11 - Coda con priorità

La Coda con priorità è una struttura dati che rappresenta un caso particolare del tipo di dato *insieme*.
A differenza delle code con tecnica FIFO, in questo modello sugli elementi è definita una relazione di ordinamento totale ($\leq$) che determina la priorità.
E' possibile interpretare questa struttura come una coda in cui gli elementi non vengono serviti in base all'ordinamento di arrivo, ma rispettando la loro priorità intrinseca.

Le operazioni principali che definiscono questo tipo di dato sono:

- **CreaPrioricoda**: genera una nuova coda vuota.
- **Inserisci**: aggiunge un nuovo elemento alla struttura.
- **Min**: restituisce l'elemento con la priorità "migliore" (il valore minimo) senza rimuoverlo.
- **CancellaMin**: estrae e rimuove l'elemento con priorità minima dalla struttura.

Specifica sintattica

Tipo di dato:

- **prioricoda**: rappresenta l'insieme delle code con priorità.
- **tipoelem**: rappresenta il tipo degli elementi contenuti nella coda.

Operatori:

- CREAPRIORICODA: $() \rightarrow$ prioricoda Genera una nuova istanza della struttura dati.
- INSERISCI: $(\text{tipoelem}, \text{prioricoda}) \rightarrow$ prioricoda Aggiunge un elemento di tipo tipoelem alla prioricoda esistente, restituendo la struttura aggiornata.
- MIN: $(\text{prioricoda}) \rightarrow$ tipoelem Restituisce l'elemento con priorità minima presente nella prioricoda.
- CANCELLAMIN: $(\text{prioricoda}) \rightarrow$ prioricoda Rimuove l'elemento minimo dalla prioricoda e restituisce la struttura modificata.

Specifica semantica

Il tipo **prioricoda** è definito come un insieme di code con priorità contenenti elementi di tipo **tipoelem**.

- **CREAPRIORICODA()** $= A$
  - **Post-condizione**: L'operazione produce un insieme A vuoto ($A = \emptyset$). Questo inizializza la struttura.
- **INSERISCI**$(x, A) = A'$
  - **Post-condizione**: La nuova struttura A' è il risultato dell'unione tra l'insieme originale A e l'elemento x ($A' = A \cup \{x\}$). L'elemento viene aggiunto all'insieme

rispettando le proprietà insiemistiche.

- **MIN**$(A) = x$
  - **Pre-condizione**: L'insieme A non deve essere vuoto ($A \neq \emptyset$). Non è possibile calcolare il minimo di una coda vuota.
  - **Post-condizione**: L'elemento restituito x appartiene ad A ed è strettamente minore di qualsiasi altro elemento y presente in A ($x < y$ per ogni $y \in A, x \neq y$). Questo operatore identifica l'elemento con la priorità maggiore (valore numerico più basso).
- **CANCELLAMIN**$(A) = A'$
  - **Pre-condizione**: L'insieme A non deve essere vuoto ($A \neq \emptyset$).
  - **Post-condizione**: La nuova struttura A' corrisponde all'insieme A privato dell'elemento minimo x ($A' = A - \{x\}$, dove $x = \text{MIN}(A)$). L'operazione rimuove effettivamente l'elemento identificato come minimo.

Rappresentazione delle strutture sequenziali

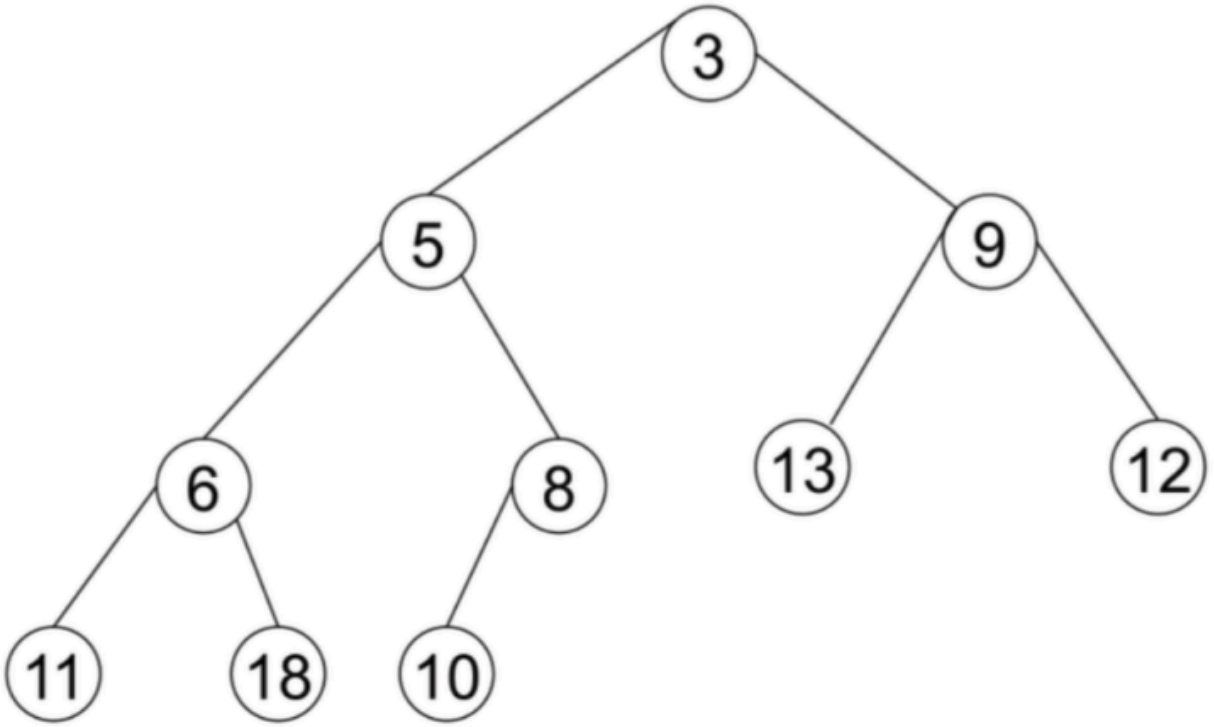
E' possibile rappresentare un coda con priorità contenenti n elementi utilizzando strutture sequenziali, come liste ordinate o liste non ordinate.
La coda con priorità è costituita da un insieme di atomi che risultano linearmente ordinati in base al loro valore di priorità.
Tuttavia, a differenza di altre strutture (come i vettori posizionali), non vi è alcuna relazione strutturale intrinseca sull'insieme delle posizioni fisiche occupate dagli elementi.
Nonostante sia teoricamente possibile l'impiego di liste, la rappresentazione di questo tipo di dato è quasi sempre associato a un unico modello concettuale di riferimento: quello dell'albero binario.

Rappresentazione con alberi binari

Per implementare una coda con priorità C, è possibile memorizzare gli elementi all'interno dei nodi di un albero binario B.
Affinché la struttura sia valida, l'albero deve soddisfare le seguenti proprietà:

1. **Bilanciamento**: L'albero B deve essere "quasi perfettamente bilanciato". Formalmente, se k è il livello massimo delle foglie, l'albero possiede esattamente $2^k - 1$ nodi di livello inferiore a k. Inoltre, tutte le foglie situate al livello k devono essere addossate a sinistra (Proprietà 1).
2. **Ordinamento**: Ogni nodo dell'albero deve contenere un elemento maggiore di quello contenuto nel proprio nodo padre (Proprietà 2). Questa relazione implica che la radice dell'albero conterrà sempre l'elemento minimo dell'intera collezione.

Esempio:
Si consideri un insieme di elementi con priorità definita: $C = \{5, 10, 8, 11, 13, 12, 9, 18, 3, 6\}$.
L'albero binario B mostrato in figura rappresenta correttamente tale coda con priorità, in quanto verifica entrambe le proprietà sopra enunciate



(*PS: ricorda che si parte sempre da 0 per contare i livelli*)

- **Verifica della Proprietà 1 (Bilanciamento)**: Il livello massimo delle foglie è 3. L'albero possiede $2^3 - 1 = 7$ nodi di livello ≤ 2 (ovvero i nodi contenenti 3, 5, 9, 6, 8, 13, 12). Le tre foglie presenti al livello 3 (contenenti 11, 18, 10) sono correttamente posizionate tutte a sinistra.
- **Verifica della Proprietà 2 (Ordinamento)**: È possibile osservare che ogni nodo contiene un valore numerico maggiore di quello del proprio genitore. Ad esempio, la radice contiene 3 (il minimo globale). I suoi figli sono 5 e 9 (entrambi > 3). A sua volta, il nodo 5 ha come figli 6 e 8 (entrambi > 5), e così via per tutto l'albero.

Realizzazione degli operatori

La realizzazione concreta delle operazioni sulla coda con priorità sfrutta le proprietà strutturali dell'albero binario B.

Operatore **MIN**: L'operazione di individuazione del minimo è immediata: essa restituisce il contenuto della radice dell'albero B. Dato che l'albero è ordinato (Proprietà 2), la radice contiene necessariamente l'elemento con priorità maggiore (valore minore).

Operatore **INSERISCI**: L'inserimento di un nuovo elemento avviene manipolando l'albero in modo da preservare le sue caratteristiche:

1. Si inserisce una nuova foglia nella posizione corretta per mantenere verificata la **Proprietà 1** (bilanciamento strutturale)2.
2. Successivamente, si fa "salire" l'elemento introdotto verso la radice, scambiandolo con i genitori se necessario, fino a quando non viene soddisfatta la **Proprietà 2** (ordinamento padre-figlio)3.

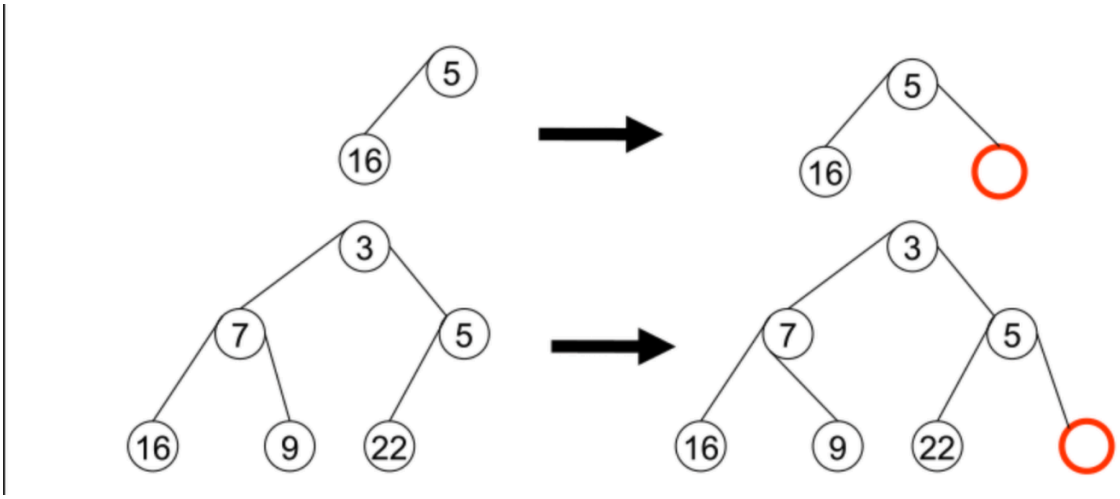
Operatore **CANCELLAMIN**:
La rimozione dell'elemento minimo (la radice) richiede una procedura di riassestamento:

1. Si cancella il nodo foglia situato al livello massimo, più a destra possibile, per garantire che la **Proprietà 1** rimanga valida.
2. Il contenuto della foglia appena cancellata viene reinserito nell'albero partendo dalla radice (sostituendo l'elemento minimo rimosso).
3. Infine, si fa "scendere" questo elemento dalla radice verso il basso, scambiandolo con i figli se necessario, affinché l'albero modificato torni a verificare anche la **Proprietà 2**.

Nello specifico, analizziamo la prima fase dell'operazione **INSERISCI:** la modifica strutturale. L'obiettivo è individuare la posizione esatta in cui aggiungere la nuova foglia affinché l'albero rimanga quasi perfettamente bilanciato (ovvero, riempito da sinistra verso destra).
Se non ci troviamo nei casi banali (albero vuoto o nodo semplice da aggiungere come figlio), l'algoritmo deve calcolare la posizione del "prossimo nodo" basandosi sulla posizione dell'ultima foglia inserita ("ultimo").

Per trovare il punto di inserimento, si esegue una procedura di attraversamento dell'albero composta da due movimenti:

1. **Salita da destra:** Si risale dal nodo "ultimo" verso il padre.
   - *Condizione:* Si continua a salire fintantoché il nodo attuale è un **figlio destro** oppure fino a quando non si raggiunge la radice.
   - *Significato:* Se stiamo risalendo da un figlio destro, significa che quel sotto-albero è già completo; dobbiamo quindi salire ancora per trovare un ramo incompleto.
2. **Discesa verso sinistra:** Una volta terminata la salita (e passati al fratello destro, se non siamo alla radice), si scende sempre verso il **figlio sinistro**.
   - *Condizione:* Si scende fino a incontrare una foglia.
   - *Significato:* Questo garantisce che il nuovo nodo venga posizionato il più a sinistra possibile nel nuovo sotto-albero o livello.



Nell'esempio illustra un caso particolare: il raggiungimento della radice durante la risalita.

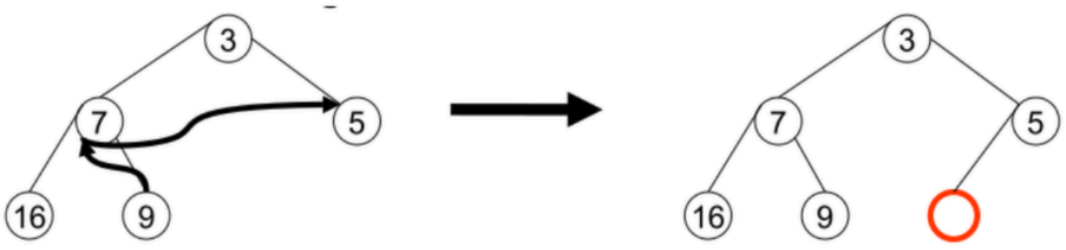
Questo accade quando l'albero (o il livello corrente) è completamente pieno fino all'estrema destra.

Dinamica dell'esempio:

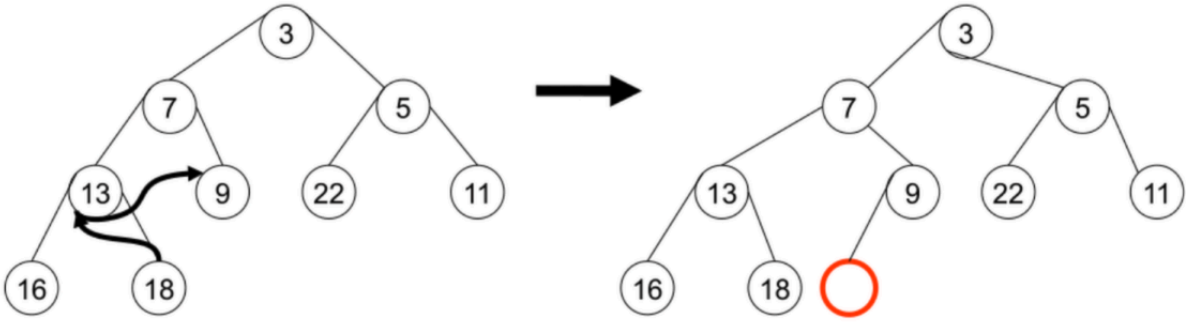
1. **Esempio Superiore (Albero piccolo):**
   - **Stato Iniziale:** L'albero è composto dalla radice (5) e dal suo figlio sinistro (16). Il nodo **16** è l'ultimo elemento inserito (è l'ultima foglia ed è un figlio sinistro).
   - **Azione:** Poiché il nodo padre (5) ha il figlio sinistro occupato ma il destro libero, il nuovo nodo (cerchio rosso) viene inserito direttamente come **figlio destro** di 5.
2. **Esempio Inferiore (Albero più esteso):**
   - **Stato Iniziale:** L'albero ha una struttura più complessa. Osservando l'ultimo livello, notiamo che l'ultima foglia inserita è il nodo **22**.
   - **Analisi Posizionale:** Il nodo **22** è il **figlio sinistro** del nodo 5.
   - **Azione:** Applicando la regola del "Fratello Destro", non è necessario risalire l'albero. Il nuovo elemento (cerchio rosso) viene inserito immediatamente accanto al 22, diventando il **figlio destro** del nodo 5.
     Conclusione: In questi scenari, la **Proprietà 1** (albero quasi perfettamente bilanciato) viene mantenuta semplicemente completando il nodo padre corrente. Non è richiesta alcuna fase di "salita" verso la radice per cercare spazio libero, poiché lo spazio è disponibile immediatamente a destra dell'ultima foglia.

Il secondo Caso, che si verifica quando la fase di salita termina **prima** di raggiungere la radice.
Durante la risalita partendo dall'ultima foglia inserita, ci si ferma perché si incontra un nodo che è **figlio sinistro** del proprio padre.
Poiché siamo saliti da sinistra, il padre ha necessariamente un figlio destro (il fratello del nodo corrente) che rappresenta la radice di un sotto-albero adiacente ancora da riempire (o parzialmente vuoto).

In questo scenario, non è necessario resettare il livello ripartendo dalla radice globale.
L'algoritmo esegue una "svolta locale":
Ci si ferma al padre del nodo da cui siamo saliti (poiché proveniamo da sinistra).
Passa al **fratello destro** dell'ultimo nodo visitato in salita.
Dal fratello destro, si scende iterativamente verso il **figlio sinistro** fino a raggiungere il livello delle foglie.



- **Stato Iniziale:** L'**ultima foglia inserita** è il nodo **9**.
- **Fase di Salita:**
  1. Si parte dal nodo **9**. Essendo un **figlio destro**, l'algoritmo risale al padre (**7**).
  2. Si analizza il nodo **7**. Questo è un **figlio sinistro** (della radice 3).
  3. *Stop:* Poiché siamo arrivati a un figlio sinistro, la risalita si interrompe. Non è necessario andare fino alla radice .
- **Svolta e Inserimento:**
  1. Si passa al **fratello destro** del nodo 7, ovvero il nodo **5**.
  2. Dal nodo 5 si dovrebbe scendere verso sinistra, ma non avendo figli, la posizione è immediatamente disponibile.
  3. Il nuovo nodo (cerchio rosso) viene inserito come **figlio sinistro** di 5.



- **Stato Iniziale:** L'albero è più profondo. L'**ultima foglia inserita** è il nodo **18**.
- **Fase di Salita:**
  1. Si parte da **18**. È un **figlio destro**, quindi si sale al padre (**13**).
  2. Si analizza il nodo **13**. Questo è un **figlio sinistro** (del nodo 7).
  3. *Stop:* La condizione di arresto è soddisfatta (siamo su un figlio sinistro). Questo ci dice che il sotto-albero radicato in 13 è completo, e dobbiamo passare al successivo.
- **Svolta e Inserimento:**
  1. Si salta direttamente al **fratello destro** del nodo 13, che è il nodo **9**.
  2. Si scende verso sinistra partendo da 9. Essendo 9 una foglia, lo spazio è libero.
  3. Il nuovo elemento (cerchio rosso) diventa il **figlio sinistro** di 9.

Conclusione Teorica: In entrambi gli esempi, la logica è quella di "saltare" dal sotto-albero appena completato (a sinistra) al suo vicino immediato (a destra). Questo movimento garantisce che l'albero venga riempito in modo compatto, livello per livello, da sinistra verso destra, senza dover ripercorrere l'intera struttura partendo dalla radice globale.

Algoritmo di inserisci

```
if l'albero è vuoto then
     inserisci l'elemento come radice
else
     if l'albero ha solo la radice (che coincide con “ultimo”) then
```

```
          inserisci l'elemento come figlio sinistro della radice
     else
          if “ultimo” è un figlio sinistro then
               inserisci l'elemento come suo fratello destro
          else
               while il nodo è un figlio destro
                    risali
               if il nodo attuale non è la radice then
                    passa al fratello destro
               scendi verso sinistra fino ad arrivare ad una foglia
               inserisci l'elemento come figlio sinistro
“ultimo” diventa la foglia inserita
while il nodo corrente non è la radice e la sua priorità è minore
di quella del padre
     scambia il contenuto di padre e figlio
```

L'algoritmo di inserimento è progettato per preservare le proprietà della coda con priorità attraverso due fasi sequenziali.
Nella **fase strutturale**, si determina la posizione della nuova foglia per mantenere l'albero bilanciato: nei casi base (albero vuoto o con sola radice), l'inserimento è immediato, mentre se l'albero è popolato, si verifica la posizione del nodo "ultimo".
Se questo è un figlio sinistro, il nuovo nodo viene aggiunto direttamente come fratello destro; altrimenti, si attiva una procedura di navigazione che risale l'albero dai figli destri per poi scendere verso l'estrema sinistra del sottoalbero successivo.
Una volta inserita la foglia e aggiornato il puntatore "ultimo", subentra la **fase di aggiustamento** (o *up-heap*): tramite un ciclo iterativo, si confronta la priorità del nuovo elemento con quella del genitore, effettuando scambi verso l'alto (risalita) finché la corretta relazione d'ordine padre-figlio non viene ripristinata.

CANCELLAMIN

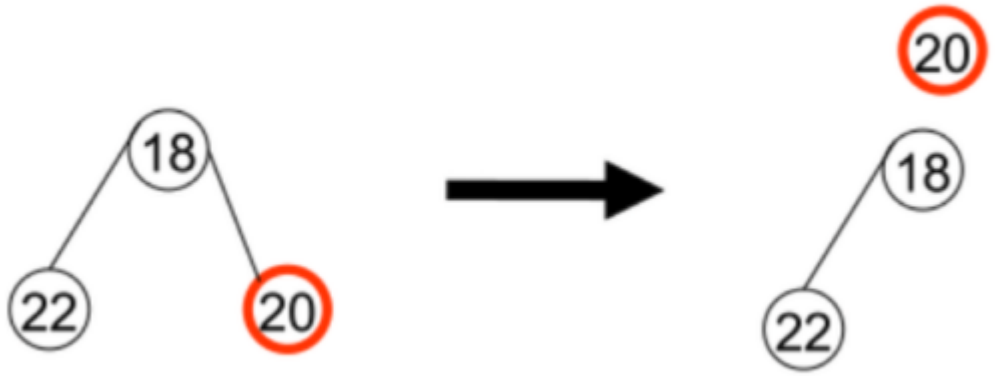
Dopo aver preso l'ultima foglia e averla spostata nella radice per coprire il buco lasciato dal minimo cancellato, abbiamo un problema: il puntatore che indicava la "fine" dell'albero ora punta a un nodo vuoto o sbagliato. Dobbiamo trovare chi è diventato la **nuova "ultima foglia"** (cioè il nodo che, nell'ordine di lettura, veniva subito prima di quello cancellato).

Per trovarlo, l'algoritmo deve "tornare indietro" rispetto all'inserimento, muovendosi in due passaggi:

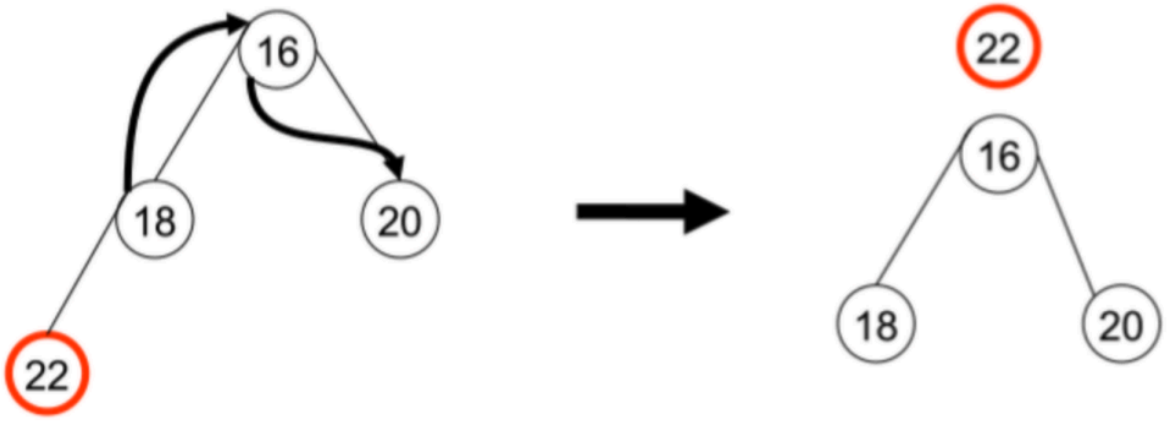
1. **Salita (Backtracking):** Si risale l'albero partendo dalla vecchia posizione.
   - Continuiamo a salire verso il padre finché siamo un **figlio sinistro**.
   - Ci fermiamo appena arriviamo alla radice o se stiamo risalendo provenendo da un **figlio destro**.

2. **Discesa:** Una volta fermata la salita, cambiamo direzione.
    - Scendiamo verso **destra** (verso il fratello o il ramo opposto).
    - Continuiamo a scendere a destra finché non tocchiamo una foglia. Quella sarà la nostra nuova posizione "ultimo".

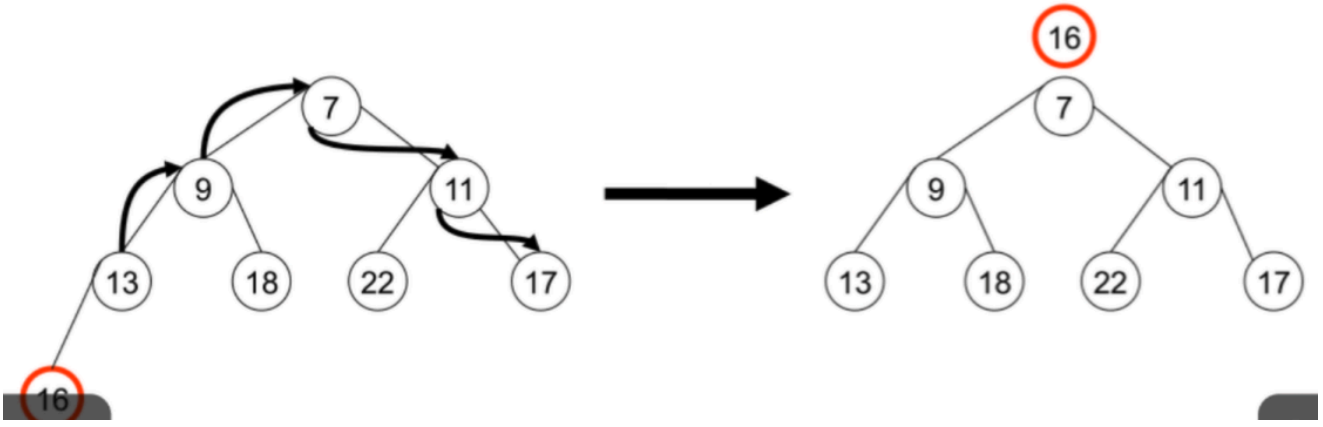
A cosa serve questa procedura?
Serve a trovare il **predecessore logico**. È fondamentale soprattutto nel caso critico in cui abbiamo appena cancellato l'unico nodo rimasto in un livello: grazie a questo movimento (salire fino alla radice e riscendere tutto a destra), il puntatore "torna a capo" alla fine del livello superiore.



Il **primo caso** della ricerca della nuova "ultima foglia", che si verifica quando la fase di risalita raggiunge la radice provenendo esclusivamente da un figlio sinistro.
Questa specifica condizione segnala che il nodo rimosso si trovava all'estremo sinistro della struttura.
Di conseguenza, per individuare il nodo predecessore (il nuovo "ultimo"), l'algoritmo deve ripartire dalla radice e procedere con una discesa verso i figli destri fino al raggiungimento di una foglia.



- **Stato Iniziale:** Il nodo da rimuovere è il **22** (cerchiato in rosso), che è l'attuale "ultima foglia".
- **Fase di Salita:**
  1. Si parte dal nodo **22**. Essendo un **figlio sinistro**, si sale al padre (**18**).
  2. Si continua a salire dal **18** (anch'esso figlio sinistro) fino al padre (**16**).
  3. Il nodo **16** è la **Radice**.
  4. *Condizione:* Siamo arrivati alla radice provenendo esclusivamente da sinistra. Questo indica che il nodo 22 era l'inizio di un livello o l'estremo sinistro assoluto.
- **Fase di Discesa (Ricerca del Predecessore):** Per trovare la *nuova* ultima foglia, dobbiamo andare all'estremo opposto dell'albero (la foglia più a destra possibile).
  1. Si riparte dalla radice (**16**).
  2. Si scende verso il **figlio destro** (**20**).
  3. Essendo 20 una foglia, la ricerca termina. Il nodo **20** diventa la nuova ultima foglia.

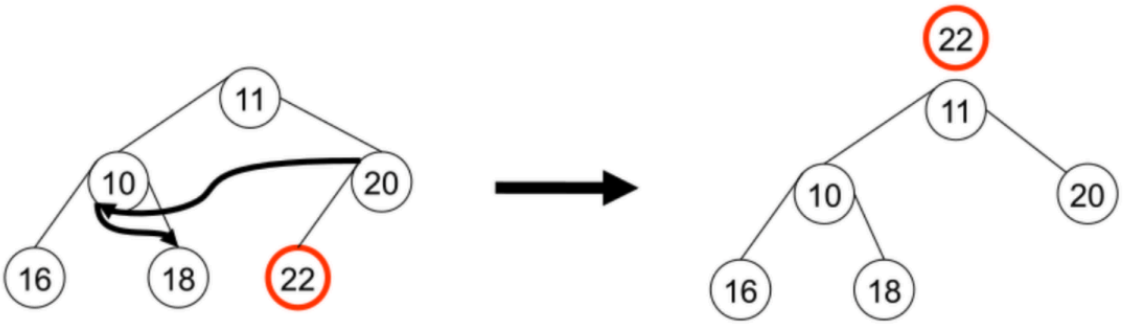


- **Stato Iniziale:** Il nodo da rimuovere è il **16** (cerchiato in rosso), situato all'estrema sinistra profonda.
- **Fase di Salita:**
  1. Si risale la catena dei padri partendo da **16**: $\rightarrow$ **13** $\rightarrow$ **9** $\rightarrow$ **7** (Radice).
  2. Tutti i nodi attraversati erano figli sinistri. Si giunge quindi alla radice.
- Fase di Discesa:
  Bisogna scendere lungo il bordo destro dell'albero per trovare l'elemento che precede logicamente il 16.
  1. Dalla radice (**7**), si scende al figlio destro (**11**).
  2. Dal nodo **11**, si continua a scendere verso il figlio destro (**17**).
  3. Arrivati alla foglia **17**, essa viene designata come nuova "ultima foglia"ù

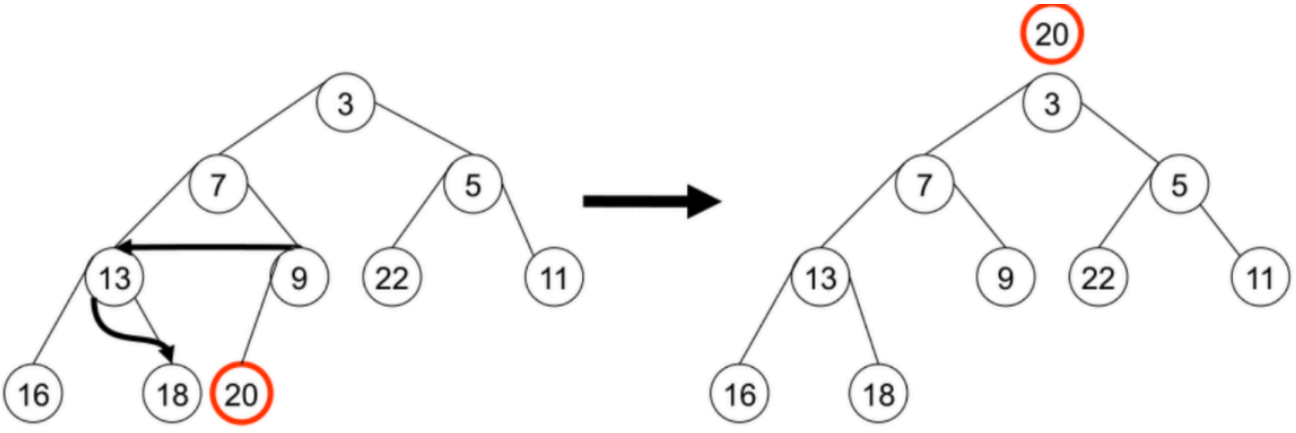
Secondo caso questo scenario rappresenta il caso "standard" o intermedio della cancellazione: il nodo da rimuovere non si trova all'estrema sinistra (non svuota un livello), ma si trova in una posizione interna o destra. L'obiettivo è trovare il nodo che lo precede logicamente nell'ordine di riempimento (ovvero, la foglia più a destra del sotto-albero precedente).

L'algoritmo procede a ritroso rispetto all'inserimento:

1. **Salita:** Si risale dai figli verso i padri.
2. **Stop:** La risalita si interrompe non appena si risale provenendo da un **figlio destro**.
   - Se veniamo da destra, significa che abbiamo appena "lasciato" un blocco di destra. Il nodo predecessore deve trovarsi necessariamente nel blocco di sinistra (il sotto-albero fratello).
3. **Svolta e Discesa:** Si passa al **fratello sinistro** del nodo da cui siamo saliti e si scende verso destra fino all'ultima foglia disponibile.



- **Stato Iniziale:** Il nodo da cancellare è il **22** (cerchiato in rosso).
- **Percorso:**
  1. Si sale da **22** (figlio sinistro) a **20**. Si continua a salire.
  2. Si sale da **20** (figlio destro) a **11** (radice).
  3. *Stop:* Poiché siamo saliti da un **figlio destro** (**20**), ci fermiamo.
  4. **Svolta:** Identifichiamo il fratello sinistro di 20, che è il nodo **10**.
  5. **Discesa:** Dal nodo 10, scendiamo verso la foglia più a destra → nodo **18**.
- **Risultato:** Il nodo **18** diventa la nuova ultima foglia.



- **Stato Iniziale:** Il nodo da cancellare è il **20** (cerchiato in rosso).
- **Percorso:**
  1. Si sale da **20** (figlio sinistro) a **9**. Si continua.
  2. Si sale da **9** (figlio destro) a **7**.

3. *Stop:* Poiché siamo saliti da un **figlio destro** (**9**), la risalita termina.
4. **Svolta:** Ci spostiamo sul fratello sinistro di 9, ovvero il nodo **13** 2.
5. **Discesa:** Dal nodo 13 scendiamo verso il figlio destro → nodo **18**.

- **Risultato:** Il nodo **18** viene impostato come nuova ultima foglia.

Conclusione Teorica:
Questa procedura permette di saltare correttamente dal "principio" di un sotto-albero destro alla "fine" del sotto-albero sinistro adiacente, garantendo che il puntatore "ultimo" retroceda di una sola posizione nell'ordine logico dell'albero completo.

Fase di aggiustamento

Dopo la modifica strutturale, l'albero contiene nella radice il valore che apparteneva all'ultima foglia. Poiché questo valore è stato spostato arbitrariamente, è molto probabile che violi la **Proprietà 2** (ordinamento padre-figlio).
Questa fase, nota come *down-heap*, serve a ripristinare la correttezza delle priorità.

L'algoritmo parte dalla radice e cerca la posizione corretta in cui collocare l'elemento "intruso" (l'ex foglia). Il movimento avviene dall'alto verso il basso:

- Si confronta il nodo attuale con i suoi figli.
- **Regola di Selezione:** Se il contenuto del nodo attuale ha una priorità *minore* (quindi è "peggiore") rispetto ai suoi figli, esso deve scendere. Per mantenere la proprietà di ordinamento, si seleziona tra i figli quello con la **priorità maggiore** (il "migliore" dei due) e lo si sposta nel nodo attuale.

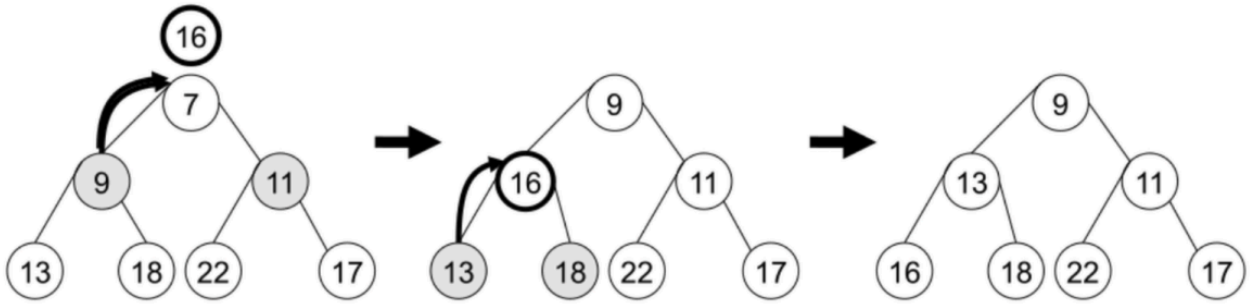
Il processo iterativo di confronto e scambio prosegue fino al verificarsi di una delle seguenti condizioni:

1. **Stabilità trovata:** Il nodo attuale ha una priorità maggiore (migliore) di entrambi i suoi figli. In questo caso, la Proprietà 2 è soddisfatta e l'elemento si ferma.
2. **Fine dell'albero:** Si raggiunge il livello delle foglie, dove non ci sono più figli con cui effettuare confronti.

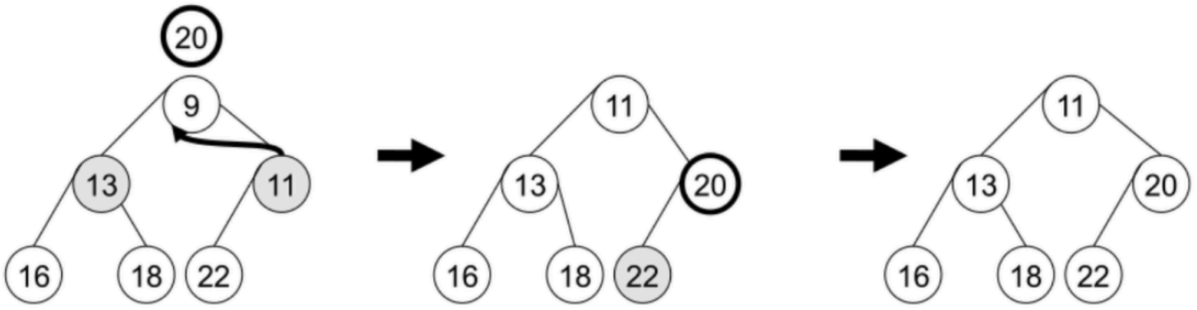
Una conseguenza fondamentale della struttura "quasi perfettamente bilanciata" (Proprietà 1) è che ogni nodo interno ha necessariamente due figli, il che implica che il confronto coinvolge quasi sempre una terna di nodi (padre, figlio sinistro, figlio destro).

- **Eccezione:** L'unico caso in cui il confronto può avvenire con un solo figlio è quando si raggiunge il penultimo livello dell'albero: qui può esistere un nodo che possiede **soltanto**

il figlio sinistro (ma mai solo il destro, per definizione).



- **Obiettivo:** Collocare correttamente il valore **16**.
- **Passo 1:**
  - Si confronta il nodo attuale (contenente 16) con i suoi figli: **9** e **11**.
  - Poiché $16 > 9$ e $16 > 11$, l'ordinamento è violato.
  - **Scelta:** Si seleziona il figlio con priorità maggiore (valore minore), ovvero il **9**.
  - **Azione:** Il 9 sale al posto del padre, e il 16 scende nel ramo sinistro 1.
- **Passo 2:**
  - Il 16 si trova ora a sinistra. Si confronta con i nuovi figli: **13** e **18**.
  - Poiché $16 > 13$, l'ordinamento è ancora violato.
  - **Scelta:** Tra 13 e 18, il minore è **13**.
  - **Azione:** Il 13 sale, il 16 scende ulteriormente.
- **Risultato:** Il 16 raggiunge una posizione di foglia dove non ha più figli con cui confrontarsi. L'albero è ordinato.



- **Obiettivo:** Collocare correttamente il valore **20**.
- **Passo 1:**
  - Si confronta il nodo attuale (contenente 20) con i suoi figli: **13** (sinistro) e **11** (destro).
  - Poiché $20 > 13$ e $20 > 11$, bisogna scendere.
  - **Scelta:** Si seleziona il figlio minore tra i due. In questo caso, $11 < 13$, quindi si sceglie il **figlio destro (11)**.
  - **Azione:** L'11 sale alla radice, e il 20 scende nel ramo destro.

- **Passo 2:**
  - Il 20 si trova ora nel nodo destro. Verifica i suoi figli.
  - In questo esempio, il nodo non ha figli (è una foglia).
- **Risultato:** Il processo termina immediatamente. Il 20 si stabilizza in quella posizione.

Conclusione Teorica:
Questi esempi evidenziano la regola fondamentale della discesa: il padre deve sempre essere minore di entrambi i figli. Pertanto, quando un elemento scende, prende obbligatoriamente il posto del figlio più piccolo dei due, garantendo che il nuovo padre sia minore di entrambi i rami sottostanti.

Algoritmo cancella CANCELLAMIN

```
if l'albero non è vuoto then
     if l'albero ha solo la radice then
          cancella l'intero albero
     else
     copia il contenuto dell'ultima foglia nella radice
     cancellala
/* inizio della fase di modifica della struttura */
     if ultimo è un figlio destro then
          il nuovo ultimo sarà il fratello sinistro
     else
          while il nodo è un figlio sinistro
               risali
          if non si è raggiunta la radice then
               passa al fratello sinistro
     scendi verso destra fino ad arrivare ad una foglia
     impostala come nuovo ultimo
     /*inizio della fase di aggiornamento delle priorità partendo dalla
radice */
while il nodo attuale non è una foglia e la sua priorità è
          minore di quella dei suoi figli do
     if sono presenti entrambi i figli then
          scegli il figlio con priorità maggiore
     else
          seleziona l’unico figlio (che è il sinistro)
     scambia il contenuto del nodo attuale con quello del figlio
selezionato
     spostati sul figlio selezionato
```

L'algoritmo gestisce la rimozione della radice distinguendo i casi banali, dove l'albero viene eliminato se contiene un solo nodo, dalle situazioni complesse che richiedono la sovrascrittura della radice con il contenuto dell'ultima foglia e la sua successiva cancellazione fisica.
La criticità risiede nell'aggiornamento del puntatore "ultimo": se il nodo rimosso era un figlio destro, il puntatore arretra immediatamente al fratello sinistro; se invece era un figlio sinistro, si attiva una procedura di navigazione che risale l'albero dai rami sinistri e, deviando sul sottoalbero adiacente a sinistra, scende lungo il margine destro fino a individuare la nuova foglia terminatrice.
Conclusa la modifica strutturale, si avvia la fase di **aggiornamento delle priorità** partendo dalla radice: tramite un ciclo iterativo, si confronta il nodo corrente con i propri discendenti e, qualora la sua priorità risulti inferiore, si effettua uno scambio con il **figlio a priorità maggiore** (selezionando il migliore tra i due o l'unico figlio sinistro disponibile), procedendo nella discesa finché l'ordinamento padre-figlio non viene ripristinato o si raggiunge una foglia.

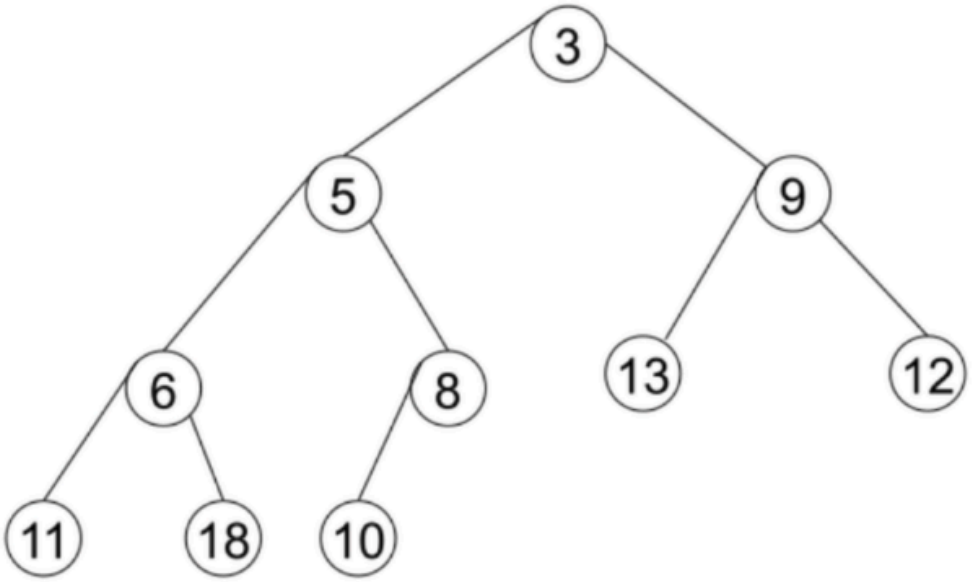
Realizzazione con Heap (Mappatura su Vettore)

La struttura astratta dell'albero binario B viene implementata concretamente mediante un **vettore sequenziale** H (Heap), dove gli elementi sono memorizzati seguendo l'ordine di visita per livelli (dalla radice alle foglie, da sinistra a destra).

In questa rappresentazione implicita, che pone la radice all'indice $H[1]$, le relazioni topologiche sono definite aritmeticamente: dato un nodo all'indice i, il suo **figlio sinistro** si trova alla posizione $2i$ e il **figlio destro** alla posizione $2i + 1$.

L'esistenza fisica di tali figli è determinata dalla dimensione totale n della struttura: un figlio esiste se e solo se il suo indice calcolato è minore o uguale a n. Coerentemente con la **Proprietà 2**, questa mappatura impone che il valore contenuto in $H[i]$ sia sempre minore dei valori contenuti negli indici dei figli esistenti.

Gli elementi dell’albero B si memorizzano nell’heap H cose segue: $H[1] = 3$, $H[2] = 5$, $H[3] = 9$, $H[4] = 6$, $H[5] = 8$,
$H[6] = 13,\ H[7] = 12,\ H[8] = 11,\ H[9] = 18,\ H[10] = 10$



L'esempio visualizza concretamente la linearizzazione dell'albero binario B nel vettore sequenziale H.
La memorizzazione avviene seguendo rigorosamente una visita per livelli: si leggono i nodi partendo dalla radice e scendendo livello per livello, procedendo sempre da sinistra verso destra.

- **Livello 0:** La radice contiene **3**, che occupa la posizione $H[1]$.
- **Livello 1:** I figli immediati sono **5** e **9**, mappati rispettivamente in $H[2]$ e $H[3]$.
- **Livello 2:** I nodi successivi (**6, 8, 13, 12**) occupano le posizioni da $H[4]$ a $H[7]$.
- **Livello 3:** Le ultime foglie (**11, 18, 10**) riempiono le posizioni da $H[8]$ a $H[10]$.
  Questa disposizione permette di verificare aritmeticamente le relazioni di parentela senza l'uso di puntatori: prendendo ad esempio il nodo **5** situato all'indice $i = 2$, il suo figlio sinistro (**6**) si trova correttamente all'indice $2i = 4$, mentre il suo figlio destro (**8**) si trova all'indice $2i + 1 = 5$.