Binary Search Trees

Ch. 03 Binary Search Trees


Binary Search

- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children’s game
  - at each step, the number of candidate items is halved
  - terminates after $O(\log n)$ steps
- Example: find(7)
**Search Tables**

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- **Performance:**
  - Searches take $O(\log n)$ time, using binary search
  - Inserting a new item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to make room for the new item
  - Removing an item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for ordered maps of small size or for maps on which searches are the most common operations, while insertions and removals are rarely performed.

---

**Binary Search Trees**

- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items and can be either
  - a node without info or
  - null
- An inorder traversal of a binary search trees visits the keys in increasing order
Search

- To search for a key \( k \), we trace a downward path starting at the root.
- The next node visited depends on the comparison of \( k \) with the key of the current node.
- If we reach an external node, the key is not found.
- Example: get(4):
  - Call TreeSearch(4, root)
  - The algorithms for nearest neighbor queries (predecessor and successor) are similar.

```
Algorithm TreeSearch(k, v)
if isExternal(v)
  return v  // v is null or empty node
if k < key(v)
  return TreeSearch(k, leftChild(v))
else if k = key(v)
  return v  // key(v) = k.
else // k > key(v)
  return TreeSearch(k, rightChild(v))
```

Minimum & Maximum

- The minimum node is null if the root is null; otherwise, it is the leftmost node.
- The maximum node is null if the root is null; otherwise, it is the rightmost node.

```
Algorithm TreeMinimum(v)
if isExternal(v)
  return v  // v is null or empty node
if isExternal(v, leftChild(v))
  return v
return TreeMinimum(k, leftChild(v))
```
Insertion

- To perform operation `put(k, o)`, we search for key `k` (using TreeSearch)
- Assume `k` is not already in the tree, and let `w` be the leaf reached by the search
- We insert `k` at node `w` and expand `w` into an internal node
- Example: insert 5

Algorithm `insertion(k, v)`

- **input**: insert key `k` into the tree rooted by `v`
- **output**: the tree root with `k` adding to `v`
- if `isExternal(v)`
  - return `newInternalNode(k)`
- if `k ≤ key(v)` // duplicate keys are okay
  - `leftChild(v) ← insertion(k, leftChild(v))`
- else if `k > key(v)`
  - `rightChild(v) ← insertion(k, rightChild(v))`
- return `v`
Deletion

- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4

Deletion (cont.)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy key(w) into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z) if external nodes are empty nodes.
- Example: remove 3
**Deletion**

Algorithm *deletion*(k, v)

**input:** delete the node containing key k

**output:** the tree without k.

if *isExternal* (v)

    return v

if k < key(v) // duplicate keys are okay

    leftChild(v) ← *deletion* (k, leftChild(v))

else if k > key(v)

    rightChild(v) ← *deletion* (k, rightChild(v))

else if *isExternal* (leftChild(v))

    return rightChild(v)

else if *isExternal* (rightChild(v))

    return leftChild(v)

node min ← *treeMinimum* (rightChild(v))

key(v) ← key(min)

rightChild(v) ← *deletion* (key(min), rightChild(v))

return v

---

**Performance**

- Consider an ordered map with *n* items implemented by means of a binary search tree of height *h*:
  - the space used is *O*(n)
  - methods get, put and remove take *O*(h) time

- The height *h* is *O*(n) in the worst case and *O*(log *n*) in the best case
Range Queries

- An additional operation that can be answered by a binary search tree is a **range query**: `findAllInRange(k_1, k_2)`: Return all the elements stored in T with key k such that \( k_1 \leq k \leq k_2 \).

- Example: Find all cars on eBay priced between $10,000 and $15,000.

- Algorithm:
  - \( \text{key}(v) < k_1 \): We recursively search the right child of \( v \).
  - \( k_1 \leq \text{key}(v) \leq k_2 \): We report \( \text{element}(v) \) and recursively search both children of \( v \).
  - \( \text{key}(v) > k_2 \): We recursively search the left child of \( v \).

Pseudo-code

- **Range-query algorithm:**

```plaintext
Algorithm RangeQuery(k_1, k_2, v):
  Input: Search keys k_1 and k_2, and a node v of a binary search tree T
  Output: The elements stored in the subtree of T rooted at v whose keys are in the range [k_1, k_2]
  if T.isExternal(v) then
    return Ø
  if k_1 \leq \text{key}(v) \leq k_2 then
    L ← RangeQuery(k_1, k_2, T.leftChild(v))
    R ← RangeQuery(k_1, k_2, T.rightChild(v))
    return L \cup \{\text{element}(v)\} \cup R
  else if \text{key}(v) < k_1 then
    return RangeQuery(k_1, k_2, T.rightChild(v))
  else if k_2 < \text{key}(v) then
    return RangeQuery(k_1, k_2, T.leftChild(v))
```

14
Visualization

- Drawing subtrees as triangles, then we visit all the shaded subtrees.

Example

- An example shows that we also need to test for the nodes we visit along the search paths for \( k_1 \) and \( k_2 \).
Types of Nodes that We Visit

- Types of nodes that we visit:
  - Let $P_1$ be the path from the root to $k_1$.
  - Let $P_2$ be the path from the root to $k_2$.

- Node $v$ is a **boundary node** if $v$ belongs to $P_1$ or $P_2$; a boundary node stores an item whose key may be inside or outside the interval $[k_1, k_2]$.

- Node $v$ is an **inside node** if $v$ is not a boundary node and $v$ belongs to a subtree rooted at a right child of a node of $P_1$ or at a left child of a node of $P_2$; an internal inside node stores an item whose key is inside the interval $[k_1, k_2]$.

- Node $v$ is an **outside node** if $v$ is not a boundary node and $v$ belongs to a subtree rooted at a left child of a node of $P_1$ or at a right child of a node of $P_2$; an internal outside node stores an item whose key is outside the interval $[k_1, k_2]$.

Performance

- Let $h$ denote the height of the binary search tree, $T$, and let $s$ be the number of elements in the range.

  - We visit no outside nodes.
  - We visit at most $2h + 1$ boundary nodes, where $h$ is the height of $T$, since boundary nodes are on the search paths $P_1$ and $P_2$ and they share at least one node (the root of $T$).
  - Each time we visit an inside node $v$, we also visit the entire subtree $T_v$ of $T$ rooted at $v$ and we add all the elements stored at internal nodes of $T_v$ to the reported set. If $T_v$ holds $s_v$ items, then it has $2s_v + 1$ nodes. The inside nodes can be partitioned into $j$ disjoint subtrees $T_1, \ldots, T_j$ rooted at children of boundary nodes, where $j \leq 2h$. Denoting with $s_i$ the number of items stored in tree $T_i$, we have that the total number of inside nodes visited is equal to
    \[
    \sum_{i=1}^{j}(2s_i + 1) = 2s + j \leq 2s + 2h.
    \]
  - Therefore, at most $2s + 4h + 1$ nodes of $T$ are visited and the operation `findAllInRange` runs in $O(h + s)$ time.
Index-Based Searching (Selection)

- Add a new operation:
  - select(i): Return the item with i-th smallest key, for $1 \leq i \leq n$.

- Main idea to support this new method:
  - Augment each node $v$ to store $n_v$, the number of elements in the subtree rooted at $v$.

Maintaining the New Fields

- We must now update $n_v$ fields when we do an insertion or deletion.
  - If we are doing an insertion at a node, $w$, in $T$ (which was previously an external node), then we set $n_w = 1$ and we increment the $n_v$ count for each node $v$ that is an ancestor of $w$, that is, on the path from $w$ to the root of $T$.
  - If we are doing a deletion at a node, $w$, in $T$, then we decrement the $n_v$ count for each node $v$ that is on the path from $w$'s parent to the root of $T$. 
Insertion Update Example

Updating the counts for inserting an element with key 27.

Search Algorithm

We can do a search based on the rank, \( i \), for the \( i \)-th smallest element.

**Algorithm** TreeSelect\((i, v, T)\):

*Input:* Search index \( i \) and a node \( v \) of a binary search tree \( T \)

*Output:* The item with \( i \)-th smallest key stored in the subtree of \( T \) rooted at \( v \)

Let \( w \leftarrow T\cdot\text{leftChild}(v) \)

if \( i \leq n_w \) then

    return TreeSelect\((i, w, T)\)

else if \( i = n_w + 1 \) then

    return (\text{key}(w), \text{element}(w))

else

    return TreeSelect\((i - n_w - 1, T\cdot\text{rightChild}(v), T)\)
Example

A search for the 10th smallest element.