APPUNTI DI STRUTTRE DATI E ALGORITMI

INDICE

CAP 01 - INTRODUZIONE

ALGORITMO

- Definizione: un algoritmo è un procedimento sistematico che consente di ottenere un risultato eseguendo, in un determinato ordine, un insieme di passi elementari corrispondenti ad azioni scelte solitamente da un insieme finito
- Caratteristiche fondamentali:
  - a sequenza di istruzioni deve essere finita (finitezza della descrizione, finitezza del procedimento, finitezza dei singoli passi)
  - le istruzioni devono essere eseguibili materialmente (realizzabilità)
  - le istruzioni devono essere espresse in modo non ambiguo (non ambiguità)
  - il procedimento deve portare ad un risultato (effettività)
- Altre proprietà desiderabili
  - un algoritmo dovrebbe funzionare correttamente anche variando alcuni aspetti del problema (generalità)
  - il numero di azioni eseguite per la soluzione del problema dovrebbe essere minimo (efficienza)
  - la sequenza di azioni che verranno eseguite per ogni insieme di dati dovrebbe essere prevedibile (determinismo)

STRUTTURE DATI

- Organizzazione efficiente dei dati
- Servono a rappresentare i dati di problemi complessi strutturando i dati elementari (bit, caratteri, interi, strighe, ...)
- Rappresentano istanze di problemi computazionali reali e concreti

ALGORITMI E PROBLEMI

Un algoritmo risolve un problema

- dal punto di vista matematico è una corrispondenza univoca fra spazio delle istanze e spazio delle soluzioni
- l'algoritmo descrive il rpocedimento che, una volta affidato ad un "esecutore", consente di "calcolare" la corripondenza

ALGORITMI E PROGRAMMI

Un programma è l'implementazione di un algoritmo in un linguaggio di prgrammazione
Sostanzialmente è la concretizzazione di un algoritmo in modo che sia eseguibile da un computer

PROBLEMI DECIDIBILI E INDECIDIBILI

PROBLEMI DECIDIBILI

Un problema è detto decidibile (o calcolabile) se esiste un algoritmo che per ogni sua istanza sia in grado di terminare la sua esecuzione pervenendo ad una soluzione

PROBLEMI INDECIDIBILI

Un problema è indecidibile (o non calcolabile) se nessun algoritmo sia in grado di terminare la sua esecuzione pervenendo ad una soluzione su ogni sua istanza

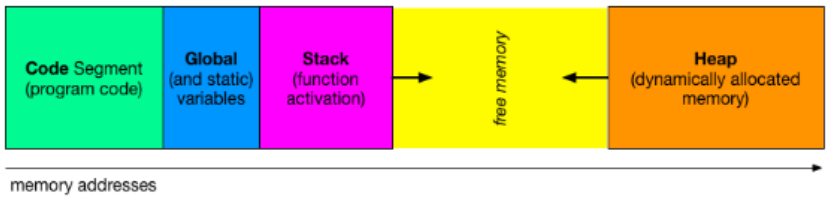
INDECIBILITA'

Un problema è indecidibile se non esiste alcun algoritmo di risoluzione che termina su ogni input restituendo un risultato

CAP 02 - ORGANIZZAZIONE DELLA MEMORIA, CHIAMATE DI FUNZIONI, RICORSIONE

L'ORGANIZZAZIONE DELLA MEMORIA

La memoria in un programma C è organizzata come lo schema seguente



Dove ogni blocco svolge una funzione differente e specifica:

1. Code Segment
   - Contiene il codice eseguibile del programma
   - E' tipicamente una regione di sola lettura, in modo da prevenire modifiche accidentali o intenzionali al codice
   - Può essere condiviso tra processi che eseguono lo stesso programma per ottimizzare la memoria
2. Variabili Globali e Statiche
   - In questa sezione si trovano:
     - Le variabili globali (visibili in tutto il programma)
     - Le variabili statiche, che mantengono il loro valore tra le diverse chiamate di funzione
   - E' diviso in:
     - Dati inizializzati: variabili con valori assegnati esplicitamente
     - Dati non inizializzati: variabili inizalizzate implicitamente a 0

       -Questa memoria è allocata all'avvio del programma e liberata alla sua terminazione
3. Stack
   - Gestisce i **Record di attivazione** delle funzioni
   - Ogni funzione che viene chiamata crea un nuovo record di attivazione nello stack
   - segue la politica LIFO (Last In First Out)
   - E' gestito automaticamente e ha dimensione limitata. un uso eccessivo prota a un stack overflow
4. Free Memory
   - Questa regione rappresenta la memoria non ancora utilizzata
   - Funziona come un'area di separazione tra lo stack e l'heap
   - Può essere espansa o ridotta dinamicamente a seconda delle esigenzxe del programma
5. Heap
   - Utilizzato per l'allocazione dinamica della memoria durante il runtime
     - Es: `malloc()` , `calloc()` , `realloc()` e deallocazione con `free()`
   - La gestione è manuale: il programmatore deve liberare esplicitamente la memoria per evitare **memory leak**
   - Si espande verso l'alto (verso l'area libera), mentre lo stack si espande verso il basso

RECORD DI ATTIVAZIONE

Ciascuna funzione crea un suo spazio specifico di emmorizzazione nello *stack* detto **record di attivazione**

Esso conteiene tutte le variabili locali della funzione, i suai parametri e i valori di ritorno

LA RICORSONE

Una funzione è detta *ricorsiva* se nella sua definizione compare un riferiemtno (chiamata) a se stessa

- La funzione ricorsiva sa risolvere direttamente dei casi particolari del problema, detti casi base, in tal caso è in grado di restituire immediatamente il risultato
- se invece le vengono passati dei dati che non costituiscono uno dei casi di base chiama se stessa passando dei dati *"ridotti"* o *"semplificati"*

Ad ogni chiamata i dati si riducono così ad arrivare, ad un certo punto, ad uno dei casi base

RICORSIONE: MECCANISMO COMPUTAZIONALE

Quando la funzione chiama se stessa, la sua esecuzione si sospende per eseguire la nuova chiamata

- L'esecuzione riprende quando la chiamata interna termina
- La sequenza di chiamate ricorsive termina quando quella più annidata incontra uno dei casi base

Ogni chiamata alloca sullo *stack* nuove istanze dei parametri e delle variabili locali all'interno del record di attivazione

RICORSIONE E INDUZIONE

La ricorsione è basata sul principio di induzione matematica:

- Se una proprietà P vale per $n = n_0$ (**caso base**)
- e si può provare che assumendola valida per n, vale anche per $n + 1$ (**passo induttivo**)
- allora P vale per ogni $n \geq n_0$

Analogamente, per risolvere il problema con un approccio ricorsivo comporta:

- l'identificazione del **caso base** in cui la soluzione sia nota
- la capacità di esprimere il caso generico in termini dello stesso problema ma in uno o più casi più semplici

CONSIDERAZIONI FINALI SULLE FUNZIONI RICORSIVE

PRO E CONTRO DELLE FUNZIONI RICORSIVE

Pro:

- Metodo particolarmente utile su strutture dati inerentemente ricorsive (alberi, grafi)
- Qualora applicabili, le funzioni ricorsive sono più chiare (ed eleganti), più semplici, più brevi e più comprensibili rispetto alle versioni iterative

Contro:

- Le risorse di memoria richieste sono generalmente superiori a quelle della corrispondente soluzionie iterativa
  - richiede temp per la gestione dello stack
  - richiede memoria

QUANDO UTILIZZARE LA RICORSIONE

- In generale qualunque problema ricorsivo può essere risolto in modo non ricorsivo (iterativo)
- E' utile quando la soluzione iterativa è difficile da individuare o in generale più complessa
- In caso sia possibile farlo, definire una versione con ricorsione in coda è certamente da preferirsi

CAP 03 - TRATTABILITA' E COMPLESSITA' COMPUTAZIONALE

COMPLESSITA'

Decidibilità

- Un problema è **decidibile** se esiste un algoritmo che fornisce una risposta corretta ("sì" o "no") per ogni input valido in **tempo finito**.

- Si concentra sulla **esistenza** di una soluzione computabile.
- **Esempio**: Stabilire se una stringa appartiene a un linguaggio regolare.

Trattabilità

- Un problema è **trattabile** se è possibile risolverlo in un tempo **ragionevole**, tipicamente polinomiale rispetto alla dimensione dell'input (classe **P**).
- Si concentra sull'**efficienza** della soluzione.
- **Esempio**: Ordinare una lista di numeri è trattabile perché ci sono algoritmi efficienti (es. QuickSort).

Sintesi

- **Decidibilità**: riguarda **se** un problema può essere risolto.
- **Trattabilità**: riguarda **quanto efficacemente** può essere risolto.

Algoritmi Polinomiali ed Esponenziali

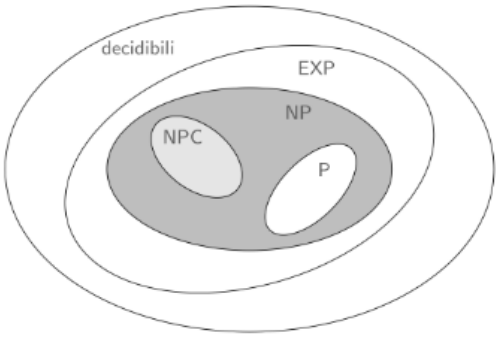
Definizione: Un algoritmo è detto polinomiale, nella dimensione del suo input n, se esistono due costanti c, $n_0 > 0$ tali che il numero di passi elementari da esso eseguiti è al più n^c, per ogni input di dimensione n e per ogni $n > n_0$.

Definizione: Un algoritmo è detto esponenziale, nella dimensione del suo input n, se esistono due costanti c, $n_0 > 0$ tali che il numero di passi elementari da esso eseguiti è al più c^n, per ogni input di dimensione n e per ogni $n > n_0$.

Problema trattabile: esiste un algoritmo polinomiale che lo risolve

Problema intrattabile: non esiste un algoritmo polinomiale che lo risolve

CLASSI DI COMPLESSITA' DEI PROBLEMI



- P classe dei problemi risolvibili in tempo polinomiale
- EXP classe dei problemi risolvibili in tempo esponenziale
- NP classe dei problemi per i quali verificare una soluzione richiede tempo polinomiale
- NPC classe dei problemi completi per NP, detti $NP - Completi$

ANALISI DELLA COMPLESSITA'

Misura di performance di un algoritmo espresse in funzione della dimensione dei dati in input n

- tempo: numero di operazioni RAM (elementari) eseguite
- spazio: numerodi celle di memoria occupate

Complessità o costo comutazionale $f(x)$ in tempo e spazio di un problema Π:

- caso pessimo o peggiore: costo massimo fra tutte le istanze di Π aventi dimensione dei dati pari a n
- caso medio: costo mediato tra tutte le istanze di Π aventi dimensione pari a n
- caso ottimo: costo minimo fra tutte le istanze di Π aventi dimensione dei dati pari a n

Notazione asintotica e ordini di complessità

Notazione Asintotica

La **notazione asintotica** descrive il comportamento di un algoritmo quando la dimensione dell'input (n) cresce verso l'infinito. Fornisce una stima della crescita del tempo di esecuzione o dello spazio richiesto.

Principali Notazioni

1. **Limitazione Superiore** $O(f(n))$:
   Rappresenta un limite superiore asintotico. Indica il caso peggiore della crescita dell'algoritmo.
   - Formalmente:

     $$g(n) \in O(f(n)) \quad \text{se e solo se esistono costanti } c > 0 \text{ e } n_0 > 0 \text{ tali che: } g(n) \leq c \cdot f(n) \quad \forall n \geq n_0$$

   - **Esempio**: Se $f(n) = 3n^2 + 5n + 2$, allora $f(n) \in O(n^2)$, perché la crescita è dominata dal termine n^2.
2. **Limitazione inferiore** $\Omega(f(n))$:
   Rappresenta un limite inferiore asintotico. Indica la crescita minima dell'algoritmo.
   - Formalmente:

     $$g(n) \in \Omega(f(n)) \quad \text{se e solo se esistono costanti } c > 0 \text{ e } n_0 > 0 \text{ tali che: } g(n) \geq c \cdot f(n) \quad \forall n \geq n_0$$

   - **Esempio**: Se $f(n) = 3n^2 + 5n + 2, allora f(n) \in \Omega(n^2)$.
3. **Limitazione** $\Theta(f(n))$:
   Indica che un algoritmo cresce esattamente al ritmo indicato.
   - Formalmente:

     $$g(n) \in \Theta(f(n)) \quad \text{se e solo se esistono costanti } c_1, c_2 > 0 \text{ e } n_0 > 0 \text{ tali che: } c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \quad \forall n \geq n_0$$

   - **Esempio**: Se $f(n) = 3n^2 + 5n + 2$, allora $f(n) \in \Theta(n^2)$.

Ordini di Complessità

Gli ordini di complessità rappresentano le classi di crescita degli algoritmi. Ecco alcuni esempi comuni:

1. **Costante** $O(1)$:
   - Il tempo di esecuzione non dipende dalla dimensione dell'input.
   - **Formula**:

     $$T(n) = c$$

   - **Esempio**: Accesso diretto a un elemento in un array, es. $A[i]$.
2. **Logaritmica** $O(\log n)$:
   - La complessità cresce molto lentamente con l'aumento dell'input.
   - **Formula**:

     $$T(n) = c \cdot \log n$$

   - **Esempio**: Ricerca binaria.
3. **Lineare** $O(n)$:
   - La complessità cresce in modo proporzionale alla dimensione dell'input.

- **Formula**:

$$T(n) = c \cdot n$$

- **Esempio**: Scansione di un array.

4. **Lineare-logaritmica** $O(n \log n)$:
   - Tipico degli algoritmi di ordinamento efficienti.
   - **Formula**:

$$T(n) = c \cdot n \cdot \log n$$

   - **Esempio**: MergeSort.
5. **Quadratica** $O(n^2)$:
   - La complessità cresce quadraticamente rispetto all'input.
   - **Formula**:

$$T(n) = c \cdot n^2$$

   - **Esempio**: Confronto di tutte le coppie in un array (es. BubbleSort).
6. **Esponenziale** $O(2^n)$:
   - La complessità cresce esponenzialmente con l'input.
   - **Formula**:

$$T(n) = c \cdot 2^n$$

   - **Esempio**: Risoluzione di problemi con enumerazione di tutte le sottoinsiemi.
7. **Fattoriale** $O(n!)$:
   - La complessità cresce estremamente rapidamente.
   - **Formula**:

$$T(n) = c \cdot n!$$

   - **Esempio**: Problema del commesso viaggiatore (con approccio brute force).

Definizioni

- **Notazione asintotica**: descrive come il tempo o lo spazio richiesto cresce con l'input.
- **Ordini di complessità**: classificano gli algoritmi in base al costo computazionale.
- La scelta di un algoritmo dipende dall'equilibrio tra efficienza e necessità pratiche.

Proprietà della notazione asintotica

- Riflessività: per ogni costante c, $c \cdot f(n)$ è $O(f(n))$ (lo stesso per Ω e Θ)
- Transitività: se $g(n) = O(f(n))$ e $f(n) = O(h(n))$ allora $g(n) = O(h(n))$ (lo stesso per Ω e Θ)
- Simmetria: $g(n) = \Theta(f(n))$ se e solo se $f(n) = \Theta(g(n))$
- Simmetria trasposta: $g(n) = O(f(n))$ se e solo se $f(n) = \Omega(g(n))$
- Somma: $g(n) + f(n) = O(max\{f(n), g(n)\})$ (lo stesso per Ω e Θ)
- Prodotto: $g(n) = O(f(n))$, $h(n) = O(q(n))$ allora $g(n) \cdot h(n) = O(f(n) \cdot q(n))$ (lo stesso per Ω e Θ)

ANALISI STRUTTURALE

Per gli algoritmi descritti in un linguaggio imperativo (come il C) è possibile definire delle regole che consentono di stimare la loro complessità computazionale direttamente dalla struttura del programma stesso

- La complessità di una sequenza di istruzioni è data dalla somma delle complessità delle singole istruzioni della sequenza

- Il costo di una chiamata a funzione è il costo del suo corpo più il passaggio dei parametri (qualora siano a loro volta delle operazioni non elementari)
  - le funzioni ricorsive saranno trattate a parte
- La complessità di una condizione `if(guardia) blocco 1 else blocco 2` è data da $costo(guardia) + max\{costo(blocco1), costo(blocco2)\}$
- La complessità di un ciclo con un numero determinato di iterazioni `for ( i = 0; i < m; i = i + 1) corpo` è data da $\sum_{i=0}^{m-1} costo(corpo)_i + 2costante$: costo del corpo all'iterazione i, più due volte un costo costante dovuto dalla valutazione di inizializzazione/incremento e condizione
- La complessità di un ciclo con un numero indeterminato di iterazioni `while (guardia) corpo` è data da $\sum_{i=0}^{m-1} costo(guardia)_i + costo(corpo)_i + costo(guardia)_m$ dove m è il massimo numerodelle volte in cui la guardia risulta soddisfatta

Esempio di analisi strutturale

```
int min_naive(int a[], int n) {        // Costo, Num. ripetizioni
   int i, j;                           // 0 1
   bool is_min;                        // 0 1
   for (i = 0; i < n; i++) {           // 2c1 n
      is_min = true;                   // c2 n
      for (j = 0; j < n ; j++)         // 2c3 n * n
         if (a[i] > a[j])              // c4 n * n
            is_min = false;            // c5 n * n
         if (is_min)                   // c6 n
            return a[i];               // c7 1
   }
}
```

$$
\begin{aligned}
T(n) &= 2c_1 \cdot n + c_2 \cdot n + 2c_3 \cdot n \cdot n + c_4 \cdot n \cdot n + c_5 \cdot n \cdot n + c_6 \cdot n + c_7 = && (1)\\
&= n^2(2c_3 + c_4 + c_5) + n(2c_1 + c_2 + c_6) + 1(c_7) && (2)\\
&= n^2c' + nc'' + c''' = && (3)\\
&= O(n^2) + O(n) + O(1) = O(n^2) && (4)
\end{aligned}
$$

CAP 04 - SEQUENZE LINEARI E ALLOCAZIONE DINAMICA DELLA MEMORIA

SEQUENZE LINEARI

- Sono delle **strutture dati astratte** costituite da n elementi $a_0, a_1, ..., a_{n-1}$ dove a_j è il $(j+1)$-esimo elemento, $0 \leq j \leq n-1$
- E', in genere, rilevante il loro ordine relativo
- Due modalità di accesso:
  - diretto: dato j si accede solo ad a_j (**array**)
  - sequenziale: dato j è necessario accedere ad $a_0, a_1, ..., a_{n-1}$ per poi accedere ad a_j (**liste**)

ALLOCAZIONE DINAMICA DELLA MEMORIA IN C

La gestione dello heap da parte del programmatore avviene, in linguaggio C, attraverso le seguenti istruzioni:

- `void* malloc(int n)` alloca nella heap un numero di byte pari a n e restituisce l'indirizzo di memoria dello spazio allocato sottoforma di un puntatore generico
- `void free(void*)` dealloca lo spazio di memoria inizialmente puntato dal puntatore passato come parametri

Si osservi che il valore restituito è di tipo `void*` :

- questa scelta è stata fatta per rendere la funzione indipendente dal tipo di dato effettivamente allocato (la funzione riserva `n` bytes per il programma)
- per poter utilizzare proficuamente quell'area di memoria sarà necessario convertire il `void*` nel tipo opportuno (ad esempio `int*` se destinato a interi, ecc.)

Come specificare a `malloc()` di quanti byte avrò effettivamente bisogno senza imparare a memoria la dimensione di tutti i tipi di dato?

- la funzione `sizeof(<tipo_di_dato>)` restituisce il numero di byte necessari per memorizzare il tipo di dato passato come parametro (es., `sizeof(int)` restituirà il numero di byte necessari per memorizzare un intero)

Per l'elaborazione è neccessario anche poter rappresentare un puntatore non inizializzato

- la costante simbolica `NULL` , di tipo puntatore, indica un puntatore che non punta a nessuna locazione di memoria

ARRAY

Allocazione in memoria degli array

La rappresentazione di un array avviene attraverso l'indirizzo a del suo primo elemento I suoi elementi sono memorizzati in posizioni consecutive, multiple della dimensione dei dati memorizzati

- **vantaggi**: tempo di accesso costrante
- **svantaggi**: dimensione fissata, a meno del trucco di sovradimensionare l'array (dimensione *fisica* vs *logica*)

Array dinamici

Alcuni linguaggi di programmazione (C++, Java, Python, C#) prevedono array che possono essere creati (e ridimensionati) dinamicamente, cerchiamo di farlo anche in C

```
float* alloca_vettore(int n) {
   float* a = (float*)malloc(n * sizeof(float));
   assert(a != NULL);
   // la memoria potrebbe non essere allocata
   // e si dovrebbe gestire l'errore
   return a;
}

void dealloca_vettore(float* a) {
   free(a);
   a = NULL;
}
```

La funzione `alloca_vettore` restituisce “il vettore” appena allocato (attraverso il puntatore al suo primo elemento)

- ciò è lecito / possibile perché il vettore è allocato nello heap
- non sarebbe corretto se il vettore fosse dichiarato come variabile locale della funzione (e, di conseguenza, allocato sullo stack), perché quando la funzione termina la memoria viene rilasciata

L'operazione caratteristica degli array dinamici è quella di ridimensionamento:

```
float* ridimensiona_vettore(float* a, int n, int d) {
   // la funzione realloc() alloca un nuovo spazio di memoria
   // e copia i valori precedentemente memorizzati nell'array:
   // richiede tempo O(min{n, d}), pari al numero di elementi da copiare
   float* a = (float*)realloc(a, d * sizeof(float));
   assert(a != NULL);
   return a;
}
```

Il codice della `realloc` coincide, **concettualmente**, con (i) un'allocazione di un nuovo vettore, (ii) la copia dei valori precedentemente memorizzati nel vettore originale e (iii) la deallocazione del vettore originale.

Osservazione d può essere più grande o più piccolo di n a seconda che vogliamo estendere o ridurre il vettore originario

L'operazione tipica di ridimensionamento su di un array è quella di aggiunta o di eliminazione di una posizione in fondo all'array

- Aggiunta di un nuovo elemento:
  - Alloca lo spazio per un nuovo array b di dimensione $n+1$: $O(1)$
  - Copia gli elementi di a in b: $O(n)$
  - Dealloca dalla memoria: $O(1)$
  - Ridenomina b come a: $O(1)$
- Rimozione di un elemento esistente:
  - Alloca lo spazio per un nuovo array b di dimensione $n-1$: $O(1)$
  - Copia gli elementi di a in b: $O(n-1) = O(n)$
  - Dealloca a dalla memoria: $O(1)$
  - Ridenomina b come a: $O(1)$

Inefficiente: richiede tempo totale $O(n)$

Approccio "Ammortizzato"

È possibile ottenere qualcosa di più efficiente quando le operazioni di inserimento/cancellazione degli elementi in fondo all'array sono più d'una?
Idea: usiamo un sovradimensionamento come per gli array memorizzati sullo stack (lasciando degli elementi **fisici** liberi per ulteriori operazioni)

- per ciascun array necessitiamo di mantenere due informazioni dimensionali:
  - n, il numero di elementi significativi (**dimensione logica**)
  - $d > n$, il numero di elementi allocati in memoria, compresi quelli non utilizzati (**dimensione fisica**)
- in particolare decidiamo di effettuare operazioni di aumento o diminuzione raddoppiando o dimezzando la dimensione dell'array

```
// estende il vettore di un elemento
float* estendi_vettore(float a[], int* n, int* d) {
   *n = *n + 1;
   if (*n >= *d) { // raddoppio
      a = ridimensiona_vettore(a, *d, 2 * (*d));
      *d = 2 * (*d);
   }
   return a;
}

// riduce il vettore di un elemento
float* riduci_vettore(float a[], int* n, int* d) {
   *n = *n - 1;
   if (*d > 1 && n <= d / 4) { // dimezzamento
      a = ridimensiona_vettore(a, *d, *d / 2);
      *d = *d / 2;
   }
   return a;
}
```

Entrambe le operazioni anno complessità **virtualmente costante**

La logica vista nelle operazioni di estensione e riduzione del vettore dinamico richiede che per rappresentarlo sia ncessario mantenere (e gestire) contemporaneamente 3 informazioni:

- `a` il puntatore all'area di memoria dell'array (primo elemento)
- `n` la dimensione logica dell'array
- `d` la dimensione fisica dell'aray

```
float* alloca_vettore_dinamico(int n, int* d) {
   float* a = (float*)malloc(n * 2 * sizeof(float));
   assert(a != NULL)
   *d = n * 2;
   return a;
}
```

Ciò comporta che sia necessario, oltre che modificarle coerentemente, “portarsele dietro” in tutte le funzioni di manipolazione

Implementazione Ingegnerizzata

Rappresentiamo tutte le informazioni che descrivono la struttura dati attraverso un **record**, detto **descrittore**, che combina insieme tutte le informazioni necessarie

```
typedef struct /* anonima */ {
   // Puntatore allo spazio di memoria allocato per il vettore nello heap
   float *dati;
   // Dimensione logica del vettore: numero di elementi effettivi
   int dimensione;
   // Dimensione fisica del vettore: numero di elementi allocati nello heap
   int capacita;
} vettore_dinamico;
```

Convenzione: tutte le operazioni di manipolazione restituiscono il nuovo descrittore (esplicitamente o attraverso il passaggio per riferimento)

Nota: l'istruzione `typedef <tipo> <nome_alias>` crea un alias per un determinato tipo di dato consentendo di omettere `struct <nome_struct>` quando dichiariamo le variabili

Funzioni di Manipolazione (`array.h`)

```
vettore_dinamico crea_vettore_dinamico(int n) {
   vettore_dinamico v;
   // non ha senso creare vettori di dimensione negativa
   assert(n >= 0);
   if (n > 0) {
      // alloca lo spazio per i dati, memorizzandone il puntatore nel descrittore
   v.dati = (float*)malloc(2 * n * sizeof(float));
      // verifica che l'allocazione abbia avuto buon esito
      assert(v.dati != NULL);
   }
   else
   {
      v.dati = NULL;
   }
   v.dimensione = n;
   v.capacita = 2 * n;
   return v;
}


void ridimensiona_vettore_dinamico(vettore_dinamico *v, int n) {
   // non ha senso creare vettori di dimensione negativa
   assert(n >= 0);
   if (n >= v->capacita) { // raddoppia
      v->dati = (float*)realloc(v->dati, 2 * n * sizeof(float));
      // verifica che la riallocazione abbia avuto buon esito
      assert(v->dati != NULL);
      v->capacita = 2 * n;
   } else if (v->capacita > 1 && n <= v->capacita / 4) { // dimezza
      // si osservi che v->capacita / 2 == n * 2 (perché n <= v->capacita / 4)
      v->dati = (float*)realloc(v->dati, 2 * n * sizeof(float));
      assert(v->dati != NULL);
      v->capacita = 2 * n;
   } else if (v->capacita == 0 && n > 0) { // passa da zero a una dimensione diversa da zero
      v->dati = (float*)malloc(2 * n * sizeof(float));
      assert(v->dati != NULL);
      v->capacita = 2 * n;
   }
   v->dimensione = n;
}
```

```c
void elimina_vettore_dinamico(vettore_dinamico *v) {
   // elimina dati
   free(v->dati);
   v->dimensione = 0;
   v->capacita = 0;
}

void stampa_vettore_dinamico(vettore_dinamico v) {
   int i;
   for (i = 0; i < v.dimensione; i++)
      printf("%g ", v.dati[i]);
   if (v.dimensione > 0)
      printf("\n");
}
```

Considerazioni sugli Array Dinamici

- **Vantaggi**
  - Dimensione **veramente** dinamica
  - Efficienza nelle operazioni di ridimensionamento di un'unità (quelle più tipiche) che richiedono tempo costante
- **Limiti**
  - Necessità di mantenere più informazioni in modo coerente (comune però anche agli array classici)
  - Vincolo forte sul tipo di dato contenuto nell'array
    - in principio dovremmo definire un nuovo descrittore e “ricopiare il codice” delle funzioni per adattare i dati ad un altro tipo diverso da `float`

LISTE

Una lista è l'implementazione concreta di una struttura dati sequenza ad accesso sequenziale.
È rappresentata dall'indirizzo a (del primo elemento della lista, detto **nodo**) e le altre posizioni sono sparse (a causa della gestione dinamica della memoria nello heap)

Struttura:

- a: indirizzo del primo elemento a_0
- l'elemento a_j contiene il dato e l'indirizzo dell'elemento a_{j+1} (se esiste)

```c
typedef struct _nodo_lista {
   // dato contenuto nel nodo corrente
   float dato;
   // successore del nodo corrente (NULL in caso non esista)
   struct _nodo_lista *succ;
} nodo_lista;
```

- il campo `dato` contiene il dato, mentre il campo `succ` indica l'elemento successivo
- il valore `NULL` indica la fine della lista
- la lista vuota è indicata da a avente valore `NULL`

Alcune osservazioni sulla definizione:

- la `struct _nodo_lista` è **ricorsiva** (ossia utilizza, nella sua definizione, un puntatore a se stessa), pertanto non può essere definita in modo “anonimo”

- Utilizzeremo la convenzione di premettere un carattere di sottilineatura `_` ai nomi che è necessario definire ma che non ci interessano

Puntatori a `struct`

Nel caso di una variabile che contenga un puntatore a `struct` la notazione per accedere ad un elemento della struct stessa può essere semplificato

```
nodo_lista* p = ...; // supponiamo p sia un nodo della lista
(*p).dato = 5;       // accesso tradizionale (dereferenzio p e accedo al campo dato)
p->dato = 5;         // accesso semplificato (accedo al campo dato puntato da p)
```

Terminologia

- Un **nodo** della lista, detto anche **elemento**, è una variabile del tipo `struct _nodo_lista` memorizzato sullo heap
- Il primo elemento della lista (a_0) è detto **testa** della lista e rappresenta il punto di accesso alla lista stessa
- L'ultimo elemento della lista (a_{n-1}) è detto **coda** della lista, talvolta può essere gestito esplicitamente anche se non è indispensabile
- Dato un nodo a_j, il nodo a_{j+1}, raggiungibile attraverso il puntatore `succ` è detto **successore** del nodo a_j
  - il successore **non esiste** nel caso $j = n - 1$, in tal caso il puntatore è `NULL`
- Il nodo a_{j-1}, se $j > 0$, è detto **predecessore** del nodo
  - **non esiste** predecessore della testa

Operazioni di Accesso

Accesso ad a_j: scandire i primi j elementi, iniziando da a_0 e accedere via via al successivo:
tempo totale $O(j+1)$ (qualora si possa partire da a_{j-k} il costo è $O(k)$)

```
nodo_lista* elemento_lista(nodo_lista* t, int j) {
   // questo frammento di codice può essere usato anche fuori dalla funzione
   int i = 0;
   nodo_lista *c = t; // t è la testa della lista, c è il nodo corrente
   while (i < j && c != NULL) {
      i++;
      c = c->succ;
   }
   // se i >= j vuol dire che la lista non aveva almeno j elementi
   // in tal caso c == NULL
   return c;
}
```

Caso pessimo: $O(n)$, con n lunghezza della lista

lunghezza

Per conoscere la lunghezza della lista, senza alcun'altra informazione, devo scandirla interamente contando di quanti elementi è costituita

```
int lunghezza_lista(nodo_lista* t) {
   int i = 0;
   nodo_lista *c = t; // t è la testa della lista, c è il nodo corrente
   while (c != NULL) {
      i++;
      c = c->succ;
   }
   return i;
}
```

Inserimento in testa

Inserimento in testa alla lista dell'elemento `s` (`t` è la testa precedente)

```
nodo_lista* inserisci_in_testa(nodo_lista *t t,
   nodo_lista *s) {
   s->succ = t;
   t = s;
   return t;
}
```

Tempo $O(1)$

Inserimento in coda

Inserimento in coda alla lista dell'elemento `s`:

```
nodo_lista* inserisci_lista_coda(nodo_lista *t t,
   nodo_lista *s) {
   nodo_lista* c = t;
   if (c == NULL) {
      t = s;
      return s;
   }
   // cerca l'ultimo elemento della lista
   while (c->succ != NULL)
      c = c->succ;
   // aggiungi l'elemento in coda
   c->succ = s;
   s->succ = NULL;
   return t;
}
```

Tempo $O(n)$, dovuto alla ricerca dell'elemento in coda attraverso `c`. Qualora si memorizzasse sempre anche il puntatore all'ultimo elemento della lista: $O(1)$

Descrittore

Analogamente al caso degli array, per riunire in un unico contenitore tutte le informazioni relative ad una lista utilizziamo un descrittore che rappresenta la lista e verrà modificato dalla funzioni di manipolazione delle liste

```
typedef struct {
   // per la manipolazione della lista e l'accesso sequenziale
   nodo_lista *testa;
   // per rendere più efficiente l'eventuale inserimento in coda
   nodo_lista *coda;
   // per rendere più efficiente l'interrogazione sulla lunghezza della lista
   int lunghezza;
} lista;
```

Convenzione: tutte le operazioni di manipolazione di liste *restituiscono* (direttamente o per passaggio per riferimento) il nuovo descrittore alla lista

Creazione dei nodi e del descrittore

```
nodo_lista *crea_nodo(float dato) {
   nodo_lista *n = (nodo_lista*)malloc(sizeof(nodo_lista));
   n->dato = dato;
   n->succ = NULL;
  return n;
}

lista crea_lista() {
   lista l;
   l.testa = NULL;
   l.coda = NULL;
   l.lunghezza = 0;
   return l;
}
```

Inserimento in testa e in coda con il descrittore

```
void aggiungi_in_testa(lista *l, float dato) {
   nodo_lista *n = crea_nodo(dato);
   if (l->lunghezza == 0)
      l->coda = n;
   n->succ = l->testa;
   l->testa = n;
   // dobbiamo mantenere la coerenza
   // anche di questo dato
   l->lunghezza++;
}

void aggiungi_in_coda(lista *l, float dato) {
   nodo_lista *n = crea_nodo(dato);
   if (l->lunghezza > 0) {
      l->coda->succ = n;
   } else {
      l->testa = n;
   }
   l->coda = n;
   // dobbiamo mantenere la coerenza
   // anche di questo dato
   l->lunghezza++;
}
```

Eliminazione in testa con il descrittore

```
void elimina_in_testa(lista *l) {
   nodo_lista *n = l->testa;
   // se la lista è vuota non c'è
   // nulla da eliminare
   if (l->lunghezza == 0)
      return;
   // altrimenti sposta i puntatori
   l->testa = l->testa->succ;
   // aggiorna la lunghezza
   l->lunghezza--;
   // elimina il nodo
   elimina_nodo(n);
   // mantieni la coda coerente
   if (l->lunghezza == 0)
      l->coda = NULL;
}
```

Eliminazione in coda con il descrittore

```
void elimina_in_coda(lista *l) {
   nodo_lista *c = l->testa, *n = l->coda;
   if (l->lunghezza == 0)
      return;
   // cerca il predecessore della coda
   // che diventerà la nuova coda
   if (l->lunghezza == 1) {
      l->testa = NULL;
      l->coda = NULL;
   }
   else
   {
      while (c->succ != n)
         c = c->succ;
      l->coda = c;
      l->coda->succ = NULL;
   }
   l->lunghezza--;
   elimina_nodo(n);
}
```

Proprietà e stampa

```
int lunghezza(lista l) {
   return l.lunghezza;
}

bool lista_vuota(lista l) {
   return l.lunghezza == 0;
}

void stampa_lista(lista l) {
   nodo_lista *n = l.testa;
   while (n != NULL) {
      printf("%g ", n->dato);
   n = n->succ;
   }
   if (l.lunghezza > 0)
      printf("\n");
}
```

Eliminazione della lista e dei nodi

```
void elimina_lista(lista *l) {
   nodo_lista *n = l->testa;
   while (n != NULL) {
      nodo_lista *s = n->succ;
      elimina_nodo(n);
      n = s;
   }
   l->testa = NULL;
   l->coda = NULL;
   l->lunghezza = 0;
}

void elimina_nodo(nodo_lista *n) {
   free(n);
   n = NULL;
}
```

Considerazioni sulle Liste

- Vantaggi
  - dimensione dinamica senza necessità di sovradimensionamento
  - possibilità di inserimento di un nuovo elemento o di eliminazione di un elemento esistente in qualunque punto della sequenza -a differenza dei vettori nei quali posso solo aggiungere ed eliminare alla fine del vettore
- Svantaggi
  - tempo di accesso dipendente dalla posizione dell'elemento
  - assimmetria nelle operazioni di accesso sequenziali per indici crescenti (facile, $O(1)$) o decrescenti (costosa, $O(n^2)$)

LISTE DOPPIE

Uno degli svantaggi della lista (semplice) è la necessità di ripartire dalla sua testa qualora sia necessario trovare il predecessore di un elemento

- ad esempio per effettuare un inserimento in coda o l'eliminazione di un nodo arbitrario

Ovviamo a questa limitazione con una struttura leggermente diversa in cui viene mantenuto anche il puntatore all'elemento precedente Ciò consente lo spostamento, seppur sequenziale, da qualunque nodo in entrambe le direzioni (avanti e indietro)

La struttura è analoga a quella della lista semplice, il descrittore è praticamente uguale, cambia la descrizione del singolo nodo

```
typedef struct _nodo_lista_doppia {
   float dato; // dato contenuto nel nodo corrente
   struct _nodo_lista_doppia* succ; // successore del nodo corrente
   struct _nodo_lista_doppia* pred; // predecessore del nodo corrente
} nodo_lista_doppia;

typedef struct {
   nodo_lista_doppia* testa;
   nodo_lista_doppia* coda;
   int lunghezza;
} lista_doppia;
```

Operazioni (`liste.h`)

Sostanzialmente simili a quelle della lista semplice, con la differenza di dover mantenere la coerenza anche del puntatore all'elemento precedente, ad esempio:

```
void aggiungi_in_testa_d(lista_doppia *l, float dato) {
   nodo_lista_doppia *n = crea_nodo_d(dato);
   if (l->lunghezza == 0)
   l->coda = n;
   if (l->lunghezza > 0)
      l->testa->pred = n; // anche il predecessore va mantenuto coerente
   n->succ = l->testa;
   l->testa = n;
   l->lunghezza++;
}
```

Complessità

Operazione	**Lista Semplice (senza puntatori aggiuntivi)**	**Lista Semplice (con puntatori aggiuntivi)**	**Lista Doppia**
Creazione	$O(1)$	$O(1)$	$O(1)$
Inserimento in testa	$O(1)$	$O(1)$	$O(1)$
Inserimento in coda	$O(n)$	$O(1)$	$O(1)$
Ricerca di un elemento	$O(n)$	$O(n)$	$O(n)$
Inserimento in un punto arbitrario	$O(n)$	$O(1)$	$O(1)$
Eliminazione in testa	$O(1)$	$O(1)$	$O(1)$
Eliminazione in coda	$O(n)$	$O(n)$	$O(1)$
Distruzione	$O(n)$	$O(n)$	$O(n)$

CAP 05 - ALGORITMI DI ORDINAMENTO

Il problema dell'ordinamento

Data una **sequenza** di n elementi e una loro relazione d'ordine $\leq$, disporre gli elementi **nell'array** in modo che risultino ordinati secondo la relazione $\leq$
Alcuni commenti:

- La relazione d'ordine non è necessariamente quella crescente e dipende dal tipo di dato contenuto nel vettore, per ora considereremo interi e la relazione $\leq$ su di essi
- La sequenza non è necessariamente contenuta in un array (ipotesi necessaria ora per le vostre conoscenze attuali)

SELECTION SORT

Idea: al passo i seleziona l'elemento di rango i ossia il minimo tra i rimanenti $n - i$ elementi e scambialo con l'elemento in posizione i

```
void selection_sort(int a[], int n) {
   int i, indice_minimo;
   for (i = 0; i < n - 1; i++) {
      indice_minimo = minimo_a_partire_da(a, n, i);
      scambia(&a[i], &a[indice_minimo]);
   }
}

int minimo_a_partire_da(int a[], int n, int i) {
   int j, m = i;
   for (j = i + 1; j < n; j++) {
      if (a[j] < a[m])
         m = j;
   }
   return m;
}
```

A meno di componenti di costo costante, al passo i-esimo il costo del corpo del `for` è pari al costo $t(i, n)$ della chiamata della funzione `minimo_a_partire_da()`

- esso non è costante ma dipende anche da i (oltre che da n)
- dunque il costo totale è pari a $\sum_{i=0}^{n-2} t(i, n) + O(1)$

Il costo della funzione $t(i, n)$ in dipendenza di i è proporzionale al numero di iterazioni del ciclo (più l'assegnamento esterno) quindi $t(i, n) = O(n - 1)$

Pertanto:

$$T(n) = \sum_{i=0}^{n-2} O(n-1) + O(1) = \tag{5}$$

$$= O(\sum_{i=0}^{n-2} n - 1) = \tag{6}$$

$$= o(\sum_{i=0}^{n-1} i) = \tag{7}$$

$$= O(n^2) \tag{8}$$

A causa del calcolo del minimo fra gli elementi rimasti anche la complessità nel caso migliore è $O(n^2)$ (e quindi anche nel caso medio)

INSERTION SORT

Idea: al passo i-esimo inserisci l'elemento in posizione i al posto giusto tra i primi elementi (già ordinati)

```
void insertion_sort(int a[], int n) {
   int i, j, prossimo;
   for (i = 1; i < n; i++) {
      // estrae il prossimo elemento da inserire
      prossimo = a[i];
      // j è la posizione candidata all'inserimento
      j = i;
      // verifica se la posizione corrente
      // è qeuella giusta
      while (j > 0 && a[j - 1] > prossimo) {
         // altrimenti fai spazio
         a[j] = a[j - 1];
         j = j - 1;
      }
      a[j] = prossimo;
   }
}
```

Per poter inserire in un qualunque punto fra gli elementi già ordinati devo fare spazio (non posso modificare un vettore creando un elemento in un punto qualunque)
Il ciclo `while`, se necessario, sposta gli elementi verso destra per fare spazio al prossimo elemento da inserire

Al passo i del `for` esterno il costo è, praticamente, dominato dal costo $t(i)$ del ciclo `while` interno

- Il ciclo `while` richiede al massimo $i+1$ iterazioni, ciascuna di costo costante
  - $t(i) = O(i+1)$ per il ciclo `while`
    In totale: $\sum_{i=0}^{n-1} O(i+1) = O(\sum_{i=0}^{n-1} i+1) = O(\frac{n(n+1)}{2}) = O(n^2)$

Osservazione: l'algoritmo richiede solo $O(n)$ operazioni quando l'array è già ordinato

- In generale, è possibile provare che l'algoritmo richiede tempo $O(nk)$ se ciascun elemento si trova al più a distanza dalla sua posizione nell'array ordinato (quindi parecchio efficiente per array quasi ordinati)

BUBBLE SORT

Idea: confrontare gli elementi a coppie e fare salire i valori più grandi verso la fine dell'array (e scendere quelli più piccoli verso l'inizio)

```
void bubble_sort(int a[], int n) {
   int i, k = n - 1;
   bool scambio = true;
   while (scambio) {
      scambio = false;
      for (i = 0; i < k; i++)
         if (a[i] > a[i + 1]) {
            scambia(&a[i], &a[i + 1])
            scambio = true;
         }
      k = k - 1;
   }
}
```

All'iterazione $i = 1, ..., n$ del ciclo `while` l'elemento di posizione $n - 1 = k$ sarà salito nella sua posizione definitiva

- la porzione di vettore compresa fra gli indici k e $n - 1$ è ordinata
- Al passo k del ciclo `while` il costo del suo corpo $t(k)$ è dominato dal ciclo for interno che richiede tempo $O(k)$

Il corpo del ciclo `while` può essere eseguito al più n volte (per $k = n - 1, ..., 0$) quindi in totale

$$T(n) = \sum_{i=0}^{n-1} t(k) = \sum_{i=0}^{n-1} O(k) = O(\sum_{i=0}^{n-1} k) = O(\frac{n(n+1)}{2}) = O(n^2)$$

Nel caso di un array già ordinato, il numero di operazioni svolte è $O(n)$, corrispondenti all'esecuzione del ciclo `for` che determina che nessuno scambio è necessario e in tal caso il ciclo `while` viene eseguito una volta sola.

RIEPILOGO DEGLI ALGORITMI

Algoritmo	Idea	Caso Pessimo	Caso Ottimo
Selection sort	Cerco (seleziono) l'elemento minimo fra quelli rimasti da ordinare e lo scambio con l'elemento corrente	$O(n^2)$	$O(n^2)$
Insertion sort	Cerco di inserire l'elemento corrente fra quelli precedenti, già ordinati	$O(n^2)$	$O(n)$
Bubble sort	Effettuo più passaggi facendo affiorare gli elementi più grandi, finché non sono necessari più scambi	$O(n^2)$	$O(n)$

LIMITE INFERIORE DI COMPLESSITA' DELL'ORDINAMENTO

Queal è, nel caso peggiore, il numero minimo di operazioni richieste da un qualuque algoritmo di ordinamento basato sul confronto di elementi?

- Il cuore degli algoritmi di ordinamento sono le operazioni di confronto, contiamo quindi tali operazioni
- Consideriamo un qualunque algoritmo di ordinamento A che usa confronti tra coppie di elementi

in t confronti (passi, operazioni), A può discernere al più 2^t situazioni distinte:

- sono infatti possibili due risposte per ogni confronto: $a_i \leq a_j$, oppure $a_i > a_j$

Il numero possibile di ordinamenti di n elementi è $n!$ (tutte le loro permutazioni)
Poichè A deve discernere tra $n!$ possibili situazioni, deve valere $2^t \geq n!$, pertanto, risolvendo in t:

$$t = \log 2^t \geq log n! = \log O(n^n) = O(\log n^n) = O(n \log n)$$

Dunque $t = \Omega(n \log n)$ è un limite inferiore alla complessità di qualunque algoritmo di ordinamento basato sui confronti

CAP 06 - STRUTTURE LINEARI: PILE E CODE

PILE

Le pile sono collezioni di elementi in cui le operazioni disponibili, come l'estrazione di un elemento, sono ristrette unicamente a quello più recentemente inserito

- Politica di accesso **Last In First Out (LIFO)**: l'ultimo elemento inserito è il primo ad essere estratto
- Operazioni:
  - `Push(p, d)` : inserisce un nuovo elemento in cima alla pila

- `Pop(p)` : estrae l'elemento in cima alla pila (opzionalmente restituendo l'informazione in esso contenuta)
- `Top(p)` : restituisce l'informazione contenuta nell'elemento in cima alla pila
- `Empty(p)` : verifica se la pila è vuota

PILE: IMPLEMENTAZIONE CON ARRAY

- Elementi della pila memorizzati in un array di dimensione iniziale predefinita
- Array ridimensionato per garantire che la dimensione sia proporzionale al numero di elementi effettivamente nella pila
- Elementi memorizzati in sequenza nell'array a partire dalla locazione iniziale, inserendoli man mano nella prima locazione disponibile
- La cima della pila corrisponde all'ultimo elemento della sequenza

PILE: IMPLEMENTAZIONE CON LISTA

- Elementi della pila memorizzati in una lista ordinata per istante di inserimento decrescente
- La cima della pila corrisponde all'inizio della lista
- Le operazioni agiscono tutte sull'elemento iniziale della lista

CODE

Collezioni di elementi in cui le operazioni disponibili, come l'estrazione di un elemento, sono ristrette unicamente a quello inserito meno recentemente

Politica di accesso **First In First Out (FIFO)**: il primo elemento inserito è il primo ad essere estratto

- Operazioni:
  - `Enqueue(q, d)` : inserisce un nuovo elemento in fondo alla coda
  - `Dequeue(q)` : estrae l'elemento in testa alla coda (opzionalmente restituendo l'informazione in esso contenuta)
  - `First(q)` : restituisce l'informazione contenuta nell'elemento in testa alla coda senza estrarlo
  - `Empty(q)` : verifica se la coda è vuota o meno

CODE: IMPLEMENTAZIONE CON ARRAY

- Elementi della coda memorizzati in un array di dimensione iniziale predefinita
- Array ridimensionato per garantire che la dimensione sia proporzionale al numero di elementi effettivamente nella coda
- Elementi memorizzati in sequenza nell'array a partire dalla locazione iniziale, inserendoli man mano nella prima locazione disponibile

CODE: IMPLEMENTAZIONE CON LISTE

- Nodi concatenati e ordinati in modo crescente secondo l'istante di inserimento
- Il primo nodo della sequenza corrisponde alla "testa" della coda ed è il nodo da estrarre nel caso di un `Dequeue`
- L'ultimo nodo corrisponde al "fondo" della coda ed è il nodo a cui concatenare un nuovo nodo, inserito mediante `Enqueue`

CODE DI PRIORITA'

- Collezioni di elementi i cui a ogni elemento è associato un valore (priorità) appartenente a un istante totalmente ordinato
- Estensione della coda: le operazioni sono le stesse della coda: `Empty` , `Enqueue` , `First` e `Dequeue`
- `First` restituisce l'elemento di priorità massima (o minima)

CODE DI PRIORITA': IMPLEMENTAZIONE CON LISTE

- Prima soluzione: *lista non ordinata*
  - La `Enqueue` richiede tempo costante, con i nuovi elementi inseriti a un estremo della lista
  - La `Dequeue` e la `First` richiedono tempo $O(n)$: perchè è necessario individuare l'elemento di priorità massima all'interno della lista

- Seconda soluzione: *lista ordinata*
  - La `Dequeue` e la `First` richiedono tempo costante: l'elemento di massima priorità si trova in testa alla lista
  - La `Enqueue` richiede tempo $O(n)$: i nuovi elementi vanno inseriti alla posizione corretta rispetto all'ordinamento

Il costo delle operazioni è sbilanciato: vi è la necessità di un implementazione che lo renda più simile

CAP 07 - ALBERI

Gli slaberi sono una struttura gerarchica, generalizzazione delle liste: più successori per ciascun nodo, detti figli.

- **Nodo**: elemento dell'albero
- **Arco**: collegamento tra due nodi
- **Radice**: nodo che si trova al livello più elevato della gerarchia
- **Foglia**: nodo che non ha figli
- **Nodo interno**: nodo intermedio fra la radice e le foglie
- **Profondità**: distanza (in numero di archi) fra qualunque nodo e la radice
- **Altezza**: profodintà massima delle foglie

Nodi

Contenitori di informazione, senza perdinta di generalità possono contenere dei dati di tipo `float`

Alberi generici, k-ari, binari

Gli alberi sono classificati attraverso il numero di figli

- nel caso generale il numero di figli è arbitrario
- altrimenti se è limitato al valore k li chiamiamo alberi k-ari
- se $k = 2$ si dicono *binari* e i figli sono denominati `Sinistro(n)` e `Destro(n)`

Aberi Binari: Implementazione

```
typedef struct _nodo {
   float dato;
   struct _nodo* sinistro;
   struct _nodo* destro;
   struct _nodo* padre; /* opzionale */
}
```

Gli alberi binari sono interamnete descritti da due puntatori, al figlio sinistro e al figlio destro del nodo corrente (con valore `NULL` se essi non esistono)

Alberi completi e bilanciati

Un albero (binario) è detto **completo** se ogni nodo ha esattamente due figli, non vuoti

Un albero (binario) è detto **completamente bilanciato** se, oltre ad essere completo, tutte le foglie hanno la stessa profondità

Un albero (binario) di altezza h è detto **completo a sinistra** se i nodi di profondità minore di h formano un albero completamente bilanciato e se i nodi di profondità h sono tutti accumulati a sinistra

L'altezza h di un albero completamente bilanciato o completo a sinistra con n nodi è $O(\log n)$

HEAP TREE

Un **Heap tree** è un albero binario completo a sinistra che soddisfa la seguente proprietà, detta **heap property**:

1. è un albero vuoto, oppure se `r` è la sua radice e `v_s`, `v_d` i suoi figli vale `r->dato` $\geq$ `v_s->dato` e `r->dato` $\geq$ `v_d->dato`
2. la proprietà vale ricorsivamente per i sottoalberi radicati in `v_s` e `v_d`

Rappresentazione Implicita in Array

Un albero binario completo a sinistra con n nodi può essere rappresentato in un array t di dimensione n con le seguenti convenzioni:

1. `t[0]` è il nodo radice
2. Dato un qualunque altro nodo i dell'albero, `Sinistro(i) = 2 * i + 1` e `Destro(i) = 2 * i + 2`, `Padre(i) = (i - 1) / 2`

Code di Priorità Implementate Tramite Heap Tree

Il fatto che l'altezza di uno heap tree sia logaritmica ci consente di effettuare le operazioni `Enqueue` e `Dequeue` in tempo $O(\log n)$

`Enqueue`:

1. Inseriamo un nuovo `n` come foglis, mantenendo lo heap tree completo a sinistra
2. Confrontiamo la priorità del nodo `n` con quella del padre e scambiamo i due nodi se la *heap property* non vale (ovvero se `n` ha priorità maggiore di quella del padre)
3. Ripetiamo il punto 2 finchè non giungiamo ad un nodo per cui la *heap property* è soddisfatta oppure quando `n` è diventato la radice

- Tempo $O(h) = O(\log n)$, il numero di passi necessari per salire dalla foglia alla radice

`Dequeue`:

1. Estraiamo la radice (l'element di priorità maggiore) e la scambiamo con la foglia più a destra `n` nell'ultimo livello (per mantenere lo heap tree completo a sinistra)
2. Confrontiamoo la priorità del nodo `n` con quella dei suoi figli e scamnbiamo `n` con il figlio di priorità massima se la *heap property* non vale (ovvero se `n` ha priorità inferiore di quella dei figli)
3. Ripetiamo il punto 2 finchè o la *heap property* è soddisfatta oppure `n` è sceso fino ad una foglia

- Tempo $O(h) = O(\log n)$, il numero di passi necessari a far scendere la nuova radice fino alla foglia

Complessità delle Operazioni

L'esecuzione di k operazioni `Empty`, `First`, `Enqueue` o `Dequeue` su un heap tree contenente inizialmente m elementi richiede tempo $O(k \log n)$ dove $n = m + k$

ALGORITMO DI ORDINAMENTO TRAMITE HEAP TREE (HEAPSORT)

Idea: uso lo stesso schema del **selection sort** ma sfruttando una struttura dati più efficiente per ottenere il massimo di un insieme di valori: usando un heap tree (ovvero una coda di priorità tramite heap tree)

Schema concettuale:

- Costruisco un heap tree con i valori del vettore aggiungendone gli elementi uno alla volta (dove il valore di priorità coincide con il dato dell'array. Tramite `Enqueue`)
- finchè ci sono elementi estraggo il massimo dall'heap tree e lo inserisco nella sua posizione definitiva nell'array ordinato (tramite `Dequeue`)

Heapsort: complessità

- n operazioni di `Enqueue` nell'albero, ciascuna delle quali ha costo $O(\log i) \subseteq O(\log n)$, dunque questa parte cosa $n \cdot O(\log n) = O(n \log n)$

- n operazioni di `Dequeue` nell'albero, ciascuna delle quali ha costo $O(\log i)$, pertanto anche questa parte costa $O(n \log n)$
- La complessità complessiva dell'algoritmo è dunque $O(n \log n)$, ossia Heapsort è un algoritmo di ordinamento **ottimo**

CAP 08 - IL PARADIGMA DIVIDE ET IMPERA

Questo paradigma è una metodologia generale per la soluzione di problemi che fa uso della ricorsione ed è strutturato in tre fasi_

- **decomposizione**: identificazione di un numero di problemi dello stesso tipo, ciascuno definito su di un insieme di dati di dimensione inferiore a quello di partenza
- **ricorsione**: soluzione ricorsiva di ciascun sottoproblema fino ad ottenere sottoproblemi di dimensioni tali da poter essere risolti direttamente
- **ricombinazione**: combinazione delle soluzioni dei sottoproblemi per fornire una soluzione al problema di partenza

TEOREMA FONDAMENTALE DELLE RICORRENZE (MASTER THEOREM)

Data la ricorrenza

$$T(n) = \begin{cases} O(1) & \text{se } n \leq n_0 \\ \alpha T(\frac{n}{\beta}) + O(f(n)) & \text{se } n > n_0 \end{cases}$$

con $f(n)$ non decrescente, $\alpha \geq 1$ e $\beta > 1$

Se esistono due costanti positive $\gamma > 0$ e n_0' tali che $\alpha f(\frac{n}{\beta}) = \gamma f(n)$ per ogni $n \geq n_0'$, allora la relazione di ricorrenza ha le seguenti soluzioni:

1. $$T(n) = O(f(n)) \text{ se } \gamma < 1$$
2. $$T(n) = O(f(n) \log_\beta n) \text{ se } \gamma = 1$$
3. $$T(n) = O(n^{\log_\beta \alpha}) \text{ se } \gamma < 1$$

CAP 09 - DIZIONARI E LE LORO IMPLEMENTAZIONI

I dizionari sono strutture dati che ci consentono di rappresentare **collezioni di elementi indicizzabili**, ovvero delle ricorrenze † fra un dato detto **chiave** e il **valore** ad essa associato.
Possono rappresentare:

- degli "array sparsi", con indici non necessariamente ontigui, ad esempio un array che contiene i soli elementi `a[3]`, `a[5]`, `a[8]` ma non gli altri
- delle corrispondenze con indici di tipo diverso dagli interi, ad esempio qualcosa che corrisponde a `a["mario rossi"]`
- degli "insiemi" qualora il solo campo chiave sia significativo

† in matematica, i termini corrispondenza o mappa sono sinonimi di funzione

N.B.: Anche le sequenze lineari $a_0, a_1, ...$ rappresentano delle corrispondenze, però:

- le chiavi sono vincolate ad essere esattamente gli interi $0, 1, ..., n-1$
- tutti gli elementi compresi fra 0 e $n-1$ sono **obbligatoriamente presenti** nella corrispondenza

DIZIONARI: STRUTTURA DEL DATO

Assumiamo che i dati siano composti da elementi con campo `chiave` e da un insieme di informazioni *satellite* (che dipendono dall'applicazione)

- siamo interessati all'operazione di **ricerca** del dato corrispondente alla **chiave** e al reperimento delle relative informazioni satellite

- il tipo di dato della chiave, denotato di seguito con U, può essere arbitrario

```
typedef struct {
   int chiave;
   float dato;
} elemento_dizionario;
```

oppure

```
typedef struct {
   char* chiave;
   struct dati dato;
} elemento_dizionario;
```

DIZIONARI: OPERAZIONI DI BASE

Dato un dizionario `d`, sono definite le seguenti operazioni:

- `ricerca(d, k)`: cerca e restituisce (un puntatore al)l'elemento `e` tale che `e->chiave` $= k \in U$ (restituisce `NULL` se non esiste)
- `inserisci(d, k, dato)`: inserisce un nuovo elemento `e` in cui `e->chiave` $= k \in U$ e `e->dato = dato`, nell'ipotesi che $k \in U$ non sia presente nell'insieme di chiavi attualmente memorizzate (chievi distinte e **univoche**)
  - se k è già presente possiamo decidere che la funzione aggiorni il dato oppure non faccia nulla, segnalando l'errore
- `cancell(d, k)`: elimina dal dizionario l'elemento con chiave k
- `appartiene(d, k)`: restituisce `true` se un elemento con chiave k compare nel dizionario (`ricerca(d, k) != NULL`)

Se il tipo di dato delle chiavi possiede un ordine $\leq$ sono possibili anche le seguenti operazioni:

- `predecssore(d, k)`: restituisce l'elemento la cui chiave è immediatamente precedente al valore k nell'ordinamento (l'elemento di chiave k, se essa non esiste)
- `successore(d, k)`: restituisce l'elemento la cui chiave è immediatamente successiva al valore k nell'ordinamento (l'elemento di chiave k, se essa non esiste)
- `intervallo(d, k1, k2)`: restituisce la sequenza di elementi le cui chiavi k sono comprese in `k1` $\leq k \leq$ `k2`
- `rango(d, k)` restituisce il numero di elementi la cui chiave viene prima di k nell'ordinamento

DIZIONARI: IMPLEMENTAZIONE CON SEQUENZE LINEARI

Come già visto, un dizionario può essere implementato attraverso una sequenza lineare (liste, liste doppie o array dinamici) modificando opportunamente il tipo di dato memorizzato (usando `elemento_dizionario` invece di `float`)

E' possibile, opzionalmente, mantenere gli elementi della sequenza **ordinati** per chiave, qualora fossero necessarie le operazioni aggiuntive

- Gli elementi della sequenza possono essere mantenuti **ordinati per chiave**, con impatti differenti sulla complessità delle operazioni a seconda della struttura utilizzata:
  i. Implementazione con Array Dinamici
     - L'ordinamento migliora la complessità della **ricerca** da $O(n)$ a $O(\log n)$, grazie alla possibilità di utilizzare la **ricerca binaria**.
     - L'inserimento diventa più costoso: per mantenere l'ordinamento, è necessario spostare gli elementi per fare spazio al nuovo elemento, con una complessità pari a $O(n)$.
     - **Vantaggi:** La ricerca è più efficiente rispetto a un array non ordinato.
  ii. Implementazione con Liste (o Liste Doppie)
     - Anche con gli elementi ordinati, la ricerca rimane $O(n)$, poiché una lista non consente l'accesso diretto agli elementi; ogni elemento deve essere visitato sequenzialmente.

    - L'inserimento richiede sempre $O(n)$, dato che è necessario attraversare la lista per trovare la posizione corretta.
    - **Svantaggi:** Non vi è alcun miglioramento nella ricerca rispetto alle liste non ordinate.
- in ogni caso, però, semplifica l'implementazione delle operazioni `predecssore(d, k)`, `successore(d, k)` e `intervallo(d, k1, k2)`

ALBERI BINARI DI RICERCA (ABR)

- È un albero vuoto, oppure se `r` è la sua radice vale:
    - `r->sinistro->chiave` < `r->chiave`
    - `r->chiave` < `r->destro->chiave`
- La proprietà vale ricorsivamente per i sottoalberi radicati in `r->sinistro` e `r->destro`.

Assunzione: Un solo nodo con una data chiave.

Questo perchè:

- **Evita ambiguità nella ricerca**: Se esistessero più nodi con la stessa chiave, non sarebbe chiaro quale nodo restituire per un'operazione di ricerca.
- **Migliora l'efficienza**: L'unicità delle chiavi garantisce che ogni operazione (inserimento, eliminazione, ricerca) abbia una complessità ben definita, senza richiedere gestione aggiuntiva di duplicati.
- **Modella un dizionario**: L'ABR è spesso utilizzato per implementare un dizionario, dove una chiave rappresenta univocamente un elemento.

Visite su Alberi Binari di Ricerca

Nel caso di un albero binario di ricerca, la visita `simmetrica` elabora gli elementi in ordine di chiave crescente

```
int ordine; //uso una variabile globale

void stampa_ordine(nodo_albero* r) {
   printf("(%d, %d)", ordine, r->chiave);
   ordine++;
}

void visita_simmetrica(nodo_albero* r, void(*elabora)(nodo_albero*)) {
   if (r == NULL)
      return;
   visita_simmetrica(r->sinistro, elabora);
   elabora(r);
   visita_simmetrica(r->destro, elabora);
}

ordine = 1;
visita_simetrica(albero.radice, stampa_ordine);
```

Ricerca negli Alberi Binari di Ricerca

Algoritmo con complessità $O(h)$ con h altezza dell'albero e dunque $O(\log n)$ se completo

```
nodo_albero* ricerca(nodo_albero* r, int chiave) {
   if (r == NULL)
      return NULL;
   if (chiave == r->chiave)
      return r;
   else if (chiave < r->chiave)
      return ricerca(r->sinistro, chiave);
   else
      return ricerca(r->destro, chiave);
}
```

Inserimento negli Alberi Binari di Ricerca

Segue la struttura dell'albero e inserisce il nuovo nodo come foglia. L'assunzione è che vi sia un solo nodo con una data chiave, pertanto in caso sia già presente l'operazione aggiorna il dato

Algoritmo con complessità $O(h)$ con h altezza dell'albero

```
nodo_albero* inserisci(nodo_albero* r, int chiave, float dato) {
   if (r == NULL)
      return crea_nodo_albero(chiave, dato);
   else if (chiave < r->chiave)
      r->sinistro = inserisci(r->sinistro, chiave, dato);
   else if (chiave > r->chiave)
      r->destro = inserisci(r->destro, chiave, dato);
   else
      r->dato =dato;
   return r;
}
```

N.B.: Per la struttura dell'operazione di inserimento, l'albero potrebbe risultare particolarmente sbilanciato (e dunque $h = O(n)$, annullando così il beneficio di avere un albero al posto di una sequenza)

Cancellazione negli Alberi Binari di Ricerca

Operazione immediata se operata su un nodo con un solo figlio (sostituisce il nodo con il figlio), più complessa nel caso di un nodo con entrambi i figli.

Algoritmo con complessità $O(h)$

```
nodo_albero* cancella(nodo_albero* r, int chiave) {
   if (r == NULL)
      return r;
   if (r->chiave == chiave) {
      if (r->sinistro == NULL)
         r = r->destro;
      else if (r->destro == NULL)
         r = r->sinistro
      else {
         w = minimo_sottoalbero(r->destro);
         r->dato = w->dato;
         r->destro = cancella(r->destro, w->chiave);
      }
   } else if (chiave < r->chiave)
      r->sinistro = cancella(r->sinistro, chiave);
   else
      r->destro = cancella(r->destro, chiave);
   return r;
}

nodo_Albero* minimo_sottoalbero(nodo_albero* r) {
   while (r->sinistro != NULL)
      r = r->sinistro;
   return r;
}
```

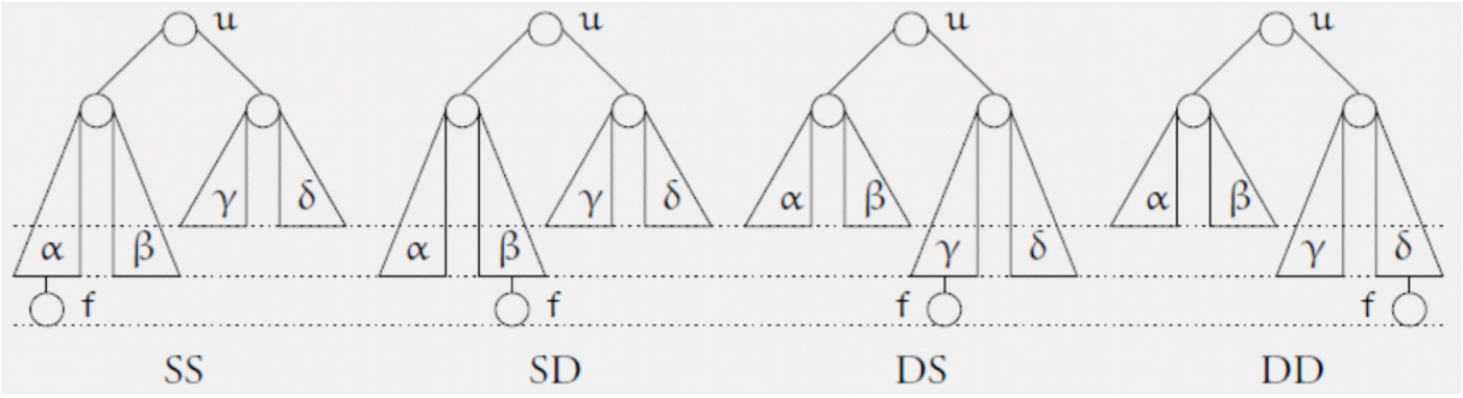
ALBERI AVL

Gli alberi AVL sono alberi **1-bilanciati**, ovvero $\left|\texttt{altezza(r->sinistro)} - \texttt{altezza(r->destro)}\right| \leq 1$

Idea di base:quando a seguito dell'inserimento l'albero si sbilancia di più di un livello, effettuando delle rotazioni per riequilibrare l'altezza

Alberi AVL: Rotazioni

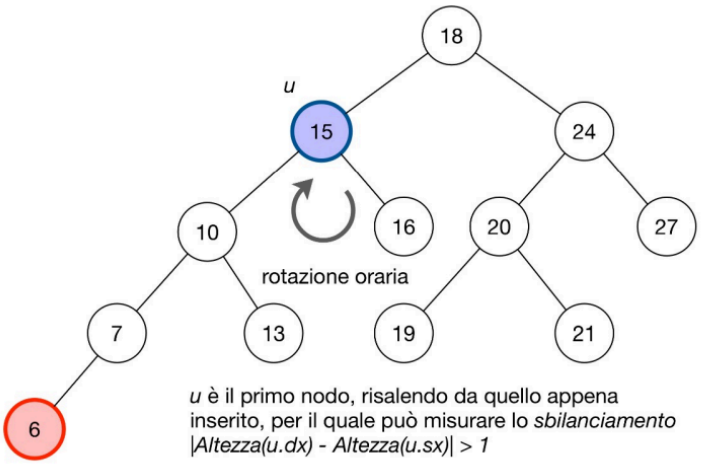
Per le rotazioni possono essere classificati 4 casi: Sinistro-Sinistro (**SS**), Sinistro-Destro (**SD**), Destro-Sinistro (**DS**), Destro-Destro (**DD**). u è il nodo critico, ovvero il nodo al livello di profondità minore per cui si rileva lo sbilanciamento

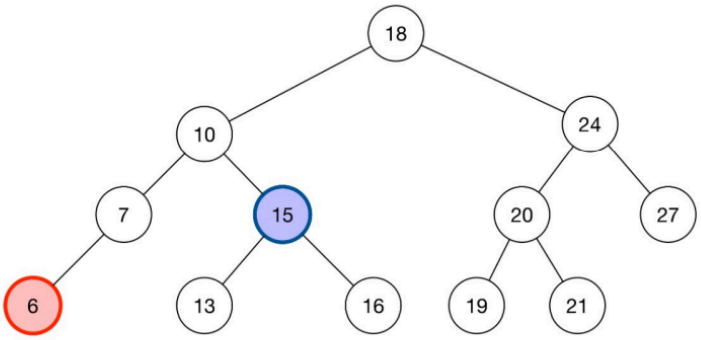


I casi sono simmetrici, quindi basterà analizzare solamente i casi SS e SD

Alberi AVL: Rotazioni, caso SS

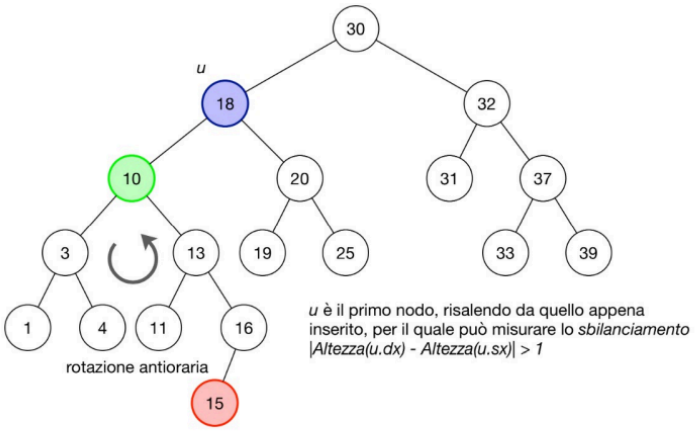
Nel caso SS, l'elemento inserito è quello con la chiave minima ed è sufficiente solamente una rotazione antioraria si u per riequilibrare l'altezza

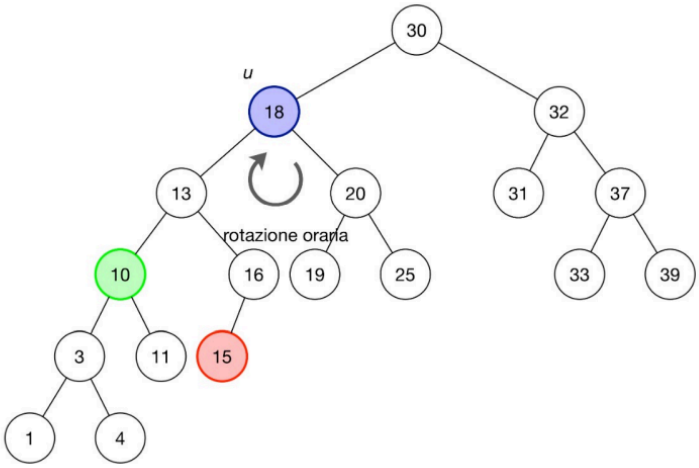


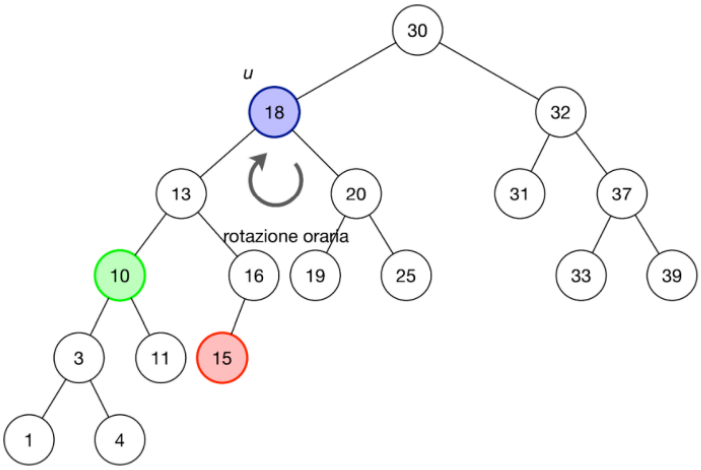


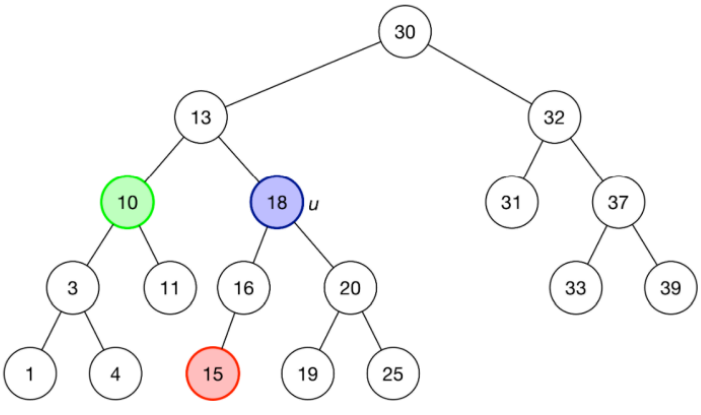
Alberi AVL: Rotazioni, caso SD

aNel caso SD, è possibile ricondurre la situazione al caso precedente (SS) attraverso una rotazione antioraria su `u->sinistro` . Una volta effettuata tale operazione è sufficiente quindi effettuare una rotazione oraria su `u` come già visto









Alberi AVL: Cancellazione

È possibile trattare il caso della cancellazione in un AVL mantenendo l'altezza bilanciata, tuttavia l'agoritmo risultante è particolarmente complesso

Usiamo un'idea alternativa per la cancellazione: utilizzare la lazy deletion, ovvero marcare logicamente un nodo come cancellato (attraverso un flag booleano) invece di eliminarlo fisicamente dall'albero

- il nodo "cancellato", deve rimanere comunque nell'albero come guida per la ricerca

Si osservi che, un numero di nodi cancellati anche se pari a metà dei nodi dell'albero, contribuisce al più di un livello (dunque un valore costante) all'altezza dell'albero stesso: possiamo scegliere tale valore come livello di soglia

Analogamente al caso degli array dinamici, quando il numero di nodi cancellati supera la metà dei nodi dell'albero, l'albero viene ricreato da zero includendo solamente i nodi non cancellati ed eliminando l'albero originale
Il costo ammortizzato di tale operazione è pari a $O(\log n)$
Infatti, prima di effettuare l'operazione di creazione di un nuovo albero si sono effettuate $\frac{n}{2}$ cancellazioni sulle quali spalmare il costo dell'operazione

- la componente di costo relativo a questa operazione è la ricerca dell'elemento contenente la chiave da eliminare, avente costo $O(\log n)$

L'operazione di copia dei nodi (non cancellati) dell'albero in un nuovo albero richiede tempo $O(n \log n)$, pertanto, ammortizzando tale costo sulle % operazioni di cancellazione otteniamo che il costo di un'operazione di cancellazione è pari a $\frac{O(n \log n)}{\frac{n}{2}} = O(\log n)$

TABELLE HASH

Funzioni Hash

Le funzioni **hash** sono delle funzioni con svariati utilizzi in informatica

- $h(m) = c$, è facile da calcolare ma è molto difficile calcolarne l'inversa $h^{-1}(c)$ per ricostruire il messaggio originale

Le funzioni hash posso no essere utilizzate per implementare un dizionario, in cui gli n elementi del dizionario sono memorizzati in un array di m elementi, ciascuno di essi alla posizione $h(k)$ (detto **tabella hash**)

Ovviamente questo funziona se $m = O(n)$ e la funzione $h(k)$ è **perfetta**, ossia non genera alcuna collisione ($k \neq k' e h(k) = h(k')$)

- In tal caso, la complessità delle operazioni è $O(1)$, posto che indichiamo con un booleano la presenza o meno di un dato nella tabella

Implementazione con Tabelle Hash con Liste di Trabocco

Nel caso generale ciò non si verifica, pertanto si utilizzno delle tabelle con **liste di trabbocco**

```
elemento_dizionario* ricerca(tabella_hash* tabella, int chiave) {
   int h = hash(chiave);
   elemento dizionario *p;
   p= ricerca_inlista(tabella[h], chiave);
   if (p != NULL)
      return p->dato;
   else
      return NULL;
}

void inserisci(tabella_hash* tabella, int chiave, float dato) {
   if (ricerca(tabella, chiave) == NULL) {
      h = hash(chiave);
      aggiungi_in_coda(tabella[h], chiave, dato);
   }
}

void cancella(tabella_hash* tabella, int chiave) {
   if (ricerca(tabella, chiave) != NULL) {
      h = hash(chiave);
      cancella_elemento_lista(tabella[h], chiave);
   }
}
```

La complessità delle operazioni è proporzionale alla lunghezza l della lista, $O(l)$

- ovviamente nel caso peggiore la funzione hash calcola lo stesso indice per tutte (o quasi) le chiavi e dunque la lunghezza è $l = O(n)$ così come la complessità

In un caso non così patologico, tuttavia possiamo assumere che la funzione di hash distribuisca in modo uniforme le chiavi fra i vari indici della tabella, in tal caso la lunghezza media della lista sarà $l = O(\frac{n}{m})$

Indicando con $\alpha = \frac{n}{m}$ il cosiddetto fattore di carico della tabella hash, possiamo esprimere la complessità delle operazioni nella forma 0(1 + a), ovvero costante a meno del fattore di carico

Tabelle Hash con Indirizzamento Aperto

Un'alternativa all'uso delle liste di trabocco consiste nel utilizzare una tabella con m elementi e più funzioni di hash alternative: se la cella della tabella risulta già occupata, si proverà con un'altra funzione a calcolare una posizione alternativa.

Nel caso dell'indirizzamento aperto è data una famiglia di m funzioni hash $h_i(k)$ che vengono provate (nell'ordine di i crescente). Ciascuna di tali funzioni devono generare una permutazione degli indici.

Esempi di famiglie di funzioni di hash:

- $h_i(k) = (h'(k) + i)\%m$ scansione lineare
- $h_i(k) = (h'(k) + ai^2 + bi + c)\%m$ scansione quadratica
- $h_i(k) = (h'(k) + i \cdot h"(k))\%m$ scansione basata su hash doppio

N.B.: A differenza dell'implementazione con liste di trabocco se la tabella è (quasi) piena non è possibile aggiungere un nuovo elemento.

Implementazione delle Tabelle Hash co indirizzamento

```
elemento_dizionario* ricerca(tabella hash* tabella, int chiave) {
   for(i = 0; i < m; i++) {
      h = hash[i](chiave);
      if (tabella[h]== NULL)
         return NULL;
      if (tabella[h->chiave == chiave])
         return tabella[h];
   }
   return NULL;
}

void inserisci(tabella_hash* tabella, int chiave, float dato) {
   elemento_dizionario* e;
   if (ricerca(tabella, chiave) == NULL) {
      i = 0;
      e = crea_elemento(chiave, dato);
      do {
         h = hash[i](chiave);
         if(tabella[h] == NULL)
            tabella[h] = e;
         i++;
      } while (tabella[h] != e);
   }
}
```

Analisi delle Tabelle Hash con Indirizzamento Aperto

Nell'analisi della complessità ci concentriamo nella misurazione delle operazioni di accesso alla tabella, assumendo che il calcolo della funzione hash richieda tempo costante.

In generale, nel caso pessimo, tutte le operazioni richiedono tempo $O(m)$, necessario a provare tutte le m funzioni di hash per trovare l'elemento cercato (eventualmente da eliminare) o una determinare una cella vuota per l'inserimento.

Cerchiamo di fare un'analisi più accurata nel caso medio
Per ciò che riguarda il numero di accessi dell'operazione di ricerca e di cancellazione in media saranno necessari $\frac{n}{m} = \alpha$ operazioni di accesso, nell'ipotesi che per ogni chiave k, $h_i(k)$ sia una delle $m!$ permutazioni degli indici generata in modo casuale e uniforme

Per l'operazione di inserimento l'analisi è leggermente più complessa.

Esprimiamo la funzione di complessità come $T(n, m)$, numero di accessi alla tabella
effettuati per trovare una cella vuota in una tabella di $m > n$ posizioni

$$T(n,m) = \begin{cases} O(1) & \text{se } n = 1 \\ 1 + T(n-1, m-1) & \text{se } n > 0 \text{ per un numero di volte pari a } \frac{n}{m} \\ 1 & \text{se } n > 0 \text{ per un numero di volte pari a } \frac{m-n}{m} \end{cases}$$

Per $T(0, m)$, la tabella è vuota e l'operazione accede direttamente ad una cella disponibile.
In caso contrario, per $m—n$ volte su m abbiamo un solo accesso quando la cella a cui si accede è disponibile, mentre per n volte su m la cella non sarà disponibile e dunque sarà necessario effettuare ulteriori $T(n—1, m—1)$ accessi (abbiamo escluso un elemento tra quelli possibili e proviamo una funzione hash in meno)

Possiamo esprimere la relazione con il valore atteso di $T(n, m)$ nel caso $n > 0$:

$$\mathbb{E}[T(n,m)] = \frac{m-n}{n} \cdot 1 + \frac{n}{m} \cdot (1 + T(n—1, m—1)) = 1 + \frac{n}{m} T(n—1, m—1)$$

E' facilmente dimostrabile per induzione su $m > n > 0$ che $T(n, m) \leq \frac{m}{m-n}$ da cui deriviamo:

$$T(n,m) \leq \frac{m}{m-n} = \frac{1}{1-\frac{n}{m}} = \frac{1}{1-\alpha} = O(1)$$

Ovvero, il numero di accessi per l'inseriemtno risulta essere, in media, costante.

CAP 10 - ALGORITMI GREEDY

Gli algoritmi "**greedy**" o "**ingordi**" sono degli algoritmi iterativi che a ciascun passo, fra un insieme di scelte, scelgono sempre la **migliore** rispetto alla situazione corrente (**migliore localmente**)

- non è detto che questa porti a una soluzione **globalmente** migliore

Questi algoritmi, in generale, costruiscono una soluzione s aggiungendone un frammento a_i alla volta.

Il problema risolto è, di solito, di **ottimizzazione**, del tipo $\min_{s \in S} f(s)$

- l'algoritmo è guidato da una funzione h : A —Y che consente di misurare l'appetibilità di ciascun frammento della soluzione e, di conseguenza, di ordinarli per valori crescenti (l'obiettivo dell'ottimizzazione è la minimizzazione)
- la funzione h è detta euristica

In alcuni casi, il criterio di ottimalità può cambiare durante la costruzione della soluzione, ad esempio, perché dipende da degli elementi già inseriti nell'insieme. In altri termini la funzione euristica dipende anche dalla soluzione finora costruita: $f : A \times S \to \mathbb{R}^{\geq 0}$

E possibile adattare lo schema algoritmico per tenerne conto, ad esempio memorizzando i frammenti della soluzione in una coda di priorità.

II valore di priorità associato a ciascun frammento ancora in coda, se potenzialmente variato a seguito della nuova soluzione, dovrà essere modificato.

SELEZIONE DELE ATTIVITA'

CAP 11 - GRAFI E PROBLEMI SU GRAFI

Rappresentano un generalizzazione degli alberi, in cui la relazione fra due nodi non è più solamente egrarchica ma può essere di qualunque altro tipo

Un grafo $G = (V, E)$ è costituito da:

- un insieme di V vertici o nodi $|V| = n$
- un insieme di $E \subseteq BV \times V$ archi, insieme di coppie $|E| = m$
  - Le possibili coppie di nodi in un grafo con n nodi sono $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$
  - il grafo è detto **sparso** se $m = O(n)$, **denso** se $m = \Theta(n^2)$

Il numero di nodi n è detto **ordine** del grafo, mentre la **dimensione** del grafo $n + m$

Un grafo di ordine n è detto **completo** qualora qualunque sua coppia di nodi sia connessa da un arco (si indica con $\mathbb{K}_n$)

Un arco del grafo e fra i nodi u e v viene rappresentato come coppia $e = (u, v)$

In generale, $(u, v) \neq (v, u)$, ovvero questo tipo di rappresentazione implica una direzionalità o asimmetria della relazione, l'arco è direttoda u a v
Un grafo con **archi dirett** viene deto **grafo diretto**

Nel caso la relazione descritta dagli archi del grafo sia simmetrica, invece, abbiamo che $(u, v) \in E \Leftrightarrow (v, u) \in E$. In tal caso siamo in presenza di un grafo indiretto

In tal caso, ho un piccolo abuso di notazione consideriamo come unico l'arco che differisce
per l'ordine dei nodi $(u, v) \equiv (v, u)$

Adiacenza/incidenza, pesi, gradi

I nodi u e v tra i quali esiste un arco $e = (u, v)$ sono detti adiacenti, l'arco e si dice incidente in u e in v

- il concetto di incidenza non guarda alla simmetria/assimetria della relazione

Un grafo è detto pesato se è definita una funzione $w : E \to \mathbb{R}$ che assegna un valore numerico ad ogni arco (detto peso).

- a esempio la distanza fra gli aeroporti può essere il peso di ciascun arco

Il grado di un nodo è il numero di archi incidenti in esso (non è legato al peso).

- Nel caso di grafi diretti si può distinguere tra grado entrante e grado uscente da u (rispettivamente, numero di archi (x, u) e (u, x))

Cammini

Un percorso fra nodi che si sposta visitando gli archi viene chiamato cammino o percorso.
Formalmente un cammino è una sequenza di $k + 1$ nodi $x_0, ..., x_k$ tale che $(x_i, x_{i+l}) \in E$ per ogni $0 \leq i \leq k$.

- kè detta lunghezza del cammino (numero di archi attraversati)
- un cammino fra due nodi u e vè un cammino in cui u e Xk v
- nel caso di grafi diretti, in generale, non necessariamente l'esistenza di un cammino da u a v garantisce l'esistenza di un cammino da v a u mentre nei grafi indiretti è sufficiente invertire la direzione in cui sono percorsi gli archi

Un cammino è detto semplice se non attraversa alcun nodo più di una volta

Due nodi per i quali esiste un cammino sono detti connessi

Un grafo in cui esiste un cammino per ogni coppia di nodi è detto connesso

Cicli,, caratterizzazione degli aleri

Un cammino in cui $x_0 = x_k$ è detto ciclo, ovvero è un cammino che, al termine, ritorna al nodo di partenza

Un grafo che non contiene cicli viene detto aciclico altrimenti è detto ciclico

Attraverso questi concetti è possibile caratterizzare gli alberi a partire dai grafi:

- è possibile stabilire l'equivalenza fra un albero e un grafo indiretto aciclico connesso con n nodi e $n—1$ archi

Cammini minimi

Un cammino minimo fra due nodi u e z è caratterizzato dall'avere lunghezza minima tra tutti i cammini possibili fra u e z

- la distanza fra due nodi u e z è la lunghezza di un cammino minimo. Se tale cammino non esiste la distanza ha valore $+\infty$

Nel caso di grafi pesati, si definisce il peso di un cammino come la somma dei pesi degli
archi attraversati dal cammino stesso 0 $\sum_{i=0}^{k-1} w(x_i, x_{x+1})$

- un cammino minimo pesato è il cammino di peso minimo fra due nodi
- la distanza pesata è il peso di un cammino minimo fra due nodi (analogamente se non esiste un cammino fra i nodi assumiamo abbia valore $+\infty$)

Grafo come struttura di dati astratta: operazioni

Creazione e Manipolazione:

- `crea_grafo(n, diretto?)` ,crea un nuovo grafo con n nodi privo di archi e diretto/indiretto a seconda del valore booleano
- `aggiungi_arco(g, u, v, w)` , aggiunge un arco (di peso w) tra u e v
- `rimuovi_arco(g, u, v)` , rimuove un arco tra u e v
- `elimina_grafo(g)` elimina il grafo

Verifica struttura:

- `esiste_arco(g, u, v)` , restituisce un valore di verità che corrisponde all'esistenza di un arco fra u e v nel grafo

Enumerazione vicinato (dipende dalla rappresentazione):

- consiste in un ciclo in cui vengono enumerati, uno alla volta, tutti i nodi adiacenti a un dato nodo u (possibilmente diretti, ovvero in uscita)

Rappresentazione Concreta dei Grafi

Assunzione: ciascun nodo è rappresentato da un indice numerico compreso fra 0 e $n-1$ (n numero totale di nodi)

Due rappresentazioni comuni:

- liste di adiacenza
- matrici di adiacenza

Rappresentazione con Liste di Adiacenza

Vettore di liste in cui l'indice dell'elemento corrisponde all'indice numerico del nodo

```
typedef struct _ nodo _ adiacenza {
   int vertice;
   float peso;
   struct nodo adiacenza* succ;
} nodo_adiacenza;

typedef struct {
   int n;
   nodo adiacenza** adiacenti;
   bool diretto;
} grafo;
```

Questo tipo di rappresentazione è particolarmente adatta per grafi sparsi, infatti in tal caso la dimensione delle liste è limitata superiormente da una costante, tipicamente non troppo grande

La complessità spaziale di rappresentazione di un grafo mediante le liste di adiacenza è $O(n+m)$

- quindi nel caso di un grafo denso (o completo) è $O(n^2)$

La lunghezza massima di una lista di adiacenza è $n—1 = O(n)$

Rappresentazione con Liste di Adiacenza: Operazioni e Costo Computazionale

- `g = crea_grafo(n)` : creazione del vettore di liste e impostazionea `NULL` di tutte le liste: $O(n)$
- `aggiungi_arco(&g, u, v, w)` : aggiunta di un elemento in coda (o in testa) alla lista: $0(1)$
- `rimuovi_arco(&g, u, v)` : rimozione di un elemento dalla lista $O(n)$
- Ricerca dell'esistenza di un arco fra due nodi:

```
esiste_arco(g, u, v):
   z = g. adiacenti [u];
   while (z != NULL) {
      if (z. vertice)
         return t rue;
      z = z.succ;
   }
   return false;
```

Complessità temporale: $O(n)$
Si può fare di meglio? Si, ad esempio mantenendo la sequenza di nodi adiacenti in un array dinamico ordinato, in tal caso la ricerca può essere svolta attraverso la ricerca binaria in tempo $O(logn)$

- Enumerazione vicinato di u :

```
z = g.adiacenti[u];
while (z != NULL) {
   /* Elabora z */
   z = z.succ;
}
```

Costo computazionale: $O(|adj(u)|) = O(n)$

- `elimina_grafo (&g)` : $O(m)$, deve rimuovere tutti gli elementi delle liste di adiacenza (pari agli archi m) e deallocare il vettore di liste $0(1)$

Rappresentazione con Matrici di Adiacenza

Matrice g di dimensione $n \times n$ in cui un valore diverso da zero per l'elemento $g[u][v]$ indica la presenza dell'arco (u, v) e il valore zero indica la sua assenza.

Il valore memorizzato può essere semplicemente un 1 nel caso di grafi non pesati o il peso dell'arco $w(u, v)$ nel caso di grafi pesati.

La complessità spaziale della memorizzazione del grafo attraverso la matrice di adiacenza è $O(n^2)$

Rappresentazione con Matrici di Adiacenza: Operazioni e Costo Computazionale

- `g = crea_g rafo(n)` : creazione della matrice e impostazione a O di tutti gli elementi: $O(n^2)$
- `aggiungi_arco(&g, u, v, w)` : impostazione a w dell'elemento (u, v) della matrice: $O(1)$
- `rimuovi_arco(&g, u, v)` : impostazione a O dell'elemento (u, v) della matrice: $O(1)$
- Esistenza di un arco (Costo: $O(1)$):

```
esiste_arco(g, u, v) {
   return g[u][v] != 0;
}
```

- Enumerazione vicinato (Costo $O(n)$):

```
for (z = 0; z < n; z++) {
   if (esiste_arco(g, u, z))
      /* Elabora vicino */
}
```

- `elimina_grafo(&g)` : deve semplicemente deallocare la matrice, $O(1)$

Comparazione delle Operazioni su Grafi nelle Diverse Implementazioni

Operazione su un grafo con n nodi, m archi	Liste di adiacenza	Matrici di adiacenza
Creazione	$O(n)$ creazione lista e impostazione a NULL	$O(n^2)$ creazione matrice e impostazione a 0
inserimento di m archi	$O(m)$	$O(m)$
Enumerazione vicinato nodo u	$O(\|adj(u)\|)$	$O(n)$
Eliminazione	$O(n+m)$ eliminazione liste	$O(1)$ deallocazione matrice

PROBLEMI SU GRAFI

Analogamente agli alberi possiamo definire dei concetti di visita. In questo caso non ha senso parlare di visita simmetrica, anticipata e posticipata ma piuttosto si parla di visita in ampiezza e in profondità.

- inoltre è necessario definire un nodo di riferimento per la partenza (non essendoci la radice)

VISTA IN AMPIEZZA

A partire da un nodo u, si visitano tutti i nodi a distanza via via crescente, ovvero prima quelli a distanza 1, poi quelli a distanza 2, e così via

- evita di esaminare ripetutamente gli stessi cammini

Nell'implementazione proposta si utilizza una coda e un vettore di booleani `raggiunto` che indica se il nodo è già stato visitato (ed elaborato) in precedenza

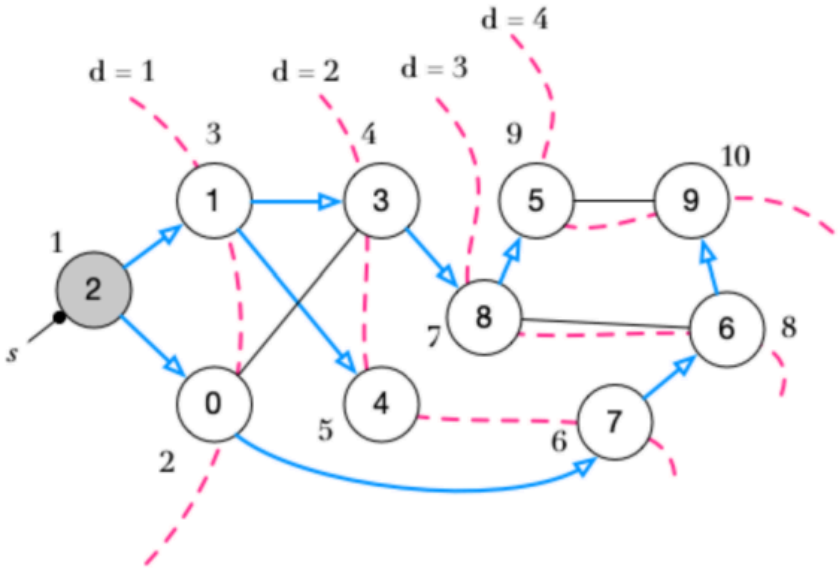
- l'uso di una coda per i nodi fa sì che questi vengano elaborati secondo una strategia FIFO e dunque per primi quelli più vicini al nodo dal quale è stato originato il cammino

```
void visita_in_ampiezza(grafo g, int s, void elabora(grafo, int)) {
   bool* raggiunto = (bool*)malloc(g.n * sizeof(bool));
   int u, v;
   // la coda è adattata per contenere degli interi
   coda_int q = crea_coda_int();

   // inizializzazione strutture ausiliarie
   for (u = 0; u < g.n; u++) {
      raggiunto [u] = false;
   }

   // partiamo dal nodo sorgente s
   enqueue_int(&q, s) ;
   raggiunto[s] = true;
   // finché ci sono nodi nella coda
   while (!empty_int(q)) {
      nodo_adiacenza* e;
      // estrai il nodo
      u = first_int(q);
      dequeue_int(&q) ;
      // applica la funzione di elaborazione al nodo corrente
      elabora(g, u) ;
      // enumera il vicinato (non raggiunto)
      PEROGNI_VICINO(g, u, e, v){
         if (!raggiunto[vl) {
            enqueue_int(&q, v) ;
            raggiunto [v] = true;
         }
      }
   }
   // eliminazione strutture ausiliarie
   elimina_coda_int (&q);
   free(raggiunto);
```

Vengono visitati (una sola volta) tutti i nodi e gli archi raggiungibili dalla sorgente

Nella figura è indicato un possibile ordine di visita di ciascun nodo a partire da $S = 2$

- l'ordine effettivo dipende da come viene enumerato il vicinato ma tutti i nodi a distanza (minima) k dalla radice vengono visitati uno di seguito all'altro



Tempo di esecuzione: $O(n+m)$

VISITA IN PROFONDITA'

A partire da un nodo u si visitano tutti i nodi lungo un cammino di distanza via via crescente. Una volta esaurito tale cammino si prosegue con il cammino successivo e così via

L'algoritmo è identico al precedente, con la sola differenza di usare una pila invece di una coda

L'uso di una pila per i nodi fa sì che questi vengano elaborati secondo una strategia LIFO e dunque per primi quelli più lontani (profondi) dal nodo dal quale è stato originato il cammino (perché scoperti più tardi)

```
void cisita_in_profondita(grafo g, int s, void elabora(grafo, int)) {
   bool* raggiunto = (bool*)malloc(g.n * sizeof(bool));
   int u, v;
   //la pila è adattata per contenere degli interi
   coda_int q = crea_pila_int();
   //inizializzazione strutture ausiliarie
   for (u = 0; u < g.n; u++)
      raggiunto[u] = falseM
   //partiamo dal nodo sorgente s
   enqueue_int(&q, s);
   raggiunto[s] = true;

   //finchè ci sono nodi nella coda
   while (!empty_int(q)) {
      nodo_adiacenza* e;
      //estrai il nodo
      u = top_int(q);
      pop_int(&q);
      // applica la funzione di elaborazione al nodo corrente
      elabora(g, u);
      //enumera il vicinato (non raggiunto)
      PEROGNI_VICINO(g, u, e, v) {
         if(!raggiunto[v]) {
            push_int(&q, v);
            raggiunto[v] = true;
         }
      }
   }
}
```

Nella figura è indicato un possibile ordine di visita di ciascun nodo a partire da $s = 2$

Inizialmente viene seguito il cammino più profondo fino al nodo di indice 4, quindi il primo nodo ancora rimasto sulla pila è il nodo di indice 5 (inserito durante l'esplorazione del vicinato di 8) e vengono seguiti i cammini più profondi a partire da tale nodo

Analogamente al caso precedente: $O(n + m)$

L'algoritmo per la visita in profondità può essere definito in maniera ricorsiva

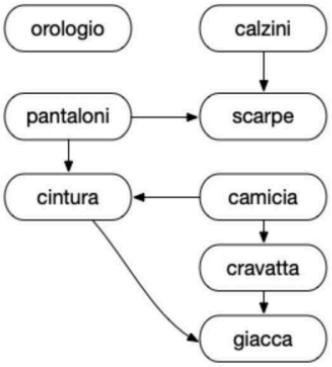
```
void _applica_in_profondita(grafo g, int u, void elabora(grafo, int), bool raggiunto[]) {
   nodo_adiacenza* e;
   int v;
   f(g, v);
   raggiunto[u] = true;
   PEROGNI_VICINO(g, u, e, v) {
      if(!raggiunto[v])
         _applica_in_profondita_ricorsiva(g, v, elabora, raggiunto);
   }
}

void applica_in_profondità_ricorsiva(grafo g, int s, void elabora(grafo, int)) {
   bool* raggiunto = (bool*)malloc(g.n * sizeof(bool));
   int uM
   for (u = 0; u < g.n; u++)
      raggiunto[u] = false;
   _applica_in_profondita_ricorsiva(g, s, elabora, raggiunto);
   free(raggiunto);
}
```

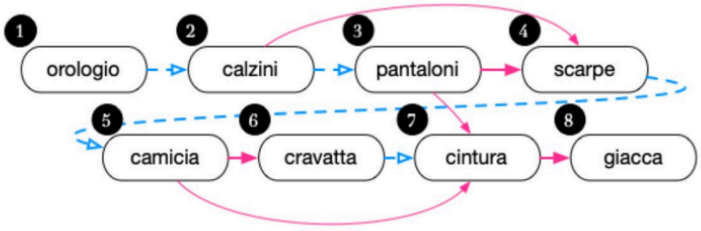
GRAFI DIRETTI ACICLICI (DAG)

Un grafo diretto aciclico può rappresentare la relazione precedenza tra una serie di attività

La visita in profondità opportunamente adattata permette di definire un ordinamento dei nodi del grafo (nel caso sia diretto e aciclico)

Restituiremo l'ordine delle attività memorizzato in una pila





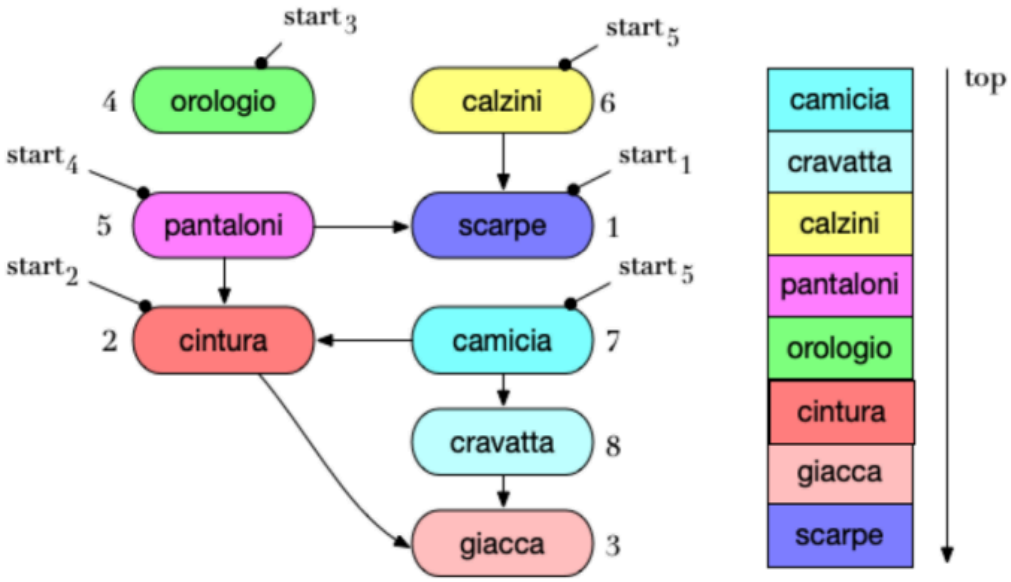
Grafi Diretti Aciclici: Implementazione

```
pila_int ordinamento_topologico(grafo g) {
   bool* raggiunto = (bool*)malloc(g.n * sizeof(bool));
   pila_int ordine = crea_pila_int();

   int s;
   for (s=0; s < g.n; s++)
      raggiunto[s] = false;

   for (s=0; s < g.n; s++)
      if (!raggiunto[s])
         _visita_in_profondita_ricorsiva_ordina(g, s, raggiunto, &ordine);

   free(raggiunto);
   return ordine;
}

void _visita_in_profondita_ricorsiva_ordine(grafo g, int u, bool raggiunto[], pila_int* p_ordine) {
   nodo_adiacenze* e;
   int v;
   raggiunto[u] = true;
   PEROGNI_VICINO(g, u, e, v) {
      if (!raggiunto[v])
         _visita_in_profondita_ricorsiva_ordine(g, v, raggiunto, p_ordine);
   }
   push_int(p_ordine, u);
}
```

Ordinamento Topologico: Esempio

I punti da cui parte/riparte la visita in profondità sono indicati da $start_i$

È possibile dimostrare che partendo da nodi senza dipendenze si minimizza l'eventuale numero di restart



Lo stack rappresenta l'ordine delle attività determinato (ne possono esistere diversi ma equivalenti)

PROBLEMI DI CAMMINO MINIMO

Cammini (pesati) minimi

Consideriamo ora il problema di esplorare un grafo pesato per determinare il percorso di distanza più breve fra due nodi.

Dato un grafo pesato (senza perdita di generalità orientato) $G = (V, E, w)$, determinare i cammini di peso minimo, in ordine di generalità crescente:

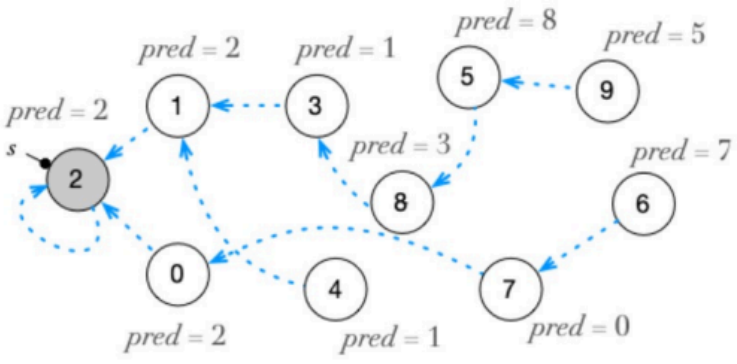
1. fra due nodi u e v
2. fra un nodo "sorgente" s e tutti gli altri nodi
3. fra uqalunque coppia di nodi

Il caso 1. può essere risolto tramite il caso 2., infatti è sufficiente scegliere $s = u$ e poi prendere il solo cammino che arriva a v. Anche il caso 3,ì. può essere risolto con il caso 2. con un ciclo su tutti i nodi sorgente

Rappresentazione dei cammini

Rappresentiamo i cammini a partire da una sorgente u come vettori i cui elementi mantengono la distanza (pesata) da u e il vertice precedente. In altre parole un cammino da u a v è rappresentato con i nodi ottenuti seguendo a ritroso il campo `pred` dell'elemento `cammini [v]` fino a raggiungere u . La sorgente è contraddistinta da avere se stessa come predecessore.

```
typedef struct {
   float distanza;
   int pred;
} cammini;
```



	π	δ
0	2	1.0
1	2	2.0
2	2	0.0
3	1	3.0
4	1	6.0
5	8	6.0
6	7	6.5
7	0	2.5
8	3	4.0
9	5	7.0

Algoritmo di Dijkstra

L'algoritmo di Dijkstra (Edsger Wybe Dijkstra, 1930-2002) è applicabile solo su grafi con pesi non negativi. Risolve il problema di determinare tutti i cammini minimi da una data sorgente attraverso una strategia greedy con un euristica dinamica $h : A \times S \to \mathbb{R}^{\geq 0}$

```
cammini* dijkstra(grafo g, int s) {
   int u, v;
   cammini* c = (cammini*)malloc(g.n * sizeof(cammini));
   coda_priorita_int q = crea_coda_priorita_int();
   for (u = 0; u < g.n; u++) {
      c[u].distanza = INT_MAX;
      c[u].pred = -1
   }
   c[s].pred = s;
   c[s].distanza = 0.0;
   for (u = 0; u < g.n; u++)
      enqueue_priorita_int(&q, u, -cammini[u].distanza);
   while (!empty_priorita_int(q)) {
      nodo_adiacenza* e;
      elemento_priorita ep = first_priorita_int(q);
      dequeue_priorita_int(&q);
      u = ep.dato;
      PEROGNI_VICINO(g, u, e, v){
         if (c[v].distanza > c[u].distanza + e->peso) {
            c[v].distanza = c[u].distanza + e->peso;
            c[v].prev = u;
            aggiorna_priorita_int(&q, v, -c[v].distanza);
         }
      }
   }
   return c;
}
```

Le strutture dati vengono inizializzate con distanze infinite e nessun predecessore tranne che per la sorgente

A ciascun passo, le stime di distanza minima (e di conseguenza il predecessore) vengono aggiornate considerando i percorsi che passano per il nodo estratto dalla coda di priorità: euristica greedy dinamica

In alcuni casi le stime possono essere riviste perchè esiste un percorso più breve passando per un altro nodo

L'algoritmo termina quando la coda di priorita è vuota e il risultato è nella tabella dei cammini

Complessità dell'algoritmo di Dijkstra

Il costo computazionale dell'algoritmo è dato principalmente da:

- tempo per la costruzione della coda di priorità t_c
- tempo di estrazione del minimo dalla coda di priorità t_e (ripetuto n volte)
- tempo di aggiornamento della priorità t_d (ripetuto m volte)
  Quindi in totale:

$$O(t_c + nt_e + mt_d)$$

Due scenari implementativi:

- Implementazione della coda di priorità mediante heap: $O((n+m)\log n)$
- Implementazione della coda di priorità mediante lista non ordinata: $O(n^2 + mn)$

MINIMO ALBERO RICOPRENTE

Il problema del minimo albero ricoprente (Minimum Spanning Tree) trova applicazioni multiple, ad esempio nella progettazione di reti di telecomunicazioni oppure nell'analisi dei dati.

Dato un grafo indiretto pesato $G = (V, E, w)$ si vuole trovare un sottografo $T = (V, E')$
con $E' \subseteq E$ che sia:

1. un albero, ovvero sia un sottografo connesso, aciclico e abbia esattamente $|E'| = n - 1$ archi
2. tra tutti i possibili alberi costruiti quello i cui archi abbiano peso minimo, ossia

$$T = \underset{T=(V,E')_{\text{è un albero}}}{\arg\min} \sum_{e \in E'} w(e)$$

L'Algoritmo di Kruskal

L'algoritmo di Kruskal si basa sull'idea di costruire incrementalmente l'albero partndo da una serie di inisiemi/alberi S_v disgiunti (costituiti inizialmente da un singolo nodo). A ciascun passo considera, in modo greedy, l'arco di peso minimo fra quelli ancora non elaborati cercando diconnettere due insiemi.

Sono possibili due casi:

- se l'arco connette due nodi appartenenti allo stesso insieme/albero può essere scartato: infatti esso chiuderebbe un ciclo e, per costruzione, essendo più pesante dei candidati già considerati non può appartenere al minimo albero ricoprente
- se l'arco invece connette due nodi u e v appartenenti a due insiemi/alberi disgiunti
- allora, per come è stato scelto, è quello di peso minimo che **attraversa il taglio** $X(S_u, S_v)$ (iltaglio fra due insiemi disgiunti di nodi S e R è costituito dagli archi che hanno un estremo in S e l'altro in R, ossia $X(S, R) = \{\{u, v\} | u \in S \wedge v \in R\}$)

```
int* kruscal(grafo g) {
   lista_archi = crea_lista_archi();
   insieme_nodi = crea_insieme_int();
   int u, v;
   coda_di_priorita_archi pq = crea_coda_di_priorita_archi();
   insieme_di_nodi s = crea_insieme_di_nodi();
   int* archi = (int*)malloc(g.n * sizeof(int));
   for (u = 0; u < g.n; u++)
      archi[u] = -1;

   for (u = 0; u < g.n; u++) {
      nodo_adiacenza* e;
      PEROGNI_VICINO(g, u, e, v) {
         enqueue(pq, {u, v}, peso(g, u, v));
      }
      crea_insieme(&s, u);
   }

   while (!empty(pq)) {
      elemento_coda_priorita_archi e = first(pq);
      dequeue(pq);
      u = e.dato.u;
      v = e.dato.v;
      if (disgiunti(s, u, v)) {
         unisci(&s, u, v);
         archi[u] = v;
         archi[v] = u;
      }
   }

   return archi;
}
```

Le strutture dati **coda di priorità** e **insieme di nodi** andrebbero adattate per rappresentare degli archi e degli insiemi disgiunti di nodi

Algoritmo di Kruskal: analisi della complessità

Ipotiziamo che gli insiemi di nodi vengano rappresentati da una lista di liste ordinate (una per ciascun insieme).

- l'inizializzazione delle strutture dati richiede tempo $O(m \log m)$ per il riempimento della coda di priorità, mentre per la struttura che mantiene gli insiemi di nodi sono necessari $O(n)$ operazioni per la creazione della lista di liste
- il ciclo ehile viene ripetuto al più m volte
  - ciascuna operazione di `dequeue` dalla coda di priorità richiedono tempo $O(\log m)$
  - ciascuna operazione di verifica della disgiunzione dei due insiemi richiede, nell'implementazione a lista, tempo $O(|l_1| + |l_2|) = O(n)$
  - ciascuna operazione di unione dei due insiemi richiede tempo $O(|l_1| + |l_2|) = O(n)$, poichè le due liste rappresentanti i due insiemi sono ordinate è possibile fonderle con una funzione simile a `merge` dell'algoritmo merge sort

Pertanto la complessità totale dell'algoritmo è $O(m \log m) + O(m \cdot n) = O(m \cdot n)$

Usando delle strutture dati per gli insiemi di nodi (union-find) la complessita totale sarebbe $O(m \log n)$