Midterm on Dec. 6 (9:10-12:00am, 大勇樓106)

- Lec 1-11, TextBook Ch1-8, 10-12

- How to prepare your midterm:
  - Understand “ALL” the materials mentioned in the slides
    - Discuss with me, your TAs, or classmates
    - Read the text book to help you understand the materials

- You are allowed to bring an A4 size note
  - Prepare your own note; write whatever you think that may help you get better scores in the midterm
Search Trees
Binary Search Trees, AVL trees, and Splay Trees
A binary search tree is a binary tree storing keys (or 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 $\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$

- External nodes do not store items

An inorder traversal of a binary search trees visits the keys in an increasing order
Search

- To search for a key \( k \), we trace a downward path starting at the root.
- The next node visited depends on the comparison of \( k \) with the key of the current node.
- If we reach a leaf, the key is not found.
- Example: get(4):
  - Call \( \text{TreeSearch}(4, \text{root}) \).
- The algorithms for \( \text{floorEntry} \) and \( \text{ceilingEntry} \) are similar.

Algorithm \( \text{TreeSearch}(k, v) \):

```plaintext
if \( \text{T.isExternal}(v) \)
   return \( v \)
if \( k < \text{key}(v) \)
   return \( \text{TreeSearch}(k, \text{T.left}(v)) \)
else if \( k = \text{key}(v) \)
   return \( v \)
else { \( k > \text{key}(v) \) }
   return \( \text{TreeSearch}(k, \text{T.right}(v)) \)
```

Diagram:
- Tree structure with nodes labeled 1, 2, 4, 6, 8, 9, and 4.
- Key comparisons indicated as <, >, and =.
- Leaf nodes represented at the bottom.
Insertion

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5
Deletion

- To perform operation remove($k$), we search for key $k$
- Assume key $k$ is in the tree, and let $v$ be the node storing $k$
- If node $v$ has a leaf child $w$, we remove $v$ and $w$ from the tree with operation removeExternal ($w$), which removes $w$ and its parent
- Example: remove 4
Deletion (cont.)

- We consider the case where the key $k$ to be removed is stored at a node $v$ whose children are both internal
  - we find the internal node $w$ that follows $v$ in an inorder traversal
  - we copy $\text{key}(w)$ into node $v$
  - we remove node $w$ and its left child $z$ (which must be a leaf) by means of operation $\text{removeExternal}(z)$

- Example: remove 3
Performance

- Consider $n$ ordered set items implemented by means of a binary search tree of height $h$
  - the space used is $O(n)$
  - methods get, put and remove take $O(h)$ time
- The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case

We want a balanced binary tree!
AVL Tree Definition

- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node $v$ of $T$, the heights of the children of $v$ can differ by at most 1

An example of an AVL tree where the heights are shown next to the nodes:
Fact: The height of an AVL tree storing $n$ keys is $O(\log n)$.

Proof: Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height $h$.

We easily see that $n(1) = 1$ and $n(2) = 2$

For $n > 2$, an AVL tree of height $h$ contains the root node, one AVL subtree of height $n-1$ and another of height $n-2$.

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 $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(h-6)$, … (by induction), $n(h) > 2^i n(h-2i)$

Solving the base case we get: $n(h) > 2^{h/2-1}$

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

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

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:

```
before insertion

after insertion
```
After Insertion

- All nodes along the path increase their height by 1
- It may violate the AVL property
Search and repair

- Let $z$ be the first violation node from the bottom along the path
- Let $y$ be $z$’s child with the higher height ($y$ is 2 greater than its sibling)
- Let $x$ be $y$’s child with the higher height
- We rebalance $z$ by calling trinode restructuring method
Trinode Restructuring

- let \((a, b, c)\) be an inorder listing of \(x, y, z\)
- perform the rotations needed to make \(b\) the topmost node of the three

\[
\begin{align*}
\text{let } & (a, b, c) \text{ be an inorder listing of } x, y, z \\
\text{perform the rotations needed to make } & b \text{ the topmost node of the three} \\
\end{align*}
\]

AVL Trees
Restructuring
(as Single Rotations)

- Single Rotations:

```
T_0
T_1
T_2
T_3
```

```
T_0
T_1
T_2

T_3
```

```
T_0
T_1
T_2

T_3
```

```
T_3
T_2
T_1
T_0
```

AVL Trees
Restructuring  
(as Double Rotations)

- double rotations:

![Diagram of double rotations in AVL Trees]

AVL Trees
Insertion Example, continued

AVL Trees
Removal

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, $w$, may cause an unbalance.

- Example:

Before deletion of 32

After deletion
Rebalancing after a Removal

- Let $z$ be the first unbalanced node encountered while travelling up the tree from $w$. Also, let $y$ be the child of $z$ with the larger height, and let $x$ be the child of $y$ with the larger height.

- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.
AVL Tree Performance

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

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

- put takes $O(\log n)$ time
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$

- remove takes $O(\log n)$ time
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
Splay Tree

- A splay tree is a binary search tree where a node is splayed after it is accessed (for a search or update).

- Deepest internal node accessed is splayed.

- Splay: Move the node to the root.

- Splaying costs $O(h)$, where $h$ is height of the tree – which is still $O(n)$ worst-case.
  - $O(h)$ rotations, each of which is $O(1)$.
Splay Tree

- which nodes are splayed after each operation?

<table>
<thead>
<tr>
<th>method</th>
<th>splay node</th>
</tr>
</thead>
</table>
| get(k)    | if key found, use that node  
if key not found, use parent of ending external node |
| put(k,v)  | use the new node containing the entry inserted                           |
| remove(k) | use the parent of the internal node that was actually removed  
from the tree (the parent of the node that the removed item was swapped with) |
Searching in a Splay Tree: Starts the Same as in a BST

- Search proceeds down the tree to found item or an external node.
- Example: Search for the item with key 11.
Example Searching in a BST, continued

- search for key 8, ends at an internal node.
Splay Trees do Rotations after Every Operation (Even Search)

- new operation: **splay**
  - splaying moves a node to the root using rotations

- right rotation
  - makes the left child \( x \) of a node \( y \) into \( y \)'s parent; \( y \) becomes the right child of \( x \)

- left rotation
  - makes the right child \( y \) of a node \( x \) into \( x \)'s parent; \( x \) becomes the left child of \( y \)

![Diagram of right rotation and left rotation in splay trees](image)
Splaying:

- “x is a left-left grandchild” means x is a left child of its parent, which is itself a left child of its parent
- p is x’s parent; g is p’s parent

start with node x

is x the root? yes → stop

no

is x a child of the root? yes

is x the left child of the root? yes → zig

no → zig

right-rotate about the root

no → zig

left-rotate about the root

is x a left-left grandchild? yes → right-rotate about g, right-rotate about p

no → zig-zig

is x a right-right right grandchild? yes → left-rotate about g, left-rotate about p

no → zig-zig

is x a right-left grandchild? yes → left-rotate about p, right-rotate about g

no → zig-zag

is x a left-right grandchild? yes → right-rotate about p, left-rotate about g

no → zig-zag
Visualizing the Splaying Cases

Splay Trees
Splaying Example

- let $x = (8,N)$
- $x$ is the right child of its parent, which is the left child of the grandparent
- left-rotate around $p$, then right-rotate around $g$

1. (before rotating)
2. (after first rotation)
3. (after second rotation)

$x$ is not yet the root, so we splay again
Splaying Example, Continued

- now $x$ is the left child of the root
- right-rotate around root

1. (before applying rotation)

2. (after rotation)

$x$ is the root, so stop
Example Result of Splaying

- tree might not be more balanced
- e.g. splay (40,X)
  - before, the depth of the shallowest leaf is 3 and the deepest is 7
  - after, the depth of shallowest leaf is 1 and deepest is 8

![Splay Trees Diagram](image)
Performance of Splay Trees

- Amortized cost of any splay operation is $O(\log n)$
- This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than $O(\log n)$ in some cases
Project Hints

- How to call google?
- How to find the reference links?
- How to encode Chinese?
HW10 (Due on Dec. 13)

Use Google and get the links!

- Get a keyword from user
- Return the urls listed in the search result
- Save the results in a hash table (we will discuss it in the next lecture)

- After this HW, you can step to the forth stage of the project
- You can apply the same technique to other search engines
Coming Up

- Recap: Binary Search Trees
  - TB Chapter 10

- Maps and Hash tables
  - TB Chapter 9 and 10