19: Binary Search Trees, 2–3 Search Trees
Lecture 19: Binary Search Trees

- Binary Search Trees
- 2-3 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.
Search example

Successful (left) and unsuccessful (right) search in a BST
Insert example

Insertion into a BST
Practice Time

- Add the key-value pairs (4,3) and (9,2) in the following BST:
3.2 Binary Search Tree Demo
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 (case 0)

- Simply delete node.

- Example: delete 52 locates a node which is a leaf and removes it.
Hibbard deletion: Delete node with one child (case 1)

- Delete node and replace it with its only child.
- Example: delete 70 locates a node which has one child and replaces it with the child.
Hibbard deletion: Delete node with two children (case 2)

- Delete node and replace it with successor (node with smallest of the larger keys).
  - Where is the smallest node of the right subtree?
    - Left most node of right subtree
  - Move successor’s child (if any) where successor was. Example: Delete 50

![Binary Search Tree Diagram](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);   // compare key to node
    if (cmp < 0)
        x.left = delete(x.left, key);  // Search for key
    else if (cmp > 0)
        x.right = delete(x.right, key);
    else {                           // key found
        if (x.right == null)         // No right child
            return x.left;
        if (x.left == null)          // No left child
            return x.right;
        Node t = x;                   // replace with successor
        x = min(t.right);             // find successor - min of x.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.
- Overall, BSTs can have $O(n)$ worst-case for search, insert, and delete. We want to do better (see future lectures).
Lecture 19: 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
Lecture 19: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance

Some slides adopted from Algorithms 4th Edition or COS246
The story so far

- The symbol table/dictionary is a fundamental data type.
- Naive implementations (arrays/linked lists sorted or unsorted) are way too slow.
- Binary search trees work well in the average case, but can grow too tall and imbalanced in the worst case.

**Question of the day**: How to balance search trees?
Order of growth for symbol table operations

<table>
<thead>
<tr>
<th></th>
<th>Worst case</th>
<th></th>
<th></th>
<th>Average case</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>Goal</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>
2-3 SEARCH TREES

2-3 tree

- **Definition**: A 2-3 tree is either empty or a
  - 2-node: one key (and associated value) and two links, a left to a 2-3 search tree with smaller keys, and a right to a 2-3 search tree with larger keys (similarly to standard BSTs), or a
  - 3-node: two keys (and associated values) and three links, a left to a 2-3 search tree with smaller keys, a middle to a 2-3 search tree with keys between the node's keys, and a right to a 2-3 search tree with larger keys.

- **Symmetric order**: In-order traversal yields keys in ascending order.

- **Perfect balance**: Every path from root to null link (empty tree) has the same length.
Example of a 2-3 tree

- 2-node, business as usual with BSTs.
  - (e.g., EJ are smaller than M and R is larger than M).
- In 3-node,
  - left link points to 2-3 search tree with smaller keys than first key,
    - (e.g., AC are smaller than E.)
  - middle link points to 2-3 search tree with keys between first and second key,
    - (e.g., H is between E and J.)
  - right link points to 2-3 search tree with keys larger than second key.
    - (e.g., L is larger than J).
Lecture 24: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance
How to search for a key

- Compare search key against (every) key in node.
- Find interval containing search key (left, potentially middle, or right).
- Follow associated link, recursively.
3.3 2–3 Tree Demo

- search
- insertion
- construction
Lecture 24: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance
How to insert into a 2-node

- Add new key to 2-node to create a 3-node.
2–3 tree demo: insertion

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

insert K
How to insert into a tree consisting of a single 3-node

- Add new key to 3-node to create a temporary 4-node.
- Move middle key in 4-node into parent.
- Split 4-node into two 2-nodes.
- Height went up by 1.
How to insert into a 3-node whose parent is a 2-node

- Add new key to 3-node to create a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent.
- Replace 2-node parent with 3-node.
How to insert into a 3-node whose parent is a 3-node

- Add new key to 3-node to create a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent creating a temporary 4-node.
- Split 4-node into two 2-nodes and pass middle key to parent.
- Repeat up the tree, as necessary.
Splitting the root

- If end up with a temporary 4-node root, split into three 2-nodes.

- Increases height by 1 but perfect balance is preserved.
2–3 tree demo: insertion

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

insert K
Lecture 24: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance
2–3 tree demo: construction

insert R
Practice Time

- Draw the 2-3 tree that results when you insert the keys: E A S Y Q U T I O N in that order in an initially empty tree.
Answer

- EASYQUATION

https://www.cs.usfca.edu/~galles/visualization/BTree.html
Lecture 24: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance
Height of 2-3 search trees

- **Worst case**: $\log n$ (all 2-nodes).

- **Best case**: $\log_3 n = 0.631 \log n$ (all 3-nodes)
  - That means that storing a million nodes will lead to a tree with height between 12 and 20, and storing a billion nodes to a tree with height between 18 and 30 (not bad!).

- Search and insert are $O(\log n)$!

- But implementation is a pain and the overhead incurred could make the algorithms slower than standard BST search and insert.

- We did provide insurance against a worst case but we would prefer the overhead cost for that insurance to be low. Stay tuned! We will see a much easier way.
## Summary for symbol table/dictionary operations

<table>
<thead>
<tr>
<th></th>
<th>Worst case</th>
<th>Average case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td><strong>2-3 search trees</strong></td>
<td>log $n$</td>
<td>log $n$</td>
</tr>
</tbody>
</table>
Lecture 24: 2-3 Search Trees

- 2-3 Search Trees
- Search
- Insertion
- Construction
- Performance
ASSIGNED READINGS AND PRACTICE PROBLEMS

Readings:

▸ Textbook: Chapter 3.3 (Pages 424-431)

▸ Website:
  
  ▸ https://algs4.cs.princeton.edu/33balanced/

Practice Problems:

▸ 3.3.2-3.3.5