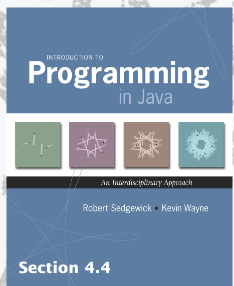
COMPUTER SCIENCE
S E D G E W I C K / W A Y N E



http://introcs.cs.princeton.edu

15. Symbol Tables

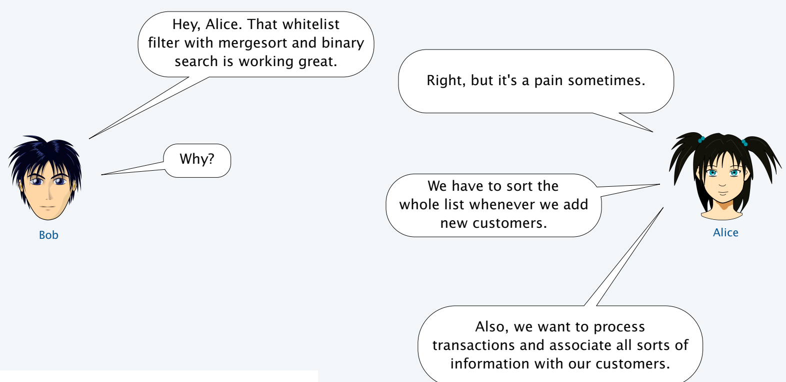
COMPUTER SCIENCE
S E D G E W I C K / W A Y N E

15. Symbol Tables

- **APIs and clients**
- A design challenge
- Binary search trees
- Implementation
- Analysis

CS.15.A.SymbolTables.API

FAQs about sorting and searching



Bottom line. Need a *more flexible* API.

Why are telephone books obsolete?

Unsupported operations
- Change the number associated with a given name.
- Add a new name, associated with a given number.
- Remove a given name and associated number

Observation. Mergesort + binary search has the same problem with add and remove.

see Sorting and Searching lecture

Associative array abstraction

Imagine using arrays whose indices are *string* values.

```
phoneNumber["Alice"] = "(212) 123-4567"
phoneNumber["Bob"]   = "(609) 987-6543"
phoneNumber["Carl"]  = "(800) 888-8888"
phoneNumber["Dave"]  = "(888) 800-0800"
phoneNumber["Eve"]   = "(999) 999-9999"
```

legal code in some programming languages (not Java)

```
transactions["Alice"] = "Dec 12 12:01AM
$111.11 Amazon, Dec 12 1:11 AM $989.99 Ebay"
...
```

```
URL["128.112.136.11"] = "www.cs.princeton.edu"
URL["128.112.128.15"] = "www.princeton.edu"
URL["130.132.143.21"] = "www.yale.edu"
URL["128.103.060.55"] = "www.harvard.edu"
```

```
IPaddr["www.cs.princeton.edu"] = "128.112.136.11"
IPaddr["www.princeton.edu"]    = "128.112.128.15"
IPaddr["www.yale.edu"]         = "130.132.143.21"
IPaddr["www.harvard.edu"]      = "128.103.060.55"
```

A fundamental abstraction

- Use *keys* to access associated *values*.
- Keys and values could be any type of data.
- Client code could not be simpler.

Q. How to implement?

Symbol table ADT

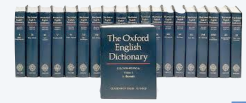
A symbol table is an ADT whose values are sets of key-value pairs, with keys all different.

Basic symbol-table operations

- Associate a given key with a given value.
  [If the key is *not* in the table, add it to the table.]
  [If the key *is* in the table, change its value.]
- Return the value associated with a given key.
- Test if a given key is in the table.
- Iterate though the keys.

Useful additional assumptions

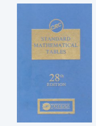
- Keys are comparable and iteration is in order.
- No limit on number of key-value pairs.
- All keys *not* in the table associate with *null*.



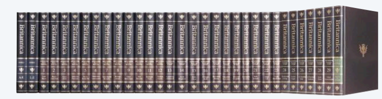
key: word
value: definition

key: time+channel
value: TV show

key: name
value: phone number



key: number
value: function value



key: term value: article

Benchmark example of symbol-table operations

Application. Count frequency of occurrence of strings in StdIn.

Keys. Strings from a sequence.
Values. Integers.

key	it	was	the	best	of	times	it	was	the	worst
value	1	1	1	1	1	1	2	2	2	1

symbol-table contents after operation

it 1	it 1	it 1	best 1	best 1	best 1	best 1	best 1	best 1	best 1
	was 1	the 1	it 1	of 1	of 1	of 1	of 1	of 1	of 1
		was 1	the 1	it 1	it 1	it 2	it 2	it 2	it 2
			was 1	the 1	the 1	the 1	the 1	the 2	the 2
				was 1	times 1	times 1	times 1	times 1	times 1
					was 1	was 1	was 2	was 2	was 2
									worst 1

change the value

Parameterized API for symbol tables

Goal. Simple, safe, and clear client code for symbol tables holding any type of data.

Java approach: Parameterized data types (generics)

- Use placeholder type names for *both* keys and values.
- Substitute concrete types for placeholder in clients.

"implements compareTo()"

Symbol Table API

public class ST<Key extends Comparable<Key>, Value>	
ST<Key, Value>()	*create a symbol table*
void put(Key key, Value val)	*associate* key *with* val
Value get(Key key)	*return value associated with* key*,* null *if none*
boolean contains(Key key)	*is there a value associated with key?*
Iterable<Key> keys()	*all the keys in the table*

Aside: Iteration (client code)

Q. How to print the contents of a stack/queue?

A. Use Java's *foreach* construct.

Enhanced for loop.
- Useful for any collection.
- Iterate through each entry in the collection.
- Order determined by implementation.
- Substantially simplifies client code.
- Works when API "`implements Iterable`".

Java foreach construct

```
Stack<String> stack = new Stack<String>();
...
for (String s : stack)
   StdOut.println(s);
...
```

public class Stack<Item> implements Iterable<Item>	
Stack<Item>()	*create a stack of objects, all of type* Item
void push(Item item)	*add* item *to stack*
Item pop()	*remove and return item most recently pushed*
boolean isEmpty()	*is the stack empty?*
int size()	*# of objects on the stack*

Performance specification. Constant-time per entry.

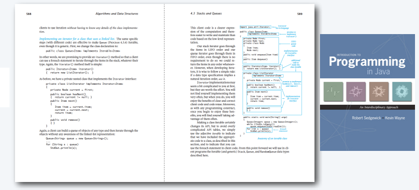
Aside: Iteration (implementation)

Q. How to "implement `Iterable`"?

A. We did it for Stack and Queue, so you don't have to.

public class Stack<Item> implements Iterable<Item>	
Stack<Item>()	*create a stack of objects, all of type* Item
void push(Item item)	*add* item *to stack*
Item pop()	*remove and return item most recently pushed*
boolean isEmpty()	*is the stack empty?*
int size()	*# of objects on the stack*

A. Implement an `Iterator` (see text pp. 588-89)



Meets performance specification. Constant-time per entry.

Bottom line. *Use iteration* in client code that uses collections.

Why ordered keys?

Natural for many applications
- Numeric types.
- Strings.
- Date and time.
- Client-supplied types (Account numbers, ...).

Enables useful API extensions
- Provide the keys in sorted order.
- Find the *k*th largest key.

Enables efficient implementations
- Mergesort.
- Binary search.
- BSTs (this lecture).

thingsorganizedneatly.tumblr.com

Symbol table client example 1: Sort (with dedup)

Goal. Sort lines on standard input (and remove duplicates).
- Key type. `String` (line on standard input).
- Value type. (ignored).

```
public class Sort
{
   public static void main(String[] args)
   {  // Sort lines on StdIn
      BST<String, Integer> st = new BST<String, Integer>();
      while (StdIn.hasNextLine())
         st.put(StdIn.readLine(), 0);
      for (String s : st.keys())
         StdOut.println(s);
   }
}
```

foreach construct

```
% more tale.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
```

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

Symbol table client example 2: Frequency counter

Goal. Compute frequencies of words on standard input.

- Key type. `String` (word on standard input).
- Value type. `Integer` (frequency count).

```
public class Freq
{
   public static void main(String[] args)
   {  // Frequency counter
      BST<String, Integer> st = new BST<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.keys())
         StdOut.printf("%8d  %s\n", st.get(s), s);
   }
}
```

```
% more tale.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
it
it
it
it
it
```

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


Symbol table client example 3: Index

Goal. Print index to words on standard input.

- Key type. `String` (word on standard input).
- Value type. `Queue<Integer>` (indices where word occurs).

```
public class Index
{
   public static void main(String[] args)
   {
      BST<String, Queue<Integer>> st;
      st = new BST<String, Queue<Integer>>();
      int i = 0;
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (!st.contains(key))
            st.put(key, new Queue<Integer>());
         st.get(key).enqueue(i++);
      }
      for (String s : st.keys())
         StdOut.println(s + " " + st.get(s));
   }
}
```

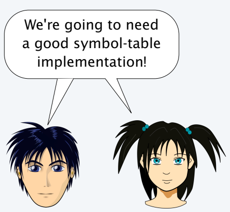
```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it
it
it
it
it
it
```

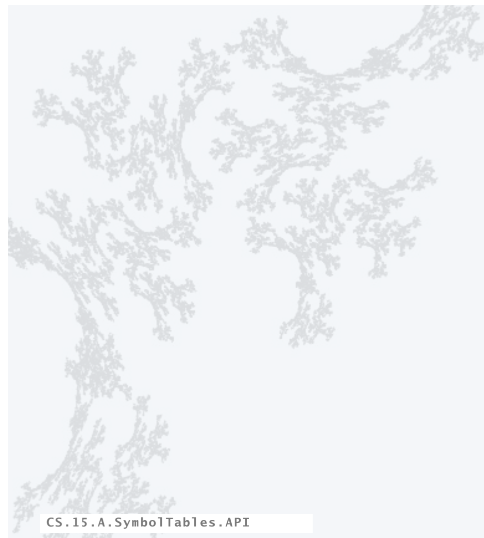
```
% java Index < tale.txt
age 15 21
belief 29
best 3
darkness 47
despair 59
epoch 27 33
foolishness 23
hope 53
incredulity 35
it 0 6 12 18 24 30 36 42 48 54
light 41
of 4 10 16 22 28 34 40 46 52 58
season 39 45
spring 51
the 2 8 14 20 26 32 38 44 50 56
times 5 11
was 1 7 13 19 25 31 37 43 49 55
winter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


Symbol-table applications

Symbol tables are *ubiquitous* in today's computational infrastructure.

application	*key*	*value*
contacts	name	phone number, address
credit card	account number	transaction details
file share	name of song	computer ID
dictionary	word	definition
web search	keyword	list of web pages
book index	word	list of page numbers
cloud storage	file name	file contents
domain name service	domain name	IP address
reverse DNS	IP address	domain name
compiler	variable name	value and type
internet routing	destination	best route
...	...	...





COMPUTER SCIENCE
SEDGEWICK / WAYNE

CS.15.A.SymbolTables.API

COMPUTER SCIENCE
SEDGEWICK / WAYNE

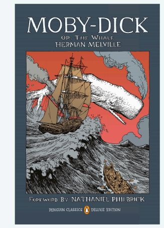
15. Symbol Tables

- APIs and clients
- **A design challenge**
- Binary search trees
- Implementation
- Analysis

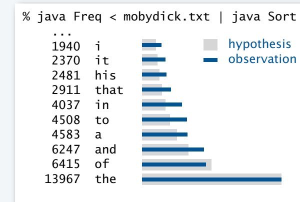
CS.15.B.SymbolTables.Challenge

Benchmark

Application. Linguistic analysis



Zipf's law (for a natural language corpus)
- Suppose most frequent word occurs about *t* times.
- 2nd most frequent word occurs about *t*/2 times.
- 3rd most frequent word occurs about *t*/3 times.
- 4th most frequent word occurs about *t*/4 times.



Goal. Validate Zipf's law for real natural language data.

Method. `% java Freq < data.txt | java Sort`

Required. Efficient symbol-table implementation.

Benchmark statistics

Goal. Validate Zipf's law for real natural language data.

Method. `% java Freq < data.txt | java Sort`

WORTSCHATZ
UNIVERSITÄT LEIPZIG

file	*description*	*words*	*distinct*
mobydick.txt	Melville's *Moby Dick*	210,028	16,834
liepzig100k.txt	100K random sentences	2,121,054	144,256
liepzig200k.txt	200K random sentences	4,238,435	215,515
liepzig1m.txt	1M random sentences	21,191,455	534,580

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

Required. Efficient symbol-table implementation.

Strawman I: Ordered array

Idea
- Keep keys in order in an array.
- Keep values in a parallel array.

Reasons (see "Sorting and Searching" lecture)
- Takes advantage of fast sort (mergesort).
- Enables fast search (binary search).

Known challenge. How big to make the arrays?

Fatal flaw. How to insert a new key?
- To keep key array in order, need to move larger entries à la insertion sort.
- Hypothesis: Quadratic time for benchmark.

easy to validate with experiments

keys	*values*
alice	121
bob	873
carlos	884
carol	712
dave	585
erin	247
eve	577
oscar	675
peggy	895
trent	557
trudy	926
walter	51
wendy	152

keys	*values*
alice	121
bob	873
carlos	884
carol	712
craig	999
dave	585
erin	247
eve	577
oscar	675
peggy	895
trent	557
trudy	926
walter	51
wendy	152

Strawman II: Linked list

Idea
- Keep keys in order in a linked list.
- Add a value to each node.

Reason. Meets memory-use performance specification.

Fatal flaw. How to search?
- Binary search requires indexed access.
- Example: How to access the middle of a linked list?
- Only choice: search *sequentially* through the list.
- Hypothesis: Quadratic time for benchmark.

easy to validate with experiments

Design challenge

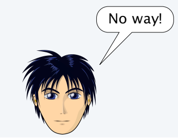
Implement scalable symbol tables.

Goal. Simple, safe, clear, and *efficient* client code.

Only slightly more costly than stacks or queues!

Performance specifications
- Order of growth of running time for `put()`, `get()` and `contains()` is logarithmic.
- Memory use is proportional to the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

Are such guarantees achievable??
Can we implement associative arrays with just log-factor extra cost??




```
phoneNumber["Alice"] = "(212) 123-4567"
```

This lecture. Yes way!

COMPUTER SCIENCE
SEDGEWICK / WAYNE

CS.15.B.SymbolTables.Challenge

COMPUTER SCIENCE
SEDGEWICK / WAYNE

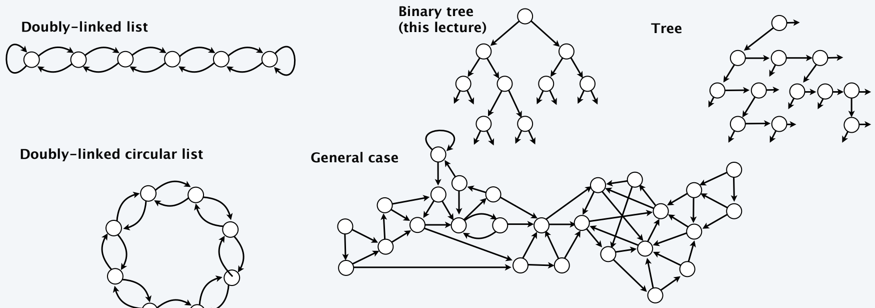
15. Symbol Tables

- APIs and clients
- A design challenge
- **Binary search trees**
- Implementation
- Analysis

CS.15.C.SymbolTables.BSTs

Doubly-linked data structures

With two links ( ) a wide variety of data structures are possible.



Maintenance can be complicated!

From the point of view of a particular object, all of these structures look the same.

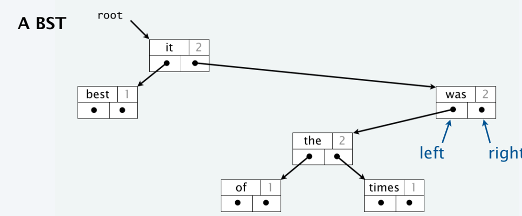
A doubly-linked data structure: binary search tree

Binary search tree (BST)

- A recursive data structure containing distinct comparable keys that is *ordered*.
- Def. A *BST* is a null or a reference to a *BST node* (the *root*).
- Def. A *BST node* is a data type that contains references to a key, a value, and two BSTs, a *left* subtree and a *right* subtree.
- Ordered. All keys in the *left* subtree of each node are *smaller* than its key and all keys in the *right* subtree of each node are *larger* than its key.

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

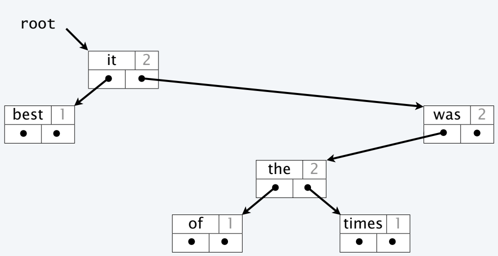
A BST



BST processing code

Standard operations for processing data structured as a binary search tree

- Search for the value associated with a given key.
- Add a new key-value pair.
- Traverse the BST (visit every node, in order of the keys).
- Remove a given key and associated value (not addressed in this lecture).

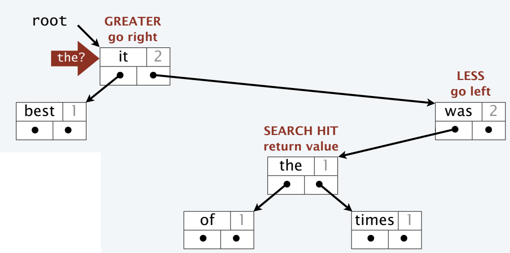


BST processing code: Search

Goal. Find the value associated with a given key in a BST.

- If *less* than the key at the current node, go *left*.
- If *greater* than the key at the current node, go *right*.

Example. get("the")



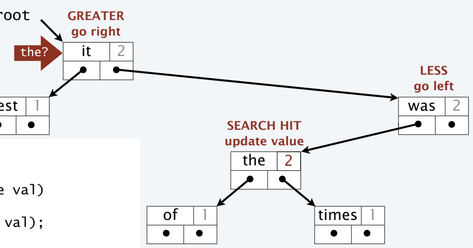

```
public Value get(Key key)
{  return get(root, key);  }
private Value get(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return get(x.left,  key);
   else if (cmp  > 0) return get(x.right, key);
   else if (cmp == 0) return x.val;
}
```

BST processing code: Associate a new value with a key

Goal. Associate a new value with a given key in a BST.

- If *less* than the key at the current node, go *left*.
- If *greater* than the key at the current node, go *right*.

Example. `put("the", 2)`



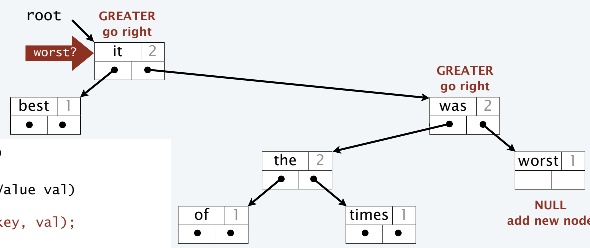

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key,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


BST processing code: Add a new key

Goal. Add a new key-value pair to a BST.

- Search for key.
- Return link to new node when *null* reached.

Example. `put("worst", 1)`

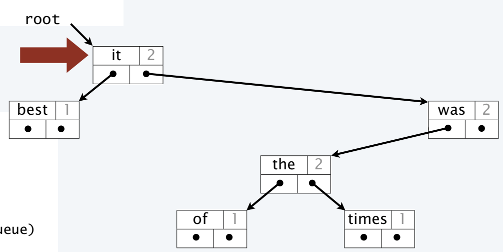



```
public void put(Key key, Value val)
{  root = put(root, key, val);  }
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key,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


BST processing code: Traverse the BST

Goal. Put keys in a BST on a queue, in sorted order.

- Do it for the left subtree.
- Put the key at the root on the queue.
- Do it for the right subtree.




```
public Iterable<Key> keys()
{
   Queue<Key> queue = new Queue<Key>();
   inorder(root, queue);
   return queue;
}
private void inorder(Node x, Queue<Key> queue)
{
   if (x == null) return;
   inorder(x.left, queue);
   q.enqueue(x.key);
   inorder(x.right, queue);
}
```

Queue

best	it	of	the	times	was

COMPUTER SCIENCE
SEDGEWICK / WAYNE

CS.15.C.SymbolTables.BSTs

15. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- **Implementation**
- Analysis

ADT for symbol tables: review

A symbol table is an idealized model of an associative storage mechanism.

An ADT allows us to write Java programs that use and manipulate symbol tables.

API

public class ST<Key extends Comparable<Key>, Value>	
ST<Key, Value>()	*create a symbol table*
void put(Key key, Value val)	*associate key with val*
Value get(Key key)	*return value associated with key, null if none*
boolean contains(Key key)	*is there a value associated with key?*
Iterable<Key> keys()	*all the keys in the table*

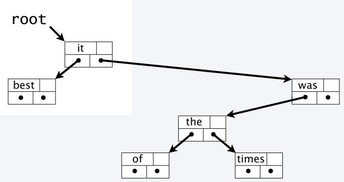
Performance specifications

- Order of growth of running time for put(), get() and contains() is logarithmic.
- Memory use is proportional to the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

Symbol table implementation: Instance variables and constructor

Data structure choice. Use a BST to hold the collection.

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root = null;

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

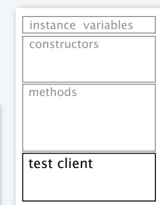


BST implementation: Test client (frequency counter)

```
public static void main(String[] args)
{
   BST<String, Integer> st = new BST<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.keys())
      StdOut.printf("%8d  %s\n", st.get(s), s);
}
```

```
% java BST < tale.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
```

What we *expect*, once the implementation is done.



BST implementation: Methods

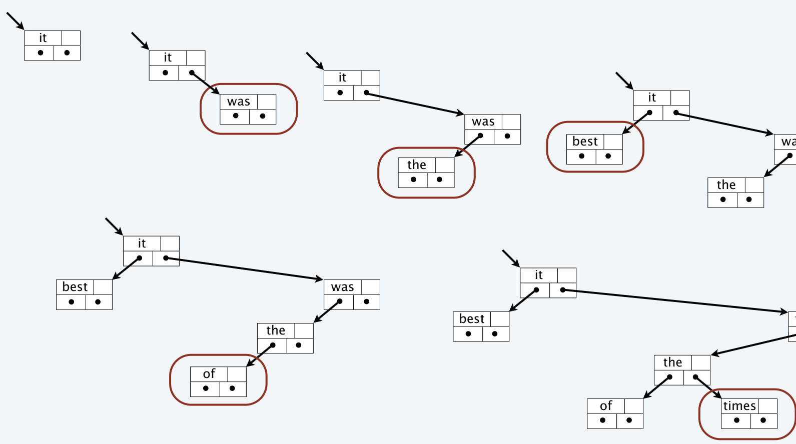
Methods define data-type operations (implement the API).


```
public class BST<Key extends Comparable<Key>, Value>
{
...

   public boolean isEmpty()
   {  return root == null;  }

   public void put(Key key, Value value)
   {  /* See BST add slides and next slide. */  }

   public Value get(Key key)
   {  /* See BST search slide and next slide. */  }

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

   public Iterable<Key> keys()
   {  /* See BST traverse slide and next slide. */  }

...
}
```


BST implementation

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root = null;                       ← instance variable

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

   public boolean isEmpty()
   {  return root == null;  }

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

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

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

   public Iterable<Key> keys()
   {
      Queue<Key> q = new Queue<Key>();
      inorder(root, q);
      return q;
   }

                                                   private methods →
   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;
   }

   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;
   }

   private void inorder(Node x, Queue<Key> q)
   {
      if (x == null) return;
      inorder(x.left, q);
      q.enqueue(x.key);
      inorder(x.right, q);
   }

   public static void main(String[] args)
   {  // Frequency counter  }                      ← test client
}
```


Trace of BST construction



COMPUTER SCIENCE

SEDGEWICK / WAYNE

CS.15.D.SymbolTables.Implementation

COMPUTER SCIENCE
SEDGEWICK / WAYNE

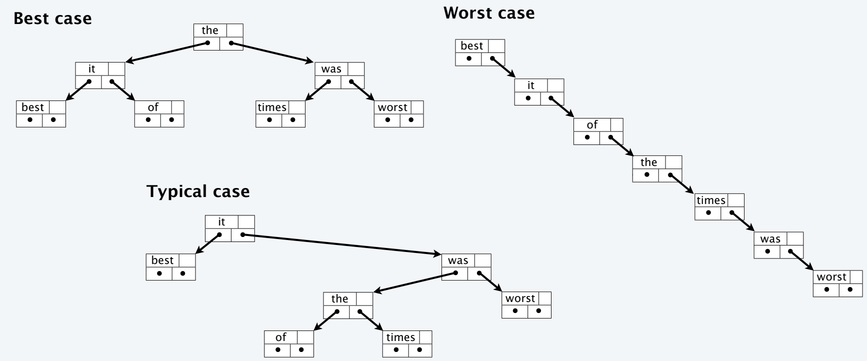
15. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- Implementation
- **Analysis**

CS.15.E.SymbolTables.Analysis

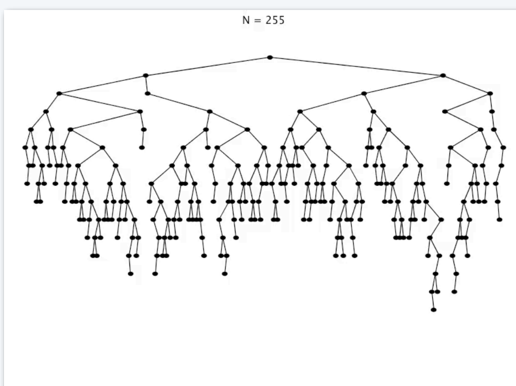
BST analysis

Costs depend on order of key insertion.



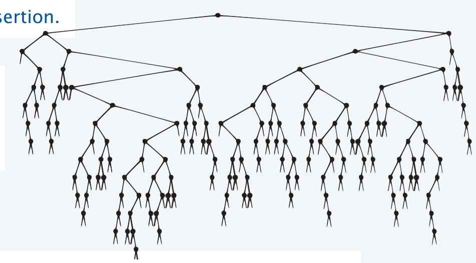
BST insertion: random order visualization

Insert keys in random order.
- Tree is roughly balanced.
- Tends to stay that way!



BST analysis

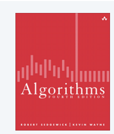
Running time depends on order of key insertion.



Model. Insert keys in random order.
- Tree is roughly balanced.
- Tends to stay that way!

Proposition. Building a BST by inserting N randomly ordered keys into an initially empty tree uses $\sim 2\ N \ln N$ (about $1.39\ N \lg N$) compares.

Proof. A very interesting exercise in discrete math.



Interested in details? Take a course in algorithms.

Benchmarking the BST implementation

BST *implements the associative-array abstraction* for randomly ordered keys.

Symbol table API

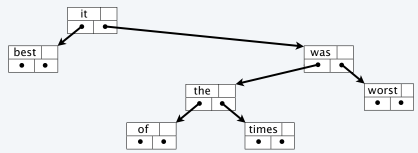
public class ST<Key extends Comparable<Key>, Value>	
ST<Key, Value>()	*create a symbol table*
void put(Key key, Value value)	*associate key with value*
Value get(Key key)	*return value associated with key, null if none*
boolean contains(Key key)	*is there a value associated with key?*
Iterable<Key> keys()	*all the keys in the table (sorted)*

for random keys (but stay tuned)

Performance specifications

- Order of growth of running time for put(), get() and contains() is logarithmic.
- Memory use is proportional to the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

Made possible by *binary tree data structure.*



Empirical tests of BSTs

Count number of words that appear more than once in StdIn.

Frequency count without the output

N	T_N (seconds)	$T_N/T_{N/2}$
1 million	5	
2 million	9	1.8
4 million	17	1.9
8 million	34	2
16 million	72	2.1
...		
1 BILLION	4608	2

```
% java Generator 1000000 ...
263934  (5 seconds)
% java Generator 2000000 ...
593973  (9 seconds)
% java Generator 4000000 ...
908795  (17 seconds)
% java Generator 8000000 ...
996961  (34 seconds)
% java Generator 16000000 ...
999997  (72 seconds)
```

```
... = 6 0123456789 | java DupsBST
```

6-digit integers

WORTSCHATZ
UNIVERSITÄT LEIPZIG

Easy to process 21M word corpus
NOT possible with brute-force

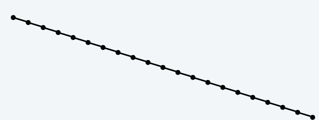
Confirms hypothesis that order of growth is $N \log N$

WILL scale

Performance guarantees

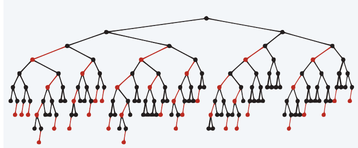
Practical problem. Keys may *not* be randomly ordered.

- BST may be unbalanced.
- Running time may be quadratic.
- Happens in practice (insert keys in order).



Remarkable resolution.

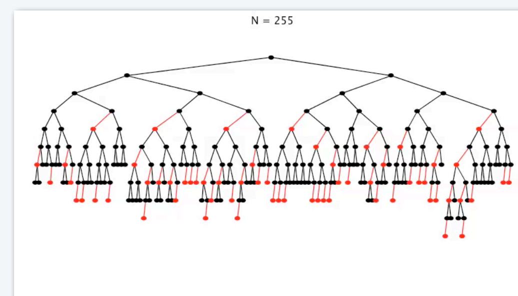
- *Balanced tree* algorithms perform simple transformations that guarantee balance.
- AVL trees (Adelson-Velskii and Landis, 1962) proved concept.
- Red-black trees (Guibas and Sedgewick, 1979) are implemented in many modern systems.



Red-black tree insertion: random order visualization

Insert keys in random order.

- Same # of black links on every path from root to leaf.
- No two red links in a row.
- Tree is roughly balanced.
- Guaranteed to stay that way!



ST implementation with guaranteed logarithmic performance

Java's TreeMap library uses red-black trees.

```
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value>
{
   private TreeMap<Key, Value> st = new TreeMap<Key, Value>();

   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 Iterable<Key> keys()      { return st.keySet();         }
}
```

Proposition. In a red-black tree of size N, `put()`, `get()` and `contains()` are *guaranteed* to use fewer than $2\lg N$ compares.

Several other useful operations also available.

Proof. A fascinating exercise in algorithmics.

Interested in details? Take a course in algorithms.

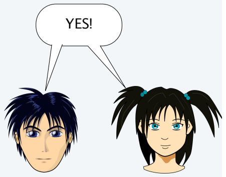
Summary

BSTs. Simple symbol-table implementation, usually efficient.
Red-black trees. More complicated variation, *guaranteed* to be efficient.
Applications. Many, many, many things are enabled by efficient symbol tables.

Example. Search among 1 trillion customers with less than 80 compares!

Example. Search among all the atoms in the universe with less than 200 compares!

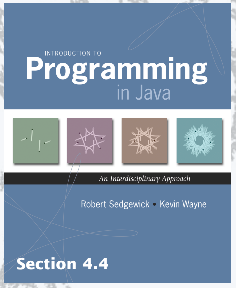
Can we implement associative arrays with just log-factor extra cost??



COMPUTER SCIENCE
SEDGEWICK / WAYNE

CS.15.E.SymbolTables.Analysis

COMPUTER SCIENCE
SEDGEWICK / WAYNE



http://introcs.cs.princeton.edu

15. Symbol Tables