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

```plaintext
  E
  / 
 A   C
 / 
R   H
 / 
M   S
```

- compare H and S (go left)

- black nodes could match the search key
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)

[Diagram showing a binary search tree with nodes labeled A, C, H, M, R, S, X, and E.]

**Successful search for H**

**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

unsuccessful search for G

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

compare G and E
(go right)

G

E

X
A
C
R

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M

unsuccessful search for G

E
S
X
A
C
R
H
M
Binary search tree operations

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

- Unsuccessful search for G

1. ![Binary search tree operations (Page 21)](image1.png)
2. ![Binary search tree operations (Page 22)](image2.png)
3. ![Binary search tree operations (Page 23)](image3.png)
4. ![Binary search tree operations (Page 24)](image4.png)

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

- Insert G

1. ![Binary search tree operations (Page 21)](image1.png)
2. ![Binary search tree operations (Page 22)](image2.png)
3. ![Binary search tree operations (Page 23)](image3.png)
4. ![Binary search tree operations (Page 24)](image4.png)
Binary search tree operations

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

- Insert G
  - Compare G and S (go left)
  - Insert G

Binary search tree operations

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

- Insert G
  - Compare G and S (go left)
  - Insert G

Binary search tree operations

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

- Insert G
  - Compare G and E (go right)
  - Insert G

Binary search tree operations

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

- Insert G
  - Compare G and S (go right)
  - Insert G
**Binary search tree operations**

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

- Insert G

![Binary search tree diagram](image)

**Binary search tree operations**

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

- Insert G

![Binary search tree diagram](image)

**Binary search tree operations**

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

- Insert G

![Binary search tree diagram](image)

**Binary search tree operations**

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

- Insert G

![Binary search tree diagram](image)
Binary search tree operations

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

insert G

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

Successful (left) and unsuccessful (right) search in a BST

successful search for R

unsuccessful search for T
**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 is equal to 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);
}

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 x.val = val;
    return x;
}
```

**Concise, but tricky, recursive code; read carefully!**

**Cost.** Number of compares is equal to 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.

---

**BST trace: standard indexing client**

Key: Value

- A: 8
- B: 12
- C: 6
- E: 1
- H: 5
- L: 11
- M: 10
- P: 10
- R: 3
- S: 7
- X: 12

**Key change:**

- A: 8
- B: 12
- C: 6
- E: 1
- H: 5
- L: 11
- M: 10
- P: 10
- R: 3
- S: 7
- X: 12

Insertion into a BST

- Insert L at this null link
- Create new node
- Node looks on the way up
Tree shape

- Many BSTs correspond to 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

<table>
<thead>
<tr>
<th>Key</th>
<th>Pseudonymikal</th>
<th>Pseudonymikal</th>
<th>Head</th>
<th>Homo</th>
<th>Micty</th>
<th>Uus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dce</td>
<td>ACD</td>
<td>ACD</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>E</td>
<td>E</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>S</td>
<td>ACD</td>
<td>ACD</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>8</td>
<td>P</td>
<td>E</td>
<td>E</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>E</td>
<td>C</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>12</td>
<td>S</td>
<td>ACD</td>
<td>ACD</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>13</td>
<td>T</td>
<td>E</td>
<td>E</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

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

BST insertion: random order visualization

Ex. Insert keys in random order.

| N = 255 | \( \max = 16 \) | \( \text{avg} = 9.1 \) | \( \text{opt} = 7.0 \) |

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 \( \sim 4.311 \ln N \).

How Tall is a Tree?

Bruce Reed
CMNS Park, France
reed@math.ucla.edu

ABSTRACT

Let \( H_n \) be the height of a random binary search tree on \( n \) nodes. We show that expected height \( H_n = \Theta(\ln n) \), useful for thinking about \( n \)-node trees, and \( V(H_n) = \Theta(1) \). We also show that \( \text{Var}(H_n) = O(1) \).

But... Worst-case height is \( N \).
(exponentially small chance when keys are inserted in random order)
ST implementations: frequency counter

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>sequential search</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>N/2</td>
</tr>
</tbody>
</table>

BST: N | N | 1.39 lg N | 1.39 lg N | stay tuned | compareTo() |

Minima and maximum

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

Q. How to find the min / max?

**Binary Search Trees**

- BSTs
- Ordered operations
- Deletion
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**

1. **Case 1.** \( k \) equals the key at root
   The floor of \( k \) is \( k \).

2. **Case 2.** \( k \) is less than the key at root
   The floor of \( k \) is in the left subtree.

3. **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.

**Subtree counts**

In each node, we store the number of nodes in the subtree rooted at that node; to implement \( \text{size}() \), return the count at the root.

Remark. This facilitates efficient implementation of \( \text{rank}() \) and \( \text{select}() \).

**Computing the floor**

```java
public Key floor(Key key) {
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
```

```java
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;
}
```
BST implementation: subtree counts

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

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

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

```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;
x.N = 1 + size(x.left) + size(x.right);
return x;
}
```

Rank

Easy recursive algorithm (4 cases!)

```java
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);
}
```

Selection

Select. Key of given rank.

```java
public Key select(int k)
{
  if (k < 0) return null;
  if (k >= size()) return null;
  Node x = select(root, k);
  return x.key;
}

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-1);
  else
    if (t == k)
      return x;
  return x;
}
```

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);
}
```
**Inorder traversal**

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

**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>lg 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>lg N</td>
<td>h</td>
</tr>
<tr>
<td>rank</td>
<td>N</td>
<td>lg 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>

**Order of growth of running time of ordered symbol table operations**

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

**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</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>(linked list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N/2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>1.39 lg N</td>
<td>yes</td>
<td>compareTo()</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.

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.

Case 1. [1 child] Delete \( t \) by replacing parent link.

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;
}
Hibbord deletion

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

Case 2. [2 children]
- Find successor $x$ of $t$.
- Delete the minimum in $t$’s right subtree.
- Put $x$ in $t$’s spot.

Hibbord 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.

Hibbord 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;
}
```

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>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential</td>
<td>$N$</td>
<td>$N$</td>
<td>$N/2$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$N/2$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$1.39 \lg N$</td>
<td>$\sqrt{N}$</td>
</tr>
</tbody>
</table>

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

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