Albero binario

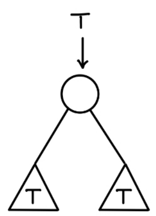
Un albero binario è una struttura dati costituita da un numero variabile n di nodi, in cui ogni nodo è costituito da un campo key, che contiene un dato, e due puntatori sx e dx, che puntano ai nodi successivi, detti ***figli***.

- ***Radice***: primo nodo dell'albero (non ha nessun padre).
- ***Foglie***: ultimi nodi dell'albero, hanno entrambi i puntatori a $NULL$.
- ***Nodi interni***: tutti i nodi non foglia.

Definiamo l'***altezza*** h di un albero come il numero di nodi da attraversare per arrivare alla fine di un albero. In generale l'altezza di un albero è: $\log_2 n \ \leq h \ \leq n$, dove n è il numero di nodi.

Definizione induttiva

Un albero T è definito in modo induttivo come:
Caso base: $T = \emptyset$
Caso induttivo: $T \neq \emptyset$



Albero binario pieno

Un albero binario **pieno** è un albero binario in cui tutti i nodi, tranne le foglie, hanno esattamente due figli.

Livello	n nodi per livello
0	1
1	2
2	4
3	8
i	2^i

Numero di nodi di un albero pieno

Possiamo quindi calcolare il numero totale di nodi di un albero pieno:

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Altezza di un albero pieno

Possiamo calcolare l'altezza di un albero pieno partendo dal numero di nodi: $n = 2^{h+1} - 1$;

Sicuramente $2^h \leq \ 2^{h+1} - 1 < 2^{h+1}$ quindi applico $\log_2$ ed ottengo:
$\log_2 2^h \ \leq \ \log_2(2^{h+1} - 1) \ < \ \log_2(2^{h+1})$ da cui : $h \ \leq \ \log_2(2^{h+1} - 1) < h + 1$.

Quindi $h = \lfloor \log_2(2^{h+1} - 1) \rfloor = \lfloor \log_2 n \rfloor$.

Visita di un albero binario

Un albero binario può essere visitato in due modi:

- Visita in ampiezza **BFS** (Breadth First Search): l'albero viene visitato "orizzontalmente", cioè per livelli, quindi si visitano prima tutti i nodi a livello 0, poi quelli a livello 1 e così via.
- Visita in profondità **DFS**(Depth First Search): l'albero viene visitato " verticalmente", cioè si visitano prima tutti i nodi lungo un percorso fino ad una foglia, poi un altro percorso, fino alla fine dell'albero.

> **Important**
>
> Il tempo di esecuzione di un algoritmo che visita un albero è lineare sul numero di nodi.

Visita in profondità

Caso base: $T = \emptyset$, termino la visita;
Caso induttivo: $T \neq \emptyset$, visito i sottoalberi.

La DFS su un albero binario può essere eseguita in 3 ordini diversi:

- **Preorder**: visito prima la radice e poi i sottoalberi sinistro e destro.
- **Inorder**: visito prima il sottoalbero sinistro, poi la radice e poi il sottoalbero destro.
- **Postorder**: visito prima i sottoalberi sinistro e destro e poi la radice.

```
def Preorder(T):
        if T != NULL:
                Visita(T->key)
                Preorder(T->sx)
                Preorder(T->dx)
```

```
def Inorder(T):
        if T != NULL:
                Inorder(T->sx)
                Visita(T->key)
                Inorder(T->dx)
```

```
def Postorder(T):
        if T != NULL:
                Postorder(T->sx)
                Postorder(T->dx)
                Visita(T->key)
```

> **Important**
>
> La memoria massima occupata dallo stack di un algoritmo ricorsivo è uguale all'altezza h.
> Visitare un albero richiede sempre memoria (stack).

Visita in ampiezza

Per effettuare una BFS su un albero binario abbiamo bisogno di una coda Q in cui inserire i nodi del livello da visitare.

```
def BFS(T):
        Q = Enqueue(Q,T)   # metto in coda la radice
        while (Q != NULL):
                x = Testa(Q)   # salvo la testa della coda in x
                Visita(x)      # visito il nodo in testa alla coda
                Q = Enqueue(Q, x->sx)  # metto in coda il nodo a sx
                Q = Enqueue(Q, x->dx)  # metto in coda il nodo a dx
                Q = Dequeue(Q)         # rimuovo la testa della coda
```

Ricerca in un albero binario

Per cercare un elemento k in un albero binario si può utilizzare il seguente algoritmo, basato su una DFS Inorder:

```
def Search(T, k):
        ret = T
        if T != NULL:
                if T->key != k:   # controllo la chiave in radice
```

```
                          ret = Search(T->sx, k)  # controllo il sottoalbero
sx
                                  if ret = NULL:
                                          ret = Search(T->dx, k)  #
controllo il sottoalbero dx
        return ret
```

- L'algoritmo termina se l'albero è vuoto o se l'elemento è stato trovato.