Arbol binario con celdas enlazadas por punteros

```
typedef int elem_t;
class cell;
class iterator_t;

//--------------------------cell--------------------------
class cell {
    friend class btree;
    friend class iterator_t;
    elem_t t;
    cell *right,*left;
    cell() : right(NULL), left(NULL) {}
};

//--------------------------iterator--------------------------

class iterator_t {
private:
    friend class btree;
    cell *ptr,*father;
    enum side_t {NONE,R,L};
    side_t side;
    iterator_t(cell *p,side_t side_a,cell *f_a) : ptr(p), side(side_a), father(f_a) { }

public:
    iterator_t(const iterator_t &q) {
        ptr = q.ptr;
        side = q.side;
        father = q.father;
    }
    bool operator!=(iterator_t q) { return ptr!=q.ptr; }
    bool operator==(iterator_t q) { return ptr==q.ptr; }
    iterator_t() : ptr(NULL), side(NONE), father(NULL) { }

    iterator_t left() {
        return iterator_t(ptr->left,L,ptr);
    }
iterator_t right() {
        return iterator_t(ptr->right,R,ptr);
    }
};

//--------------------------tree--------------------------

class btree {
private:
    cell *header;
    iterator_t tree_copy_aux(iterator_t nq, btree &TT,iterator_t nt) {
        nq = insert(nq,TT.retrieve(nt));
        iterator_t m = nt.left();
        if (m != TT.end()) tree_copy_aux(nq.left(),TT,m);
        m = nt.right();
        if (m != TT.end()) tree_copy_aux(nq.right(),TT,m);
        return nq;
    }
public:
    static int cell_count_m;
    static int cell_count() { return cell_count_m; }
    btree() {
        header = new cell;
        cell_count_m++;
        header->right = NULL;
        header->left = NULL;
    }
    btree(const btree &TT) {
        if (&TT != this) {
        header = new cell;
        cell_count_m++;
        header->right = NULL;
        header->left = NULL;
        btree &TTT = (btree &) TT;
        if (TTT.begin()!=TTT.end())
            tree_copy_aux(begin(),TTT,TTT.begin());
        }
    }
```

```
    ~btree() { clear(); delete header; cell_count_m--; }
    elem_t & retrieve(iterator_t p) { return p.ptr->t; }
    iterator_t insert(iterator_t p,elem_t t) {
        cell *c = new cell;
        cell_count_m++;
        c->t = t;
        if (p.side == iterator_t::R)
        p.father->right = c;
        else p.father->left = c;
        p.ptr = c;
        return p;
    }
    iterator_t erase(iterator_t p) {
        if(p==end()) return p;
        erase(p.right());
        erase(p.left());
        if (p.side == iterator_t::R)
        p.father->right = NULL;
        else p.father->left = NULL;
        delete p.ptr;
        cell_count_m--;
        p.ptr = NULL;
        return p;
    }

    iterator_t splice(iterator_t to,iterator_t from) {
        cell *c = from.ptr;
        from.ptr = NULL;
        if (from.side == iterator_t::R)
            from.father->right = NULL;
        else
        from.father->left = NULL;
        if (to.side == iterator_t::R) to.father->right = c;
        else to.father->left = c;
        to.ptr = c;
        return to;
    }
    iterator_t find(elem_t t) { return find(t,begin()); }
    iterator_t find(elem_t t,iterator_t p) {
        if(p==end() || p.ptr->t == t) return p;
        iterator_t l = find(t,p.left());
        if (l!=end()) return l;
        iterator_t r = find(t,p.right());
        if (r!=end()) return r;
        return end();
    }
    void clear() { erase(begin()); }
    iterator_t begin() {
        return iterator_t(header->left,
        iterator_t::L,header);
    }
    iterator_t end() { return iterator_t(); }

    void lisp_print(iterator_t n) {
        if (n==end()) { cout << "."; return; }
        iterator_t r = n.right(), l = n.left();
        bool is_leaf = r==end() && l==end();
        if (is_leaf) cout << retrieve(n);
        else {
            cout << "(" << retrieve(n) << " ";
            lisp_print(l);
            cout << " ";
            lisp_print(r);
            cout << ")";
        }
    }
    void lisp_print() { lisp_print(begin()); }
};
```

Implementacion de Conjuntos mediante vectores de bits

```
typedef int iterator_t;

class set {
private:
    vector<bool> v;
    iterator_t next_aux(iterator_t p) {
        while (p<N && !v[p]) p++;
        return p;
    }
    typedef pair<iterator_t,bool> pair_t;
public:
    set() : v(N,0) { }
    set(const set &A) : v(A.v) {}
    ~set() {}
    iterator_t lower_bound(elem_t x) {
        return next_aux(indx(x));
    }
    pair_t insert(elem_t x) {
        iterator_t k = indx(x);
        bool inserted = !v[k];
        v[k] = true;
        return pair_t(k,inserted);
    }
    elem_t retrieve(iterator_t p) { return element(p); }
    void erase(iterator_t p) { v[p]=false; }
    int erase(elem_t x) {
        iterator_t p = indx(x);
        int r = (v[p] ? 1 : 0);
        v[p] = false;
        return r;
    }
    void clear() { for(int j=0; j<N; j++) v[j]=false; }
    iterator_t find(elem_t x) {
        int k = indx(x);
        return (v[k] ? k : N);
    }
    iterator_t begin() { return next_aux(0); }
    iterator_t end() { return N; }
    iterator_t next(iterator_t p) { next_aux(++p); }
    int size() {
        int count=0;
        for (int j=0; j<N; j++) if (v[j]) count++;
        return count;
    }
    friend void set_union(set &A,set &B,set &C);
    friend void set_intersection(set &A,set &B,set &C);
    friend void set_difference(set &A,set &B,set &C);
};

void set_union(set &A,set &B,set &C) {
    for (int j=0; j<N; j++) C.v[j] = A.v[j] || B.v[j];
}
void set_intersection(set &A,set &B,set &C) {
    for (int j=0; j<N; j++) C.v[j] = A.v[j] && B.v[j];
}
void set_difference(set &A,set &B,set &C) {
    for (int j=0; j<N; j++) C.v[j] = A.v[j] && ! B.v[j];
}
```

Diccionario implementado por tablas de dispersion abiertas con listas desordenadas

```
typedef int key_t;

class hash_set;

//-----------------iterator---------------

class iterator_t {
    friend class hash_set;
private:
    int bucket;
    std::list<key_t>::iterator p;
    iterator_t(int b,std::list<key_t>::iterator q)
        : bucket(b), p(q) { }
public:
    bool operator==(iterator_t q) {
        return (bucket == q.bucket && p==q.p);
    }
    bool operator!=(iterator_t q) {
        return !(*this==q);
    }
    iterator_t() { }
};
typedef int (*hash_fun)(key_t x);

//-----------------hash_set---------------

class hash_set {
private:
    typedef std::list<key_t> list_t;
    typedef list_t::iterator listit_t;
    typedef std::pair<iterator_t,bool> pair_t;
    hash_set(const hash_set&) {}
    hash_set& operator=(const hash_set&) {}
    hash_fun h;
    int B;
    int count;
    std::vector<list_t> v;
    iterator_t next_aux(iterator_t p) {
        while (p.p==v[p.bucket].end() && p.bucket<B-1) {
            p.bucket++;
            p.p = v[p.bucket].begin();
        }
        return p;
    }
public:
    hash_set(int B_a,hash_fun h_a) : B(B_a), v(B), h(h_a), count(0) { }
    iterator_t begin() {
        iterator_t p = iterator_t(0,v[0].begin());
        return next_aux(p);
    }
    iterator_t end() {
        return iterator_t(B-1,v[B-1].end());
    }
    iterator_t next(iterator_t p) {
        p.p++; return next_aux(p);
    }
    key_t retrieve(iterator_t p) { return *p.p; }
    pair_t insert(const key_t& x) {
        int b = h(x) % B;
        list_t &L = v[b];
        listit_t p = L.begin();
        while (p!= L.end() && *p!=x) p++;
        if (p!= L.end())
            return pair_t(iterator_t(b,p),false);
        else {
            count++;
            p = L.insert(p,x);
            return pair_t(iterator_t(b,p),true);
        }
    }
    iterator_t find(key_t& x) {
        int b = h(x) % B;
        list_t &L = v[b];
```

```
        listit_t p = L.begin();
        while (p!= L.end() && *p!=x) p++;
        if (p!= L.end())
            return iterator_t(b,p);
        else return end();
    }
    int erase(const key_t& x) {
        list_t &L = v[h(x) % B];
        listit_t p = L.begin();
        while (p!= L.end() && *p!=x) p++;
        if (p!= L.end()) {
            L.erase(p);
            count--;
            return 1;
        } else return 0;
    }
    void erase(iterator_t p) {
        v[p.bucket].erase(p.p);
    }
    void clear() {
        count=0;
        for (int j=0; j<B; j++) v[j].clear();
    }
    int size() { return count; }
};
```

Implementacion de diccionario con tablas de dispersion cerrada y redispersion continua.

```
typedef int iterator_t;
typedef int (*hash_fun)(key_t x);
typedef int (*redisp_fun)(int j);

int linear_redisp_fun(int j) { return j; }

class hash_set {
private:
    hash_set(const hash_set&) {}
    hash_set& operator=(const hash_set&) {}
    int undef, deleted;
    hash_fun h;
    redisp_fun rdf;
    int B;
    int count;
    std::vector<key_t> v;
    std::stack<key_t> S;
    iterator_t locate(key_t x,iterator_t &fdel) {
        int init = h(x);
        int bucket;
        bool not_found = true;
        for (int i=0; i<B; i++) {
            bucket = (init+rdf(i)) % B;
            key_t vb = v[bucket];
            if (vb==x || vb==undef) break;
            if (not_found && vb==deleted) {
                fdel=bucket;
                not_found = false;
            }
        }
        if (not_found) fdel = end();
        return bucket;
    }
    iterator_t next_aux(iterator_t bucket) {
        int j=bucket;
        while(j!=B && (v[j]==undef || v[j]==deleted)) {
            j++;
        }
        return j;
    }
public:
    hash_set(int B_a,hash_fun h_a, key_t undef_a,key_t deleted_a, redisp_fun rdf_a=&linear_redisp_fun) :
B(B_a), undef(undef_a), v(B,undef_a), h(h_a), deleted(deleted_a), rdf(rdf_a), count(0) { }
    std::pair<iterator_t, bool>
    insert(key_t x) {
        iterator_t fdel;
        int bucket = locate(x,fdel);
        if (v[bucket]==x)
            return std::pair<iterator_t,bool>(bucket,false);
        if (fdel!=end()) bucket = fdel;
        if (v[bucket]==undef || v[bucket]==deleted) {
            v[bucket]=x;
            count++;
            return std::pair<iterator_t,bool>(bucket,true);
        } else {
            std::cout << "Tabla de dispersion llena!!\n";
            abort();
        }
    }
```

```
    key_t retrieve(iterator_t p) { return v[p]; }
    iterator_t find(key_t x) {
        iterator_t fdel;
        int bucket = locate(x,fdel);
        if (v[bucket]==x) return bucket;
        else return(end());
    }
    int erase(const key_t& x) {
        iterator_t fdel;
        int bucket = locate(x,fdel);
        if (v[bucket]==x) {
            v[bucket]=deleted;
            count--;
            // Trata de purgar elementos `deleted'
            // Busca el siguiente elemento `undef'
            int j;
            for (j=1; j<B; j++) {
                op_count++;
                int b = (bucket+j) % B;
                key_t vb = v[b];
                if (vb==undef) break;
                S.push(vb);
                v[b]=undef;
                count--;
            }
            v[bucket]=undef;
            // Va haciendo erase/insert de los elementos
            // de atras hacia adelante hasta que se llene
            // `bucket'
            while (!S.empty()) {
                op_count++;
                insert(S.top());
                S.pop();
            }
            return 1;
        } else return 0;
    }
    iterator_t begin() {
        return next_aux(0);
    }
    iterator_t end() { return B; }
    iterator_t next(iterator_t p) {
        return next_aux(p++);
    }
    void clear() {
        count=0;
        for (int j=0; j<B; j++) v[j]=undef;
    }
    int size() { return count; }
};
```

Implementacion de conjuntos con Arboles Binarios de busqueda (ABB)

```
// Forward declarations
template<class T>
class set;
template<class T> void
    set_union(set<T> &A,set<T> &B,set<T> &C);
template<class T> void
    set_intersection(set<T> &A,set<T> &B,set<T> &C);
template<class T> void
    set_difference(set<T> &A,set<T> &B,set<T> &C);

template<class T>
class set {
private:
    typedef btree<T> tree_t;
    typedef typename tree_t::iterator node_t;
    tree_t bstree;
    node_t min(node_t m) {
        if (m == bstree.end()) return bstree.end();
        while (true) {
            node_t n = m.left();
            if (n==bstree.end()) return m;
            m = n;
        }
    }

    void set_union_aux(tree_t &t,node_t n) {
        if (n==t.end()) return;
        else {
            insert(*n);
            set_union_aux(t,n.left());
            set_union_aux(t,n.right());
        }
    }
    void set_intersection_aux(tree_t &t, node_t n, set &B) {
        if (n==t.end()) return;
        else {
            if (B.find(*n)!=B.end()) insert(*n);
            set_intersection_aux(t,n.left(),B);
            set_intersection_aux(t,n.right(),B);
        }
    }
    void set_difference_aux(tree_t &t, node_t n, set &B) {
        if (n==t.end()) return;
        else {
            if (B.find(*n)==B.end()) insert(*n);
            set_difference_aux(t,n.left(),B);
            set_difference_aux(t,n.right(),B);
        }
    }
    int size_aux(tree_t t,node_t n) {
        if (n==t.end()) return 0;
        else return 1+size_aux(t,n.left())
        +size_aux(t,n.right());
    }
public:
    class iterator {
    private:
        friend class set;
        node_t node;
        tree_t *bstree;
        iterator(node_t m,tree_t &t) : node(m), bstree(&t) {}
        node_t next(node_t n) {
            node_t m = n.right();
            if (m!=bstree->end()) {
                while (true) {
                    node_t q = m.left();
                    if (q==bstree->end()) return m;
                    m = q;
                }
            } else {
                // busca el padre
                m = bstree->begin();
                if (n==m) return bstree->end();
                node_t r = bstree->end();
```

```
                while (true) {
                    node_t q;
                    if (*n<*m) { q = m.left(); r=m; }
                    else q = m.right();
                    if (q==n) break;
                    m = q;
                }
                return r;
            }
        }
    public:
        iterator() : bstree(NULL) { }
        iterator(const iterator &n) : node(n.node), bstree(n.bstree) {}
        iterator& operator=(const iterator& n) {
            bstree=n.bstree;
            node = n.node;
        }
        const T &operator*() { return *node; }
        const T *operator->() { return &*node; }
        bool operator!=(iterator q) {
            return node!=q.node; }
        bool operator==(iterator q) { return node==q.node; }
        // Prefix:
        iterator operator++() {
            node = next(node);
            return *this;
        }
        // Postfix:
        iterator operator++(int) {
            node_t q = node;
            node = next(node);
            return iterator(q,*bstree);
        }
    };
private:
    typedef pair<iterator,bool> pair_t;
public:
    set() {}
    set(const set &A) : bstree(A.bstree) {}
    ~set() {}
    pair_t insert(T x) {
        node_t q = find(x).node;
        if (q == bstree.end()) {
            q = bstree.insert(q,x);
            return pair_t(iterator(q,bstree),true);
        } else return pair_t(iterator(q,bstree),false);
    }
    void erase(iterator m) {
        node_t p = m.node;
        node_t qr = p.right(),
            ql = p.left();
        if (qr==bstree.end() && ql==bstree.end())
            p = bstree.erase(p);
        else if (qr == bstree.end()) {
            btree<T> tmp;
            tmp.splice(tmp.begin(),ql);
            p = bstree.erase(p);
            bstree.splice(p,tmp.begin());
        } else if (ql == bstree.end()) {
            btree<T> tmp;
            tmp.splice(tmp.begin(),p.right());
            p = bstree.erase(p);
            bstree.splice(p,tmp.begin());
        } else {
            node_t r = min(qr);
            T minr = *r;
            erase(iterator(r,bstree));
            *p = minr;
        }
    }
    int erase(T x) {
        iterator q = find(x);
        int ret;
        if (q==end()) ret = 0;
        else {
            erase(q);
            ret = 1;
        }
        return ret;
    }
```

```
    void clear() { bstree.clear(); }
    iterator find(T x) {
        node_t m = bstree.begin();
        while (true) {
            if (m == bstree.end())
                return iterator(m,bstree);
            if (x<*m) m = m.left();
            else if (x>*m) m = m.right();
            else return iterator(m,bstree);
        }
    }
    iterator begin() {
        return iterator(min(bstree.begin()),bstree);
    }
    iterator end() {
        return iterator(bstree.end(),bstree);
    }
    int size() {
        return size_aux(bstree,bstree.begin()); }
    friend void
        set_union<T>(set<T> &A,set<T> &B,set<T> &C);
    friend void
        set_intersection<>(set<T> &A,set<T> &B,set<T> &C);
    friend void
        set_difference<>(set<T> &A,set<T> &B,set<T> &C);
    friend void f();
};

//------------------Funciones-----------------

template<class T> void set_union(set<T> &A,set<T> &B,set<T> &C) {
    C.clear();
    C.set_union_aux(A.bstree,A.bstree.begin());
    C.set_union_aux(B.bstree,B.bstree.begin());
}

template<class T> void set_intersection(set<T> &A,set<T> &B,set<T> &C) {
    C.clear();
    C.set_intersection_aux(A.bstree, A.bstree.begin(),B);
}

// C = A - B
template<class T> void set_difference(set<T> &A,set<T> &B,set<T> &C) {
    C.clear();
    C.set_difference_aux(A.bstree,
    A.bstree.begin(),B);
}
```