Esercizio: Inserzione in albero

```
void insert(Tree t, int k){     // Tree == Nodo *

    // crea il nodo foglia da inserire contenente la chiave
     struct Nodo* e = malloc(sizeof(struct Nodo));
     e->key = k;
     e->left = e->right = NULL;

     struct Nodo* p;
     struct Nodo* x =  t;

    // cerca la posizione della foglia nell'albero
    while (x != NULL){
          p = x;
          if (x->key < k) x = x->right;
          else x = x->left;
     }
    // se l'albero è vuoto imposta e come radice dell'albero
    if (t == NULL)  t = e;          // Ops!
    else{
          if (p->key < k) p->right = e;
          else p->left = e;
```

Esercizio: Inserzione in albero

```
void insert(Tree t, int k){ ....  }

int main() {

     ....
     Tree t= NULL;

     // inserisce tutte le chiavi in t

     for(i=0; i < numchiavi; i++){
          scanf("%d",&k);
          insert(t, k);          // Errore: insert non può modificare t !!
     }

    ....
}
```

Esercizio: Inserzione in albero

```
void insert(Tree t, int k){ ....  }

int main() {

     ....
     Tree t= NULL;

     // inserisce tutte le chiavi in t

     for(i=0; i < numchiavi; i++){
          scanf("%d",&k);
          insert(t, k);          // Errore: insert non può modificare t !!
     }

     ....
}
```

Possibile soluzione:
Usare il passaggio per riferimento
e passare **Tree*** invece di **Tree**

Esercizio: Inserzione in albero (corretto)

```
void insert(Tree *t, int k){     // Tree == Nodo *

    // crea il nodo foglia da inserire contenente la chiave
     struct Nodo* e = malloc(sizeof(struct Nodo));
     e->key = k;
     e->left = e->right = NULL;

     struct Nodo* p;
     struct Nodo* x =  *t;

    // cerca la posizione della foglia nell'albero
    while (x != NULL){
          p = x;
          if (x->key < k) x = x->right;
          else x = x->left;
     }
    // se l'albero è vuoto imposta e come radice dell'albero
    if (*t == NULL)  *t = e;
    else{
          if (p->key < k) p->right = e;
          else p->left = e;
    }
```

Esercizio: Inserzione in albero

```
void insert(Tree *t, int k){    ....  }

int main() {

     ....
     Tree t= NULL;

     // inserisce tutte le chiavi in t

     for(i=0; i < numchiavi; i++){
          scanf("%d",&k);
          insert(&t, k);                    // Adesso funziona
     }

    ....
}
```

Attenzione all'allocazione !

```
void insert(Tree *t, int k){     // Tree == Nodo *

    // crea il nodo foglia da inserire contenente la chiave
     struct Nodo e;   // e verrà deallocato al termine di insert !
     e->key = k;
     e->left = e->right = NULL;

     struct Nodo* p;
     struct Nodo* x =  *t;

    // cerca la posizione della foglia nell'albero
    while (x != NULL){
          p = x;
          if (x->key < k) x = x->right;
          else x = x->left;
     }
    ...

// t ora contiene puntatori non validi !
```

Attenzione all'allocazione !

```
void insert(Tree *t, int k){     // Tree == Nodo *

    // crea il nodo foglia da inserire contenente la chiave
     struct Nodo e;   // e verrà deallocato al termine di insert
     e->key = k;
     e->left = e->right = NU

     struct Nodo* p;
     struct Nodo* x =  *t;

    // cerca la posizione della foglia
    while (x != NULL){
          p = x;
          if (x->key < k) x = x->right;
          else x = x->left;
     }
    ...

// t ora contiene puntatori non validi !
```

Utilizzare *malloc* per rendere gli oggetti creati all'interno di una funzione persistenti

Attenzione all'allocazione !

```
void insert(Lista *l, Nodo *n);

....

Nodo n;                                  // Non funziona:
for ( i = 0; i < N; i++ )
   {                                     // nessun nodo creato
      scanf("%d", &(n.val));             // sovrascrive lo stesso nodo
      inserisci(l, &n);
   }
```

Attenzione all'allocazione !

```
void insert(Lista *l, Nodo *n);

 ....

 Nodo *n;
 for ( i = 0; i < N; i++ )
    {
       n = malloc(sizeof(Nodo));
       scanf("%d", n->val));
       inserisci(l, n);
    }
```

Esercizio 1

Si implementi la funzione:

```
void PruneTree(tree *t, int d)
```

Che dati in input un albero t ed un intero d rimuove da t tutti i nodi aventi meno di d nodi discendenti. La funzione deve liberare con una chiamata a *free* la memoria occupata da ciascun nodo rimosso. Testare la funzione su un input definito a programma.

N.B. Un nodo x si dice discendente del nodo y se x è contenuto nel sottoalbero radicato in y

Esercizio 2

Si scriva un programma che legga da tastiera un intero N ed array *A* di *N* interi distinti e che stampi *SI\n* se *A è* un heap di minimo o *NO\n* altrimenti.

Esercizio 3

Si implementi la funzione:

```
void PruneList(Lista *l, int k)
```

Che dati in input una lista monodirezionale l ed un intero k rimuove da l tutti i nodi contenenti un valore minore di k.
Testare la funzione su un input definito a programma.

Esercizio 4

Si implementi la funzione:

Lista MergeList(Lista *p, Lista *q)

Che dati in input due liste di interi p e q *ordinate*, generi una lista di interi contenente tutti gli elementi di p e q in ordine. La funzione MergeList *può distruggere* le due liste p e q ma *non può allocare nuova memoria* (ad esempio, non è possibile allocare un array e trasferire in esso il contenuto di una lista).Testare la funzione su un input definito a programma.