Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
Today

- 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
**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.
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.
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 */
    }
}

root of BST
**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](image-url)
**Search.** If less, go left; if greater, go right; if equal, search hit.

*successful search for H*

compare H and S  
(go left)

black nodes could  
match the search key
**Search.** If less, go left; if greater, go right; if equal, search hit.

![Binary search tree operations diagram](image)

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)
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 diagram]

- **R** compares **H** and **R** (go left)
- **E**
- **A**
- **C**
- **H**
- **M**
- **S**
- **X**
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

```
        S
      /   \
    E     X
  /     /  \
A      C   R
  /  \
C    H
   /  \
  M
```
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
**Search.** If less, go left; if greater, go right; if equal, search hit.

**unsuccessful search for G**

compare G and E
(go right)
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 R (go left))
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*

![Diagram of a binary search tree with the search path highlighted. The path from the root S to the leaf G is shown.](image)
**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.

![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 operations

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

---

**insert G**

- compare G and E
  - (go right)

Diagram:

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

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

![Binary search tree diagram]

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

insert G

compare G and R
(go left)
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 diagram](image)
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 diagram]

- No more tree (insert here)
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
```

[Binary search tree diagram]

- S
- E
  - A
  - C
  - G
- R
  - H
  - M
- X
**Get.** Return value corresponding to given key, or `null` if no such key.

### Successful search for R
- R is less than S, so look to the left.
- Black nodes could match the search key.

### Unsuccessful search for T
- T is greater than S, so look to the right.
- T is less than X, so look to the left.
- Link is null, so T is not in tree (search miss).

### Successful search for R
- R is greater than E, so look to the right.
- Gray nodes cannot match the search key.

### Found R (search hit)
- Found R (search hit) so return value.
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 ⇒ reset value.
- Key not in tree ⇒ 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.

concise, but tricky, recursive code; read carefully!

Always assign the subtree returned from recursive call to a child, but does it actually change in each call?
BST trace: standard indexing client

key   value
S     0
E     1
A     2
R     3
C     4
H     5
E     6
X     7
A     8
M     9
P     10
L     11
E     12

red nodes are new
black nodes are accessed in search
gray nodes are untouched
changed value
changed value
changed value
changed value
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.
BST insertion: random order visualization

**Ex.** Insert keys in random order.
Correspondence between BSTs and quicksort partitioning

Remark. Correspondence is 1-1 if array has no duplicate keys.
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><strong>sequential search (unordered list)</strong></td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>binary search (ordered array)</strong></td>
<td>lg N</td>
<td>lg N</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td>N</td>
<td>lg N</td>
<td>lg N</td>
<td>stay tuned</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 $\leq k$ 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;
}
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.

\textbf{Remark.} This facilitates efficient implementation of \texttt{rank()} and \texttt{select()}. 
**BST implementation: subtree counts**

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

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

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

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;
}
```

number of nodes in subtree

ok to call when x is null
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);
}
```
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.

```
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.
Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.
# 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>I</td>
<td>N</td>
<td>h</td>
</tr>
<tr>
<td>min / max</td>
<td>N</td>
<td>I</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>I</td>
<td>h</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>N log N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

$h = \text{height of BST}$

(order 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>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</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</td>
<td>N</td>
<td>lg 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>lg N</td>
</tr>
</tbody>
</table>

**Next.** Deletion in BSTs.
To remove a node with a given key:

- Set its value to \texttt{null}.
- Leave key in tree to guide searches (but don't consider it equal to search key).

\textbf{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.

\textbf{Unsatisfactory solution.} Tombstone (memory) overload.
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.

node to delete

search for key E

go right, then go left until reaching null left link

successor min(t.right)

t.left
deleteMin(t.right)

update links and node counts after recursive calls

x has no left child

but don't garbage collect x

still a BST
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.

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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</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</td>
<td>N</td>
<td>lg 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>lg N</td>
</tr>
</tbody>
</table>

- Other operations also become √N if deletions allowed.

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