Python Lists, Tuples, Sets, Dicts

List

General purpose
Most widely used data structure
Grow and shrink size as needed
Sequence type
Sortable

Tuple

Immutable (can’t add/change)
Useful for fixed data
Faster than Lists
Sequence type

Set

Store non-duplicate items
Very fast access vs Lists
Math Set ops (union, intersect)
Unordered

Dict

Key/Value pairs
Associative array, like Java HashMap
Unordered

SEQUENCES (String, List, Tuple)

• indexing:	x[6]
• slicing:	x[1:4]
• adding/concatenating:	+
• multiplying:	*
• checking membership:	in/not in
• iterating	for i in x:
• len(sequence1)	
• min(sequence1)	
• max(sequence1)	
• sum(sequence1[1:3]])	
• sorted(list1)	
• sequence1.count(item)	
• sequence1.index(item)	

- **indexing**
  - Access any item in the sequence using its index

String

```
x = 'frog'
print (x[3])                        # prints 'g'
```

List

```
x = ['pig', 'cow', 'horse']
print (x[1])                        # prints 'cow'
```

- **slicing**
  - Slice out substrings, sublists, subtuples using indexes
  [start : end+1 : step]

```
x = 'computer'
```

Code	Result	Explanation
`x[1:4]`	'omp'	Items 1 to 3
`x[1:6:2]`	'opt'	Items 1, 3, 5
`x[3:]`	'puter'	Items 3 to end
`x[:5]`	'compu'	Items 0 to 4
`x[-1]`	'r'	Last item
`x[-3:]`	'ter'	Last 3 items
`x[:-2]`	'comput'	All except last 2 items

- **adding / concatenating**
  - Combine 2 sequences of the same type using +

String

```
x = 'horse' + 'shoe'
print (x)                    # prints 'horseshoe'
```

List

```
x = ['pig', 'cow'] + ['horse']
print (x)                    # prints ['pig', 'cow', 'horse']
```

- **multiplying**
  - Multiply a sequence using *

String

```
x = 'bug' * 3
print (x)            # prints 'bugbugbug'
```

List

```
x = [8, 5] * 3
print (x)            # prints [8, 5, 8, 5, 8, 5]
```

- **checking membership**
  - Test whether an item is **in** or **not in** a sequence

String

```
x = 'bug'
print ('u' in x)                    # prints True
```

List

```
x = ['pig', 'cow', 'horse']
print ('cow' not in x)              # prints False
```

- **iterating**
  - Iterate through the items in a sequence

Item

```
x = [7, 8, 3]
for item in x:
    print (item * 2)                # prints 14, 16, 6
```

Index & Item

```
x = [7, 8, 3]
for index, item in enumerate(x):
    print (index, item)             # prints 0 7, 1 8, 2 3
```

- **number of items**
  - Count the number of items in a sequence

String

```
x = 'bug'
print (len(x))                      # prints 3
```

List

```
x = ['pig', 'cow', 'horse']
print (len(x))                      # prints 3
```

- **minimum**
  - Find the minimum item in a sequence lexicographically
  - alpha or numeric types, but cannot mix types

String

```
x = 'bug'
print (min(x))                          # prints 'b'
```

List

```
x = ['pig', 'cow', 'horse']
print (min(x))                          # prints 'cow'
```

- # maximum
  - Find the maximum item in a sequence
  - alpha or numeric types, but cannot mix types

String

```
x = 'bug'
print (max(x))                         # prints 'u'
```

List

```
x = ['pig', 'cow', 'horse']
print (max(x))                         # prints 'pig'
```

- **sum**
  - Find the sum of items in a sequence
  - entire sequence must be numeric type

String -> Error

```
x = [5, 7, 'bug`]
print (sum(x))                    # error!
```

List

```
x = [2, 5, 8, 12]
print (sum(x))                    # prints 27
print (sum(x[-2:]))               # prints 20
```

- sorting
  - Returns a new list of items in **sorted** order
  - Does not change the original list

String

```
x = 'bug'
print (sorted(x))             # prints ['b', 'g', 'u']
```

List

```
x = ['pig', 'cow', 'horse']
print (sorted(x))           # prints ['cow', 'horse', 'pig']
```

- **count (item)**
  - Returns count of an item

String

```
x = 'hippo'
print (x.count('p'))                          # prints 2
```

List

```
x = ['pig', 'cow', 'horse', 'cow']
print (x.count('cow'))                        # prints 2
```

- **index (item)**
  - Returns the index of the first occurrence of an item

String

```
x = 'hippo'
print (x.index('p'))                    # prints 2
```

List

```
x = ['pig', 'cow', 'horse', 'cow']
print (x.index('cow'))                  # prints 1
```

- **unpacking**
  - Unpack the n items of a sequence into n variables

```
x = ['pig', 'cow', 'horse']
a, b, c = x                          # now a is 'pig'
                                     # b is 'cow',
                                     # c is 'horse'
```

Note:
The number of variables must exactly match the length of the list.

LISTS

All operations from Sequences, plus:

- constructors:
- del list1[2] delete item from list1
- list1.append(item) appends an item to list1
- list1.extend(sequence1) appends a sequence to list1
- list1.insert(index, item) inserts item at index
- list1.pop() pops last item
- list1.remove(item) removes first instance of item
- list1.reverse() reverses list order
- list1.sort() sorts list in place

- **constructors - creating a new list**

```
x = list()
x = ['a', 25, 'dog', 8.43]
x = list(tuple1)
```

List Comprehension:

```
x = [m for m in range(8)]
```

resulting list: `[0, 1, 2, 3, 4, 5, 6, 7]`

```
x = [z**2 for z in range(10) if z>4]
```

resulting list: `[25, 36, 49, 64, 81]`

- **delete**
  – Delete a list or an item from a list

```
x = [5, 3, 8, 6]
del(x[1])                    # [5, 8, 6]

del(x)                       # deletes list x
```

- **append**
  - Append an item to a list

```
x = [5, 3, 8, 6]
x.append(7)                    # [5, 3, 8, 6, 7]
```

- **extend**
  - Append an sequence to a list

```
x = [5, 3, 8, 6]
y = [12, 13]
x.extend(y)                  # [5, 3, 8, 6, 7, 12, 13]
```

- **insert**
  - Insert an item at given index `x.insert(index, item)`

```
x = [5, 3, 8, 6]
x.insert(1, 7)           # [5, 7, 3, 8, 6]

x.insert(1,['a','m'])   # [5, ['a', 'm'], 7, 3, 8, 6]
```

- **pop**
  - Pops last item off the list, and returns item

```
x = [5, 3, 8, 6]
x.pop()                  # [5, 3, 8]
                         # and returns the 6


print(x.pop())           # prints 8
                         # x is now [5, 3]
```

- **remove**
  - Remove first instance of an item

```
x = [5, 3, 8, 6, 3]
x.remove(3)                 # [5, 8, 6, 3]
```

- **reverse**
  - Reverse the order of the list

```
x = [5, 3, 8, 6]
x.reverse()                    # [6, 8, 3, 5]
```

- **sort**
  - Sort the list in place

```
x = [5, 3, 8, 6]
x.sort()                  # [3, 5, 6, 8]
```

Note:

sorted(x) returns a *new* sorted list without changing the original list x.

x.sort() puts the items of x in sorted order (sorts in place).

TUPLES

- Support all operations for Sequences
- Immutable, but member objects may be mutable
- If the contents of a list shouldn’t change, use a tuple to prevent items from accidently being added, changed or deleted
- Tuples are more efficient than lists due to Python’s implementation

- **constructors - creating a new tuple**

```
x = ()                # no-item tuple
x = (1,2,3)
x = 1, 2, 3           # parenthesis are optional
x = 2,                # single-item tuple
x = tuple(list1)      # tuple from list
```

- **immutable**
  - But member objects may be mutable

```
x = (1, 2, 3)
del(x[1])                    # error!
x[1] = 8                     # error!

x = ([1,2], 3)               # 2-item tuple: list and int
del(x[0][1])                 # ([1], 3)
```

- **constructors - creating a new set**

```
x = {3,5,3,5}                 # {5, 3}
x = set()                     # empty set
x = set(list1)                # new set from list
                              # strips duplicates
```

Set Comprehension:

```
x = {3*x for x in range(10) if x>5}
```

resulting set: `{18, 21, 24, 27} but in random order`

- basic set operations

Description	Code
Add item to set x	`x.add(item)`
Remove item from set x	`x.remove(item)`
Get length of set x	`len(x)`
Check membership in x	`item in x` `item not in x`
Pop random item from set x	`x.pop()`
Delete all items from set x	`x.clear()`

- standard mathematical set operations

Set Function	Description	Code
Intersection	AND	`set1 & set2`
Union	OR	`set1 | set2`
Symmetric Difference	XOR	`set1 ^ set2`
Difference	In set1 but not in set2	`set1 - set2`
Subset	set2 contains set1	`set1 <= set2`
Superset	set1 contains set2	`set1 >= set2`

- **constructors - creating a new dict**

```
x = {'pork':25.3, 'beef':33.8, 'chicken':22.7}
x = dict([('pork', 25.3),('beef', 33.8),('chicken', 22.7)])
x = dict(pork=25.3, beef=33.8, chicken=22.7)
```

- basic dict operations

Description	Code
Add or change item in dict x	`x['beef'] = 25.2`
Remove item from dict x	`del x['beef']`
Get length of dict x	`len(x)`
Check membership in x (only looks in keys, not values)	`item in x` `item not in x`
Delete all items from dict x	`x.clear()`
Delete dict x	`del x`

- **accessing keys and values in a dict**

```
x.keys()       # returns list of keys in x
x.values()     # returns list of values in x
x.items()      # returns list of key-value tuple pairs in x

item in x.values()   # tests membership in x: returns boolean
```

- **iterating a dict**

```
for key in x:                       # iterate keys
    print(key, x[key])              # print all key/value pairs

for k, v in x.items():              # iterate key/value pairs
    print(k, v)                     # print all key/value pairs
```

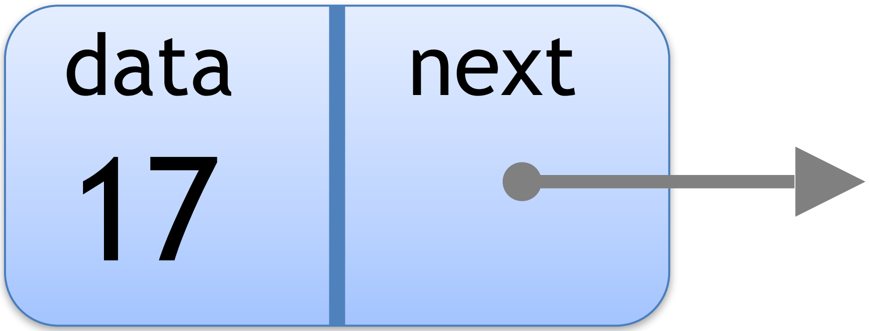
Note:

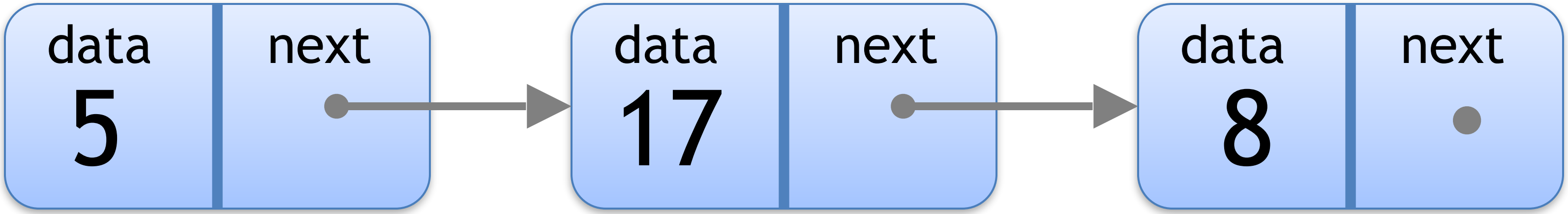
Entries in a dict are in random order.

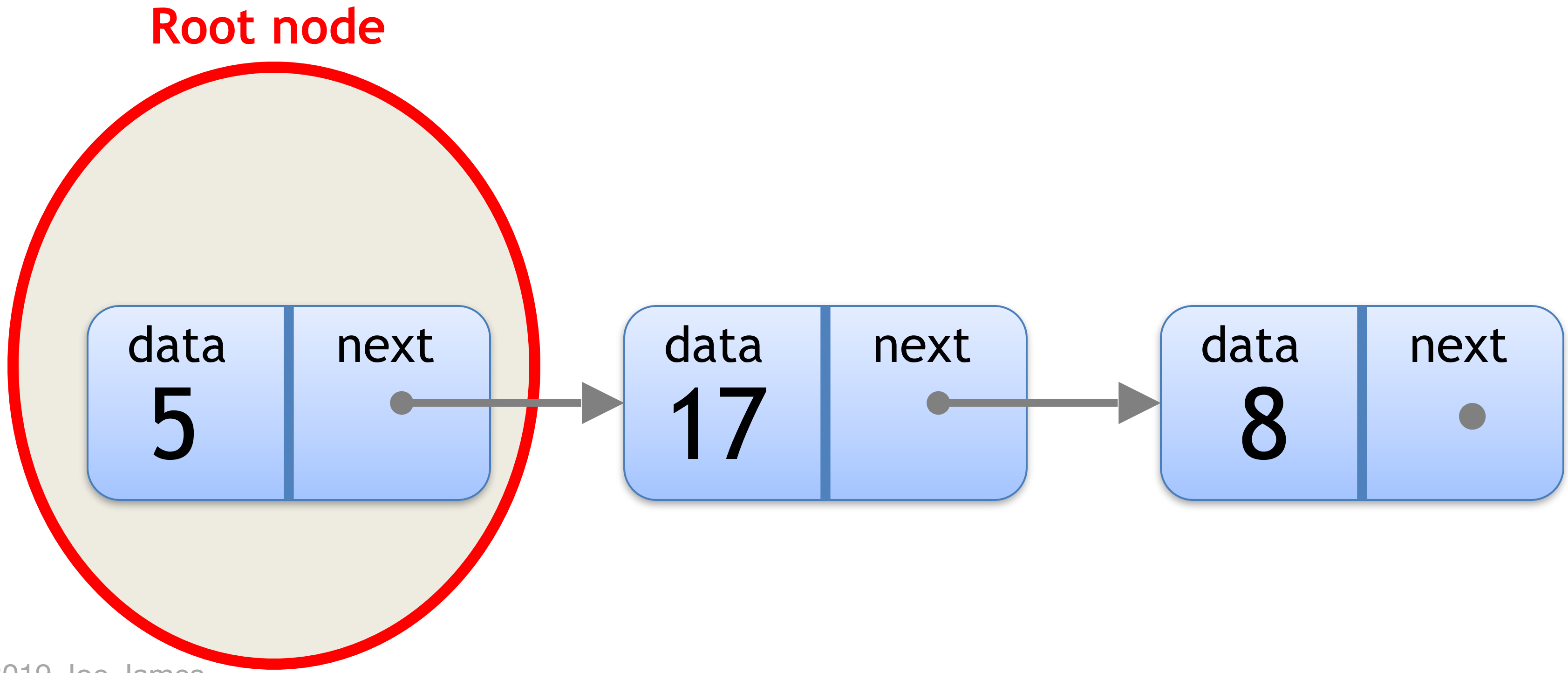
Python Linked Lists

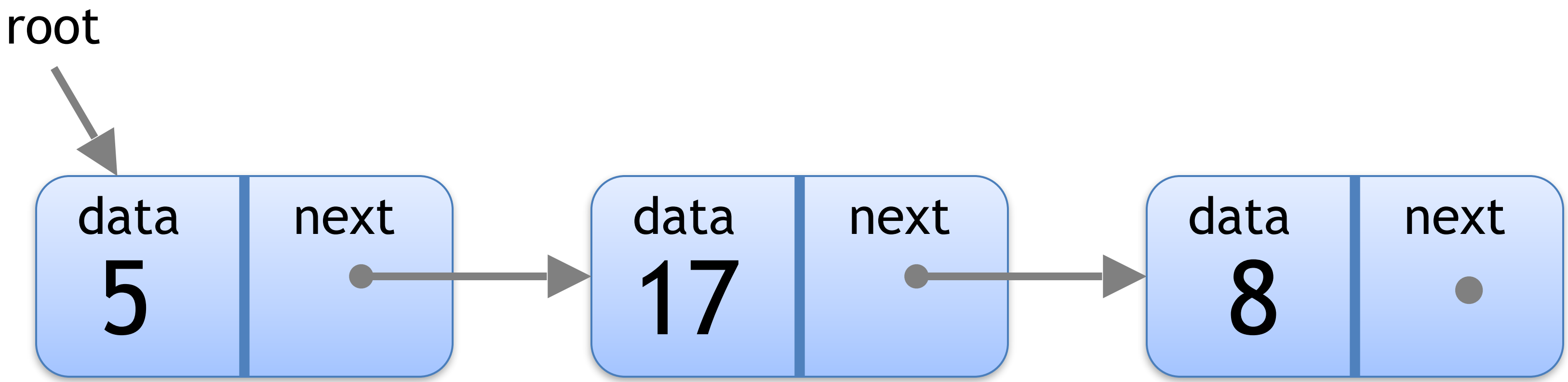
Every Node has 2 parts:

data and a pointer to the **next** Node




data
5
next
data
17
next
data
8
next


Root node
data
5
next
data
17
next
data
8
next


root
data
5
next
data
17
next
data
8
next

Linked Lists

Attributes:

root - pointer to the beginning of the List

size - number of nodes in List

Operations:

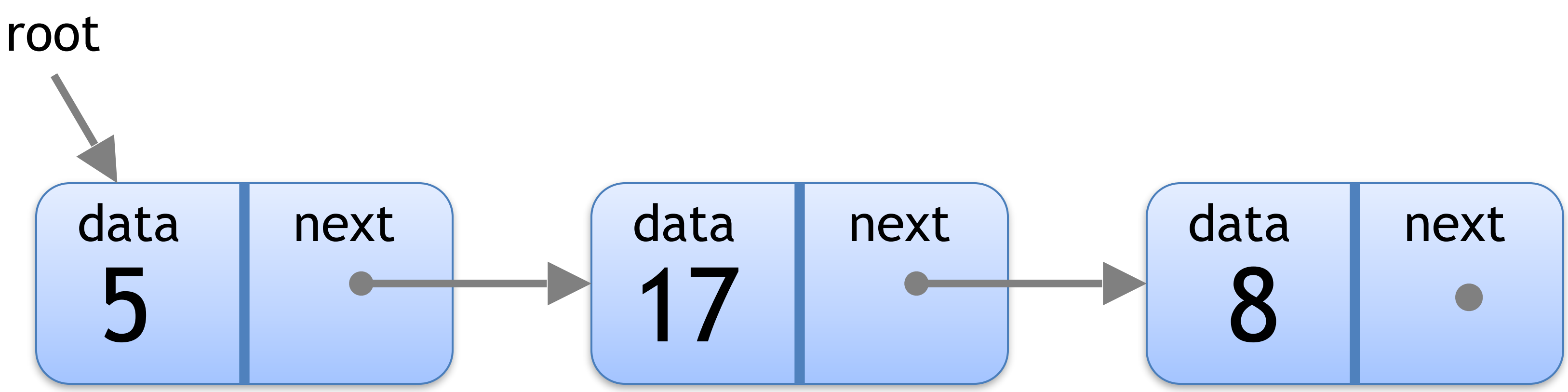
find(data)

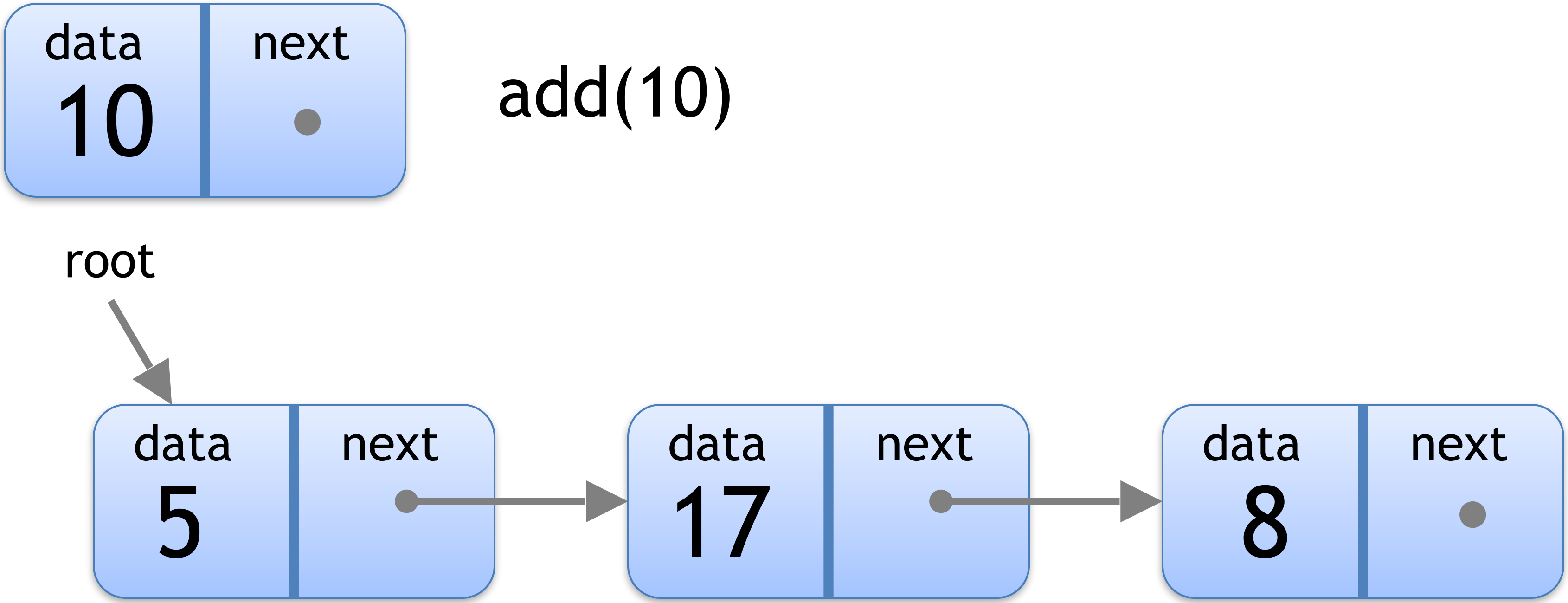
add(data)

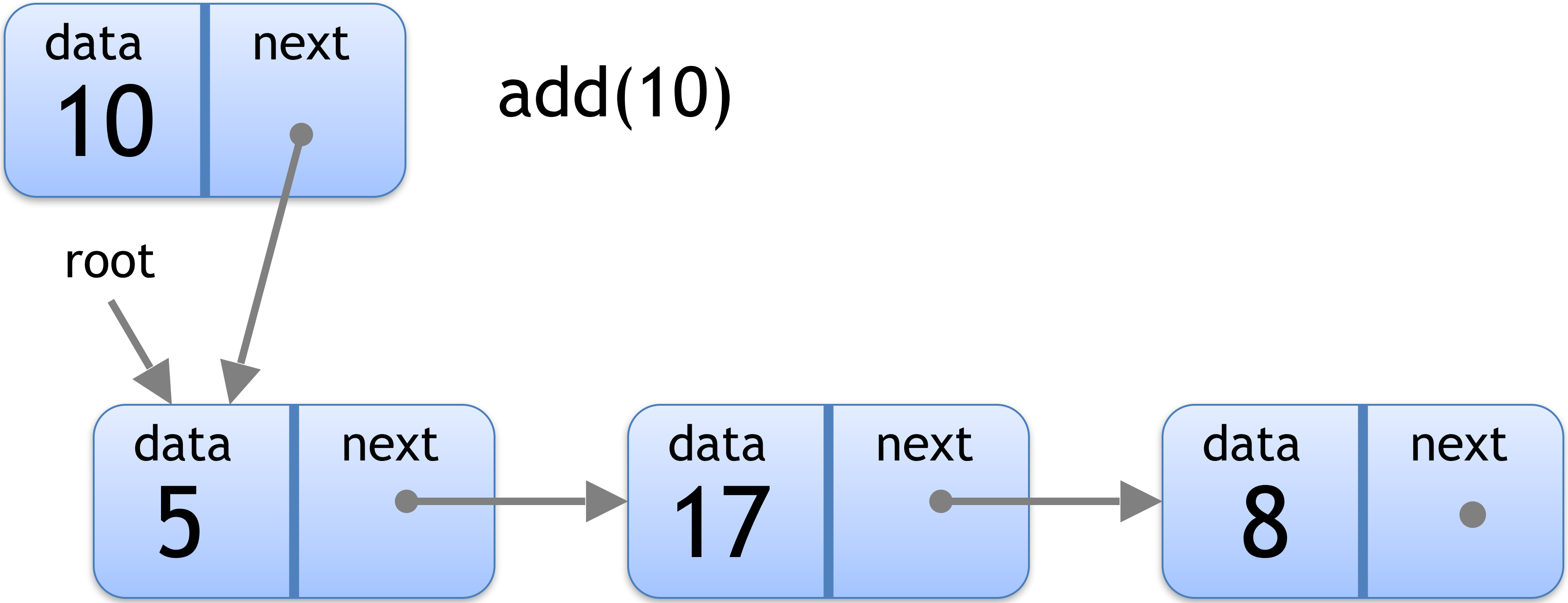
remove(data)

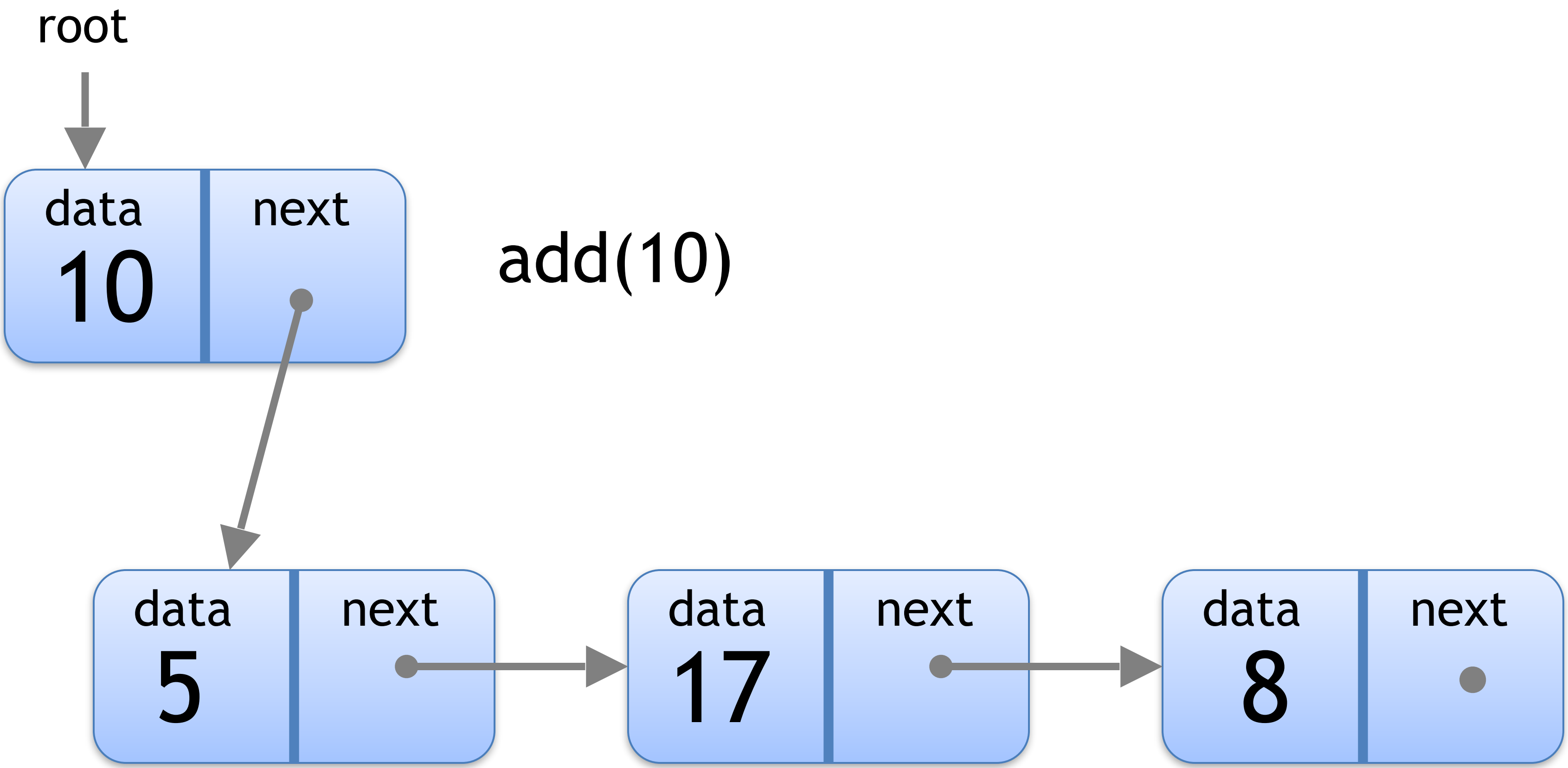
print_list()

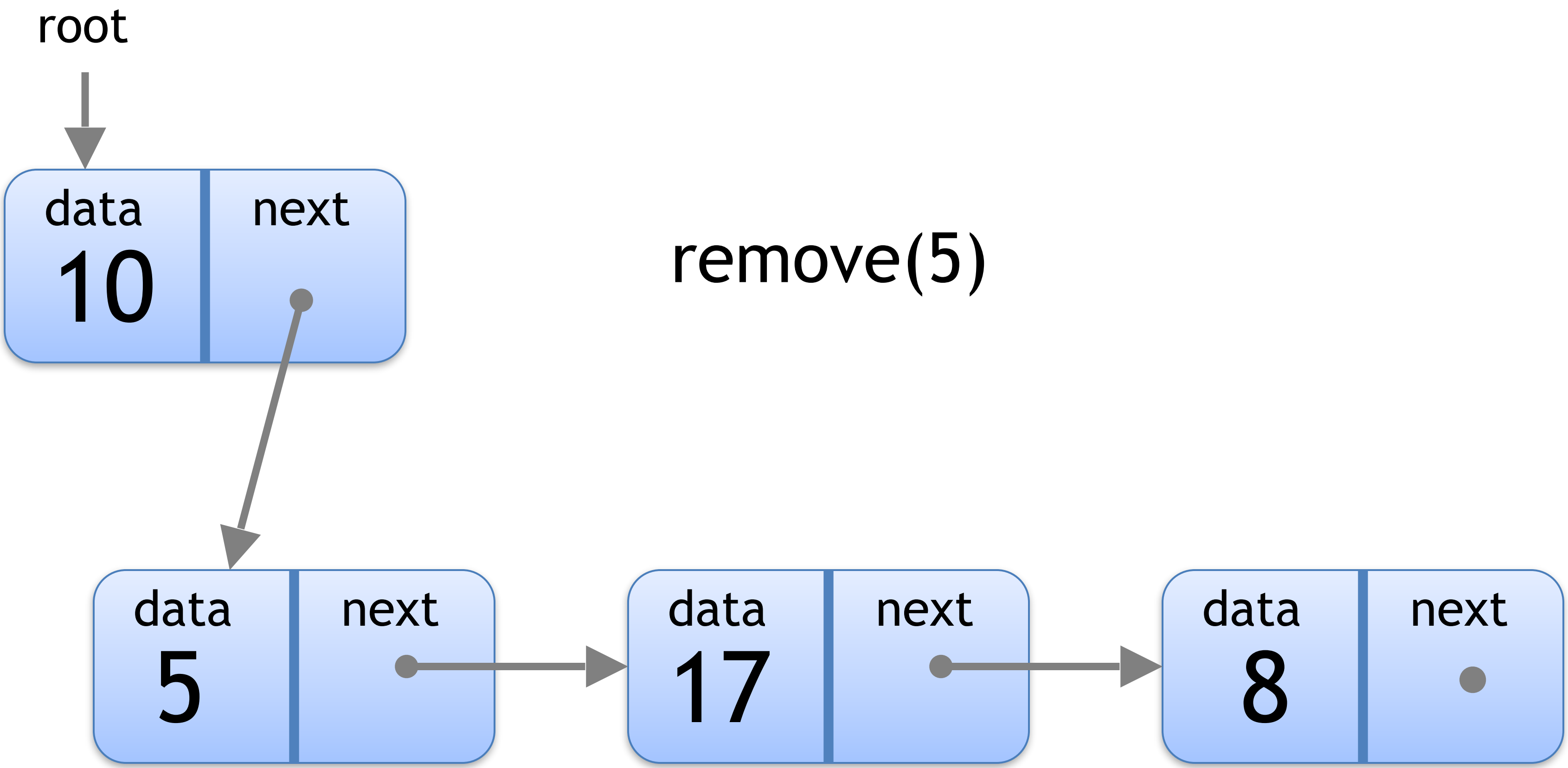
add(10)



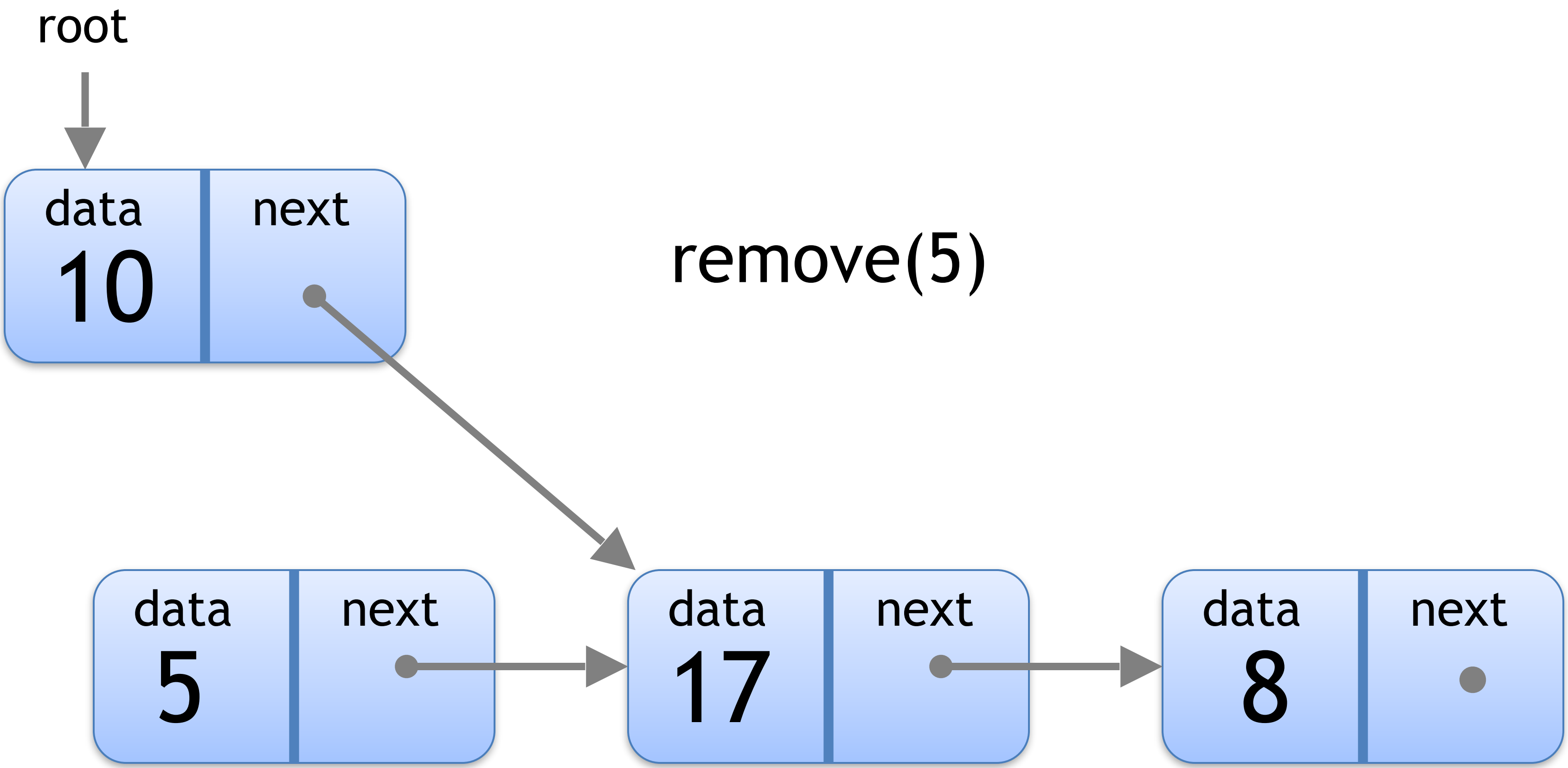

data
10
next
add(10)
root
data
5
next
data
17
next
data
8
next

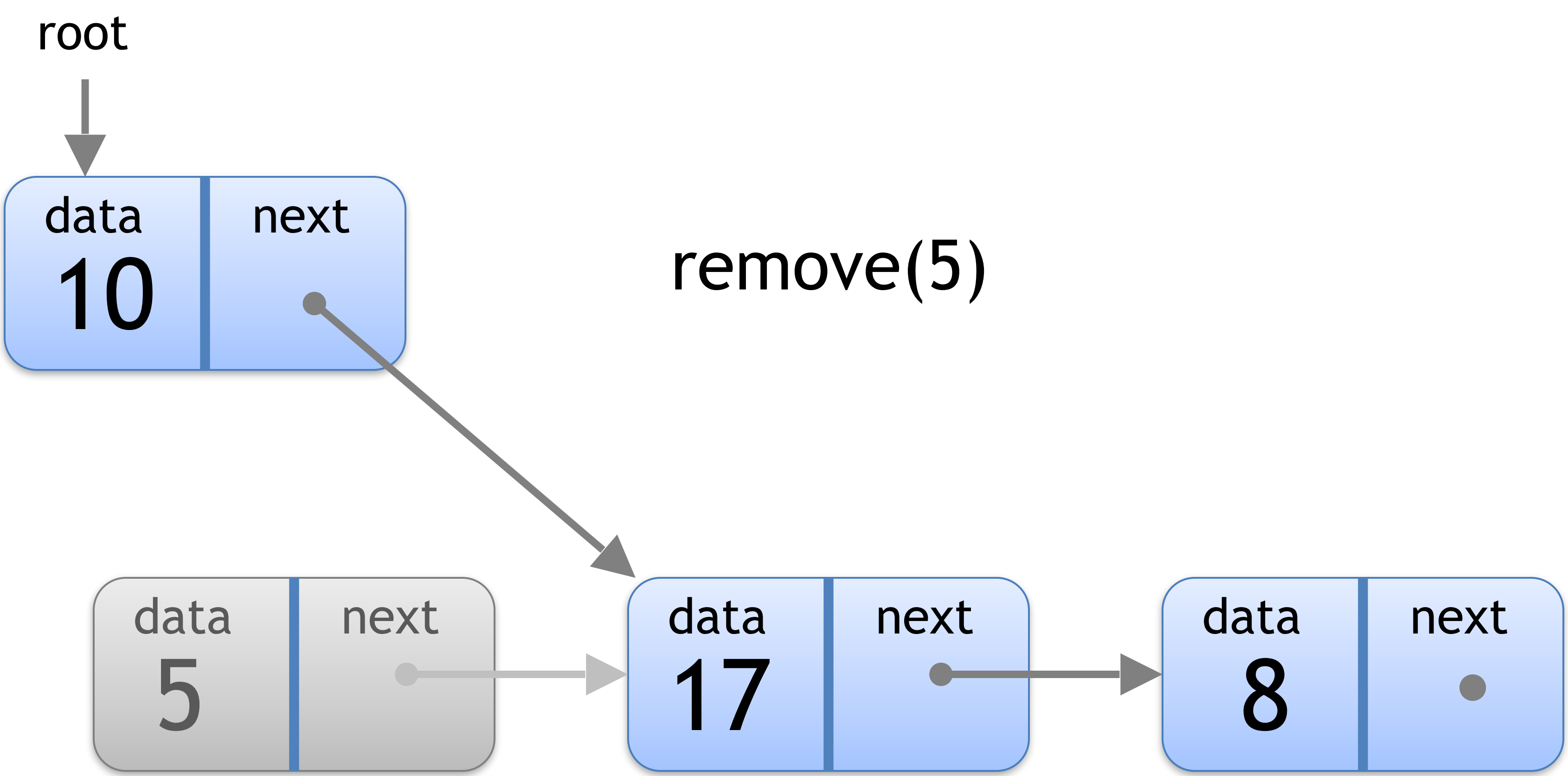

data
10
next
add(10)
root
data
5
next
data
17
next
data
8
next


root
data
10
next
add(10)
data
5
next
data
17
next
data
8
next



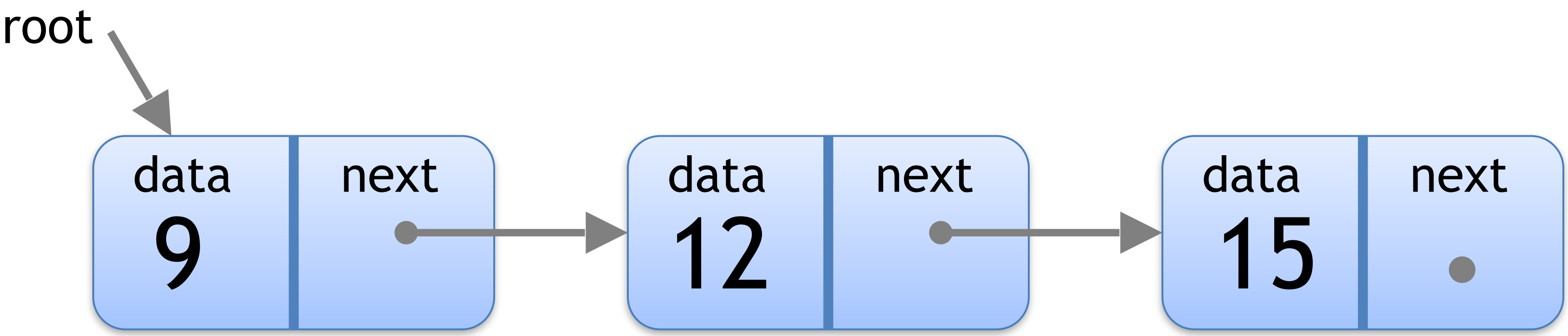
remove(5)


root
data
10
next
remove(5)
data
5
next
data
17
next
data
8
next

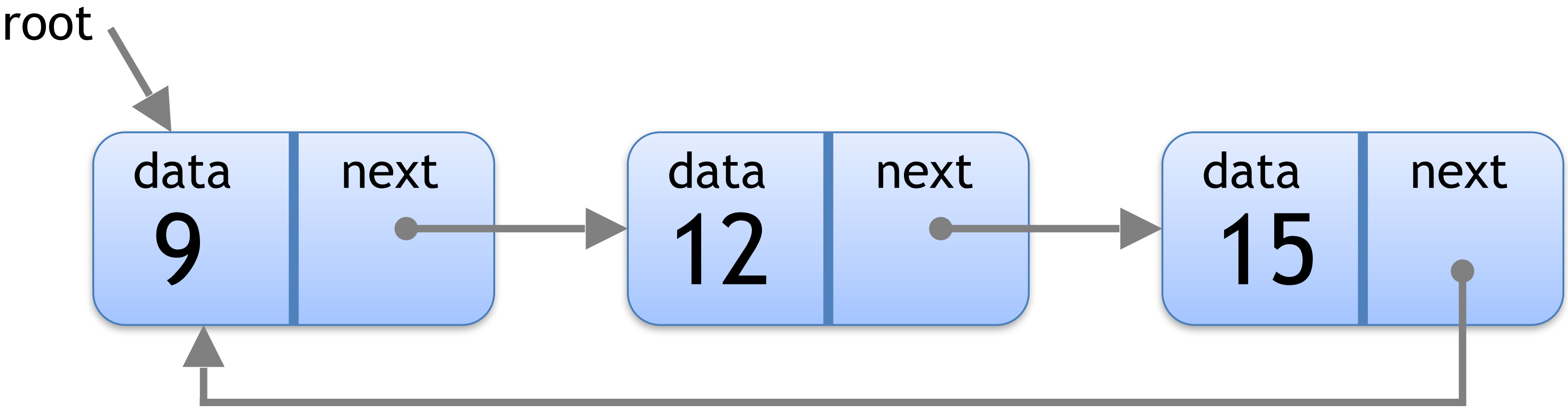

root
data
10
next
remove(5)
data
5
next
data
17
next
data
8
next

Python Circular Linked Lists

Regular Linked List

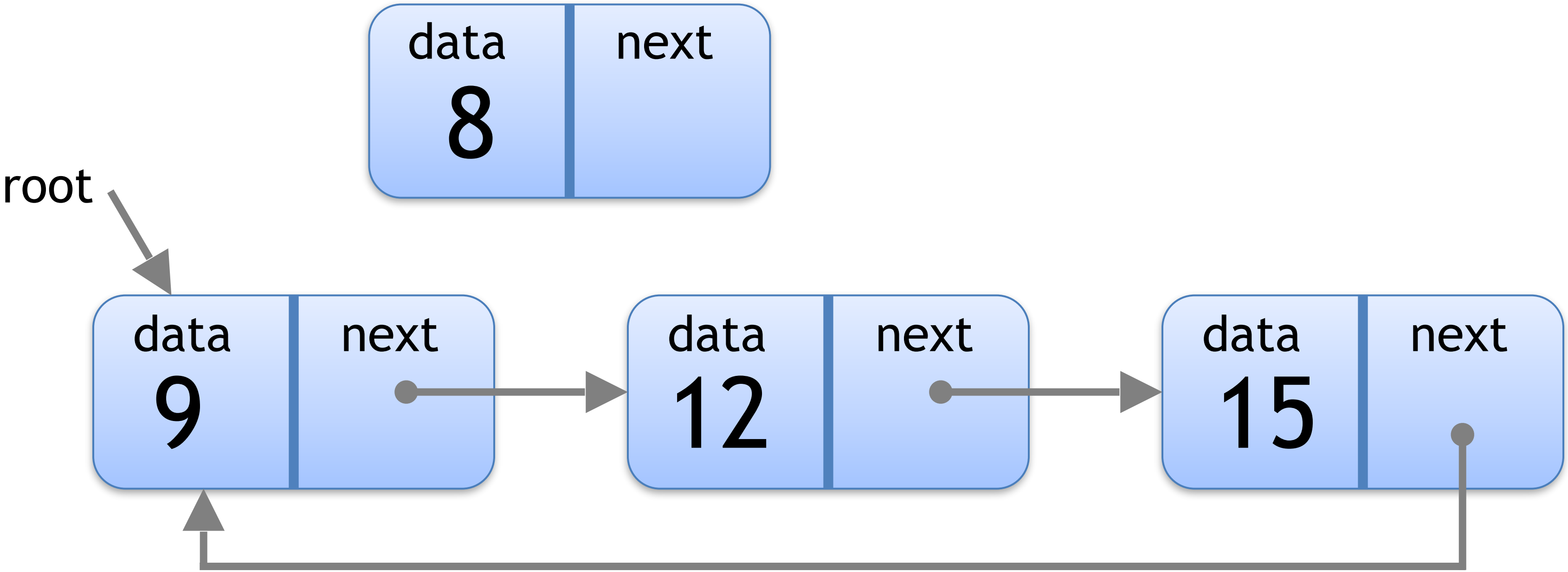


Circular Linked List



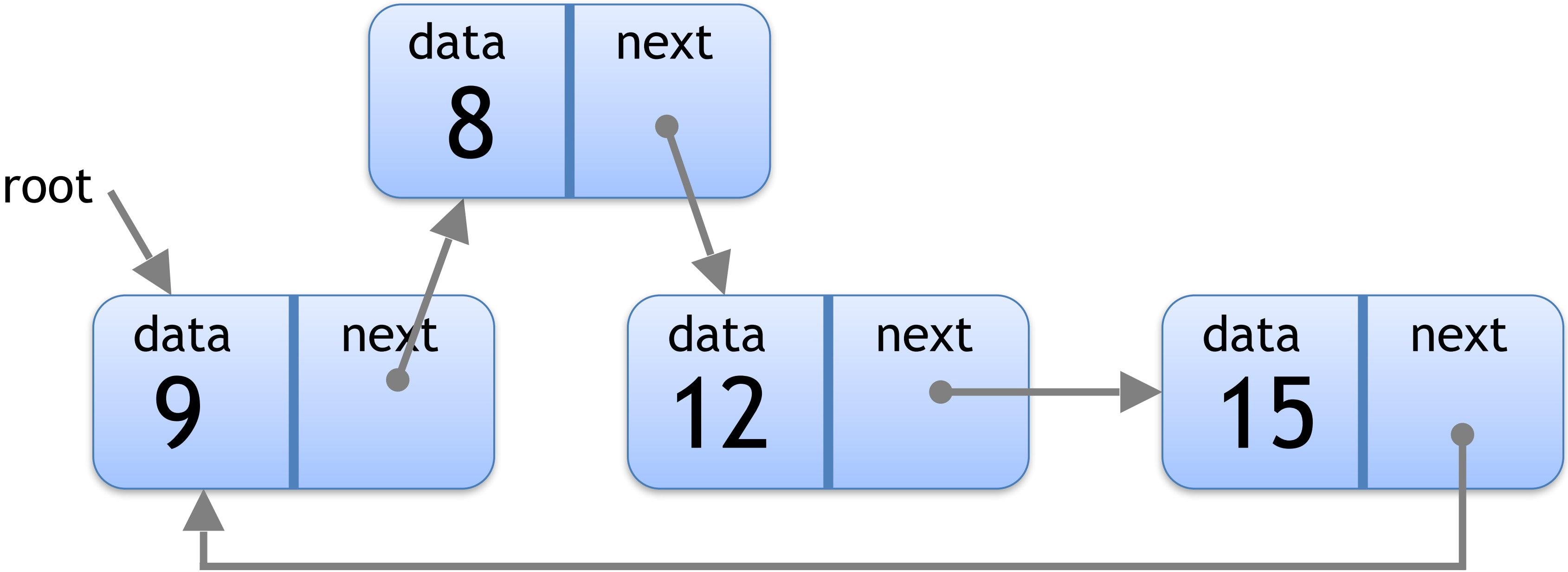
Circular Linked List

add(8)



Circular Linked List

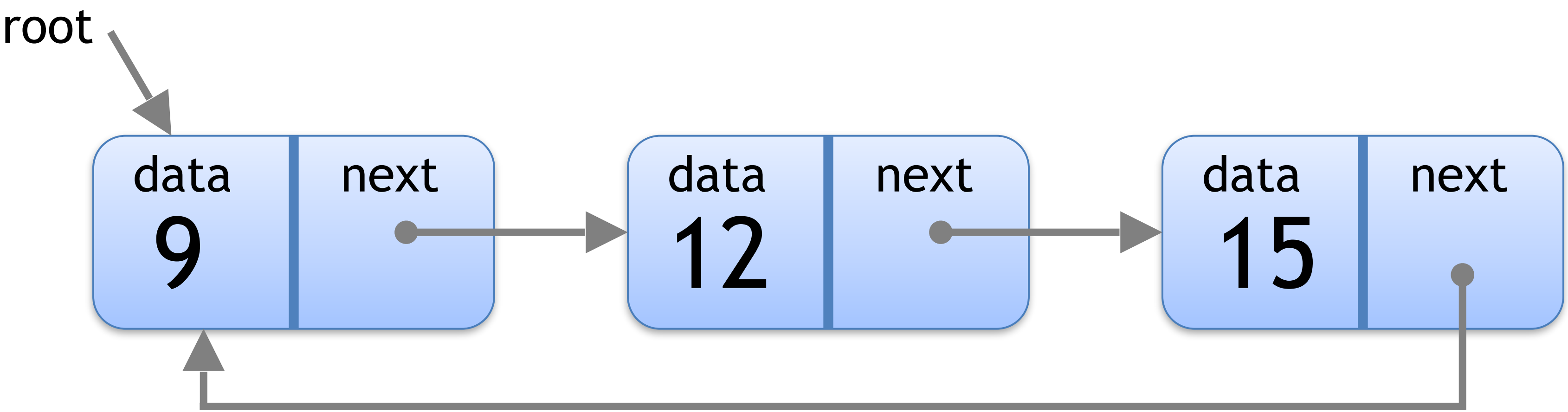
add(8)



Circular Linked List

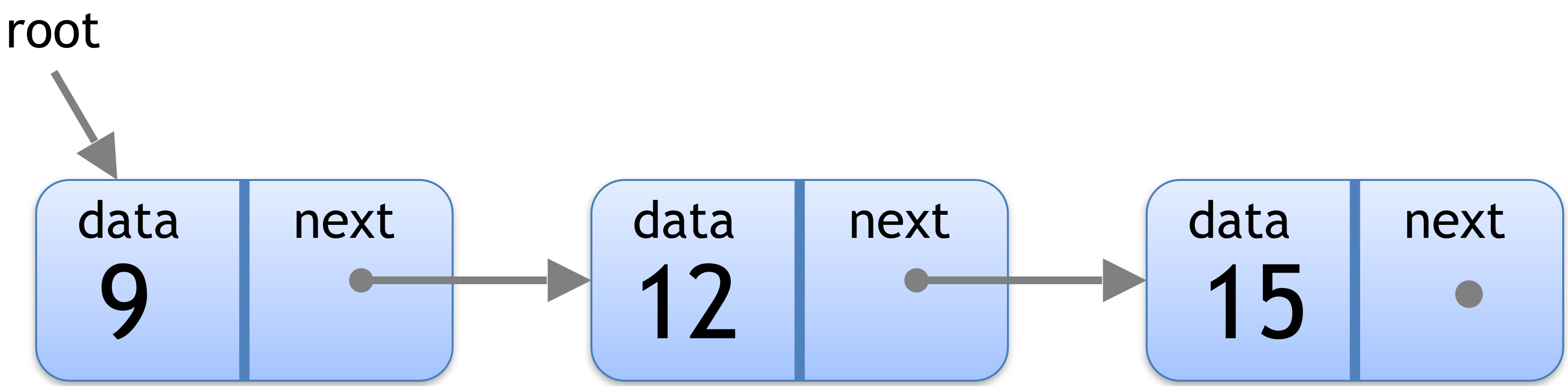
Advantage over regular (singly) linked lists:

- Ideal for modeling continuous looping objects, such as a Monopoly board or a race track.



Python Doubly Linked Lists

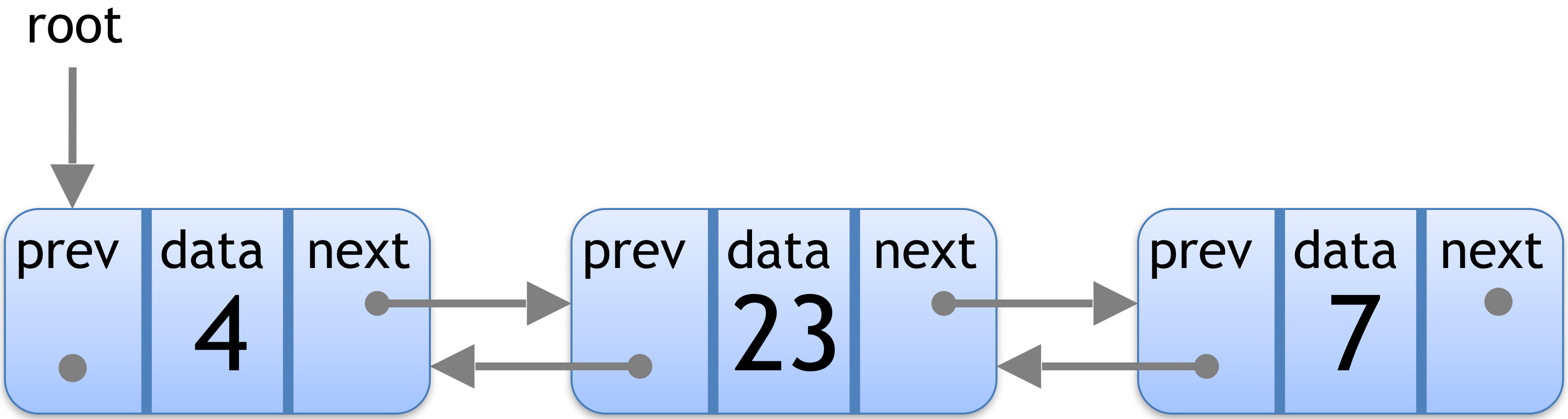
Regular Linked List



Doubly Linked List

Every Node has 3 parts:
data and pointers to
previous and **next** Nodes

Doubly Linked List



Doubly Linked List

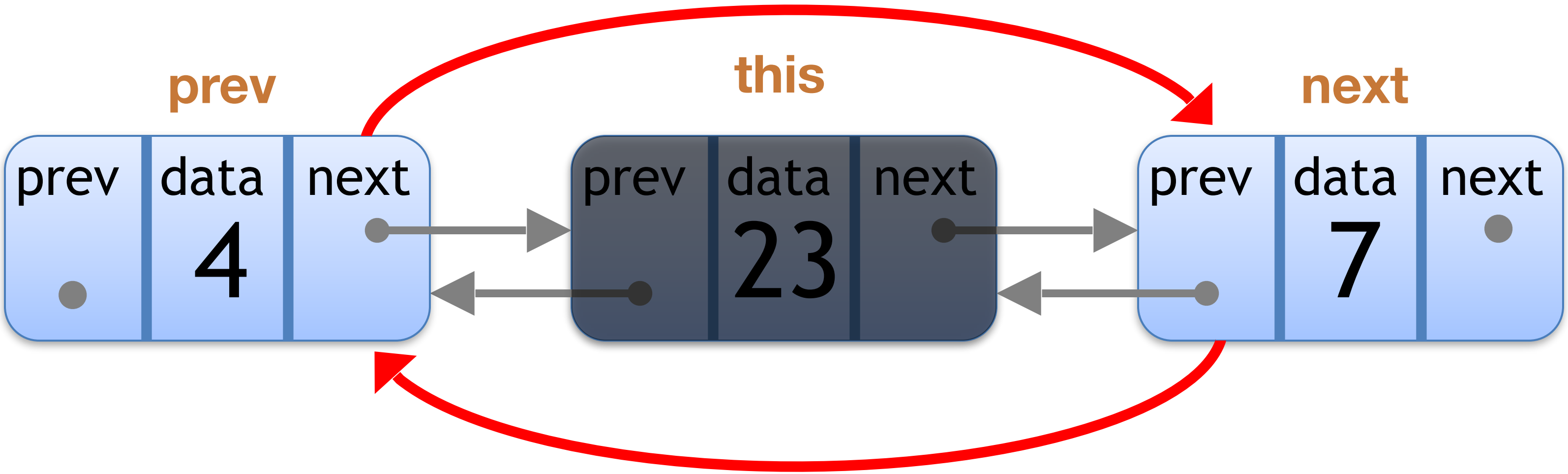
Delete Node

Doubly Linked List

Delete Node

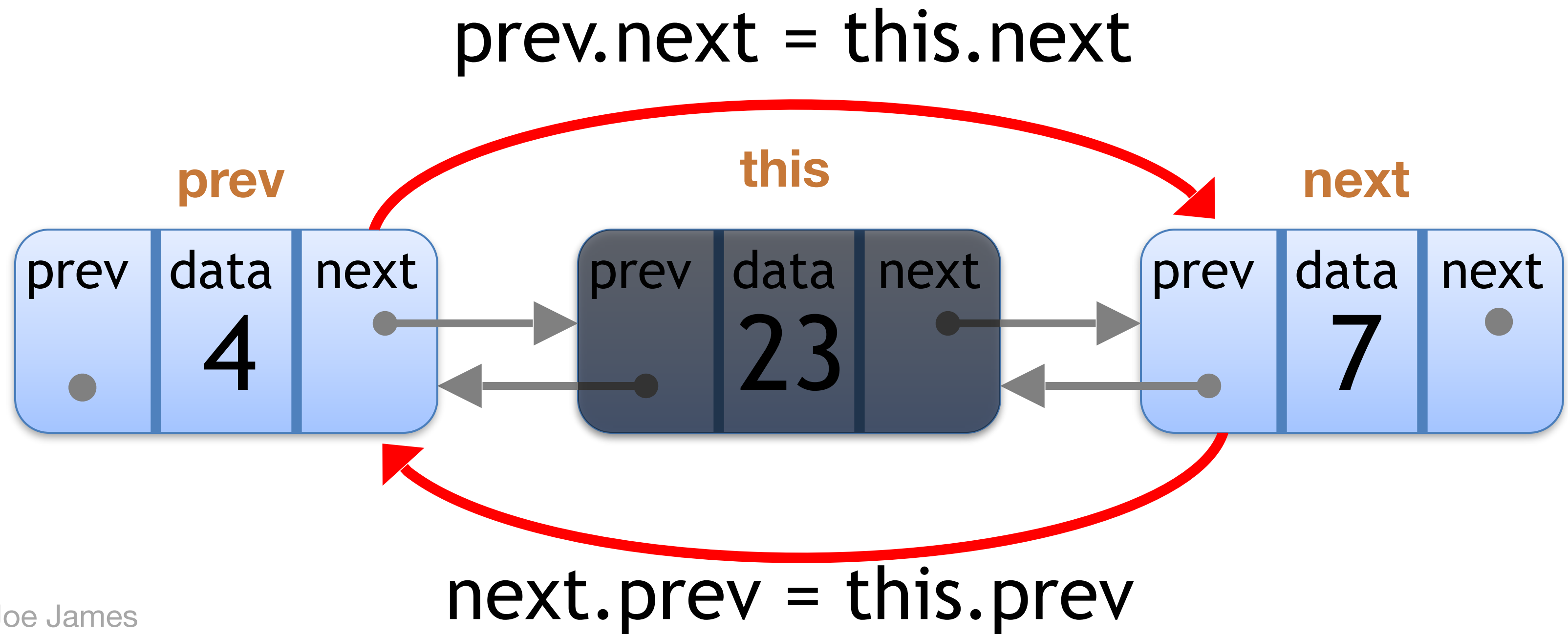
Doubly Linked List

Delete Node



Doubly Linked List

Delete Node



Doubly Linked List

Advantages over regular (singly) linked lists:

- Can iterate the list in either direction
- Can delete a node without iterating through the list (if given a pointer to the node)

Stacks and Queues

Stacks

Push an item onto the stack

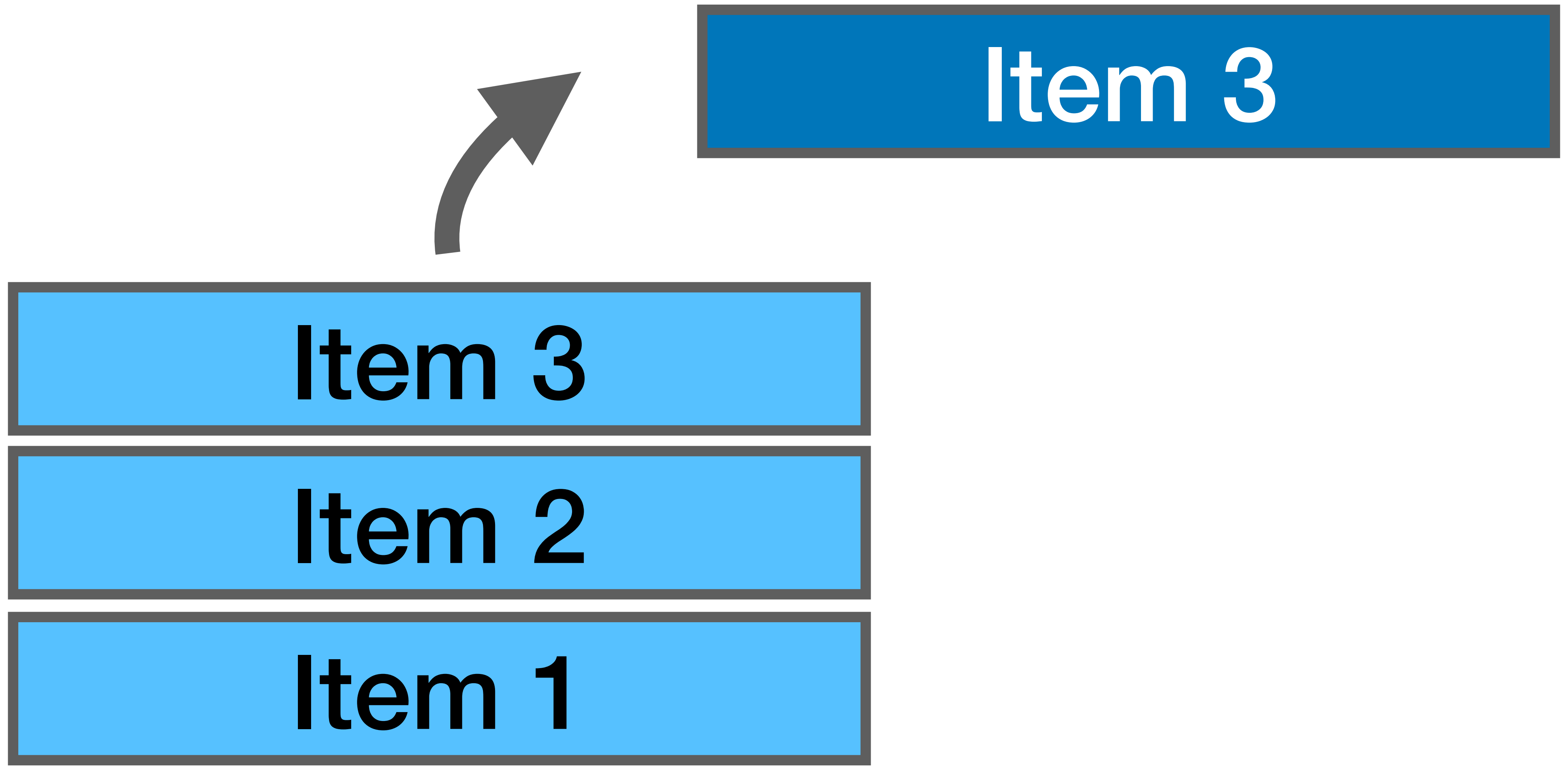
Pop an item off of the stack

LIFO: Last-In First-Out

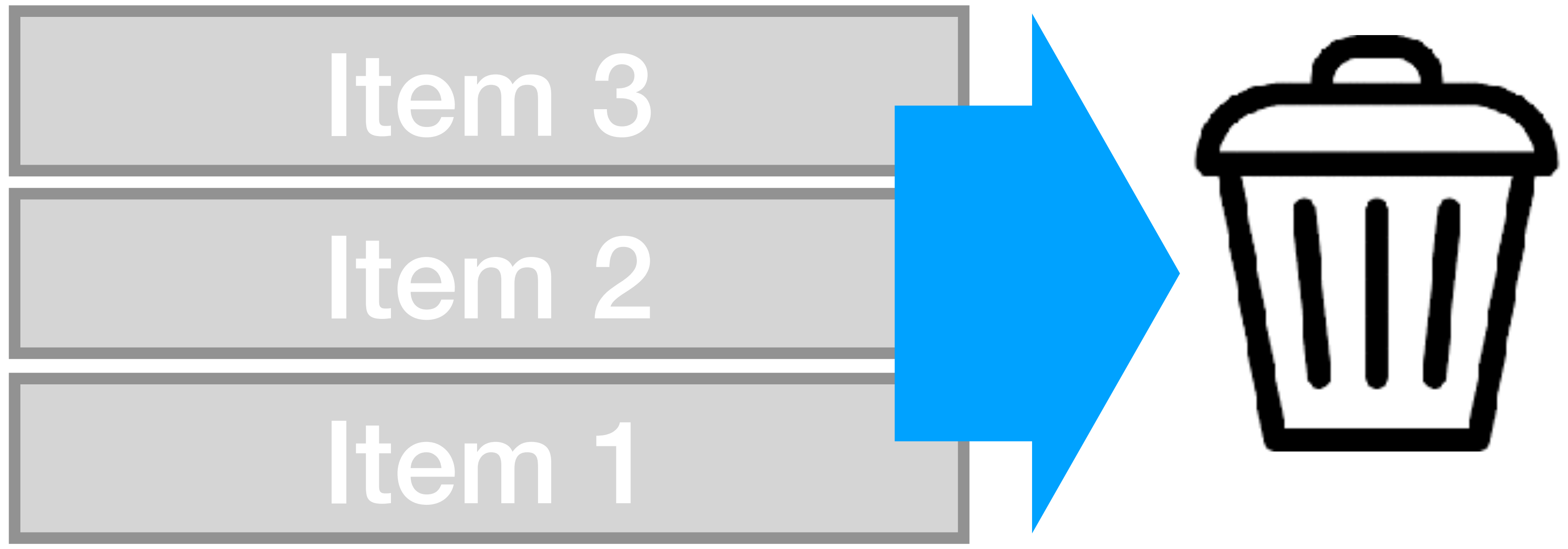
All **push** and **pop** operations are to/from the **top** of the stack.

Stacks

Peek - get item on top of stack, without removing it.



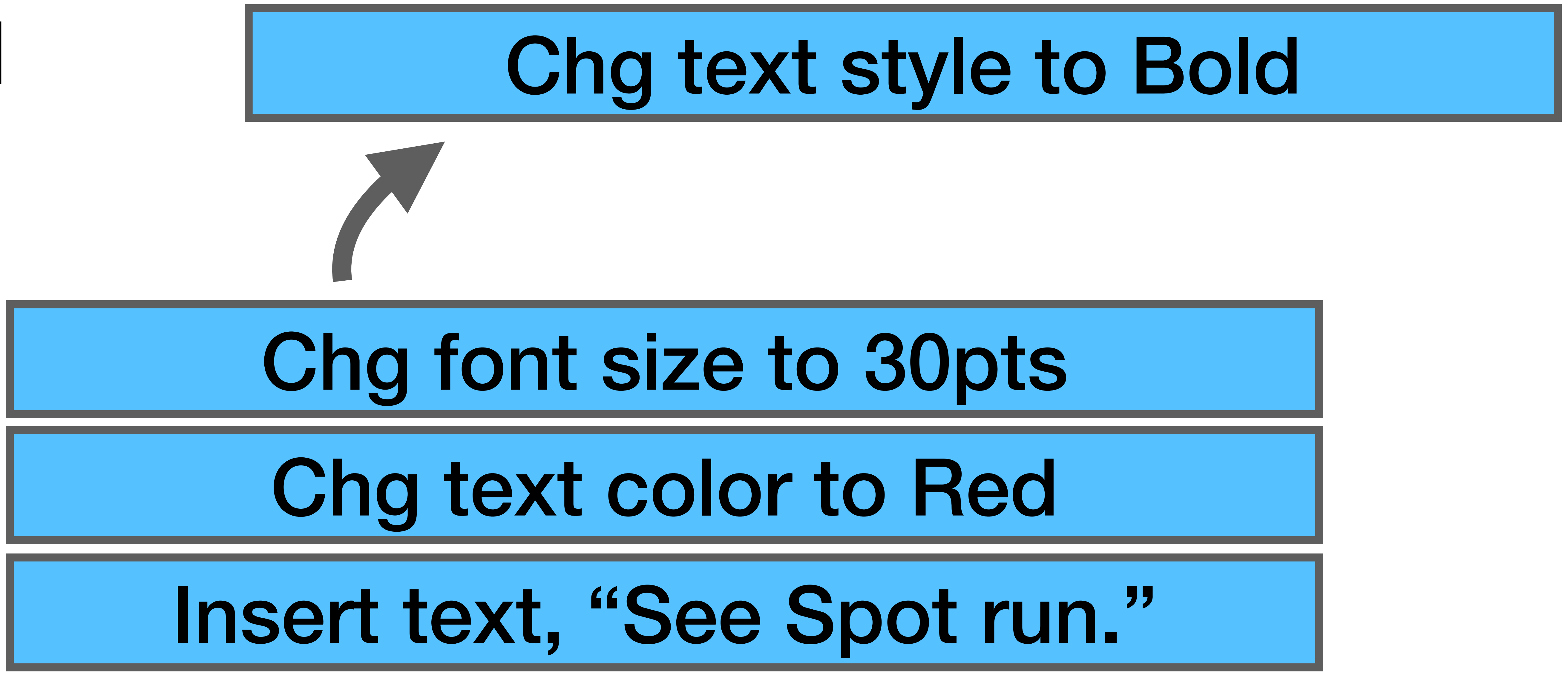
Clear all items from stack



Stacks Use Case

Undo - track which commands have been executed.
Pop last command off command stack to undo it.

Pop last command
to undo bold.



Queues

Enqueue - add an item to the end of the line.

Dequeue - remove an item from the front of the line.

FIFO: First-In First-Out

Enqueue on one end, and Dequeue from the other end.

Queues Use Cases

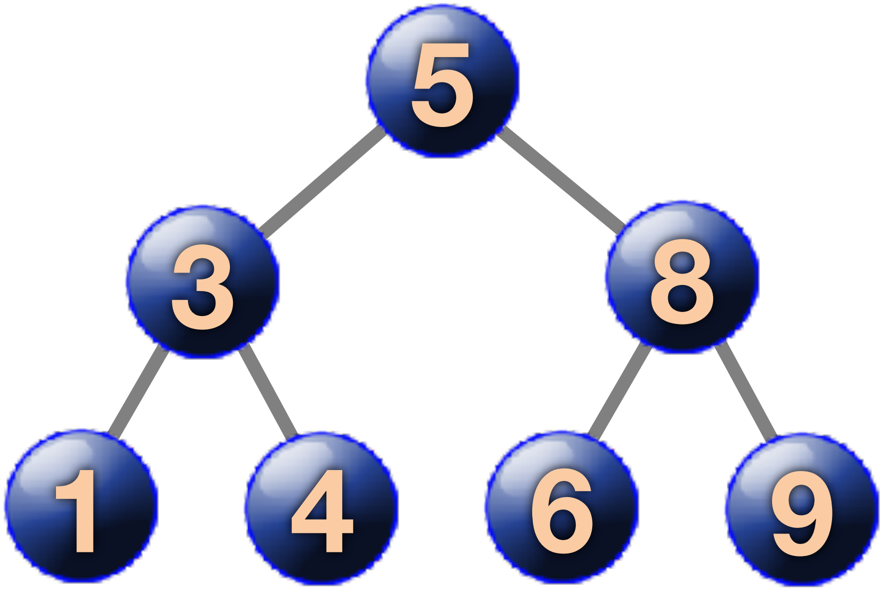
Queues are good for modeling anything you wait in line for.

Bank tellers. Placing an order at McDonalds.

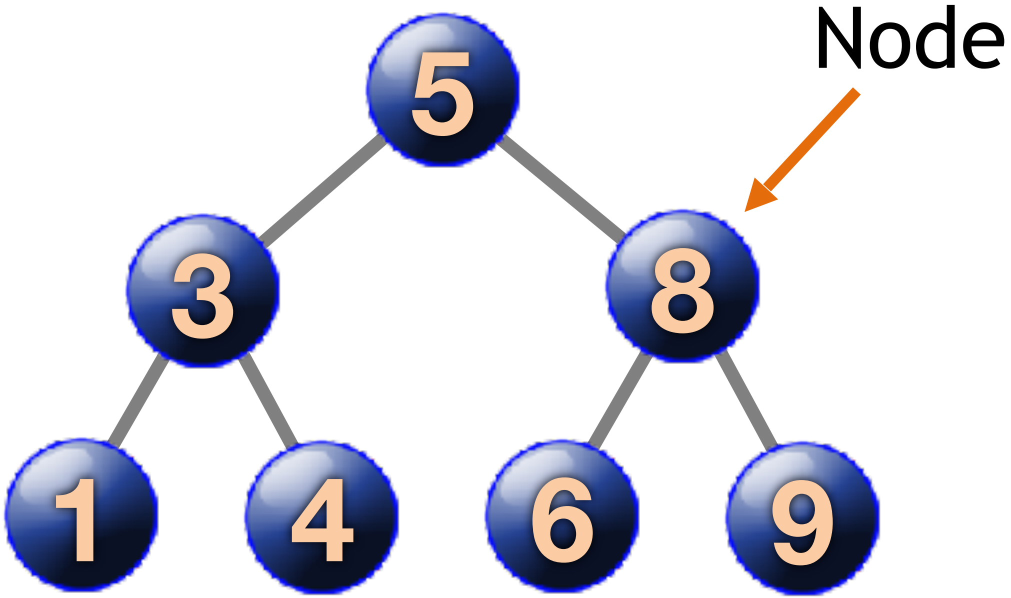
DMV customer service. Supermarket checkout.

Binary Search Trees

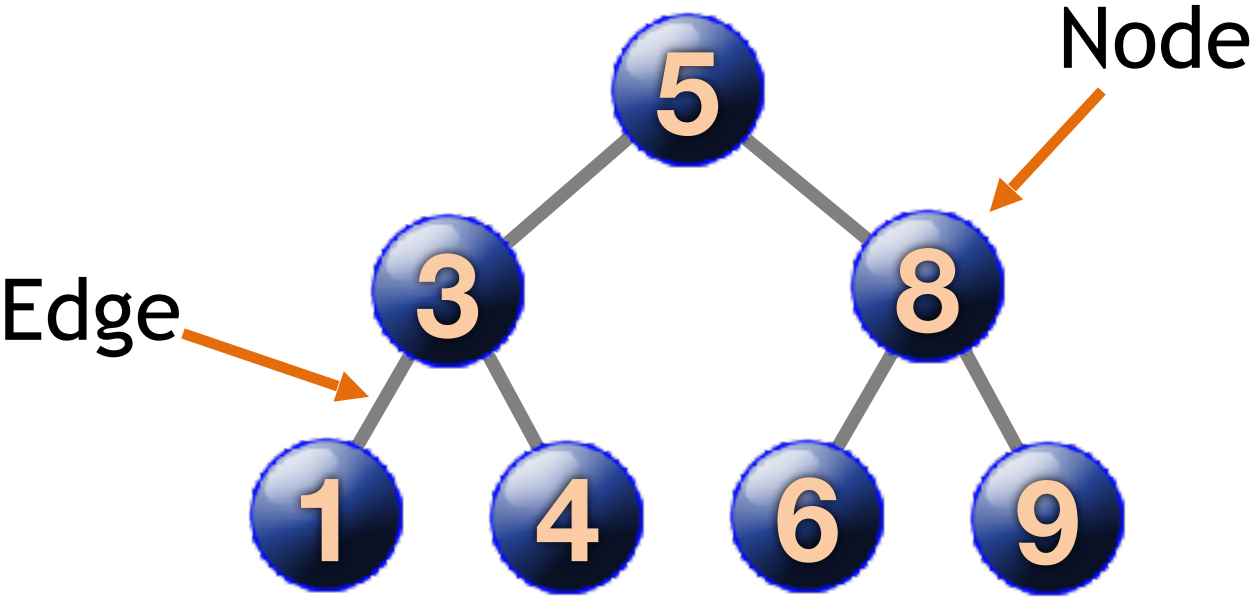
Tree



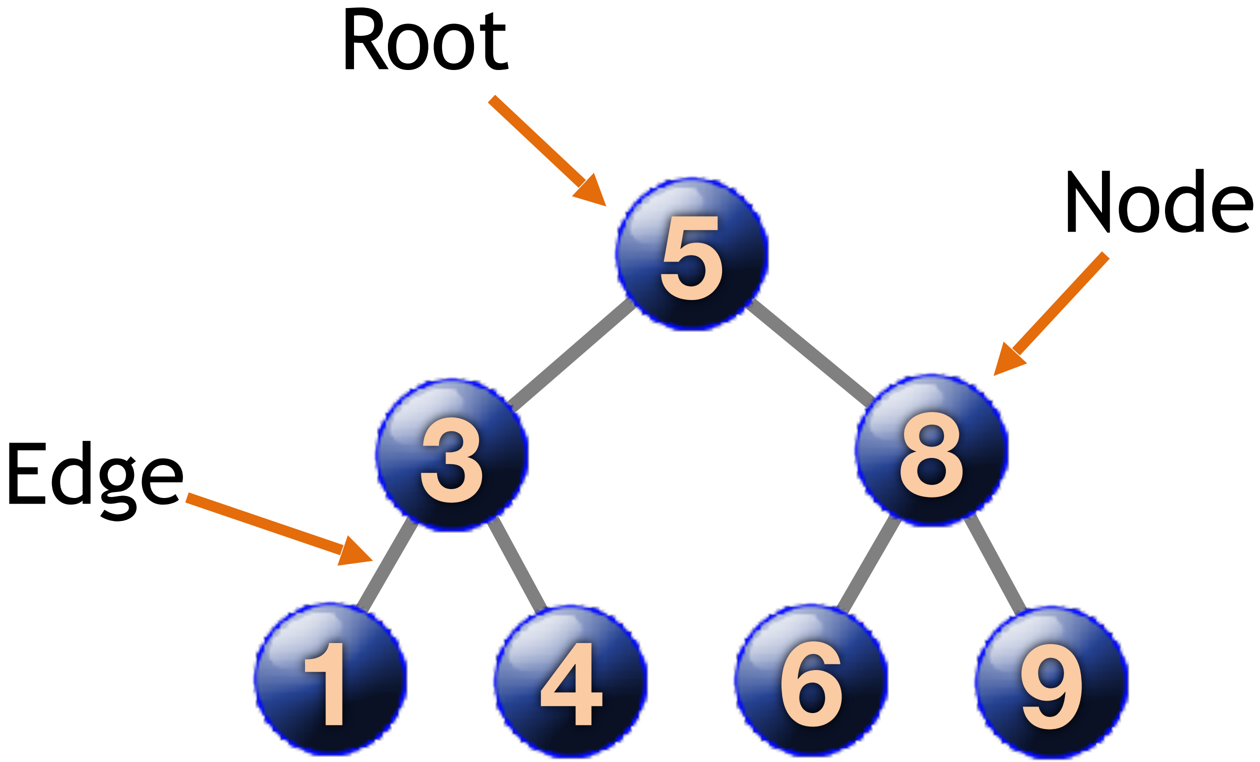
Tree

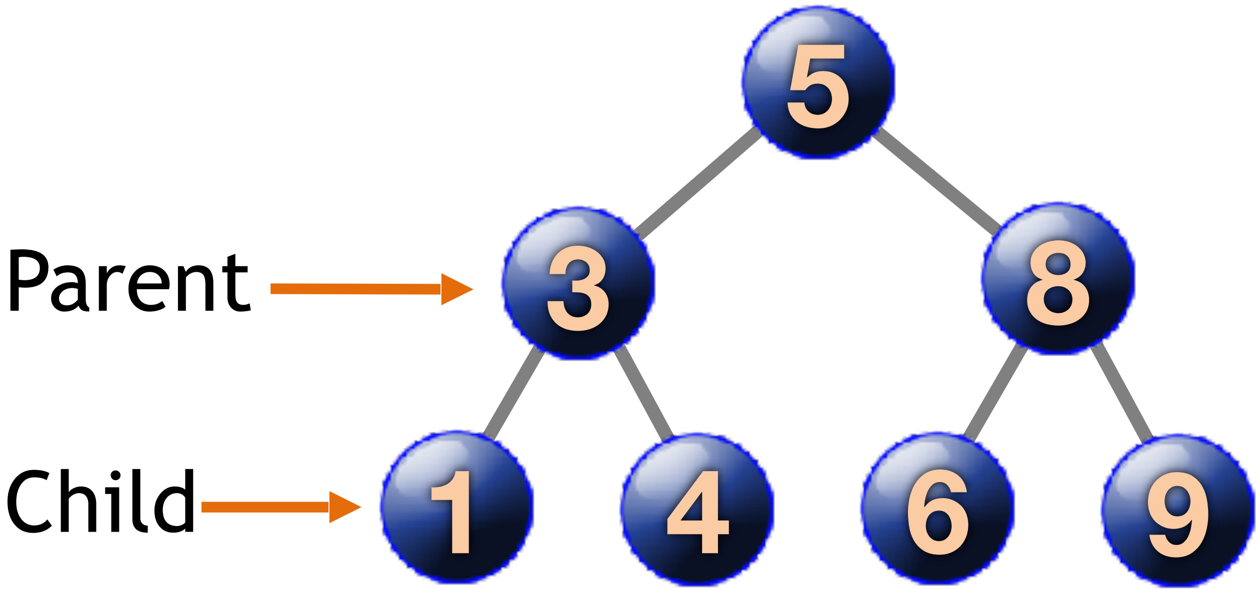


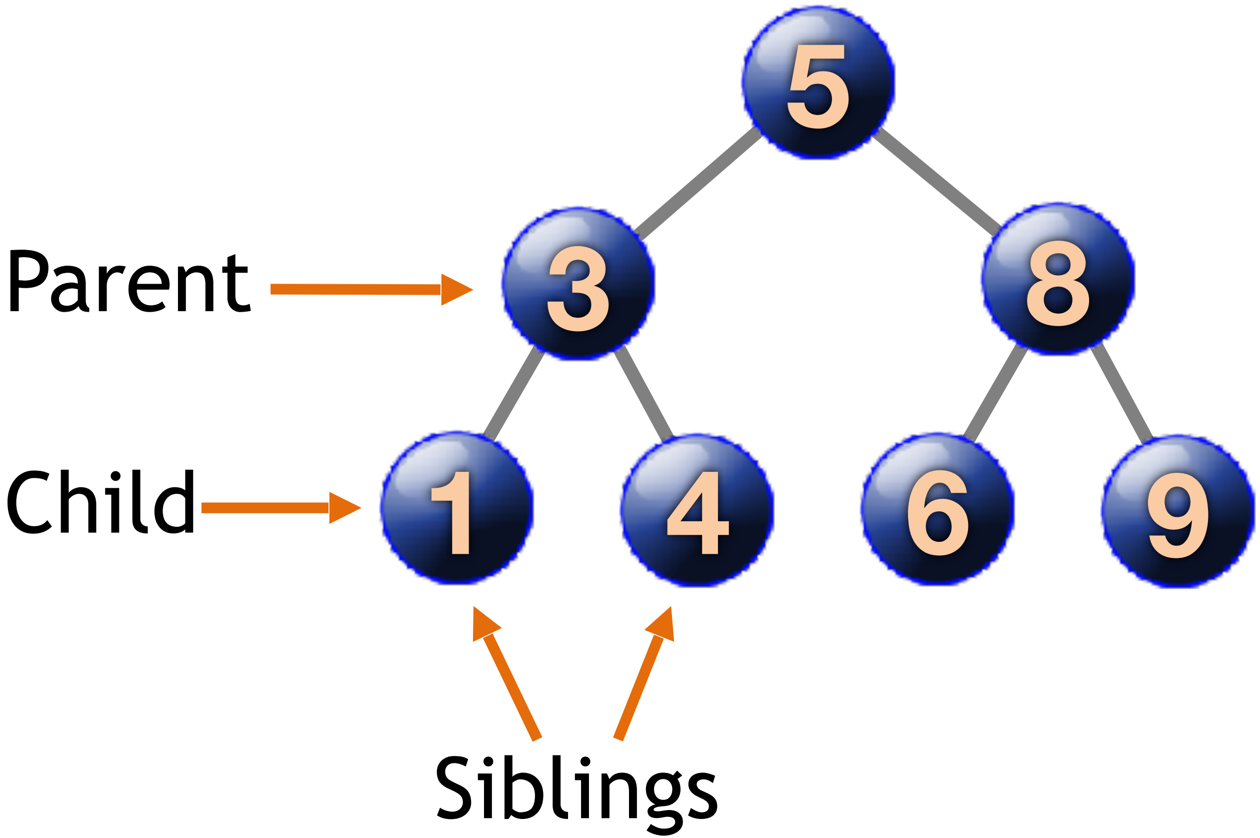
Tree

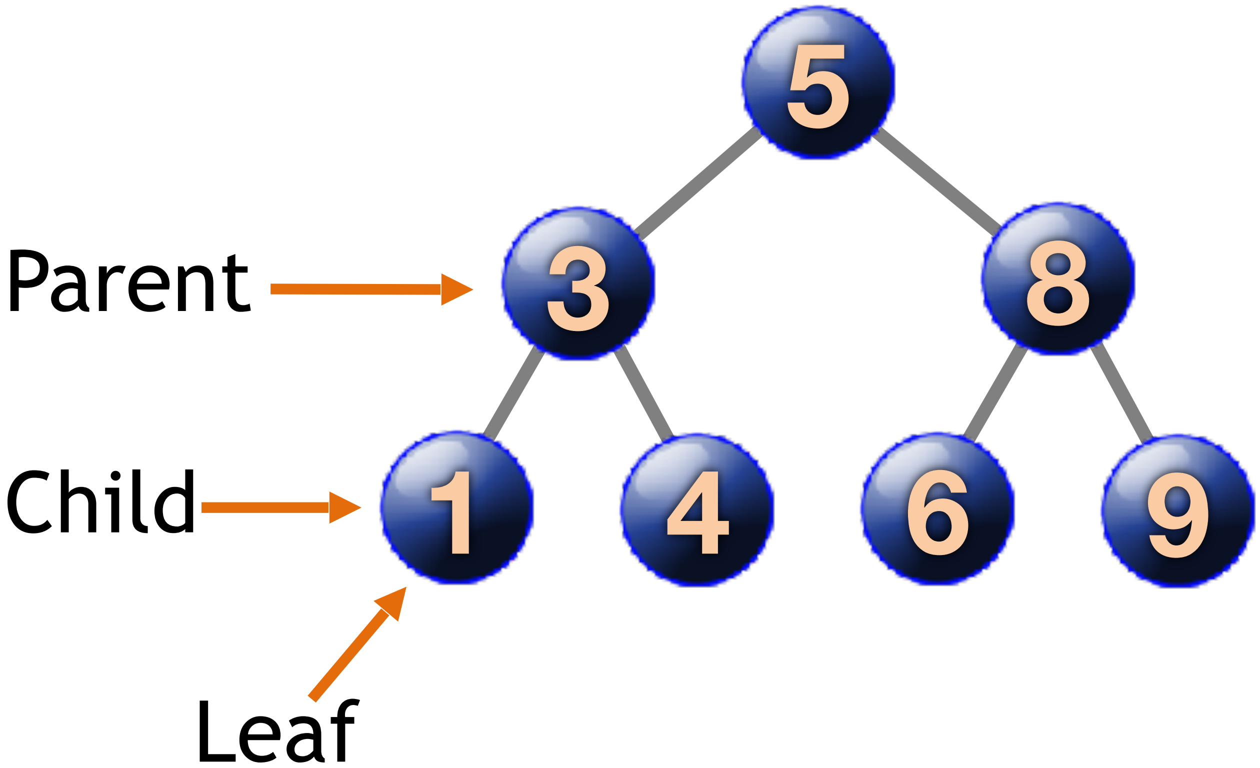


Tree

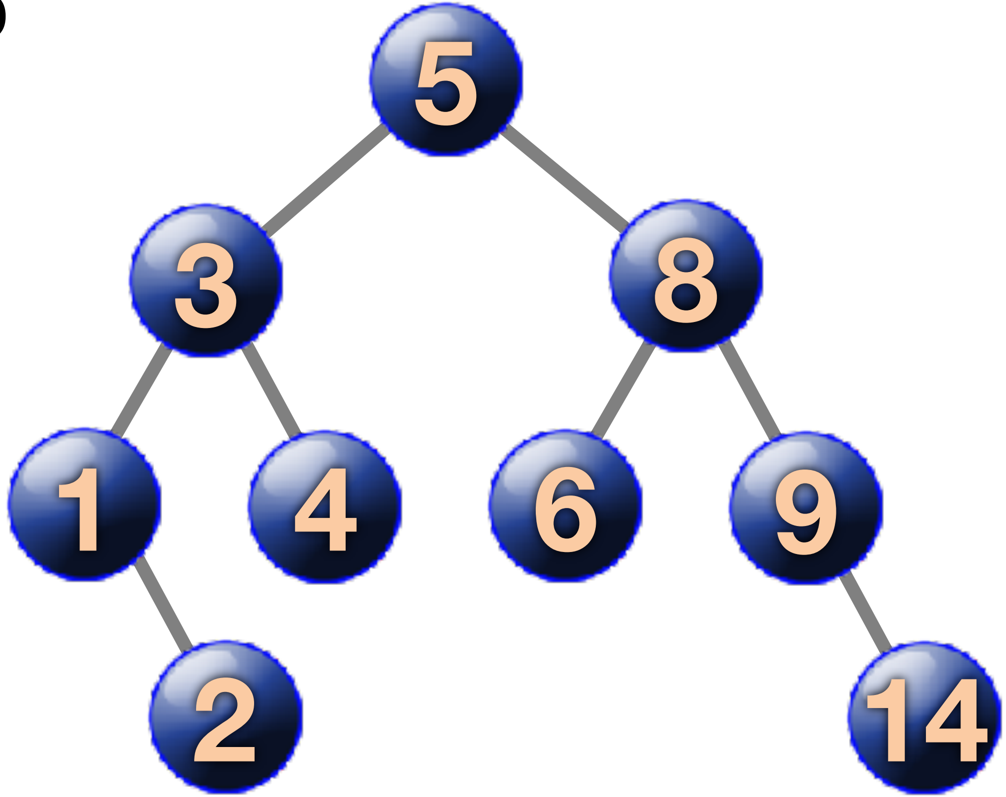


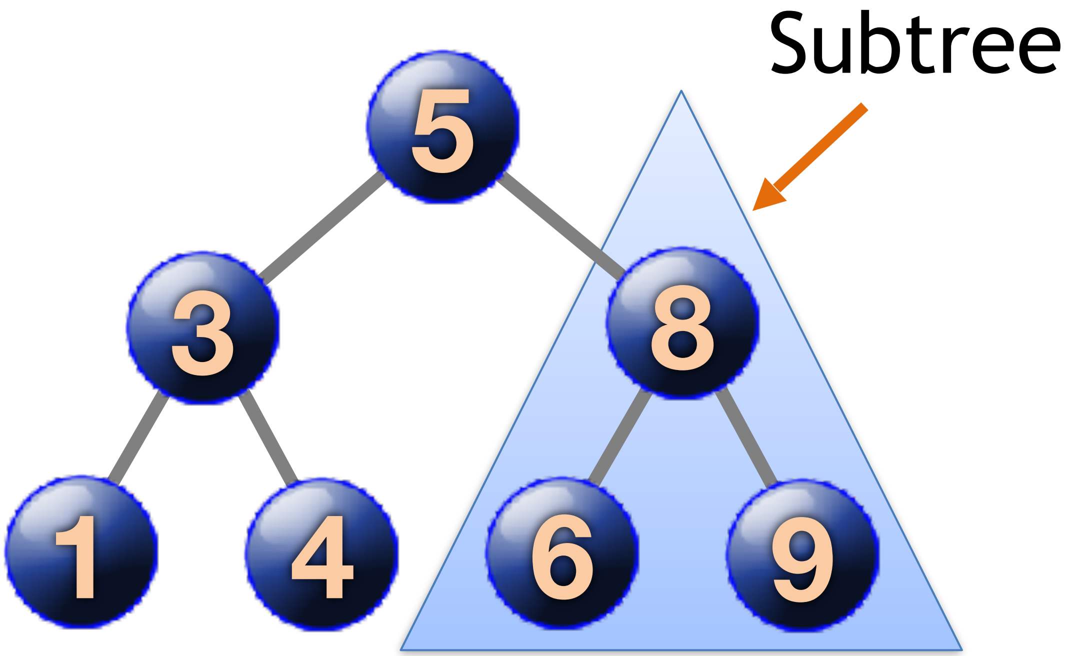

Parent
Child
5
3
8
1
4
6
9

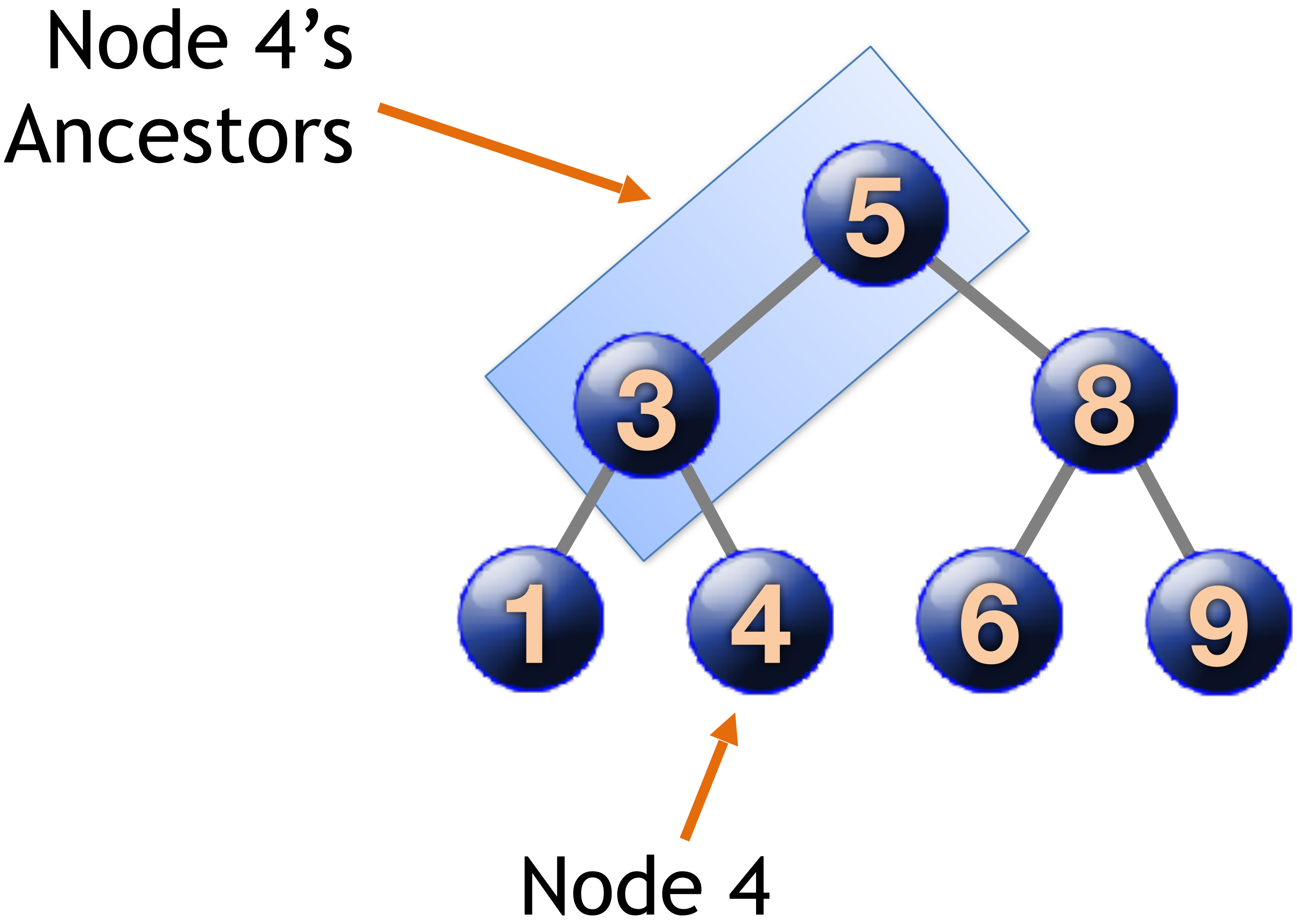

5
3
8
1
4
6
9
Parent
Child
Siblings


5
3
8
1
4
6
9
Parent
Child
Leaf

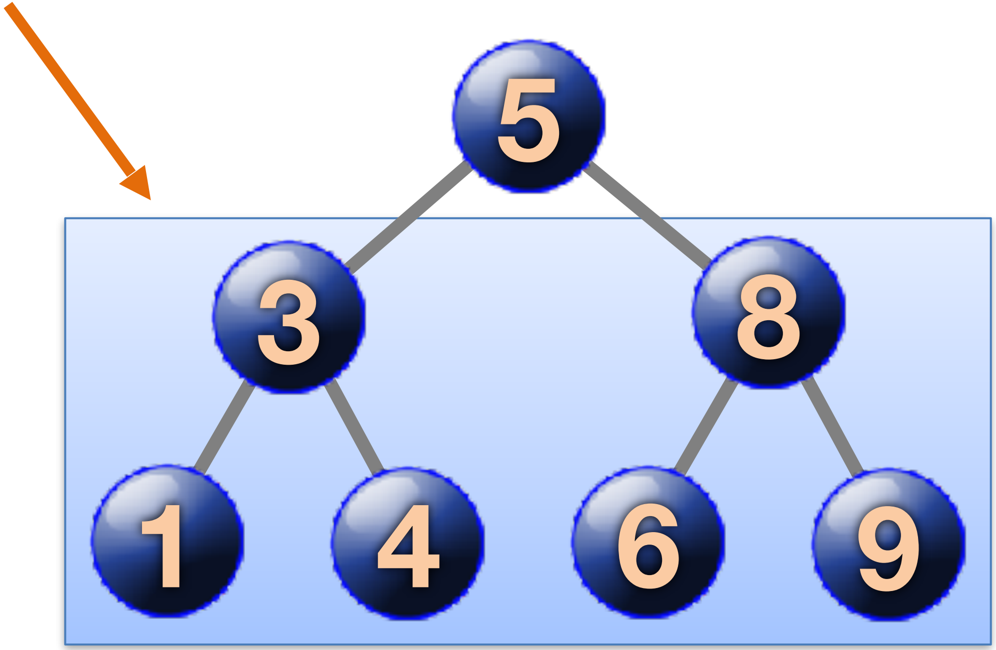
Binary Tree - each node can have up to 2 child nodes.




Subtree
5
3
8
1
4
6
9

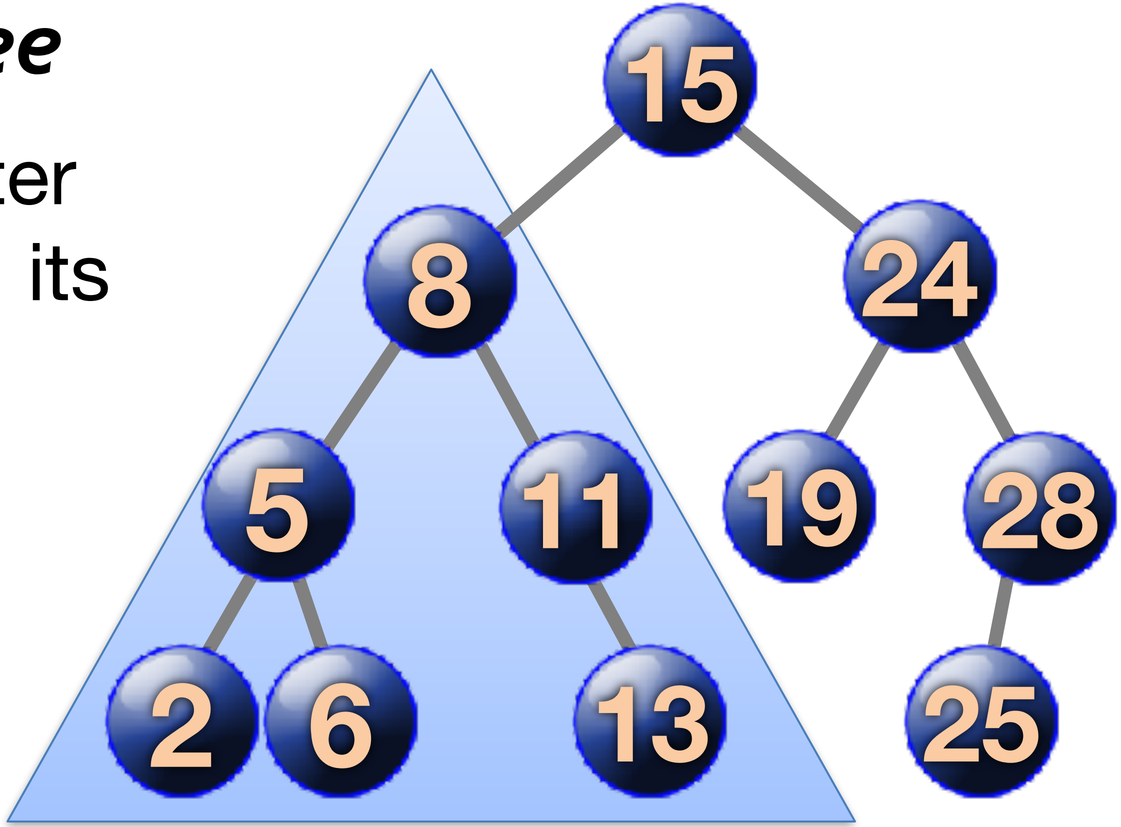

Node 4's
Ancestors
5
3
8
1
4
6
9
Node 4

Node 5's Descendants



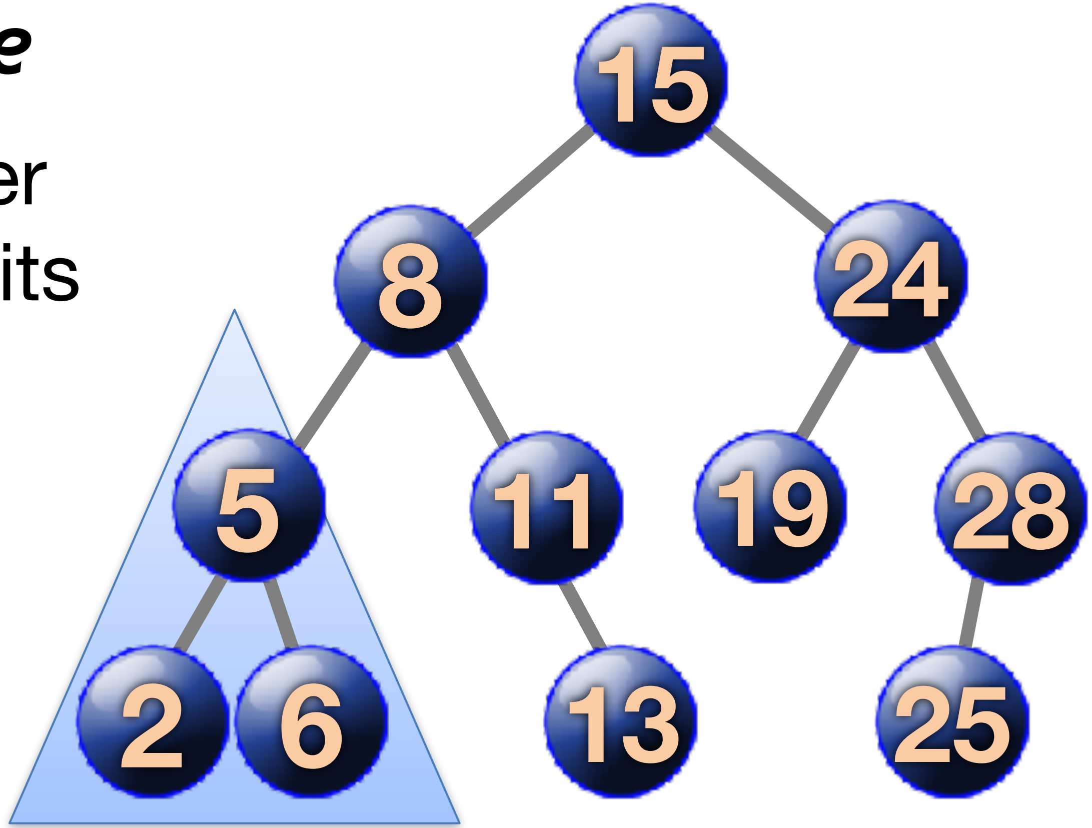
Binary Search Tree

- each node is greater than every node in its left subtree



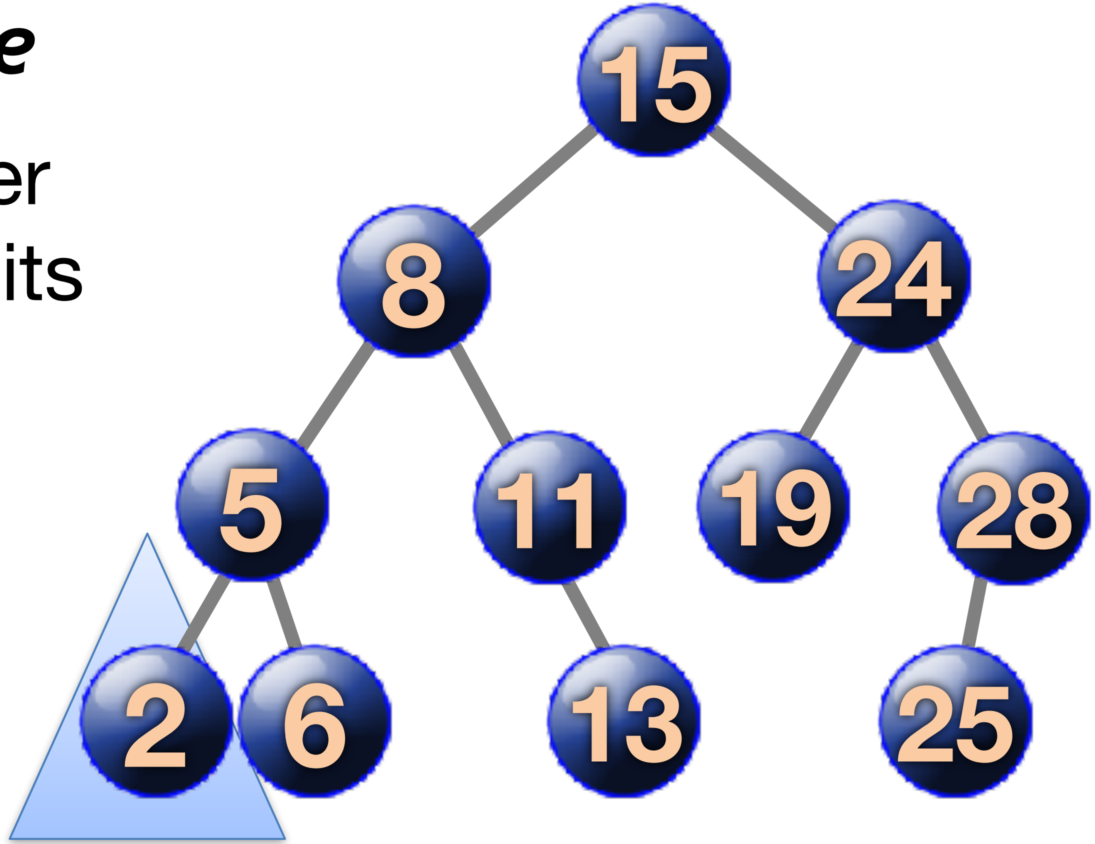
Binary Search Tree

- each node is greater than every node in its left subtree



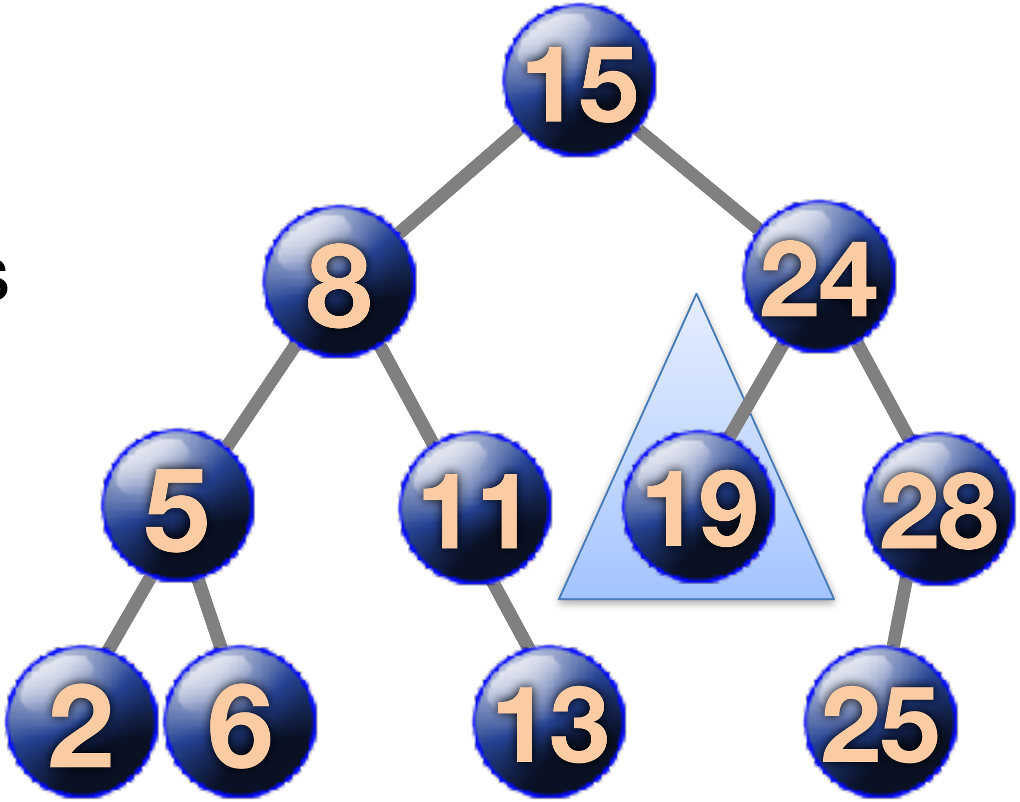
Binary Search Tree

- each node is greater than every node in its left subtree



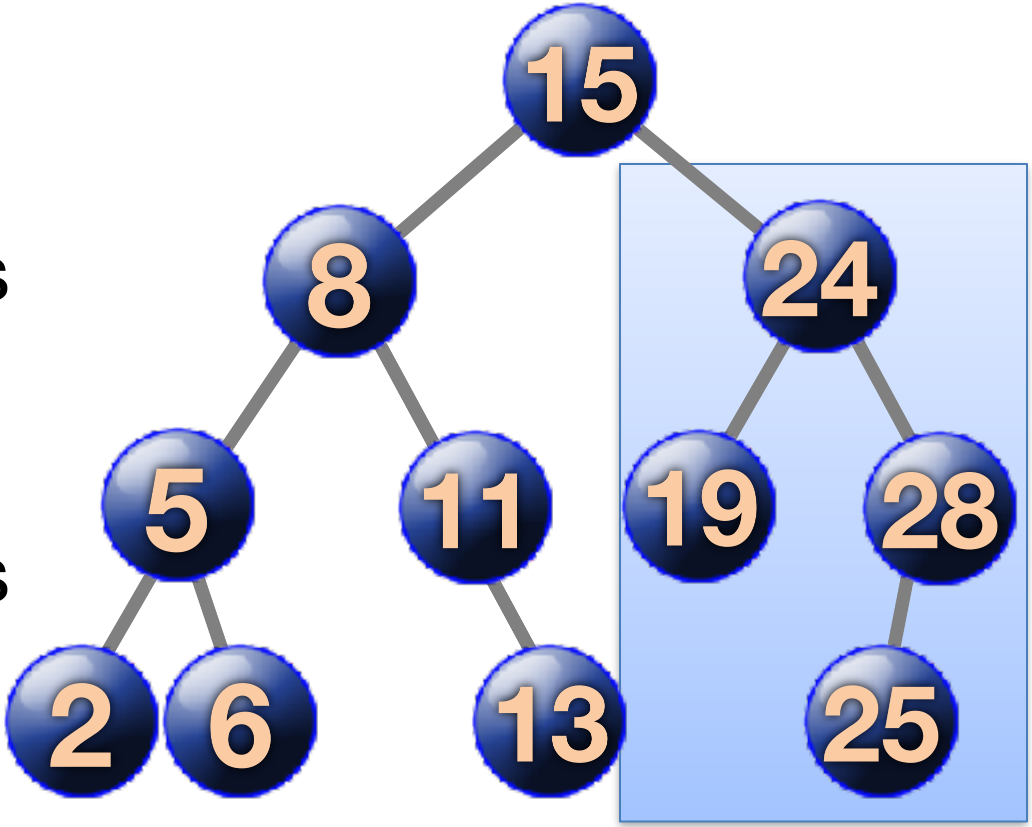
Binary Search Tree

- each node is greater than every node in its left subtree



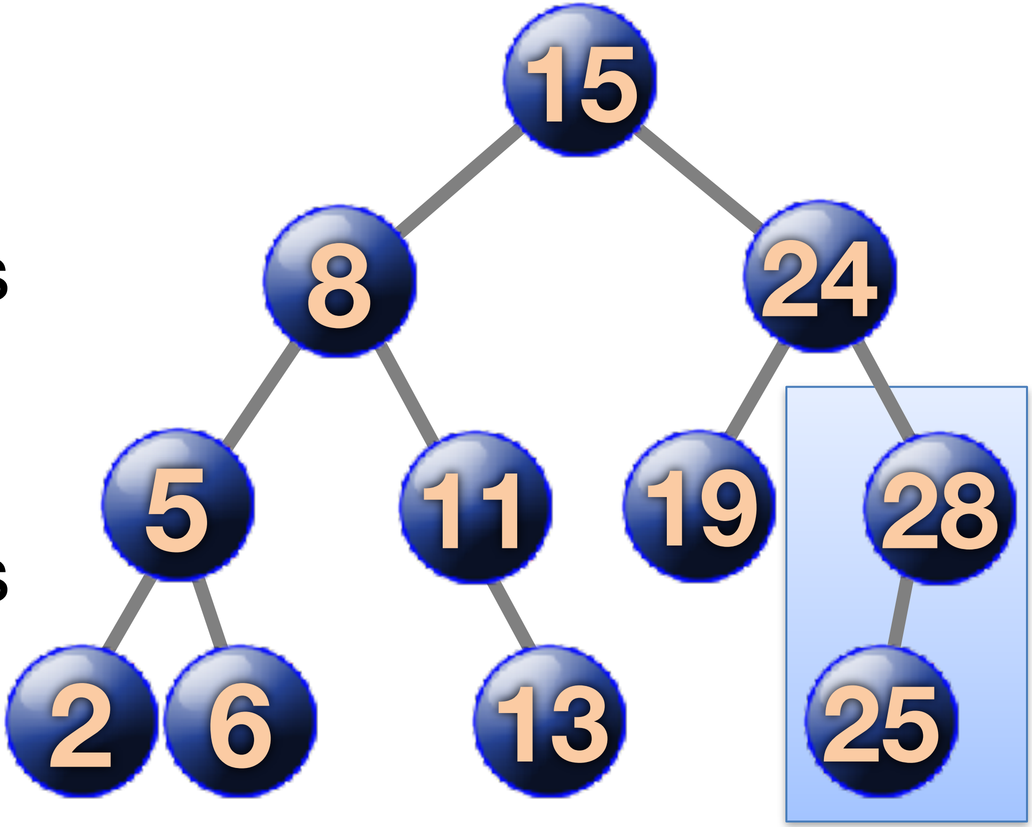
Binary Search Tree

- each node is greater than every node in its left subtree
- each node is less than every node in its right subtree



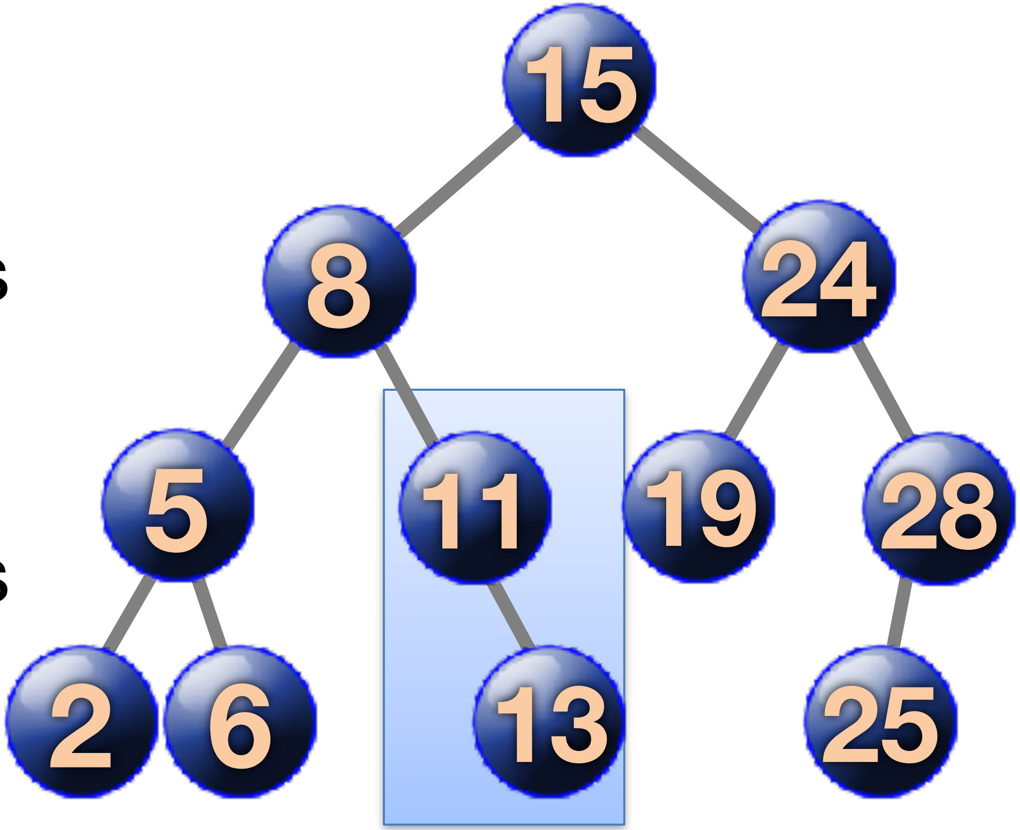
Binary Search Tree

- each node is greater than every node in its left subtree
- each node is less than every node in its right subtree



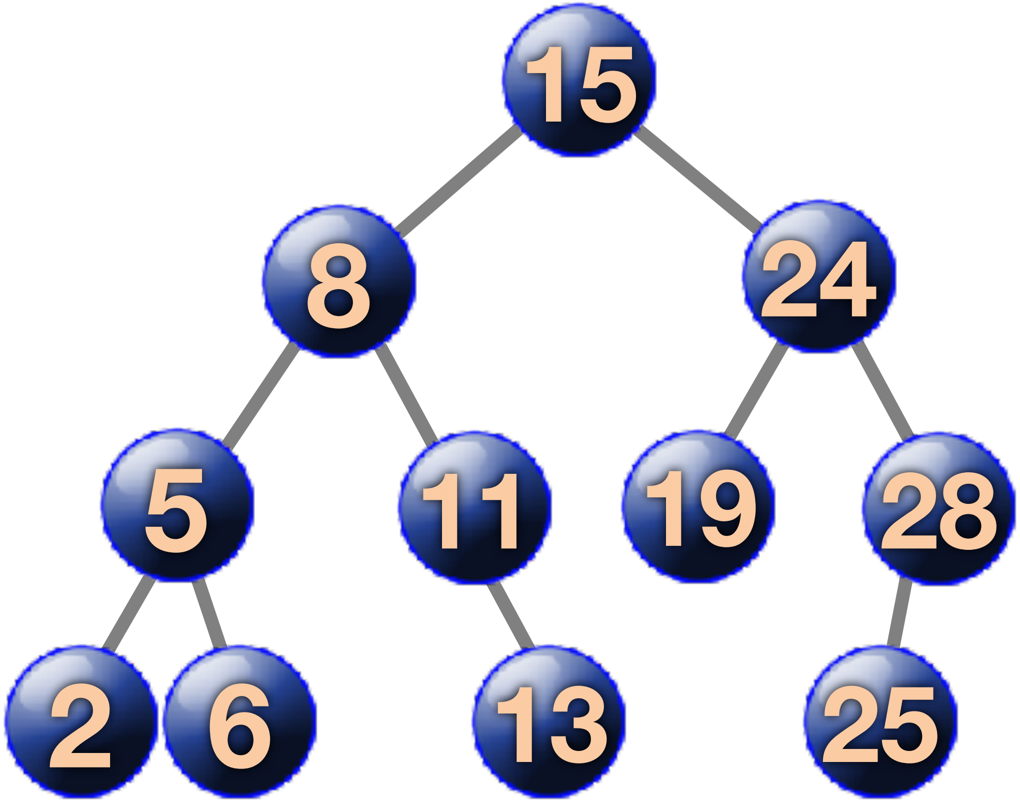
Binary Search Tree

- each node is greater than every node in its left subtree
- each node is less than every node in its right subtree



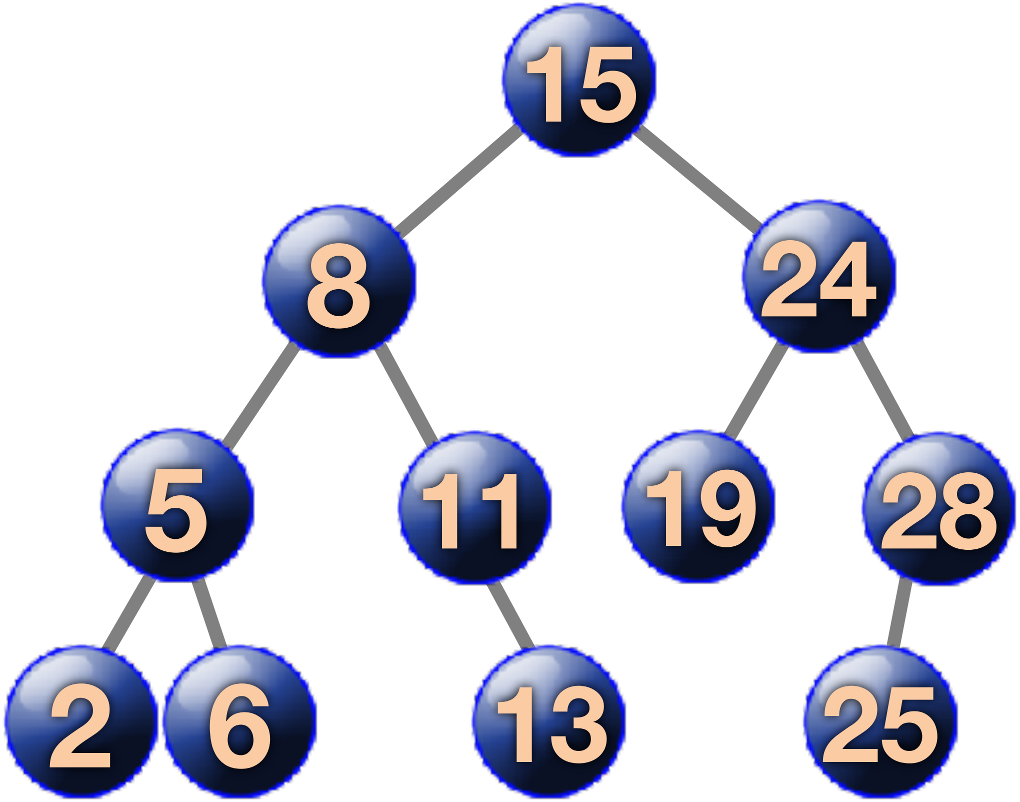
BST Operations

- Insert
- Find
- Delete
- Get_size
- Traversals



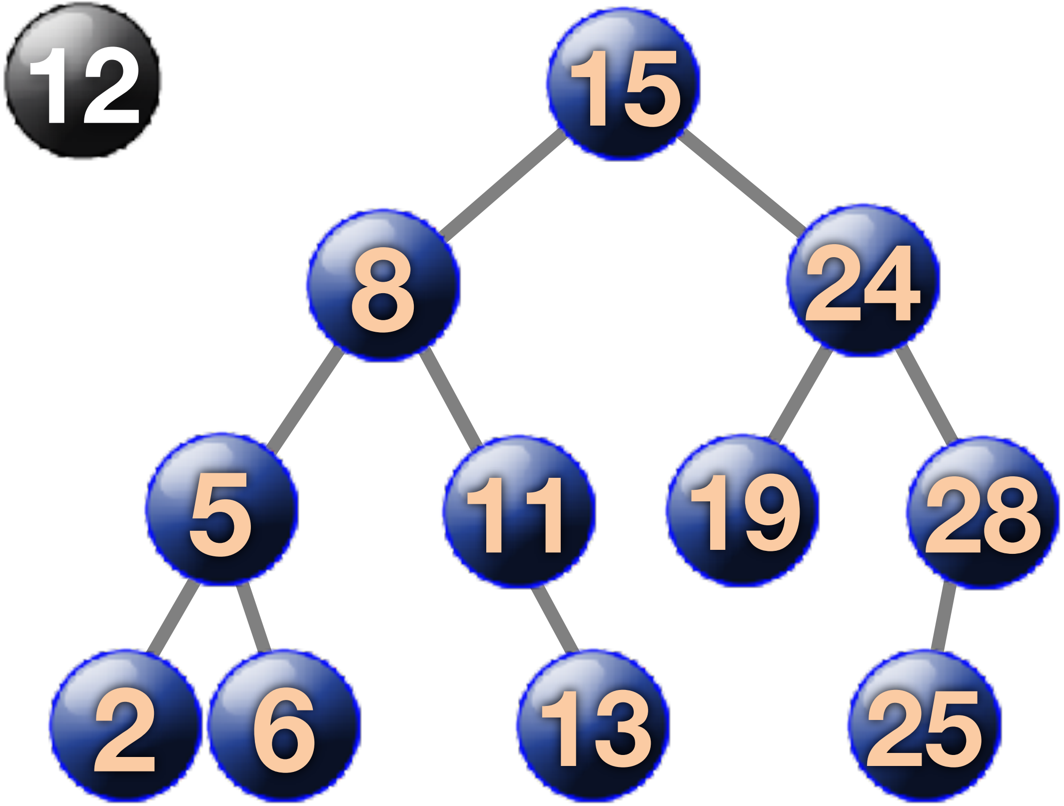
BST Insert

- Start at root
- Always insert as a leaf



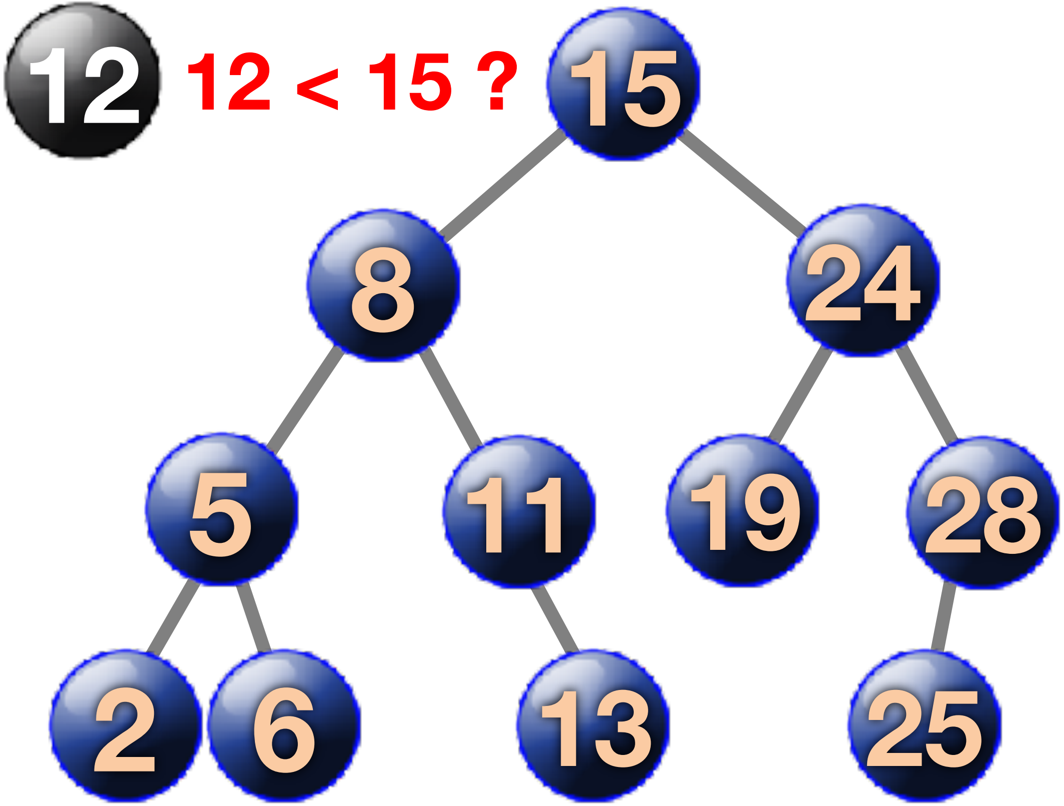
BST Insert

- Start at root
- Always insert as a leaf



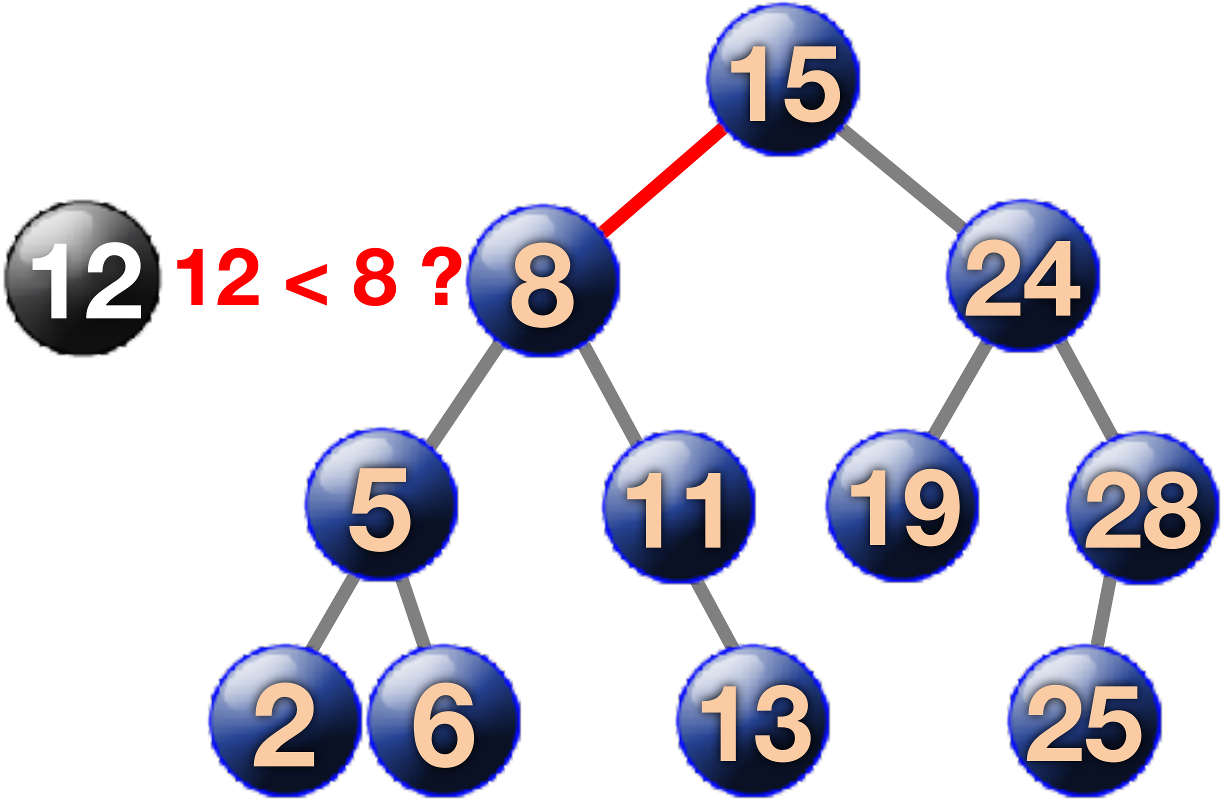
BST Insert

- Start at root
- Always insert as a leaf



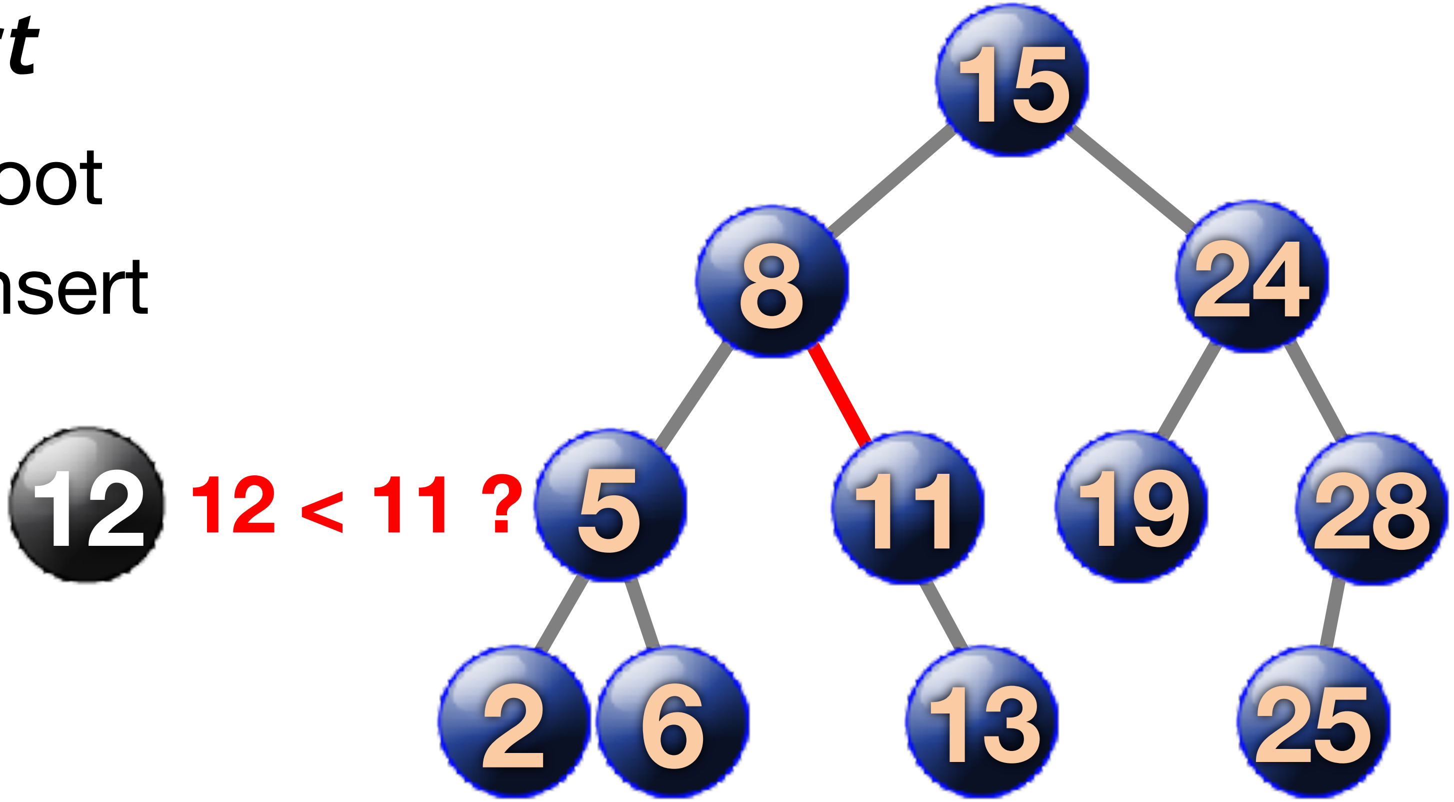
BST Insert

- Start at root
- Always insert as a leaf



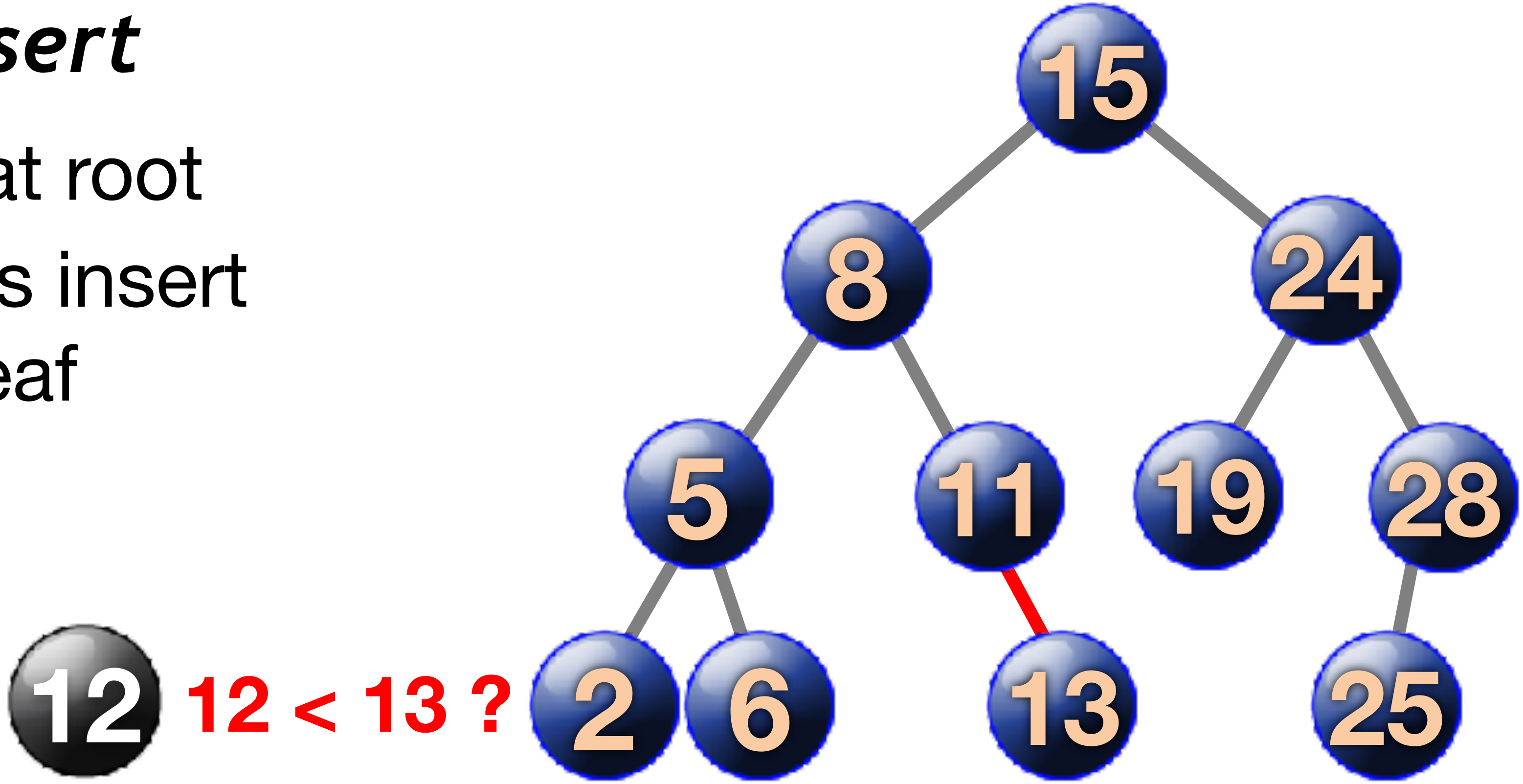
BST Insert

- Start at root
- Always insert as a leaf



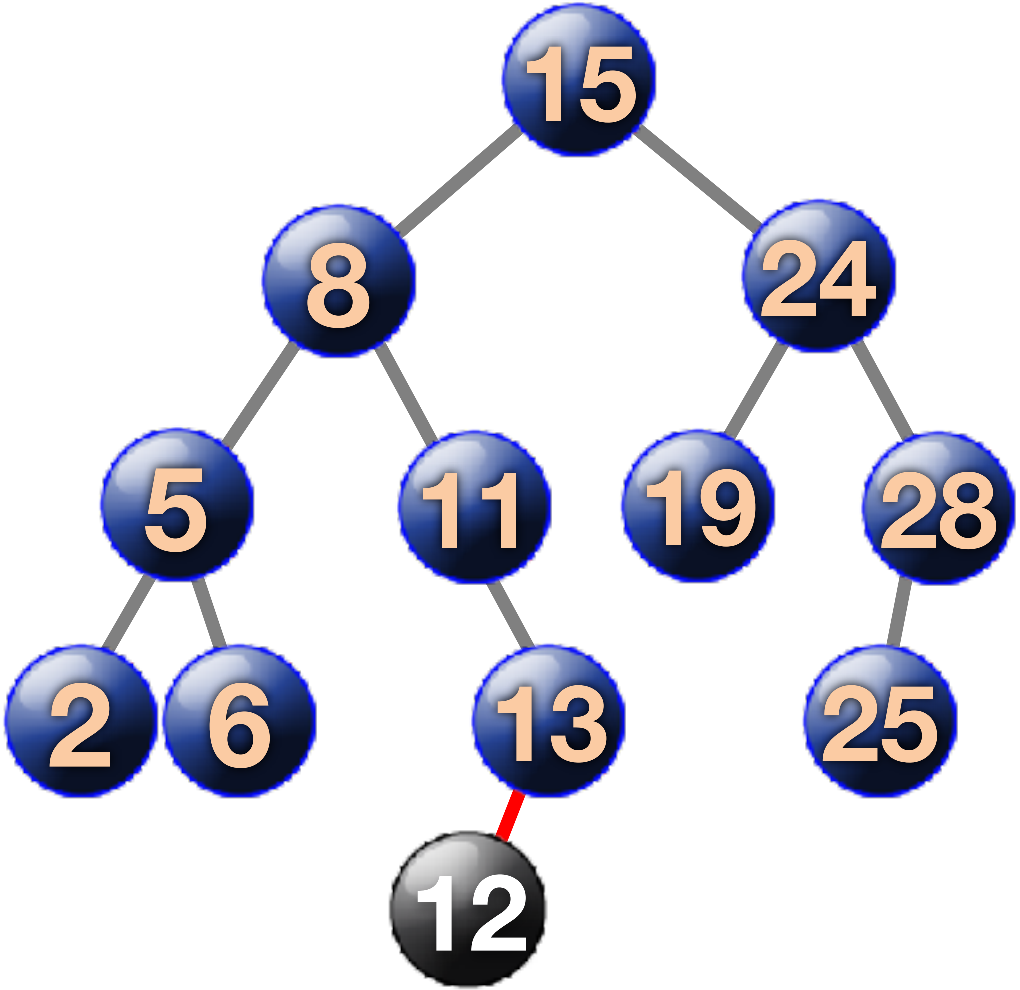
BST Insert

- Start at root
- Always insert as a leaf



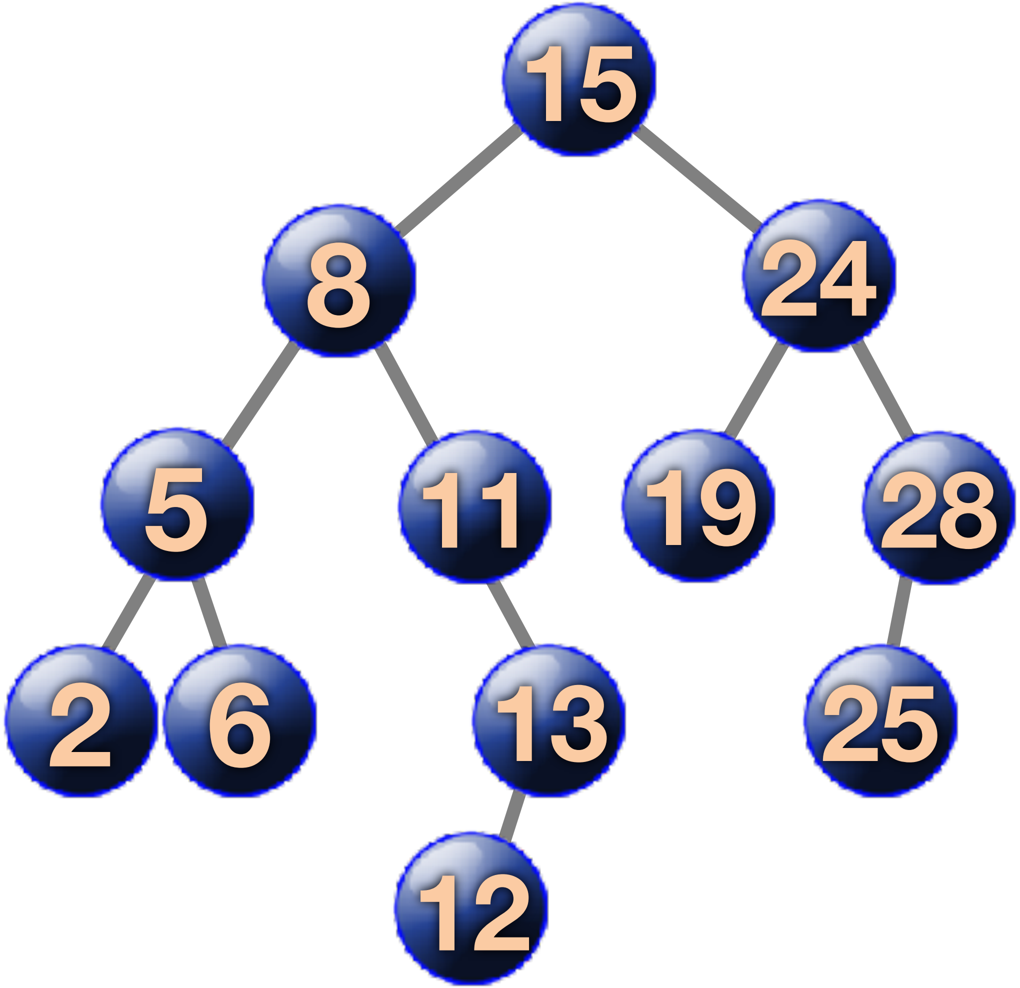
BST Insert

- Start at root
- Always insert as a leaf



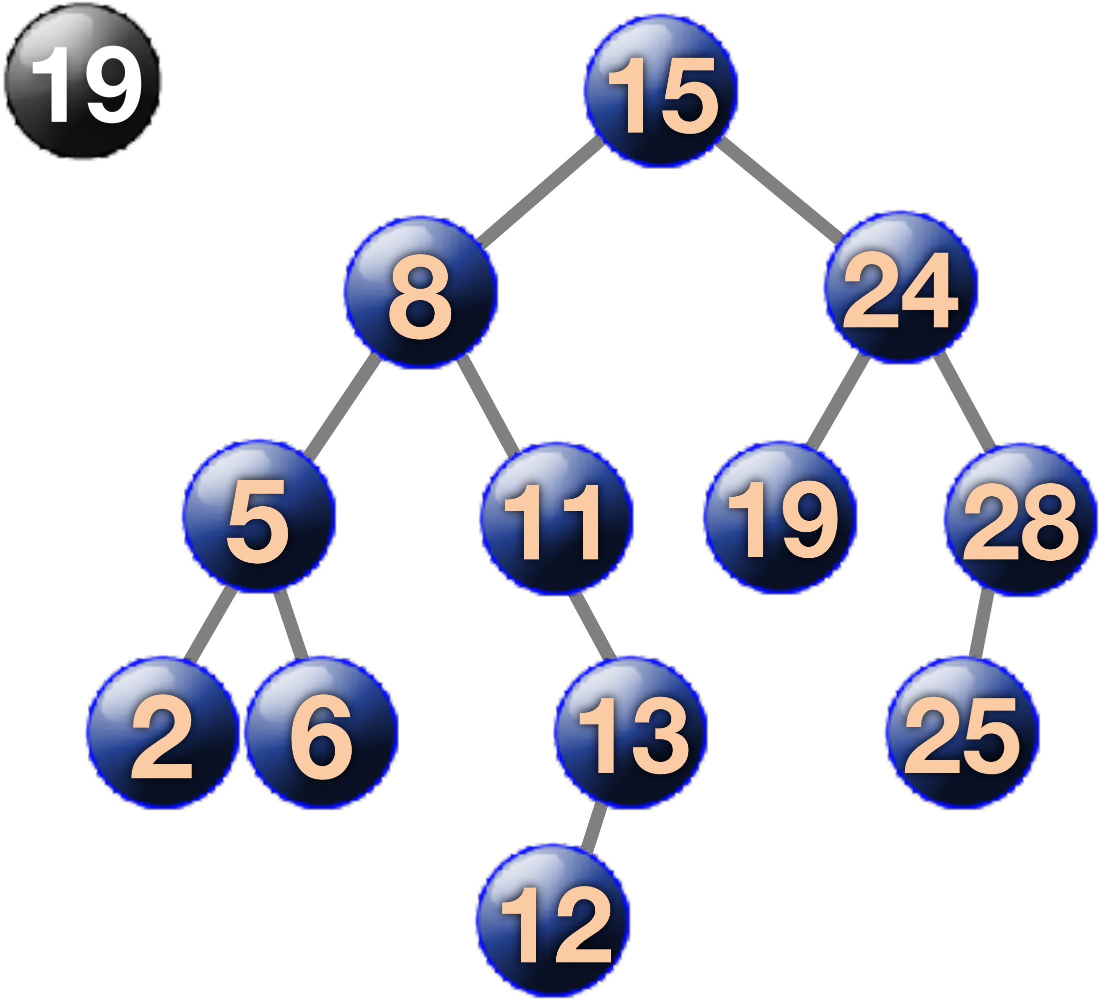
BST Find

- Start at root



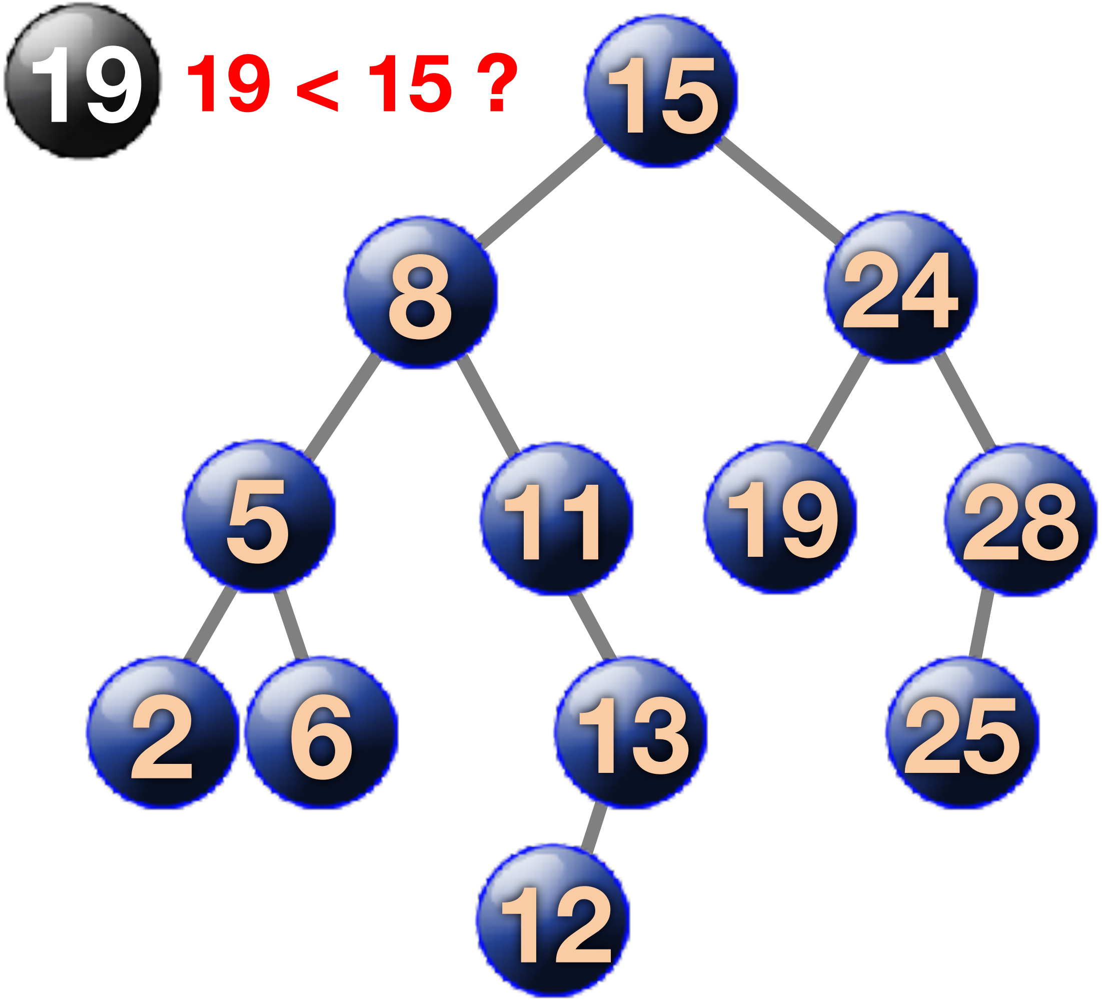
BST Find

- Start at root



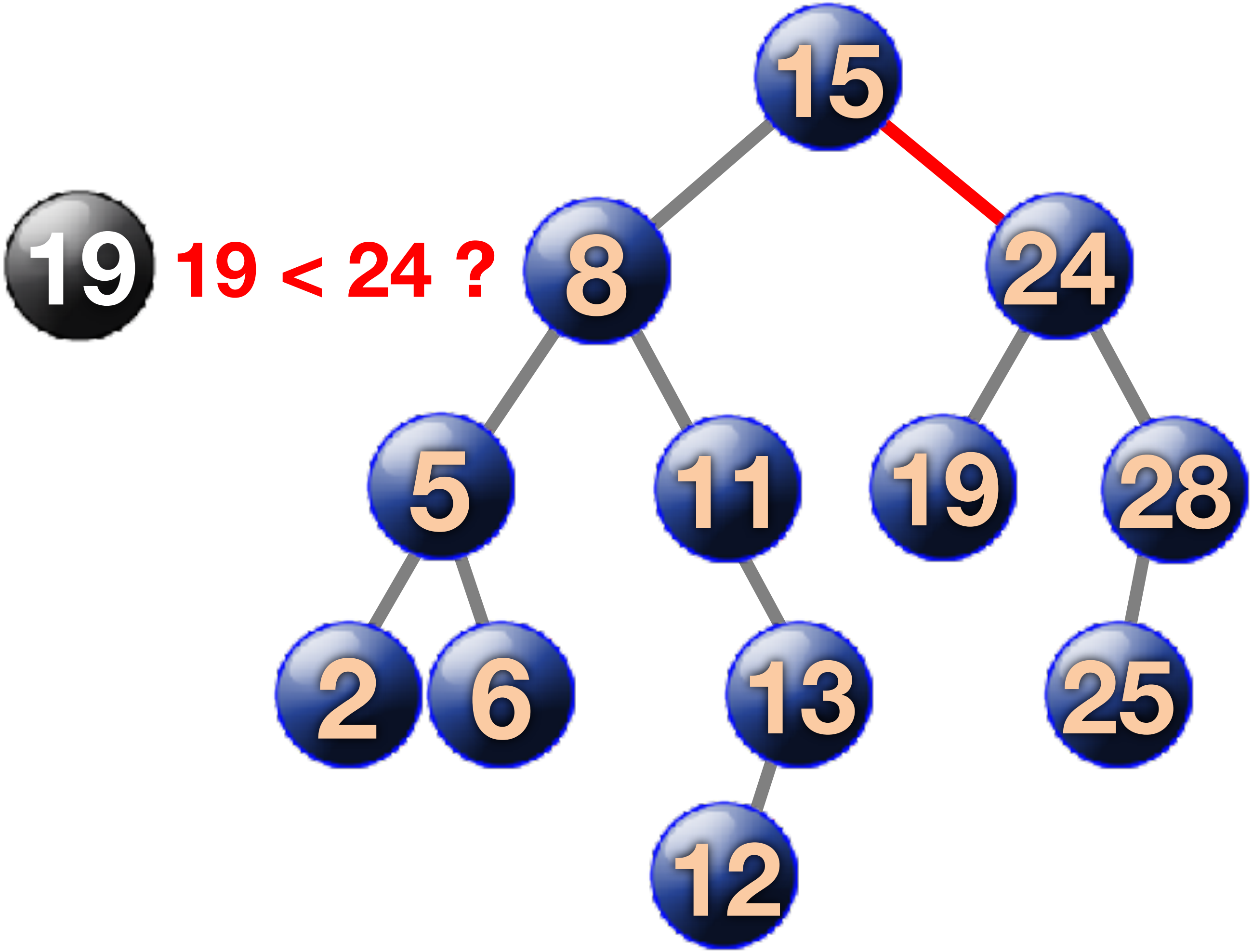
BST Find

- Start at root



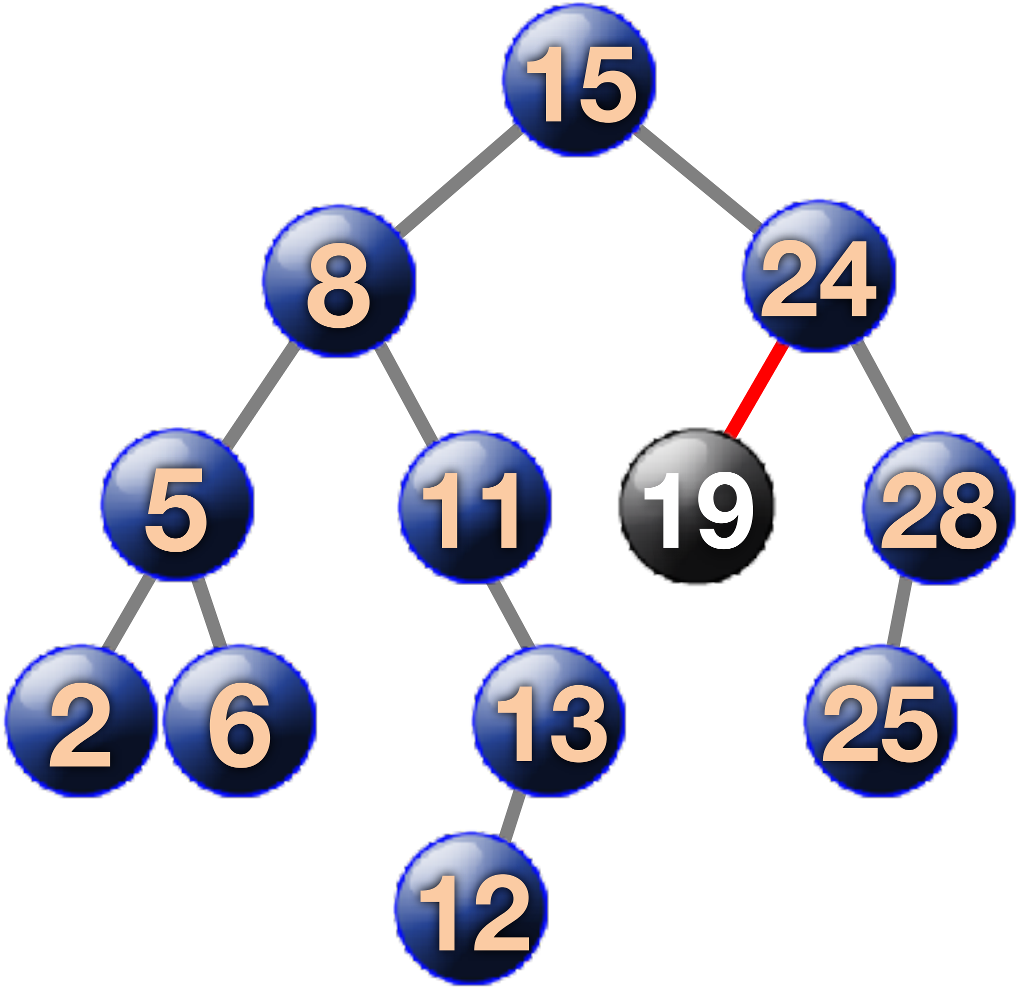
BST Find

- Start at root



BST Find

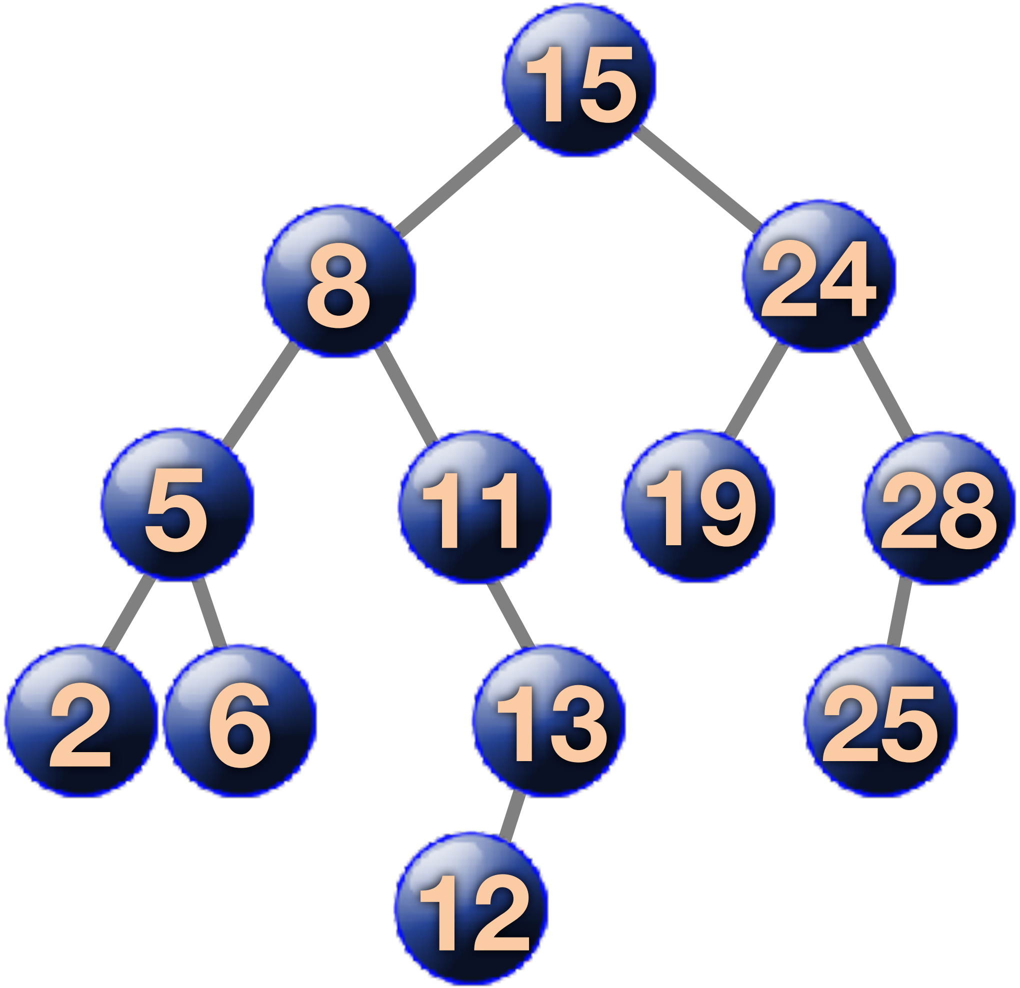
- Start at root
- Return the data if found, or False if not found



BST Delete

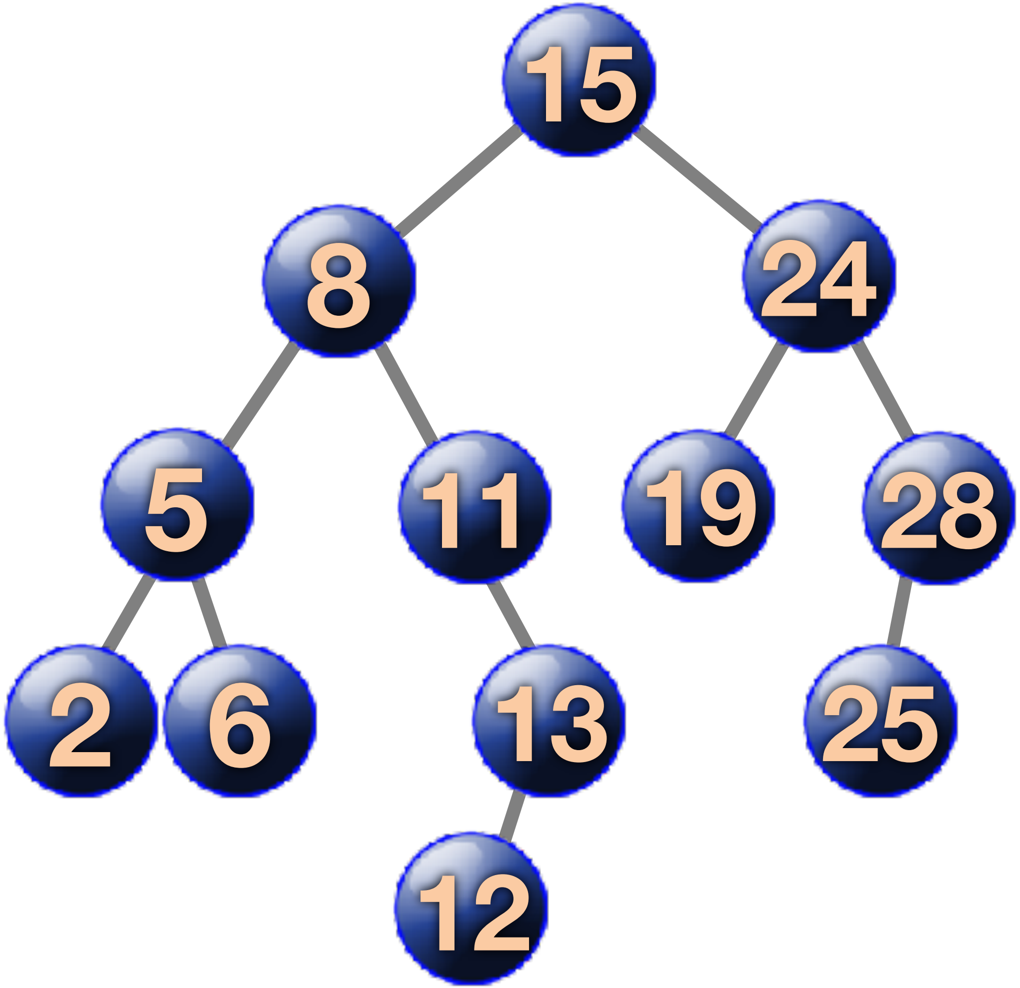
3 possible cases:

- leaf node
- 1 child
- 2 children



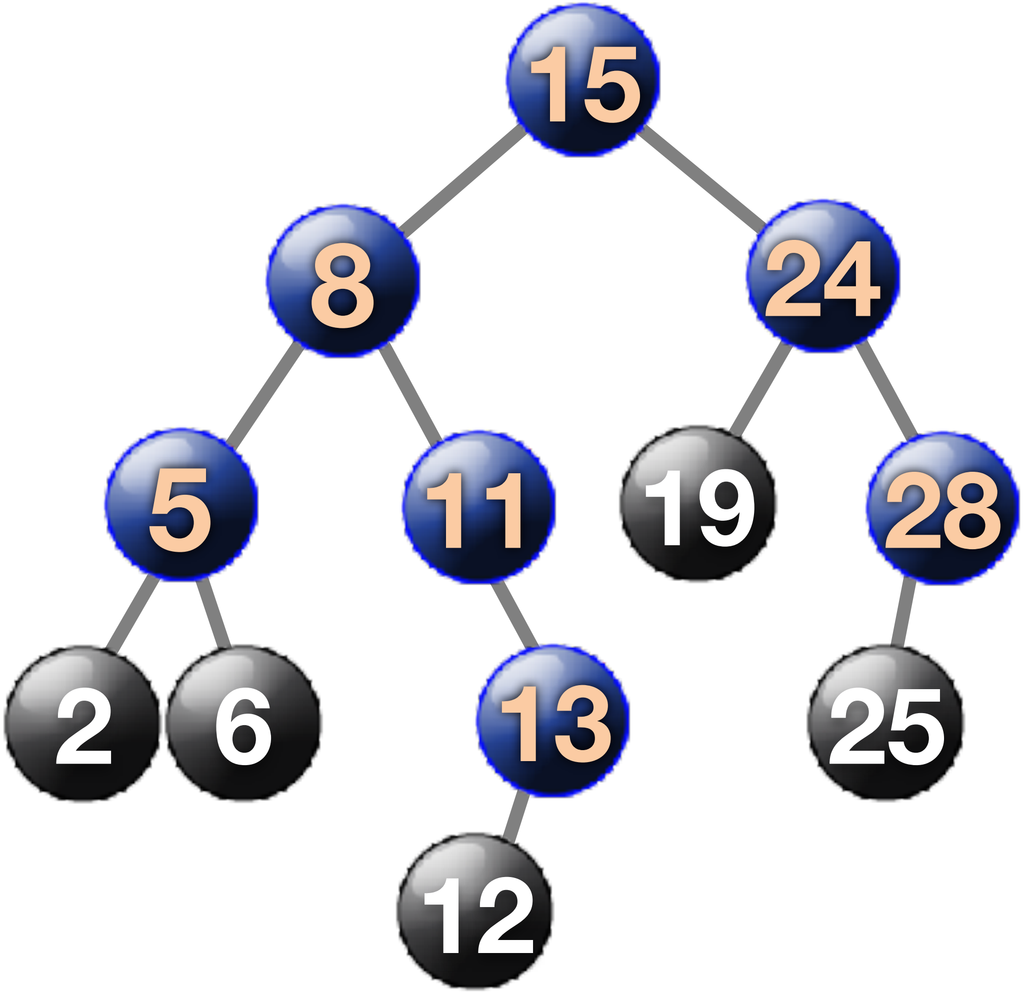
BST Delete

- leaf node



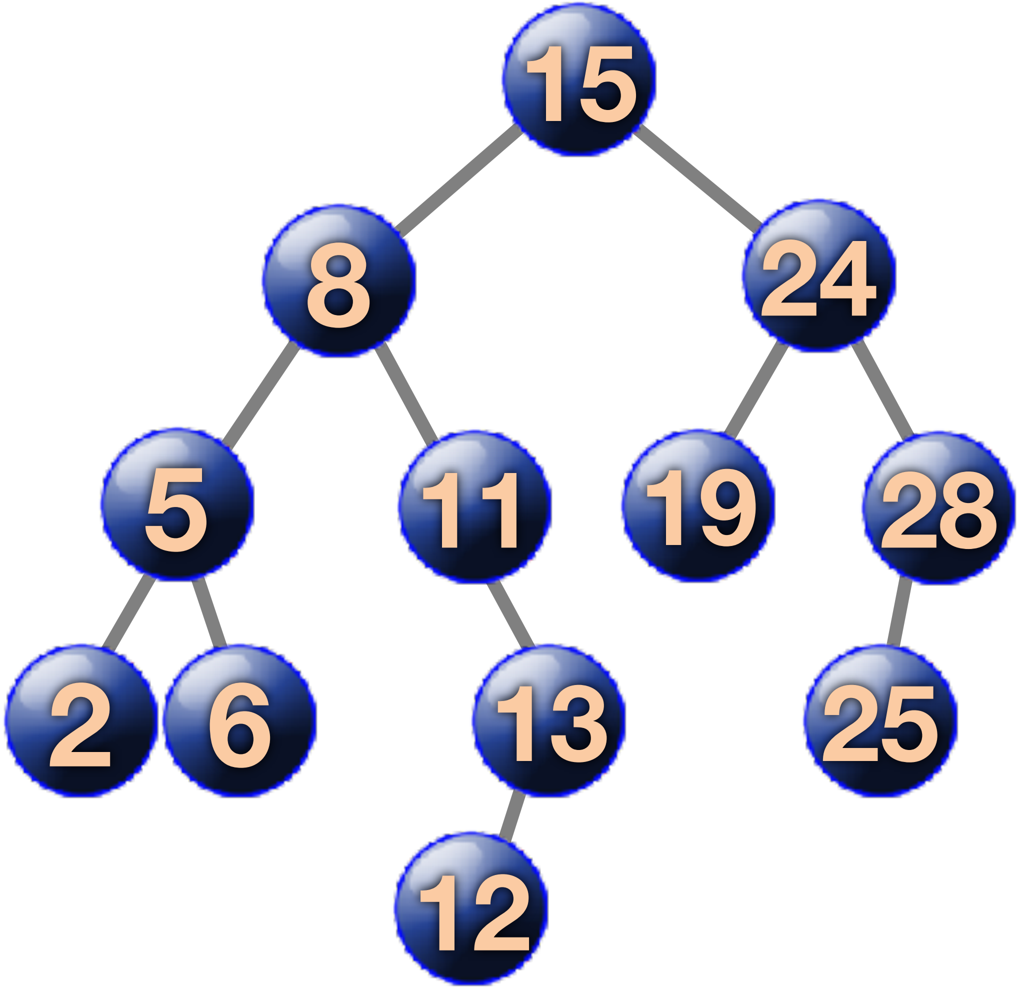
BST Delete

- leaf node
  - just delete the leaf node



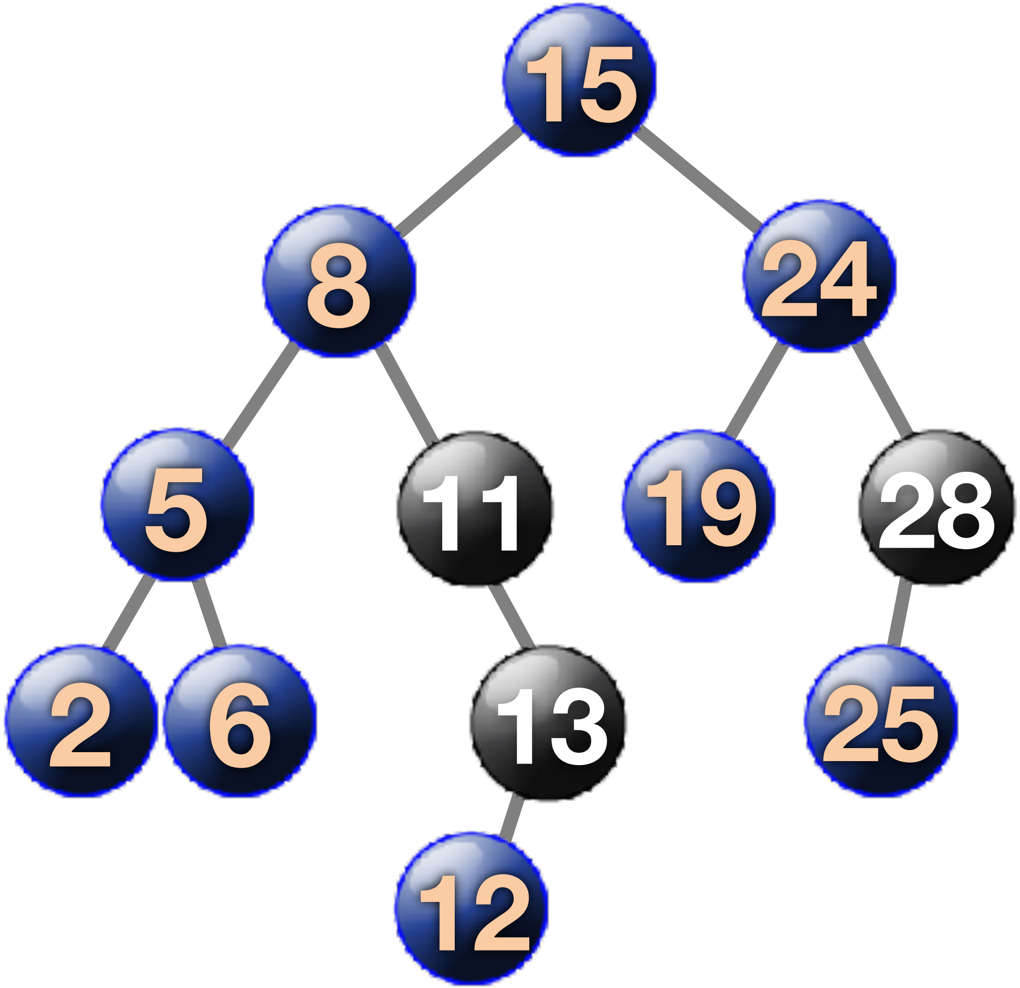
BST Delete

- 1 child



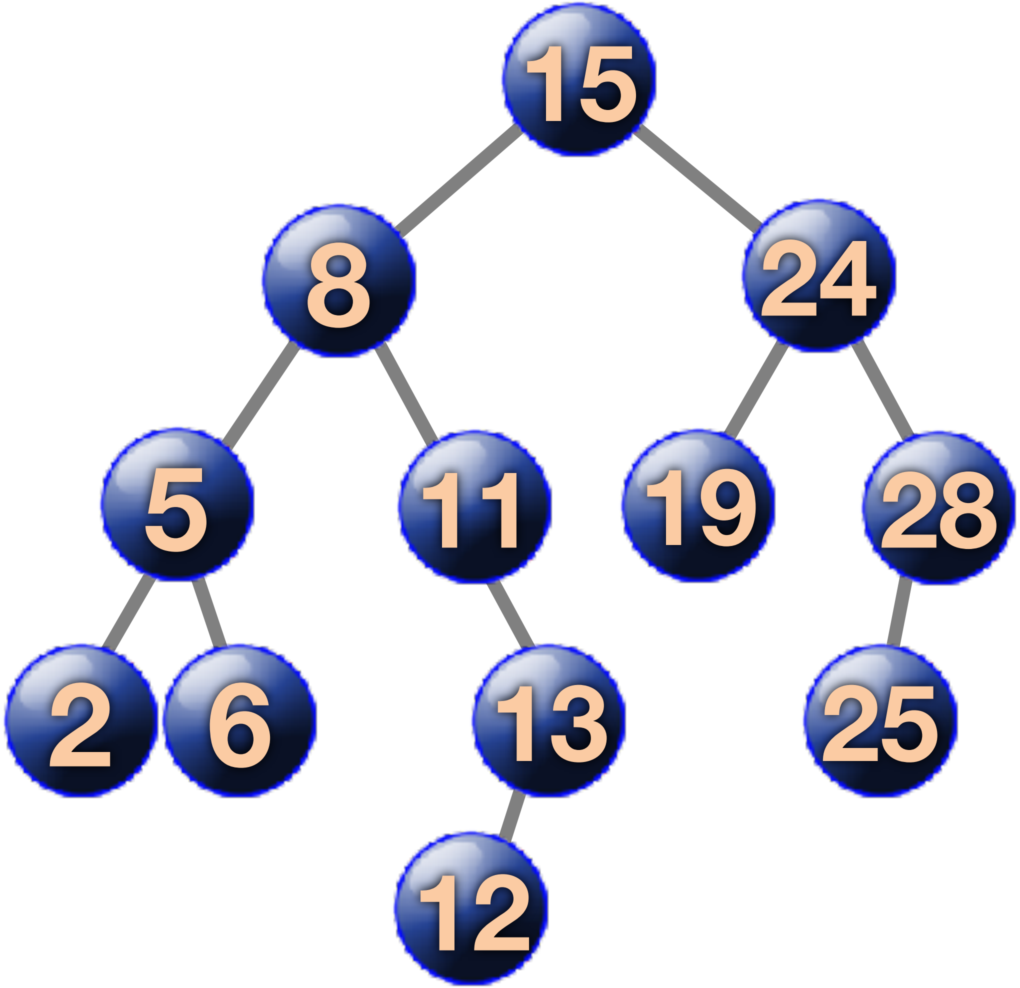
BST Delete

- 1 child
  - promote the child to the target node's position



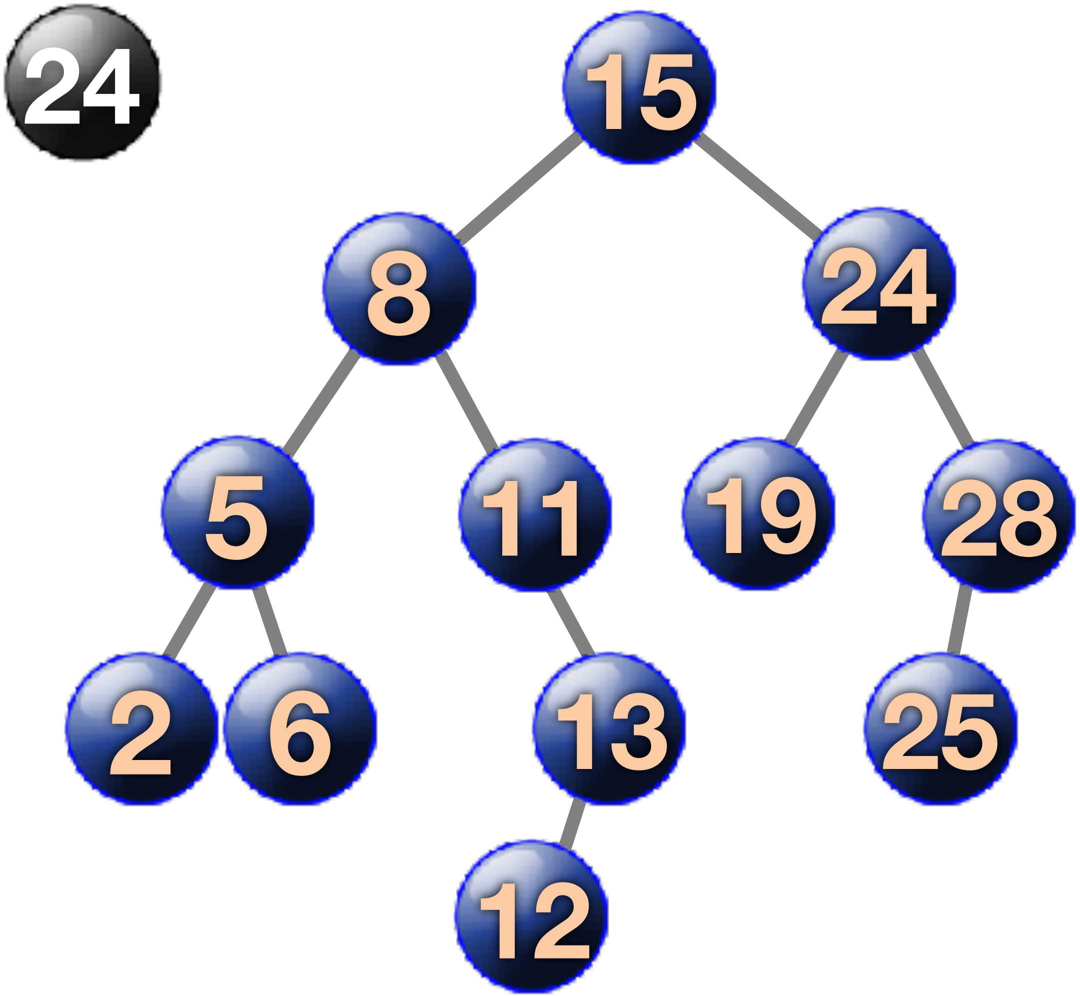
BST Delete

- 2 children



BST Delete

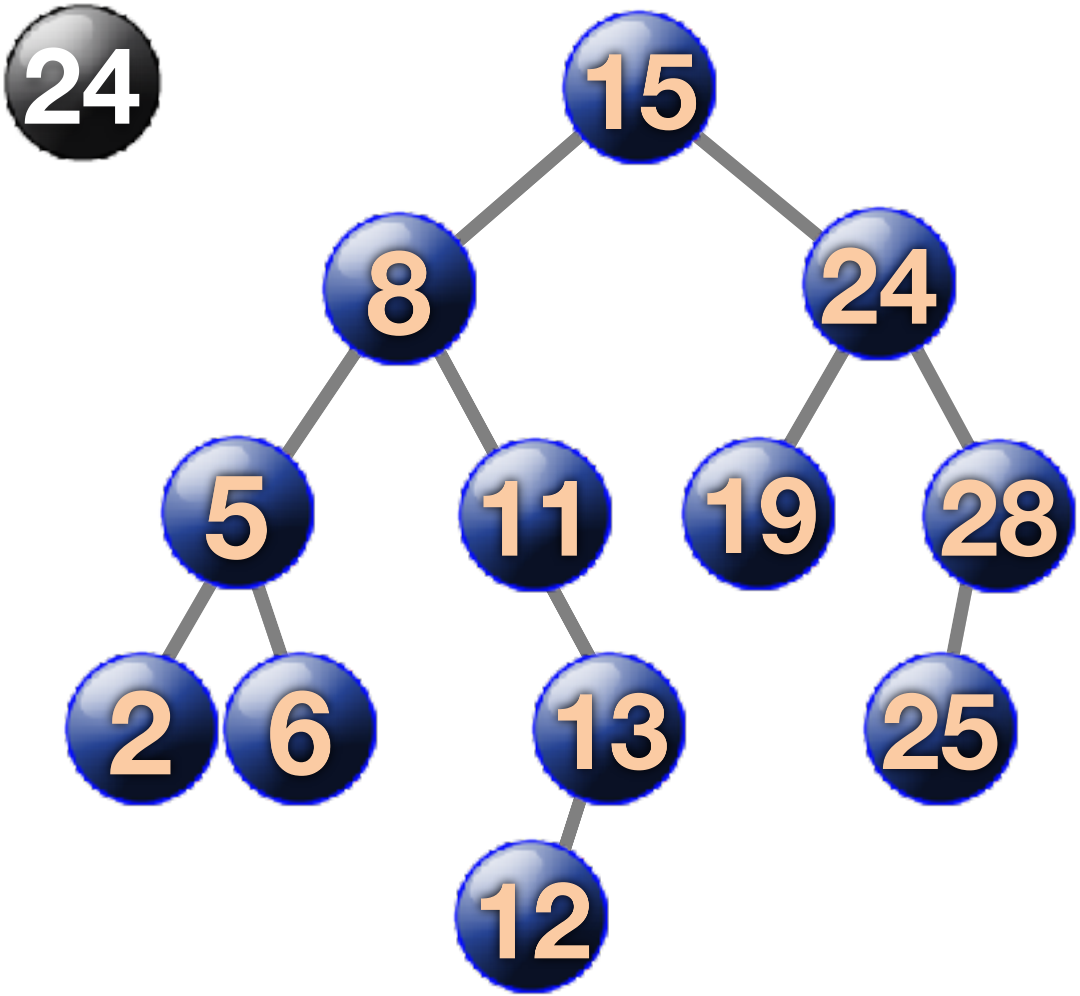
- 2 children



BST Delete

- 2 children

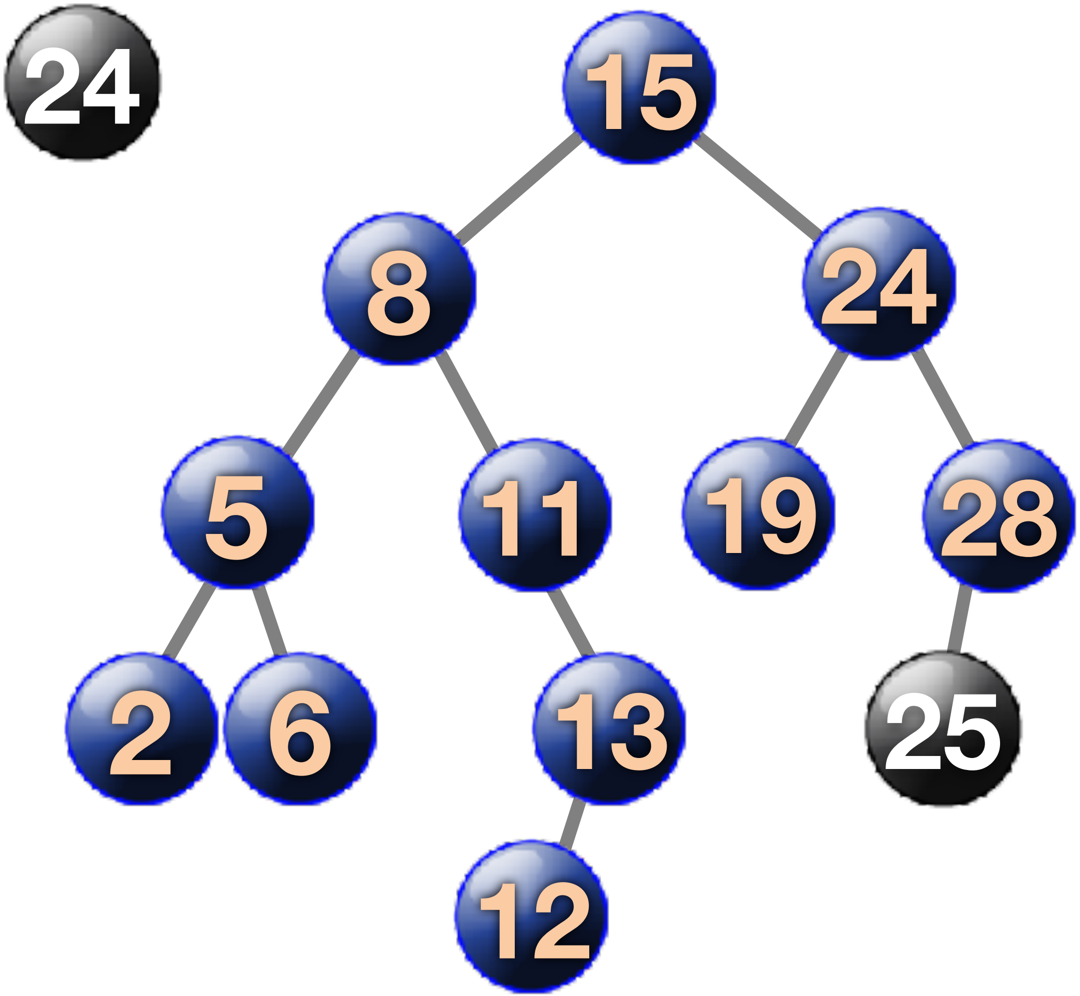
Find the next higher node



BST Delete

- 2 children

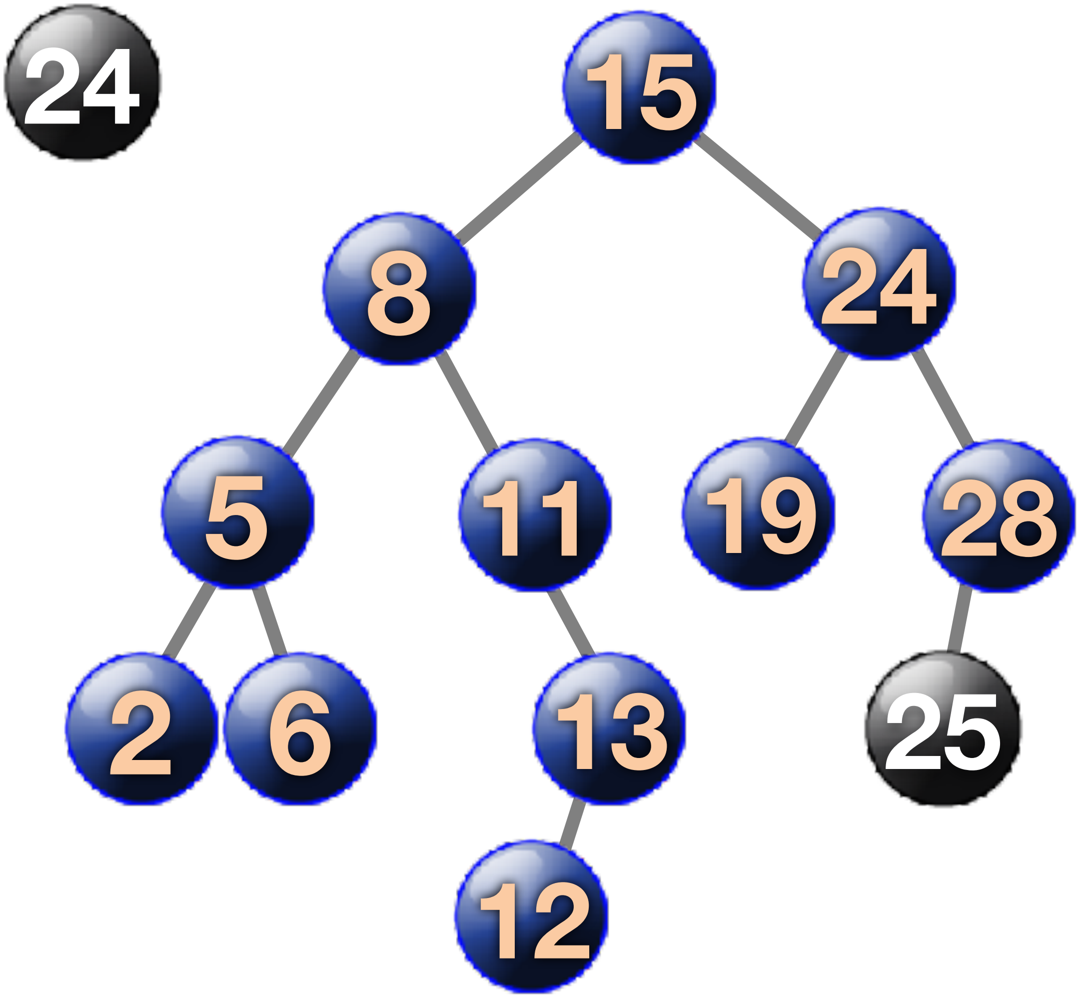
Find the next higher node



BST Delete

- 2 children

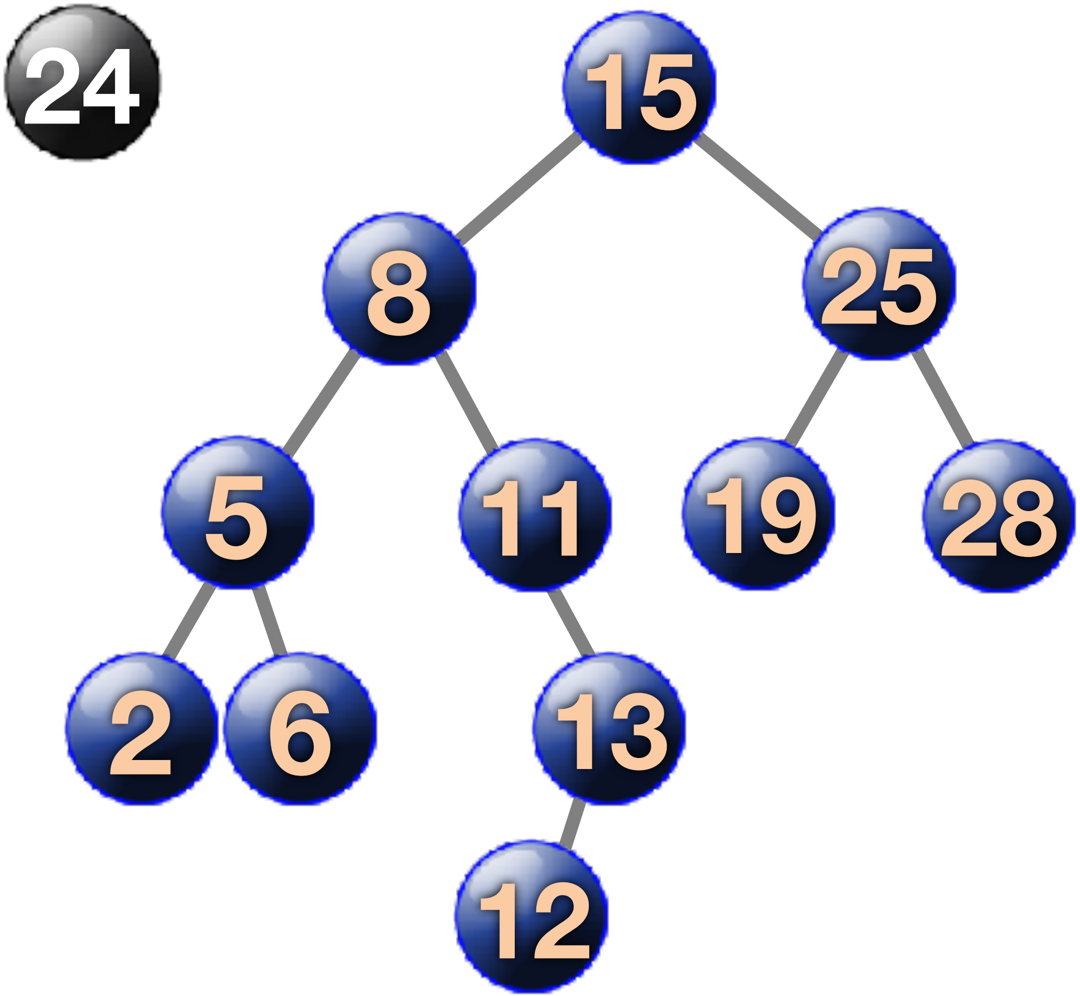
Find the next higher node, change 24 to 25, then delete node 25



BST Delete

- 2 children

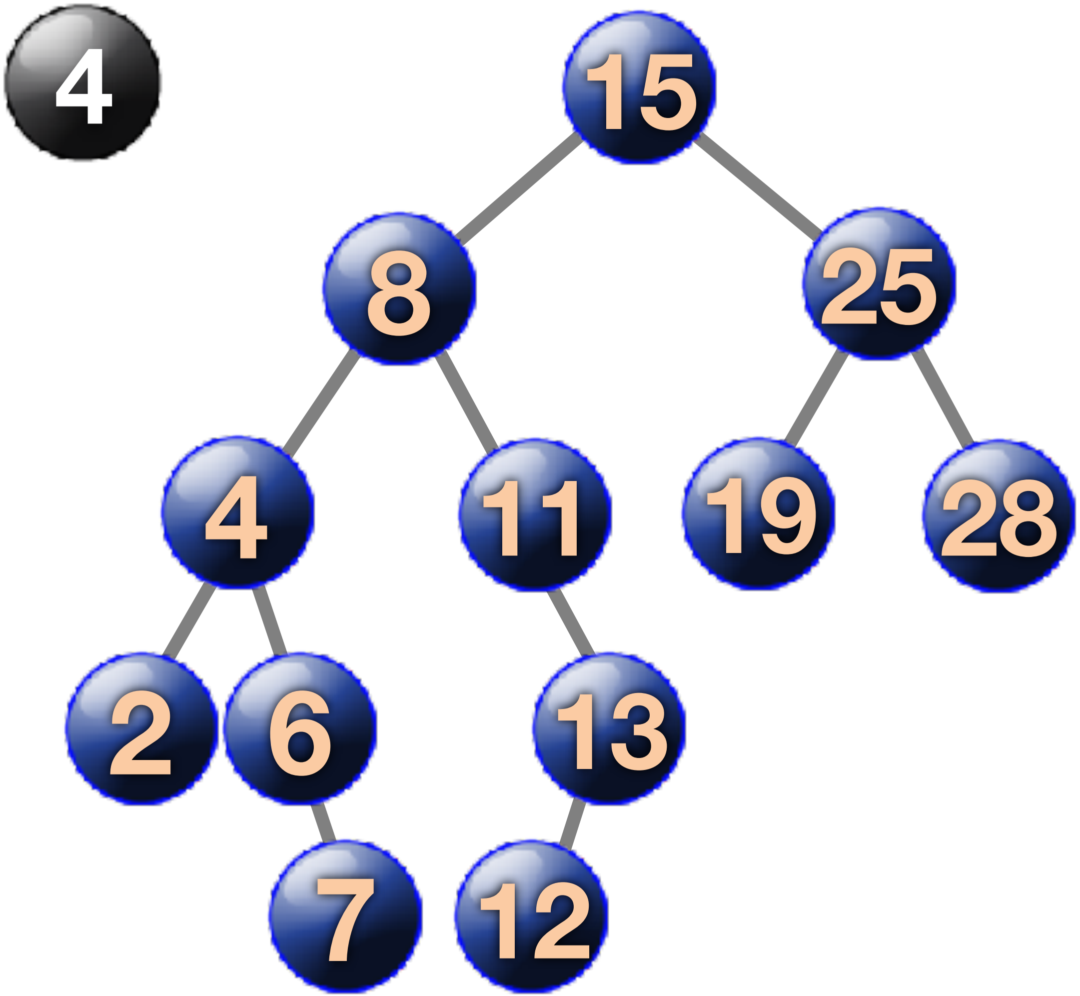
Find the next
higher node,
change 24 to 25, then
delete node 25



BST Delete

- 2 children

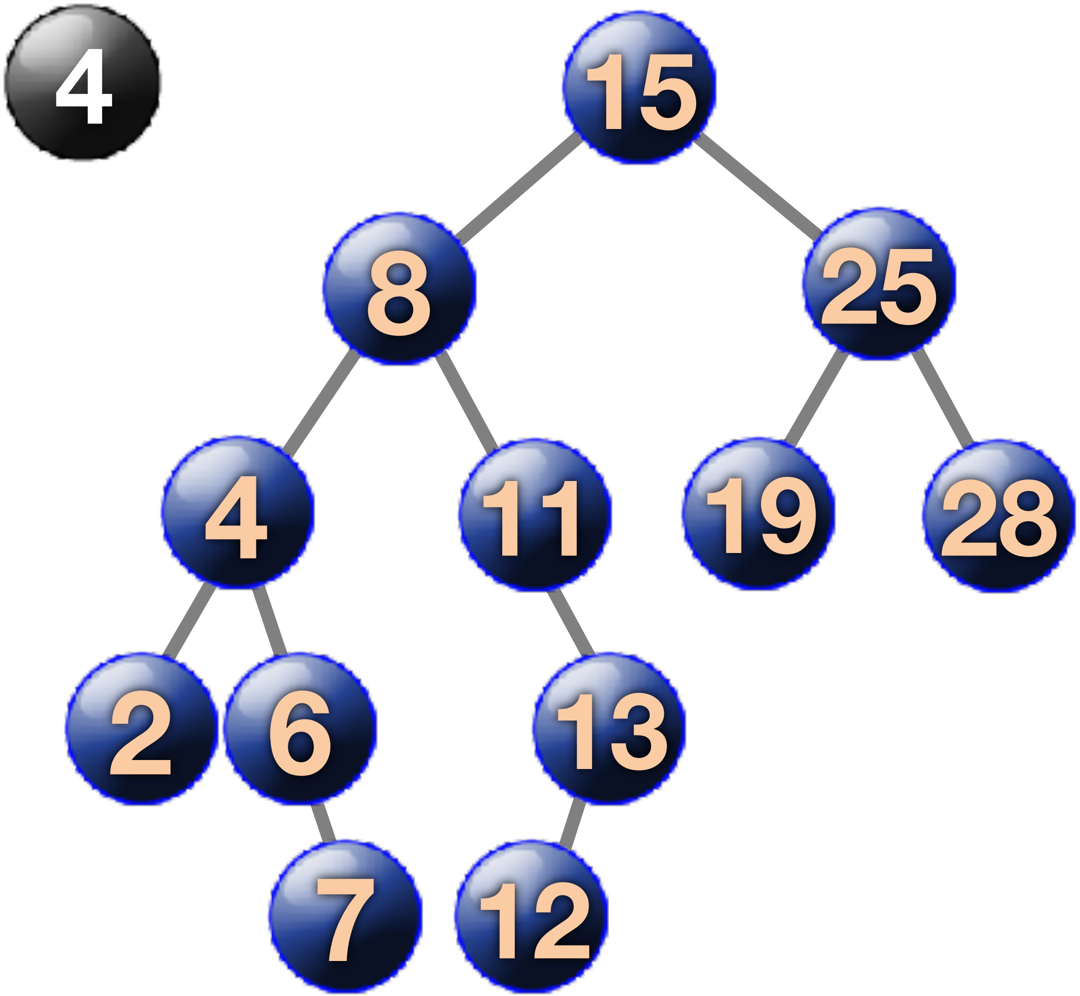
Find the next higher node,



BST Delete

- 2 children

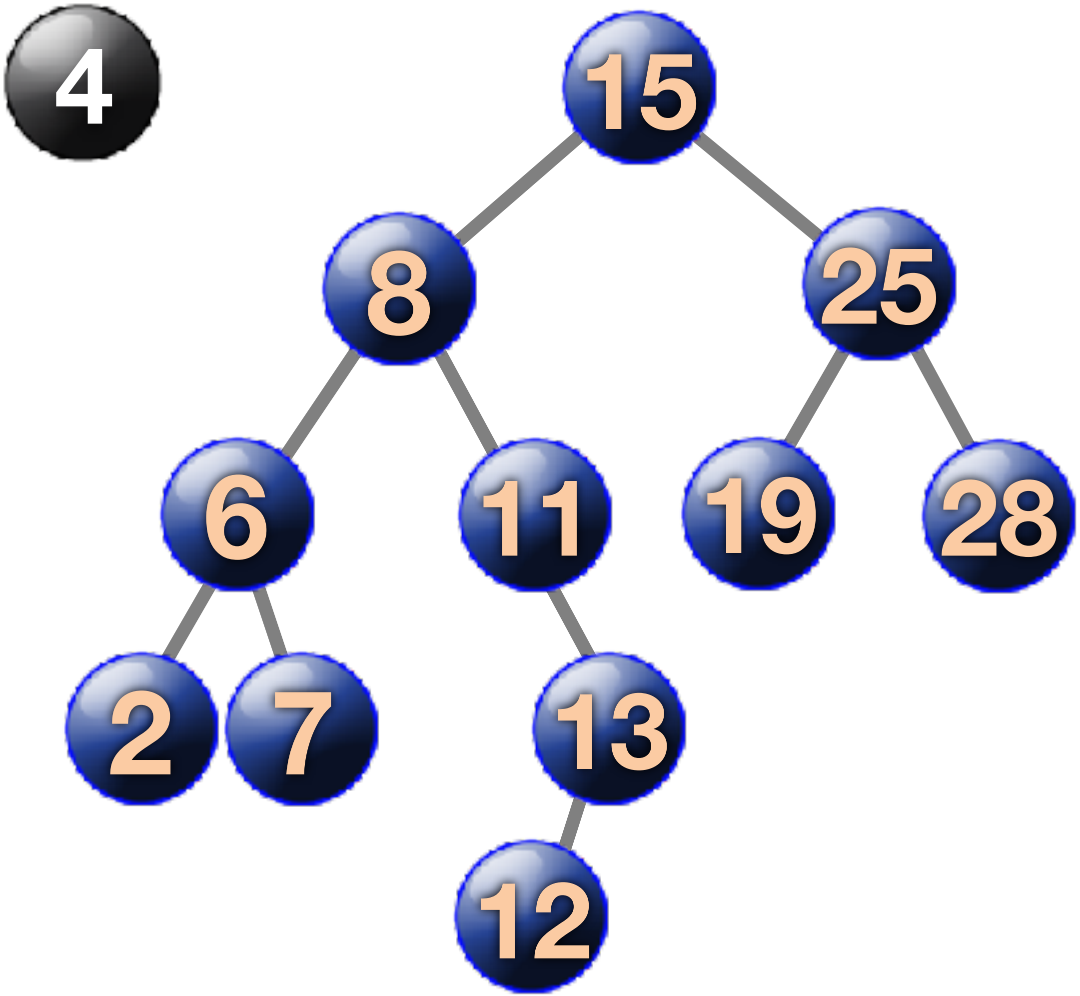
Find the next higher node, change 4 to 6, then delete node 6



BST Delete

- 2 children

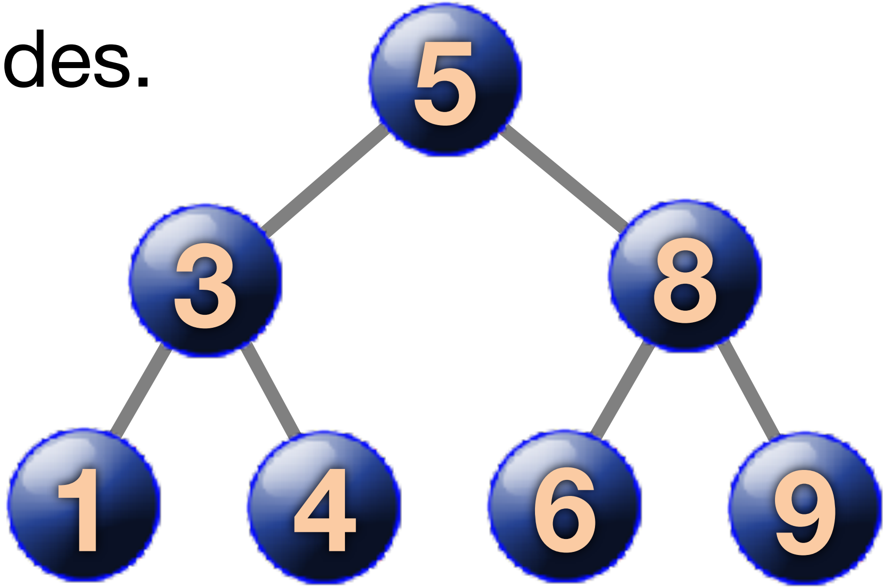
Find the next higher node, change 4 to 6, then delete node 6



Get_size

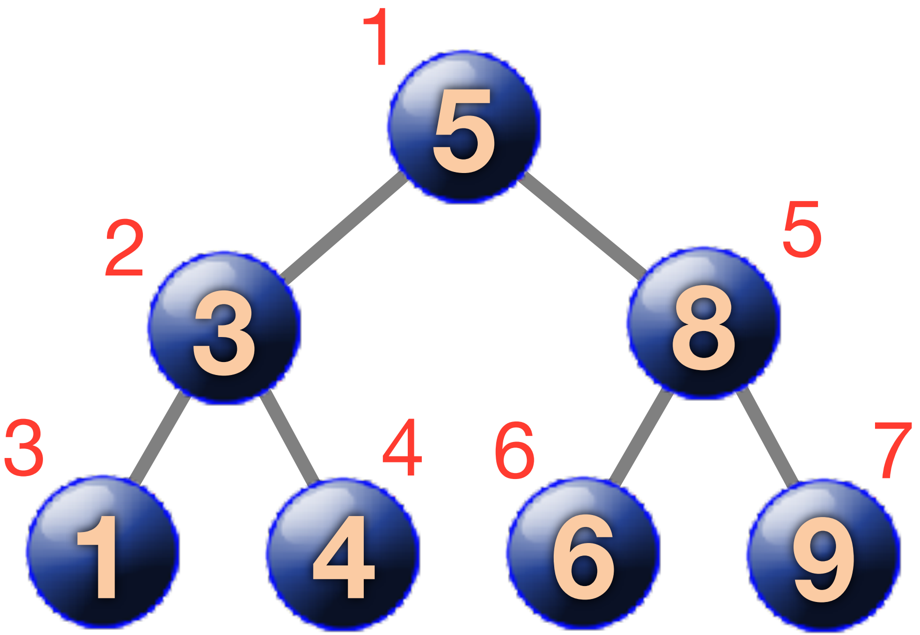
Returns number of nodes.
Works recursively

size = 1
+ size(left subtree)
+ size(right subtree)



Preorder Traversal

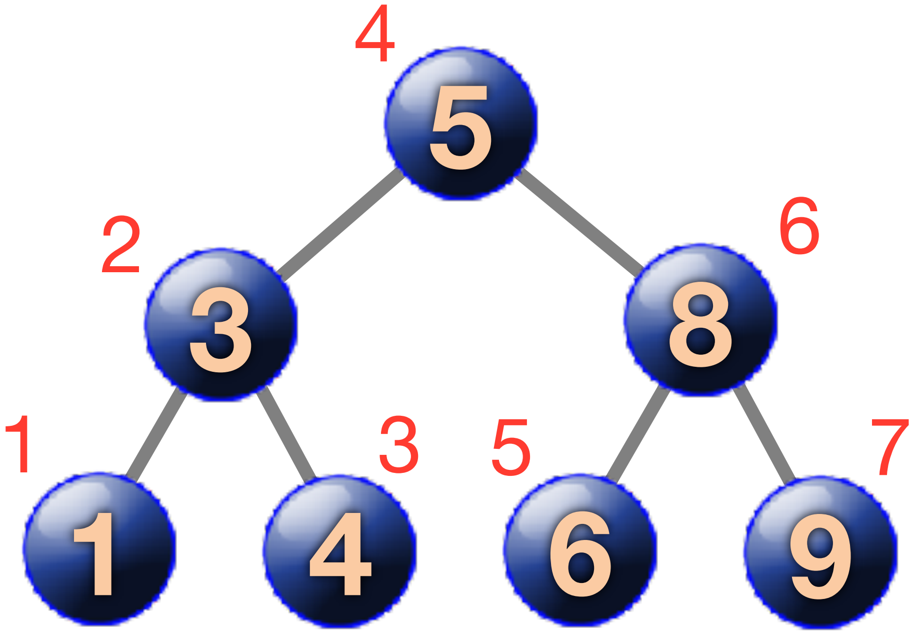
Visit root before visiting the root's subtrees.



Inorder Traversal

Visit root between visiting the root's subtrees.

Gives values in sorted order.



Advantages of Binary Search Trees?

Advantages of Binary Search Trees?

Because trees use recursion for most operations, they are fairly easy to implement.

Advantages of Binary Search Trees?

SPEED

Advantages of Binary Search Trees?

SPEED

Insert, Delete, Find in

$O(h) = O(\log n)$

Advantages of Binary Search Trees?

SPEED

In a balanced BST
with 10,000,000 nodes
Find takes 30 comparisons!

Advantages of Binary Search Trees?

SPEED

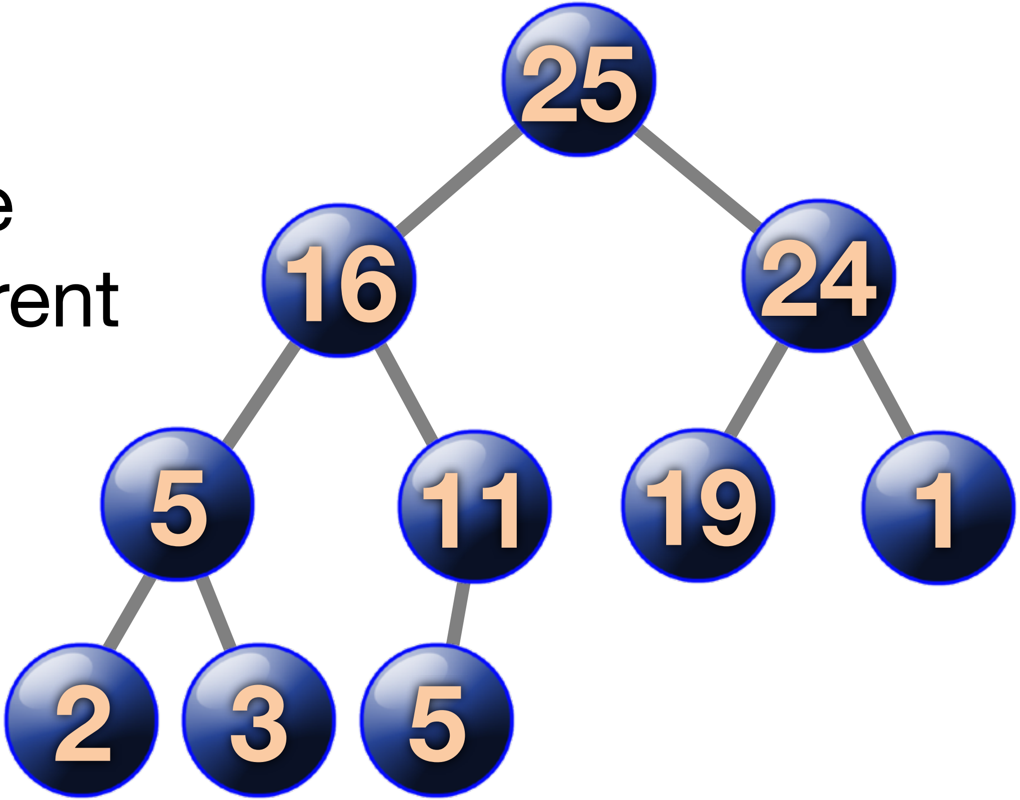
Why are trees so fast?

Because each comparison *cuts in half* the number of nodes to search.

Python MaxHeap

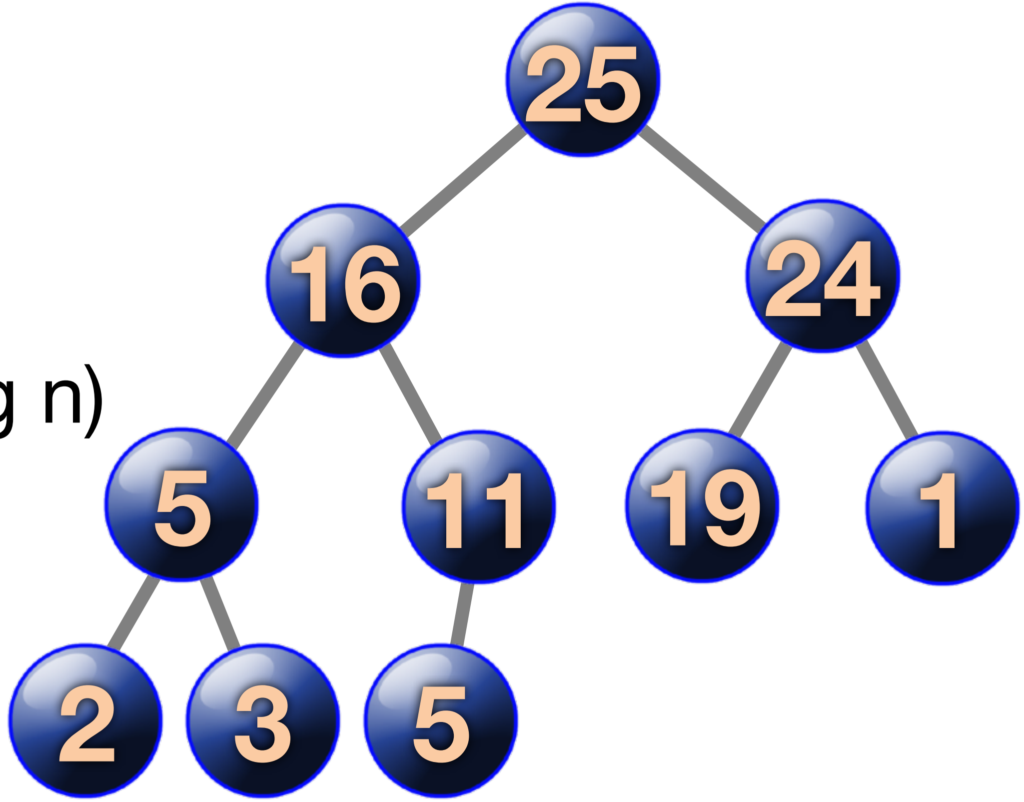
What is a MaxHeap?

- Complete Binary Tree
- Every node <= its parent

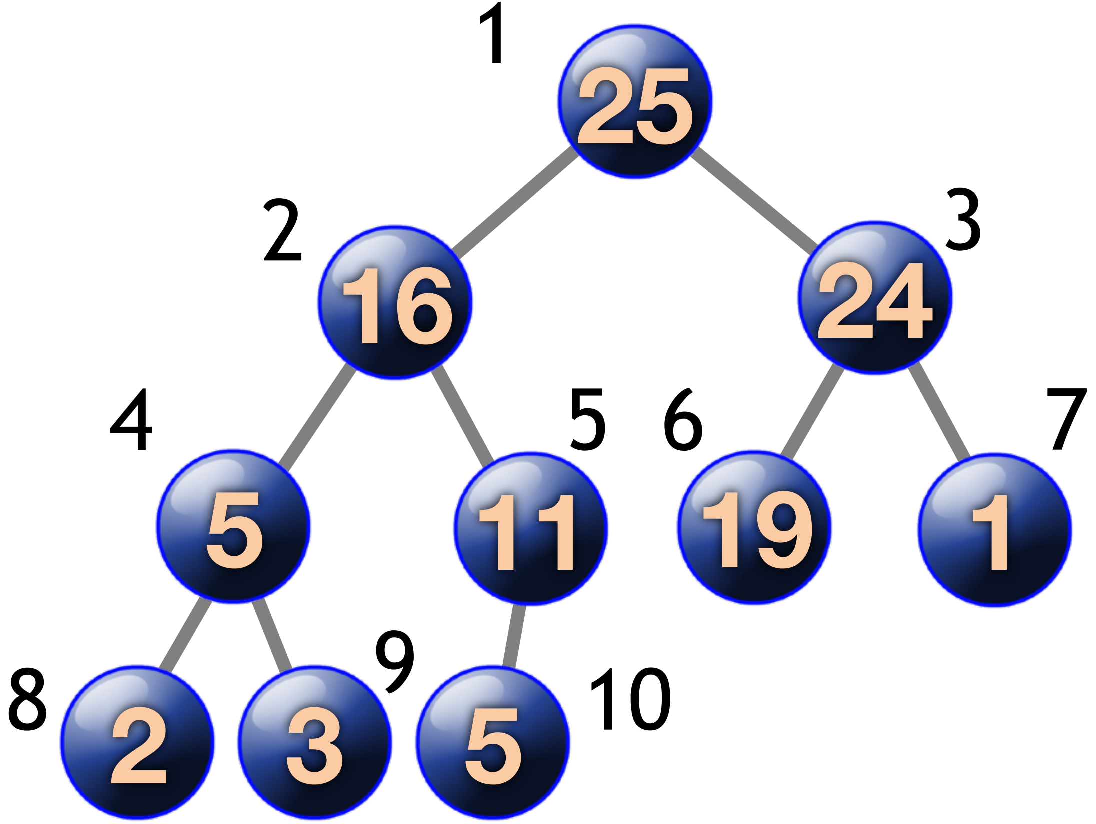


MaxHeap is FAST!

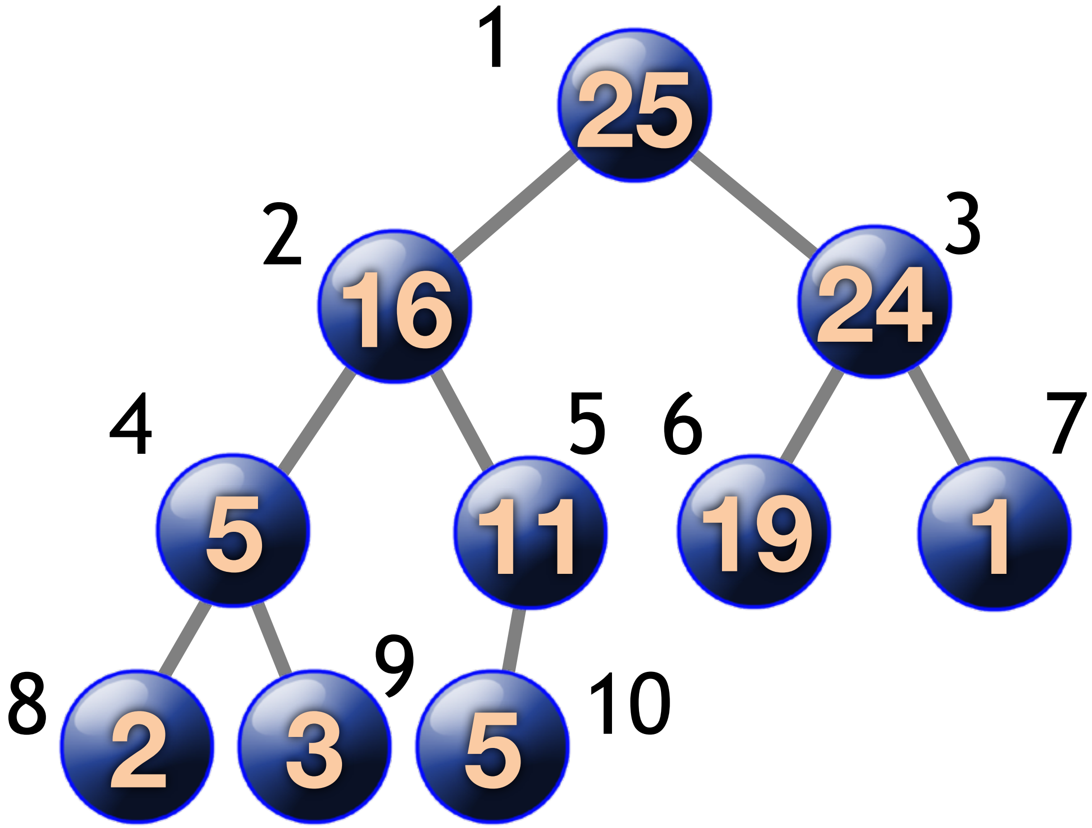
- Insert in O(log n)
- Get Max in O(1)
- Remove Max in O(log n)



Easy to implement using a List



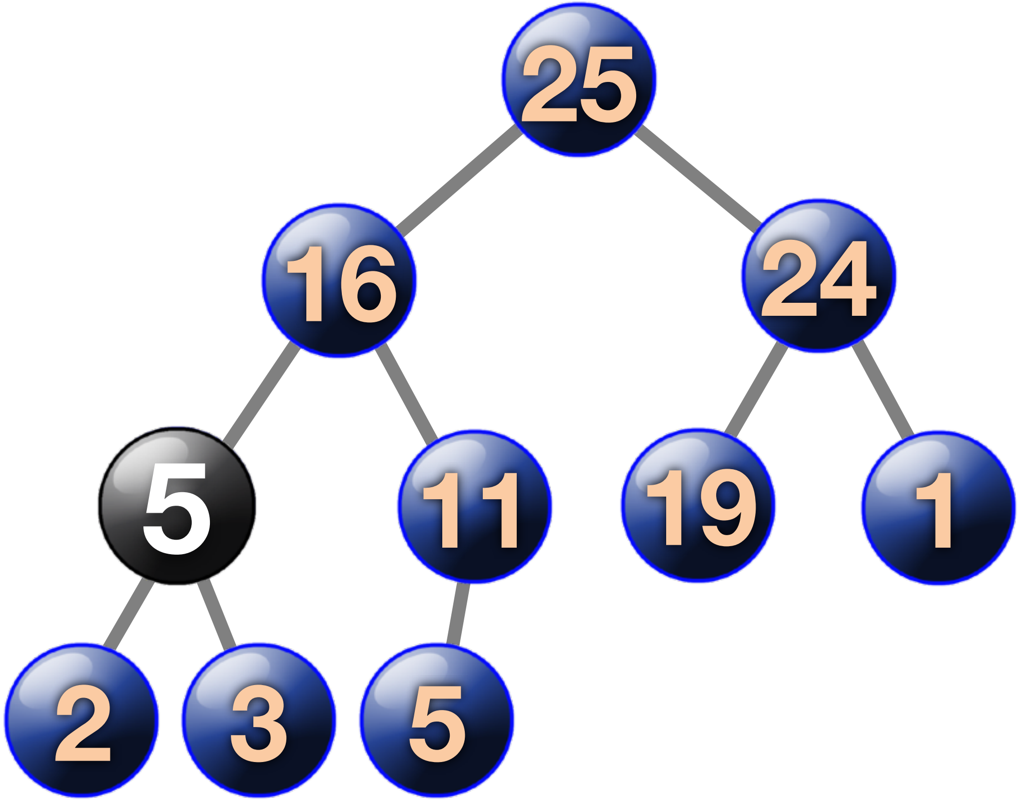
Easy to implement using a List



1	2	3	4	5	6	7	8	9	10
25	16	24	5	11	19	1	2	3	5

Easy to implement using a List

i = 4

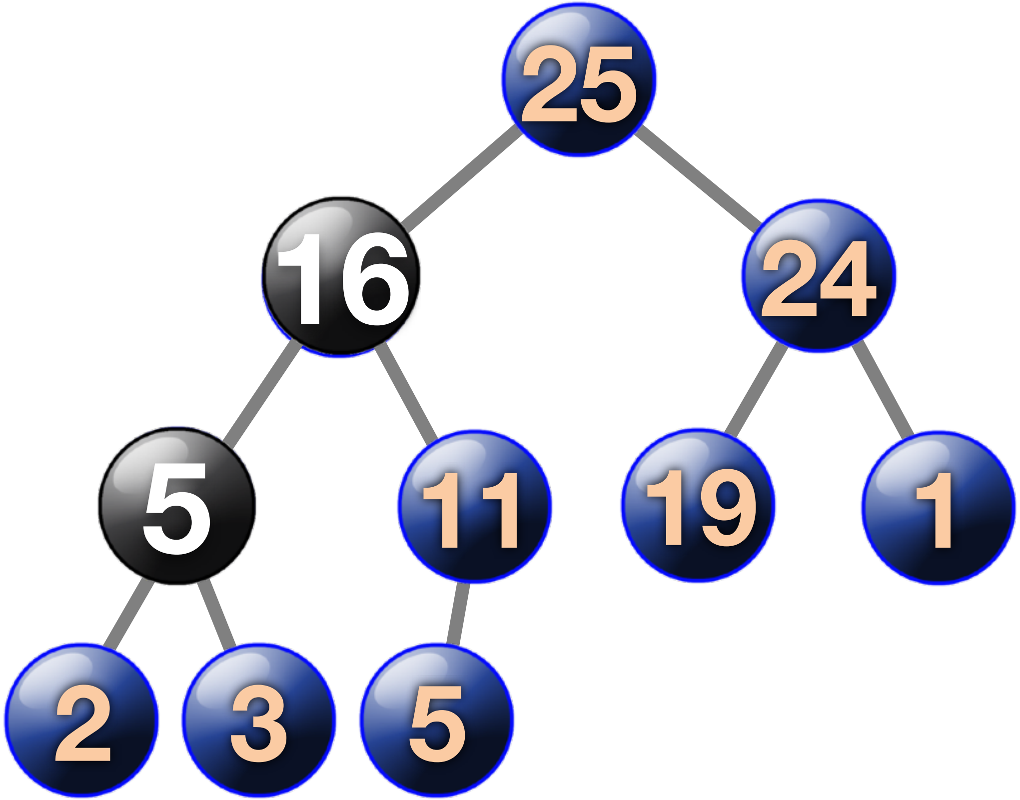


1	2	3	4	5	6	7	8	9	10
25	16	24	5	11	19	1	2	3	5

Easy to implement using a List

i = 4

parent(i) = i/2 = 2



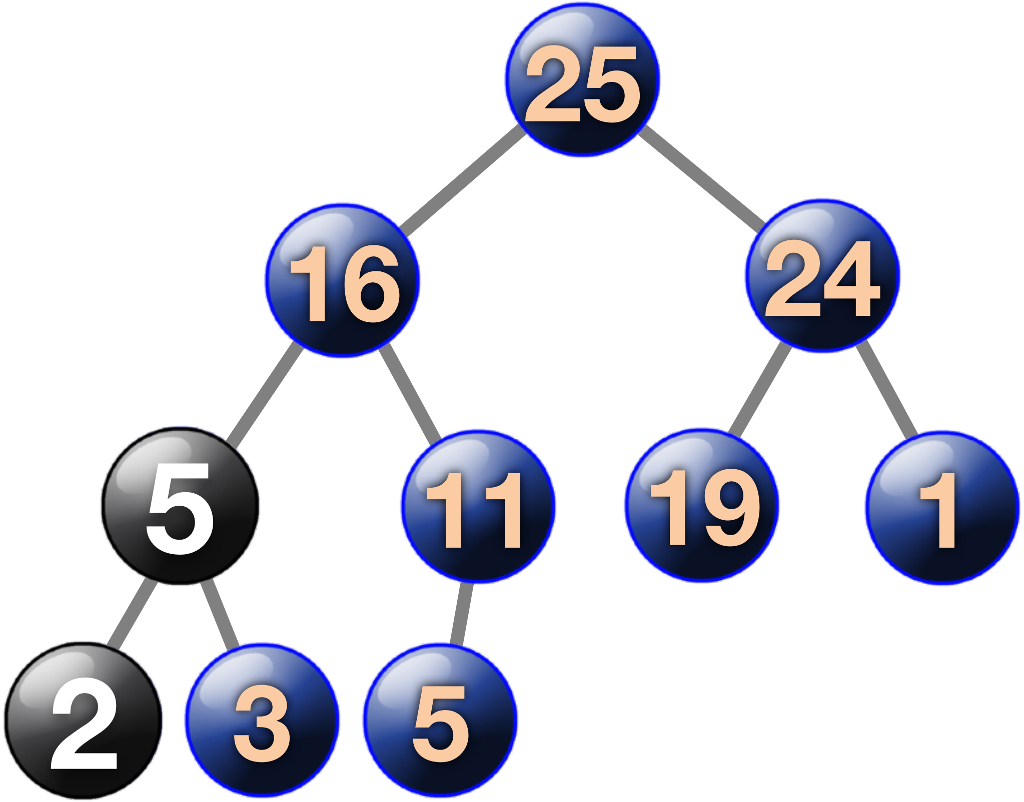
1	2	3	4	5	6	7	8	9	10
25	16	24	5	11	19	1	2	3	5

Easy to implement using a List

i = 4

parent(i) = i/2 = 2

left(i) = i * 2 = 8



1	2	3	4	5	6	7	8	9	10
25	16	24	5	11	19	1	2	3	5

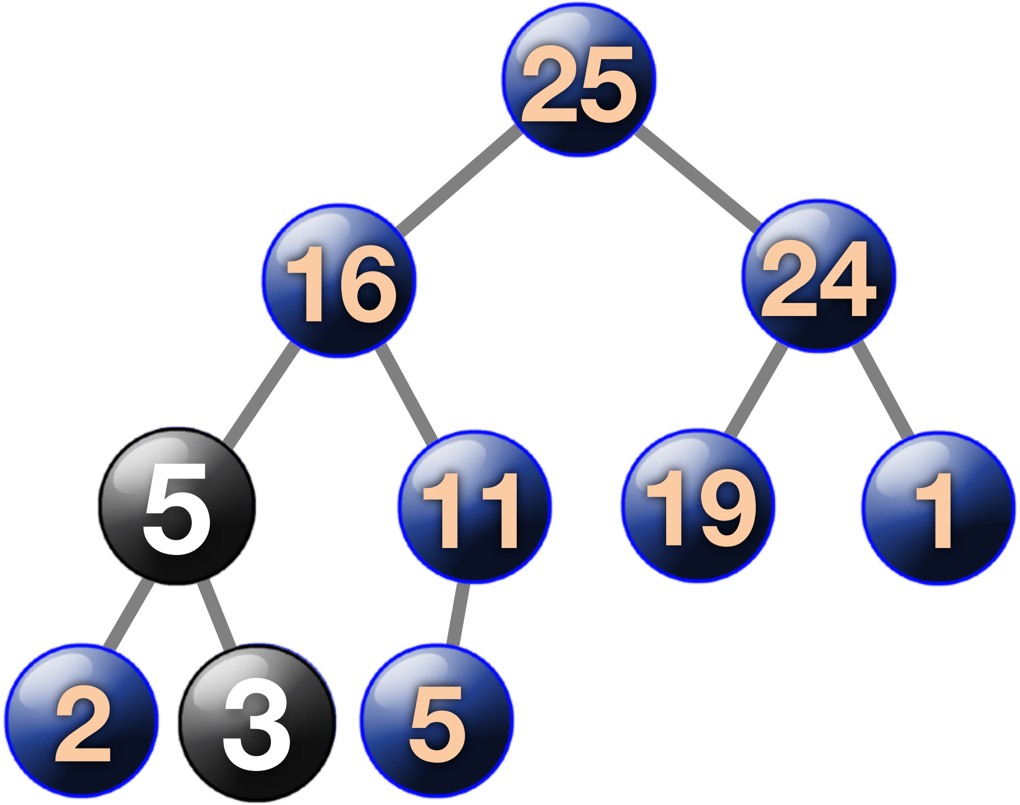
Easy to implement using a List

$i = 4$

$\text{parent}(i) = i/2 = 2$

$\text{left}(i) = i * 2 = 8$

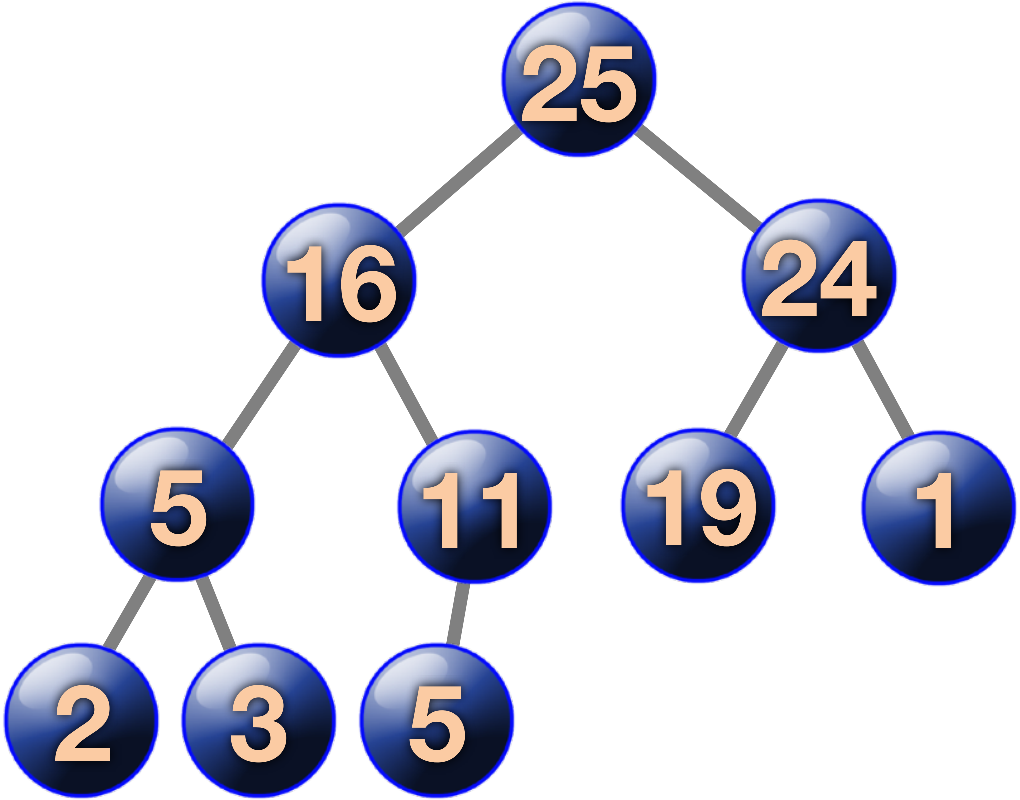
$\text{right}(i) = i * 2 + 1 = 9$



1	2	3	4	5	6	7	8	9	10
25	16	24	5	11	19	1	2	3	5

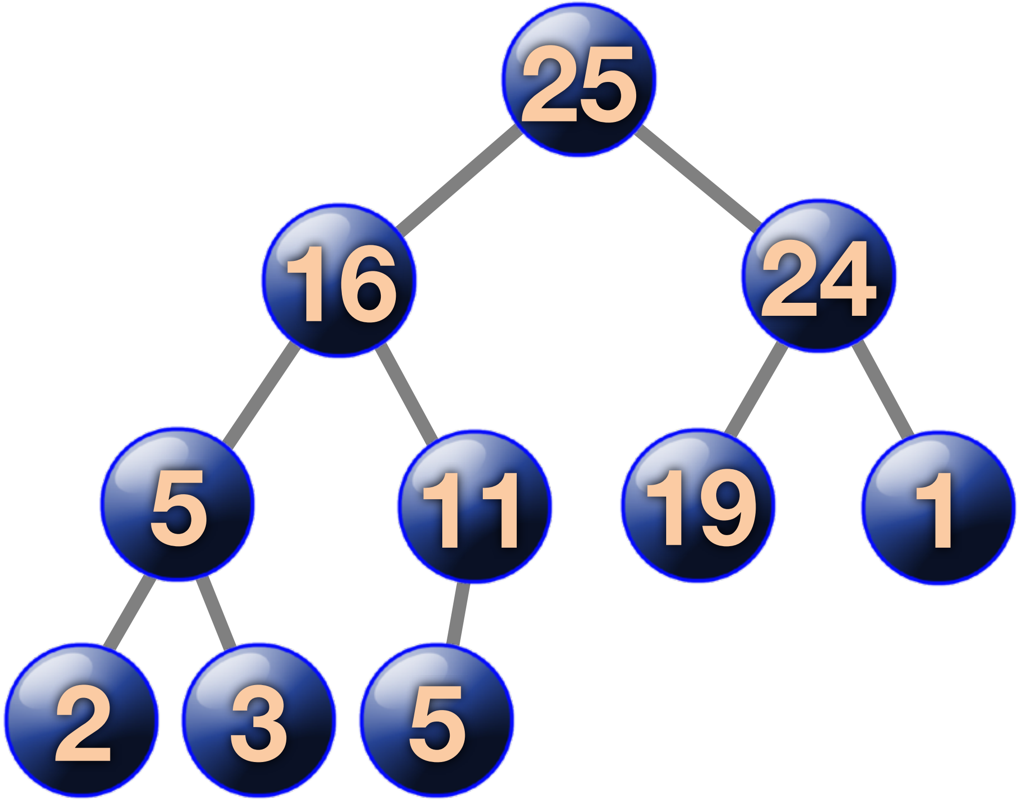
MaxHeap Operations

- Push (insert)
- Peek (get max)
- Pop (remove max)



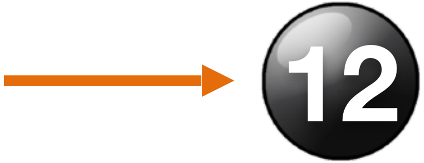
Push

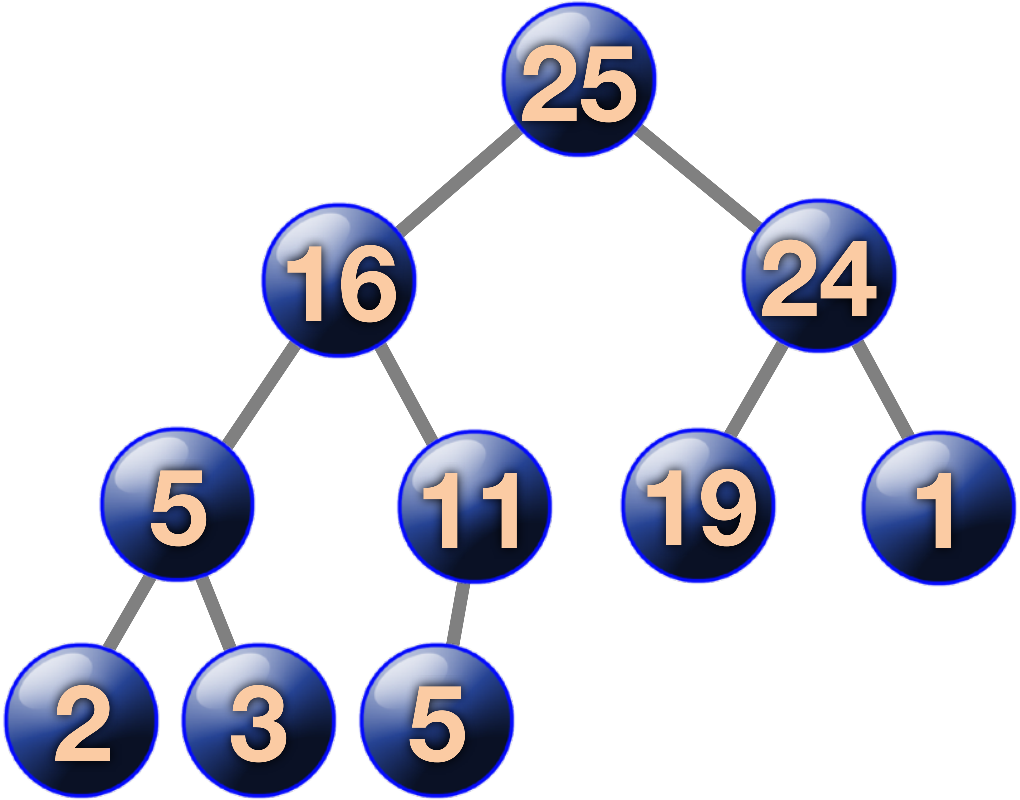
- Add value to end of array
- Float it Up to its proper position



Push

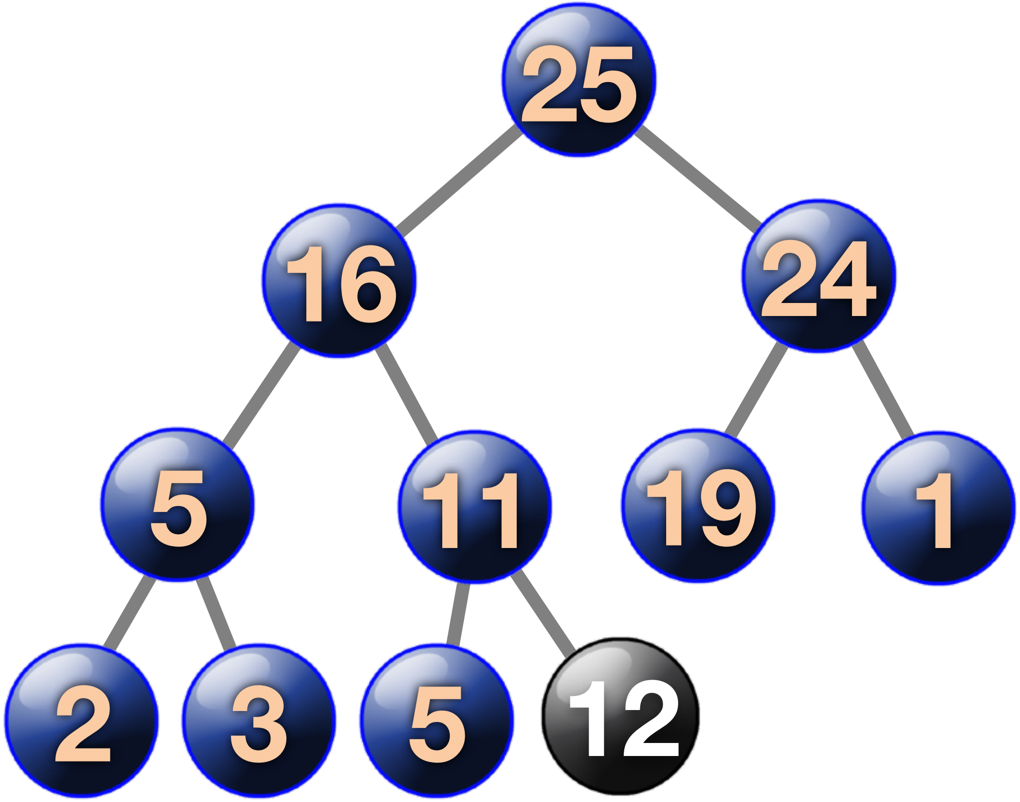
- Add value to end of array
- Float it Up to its proper position





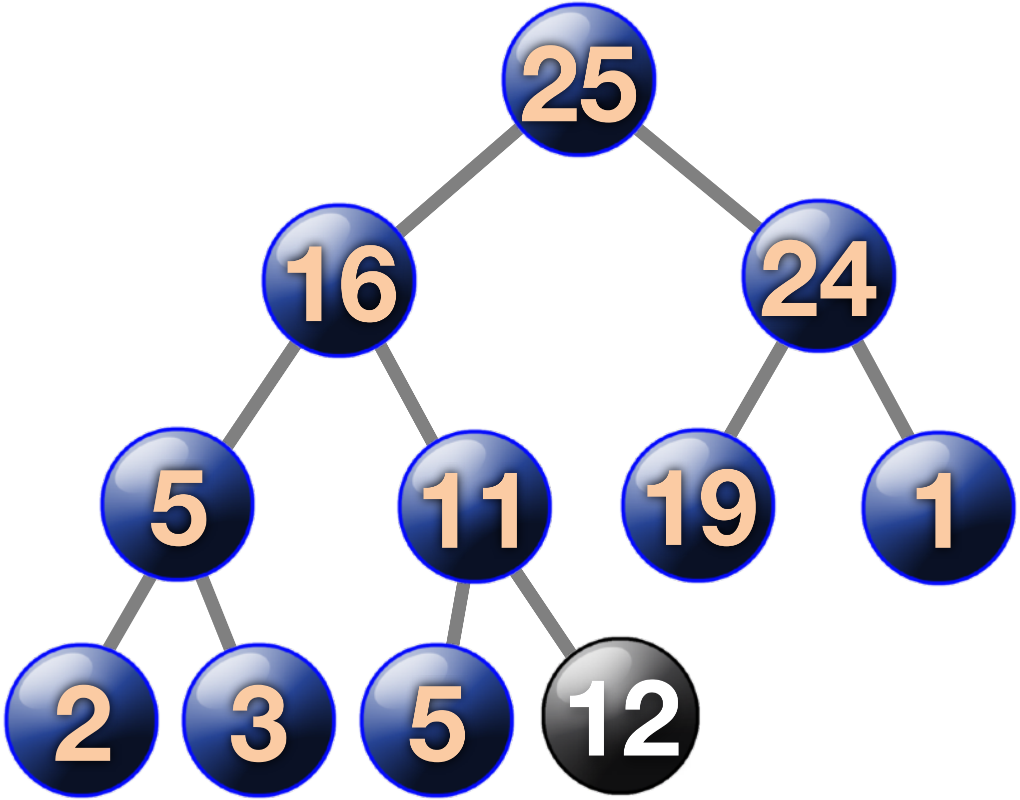
Push

- Add value to end of array
- Float it Up to its proper position



Push

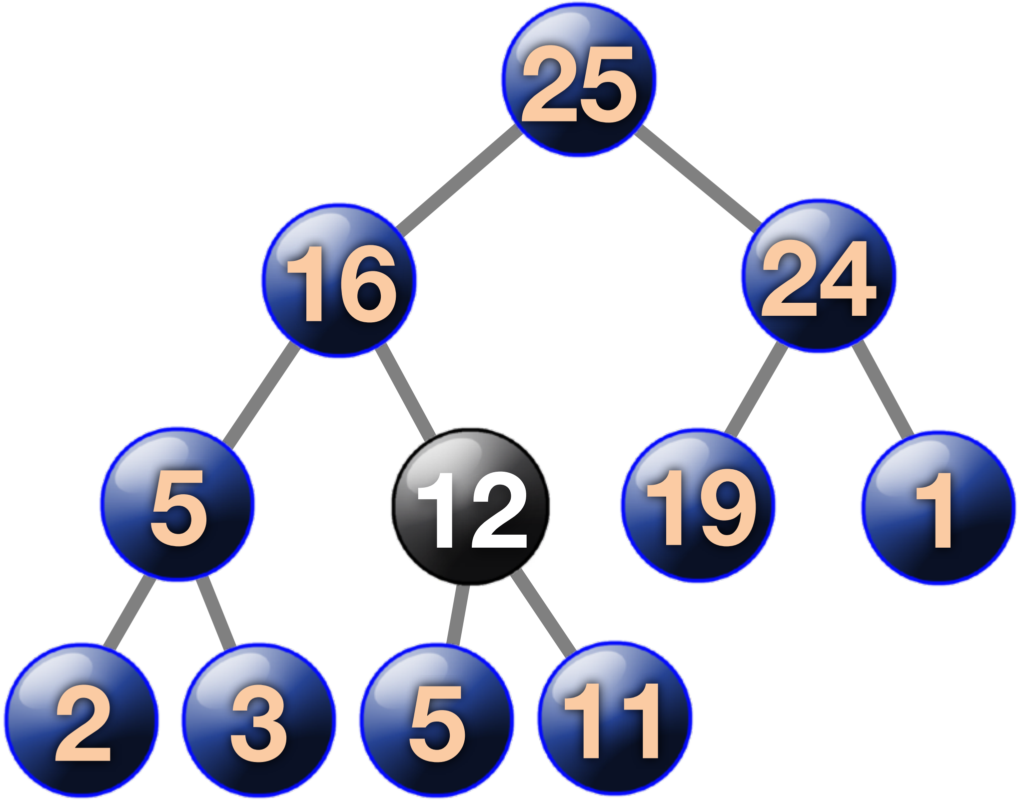
- Add value to end of array
- Float it Up to its proper position



12 > 11 ?

Push

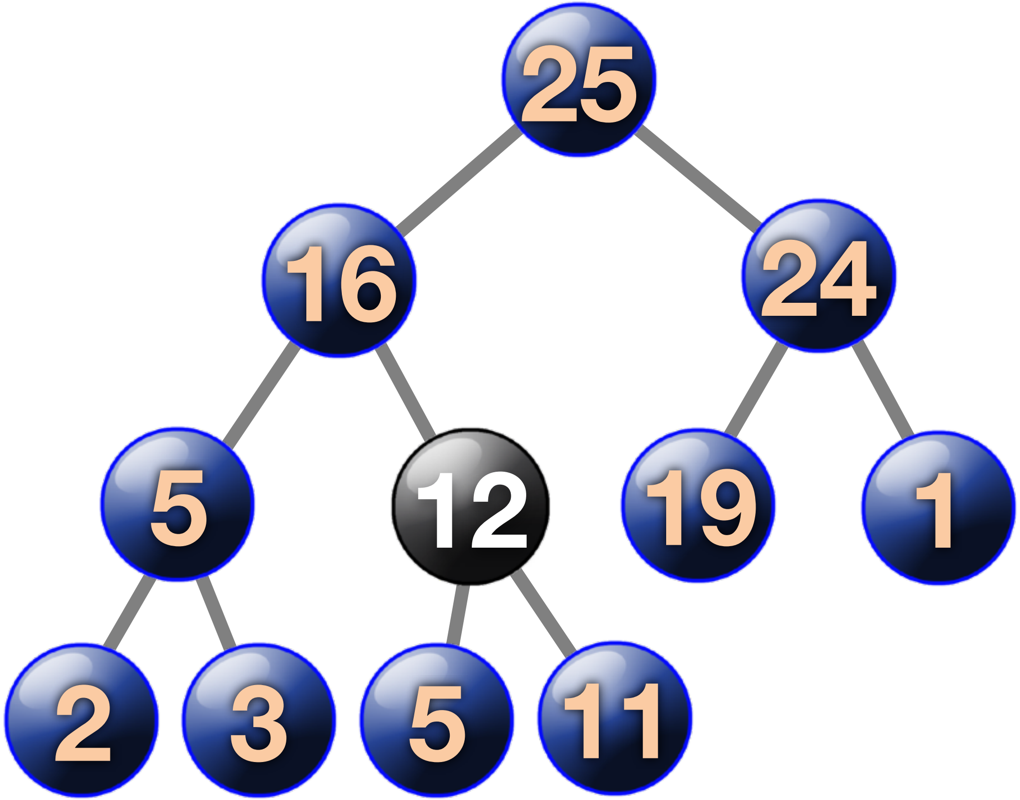
- Add value to end of array
- Float it Up to its proper position



12 > 11 ?

Push

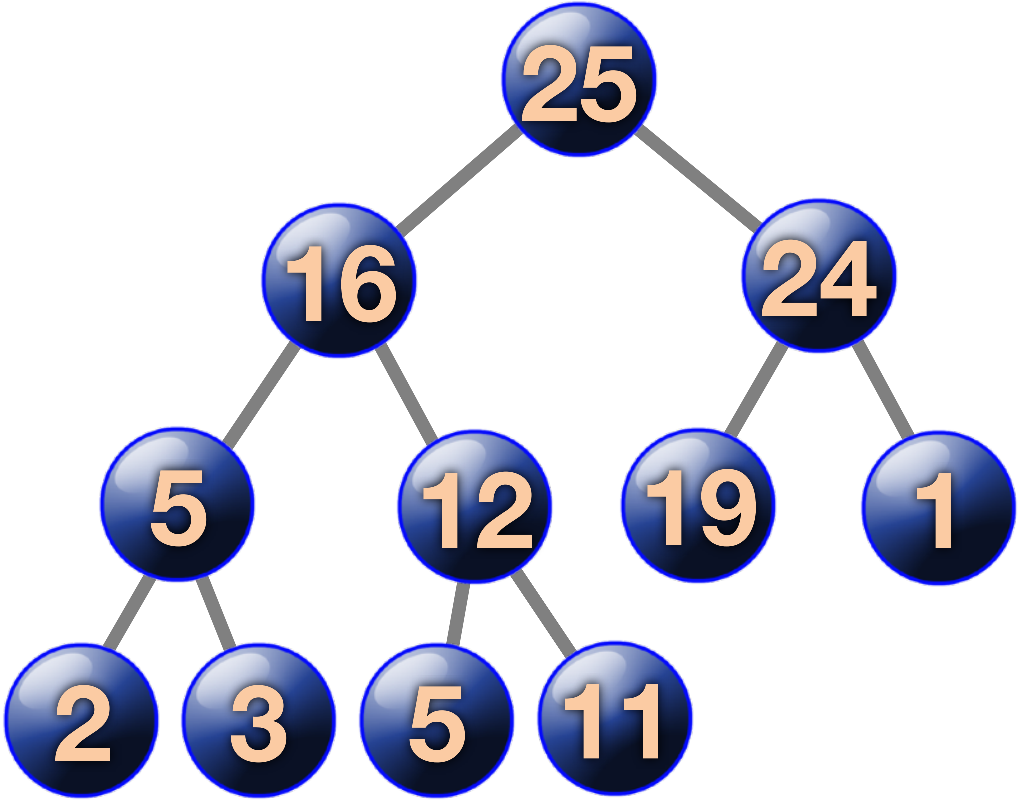
- Add value to end of array
- Float it Up to its proper position



12 > 16 ?

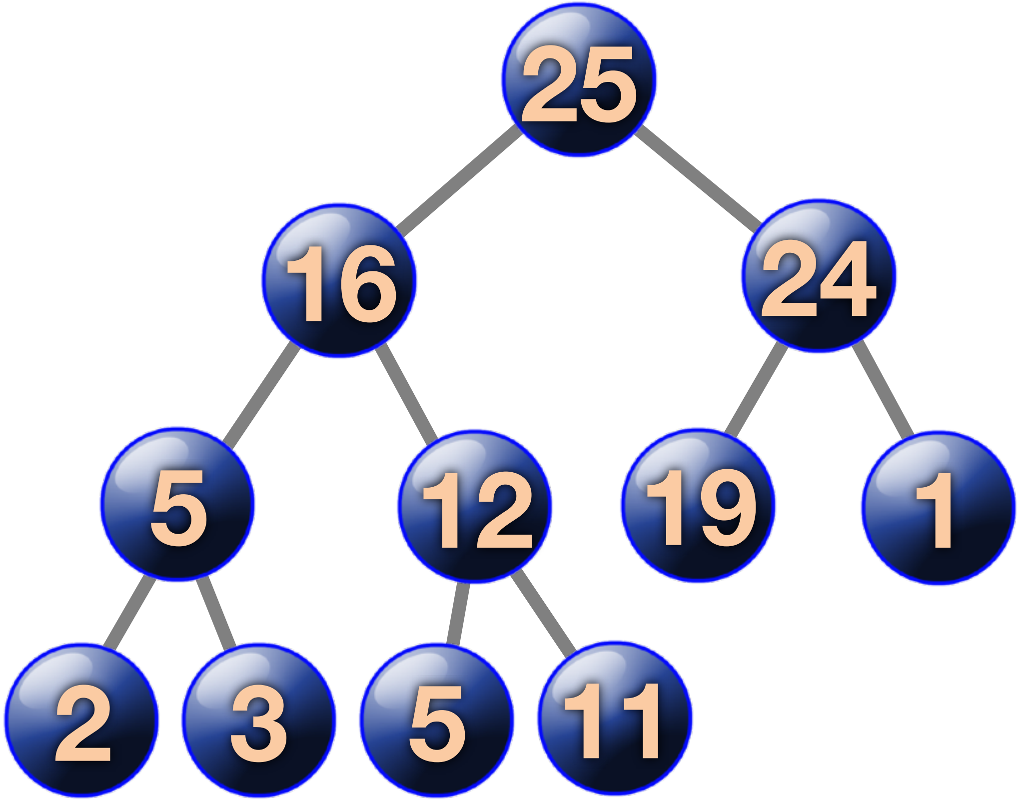
Peek

- Return the value at heap[1]



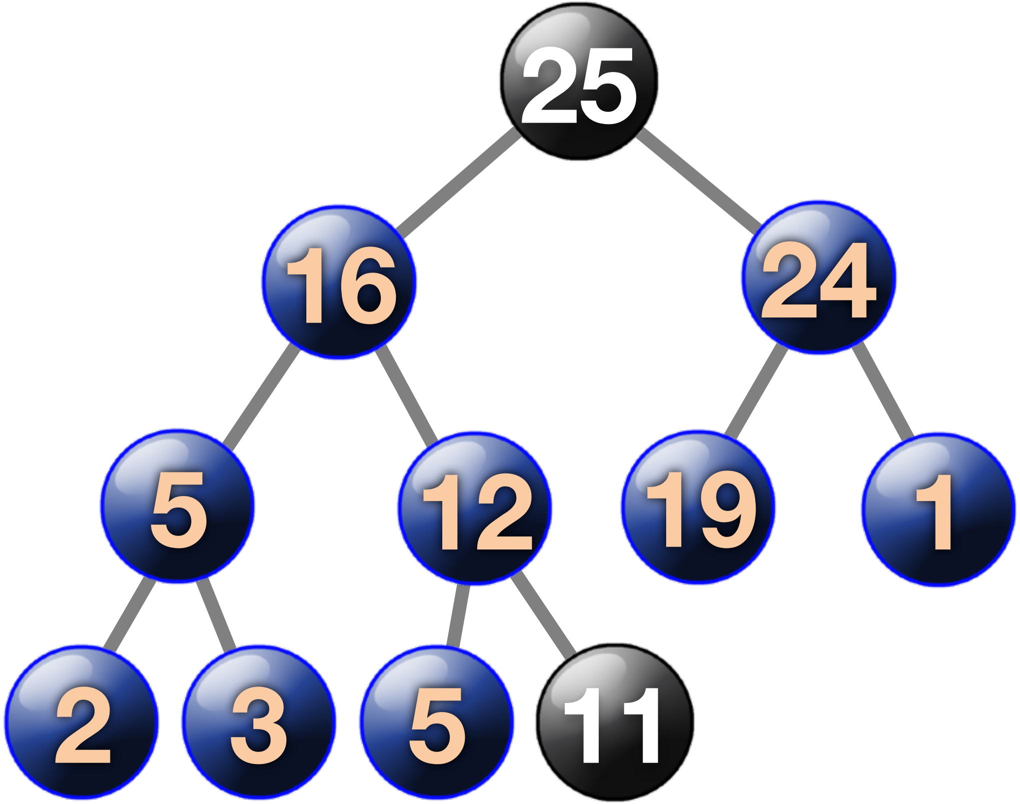
Pop

- Move max to end of array
- Delete it
- Bubble Down the item at index 1 to its proper position
- Return max



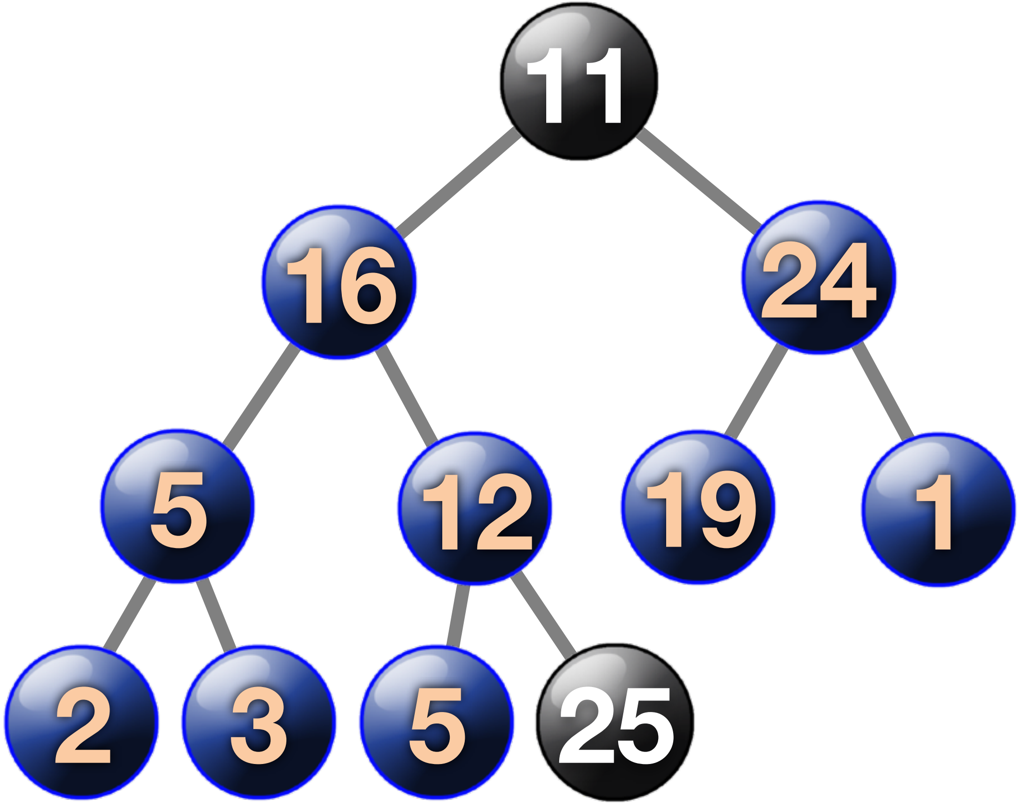
Pop

- Move max to end of array
- Delete it
- Bubble Down the item at index 1 to its proper position
- Return max



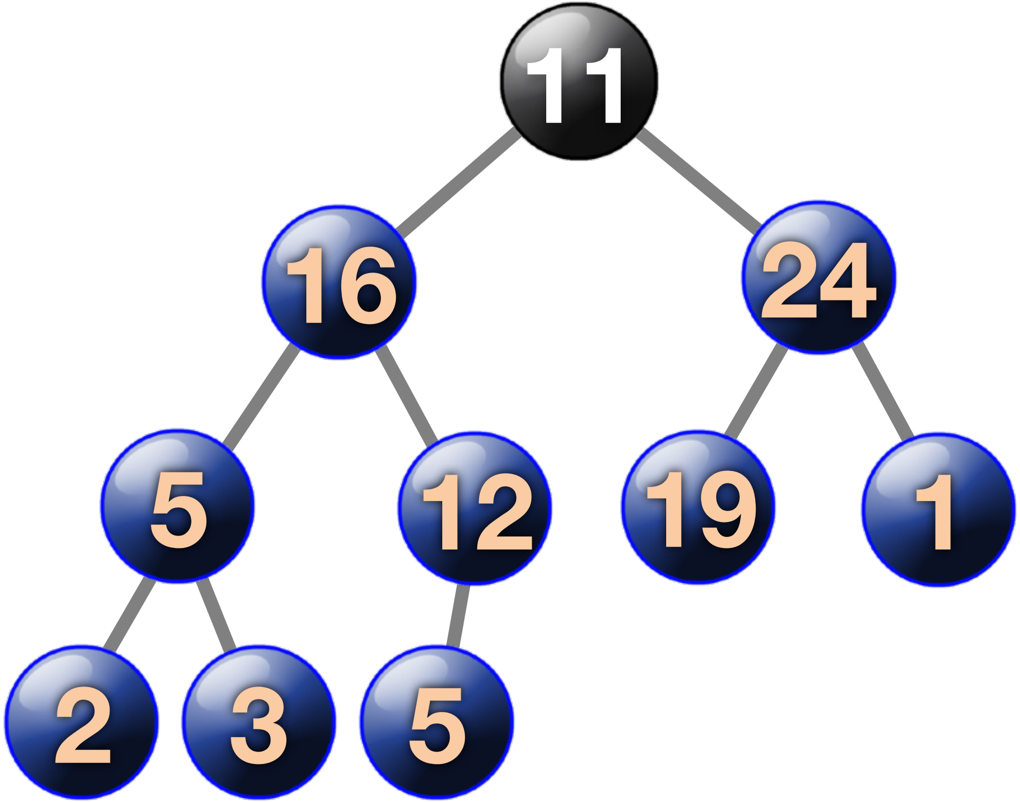
Pop

- Move max to end of array
- Delete it
- Bubble Down the item at index 1 to its proper position
- Return max



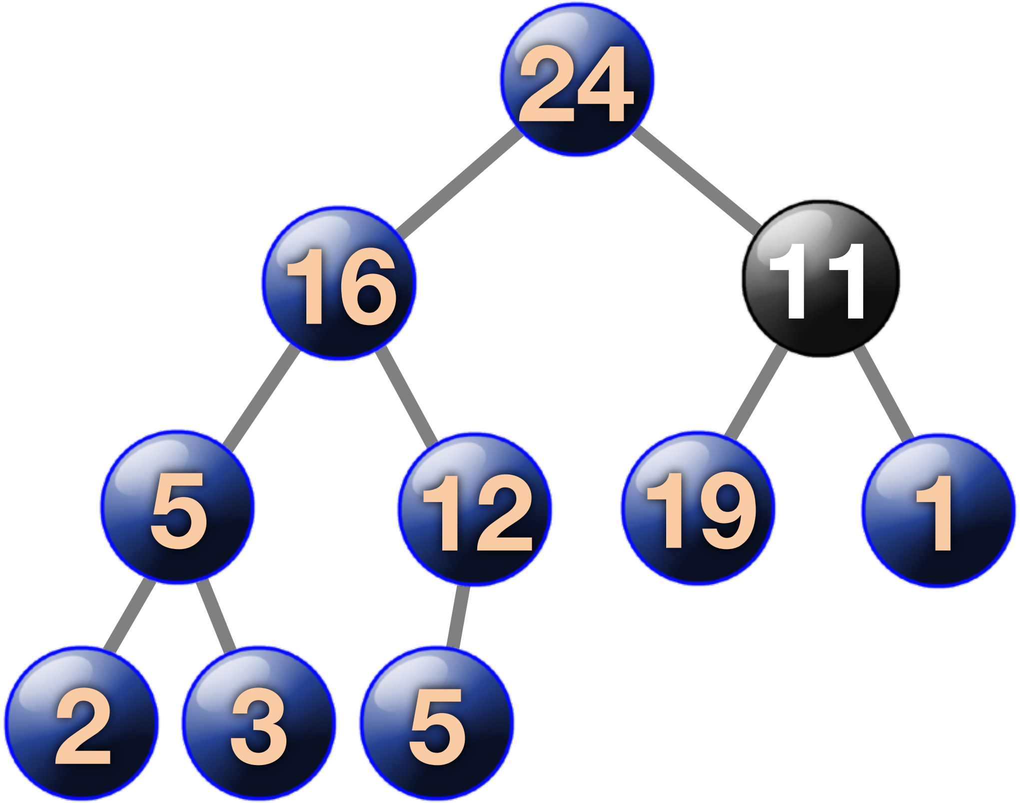
Pop

- Move max to end of array
- Delete it
- Bubble Down the item at index 1 to its proper position
- Return max



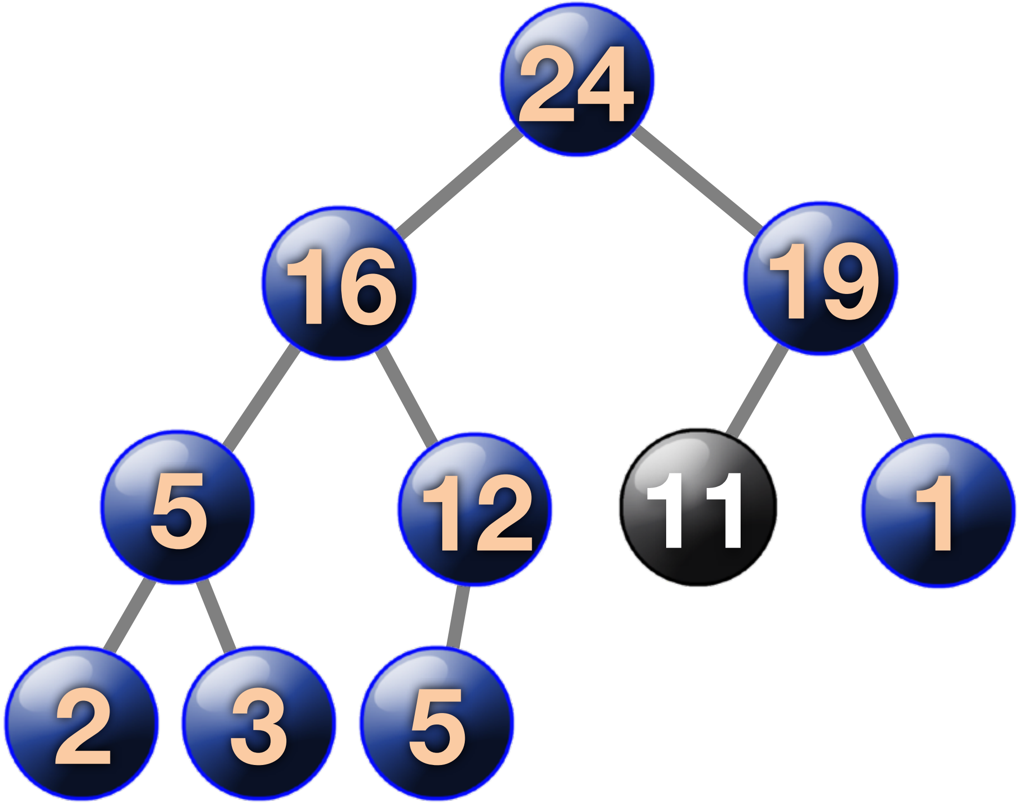
Pop

- Move max to end of array
- Delete it
- Bubble Down the item at index 1 to its proper position
- Return max



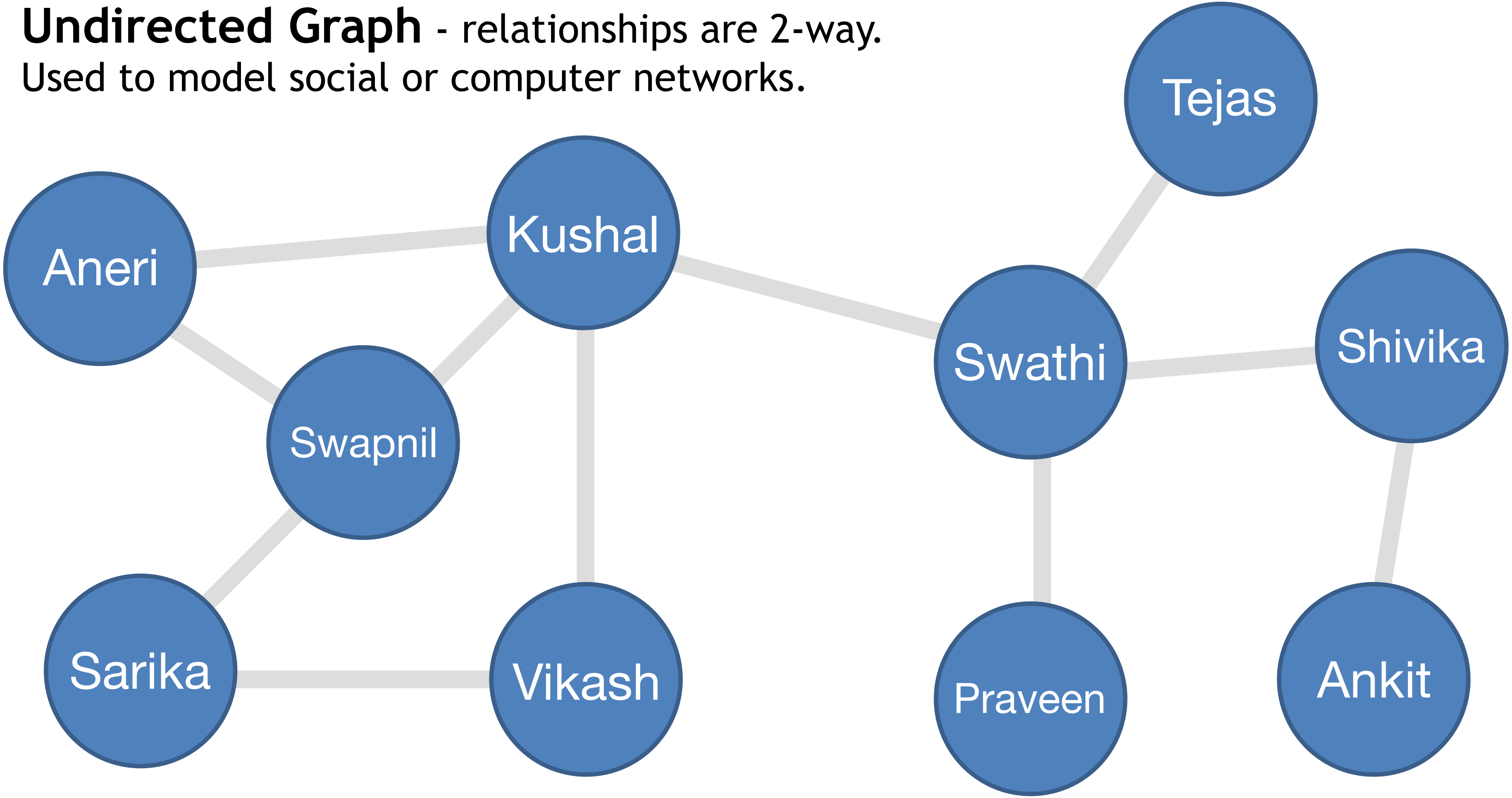
Pop

- Move max to end of array
- Delete it
- Bubble Down the item at index 1 to its proper position
- Return max

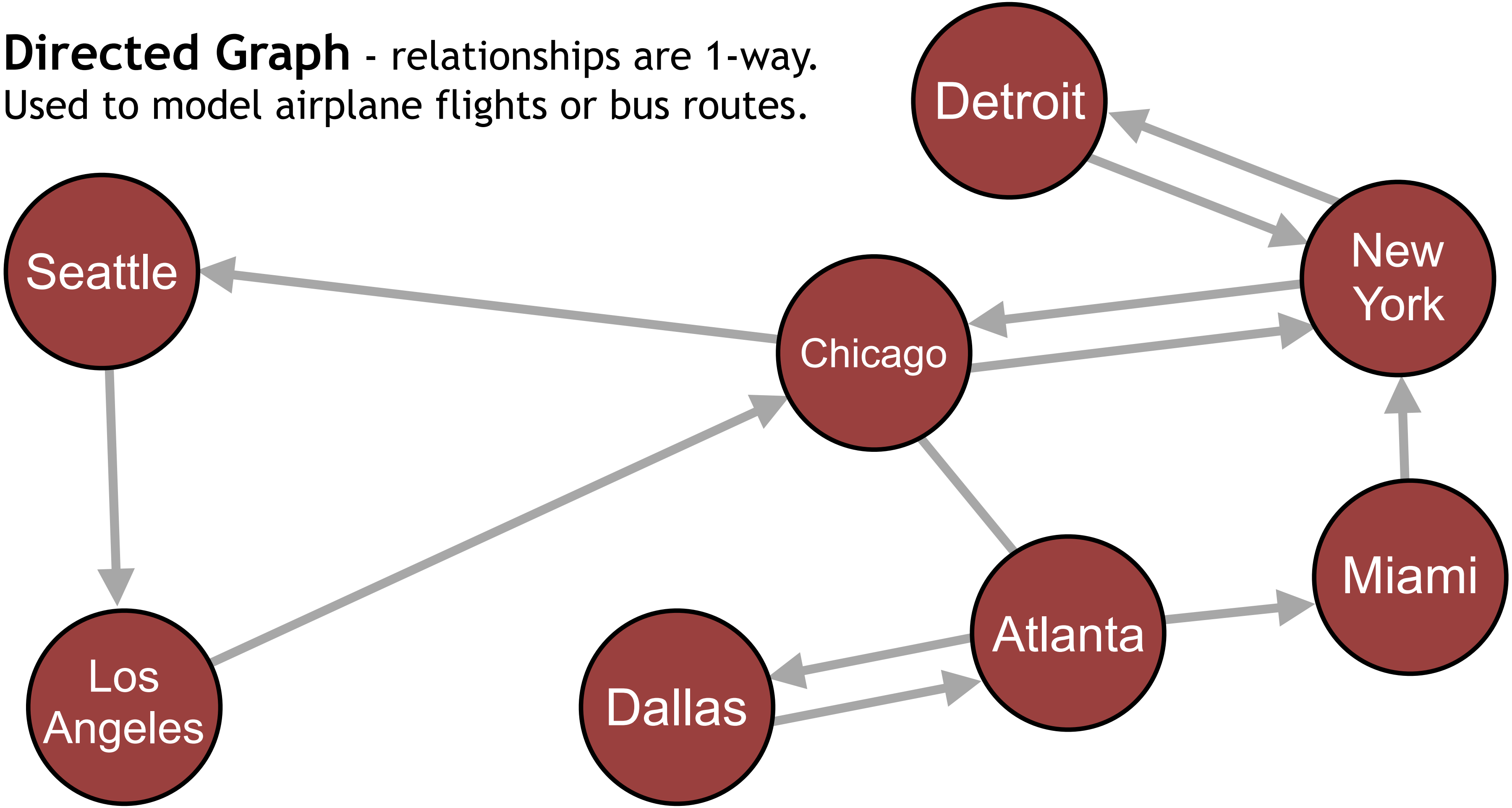


Graphs

Undirected Graph - relationships are 2-way.
Used to model social or computer networks.



Directed Graph - relationships are 1-way.
Used to model airplane flights or bus routes.



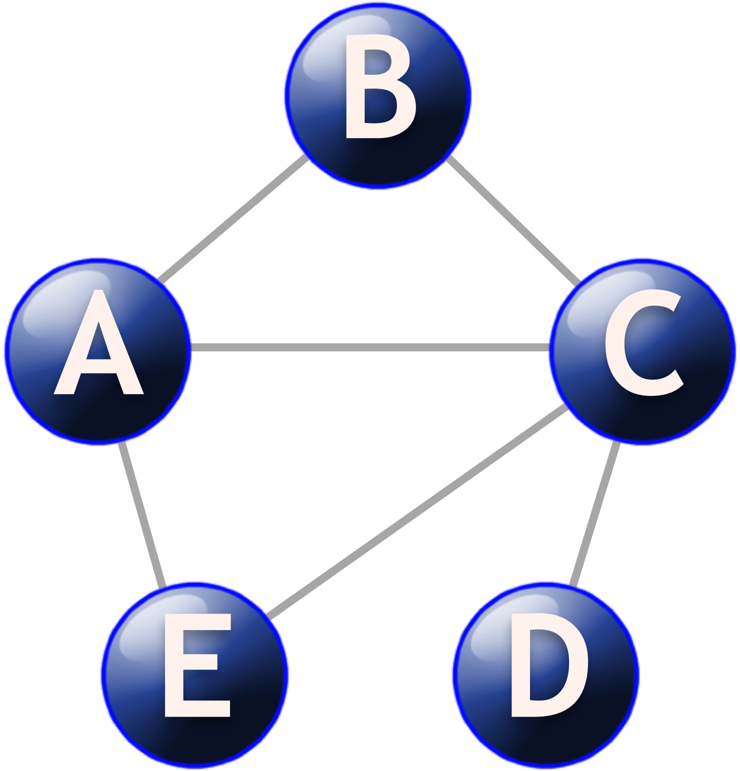
Adjacency Matrix

Matrix of neighbors stored centrally in Graph object

Adjacency List

List of neighbors stored in each vertex

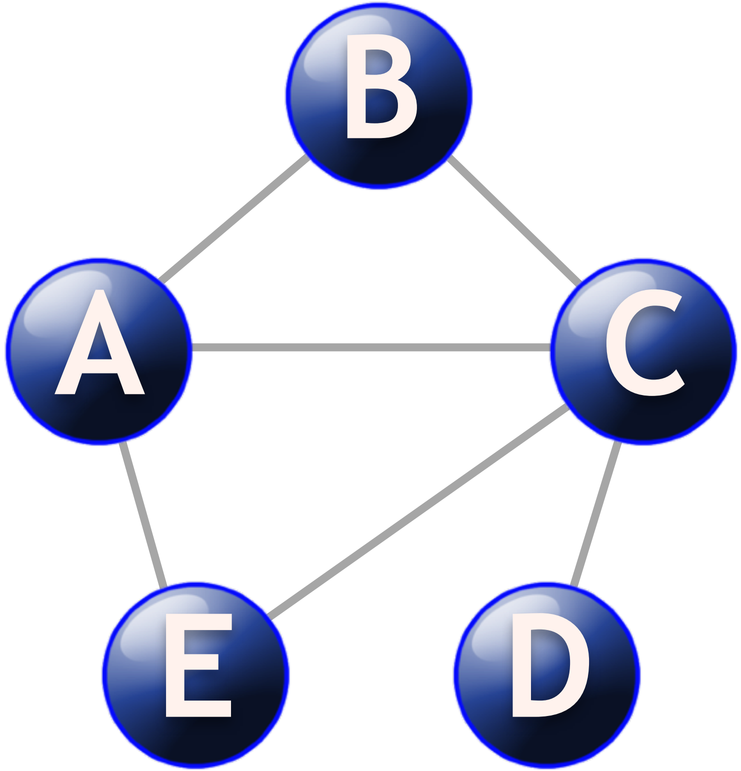
Undirected Graph



Adjacency List

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Undirected Graph



Adjacency List

A: B, C, E   Stored in Node A

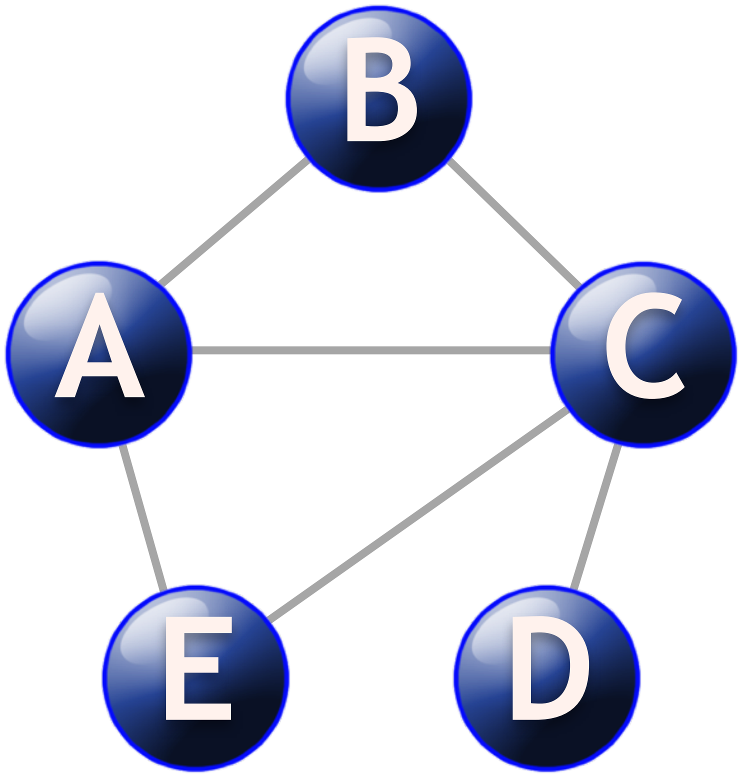
B: A, C

C: A, B, D, E

D: C

E: A, C

Undirected Graph

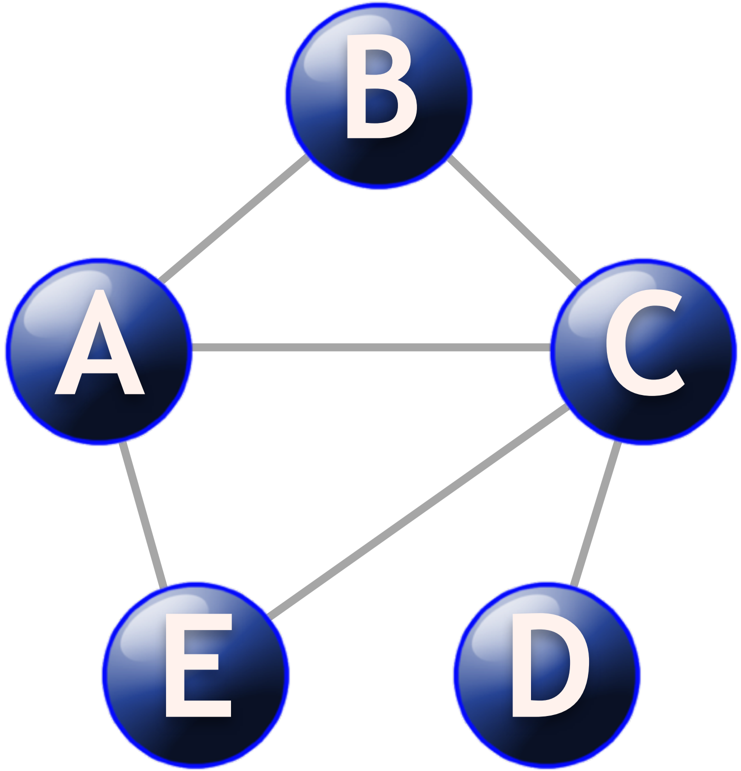


Adjacency List

A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C

Stored in Node B

Undirected Graph



Adjacency Matrix

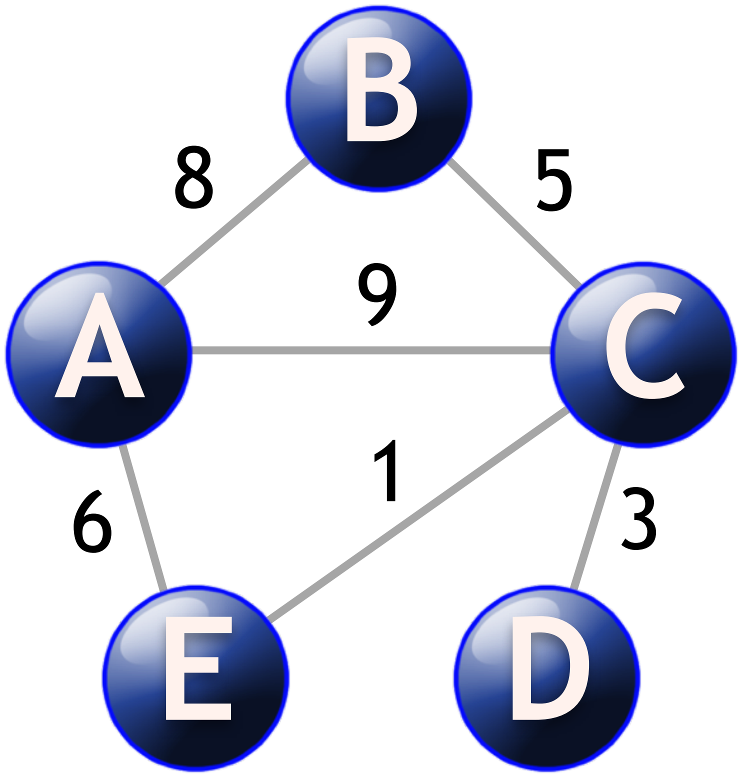
to

from

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	0
E	1	0	1	0	0

Stored in Graph

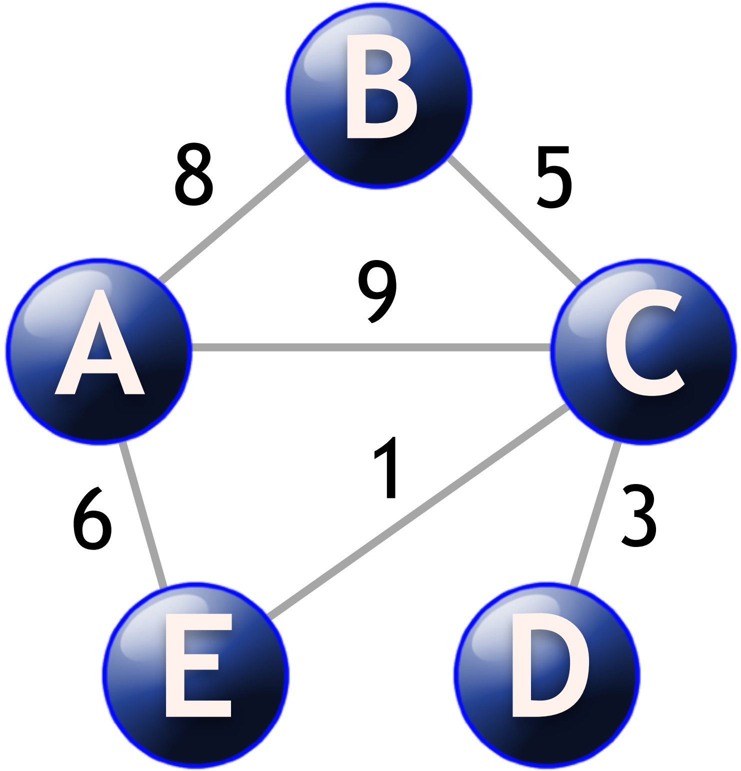
Undirected Graph



Weighted Edges?

Much easier to implement
with
Adjacency Matrix

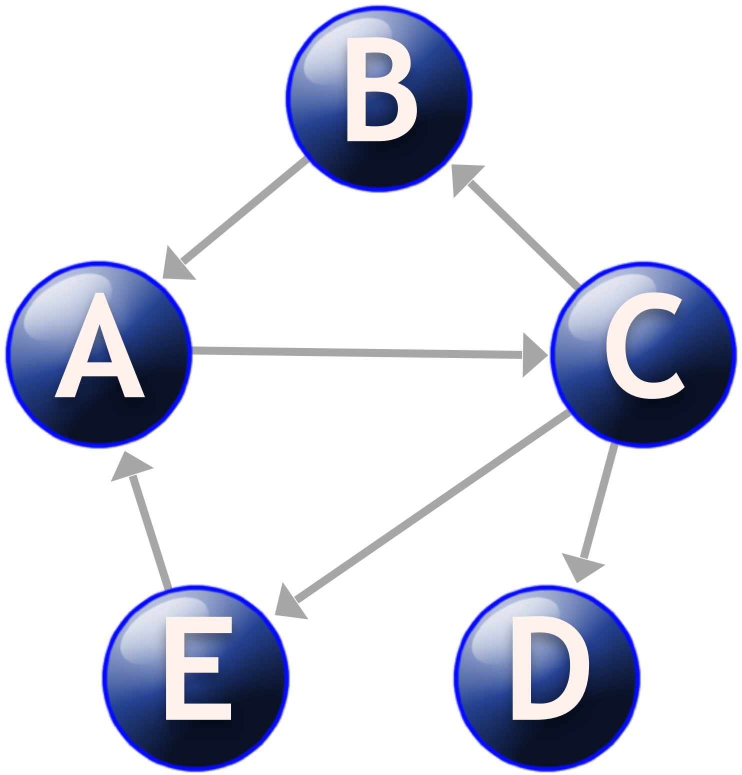
Undirected Graph



Adjacency Matrix

	A	B	C	D	E
A	0	8	9	0	6
B	8	0	5	0	0
C	9	5	0	3	1
D	0	0	3	0	0
E	6	0	1	0	0

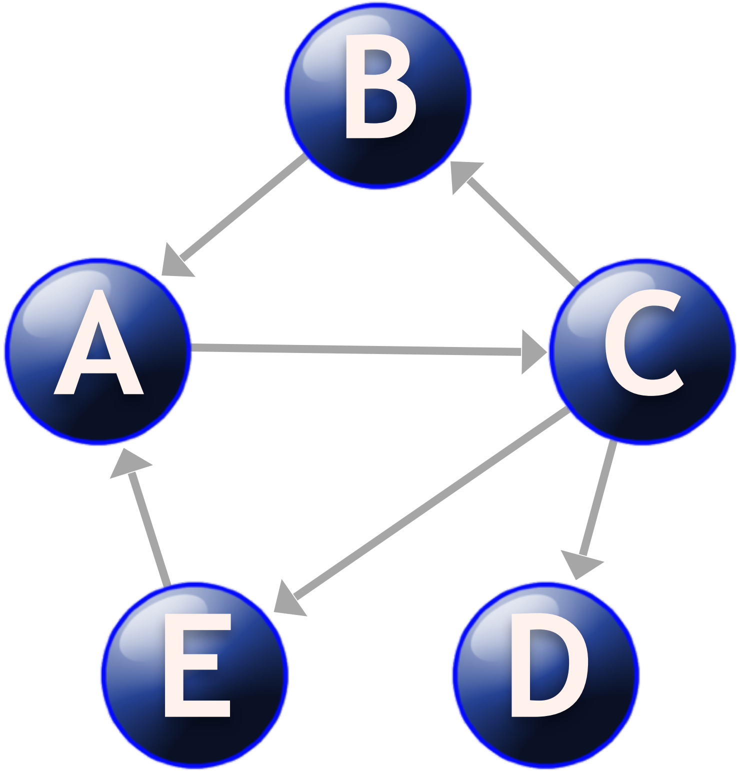
Directed Graph



Adjacency List

A: C
B: A
C: B, D, E
D:
E: A

Directed Graph



Adjacency Matrix

to

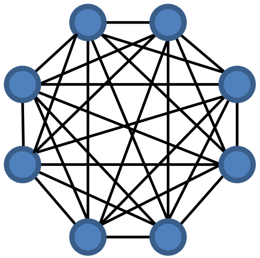
from

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	0
C	0	1	0	1	1
D	0	0	0	0	0
E	1	0	0	0	0

Which is Better?

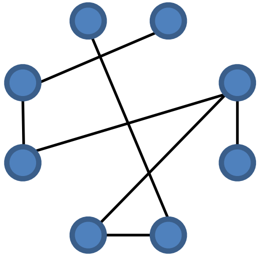
Dense Graph –

graph where $|E| = |V|^2$



Sparse Graph –

graph where $|E| = |V|$



Which is Better?

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	0
C	0	1	0	1	1
D	0	0	0	0	0
E	1	0	0	0	0

Adjacency Matrix takes up $|V|^2$ space, regardless how dense the graph

Matrix for a graph with 10,000 vertices will take up at least 100,000,000 Bytes

Which is Better?

Adjacency List

- Pro: Faster and uses less space for Sparse graphs
- Con: Slower for Dense graphs

Adjacency Matrix

- Pro: Faster for Dense graphs
- Pro: Simpler for Weighted edges
- Con: uses more space