Problem Set 4
docente: Luciano Gualà

Esercizio 1 *(Aggiungendo un'operazione a una Pila)*
Progettare una struttura dati che implementa un tipo di dato *Pila* che mantiene una sequenza di elementi con chiave e che, oltre le classiche operazioni di `Top`, `Pop` e `Push`, consente un'operazione aggiuntiva chiamata `Min`. Tale operazione restituisce il puntatore all'elemento di chiave minima contenuto nella pila. Tutte le operazioni devono avere complessità temporale $O(1)$ nel caso peggiore.

Soluzione esercizio 1
La struttura dati è una variazione della classica implementazione del tipo di dato *Pila* tramite una lista. Ogni ogni elemento e della lista conterrà, oltre ai campi relativi alla chiave ($e.key$), all'oggetto ($e.obj$) ed al puntatore all'elemento successivo ($e.next$) anche un puntatore all'oggetto con chiave minore tra quelli che lo seguono, oppure a se stesso ($e.min$). Una rappresentazione della struttura è mostrata in figura 1.

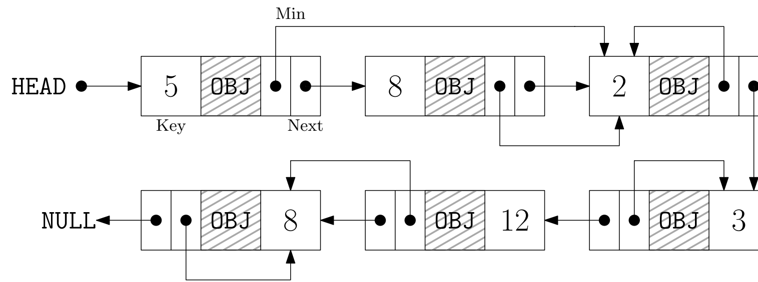

Figura 1: Rappresentazione della struttura dati che implementa il tipo "Pila con Minimo".

Se supponiamo di saper mantenere una simile struttura, l'implementazione della funzione `Min` diventa banale: è sufficiente restituire il puntatore contenuto nel primo elemento della pila. Mostriamo ora come adattare le altre operazioni per tener conto delle modifiche effettuate. Chiamiamo `HEAD` il puntatore al primo elemento della pila.

L'operazione `Push` prende in input un oggetto e la relativa chiave, crea un nuovo elemento e ed inizializza i relativi campi. Al puntatore $e.min$ viene assegnato l'indirizzo dell'oggetto con chiave minore tra quello appena inserito e quello dell'elemento puntato da `HEAD` (se esiste). Infine al puntatore $e.next$ viene assegnato l'indirizzo di `HEAD`, ed in `HEAD` viene memorizzato l'indirizzo di e.

Le operazioni `Pop` e `Top` rimangono sostanzialmente invariate.

Di seguito è riportata una possibile implementazione della struttura in un ipotetico linguaggio di alto livello simile al C.

Algorithm 1: Implementazione del tipo di dato "Pila com Minimo"

```
struct Element
    key Key
    obj Obj
    Element* Min
    Element* Next

Element* HEAD = NULL

procedure Push(key k, obj o)
    Element* e = new Element()
    e → Key = k
    e → Obj = o
    e → Min = e
    e → Next = HEAD
    if HEAD ≠ NULL ∧ HEAD → Min → Key < k then
        e → Min = HEAD → Min
    HEAD = e

procedure Pop()
    if Top ≠ NULL then
        e = HEAD → Next
        delete HEAD
        HEAD = e

funcion Top()
    if Top ≠ NULL then
        return HEAD → Obj
    return NULL

funcion Min()
    if Top ≠ NULL then
        return HEAD → Min → Obj
    return NULL
```

Esercizio 2 *(Un dizionario un po' più ricco)*
Progettare una struttura dati che implementa un tipo di dato *Dizionario* che, oltre le classiche operazioni di `Search`, `Insert` e `Delete`, fornisce anche le seguenti due operazioni aggiuntive:

- `ElementOfRank`(i): dato un intero i, ritorna il puntatore all'elemento di *rango* i, ovvero l'i-esimo minimo contenuto nel dizionario.
- `RankOf`(x): dato il puntatore x a un elemento, ritorna il rango dell'elemento puntato da x, ovvero la posizione dell'elemento nella sequenza ordinata (in ordine crescente di chiave) degli elementi nel dizionario.

Tutte le operazioni devono avere complessità temporale $O(\log n)$ dove n è il numero di elementi presenti nel dizionario.

Suggerimento: si modifichi un albero AVL aggiungendo a ogni nodo v un campo $size(v)$ che contiene il numero di nodi presenti nel sottoalbero radicato in v. Come è possibile usare tale informazione aggiuntiva per implementare le due nuove operazioni? E' possibile mantenere efficientemente questa informazione aggiuntiva?

Soluzione esercizio 2
Come suggerito è possibile aggiungere ad ogni nodo v di un albero AVL un campo $size(v)$ contenente il numero di nodi presenti nel sottoalbero radicato in v.

Dato un nodo v, indicheremo con $v.value$, $v.left$, $v.right$, $v.parent$ rispettivamente, l'elemento associato, il figlio sinistro, il figlio destro ed il padre di v oppure `NULL` nel caso non esistano. Per comodità poniamo $size(\texttt{NULL}) = 0$.

L'operazione `ElementOfRank`(i) può essere implementata in modo simile all'operazione `Search`, partendo dalla radice e scendendo nell'albero fino ad individuare il nodo di rango i. Durante questa operazione si tiene traccia del numero di nodi con chiave inferiore a quella del nodo corrente v che non si trovano nel sottoalbero radicato in v. Chiameremo questa quantità P_v. Inizialmente, quando si considera la radice r, P_r è 0.

Notiamo che il rango di un nodo v è pari a $P_v + size(v.left) + 1$.

Se si sta visitando un nodo v ed il suo rango è superiore ad i si procede nel suo sottoalbero sinistro, radicato in $u = v.left$, che contenente gli elementi precedenti nell'ordinamento. Il nuovo valore P_u è pari a P_v perché sia v che tutti gli elementi nel sottoalbero destro di v avranno chiavi maggiori di quella del nodo u.

Viceversa, se il rango di v è inferiore ad i si procede verso il sottoalbero destro, radicato in $u = v.left$, e vale $P_u = P_v + size(v.left) + 1$ perché sia v che tutti gli elementi nel suo sottoalbero sinistro devono essere più piccoli di v.

Dal momento che, così facendo, il calcolo del rango di un nodo richiede tempo costante, l'intera procedura richiederà un tempo proporzionale all'altezza di albero AVL che è logaritmica nel numero di nodi.

Una possibile implementazione di `RankOf`(x) è del tutto simile alla procedura descritta in precedenza, in cui la scelta del successivo nodo da visitare viene effettuata in base alla relazione d'ordine tra la chiave del nodo attuale e quella del nodo x.

In alternativa è possibile risalire dal nodo x sino alla radice sommando, di volta in volta, il numero di nuovi elementi con chiave inferiore ad x. Essi sono inizialmente $size(x.left)$ ed aumentano di $size(v.left)$ tutte le volte che si risale da un figlio destro al suo genitore v.

Resta ora da far vedere che le dimensioni dei sottoalberi possono essere mantenute in maniera efficiente. Per fare ciò è sufficiente considerare rotazioni semplici ed inserimenti e cancellazioni di foglie.[1]

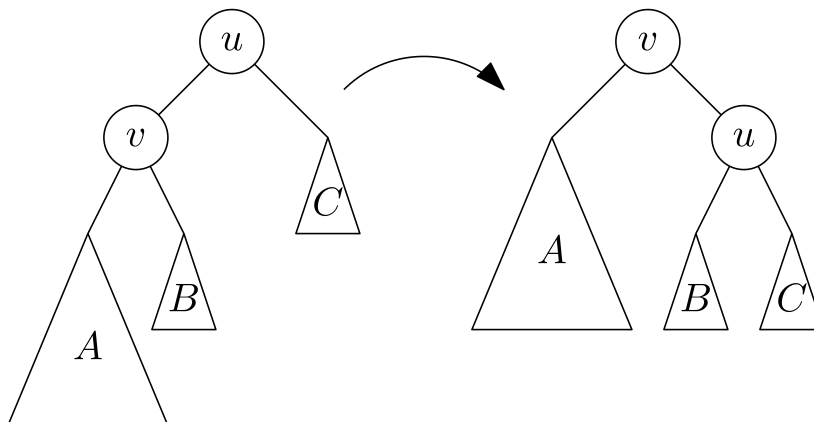

Figura 2: Rotazione destra di un albero AVL.

- Nel caso dell'inserimento di una foglia v è sufficiente porre $size(v) = 1$ quindi procedere dalla nuova foglia inserita verso la radice, aumentando di 1 la dimensione associata ai nodi incontrati. Il tempo richiesto è proporzionale all'altezza.
- Nel caso della cancellazione di una foglia si procede in modo simile, decrementando di 1 la dimensione associata ai nodi tra la foglia eliminata e la radice.
- Per quanto riguarda le rotazioni semplici consideriamo, per simmetria, la sola rotazione destra mostrata in figura 2. Le dimensioni associate a tutti i nodi dei sottoalberi A, B e C rimangono invariate. La dimensione del nuovo sottoalbero di u sarà pari alla somma delle dimensioni dei sottoalberi B e C più 1, cioè a $size(u.left) + size(u.right) + 1$. Infine, $size(v)$ può essere calcolata come $size(v.left) + size(v) + 1$.

[1] Cercare un elemento lascia invariate le dimensioni dei sottoalberi. Per scambiare due elementi (procedura richiesta per la cancellazione) è sufficiente modificare solo chiave e valore, lasciando inalterate le dimensioni dei sottoalberi. Le rotazioni doppie sono applicazioni ripetute di rotazioni semplici.

Algorithm 2: Pseudocodice delle operazioni `ElementOfRank`(i) e `RankOf`(x)

```
r ← radice dell'albero AVL.

funcion ElementOfRank(i)
    return SearchRank (0, i, r)

funcion SearchRank(p, i, v)
    rank ← p + size(v.left) + 1
    if rank = i then
        return v.value
    else if rank < i then
        return SearchRank (p + size(v.left) + 1, i, v.right)
    else
        return SearchRank (p, i, v.right)

funcion RankOf(x)
    rank ← 1 + size(x.left)
    while x.parent ≠ NULL do
        if x.parent.right = x then
            rank ← rank + size(x.parent.left)
        x ← x.parent
    return rank
```

Esercizio 3 *(Ancora sugli oracoli: il problema del* Minimum Range Query*)*
Sia A un vettore di n valori reali. Progettare un algoritmo che, dato A, costruisca un *oracolo* (ovvero una struttura dati) che sia in grado di rispondere in tempo $O(1)$ a *query* (ovvero domande) del seguente tipo: dati due interi i, j, calcolare l'indice dell'elemento di valore minimo nella porzione $A[i; j]$ del vettore.

Si noti che una soluzione semplice al problema è quella di precalcolare tutte le risposte alle $\Theta(n^2)$ query e memorizzarle in una matrice. In questa soluzione, però, l'oracolo (ovvero la matrice delle risposte) ha dimensione $\Theta(n^2)$. Vogliamo, invece, fare meglio in termini di memoria occupata dall'oracolo la cui dimensione richiediamo essere $O(n \log n)$. Non imponiamo invece nessun vincolo sulla complessità temporale necessaria per costruire l'oracolo.

Suggerimento: un'idea potrebbe essere quella di memorizzare solo le risposte a un sottoinsieme delle $\Theta(n^2)$ query. Tale sottoinsieme deve avere dimensione $O(n \log n)$ e deve comunque consentire di rispondere a una generica query in tempo costante.

Soluzione esercizio 3
La struttura è costituita da una collezione di vettori A_k con $0 \le k \le \lfloor \log n \rfloor$. Nell'$i$-esima cella del vettore A_k sarà memorizzato l'indice del minimo tra (al più) 2^k elementi del vettore A, in particolare tra quelli con indice compreso tra i ed $i+2^k-1$. Secondo questa definizione si ha $A_0[i] = i$. Si noti che $n \cdot \sum_{i=0}^{\lfloor \log n \rfloor} 2^i = O(n^2)$ è un upper bound al tempo richiesto per costruire tale struttura e che lo spazio complessivamente occupato dai vettori è $O(n \log n)$, come richiesto.

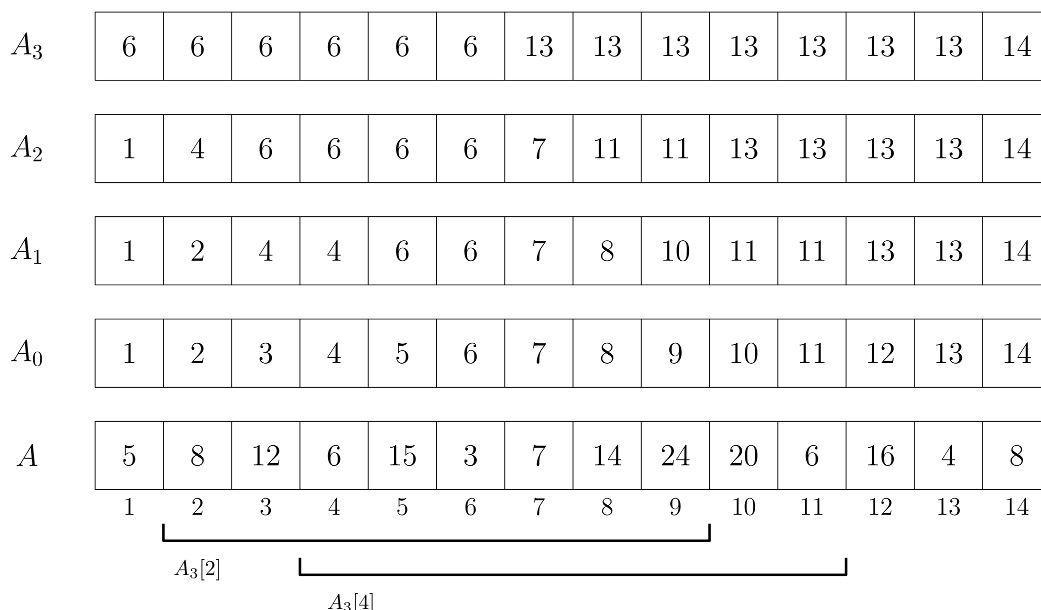

Figura 3: Oracolo per il problema del `Minimum Range Query`.

Dopo aver costruito tutti i vettori A_k è possibile rispondere in tempo costante alle query. Per conoscere il minimo del sottovettore $A[i;j]$ si consida il numero $\ell = j - i + 1$ di elementi di $A[i,j]$, e si procede nel seguente modo:

- Si determina l'indice k del vettore in cui sono stati precalcolati i minimi per gruppi di elementi di dimensione pari ad ℓ, se ℓ non è una potenza di 2 si considera il vettore di indice immediatamente inferiore. Vale: $k = \lfloor \log \ell \rfloor$.
- Si accede opportunamente a 2 elementi del vettore A_k. Ognuno dei due conterrà l'indice dell'elemento minimo in una porzione del vettore A. Gli elementi saranno scelti in modo tale che, complessivamente, vengano considerati tutti (e soli) gli elementi tra $A[i]$ ed $A[j]$. Ciò può essere fatto accedendo a: $A_k[i]$ e $A_k[j - 2^k + 1]$.
- Tra i due indici contenuti in $A_k[i]$ e $A_k[j - 2^k + 1]$ si seleziona quello che corrisponde all'elemento di valore minimo. Dal momento che l'operazione di minimo è associativa, l'indice ottenuto corrisponde proprio al minimo elemento in $A[i;j]$.

Ad esempio nel vettore di figura 3, l'elemento di indice minimo tra $A[2]$ ed $A[11]$ può essere trovato considerando $A_3[2]$ e $A_3[4]$. Il primo elemento contiene l'indice del valore minimo in $A[2;9]$ mentre il secondo contiene quello del valore minimo in $A[4;11]$. Il valore minimo risulta essere $A[6] = 3$.

Si noti che, per la scelta di k, non è mai possibile che i due intervalli considerati non abbiano intersezione: se così non fosse si sarebbe potuto scegliere un valore di k superiore.