

# Strutture Dati Dinamiche

Slide Credits Prof Alessandro Campi

# Che cos'è una struttura dati

- Per **struttura dati** si intende comunemente
  - la rappresentazione dei dati e
  - le operazioni consentite su tali dati
- In linguaggi più evoluti del C esistono *librerie* di strutture dati
  - la Standard Template Library in C++
  - le Collection in Java
  - ...

# Strutture dati più comuni

- Lista concatenata (linked list)
- Pila (stack)
- Coda (queue)
- Insieme (set)
- Multi-insieme (multiset o bag)
- Mappa (map)
- Albero (tree)
- Grafo (graph)
- ...

# Lista Concatenata

- Una vecchia conoscenza, ormai!

```
typedef struct Nodo {  
    Tipo dato;  
    struct Nodo *next;  
} nodo;  
typedef nodo * lista;
```

***L'abbiamo già trattata come TDA***

# Pila (o stack)

- È una struttura dati con accesso limitato all'elemento "in cima", che è quello inserito più recentemente (LIFO: last in, first out)
- Nomi standard delle operazioni:
  - **Push**
    - Inserimento in cima
  - **Pop**
    - prelievo e rimozione dell'elemento in cima, che è per definizione l'ultimo elemento inserito
  - **Top o Peek**
    - Prelievo senza rimozione (cioè "sola lettura") dell'elemento in cima

# Applicazioni della pila

- Controllo di bilanciamento di simboli (parentesi, tag XML, blocchi C, ...)
  - Per ogni simbolo di "apertura" si impila un segnaposto che sarà rimosso quando si incontra il simbolo duale di "chiusura"
    - Se il simbolo non corrisponde al segnaposto sulla pila, allora la sequenza non è bilanciata
    - Esempio:  $\{[][\{\}[](){}]()\}\{[()]()\}\{[]\} \rightarrow \text{ok}$   $\{\}\{[]\} () \rightarrow \text{ko}$
- Implementazione di chiamate a funzione
  - Pila di sistema e record di attivazione
- Valutazione di espressioni aritmetiche

# Pila (o stack): implementazione

- PILA (stack):
  - I nuovi nodi possono essere aggiunti e cancellati solo dalla cima (top) della pila
  - La base della pila è indicata da un puntatore a **NULL**
  - È una versione **vincolata** (nelle operazioni) della lista concatenata
- *push*
  - Aggiunge un nuovo nodo alla cima della pila (unica opzione per inserire)
  - Passa la testa per riferimento, quindi non restituisce nulla al chiamante (void)
- *pop*
  - Cancella un nodo dalla cima della pila
  - Memorizza il valore cancellato
  - Passa la testa per riferimento e quindi restituisce un valore booleano al chiamante
    - **true** se l'operazione ha avuto successo

```
typedef struct sNode {
    int data;
    struct sNode * nextPtr;
} StackNode;
typedef StackNode *StackNodePtr;

void push( StackNodePtr *, int );
int pop( StackNodePtr * );
int isEmpty( StackNodePtr );
void printStack( StackNodePtr );
void instructions( void );
```

```
typedef struct sNode {
    int data;
    struct sNode * nextPtr;
} StackNode;
typedef StackNode *StackNodePtr;

void push( StackNodePtr *, int );
int pop( StackNodePtr * );
int isEmpty( StackNodePtr );
void printStack( StackNodePtr );
void instructions( void );
```

```

int main() {
    StackNodePtr stackPtr = NULL; /* punta alla base (= alla cima!) della pila */
    int choice, value;
    do {
        instructions();
        scanf( "%d", &choice );
        switch ( choice ) {
            case 1: printf( "Enter an integer: " ); /* caso push */
                    scanf( "%d", &value );
                    push( &stackPtr, value );
                    printStack( stackPtr );
                    break;
            case 2: if ( !isEmpty( stackPtr ) ) /* caso pop */
                    printf( "The popped value is %d.\n", pop( &stackPtr ) );
                    printStack( stackPtr );
                    break;
            case 3: printf( "End of run.\n" );
                    break;
            default: printf( "Invalid choice.\n\n" );
                    break;
        }
    } while ( choice != 3 );
    return 0;
}

```

```

int main() {
    StackNodePtr stackPtr = NULL; /* punta alla base (= alla cima!) della pila */
    int choice, value;
    do {
        instructions();
        scanf( "%d", &choice );
        switch ( choice ) {
            case 1: printf( "Enter an integer: " ); /* caso push */
                    scanf( "%d", &value );
                    push( &stackPtr, value );
                    printStack( stackPtr );
                    break;
            case 2: if ( !isEmpty( stackPtr ) ) /* caso pop */
                    printf( "The popped value is %d.\n", pop( &stackPtr ) );
                    printStack( stackPtr );
                    break;
            case 3: printf( "End of run.\n" );
                    break;
            default: printf( "Invalid choice.\n\n" );
                    break;
        }
    } while ( choice != 3 );
    return 0;
}

```

Push e pop modificano la lista, **devo passare l'indirizzo del puntatore alla lista** (i.e., l'indirizzo nello stack). Altrimenti restituisco la pila e la sovrascrivo, come visto precedentemente

```
void instructions( void ) {  
    printf( "Enter choice:\n");  
    printf( "1 to push a value on the stack\n");  
    printf( "2 to pop a value off the stack\n");  
    printf( "3 to end program\n\n> ");  
}
```

```
void push( StackNodePtr *topPtr, int info ) {  
    StackNodePtr newPtr;  
    newPtr = malloc( sizeof( StackNode ) );  
    if ( newPtr != NULL ) {  
        newPtr->data = info;  
        newPtr->nextPtr = *topPtr;  
        *topPtr = newPtr;  
    } else  
        printf( "%d not inserted. No memory available.\n", info );  
}
```

```

int pop( StackNodePtr *topPtr ) {
    StackNodePtr tempPtr = *topPtr;
    int popValue = (*topPtr)->data;
    *topPtr = (*topPtr)->nextPtr;
    free( tempPtr );
    return popValue;
}

void printStack( StackNodePtr currentPtr ) {
    if ( currentPtr == NULL )
        printf( "The stack is empty.\n\n" );
    else {
        printf( "The stack is:\n" );
        while ( currentPtr != NULL ) {
            printf( "%d --> ", currentPtr->data );
            currentPtr = currentPtr->nextPtr;
        }
        printf( "NULL\n\n" );
    }
}

int isEmpty( StackNodePtr topPtr ) { return topPtr == NULL; }

```

# Coda (o queue)

- In una coda l'accesso è ristretto all'elemento inserito meno recentemente (FIFO)
- Le operazioni tipiche supportate dalla coda sono:
  - **enqueue** o **offer**
    - aggiunge un elemento in coda
  - **dequeue** o **poll** o **remove**
    - preleva e cancella l'elemento di testa
  - **getFront** o **peek**
    - preleva ma non cancella l'elemento di testa

```
typedef struct qNode {
    int data;
    struct qNode *nextPtr; // definizione identica a lista
} QueueNode;
typedef QueueNode *QueueNodePtr; // puntatore a coda

void printQueue( QueueNodePtr );
int isEmpty( QueueNodePtr );
int dequeue( QueueNodePtr *, QueueNodePtr * );
void enqueue( QueueNodePtr *, QueueNodePtr *, int );
void instructions( void );
```

```

int main() {
    QueueNodePtr firstPtr = NULL, lastPtr = NULL;
    int choice, item;
    do {
        instructions(); scanf( "%d", &choice );
        switch( choice ) {
            case 1: printf( "Enter an integer: " ); scanf( "\n%d", &item );
                    enqueue(&firstPtr, &lastPtr, item ); printQueue( firstPtr );
                    break;
            case 2: if ( !isEmpty( firstPtr ) ) {
                    item = dequeue( &firstPtr, &lastPtr );
                    printf( "%d has been dequeued.\n", item );
                }
                    printQueue( firstPtr );
                    break;
            case 3: printf( "End of run.\n" ); break;
            default: printf( "Invalid choice.\n\n" ); break;
        }
    } while ( choice != 3 );
    return 0;
}

```

Inserimento sempre con doppi puntatori

```

void instructions( void ) {
    printf ( "Enter your choice:\n" );
    printf ( "    1 to add an item to the queue\n" );
    printf ( "    2 to remove an item from the queue\n" );
    printf ( "    3 to end\n\n> " );
}

void enqueue( QueueNodePtr *firstPtr, QueueNodePtr *lastPtr, int value ) {
    QueueNodePtr newPtr;
    newPtr = malloc( sizeof( QueueNode ) ); // se memoria piena restituisce NULL
    if ( newPtr != NULL ) { // prepara il nodo
        newPtr->data = value;
        newPtr->nextPtr = NULL;
        if ( isEmpty( *firstPtr ) ) // se è NULL la testa, la lista è vuota
            *firstPtr = newPtr;
        else ( *lastPtr )->nextPtr = newPtr; // altrimenti accoda in fondo
        *lastPtr = newPtr; // la coda è cambiata e devo aggiornare il puntatore
        nello stack. N.B: non occorre scorrere la lista per trovare la coda
    }
    else printf( "%c not inserted. No memory available.\n", value);
}

int isEmpty( QueueNodePtr firstPtr ) { return firstPtr == NULL; }

```

```

int dequeue( QueueNodePtr *firstPtr, QueueNodePtr *lastPtr ) { // prende val e rimuove la testa
    QueueNodePtr tempPtr = *firstPtr; // tiene traccia della testa
    int value = ( *firstPtr )->data; // prende il valore da restituire
    *firstPtr = ( *firstPtr )->nextPtr; // sposta la testa
    if ( *firstPtr == NULL )
        *lastPtr = NULL; // se la testa punta a NULL, la coda è ora vuota, aggiorna
    free( tempPtr ); // altrimenti crea un dangling pointer.
    return value;
}

```

- Tiene traccia con `tempPtr` della testa  
`tempPtr = *firstPtr;`
- Restituisce il valore della testa  
`*firstPtr = ( *firstPtr )->nextPtr;`
- Sposta la testa al secondo nodo  
`*firstPtr = ( *firstPtr )->nextPtr;`
- Libera la memoria allocata sulla testa precedente  
`free( tempPtr );`

```

int dequeue( QueueNodePtr *firstPtr, QueueNodePtr *lastPtr ) { // prende val e rimuove la testa
    QueueNodePtr tempPtr = *firstPtr; // tiene traccia della testa
    int value = ( *firstPtr )->data; // prende il valore da restituire
    *firstPtr = ( *firstPtr )->nextPtr; // sposta la testa
    if ( *firstPtr == NULL )
        *lastPtr = NULL; // se la testa punta a NULL, la coda è ora vuota, aggiorna
    free( tempPtr ); // altrimenti crea un dangling pointer.
    return value;
}

void printQueue( QueueNodePtr currentPtr ) { // abbastanza normale, scorre da testa
    if ( currentPtr == NULL )
        printf( "Queue is empty.\n\n" );
    else {
        printf( "The queue is:\n" );
        while ( currentPtr != NULL ) {
            printf( "%d --> ", currentPtr->data );
            currentPtr = currentPtr->nextPtr;
        }
        printf( "NULL\n\n" );
    }
}

```

# Insieme (o set)

- È come la lista, ma con il vincolo di non ammettere valori duplicati
- L'ordine in cui appaiono gli elementi nella lista è trasparente all'utente
  - Necessariamente, poiché non è significativo
- Di solito, per motivi di convenienza, gli insiemi si realizzano tramite liste ordinate
  - Così si velocizzano le operazioni di **ricerca** (**dicotomica / binary search**) e **inserimento**

# Prototipi per l'insieme

- `int add ( Tipo item, Insieme * i );`
  - aggiunge l'elemento dato all'insieme e restituisce un valore (booleano) che indica se l'operazione ha avuto successo
- `int remove ( Tipo item, Insieme * i );`
  - rimuove l'elemento indicato dall'insieme e restituisce un valore (booleano) che indica se l'operazione ha avuto successo
- `int contains ( Tipo item, Insieme i );`
  - verifica se l'elemento dato è presente nell'insieme (restituisce un valore booleano)

# Albero binario

- Un albero binario è una struttura dati dinamica in cui i nodi sono connessi tramite “rami” ad altri nodi in modo che:
  - c'è un nodo di partenza (la “radice”)
  - ogni nodo (tranne la radice) è collegato ad uno e un solo nodo “padre”
  - ogni nodo è collegato al massimo a **due** altri nodi, detti “figli” (max due → *binario*)
  - i nodi che non hanno figli sono detti “foglie”

# Struttura di un albero binario

```
typedef struct El {  
    Tipo dato;  
    struct El *left;  
    struct El *right;  
} Nodo;  
typedef Nodo * Tree;
```

# Struttura di un albero binario

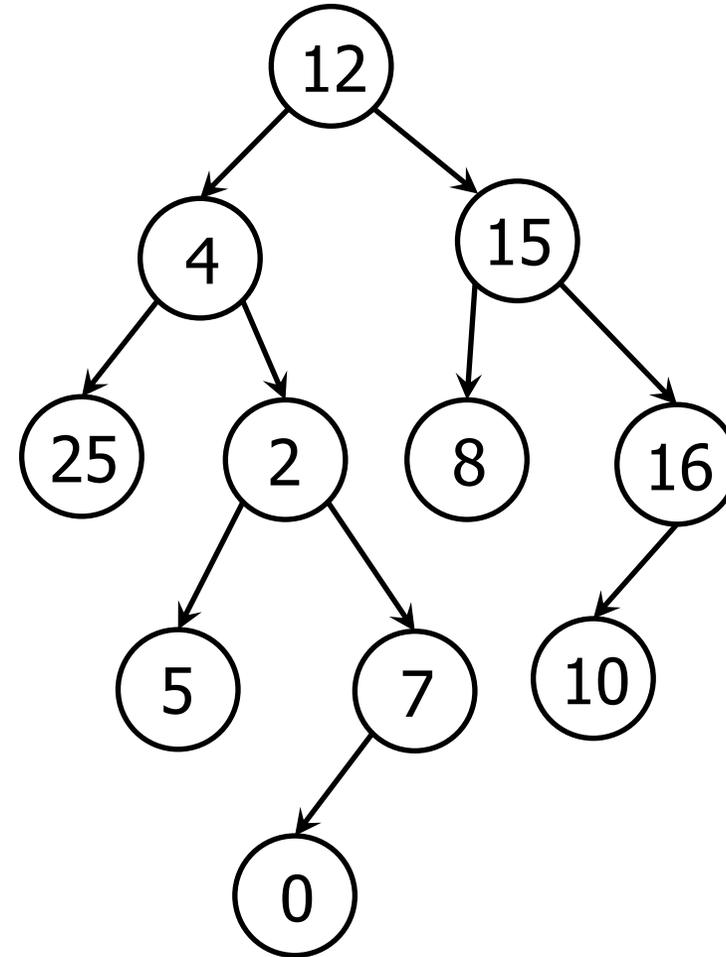
```
typedef struct El {  
    Tipo dato;  
    struct El *left;  
    struct El *right;  
} Nodo;  
typedef Nodo * Tree;
```

***STRUTTURA  
RICORSIVA***

***Un albero è un nodo da cui  
"spuntano" due... alberi!  
(l'albero destro e l'albero sinistro)***

# Struttura di un albero binario

```
typedef struct El {  
    int dato;  
    struct El *left;  
    struct El *right;  
} Nodo;  
typedef Nodo * Tree;
```



# Piccoli esercizi su alberi binari

**NATURALMENTE** in versione ricorsiva!

Per capire quanto sia più semplice formulare questi problemi e le relative soluzioni in forma ricorsiva è sufficiente ... **provare** a fare **diversamente!**

Esercizi "facili":

Conteggio dei nodi

Calcolo della profondità

Ricerca di un elemento

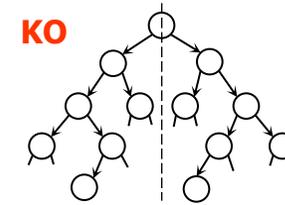
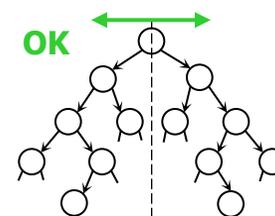
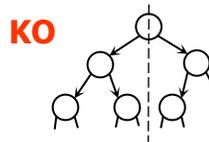
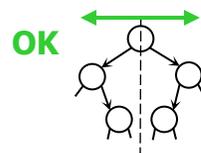
Conteggio dei nodi foglia

Conteggio dei nodi non-foglia

Conteggio dei nodi su livello pari/dispari

Un esercizio difficile:

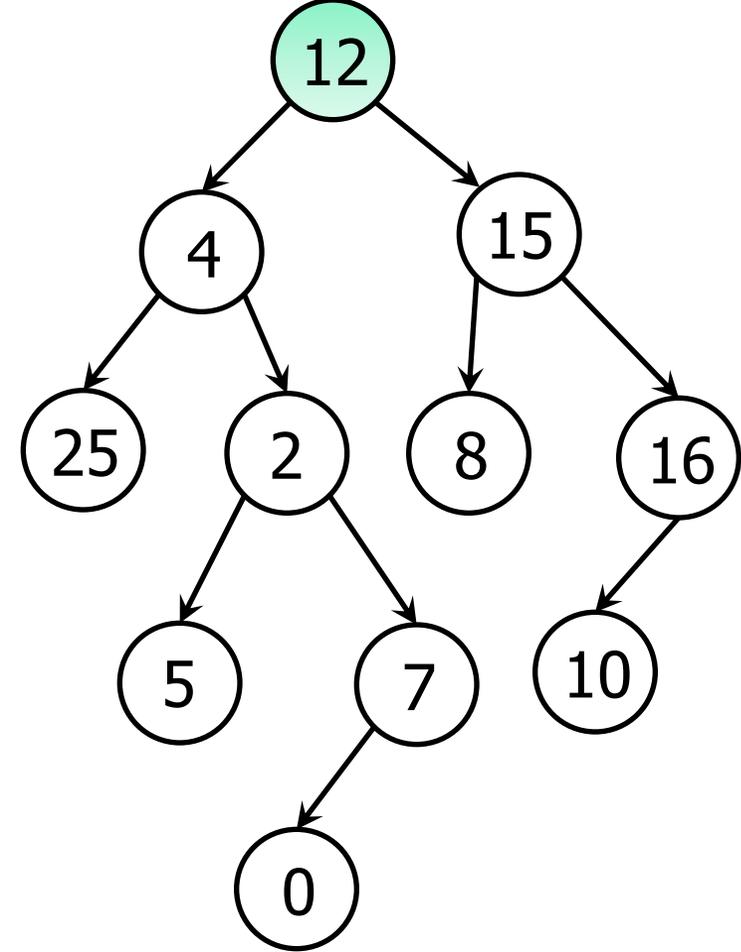
Verifica di "simmetria speculare" dell'albero:



# Conteggio dei nodi

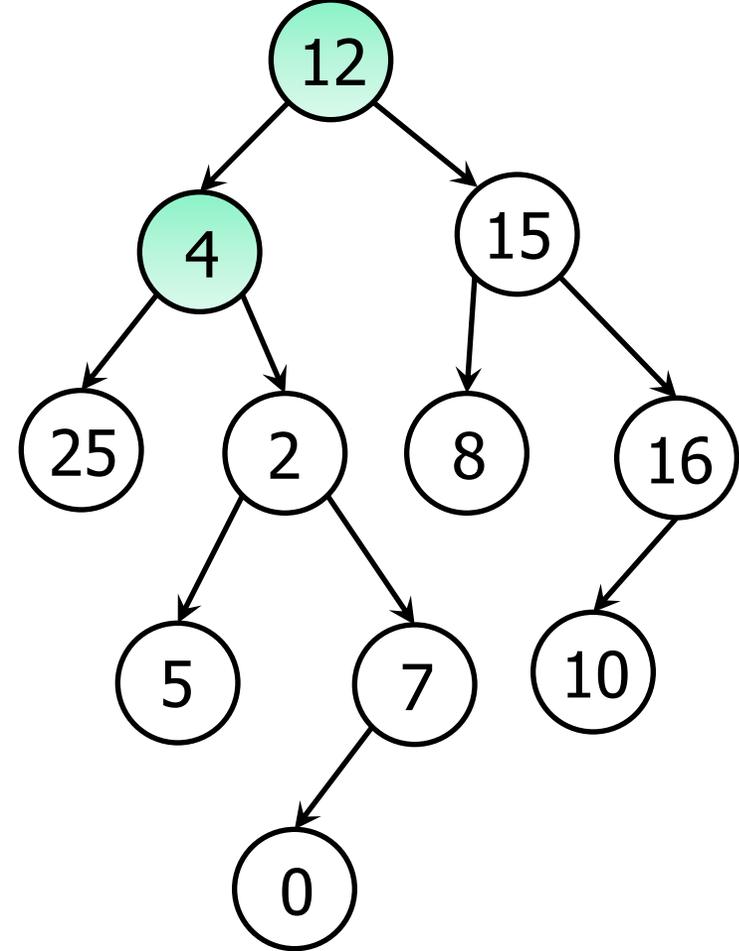
```
int contaNodi ( Tree t ) {  
    if ( t == NULL )  
        return 0;  
    else  
        return ( contaNodi(t->left) +  
                contaNodi(t->right)  
                + 1 ); /* c'è anche il nodo corrente */  
}
```

```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```



```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

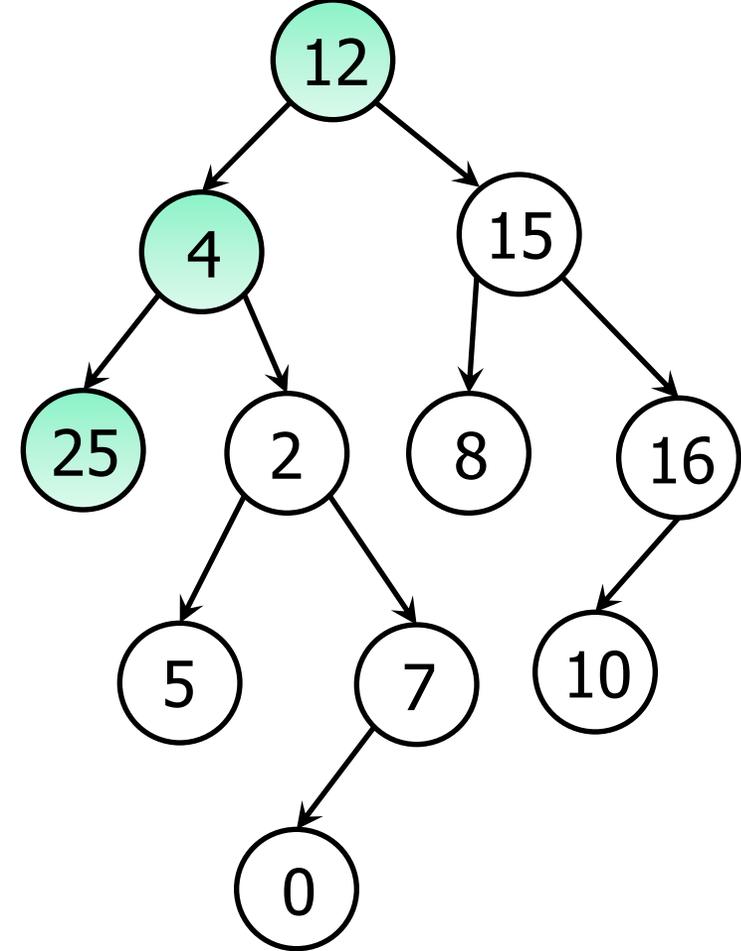
```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```



```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

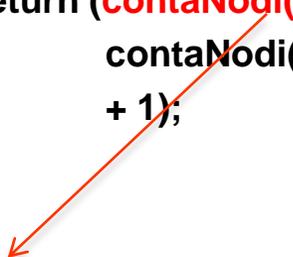
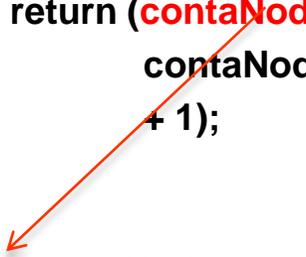
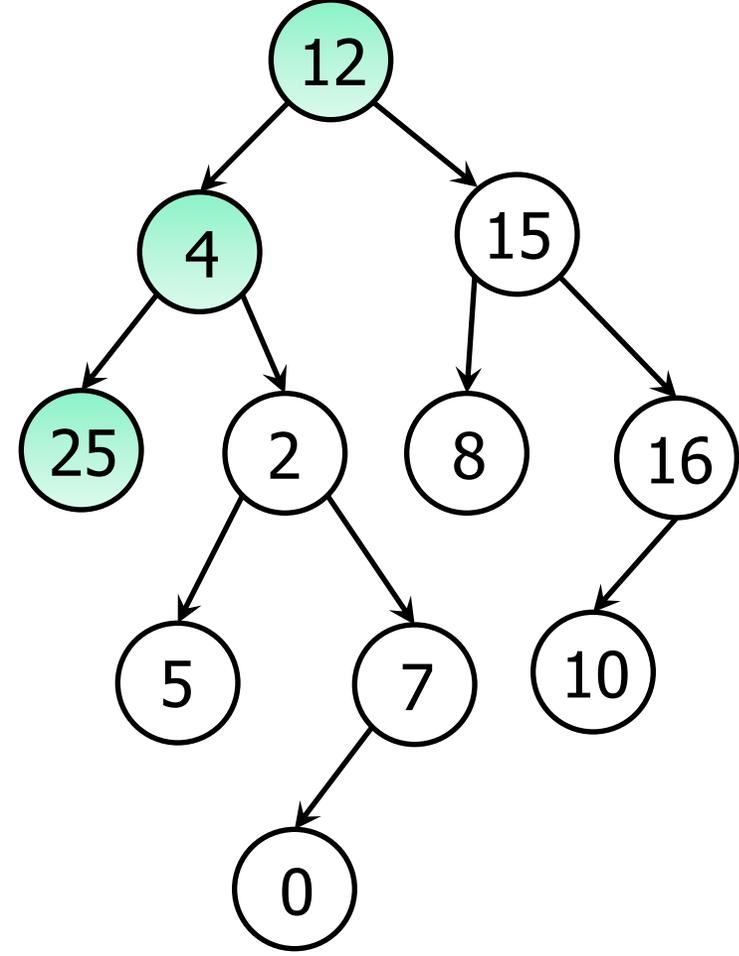


```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

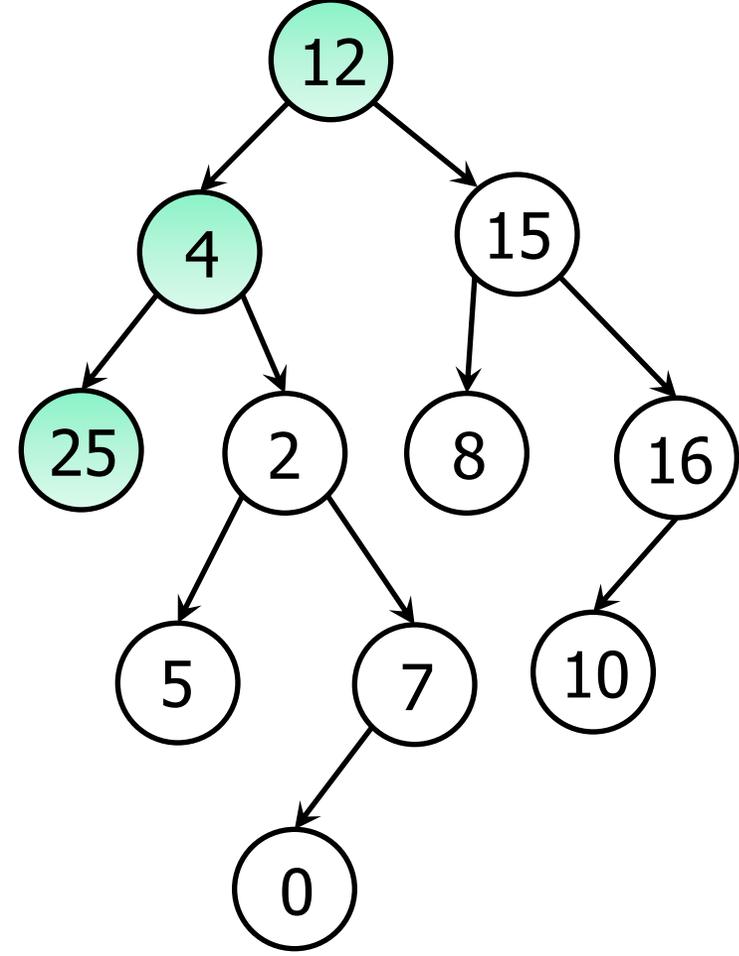


```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```

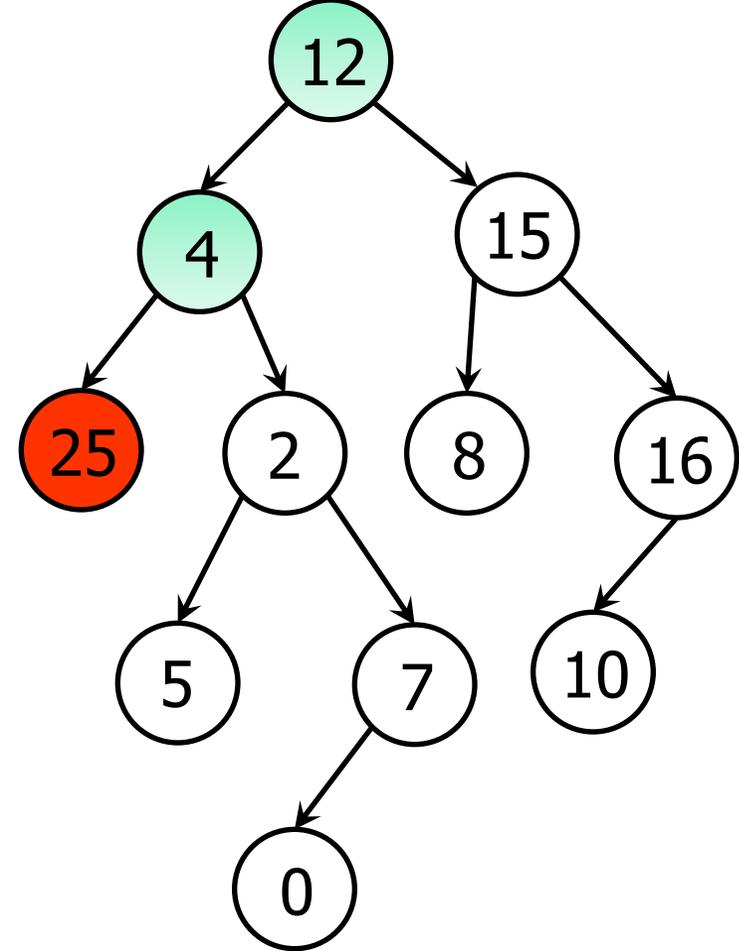
```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```



```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return ( contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```



```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```

```

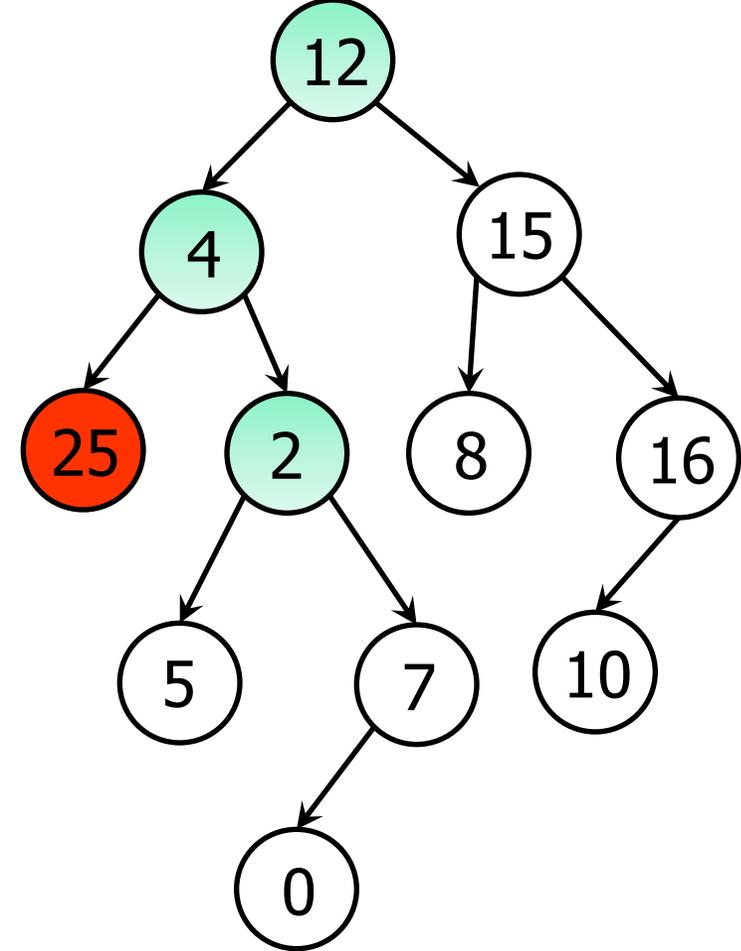
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             contaNodi(t->right)
             + 1);
}

```

```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```

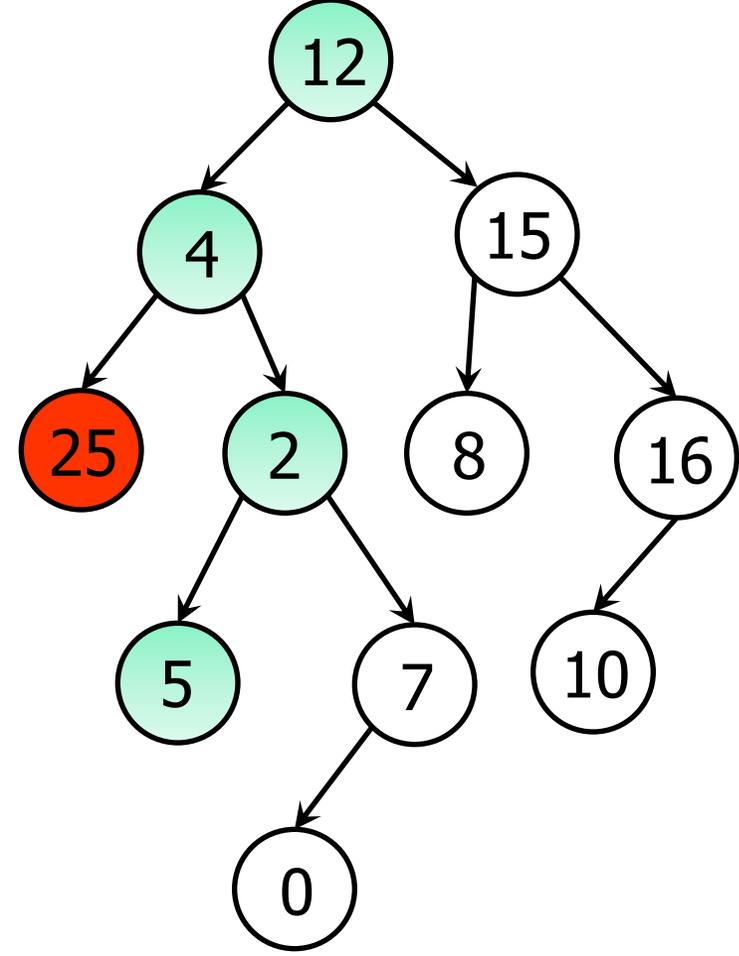


```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            1
            + contaNodi(t->right)
            + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return (contaNodi(t->left) +
            contaNodi(t->right)
            + 1);
}
```



```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```

```

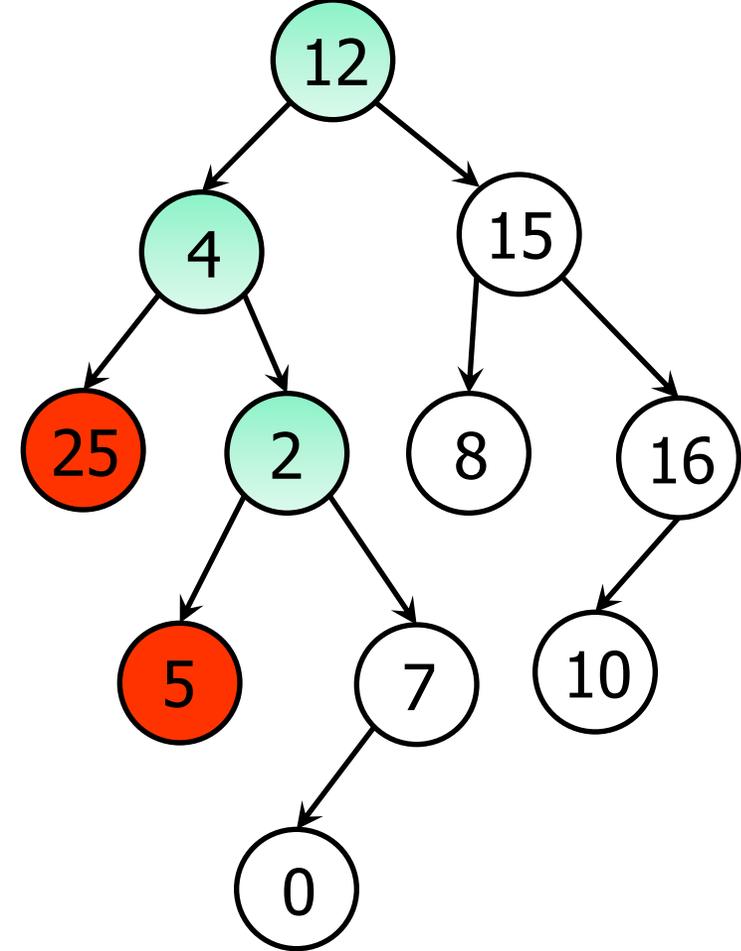
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             contaNodi(t->right)
             + 1);
}

```

```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( 1
             contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```

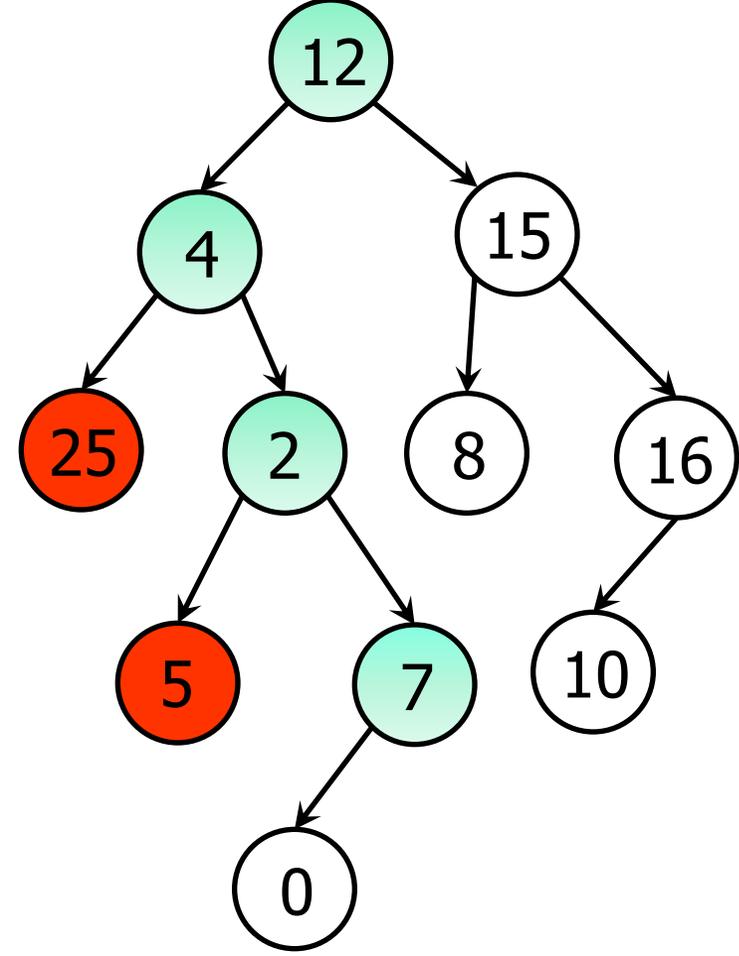


```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

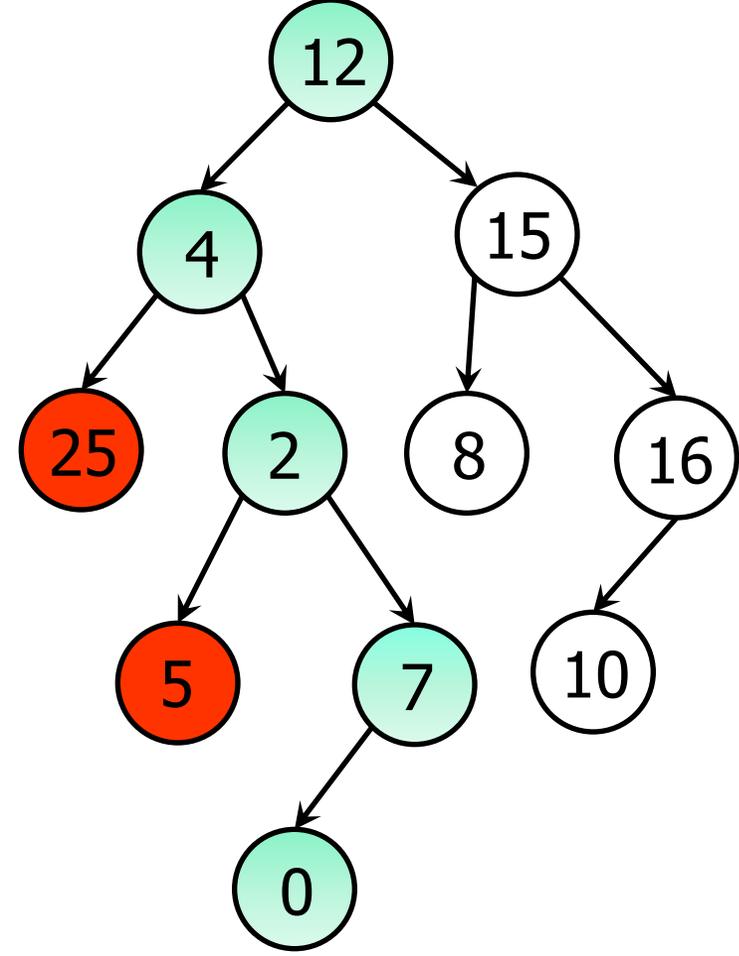


```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```



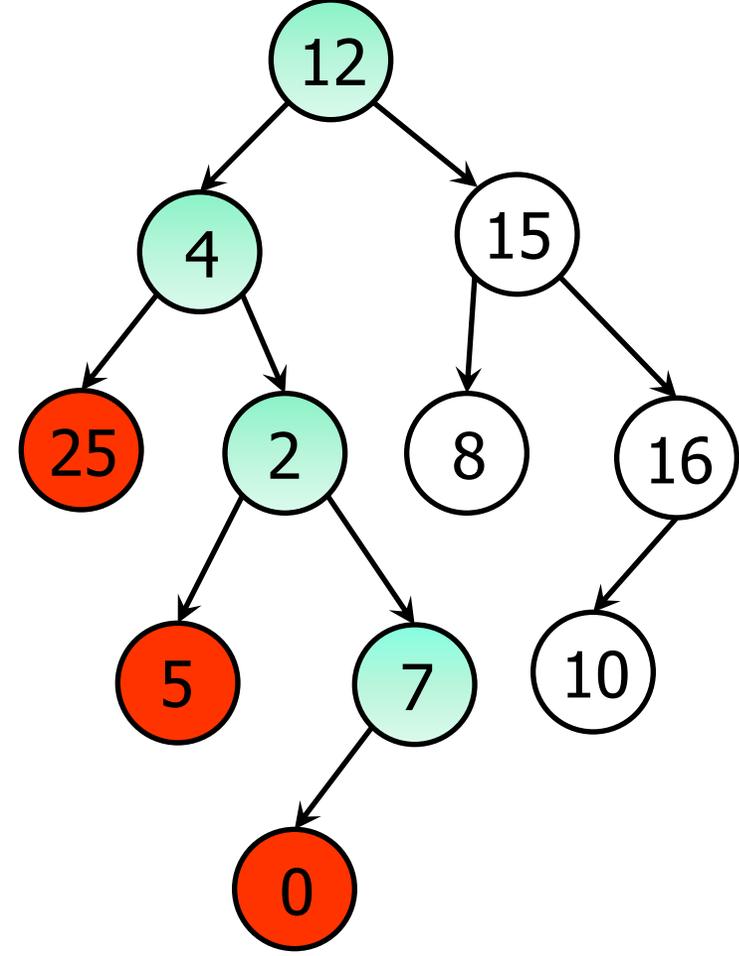
```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             + 1);
}
```

```
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             + 1);
}
```



```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```

```

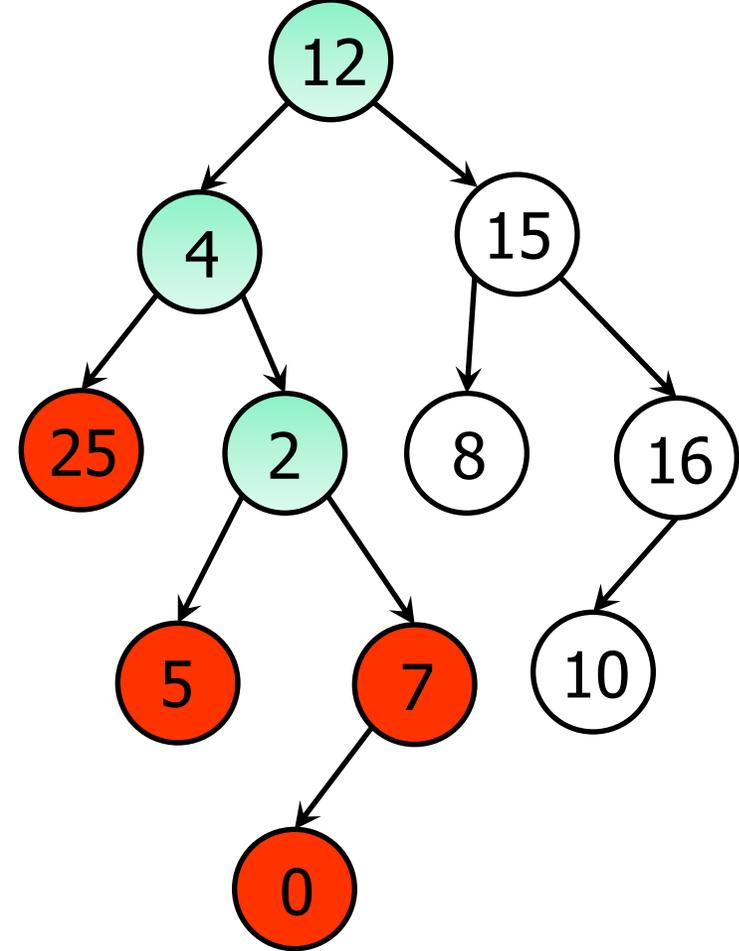
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             contaNodi(t->right)
             + 1);
}

```

```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             1
             contaNodi(t->right)
             + 1);
}

```



```

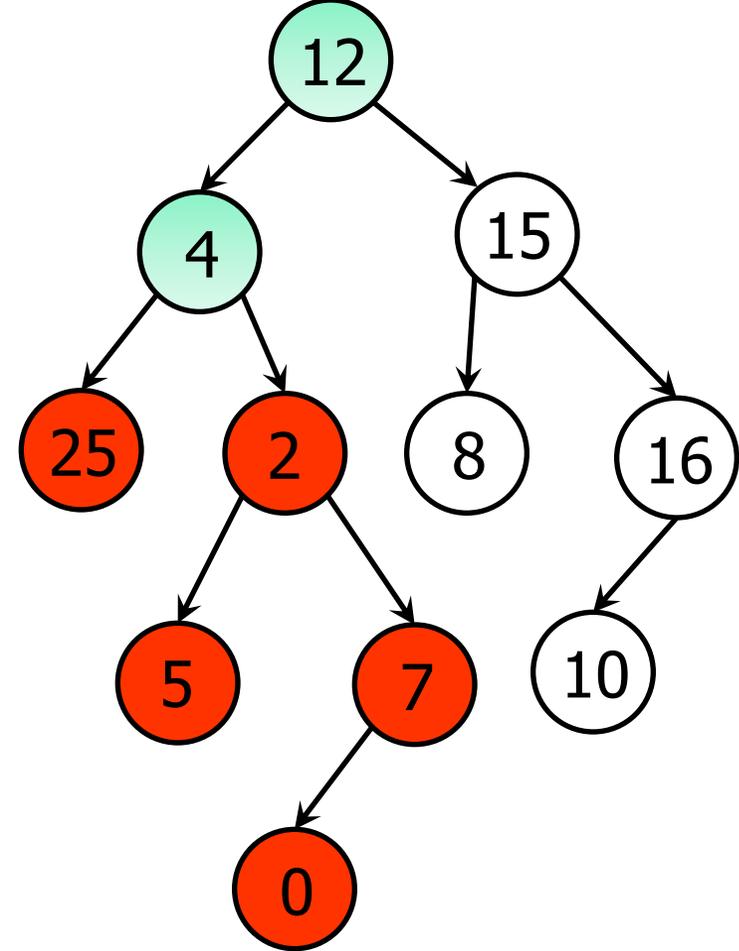
int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```

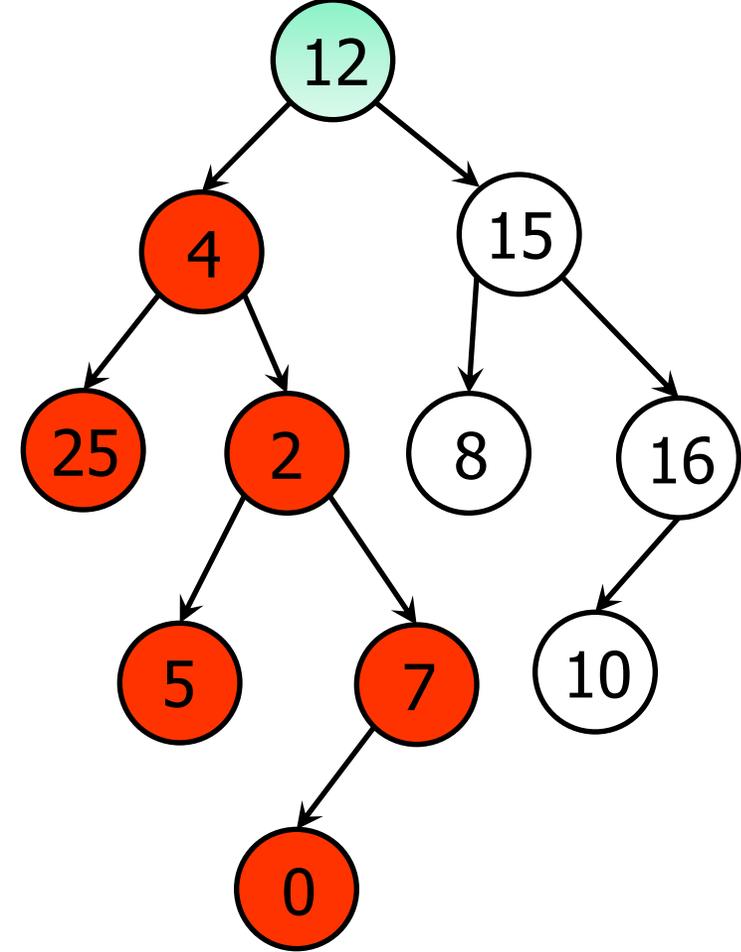
```

int contaNodi ( tree t ) {
  if ( t == NULL )
    return 0;
  else
    return ( contaNodi(t->left) +
             contaNodi(t->right)
             + 1);
}

```



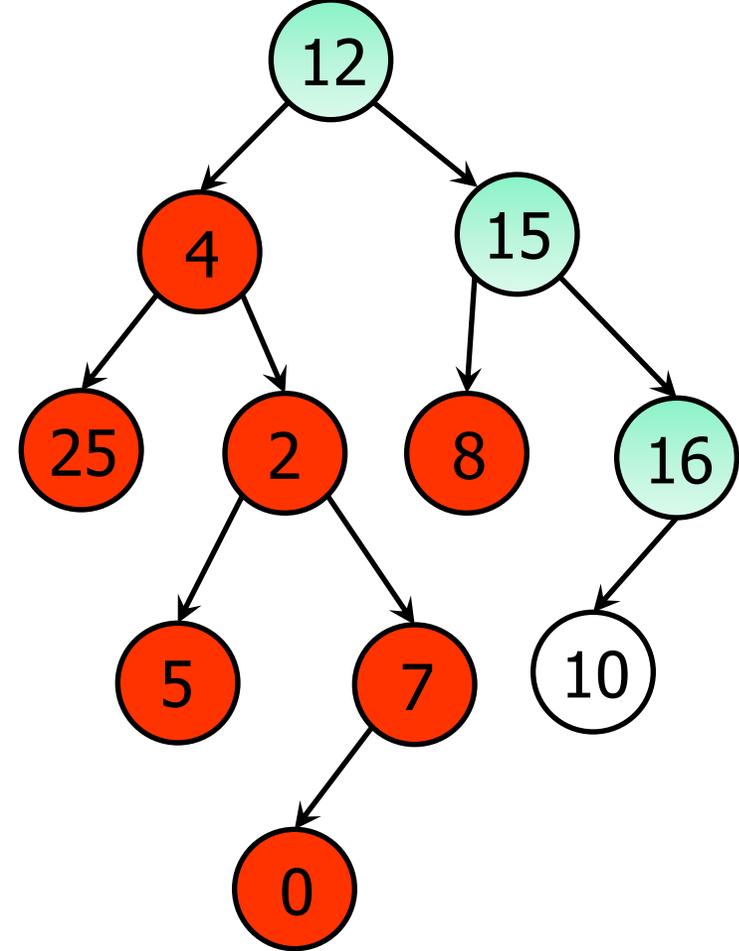
```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```



```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

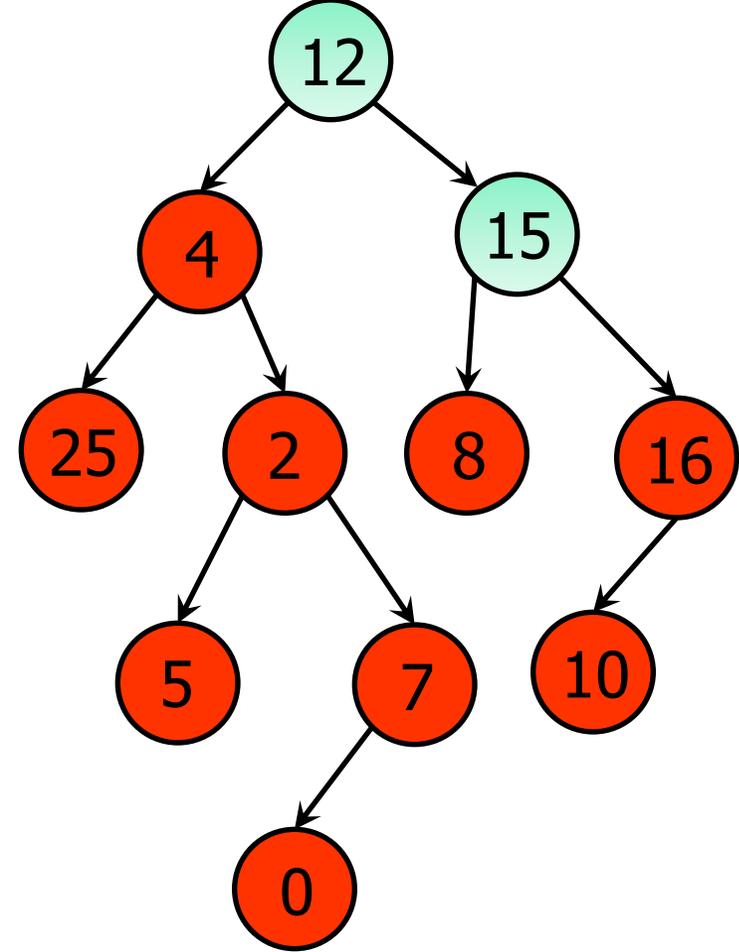
```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

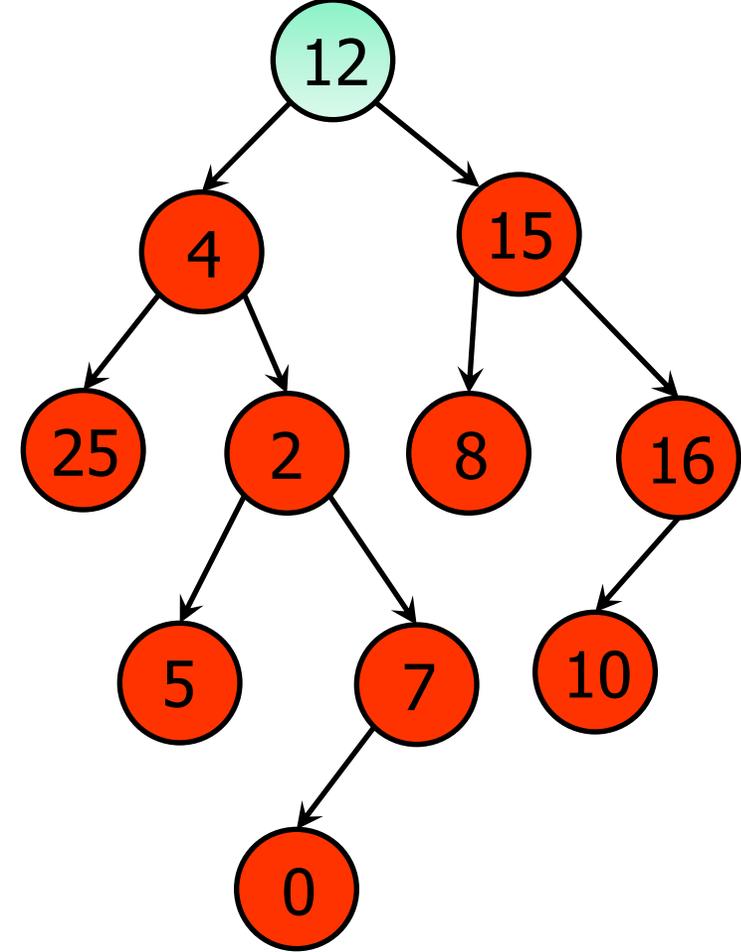


```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```

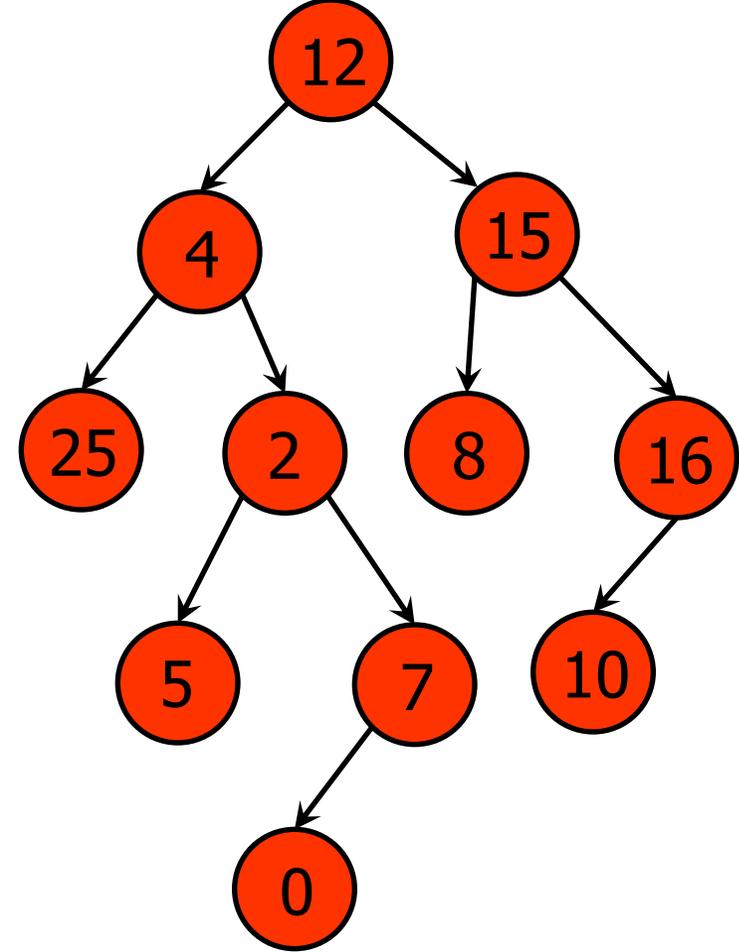
```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```



```
int contaNodi ( tree t ) {  
  if ( t == NULL )  
    return 0;  
  else  
    return (contaNodi(t->left) +  
            contaNodi(t->right)  
            + 1);  
}
```



# Restituisce 11



# Calcolo della profondità massima

```
int depth ( Tree t ) {  
    int D, S;  
    if (t == NULL)  
        return 0;  
    S = depth( t->left );  
    D = depth( t->right );  
    if ( S > D )  
        return S + 1;  
    else  
        return D + 1;  
}
```

# Calcolo della profondità

```
int max(int a,int b) {  
    if(a>b)  
        return a;  
    else  
        return b;  
}
```

```
int depth ( Tree t ) {  
    if (t == NULL)  
        return 0;  
    return max(depth( t->left ),depth( t->right ))+1;  
}
```

# Ancora la profondità (variante)

```
int depth (Tree t, int currentDepth) {  
    int D, S;  
    if ( t == NULL )  
        return currentDepth;  
    else {  
        S = depth( t->left, currentDepth+1 );  
        D = depth( t->right, currentDepth+1 );  
        if ( S > D )  
            return S;  
        else  
            return D;  
    }  
}
```

Questa versione utilizza il concetto "sussidiario" di *livello di profondità del nodo corrente*, che è incrementato di una unità ad ogni chiamata che scende "più in profondità"

Va invocato nel main passando 0 come secondo parametron, i.e., `d = depth(pt, 0);`

# Ricerca di un elemento in un albero

Restituisce NULL se non trova nell'albero t il dato d, altrimenti restituisce il puntatore al primo nodo che lo contiene, effettuando la visita di t in preordine sinistro

```
Tree trova ( Tree t, Tipo d) {  
    Tree temp;  
    if ( t == NULL)  
        return NULL  
    if ( t->dato == d ) /* Se Tipo ammette l'operatore == */  
        return t;  
    temp = trova( t->left, d );  
    if ( temp == NULL )  
        return trova( t->right, d );  
    else  
        return temp;  
}
```

## Ricerca di un elemento in un albero ordinato

Restituisce NULL se non trova nell'albero t il dato d, altrimenti restituisce il puntatore al primo nodo che lo contiene, effettuando la visita di t in preordine sinistro

```
Tree trova ( Tree t, Tipo d) {  
    Tree temp;  
    if ( t == NULL)  
        return NULL  
    if ( t->dato == d ) /* Se Tipo ammette l'operatore == */  
        return t;  
    if(t->dato > d)  
        return trova( t->left, d );  
    else  
        return trova( t->right, d );  
}
```

# Ricerca di un elemento in un albero

Restituisce NULL se non trova nell'albero t il dato d, altrimenti restituisce il puntatore al primo nodo che lo contiene, effettuando la visita di t in preordine sinistro

```
tree trova ( tree t, Tipo d) {  
    tree temp,temp2;  
    if ( t == NULL)  
        return NULL  
    if ( t->dato == d ) /* Se Tipo ammette l'operatore == */  
        return t;  
    temp = trova( t->left, d );  
    temp2 = trova( t->right, d );  
    if(temp2!=NULL)  
        return temp2;  
    else  
        return temp;  
}
```

# Numero di nodi foglia

```
int leaves ( Tree t ) {  
    if ( t == NULL )  
        return 0;  
    if ( t->left == NULL && t->right == NULL);  
        return 1;  
    return leaves( t->right ) + leaves(t->left);  
}
```

# Numero di nodi foglia

```
int leaves ( Tree t ) {  
    if ( t->left == NULL && t->right == NULL);  
        return 1;  
    return leaves( t->right ) + leaves(t->left);  
}
```

**NOOOOOOOOOOOOOOOOOO,  
È indispensabile tenere  
if ( t == NULL )  
 return 0;**

Darebbe errore nei nodi con un solo figlio, tipo t->left NULL e t->right != NULL  
Perché la condizione è falsa e quindi si passa a valutare t->left e qui (bisognerebbe controllare se == NULL come prima cosa), da errore quando prova ad accedere a t->right o t->left

# Numero di nodi non-foglia

```
int branches ( Tree t ) {  
    if ( t == NULL ||  
        ( t->left == NULL && t->right == NULL) )  
        return 0;  
    return 1 + branches( t->right ) + branches(t->left);  
}
```

*oppure...*

```
int branches2 ( Tree t ) {  
    return contaNodi( t ) – leaves( t );  
}
```

Si noti che la seconda versione scandisce due volte l'intero albero, impiegando circa il doppio del tempo

# Numero di nodi su livelli dispari

```
int oddCount ( tree t, int level ) {  
    int n;  
    if ( t == NULL )  
        return 0;  
    n = oddCount( t->left, level+1 ) + oddCount( t->right, level+1 );  
    if ( level%2 == 1 ) /* == 0 → per la funzione evenCount */  
        n++;  
    return n;  
}
```

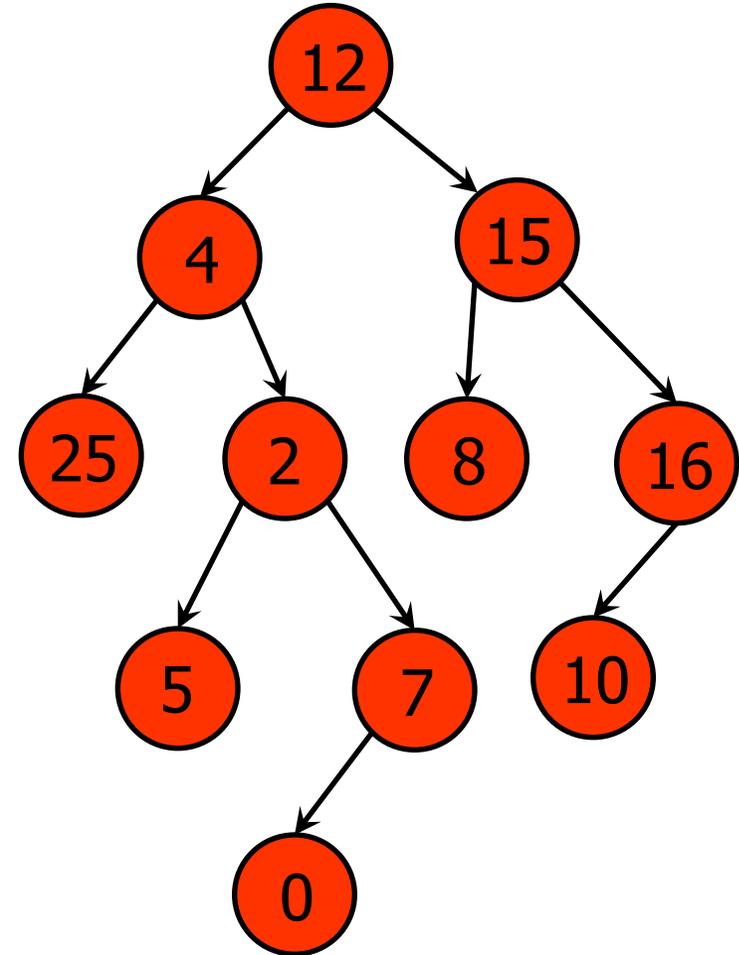
Questa formulazione richiede che la prima chiamata abbia la forma:

```
tree t = costruisciAlbero( ... );  
int x = oddCount( t, 0 ); /* Considerando pari il livello della radice */
```

Per la funzione evenCount (nodi su livelli pari) si ragiona in modo perfettamente simmetrico, o si fa ancora contaNodi – oddCount

# Numero di nodi su livelli dispari

```
int oddCount ( tree t, int level ) {  
    int n;  
    if ( t == NULL )  
        return 0;  
    n = oddCount( t->left, level+1 ) +  
        oddCount( t->right, level+1 );  
    if ( level%2 == 1 )  
        n++;  
    return n;  
}
```



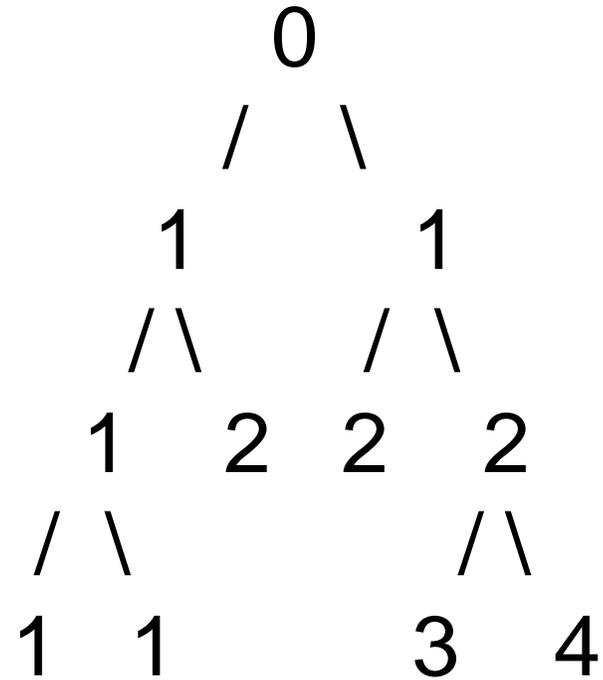
# Stampa Albero

```
void print(Tree t){
    if(t==NULL)
        return;
    else{printf(" (");
        print(t->left);
        printf(" %d ",t->val);
        print(t->right);
        printf(") ");
    }
}

void stampa(Tree T){
    print(T);
    printf("\n");}
```

# Stampa Albero

Esempio di albero



che sarà stampato dal programma così:

```
((((1) 1 (1)) 1 (2)) 0 ((2) 1 ((3) 2 (4))))
```

# Popola Albero

Scrivere una funzione che dato un albero di ricerca (i.e. ordinato) inserisca un nuovo elemento mantenendo l'albero ordinato.

```
Tree insert(Tree t, int x){
    Tree tmp;

    if(t==NULL){
        tmp=(Tree)malloc(sizeof(Nodo));
        tmp->v=x;
        tmp->left=NULL;
        tmp->right=NULL;
        return tmp;
    }
    if(x < t->v){
        t->left=insert(t->left,x);
    }else if(x > t->v){
        t->right=insert(t->right,x);
    }
    return t;}

```

```
int main(){
    Tree t=NULL;

    t=insert(t,12);
    t=insert(t,1);
    t=insert(t,12);
    t=insert(t,20);
    t=insert(t,21);
    t=insert(t,15);

    printf("somma: %d, max val: %d\n",
    somma(t), cercaMax(t));

    return 0;
}

```

# Numero di nodi su livelli pari

Consideriamo che il livello della radice sia pari.

Possiamo scrivere una versione di `evenCount` che non necessita di propagare il concetto di livello "in avanti" nelle chiamate ricorsive.

```
int evenCount ( tree t ) {  
    int n = 1; /* Il nodo corrente è sempre su livello pari, per ipotesi induttiva*/  
    if ( t == NULL )  
        return 0;  
    if ( t->left != NULL )  
        n += evenCount( t->left->left ) + evenCount( t->left->right );  
    if ( t->right != NULL )  
        n += evenCount( t->right->left ) + evenCount( t->right->right );  
    return n;  
}
```

La prima chiamata sarà allora:

```
tree t = costruisciAlbero( ?filename? );  
int x = evenCount( t );
```

# Cancellare un albero

```
void cancella(tree *t) {  
    if (*t!=NULL) {  
        if ((*t)->left != NULL)  
            cancella(&((*t)->left));  
        if ((*t)->right != NULL)  
            cancella(&((*t)->right));  
        free (*t);  
        *t=NULL;  
    }  
}
```

# Componibilità delle strutture dati

- Alberi di code
- Liste di liste
- Alberi di liste di code
- ...
- Il “dato” contenuto nei nodi della struttura contenente è, in questi casi, una istanza di struttura dinamica del tipo che vogliamo sia il “contenuto”
  - del resto abbiamo sempre detto che il **Tipo** del contenuto dei nodi è assolutamente “libero”

# Albero n-ario

- È una generalizzazione dell'albero binario
  - ogni nodo ha un numero arbitrario di figli
- Può anche essere pensato come un grafo
  - Connesso
    - Cioè esiste un cammino di archi tra qualunque nodo e qualunque altro nodo nel grafo
  - Aciclico
  - Tale per cui
    - Ogni nodo (tranne uno, detto radice) ha un solo arco entrante
- Si usa ad esempio per rappresentare tassonomie e organizzazioni gerarchiche

# Grafo

- Un grafo è un insieme di vertici (o nodi) collegati da lati (o archi)
- Può essere rappresentato, ad esempio, come la coppia dei due insiemi seguenti:
  - un insieme di vertici
  - un insieme di coppie di vertici (gli archi)