Algorithm Tutorials

by Tanuj Khattar

tanujkhattar in Data Structures ◎ January 10, 2016January 16, 2017 ◎ 1,930 Words

Treaps : One Tree to Rule 'em all ..!! Part-2

Introduction : This is a continuation of my article on treaps (Link to Part1 (https://tanujkhattar.wordpress.com/2016/01/10/treaps-one-tree-to-rule-em-all-part-1/)). In this article, I would be covering Implicit Treaps.
Pr-requisites : Basics of treap(covered in part1). Interval trees (like segment tree or fenwick tree).

Implicit Treap : What all to expect ?

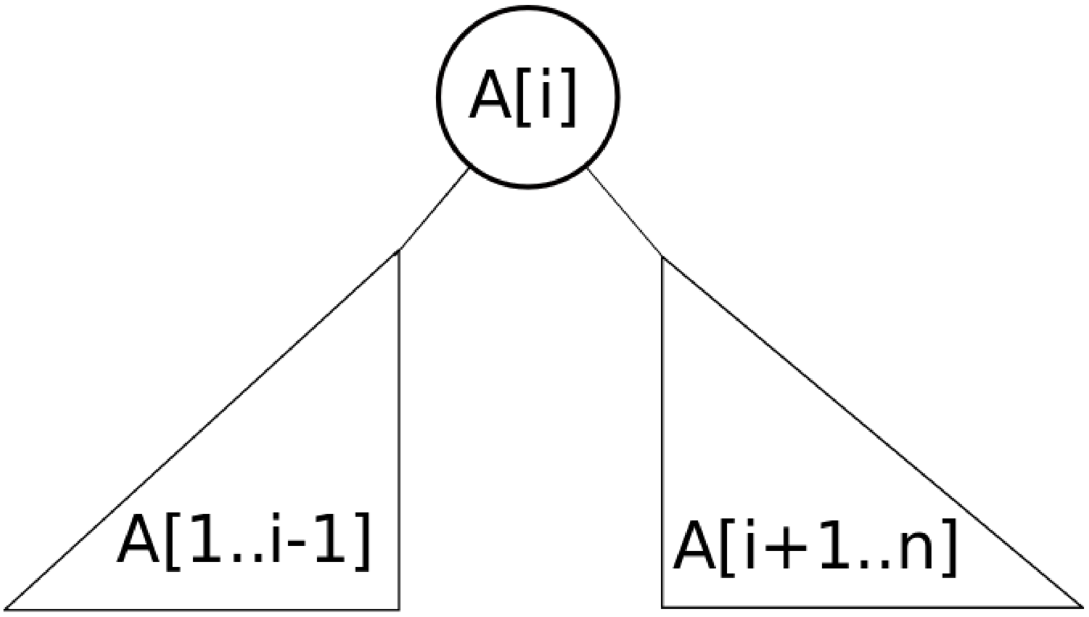
Implicit treap can be viewed as a dynamic array which supports the following operations , each in O(logN) time :

- Insert an element at any position.
- Delete an element at any position.
- Cut an array A[1..n] at any pos such that it is divided into two different arrays B[1..pos] , C[pos…n] .
- Merge two different arrays P[1..n1] , Q[1..n2] into a single array R[1..n1,n+1,…n2].
- Maintain any objective function and query over an arbitrary interval. (All the operations supported by a segment tree including range updates using lazy propagation).

In short, we get all the operations supported by a segment tree along with the power to split an array into two parts and merge two different arrays into a single one, both of them in O(logN) time.

How to implement ?

The key idea in implicit treap is that now , we use the array indices of elements as keys , instead of the values . Hence , the tree would now look like :



where **i** would close to n/2 . (because probabilistically balanced BST) .

Why "Implicit" Treap ?

- Since we are using the array index as the key of the BST this time, with each update (insertion / deletion) we will have to change O(n) values (the index of O(n) nodes would change upon an insertion/deletion of an element in array). This would be very slow.
- To avoid this, we will not explicitly store the index **i** (i.e. the "key" or Bk value) at each node in the implicit treap and calculate this value on the fly.
- Hence the name **Implicit Treap** because the key values are not stored explicitly and are **implicit**.
- The key value for any node **x** would be 1 + no of nodes in the BST that have key values less than x . (where node x means node representing A[x]).
- Note that nodes having key less than **x** would occur not only in the left subtree of x , but also in the left subtree of all the parents p of x such that x occurs in the right subtree of p.
- Hence the key for a node t = sz(t->l) + sz(p->l) for all parents of t such that t occurs in the right subtree of p.

Given the above modifications for an implicit treap, the new split and merge functions can be realized as follows :

```
typedef node* pnode;
int sz(pnode t){
    return t?t->size:0;
}
void upd_sz(pnode t){
    if(t)t->size=sz(t->l)+1+sz(t->r);
}
void split(pnode t,pnode &l,pnode &r,int pos,int add=0){
    if(!t)return void(l=r=NULL);
    int curr_pos = add + sz(t->l);
    if(curr_pos<=pos)//element at pos goes to "l"
        split(t->r,t->r,r,pos,curr_pos+1),l=t;
    else
        split(t->l,l,t->l,pos,add),r=t;
    upd_sz(t);
}
void merge(pnode &t,pnode l,pnode r){ //l->leftarray,r->rightarray,t->resul
    if(!l || !r) t = l?l:r;
    else if(l->prior>r->prior)merge(l->r,l->r,r),t=l;
    else    merge(r->l,l,r->l),t=r;
    upd_sz(t);
}
```

In the above code,

- **split :** Splits the array A[1..n] into two parts : L[1..pos] , R[pos+1..n] , about "pos". Note that A[pos] comes in the left part.
- **merge :** merges L[1..n1] , R[1..n2] to form A[1..n1,n1+1,…n2] . Note that the condition for merge in treap (i.e. greatest element in l <= smallest element in r) is satisified here since the keys are array indices and we are assuming that mergin L and R would result in A such that first n1 elements of A come from L and next n2 from R.

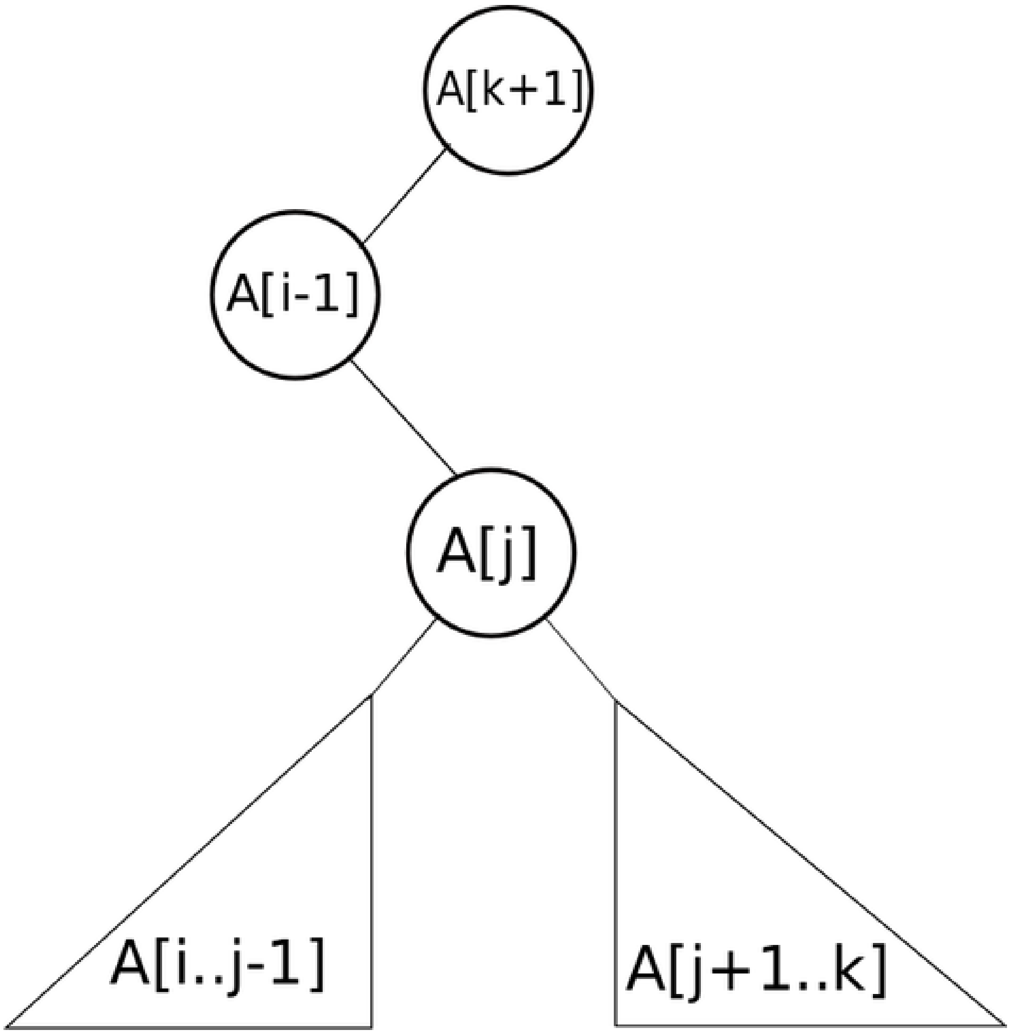
Additional Operations

Other operations supported by implicit treap can be realized using the basic split and merge operations (with a little modification as we shall see) .

1. **insert(x,i) : insert x at position "i"**
   To insert an element at position "i", we split the treap about pos = i-1 such that we get two treaps , L[1..i-1] and R[i .. n]. Now we merge L and x , and then merge the resulting treap with R.
2. **delete(i) : delete A[i] from the array**
   To erase an element at position "i", we split the treap about pos = i -1 such that we get two treaps L[1…i-1] and R[i…n]. We again split R about pos = i such we get L' [i] and R'[i+1…n]. We now merge L[1..i-1] and R'[i+1..n].

Operations of Interval Tree

Like segment tree, here also each node of treap represents a range. eg :



In the picture above, the node with key = j , represents the range [i … k]. Hence, just like a segment tree, we can compute the value of any objective function **f** for a node from the answers of its children. Hence to support such operations , we need to do the following modifications :

- First , each node would now contain an extra field "f" denoting the value of the objective function for the range represented by that node.
- We create a function operation(t) which calculates the value of "f" from the values of the left and right childrent of t . (very similar to function upd_sz(t) above)
- We insert calls to this function operation(t) , at the end of all functions which modify the tree (eg split and merge) . (Very similar to the calls to upd_sz(t)).
- To answer a query of [x..y] , we can split the treap into L[1..x-1] , M[x..y] , R[y+1..n] using two calls to split(). The root node of treap M,would now contain the answer for the range [x…y]. We again merge these treaps to get back the original treap using two calls to merge(). Since each call to split and merge is O(logN) , overall complexity remains O(logN).
- To do a point update at pos x , we can split the treap into L[1..x-1] , M[x] , R[x+1 .. n] using two calls to split(). Then modify the values in M and again merge them to get back the original treap using two calls to merge(). Complexity remains O(logN) .

- For a lazy range update at [x..y] , we need to maintain an additional field "lazy" at each node.
- We create a function push(t) which propagates the lazy update from a node to its children.
- We insert calls to this function push(t) , at the beginning of all the functions which modify the tree (eg split and merge) such that before the position of a node is changed, the lazy is propagated to its children. Hence, the data is not lost with any change in structure of the tree.
- To handle the range update query [x..y], we again the split the treap into L[1..x-1], M[x..y],R[y+1..n] using two calls to split(). Then , we update the lazy value of M and merge them to get back the original treap using 2 calls to merge. Since push is inserted in the beginning of merge, it ensures proper propagation of the lazy.

For a better understanding, look at the following complete c++ implementation of treap as interval tree :

```cpp
typedef struct node{
    int prior,size,val,sum,lazy;
    //value in array,info of segtree,lazy update
    struct node *l,*r;
}node;typedef node* pnode;
int sz(pnode t){
    return t?t->size:0;
}
void upd_sz(pnode t){
    if(t)t->size=sz(t->l)+1+sz(t->r);
}
void lazy(pnode t){
    if(!t || !t->lazy)return;
    t->val+=t->lazy;//operation of lazy
    t->sum+=t->lazy*sz(t);
    if(t->l)t->l->lazy+=t->lazy;//propagate lazy
    if(t->r)t->r->lazy+=t->lazy;
    t->lazy=0;
}
void reset(pnode t){
    if(t)t->sum = t->val;//lazy already propagated
}
void combine(pnode& t,pnode l,pnode r){//combine segtree ranges
    if(!l || !r)return void(t = l?l:r);
    t->sum = l->sum + r->sum;
}
void operation(pnode t){//operation of segtree
    if(!t)return;
    reset(t);//node represents single element of array
    lazy(t->l);lazy(t->r);//imp:propagate lazy before combining l,r
    combine(t,t->l,t);combine(t,t,t->r);
}
void split(pnode t,pnode &l,pnode &r,int pos,int add=0){
    if(!t)return void(l=r=NULL);
    lazy(t);int curr_pos = add + sz(t->l);
    if(curr_pos<=pos)//element at pos goes to "l"
        split(t->r,t->r,r,pos,curr_pos+1),l=t;
    else    split(t->l,l,t->l,pos,add),r=t;
    upd_sz(t);operation(t);
}
void merge(pnode &t,pnode l,pnode r){//result/left/right array
    lazy(l);lazy(r);
    if(!l || !r) t = l?l:r;
    else if(l->prior>r->prior)merge(l->r,l->r,r),t=l;
    else    merge(r->l,l,r->l),t=r;
    upd_sz(t);operation(t);
}
pnode init(int val){
    pnode ret = (pnode)malloc(sizeof(node));
    ret->prior=rand();ret->size=1;
    ret->val=val;ret->sum=val;ret->lazy=0;
    return ret;
}
int range_query(pnode t,int l,int r){//[l,r]
    pnode L,mid,R;
    split(t,L,mid,l-1);split(mid,t,R,r-l);//note: r-l!!
    int ans = t->sum;
    merge(mid,L,t);merge(t,mid,R);
    return ans;
}
void range_update(pnode t,int l,int r,int val){//[l,r]
    pnode L,mid,R;
    split(t,L,mid,l-1);split(mid,t,R,r-l);//note: r-l!!
    t->lazy+=val; //lazy_update
    merge(mid,L,t);merge(t,mid,R);
}
```

Problems for Practice (will be updated as and when found)

- Card Shuffle (http://www.codechef.com/problems/CARDSHUF/)
- SPOJ.com – Problem GSS6 (http://www.spoj.com/problems/GSS6/)
- SPOJ.com – Problem GSS3 (http://www.spoj.com/problems/GSS3/)
- http://www.spoj.com/problems/HOR… (http://www.spoj.com/problems/HORRIBLE/)
- http://www.spoj.com/problems/QMAX3VN/en/ (http://www.spoj.com/problems/QMAX3VN/en/)
- 3580 — SuperMemo (http://poj.org/problem?id=3580)
- https://www.hackerearth.com/july… (https://www.hackerearth.com/july-clash-15/algorithm/upgrade/) [A really interesting problem]

Conclusion :

It is really difficult to explain the working of implicit treap in words , so i would recommend you to go through the code again and again unless you are sure why things are working (especially lazy propagation) .

Note : Although implicit treaps give us great power in terms of the operations they support and it seems that we do not need segment tree/fenwick tree anymore (apart from the code length of course) , but I would strongly recommend you to use treap only when required and prefer segment tree/fenwick tree otherwise because although all the operations of treap are O(logN), it has a large constant factor hidden, unlike segment tree/fenwick tree. eg (see Submission #10520784 – Codeforces (http://codeforces.com/contest/380/submission/10520784) and Submission #10520750 – Codeforces (http://codeforces.com/contest/380/submission/10520750)).

Hope the article helped you understand implicit treaps.
Comments and feedbacks are most welcome.
Happy Coding 🙂

References

- Декартово дерево (treap, дерамида) (http://e-maxx.ru/algo/treap)

Note : Link to original post (https://threads-iiith.quora.com/Treaps-One-Tree-t

3 thoughts on "Treaps : One Tree to Rule 'em all ..!! Part-2"

Anand says:
April 8, 2016 at 5:22 am
What is amp in the treap?

Reply

tanujkhattar says:
April 8, 2016 at 7:28 am
It was a wordpress formatting error. Sorry. The post has been updated.

Reply

Nalin Bhardwaj says:
April 29, 2016 at 10:40 am
Hi
I wrote an implicit treap with a slightly different logic and much different implementation. My implementation seems to work much faster than yours, by basically getting rid of the constant factor by erasing/using much lesser memory. For the problem [380C](http://codeforces.com/contest/380/problem/C) for example, my code runs within the TL of 1 second. I tested the amount of time and memory your code takes on the problem, by changing the TL of the problem to 10 seconds, and in comparison your code takes nearly thrice as much time and memory. See screenshot ![comparison](http://i.imgur.com/qcSwnU3.png) the second submission is same as the code which TLEs in your note. You can read my code [here](http://codeforces.com/contest/380/submission/17564579).

Reply