Algorithms and Data Structures (II)

Gabriel Istrate

March 18, 2020

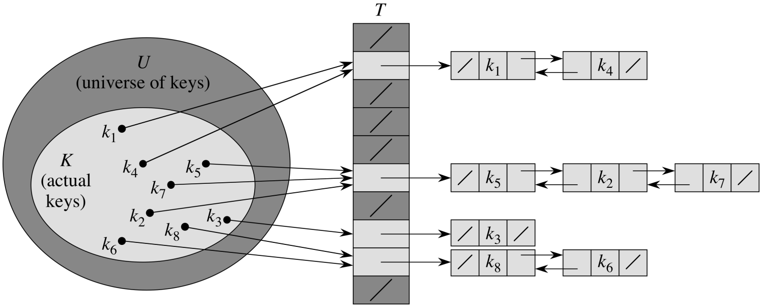
- Wrap up hash tables.
- Skip lists.
- Binary search trees
- Randomized binary search trees

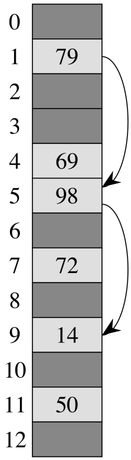
Where are we ?

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*
- *Interface* (generic interface)
  - **INSERT**(D, k) adds a key k to the dictionary D
  - **DELETE**(D, k) removes key k from D
  - **SEARCH**(D, k) tells whether D contains a key k

- A *dictionary* is an abstract data structure that represents a set of elements (or keys)
  - a *dynamic set*
- *Interface* (generic interface)
  - **INSERT**(D, k) adds a key k to the dictionary D
  - **DELETE**(D, k) removes key k from D
  - **SEARCH**(D, k) tells whether D contains a key k
- *Implementation (so far)*
  - direct access tables. Linked lists. Hash tables.


T
U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_5
k_7
k_2
k_3
k_8
k_6
k_1
k_4
k_5
k_2
k_7
k_3
k_8
k_6



Three versions: linear/quadratic/double probing.

Algorithm	***Average** Complexity (Search successful/not)*	
INSERT/SEARCH/DELETE, CHAINING:	$O(1+\alpha)$	✓
SEARCH, LINEAR PROBING:	$\frac{1}{2}(1+\frac{1}{1-\alpha}), \frac{1}{2}(1+\frac{1}{1-\alpha^2})$	✓
SEARCH, QUADRATIC PROBING:	$1 - ln(1-\alpha) - \frac{\alpha}{2}, \frac{1}{1-\alpha} - \alpha - ln(1-\alpha)$	✓
SEARCH, DOUBLE HASHING:	$\frac{1}{\alpha} ln(1-\alpha), \frac{1}{1-\alpha}$	✓

Reference, probing complexities: Drozdek/Knuth.

- practical ! ✓
- Hard to analyze mathematically: those results under uniform hashing (not at all clear) $\times$
- Somewhat hard to engineer. $\times$.

Perhaps $O(1(+\alpha))$ too ambitious ? Something, say $O(logn)$?

Perhaps $O(1(+\alpha))$ too ambitious ? Something, say $O(logn)$?

In practice $log(n)$ is a small number !

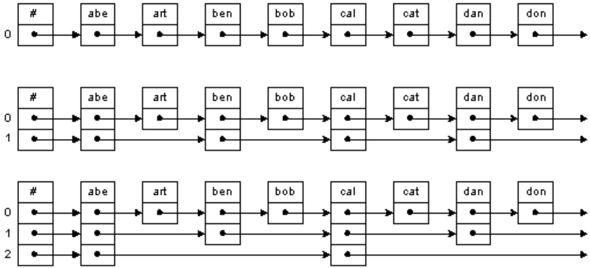
Caution

Topic **not** in Cormen. See Drozdek for details/C++ implementation.

- Problem with linked list: *search is slow !*... even when elements sorted.
- Solution: lists of ordered elements that allow skipping some elements to speed up search.
- Skip lists: variant of ordered linked lists that makes such search possible.

More advanced data structure (W. Pugh "Skip lists: a Probabilistic Alternative to Balanced Trees", Communication of the ACM 33(1990), pp. 668-676.) *If anyone curious/interested in data structures/algorithms, can give paper to read; taste how a research article looks like.*

Skip lists



Where does this ever get applied ? ...

Skip lists in real life

According to Wikipedia:

- MemSQL - skip lists as prime indexing structure for its database technology.
- Cyrus IMAP server - "skiplist" backend DB implementation
- Lucene uses skip lists to search delta-encoded posting lists in logarithmic time.
- QMap (up to Qt 4) template class of Qt that provides a dictionary.
- Redis, ANSI-C open-source persistent key/value store for Posix systems, skip lists in implementation of ordered sets.
- nessDB, a very fast key-value embedded Database Storage Engine.
- skipdb: open-source DB format using ordered key/value pairs.
- ConcurrentSkipListSet and ConcurrentSkipListMap in the Java 1.6 API.

According to Wikipedia:

- Speed Tables: fast key-value datastore for Tcl that use skiplists for indexes and lockless shared memory.
- leveldb, a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values
- MuQSS Scheduler for the Linux kernel uses skip lists
- SkipMap uses skip lists as base data structure to build a more complex 3D Sparse Grid for Robot Mapping systems.

What we want

$k = 1, \ldots, \lfloor \log_2(n) \rfloor$, $1 \leq i \leq \lfloor n/2^{k-1} \rfloor - 1$.

- Item $2^{k-1} \cdot i$ points to item $2^{k-1} \cdot (i+1)$.
- every second node points to positions two node ahead,
- every fourth node points to positions four nodes ahead,
- every eigth node points to positions eigth nodes ahead,
- $\ldots \ldots$, and so on.

- **Different number of pointers in different nodes in the list !**
- half the nodes only one pointer.
- a quarter of the nodes two pointers,
- an eigth of the nodes four pointers,
- $\ldots \ldots$, and so on.
- $n \log_2(n)/2$ pointers.

1. First follow pointers on the highest level until a larger element is found or the list is exhausted.
2. If a larger element is found, restart search from its predecessor, this time on a lower level.
3. Continue doing this until element found, or you reach the first level and a larger element or the end of the list.

Major problem

- When inserting/deleting a node, pointers of prev/next nodes have to be restructured.
- Solution: rather than equal spacing, random spacing on a level.
- Invariant: Number of nodes on each level: equal, in expectation to what it would be under equal spacing

Principle

If you're traveling 10 meters in 10 steps, a step is on average one meter.

Inserting and deleting nodes (II)

- Level numbering: start with zero.
- New node inserted: probability 1/2 on first level, 1/4 second level, 1/8 third level, . . ., etc.
- Function *chooseLevel*: chooses randomly the level of the new node.
- Generate random number. If in [0,1/2] level 1, [1/2,3/4] level 2, etc.
- To delete node: have to update all links.

Computing the i'th element faster than in $O(i)$

- If we record "step sizes" in our lists we can even mimic indexing !
- Start on highest level.
- If step too big, restart search from predecessor, this time on a lower level.
- Continue doing this until element found.

Update "step sizes" by insertion/deletion

Easy if you have doubly linked lists.

- On deletion: $pred[i].size+ = deleted.size$ on all levels i.
- On insertion: Simply keep track of predecessors and index of the inserteed sequence.

Method	***Average***	***Worst-Case***
SPACE: ✓	$O(n)$	$O(nlog(n))$
SEARCH: ✓	$O(\log(n))$	$O(n)$
INSERT: ✓	$O(\log(n))$	$O(n)$
DELETE: ✓	$O(\log(n))$	$O(n)$

- quite practical ! ✓
- Probabilistic, worst-case still bad. ×
- Not completely easy to implement. ×.

Compared to what ?

Binary search trees. Will learn about them next.

- A ***binary search tree*** implements of a *dynamic set*
  - over a ***totally ordered domain***

- A ***binary search tree*** implements of a *dynamic set*
  - over a ***totally ordered domain***

- *Interface*
  - **TREE-INSERT**(T, k) adds a key k to the dictionary D
  - **TREE-DELETE**(T, k) removes key k from D
  - **TREE-SEARCH**(T, x) tells whether D contains a key k

- A ***binary search tree*** implements of a *dynamic set*
  - over a ***totally ordered domain***

- *Interface*
  - **TREE-INSERT**(T, k) adds a key k to the dictionary D
  - **TREE-DELETE**(T, k) removes key k from D
  - **TREE-SEARCH**(T, x) tells whether D contains a key k
  - *tree-walk:* **INORDER-TREE-WALK**(T), etc.

- A ***binary search tree*** implements of a *dynamic set*
  - over a ***totally ordered domain***

- *Interface*
  - **TREE-INSERT**(T, k) adds a key k to the dictionary D
  - **TREE-DELETE**(T, k) removes key k from D
  - **TREE-SEARCH**(T, x) tells whether D contains a key k
  - *tree-walk:* **INORDER-TREE-WALK**(T), etc.
  - **TREE-MINIMUM**(T) finds the smallest element in the tree
  - **TREE-MAXIMUM**(T) finds the largest element in the tree

- A ***binary search tree*** implements of a *dynamic set*
  - over a ***totally ordered domain***

- *Interface*
  - **TREE-INSERT**(T, k) adds a key k to the dictionary D
  - **TREE-DELETE**(T, k) removes key k from D
  - **TREE-SEARCH**(T, x) tells whether D contains a key k
  - *tree-walk:* **INORDER-TREE-WALK**(T), etc.
  - **TREE-MINIMUM**(T) finds the smallest element in the tree
  - **TREE-MAXIMUM**(T) finds the largest element in the tree
  - *iteration:* **TREE-SUCCESSOR**(x) and **TREE-PREDECESSOR**(x) find the successor and predecessor, respectively, of an element x

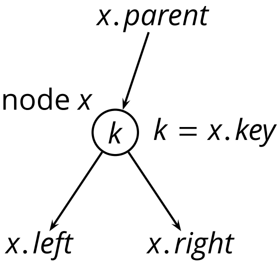
- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***

- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***
  - $T.root$ is the root node of tree T

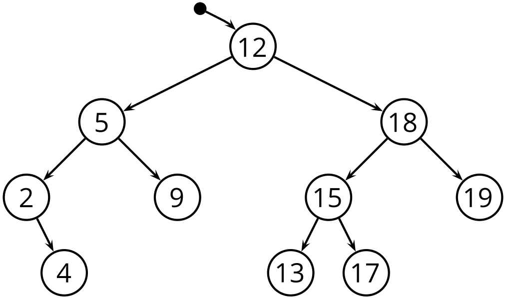
- *Implementation*
  - T represents the tree, which consists of a set of ***nodes***
  - $T.root$ is the root node of tree T

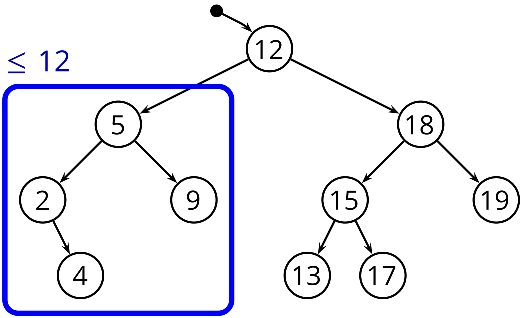
Node x

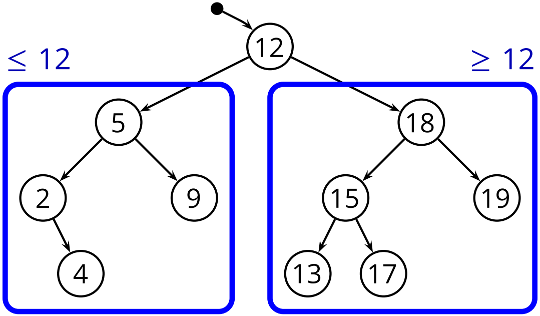
- $x.parent$ is the parent of node x
- $x.key$ is the key stored in node x
- $x.left$ is the left child of node x
- $x.right$ is the right child of node x

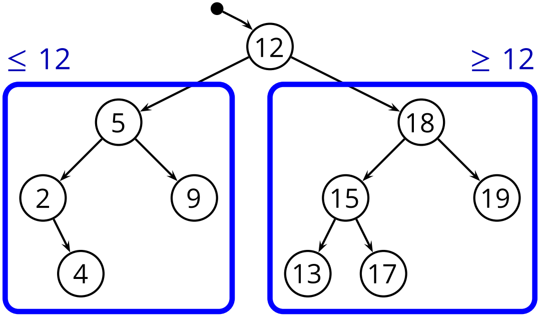


Binary Search Trees (3)



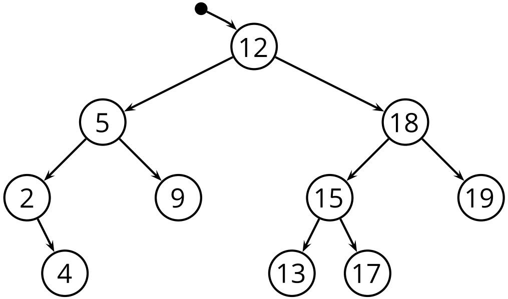

≤ 12
12
5
18
2
9
15
19
4
13
17


≤ 12
≥ 12
12
5
18
2
9
15
19
4
13
17

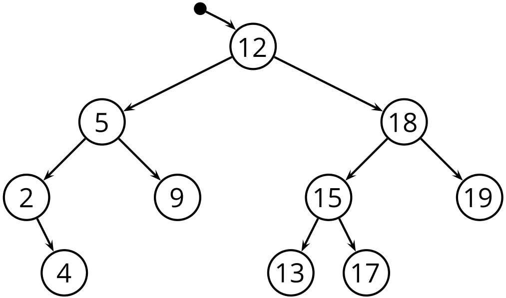


- ***Binary-search-tree property***
  - for all nodes x, y, and z
  - $y \in \textit{left-subtree}(x) \Rightarrow y.key \leq x.key$
  - $z \in \textit{right-subtree}(x) \Rightarrow z.key \geq x.key$

- We want to go through the set of keys *in order*



- We want to go through the set of keys *in order*



2 4 5 9 12 13 15 17 18 19

- A recursive algorithm

- A recursive algorithm

```
INORDER-TREE-WALK(x)
    if x ≠ NIL
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```

- A recursive algorithm

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

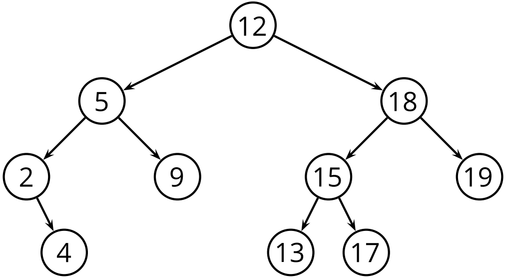
And then we need a “starter” procedure

```
INORDER-TREE-WALK-START(T)
1  INORDER-TREE-WALK(T.root)
```

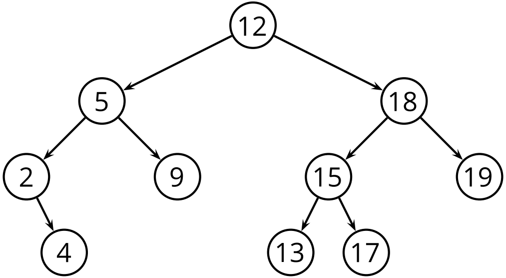
```
PREORDER-TREE-WALK(x)
    if x ≠ NIL
        print x.key
        PREORDER-TREE-WALK(x.left)
        PREORDER-TREE-WALK(x.right)
```

Preorder Tree Walk

```
PREORDER-TREE-WALK(x)
  if x ≠ NIL
      print x.key
      PREORDER-TREE-WALK(x.left)
      PREORDER-TREE-WALK(x.right)
```



```
PREORDER-TREE-WALK(x)
  if x ≠ NIL
      print x.key
      PREORDER-TREE-WALK(x.left)
      PREORDER-TREE-WALK(x.right)
```



12 5 2 4 9 18 15 13 17 19

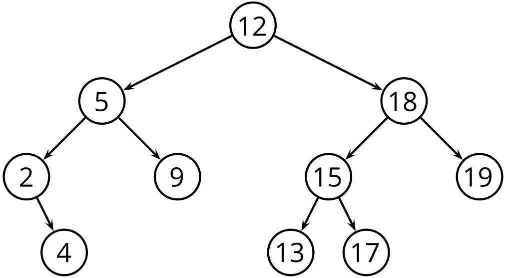
Postorder Tree Walk

Postorder Tree Walk

```
POSTORDER-TREE-WALK(x)
if x ≠ NIL
    POSTORDER-TREE-WALK(x.left)
    POSTORDER-TREE-WALK(x.right)
    print x.key
```

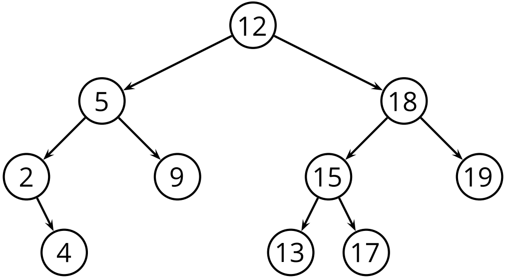
Postorder Tree Walk

```
POSTORDER-TREE-WALK(x)
    if x ≠ NIL
        POSTORDER-TREE-WALK(x.left)
        POSTORDER-TREE-WALK(x.right)
        print x.key
```



Postorder Tree Walk

```
POSTORDER-TREE-WALK(x)
if x ≠ NIL
    POSTORDER-TREE-WALK(x.left)
    POSTORDER-TREE-WALK(x.right)
    print x.key
```



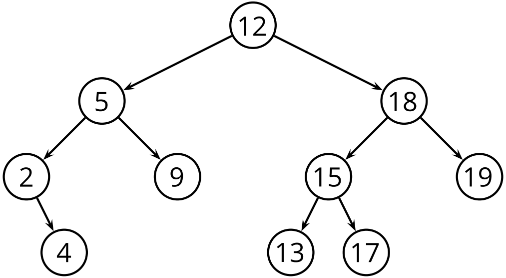
4 2 9 5 13 17 15 19 18 12

Reverse-Order Tree Walk

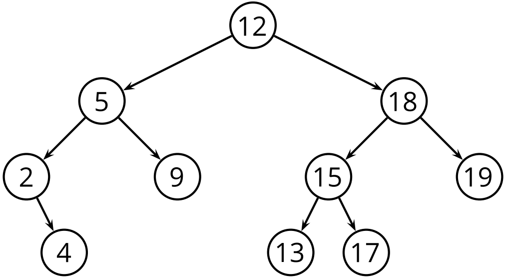
```
REVERSE-ORDER-TREE-WALK(x)
if x ≠ NIL
        REVERSE-ORDER-TREE-WALK(x.right)
        print x.key
        REVERSE-ORDER-TREE-WALK(x.left)
```

Reverse-Order Tree Walk

```
REVERSE-ORDER-TREE-WALK(x)
    if x ≠ NIL
        REVERSE-ORDER-TREE-WALK(x.right)
        print x.key
        REVERSE-ORDER-TREE-WALK(x.left)
```



```
REVERSE-ORDER-TREE-WALK(x)
  if x ≠ NIL
      REVERSE-ORDER-TREE-WALK(x.right)
      print x.key
      REVERSE-ORDER-TREE-WALK(x.left)
```



19 18 17 15 13 12 9 5 4 2

Application of postorder: Computing Arithmetic Expressions

- Arithmetic expressions can be represented by syntax trees.
- Given an expression represented by tree, compute its value !
- Each tree node: value field.
- Postorder traversal prints postfix notation/computes the value.

Complexity of Tree Walks

- The general recurrence is

$$T(n) = T(n_L) + T(n - n_L - 1) + \Theta(1)$$

- The general recurrence is

$$T(n) = T(n_L) + T(n - n_L - 1) + \Theta(1)$$

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

- The general recurrence is

$$T(n) = T(n_L) + T(n - n_L - 1) + \Theta(1)$$

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

We could prove this using the *substitution method*

- The general recurrence is

$$T(n) = T(n_L) + T(n - n_L - 1) + \Theta(1)$$

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

We could prove this using the *substitution method*

- Can we do better?

- The general recurrence is

$$T(n) = T(n_L) + T(n - n_L - 1) + \Theta(1)$$

INORDER-TREE-WALK	$\Theta(n)$
PREORDER-TREE-WALK	$\Theta(n)$
POSTORDER-TREE-WALK	$\Theta(n)$
REVERSE-ORDER-TREE-WALK	$\Theta(n)$

We could prove this using the *substitution method*

- Can we do better? No!
  - the length of the output is $\Theta(n)$

Minimum and Maximum Keys

- Recall the *binary-search-tree property*
  - for all nodes x, y, and z
  - $y \in \textit{left-subtree}(x) \Rightarrow y.\textit{key} \leq x.\textit{key}$
  - $z \in \textit{right-subtree}(x) \Rightarrow z.\textit{key} \geq x.\textit{key}$

Minimum and Maximum Keys

- Recall the *binary-search-tree property*
  - for all nodes x, y, and z
  - $y \in \textit{left-subtree}(x) \Rightarrow y.key \le x.key$
  - $z \in \textit{right-subtree}(x) \Rightarrow z.key \ge x.key$

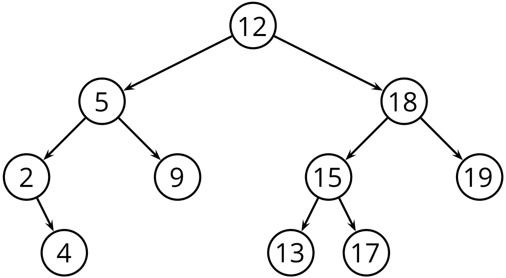
- So, the minimum key is in all the way to the left
  - similarly, the maximum key is all the way to the right

```
TREE-MINIMUM(x)
    while x.left ≠ NIL
        x = x.left
    return x
```

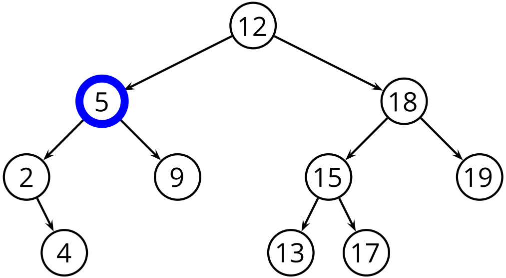
```
TREE-MAXIMUM(x)
    while x.right ≠ NIL
        x = x.right
    return x
```

- Given a node x, find the node containing the next key value

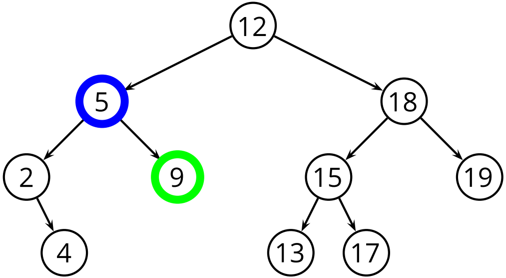
- Given a node *x*, find the node containing the next key value



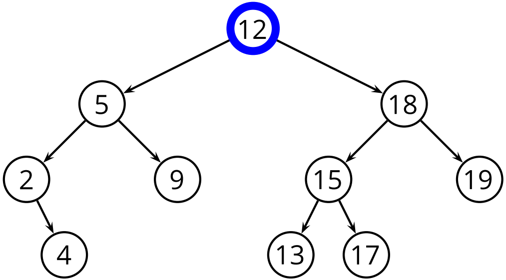
- Given a node *x*, find the node containing the next key value



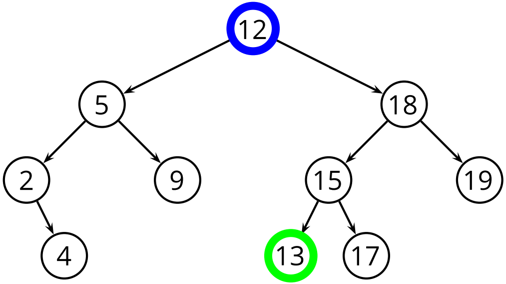
- Given a node *x*, find the node containing the next key value



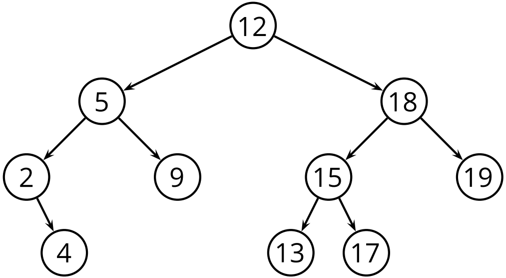
- Given a node *x*, find the node containing the next key value



- Given a node *x*, find the node containing the next key value

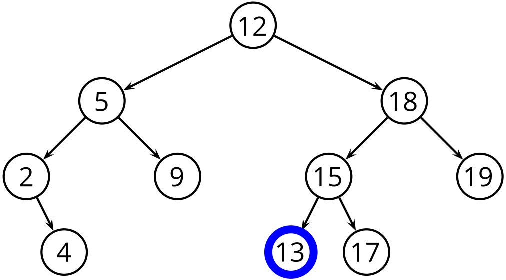


- Given a node *x*, find the node containing the next key value



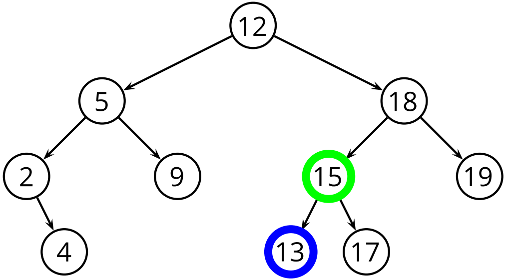
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

- Given a node *x*, find the node containing the next key value



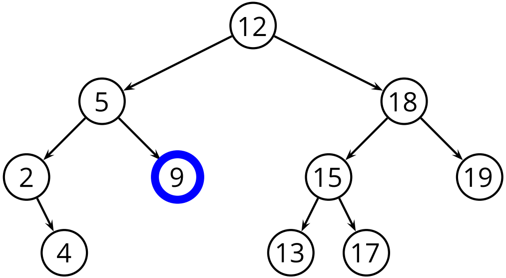
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

- Given a node *x*, find the node containing the next key value



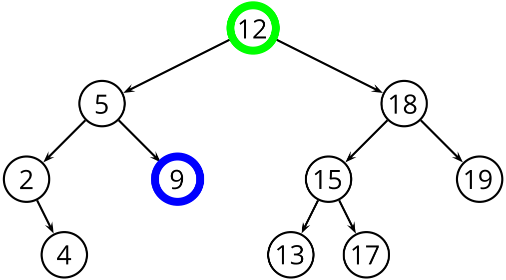
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

- Given a node *x*, find the node containing the next key value



- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

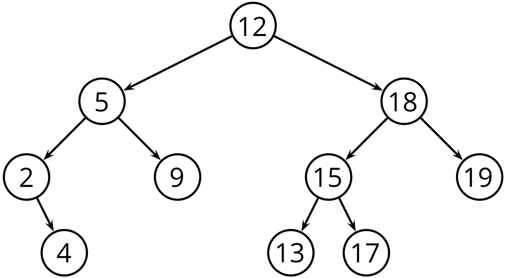
- Given a node x, find the node containing the next key value



- The successor of x is the *minimum* of the *right* subtree of x, if that exists

Successor and Predecessor

- Given a node *x*, find the node containing the next key value



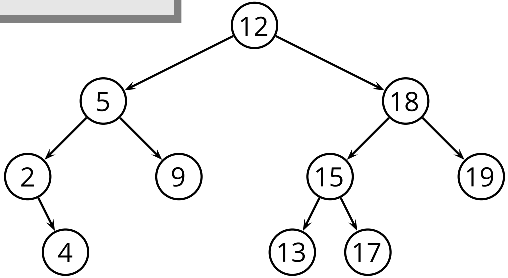
- The successor of *x* is the *minimum* of the *right* subtree of *x*, if that exists

- Otherwise it is the *first ancestor a* of *x* such that *x* falls in the *left* subtree of *a*

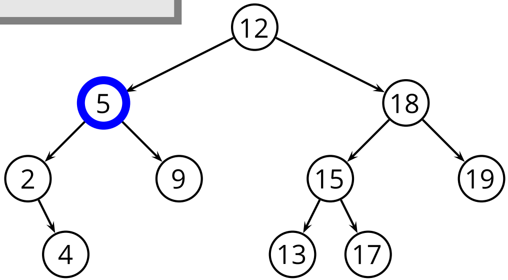
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```

Successor and Predecessor(2)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
    x = y
    y = y.parent
return y
```

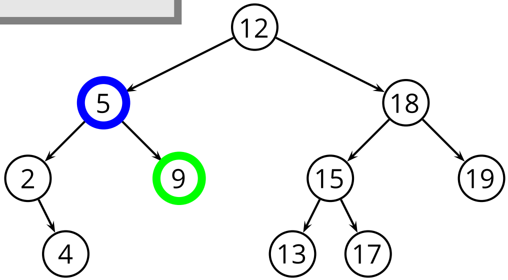


```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
  y = x.parent
  while y ≠ NIL and x = y.right
        x = y
        y = y.parent
  return y
```



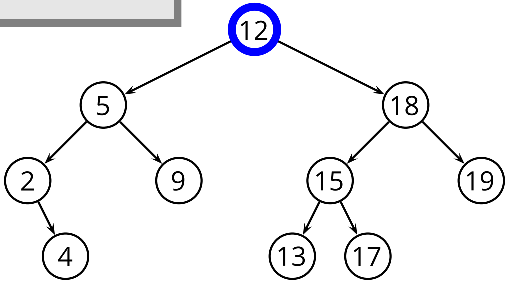
Successor and Predecessor(2)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```

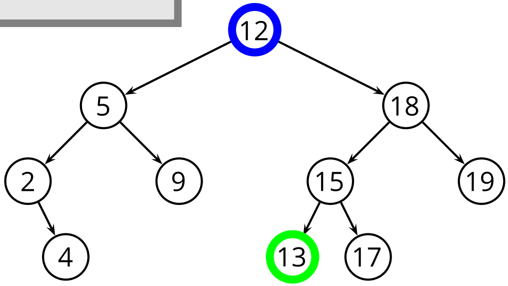


Successor and Predecessor(2)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```

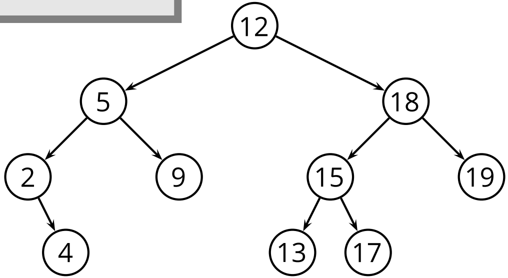


```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```

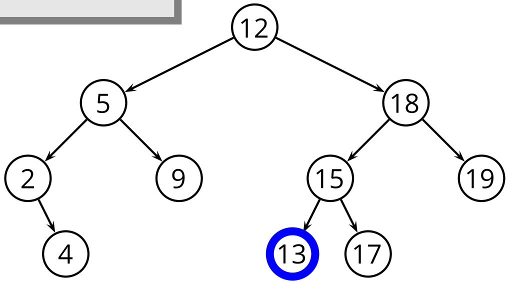


Successor and Predecessor(2)

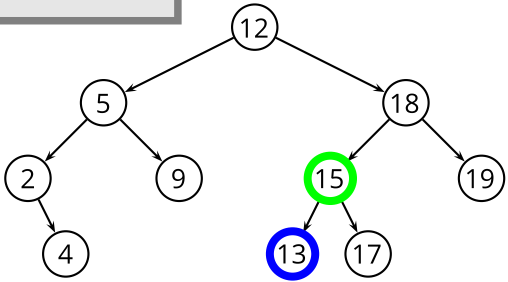
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
    x = y
    y = y.parent
return y
```



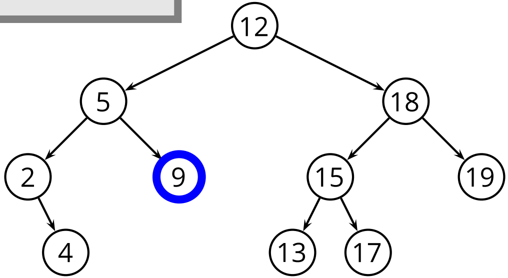
```
TREE-SUCCESSOR(x)
    if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
    y = x.parent
    while y ≠ NIL and x = y.right
        x = y
        y = y.parent
    return y
```



```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```

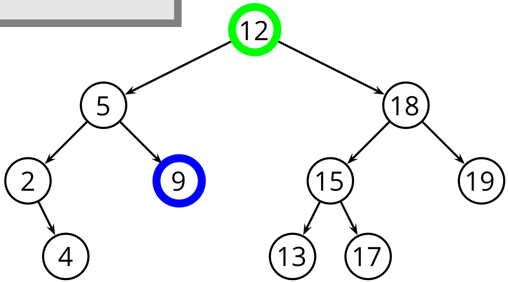


```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```



Successor and Predecessor(2)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
        return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x = y.right
        x = y
        y = y.parent
return y
```



- *Binary search* (thus the name of the tree)

- *Binary search* (thus the name of the tree)

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Search

- *Binary search* (thus the name of the tree)

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
        return x
if k < x.key
        return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Is this correct?

Search

- *Binary search* (thus the name of the tree)

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
        return x
if k < x.key
        return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Is this correct? Yes, thanks to the *binary-search-tree property*

- *Binary search* (thus the name of the tree)

```
TREE-SEARCH(x, k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

- Is this correct? Yes, thanks to the *binary-search-tree property*
- Complexity?

Search

- *Binary search* (thus the name of the tree)

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
      return x
if k < x.key
      return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Is this correct? Yes, thanks to the *binary-search-tree property*

- Complexity?

$$T(n) = \Theta(\textit{depth of the tree})$$

- *Binary search* (thus the name of the tree)

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
        return x
if k < x.key
        return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Is this correct? Yes, thanks to the *binary-search-tree property*
- Complexity?

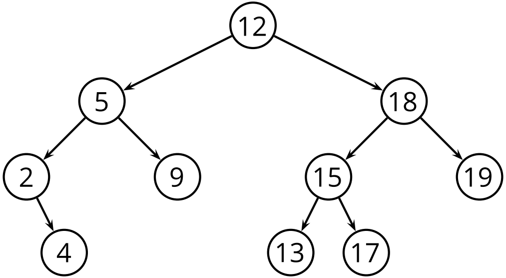
$$T(n) = \Theta(\textit{depth of the tree})$$

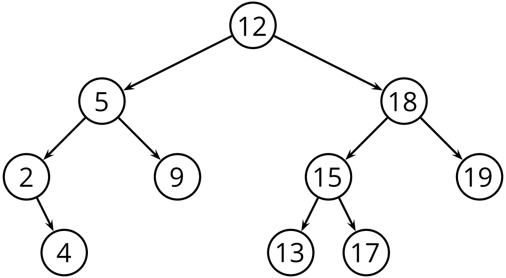
$$T(n) = O(n)$$

- Iterative *binary search*

- Iterative *binary search*

```
ITERATIVE-TREE-SEARCH(T, k)
x = T.root
while x ≠ NIL ∧ k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```


12
5
18
2
9
15
19
4
13
17



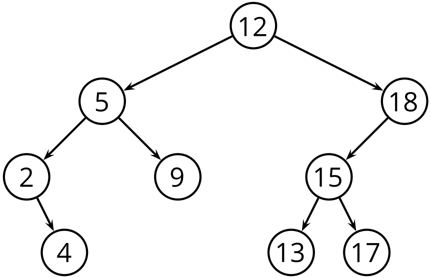
- *Idea*
  - in order to insert x, we *search* for x (more precisely $x.key$)
  - if we don't find it, we add it where the search stopped

Insertion (2)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

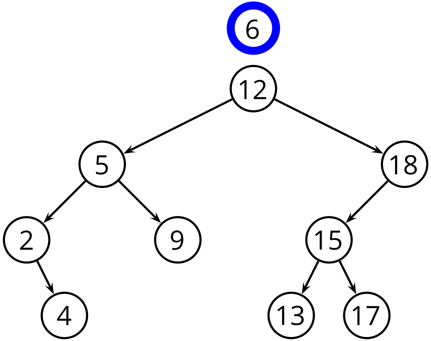
Insertion (2)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
else if z.key < y.key
        y.left = z
    else y.right = z
```



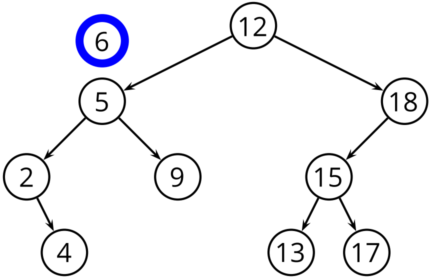
Insertion (2)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
else if z.key < y.key
        y.left = z
    else y.right = z
```



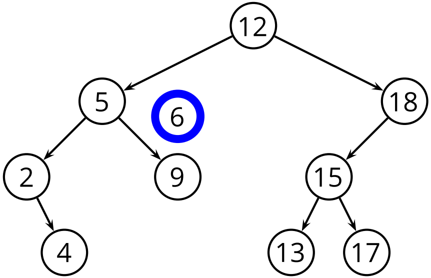
Insertion (2)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.parent = y
if y = NIL
        T.root = z
else if z.key < y.key
                y.left = z
        else y.right = z
```

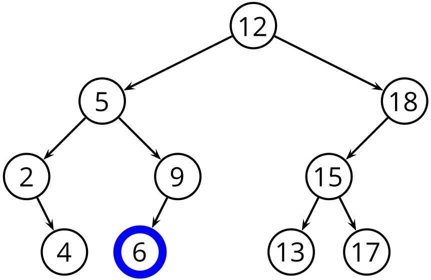


Insertion (2)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
else if z.key < y.key
        y.left = z
    else y.right = z
```

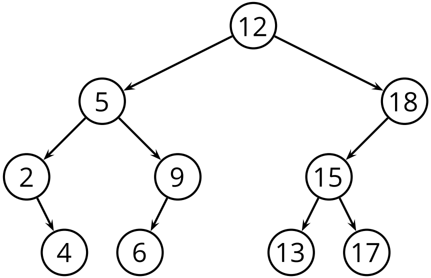


```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.parent = y
if y = NIL
      T.root = z
else if z.key < y.key
            y.left = z
      else y.right = z
```



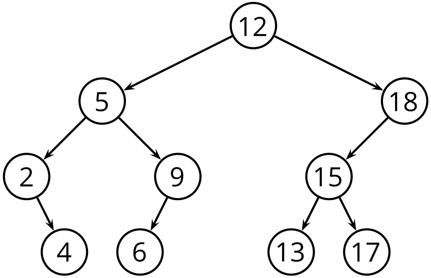
Insertion (2)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y = NIL
    T.root = z
else if z.key < y.key
        y.left = z
    else y.right = z
```



TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.parent = y
if y = NIL
      T.root = z
else if z.key < y.key
            y.left = z
      else y.right = z
```



$$T(n) = \Theta(h)$$

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
    - i.e., with a random insertion order
- $h = O(n)$ in some particular cases

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases
  - i.e., with ordered sequences

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases
  - i.e., with ordered sequences
  - the problem is that the "worst" case is not that uncommon

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases
  - i.e., with ordered sequences
  - the problem is that the “worst” case is not that uncommon
- *Idea:* use randomization to turn all cases in the average case

- *Idea 1:* insert every sequence as a random sequence

- *Idea 1:* insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a *random permutation* of A

- *Idea 1:* insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a *random permutation* of A
  - problem: A is not necessarily known in advance

- *Idea 1:* insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a *random permutation* of A
  - problem: A is not necessarily known in advance

- *Idea 2:* we can obtain a random permutation of the input sequence by randomly alternating two insertion procedures
  - *tail insertion:* this is what **TREE-INSERT** does

- *Idea 1:* insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a *random permutation* of A
  - problem: A is not necessarily known in advance
- *Idea 2:* we can obtain a random permutation of the input sequence by randomly alternating two insertion procedures
  - *tail insertion:* this is what **TREE-INSERT** does
  - *head insertion:* for this we need a new procedure **TREE-ROOT-INSERT**
    - inserts n in T as if n was inserted as the first element

```
TREE-RANDOMIZED-INSERT1(T, z)
    r = uniformly random value from {1, ..., t.size + 1
    if r = 1
        TREE-ROOT-INSERT(T, z)
    else TREE-INSERT(T, z)
```

TREE-RANDOMIZED-INSERT1(T, z)

```
r = uniformly random value from {1, ..., t.size + 1
if r = 1
        TREE-ROOT-INSERT(T, z)
else TREE-INSERT(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?

Randomized Insertion (2)

```
TREE-RANDOMIZED-INSERT1(T, z)
   r = uniformly random value from {1, ..., t.size + 1
   if r = 1
         TREE-ROOT-INSERT(T, z)
   else TREE-INSERT(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom

```
TREE-RANDOMIZED-INSERT1(T, z)
    r = uniformly random value from {1, ..., t.size + 1
    if r = 1
        TREE-ROOT-INSERT(T, z)
    else TREE-INSERT(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom
- It is true that any node has the same probability of being inserted at the top

```
TREE-RANDOMIZED-INSERT1(T, z)
    r = uniformly random value from {1, . . . , t.size + 1
    if r = 1
        TREE-ROOT-INSERT(T, z)
    else TREE-INSERT(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom

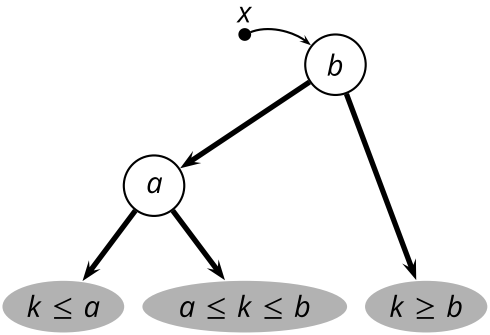
- It is true that any node has the same probability of being inserted at the top
  - this suggests a recursive application of this same procedure

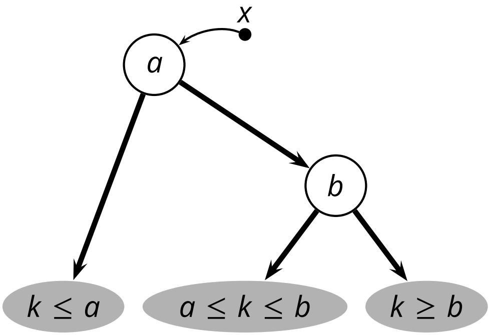
```
TREE-RANDOMIZED-INSERT(t, z)
if t = NIL
        return z
r = uniformly random value from {1, . . . , t.size + 1
if r = 1                                    // Pr[r = 1] = 1/(t.size + 1)
        z.size = t.size + 1
        return TREE-ROOT-INSERT(t, z)
if z.key < t.key
        t.left = TREE-RANDOMIZED-INSERT(t.left, z)
else t.right = TREE-RANDOMIZED-INSERT(t.right, z)
t.size = t.size + 1
return t
```

Randomized Insertion (3)

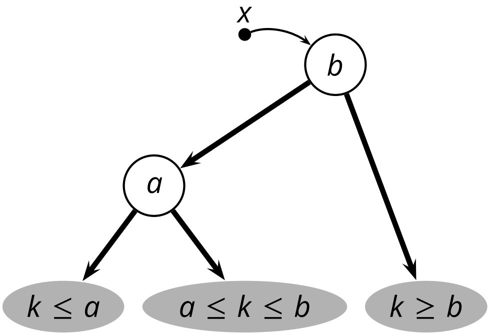
```
TREE-RANDOMIZED-INSERT(t, z)
if t = NIL
        return z
r = uniformly random value from {1, . . . , t.size + 1
if r = 1                                  // Pr[r = 1] = 1/(t.size + 1)
        z.size = t.size + 1
        return TREE-ROOT-INSERT(t, z)
if z.key < t.key
        t.left = TREE-RANDOMIZED-INSERT(t.left, z)
else t.right = TREE-RANDOMIZED-INSERT(t.right, z)
t.size = t.size + 1
return t
```

- Looks like this one really simulates a random permutation. . .

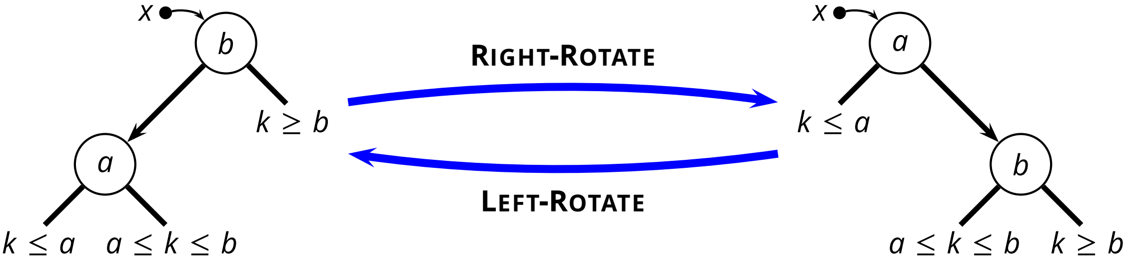

x
b
a
k ≤ a
a ≤ k ≤ b
k ≥ b



- $x = \textbf{Right-Rotate}(x)$



- x = **RIGHT-ROTATE**(x)
- x = **LEFT-ROTATE**(x)

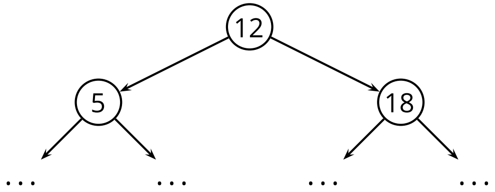


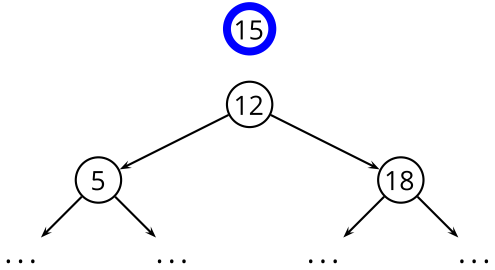
RIGHT-ROTATE(x)

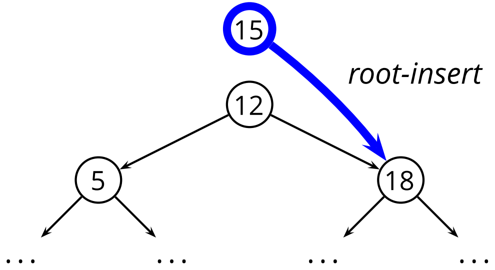
```
l = x.left
x.left = l.right
l.right = x
return l
```

LEFT-ROTATE(x)

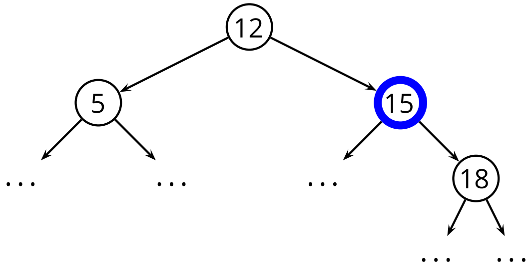
```
r = x.right
x.right = r.left
r.left = x
return r
```


12
5
18
. . .
. . .
. . .
. . .

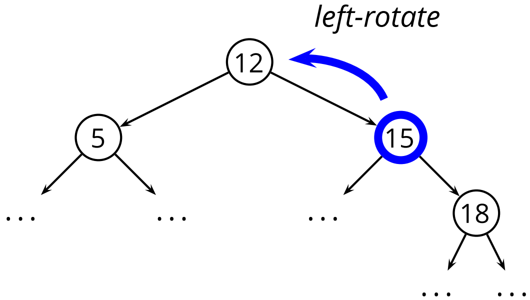

15
12
5
18
...
...
...
...



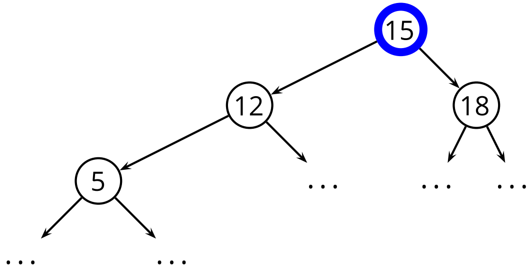
1. Recursively insert *z* at the root of the appropriate subtree (right)



1. Recursively insert z at the root of the appropriate subtree (right)



1. Recursively insert z at the root of the appropriate subtree (right)
2. Rotate x with z (left-rotate)



1. Recursively insert z at the root of the appropriate subtree (right)
2. Rotate x with z (left-rotate)

Root Insertion (2)

```
TREE-ROOT-INSERT(x, z)
if x = NIL
        return z
if z.key < x.key
        x.left = TREE-ROOT-INSERT(x.left, z)
        return RIGHT-ROTATE(x)
else x.right = TREE-ROOT-INSERT(x.right, z)
        return LEFT-ROTATE(x)
```

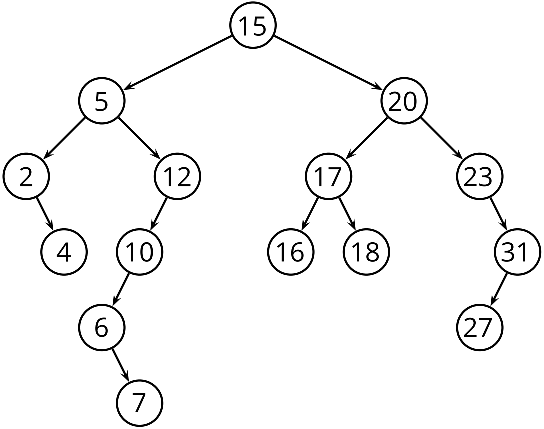
- General strategies to deal with complexity in the worst case

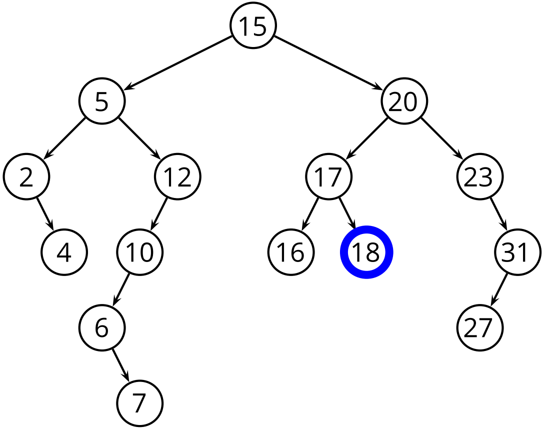
- General strategies to deal with complexity in the worst case
  - *randomization:* turns any case into the average case
    - the worst case is still possible, but it is extremely improbable

- General strategies to deal with complexity in the worst case
  - *randomization:* turns any case into the average case
    - the worst case is still possible, but it is extremely improbable
  - *amortized maintenance:* e.g., balancing a BST or resizing a hash table
    - relatively expensive but "amortized" operations

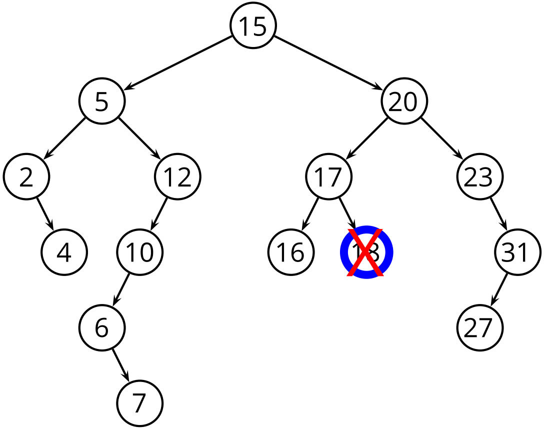
Observation

- General strategies to deal with complexity in the worst case
  - *randomization:* turns any case into the average case
    - the worst case is still possible, but it is extremely improbable
  - *amortized maintenance:* e.g., balancing a BST or resizing a hash table
    - relatively expensive but “amortized” operations
  - *optimized data structures:* a self-balanced data structure
    - guaranteed $O(\log n)$ complexity bounds

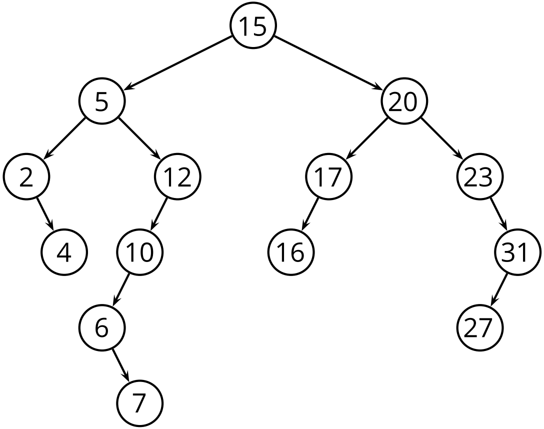

15
5
20
2
12
17
23
4
10
16
18
31
6
27
7



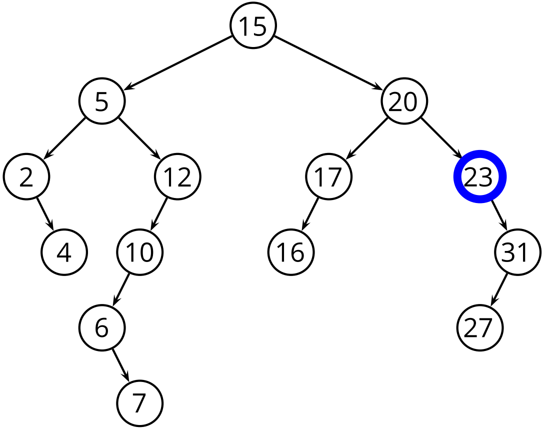
1. *z* has no children



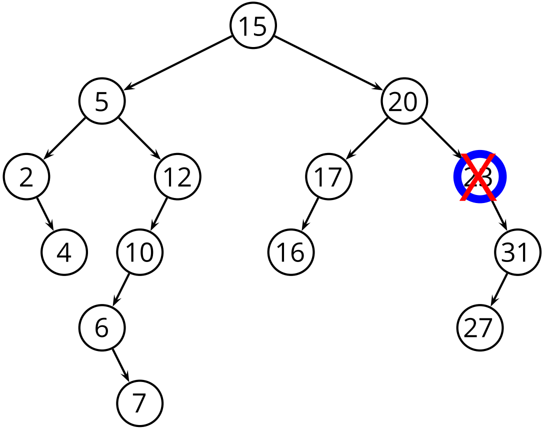
1. *z* has no children
   - simply remove *z*



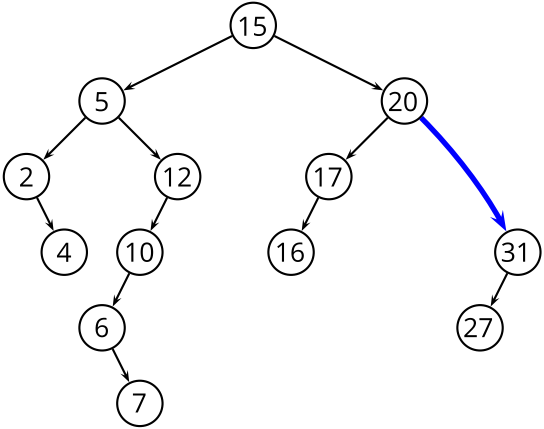
1. z has no children
   - simply remove z



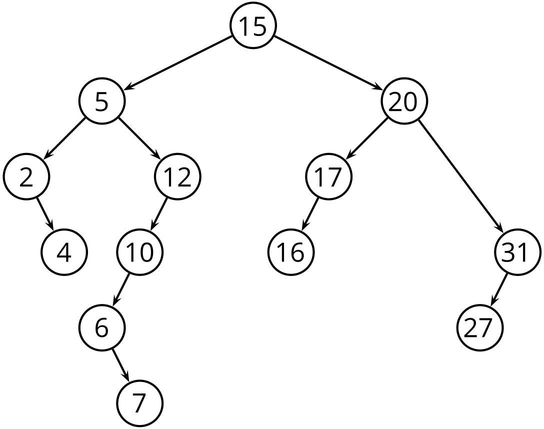
1. z has no children
   - simply remove z
2. z has one child



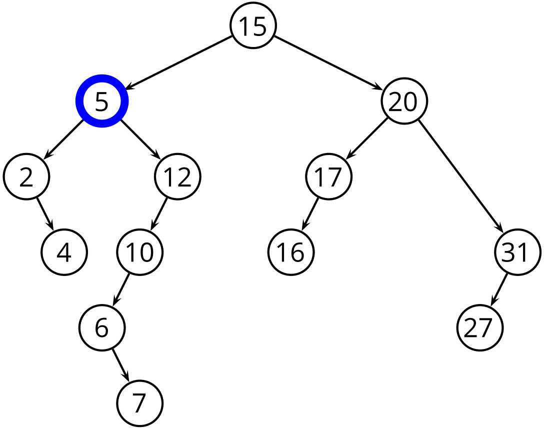
1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*



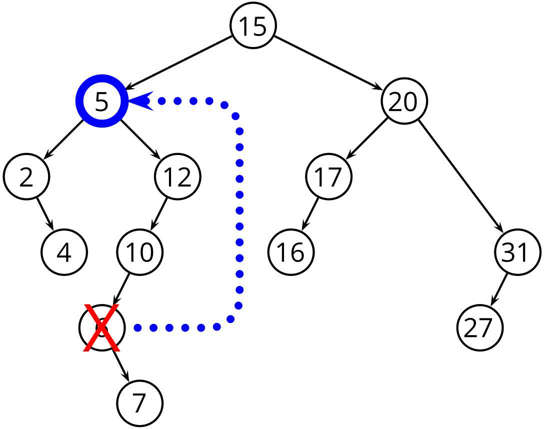
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$



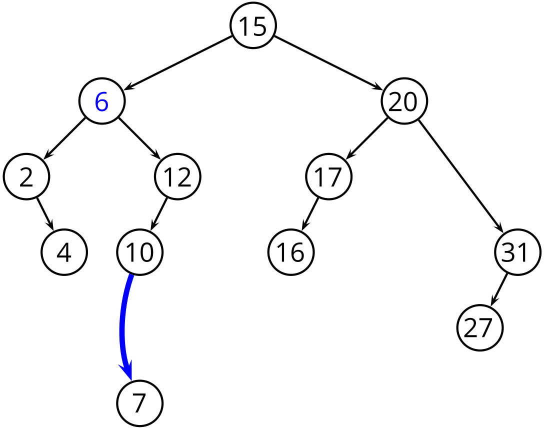
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children
   - replace z with $y = \textbf{TREE-SUCCESSOR}(z)$
   - remove y (1 child!)



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$
3. z has two children
   - replace z with $y =$ **Tree-Successor**(z)
   - remove y (1 child!)
   - connect $y.parent$ to $y.right$

```
TREE-DELETE(T, z)
if z.left = NIL or z.right = NIL
    y = z
else y = TREE-SUCCESSOR(z)
if y.left ≠ NIL
    x = y.left
else x = y.right
if x ≠ NIL
    x.parent = y.parent
if y.parent == NIL
    T.root = x
else if y = y.parent.left
        y.parent.left = x
    else y.parent.right = x
if y ≠ z
    z.key = y.key
    copy any other data from y into z
```