Data Structures

Gabriel Istrate
Course 5

Outline

- Hash tables.
- 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

Where are we ?

- 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

Where are we ?

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set
- Interface (generic interface)
  - $\mathrm{Insert}(D, k)$ adds a key k to the dictionary D
  - $\mathrm{Delete}(D, k)$ removes key k from D
  - $\mathrm{Search}(D, k)$ tells whether D contains a key k
- Implementation (so far)
  - direct access tables. Linked lists. (Skip lists). Hash tables.

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \rightarrow \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy $(O(1))$ to compute given k

```
Hash-Insert(T, k)
1   T[h(k)] = true
```

```
Hash-Delete(T, k)
1   T[h(k)] = false
```

```
Hash-Search(T, k)
1   return T[h(k)]
```

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

```
Hash-Insert(T, k)
1  T[h(k)] = true
```

```
Hash-Delete(T, k)
1  T[h(k)] = false
```

```
Hash-Search(T, k)
1  return T[h(k)]
```

Are these algorithms always correct?

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

```
Hash-Insert(T, k)
1  T[h(k)] = true
```

```
Hash-Delete(T, k)
1  T[h(k)] = false
```

```
Hash-Search(T, k)
1  return T[h(k)]
```

Are these algorithms always correct? No!

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \rightarrow \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

```
Hash-Insert(T, k)
1   T[h(k)] = true
```

```
Hash-Delete(T, k)
1   T[h(k)] = false
```

```
Hash-Search(T, k)
1   return T[h(k)]
```

Are these algorithms always correct? No! What if two distinct keys $k_1 \neq k_2$ collide? (I.e., $h(k_1) = h(k_2)$)

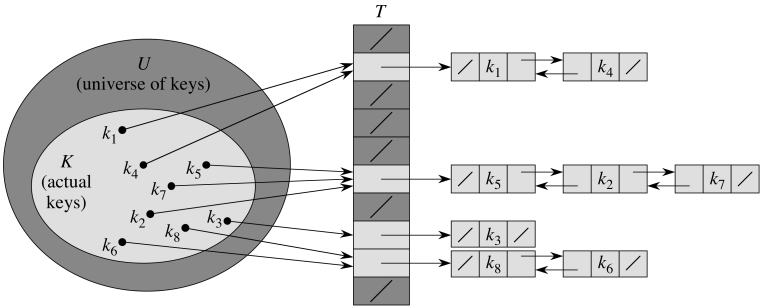
Hash tables

- Work well ”on the average”
- Analogy: throw T balls at random into N bins.
- If T << N (in fact $T = o(\sqrt{N})$) then with high-probability no two balls land in the same bin.
- On the average: T/N balls in each bin.
- Want our hash-function to be ”random-like”: elements of U ”thrown out uniformly” by h onto elements of T.

Hashing With Chaining

- Store all objects that map to the same bucket in a linked list.
- ”Hope” that hash function is ”uniform enough”, linked lists are not too large, set operations are efficient.

Hashing with Chaining



Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \ldots |T|\}$ (where $|T| = T.\text{length}$)

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \rightarrow \{1 \ldots |T|\}$ (where $|T| = T.\text{length}$)
$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$
(The formalism is actually a bit more complicated.)

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \dots |T|\}$ (where $|T| = T.\text{length}$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \rightarrow \{1 \ldots |T|\}$ (where $|T| = T.\text{length}$)

$$Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \dots |T|\}$ (where $|T| = T.\text{length}$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time

- Therefore, the complexity of Chained-Hash-Search is

$$\boxed{\Theta(1 + \alpha)}$$

Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists,store all the elements in the table
    - this implies $\alpha \leq 1$

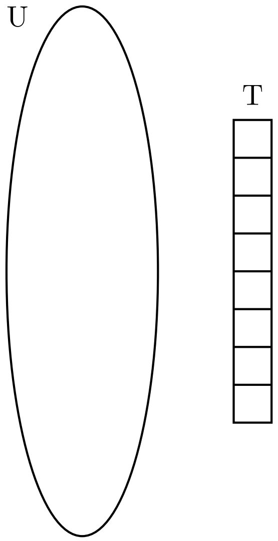
Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists,store all the elements in the table
  - this implies $\alpha \leq 1$
- When a collision occurs, simply find another free cell in T

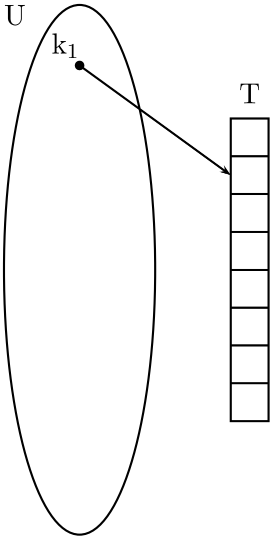
Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists,store all the elements in the table
    - this implies $\alpha \leq 1$
- When a collision occurs, simply find another free cell in T
- A sequential "probing" method may not be optimal
    - can you imagine why?

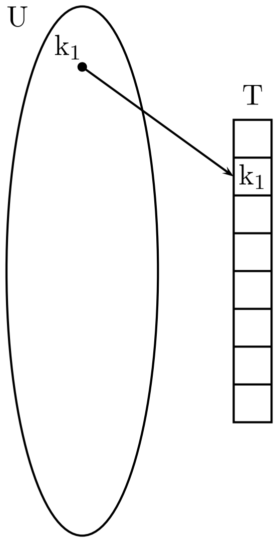
Open-Address Hash Table



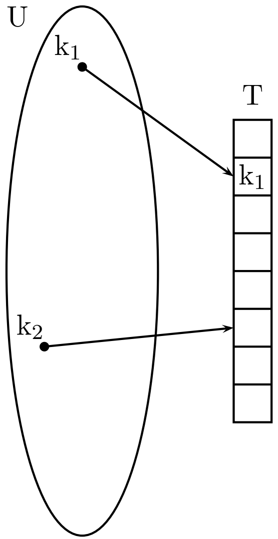
Open-Address Hash Table



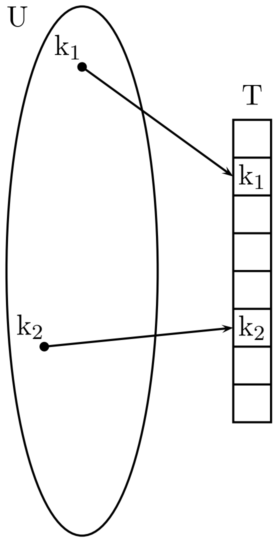
Open-Address Hash Table



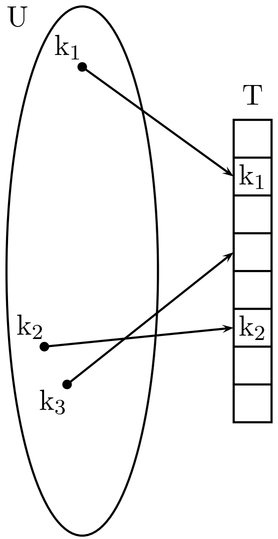
Open-Address Hash Table



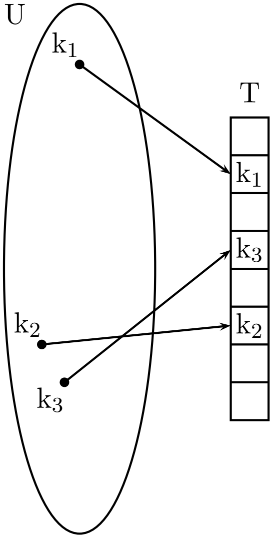
Open-Address Hash Table



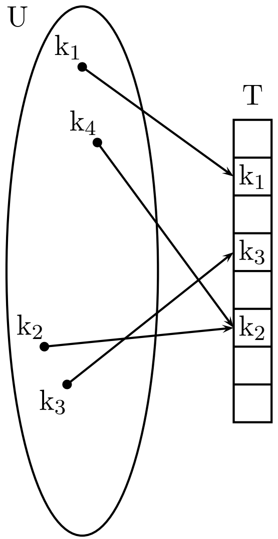
Open-Address Hash Table



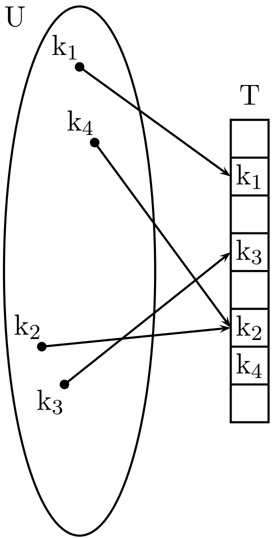
Open-Address Hash Table



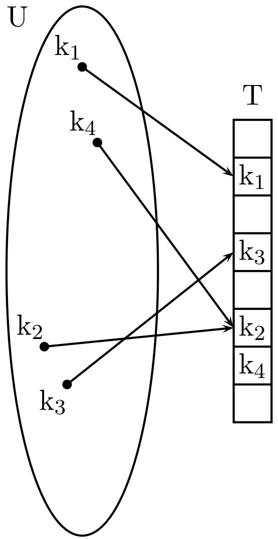
Open-Address Hash Table



Open-Address Hash Table



Open-Address Hash Table



```
Hash-Insert(T, k)
1   $j = h(k)$
2   for $i = 1$ to T.length
3       if $T[j] == nil$
4           $T[j] = k$
5           return j
6       elseif $j < T.length$
7           $j = j + 1$
8       else $j = 1$
9   error "overflow"
```

Open-Addressing (3)

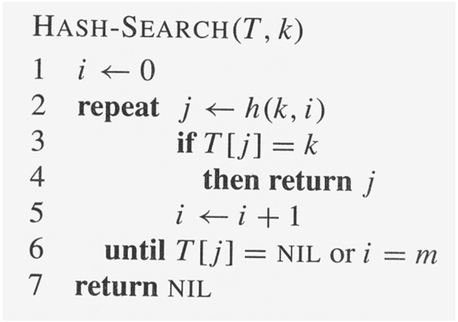
```
Hash-Insert(T, k)
1  for i = 1 to T.length
2      j = h(k, i)
3      if T[j] == nil
4          T[j] = k
5          return j
6  error "overflow"
```

Open-Addressing (3)

```
Hash-Insert(T, k)
1 for i = 1 to T.length
2 j = h(k, i)
3       if T[j] == nil
4             T[j] = k
5             return j
6 error "overflow"
```

- Notice that $h(k, \cdot)$ must be a permutation
  - i.e., $h(k, 1), h(k, 2), \ldots, h(k, |T|)$ must cover the entire table T

Procedure HASH-SEARCH



```
HASH-SEARCH($T, k$)
1  $i \leftarrow 0$
2  repeat  $j \leftarrow h(k, i)$
3             if $T[j] = k$
4                then return $j$
5             $i \leftarrow i + 1$
6     until $T[j] =$ NIL or $i = m$
7  return NIL
```

Open-address hashing

- Deletion: difficult. Marking NIL does not work.
- Doing so might make it impossible to retrieve any key during whose insertion probed slot i and found it occupied.
- One solution: DELETED instead of NIL. Problem: search time no longer dependent on load factor.
- Techniques for probing: linear probing, quadratic probing and double hashing.
- Linear probing: given auxiliary hash function $h' : U \rightarrow \{0, \ldots, m-1\}$, use hash function
$$h(k,i) = (h'(k) + i) \bmod m.$$
- Easy to implement but suffers from problem called primary clustering.
- Long runs of occupied slots build up, increasing average search time.

Open-address hashing

- Quadratic probing
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m.$$
- Works much better than linear probing, but to make use of full hash table the values of c_1, c_2, m are constrained.
- Suffers from secondary clustering: if two keys have the same initial probe position then their probe sequences are the same.

Open-address hashing

- Double hashing
$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$
- Among the best methods for open addressing.
- $h_2(k)$ must be relative prime to m. One solution is m a power of two and $h_2(k)$ odd.
- Another one: m prime, $h_2(k) < m$.
- Given an open address hash table with load factor $\alpha = n/m < 1$ the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Double hashing

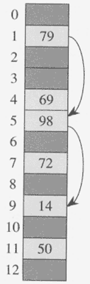


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Perfect hashing

- Hashing can provide worst-case performance when the set of keys is static: once stored in the table, the set of keys never changes.
- Example: set of files on a DVD-R (finished).
- Perfect hashing: the worst-case number of accesses to perform a search is O(1).
- Idea: two-level hashing with universal hashing at each level.
- First level: the n keys are hashed into m slots using a hashing function chosen from a family of universal hash functions.
- Instead of chaining: Use (small) secondary table S_j with an associated hash function h_j.
- Choosing h_j carefully guarantees no collisions.

Perfect hashing with chaining

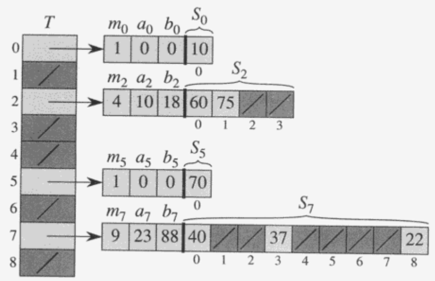


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

Perfect hashing: design

- n_j = number of elements that hash to slot j.
- We let $m_j = |S_j| = n_j^2$.
- Idea: if $m = n^2$ and we store n keys in a table of size $m = n^2$ using a hash function randomly chosen from a set of universal hash function then the collision probability is at most 1/2.
- Find a good hash function using O(1) trials.
- Expected amount of memory O(n).
- Why this works: proof omitted (see Cormen if curious).

Good hash functions: Single Hashing

Caution

- The area of designing good hash function huge.
- Theoreticians and practitioners as well.
- Many hash functions good for specific goal.
- Appearing next does not mean you should blindly use them !
- Drozdek: discusses some more "practical" examples.
- Here: we follow CORMEN, concentrate on general ideas.

Requirements

- A good hash function satisfies (approximately) the assumption of uniform hashing.
- Unfortunately, usually we don't know probability distribution of the keys, and keys might not be drawn independently.

Good hash functions

- Good case: if items are random real numbers k uniformly distributed in $[0, 1)$, $h(k) = \lfloor km \rfloor$ satisfies simple uniform hashing conditions.
- Most hash functions assume universe of keys are the natural numbers.
- E.g. character string = integer in base 128 notation.
- Identifier pt. ASCII $p = 112$, $t = 116$, becomes $112 \cdot 128 + 116 = 14452$.
- Division method: $h(k) = k \bmod m$. Avoid some values of m,e.g. powers of two. Indeed, if $m = 2^p$ then $h(k) =$ the p lowest bits of k. Unless we know that p lowest bits of keys are uniform not a good idea.
- Prime not too close to an exact power of two = often a good choice.

Folding

- Input: broken into pieces.
- Combined in some way.

Example

- SSN (American CNP): 123456789.
- Divide into three parts: 123, 456, 789.
- Add these: 1368.
- Reduced modulo table-size (1000): 368

Good hash functions: Multiplication method

- Multiplication method: Two stage procedure
- First, multiply key by constant A in range $0 < A < 1$, extract fractional part.
- Then multiply by m, extract floor.
- $h(k) = \lfloor m(kA \bmod 1) \rfloor$.
- Value of m not critical. $m = 2^p$.
- Easy implementation. Restrict $A = s/2^w$, w=machine word size.
- Better with some values of A than other. Knuth suggests $A \sim (\sqrt{5}-1)/2$ will work well.

Multiplication-method of hashing

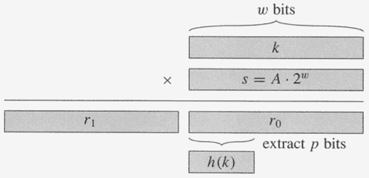


Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

Multiply-shift hashing

- $m = 2^l$, $l \leq w$, $w =$ word size.
- fixed w-bit positive integer $a = A2^w$, $0 < A < 1$.

Hashing $h = h_a(k)$ implemented as follows:

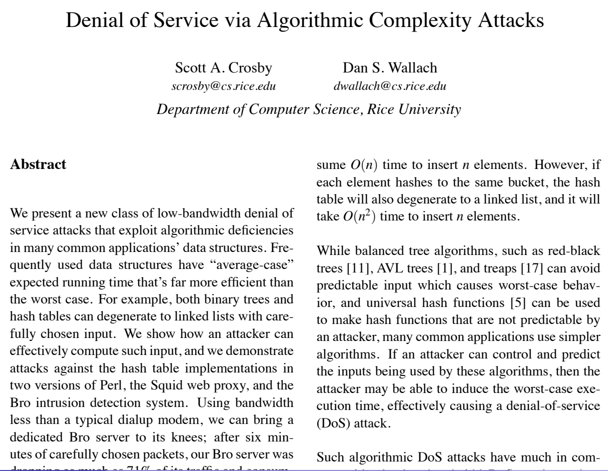
- First multiply k by w-bit a. Result: $r_1 \cdot 2^w + r_0$.
- $h_a(k) =$ most significant l bits of r_0.
- fast but no guarantees of performance.

$$h_a(k) = (ka \bmod 2^w) >>> (k - l).$$

Universal hashing

- Any fixed hash function vulnerable to worst case behavior:
- if ”adversary” chooses n keys that all hash to the same slot, this yields average search time of θ(n).
- Practical example: Crosby and Wallach (USENIX’03) have shown that one can slow down to a halt systems by attacking implementations of hash tables in Perl, squid web proxy, Bro intrusion detection.
- Cause: hashing mechanism known (due to, e.g. publicly available implementation).
- Solution: choose hash function randomly, independent of the keys that are going to be stored.

Attack when hashing mechanism known



Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby *scrosby@cs.rice.edu*

Dan S. Wallach *dwallach@cs.rice.edu*

Department of Computer Science, Rice University

Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. Frequently used data structures have "average-case" expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six minutes of carefully chosen packets, our Bro server was

sume $O(n)$ time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take $O(n^2)$ time to insert n elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Such algorithmic DoS attacks have much in com-

Universal hashing

- $\mathcal{H}$ finite collection of hash functions that map universe U into $\{0, 1, \ldots, m-1\}$.
- Such a collection is called universal if for every keys $k \neq l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}/m|$.
- ϵ-universal: probability of any fixed collision $\leq \epsilon$.
- Suppose a hash function h is chosen from a universal collection of hash functions, and is used to hash n keys into a table T of size m (using chaining).
- If key is not in the table expected length of the list that k hashes to is at most α.
- If key is not in the table expected length of the list that k hashes to is at most $1 + \alpha$.

A universal class of hash functions

- Due to Carter and Wegman.
- $a \equiv b \pmod{p}$ if $p | (a - b)$.
- Z_p: integers modulo p. p prime.
- How do we choose p ? So that all keys are in the range 0 to $p - 1$.
- m: number of slots in the hash table.
- $a \in Z_p^*$, $b \in Z_p$.
- $h_{a,b}(k) = ((ak + b) \pmod{p}) \pmod{m}$.
- $\mathcal{H}_{p,m} = \{h_{a,b} : a \in Z_p^*, b \in Z_p\}$.
- Other applications of this set of hash functions: pseudo-random generators.

Theoretical guarantee:

Theorem

Using universal hashing by chaining in an initially empty hash table with m slots, it takes $\Theta(s)$ expected time to handle s operations (INSERT, DELETE, SEARCH) containing $n = O(m)$ INSERT operations.

Proof: see Cormen.

Practical recommendation for universal hashing

$$\mathcal{H} = \{h_a : a \text{ is odd}, 1 \leq a < m \text{ and } h_a(k) = (ka \bmod 2^w) >>> w - 1$$

THEOREM: $\mathcal{H}$ is 2/m-universal.

- Weaker collision probability ...
- ... but fast to compute.

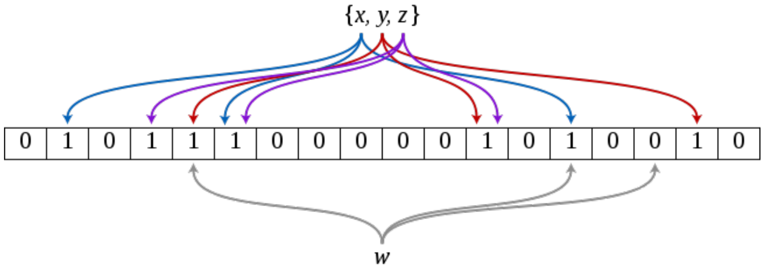
Hashing: there is more

- Cryptographic hash functions: hash functions with good security properties.
- Most well-known cryptographic hash function: md5 (Rabin). You probably have encountered it if you downloaded anything large from the web.
- (sha-1), sha-2, sha-3.
- SHA-256: NIST standard: 256-bit output.
- (Some) attacks on sha-1 (CWI Amsterdam, 2017)

SHA1(”The quick brown fox jumps over the lazy dog”) gives hexadecimal: 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12 gives Base64 binary to ASCII text encoding: L9ThxnotKPzthJ7hu3bnORuT6xI=

Bloom filters

- Probabilistic data structure. Used to test membership of an element in a dataset.
- NO answer: correct
- YES anwer: possibly false positive.

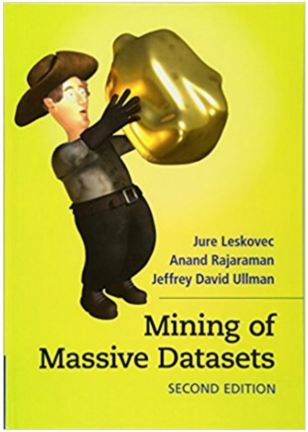


Hashing: where to go from here

- MapReduce model for grid computing: programming with hash functions.
- Data: key-value pairs.
- Map: applied in parallel to every pair (keyed by k1). Produces a list of pairs (keyed by k2) for each call.
- Mapreduce collects all pairs with the same k2 and groups them together.
- Reduce: applied in parallel to each group, which in turn produces a collection of values in the same domain.

Hashing: where to go from here

- Cormen last edition: hashing for hierarchical memory models wee hashing
- Locality-sensitive hashing: reduces the dimensionality of high-dimensional data. LSH hashes input items so that similar items map to the same 'buckets' with high probability.



Hashing in programming languages

- Python: dictionaries.
- Hash tables in STL: some implementations (e.g. SGI). Most functionality provided by associative container map (implemented using red-black trees).
- However: C++-11: two implementations, std::unordered_map and std::unordered_set.

Hash Tables: Scorecard

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. $\times$.

What can we do ?

Perhaps O(1(+α)) too ambitious ? Something, say O(logn)?

What can we do ?

Perhaps O(1(+α)) too ambitious ? Something, say O(logn)?

In practice log(n) is a small number !

Binary Search Trees

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

Binary Search Trees

- A binary search tree implements of a dynamic set
    - over a totally ordered domain
- Interface
    - $\text{Tree-Insert}(T, k)$ adds a key k to the dictionary D
    - $\text{Tree-Delete}(T, k)$ removes key k from D
    - $\text{Tree-Search}(T, x)$ tells whether D contains a key k

Binary Search Trees

- A binary search tree implements of a dynamic set
  - over a totally ordered domain
- Interface
  - $\mathrm{Tree\text{-}Insert}(T, k)$ adds a key k to the dictionary D
  - $\mathrm{Tree\text{-}Delete}(T, k)$ removes key k from D
  - $\mathrm{Tree\text{-}Search}(T, x)$ tells whether D contains a key k
  - tree-walk: $\mathrm{Inorder\text{-}Tree\text{-}Walk}(T)$, etc.

Binary Search Trees

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

- Interface
  - $\text{Tree-Insert}(T, k)$ adds a key k to the dictionary D
  - $\text{Tree-Delete}(T, k)$ removes key k from D
  - $\text{Tree-Search}(T, x)$ tells whether D contains a key k
  - tree-walk: $\text{Inorder-Tree-Walk}(T)$, etc.
  - $\text{Tree-Minimum}(T)$ finds the smallest element in the tree
  - $\text{Tree-Maximum}(T)$ finds the largest element in the tree

Binary Search Trees

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

- Interface
  - $\mathrm{Tree\text{-}Insert}(T, k)$ adds a key k to the dictionary D
  - $\mathrm{Tree\text{-}Delete}(T, k)$ removes key k from D
  - $\mathrm{Tree\text{-}Search}(T, x)$ tells whether D contains a key k
  - tree-walk: $\mathrm{Inorder\text{-}Tree\text{-}Walk}(T)$, etc.
  - $\mathrm{Tree\text{-}Minimum}(T)$ finds the smallest element in the tree
  - $\mathrm{Tree\text{-}Maximum}(T)$ finds the largest element in the tree
  - iteration: $\mathrm{Tree\text{-}Successor}(x)$ and $\mathrm{Tree\text{-}Predecessor}(x)$ find the successor and predecessor, respectively, of an element x

Binary Search Trees (2)

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

Binary Search Trees (2)

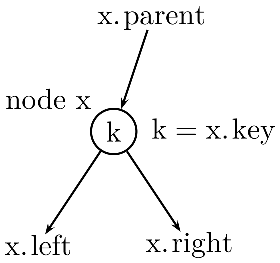
- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T

Binary Search Trees (2)

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

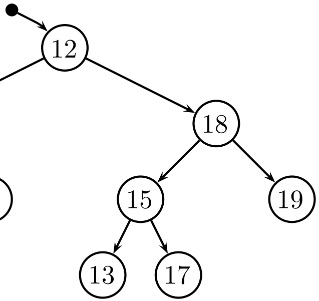
Node x

- 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

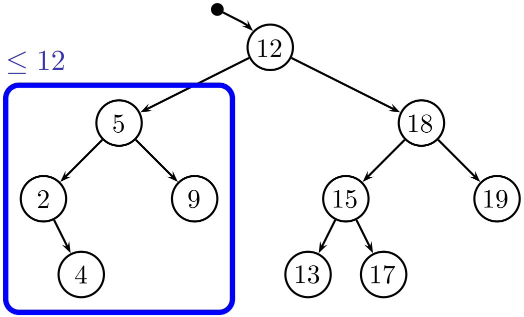


Binary Search Trees (3)

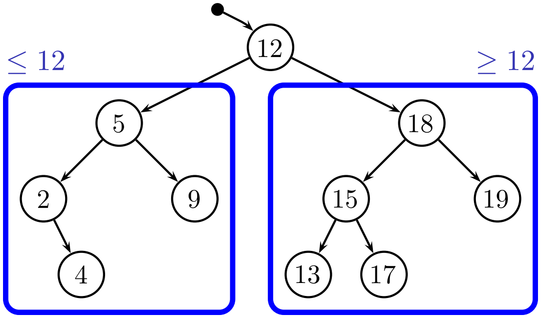
Binary Search Trees (3)



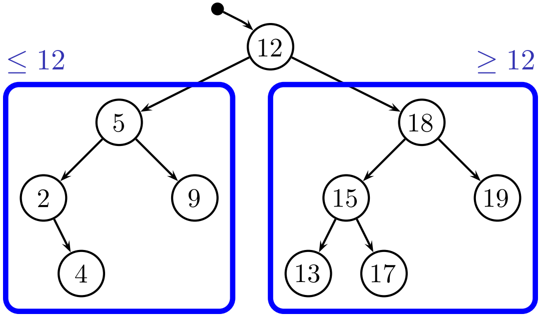
Binary Search Trees (3)



Binary Search Trees (3)



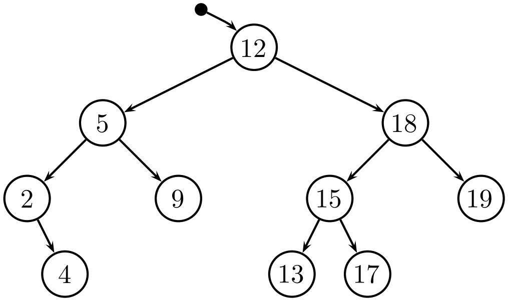
Binary Search Trees (3)



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

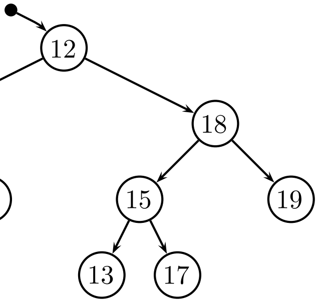
Inorder Tree Walk

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



Inorder Tree Walk

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



2 4 5 9 12 13 15 17 18 19

Inorder Tree Walk (2)

- A recursive algorithm

Inorder Tree Walk (2)

- 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)
```

Inorder Tree Walk (2)

- A recursive algorithm

```
Inorder-Tree-Walk(x)
1  if x $\neq$ 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)
```

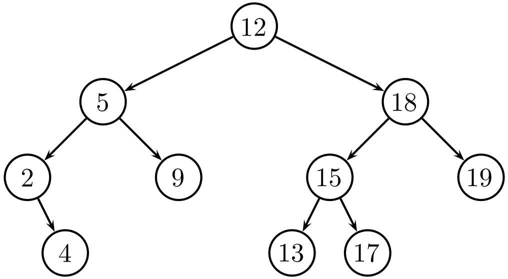
Preorder Tree Walk

Preorder Tree Walk

```
Preorder-Tree-Walk(x)
1  if $x \neq$ nil
2      print x.key
3      Preorder-Tree-Walk(x.left)
4      Preorder-Tree-Walk(x.right)
```

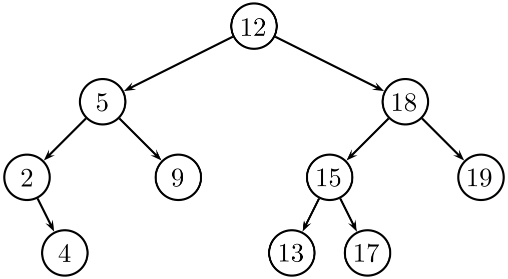
Preorder Tree Walk

```
Preorder-Tree-Walk(x)
1  if x ≠ nil
2      print x.key
3      Preorder-Tree-Walk(x.left)
4      Preorder-Tree-Walk(x.right)
```



Preorder Tree Walk

```
Preorder-Tree-Walk(x)
1  if x ≠ nil
2      print x.key
3      Preorder-Tree-Walk(x.left)
4      Preorder-Tree-Walk(x.right)
```



12 5 2 4 9 18 15 13 17 19

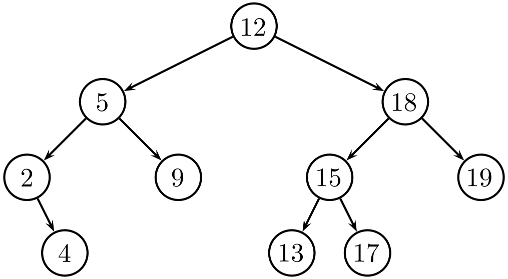
Postorder Tree Walk

Postorder Tree Walk

```
Postorder-Tree-Walk(x)
1  if x $\neq$ nil
2      Postorder-Tree-Walk(x.left)
3      Postorder-Tree-Walk(x.right)
4      print x.key
```

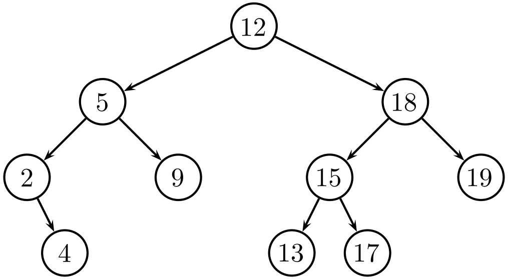
Postorder Tree Walk

```
Postorder-Tree-Walk(x)
1  if x ≠ nil
2        Postorder-Tree-Walk(x.left)
3        Postorder-Tree-Walk(x.right)
4        print x.key
```



Postorder Tree Walk

```
Postorder-Tree-Walk(x)
1  if x ≠ nil
2      Postorder-Tree-Walk(x.left)
3      Postorder-Tree-Walk(x.right)
4      print x.key
```



4 2 9 5 13 17 15 19 18 12

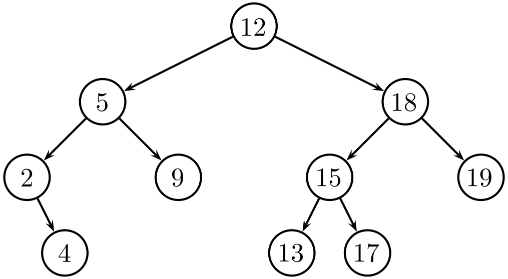
Reverse-Order Tree Walk

Reverse-Order Tree Walk

```
Reverse-Order-Tree-Walk(x)
1  if x $\neq$ nil
2      Reverse-Order-Tree-Walk(x.right)
3      print x.key
4      Reverse-Order-Tree-Walk(x.left)
```

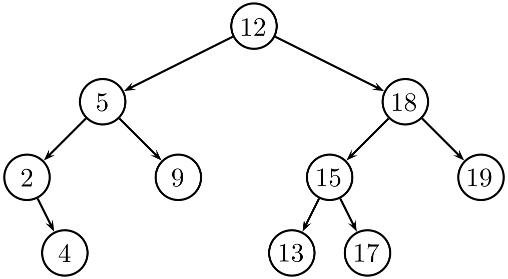
Reverse-Order Tree Walk

```
Reverse-Order-Tree-Walk(x)
1  if x $\neq$ nil
2      Reverse-Order-Tree-Walk(x.right)
3      print x.key
4      Reverse-Order-Tree-Walk(x.left)
```



Reverse-Order Tree Walk

```
Reverse-Order-Tree-Walk(x)
1   if x $\neq$ nil
2       Reverse-Order-Tree-Walk(x.right)
3       print x.key
4       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

Complexity of Tree Walks

- The general recurrence is

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

Complexity of Tree Walks

- 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	$\Theta(n)$

Complexity of Tree Walks

- 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

Complexity of Tree Walks

- 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?

Complexity of Tree Walks

- 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

Minimum and Maximum Keys

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

Minimum and Maximum Keys

- Recall the binary-search-tree property
  - for all nodes x, y, and z
  - $y \in \text{left-subtree}(x) \Rightarrow y.key \leq x.key$
  - $z \in \text{right-subtree}(x) \Rightarrow z.key \geq x.key$
- 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)
1  while $x.left \neq nil$
2      $x = x.left$
3  return x
```

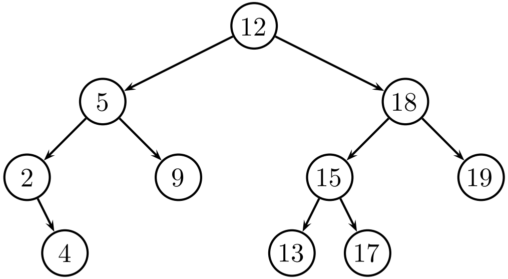
```
Tree-Maximum(x)
1  while $x.right \neq nil$
2      $x = x.right$
3  return x
```

Successor and Predecessor

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

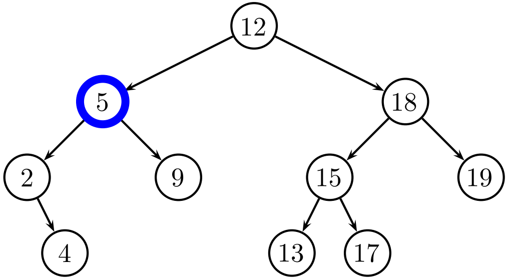
Successor and Predecessor

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



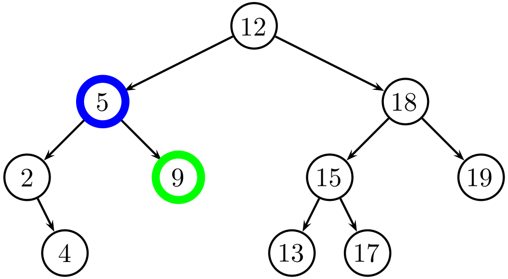
Successor and Predecessor

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



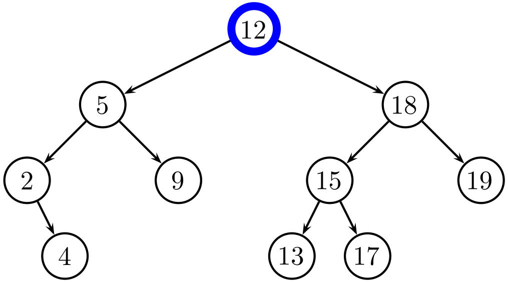
Successor and Predecessor

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



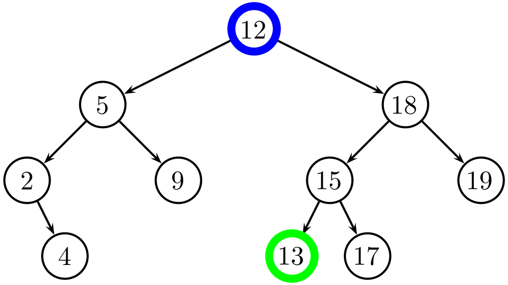
Successor and Predecessor

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



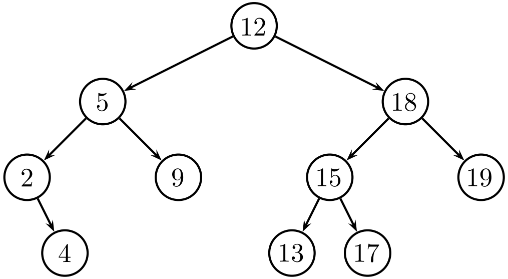
Successor and Predecessor

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



Successor and Predecessor

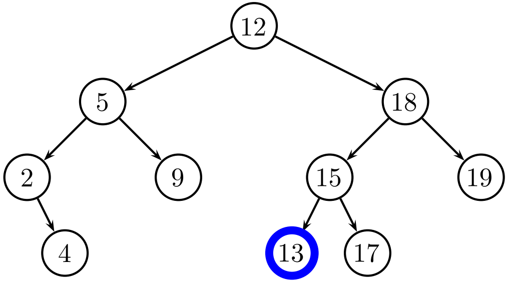
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

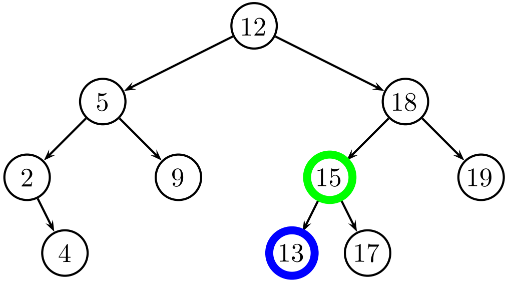
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

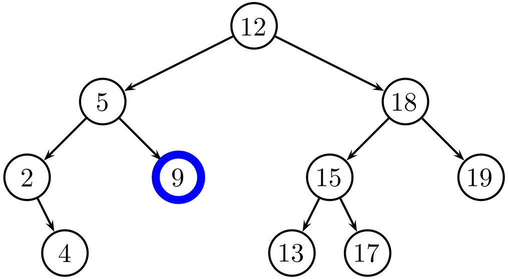
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

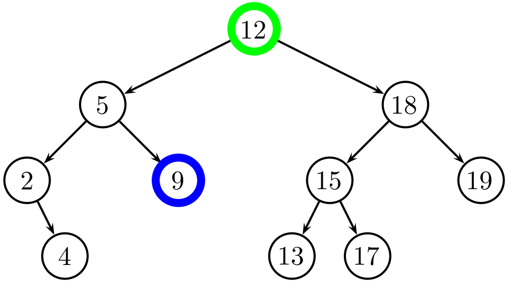
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

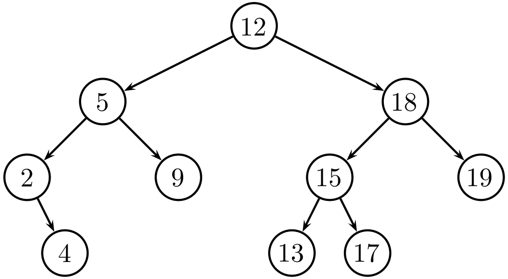
- Given a node x, find the node containing the next key value



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

Successor and Predecessor

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



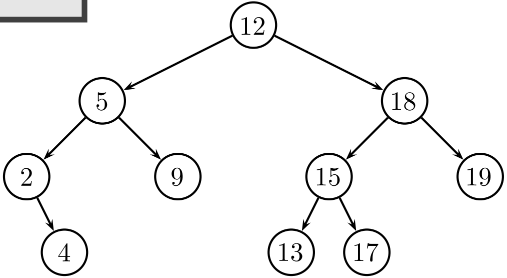
- The successor of x is the minimum of the right subtree of x, if that exists
- Otherwise it is the first ancestor a of x such that x falls in the left subtree of a

Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right $\neq$ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y $\neq$ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```

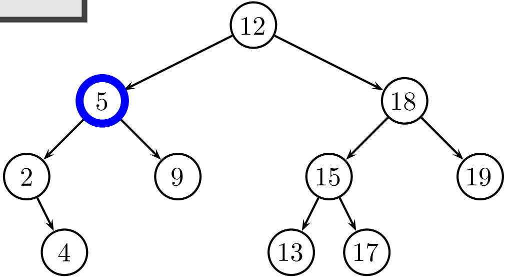
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right $\neq$ nil
2       return Tree-Minimum(x.right)
3  y = x.parent
4  while y $\neq$ nil and x = y.right
5       x = y
6       y = y.parent
7  return y
```



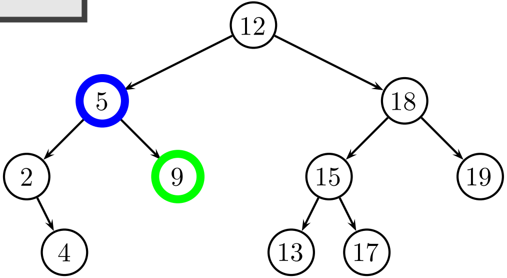
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



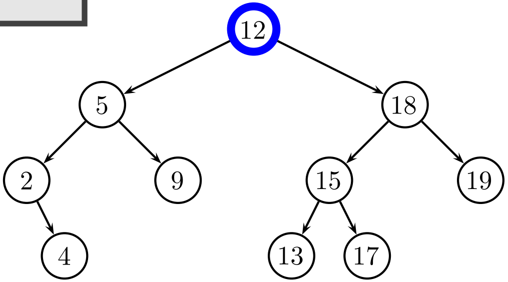
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



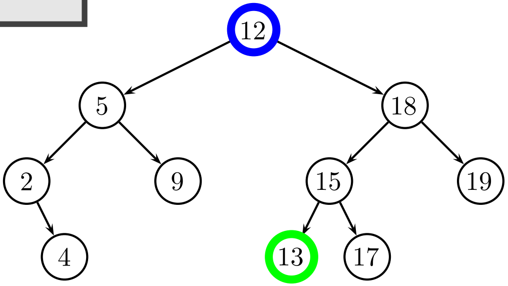
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



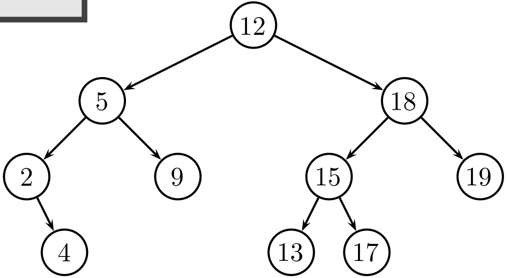
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



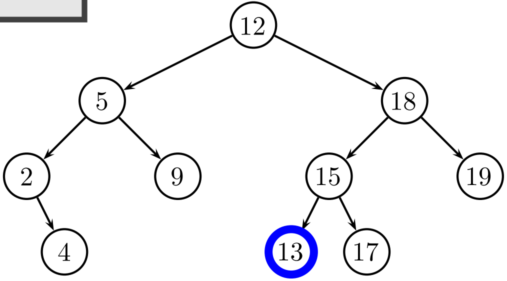
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



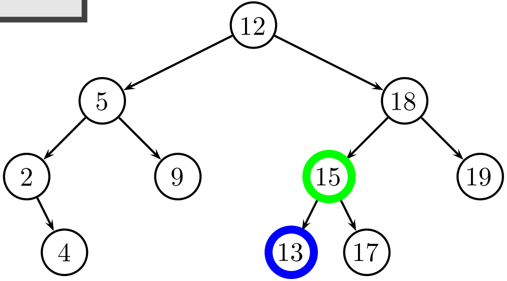
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



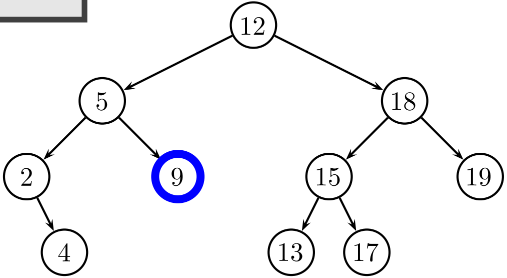
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



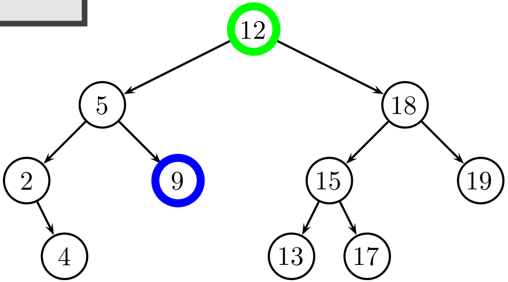
Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2      return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5      x = y
6      y = y.parent
7  return y
```



Successor and Predecessor(2)

```
Tree-Successor(x)
1  if x.right ≠ nil
2        return Tree-Minimum(x.right)
3  y = x.parent
4  while y ≠ nil and x = y.right
5        x = y
6        y = y.parent
7  return y
```



Search

Search

- Binary search (thus the name of the tree)

Search

- 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

Search

- 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)
```

- 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)
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)
```

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

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

Search

- 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)
```

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

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

$$\boxed{T(n) = O(n)}$$

Search (2)

Search (2)

- Iterative binary search

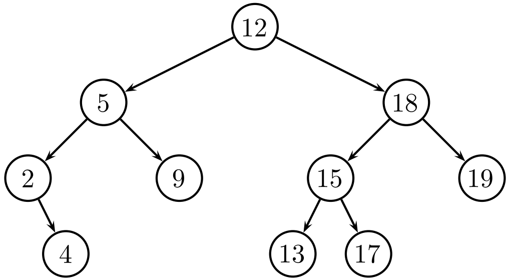
Search (2)

- Iterative binary search

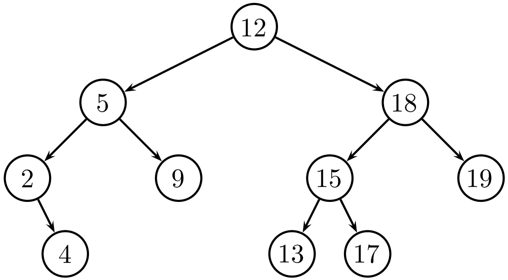
```
Iterative-Tree-Search(T, k)
1   x = T.root
2   while x ≠ nil ∧ k ≠ x.key
3       if k < x.key
4           x = x.left
5       else x = x.right
6   return x
```

Insertion

Insertion



Insertion



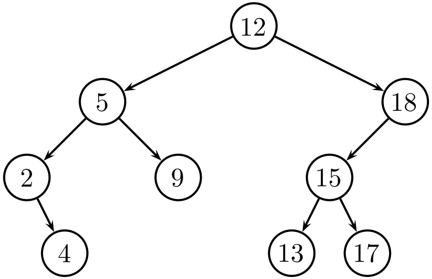
- Idea
  - in order to insert x, we search for x (more precisely x. key)
  - if we don't find it, we add it where the search stopped

Insertion (2)

```
Tree-Insert(T, z)
    y = nil
    x = T.root
    while x $\neq$ 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
```

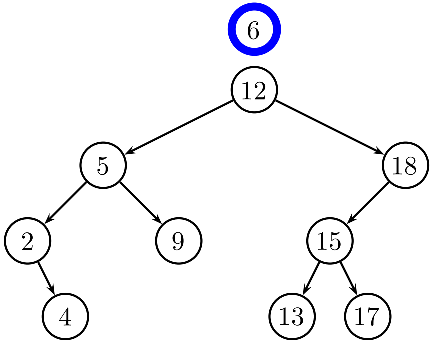
Insertion (2)

```
Tree-Insert(T, z)
1   y = nil
2   x = T.root
3   while x ≠ nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.parent = y
9   if y = nil
10      T.root = z
11  else if z.key < y.key
12          y.left = z
13      else y.right = z
```



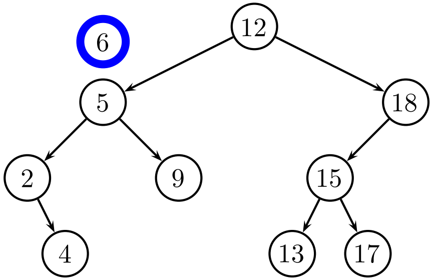
Insertion (2)

```
Tree-Insert(T, z)
1   y = nil
2   x = T.root
3   while x ≠ nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.parent = y
9   if y = nil
10      T.root = z
11  else if z.key < y.key
12          y.left = z
13      else y.right = z
```



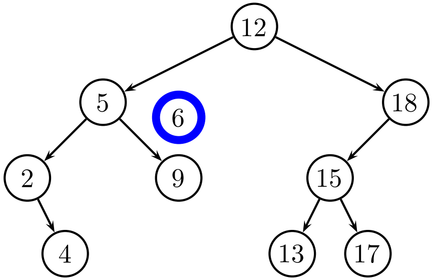
Insertion (2)

```
Tree-Insert(T, z)
1   y = nil
2   x = T.root
3   while x ≠ nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.parent = y
9   if y = nil
10      T.root = z
11  else if z.key < y.key
12          y.left = z
13      else y.right = z
```



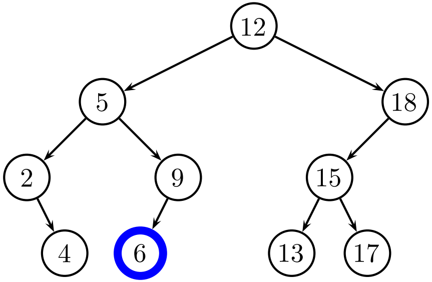
Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y = nil
10        T.root = z
11  else if z.key < y.key
12              y.left = z
13        else y.right = z
```



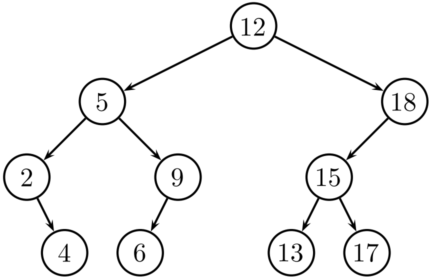
Insertion (2)

```
Tree-Insert(T, z)
1   y = nil
2   x = T.root
3   while x ≠ nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.parent = y
9   if y = nil
10      T.root = z
11  else if z.key < y.key
12          y.left = z
13      else y.right = z
```



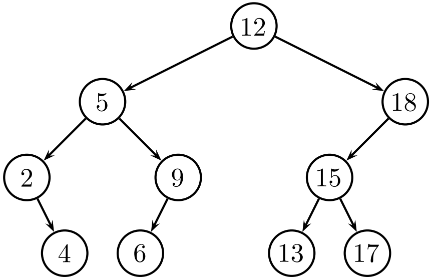
Insertion (2)

```
Tree-Insert(T, z)
1   y = nil
2   x = T.root
3   while x $\neq$ nil
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.parent = y
9   if y == nil
10      T.root = z
11  else if z.key < y.key
12          y.left = z
13      else y.right = z
```



Insertion (2)

```
Tree-Insert(T, z)
 1  y = nil
 2  x = T.root
 3  while x ≠ nil
 4          y = x
 5          if z.key < x.key
 6                  x = x.left
 7          else x = x.right
 8  z.parent = y
 9  if y = nil
10          T.root = z
11  else if z.key < y.key
12                  y.left = z
13          else y.right = z
```



$$\boxed{T(n) = \Theta(h)}$$

Observation

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

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

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

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

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

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

Randomized Insertion

- Idea 1: insert every sequence as a random sequence

Randomized Insertion

- 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

Randomized Insertion

- 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

Randomized Insertion

- 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

Randomized Insertion

- 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

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly rand. val. from $\{1, \ldots, t.size + 1\}$
2  if r = 1
3      Tree-Root-Insert(T, z)
4  else Tree-Insert(T, z)
```

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly rand. val. from $\{1, \ldots, t.size + 1\}$
2  if $r = 1$
3      Tree-Root-Insert(T, z)
4  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)
1  r = uniformly rand. val. from $\{1, \ldots, t.size + 1\}$
2  if r = 1
3      Tree-Root-Insert(T, z)
4  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

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly rand. val. from $\{1, \ldots, t.size + 1\}$
2  if r = 1
3        Tree-Root-Insert(T, z)
4  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

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)
1  r = uniformly rand. val. from $\{1, \ldots, t.size + 1\}$
2  if r = 1
3      Tree-Root-Insert(T, z)
4  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

Randomized Insertion (3)

Randomized Insertion (3)

```
Tree-Randomized-Insert(t, z)
 1  if $t = nil$
 2      return z
 3  $r$ = uniformly random value from $\{1, \ldots, t.size + 1\}$
 4  if $r = 1$                      // $\Pr[r = 1] = 1/(t.size + 1)$
 5      $z.size = t.size + 1$
 6      return Tree-Root-Insert(t, z)
 7  if $z.key < t.key$
 8      $t.left$ = Tree-Randomized-Insert($t.left$, z)
 9  else $t.right$ = Tree-Randomized-Insert($t.right$, z)
10  $t.size = t.size + 1$
11  return t
```

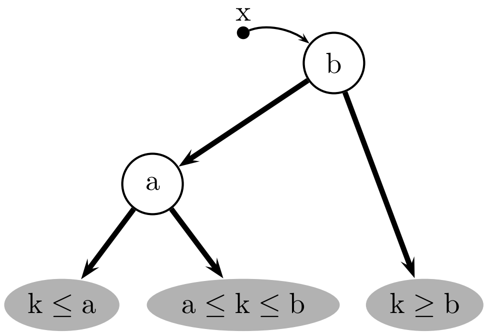
Randomized Insertion (3)

```
Tree-Randomized-Insert(t, z)
 1  if t = nil
 2      return z
 3  r = uniformly random value from $\{1, \dots, t.size + 1\}$
 4  if r = 1                          // $Pr[r = 1] = 1/(t.size + 1)$
 5      z.size = t.size + 1
 6      return Tree-Root-Insert(t, z)
 7  if z.key < t.key
 8      t.left = Tree-Randomized-Insert(t.left, z)
 9  else t.right = Tree-Randomized-Insert(t.right, z)
10  t.size = t.size + 1
11  return t
```

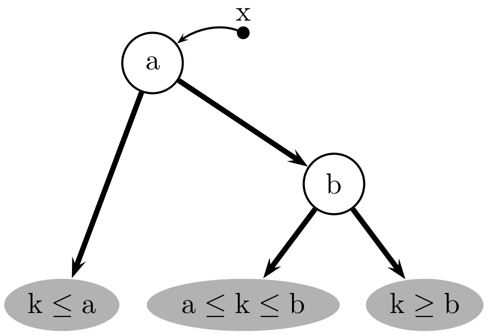
- Looks like this one really simulates a random permutation...

Rotation

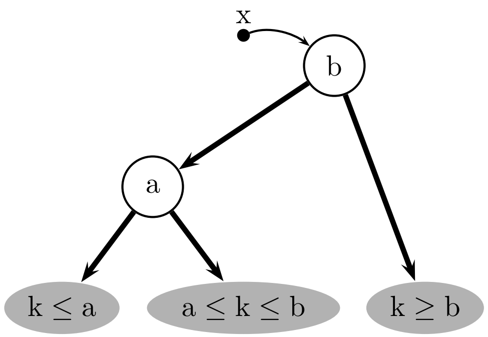
Rotation



Rotation

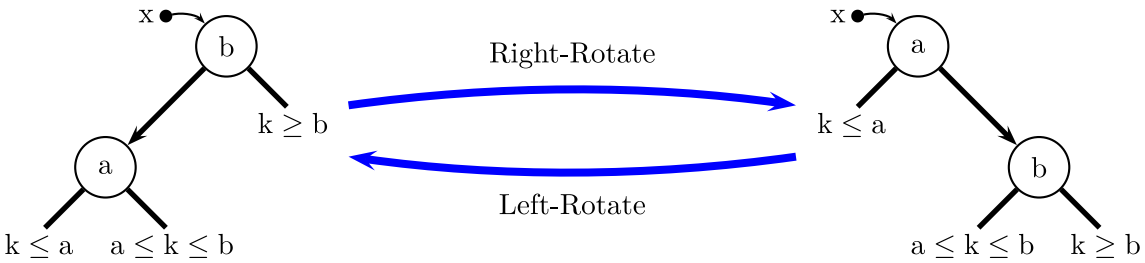


- x = Right-Rotate(x)



- x = Right-Rotate(x)
- x = Left-Rotate(x)

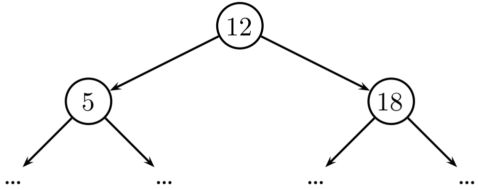
Rotation



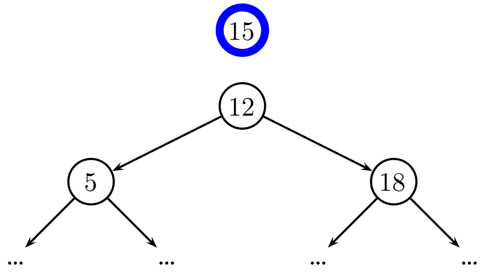
```
Right-Rotate(x)
1   l = x.left
2   x.left = l.right
3   l.right = x
4   return l
```

```
Left-Rotate(x)
1   r = x.right
2   x.right = r.left
3   r.left = x
4   return r
```

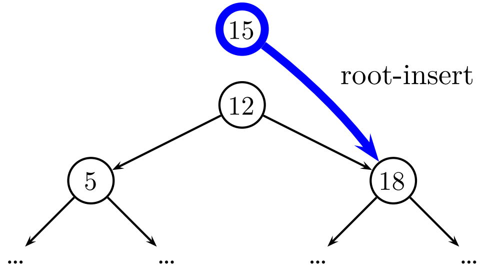
Root Insertion



Root Insertion

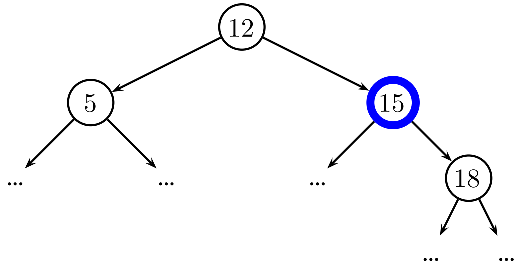


Root Insertion



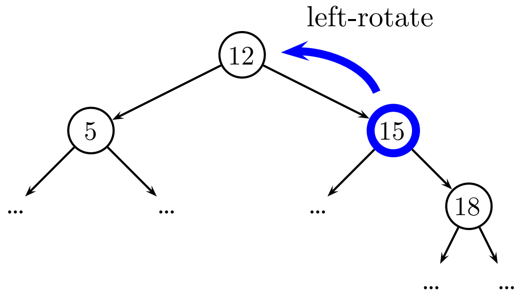
1. Recursively insert z at the root of the appropriate subtree (right)

Root Insertion



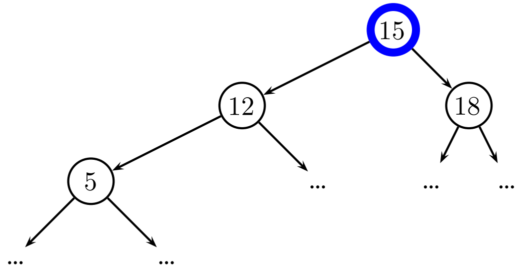
1. Recursively insert z at the root of the appropriate subtree (right)

Root Insertion



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

Root Insertion



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)
```

Observation

- General strategies to deal with complexity in the worst case

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

Observation

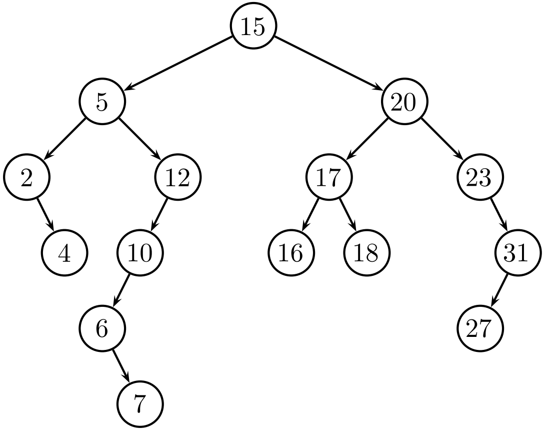
- General strategies to deal with complexity in the worst case
  - randomization: turns any case into the average case
    - the worst case is still possible, but it is extremely improbable
  - amortized maintenance: e.g., balancing a BST or resizing a hash table
    - relatively expensive but “amortized”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

Observation

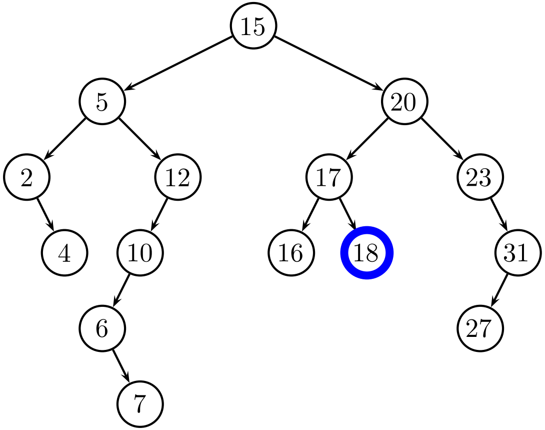
- General strategies to deal with complexity in the worst case
  - randomization: turns any case into the average case
    - the worst case is still possible, but it is extremely improbable
  - amortized maintenance: e.g., balancing a BST or resizing a hash table
    - relatively expensive but "amortized"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

Deletion

Deletion

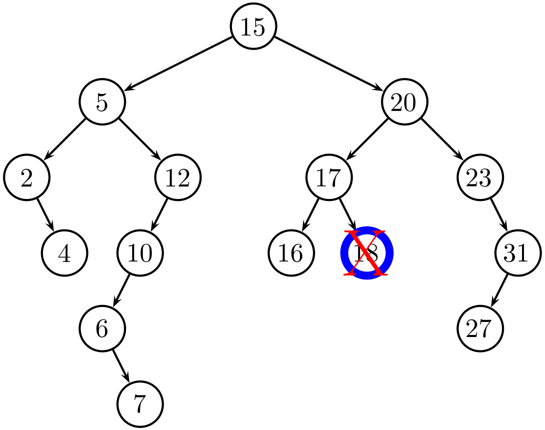


Deletion



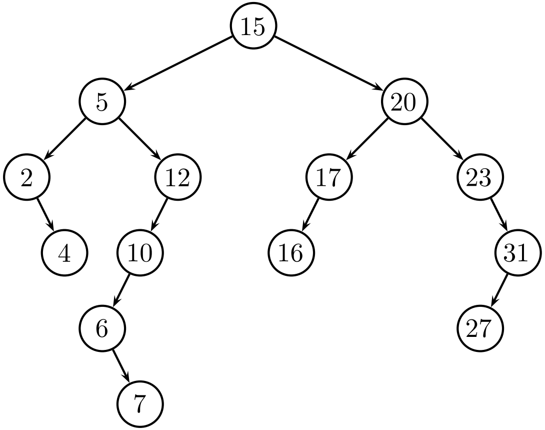
1. z has no children

Deletion



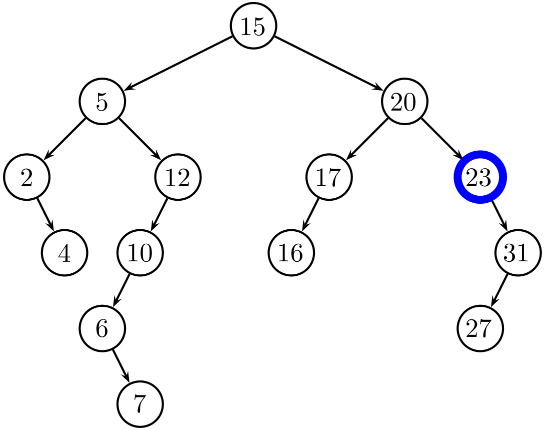
1. z has no children
   - simply remove z

Deletion



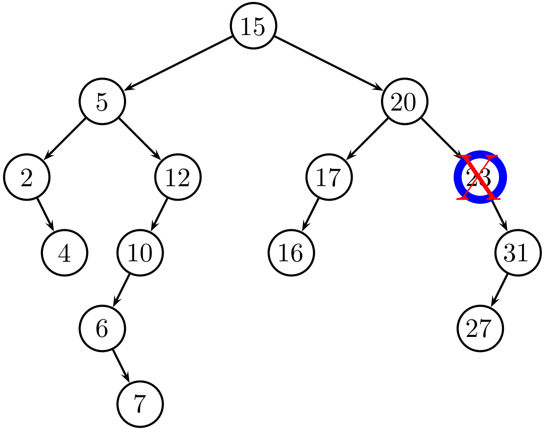
1. z has no children
    - simply remove z

Deletion



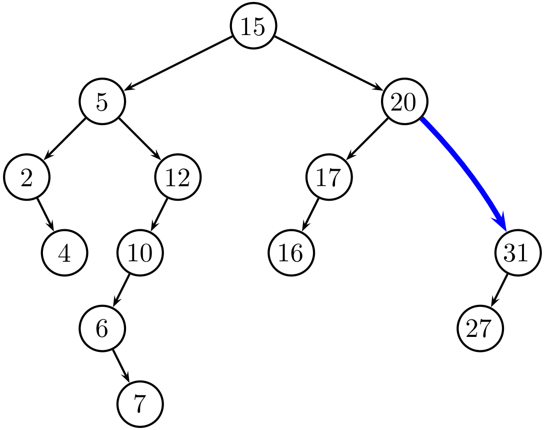
1. z has no children
    - simply remove z
2. z has one child

Deletion



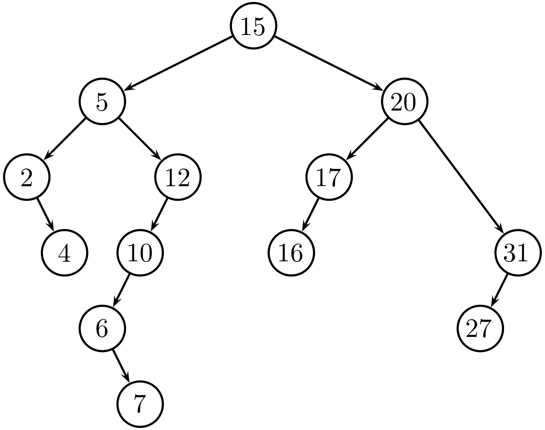
1. z has no children
   - simply remove z
2. z has one child
   - remove z

Deletion



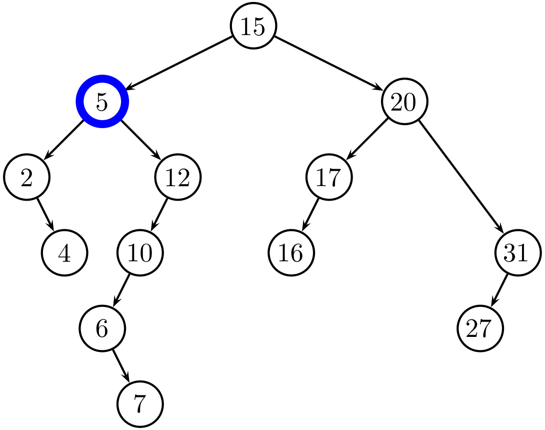
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Deletion



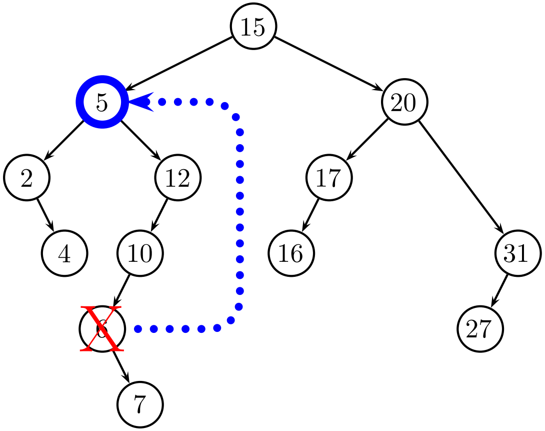
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Deletion



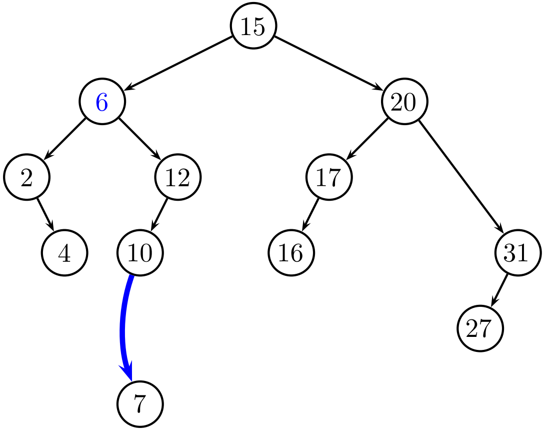
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children

Deletion



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!)

Deletion



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

Deletion (2)

```
Tree-Delete(T, z)
if z.left = nil or z.right = nil
    y = z
else y = Tree-Successor(z)
if y.left $\neq$ nil
    x = y.left
else x = y.right
if x $\neq$ 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 $\neq$ z
    z.key = y.key
    copy any other data from y into z
```