Strutture Dati

Lezione 20
Ordinamento mediante heap e con radice

Oggi parleremo di …

- Ordinamento mediante heap (heapsort)
  - algoritmo
  - implementazione
  - analisi della complessità
- Ordinamento con radice (radixsort)
  - algoritmo
  - implementazione
  - analisi della complessità

Ordinamento mediante heap

- L'algoritmo heapsort
  - è una variazione dell'algoritmo di Selezione in cui la ricerca dell'elemento massimo è facilitata dall'utilizzo di una opportuna struttura dati
  - la struttura dati è l'*heap*
  - in uno heap l'elemento massimo può essere acceduto in tempo costante.

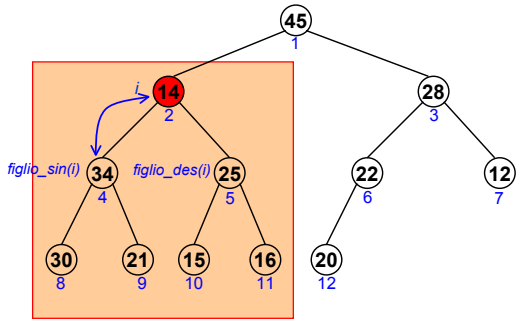
Ordinamento mediante heap

- Gli n elementi da ordinare vengono inseriti in un heap inizialmente vuoto.
- Gli elementi vengono estratti dall'heap, uno alla volta ed inseriti in fondo alla sequenza
  - ad ogni passo si estrae il massimo, lista[1]
  - lo si alloca in fondo alla sequenza
  - si adatta l'albero binario per ottenere l'heap, ridotto di 1 elemento
- Si genera la permutazione lista[1]≤lista ≤[2] ≤ … ≤ lista[n].

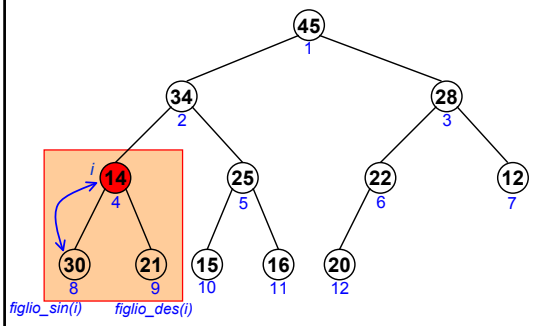
Heapsort: costruzione dell'heap

- Bisogna risolvere innanzitutto il problema seguente:
- Dati due heap H_1 e H_2 con radici figlio_sinistro(i) e figlio_destro(i) e un nuovo elemento v in posizione i adattare l'array A per ottenere l'heap massimo.
- Se lo heap H con radice $A[i]=v$ viola la proprietà dell'heap allora:
  - porre in $A[i]$ la *più grande* tra le radici degli *heap* H_1 e H_2
  - ripetere l'adattamento del sottoalbero selezionato (con radice $A[2i]$ o $A[2i+1]$) e dell'elemento v (ora in posizione $A[2i]$ o $A[2i+1]$).

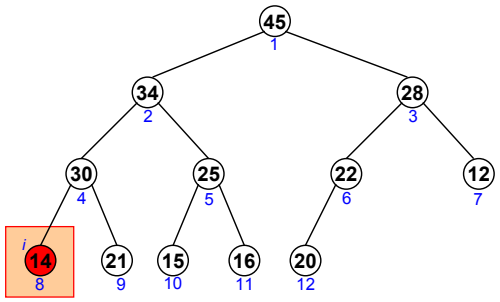
Heapsort: adattamento dell'heap



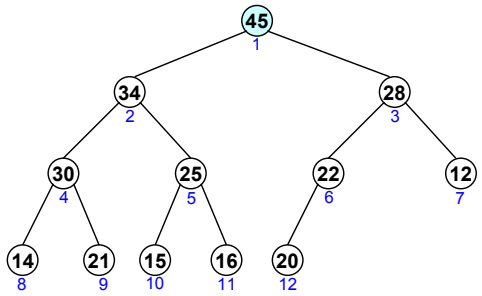
Heapsort: adattamento dell'heap



Heapsort: adattamento dell'heap



Heapsort: adattamento dell'heap



Heapsort: adattamento dell'heap

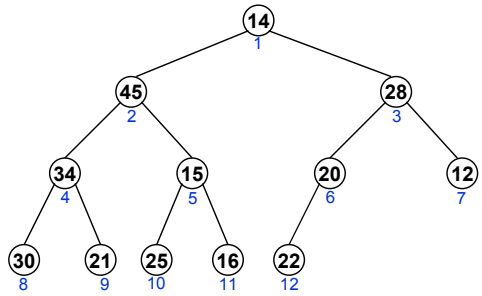
```
void adatta(elemento lista[ ], int radice, int n)
// adatta l'albero binario per ottenere l'heap
{
 int figlio, chiaveradice;
 elemento temp;

 temp = lista[radice];
 chiaveradice = lista[radice].chiave;
 figlio = 2*radice;       // figlio a sinistra
 while(figlio <= n) {
        if((figlio < n) && (lista[figlio].chiave<lista[figlio+1].chiave))
                figlio ++;
        if(chiaveradice > lista[figlio].chiave)
                        // confronta la radice e figlio max
                break;
        else {
                lista[figlio/2] = lista[figlio];
                        // si sposta verso il padre
                figlio *= 2;
             }
        }
 lista[figlio/2] = temp;
}
```

Heapsort: costruzione dell'heap

- Per costruire uno heap da un array di *n* elementi si utilizza l'algoritmo `adatta`, inserendo ogni elemento dell'array e risistemando gli elementi fuori posto:
  - gli ultimi $\lceil n/2 \rceil$ elementi dell'array sono foglie, cioè radici di sottoalberi vuoti, quindi sono già degli *heap*
  - è sufficiente inserire nello *heap* solo i primi $\lfloor n/2 \rfloor$ elementi, utilizzando `adatta` per ripristinare la proprietà dell'*heap* sul sottoalbero del nuovo elemento.

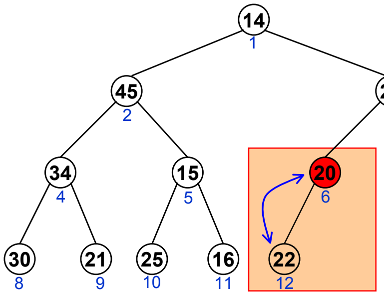
Heapsort: costruzione dell'heap

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



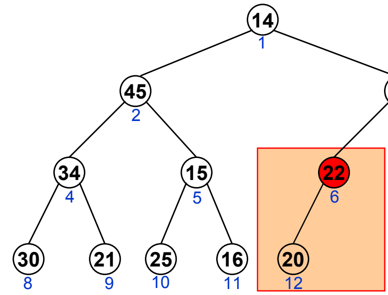
Heapsort: costruzione dell’heap

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



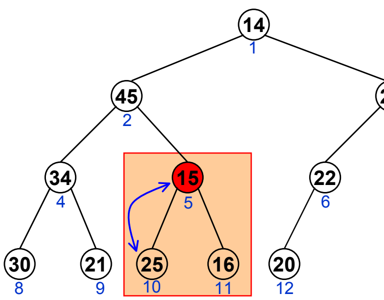
Heapsort: costruzione dell’heap

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



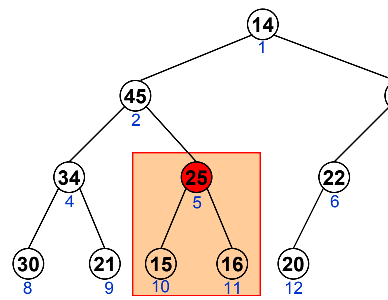
Heapsort: costruzione dell’heap

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



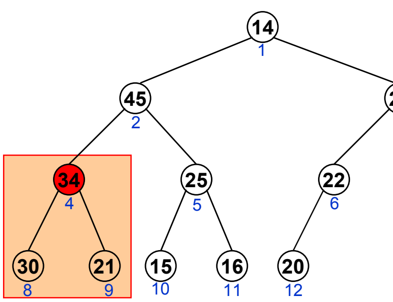
Heapsort: costruzione dell’heap

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



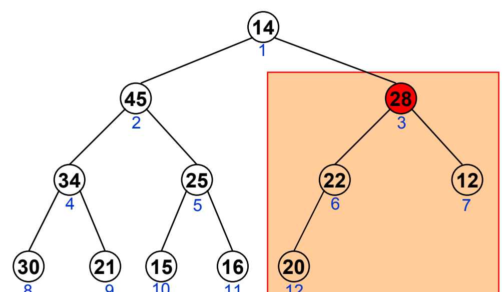
Heapsort: costruzione dell’heap

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



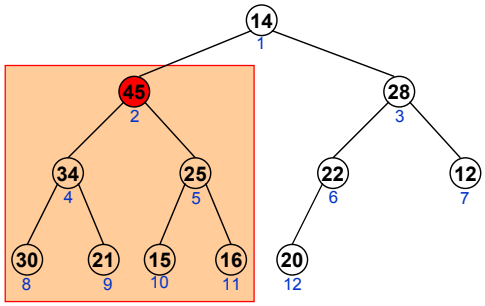
Heapsort: costruzione dell’heap

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



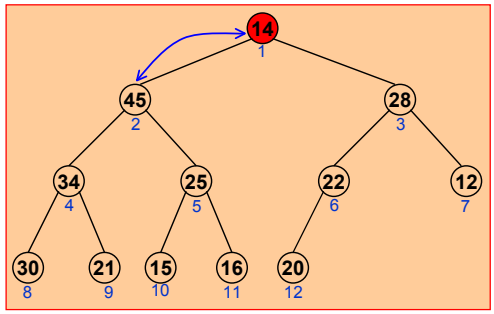
Heapsort: costruzione dell'heap

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



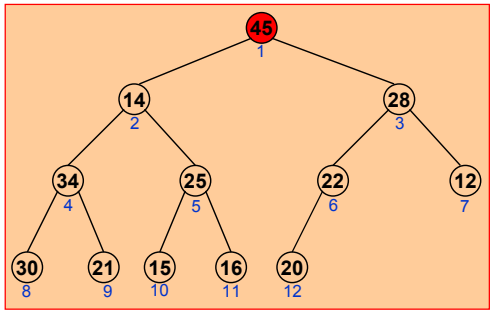
Heapsort: costruzione dell'heap

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



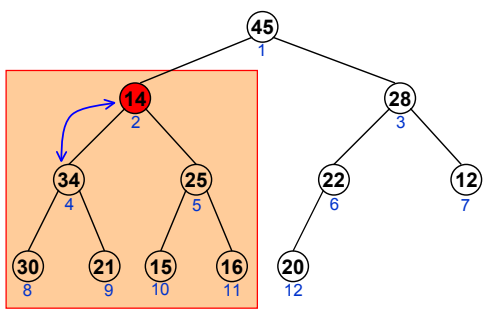
Heapsort: costruzione dell'heap

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



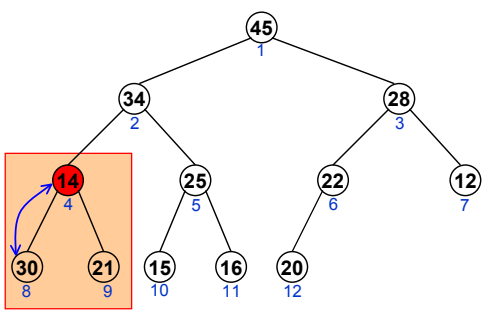
Heapsort: costruzione dell'heap

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



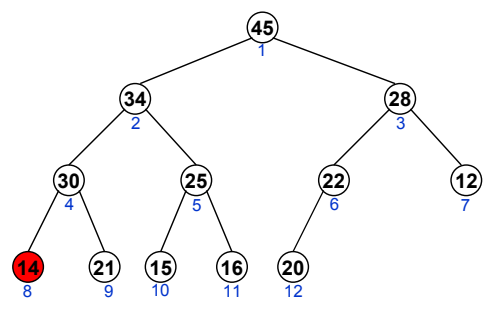
Heapsort: costruzione dell'heap

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

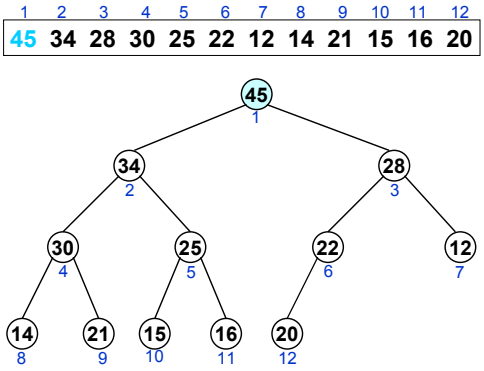


Heapsort: costruzione dell'heap

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



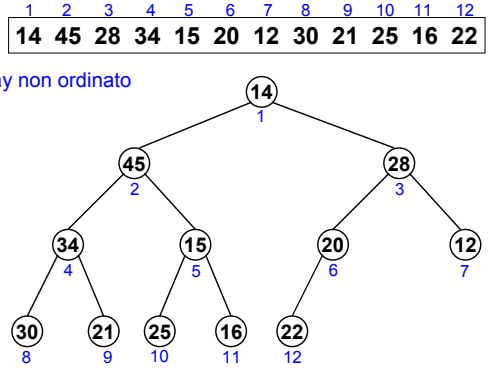
Heapsort: costruzione dell'heap



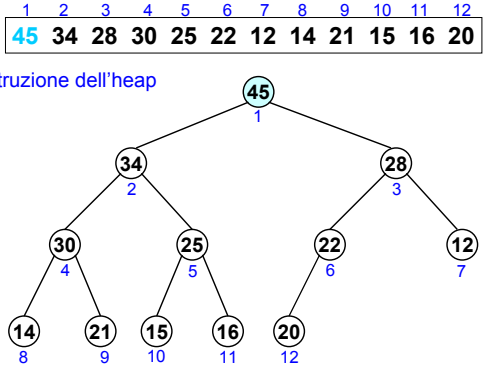
Ordinamento mediante heap

- L'heapsort è una variazione della selezione in cui la ricerca dell'elemento massimo è facilitata dal mantenimento della sequenza in uno heap.
- Si costruisce uno *heap* a partire dall'array non ordinato in input.
- Viene sfruttata la proprietà degli *heap* per cui la radice dello *heap* è sempre il massimo:
  - si scandiscono tutti gli elementi dell'array a partire dall'ultimo e ad ogni iterazione
    - la radice viene scambiata con l'elemento nell'ultima posizione corrente dello *heap*
    - viene ridotta la dimensione dello *heap*
    - viene ripristinato lo *heap* con `adatta`.

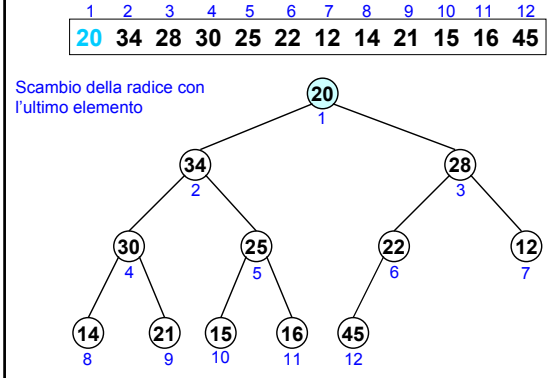
Ordinamento mediante heap



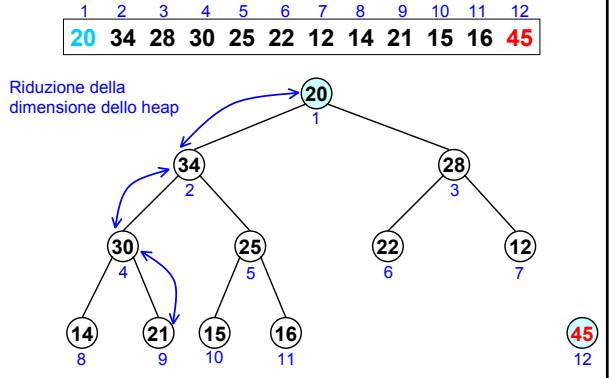
Ordinamento mediante heap



Ordinamento mediante heap



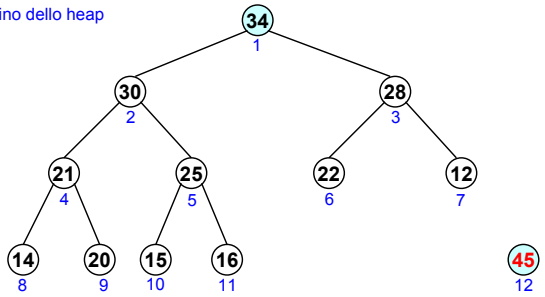
Ordinamento mediante heap



Ordinamento mediante heap

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

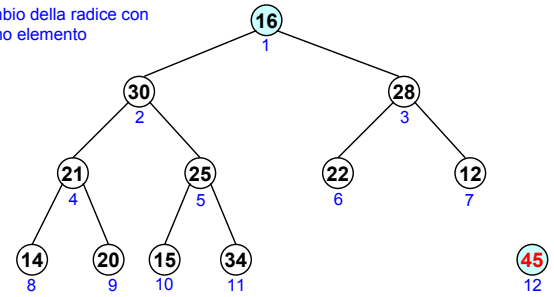
Ripristino dello heap



Ordinamento mediante heap

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

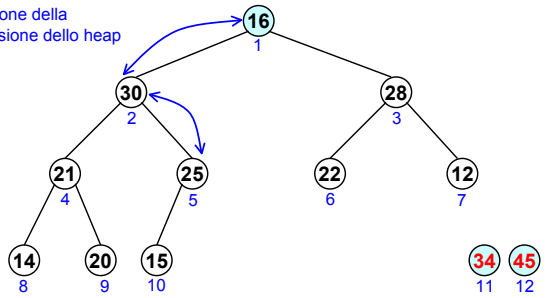
Scambio della radice con l'ultimo elemento



Ordinamento mediante heap

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

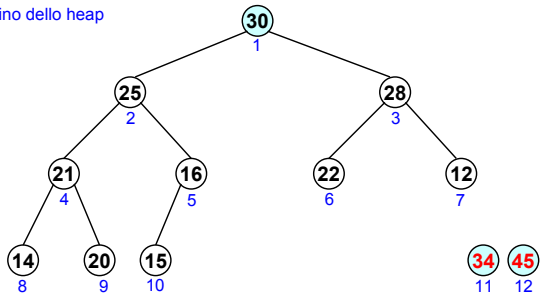
Riduzione della dimensione dello heap



Ordinamento mediante heap

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

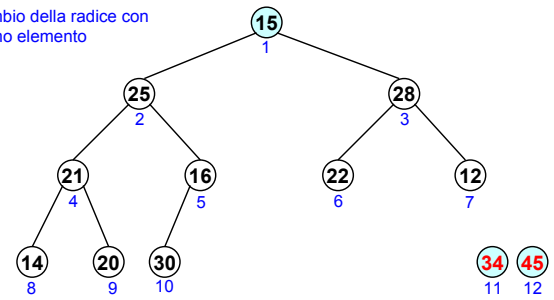
Ripristino dello heap



Ordinamento mediante heap

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

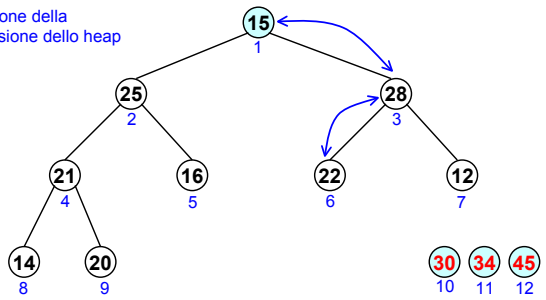
Scambio della radice con l'ultimo elemento



Ordinamento mediante heap

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

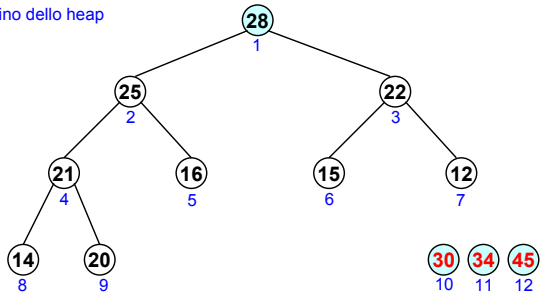
Riduzione della dimensione dello heap



Ordinamento mediante heap

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

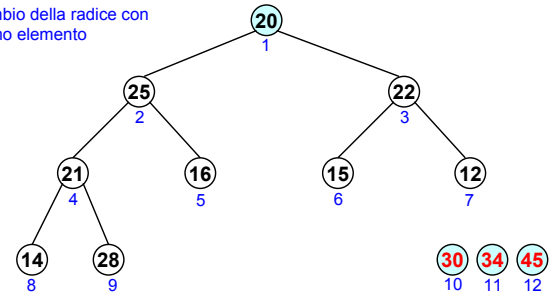
Ripristino dello heap



Ordinamento mediante heap

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

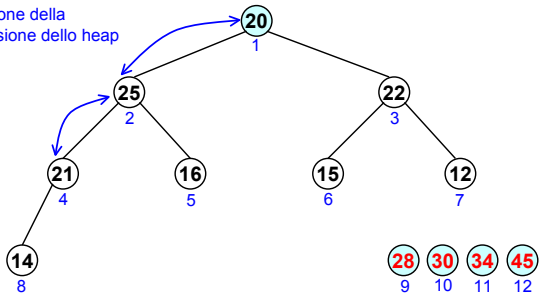
Scambio della radice con l'ultimo elemento



Ordinamento mediante heap

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

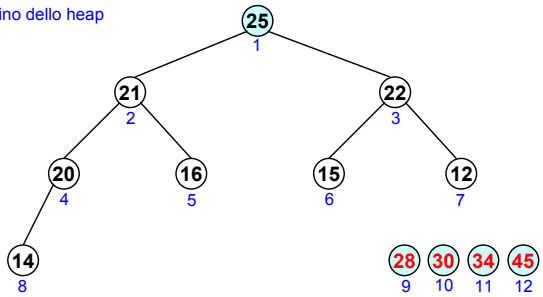
Riduzione della dimensione dello heap



Ordinamento mediante heap

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

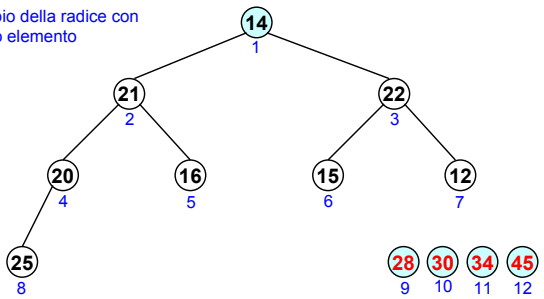
Ripristino dello heap



Ordinamento mediante heap

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

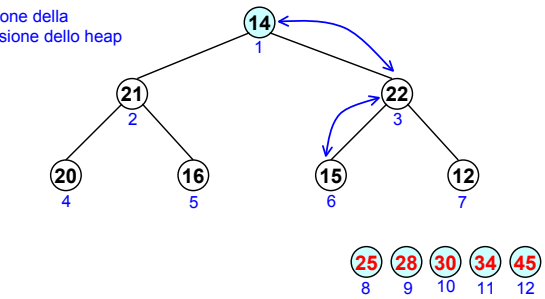
Scambio della radice con l'ultimo elemento



Ordinamento mediante heap

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

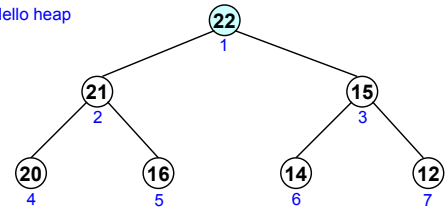
Riduzione della dimensione dello heap



Ordinamento mediante heap

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

Ripristino dello heap

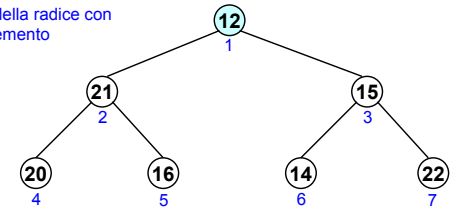


25 8 28 9 30 10 34 11 45 12

Ordinamento mediante heap

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

Scambio della radice con l'ultimo elemento

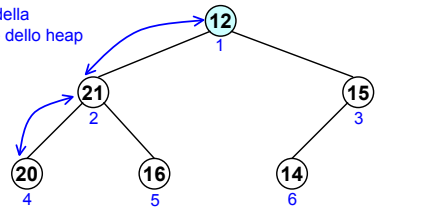


25 8 28 9 30 10 34 11 45 12

Ordinamento mediante heap

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

Riduzione della dimensione dello heap

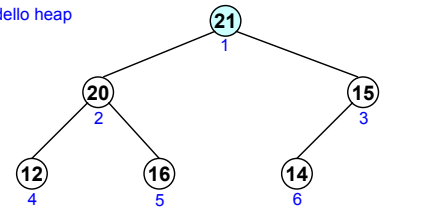


22 7 25 8 28 9 30 10 34 11 45 12

Ordinamento mediante heap

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

Ripristino dello heap

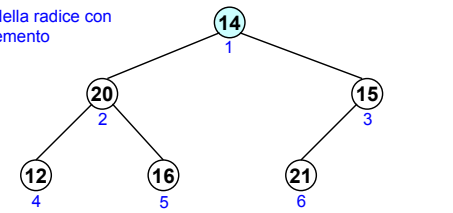


22 7 25 8 28 9 30 10 34 11 45 12

Ordinamento mediante heap

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

Scambio della radice con l'ultimo elemento

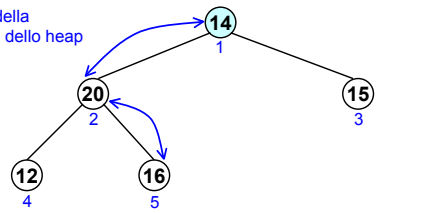


22 7 25 8 28 9 30 10 34 11 45 12

Ordinamento mediante heap

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

Riduzione della dimensione dello heap

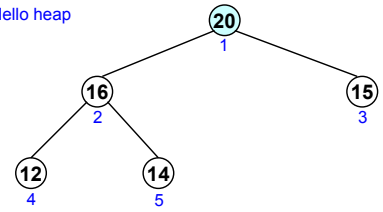


21 6 22 7 25 8 28 9 30 10 34 11 45 12

Ordinamento mediante heap

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

Ripristino dello heap

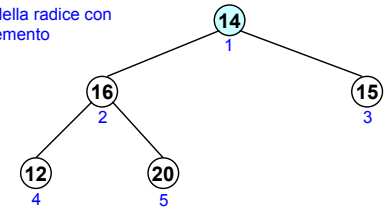


21	22	25	28	30	34	45
6	7	8	9	10	11	12

Ordinamento mediante heap

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

Scambio della radice con l'ultimo elemento

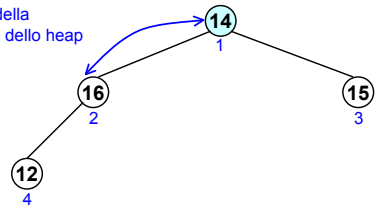


21	22	25	28	30	34	45
6	7	8	9	10	11	12

Ordinamento mediante heap

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

Riduzione della dimensione dello heap

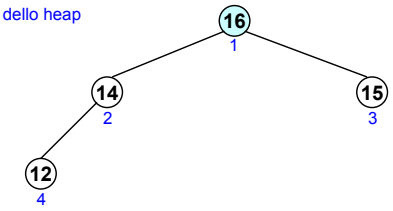


20	21	22	25	28	30	34	45
5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Ripristino dello heap

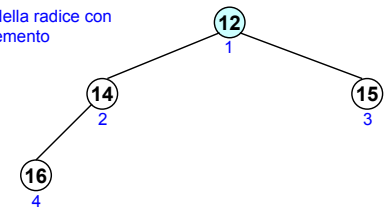


20	21	22	25	28	30	34	45
5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Scambio della radice con l'ultimo elemento

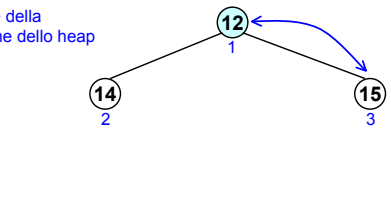


20	21	22	25	28	30	34	45
5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Riduzione della dimensione dello heap

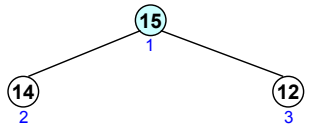


16	20	21	22	25	28	30	34	45
4	5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Ripristino dello heap

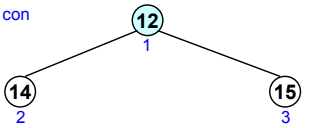


16	20	21	22	25	28	30	34	45
4	5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Scambio della radice con l'ultimo elemento

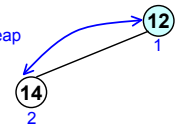


16	20	21	22	25	28	30	34	45
4	5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Riduzione della dimensione dello heap

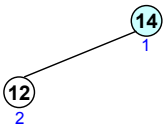


15	16	20	21	22	25	28	30	34	45
3	4	5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Ripristino dello heap

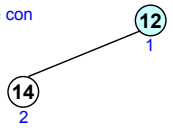


15	16	20	21	22	25	28	30	34	45
3	4	5	6	7	8	9	10	11	12

Ordinamento mediante heap

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

Scambio della radice con l'ultimo elemento



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

Ordinamento mediante heap

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

Riduzione della dimensione dello heap

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

Ordinamento mediante heap

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

L'array è ora ordinato!

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

Ordinamento mediante heap

```
void heapsort(elemento lista[ ], int n)
{
 //      Si costruisce lo heap associato all'insieme da ordinare lista
 //      si estrae di volta in volta il massimo lista[1]
 //      lo si alloca in fondo alla sequenza
 //      e si opera nuovamente sullo heap ridotto di un elemento
 //      si genera cosi la permutazione lista[1]<lista[2]<...<lista[n]

 int i;

 for(i = n/2; i > 0; i--)
        adatta(lista, i, n);

 for(i = n-1; i > 0; i--)  {
        SWAP(&lista[1], &lista[i+1]);  //Scambia lista[1] con lista[i]
        adatta(lista, 1, i);
        }
}
```

Complessità dello heapsort

- Sia n tale che $2^{k-1} \le n < 2^k$, con k l'altezza dell'albero
  - Il numero di nodi al livello i è 2^{i-1}
- Nel primo `for` la funzione `adatta` viene chiamata per ogni nodo cha ha un figlio.
- Il tempo è dato dalla somma, su ogni livello, i, del numero di nodi di un livello, 2^{i-1} , per la distanza massima in cui il nodo può spostarsi, k-i.

```
void heapsort(elemento lista[ ], int n)
{
 int i;

 for(i = n/2; i > 0; i--)
        adatta(lista, i, n);

 for(i = n-1; i > 0; i--)  {
        SWAP(&lista[1], &lista[i+1]);
        adatta(lista, 1, i);
        }
}
```

$$\sum_{i=1}^{k} 2^{i-1}(k-i) = \sum_{i=1}^{k-1} 2^{k-i-1} i = \sum_{i=1}^{k-1} 2^{k-1} 2^{-i} i \le \sum_{i=1}^{k-1} n 2^{-i} i = n \sum_{i=1}^{k-1} \frac{i}{2^i} < 2n = O(n)$$

Complessità dello heapsort

- Nel secondo `for` la funzione `adatta` viene chiamata n-1 volte con altezza massima $\lceil \log_2(n+1) \rceil$.
- Il tempo è quindi $O(n \log_2 n)$.

```
void heapsort(elemento lista[ ], int n)
{
 int i;

 for(i = n/2; i > 0; i--)
        adatta(lista, i, n);   } = O(n)

 for(i = n-1; i > 0; i--)  {
        SWAP(&lista[1], &lista[i+1]);
        adatta(lista, 1, i);
        }
}
```

La complessità dello heapsort è $O(n \log_2 n)$

Ordinamento con radice

- Consente di ordinare una lista di record con più campi chiave.
- Siano K^0, K^1, …, K^{r-1}, le r chiavi su cui effettuare l'ordinamento
  - K^0 è la chiave più significativa
  - K^{r-1} è la chiave meno significativa
- Una lista di record R_0, R_1, …, R_{n-1} è ordinata rispetto alle chiavi K^0, K^1, …, K^{r-1} se e soltanto se

$$\left(K_i^0, K_i^1, \ldots, K_i^{r-1}\right) \le \left(K_{i+1}^0, K_{i+1}^1, \ldots, K_{i+1}^{r-1}\right) \quad \text{con } 0 \le i < n-1$$

- Un ordinamento per chiavi multiple può procedere
  - iniziando dal campo più significativo, MSD (Most Significant Digit)
  - iniziando dal campo meno significativo, LSD (Least Significant Digit)

Ordinamento con radice

- L'ordinamento di un mazzo di carte su 2 campi chiave, seme e valore con
  - K^0 seme: ♣<♦<♥<♠
  - K^1 valore: 2<3<….<10<J<Q<K<A

  produce la sequenza 2♣,…,A♣, 2♦,…,A♦,…,2♠,…,A♠
- Ordinando le carte con un metodo MSD,
  - otteniamo 4 pile di carte tutte dello stesso seme, che devono essere poi ordinate separatamente in base al valore delle carte
  - le pile devono essere riunite in modo che la pila di ♠ si trovi in fondo al mazzo e la pila di ♣ si trovi in cima.
- Ordinando le carte con un metodo LSD,
  - otteniamo le 13 pile di carte tutte dello stesso valore, da riunire in una singola pila ponendo i 2 in cima e gli assi in fondo
  - le carte devono essere riordinate in base al seme.

Ordinamento con radice

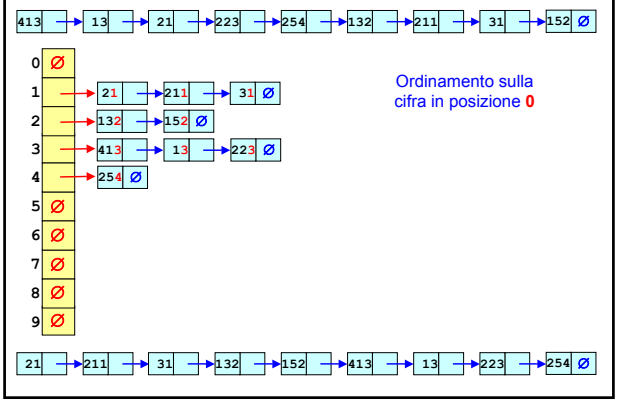
- Nell'ordinamento vengono creati dei recipienti (bin) per rappresentare i differenti valori delle chiavi.
- E' possibile utilizzare un ordinamento MSD o LSD anche in presenza di un unico campo chiave.
- Un numero intero è costituito da più cifre K^i, ordinate dalla meno significativa, in ultima posizione, alla più significativa, in prima posizione.
  - ogni cifra si trova nell'intervallo $0 \le K^i \le 9$, l'ordinamento richiede 10 bin.
- In un ordinamento con radice la chiave di ordinamento viene scomposta in cifre utilizzando una radice r
  - con una radice r, occorrono r bin per effettuare l'ordinamento su ogni cifra.

Ordinamento con radice

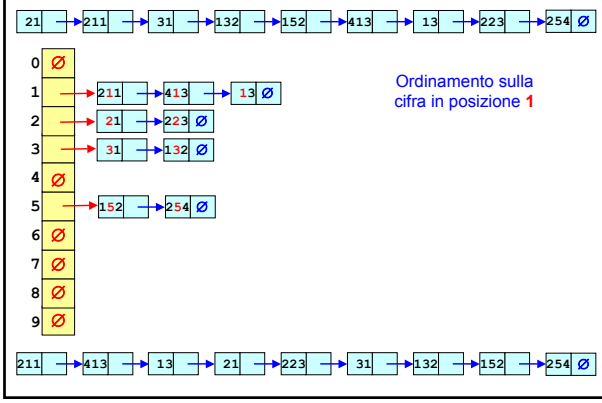
- Siano $R_0, R_1, \ldots, R_{n-1}$ i record da ordinare, rappresentati mediante una lista concatenata.
- Ogni record ha chiavi formate da d elementi $x_0, x_1, \ldots, x_{d-1}$ con $0 \le x_i < r$.
- I bin sono implementati come code di dati
  - davanti[i], $0 \le i < r$, punta al primo record nel bin i
  - dietro[i], $0 \le i < r$, punta all'ultimo record nel bin i.

```
#define MAX_CIFRE 3  // numeri compresi tra 0 e 999
#define RADICE_SIZE 10

typedef struct list_node *list_pointer;
typedef struct list_node {
        int chiave[MAX_CIFRE];
        list_pointer link;
        }lista;
```

Ordinamento con radice: esempio



Ordinamento con radice: esempio



Ordinamento con radice: esempio

Ordinamento con radice

```
list_pointer radice_sort(list_pointer ptr)
{
 list_pointer davanti[RADICE_SIZE],  dietro[RADICE_SIZE];
 int i, j, cifra;

 for(i = MAX_CIFRE-1;  i >=0;  i--) {
      for(j = 0;  j < RADICE_SIZE; j++)
           davanti[j] = dietro[j] = NULL;
      while(ptr)  {
           cifra = ptr->chiave[i];
           if (!davanti[cifra])     davanti[cifra] = ptr;
           else                     dietro[cifra]->link = ptr;
           dietro[cifra] = ptr;
           ptr = ptr->link;
           }
      // ristabilisce la lista concatenata per il passo succ
       ptr = NULL;
       for (j = RADICE_SIZE-1; j>=0; j--)
           if(davanti[j])  {
                  dietro[j]->link = ptr;
                  ptr = davanti[j];
                  }
  }
 return ptr;
}
```

$O(MAX\_CIFRE(RADICE\_SIZE+n))$