# 4.4 Symbol Tables and BSTs


*"I no longer teach the meaning of life. I now focus on search engine optimization, which, in my opinion, is the meaning of life."*

# Symbol Table

Symbol table. Key-value pair abstraction.

- Insert a key with specified value.
- Given a key, search for the corresponding value.

Ex. [DNS lookup]

- Insert URL with specified IP address.
- Given URL, find corresponding IP address.

| URL | IP address |
|---|---|
| www.cs.princeton.edu | 128.112.136.11 |
| www.princeton.edu | 128.112.128.15 |
| www.yale.edu | 130.132.143.21 |
| www.harvard.edu | 128.103.060.55 |
| www.simpsons.com | 209.052.165.60 |

key value

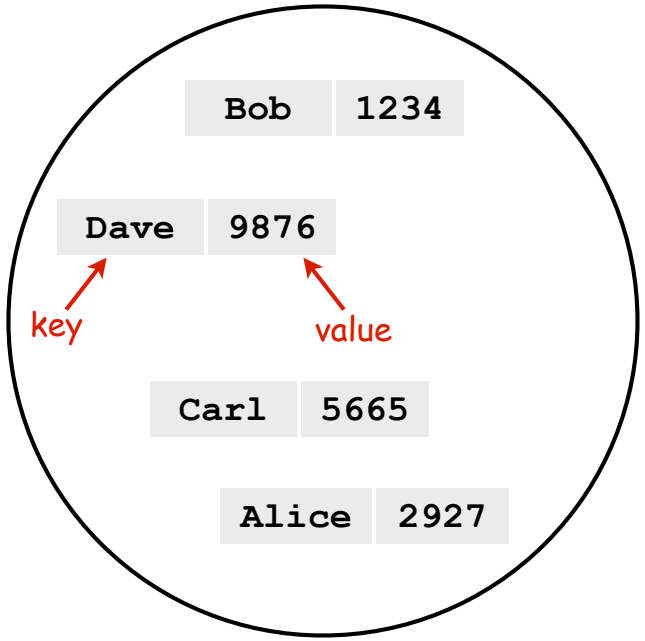
# Symbol Table Applications

| Application | Purpose | Key | Value |
|---|---|---|---|
| phone book | look up phone number | name | phone number |
| bank | process transaction | account number | transaction details |
| file share | find song to download | name of song | computer ID |
| file system | find file on disk | filename | location on disk |
| dictionary | look up word | word | definition |
| web search | find relevant documents | keyword | list of documents |
| book index | find relevant pages | keyword | list of pages |
| web cache | download | filename | file contents |
| genomics | find markers | DNA string | known positions |
| DNS | find IP address given URL | URL | IP address |
| reverse DNS | find URL given IP address | IP address | URL |
| compiler | find properties of variable | variable name | value and type |
| routing table | route Internet packets | destination | best route |

# Symbol Table API

| public class ST<Key extends Comparable<Key>, Value> | |
|---|---|
| ST() | *create a symbol table* |
| void put(Key key, Value v) | *put* key-value *pair into the table* |
| Value get(Key key) | *return value paired with* key, null *if* key *not in table* |
| boolean contains(Key key) | *is there a value paired with* key? |

symbol table is a
set of key-value pairs

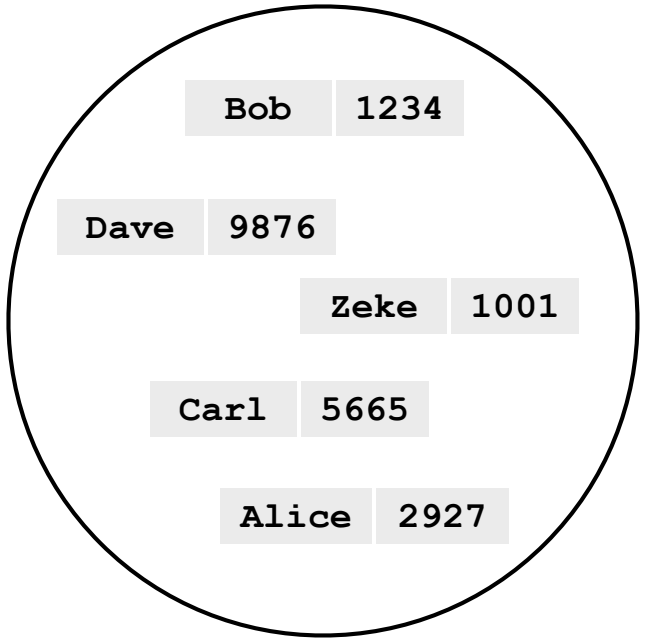


# Symbol Table API

| public class ST<Key extends Comparable<Key>, Value> | |
|---|---|
| ST() | *create a symbol table* |
| void put(Key key, Value v) | *put* key-value *pair into the table* |
| Value get(Key key) | *return value paired with* key, null *if* key *not in table* |
| boolean contains(Key key) | *is there a value paired with* key? |

```
put("Zeke", 1001);
```

adds key-value pair

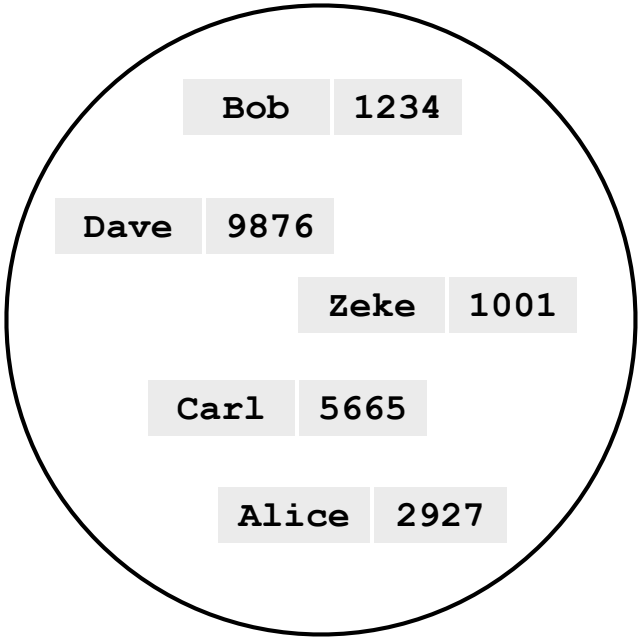


# Symbol Table API

| public class ST<Key extends Comparable<Key>, Value> | |
|---|---|
| ST() | *create a symbol table* |
| void put(Key key, Value v) | *put* key-value *pair into the table* |
| Value get(Key key) | *return value paired with* key, null *if* key *not in table* |
| boolean contains(Key key) | *is there a value paired with* key? |

```
get("Alice");
```

returns 2927



# Symbol Table API

public class *ST<Key extends Comparable<Key>, Value>

| | | |
|---|---|---|
| | *ST() | *create a symbol table* |
| void | put(Key key, Value v) | *put* key-value *pair into the table* |
| Value | get(Key key) | *return value paired with* key, null *if* key *not in table* |
| boolean | contains(Key key) | *is there a value paired with* key? |

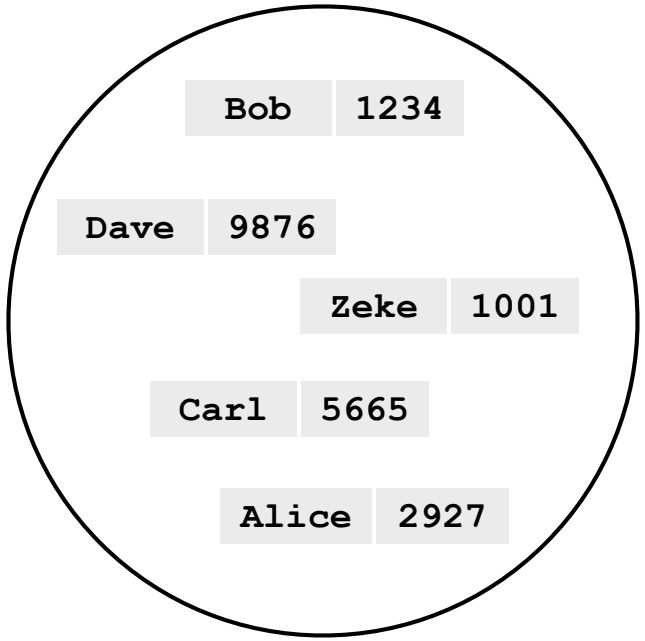
*Note: Implementations should also implement the* Iterable<Key> *interface to enable clients to access keys in sorted order with foreach loops.*

```
contains("Alice");
```

returns true

```
contains("Fred");
```

returns false



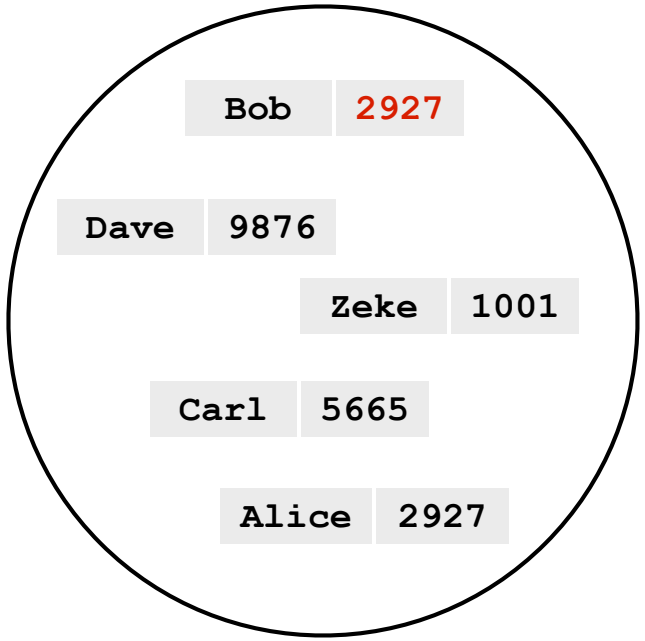
# Symbol Table API

| public class *ST<Key extends Comparable<Key>, Value> | |
|---|---|
| *ST() | *create a symbol table* |
| void put(Key key, Value v) | *put* key-value *pair into the table* |
| Value get(Key key) | *return value paired with* key, null *if* key *not in table* |
| boolean contains(Key key) | *is there a value paired with* key? |

*Note: Implementations should also implement the* Iterable<Key> *interface to enable clients to access keys in sorted order with foreach loops.*

`put("Bob", 2927);`
changes Bob's value

"Associative array" notation
`st["Bob"] = 2927;`
is legal in some languages
(not Java)



# Symbol Table Client Example 1: Index

## Indexing

- Key: string
- Value: Queue of integers
- Read a key from standard input.
- If key is in symbol table, add its position to queue
  If key is not in symbol table, create a queue first

```
% more tiny.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
```

```
public class Index
{
   public static void main(String[] args)
   {
      ST<String, Queue> st = new ST<String, Queue>();

      int i = 0;
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (!st.contains(key)) st.put(key, new Queue());
         st.get(key).enqueue(i++);
      }

      for (String s : st)
         StdOut.println(s + " " + st.get(s));

   }
}
```

key type

value type

enhanced for loop (stay tuned)

```
%  java Index < tiny.txt
age 15 21
belief 29
best 3
darkness 47
despair 59
epoch 27 33
foolishness 23
hope 53
incredulity 35
it 0 6 12 18 24 30 36 42 48 54
light 41
of 4 10 16 22 28 34 40 46 52 58
season 39 45
spring 51
the 2 8 14 20 26 32 38 44 50 56
times 5 11
was 1 7 13 19 25 31 37 43 49 55
winter 57
wisdom 17
worst 9
```

# Symbol Table Client Example 2: Frequency Counter

Frequency counter. [e.g., web traffic analysis, linguistic analysis]

- Key: string
- Value: `Integer` counter
- Read a key from standard input.
- If key is in symbol table, increment counter by 1;
  If key is not in symbol table, insert it with counter = 1.

```
public class Freq
{
   public static void main(String[] args)
   {
      ST<String, Integer> st = new ST<String, Integer>();

      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (st.contains(key)) st.put(key, st.get(key) + 1);
         else                  st.put(key, 1);
      }

      for (String s : st)
         StdOut.println(st.get(s) + " " + s);

   }
}
```

key type

value type

calculate frequencies

enhanced for loop (stay tuned)

print results

```
$ java Freq < tiny.txt
2 age
1 belief
1 best
1 darkness
1 despair
2 epoch
1 foolishness
1 hope
1 incredulity
10 it
1 light
10 of
2 season
1 spring
10 the
2 times
10 was
1 winter
1 wisdom
1 worst
```

# Sample datasets

Linguistic analysis. Compute word frequencies in a piece of text.

| File | Description | Words | Distinct |
|---|---|---|---|
| mobydick.txt | Melville's Moby Dick | 210,028 | 16,834 |
| leipzig100k.txt | 100K random sentences | 2,121,054 | 144,256 |
| leipzig200k.txt | 200K random sentences | 4,238,435 | 215,515 |
| leipzig1m.txt | 1M random sentences | 21,191,455 | 534,580 |

Reference: Wortschatz corpus, Univesität Leipzig
http://corpora.informatik.uni-leipzig.de

# Zipf's Law

Linguistic analysis. Compute word frequencies in a piece of text.

```
% java Freq < mobydick.txt
4583 a
2 aback
2 abaft
3 abandon
7 abandoned
1 abandonedly
2 abandonment
2 abased
1 abasement
2 abashed
1 abate
…
```

```
% java Freq < mobydick.txt | sort -rn
13967 the
6415 of
6247 and
4583 a
4508 to
4037 in
2911 that
2481 his
2370 it
1940 i
1793 but
…
```

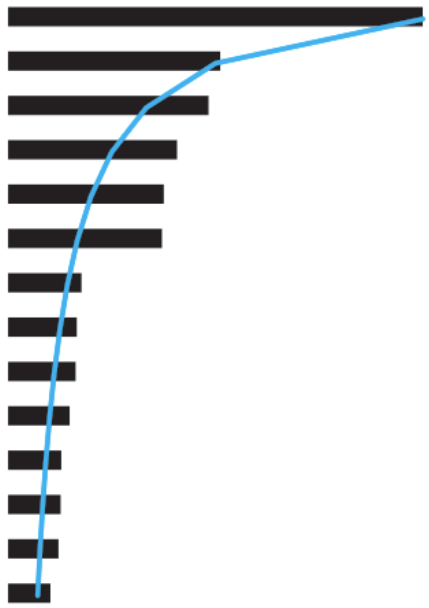
e.g., most frequent word occurs about twice as often as second most frequent one

Zipf's law. Frequency of $i^{th}$ most common word is inversely proportional to $i$.

# Zipf's Law

Linguistic analysis. Compute word frequencies in a piece of text.

```
% java Freq < leipzig1m.txt | sort -rn
1160105 the
593492 of
560945 to
472819 a
435866 and
430484 in
205531 for
192296 The
188971 that
172225 is
148915 said
147024 on
141178 was
118429 by
…
```



Zipf's law. Frequency of $i^{th}$ most common word is inversely proportional to $i$.

Challenge: Develop symbol-table implementation for such experiments.

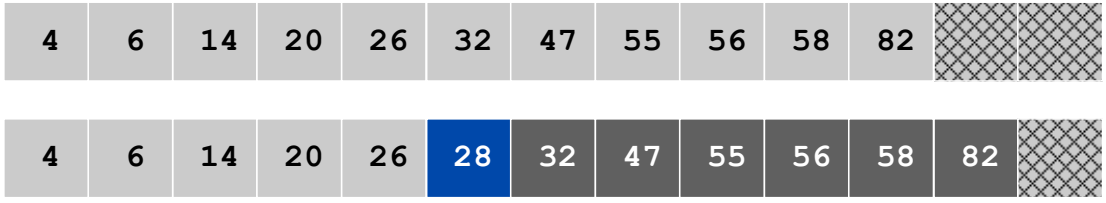
# Symbol Table: Elementary Implementations

## Unordered array.

- Put: add key to the end (if not already there).
- Get: scan through all keys to find desired value.


## Ordered array.

- Put: find insertion point, and shift all larger keys right.
- Get: binary search to find desired key.




*insert 28*

# Symbol Table: Implementations Cost Summary

Unordered array. Hopelessly slow for large inputs.

Ordered array. Acceptable if many more searches than inserts;
too slow if large number of inserts.

| | Running Time | | Frequency Count | | | |
|---|---|---|---|---|---|---|
| implementation | get | put | Moby | 100K | 200K | 1M |
| unordered array | $N$ | $N$ | 170 sec | 4.1 hr | - | - |
| ordered array | $\log N$ | $N$ | 5.8 sec | 5.8 min | 15 min | 2.1 hr |

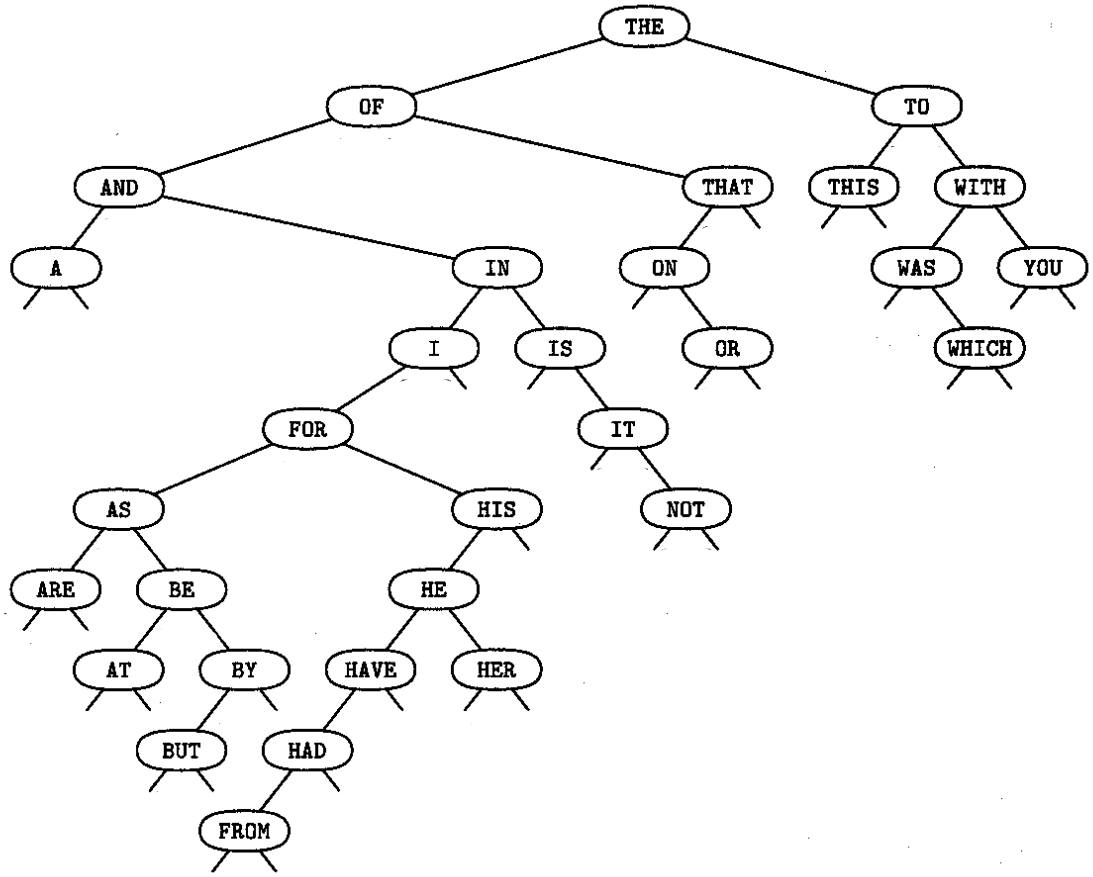
too slow: ~$N^2$ to build entire table

doubling test: quadratic

Challenge. Make all ops logarithmic.

Note: Linked lists are not much help (have to traverse list)

# Binary Search Trees




Reference: Knuth, The Art of Computer Programming

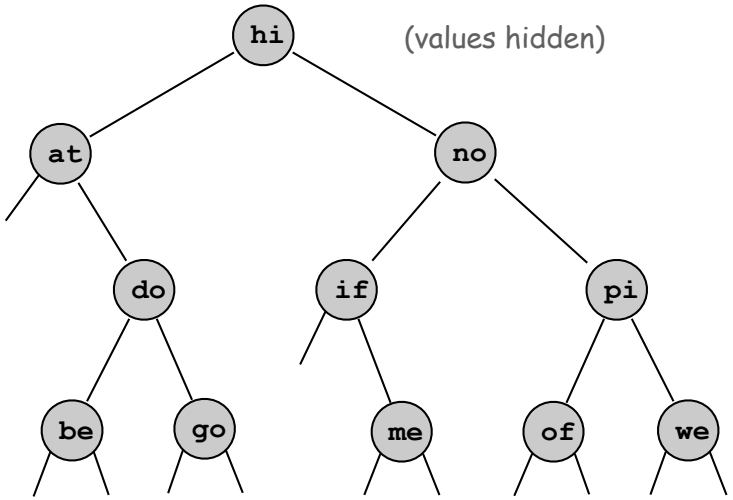
# Binary Search Trees

Def. A binary search tree is a binary tree, with keys in symmetric order.

Binary tree is either:

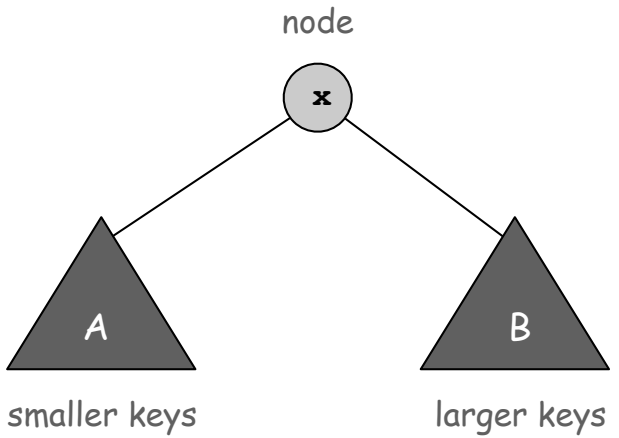
- Empty.
- A key-value pair and two binary trees.

we suppress values from figures

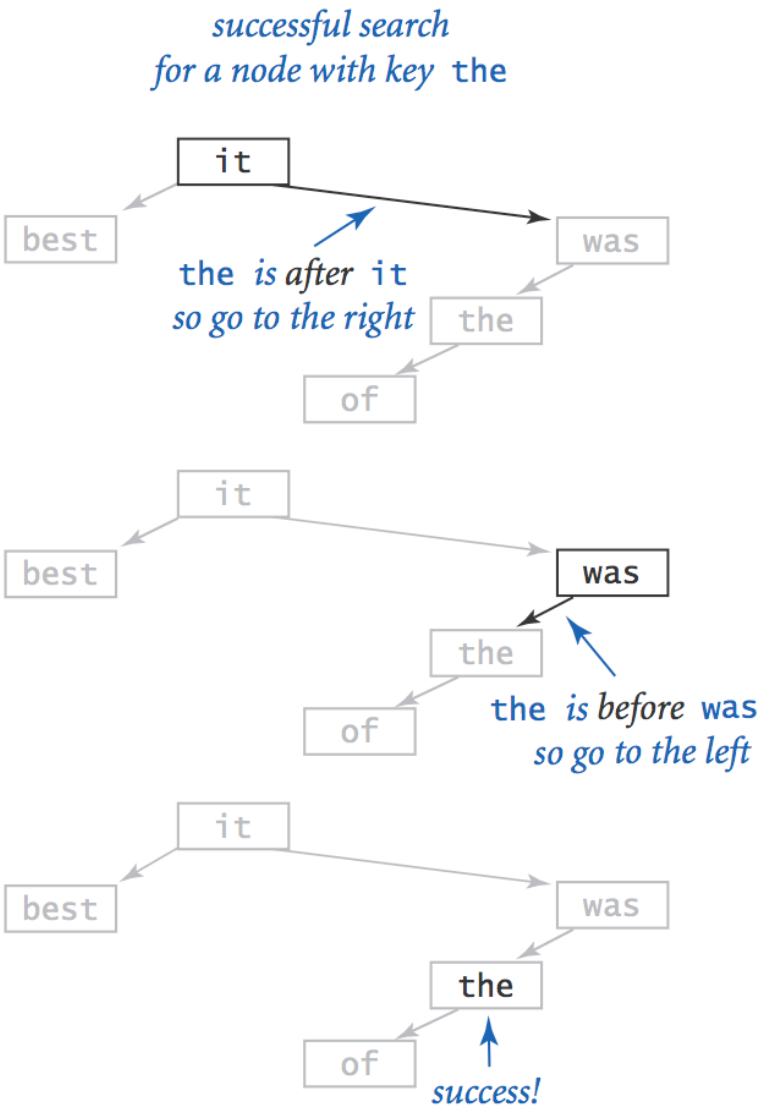


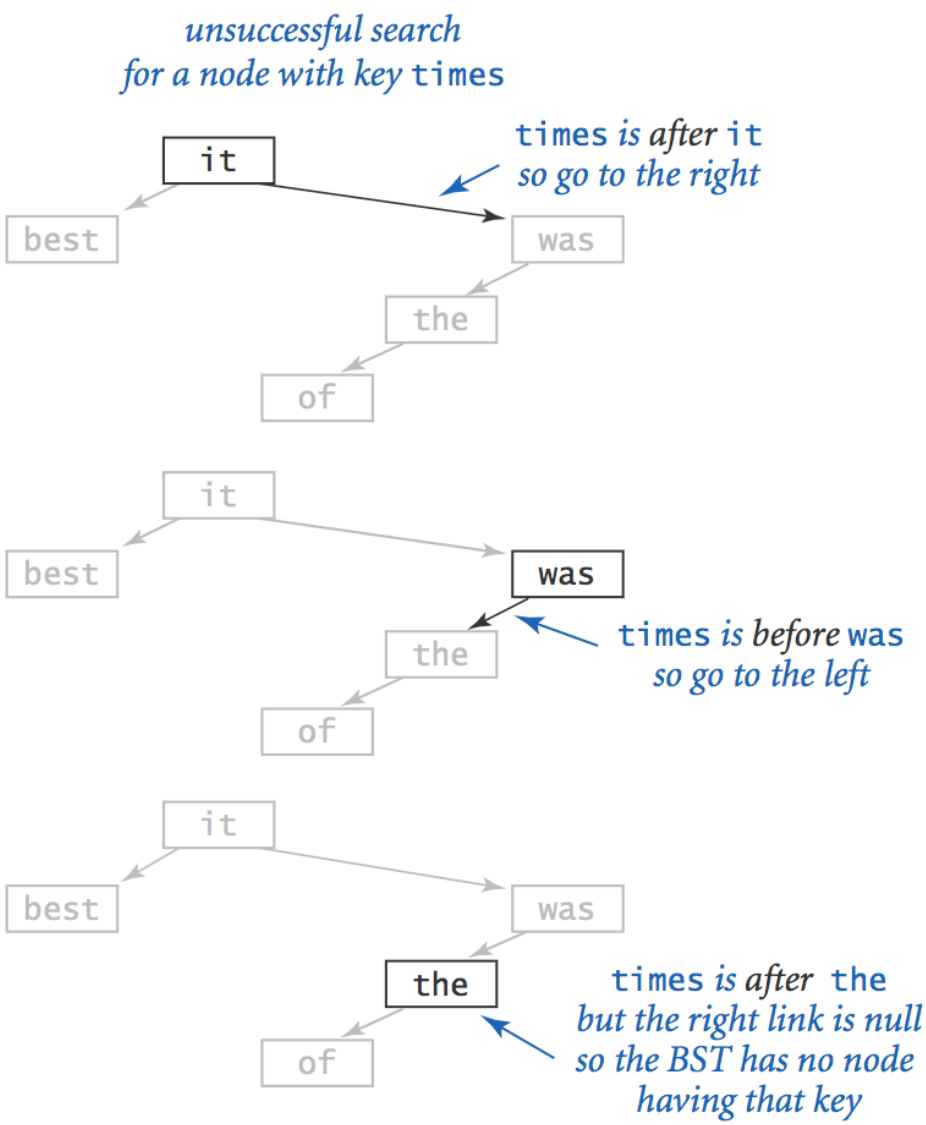

Symmetric order.

- Keys in left subtree are smaller than parent.
- Keys in right subtree are larger than parent.

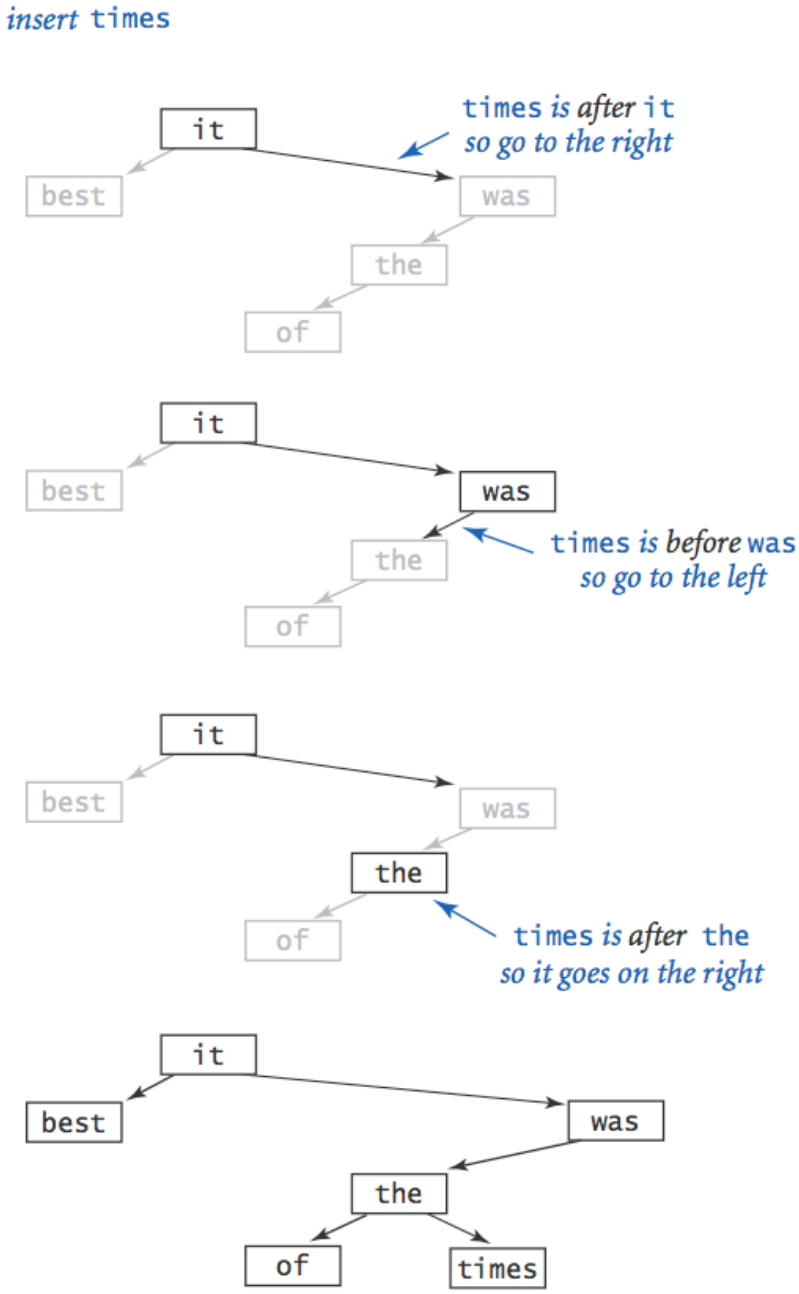


# BST Search

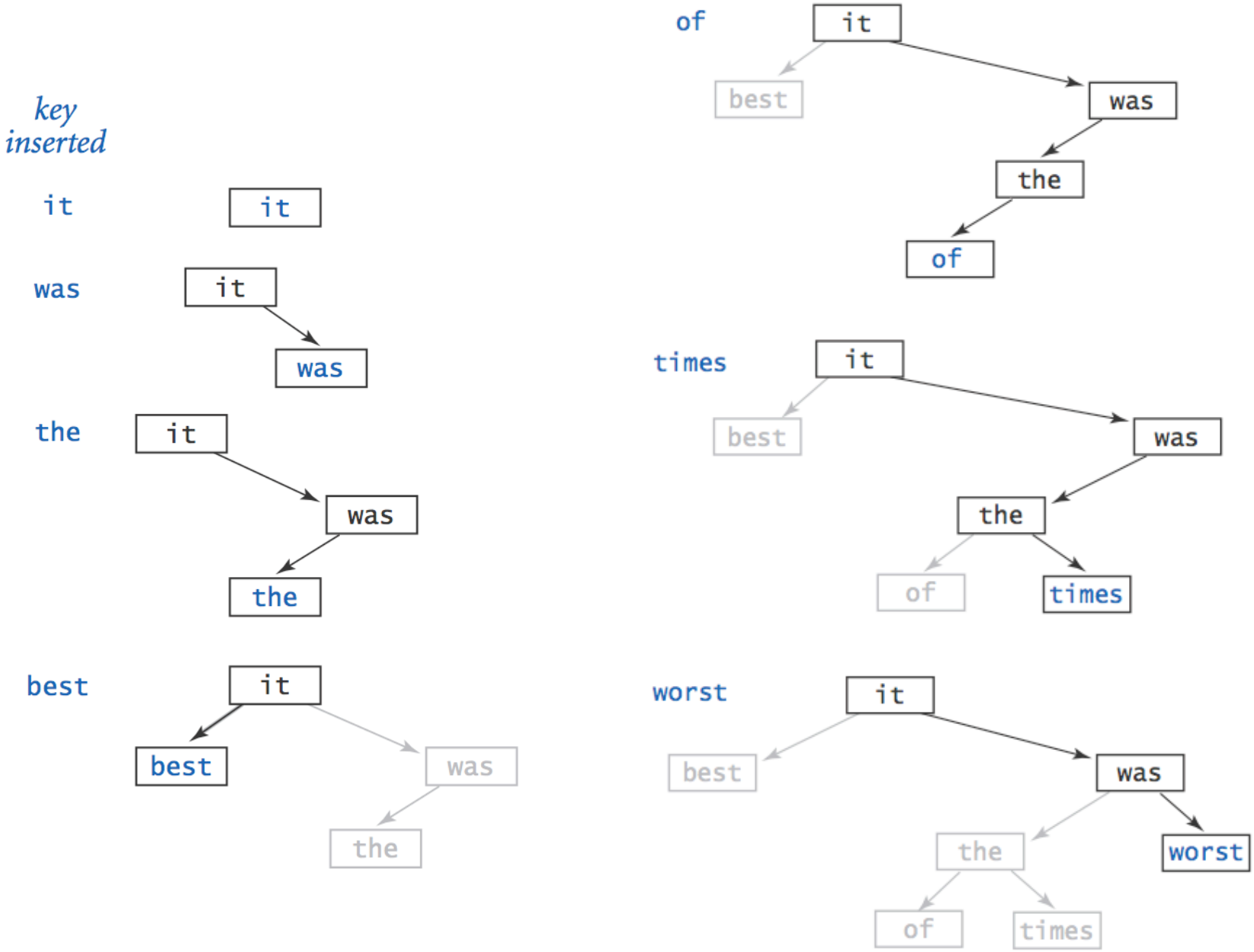

successful search
for a node with key the
it
best
was
the
of
the is after it
so go to the right
the is before was
so go to the left
success!


unsuccessful search
for a node with key times
it
best
was
the
of
times is after it
so go to the right
times is before was
so go to the left
times is after the
but the right link is null
so the BST has no node
having that key

# BST Insert


insert times
it
best
was
the
of
times is after it
so go to the right
it
best
was
the
of
times is before was
so go to the left
it
best
was
the
of
times is after the
so it goes on the right
it
best
was
the
of
times

# BST Construction



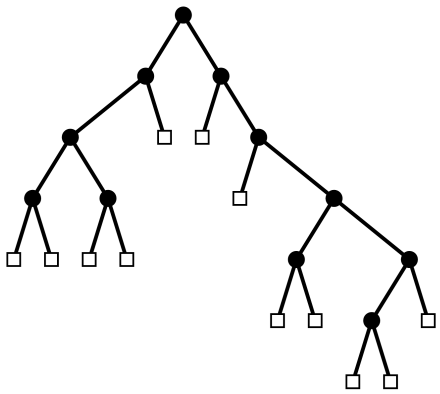
# Binary Search Tree: Java Implementation

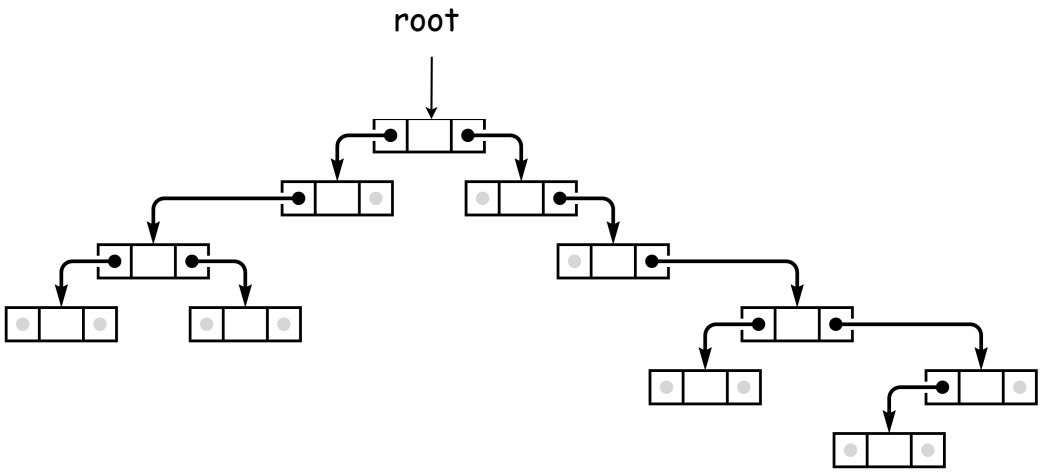
To implement: use two links per `Node`.

A `Node` is comprised of:

- A key.
- A value.
- A reference to the left subtree.
- A reference to the right subtree.

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
}
```





# BST: Skeleton

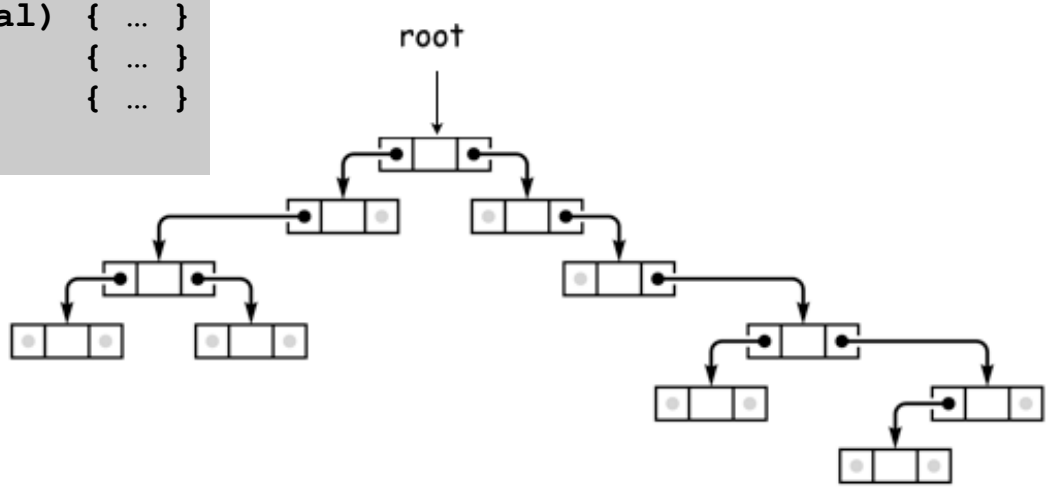
BST. (with generic keys and values).

```
public class BST<Key, Value>
{
   private Node root;   // root of the BST

   private class Node
   {
      private Key key;
      private Value val;
      private Node left, right;

      private Node(Key key, Value val)
      {
         this.key = key;
         this.val = val;
      }
   }

   public void put(Key key, Value val)  { … }
   public Value get(Key key)            { … }
   public boolean contains(Key key)     { … }
}
```



# BST: Get

Get. Return `val` corresponding to given `key`, or `null` if no such key.

```
public Value get(Key key)
{
   return get(root, key);
}

private Value get(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0)  return get(x.left,  key);
   else if (cmp > 0)  return get(x.right, key);
   else if (cmp == 0) return x.val;
}

public boolean contains(Key key)
{
   return (get(key) != null);
}
```

# BST: Put

Put. Associate `val` with `key`.

- Search, then insert.
- Concise (but tricky) recursive code.

```
public void put(Key key, Value val)
{
   root = put(root, key, val);
}

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key, val);
   else              x.val = val;
   return x;
}
```

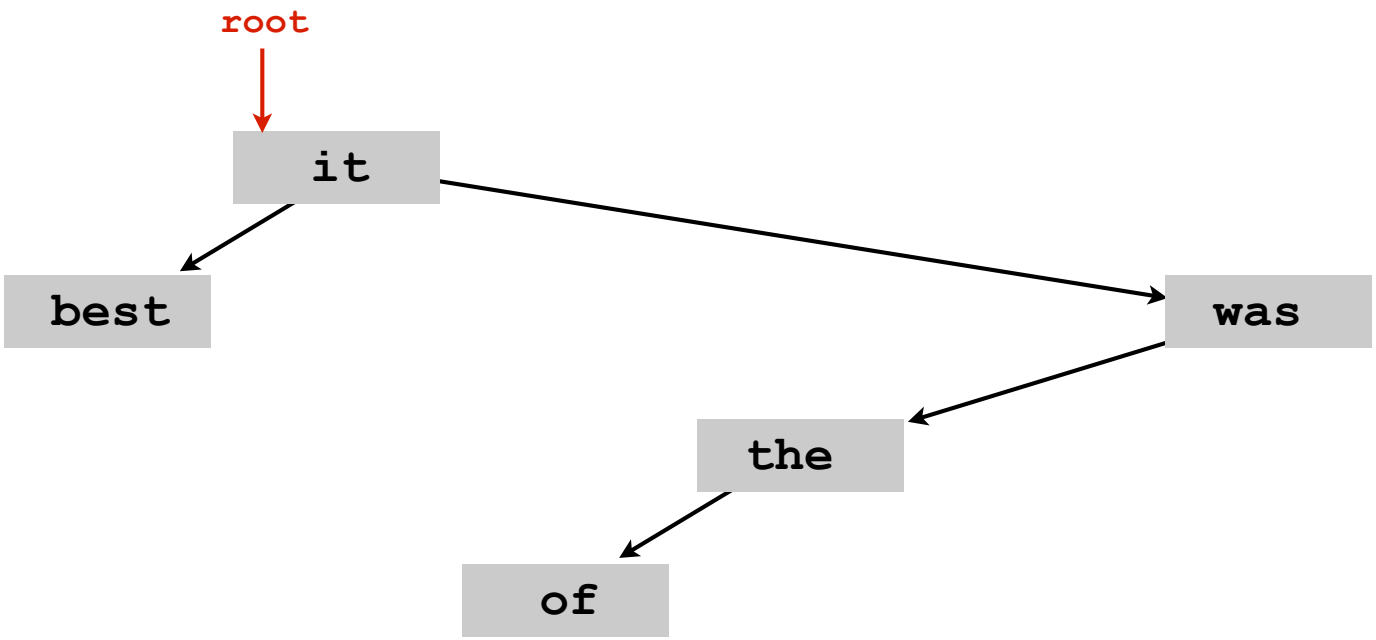
overwrite old value with new

# Inserting a new node in a BST

```
public void put(Key key, Value val)
{
   root = put(root, key, val);
}
```

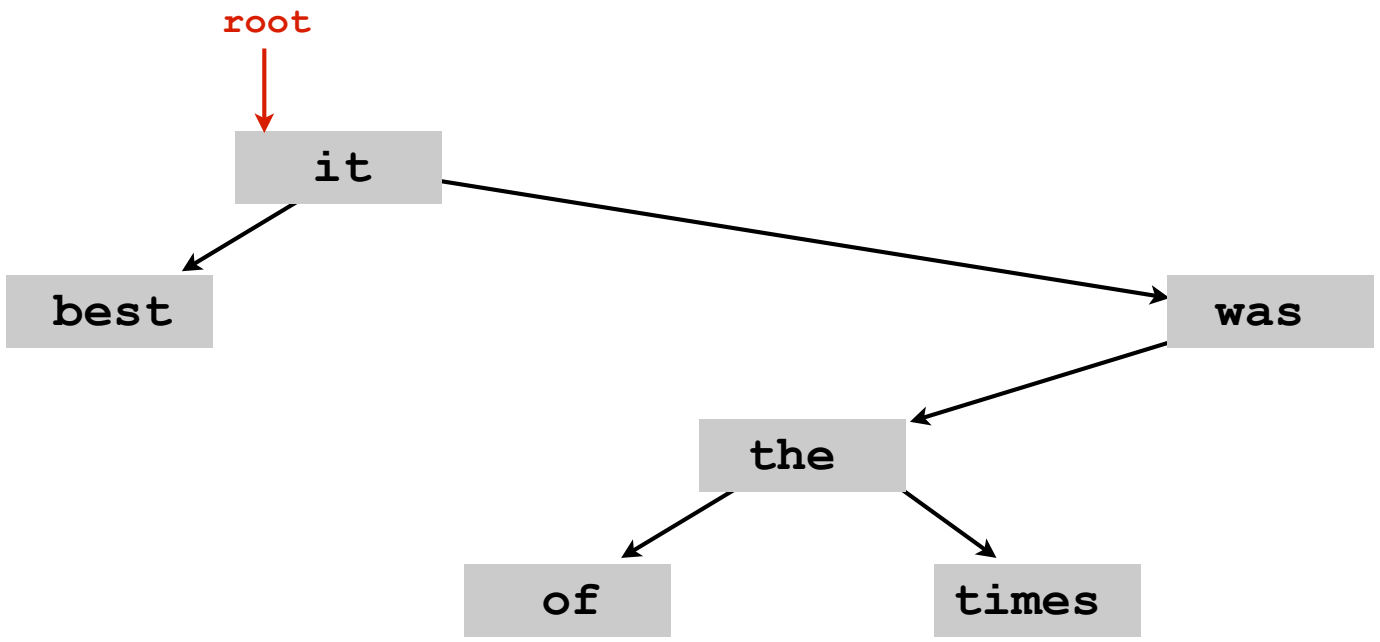
key

times



# Inserting a new node in a BST

```
public void put(Key key, Value val)
{
   root = put(root, key, val);
}
```



# BST Implementation: Practice

Bottom line. Difference between a practical solution and no solution.

| | Running Time | | Frequency Count | | | |
|---|---|---|---|---|---|---|
| implementation | get | put | Moby | 100K | 200K | 1M |
| unordered array | $N$ | $N$ | 170 sec | 4.1 hr | - | - |
| ordered array | $\log N$ | $N$ | 5.8 sec | 5.8 min | 15 min | 2.1 hr |
| BST | ? | ? | .95 sec | 7.1 sec | 14 sec | 69 sec |

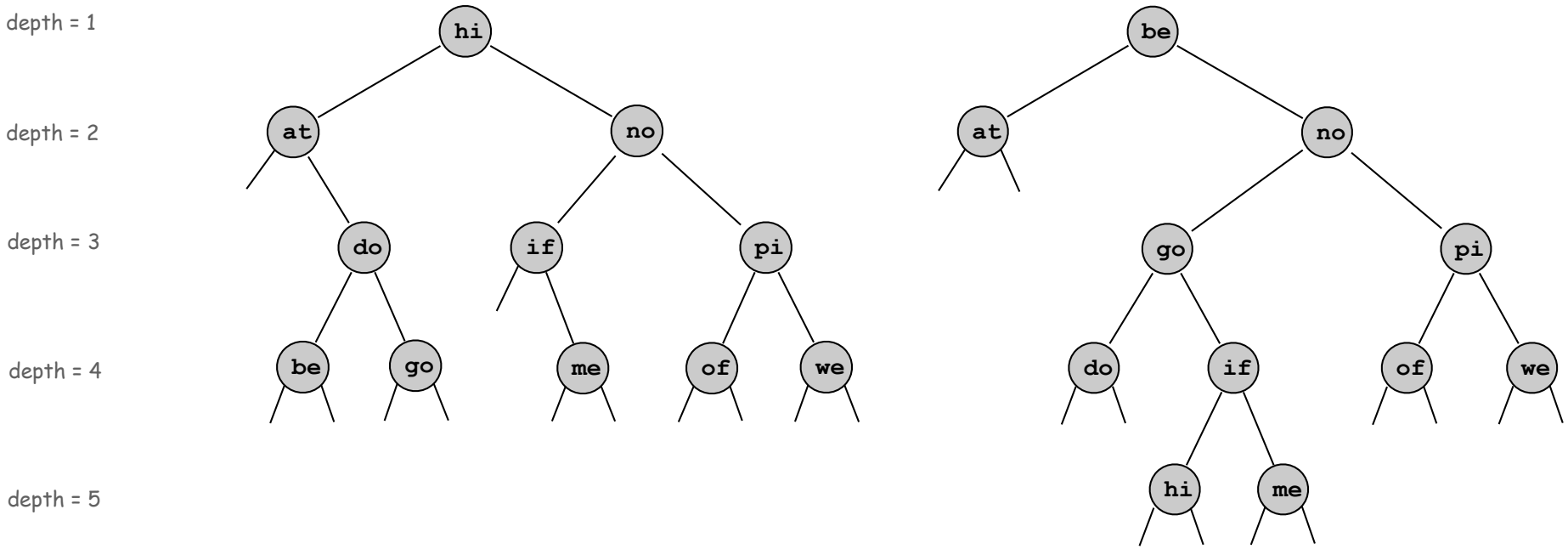
doubling test: scalable

# BST: Analysis

Running time per put/get.

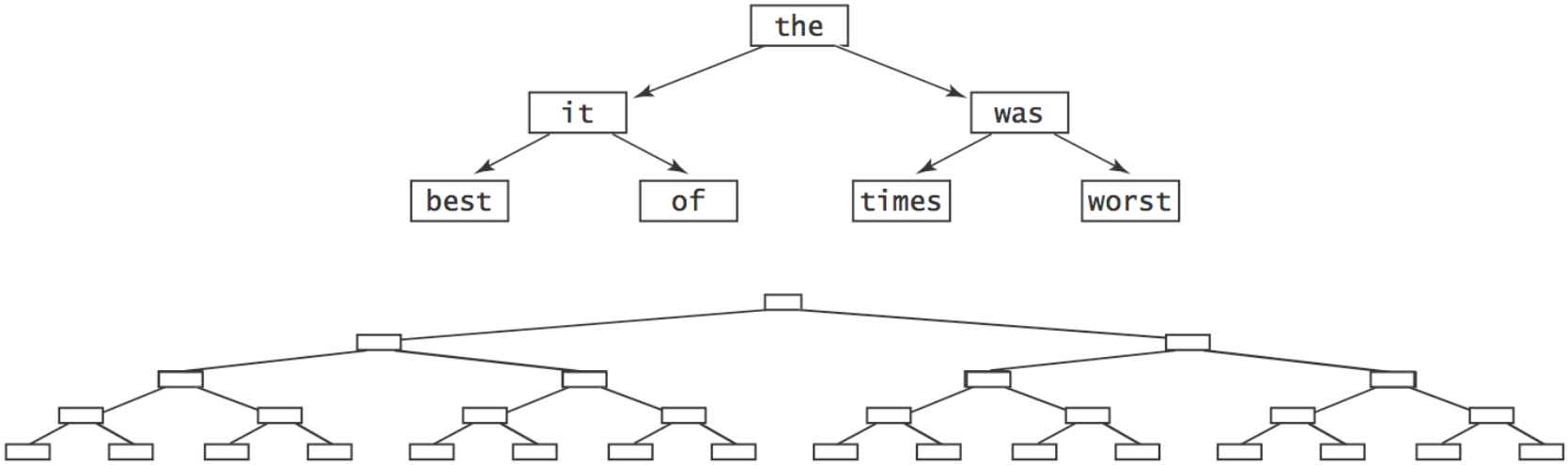
- There are many BSTs that correspond to same set of keys.
- Cost is proportional to depth of node.

number of nodes on path from root to node



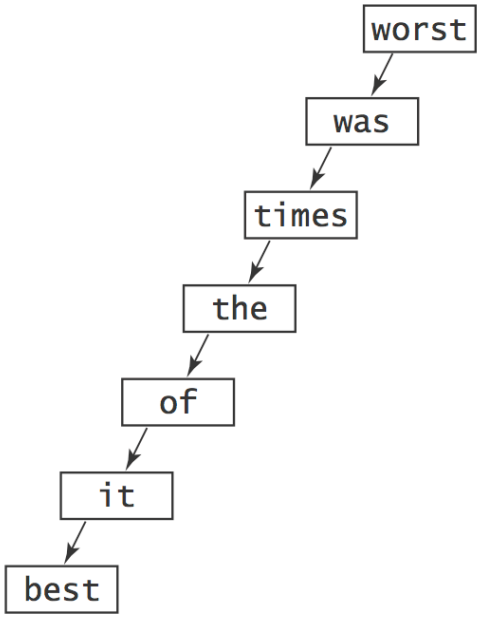
# BST: Analysis

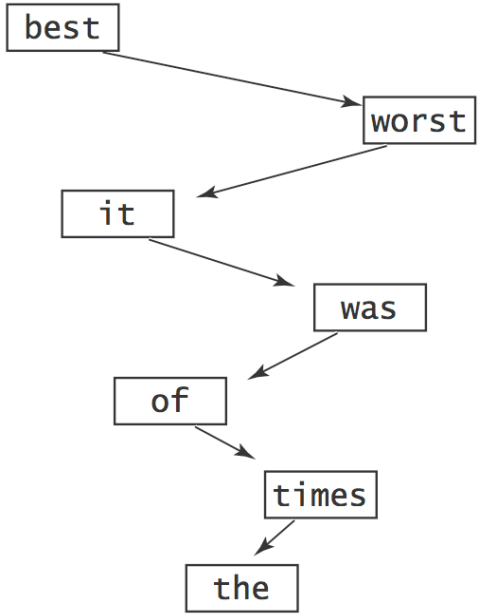
Best case. If tree is perfectly balanced, depth is at most $\lg N$.

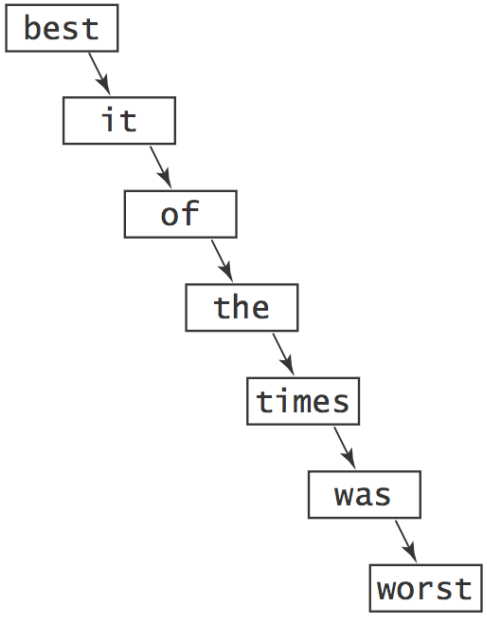


# BST: Analysis

Worst case. If tree is unbalanced, depth is $N$.



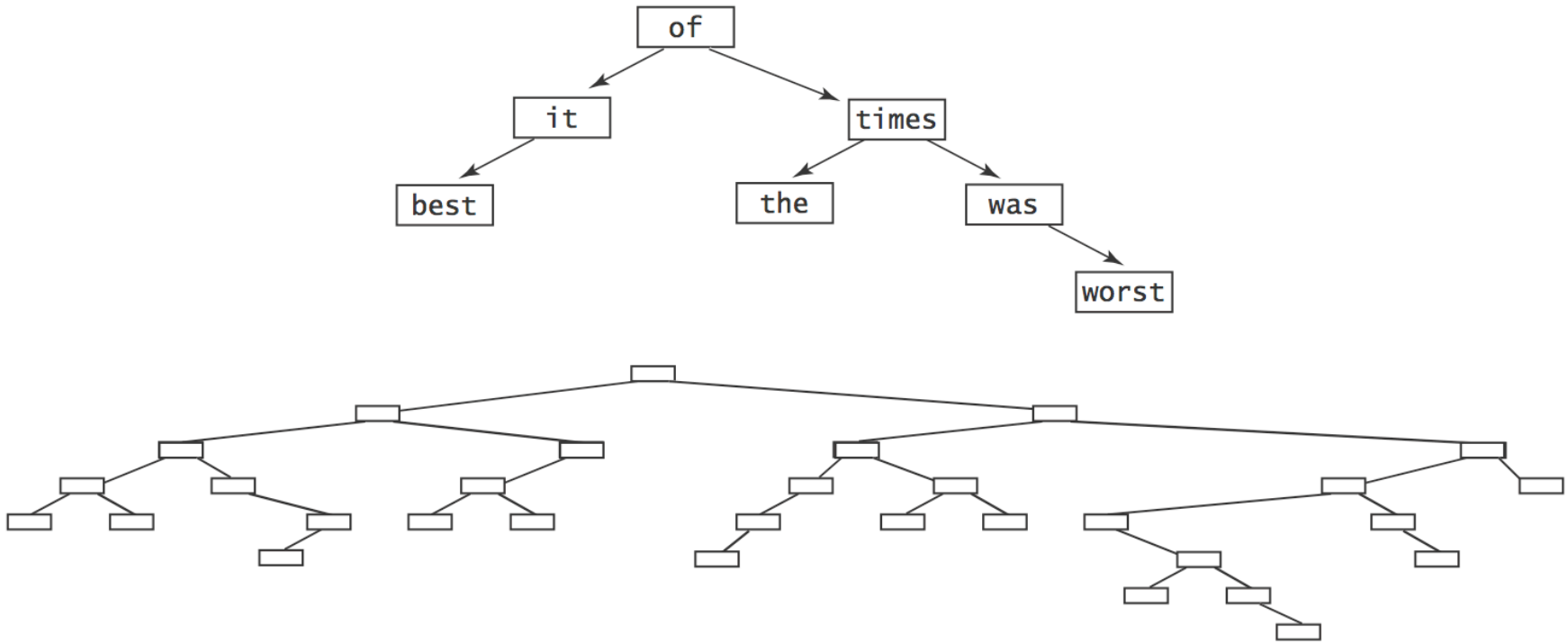



# BST: Analysis

Average case. If keys are inserted in random order, average depth is $2 \ln N$.

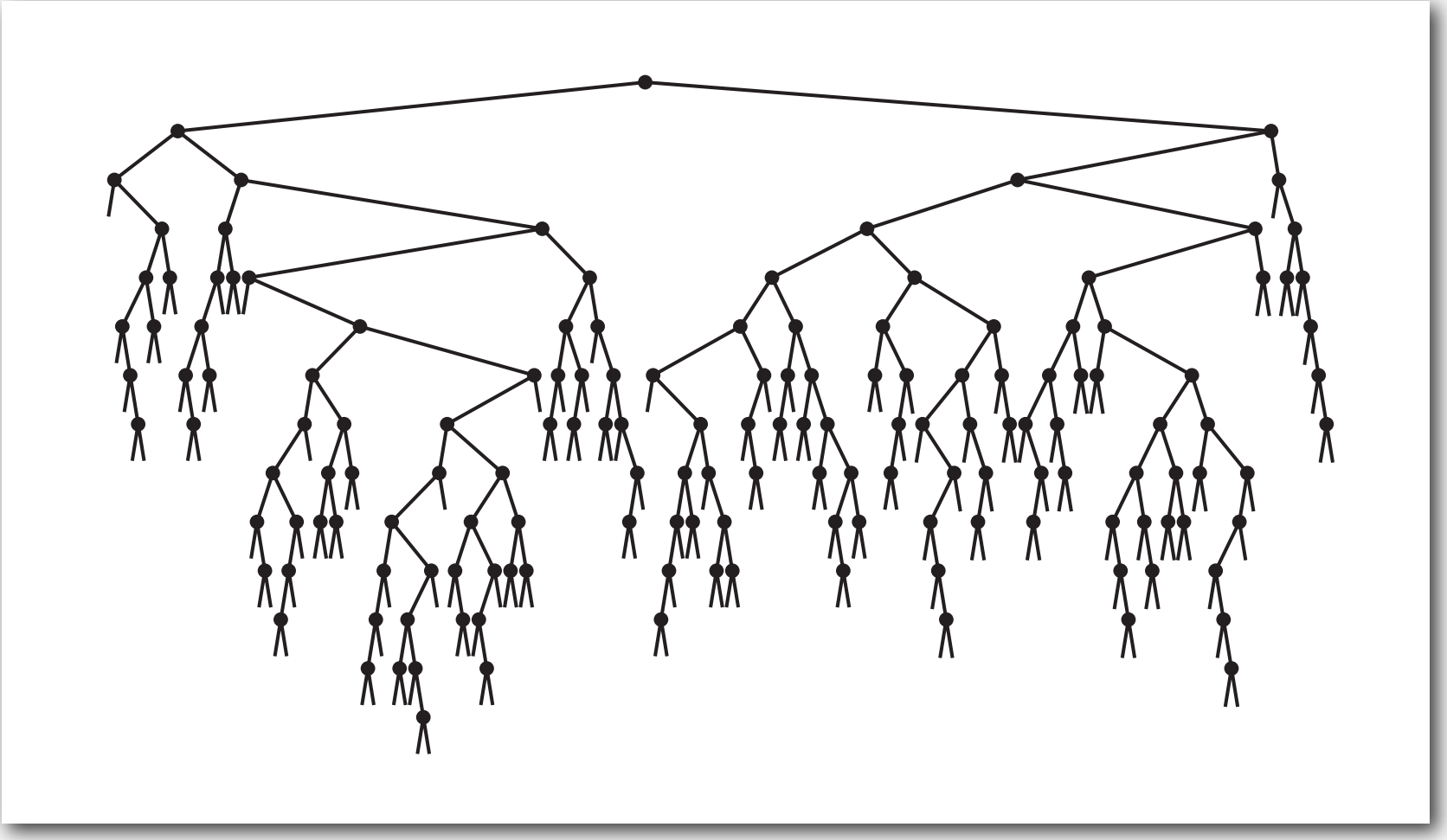
requires proof
(see COS 226)




*Typical BSTs constructed from randomly ordered keys*

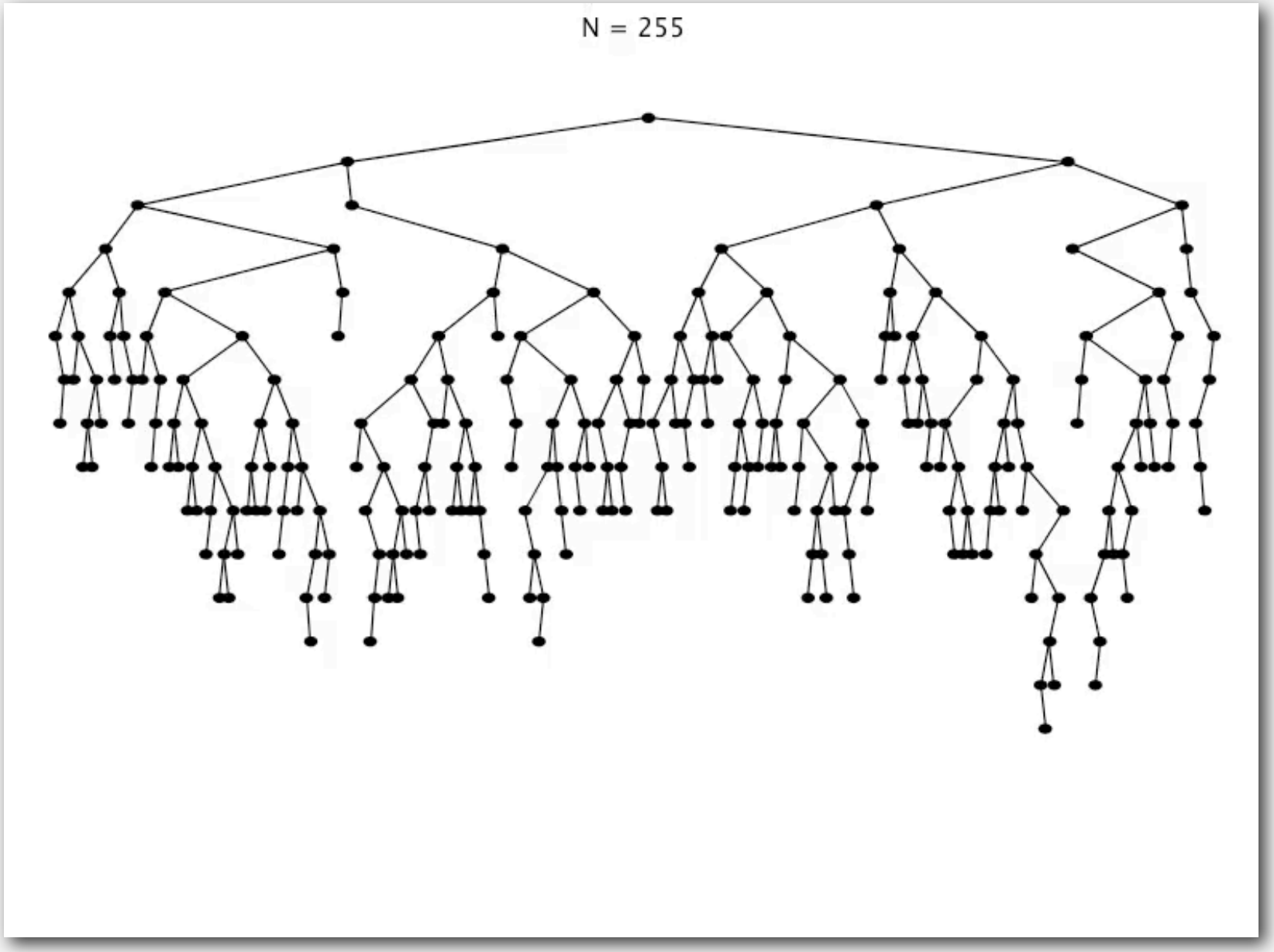
# BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.



# BST insertion: random order visualization

Ex. Insert keys in random order.



# Symbol Table: Implementations Cost Summary

BST. Logarithmic time ops if keys inserted in random order.

| | Running Time | | Frequency Count | | | |
|---|---|---|---|---|---|---|
| implementation | get | put | Moby | 100K | 200K | 1M |
| unordered array | $N$ | $N$ | 170 sec | 4.1 hr | - | - |
| ordered array | $\log N$ | $N$ | 5.8 sec | 5.8 min | 15 min | 2.1 hr |
| BST | $\log N$ † | $\log N$ † | .95 sec | 7.1 sec | 14 sec | 69 sec |

† assumes keys inserted in random order

Q. Can we guarantee logarithmic performance?

# Red-Black Tree

Red-black tree. A clever BST variant that guarantees depth $\leq 2 \lg N$.

see COS 226

Java red-black tree library implementation

```
import java.util.TreeMap;
import java.util.Iterator;

public class ST<Key>, Value> implements Iterable<Key>
{

   private TreeMap<Key, Value> st = new TreeMap<Key, Val>();

   public void put(Key key, Value val)
   {
      if (val == null) st.remove(key);
      else             st.put(key, val);
   }
   public Value get(Key key)         { return st.get(key);            }
   public Value remove(Key key)      { return st.remove(key);         }
   public boolean contains(Key key)  { return st.containsKey(key);    }
   public Iterator<Key> iterator()   { return st.keySet().iterator(); }
}
```

# Red-Black Tree

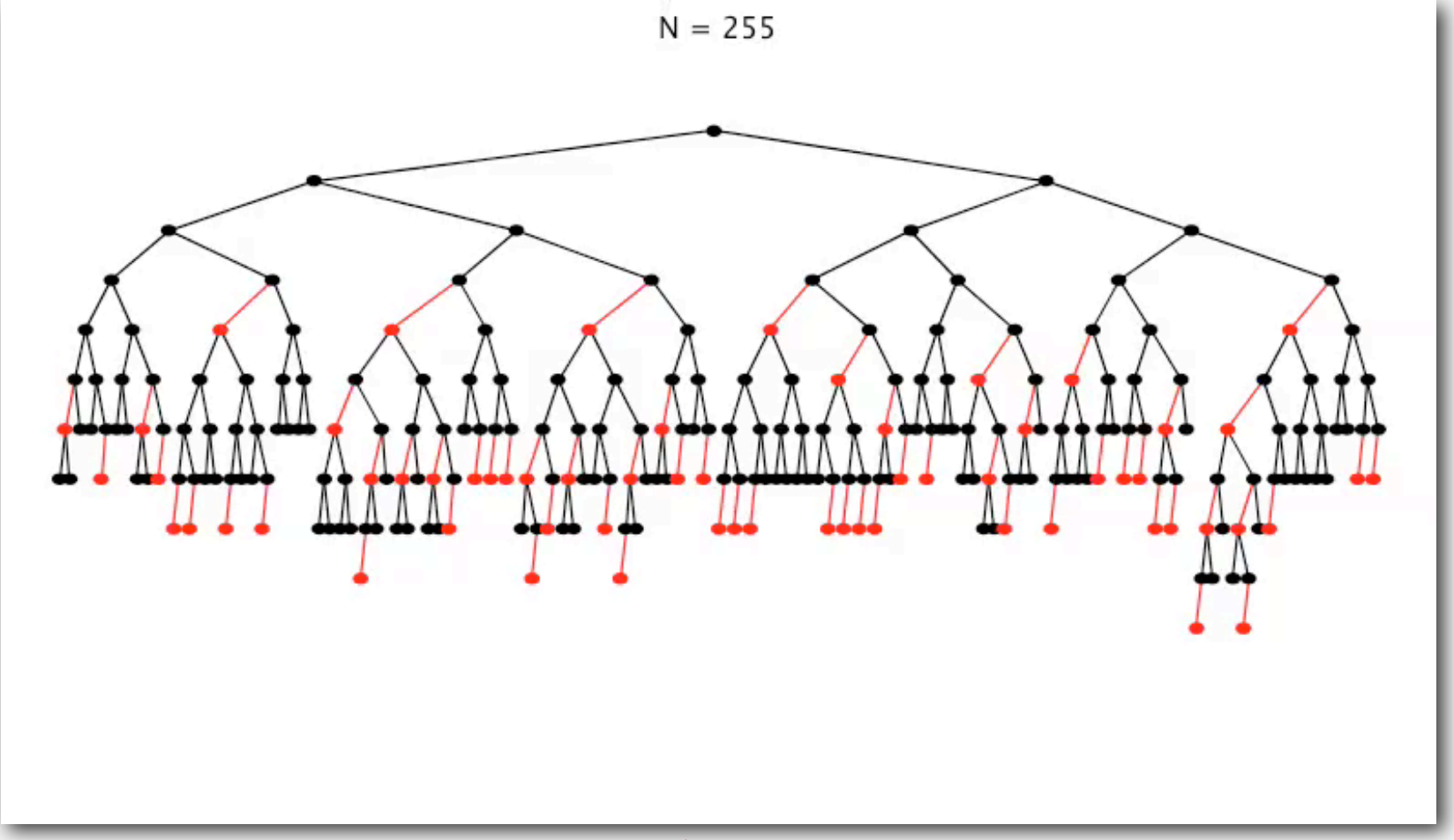
Red-black tree. A clever BST variant that guarantees depth $\leq 2 \lg N$.

see COS 226

| | Running Time | | Frequency Count | | | |
|---|---|---|---|---|---|---|
| implementation | get | put | Moby | 100K | 200K | 1M |
| unordered array | $N$ | $N$ | 170 sec | 4.1 hr | - | - |
| ordered array | $\log N$ | $N$ | 5.8 sec | 5.8 min | 15 min | 2.1 hr |
| BST | $\log N$ † | $\log N$ † | .95 sec | 7.1 sec | 14 sec | 69 sec |
| red-black | $\log N$ | $\log N$ | .95 sec | 7.0 sec | 14 sec | 74 sec |

† assumes keys inserted in random order

# Insertion in a LLRB tree: visualization



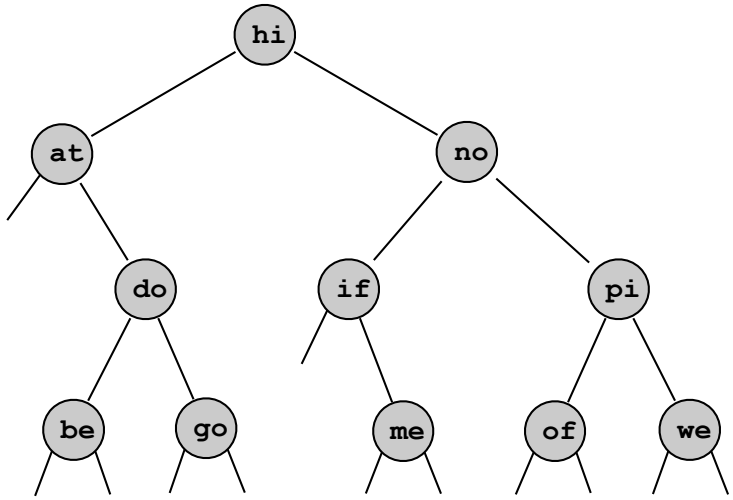

255 random insertions

# Iteration

# Inorder Traversal

Inorder traversal.

- Recursively visit left subtree.
- Visit node.
- Recursively visit right subtree.




```
inorder: at be do go hi if me no of pi we
```

```
public inorder()
{  inorder(root);  }

private void inorder(Node x)
{
   if (x == null) return;
   inorder(x.left);
   StdOut.println(x.key);
   inorder(x.right);
}
```

# Enhanced For Loop

Enhanced for loop. Enable client to iterate over items in a collection.

```
ST<String, Integer> st = new ST<String, Integer>();
…

for (String s : st)
   StdOut.println(st.get(s) + " " + s);
```

# Enhanced For Loop with BST

BST. Add following code to support enhanced for loop (uses a stack).

see COS 226 for details

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class BST<Key extends Comparable<Key>, Value> implements Iterable<Key>
{
   private Node root;

   private class Node { … }

   public void put(Key key, Value val) { … }
   public Value get(Key key)           { … }
   public boolean contains(Key key)    { … }

   public Iterator<Key> iterator() { return new Inorder(); }
   private class Inorder implements Iterator<Key>
   {
       Inorder() { pushLeft(root); }
       public boolean hasNext() { return !stack.isEmpty();                    }
       public Key next()
       {
          if (!hasNext()) throw new NoSuchElementException();
          Node x = stack.pop();
          pushLeft(x.right);
          return x.key;
       }
       public void pushLeft(Node x)
       {
          while (x != null) {
             stack.push(x);
             x = x.left;
          }
       }
   }
}
```

# Symbol Table: Summary

Symbol table. Quintessential database lookup data type.

Choices. Ordered array, unordered array, BST, red-black, hash, ….

- Different performance characteristics.
- Fast search, insert, and ordered iteration is available.
- Java libraries: `TreeMap`, `HashMap`.

Remark. Better symbol table implementation improves all clients.