4.4 Symbol Tables

INTRODUCTION TO

Programming
in Java

An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne

Symbol Table

Symbol table. Key-value pair abstraction.
- Insert (or Put) a key with specified value.
- Given a key, search for (or Get) 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>
```
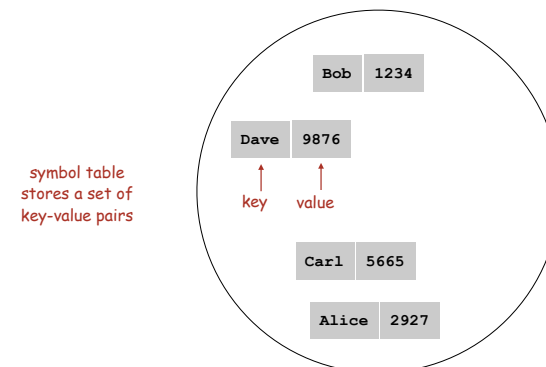
*ST()	*create a symbol table*
void put(Key key, Value v)	*put key-value pair into the table*
Value get(Key key)	*return value paired with key, null if key not in table*
boolean contains(Key key)	*is there a value paired with key?*

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

symbol table
stores a set of
key-value pairs

Bob   1234

Dave 9876

key   value

Carl 5665

Alice 2927

Symbol Table API

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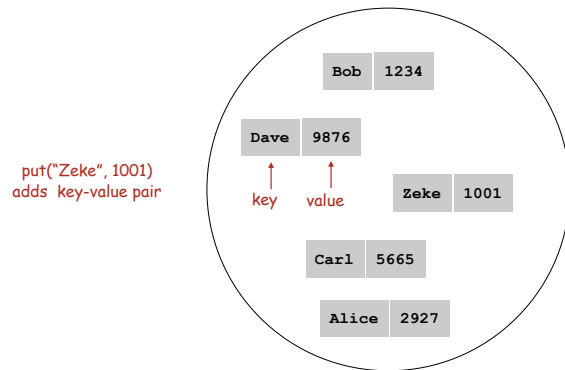
*ST()	*create a symbol table*
void put(Key key, Value v)	*put key-value pair into the table*
Value get(Key key)	*return value paired with key, null if key not in table*
boolean contains(Key key)	*is there a value paired with key?*

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

put("Zeke", 1001)
adds key-value pair

Bob 1234
Dave 9876
key value
Zeke 1001
Carl 5665
Alice 2927

5

Symbol Table API

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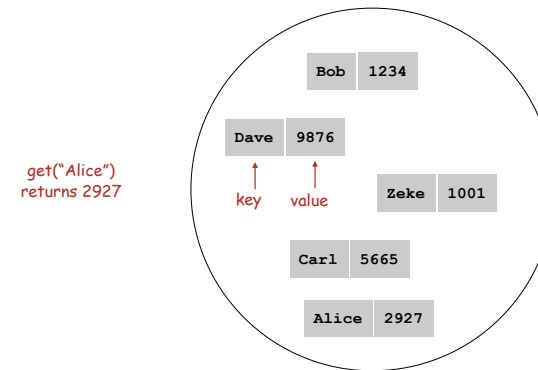
*ST()	*create a symbol table*
void put(Key key, Value v)	*put key-value pair into the table*
Value get(Key key)	*return value paired with key, null if key not in table*
boolean contains(Key key)	*is there a value paired with key?*

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

get("Alice")
returns 2927

Bob 1234
Dave 9876
key value
Zeke 1001
Carl 5665
Alice 2927

6

Symbol Table API

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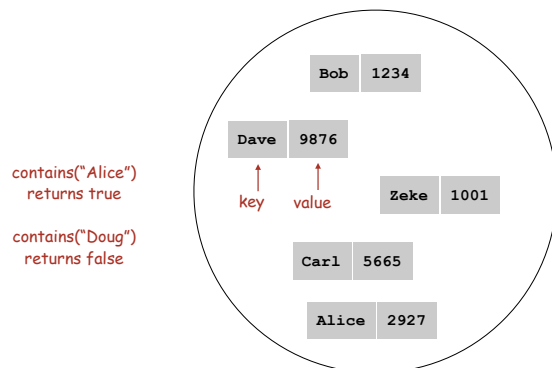
*ST()	*create a symbol table*
void put(Key key, Value v)	*put key-value pair into the table*
Value get(Key key)	*return value paired with key, null if key not in table*
boolean contains(Key key)	*is there a value paired with key?*

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("Doug")
returns false

Bob 1234
Dave 9876
key value
Zeke 1001
Carl 5665
Alice 2927

7

Symbol Table API

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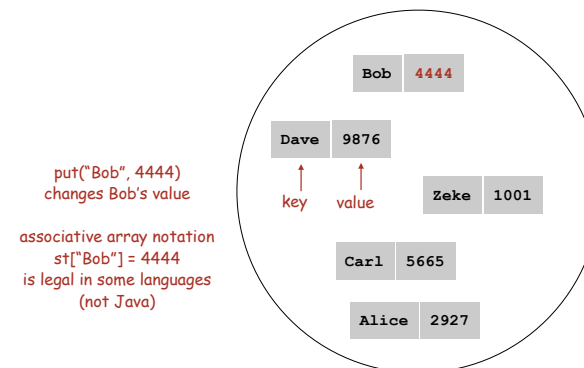
*ST()	*create a symbol table*
void put(Key key, Value v)	*put key-value pair into the table*
Value get(Key key)	*return value paired with key, null if key not in table*
boolean contains(Key key)	*is there a value paired with key?*

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

put("Bob", 4444)
changes Bob's value

associative array notation
st["Bob"] = 4444
is legal in some languages
(not Java)

Bob 4444
Dave 9876
key value
Zeke 1001
Carl 5665
Alice 2927

8

Symbol Table Sample Client

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

    st.put("www.cs.princeton.edu", "128.112.136.11");
    st.put("www.princeton.edu",    "128.112.128.15");
    st.put("www.yale.edu",         "130.132.143.21");

        st["www.yale.com"] = "209.052.165.60"
    StdOut.println(st.get("www.cs.princeton.edu"));
    StdOut.println(st.get("www.harvardsucks.org"));
    StdOut.println(st.get("www.yale.edu"));
}
                         st["www.yale.edu"]
```

```
128.112.136.11
null
130.132.143.21
```

Symbol Table Client: Frequency Counter

Frequency counter. [e.g., web traffic analysis, linguistic analysis]
- Read in a key.
- If key is in symbol table, increment count by one;
- If key is not in symbol table, insert it with count = 1.

```java
public class Freq {
    public static void main(String[] args) {          key type    value type
        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);
        }
                                                     calculate frequencies
            foreach loop (stay tuned)
        for (String s : st)
            StdOut.println(st.get(s) + " " + s);
                                                     print results
    }
}
```

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
…
```

Zipf's law. In natural language, frequency of ith most common word is inversely proportional to i.
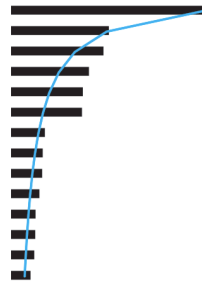
e.g., most frequent word occurs about twice
as often as second most frequent one

Zipf's Law

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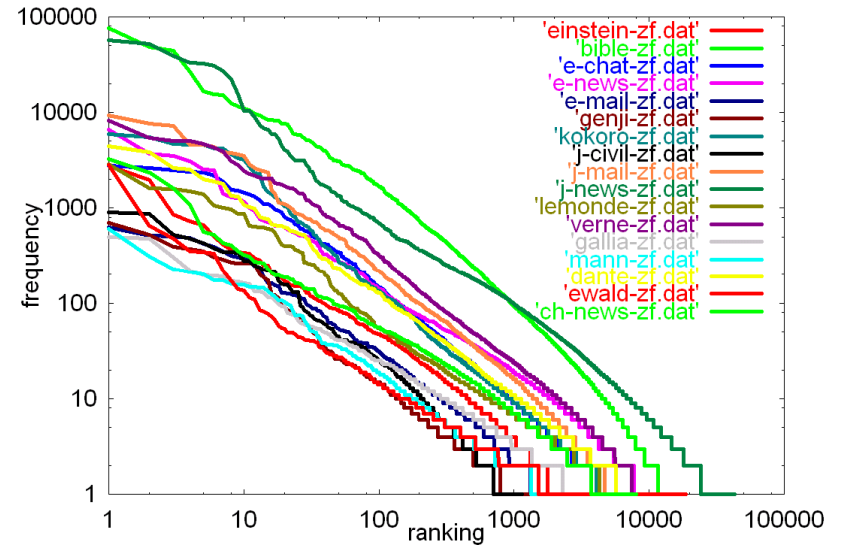
```
% java Freq < leipzig1m.txt | sort -rn
1160105 the
593492 of
560945 to
472819 a
435866 and
430484 in
205531 for
192296 The
188971 that
172225 is
148915 said
147024 on
141178 was
118429 by
…
```

Zipf's law. In natural language, frequency of ith most common word is inversely proportional to i.

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

Zipf's Law

'einstein-zf.dat'
'bible-zf.dat'
'e-chat-zf.dat'
'e-news-zf.dat'
'e-mail-zf.dat'
'genji-zf.dat'
'kokoro-zf.dat'
'j-civil-zf.dat'
'j-mail-zf.dat'
'j-news-zf.dat'
'lemonde-zf.dat'
'verne-zf.dat'
'gallia-zf.dat'
'mann-zf.dat'
'dante-zf.dat'
'ewald-zf.dat'
'ch-news-zf.dat'

Credit: Kumiko Tanaka-Ishii, University of Tokyo

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.

| 32 | 26 | 47 | 82 | 4 | 20 | 58 | 56 | 14 | 6 | 55 | | |

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

| 4 | 6 | 14 | 20 | 26 | 32 | 47 | 55 | 56 | 58 | 82 | | |

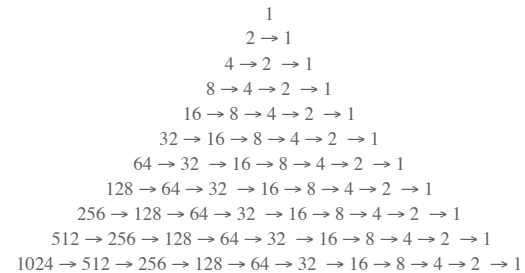| 4 | 6 | 14 | 20 | 26 | 28 | 32 | 47 | 55 | 56 | 58 | 82 | |

insert 28

Binary Search: Mathematical Analysis

Analysis. To binary search in an array of size N: do one compare, then binary search in an array of size $N / 2$.

$$N \to N/2 \to N/4 \to N/8 \to \dots \to 1$$

Q. How many times can you divide a number by 2 until you reach 1?
A. $\log_2 N$.

$$1$$
$$2 \to 1$$
$$4 \to 2 \to 1$$
$$8 \to 4 \to 2 \to 1$$
$$16 \to 8 \to 4 \to 2 \to 1$$
$$32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$512 \to 256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$1024 \to 512 \to 256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$

Symbol Table: Implementations Cost Summary

Unordered array. Hopelessly slow for large inputs.

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

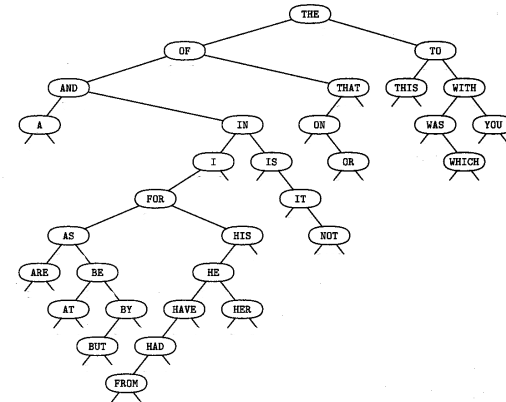| implementation	get	put	Moby	100K	200K	1M
			Running Time	Frequency Count		
unordered array	N	N	170 sec	4.1 hr	-	-
ordered array	$\log N$	N	5.8 sec	5.8 min	15 min	2.1 hr

too slow (N^2 to build table)

doubling test
(quadratic in # of distinct words)

Challenge. Make all ops logarithmic.

18

Binary Search Trees

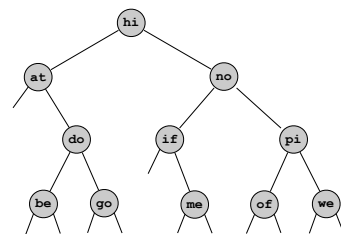Reference: Knuth, The Art of Computer Programming

19

Binary Search Trees

Def. A binary search tree is a binary tree in symmetric order.
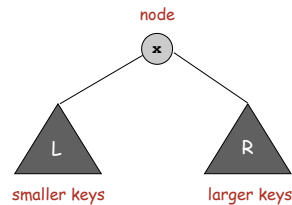
Binary tree is either:
- Empty.
- A key-value pair and two binary trees.

we suppress values from figures
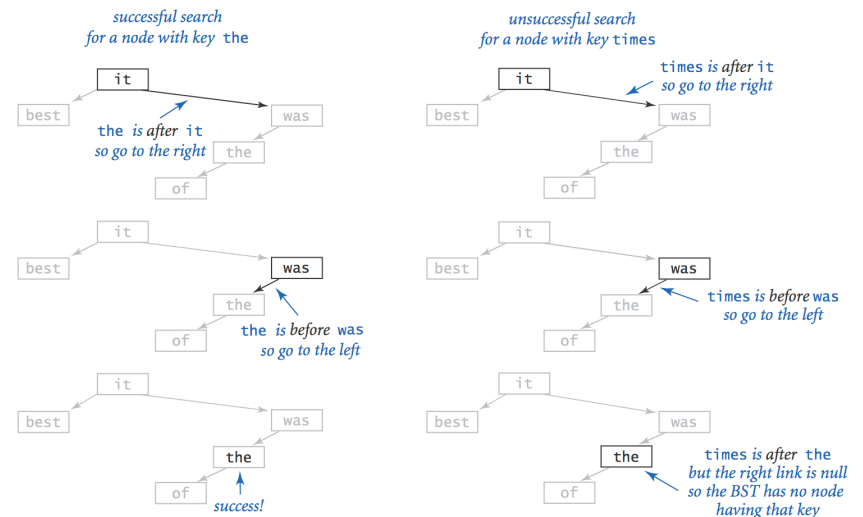

Symmetric order.
- Keys in left subtree are smaller than parent.
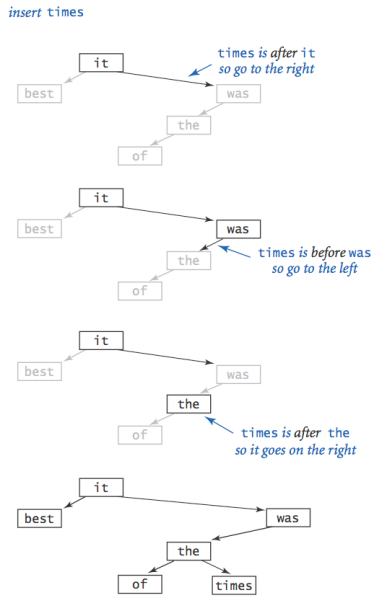- Keys in right subtree are larger than parent.

node

smaller keys   larger keys

20

BST Search (or Get)

*successful search
for a node with key* the

*unsuccessful search
for a node with key* times

the *is after* it
so go to the right

the *is before* was
so go to the left

success!

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

21

BST Insert (or Put)

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

*key
inserted*

of

it
best   was
the
of

it

it
was

it   it
was

the   it
was
the

times   it
best   was
the
of   times

best   it
best   was
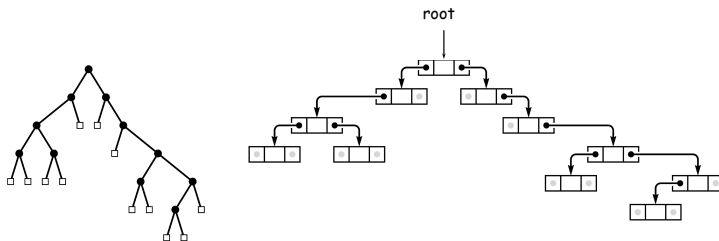the

worst   it
best   was
the   worst
of   times

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;
}
```

root

BST: Skeleton

BST. Allow generic keys and values.

requires `Key` to provide `compareTo()` method;
see textbook for details

```
public class BST<Key extends Comparable<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 (or Search)

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

```java
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              return x.val;  //found key!
}

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

negative if less,
zero if equal,
positive if greater

BST: Put (or Insert)

Put. Associate `val` with `key`.
- Search, then insert.
- Concise (but tricky) recursive code.

```java
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 value

BST Implementation: Practice

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

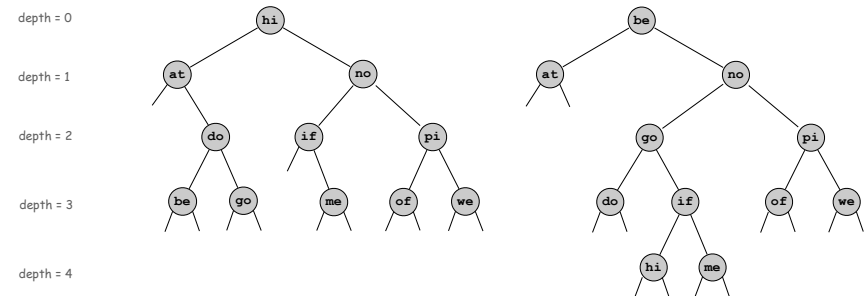|	Running Time		Frequency Count			
implementation	get	put	Moby	100K	200K	1M
unordered array	N	N	170 sec	4.1 hr	-	-
ordered array	$\log N$	N	5.8 sec	5.8 min	15 min	2.1 hr
BST	?	?	.95 sec	7.1 sec	14 sec	69 sec

BST: Analysis

Running time per put/get.
- There are many BSTs that correspond to same set of keys.
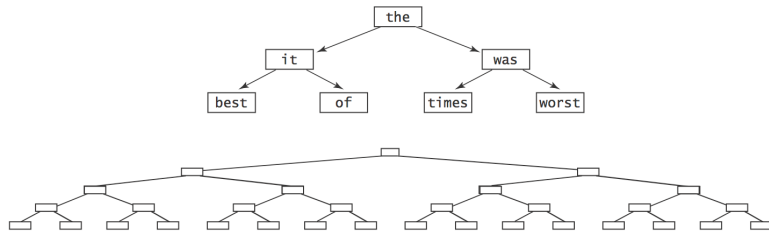- Cost is proportional to depth of node.

number of links on path from root to node

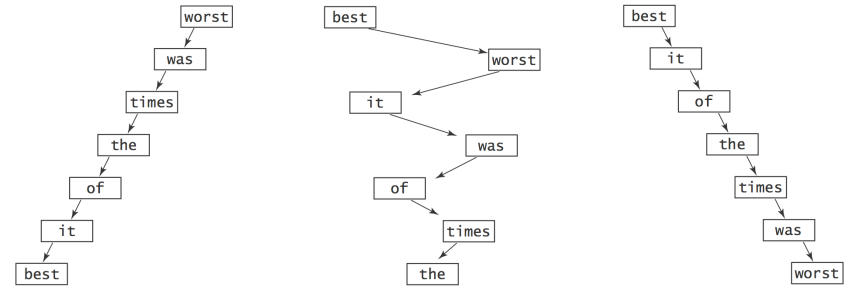Best case. If tree is perfectly balanced, depth is at most $\lg N$.
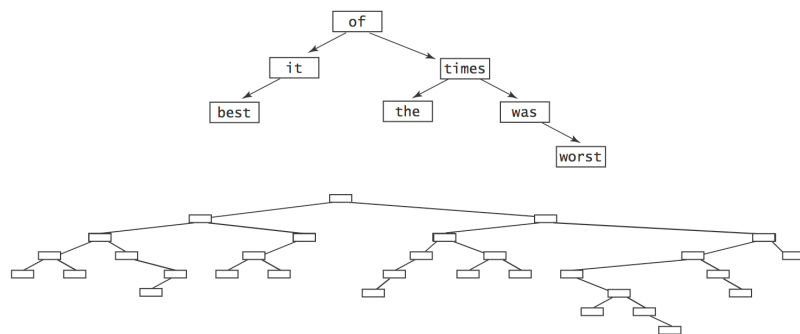

Worst case. If tree is unbalanced, depth can be N.

Average case. If keys are inserted in random order, trees stay ~flat,
and average depth is $2 \ln N$.

requires proof
(see COS 226)

Typical BSTs constructed from randomly ordered keys

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

implementation	Running Time		Frequency Count			
	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?

Slide 34

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

see COS 226

```
import java.util.TreeMap;          Java red-black tree library implementation
import java.util.Iterator;

public class ST<Key extends Comparable<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(); }
}
```

34

Slide 35

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

see COS 226

implementation	Running Time		Frequency Count			
	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

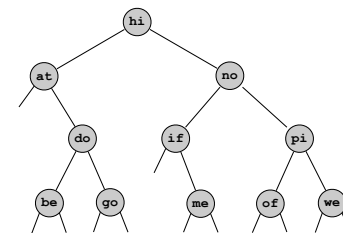† assumes keys inserted in random order

35

Slide 36

Iteration

36

Slide 37

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);
}
```

37

Enhanced For Loop

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

```
BST<String, Integer> bst = new BST<String, Integer>();

...

for (String s : bst) {
   StdOut.println(bst.get(s) + " " + s);
}
```

Enhanced For Loop with BST

BST. Add following code to support enhanced for loop.

```
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> {

      private Stack<Node> stack = new Stack<Node>();

      Inorder() { pushLeft(root); }

      public void remove()     { throw new UnsupportedOperationException(); }
      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.
• Java libraries: `TreeMap, HashMap`.

Remark. Better symbol table implementation improves all clients.

Other Types of Trees

Other types of trees.
- Ancestor tree.

root
Charles
dad — mom
Philip — Elizabeth II
Andrew — Alice — George VI — Elizabeth
George I — Olga — Louis — Victoria — George V — Mary — Claude — Celia

Other types of trees.
- Ancestor tree.
- Parse tree: represents the syntactic structure of a statement, sentence, or expression.

(10 * 12) + 7

Other types of trees.
- Ancestor tree.
- Parse tree.
- Unix file hierarchy.

/
bin — lib — etc — u
aaclarke — cos126 — zrnye
files — grades — submit
hamming — guitar — tsp
Point.java — TSP.java — tsp13509.txt

Other types of trees.
- Ancestor tree.
- Parse tree.
- Unix file hierarchy.
- Phylogeny tree.

gut bacteria
trees
mushrooms
fish
mammals
birds
dragonflies
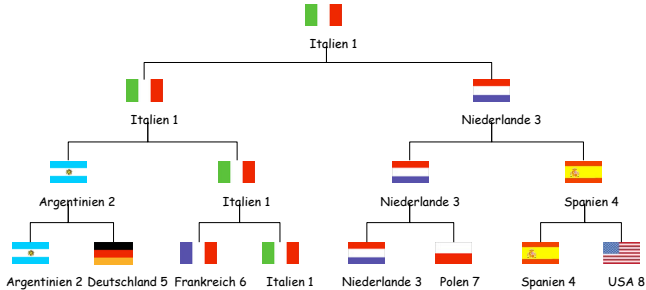beetles

Other Types of Trees

Other types of trees.
- Ancestor tree.
- Parse tree.
- Unix file hierarchy.
- Phylogeny tree.
- GUI containment hierarchy.
- **Tournament trees.**

```
                              Italien 1

              Italien 1                      Niederlande 3

       Argentinien 2      Italien 1     Niederlande 3     Spanien 4

  Argentinien 2  Deutschland 5  Frankreich 6  Italien 1   Niederlande 3  Polen 7   Spanien 4   USA 8
```

Reference: Tobias Lauer

46