Binary Search Tree (BST)

- Last lecture, we talked about binary search & linear search
  - One had high cost for reorganisation,
  - The other had high cost for searching
- In this lecture we will use Binary Trees, for searching
- Plan in a nutshell:
  - Assert a more strict property compared to the Heap-Property (in priority-queues), Remember what that was?
  - Know exactly which subtree to look for at each node

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
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.

- unsuccessful search for G

Binary search tree operations

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

Binary search tree operations

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

- unsuccessful search for G
  - compare G and S (go left)
Search. If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

Binary search tree operations

unsuccessful search for G

compare G and E (go right)

unsuccessful search for G

compare G and R (go left)

unsuccessful search for G

compare G and E (go right)
Binary search tree operations

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

unsuccessful search for G

unsuccessful search for G

unsuccessful search for G

unsuccessful search for G

compare G and H (go left)

no more tree (search miss)
Binary search tree operations

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

```
insert G
```

![Binary search tree operations](image1)

Binary search tree operations

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

```
insert G
```

![Binary search tree operations](image2)

Binary search tree operations

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

```
insert G
```

![Binary search tree operations](image3)

Binary search tree operations

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

```
insert G
```

![Binary search tree operations](image4)
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 $G$ into the tree.

```
  E
 / \
S   X
 /   \
A   R
   / \ 
  C   H
   / \ 
  G   M
```

```
  E
 /   \
S     X
 /     \
A       R
   /     \
  C       H
   /     \
  G       M
```

- Insert $G$ into the tree.

```
  E
 /   \
S     X
 /     \
A       R
   /     \
  C       H
   /     \
  G       M
```

```
  E
 /   \
S     X
 /     \
A       R
   /     \
  C       H
   /     \
  G       M
```

- Insert $G$ into the tree.

```
  E
 /   \
S     X
 /     \
A       R
   /     \
  C       H
   /     \
  G       M
```
**BST search**

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

- Black nodes could match the search key.
- Gray nodes cannot match the search key.

**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.

**Cost.** Number of compares is equal to 1 + 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 1 + depth of node.

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

**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 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().

BST implementation: subtree counts

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

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

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

number of nodes in subtree

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.

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 t;
}
**Inorder traversal**

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

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

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

```java
private 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>

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

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

```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.
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.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.

If we always delete from the same side, the shape of tree will be not random, the right subtrees are trimmed!

Surprising consequence. Trees not random (!) ⇒ $\sqrt{N}$ 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 iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>chained list (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>1.39 lg N</td>
<td>√N</td>
<td>yes</td>
</tr>
</tbody>
</table>

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

Other operations also become √N if deletions allowed.