Treaps

Salvador Roura

Modern languages like C++ have powerful libraries that implement sets and maps. The typical operations include inserting and erasing keys, exact search for keys (*find* ()), approximate search (*lower_bound*(), *upper_bound*()), full set and map traversals, increasing and decreasing iterators, etc.

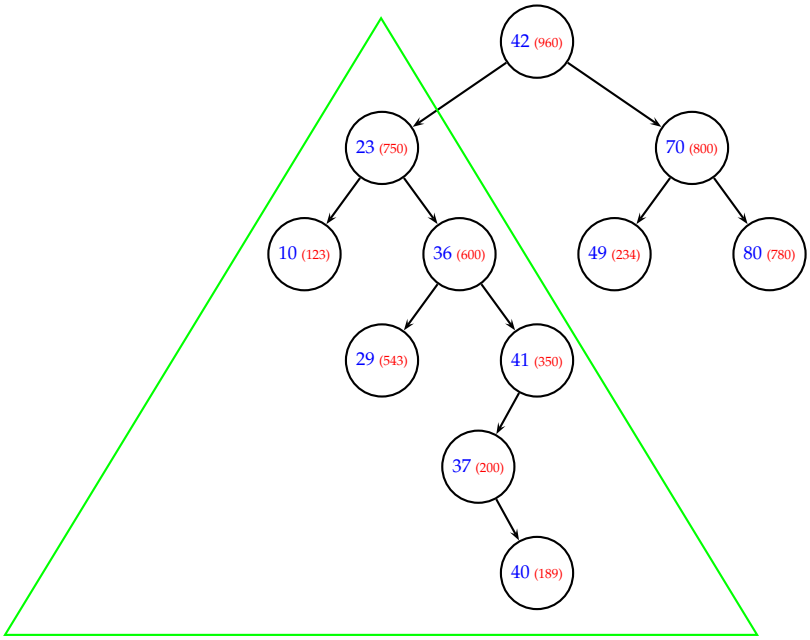
However, sometimes we require extra operations that are not included in the standard libraries. For instance, assume that we need rank operations, like computing the rank of a given key, or finding the i-th key of a set or map. In those cases, it may be necessary to manually implement some variant of balanced binary search tree.

There are many options, among them:

- Splay trees. Nice from a theoretical point of view, but a bit harder to program than other options, with amortized (not worst-case) logarithmic cost bounds, and a large constant in practice.
- Randomized BSTS. Perhaps the easiest to implement. However, RBSTs provide average (not worst-case) logarithmic cost bounds, and a constant that may be large in practice, because of the heavy use of random numbers, which are not cheap.
- Red-black/AVL trees. Logarithmic worst-case bounds, and the smallest constant in practice. But not the simplest to program.
- Treaps. Almost as easy to implement as randomized BSTs, with the same theoretical cost bounds, but a relatively small constant in practice. Treaps are the default choice for manually implemented sets/maps for most participants of programming competitions.

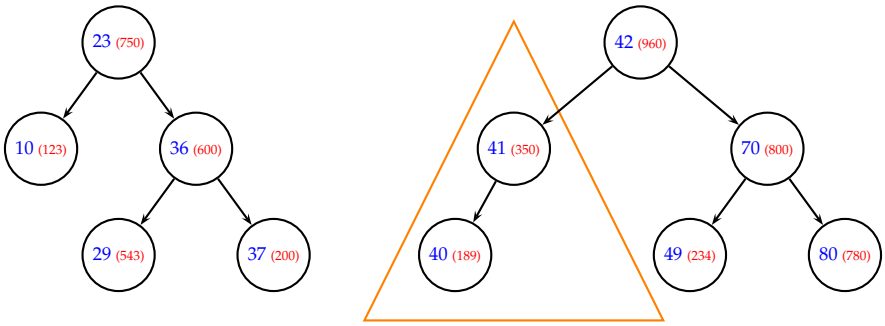
What is a treap? As the name tries to suggest, it is a tree and also a heap. Every inserted key is associated with a random priority (say, an integer number). For simplicity, assume that all the associated priorities are different. Let x be the key with the largest priority. Then, the treap for the set of pairs (key, priority) is the only BST with x at its root, the unique treap for the pairs with keys smaller than x as its left subtree, and the unique treap for the pairs with keys larger than x as its right subtree.

For instance, suppose that the set of pairs is (10, 123), (23, 750), (29, 543), (36, 600), (37, 200), (40, 189), (41, 350), (42, 960), (49, 234), (70, 800), (80, 780). The unique treap is



If we only look at the keys (in blue), we have a BST. If we only look at the priorities (in red), we have a sort of a heap (the largest at the top; the same property holds recursively).

Treaps can be implemented in several ways. One popular option is to build most procedures on just two basic operations: split and merge. Splitting a treap with respect to a key x results in two treaps: one with the keys smaller than x, the other with the keys larger than x. (If x is present we remove it.) Note that those two treaps are built from the original treap, which is lost. For instance, splitting the treap above with respect to 38 produces these two treaps:



To merge is the opposite to split: Given two treaps, the first with keys smaller than the second, the result of merging them is the unique treap with all the keys together. For instance, merging the last two treaps produces again the first treap.

Fortunately, splitting and merging are easily implemented. Below you have C++ code for treaps. First, the definitions of types.

```
// a treap "is" the pointer to its root
using Treap = struct Node*;

struct Node {
  Key key;        // Key should be any comparable type
  int pri, cnt;   // priority, counter
  Treap l, r;     // left and right pointers
};
```

We include a *cnt* field for rank operations. It stores the number of keys of the (sub)treap rooted at each node. We are ready to implement the *split* () function.

```
// number of keys of the treap rooted at t
int size(Treap t) { return (t ? t->cnt : 0); }

// updates the counter of t
Treap update(Treap t) {
  t->cnt = 1 + size(t->l) + size(t->r);
  return t;
}

// splits t with respect to x; the result is placed in l and r
void split(Key x, Treap t, Treap &l, Treap &r) {
  if (t == 0) l = r = 0;
  else if (x < t->key) split(x, t->l, l, t->l), r = update(t);
  else if (x > t->key) split(x, t->r, t->r, r), l = update(t);
  else l = t->l, r = t->r, delete t;
}
```

The first two functions are auxiliary. Observe that we use plain 0s for null pointers.

To better understand the third function, consider the first treap of the previous page. When splitting at its root with respect to $x = 38$, the condition that evaluates to true is $x < t\text{->}key$. So, we recursively split the left subtreap of t (in green in the previous page), and place the resulting treaps at l, and as the left subtreap of t (in orange). Note that (the root of) t becomes (the root of) r after updating its *cnt* field.

Next, we have this code for merging treaps.

```
// merges l and r into one treap
Treap merge(Treap l, Treap r) {
  if (l == 0) return r;
  if (r == 0) return l;
  if (l→pri > r→pri) {
    l→r = merge(l→r, r);
    return update(l);
  }
  else {
    r→l = merge(l, r→l);
    return update(r);
  }
}
```

Consider again the examples on page 2. If l and r are the treaps at the bottom of the page, in the first step we enter the **else** branch, we choose the node with a 42 as the root of the treap, and we put the result of recursively merging l and the left subtreap of r (in orange), as the new left subtreap (in green) of r.

Now, it is easy to insert and erase keys:

```
mt19937 rng;

Treap erase(Key x, Treap t) {
  Treap l, r;
  split(x, t, l, r);
  return merge(l, r);
}

Treap insert(Key x, Treap t) {
  Treap l, r;
  split(x, t, l, r);
  return merge(merge(l, new Node{x, int(rng()), 1, 0, 0}), r);
}
```

Several comment are in order. First, *rng* is a random number generator of the type *mt19937*, included in the <*random*> library. It generates better (pseudo) random integers than the conventional *rand*() function, and much faster.

Erasing a key x from a treap t is trivial: First, we split t with respect to x. This step removes x (if it was present). Afterwards, we join l and r together. Inserting a key x into a treap t is not much harder: First, we split t with respect to x. Next, we merge l with a treap with a single node storing x. Finally, we join l (plus x) and r together. Note that, if x was present, the splitting phase would remove it, so that we would end with exactly one x in the treap (almost surely with a different priority, but who cares?).

We include the most typical rank functions, which make use of the *cnt* field. This code is independent on the variant of balanced tree that we use.

```
// returns the i-th key of t, assuming 0 ≤ i < size(t)
Key find_ith (int i, Treap t) {
  int s = size(t→l);
  if (i < s) return find_ith (i, t→l);
  if (i > s) return find_ith (i − s − 1, t→r);
  return t→key;
}

// computes the rank of x inside t
int compute_rank(Key x, Treap t) {
  if (t == 0) return 0;
  int s = size(t→l);
  if (x < t→key) return compute_rank(x, t→l);
  if (x > t→key) return s + 1 + compute_rank(x, t→r);
  return s;
}
```

The first procedure assumes that i is a correct value, that is, that the treap has at least $i + 1$ keys. Note that the rank is 0-based. For instance, calling *find_ith* () with $i = 3$ on the first treap in page 2 returns 36.

The second procedure is the opposite: Given a key, it computes its (0-based) rank. For instance, calling *compute_rank*() with $x = 36$ on the first treap in page 2 returns 3. Note that x may not be in the treap. In that case, it computes the rank that x would have if we inserted x in the treap. For instance, calling *compute_rank*() with $x = 34$ on the first treap in page 2 also returns 3.

The expected cost of each recursive function in these notes is logarithmic, and with high probability. As a matter of fact, measured as the number of visited nodes, the expected cost of *split* (), *merge*(), *erase* (), *insert* (), *find_ith* () and *compute_rank*() is exactly the same as for randomized BSTs.

C++ order statistic trees

An *order statistic tree* (OST) is a BST with its typical instructions, plus efficient rank operations, that is, exactly the data structure that we have just coded. Luckily, there is a C++ extension that directly implements an OST. Here we have an example that shows its usage.

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
#include <iostream>
using namespace std;

using OST = tree<string, int, less<string>, rb_tree_tag,
                 tree_order_statistics_node_update >;

int main() {
  OST T;
  T["hello"] = 30;
  T["bye"] = 20;
  T["hi"] = 10;

  for (int i = 0; i < 3; ++i) {
    auto it = T.find_by_order(i);
    cout << it->first << ' ' << it->second << endl;
  }

  cout << T.order_of_key("bye") << endl;
  cout << T.order_of_key("hello") << endl;
  cout << T.order_of_key("hi") << endl;
  cout << T.order_of_key("a") << endl;
  cout << T.order_of_key("g") << endl;
  cout << T.order_of_key("help") << endl;
  cout << T.order_of_key("i") << endl;
}
```

Note that we need two extra **include**s and one extra **namespace** shown at the top of the code. Afterwards, we can define the type *OST* as a *map*<**string**,**int**> with additional rank operations. We could also define a *set*<**string**> by using **null_type** as second parameter.

In the first part of the main we declare an object *T* of type *OST*, and we insert in *T* information for three words with a standard *map* operation.

Afterwards, we search by rank with all possible values for i from 0 to 2. Take into account that *find_by_order* (i) returns an iterator to the i-th node of the tree. Since T is implemented on top of *pair*<**string**, **int**>, to access the **string** key and the associated **int** value we use *first* and *second*. This is what the second part of the code prints:

```
bye 20
hello 30
hi 10
```

Finally, we ask for the ranks of several given keys, either present or absent. This is what the third part of the code prints:

```
0
1
2
0
1
2
3
```

A final comment: I do not know under which versions of C++ this code works. It does compile with the GNU C++ compiler in my computer under Ubuntu though.