23: Binary Search Trees

Some slides adopted from Princeton CS236 course or Algorithms, 4th Edition by David Kauchak

Alexandra Papoutsaki
Lecture 23: Binary Search Trees

- Dictionaries
- Binary Search Trees
Dictionaries

- Also known as: symbol tables, maps, indices, associative arrays.
- Key-value pair abstractions that support two operations:
  - Insert a key-value pair.
  - Given a key, search for the corresponding value.
- Supported either with built-in or external libraries by the majority of programming languages.
Basic symbol table API

- `public class ST <Key extends Comparable<Key>, Value>`
- `ST(): create an empty symbol table. By convention, values are not null.`
- `void put(Key key, Value val): insert key-value pair.`
  - Overwrites old value with new value if key already exists.
- `Value get(Key key): return value associated with key.`
  - Returns null if key not present.
- `boolean contains(Key key): is there a value associated with key?`
- `Iterable keys(): all the keys in the symbol table.`
- `void delete(Key key): delete key and associated value.`
- `boolean isEmpty(): is the symbol table empty?`
- `int size(): number of key-value pairs.`
Ordered symbol tables

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>min()</td>
<td>09:00:00 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:00:03 Phoenix</td>
</tr>
<tr>
<td></td>
<td>09:00:13 Houston</td>
</tr>
<tr>
<td>get(09:00:13)</td>
<td>09:00:59 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:01:10 Houston</td>
</tr>
<tr>
<td>floor(09:05:00)</td>
<td>09:03:13 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:10:11 Seattle</td>
</tr>
<tr>
<td>select(7)</td>
<td>09:10:25 Seattle</td>
</tr>
<tr>
<td></td>
<td>09:14:25 Phoenix</td>
</tr>
<tr>
<td></td>
<td>09:19:32 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:19:46 Chicago</td>
</tr>
<tr>
<td>keys(09:15:00, 09:25:00)</td>
<td>09:21:05 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:22:43 Seattle</td>
</tr>
<tr>
<td></td>
<td>09:22:54 Seattle</td>
</tr>
<tr>
<td></td>
<td>09:25:52 Chicago</td>
</tr>
<tr>
<td>ceiling(09:30:00)</td>
<td>09:35:21 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:36:14 Seattle</td>
</tr>
<tr>
<td>max()</td>
<td>09:37:44 Phoenix</td>
</tr>
</tbody>
</table>

size(09:15:00, 09:25:00) is 5
rank(09:10:25) is 7
Ordered symbol table API

- **Key min()**: smallest key.
- **Key max()**: largest key.
- **Key floor(Key key)**: largest key less than or equal to given key.
- **Key ceiling(Key key)**: smallest key greater than or equal to given key.
- **int rank(Key key)**: number of keys less that given key.
- **Key select(int k)**: key with rank k.
- **Iterable keys()**: all keys in symbol table in sorted order.
- **Iterable keys(int lo, int hi)**: keys in \([lo, \ldots, hi]\) in sorted order.
Printed symbol tables are all around us

- **Dictionary**: key = word, value = definition.
- **Encyclopedia**: key = term, value = article.
- **Phonebook**: key = name, value = phone number.
- **Math table**: key = math functions and input, value = function output.

- **Unsupported operations:**
  - Add a new key and associated value.
  - Remove a given key and associated value.
  - Change value associated with a given key.
Lecture 23: Binary Search Trees

- Dictionaries
- Binary search Trees
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 dictionaries, therefore each node holds a key-value pair. Other implementations hold only a key.
## Differences between heaps and BSTs

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

*: when BSTs used to implement dictionaries.
BST representation of dictionaries

- 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`.
BST and Node implementation

```java
public class BST<Key extends Comparable<Key>, Value> {
    private Node root; // root of BST

    private class Node {
        private Key key; // sorted by key
        private Value val; // associated value
        private Node left, right; // roots of left and right subtrees
        private int size; // #nodes in subtree rooted at this

        public Node(Key key, Value val, int size) {
            this.key = key;
            this.val = val;
            this.size = size;
        }
    }
}
```
Search for a key

- If less than key in node go to left subtree.
- If greater than key in node go to right subtree.
- If given key and key at examined node are equal, search hit.
- Return value corresponding to given key, or null if no such key.
  - In other implementations, 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;
}
Practice Time

- Search for the keys 4 and 9 in the following BST:
Insert

- If less than key in node go left.
- If greater than key in node 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
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;
  }
Practice Time

- Add the key-value pairs (4,3) and (9,2) in the following BST:
3.2 Binary Search Tree Demo
Tree shape

- The same set of keys can result to 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 $O(\log n)$.
  - If $n$ distinct keys are inserted into a BST in random order, the expected height of tree is $O(\log n)$. [Reed, 2003].
- Worst case height is $n$ but highly unlikely.
  - Keys would have to come (reversely) sorted!
- All ordered operations in a dictionary implemented with a BST depend on the height of the BST.
Hibbard deletion: Delete node which is a leaf

- Simply delete node.
- Example: delete 52 locates a node which is a leaf and removes it.
Hibbard deletion: Delete node with one child

- Delete node and replace it with its child.

- Example: delete 70 locates a node which has one child and replaces it with the child.
Hibbard deletion: Delete node with two children

- Delete node and replace it with successor (node with smallest of the larger keys). Move successor’s child (if any) where successor was.

- Example: delete 50 locates a node which has two children. Successor is 51.
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;
}
Practice Time

- Delete the node 21 following Hibbard’s deletion
Answer

- Delete the node 21 following Hibbard’s deletion
Hibbard’s deletion

- Unsatisfactory solution. If we were to perform many insertions and deletions the BST ends up being not symmetric and skewed to the left.

  - Extremely complicated analysis, but average cost of deletion ends up being $\sqrt{n}$. Let’s simplify things by saying it stays $O(\log n)$.

  - No one has proven that alternating between the predecessor and successor will fix this.

- Hibbard devised the algorithm in 1962. Still no algorithm for efficient deletion in Binary Search Trees!

- Overall, BSTs can have $O(n)$ worst-case for search, insert, and delete. We want to do better (see future lectures).
Lecture 23: Binary Search Trees

- Dictionaries
- Binary Search Trees
Readings:

- Textbook: Chapters 3.1 (Pages 362–386) and 3.2 (Pages 396–414)

- Website:
  - https://algs4.cs.princeton.edu/31elementary/
  - https://algs4.cs.princeton.edu/32bst/

- Visualization:
  - https://visualgo.net/en/bst

Practice Problems:

- 3.1.1-3.1.6, 3.2.1-3.2.13