# Symbol Tables

- API
- basic implementations
- iterators
- Comparable keys
- challenges

References: Algorithms in Java, Chapter 12.
Intro to Algs and Data Structs, Chapter 4.
Intro to Programming, Section 4.4.





## Symbol Tables

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

Example: DNS lookup.
- Insert URL with specified IP address.
- Given URL, find corresponding IP address

| URL (key) | IP address (value) |
|---|---|
| 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

Can interchange roles: given IP address find corresponding URL



API
basic implementations
iterators
Comparable keys
challenges

## 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

Associative array abstraction. Unique value associated with each key.

Symbol table API.

- `put(key, val)` insert the key-value pair
- `get(key)` search: return value associated with given key
- `remove(key)` remove the key
- `contains(key)` is given key present?
- `iterator()` return iterator over all keys

Our conventions.

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Implementations all have

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

- Method `put()` overwrites old value with new value.

```
a[key] = val;
```

Some languages (not Java) allow this notation



## ST client: make a dictionary and process lookup requests

Command line arguments

- a comma-separated value (CSV) file
- key field
- value field

Example 1: DNS lookup

URL is key IP is value

```
% java Lookup ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found
```

IP is key URL is value

```
% java Lookup ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found
```

```
% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
sohu.com,61.135.133.103
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...
```



## ST client: make a dictionary and process lookup requests

```
public class Lookup
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);
      String[] database = in.readAll().split("\n");

      ST<String, String> st = new ST<String, String>();
      for (int i = 0; i < database.length; i++)
      {
         String[] tokens = database[i].split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }

      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if (!st.contains(s)) StdOut.println("Not found");
         else                 StdOut.println(st.get(s));
      }
   }
}
```

process input

build symbol table

process lookups



## ST client: make a dictionary and process lookup requests

Command line arguments

- a comma-separated value (CSV) file
- key field
- value field

Example 2: Amino acids

codon is key name is value

```
% % java Lookup amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine
```

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
CGA,Arg,R,Arginine
CGG,Arg,R,Arginine
ATT,Ile,I,Isoleucine
ATC,Ile,I,Isoleucine
ATA,Ile,I,Isoleucine
ATG,Met,M,Methionine
...
```

## ST client: make a dictionary and process lookup requests

Command line arguments

- a comma-separated value (CSV) file
- key field
- value field

Example 3: Class lists

login is key   name is value

```
% java Lookup classlist.csv 3 1
jsh
Jeffrey Scott Harris
dgtwo
Daniel Gopstein
ye
Michael Weiyang Ye
```

login is key   precept is value

```
% java Lookup classlist.csv 3 2
jsh
P01A
dgtwo
P01
```

```
% more classlist.csv
10,Bo Ling,P03,bling
10,Steven A Ross,P01,saross
10,Thomas Oliver Horton
Conway,P03,oconway
08,Michael R. Corces
Zimmerman,P01A,mcorces
09,Bruce David Halperin,P02,bhalperi
09,Glenn Charles Snyders Jr.,P03,gsnyders
09,Siyu Yang,P01A,siyuyang
08,Taofik O. Kolade,P01,tkolade
09,Katharine Paris
Klosterman,P01A,kkloster
SP,Daniel Gopstein,P01,dgtwo
10,Sauhard Sahi,P01,ssahi
10,Eric Daniel Cohen,P01A,edcohen
09,Brian Anthony Geistwhite,P02,bgeistwh
09,Boris Pivtorak,P01A,pivtorak
09,Jonathan Patrick
Zebrowski,P01A,jzebrows
09,Dexter James Doyle,P01A,ddoyle
09,Michael Weiyang Ye,P03,ye
08,Delwin Uy Olivan,P02,dolivan
08,Edward George Conbeer,P01A,econbeer
09,Mark Daniel Stefanski,P01,mstefans
09,Carter Adams Cleveland,P03,cclevela
10,Jacob Stephen Lewellen,P02,jlewelle
10,Ilya Trubov,P02,itrubov
09,Kenton William Murray,P03,kwmurray
07,Daniel Steven Marks,P02,dmarks
09,Vittal Kadapakkam,P01,vkadapak
10,Eric Ruben Domb,P01A,edomb
07,Jie Wu,P03,jiewu
08,Pritha Ghosh,P02,prithag
10,Minh Quang Anh Do,P01,mqdo
...
```



## Keys and Values

Associative array abstraction.

- Unique value associated with each key
- If client presents duplicate key, overwrite to change value.

```
a[key] = val;
```

Key type: several possibilities

1. Assume keys are any generic type, use `equals()` to test equality.
2. Assume keys are `Comparable`, use `compareTo()`.
3. Use `equals()` to test equality, make some other assumptions.

Value type. Any generic type.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `BigInteger`.
- Mutable in Java: `Date`, `GregorianCalendar`, `StringBuilder`.



## Elementary ST implementations

Unordered array
Ordered array
Unordered linked list
Ordered linked list

Why study elementary implementations?

- API details need to be worked out
- performance benchmarks
- method of choice can be one of these in many situations
- basis for advanced implementations

Always good practice to study elementary implementations

## Unordered array ST implementation

Maintain parallel arrays of keys and values.

Instance variables
- array `keys[]` holds the keys.
- array `vals[]` holds the values.
- integer `N` holds the number of entries.

Need to use standard array-doubling technique

`N = 6`

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `keys[]` | it | was | the | best | of | times |
| `vals[]` | 2 | 2 | 1 | 1 | 1 | 1 |

Alternative: define inner type for entries
- space overhead for entry objects
- more complicated code



## Unordered array ST implementation (skeleton)

```
@SuppressWarnings("unchecked")

public class UnorderedST<Key, Val>
{
   private Val[] vals;
   private Key[] keys;
   private int N = 0;

   public UnorderedST(int maxN)
   {
     keys = (Key[]) new Object[maxN];
     vals = (Val[]) new Object[maxN];
   }

   public boolean isEmpty()
   {  return N == 0; }

   public void put(Key key, Val, val)
   // see next slide

   public Val get(Key key)
   // see next slide
}
```

← suppress annoying message about ugly casts

← parallel arrays lead to cleaner code than defining a type for entries

← standard array doubling code omitted

← standard ugly casts



## Unordered array ST implementation (search)

```
public Val get(Key key)
{
   for (int i = 0; i < N; i++)
      if (keys[i].equals(key))
         return vals[i];
   return null;
}
```

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `keys[]` | it | was | the | best | of | times |
| `vals[]` | 2 | 2 | 1 | 1 | 1 | 1 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `keys[]` | it | was | the | best | of | times |
| `vals[]` | 2 | 2 | 1 | 1 | 1 | 1 |

`get("worst")` returns `null`

Key, Value are generic and can be any type

Java convention: all objects implement equals()



## Unordered array ST implementation (insert)

```
public void put(Key key, Val, val)
{
   int i;
   for (int i = 0; i < N; i++)
      if (key.equals(keys[i]))
         break;
   vals[i] = val;
   keys[i] = key;
   if (i == N) N++;
}
```

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `keys[]` | it | was | the | best | of | times |
| `vals[]` | 2 | 2 | 1 | 1 | 1 | 1 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| `keys[]` | it | was | the | best | of | times |
| `vals[]` | 2 | 2 | 2 | 1 | 1 | 1 |

`put("the", 2)` overwrites the 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| `keys[]` | it | was | the | best | of | times | worst |
| `vals[]` | 2 | 2 | 2 | 1 | 1 | 1 | 1 |

`put("worst", 1)` adds a new entry

Associative array abstraction
- must search for key and overwrite with new value if it is there
- otherwise, add new key, value at the end (as in stack)

## Java conventions for `equals()`

All objects implement `equals()` but default implementation is `(x == y)`

is the object referred to by x
the same object that is referred to by y?

Customized implementations.

`String, URL, Integer.`

User-defined implementations.

Some care needed (example: type of argument must be `Object`)

Equivalence relation. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is `true`.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: If `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is `false`.
- Consistency: Multiple calls to `x.equals(y)` return same value.



## Implementing `equals()`

Seems easy

```
public       class PhoneNumber
{
   private       int area, exch, ext;

   ...

   public boolean equals(PhoneNumber y)
   {




      PhoneNumber a = this;
      PhoneNumber b = (PhoneNumber) y;
      return (a.area == b.area)
             && (a.exch == b.exch)
             && (a.ext == b.ext);
   }
}
```



## Implementing `equals()`

Seems easy, but requires some care

enforce immutability

```
public final class PhoneNumber
{
   private final int area, exch, ext;

   ...

   public boolean equals(   Object   y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      PhoneNumber a = this;
      PhoneNumber b = (PhoneNumber) y;
      return (a.area == b.area)
             && (a.exch == b.exch)
             && (a.ext == b.ext);
   }
}
```

Must be Object.
Why? Experts still debate.

Optimize for true object equality

If I'm executing this code,
I'm not null.

Objects must be in the same class.



## Linked list ST implementation

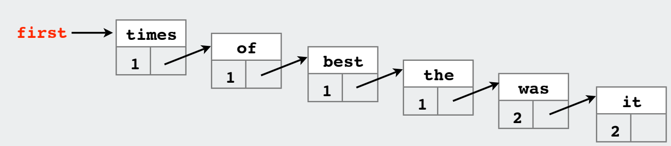
Maintain a linked list with keys and values.

inner `Node` class

- instance variable `key` holds the key
- instance variable `val` holds the value

instance variable

- node `first` refers to the first node in the list



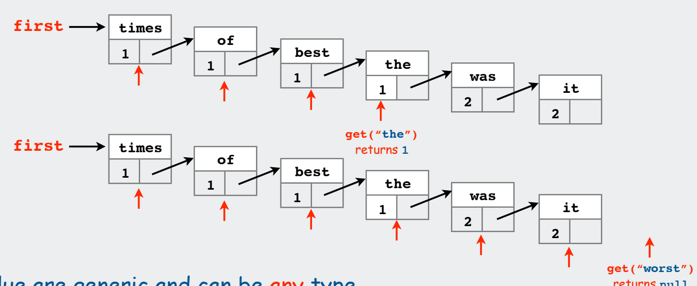
## Linked list ST implementation (skeleton)

```
public class LinkedListST<Key, Val>
{
    private Node first;            ← instance variable

    private class Node             ← inner class
    {
        Key key;
        Val val;
        Node next;
        Node(Key key, Val val, Node next)
        {
            this.key   = key;
            this.val = val;
            this.next  = next;
        }
    }

   public void put(Key key, Val, val)
   // see next slides

   public Val get(Key key)
   // see next slides
}
```



## Linked list ST implementation (search)

```
public Val get(Key key)
{
   for (Node x = first; x != null; x = x.next))
      if (key.equals(x.key))
         return vals[i];
   return null;
}
```




Key, Value are generic and can be any type

Java convention: all objects implement equals()

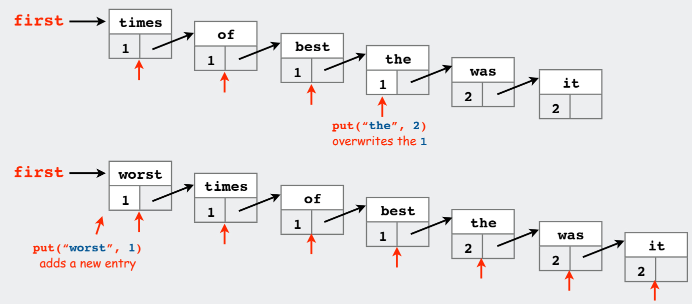


## Linked list ST implementation (insert)

```
public void put(Key key, Val, val)
{
   for (Node x = first; x != null; x = x.next)
      if (key.equals(x.key))
         { x.value = value; return; }
   first = new Node(key, value, first);
}
```

Associative array abstraction

- must search for key and, if it is there, overwrite with new value
- otherwise, add new key, value at the beginning (as in stack)






API
basic implementations
iterators
Comparable keys
challenges

## Iterators

Symbol tables should be `Iterable`

Q. What is `Iterable`?
A. Implements `iterator()`

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an iterator?
A. Implements `hasNext()` and `next()`.

`java.util.Iterator`

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

optional in Java
use at your own risk

Q. Why should symbol tables be iterable?
A. Java language supports elegant client code for iterators

"foreach" statement

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

equivalent code

```
Iterator<String> i = st.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(st.get(s));
}
```



## Iterable ST client: count frequencies of occurrence of input strings

Standard input: A file (of strings)
Standard output: All the distinct strings in the file with frequency

```
% 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

% java FrequencyCount < tiny.txt
2 age
1 best
1 foolishness
4 it
4 of
4 the
2 times
4 was
1 wisdom
1 worst
```

tiny example
24 words
10 distinct

```
% 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
we had everything before us
we had nothing before us
...
% java FrequencyCount < tale.txt
2941 a
1 aback
1 abandon
10 abandoned
1 abandoning
1 abandonment
1 abashed
1 abate
1 abated
5 abbaye
2 abed
1 abhorrence
1 abided
1 abiding
1 abilities
2 ability
1 abject
1 ablaze
17 able
1 abnegating
```

real example
137177 words
9888 distinct



## Iterable ST client: count frequencies of occurrence of input strings

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

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

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

   }
}
```

read a string
insert
increment
print all strings

Note: Only slightly more work required to build an index of all of the places where each key occurs in the text.



## Iterators for array, linked list ST implementations

```
import java.util.Iterator;
public class UnorderedST<Key, Value>
             implements Iterable<Key>
{
   ...

   public Iterator<Key> iterator()
   { return new ArrayIterator(); }

   private class ArrayIterator
      implements Iterator<Key>
   {
      private int i = 0;

      public boolean hasNext()
      {  return i < N;  }

      public void remove() { }

      public Key next()
      {  return keys[i++];  }
   }
}
```

```
import java.util.Iterator;
public class LinkedListST<Key, Value>
             implements Iterable<Key>
{
   ...

   public Iterator<Key> iterator()
   { return new ListIterator();  }

   private class ListIterator
             implements Iterator<Key>
   {
      private Node current = first;

      public boolean hasNext()
      {  return current != null;  }

      public void remove() { }

      public Key next()
      {
         Key key = current.key;
         current = current.next;
         return key;
      }
   }
}
```

## Iterable ST client: A problem?

Use **UnorderedST** in **FrequencyCount**

```
% 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

% java FrequencyCount < tiny.txt
4 it
4 was
4 the
1 best
4 of
2 times
1 worst
2 age
1 wisdom
1 foolishness
```

Use **LinkedListST** in **FrequencyCount**

```
% 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

% java FrequencyCount < tiny.txt
1 foolishness
1 wisdom
2 age
1 worst
2 times
4 of
1 best
4 the
4 was
4 it
```

Clients who use Comparable keys might expect ordered iteration

- not a requirement for some clients
- not a problem if postprocessing, e.g. with sort or grep
- not in API



API
basic implementations
iterators
**Comparable keys**
challenges

## Ordered array ST implementation

Assume that keys are Comparable
Maintain parallel arrays with keys and values that are sorted by key.

Instance variables

- `keys[i]` holds the `i`th smallest key
- `vals[i]` holds the value associated with the `i`th smallest key
- integer `N` holds the number of entries.

Note: no duplicate keys

N = 6

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| keys[] | best | it | of | the | times | was |
| vals[] | 1 | 2 | 1 | 1 | 1 | 2 |

Need to use standard array-doubling technique

Two reasons to consider using ordered arrays

- provides ordered iteration (for free)
- can use binary search to significantly speed up search



## Ordered array ST implementation (skeleton)

```
public class OrderedST
          <Key extends Comparable, Val>
          implements Iterable<Key>
{
   private Val[] vals;
   private Key[] keys;
   private int N = 0;
   public UnorderedST(int maxN)
   {
     keys = (Key[]) new Object[maxN];
     vals = (Val[]) new Object[maxN];
   }

   public boolean isEmpty()
   {  return N == 0; }


   public void put(Key key, Val, val)
   // see next slides

   public Val get(Key key)
   // see next slides
}
```

standard array iterator code omitted

standard array doubling code omitted

## Ordered array ST implementation (search)

### Keeping array in order enables binary search algorithm

```
public Val get(Key key)
 {
     int i = bsearch(key);
     if (i == -1) return null;
     return vals[i];
 }
```

```
private int bsearch(Key key)
 {
     int lo = 0, hi = N-1;
     while (lo <= hi)
     {
         int m = lo + (hi - lo) / 2;
         int cmp = key.compareTo(keys[m]);
         if      (cmp < 0) hi = m - 1;
         else if (cmp > 0) lo = m + 1;
         else return m;
     }
     return -1;
 }
```

| lo | | | | m | | | | hi |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | 4 | | | | 8 |
| age | best | it | of | the | times | was | wisdom | worst |
| 2 | 1 | 4 | 3 | 4 | 2 | 4 | 1 | 1 |

| | | | |
|---|---|---|---|
| 0 | 1 | | 3 |
| age | best | it | of |
| 2 | 1 | 4 | 3 |

| | |
|---|---|
| 2 | 3 |
| it | of |
| 4 | 3 |

| |
|---|
| 3 |
| of |
| 3 |

get("of") returns 3



## Binary search analysis: Comparison count

Def. T(N) ≡ number of comparisons to search in an ST of size N

$$= \underbrace{T(N/2)}_{\text{left or right half}} + \underbrace{1}_{\text{middle}}$$

**Binary search recurrence** $T(N) = T(N/2) + 1$

for $N > 1$, with $T(1) = 0$

- not quite right for odd N
- same recurrence holds for many algorithms
- same number of comparisons for any input of size N.

**Solution of binary search recurrence** $T(N) \sim \lg N$

- true for all N, as long as integer approx of N/2 is within a constant
- easy to prove when N is a power of 2.

can then use induction for general N
(see COS 341)



## Binary search recurrence: Proof by telescoping

$T(N) = T(N/2) + 1$
for $N > 1$, with $T(1) = 0$

(assume that N is a power of 2)

Pf.

| | |
|---|---|
| $T(N) = T(N/2) + 1$ | given |
| $= T(N/4) + 1 + 1$ | telescope (apply to first term) |
| $= T(N/8) + 1 + 1 + 1$ | telescope again |
| . . . | |
| $= T(N/N) + 1 + 1 + . . . + 1$ | stop telescoping, T(1) = 0 |
| $= \lg N$ | |

$T(N) = \lg N$



## Ordered array ST implementation (insert)

### Binary search is little help for `put()`: still need to move larger keys

```
public Val put(Key key, Val val)
 {
    int i = bsearch(key);
    if (i != -1)
    {  vals[i] = val; return;  }

    for ( i = N; i > 0; i-- )
    {
        if key.compareTo(keys[i-1] < 0) break;
        keys[i] = keys[i-1];
        vals[i] = vals[i-1];
    }
    vals[i] = val;
    keys[i] = key;
    N++;
 }
```

overwrite with new value if key in table

move larger keys to make room if key not in table

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| age | best | it | of | the | times | was | wisdom | worst |
| 2 | 1 | 4 | 4 | 4 | 2 | 4 | 1 | 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| age | best | foolish | it | of | the | times | was | wisdom | worst |
| 2 | 1 | 1 | 4 | 4 | 4 | 2 | 4 | 1 | 1 |

put("foolish")

## Ordered array ST implementation: an important special case

Test whether key is equal to or greater than largest key

```
public Val put(Key key, Val val)
{
   if (key.compareTo(keys[N-1]) == 0)
   {  vals[N-1] = val; return;  }

   if (key.compareTo(keys[N-1]) > 0)
   {
      vals[N] = val;
      keys[N] = key;
      N++;
      return;
   }
}
```

If either test succeeds, constant-time insert!

Method of choice for some clients:

- sort database by key
- insert N key-value pairs in order by key
- support searches that never use more than lg N compares
- support occasional (expensive) inserts



## Ordered linked-list ST implementation

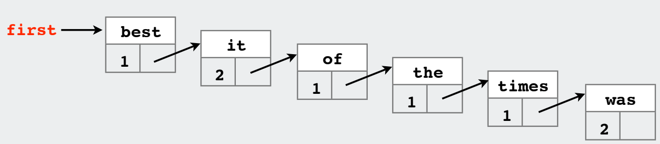
Binary search depends on array indexing for efficiency.

Jump to the middle of a linked list?

Advantages of keeping list in order for Comparable keys:

- support ordered iterator (for free)
- cuts search/insert time in half (on average) for random search/insert

[ code omitted ]






API
basic implementations
iterators
Comparable keys
**challenges**

## Searching challenge 1A:

Problem: maintain symbol table of song names for an iPod
Assumption A: hundreds of songs

Which searching method to use?

1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

## Searching challenge 1B:

Problem: maintain symbol table of song names for an iPod
Assumption B: thousands of songs

Which searching method to use?
1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough



## Searching challenge 2A:

Problem: IP lookups in a web monitoring device
Assumption A: billions of lookups, millions of distinct addresses

Which searching method to use?
1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough



## Searching challenge 2B:

Problem: IP lookups in a web monitoring device
Assumption B: billions of lookups, thousands of distinct addresses

Which searching method to use?
1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough



## Searching challenge 3:

Problem: Frequency counts in "Tale of Two Cities"

Assumptions: book has 135,000+ words
about 10,000 distinct words

Which searching method to use?
1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough

## Searching challenge 4:

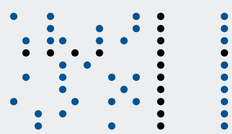
Problem: Spell checking for a book
Assumptions: dictionary has 25,000 words
book has 100,000+ words

Which searching method to use?
1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough



## Searching challenge 5:

Problem: Sparse matrix-vector multiplication
Assumptions: matrix dimension is billions by billions
average number of nonzero entries/row is ~10

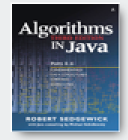


**A * x = b**

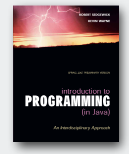
Which searching method to use?
1) unordered array
2) ordered linked list
3) ordered array with binary search
4) need better method, all too slow
5) doesn't matter much, all fast enough



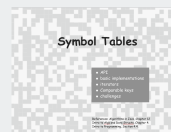
## Summary and roadmap




- basic algorithmics
- no generics
- more code
- more analysis
- equal keys in ST (not associative arrays)




- iterators
- ST as associative array (all keys distinct)
- BST implementations
- applications




- distinguish algs by operations on keys
- ST as associative array (all keys distinct)
- important special case for binary search
- challenges



## Elementary implementations: summary

Ordered array
- binary search reduces search time to lg N for `get()`
- need to move large elements for `put()`

Linked list
- need to scan whole list for `get()`
- need to scan whole list for `put()` to implement associative array
- can only save half (on average) by keeping in order

studying STs for the midterm? Start here.

| implementation | worst case search | worst case insert | average case search | average case insert | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|
| unordered array | N | N | N/2 | N/2 | no | `equals()` |
| ordered array | lg N | N | lg N | N/2 | yes | `Comparable` |
| unordered list | N | N | N/2 | N | no | `equals()` |
| ordered list | N | N | N/2 | N/2 | yes | `Comparable` |

Challenge.
Efficient implementations of `get()` and `put()` and ordered iteration.

(Ordered array meets challenge if keys arrive approximately in order)