**[Algorithm Gym :: Data structures](http://codeforces.com/blog/entry/15729)**

**Trees**

Trees are one of the most useful data structures.A tree is a connected-acyclic graph.There are too many types of trees, like : rooted trees, weighted trees, directed trees, tries, etc.

**Partial sum**

There are two types of problems solvable by partial sum.

**1.**Problems which you are asked to answer some queries about the sum of a part of elements (without modify queries). Solution of all of this problems are the same. You just need to know how to solve one of them.

Example : You are asked some queries on an array *a*1, *a*2, ...*a*, *n*. Each query give you numbers *l* and *r* and you should print*al* + *al*+ 1 + ... + *ar* .

Solution : You need to build another array *s*1, *s*2, ..., *sn* which *si* = *a*1 + *a*2 + ... + *ai* and answer is *sr* - *sl*- 1 .

**2.**Problems which you are asked to perform some queries asking you to modify a part of elements (without printing queries.) Solution of all of this problems are the same. You just need to know how to solve one of them.

Example : You need to perform some queries on an array *a*1, *a*2, ...*a*, *n*. Each query give you numbers *l*, *r* and *v* and for each *i* such that*l* ≤ *i* ≤ *r* you should increase *ai* by *v*, and then after performing all queries, you should print the whole array.

Solution : You should have another array *p*1, *p*2, ..., *pn* which, all of its members are initially 0, for each query, you should increase *pl* by *v*and decrease *pr*+ 1 by *v* .

An then, for each *i*, starting from 1 you should increase *pi* by *pi*- 1. So, final array would be *a*1 + *p*1, *a*2 + *p*2, ..., *an* + *pn* .

Hard problem of partial sum : [Troynacci Query](http://codeforces.com/gym/100571/problem/B)

**Disjoint sets**

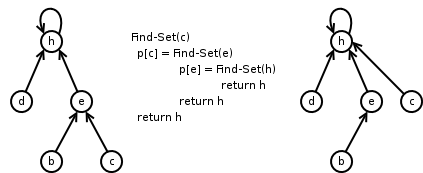
Disjoint sets are also useful data structures. Using them is fast and easy. We use theme in many algorithms, like Kruskal's and Prim's. Disjoint sets, or DSU (Disjoint Sets Union) as their name, are sum sets. Imagine we have some boxes and some tools and initially each tool is in one box. Mostly, we are given some queries and ask to merge two boxes or print the members of a box or find which box is some tool in.

For rest of these, let's consider that initially there is exactly one tool in a box.That is, we have *n* tools and *n* boxes and initially, tool number*i* is in box number *i*.

For this propose, we can use so many containers.Like :

**Trees**

Trees are the most useful containers for DSU. For each vertex, we keep it's parent (and parrent of the root is -1). So, initially are parents are set to -1, and we have queries to find the root of each box(having the root, we can easily find the box's index) and queries for merging two trees. For better time complexity, every time we want to find the root of each vertex, we set it's parent to the root for the next queries.And while merging, we always want to minimize the height of the tree, so when we want to merge the boxes, it's like we put all the tools of the box with fewer tools in the other box.



The best way I've seen to code this kind of DSU, is style of [**bmerry**](http://codeforces.com/profile/bmerry) : (C++)

*int root(int v){ return par[v] < 0 ? v : (par[v] = root(par[v]) ); }*

*void merge(int x,int y){* // x and y are some tools (vertices)

*if((x = root(x)) == (y = root(y))*

*return ;*

*if(par[y] < par[x])* // balancing the height of the tree

*swap(x, y);*

*par[x] += par[y];*

*par[y] = x;*

*}*

In the code above, for each root *v*, *par*[*v*] equals the negative of number of tools in that box.

**Arrays, vectors**

We keep tools in a vector (or an array) and when we have a query to merge two boxes, we put all the tools of the box with fewer tools in the other box. The time complexity is good because for each tool, we take and put it in an other box at most *log*(*n*) times (each time the size of the vector will be at least doubled). So time complexity would be *O*(*n*.*log*(*n*)) .

**Sets (red-black trees)**

Other way is to keep them in a red-black tree (in C++ it's set). We do exactly like vectors, so time complexity would be *O*(*n*.*log*2(*n*)) . (One *log* is for inserting).

**Problems** : [Hamro and tools](http://codeforces.com/group/L1Sf9F4uBt/contest/200499/problem/E), [TROY Query](http://codeforces.com/gym/100571/problem/F) (Join the group [ACM-OI](http://codeforces.com/group/L1Sf9F4uBt) first)

**Tries**

Tries are some kind of rooted trees in which each edge has a character on it. Actually, trie is some kind of DFA (Determining Finite Automata). For a bunch of strings, their trie is the smallest rooted tree with a character on each edge and each of these strings can be build by writing down the characters in the path from the root to some node. It's advantage is, LCP (Longest Common Prefix) of two of these strings is the *LCA* (Lowest Common Ancestor) of their nodes in the trie(a node that we can build the string by writing down the characters in the path from the root to that node). Generating the trie :

Root is vertex number 0 (C++)

*int x[MAX\_NUMBER\_OF\_NODES][MAX\_ASCII\_CODE], next = 1; //initially all numbers in x are -1*

*void build(string s){*

*int i = 0, v = 0;*

*while(i < s.size()){*

*if(x[v][s[i]] == -1)*

*v = x[v][s[i++]] = next ++;*

*else*

*v = x[v][s[i++]];*

*}*

*}*

**Problem** : [A lot of games](http://codeforces.com/contest/456/problem/D)

**Suffix array**

Suffix array is a data structure that helps you sort all the suffixes in lexicography order. This array consists of integers, the beginning of suffixes.

There are two ways to achieve this goal :

**One)** Non-deterministic algorithm : Use Robin-Carp and for check if a suffix is lexicographically less than another one, find their *LCP* using binary search + hash and then check the next character after their *LCP*. Code :

*namespace HashSuffixArray {*

*const int MAXN = 1 << 21;*

*typedef unsigned long long hash;*

*const hash BASE = 137;*

*int N;*

*char \* S;*

*int sa[MAXN];*

*hash h[MAXN], hPow[MAXN];*

*#define getHash(lo, size) (h[lo] - h[(lo) + (size)] \* hPow[size])*

*inline bool sufCmp(int i, int j)*

*{*

*int lo = 1, hi = min(N - i, N - j);*

*while (lo <= hi)*

*{*

*int mid = (lo + hi) >> 1;*

*if (getHash(i, mid) == getHash(j, mid))*

*lo = mid + 1;*

*else*

*hi = mid - 1;*

*}*

*return S[i + hi] < S[j + hi];*

*}*

*void buildSA()*

*{*

*N = strlen(S);*

*hPow[0] = 1;*

*for (int i = 1; i <= N; ++i)*

*hPow[i] = hPow[i - 1] \* BASE;*

*h[N] = 0;*

*for (int i = N - 1; i >= 0; --i)*

*h[i] = h[i + 1] \* BASE + S[i], sa[i] = i;*

*stable\_sort(sa, sa + N, sufCmp);*

*}*

*}* // end namespace HashSuffixArray

**Two)** Deterministic algorithm : We sort them *log*(*MaxLength*) steps, in the *i* - *th* step (counting from 0), we sort them according to their first 2*i* characters and put the suffixes whit the same prefix with 2*i* characters in the same buckets.

Code :

/\*

Suffix array O(n lg^2 n)

LCP table O(n)

\*/

*#include <cstdio>*

*#include <algorithm>*

*#include <cstring>*

*using namespace std;*

*#define REP(i, n) for (int i = 0; i < (int)(n); ++i)*

*namespace SuffixArray*

*{*

*const int MAXN = 1 << 21;*

*char \* S;*

*int N, gap;*

*int sa[MAXN], pos[MAXN], tmp[MAXN], lcp[MAXN];*

*bool sufCmp(int i, int j)*

*{*

*if (pos[i] != pos[j])*

*return pos[i] < pos[j];*

*i += gap;*

*j += gap;*

*return (i < N && j < N) ? pos[i] < pos[j] : i > j;*

*}*

*void buildSA()*

*{*

*N = strlen(S);*

*REP(i, N) sa[i] = i, pos[i] = S[i];*

*for (gap = 1;; gap \*= 2)*

*{*

*sort(sa, sa + N, sufCmp);*

*REP(i, N - 1) tmp[i + 1] = tmp[i] + sufCmp(sa[i], sa[i + 1]);*

*REP(i, N) pos[sa[i]] = tmp[i];*

*if (tmp[N - 1] == N - 1) break;*

*}*

*}*

*void buildLCP()*

*{*

*for (int i = 0, k = 0; i < N; ++i) if (pos[i] != N - 1)*

*{*

*for (int j = sa[pos[i] + 1]; S[i + k] == S[j + k];)*

*++k;*

*lcp[pos[i]] = k;*

*if (k)--k;*

*}*

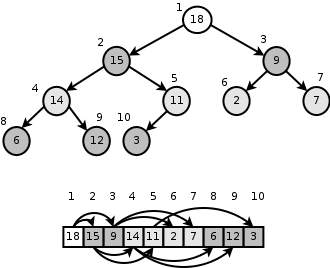
*}*

*}* // end namespace SuffixArray

**Heaps**

A heap is a binary rooted tree (a rooted tree that each node has at most 2 children) and each vertex has a value.

**Heap property** : Heap usually has a property, like the value of each vertex is equal to or greater than the value of its child(ren) (we call this a max heap). We can use heaps in heap sort.

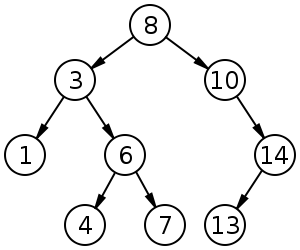


**Fibonacci heaps**

A fibonacci heap is a kind of heap with better complexities. We don't need to know what a fibonacci heap is.C++ already has one,priority\_queue.

**Binary Search Tree (BST)**

A binary search tree (BST) is a binary rooted tree that every node has a value, and for each node, the value of every node in its left child's subtree is less than its value and the value of every node in its right child's subtree is greater than that. Usually we perform some queries on BSTs, like inserting, deleting, asking and ... .

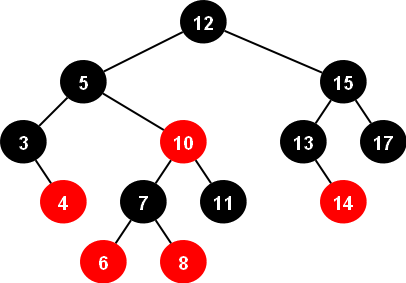


Binary search trees are too useful.

**Red-black trees**

A red-black tree is a kind of BST that after each query, BST will be balanced in such a way that it's height remains *O*(*log*(*n*)).

C++ already has a red-black tree inside, set . You can read about them in C++ references.



Unfortunately, set has not any function to find the *k* - *th* smallest minimum or find the index of an element, bust there is a data structure in C++ with does it in *O*(*log*(*n*))(also contains all set functions), tree :

*#include<bits/stdc++.h>*

*#include<ext/pb\_ds/assoc\_container.hpp>*

*#include<ext/pb\_ds/tree\_policy.hpp>*

*using namespace \_\_gnu\_pbds;*

*using namespace std;*

*template <typename T>*

*using ordered\_set = tree<T, null\_type, less<T>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;*

*int main(){*

*ordered\_set<int> s;*

*s.insert(1);*

*s.insert(3);*

*cout << s.order\_of\_key(2) << endl;* // the number of elements in the s less than 2

*cout << \*s.find\_by\_order(0) << endl;* // print the 0-th smallest number in s(0-based)

*}*

**SQRT Decomposition**

Suppose we have an array *a*1, *a*2, ..., *an* and . We partition this array into *k* pieces each containing *k* elements of *a*. Doing this, we can do a lot of things in . Usually we use them in the problems with modify and ask queries.

**Problems** : [Holes](http://codeforces.com/problemset/problem/13/E), [DZY Loves Colors](http://codeforces.com/contest/444/problem/C), RMQ (range minimum query) problem

**Sparse Table**

The main problem that we can solve is RMQ problem, we have an array *a*1, *a*2, ..., *an* and some queries. Each query gives you numbers *l*and *r* (*l* ≤ *r*) and you should print the value of *min*(*al*, *al*+ 1, ..., *ar*) .

Solving using Sparse Table : For each *i* that 1 ≤ *i* ≤ *n* and for each *j* that 0 ≤ *j* and *i* + 2*j* - 1 ≤ *n*, we keep the value of *min*(*ai*, *ai*+ 1, ..., *ai*+ 2*j* -1) in *st*[*i*][*j*] (preprocess) :

(code is 0-based)

*for(int j = 0;j < MAX\_LOG;j++)*

*for(int i = 0; i < n;i ++)if(i + (1 << j) - 1 < n)*

*st[i][j] = (j ? min(st[i][j-1], st[i + (1 << (j-1)) - 1][j-1]): a[i]);*

And then for each query, first of all, find the maximum *x* such that 2*x* ≤ *r* - *l* + 1 and answer is *min*(*st*[*l*][*x*], *st*[*r* - 2*x* + 1][*x*]) .

So, the main idea of Sparse Table, is to keep the value for each interval of length 2*k* (for each *k*). You can use the same idea for *LCA* problem and so many other problems.

So preprocess will be in *O*(*n*.*log*(*n*)) and query will be in *O*(1)

**Problems** : [Strip](http://codeforces.com/contest/487/problem/B), [GCDSSQ](http://codeforces.com/contest/475/problem/D), [LCM Query](http://codeforces.com/gym/100570/problem/A) .

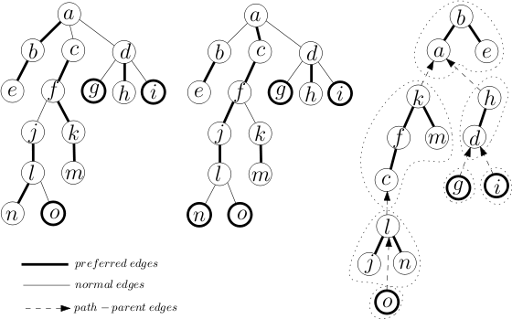
**Heavy light decomposition**

Heavy light decomposition is a way to partition a tree's vertices (or edges) in a good way. In this kind of decomposition, we have some chains, and each vertex belongs to only one chain.

If vertex *v* is the parent of *u* size\_of\_subtree\_of(*v*)/2 < size\_of\_subtree\_of(*u*), *u* and *v* are in a chain and we call the edge *uv*, heavy, otherwise light.

There is at most one such child for each vertex *v*. If we consider the path from any vertex *v* to the root, there will be at most *log*(*n*) light edges there (go from *v* to the root, every time we see a light edge, size of subtree will be at least doubled). So, the number of chains on the way = *O*(*log*(*n*)) .

In each of these chains, we can contain a container or another data structure like segment tree or etc.



**Problem** : [GRASS PLANTING](http://usaco.org/index.php?page=viewproblem2&cpid=102)

**Fenwick**

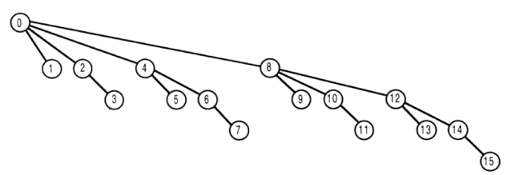
Suppose that we have *n* elements numbered from 1 to *n*.

Fenwick or BIT(Binary Indexed Tree) is a data structure with *n* nodes that node number *i* has some information about elements in the interval (*i* -  *i*& - *i*, *i*] .

Actually, you don't need to know what each node contains. The only thing you should know, it this (then you can change and convert it) :

We have an array *a*1, *a*2, ..., *an* and all of them are initially 0. We are gives some queries, 1.increase *ap* by *val* and print *a*1 + *a*2 + ... + *ap*.

Only thing you should know is that how to solve this problem using Fenwick (and then you can change it and solve so many problems).



We perform each query in *O*(*log*(*n*)). Code : (1-based)

*int fen[MAX\_N];*

*void update(int p,int val){*

*for(int i = p;i <= n;i += i & -i)*

*fen[i] += val;*

*}*

*int sum(int p){*

*int ans = 0;*

*for(int i = p;i;i -= i & -i)*

*ans += fen[i];*

*return ans;*

*}*

Please note that it should be **1-based**. It can't be done 0-based.

**Problems** : [Inversions](http://acm.sgu.ru/problem.php?contest=0&problem=180), [Pashmak and Parmida's problem](http://codeforces.com/contest/459/problem/D), [BST](http://hsin.hr/coci/archive/2008_2009/contest3_tasks.pdf) .

**Segment tree**

We have an array of elements and some queries on intervals. So, we will be glad if we can split this interval to *O*(*log*(*n*)) intervals that we have actually some information about them. Segment tree does that for you. Segment tree is a tree that each of it's nodes belongs to an interval.

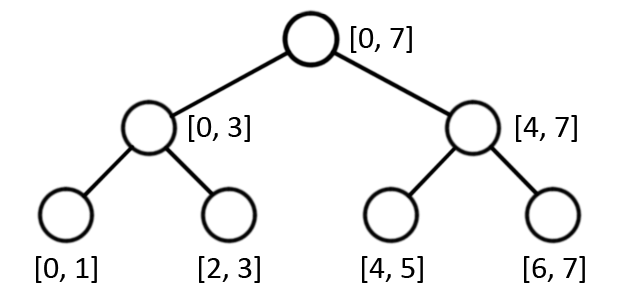
Root of the tree belongs to the interval [0, *n*) (0-based).

Each node, has 0 or two children. Left and right. If a node's interval is [*l*, *r*) and *l* + 1 ≠ *r*, the interval of its children will be [*l*, *mid*) and[*mid*, *r*) in order where , so the height of this tree will be *O*(*log*(*n*)) .

Each node has an index, we consider that root has index 1 and the children of a vertex *x* will have indices 2*x* and 2*x* + 1 in order.

Segment tree is the most useful data structure and every problem solvable by Fenwick is also solvable by Segment tree. If the size of the root's interval is *n*, segment tree could have up to 4*n* nodes. To split an interval to some nodes of this tree, we will act like this :

Suppose that *S* is the set of nodes which their union is [*x*, *y*) and no two different nodes in *S* have nonempty intersection. A node *i* with interval [*l*, *r*) is in *S* if and only if *x* ≤ *l* ≤ *r* ≤ *y* and if it has a parent with interval [*b*, *e*), *x* > *l* or *r* > *y* .



C++ code :

*vector<int> s;*

*void split(int x,int y, int id = 1,int l = 0, int r = n)*

*{* // id is the index of the node

*if(x >= r or l >= y)*

*return ;* // in this case, intersect of [l,r) and [x,y) is empty

*if(x <= l && r <= y){*

*s.push\_back(id);*

*return ;*

*}*

*int mid = (l+r)/2;*

*split(x,y,id \* 2,l,mid);*

*split(x,y,id \* 2 + 1,mid,r);*

*}*

Example :

We have an array *a*1, *a*2, ..., *an* and *q* queries. There are 2 types of queries.

1. *S* *l* *r*, Print *al*, *al*+ 1, ..., *ar*
2. *M* *p* *x*, Modify *ap* to *x*, it means *ap* = *x* .

First of all we need to build the segment tree, for each node we keep the sum of its interval, for node *i* we call it *s*[*i*], so we should build the initial segment tree.

*void build(int id = 1,int l = 0,int r = n){*

*if(r - l < 2){* // l + 1 == r

*s[id] = a[l];*

*return ;*

*}*

*int mid = (l+r)/2;*

*build(id \* 2, l, mid);*

*build(id \* 2 + 1, mid, r);*

*s[id] = s[id \* 2] + s[id \* 2 + 1];*

*}*

So, before reading the queries, we should call *build*() .

Modify function :

*void modify(int p,int x,int id = 1,int l = 0,int r = n){*

*s[id] += x - a[p];*

*if(r - l < 2){* //l = r - 1 = p

*a[p] = x;*

*return ;*

*}*

*int mid = (l + r)/2;*

*if(p < mid)*

*modify(p, x, id \* 2, l, mid);*

*else*

*modify(p, x, id \* 2 + 1, mid, r);*

*}*

(We should call *modify*(*p*,  *x*)) Ask for sum function :

*int sum(int x,int y,int id = 1,int l = 0,int r = n){*

*if(x >= r or l >= y)*

*return 0;*

*if(x <= l && r <= y)*

*return s[id];*

*int mid = (l+r)/2;*

*return sum(x, y, id \* 2, l, mid) +*

*sum(x, y, id \* 2 + 1, mid, r);*

*}*

(We should call *sum*(*l*,  *r*))

**Lazy propagation**

Imagine we have updates on intervals, what should we do ?

Example :

We have an array *a*1, *a*2, ..., *an* and *q* queries. There are 2 types of queries.

1. *S* *l* *r*, Print *al*, *al*+ 1, ..., *ar*
2. *I* *l* *r* *x*, for each *i* such that *l* ≤ *i* < *r*, increase *ai* by *x*.

We shouldn't update all the nodes in this interval, just the maximal ones, then pass it to children when we need. This trick is called **Lazy Propagation**, so we should have another array *lazy* (for nodes) which are initially 0 and every time we want to perform increase query, increase *lazy*[*id*] with *x*.

As above, we also should have an array *s* for nodes.

So, *build* function will be same as above. But we need some more functions :

A function to update a node :

*void upd(int id,int l,int r,int x){* //increase all members in this interval by x

*lazy[id] += x;*

*s[id] += (r - l) \* x;*

*}*

A function to pass the update information to its children :

*void shift(int id,int l,int r){* //pass update information to the children

*int mid = (l+r)/2;*

*upd(id \* 2, l, mid, lazy[id]);*

*upd(id \* 2 + 1, mid, r, lazy[id]);*

*lazy[id] = 0;* // passing is done

*}*

A function to perform increase queries :

*void increase(int x,int y,int v,int id = 1,int l = 0,int r = n){*

*if(x >= r or l >= y)*

*return ;*

*if(x <= l && r <= y){*

*upd(id, l, r, v);*

*return ;*

*}*

*shift(id, l, r);*

*int mid = (l+r)/2;*

*increase(x, y, v, id \* 2, l, mid);*

*increase(x, y, v, id\*2+1, mid, r);*

*s[id] = s[id \* 2] + s[id \* 2 + 1];*

*}*

(We should call *increase*(*l* *r* *x*))

A function to answer to queries asking about the sum :

*int sum(int x,int y,int id = 1,int l = 0,int r = n){*

*if(x >= r or l >= y)*

*return 0;*

*if(x <= l && r <= y)*

*return s[id];*

*shift(id, l, r);*

*int mid = (l+r)/2;*

*return sum(x, y, id \* 2, l, mid) +*

*sum(x, y, id \* 2 + 1, mid, r);*

*}*

(We should call *sum*(*l*,  *r*))

**Problems** : [GSS1](http://www.spoj.com/problems/GSS1/), [GSS3](http://www.spoj.com/problems/GSS3/), [MULTQ3](http://www.spoj.com/problems/MULTQ3), [DQUERY](http://www.spoj.com/problems/DQUERY), [KQUERY](http://www.spoj.com/problems/KQUERY), [POSTERS](http://www.spoj.com/problems/POSTERS), [PATULJCI](http://www.spoj.com/problems/PATULJCI), [New Year Domino](http://codeforces.com/problemset/problem/500/E), [Copying Data](http://codeforces.com/problemset/problem/292/E), [DZY Loves Fibonacci Numbers](http://codeforces.com/problemset/problem/446/C), [FRBSUM](http://www.codechef.com/JAN14/problems/FRBSUM)

**Persistent data structures**

Consider we have some elements, you perform some updates on it and then,and after performing all of them, you want to have the information about the elements, after each update.

For this propose, you got a data structure and somehow, you save the version of that data structure.

The most useful data structure for this propose is segment tree, I will explain persistent segment tree and all other data structures (like Fenwick) are like that.

**Persistent segment tree**

Example problem :

We have an array *a*1, *a*2, ..., *an* and at first *q* update queries and then *u* ask queries which you have to answer online.

Each update query gives you numbers *p* and *v* and asks you to increase *ap* by *v* .

Each ask query, gives you three numbers *i* and *x* and *y* and asks you to print the value of *ax* + *ax*+ 1 + ... + *ay* after performing *i* - *th*query.

Each update query, changes the value of *O*(*log*(*n*)) nodes in the segment tree, so you should keep rest of nodes (not containing *p*) and create *log*(*n*) new nodes. Totally, you need to have *q*.*log*(*n*) nodes. So, you can not use normal segment's indexing, you should keep the index of children in the arrays *L* and *R*.

If you update a node, you should assign a new index to its interval (for *i* - *th* query).

You should keep an array *root*[*q*] which gives you the index of the interval of the root ( [0, *n*) ) after performing each query and a number*ir* = 0 which is its index in the initial segment tree (ans of course, an array *s*[*MAXNODES*] which is the sum of elements in that node). Also you should have a *NEXT\_FREE\_INDEX = 1* which is always the next free index for a node.

First of all, you need to build the initial segment tree :

(In these codes, all arrays and queries are **0-based**)

*void build(int id = ir,int l = 0,int r = n){*

*if(r - l < 2){*

*s[id] = a[l];*

*return ;*

*}*

*int mid = (l+r)/2;*

*L[id] = NEXT\_FREE\_INDEX ++;*

*R[id] = NEXT\_FREE\_INDEX ++;*

*build(L[id], l, mid);*

*build(R[id], mid, r);*

*s[id] = s[L[id]] + s[R[id]];*

*}*

(So, we should call *build*() )

Update function : (its return value, is the index of the interval in the new version of segment tree and *id* is the index of old one)

*int upd(int p, int v,int id,int l = 0,int r = n){*

//index of the node in new version of segment tree

*int ID = NEXT\_FREE\_INDEX ++;*

*if(r - l < 2){*

*s[ID] = (a[p] += v);*

*return ID;*

*}*

*int mid = (l+r)/2;*

//in case of not updating the interval of left child or right child

*L[ID] = L[id], R[ID] = R[id];*

*if(p < mid)*

*L[ID] = upd(p, v, L[ID], l, mid);*

*else*

*R[ID] = upd(p, v, R[ID], mid, r);*

*return ID;*

*}*

(For the first query (with index 0) we should run *root*[0] = *upd*(*p*,  *v*,  *ir*) and for the rest of them, for *j* - *th* query se should run*root*[*j*] = *upd*(*p*,  *v*,  *root*[*j* - 1]) )

Function for ask queries :

*int sum(int x,int y,int id,int l = 0,int r = n){*

*if(x >= r or l >= y)*

*return 0;*

*if(x <= l && r <= y)*

*return s[id];*

*int mid = (l+r)/2;*

*return sum(x, y, L[id], l, mid) +*

*sum(x, y, R[id], mid, r);*

*}*

(So, we should print the value of *sum*(*x*, *y*, *root*[*i*]) )

**Problems** : [Sign on Fence](http://codeforces.com/problemset/problem/484/E), [MKTHNUM](http://www.spoj.com/problems/MKTHNUM/), [COT](http://www.spoj.com/problems/COT/), [The Classic Problem](http://codeforces.com/problemset/problem/464/E)