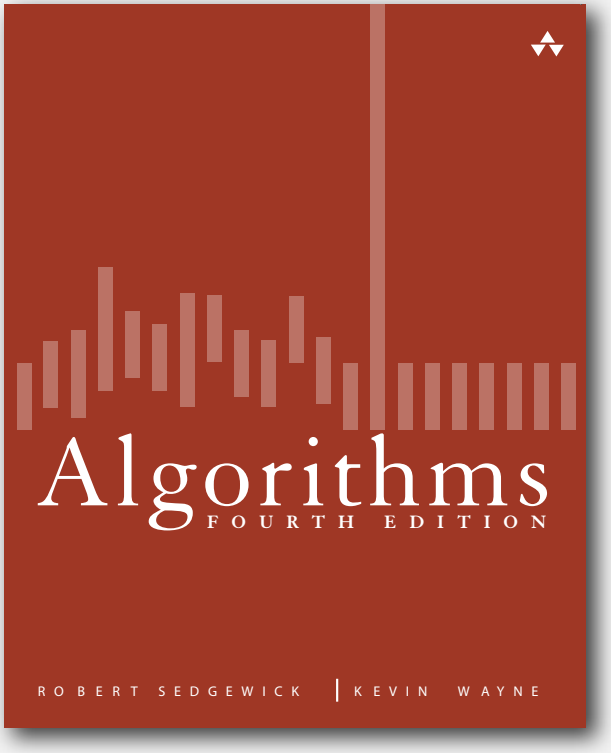
# 3.1 Symbol Tables




- API
- sequential search
- binary search
- ordered operations

- API
- sequential search
- binary search
- ordered operations

# Symbol tables

## Key-value pair abstraction.

- Insert a value with specified key.
- 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 |

# Symbol table applications

| application | purpose of search | key | value |
|---|---|---|---|
| dictionary | find definition | word | definition |
| book index | find relevant pages | term | list of page numbers |
| file share | find song to download | name of song | computer ID |
| financial account | process transactions | account number | transaction details |
| web search | find relevant web pages | keyword | list of page names |
| compiler | find properties of variables | variable name | type and value |
| routing table | route Internet packets | destination | best route |
| DNS | find IP address given URL | URL | IP address |
| reverse DNS | find URL given IP address | IP address | URL |
| genomics | find markers | DNA string | known positions |
| file system | find file on disk | filename | location on disk |

# Basic symbol table API

*Associative array abstraction.* Associate one value with each key.

| public class ST<Key, Value> | | |
|---|---|---|
| | ST() | *create a symbol table* |
| void | put(Key key, Value val) | *put key-value pair into the table* (*remove* key *from table if value is* null) |
| Value | get(Key key) | *value paired with* key (null *if* key *is absent*) |
| void | delete(Key key) | *remove* key *(and its value) from table* |
| boolean | contains(Key key) | *is there a value paired with* key*?* |
| boolean | isEmpty() | *is the table empty?* |
| int | size() | *number of key-value pairs in the table* |
| Iterable<Key> | keys() | *all the keys in the table* |

← `a[key] = val;`

← `a[key]`

## Conventions

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

## Intended consequences.

- Easy to implement `contains()`.

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

- Can implement lazy version of `delete()`.

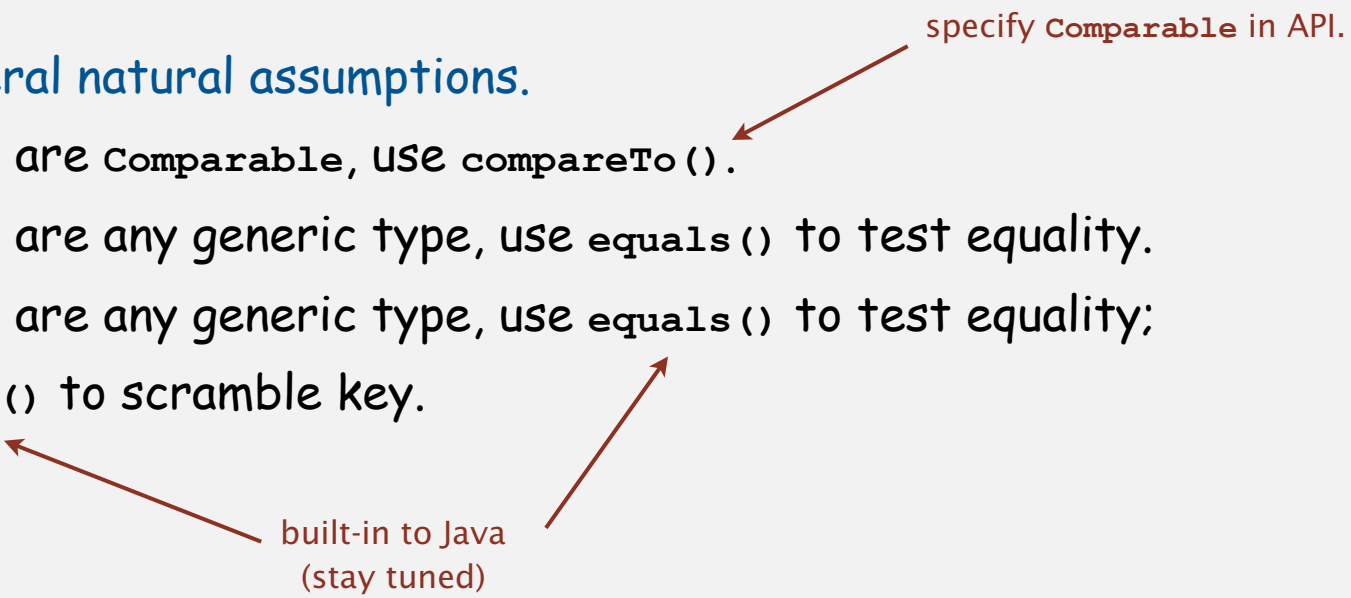
```
public void delete(Key key)
{  put(key, null);  }
```

# Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are `Comparable`, use `compareTo()`.
- Assume keys are any generic type, use `equals()` to test equality.
- Assume keys are any generic type, use `equals()` to test equality; use `hashCode()` to scramble key.




Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `Double`, `java.io.File`, ...
- Mutable in Java: `StringBuilder`, `java.net.URL`, arrays, ...

# Equality test

All Java classes inherit a method `equals()`.

Java requirements. 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`.

} equivalence relation

Default implementation. `(x == y)` — do `x` and `y` refer to the same object?

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, ...

User-defined implementations. Some care needed.

# Implementing equals for user-defined types

Seems easy.

```
public       class Date implements Comparable<Date>
{
   private final int month;
   private final int day;
   private final int year;
   ...

   public boolean equals(Date that)
   {









      if (this.day   != that.day  ) return false;
      if (this.month != that.month) return false;
      if (this.year  != that.year ) return false;
      return true;
   }
}
```

check that all significant fields are the same

# Implementing equals for user-defined types

Seems easy, but requires some care.

typically unsafe to use `equals()` with inheritance (would violate symmetry)

```
public final class Date implements Comparable<Date>
{
   private final int month;
   private final int day;
   private final int year;
   ...

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

      if (y == null) return false;

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

      Date that = (Date) y;
      if (this.day   != that.day  ) return false;
      if (this.month != that.month) return false;
      if (this.year  != that.year ) return false;
      return true;
   }
}
```

must be `Object`.
Why? Experts still debate.

optimize for true object equality

check for `null`

objects must be in the same class
(religion: `getClass()` vs. `instanceof`)

cast is guaranteed to succeed

check that all significant fields are the same

# Equals design

## "Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type and cast.
- Compare each significant field:
  - if field is a primitive type, use ==
  - if field is an object, use `equals()` ← apply rule recursively
  - if field is an array, apply to each entry ← alternatively, use `Arrays.equals(a, b)` or `Arrays.deepEquals(a, b)`, but not `a.equals(b)`

## Best practices.

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

`x.equals(y)` if and only if `(x.compareTo(y) == 0)`

# ST test client for traces

Build ST by associating value $i$ with $i^{th}$ string from standard input.

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

| keys | S | E | A | R | C | H | E | X | A | M | P | L | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| values | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**output**

```
A 8
C 4
E 12
H 5
L 11
M 9
P 10
R 3
S 0
X 7
```

# ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.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 FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

tiny example
(60 words, 20 distinct)

real example
(135,635 words, 10,769 distinct)

real example
(21,191,455 words, 534,580 distinct)

# Frequency counter implementation

```
public class FrequencyCounter
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);
      ST<String, Integer> st = new ST<String, Integer>();
      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (word.length() < minlen) continue;
         if (!st.contains(word)) st.put(word, 1);
         else                    st.put(word, st.get(word) + 1);
      }
      String max = "";
      st.put(max, 0);
      for (String word : st.keys())
         if (st.get(word) > st.get(max))
            max = word;
      StdOut.println(max + " " + st.get(max));
   }
}
```

create ST

ignore short strings

read string and update frequency

print a string with max freq

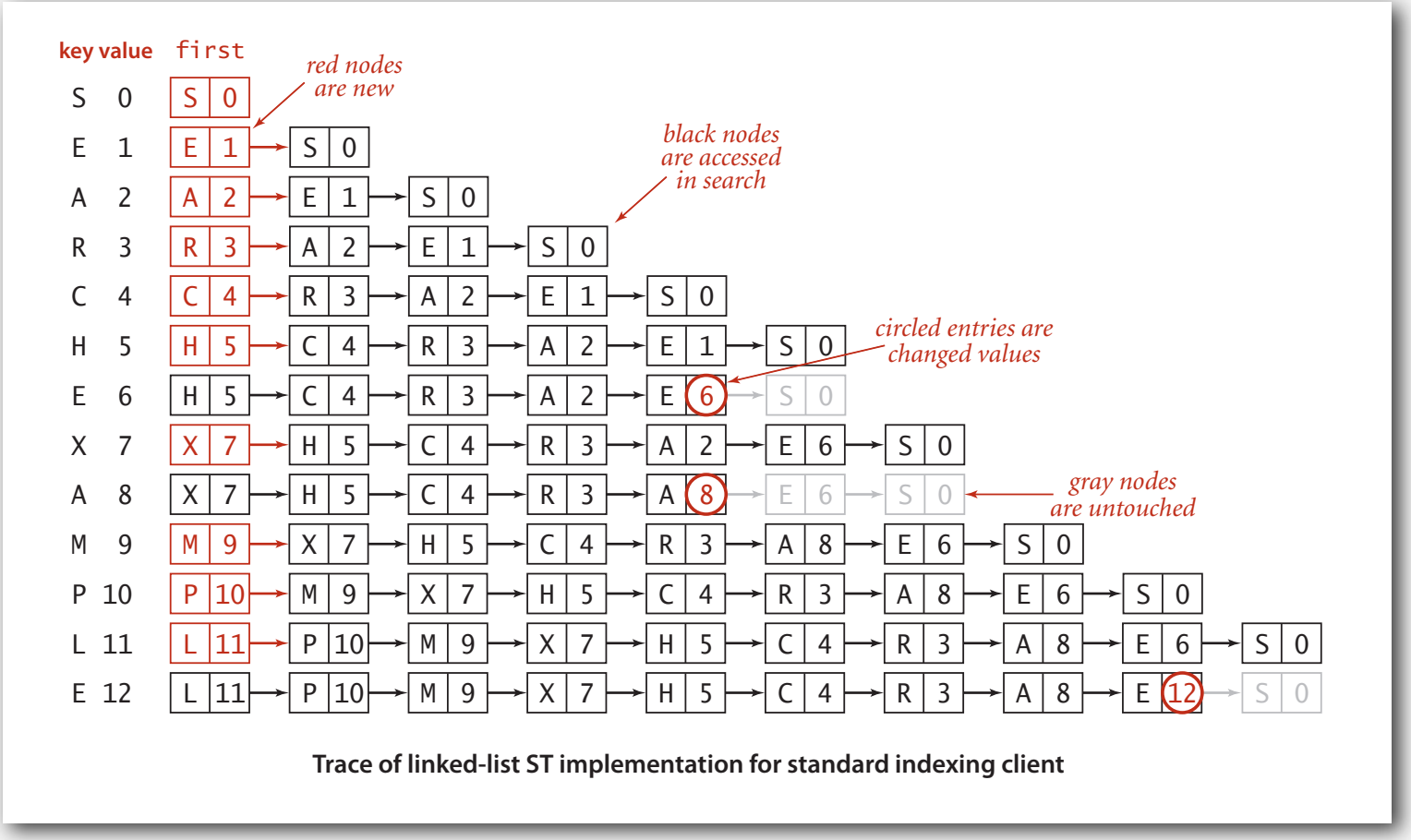
- API
- **sequential search**
- binary search
- ordered operations

# Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

Insert. Scan through all keys until find a match; if no match add to front.




Trace of linked-list ST implementation for standard indexing client

# Elementary ST implementations: summary

| ST implementation | worst-case cost (after N inserts) | | average case (after N random inserts) | | ordered iteration? | key interface |
|---|---|---|---|---|---|---|
| | search | insert | search hit | insert | | |
| sequential search (unordered list) | N | N | N / 2 | N | no | `equals()` |

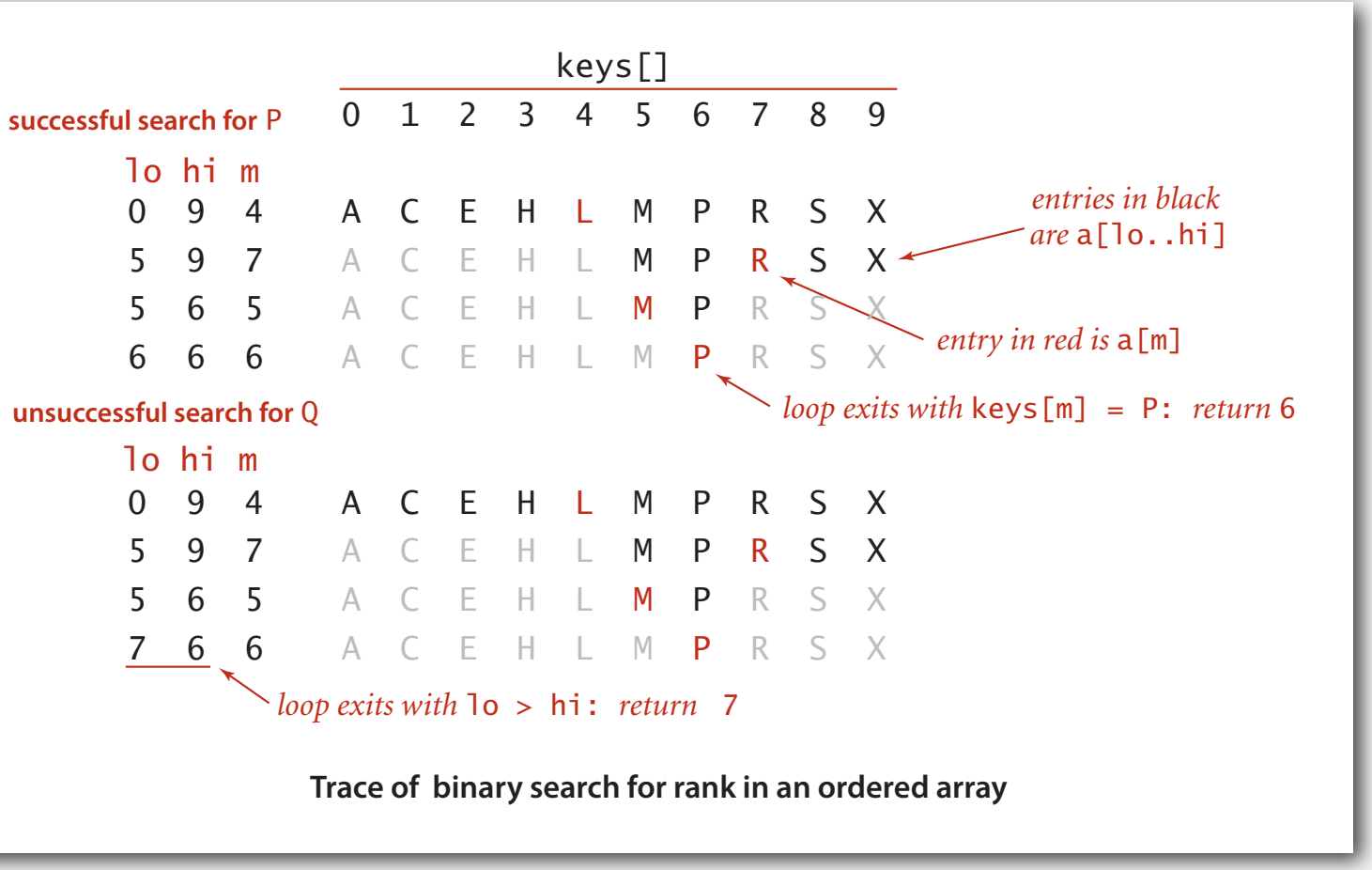
Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- **binary search**
- ordered symbol table ops

# Binary search

**Data structure.** Maintain an ordered array of key-value pairs.

**Rank helper function.** How many keys < $k$ ?




**Trace of binary search for rank in an ordered array**

# Binary search: Java implementation

```
public Value get(Key key)
{
   if (isEmpty()) return null;
   int i = rank(key);
   if (i < N && keys[i].compareTo(key) == 0) return vals[i];
   else  return null;
}
```

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

number of keys < `key`

## Binary search: mathematical analysis

Proposition. Binary search uses $\sim \lg N$ compares to search any array of size $N$.

Pf. $T(N) \equiv$ number of compares to binary search in a sorted array of size $N$.

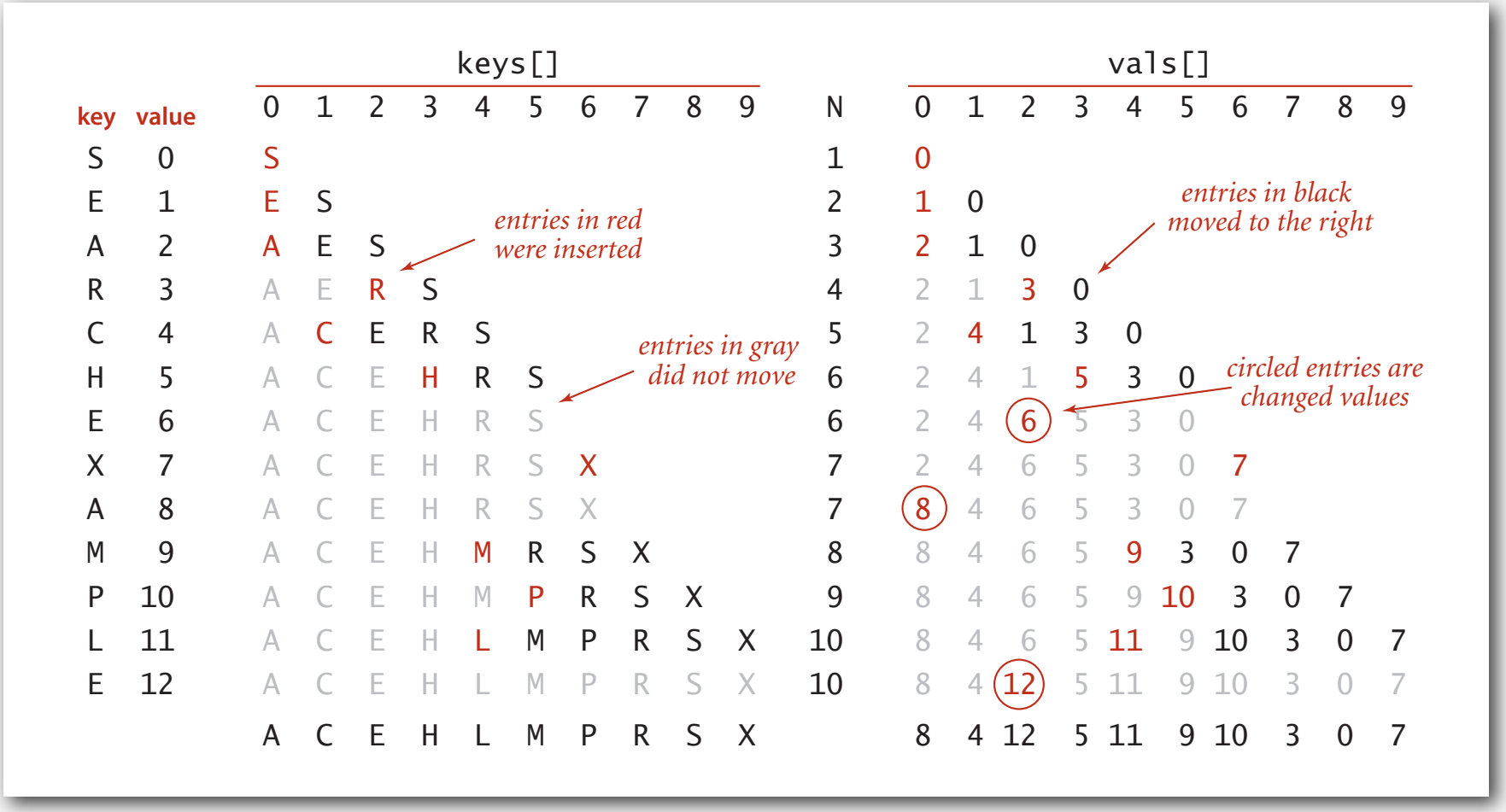
$$\leq T(\lfloor N / 2 \rfloor) + 1$$

↑ left or right half

Recall lecture 2.

# Binary search: trace of standard indexing client

**Problem.** To insert, need to shift all greater keys over.




| key | value | keys[0] | keys[1] | keys[2] | keys[3] | keys[4] | keys[5] | keys[6] | keys[7] | keys[8] | keys[9] | N | vals[0] | vals[1] | vals[2] | vals[3] | vals[4] | vals[5] | vals[6] | vals[7] | vals[8] | vals[9] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | 0 | S | | | | | | | | | | 1 | 0 | | | | | | | | | |
| E | 1 | E | S | | | | | | | | | 2 | 1 | 0 | | | | | | | | |
| A | 2 | A | E | S | | | | | | | | 3 | 2 | 1 | 0 | | | | | | | |
| R | 3 | A | E | R | S | | | | | | | 4 | 2 | 1 | 3 | 0 | | | | | | |
| C | 4 | A | C | E | R | S | | | | | | 5 | 2 | 4 | 1 | 3 | 0 | | | | | |
| H | 5 | A | C | E | H | R | S | | | | | 6 | 2 | 4 | 1 | 5 | 3 | 0 | | | | |
| E | 6 | A | C | E | H | R | S | | | | | 6 | 2 | 4 | 6 | 5 | 3 | 0 | | | | |
| X | 7 | A | C | E | H | R | S | X | | | | 7 | 2 | 4 | 6 | 5 | 3 | 0 | 7 | | | |
| A | 8 | A | C | E | H | R | S | X | | | | 7 | 8 | 4 | 6 | 5 | 3 | 0 | 7 | | | |
| M | 9 | A | C | E | H | M | R | S | X | | | 8 | 8 | 4 | 6 | 5 | 9 | 3 | 0 | 7 | | |
| P | 10 | A | C | E | H | M | P | R | S | X | | 9 | 8 | 4 | 6 | 5 | 9 | 10 | 3 | 0 | 7 | |
| L | 11 | A | C | E | H | L | M | P | R | S | X | 10 | 8 | 4 | 6 | 5 | 11 | 9 | 10 | 3 | 0 | 7 |
| E | 12 | A | C | E | H | L | M | P | R | S | X | 10 | 8 | 4 | 12 | 5 | 11 | 9 | 10 | 3 | 0 | 7 |
| | | A | C | E | H | L | M | P | R | S | X | | 8 | 4 | 12 | 5 | 11 | 9 | 10 | 3 | 0 | 7 |

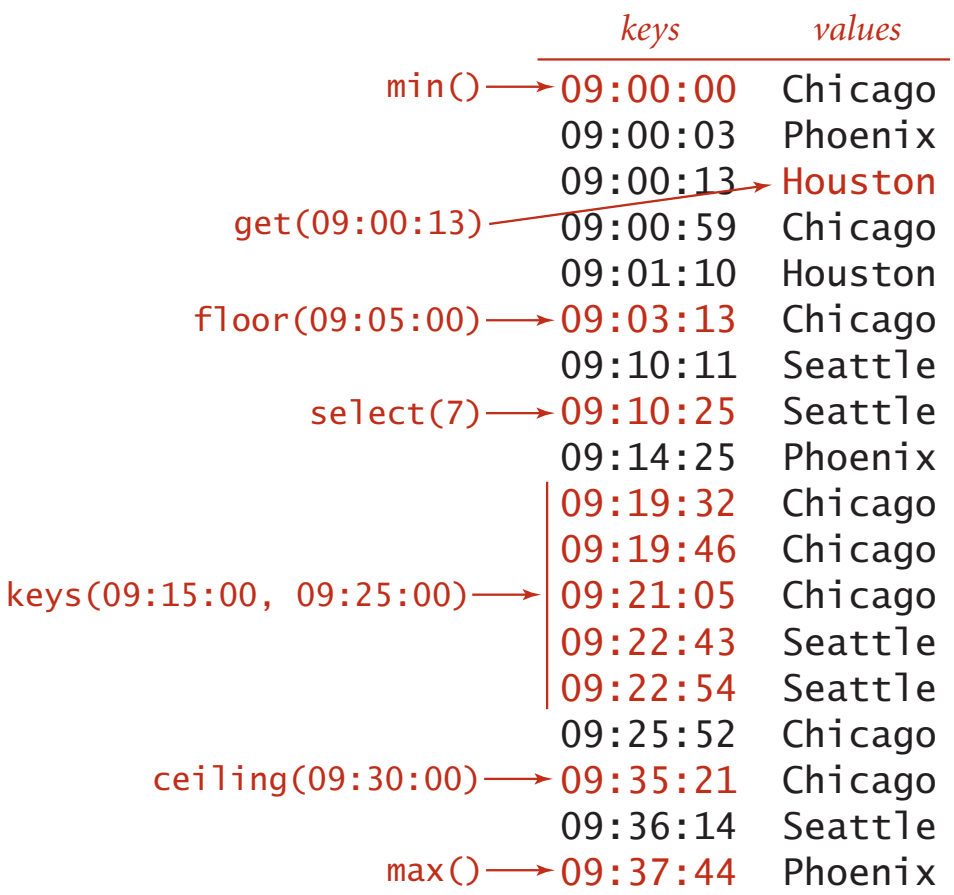
# Elementary ST implementations: summary

| ST implementation | worst-case cost (after N inserts) | | average case (after N random inserts) | | ordered iteration? | key interface |
|---|---|---|---|---|---|---|
| | search | insert | search hit | insert | | |
| sequential search (unordered list) | N | N | N / 2 | N | no | `equals()` |
| binary search (ordered array) | log N | N | log N | N / 2 | yes | `compareTo()` |

Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- binary search
- **ordered operations**

# Ordered symbol table API




**Examples of ordered symbol-table operations**

# Ordered symbol table API

| public class ST<Key extends Comparable<Key>, Value> | | |
|---|---|---|
| | ST() | *create an ordered symbol table* |
| void | put(Key key, Value val) | *put key-value pair into the table*<br>*(remove* key *from table if value is* null*)* |
| Value | get(Key key) | *value paired with* key<br>*(*null *if* key *is absent)* |
| void | delete(Key key) | *remove* key *(and its value) from table* |
| boolean | contains(Key key) | *is there a value paired with* key*?* |
| boolean | isEmpty() | *is the table empty?* |
| int | size() | *number of key-value pairs* |
| Key | min() | *smallest key* |
| Key | max() | *largest key* |
| Key | floor(Key key) | *largest key less than or equal to* key |
| Key | ceiling(Key key) | *smallest key greater than or equal to* key |
| int | rank(Key key) | *number of keys less than* key |
| Key | select(int k) | *key of rank* k |
| void | deleteMin() | *delete smallest key* |
| void | deleteMax() | *delete largest key* |
| int | size(Key lo, Key hi) | *number of keys in* [lo..hi] |
| Iterable<Key> | keys(Key lo, Key hi) | *keys in* [lo..hi]*, in sorted order* |
| Iterable<Key> | keys() | *all keys in the table, in sorted order* |

# Binary search: ordered symbol table operations summary

| | sequential search | binary search |
|---|---|---|
| search | N | lg N |
| insert | 1 | N |
| min / max | N | 1 |
| floor / ceiling | N | lg N |
| rank | N | lg N |
| select | N | 1 |
| ordered iteration | N log N | N |

**order of growth of the running time for ordered symbol table operations**