**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 URL with specified IP address.
- Given URL, find corresponding IP address.

<table>
<thead>
<tr>
<th>URL</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>
**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 given URL</td>
<td>URL</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>find URL given IP address</td>
<td>IP address</td>
<td>URL</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>

**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()`.
  ```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: `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.

Default implementation. (x == y)

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

User-defined implementations. Some care needed.

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(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;
    }
```

Implementing equals for user-defined types

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

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

```
% 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 10 < tinyTale.txt
it 10
% java FrequencyCounter 8 < tale.txt
business 122
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

**Symbol Tables**

- API
- Elementary implementations
- 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.

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>worst case search</th>
<th>worst case insert</th>
<th>average case search hit</th>
<th>average case insert</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N / 2</td>
<td>N</td>
<td>no</td>
<td>equals()</td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.

Binary search

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

Rank helper function. How many keys < k?

keys[]

successful search for P

0 1 2 3 4 5 6 7 8 9
0 9 4 A E H L M P R S X
5 9 7 A E H L M P R S X
5 6 5 A E H L M P R S X
6 6 6 A E H L P R S X

unsuccessful search for Q

0 1 2 3 4 5 6 7 8 9
0 9 4 A E H L M P R S X
5 9 7 A E H L M P R S X
5 6 5 A E H L M P R S X
7 6 6 A E H L P R S X

Trace of binary search for rank in an ordered array
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;
}
```

Binary search: mathematical analysis

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

**Pf.** $T(N) = \text{number of compares to binary search in a sorted array of size } N$. 

$$T(N) \leq T\left(\frac{N}{2}\right) + 1$$

Recall lecture 2.

Binary search: trace of standard indexing client

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

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>N 0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>S 0 S S</td>
<td>S 1 0</td>
</tr>
<tr>
<td>E 1 E S</td>
<td>E 2 1 0</td>
</tr>
<tr>
<td>A 2 A E S</td>
<td>A 3 2 1 0</td>
</tr>
<tr>
<td>R 3 A E R S</td>
<td>R 4 2 1 3 0</td>
</tr>
<tr>
<td>C 4 A C E R S</td>
<td>C 5 2 4 1 3 0</td>
</tr>
<tr>
<td>H 5 A C E H R S</td>
<td>H 6 2 4 1 5 3 0</td>
</tr>
<tr>
<td>E 6 A C E H R S</td>
<td>E 7 2 4 6 5 3 0 7</td>
</tr>
<tr>
<td>X 7 A C E H R S X</td>
<td>X 8 2 4 6 5 3 0 7</td>
</tr>
<tr>
<td>A 8 A C E H R S X</td>
<td>A 9 2 4 6 5 3 0 7</td>
</tr>
<tr>
<td>M 9 A C E H M R S X</td>
<td>M 10 2 4 6 5 9 3 0 7</td>
</tr>
<tr>
<td>P 10 A C E H M P R S X</td>
<td>P 11 2 4 6 5 9 10 3 0 7</td>
</tr>
<tr>
<td>I 11 A C E H L M P R S X</td>
<td>I 12 2 4 6 5 11 9 10 3 0 7</td>
</tr>
<tr>
<td>E 12 A C E H L M P R S X</td>
<td>E 13 2 4 6 5 11 9 10 3 0 7</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search</td>
<td>log N</td>
<td>N / 2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.

Ordered symbol table API

public class ST<Key extends Comparable<Key>, Value> {
    create an ordered symbol table
    void put(Key key, Value val)  // put key-value pair into the table
    Value get(Key key)            // value paired with 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
    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]
    boolean contains(Key, Key)    // keys in [lo..hi], in sorted order
    Iterable<Key> keys()          // all keys in the table, in sorted order
}

Examples of ordered symbol-table operations

key  values
min()  09:00:00 Chicago
       09:00:03 Phoenix
       09:00:11 Houston
get(09:00:13)  09:00:59 Chicago
floor(09:01:00)  09:03:13 Chicago
select(7)  09:10:21 Seattle
keys(09:15:00, 09:25:00)  09:14:25 Phoenix
                       09:19:32 Chicago
                       09:19:46 Chicago
                       09:21:05 Chicago
                       09:22:43 Seattle
                       09:22:54 Seattle
                       09:25:52 Chicago
ceil(09:30:00)  09:35:23 Chicago
                09:36:14 Seattle
max()  09:37:44 Phoenix
size(09:15:00, 09:25:00)  6
rank(09:10:25)  7

Examples of ordered symbol-table operations
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>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>min / max</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>rank</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>select</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>$O(N \log N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

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

---

Search Algorithms

- Symbol Tables
  - 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).

**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.
**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.

```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;
    }
}
```

Key and Value are generic types; Key is Comparable.

**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 */
    }
}
```

**Binary search tree operations**

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

- **Successful search for H**

**Binary search tree operations**

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

- **Successful search for H**
Binary search tree operations

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

Successful search for H

Binary search tree operations

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

Successful search for H

Binary search tree operations

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

Successful search for H

Binary search tree operations

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

Successful search for H

compare H and E (go right)

compare H and R (go left)
**Binary search tree operations**

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

successful search for H

![Binary search tree diagram](image1)

unsuccessful search for G

![Binary search tree diagram](image2)
**Binary search tree operations**

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

unsuccessful search for G

[Diagram of a binary search tree with nodes A, C, H, M, E, R, S, X and paths indicating unsuccessful search for G]

**Binary search tree operations**

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

unsuccessful search for G

[Diagram of a binary search tree with nodes A, C, H, M, E, R, S, X and paths indicating unsuccessful search for G]

**Binary search tree operations**

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

unsuccessful search for G

[Diagram of a binary search tree with nodes A, C, H, M, E, R, S, X and paths indicating unsuccessful search for G]

**Binary search tree operations**

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

unsuccessful search for G

[Diagram of a binary search tree with nodes A, C, H, M, E, R, S, X and paths indicating unsuccessful search for G]

[Diagram of a binary search tree with nodes A, C, H, M, E, R, S, X and arrows indicating compare G and R (go left)]
Binary search tree operations

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

unsuccessful search for G

Binary search tree operations

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

unsuccessful search for G

Binary search tree operations

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

unsuccessful search for G

Binary search tree operations

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

unsuccessful search for G
**Binary search tree operations**

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

- Insert G

**Binary search tree operations**

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

- Insert G

**Binary search tree operations**

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

- Insert G

**Binary search tree operations**

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

- Insert G
**Binary search tree operations**

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

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

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

**Insert.** If less, go left; if greater, go right; if null, insert.
Binary search tree operations

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

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

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

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

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

![Diagram of BST search](image)

- **Successful search for R:**
  - R is less than S so look to the left.
  - Black nodes could match the search key.
  - Found R (search hit) so return value.

- **Unsuccessful search for T:**
  - T is greater than S so look to the right.
  - Gray nodes cannot match the search key.
  - T is not in tree (search miss).

**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.

![Diagram of BST insert](image)

**BST insert: Java implementation**

Put. Associate value with key.

<table>
<thead>
<tr>
<th>Concise, but tricky, recursive code; read carefully!</th>
</tr>
</thead>
</table>

```java
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 1 + depth of node.
**BST trace: standard indexing client**

- Black nodes are accessed in search.
- Red nodes are new.
- Gray nodes are untouched.

**Tree shape**

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

**Remark.** Tree shape depends on order of insertion.

**Correspondence between BSTs and quicksort partitioning**

- 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 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: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N/2$</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>$\lg N$</td>
<td>$N$</td>
<td>$\lg N$</td>
<td>$N/2$</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>$1.39 \lg N$</td>
<td>$1.39 \lg N$</td>
</tr>
</tbody>
</table>

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?

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.

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().

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 if (cmp == 0) return size(x.left);
    else if (cmp == 0) return size(x.left);
}

BST implementation: subtree counts

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

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;
}

Selection

Select. Key of given rank.

public Key select(int k) {
    if (k < 0) return null;
    if (k >= size()) return null;
    Node x = select(root, k);
    ... return select(x.right, k-t-1);
    else if (t < k)
    return select(x.left, k-t);
    else if (t == k)
    return x;
}

private Node select(Node x, int k) {
    if (x == null) return null;
    int t = size(x.left);
    if (t >= k)
    return select(x.left, k);
    else if (t > k)
    return select(x.right, k-t);
    else if (t == k)
    return x;
}
Inorder traversal

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

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

```java
class Node {
    public Key key;
    public Node left, right;
}

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

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>$1$</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>

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

**Binary Search Trees**

- BSTs
- Ordered operations
- Deletion
**ST implementations: summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Iteration</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>lg N</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>yes</td>
</tr>
</tbody>
</table>

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. \( \sim 2 \ln N' \) 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.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.

```java
node to delete
```

```java
replace with null link
```

```java
available for garbage collection
```
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.

**Case 2.** [2 children]
- Find successor \( x \) of \( t \).
- Delete the minimum in \( t \)'s right subtree.
- Put \( x \) in \( t \)'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;
        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;
}
```

---

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>lookup</th>
<th>insert</th>
<th>delete</th>
<th>iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Other operations also become $\sqrt{N}$ if deletions allowed.

**Red-black BST.** Guarantee logarithmic performance for all operations.