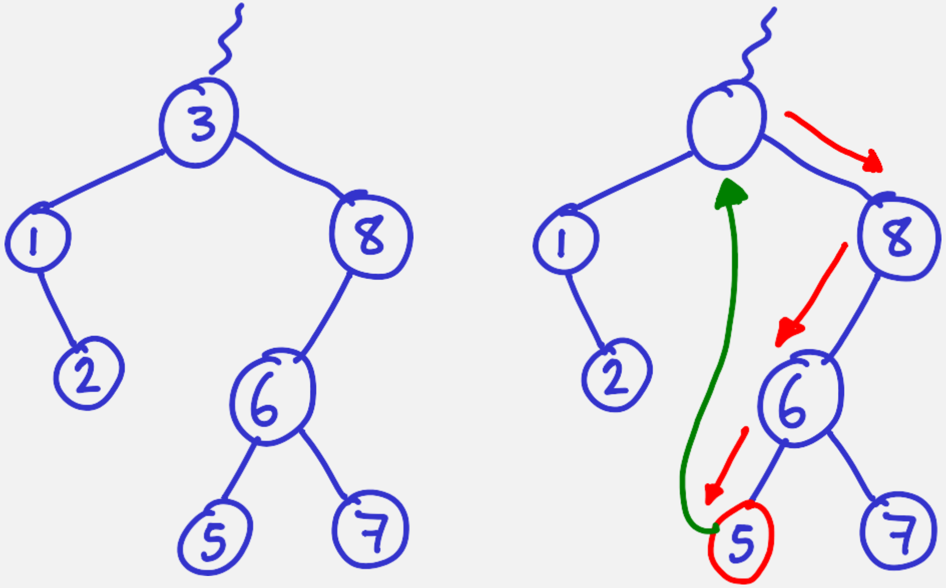
delete(3) → find successor & replace



By definition, successor is the last node visited on a ↙ path from R-child(3)

BINARY SEARCH TREES

delete(3) → find successor
& replace

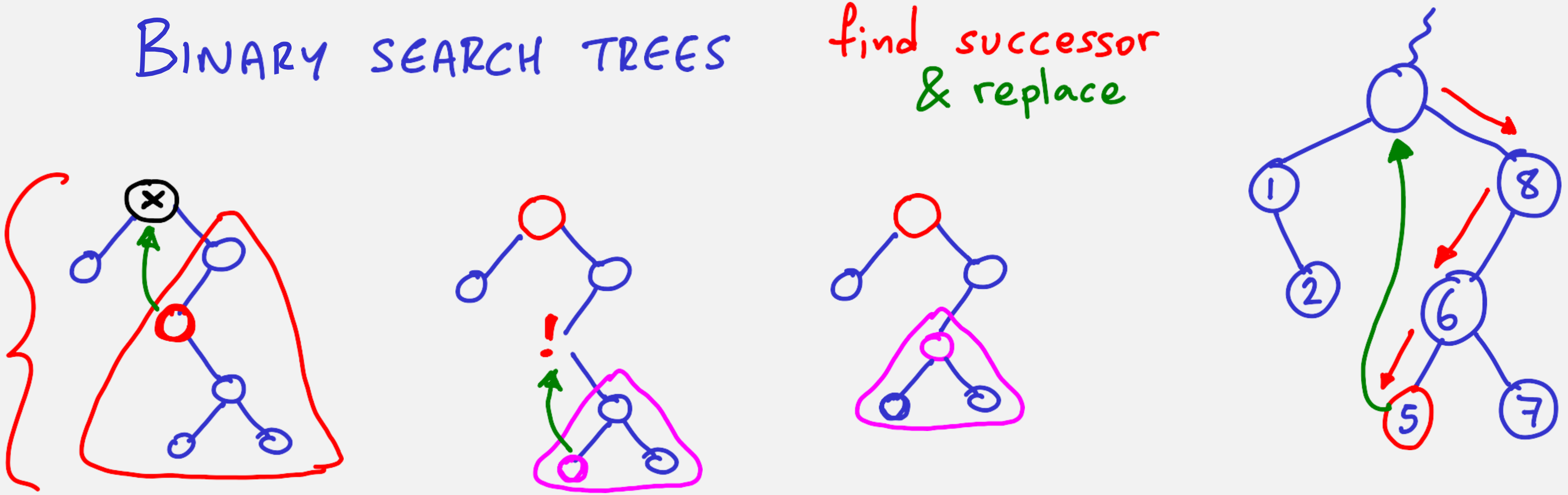


By definition,
successor is
the last node visited on a
↙ path from R-child(3)

```c
struct node* deleteNode(struct node* root, int key)
{
// base case
if (root == NULL)
return root;

if (key < root->key)
root->left = deleteNode(root->left, key);

else if (key > root->key)
root->right = deleteNode(root->right, key);

// if key is same as root's key?
else {
```

```c
// node with only one child or no child:
if (root->left == NULL) {
struct node* temp = root->right;
free(root);
return temp;}

else if (root->right == NULL) {
struct node* temp = root->left;
free(root);
return temp;}

// node with two children:
struct node* temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);
}
return root;
```

BINARY SEARCH TREES




```
struct node* minValueNode(struct node* node)
{
struct node* current = node;

/* loop down to find the leftmost leaf */
while (current && current->left != NULL)
current = current->left;
return current;
}
```

Tree Traversals

Three ways:

Pre-order: visit node, left subtree, right subtree.

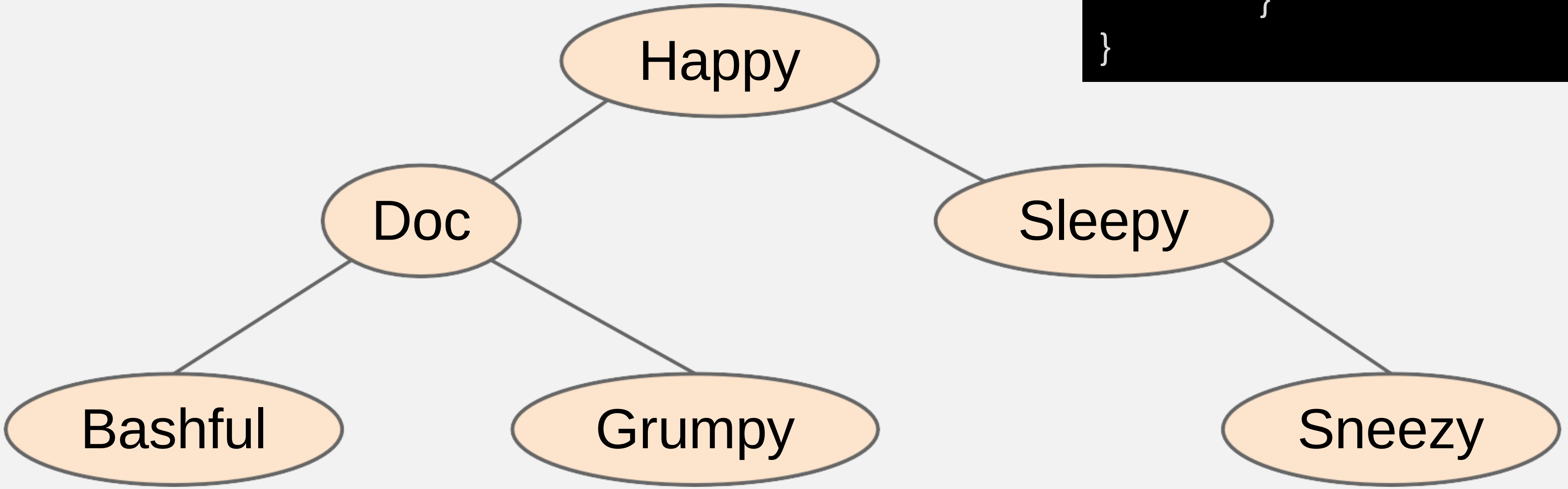
In-order: visit left subtree, node, right subtree.

Post-order: visit left subtree, right subtree, node.

(think of the “visit” as a print operation)

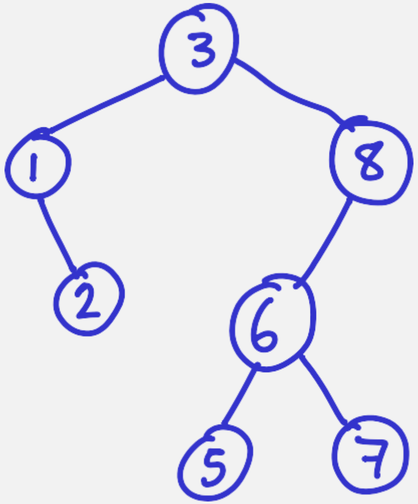
Example Tree: 6 of the 7 Dwarves

```c
void traverse(struct node* root)
{
if (root != NULL) {
		traverse(root->left);
		printf("%s", root->key);
		traverse(root->right);
		}
}
```



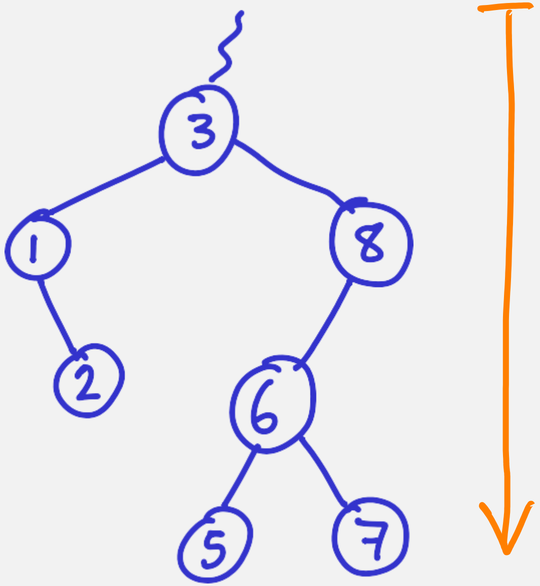
In-order: Bashful, Doc, Grumpy, Happy, Sleepy, Sneezy
Pre-order: Happy, Doc, Bashful, Grumpy, Sleepy, Sneezy
Post-order: Bashful, Grumpy, Doc, Sneezy, Sleepy, Happy

BINARY SEARCH TREES – BUILT RANDOMLY



Insert n elements into a BST in the order that they're given.

BINARY SEARCH TREE SUMMARY



INSERT → SEARCH → O(depth)

DELETE → SEARCH

O(log n)

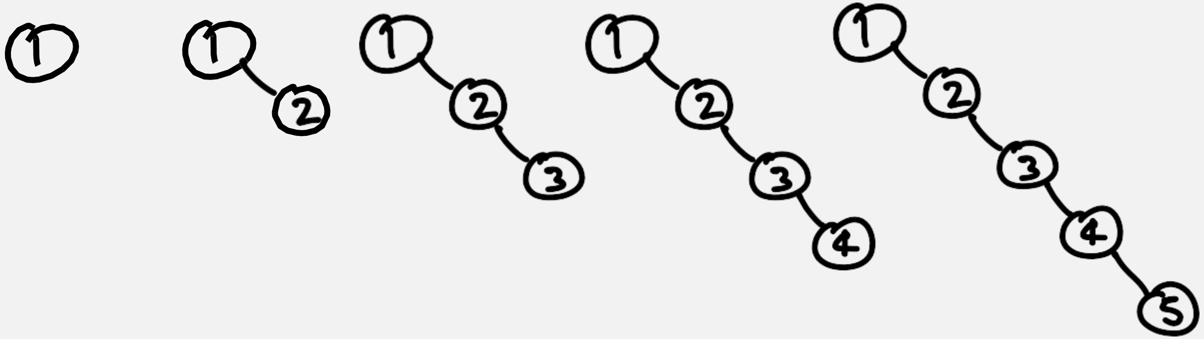
We should keep the tree balanced as much as possible

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be ?

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be ?

↳ already sorted input, reverse-sorted, nearly sorted...

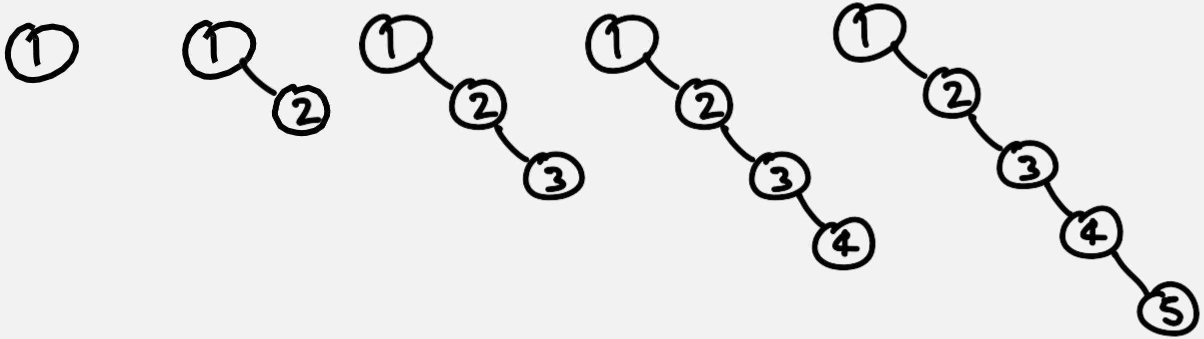
1 2 3 4 5 →



- What is the worst-case time complexity, and why?
- How unbalanced could the tree be?

↳ already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 →

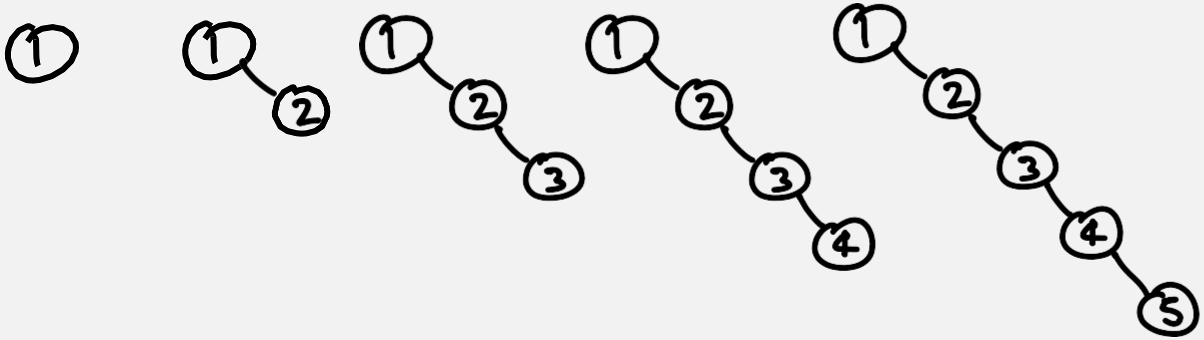


O(n) depth

- What is the worst-case time complexity, and why?
- How unbalanced could the tree be?

↳ already sorted input, reverse-sorted, nearly sorted...

1 2 3 4 5 →



$O(n)$ depth

- What would be ideal?

$O(\log n)$