Extended and Scapegoat Trees
Overview

- In today’s lecture, we will discuss two unrelated topics that arise in the programming assignment:
  - Extended Binary Search Trees
  - Scapegoat Trees
- We will also discuss the SG Tree, which is featured in the Programming Project, Part 1
Extended Binary Search Trees

- Extended Binary Tree (from Lecture 3)
  - Internal nodes: Have exactly 2 children
  - External nodes: Have 0 children

- Basic properties
  - Any extended binary tree with \( n \) internal nodes has \( n + 1 \) leaves
Extended Binary Search Trees

- Each external node contains an entry, a key-value pair, \((x, v)\)
- Each internal node contains a splitter, \(s\)
  - If \(x \leq s\) → Left subtree
  - If \(x > s\) → Right subtree

Note that a key can be both a splitter and part of a key-value pair.
Extended Binary Search Trees

Why?

- **Memory locality**: We saw with B+ trees, we can store many splitters in a single node, increasing fan-out, thus decreasing tree height

- **Heterogenous data**: In some applications the data and splitters are different
  - Example: *Binary-space partition tree*
    - Data are points
    - Splitters are lines

![Binary-space partition tree diagram](image)
Extended Binary Search Trees

Differences with standard (unbalanced) binary search trees

- **find(x):**
  - Descend to the external node, as directed by internal nodes
  - If key matches - then found, else not
  - **Warning:** Matching a splitter means nothing!

- **Example:**
  - find(7) - yes
  - find(15) - no
  - find(10) - no! (even though root matches)
Extended Binary Search Trees

Differences with standard (unbalanced) binary search trees

- **insert(x,v):**
  - Descend to the external node. Let $y$ be its key. If $x = y$ - duplicate-key error
  - Create a new external node for $x$ and internal node to split between $x$ and $y$
  - Splitter $s$ satisfies: $\min(x, y) \leq s < \max(x, y)$
Extended Binary Search Trees

Differences with standard (unbalanced) binary search trees

- **delete(x):**
  - Descend to the external node. Let $y$ be its key. If $x \neq y$ - key-not-found error
  - Replace this node and its parent with its sibling

![Diagram showing delete(9) operation on a binary search tree]
Scapegoat Trees
Another Amortized Dictionary Data Structure

- **Amortized cost** -
  - The total cost divided by the number of operations
  - **Splay trees** - Amortized cost $O(\log n)$ for dictionary operations, even though any single operation may take $O(n)$ time

- **Are there other efficient dictionaries in the amortized sense?** **Scapegoat trees!**

- **Origins:**
  - Original idea by Arne Andersson (of AA-Tree fame), 1989
  - Rediscovered by Galperin and Rivest, 1993 (gave the name “Scapegoat Tree”)

- **Resources:**
  - [http://opendatastructures.org/versions/edition-0.1g/ods-python/8_Scapegoat_Trees.html](http://opendatastructures.org/versions/edition-0.1g/ods-python/8_Scapegoat_Trees.html)
  - [http://opendatastructures.org/newhtml/ods/latex/scapegoat.html](http://opendatastructures.org/newhtml/ods/latex/scapegoat.html)
Scapegoat Trees

Another Amortized Dictionary Data Structure

- Why should we care?
  - Amortized structures are often simpler than worst-case efficient structures
  - The update rules for scapegoat trees can be adapted to many other search trees where rotations cannot be applied (e.g., spatial decomposition trees)
  - The SG Tree in the programming assignment is a variant of the scapegoat tree
Scapegoat Trees
Overview - Balance through Rebuilding

- **Insertion:**
  - Insert just as in a standard (unbalanced) binary tree
  - Monitor the depth of the inserted node after each insertion
  - If it is too high, then there must be at least one node on the search path that has poor weight balance (left and right children have very different sizes)
  - Find such a node - it’s the scapegoat! (It is given the blame for the high depth)
  - Rebuild the subtree rooted at this node so that it is perfectly balanced

- **Deletion:**
  - Delete as in a standard (unbalanced) binary tree
  - Once the number of deletions is sufficiently large relative to the entire tree size, rebuild the entire tree so it is perfectly balanced
Scapegoat Trees

Overview - Balance through Rebuilding

- **How to rebuild a subtree?**
  - Perform an inorder traversal of the subtree, and **copy** the $n$ elements to a (sorted) array $A[0 \ldots n - 1]$
  - Take the **median** of the array as the root, and **recursively** build left and right subtrees from the two halves of the array

- **buildSubtree(A, i, k):** Build subtree for $k$-element subarray $A[i \ldots i + k - 1]$
  - If $k = 0$, return null
  - Otherwise, let $m = \left\lfloor \frac{k}{2} \right\rfloor$. Create new node $p$ with median key, $A[i + m]$
    - $p$.left = buildSubtree(A, i, m)
    - $p$.right = buildSubtree(A, i+m+1, k-m-1)

- A subtree with $n$ nodes can be rebuilt in $O(n)$ time
A scapegoat tree stores **no balance or height information** with the nodes.

In addition to the tree we maintain:
- $n$ - the current number of nodes in the tree
- $m$ - an upper bound on the tree size (we maintain: $n \leq m \leq 2n$)

**Height condition:** never exceeds $\log_{3/2} m$ ($\Rightarrow$ Tree height is $O(\log n)$)

**Size condition:**
- Initially: $n = m = 0$
- After insertion: $n++, m++$
- After deletion: $n --$ (but do not change $m$)
- If $2n < m$, rebuild the entire tree, and set $m = n$
Scapegoat Trees

Overview - More Details

- **find(x):**
  - Identical to any binary search time (time: $O(\log n)$)

- **delete(x):**
  - Identical to delete for an unbalanced binary tree
  - Decrement $n$ (but do not change $m$)
  - If $2n < m$, rebuild the entire tree, and set $m = n$
Scapegoat Trees
Overview - More Details

- **insert(x):**
  - Same as standard binary search tree insertion, keep track of inserted node’s depth (number of edges from the root)
  - Increment both $n$ and $m$
  - If inserted depth exceeds $\log_{3/2} m$:
    - Walk back up the search path until we find the first node $u$ such that
      \[
      \frac{\text{size}(u.\child)}{\text{size}(u)} > \frac{2}{3}
      \]
    - Here $\text{size}(u)$ is the number of nodes in $u$’s subtree and $u.\child$ is $u$’s child on search path
    - A node on the insertion path satisfying this is called a **candidate scapegoat**
    - Rebuild the subtree rooted at $u$
Scapegoat Trees
Overview - More Details

- insert(5):

```
13  
 /  
12 15  
 /  
9 17  
 /  
2  
 /  
1 7  
 /  
0 4  
```

```
13  
 /  
12 15  
 /  
9 17  
 /  
2  
 /  
1 7  
 /  
0 4  
```

```
13  
 /  
12 15  
 /  
4 17  
 /  
1 7  
 /  
0 2  
```

6 > log_3^11 ≈ 5.9!!
Scapegoat Trees
Overview - More Details

- Will we always find a **scapegoat node**? Yes!
- Is it **unique**? No! (9, 12, and 13 are all **candidate scapegoats**)
- **Lemma**: If there exists a node $p$ such that $\text{depth}(p) > \log^{3/2} m$, then $p$ has an ancestor $u$ that is a **candidate scapegoat**, that is,
  \[
  \frac{\text{size}(u.\text{child})}{\text{size}(u)} > \frac{2}{3}
  \]
- **Proof**: By contradiction.
  - Suppose that for every node $u$ along the path to $p$, $\text{size}(u.\text{child}) \leq \left(\frac{2}{3}\right)\text{size}(u)$
  - Letting $k = \text{depth}(p)$, by induction have $\text{size}(p) \leq \left(\frac{2}{3}\right)^k n$
  - Since $\text{size}(p) \geq 1$, this implies $\left(\frac{3}{2}\right)^k \leq n$, implies $k \leq \log_{3/2} n \leq \log_{3/2} m$, contradiction
Scapegoat Trees

Overview - More Details

- How do we compute \(\text{size}(u)\) for each node \(u\)?
- Two methods:
  1. Maintain a separate field, \(u\.size\), for each node storing the size of \(u\)'s subtree (and update as needed)
  2. Compute it on the fly, after each insertion that requires rebalancing:
     - Walk up the search path toward the root
     - Let \(u\) be any ancestor of the inserted node. Assume we know \(\text{size}(u)\).
     - We want to compute \(\text{size}(u\.parent)\):
       - Let \(u'\) be \(u\)'s sibling. Traverse the subtree rooted at \(u'\) and count the number of nodes.
       - Set \(\text{size}(u\.parent) = 1 + \text{size}(u) + \text{size}(u')\)
       - This may seem costly, but it can all be done within the amortized time bound!
Theorem: Starting with an empty tree, any sequence of \( k \) dictionary operations costs a total of \( O(k \log k) \)

Proof: (Sketch)
- **Find**: Cost is \( O(\log n) \) always (by height bound)
- **Delete**: In order to rebuild a tree due to deletions, *at least half* the entries have been deleted. A *token-based analyses* (recall stacks and rehashing from earlier lectures) can be applied here.
- **Insert**: This is analyzed by a *potential argument*. Intuitively, after any subtree of size \( k \) is rebuilt it takes \( O(k) \) insertions to force this subtree to be rebuilt again. Charge the rebuilding time against these “cheap” insertions.

Corollary: The amortized complexity of the scapegoat tree with at most \( n \) nodes is \( O(\log n) \)
**SG Tree**

A data structure invented just for the programming assignment

- **Overview - An SG Tree is:**
  - An extended binary search tree that is rebalanced like a scapegoat tree
  - Updated concepts:
    - The size of an internal node is the number of external nodes in its subtree
    - The height of a node is the maximum number of edges to any external node
  - **Similarities** with the scapegoat tree:
    - Maintain total size $n$ and upper bound $m$, where $n \leq m \leq 2n$
    - **Height condition**: Rebuild if tree height exceeds $\log_{3/2} m$ ($\Rightarrow$ Tree height is $O(\log n)$)
    - **Candidate scapegoat**: Any node on search path such that $\frac{\text{size}(u.\text{child})}{\text{size}(u)} > \frac{2}{3}$
    - **Deletion condition**: If $2n < m$, rebuild the entire tree, and set $m = n$
SG Tree

- Differences from the scapegoat tree:
  - **Nodes:** Two types of nodes:
    - **External** - store data only (a city for the programming assignment)
    - **Internal** - store splitter, left, right, subtree height, and subtree size
  - **Scapegoat Node:**
    - When insertion causes the tree’s height to exceed $\log_{3/2} m$, if multiple nodes satisfy the scapegoat condition, we chose the one **closest to the root**
    - Why? By rebuilding the **largest subtree**, we make the overall tree **more balanced**
SG Tree

- Conventions:
  - To avoid floating-point round-off errors, use integer arithmetic to test the scapegoat condition:
    \[ 2 \cdot \text{size}(u) < 3 \cdot \text{size}(u.\text{child}) \Rightarrow u \text{ is candidate scapegoat} \]
  - When inserting a new external node, the parent’s splitter is taken from its left child
SG Tree

- More conventions:
  - When rebuilding a subtree with $k$ external nodes:
    - If $k$ is even, split the internal nodes evenly among the left and right subtrees
    - If $k$ is odd, the left subtree gets $\lfloor k/2 \rfloor$ external nodes and the right subtree gets $\lceil k/2 \rceil$
    - More formally: When splitting the $k$-element subarray $A[i \ldots i + k - 1]$:
      - Set $m = \lfloor k/2 \rfloor$
      - Build left subtree with $m$ keys: $A[i \ldots i + m - 1]$
      - The splitter is $A[i + m - 1]$
      - Build right subtree with $k - m$ keys: $A[i + m \ldots i + k - 1]$
  - This convention results in the most even split and most balanced splitter value
SG Tree

Implementation hints

- Abstract class Node and two derived classes
  - `ExternalNode` - stores just a city object
  - `InternalNode` - stores splitter (a city), left, right, size, and height

- Take advantage of virtual functions when defining node operations
  - Don’t do this:
    ```
    Node insert(Node p) {
      if (p.isExternal) {
        ExternalNode pe = (ExternalNode) p;
        /* external node processing */
      }
      else {
        InternalNode pi = (InternalNode) p;
        /* internal node processing */
      }
    }
    ```
SG Tree

Implementation hints

- Instead, do this:

```java
abstract class Node {
    // ...
    abstract Node insert(Key x);
}
class InternalNode extends Node {
    // ...
    Node insert(Key x) { ... } // insertion at internal node
}
class ExternalNode extends Node {
    // ...
    Node insert(Key x) { ... } // insertion at external node
}
```
SG Tree

Implementation hints

- **Your SGTree class:**
  - **Generic**? It’s up to you.
  - We don’t maintain key-value pairs. We store city objects.
  - The print command assumes that the data object has a name and (x,y) coordinates.
  - We made ours generic, but the data type must support getName(), getX(), and getY().
  - Use inner classes for nodes:
    - Node, InternalNode, ExternalNode
  - **Private data:**

    ```java
    Node root;
    Comparator comparator; (Optional. Given with the constructor)
    Document resultsDoc; (Needed by print command)
    int n, m; (Used by the scapegoat functions)
    ```
SG Tree

Implementation hints

- **insert(x):**
  - Insert the key using the *standard recursive insertion algorithm*
  - Some modifications needed because we have an extended tree
  - While backing out from recursion, *update the size and height values for each node*
  - Increment both $n$ and $m$
  - If the new tree height *exceeds* $\log_{3/2} m$:
    - Traverse the search path from root down until finding the first candidate scapegoat
      \[ 2 \cdot \text{size}(u) < 3 \cdot \text{size}(u.\text{child}) \]
    - **Rebuild** this subtree (Note: $u$ must be an internal node)
    - (Be sure that your rebuilding function updates heights and sizes for all nodes)
SG Tree

Implementation hints

- **delete(x):**
  - Delete the key using the *standard recursive deletion* algorithm
    - Some modifications needed because we have an extended tree
  - While backing out from recursion, *update the size and height* values for each node
  - **Decrement** $n$ but not $m$
  - If $2n < m$:
    - **Rebuild** the entire tree
    - Set $m = n$
SG Tree

Implementation hints

- Write utilities for handling size and height:
  - getSize(Node p): return (p.isExternal ? 1 : p.size)
  - getHeight(Node p): return (p.isExternal ? 0 : p.height)
  - InternalNode.update():
    size = getSize(left) + getSize(right);
    height = 1 + max(getHeight(left), getHeight(right));

- Write a debugging utility for “pretty printing” your tree
  - Call this function after each major operation (insert, delete, subtree rebuilding)

- Insert a boolean flag (e.g., DEBUG), which you can turn on and off for debugging
SG Tree

Implementation hints

- **Problem:**
  - My SG Tree is ordered by \((x, y)\)-coordinates. How do I delete a city given just its name?

- **Answer:**
  - This is why we have the binary-search tree (which is ordered by name)
  - Create a “bogus city” with just a name (no coordinates)
  - Find this city in your binary-search tree and save this “complete city”
  - Delete this complete city from both data structures
Example of a rebuild operation

Supplemental
Example of SG-Tree operations

- Insertion of 13, 15, 17, 12, 16, 5, 2, 7, 9, 14, 16, 20

Initial tree:

- Insertion of 13:
  - $3 \leq \log_2 5 \approx 3.97$

- Insertion of 9:
  - $5 \leq \log_2 9 \approx 5.42$

- Insertion of 14:
  - $6 \leq \log_2 13 \approx 6.33$

Final tree:

- Rebuilding after insertion of 12

- Scapegoat candidates:

- Insertion of 7:
  - $7 > \log_2 14 \approx 6.5!!$
Another example of SG-Tree operations

$3 \leq \log_2 4 \approx 3.42$

$4 > \log_3 5 \approx 1.66$f

$5 \leq \log_2 10 \approx 3.32$
Summary

- **Extended Binary Search Trees**
  - Data stored only at the leaves (external nodes)
  - Internal nodes are used only for locating the data

- **Scapegoat Trees**
  - Another amortized binary search tree data structure
  - Rebalancing through rebuilding subtrees
  - Unlike splay trees, height is guaranteed to be $O(\log n)$

- **SG Tree**
  - An extended-tree variant of the scapegoat tree