Binary Search Trees

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού
Κώστας Χατζηκοκολάκης

Search

• Searching for a specific value within a large collection is fundamental
• We want this to be efficient even if we have billions of values!
• So far we have seen two basic search strategies:
  - **sequential** search: slow
  - **binary** search: fast
  ◦ but only for **sorted** data

## Sequential search

We already saw that the complexity is $O(n)$.

```c
int sequential_search(int target, int array[], int size) {
    for (int i = 0; i < size; i++)
        if (array[i] == target)
            return i;

    return -1;
}
```

## Binary search

```c
int binary_search(int target, int array[], int size) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int middle = (low + high) / 2;

        if (target == array[middle])
            return middle; // βρέθηκε
        else if (target > array[middle])
            low = middle + 1; // συνεχίζουμε στο πάνω μισό
        else
            high = middle - 1; // συνεχίζουμε στο κάτω μισό
    }

    return -1;
}
```

**Important**: the array needs to be **sorted**
Binary search example
At each step the search space is cut in half.

Complexity of binary search

- **Search space**: the elements remaining to search
  - those between `low` and `right`
- The size of the search space is cut in half at each step
  - After step \( i \) there are \( \frac{n}{2^i} \) elements remaining
- We **stop** when \( \frac{n}{2^i} < 1 \)
  - in other words when \( n < 2^i \)
  - or equivalently when \( \log n < i \)
- So we will do at most \( \log n \) steps
  - complexity \( O(\log n) \)
  - **30 steps** for one billion elements

Conclusions

- Binary search is fundamental for efficient search
- But we need **sorted data**
- Maintaining a sorted array **after an insert** is hard
  - complexity?
- How can we keep data sorted and **simultaneously** allow efficient inserts?
Binary Search Trees (BST)

A binary search tree (δυαδικό δέντρο αναζήτησης) is a binary tree such that:

- every node is larger than all nodes on its left subtree
- every node is smaller than all nodes on its right subtree

Note

- No value can appear twice (it would violate the definition)
- Any compare function can be used for ordering.
  (with some mathematical constraints, see the piazza post)

Example

A different tree with the same values!
BST operations

- Container operations
  - **Insert / Remove**
- **Search** for a given value
- **Ordered** traversal
  - Find **first / last**
  - Find **next / previous**
- So we can use BSTs to implement
  - **ADTMap** (we need search)
  - **ADTSet** (we need search and ordered traversal)

Search

We perform the following procedure **starting at the root**

- If the tree is empty
  - **target** does not exist in the tree
- If **target = current_node**
  - Found!
- If **target < current_node**
  - continue in the **left subtree**
- If **target > current_node**
  - continue in the **right subtree**

Search example

Searching for 8

Found 8
**Complexity of search**

- How many steps will we make in the worst case?
  - We will traverse a path from the root to the tree
  - $h$ steps max (the **height** of the tree)
- But how does $h$ relate to $n$?
  - $h = O(n)$ in the worst case!
  - when the tree is essentially a degenerate “list”

**Searching in this tree is slow**

- This is a very common pattern in trees
  - Many operations are $O(h)$
  - Which means worst-case $O(n)$
- Unless we manage to **keep the tree short**!
  - We already saw this in **complete** trees, in which $h \leq \log n$
- Unfortunately maintaining a complete BST is not easy (why?)
  - But there are other methods to achieve the same result
    - AVL, B-Trees, etc
  - We will talk about them later
Inserting a new value

• Inserting a value is very similar to search
• We follow the same algorithm as if we were searching for value
  - If value is found we stop (no duplicates!)
  - If we reach an empty subtree insert value there

Insert example

Inserting e

Insert example

Inserting b
Inserting d
Inserting f
Inserting a
Inserting g
**Insert example**

Inserting c

**Complexity of insert**

- Same as search
- $O(h)$
  - So $O(n)$ unless the tree is short

**Deleting a value**

- We might want to delete any node in a BST
- Easy case: node has as most 1 child
- Connect the child directly to node’s parent
- BST property is preserved (why?)

**Deleting a value**

- Hard case: node has two children (eg. 10)
- Find the next node in the order (eg. 12)
  - left-most node in the right sub-tree!
    (or equivalently the previous node)
- We can replace node’s value with next’s
  - this preserves the BST property (why?)
- And then delete next
  - This has to be an easy case (why?)
Delete 4 (easy).

Delete 10 (hard). Replace with 7 and it becomes easy.

Finding the node to delete is $O(h)$

Finding the next/previous is also $O(h)$
Ordered traversal: first/last

- How to find the first node?
  - simply follow left children
  - \( O(h) \)
  - same for last

Ordered traversal: next

- How to find the next of a given node?
  - Easy case: the node has a right child
    - find the left-most node of the right subtree
    - we used this for delete!
  - Hard case: no right-child, we need to go up!

General algorithm for any node.
Perform the following procedure starting at the root

```c
find_next(current_node, node) {
    if (node == current_node) {
        // the target is the root, the next is the first node of its right subtree (or NULL if none)
        return node_find_min(right_child);
    } else if (node > current_node) {
        // if the target is in the left subtree, return the next in order
        return node_find_next(node->right, compare, target);
    } else {
        // if the target is in the right subtree, we need to go up
        res = node_find_next(node->left, compare, target);
        return res != NULL ? res : node;
    }
}
```

Complexity of next

- Similar to search, traversing the tree from the root to the leaves
  - so \( O(h) \)
- We can do it faster by keeping more structure
  - We can keep a bidirectional list of all nodes in order
    - \( O(1) \) to find next, no extra complexity to update
- More advanced: keep a link to the parent
  - Find the next by going up when needed
  - Can you find the algorithm?
  - Real-time complexity is still \( O(h) \) if we traverse to the root
  - But what about amortized-time?
Rotations

- **Rotation (περιστροφή)** is a fundamental operation in BSTs
  - swaps the role of a node and one of its children
  - while still preserving the BST property

- **Right rotation**
  - swap a node $h$ and its left child $x$
  - $x$ becomes the root of the subtree
  - the right child of $x$ becomes left child of $h$
  - $h$ becomes a right child of $x$

- **Left rotation**
  - symmetric operation with right child

---

**Example: right rotation**

```
A
  E  h
     \\
    x
C
  R  Y
    \\
   h
H
```

```
A
  E
     \\
    x
C
  R  Y
    \\
   h
H
```

```
A
  E
     \\
    x
C
  R  Y
    \\
   h
H
```

```
A
  E
     \\
    x
C
  R  Y
    \\
   h
H
```
Example: left rotation

Complexity of rotation

• Only changing a few pointers
• No traversal of the tree!
• So $O(1)$
Root insertion

- Goal
  - insert a new element
  - place it at the root of the tree
- Simple recursive algorithm using rotations
  1. If empty: trivial
  2. Recursively insert in the left/right subtree
     - depending on whether the value is smaller than the root or not
     - after the recursive call finishes we have a proper BST
     - with the value as the root of the left/right subtree
  3. Rotate left or right
     - the value comes at the root!

Example: root insertion

We are inserting G. The recursive algorithm is first called on the root A, then it makes recursive calls on the right subtree S, then on E, R, H, and finally a recursive call is made on the empty left subtree of H.

G is inserted in the empty left subtree of H.

The call on H does a right rotation, G moves up.
The call on R does a right rotation, G moves up.

The call on E does a left rotation, G moves up.

The call on R does a right rotation, G moves up.

The call on A does a left rotation, G arrives at the root.
**Complexity of root insertion**

- The algorithm is similar to a normal insert
  - traversing the tree towards the leaves: $O(h)$
- With an **extra rotation** at every step
  - which is $O(1)$
- So still $O(h)$

**Readings**

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*
  - Chapter 5. Sections 5.6 and 6.5.
  - Chapter 9. Section 9.7.
- R. Sedgewick. Αλγόριθμοι σε C.
  - Κεφ. 12.
  - Section 9.3 and 10.1