	ABR	2-3 ALBERO	MAX-HEAP
PROPRIETÀ	ongi nodo ha chiave >= della chiave del figlio sx e <= della chiave del figlio dx. Un ABR può essere sbilanciato.	I dati sono contenuti nelle foglie in ordine crescente, I nodi interni contengono gli indici S e M. S è la chiave più grande contenuta nel sottoalbero sinistro. M è la chiave più grande del sottoalbero centrale ed è valorizzata solo se il nodo ha 3 figli. L'albero2-3 è sempre bilanciato	ogni nodo ha chiave >= della chiave di entrambi I figli. Lo heap è un albero bilanciato dove il valore massimo (o minimo nel caso di min heap) è contenuto nella radice
INSERT(tutte in foglia)	deve solo essere rispettata la proprietà ABR	Fratelli < 3 → inserimento + agg indici Fratelli > 3 → inserimento + split + agg inidici + eventuale aggiunta di un livello	prima foglia libera a sx nell'ultimo liv. Se la prop di heap non è rispettata allora swap ricorsivo con gli antenati (reheapification upward)
DELETE	nodo foglia → cancellazione semplice nodo interno con 1 figlio → swap con il figlio e cancellazione nodo nodo interno con 2 figli → trovare il nodo che sostituisca quello corrente (che verrà eliminato) che sarà il successore (min del sottoalb dx) o il predecessore (max del sottoalb sx). Una volta individuato basta fare lo swap ricorsivo fino ad esso e procedere alla cancellazione	2 fratelli → eliminazione nodo e aggiornamento indici 1 fratello.fratello dx del padre con 3 figli → padre e fratello del padre si prendono entrambi 2 figli, agg inidici.fratello dx del padre con 2 figli → fuse nodo padre con fratello, agg indici	cancellazione nodo radice, inserimento dell'ultimo nodo foglia nella radice. Swap ricorsivo con i nodi discendenti (reheapification downward)

SEARCH	SEARCH (T, k) {
	if (T == null || T.key === k) {
	return T;
	}
	if (T.key < k) {
	return search(T.right, k);
	}
	return search(T.left, k);
	}
		SEARCH (T: INode, k: number) {
		if (isLeaf(T)) {
		if (T.key === k) {
		return T;
		}
		else {
		return null;
		}
		}
		if (k <= T.s) {
		return search(T.v1, k);
		}
		else if (hasTwoChildren(T) || k <= T.m) {
		return search(T.v2, k);
		}
		else {
		return search(T.v3, k)
		}
		}
			SEARCH (S, k, i = 0) {
			if (i >= S.length || S[i] === null) {
			return null;
			}
			if (S[i] === k) {
			return i;
			}
			const left = 2 * i + 1;
			const right = 2 * i + 2;
			return search(S, k, left) || search(S, k, right);
			}

MIN	MIN (T) {
	if (node.left == null) {
	return node.value;
	}
	return min(node.left);
	}
		MIN (T: INode) {
		if (isLeaf(T)) {
		return T.key;
		}
		return min(T.v1);
		}
			MIN (S, i=0) {
			if(!S[i]) {
			return +Infinity;
			}
			const left = 2*i + 1;
			const right = 2*i + 2;
			return Math.min(S[i], min(S, left), min(S, right));
			}

MAX	MAX (T) {
	if (!T.right) {
	return T.value;
	}
	return max(T.right);
	}
		MAX (T: INode) {
		if (isLeaf(T)) {
		return T.key;
		}
		if (exists(T.m)) {
		return max(T.v3)
		}
		else {
		if (exists(T.v2)) {
		return max(T.v2);
		}
		return max(T.v1)
		}
		}
			estrazione nodo radice e eventualmente reheapification downward

MAX-HEAPIFY			Per funzionare sottoalb sx e sottoalb dx devono avere la prop di max-heap. La signature è (S, i) → void ed esegue il processo di reheapification downward su un nodo con un dato indice. Costo O(logn) O(h)

BUIILD-MAX-HEAP			Costruisce un max-heap a partire da un array non ordinato. Chiama la proceduce max-heapify a partire dall'ultimo nodo interno (non radice).
			BUILD-MAX-HEAP(S, n) {
			for (let i = Math.floor(n / 2); i >= 1; i--) {
			MAX-HEAPIFY(S, i);
			}
			}
			
			Costo O(n)

HEAPSORT			prende un array da ordinare, chiama la procedura build-max-heap poi rimuove il nodo radice (chè a questo punto è garantito essere il max) che viene inserito in un array di appoggio (ordinato). Quindi viene inserito l'ultimo nodo foglia al posto della radice e viene ripetuta la procedura finchè l'array non è ordinato. Costo O(n logn)