Alcuni limiti sulle mappe

Non è possibile avere coppie diverse con medesima chiave, come per esempio è necessario fare per una coppia (costo, prodotto).
È possibile modificare, magari fare un elenco di coppie in base alla chiave, per ad esempio, fare RANGE QUERY per $k_1 \leq k \leq k_2$.

mappe ordinate:

Gli oggetti sono sempre coppia CHIAVE-VALORE, ma sono definiti secondo ORDINAMENTO TOTALE

ha senso definire quindi le interrogazioni di tipo nearest neighbour
- dato k dare la chiave più vicina di valore $max \leq k$
- dato k dare la chiave minima che è $\geq k$

Definire le operazioni:
- get(k): restituisce la coppia, o le coppie, con chiave k
- put(k,o): inserisce la coppia (k,o)
- remove(k): rimuove la coppia (o le coppie) con chiave k
- subMap(k_1, k_2) restituisce una mappa con tutte le chiavi $k_1 \leq k \leq k_2$

VANTAGGI
- il get ha costo O(log n) per ricerca binaria
- possibilità di fare subMap

SVANTAGGI
- l'aggiunta ha costo O(n)

SubMap:

1) trova la posizione più a sinistra $i_1 = find(k_1)$

2) trova posizione più a destra $i_2 = find(k_2)$
   higherEntry(k_2)

3) restituire porzione mappa con entry $[i_1, \dots, i_2]$

```
higherEntry(k){
  j = find(k)   //restituisce pos ultimo elemento == k o primo > k
  while (j < size()-1 && k == table[j].getKey())
      j+1;
  return j
}
```

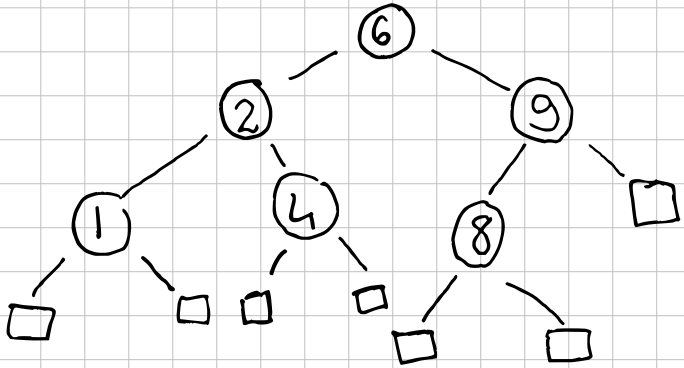
↳ Vi è costo elevato di inserimento e cancellazioni: $O(n)$, trovo un implementazione più "efficiente"

Alberi Binari di Ricerca (BST)

Albero binario i cui nodi contengono coppie CHIAVE - VALORE.
La proprietà delle bst è che, se v e w sono rispettivamente il figlio sinistro e destro di un nodo u, allora

$$key(v) \leq key(u) \leq key(w)$$

N.B. i nodi più esterni hanno associati NULL.



la ricerca di una chiave diventa facile, in quanto si seguono i percorsi secondo la proprietà. traceSearch()

l'aggiunta di una chiave arriverà sempre su un nodo "foglia". Si inserisce il nodo e lo si espande

RIMOZIONI

Cerco se la chiave k esiste sull'albero. Se esiste possiamo basarci su 3 possibilità:

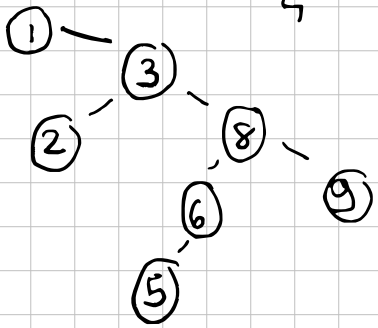
- chiavi esterne
- chiave v ha come figlio una foglia w
- i figli di v sono INTERNI

per chiavi esterne basta eliminare la chiave.

Se la chiave è esterna, è possibile usare successor, SCAMBIANDO le chiavi

- trova il nodo w con chiave immediatamente successiva a k (successor(k))
- copia key(w) e valore in v
- rimuovi w e il figlio sinistro.

es. remove (3)



- successor(3) = succ.
- find(3) : node
  node.key = succ.key
  node.value ..
- succ.remove.

Prestazioni:

le prestazioni set, put e remove hanno complessità $\Theta(h)$, con h l'altezza dell'albero. Se $h \leq n \log$ si parla di AVL

Predecessor e successor

l'idea del predecessor, successor è simmetrica, è quella che data la chiave k, il suo predecessore è nel sottoalbero sinistro, se esso ha chiave massima di k. Se invece la chiave è maggiore del nodo visitato, allora il predecessore può trovarsi nell'albero destro solo se contiene una chiave minore o uguale alla k, altrimenti è proprio il nodo stesso

SubMap

```
Algorithm subMap(v, k1, k2, buffer){
    if (v == null) return;
    if (k1 > v.key)
        subMap(v.rightChild(), k1, k2, buffer);
    else{
        subMap(v.leftChild(), k1, k2, buffer)
        if (k2 >= v.key) {
            buffer.add(v.pair);
            submap(v.rightChild(), k1, k2, buffer);
        }
    }
}
```

La visita che può rendere, in uscita, una visita ordinata, in maniera crescente è la visita SIMMETRICA

Come si è detto, l'efficienza degli algoritmi dipende per di più dall'altezza dell'albero, che nel caso migliore sarà $\log n$, ma nel caso peggiore anche n

↳ Uso AVL

AVL trees

Si usa un fattore di bilanciamento $\beta(v)$ di un nodo v, per capire.

$$\beta(v) = height(v.left) - height(v.right)$$

un albero si dice **BILANCIATO IN ALTEZZA** se ogni nodo v ha fattore di bilanciamento $|\beta(v)| \leq 1$

È in generale tenuto come RECORD dentro le informazioni del nodo v

Proposizione L'altezza di un albero AVL ad n nodi è $\log(n)$

dim. indico con $n(h)$ il numero minimo di nodi di un albero AVL di altezza h.
Si inizia osservando che $n(1) = 1$, poiché un nodo di altezza 1 ha SEMPRE 1 solo nodo, e $n(2) > 2$ poiché un AVL bilanciato deve avere ALMENO 2 nodi.

Ora, un albero AVL con il numero minimo di nodi e di altezza $h \geq 3$ è tale che entrambi i suoi sottoalberi sono AVL con numero minimo di nodi:
- 1 di altezza $h-1$
- 1 di altezza $h-2$
- 1 radice

} → fattore di bilanciamento ≤ 1 (MINIMO 1, MASSIMO 0)

Quindi,

$$n(h) = \begin{cases} h & \text{per } h \leq 2 \\ 1 + n(h-1) + n(h-2) \end{cases}$$

fibonacci!

Inizio a costruire limiti superiori.

$$n(h-1) > n(h-2)$$

$$n(h) > 2\,n(h-2) <$$

$$h(h) > 2(2\,n(h-4))$$

$$\vdots$$

$$h(n) > 2^i\, n(h-2i)$$

$$h - 2i > 2$$

$$\rightarrow i = \frac{h}{2} - 1$$

$$\Downarrow$$

$$h(n) > 2^{\frac{h}{2}-1}$$

Faccio il log di entrambi i membri

$$\log(h(n)) > \frac{h}{2} - 1$$

$$\Rightarrow h < 2\log(h(n)) + 2$$

↳ l'albero AVL che memorizza n voci ha un'altezza minore di $2\log(n) + 2$