```
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector
#include <fstream>
#include <math.h>       /* floor */
#include <stdlib.h>
#include <cmath>        /* pow  */

using namespace std;

struct Node{
   int value;
   Node * left;
   Node * right;

   Node( int i ): value(i) , left(NULL) , right(NULL) {}
};

class BinTree{
   Node * root_;
public:
   BinTree() { root_ = NULL ; }

   Node * getRoot() { return root_; cout << "getRoot" <<
endl;}

   void insert( int i )    {
      Node * node = new Node(i);

      Node * pre = NULL;
      Node * post = root_;
      while( post != NULL){
         pre = post;
         if( i <= post->value ){
            post = post->left;
         }
         else {
            post = post->right;
         }
      }
      if( pre == NULL )  root_ = node;
      else if( i <= pre->value ){
         pre->left = node;
      }
      else{
         pre->right = node;
      }
      return;
   }
};

/*soluzione proposta dagli studenti*/
int funzione (Node* n, int altezza){
            if (!n) return 0;
            int altezza2 = altezza+1;
            int sx = funzione(n->left,  altezza2);
            int dx = funzione(n->right, altezza2);

            if ( (altezza%2 == 0 && (sx+dx)%2 == 0) ||
(altezza%2 != 0 && (sx+dx)%2 != 0) ){
                        cout<<n->value<<endl;
            }
            return sx+dx+n->value;
}

int main(){
   int N ;
   int x ;
   BinTree albero ;

   cin >> N ;

   // riempio l' albero
   for(int i=0 ; i<N ; ++i )   {
      cin >> x;
      albero.insert(x);
   }
   cout<<"==========="<<endl;
   funzione(albero.getRoot(), 1);
   cout<<endl;
}
```

17/02/2021

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca coerentemente con classi e strutture pubblicate sul TEAM della prova e con il main di seguito riportato.

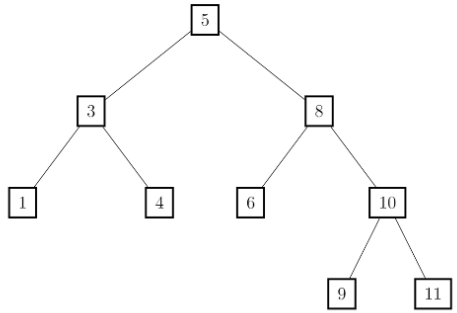
```
int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )    {
        cin >> x;
        albero.insert(x);
    }

    E' possibile dichiarare fino a massimo 3 variabili

    Chiamata di funzione

}
```

Per convenzione, si riferiscono come *concordi* i nodi la cui altezza e' pari (o dispari) come la sommatoria dei labels dei loro discendenti. Si definisca una funzione la cui chiamata sfrutti le *eventuali* variabili dichiarate nel main, e produca **la stampa dei labels dei nodi *concordi*. La radice, per convenzione, ha altezza 1.** La complessita' della soluzione deve essere la minima possibile.

Esempio

Input

```
9
5
3
8
1
4
6
10
9
11
```



Output

```
9
11
8
```

```
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector
#include <fstream>
#include <math.h>       /* floor */
#include <stdlib.h>
#include <cmath>        /* pow  */

using namespace std;

struct Node{
   int value;
   Node * left;
   Node * right;

   Node( int i ): value(i) , left(NULL) , right(NULL) {}
};

class BinTree{
   Node * root_;
public:
   BinTree() { root_ = NULL ; }
   Node * getRoot() { return root_; cout << "getRoot" <<
endl;}

void insert( int i ){
      Node * node = new Node(i);

      Node * pre = NULL;
      Node * post = root_;
      while( post != NULL)    {
         pre = post;
         if( i <= post->value )          {
            post = post->left;
         }
         else          {
            post = post->right;
         }
      }

      if( pre == NULL )
         root_ = node;
      else if( i <= pre->value )       {
         pre->left = node;
      }
      else       {
         pre->right = node;
      }
      return;
   }
};

// -------------------------METODO ESTERNO DA
AGGIUNGERE-------------------------
/*soluzione proposta da uno studente*/
int somm (Node* tree,int altezza) {
   if (!tree) return 0;
   if (altezza++%2!=0) {
      if (tree->left== nullptr || tree->right== nullptr)
         return tree->value + somm(tree->left,altezza) +
somm (tree->right,altezza);
   }
   return somm(tree->left,altezza) + somm (tree-
>right,altezza);
}

int main(){
   int N ;
   int x ;
   BinTree albero ;
   cin >> N ;
   // Inserimento elementi nell' albero
   for(int i=0 ; i<N ; ++i )   {
      cin >> x;
      albero.insert(x);
   }

   //CHIAMATA AL METODO---------
   int somma=somm(albero.getRoot(),1);
   cout <<"Somma label nodi incompleti: " <<somma
<<endl;
}
```

27/01/2021

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca coerentemente con classi e strutture pubblicate sul TEAM della prova e con il main di seguito riportato.

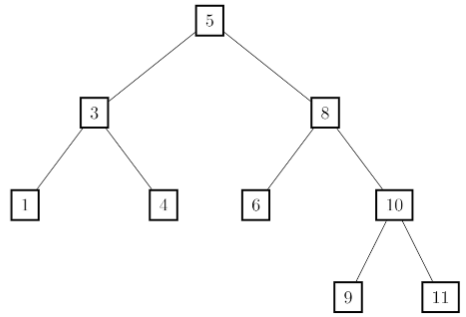
```
int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )    {
        cin >> x;
        albero.insert(x);
    }

    E' possibile dichiarare fino a massimo 3 variabili

    Chiamata di funzione
}
```

Per convenzione, si riferiscono come *incompleti* i nodi con meno di due figli. Si definisca una funzione la cui chiamata sfrutti le eventuali variabili dichiarate nel main, e produca la stampa della **sommatoria dei label dei nodi incompleti di altezza dispari. La radice, per convenzione, ha altezza 1.** La complessita' della soluzione deve essere la minima possibile.

Esempio

Input

```
9
5
3
8
1
4
6
10
9
11
```



Output

```
11
```

```
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector
#include <fstream>
#include <math.h>       /* floor */
#include <stdlib.h>
#include <cmath>        /* pow  */

using namespace std;

struct Node{
   int value;
   Node * left;
   Node * right;
   Node( int i ): value(i) , left(NULL) , right(NULL) {}
};

class BinTree{
   Node * root_;
public:
   BinTree() { root_ = NULL ; }
   Node * getRoot() { return root_; cout << "getRoot" <<
endl;
}

 void insert( int i )    {
       Node * node = new Node(i);
       Node * pre = NULL;
       Node * post = root_;
       while( post != NULL)       {
          pre = post;
          if( i <= post->value )          {
             post = post->left;
          }
          else          {
             post = post->right;
          }
       }

       if( pre == NULL )
          root_ = node;
       else if( i <= pre->value )       {
          pre->left = node;
       }
       else       {
          pre->right = node;
       }
       return;

    }
};

// ---METODO ESTERNO DA AGGIUNGERE-
/*soluzione proposta da uno studente*/
int concord (Node* tree,int padre) {
   if (!tree) return 0;
   if (padre==0 || (tree->value%2==0 && padre%2==0) ||
(tree->value%2!=0 && padre%2!=0))
       return tree->value + concord(tree->left,tree->value) +
concord(tree->right,tree->value);
   return concord(tree->left,tree->value) + concord(tree-
>right,tree->value);
}
//============MAIN========
int main(){
   int N ;
   int x ;
   BinTree albero ;
   cin >> N ;
   // Inserimento elementi nell' albero
   for(int i=0 ; i<N ; ++i )   {
      cin >> x;
      albero.insert(x);
   }
 // ---------CHIAMATA AL METODO-------
   cout <<"Somma dei label concordi: "
<<concord(albero.getRoot(),0) <<endl;

}
```

11/01/2021

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca coerentemente con classi e strutture pubblicate sul TEAM della prova e con il main di seguito riportato.

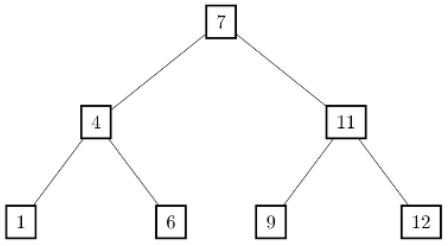
```
int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )    {
        cin >> x;
        albero.insert(x);
    }

    E' possibile dichiarare fino a massimo 3 variabili

    Chiamata di funzione

}
```

Si definisca una funzione la cui chiamata sfrutti le eventuali variabili dichiarate nel main, e produca la stampa della sommatoria dei label dei nodi *concordi*. Un nodo e' *concorde* **se ha label pari (o dispari) come anche il suo nodo padre**. La radice dell'albero per convenzione e' concorde. La complessita' della soluzione deve essere la minima possibile.

Esempio

Input

```
7
7
4
11
1
6
9
12
```



Output

```
33
```

```
#include <iostream>
using namespace std;

struct node {
   int label;
   node* left;
   node* right;

   node(int i): label(i), left(NULL), right(NULL){}
};

class binTree{
   node* root;

public:
   binTree(){root=NULL;}
   node* getRoot();
   void insert(int i);
};

node* binTree::getRoot() {
   return root;
}

void binTree::insert(int i) {
   node* nodo = new node(i);
   node* pre = NULL;
   node* post = root;

   while(post != NULL){
      pre = post;
      if(i <= post->label)
         post = post->left;
      else
         post = post->right;
   }

   if(pre == NULL)
      root = nodo;
   else if(i <= pre->label)
      pre->left = nodo;
   else
      pre->right = nodo;
}

/*soluzione fornita da uno studente: potrebbe non essere
la migliore, ma almeno funziona*/
void fun(node*tree,int& npari,int& ndisp){
   if(!tree){
      npari = ndisp=0;
      return;
   }

   int pari_left, disp_left, pari_right, disp_right;

   fun(tree->left,pari_left,disp_left);
   fun(tree->right,pari_right,disp_right);

   npari = pari_left + pari_right;
   ndisp = disp_left + disp_right;

   if(npari > ndisp)
      cout<<tree->label<<endl;
   if(tree->label%2 == 0)
      npari++;
   else
      ndisp++;
}

int main() {
   int N,x;
   cin>>N;

   binTree albero;
   for(int i = 0; i < N; i++){
      cin>>x;
      albero.insert(x);
   }
   int npari, ndisp;
   npari = ndisp = 0;
   fun(albero.getRoot(), npari, ndisp);
return 0;}
```

29/06/2020

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca coerentemente con classi e strutture pubblicate il 25/06/2020 e con il main di seguito riportato.

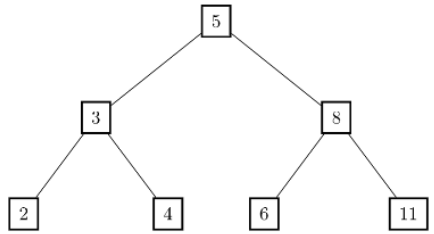
```
int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )    {
        cin >> x;
        albero.insert(x);
    }

    E' possibile dichiarare fino a massimo 3 variabili

    Chiamata di funzione

}
```

Si definisca una funzione che correttamente chiamata nel main, sfrutti le eventuali variabili dichiarate subito prima della chiamata di funzione al fine di **stampare i label dei nodi n tali che il sottoalbero radicato in n contiene piu' (strettamente maggiore) nodi con label pari che nodi con label dispari. La radice del sottoalbero e' esclusa dal conteggio.**
La complessita' della soluzione deve essere la minima possibile.

Esempio

Input

```
7
5
3
8
2
4
6
11
```



Output

```
3
5
```

```
#include <iostream>
using namespace std;

struct node{
   int label;
   node* left;
   node* right;

   node(int i): label(i), left(NULL), right(NULL){}
};

class binTree{
   node* root;

public:
   binTree(){root=NULL;}
   node* getRoot();
   void insert(int i);
};

node* binTree::getRoot() {
   return root;
}

void binTree::insert(int i) {
   node* nodo = new node(i);
   node* pre = NULL;
   node* post = root;

   while(post != NULL){
      pre = post;
      if(i <= post->label)
         post = post->left;
      else
         post = post->right;
   }

   if(pre == NULL)
      root = nodo;
   else if(i <= pre->label)
      pre->left = nodo;
   else
      pre->right = nodo;
}

/*soluzione fornita da uno studente: potrebbe non essere
la migliore, ma almeno funziona*/
int fun(node* tree, int& nfoglie){
   if(!tree) return 0;

   if(!tree->left && !tree->right){
      nfoglie = 1;
      return 1;
   }

   int fogliesx, fogliedx;
   fogliesx = fogliedx = 0;

   int h = max(fun(tree->left,fogliesx),fun(tree-
>right,fogliedx));

   nfoglie = fogliedx + fogliesx;

   if(h<nfoglie)
      cout<<tree->label<<endl;

   return h+1;
}

int main() {
   int N,x;
   cin>>N;

   binTree albero;
   for(int i = 0; i < N; i++){
      cin>>x;
      albero.insert(x);
   }

   int foglie = 0;
   fun(albero.getRoot(), foglie);
   return 0;}
```

29/06/2020

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca coerentemente con classi e strutture pubblicate il 25/06/2020 e con il main di seguito riportato.

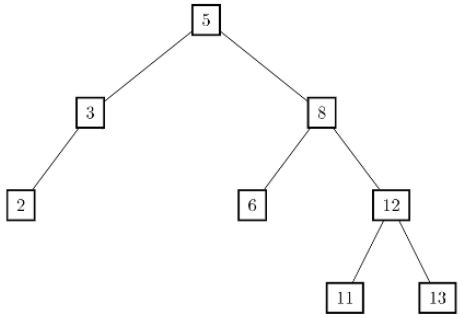
```
int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )    {
        cin >> x;
        albero.insert(x);
    }

    E' possibile dichiarare fino a massimo 3 variabili

    Chiamata di funzione

}
```

Si definisca una funzione che correttamente chiamata nel main, sfrutti le eventuali variabili dichiarate subito prima della chiamata di funzione al fine di **stampare i label dei nodi n tali che il sottoalbero radicato in n ha altezza strettamente minore del numero di foglie discendenti. Per altezza si intende la distanza massima tra radice del sottoalbero e una sua foglia.** La complessita' della soluzione deve essere la minima possibile.

Esempio

Input

```
8
5
3
8
2
6
12
11
13
```



Output

```
12
8
5
```

```
#include <iostream>     // std::cout
```

```
#include <algorithm>    // std::sort
#include <vector>       // std::vector
#include <fstream>
#include <math.h>       /* floor */
#include <stdlib.h>
#include <cmath>        /* pow  */

using namespace std;

struct Node{
   int value;
   Node * left;
   Node * right;
   Node( int i ): value(i) , left(NULL) , right(NULL) {}
};

class BinTree{
   Node * root_;
public:
   BinTree() { root_ = NULL ; }
   Node * getRoot() { return root_; cout << "getRoot" <<
endl;}

void insert( int i )    {
      Node * node = new Node(i);

      Node * pre = NULL;
      Node * post = root_;
      while( post != NULL)       {
         pre = post;
         if( i <= post->value )          {
            post = post->left;
         }
         else          {
            post = post->right;
         }
      }

      if( pre == NULL ) root_ = node;
      else if( i <= pre->value )       {
         pre->left = node;
      }
      else       {
         pre->right = node;
      }
      return;
   }
};

int plusParent( Node * tree, int altezza, int padre)
{
   // Nodo non trovato
   if( tree == NULL)  {
      return 0;
   }
   if (altezza++%2)  {
         return plusParent( tree->left , altezza, 0) +
plusParent( tree->right , altezza, 0) + tree->value - padre;
   }
   else   {
         return plusParent( tree->left , altezza, tree->value) +
plusParent( tree->right , altezza, tree->value) + 0;
   }
}

int main()
{
   int N ;
   int x ;
   BinTree albero ;

   cin >> N ;

   // riempio l' albero
   for(int i=0 ; i<N ; ++i )   {
      cin >> x;
      albero.insert(x);
   }

   cout<<plusParent(albero.getRoot(), 0, 0)<<endl;
}
```

08/06/2020

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca coerentemente con classi e strutture ricevute via email il 04/06/2020 e con il main di seguito riportato.

```
int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )   {
        cin >> x;
        albero.insert(x);
    }

    cout<<[redacted]<<endl;
}
```

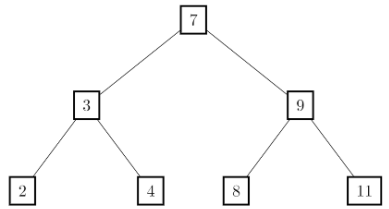
Si definisca una funzione che correttamente chiamata nel main in corrispondenza della banda nera, risulti nella stampa di un intero K. K e' ottenuto come:

- **Si dica $D(n)$, la differenza tra il valore di un nodo n e il valore del suo nodo padre; non esistendo padre per la radice, il valore da passare in tal caso e' 0;**
- **Si indichino con p tutti i nodi di altezza pari, considerando l'altezza della radice dell'albero pari a 1;**
- **K risulta come la sommatoria delle operazioni D sui nodi di altezza pari $K = \sum D(p)$;**

Esempio

Input

```
7
7
3
9
2
4
8
11
```



Output

```
-2
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


struct Node{
	int id;
	Node* left;
	Node* right;
	char type;
	int g;
	Node(int val, char t):id(val), left(NULL),
right(NULL), type(t), g(0){}
};


class binTree{
	private:
		Node* root;
	public:
		binTree(){root = NULL;}
		Node* getRoot(){return root;}

		void insert(int val, char t){
			Node* n = new Node(val, t);
			Node* pre = NULL;
			Node* post = root;

			while(post != NULL){
				pre = post;
				if (val <= pre->id) post =
post->left;
				else post = post->right;
			}

			if (pre == NULL) root = n;
			else if (val <= pre->id) pre->left
= n;
			else pre->right = n;
		}
};
/*STATES:
		0 -> partito dal server
		1 -> arrivato al filtro
		2 -> client raggiunto
*/
int calculate(Node* n, int state){
	int add = 0;
	if (n->type == 'C'){
		if (state != 0)	{state = 2; add += 1;}
	}
	else if (n->type == 'S'){
		state = 0;
	}
	else if (n->type == 'F'){
		if (state == 0) {state = 1;}
	}
	int sx = 0;
	int dx = 0;
	if (n->left  != NULL) sx = calculate(n->left , state);
	if (n->right != NULL) dx = calculate(n->right,
state);
	if (state == 0 && n->type=='S') {n->g = sx+dx;
return 0;}
	return sx+ dx + add;
}

void print(Node* n){
	if (n->left != NULL) print(n->left);
    if(n->type == 'S') cout<< n->id <<" "<< n->g <<endl;
	if (n->right != NULL) print(n->right);
}
int main(){
	int n;
	cin>>n;

	binTree b;
	for (int i = 0; i < n; i++){
		int tmp;
		char t;
		cin>>tmp>>t;
		b.insert(tmp, t);
	}
	int state = 0;
	calculate(b.getRoot(), state);
	print(b.getRoot());
	return 0;}
```

27/06/2019

Si consideri un sistema per la gestione di una rete informatica composta da N nodi. Ciascun nodo è caratterizzato da un ID intero e positivo, e da un tipo tra *Server*, *Client*, *Filtro* e *Router*, rappresentati dai caratteri S, C, F e R rispettivamente. I nodi della rete sono memorizzati tramite un albero binario di ricerca (ABR) usando l'ID come etichetta.

Un cammino che raggiunge un *Client* si dice *completo* se ha origine da un nodo *Server*, attraversa **almeno** un nodo *Filtro* e **nessun altro** *Server*.

Per ciascun *Server*, si definisce il numero g di *Clienti* serviti come il numero di cammini completi che hanno origine dal server stesso.

Scrivere un programma che consideri i *Server* in ordine di ID non decrescente, e stampi per ciascuno di essi il suo ID e il numero di *Client* da lui serviti. (complessità al più $\mathcal{O}(n)$).

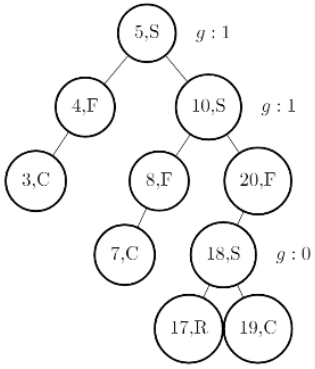
L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti una coppia $ID, tipo$ ciascuna, con i valori separati da uno spazio.

L'**output** contiene una coppia ID, g ciascuna, con i valori separati da uno spazio.

Esempio

Input

```
10
5 S
4 F
10 S
3 C
8 F
20 S
7 C
18 S
19 C
17 R
```



Output

```
5 1
10 1
18 0
```

```
#include <iostream>
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct node{
	int value;
	node* left;
	node* right;
	int h;

	node(int val): value(val), left(NULL),
right(NULL), h(0){}
};

class binTree{
	private:
		node* root;

	public:
		binTree(){root = NULL;}
		node* getRoot(){return root;}
		void insert(int val){
			node* n = new node(val);
			node* pre = NULL;
			node* post = root;

			while (post != NULL){
				pre = post;

				if (val <= pre->value)
post = post->left;
				else post = post->right;
			}
			if (pre == NULL) root = n;
			else if (val <= pre->value) pre-
>left = n;
			else pre->right = n;
		}
};

int max (int n1, int n2){ return n1>n2 ? n1 : n2;}

int calculate(node* n){
	if (n->left == NULL && n->right == NULL) {
		n->h = 1;
		return 1;
	}
	int sx = 0;
	int dx = 0;
	if (n->left != NULL)  sx = calculate(n->left);
	if (n->right != NULL) dx = calculate(n->right);
	int h = max(sx, dx) + 1;
	n->h = h;
	return h;
}

bool check(node* n1, node* n2){return n1->h > n2->h ?
true : false;}

void print(node* n, int &k, int m){
	if (n->left != NULL) print(n->left, k, m);
	if (n->right != NULL) print(n->right, k, m);

	if (k > 0 && n->h == m) {
		cout<< n->value <<endl;
		k--;
	}
}
int main (){
	int n,k;
	cin>>n>>k;
	binTree b;
	for (int i = 0; i < n; i++) {
		int tmp = 0;
		cin>>tmp;
		b.insert(tmp);
	}
	calculate(b.getRoot());
	print(b.getRoot(), k, b.getRoot()->h/2);
	return 0;}
```

06/06/2019

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere. Per ogni nodo x si indica con $h(x)$ la massima distanza tra x e le foglie del suo sottoalbero. Per la radice r si definisce H come $H = h(r)$, e per ogni foglia f vale $h(f) = 1$.

Scrivere un programma che stampi in maniera non decrescente al più le prime K etichette tali per cui $h(x) = H/2$ (complessità al più $\mathcal{O}(n)$).

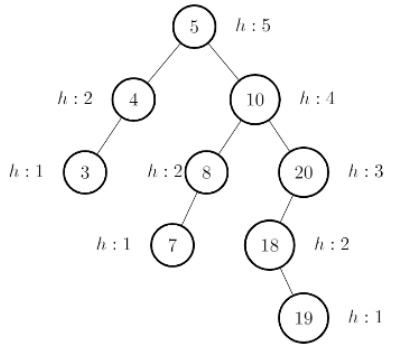
L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e K. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
9 2
5
4
10
3
8
20
7
18
19
```



Output

```
4
8
```

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>

using namespace std;

struct node{
	int value;
	node* left;
	node* right;
	int m;
	int p;
	int c;
	int d;
	node(int val, int massa): value(val), left(NULL),
right(NULL), m(massa), p(0), d(0), c(0){}
};

class binTree{
	private:
		node* root;
		vector<node*> v;

	public:
		binTree(){root = NULL;}
		node* getRoot(){return root;}

		void insert(int val, int m){
			node* tmp = new node(val, m);
			node* pre = NULL;
			node* post = root;
			int d = 0;
			while (post!=NULL){
				pre = post;
				if (val <= pre->value)
post = post->left;
				else post = post->right;
				d++;
			}
			tmp->d = d;
			v.push_back(tmp);
			if (pre == NULL) root = tmp;
			else if (val <= pre->value) pre-
>left = tmp;
			else pre->right = tmp;
		}
		vector<node*> getV(){return v;}
};

int calculate(node* n){
	if (n->left == NULL && n->right == NULL){
		int p = (n->m*2) - n->d;
		n->p = p;
		n->c = 0;
		return n->p;
	}
	int p = n->m - n->d;
	n->p = p;
	int sx = 0;
	int dx = 0;
	if (n->left != NULL)  sx = calculate (n->left);
	if (n->right != NULL) dx = calculate (n->right);
	int c = sx + dx;
	n->c = c;
	c += p;
	return c;
}

bool check(node* n1, node* n2){
	return n1->c > n2->c ? true : false;
}

int min(int n1, int n2){
	return n1<n2 ? n1 : n2;
}

void print(binTree b, int k){
	vector<node*> v = b.getV();
	sort(v.begin(), v.end(), check);
	int m = min(k, v.size());
	for (int i = 0; i < m; i++) cout<<v[i]->c<<endl;
}

int main (){
	int n, k;
	cin>>n>>k;
	binTree b;
	for (int i = 0; i < n; i++){
		int val = 0;
		int m = 0;
		cin>>val>>m;
		b.insert(val, m);
	}
	calculate(b.getRoot());
	print(b, k);
	return 0; }
```

18/02/2019

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere, in cui ciascun nodo è caratterizzato da una intero M, detto **massa**. Per ogni nodo x si indica con D la sua distanza dalla radice dell'albero. Per ogni nodo x può essere calcolato il suo **peso** P nella seguente maniera:

- Se x è foglia: $P = (M \times 2) - D$;
- In tutti gli altri casi: $P = M - D$;

Si definisce **carico** di uno nodo x, la somma dei pesi di tutti i nodi facenti parte del sottoalbero radicato in x, x escluso. Scrivere un programma che:

- legga da tastiera una sequenza di N coppie etichetta,massa e li inserisca in un ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti. (le etichette $\leq$ vanno inserite a sinistra).
- stampi al più i primi K valori di P ordinati in maniera decrescente (complessità al più $\mathcal{O}(nlogn)$).

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e K. Seguono N righe contenenti una coppia $\{etichetta, massa\}$ ciascuna. L'**output** contiene gli elementi della soluzione, uno per riga.

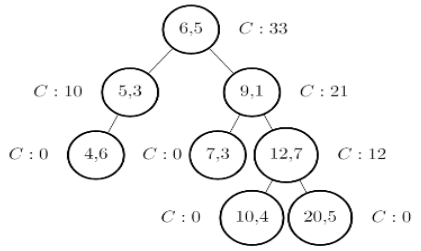
Esempio

Input

```
8 4
6 5
5 3
4 6
9 1
12 7
7 3
10 4
20 5
```

Output

```
33
21
12
10
```



```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct node{
	int value;
	node* left;
	node* right;
	int f;
	bool active;
	node(int val): value(val), right(NULL),
left(NULL), f(0), active(true){}

	int check(){
		if(left != NULL && right!= NULL &&
left->left == NULL && left->right == NULL &&
		right->left == NULL && right->right ==
NULL){ return 1;}

		else return 0;
	}
};

class binTree{
	private:
		node* root;
		vector<node*> v;
	public:
		binTree(){root = NULL;}
		node* getRoot(){return root;}
		void insert(int val){
			node* tmp = new node(val);
			node* post = root;
			node* pre = NULL;
			v.push_back(tmp);
			while (post!=NULL){
				pre = post;
				if (val <= pre->value)
post = post->left;
				else post = post->right;
			}
			if (pre == NULL) root = tmp;
			else if (val <= pre->value) pre-
>left = tmp;
			else pre->right = tmp;
		}
		vector<node*> getV(){return v;}

};

int f(node* n){
	bool neq = 1;

	if (n->left == NULL && n->right == NULL){
	//sono in una foglia
		n->active = false;
			//disattivo dal conteggio
		n->f = 0;
		return 0;
	}

	if(n->check() == 1) { //caso base eq
		n->f = -1;
		f(n->left);
		f(n->right);
		return -1;
	}


	if((n->left != NULL && n->right == NULL) || (n-
>left == NULL && n->right != NULL)){
		if (n->left != NULL){
	    if (n->left->left != NULL || n->left->right !=
NULL  ) neq = 0;
	  }

	 if (n->right != NULL){
	    if (n->right->left != NULL || n->right->right !=
NULL) neq = 0;
	  }
	else neq = 0;

	int sx = 0;
	int dx = 0;

	if (n->left != NULL)  sx = f(n->left);
	if (n->right != NULL) dx = f(n->right);


	int tot = sx + dx;
	if (neq) tot++;

	n->f = tot;
	return tot;
}

bool check(node* n1, node* n2){
	return n1->f > n2->f ? true : false;
}

void print(binTree b){
	vector<node*> v = b.getV();
	sort(v.begin(), v.end(), check);

	//Rimuovo i nodi disattivati
	for (int i = 0; i < v.size(); i++){
		if (v[i]->active == false) v[i] == NULL;
	}

	int i = 0;
	while (v[i]->active == false && i < v.size()) i++;

	node* winner = v[i];
	int value = v[i]->f;

	//scelgo l'etichetta più piccola a parità di f
	while (v[i]->f == value){
```

```
            if (v[i]->active && v[i]->value < winner->value) winner = v[i];
            i++;
        }

        cout<<winner->value<<endl<<winner->f<<endl;
}


int main (){

        int n;
        do{cin>>n;}while(n<=0);

        binTree b;
        for (int i = 0; i < n; i++){
            int tmp;
            cin>>tmp;
            b.insert(tmp);
        }

        f(b.getRoot());
        print(b);

        return 0;
}
```

31/01/2019

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere. Un nodo si dice di *equilibrato* se

- non è una foglia;
- ha entrambi i figli che sono foglie.

Un nodo si dice invece di *non_equilibrato* se

- non è una foglia;
- ha un solo figlio e quest'ultimo è una foglia.

Per ciascun nodo x si indica con eq e neq rispettivamente il numero di nodi *equilibrati* e *non_equilibrati* che si trovano nel sottoalbero radicato in x, x incluso. Infine, si definisce per ogni nodo x la funzione $f = neq - eq$. Scrivere un programma che:

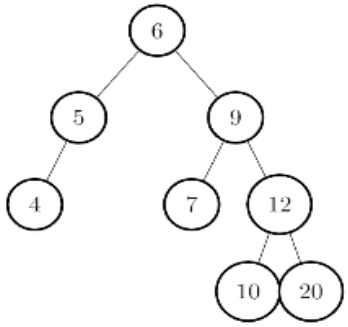
- legga da tastiera una sequenza di N interi e li inserisca in un ABR utilizzando il valore intero come etichetta. I valori devono essere inseriti nello stesso ordine con cui vengono letti. (le etichette $\leq$ vanno inserite a sinistra).
- calcoli f per ogni nodo x(complessità al più $\mathcal{O}(n)$).
- stampi l'etichetta e il valore di f del nodo con il valore di f più alto. A parità di f si considerino le etichette in ordine crescente (complessità al più $\mathcal{O}(n)$).

L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un *intero* ciascuna.
L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
8
6
5
4
9
12
7
10
20
```



Output

```
5
1
```

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


struct node{
	int value;
	node* left;
	node* right;
	int D;

	node(int val): value(val), left(NULL),
right(NULL), D(-1){}
	char getType(){return (value%2)==0 ? 'P' : 'D';}
};

class binTree{
	private:
		node* root;
		vector<node*> v;

	public:
		binTree(){
			root = NULL;
		}

		node* getRoot(){return root;}

		void insert(int value){
			node* tmp  = new node(value);
			v.push_back(tmp);
			node* pre  = NULL;
			node* post = root;

			while (post != NULL){
				pre = post;
				if (value <= pre->value)
post = post->left;
				else post = post->right;
			}

			if (pre == NULL) root = tmp;
			else if (value <= pre->value) pre-
>left = tmp;
			else pre->right = tmp;
		}


		vector<node*> getV(){ return v;}
};


bool check(node* n1, node* n2){ return n1->D > n2->D ?
true : false;}

void print(binTree b){
	vector<node*> v = b.getV();
	sort(v.begin(), v.end(), check);

	vector<int> output;
	for (int i = 0; i < v.size(); i++){
		if (v[i]->D == -2) break;

		if(i > 0 && v[i-1]->D != v[i]->D)
output.push_back(v[i]->D);
		if (i == 0) output.push_back(v[i]->D);
	}

	for (int i = 0; i < output.size(); i++){
		cout<<output[i]<<endl;
	}
}


int max(int n1, int n2){return n1>n2 ? n1: n2;}

int* D(node* n){

	if (n->left == NULL && n->right == NULL){
		n->D = -2;

		int* tmp = new int[2];
		if(n->getType() == 'D'){ //dispari ->
tmp[0] pari -> tmp[1]
			tmp[0] = 0;
			tmp[1] = -1;
		}

		else{
			tmp[0] = -1;
			tmp[1] = 0;
		}

		return tmp;
	}

	char type = n->getType();
	int distlp = -1;
	int distld = -1;
	int distrp = -1;
	int distrd = -1;


	if (n->left != NULL){
		int* v = D(n->left);
		distld = v[0];
	  distlp = v[1];
		delete[] v;

		if (distld != -1) distld++;
	  if (distlp != -1) distlp++;
	}

	if (n->right != NULL){
		int* v = D(n->right);
		distrd = v[0];
	  distrp = v[1];
		delete[] v;

		if (distrd != -1) distrd++;
```

```
        if (distrp != -1) distrp++;
    }


    int d = max(distld, distrd);
    int p = max(distlp, distrp);

    n->D = type=='D' ? d : p;
    int* tmp = new int[2];
    tmp[0] = d;
    tmp[1] = p;

    return tmp;
}


int main (){

    int n = 0;

    do{
        cin>>n;
    }while(n<=0);


    binTree b;
    for (int i = 0; i < n; i++){
        int tmp = 0;
        cin>>tmp;
        b.insert(tmp);
    }

    int* v = D(b.getRoot());
    delete[] v;

    print(b);

    return 0;
}
```

13/09/2018

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere. Un nodo si dice di *tipo* pari (dispari) se ha un'etichetta di valore pari (dispari).

Per ciascun nodo x che non sia una foglia, si definisce D come la massima distanza tra x e una sua foglia dello stesso tipo. Nel caso nel sottoalbero radicato in x non sia presente una foglia dello stesso tipo, $D = -1$.

Scrivere un programma che:

- legga da tastiera una sequenza di N interi e li inserisca in un ABR utilizzando il valore intero come etichetta. I valori devono essere inseriti nello stesso ordine con cui vengono letti. (le etichette $\leq$ vanno inserite a sinistra).
- calcoli D per ogni nodo non foglia (complessità al più $\mathcal{O}(n)$).
- stampare in ordine decrescente e senza duplicati i valori di D (complessità al più $\mathcal{O}(nlogn)$).

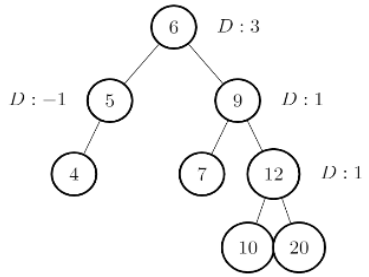
L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un *intero* ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
8
6
5
9
4
12
7
10
20
```



Output

```
3
1
-1
```

```
#include <iostream>
#include <vector>

using namespace std;

struct Node {
	int value;
	Node* left;
	Node* right;
	int c;
	int d;
	Node(int val): value(val), left(NULL),
right(NULL), c(0), d(0){}
      int check(){return value%2 == 0 ? 1 : 0;} //ritorna 0 se
è dispari e uno se è pari

	bool foglia(){
		return (left == NULL && right == NULL)
? true : false;
	}
};

class binTree{
	private:
		Node* root;
	public:
		binTree(){root = NULL;}
		Node* getRoot(){return root;}

		void insert(int val){
			Node* n = new Node(val);
			Node* pre = NULL;
			Node* post = root;
			while (post != NULL){
				pre = post;
				if (val<=pre->value)
post = post->left;
				else post = post->right;
			}
			if (pre == NULL) root = n;
		else if (val <= pre->value) pre->left = n;
				else pre->right = n;
			}
};
vector<int> calculate(Node* n){
	if (n->foglia()){
		vector<int> v;
		v.push_back(0);v.push_back(0);
		return v;
	}
	vector<int> sx; sx.push_back(0); sx.push_back(0);
	vector<int> dx; dx.push_back(0); dx.push_back(0);

	if (n->left != NULL)  {
		sx = calculate(n->left);
		if (n->left->foglia())
			if (n->left->check()  == n-
>check()) sx[1]+=1; else sx[0]+=1;
	}

	if (n->right != NULL)  {
	  dx = calculate(n->right);
	  if (n->right->foglia())
	      if (n->right->check()  == n->check())
dx[1]+=1; else dx[0]+=1;
	}
	n->c = sx[1]+dx[1]; //in posizione 0 ci sono i
discordi
	n->d = sx[0]+dx[0]; //in posizione 1 ci sono i
concordi

	vector<int> output;
	output.push_back(n->d);
	output.push_back(n->c);
	return output;
}
void print(Node* n){
	if (n->left != NULL) print(n->left);
	if (n->c - n->d >= 0) cout<<n->value<<endl;
	if (n->right != NULL) print(n->right);
}
int main(){
	int n;
	cin>>n;
	binTree b;
	for (int i = 0; i < n; i++){
		int tmp;
		cin>>tmp;
		b.insert(tmp);
	}
	calculate(b.getRoot());
	print(b.getRoot());
	return 0;}
```

01/02/2018

Esercizio

[**leggere il testo prestando particolare attenzione alle definizioni**]
Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Siano date le seguenti definizioni:

- una foglia si dice *concorde* se ha etichetta *pari* (*dispari*) ed è figlia di un padre con etichetta *pari* (*dispari*);
- una foglia si dice *discorde* se ha etichetta *pari* (*dispari*) ed è figlia di un padre con etichetta *dispari* (*pari*);
- Per ciascun nodo x, si indica con $nC(x)$ e $nD(x)$ il numero di foglie rispettivamente concordi e discordi del sottoalbero radicato in x. Per una foglia tali valori sono entrambi nulli.

Scrivere un programma che:

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti;
- stampi le etichette dei nodi tali per cui $nC(x) - nD(x) \geq 0$, ordinati per etichetta non decrescente. (complessità al più $\mathcal{O}(n)$)

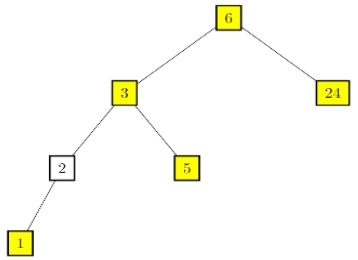
L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
6
6
3
2
5
24
1
```



Output

```
1
3
5
6
24
```

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

using namespace std;

const int nullptr = 0;

struct Node {
   int val;
   int dx;
   int lx;

   Node *left;
   Node *right;

   explicit Node(int n): val(n), dx(0), lx(0) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int n) {
   Node *NewNode = new Node(n);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      // Ogni volta che scendo di livello incremento la
distanza dalla radice
      NewNode->dx++;

      if (n <= actual->val)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->val)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

int CountLX(Node *root) {
   if (root == nullptr) return 0;
   int l = CountLX(root->left);
   int r = CountLX(root->right);

   root->lx = ((l > r)?l:r);

   return root->lx + 1;
}

void Print(Node *root, const int &K) {
   static int h = 0;
   if (root == nullptr) return;
   Print(root->left, K);

   if (h++ >= K) {
      h = 0;
      return;
   }

   if (root->lx - root->dx <= 1 &&
      root->lx - root->dx >= -1) {
      cout << root->val << endl;
   }

   Print(root->right, K);
}

int main(void) {
   int N, K;
   int tmp;

   cin >> N >> K;
   for (int i = 0; i < N; ++i) {
      cin >> tmp;
      Insert(tmp);
   }

   CountLX(root);
   Print(root, K);

   return 0;
}
```

29/06/2017

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi non negativi e li inserisca dentro una particolare *tabella hash*, in cui a ciascun indirizzo è associato un albero binario di ricerca (ABR). Ciascun ABR mantiene una sola entrata per ogni valore, tenendo traccia del numero di duplicati per ciascuno di essi. Dato un ABR, si definisce D come l'etichetta del nodo con più duplicati, e a parità di questi ultimi si considera il valore di etichetta più alto. Nel caso l'albero sia vuoto, $D = -1$.

Scrivere un programma che legga da tastiera una sequenza di N interi x e per ciascuno di essi

- individui l'indirizzo corrispondente utilizzando la seguente funzione hash:

  $$h(x) = \{[(\ a \times x\ ) +\ b\ ]\ \%\ p\ \}\ \%\ S$$

  dove p=999149 , a=1000 e b=2000;
- lo inserisca nell'ABR associato all'indirizzo calcolato al punto precedente.

Il programma dovrà poi stampare al più i primi K valori di D non negativi, ordinati in maniera decrescente.

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N, K e S separati da uno spazio. Seguono N righe contenenti un intero ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
6 2 3
9
17
16
7
6
7
```

Output

```
17
9
```

```
#include <iostream>

using namespace std;

const int nullptr = 0;

struct Node {
   int val;
   int L;

   Node *left;
   Node *right;

   explicit Node(int n): val(n), L(0) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int n) {
   Node *NewNode = new Node(n);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      if (n <= actual->val)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->val)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

int Proprieta(Node * root) {
   // La uso per ricordarmi se il padre è pari o dispari
   static bool father_pari;
   bool pari;
   if (root == nullptr) return 0;
   if (root->left != nullptr ||   // Se non siamo su una foglia
aggiorno il
       root->right != nullptr) {  // valore di father_pari
       pari = father_pari = (root->val % 2 == 0)? true: false;
   } else {
       // Se siamo su una foglia controllo se il valore
dell'etichetta è pari
       // e se è uguale al negato del padre (pari == !dispari) e
lo restituisco
       // come risultato (quindi 1 se la proprietà è verificata, 0
altrimenti)
       return ((root->val % 2 == 0)? true: false) == !
father_pari;
   }

   int LLeft, LRight;
   LLeft = Proprieta(root->left);
   // Nella chiamata ricorsiva il valore potrebbe cambiare, lo
riaggiorno
   father_pari = pari;
   LRight = Proprieta(root->right);

   root->L = LLeft + LRight;

   return root->L;
}

void Print(Node *root) {
   if (root == nullptr) return;
   Print(root->left);
   cout <<root->L << endl;
   Print(root->right);
}

int main(void) {
   int N;
   int tmp;
   cin >> N;
   for (int i = 0; i < N; ++i) {
      cin >> tmp;
      Insert(tmp);
   }
   Proprieta(root); // O(n)
   Print(root);  // O(n)

   return 0;
}
```

08/06/2017

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere. Si dice che un nodo x soddisfa la proprietà P se sono verificate le seguenti condizioni:

- x è una foglia;
- x ha etichetta di valore pari (dispari) ed è figlio di un nodo con etichetta dispari (pari);

Dato un nodo k si definisce inoltre la grandezza L come il numero di nodi facenti parte del sottoalbero radicato in k, escluso k stesso, che soddisfano la proprietà P.

Si scriva un programma che

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- consideri i nodi dell'albero in ordine non decrescente di etichetta e per ciascuno stampi il valore L corrispondente (complessità al più $\mathcal{O}(n)$).

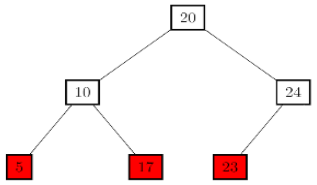
L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene i valori della soluzione, uno per riga.

Esempio

Input

```
6
20
10
5
17
24
23
```



Output

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

```
#include <iostream>
#include <cstring>

using namespace std;

const int nullptr = 0;

struct Node {
   int val;

   Node *left;
   Node *right;

   explicit Node(int n): val(n) {
      left = right = nullptr;
   }
} *root = nullptr;

inline int Max(const int &h) {
   if (h < 1) return 0;
   if (h == 1) return 1;
   return 2 << (h-2);
}

void Insert(const int n, int * const &v) {
   Node *NewNode = new Node(n);

   Node *actual, *prec;
   actual = prec = root;
   int h = 1;
   while (actual != nullptr) {
      prec = actual;
      h++;

      if (n <= actual->val)
         actual = actual->left;
      else
         actual = actual->right;
   }
   v[h-1]++;

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->val)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

void print(Node * root, const int &h) {
   static int lvl = 1;
   static bool flag;
   if (lvl <= 1) flag = false;
   if (root == nullptr || flag || lvl > h) return;
   if (lvl == h) {
      cout << root->val;
      flag = true;
      return;
   }

   lvl++;
   print(root->right, h);
   print(root->left, h);
   lvl--;
}

int main() {
   int N;
   int tmp;

   cin >> N;
   int *v = new int[N];
   memset(v, 0, sizeof(int) * N);

   for (int i = 0; i < N; ++i) {
      cin >> tmp;
      Insert(tmp, v);
   }

   float max_score = 0;
   int h_max_score = 1;
   for (int i = 0; i < N; ++i) {
      float score = (static_cast<float>(v[i])/(i+1))*Max(i+1);
      if (score >= max_score) {
         max_score = score;
         h_max_score = i + 1;
      }
   }

   print(root, h_max_score);
   return 0;
}
```

17/02/2017

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere. Per ogni livello h dell'albero si definisce lo score $S(h)$ come

$$S(h) = \frac{N(h)}{h} \times Max(h) \quad (1)$$

dove

- $N(h)$ è il numero di nodi che si trovano al livello h;
- $Max(h)$ è il numero massimo di nodi che possono trovarsi al livello h;

NOTA: la radice dell'albero si trova al livello 1; i nodi figli della radice si trovano al livello 2: etc.

Si scriva un programma che

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- individui il livello con score più alto; a parità di score si consideri il livello con valore più alto (complessità al più $\mathcal{O}(n)$);
- stampi il valore dell'etichetta più *grande* tra quelle che si trovano al livello trovato al punto precedente (complessità al più $\mathcal{O}(n)$).

L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene la soluzione.

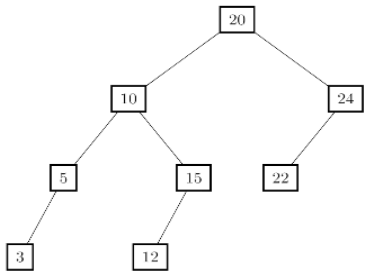
Esempio

Input

```
8
20
10
5
15
3
24
12
22
```

Output

```
12
```



```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int ZIGZAG = 0;

struct node{
	int value;
	node* left;
	node* right;

	node(int val): value(val), left(NULL), right(NULL)
{}
};

class binTree{
	private:
		node* root;

	public:
		binTree(){root = NULL;}
		node* getRoot(){return root;}

		void insert(int val){
			node* n = new node(val);
			node* pre = NULL;
			node* post = root;


			while (post!=NULL){
				pre = post;
				if (val <= pre->value)
post = post->left;
				else post = post->right;
			}

			if (pre == NULL) root = n;
			else if (val <= pre->value) pre-
>left = n;
			else pre->right = n;
		}
};


void zigzag(node* n){
	if(n->left != NULL && n->right != NULL){
		zigzag(n->left);
		zigzag(n->right);

		return;
	}

	if (n->left != NULL && n->right == NULL){
		if (n->left->right != NULL && n->left-
>left == NULL) ZIGZAG++;
		zigzag(n->left);
	}

	if (n->right != NULL && n->left == NULL){
	  if (n->right->left != NULL && n->right->right ==
NULL) ZIGZAG++;
	zigzag(n->right);
	}
}

int main(){
	int n;
	cin>>n;

	binTree b;
	for (int i = 0; i < n; i++){
		int tmp = 0;
		cin>>tmp;
		b.insert(tmp);
	}

	zigzag(b.getRoot());
	cout<<ZIGZAG<<endl;
	return 0;
}
```

31/01/2017

Esercizio

Si consideri un sistema per analizzare la struttura di alberi binari di ricerca (ABR) aventi etichette intere. L'obiettivo del sistema è contare il numero di **ZigZag** (definiti in seguito) all'interno dell'albero.
Due nodi x e y formano uno **ZigZag** se tutte le seguenti condizioni sussistono:

- x è nodo padre di y;
- x ha uno e un solo figlio, e tale figlio è destro (sinistro);
- y ha uno e un solo figlio, e tale figlio è sinistro (destro);

Si scriva un programma che

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- stampi il numero di **ZigZag** presenti nell'albero; (complessità al più $\mathcal{O}(n)$)

L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene la soluzione.

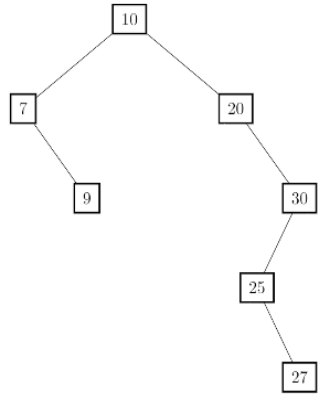
Esempio

Input

```
7
10
20
7
9
30
25
27
```

Output

```
2
```



```
#include <iostream>

using namespace std;

const int nullptr = 0;

struct Node {
   int label;
   int conto;

   Node *left;
   Node *right;

   explicit Node(int n):label(n), conto(0), left(nullptr),
right(nullptr) {}
} *root = nullptr;

void Insert(Node *tree, int val) {
   Node *NewNode = new Node(val);

   Node *actual, *prec;
   actual = prec = root;
   while (actual) {
      prec = actual;

      if (val <= actual->label)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (!root)
      root = NewNode;
   else if (val <= prec->label)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

int Conto(Node *tree) {
   if (tree == nullptr) return 0;
   int LConto = Conto(tree->left);
   int RConto = Conto(tree->right);

   if (!tree->left && !tree->right) {
      if (tree->label == 0)
         return 2;
      else if (tree->label % 2 == 0)
         return -1;
      else
         return 1;
   } else {
      tree->conto += LConto + RConto;
      return tree->conto;
   }
}

void Print(Node *tree, int K) {
   if (tree == nullptr) return;
   Print(tree->left, K);
   if (tree->conto > K)
      cout << tree->label << endl;
   Print(tree->right, K);
}

int main()  {
   int N, K;
   cin >> N >> K;

   for (int i = 0, tmp; i < N; ++i) {
      cin >> tmp;
      Insert(root, tmp);
   }

   Conto(root);
   Print(root, K);

   return 0;
}
```

11/01/2017

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) aventi etichette intere.
Dato un nodo x si definisce il valore intero $conto(x)$ calcolato nella seguente maniera e considerando il **sottoalbero radicato in x**:

- è inzialmente uguale a 0;
- +1 per ogni foglia con etichetta dispari;
- -1 per ogni foglia con etichetta pari;
- +2 per ogni foglia con etichetta di valore 0.

Nel caso x sia una foglia, $conto(x) = 0$.

Si scriva un programma che

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- calcoli il valore $conto(x)$ per ciascun nodo dell'albero; (complessità al più $\mathcal{O}(n)$)
- stampi le etichette dei nodi tali per cui $conto > K$, ordinati per etichetta in maniera non decrescente. (complessità al più $\mathcal{O}(n)$)

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e K separati da uno spazio. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

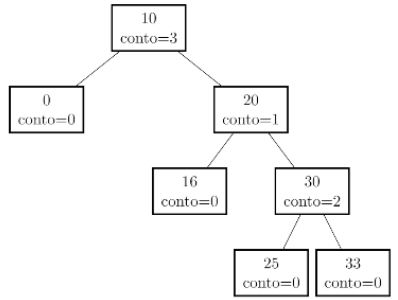
Esempio

Input

```
7 1
10
0
20
16
30
25
33
```

Output

```
10
30
```



```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <vector>

using namespace std;

const int nullptr = 0;

const int a = 1000;
const int b = 2000;
const int p = 999149;
int N, K, S;

inline int Hash(const int &ID) {
   return ((( a * ID ) + b ) % p ) % S;
}
struct Node {
   int val;
   Node *left;
   Node *right;

   explicit Node(int n): val(n), left(nullptr), right(nullptr) {}
} **table;

struct H {
   int height;
   int ID;
   H(int val, int id): height(val), ID(id) {}
   bool operator() (const H &a, const H &b) const {
      if (a.height == b.height)
         return a.ID < b.ID;
      return a.height > b.height;
   }
};

void Insert(Node *&root, const int &val) {
   Node *newNode = new Node(val);

   Node *actual, *prec;
   actual = prec = root;
   while(actual != nullptr) {
      prec = actual;
      if (val <= actual->val)
         actual = actual->left;
      else
         actual = actual->right;
   }
   if (root == nullptr)
      root = newNode;
   else if (val <= prec->val)
      prec->left = newNode;
   else
      prec->right = newNode;
}

int Height(Node *root) {
   if (root == nullptr) return 0;
   int HLeft = Height(root->left);
   int HRight = Height(root->right);
   return 1 + ((HLeft > HRight)? HLeft: HRight);
}

void PrintThree(Node *root) {
   if (root == nullptr) return;
   PrintThree(root->left);
   cout << root->val << endl;
   PrintThree(root->right);
}

int main(void) {
   cin >> N >> K >> S;
   table = new Node *[S];
   memset(table, nullptr, sizeof(Node *) * S);

   for (int i = 0; i < N; ++i) {
      int hash, val;
      cin >> val;
      hash = Hash(val);
      Insert(table[hash], val);
   }
   vector<H> v;
   for (int i = 0; i < S; ++i) {
      H h(Height(table[i]), i);
      v.push_back(h);
   }
   H h(0,0);
   sort(v.begin(), v.end(), h);
   K = ( K > S )? S: K;
   for (int i = 0; i < K; ++i) {
      cout << v[i].ID << endl;
   }
   return 0;
}
```

12/09/2016

Esercizio

Si consideri un sistema di memorizzazione che legga una sequenza di N interi e li inserisca dentro una particolare *tabella hash*, in cui a ciascun indirizzo è associato un albero binario di ricerca (ABR). Scrivere un programma che legga da tastiera una sequenza di N interi x e per ciascuno di essi

- individui l'indirizzo corrispondente utilizzando la seguente funzione hash:

$$h(x) = \{[(a \times ID) + b] \% p \} \% S$$

  dove p=999149 , a=1000 e b=2000;

- lo inserisca nell'ABR associato all'indirizzo calcolato al punto precedente.

Il programma dovrà poi calcolare l'altezza di ciascun ABR e stampare i primi K indirizzi della *tabella Hash*, ordinati in maniera decrescente in base all'altezza dell'ABR corrispondente. A parità di altezza scegliere l'indirizzo con valore minore. Nel caso il numero di indirizzi sia inferiore a K, stampare quelli disponibili.

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N, K e S separati da uno spazio. Seguono N righe contenenti un intero ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
6 2 3
9
17
16
7
6
3
```

Output

```
2
0
```

```
#include <iostream>

using namespace std;

const int nullptr = 0;

struct Node {
   int val;
   int dx, lx;

   Node *left;
   Node *right;

   explicit Node(int n): val(n), left(nullptr), right(nullptr) {
      dx = lx = 0;
   }
} *root;

int Insert(Node *&root, const int &n) {
   static int count = 0;  // Variabile contenente la distanza
dalla radice
   if (root == nullptr) {
      root = new Node(n);
      root->dx = count;
      // Dopo l'inserimento si azzera count siccome è una
variabile statica
      count = 0;
      return 1;
   }

   int HLeft, HRight;
   HLeft = 0;
   HRight = 0;
   count++;
   if (n <= root->val) {
      HLeft  = Insert(root->left, n);
      if (root->right != nullptr)
         HRight = 1 + root->right->lx;
   } else {
      if (root->left != nullptr)
         HLeft = 1 + root->left->lx;
      HRight = Insert(root->right, n);
   }

   root->lx = ((HLeft > HRight)? HLeft: HRight);

   return 1 + root->lx;
}

int N, K, J;
void Print(Node *root) {
   if (root == nullptr) return;
   Print(root->left);
   if (root->lx - root->dx <= 1 && root->lx - root->dx >= -1
      && J++ < K)
      cout << root->val << endl;
   Print(root->right);
}

int main() {
   cin >> N >> K;

   for (int i = 0; i < N; ++i) {
      int tmp;
      cin >> tmp;
      Insert(root, tmp);
   }
   J = 0;
   Print(root);

   return 0;
}
```

20/07/2016

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) in grado di memorizzare nodi ad etichette intere.
Dato un nodo x:
- si definisce d_x come la distanza tra x e la radice;
- si definisce l_x come la massima distanza tra x e le foglie contenute nel sottoalbero radicato in esso;
- x si dice **mediano** se i valori di l_x e d_x differiscono al più di uno.

Si scriva un programma che

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- verifichi per ogni nodo che sia mediano o meno; (complessità al più $\mathcal{O}(n)$)
- stampi le etichette dei primi K nodi mediani ordinati per etichetta in maniera non decrescente. Nel caso i valori fossero meno di K, stampare quelli disponibili. (complessità al più $\mathcal{O}(n)$)

NON è permesso utilizzare strutture dati (array/vector) di appoggio.

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e K separati da uno spazio. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

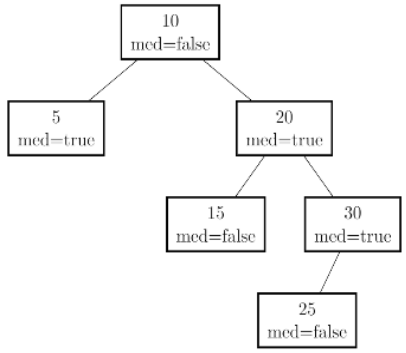
Esempio

Input

```
6 2
10
5
20
15
30
25
```

Output

```
5
20
```



```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const int nullptr = 0;

int N, K;
vector<int> v;

struct Node {
   int label;
   int d, lsx, ldx;

   Node *left;
   Node *right;

   explicit Node(int n):label(n), left(nullptr), right(nullptr) {
      d = lsx = ldx = 0;
   }
} *root = nullptr;

void Insert(Node *&root, int n) {
   Node *NewNode = new Node(n);

   int d = 0;
   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;
      if (actual->label >= n)
         actual = actual->left;
      else
         actual = actual->right;
      d++;
   }

   NewNode->d = d;

   if (root == nullptr)
      root = NewNode;
   else if (prec->label >= n)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

void LsxLdx(Node *&root) {
   if (root == nullptr) return;
   LsxLdx(root->left);
   LsxLdx(root->right);
   if (root->left) {
      if (root->left->left == nullptr && root->left->right ==
nullptr)
         root->lsx++;
      root->lsx += root->left->lsx;
      root->ldx += root->left->ldx;
   }
   if (root->right) {
      if (root->right->left == nullptr && root->right->right
== nullptr)
         root->ldx++;
      root->lsx += root->right->lsx;
      root->ldx += root->right->ldx;
   }
}

void CalcV(Node *&root) {
   if (root == nullptr) return;
   v.push_back((root->d * root->lsx) + (K * root->ldx) );
   CalcV(root->left);
   CalcV(root->right);
}

int main(void) {
   int tmp;
   cin >> N >> K;
   for (int i = 0; i < N; ++i) {
      cin >> tmp;
      Insert(root, tmp);
   }
   LsxLdx(root);

   CalcV(root);
   sort(v.begin(), v.end());
   for (vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
      cout << *i << endl;
   }
   return 0;
}
```

29/06/2016

Esercizio

[**leggere il testo prestando particolare attenzione alle definizioni**]
Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) in grado di memorizzare nodi ad etichette intere. Siano date le seguenti definizioni:
- una foglia si dice *sinistra* (*destra*) se è figlio sinistro (destro) di un nodo padre;
- per ogni nodo x si definisce lsx (ldx) il numero delle foglie sinistre (destre) che fanno parte del sottoalbero radicato in x;
- per ogni nodo si definisce d la distanza di detto nodo dalla radice dell'albero;
- per ogni nodo si definisce $v = (d \times lsx) + (K \times ldx)$, con K valore intero.

Si scriva un programma che

- legga da tastiera N etichette e le inserisca all'interno dell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- calcoli v per ogni nodo dell'albero; (complessità al più $\mathcal{O}(n)$)
- stampi i valori v ordinati in maniera non decrescente. (complessità al più $\mathcal{O}(n \log n)$)

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e K separati da uno spazio. Seguono N righe contenenti un'etichetta ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

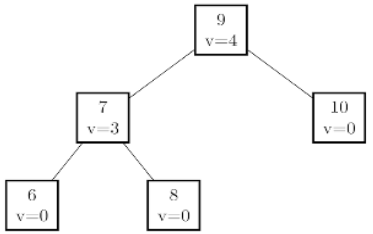
Esempio

Input

```
5 2
9
7
6
8
10
```

Output

```
0
0
0
3
4
```



```
#include <iostream>

using namespace std;

const int nullptr = 0;

struct Node {
   int ID;
   int P;
   int Cmax;
   int C;
   Node *left;
   Node *right;

   explicit Node(int id, int p, int cmax): ID(id), P(p),
Cmax(cmax), C(0) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int id, const int p, const int cmax) {
   Node *NewNode = new Node(id, p, cmax);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      if (id <= actual->ID)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (id <= prec->ID)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}
```

```
bool Integrity(Node *&root) {
   if (root == nullptr) return true;
   bool integrity = Integrity(root->left);
   integrity = Integrity(root->right) && integrity;
   if (root->left != nullptr)
      root->C += root->left->C + root->left->P;
   if (root->right != nullptr)
      root->C += root->right->C + root->right->P;

   if (root->C > root->Cmax) return false;
   return integrity;
}

void Print(Node *root) {
   if (root == nullptr) return;
   Print(root->left);
   if (root->C > root->Cmax)
      cout << root->ID << endl;
   Print(root->right);
}

int main(void) {
   int N;
   int id, p, cmax;
   cin >> N;

   for (int i = 0; i < N; ++i) {
      cin >> id >> p >> cmax;
      Insert(id, p, cmax);
   }

   if(!Integrity(root)){
      cout << "no" << endl;
      Print(root);
   } else {

      cout << "ok" << endl;
   }
   return 0;
}
```

08/06/2016

Esercizio

Si consideri un sistema per la gestione di alberi binari di ricerca (ABR) in grado di memorizzare nodi con le seguenti caratteristiche.

Ogni nodo è caratterizzato da un intero ID, da un intero P che ne rappresenta il **peso**, e da un intero $Cmax$ che ne rappresenta il **carico massimo**. Il **carico** di un nodo n è definito come la somma dei pesi di tutti i nodi facenti parte del sottoalbero radicato in n, escluso quest'ultimo. Il carico di un nodo è **tollerabile** se $\leq$ rispetto al suo carico massimo. Un ABR si dice *integro* se tutti i suoi nodi hanno un carico tollerabile.

Il sistema dovrà inserire i nodi all'interno dell'ABR usando l'ID come etichetta. I valori devono essere inseriti nello stesso ordine con cui vengono letti (per convenzione, le etichette $\leq$ vanno inserite a sinistra).

Scrivere un programma che:

- legga da tastiera una sequenza di N triple $[ID, P, Cmax]$ ciascuna rappresentante un nodo, e le inserisca in un ABR;
- stampi la stringa 'ok' se l'ABR è integro e 'no' altrimenti;
- in caso l'ABR **non** sia integro, stampi l'ID dei nodi con carico **non** tollerabile in ordine di ID non decrescente.

L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti una tripla $[ID, P, Cmax]$ ciascuna, con gli elementi separati da uno spazio.

L'**output** contiene gli elementi della soluzione, uno per riga.

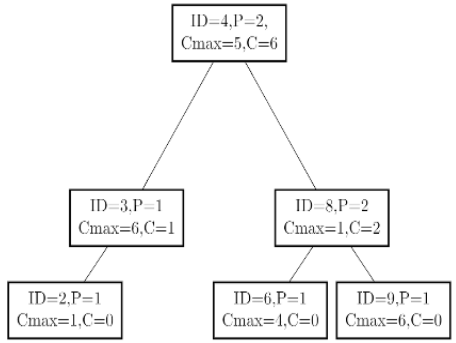
Esempio

Input

```
6
4 2 5
8 2 1
6 1 4
9 1 6
3 1 6
2 1 1
```

Output

```
no
4
8
```



```
#include <iostream>
#include <string>
#include <cstring>

using namespace std;

const int nullptr = 0;

const int p = 999149;
const int a = 1000;
const int b = 2000;

inline int Hash(const int &id, const int &N) {
   return (((a * id) + b) % p) % (2 * N);
}

struct Conto {
   int ID;
   string Cognome;
   Conto *next;

   Conto(int id, string cognome):
      ID(id),
      Cognome(cognome),
      next(nullptr) {}
} **hashTable;

void Insert(int id, string cognome, int N) {
   Conto *NewConto = new Conto(id, cognome);

   int hash = Hash(id, N);

   if (hashTable[hash] == nullptr) {
      hashTable[hash] = NewConto;
   } else {
      Conto *actual, *prec;
      actual = prec = hashTable[hash];
      while (actual != nullptr && ( cognome > actual-
>Cognome ||
            cognome == actual->Cognome && id > actual-
>ID) ) {
         prec = actual;
         actual = actual->next;
      }

      NewConto->next = actual;
      if (actual == hashTable[hash])
         hashTable[hash] = NewConto;
      else
         prec->next = NewConto;
   }
}

void Print(const int &N) {
   int maxId = 0;
   int maxVal = 0;
   Conto *tmp;
   for (int i = 0; i < 2*N; ++i) {
      int count = 0;
      tmp = hashTable[i];
      while (tmp) {
         count++;
         tmp = tmp->next;
      }
      if (count > maxVal) {
         maxVal = count;
         maxId = i;
      }
   }

   cout << hashTable[maxId]->ID << endl;
}

int main(void) {
   int N;
   cin >> N;

   hashTable = new Conto *[2*N];
   memset(hashTable, nullptr, sizeof(Conto *) * 2*N);

   int id;
   string cognome;
   for (int i = 0; i < N; ++i) {
      cin >> id >> cognome;
      Insert(id, cognome, N);
   }
   Print(N);
   return 0;
}
```

27/01/2016

Esercizio

Si consideri un sistema bancario che memorizza i dati relativi a conti correnti. Ciascun conto corrente è identificato da un **ID** univoco e intero, e da una stringa **Cognome** che ne specifica il titolare. Il sistema mantiene questi dati dentro una tabella hash con liste di trabocco (metodo di concatenazione). Scrivere un programma che

- legga da tastiera una sequenza di N coppie (**ID**,**Cognome**) ciascuna rappresentante un conto corrente;
- salvi dentro una *tabella Hash* le informazioni relative ai conti correnti, utilizzando la seguente funzione hash:

  $$h(ID) = \{|(\ a \times ID\ ) +\ b\ |\ \%\ p\ \}\ \%\ 2N$$

  dove p=999149 , a=1000 e b=2000;
- identifichi l'indirizzo della *tabella Hash* che ha generato più collisioni. A parità di numero di collisioni scegliere l'indirizzo con indice minore;
- dato l'indirizzo ottenuto al passo precedente, stampi l'**ID** del primo elemento secondo l'ordine lessicografico per **Cognome**. A parità di **Cognome**, scegliere l'elemento con **ID** più basso.

L'**input** è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti una coppia (**intero**,**stringa**).

L'**output** contiene la soluzione.

Esempio

Input

```
5
43047 Kilmister
43046 Bowie
58 Frey
1957 Scola
1900 Rickman
```

Output

```
1900
```

```
#include <iostream>
#include <cstring>

using namespace std;
const int nullptr = 0;

struct Node {
   int valore;
   Node *left;
   Node *right;
   explicit Node(int val): valore(val), left(nullptr),
right(nullptr) {}
} **threes;

void InsertAt(int val, int id, const int &D) {
   if (id >= D) return;
   Node *NewNode = new Node(val);
   Node* actual, *prec;
   actual = prec = threes[id];
   while(actual != nullptr) {
      prec = actual;
      if (val <= actual->valore)
         actual = actual->left;
      else
         actual = actual->right;
   }
   if (threes[id] == nullptr)
      threes[id] = NewNode;
   else if (val <= prec->valore)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}
int Height(Node *root) {
   if (root == nullptr) return 0;
if (root->left == nullptr && root->right == nullptr) return 0;
   int HLeft = Height(root->left);
   int HRight = Height(root->right);
   return 1 + ((HLeft > HRight)? HLeft: HRight);
}
void LowHigh(int &low, int &high, const int &D) {
   low = high = 0;
   if (D <= 0) return;
   int HLow, HHigh;
   HLow = HHigh = Height(threes[0]);
   for (int i = 0; i < D; ++i) {
      int tmp = Height(threes[i]);
      if (tmp < HLow) {
         HLow = tmp;
         low = i;
      }
      if (tmp >= HHigh) {
         HHigh = tmp;
         high = i;
      }
   }
}
void MergeLowHigh(Node *low, const int &high, const int
&D) {
   if (low == nullptr) return;
   MergeLowHigh(low->left, high, D);
   InsertAt(low->valore, high, D);
   MergeLowHigh(low->right, high, D);
}
void PrintThree(Node *root) {
   if (root == nullptr) return;
   PrintThree(root->left);
   cout << root->valore << endl;
   PrintThree(root->right);
}
void PrintLeaves(Node *root) {
   if (root == nullptr) return;
   PrintLeaves(root->left);
   if (!root->left && !root->right)
      cout << root->valore << endl;
   PrintLeaves(root->right);
}
int main(void) {
   int N, D;
   cin >> N >> D;
   threes = new Node *[D];
   memset(threes, nullptr, sizeof(Node *) * D);
   for (int i = 0; i < N; ++i) {
      int val, id;
      cin >> val >> id;

      InsertAt(val, id, D);
   }
   int low, high;
   LowHigh(low, high, D);
   MergeLowHigh(threes[low], high, D);
   PrintLeaves(threes[high]);
   return 0;
}
```

11/01/2016

Esercizio

Scrivere un programma che legga da tastiera una sequenza di N coppie di interi $[valore, ID]$ e le utilizzi per riempire D alberi binari di ricerca (**senza** ribilanciamento). Ciascun albero è caratterizzato da un ID intero compreso tra 0 e $D-1$. Per ciascuna coppia letta, il primo intero indica il valore che deve essere inserito, mentre il secondo intero indica l'ID dell'albero di destinazione. I valori devono essere inseriti nello stesso ordine con cui vengono letti. (per convenzione, i valori $\leq$ vanno inseriti a sinistra)

Il programma deve:

- identificare *low* come l'ID dell'albero con la più piccola altezza. A parità di altezza, scegliere l'albero con ID minore.
- identificare *high* come l'ID dell'albero con la più grande altezza. A parità di altezza, scegliere l'albero con ID maggiore.
- considerare i nodi dell'albero con ID *low* in ordine crescente e inserirli nell'albero con ID *high*
- stampare in ordine crescente le etichette delle foglie dell'albero risultante

NOTA: L'altezza del nodo radice è **0**.

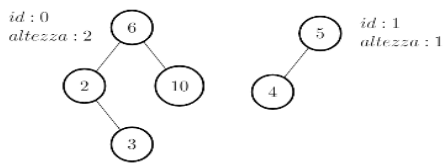
L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e D separati da uno spazio. Seguono N righe contenenti una coppia $[intero, intero]$ ciascuna, con gli elementi della coppia separati da uno spazio.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
6 2
6 0
2 0
5 1
4 1
10 0
3 0
```



Output

```
5
10
```

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <vector>

using namespace std;
const int nullptr = 0;

struct Node {
   int valore;
   Node *left;
   Node *right;

   explicit Node(int val): valore(val), left(nullptr),
right(nullptr) {}
} **threes;

struct elem {
   int ID;
   int NAtH;
   bool operator()(elem a, elem b) {
      if (a.NAtH == b.NAtH) return a.ID > b.ID;
      return a.NAtH > b.NAtH;
   }
};

void InsertAt(int val, int id, const int &D) {
   if (id >= D) return;
   Node *NewNode = new Node(val);
   Node* actual, *prec;
   actual = prec = threes[id];
   while(actual != nullptr) {
      prec = actual;
      if (val <= actual->valore)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (threes[id] == nullptr)
      threes[id] = NewNode;
   else if (val <= prec->valore)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}
int Height(Node *root) {
   if (root == nullptr) return 0;
   if (root->left == nullptr && root->right == nullptr) return
0;
   int HLeft = Height(root->left);
   int HRight = Height(root->right);
   return 1 + ((HLeft > HRight)? HLeft: HRight);
}
int MinHeight(const int &D) {
   if (D <= 0) return 0;
   int min = Height(threes[0]);
   for (int i = 1, tmp; i < D; ++i)
      if (min > (tmp = Height(threes[i])) ) min = tmp;

   return min;
}
int NodesAtH(Node *tree, int h, int k = 0) {
   if (tree == nullptr) return 0;
   if (k > h) return 0;

   int out = NodesAtH(tree->left, h, k+1) + NodesAtH(tree-
>right, h, k+1);
   if (h == k)
      out++;
   return out;
}
int main(void) {
   int N, D;
   cin >> N >> D;
   threes = new Node *[D];
   memset(threes, nullptr, sizeof(Node *) * D);
   for (int i = 0; i < N; ++i) {
      int val, id;
      cin >> val >> id;
      InsertAt(val, id, D);
   }
   int min = MinHeight(D);
   vector<elem> v;
   for (int i = 0; i < D; ++i) {
      elem tmp;
      tmp.ID = i;
      tmp.NAtH = NodesAtH(threes[i], min);
      v.push_back(tmp);
   }
   sort(v.begin(), v.end(), *v.begin());
  for(vector<elem>::iterator it=v.begin(); it!=v.end(); ++it) {
      cout << it->ID <<  endl;
   }
   return 0;
}
```

20/07/2015

Esercizio

Scrivere un programma che legga da tastiera una sequenza di N coppie di interi $[valore, ID]$ e le utilizzi per riempire D alberi binari di ricerca (**senza** ribilanciamento). Ciascun albero è caratterizzato da un ID intero compreso tra 0 e $D-1$. Per ciascuna coppia letta, il primo intero indica il valore che deve essere inserito, mentre il secondo intero indica l'ID dell'albero di destinazione. I valori devono essere inseriti nello stesso ordine con cui vengono letti. (per convenzione, i valori $\leq$ vanno inseriti a sinistra)

Il programma deve:

- calcolare h come la più piccola altezza tra quelle di tutti gli alberi;
- stampare l'ID degli alberi ordinati in maniera decrescente in base al numero di nodi che ciascuno di essi ha ad altezza h. In caso di parità, ordinare per id decrescente.

NOTA: L'altezza del nodo radice è **0**.

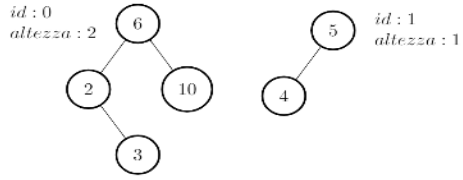
L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N e D separati da uno spazio. Seguono N righe contenenti una coppia $[intero, intero]$ ciascuna, con gli elementi della coppia separati da uno spazio.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input

```
6 2
6 0
2 0
5 1
4 1
10 0
3 0
```



Output

```
0
1
```

```
#include <iostream>

using namespace std;

const int nullptr = 0;

struct Node {
   int label;
   int lsum;
   int rsum;

   Node *left;
   Node *right;

   explicit Node(int n):label(n), lsum(0), rsum(0) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(Node *tree, int val) {
   Node *NewNode = new Node(val);

   Node *actual, *prec;
   actual = prec = root;
   while (actual) {
      prec = actual;

      if (val <= actual->label)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (!root)
      root = NewNode;
   else if (val <= prec->label)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}
```

```
void Sum(Node *tree) {
   if (tree == nullptr) return;
   Sum(tree->left);
   Sum(tree->right);

   if (tree->left != nullptr)
      tree->lsum = tree->left->lsum + tree->left->rsum +
tree->left->label;
   if (tree->right != nullptr)
      tree->rsum = tree->right->lsum + tree->right->rsum +
tree->right->label;
}

void Print(Node *tree, const int &K) {
   if (tree == nullptr) return;
   Print(tree->left, K);
   if (tree->lsum * K < tree->rsum)
      cout << tree->label << ' ';
   Print(tree->right, K);
}

int main()  {
   int N, K;
   cin >> N >> K;

   for (int i = 0, tmp; i < N; ++i) {
      cin >> tmp;
      Insert(root, tmp);
   }

   Sum(root);
   Print(root, K);

   return 0;
}
```


Esercizio

Scrivere un programma che legga da tastiera un sequenza di N interi positivi e li inserisca in un albero binario di ricerca (**senza** ribilanciamento) nello stesso ordine con il quale vengono forniti in input.

Per ogni nodo u dell'albero si definiscono le seguenti grandezze

- $S(u)$ come la somma delle chiavi dei nodi del sottoalbero sinistro radicato in u;
- $D(u)$ come la somma delle chiavi dei nodi del sottoalbero destro radicato in u.

Si dice che il nodo u soddisfa la proprietà P se $S(u) \times K < D(u)$, con K intero ≥ 0.

Il programma deve stampare in ordine non decrescente le chiavi dei nodi che soddisfano la proprietà P.

L'input è formattato nel seguente modo: la prima riga contiene l'intero N, la seconda contiene l'intero K. Seguono N righe contenenti un intero ciascuna.

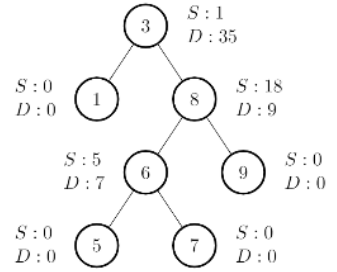
L'output è formato da una sola riga contenente gli elementi della soluzione separati da uno spazio.

NOTA: La soluzione deve avere complessità lineare nel numero di nodi.

Esempio

Input

```
7
2
3
8
6
7
5
9
1
```



Output

```
3
```

```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

using namespace std;

const int nullptr = 0;

struct Node {
   int val;

   Node *left;
   Node *right;

   explicit Node(int n): val(n) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int n) {
   Node *NewNode = new Node(n);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      if (n <= actual->val)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->val)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}
```

```cpp
int Verifica(Node *root, bool &flag) {
   if (root == nullptr) return 0;

   // La proprietà deve essere vera per ogni nodo, se non è verificata per un
   // nodo posso fermarmi senza visitare il resto dell'albero
   if (flag == false) return 0;

   int LHeight = Verifica(root->left, flag);
   int RHeight = Verifica(root->right, flag);

   flag = flag && (LHeight - RHeight >= -1 && LHeight - RHeight <= 1);

   return ((LHeight > RHeight)? LHeight:RHeight) + 1;
}

int main(void) {
   int N;
   int tmp;

   cin >> N;
   for (int i = 0; i < N; ++i) {
      cin >> tmp;
      Insert(tmp);
   }

   bool f = true;
   Verifica(root, f); // Si assume che `f` inizialmente sia impostato a `true`
   cout << (f? "ok":"no");

   return 0;
}
```

12/01/2015

Esercizio

Scrivere un programma che legga da tastiera un sequenza di N interi positivi e li inserisca in un albero binario di ricerca (**senza** ribilanciamento) nello stesso ordine con il quale vengono forniti in input.

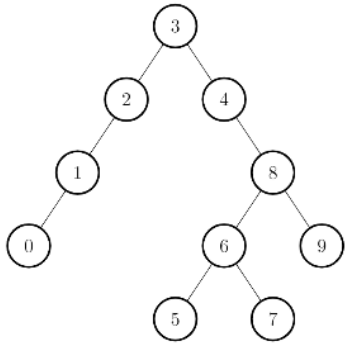
Il programma deve verificare che l'albero soddisfi la seguente proprietà: *per ogni nodo* dell'albero, le altezze dei suoi sottoalberi sinistro e destro devono differire al massimo di uno.

L'input è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un intero ciascuna. L'output è formato da una sola riga contenente la stringa *ok* qualora la proprietà sopra descritta sia verificata, la stringa *no* altrimenti.

Esempio 1

Input

```
10
3
2
1
4
0
8
6
7
5
9
```



Output

```
no
```

```
#include <iostream>

using namespace std;

const int nullptr = 0;

struct Node {
   int val;
   int I;
   int F;

   Node *left;
   Node *right;

   explicit Node(int n): val(n), I(0), F(n) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int n) {
   Node *NewNode = new Node(n);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      if (n <= actual->val)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->val)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}
```

```
int CalcIF(Node *root) {
   if (root == nullptr) return 0;
   int a = CalcIF(root->left);
   int b = CalcIF(root->right);
   if (root->left == nullptr && root->right == nullptr) {
      return 0;
   } else {
      root->F = (root->left != nullptr)? root->left->F: 0;
      root->F += (root->right != nullptr)? root->right->F: 0;
      root->I = root->val + a + b;
      return root->I;
   }
}

void Print(Node *root) {
   if (root == nullptr) return;
   Print(root->left);
   if (root->I <= root->F)
      cout << root->val << ' ';
   Print(root->right);
}

int main(void) {
   int N;
   int tmp;


   cin >> N;
   for (int i = 0; i < N; ++i) {
      cin >> tmp;
      Insert(tmp);
   }

   CalcIF(root);  // O(n)
   Print(root);   // O(n)

   return 0;
}
```

02/07/2014

Esercizio

Scrivere un programma che legga da tastiera un sequenza di N interi positivi e li inserisca in un albero di ricerca (**senza** ribilanciamento) nello stesso ordine con il quale vengono forniti in input.

Per ogni nodo u, si definiscono

- $I(u)$ come la somma delle chiavi dei nodi **interni** del sottoalbero radicato in u, includendo la chiave di u;
- $F(u)$ come la somma delle chiavi delle **foglie** nel sottoalbero radicato in u.

Il programma deve stampare, **in ordine non decrescente**, le chiavi di tutti i nodi u tali che $I(u) \leq F(u)$.

Non è consentito usare array ausiliari per memorizzare i valori delle chiavi.

L'input è formattato nel seguente modo: la prima riga contiene l'intero N. Seguono N righe contenenti un intero ciascuna.

L'output è formato da una sola riga contenente gli elementi della soluzione separati da uno spazio.

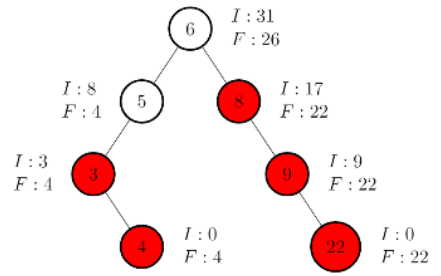
Esempio

Input

```
7
6
5
3
4
8
9
22
```

Output

```
3 4 8 9 22
```



La figura dell'esempio mostra anche, per ogni nodo u, i valori di $I(u)$ e $F(u)$. In rosso si evidenziano i nodi u tali che $I(u) \leq F(u)$.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

using namespace std;

const int nullptr = 0;

struct Node {
   int intero;
   string stringa;

   Node *left;
   Node *right;

   explicit Node(int n, const string &str): intero(n),
stringa(str) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int n, const string &str) {
   Node *NewNode = new Node(n, str);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      if (n <= actual->intero )
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->intero)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

void CognomiTarget(Node *root, const int&D,
vector<string> &v, int h = 0) {
   if (root == nullptr) return;
   if (h > D) return; // Evito di scendere ai livelli inferiori al
problema
   CognomiTarget(root->left, D, v, h+1);
   if (h == D)
      v.push_back(root->stringa);
   CognomiTarget(root->right, D, v, h+1);
}

int main(void) {
   int N, D;
   int intero;
   string stringa;


   cin >> N >> D;
   for (int i = 0; i < N; ++i) {
      cin >> intero >> stringa;
      Insert(intero, stringa);
   }

   vector<string> v;
   CognomiTarget(root, D, v);  // O(n)
   sort(v.begin(), v.end());   // k = 2^D, O(klogk)

   cout << v.size() << endl;

   for (vector<string>::iterator it = v.begin(); it != v.end(); +
+it) {
      cout << *it << endl;
   }

   return 0;
}
```

10/06/2014

Esercizio

Scrivere un programma che legga da tastiera un intero N, una sequenza A di N coppie [**intero,stringa**] ed un intero D. Ogni coppia rappresenta il nodo di un albero binario di ricerca. Il programma deve:

- Inserire uno alla volta, nell'ordine dato, le coppie [**intero,stringa**] di A in un albero binario di ricerca **senza ribilanciamento**. L'inserimento deve essere tale per cui, per un qualsiasi nodo, il sottalbero sinistro contenga le coppie il cui valore intero è **minore o uguale** del valore intero del nodo, mentre il sottoalbero destro contiene le coppie il cui valore intero è **maggiore**.
- Stampare **in ordine lessicografico** tutte le stringhe che si trovano nell'albero in nodi a profondità D. Si ricorda che la profondità di un nodo è uguale alla sua **distanza dalla radice e che la profondità della radice è 0**

L'input è formattato nel seguente modo. La prime due righe contengono i due interi N e D. Seguono $2N$ righe, due righe per coppia. Per ogni coppia abbiamo la prima riga che contiene il valore intero e la seconda che contiene la stringa associata. Si assume che N è sempre maggiore di zero.

L'output, se esistono nodi a profondità D, deve stampare nella prima riga il numero di nodi trovati e successivamente una riga per ogni stringa trovata. **Si ricorda che le stringhe vanno restituite in ordine lessicografico**. Se non esistono nodi a profondità D, il programma stampa solo 0.

```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

using namespace std;

const int nullptr = 0;

struct Node {
   int key;
   string val;

   Node *left;
   Node *right;

   explicit Node(int n, const string &v): key(n), val(v) {
      left = right = nullptr;
   }
} *root = nullptr;

void Insert(const int &n, const string &v) {
   Node *NewNode = new Node(n, v);

   Node *actual, *prec;
   actual = prec = root;
   while (actual != nullptr) {
      prec = actual;

      if (n <= actual->key)
         actual = actual->left;
      else
         actual = actual->right;
   }

   if (root == nullptr)
      root = NewNode;
   else if (n <= prec->key)
      prec->left = NewNode;
   else
      prec->right = NewNode;
}

void foo(Node *root, const int &K, vector<string> &v) {
   if (root == nullptr) return;
   if (root->key == K) return;
   v.push_back(root->val);

   foo(root->left, K, v);
   foo(root->right, K, v);
}

int main(void) {
   int N, K;
   int key;
   string val;

   cin >> N;
   for (int i = 0; i < N; ++i) {
      cin >> key >> val;
      Insert(key, val);
   }
   cin >> K;

   vector<string> v;
   foo(root, K, v); // O(n)

   // min_element è una funzione della libreria standard per
trovare il minimo
   // in un vettore. E' stata usata per comodità, si poteva
implementare
   // facilmente
   if (v.size() >= 1)
      cout << *min_element(v.begin(), v.end()); // O(n)
   else
      cout << "vuoto";

   return 0;
}
```

29/05/2014

Esercizio

Il programma deve leggere una sequenza di N coppie *chiave* e *valore*. Le N chiavi sono interi positivi e distinti per le quali deve essere costruito un albero binario di ricerca NON bilanciato. Per l'inserimento delle coppie nell'albero si deve rispettare il loro ordine nella sequenza.

Al programma viene data in input una chiave intera K che si può assumere essere presente tra le chiavi della sequenza. Sia u il nodo dell'albero avente chiave K. Il programma deve indentificare la stringa lessicograficamente minore tra tutte le stringhe dei nodi che **NON** si trovano nel sottoalbero radicato in u.

L'input è formattato nel seguente modo. La prima riga contiene l'intero N. Seguono poi $2N$ righe, due righe per coppia. La prima riga della coppia contiene la chiave, mentre la seconda contiene il valore. L'ultima riga dell'input contiene il valore K.

L'output è costituito da una singola riga contenente la stringa identificata dal programma. Il programma stampa `vuoto` il nodo u è la radice dell'albero binario di ricerca e, quindi, la stringa minima non esiste.