Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

Alberi binari di ricerca

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`).

Funzioni per cercare una chiave

Versione ricorsiva:

```
Bit_node bist_search( Bit_node p, Key k )
{
        if ( !p )
                return NULL;

        int res = cmp( k, key( p -> item );

        if ( res == 0 )
                return p;

        if (res < 0 )
                return search( root->right, key );

        return search( root->left, key );
}
```

Versione iterativa:

```
Item bist_search_it( Bit_node p, Key k ) {
        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 ausiliaria per identificare il padre di un dato nodo

Per inserire o cancellare un nodo, è necessario prima cercare la posizione del nodo nell'albero, e in particolare è comodo individuare la posizione di suo padre. (sarà un nodo con al più un figlio).

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 `parent`. Notate che `parent` e `p` sono indirizzi di puntatore a nodo, e servono per passare "per riferimento" tali puntatori, dal momento che la funzione serve a modificarne il valore. Nel caso in cui non esistano nodi con chiave `k`, restituisce -1, `*p` e' `NULL` e `*parent` punta alla foglia alla quale attaccare eventualmente un nuovo nodo di chiave `k`. Si parte dalla radice e si scende a destra o sinistra a seconda del risultato del confronto tra la chiave `k` e la chiave del nodo corrente.

```
int bist_searchparent ( Bit_node r, Key k, Bit_node *parent, Bit_node *p ) {
        int res;
        *parent = NULL;
        *p = r;

        if ( !r )
                return -1;

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

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

        return 0;
}
```

Funzione per inserire un nuovo elemento

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

        /* se l'albero e' vuoto: */
        if ( q == NULL ) {
                *r = new;
                return;
        }

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

        /* altrimenti inserisco il nuovo nodo come figlio di parent: */
        if ( cmp (k, key( parent -> item ) ) < 0 )
                parent -> l = new;
        else
                parent -> r = new;
}
```

Funzione per cercare una chiave usando `search_parent`

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

Funzione per cancellare l'elemento con una cerca chiave

Quando elimino un nodo, i suoi sottoalberi destro e sinistro restano senza padre. Per mantenere la struttura ad albero, si può sostituire il nodo eliminato con un altro nodo `s`. Ci sono tre possibili casi:

1. : se `x` non ha figli, `x` è sostituito dall'albero vuoto, quindi s vale NULL
2. : se `x` ha un unico figlio, allora `s` è il figlio di `x`
3. : se `x` ha due figli, allora `s` è il massimo tra i nodi minori di `x`, ovvero il nodo che precede `x` nell'ordinamento; questo significa che s è il nodo più a destra del sottoalbero sinistro di `x`.

Una volta determinato `s`, bisogna far diventare figli di `s` quelli che erano figli del nodo eliminato, e sistemare gli eventuali figli di `s`. Inoltre `s` deve essere collegato, come figlio, al padre del nodo eliminato (oppure diventare radice, nel caso in cui lo fosse già `x`).

NB: la radice `*r` può dunque venire modificata, per questo ne passiamo l'indirizzo (in alternativa si potrebbe restituire la nuova radice, modificando il prototipe della funzione).

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

if ( bist_searchparent ( *r, k, &xp, &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 /* x ha un solo figlio */
                s = x -> l != NULL ? x -> l : x -> r;
        else {
                /* x ha due figli;
                cerco s, il massimo del sottoalbero di sinistra di x */
                Bit_node sp = x;
                s = x -> l;
                while ( s -> r ) {
                        sp = 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 sp, allora sp non e' x  */
                if ( sp -> r == s ) {
                        sp -> 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 ( xp -> l == x) // x e' figlio sinistro
                xf -> l = s;
        else // x è figlio destro
                xf -> r = s;

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