TREE ADT

BST

AVL

TREE ADT
MOTIVATIONS

Lists - Linear
Trees - Logarithmic

File Systems

Arithmetic Expressions

Compiler Designs

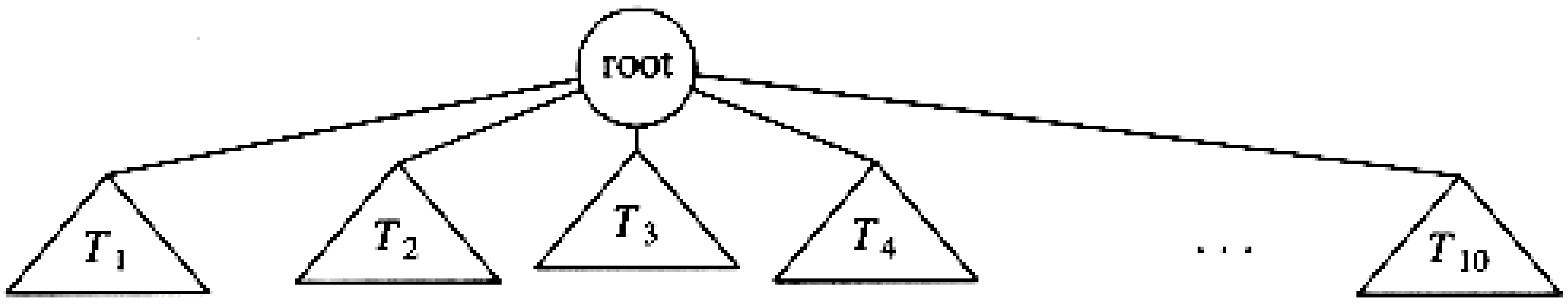
TREE

A connected graph with no cycles.

TREE

A tree consists of a distinguished node r (the root), and zero or more sub trees, T_1, T_2 …, T_k each of whose roots are connected by a directed edge to r.

TREES



TREE

THEOREMS

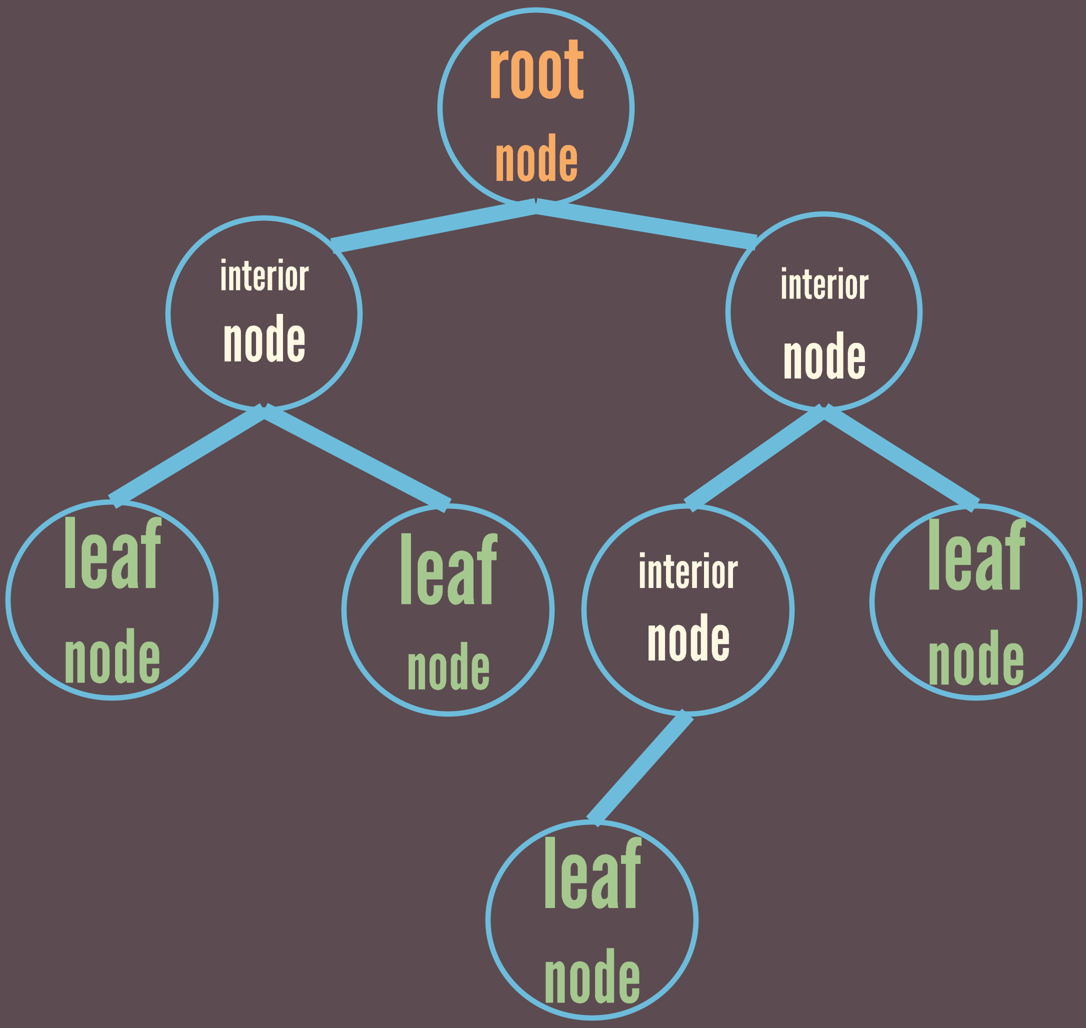
Any two vertices are connected by a unique path.

TREE

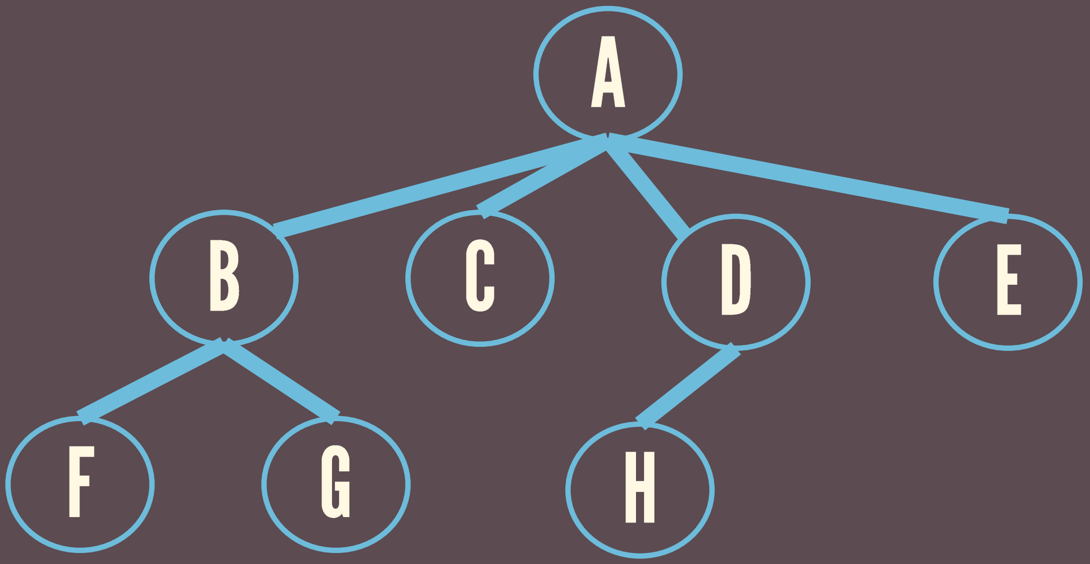
THEOREMS

The number of edges in a tree is $|V(G)| - 1$

ROOT INTERIOR NODE LEAF NODE



SIBLINGS PARENT CHILD GRANDCHILD ANCESTORS DESCENDANTS



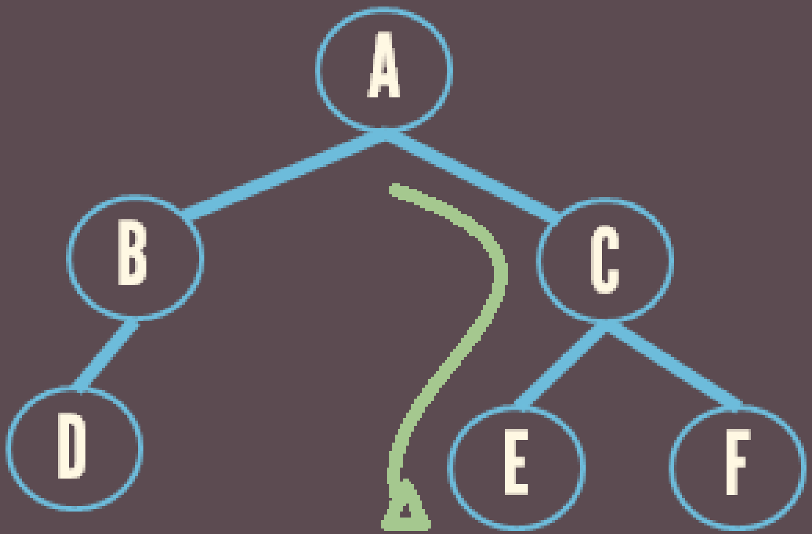
PATH

from node n_1 to n_k

Sequence of nodes n_1, n_2, … n_k such that n_i is the parent of n_{i+1}

$1 \leq i \leq k$

PATH



ACE

length of the
PATH

The number of edges on the path.

HEIGHT

of node n_i

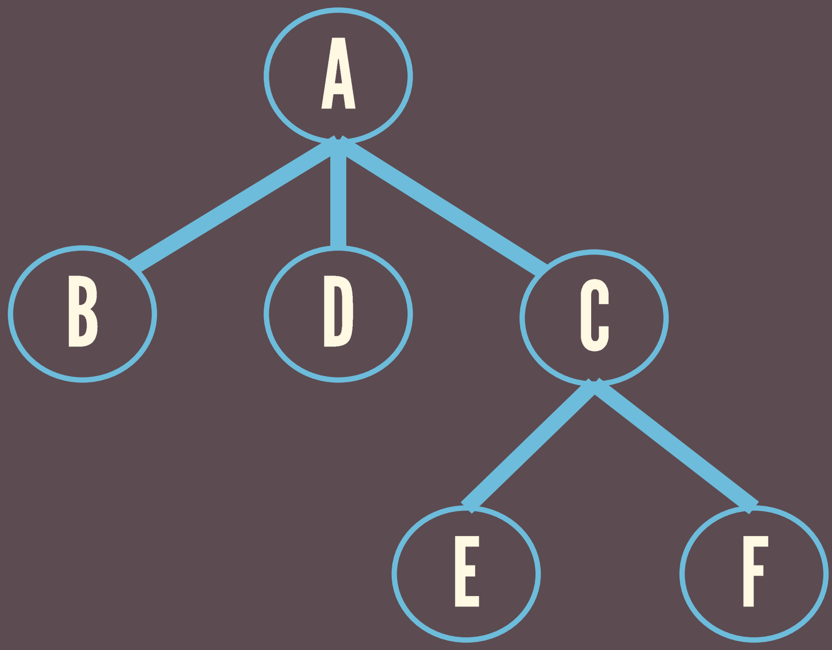
The height of node n_i is the longest path from n_i to a leaf.

DEPTH

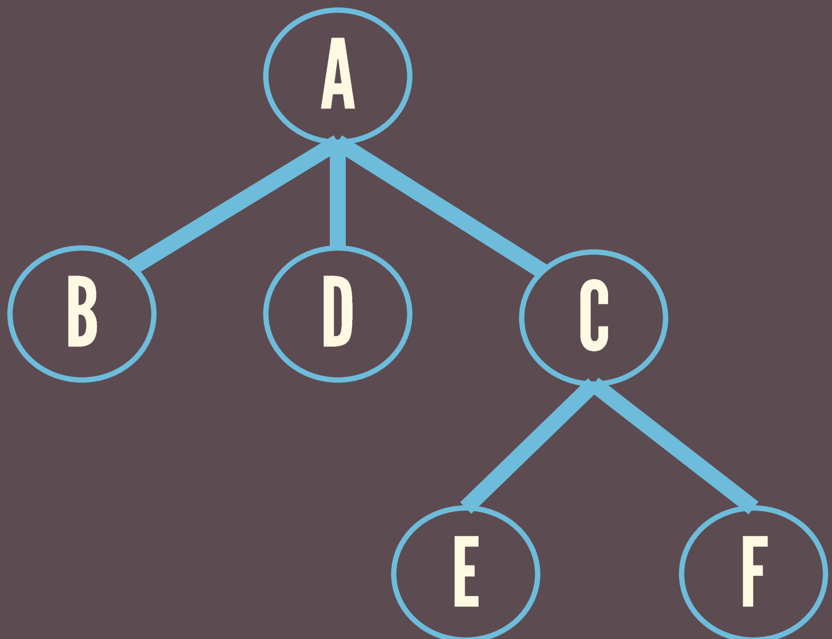
of node n_i

The length of the unique path from the root to n_i

Height of A = 2
Height of B = 0
Height of D = 0
Height of C = 1
Height of E = 0
Height of F = 0
Height of the tree = 2



DEPTH	LEVEL
0	0
1	1
2	2



IMPLEMENTATIONS

LINKED REPRESENTATION

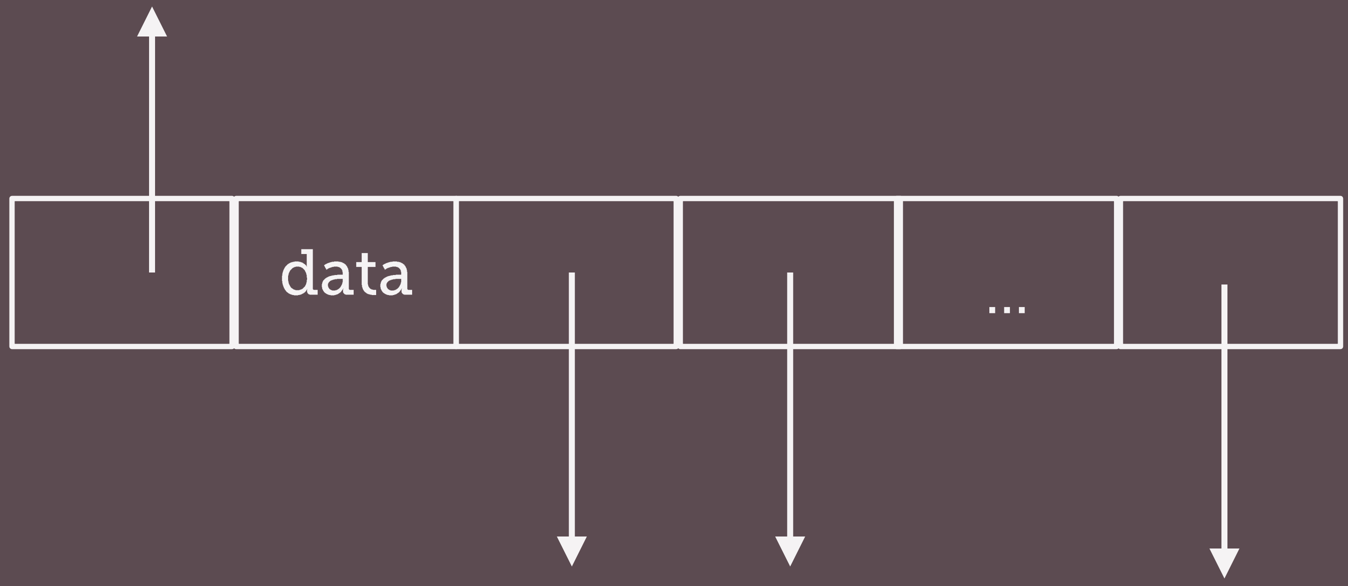
FIRST CHILD, NEXT SIBLING REPRESENTATION

LINKED REPRESENTATION

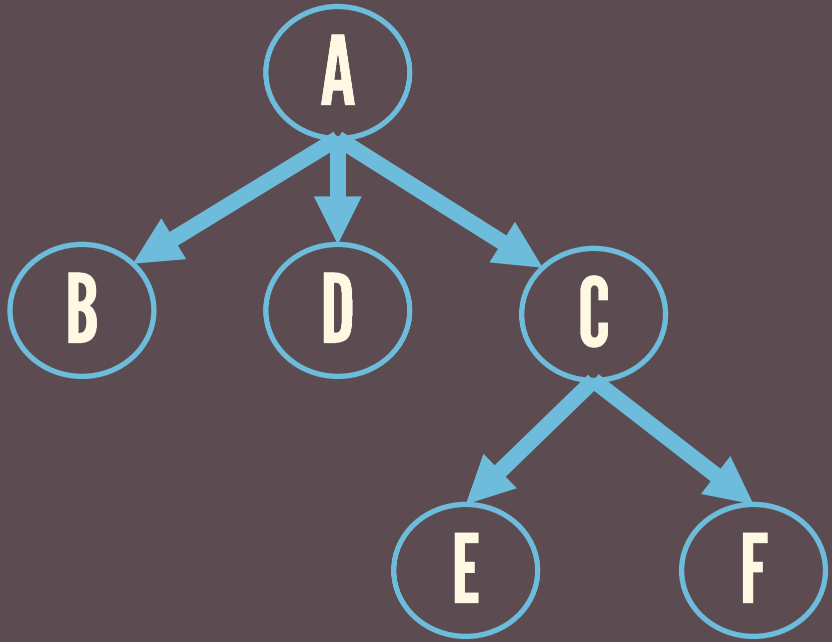
Each node besides its data has a pointer to each child of the node.

```
typedef struct node{
  int data;
  struct node *parent;
  struct node *child1;
  struct node *child2;
  ...
  struct node *childk;
}tree;
```

LINKED REPRESENTATION



LINKED REPRESENTATION



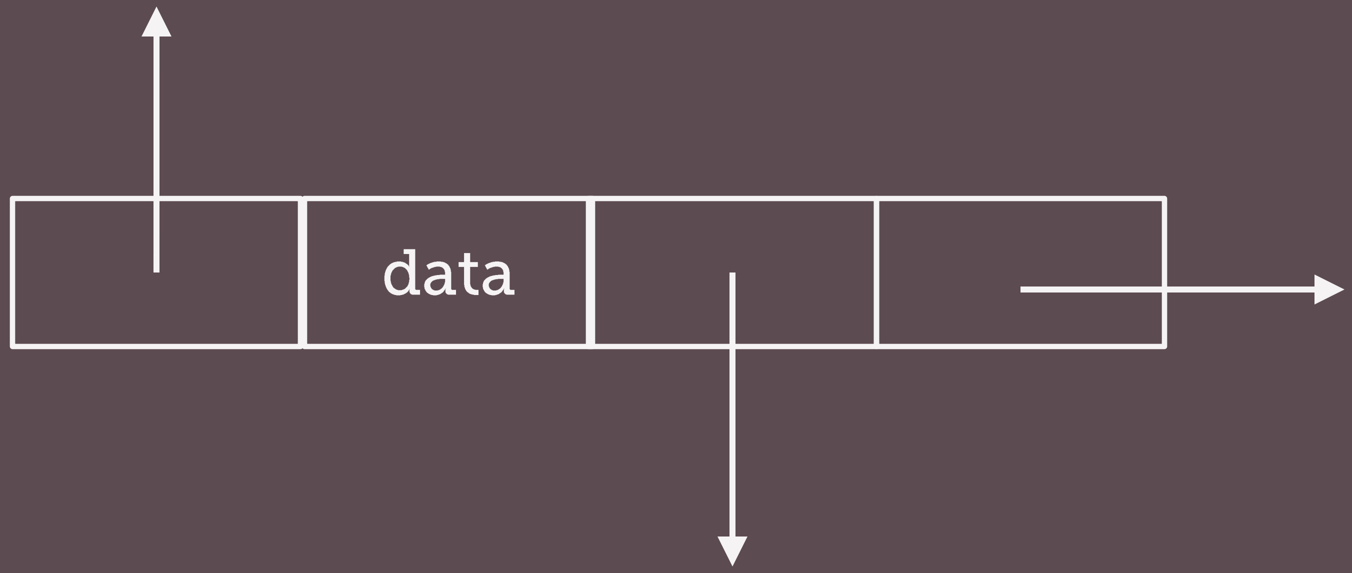
FIRST CHILD, NEXT SIBLING
REPRESENTATION

Keep the children of each node in a linked list of tree nodes.

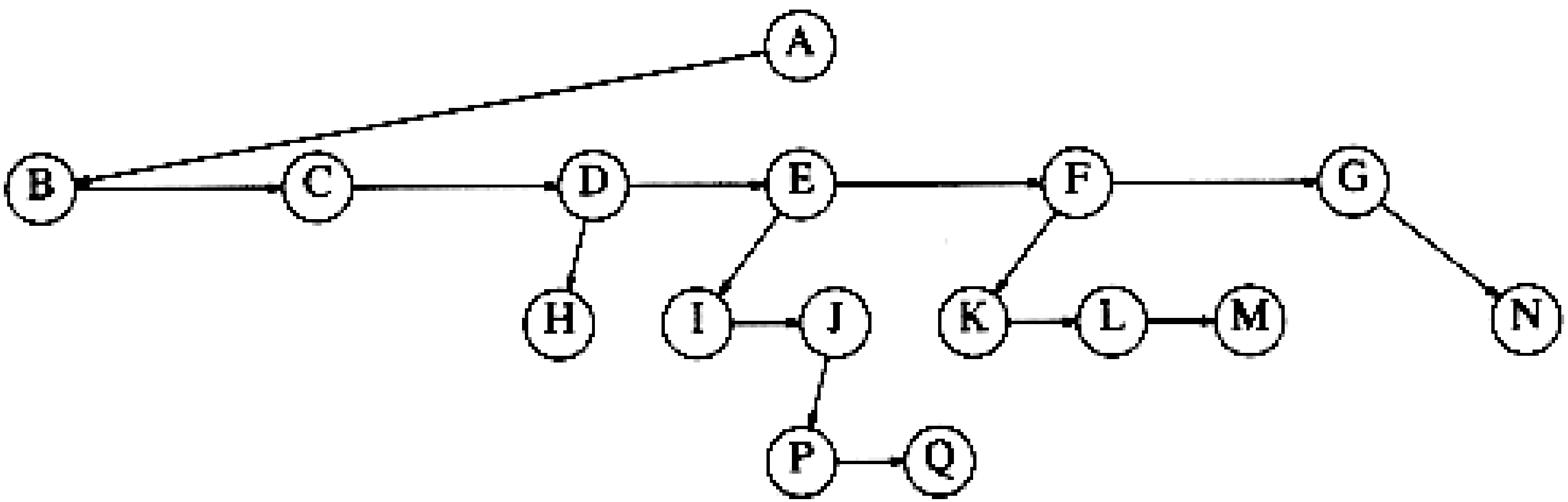
```c
typedef struct node{
  int data;
  struct node *parent;
  struct node *first_child;
  struct node *next_sibling;
}tree;
```

FIRST CHILD, NEXT SIBLING

REPRESENTATION



FIRST CHILD, NEXT SIBLING REPRESENTATION



BINARY TREES

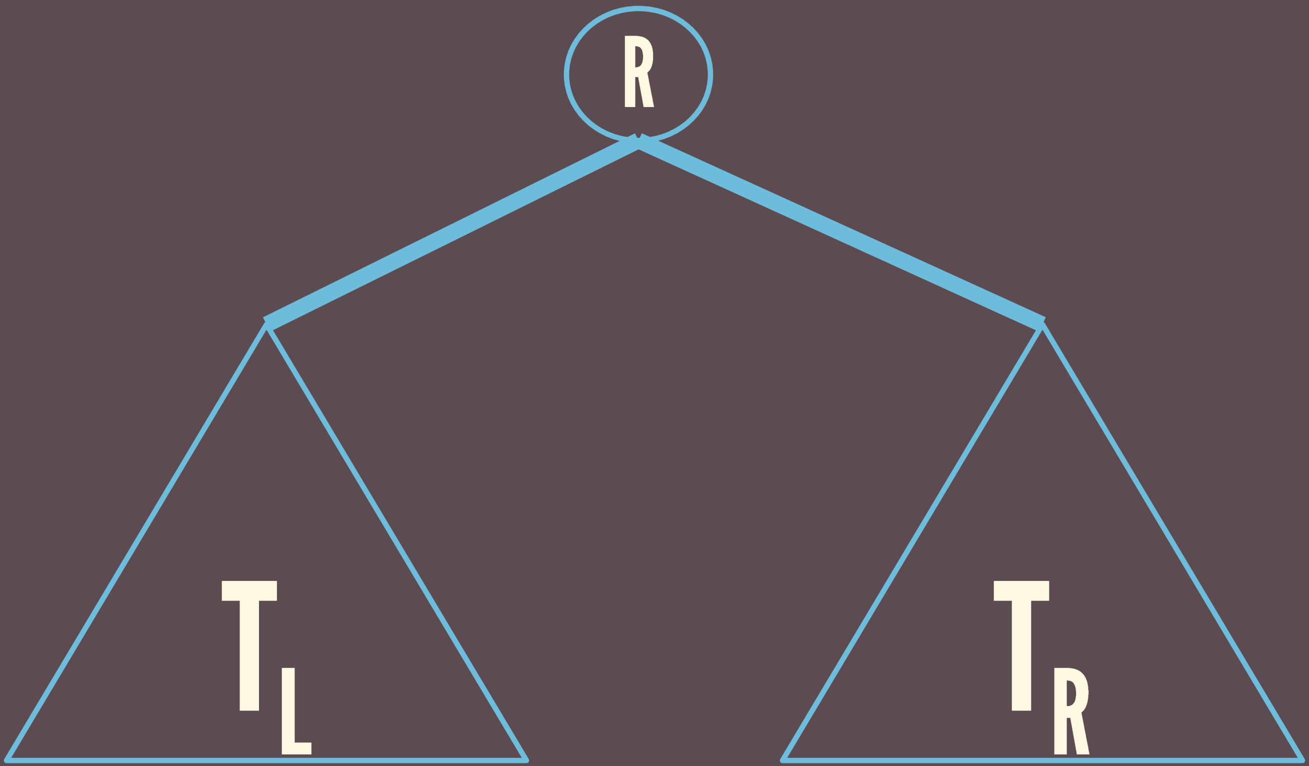
BINARY TREE

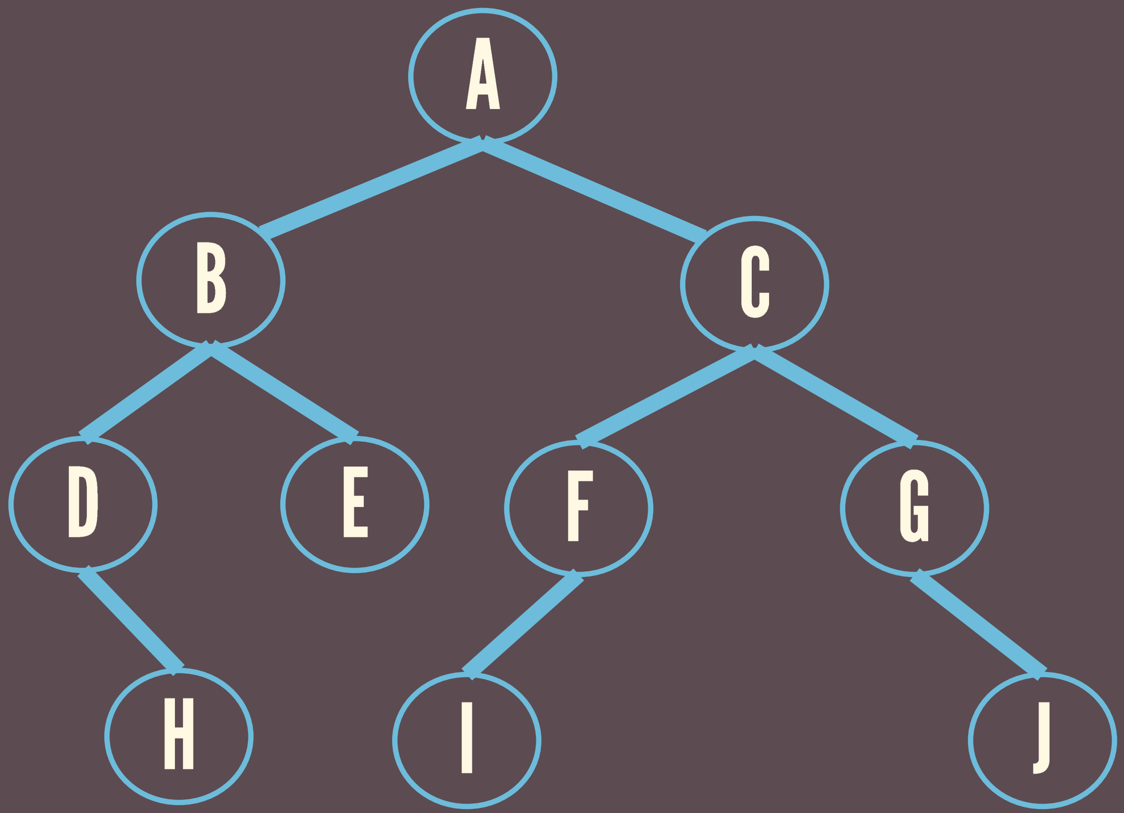
A tree in which no node can have more than two children.

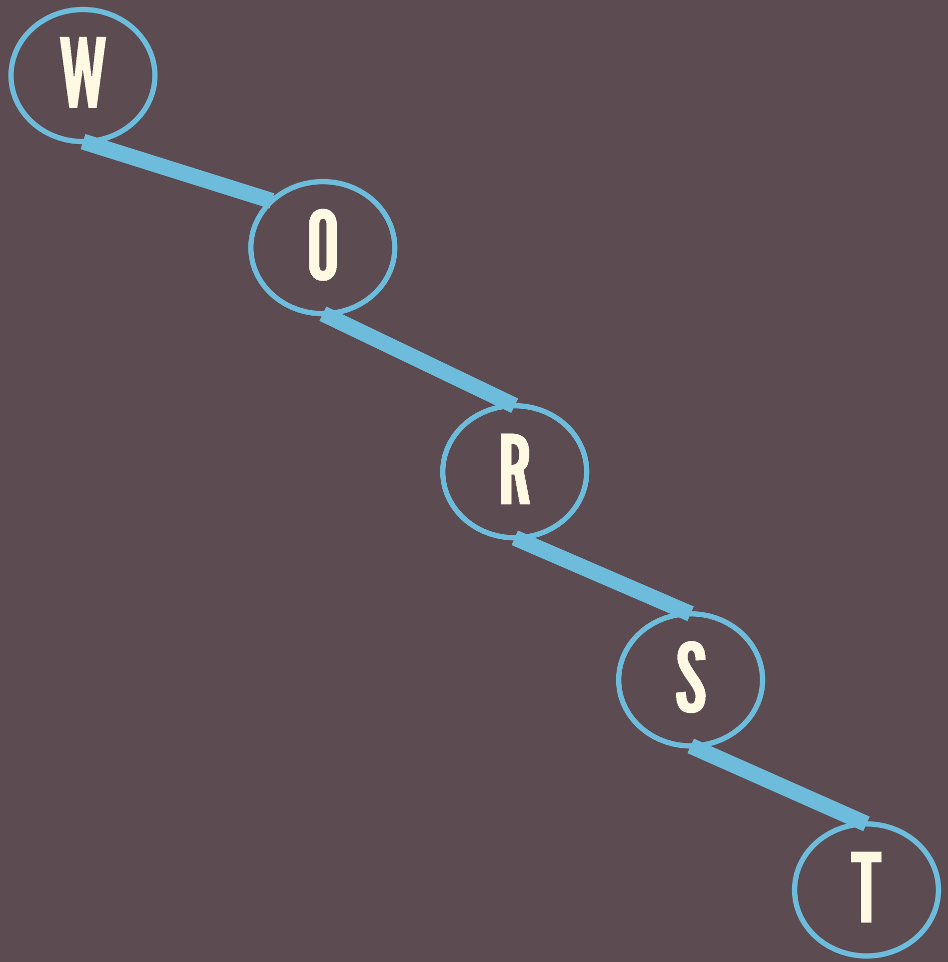
BINARY TREE

A tree where each node has either

- no children
- a left child
- a right child
- both left and right child


R
T_L
T_R

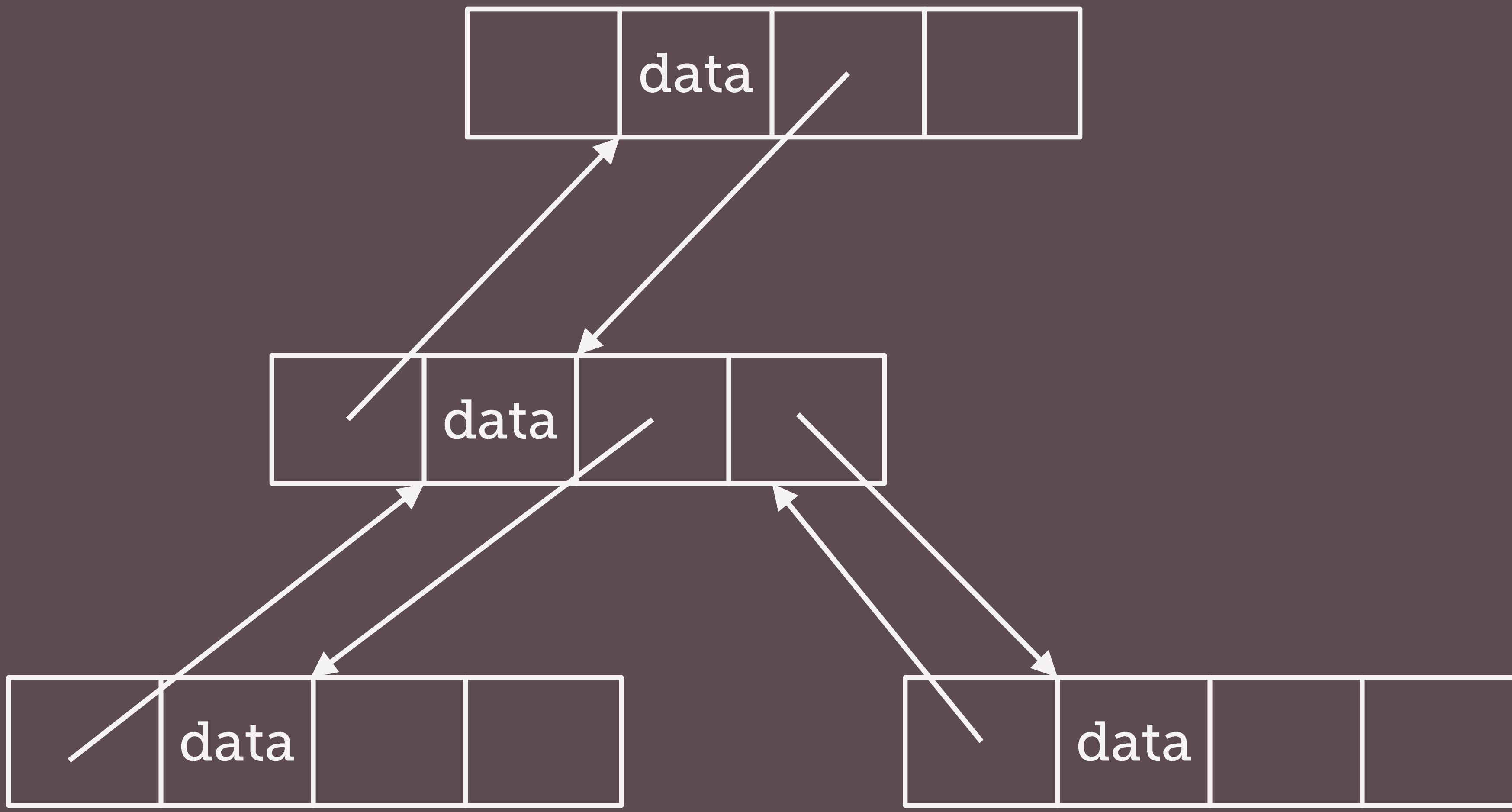

A
B
C
D
E
F
G
H
I
J


W
O
R
S
T

IMPLEMENTATION

LINKED REPRESENTATION

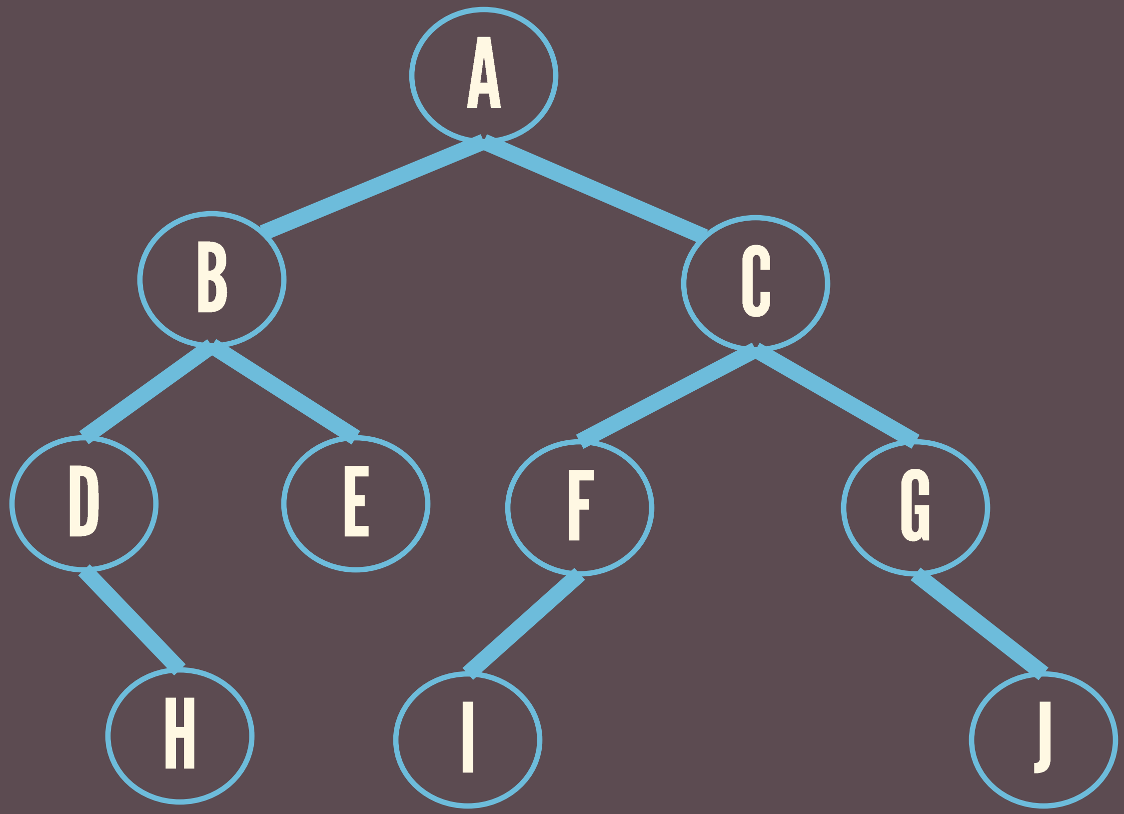
```
typedef struct node{
  int data;
  struct node *parent;
  struct node *left;
  struct node *right;
}tree;
```


data
data
data
data

full

LEVEL

Level i is full if there are exactly 2^i nodes at this level.


A
B
C
D
E
F
G
H
I
J

BINARY TREE TRAVERSALS

PREORDER

INORDER

POSTORDER

PREORDER

Visit root node, then left subtree and finally the right subtree.

```
preorder(tree *node){

  if(node!=NULL){
      print node->data
      preorder(node->left);
      preorder(node->right);
  }

}
```

INORDER

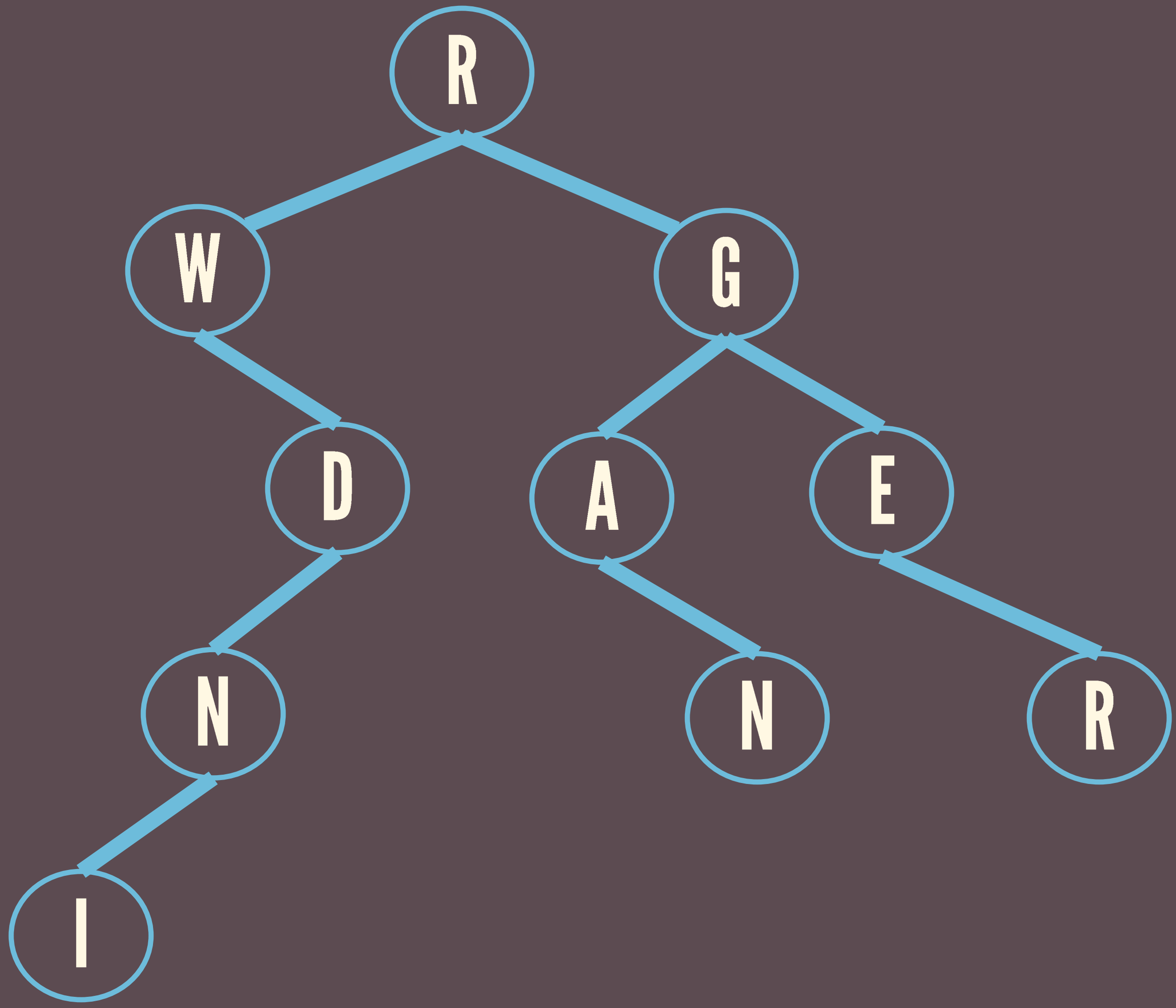
Visit left subtree, then root node and finally the right subtree.

```
inorder(tree *node){

  if(node!=NULL){
      inorder(node->left);
      print node->data
      inorder(node->right);
  }

}
```

POSTORDER

Visit left subtree, then right subtree and finally the root node.

```
postorder(tree *node){

  if(node!=NULL){
      postorder(node->left);
      postorder(node->right);
      print node->data
  }

}
```


R
W
G
D
A
E
N
N
R
I

PREORDER: R W D N I G A N E R

INORDER: W I N D R A N G E R

POSTORDER: _ _ _ _ _ _ _ _ _ _

PREORDER: R W D N I G A N E R

INORDER: W I N D R A N G E R

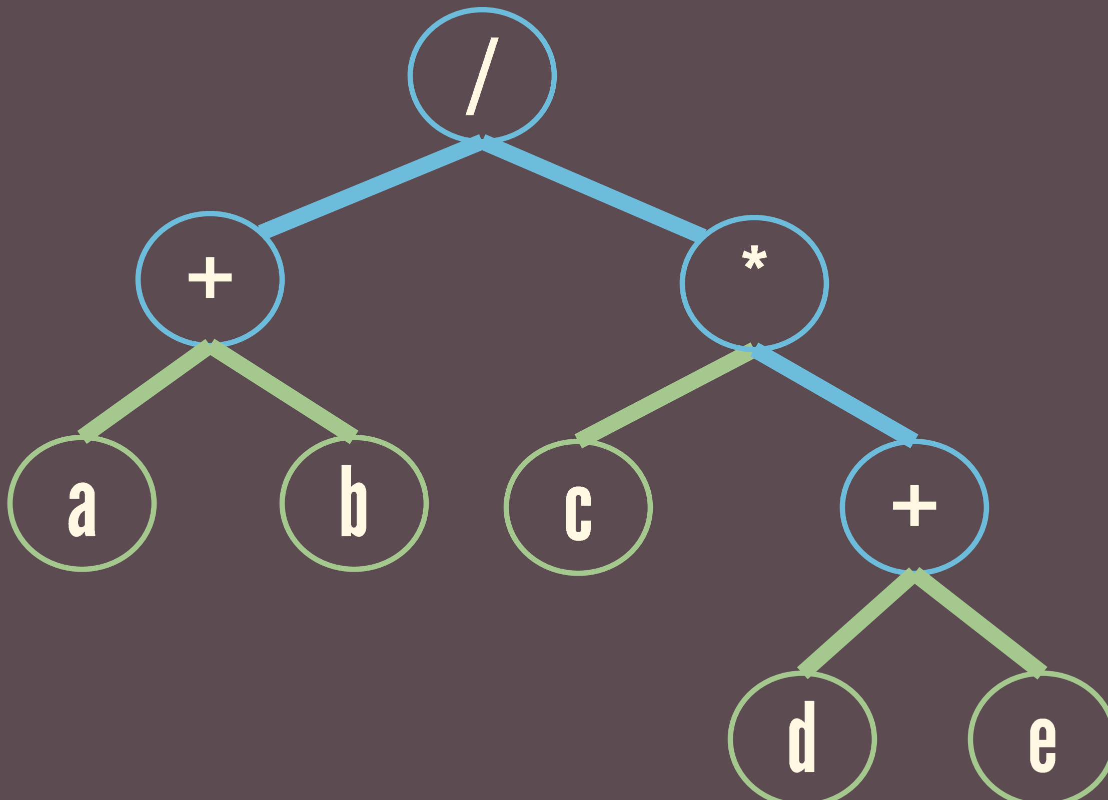
POSTORDER: I N D W N A R E G R

EXPRESSION TREES

LEAVES
OPERANDS

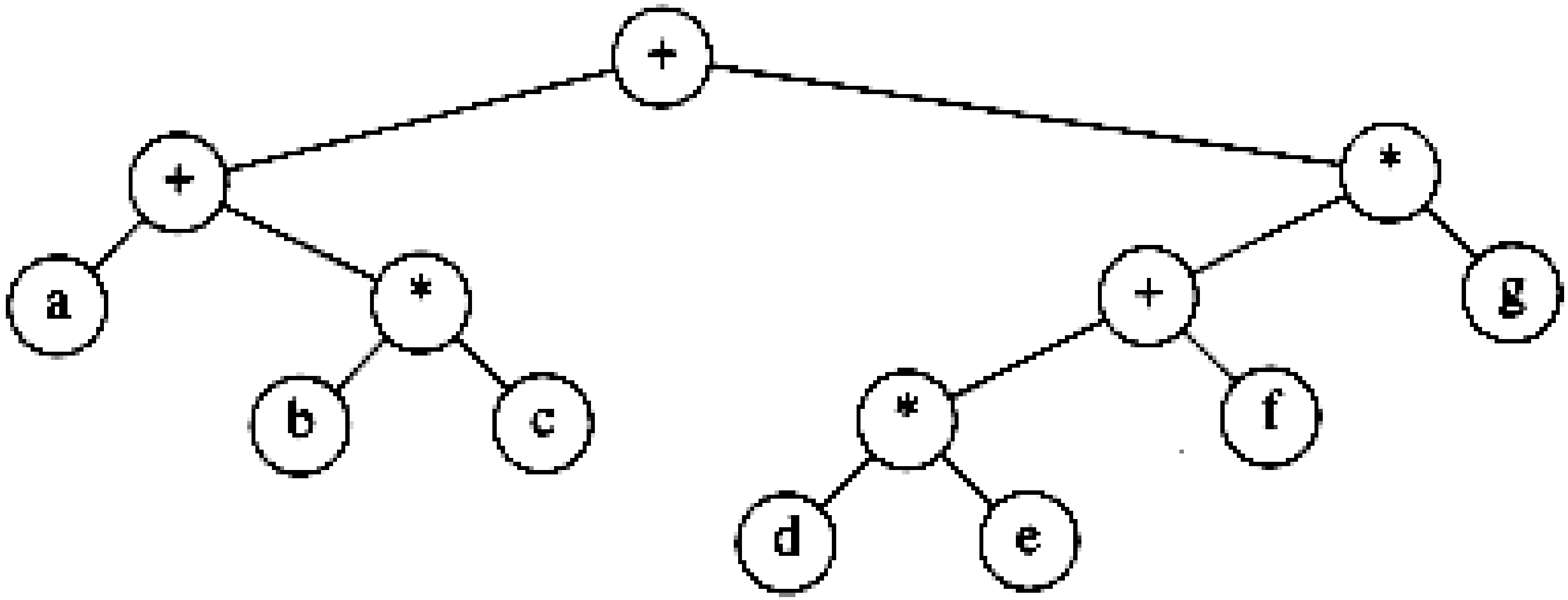
INTERNAL NODES
OPERATORS

$(a + b) / (c * (d + e))$

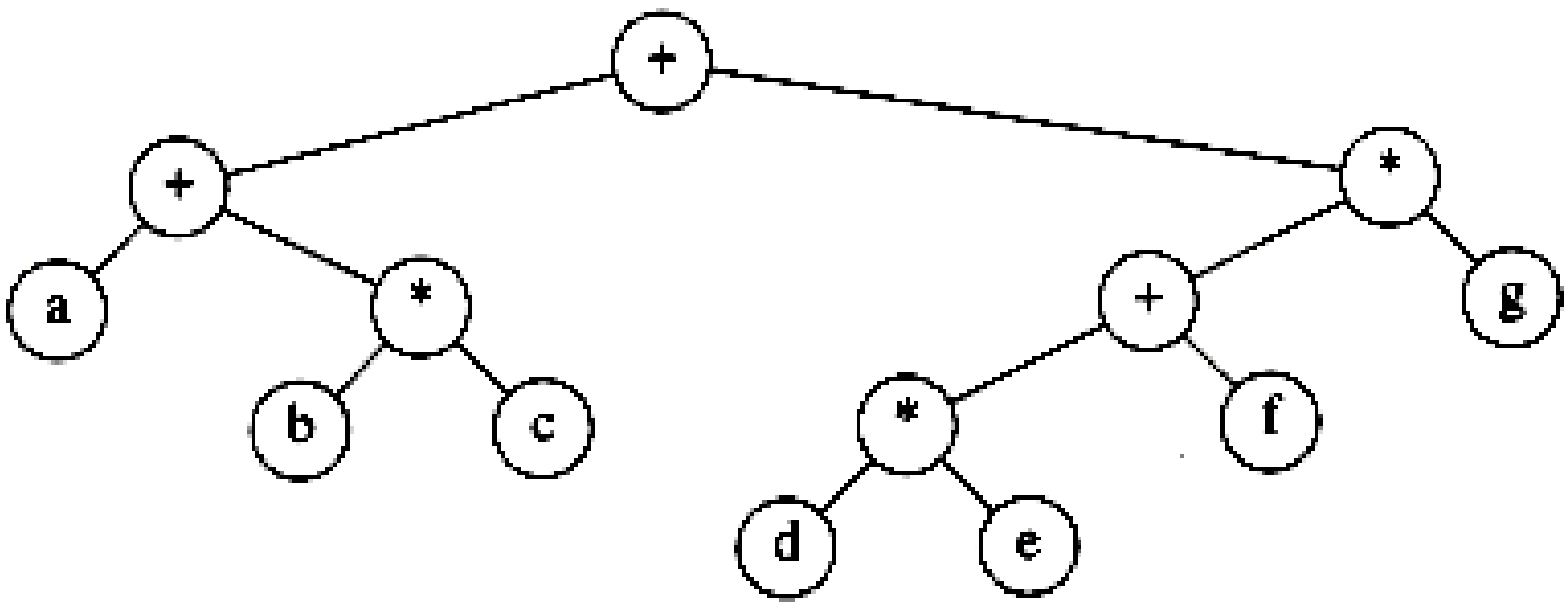


EXPRESSION TREE TRAVERSALS

PREORDER PREFIX	INORDER INFIX	POSTORDER POSFIX

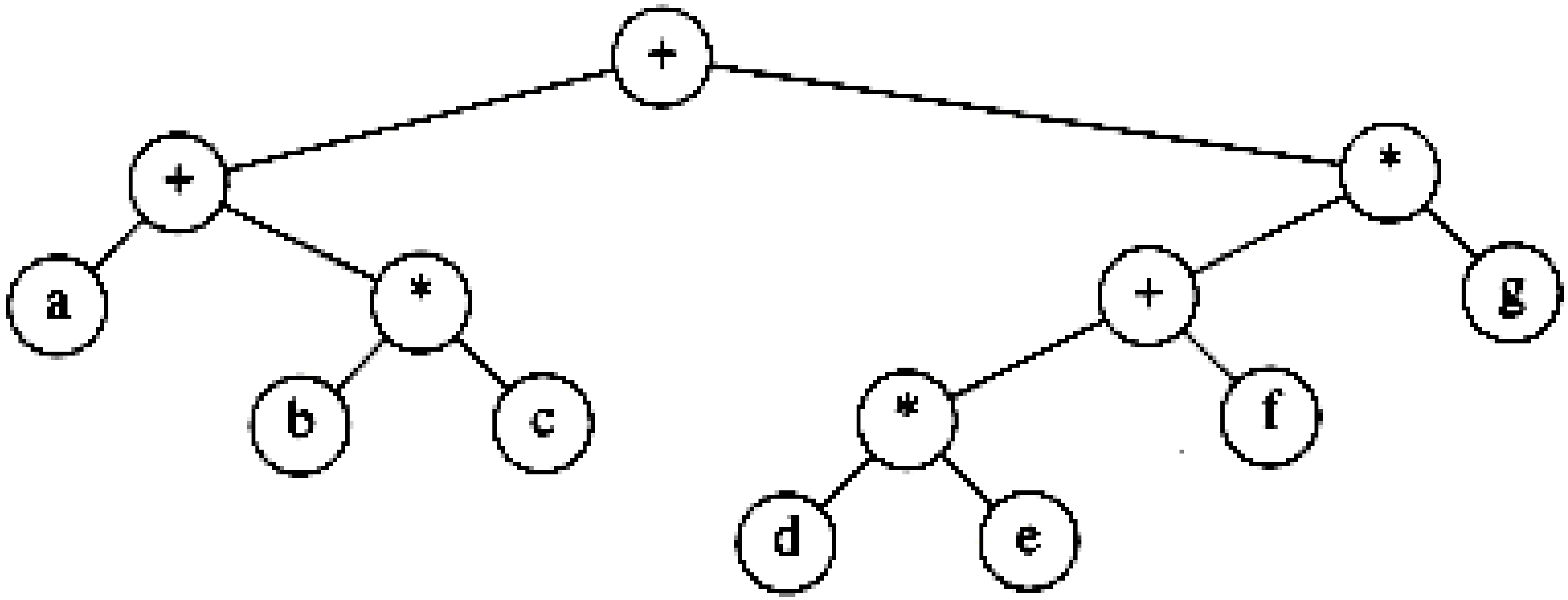


(a + b * c) + ((d * e + f) * g)

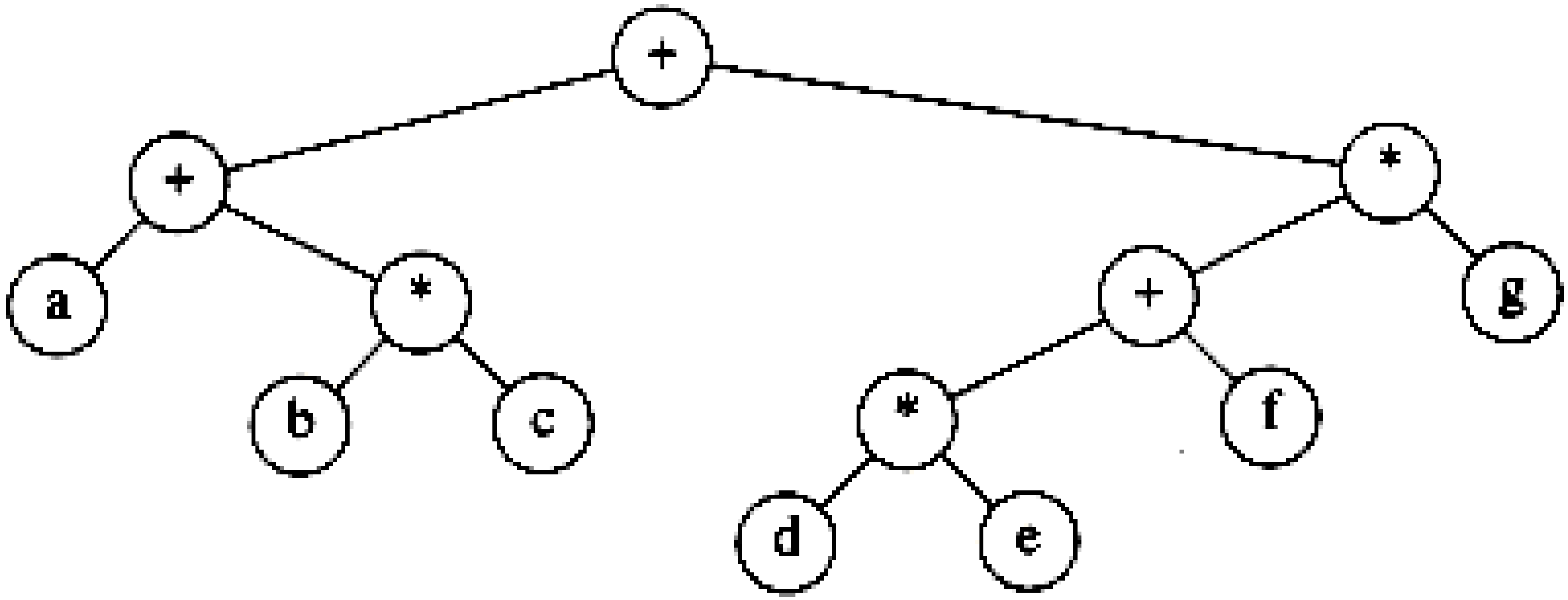


PREFIX

+ + a * b c * + * d e f g



POSTFIX a b c * + d e * f + g * +



INFIX

a+b*c+d*e+f*g

ALGORITHM TO CONSTRUCT

EXPRESSION TREES

Convert the expression to postfix.

Use a stack.

Read the expression (postfix) one symbol at a time:

if the symbol is an operand,

- create a one-node tree
- push a pointer to it onto a stack

Read the expression (postfix) one symbol at a time:

if the symbol is an operator,

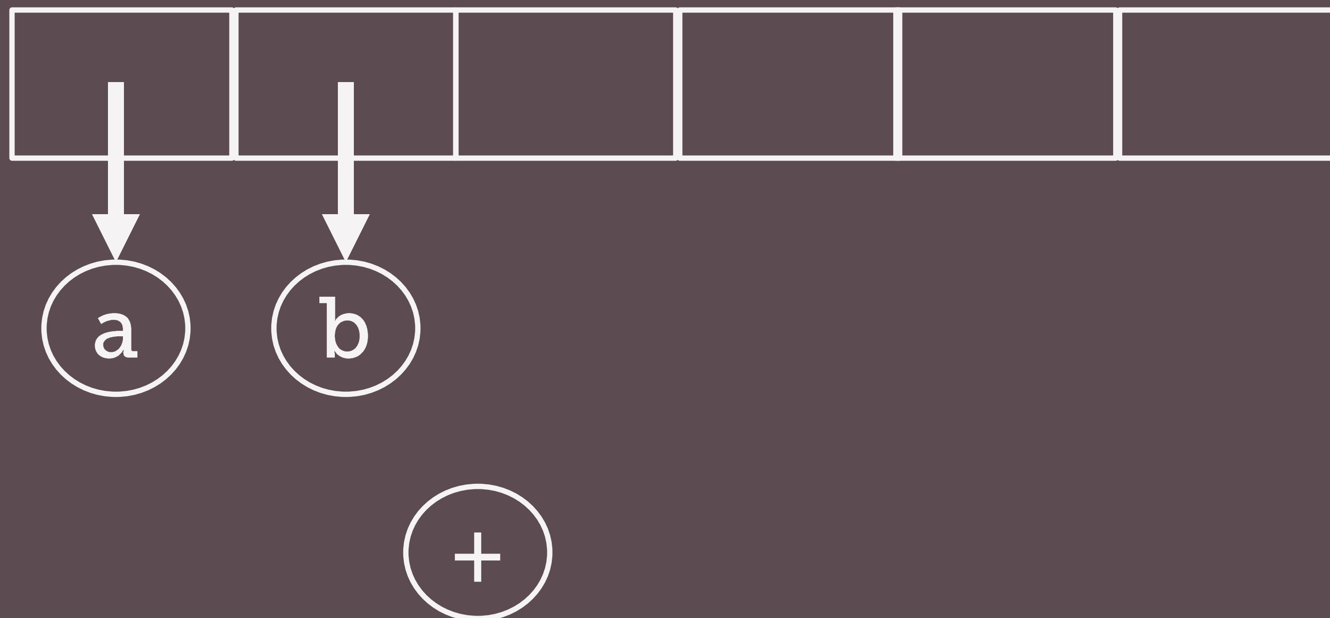
- pop two pointers to two trees (T_1 and T_2).
- Form a new tree whose root is the operator with left and right child pointing to T_1 and T_2 respectively.
- push onto the stack a pointer to this new tree.

EXAMPLE

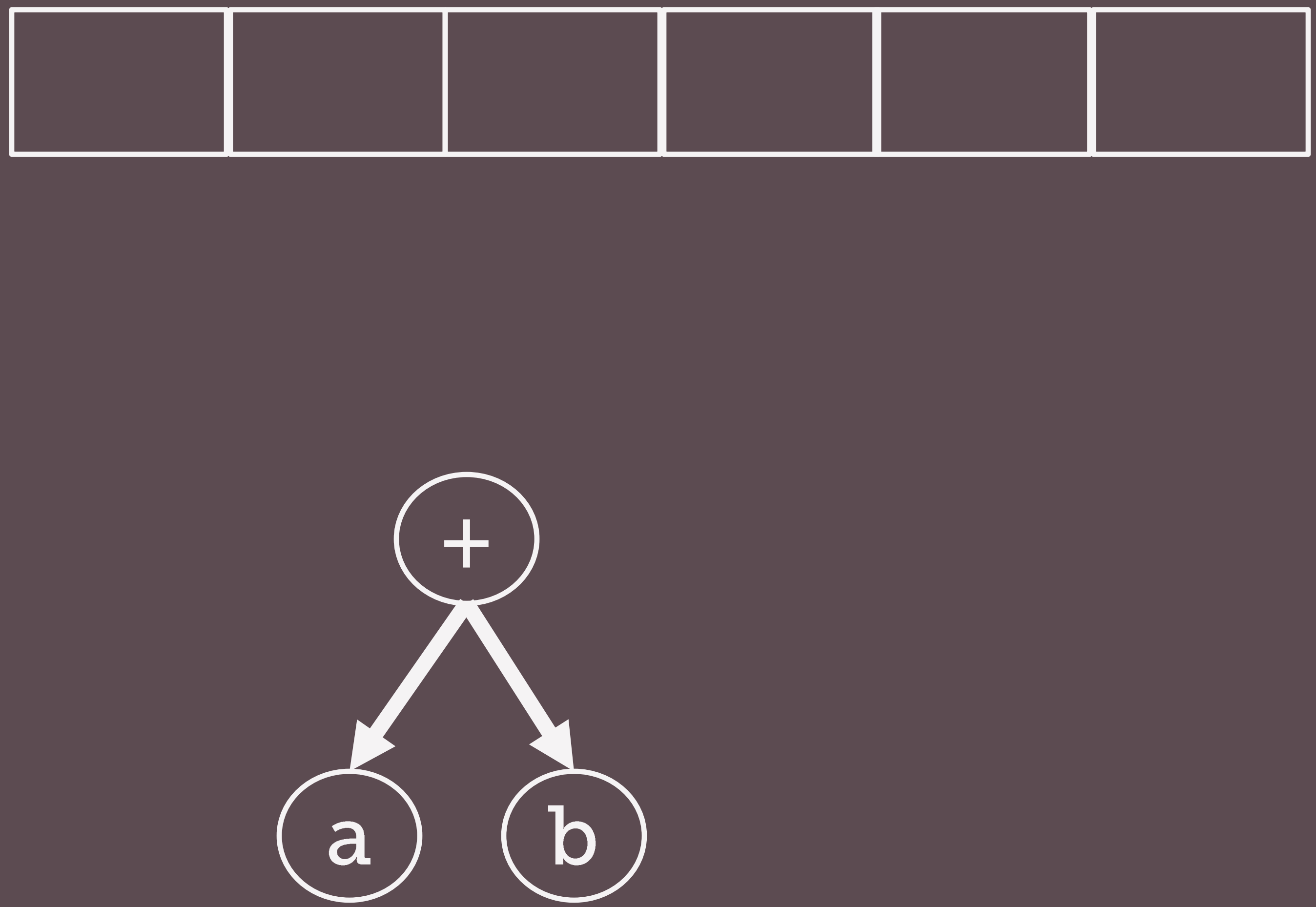
a b + c d e + * *

a b + c d e + * *

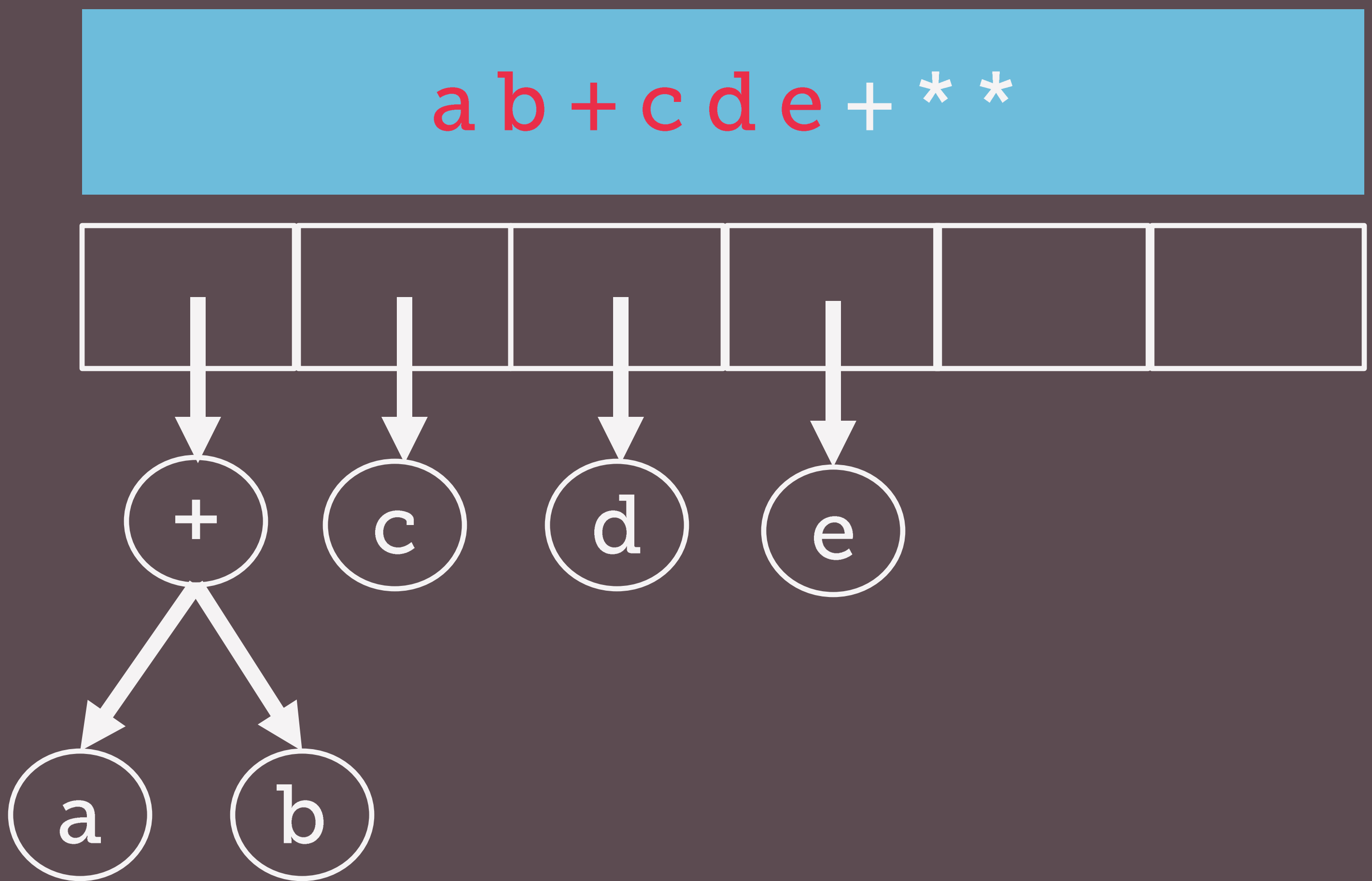
a b + c d e + * *



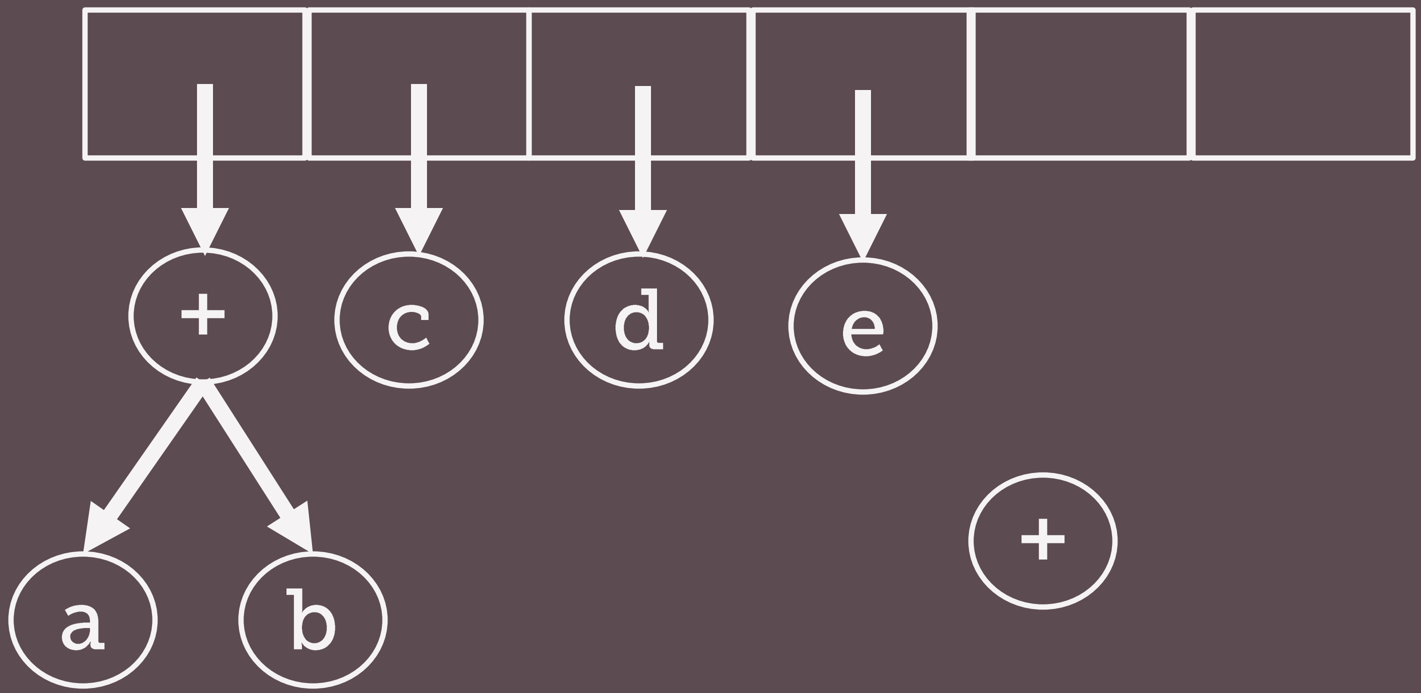
a b + c d e + * *

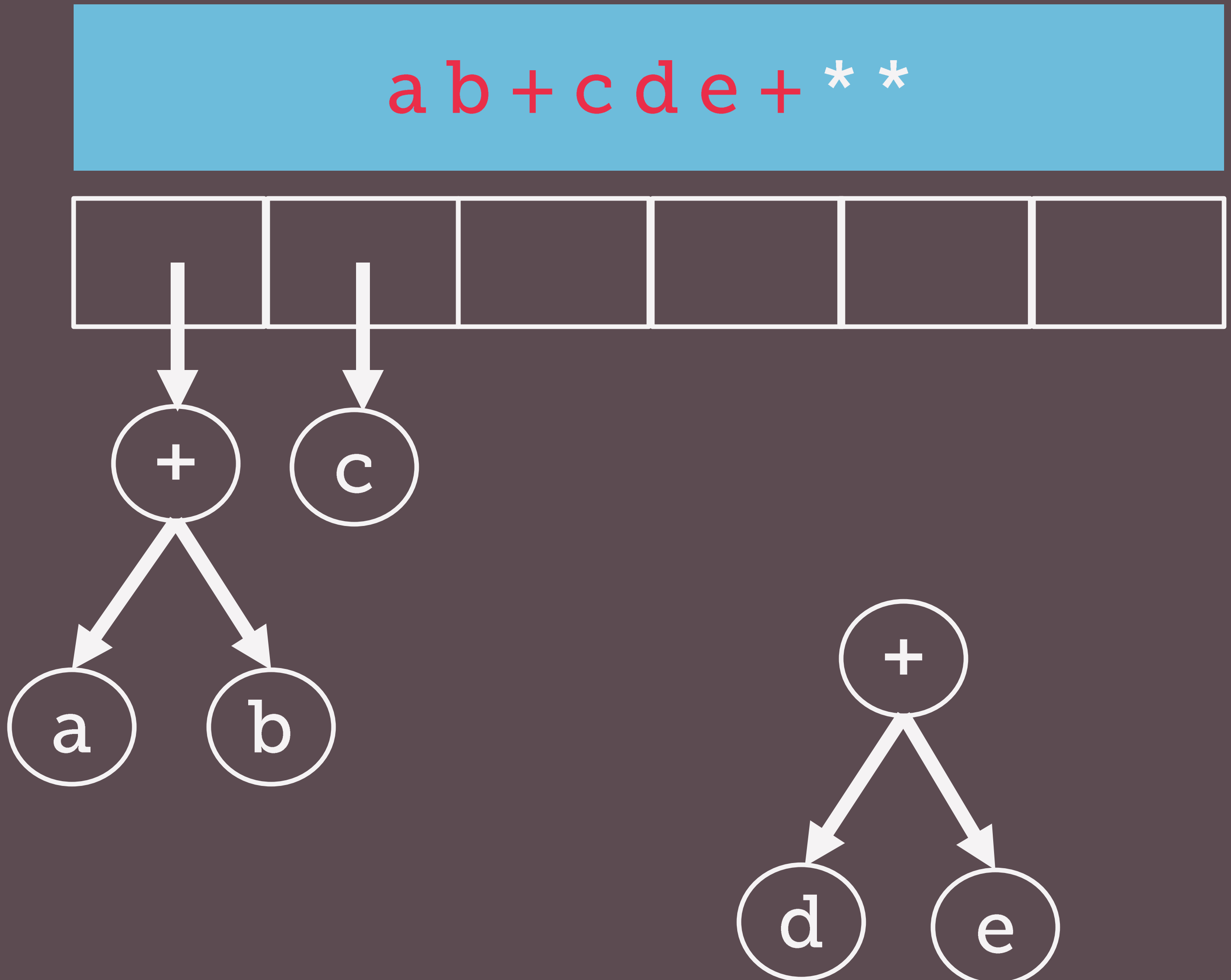


a b + c d e + * *

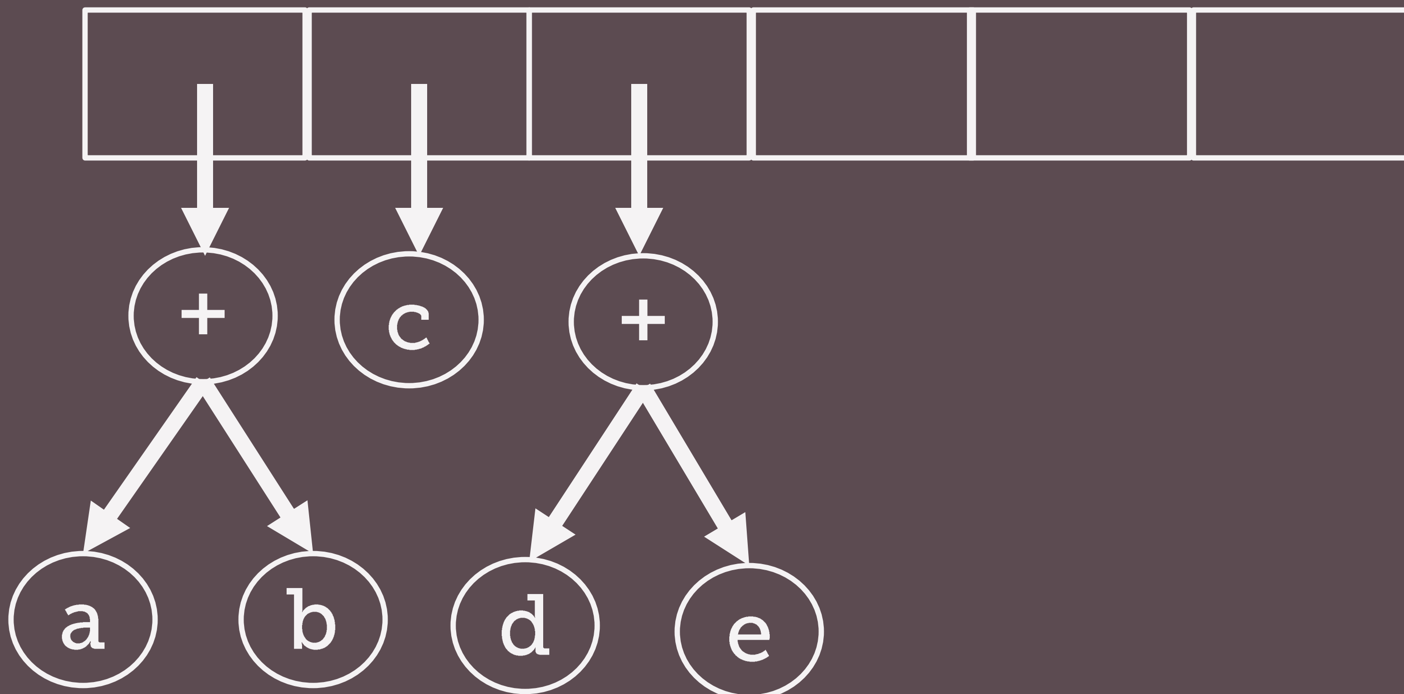

a b + c d e + * *
+
c
d
e
a
b

a b + c d e + * *

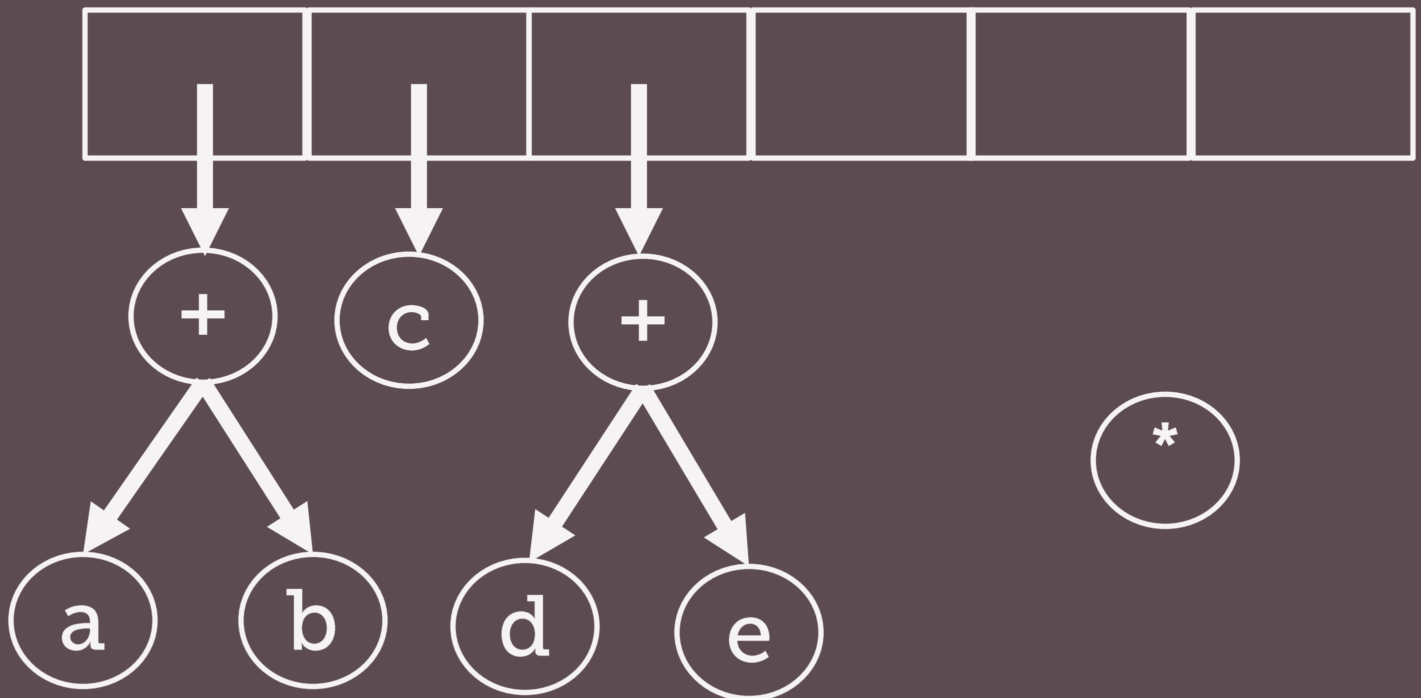


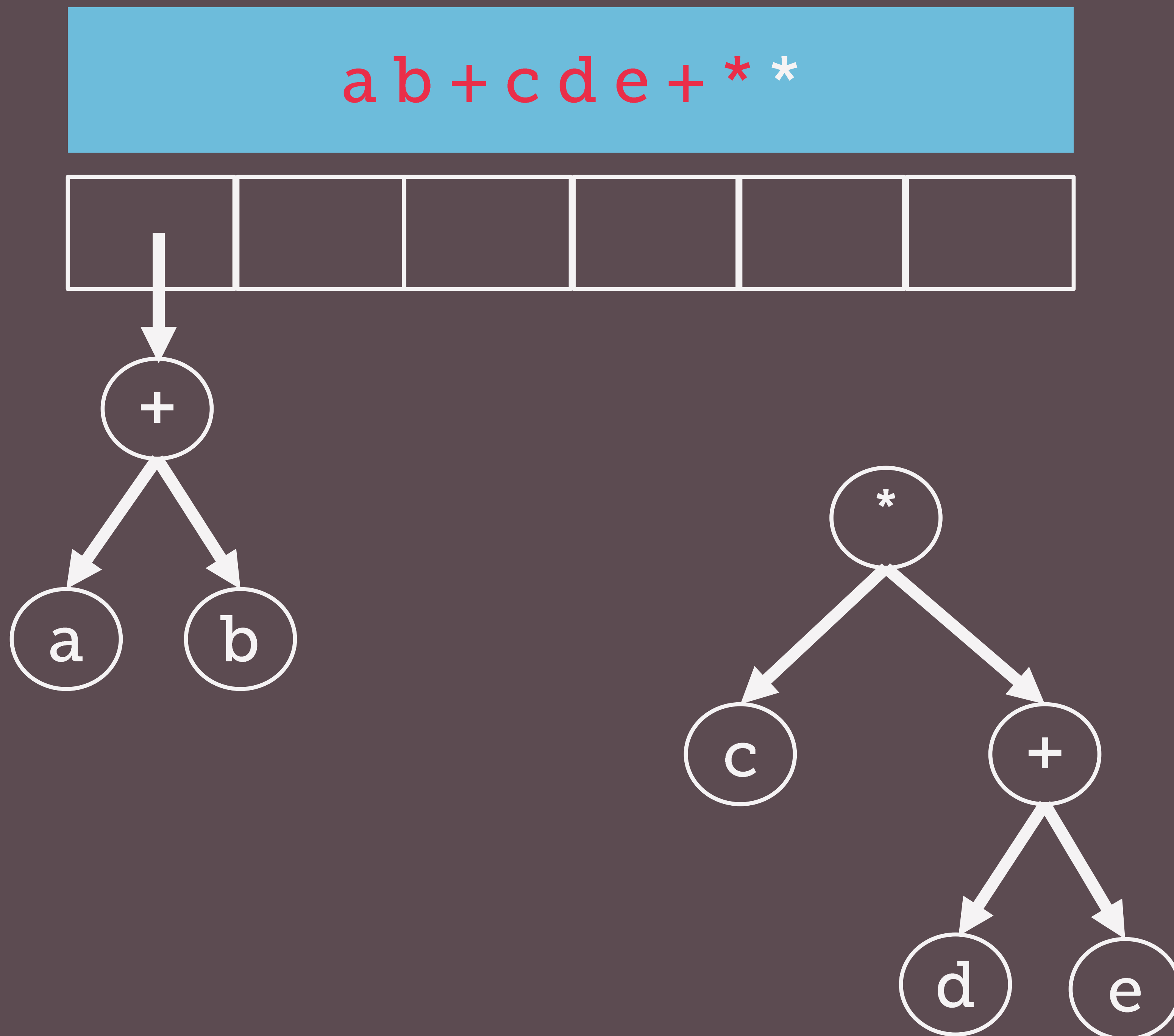

a b + c d e + * *
+
c
a
b
+
d
e

a b + c d e + * *

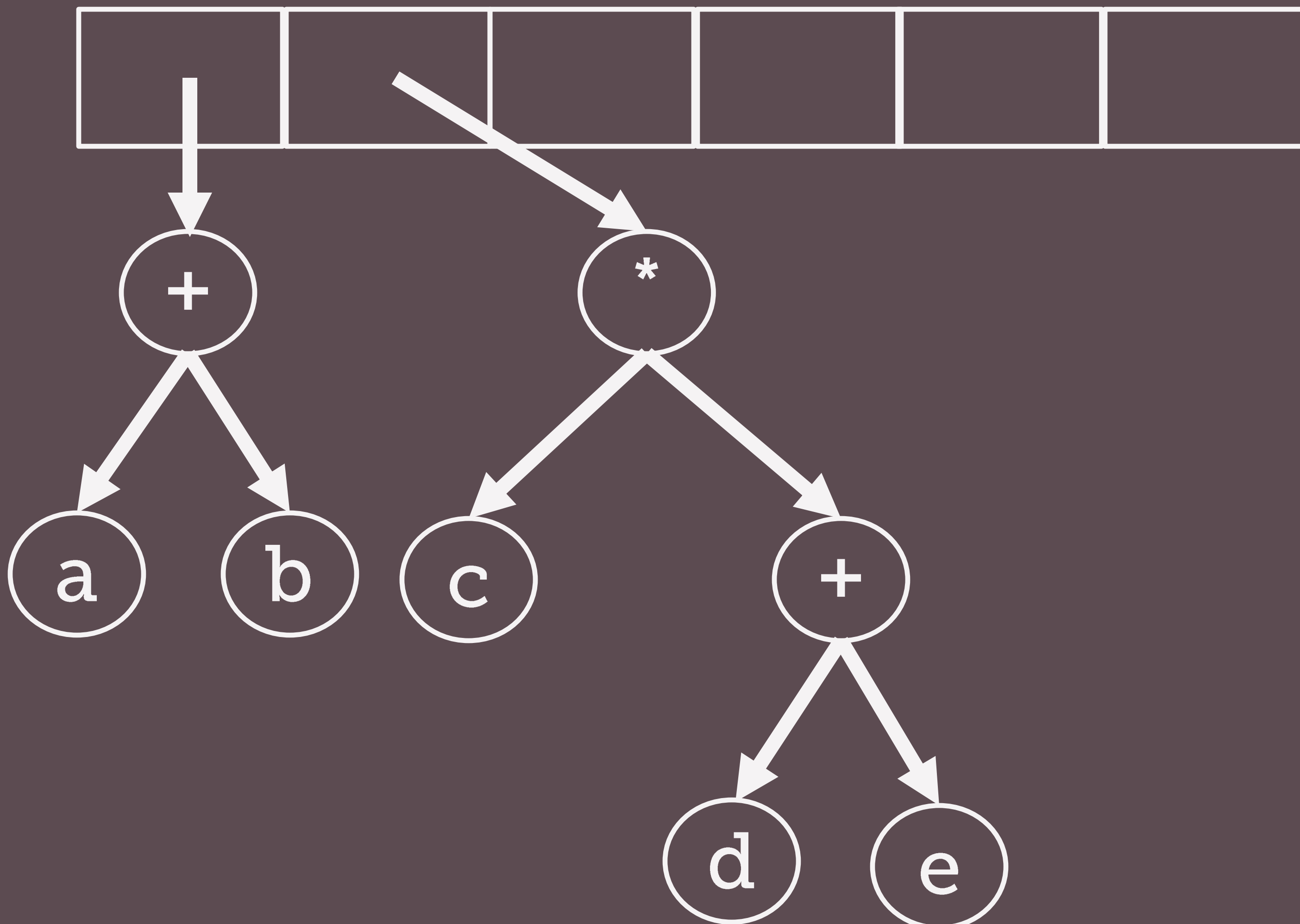


a b + c d e + * *

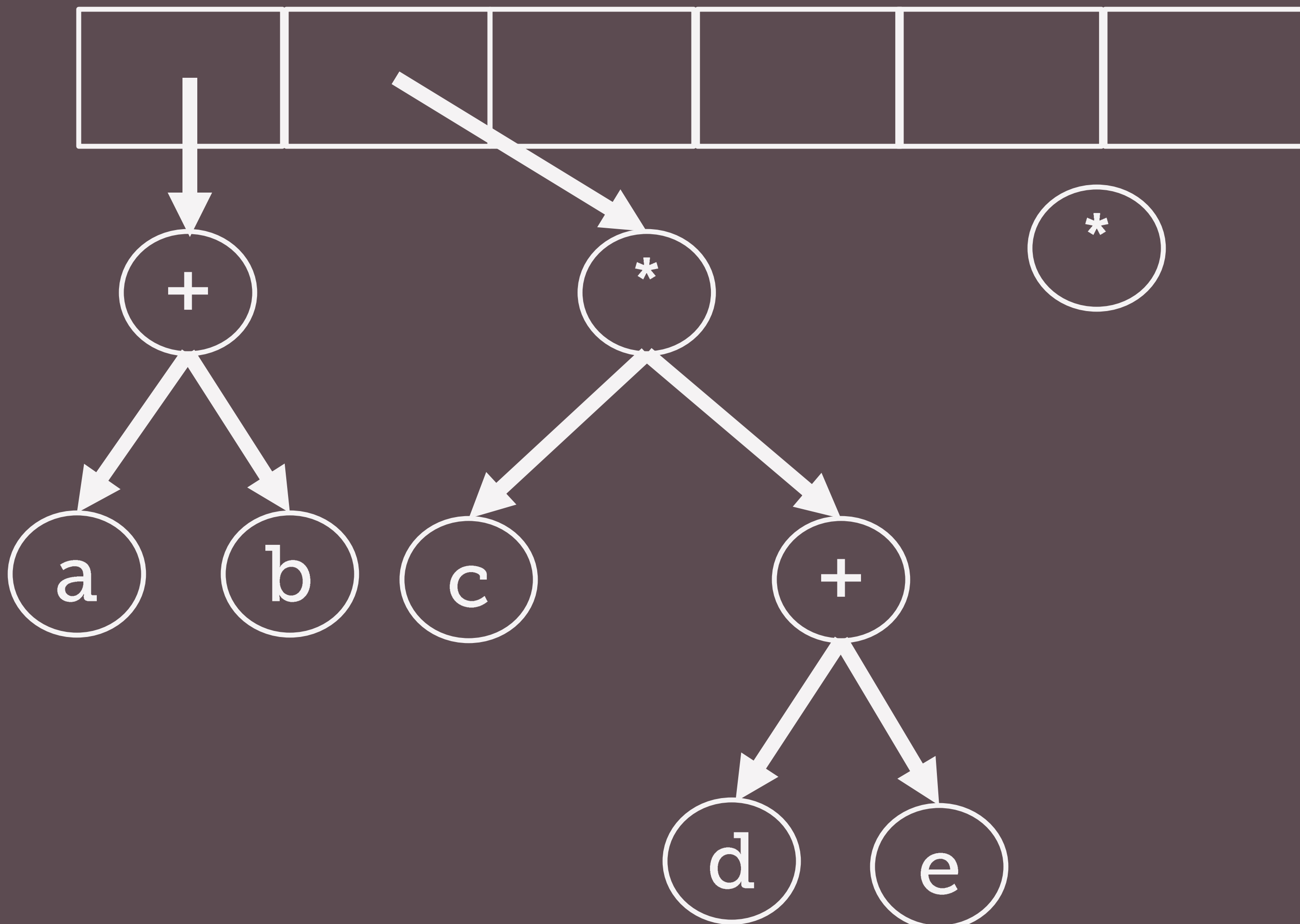



a b + c d e + * *
+
a
b
*
c
+
d
e

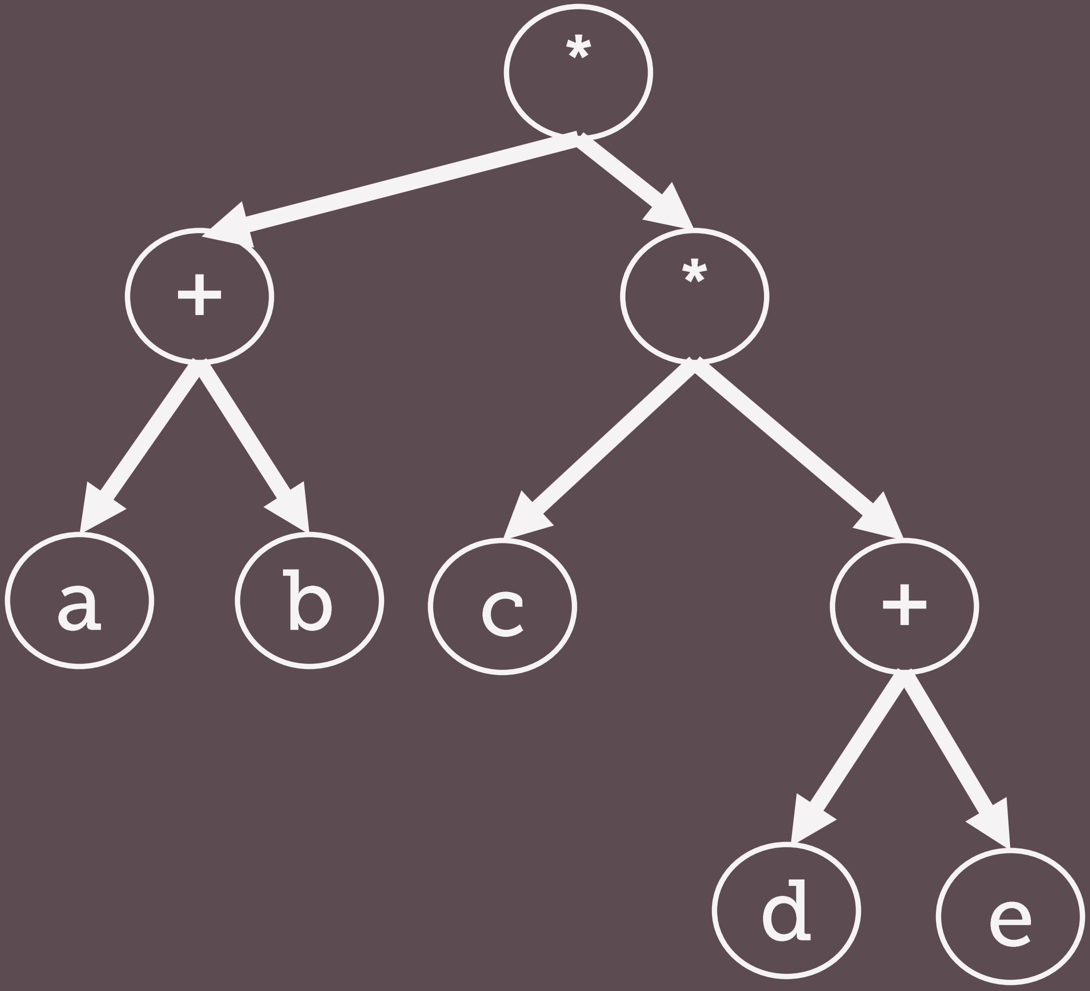
a b + c d e + * *



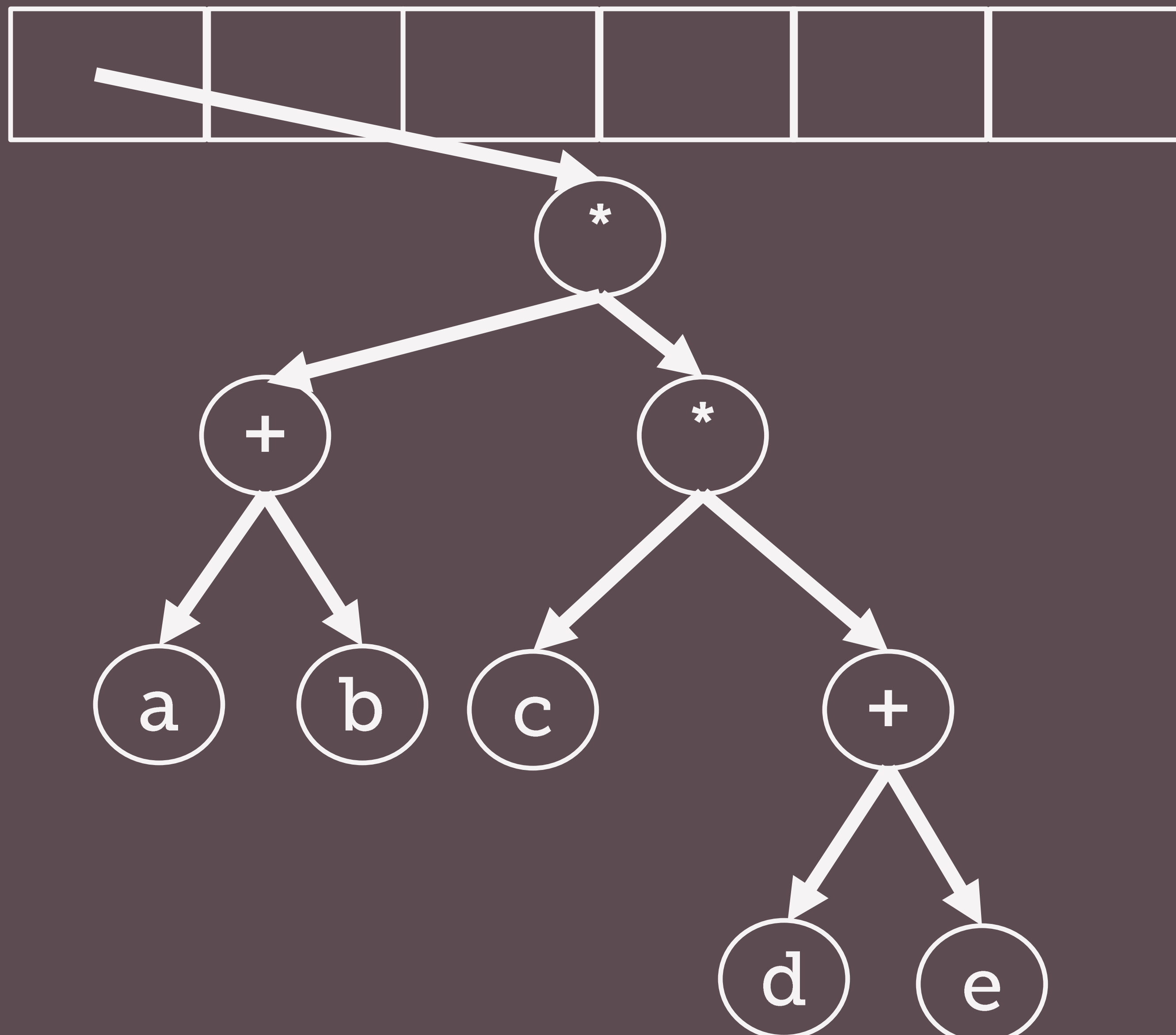
a b + c d e + * *



a b + c d e + * *



a b + c d e + * *



SEARCH TREE ADT

BST

AVL

BST

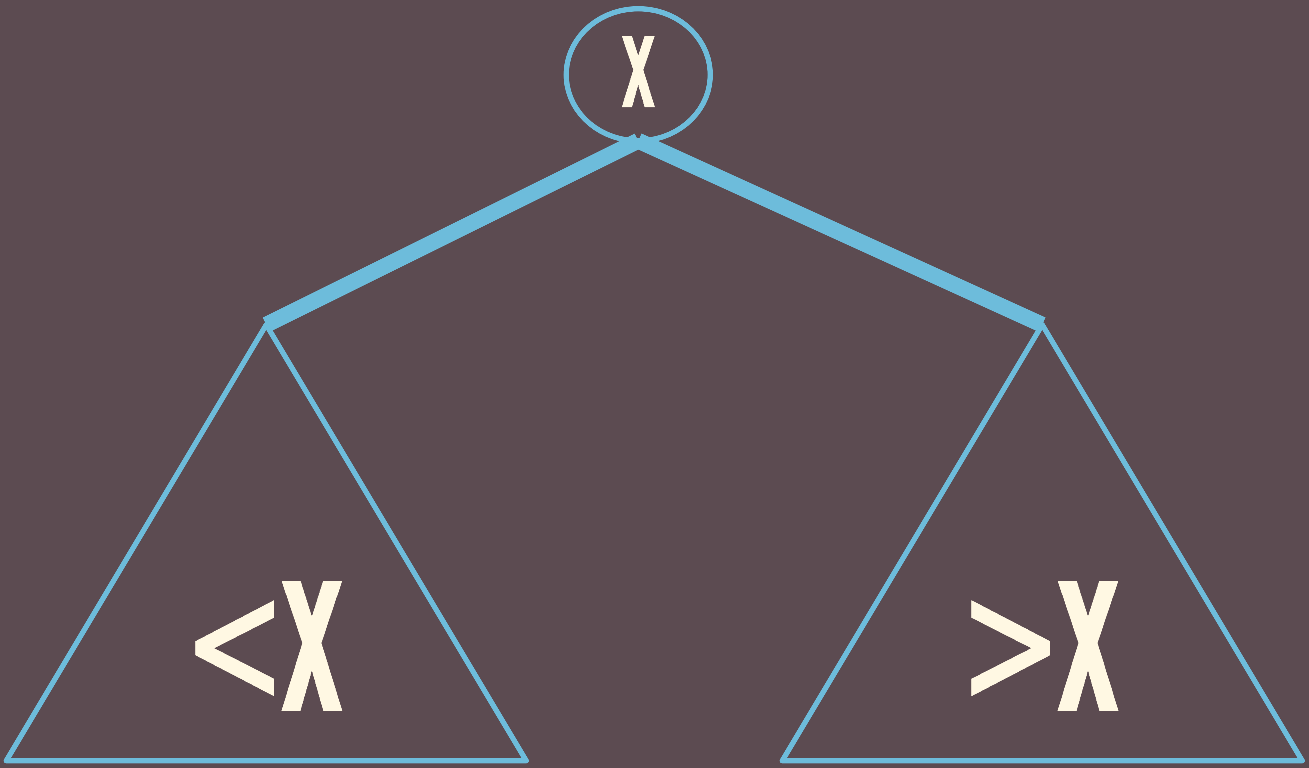
BINARY SEARCH TREE

BST

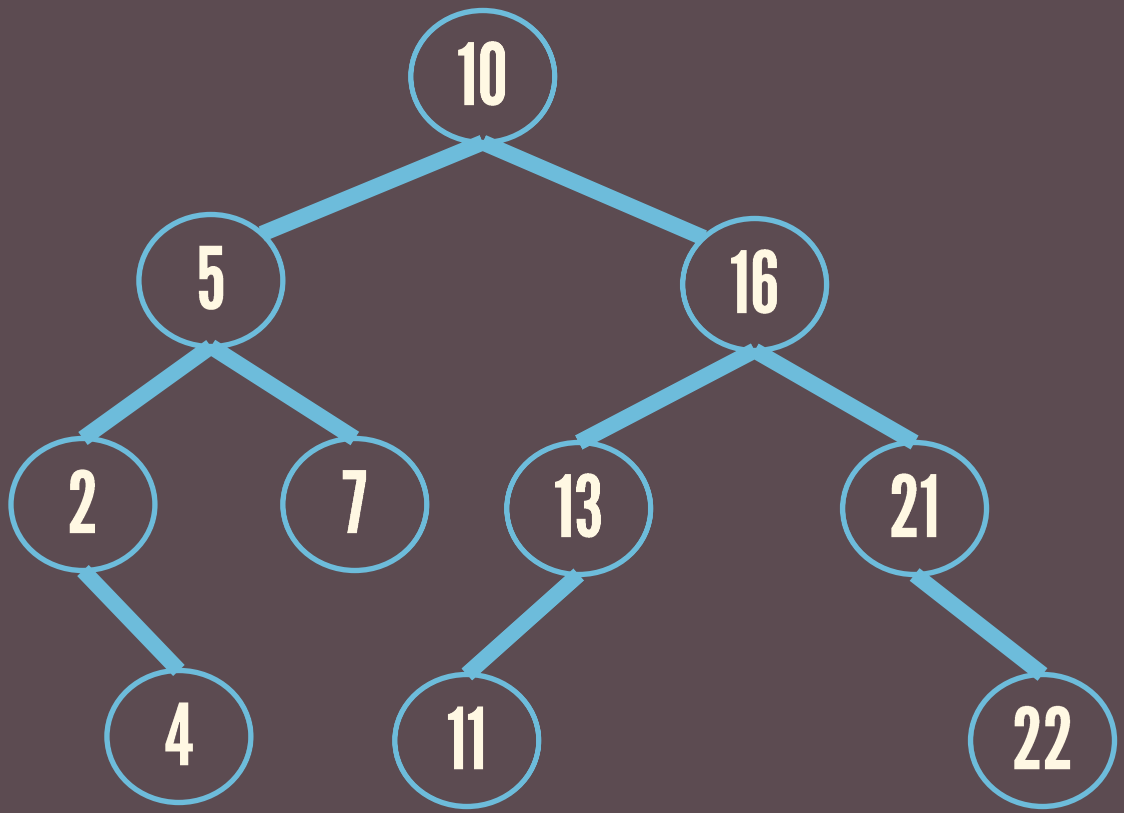
For every node, X in the tree,

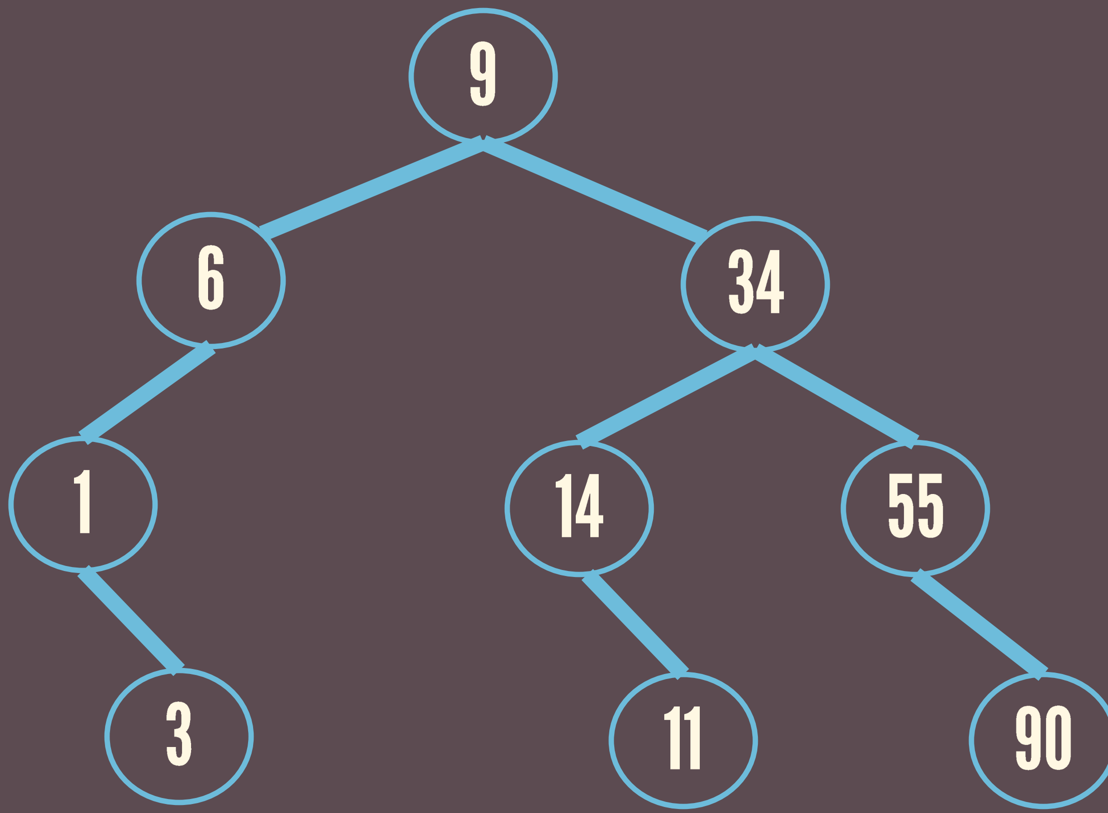
the values of all the keys in the left subtree are less than the key value in X; and

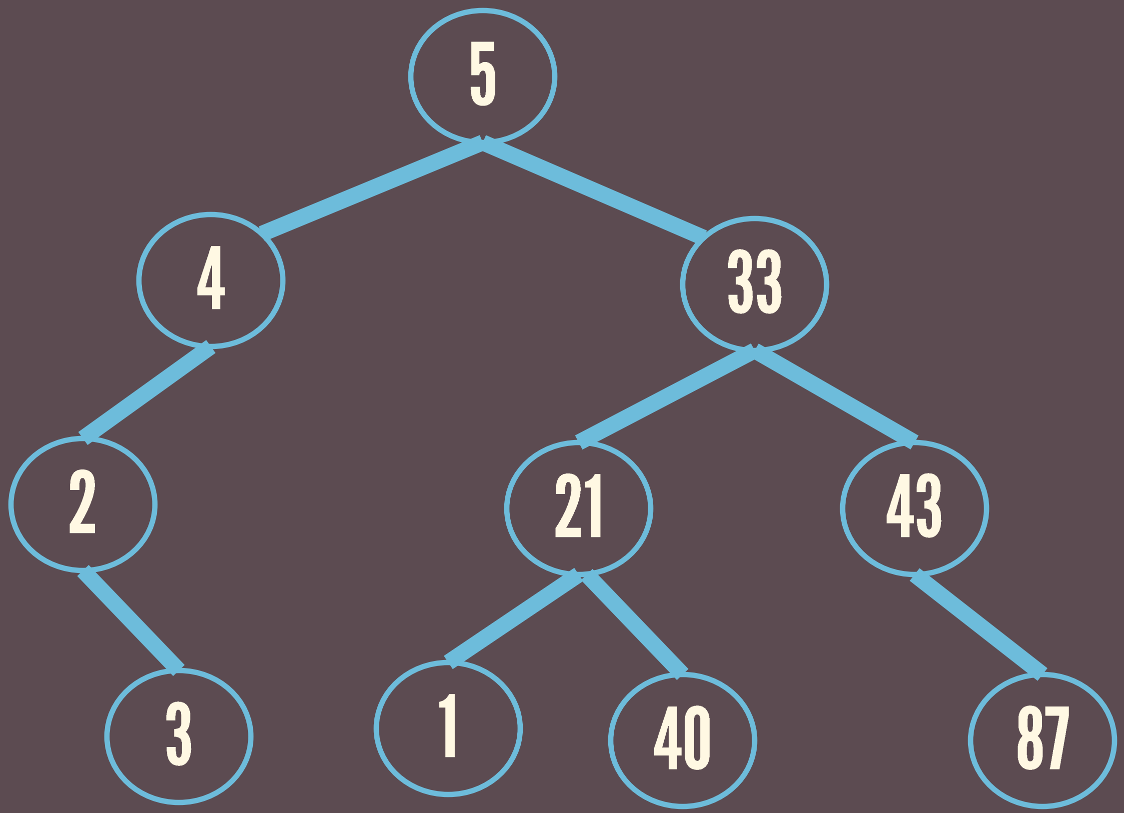
the values of all the keys in the right subtree are larger than the key value in X.

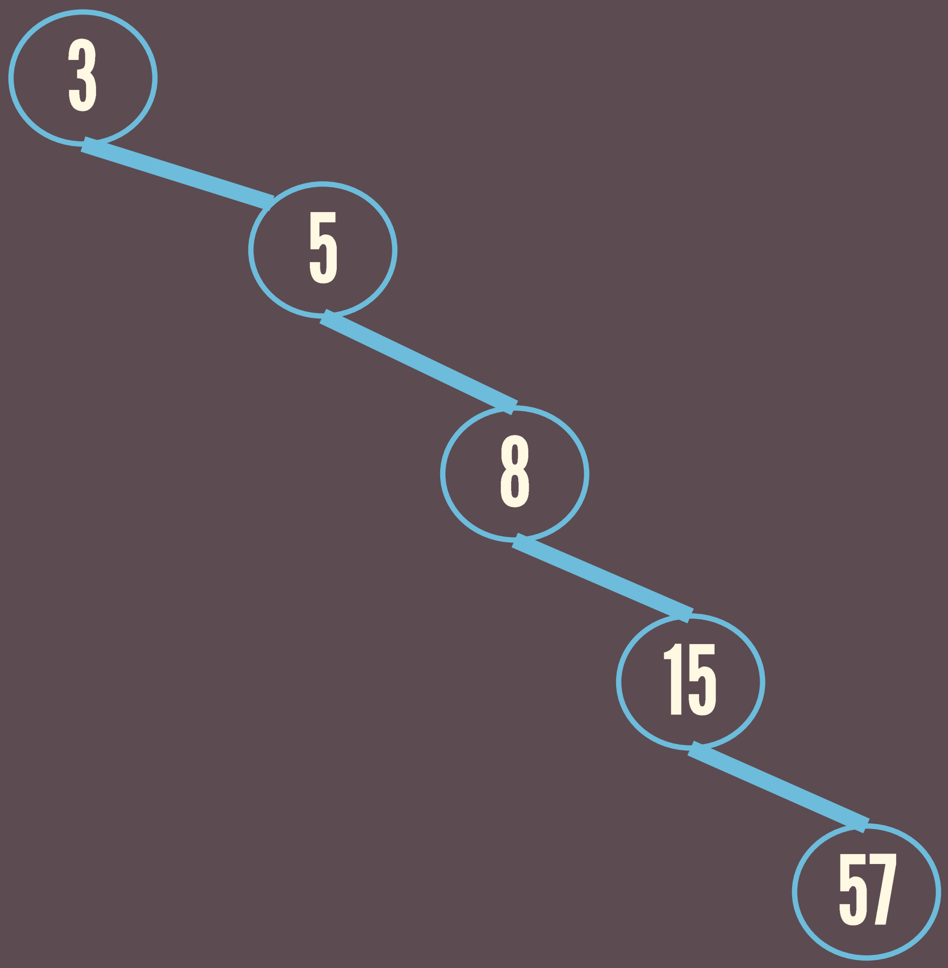


X
<X
>X


10
5
16
2
7
13
21
4
11
22


9
6
34
1
14
55
3
11
90


5
4
33
2
21
43
3
1
40
87


3
5
8
15
57

BST

OPERATIONS

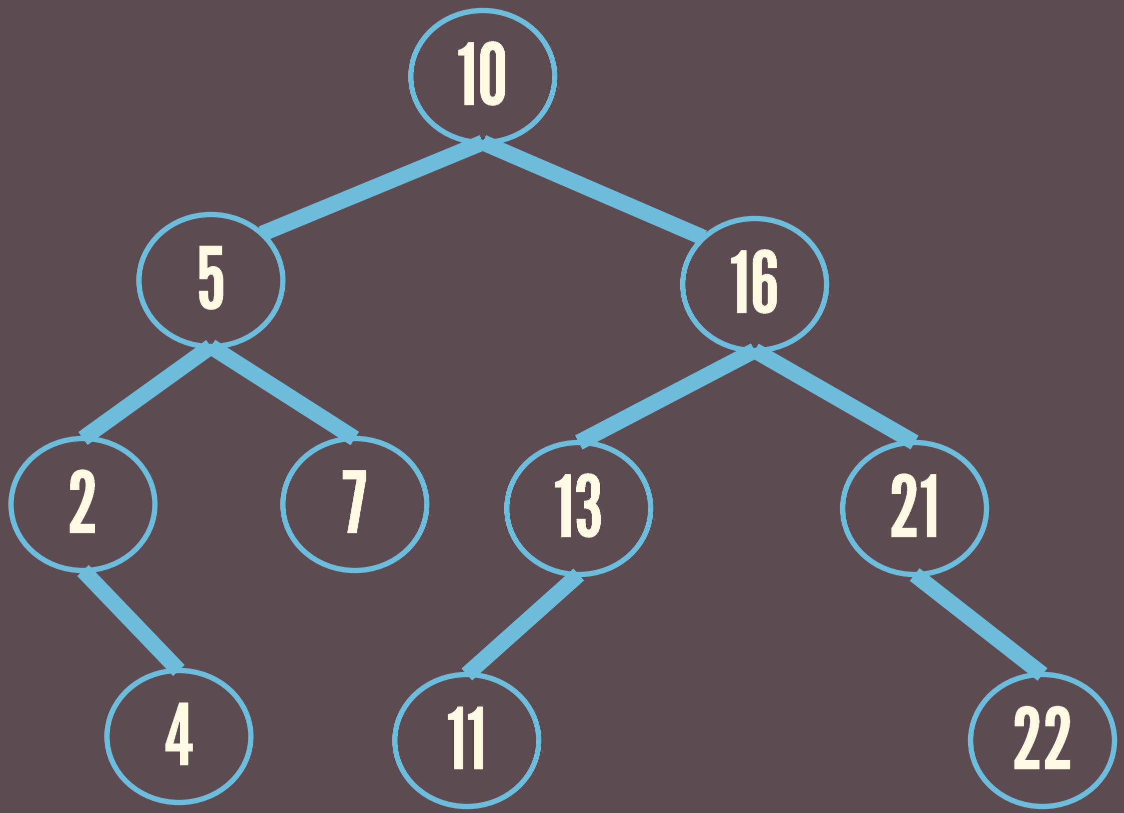
find
insert
delete
minimum
maximum
successor
predecessor

IMPLEMENTATIONS

recursive
non-recursive

IMPLEMENTATIONS

`find()`


10
5
16
2
7
13
21
4
11
22

```
typedef struct node{
  int data;
  struct node *left;
  struct node *right;
}BST;

BST *t;
```

```
BST find(int x, BST *t){

   if(t==NULL)
        return NULL;
   if(x < t->data)
        return ( ______________________ );
   if(x > t->data)
        return ( ______________________ );
   else
        return t;
}
```

```
BST find(int x, BST *t){

   if(t==NULL)
         return NULL;
   if(x < t->data)
         return (find(x, t->left) );
   if(x > t->data)
         return (find(x, t->right));
   else
         return t;
}
```

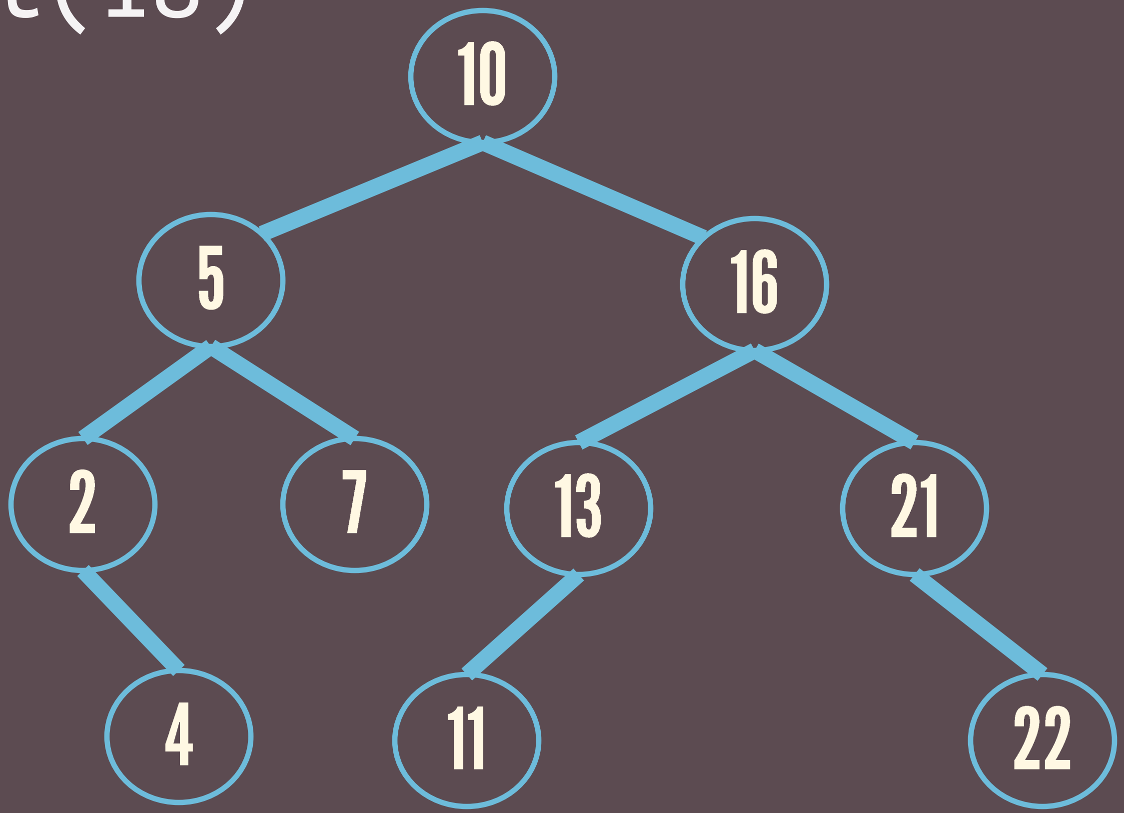
IMPLEMENTATIONS

`find()`

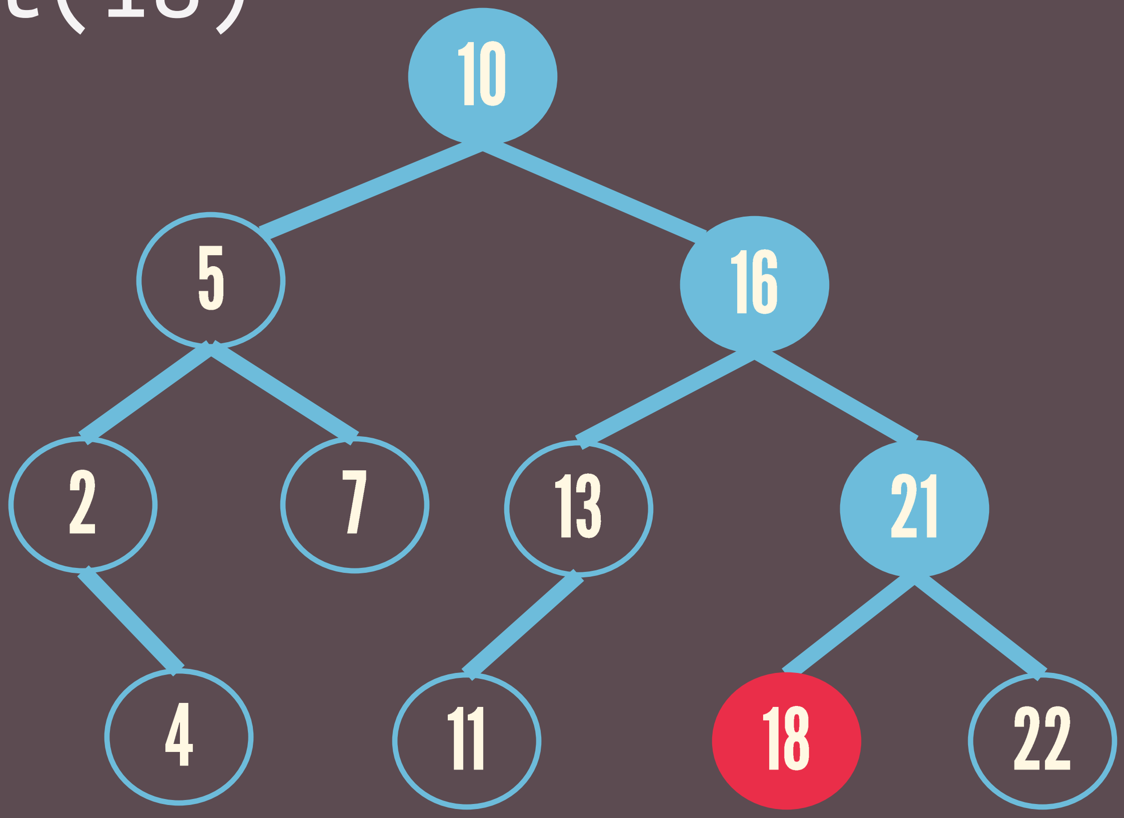
IMPLEMENTATIONS

`insert()`

```
insert(18)
```



insert(18)



IMPLEMENTATIONS

minimum()

IMPLEMENTATIONS

`maximum()`

IMPLEMENTATIONS

`predecessor()`

IMPLEMENTATIONS

`successor()`

IMPLEMENTATIONS

printBST()

```
void printBST(BST *root,int tabs){
  int i;
  if(root!=NULL){
    printBST(root->right,tabs+1);
    for(i=0;i<tabs;i++)
        printf("\t");
    printf("%3i\n",root->value);
    printBST(root->left,tabs+1);


  }
}
```

IMPLEMENTATIONS

`delete()`

delete()

3 cases

Node is a leaf

Node has one child

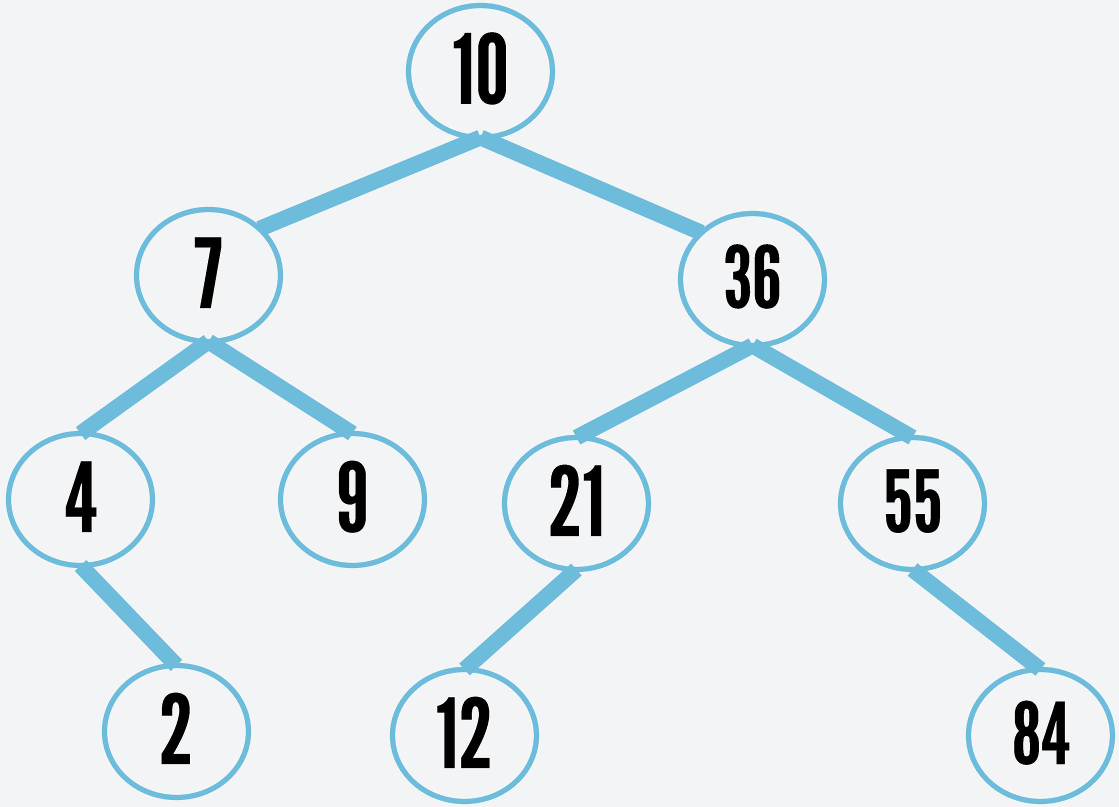
Node has two children

Node is a leaf

The node can be deleted immediately.

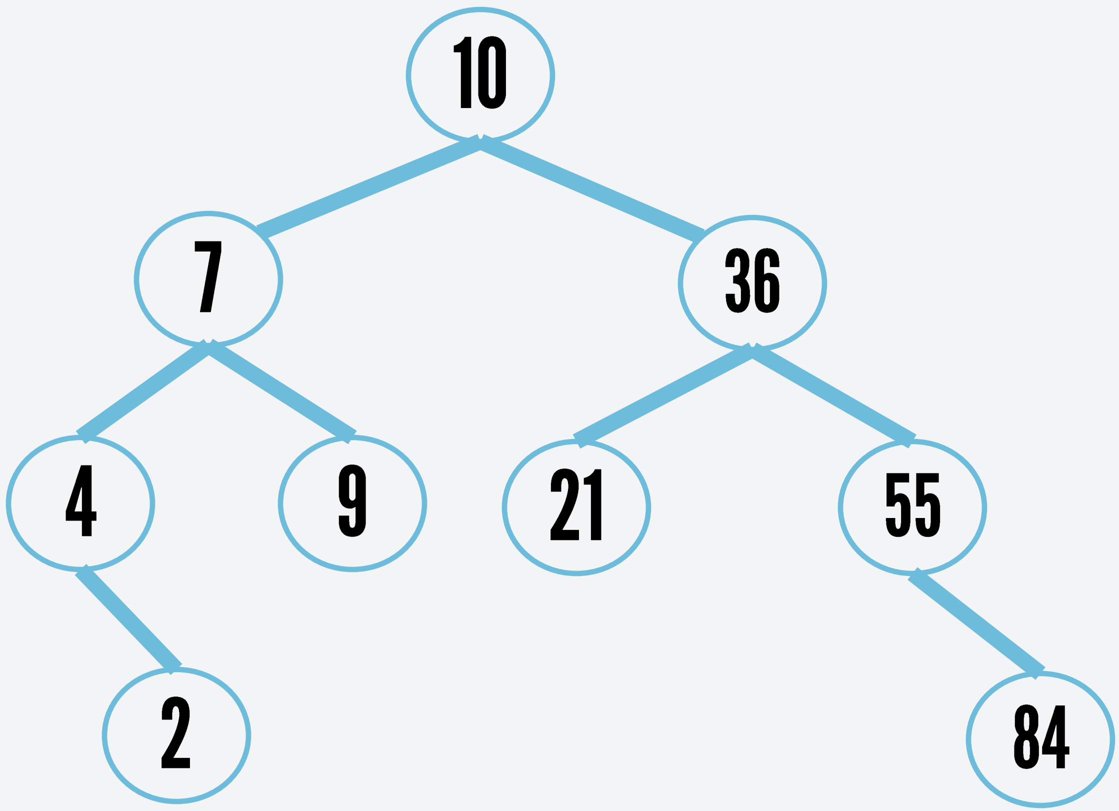
Node is a leaf

`delete(12)`



`delete(12)`

Node is a leaf

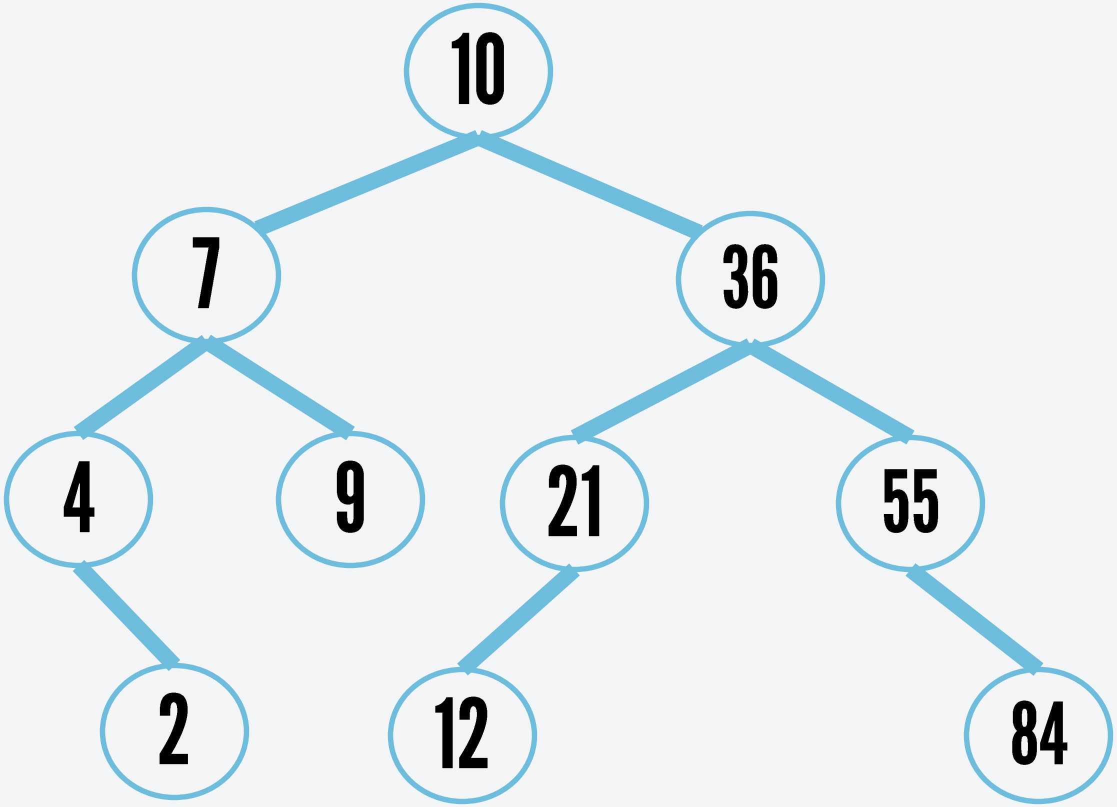


Node has one child

Its parent adjusts a pointer to bypass the node.

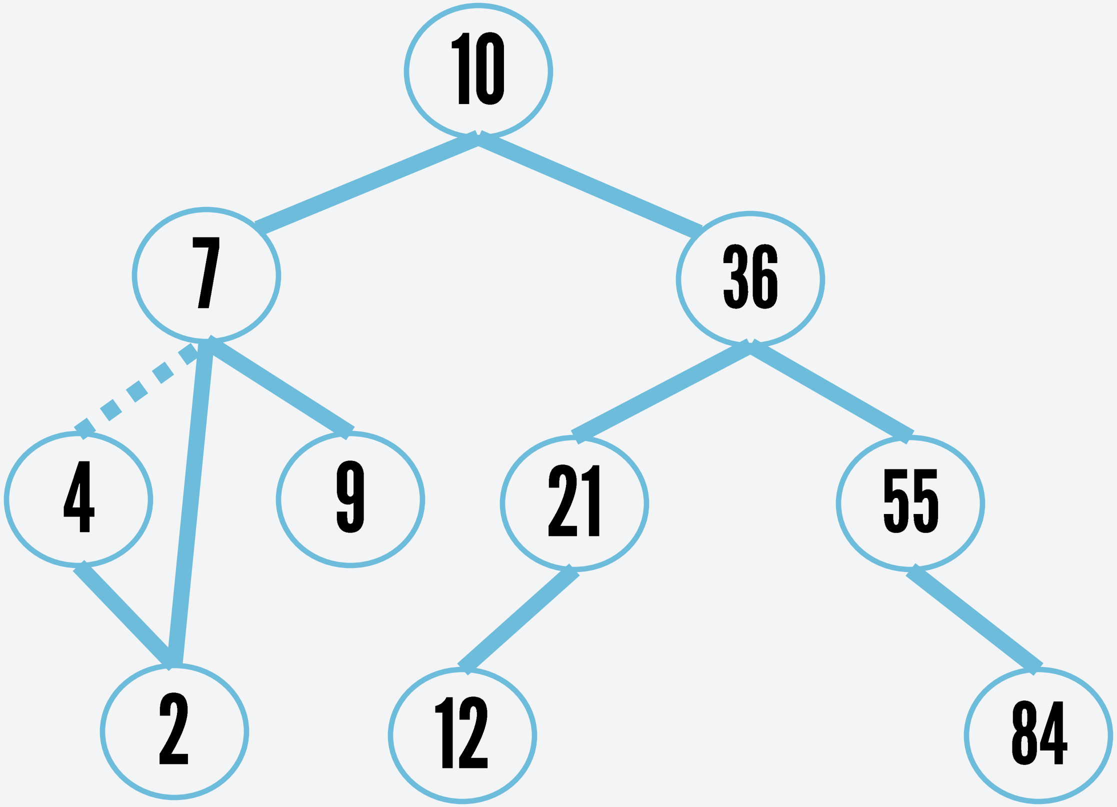
delete(4)

Node has one child



delete(4)

Node has one child



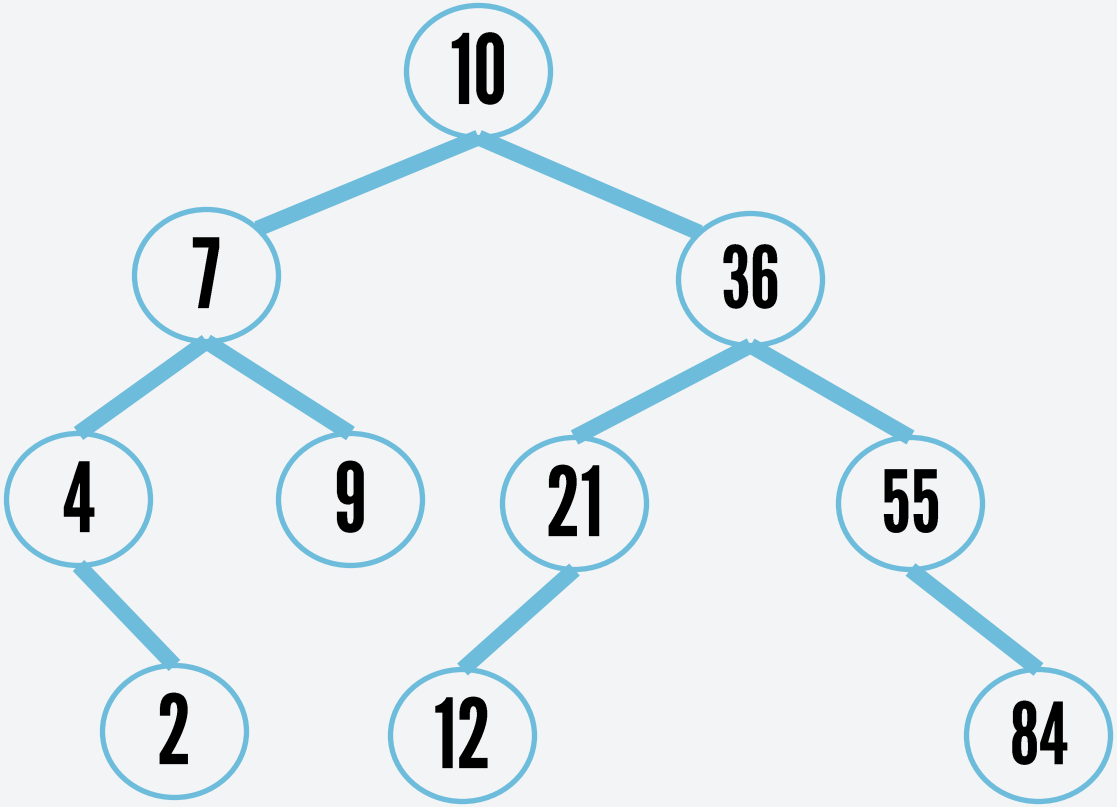
Node has two children

Replace this node with the smallest key of the right subtree.

Recursively delete this node.

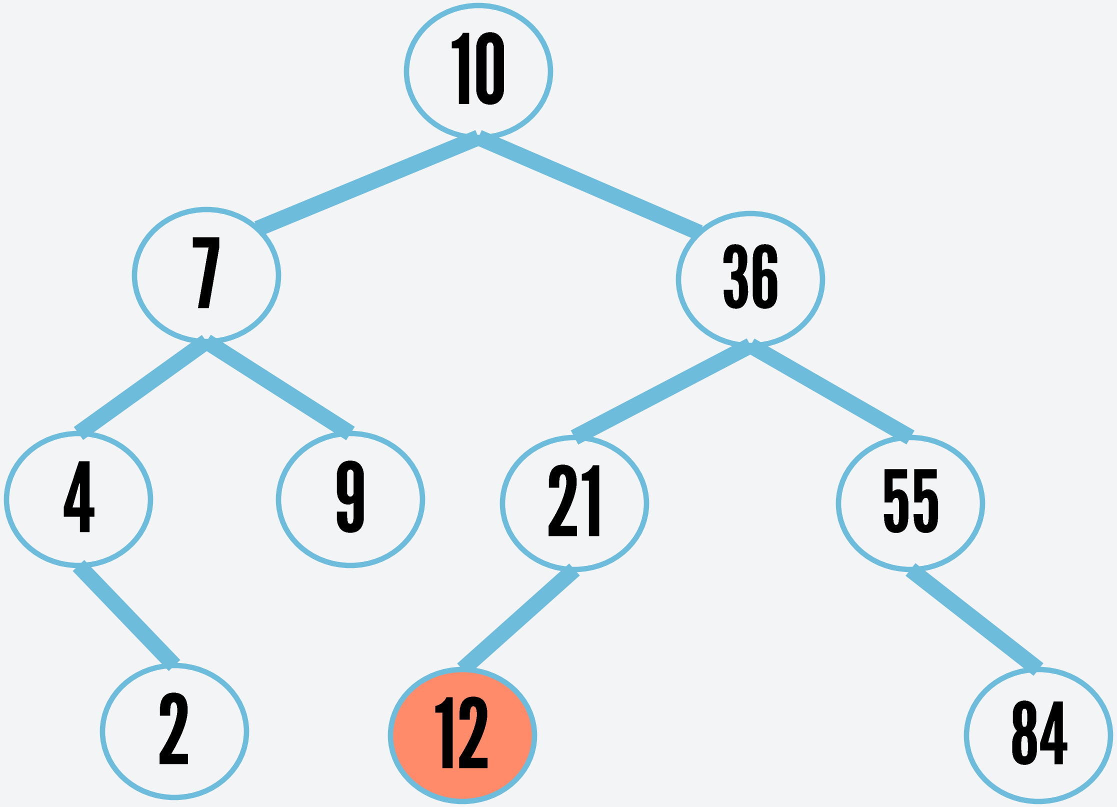
Node has two children

delete(10)



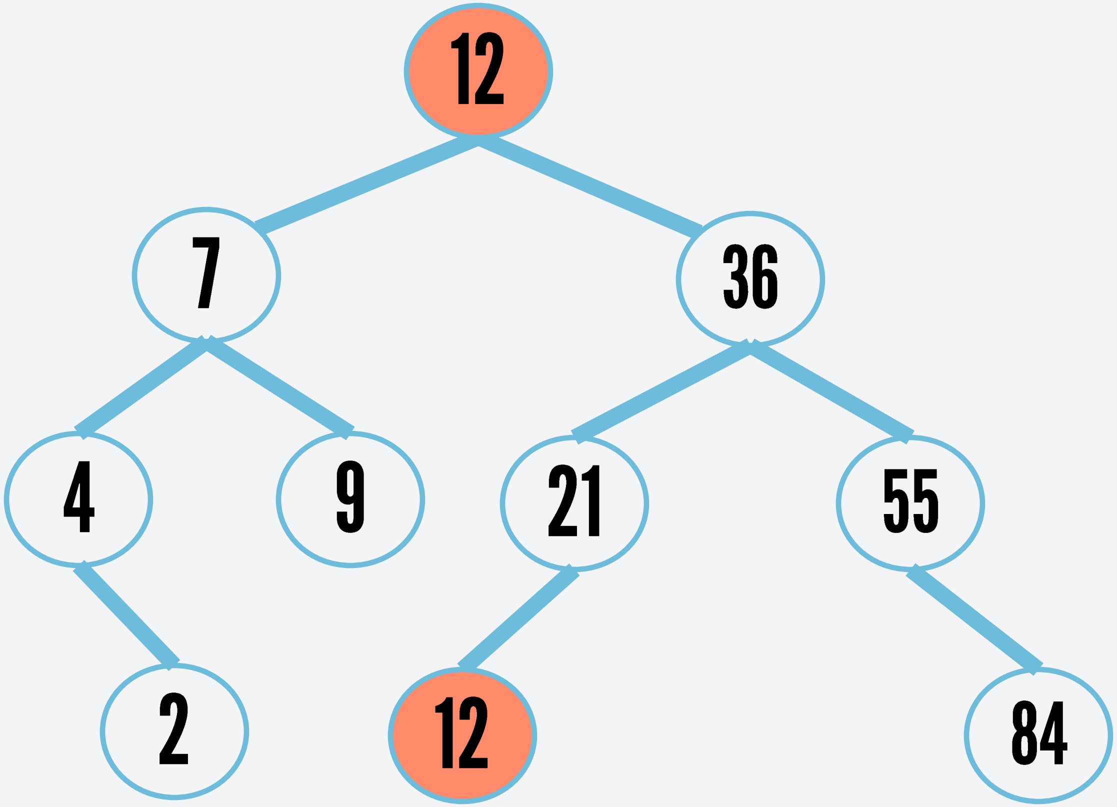
Node has two children

`delete(10)`



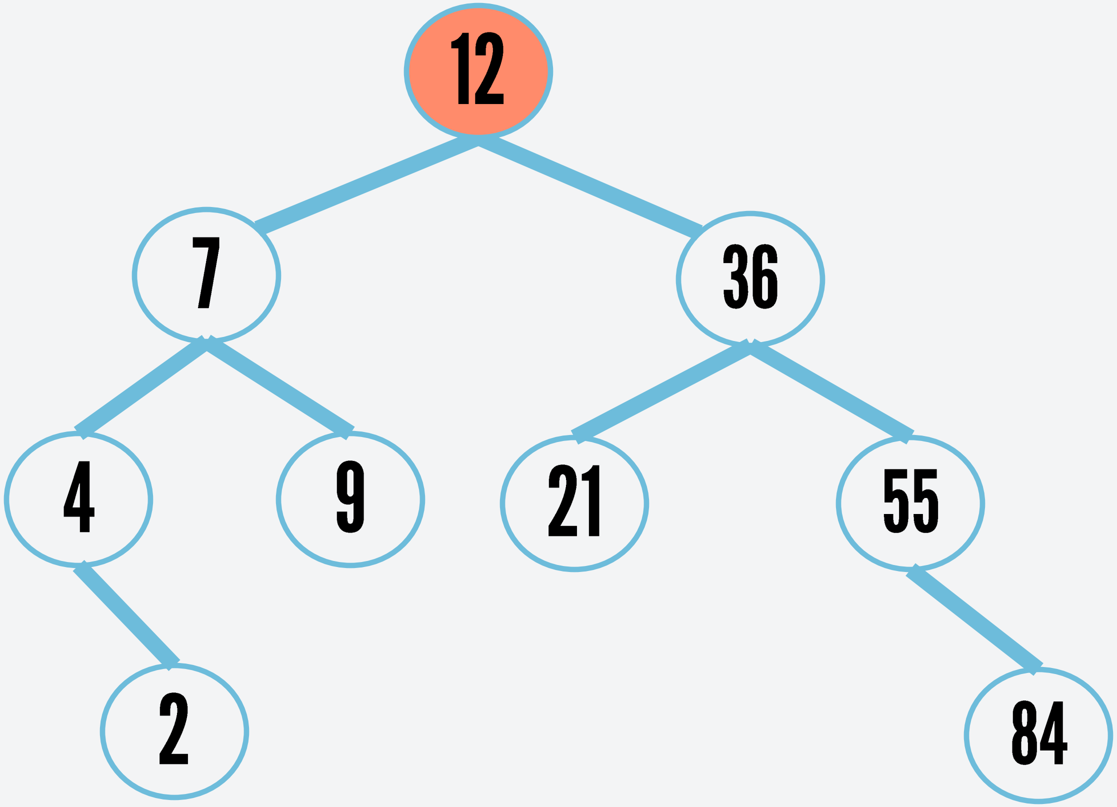
Node has two children

`delete(10)`



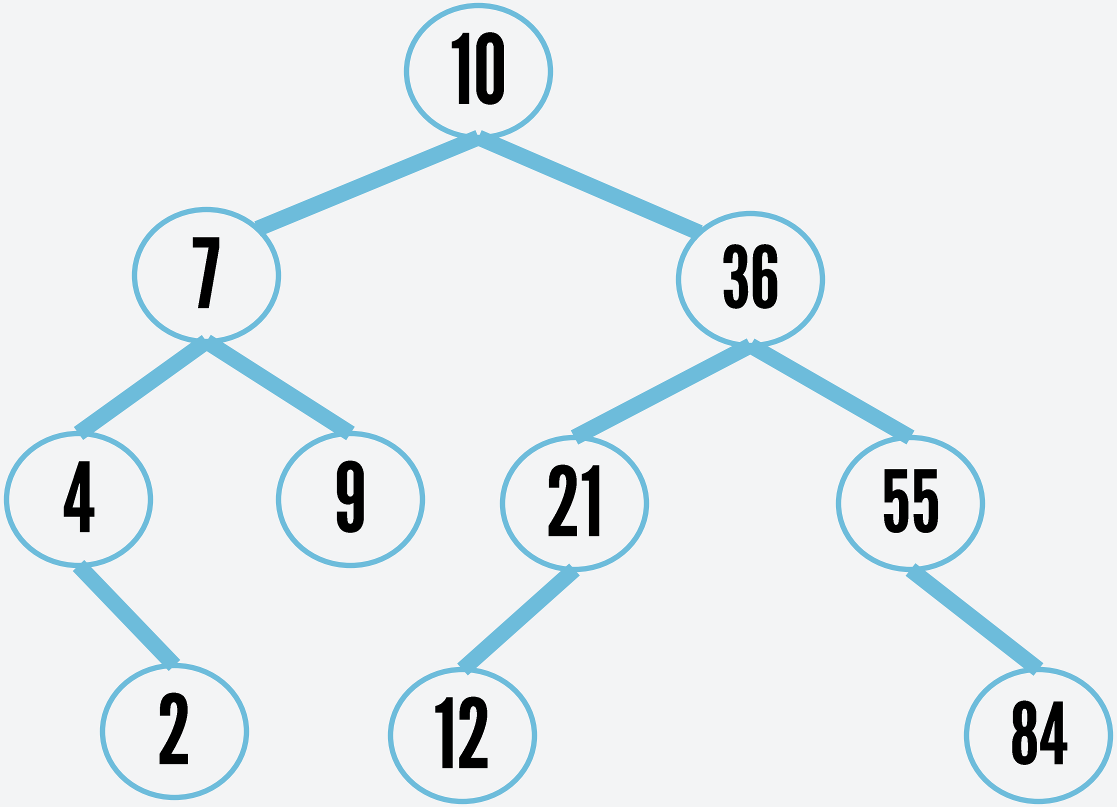
Node has two children

`delete(10)`



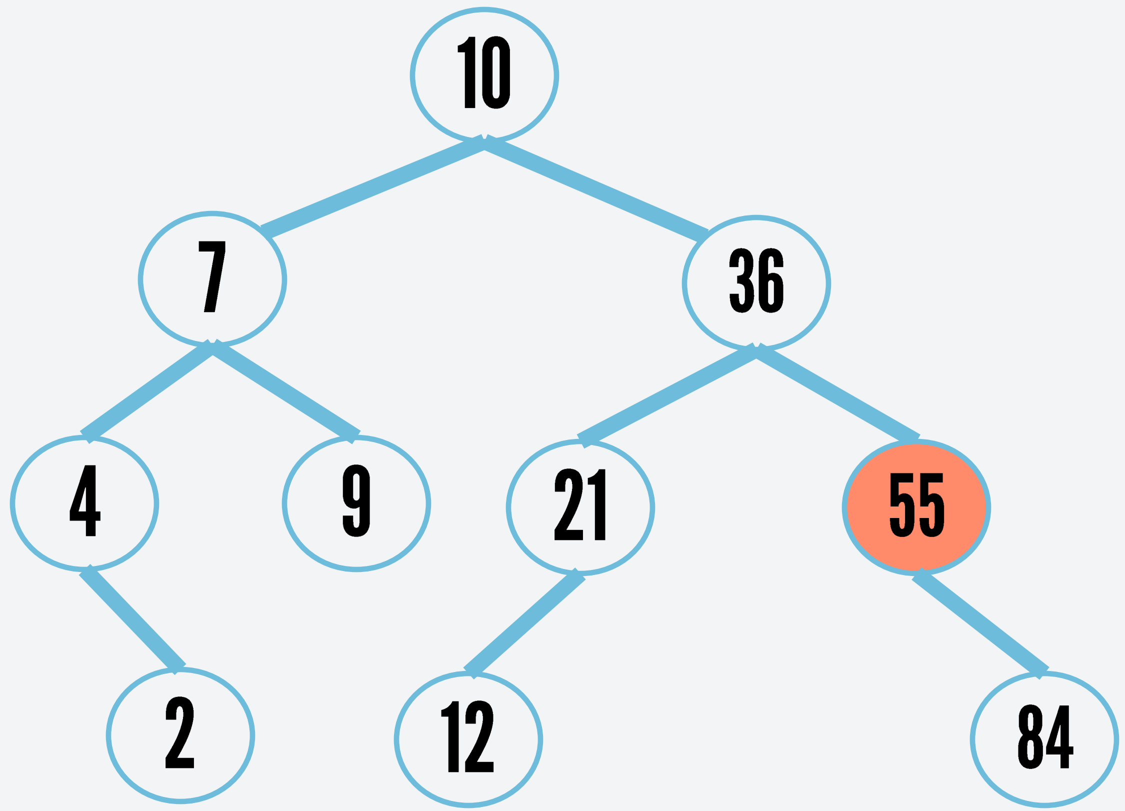
```
delete(36)
```

Node has two children



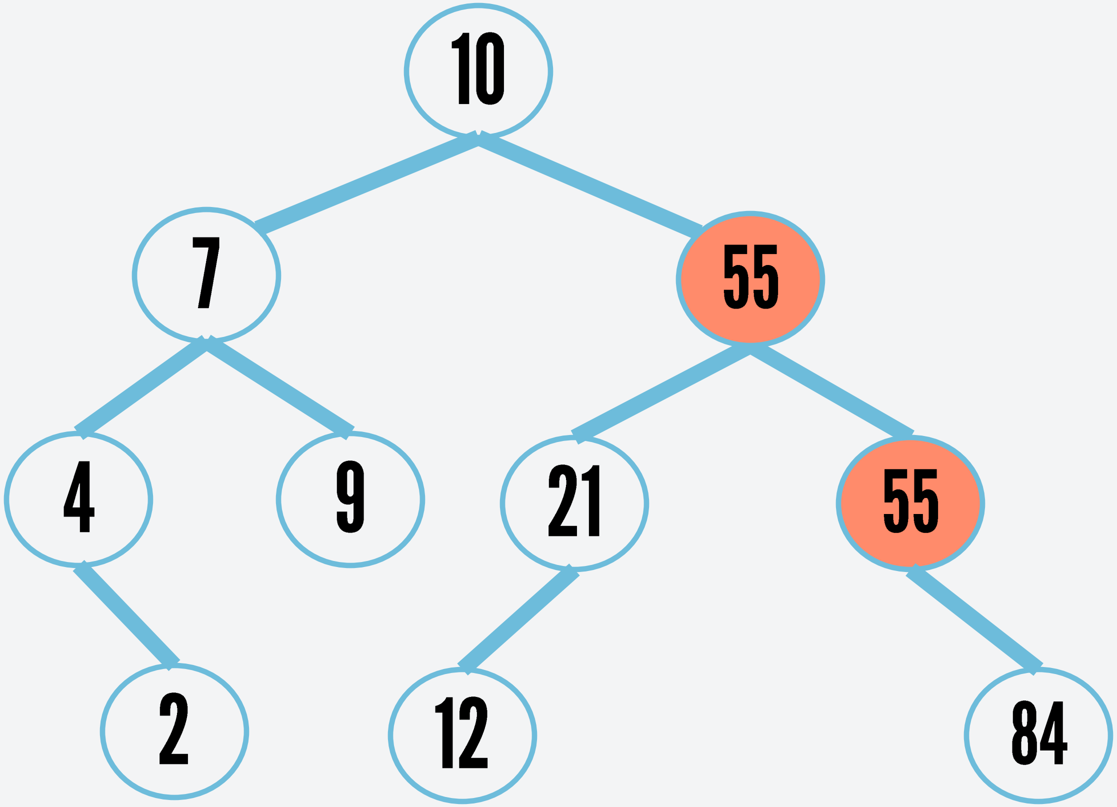
Node has two children

delete(36)



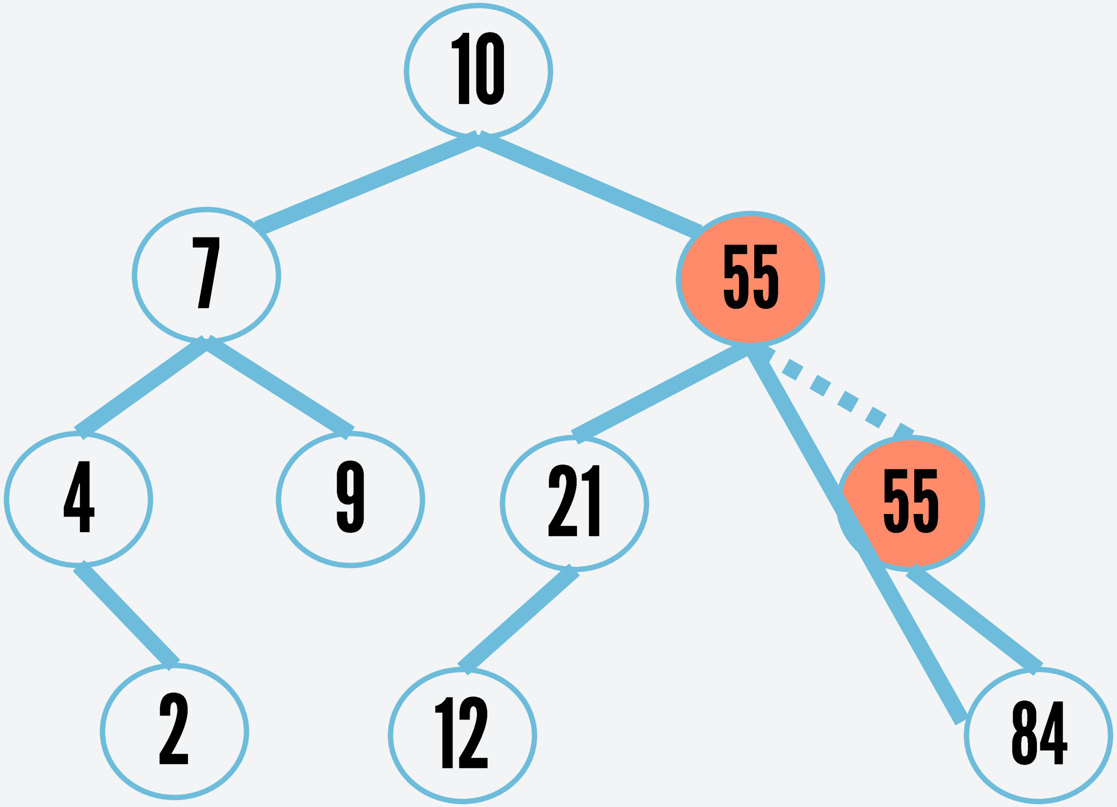
```
delete(36)
```

Node has two children



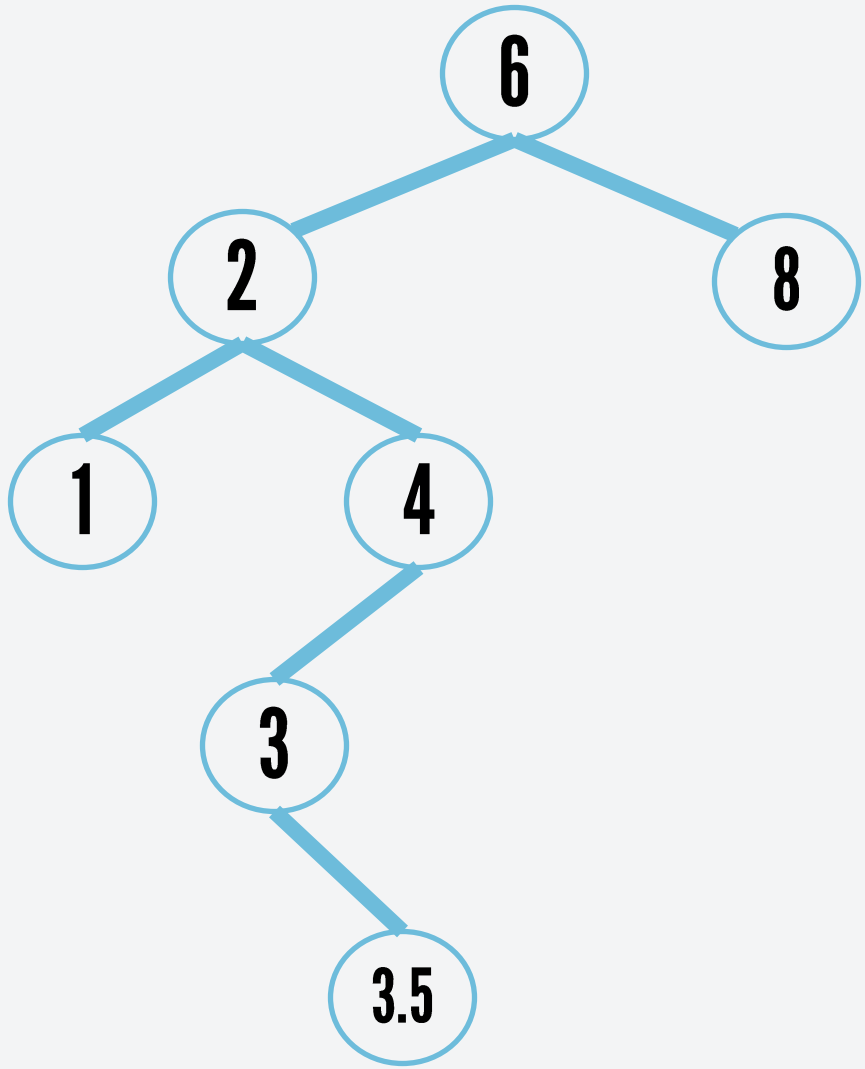
delete(36)

Node has two children



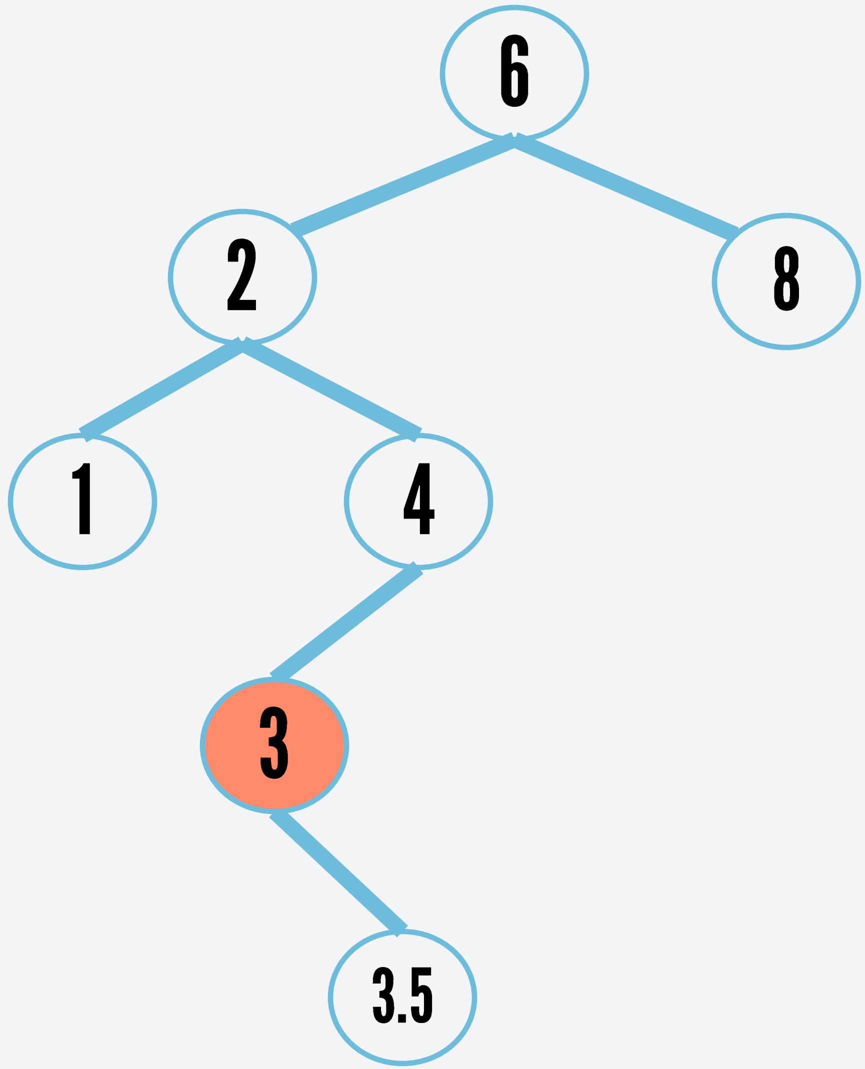
```
delete(2)
```

Node has two children



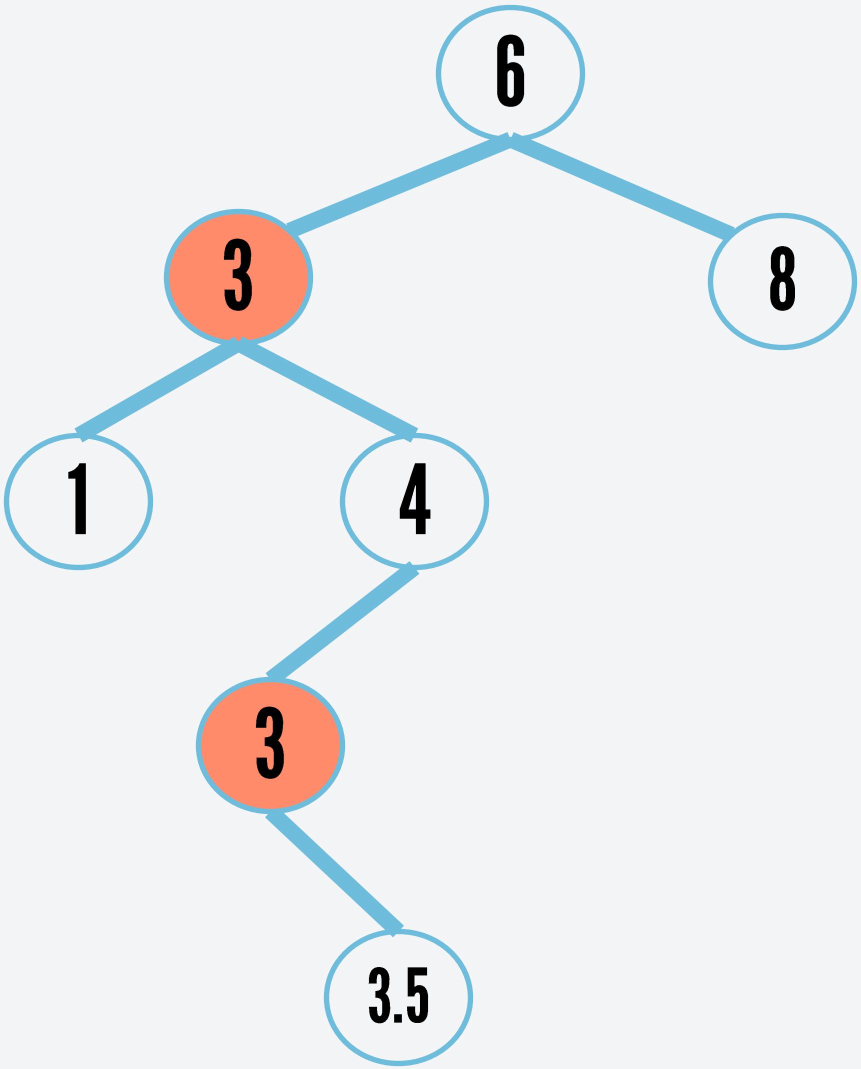
Node has two children

```
delete(2)
```



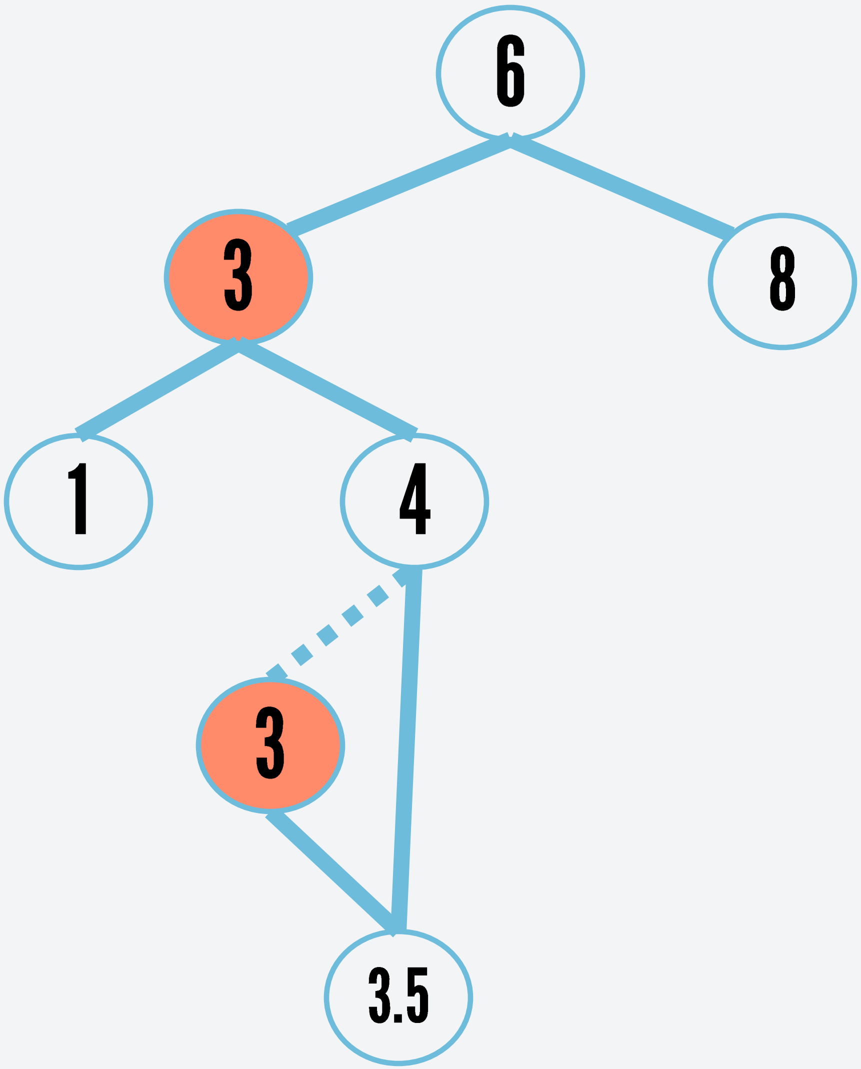
delete(2)

Node has two children



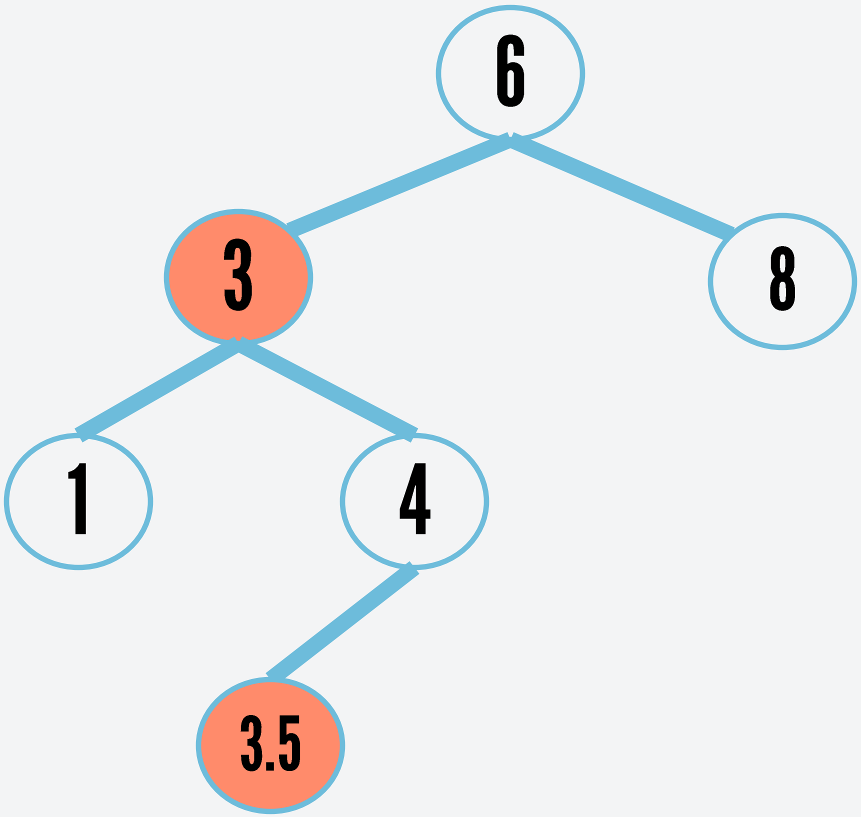
Node has two children

`delete(2)`



Node has two children

`delete(2)`



AVL TREE

ADELSON-VELSKII AND LANDIS' TREE

AVL TREE

A binary search tree with a balance condition.

AVL TREE

For every node in the tree, the height of its left and right subtrees can differ by at most 1.

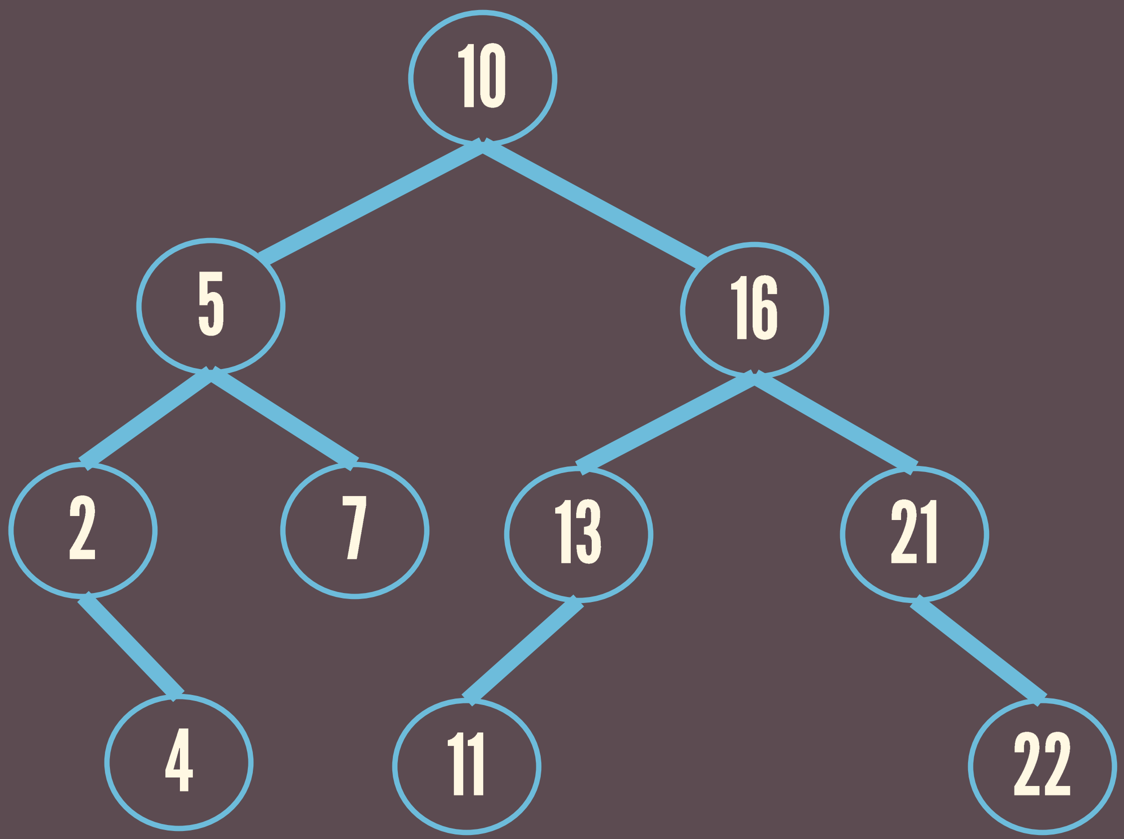
AVL TREE

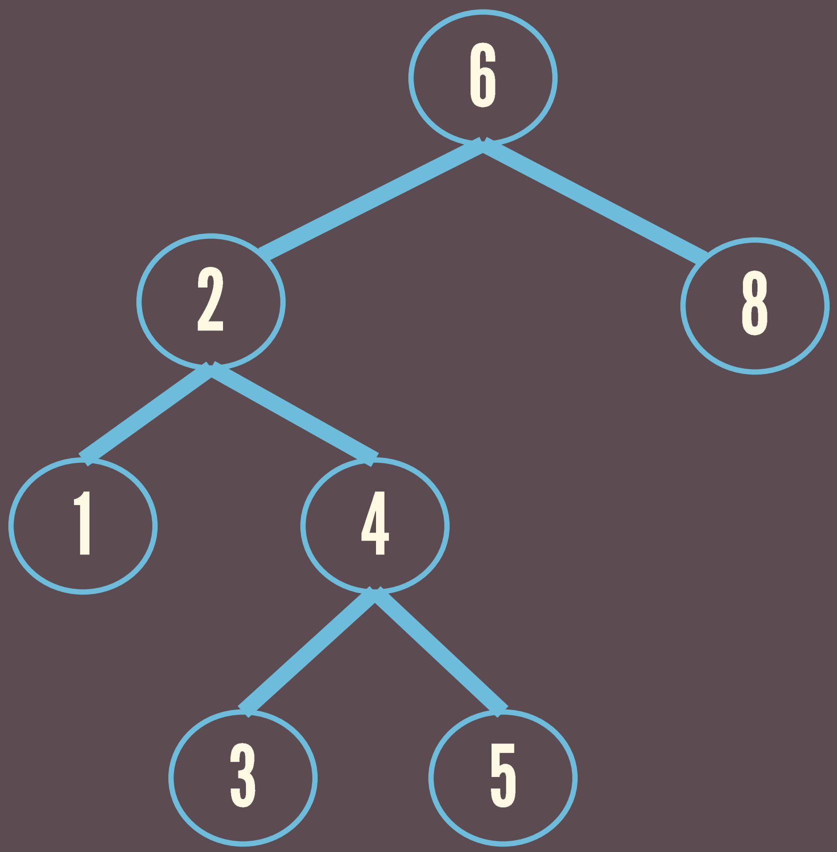
If anytime they differ by more than one, rebalancing is done to restore this property.

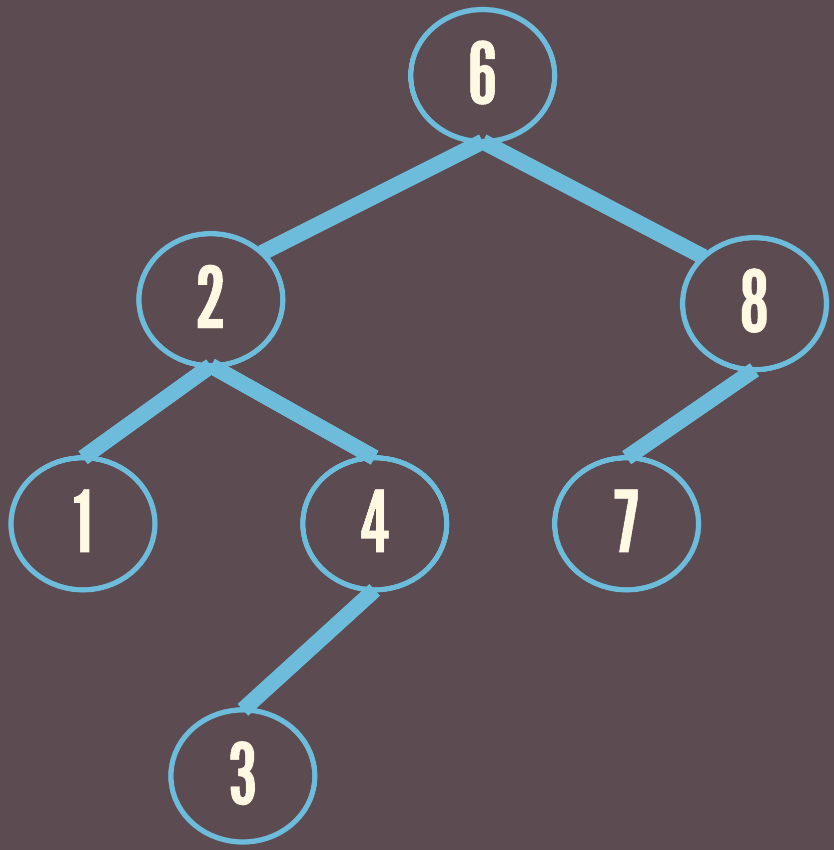
NOTE

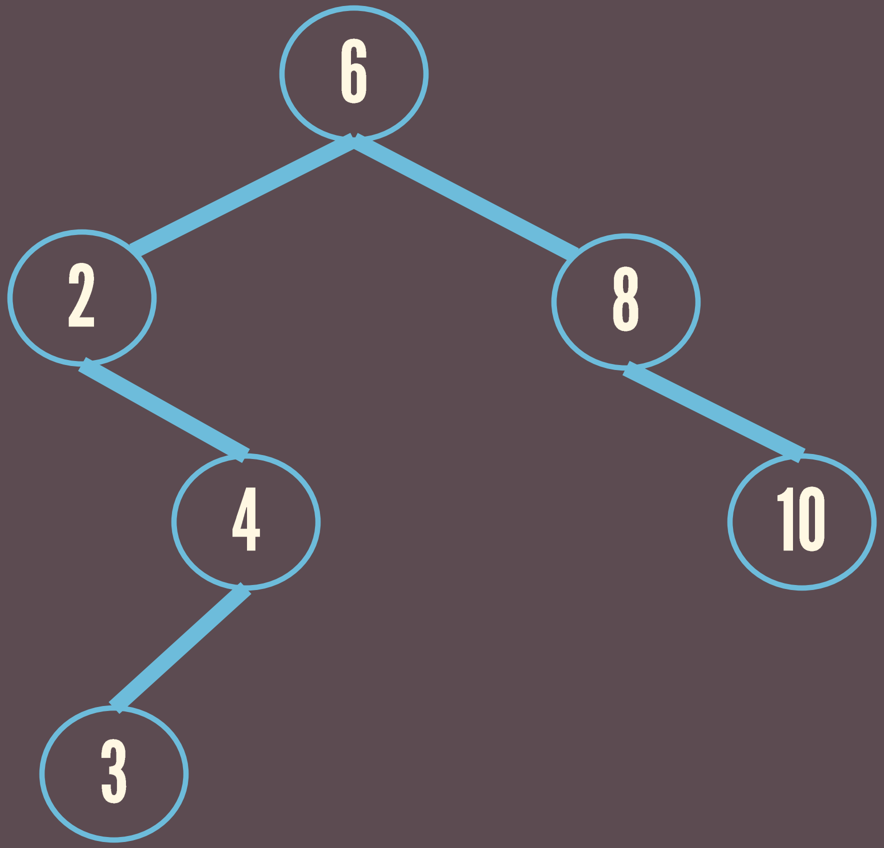
The height of an empty tree is -1.

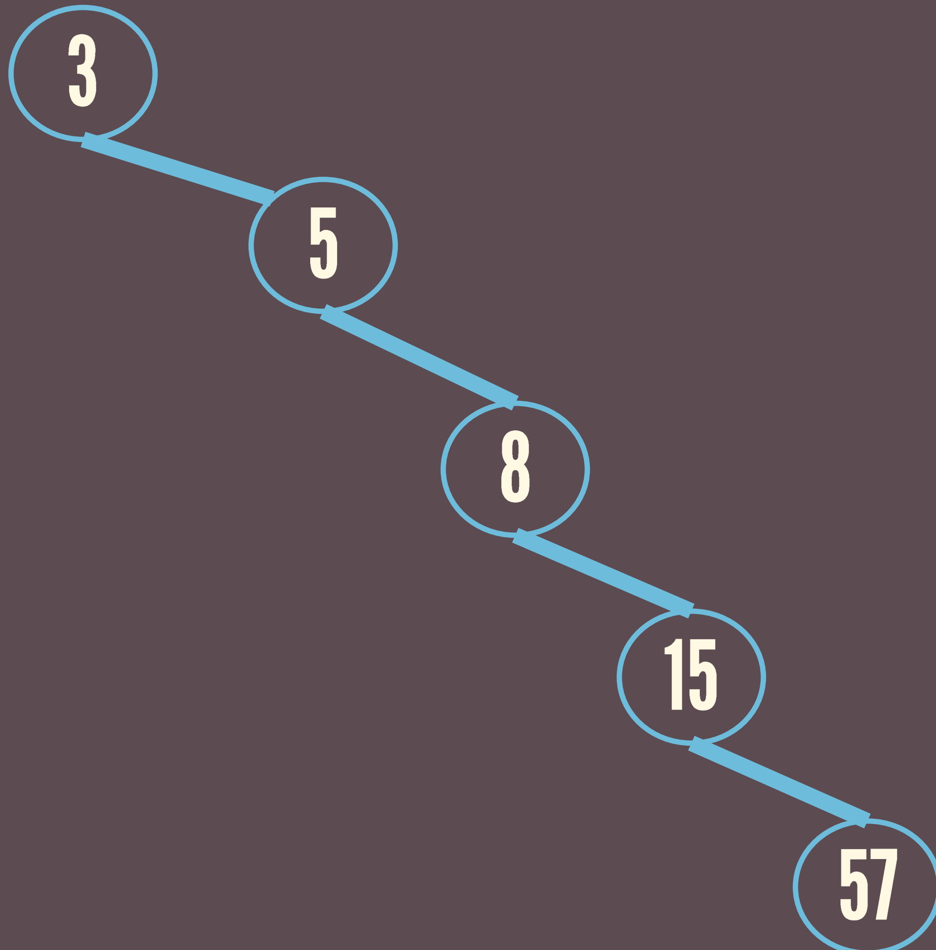
```c
typedef struct node{
  int data;
  struct node *left;
  struct node *right;
  int height;
}BST;

BST *t;
```

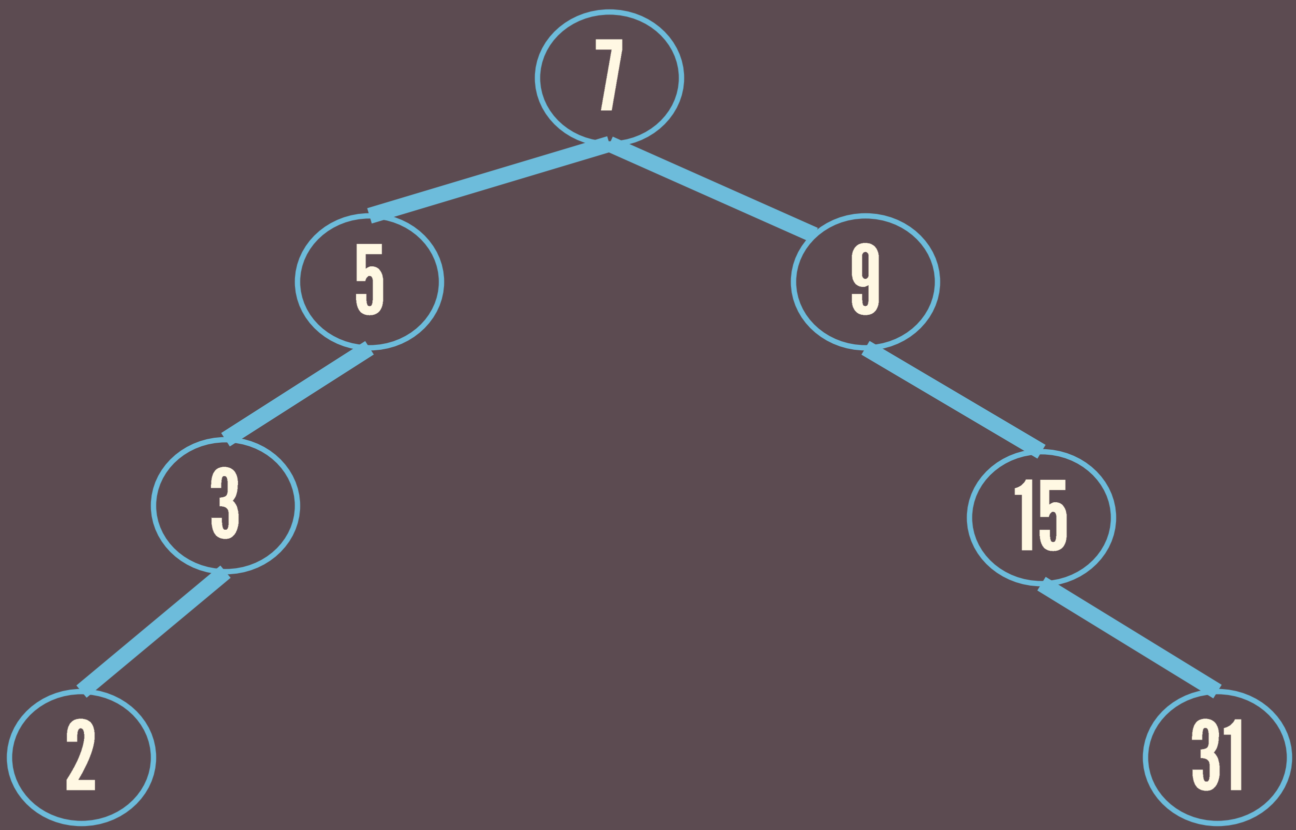

10
5
16
2
7
13
21
4
11
22


6
2
8
1
4
3
5


6
2
8
1
4
7
3


6
2
8
4
10
3


3
5
8
15
57


7
5
9
3
15
2
31

AVL

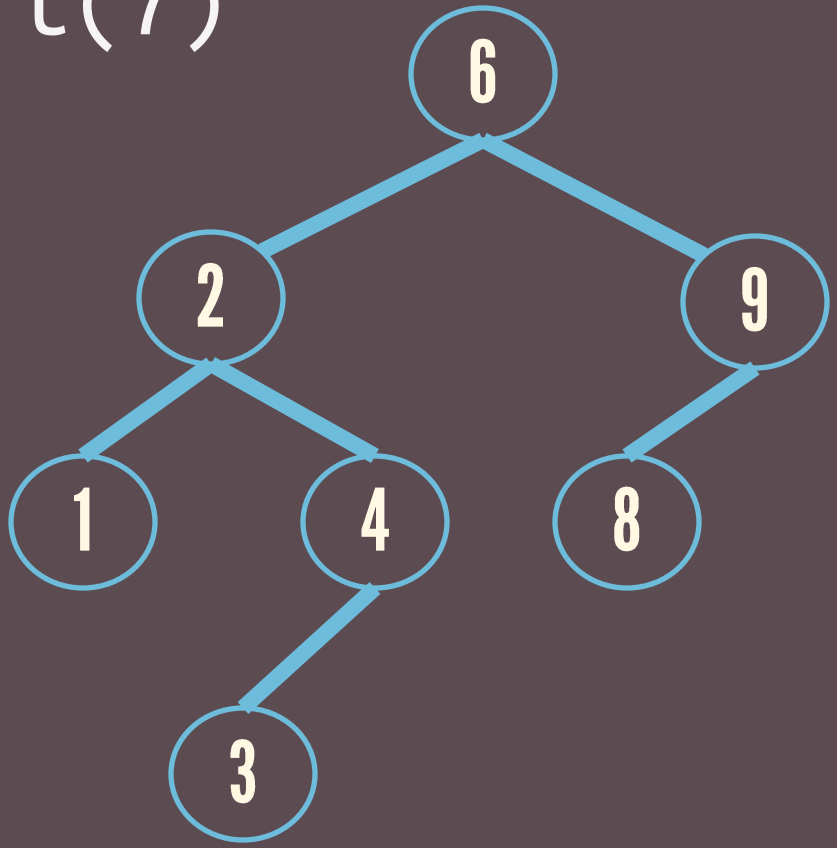
OPERATIONS

find
insert (with rotations)
delete (with rotations)
minimum
maximum
successor
predecessor

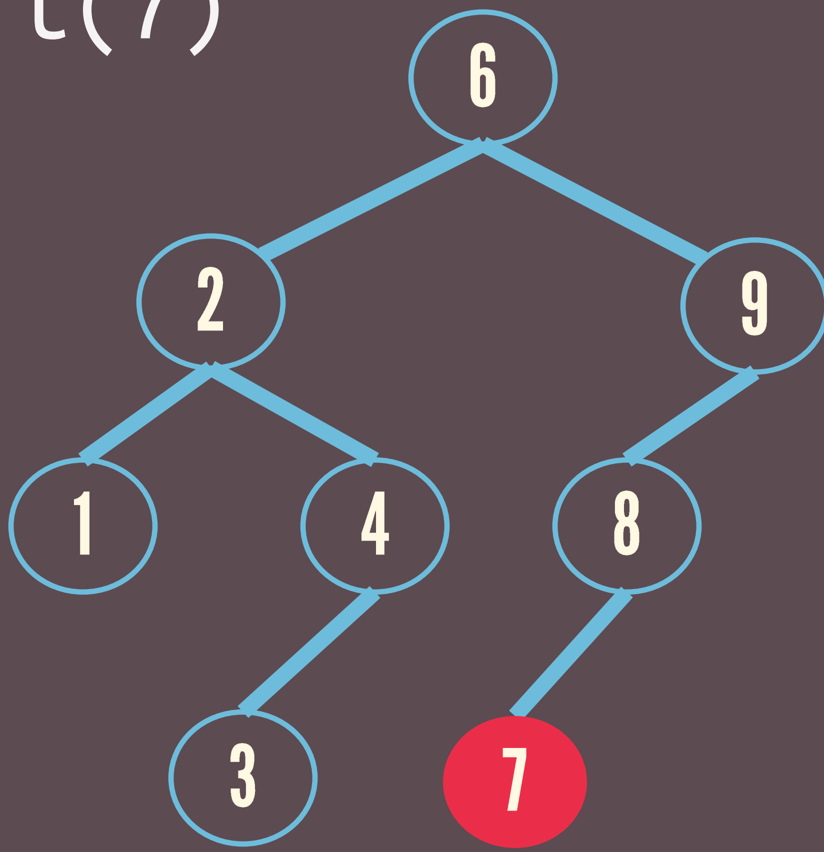
ROTATIONS

Rotations are done to maintain the AVL property.

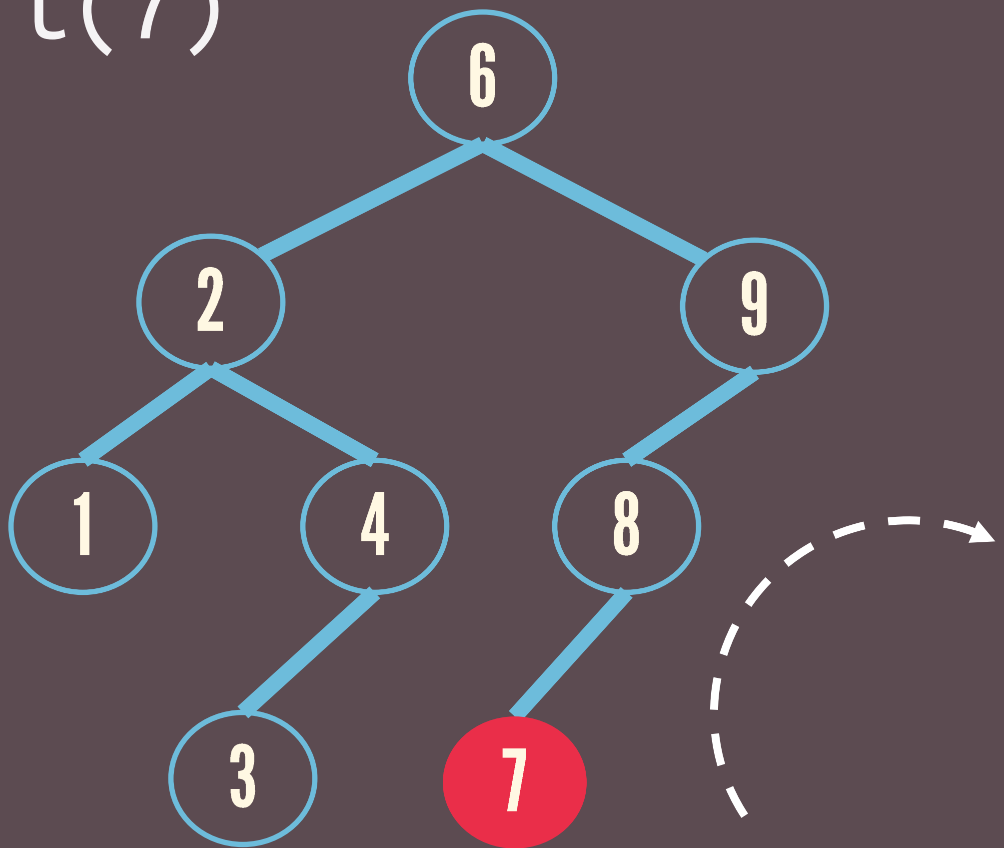
insert(7)



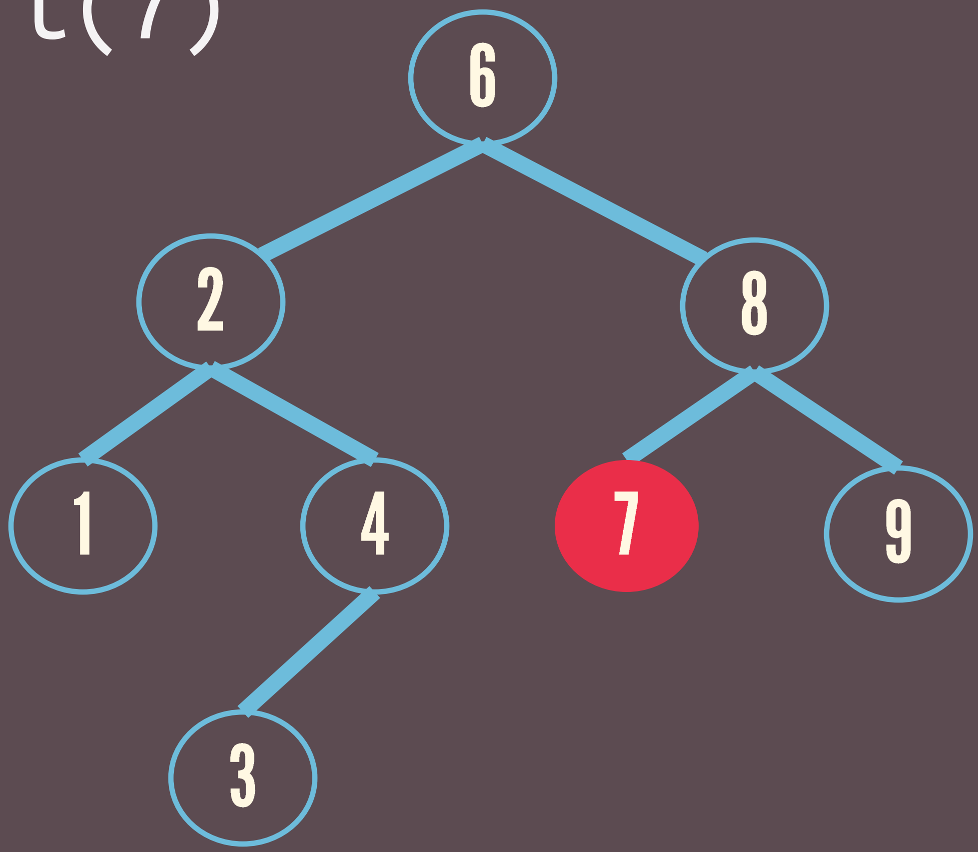
insert(7)



insert(7)



insert(7)



ROTATIONS

INSERT OPERATION

Single:

Left Rotate
Right Rotate

ROTATIONS

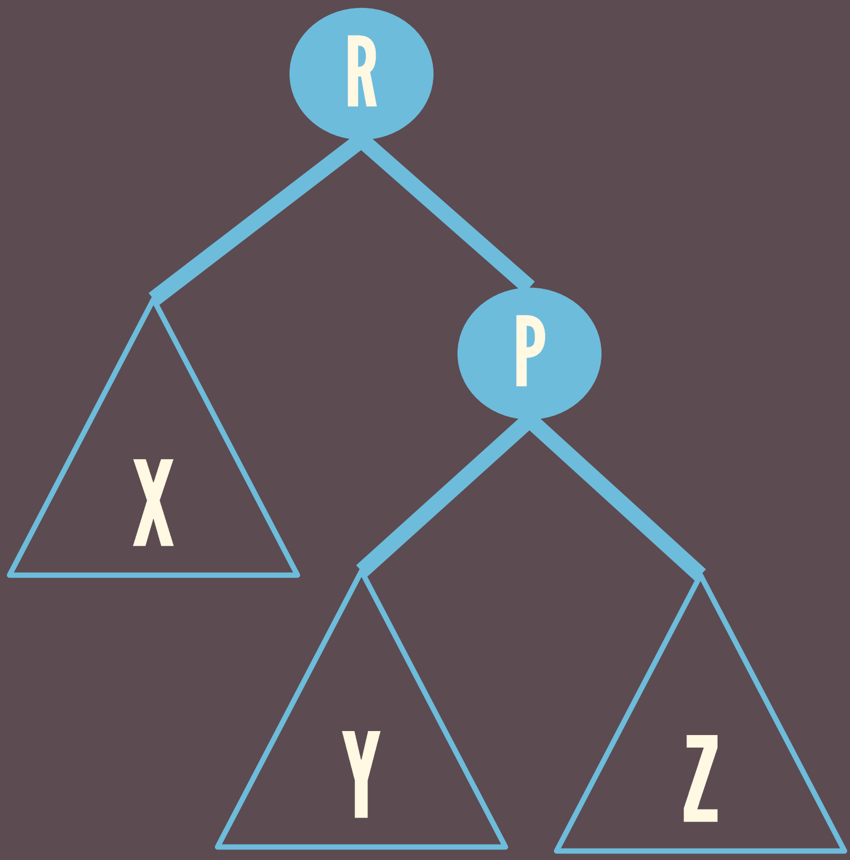
INSERT OPERATION

Double:

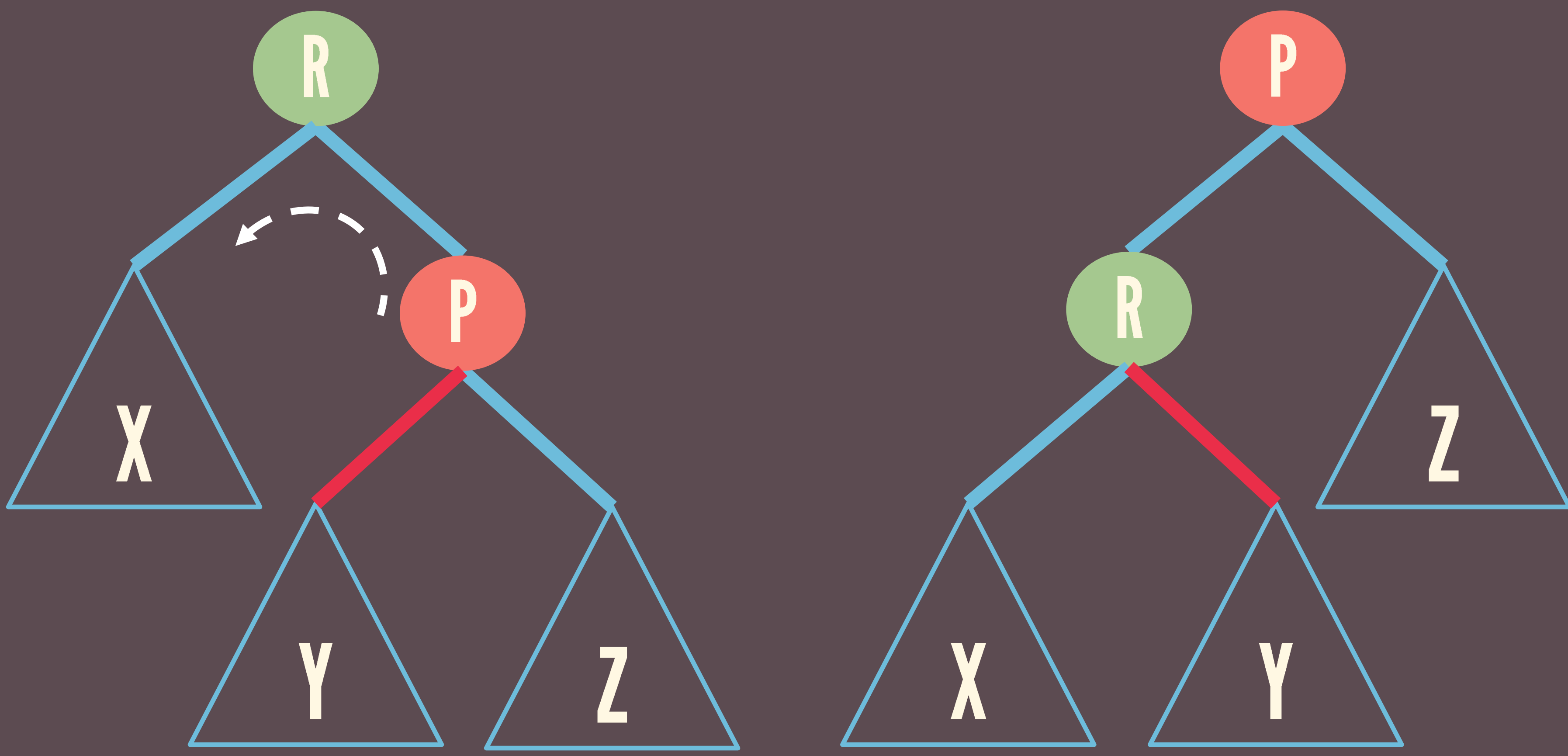
Left-right Rotate
Right-left Rotate

ROTATIONS

ILLUSTRATED



LEFT ROTATE

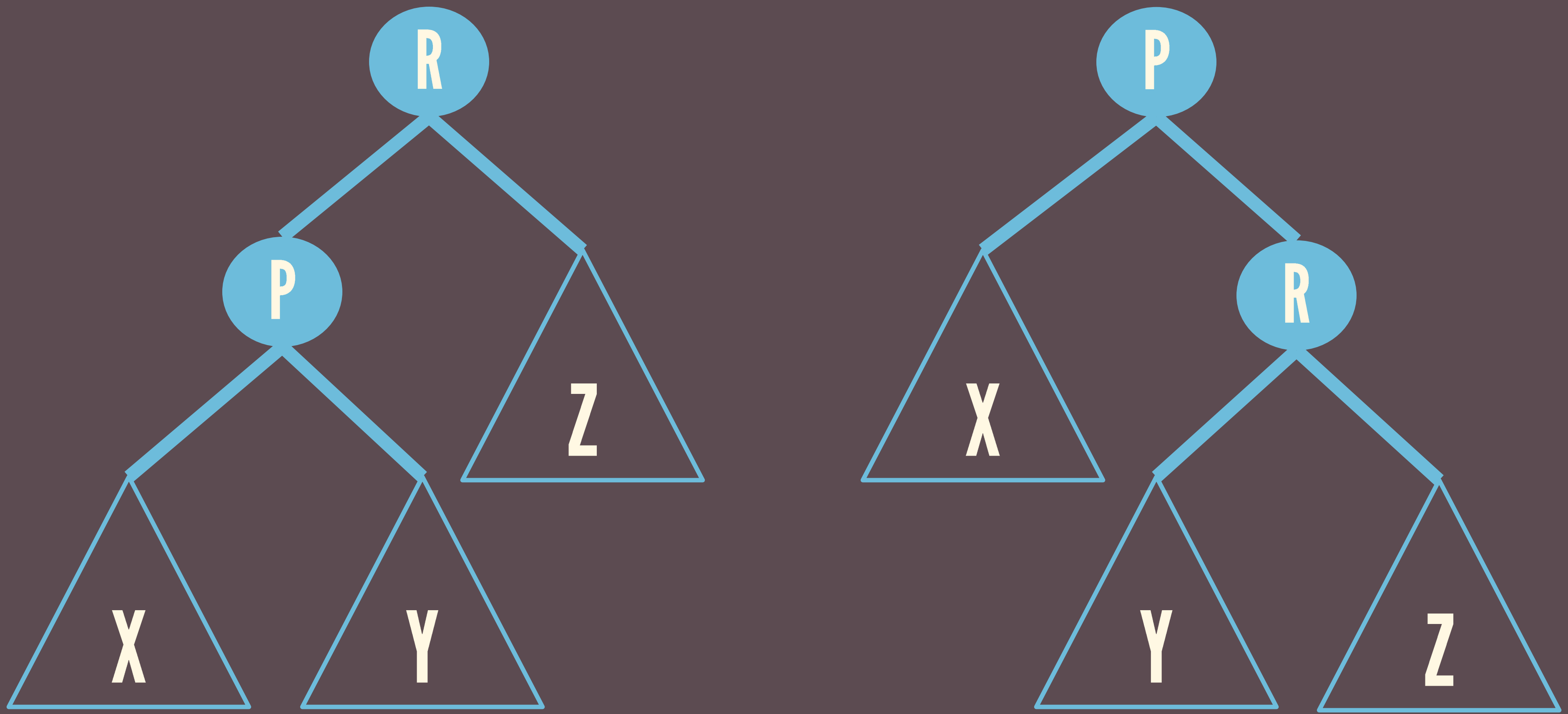


LEFT ROTATE

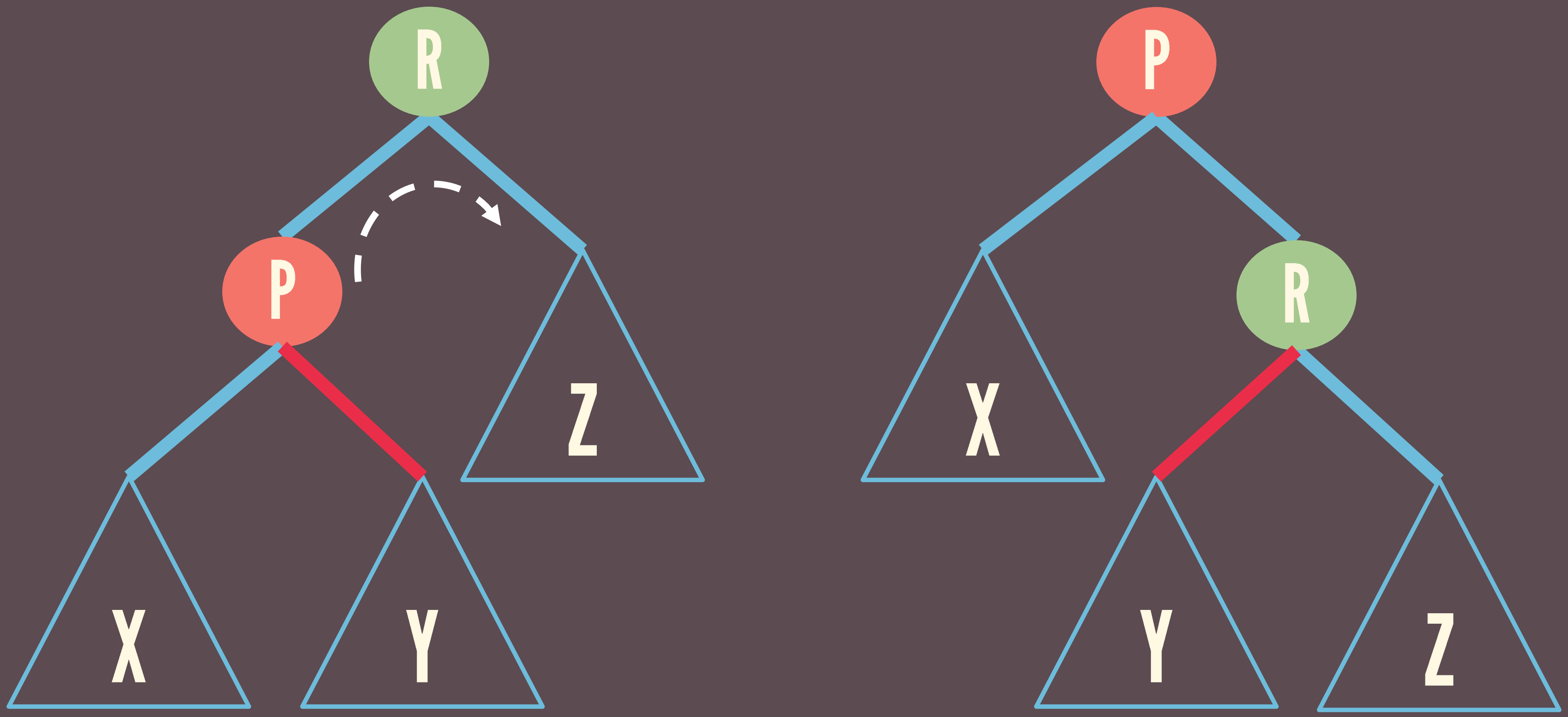
LEFT ROTATE

R becomes the left child of P

Y becomes the right child of R



RIGHT ROTATE

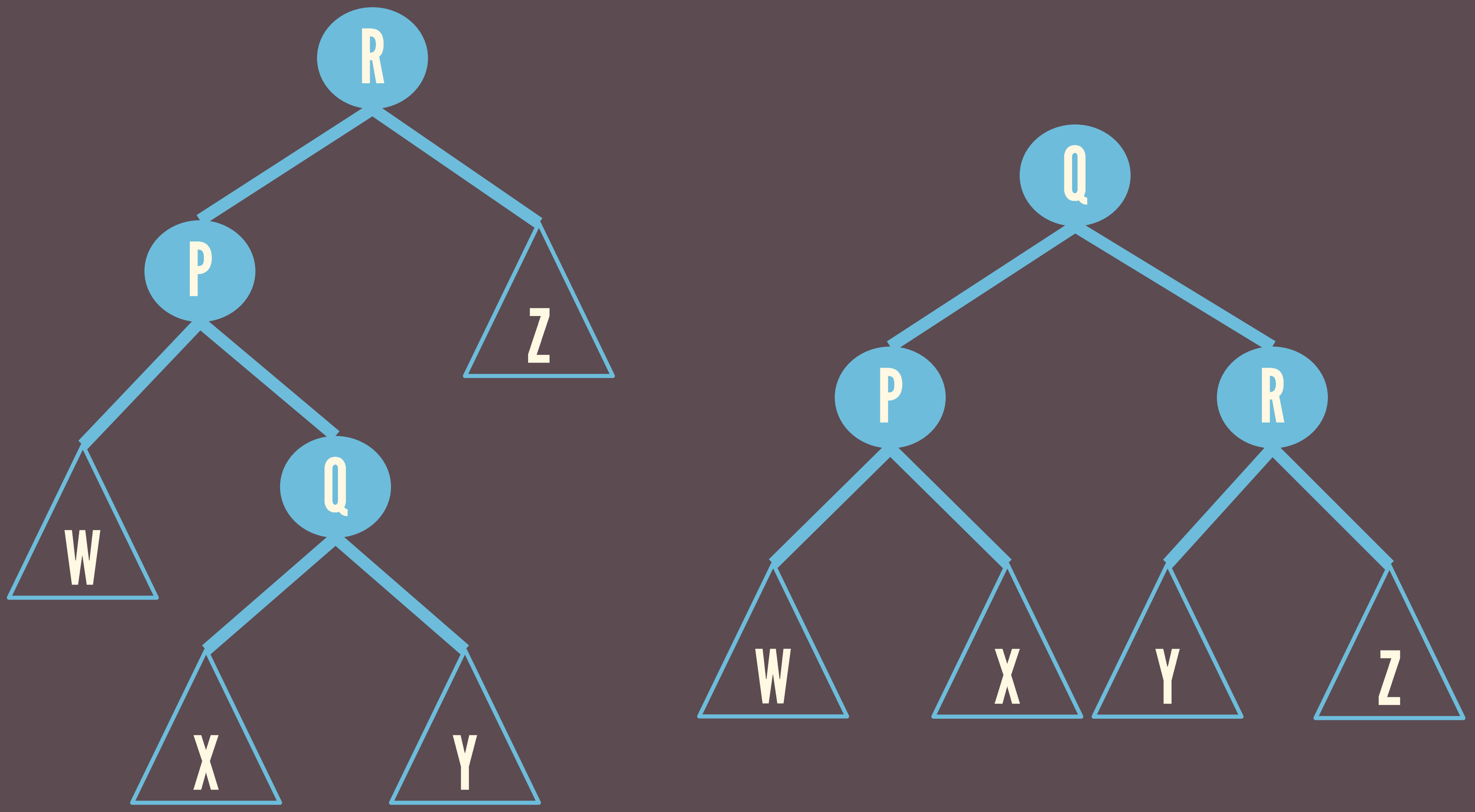


RIGHT ROTATE

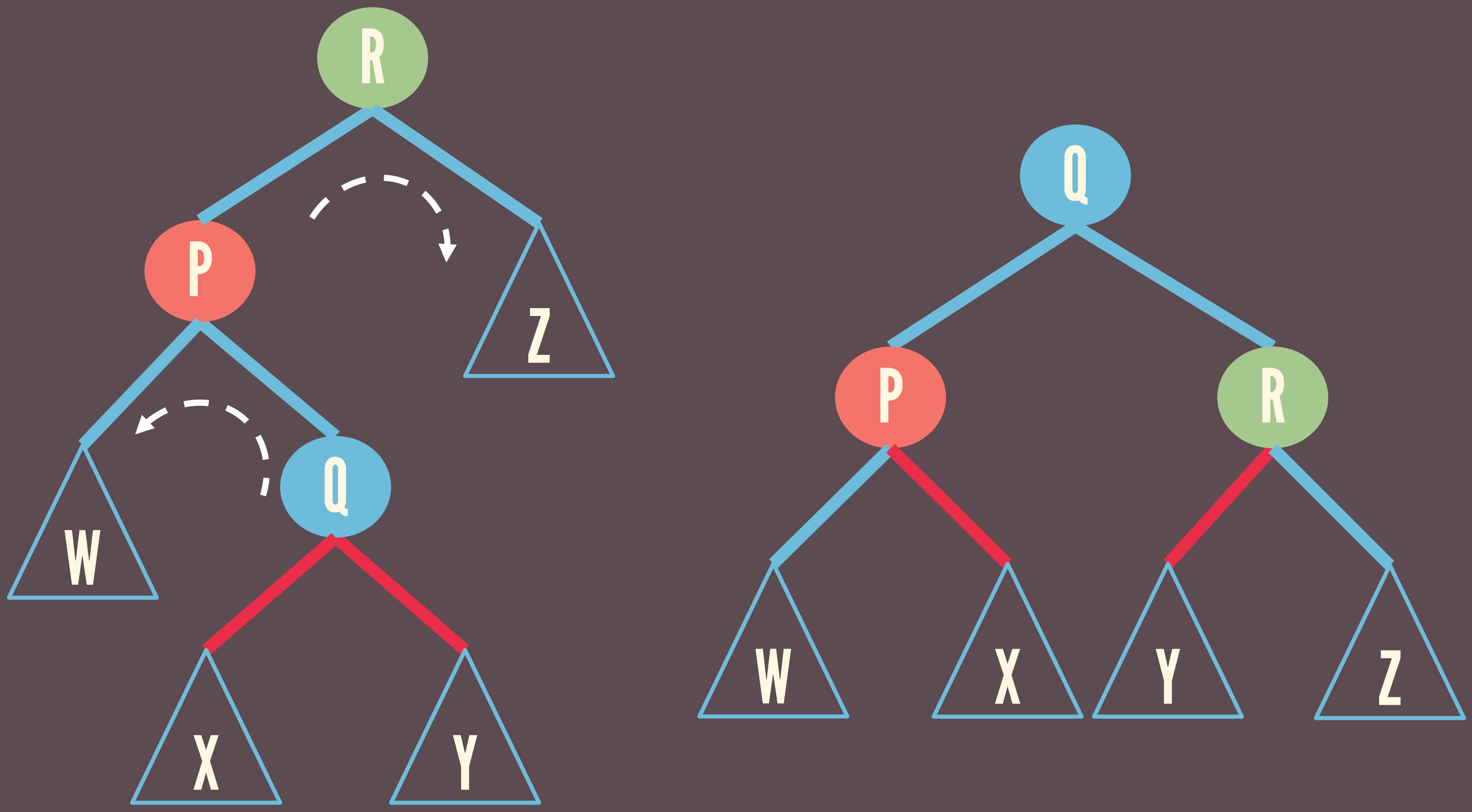
RIGHT ROTATE

R becomes the right child of P

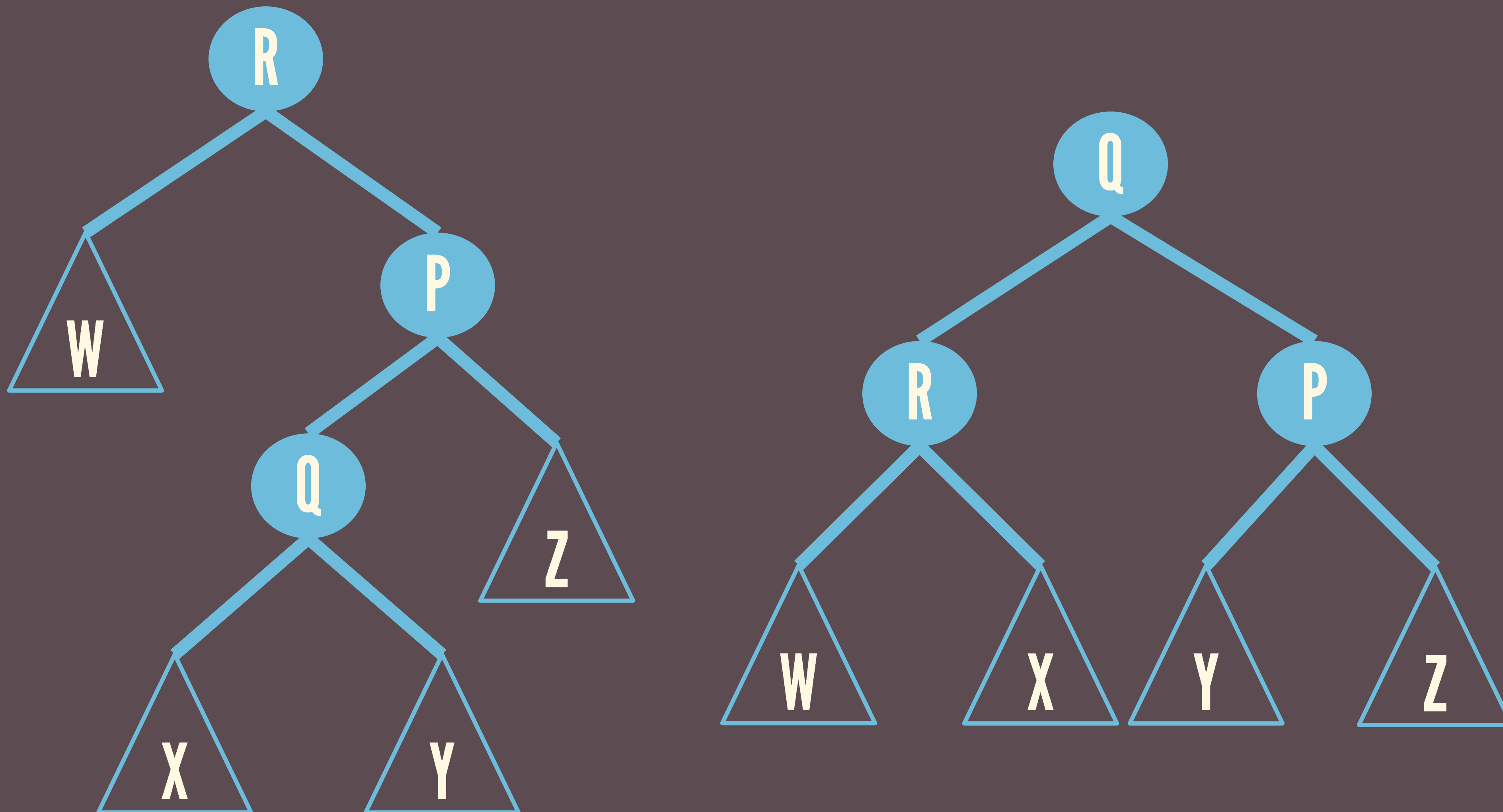
Y becomes the left child of R



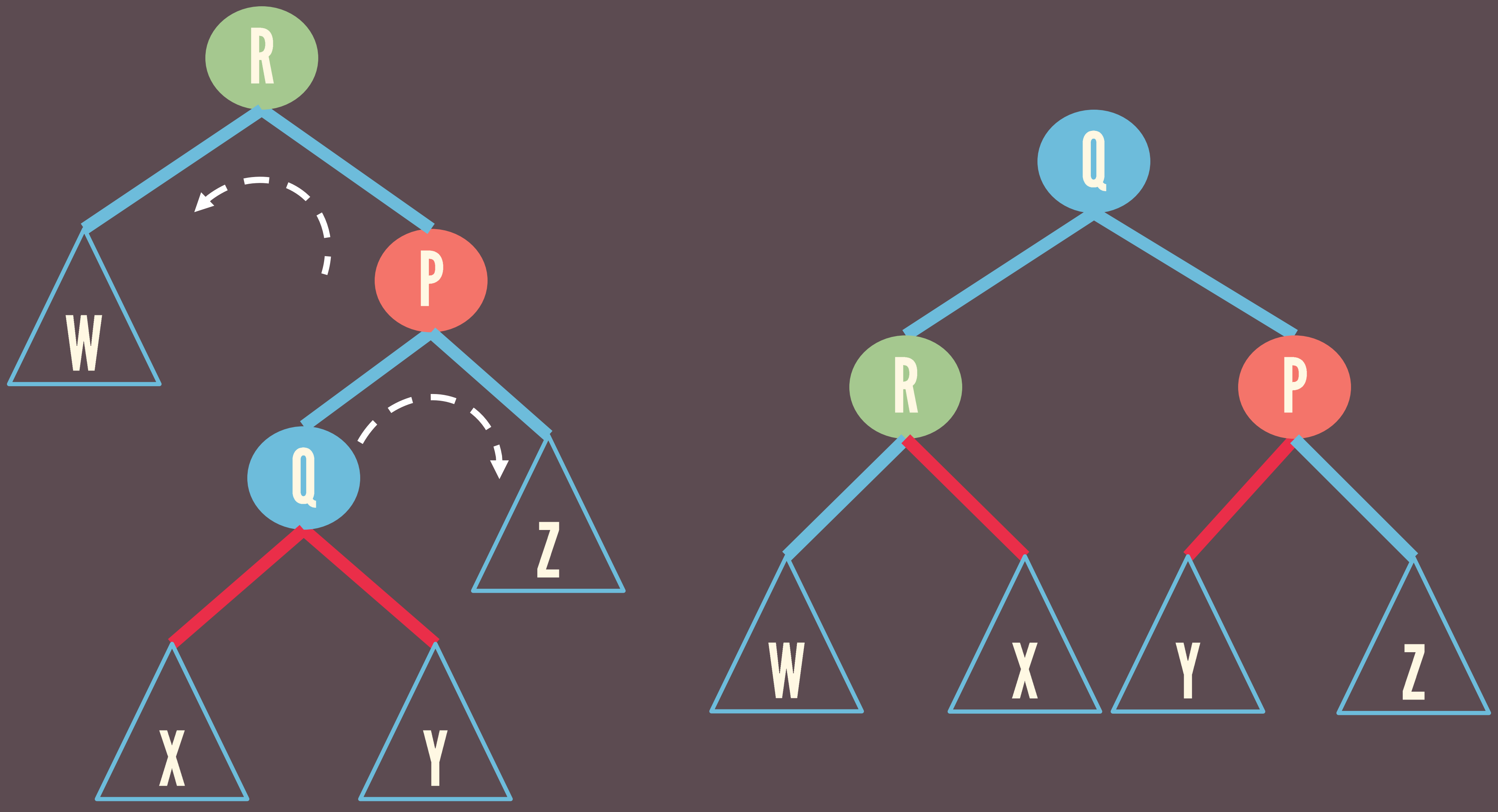
LEFT-RIGHT ROTATE



LEFT-RIGHT ROTATE



RIGHT-LEFT ROTATE

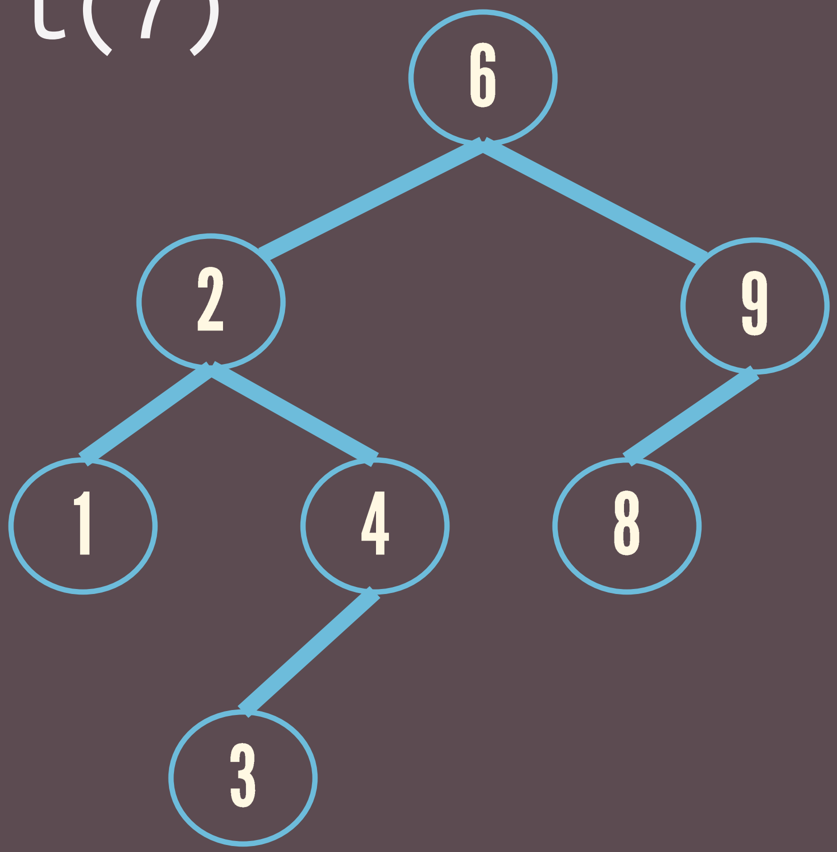


RIGHT-LEFT ROTATE

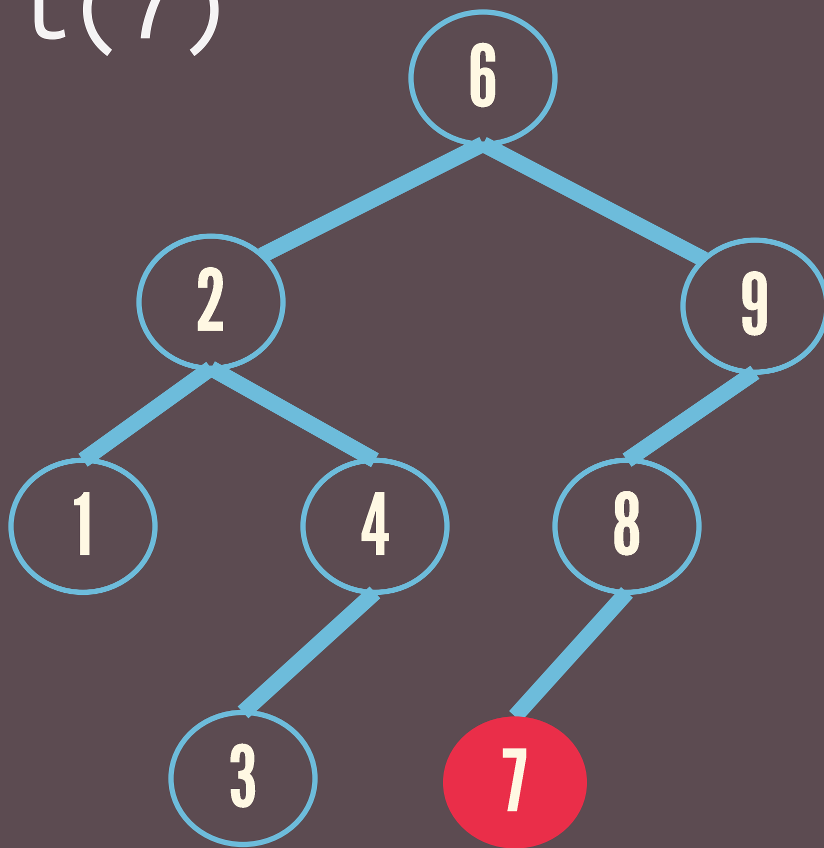
WHEN TO USE WHAT ROTATION

4 CASES

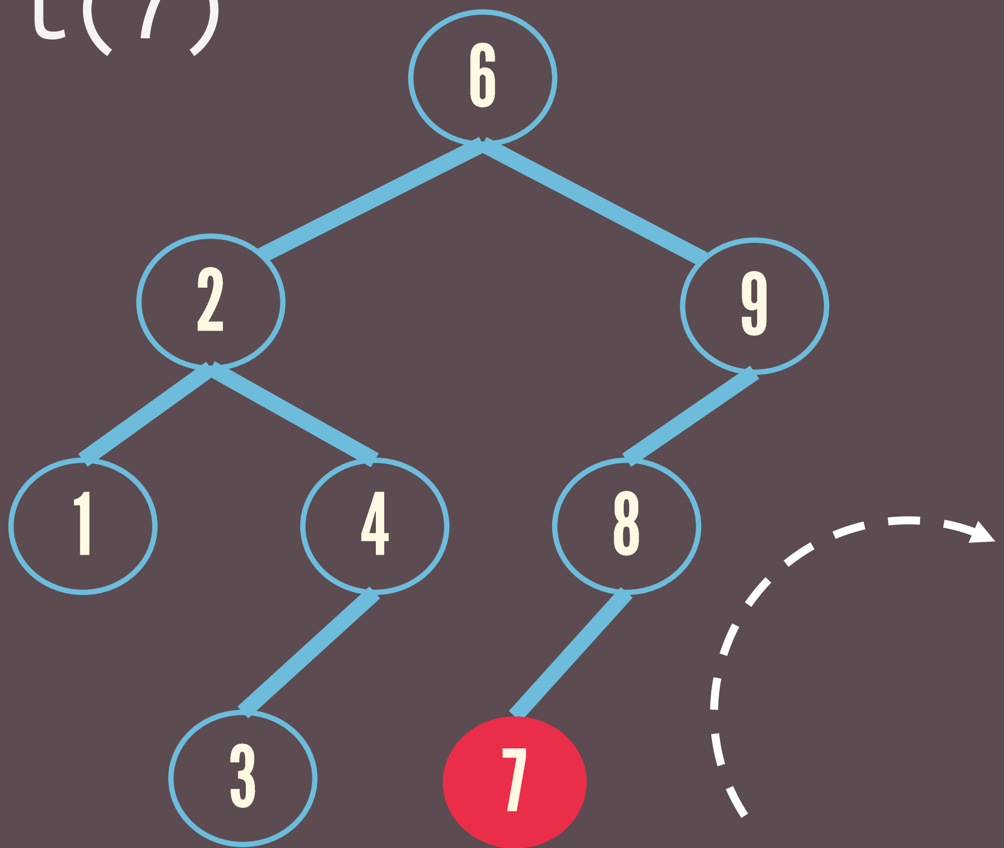
insert(7)



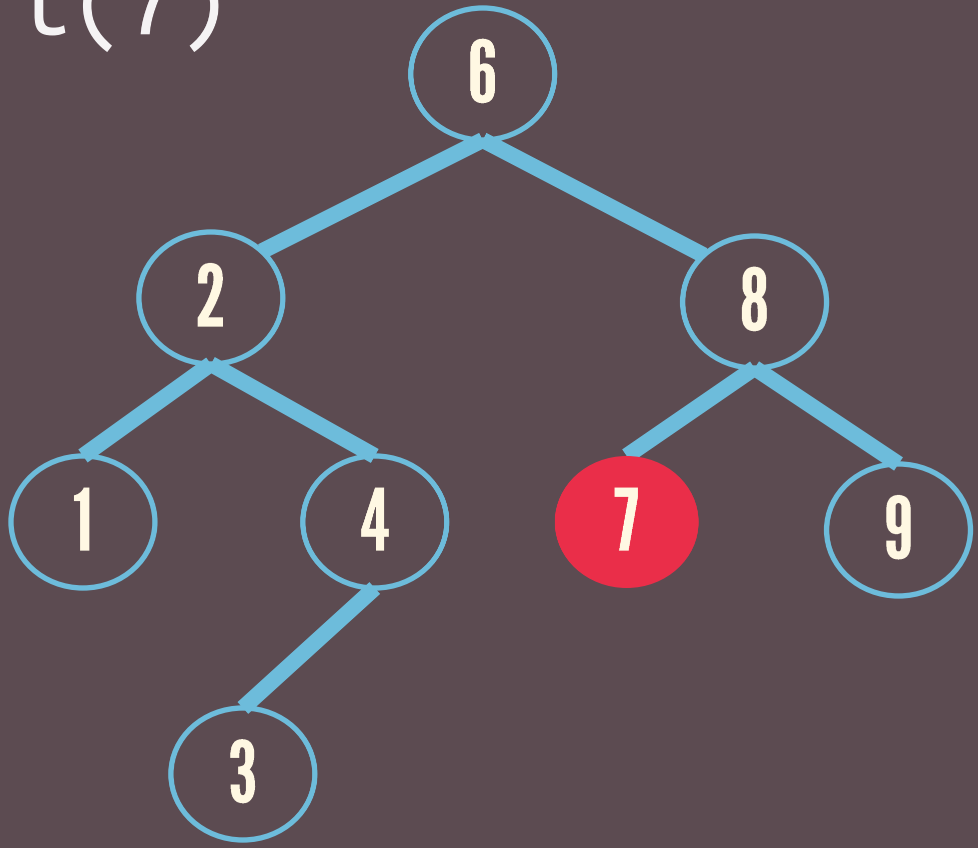
insert(7)



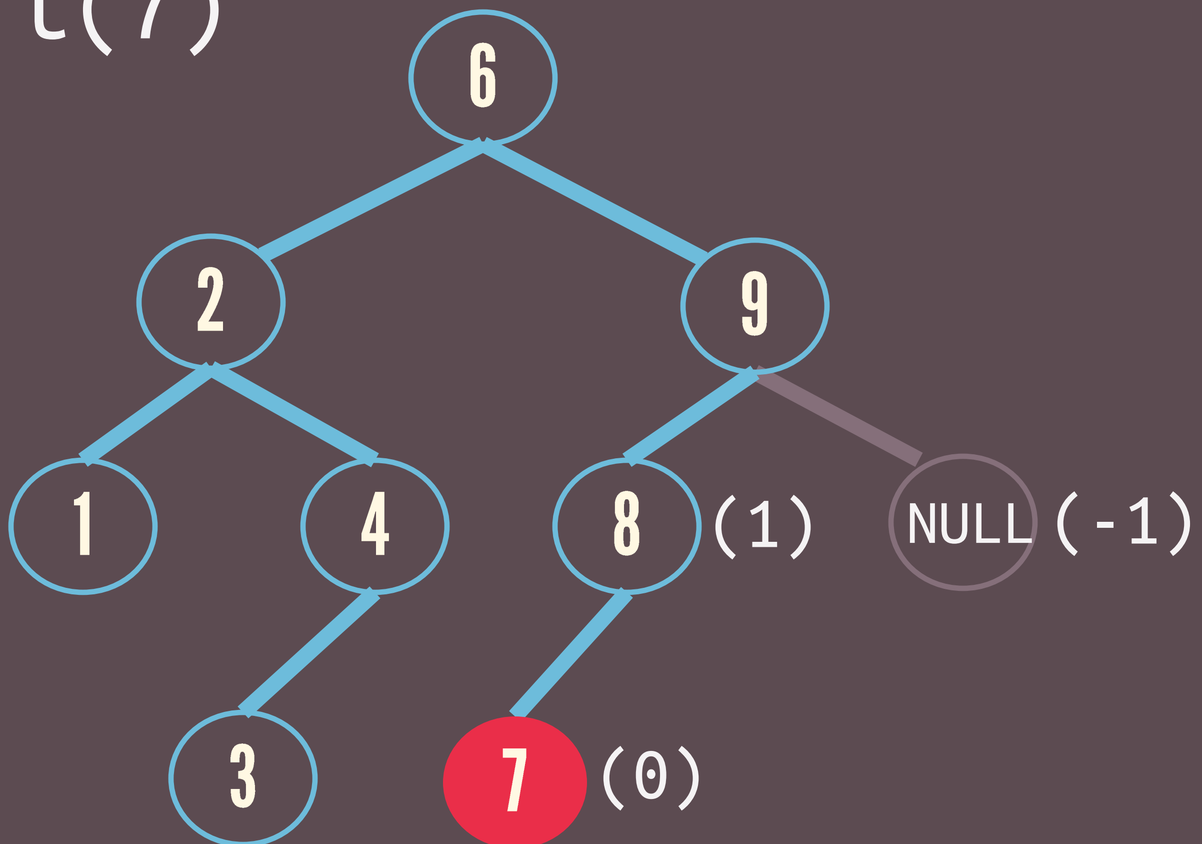
insert(7)



insert(7)



insert(7)



```
insertNode(){

   algorithm for inserting a node.
   update height of nodes.
   fixUp()

}
```

```
fixUp(){

   start at the node inserted and travel
   up the tree:
     if an imbalance is found,
        check the four cases and do the
        appropriate rotation.
     update height of the nodes.

}
```

fixUp()

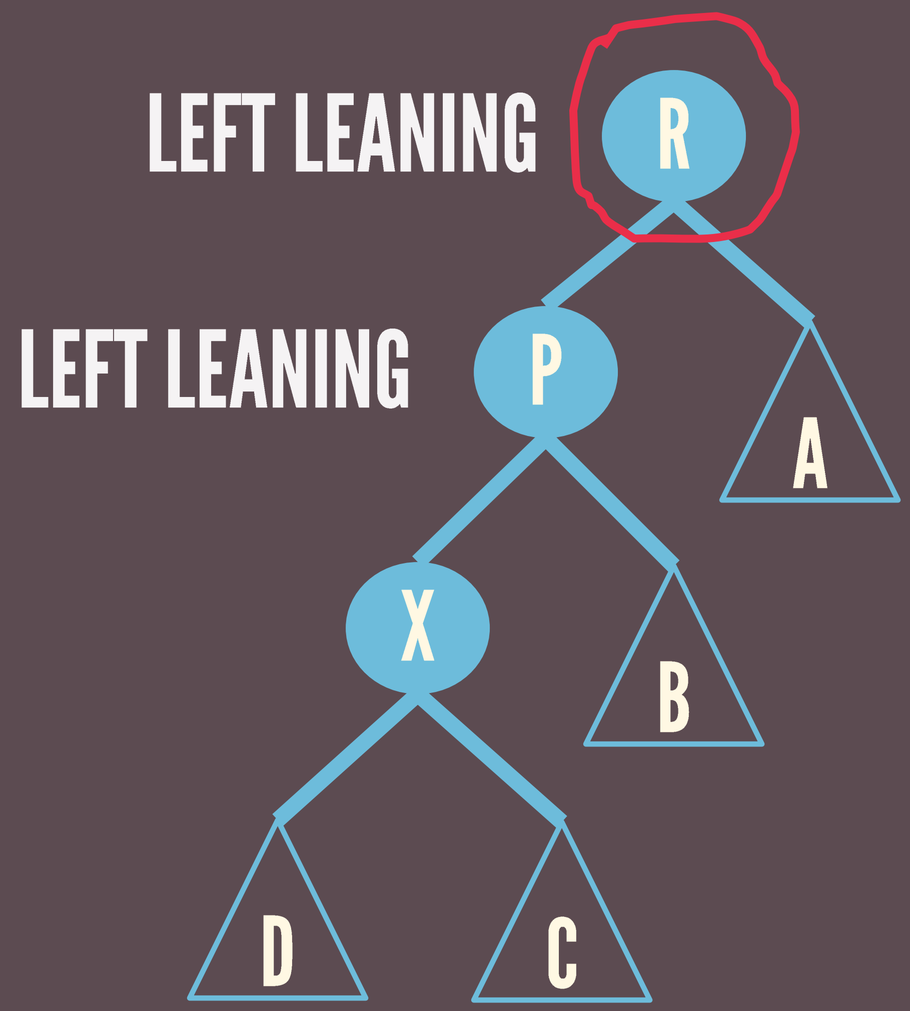
rotation is made where the imbalance is found.

#1

LEFT LEFT CASE

RIGHT ROTATE

#1



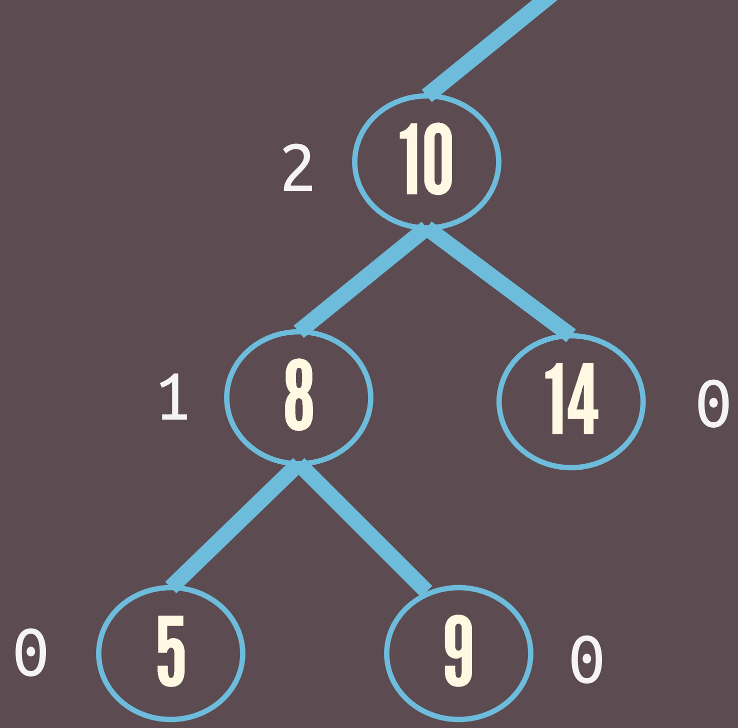
R – root

P – pivot

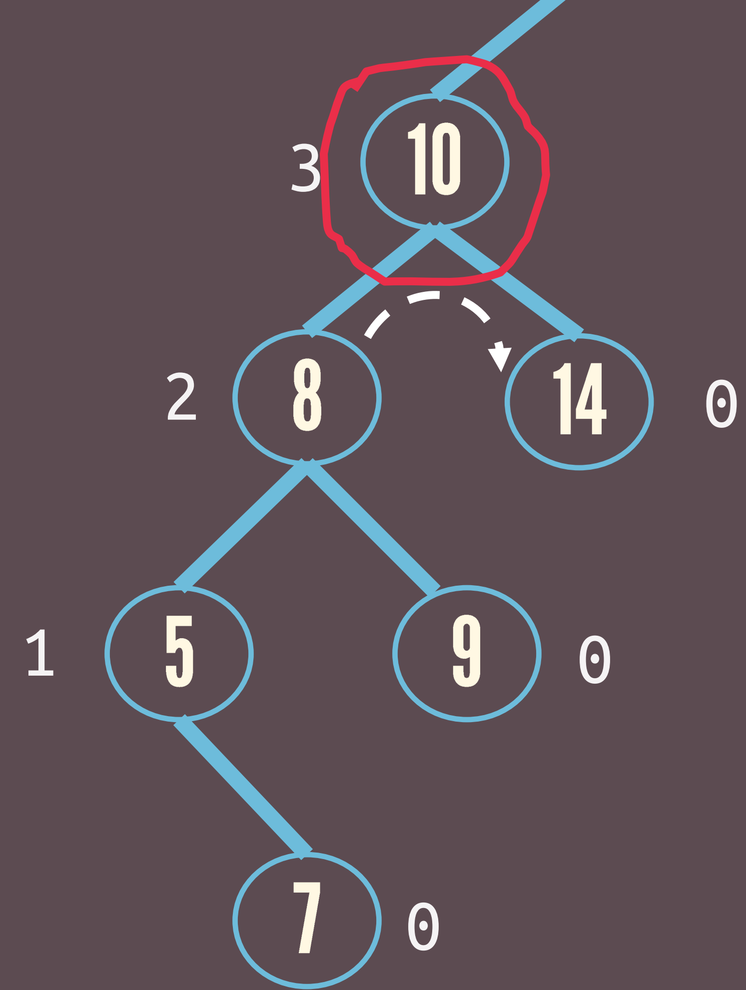
#1

```
fixUp(){

   start at the node inserted and travel
   up the tree:
      if an imbalance is found,
         if pivot is left leaning and
         root is left leaning
           do a right rotation on root.
      update height of the nodes.

}
```

#1

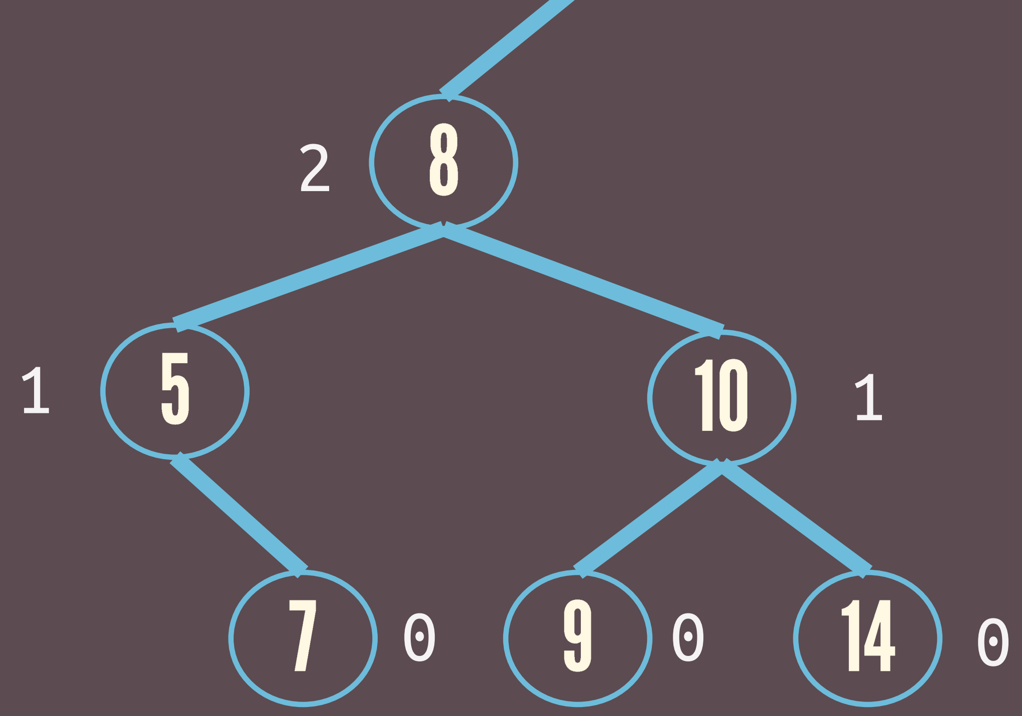



```
insert(7)
```




```
insert(7)
```

#1



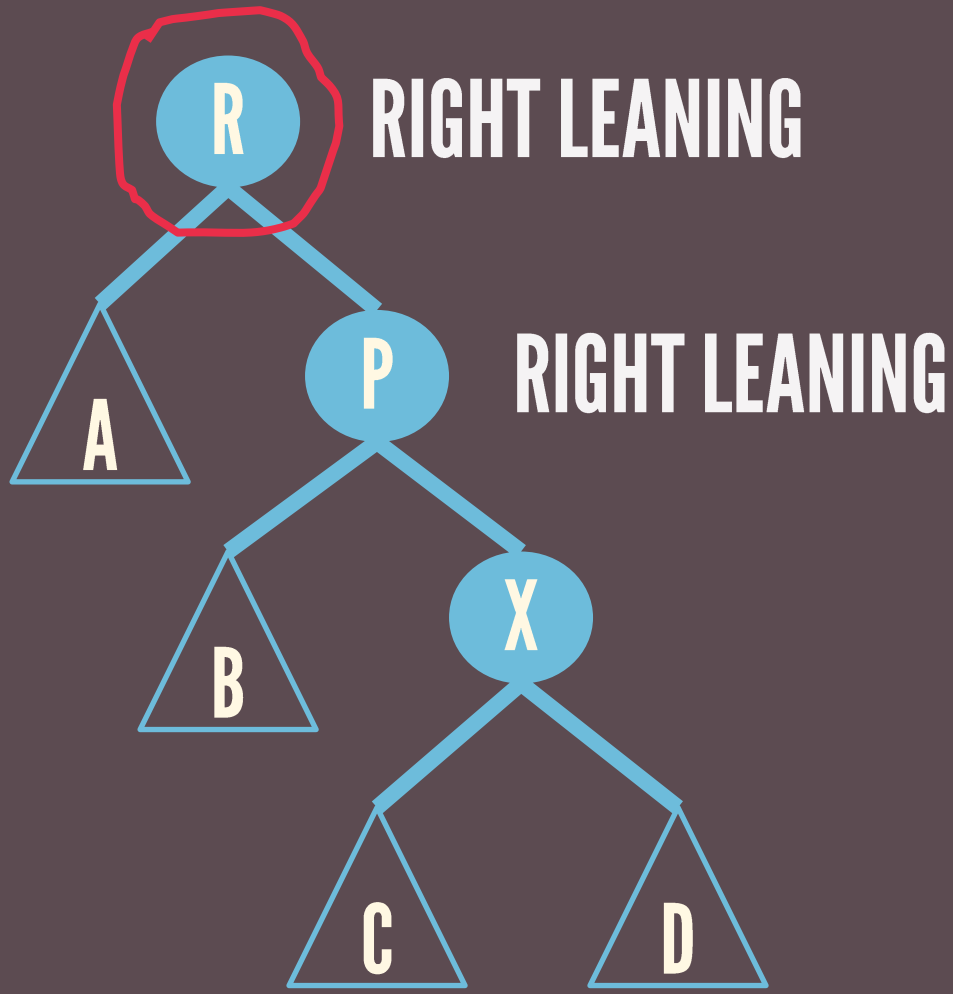

```
insert(7)
```

#2

RIGHT RIGHT CASE

LEFT ROTATE

#2



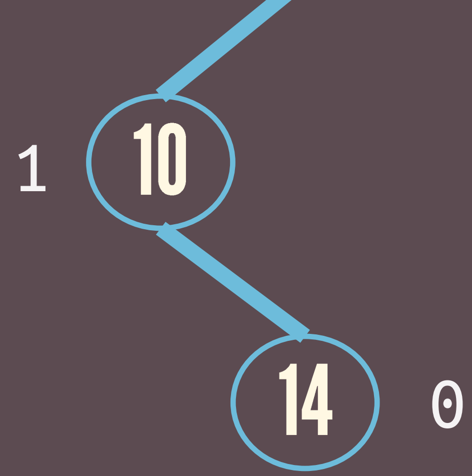
R – root

P – pivot

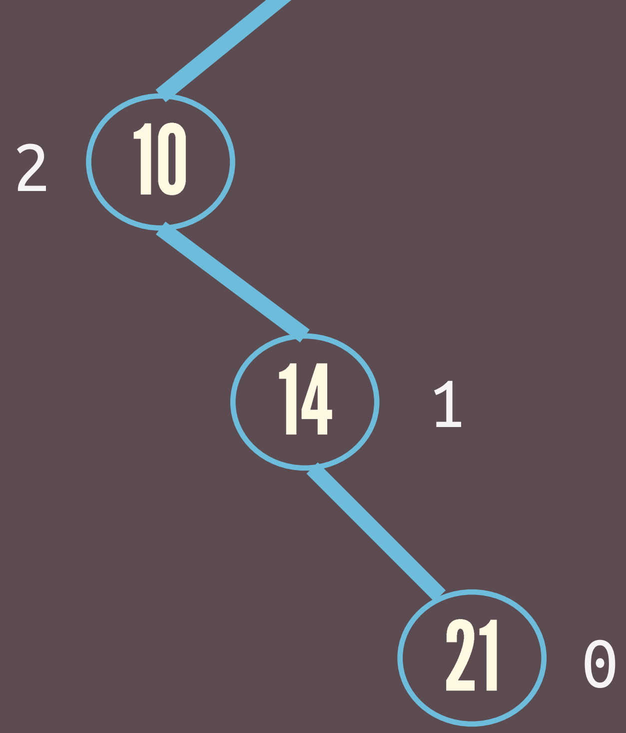
#2

```
fixUp(){

   start at the node inserted and travel
   up the tree:
      if an imbalance is found,
         if pivot is right leaning and
         root is right leaning
            do a left rotation on root.
      update height of the nodes.

}
```

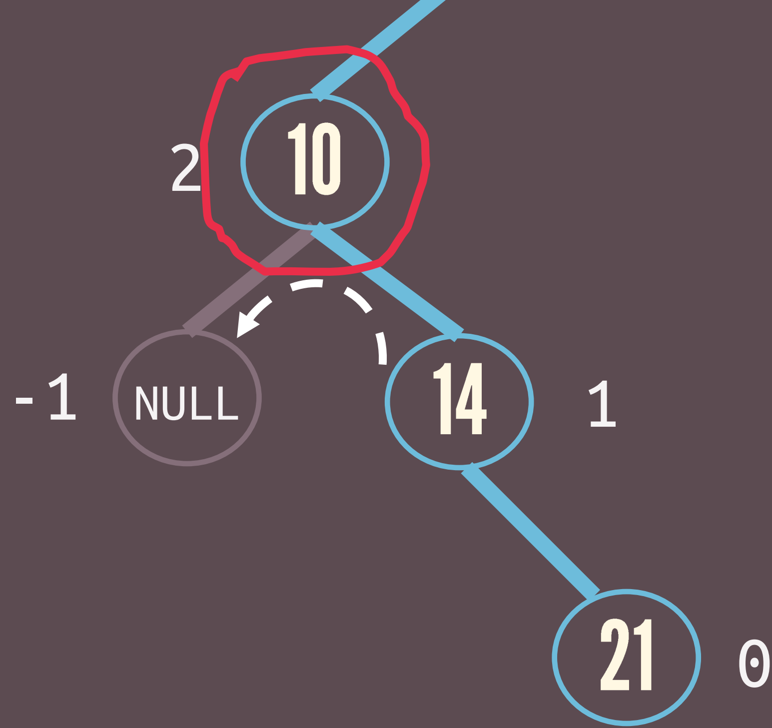
#2



insert(21)

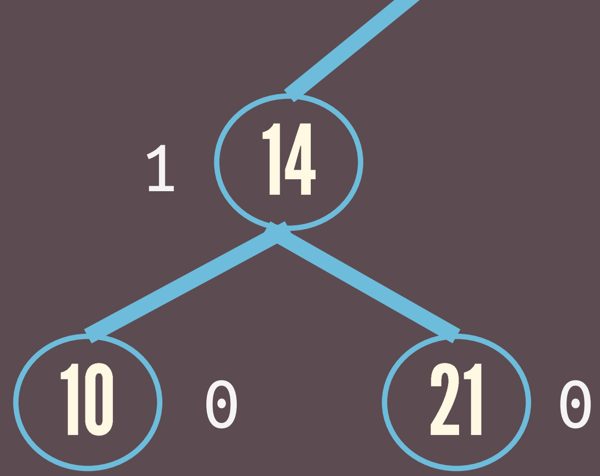


insert(21)



insert(21)

#2

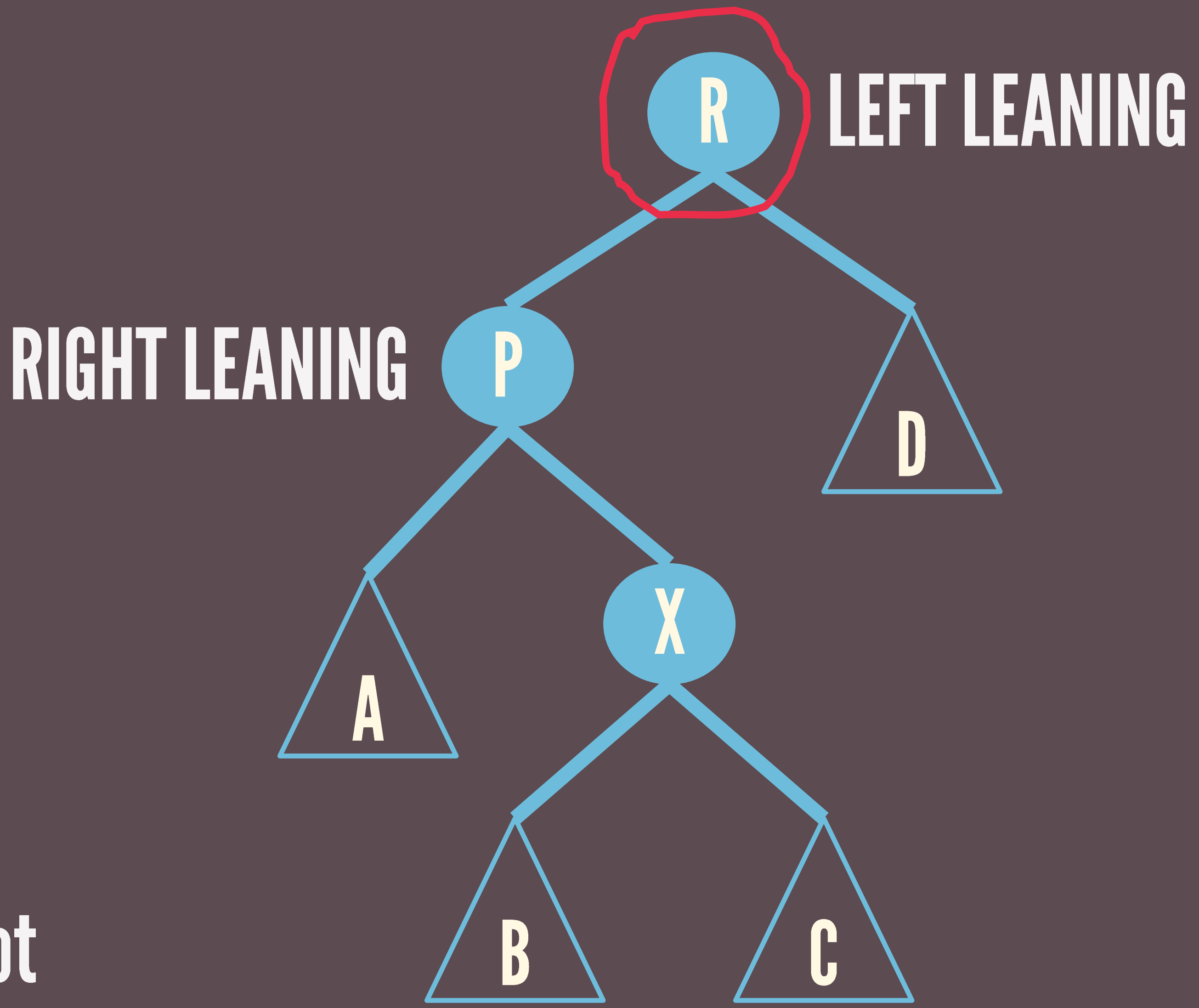


insert(21)

LEFT RIGHT CASE

LEFT RIGHT ROTATE

#3



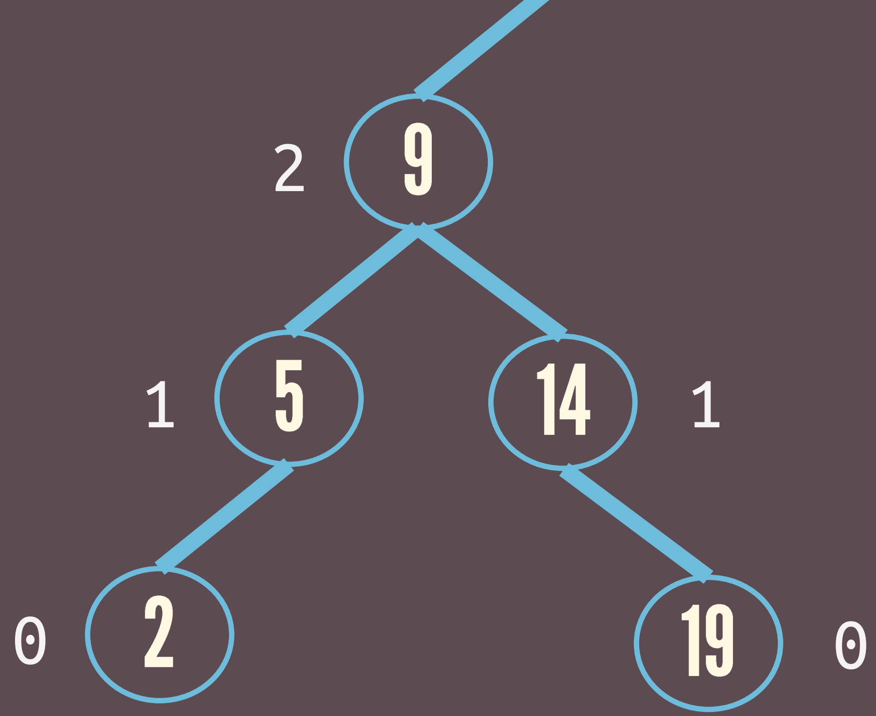
R – root

P – pivot

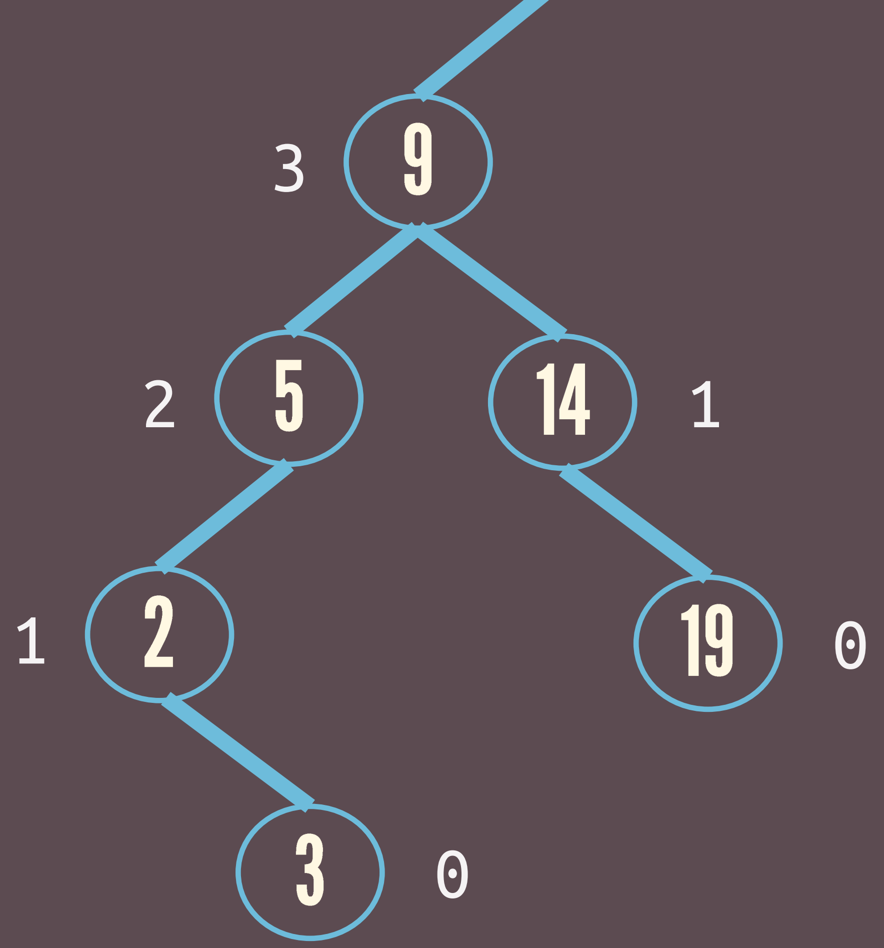
#3

```
fixUp(){

   start at the node inserted and travel
   up the tree:
      if an imbalance is found,
         if pivot is right leaning and
         root is left leaning
            do a left rotation on pivot.
            do a right rotation on root.
      update height of the nodes.

}
```

#3



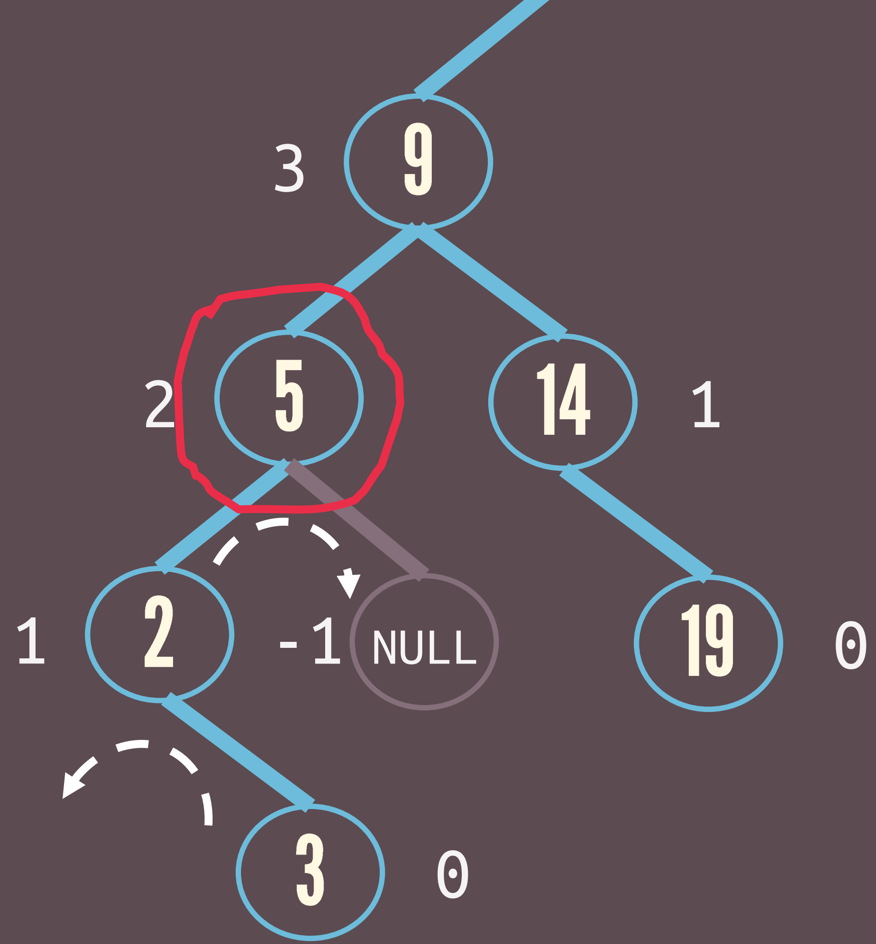
insert(3)



#3

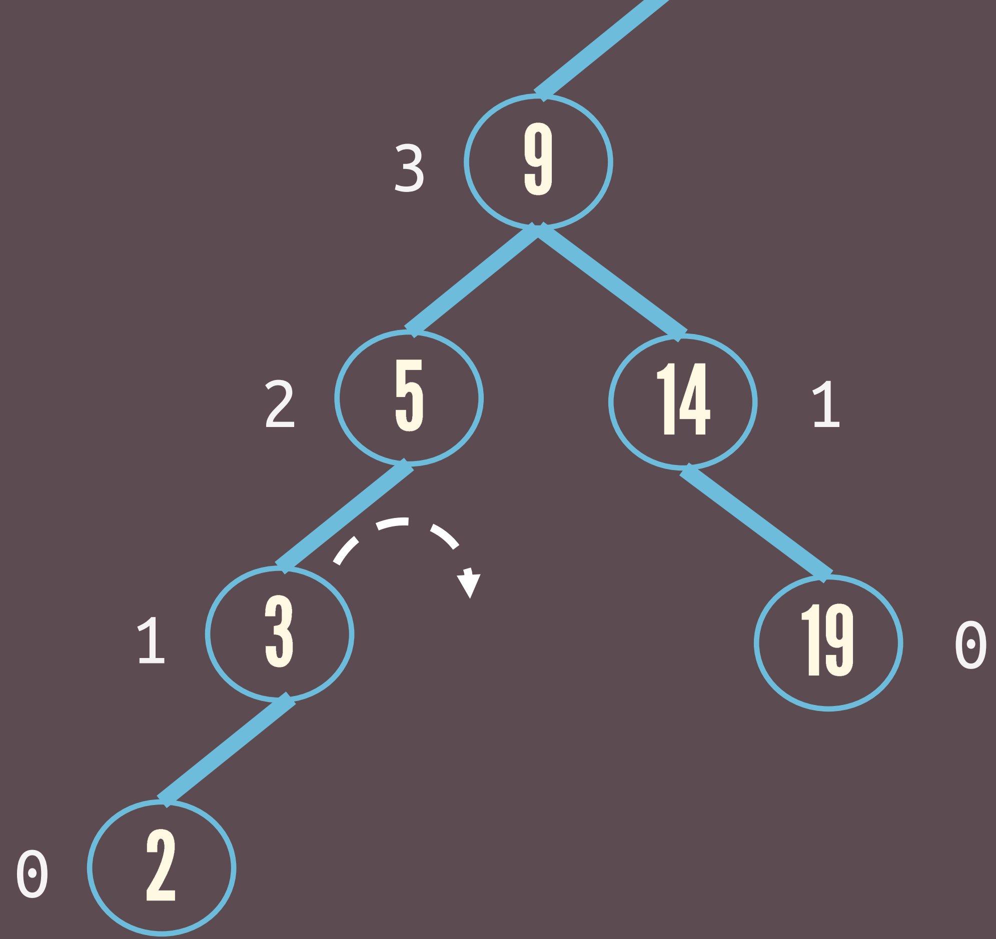
insert(3)

#3



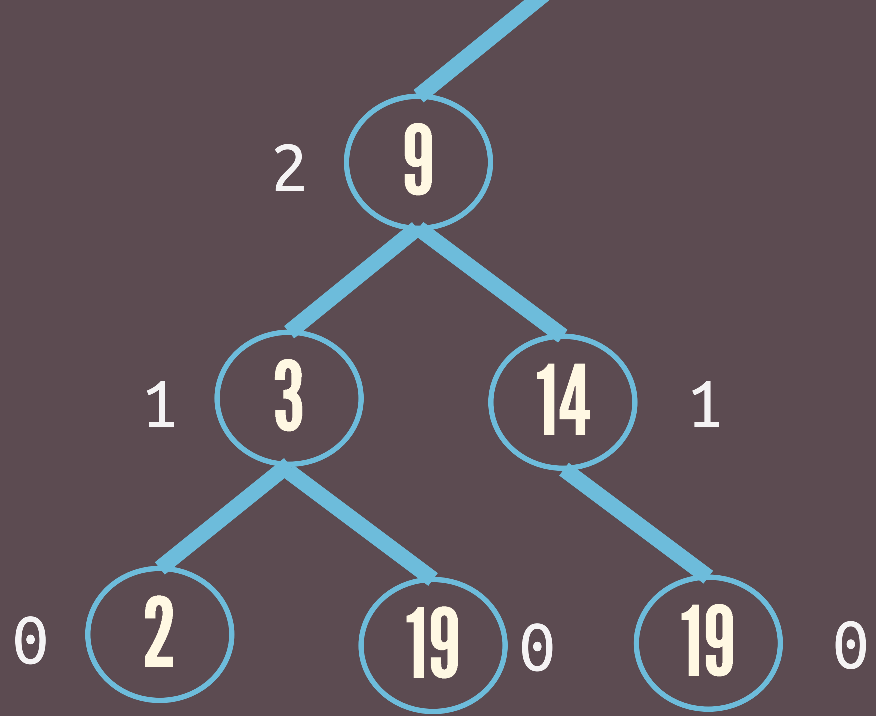
insert(3)

#3



insert(3)

#3

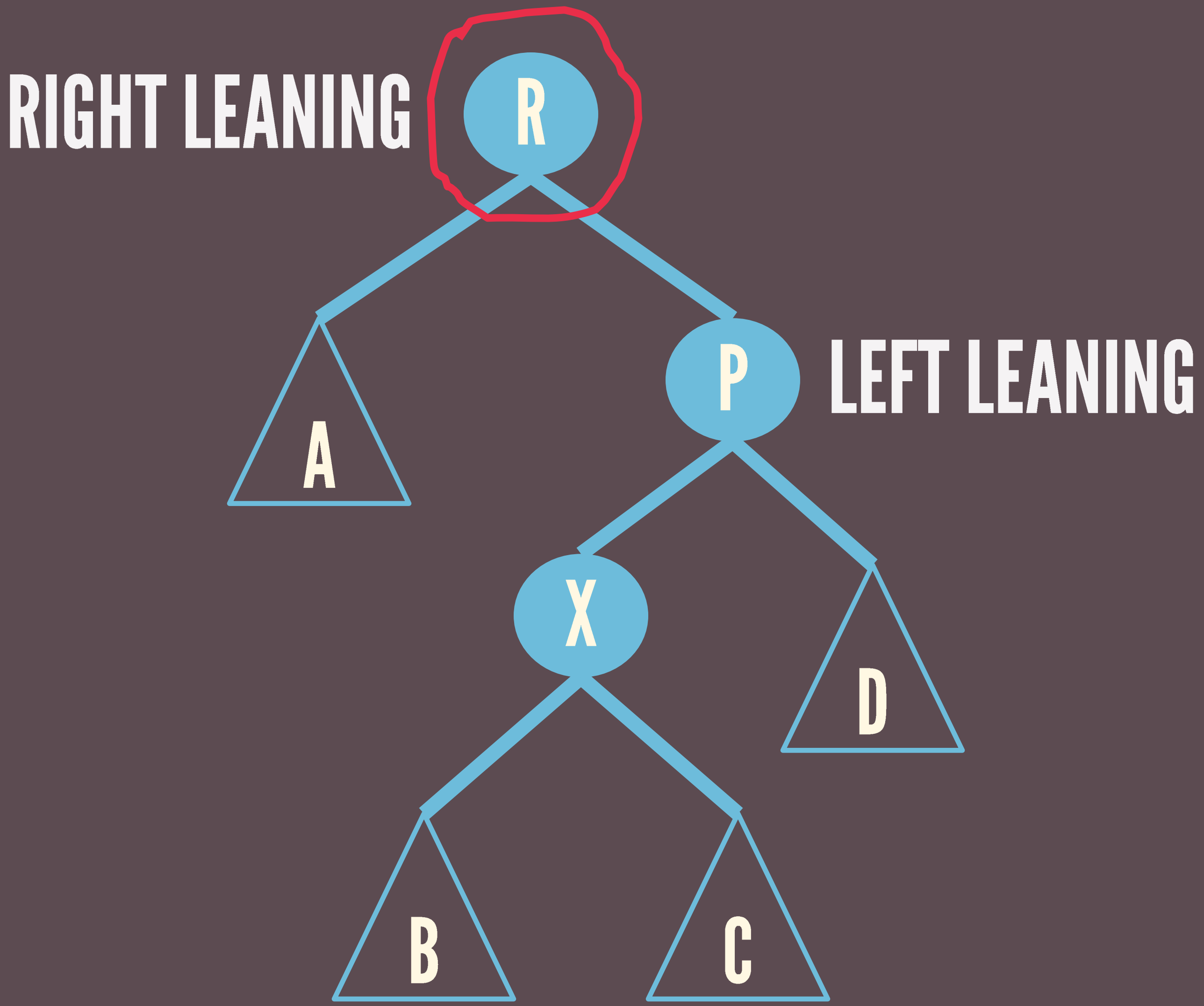


insert(3)

RIGHT LEFT CASE

RIGHT LEFT ROTATE

#4

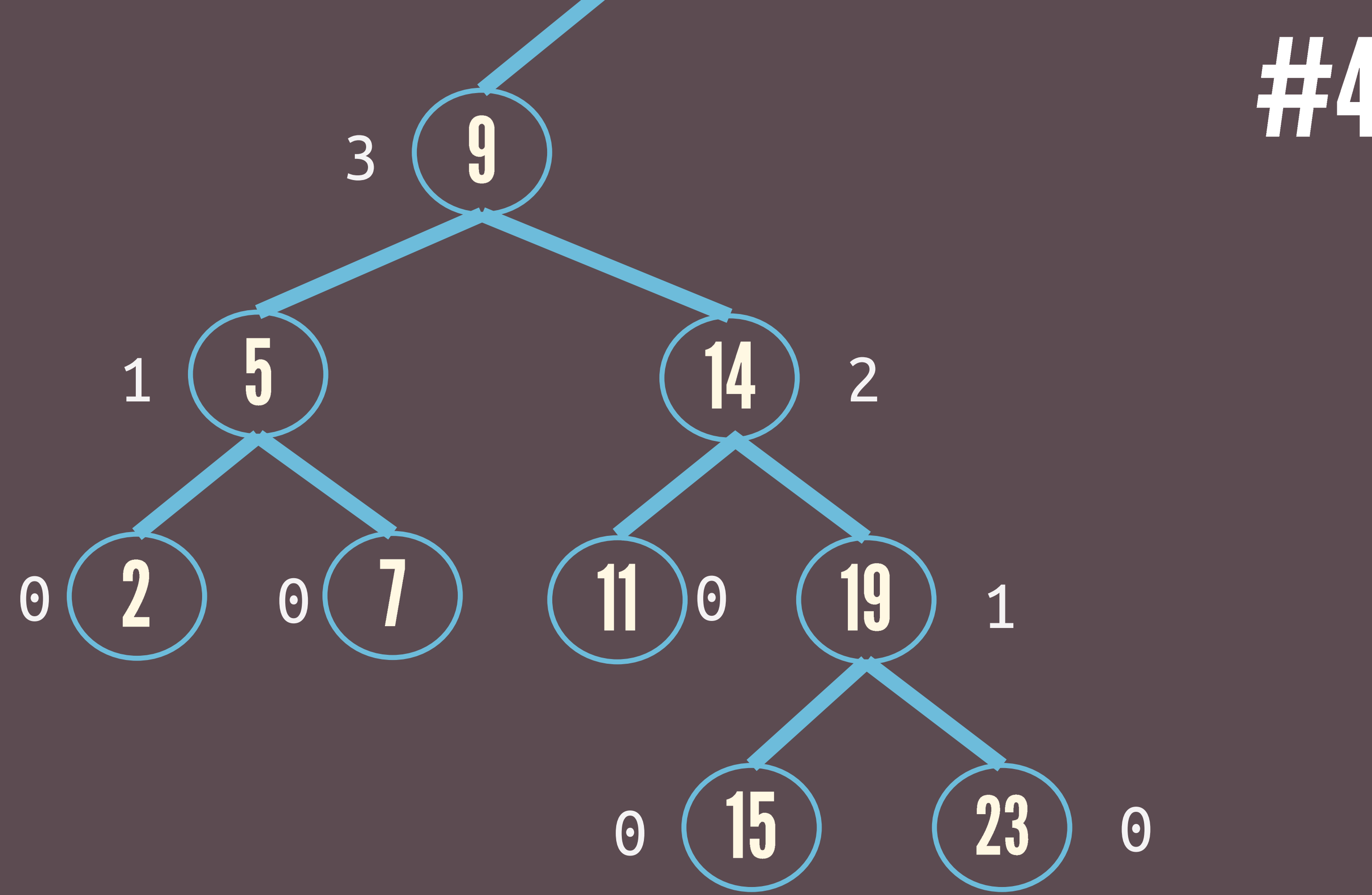


R – root

P – pivot

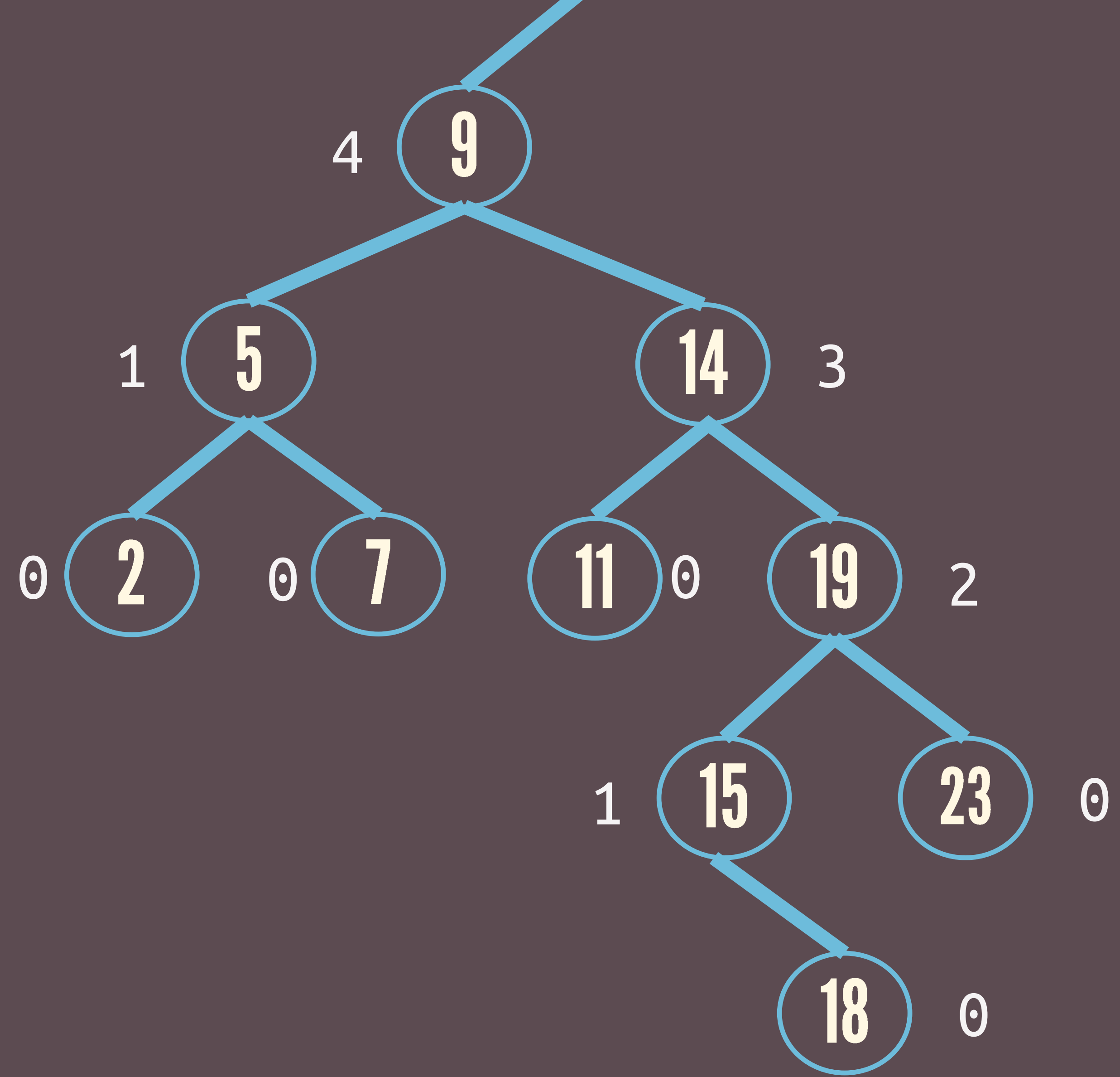
#4

```
fixUp(){

  start at the node inserted and travel
  up the tree:
    if an imbalance is found,
      if pivot is left leaning and
      root is right leaning
        do a right rotation on pivot.
        do a left rotation on root.
    update height of the nodes.

}
```



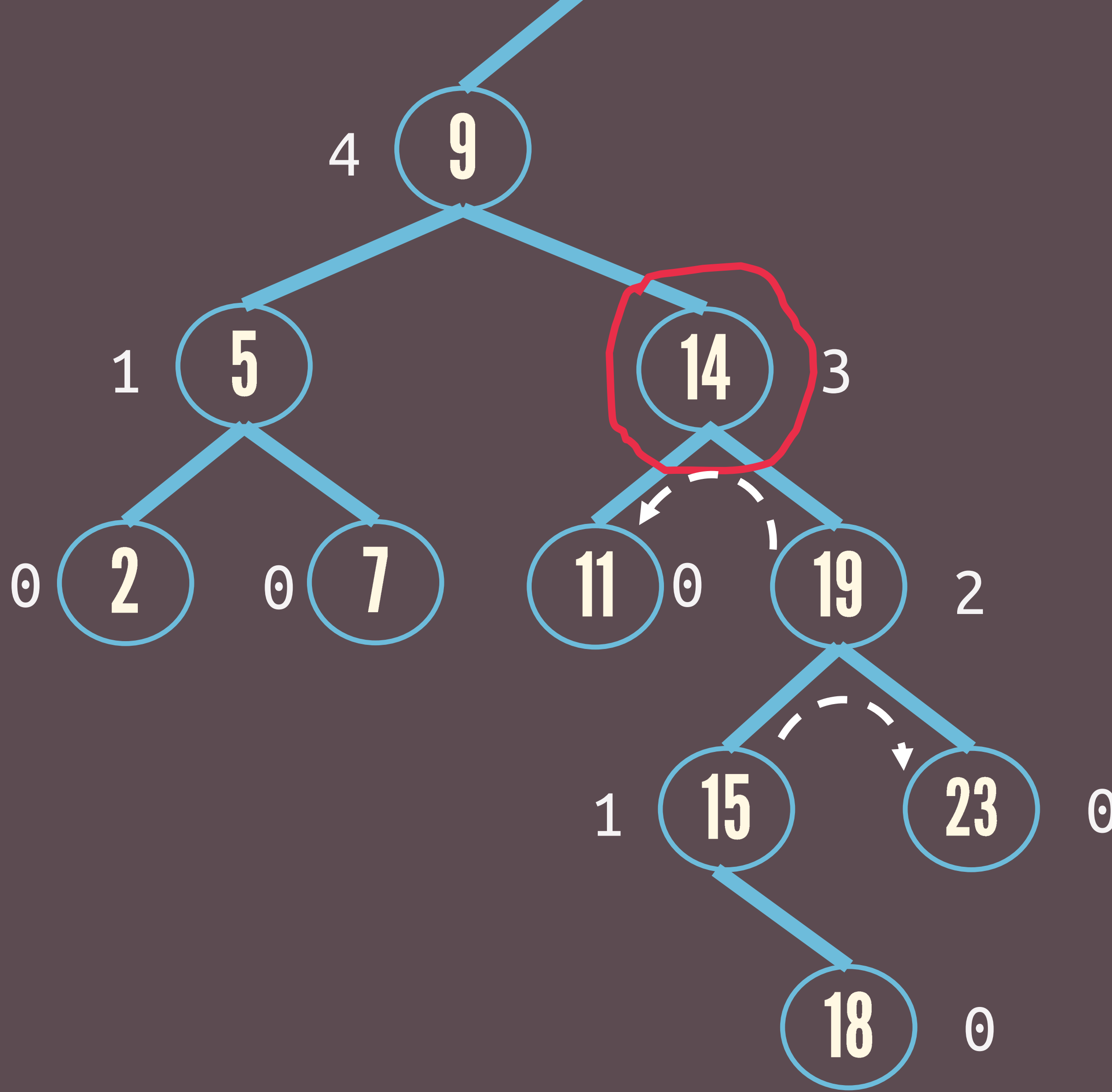
insert(18)

#4



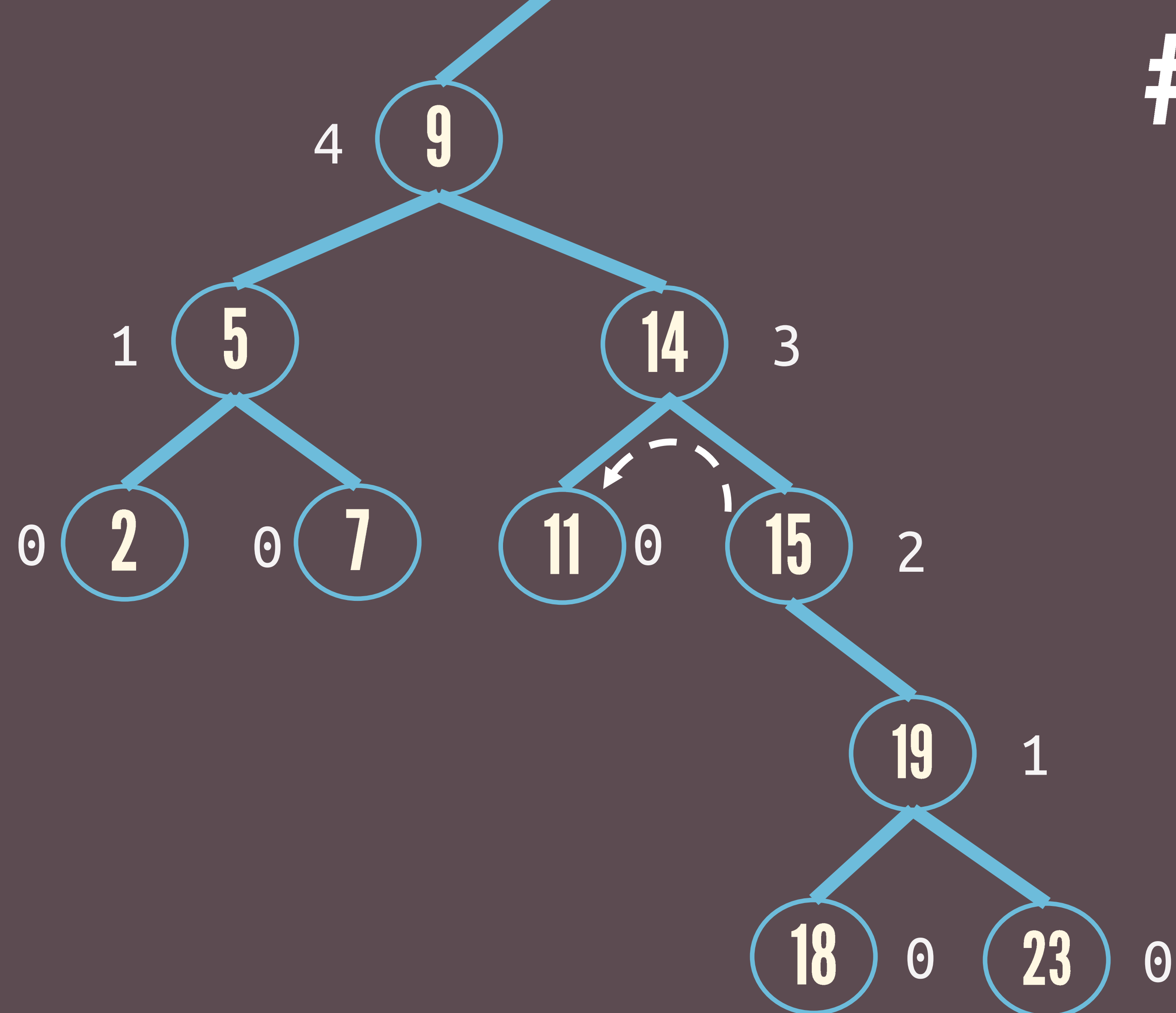
`insert(18)`

#4



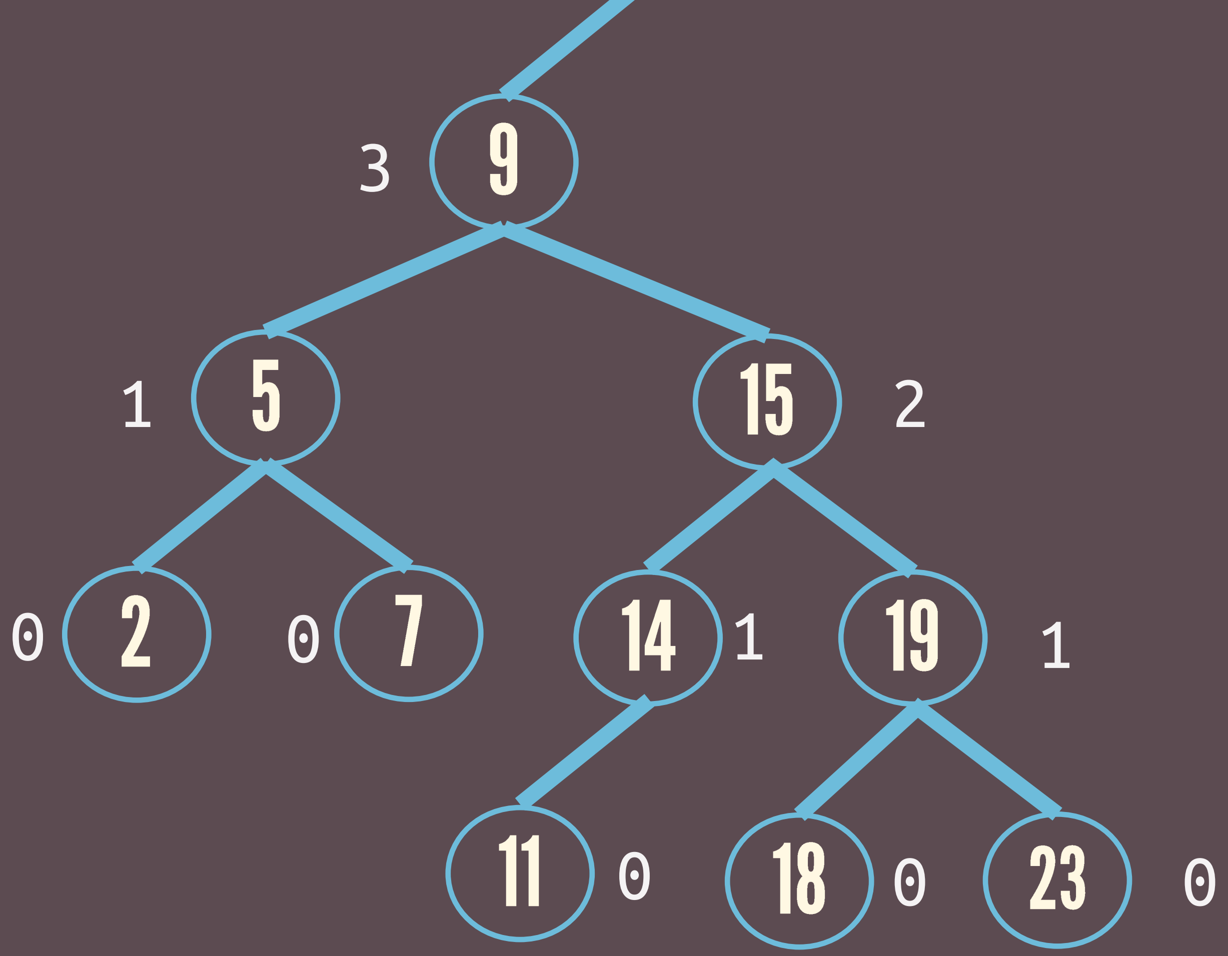
insert(18)

#4



insert(18)

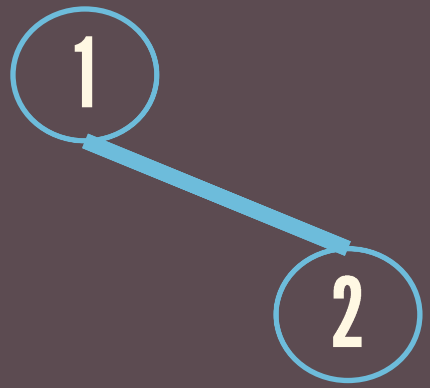
#4

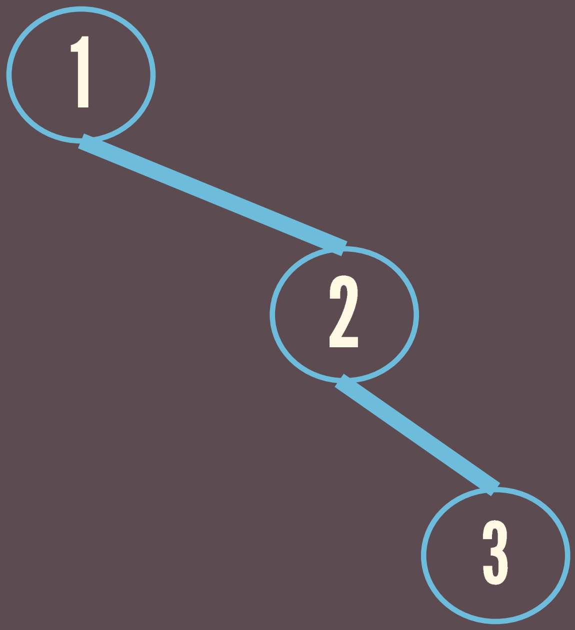


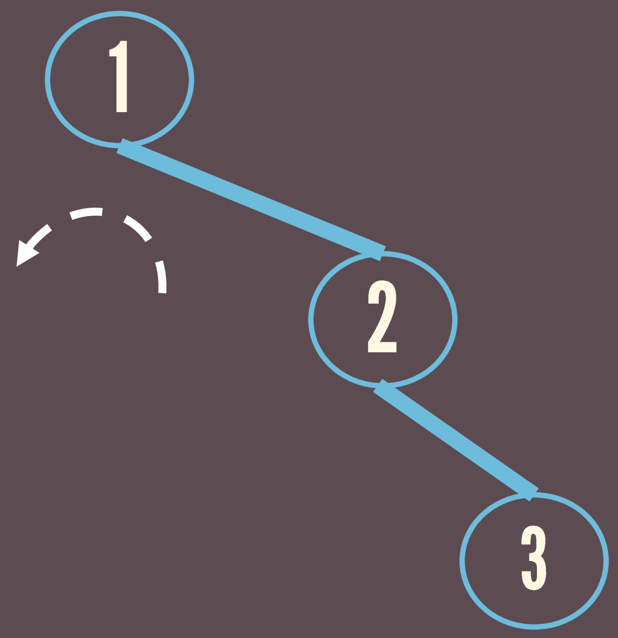
insert(18)

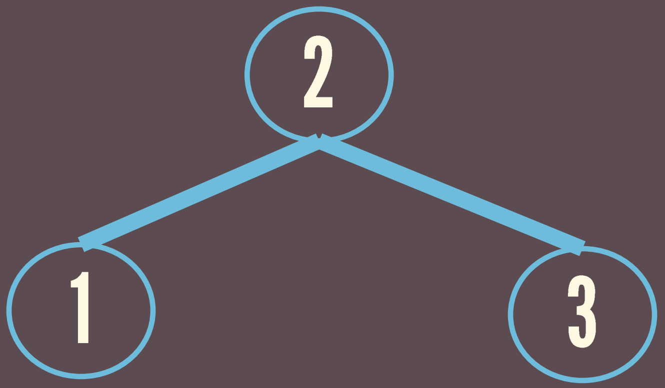
```
 insert(1)
 insert(2)
 insert(3)
 insert(4)
 insert(5)
 insert(6)
 insert(7)
insert(15)
insert(14)
insert(13)
```

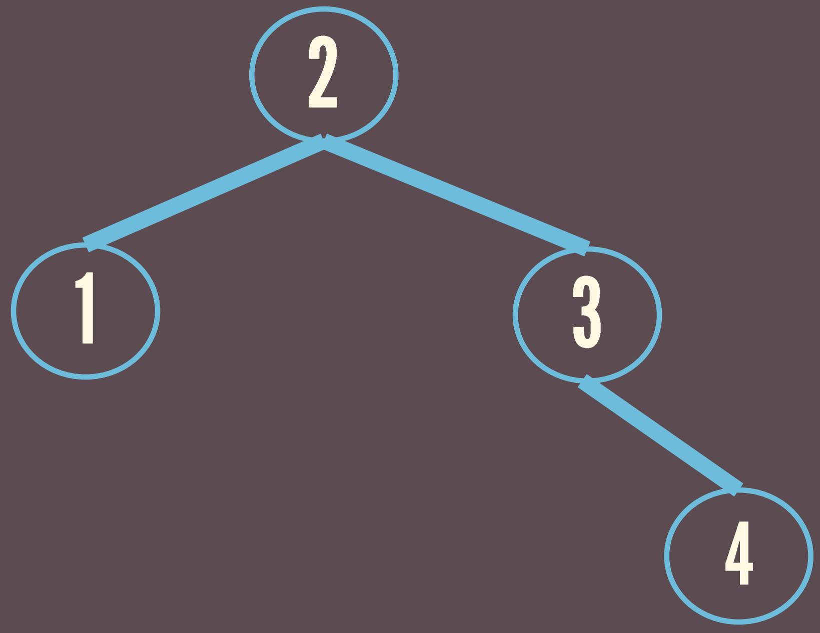
1

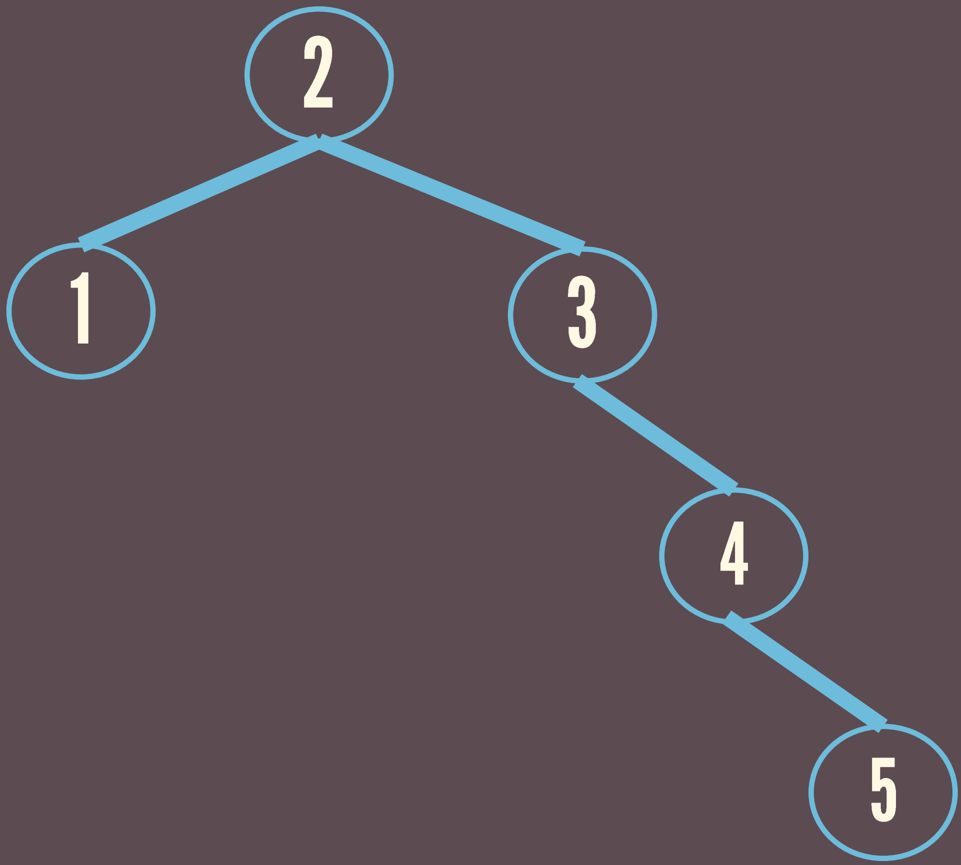

1
2

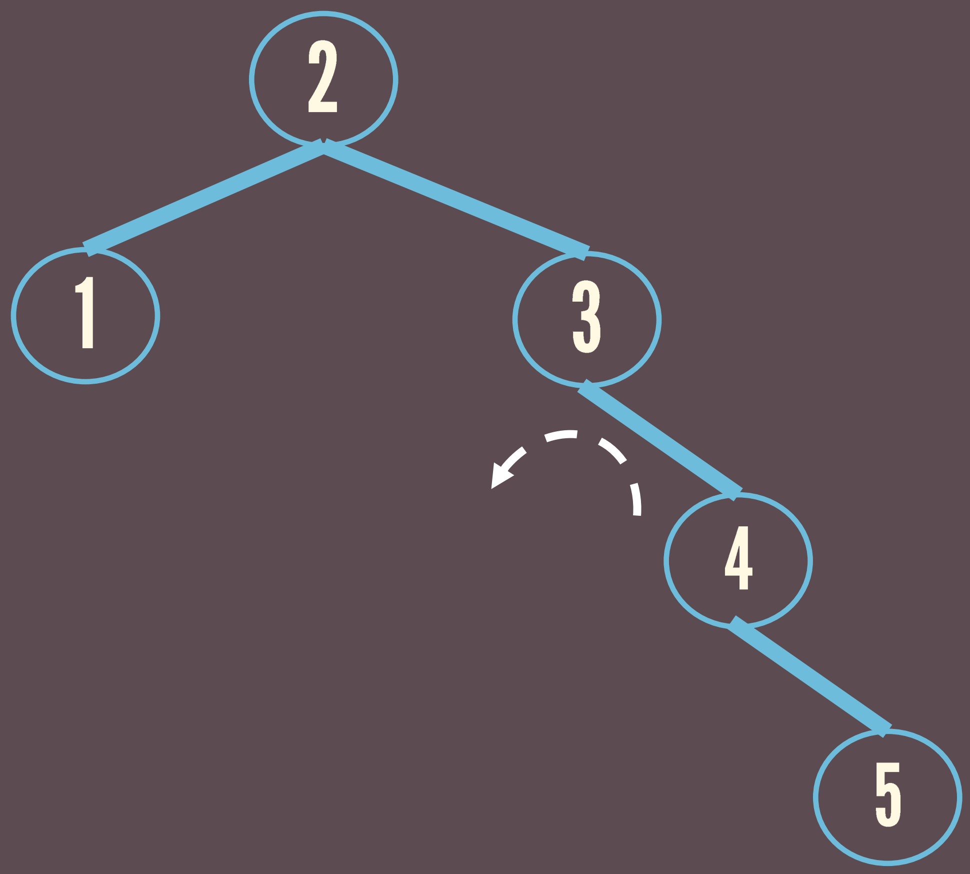

1
2
3

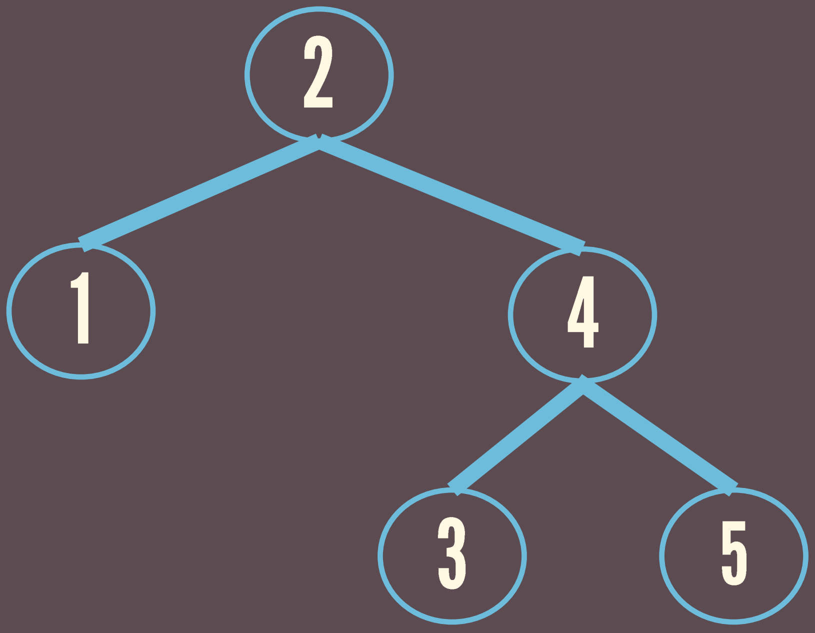

1
2
3

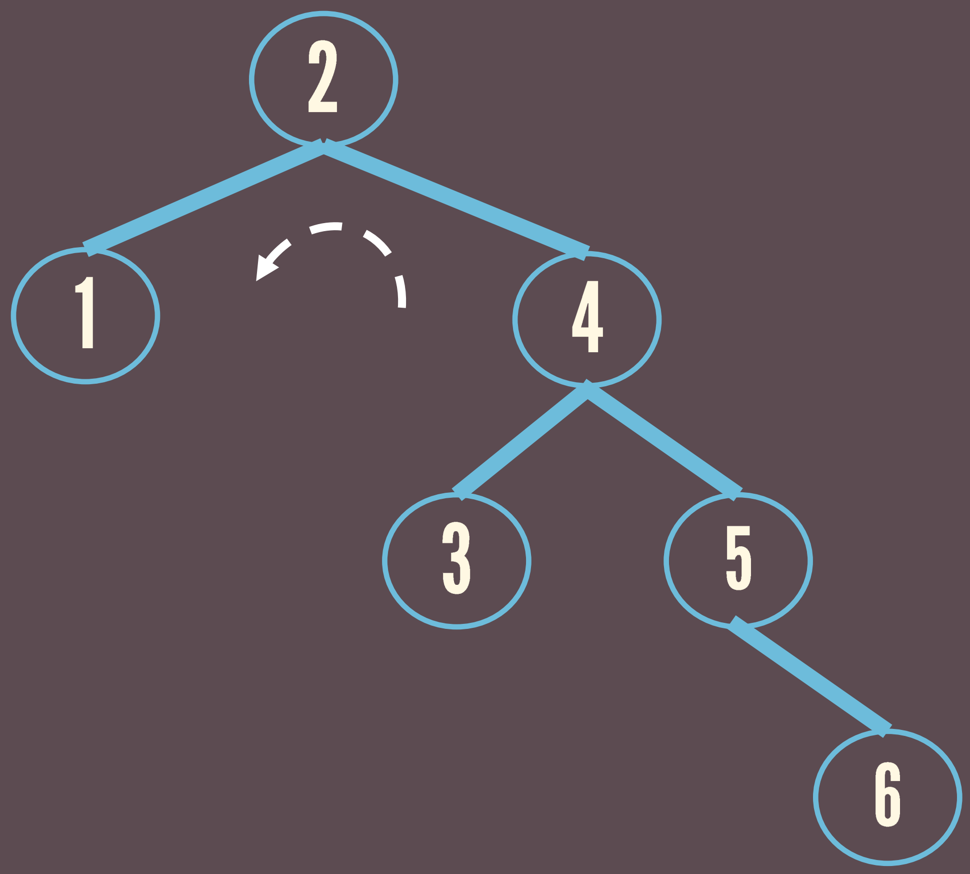

2
1
3

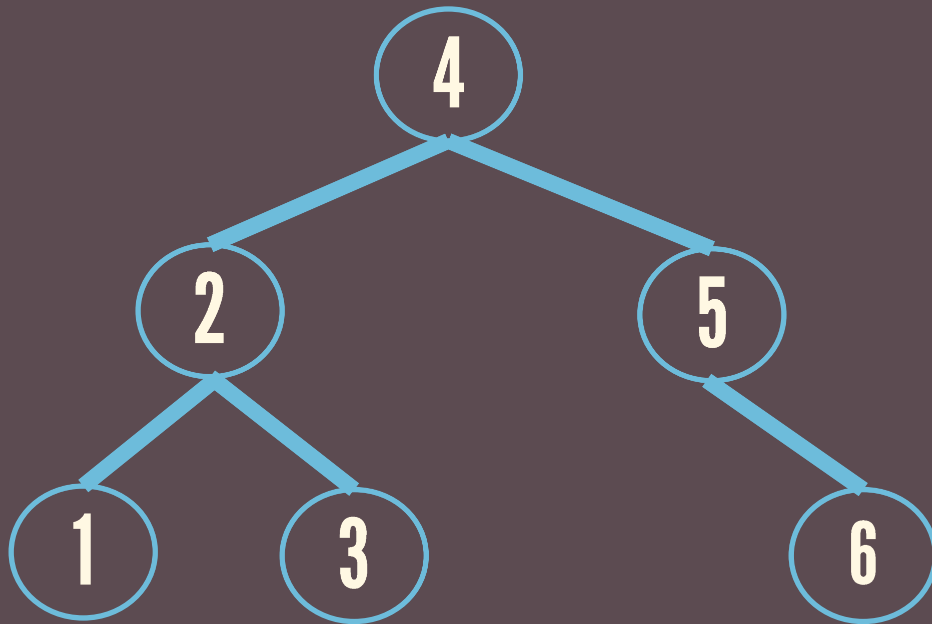

2
1
3
4

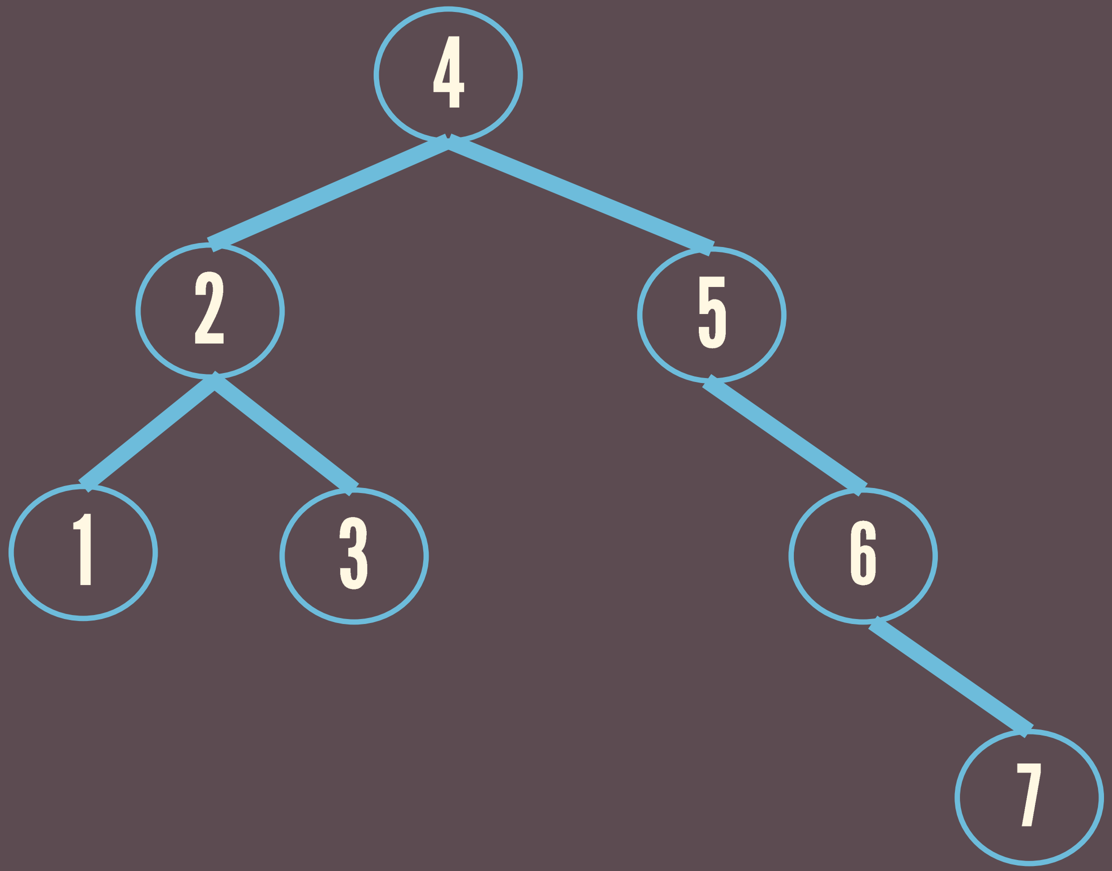

2
1
3
4
5

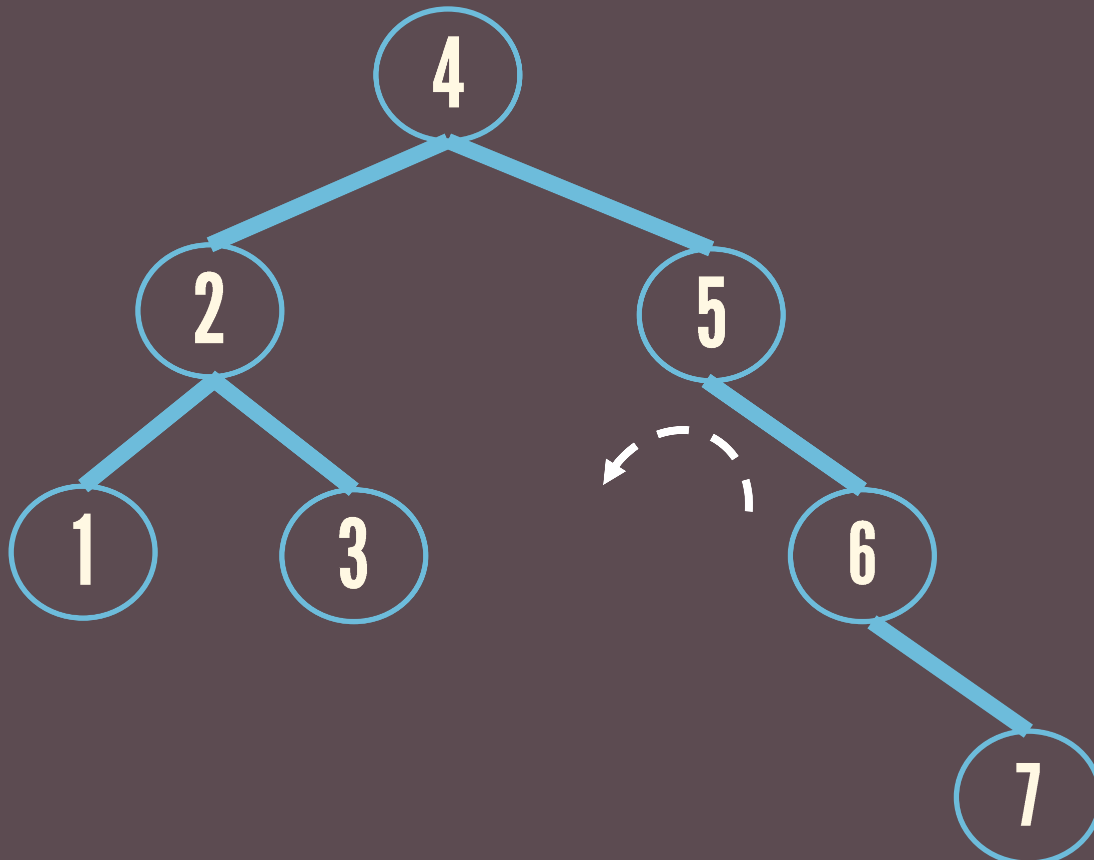

2
1
3
4
5

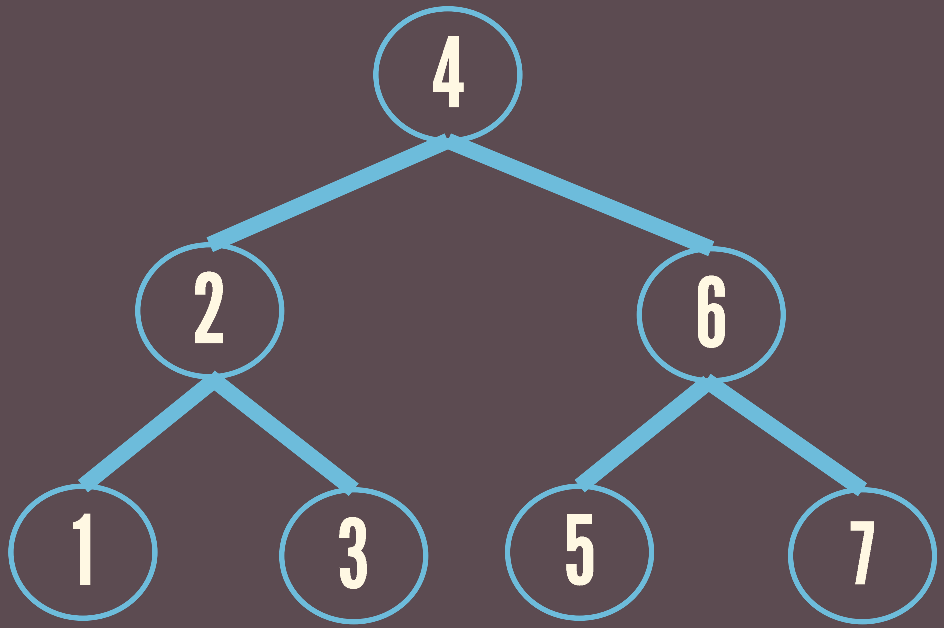

2
1
4
3
5

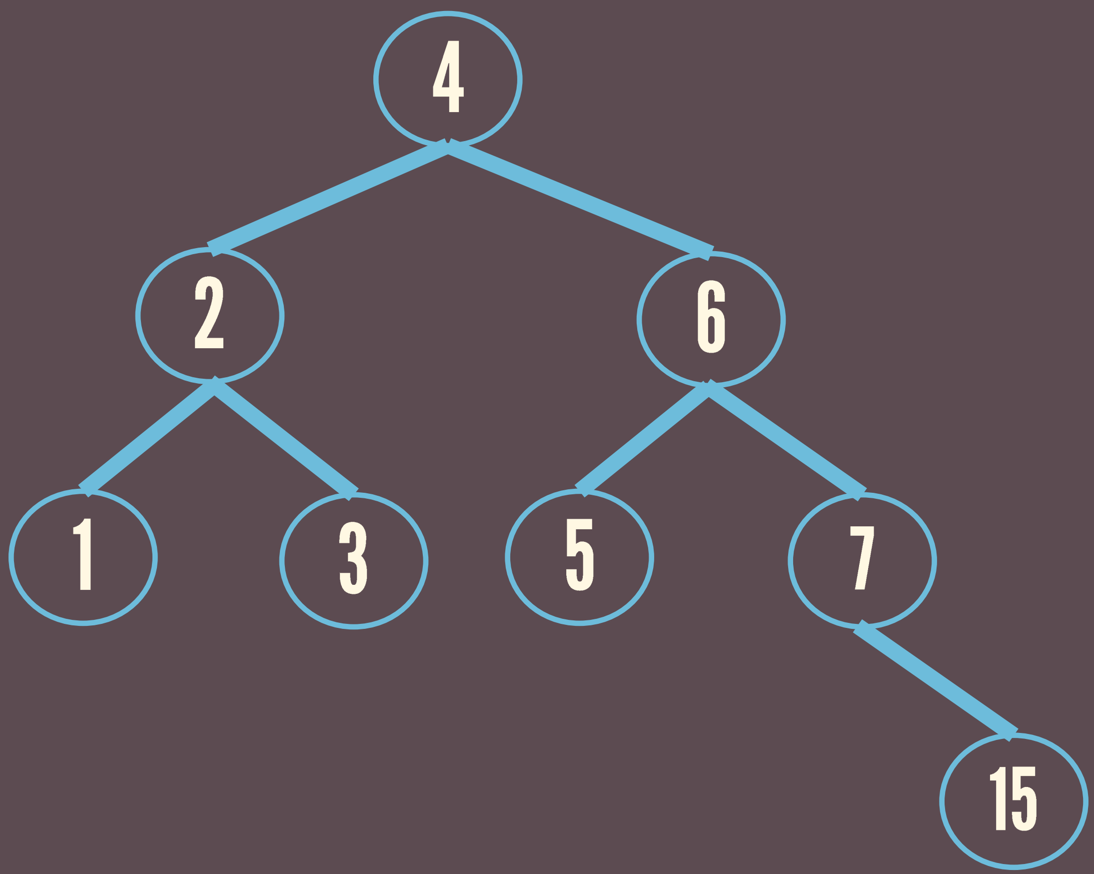

2
1
4
3
5
6

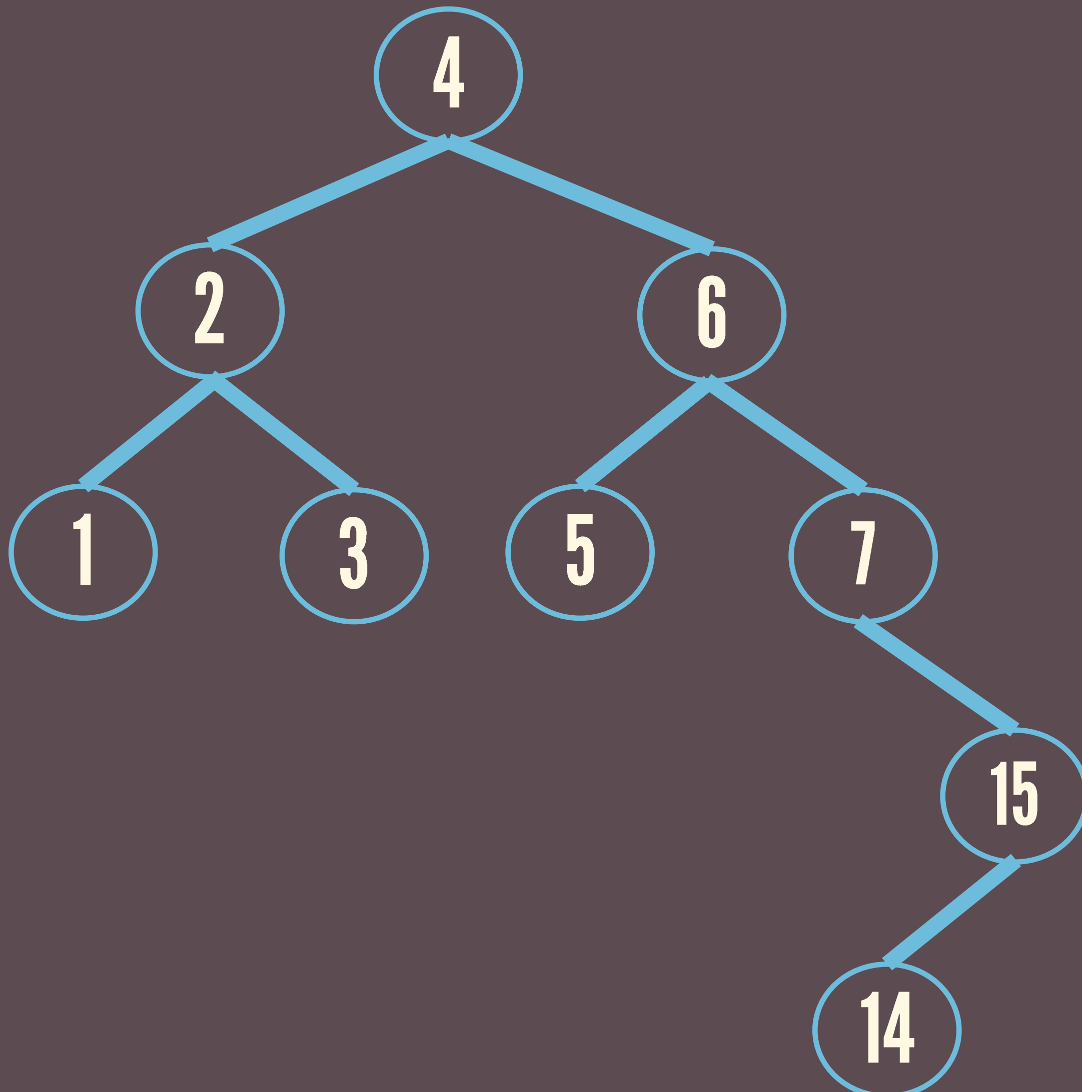

4
2
5
1
3
6

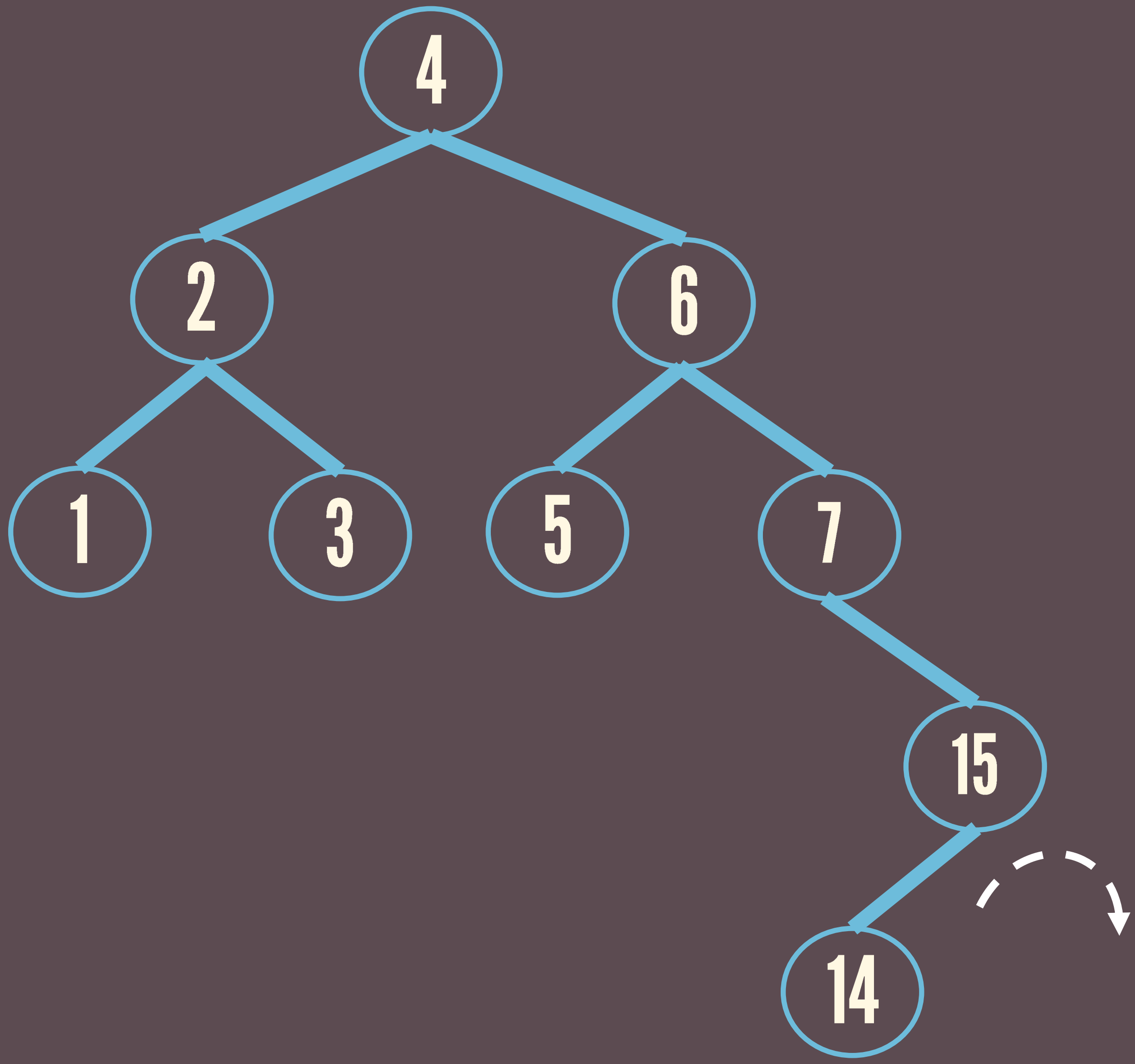

4
2
5
1
3
6
7

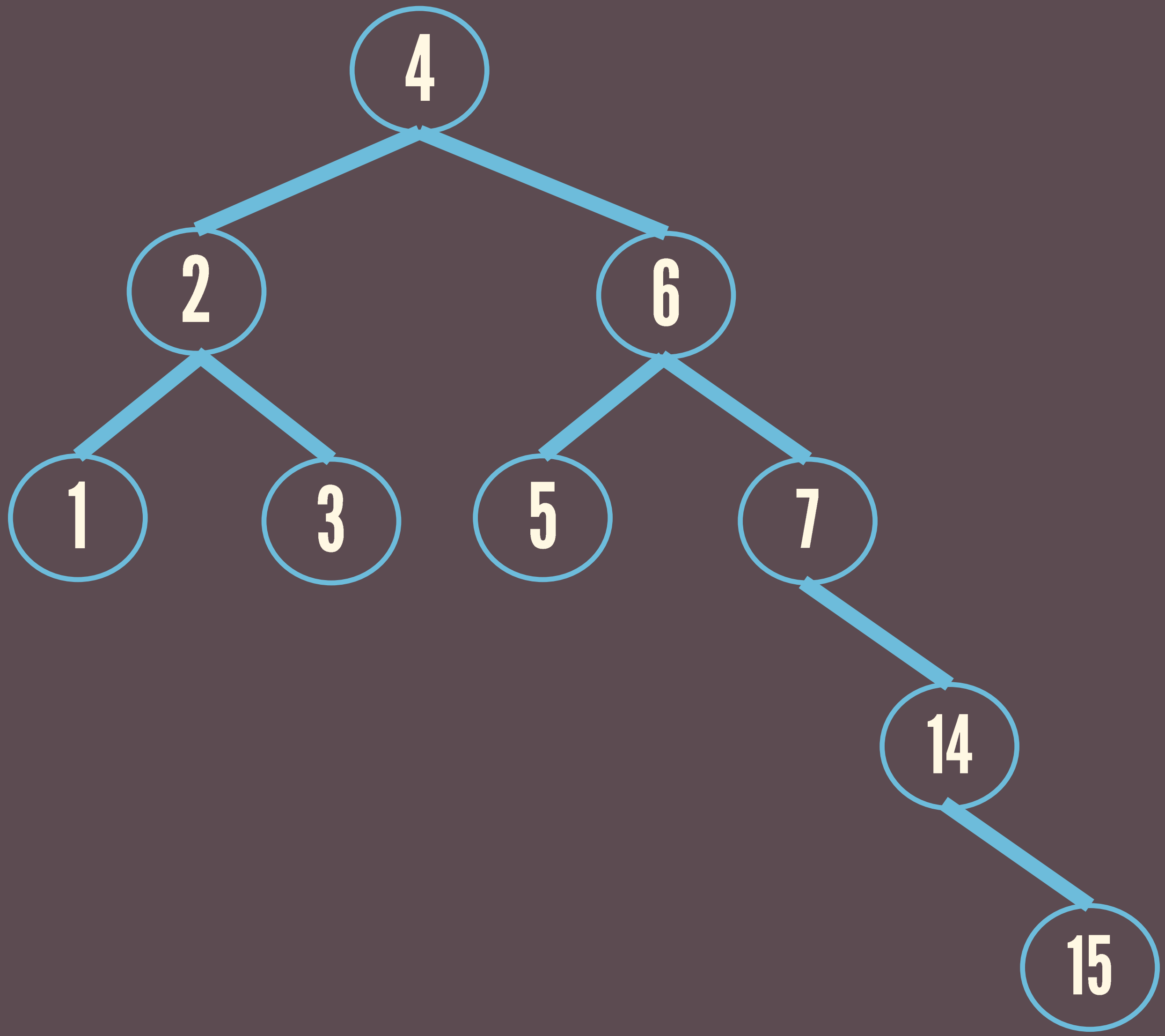

4
2
5
1
3
6
7

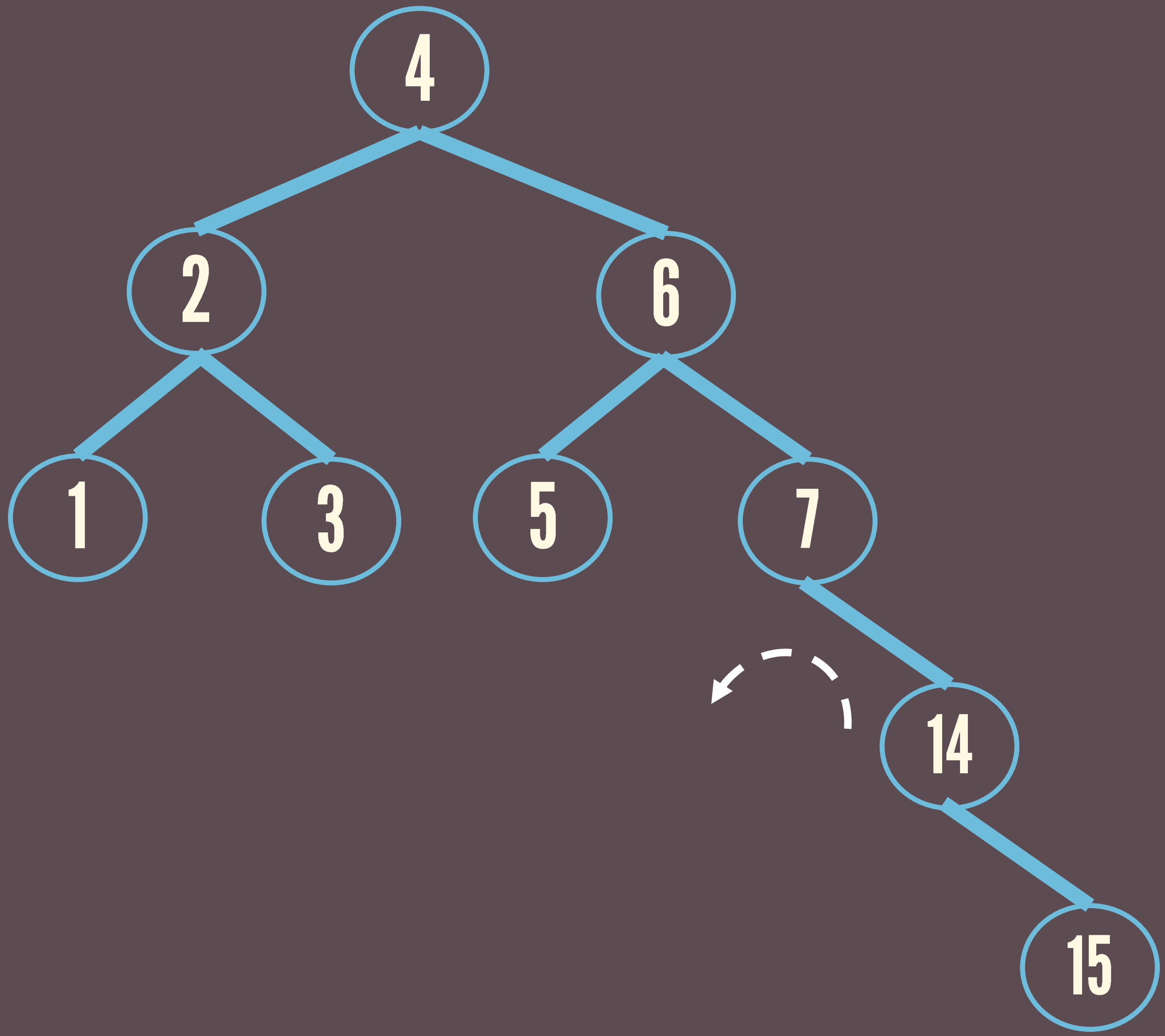

4
2
6
1
3
5
7

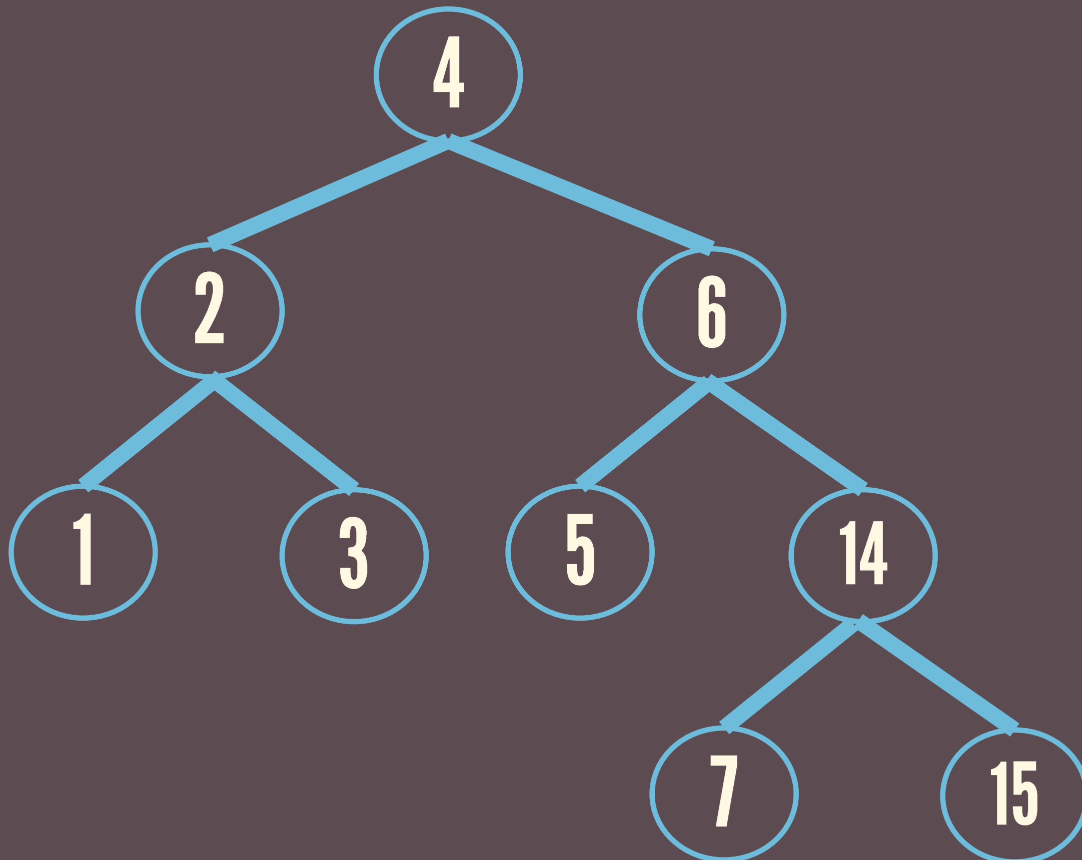

4
2
6
1
3
5
7
15

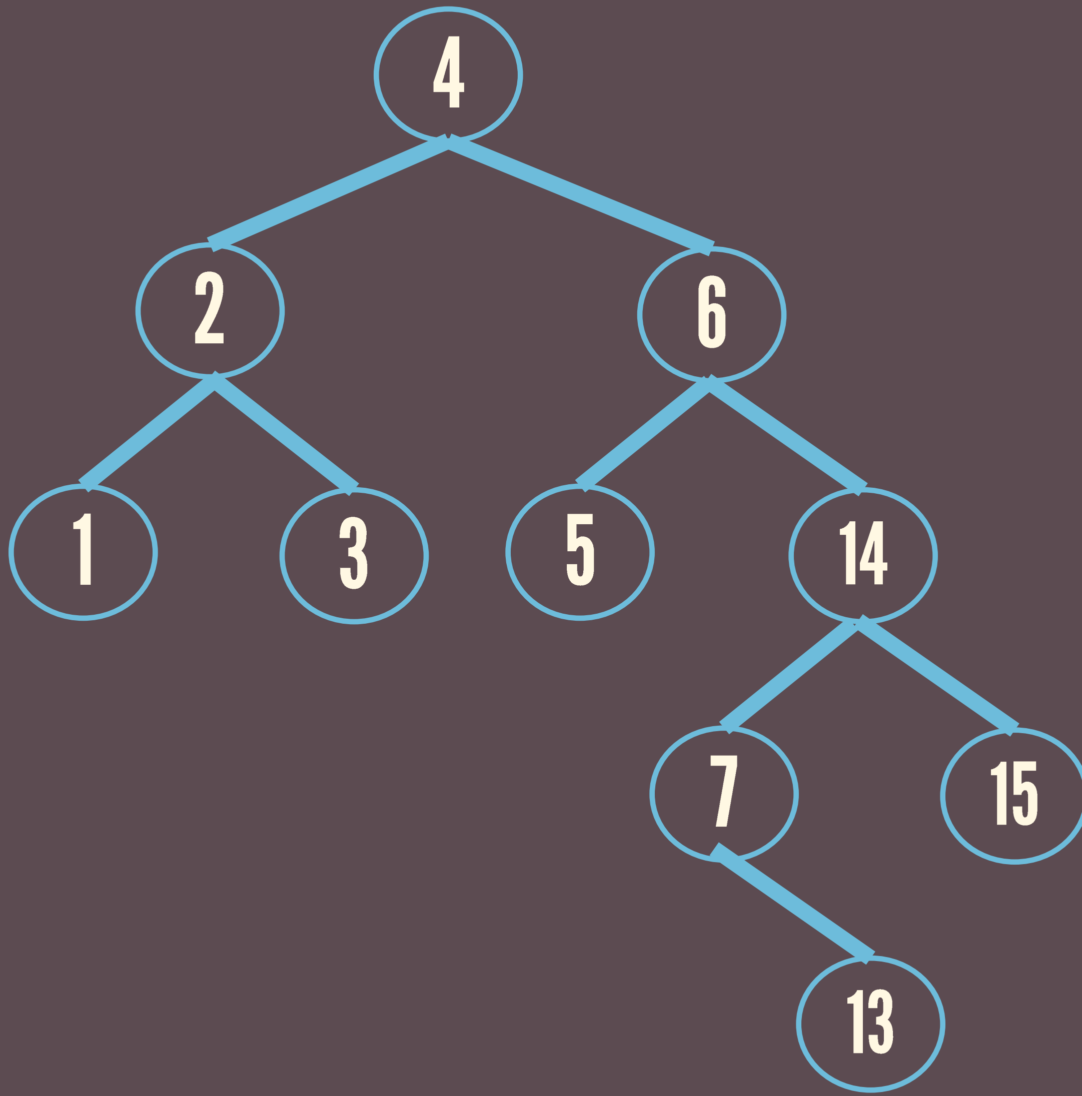

4
2
6
1
3
5
7
15
14

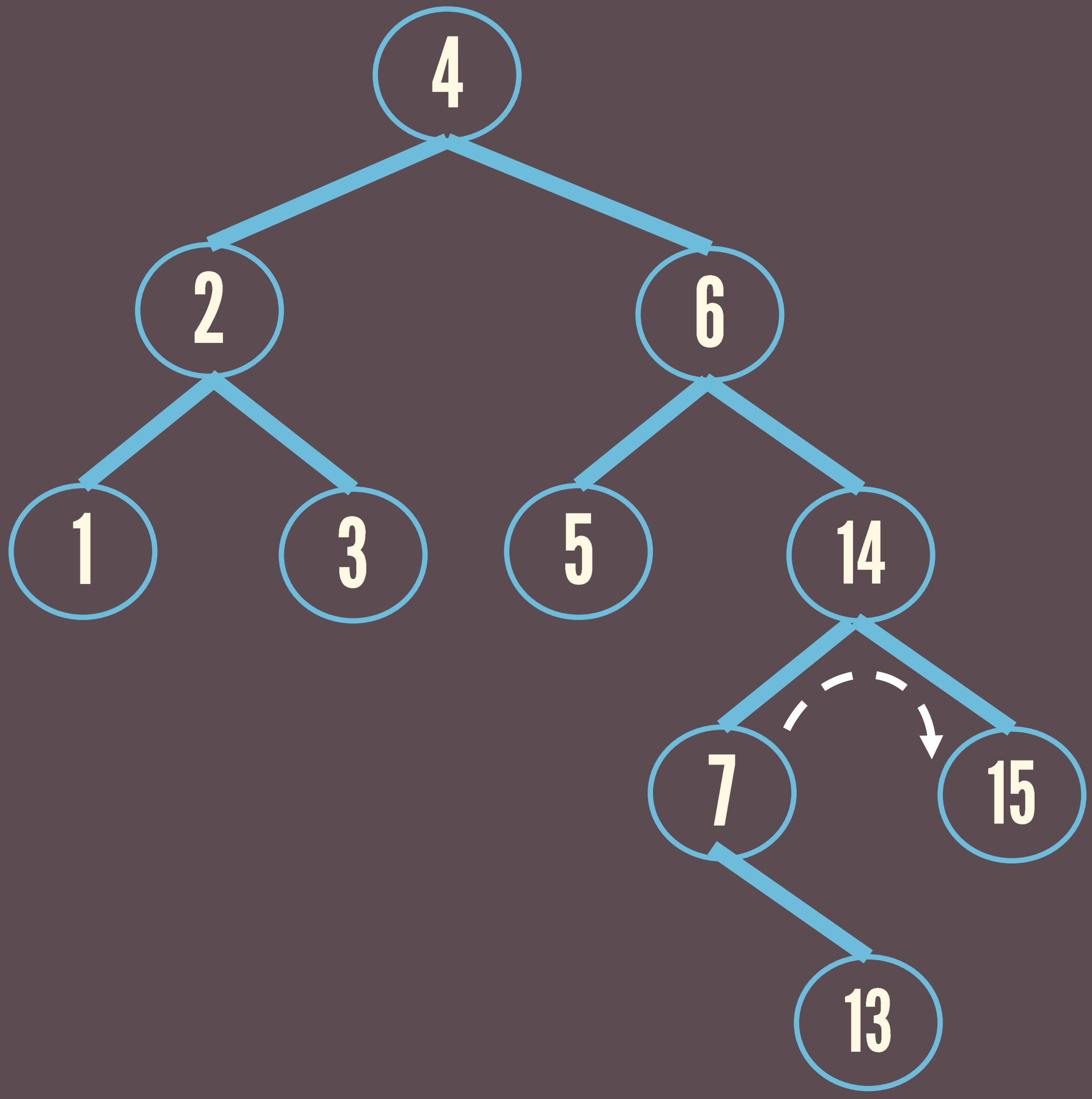

4
2
6
1
3
5
7
15
14

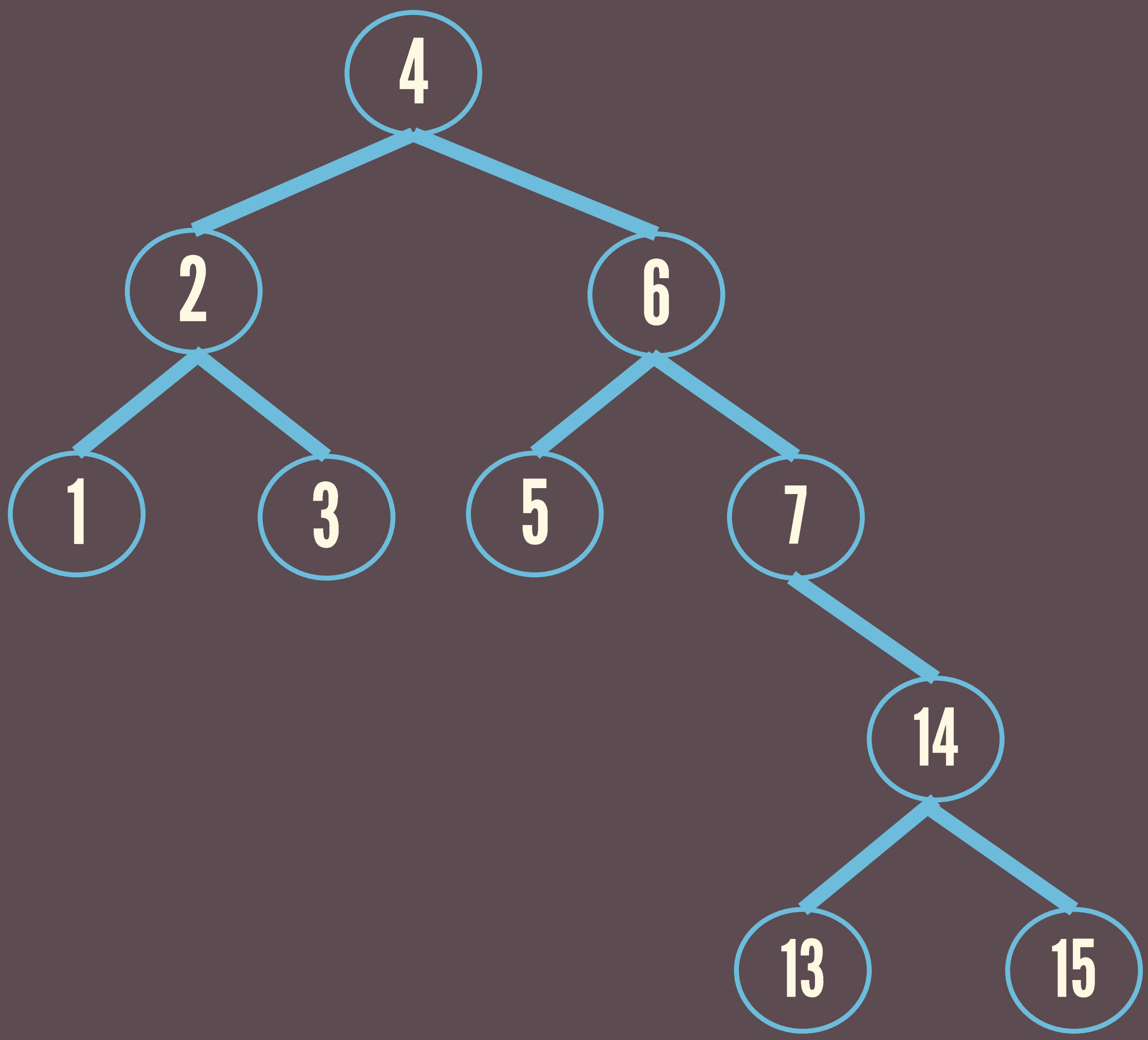

4
2
6
1
3
5
7
14
15

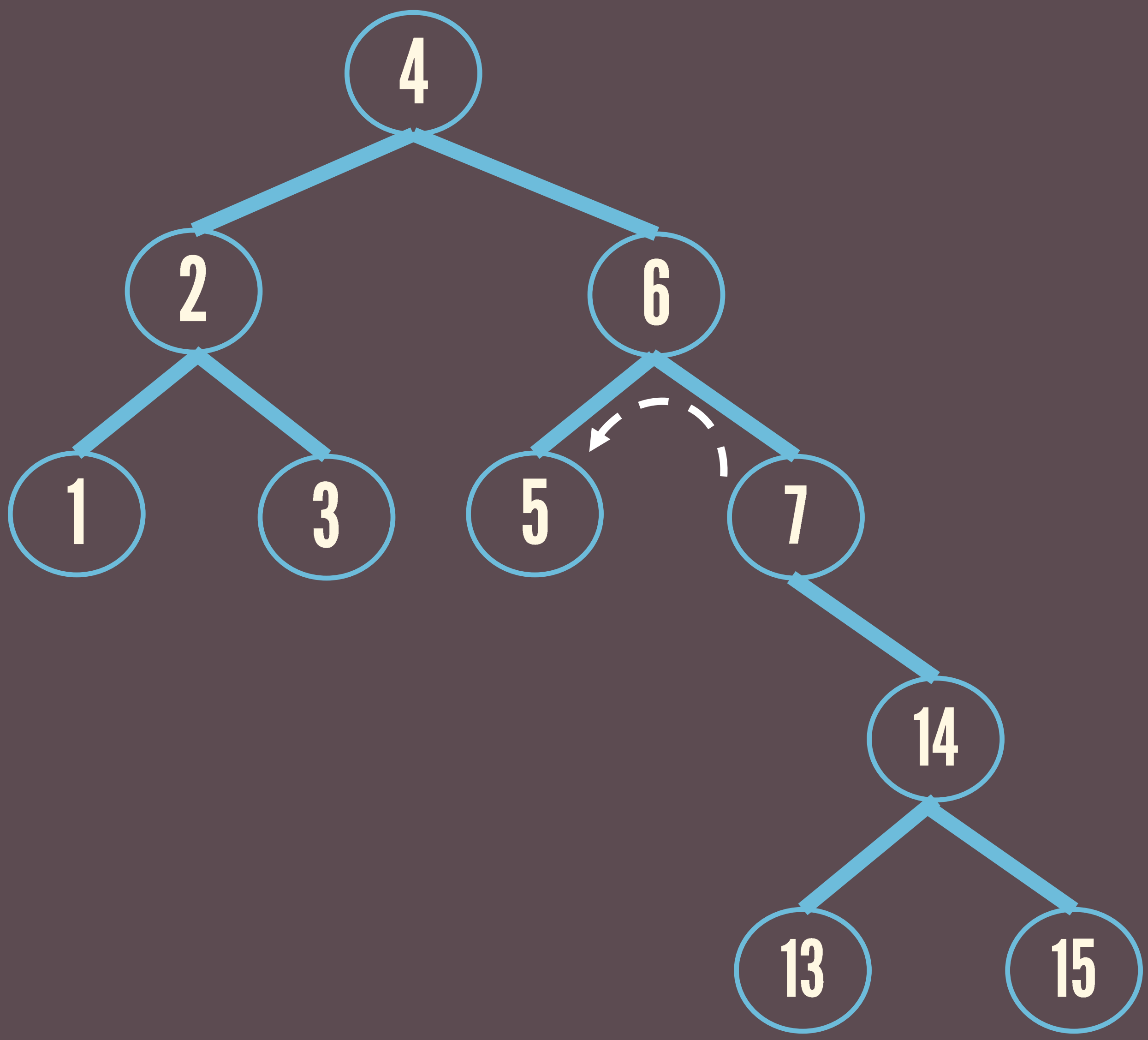

4
2
6
1
3
5
7
14
15

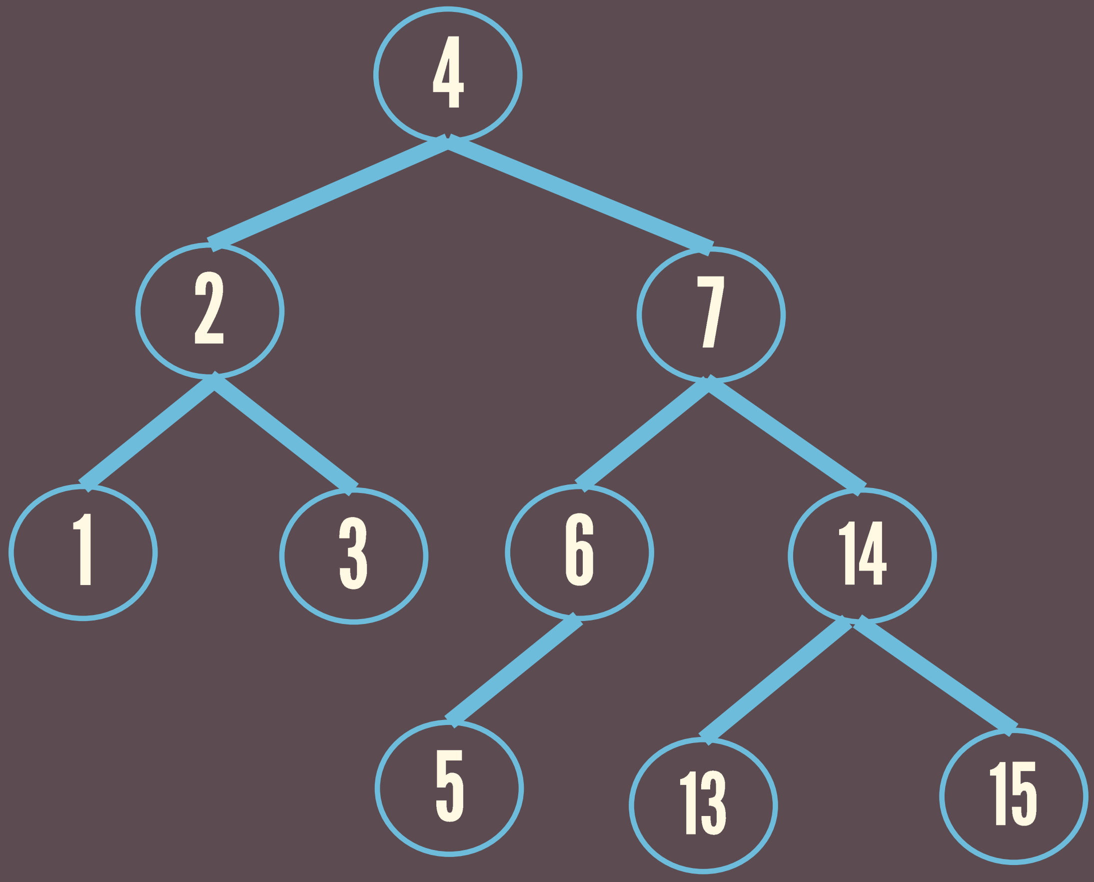

4
2
6
1
3
5
14
7
15


4
2
6
1
3
5
14
7
15
13


4
2
6
1
3
5
14
7
15
13


4
2
6
1
3
5
7
14
13
15


4
2
6
1
3
5
7
14
13
15


4
2
7
1
3
6
14
5
13
15

AVL

OPERATIONS

- find
- insert (with rotations)
- delete (with rotations)
- minimum
- maximum
- successor
- predecessor

```
deleteNode(){

   if the node is a leaf,
     simply remove it.
   if the node is not a leaf,
     replace it with either its redecessor
     or its successor and recursively
     delete that node.
   perform fixup() (the insert fixup) on
   the parent of the repacement up to the
   root.

}
```