CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

24: Binary Search Trees

Alexandra Papoutsaki
LECTURES

Mark Kampe
LABS
Lecture 24: Binary Search Trees

- Binary Search Trees
- Ordered Operations
- Deletion in BSTs
BINARY SEARCH TREES

Definitions

- **Binary Search Tree**: A binary tree in symmetric order.
- **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.
- Our textbook uses BSTs to implement symbol tables, therefore each node holds a key-value pair. Other implementations (like today’s lab) hold only a key.
# Differences between heaps and BSTs

<table>
<thead>
<tr>
<th></th>
<th>Heap</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported operations</td>
<td>Insert, delete max</td>
<td>insert, search, delete, ordered operations</td>
</tr>
<tr>
<td>What is inserted</td>
<td>Keys</td>
<td>Key-value pairs</td>
</tr>
<tr>
<td>Underlying data structure</td>
<td>(Resizing) array</td>
<td>Linked nodes</td>
</tr>
<tr>
<td>Tree shape</td>
<td>Complete binary tree</td>
<td>Depends on data</td>
</tr>
<tr>
<td>Ordering of keys</td>
<td>Heap-ordered</td>
<td>Symmetrically-ordered</td>
</tr>
<tr>
<td>Duplicate keys allowed?</td>
<td>Yes</td>
<td>No*</td>
</tr>
</tbody>
</table>

*: depends on implementation.
BST representation

- We will use an inner class Node that is composed by:
  - A Key that is comparable and a Value
  - A reference to the root nodes of the left (smaller keys) and right (larger keys) subtrees.
  - Potentially, the total number of nodes in the subtree that has root this node.
- A BST has a reference to a Node root.
Node representation

```java
private class Node {
    private Key key; // sorted by key
    private Value val; // associated data
    private Node left, right; // left and right subtrees
    private int size; // number of nodes in subtree

    public Node(Key key, Value val, int size) {
        this.key = key;
        this.val = val;
        this.size = size;
    }
}
```
3.2 Binary Search Tree Demo
Search

- If less go left.
- If greater go right.
- If equal, search hit.
- Return value corresponding to given key, or null if no such key.
  - In other implementations (including today’s lab), you return the last node you reached.
- Number of compares is equal to the depth of the node + 1.
Search example

Successful (left) and unsuccessful (right) search in a BST
Search - iterative implementation

```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;
}
```
Search - recursive implementation

- public Value get(Key key) {
  return get(root, key);
}
- private Value get(Node x, Key key) {
  if (x == null)
    return null;
  int cmp = key.compareTo(x.key);
  if (cmp < 0)
    return get(x.left, key);
  else if (cmp > 0)
    return get(x.right, key);
  else
    return x.val;
}
Insert

- If less go left.
- If greater go right.
- If null, insert.
- If already exists, update value.
- Number of compares is equal to the depth of the node + 1.
Insert example

Insertion into a BST

- Inserting L
- Search for L ends at this null link
- Create new node
- Reset links and increment counts on the way up
Insert

- **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, 1);
  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
    x.val = val;
  x.size = 1 + size(x.left) + size(x.right);
  return x;
}
Tree shape

- The same set of keys can result in different BSTs based on their order of insertion.
- Number of compares for search/insert is equal to depth of node +1.
BSTs mathematical analysis

- If $n$ distinct keys are inserted into a BST in random order, the expected number of compares of search/insert is $2 \ln n$ (or $1.39 \log n$).

- If $n$ distinct keys are inserted into a BST in random order, the expected height of tree is $4.311 \ln n$ [Reed, 2003].

- Worst case height is $n$ but highly unlikely.
  - Keys would have to come (reversely) sorted!
Correspondence between BSTs and quicksort partitioning

- If array has no duplicate keys 1-1 correspondence.
- In quicksort, pivot separates array in elements that are smaller in its left subarray and larger in its right subarray.
- In BST, root separates tree in elements that are smaller in its left subtree and larger in its right subtree.
- This is why the mathematical analysis for BSTs was the same with quicksort’s partitioning (the expected number of compares of search/insert is $2 \ln n$ as is the number of compares in quicksort).
Lecture 24: Binary Search Trees

- Binary Search Trees
- Ordered Operations
- Deletion in BSTs
Minimum and maximum

- **Minimum**: go all the way left until you find a node with no left child.

- **Maximum**: go all the way to the right until you find a node with no right child.

```java
public Key min() {
    return min(root).key;
}

private Node min(Node x) {
    if (x.left == null)
        return x;
    else
        return min(x.left);
}
```
Floor

- **Floor**: Largest key in BST \( \leq \) query key \( k \).
- **Case 1**: \( k \) equals the key in node
  - Floor of \( k \) is \( k \).
- **Case 2**: \( k \) is less than key in node
  - Floor of \( k \) is in left subtree.
- **Case 3**: \( k \) is greater than key in node
  - Floor of \( k \) is in right subtree if there is any key \( \leq k \) in right subtree.
  - Else, floor is the key in node.
- Same idea for ceiling (smallest key in BST \( \geq \) query key)
Floor

```java
public Key floor(Key key) {
    Node x = floor(root, key);
    if (x == null)
        return null;
    else
        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;
}
```
ORDERED OPERATIONS

Rank

- **Rank**: How many keys < query key k.
- k < key: Recur on left subtree?
- k == key: Everything in left subtree.
- k > key: Everything in left subtree + 1 + recur on right
Rank

- **Rank**: How many keys < query key $k$.

```java
public int rank(Key key) {
    return rank(key, root);
}

// Number of keys in the subtree less than key.
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
        return size(x.left);
}
```
## Order of growth for ordered symbol table operations

<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>$\log n$</td>
<td>$h$</td>
</tr>
<tr>
<td>insert</td>
<td>$n$</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>$\log n$</td>
<td>$h$</td>
</tr>
<tr>
<td>rank</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$h$</td>
</tr>
<tr>
<td>select</td>
<td>$n$</td>
<td>$1$</td>
<td>$h$</td>
</tr>
</tbody>
</table>

- Worst case search and insert are $O(n)$ for BSTs. Not great!
Lecture 24: Binary Search Trees

- Binary Search Trees
- Ordered Operations
- Deletion in BSTs
Delete minimum key

- Go left until finding a node with null left subtree.
- Replace the link to that node with its right subtree.
- 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.size = size(x.left) + size(x.right) + 1;
    return x;
}
```

- Symmetric for delete maximum
Hibbard deletion: Delete node which is a leaf

- Delete node by setting parent link to null.
- Example: delete 52 locates a node which is a leaf.
Hibbard deletion: Delete node with one child

- Delete node by replacing parent link.
- Example: delete 70 locates a node which has one child.
Hibbard deletion: Delete node with two children

- Delete node and replace it with successor (node with smallest of the larger keys)
- Example: delete 50 locates a node which has two children. Successor is 51.

https://visualgo.net/en/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;
        if (x.left == null)
            return x.right;
        Node t = x;  // replace with successor
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.size = size(x.left) + size(x.right) + 1;
    return x;
}
Hibbard deletion

- Unsatisfactory solution. If we were to perform many insertions and deletions the BST ends up being not symmetric and skewed to the left.
  - The cost is $\sqrt{n}$ (extremely complicated analysis).
  - No one has proved that alternating between predecessor and successor will fix this.
- Hibbard devised the algorithm in 1962. Still no algorithm for efficient deletion in BST.
- Overall, BSTs can have $O(n)$ worst-case for search, insert, and delete. We want to do better (see future lectures).
Lecture 24: Binary Search Trees

- Binary Search Trees
- Ordered Operations
- Deletion in BSTs
Readings:

- Textbook: Chapter 3.2 (Pages 396-414)
- Website: [https://algs4.cs.princeton.edu/32bst/](https://algs4.cs.princeton.edu/32bst/)

Practice Problems:

- 3.2.1-3.2.13