Alberi

1 Introduzione agli alberi binari

In teoria dei grafi, un *albero* nella sua accezione più generale corrisponde ad una particolare tipologia di *grafo* (non orientato, connesso e aciclico). Per un'esaustiva comprensione dei concetti più teorici si rimanda dunque al capitolo inerente i grafi. Per gli argomenti trattati in questo capitolo, è sufficiente comprendere come può essere implementata una struttura dati corrispondente ad un **albero binario**.

Un albero binario è composto da **nodi** connessi tra loro mediante certe regole. I nodi della struttura sono i vari oggetti, ognuno contenente un campo chiave e dati satelliti; inoltre i nodi contengono i campi *left*, *right* e *p* che puntano ad altri nodi denominati rispettivamente **figlio sinistro**, **figlio destro** e **padre** del nodo. Ogni nodo di un albero binario può dunque avere *al più due nodi figli e un unico nodo padre*. Un albero binario presenta un nodo denominato radice (*root*), a cui non connesso alcun nodo padre (nodo 1, nella *Figura 1*). Infine, nel caso degli alberi binari, non è solo importante il numero di figli di un nodo (0, 1 o 2), ma anche la *posizione* del figlio: figlio destro o figlio sinistro.

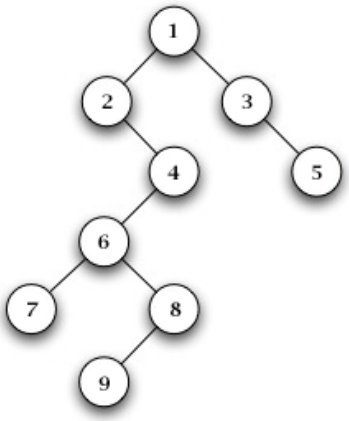

Figure 1: Esempio di albero binario etichettato con interi

Per ciò che riguarda il linguaggio di programmazione C, i nodi possono essere implementati mediante *struct*, tra i cui campi figurano i puntatori che provvedono alla connessione con il figlio sinistro e il figlio destro (il puntatore al padre solitamente non viene inserito).

Codice d'esempio:

```
struct nodo {
   int key;
   struct nodo *left;
   struct nodo *right;
}
```

Questo non è l'unico modo con cui può essere implementato un albero binario in C: anche un semplice array può essere utilizzato, come vedremo nella sezione del capitolo riguardante gli *heap*.

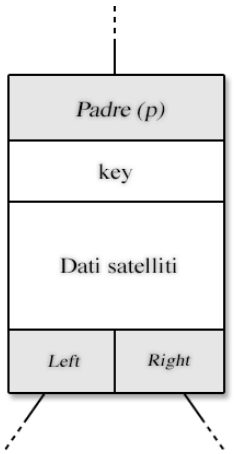

Figure 2: Possibile struttura di un nodo di albero binario, con le celle grigie puntatori ad altri nodi.

Una definizione ricorsiva di albero binario è la seguente: un albero binario è un insieme finito di nodi e può essere:

- un insieme vuoto, che non contiene nodi, oppure
- è composto da tre insiemi disgiunti di nodi: un nodo radice e due alberi binari, rispettivamente *sottoalbero sinistro* della radice e *sottoalbero destro* della radice.

Dato un albero binario T e un suo nodo x, un *sottoalbero con radice nel nodo* x è l'albero indotto dai discendenti del nodo x e avente come radice x. Un nodo qualsiasi y in

un cammino unico da r (radice) a x è detto *antenato* di x. Se y è un antenato di x, allora x è un *discendente* di y. Se $x \neq y$, allora si parla di *antenato proprio* e *discendente proprio*.

Vengono ora elencate ulteriori definizioni:

- *Foglia*: un nodo senza figli è definito come foglia;
- *Nodo interno*: un nodo avente almeno un figlio;
- *Profondità di un nodo*: se la profondità di un nodo è p i suoi figli non vuoti hanno profondità $p + 1$. La radice r ha profondità 0, i suoi figli sinistro e destro hanno profondità 1, i nipoti profondità 2 e così via;
- *Altezza di un albero*: massima profondità raggiunta dalle sue foglie o, in altri termini, profondità del cammino più lungo;
- *Albero binario completo*: albero binario in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado 2 (due figli).

In un **albero binario completo**, la radice ha 2 figli a profondità 1, ciascuno dei quali ha 2 figli alla profondià 2 e così via. Quindi, il numero di foglie alla profondità h è 2^h. Il numero totale di nodi risulta essere $2^{(h+1)} - 1$. L'altezza di un albero binario completo con p foglie è $\log_2(p)$.

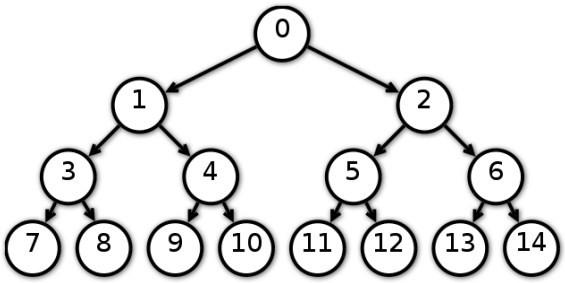

Figure 3: Esempio di un albero binario completo

2 Albero binario di ricerca

Un **albero binario di ricerca** è un albero binario in cui i nodi sono disposti ordinatamente a seconda del valore chiave che li identifica, in modo che valori minori di quelli del nodo di partenza siano memorizzati nei figli a sinistra e i valori più grandi nei figli a destra.

Sia x un nodo dell'albero binario di ricerca. Se y è un nodo nel sottoalbero sinistro di x allora il valore *key* di y deve essere minore del valore *key* di x. Viceversa, se y è un nodo

nel sottoalbero destro di x allora il valore *key* di y deve essere maggiore del valore *key* di x.

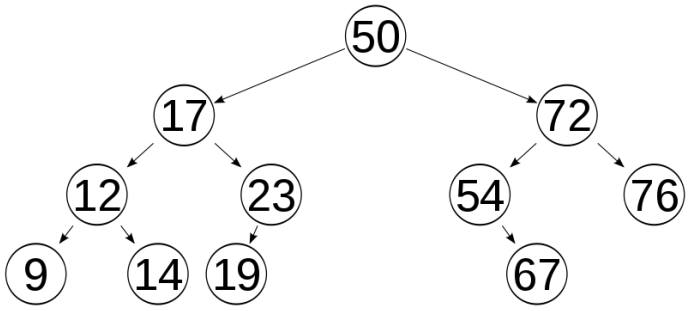

Figure 4: Esempio di un albero binario di ricerca

Gli alberi di ricerca binari rappresentano una struttura di dati che supporta in modo efficiente le operazioni SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE. La proprietà di ordinamento delle chiavi che contraddistingue un albero binario di ricerca permette di eseguire le operazioni più comuni, come ad esempio la ricerca di un elemento, con un tempo $O(h)$, dove h è l'altezza dell'albero. Per un albero completo con n nodi la complessità risulta essere $\Theta(\lg\ n)$ nel caso peggiore. Tuttavia, la struttura dell'albero può degenerare in una catena lineare di n elementi, con conseguente deterioramento delle prestazioni a $\Theta(n)$ nel caso peggiore.

2.1 Attraversamento

È possibile elencare ordinatamente tutte le chiavi di un albero binario di ricerca con un semplice algoritmo ricorsivo chiamato *attraversamento simmetrico di un albero* (inorder). Tale algoritmo segue la regola per la quale, dato un nodo di un albero binario, prima si visita il figlio sinistro e se tale nodo non esiste si visita il nodo di partenza (stampandone il valore *key*) prima di proseguire con la visita del figlio destro (se esiste). Tale regola viene applicata a ogni nodo raggiunto, in modo ricorsivo.

INORDER-VISIT
1
2
3
4

Con una minima modifica all'algoritmo sopra riportato, è possibile definire anche un'*attraversamento anticipato di un albero* (preorder), con cui la radice viene elencata

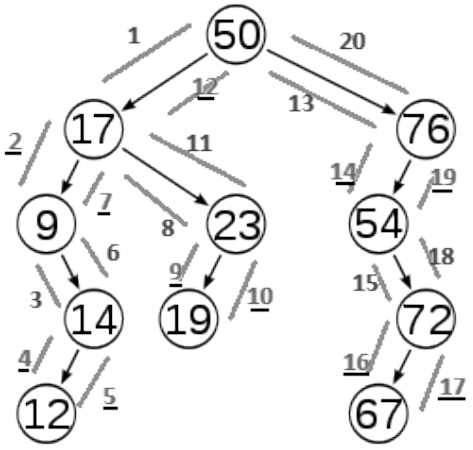

Figure 5: Attraversamento simmetrico di un albero binario (gli indici sottolineati indicano che la chiave toccata viene aggiunta all'elenco)

prima dei valori dei suoi sottoalberi, e un'*attraversamento posticipato di un albero* (postorder), il quale elenca la radice dopo i valori dei suoi sottoalberi.

Considerando l'albero binario di ricerca della *Figura 5* , gli elenchi risultanti dai tre metodi di attraversamento sono i seguenti:

Ordine	**Sequenza**
In-Order	9, 12, 14, 17, 19, 23, 50, 54, 67, 72, 76
Pre-Order	50, 17, 9, 14, 12, 23, 19, 76, 54, 72, 67
Post-Order	12, 14, 9,19, 23, 17, 67, 72, 54, 76, 50

2.2 Ricerca

L'algoritmo di ricerca per un albero binario di ricerca, dato un puntatore alla radice dell'albero e una chiave k, restituisce un puntatore a un nodo con chiave k se esiste, altrimenti restituisce il valore NIL.

SEARCH
1
2
3
4
5

La ricerca di un elemento in un albero binario di ricerca è basata su di un efficiente algoritmo a carattere ricorsivo. La procedura inizia la sua ricerca dalla radice e prosegue verso il basso lungo l'albero, quindi il tempo di esecuzione è $O(h)$, dove h è l'altezza dell'albero.

Per ogni nodo x che viene visitato, si confronta la chiave k con $key[x]$. Se le due chiavi sono uguali la ricerca termina, altrimenti la ricerca prosegue lungo il sottoalbero sinistro se key è minore di $key[x]$ o lungo il sottoalbero destro se key è maggiore di $key[x]$. Tale processo restituisce il risultato corretto grazie alla proprietà fondamentale degli alberi binari di ricerca, per la quale se k è minore di $key[x]$, k non può essere memorizzata nel sottoalbero destro del nodo x (e viceversa).

2.3 Inserimento

La procedura di inserimento di un elemento è relativamente semplice e comparabile a quella di ricerca sotto molti aspetti. Ovviamente, è sempre necessario inserire un elemento mantenendo valide tutte le proprietà strutturali dell'albero binario di ricerca.

INSERT
1
2
3
4
5
6
7
8
9
10
11
12
13

La ricerca della posizione in cui inserire l'elemento parte dalla radice dell'albero e si sposta verso il basso, nei sottoalberi sinistro o destro a seconda dell'esito del confronto tra la chiave dell'elemento visitato e quella dell'elemento da inserire. Un puntatore aggiuntivo mantiene sempre disponibile l'accesso al padre del nodo visitato dal puntatore primario: quando a quest'ultimo viene assegnato il valore NIL è stata raggiunta la posizione in cui inserire l'elemento; l'inserimento vero e proprio fa uso del puntatore secondario per l'accesso alla regione di memoria richiesta.

Analogamente alle altre operazioni elementari con gli alberi di ricerca, la procedura di inserimento viene eseguita in tempo $O(h)$, con h l'altezza dell'albero.

2.4 Cancellazione

La cancellazione di un elemento da un albero binario di ricerca è un'operazione più complicata, che varia a seconda che il nodo da cancellare non abbia figli o ne abbia uno o due.

Se il nodo da cancellare z non ha figli (z è una foglia dell'albero), è sufficiente operare sul padre $p[z]$ per sostituire z con il valore NIL.

Se il nodo z ha un figlio, è necessario creare un nuovo collegamento tra il padre e il figlio di z prima di effettuare la cancellazione.

Il caso con due figli collegati a z è il più complicato. E' possibile operare sia sul sottoalbero sinistro del nodo da cancellare che su quello destro. Il primo passo consiste nel trovare l'elemento con chiave maggiore a tutte le altre nel sottoalbero sinistro (l'elemento più a destra del sottoalbero) oppure l'elemento con chiave minore nel sottoalbero destro (l'elemento più a sinistra del sottoalbero). Una volta individuato l'elemento descritto, che indichiamo con x, è necessario creare un collegamento tra il padre e l'eventuale figlio dell'elemento stesso (x per definizione avrà al più un figlio); successivamente si sovrascrivono la chiave e i dati satelliti dell'elemento x nell'elemento z, per poi di procedere alla cancellazione di x.

Nonostante la maggiore complessità di codice, la procedura viene eseguita sempre nel tempo $O(h)$, in un albero di altezza h.

3 Heap

Un *heap* (binario) è un albero binario con una *proprietà strutturale* e una *proprieà funzionale* specifiche:

- *Proprietà strutturale*: è un albero binario *quasi completo* (tutti i livelli di profondià sono completi ad eccezione dell'ultimo, che può essere non completo ma che viene riempito sempre da sinistra a destra).
- *Propreità funzionale*: per ogni nodo che non è la radice dell'albero, la chiave di tale nodo è sempre *minore o uguale* alla chiave del padre.

Da queste osservazioni sulla struttura di un heap si evince che la chiave con valore massimo tra quelle memorizzate risiede nella radice dell'albero (l'heap è del tipo *max-heap*). È tuttavia possibile imporre che le chiavi dei vari nodi siano sempre *maggiori o uguali* delle chiavi dei rispettivi padri, cosicché nella radice dell'albero si trovi la chiave minore (in questo caso si parla di *min-heap*). Gli heap sono alla base dell'algoritmo di ordinamento denominato *heapsort*; inoltre la struttura heap realizza un'efficiente coda di priorità.

La struttura dati di un heap prevede dunque un albero binario. Per ciò che riguarda l'implementazione, nel caso degli heap si adotta un semplice **array** A avente due attributi: *lenght*$[A]$, che indica la dimensione dell'array, e *heap-size*$[A]$, che indica il numero degli elementi dell'heap memorizzati nell'array (il numero deve essere minore o uguale alla dimensione dell'array).

Per registrare nell'array le informazioni di parentela tra i vari nodi si adottano le seguenti regole: se i è l'indice di un nodo, la posizione nell'array in cui si trova il **figlio sinistro** è $2i + 1$, quella del **figlio destro** è $2i + 2$. La radice dell'albero è registrata in $A[0]$.

La *Figura 6* riassume i concetti descritti fino a questo punto, mostrando un esempio di heap e la relativa implementazione con l'array (nel caso d'esempio la dimensione dell'array coincide con la dimensione dell'heap).

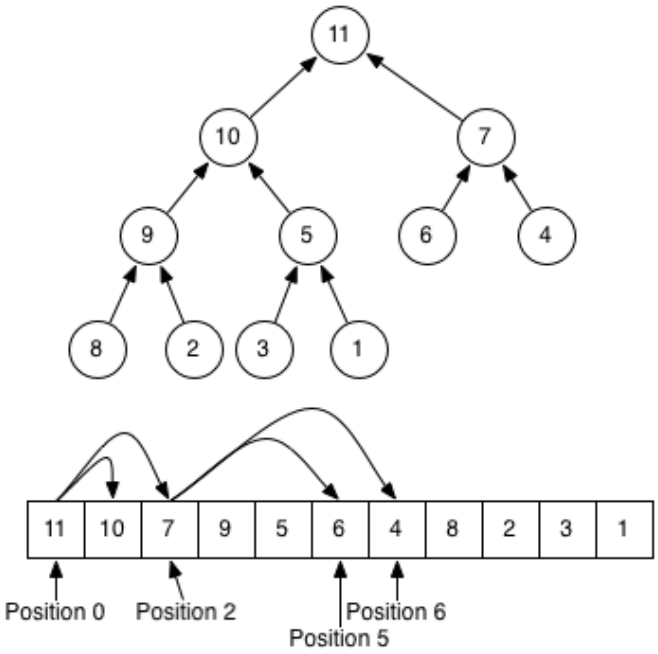

Figure 6: Esempio di un heap con relativa implementazione nell'array

3.1 Operazioni sugli heap

Le operazioni fondamentali eseguite sugli heap richiedono un tempo $O(\lg n)$, con n uguale al numero degli elementi. Tra queste andremo ad analizzare la procedura di inserimento.

Successivamente verranno introdotte ulteriori procedure specifiche degli heap spiegando come vengono utilizzate in un algoritmo di ordinamento e in una coda di priorità.

Per la successiva parte del capitolo si assume che l'heap sia del tipo **max-heap** (nella radice dell'albero è memorizzata la chiave con il valore massimo).

3.1.1 Inserimento

Il primo passo necessario per inserire un elemento in un heap consiste nell'aggiungere una foglia all'albero binario seguendo la *proprietà strutturale* (riempimento dell'ultimo livello da sinistra a destra). Anche la *proprietà funzionale* deve essere mantenuta, dunque è necessario seguire il cammino dalla foglia fino, al più, alla radice dell'albero, comparando il valore chiave dei nodi attraversati con quello dell'elemento che si desidera inserire: se necessario si deve far "discendere" i nodi già memorizzati lungo il ramo di appartenenza fino a quando non viene trovata lo posizione corretta dove inserire il nuovo elemento, tale per cui la chiave del padre risulti maggiore della propria (oppure la posizione corrisponde alla radice dell'albero).

La procedura fondamentale di INSERT, come già anticipato, risulta avere complessità temporale $O(\lg n)$.

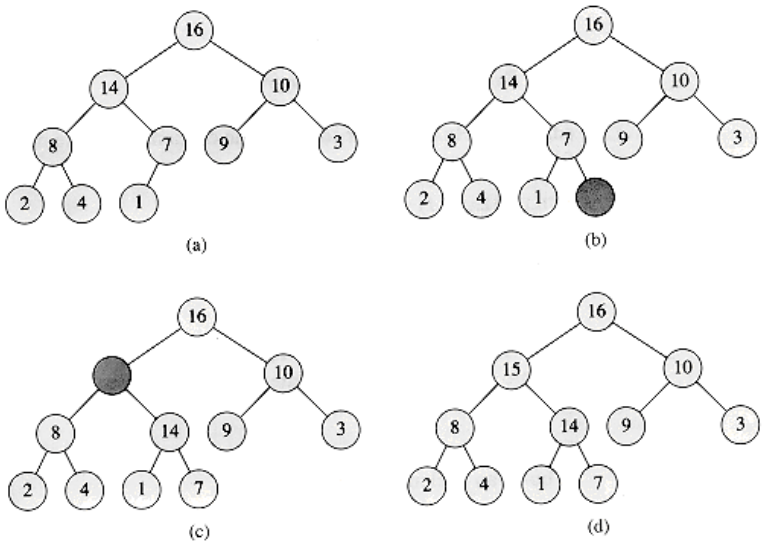

Figure 7: Inserimento dell'elemento 15 nell'heap d'esempio

3.1.2 Heapify

HEAPIFY è un'operazione che riceve come input l'array A e un indice i dell'array. Nel caso in cui $A[i]$ sia più piccolo dei suoi figli LEFT(i) o RIGHT(i), si viola la proprietà funzionale degli heap. Supponendo che i sottoalberi con radici LEFT(i) e RIGHT(i) siano effettivamente heap, la procedura di HEAPIFY consente al valore $A[i]$ di "scendere" in modo opportuno lungo l'albero: al termine dell'operazione il sottoalbero con radice in i diventa un heap.

Assumendo che LEFT(i) e RIGHT(i) siano heap, il primo passo consiste nello scambiare il valore $A[i]$ con il massimo valore tra $A[i]$, LEFT(i) e RIGHT(i). Se lo scambio è avvenuto con il figlio sinistro si applica ricorsivamente HEAPIFY nel sottoalbero con radice in LEFT(i), se lo scambio è avvenuto con il figlio destro HEAPIFY viene eseguito sul sottoalbero destro. È necessario applicare ricorsivamente HEAPIFY sui sottoalberi in quanto a seguito di uno scambio di valori non è detto che il sottoalbero interessato mantenga le proprietà di un heap (bisogna verificare con il nuovo valore "ereditato" dal padre). La procedura termina non appena $A[i]$ risulti essere il più grande tra $A[i]$, LEFT(i) e RIGHT(i) (o se si raggiunge una foglia).

HEAPIFY è un'operazione con complessità logaritmica: $T(\mathrm{n})=O(\lg n)$.

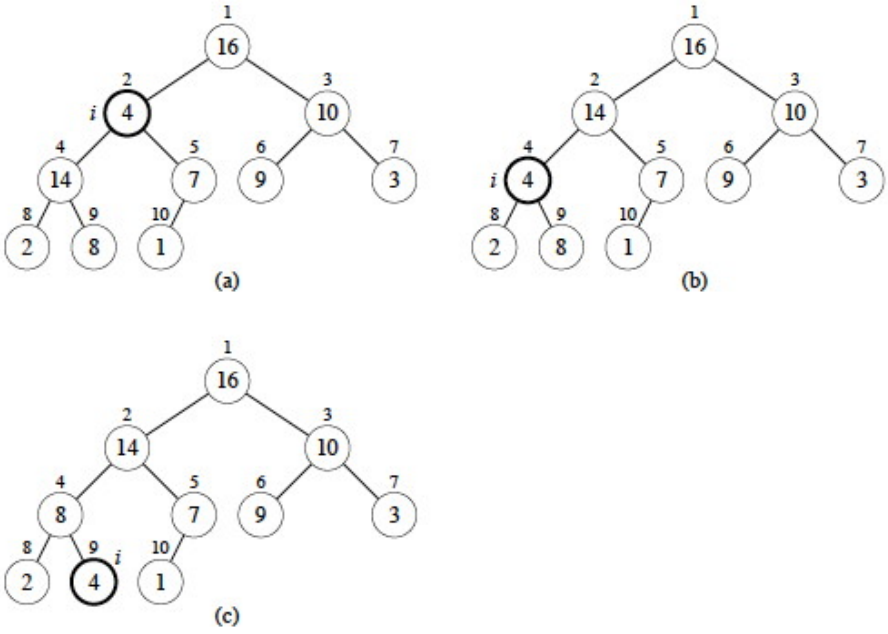

Figure 8: Heapify applicato al nodo 2 dell'heap d'esempio

3.1.3 Build-Heap

BUILDHEAP è la procedura che permette di riposizionare gli elementi memorizzati in un array generico A per convertire A in un heap. Tale procedura consiste nell'applicare ripetutamente HEAPIFY, partendo dal nodo padre della foglia più a destra per poi proseguire con i rimanenti nodi interni, seguendo a ritroso gli indici dell'array, fino alla radice (indice 0). Come risultato l'array viene convertito in un heap.

BUILDHEAP è un'operazione con complessità lineare: $T(\mathrm{n})=O(n)$.

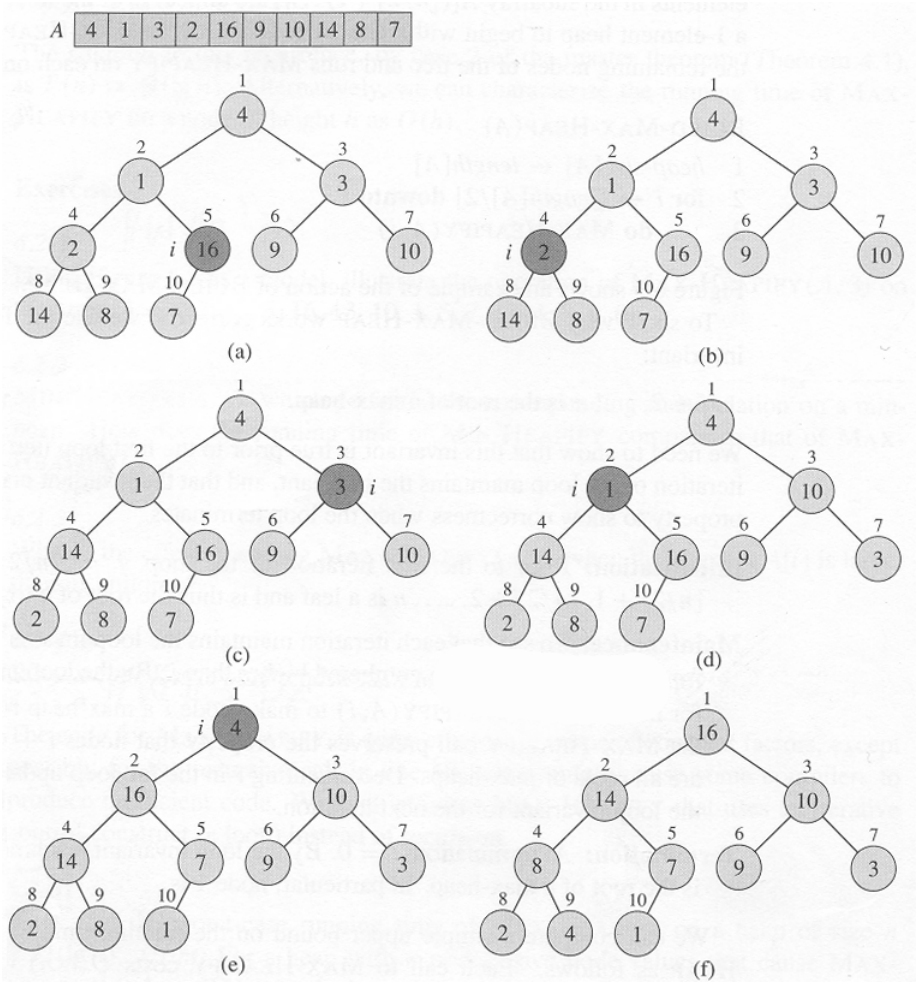

Figure 9: BUILDHEAP applicato ad un array generico A. Per ogni figura viene applicato HEAPIFY al nodo più scuro

3.2 Heapsort

L'algoritmo heapsort è un algoritmo di ordinamento efficiente, il cui tempo di esecuzione è limitato a $O(n \lg n)$. Heapsort si basa sulle operazioni eseguibili sugli heap precedentemente descritte.

L'algoritmo ha come input l'array $A[0,...,(n-1)]$ da ordinare. Il primo passo consiste nel convertire tale array in un heap utilizzando BUILDHEAP. Al termine del processo di BUILDHEAP, l'elemento dell'array con valore più alto è memorizzato nella radice dell'albero, cioè nella posizione 0 dell'array: si può dunque spostare tale elemento nella sua posizione finale, scambiandolo con l'ultimo elemento $A[n-1]$. Se ora si riduce di una unità il valore di *heap-size*$[A]$, scartando dall'heap il nodo già posizionato in modo definitivo, si può notare che i figli della radice restano heap, e solo la nuova radice (ottenuta come risultato del precedente scambio) può violare le proprietà degli heap: per ottenere nuovamente un heap corretto è sufficiente applicare HEAPIFY alla radice. Successivamente possiamo essere certi che il nuovo valore massimo dell'heap (il secondo maggiore nell'array) sia memorizzato nella radice, e si procede con un nuovo scambio tra $A[0]$ e $A[n-2]$. Ora si può ridurre nuovamente il valore di *heap-size*$[A]$ e continuare con la medesima procedura, fino a un heap di dimensione 2. Al termine, l'array sarà ordinato.

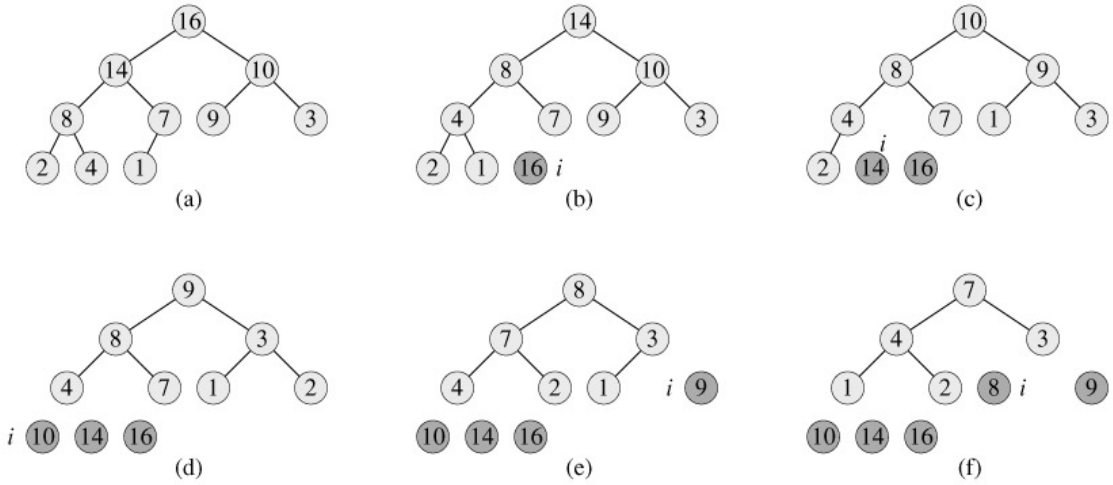

Figure 10: (a) La struttura iniziale subito dopo BUILDHEAP. (b)-(f) I primi 5 passi dell'algoritmo di ordinamento (la struttura è quella subito successiva al processo di HEAPIFY eseguito alla radice). Solo i nodi chiari restano nell'heap, quelli scuri sono ordinati

3.3 Code di priorità

Una coda di priorità è una struttura dati in cui ogni elemento memorizzato è associato a una chiave, che ne indica il livello di priorità; una coda di *max-priorità* supporta le seguenti operazioni:

- INSERT: inserisce un elemento nella struttura;
- MAXIMUM: restituisce l'elemento con la chiave più grande;
- EXTRACT-MAX: restituisce l'elemento con la chiave più grande eliminandolo dalla coda di priorità;
- INCREASE-KEY: aumenta il valore chiave di un dato elemento;

Tale struttura di dati è fondamentale in numerose applicazioni, quali la programmazione dei lavori con diversi livelli priorità o la simulazione controllata di eventi.

Un heap può implementare un'efficiente coda di priorità. L'operazione MAXIMUM, ad esempio, consiste semplicemente nell'accedere all'elemento radice dell'heap, con costo unitario: $\Theta(1)$. Tutte le altre operazioni sopra elencate, inerenti le code di priorità, hanno un costo pari a $O(\lg n)$ nel caso di un'implementazione con heap di n elementi.