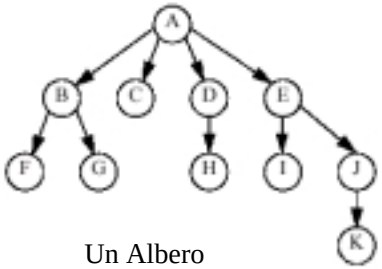
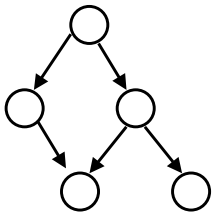
ALBERI

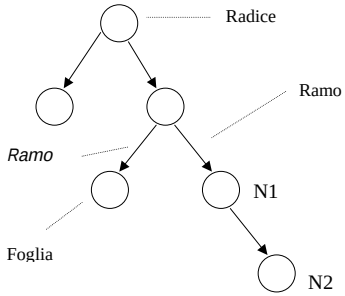
Un **Albero** è un grafo orientato *connesso* e *aciclico*, in cui inoltre esattamente un nodo (detto *radice*) non ha nessun arco entrante, e ogni altro nodo ha esattamente un arco entrante.
Ogni nodo viene di norma utilizzato per contenere opportune informazioni: un albero è una struttura dati atta a rappresentare un insieme di informazioni strutturate in maniera gerarchica.



Un Albero



Un grafo aciclico che non è un albero: due archi entranti in uno stesso nodo



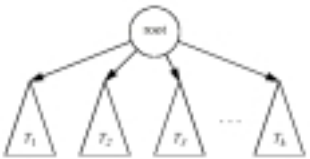
<u>Terminologia ad hoc per gli alberi:</u>

- Un albero è formato da:
  Nodo = Elemento dell'albero in cui vengono contenute le informazioni.
  Radice = Primo nodo dell'albero (il nodo senza archi entranti).
  Ramo = Collegamento (arco orientato) tra una coppia di nodi dell'albero.
  Foglia = Nodo da cui non escono rami.
  Sottoalbero: un sottoinsieme connesso di un albero

- Se in un albero un ramo va da un nodo *n1* ad un nodo *n2*:
  n1 è *padre* di *n2*
  n2 è *figlio* di *n1*

- n1 è *antenato* di n2 se n1 è padre di n2 oppure se n1 è padre di un antenato di n2.
- n2 è *discendente* di n1 se n1 è antenato di n2.
- Altezza = Lunghezza del cammino più lungo dalla radice a una foglia. Es. un nodo con solo radice ha altezza 0, uno con radice e una foglia ha altezza 1, ecc. A volte per convenzione altezza di albero vuoto (senza nodi) è –1.

Albero può essere definito ricorsivamente: base = albero vuoto; passo = radice con sottoalberi.

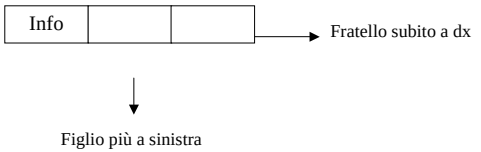


Rappresentazione ricorsiva di un albero

Def. ricorsiva: altezza albero vuoto = -1; altezza albero non vuoto: 1+ altezza max sottoalberi.

In generale ogni nodo ha un numero indefinito di figli. Esistono degli alberi particolari che hanno delle limitazioni sul numero massimo di figli. I più utilizzati sono gli alberi binari. In questi alberi di solito i figli sono individuati dallo loro posizione (nella rappresentazione, da sin a dx).

Per la rappresentazione in memoria si utilizza di solito una struttura dati con due puntatori:

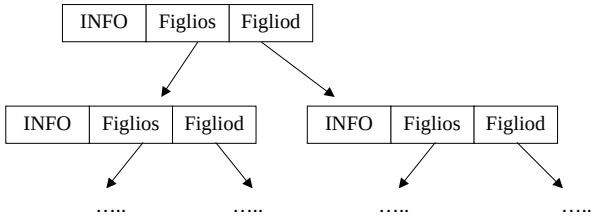


ALBERI BINARI

Sono gli alberi che hanno al più due figli per ogni nodo, distinti esplicitamente in un figlio sx e in uno dx.
Definizione di albero binario:

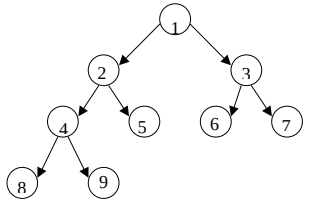
- Un albero vuoto
- nodo + albero sx + albero dx

La struttura dati utilizzata per rappresentare un albero binario è la seguente:



VISITA DI UN ALBERO

La ricerca di un elemento in un albero puó essere effettuata visitando in un ordine opportuno tutti i nodi, confrontando di volta in volta l'elemento con l'informazione del nodo.
Esistono vari ordini di visita di un albero. Considerando ad esempio il seguente albero si ha:



- *in ampiezza* si visita un livello alla volta, da sx a dx: 1 2 3 4 5 6 7 8 9
- *in profondità*:
  - *in preordine (sx)* si visita prima la radice, poi il sottoalbero di sinistra, infini il sottoalbero di destra: 1 2 4 8 9 5 3 6 7
  - *in ordine (sx)* si visita prima il sottoalbero di sinistra, poi la radice, infine il sottoalbero di destra: 8 4 9 2 5 1 6 3 7
  - *in postordine (sx)* si visita prima il sottoalbero di sinistra, poi il sottoalbero di destra, infine la radice: 8 9 4 5 2 6 7 3 1

Codifica in C degli alberi e delle visite in profondità

```
/* NB: non e' ADT */
typedef struct nodo * pnodo; /* tipo puntatore al nodo     */
struct nodo {
  int etichetta; /* valore del nodo */
  pnodo sx, dx;     /* puntatori ai sottoalberi sx e dx */
};
/* funzioni di visita a discesa ricorsiva   */
void preorder (pnodo albero)     /* visita in ordine anticipato   */
void inorder (pnodo albero)  /* visita in ordine simmetrico */
void postorder (pnodo albero)    /* visita in ordine differito */

void preorder (pnodo albero) {
  if (albero != (pnodo) NULL) {
    printf("%d ", albero->etichetta);
    preorder(albero->sx); /* ricorri su sottoalbero sx */
    preorder(albero->dx); /* ricorri su sottoalbero dx */
  } /* if */
  return;
```



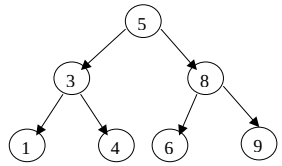
```
} /* preorder */

void inorder (pnodo albero) {
  if (albero != (pnodo) NULL) {
    inorder(albero->sx); /* ricorri su sottoalbero sx   */
    printf("%d ", albero->etichetta);
    inorder(albero->dx); /* ricorri su sottoalbero dx   */
  } /* if */
  return;
} /* inorder */

void postorder (pnodo albero) {
  if (albero != (pnodo) NULL) {
    postorder(albero->sx); /* ricorri su sottoalbero sx     */
    postorder(albero->dx); /* ricorri su sottoalbero dx     */
    printf("%d ", albero->etichetta);
  } /* if */
  return;
} /* postorder */
```

ALBERI BINARI DI RICERCA (Alberi Ordinati)

Lo scopo di questi alberi è di consentire la ricerca di elementi in tempo logaritmico. Un albero binario è detto *Albero Binario di Ricerca* se per ogni nodo gli elementi che stanno nel sottoalbero di sinistra sono tutti minori di quelli che si trovano nel sottoalbero di destra. NB: La stampaInOrdine di un albero ordinato ritorna i valori in ordine.



Albero di ricerca

Alb di sx è di ricerca, quello di dx no

ADT Albero Binario di ricerca: searchtree.h

```
typedef int ElementType;
struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree; /*Puntatore a Nodo: come Position */
SearchTree MakeEmpty( SearchTree T );
Position Find( ElementType X, SearchTree T );
Position FindMin( SearchTree T );
Position FindMax( SearchTree T );
SearchTree Insert( ElementType X, SearchTree T );
SearchTree Delete( ElementType X, SearchTree T );
ElementType Retrieve( Position P );
```


Implementazione: seachtree.c

```
#include "tree.h"
#include <stdlib.h>

struct TreeNode    {
   ElementType Element;
   SearchTree  Left;
   SearchTree  Right;
};
```


Ricerca binaria

Vantaggio di Alberi di ricerca: possibilità di utilizzare ricerca binaria invece di ricerca sequenziale

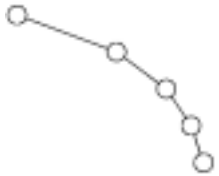
```
Position  Find( ElementType X, SearchTree T )      {
   if( T == NULL )
         return NULL;
   if( X < T->Element )
         return Find( X, T->Left );
   else if( X > T->Element )
         return Find( X, T->Right );
   else
         return T;
                                        }
```

ANALISI DELLA COMPLESSITA' DELLA RICERCA

Per la ricerca di un particolare numero all'interno dell'albero, l'albero viene diviso più o meno in due ad ogni iterazione(esattamente in due se l'albero è bilanciato),con un numero di confronti pari a quello della comune ricerca binaria. Un albero di altezza **h** richiede , al più, **h** passi per ogni ricerca. Avendo **n** nodi in un albero *perfettamente bilanciato* (o quasi…) si ha:

$$\mathbf{h} = \log_2 \mathbf{n}$$

Questo perché tutti i rami hanno la stessa **h** e tutti i nodi hanno due figli.Dunque gli alberi binari, in questa forma, permettono l'implementazione di una struttura dinamica con tempo di ricerca logaritmico. Nel caso pessimo, tuttavia, l'albero si riduce a una struttura lineare, con **h** = Θ(n):



In questo caso la complessità non è più logaritmica, ma O(**n**). Peró, se tutti gli alberi binari di una certa lunghezza *n* sono egualmente probabili, si dimostra che per la ricerca si hanno ~$1{,}38\log_2 n$ confronti in media, cioè *Θ(lg n)*. Esistono tecniche relativamente complicate che cercano di mantenere gli alberi bilanciati per avere casi pessimi in *Θ(lg n)*, e che nel caso medio hanno un numero di confronti limitato da $\log_2 n$ (AVL, Red-Black Trees, AA-Trees, Splay Trees…).

Inserimento in un albero binario

Albero binario prima e dopo l'inserimento di 6


```
SearchTree  Insert( ElementType X, SearchTree T )      {
   if( T == NULL )   {/* Create and return a one-node tree */
        T = malloc( sizeof( struct TreeNode ) );
       if( T == NULL ) {
          printf( "Out of space!!!" ); exit(0);
        }
          else  {
          T->Element = X;
          T->Left = T->Right = NULL;
        }
   }
   else if( X < T->Element )
        T->Left = Insert( X, T->Left );
    else if( X > T->Element )
        T->Right = Insert( X, T->Right );
   /* Else X is in the tree already; we'll do nothing */
return T;
}
```


Rimozione di un elemento

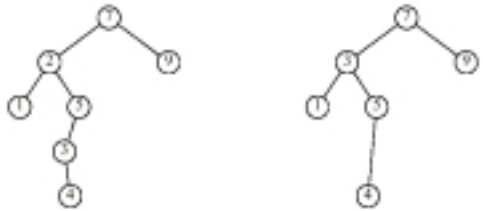
Se il nodo da rimuovere è una foglia, l'eliminazione è immediata;

Se il nodo da rimuovere ha solo uno dei due sottoalberi vuoto: elimina il nodo e "riaggancia" il sottoalbero:

Eliminazione del nodo 5

Se i due sottoalberi sx e dx del nodo da rimuovere sono entrambi non vuoti:

Cancella il nodo e sostituiscilo con il nodo più piccolo del sottoalbero destro (e rimuovi quest'ultimo).



Cancellazione del nodo 2, con due figli

Il nodo più piccolo del sottoalbero dx è proprio il nodo più a sx: è una foglia ed è facile da trovare e rimuovere:

```
Position FindMin( SearchTree T )   {
        if( T == NULL )
            return NULL;
        else
             if( T->Left == NULL )
            return T;
        else
            return FindMin( T->Left );}
```


Codice C della eliminazione, in versione ricorsiva.

```
SearchTree Delete( ElementType X, SearchTree T )    {
        Position TmpCell;
        if( T == NULL )  {printf( "Element not found" ); return NULL;}
        else
        if( X < T->Element )  /* Go left */ T->Left = Delete( X, T->Left );
        else
        if( X > T->Element )  /* Go right */   T->Right = Delete( X, T->Right );
        else  /* Found element to be deleted */
        if( T->Left && T->Right )  /* Two children */   {
            /* Replace with smallest in right subtree */
            TmpCell = FindMin( T->Right );  T->Element = TmpCell->Element;
            T->Right = Delete( T->Element, T->Right );
        }
        else  /* One or zero children */   {
            TmpCell = T;
            if( T->Left == NULL ) T = T->Right; /* Also handles 0 children */
            else if( T->Right == NULL ) T = T->Left;
            free( TmpCell );
        }
        return T;       }
```


Altre funzioni

```
SearchTree  MakeEmpty( SearchTree T ) {
    if( T != NULL ) {
        MakeEmpty( T->Left );
        MakeEmpty( T->Right );
        free( T );
    }
    return NULL;
}


ElementType Retrieve( Position P )      {
    return P->Element;
}
```


Uso alberi di ricerca:

```
#include "tree.h"
#include <stdio.h>
main( ) {
    SearchTree T;   Position P;    int i;    int j = 0;
    T = MakeEmpty( NULL );
    for( i = 0; i < 50; i++, j = ( j + 7 ) % 50 )
        T = Insert( j, T );
    for( i = 0; i < 50; i++ )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );
    for( i = 0; i < 50; i += 2 )
        T = Delete( i, T );
    for( i = 1; i < 50; i += 2 )
        if( ( P = Find( i, T ) ) == NULL || Retrieve( P ) != i )
            printf( "Error at %d\n", i );
    for( i = 0; i < 50; i += 2 )
        if( ( P = Find( i, T ) ) != NULL )
            printf( "Error at %d\n", i );
    printf("Min is %d, Max is %d\n", Retrieve( FindMin(T)), Retrieve(FindMax(T)));
    return 0;}
```

HASHING (trasformazione di chiavi)

NB: L'argomento non è in programma.

Lo hashing è una tecnica che permette, sotto certe ipotesi, l'inserimento e la ricerca in tempo costante all'interno di un array *statico*.

Ricerca \ Dati	10^3 dati	10^6 dati	10^9 dati
Ricerca sequenziale	500	500000	500000000
Ric. Albero binario	15	30	45
Tecnica Hash	2.5	2.5	2.5

Dato un insieme S di elementi e un array *table* di dimensioni *tableSize*, si definisce una funzione:

H: S $\rightarrow$ *{0..tableSize-1}*, che associa a ogni elemento s $\in$ S un indice della tabella.
Se H fosse iniettiva, $H(s_1) \neq H(s_2)$ per $s_1 \neq s_2$. Allora per cercare se s_1 è già presente in table: *table[H(s_1)]* ***ricerca in tempo costante O(1)***

In genere, H NON è iniettiva, anche perché di solito |S| >> tableSize (ad esempio, possibili configurazioni negli scacchi sono troppe per essere memorizzabili tutte in un array).
Se $H(s_1) = H(s_2)$: COLLISIONE. Tecniche hasing si distinguono per come gestiscono collissioni (tipicamente, spostando gli elementi duplicati in un'altra posizione tabella: la ricerca deve quindi esaminare più celle).

Con tecniche molto semplici si riesce a garantire che, *qualunque sia la dimensione della tabella, se questa non è completamente riempita, la ricerca e l'inserimento avvengono ancora in tempo costante con probabilità molto alta* (caso pessimo lineare, ma non capita mai).

Ad es., se tabella riempita fino al 50%, per la ricerca si devono esaminare in media 2.5 celle, mentre se tabella rimepita fino al 90%, si devono esaminare in media 5 celle. Con tecniche più complesse si riesce anche a fare meglio...

Tipiche applicazioni: symbol tables dei compilatori, transposition tables dei giochi, spell checkers.
Difetto: con le tecniche hash la cancellazione degli elementi è molto inefficiente. Inoltre array deve avere dimensioni fissate staticamente.

Header File per Hash Table

```
struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable( int TableSize );
void DestroyTable( HashTable H );
Position Find( ElementType Key, HashTable H );
void Insert( ElementType Key, HashTable H );
ElementType Retrieve( Position P );
/* Routines such as Delete are MakeEmpty are omitted */
```

Esempi di funzione di hash per stringhe

```
typedef unsigned int Index;
Index   Hash1(char *Key, int TableSize )     {
          unsigned int HashVal = 0;
          while( *Key != '\0' )
               HashVal += *Key++;
          return HashVal % TableSize;
}


Index    Hash2(char *Key, int TableSize )    {
           return ( Key[ 0 ] + 27 * Key[ 1 ] + 729 * Key[ 2 ] )  % TableSize;
      }
```