Cartesian tree, also knwon as Treaps
Theory and applications

Gleb Evstropov
Andrew Stankevich

September 27, 2019

1 Some notations

- u, v, w — some nodes of the binary search tree;
- $parent(v)$ — the parent of some node v in the binary search tree. If v is the root then $parent(v) = NIL$;
- $left(v)$ — left child of some node v in the binary search tree. If the left subtree is empty, then $left(v) = NIL$;
- $right(v)$ — right child of some node v in the binary search tree. If the right subtree is empty, then $right(v) = NIL$;
- $key(v)$ — the value of a node v that affects the tree structure;
- $x(v)$ — another way to denote keys in Cartesian trees. Usually, $x(v) = key(v)$.
- $y(v)$ — some additional value associated with the node v and used to build the tree;
- $subtree(v)$ — the set of all nodes that lie inside the subtree of some node v (v is also included);
- $size(v)$ — the size of the subtree of some node v;
- $x_l(v)$ — the minimum key in the subtree of the node v, that is:
$$x_l(v) = \min_{u \in subtree(v)} key(u)$$
- Same as $x_l(v)$ we define $x_r(v)$ as the maximum key in the subtree of the node v:
$$x_r(v) = \max_{u \in subtree(v)} key(u)$$
- $depth(v)$ is the length of the path from $root$ to v. $depth(root) = 0$.
- $height(v)$ is the difference $max(depth(u)) - depth(v)$, where $u \in subtree(v)$.

2 Key points and definitions

- Greedy algorithm of finding an increasing subsequence: take first element that is greater than current, "left ladder". The expected length of the result on a random permutation is $O(\log n)$.

- BST stands for *binary search tree*, that is a binary rooted tree with some keys associated with every node, and the following two conditions hold:

$$key(u) < key(v), \forall u, v : u \in subtree(left(v))$$

  and

$$key(u) > key(v), \forall u, v : u \in subtree(right(v))$$

- For any pair of nodes of any binary search tree v and u:
  $u \in subtree(v)$ if and only if $x_l(v) \leq key(u) \leq x_r(v)$

- For any tree and some keys stored in nodes of that tree we say that *heap condition* holds if for any v that is not the root:

$$key(parent(v)) \geq key(v)$$

- Binary search tree of size n is balanced if it's height is $O(\log n)$.

- *Cartesian tree* or *treap* is a balanced binary search tree, where each node is assigned some random values $y(v)$, which satisfy to the heap condition. Hereafter we will treat $y(v)$ as a random permutation.

- Cartesian tree is uniquely determined by a set of pairs (x_i, y_i), such that all x_i are pairwise distinct and all y_i are pairwise distinct.

- Node v is an ancestor of a node u if and only if for every $w \neq v$ such that $min(key(v), key(u)) \leq key(w) \leq max(key(v), key(u))$ it's y is smaller than the y of v, i.e. $y(v) > y(w)$.

- Linear algorithm to build Cartesian tree having a sorted pairs using stack.

- The expected depth of an i-th node (in the order of left-right traversal) is

$$\sum_{j=0}^{j<n} \frac{1}{|j-i|+1} \leq 2 \cdot \sum_{j=1}^{j \leq n} \frac{1}{j} = O(\log n)$$

  .

3 Some problems to think about

- Prove that if in merge operation one picks the root equiprobable (i.e. both l and r has probability 0.5 to become the root), the expected height is $\frac{n}{2}$.
- Prove that if in merge operation one picks the root proportional to the height (i.e. l has probability $\frac{height(l)}{height(l)+height(r)}$ to become the root), the expected height is $\sqrt{n}$.
- Prove that it's impossible to build Cartesian tree in linear time if the set of keys is not-sorted.
- Prove that it's impossible to merge two Cartesian trees of size n in $o(n)$ time, if there are no guarantees on key ranges.
- What is the expected height of the tree if we pick y's as integers in range from 0 to p? Consider the case $p = 1$ first.
- Can you prove $O(\log n)$ expectation for $height(root)$?

4 Operations with treaps and some pseudocode snippets

```
Tree find(Tree a, Key v)
    if a == NIL then
        return NIL
    if x(a) == v then
        return a
    else if x(a) < v then
        return find(right(a), v)
    else
        return find(left(a), v)

Tree merge(Tree a, Tree b)
    if a == NIL then return b
    if b == NIL then return a
    if y(a) < y(b) then
        right(a) = merge(right(a), b)
        return a
    else
        left(b) = merge(a, left(b))
        return b
```

- Why is it incorrect to write `left(b) = merge(left(b), a)`?

- Note that $x(v)$ is not used in merge

```
pair<Tree, Tree> split(Tree a, Key s) // first < s <= second
    if a == NIL then return (NIL, NIL)
    if x(a) < s then
        (u, v) = split(right(a), s)
        right(a) = u
        return (a, v)
    else
        (u, v) = split(left(a), s)
        left(a) = v
        return (u, a)
```

- What if we want `first <= s < second`?
- We can now insert a new value.

```
Tree insert(Tree a, Key nv)  // assume nv not in a
    (u, v) = split(a, nv)
    Tree t = Tree(key = nv, y = random, left = right = NIL)
    u = merge(u, t)
    u = merge(u, v)
    return u
```

- Develop another version of insert that only uses one split and no merges. Hint: descend as searching for an element to insert, until you cannot go down because of heap property. Insert here.
- We can now remove a value.

```
Tree remove(Tree a, Key rv)
    (u, v) = split(a, rv)
    (NIL, v) = split(v, rv + 1)
    a = merge(u, v)
    return a
```

- Develop another version of remove that only uses one mege and no splits. Hint: find an element and remove it. What should we do with its children?
- How can we remove if we cannot consider $rv + 1$ (example: floating point numbers, strings)?
- Find the k-th element if we store subtree sizes (indexed from 0).

```
Tree find_kth(Tree a, int k)
    if size(left(a)) == k then
        return a
    else if size(left(a)) < k then
        return find(right(a), k - size(left(a)) - 1)
    else
        return find(left(a), k)
```

- We can also cut away the first k elements if we store subtree sizes.

```
pair<Tree, Tree> split_k(Tree a, int k) // size(first) = k
    if a == NIL then return (NIL, NIL) // assert k == 0
    if size(left(a)) >= k then
        (u, v) = split_k(left(a), k)
        left(a) = v
        return (u, a)
    else
        (u, v) = split(right(a), k - size(left(a)) - 1)
        right(a) = u
        return (a, v)
```

- Note that now both `split_k` and `merge` don't use $x(v)$.
- We can make a similar functions that insert the element to the k-th position or remove the element from the k-th position.
- We can remove $x(v)$ and store any $z(v)$ instead. Now we get a data structure that supports the following operations:
  - maintain an array $z[0..n-1]$
  - set/get element $z[i]$
  - insert element after $z[i]$
  - remove element $z[i]$
  - concatenate two arrays
  - split array into two by cutting away the first k elements
- It is sometimes called a *rope*.