Betrachten wir nun den Fall, daß k Elemente $b_1 < b_2 < ... < b_k$ in einen 2-3-Baum mit den Elementen $a_1 < a_2 < ... < a_n$ eingefügt werden. Als erstes fügen wir die Elemente b_1 und b_k mittels Algorithmus 2.31 ein. Damit ist sichergestellt, daß alle anderen Elemente zwischen das kleinste und das größte Element des 2-3-Baums eingefügt werden müssen, d.h. $a_1 < b_i < a_n$ mit $2 \leq i \leq k-1$.
Im 2. Schritt suchen wir mittels Algorithmus 2.30 die entsprechenden Einfügepositionen parallel.
Im 3. Schritt wird die Folge $b_2 < ... < b_{k-1}$ in Intervalle $B_1, ..., B_{n-1}$ eingeteilt, so daß für alle $i \in \{1, ..., n-1\}$ und alle $b \in B_i$ gilt $a_i \leq b < a_{i+1}$.
D.h. alle Elemente in einem B_i müssen an die gleiche Position im Ausgangsbaum eingefügt werden.

Dabei unterscheidet man nun 2 Fälle:

1. Für alle $i \in \{1, ..., n-1\}$ gilt $\mid B_i \mid \leq 1$, d.h. für alle Elemente b_j existieren 2 Elemente a_i, a_{i+1} mit $b_{j-1} \leq a_i \leq b_j < a_{i+1} \leq b_{j+1}$ (falls b_{j-1} und b_{j+1} exisiteren)
2. Es gibt B_i mit $\mid B_i \mid > 1$

Betrachten wir nun den Algorithmus für den 1. Fall.
Algorithmus (einfaches Einfügen)
Eingabe: 2-3-Baum A mit $a_1 < a_2 < ... < a_n$ und k Elemente $b_1 < b_2 < ... < b_k$
Ausgabe: 2-3-Baum der alle Elemente a_i und b_j enthält.

1. füge b_1 und b_k mittels Algorithmus 2.31 ein
2. for $2 \leq i \leq k-1$ pardo
   Suche mittels Algorithmus 2.30 für jedes b_i die entsprechende Position in A
3. for $2 \leq i \leq k-1$ pardo
   füge b_i mittels Algorithmus 2.31 in A ein.

Das funktioniert ohne Probleme, da sich die Anzahl der Söhne eines inneren Knoten in A maximal verdoppelt. Diese Knoten müssen dann in maximal 2 neue innere Knoten aufgeteillt werden. Für die Laufzeit ergibt sich folgendes:

1. Schritt $O(\log n)$
2. Schritt $O(\log n)$ und Arbeit $O(k \cdot \log n)$
3. Schritt $O(\log n)$ und Arbeit $O(k \cdot \log n)$

Damit haben wir gesamt $O(\log n)$ parallele Zeit und $O(k \cdot \log n)$ Arbeit.
Betrachten wir nun den 2. Fall, also $\mid B_i \mid > 1$.

Im schlimmsten Fall liegen hier alle Elemente im gleichen Intervall.

Algorithmus (paralleles Einfügen)
Eingabe: wie oben
Ausgabe: wie oben

1. füge b_1 und b_k mittels Algorithmus 2.31 ein
2. for $2 \leq i \leq k-1$ pardo
   Suche die richtige Stelle für b_i mittels Algorithmus 2.30
3. fasse alle b_j in Intervalle B_i zusammen, so daß für alle $b \in B_i$ gilt $a_i \leq b \leq a_{i+1}$
   {dabei kann $\mid B_i \mid > 1$ sein}
4. Wir fügen nun das mittlere Element aller B_i parallel ein und ordnen die B_i neu. Dabei halbiert sich die Anzahl der Elemente in B_i aber die Anzahl der Intervalle verdoppelt sich im schlimmsten Fall. Die Position der neuen Intervalle ist nun bereits durch die Position der Alten fest bestimmt. Wir können unmittelbar mit Schritt 3 Fortfahren bis alle Elemente eingefügt sind.

Betrachten wir nun die benötigte Laufzeit:
1. Schritt benötigt $O(\log n)$ Zeit und Arbeit.
Das Suchen in Schritt 2 $O(\log n)$ parallele Zeit und $O(k \cdot \log n)$ Arbeit.
Schritt 3 und 4 werden solange durchlaufen bis alle Elemente eingefügt sind. Im *worst-case* gibt es nur ein Intervall, also werden alle b_i an der selben Position eingefügt.
Dabei wird Schritt 3 und 4 $O(\log k)$ mal wiederholt. Dies führt zu einer parallelen Zeit von $O(\log k \cdot \log n)$ und Arbeit $O(k \cdot \log n)$.
Um den Zeitbedarf zu reduzieren setzen wir das Pipelining-Prinzip ein. Betrachtet man das Einfügen eines Elementes genau, sieht man, daß sich das Einfügen und Ausbalancieren wie eine Welle, von der Blattebene bis zur Wurzel, durch den Baum bewegt.
Der zweite Einfügedurchlauf kann also schon beginnen, wenn die erste *Welle* die unterste Ebene verlassen hat. Das gleiche gilt für alle $O(\log k)$ Durchläufe.
Damit reduzieren wir den Zeitbedarf auf $O(\log k + \log n)$ ohne die Arbeit zu erhöhen.
Das parallele Einfügen braucht keinen parallelen Zugriff auf die gleiche Speicherzelle. Allerdings liest das gleichzeitige Suchen die Elemente des 2-3-Baums parallel. Deshalb brauchen wir dafür eine CREW-PRAM.