Self-balancing Binary Search Trees

AVL tree, Treap and Ordered statistics tree

Yaseen Mowzer

3rd IOI Training Camp 2017 (4 March 2017)

Outline

What is a binary search tree?

Definition

For each node in a binary search tree, the value of a left child is less than its parent and the value of the right child is greater than or equal to its parent.

What is a binary search tree?

Definition

For each node in a binary search tree, the value of a left child is less than its parent and the value of the right child is greater than or equal to its parent.

Operations

- Search
- Insert
- Boundry search (e.g. Find the largest element $< v$)
- Delete

How do we structure a BST

We use a recursive data structure.

```
struct node {
    T value;
    node *left;
    node *right;
    node (T val) : value {val} {}
};
```

How do we structure a BST

We use a recursive data structure.

```
struct subtree {
  T value_of_root;
  subtree *left_subtree;
  subtree *right_subtree;
  subtree (T val) : value_of_root {val} {}
};
```

How to find an element in a BST

```
node *search(node *p, int value) {
   if (p == NULL)
        return NULL;
   else if (value < p->value)
        return search(p->left, value);
   else if (value > p->value)
        return search(p->right, value);
   else
        return p;
}
```

How to add an element to a BST

```
void insert(node **u, int value) {
  node *p = *u;
  if (p == NULL)
    *u = new node(value);
  else if (value < p->value)
    insert(&p->left, value);
  else
    insert(&p->right, value);
}
```

Similarly we can write a boundry find function

Find the smallest value $> k$

```
node *lower_bound(node *p, int value) {
  if (p == NULL) return NULL;
  if (value < p->value) {
    if (p->left == NULL) return p;
    return lower_bound(p->left, value);
  } else {
    if (p->right == NULL) return p;
    return lower_bound(p->right, value);
  }
}
```

Time complexity

- Insertion: $O(\log N)$ if insertions are random
- Search: $O(\log N)$ if insertions were random
- Boundry find: $O(\log N)$ if inserts were random

Time complexity

- Insertion: O(log N) if insertions are random
- Search: O(log N) if insertions were random
- Boundry find: O(log N) if inserts were random

But what if the insertions aren't random?

Binary search trees can be horribly unbalanced

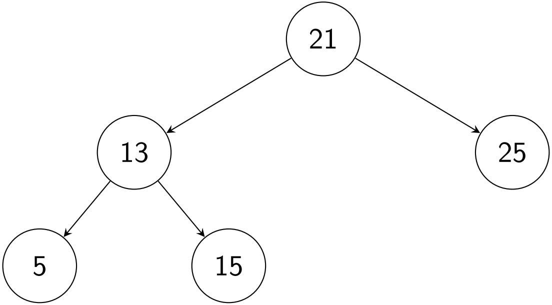


Figure: A well balanced tree

Binary search trees can be horribly unbalanced

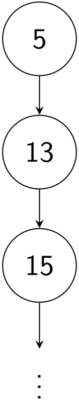
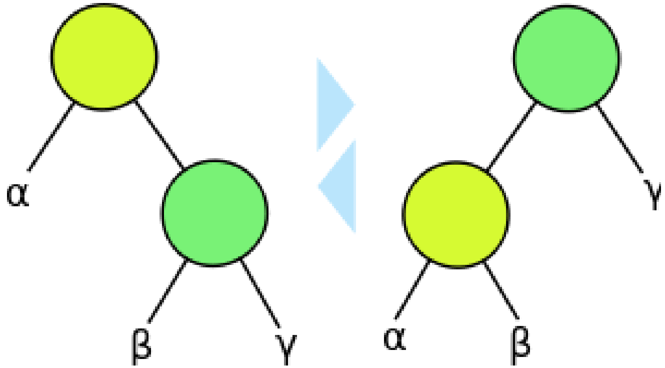


Figure: A terribly balanced tree

Operations on this tree are $O(N)$

Trees can be balanced with tree rotations



- α, β, γ could be `NULL` but the yellow and green nodes must not be `NULL`
- Tree rotations preserve the binary search tree property, but change the height

How to rotate

```
void rotate_left(node **u) {
  node *a = *u;
  node *b = a->right;
  a->right = b->left;
  b->left = a;
  *u = b;
}
void rotate_right(node **u) {
  node *a = *u;
  node *b = a->left;
  a->left = b->right;
  b->right = a;
  *u = b;
}
```

What about deletion

- Deletion is hard
- Rather mark a node as deleted if time complexity allows
- Otherwise rotate a node to a leaf and then remove it.

The red black tree invariants

1. Each node is either red or black.
2. The root is black.
3. The leaves are all NULL pointers and they are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NULL nodes contains the same number of black nodes.

From 4 and 5 we can intuitively see that the longest height will be twice as long as the shortest height.

Where and how is it used

- Used by C++ set, map.
- Fast insertions
- Query a bit slow since tree is not perfectly balanced
- Difficult to code, rather use another tree

Where and how is it used

- Used by C++ set, map.
- Fast insertions
- Query a bit slow since tree is not perfectly balanced
- Difficult to code, rather use another tree

But if C++ has a red-black trees why bother coding your own?

Ordered statistics trees

set and map are missing two important functions.

- Select (Find the element that is greater than exactly n other elements)
- Rank (Find the number of elements less than this element)

We need to augment our tree with more information

```
struct T {
    T value;
    node *left;
    node *right;
    int size;
}
```

C++ doesn't allow augmentation so we have to write our own BBST

Select code

```
node *select(node *p, int index) {
   if (index == p->left->size) {
       return p;
   } else if (index < p->left->size) {
       return select(p->left, index);
     } else {
       int r = index - p->left->size - 1
       return select(p->right, r);
   }
 }
```

Rank

```
int rank(node *p, int value) {
  if (value <= p->value) {
    return rank(p->left, value);
  } else {
    return 1 + p->left->size
             + rank(p->right, value);
  }
}
```

AVL

- AVL trees are a type of balanced binary search tree that is reasonable to code.
- They are very rigidly balanced
- This means querying is fast, but inserting is slow
- Use when you have a high query to insertion ratio.

Augmentation

```
struct node {
  // Binary tree stuff
  int height;
}
```

AVL tree invariant

- In each subtree let h_l be the height of the left subtree and h_r be the height of the right subtree then $|h_l - h_r| < 2$.
- This means the difference in height is at most 2.
- Note this is stronger than a red black tree

Update

```
void update(node *p) {
  // Other things such as size for OS-trees
  p->height = 1 + max(p->left->height,
                      p->right->height);
}
```

Insertion

```
int hdiff(node *u) {
  return u->right->height - u->left->height;
}
void insert(node **u, int val) {
  // Normal insertion
  update(*u);
  // Fix up
  if (hdiff(*u) < -1) { // Leans left
    if (hdiff((*u)->left)> 1) { // Leans inner right
      rotate_left(&(*u)->left); update((*u)->left);
    }
    rotate_right(u); update(*u);
  } else if (hdiff(*u) > 1) { // Leans right
    if (hdiff((*u)->right) < -1) { // Leans inner left
      rotate_left(&(*u)->right); update((*u)->right);
    }
    rotate_left(u); update(*u);
  }
}
```

Heaps

- The next data structure requires us to understand heaps.

Heaps

- The next data structure requires us to understand heaps.
- C++ uses a binary max-heap for a priority queue.

Heaps

- The next data structure requires us to understand heaps.
- C++ uses a binary max-heap for a priority queue.
- A tree satisfies the heap property if every node in the tree is less than each of its children. (min heap)

Heaps

- The next data structure requires us to understand heaps.
- C++ uses a binary max-heap for a priority queue.
- A tree satisfies the heap property if every node in the tree is less than each of its children. (min heap)
- Corollary: By transitivity, every node in a heap is less than all of its descendants.
- Corollary: The root is the smallest element.
- Generally to update a heap we add an at some leaf point and then "swap" a child with the parent recursively if the parent is greater than the child.

Treaps!

- Recall that when the values are randomly inserted the binary search tree is balanced
- We want to simulate "random insertion".

Combine a heap and a binary search tree!

```
struct node {
  // other node data
  int priority;
};
```

We add a random priority to cause random rotations.

Insertion

```
void insert(node **u, int value) {
  // Insert node with random priority

  // Fixup
  if ((*u)->left->priority
    < (*u)->priority) {
    rotate_right(u);
  } else if ((*u)->right->priority
           < (*u)->priority) {
    rotate_left(u);
  }
}
```

Why use a treap

- Probabilistic balancing results in two rotations on average.
- Fast insertion.
- Not as fast querying
- Use when query to insertion ratio is not high.

Cool stuff you can do with treaps

- Deletion
  - Give a node a low priority and let it bubble to the bottom.
  - Easily remove a leaf node.
- Split
  - Create two treaps such that
    - All nodes in the left treap is less than v.
    - All nodes in the right treap are greater than or equal to v.
  - Give a node (v) with a high priority, bubble it to the top.
  - The left node has all elements less than v.
  - The right node has all elements greater than or equal v.