Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

mercoledì 15 novembre 2017

Questa scheda raccoglie alcune funzioni di base per la manipolazione di alberi binari di ricerca.

Usiamo il tipo `Item` per riferirci in maniera generica agli elementi dell'insieme ordinato su cui facciamo le ricerche. A seconda degli usi, `Item` dovrà essere declinato in maniera diversa, ad esempio come **`int`** o come **`struct`** `occ`.

Gli `Item` appartengono ad un insieme ordinato, e usiamo il tipo `Key` per indicare genericamente la componente di `Item` che usiamo per confrontare elementi e fare le ricerche. Anche `Key` andrà declinato a seconda dei casi (in molti casi potrà coincidere esattamente con `Item`, per esempio nel caso degli interi).

Inoltre prevediamo una funzione `cmp (Key k1, Key k2)` per confrontare le chiavi. Anche questa funzione va declinata a seconda dei casi (ad esempio nel caso delle stringhe si potrà usare `strcmp` da `strings.h`).

Funzione ausiliaria per identificare il padre di un dato nodo

La funzione `bist_searchparent` cerca il nodo con chiave `k` nel sottoalbero di radice `r`, ne memorizza l'indirizzo in `p` e memorizza l'indirizzo del padre in `pf`. Nel caso in cui non esistano nodi con chiave `k`, restituisce -1, `*p` e' `NULL` e `*pf` punta alla foglia alla quale attaccare eventualmente `k`.

```
int bist_searchparent ( Bit_node r, Key k, Bit_node *pf, Bit_node *p ) {

        int res;
        *pf = NULL;
        *p = r;
        if ( !r )
        return -1;

        while ( *p && ( res = cmp( k, key( (*p) -> item ) ) ) != 0 ) {
                *pf = *p;
                *p = res < 0 ? (*p) -> l : (*p) -> r;
        }

        if ( *p == NULL ) /* non ci sono nodi con chiave k */
        return -1;
        return 0;
}
```

Funzioni per cercare una chiave

Versione ricorsiva (usa la funzione ausiliaria `search_parent`):

```
Item bist_search ( Bit_node r, Key k ) {
        Bit_node pf = NULL, p = NULL;
        if ( bist_searchparent ( r, k, &pf, &p ) == 0 )
        return p -> item;
```

```
        else
        return NULLitem;
}
```

Versione iterativa (non usa la funzione ausiliaria `search_parent`):

```
Item bist_search_it( Bit_node p, Key k ) {
if ( p ) {
int res;
while ( p && ( res = cmp( k, key( p -> item ) ) ) != 0 )
p = res < 0 ? p -> l : p -> r;
}
if ( p == NULL ) return NULLitem;
else return p -> item;
}
```

Funzione per inserire un nuovo elemento

```
void bist_insert( Bit_node *r, Item item ) {
        Bit_node qf, q = *r, new = bit_new( item );
        Key k = key(item);

        if ( q == NULL ) {
                /* inserisco nell'albero vuoto */
                *r = new;
                return;
        }

        if ( bist_searchparent ( *r, k, &qf, &q ) == 0 ) {
                /* la chiave c'e' gia' , non inserisco niente */
                printf( "%d c'e' gia' \n" );
                return;
        }

        /* qf e' il padre del nuovo nodo */
        if ( cmp (k, key( qf -> item ) ) < 0 )
        qf -> l = new;
        else
        qf -> r = new;
}
```

Funzione per cancellare un nuovo elemento

```
int bist_delete( Bit_node *r, Key k ) {
Bit_node x, xf, s = NULL;

if ( bist_searchparent ( *r, k, &xf, &x ) == -1 )
/* non ci sono nodi con chiave k, non faccio niente! */
return -1;

/* cerco il nodo s che deve sostituire x */
if ( x -> l == NULL && x -> r == NULL ) /* x non ha figli */
```

```
s = NULL;
else if ( x -> l == NULL || x -> r == NULL )
s = x -> l != NULL ? x -> l : x -> r; /* x ha un solo figlio */
else {
/* x ha due figli; cerco s, il massimo del sottoalbero di sinistra di x */
Bit_node sf = x;
s = x -> l;
while ( s -> r ) {
sf = s;
s = s -> r;
}

/* s non ha figlio destro: avrà  come nuovo figlio destro il figlio destro di x */
s -> r = x -> r;

/* se s e' figlio destro di sf, di sicuro non ha figlio destro */
if ( sf -> r == s ) {
sf -> r = s -> l;
s -> l = x -> l;
}
}

/* sostituisco x con s. Se x e' la radice, diventa la nuova radice */
if ( x == *r ) // x e' la radice
*r = s; // nuova radice
else if ( xf -> l == x) // x e' figlio sinistro
xf -> l = s;
else // x è figlio destro
xf -> r = s;

bit_destroy(x);
return 0;
}
```
