Q&A

Domande Frequenti

▼ **Fornire la specifica sintattica e semantica degli operatori *cancNodo* e *cancArco* per la struttura dati grafo**

cancellaNodo(u,G) = G'

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

cancellaArco(u,v,G) = G'

PRE:	G = (N,A)	u ∈ N, v ∈ N, (n,v) ∈ A
POST:	G' = (N, A')	A' = A \ (u,v)

▼ **Spiegare la realizzazione di *grafi* mediante *matrice di adiacenza* e *matrice di incidenza*, fornendo vantaggi e svantaggi di ognuna**

Tra le possibili realizzazioni della struttura *grafo* abbiamo:

- **Matrice di Adiacenza**
  Una matrice di adiacenza è una matrice *NxN* dove gli indici di ogni riga e colonna indicando un nodo del grafo.
  Per indicare l'adiacenza di due nodi, cioè che i due nodi sono connessi da un arco, si assegna un valore specifico alla posizione indicata dai rispettivi indici di riga e colonna.
  Ad esempio, dati due nodi *1,3*, avremo che, se questi due sono adiacenti, il valore della matrice in posizione *(1,3)* sarà *1*, altrimenti se non sono connessi da una arco, il valore sarà *0*.
  Se si vogliono memorizzare altre informazioni, come ad esempio il peso o etichetta dell'arco, si può adottare una **matrice di adiacenza estesa** che include campi per memorizzare altre informazioni in corrispondenza dell'indice di riga.
  Il vantaggio di questa rappresentazione è la facile gestione della struttura, mentre c'è un considerevole spreco di spazio che aumenta al crescere delle dimensioni della matrice, proprio perché i nodi devono essere presenti anche se non ci sono archi a connetterli.

- **Matrice di Incidenza**
  La rappresentazione con **matrice di incidenza**, invece, prevede una matrice di dimensioni *NxM*, dove gli indici di riga corrispondono sempre ai nodi, mentre sulle colonne troviamo tutti gli archi che partono o che entrano in un determinato nodo.
  Ad esempio, dato un grafo *orientato* ed una coppia di nodi *1,3* ed un arco che li connette avremo che, il valore in corrispondenza della riga di *1* e la colonna nella quale è rappresentato l'arco sarà:
  → -1 se l'arco esce da *1* ed entra in *3*;
  → +1 se l'arco esce da *3* ed entra in *1*.
  → 0 se l'arco non riguarda il nodo sulla riga
  Se il grafo non è orientato troveremo invece solo valori 0 e 1.
  Il vantaggio è che è molto semplice ritrovare le adiacenze, ma il punto negativo resta lo spreco di spazio, oltre ad una gestione leggermente più complicata della matrice di adiacenza.

▼ **Fornire in pseudocodice l'algoritmo di *ricerca di ampiezza (BFS)* per *grafi*, motivando l'utilizzo di altre strutture dati per la sua implementazione**

L'implementazione della BFS per un grafo si avvale della struttura dati "coda" per la sua caratteristica FIFO.
Infatti questo algoritmo visita prima tutti i nodi adiacenti al nodo corrente, prima di passare al prossimo nodo, e per fare questo si utilizza una coda.

I vertici, quindi, verranno visitati in ordine di distanza dal vertice di partenza.

```
BFS(G: grafo; n: nodo):
        creaCoda(Q)
        inCoda(u,Q)
        while not (codaVuota(Q)) do
                u = leggiCoda(Q)
                fuoriCoda(Q)
                esamina u e marcalo come “visitato”
                for (tutti i nodi v ∈ A(u)) do
                        esamina l’arco(u,v)
                        if (v non è marcato “visitato” e v ∉ Q)
                                inCoda(v,Q)
```

▼ Spiegare il concetto di *collisione* e le corrispondenti tecniche di gestione per *dizionari*

Una **dizionario** è una struttura dati formata da coppie <*chiave, valore*> nella quale le chiavi non possono essere ripetute.
Attraverso la chiave è possibile ritrovare la posizione della coppia; in particolare, soffermandosi sull’implmentazione con *hash table* avremo che, tramite una funzione di hashing la chiave (o parte di essa) viene elaborata per ottenere la posizione nella quale la coppia verrà memorizzata.

Una buona funzione di hashing deve minimizzare la possibilità di ottenere due posizioni uguali con chiavi diverse, ma allo stesso tempo deve essere computabile in tempo ragionevole, ragion per cui utilizzare l’intera chiave nella funzione, specie se lunga, non è accettabile.
Si deve ricorrere quindi ad un compromesso.

Ogni dizionario così implementato deve prevedere la gestione delle *collisioni*, che si verificano proprio quando la funzione di hashing genera due output uguali con chiavi diverse.

Due possibili soluzioni sono:

- **scansione lineare**: le posizioni libere sono contrassegnate con una chiave fittizia “libero” e, se esiste già una coppia nella posizione indicata dalla funzione di hashing, si procede a memorizzare la coppia nella prima posizione successiva disponibile.
  Questa soluzione, però, può dar vita ad agglomerati;
- **liste di trabocco**: ogni posizione ha una lista ad essa associata nella quale vengono memorizzate le coppie in caso di collisione.

▼ Fornire la specifica di *Problema di Ricerca*

Un **problema di ricerca** è un problema nel quale si vuole conoscere:
- una delle soluzioni ammissibili, se esistono;
- l’assenza di soluzioni, nel caso non esistano soluzioni ammissibili.

Per indicare l’assenza di soluzioni si introduce il simbolo $\perp$.

Formalmente, un problema di ricerca P è una quintupla:

$$< I, S, R, S \cup \{\perp\}, q_{ric} >$$

dove:

- I: insieme degli input, cioè le variabili in input per il problema;
- S: insieme delle soluzioni, cioè lo scopo della soluzione, che contiene tutti i valori ammissibili e che rappresentano soluzione;
- $R \subseteq I \times S$: relazione caratteristica, cioè l’insieme di vincoli tra le variabili di input e quelle dello scopo della soluzione;
- q_{ric}: quesito. q_{ric} definisce una relazione su R, identificata da R_{qric} che contiene:
  - ogni coppia dell’insieme R;
  - una coppia $< i, \perp >$ per ogni istanza i di un problema P, che indica l’assenza di soluzioni per una data istanza.

▼ Spiegare la tecnica algoritmica *Greedy*

La tecnica **greedy** rientra nelle tecniche appartenenti al paradigma generativo, cioè quel paradigma nel quale le tecniche generano le varie soluzioni (invece di ricercarle nello spazio di ricerca, come invece avviene nel paradigma selettivo).

In questa tecnica, il processo di generazione della soluzione viene diviso in *stadi*, mentre le soluzioni sono formate da componenti.

Ad ogni stadio, si aggiunge una componente alla soluzione; la scelta del valore della componente avviene in base alle condizioni in quello specifico momento: infatti l'algoritmo prenderà la scelta migliore in base a quelle disponibili, senza però tornare indietro per riconsiderare decisioni prese in passato; questa caratteristica è quello che caratterizza quest'algoritmo.

Grazie a questa caratteristica, la greedy rappresenta un buon compromesso tra qualità della soluzione e tempo di esecuzione: infatti, seppur non sempre garantisce il ritrovamento della soluzione migliore, il tempo di esecuzione sarà nettamente inferiore a quello di tecniche come il backtracking.

Per applicare questa tecnica ad un problema di ottimizzazione, però, è necessario che siano verificate alcune proprietà:

- *scelta greedy*: prendendo decisioni che sono ottimi locali si ottiene una soluzione che è ottimo globale;
- *sottostruttura ottima*: data una soluzione ottimale ad un problema, questa contiene a sua volta soluzioni ottimali ai sottoproblemi.

La tecnica greedy si presta bene ai problemi di scheduling, nei quali i task devono essere eseguiti in base ad un criterio.

▼ Fornire la specifica sintattica e semantica degli operatori *costrBinAlbero* e *cancSottoBinAlbero* per la struttura dati Alberi binari

costrBinAlbero(T,T') = T''

POST:	T'' si ottiene introducendo automaticamente un nodo radice r' (radice di T'') che avrà come sottoalbero sinistro T e come sottoalbero destro T'
se T = Λ e T' = Λ	T'' avrà solo la radice
se T ≠ Λ e T' = Λ	T'' avrà solo un sottoalbero sinistro
se T= Λ e T' ≠ Λ	T'' avrà solo un sottoalbero destro

cancSottoBinAlbero(u,T) = T'

PRE:	T ≠ Λ, u ∈ N
POST:	T' e ottenuto eliminando il sottoalbero di radice u e tutti i suoi discendenti

▼ Spiegare la tecnica algoritmica del *backtracking*

La tecnica algoritmica del backtracking rientra tra le tecniche del paradigma selettivo, nel quale le tecniche selezionano le soluzioni dallo spazio di ricerca.

Il backtracking opera dividendo il processo di ispezione dello spazio di ricerca in *n* stadi e la soluzione in *componenti*; dopo *i* stadi *(i<n)* si avrà una soluzione parziale. Questa soluzione parziale viene valutata e, se essa non viola vincoli ed è ammissibile, si continua per quel "ramo di esecuzione", altrimenti l'algoritmo torna indietro e tenta altre vie.

Questa tecnica può essere anche implementata usando degli alberi, dove sui nodi troveremo le varie componenti ed i cammini saranno i diversi "rami di esecuzione"; sarà quindi possibile aggiungere il prossimo componente alla soluzione in tanti nodi quanti sono i figli del nodo corrente.
La radice di quest'albero sarà una soluzione fittizia, mentre ogni nodo foglia sarà una soluzione ammissibile.

Questa tecnica si presta bene ad essere applicata a problemi di ricerca ma anche di ottimizzazione; in particolare, proprio per la sua caratteristica del backtracking, è adatta per la ricerca della soluzione ottimale (ovviamente usando una funzione "obiettivo" per valutare la qualità della soluzione).

Per quanto riguarda le performance, quanto appena detto ha un costo in termini di tempo di esecuzione, che risulta maggiore a quello di altre tecniche come la greedy.

▼ **Motivare le differenze fra una tecnica greedy e una di backtracking quando applicate ad un problema di ottimizzazione**

Queste due tecniche, grazie alle loro caratteristiche, forniscono particolari vantaggi e svantaggi nella risoluzione di problemi di ottimizzazione.

Una tecnica greedy, applicata a problemi di ottimizzazione, genererà una soluzione buona (anche se probabilmente non la migliore) in tempo ragionevole; questo perché, un algoritmo greedy, genera una soluzione prendendo la miglior decisione possibile ad ogni stadio, senza tornare mai indietro.
Questo garantirà che il problema venga risolto in tempo ragionevole, ma proprio a causa del fatto che le decisioni passate non vengono riconsiderate, la soluzione in output potrebbe non essere la migliore.

Un algoritmo di backtracking, invece, garantirà il ritrovamento della soluzione ottimale dal momento che stabilisce una soluzione ottima locale e la compara con tutte le possibili soluzioni, date da tutte le possibili combinazioni di elementi.
Se la soluzione parziale non sarà migliore di quella ottima locale, verrà scartata e si tornerà indietro per perseguire strade alternative.
Questo processo garantirà il ritrovamento della soluzione ottimale, ma i tempi necessari saranno di gran lunga maggiori, dal momento che la complessità non cresce linearmente al crescere del numero di elementi.

▼ **Descrivere come uno *stack* e una *coda* possano essere rappresentati mediante *vettore***

Si può rappresentare uno **stack** mediante un vettore gestendo il vettore in ordine LIFO; in pratica, l'ultimo elemento aggiunto nel vettore, dovrà essere il primo ad essere "servito" (es. se si usa l'operatore *pop* su uno stack di 5 elementi, sarà rimosso l'elemento che corrisponde all'indice *i=4*).
Conviene utilizzare il vettore aggiungendo in coda e non in testa, per evitare lo spostamento di tutti gli elementi prima dell'aggiunta di un nuovo elemento o dopo la rimozione.
Conviene inoltre tenere un puntatore di testa per lo stack, in modo da agevolare ulteriormente le varie operazioni.
Questo approccio risulta abbastanza semplice da gestire; un potenziale svantaggio può essere dato dal fatto che, se l'allocazione del vettore non è dinamica, può esserci spreco di spazio (vettore allocato con dimensione maggiore a quella necessaria) oppure lo spazio allocato potrebbe non essere sufficiente.

Allo stesso modo si può utilizzare un vettore per rappresentare una coda: si aggiungeranno gli elementi in coda al vettore e si leggeranno/rimuoveranno dalla testa.
In altre parole, bisognerà gestire il vettore con ordine FIFO.
Conviene inoltre avere un puntatore di testa ed uno di coda per agevolare ulteriormente le operazioni.
Questa rappresentazione presenta alcuni svantaggi: ogni volta che si rimuove un elemento dalla coda, tutti gli altri elementi dovranno essere spostati; presenta inoltre le stesse limitazioni che riguardano lo spazio della rappresentazione della pila con vettore.

▼ **Fornire in pseudocodice l'algoritmo di visita in *Post-Ordine* per *alberi binari***

```
visitaPostordine(r: radice; T: albero):
    if (il sottoalbero non è vuoto) do
        visitaPostordine(figlio sinistro di r, T)
        visitaPostordine(figlio destro di r, T)
        esamina il nodo r
```

▼ **Facendo riferimento ad uno specifico problema di ottimizzazione spiegare ed illustrare l'esecuzione di una strategia greedy applicata ad una istanza di quel problema**

Problema dello Zaino: dati n elementi ai quali è associato un profitto p ed un costo c, bisogna trovare la combinazione di elementi da inserire nello zaino in modo tale da massimizzare il profitto, senza però eccedere un dato budget B.

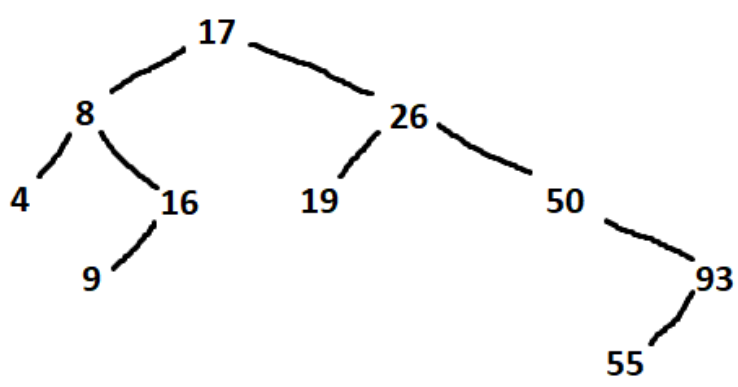
Un approccio greedy a questo problema può essere applicato come segue:

- dati un vettore degli elementi, uno dei costi ed uno pei profitti, ricavare il rapporto costo/profitto;
- ordinare i vettori in base al rapporto costi/profitto;
- ora l'algoritmo scansiona il vettore degli elementi ed inserisce l'elemento nel vettore di output X, solo se il costo dell'elemento corrente non supera il budget rimanente. Il vettore X conterrà solo valori 0/1 in base a se l'*i-esimo* elemento del vettore degli elementi è stato aggiunto o meno;
- scansionati tutti gli oggetti l'algortimo restituisce il vettore X.

E evidente che l’algoritmo utilizza una strategia greedy, in quanto prende la decisione di aggiungere l’elemento o meno senza tornare mai indietro per sostituire gli altri elementi, ma decide solo in base alla condizione attuale del budget.

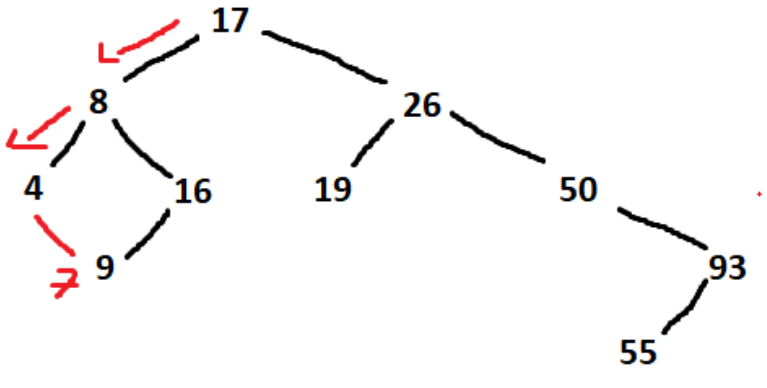
▼ **Riportare l'albero binario di ricerca corrispondente ad una coda con priorità, dopo aver inserito nell'ordine i seguenti elementi: 17, 26, 8, 50, 16, 19, 93, 4, 9 e 55.**
Poi illustrare il processo di inserimento del 7.
Dopo l'inserimento del 7 illustrare il processo di rimozione del minimo.

Inserimento dei nodi nell’ordine dato:

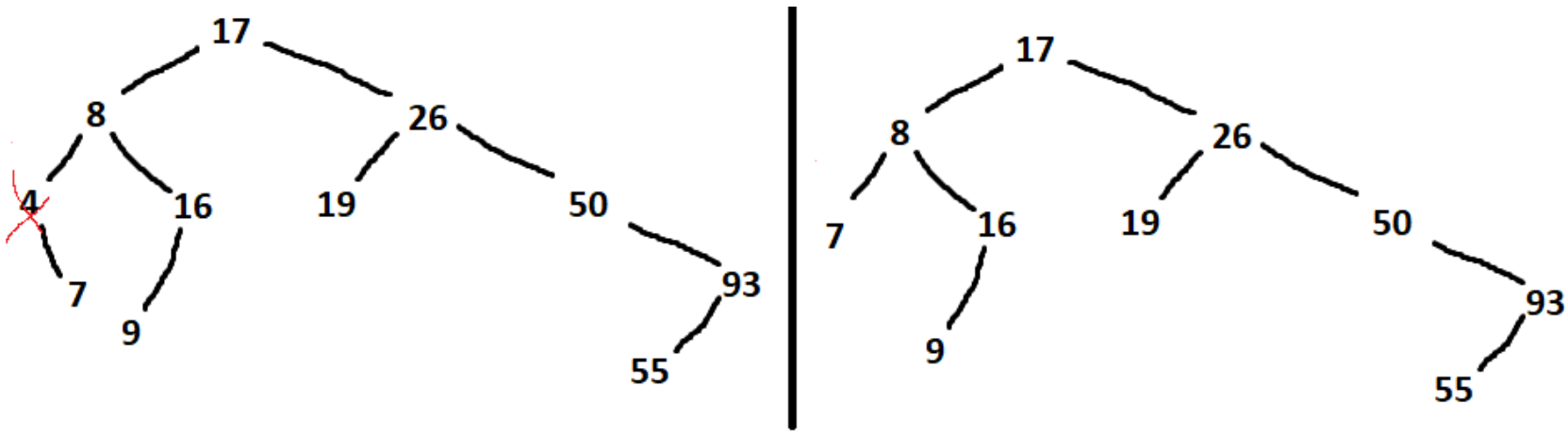


Inserimento 7:

- 7 è minore di 17, quindi si segue l’arco che porta al figlio sinistro;
- 7 è minore di 8, quindi si segue l’arco che porta al figlio sinistro;
- 7 è maggiore di 4, e 4 è foglia, allora 7 diventa figlio destro di 4.



Rimozione minimo: si segue il cammino 17 → 8 → 4.
4 è foglia, quindi è il più piccolo. 7 diventa il figlio sinistro di 8.



▼ **Spiegare la realizzazione di *alberi n-ari* mediante *vettore dei padri*, *liste di figli* e con *cursori*, fornendo vantaggi e svantaggi di ognuna**

Nella rappresentazione con *vettori di padri* si utilizza un vettore nel quale i nodi dell’albero corrispondono agli indici del vettore.
Il valore corrispondente ad ogni indice sarà, per tutti i nodi tranne che per la radice, il riferimento alla posizione del nodo padre.

Questa rappresentazione agevola la visita dei nodi che si trovano in percorsi che vanno dalla foglia alla radice, ma risulta complesso aggiungere/rimuovere sottoalberi.

Nella rappresentazione con *liste di figli*, invece, si utilizza sempre un vettore che in ogni indice contiene:

- etichetta del nodo;
- riferimento alla lista dei nodi figlio.

A loro volta, le liste di figli, conterranno il riferimento al nodo successore.
Questa rappresentazione semplifica, anche se di poco, le operazioni di che risultavano più complesse nella realizzazione con vettore di padri.

La rappresentazione con *cursori* utilizza invece un vettore (o meglio, 3 vettori) che contiene (in corrispondenza dello stesso indice):

1. riferimento al nodo figlio;
2. etichetta del nodo padre;
3. riferimento al fratello successivo.

Eventualmente può essere aggiunto anche un ulteriore cursore al nodo padre.
Questa rappresentazione permette l'implementazione della struttura in linguaggi che non supportano la gestione di puntatori, ma le operazioni risultano complesse e macchinose; la rappresentazione comporta inoltre un considerevole utilizzo di spazio.

▼ Spiegare cosa e un *problema di ottimizzazione* e quali tecniche si possono adottare per risolverlo

Risolvere un problema di ottimizzazione equivale a trovare la soluzione ottimale ad un problema, tra tutte le soluzioni ammissibili.

Formalmente, un problema di ottimizzazione è una quintupla:

$$< I, S, R, S \cup \{\perp\}, q_{ott}(M, m, \subseteq) >$$

dove:

I: insieme degli input;

S: insieme delle soluzioni;

$R \subseteq I \times S$: relazione caratteristica;

M: insieme di elementi;

m: funzione obiettivo (come ad esempio la funzione costo);

$\subseteq$: relazione di ordinamento totale su M.

Inoltre avremo R_{qott}, una relazione su $R \subseteq I \times S$ tale che, per ogni istanza i di un problema ed ogni soluzione s abbiamo:

- una coppia (i, s) che rappresenta la soluzione migliore per l'istanza;
- una coppia $(i, \perp)$ che rappresenta l'assenza di soluzioni per l'istanza.

Per risolvere un problema di ottimizzazione possiamo applicare sia tecniche del paradigma selettivo che tecniche del paradigma generativo.

Una delle possibili tecniche del paradigma selettivo è il backtracking; il backtracking suddivide il processo di ricerca della soluzione in stadi.
In ogni stadio verrà aggiunta una componente della soluzione, che verrà poi valutata in base alla soluzione migliore corrente; se questa è migliore diventerà la nuova soluzione migliore corrente, altrimenti la ricerca viene interrotta e l'algoritmo tornerà indietro per tentare strade migliori.
La ricerca si interrompe quando tutto lo spazio è stato esaminato.

Un'altra possibile tecnica da applicare è la greedy del paradigma generativo. Anche questa opera in stadi ma, al contrario del backtracking, qui le soluzioni vengono generate e non ricercate.
Questa tecnica è caratterizzata dal fatto che prende la decisione migliore in un determinato momento senza mai tornare indietro e potrebbe restituire una soluzione non ottimale, infatti rappresenta un compromesso qualità/tempo.
Per applicarla in un problema di decisione devono essere verificate due proprietà:
- scelta greedy: si può ottenere una soluzione ottima globale prendendo decisioni che sono ottimi locali;
- sottostruttura ottima: ogni soluzione ottima per un problema contiene una soluzione ottima per i suoi sottoproblemi.

▼ **Descrivere come un *albero binario* può essere utilizzato per rappresentare una *coda con priorità*, descrivendo inoltre come si effettuano le operazioni di inserimento e cancellazione di elementi**

E possibile implementare una coda di priorità attraverso l'utilizzo di alberi binari; invece però di utilizzare un albero binario senza nessun particolare ordinamento dei nodi, si utilizza una albero che soddisfa la proprietà di heap.

La proprietà di heap dice che, nell'albero (detto binary heap), i nodi figlio devono sempre avere valore maggiore/minore del nodo padre (in base a se si sta parlando di un max o min heap).

Questa struttura si presta particolarmente bene per l'implmentazione della coda, dal momento che il nodo radice conterrà sempre il valore massimo/minimo.

Ad ogni operazione di inserimento o cancellazione deve essere verificato il soddisfacimento della proprietà di heap; ciò significa che, se la proprietà è stata violata con l'aggiunta/rimozione dell'elemento, si passa ad una seconda fase nella quale questa deve essere ristabilita, scambiando i nodi padre e figlio fin quando la proprietà non risulta nuovamente soddisfatta.

▼ **Spiegare la realizzazione di alberi binari mediante *vettore* e *collegata con cursori*, fornendo vantaggi e svantaggi di ognuna**

La rappresentazione di alberi binari con **vettori** utilizza un semplice vettore per memorizzare l'abero con tutti i suoi figli. Nello specifico avremo che:

- in prima posizione si troverà la radice dell'albero;
- per ogni sottoalbero, il suo figlio sinistro si troverà all'indice $2i$;
- per ogni sottoalbero, il suo figlio destro si troverà all'indice $2i + 1$, dove i è l'indice della radice.

Questa rappresentazione è semplice da implementare, ma presenta alcuni svantaggi come lo spreco di spazio, dal momento che non tutti gli elementi del vettore conterranno un nodo dell'albero. Inoltre rimuovere sottoalberi risulta abbastanza complicato e si è soggetti alle limitazioni sullo spazio poste dal vettore.

La rappresentazione **collegata con cursori** utilizza invece un vettore dove verranno memorizzati, per ogni indice, riferimento al figlio sinistro di un nodo, il nodo in questione e riferimento al figlio destro del nodo; il riferimento sarà semplicemente l'indice del vettore dove si troverà il figlio.
Questa rappresentazione è utile quando si usano linguaggi che non permettono la gestione di puntatori, ma risulta abbastanza complicata da gestire, oltre ad avere le limitazioni dei vettori.

▼ **Fornire la specifica sintattica e semantica degli operatori *insSottoAlbero* e *insPrimoSottoAlbero* per la struttura dati Alberi n-ari**

insSottoAlbero(T, T', u) = T''
PRE: T $\neq \wedge$, T' $\neq \wedge$, u $\in$ N, u non è radice di T
POST: T'' è ottenuto da T aggiungendo il sottoalbero di radice r'

insPrimoSottoAlbero(T, T', u) = T''
PRE: T $\neq \wedge$, T' $\neq \wedge$, u $\in$ N
POST: T'' è ottenuto aggiungendo l'albero T' la cui radice r' è il nuovo primo figlio di u

▼ **Fornire in pseudocodice l'algoritmo di ricerca in profondità (DFS) per Alberi N-Ari**

Ci sono tre modi per effettuare una ricerca in profondità in un albero, cioè attraverso le modalità di visita: previsita, postvisita e invisita.
Utilizzando la previsita avremo:

```
DFS(r: radice, t: tipoelem):
    se (r è foglia)
        return;

    se (r.valore == tipoelem)
        return true,
    altrimenti
        r diventa il suo prossimo figlio
            DFS(r, t)
```

▼ **Spiegare la strategia di risoluzione per il problema della *ricerca del cammino minimo* in un *grafo* adottata da un algoritmo a scelta del candidato**

Una delle tecniche per trovare il cammino minimo in un grafo consiste nell'usare l'**algortimo di Dijkstra**.
Quest'algoritmo può essere usato con grafi orientati e pesati (con pesi positivi).

L'algoritmo usa:

- una *coda di priorità* per memorizzare coppie <*indice, distanza*>;
- un *vettore delle distanze*, nel quale i nodi corrispondono agli indici dell'array.

Prima di iniziare la ricerca l'algoritmo inizializza il vettore delle distanze a $+\infty$ per tutti i nodi tranne che per il nodo sorgente (che verrà inizializzato a 0), che sarà il nodo dal quale verranno calcolate le distanze; la scelta di questo nodo è a discrezione dell'utente.
Si inserisce inoltre la coppia <*indice nodo sorgente, 0*> nella coda, che sarà il primo elemento ad essere usato.

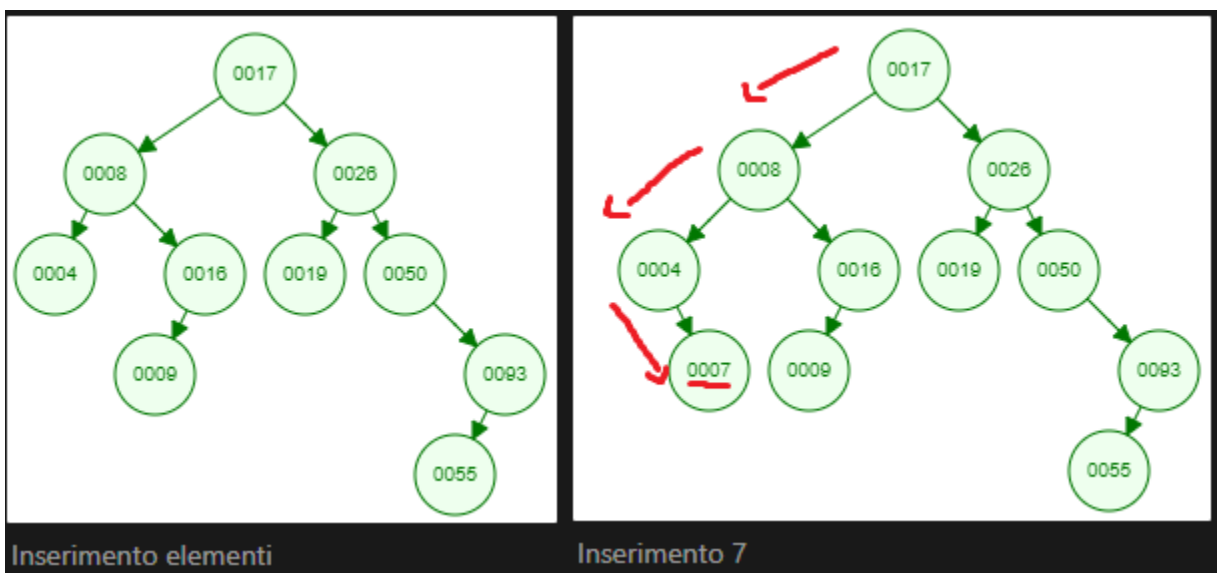
L'utilizzo della distanza $+\infty$ assume che tutti i nodi siano irraggiungibili inizialmente; ciò significa che, se alla fine del processo ci sono nodi ancora con quel valore, questi sono irraggiungibili.

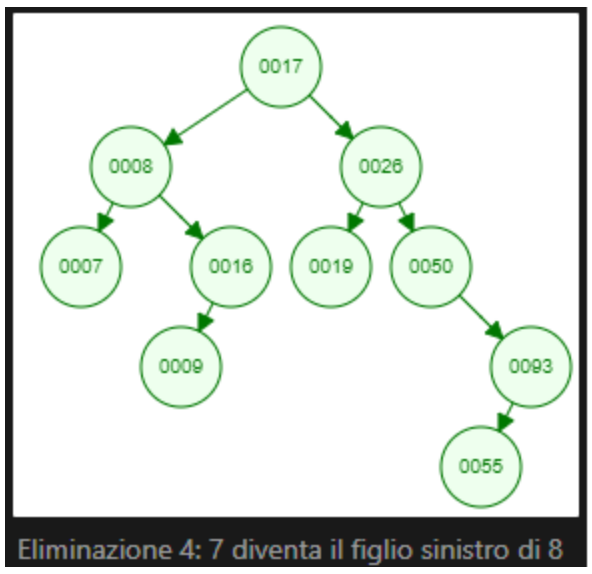
L'algoritmo inizia la ricerca e segue questi steps:

1. Legge la coppia <*indice, distanza*> dalla coda (essendo una coda di priorità sceglierà quella con chiave/distanza minore.
   Se il valore *distanza* della coda è maggiore a quello contenuto nell'array la coppia viene rimossa e si sceglie la prossima, altrimenti si prosegue;
2. Sceglie un vicino del nodo e calcola la distanza del percorso dal nodo letto dalla coda al vicino e se questo è inferiore di quello nell'array lo aggiorna; inserisce inoltre la coppia <*indice vicino, distanza*> nella coda.
   Prosegue fin quando tutti i vicini sono stati visitati.
   Terminato questo step rimuove la coppia letta dalla coda.
3. Ripete gli steps *1* e *2* fin quando la coda risulta vuota.

L'algoritmo restituirà in **output** il vettore contente i costi/distanze per tutti i cammini a partire dal nodo sorgente. In base all'implementazione sarà possibile anche mostrare i cammini relativi alle distanze nel vettore.

▼ **Riportare l'*albero binario di ricerca* corrispondente ad una *coda con priorità* dopo aver inserito nell'ordine i seguenti elementi: 17, 26, 8, 50, 16, 19, 93, 4, 9 e 55.**
Poi illustrare il processo di inserimento del 7.
Dopo l'inserimento del 7 illustrare il processo di rimozione del minimo.





Eliminazione 4: 7 diventa il figlio sinistro di 8

▼ **Fornire in pseudocodice l'algoritmo di *ricerca di profondità (DFS)* per *grafi***

DFS(G: grafo; u: nodo):
esamina il nodo u e marcalo "visitato"
for(tutti i nodi v ∈ A(u)) **do**
esamina l'arco(u,v)
if (v non è marcato "visitato")
DFS(v,G)

▼ **Fornire la specifica di Problema di Ottimizzazione**

Risolvere un *problema di ottimizzazione* consiste nel trovare la soluzione ottimale ad un dato problema.

Formalmente, un problema di ottimizzazione è una quintupla:

$$< I, S, R, S \cup \{\perp\}, q_{ott}(M, m, \subseteq) >$$

dove:

- M è un insieme qualsiasi;
- m è la funzione obiettivo (come ad esempio una funzione costo);
- $\subseteq$ è una relazione di ordinamento totale su M
- q_{ott} definisce una relazione definita su $R \subseteq I \times S$ detta R_{qott} tale che, data una soluzione S ed un istanza i, avremo:
  - una coppia $< i, S >$ per la quale non esiste soluzione migliore per l'istanza;
  - una coppia $< i, \perp >$ che rappresenta l'assenza di soluzioni per l'istanza.

▼

▼ **Facendo riferimento ad uno specifico *problema di ricerca* spiegare ed illustrare l'esecuzione di una strategia di *backtracking* applicata ad una istanza di quel problema.**

Uno dei *problemi di ricerca* ai quali può essere applicato il *backtracking* è il *problema* delle *n-regine.*
Il problema è il seguente: date *n* regine, in una scacchiera *nxn*, posizionare le regine in modo tale che non si attacchino tra loro.

Riepilogo Alberi

Alberi Binari e Proprietà

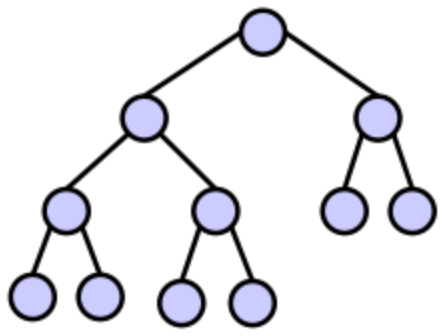
Un **albero binario** si dice **completo** se

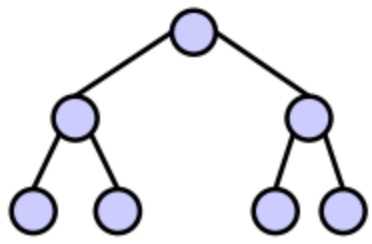
- tutte le foglie hanno la stessa profondità *h;*
- tutti i nodi interni hanno 2 figli.

Possiamo inoltre dire che, un **albero binario completo** di altezza h ha $2^{h+1} - 1$ nodi.

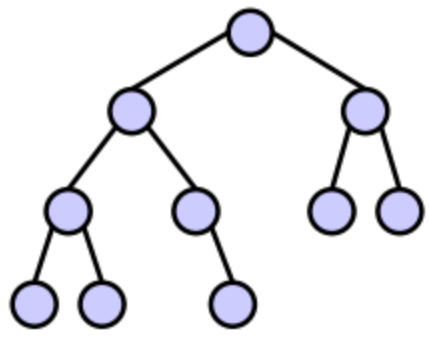
Un albero binario si dice **quasi completo** se tutti i livelli, tranne al più l'ultimo, sono completi; nell'ultimo livello possono mancare alcune foglie consecutive a partire dall'ultima foglia a destra.

Un albero binario si dice **bilanciato** quando la differenza di altezza tra sottoalbero sinistro e destro è al più di 1, ed i due sottoalberi sono bilanciati.





Alberi quasi completi



Albero NON quasi completo

Binary Heap

Un *binary heap* è un binary tree che è completo/quasi completo e obbedisce alla proprietà di heap.
La proprietà di heap dice che il valore chiave del nodo padre deve essere sempre maggiore/minore del valore chiave dei nodi figlio, in base a se è un max o min heap.

Dopo le operazioni di inserimento e cancellazione di un nodo, bisogna verificare che la proprietà di heap non venga violata dal nuovo nodo; se questa viene violata è necessario ristabilirla scambiando padri e figli fino a quando la proprietà non è nuovamente soddisfatta.

Binary Search Tree

Gli **alberi binari di ricerca** sono strutture dati che supportano molte operazioni sugli insiemi dinamici.

Questi alberi non sono altro che alberi binari (con massimo due figli per nodo/sottoalbero) nei quali:

- il **figlio sinistro** è minore o uguale al genitore;
- il **figlio destro** è maggiore del genitore.

Questa proprietà si estende a tutto l'albero, cioè tutti i nodi nel sottoalbero sinistro (dalla radice) dovranno essere minori della radice, e tutti quelli del sottoalbero destro dovranno essere maggiori.