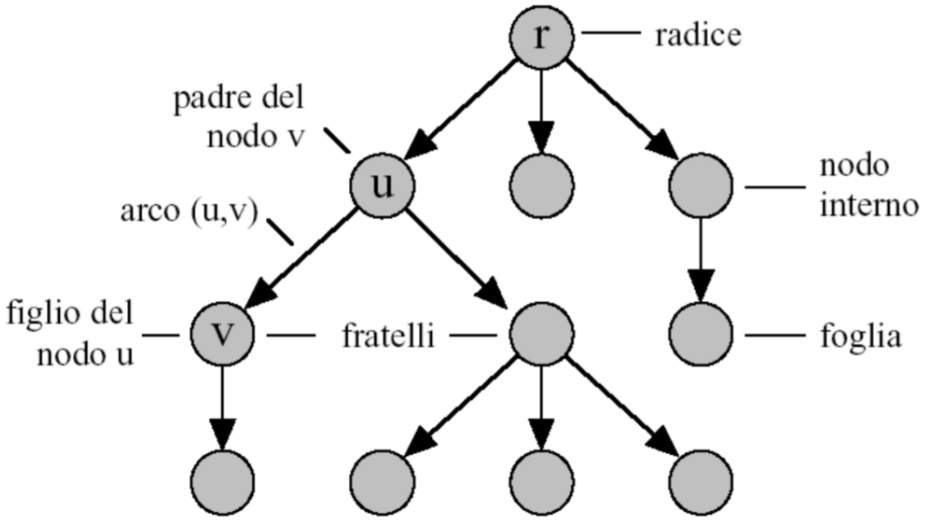
ALBERI

In questo capitolo verranno studiati gli alberi, che rappresentano **gerarchie di oggetti** in cui ad ogni oggetto (*nodo **padre***) sono associati più oggetti (*nodi **figli***) e ogni oggetto ha **un solo nodo padre** tranne l'oggetto iniziale che costituisce la ***radice*** dell'albero; i nodi senza figli sono detti ***foglie***. Alberi particolari sono gli *alberi binari*, in cui ogni nodo ha **al più due figli**. Altri tipi di albero sono gli *alberi ennari* (cioè con numero *n* di figli dove *n* può essere maggiore di 2) e gli *alberi generali* (cioè con numero illimitato di figli).

Le rappresentazioni degli alberi possono essere indicizzate, cioè tramite l'uso di strutture dati statiche, o collegate, cioè tramite l'uso di strutture dati dinamiche.

Per la definizione della classe albero adopereremo una rappresentazione cosiddetta "funzionale", cioè la scansione degli elementi non avviene attraverso un iteratore ma ricorsivamente spostandosi su sottostrutture. gli *alberi ennari* (cioè con numero *n* di figli dove *n* può essere maggiore di 2) e gli *alberi generali* (cioè con numero illimitato di figli).



ALBERI BINARI

Definizioni Generali

Un albero binario è una collezione di oggetti (detti *nodi*) organizzati gerarchicamente. Al livello più alto vi è il nodo *radice* a cui sono collegati zero, uno o due nodi *figli* a cui sono, a loro volta, collegati ulteriori nodi figli e così via fino ad arrivare ai nodi che non hanno figli e sono quindi chiamati *foglie*.

Più formalmente un *albero binario A* è o una collezione vuota di nodi (*albero nullo* o *vuoto*) o una tripla costituita da un nodo *R* (detto *radice*) e da 2 (sotto-)alberi, chiamati rispettivamente *sottoalbero di sinistra* e *sottoalbero di destra* di *R*. Se il sottoalbero di sinistra (o di destra) non è vuoto allora la sua radice è detta *figlio sinistro* (rispettivamente, *destro*) di *R* che, a sua volta, è detto il *padre* di tale nodo. Un nodo che ha ambedue i figli è detto nodo *pieno*; invece se non ha alcun figlio allora *R* è detto nodo *foglia*.

Un albero è raffigurato nel seguente modo (vedi Figura 1); i **nodi** sono indicati con cerchietti e le relazioni da ciascun nodo ai nodi figli sono rappresentati da *rami* (detti anche ***archi***) che vanno dall'alto verso il basso. Pertanto la radice dell'albero è il nodo più in alto ed è l'unico nodo che non ha rami entranti; gli altri nodi hanno esattamente un ramo entrante. Pertanto **se il numero di nodi è *n* il numero di rami è *n*-1.** I nodi foglia non hanno rami uscenti.

Il ***livello*** di un nodo è definito nel seguento modo: **la radice ha livello 1 ed ogni nodo *N* ha il livello *p*+1**, **dove *p* è il livello del nodo padre**, cioè *p*+1 è il numero di nodi compresi nella sequenza di rami da *N* alla radice, *N* e radice inclusi. **La *profondità* di un albero è il massimo livello dei suoi nodi**. **L'albero vuoto ha profondità 0.**

Un albero binario con profondità *k* ha al minimo *k* nodi (*albero degenere*) e al più 2^k -1 nodi (*albero pieno*). Un **albero degenere** è di fatto una **lista di *k* elementi** (vedi Figura 1a) mentre, **in un albero pieno, tutti i nodi a livello inferiore di *k* sono pieni** (vedi Figura 1b), per cui ha 2^0=1 nodo al primo livello, 2^1 nodi al secondo livello, 2^2 nodi al terzo livello e cosi via fino a 2^{k-1} nodi all'ultimo livello e, quindi, il numero totale di nodi è 2^k -1 poichè

$$\sum_{i=0}^{k-1} 2^i = (2-1)\sum_{i=0}^{k-1} 2^i = 2\sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} 2^i = \sum_{i=1}^{k} 2^i - \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

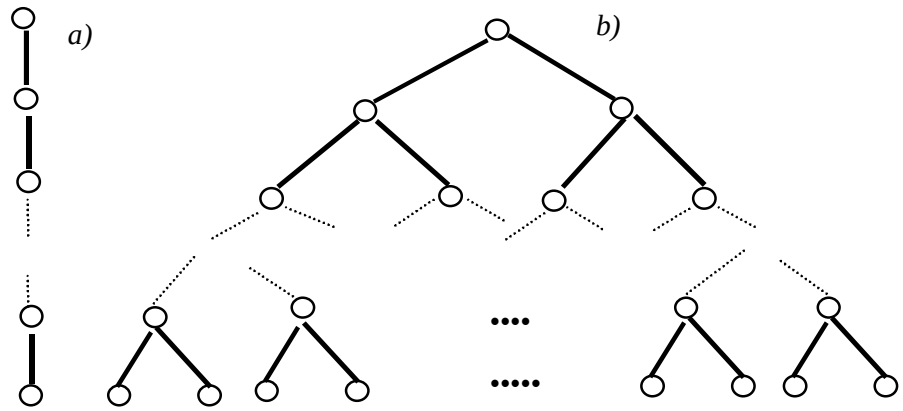
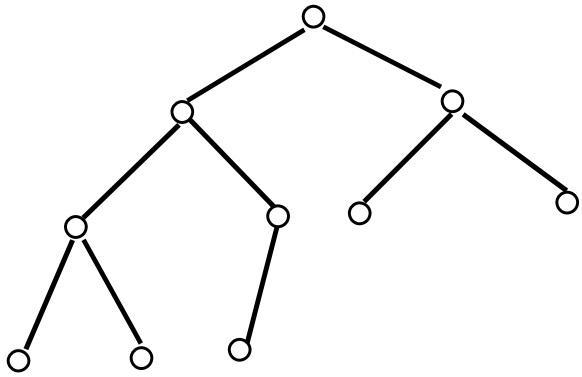


Figura 1. a) *Albero degenere* b) *Albero pieno*

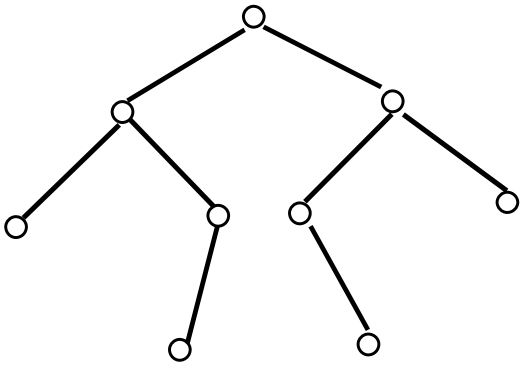
Un albero binario di profondità *k* è *completo* se i nodi dei livelli inferiori a *k* formano un albero pieno e per ogni nodo il sottoalbero di sinistra ha profondità non minore di quello di destra. In altre parole **in un albero completo tutti i livelli tranne l'ultimo sono riempiti e l'ultimo livello è riempito da sinistra a destra** per cui eventuali nodi mancanti sono collocati tutti nella parte destra del livello (vedi Figura 2a).

Un albero binario è ***bilanciato* se per ogni nodo la profondità del sottoalbero di destra e la profondità del sottoalbero di sinistra differiscono al più di uno** (vedi Figura 2b).

Un albero con *n* nodi può avere al massimo profondità *n* (caso di albero degenere) e **al minimo profondità $\lceil \log(n+1) \rceil = \Theta(\log(n))$.** Sia alberi bilanciati che completi hanno la minima profondità. Lo stesso vale ovviamente per gli alberi pieni ma non è detto che esista un albero pieno con *n* nodi. Infatti è facile verificare che se $n+1$ è una potenza di 2 allora albero pieni e alberi completi esistono e coincidono mentre per gli altri valori di *n* non esistono alberi pieni mentre esistono ovviamente alberi completi. Per questa ragione la nozione di albero pieno non è in pratica utile e viene sostituita dalla nozione più generale di albero completo o dalla nozione più pratica di albero bilanciato.



a) Albero completo di profondità 4



b) Albero bilanciato di profondità 4

Figura 2. *Albero completo e albero bilanciato*

Metodi di Rappresentazione degli alberi

Rappresentazioni indicizzate

Vedremo ora due delle più comuni rappresentazioni di alberi basate su array: il *vettore dei padri* e il *vettore posizionale*. Entrambe richiedono spazio O(n) per un albero con n nodi. L'idea di base è quella di rappresentare ogni nodo dell'albero con una cella di un array che contiene l'informazione associata al nodo, più eventualmente altri indici che consentono di raggiungere altri nodi dell'albero. Sebbene di facile realizzazione, le rappresentazioni basate su array rendono tipicamente difficoltoso l'inserimento e la cancellazione di nodi nell'albero.

Vettore dei padri

La più semplice rappresentazione possibile per un albero è forse quella basata sul vettore dei padri. Sia T=(N,A) un albero con n nodi numerati da 0 a (n-1). Un vettore dei padri è un array P di dimensione n le cui celle contengono coppie ***(info, parent)*** : per ogni indice v in [0, n-1], P[v].info è il contenuto informativo del nodo v, mentre P[v].padre = u se e solo se vi è un arco (u,v) in A. Se invece v è la radice, allora P[v].parent=null.

Si noti che, usando un vettore dei padri, da ogni nodo è possibile risalire in tempo O(1) al proprio padre, mentre trovare un figlio richiede una scansione dell'array in tempo O(n).

Vettore posizionale

Nel caso particolare di alberi d-ari completi, con d>=2, è possibile usare una rappresentazione indicizzata dove ogni nodo ha una posizione prestabilita nella struttura. Sia T=(N,A) un albero d-ario con **n nodi numerati da 1 a n**. Un vettore posizionale è un arra P di dimensione n+1 tale che P[v] contiene l'informazione associata al nodo v, e tale che **l'informazione associata all'i-esimo figlio di v è in posizione P[d*v+1], per i compreso tra 0 e (d-1)**.

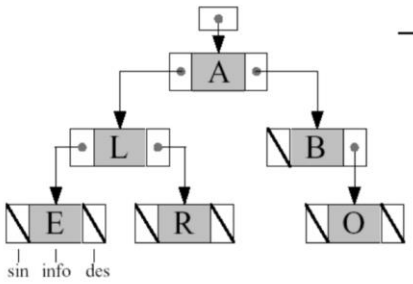
Ad esempio, dato il nodo 1 di un albero binario, il figlio sinistro di 1 sarà in posizione 2*1+0=2 ed il figlio destro di 1 sarà in posizione 2*1+1=2. **Per semplicità, la posizione 0 di P non viene utilizzata.**

Utilizzando la formula inversa, dato un nodo v è possibile risalire al padre di v utilizzando la formula int(v/d).

Si noti che, in questa rappresentazione, le relazioni tra i nodi non sono esplicite, ma definite implicitamente dalla posizione relativa dei nodi della struttura. Si noti che da ogni nodo v è possibile risalire in tempo O(1) sia al proprio padre (che ha indice int(v/d) se v non è la radice), che a uno qualunque dei figli.

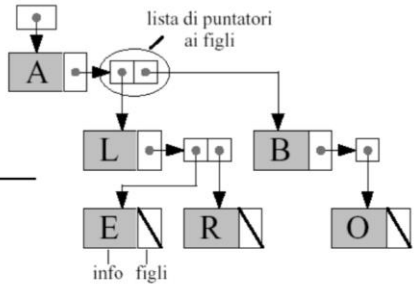
Rappresentazioni collegate

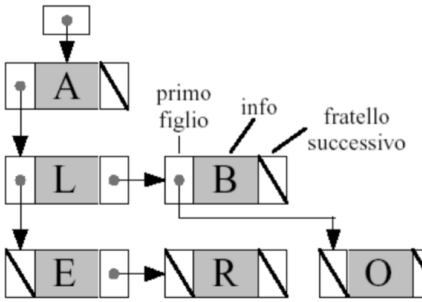
Sono possibili tre tipologie di rappresentazioni collegate: (1) *puntatori ai figli* (2) *Lista dei figli* (3) primo figlio-fratello successivo. Queste sono raffigurate nelle seguenti immagini:



Rappresentazione con puntatori ai figli (nodi con numero limitato di figli)

Rappresentazione con liste di puntatori ai figli (nodi con numero arbitrario di figli)





Rappresentazione di tipo primo figlio-fratello successivo (nodi con numero arbitrario di figli)

Le rappresentazioni collegate consentono naturalmente di occupare lo spazio minimo necessario ai nodi presenti nell'albero, al costo dello spazio aggiuntivo dei puntatori. Si noti che alcune di tali rappresentazioni sono adatte alla rappresentazione di alberi generici, in cui non è noto a priori il numero massimo di figli di ciascun nodo.

Rappresentazione di Alberi Binari

In questa sezione presentiamo una definizione generale di albero binario cominciando dalla definizione di *Snodo,* la struttura di memorizzazione di un nodo composto dalla parte informativa, il puntatore al possibile padre e i puntatori ai due possibili figli.

```
enum Direzione { SIN=0, DES=1 };

template <class T>
struct SNodo{
      T vinfo; // parte informativa
      SNodo *ppadre, *pfiglio[2]; // puntatori al padre e ai due figli
      SNodo( const T& inf ): vinfo(inf)
      {     ppadre = pfiglio[SIN] = pfiglio[DES] = 0;
      }
      ~SNodo( ) {delete pfiglio[SIN]; delete pfiglio[DES];}
};
```

Passiamo ora a definire la classe degli alberi binari adottando una realizzazione di tipo funzionale per cui la visita di un albero avviene attraverso l'accesso ai sottoalberi e non attraverso la scansione dei vari nodi come per le liste. Per evitare costi elevatissimi, alberi e sottoalberi condividono nodi per cui sono possibili effetti collaterali tra di essi.

La classe degli alberi binari contiene la posizione della radice e le classiche funzioni di utilizzo. In particolare i figli possono essere indirizzati direttamente tramite l'indice *SIN/DES*.

```
template <class T>
class AlberoB
{
protected:
      SNodo<T>* pradice; // puntatore alla radice
public:

      //    FUNZIONI NON COSTANTI
    AlberoB () : pradice(0) {};

    AlberoB ( const T& a) {
        pradice = new SNodo<T>(a);
    };

      //    inserisce l'albero AC come figlio d = SIN/DES
    void insFiglio ( Direzione d, AlberoB& AC ) {
        assert( !nullo() );
        assert( figlio(d).nullo() );
        if ( !AC.nullo() ) {
            pradice->pfiglio[d]=AC.pradice;
            AC.pradice->ppadre=pradice;
        }
    };

      //    estrae il figlio d = SIN/DES
    AlberoB<T> estraiFiglio ( Direzione d ) {
        assert( !nullo() );
        AlberoB<T> A = figlio(d);
        A.pradice->ppadre=0;
        pradice->pfiglio[d] = 0;
```

```
        return A;
    };

    // modifica il contenuto informativo della radice
    void modRadice ( const T& a ) {
        assert( !nullo() );
        pradice->vinfo = a;
    };

      // svuota l'albero cancellandone tutti i nodi
    void svuota() { delete pradice; pradice = 0; };

      // svuota l'albero ma senza cancellare i nodi
    void annulla() { pradice = 0; };

    //      FUNZIONI COSTANTI
      bool nullo() const { return pradice == 0; };

      // restituisce una copia dell'albero
    AlberoB<T> copia () const {
        if ( nullo() ) return AlberoB<T>();
        AlberoB<T> AC(radice());
        AlberoB<T> fs = figlio(SIN).copia();
        AlberoB<T> fd = figlio(DES).copia();
        AC.insFiglio(SIN,fs);
        AC.insFiglio(DES,fd);
        return AC;
    } ;

      //    mostra l'info della radice
    const T& radice () const {
        assert( !nullo() ); // se sto chaimando radice su un albero nullo =>
esegui l'assert
        return pradice->vinfo;
    };

      // restituisce true se la radice è nodo foglia
    bool foglia () const {
        return !nullo()&&figlio(SIN).nullo()&& figlio(DES).nullo();
    };

      // restituisce il figlio d = SIN/DES
    AlberoB<T> figlio ( Direzione d ) const {
        assert( !nullo() );
        AlberoB<T> AC;
        AC.pradice = pradice->pfiglio[d];
        return AC;
    };

      //    restituisce il padre eventualmente nullo
    AlberoB<T> padre () const {
        assert( !nullo() );
        AlberoB<T> AC;
        AC.pradice = pradice->ppadre;
        return AC;
    };

};
```

La complessità spaziale di *AlberoB* è $\Theta(n\ (K+P))$, dove n è il numero di nodi nell'albero, K è la dimensione di un oggetto di tipo T, e P è lo spazio necessario per memorizzare un tipo indirizzo. Assumendo che K e P siano costanti, la complessità spaziale diventa lineare in n. La funzione *svuota* e la funzione *copia* hanno complessità $\Theta(n)$ mentre le altre funzioni hanno complessità temporale costante. Ovviamente l'operatore "=" e il costruttore per copia, che sono quelli di default e , quindi, di tipo superficiale, hanno complessità costante.

Di seguito mostriamo una funzione che calcola la profondità di un albero. Successivamente una funzione, non efficiente, che verifica se un albero sia bilanciato o meno riutilizzando la funzione appena definita. Infine, una funzione efficiente che calcola la profondità e, nello stesso tempo, verifica se esso sia bilanciato o meno.

```
template <class T>
int profondita( AlberoB<T> A )
{
      if (A.nullo()) return 0;
      int p1=profondita(A.figlio(SIN));
      int p2=profondita(A.figlio(DES));
      return (p1>p2)? p1+1 : p2+1;

}
```

Complessità lineare nel numero di nodi.

```
template <class T>
bool bilanciato(AlberoB<T> A )
{
      if (A.nullo()) return true;
      int p1=profondita(A.figlio(SIN));
      int p2=profondita(A.figlio(DES));
      bool b1=(abs(p1-p2)<=1);
      bool b2=bilanciato(A.figlio(SIN));
      bool b3=bilanciato(A.figlio(DES));

      return b1&&b2&&b3;
}
```

ATTENZIONE! In questo caso per ciascun nodo ricalcolo la profondità…l'algoritmo ha complessivamente una complessità esponenziale (per ogni nodo devo elaborare una funzione lineare nel numero d nodi del sotto-albero)

```
template <class T>
bool alberoBilanciatoProfondo( AlberoB<T> A, int& p )
{     if ( A.nullo() )
      {     p = 0;
            return true;
      }
      Card p1, p2;
      bool b1 = alberoBilanciatoProfondo(A.figlio(SIN),p1);
      bool b2 = alberoBilanciatoProfondo(A.figlio(DES),p2);
      p = max(p1,p2)+1;
      return b1 && b2 && (abs(p1-p2) <= 1);
}
```

La complessità è $\Theta(n)$, cioè lineare nel numero di nodi.

Visita di Alberi Binari

Negli alberi generali sono rilevanti quattro tipi di visita:

(i) *visita in pre-ordine (o anticipata),* in cui viene prima visitata la radice e poi i due sottoalberi;

(ii) *visita in post-ordine (o posticipata)*, in cui vengono visitati i due sottoalberi e poi la radice;

(iii) *visita simmetrica (o infissa)* in cui viene visitato prima il sottoalbero di sinistra, poi la radice e infine il sottoalbero di destra

(iv) *visita per livelli*, in cui viene visitata la radice (nodo di livello 1), poi i nodi di livello 2 e così via fino ai nodi dell'ultimo livello.

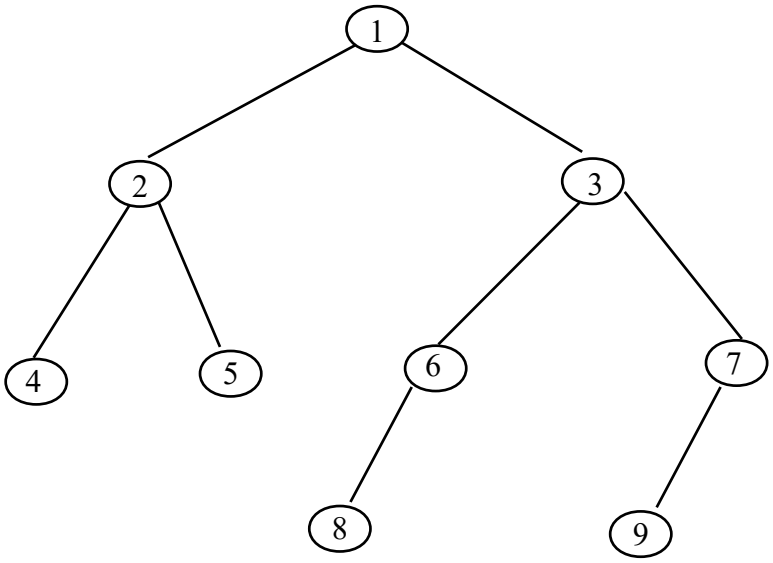


Figura 4. *Albero Binario*

Esempio 2. Consideriamo l'albero generale in Figura 4. Nella visita anticipata, i nodi vengono scanditi in quest'ordine: 1, 2, 4, 5, 3, 6, 8, 7, 9; nella visita posticipata in quest'ordine: 4, 5, 2, 8, 6, 9, 7, 3, 1; nella visita infissa in quest'ordine: 4, 2, 5, 1, 8, 6, 3, 9, 7; infine, nella visita per livelli in quest'ordine: 1, 2, 3, 4, 5, 6, 7, 8, 9; .

Presentiamo ora una funzione che effettua la visita infissa di un albero binario, restituendo la lista dei nodi nell'ordine della visita.

```
#include <list>
template <class T>
void visitaSimmetrica( const AlberoB<T>& A, list<T>& L )
{     if ( !A.nullo() )
      {     visitaSimmetrica(A.figlio(SIN),L);  //**
            L.push_back(A.radice());            //**
            visitaSimmetrica(A.figlio(DES),L);  //**
      }
}
```

Per provarla basta lanciare questo semplice codice:

```
int main()
{     AlberoB<int> A;
      ///CREA UN ALBERO…
      list<int> L;
      visitaSimmetrica(A, L );
      for (auto a:L) cout<<a<<" ";
      cout<<endl;

    return 0;
}
```

Per richiamare la funzione bisogna accertarsi che *L* sia inizialmente vuota. Le funzioni per le visite in pre-ordine e in post-ordine si ottengono cambiando la sequenza delle istruzioni asteriscate: per la visita anticipata l'istruzione *L.push_back(A.radice())* precederà le due chiamate ricorsive mentre nella visita posticipata le seguirà. Poichè ogni nodo è ritrovato esattamente una volta, tutte e tre le funzioni di visita hanno complessità $\Theta(n)$, dove *n* è il numero di nodi.
La scrittura della visita per livelli è più complessa in quanto non è più immediato utilizzare la ricorsione mentre per poter utilizzare uno schema iterativo è necessario introdurre una struttura dati apposita, la coda.

```
#include <queue>
void visitaPerLivelli(const AlberoB<int>& a) {
    if (a.nullo())
        return;

    std::queue<AlberoBInt> q;
    q.push(a);

    while(!q.empty()) {
        AlberoBInt temp = q.front(); // q.front() restituisce l'oggetto in
                                     //testa alla coda SENZA RIMUOVERLO
        q.pop();              // q.pop() rimuove l'oggetto in testa alla coda

            // elaborare l'albero temp
        std::cout << temp.radice() << ' ';

        if (!temp.figlio(SIN).nullo())
            q.push(temp.figlio(SIN));
        if (!temp.figlio(DES).nullo())
            q.push(temp.figlio(DES));
    }
}
```

La funzione *visitaPerLivelli* ha anch'essa una complessità di $\Theta(n)$ in quanto il ciclo di *while* è eseguita tante volte quanti sono i nodi inseriti nella coda e questi sono pari al numero totale di

figli nell'albero, cioè *n*-1. Possiamo quindi dire che tutte le quattro funzioni di visita presentate sono algoritmi ottimi.

Ricerca in Alberi Binari

Si vuole ora realizzare una funzione che ricevuto un albero A di elementi di tipo T ed un elemento v di tipo T restituisca il sotto-albero la cui radice contiene v, se esiste, altrimenti restituisce un albero nullo.

```
template <class T>
AlberoB<T> cerca(const AlberoB<T>& a, T v) {
    if (a.nullo() || a.radice() == v)
        return a;

    // a questo punto sono in un sottoalbero che non ha v come valore
      //informativo e non è nullo

    AlberoB<T> temp = cerca(a.figlio(SIN), v);
    if (!temp.nullo())
        return temp;
    // a questo punto so che nel sottoalbero sinistro non c'era un albero con
      // v come valore informativo
    return cerca(a.figlio(DES), v);
}
```

ALBERI BINARI DI RICERCA

4.2.1 Alberi Binari non Bilanciati

Sia dato un albero binario *A* tale che esiste un ordinamento totale *O* per l'insieme dei suoi nodi. L'albero binario *A* è un *albero binario di ricerca* (rispetto ad *O*) se **per ogni nodo *v* di esso tutti i nodi nel sottoalbero sinistro di *v* precedono *v* e tutti i nodi nel sottoalbero destro non precedono (cioè seguono o sono uguali a) *v*** nell'ordinamento *O*.

Le operazioni fondamentali sono cercare/ inserire/ cancellare un oggetto con particolare chiave d'ordinamento, spostarsi al nodo con valore di chiave successiva o o precedente a quella del nodo indicato, posizionarsi direttamente sul nodo con valore di chiave minimo (*primo* oggetto nell'ordinamento) o massimo (*ultimo* oggetto nell'ordinamento).

La struttura dati AlberoBR può essere definita estendendo la classe AlberoB precedentemente introdotta, in cui si rimuove la possibilità di inserire in un punto qualsiasi e si definiscono le seguenti interfacce principali:

```
AlberoB<T>::svuota; // svuota l'albero
AlberoB<T>::foglia; // true se la radice è nodo foglia
AlberoB<T>::nullo; // true se albero nullo
AlberoB<T>::radice; // mostra l'info della radice
void  ins( const T& a ) // Inserimento ordinato
void  canc( const T& a ) // Cancellazione di un nodo
AlberoBR<T> padre ( ) const
AlberoBR<T> figlio ( Direzione d ) const
bool cerca (const T& a)
list<T> compresiTra (const T& a1, const T& a2)
```

La complessità della funzione di ricerca di un elemento ha complessità $\Theta(n)$ nel caso peggiore, $\Theta(\log n)$ nel caso medio e $\Theta(1)$ nel caso migliore.

4.2.2 Alberi Binari Bilanciati

Abbiamo visto che la complessità della ricerca, e quindi delle funzioni di cancellazione e inserimento, hanno complessità lineare nel caso peggiore. Per rendere sempre efficiente un albero binario di ricerca è quindi necessario evitare un suo sbilanciamento attraverso un'opportuna ristrutturazione dell'albero. Una soluzione a questo problema, introdotta da Adelson-Velskii e Landis nel 1962, è basata su strutture ad albero che prendono il nome di alberi AVL.

Una *albero AVL* è un albero binario di ricerca bilanciato in cui ad ogni nodo viene associato un valore +1, 0, -1, detto *fattore di bilanciamento*, pari alla profondità dell'albero di sinistra meno la profondità del sottoalbero di destra. Gli algoritmi per la gestione degli alberi AVL, pur non essendo complicati, sono alquanto lunghi e noiosi e pertanto ci limitiamo a descrivere

informalmente come essi mantengano il bilanciamento di un albero sfruttando l'informazione fornita dal fattore di bilanciamento.

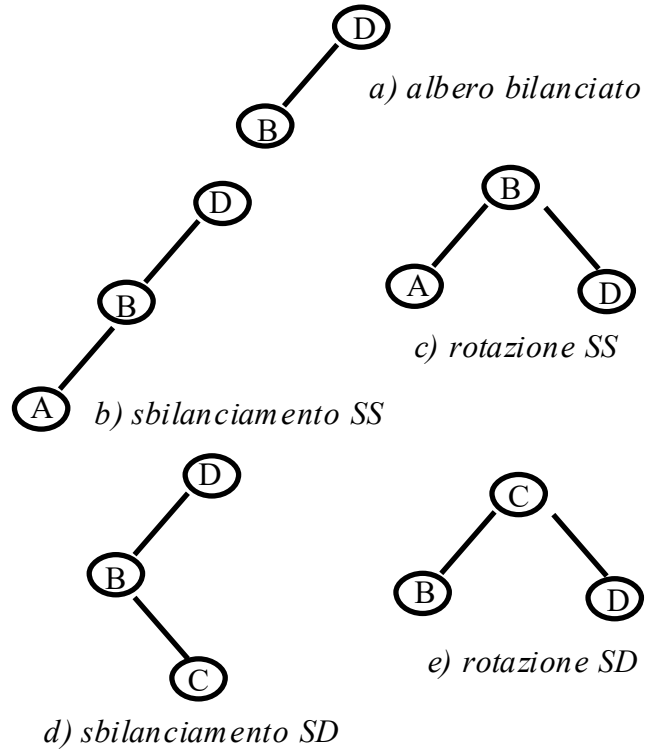


Figura 4. *Rotazione SS e SD di alberi AVL*

Quando si effettua un inserimento o una cancellazione, bisogna risalire in tempo $O(\log n)$ dal nodo inserito o cancellato fino alla radice per eventualmente aggiornare il fattore di bilanciamento dei vari nodi. Se accade che un fattore di bilanciamento diventa 2 o -2 bisogna ristabilire il bilanciamento mediante alcune operazioni sui nodi dette *rotazioni*. Sono possibili 4 tipi di rotazione: *SS* (sinistra-sinistra), *SD* (sinistra-destra), *DS* (destra-sinistra) e *DD* (destra-destra). In Figura 4 sono mostrate le rotazioni SS e SD per il semplice caso di albero con soli due nodi a cui viene aggiunto un terzo nodo che lo sbilancia; le altre due rotazioni sono simmetriche. Recentemente al posto degli alberi AVL si preferisce usare i cosidetti *alberi rosso-neri*.