Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
BSTs
Ordered operations
Deletion
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
**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

![Binary search tree diagram](image-url)
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 */
  }
}
Search. If less, go left; if greater, go right; if equal, search hit.

**successful search for H**

---

**Binary search tree operations**
Binary search tree operations

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

successful search for H

black nodes could match the search key

compare H and S (go left)
Binary search tree operations

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

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

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

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

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

**Successful search for H**

![Diagram of a binary search tree with nodes A, C, H, M, R, S, and X. The path from the root to the node H is highlighted, showing a successful search.](image-url)
**Binary search tree operations**

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

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

unsuccessful search for G

G compared with S (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

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

unsuccessful search for G
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

compare G and H (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 $G$

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 diagram](image)
**Insert.** If less, go left; if greater, go right; if null, insert.

```
insert G
```

```
G

compare G and S
(go left)
```

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

**Binary search tree operations**
**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.

```plaintext
insert G
```

```
    S
     |
   --
     |
   X

    G
     
  ---
     |
  E

    A
     |
   --
     |
     C

    R
     |
   --
     |
     H

    M
```
**Binary search tree operations**

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

```
insert G
```

![Binary search tree diagram](image)
Insert. If less, go left; if greater, go right; if null, insert.

```
insert G
```

```mermaid
graph TD
```

compare G and R
(go left)
**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.

![Insertion Diagram]

- Insert G
- Compare G and H (go left)
- Go left

Diagrams:
- Tree structure with nodes labeled A, C, E, G, H, M, R, S, and X.
**Binary search tree operations**

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

```
insert G
```

![Binary search tree diagram with insertion of G]
**Binary search tree operations**

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

```
insert G
```


no more tree
(insert here)
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 tree diagram](image)
**Get.** Return value corresponding to given key, or `null` if no such key.

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

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

### Unsuccessful Search for `E`
- Gray nodes cannot match the search key.
- `E` is greater than `E` so look to the right.
- Link is null so `T` is not in tree (search miss).
**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**

**Put.** Associate value with key.

Search for key, then two cases:
- Key in tree $\Rightarrow$ reset value.
- Key not in tree $\Rightarrow$ add new 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
        if (cmp == 0)
            x.val = val;
    return x;
}
```

Cost. Number of compares is equal to 1 + depth of node.
BST trace: standard indexing client

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

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

The diagram shows a binary search tree with keys and values, demonstrating operations such as insertions and searches, along with the changes in node colors and values.
Many BSTs correspond to same set of keys.
Number of compares for search/insert is equal to $1 + \text{depth of node}$.

**Remark.** Tree shape depends on order of insertion.
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 $N$ distinct keys are inserted into a BST in random order, the expected number of compares for a search/insert is $O(\log N)$.

**Pf.** 1-1 correspondence with quicksort partitioning.

**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

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>N/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>lg N</td>
<td>stay tuned</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>lg N</td>
<td></td>
<td></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 \( \leq \) to a given key.

**Ceiling.** Smallest key \( \geq \) to a given key.

**Q.** How to find the floor /ceiling?
Computing the floor

Case 1. \([k \text{ equals the key at root}]\)
The floor of \(k\) is \(k\).

Case 2. \([k \text{ is less than the key at root}]\)
The floor of \(k\) is in the left subtree.

Case 3. \([k \text{ is greater than the key at root}]\)
The floor of \(k\) is in the right subtree (if there is any key \(\leq k\) in right subtree); otherwise it is the key in the root.
Computing the floor

```java
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;
}
```

Diagram:
- Finding floor(G)
  - G is greater than S so floor(G) could be on the right
  - G is less than S so floor(G) must be on the left
  - floor(G) in left subtree is null
  - Result
Subtree counts

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

Remark. This facilitates efficient implementation of \texttt{rank()} and \texttt{select()}. 
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;
}
**Rank.** How many keys \(< k \)?

Easy recursive algorithm (3 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.

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.

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

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

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

Recursive calls: S E A C E S R H M X
Queue: S E A C E S R H M X
Function call stack: A C E H M R S X
### BST: ordered symbol table operations summary

<table>
<thead>
<tr>
<th></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>

**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**
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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</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>N</td>
<td>N</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>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.
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.** $O(\log N')$ 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.
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 $t$.
- Delete the minimum in $t$’s right subtree.
- Put $x$ in $t$’s spot.

$x$ has no left child
but don’t garbage collect $x$
still a BST

node to delete
search for key E
t.left
deleteMin(t.right)
successor min(t.right)
go right, then go left until reaching null left link
update links and node counts after recursive calls
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;
}
```

- **Search for key**
- **No right child**
- **Replace with successor**
- **Update subtree counts**
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 (!) $\Rightarrow \sqrt{N}$ 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 iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
</tbody>
</table>

Other operations also become √N if deletions allowed.

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