### 3.1 Symbol Tables

- **API**
- **elementary implementations**
- **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 domain name with specified IP address.
- Given domain name, find corresponding IP address.

<table>
<thead>
<tr>
<th>domain name</th>
<th>IP address</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cs.princeton.edu">www.cs.princeton.edu</a></td>
<td>128.112.136.11</td>
</tr>
<tr>
<td><a href="http://www.princeton.edu">www.princeton.edu</a></td>
<td>128.112.128.15</td>
</tr>
<tr>
<td><a href="http://www.yale.edu">www.yale.edu</a></td>
<td>130.132.143.21</td>
</tr>
<tr>
<td><a href="http://www.harvard.edu">www.harvard.edu</a></td>
<td>128.103.060.55</td>
</tr>
<tr>
<td><a href="http://www.simpsons.com">www.simpsons.com</a></td>
<td>209.052.165.60</td>
</tr>
</tbody>
</table>

*“Smart data structures and dumb code works a lot better than the other way around.”* —Eric S. Raymond
Symbol table applications

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

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and \( N - 1 \).

Language support.
- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```java
public class ST<Key, Value>

ST() 
    create an empty symbol table
void put(Key key, Value val) 
    put key-value pair into the table
Value get(Key key) 
    value paired with key
boolean contains(Key key) 
    is there a value paired with key?
void delete(Key key) 
    remove key (and its value) from table
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
```

Conventions

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

Intended consequences.
- Easy to implement contains().

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

- Can implement lazy version of delete().

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

Implementing equals for user-defined types

Seems easy.

```
public final class Date implements Comparable<Date> {
    private final int month;
    private final int day;
    private final int year;
    ...
    public boolean equals(Date that) {
        if (this.month != that.month) return false;
        if (this.day != that.day) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

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.

Default implementation. (x == y)
Customized implementations. Integer, Double, String, java.io.File, ...
User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy, but requires some care.

```
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 == null) return false;
        if (y.getClass() != this.getClass()) return false;
        if (y == this) return true;
        if (y.getClass() == this.getClass())
            if (this.day != that.day) return false;
            if (this.month != that.month) return false;
            if (this.year != that.year) return false;
            return true;
    }
}
```
ST test client for traces

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

```java
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));
    }
}
```

<table>
<thead>
<tr>
<th>keys</th>
<th>S</th>
<th>E</th>
<th>A</th>
<th>R</th>
<th>C</th>
<th>H</th>
<th>X</th>
<th>A</th>
<th>M</th>
<th>P</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

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
```

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));
    }
}
```
3.1 SYMBOL TABLES

- API
- elementary implementations
- ordered operations

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.

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

Binary search in an ordered array

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

Rank helper function. How many keys < k?

Successful search for P

<table>
<thead>
<tr>
<th>keys[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C E H L M P R S X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

entries in black
are a[lo..hi]
entry in red is a[m]
loop exits with keys[m] = P: return 6

Unsuccessful search for Q

<table>
<thead>
<tr>
<th>keys[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C E H L M P R S X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

loop exit with lo > hi: return 7
Binary search: Java implementation

```java
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;
}
```

Elementary symbol tables: quiz 1
Implementing binary search was

A. Easier than I thought.
B. About what I expected.
C. Harder than I thought.
D. Much harder than I thought.
E. I don't know. (Well, you should!)

Problem. Given an array with all 0s in the beginning and all 1s at the end, find the index in the array where the 1s start.

**Variant 1.** You are given the length of the array.

**Variant 2.** You are not given the length of the array.

**Find the First 1**

**Problem.** Given an array with all 0s in the beginning and all 1s at the end, find the index in the array where the 1s start.

**Input**

```
0 0 0 0 0 ... 0 0 0 0 0 1 1 1 ... 1 1 1
```

**Binary search: trace of standard indexing client**

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

```
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>R</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
</tr>
<tr>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
</tr>
<tr>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 N</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>R</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
</tr>
<tr>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
</tr>
<tr>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
</tr>
</tbody>
</table>
```

Entries in red were inserted.
Entries in black moved to the right.
Entries in gray did not move.
Entries in black were changed.
Circled entries are changed values.
**Elementary ST implementations: summary**

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td>equals()</td>
</tr>
<tr>
<td>binary search</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

**Challenge.** Efficient implementations of both search and insert.

### 3.1 Symbol Tables

**Ordered symbol table API**

```java
public class ST<Key extends Comparable<Key>, Value>

Method                  Description
----------------------------------------------------------------
min()                   smallest key
max()                   largest key
floor(Key key)          largest key less than or equal to key
ceiling(Key key)        smallest key greater than or equal to key
rank(Key key)           number of keys less than key
select(int k)           key of rank k
deleteMin()             delete smallest key
deleteMax()             delete largest key
size(Key lo, Key hi)    number of keys between lo and hi
keys()                  all keys, in sorted order
keys(Key lo, Key hi)    keys between lo and hi, in sorted order
```

**Examples of ordered symbol table API**

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>min()</td>
<td>09:00:00 Chicago</td>
</tr>
<tr>
<td>09:00:03 Phoenix</td>
<td></td>
</tr>
<tr>
<td>09:00:13 Houston</td>
<td></td>
</tr>
<tr>
<td>get(09:00:13)</td>
<td>09:00:59 Chicago</td>
</tr>
<tr>
<td>09:01:10 Houston</td>
<td></td>
</tr>
<tr>
<td>floor(09:05:00)</td>
<td>09:03:13 Chicago</td>
</tr>
<tr>
<td>09:10:11 Seattle</td>
<td></td>
</tr>
<tr>
<td>select(7)</td>
<td>09:10:25 Seattle</td>
</tr>
<tr>
<td>09:14:25 Phoenix</td>
<td></td>
</tr>
<tr>
<td>09:19:32 Chicago</td>
<td></td>
</tr>
<tr>
<td>09:19:46 Chicago</td>
<td></td>
</tr>
<tr>
<td>keys(09:15:00, 09:25:00)</td>
<td>09:21:05 Chicago</td>
</tr>
<tr>
<td>09:22:43 Seattle</td>
<td></td>
</tr>
<tr>
<td>09:22:54 Seattle</td>
<td></td>
</tr>
<tr>
<td>09:25:52 Chicago</td>
<td></td>
</tr>
<tr>
<td>ceiling(09:30:00)</td>
<td>09:35:21 Chicago</td>
</tr>
<tr>
<td>09:36:14 Seattle</td>
<td></td>
</tr>
<tr>
<td>max()</td>
<td>09:37:44 Phoenix</td>
</tr>
<tr>
<td>size(09:15:00, 09:25:00)</td>
<td>5 keys</td>
</tr>
<tr>
<td>rank(09:10:25)</td>
<td>7</td>
</tr>
</tbody>
</table>
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion

## Binary search: ordered symbol table operations summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Insert</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Min / Max</td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td>Floor / Ceiling</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Rank</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Select</td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td>Ordered Iteration</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

Order of growth of the running time for ordered symbol table operations

### 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).

**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.
**Binary search tree demo**

**Search.** If less, go left; if greater, go right; if equal, search hit.

successful search for H

![Binary search tree](image)

**Insert.** If less, go left; if greater, go right; if null, insert.

insert G

![Binary search tree](image)

**BST representation in Java**

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

A Node is composed of four fields:
- A Key and a Value.
- A reference to the left and right subtree.

A Node is composed of four fields:
- A Key and a Value.
- A reference to the left and right subtree.

```java
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;
    }
}
```

### BST implementation (skeleton)

```java
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 */
    }
}
```

Key and Value are generic types; Key is Comparable.
BST search: Java implementation

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

```java
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 = 1 + depth of node.

BST insert: Java implementation

Put. Associate value with key.

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

```java
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 = 1 + 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.

Cost. Number of compares = 1 + depth of node.

Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.

Bottom line. Tree shape depends on order of insertion.
Inorder traversal

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

```java
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    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.

---

Binary search trees: quiz 1

In what order does the `traverse(root)` code print out the keys in the BST?

```java
private void traverse(Node x)
{
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key);
    traverse(x.right);
}
```

A. A C E H M R S X
B. A C E R H M X S
C. S E A C R H M X
D. C A M H R E X S
E. I don't know.

---

Binary search trees: quiz 2

What is the name of this sorting algorithm?

1. **Shuffle** the keys.
2. **Insert** the keys into a BST, one at a time.
3. **Do an inorder traversal** of the BST.

A. Insertion sort.
B. Mergesort.
C. Quicksort.
D. None of the above.
E. I don't know.
Correspondence between BSTs and quicksort partitioning

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

BSTs: mathematical analysis

Proposition. If $N$ distinct keys are inserted into a BST 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 $N$ distinct keys are inserted into a BST in random order, the expected height is $\sim 4.311 \ln N$.

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>search</td>
<td>insert</td>
</tr>
<tr>
<td>search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>insert</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>search hit</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>insert</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>sequential search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search</td>
<td>log $N$</td>
<td>log $N$</td>
<td>log $N$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why not shuffle to ensure a (probabilistic) guarantee of log $N$?

3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
Minimum and maximum

Minimum. Smallest key in table.
Maximum. Largest key in table.

Q. How to find the min / max?

Computing the floor

Floor. Find largest key ≤ k?

Case 1. [ key in node x = k ]
The floor of k is k.

Case 2. [ key in node x > k ]
The floor of k is in the left subtree of x.

Case 3. [ key in node x < k ]
The floor of k can't be in left subtree of x; it is either in the right subtree of x or it is the key in node x.

Floor and ceiling

Floor. Largest key ≤ a given key.
Ceiling. Smallest key ≥ a given key.

Q. How to find the floor / ceiling?

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)
{
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0) return floor(x.left, key);
    else return floor(x.right, key);
}

if (t != null) return t;
else return x;
### Rank and select

**Q.** How to implement `rank()` and `select()` efficiently?

**A.** In each node, store the number of nodes in its subtree.

![Tree Diagram]

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

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

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

  private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    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.count = 1 + size(x.left) + size(x.right);
    return x;
  }
}
```

### Computing the rank

**Rank.** How many keys < \(k\)?

**Case 1.** [key in node = \(k\)]
All keys in left subtree < \(k\);
no key in right subtree < \(k\).

**Case 2.** [key in node \(x\) > \(k\)]
No key in right subtree < \(k\);
recursively compute rank in left subtree.

**Case 3.** [key in node \(x\) < \(k\)]
All keys in left subtree < \(k\);
some keys in right subtree may be < \(k\).

![Tree Diagram]

```
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 if (cmp == 0) return size(x.left);
}
```
### BST: ordered symbol table operations summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential Search</th>
<th>Binary Search</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$h$</td>
</tr>
<tr>
<td>insert</td>
<td>$N$</td>
<td>$N$</td>
<td>$h$</td>
</tr>
<tr>
<td>min / max</td>
<td>$N$</td>
<td>$1$</td>
<td>$h$</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$h$</td>
</tr>
<tr>
<td>rank</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$h$</td>
</tr>
<tr>
<td>select</td>
<td>$N$</td>
<td>$1$</td>
<td>$h$</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

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

Order of growth of running time of ordered symbol table operations

### ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops?</th>
<th>Key Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

Next. Deletion in BSTs.

### 3.2 Binary Search Trees

- **BSTs**
- **ordered operations**
- **deletion**

#### BST deletion: lazy approach

To remove a node with a given key:
- Set its value to `null`.
- Leave key in tree to guide search (but don’t consider it equal in search).

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 (memory) 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.

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

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

Hibbard deletion

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

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

Case 1. [1 child] Delete τ by replacing parent link.

Case 2. [2 children]
- Find successor x of τ.
- Delete the minimum in τ’s right subtree.
- Put x in τ’s spot.
Hibbard deletion: Java implementation

```java
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;
        if (x.left == null) return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.count = size(x.left) + size(x.right) + 1;
    return x;
}
```

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.

Surprising consequence. Trees not random (!) $\Rightarrow \sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops?</th>
<th>Key Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>unordered list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>ordered array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next lecture. Guarantee logarithmic performance for all operations.