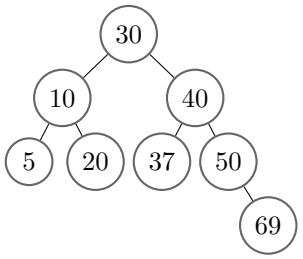
Giorno 6

1 Alberi Binari di Ricerca

Un albero binario di ricerca (*ABR*, o *BST*, *Binary Search Tree*) è un albero binario tale che, per ogni nodo interno v, tutti gli elementi del sottoalbero sinistro di v hanno chiave $< v.key$, e tutti gli elementi del sottoalbero destro hanno chiave $\geq v.key$.



1.1 Ricerca in un ABR

La ricerca di un elemento k di un ABR è, come suggerisce il nome, la ricerca binaria:

```
1  ABR-Search(v, k):
2      if(v == nil): return nil;
3      if(v.key == k): return v;
4      if(k ≥ v.key): return ABR-Search(v.right, k);
5      else: return ABR-Search(v.left, k);
```

Come la ricerca binaria su array, la ricerca binaria ha un tempo di esecuzione $O(h) = O(\log n)$ al caso medio, dove h è l'altezza dell'albero.

1.2 Inserimento in un ABR

```
1  ABR-Insert(r, k):
2      n = new nodo di chiave k;
3      v = r;
4      u = nil;
5      while(v ≠ nil):
6          u = v;
7          if(k ≥ v.key): v = v.right;
8          else: v = v.left;
9      if(k ≥ u.key): u.right = n;
10     else: u.left = n;
```

Anche questa procedura ha costo $O(h)$, poiché al caso peggiore si inserisce in un elemento del livello più basso, il che comporta h iterazioni del while.

1.3 Max, Min, Successore, Predecessore

Il massimo di un albero di radice v è l'elemento più a destra (min duale), costo $O(h)$;

```
1  ABR-Max(r):
2      v = r;
3      while(v.right ≠ nil):
4          v = v.right;
5      return v;
```

```
1  ABR-Min(r):
2      v = r;
3      while(v.left ≠ nil):
4          v = v.left;
5      return v;
```

Il successore può essere il minimo del sottoalbero destro, o in assenza di questo l'ancestor più vicino che contiene v nel suo sottoalbero sinistro. (pred duale)

```
1  ABR-Succ(v, k):
2      if(v.right ≠ nil):
3          return ABR-Min(v.right);
4      else:
5          p = v.p;
6          u = v;
7          while(p ≠ nil && p.left ≠ u):
8              u = p;
9              p = u.p;
10         return p;
```

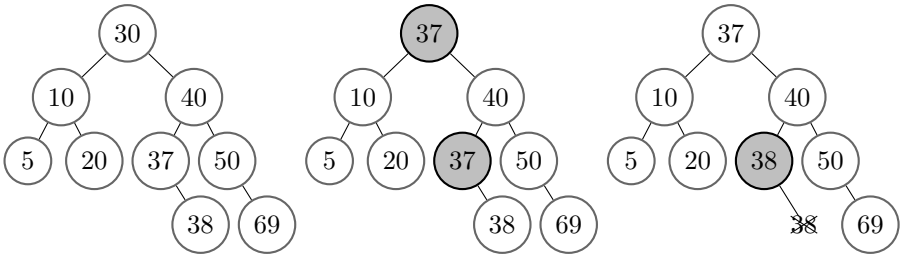
```
1  ABR-Pred(v, k):
2      if(v.left ≠ nil):
3          return ABR-Max(v.left);
4      else:
5          p = v.p;
6          u = v;
7          while(p ≠ nil && p.right ≠ u):
8              u = p;
9              p = u.p;
10         return p;
```

1.4 Eliminazione

Ci sono tre casi:

1. Il nodo v da eliminare non ha figli: in tal caso semplicemente si rimuovono i riferimenti dal parent.
2. v ha un figlio: si sostituisce il figlio a v, correggendo il riferimento al parent del figlio.
3. v ha due figli: si copia il valore (+ dati satelliti) del successore di v in v, si pone il successore di v uguale al suo figlio destro (anche *nil*). Sicuramente il successore ha al più un figlio destro poiché se avesse un figlio sinistro il successore sarebbe lui.

```
1  ABR-Delete(v, k):
2      if(v è una foglia):
3          if(v.p.right == v): v.p.right = nil;
4          else: v.p.left = nil;
5      elif(v ha un figlio dx):
6          if(v.p.right == v): v.p.right = v.right;
7          else: v.p.left = v.right;
8      elif(v ha un figlio sx):
9          if(v.p.right == v): v.p.right = v.left;
10         else: v.p.left = v.left;
11     else:
12         u = ABR-Succ(v);
13         <copio key e dati satelliti di u in v>;
14         u = u.left;
```



Esempio, cancellazione di 30

La cancellazione ha il costo della ricerca del successore, ossia $O(h)$.

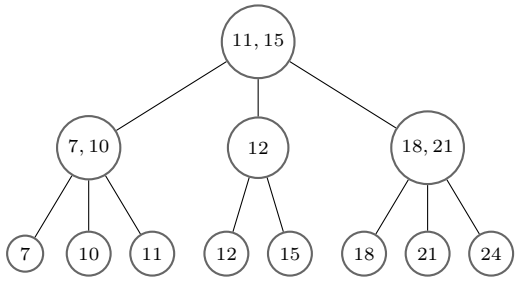
Il problema degli ABR è che spesso tendono a diventare molto sbilanciati; servirebbero degli alberi con le stesse proprietà degli ABR, ma necessariamente bilanciati (alberi 2-3).

2 Alberi 2-3

Gli alberi 2-3 sono alberi tali che **ogni nodo interno abbia 2 o 3 figli, e che tutte le foglie siano sullo stesso livello**. I valori inseriti si trovano **tutti nelle foglie**, ordinati per chiave da sinistra verso destra, mentre ogni nodo interno x ha due attributi:

1. $x.S$ è il **massimo del sottoalbero sinistro**;
2. $x.M$, se il nodo ha tre figli, indica il **massimo del sottoalbero centrale**.

Spesso l'i-esimo figlio (ordinato) del nodo x è indicato con x_i.



L'attributo a sinistra dei nodi con tre figli è S, l'attributo a destra M. M è nullo nel nodo (12).

Tra le principali **proprietà** abbiamo:

1. L'altezza h di un albero 2-3, rispetto al numero di foglie f è tale che: $\log_2 f \leq h \leq \log_3 f$
2. Il numero di nodi n è tale che: $2^{h+1} - 1 \leq n \leq \dfrac{3^{h+1} - 1}{2}$

2.1 Ricerca in un Albero 2-3

In modo simile alla ricerca binaria (e ternaria), si parte dalla radice e si considera, guardando S ed M, in quale sottoalbero si potrebbe trovare la chiave cercata.

```
1  Search(r, k):
2      if(r è una foglia): return r.key == k ? r : nil;
3      if(k ≤ r.S): return Search(r1, k);
4      elif(r ha due figli || k ≤ r.M): return Search(r2, k);
5      else: : return Search(r3, k);
```

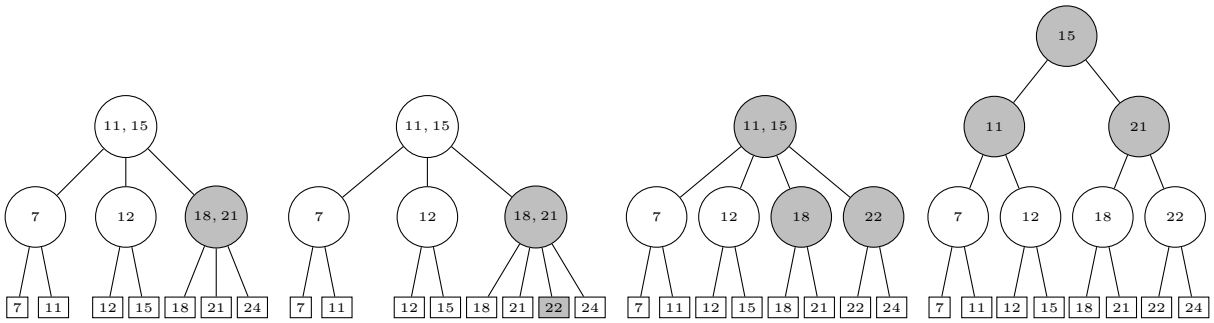
Il costo della ricerca al caso pessimo è $O(h)$.

2.2 Inserimento e cancellazione

2.2.1 Inserimento

Per inserire un elemento x in un albero 2-3, si cerca anzitutto la chiave dell'elemento, per individuare il nodo p che ne sarebbe padre. Poi ci sono due possibilità:

1. p ha due figli: Si inserisce ordinatamente (scambiando poi l'ordine dei figli se necessario) il terzo figlio x.
2. p ha tre figli: Si applica la procedura split:
   (a) Si inserisce ordinatamente x come quarto figlio temporaneo di p.
   (b) Si creano due nuovi nodi n, m da due figli, tali che n ha come figli p_1 e p_2, ed m ha come figli p_3 e p_4 e si pongono come figli del padre di p:
      i. Se il padre di p aveva due figli, adesso ne ha 3 ok.
      ii. Se il padre di p aveva tre figli, adesso ne ha 4, si applica split al padre di p. Se questo era la radice, si crea una nuova radice.



Inserimento di 22

2.2.2 Cancellazione

Per la cancellazione di un elemento x ci sono tre possibilità:

1. x è la radice $\implies$ l'albero diventa vuoto.
2. Il padre di x ha tre figli: ne rimangono due ok;
3. Il padre di x ha due figli: ne rimane 1; si applica la procedura Fuse, duale a Split:
   (a) Se il fratello "più vicino" del padre di x ha 2 figli, lo "fondiamo" col padre di x, creando un unico nodo che ha come figli i tre figli dei due nodi fusi.
   (b) Se il fratello "più vicino" del padre di x ha 3 figli, creiamo due nuovi nodi e dividiamo ordinatamente i figli tra questi due.
      i. Se il padre di $x.p$ aveva tre figli, adesso ne ha due, ok.
      ii. Se il padre di $x.p$ aveva due figli, adesso ne ha uno, si applica Fuse sul padre di $x.p$.