Trees

Tree terminology

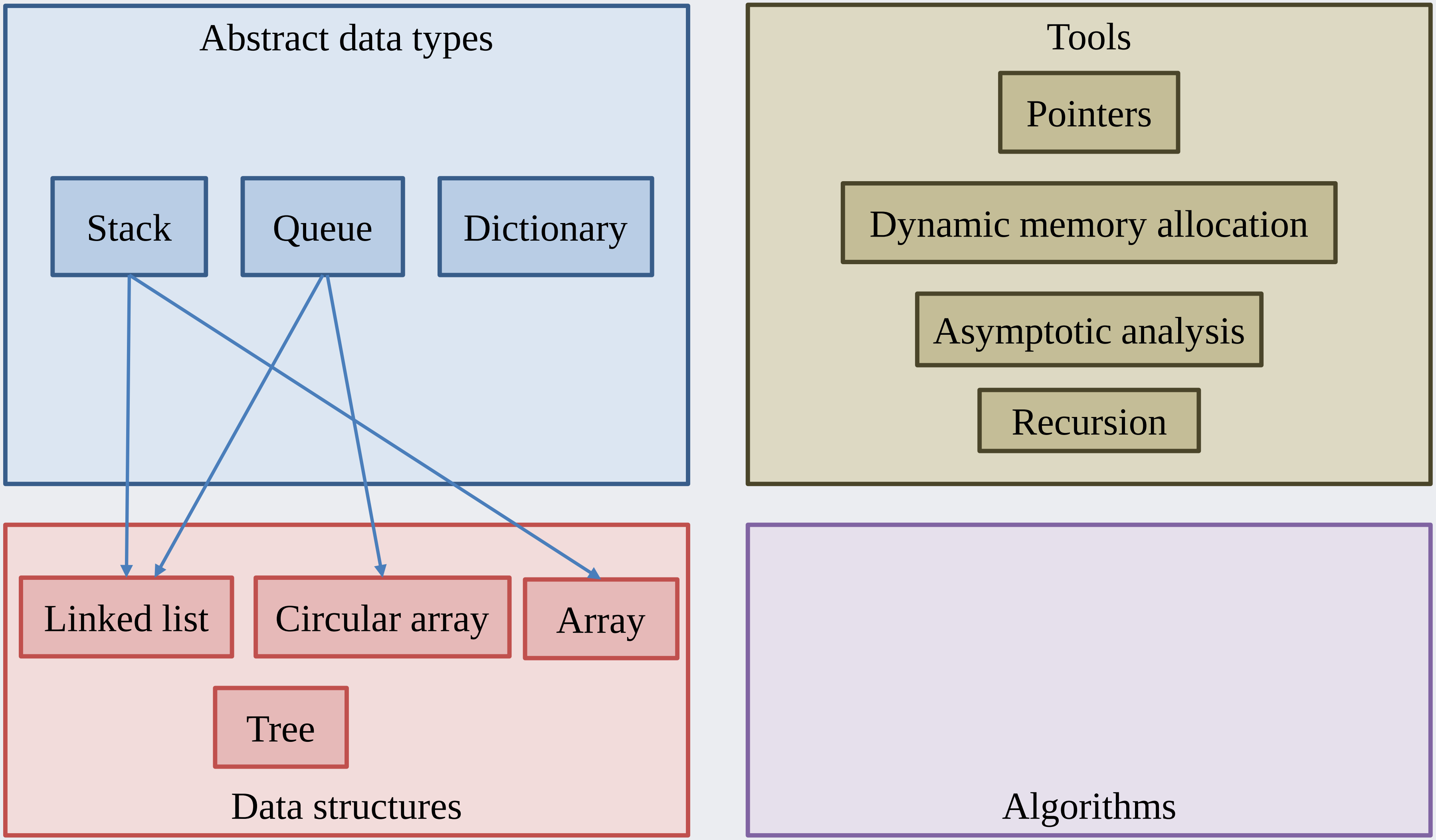
Tree traversal

Search trees

Analysis

The adventure thus far...

CPSC 259 topics up to this point



Review: Dictionary ADT

- Abstract data type (ADT) – a mathematical description of an object and a set of operations on the object
  - Alternatively, a collection of data and the operations for accessing the data

- Example: Dictionary ADT
  - Stores pairs of strings: (word, definition)
  - Operations:
    - Insert(word, definition)
    - Remove(word)
    - Lookup(word)

Insert

- Feet
  - Useful for something, presumably

Find(Z125 Pro)

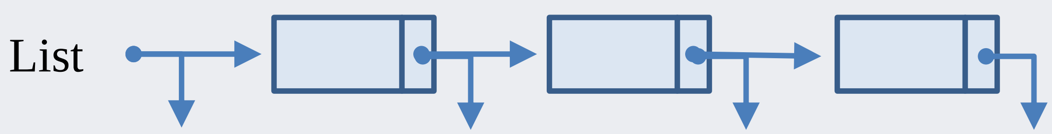
- Z125 Pro
  - Fun in the sun!

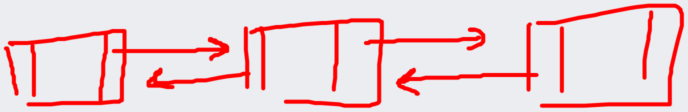
- Super 9 LC
  - Smell like a lawnmower
- Z125 Pro
  - Fun in the sun!
- CB300F
  - For the mild-mannered commuter

data storage implemented with a data structure

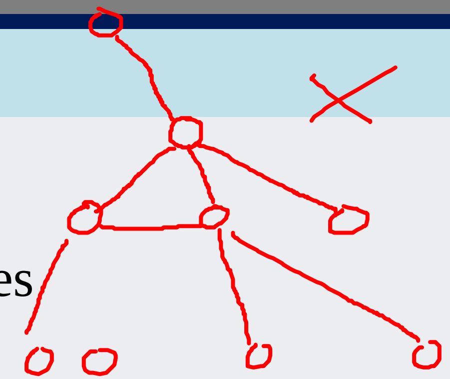
Review: linked lists

- Linked lists are constructed out of *nodes*, consisting of
  - a data element
  - a pointer to another node
- Lists are constructed as chains of such nodes

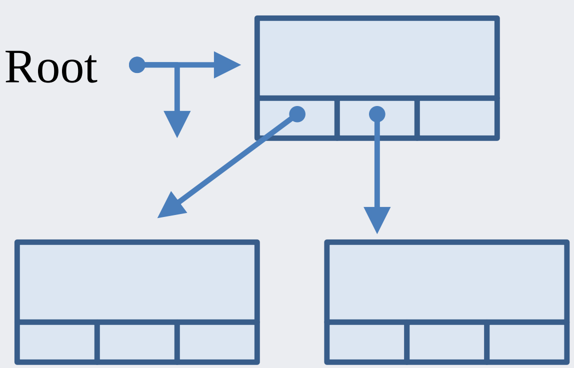


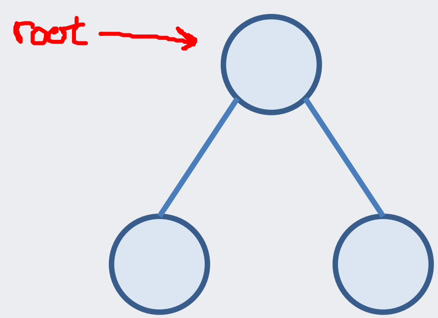


Trees



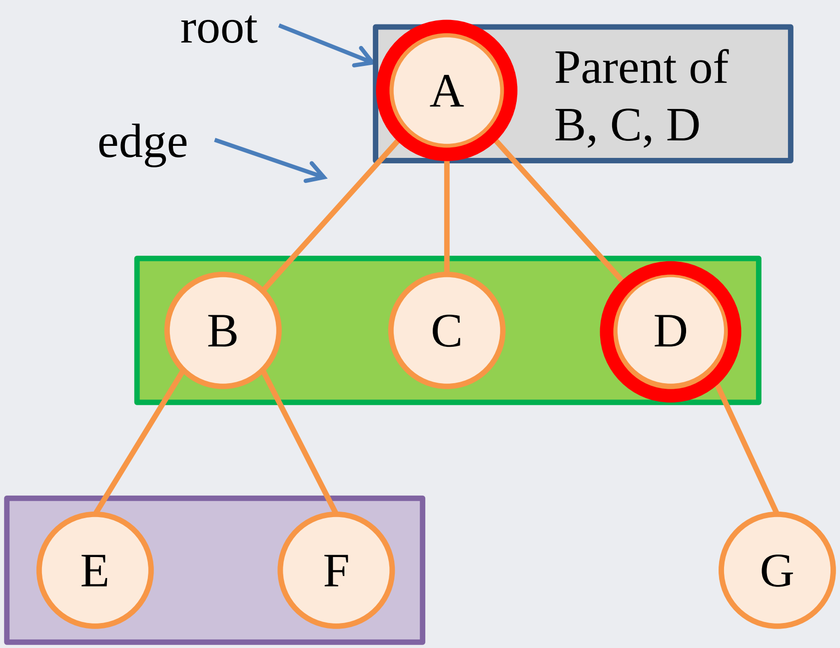
- Trees are also constructed from nodes
  - Nodes may now have pointers to one or more other nodes
- A set of nodes with a single starting point
  - called the *root* of the tree (root node)
- Each node is connected by an *edge* to another node
- A tree is a *connected* graph
  - There is a *path* to every node in the tree
  - A tree has one less edge than the number of nodes



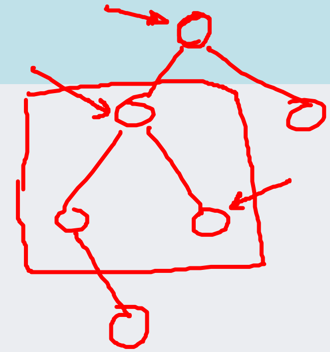


Tree relationships

- Node *v* is said to be a *child* of *u*, and *u* the *parent* of *v* if
  - There is an edge between the nodes *u* and *v*, and
  - *u* is above *v* in the tree,
- This relationship can be generalized
  - E and F are *descendants* of A ✓
  - D and A are *ancestors* of G ✓
  - B, C and D are *siblings* ✓
  - F and G are? ✗



More tree terminology



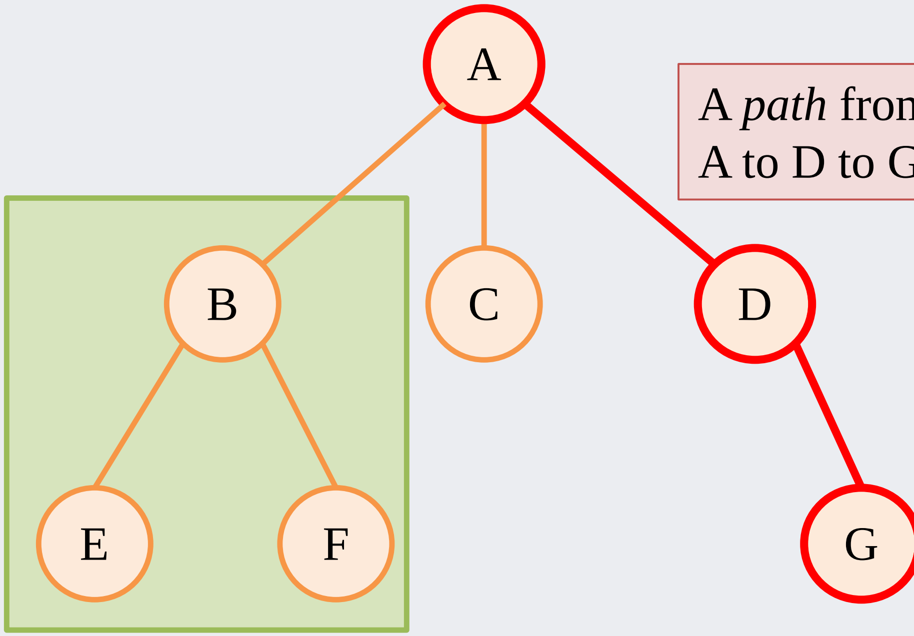
- A *leaf* is a node with no children
- A *path* is a sequence of nodes $v_1 \ldots v_n$
  - where v_i is a parent of v_{i+1} $(1 \leq i \leq n)$
- A *subtree* is any node in the tree along with all of its descendants
- The *degree* of a node is the number of children the node has
- *Branching factor*: the maximum degree of any node in the tree
- A *binary tree* is a tree with at most two children per node, i.e. branching factor = 2

  - The children are referred to as *left* and *right*
  - We can also refer to left and right subtrees

Tree terminology example

n = number of nodes $\boxed{7}$

edges = n - 1



A *path* from A to D to G

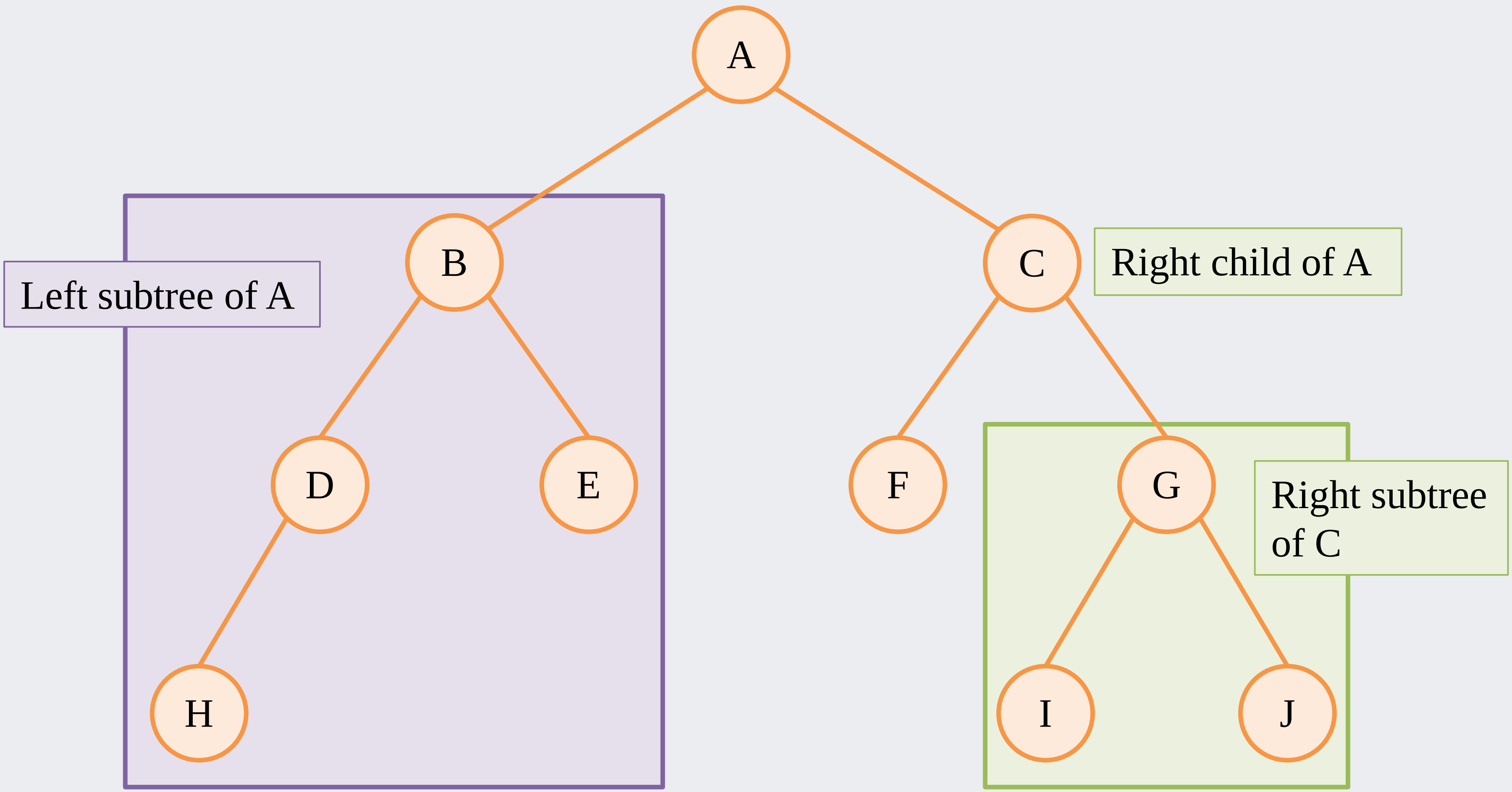
A *subtree* rooted at B

C, E, F, G are *leaf* nodes

Degree of A: 3
Degree of B: 2
Degree of C: 0
Branching factor: 3

Binary tree terminology

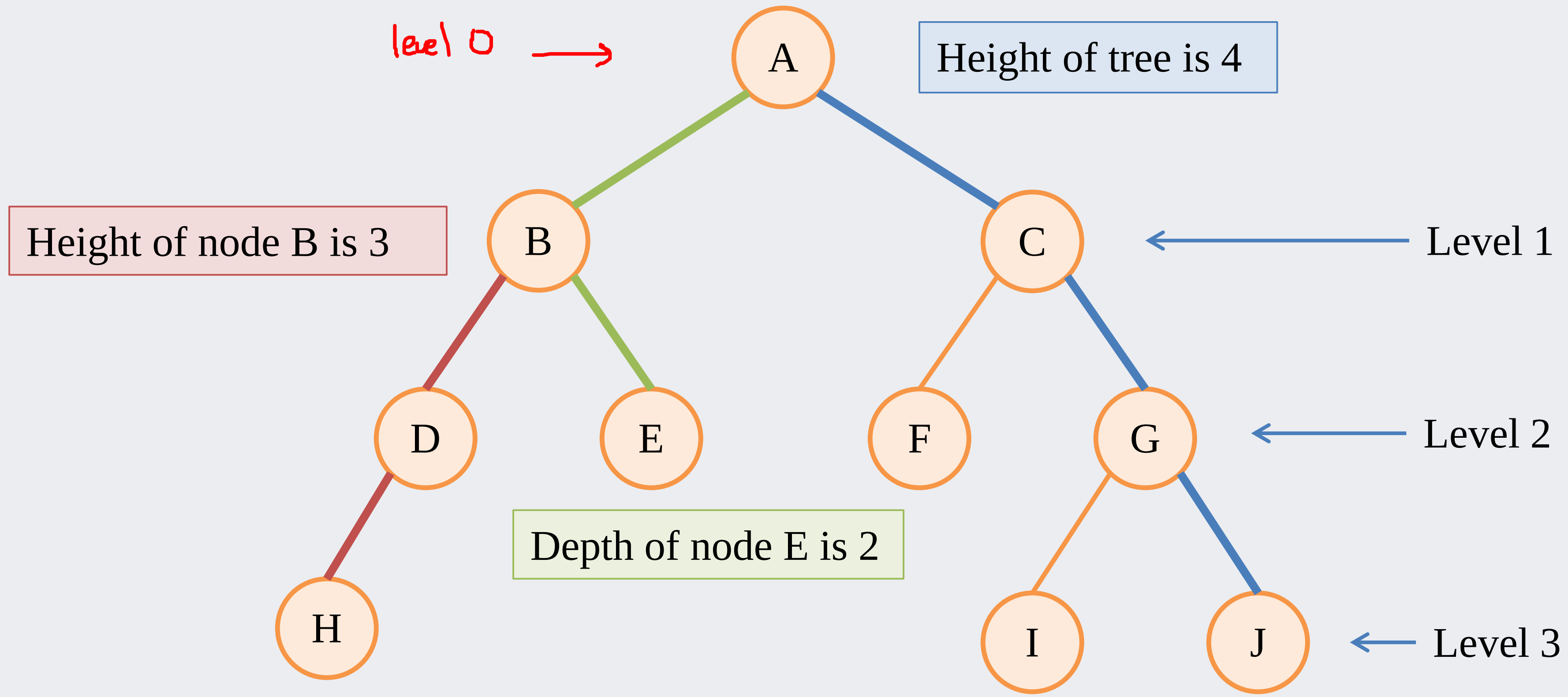
every node has at most 2 children



Measuring trees

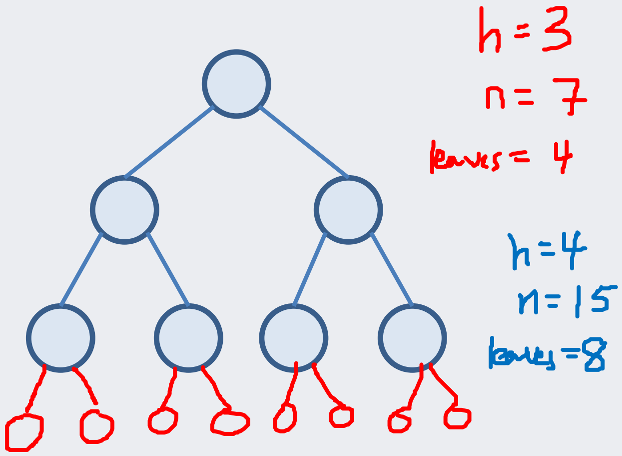
- The *height* of a node *v* is the number of nodes in the longest path from *v* to a leaf (the furthest reachable leaf)
  - The height of the tree is the height of the root
  - An empty tree has height 0, a tree containing only a root has height 1
- The *depth* of a node *v* is the number of edges in the path from *v* to the root
  - This is also referred to as the *level* of a node
- Note that there is a slightly different formulation of the height of a tree
  - where the height of an empty tree is -1
  - but for this course we will use the definition of empty tree height is zero

Tree measurements explained



Perfect binary trees

- A binary tree is *perfect*, if
  - No node has only one child
  - And all the leaves have the same depth
- A perfect binary tree of height h has
  - $2^h - 1$ nodes, of which 2^{h-1} are leaves
- Perfect trees are also *complete*



Height of a perfect tree

binary

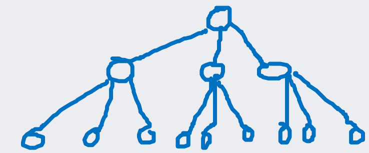
height: h

- Each level doubles the number of nodes
  - Level 0 has 1 node (2^0)
  - Level 1 has 2 nodes (2^1) or 2 times the number in Level 0
  - Level 2 has 4 nodes (2^2)

$\ell = h - 1$

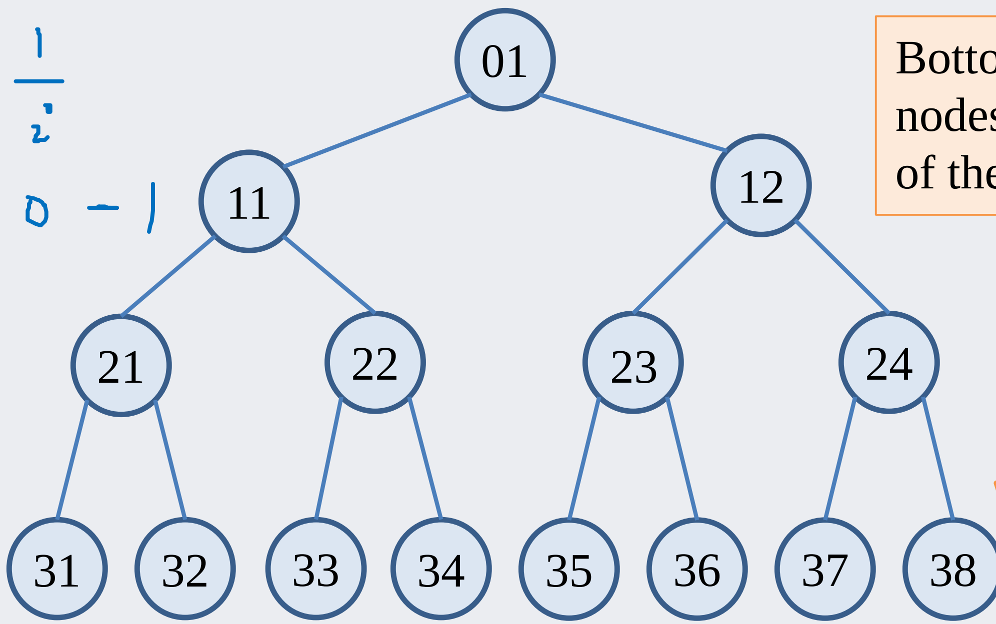
- Therefore a perfect tree with $\cancel{h}$ ℓ levels has $2^{\cancel{h}\,\ell+1} - 1$ nodes

generally, a perfect k-ary
every node has k children,
all leaves at same level



$\frac{1}{2^4}\quad \frac{1}{2^3}\quad \frac{1}{2^2}\quad \frac{1}{2^1}\quad \frac{1}{2^0}$

$= \frac{1}{2^5}\quad 0\quad 0\quad 0\quad 0\quad 0\quad - 1$

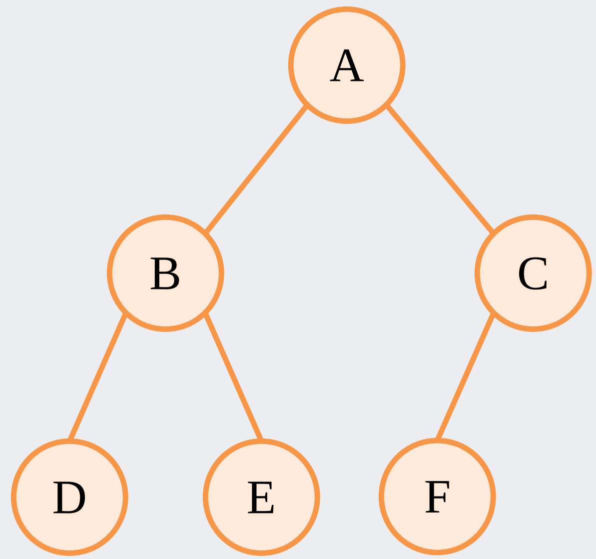


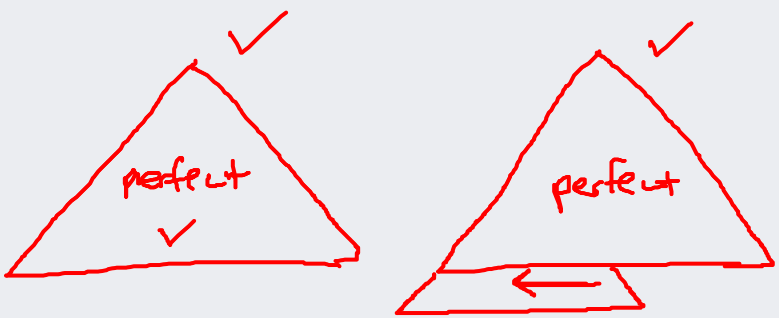
Bottom level has 2^h nodes, i.e. just over half of the nodes are leaves

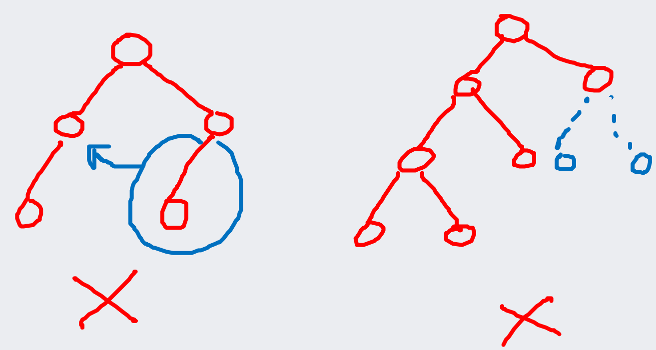
Complete binary trees

"almost perfect"

- A binary tree is *complete* if
  - The leaves are on at most two different levels,
  - The second to bottom level is completely filled in and
  - The leaves on the bottom level are as far to the left as possible





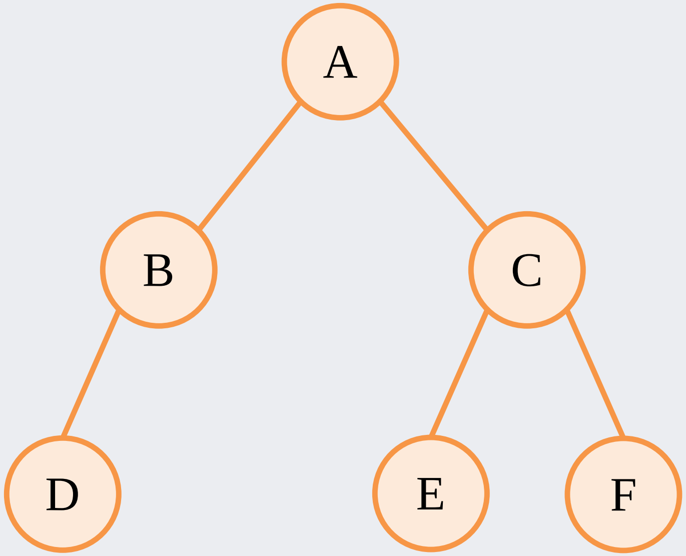


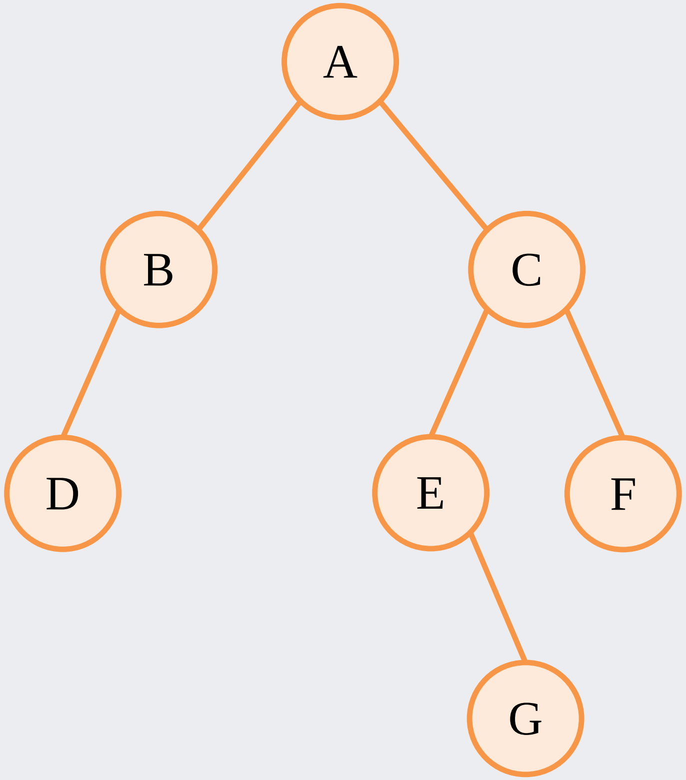
Balanced binary trees

- A binary tree is *balanced* if
  - Leaves are all about the same distance from the root
  - The exact specification varies
- Sometimes trees are balanced by comparing the height of nodes
  - e.g. the height of a node’s right subtree is at most one different from the height of its left subtree (e.g. AVL trees)
- Sometimes a tree's height is compared to the number of nodes
  - e.g. red-black trees

Balanced binary trees

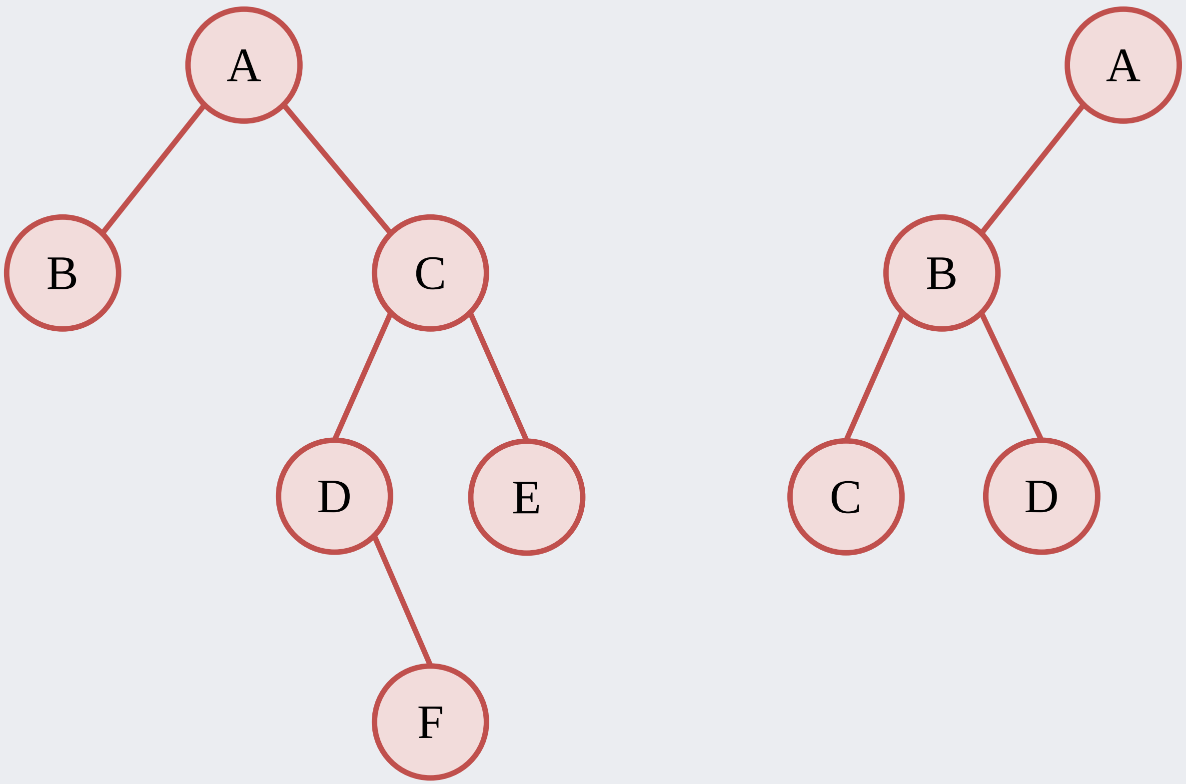
by AVL property: L/R subtree heights differing ≤ 1





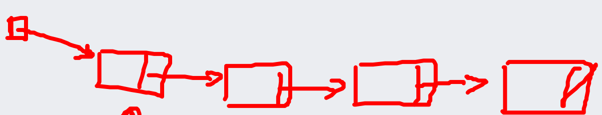
Imbalanced binary trees

by AVL property



- A traversal algorithm for a binary tree visits each node in the tree
  - Typically, it will do something while visiting each node!
- Traversal algorithms are naturally recursive
- There are three traversal methods
  - inOrder
  - preOrder
  - postOrder

Linked lists

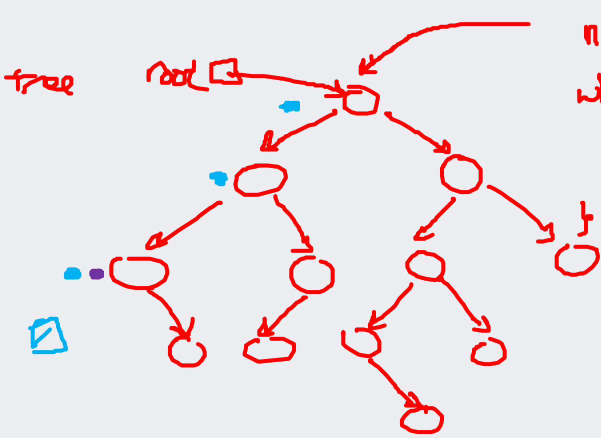


node* p

```
while (p != NULL) {
   do something interesting with p;
   p = p->next;
}
```

tree

root



node* p

```
while (p != NULL) {
   work with p
   p = p->left;
}
```

inOrder traversal algorithm

```c
void inOrder(BNode* nd)
{
  if (nd != NULL)
  {
    inOrder(nd->left);
    visit(nd);
    inOrder(nd->right);
  }
}
```

```c
typedef struct BNode
{
  int data;
  struct BNode* left;
  struct BNode* right;
} BNode;
```

The `visit` function would do whatever the purpose of the traversal is (e.g. print the data value of the node).

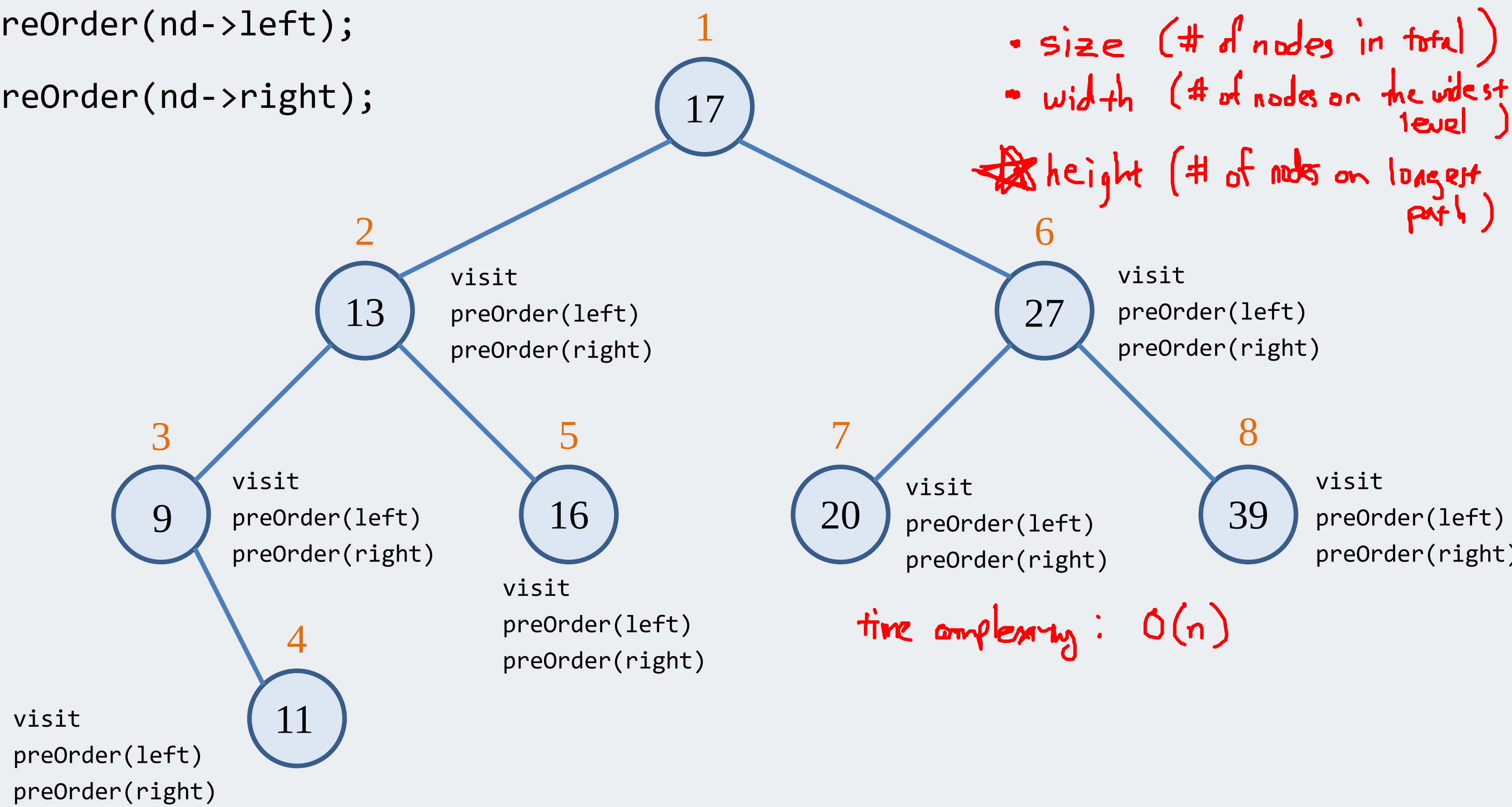
preOrder traversal

```
visit(nd);
preOrder(nd->left);
preOrder(nd->right);
```

nd
left
right

space complexity
(how much stack/heap space gets used)?

- size (# of nodes in total)
- width (# of nodes on the widest level)
- ~~☆~~ height (# of nodes on longest path)

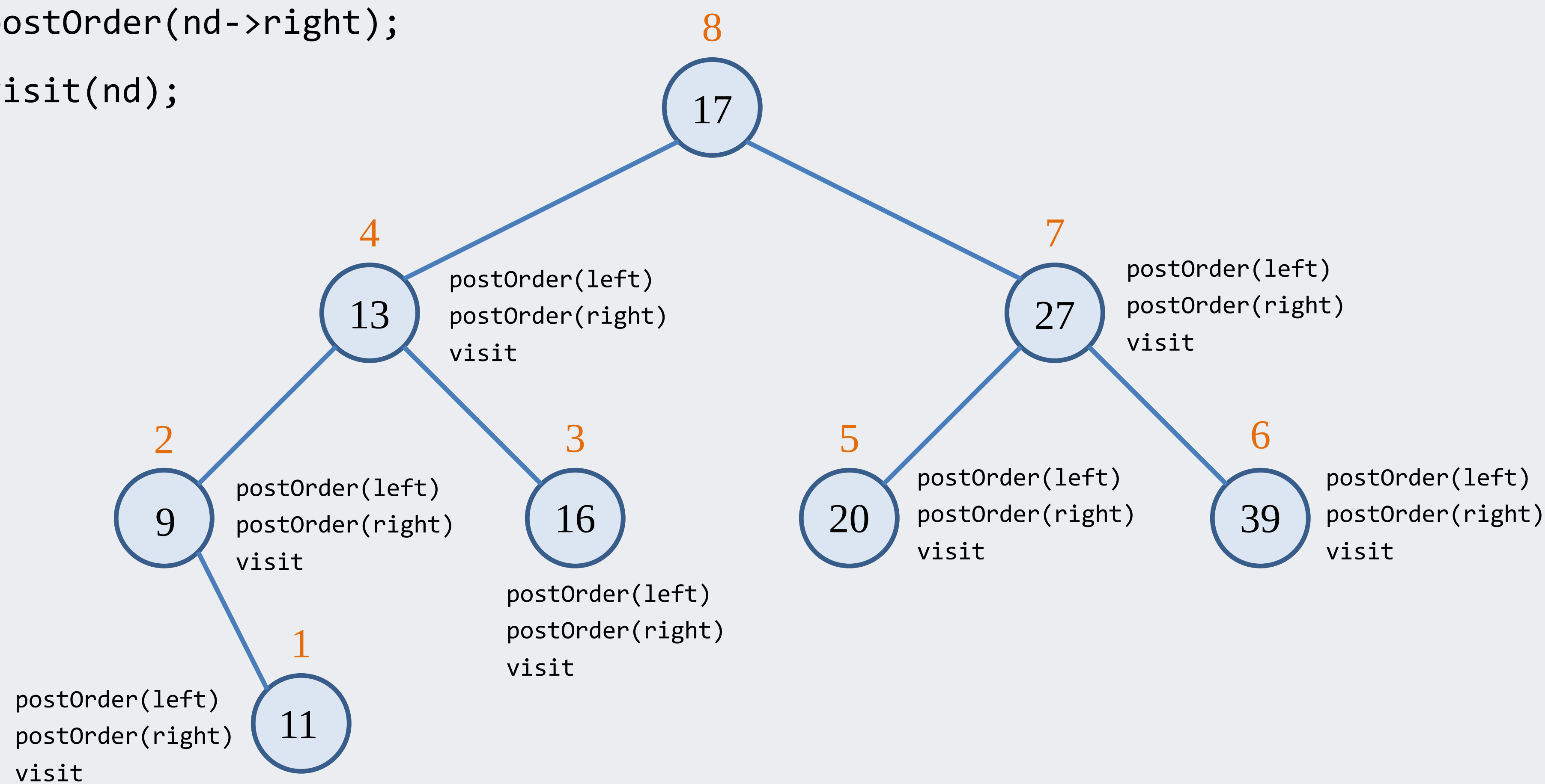


time complexity: $O(n)$

postOrder traversal

left
right
current

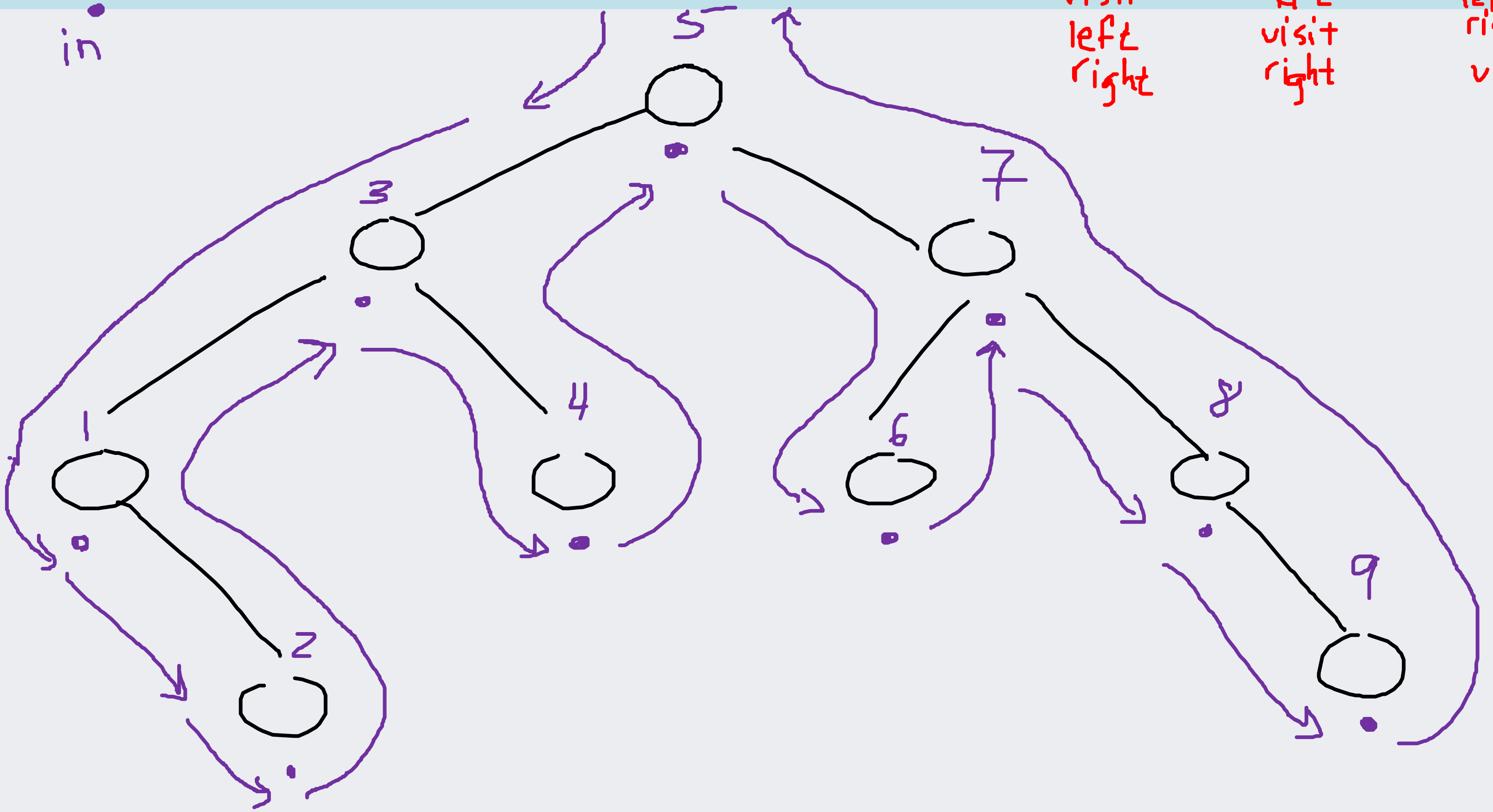
```
postOrder(nd->left);
postOrder(nd->right);
visit(nd);
```



preorder ○ post

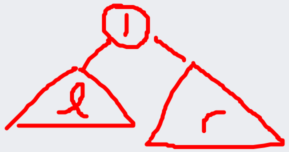
in

pre	in	post
visit	left	left
left	visit	right
right	right	visit



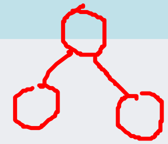
computing the number of nodes in a tree
counting

```
int count (BNode* nd) {
    if (nd == NULL)
        return 0;
    else {
        int l = count(nd->left);
        int r = count(nd->right);
        int total = 1 + l + r;
        return total;
    }
}
```



post-order traversal!

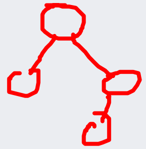
deallocate every node (using free())



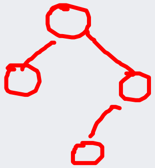
post-order

copy a tree

source tree



copied tree



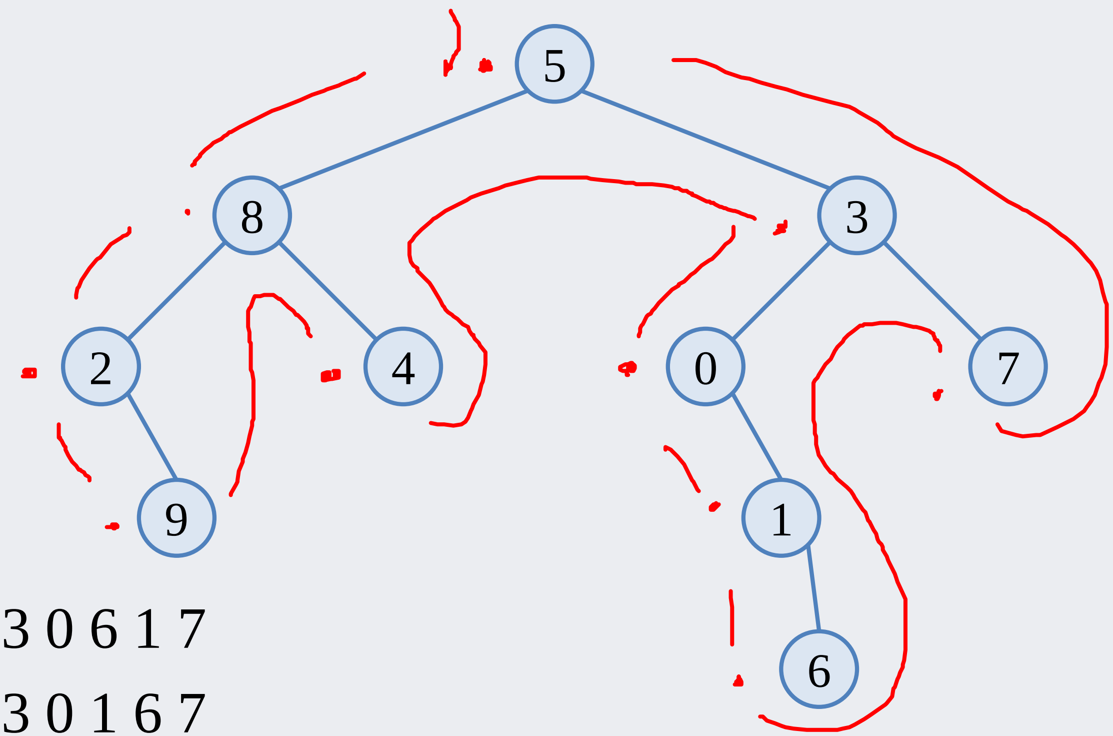
pre-order
- allocate node
- assign children as recursive return

pre-order
- root of subtree is visited first

post-order
- root of subtree is visited last

in-order
- structurally, tells left/right positional information

- What will be the sequence visited by a pre-order traversal from the root?



pre-order:
root
left
right

A. 5 8 9 2 4 3 0 6 1 7

B. 5 8 2 9 4 3 0 1 6 7

C. 5 8 3 2 4 0 7 9 1 6

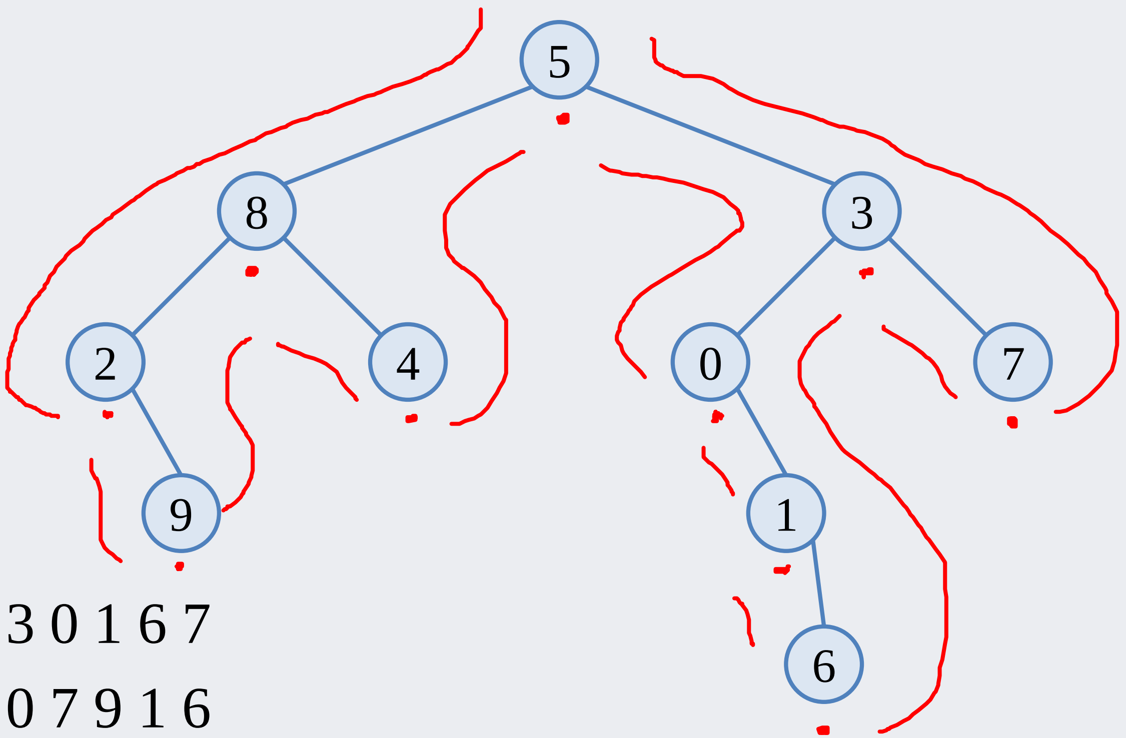
D. 6 1 0 7 3 5 9 2 4 8

E. 9 2 4 8 6 1 0 7 3 5

$5 \ \underbrace{8\ 2\ 9\ 4}_{\text{left}} \ \underbrace{3\ 0\ 1\ 6\ 7}_{\text{right}}$

iClicker 09.2

- What will be the sequence visited by an in-order traversal from the root?



in-order:
left
root
right

A. 5 8 2 9 4 3 0 1 6 7

B. 5 8 3 2 4 0 7 9 1 6

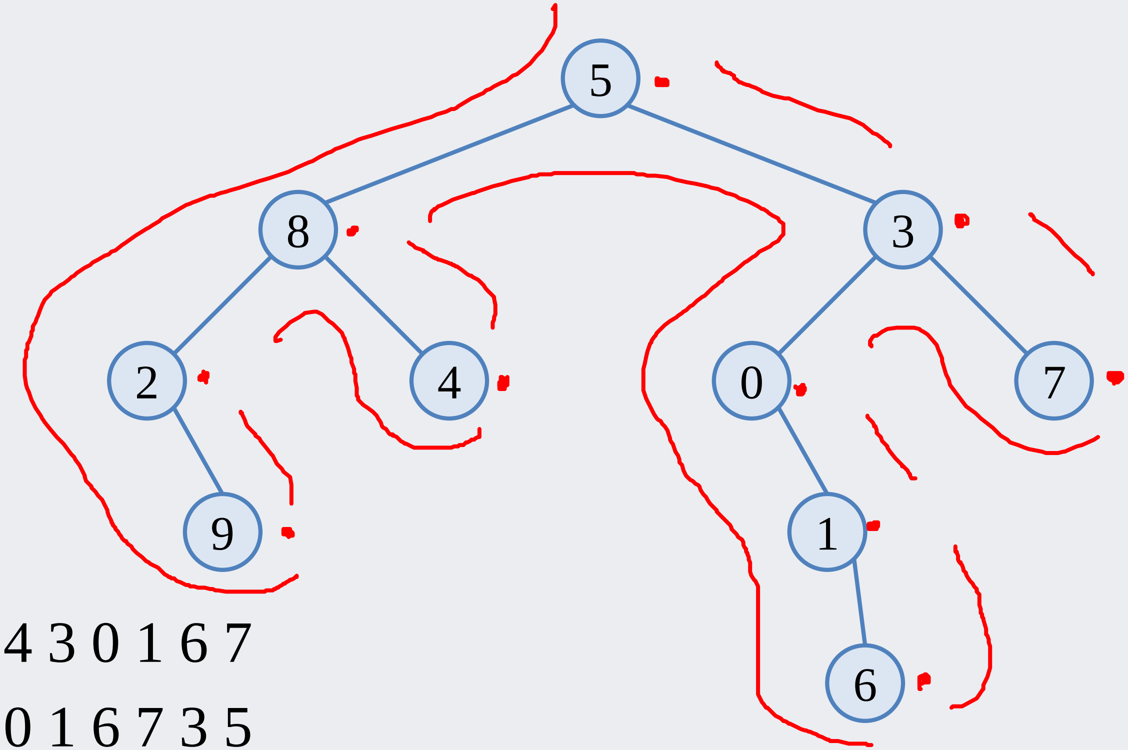
C. 6 1 0 7 3 5 9 2 4 8

D. 2 9 8 4 5 0 1 6 3 7

E. 9 2 4 8 6 1 0 7 3 5

$\underbrace{2\ 9\ 8\ 4}_{\text{left}}\ 5\ \underbrace{0\ 1\ 6\ 3\ 7}_{\text{right}}$

- What will be the sequence visited by a post-order traversal from the root?



post-order:
left
right
root

A. 5 8 2 9 4 3 0 1 6 7

B. 2 9 8 4 0 1 6 7 3 5

C. 6 1 0 7 3 5 9 2 4 8

D. 9 2 ~~8 4~~ 4 8 6 1 0 7 3 5

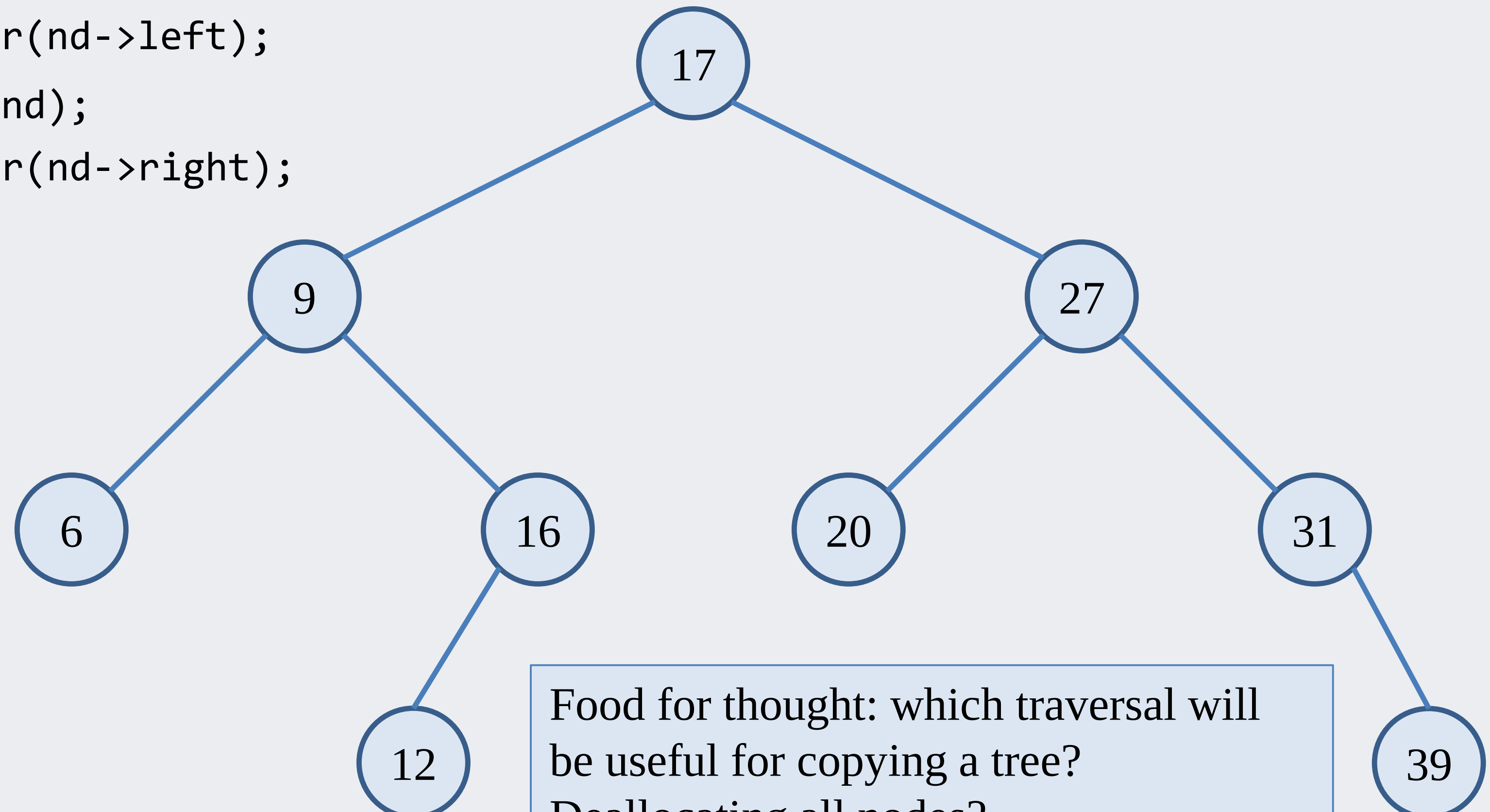
E. 2 9 8 4 5 0 1 6 3 7

$\underbrace{9\ 2\ 4\ 8}_{\text{left}}\ \underbrace{6\ 1\ 0\ 7\ 3}_{\text{right}}\ 5$

Exercise

- What will be printed by an in-order traversal of the tree?
  - preOrder? postOrder?

```
inOrder(nd->left);
visit(nd);
inOrder(nd->right);
```

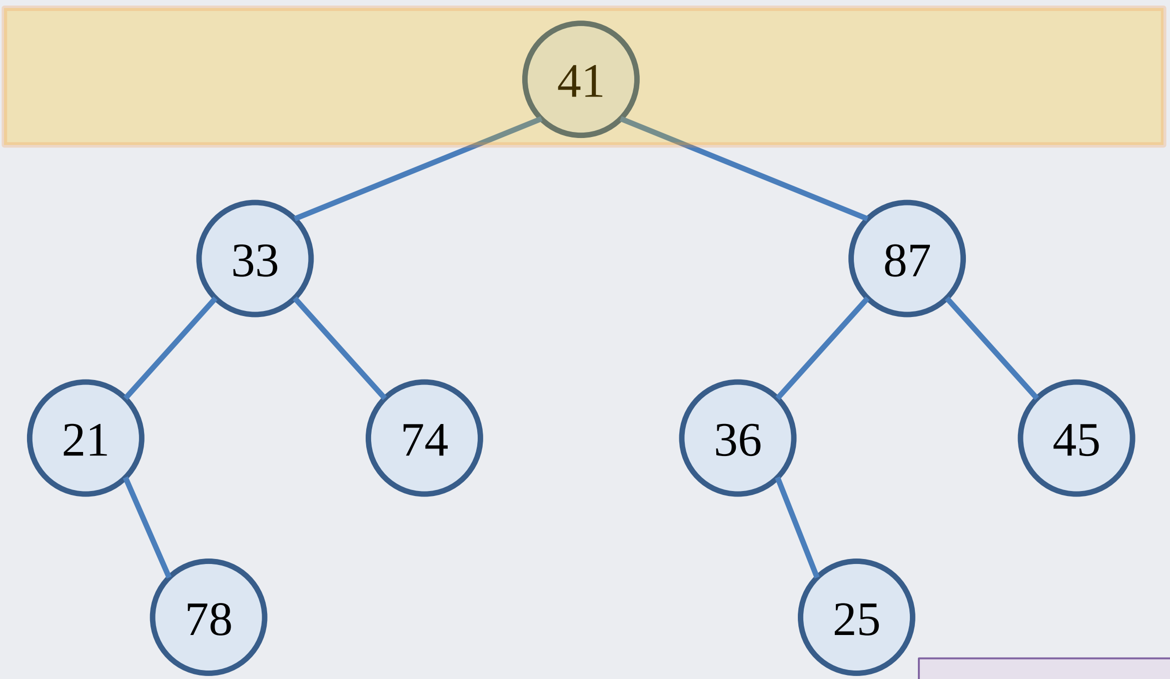


Food for thought: which traversal will be useful for copying a tree? Deallocating all nodes?

Before we move on...

Another type of tree traversal

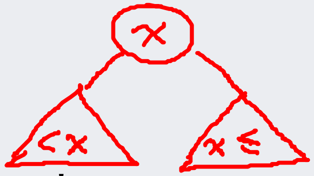
- We have seen pre-order, in-order, post-order traversals
- What about a traversal that visits every node in a level before working on the next level?
  - level-order traversal



Implement using iteration with a Queue ADT

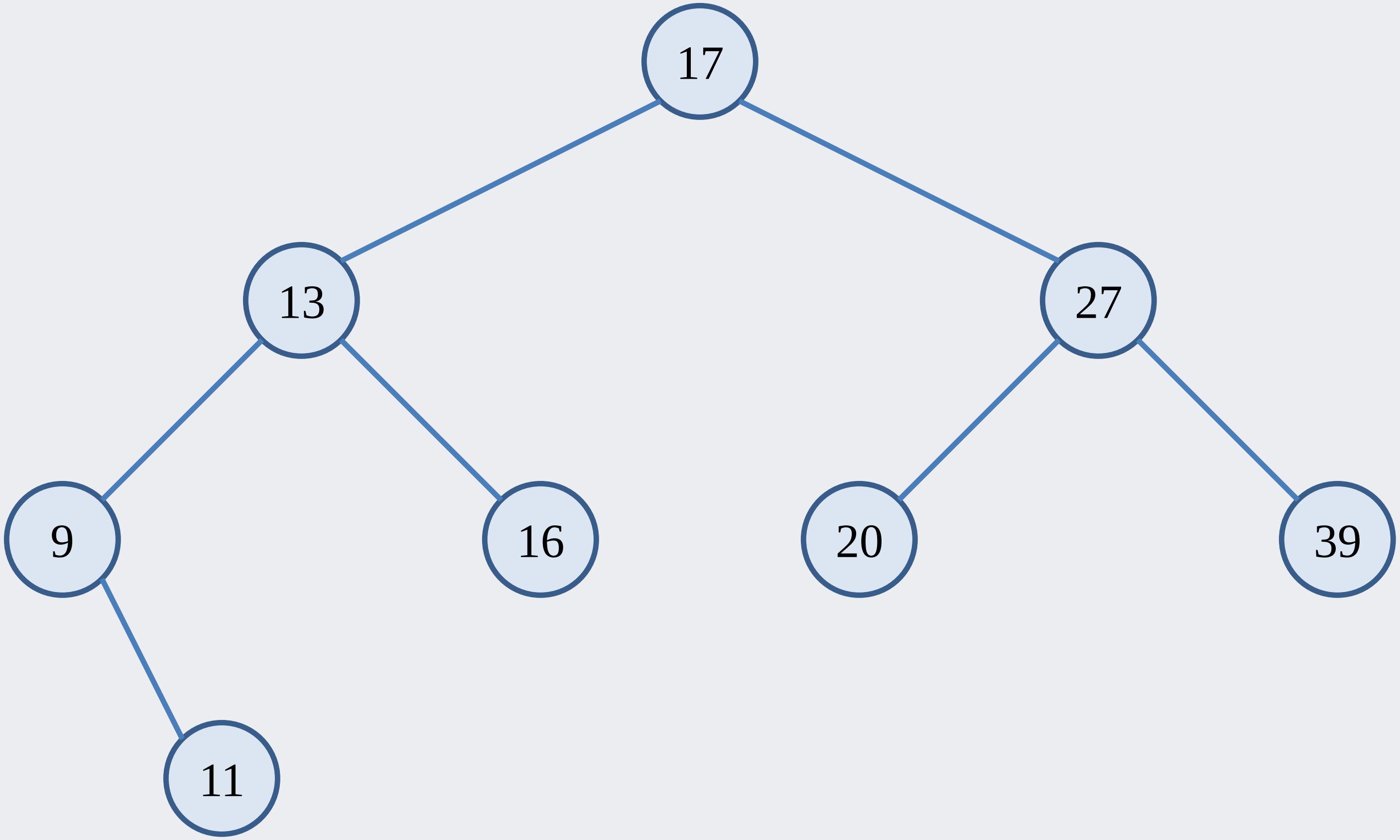
Binary search trees

A data structure for the Dictionary ADT?



- A binary search tree is a binary tree with a special property
  - For all nodes in the tree:
    - All nodes in a left subtree have labels *less* than the label of the subtree's root
    - All nodes in a right subtree have labels *greater* than or equal to the label of the subtree's root
- Binary search trees are *fully ordered*

BST example

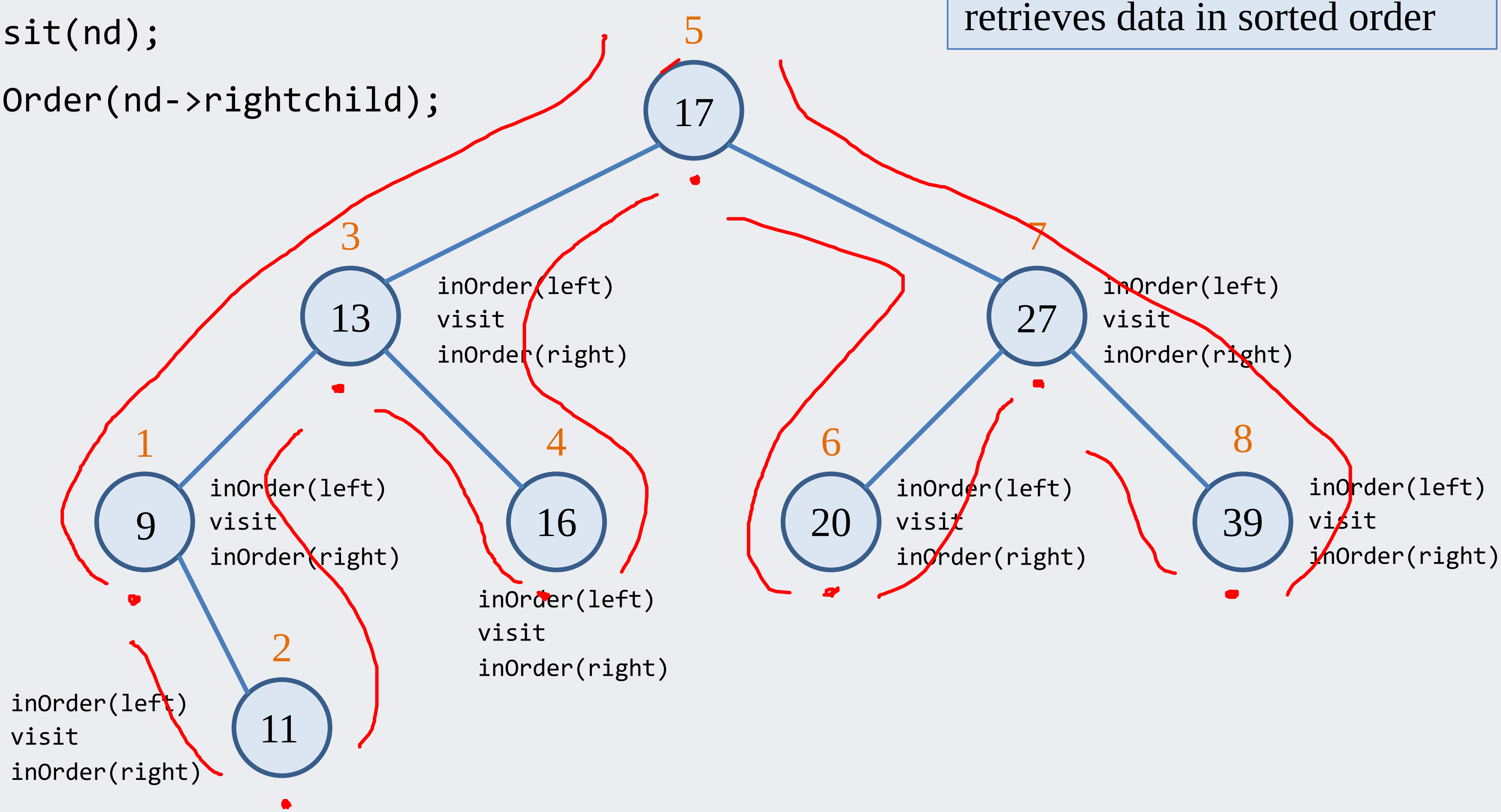


BST inOrder traversal

9 11 13 16 17 20 27 39

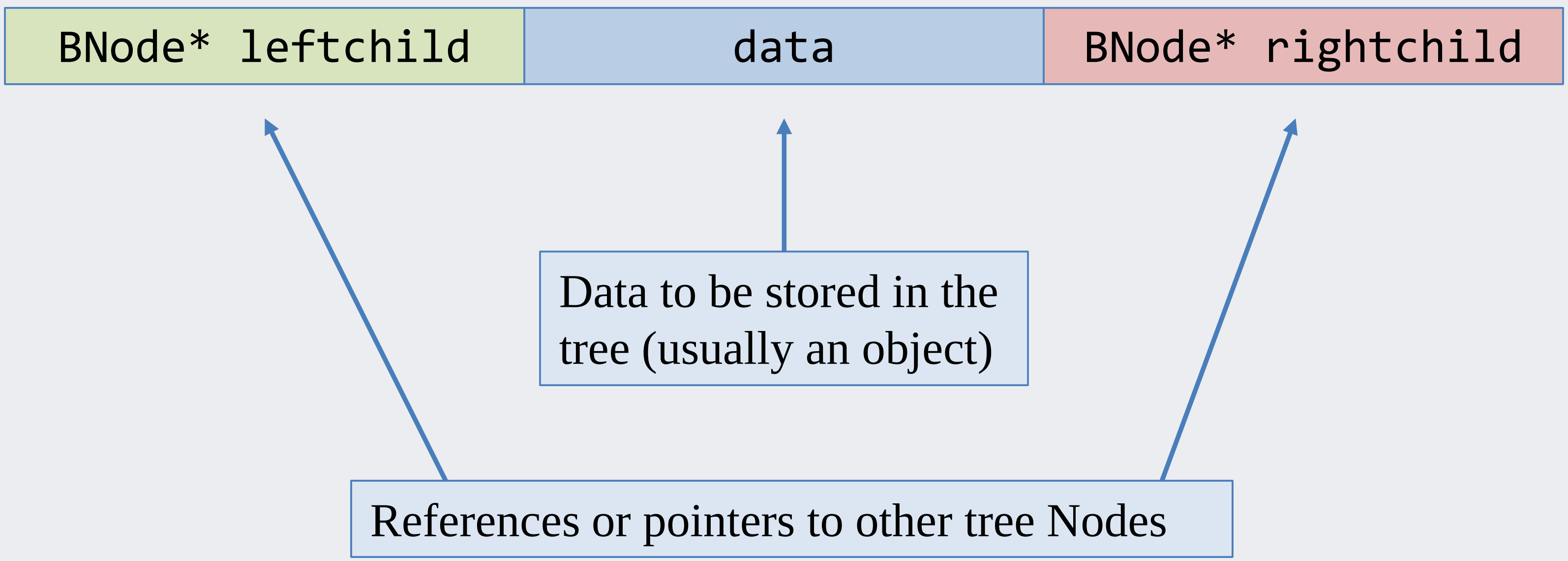
```
inOrder(nd->leftchild);
visit(nd);
inOrder(nd->rightchild);
```

inOrder traversal on a BST retrieves data in sorted order



BST implementation

- Binary search trees can be implemented using a reference structure
- Tree nodes contain data and two pointers to nodes

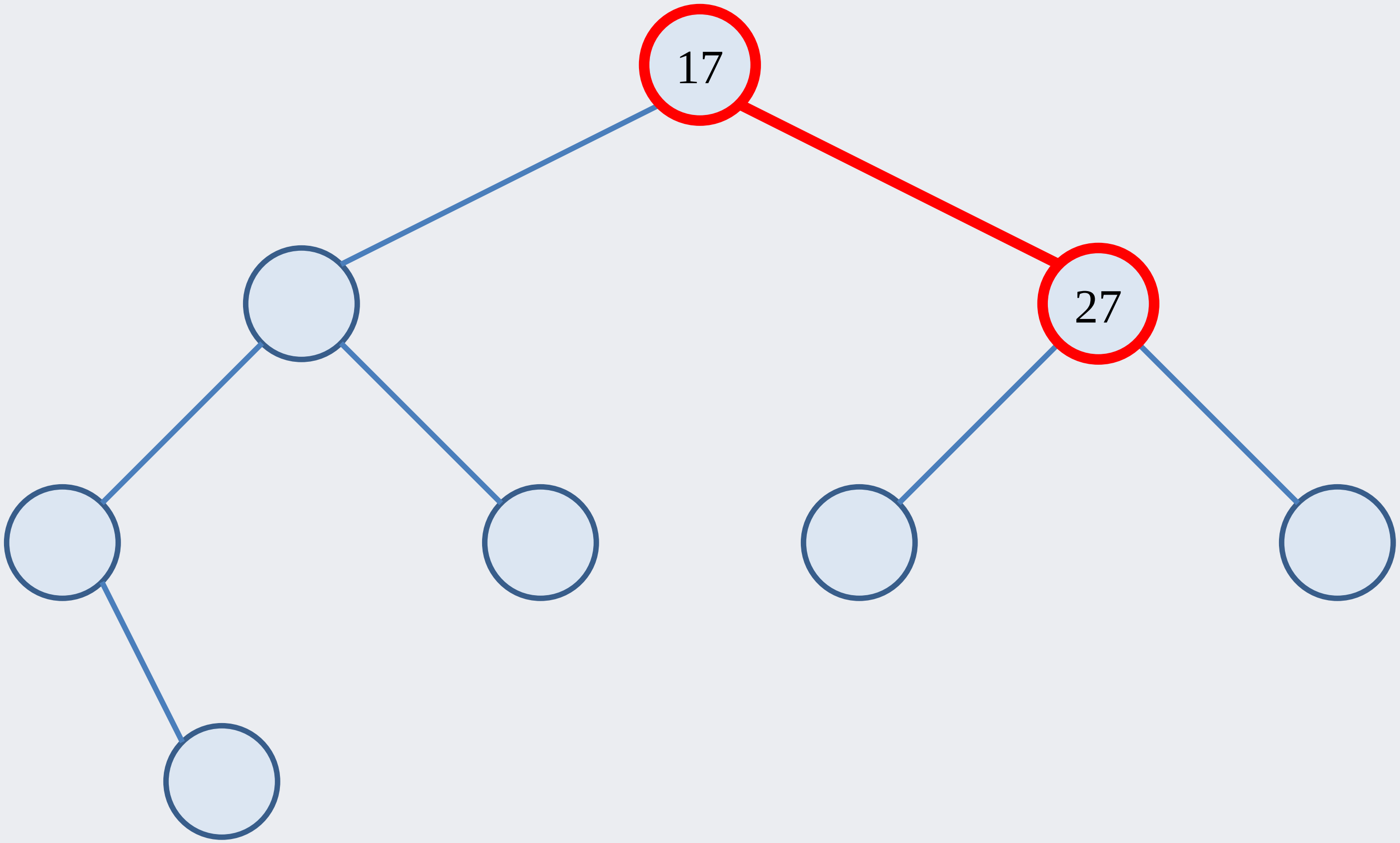


BST search

- To find a value in a BST search from the root node:
  - If the target is less than the value in the node search its left subtree
  - If the target is greater than the value in the node search its right subtree
  - Otherwise return true, (or a pointer to the data, or …)
- How many comparisons?
  - One for each node on the path
  - Worst case: height of the tree + 1

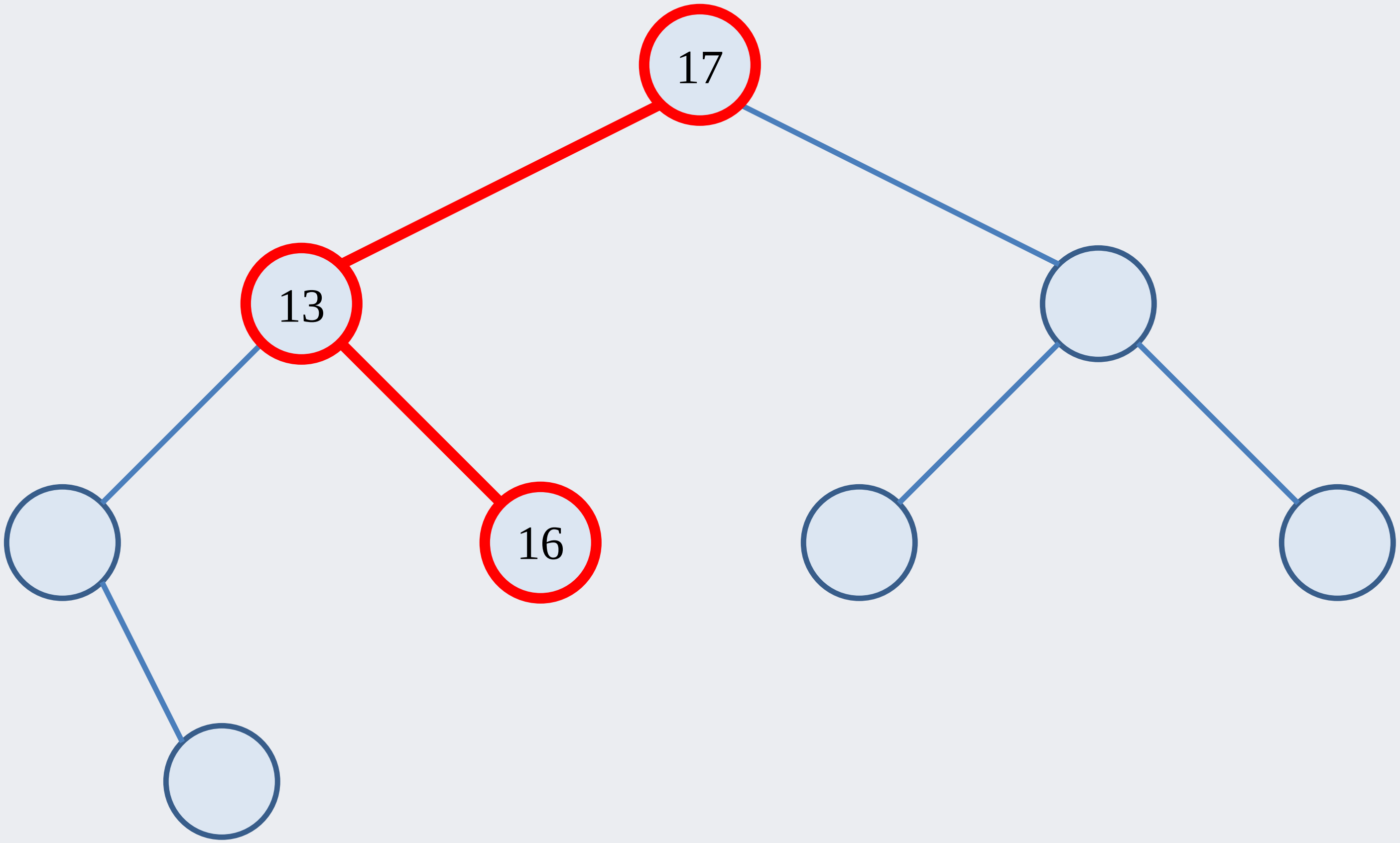
BST search example

```
search(27);
```



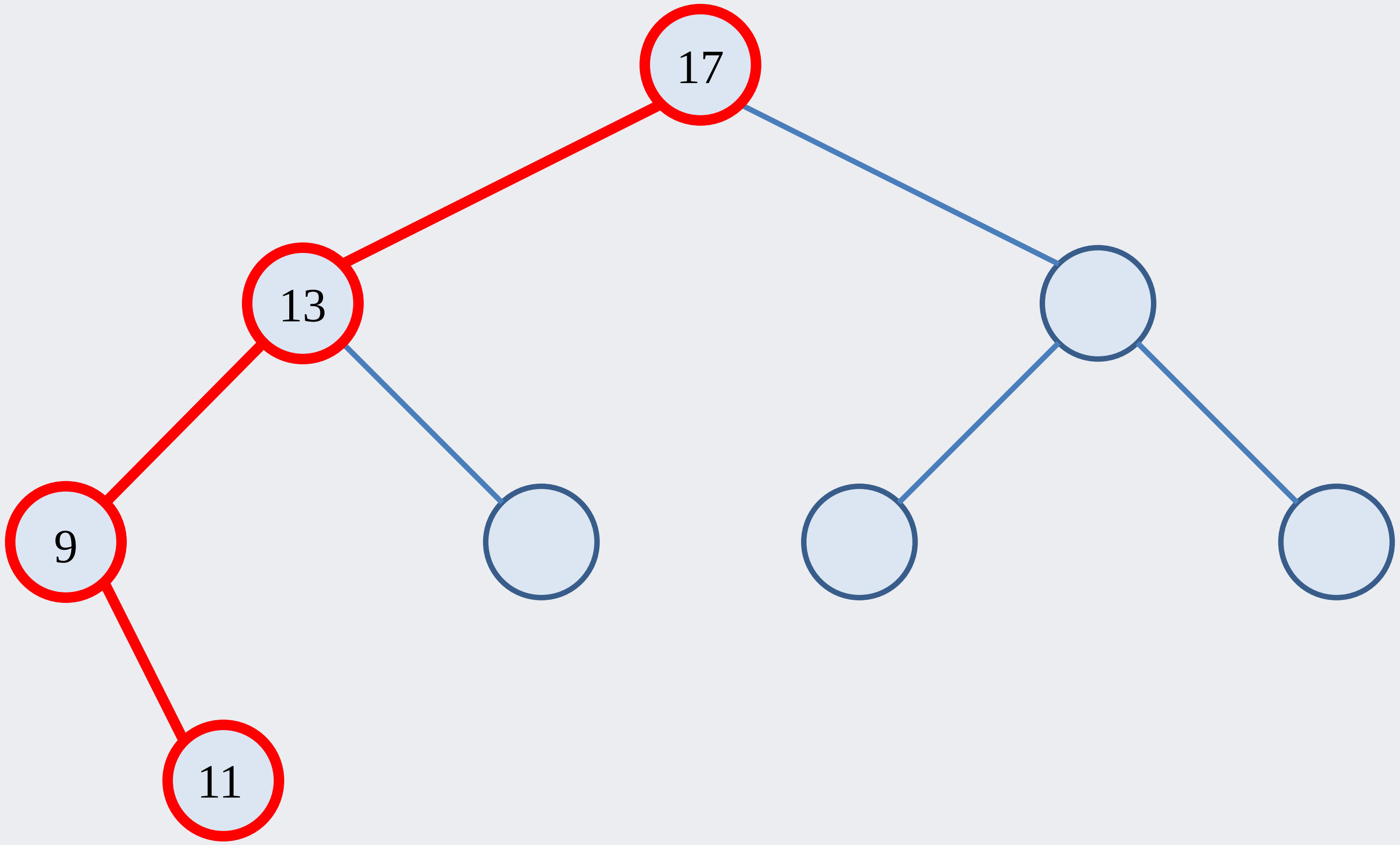
BST search example

```
search(16);
```



BST search example

```
search(12);
```



Search implementation

- Search can be implemented iteratively or recursively

```
int search(BNode* nd, int key) {
  if (nd == NULL) return FALSE;
  else if (nd->data == key) return TRUE;
  else {
    if (key < nd->data)
      return search(nd->left, key);
    else
      return search(nd->right, key);
  }
}
```

BST insertion

- The BST property must hold after insertion
- Therefore the new node must be inserted in the correct position
  - This position is found by performing a search
  - If the search ends at the (null) left child of a node make its left child refer to the new node
  - If the search ends at the right child of a node make its right child refer to the new node
- The cost is about the same as the cost for the search algorithm, O(*height*)

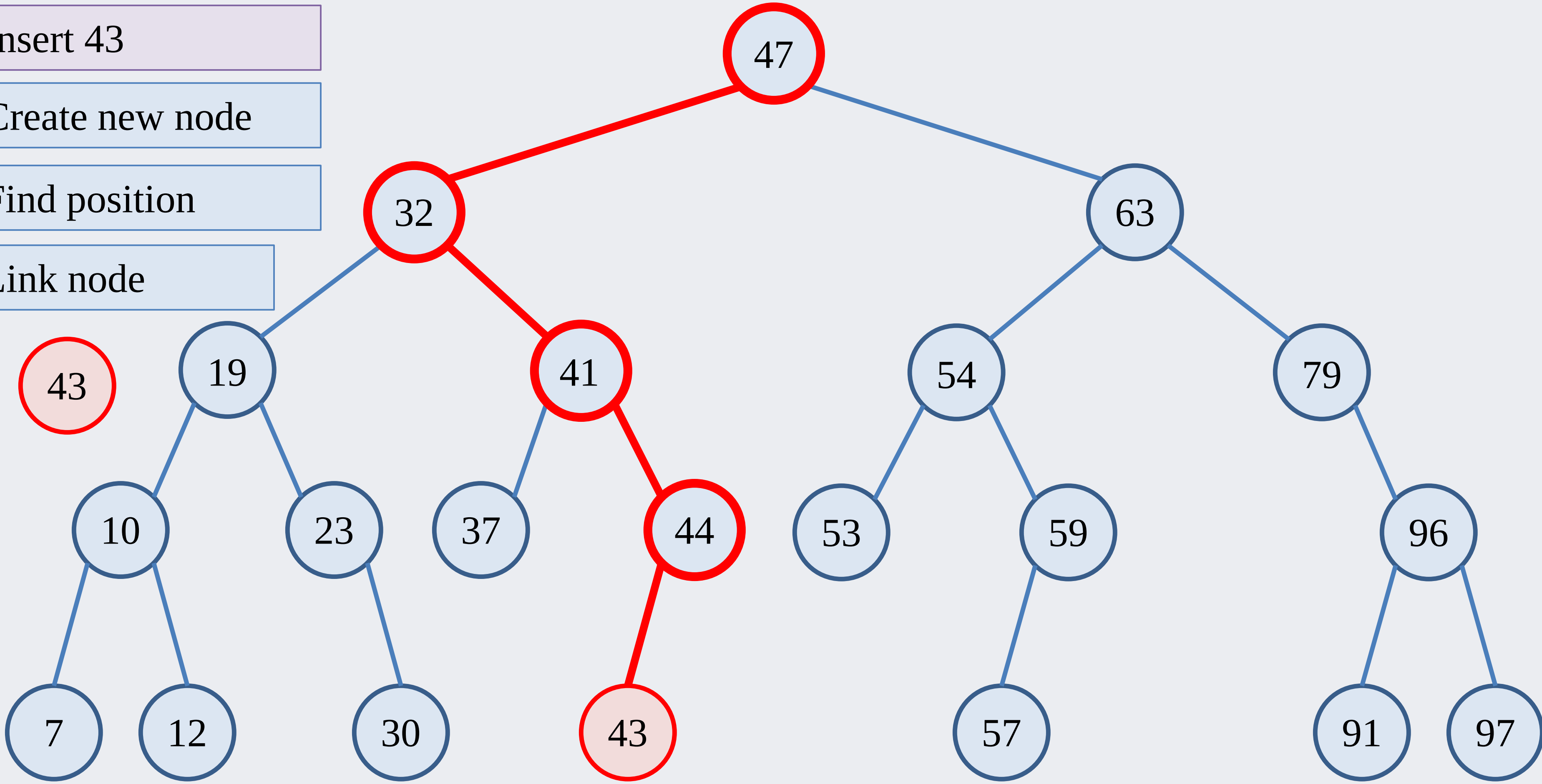
BST insertion example

Insert 43

Create new node

Find position

Link node



BST insertion example

Create new BST

Insert 3

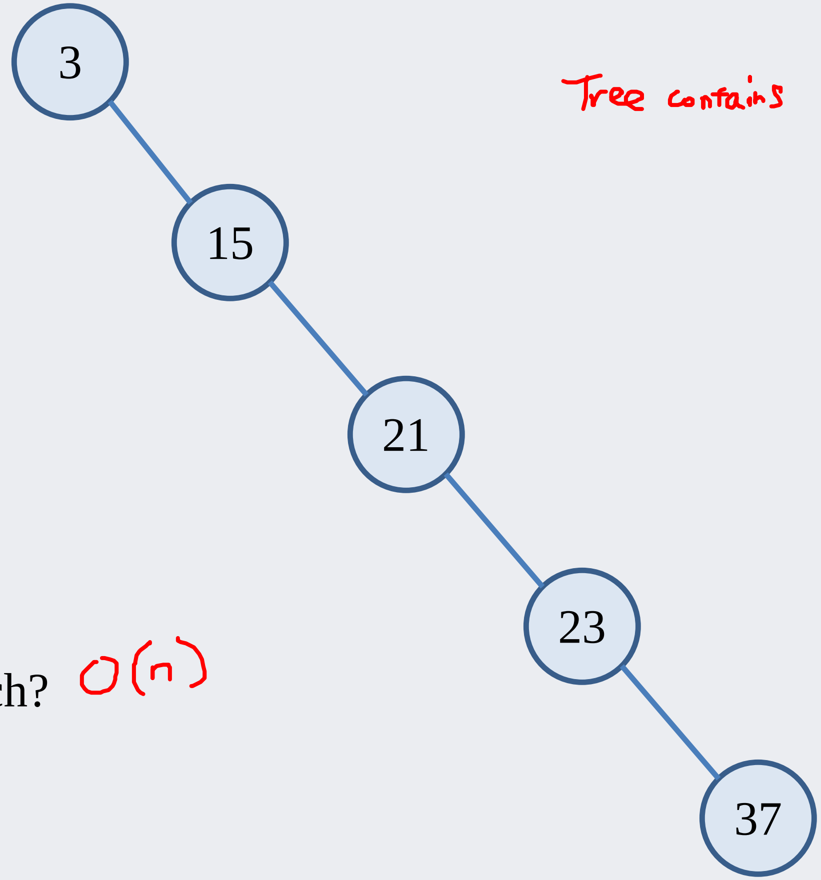
Insert 15

Insert 21

Insert 23

Insert 37

Search 45



Tree contains n keys

How many operations for Search? O(n)

Complexity?

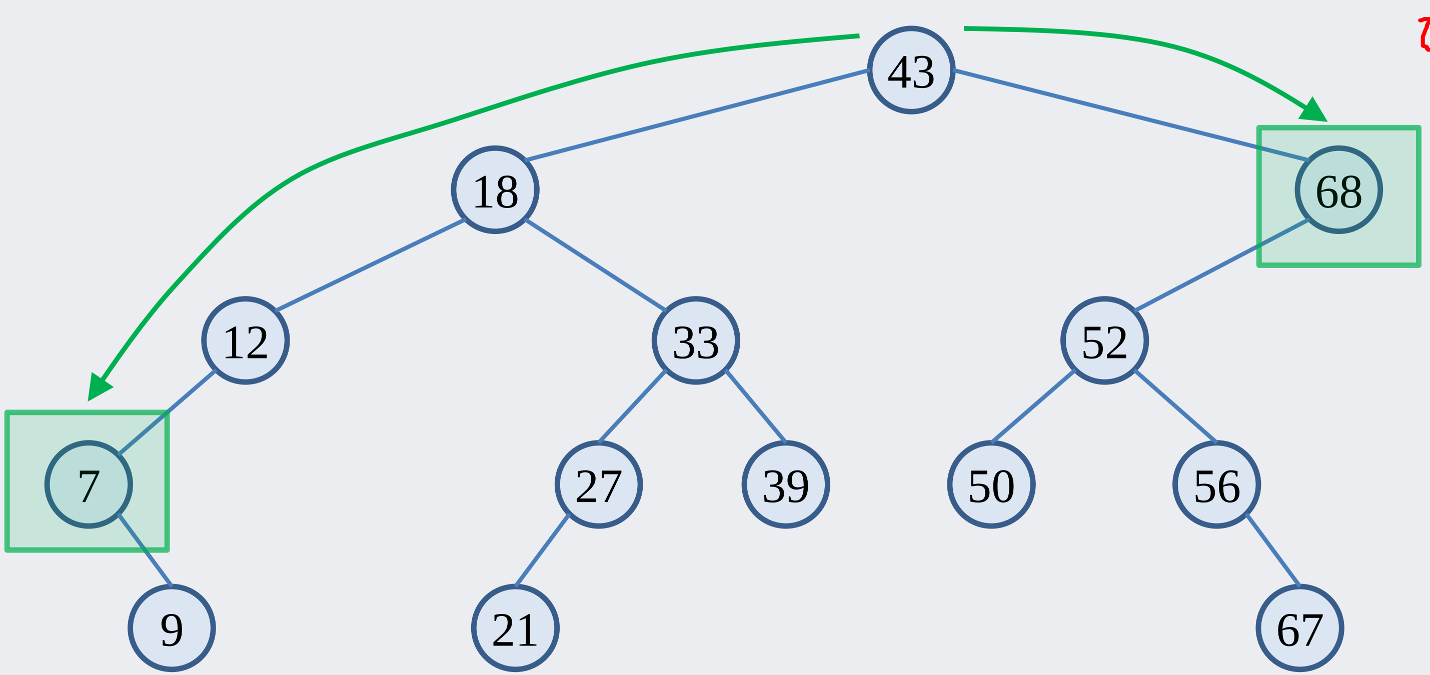
Insert implementation

- Insert can also be implemented iteratively or recursively

```
BNode* insert(BNode* nd, int key) {
  if (nd == NULL) {
    BNode* newnode = (BNode*) malloc(sizeof(BNode));
    newnode->data = key;
    newnode->left = NULL;
    newnode->right = NULL;
    return newnode;
  }
  else {
    if (key < nd->data)
      nd->left = insert(nd->left, key);
    else
      nd->right = insert(nd->right, key);
    return nd;
  }
}
```

Find Min, Find Max

- Find minimum:
  - From the root, keep following left child links until no more left child exists (i.e. NULL)
- Find maximum:
  - From the root, follow right child links until no more right child exists



O(height)

height ∈ O(n)

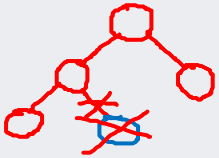
findMin

Iterative implementation

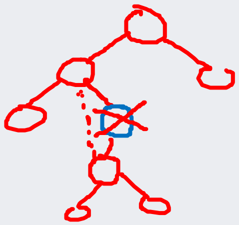
```
int findMin(BNode* nd) {
  BNode* curr = nd;
  if (nd == NULL)
    return -1;
  else {
    while (curr->left != NULL)
      curr = curr->left;
    return curr->data;
  }
}
```

`findMax` is implemented symmetrically

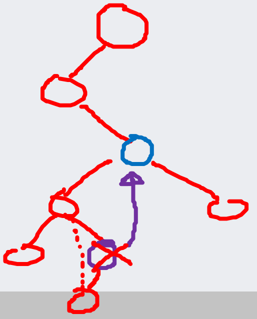
- After removal the BST property must hold
- Removal is not as straightforward as search or insertion
  - With insertion the strategy is to insert a new leaf
  - Which avoids changing the internal structure of the tree
  - This is not possible with removal
    - Since the removed node's position is not chosen by the algorithm
- There are a number of different cases that must be considered
  - The node to be removed has no children
    - Remove it (assigning null to its parent's reference)



  - The node to be removed has one child
    - Replace the node with its subtree



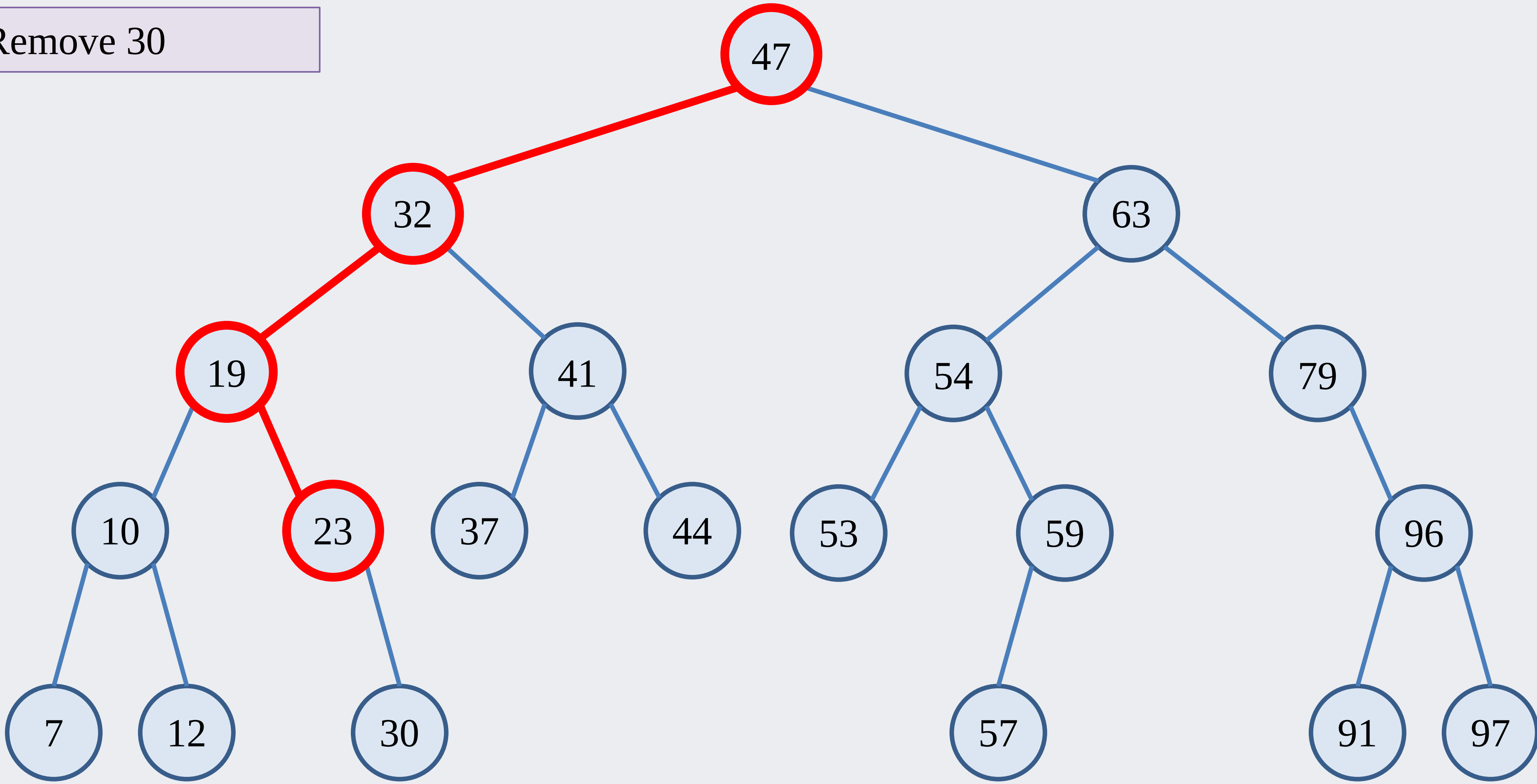
  - The node to be removed has two children
    - …??



BST removal

Target is a leaf node (no children)

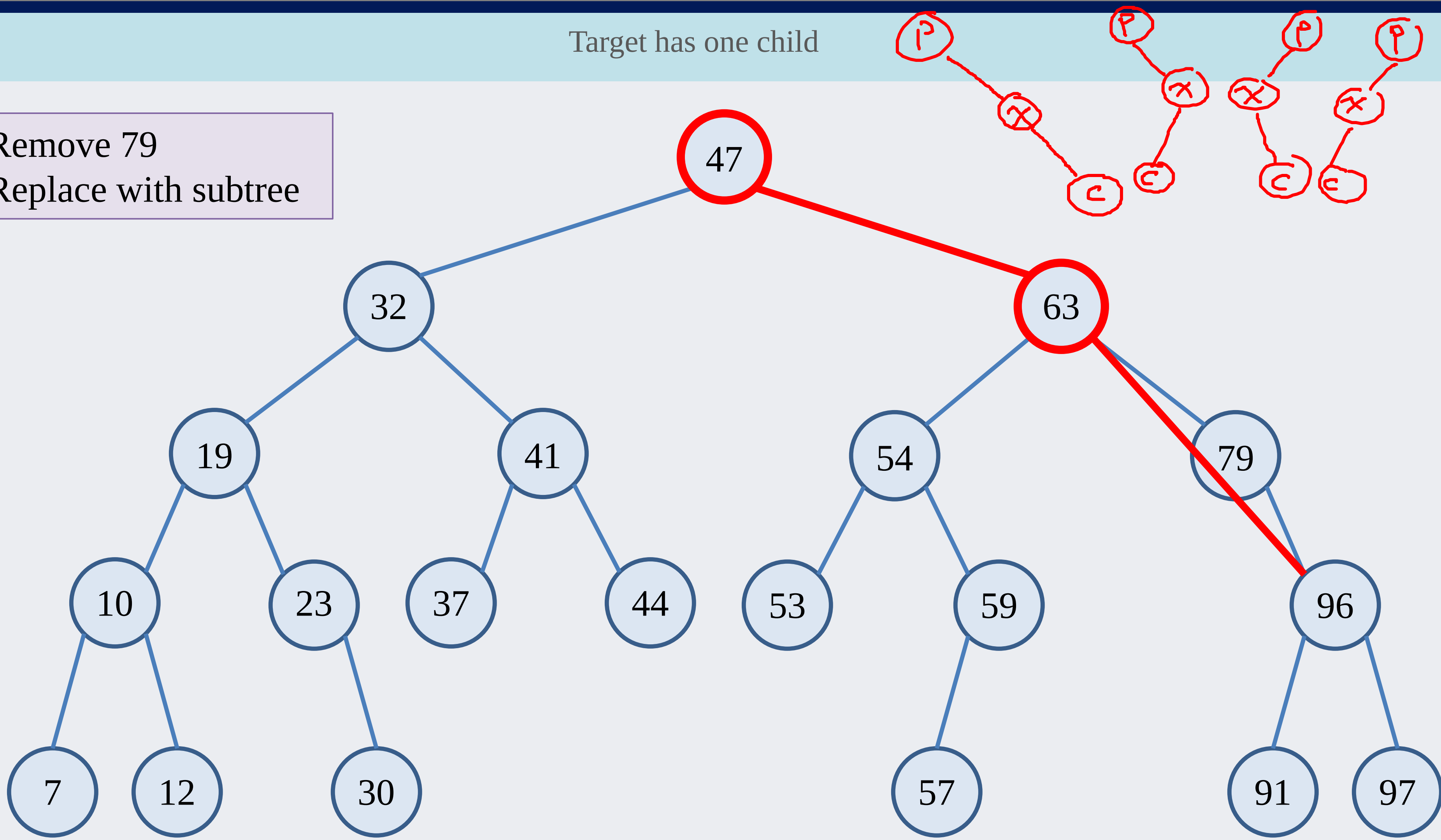
Remove 30



BST removal

Target has one child

Remove 79
Replace with subtree



Looking at the next node

- One of the issues with implementing a BST is the necessity to look at both children
  - And, just like a linked list, *look ahead* for insertion and removal
  - And check that a node is null before accessing its member variables
- Consider removing a node with one child in more detail

Looking ahead

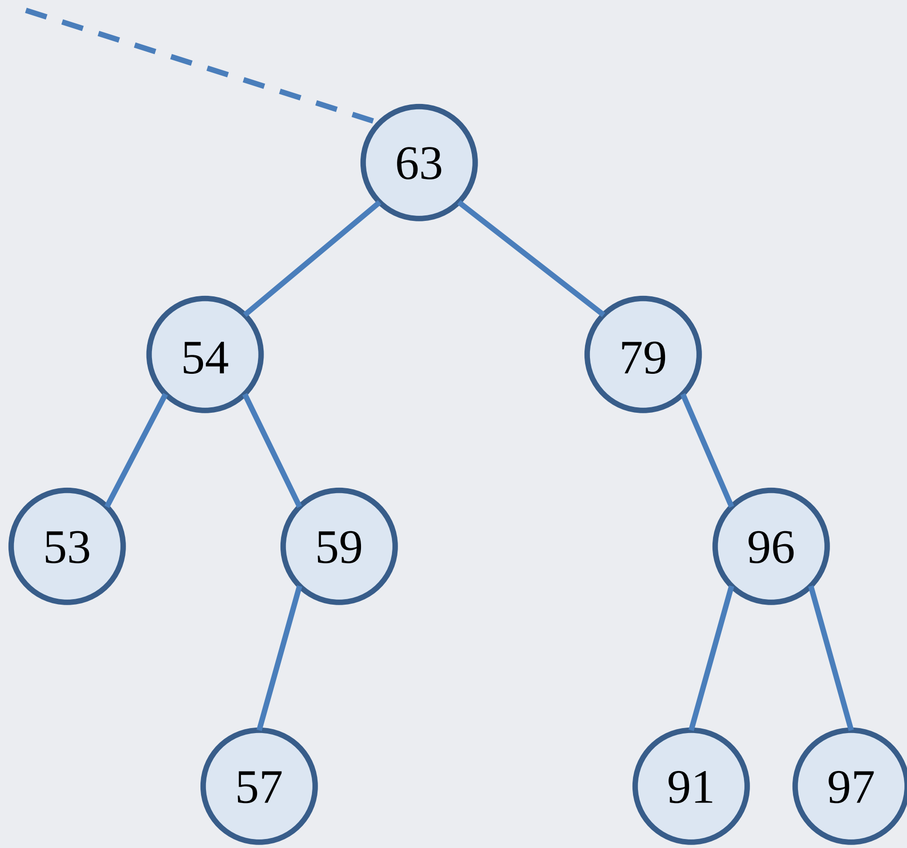
Remove 59

Step 1: We need to find the node to remove and its parent

To make the correct link, we need to know if the node to be removed is a left or right child

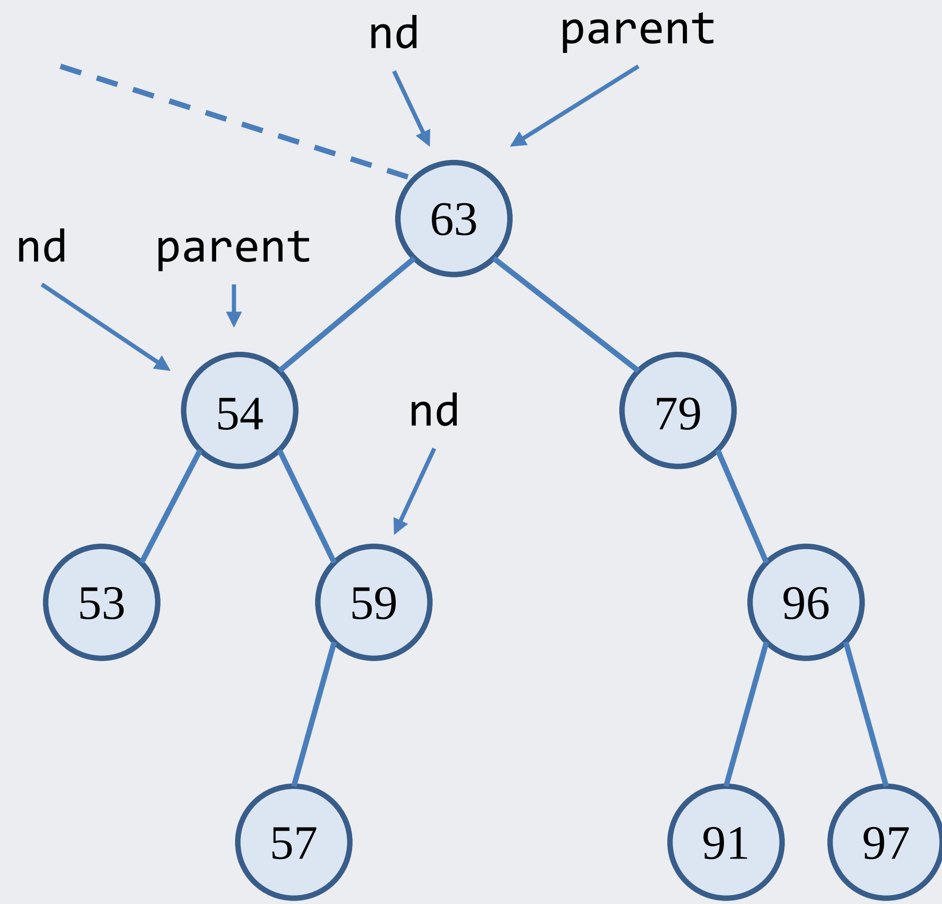
->data

```
while (nd != target) {
  if (nd == NULL)
    return;
  if (target < nd->data) {
    parent = nd;
    nd = nd->left;
    isLeftChild = true;
  }
  else {
    parent = nd;
    nd = nd->right;
    isLeftChild = false;
  }
}
```



Remove 59

```
while (nd != target) {
  if (nd == NULL)
    return;
  if (target < nd->data) {
    parent = nd;
    nd = nd->left;
    isLeftChild = true;
  }
  else {
    parent = nd;
    nd = nd->right;
    isLeftChild = false;
  }
}
```



Now we have enough information to detach 59, after attaching its child to 54.

isLeftChild = false

Removing a node with 2 children

- The most difficult case is when the node to be removed has two children
  - The strategy when the removed node had one child was to replace it with its child
  - But when the node has two children problems arise
- Which child should we replace the node with?
  - We could solve this by just picking one …
- But what if the node we replace it with also has two children?
  - This will cause a problem

Removed node has 2 children

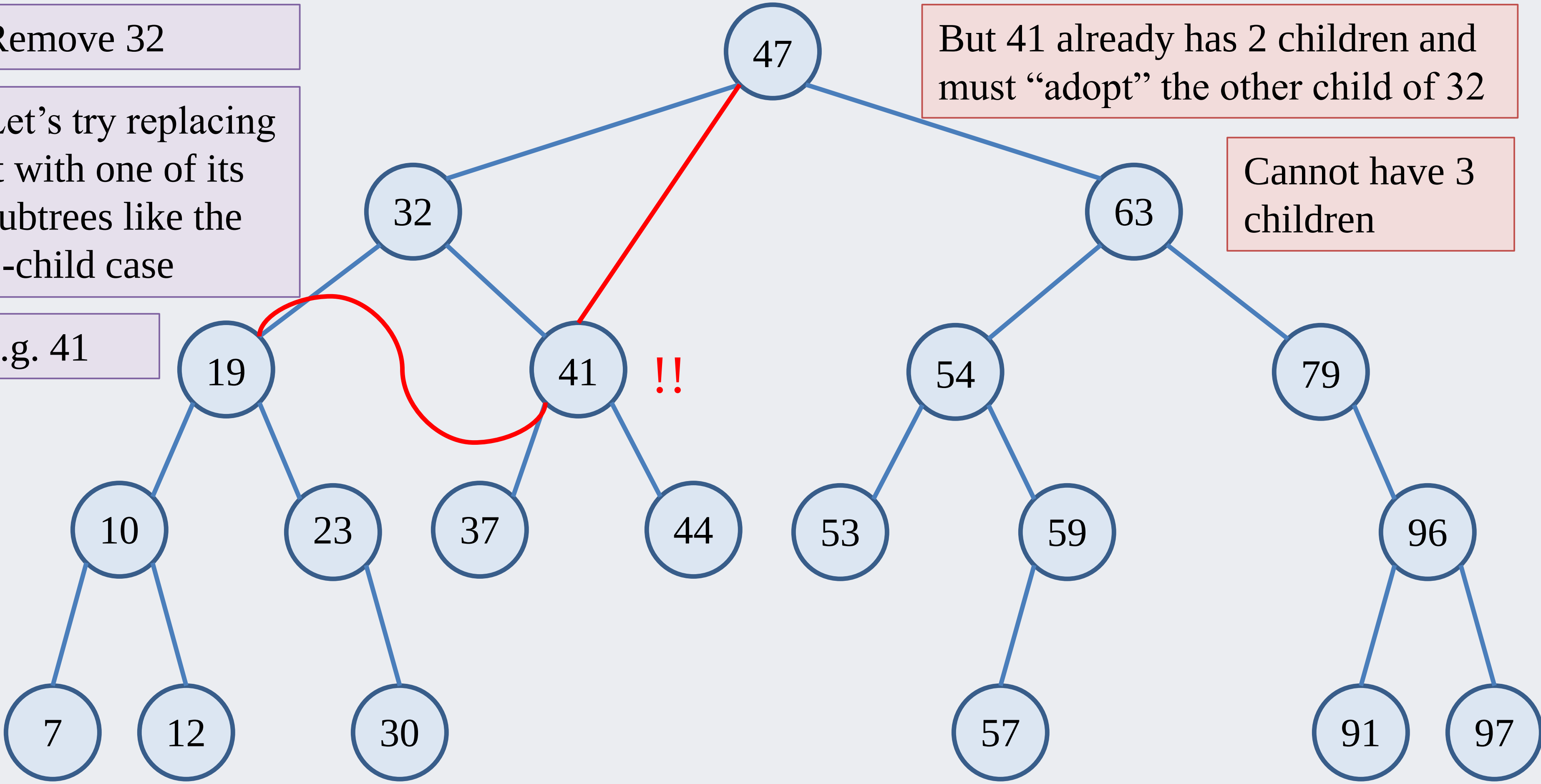
Remove 32

Let's try replacing it with one of its subtrees like the 1-child case

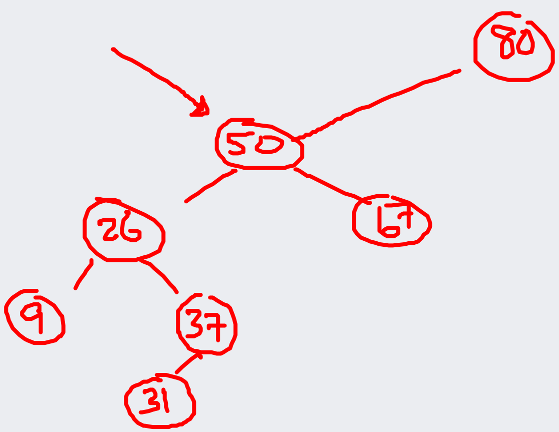
e.g. 41

But 41 already has 2 children and must "adopt" the other child of 32

Cannot have 3 children



- When a node has two children, instead of replacing it with one of its children, find its *predecessor*
  - A node's predecessor is the *right most* node of its *left subtree*
  - The predecessor is the node in the tree with the largest value less than the node's value
- The predecessor cannot have a right child and can therefore have at most one child
  - Why?



In-order traversal:

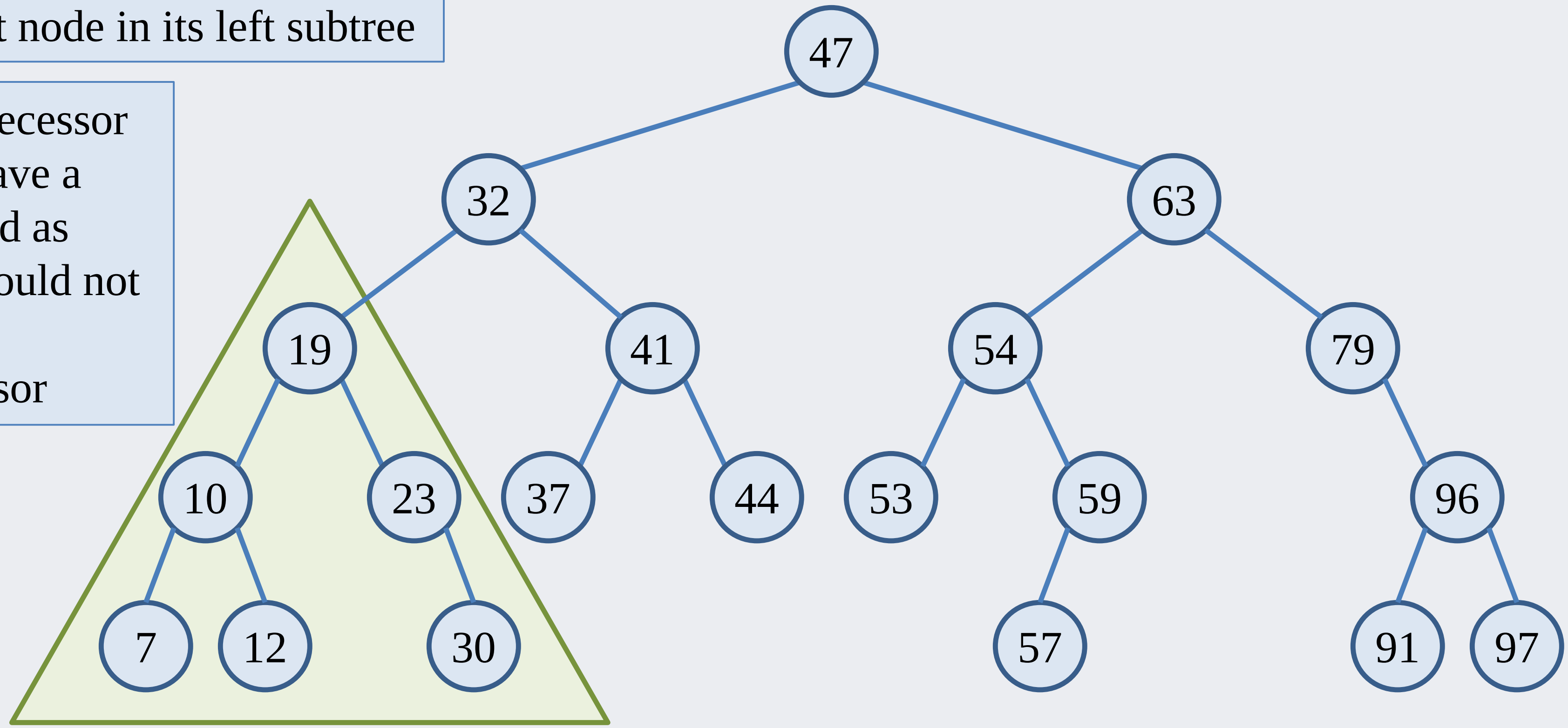
9 26 31 37 50 67 80

Predecessor node

32’s predecessor

The predecessor of 32 is the rightmost node in its left subtree

The predecessor cannot have a right child as then it would not be the predecessor

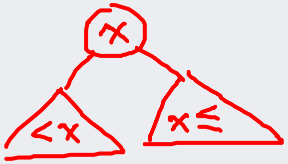


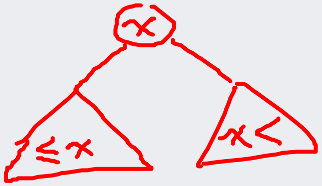
Why use the predecessor?

- The predecessor has some useful properties
  - Because of the BST property it must be the largest value less than its ancestor’s value
    - It is to the right of all of the nodes in its ancestor’s *left* subtree so must be greater than them
    - It is less than the nodes in its ancestor’s *right* subtree
  - It can have at most only one child
- These properties make it a good candidate to replace its ancestor

What about the successor?

- The successor to a node is the left most child of its right subtree
  - It has the smallest value greater than its ancestor's value
  - And cannot have a left child
- The successor can also be used to replace a removed node
  - Pick either one, but be consistent!



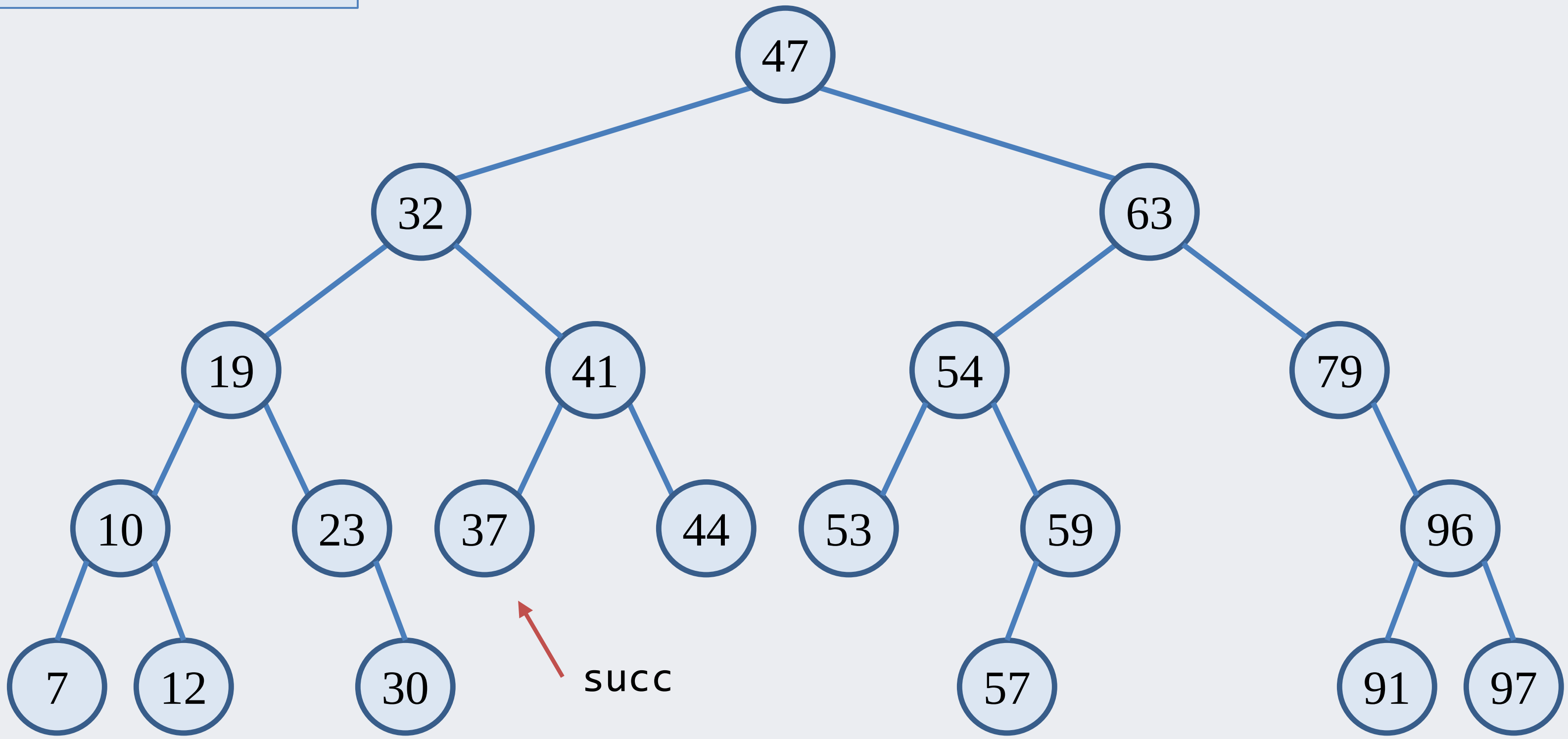


Removed node has 2 children

Replacement with successor

Remove 32

Find successor and replace data



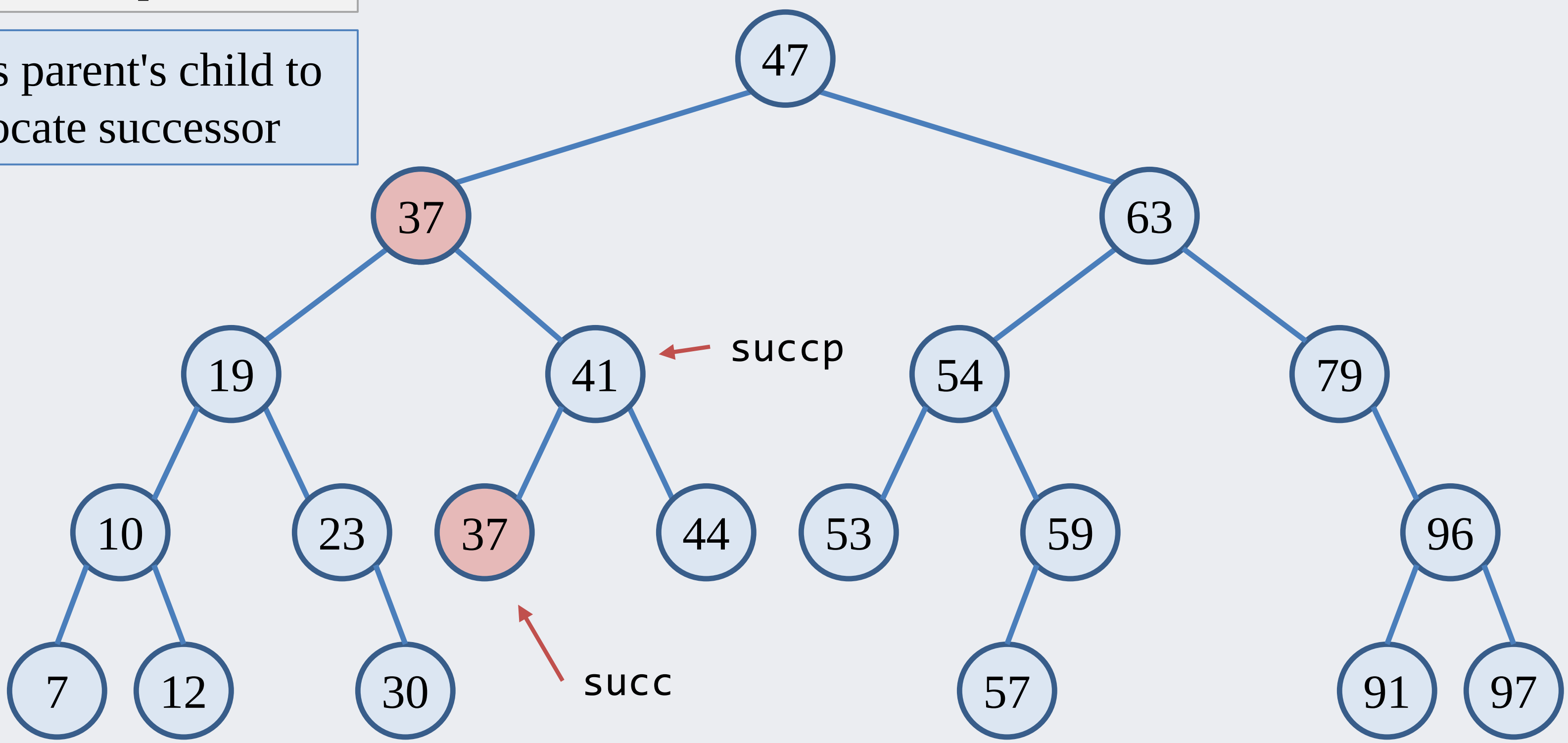
Removed node has 2 children

Replacement with successor

Remove 32

Find successor and replace data

Set successor's parent's child to null and deallocate successor



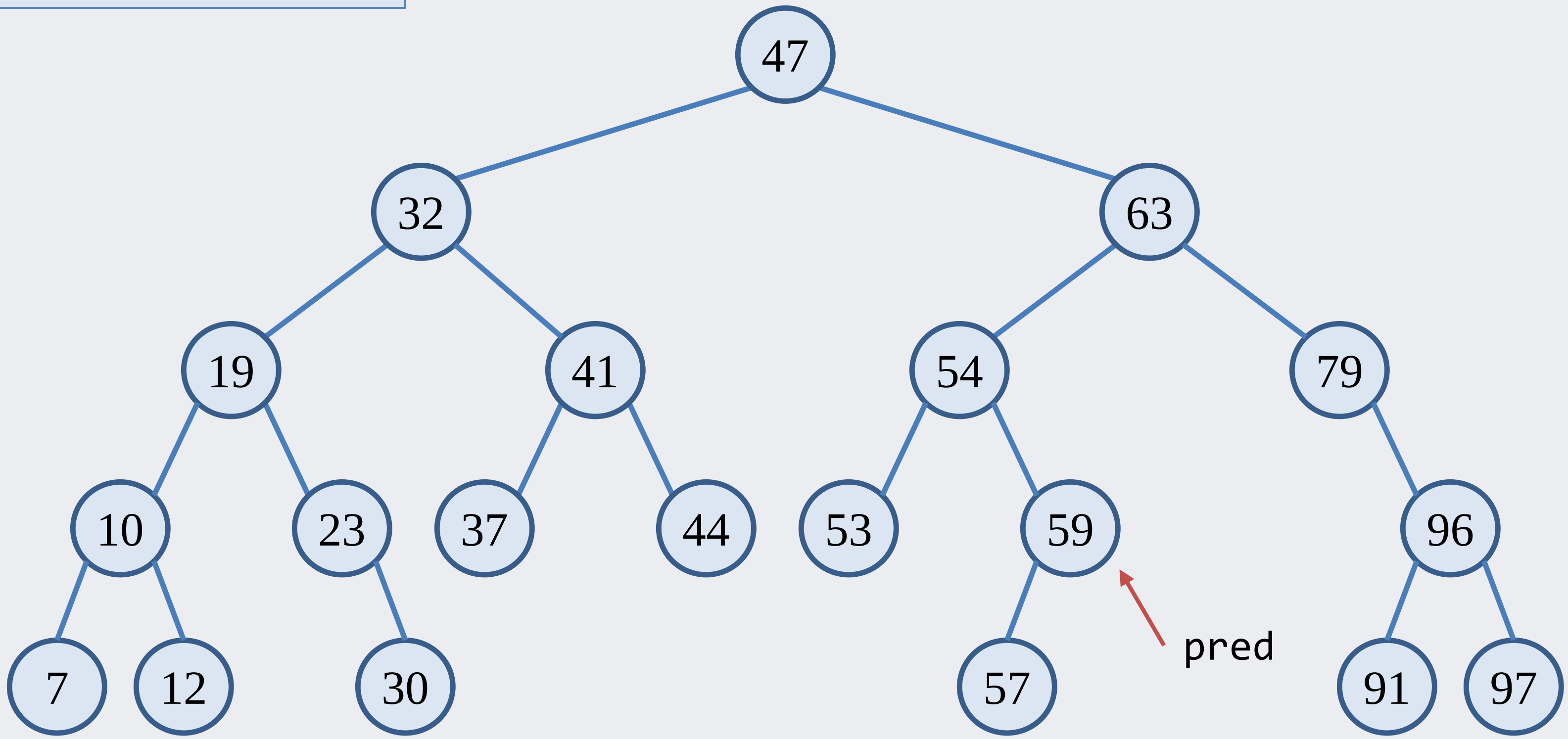
In this example the successor had no subtree

Removed node has 2 children

Replacement with predecessor

Remove 63

Find predecessor and replace data



Removed node has 2 children

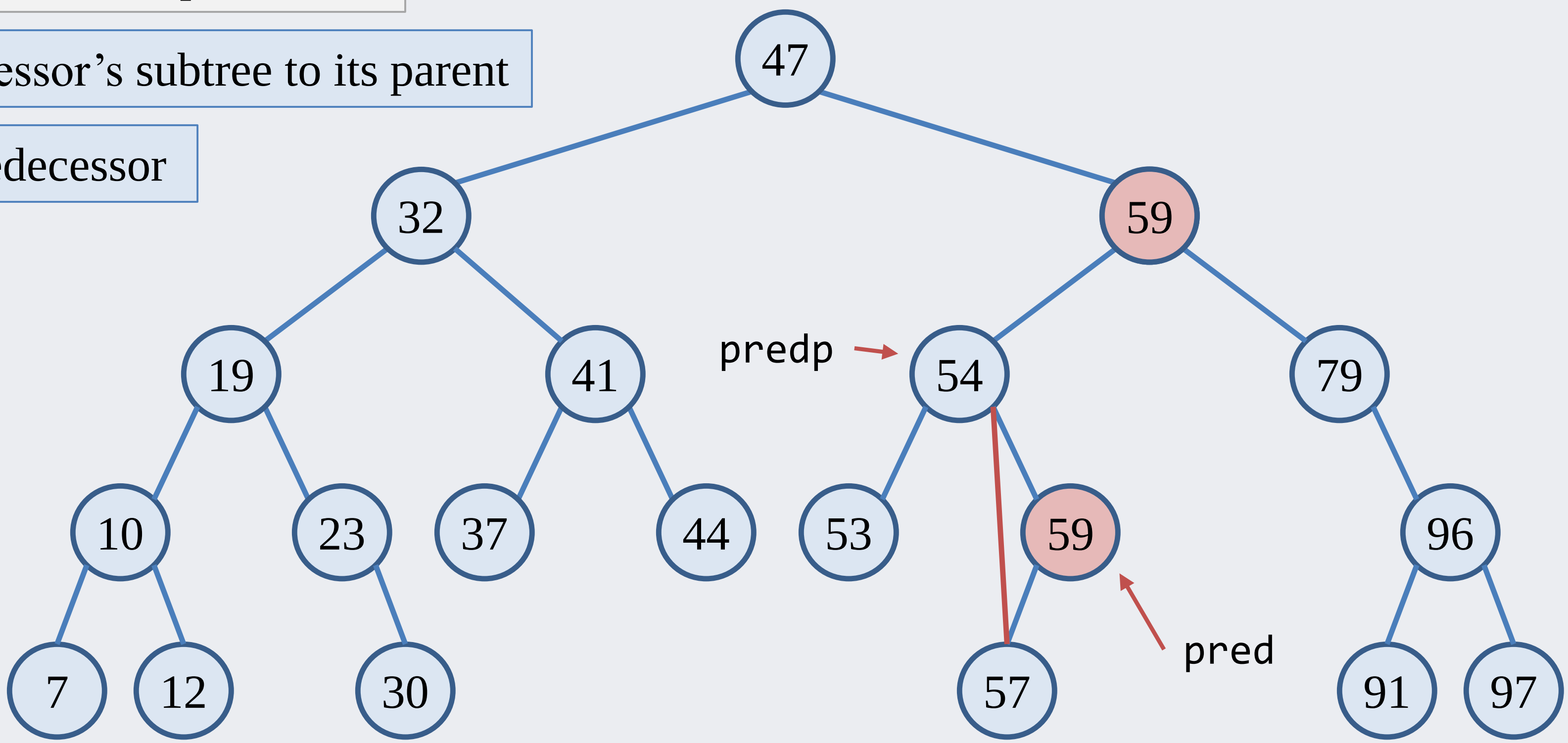
Replacement with predecessor

Remove 63

Find predecessor and replace data

Attach predecessor's subtree to its parent

Deallocate predecessor



BST efficiency

- The efficiency of BST operations depends on the *height* of the tree
  - All three operations (search, insert and delete) are O(height)
  - If the tree is complete the height is $\lceil \log(\cancel{\text{height}}) \rceil$ n
  - What if it isn't complete?

Height of a BST

Insert 7

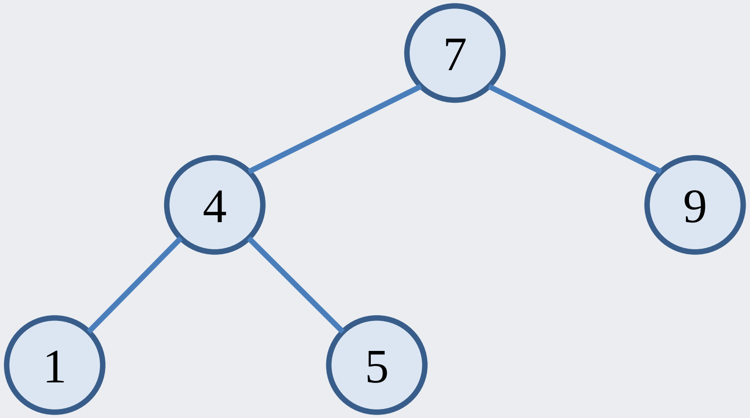
Insert 4

Insert 1

Insert 9

Insert 5

This is a complete BST



$\text{height} = \lfloor \log_2 n \rfloor + 1 = 3$

Height of a BST

Insert 9

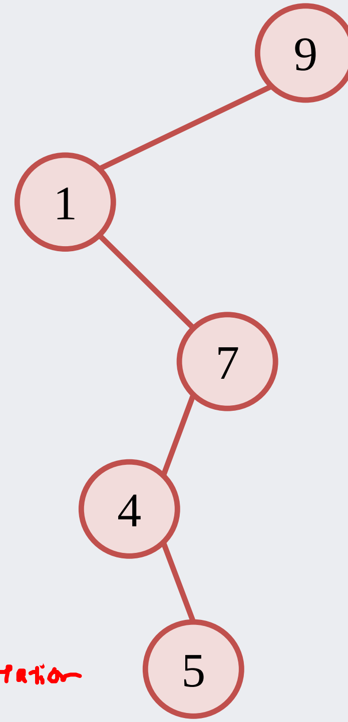
Insert 1

Insert 7

Insert 4

Insert 5

This is a linked list with extra unused pointers



height $= n = 5$

for Dictionary ADT implementation

	worst case
insert	$O(n)$
remove	$O(n)$
search	$O(n)$

for randomly ordered insertions/removals, tree tends to have height $\in O(\log n)$

Balanced BSTs

- It would be ideal if a BST was always close to complete
  - i.e. balanced
- How do we guarantee a balanced BST?
  - We have to make the structure and / or the insertion and removal algorithms more complex
    - e.g. **`red - black`** trees, AVL trees
    - These structures are outside the scope of this course, but you may read Thareja Chapter 10.4 – 10.5 for interest

Readings for this lesson

- Thareja
  - Chapter 9.1 – 9.4
  - Chapter 10.1 – 10.2.9