**Ciclo di sviluppo del software**

- Studio di fattibilità
- Raccolta e analisi dei requisiti
- Progettazione
- Verifica
- Manutenzione

**Qualità dei programmi**

- Esterne
  - Correttezza
  - Efficienza
  - Robustezza
  - Usabilità
- Interne
  - Riusabilità
  - Modularità
  - Estensibilità
  - Portabilità e compatibilità
  - Leggibilità
  - Bontà della documentazione

**Fase di progettazione**

- Principi
- Tecniche
- Strumenti

**Principi di progettazione: ASTRAZIONE**

Tecniche di astrazione:
- Programmazione strutturata
- Modularizzazione
- Astrazione dati

Una descrizione specifica di un sistema che pone enfasi su alcuni dettagli o proprietà e ne elimina altri. Fornisce un buon modo per controllare la complessità e garantire la continuità di sistemi software complessi.
I requisiti o le funzionalità del sistema giocano il ruolo di osservazioni che devono essere "spiegate". Il processo di astrazione prevede il decidere di quali caratteristiche sono rilevanti e quali parametri andrebbero inclusi. Quale formalismo descrittivo adottare e come validare il modello.

**Astrazione per**

**- Decomporre e rappresentare**

La scomposizione di un problema in sottoproblemi è un metodo semplice e intuitivo per affrontare problemi complessi. Si astrae dalle caratteristiche peculieri dei sottoproblemi e ci si concentra sulla loro funzionalità e interazione

**- Generalizzare**

Si astrae sempre quando si generalizza: rendere generale un metodo solutivo messo a punto per un problema particolare impone di comprendere l'essenza del metodo prescindendo dalle peculiarità del caso.

**- Astrazione su dati e funzioni**

Astrazione sui dati: consente di far riferimento a strutture algebrico-matematiche, caratterizzate da valori e da operazioni su tali valori. Si prescinde dal modo in cui queste strutture sono organizzate e realizzate mediante i costrutti di un linguaggio di programmazione.

Astrazione sulle funzioni: consente di concentrare l'attenzione su “cosa” fa una particolare operazione piuttosto che sul “come” è fatta. Si astrae dalle modalità di realizzazione, ci si concentra sul compito o sulle funzionalità di un segmento di programma.

## INTRODUZIONE OO PROGRAMMING

**Oggetto:** è definito come un gruppo di procedure che condividono uno stato, un oggetto è una collezione di dati (stato dell'oggetto) e procedure in grado di alterare lo stato (comportamento).
**Classe:** è una collezione di oggetti che condividono gli stessi attributi, dove un attributo è il tipo di un dato o di un metodo che manipolano i dati.

**Un linguaggio OO supporta**
- Information hiding (incapsulamento)
- Data abstraction
- Message passing (polimorfismo)
- Ereditarietà e dynamic binding

Gli oggetti sono costituiti da dati privati e da operazioni permesse su tali dati.
Gli oggetti comunicano fra di loro con il passaggio di messaggi che non sono altro che chiamate a procedure.

**Classi e polimorfismo**
Una classe è una descrizione di oggetto da instanziare, un tipo è una descrizione di variabili da dichiarare. Nei linguaggi OO una classe è una collezione di oggetti in cui ogni oggetto della classe include gli stessi metodi e le stesse variabili ma può includere diversi valori per i dati.

**Ereditarietà**
I linguaggi OO supportano oggetti, classi di oggetti e ereditarietà di attributi di una sottoclasse da una classe ad un livello più alto nella gerarchia. Quando un oggetto riceve un messaggio, verifica l'esistenza di un metodo per rispondere, se non c'e' verifica l'esistenza nella gerarchia delle classi.

Gli attributi possono essere ridefiniti, specifici o ereditati.
Ereditarietà multipla: Alcuni oggetti possono ereditare da più classi, in questo caso si può verificare il problema del name clash fra metodi, per risolvere si applica l'operatore di scope resolution ::

## ALGEBRE DI DATI

**Astrazione funzionale**
Un programma corrisponde alla tripla {D,A,R}, si può dire che siccome trasforma risultati iniziali in risultati allora definisce un nuovo operatore sui dati. Il repertorio di operatori disponibili sui dati può essere ampliato scrivendo programmi.
L'astrazione funzionale permette di potenziare il linguaggio introducendo nuovi operatori definiti attraverso sottoprogrammi in cui si fa uso di operatori di base gia' disponibili. L'intestazione del sottoprogramma ne introduce il nome e gli argomenti dell'operatore, mentre il corpo ne descrive le regole di comportamento. I costrutti per realizzare l'astrazione funzionale consentono di definire una specifica e una realizzazione. La specifica definisce cosa ci si aspetta dalla funzione, cioè la relazione che caratterizza il legame tra dati di ingresso e risultati. Mentre la realizzazione definisce come il risultato viene ottenuto

**Astrazione dati**
Ricalca ed estende il concetto di astrazione funzionale, cosi come l'astrazione funzionale permette di ampliare l'insieme dei modi di operare sui dati, l'astrazione dati permette di ampliare i tipi di dati disponibili attraverso l'introduzione di nuovi tipi di dati e di operatori.
L'astrazione funzionale evidenzia operazioni ricorrenti o ben caratterizzate all'interno della soluzione di un problema, l'astrazione dati sollecita ad individuare le organizzazioni di dati più adatte alla soluzione del problema.

**Algebra**
Un astrazione dati consente un estensione dell'algebra dei dati. Un algebra è un sistema matematico costituito da un dominio, cioè un insieme di valori e da un insieme di funzioni applicabili su tali valori. Il numero, il tipo e le proprietà delle funzioni sono gli elementi essenziali di un algebra.
Corrispondenza tra tipo astratto e algebra:
insiemi di valori → domini di definizione
funzioni → operazioni associate

Come si estende l'algebra dei dati disponibile in un linguaggio di programmazione? Molti linguaggi forniscono i mezzi per definire e creare nuovi tipi di dati ma non tutti permettono di fare astrazione dati o creare tipi di dati astratti. Un tipo di dati si dice astratto se le operazioni applicabili sugli oggetti che li rappresentano sono isolate dai dettagli usati per realizzare il tipo.

Un algebra dei dati può essere costituita da:
A- Una famiglia di insiemi (insieme di dati)
B- Una famiglia di operatori sui dati (operatori)
C- Un repertorio di simboli o nomi per indicare l'insieme di dati
D- Un repertorio di simboli o nomi per indicare gli operatori
E- Un repertorio di simboli detti costanti per indicare elementi singoli degli insiemi di dati

esempio:
A- La famiglia di insiemi comprende: numeri interi, booleani, stringhe
B- la famiglia di operatori comprende: op aritmetici, di congiunzione disgiunzione e negazione per i booleani, operatori di confronto fra interi, operatori di concatenazione.
C- Il repertorio sei simboli per l'insieme dei dati potrebbe essere: integer, boolean, stringhe
D- il repertorio di simboli per gli operatori: + - * / and or not > < = substr &
E- Gli elementi singoli per gli insiemi di dati (le costanti): ogni intero viene indicato da una sequenza di cifre decimali preceduta eventualmente da + o - . Ogni stringa viene indicata da una sequenza di caratteri racchiusi tra apici, i valori booleani sono indicati da true e false

Anche un astrazione dati è costituita da:
- specifica: ha il compito di descrivere sinteticamente il tipo dei dati e gli operatori che li caratterizzano.
- realizzazione: stabilisce come i dati e gli operatori vengono ricondotti ai tipi di dati e agli operatori già disponibili.

**Requisiti dell'astrazione di dati**
**Requisito di astrazione**
Il requisito di astrazione è verificato quando:
i programmi che usano un astrazione possono essere scritti in modo da non dipendere dalle scelte di realizzazione.
La mancanza del requisito di astrazione:
si manifesta con l'impossibilità di dichiarare direttamente variabili con il nuovo tipo definito.

**Requisito di protezione**
E' verificato se sui nuovi dati si può lavorare esclusivamente con gli operatori definiti all'atto della specifica.
La mancanza del requisito di protezione si manifesta con la possibilità di lavorare con gli operatori definiti per i nuovi dati anche su dati con rappresentazioni simili ma non omogenei per tipo.

**Specifica e realizzazione**
Una specifica ha il compito di descrivere sinteticamente il tipo di dei dati e gli operatori che li caratterizzano, si distingue in:
- specifica sintattica: l'elenco dei nomi dei tipi di dato utilizzati per definire la struttura, delle operazioni specifiche della struttura e delle costanti. I domini di partenza e arrivo, cioè il tipo degli operandi e del risultato per ogni operatore.

- specifica semantica: definisce il significato dei nomi introdotti con la specifica sintattica. Associa un insieme ad ogni nome di tipo introdotto nella specifica sintattica, un valore ad ogni costante, una funzione ad ogni nome di operatore esplicitando precondizioni (quando l'operatore è applicabile) e postcondizioni(stabilisce come il risultato è vincolato agli argomenti dell'operatore).

La realizzazione riconduce la specifica ai tipi di dato primitivo e agli operatori già disponibili. Alla stessa specifica possono corrispondere diverse realizzazioni più o meno efficienti.

Nel valutare l'efficienza di sottoprogrammi che utilizzano tipi di dato primitivi si prescinde dalle caratteristiche della macchina, assumendo di far riferimento ad un organizzazione molto semplice. Si fa riferimento a dati contenuti in memoria, memoria organizzata in celle, celle accessibili tramite indirizzo, indirizzi che sono interi consecutivi, dati elementari che non decidono la capacità di una cella, variabili contenute in celle di indirizzo noto il cui accesso richiede tempo costante.

Un costrutto adatto alla realizzazione di astrazione dati dovrà:
dichiarare esplicitamente quai sono i nuovi tipi di dati e quali sono i nuovi operatori. Definire la rappresentazione dei nuovi dati in termini di dati pre-esistenti. Contenere un insieme di sottoprogrammi che realizzino gli operatori definiti sui nuovi dati.

**Strutture dati**
Come è noto, il tipo di dato è un modello matematico che consiste nella definizione di una collezione di valori (dominio) su quali sono ammesse certe operazioni (funzioni).
Una struttura di dati è un particolare tipo di dato, caratterizzato più dalla organizzazione imposta agli elementi che la compongono  che dal tipo degli elementi componenti. Le strutture solitamente disponibili nei linguaggi di programmazione sono tabelle monodimensionali i cui elementi sono contenuti in memoria in locazioni contigue.

In generale distinguiamo strutture:
lineari: dati disposti in sequenza
non lineari: in cui non è individuata una sequenza
a dimensione fissa (statiche): in cui il numero di elementi non può variare nel tempo
a dimensione variabile (dinamiche): in cui il numero di elementi può variare nel tempo
omogenee: dati dello stesso tipo
non omogenee: dati non dello stesso tipo

**Tecniche di specifica**
Due tecniche per la scrittura di specifiche sono:
- Specifiche assiomatiche (o algebriche)

- Modelli astratti (approccio costruttivo)
La differenza tra le due tecniche sta nel come viene definita la semantica degli operatori. Le specifiche assiomatiche sono self-contained, cioè specificano ogni oggetto come composizione di funzioni. I modelli astratti definiscono la semantica delle operazioni in termini di un altro ben definito tipo di dato. La chiave di queste e altre tecniche di specifiche sta nel fatto che la descrizione della semantica non fa riferimento alla realizzazione.

**Specifica di un dato astratto**
Possiamo definire una specifica di un tipo astratto come una tripla (D,F,A) con:
D: l'insieme di tutti i domini per la definizione del dato. Il tipo del dato che stiamo definendo indicato con il simbolo d viene chiamato dominio designato, tutti gli altri domini sono detti domini ausiliari.
F: insieme degli operatori
A: insieme degli assiomi o regole che descrivono la semantica degli operatori

**Tipi di operatori**
Di base: costruttori, modificatori, osservatori
Aggiuntivi: distruttori, iteratori

**Specifica assiomatica (o algebrica)**
Le specifiche assiomatiche definiscono il comportamento di un tipo di dato astratto dando gli assiomi che legano tra loro gli operatori.

**C++ GUIDELINES**

**Reference**
Un riferimento è un nome alternativo per identificare un oggetto, un alias. Di solito vengon ousati come parametri di funzione.
Un riferimento deve essere inizializzato quando creato e poi non può essere più cambiato. Non esistono reference null.

**Pointer**
Un puntatore punta ad una locazione di memoria e conserva l'indirizzo di quella locazione, può far riferimento a qualsiasi cosa.
Un puntatore si dichiara: int* p;
Un modo per ottenere un valore puntatore è attraverso l'operatore indirizzo: int* p = &ival;
Il contenuto della memoria al quale punta un puntatore è accessibile
con l'operatore deferenziatore: cout << *p << endl; //prints 1024
*p = 1025;

I puntatori possono essere deferenziati,  i riferimenti no
I puntatori possono essere indefiniti o null, i riferimenti no
I puntatori possono essere cambiati per puntare ad altro, i riferimenti no

**Alias**
Un nome può essere definito come sinonimo di un nome di tipo già esistente
Tradizionalmente si usa il typedef, nel nuovo standard esiste una nuova dichiarazione di alias

```
using newType = existingType; //C++11
typedef existingType newType; //equivalent
```

**auto e decltype (C++11)**
Auto permette al compilatore di dedurre il tipo da un inizializzatore
auto it = v.begin(); //begin() returns vector<int>::iterator

Quando si usa auto per dichiarare variabili, i top-level & e const vengono rimossi

```
int i = 0;
int& r = i;
auto a = r; // a is int, the value is 0
auto& b = r; // b is ref to int
const int ci = 1;
auto c = ci; // c is int, the value is 1
const auto d = ci; // d is const int, the value is 1
```

A volte vogliamo definire una variabile con un tipo che il compilatore deduce da un espressione ma non vogliamo assegnare il valore dell'espressione alla variabile.

```
sometype f() { ... }
decltype(f()) sum = 0;
vector<Point> v;
...
for (decltype(v.size()) i = 0; i != v.size(); ++i) {
...
}
```
f() and v.size() are not evaluated.

**Array su stack**

```
void f() {
int a = 5;
int x[3]; // size must be a compile-time constant
for (size_t i = 0; i != 3; ++i) {
x[i] = (i + 1) * (i + 1);
}
...
}
```

**Array su heap**

```
void g(size_t n) {
int a;
int* px = new int[n]; // size may be dynamic, >= 0
for (size_t i = 0; i != n; ++i) {
px[i] = (i + 1) * (i + 1);
}
...
delete[] px; // note []
}
```

Ad un heap-allocated array si accede con puntatori, un heap-alloc array non contiene informazioni sulla sua lunghezza. Le funzioni iteratore begin() e end() non possono essere utilizzate su heap-alloc array. Delete[] è necessaria per cancellare l'array altrimenti gli oggetti nell'array non verranno distrutti.

**Passaggio di parametri**

I paramentri di una funzione sono passati per valore di default, se si vuole cambiare il valore del parametro reale si può passare l'argomento per riferimento.

```
void swap(int& v1, int& v2) {
int tmp = v2;
v2 = v1;
v1 = tmp;
}
int main() {
int i = 10;
int j = 20;
swap(i, j);
}
```

Il passaggio per riferimento viene usato anche per passare oggetti di grandi dimensioni senza farne una copia. La parola chiave const è essenziale se non si vuole modificare l'oggetto.

**Iteratori**
Un iteratore punta ad uno degli elementi di una collection, si può usare * per la deferenziazione dell'elemento al quale punta. Può essere incrementato con ++ per farlo puntare all'elemento successivo.

```
vector<int> v;
...
for (vector<int>::iterator it = v.begin();
it != v.end(); ++it)
*it = 0;
```

Meglio con auto (C++11)

```
for (auto it = v.begin(); it != v.end(); ++it)
*it = 0;
```

Ancor meglio con un for range-based (C++11)

```
for (int& e : v)
e = 0;
```

**Copia di oggetti: inizializzazione e assegnamento**
Inizializzazione: un nuovo oggetto viene inizializzato con una copia di un altro oggetto. Questo viene gestito dal costruttore di copia della classe.

```
Person p1 = ("bob");
Person p3 = p1;
Person p4("Alice");
```

```
Classname(const Classname&)
```

Assegnamento: Un oggetto esistente viene sovrascritto con una copia di un altro oggetto. Questo viene gestito dall'operatore di assegnamento della classe.

```
P4 = p1;
```

```
Classname& operator= (const Classname&)
```

**Funzioni copia**
In ogni classe è possibile implementare il costruttore di copia e l'operatore di assegnamento in modo da avere il comportamento desiderato. Quando non presenti il compilatore ne sintetizza uno che effettua la copia membro a membro. Non è necessario scrivere costruttori di copoia e operatori di assegnamento se la classe non usa risorse dinamiche.

**Costruttore di spostamento**
Se siamo certi che l'oggetto dal quale copiamo non verrà più usato, dovremmo scrivere il costruttore di copia nel seguente modo, detto costruttore di spostamento:

```
String(String& rhs) : chars(rhs.chars) {
rhs.chars = nullptr;
}
```

I valori temporanei possono essere spostati perché distrutti dopo l’uso, il compilatore riconosce i valori temporanei:

```
String s1("abc");
String s2("def");
String s3 = s1 + s2; // the result of ’+’ is a temporary value
void f(String s);
f("abcd"); // => f(String("abcd")), the argument is a
temporary
String g() {
...
return ...; // the return value is a temporary
}
```

**lvalue e rvalue (C++11)**
Un lvalue è persistente (variabili) mentre un ralue non lo è

```
String s1("abc");
String s2("def");
String& sref = s1; // reference bound to a variable
String&& srr = s1 + s2; // rvalue reference bound to a temporary
```

Ora che si ha un riferimento ad un rvalue si può scrivere il costruttore di spostamento

```
String(String&& rhs) noexcept : chars(rhs.chars) {
rhs.chars = nullptr;
}
```

noexcept (C++11) dice al compilatore che il costruttore non lancierà nessuna eccezione.
Una classe che può essere spostata deve avere anche un operatore di assegnamento per movimento.

```
String& operator=(String&& rhs) noexcept {
if (&rhs == this) {
return *this;
}
delete[] chars;
chars = rhs.chars;
rhs.chars = nullptr;
}
```

**Idiomi di costruzione**
Quando una classe gestisce risorse dinamiche deve avere
– Un distruttore
– Un costruttore di copia
– Un operatore di assegnamento
– Un costruttore di spostamento
– Un operatore di assegnamento per movimento
Il costruttore deve inizializzare le componenti della classe
Il costruttore di copia deve fare una copia profonda delle componenti
L'operatore di assegnamento deve rilasciare le vecchie risorse allocate
Il distruttore rilascia tutte le risorse

**Allocazione dinamica di memoria**
Allocazione di variabili
```
int *pnValue = new int; // dynamically allocate an integer
*pnValue = 7; // assign 7 to this integer
```

Quando allochiamo dinamicamente dobbiamo deallocare
```
delete pnValue; // unallocate memory assigned to pnValue
pnValue = 0;
```
L'operatore delete dealloca l'area puntata dal puntatore ma non il puntatore stesso

Allocazione dinamica di vettori
```
int nSize = 12;
int *pnArray = new int[nSize]; // nSize does not need to be constant!
pnArray[4] = 7;
delete[] pnArray;
```

**Memory leak**
La memoria allocata dinamicamente non ha una visibilità effettiva: resta allocata fin quando non viene esplicitamente deallocata, però i puntatori per accedere a tale memoria hanno regole di visibilità.

```
void doSomething(){
int *pnValue = new int;
}
```

La precedente funzione alloca un intero dinamicamente ma non lo dealloca mai. Quando la funzione termina, pnValue non è più visibile (è una normale variabile). Poichè pnValue è l'unica variabile che conosceva l'indirizzo dell'intero allocato dinamicamente, quando pvValue viene distrtutta non ci sarà più nessun modo per far riferimento a quell'area di memoria allocata dinamicamente. Un memory leak si verifica anche quando un puntatore ad un area di memoria allocata dinamicamente viene riassegnato.

**Passaggio di parametri**
Un modo per passaer un parametro per valore evitando di modificarlo nella funzione ma senza farne una copia è usando il passaggio per const reference.
```
void foo(const int &x){
x = 6; // x is a const reference and can not be changed!
}
```
E' possibile passare i parametri anche per indirizzo:
```
void foo(int *pValue){
*pValue = 6;
}
int main(){
int nValue = 5;
}
```
Il passaggio per riferimento ha una sintassi più chiara, i riferimenti sono più sicuri e facili da usare. In termini di efficienza sono la stessa cosa. Quando si passa per indirizzo in realtà l'indirizzo è passato per valore.

## 3. STRUTTURE DI DATI LINEARI

Consideriamo le strutture di dati che si sviluppano in una dimensione e possono essere considerate come sequenze di oggetti, cioè le strutture lineari di dati. In esse è importante l'esistenza di una relazione d'ordine tra gli oggetti che ci aiuta a individuarli e selezionarli.
Per distinguerle si considerano:
a) i modi di individuare la posizione in cui operare nella sequenza (modo di accesso)
b) i modi di agire nella posizione individuata.

Modi di accesso:
- accesso diretto (vettore)
- accesso per scansione (lista)
- accesso agli estremi (pila e coda)

Modi di agire nella posizione:
- lettura del valore di un componente (ispezione)
- aggiornamento del valore (cambio di valore)
- inserimento di un nuovo componente (scrittura)
- rimozione di un componente (cancellazione)

### LISTE

Una lista è una sequenza finita, anche vuota, di elementi dello stesso tipo. A differenza dell'insieme, nella lista un elemento può comparire più volte in posizioni diverse.
Gli elementi della lista sono definiti atomi o nodi.
Si può accedere direttametne solo al primo elemento della sequenza, per accedere agli altri elementi è necessario scandire sequenzialmente gli elementi che lo precedono. E' possibile inserire, cancellare e modificare gli elementi. Poichè la lista è a dimensione variabile, queste opearzioni che alterano la dimensione sono fondamentali, mentre nei vettori non sono concesse. La lista dunque è una struttura dati dinamica.

### Specifica sintattica

Tipi: lista, posizione, boolean, tipoelem
Operatori:
– crealista : () → lista
- listavuota : ( lista) → boolean
– leggilista : (posizione, lista) → tipoelem
– scrivilista : (tipoelem,posizione,lista) → lista
– primolista : (lista) → posizione
– finelista : (posizione, lista) → boolean
– succlista : (posizione, lista) → posizione
– predlista : (posizione, lista) → posizione
– inslista (tipoelem,posizione,lista) → lista
– canclista : (posizione, lista) → lista

**specifica semantica**

Tipi:
- Lista: insieme delle sequenze l = <a1,a2,…,an>, n >= 0 di elementi di tipo tipoelem dove l'i-esimo elemento ha valore ai e posizione pos(i)
- boolean: insieme dei valori di verità

Operatori:
- creaLista() = l'
    POST: l' = <> (seq. Vuota)

- listaVuota(l) = b
    POST: b = true se l = <>
        b = false altrimenti

- leggiLista(p,l) = a
    PRE: p = pos(i) con 1 <= i <= n
    POST: a = a(i)

- scriviLista(a,p,l) = l'
    PRE: p = pos(i) 1 <= i <= n
    POST: l' = <a1,a2,…,ai-1,a,ai+1,…,an>

- primoLista(l) = p
    POST: p = pos(1)

- fineLista(p,l) = b
    PRE: p = pos(i) 1 <= i <= n+1
    POST: b = true se p = pos(n+1)
        b = false altrimenti

- succLista(p,l) = q
    PRE: p = pos(i) 1 <= i <= n
    POST: q = pos(i+1)

- predLista(p,l) = q
    PRE: p = pos(i) 2 <= i <= n
    POST: q = pos(i-1)

- insLista(a,p,l) = l'
    PRE: p = pos(i) 1 <= i <= n+1
    POST: se 1 <= i <= n : l' = <a1,…,ai-1,a,ai,ai+1,…,an>
        se i = n+1 : l' = <a1,a2,…,an,a>
        se i = 1 e l=<>: l' = <a>

- cancLista(p,l) = l'
    PRE: p = pos(i) 1 <= i <= n
    POST: l' = <a1,…,ai-1,ai+1,…,an>

Dalle specifiche emerge che per accedere ad un elemento occorre conoscerne la posizione. Questo nelle liste è possibile solo per il primo elemento. L'unico operatore che dà come risultato la

posizione senza altre informazioni è primoLista. Per gli altri elementi la posizione si ottiene coonoscendo a priori la posizione dell'elemento precedente o successivo e applicando succLista o precLista. Dunque per accedere ad un generico elemento occorre scandire la lista a partire dal primo elemento.

**Realizzazione sequenziale con vettore**
Una lista può essere rappresentata usando un vettore, poiché il numero di elementi che compongono la lista può variare, si utilizza una variabile primo per il valore dell'indice del vettore in cui è memorizzato il primo elemento della lista. Si utilizza un altra variabile lunghezza per indicare il numero di elementi di cui è composta la lista. Questa rappresentazione consente di realizzare alcune delle operazioni definite per la lista. Il vero problema riguarda l'inserzione e la rimozione di componenti in quanto l'inserimento di un nuovo elemento nella lista comporterebbe lo spostamento verso il basso di tutti gli elementi successivi.

**Rappresentazione collegata**
L'idea fondamentale della rappresentazione collegata è quella di memorizzare gli elementi di una lista associando ad essi una particolare informazione (riferimento) che permette di individuare la locazione in cui è memorizzato l'elemento successivo.

**Realizzazione con cursori**
Viene utilizzato sempre un vettore per l'implementazione della lista ma si riesce a superare, attraverso i riferimenti il problema dell'aggiornamento (inserimento o cancellazione di un elemento) I riferimenti si realizzano tramite cursori, cioè variabili intere o enumerative il cui valore è interpretato come indice di un vettore. Si definisce un vettore spazio che contiene tutte le liste, ognuna individuata da un cursore iniziale, contiene tutte le celle libere organizzate in una lista detta “listalibera”.

**Realizzazione con puntatori**
Un altra possibile realizzazione di lista è quella mediante l'uso congiunto del tipo puntatore e del tipo record. Questa è sicuramente la più efficace realizzazione della rappresentazione collegata. Il tipo puntatore è un tipo di dato i cui valori rappresentano indirizzi di locazioni di memoria. Le operazioni disponibili su una variabile puntatore p sono:
- l'accesso alla locazione il cui indirizzo è memorizzato in p
- la richiesta di una nuova locazione di memoria e la memorizzazione dell'indirizzo in (new)
- il rilascio della locazoine di memoria il cui indirizzo è memorizzato in p (delete)

## 4. PILA

**Specifica sintattica**
Una pila è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi solo da un estremo (testa). Può essere visto come un caso speciale di lista in cui l'ultimo elemento inserito è il primo ad essere rimosso, non è possibile accedere a qualsiasi altro elemento che non sia quello in testa

tipi: pila, boolean, tipoelem
Operatori:
– creapila: ( ) → pila
– pilavuota: (pila) → boolean
– leggipila: (pila) → tipoelem
– fuoripila: (pila) → pila
– inpila: (tipoelem, pila) → pila

**Specifica semantica**

Tipi:
pila = insieme delle sequenze P = <a1,…,an> con n >= 0 di elementi di tipo tipoelem gestiti con accesso LIFO

boolean = insieme dei valori di verità
Operatori:

- creaPila = p
  POST: p = <>

- pilaVuota(p) = b
  POST: b = vero se p = <>, b = falso atrimenti

- leggiPila(p) = a (top)
  PRE: p = <a1,…,an> n >=1
  POST: a = a1

-fuoriPila(p) = p' (pop)
  PRE: p = <a1,…,an> n >=1
  POST p'=<a2,…,an> se n>1, p'=<> se n = 1

- inPila(a,p)=p' (push)
  PRE: p = <a1,…,an> n >=0
  POST: p' = <a,a1,…,an>

**Realizzazioni**
La pila è un caso particolare di lista e ogni realizzazione descritta per la lista funziona anche per la pila. Possiamo definire la corrispondenza tra gli operatori:

creapila() → crealista()
pilavuota(p) → listavuota(p)
leggipila(p) → leggilista(primolista(p),p)
fuoripila(p) → canclista(primolista(p),p)
inpila(a,p) → inslista(a,primolista(p),p)

**Realizzazione con vettore**
Con questa realizzazione ogni operatore richiede tempo costante per essere eseguito. La rappresentazione mediante array presenta due svantaggi: richiede di determinare a priori un limite al numero massimo di elementi della pila, lo spazio di memoria utilizzato è indipendente dal numero effettivo di elementi.
Gli elementi vanno memorizzati in ordine inverso, mantenendo un cursore alla testa della pila (l'ultimo elemento inserito in fondo al vettore). Per contro gli inserimenti e le cancellazioni non richiedono spostamenti perché effettuati ad una estremità dell'array.

**Realizzazione con puntatori**
Ci riferiamo alla pila con un puntatore alla cella che si trova in cima.

**Pile e ricorsione**

Una delle applicazioni per le pile riguarda l'esecuzione di programmi ricorsivi. L'esecuzione di una procedura ricorsiva prevede il salvataggio dei dati su cui lavora la procedura al momento della chiamata ricorsiva. Le diverse chiamate attive vengono organizzate su una pila, quella più recente è quella che si conclude prima. Nella pila vanno salvati i parametri e il punto di ritorno. Grazie alle pile è sempre possibile dato un programma ricorsivo, trasformarlo in iterativo.

La ricorsione può anche essere usata per formalizzare un ampia classe di strutture dati che mostrano caratteristiche ricorsive nella struttura.
- aggregato di dati eventualmente vuoto: livello assiomatico
- aggregate non vuoto in cui è idividuato un primo elemento che insieme ai successivi costituisce una sequenza.

Se indichiamo la sequenza di n elementi con sn = a1,a2,…,an
allora sn sarà:
- per n = 0, s0={} livello assiomativo
- per n > 0 sn={sn-1,an}

## 5. CODE

Una coda è un tipo astratto che consente di rappresentare una sequenza di elementi in cui è possibile aggiungere elementi da un estremo e togliere elementi dall'altro estremo.

**Specifica sintattica**
Tipi: coda, boolean, tipoelem
Operatori:
– creacoda: () → coda
– codavuota: (coda) → boolean
– leggicoda: (coda) → tipoelem
– fuoricoda: (coda) → coda
– incoda: (tipoelem, coda) → coda

**Specifica semantica**
Tipi:
- coda: insieme delle sequenzi q = <a1,a2,…,an> n >= 0 di elementi di tipo tipoelem caratterizzata dall'accesso fifo.
- boolean: insieme dei valori di verità

Operatori:
- creacoda = q'
    POST: q' = <>

- codavuota(q) = b
    POST: b = vero se q = <> altrimenti b = falso

- leggicoda(q) = a
    PRE: q = <a1,a2,…,an> con n >= 1
    POST: a = a1

- fuoricoda(q) = q'
    PRE: q = <a1,a2,…,an> con n >= 1
    POST: q' = <a2,…,an> se n >1, q' = <> se n = 1
- incoda(a,q) = q'

PRE: q = <a1,…,an> con n >= 0
POST: q' = <a1,…,an,a>

**Rappresentazione**
In generale le possibili rappresentazioni delle code sono analoghe a quelle delle pile con l'attenzione di consentire l'accesso sia al primo elemento inserito (per tirarlo fuori) sia all'ultimo inserito (per inserirne un altro in coda).

**Realizzazione con vettore circolare**
Per le code la rappresentazione sequenziale non è agevole come per le pile, è utile gestire l'array in modo circolare. Il vettore è inteso come un array di maxlung elementi con indice da a 0 a maxlung -1, in cui consideriamo l'elemento di indice 0 come successore a quello di indice maxlung-1.
Il valore di PRIMO indica la posizione nell'array del primo elemento memorizzato, ULTIMO si riferisce alla posizione dell'ultimo elemento inserito.

## 6. INSIEMI

Un insieme è una collezione di elementi di tipo omogeneo. A differenza delle liste gli elementi non sono caratterizzati da una posizione ne possono apparire piu' di una volta.
Il numero di elementi detta cardinalità rappresenta la dimensione dell'insieme.
La relazione fondamentale è quella di appartenenza da cui deriva l'inclusione.

**Specifica sintattica**
Tipi
insieme , boolean , tipoelem

Operatori
– Creainsieme: ( ) → insieme
– Insiemevuoto: ( insieme ) → boolean
– Appartiene: (tipoelem , insieme ) → boolean
– Inserisci: (tipoelem , insieme ) → insieme
– Cancella: (tipoelem , insieme ) → insieme
– Unione: ( insieme , insieme ) → insieme
– Intersezione: ( insieme , insieme ) → insieme
– Differenza: ( insieme , insieme ) → insieme

**Specifica semantica**
Tipi
Insieme: famiglia di insiemi costituita da elementi di tipo tipoelem
boolean: insieme vaolri verità

Operatori:
- creaInsieme = A
POST: A = {}

- insiemeVuoto(A) = b
POST: b = vero se A = {} altrimenti b = falso

- appartiene(x,A) = b
POST: b = vero se x app. A, b = falso altrimenti
-inserisci(x,A) = A'

PRE: x non app. A
POST: A' = A U {x}  se x app A: A' = A

-cancella (x,A) = A'
PRE: x app A
POST: A' = A \ {x}  se x non app A: A'=A

- unione(A,B) = C
POST: C = A U B

- intersezione(A,B) = C
POST: C = A ^ B

- differenza(A,B) = C
POST: C = A \ B

**Realizzazioni**

**Rappresentazione con vettore booleano**

Per linguaggi che non dispongono del tipo insieme è possibile rappresentare un insieme A i cui elementi sono ad esempio interi in [1,n] attraverso un vettore booleano di bit, il cui k-esimo valore sarà vero se k app A e falso altrimenti (vettore caratteristico).
Un altra possibile rappresentazione si avvale di una lista i cui elementi sono quelli dell'insieme.

**Rappresentazione con liste non ordinate**

L'occupazione di memoria è proporzionale al numero di elementi presenti nell'insieme.
L'inserimento avviene in testa alla lista semplice con cui è realizzato l'insieme.

**Rappresentazione con liste ordinate**

Se è definita una relazione <= di ordinamento totale sugli elementi dell'insieme, esso può essere rappresentato con una lista ordinata per valori crescenti degli elementi utilizzando due puntatori che scorrono ognuno su un insieme.

**Mfset**

Un mfset (merge-find-set) è una partizione di un insieme finito in sottoinsiemi disgiunti detti componenti. L'MFSet (Merge-Find Set), altrimenti conosciuto come struttura dati union-find, è una struttura dati derivante dal concetto di Insieme delle parti, per cui dato un insieme finito di elementi a volte risulta utile partizionarli in insiemi disgiunti.
Le operazioni consentite permettono di:
- stabilire a quale componente appartiene un elemento generico
- unire due componenti distinte in una sola componente lasciando inalterate le componenti rimanenti

**Specifica sintattica**

Tipi: insieme, boolean, tipo elem, mfset, componente
Operatori:
creamfset: (insieme) → mfset
fondi: (tipoelem,tipoelem,mfset ) → mfset
trova: (tipoelem,mfset ) → componente

**Specifica semantica**

Tipi
- insieme: famiglia di insiemi costituita da elementi di tipo tipoelem
- mfset: famiglia di partizioni di insiemi di elementi di tipo tipoelem
- componente: sottoinsieme di insieme che è elemento di mfset

Operatori
- creamfset(A) = S
    POST: S è una famiglia di n = |A| componenti c1,c2,…,cm ognuno delle quali contiene uno e un solo elemento di A e tali che U ci = A con 1<= i <= n

- fondi (x,y,S) = S'
    PRE: x e y appartengono a componenti distinte cx e cy di S
    POST: S' è costituito da tutte le componenti che non contengono x e y e da una nuova componente ottenuta dall'unione delle due compnenti cx e xy

-trova (x,S) = c
    PRE: x appartiene ad una componente di S
    POST: c è l'identificatore della componente a cui x appartiene

altra rappresentazione di trova

trova: (tipoelem, tipoelem, mfset) → boolean
- trova(x,y,S) = b
    PRE: x e y appartengono a componenti di S
    POST: b = vero se x e y app alla stessa componente, falso altrimenti.

Gli mfset si realizzano attraverso strutture ad albero.

## 7. DIZIONARI

Esistono delle applicazioni che pur richiedendo una struttura dati del tipo insieme non richiedono tutte le operazioni definite sull'insieme. Questo sottotipi del tipo insieme sono i dizionari. Gli elementi sono generalmente tipi strutturati ai quali si accede per mezzo di un riferimento a un campo chiave. Gli elementi assumono la forma di una coppia costituita da (chiave, valore).

La caratteristica della chiave è legata all'applicazione (in un dizionario di trattamento testi la chiave individua una parola mentre in un magazzino la chiave può essere il codice del pezzo)
Il valore associato invece rappresenta l'informazione associata per scopi di gestione.

**Operazioni**
Le operazioni applicate ad un dizionario devono consentire la verifica dell'esistenza di una definita chiave e deve essere possibile l'inserimento di nuove coppia chiave valore come pure la cancellazione. Può essere utile anche il recupero delle informazioni presenti nell'attributo oppure la loro eventuale modifica. Poichè possiamo definirli come un caso particolare di insieme, la specifica per i dizionari è identica a quella del tipo di dato insieme.
Le operazioni ammesse sono: crea, appartiene, inserisci, cancella.
In alcuni casi ritroviamo anche: recupera, aggiorna.

**Specifica sintattica**

Tipi
dizionario, boolean, chiave, valore
Operatori
– creadizionario: ( ) → dizionario
– dizionariovuoto: (dizionario) → boolean
– appartiene: (chiave, dizionario) → boolean
– inserisci: (<chiave, valore>, dizionario ) → dizionario
– cancella: (chiave, dizionario) → dizionario
– recupera: (chiave, dizionario) → valore

**Specifica semantica**
Tipi:
Dizionario: famiglia di dizionari costituita da coppie di tipo <chiave, valore>
boolean: insieme valori verità

Operatori:
- creaDizionario: D
    POST: D = {}

- dizionarioVuoto (D) = b
    POST: b = vero se D = {}, altrimenti falso

- appartiene(k,D) = b
    POST: b = vero se esiste una coppia <k'.v> app D tale che k' = k, b = falso altrimenti

- inserisci (<k,v>, D) = D'
    POST: D' = D U {<k,v>} se non esiste una coppia <k',v'> tale che k' = k
        D' = D \ {<k',v'>} U {<k,v>} se esiste già una coppia <k',v'> tc k' = k

- cancella(k,D) = D'
    PRE: esiste una coppia <k',v'> tc k' = k
    POST: D' = D \ {<k',v'>}

- recupera(k,D) = v
    PRE: esiste una coppia <k',v'> app D tc k = k'
    POST: v = v'

**Rappresentazioni**
Oltre alle realizzazioni viste per l'insieme che si rifanno alla rappresentazione con vettore booleano (vettore caratteristico) e alla rappresetanzione mediante lista, ci sono realizzazioni più efficienti mediante vettori ordinati e tabelle hash.

**Rappresentazione con vettore ordinato**
Si utilizza un vettore con un cursore all'ultima posizione occupata. Avendo definito una relazione di ordine totale <= sulle chaivi, queste si memorizzano nel vettore in posizioni contigue e in ordine crescente a partire dalla prima posizione. Per verificare l'appartenenza di un elemento o chiave k, si utilizza la ricerca binaria, cioè si confronta il valore da ricercare k con il valore v che occupa la posizione centrale e si stabilisce in quale metà continuare la ricerca.

**Rappresentazione con tabella hash**
Esiste una tecnica denominata hash che si appoggia su una struttura dati tabellare che si presta ad essere usata per realizzare dizionari. Con questa struttura le operazioni di ricerca e modifica di un

dizionario possono operare in tempi costatnti e indipendenti sia dalla dimensione del dizionario che dall'insieme dei valori che verranno gestiti.
- idea base: ricavare la posizione che la chiave occupa in un vettore dal valore della chiave.
- esistono diverse varianti che comunque si possono far risalire ad una forma statica e ad una forma dinamica o estensibile.
- la prima fa uso di strutture o tabelle di dimensione prefissata, mentre l'hash dinamico è in grado di modificare dinamicamente le dimensioni della tabella hash sulla base del numero di elementi che vengono via via inseriti o eliminati. Faremo riferimento alla forma statica.

L'hash statico può assumere a sua volta due forme diverse:
- hash chiuso: consente di inserire un insieme limitato di valori in uno spazio a dimensione fissa
- hash aperto: consente di memorizzare un insieme di valori di dimensione qualsiasi in uno spazio potenzialmente illimitato

Entrambe queste varianti però utilizzano una sottostante tabella hash a dimensione fissa costituita da una struttura allocata sequenzialmente in memoria che assume la forma di un array.

- Nel caso di hash chiuso la struttura sarà composta da un certo numero (maxbucket) di contenitore di uguale dimensione denominati bucket. Ognuno di questi contenitori può mantenere al proprio interno al massimo un numbero nb = 1 di elementi che comprenderanno la chiave e il corrispondente valore (nel caso nb = 1 ogni bucket avrà una sola coppia (chiave, valore))

- Nel caso di hash aperto la struttura sarà composta da un certo numero indeterminato di contenitori bucket.

- In ambedue i casi viene usata una funzione aritmentica allo scopo di calcolare, partendo dalla chiave, la posizione in tabella delle informazioni contenute nell'attributo collegato alla chiave.

La rappresentazione hash è un alternativa efficace all'indirizzamento diretto in un vettore perché la dimensione è proporzionale al numero di chiavi attese. Se k è l'insieme di tutte le possibili chiavi distinte e v il è il vettore di dimensione m in cui si memorizza il dizionario, la soluzione ideale è la funzione di accesso h: K → {1,…,m} ,che permetta di ricavare la posizione h(k) della chiave k nel vettore v cosi che, se k1 app K e k2 app K, k1 != k2 si ha h(k1) != h(k2)

Utilizzando m = |K| si ha garanzia di biunivocità e di poter accedere direttamente alla posizione contenente la chiave. Se |K| è grande, si ha spreco enorme di memoria. La dimendione m del vettore va scelta in base al numero di chiavi attese. La soluzione di compromesso è scegliere un m > 1 ma molto minore di |K|

**Collisioni**
Una collisione si verifica quando chiavi diverse producono lo stesso risultato della funzione. Esistono funzioni hash più o meno buone anche se le collisioni non si potranno evitare del tutto.

- Gestione con scansione lineare: Se h(k) per qualche chiave k indica una posizione già occupata, si ispeziona la posizione successiva nel vettore. Se la posizione è piena, si prova con la seguente fino a trovare una posizione libera o trovare che la tabella è completamente piena.
Una posizione libera può venire facilmente segnalata in fase di realizzazione da una chiave fittizia “libero”. Per la cancellazione è più semplice sostituire l'oggetto cancellato con una chiave fittizia “cancellato” che dovrebbe essere facilmente distinguibile da altre chiavi reali e dalla chiave “libero”. La strategia lineare può produrre nel tempo il casuale addensamento di informazioni in certi tratti della tabella (agglomerati) piuttosto che una loro dispersione.
Quale che sia la funzione hash adottata, deve essere prevista una strategia per gestire il problema

degli agglomerati e delle collisioni. In definitiva:
- occorre una funzione hash, calcolabile velocemente e che distribuisca le chiavi uniformemente in v, in modo da ridurre le collisioni.
- occorre un metodo di scansione per la soluzione delle collisioni utile a reperire chiavi che hanno trovato la posizione occupata e che non provochi la formazione di agglomerati di chiavi.
- la dimensione m del vettore v deve essere una sovrastima del numero delle chiavi attese, per evitare di riempire v completamente.

Per definire funzioni hash è conveniente considerare la rappresentazione binaria dei caratteri e bin(k) è data dalla loro concatenazione.

**Generazione hash**
- Denotiamo con int(b) il numero intero rappresentato da una stringa binaria b. Indichiamo da 0 a m-1 gli elementi di v.

- 1) h(k) = int(b), dove b è un sottoinsieme di p bit di bin(k), solitamente estratti nelle posizioni centrali.
- 2) h(k) = int(b), dove b è dato dalla somma modulo 2 effettuata bit a bit di diversi sottoinsiemi di p bit di bin(k)
- 3) h(k) = int(b), dove b è un sottoinsieme di p bit estratti dalle posizioni centrali di bin(int(bin(k))2)
- h(k) è uguale al resto della divisione int(bin(k)) \ m (m è dispari; se fosse uguale a 2p, due numeri con gli stessi p bit finali darebbero sempre luogo ad una collisione)

L'ultima funzione hash proposta è la migliore dal punto di vista probabilistico e fornisce un eccellente distribuzione degli indirizzi h(k) nell'intervallo [0, m-1]

Esempio

Funzione hash: si supponga che le chiavi siano di 6 caratteri alfanumerici (chiavi più lunghe sono troncate, mentre chiavi più corte sono espanse a destra con spazi). Si assuma ord(a)=1, ord(b)=2, ... , ord(z)=26 e ord(b)=32, dove b indica lo spazio, con rappresentazione di ogni ordinale su 6 bit.
Per le chiavi weber e webern si ottiene:
– bin(weberb) = 010111 000101 000010 000101 010010 100000 0000
– bin(webern) = 010111 000101 000010 000101 010010 001110 0000
dove si sono evidenziati per chiarezza i gruppi di 6 bit che rappresentano ciascun carattere. Calcoliamo gli indirizzi hash ottenuti con le funzioni definite di seguito in (a), (b) e (d):
–
(a) sia m = 256 = 28. Estraendo gli 8 bit dalla posizione 15 alla 22 di bin(k), h(weber) = h(webern) = int(00100001) = 33. Le due chiavi danno pertanto luogo ad una collisione
–
(b) sia ancora m = 256 = 2 8 e si calcoli b raggruppando bin(k) in cinque gruppi di 8 bit (dopo aver espanso a destra bin(k) con quattro zeri). h(weber) = int(11000011) = 195, poiché
●
–
11000011 = 01011100 + 01010000 + 10000101 + 01001010 + 00000000
dove + indica la somma bit a bit modulo 2. Analogamente,

h(webern) = int(00100001) = 33, poiché

- 00100001 = 01011100 + 01010000 + 10000101 + 01001000 + 11100000
– benché h(webern) sia uguale a 33, come nel caso (a), la collisione é stata eliminata.
– (d) sia m = 383. utilizzando 6 bit, int(bin(k)) è un numero in base 2 6 = 64.
– int(bin(webern))= 23 * 645 + 5 * 644 + 2 * 643 + 5 * 642 + 18 * 641 + 14 * 640= 64(64(64(64(23*64)+5)+2)+5)+18)+14
– h(webern) e h(weber) sono ottenuti prendendo il resto della divisione int(bin(.....))/383

**Hash aperto**
Una tecnica che evita la formazione di agglomerati è quella dell'hash aperto che richiede che la tabella hash mantenga la lista degli elementi le cui chiavi producono lo stesso valore di funzione (trasformata).
La tabella hash viene realizzata definendo un array di liste di bucket (liste di trabocco).
La funzione hash viene utilizzata per determinare quale lista potrebbe contenere l'elemento che possiede una determinata chiave in omdo da poter attivare una successiva operazione di ricerca nella lista corrispondente e di restituire la posizione del bucket che contiene la chiave.

**Liste di trabocco**
Il vettore v contiene in ogni posizione un puntatore ad una lista.

**Metodi di scansione**
Nella realizzazione con hash e liste di trabocco si è usato un metodo di scansione esterno contrapposto alla scansione lineare che è un metodo di scansione interno. Altri metodi interni o chiusi sono: scansione quadratica, scansione pseudocasuale, hashing doppio.

**Scansione interna**
Chiamiamo fi la funzione che viene utilizzata l'i-esima volta che si trova occupata una posizione del vettore v, i >= 0, per i = 0, f0 = h

- scansione lineare:
    - fi = (h(k) + h*i) mod m
    - h è un intero positivo primo con m
    – h rappresenta la distanza tra due posizioni successive esaminate nella scansione (se h = 1, scansione a passo unitario)
    - essendo h e m primi tra loro, vengono esaminate tutte le posizioni di v prima di riconsiderare le posizioni già esaminate
    svantaggio: non riduce la formazione di agglomerati

- scansione quadratica:
    - fi = (h(k) + h*i + i(i-1)/2) mod m
    - m è primo
    - la distanza tra due posizioni successive nella sequenza è variabile, quindi la possibilità di agglomerati è ridotta.
    Svantaggio: la sequenza di scansione non include tutte le posizioni di v (svantaggio trascurabile per m non troppo piccolo)

- scansione pseudocasuale:

- fi = (h(k) + pi) mod m
- pi è l'i-esimo numero generato da un generatore di numeri pseudocasuali, che genera gli interi tra 1 e m una sola volta in un ordine qualunque

- hashing doppio
  - fi = (h(k) + i * f(k)) mod m
  - f è un altra funzine hash diversa da h

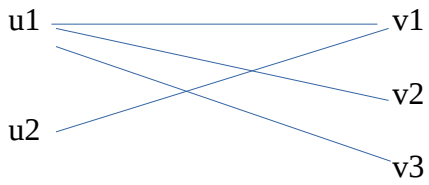
Usando metodi di scansione interna e potendo cancellare chiavi, non si è mai sicuri che raggiunta una posizione vuota nella ricerca di k, tale chiave non si trovi in un altra posizione di v, poiché la posizione ora vuota era occupata quando k è stata inserita.

Bisogna dunque scandire anche le posizioni in cui si è cancellato e fermarsi o sulla posizione mai riempita o dopo essere tornati su una posizione già scandita.
Ciò aumenta il tempo di ricerca, è meglio utilizzare il metodo di scansione esterna se sono previste molte cancellazioni.

## 8 - ALBERI BINARI

### Grafi e Alberi: generalità

Ricordando che una relazione tra due insiemi U e V è un sottoinsieme A di UxV, si può dare una descrizione di A in forma diagrammatica scrivendo tutti gli elementi di U, tutti gli elementi di V e congiungendoli. Questa rappresentazione è detta grafo (bipartito).




Se U = V → A app UxU. In tal caso gli elementi di U vengono scritti una sola volta e viene segnata una freccia sulla congiunzione da ui a uj per quelle coppie <ui, uj> app A. Si parla in questo caso di grafo orientato.
Gli elementi u app U vengono detti nodi o vertici del grafo orientato. Le linee di congiunzione vengono dette archi. Nel caso le coppie <ui, uj> sono congiunte sia dall'arco (i,j) che dall'arco (j,i) si può usare un unica connessione senza freccia. Si parla in questo caso di grafi non orientati, usati per rappresentare relazioni simmetriche tra oggetti.

### Grafi orientati: definizioni

Un grafo orientato G è una coppia <N,A> dove N è un insieme finito non vuoto (di nodi) e A contenuto in NxN è un insieme finito di coppie ordinate di nodi, detti archi. Se <ui, uj> app A nel grafo vi è un arco da ui a uj

Esempio: N = {u1,u2,u3}  A = {(u1,u1),(u1,u2),(u2,u1),(u1,u3)}

**cammino**

In un grafo orientato G, un cammino è una sequenza di nodi u0,u1,…,uk tali che:
(ui,ui+1) app A, per i = 0,1,…,k-1

Il cammino parte dal nodo u0, attraverso i nodi u1,..,uk-1 e arriva al nodo uk. Ha lunghezza uguale a k. Se non ci sono nodi ripetuti il cammino è semplice (ui != uj per 0 =< i =< j =< k). Se u0 = uk il cammino è chiuso. Un cammino sia sempliche che chiuso è un ciclo.

**Grafo completo**
Un grafo è detto completo se per ogni coppia di nodi ui, uj app N esiste un arco che va da ui ad uj. (A = NxN).

**Grafo connesso**
Definiremo grafo connesso un grafo G = <N,A> in cui dati u e v app N esiste un cammino da u a v oppure un cammino da v a u. G è detto fortemento connesso se per ogni coppia di nodi u e v esiste almeno un cammino da u a v ed almeno un cammino da v ad u.

**Grafo non orientato: definizioni**
Definiremo grafo non orientato un grafo G = <N,A> nel quale gli archi sono formati da coppie non ordinate. Mentre in un grafo orientato (ui, uj) e (uj, ui) indicano due archi distinti, in un grafo non orientato indicano lo stesso arco. I nodi congiunti da un arco sono adiacenti. Anche nei grafi non orientati troviamo nozioni analoghe a quelle di cammino (catena) e di ciclo (circuito).

Il grafo è una struttura dati di grande generalità alla quale si possono ricondurre strutture più semplici: le liste possono essere considerate un caso particolare di grafo, come pure gli alberi.

**Alberi**
L'albero èuna struttura informativa fondamentale utile per rappresentare:
- Partizioni successive di un insieme in sottoinsiemi disgiunti
- Organizzazioni gerarchiche di dati
- Procedimenti decisionali enumerativi

Un particolare tipo di grafo è l'albero, definito matematicamente come una coppia T=(N,A)
dove N è un insieme finito di nodi ed A è anche un insieme di coppie non ordinate (albero libero) tali che:
- Il numero di archi è uguale al numero di nodi meno uno |A| = |N| - 1
- T è connesso, ovvero per ogni coppia di nodi u e v in n, esiste una sequenza di nodi distinti u0, u1, …,uk tali che u = u u0, v = uk e la coppia <u1, ui+1> è un arco di A per i = 0,…,k-1.

Un albero radicato è ottenuto da un albero libero designando arbitrariamente un nodo r come radice e ordinando i nodi per livelli. La radice r è a livello 0 e tutti i nodi u, <u,r> app A sono figli di r e si trovano a livello 1 (r è padre). Nodi con lo stesso padre sono fratelli, nodi terminali senza figli sono detti foglie. Un albero ordinato è ottenuto da uno radicato stabilendo un ordinamento tra nodi allo stesso livello.

Se T è un albero ed n e m sono nodi di T si dice che:
- m è discendente di n se n è antenato di m cioè se n = m (antenato proprio) oppure se n è genitore di un antenato di m.
- un nodo è interno se non è foglia.
- una linea di T connette due nodi, uno dei quali è genitore dell'altro.
- un cammino in T è la sequenza di linee che unisce due nodi, uno dei quali è antenato dell'altro: la lunghezza di un cammino è costituita dal numero di linee che lo compongono.
- l'altezza di un nodo è la lunghezza del cammino più lungo da quel nodo ad una foglia.

- la profondità di un nodo è la lunghezza del cammino dalla radice a quel nodo.

Definiamo albero di ordine K un albero in cui ogni nodo ha al massimo k figli.

In un albero valgono le seguenti proprietà:
- Un albero è un grafo aciclico, in cui per ogni nodo c'è un solo arco entrante (tranne che per la radice)
- Un albero è un grafo debolmente connesso
- Se esiste un cammino che va da un nodo u ad un altro nodo v, tale cammino è unico.
- In un albero esiste un solo cammino che va dalla radice a qualunque altro nodo.
- Tutti i nodi di un albero T (tranne r) possono essere ripartiti in insiemi disgiunti ciascuno dei quali individua un albero (dato un nodo u, i suoi discendenti costituiscono un albero detto sottoalbero di radice u)

**La natura ricorsiva degli alberi**
Un albero può essere definito ricorsivamente

- Un albero è un insieme non vuoto di nodi ai quali sono associate delle informazioni.
- Tra i nodi esiste un nodo particolare che è la radice (livello 0)
- Gli altri nodi sono partizionati in sottoinsiemi che sono a loro volta alberi (livelli successivi)

vale a dire, un albero è:
vuoto o costituito da un solo nodo (detto radice)
oppure è una radice cui sono connessi altri alberi

**Alberi binari**
Sono particolari alberi ordinati in cui ogni nodo ha al più due figli e si fa sempre distinzione tra figlio sinistro che viene prima nell'ordinamento e il figlio destro. Un albero binario è un grafo orientato che o è vuoto o è costituito da un solo nodo o è formato da un nodo N detto radice e da due sottoalberi binari che vengono chiamati sottoalbero sinistro e destro

**Specifica sintattica**
TIPI: ALBEROBIN, BOOLEANO, NODO
CREABINALBERO : ( ) → ALBEROBIN
BINALBEROVUOTO : (ALBEROBIN) → BOOLEANO
BINRADICE : (ALBEROBIN) → NODO
BINPADRE : (NODO,ALBEROBIN) → NODO
FIGLIOSINISTRO : (NODO,ALBEROBIN) → NODO
FIGLIODESTRO : (NODO,ALBEROBIN) → NODO
SINISTROVUOTO : (NODO,ALBEROBIN) → BOOLEANO
DESTROVUOTO : (NODO,ALBEROBIN) → BOOLEANO
COSTRBINALBERO : (ALBEROBIN,ALBEROBIN) → ALBEROBIN
CANCSOTTOBINALBERO : (NODO,ALBEROBIN) → ALBEROBIN

**Specifica semantica**
Tipi: AlberoBin: insieme degli alberi binari T=(N,A), nei quali ad ogni nodo è associato un LIVELLO, BOOLEANO, NODO

CreaBinAlbero = T'
POST: T' = (Ins.Vuoto, Ins.Vuoto) = Λ

BinAlberoVuoto(T) = b

POST: b = vero se T = Λ, b = falso altrimenti

BinRadice(T) = u
PRE: T != Λ
POST: u è radice di T e livello(u) = 0

BinPadre(u,T) = v
PRE: T != Λ, u app N, livello(u) > 0
POST: v è padre di u con (v,u) app A e livello(u) = livello(v) + 1

FiglioSinistro(u,T) = v
PRE: T != Λ, u app N, u ha un figlio sinistro
POST: v è figlio sinistro di u in T

FiglioDestro(u,T) = v
PRE: T != Λ, u app N, u ha un figlio destro
POST: v è figlio destro di u in T

SinistroVuoto(u,T) = b
PRE: T != Λ, u app N
POST: b = vero se u non ha figlio sinistro, b = falso altrimenti

DestroVuoto(u,T) = b
PRE: T != Λ, u app N
POST: b = vero se u non ha figlio destro, b = falso altrimenti

CostrBinAlbero(T,T') = T''
POST: T'' si ottiene da T e da T' introducendo automaticamente un nuovo nodo r'' (radice di T'') che avrà come sottoalbero sinistro T e sottoalbero destro T'.
- Se T = Λ e T' = Λ l'operatore inserisce la sola radice r''
- Se T = Λ, r'' non ha figlio sinistro
- Se T'= Λ r'' non ha figlio destro

CancSottoBinAlbero(u,T) = T'
PRE: T != Λ, u app N
POST: T' è ottenuto da T eliminando il sottoalbero di radice u, con tutti i suoi discendenti
Valida per alberi di ogni ordine, agisce potando dal nodo u

Altri due operatori utili:
Specifica sintattica
Tipi: TipoElem tipo dell'etichetta

LeggiNodo :(Nodo, AlberoBin) → TipoElem
ScriviNodo :(TipoElem,Nodo,AlberoBin) → AlberoBin

Specifiche semantiche
LeggiNodo (n,T) = a
PRE: n è un nodo di T, n app N
POST: a è il valore associato al nodo n in T

ScriviNodo (a,n,T) = T'

PRE: n è un nodo di T, n app N
POST: T’ è il nuovo albero corrispondente al vecchio T con il valore a assegnato al nodo n

L’algebra presentata rappresenta una scelta precisa di progetto, si è scelto di enfatizzare la natura ricorsiva degli alberi e di costruire l’albero binario dal basso verso l’alto, cioè dalle foglie verso la radice. Non sempre questa è la scelta migliore, sopratutto se l’albero è usato per rappresentare un processo decisionale è preferibile un algebra che preveda di costruire l’albero dall’alto verso il basso inserendo prima la radice e poi i nodi figli e cosi via.

In tal caso rimangono validi tutti gli operatori tranne quello di costruzione che andrebbe sostituito con 3 nuovi operatori, uno dedicato all’inserimento della radice e gli altri due dedicati all’inserimento di figlio sinistro e destro.

Specifica sintattica

InsBinRadice :(nodo,alberobin) → alberobin
InsFiglioSinistro :(nodo, alberobin) → alberobin
InsFiglioDestro :(nodo,alberobin) → alberobin

Specifica semantica
InsBinRadice(u,T) = T’
PRE: T = Λ
POST: T’ = (N,A), N = {u}, Livello(u) = 0, A = insVuoto

InFiglioSinistro(u,T) = T’
PRE: T != Λ, u app N, SinistroVuoto(u,T) = vero
POST: N’ = N U {v}, T’ è ottenuto da T aggiungendo v come figlio sinistro di u

InFiglioDestro(u,T) = T’
PRE: T != Λ, u app N, DestroVuoto(u,T) = vero
POST: N’ = N U {v}, T’ è ottenuto da T aggiungendo v come figlio destro di u

**Algoritmi di visita**
Sono algoritmi che consentono di analizzare tutti i nodi dell’albero in un ordine definito. Risultano importanti in problemi per i quali, ad esempio, si debba ricercare in quale nodo o a quale livello è contenuto in etichetta un dato valore. Oppure quando si vuole esplorare l’albero per verificarne la profondità.

La visita di un abero consiste nel seguire una rotta di iviaggio che consenta di esaminare ogni nodo dell’albero esattamente una volta.

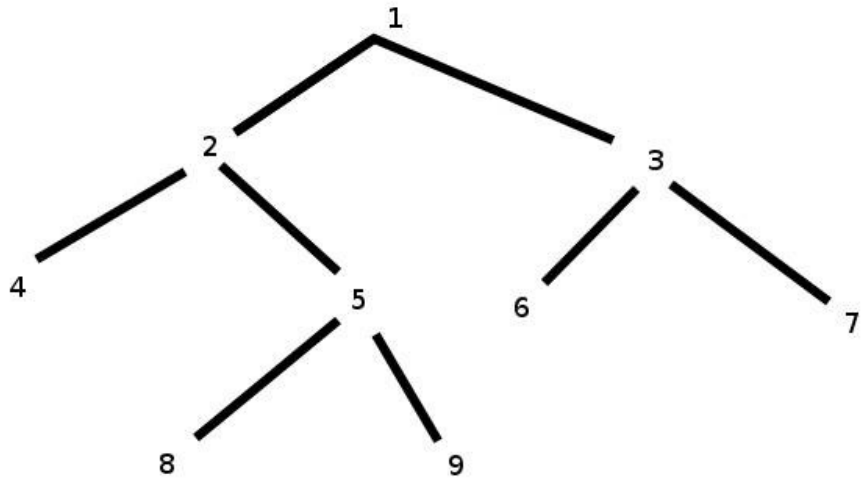
- Visita in pre-ordine: Si applica ad un albero non vuoto e richiede prima l’analisi della radice dell’albero e poi, la visita, effettuata con lo stesso metodo dei due sottoalberi, prima il sinistro, poi il destro.
- Visita in post-ordine: Prima la visita dei sottoalberi, prima il sinistro e poi il destro e in seguito l’analisi della radice dell’albero.
- Visita simmetrica: Richiede prima la visita del sottoalbero sinistro, poi l’analisi della radice e poi la visita del sottoalbero destro

esempio:

Visita in pre-ordine:
1 – 2 – 4 – 5 – 8 – 9 – 3 – 6 – 7

Visita in post-ordine:
4 – 8 – 9 – 5 – 2 – 6 – 7 – 3 – 1

Visita simmetrica: 4 – 2 – 8 – 5 – 9 – 1 – 6 – 3 – 7




Gli algoritmi si possono formulare anche in modo ricorsivo:

Visita in preordine:
Se l'alberto non è vuoto
Allora
1 - Analizza la radice di T
2 – Visita in preordine il sottoalbero sinistro di T
3 – Visita in preordine il sottoalbero destro di T
Fine

Una possibile applicazione è l'albero di analisi (parse tree) per rappresentare espressioni da valutaer cominciando dal basso verso l'alto.

**Rappresentazioni**

**Rappresentazione sequenziale**
Una possibile rappresentazione di un albero binario è quella sequenziale mediante vettore. La radice è in prima posizione, per il generico nodo p memorizzato in posizione i, se esiste il figlio sinistro esso è memorizzato in posizione 2*i, se esiste il figlio destro esso è memorizzato in posizione 2*i+1
Problemi: Alcune componenti del vettore non corrispondono ad alcun nodo dell'albero (in caso l'albero non è completo). Questo potrebbe generare dei problemi perche abbiamo bisogno di un valore con cui esprimere il "non definito".
La soluzione è quella di utilizzare un campo per ogni componente dell'array che varrà vero se è presente un nodo dell'albero e falso altrimenti.
Tuttavia:
- Alberi ninari non completi vengono rappresentati con uno spreco di memoria.
- E' imposto un limite massimo per il numero di nodi nell'albero.
- Le operazioni di aggiunta ed eliminazione di nodi o di sottoalberi comportano diversi spostamenti nell'array.

**Rappresentazione collegata**
Ogni valore T del tipo albero può essere rappresentato con una lista nel modo seguente:

- Se T  vuoto, la lista che lo rappresenta è la lista vuota
- Se T non è vuoto, la lista che lo rappresenta è formata da 3 elementi:
    - Il primo è l'atomo che rappresenta la radice di T
    - Il secondo è una lista che rappresenta, con lo stesso metodo, il sottoalbero sinistro di T
    - Il terzo è un altra lista che rappresenta il sottoalbero destro di T

Un primo metodo richiede di utilizzare un array in modo che ad ogni nodo dell'albero corrisponda una componente dell'array in cui sono memorizzate le informazioni: nodo, riferimento a figlio sin, riferimento a figlio des. Il riferimento è il valore dell'indice in corrispondenza del quale si trova la componente che corrisponde al figlio sin o des. Se il figlio non esiste il riferimento ha valore 0.

Proprio perché l'albero binario può essere visto come una lista, è possibile usare puntatori invece che cursori e la mancanza di un figlio viene indicata col valore nil nell'apposito campo. Prevediamo un campo per il figlio destro, uno per il sinistro e per ragioni di efficienza un campo per il padre.

Problemi da risolvere:
- Numero di nodi per sottoalbero
- Ricerca binaria

## 9 – ALBERI N-ARI

### Specifica sintattica

Tipi:
albero, boolean, nodo

Operatori:
CREAALBERO:( )  albero
ALBEROVUOTO:(albero)  boolean
INSRADICE:(nodo, albero)  albero
RADICE:(albero)  nodo
PADRE:(nodo, albero)  nodo
FOGLIA:(nodo, albero)  boolean
PRIMOFIGLIO:(nodo, albero)  nodo
ULTIMOFRATELLO:(nodo, albero)  boolean
SUCCFRATELLO:(nodo, albero)  nodo
INSPRIMOSOTTOALBERO:(nodo,albero,albero)albero
INSSOTTOALBERO: (nodo, albero, albero)  albero
CANCSOTTOALBERO:(nodo, albero)  albero

### Specifica semantica

Tipi:
albero = insieme degli alberi ordinati T = <N,A> in cui ad ogni nodo n in N è associato il livello(n)

boolean = insieme dei valori di verità
nodo = insieme qualsiasi (non infinito)

Operatori:
CreaAlbero = T'
POST: T' = (ins.Vuoto, ins.Vuoto) = Λ (albero vuoto)

AlberoVuoto(T) = b
POST: b = vero se T = Λ, falso altrimenti

InsRadice(u,T) = T'
PRE: T = Λ
POST: T' = (N,A), N = {u}, livello(u) = 0, A = ins.Vuoto

Radice(T) = u
PRE: T != Λ
POST: u è radice di T, livello(u) = 0

Padre(u,T) = v
PRE: T != Λ , u app N, livello(u) > 0
POST: v è padre di u, <v,u> app A, livello(u) = livello(v)+1

Foglia(u,T) = b
PRE: T != Λ , u app N
POST: b = vero se non esiste un v app N tale che <u,v> app A e livello(v) = livello(u)+1, b falso altrimenti

PrimoFiglio(u,T) = v
PRE: T != Λ , u app N, Foglia(u,T) = falso
POST: <u,v> app A, livello(v) = livello(u)+1
v è primo secondo la relazione d'ordine stabilita tra i figli di u

UltimoFratello(u,T) = b
PRE: T != Λ , u app N
POST: b = vero se non esistono altri fratelli di u che lo seguono nella relazione d'ordine, falso altrimenti

SuccFratello(u,T) = v
PRE: T != Λ , u app N, ultimo fratello(u,T) = falso
POST: v è il fratello di u che lo segue nella relazione d'ordine

InsPrimoSottoAlbero(u,T,T') = T''
PRE: T != Λ , T' != Λ , u app N
POST: T'' è ottenuto da T aggiungendo l'albero T' la cui radice r' è il nuovo PrimoFiglio di u

InsSottoAlbero(u,T,T') = T''
PRE: T != Λ , T' != Λ , u app N
POST: T'' è ottenuto da T aggiungendo l'albero T' la cui radice r' diventa il nuovo fratello che segue u nella relazione d'ordine

CancSottoAlbero(u,T) = T'

PRE: T != Λ , u app N
POST: T’ è ottenuto da T togliendovi il sottoalbero di radice u, cioè u e tutti i suoi discendenti.

**Visita di alberi**
Sia T un albero non vuoto di radice r. Se r non è foglia e ha k > 0 figli, siano T1, T2,…,Tk. Gli ordini di visita sono:
PREVISITA (preordine): Consiste nell’esaminare r e poi nell’ordine, effettuare la previsita di T1,T2,…,Tk
POSTVISITA (postordine): Consiste nel fare prima la postvisita di T1,T2,…,Tk e poi esaminare la radice r.
INVISITA (ord. Simmetrico): Consiste nel fare, nell’ordine la invisita di T1,T2,…,Ti, nell’esaminare r e poi effettuare nell’ordine la invisita di Ti+1,…,Tk per un prefissato i >= 1

**Equivalenza di alberi n-ari e binari**
E’ evidente che si tratta di una equivalenza ai fini della pre-visita. E’ sempre possibile rappresentare un albero n-ario ordinato T con un albero binario B avente gli stessi nodi e la stessa radice: in B ogni nodo ha come figlio sinistro il primo figlio figlio in T è come figlio destro il fratello successivo in T. Si nota che le sequenze di nodi esaminati su T e su B coincidono se T e B sono visitati in previsita.

**Rappresentazioni**
**Rappresentazione con vettore di padri**
Se numeriamo i nodi di T da 1 a n, la più semplice realizzazione sequenziale consiste nell’usare un vettore che contiene per ogni nodo i(1 =< i =< n) il cursore al padre. E’ più facile cosi visitare i nodi lungo percorsi che vanno da foglie a radice. E’ invece piu complesso inserire e cancellare sottoalberi.

**Rappresentazione attraverso liste di figli**
E’ una variante molto usata poiché non si può associare ad ogni elemento un numero di puntatori uguale al massimo dei figli.
Comprende: Il vettore dei nodi in cui oltre alle eventuali etichette si memorizza il riferimento iniziale di una lista associata ad ogni nodo. Queste liste vengono dette liste dei figli. La lista associata al generico nodo i contiene tanti elementi quanti sono i successori di i. Ciascun elemento è il riferimento ad uno dei successori.

**Rappresentazoine mediante lista primofiglio/fratello**
Prevede la gestione di una lista e questo può essere fatto imponendo che tutti gli alberi condividano un area comune e che ogni cella contenga esattamente due cursori: Uno al primofiglio ed uno al fratello successivo, è possibile prevedere anche un cursore al genitore.

**Rappresentazione collegata mediante lista dinamica**
Da un punto di vista formale l’albero n-ario può essere rappresentato mediante liste secondo le seguenti regole:
- se l’albero è vuoto la lista che lo rappresenta è vuota.
- altrimenti l’albero è composto da una radice e da k sottoalberi T1,…,Tk e la lista è fatta da k+1 elementi: il primo rappresenta la radice mentre gli altri sono gli alberi T1,…,Tk.
In generale, la radice dell’albero viene memorizzata nel primo elemento della lista che contiene il riferimento ad una lista di elementi, uno per ogni sottoalbero., ciascuno di questi elementi contiene a sua volta il riferimento iniziale alla lista che rappresenta il corrispondente sottoalbero.
UNA POSSIBILE REALIZZAZIONE PREVEDE RECORD E PUNTATORI, MA IL RECORD VA INTESO CON VARIANTI: PER ESEMPIO, SI PUO’ PREVEDERE UN RECORD CON TRE

CAMPI, UNO PER LA PARTE INFORMAZIONE E DUE PER I PUNTATORI. PER OGNI RECORD SARÀ' SEMPRE SIGNIFICATIVO UNO DEI CAMPI PUNTATORE, MA QUANDO L'ATOMO RAPPRESENTA UN NODO EFFETTIVO DELL'ALBERO SARA' UTILIZZATA L'ETICHETTA E UN PUNTATORE, QUANDO RAPPRESENTA UN ATOMO "DI SERVIZIO" SARANNO UTILIZZATI DUE PUNTATORI.

**Realizzazione di MFSET**
Come è noto un mfset è una partizione di un insieme finito in sottoinsiemi disgiunti detti componenti.
E' possibile rappresentarlo mediante una foresta di alberi radicati.
In cui ciascuno albero rappresenta una componente, le componenti iniziali di mfset sono i nodi attraverso operazioni successive di fondi e trova si crea la struttura.
L'operazione fondi combina due alberi nello stesso albero, si realizza imponendo che una delle due radici diventi nuovo figlio dell'altra. L'operazione trova verifica se due elementi sono nel medesimo albero. Si realizza accedendo ai nodi contenenti gli elementi e risalendo da tali nodi, attraverso i padri, fino ad arrivare alle radici.

## 10. CODE CON PRIORITA'

Sono un caso particolare di insieme, sugli elementi del quale è definita una relazione "=<" di ordinamento totale. E' possibile inserire un nuovo elemento o estrarre l'elemento minimo. La specifica è simile a quella del tipo di dato insieme, le operazioni ammesse sono:
Crea, InserisciMin – secondo gli operatori del tipo insieme, la nuova operazione è cancellamin, che rimuove l'elemento minimo.
Il nome deriva dall'interpretazione che A è una coda di elementi che devon oessere serviti rispettandone la priorità. In generale, le priorità possono intendersi come proprietà associate agli elementi che talvolta possono avere la stessa priorità.

**Specifica sintattica**
Tipi: PioriCoda, TipoElem
Operatori:
CreaPioriCoda :() → PrioriCoda
Inserisci :(TipoElem,PioriCoda)→PrioriCoda
Min :(PioriCoda) → TipoElem
CancellaMin :(PioriCoda) → PioriCoda

**Specifica semantica**
Tipi: PioriCoda: Insieme di code con priorità con elementi di tipo TipoElem

Operatori:
CreaPioriCoda = A
POST: A = ins vuoto

Inserisci(x,A) = A'
POST: A' = A U {x} (se x app A allora A = A')

Min(A) = x
PRE: A != ins vuoto
POST: x app A x < y per ogni y app A, x != y

CancellaMin(A) = A'
PRE: A != ins vuoto

POST: A’ = A – {x} con x = Min(A)

**Rappresentazione con strutture sequenziali**
Si può rappresentare una coda con priorità di n elementi utilizzando strutture sequenziali come liste ordinate e liste non ordinate. Tuttavia poiché la coda con priorità è costituita da un insieme di atomi linearmente ordinati ma senza alcuna relazione strutturale sull’insieme delle posizioni, la sua rappresentazione è associata al modello dell’albero binario.

**Rappresentazione con laberi binari**
Gli elementi di una coda con priorità C possono essere memorizzati nei nodi di un albero binario B che deve avere le seguentu proprietà:

1 – L’albero B è quasi perfettamente bilanciato
- se K è il livello massimo delle foglie, allora B ha esattamente (2^k) – 1 nodi di livello minore di K
- Tutte le foglie di livello K sono addossate a sinistra

2 – L’albero B è parzialmente ordinato
- Ogni nodo contiene un elemento di C che è maggiore di quello del padre

Per realizzare gli operatori si osservi che:
- MIN restituisce il contenuto della radice dell’albero B
- INSERISCI deve inserire una nuova foglia in modo da mantenere verificata la proprietà 1 e quindi far “salire” l’elemento introdotto fino a verificare la proprietà 2
- CANCELLAMIN prevede la cancellazione della foglia di livello massimo più a destra in modo da mantenere verificata la proprietà 1 ed il reinserimento del contenuto della foglia cancellata nell’alnero partendo dalla radice e facendolo scendere in modo che l’albero verifichi anche la proprietà 2

**Osservazione**
INSERISCI e CANCELLAMIN prevedono due fasi, la prima di modifica della struttura dell’albero (condizionata dalla proprietà 1) e la seconda di aggiustamento degli elementi in base alle piorità (proprietà 2). Nella prima fase si deve necessariamenet fare riferimento alla foglia all’estrema destara dell’ultimo livello. Questo evidenza l’utilità di mantenere traccia di tale foglia.

**Analizziamo INSERISCI**
Prima fase (modifica struttura):
- L’albero è vuoto: basta aggiungere l’elemento come radice
- L’albero è costituito dalla sola radice: l’elemento viene aggiunto come figlio sinistro della radice
- L’ultima foglia è un nodo figlio sinistro: l’elemento viene aggiunto come fratello destro
In questa fase abbiamo un primo passo di “salita verso destra” ed uno successivo di “discesa verso sinistra”. Il processo di salite prosegue fintanto che il nodo considerato è un figlio destro (livello completo) oppure non è stata raggiunta la radice. Mentre il processo di discesa viene iterato finquando non si trova una foglia. Nel secondo caso, cioe’ se non si raggiunge alla radice oppure si giunge da un figlio sinistro, si scende partendo dal fratello destro dell’ultimo nodo che stato visitato in salita come figlio destro.

Seconda fase (aggiustamento):
Si parte dalla foglia inserita e si confronta il valore di priorità del nodo figlio con quello del nodo padre, i valori vengono scambiati se non soddisfano la proprietà 2 e quindi si risale l’albero verso la

radice. Questo processo di confronto-scambio si ripete fino a quando non si trovano due elementi che soddisfano già la proprietà 2 oppure non si arriva alla radice.

**Analizziamo CANCELLAMIN**
La prima fase, quella di modifica della struttura, prevede la cancellazione della foglia di livello massimo più a destra, mentre nella seconda fase va risistemato il contenuto.

- L'albero è costituito dalla sola radice: si cancella l'intero albero e non è necessaria la seconda fase.
- Si tratta dell'ultima foglia o di un nodo figlio destro: dopo la cancellazione l'ultima foglia diventa il nodo fratello sinistro.

Caso generale:
Fase di modifica: Si procede attraverso passi di "salita da sinistra" e successivi passi di discesa verso destra" alla ricerca dell'ultima foglia. Il processo di salita prosegue fintanto che il nodo considerato è un figlio sinistro oppure non hè stata raggiunta la radice. Mentre il processo di discesa viene iterato fino a quando non si trova una foglia.

Fase di aggiustamento: L'aggiustamento degli elementi avviene in base alle priorità. Si parte dalla radice e si ricerca la posizione dove inserire il contenuto nella foglia cancellata in modo da soddisfare la proprietà 2. Per ogni nodo esaminato se ne considerano i figli e se il montenuto del nodo cancellato ha piorità minore, si scrive nel nodo attuale il contenuto del figlio con priorità maggiore, valutando se la posizione di tale figlio può accogliere i contenuto del nodo cancellato. Questo processo si ripete fino a quando non si trova una configurazione che soddisfa la proprietà 2 e quindi il contenuto del nodo cancellato ha priorità maggiore di entrambi i figli del nodo attuale.

**Realizzazione con heap**
Gli elementi di B possono essere disposti in un vettore H (heap) nell'ordine in cui si incontrano visitando l'albero per livelli crescenti ed esaminando da sinistra verso destra i nodi allo stesso livello. In tal caso si ha che:
- H[1] è l'elemento contenuto nella radice di B
- H[2i] e H[2i+1] sono gli elementi corrispondenti al figlio sinistro e destro di H[i]
Se B contiene n elementi allora il figlio sinistro (destro) di H[i] non esiste nell'albero se e solo se 2i > n (2i+1 > n). Inoltre per la proprietà 2, se il figlio sinistro (destro) di H[i] esiste, allora H[2i]] > H[i] (H[2i+1] > H[i])

## 11. GRAFI
Il grafo è una struttura composta da nodi e archi che rappresenta una relazione binaria sull'insieme costituito dai nodi. In generale i nodi sono usati per rappresentare oggetti e gli archi per rappresentare relazioni tra coppie di oggetti. Consideriamo di seguito solo grafi orientati o diretti, nei quali gli archi hanno direzione da un certo nodo (di partenza) ad un altro nodo (di arrivo).
In tal caso il grafo G = (N,A) dove n è l'insieme finito dei nodi prevede che a sia un insieme finito di coppie ordinate di nodi, rappresentanti gli archi orientati. Poichè ogni grafo non orientato G' può essere visto come un grafo orientato G, ottenuto da G', sostituendo ogni arco (i,j) con i due archi e (j,i) e (i,j).

**Grafi orientati**
Un grafo orientato G è una coppia <N,A> dove N è un insieme finito non vuoto (insieme di nodi) e A app NxN è un insieme finito di coppie ordinate di nodi, detti archi (o spigoli o linee). Se <ui,uj> app A nel grafo vi è un arco da ui a uj.

**Grafi non orientati**

Definiremo un grafo non orientato un grafo G = <N,A> nel quale gli archi sono formati da coppie non ordinate. Mentre in un grafo orientato (ui,uj) e (uj,ui) indicano due archi distinti, in un grafo non orientato indicano lo stesso arco che incide sui due nodi. I nodi congiunti da un arco sono adiacenti, Anche nei grafi non orientati troviamo nozioni analoghe a quelle di cammino (catena) e di cilo (circuito).

**Specifica sintattica**
Tipi: grafo, boolean, nodo, lista, tipoelem
Operatori:
– crea:() → grafo
– vuoto:(grafo) → boolean
– insnodo: (nodo,grafo) → grafo
– insarco:(nodo,nodo,grafo)→ grafo
– cancnodo: (nodo,grafo) → grafo
– cancarco: (nodo,nodo,grafo) → grafo
– adiacenti: (nodo,grafo) → lista
– esistenodo: (nodo,grafo) → boolean
– esistearco: (nodo,nodo,grafo)→ boolean operatori
– legginodo: (nodo, grafo) → tipoelem aggiuntivi
– scrivinodo: (tipoelem, nodo, grafo) →grafo

**Specifica semantica**
Tipi:
grafo: insieme G = (N,A) con N sottoinsieme finito di elementi di tipo “nodo” e A incluso NxN
nodo: insieme finito qualsiasi
lista: lista di elementi di tipo nodo
boolean: insieme dei valori di verità

Operatori:
crea = G
POST: G = (N,A) con N = ins vuoto e A = ins vuoto

vuoto(G) = b
POST: b = vero se N = ins vuoto e A = ins vuoto; b falso altrimenti

insnodo(u,G) = G’
PRE: G = (N,A) u non app N
POST: G’ = (N’,A), N’ = N U {u}

insarco(u,v,G) = G’
PRE: G = (N,A), u app N, v app N, (u,v) non app A
POST: G’ = (N,A’), A’ = A U {(u,v)}

cancnodo(u,G) = G’
PRE: G = (N,A), u app N e non esiste v app N tale che (u,v) app A oppure (v,u) app A
POST: G’ = (N’,A), N’ = N \ {u}

cancarco(u,v,G) = G’
PRE: G = (N,A), u app N, v app N, (u,v) app A
POST: G’ = (N,A’), A’ = A \ {(u,v)}

adiacenti:(u,G) = L

PRE: G = (N,A), u app N
POST: L è una lista che contiene una e una sola volta gli elementi di A(u) = {v app N | (u,v) app A}

Quando ai nodi o agli archi sono associate informazioni, si parla di grafi etichettati: in tal caso vanno introdotti nuovi operatori per ritrovare o modificare le informazioni associate ai nodi e agli archi.

**Rappresentazione con matrice di adiacenza**
La più semplice rappresentazione utilizza una matrice NxN, E = [eij] tale che eij = 1 nel caso (i,j) app A, mentre eij = 0 se (i,j) non app A
Se il grafo è pesato, nella matrice si utilizzano i pesi degli archi al posto degli elementi binari. Se pij è il peso dell'arco (i,j) allora l'elemento della matrice E diventa pij se (i,j) app A e +inf (-inf) se (i,j) non app A.
E' possibile utilizzare la medesima rappresentazione per grafi non orientati: ne risulterà una matrice simmetrica rispetto alla diagonale principale (eij = eji)

**Rappresentazione con matrice d'incidenza**
Un grafo G = (N,A) può anche essere rappresentato mediante una matrice (NxM), B = [bik] nella quale ciascuna riga rappresenta un nodo e ciascuna colonna rappresenta un arco.
Per un grafo non orientato bik = 1 se l'arco j-esimo è incidente nel nodo i, 0 altrimenti
Nel caso di grafi orientati o diretti, il generico elemento bij di B diviene:
a) +1 se l'arco j-esimo entra nel nodo i
b) -1 se l'arco j-esimo esce dal nodo i
c) 0 altrimenti

In questo caso però, dato un nodo, non è facile ricavarne l'insieme di adiacenza. Per calcolare A(u) è necessario scandire la riga u di B alla ricerca delle colonne k tale che buk = -1 e per ogni colonna k scandire l'indire di riga i tale che bik = +1

**Rappresentazione con vettore di adiacenza: grafo orientato**
E' possibile rappresentare il grafo (N,A) con due vettori, il vettore NODI e il vettore ARCHI
Il vettore NODI è formato da N elementi e NODI(i) contiene un cursore alla posizione di ARCHI a partire dalla quale è memorizzato A(i)

**grafo non orientato**
E' necessario rappresentare ogni arco due volte, se i grafi sono etichettati sui nodi e/o sugli archi, i pesi possono essere memorizzati in vettori PESINODI(n) e PESIARCHI(m)

**Rappresentazione con lista di adiacenza**
E' possibile anche utilizzare un vettore di nodi A ed n liste. Una generica componente A(i) del vettore è il puntatore alla lista i-esima in cui sono memorizzati i nodi adiacenti a i.

**Matrice di adiacenza estesa**
Contiene campi aggiuntivi:
LABEL: etichetta del nodo
MARK: 0 se il nodo è stato rimosso, 1 se è presente
ARCHI: la somma dei nodi entranti ed uscenti

**Rappresentazione con struttura a puntatori**

E’ la versione dinamica della matrice di adiacenza estesa. Il nodo v con etichetta e, h archi entranti e k archi uscenti è rappresentato mediante un record.

**Visita di un grafo**
Esistono dei metodi sistematici per esplorare un grafo “visitando” almeno una volta ogni nodo ed ogni arco di un grafo non orientato e connesso oppure orientato e fortemente connesso.
Ricordiamo che grafo connesso è un grafo G = <N,A> in cui, dati u e v app N esiste un cammino da u a v o un cammino da v a u. G è detto fortemente connesso se per ogni coppia di nodi u e v esiste almeno un cammino da u a v ed almeno un cammino da v ad u.

**Depth first search (DFS)**
L’algoritmo prevede che contassegnamo un vertice appena lo visitiamo e poi cerchiamo di spostarci in un vertice adiacente non contrassegnato. Se non ci sono vertici non marcati indietreggiamo lungo i vertici già visitati finché non arriviamo ad un vertice che risulta adiacente ad uno non visitato e continuiamo cosi il procedimento

**Breadth first search (BFS)**
In questo schema ogni vertice adiacente al vertice corrente è visitato prima di spostarsi dal vertice corrente stesso. I nodi sono visitati in ordine di distanza crescente dal nodo di partenza u, dove la distanza da u ad un generico nodo v è il minimo numero di archi in un cammino da u a v. Conviene tenere in una coda i vertici visitati ma non completamente “esaminati” cosi che, quando siamo pronti a spostarci su un vertice adiacente al corrente, possiamo ritornare al vecchio vertice corrente dopo il nostro movimento.

**Algoritmo generale di visita**
Esaminato un nodo iniziale i, si visita un nodo j collegato a i e, successivamente, si considera un nuovo nodo k, diverso da i e da j, e adiacente a i oppure a j. Si utilizza un insieme R dei nodi visitati e un insieme Q dei nodi utili per il proseguimento della ricerca. Infine è utilizzato l’insieme B in cui vengono memorizzati gli archi utilizzati per la scoperta dei nuovi nodi.

Il critero di scelta del nodo stabilisce la modalità di visita.

Nella ricerca in profondità, che è diretta estensione della visita in ordine anticipato di un albero radicato, tra i vari nodi utili per il proseguimento della ricerca viene sempre scelto quello visitato più recentemente.

Nella ricerca in ampiezza i nodi sono visitati in ordine di distanza crescente dal nodo di partenza r, dove la distanza da r ad un generico nodo u è il numero di archi in un cammino da r ad u che sia il più corto possibile. Dunque la ricerca in ampiezza fornisce sempre un cammino “ottimo”.

Le procedure DFS e BFS sono metodi di visita sistematica che vengono usati per risolvere molti problemi: ad esempio verificare se un grafo non orientato è connesso oppure no, trovare tutti i sottografi connessi di un grafo non orientato.

Inoltre entrambe le visite, connettendo i nodi marcati ad uno ad uno a partire da un generico nodo i, generano un albero radicato nel nodo iniziale i.

Un problema classico è quello dell’individuazione in un grafo orientato del percorso più breve che unisce due generici nodi del grafo.

Sia G = (N,A) un grafo orientato etichettato negli archi con valori interi positivi (pesi). Partendo dal

nodo r app N per ogni nodo u app N, si cacola il percorso (cammino o catena) che connette r ad u tale che la somma dei pesi asociati agli archi che compongono il cammino sia minima.

## 12. TEORIA DELLA COMPLESSITA'

Lo studio della computabilità aiuta a stabilire quali problemi ammettono una soluzione algoritmica e quali no, per i problemi computabili è interessante conoscere la complessità degli algoritmi che li risolvono.

**Complessità di un algoritmo** : è una misura delle risorse di calcolo consumate durante la computazione.
**Efficienza di un algoritmo** : è inversamente proporzionale alla sua complessità.

Intuitivamente un programma è più efficiente di un altro se la sua esecuzione richiede meno risorse di calcolo (tempo di elaborazione, quantità di memoria).
Non possiamo misurare l'efficienza di un algoritmo in secondi perche non è attendibile, bisognerebbe considerare le condizioni in cui si effettuano le prove (elaboratore, compilatore, modalità ingresso dati, significatività dei dati di prova)
Utilizzando i secondi come unità di tempo non otteniamo una valutazione oggettiva.

In genere si valuta la ontà di un algoritmo sulla base del comportamento che questi presentano al crescere della dimensione del problema.
Questo studio è importante anche perché permette di stabilire se un problema ammette algoritmi risolutivi praticamente computabili.

L'analisi dell'efficienza di un programma è basaat sull'ipotesi che il costo di ogni istruzione semplice e di ogni operazione di confronto sia pari ad UNA UNITA' DI COSTO.
Indipendentemente da linguaggio o dal calcolatore usato.

Ci limitiamo a contare le operazioni eseguite o alcune operazioni chiave ammetendo che il tempo complessivo di esecuzione sia proporzionale al numero di tali operazioni.
Studiamo le funzioni di complessità nel loro ordine di grandezza, trattando come non significative le costanti moltiplicative.

Il costo di esecuzione di un programma dipende anche dai dati di ingresso, ad esempio il costo di un programma di ordinamento dipende dalle dimensioni dell'insieme di input
Caratteriziamo tale dimensione mediante un intero n.
- In una matrice, n sara' il numero dei suoi elementi.
- in un insieme, n sara' il numero dei suoi elementi
- in un grafo, n è l'ineieme dei nodi, oppure degli archi, o la somma dei due

**Complessità in spazio** è il massimo spazio invaso dalla memoria durante l'esecuzione dell'algoritmo il quale può costruire insiemi di dati intermedi o di servizio. Anche in questo caso ci si limita allo studio della complessità asintotica, ma poiché abbiamo a disposizione memorie grandissime, studieremo la **complessità in tempo**

### Complessità in tempo e asintotica

Fissata la dimensione n, il tempo che un algoritmo impiega a risolvere il problema è la complessità in tempo. Dobbiamo esprimere la complessità in tempo come funzione di n, ci limiteremo a studiare il cimportamento di tale funzione al crescere di n, considerando solo i termini prevalenti e tralasciando anche le costanti moltiplicative.

**Modello di costo**
Useremo il modello di costo di un linguaggio di programmazione lineare.
- Assumiamo che il costo di esecuzione di un istruzione semplice sia unitario
- Il costo di esecuzione di un istruzione composta è pari alla somma dei costi delle istruzioni che la compongono.
- Il costo di un operazione di selezione (if) è dato dal costo test condizione + max(s1,s2) nel caso peggiore
- Il costo di un while cond do S1: costo cond + (costo cond + costo S1) * n
- costo repeat: reepeat S1 until cond : (costo cond + costo S1) * n
-costo for: for i = 1 to n do S1: 2 + (costo S1 + 3) * n

**Complessità e configurazioni**
La complessità di un algoritmo non può sempre essere caratterizzata da una sola funzione di complessità. A parità di dimensione di dati, il tempo di esecuzione può variare in base alla configurazione dei dati. Per questo, si considerano di solito 3 differenti tipi di complessità: complessità nel caso medio, ottimo e pessimo.
- Caso Pessimo: si ottiene considerando quella particolare configurazione che a parità di dimensione dei dati dà luogo al massimo tempo di calcolo. Tale funzione fornisce un limite superiore alla complessità entro cui il funzionamento dell'algoritmo è sempre garantito.
- Caso ottimo: si ottiene considerando la configurazione che dà luogo al minimo tempo di calcolo.
- Caso medio: si riferisce al tempo di calcolo mediato su sutte le possibili configurazioni dei dati, sempre per una data dimensione n.

**Riassumendo**
- La complessità di un algoritmo è funzione della dimensione dei dati, ovvero della model dei dati del problema da risolvere.
- Determinare la complessità in tempo (o in spazio) significa determinare una funzione di complessità f(n) che fornisca la misura del tempo (o dello spazio di memoria occupato) al variare della dimensione dei dati, n.
- Le funzioni di complessità sono caratterizzate da due proprietà: assumono solo valori positivi, sono crescenti rispetto alla dimensione dei dati.

**Comportamento asintotico**
Per semplificare l'analisi si stabilisce il comportamento asintotico della funzione quando le dimensioni dell'input tendono all'infinito. Si ignorano le costanti moltiplicative e i termini additivi di ordine inferiore.

**Notazioni**
O(f(n)) è l'insieme di tutte le funzioni g(n) tali che esistono due costanti positive c ed m per cui:
g(n) <= c * f(n) per ogni n >= m → g(n) app O(f(n))

Applicata alla funzione di complessità, la notazione O delimita superiormente la crescita.
Es. f(n) = 3n^2 + 3n – 1 è un O(n^2) dal momento che: per ogni n >=3: f(n) <= 4n^2

Ω(f(n)) è l'insieme di tutte le funzioni g(n) tali che esistono due costanti positive c e m per cui:
g(n) <= c * f(n) per ogni n >= m

g(n) app Ω(f(n)) fornisce un limite inferiore al comportamento asintotico della funzione

θ(f(n)) è l'insieme di tutte le funzioni g(n) che sono sia Ω(f(n)) che O(f(n)). E' l'insieme di tutte le funzioni g(n) tali che esistano tre costanti positive c, d, m per cui:
d * f(n) >= g(n) >= c * f(n) per ogni n >= m

Abbiamo che la g si comporta asintoticamente esattamente come la f.

Esempio: h(n) = 3n^2 +3n -1

è di ordine n^2 perché esistono le costanti c = 4 e n = 3 per cui h(n) < 4n^2 per n >= 3

Notiamo che qualsiasi polinomio di grado k è di ordine n^k e in accordo con la definizione, è anche di ordine n^j con j > k

g(n) = 3n^2 + 2n + 1
è O(n^2) e anche O(n^3) ma è anche di ordine Ω(4n^2), Ω(n^2), Ω(n). Inoltre è di ordine θ(n^2), θ(4n^2) ma non di ordine θ(n) e neppure di ordine θ(n^3)

**Proprietà**

- g app O(f) implica (f+g) app O(f)
- f1 app O(g1), f2 app O(g2) implica f1 + f2 app O(g1+g2)
- O e Ω sono relazioni riflessive e transitive
- θ è una relazione di equivalenza
- f app O(g) se e solo se g app Ω(f)

**Valutazione complessità**
Diamo delle regole per aiutare a trovare la delimitazione superiore della complessità.

Regola 1 – Supponiamo che il programma sia composto di due parti P e Q da eseguire sequenzialmente e che i costi di P e Q siano S(n) = O(f(n)) e T(n) = O(g(n)). Allora il costo del programma è O(max(f(n),g(n))).

Dimostrazione: In questo caso il costo complessivo è: S(n) + T(n)
Notiamo:
1. S(n) = O(f(n)) → esistono a' e n' tc S(n) < a' * f(n) per n > n'
2. T(n) = O(g(n)) → esistono a'' e n'' tc T(n) < a'' g(n) per n > n''
Ne deriva che, per n > max (n',n''):
S(n) + T(n) < a' * f(n) + a'' * g(n) < (a' + a'') * max(f(n),g(n))

Quindi, O(S(n) + T(n)) è proprio O(max(f(n),g(n)))

Regola 2 – Supponiamo che un programma richieda per k volte l'esecuzione di una istruzione

composta o l'attivazione di una procedura, e sia fi(n) il costo relativo all'esecuzione i-esima, i = 1, …,k. Il costo complessivo del programma è pari a O(Σi fi(n))

**Istruzione dominante –** Sia dato un programma P il cui costo di esecuzione è t(n). Una istruzione di P si dice dominante se per ogni interno n, essa viene eseguita, nel caso peggiore di input avente dimensione n, un numero di volte d(n) che verifica la seguente condizione:

t(n) < a * d(n) + b

In altre parole un istruzione dominante viene eseguita un numero di volte proporzionale al costo di esecuzione di tutto l'algoritmo. E' importante osservare che in un programma più istruzioni possono essere dominanti, ma può anche accadere che il programma non contenga affatto istruzioni dominanti.

Regola 3 – Supponiamo che un programma contenga un istruzione dominante, che nel caso peggiore di input di dimensione n viene eseguita d(n) volte. La delimitazione superiore alla complessità del programma è O(d(n))

Per individuare un istruzione dominante è sufficiente esaminare in molti casi le operazioni che sono contenute nei cicli più interni del programma.

Grazie all'analisi della complessità asintotica possiamo categorizzare gli algoritmi nelle seguenti classi di complessità:

- costante O(1) <
- logaritmica O(log n) <
- lineare O(n) <
- nlog O(nlogn) <
- quadratica O(n^2) <
- cubica O(n^3) <
- esponenziale O(a^n) con a >1

Gli algoritmi con complessità costante sono più efficienti di quelli con complessità logaritmica e cosi via.
La distinzione di maggiore interesse si ha tra:
algoritmi polinomiali: complessità O(n^k), k > 0
es: O(log n), O(n^10)

algoritmi esponenziali: complessità O(a^g(n)) con a > 1 g(n) crescente
es: O(2^n), O(2^log n), O(n^log n)

Gli algoritmi di importanza pratica sono quelli polinomiali.

## 13. CLASSIFICAZIONE DEI PROBLEMI

I problemi si possono classificare in:
- Problemi di ricerca
- Problemi di decisione
- Problemi di ottimizzazione

Nella fase di concettualizzazione si fornisce una specifica del problema, stabilendo le caratteristiche strutturali del problema.
Nella fase di realizzazione si passa dalla specifica del problema alle scelte per la sua risoluzione.

**Specifica del problema**
- Scelta dell'input: si stabilisce che i valori di alcune variabili in gioco siano i dati di ingresso del problema, l'insieme di tali valori è lo SPAZIO DI INPUT del problema.
- Scelta dello scopo della risoluzione: si stabilisce che i valori di alcune variabili in gioco rappresentano le soluzioni del problema. l'insieme di tali valori è lo SPAZIO DELLE SOLUZIONI.
- Identificazione del legame che i vincoli del problema impongono tra lo spazio di input e spazio delle soluzioni. Ciò conduce alla RELAZIONE CARATTERISTICA del problema.
- Determinazione di quali informazioni in uscita si vuole che il processo risolutivo produca. Questo stabilisce lo SPAZIO DI OUTPUT e il QUESITO del problema.

Esempio: determinare la posizione di un numero intero in un vettore di interi
Variabili: vettore V, indice K, num intero M

SPAZIO DI INPUT: insieme delle possibili coppie formate da un vettore V e un intero M
SPAZIO DELLE SOLUZIONI: insieme degli indici di ogni possibile vettore
RELAZIONE CARATTERISTICA: associa ad ogni coppia (V,M) una posizione K se e solo se il K-esimo elemento di V coincide con M. La relazione la chiamiamo R-vett*
SPAZIO DI OUTPUT: contiene lo spazio delle soluzioni più un particolare simbolo che denota l'assenza di soluzioni.
QUESITO: stabilisce che per ogni manifestazione che ammette soluzione, vogliamo come output un qualunque elemento dello spazio delle soluzioni legato a quella manifestazione dalla relazione caratteristica.

Sia:
V=[7,5,3,7,2]

Rvett <[7,5,3,7,2],7> → 1
<[7,5,3,7,2],5> → 2
<[7,5,3,7,2],7> → 4

Poichè esiste un numero infinito di vettori, Rvett è di dimensione infinita. Consideriamo l'elemento dello spazio di input formato da V e da M = 7.
Rvett associa a tale elemento 2 possibili valori dello spazio delle soluzioni: 1 e 4
Il quesito stabilisce che nel problema considerato, possiamo essere interessati a una qualunque, a tutte, ecc soluzioni.

**Specifica di un problema, definizione:**
La specifica di un problema è una quintupla <I,S,R,O,Q> dove:
- I è lo spazio di input
- S è lo spazio delle soluzioni
- R incluso I x S è una relazione su I ed S detta relazione caratteristica
- O è lo spazio di output

- Q è una regola, detta quesito, che sulla base della relazione caratteristica R consente di definire una relazione su I e O : Rq inc I x O

**Soluzione di un problema per una sua istanza, definizione:**
Sia P = <I,S,R,O,Q> un problema e sia i app I una sua istanza, una soluzione Si di P per i è un elemento di S per cui vale: <i,Si> app R

**Risposta ad un problema per una sua istanza, definizione:**
Sia P = <I,S,R,O,Q> un problema e sia i una sua istanza, una risposta ri a P per i è un elemento di O per cui vale: <i,ri> app Rq

Dove Rq è la relazione su I e O definita in base ad R applicando la regola Q

**Problemi di ricerca**
Tra tutte le possibili soluzioni siamo interessati a conoscere una qualunque delle soluzioni del problema per l'istanza che stiamo considerando, oppure nel caso non esista una soluzione, siamo interessati a saperlo. Introduciamo nello spazio di output il simbolo $\perp$ che individua l'assenza di soluzioni.
Problema di ricerca P = <I,S,R,S U {$\perp$], qric>
dove qric è la regola che definisce in base ad R la relazione Rqric contenente tutti e soli i seguenti elementi:
- ogni coppia contenuta in R
- una coppia <i, $\perp$> per ogni istanza di i di P per la quale P non ha soluzioni

In questo caso siamo interessati, per ogni istanza, a calcolare UNA delle soluzioni OPPURE a segnalare il fatto che non ne esistono. Si dice: trovare una soluzione ammissibile, cioè che soddisfi la relazione.

**Esempio: problema n regine**
Dato un intero positivo n, si determini un posizionamento di n regine in una scacchiera n x n tale che nessuna regina minacci qualche altra regina.
Specifica del problema:
<N+,D,R,D U {$\perp$}, qric> dove:
N+: insieme dei numeri positivi
D: insieme delle disposizioni di regine in scacchiere di ogni dimensione con num regine = dim scacchiera)
R: relazione che contiene tutte e sole le coppie <x,y> in cui x app N+ e y è una disposizione di x regine nella scacchiera x * x

**Problemi di decisione**
La risposta è vero o falso a seconda che il dato di ingresso soddisfi o meno una certa proprietà. Dunque lo spazio di output contiene solo i due valori di verità.
**Definizione**: un problema di decisione P è un problema specificato con una quintupla del tipo:
<I,S,R,{TRUE,FALSE},qdec>
dove il quesito qdec definisce la funzione: Rqdec: I → {TRUE,FALSE}
tale che per ogni istanza i di P, Rqdec(i) vale:
TRUE se esiste s app S tale che <i,s> app R
FALSE altrimenti

In linea di principio un problema di decisione può essere risolto mediante due approcci:
- cercare una risposta al corrispondente problema di ricerca, detto problema sottostante:
P '=<I,S,R,S U {$\perp$},qric>

sfruttando il fatto che dato un i app I la riposta a P per i è FALSE solo se la risposta a P' coincide con $\perp$ mentre in ogni altro caso la risposta a P è TRUE (approccio costruttivo)

**esempio: problema di partizionamento**
Dati k numeri interi positivi n1,…,nk la cui somma è 2m. Decvidere se i k numeri possono essere ripartiti in due insiemi in modo che la somma dei componenti di ciascun gruppo sia m.
Dunque se na1,…,nap e nb1,…,nbq sono i due gruppi (cioè p+q = k) allora:
sommatoria(da i = 1 a p) nai = sommatoria(da j = 1 a q) nbj = m

I: insieme di tutti i possibili insiemi di interi la cui somma è pari
S: insieme di tutte le possibili coppie di insiemi di interi
R: insieme di tutte e sole le coppie <x,y> tale che:
- x app I, y app S
- y = <y1,y2> è una coppia di insiemi tale che y1 ^ y2 = ins vuoto e y1 U y2 = x
e sommatoria(y1) = sommatoria(y2)

## problemi di ottimizzazione

Alle soluzioni ammissibili è associata una misura (costo, obiettivo): risolvere il problema non significa trovare una qualunque soluzione ma la migliore soluzione secondo la misura o criterio fissato.

P = <I,S,R,S U {$\perp$}, qott(M,m,⊆)> dove:

- M è un insieme qualsiasi
- m è una funzione del tipo I x S → M detta funzione obiettivo di P: per una certa istanza i ∈ I, il valore m(i,s) rappresenta una misura dell'elemento s nello spazio delle soluzioni.
- ⊆ è una relazione di ordinamento su M (x ⊆ y "x migliore di y)
- qott(M,m,⊆) è una regola che definisce in base ad R la relazione Rqott ⊆ IxS su I ed S cosi da individuare:
  a) una coppia (i,s) per ogni i tale che non esiste s' associata ad i migliore.
  b) una coppia (i,$\perp$) per ogni i per la quale P non ha soluzioni

**esempio:**
Dato un insieme W di parole, determinare una parola A ∈ W tale che non esistano altre parole in W:
P1 – Maggiori di A secondo l'ordine alfabetico
P2 – Di lunghezza maggiore di quella di A
P3 – Composte usando tutte le lettere dell'alfabeto che compongono A, più altre lettere.

Lo schema generale: $<2^w, W, \{(X,y)|X\in 2^w \wedge y\in X\}, W \cup \{\perp\}, qott(M,m,\subseteq)$
dove:
W è l'insieme di parole
$2^w$ è l'insieme di tutti gli insiemi di parole
M è un differente insieme per ciascun problema

Per P1:
M = W
Funzione obiettivo: m: $2^w$ x W → M
Per ogni i app I, m(i,W) è uguale a W
La relazione ⊆ è la relazione di ordinamento alfabetico

Per P2:
M = N
La funzione obiettivo: m: $2^w$ x W → M
Per ogni i app I, m(i,W) è la lunghezza di W
La relazione ⊆ è la relazione di ordinamento >= tra numeri

Per P3:
M è l'insieme di tutti gli insiemi di lettere dell'alfabeto
La funzione obiettivo: m: $2^w$ x W → M
Per ogni i app I, m(i,W) restituisce l'insieme delle lettere che compongono W
La relazione è quella di contenimento ⊆ tra insiemi

**Problema dello zaino**
n articoli, Pi profitto per articolo i, Ci costo per per articolo i, B budget totale.
Consideriamo:
P1,…,Pn,C1,…,Cn,B (quindi 2n+1 interi positivi)

Troviamo n valori interi x1,…,xn tale che:

Xi app {0,1} e 1 <= i <= n
Sommatoria(da i = 1 a n) Pi * Xi abbia valore max
Sommatoria(da i = 1 a n) Ci * Xi <= B

Lo spazio di input I è l'insieme delle coppie (n,Q) dove n è un intero positivo e Q una sequenza di 2n+1 interi positivi. L'elemento di questo spazio è:
(n,(P1,…,Pn,C1,…,Cn,B))

Lo spazio delle soluzioni S è l'insieme delle sequenze finite:
<X1,…,Xn> con n app N+ e Xi app {0,1}

La relazione caratteristica è data dall'insieme delle coppie:
(n,(P1,…,Pn,C1,…,Cn,B)) , (x1,…,xn) tali che:
Sommatoria(da i = 1 a n) Ci * Xi <= B

Il quesito è qott(N,m,>=) dove:
m = Sommatoria(da i = 1 a n) Pi * Xi

Dunque la specifica di un problema determina la struttura del problema in modo indipendentemente dal modo in cui il problema verrà risolto. Lo spazio delle soluzioni è caratteristica generale del problema e non fornisce indicazioni sulle soluzioni per una generica istanza.

Uno strumento che aiuti a caratterizzare le potenziali soluzioni ad una generica istanza di un problema per orientare la scelta di un algorimo risolutivo: lo spazio di ricerca.

**Lo spazio di ricerca**
E' necessario disporre di uno strumento che aiuti a caratterizare le potenziali soluzioni di una generica istanza del problema.
Determinare lo spazio di ricerca per un problema P significa stabilire un metodo che, per ogni istanza i di P, consente di definire un insieme con associate due funzioni:
- La funzione di ammissibilità: che permette di verificare se un elemento dello spazio di ricerca corrisponde effettivamente ad una soluzione per i.
- La funzione di risposta: che permette di ottenere, dagli elementi dello spazio di ricerca, le corrispondenti risposte per i.

Sia i una istanza di un problema P = <I,S,R,S U {⊥}, Q>
Uno spazio di ricerca di P per i è costituito da:
1- Un insieme Zi con associate due funzioni:
- Funzione di ammissibilità: a: Zi → {TRUE,FALSE}
- Funzione di risposta: o: Zi → S

Che soddisfano le seguenti condizioni:
1) Per ogni elemento z di Zi, a(z) = TRUE se e solo se o(z) è soluzione di P per i
2) i ha risposta positiva se e solo se vi è almeno un elemento z di Zi per cui a(z)=TRUE e o(z) è una risposta a P per i.

2- Un metodo per rappresentare ogni elemento di Zi mediante una struttura di dati (vettore, matrice, lista, albero, …) e per esprimere la funzione di ammissibilità e di risposta in termini di tale struttura.

Lo spazio di ricerca associato ad una istanza i di P caratterizza le soluzioni e le corrispondenti risposte a P per l'istanza i

**Struttura dello spazio di ricerca per problemi di ricerca con risposta positiva(todo)**

**Struttura dello spazio di ricerca per problemi di ottimizzazione con risposta positiva(todo)**

**esempio: ricerca di un elemento in un vettore**
Consideriamo una generica istanza del problema con input un vettore V di n elementi e un numero M

- Lo spazio di ricerca associato a tale istanza è l'insieme devi valori che sono indici del vettore (interi da 1 a n)

- La funzione di ammissibilità verifica, dato un j, se V[j] = M

- La funzione di risposta restituisce j

- La struttura dati per rappresentare i singoli elementi dell'insieme è una semplice variabile di tipo intero.

**Differenze tra spazio delle soluzioni e spazio di ricerca per un problema P**

- Lo spazio delle soluzioni è una componente della specifica del problema e fa globalmente riferimento a tutte le possibili soluzioni per il problema.
- Lo spazio di ricerca fornisce uno strumento per caratterizzare le soluzioni di ogni singola istanza del problema
- Lo spazio di ricerca determina anche per ogni istanza, una struttura dati e un meccanismo per verificare, tramite la formulazione della funzione di ammissibilità in termini di tale struttura, se una sua configurazione corrisponde o meno ad una soluzione per l'istanza e un metodo per derivare, tramite la funzione di risposta, una risposta all'istanza.

**Spazio di ricerca per un problema P**
Esistono diversi modi per determinare uno spazio di ricerca di un problema.
Criteri per valutare la qualità di una scelta rispetto ad un'altra:
- Lo spazio di ricerca deve caratterizzare le soluzioni ad una istanza in modo non ridondante: è opportuno scegliere una struttura dati che escluda a priori configurazioni che non corrispondono a priori ad alcuna soluzione

- Lo spazio di ricerca non deve essere una banale riformulazione del problema e della sua relazione caratteristica, ne deve nascondere la difficoltà di un problema ma fornire elementi significativi per la comprensione della struttura del problema e delle sue soluzioni.

**Esempio: problema delle n regine (todo)**

**Tecniche algoritmiche**
Sono basate su diversi paradigmi di utilizzo dello spazio di ricerca e consideriamo:
- La tecnica di enumerazione
- La tecnica di backtracking
- La tecnica golosa (greedy)
- La tecnica divide et impera

I paradigmi sono selettivo e generativo.

Dal paradigma SELETTIVO sono create tecniche di progetto di algoritmi che, per l'istanza del problema presa in considerazione, visitano lo spazio di ricerca tentando di trovare un elemento ammissibile.
Ogni algorimo fa riferimento all'intero spazio di ricerca che viene esplorato con sistematicità in una definita modalità (enumerativa e backtracking)

Dal paradigma GENERATIVO scaturiscono tecniche di progetto di algoritmi che generano direttamente la soluzione senza selezionarla tra gli elementi dello spazio di ricerca. N queso paradigma lo spazio di ricerca è considerato esclusivamente in fase di progetto allo scopo di caratterizzare le soluzioni del problema. (tecnica golosa, divide et impera)

bentelant