Search Trees
Outline

- Binary Search Trees
- AVL Trees
- Splay Trees
Learning Outcomes

From this lecture, you should be able to:

- Define the properties of a binary search tree.
- Articulate the advantages of a BST over alternative data structures for representing an ordered map.
- Implement efficient algorithms for finding, inserting and removing entries in a binary search tree.
- Articulate the reason for balancing binary search trees.
- Identify advantages and disadvantages of different algorithms (AVL, Splaying) for balancing BSTs.
- Implement algorithms for balancing BSTs (AVL, Splay).
Outline

- Binary Search Trees
- AVL Trees
- Splay Trees
Binary Search Trees

- A binary search tree is a **proper** binary tree storing key-value entries at its internal nodes and satisfying the following property:
  - Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. We have $\text{key}(u) < \text{key}(v) < \text{key}(w)$

- We will assume that external nodes are ‘placeholders’: they do not store entries (makes algorithms a little simpler)

- An in-order traversal of a binary search tree visits the keys in increasing order

- Binary search trees are ideal for maps with ordered keys.

![Binary Search Tree Diagram]
Binary Search Tree

All nodes in left subtree $< \text{ Any node } < \text{ All nodes in right subtree}$
Search: Loop Invariant

- Maintain a sub-tree.
- If the key is contained in the original tree, then the key is contained in the sub-tree.
Search: Define Step

- Cut sub-tree in half.
- Determine which half the key would be in.
- Keep that half.

If key < root, then key is in left half.
If key = root, then key is found
If key > root, then key is in right half.
Search: Algorithm

- To search for a key $k$, we trace a downward path starting at the root.
- The next node visited depends on the outcome of the comparison of $k$ with the key of the current node.
- If we reach a leaf, the key is not found and return of an external node signals this.
- Example: find(4):

  - Call TreeSearch(4, root)

**Algorithm** *TreeSearch*(p, k)

if $p$ is external then
  return $p$
else if $k == \text{key}(p)$ then
  return $p$
else if $k < \text{key}(p)$
  return *TreeSearch*(left($p$), k)
else if $k > \text{key}(p)$
  return *TreeSearch*(right($p$), k)
Insertion

- To perform operation \text{insert}(k, v), we search for key \( k \) (using TreeSearch)
- Suppose \( k \) is not already in the tree, and let \( p \) be the leaf reached by the search
- We expand \( p \) into an internal node and insert the entry at \( p \).
- Example: put \((5, v)\)
Insertion

- Suppose we search for key $k$ (using TreeSearch) and find it at position $p$.
- Then we simply update the value of the entry at $p$.
- Example: put(4, $v$)

![Binary Search Tree with update value to $v$]
Deletion

- To perform operation \texttt{remove}(k), we search for key \texttt{k}
- Suppose key \texttt{k} is in the tree, and let \texttt{p} be the position storing \texttt{k}
- If position \texttt{p} has only one internal leaf child \texttt{r}, we remove the node at \texttt{p} and promote \texttt{r}.
- Example: remove(4)
Deletion (cont.)

- If \( v \) has two internal children:
  - we find the internal position \( r \) that precedes \( p \) in an in-order traversal (this node has the largest key less than \( k \))
  - we copy the entry stored at \( r \) into position \( p \)
  - we now delete the node at position \( r \) (which cannot have a right child) using the previous method.

- Example: remove(8)
Performance

- Consider a map with \( n \) items implemented by means of a linked binary search tree of height \( h \)
  - the space used is \( O(n) \)
  - methods find, insert and remove take \( O(h) \) time

- The height \( h \) is \( O(n) \) in the worst case and \( O(\log n) \) in the best case

- It is thus worthwhile to balance the tree (next topic)!
Assignment 3 Q1:

- findAllInRange(k1, k2)
- Step 1: Find Lowest Common Ancestor
  - Example 1: k1 = 20, k2 = 52
  - Example 2: k1 = 41, k2 = 53
Assignment 3 Q1:

- Step 2: Find all keys in left subtree above $k_1$

- Example: $k_1 = 41$, $k_2 = 53$
Assignment 3 Q1:

- Step 3: Add lowest common ancestor

  - Example: $k_1 = 41$, $k_2 = 53$
Assignment 3 Q1:

- Step 4: Find all keys in right subtree below k2
  - Example: k1 = 41, k2 = 53
Maps and Trees in net.datastructures

- **TreeMap** instantiates a **BalanceableBinaryTree** to store entries.
Maps and Trees in net.datastructures

- **TreeMap** instantiates each **Entry** as a **MapEntry**.
- **Treemap** provides `root()`, `left()`, `right()`, `isExternal()`, … to navigate the binary search tree.
Outline

- Binary Search Trees
- AVL Trees
- Splay Trees
AVL Trees

- The AVL tree is the first balanced binary search tree ever invented.

- It is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."
AVL Trees

- AVL trees are balanced.
- An AVL Tree is a **binary search tree** in which the heights of siblings can differ by at most 1.
Height of an AVL Tree

Claim: The height of an AVL tree storing $n$ keys is $O(\log n)$. 
Proof: We compute a lower bound \( n(h) \) on the number of internal nodes of an AVL tree of height \( h \).

Observe that \( n(1) = 1 \) and \( n(2) = 2 \).

For \( h > 2 \), a minimal AVL tree contains the root node, one minimal AVL subtree of height \( h - 1 \) and another of height \( h - 2 \).

That is, \( n(h) = 1 + n(h - 1) + n(h - 2) \).

Knowing \( n(h - 1) > n(h - 2) \), we get \( n(h) > 2n(h - 2) \). So

\[
n(h) > 2n(h - 2), n(h) > 4n(h - 4), n(h) > 8n(n - 6), ..., n(h) > 2^i n(h - 2i)
\]

If \( h \) is even, we let \( i = h/2 - 1 \), so that \( n(h) > 2^{h/2 - 1} n(2) = 2^{h/2} \).

If \( h \) is odd, we let \( i = h/2 - 1/2 \), so that \( n(h) > 2^{h/2 - 1/2} n(1) = 2^{h/2 - 1/2} \).

In either case, \( n(h) > 2^{h/2 - 1} \).

Taking logarithms: \( h < 2 \log(n(h)) + 2 \).

Thus the height of an AVL tree is \( O(\log n) \).
Insertion

height = 2

7

1
4
0
0
0

Insert(2)

height = 3

7

2
4
0
0
0

Problem!
Insertion

- Imbalance may occur at any ancestor of the inserted node.
Insertion: Rebalancing Strategy

- Step 1: Search

Starting at the inserted node, traverse toward the root until an imbalance is discovered.
Insertion: Rebalancing Strategy

- **Step 2: Repair**

  - The repair strategy is called **trinode restructuring**.

  - 3 nodes $x$, $y$ and $z$ are distinguished:
    - $z =$ the parent of the high sibling
    - $y =$ the high sibling
    - $x =$ the high child of the high sibling

  - We can now think of the subtree rooted at $z$ as consisting of these 3 nodes plus their 4 subtrees
Insertion: Rebalancing Strategy

Step 2: Repair

- The idea is to rearrange these 3 nodes so that the middle value becomes the root and the other two becomes its children.

- Thus the grandparent – parent – child structure becomes a triangular parent – two children structure.

- Note that $z$ must be either bigger than both $x$ and $y$ or smaller than both $x$ and $y$.

- Thus either $x$ or $y$ is made the root of this subtree.

- Then the subtrees $T_0 - T_3$ are attached at the appropriate places.

- Since the heights of subtrees $T_0 - T_3$ differ by at most 1, the resulting tree is balanced.
Insertion: Trinode Restructuring Example

Note that $y$ is the middle value.

 Restructure

One is $h-3$ & one is $h-4$
Insertion: Trinode Restructuring - 4 Cases

- There are 4 different possible relationships between the three nodes x, y and z before restructuring:

1. \(x < y < z\)
2. \(z < y < x\)
3. \(y < x < z\)
4. \(z < x < y\)
Insertion: Trinode Restructuring - 4 Cases

This leads to 4 different solutions, all based on the same principle.

$x < y < z$

$z < y < x$

$y < x < z$

$z < x < y$
Insertion: Trinode Restructuring - Case 1

Note that \( y \) is the middle value.
Insertion: Trinode Restructuring - Case 2

Restructure

one is h-3 & one is h-4
Insertion: Trinode Restructuring - Case 3

Restructure

one is h-3 &
one is h-4

y

T3

h-3

h-2

h-1

x

y

T0

T1

T2

T3

h-3

h-2

h-1

h-3

one is h-3 &
one is h-4

z

height = h

h-3

h-2

h-1

h-2

h-1

h-3

h-3

z

h-3

h-2

h-1

h-3

h-3
Insertion: Trinode Restructuring - Case 4

The diagram illustrates the process of restructuring a trinode in a balanced binary tree to maintain height balance. The original tree has a node z at height h, with sub-trees T_0, T_1, and T_2 at heights h-3, h-2, and h-2, respectively.

The restructuring involves moving the subtree rooted at y with height h-1 up to join the subtree rooted at x, which has height h-2. This results in a new structure where the subtrees T_0, T_1, T_2, and T_3 are balanced, with one subtree at height h-3 and another at height h-4.

Key steps in the restructuring:
- Move the subtree rooted at y up to connect with the subtree rooted at x.
- Ensure all subtrees maintain the height balance.

The diagram visually represents the balance and structure before and after the restructuring process.
Insertion: Trinode Restructuring - The Whole Tree

- Do we have to repeat this process further up the tree?
- No!
  - The tree was balanced before the insertion.
  - Insertion raised the height of the subtree by 1.
  - Rebalancing lowered the height of the subtree by 1.
  - Thus the whole tree is still balanced.
End of Lecture

Nov 12, 2015
Removal

- Imbalance may occur at an ancestor of the removed node.
Removal: Rebalancing Strategy

➢ Step 1: Search

- Let \( w \) be the node actually removed (i.e., the node matching the key if it has a leaf child, otherwise the node directly preceding in an in-order traversal.

- Starting at \( w \), traverse toward the root until an imbalance is discovered.

![Tree diagram showing height = 3 and a problem]

Problem!
Removal: Rebalancing Strategy

Step 2: Repair

- We again use trinode restructuring.

- 3 nodes x, y and z are distinguished:
  - z = the parent of the high sibling
  - y = the high sibling
  - x = the high child of the high sibling (if children are equally high, keep chain linear)
Removal: Rebalancing Strategy

Step 2: Repair

- The idea is to rearrange these 3 nodes so that the middle value becomes the root and the other two becomes its children.

- Thus the **grandparent – parent – child** structure becomes a triangular **parent – two children** structure.

- Note that \( z \) must be either bigger than both \( x \) and \( y \) or smaller than both \( x \) and \( y \).

- Thus either \( x \) or \( y \) is made the root of this subtree, and \( z \) is lowered by 1.

- Then the subtrees \( T_0 – T_3 \) are attached at the appropriate places.

- Although the subtrees \( T_0 – T_3 \) can differ in height by up to 2, after restructuring, sibling subtrees will differ by at most 1.
Removal: Trinode Restructuring - 4 Cases

There are 4 different possible relationships between the three nodes $x$, $y$ and $z$ before restructuring:

1. $x < y < z$
2. $z < y < x$
3. $y < x < z$
4. $z < x < y$
Removal: Trinode Restructuring - Case 1

Note that y is the middle value.

Restructure
Removal: Trinode Restructuring - Case 2

Restructure

height = h

T₀

T₁

T₂

T₃

h-3

h-1

h-2

h-3

h-2 or h-3

h-1 or h-2

h-3

h-2 or h-3

h-3 or h-3 & h-4

h-3 or h-3 & h-4
Removal: Trinode Restructuring - Case 3

- The height of TreeNode $z$ is $h$.
- The removal of $z$ requires restructuring.
- After restructuring, the height of TreeNode $x$ becomes $h-1$.

Restructure:

- The restructuring involves adjusting the heights of $T_0$, $T_1$, and $T_2$ to ensure the overall height remains $h-3$.

The figure illustrates the process of restructuring, showing the changes in heights and the restructuring of the subtrees to maintain the binary tree's properties.
Removal: Trinode Restructuring - Case 4

```
\begin{align*}
& \text{Restructure} \\
& \text{height} = h
\end{align*}
```
Removal: Rebalancing Strategy

- Step 2: Repair

- Unfortunately, trinode restructuring may reduce the height of the subtree, causing another imbalance further up the tree.

- Thus this search and repair process must in the worst case be repeated until we reach the root.
Java Implementation of AVL Trees

Please see text
Running Times for AVL Trees

- a single restructure is $O(1)$
  - using a linked-structure binary tree

- find is $O(\log n)$
  - height of tree is $O(\log n)$, no restructures needed

- insert is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring is $O(1)$

- remove is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
AVLTree Example
Outline

- Binary Search Trees
- AVL Trees
- Splay Trees
Splay Trees

- Self-balancing BST
- Invented by Daniel Sleator and Bob Tarjan
- Allows quick access to recently accessed elements
- Bad: worst-case $O(n)$
- Good: average (amortized) case $O(\log n)$
- Often perform better than other BSTs in practice
Splaying

- Splaying is an operation performed on a node that iteratively moves the node to the root of the tree.
- In splay trees, each BST operation (find, insert, remove) is augmented with a splay operation.
- In this way, recently searched and inserted elements are near the top of the tree, for quick access.
3 Types of Splay Steps

- Each splay operation on a node consists of a sequence of splay steps.
- Each splay step moves the node up toward the root by 1 or 2 levels.
- There are 2 types of step:
  - Zig-Zig
  - Zig-Zag
  - Zig
- These steps are iterated until the node is moved to the root.
Zig-Zig

- Performed when the node \( x \) forms a linear chain with its parent and grandparent.

- i.e., right-right or left-left

\[
x \quad \begin{align*}
T_1 & \quad T_2 \\
y & \quad T_3 \\
z
\end{align*}
\]

\[
x \quad \begin{align*}
T_1 & \quad T_2 \\
y & \quad T_3 \\
z
\end{align*}
\]
Zig-Zag

- Performed when the node $x$ forms a non-linear chain with its parent and grandparent
  - i.e., right-left or left-right
Zig

- Performed when the node $x$ has no grandparent
  - i.e., its parent is the root

```
+---+---+---+
| x |   |   |
+---+---+---+
     +---+---+
     |   |   |
     |   |   |
+---+---+---+
| w | T_3 | T_4 |
+---+---+---+
     +---+---+
     |   |   |
     |   |   |
+---+---+---+
| T_1 | T_2 |   |
+---+---+---+
```

```
+---+---+---+
| x |   |   |
+---+---+---+
     +---+---+
     |   |   |
     |   |   |
+---+---+---+
| w | T_1 | T_2 |
+---+---+---+
     +---+---+
     |   |   |
     |   |   |
+---+---+---+
| T_3 | T_4 |   |
+---+---+---+
```
Splay Trees & Ordered Maps

- which nodes are splayed after each operation?

<table>
<thead>
<tr>
<th>method</th>
<th>splay node</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(k)</td>
<td>if key found, use that node</td>
</tr>
<tr>
<td></td>
<td>if key not found, use parent of external node where search terminated</td>
</tr>
<tr>
<td>insert(k,v)</td>
<td>use the new node containing the entry inserted</td>
</tr>
<tr>
<td>remove(k)</td>
<td>use the parent of the internal node ( w ) that was actually removed from</td>
</tr>
<tr>
<td></td>
<td>the tree (the parent of the node that the removed item was swapped with)</td>
</tr>
</tbody>
</table>
If \( v \) has two internal children:

- we find the internal position \( r \) that precedes \( p \) in an in-order traversal (this node has the largest key less than \( k \))
- we copy the entry stored at \( r \) into position \( p \)
- we now delete the node at position \( r \) (which cannot have a right child) using the previous method.

Example: remove(8) - which node will be splayed?
End of Lecture

Nov 17, 2015
Splay Tree Example
Performance

- Worst-case is $O(n)$

- Example:
  - Find all elements in sorted order
  - This will make the tree a left linear chain of height $n$, with the smallest element at the bottom
  - Subsequent search for the smallest element will be $O(n)$
Performance

Average-case is $O(\log n)$

- Proof uses amortized analysis
- We will not cover this

Operations on more frequently-accessed entries are faster.

- Given a sequence of $m$ operations on an initially empty tree, the running time to access entry $i$ is:
  
  $O\left(\log\left(\frac{m}{f(i)}\right)\right)$

  where $f(i)$ is the number of times entry $i$ is accessed.
Other Forms of Search Trees

- (2, 4) Trees
  - These are multi-way search trees (not binary trees) in which internal nodes have between 2 and 4 children.
  - Have the property that all external nodes have exactly the same depth.
  - Worst-case $O(\log n)$ operations
  - Somewhat complicated to implement

- Red-Black Trees
  - Binary search trees
  - Worst-case $O(\log n)$ operations
  - Somewhat easier to implement
  - Requires only $O(1)$ structural changes per update
Summary

- Binary Search Trees
- AVL Trees
- Splay Trees
Learning Outcomes

From this lecture, you should be able to:

- Define the properties of a binary search tree.
- Articulate the advantages of a BST over alternative data structures for representing an ordered map.
- Implement efficient algorithms for finding, inserting and removing entries in a binary search tree.
- Articulate the reason for balancing binary search trees.
- Identify advantages and disadvantages of different algorithms (AVL, Splaying) for balancing BSTs.
- Implement algorithms for balancing BSTs (AVL, Splay).