BST: Alberi Binari di Ricerca

BST: Alberi Binari di Ricerca	Gli **alberi binari di ricerca** sono strutture dati che supportano molte operazioni sugli insiemi dinamici. Questi alberi non sono altro che alberi binari (con massimo due figli per nodo/sottoalbero) nei quali: - il **figlio sinistro** è minore o uguale al genitore; - il **figlio destro** è maggiore del genitore. Questa proprietà si estende a tutto l'albero, cioè tutti i nodi nel sottoalbero sinistro (dalla radice) dovranno essere minori della radice, e tutti quelli del sottoalbero destro dovranno essere maggiori. La **complessità** delle operazioni dipende dalla **profondità** dell'albero. Gli alberi di ricerca possono essere usati come **dizionari** o **code di priorità**, anche se si tende a preferire l'implementazione della coda con heap, dal momento che le operazioni hanno una complessità computazionale (solitamente) inferiore.
Operazioni Consentite	Le **operazioni consentite** sono: - **search(set, key)** Restituisce l'elemento di un insieme dato che corrisponde alla chiave. - **minimum(set)**, **maximum(set)** Restituisce l'elemento con chiave più piccolo/grande da un insieme **totalmente ordinato**. - **successor(set, element)**, **predecessor(set, element)** Dato un elemento la cui chiave appartiene all'insieme, restituisce il prossimo elemento (più grande) o il precedente (più piccolo). - **insert(set, element)**, **delete(set, element)** Inserimento e rimozione di elementi dall'insieme
Approfondimento: Rimozione di un Elemento	Nella **rimozione** di un elemento da un BST bisogna innanzitutto verificare che il nodo relativo al valore chiave che vogliamo rimuovere si trovi nell'albero (*search*). Se il valore è presente, bisogna considerare 4 casi: 1. **il nodo da rimuovere è foglia**: si rimuove semplicemente il nodo; 2. **il nodo da rimuovere ha solo un sottoalbero sinistro:** il genitore del nodo da rimuovere avrà come successore il sottoalbero (destro o sinistro, in base a dove si trova il nodo in questione) il sottoalbero (sinistro) del nodo da rimuovere; 3. **il nodo da rimuovere ha solo un sottoalbero destro:** il genitore del nodo da rimuovere avrà come successore il sottoalbero (destro o sinistro, in base a dove si trova il nodo in questione) il sottoalbero (destro) del nodo da rimuovere;

	4. **il nodo da rimuovere ha un sottoalbero destro e sinistro:** si sostituisce il nodo da rimuovere con il **nodo maggiore del suo sottoalbero sinistro**, oppure equivalentemente con il **nodo minore del suo sottoalbero destro**. A questo punto, però, si avrà un valore duplicato nell'albero; si richiama allora la funzione di eliminazione sul nodo copiato. Tutto questo si ripete fin quando la funzione non ricade nei primi 3 casi. Risulta quindi utile scrivere una funzione ricorsiva.
Alberi Rosso-Neri	Gli **alberi rosso-neri** nascono per migliorare le prestazioni dei BST. Le operazioni in un comune **BST** hanno **complessità lineare**, che quindi cresce all'aumentare della profondità dell'albero. In un **albero rosso-nero**, invece, le operazioni avranno complessità ***log-lineare*** $O(log\ n)$ nel caso peggiore. In questi alberi, ad ogni nodo viene anche associato un attributo "colore" che può essere rosso o nero. Questi alberi devono rispettare alcune **proprietà** (in aggiunta a quelle dei BST): 1. Ogni nodo ha colore rosso o nero. 2. Il nodo root inizialmente è nero. 3. Ogni foglia è nera e contiene elemento null; 4. Entrambi i figli di ciascun nodo rosso sono neri; 5. Ogni cammino da un nodo a una foglia nel suo sottoalbero contiene lo stesso numero di nodi neri. Questi vincoli rafforzano una proprietà critica degli alberi rosso-neri: che **il cammino più lungo dal nodo root a una foglia è al massimo lungo il doppio del cammino più breve**.
AVL Tree	Un **AVL Tree** è un BST bilanciato in altezza nel quale, per ogni nodo *x*, l'altezza del sottoalbero sinistro e destro di *x* differisce al più di 1. Nell'implementazione di questo albero, i nodi avranno un attributo extra, cioè la sua altezza. Per l'inserimento di nodi si può usare la procedura di inserimento dei BST e poi bilanciare l'albero, oppure usare una procedura ricorsiva che segue la proprietà dell'AVL Tree, impiegando un tempo *log-lineare*.
Treaps	Un **Treap** è un BST che unisce le caratteristiche di un BST a quelle di un Heap. Ogni nodo ha una **chiave unica** ed un valore **priorità** scelto in modo casuale. L'albero viene **ordinato per chiave** secondo la proprietà di un **min heap**.