Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

Philip Bille

Nearest Neighbor

- Nearest neighbor. Maintain dynamic set $S$ supporting the following operations. Each element has key $x.key$ and satellite data $x.data$.
- $\text{PREDECESSOR}(k)$: return element with largest key $\leq k$.
- $\text{SUCCESSOR}(k)$: return element with smallest key $\geq k$.
- $\text{INSERT}(x)$: add $x$ to $S$ (we assume $x$ is not already in $S$)
- $\text{DELETE}(x)$: remove $x$ from $S$.

3 5 6 10 12 20 24

$\text{PREDECESSOR}(8) \quad k = 8 \quad \text{SUCCESSOR}(8)$

Nearest Neighbor

- Applications.
  - Searching for similar data (typically multidimensional)
  - Routing on the internet.

- Challenge. How can we solve problem with current techniques?
Nearest Neighbor

1. **Solution 1:** linked list. Maintain $S$ in a doubly-linked list.
   - **PREDECESSOR**($k$): linear search for largest key $\leq k$.
   - **SUCCESSOR**($k$): linear search for smallest key $\geq k$.
   - **INSERT**($x$): insert $x$ in the front of list.
   - **DELETE**($x$): remove $x$ from list.

   - **Time.**
     - **PREDECESSOR** and **SUCCESSOR** in $O(n)$ time ($n = |S|$).
     - **INSERT** and **DELETE** in $O(1)$ time.
   - **Space.**
     - $O(n)$.

2. **Solution 2:** Sorted array. Maintain $S$ in a sorted array.

   - **PREDECESSOR**($k$): binary search for largest key $\leq k$.
   - **SUCCESSOR**($k$): binary search for smallest key $\geq k$.
   - **INSERT**($x$): build new array of size +1 with $x$ inserted.
   - **DELETE**($x$): build new array of size -1 with $x$ removed.

   - **Time.**
     - **PREDECESSOR** and **SUCCESSOR** in $O(\log n)$ time.
     - **INSERT** and **DELETE** in $O(n)$ time.
   - **Space.**
     - $O(n)$.

### Data structure comparison

<table>
<thead>
<tr>
<th>Data structure</th>
<th>PREDECESSOR</th>
<th>SUCCESSOR</th>
<th>INSERT</th>
<th>DELETE</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>sorted array</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- **Challenge.** Can we do significantly better?
**Binary tree.** Rooted tree, where each internal vertex has a left child and/or a right child.

**Binary search tree.** Binary tree that satisfies the search tree property.

**Search tree property.**
- Each vertex stores an element.
- For each vertex v:
  - all vertices in left subtree are \( \leq v.\text{key} \).
  - all vertices in right subtree are \( \geq v.\text{key} \).

**Binary Search Trees**

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees

**Insertion**

- \textsc{Insert}(x): start in root. At vertex \( v \):
  - if \( x.\text{key} \leq v.\text{key} \) go left.
  - if \( x.\text{key} > v.\text{key} \) go right.
  - if null, insert \( x \)
Insertion

- **INSERT(x)**: start in root. At vertex v:
  - if x.key ≤ v.key go left.
  - if x.key > v.key go right.
  - if null, insert x

- **Exercise.** Insert following sequence in binary search tree: 6, 14, 3, 8, 12, 9, 34, 1, 7

---

Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- **Predecessor and Successor**
- Deletion
- Algorithms on Trees

---

Insertion

```
INSERT(x, v)
if (v == null) return x
if (x.key ≤ v.key)
    v.left = INSERT(x, v.left)
if (x.key > v.key)
    v.right = INSERT(x, v.right)
```

- **Time.** $O(h)$

---

Predecessor

- **PREDECESSOR(k)**: start in root. At vertex v:
  - if v == null: return null.
  - if k == v.key: return v.
  - if k < v.key: go left.
  - if k > v.key: search in right subtree.
    - If element x with key ≤ k in right subtree return x.
    - Otherwise, return v

---
**Predecessor**

\[
\text{PREDECESSOR}(v, k) =
\begin{cases}
\text{null} & \text{if } v = \text{null} \\
\text{v} & \text{if } v.\text{key} = k \\
\text{PREDECESSOR}(v.\text{left}, k) & \text{if } k < v.\text{key} \\
\text{PREDECESSOR}(v.\text{right}, k) & \text{if } k \geq v.\text{key}
\end{cases}
\]

- **Time.** $O(h)$
- **SUCCESSOR** with similar algorithm in $O(h)$ time.

---

**Deletion**

\[
\text{DELETE}(x):
\begin{cases}
0 \text{ children: remove } x. \\
1 \text{ child: } \text{splice } x. \\
2 \text{ children: find } y = \text{vertex with smallest key }> x.\text{key}. \text{Splice } y \text{ and replace } x \text{ by } y.
\end{cases}
\]

- **Time.** $O(h)$

---

**Binary Search Trees**

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees
### Nearest Neighbor

<table>
<thead>
<tr>
<th>Data structure</th>
<th>PREDECESSOR</th>
<th>SUCCESSOR</th>
<th>INSERT</th>
<th>DELETE</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked list</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>sorted array</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>binary search tree</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(n)</td>
</tr>
<tr>
<td>balanced binary search</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

- **Height**: Depends on sequence of operations.
  - \( h = \Omega(n) \) worst-case and \( h = \Theta(\log n) \) on average.
- **Balanced binary search trees**.
  - Possible to efficiently maintain binary search with height \( O(\log n) \) (2-3 tree, AVL-trees, red-black trees, ..)
  - Even better bounds possible with advanced data structures.

### Binary Search Trees

- **Nearest neighbor**
  - PREDECESSOR\( (k) \): return element with largest key \( \leq k \).
  - SUCCESSOR\( (k) \): return element with smallest key \( \geq k \).
  - INSERT\( (x) \): add \( x \) to \( S \) (we assume \( x \) is not already in \( S \))
  - DELETE\( (x) \): remove \( x \) from \( S \).
- **Other operations on binary search trees**.
  - SEARCH\( (k) \): determine if element with key \( k \) is in \( S \) and return it if so.
  - TREE-SEARCH\( (x, k) \): determine if element with key \( k \) is in subtree rooted at \( x \) and return it if so.
  - TREE-MIN\( (x) \): return the smallest element in subtree rooted at \( x \).
  - TREE-MAX\( (x) \): return the largest element in subtree rooted at \( x \).
  - TREE-PREDECESSOR\( (x) \): return element with largest key \( \leq x.key \).
  - TREE-SUCCESSOR\( (x) \): returner element with smallest key \( \geq x.key \).

### Algorithms on Trees

- **Previous algorithms**.
  - Heaps (MAX, EXTRACT-MAX, INCREASE-KEY, INSERT, ..)
  - Union find (INIT, UNION, FIND, ..)
  - Binary search trees (PREDECESSOR, SUCCESSOR, INSERT, DELETE, ..)
- **Challenge**: How do we design algorithms on binary trees?
**Algorithms on Trees**

- **Recursion on binary trees.**
  - Solve problem on $T(v)$:
    - Solve problem recursively on $T(v \text{.left})$ and $T(v \text{.right})$.
    - Combine to get solution for $T(v)$.

**Tree Traversals**

- **Inorder traversal.**
  - Visit left subtree recursively.
  - Visit vertex.
  - Visit right subtree recursively.
  - Prints out the vertices in a binary search tree in sorted order.

- **Preorder traversal.**
  - Visit vertex.
  - Visit left subtree recursively.
  - Visit right subtree recursively.

- **Postorder traversal.**
  - Visit left subtree recursively.
  - Visit right subtree recursively.
  - Visit vertex.

**Example.** Compute $\text{size}(v)$ (= number of vertices in $T(v)$).
- If $v$ is empty: $\text{size}(v) = 0$
- Otherwise: $\text{size}(v) = \text{size}(v\text{.left}) + \text{size}(v\text{.right}) + 1$.

**Time.** $O(\text{size}(v))$

**Inorder traversal**

```java
INORDER(v)
    if (v == null) return
    INORDER(v.left)
    print v.key
    INORDER(v.right)
```

**Preorder traversal**

```java
PREORDER(v)
    if (v == null) return
    print v.key
    PREORDER(v.left)
    PREORDER(v.right)
```

**Postorder traversal**

```java
POSTORDER(v)
    if (v == null) return
    POSTORDER(v.left)
    POSTORDER(v.right)
    print v.key
```

**Time.** $O(n)$
Binary Search Trees

- Nearest Neighbor
- Binary Search Trees
- Insertion
- Predecessor and Successor
- Deletion
- Algorithms on Trees