trees

are lists enough?

for correctness — sure

want to efficiently access items

better than linear time to find something

want to represent relationships more naturally

inter-item relationships in lists

$\boxed{1} \leftrightarrow \boxed{2} \leftrightarrow \boxed{3} \leftrightarrow \boxed{4} \leftrightarrow \boxed{5}$

List: *nodes* related to predecessor/successor

trees

trees: allow representing more relationships

(but not arbitrary relationships — see graphs later in semester)

restriction: single path from *root* to every node

implies single path from every node to every other node (possibly through root)

natural trees: phylogenetic tree

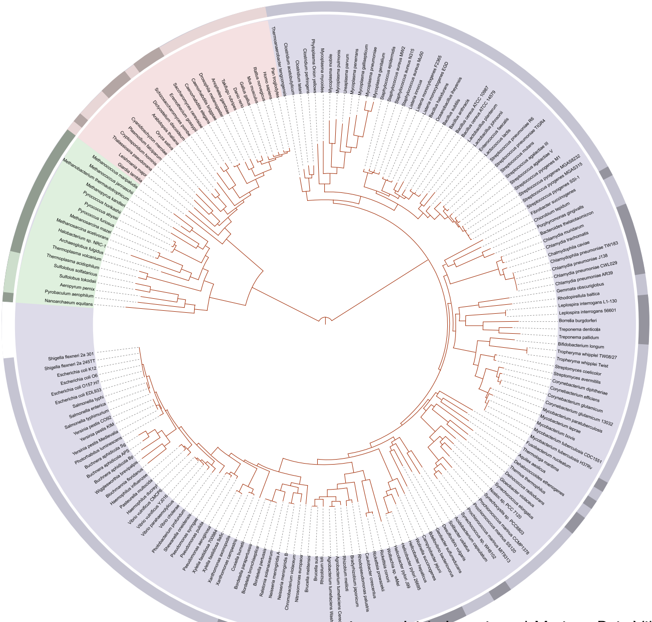


image: Ivicia Letunic and Mariana Ruiz Villarreal, via the tool iTOL (Interative Tree of Life), via Wikipedia

natural trees: phylogenetic tree (zoom)

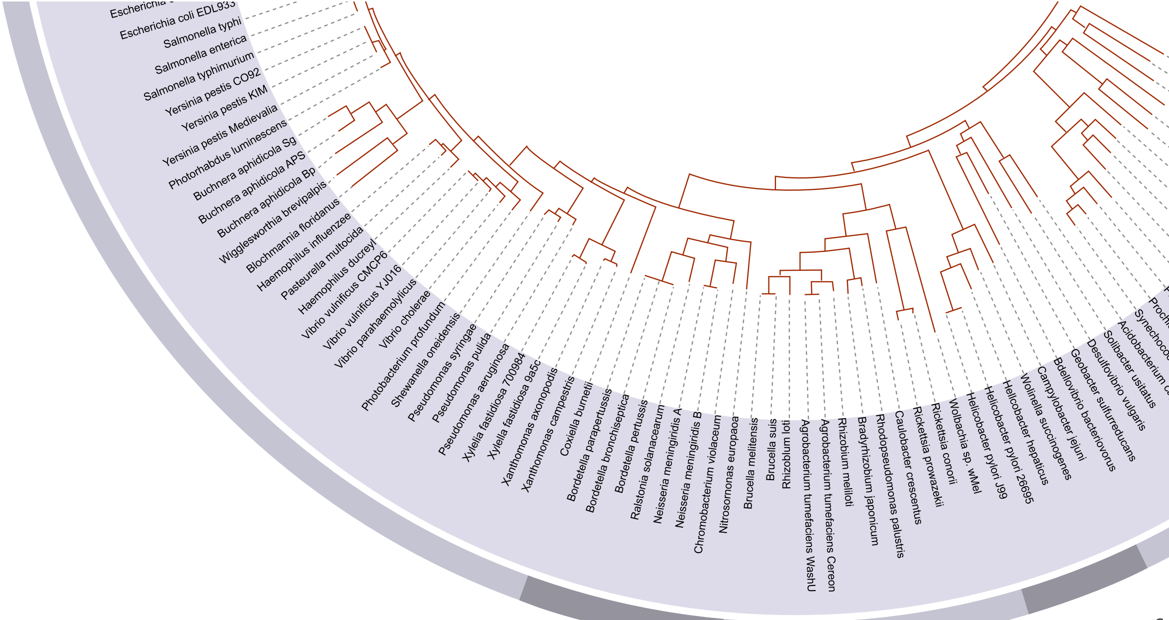


image: Ivicia Letunic and Mariana Ruiz Villarreal, via the tool iTOL (Interative Tree of Life), via Wikipedia

natural trees: Indo-European languages

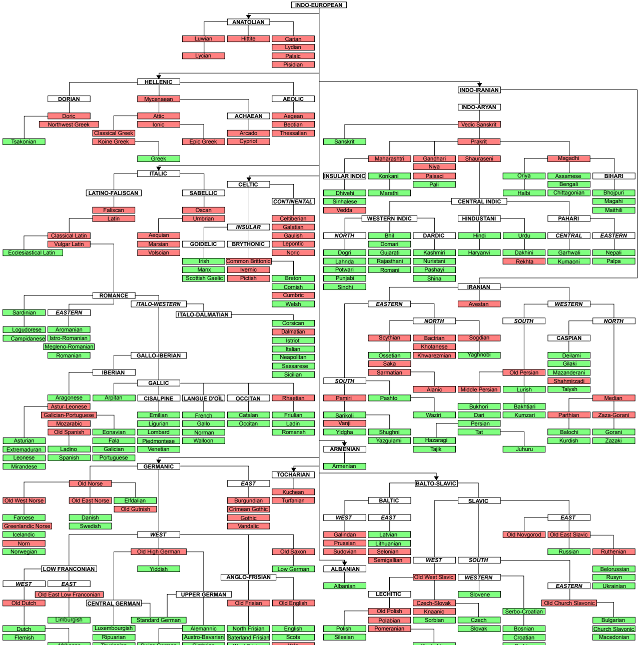
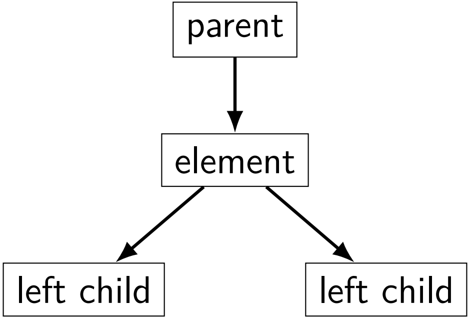


image: via Wikipedia/Mandrak

list to tree

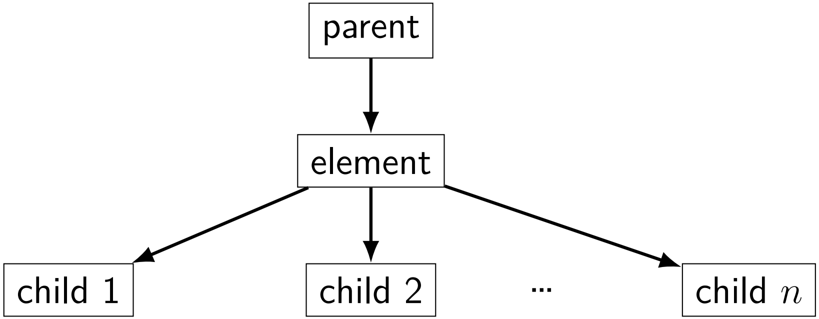
list — up to 2 related nodes

binary tree — up to 3 related nodes (list is special-case)

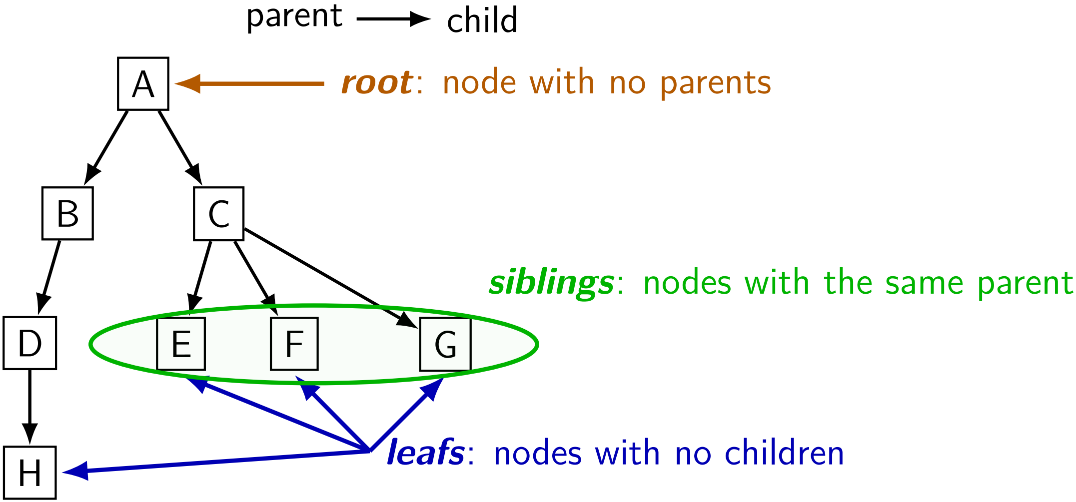


more general trees

tree — any number of relationships (binary tree is special case)
at most one parent

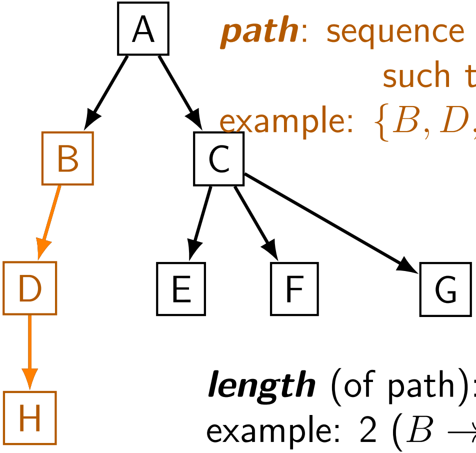


tree terms (1)



parent → child
A
root: node with no parents
B
C
siblings: nodes with the same parent
D
E
F
G
leafs: nodes with no children
H

paths and path lengths

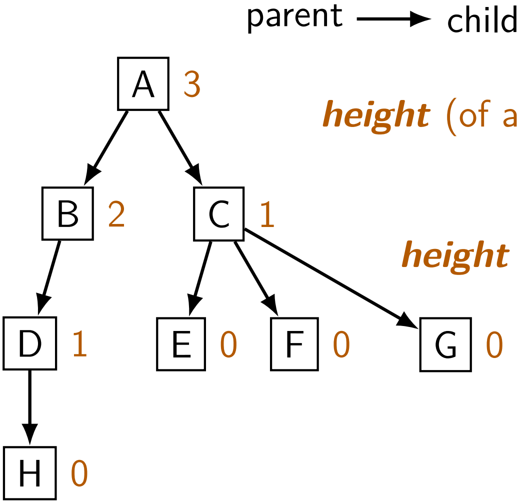


path: sequence of nodes $n_1, n_2, \ldots, n_k$
such that n_i is parent of n_{i+1}
example: $\{B, D, H\}$

length (of path): number of *edges* in path
example: 2 ($B \to D$ and $D \to H$)

internal path length: sum of depth of nodes
example: $6 = 1 + 2 + 3$

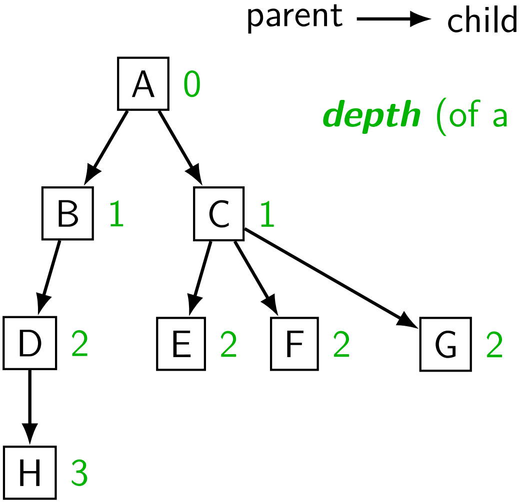
tree/node height



height (of a node): length of longest path to leaf

height (of a tree): height of tree's root
(this example: 3)

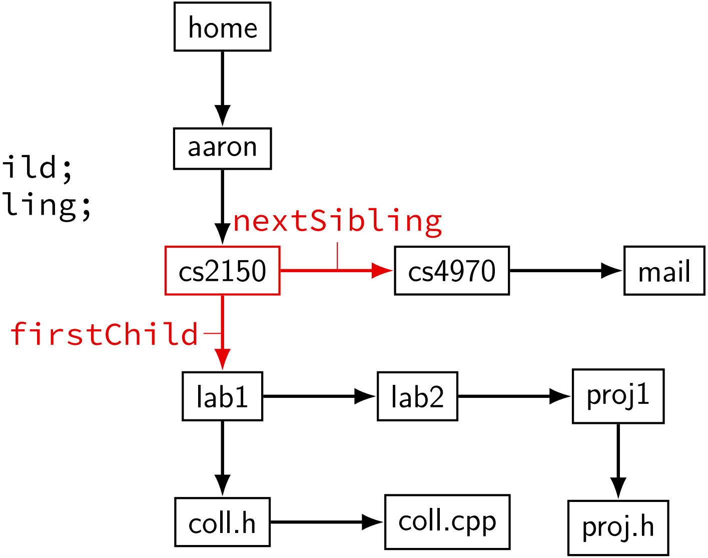
tree/node depth



depth (of a node): length of path to root

first child/next sibling

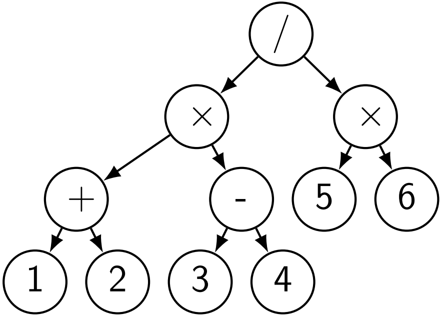
```
class TreeNode {
  private:
    string element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
  public:
    ...
};
```



another tree representations

```
class TreeNode {
  private:
    string element;
    vector<TreeNode *> children;
  public:
    ...
};

// and more --- see when we talk about graphs
```

tree traversal



pre-order: `/ * + 1 2 - 3 4 * 5 6`
in-order: `(((1+2) * (3-4)) / (5*6))` (parenthesis optional?)
post-order: `1 2 + 3 4 - * 5 6 * /`

pre/post-order traversal printing

(this is pseudocode)

```
TreeNode::printPreOrder() {
    this->print();
    for each child c of this:
        c->printPreOrder()
}

TreeNode::printPostOrder() {
    for each child c of this:
        c->printPostOrder()
    this->print();
}
```

in-order traversal printing

(this is pseudocode)

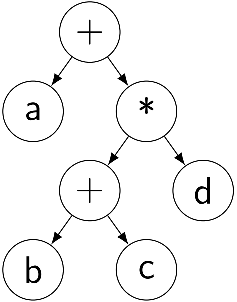
```
BinaryTreeNode::printInOrder() {
    if (this->left)
        this->left->printInOrder();
    cout << this->element << "␣";
    if (this->right)
        this->right->printInOrder();
}
```

post-order traversal counting

(this is pseudocode)

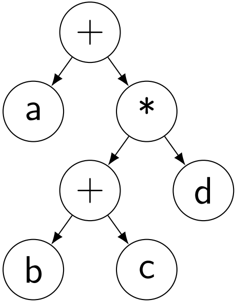
```
int numNodes(TreeNode *tnode) {
  if ( tnode == NULL )
      return 0;
  else {
      sum=0;
      for each child c of tnode
          sum += numNodes(c);
      return 1 + sum;
  }
}
```

expression tree and traversals




```
(a + ((b + c) * d))
```

expression tree and traversals



infix: `(a + ((b + c) * d))`

postfix: `a b c + d * +`

prefix: `+ a * + b c d`

postfix expression to tree

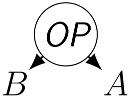
use a stack of trees

number $n \rightarrow$ push(n)

operator $OP \rightarrow$

pop into A, B; then

push

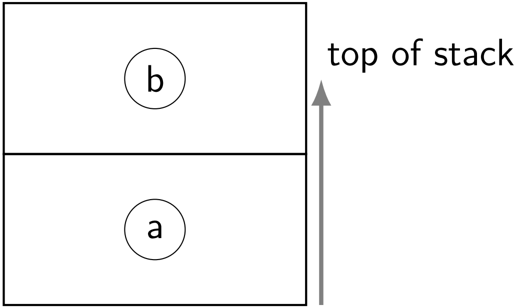


example

a b + c d e + * *

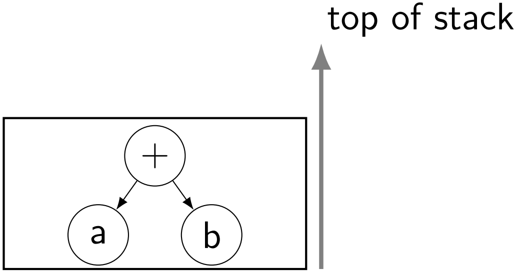
example

a b + c d e + * *



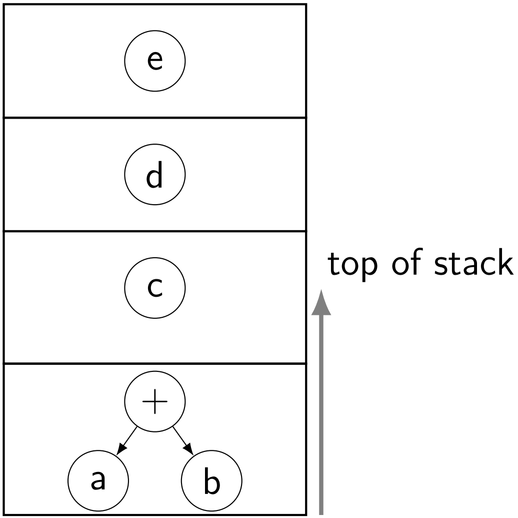
example

a b + c d e + * *



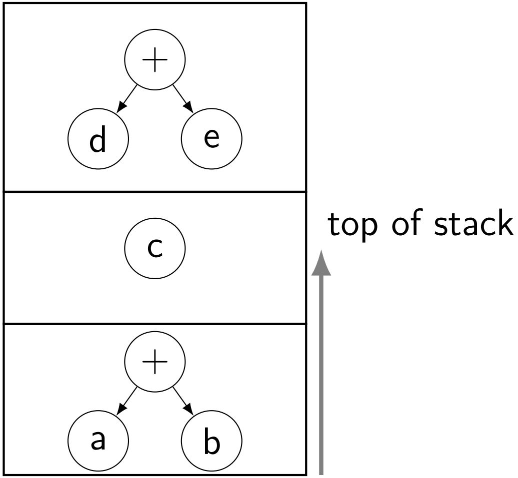
example

a b + c d e + * *



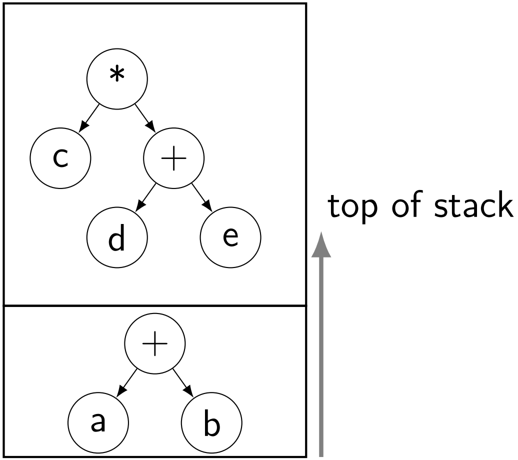
example

a b + c d e + * *



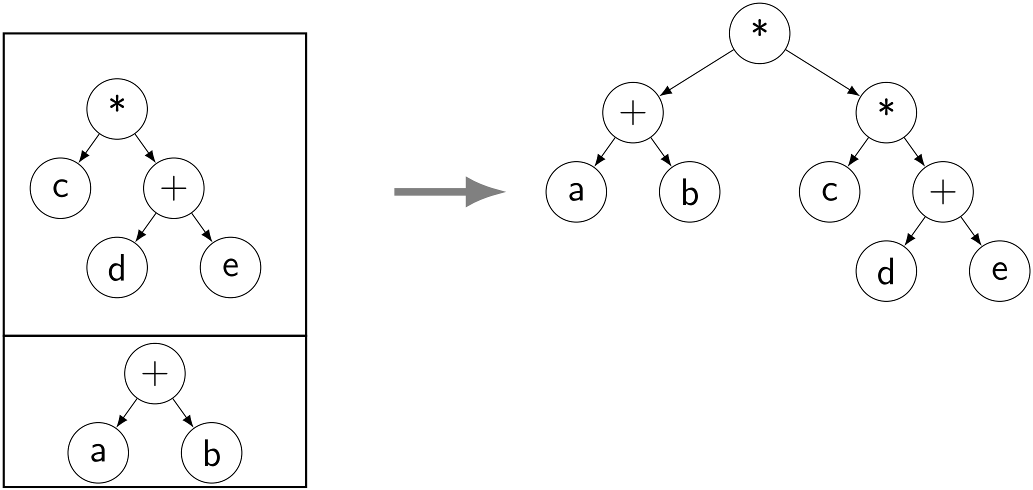
example

a b + c d e + * *



example

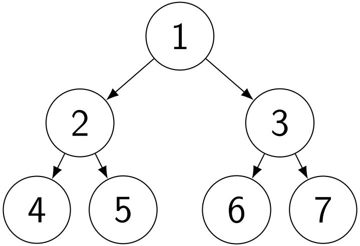
a b + c d e + * *

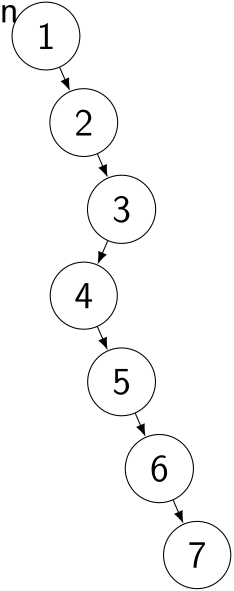


binary trees

all nodes have *at most* 2 children

```
class BinaryNode {
  ...
  int element;
  BinaryNode *left;
  BinaryNode *right;
};
```

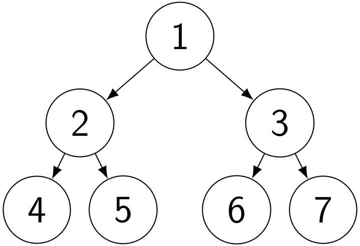


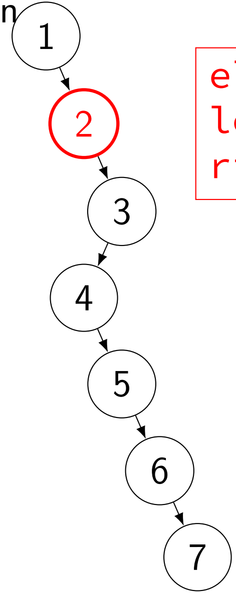


binary trees

all nodes have *at most* 2 children

```
class BinaryNode {
  ...
  int element;
  BinaryNode *left;
  BinaryNode *right;
};
```



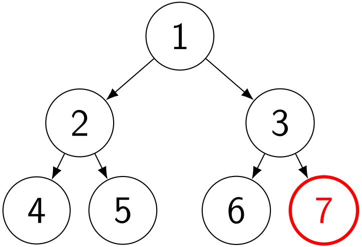


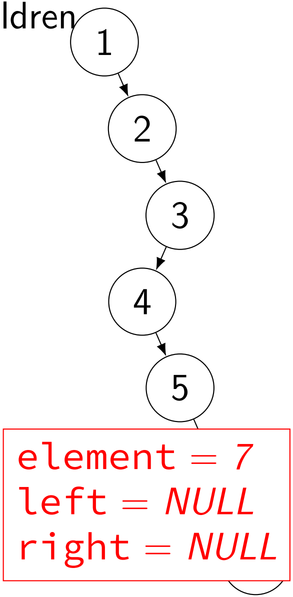
element = *2*
left = *NULL*
right = *addr of node 3*

binary trees

all nodes have *at most* 2 children

```
class BinaryNode {
  ...
  int element;
  BinaryNode *left;
  BinaryNode *right;
};
```





binary search trees

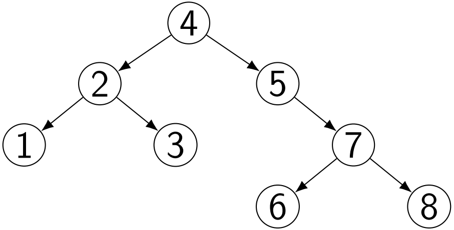
binary tree **and**...

each node has a *key*

for each node:

keys in node's left subtree are less than node's

keys in node's right subtree are greater than node's



binary search trees

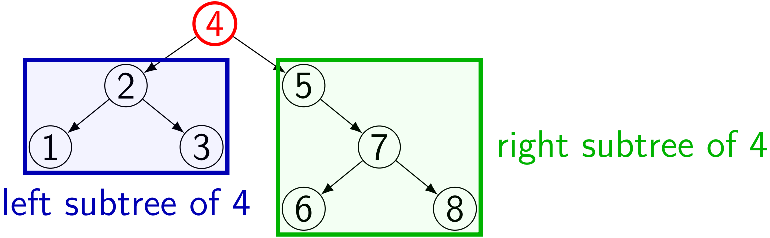
binary tree **and**...

each node has a *key*

for each node:

keys in node's left subtree are less than node's

keys in node's right subtree are greater than node's



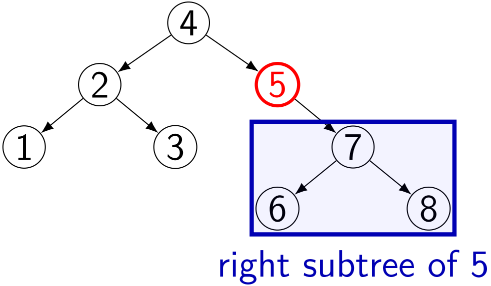
binary search trees

binary tree **and**...

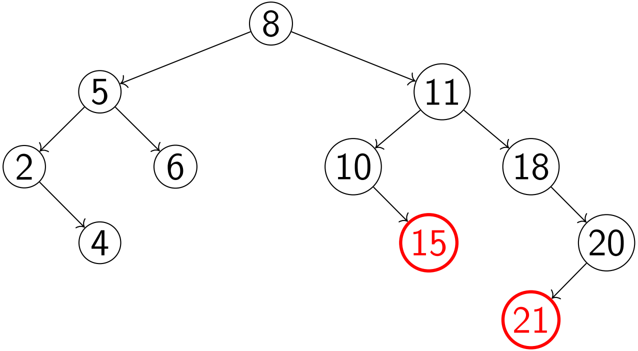
each node has a *key*

for each node:

keys in node's left subtree are less than node's
keys in node's right subtree are greater than node's



not a binary search tree



binary search tree versus binary tree

binary search trees are a kind of binary tree

...but — often people say “binary tree” to mean “binary search tree”

BST: find

(pseudocode)

```
find(node, key) {
    if (node == NULL)
        return NULL;
    else if (key < node->key)
        return find(node->left, key)
    else if (key > node->key)
        return find(node->right, key)
    else // if (key == node->key)
        return node;
}
```

BST: insert

(pseudocode)

```
insert(Node *&node, key) {
    if (node == NULL)
        node = new BinaryNode(key);
    else if (key < node->key)
        insert(node->left, key);
    else if (key < root->key)
        insert(node->right, key);
    else // if (key > root->key)
        ; // duplicate -- no new node needed
}
```

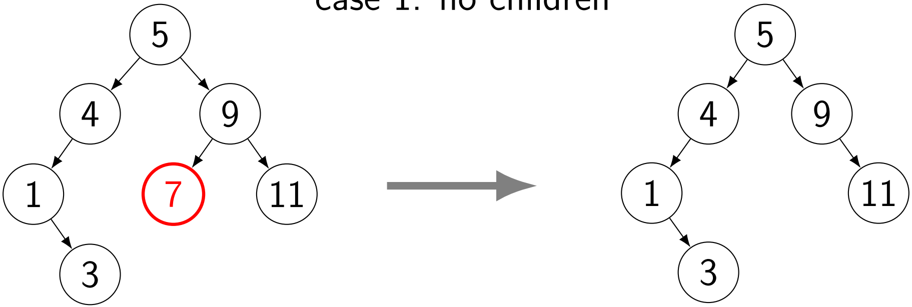
BST: findMin

(pseudocode)

```
findMin(Node *node, key) {
    if (node->left == NULL)
        return node;
    else
        insert(node->left, key);
}
```

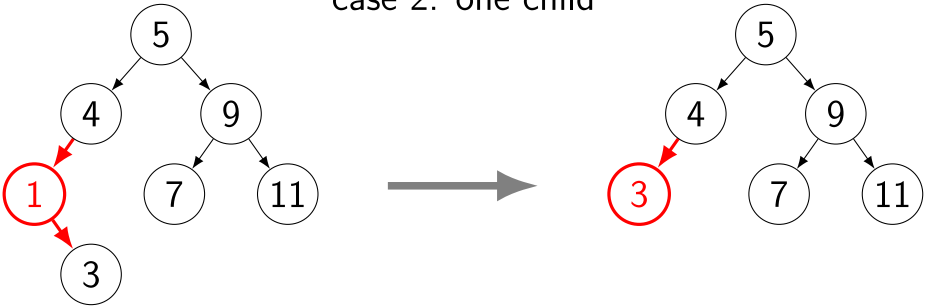
BST: remove (1)

case 1: no children



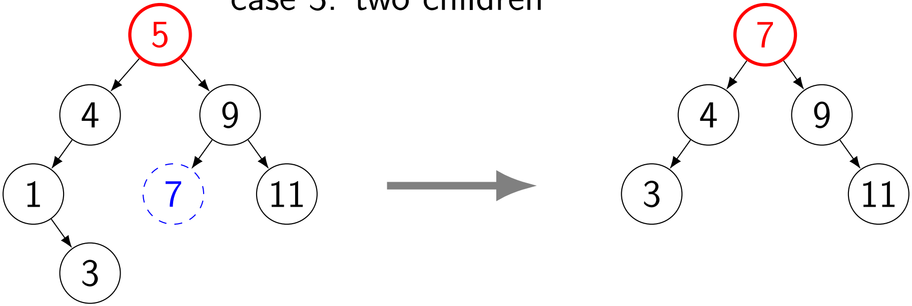
BST: remove (2)

case 2: one child



BST: remove (3)

case 3: two children

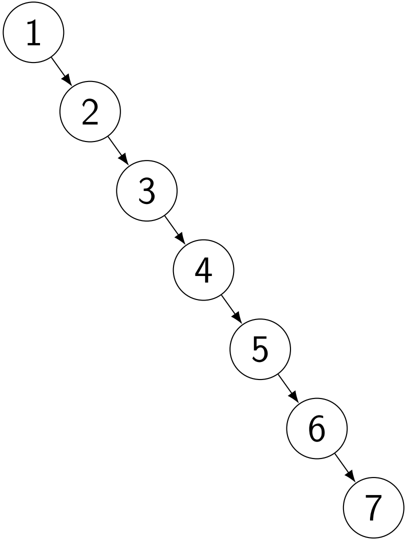


replace with minimum of right subtree

(alternately: maximum of left subtree, ...)

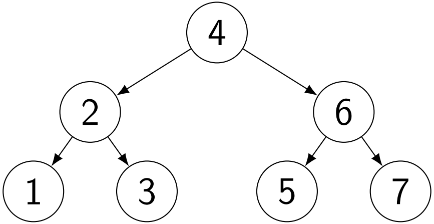
binary tree: worst-case height

n-node BST: worst-case height/depth $n - 1$



binary tree: best-case height

height h: at most $2^{h+1} - 1$ nodes



binary tree: proof best-case height is possible

proof **by induction**: can have $2^{h+1} - 1$ nodes in h-height tree

$\mathbf{h = 0}$: $h = 0$: exactly one node; $2^{h+1} - 1 = 1$ nodes

$\mathbf{h = k \to h = k + 1}$:

start with *two copies* of a maximum tree of height k

create a new tree as follows:

- create a new root node
- add edges from the root node to the roots of the copies

the height of this new tree is $k + 1$

- path of length k in old tree + either new edge

the number of nodes is

$2(2^{k+1} - 1) + 1 = 2^{k+1+1} - 2 + 1 = 2^{k+1+1} - 1$

binary tree: best-case height is best

(informally)

property of trees in root:

- except for the leaves, every node in tree has 2 children

no way to add nodes without increasing height

- add below leaf — longer path to root — longer height
- add above root — every old node has longer path to root

binary tree height formula

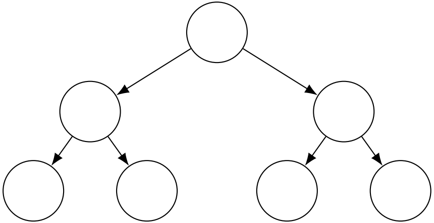
n: number of nodes

h: height

$$\begin{aligned} n+1 &\leq 2^{h+1} \\ \log_2(n+1) &\leq \log_2\left(2^{h+1}\right) \\ \log(n+1) &\leq h+1 \\ h &\geq \log_2(n+1)-1 \end{aligned}$$

shortest tree of n nodes: $\sim \log_2(n)$ height

perfect binary trees



a binary tree is perfect if

all leaves have same depth

all nodes have zero children (leaf) or two children

exactly the trees that achieve $2^{h+1} - 1$ nodes

AVL animation tool

`http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html`

AVL tree idea

AVL trees: one of many balanced trees —

search tree *balanced* to keep height $\Theta(\log n)$
avoid "tree is just a long linked list" scenarios

gaurentees $\Theta(\log n)$ for find, insert, remove

AVL $=$ Adelson-Velskii and Landis

AVL gaurentee

the height of the left and right subtrees of *every node* differs by at most one

AVL state

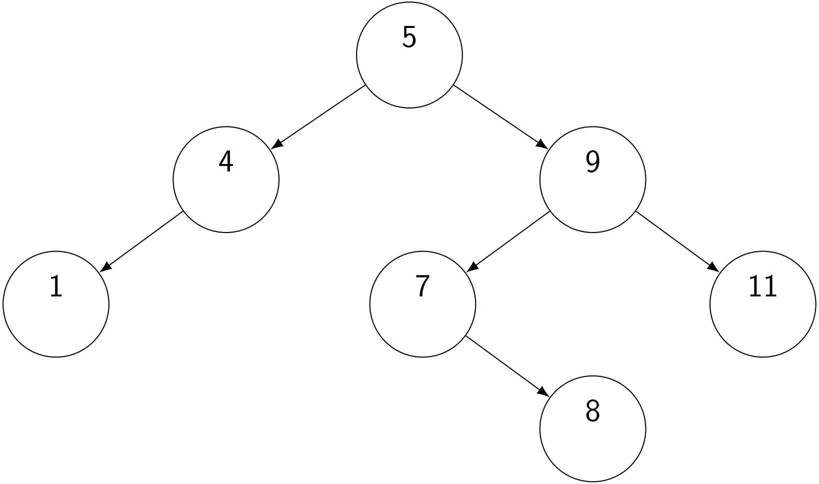
normal binary search tree stuff:

- data; and left, right, parent pointers

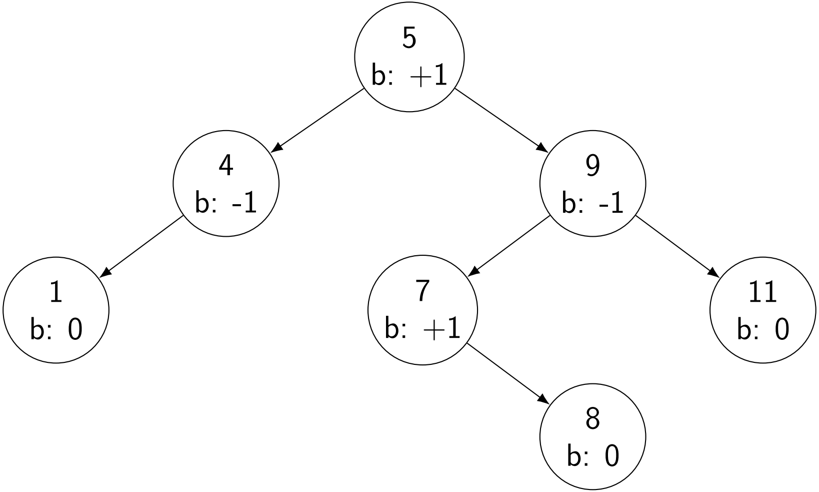
additional AVL stuff:

- height of right subtree minus height of left subtree
  - called "balanced factor"
  - -1, 0, +1
- (kept up to date on insert/delete — computing on demand is too slow)

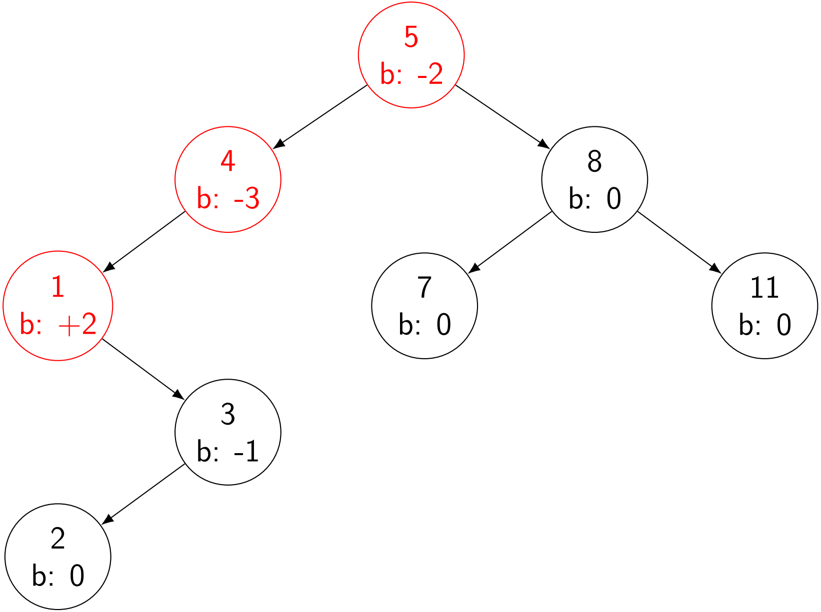
example AVL tree



example AVL tree



example non-AVL tree



AVL tree algorithms

find — exactly the same as binary search tree

just ignore balance factors

insert — two extra steps:

update balance factors

"fix" tree if it became unbalanced

AVL tree algorithms

find — exactly the same as binary search tree

just ignore balance factors

insert — two extra steps:

update balance factors

"fix" tree if it became unbalanced

runtime for both $\Theta(d)$ where d is depth of node found/inserted

max balance factor ± 1 at root

max depth of node is $\Theta(\log_2 n + 1) = \Theta(\log n)$

AVL insertion cases

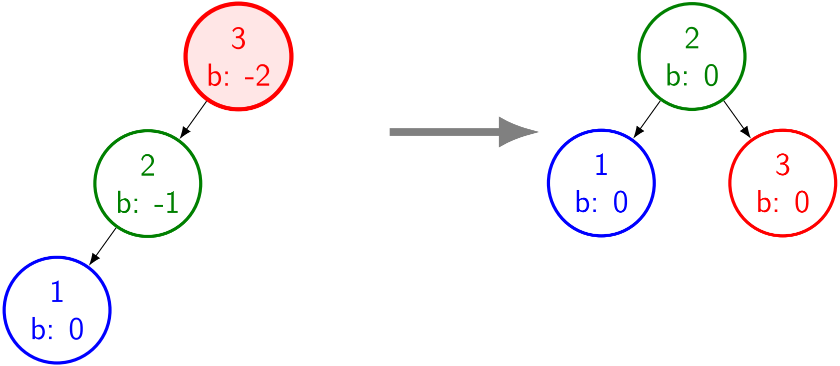
simple case: tree remains balanced

otherwise:
let x be deepest imbalanced node ($+2/$-2 balance factor)

- insert in left subtree of left child of x: single rotation right
- insert in right subtree of right child of x: single rotation left
- insert in right subtree of left child of x: double left-right rotation
- insert in left subtree of right child of x: double right-left rotation

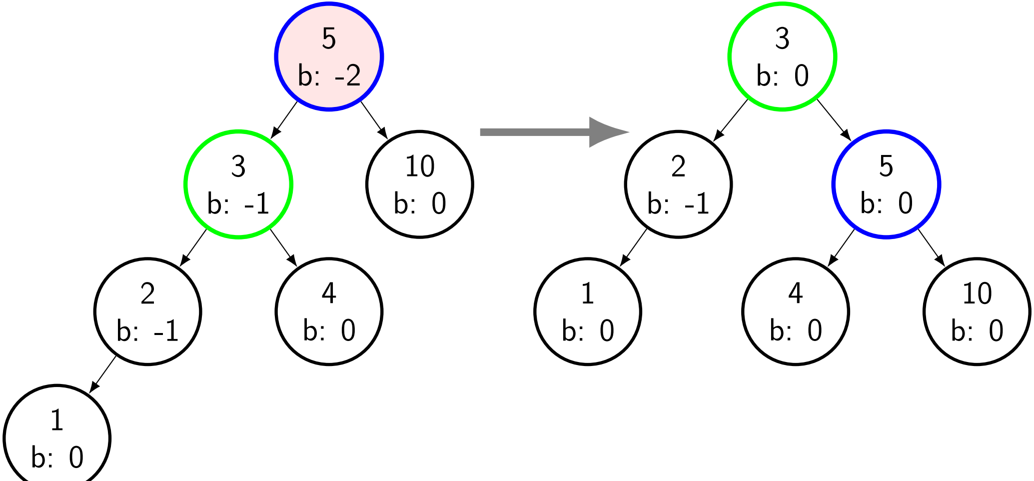
AVL: simple right rotation

just inserted 0
unbalanced root becomes new left child



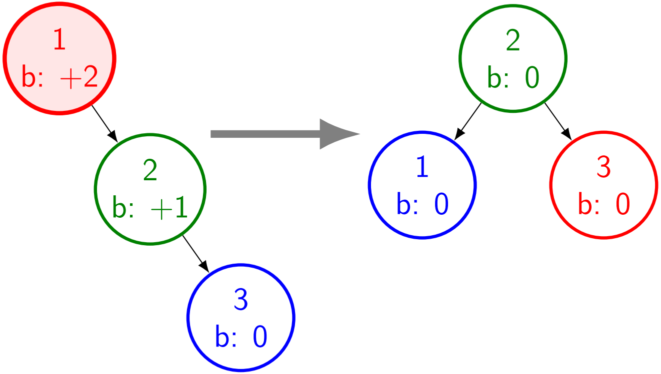
AVL: less simple right rotation (1)

just inserted 0
unbalanced root becomes new left child



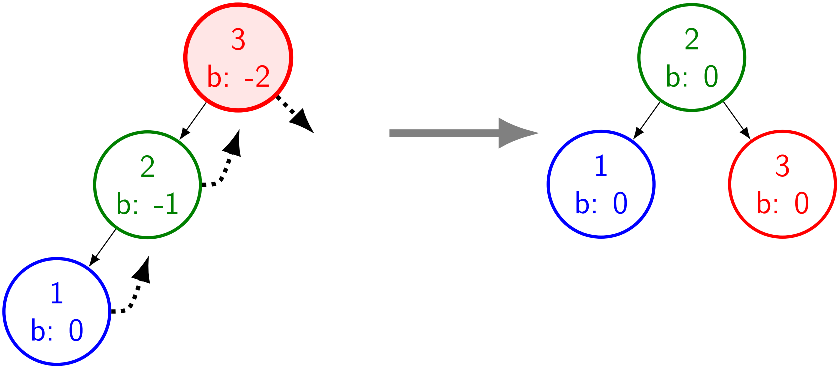
AVL: simple left rotation

just inserted 1
deepest unbalanced node is 3

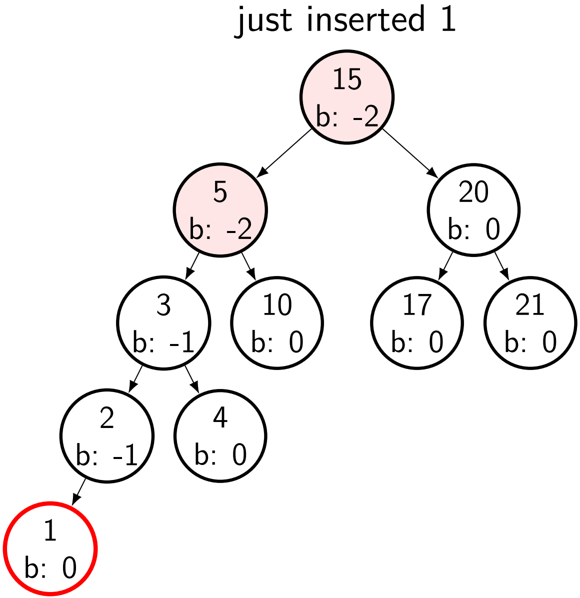


AVL rotation: up and down

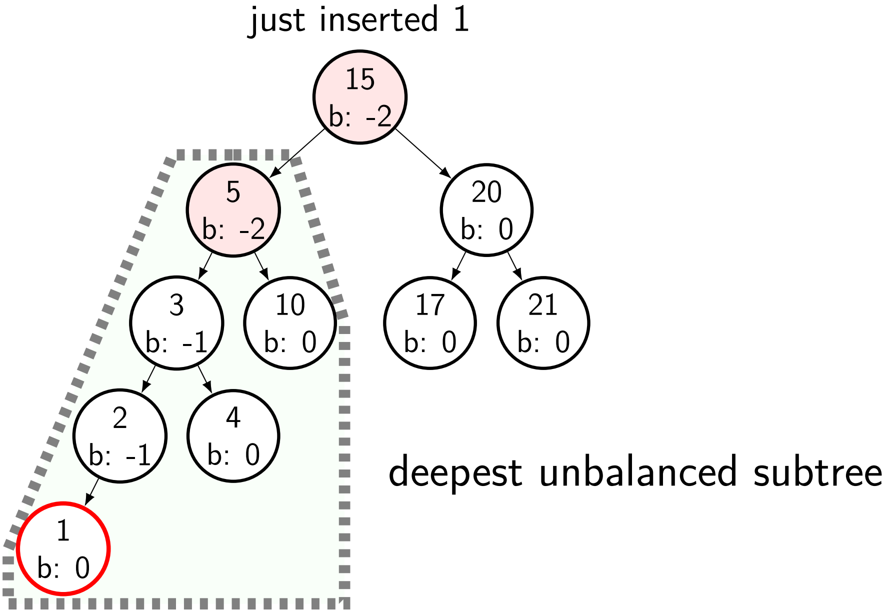
at least one node moves up (this case: 1 and 2)
at least one node moves down (this case: 3)



AVL: less simple right rotation (2)

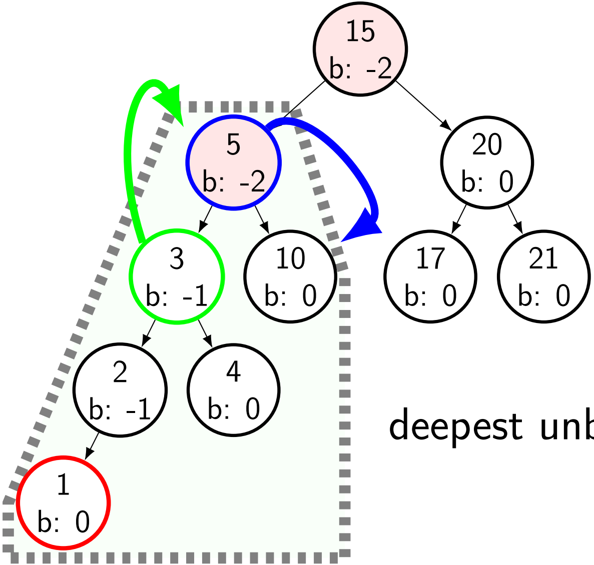


AVL: less simple right rotation (2)



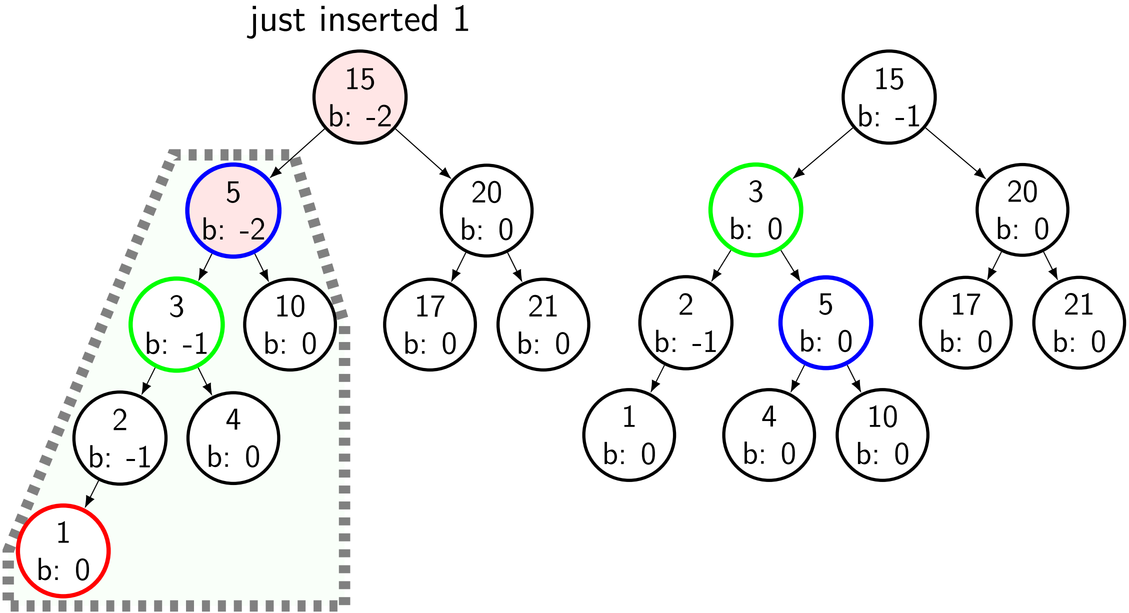
AVL: less simple right rotation (2)

just inserted 1

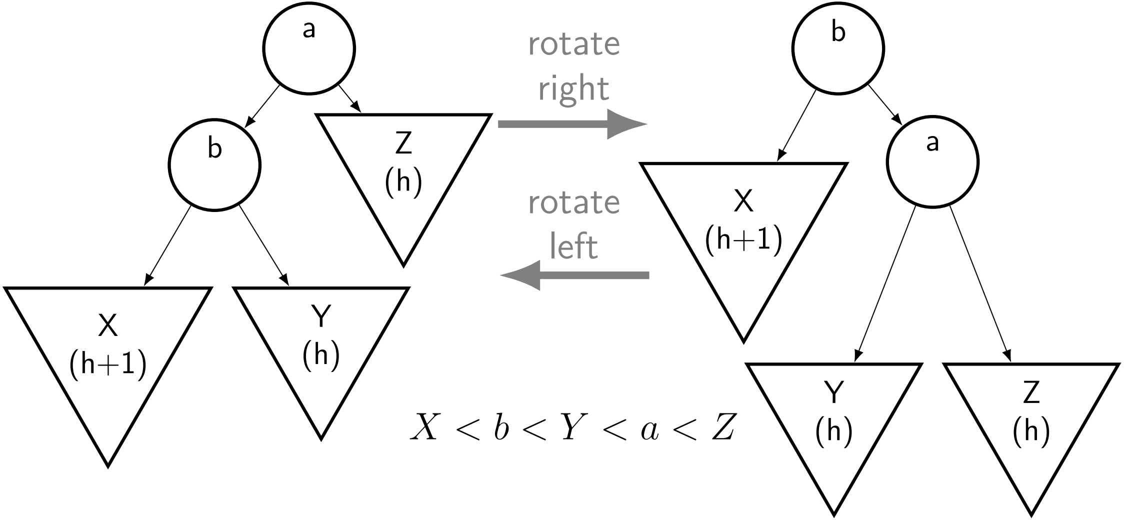


deepest unbalanced subtree

AVL: less simple right rotation (2)

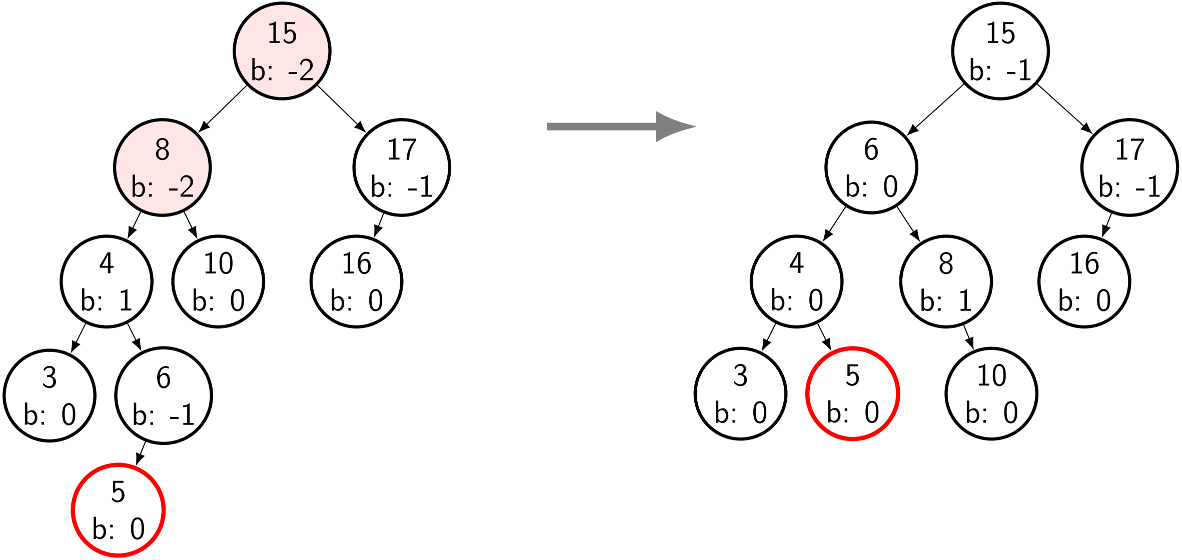


general single rotation



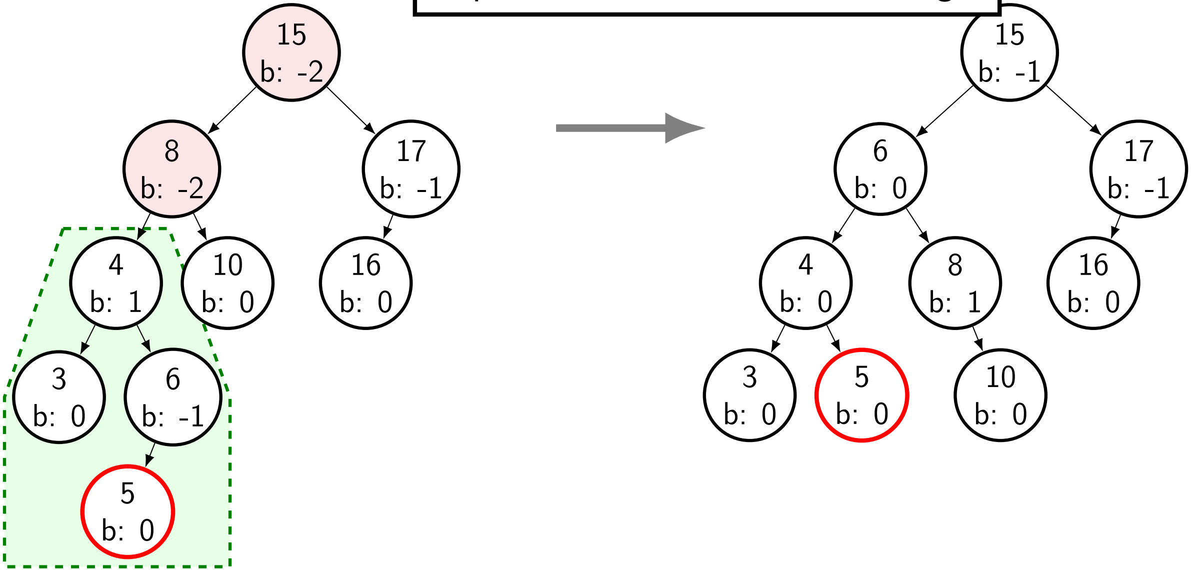
$$X < b < Y < a < Z$$

double rotation



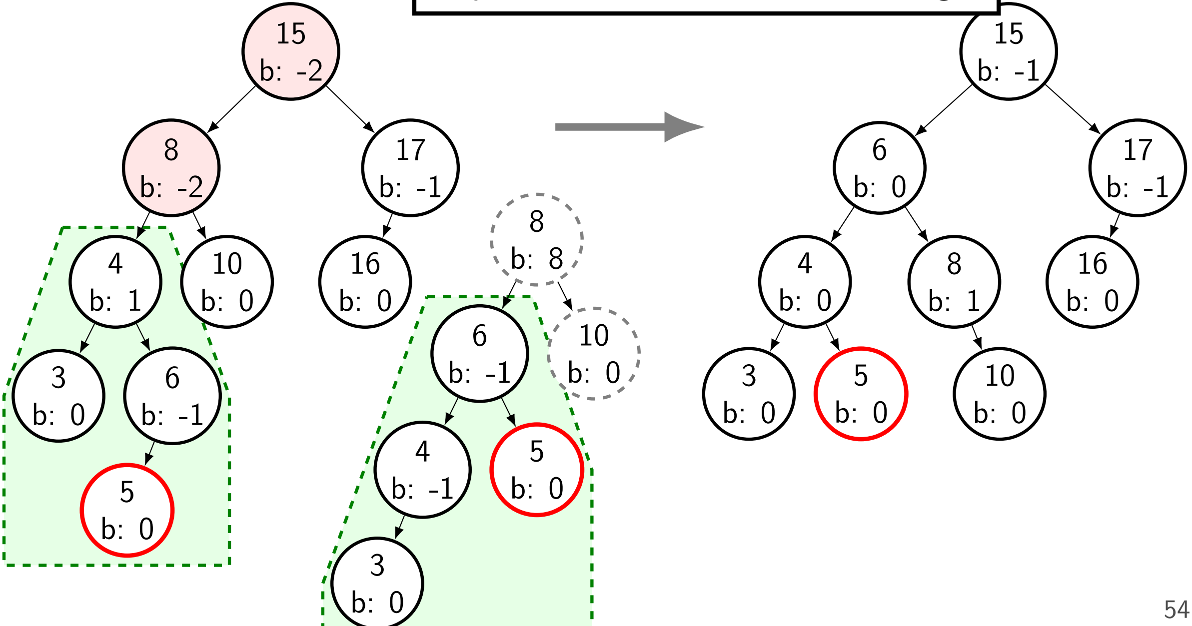
double rotation

step 1: rotate subtree left
step 2: rotate imbalanced tree right



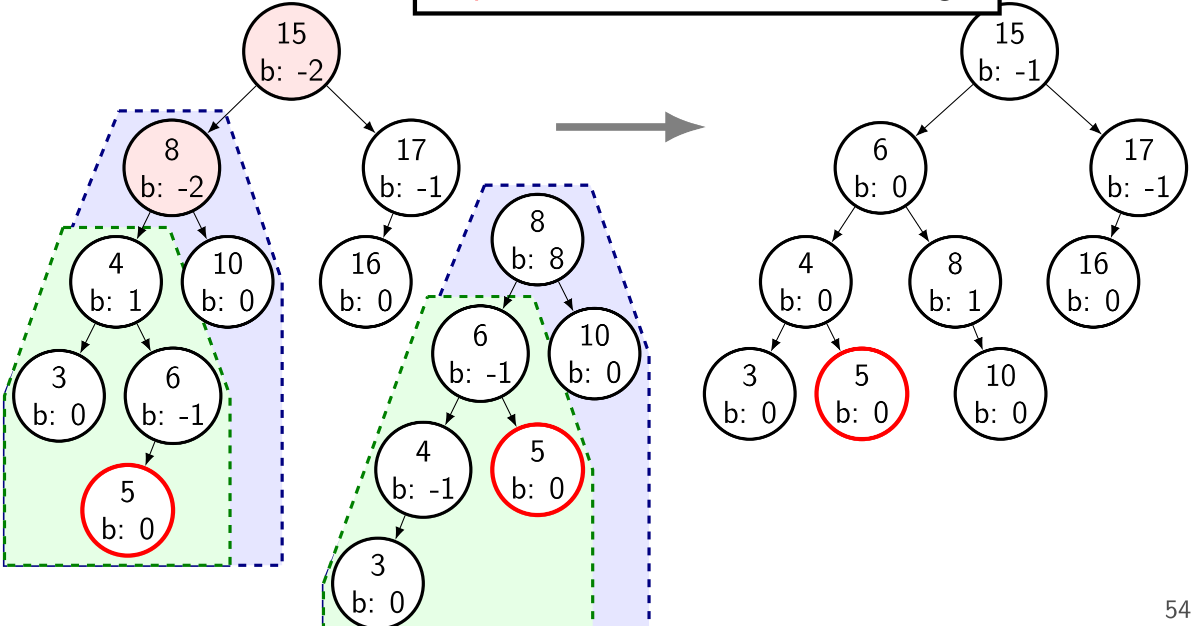
double rotation

step 1: rotate subtree left
step 2: rotate imbalanced tree right

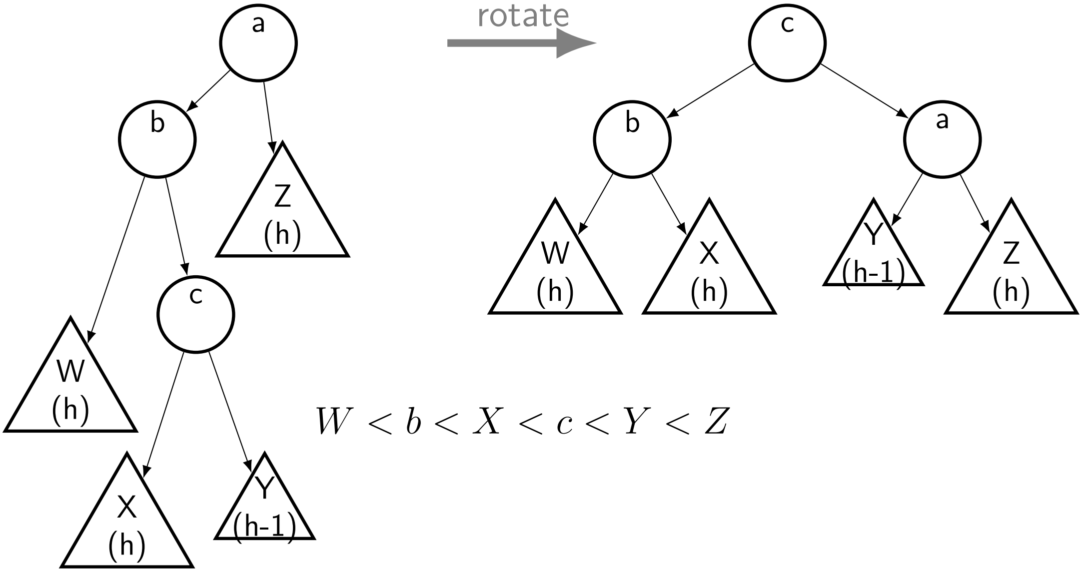


double rotation

step 1: rotate subtree left
step 2: rotate imbalanced tree right

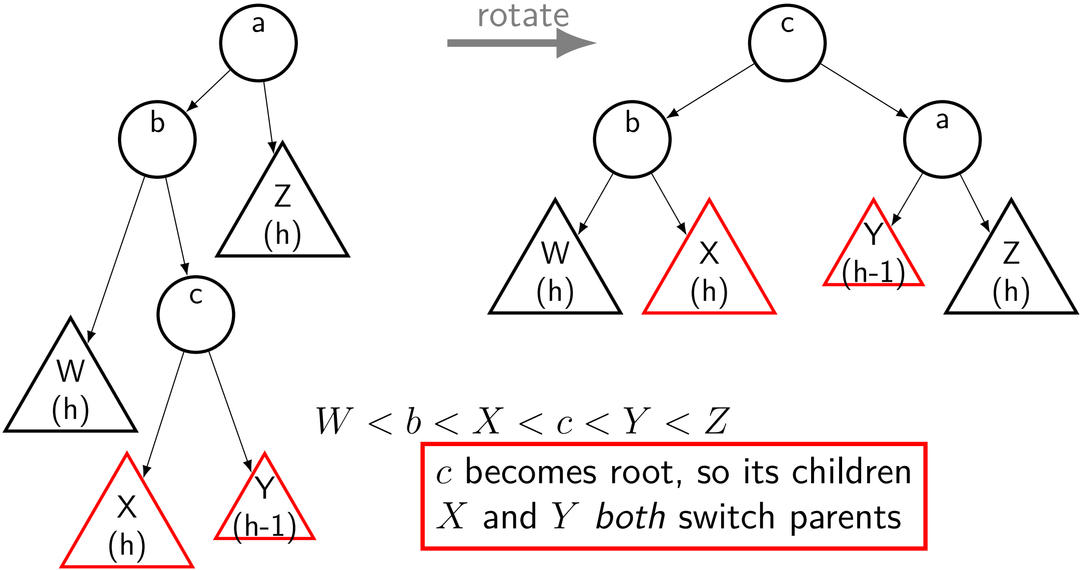


general double rotation



$$W < b < X < c < Y < Z$$

general double rotation



$W < b < X < c < Y < Z$

c becomes root, so its children X and Y *both* switch parents

double rotation names

sometimes "double left"

    first rotation left, or second?

us: "double left-right"

    rotate child tree left
    rotate parent tree right

"double right-left"

    rotate child tree right
    rotate parent tree left

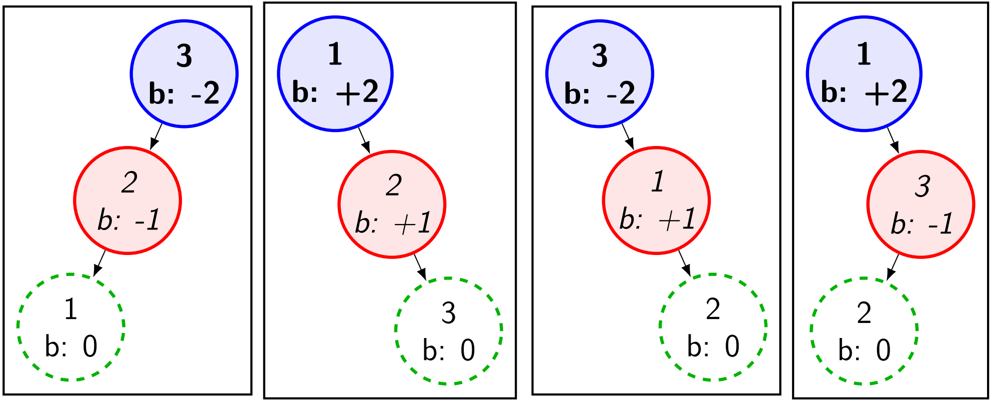
AVL insertion cases

simple case: tree remains balanced

otherwise:
let x be deepest imbalanced node ($+2$/-2 balance factor)

- insert in left subtree of left child of x: single rotation right
- insert in right subtree of right child of x: single rotation left
- insert in right subtree of left child of x: double left-right rotation
- insert in left subtree of right child of x: double right-left rotation

AVL insert cases (revisited)



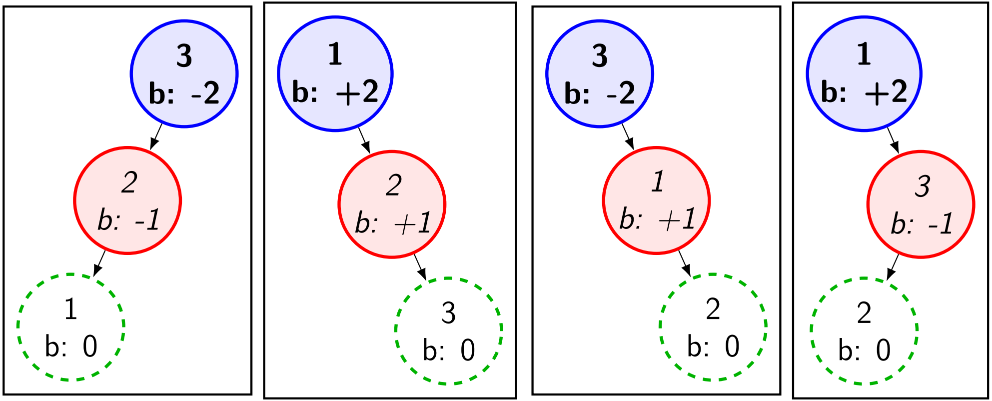
single left

single right

left-right

right-left

AVL insert cases (revisited)



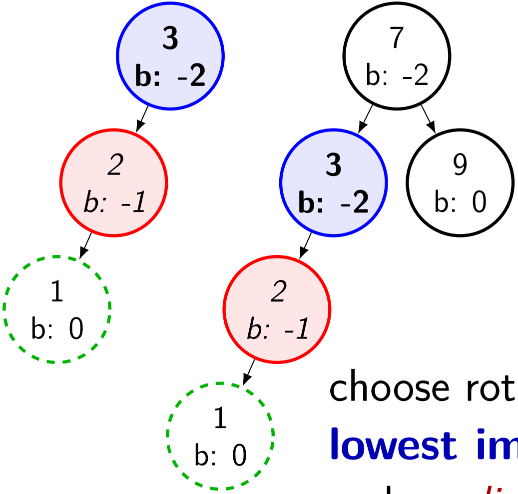
single left | single right | left-right | right-left

choose rotation based on **lowest imbalanced node**

and on *direction of insertion*

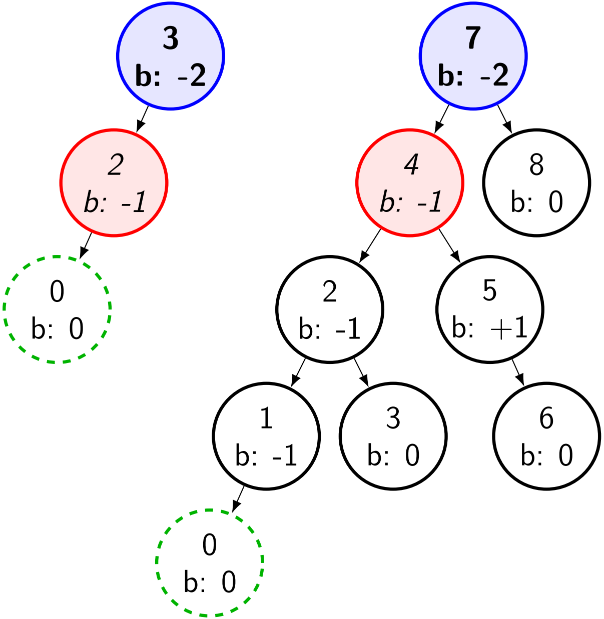
(inserted node is green+dashed)

AVL insert case: detail (1)



choose rotation based on
lowest imbalanced node
and on *direction of insertion*
(inserted node is green+dashed)

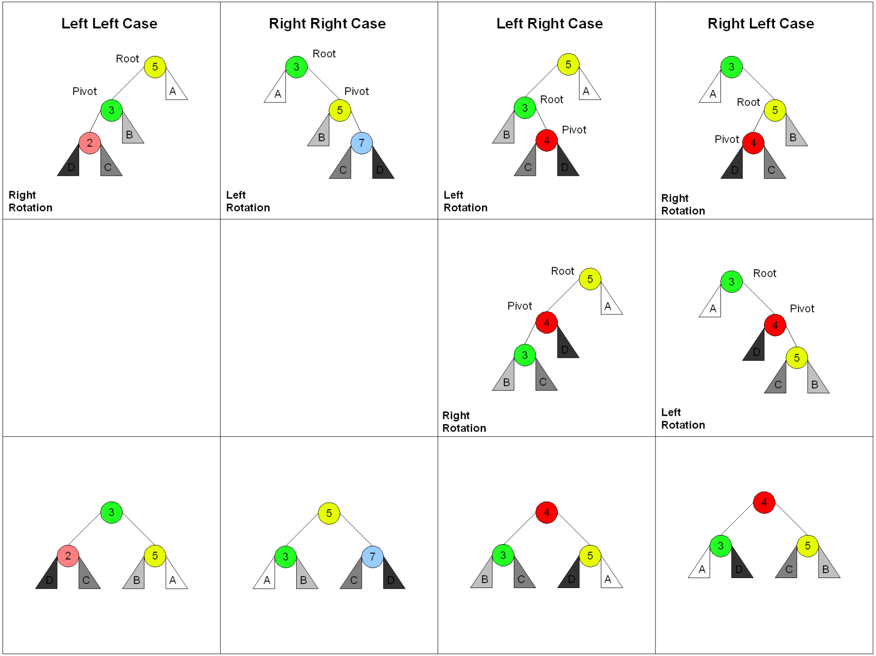
AVL insert case: detail (2)



choose using **lowest imbalanced node** and on *direction of insertion* (inserted node is green+dashed)

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



AVL tree: runtime

worst depth of node: $\Theta(\log_2 n + 2) = \Theta(\log n)$

find: $\Theta(\log n)$

worst case: traverse from root to worst depth leaf

insert: $\Theta(\log n)$

worst case: traverse from root to worst depth leaf
then back up (update balance factors)
then perform constant time rotation

remove: $\Theta(\log n)$

left as exercise (similar to insert)

print: $\Theta(n)$

visit each of n nodes

other types of trees

many kinds of *balanced trees*

not all binary trees

different ways of tracking balance factors, etc.

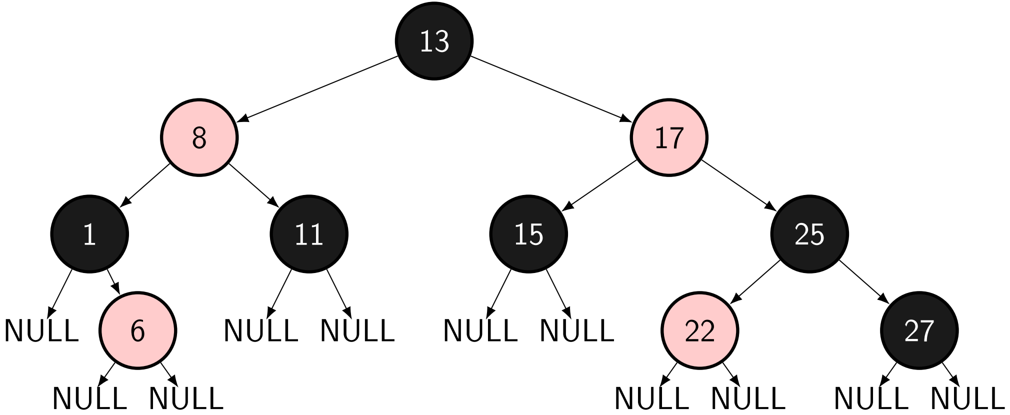
different ways of doing tree rotations or equivalent

red-black trees

each node is **red** or **black**

null leafs considered nodes to aid analysis (still null pointers...)

rules about when nodes can be red/black gaurentee maximum depth



red-black tree rules

root is **black**

counting null pointers as nodes, leaves are **black**

a **red** node's children are **black**

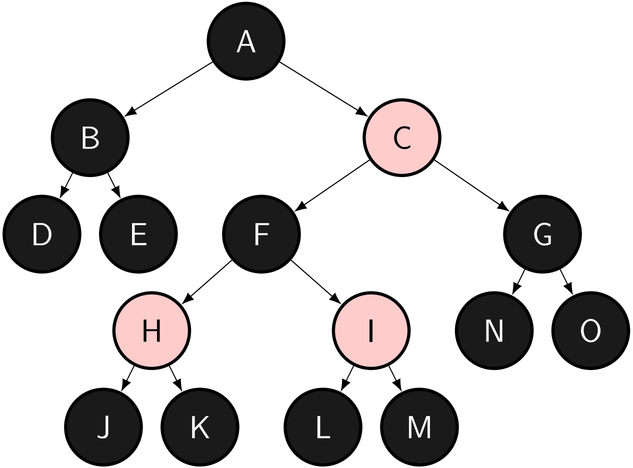
$\rightarrow$ a **red** node's parents are **black**

every simple path from node to leaf under it contains same number of black nodes

(property holds regardless of whether null pointers are considered nodes)

worst red-black tree imbalance

same number of black nodes on paths to leaves
$\rightarrow$ factor of 2 imbalance max



red-black insert

default: insert as **red**, but…

(1) if new node is root: color **black**

(2) if parent is black: keep child **red**

(3) if parent and uncle is **red**: adjust several colors

(4) if parent is **red**, uncle is **black**, new node is right child

perform a rotation, then go to case 5

(5) if parent is **red**, uncle is **black**, new node is left child

perform a rotation

red-black insert

default: insert as **red**, but...

(1) if new node is root: color **black**

(2) if parent is black: keep child **red**

(3) if parent and uncle is **red**: adjust several colors

(4) if parent is **red**, uncle is **black**, new node is right child

perform a rotation, then go to case 5

(5) if parent is

perform a

property: "children of **red** node are **black**"
no change in # of **black** nodes on paths

red-black insert

default: insert as **red**, but...

(1) if new node is root: color **black**

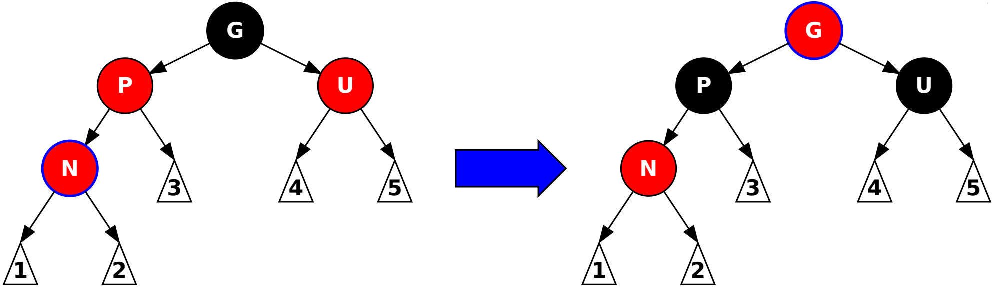
(2) if parent is black: keep child **red**

(3) if parent and uncle is **red**: adjust several colors

(4) if parent is **red**, uncle is **black**, new node is right child
perform a rotation, then go to case 5

(5) if parent is **red**, uncle is **black**, new node is left child
perform a rotation

case 3: parent, uncle are red



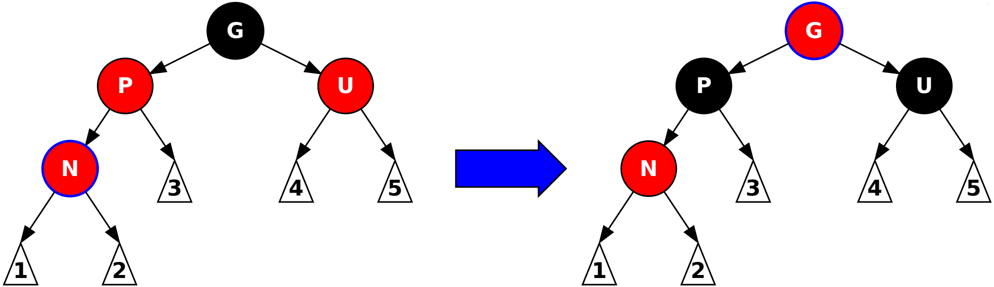
make grandparent **red**, parent and uncle **black**

(property: every path to leaf has same number of black nodes)

just swapped grandparent and parent/uncle in those paths

image: Wikipedia/Abloomfi

case 3: parent, uncle are red



make grandparent **red**, parent and uncle **black**

(property: every path to leaf has same number of black nodes)

just swapped grandparent and parent/uncle in those paths

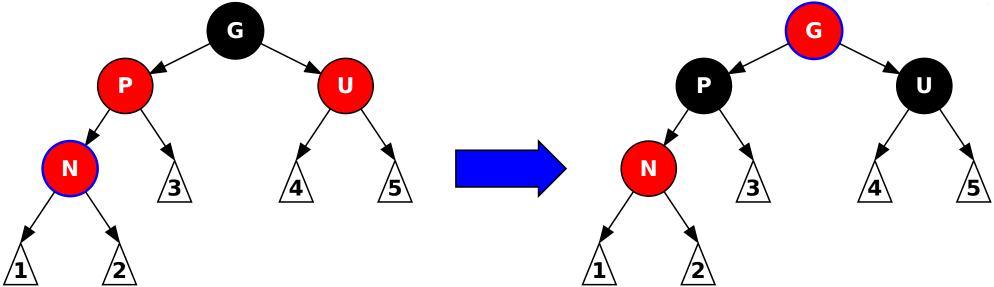
but...what if grandparent's parent is red?

(property: children of red node are black)

solution: recurse to the grandparent, as if it was just inserted

image: Wikipedia/Abloomfi

case 3: parent, uncle are red



make grandparent **red**, parent and uncle **black**

(property: every path to leaf has same number of black nodes)

just swapped grandparent and parent/uncle in those paths

but...what if grandparent's parent is red?

(property: children of red node are black)

solution: recurse to the grandparent, as if it was just inserted

image: Wikipedia/Abloomfi

red-black insert

default: insert as **red**, but...

(1) if new node is root: color **black**

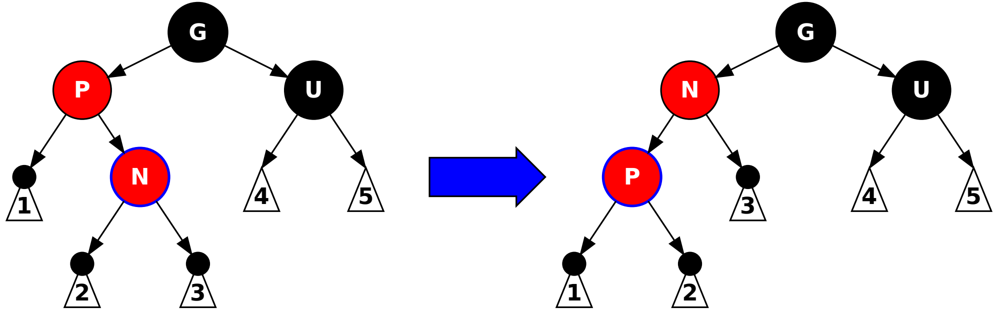
(2) if parent is black: keep child **red**

(3) if parent and uncle is **red**: adjust several colors

(4) if parent is **red**, uncle is **black**, new node is right child
perform a rotation, then go to case 5

(5) if parent is **red**, uncle is **black**, new node is left child
perform a rotation

case 4: parent red, uncle black, right child



perform left rotation on parent subtree and new node

now case 5 (but new node is P, not N)

image: Wikipedia/Abloomfi

red-black insert

default: insert as **red**, but...

(1) if new node is root: color **black**

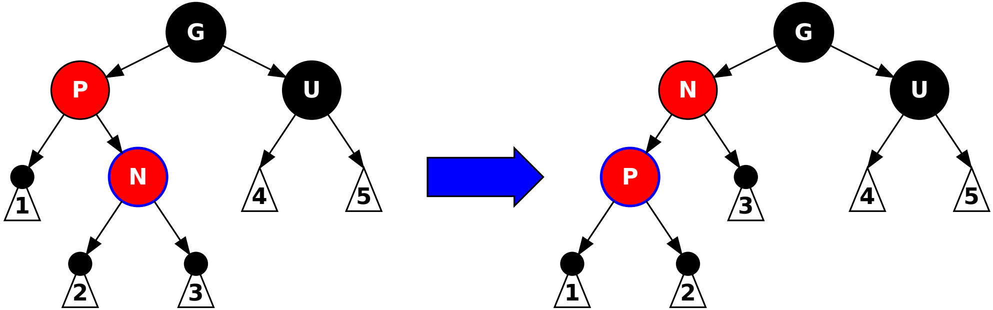
(2) if parent is black: keep child **red**

(3) if parent and uncle is **red**: adjust several colors

(4) if parent is **red**, uncle is **black**, new node is right child
 perform a rotation, then go to case 5

(5) if parent is **red**, uncle is **black**, new node is left child
 perform a rotation

case 5: parent red, uncle black, left child



perform right rotation of grandparent and parent

(property: red parent's children are black)

(property: every path to leaf has same number of black nodes)

image: Wikipedia/Abloomfi

RB-tree: removal

start with normal BST remove of x, but...

instead find next highest/lowest node y

  can choose node *with at most one child*
  ("bottom" of a left or right subtree)

swap x and y's value, then replace y with its child

several cases for color maintainence/rotations

RB tree: removal cases

N: node just replaced with child; S: its sibling; P: its parent

(1): N is new root

(2): S is **red**

(3): P, S, and S's children are **black**

(4): S and S's children are **black**

(5): S is **black**, S's left child is **red**, S's right child is **black**, N is left child of P

(6): S is **black**, S's right child is **red**, N is left child

details: see, e.g., Wikipedia article

why red-black trees?

a lot more cases...but

a lot less rotations

...because tree is kept less rigidly balanced

red-black trees end up being faster in practice

splay trees

tree that's fast for recently used nodes

self-balancing binary search tree

keeps recent nodes near the top

simpler to implement than AVL or RB trees

‘splaying’

every time node is accessed (find, insert, delete)...

“splay” tree around that node

make the node the new tree root

‘splaying’

every time node is accessed (find, insert, delete)...

“splay” tree around that node

make the node the new tree root

$\Theta(h)$ time — where h is tree height

‘splaying’

every time node is accessed (find, insert, delete)...

“splay” tree around that node

make the node the new tree root

$\Theta(h)$ time — where h is tree height

worst-case height: $\Theta(n)$ — linked-list case

amortized complexity

splay tree insert/find/delete is amortized $O(\log n)$ time

informally: average insert/find/delete: $O(\log n)$

more formally: m operations: $O(m \log n)$ time (where n: max size of tree)

splay tree pro/con

can be *faster* than AVL, RB-trees in practice

take advantage of frequently accessed items

simpler to implement

but worst case find/insert is $\Theta(n)$ time

amortized analysis: vector growth

vector insert algorithm:

- if not big enough, double capacity
- write to end of vector

amortized analysis: vector growth

vector insert algorithm:

if not big enough, double capacity

write to end of vector

doubling size — requires copying! — $\Theta(n)$ time

$\Theta(n)$ worst case per insert

but average...?

counting copies (1)

suppose initial capacity 100 + insert 1600 elements

- $100 \rightarrow 200$: 100 copies
- $200 \rightarrow 400$: 200 copies
- $400 \rightarrow 800$: 400 copies
- $800 \rightarrow 1600$: 800 copies
- total: 1500 copies

total operations: 1500 copies + 1600 writes of new elements

- about 2 operations per insert

counting copies (2)

more generally: for N inserts

about N copies + N writes

why? K to $2K$ elements: K copies

N inserts: $1 + 2 + 4 + \ldots + N/4 + N/2$ copies

(and a bit better if initial capacity isn't 1)

$\Theta(n)$ worst case

but $\Theta(n)$ time for n inserts

$\rightarrow$ $O(1)$ amortized time per insert

trees are not great for...

ordered, unsorted lists

- list of TODO tasks

being easy/simple to implement

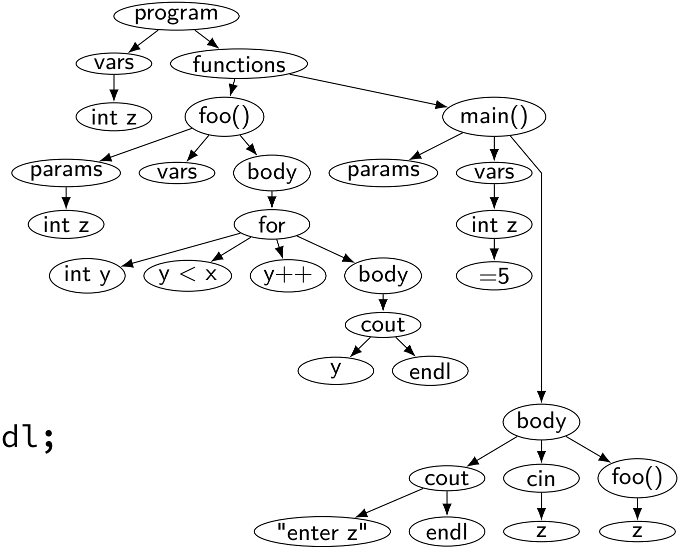
- compare, e.g., stack/queue

$\Theta(1)$ time

- compare vector
- compare hashtables (almost)

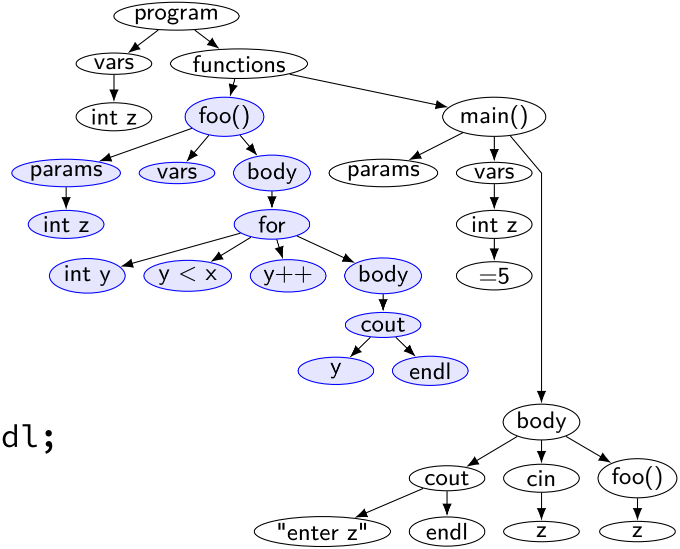
programs as trees

```
int z;

int foo (int x) {
  for (int y = 0;
         y < x;
         y++)
     cout << y << endl;
}

int main() {
  int z = 5;
  cout << "enter x" << endl;
  cin >> z;
  foo(z);
}
```

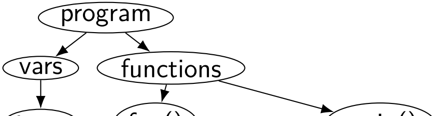


programs as trees

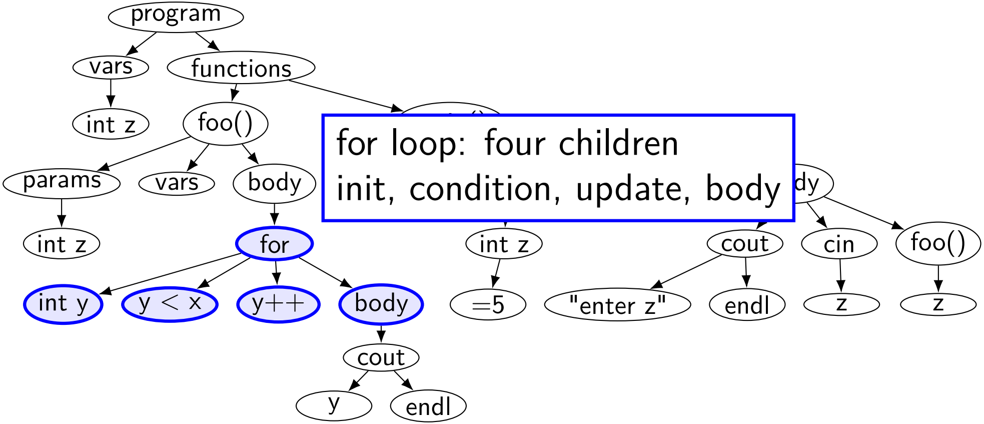
```
int z;

int foo (int x) {
  for (int y = 0;
        y < x;
        y++)
    cout << y << endl;
}

int main() {
  int z = 5;
  cout << "enter x" << endl;
  cin >> z;
  foo(z);
}
```



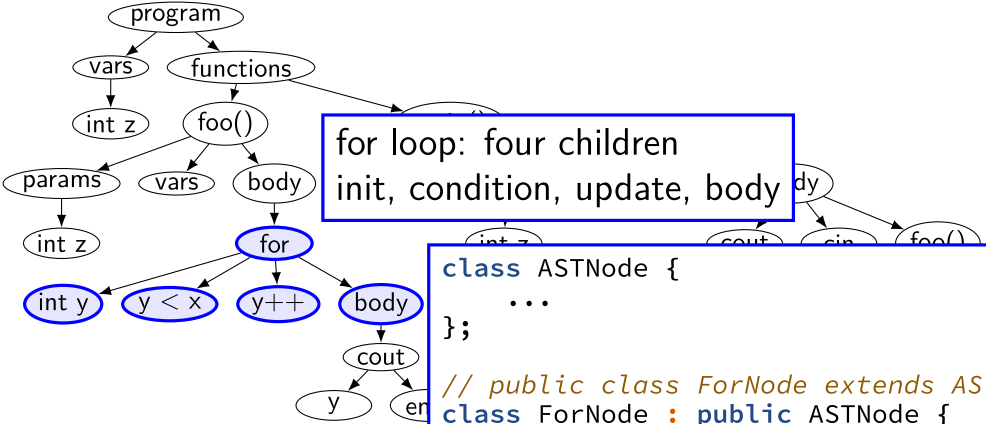
abstract syntax tree



abstract syntax tree



abstract syntax tree



for loop: four children

init, condition, update, body

```
class ASTNode {
    ...
};

// public class ForNode extends ASTNode
class ForNode : public ASTNode {
    ...
private:
    ASTNode *init, *condition,
            *update, *body;
};
```

AST applications

"abstract syntax tree" = "parse tree"

part of how compilers work

do some tree traversal to do...

- code generation — e.g. `ASTNode::outputCode()` method
- optimization
- type checking...

using AST to compare programs

comparing trees is a good way to compare programs...

while ignoring:

- function/method order (e.g. sort function nodes by length)
- variable names (e.g. ignore variable names when comparing)
- comments
- ...

part of many software plagerism/copy+paste detection tools