

La struttura di dati coda di priorità (heap)

- E' un particolare insieme, costituito da elementi che dispongono di una proprietà (chiave) sulla quale è definita una relazione di ordinamento totale
- Operazioni consentite (Min-Priorità):
 - Inserimento di un nuovo elemento: $S \leftarrow S \cup \{e\}$
 - Selezione minimo: restituisce l'elemento di S con la chiave x più piccola
 - Cancellazione minimo: restituisce l'elemento di S che ha la chiave x più piccola e lo elimina dall'insieme
 $S \leftarrow S - \{e\}$, con $e = \text{MIN}(S)$ rispetto a x
 - Cancellazione di un elemento
 - Incremento (decremento) della chiave: incrementa (decrementa) il valore della chiave x di e di una quantità k
- Max-Priorità: stesse operazioni sostituendo massimo a minimo

Applicazioni della Coda di Priorità

- Max-Priorità: programmare la sequenza di esecuzione di operazioni su risorse condivise (ad esempio un computer)
- Min-Priorità: gestione di eventi, dove la chiave rappresenta il tempo (es. coda di un pronto soccorso)

Esempio

Gestione di processi: ad ogni processo viene associata una priorità. Una coda con priorità permette di conoscere in ogni istante il processo con priorità maggiore. In qualsiasi momento i processi possono essere eliminati dalla coda o nuovi processi con priorità arbitraria possono essere inseriti nella coda.

Per implementare efficientemente una coda con priorità utilizzeremo una struttura dati chiamata *heap*

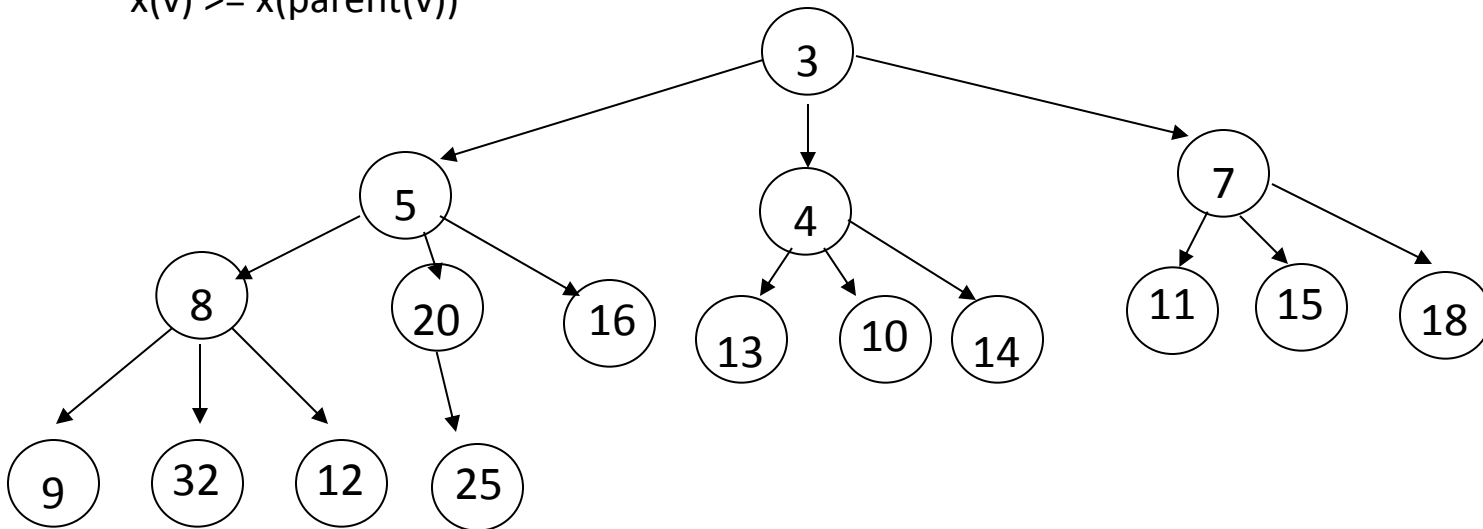
Esistono diverse implementazioni della coda di Priorità:

- D-heap (generalizzazione degli heap binari)
- Heap Binomiali
- Heap di Fibonacci

Struttura dati Heap: d-heap

Una d-heap e' un albero radicato d-ario che:

1. E' quasi completo: completo almeno fino al penultimo livello
2. Ogni nodo v contiene un elemento e ed una chiave $x(v)$ sul cui dominio e' definita una relazione di ordinamento totale
3. Ogni nodo n diverso dalla radice ha la chiave non minore del padre
 $x(v) \geq x(\text{parent}(v))$

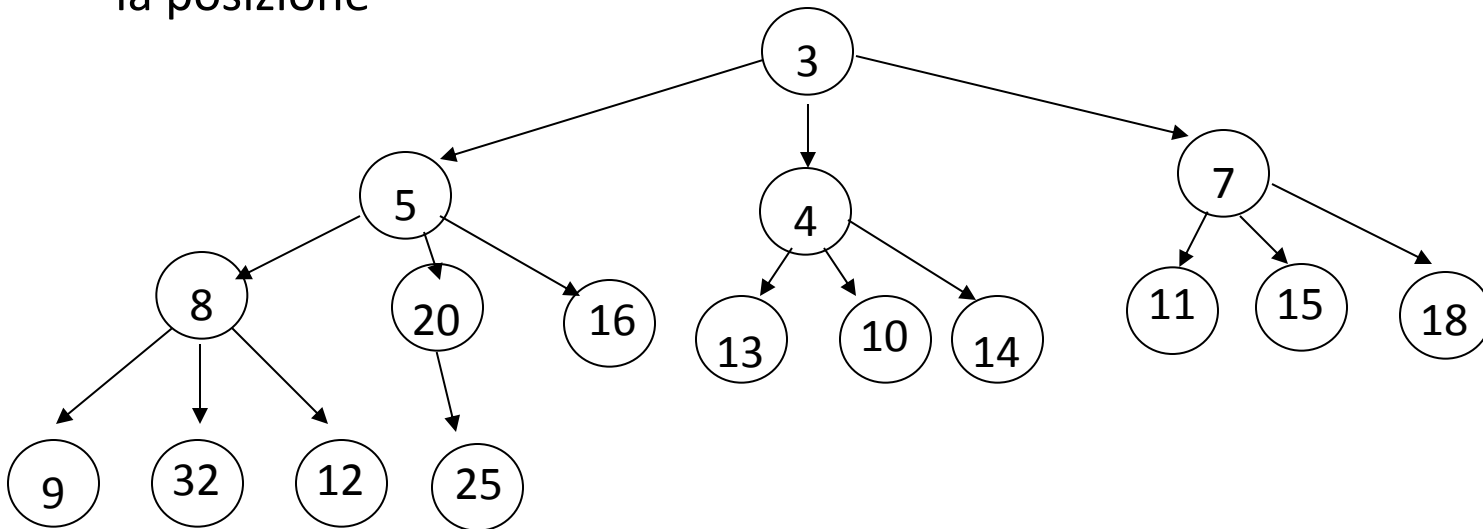


d-heap con $d = 3$ e 17 nodi

Struttura dati Heap: d-heap

Proprietà:

- Dato un d-heap con n nodi, l'albero ha altezza $O(\log_d n)$
- La radice dell'albero contiene sempre la chiave di valore minimo (o massimo), grazie alla proprietà 3 (ordinamento heap)
- Può essere rappresentato con un vettore considerando in modo implicito la posizione



Struttura dati Heap: d-heap

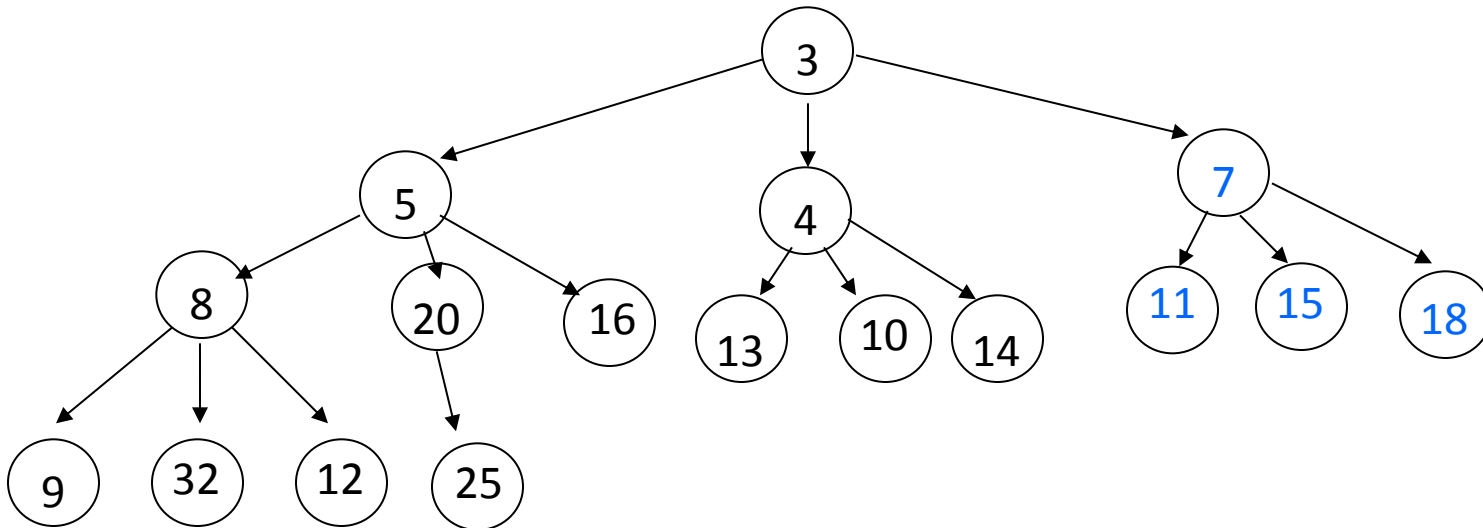
Esempio di rappresentazione vettoriale:

Con $d = 3$

Dato il padre i , i figli sono

$3i - 1, 3i, 3i + 1$ (in generale $d*i - d + 2, \dots, d*i + 1$)

Chiave	3	5	4	7	8	20	16	13	10	14	11	15	18	9	32	12	25
Pos.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17



Struttura dati Heap: d-heap

Operazioni fondamentali

FindMin(T)	trova il minimo dell'insieme T
Insert(elemento e, chiave x)	inserisce l'elemento e in T
DeleteMin()	elimina il minimo dell'insieme T
Delete(elemento e)	elimina e dall'insieme T
Increase(elemento e, valore d)	incrementa di d la chiave x di e
Decrease(elemento e, valore d)	decrementa di d la chiave x di e

Procedure di supporto: utili a riottenere la proprietà di ordinamento heap per nodi v che la violano

MoveUp(v), MoveDown(v)

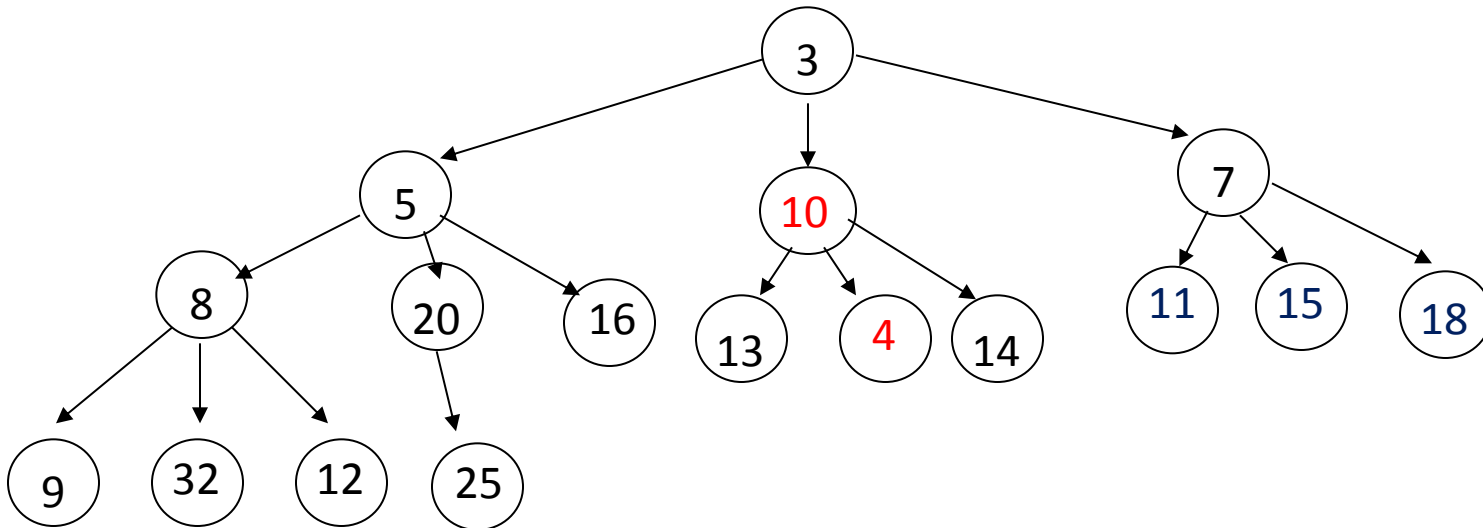
MoveUp(nodo v)

Dato un nodo v , lo scambia con il padre finchè v non soddisfa l'ordinamento a heap

MoveUp(v)

While ($v \neq \text{root}(T)$ and $x(v) < x(\text{parent}(v))$)

Scambia v e $\text{parent}(v)$ in T



MoveUp(nodo v)

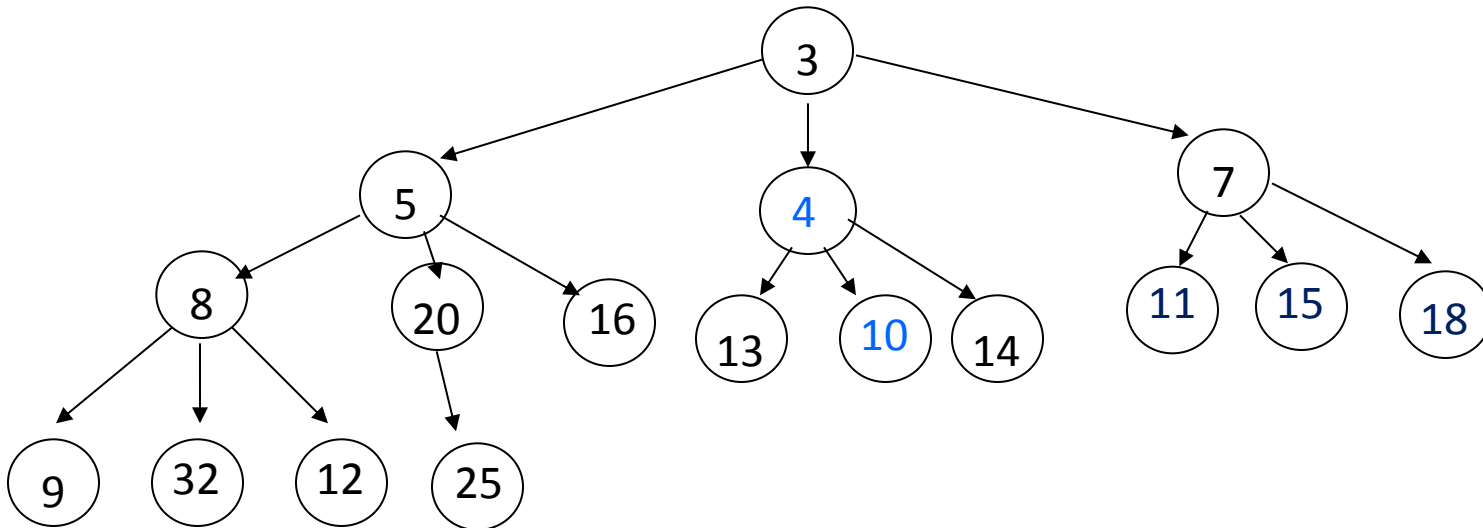
Dato un nodo v , lo scambia con il padre finchè v non soddisfa l'ordinamento a heap

MoveUp(v)

While ($v \neq \text{root}(T)$ and $x(v) < x(\text{parent}(v))$)

Scambia v e $x(\text{parent}(v))$ in T

$T(n) = O(\log_d n)$



MoveDown(nodo v)

Dato un nodo v , lo scambia con il minore tra i figli finchè v non soddisfa l'ordinamento a heap

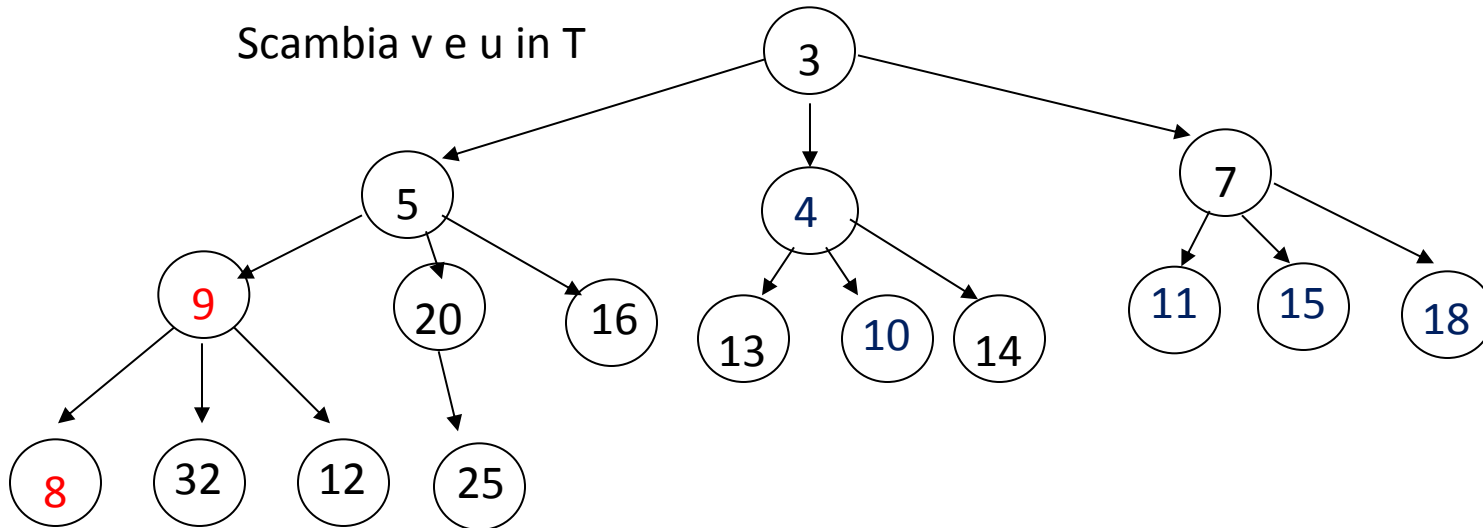
MoveDown(v)

Repeat

Sia u il figlio di v con $x(u)$ minima

If (v non ha figli o $x(v) \leq x(u)$) termina

Scambia v e u in T



MoveDown(nodo v)

Dato un nodo v , lo scambia con il minore tra i figli finchè v non soddisfa l'ordinamento a heap

MoveDown(v)

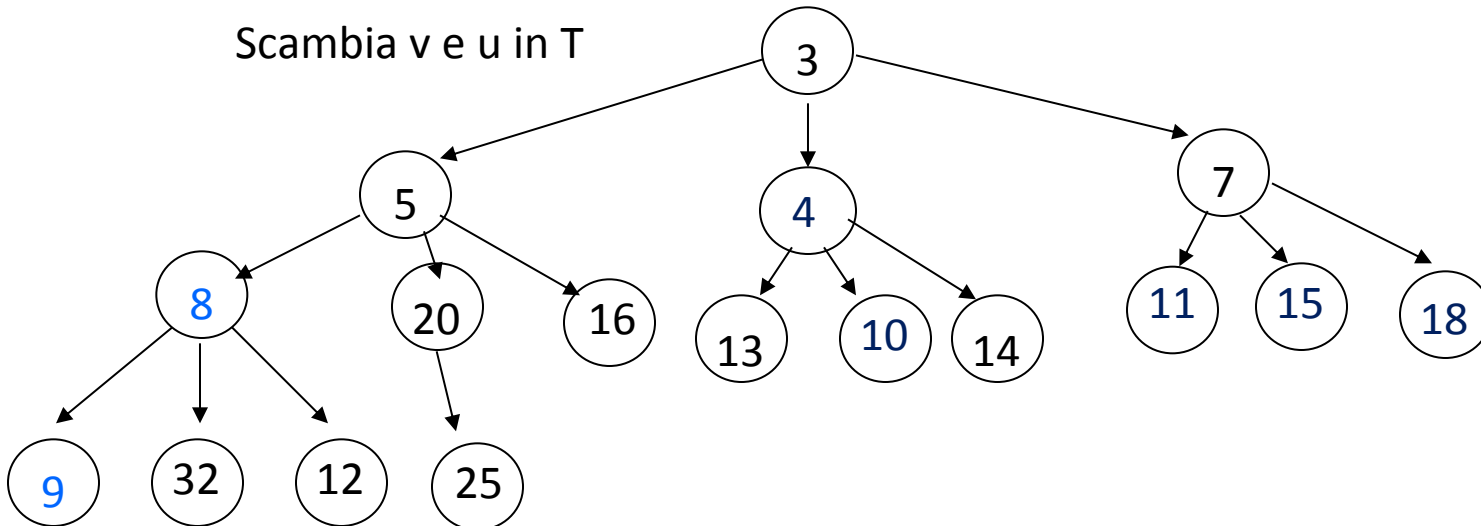
Repeat

Sia u il figlio di v con $x(u)$ minima

If (v non ha figli o $x(v) \leq x(u)$) termina

Scambia v e u in T

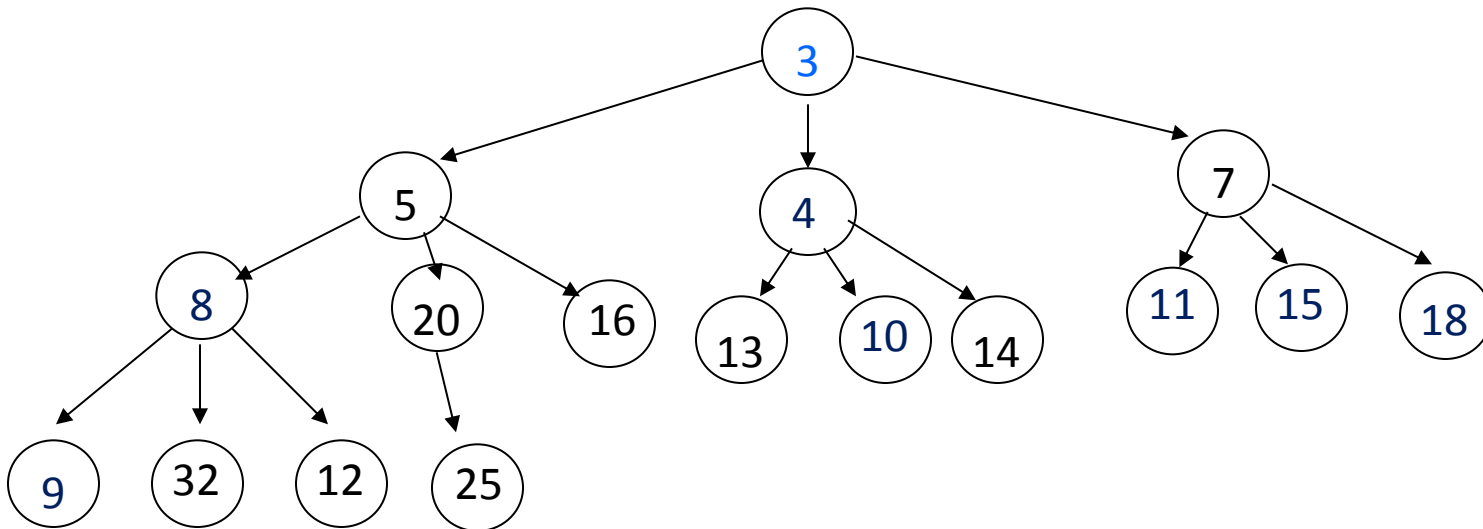
$$T(n) = O(d * \log_d n)$$



FindMin(T)

Restituisce l'elemento radice di T

$$T(n) = O(1)$$

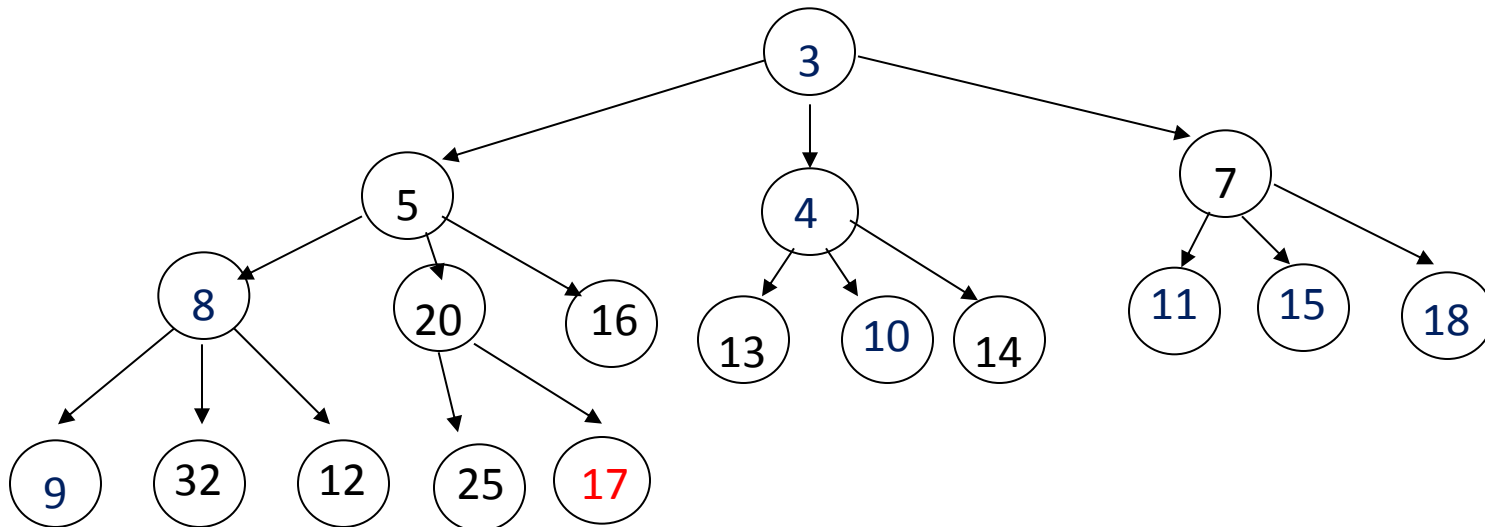


Insert(elemento e, chiave x)

Occorre creare un nuovo nodo v contenente un elemento e di chiave x come foglia (qualsiasi) di T .

Tale foglia deve rispettare la proprietà di ordinamento heap tramite la chiamata MoveUp, che opera gli scambi necessari

$T(n) = O(\log_d n)$ dovuta a MoveUp

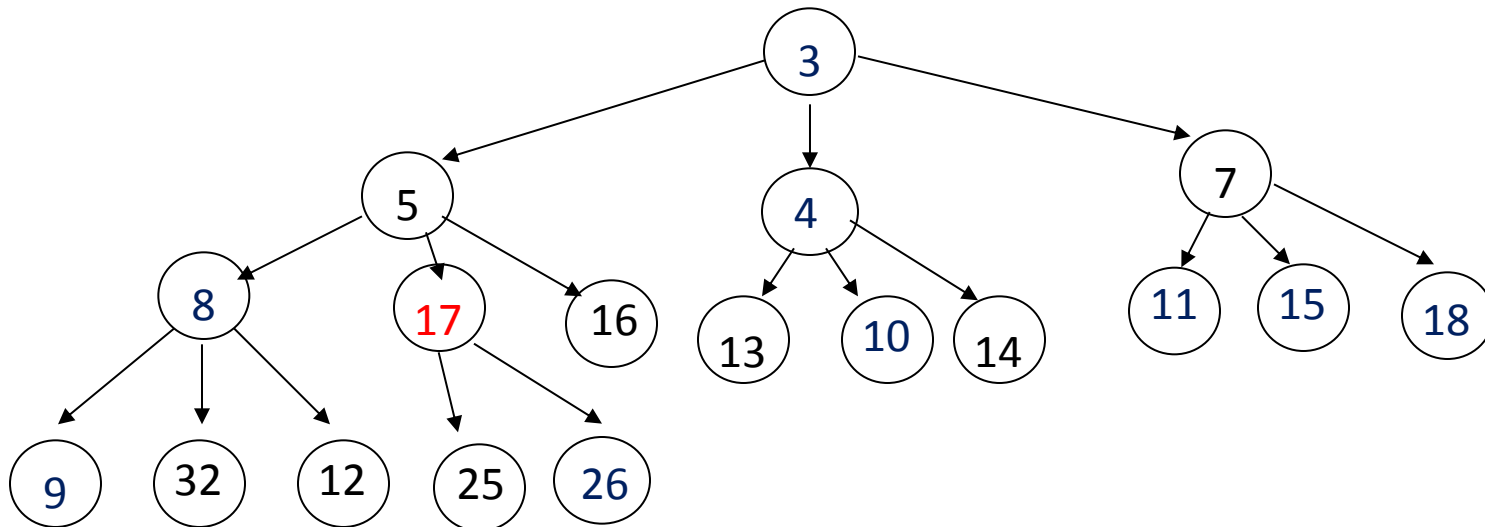


Delete(elemento e) deleteMin()

Viene scambiato il nodo v dell'elemento e (o la radice) con una foglia qualunque p , poi elimina p .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveDown}(v)$

$T(n) = O(d * \log_d n)$ dovuta a MoveDown

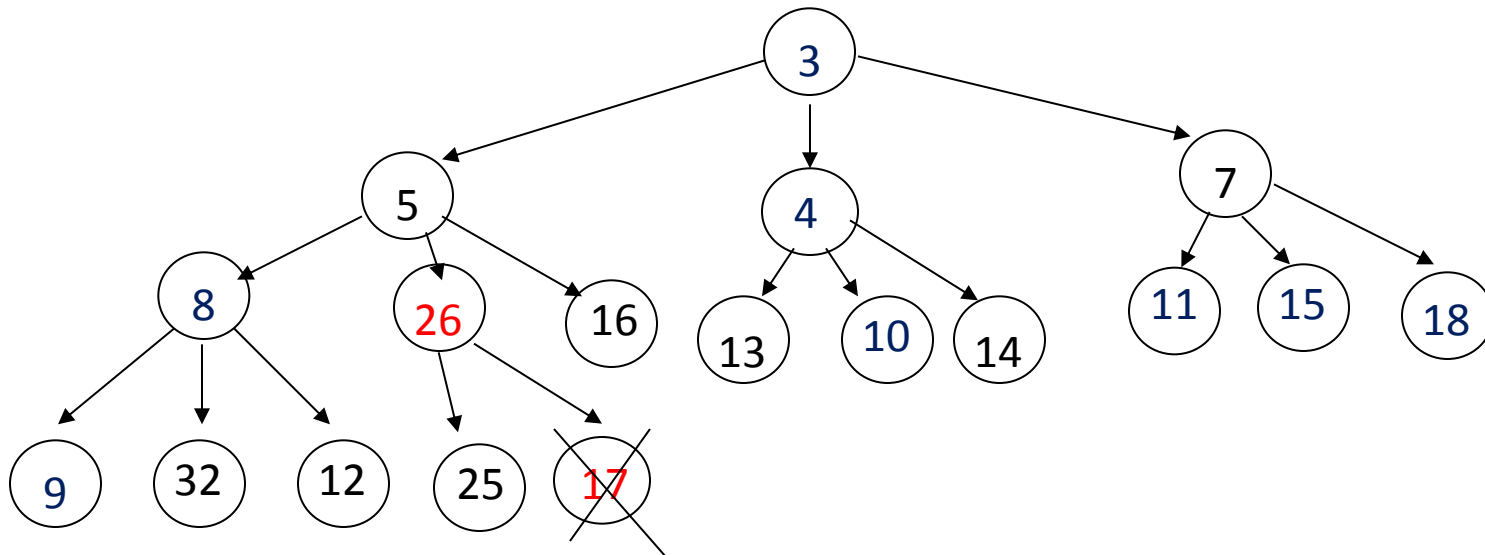


Delete(elemento e) deleteMin()

Viene scambiato il nodo v dell'elemento e (o la radice) con una foglia qualunque p , poi elimina p .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveDown}(v)$

$T(n) = O(d * \log_d n)$ dovuta a MoveDown

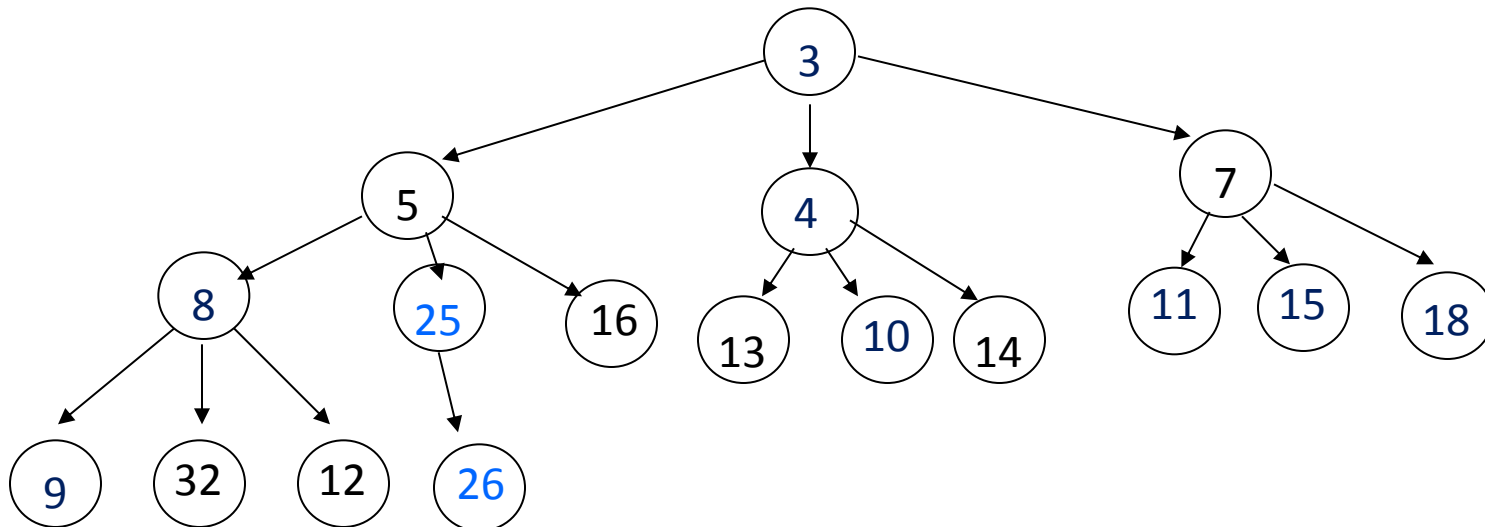


Delete(elemento e) deleteMin()

Viene scambiato il nodo v dell'elemento e (o la radice) con una foglia qualunque p , poi elimina p .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveDown}(v)$

$T(n) = O(d * \log_d n)$ dovuta a MoveDown

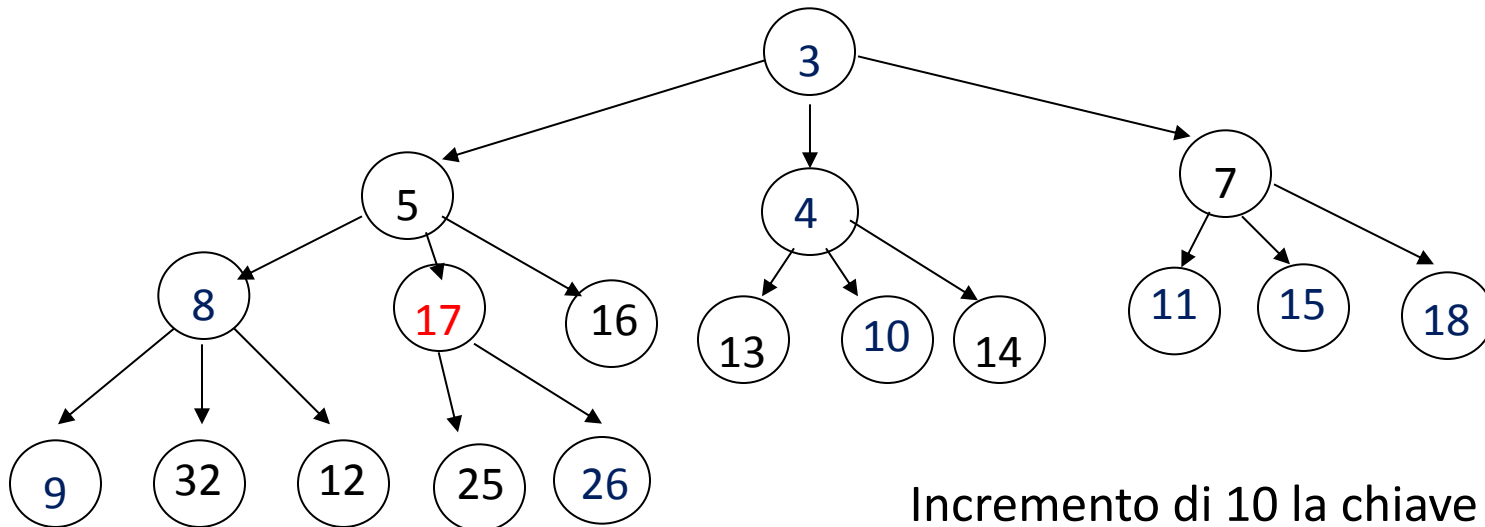


Increase(elemento e, valore d)

Incrementa di d il valore della chiave x del nodo v contenente l'elemento e .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveDown}(v)$

$T(n) = O(d * \log_d n)$ dovuta a MoveDown

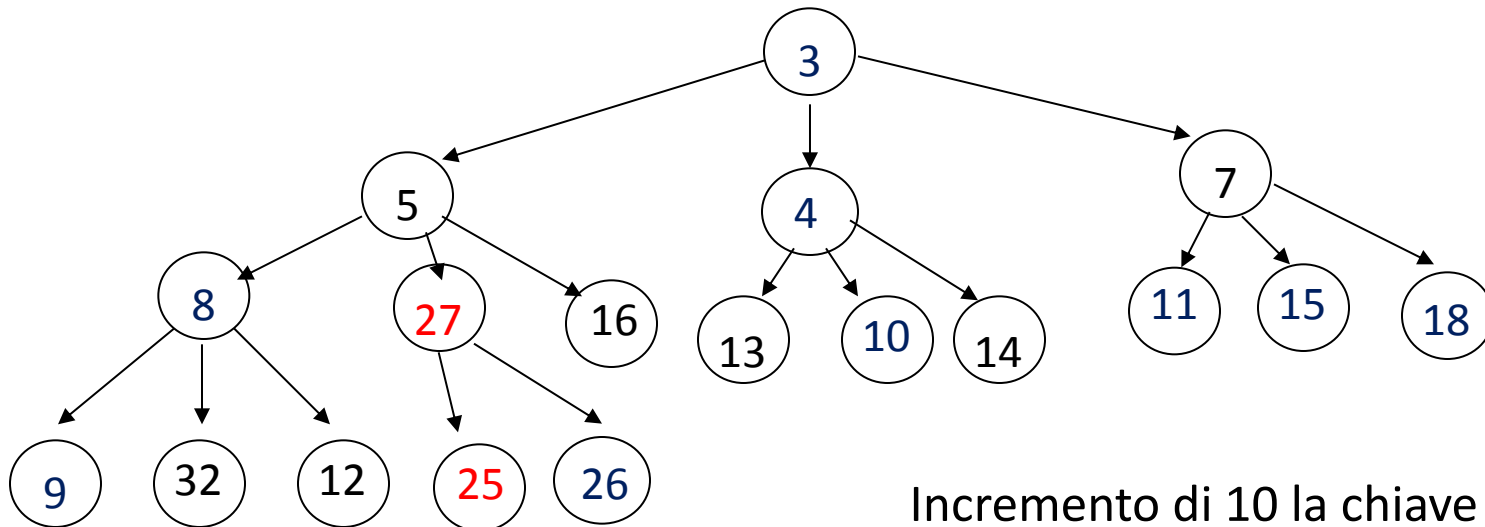


Increase(elemento e, valore d)

Incrementa di d il valore della chiave x del nodo v contenente l'elemento e .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveDown}(v)$

$T(n) = O(d * \log_d n)$ dovuta a MoveDown

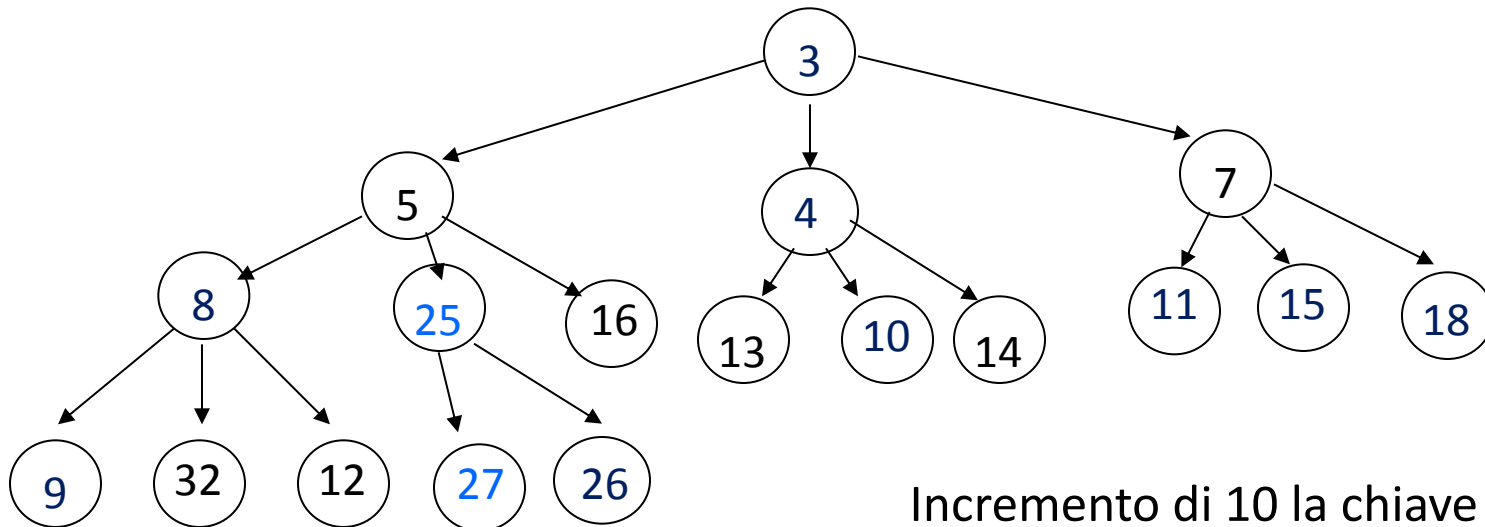


Increase(elemento e, valore d)

Incrementa di d il valore della chiave x del nodo v contenente l'elemento e .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveDown}(v)$

$T(n) = O(d * \log_d n)$ dovuta a MoveDown



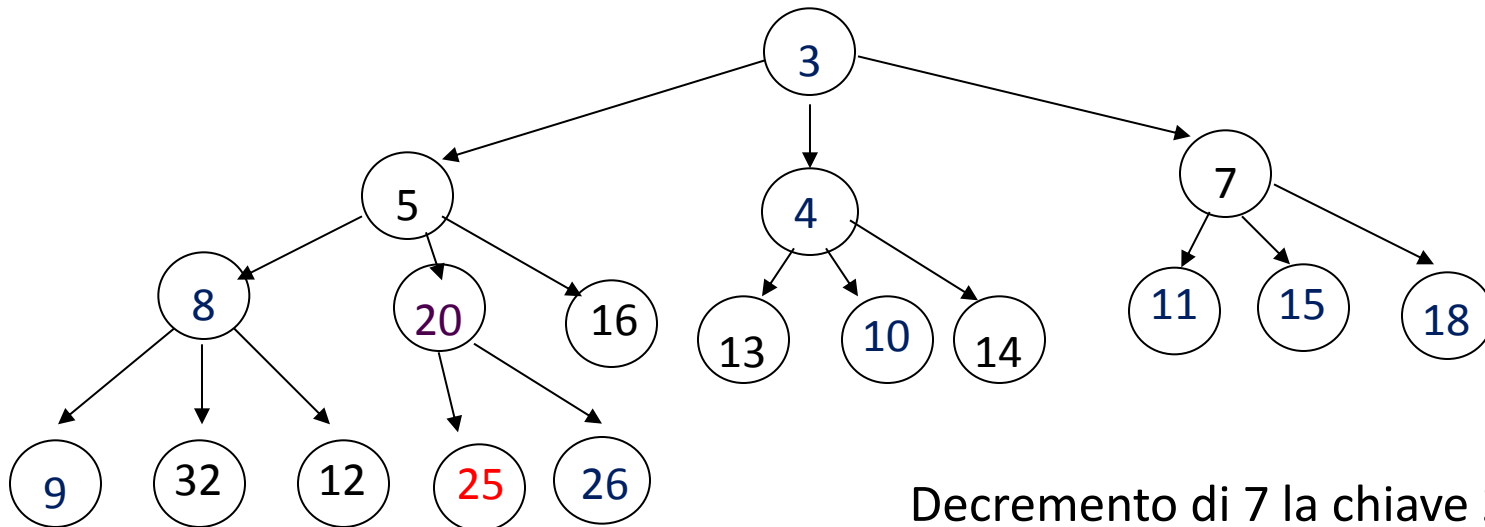
Incremento di 10 la chiave 17

Decrease(elemento e, valore d)

Decrementa di d il valore della chiave x del nodo v contenente l'elemento e .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveUp}(v)$

$T(n) = O(\log_d n)$ dovuta a MoveUp

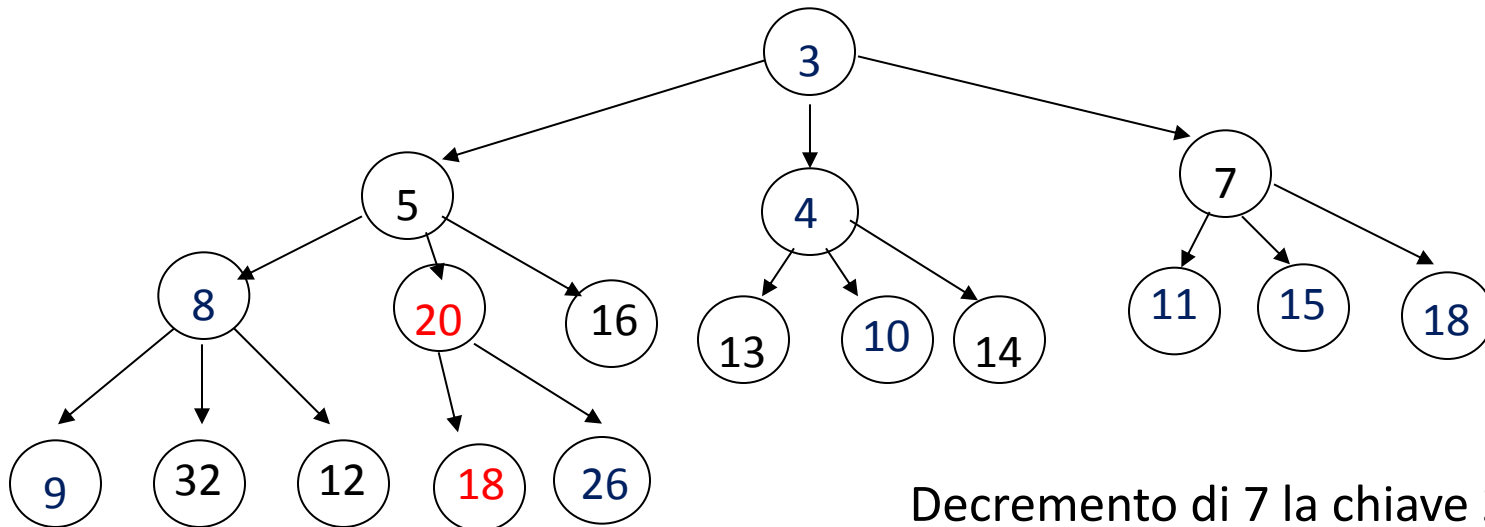


Decrease(elemento e, valore d)

Decrementa di d il valore della chiave x del nodo v contenente l'elemento e .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveUp}(v)$

$T(n) = O(\log_d n)$ dovuta a MoveUp



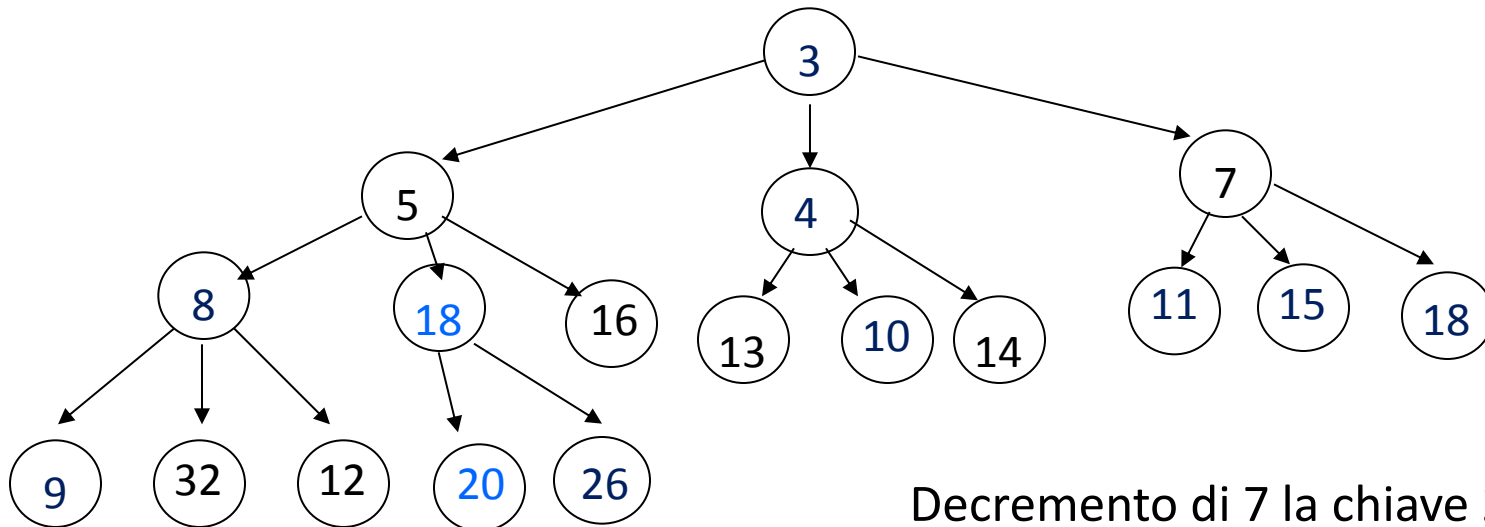
Decremento di 7 la chiave 25

Decrease(elemento e, valore d)

Decrementa di d il valore della chiave x del nodo v contenente l'elemento e .

L'ordinamento heap viene ripristinato attraverso la procedura $\text{MoveUp}(v)$

$T(n) = O(\log_d n)$ dovuta a MoveUp



Decremento di 7 la chiave 25

Heap Binaria (max)

Una heap binaria è un albero radicato binario T che risulta:

1. Completo almeno fino al penultimo livello (le foglie sono compattate a sinistra)
2. Ogni nodo v contiene un elemento e ed una chiave $x(v)$ sul cui dominio è definita una relazione di ordinamento totale
3. Ogni nodo n diverso dalla radice ha la chiave non maggiore del padre
$$x(v) \leq x(\text{parent}(v))$$

Proprietà

- Il massimo è contenuto nella radice di T
- L'altezza di T è $O(\log_2 n)$

Heap Binaria

ESEMPIO

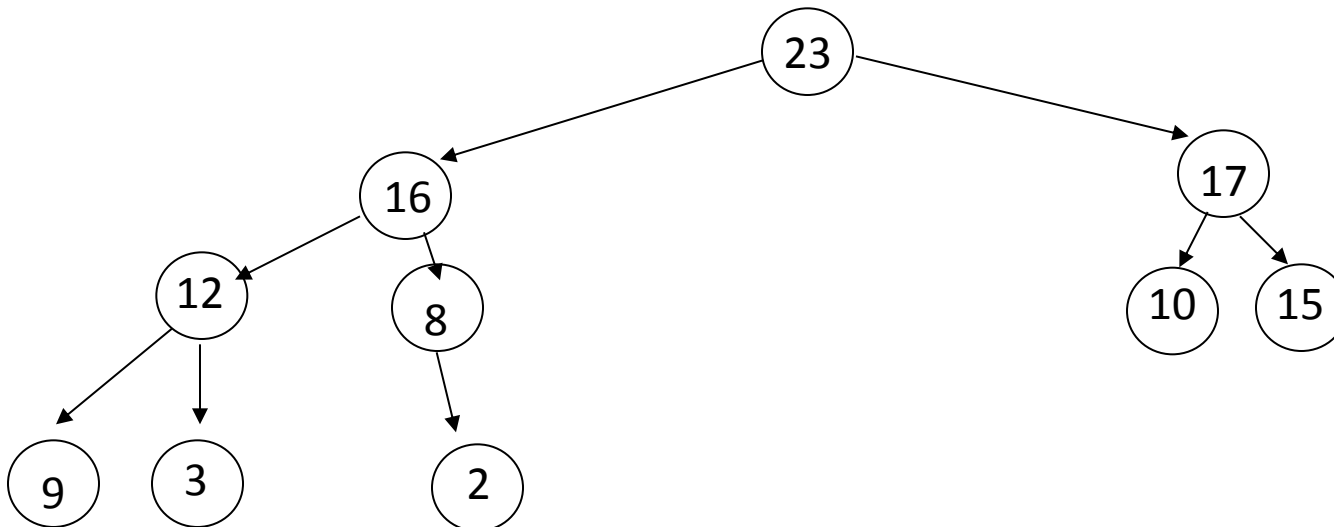
Albero con 10 nodi

Chiave	23	16	17	12	8	10	15	9	3	2
Pos.	1	2	3	4	5	6	7	8	9	10

Padre $i = 4$

$\text{FiglioSinistro}(i) = 2*i = 8$

$\text{FiglioDestro}(i) = 2*i + 1 = 9$



Heap Binaria con vettore

- La radice dell'albero è $A[1]$
- L'indice del padre di un nodo di posizione i è $i/2$ (estremo inferiore)
- L'indice del figlio sinistro di un nodo i è $2i$
- L'indice del figlio destro di un nodo i è $2i + 1$

Parent(i)
 return $i/2$;

Left(i)
 return $2i$;

Right(i)
 return $2i + 1$;

Costruzione Heap binaria

- A seguito di varie operazioni sullo heap può accadere che un nodo violi la proprietà dello heap
- La procedura **Heapify** prende in ingresso uno heap A e l'indice i di un nodo che potenzialmente viola la proprietà e ristabilisce la proprietà di ordinamento parziale sull'intero heap
- Si assume che i sottoalberi figli del nodo i siano radicidi heap che rispettano la proprietà di ordinamento parziale

- L'idea è di far “**affondare**” il nodo che viola la proprietà di ordinamento parziale fino a che la proprietà non viene ripristinata
- Per fare questo si determina il nodo figlio più grande e si scambia il valore della chiave fra padre e figlio
- Poi si procede ricorsivamente sul nodo figlio per cui e' avvenuto lo scambio

Vediamo come cambia MoveDown nella Heap Binaria

MoveDown(v , T)

 If v è foglia return

 else

 If ($x(\text{left}(v)) > x(\text{right}(v))$) then $u = \text{left}(v)$
 else $u = \text{right}(v)$

 If ($x(v) \leq x(u)$) then
 Scambia v e u in T
 MoveDown(u , T)

$T(n) = O(\log_2 n)$

Costruzione Heap binaria

Dato un albero T **Heapify** lo rende una Heap Binaria

Heapify(T)

 If (T è vuoto) return

 Else

 heapify(left(T))

 heapify(right(T))

 MoveDown(root(T), T)

n dimensione input

2 numero chiamate ricorsive all'algoritmo

n/2 dimensione dell'input chiamata ricorsiva

$O(\log_2 n)$ costo di suddivisione e ricostruzione soluzione
(MoveDown)

$$C = O(n) + O(\log_2 n) = O(n)$$

Coda di priorit  con Heap binaria

Risulta semplice implementare le varie operazioni di una coda con priorit  utilizzando uno heap

- Extract Max: basta restituire la radice dello heap
- Heap Extract Max: dopo la restituzione dell'elemento massimo, posiziona l'ultimo elemento dello heap (non il pi  piccolo!) nella radice ed esegue Heapify per ripristinare la propriet  di ordinamento parziale
- Heap Insert: la procedura inserisce il nuovo elemento come elemento successivo all'ultimo e lo fa salire fino alla posizione giusta facendo "scendere" tutti padri

Uso della struttura dati Heap

Algoritmo HeapSort

Basato su una heap binaria

1. Costruisci heap mediante heapify $O(n)$
2. Estrai il massimo per $n - 1$ volte $O(n * \log_2 n)$
Memorizzandolo nella posizione liberata (l'ultima occupata dal vettore)

Complessità: $O(n * \log_2 n)$

Implementazione in linguaggio C

```
void heapify( int *A, int i, int heap_size ) {  
    int l, r, largest = i;  
    l = left(i);  
    r = right(i);  
    if (( l <= heap_size ) && ( A[l] > A[i] ))  
        largest = l;  
    if (( r <= heap_size ) && ( A[r] > A[largest] ))  
        largest = r;  
    if ( largest != i ) {  
        scambia( &A[i], &A[largest] );  
        heapify( A, largest, heap_size );  
    }  
    return;}  

```

Implementazione in linguaggio C

```
void build_heap( int *A, int heap_size ) {  
    int i;  
    for( i = heap_size/2; i >= 1; i-- )  
        heapify( A, i, heap_size );  
    return;  
}  
void heapsort( int *A, int dim ) {  
    int i, heap_size = dim;  
  
    for ( i = dim; i >= 2; i-- ) {  
        scambia( &A[1], &A[i] );  
        heapify( A, 1, --heap_size );  
    }  
}
```

Implementazione in linguaggio C

```
int riempivettore(int *A) {  
    int i;  
    for( i = 1; i < MAX; i++ )  
        A[i] = MAX - i;  
    return MAX-1;  
}
```

```
void stampavettore(int *A) {  
    int i;  
    printf("\n");  
    for( i = 1; i < MAX; i++ )  
        printf(" %d", A[i]);  
    return;  
}
```

Implementazione in linguaggio C

```
void main()
{
    int vettore[MAX];
    int n; /*numero di elementi*/

    n=riempivettore(vettore);
    build_heap(vettore,n);
    heapsort(vettore,n);
    stampavettore(vettore);

    return;
}
```

Coda Heap in linguaggio C

```
void MoveUp( int *A, int i, int heap_size ){  
  
    while (i!=1 && A[i] > A[Parent(i)]){  
        scambia( &A[i], &A[Parent(i)] );  
        i = Parent(i);  
    }  
}
```

Coda Heap in linguaggio C

```
void main() {
    int vettore[MAX], int n; /*numero di elementi*/
    int scelta, int el; char invio;
    n=riempivettore(vettore);
    build_heap(vettore,n);
    do {
        printf("\n\nMenu' Principale - CODA HEAP\n");
        printf("1. Inserisci elemento in coda\n");
        printf("2. Visualizza il prossimo elemento\n");
        printf("3. Servi il prossimo elemento\n");
        printf("0. Termina il programma\n");
        scanf("%d", &scelta);
        scanf("%c", &invio);
    }
```

Coda Heap in linguaggio C

```
switch (scelta) {  
    case 0:  
        printf("Uscita dal programma\n");  
        break;  
    case 1:  
        printf("Inserire il numero dell'elemento: ");  
        el=getElement();  
        n++;  
        vettore[n] = el;  
        MoveUp(vettore,n,n);  
        break;  
    case 2:  
        printf("\nIl prossimo elemento e': %d", vettore[1]);  
        break;
```

Coda Heap in linguaggio C

```
case 3:
    printf("\nIl prossimo elemento e': %d (eliminato)",
           vettore[1]);
    vettore[1] = vettore[n];
    n--;
    heapify(vettore,1,n);
    break;
default:
    printf("la scelta non e' corretta, inserirne un'altra\n");
}
} while(scelta != 0);
heapsort(vettore,n);
stampavettore(vettore, n);
return; }
```


Esercitazione Heap Binaria in linguaggio C

Data le seguente struttura

```
Typedef struct {  
    char Nome [20];  
    char CF [17];  
    char Referto [100];  
    char CodiceAccesso;  
} Persona;
```

Relative agli accessi ad un pronto soccorso di un ospedale, realizzare:

- 1.ADT Heap Binaria sulla chiave CodiceAccesso*
- 2.Acquisizione dati da stdin e simulazione degli accessi al pronto soccorso*
- 3.(Facoltativo) Aggiungere alla chiave della heap un intero che rappresenta l'ora di arrivo*