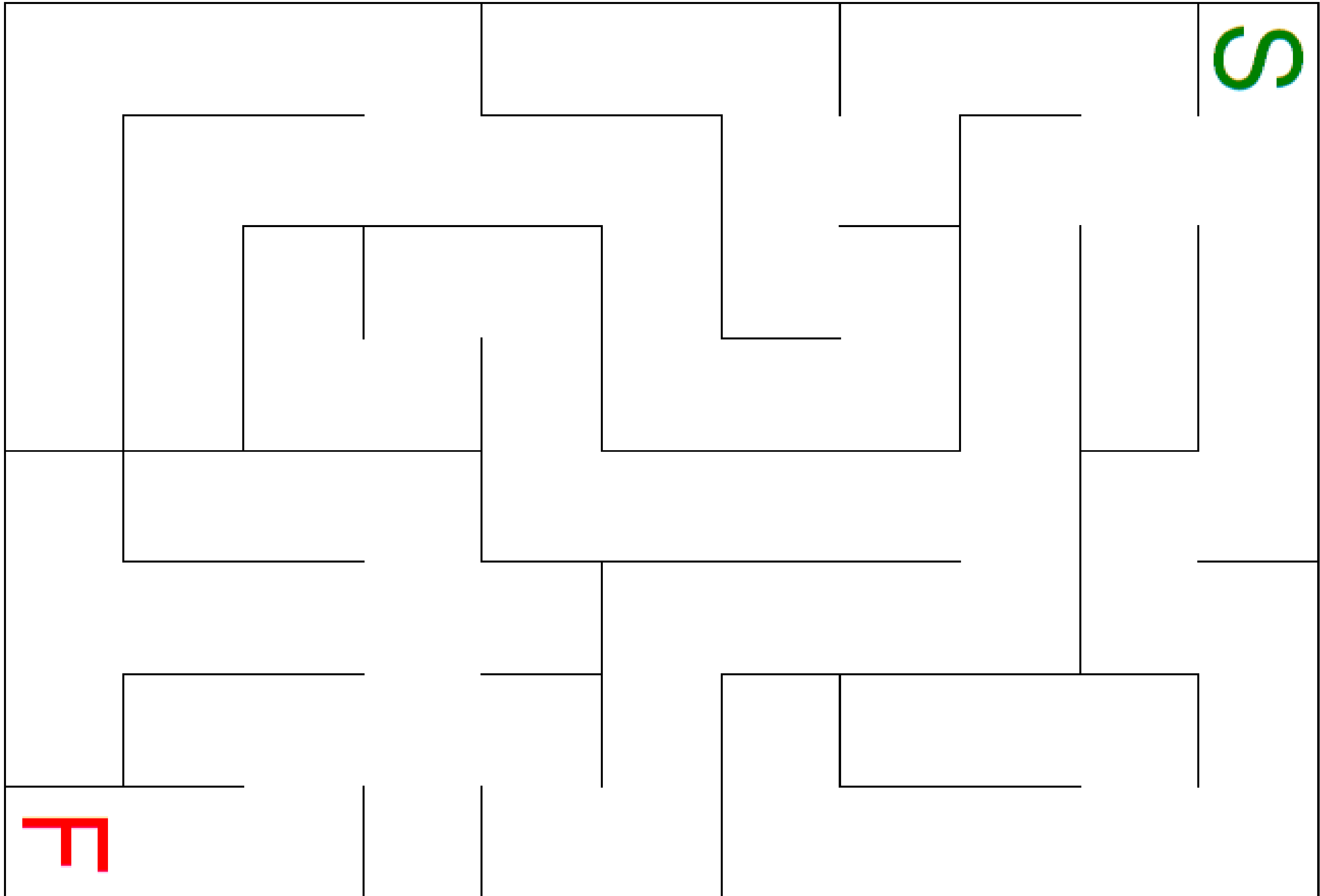
Progetto Labirinto

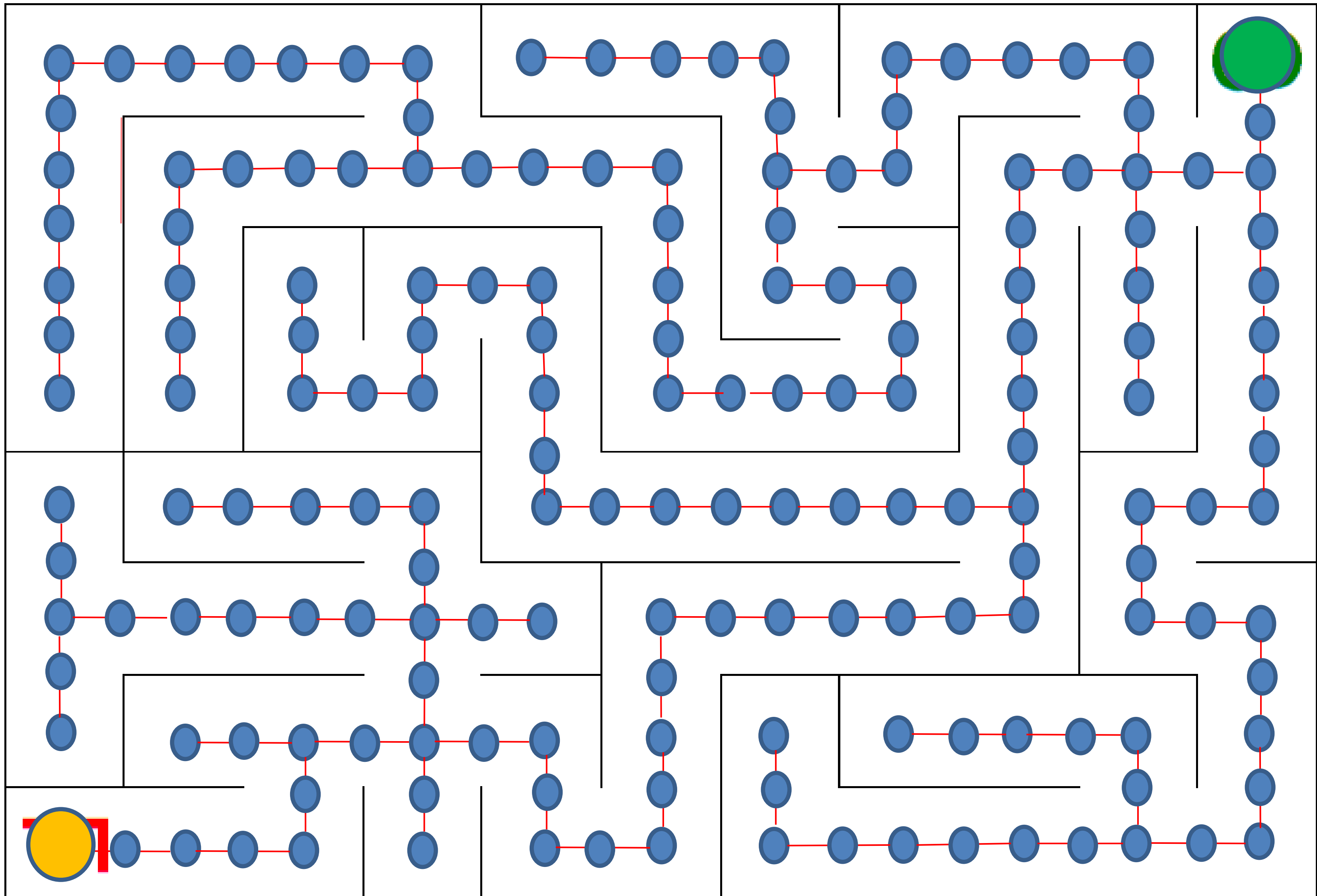
Laboratorio di

Algoritmi e Strutture Dati

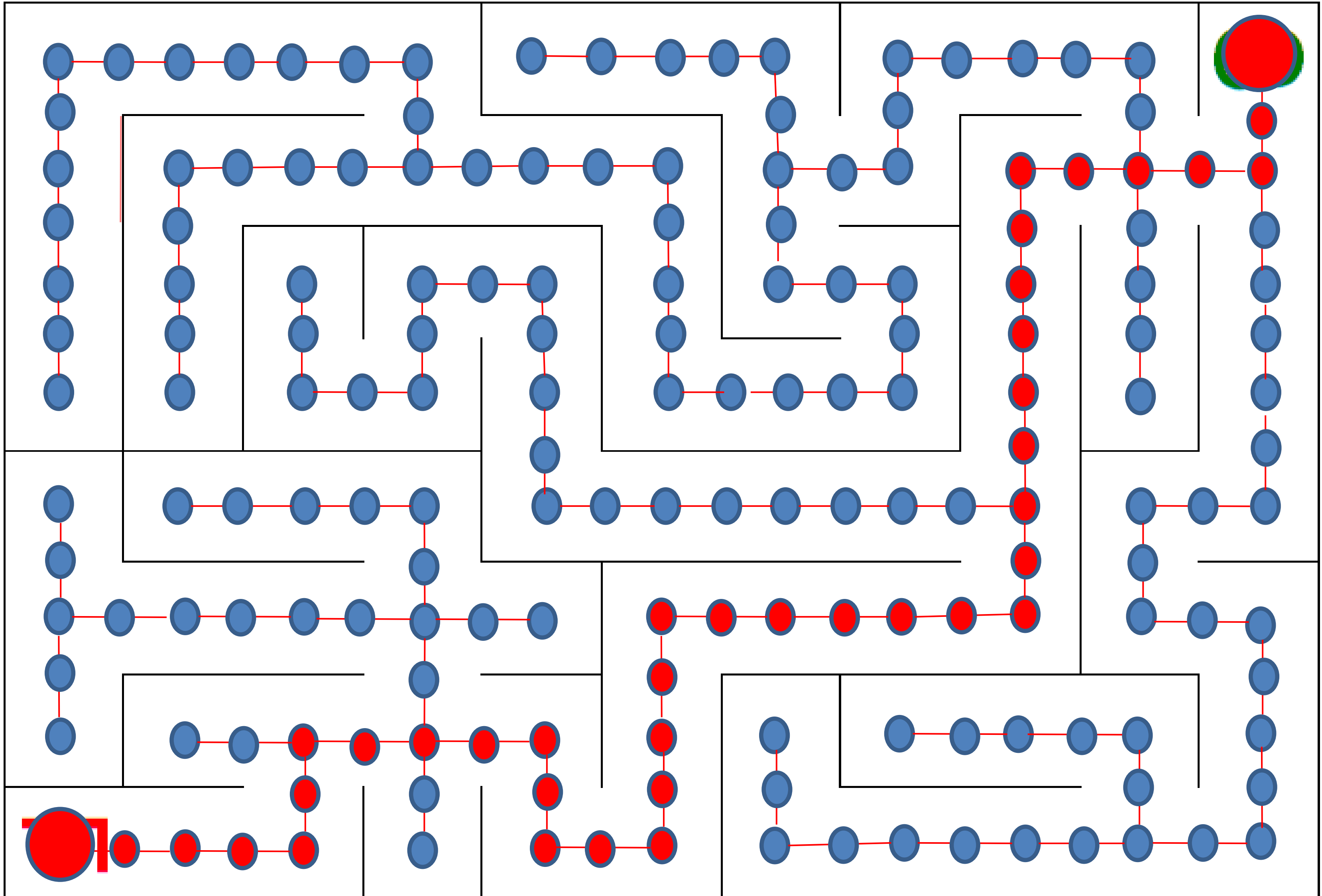
Labirinto



Labirinto come grafo esplicito



Percorso minimo



Labirinto come griglia (grafo implicito)

9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
9	1	1	1	1	1	1	1	9	1	1	1	1	1	9	1	1	1	1	1	9	1	9
9	1	9	9	9	9	9	1	9	9	9	9	9	1	9	1	9	9	9	1	9	1	9
9	1	9	1	1	1	1	1	1	1	1	1	9	1	1	1	9	1	1	1	1	1	9
9	1	9	1	9	9	9	9	9	9	9	1	9	1	9	9	9	1	9	1	9	1	9
9	1	9	1	9	1	9	1	1	1	9	1	9	1	1	1	9	1	9	1	9	1	9
9	1	9	1	9	1	9	1	1	1	9	1	9	9	9	1	9	1	9	1	9	1	9
9	1	9	1	9	1	9	1	9	1	9	1	1	1	1	1	9	1	9	1	9	1	9
9	9	9	9	9	9	9	9	9	1	9	9	9	9	9	9	9	1	9	9	9	1	9
9	1	9	1	1	1	1	1	9	1	1	1	1	1	1	1	1	1	9	1	1	1	9
9	1	9	9	9	9	9	1	9	9	9	9	9	9	9	9	9	1	9	1	9	9	9
9	1	1	1	1	1	1	1	1	1	9	1	1	1	1	1	1	1	9	1	1	1	9
9	1	9	9	9	9	9	1	9	9	9	1	9	9	9	9	9	9	9	9	9	1	9
9	1	9	1	1	1	1	1	1	1	9	1	9	1	9	1	1	1	1	1	9	1	9
9	9	9	9	9	1	9	1	9	1	9	1	9	1	9	9	9	9	9	1	9	1	9
9	1	1	1	1	1	9	1	9	1	9	1	9	1	1	1	1	1	1	1	1	1	9
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

Percorso minimo

9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
9 1 1 1 1 1 1 1 9 1 1 1 1 1 9 1 1 1 1 1 9 1 9
9 1 9 9 9 9 9 1 9 9 9 9 9 1 9 1 9 9 9 1 9 1 9
9 1 9 1 1 1 1 1 1 1 1 1 9 1 1 1 9 1 1 1 1 1 9
9 1 9 1 9 9 9 9 9 9 9 1 9 1 9 9 9 1 9 1 9 1 9
9 1 9 1 9 1 9 1 1 1 9 1 9 1 1 1 9 1 9 1 9 1 9
9 1 9 1 9 1 9 1 1 1 9 1 9 9 9 1 9 1 9 1 9 1 9
9 1 9 1 9 1 9 1 9 1 9 1 1 1 1 1 9 1 9 1 9 1 9
9 9 9 9 9 9 9 9 9 1 9 9 9 9 9 9 9 1 9 9 9 1 9
9 1 9 1 1 1 1 1 9 1 1 1 1 1 1 1 1 1 9 1 1 1 9
9 1 9 9 9 9 9 1 9 9 9 9 9 9 9 9 9 1 9 1 9 9 9
9 1 1 1 1 1 1 1 1 1 9 1 1 1 1 1 1 1 9 1 1 1 9
9 1 9 9 9 9 9 1 9 9 9 1 9 9 9 9 9 9 9 9 9 1 9
9 1 9 1 1 1 1 1 1 1 9 1 9 1 9 1 1 1 1 1 9 1 9
9 9 9 9 9 1 9 1 9 1 9 1 9 1 9 9 9 9 9 1 9 1 9
9 1 1 1 1 1 9 1 9 1 1 1 9 1 1 1 1 1 1 1 1 1 9
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

Ricerca del percorso ottimo

La nozione di ***percorso ottimo*** dipende dal problema.

Se ogni passo lungo un percorso ha lo stesso costo, collassa con la nozione di ***percorso minimo***.

Può essere risolto tramite un algoritmo di ricerca (visita) del grafo sottostante.

Ad esempio, con una semplice ***visita in ampiezza*** (**BFS**) si può trovare il percorso minimo.

Per grafi di dimensioni medio-grandi la visita in ampiezza risente di scarsa efficienza.

Ricerca euristica

Variante della ricerca in ampiezza in cui ad ogni nodo visitato è associato un ***valore che stima la bontà*** del percorso completo fino all'obiettivo.

La ricerca del percorso ottimo procede scegliendo di "*estendere*" il percorso parziale che termina nel nodo con stima migliore.

Intuitivamente, ad ogni passo si estende il percorso localmente più promettente.

Si sfrutta una ***funzione euristica*** che ***stima*** la distanza di un nodo dall'obiettivo.

Ricerca euristica

Ad ogni nodo visitato v si associa un valore composto da due funzioni:

$$f(v) = g(v) + h(v)$$

dove

- $g(v)$ è una valutazione del costo del miglior percorso finora trovato dal nodo di partenza s a v;
- $h(v)$ è una stima del costo per raggiungere la destinazione a partire da v.

Ricerca euristica

$g(v)$ potrebbe, ad esempio, misurare la lunghezza del percorso migliore trovato dalla sorgente s al nodo v;

$h(v)$ dipende fortemente dal dominio applicativo. Nel caso di un labirinto bidimensionale potrebbe misurare la distanza, misurata sulla griglia ignorando i muri, tra $v = (x_v, y_v)$ e il nodo destinazione $t = (x_t, y_t)$.

Esempi:

- Distanza Manhattan: $h(v) = (|x_t - x_v| + |y_t - y_v|)$
- Distanza euclidea: $h(v) = \sqrt{(x_t - x_v)^2 + (y_t - y_v)^2}$

Algoritmo A*

Popolare algoritmo di ricerca euristica su alberi e grafi che deriva dalla **BFS**.

Come la **BFS**, distingue tre insiemi di vertici:

- Vertici non raggiunti
- **Open-set**: vertici raggiunti ma non ancora espansi
- **Closed-set**: vertici già espansi

Open-set è l'analogo dei vertici grigi nella BFS

Closed-set è l'analogo dei vertici neri nella BFS

Algoritmo A*

L'algoritmo è inizializzato con **Closed-set** vuoto e **Open-set** contenente la sorgente.

A ogni iterazione sceglie per l'espansione il vertice **x** più promettente (quello col miglior valore **f(x)**) da **Open-set**.

Genera i suoi adiacenti (**y**) e per ciascuno di essi:

1. Calcola il valore di **f(y)**;
2. Se **y** è presente in **Open-set** aggiorna, se necessario, la stima **f(y)** in **Open-set**;
3. Se **y** non è in **Open-set** ma è in **Closed-set** e la stima **f(y)** è migliorata rispetto a prima, **y** viene tolto da **Closed-set** e inserito in **Open-set**, aggiornandone la stima;
4. Altrimenti, aggiorna la stima di **y** e lo inserisce in **Open-set**.

Algoritmo A*

Dalla descrizione si nota che **A*** non espande tutti i nodi grigi (in Open-Set) ad ogni passo, differentemente dalla **BFS**.

Ma può dover *riconsiderare più volte lo stesso vertice* (passo 3). Nel caso peggiore, tante volte quanti sono i percorso dalla sorgente al vertice.

Se il percorso ottimo consiste di **k** archi, non necessariamente tutti i vertici a quella distanza verranno visitati (come per **BFS**).

Molto spesso è molto più efficiente della **BFS**.

```
A-Star(G,source,target)
 f(source) = h(source,target)
 Closed = ∅
 Open = {source}
 WHILE Open ≠ ∅ OR found DO
   x = extract-min(Open)
   IF x = targer THEN found = true
   FOR each y ∈ Adj(x) DO
     cost = g(y) + h(y,target)
     IF y ∈ Open AND cost < f(y) THEN
          Aggiorna(Open,y,cost)
     ELSE IF y ∈ Closed AND cost < f(y) THEN
          Delete(Closed,y)
          f(y) = cost
          Insert(Open,y)
     ELSE f(y) = cost
          Insert(Open,y)
 IF found THEN path = Generate-path(source,target)
 ELSE path = NIL
 return path
```

Algoritmo A*

Si noti che **A*** non espande ad ogni passo tutti i nodi grigi (in Open-Set), differentemente dalla **BFS**.

Può però dover *riconsiderare più volte lo stesso vertice* (passo 3). Nel caso peggiore, tante volte quanti sono i percorsi dalla sorgente al vertice.

Se il percorso ottimo consiste di **k** archi, non necessariamente tutti i vertici a distanza **k** vengono visitati (come accade per **BFS**).

Se la stima $h(v) \leq h^*(v)$ (dove $h^*(v)$ denota la distanza reale dalla destinazione), allora **A*** è sempre garantito trovare il percorso ottimo.

È molto spesso molto più efficiente della **BFS**.

È consigliabile realizzare **Open-Set** come una *Coda a Priorità* al fine di poter eseguire efficientemente le operazioni di estrazione del minimo e di aggiornamento delle stime.

Code a priorità

- Una ***coda a priorità*** permette di rappresentare un insieme di elementi su cui è definita una relazione d'ordine.
- Poiché mantenere un ordine totale è computazionalmente troppo costoso, gli elementi vengono mantenuti in coda secondo un ***ordine parziale***.
- Garantisce efficienza per le seguenti operazioni:
  - **Insert(Q,k)**: inserimento di una chiave;
  - **Minimo(Q)** [risp. **Massimo(Q)**]: ritorna l'elemento minimo o massimo (a seconda dell'ordinamento scelto);
  - **Extract-Min(Q)** [risp. **Extract-Max(Q)**]: estrae dalla coda l'elemento minimo o massimo (a seconda dell'ordinamento scelto);
  - **Decrease-Key(Q,k,v)** [risp. **Increase-Key(Q,k,v)**]: aggiorna il valore dell'elemento **k**, se necessario, e ripristina l'ordine parziale;
  - **Delete(Q,k)**: rimuove l'elemento puntato da **k** dalla coda.

Heap binari e code a priorità

Una *Coda a Priorità* può essere implementata facilmente tramite uno *Heap Binario*.

Un *Heap Binario* è un *albero binario* tale che per ogni nodo *i*:

- **tutte le *foglie* hanno *profondità h* o *h-1*, dove *h* è l'*altezza dell'albero*;**
- **tutti i *nodi interni* hanno *grado 2*, eccetto al più uno;**
- **entrambi i *nodi j* e *k* figli di *i* sono *NON maggiori* (alternativamente *NON minori*) di *i*.**

Condizioni **1** e **2** definiscono la **forma dell'albero**

Heap binari e code a priorità

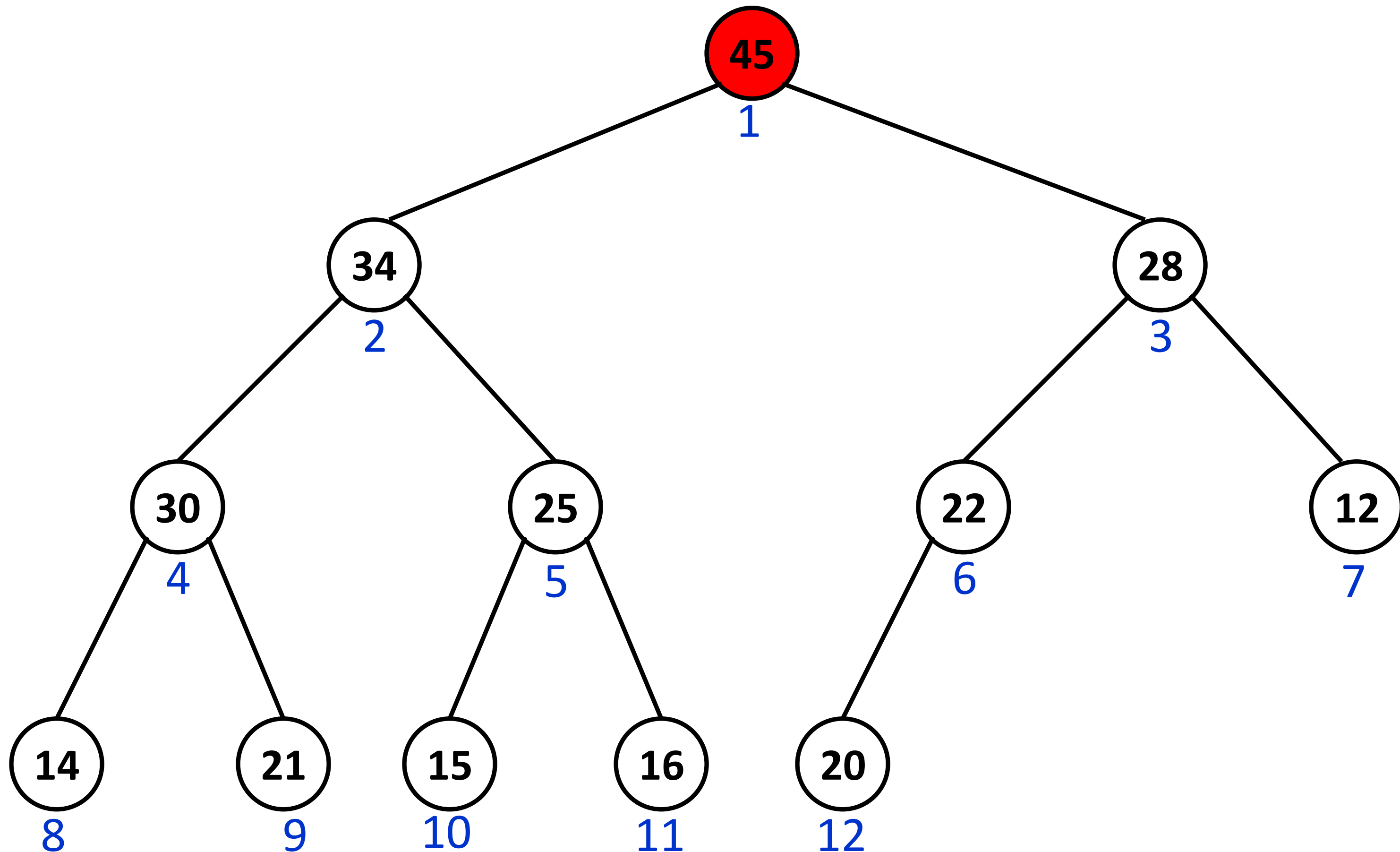
Una *Coda a Priorità* può essere implementata facilmente tramite uno *Heap Binario*.

Un *Albero Heap* è un *albero binario* tale che per ogni nodo *i*:

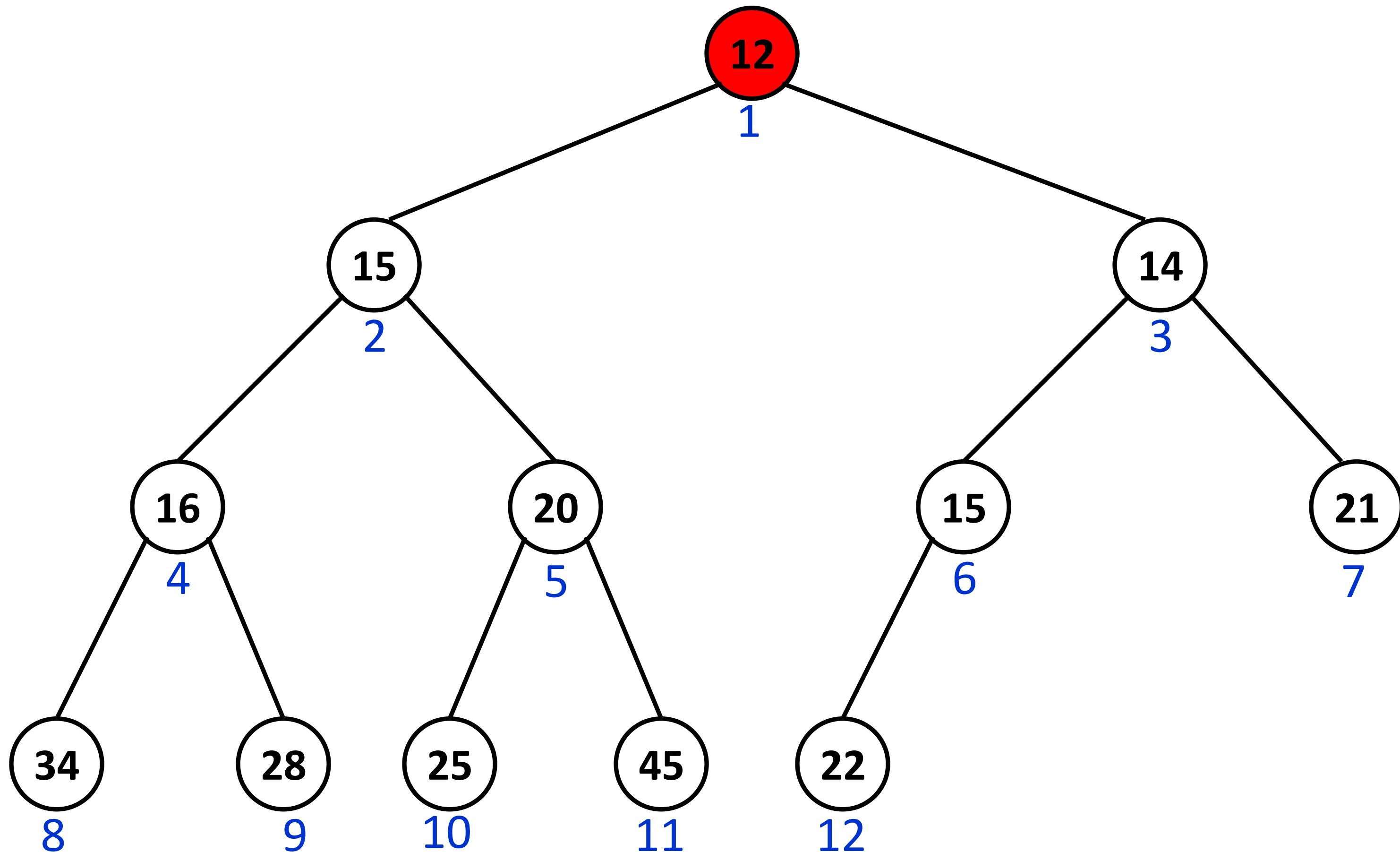
- **tutte le *foglie* hanno *profondità h* o *h-1*, dove *h* è l'*altezza dell'albero*;**
- **tutti i *nodi interni* hanno *grado 2*, eccetto al più uno;**
- **entrambi i *nodi j* e *k* figli di *i* sono *NON maggiori* (alternativamente *NON minori*) di *i*.**

Condizione **3** definisce l'**etichettatura dell'albero**

Esempio di Max Heap

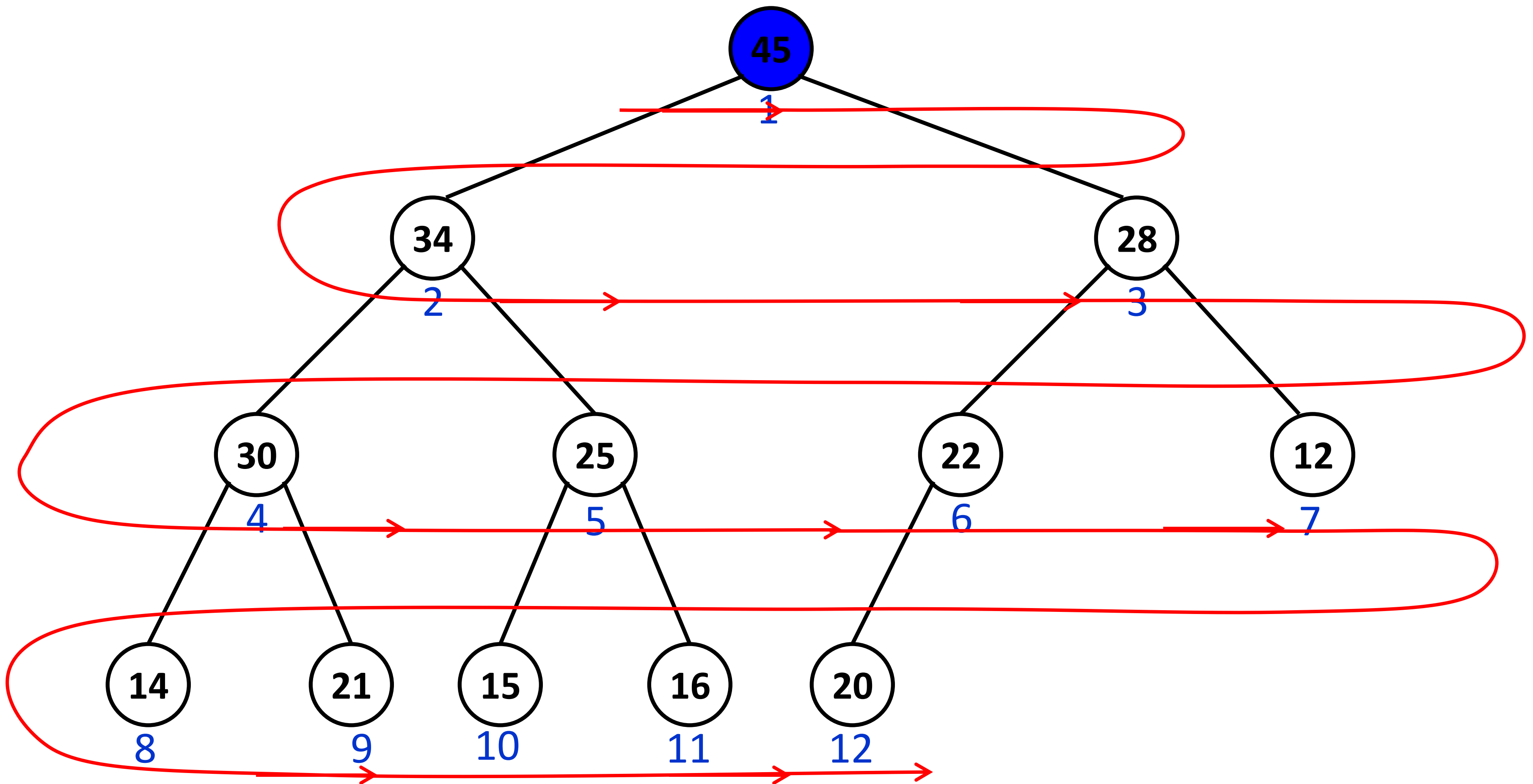


Esempio di Min Heap



Realizzazione di Heap: array

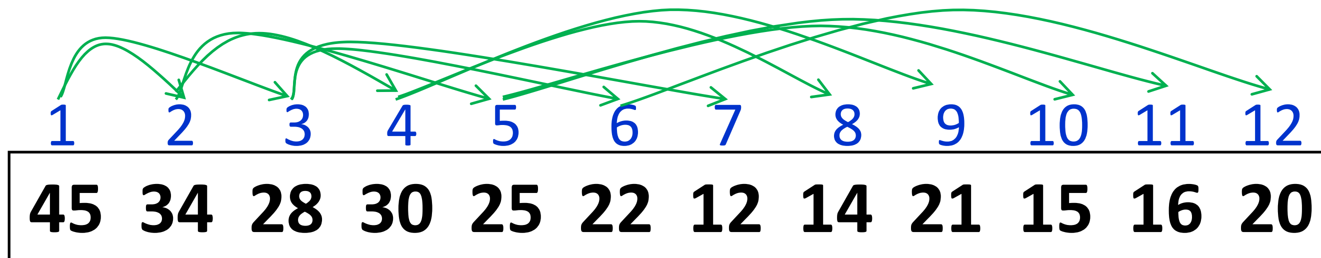
1	2	3	4	5	6	7	8	9	10	11	12
45	**34**	**28**	**30**	**25**	**22**	**12**	**14**	**21**	**15**	**16**	**20**



Realizzazione di Heap: array

Uno *Heap* può essere realizzato come un array *A* in cui:

- **la radice dello *Heap* sta nella prima posizione *A*[1] dell'array**
- **se il nodo *k* dello *Heap* sta nella posizione *i* dell'array (cioè *A*[*i*]):**
  - **il figlio sinistro di *k* sta nella posizione 2*i***
  - **il figlio destro di *k* sta nella posizione 2*i* +1**



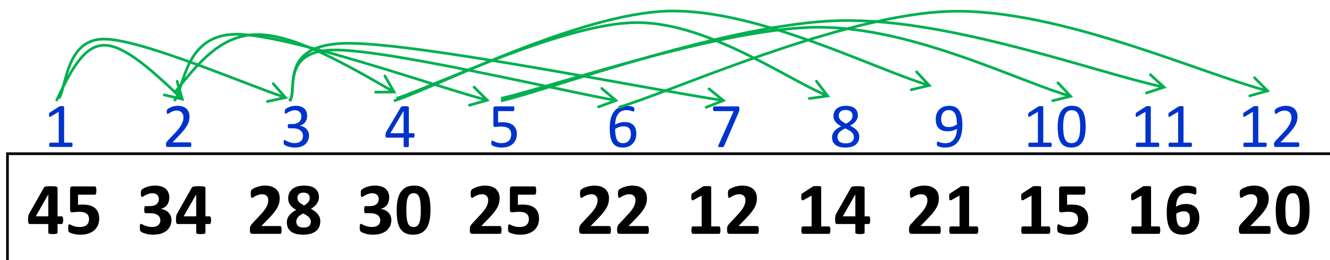
Realizzazione di Heap: array

```
SINISTRO(i)
   return 2i

DESTRO(i)
   return 2i + 1

PADRE(i)
   return ⌊i/2⌋
```

heapsize[A] $\leq n$ **è la lunghezza dello *Heap***

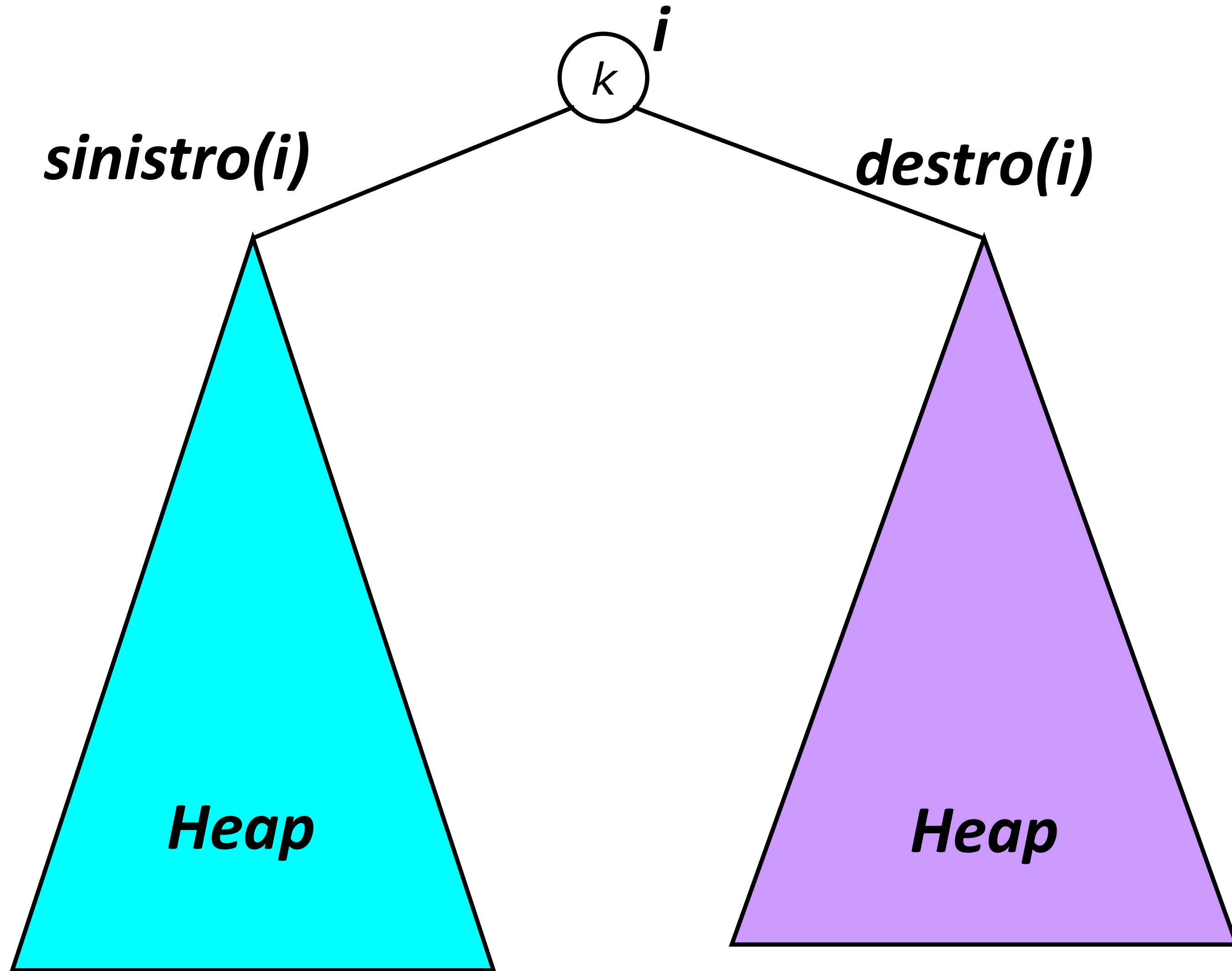


Heap: Funzioni di base

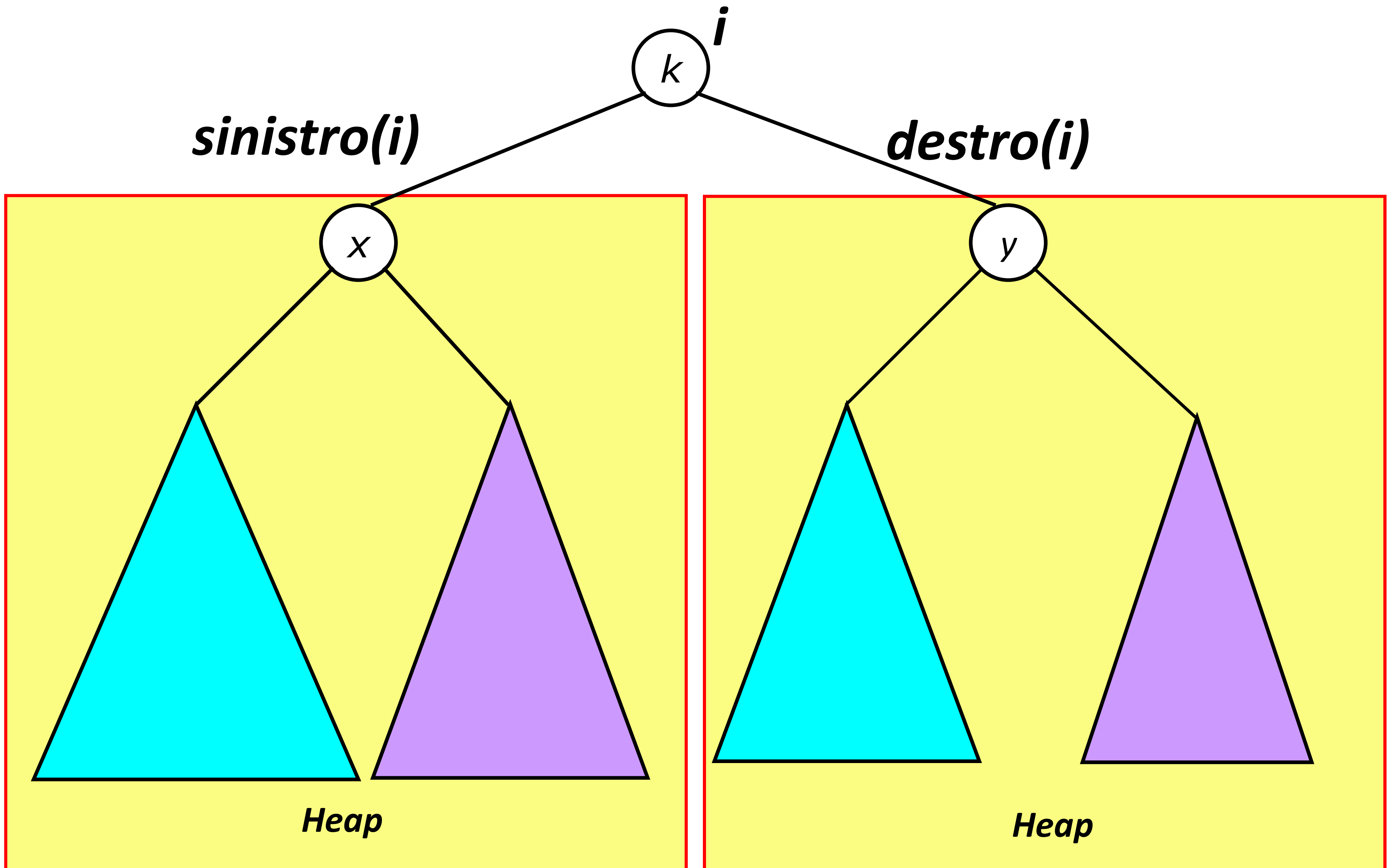
- `Heapify(`***`A,i`***`)`: ripristina la proprietà di **Heap** al sottoalbero radicato nella posizione ***i***, assumendo che i suoi sottoalberi destro e sinistro siano già degli **Heap**.

- `Build-Heap(`***`A`***`)`: produce uno **Heap** a partire dall'array ***A*** arbitrario.

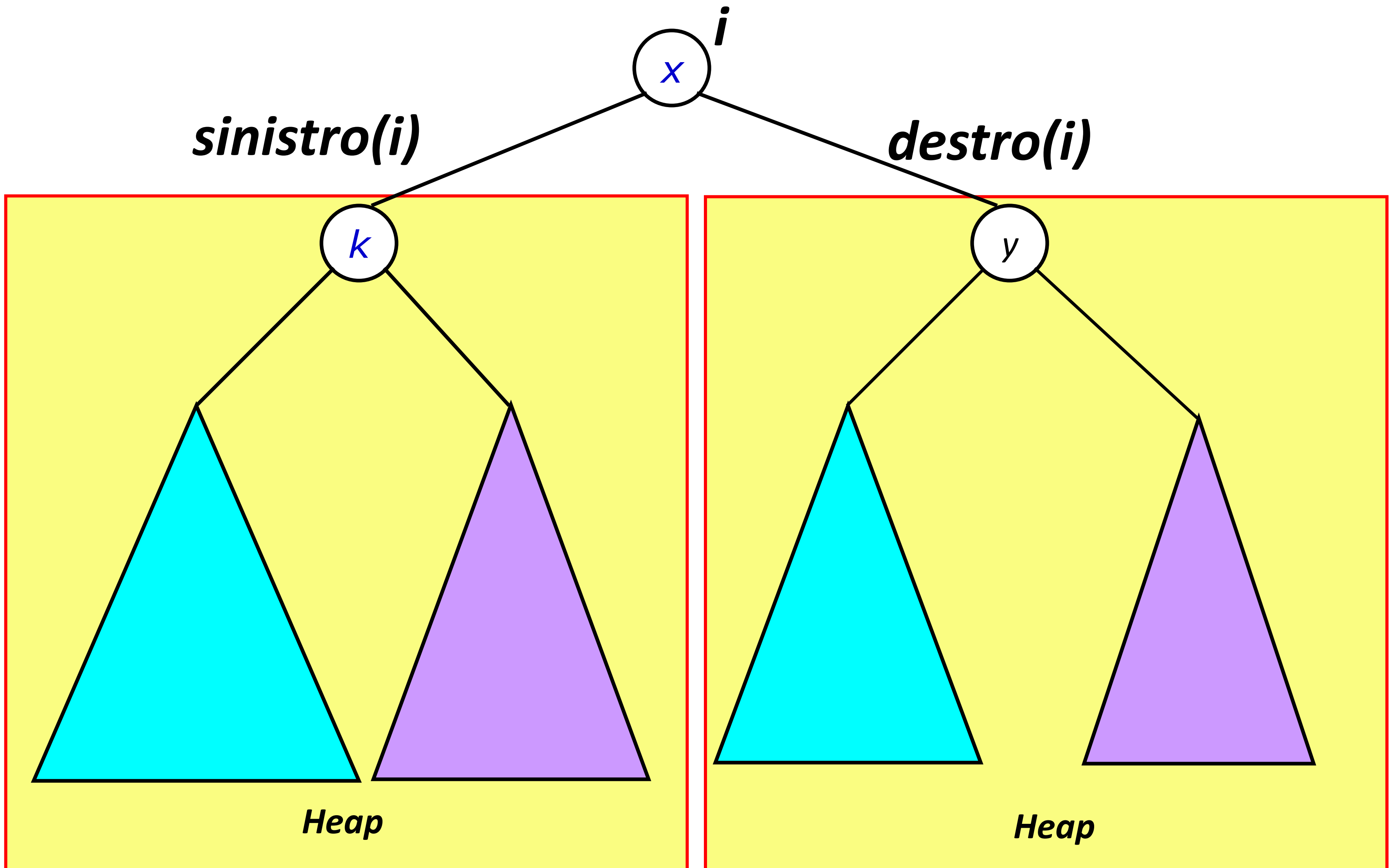
Heapify: Intuizioni



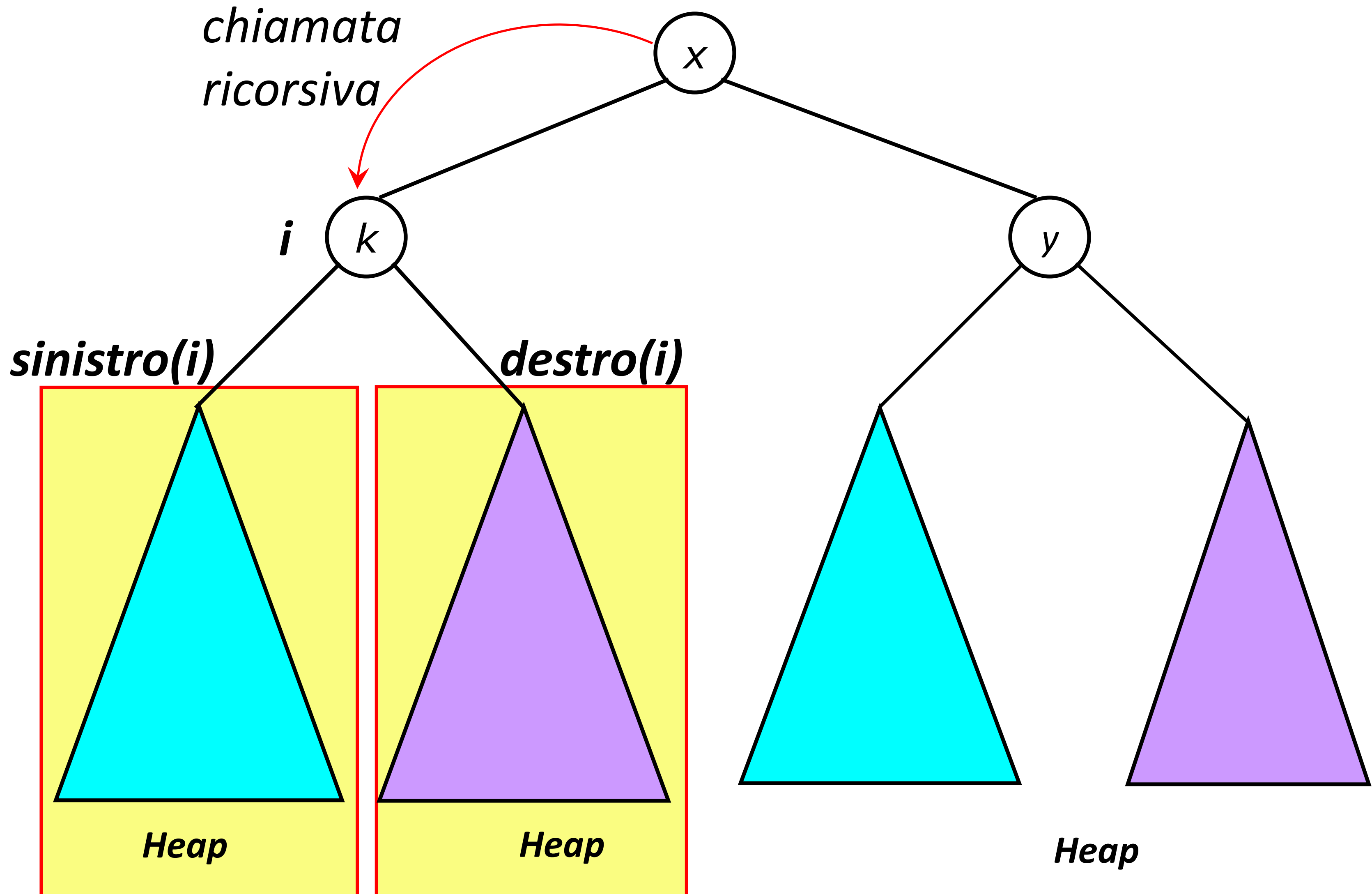
Heapify: Intuizioni



Heapify: x > k e y



Heapify: x > k e y



Algoritmo di Heapify

```
Heapify(A,i)
    l = SINISTRO (i)
    r = DESTRO(i)
    IF l ≤ heapsize[A] AND A[l] > A[i]
        THEN max = l
        ELSE max = i
    IF r ≤ heapsize[A] AND A[r] > A[max]
        THEN max = r
    IF maggiore ≠ i
        THEN "scambia A[i] e A[max]"
             Heapify(A,max)
```

Tempo di esecuzione: lineare sull'altezza dell'albero heap
logaritmico sul numero di elementi dello heap

Build-Heap: Intuizioni

`Build-Heap(A)`: utilizza l'algoritmo **`Heapify`**, per inserire ogni elemento dell'array in uno ***Heap***, risistemando sul posto gli elementi:

- gli ultimi $\lceil n/2 \rceil$ elementi dell'array sono foglie, cioè radici di sottoalberi vuoti, quindi sono già degli ***Heap***
- è sufficiente richiamare **`Heapify`** sui primi $\lfloor n/2 \rfloor$ elementi (in ordine decrescente di indice), per ripristinare la proprietà ***Heap*** sui sottoalberi in essi radicati.

L’algoritmo Build-Heap

```
Build-Heap(A)
    n = length[A]
    heapsize[A] = n
    FOR i =⌊n/2⌋ DOWNTO 1 DO
        Heapify(A,i)
```

Tempo di esecuzione: lineare sul numero **n** di elementi dello heap

Estrazione del massimo in una Coda a Priotità

```
Extract-Max(A)
   IF heapsize[A] < 1 then
            Error ``heap underflow''
            return NIL
   ELSE
       max = A[1]
       A[1] = A[heapsize[A]]
       heapsize[A] = heapsize[A] - 1
       Heapify(A,1)
return max
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Aggiornamento in Coda a Priorità

```
Decrease-Key(A,i,key)
   IF key > A[i] THEN
       Error ``nuova chiave più grande''
    A[i] = key
    Heapify(i)
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Aggiornamento in Coda a Priorità

```
Increase-Key(A,i,key)
   IF key < A[i] THEN
       Error ``nuova chiave più piccola''
    A[i] = key
    WHILE i > 1 AND A[Padre(i)] < A[i] DO
       Swap(A[i],A[Padre(i)])
       i = Padre(i)
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Inserimento in Coda a Priorità

```
Insert-Key(A,key)
    Heapseize[A] = Heapseize[A]+1
    A[heapzise] = -∞
    Increase-Key(A,Heapseize[A],key)
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Cancellazione in Coda a Priorità

```
Delete_Heap(A,i)
  IF heapsize[A] < 1 THEN
          Error ''heap underflow''
  ELSE
      A[i] = A[heapsize[A]]
      Heapify(A,i)
  A[heapsize[A]] = -∞
  heapsize[A] = heapsize[A] - 1
```

Tempo di esecuzione logaritmico sul numero **n** di elementi della coda

Code a Priorità come Alberi puntati

Una ***Coda a Priorità*** può, ovviamente, essere realizzata tramite un Albero Binario Puntato.

Può essere necessario prevedere in ogni nodo anche un ***puntatore al padre***, per poter risalire fino alla radice (ad esempio, se si prevede la funzione `Increase_key()`).

Vanno opportunamente modificate le funzioni `Delete_Heap()` e `Extract-Max()`.

Progetto Labirinto

Input da File:

1. una descrizione del labirinto come griglia di dimensioni **NxM**, utilizzando simboli differenti per i muri (“**-**” e “**|**”) e per i corridoi (spazi “ ”).
2. Una locazione della griglia **$s=(x_s,y_s)$** che rappresenta la posizione di partenza;
3. Una locazione della griglia **$t=(x_t,y_t)$** che rappresenta la posizione di arrivo.

Rappresentare il labirinto

- come grafo esplicito;
- come griglia (mantenendo il grafo implicito).

Progetto Labirinto

Implementare la ricerca del percorso ottimo (più breve) per raggiungere l'arrivo **t** dal punto di partenza **s**, utilizzando:

- Ricerca in ampiezza **BFS**;
- Ricerca euristica con l'algoritmo **A***.

Con entrambe le rappresentazioni del labirinto (grafo o grigli).

Potete, inoltre, prevedere anche una funzionalità di visualizzazione a terminale del labirinto (usando i caratteri) e del percorso risultante.