# 3.1 Symbol Tables


- 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 |

key → URL; value → IP address



## 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 |



## 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)* ← `a[key] = val;` |
| Value get(Key key) | *value paired with* key *(*null *if* key *is absent)* ← `a[key]` |
| 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* |

**API for a generic basic symbol table**

## 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.

specify `Comparable` in API.

- 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 and `hashCode()` to scramble key.

built-in to Java (stay tuned)

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `Double`, `File`, ...
- Mutable in Java: `Date`, `StringBuilder`, `Url`, ...



## 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

do `x` and `y` refer to the same object?

Default implementation. `(x == y)`

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

User-defined implementations. Some care needed.



## Implementing equals for user-defined types

Seems easy

```
public       class Record
{
   private final String name;
   private final long val;
   private final int id;
   ...

   public boolean equals(Record y)
   {

      Record that =          y;
      return (this.val == that.val) &&
             (this.id  == that.id)  &&
             (this.name.equals(that.name));
   }
}
```

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 Record
{
   private final String name;
   private final long val;
   private final int id;
   ...

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

      if (y == null) return false;

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

      Record that = (Record) y;
      return (this.val == that.val) &&
             (this.id  == that.id)  &&
             (this.name.equals(that.name));
   }
}
```

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`)

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()`
  - if field is a primitive array, apply to each element

apply rule recursively

or use `Arrays.deepEquals()`

Best practices.

- Compare fields mostly likely to differ first.
- No need to use calculated fields that depend on other fields.



## 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>();
   String[] a = StdIn.readAll().split("\\s+");
   for (int i = 0; i < a.length; i++)
      st.put(a[i], 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 9
M 11
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





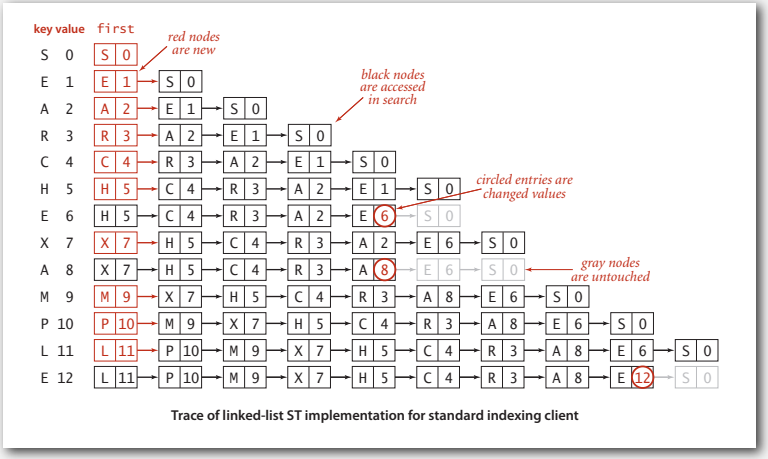


## Sequential search in a linked list

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.

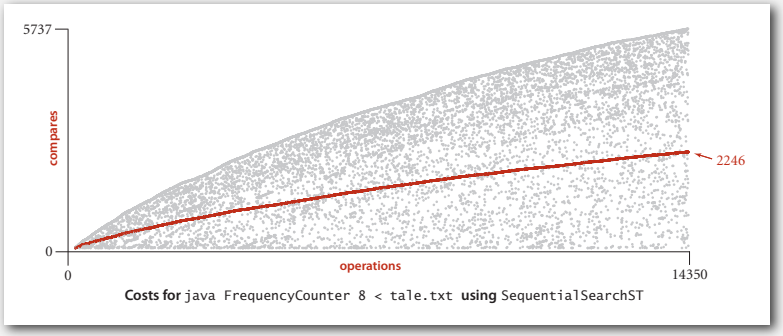



Trace of linked-list ST implementation for standard indexing client



## Elementary ST implementations: summary

| ST implementation | worst case | | average case | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|
| | search | insert | search hit | insert | | |
| sequential search (unordered list) | N | N | N / 2 | N | no | `equals()` |




Costs for `java FrequencyCounter 8 < tale.txt` using `SequentialSearchST`

Challenge. Efficient implementations of both search and insert.

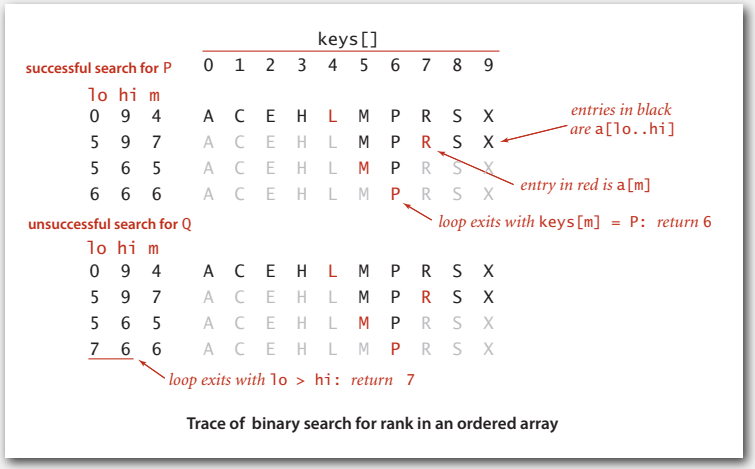
- API
- sequential search
- **binary search**
- ordered symbol table ops



## 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)                          number of keys < 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;
}
```



## Binary search: mathematical analysis

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

Pf. $T(N)$ ≡ 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 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | N | vals[] 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 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 |

*entries in red were inserted*

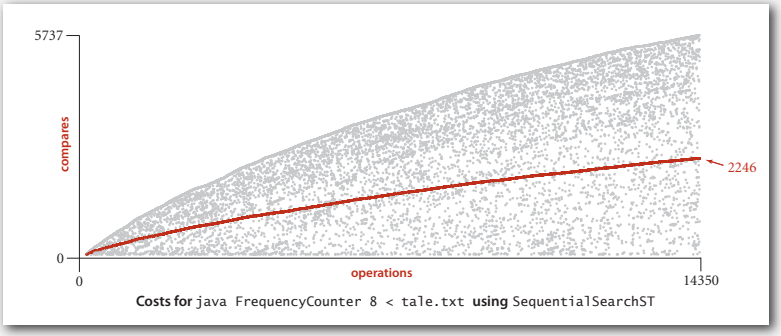
*entries in gray did not move*

*entries in black moved to the right*

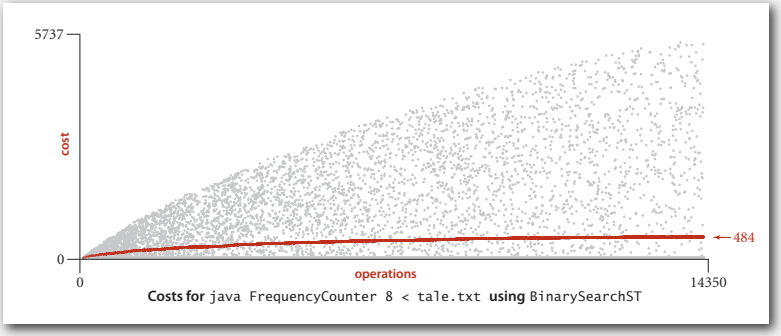
*circled entries are changed values*



## Elementary ST implementations: frequency counter




Costs for `java FrequencyCounter 8 < tale.txt` using `SequentialSearchST`




Costs for `java FrequencyCounter 8 < tale.txt` using `BinarySearchST`



## Elementary ST implementations: summary

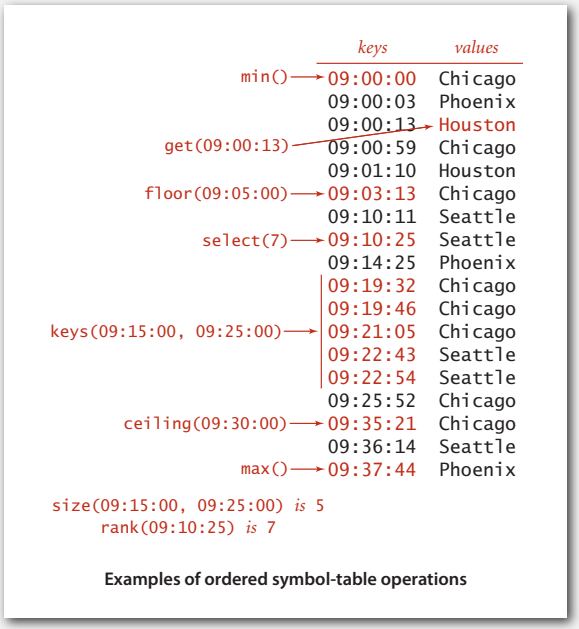
| ST implementation | worst case | | average case | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|
| | search | insert | search hit | insert | | |
| sequential search (unordered list) | N | N | N / 2 | N | no | `equals()` |
| binary search (ordered array) | log N | N | log 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 (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* |
| `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* |

**API for a generic ordered symbol table**



## 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 |

**worst-case running time of ordered symbol table operations**



# 3.2 Binary Search Trees

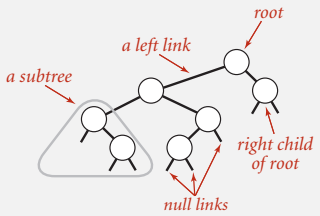

- BSTs
- ordered operations
- deletion

## Binary search trees

Definition. A BST is a binary tree in symmetric order.

A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).

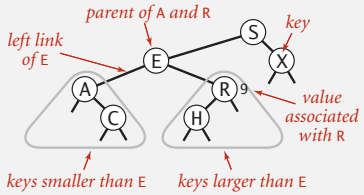



Anatomy of a binary tree

Symmetric order.

Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.




Anatomy of a binary search tree



## BST representation in Java

Java definition. A BST is a reference to a root `Node`.

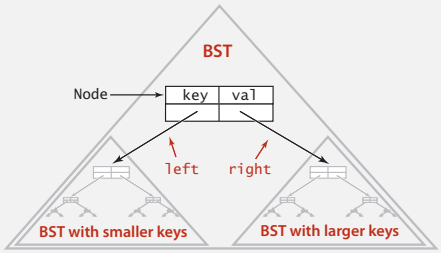
A `Node` is comprised of four fields:

- A `Key` and a `Value`.
- A reference to the left and right subtree.

smaller keys larger keys

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

`Key` and `Value` are generic types; `Key` is `Comparable`




Binary search tree



## BST implementation (skeleton)

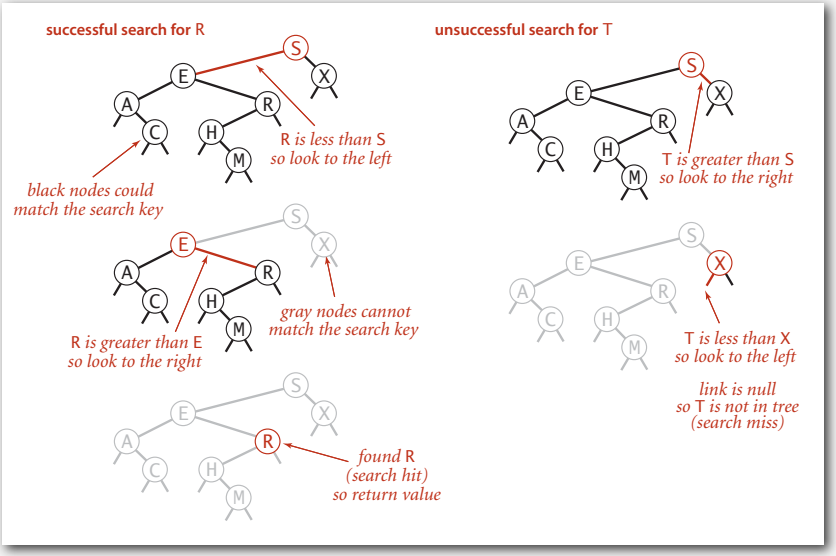
```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

   private class Node
   {  /* see previous slide */  }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

root of BST



## BST search

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



## BST search: Java implementation

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

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

Cost. Number of compares is equal to depth of node.

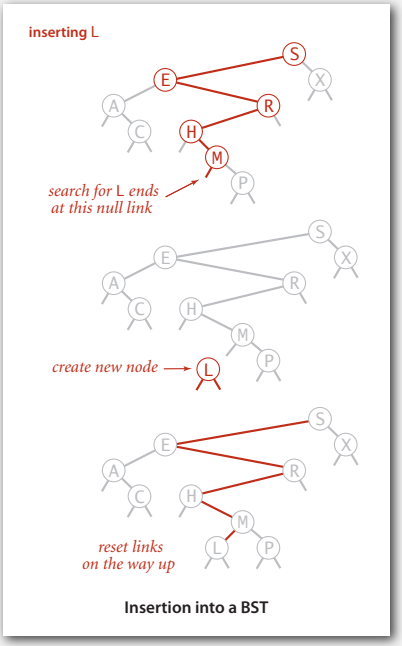


## BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.




Insertion into a BST



## BST insert: Java implementation

Put. Associate value with key.

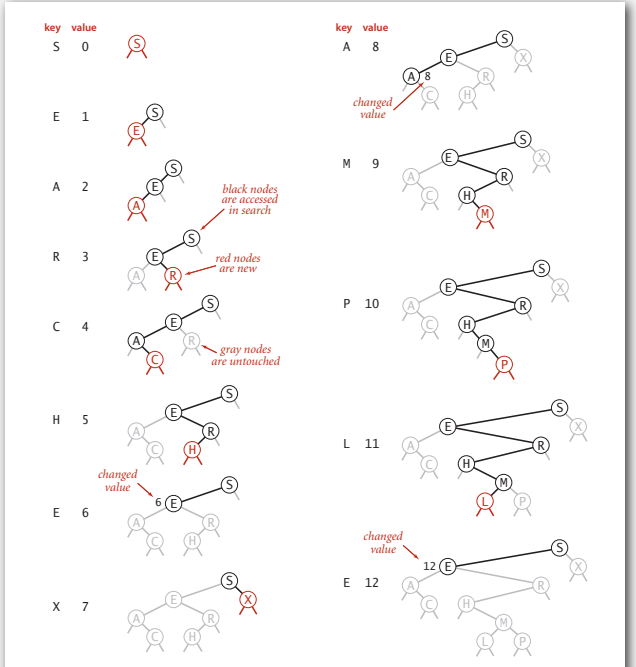
concise, but tricky, recursive code; read carefully!

```
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 if (cmp == 0)
      x.val = val;
   return x;
}
```

Cost. Number of compares is equal to depth of node.

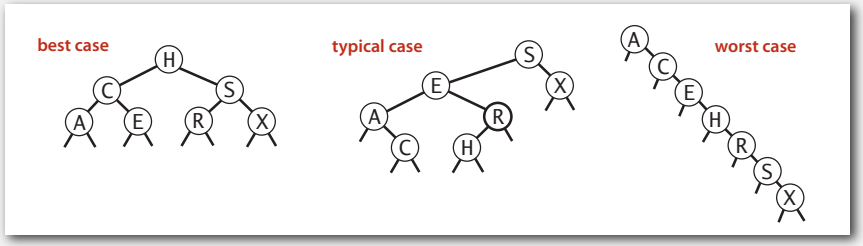


## BST trace: standard indexing client



## Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to depth of node.

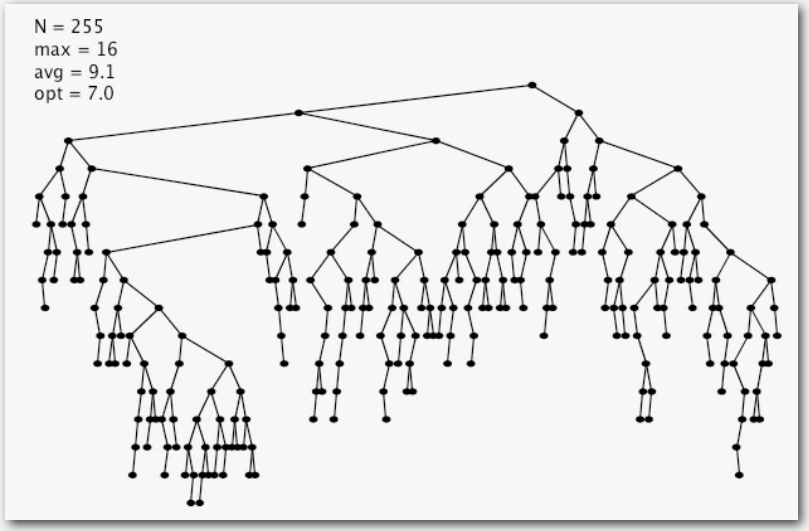



Remark. Tree shape depends on order of insertion.



## BST insertion: random order

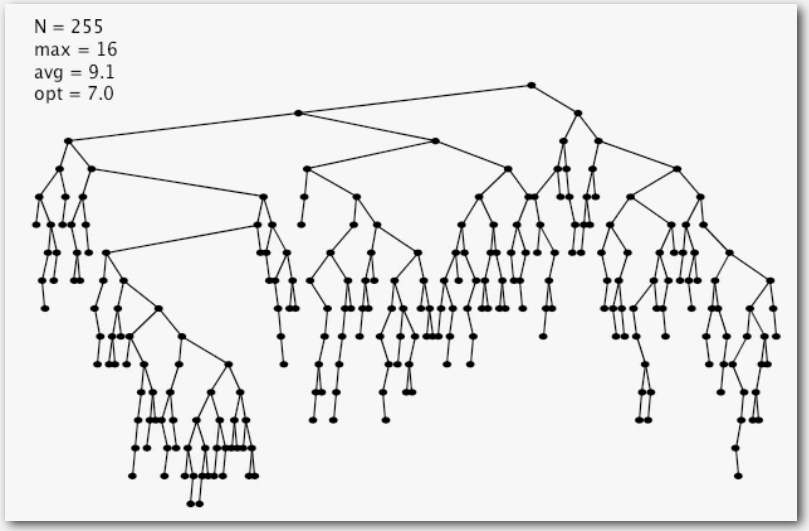
Observation. If keys inserted in random order, tree stays relatively flat.





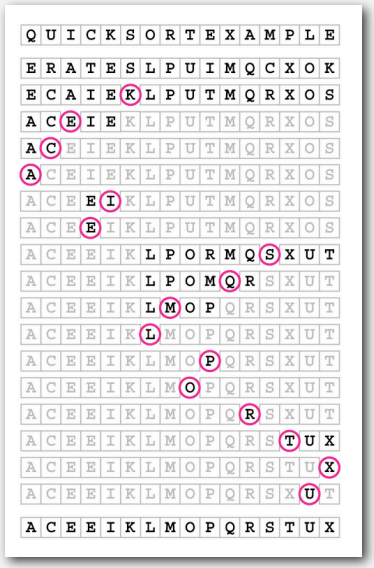

## BST insertion: random order visualization

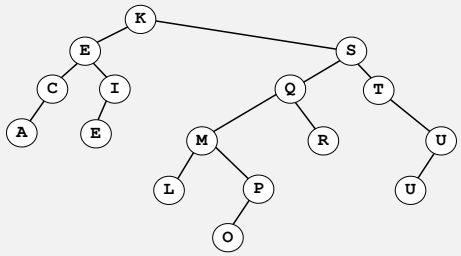
Ex. Insert keys in random order.






## Correspondence between BSTs and quicksort partitioning





Remark. Correspondence is 1-1 if array has no duplicate keys.

## BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

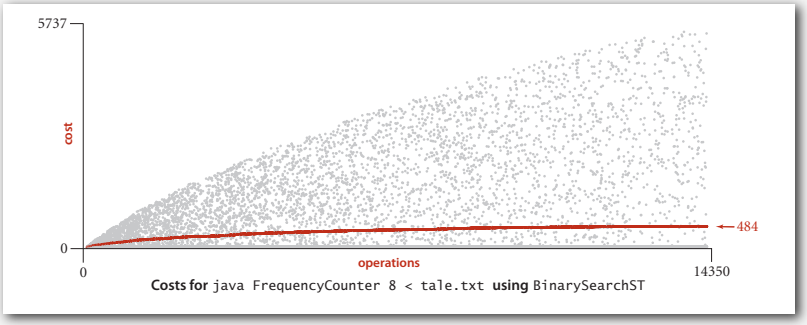
Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

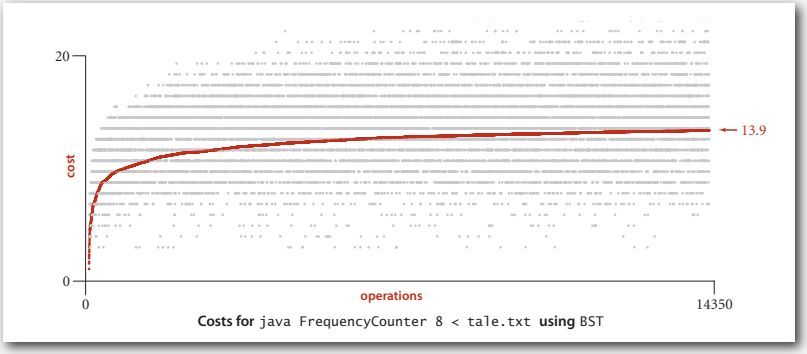
But... Worst-case height is $N$.
(exponentially small chance when keys are inserted in random order)



## ST implementations: frequency counter



Costs for `java FrequencyCounter 8 < tale.txt` using `BinarySearchST`



Costs for `java FrequencyCounter 8 < tale.txt` using `BST`



## ST implementations: summary

| implementation | guarantee | | average case | | ordered ops? | operations on keys |
|---|---|---|---|---|---|---|
| | search | insert | search hit | insert | | |
| sequential search (unordered list) | N | N | N/2 | N | no | `equals()` |
| binary search (ordered array) | lg N | N | lg N | N/2 | yes | `compareTo()` |
| BST | N | N | 1.39 lg N | 1.39 lg N | ? | `compareTo()` |

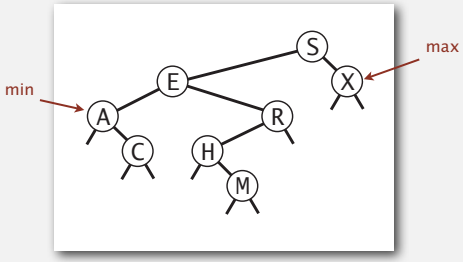


- BSTs
- **ordered operations**
- deletion

## Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.



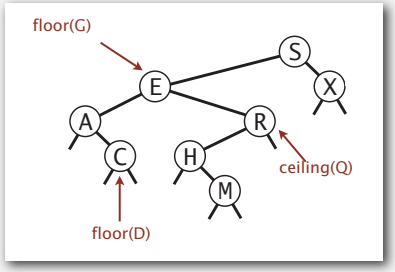

Q. How to find the min / max?



## Floor and ceiling

Floor. Largest key ≤ to a given key.

Ceiling. Smallest key ≥ to a given key.




Q. How to find the floor /ceiling?



## Computing the floor

Case 1. [$k$ equals the key at root]

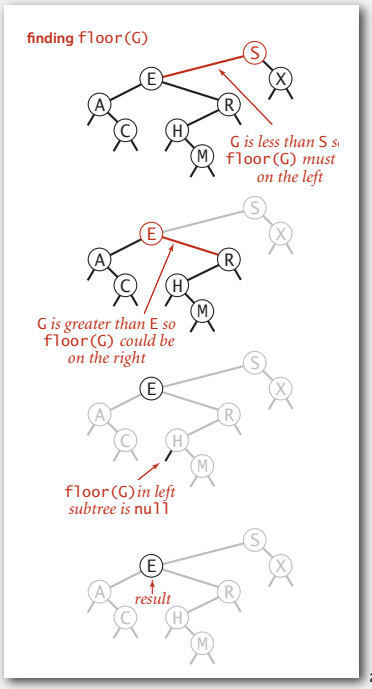
The floor of $k$ is $k$.

Case 2. [$k$ is less than the key at root]

The floor of $k$ is in the left subtree.

Case 3. [$k$ is greater than the key at root]

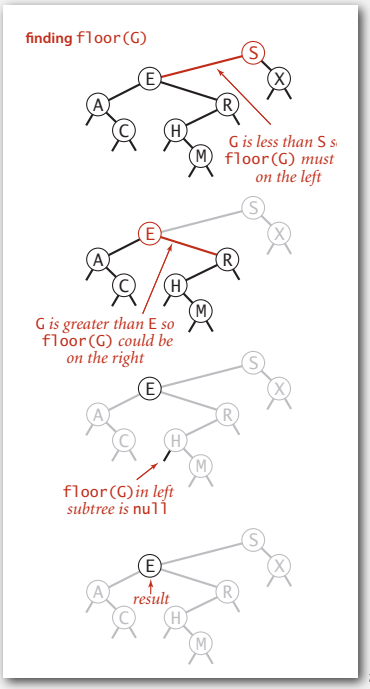
The floor of $k$ is in the right subtree (if there is any key ≤ $k$ in right subtree); otherwise it is the key in the root.





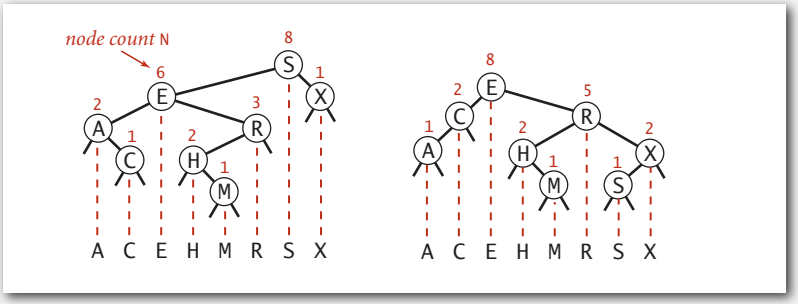

## Computing the floor

```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```



## Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node.
To implement `size()`, return the count at the root.




Remark. This facilitates efficient implementation of `rank()` and `select()`.



## BST implementation: subtree counts

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

nodes in subtree

```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;
   return x.N;
}
```

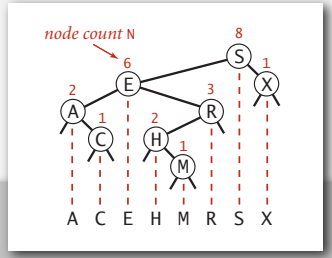
```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```



## Rank

Rank. How many keys $< k$ ?

Easy recursive algorithm (4 cases!)



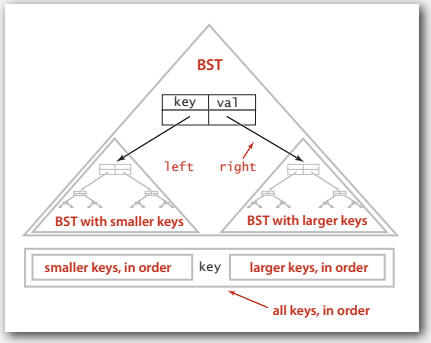

```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return rank(key, x.left);
   else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
   else              return size(x.left);
}
```



## Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

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

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




Property. Inorder traversal of a BST yields keys in ascending order.

## Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
inorder(S)
  inorder(E)
    inorder(A)
      enqueue A
      inorder(C)
        enqueue C
    enqueue E
    inorder(R)
      inorder(H)
        enqueue H
        inorder(M)
          enqueue M
      print R
  enqueue S
  inorder(X)
    enqueue X
```

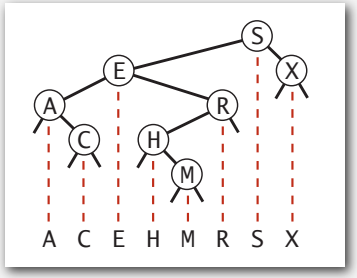
recursive calls

```
A
C
E
H
M
R
S
X
```

queue

```
S
S E
S E A
S E A C
S E R
S E R H
S E R H M
S X
```

function call stack






## BST: ordered symbol table operations summary

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

h = height of BST (proportional to log N if keys inserted in random order)

**worst-case running time of ordered symbol table operations**



- BSTs
- ordered operations
- **deletion**



## ST implementations: summary

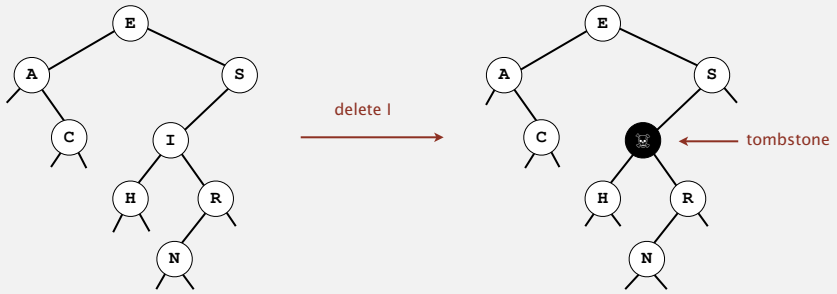
| implementation | guarantee | | | average case | | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (linked list) | N | N | N | N/2 | N | N/2 | no | `equals()` |
| binary search (ordered array) | lg N | N | N | lg N | N/2 | N/2 | yes | `compareTo()` |
| BST | N | N | N | 1.39 lg N | 1.39 lg N | ? ? ? | yes | `compareTo()` |

Next. Deletion in BSTs.

## BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide searches (but don't consider it equal to search key).




Cost. $2 \ln N'$ per insert, search, and delete (if keys in random order), where $N'$ is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone overload.

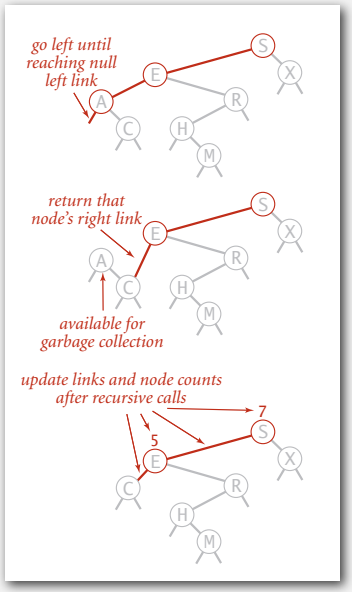


## Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```

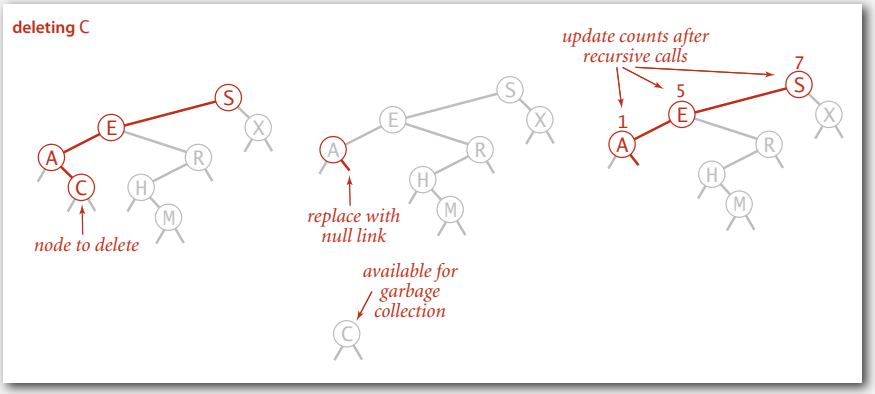





## Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

Case 0. [0 children] Delete $t$ by setting parent link to null.

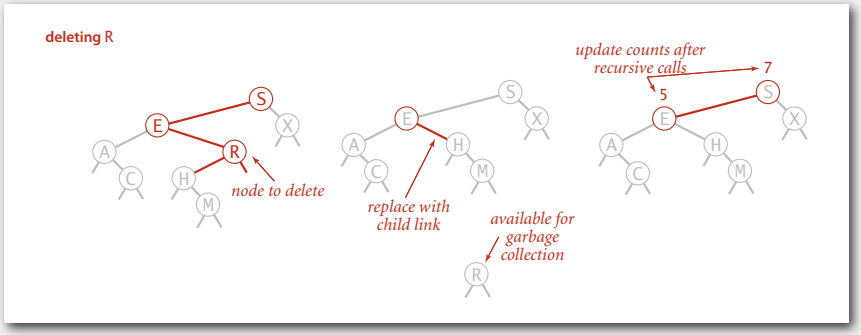





## Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

Case 1. [1 child] Delete $t$ by replacing parent link.

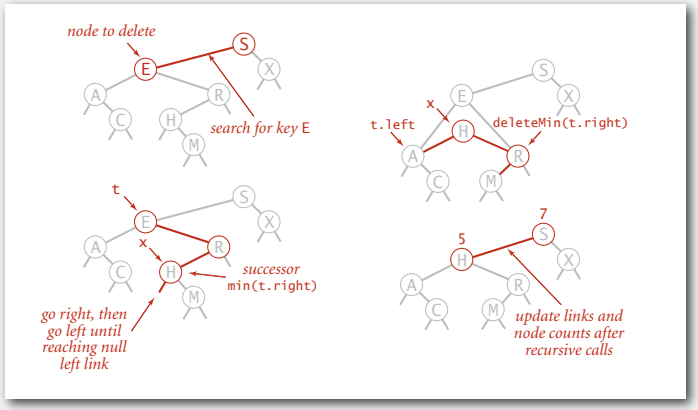


## Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

Case 2. [2 children]

- Find successor $x$ of $t$. ← x has no left child
- Delete the minimum in $t$'s right subtree. ← but don't garbage collect x
- Put $x$ in $t$'s spot. ← still a BST






## Hibbard deletion: Java implementation

```
public void delete(Key key)
{  root = delete(root, key);  }

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

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

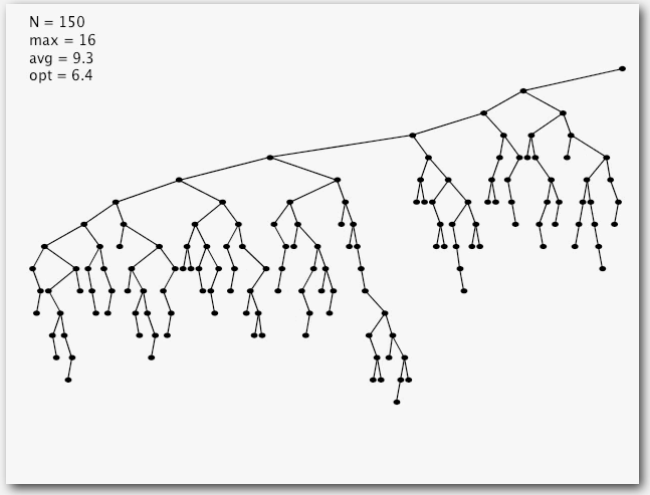
replace with successor

update subtree counts



## Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.




Surprising consequence. Trees not random (!) ⇒ sqrt ($N$) per op.

Longstanding open problem. Simple and efficient delete for BSTs.



## ST implementations: summary

| implementation | guarantee | | | average case | | | ordered iteration? | operations on keys |
|---|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (linked list) | N | N | N | N/2 | N | N/2 | no | `equals()` |
| binary search (ordered array) | lg N | N | N | lg N | N/2 | N/2 | yes | `compareTo()` |
| BST | N | N | N | 1.39 lg N | 1.39 lg N | √N | yes | `compareTo()` |

other operations also become √N if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.