COMP20003 Workshop Week 5

Binary Trees & BST, Traversal
AVL & Rotations

Practical Issues:
- defining a tree node, and a tree or bst
- bst_insert
- “on paper” AVL rotation

Assignment 1 (?)

ASS1 situation: send me a letter, private if you prefer

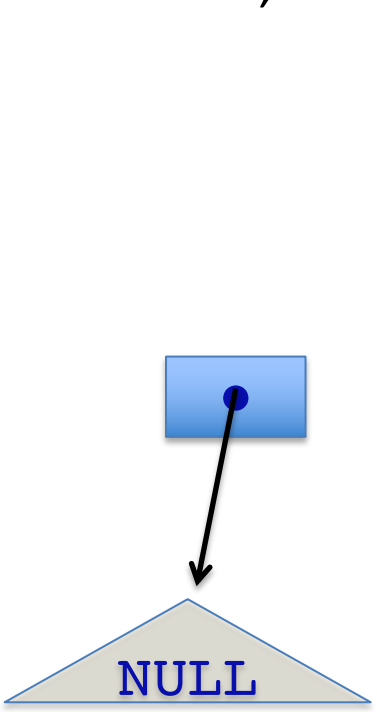
A. Passed
B. Got more than 7/22 tests passed
C. Got <= 7/22 tests passed, and won't do late submit
D. Doing, or will do late submission

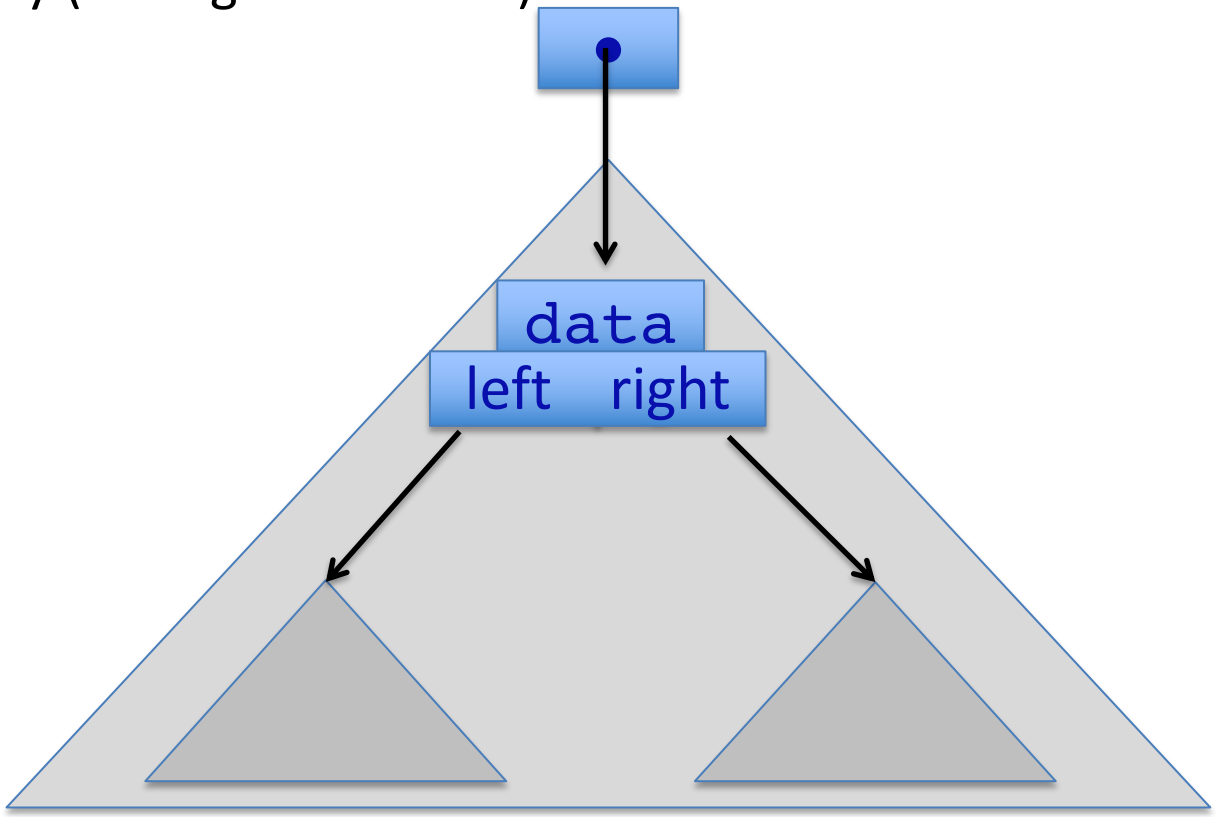
Review: Binary trees and BST

A Binary Tree can have up to 3 components:

- a *root node* (with some `data`) which is connected to:
- a *left sub-tree* (which is a tree), and a
- a *right sub-tree* (which is another tree).

It is convenient to defined a tree as a pointer to its root node. In the base case, a tree is empty (having NULL value).



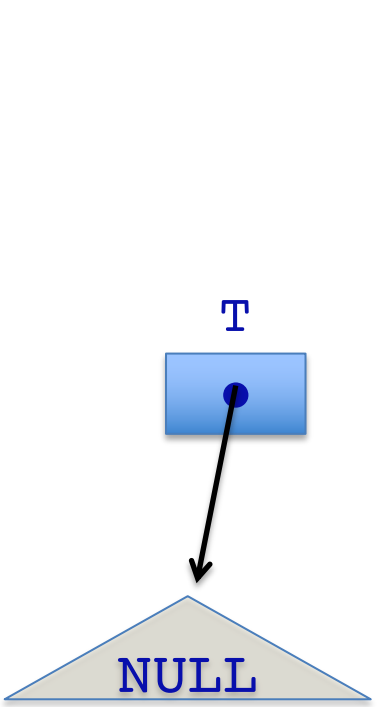


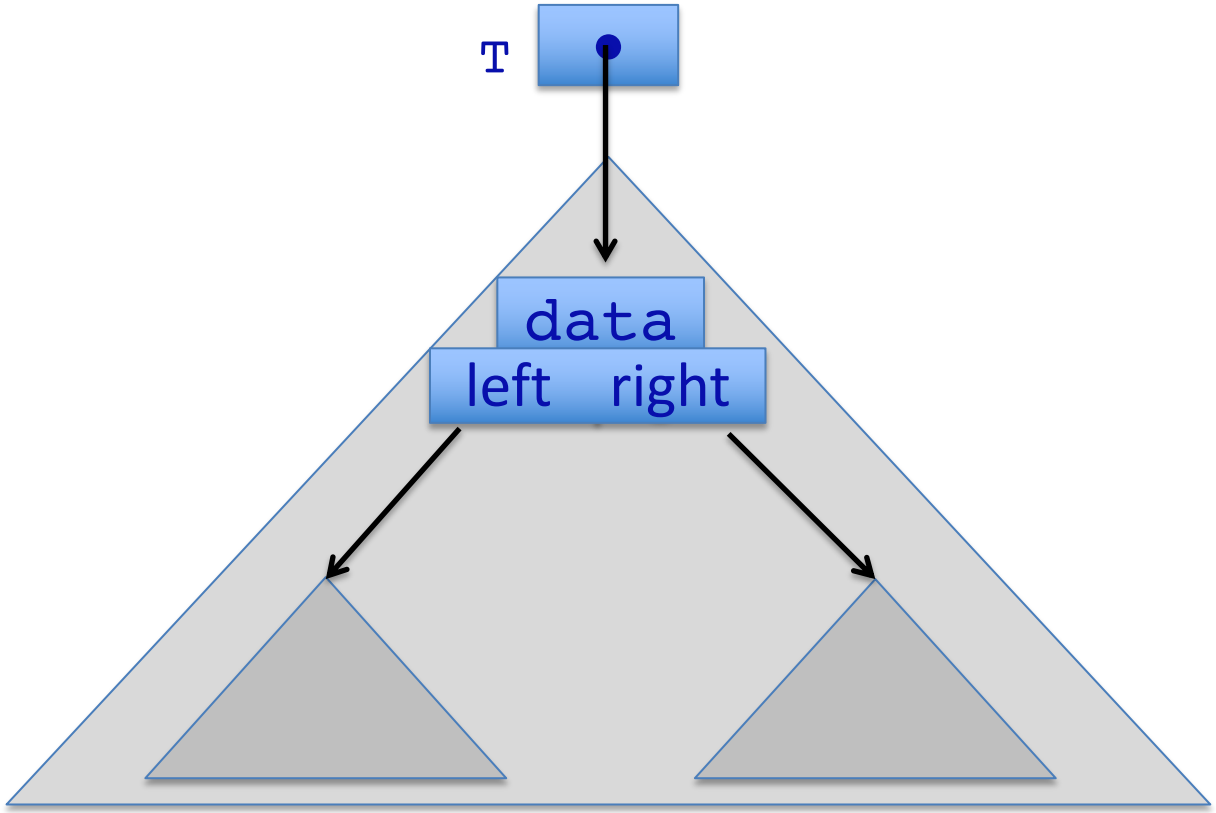
Review: Binary trees and BST

- *root node, left sub-tree, right sub-tree*
  = parent , left child , right child

If a binary tree is defined by a data type `tree_t`, then

- `tree_t` is a pointer to a tree node
- with `tree_t T;` `T` can either be `NULL` or have 2 children `T->left` and `T->right`





Declaring trees: code examples

Model 1: in ED workspace style	Model 2:	Notes

`struct bst {`
`  struct data data;`
`  struct bst *left;`
`  struct bst *right;`
`};`
`struct bst *t= NULL;`
	`typedef struct t_node *tree_t;`
	`struct t_node {`
	`  data_t *data;`
	`  tree_t left;`
	`  tree_t right;`
	`};`
	`tree_t t= NULL;`
		a tree node
		a data
		left chilld
		right child
		
		at the start,
		t is empty

Note that often `data_t` include a special field `key`. And:

In programming practice:

- the most convenient way for data is `void *data`

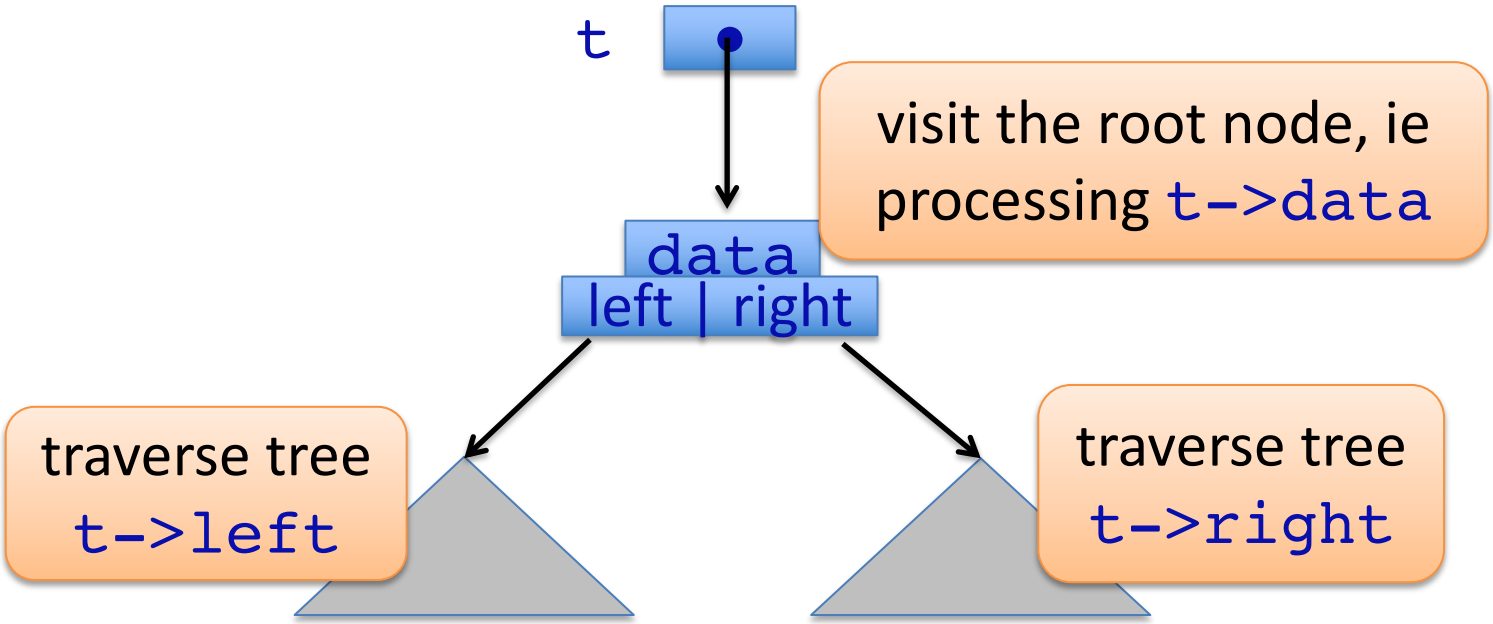
In lecture/workshop demonstrations:

- the effective way is `int key (`ignoring other components of data)

Tree/BST traversal= visiting all nodes of a tree

Tree traversal= visit all nodes of a tree in a systematic way.

For a non-empty tree , there are 3 works , and they can be done in any order!



Tree/BST traversal

Dictionary:

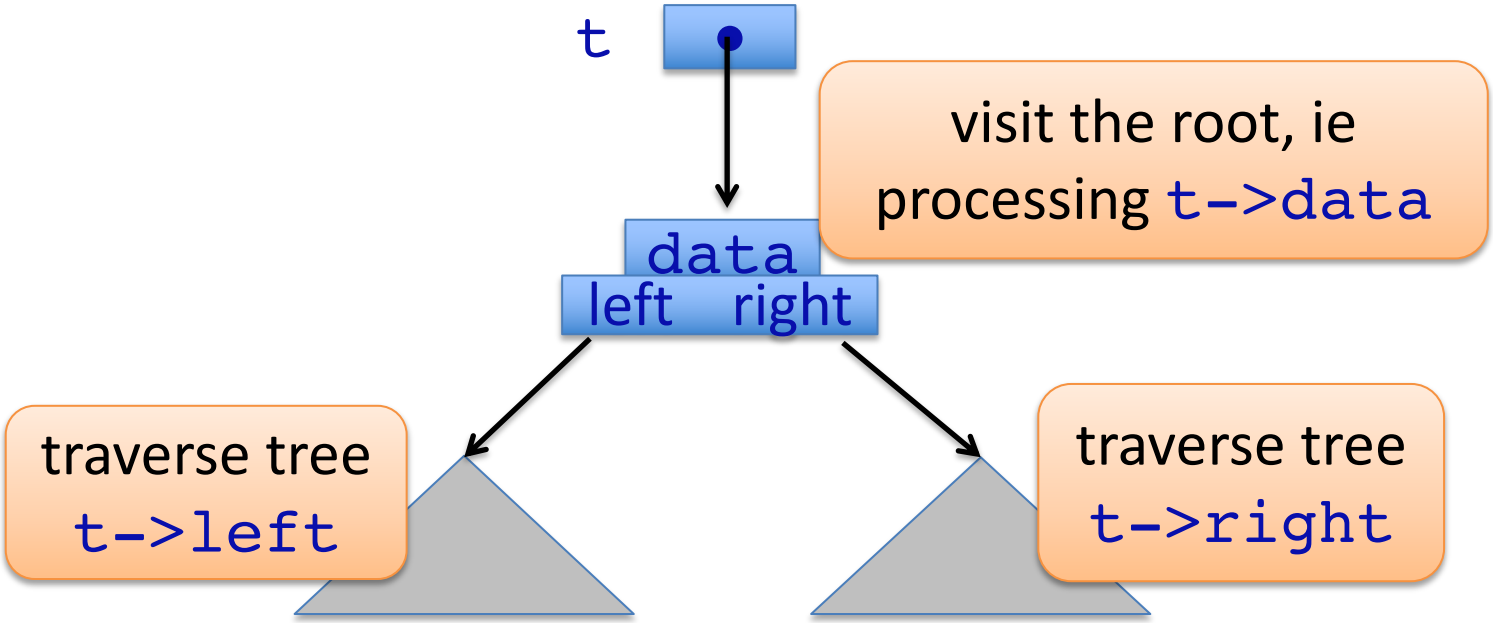
visit root: do some processing with `t->data`.

traverse left tree: recursive call to the same function for `t->left`.

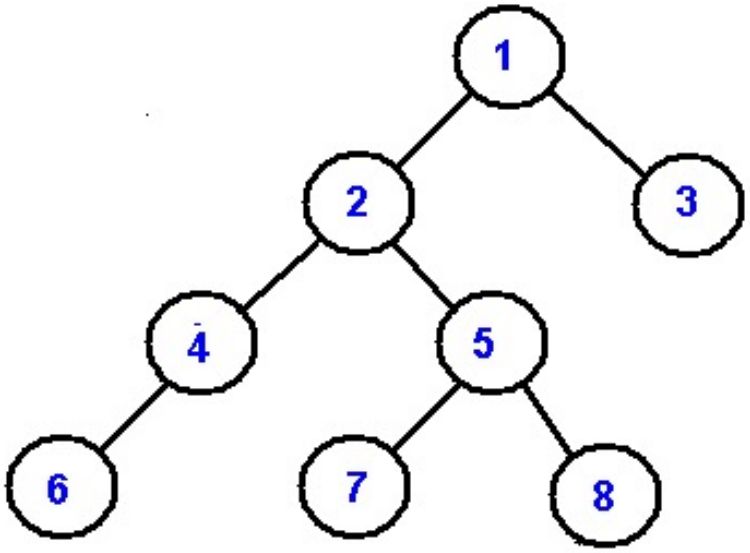
Depending on when to visit the root node, we have:

- *pre-order* (visit root first),
- *post-order* (visit root last), and
- *in-order* (visit root in the middle)

of the three works



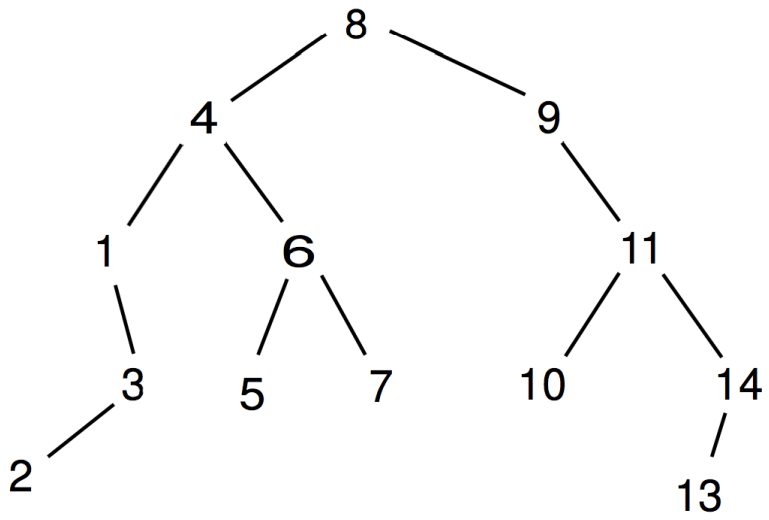
Tree Traversal Examples



List the nodes in order visited by:

- in-order :
- pre-order :
- post-order :

BST



Exercise (supposing data is just `int key`)

Write a C function for:

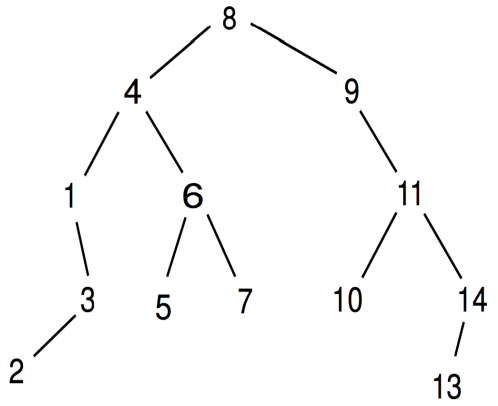
- printing a BST's keys in increasing order
- printing a BST's keys in decreasing order

```
struct bst {
  int key;
  struct bst *left;
  struct bst *right;
};
typedef struct bst *tree_t;
```

```
void printBST(                    ) {




}
```



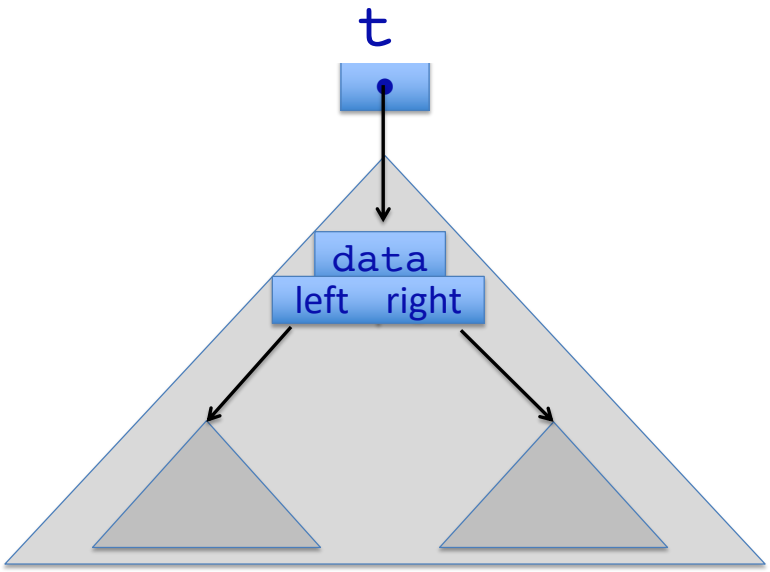
Exercise

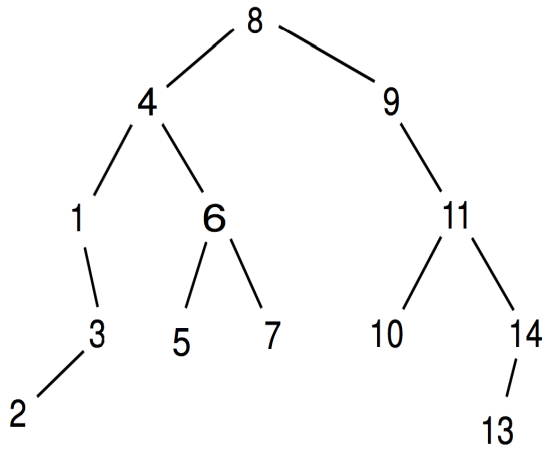
What traversal order should be used for:

- copying a tree ?

- free a tree ?

```
struct bst {
   ...
   struct bst *left;
   struct bst *right;
};
```





Programming : How to implement bstInsert?

Function header for `struct bst *t`

```
???   bstInsert( ???    );
```

How to implement bstInsert?

Is this function correct ? Supposing `t` is `NULL` at the beginning.

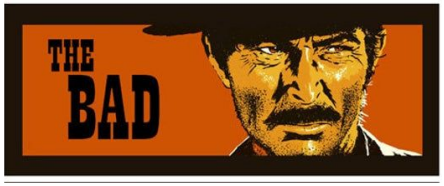
```
struct bst *bstInsert(struct bst *t, int key) {
  if (t==NULL) {
    t= malloc(*t);
    t->key= key;
    t->left= t->right= NULL;

  } else if (key < t->key)
        bstInsert(t->left, key);
  else …
  return t;
}
```

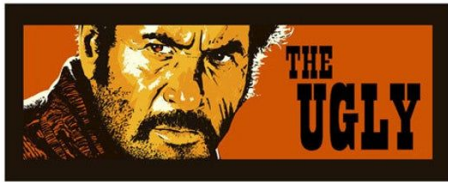
struct bst *t= NULL;
t= bst_insert(t, 10);

Notes: For `tree_t`, replace `struct bst*` with `tree_t:`

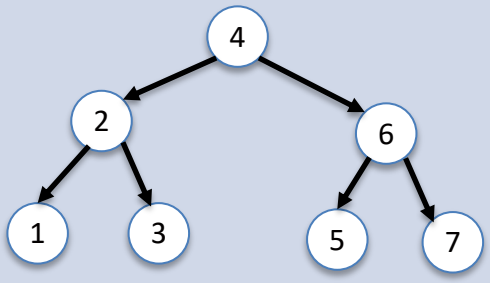
BST efficiency depends on the order of input data

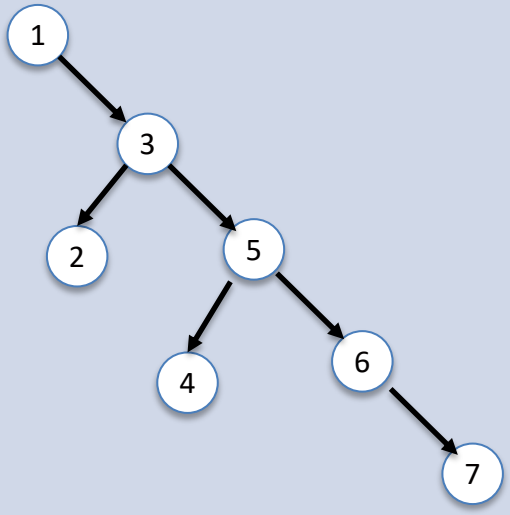


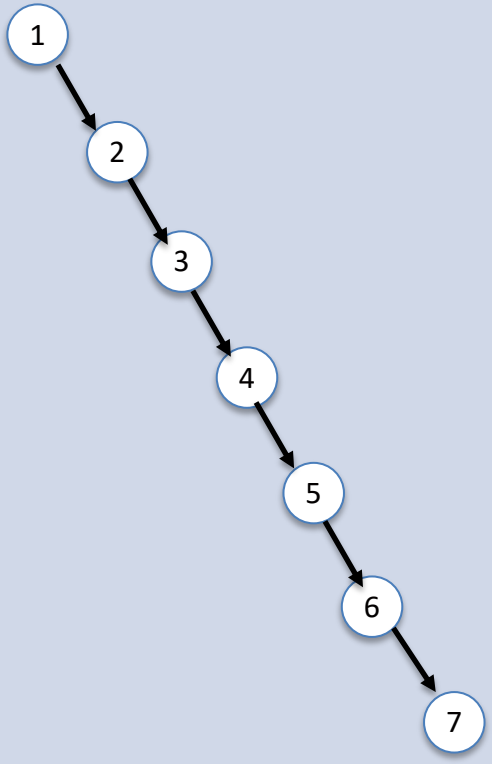
AND



4 6 2 1 3 7 5	1 3 5 2 4 6 7	1 2 3 4 5 6 7







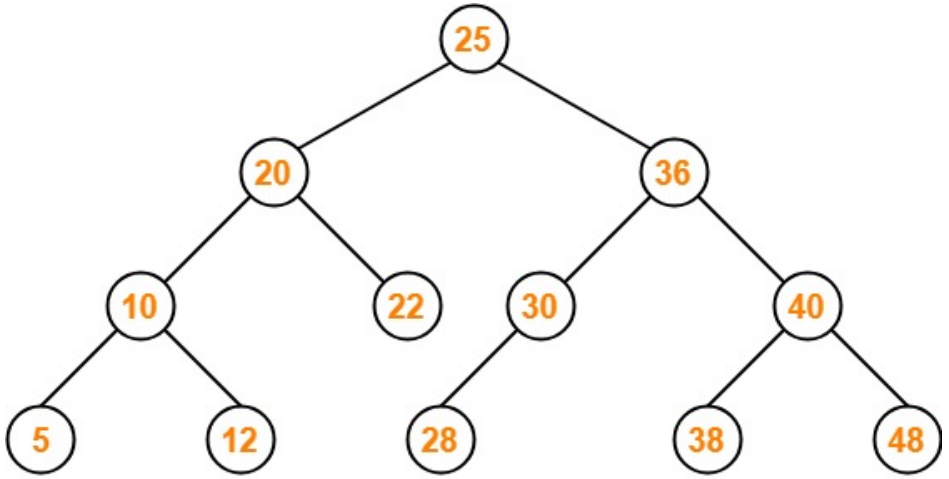
Want The Good, no matter what's the data input order? Use AVL (or …)!

Good-Bad_Ugly Picture Source: https://www.pinterest.com.au/pin/170573904624610413/

The Goods and the Bads of BST

The Goods:

Average performance for both `insert` and `search` is `O(log n)`

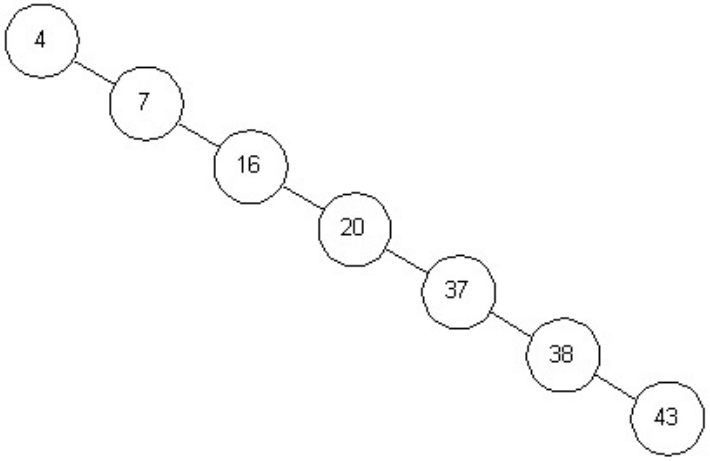


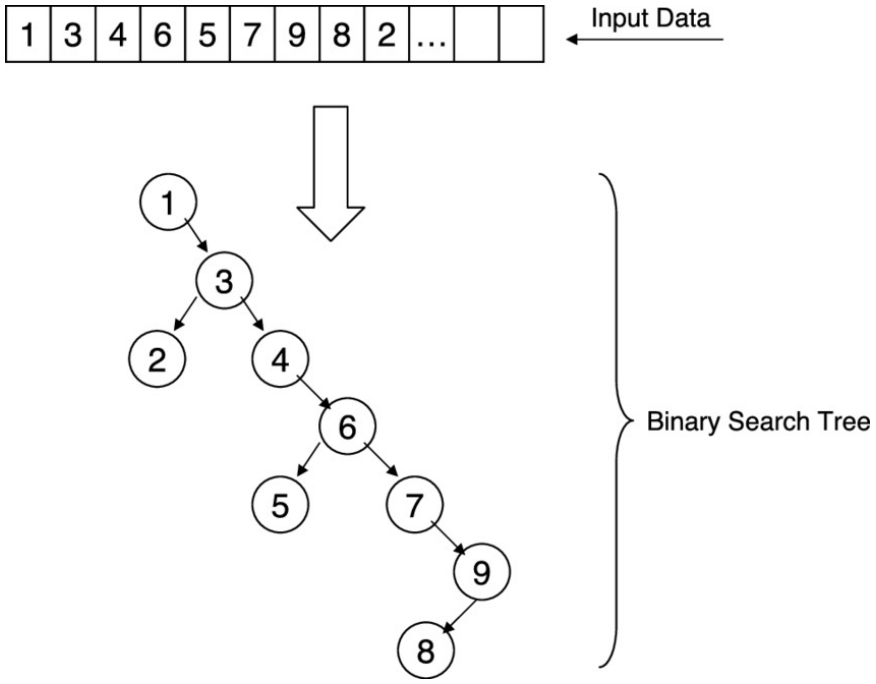
Binary Search Tree

The height of the tree is around $\log_2 n$ in average

The Goods and the Bads of BST

The Bads: *Worst-case* performance for both `insert` and `search` is `O(n)`





The height of the tree could be around `n`

AVL = keeping BST tree honest (ie. balanced)!

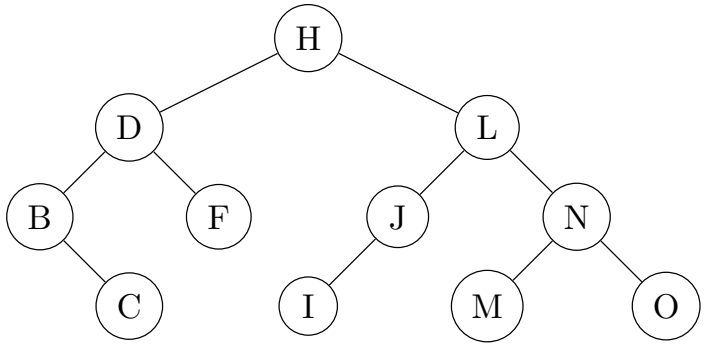
Efficiency?

Operation	Case	Binary Trees	
		BST	AVL
Insert	Average	O( )	O( )
	Worst	O( )	O( )
Search	Average	O( )	O( )
	Worst	O( )	O( )

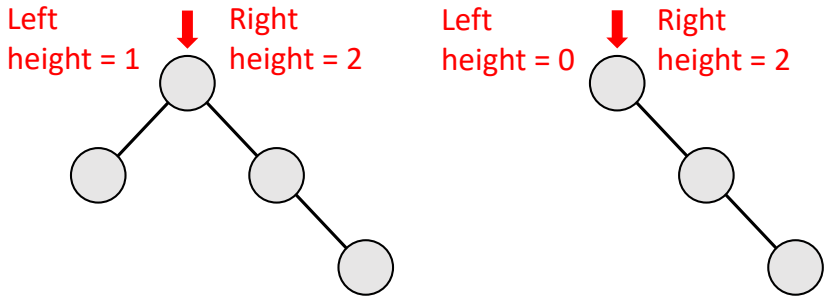
How to know if a node/tree is imbalanced?

A node is *balanced* the heights of its left tree and its right tree differ by at most 1

A tree is balanced each of its nodes is balanced.



- Is this node balanced?



Balanced:
Difference is <=1

Unbalanced:
Difference is >1

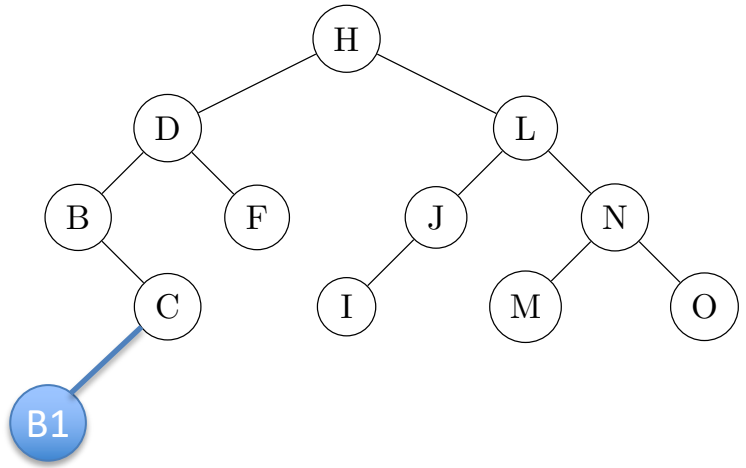
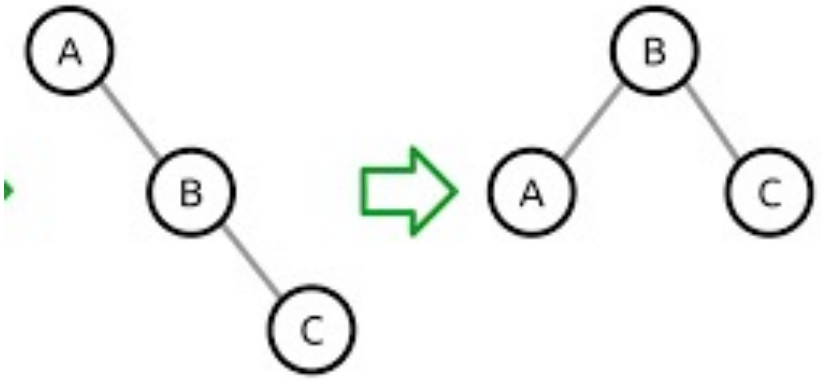


fig. from lecture slides

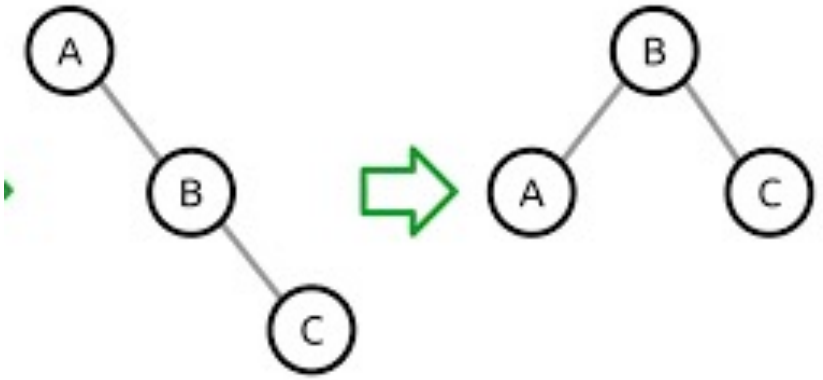
Two Basic Rotations: 1) Single Rotation for Sticks



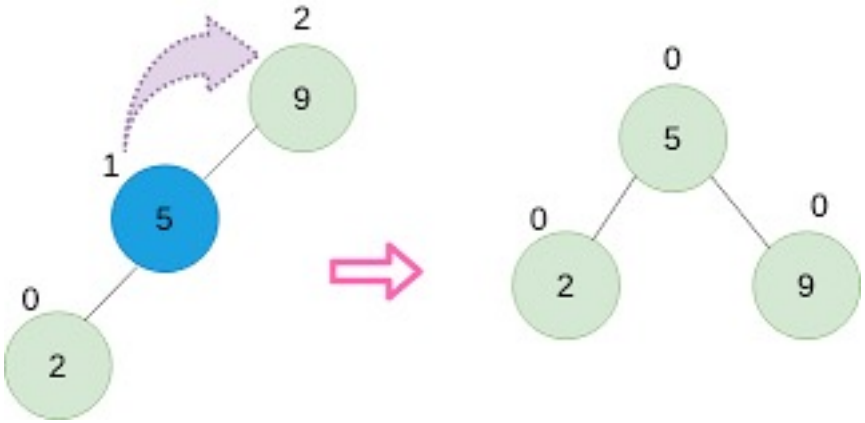
single **L rotation** applied to root→right_child

single **R rotation** applied to root→left_child

Two Basic Rotations: 1) Single Rotation for Sticks

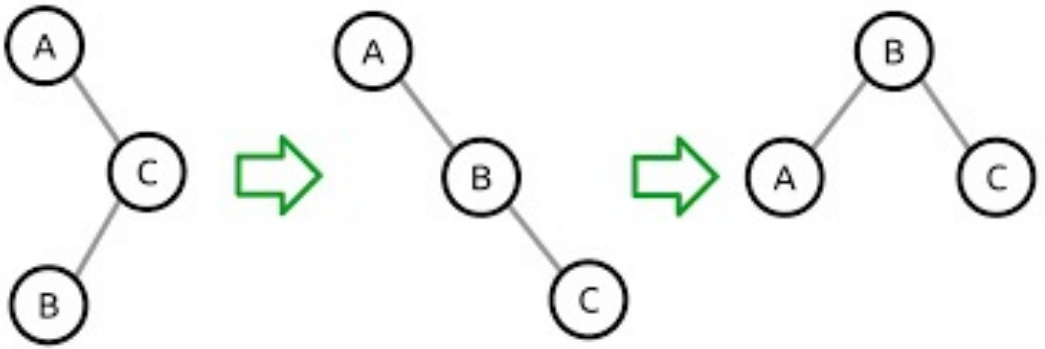


single **L rotation** applied to root→right_child



single **R rotation** applied to root→left_child

Two Basic Rotations: 2) Double Rotation for non-stick

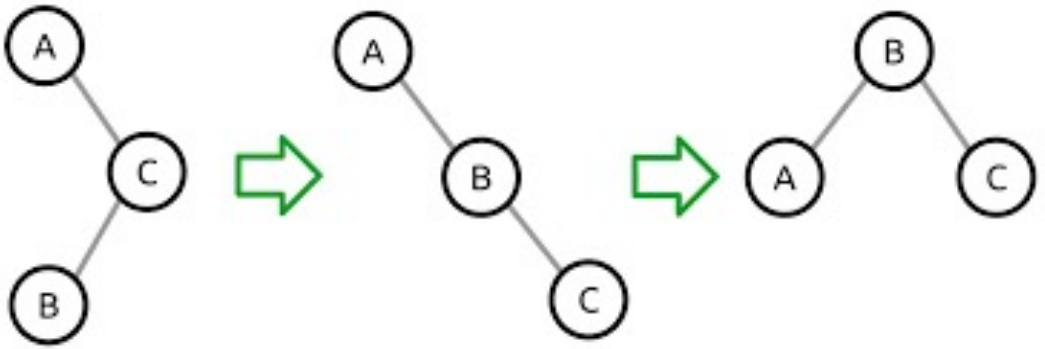


1) a single rotation to child→grandchild to turn root to a stick
2) another single rotation to balance the stick

Here we have a **RL double rotation**

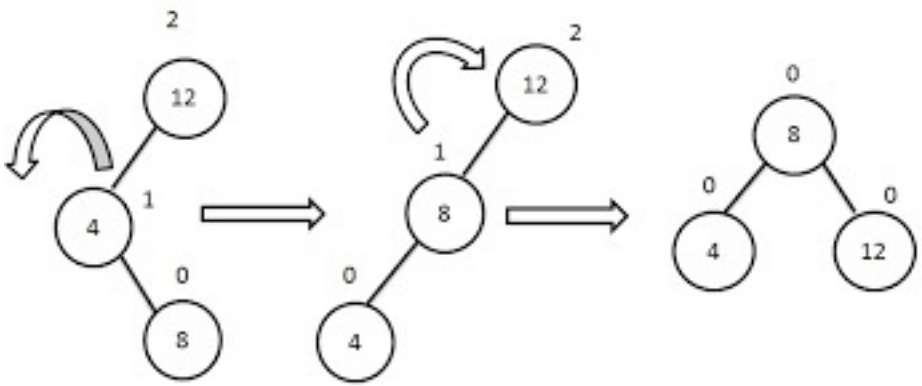
Similar for RL (first right, second left) double rotation.

Two Basic Rotations: 2) Double Rotation for non-stick



1) a single R-rotation to child→grandchild to turn root to a stick
2) another single L-rotation to balance the stick

Here we have a **RL double rotation**



Similar for LR (first left, second right) double rotation.

Using Rotations to rebalance AVL

Problem: When inserting to AVL, it might become unbalanced

Approach: Rotations (Rotate WHAT?, and HOW?)

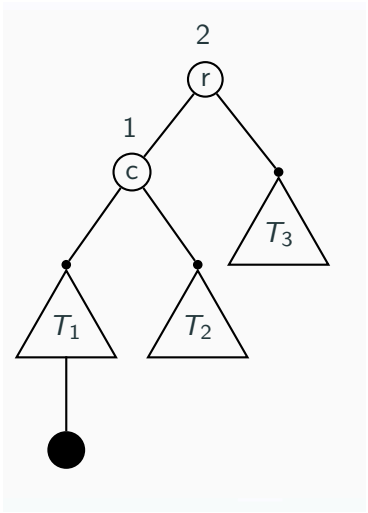
Rotate WHAT?

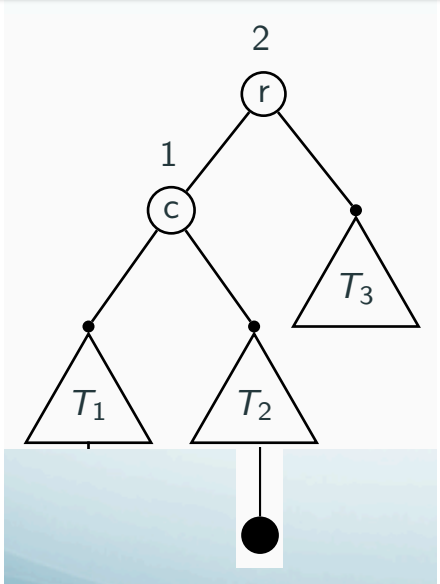
The *lowest* subtree X which is unbalance

HOW

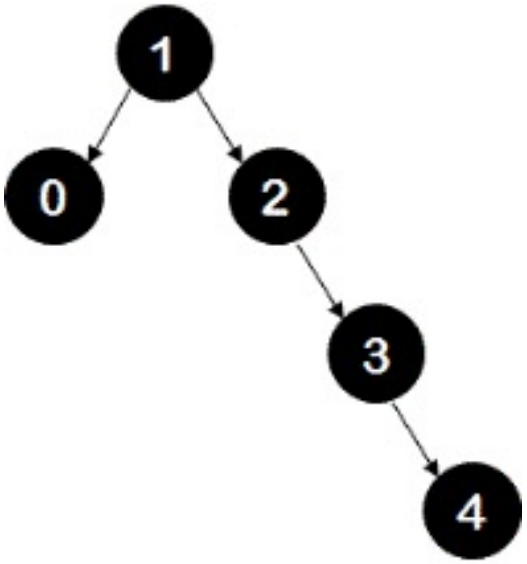
Consider *the first 3 nodes* X→A→B in the path from root X to the just-inserted node

Apply a single rotations if that path is a stick, double rotation otherwise

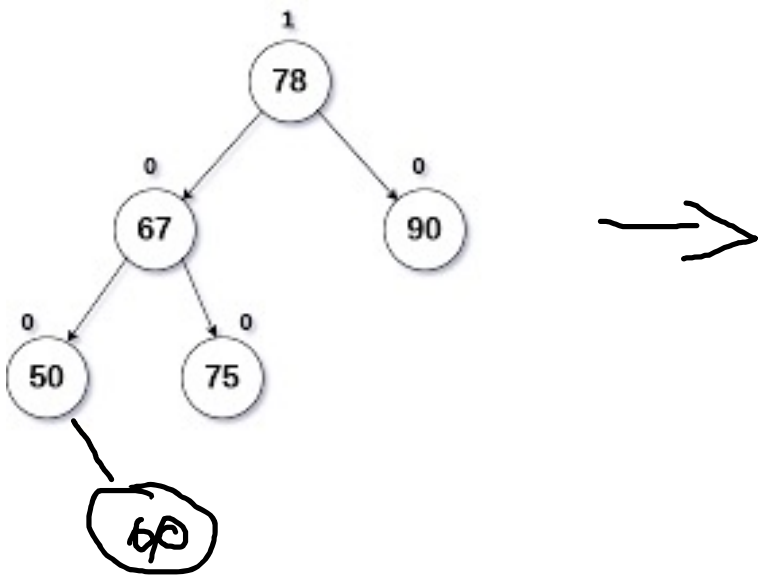


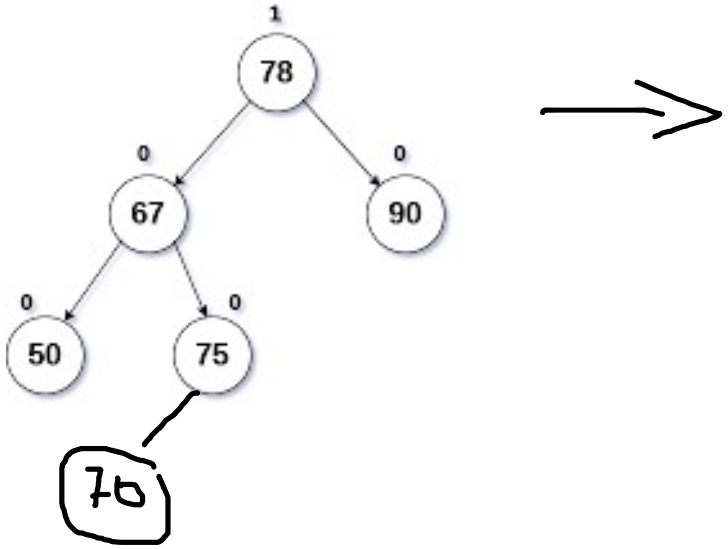


Examples



Examples: rebalance after inserting 60? 70 ?





Lab: Finish Your Ass1 OR

- Test your BST understanding with Lessons → Week 5 Workshop → Workshop Polls (used with Grady's slides Week 5 Workshop → Workshop Slides)
- BST's insert and free: Implement function `bstInsert` and `freeTree` under Programming 4.1. To do that:
  - You can fork and use the space Week 5 Anh Workshop that also contains a simple function for printing tree
  - Note that other than the above function, Week 5 Anh Workshop is the same as Week 5 Workshop
- Try Programming Challenge 4.1

The **proven most effective way** to ask for Anh's help with the ass1 code, use it now and even after the workshop:

- Share your assignment's workspace with avo@unimelb.edu.au
- Send avo an email with a bit of details on your problem, including the filename and line numbers, Anh will put comments directly there
- Best time for Anh's response: during workshops, or 6:00-7:00AM, 7:30-10:00PM

This set of slides is available for downloading from now at github.com/anhvir/c203. The file will be removed after 10 minutes (but you can ask Anh to send later)

Check your Lab code: Version 1

```
struct bst *bstInsert(struct bst *t, int key) {
  if (t==NULL) {
    t= malloc(*t);
    t->key= key;
    t->left= t->right= NULL;
  } else if (key < t->key)
    bst->left= bst_insert(t->left, key);
  else …
  return t;
}
struct bst *t= NULL;
t= insert(t, 10);
```

Notes: For `tree_t`, replace `struct bst*` with `tree_t:`

Check your Lab code: version 2

```
void bst_insert(struct bst **t, int key) {
  if (*t==NULL) {
    *t= malloc(**t);
    (*t)->key= key;
    (*t)->left= (*t)->right= NULL;
  } else if (key < (*t)->key)
              bst_insert(&((*t)->left), key);
  else …
}
```

Notes: For `tree_t`, replace `struct bst*` with `tree_t:`

Evolution? Concrete data structures for dict

Operation	Case	Arrays / Linked Lists	Sorted Arrays	Binary Trees		B - Trees	
				BST	AVL	2-3 Tree	general B-Tree
Insert	Average	O(1)	O(n)	O(log n)	O(log n)		
	Worst	O(1)	O(n)	O(n)	O(log n)		
Search	Average	O(n)	O(log n)	O(log n)	O(log n)		
	Worst	O(n)	O(log n)	O(n)	O(log n)		

Concrete data structures for dict

Operation	Case	Arrays / Linked Lists	Sorted Arrays	BST	AVL
Insert	Average	O(1)	O(n)	O( )	O( )
	Worst	O(1)	O(n)	O( )	O( )
Search	Average	O(n)	O(log n)	O( )	O( )
	Worst	O(n)	O(log n)	O( )	O( )

Concrete data structures for dict

Operation	Case	Arrays / Linked Lists	Sorted Arrays	BST	AVL
Insert	Average	O(1)	O(n)	O(logn)	O(logn)
	Worst	O(1)	O(n)	O(n)	**O(logn)**
Search	Average	O(n)	O(log n)	O(logn)	O(logn)
	Worst	O(n)	O(log n)	O(n)	**O(logn)**
Search overall		O(n)	**Θ**(logn)	O(n)	**Θ**(logn)