On $O(1)$ concatenation of deques with heap order

Chandrasekhar Boyapati C. Pandu Rangan

Department of Computer Science and Engineering
Indian Institute of Technology, Madras 600036, India
Email: rangan@iitm.ernet.in

March 1995

Abstract

We present a data structure to implement deques with heap order that supports the operations *find minimum, push, pop, inject, eject and concatenate two deques*, each in $O(1)$ time.

1 Introduction

A *deque with heap order* is a deque (double-ended queue) such that each item has a real-valued key and the operation of finding an item of minimum key is allowed as well as the usual deque operations. Queues with heap order have applications in paging and river routing [3] [4] [2].

Gajeuska and Tarjan [3] have described a data structure that supports in $O(1)$ worst case time the usual deque operations *push, pop, inject, eject* and the *find minimum* operation. They also mentioned that Kosaraju's methods [5] extend to support $O(1)$ time concatenation of deques **when the number of deques is fixed.** However, for the case of a variable number of deques, they left the same as an open problem. In [1], Buchsbaum, Rajamani and Tarjan partially answered this question by describing a data structure that supports all the operations mentioned above in $O(1)$ amortized time using the technique of bootstrapping and path compression.

In this paper we first present a simple data structure that also supports all the above mentioned operations in $O(1)$ amortized time even when the number of deques is arbitrarily large, without using the technique of bootstrapping or path compression. Also, this data structure is flexible enough to support operations like given the actual position of a key, *insert a key next to it, or delete it, or change its value* in $O(\log n)$ worst case time, while the data structure in [3] or [1] takes $O(n)$ worst case time for each of these operations. Finally,

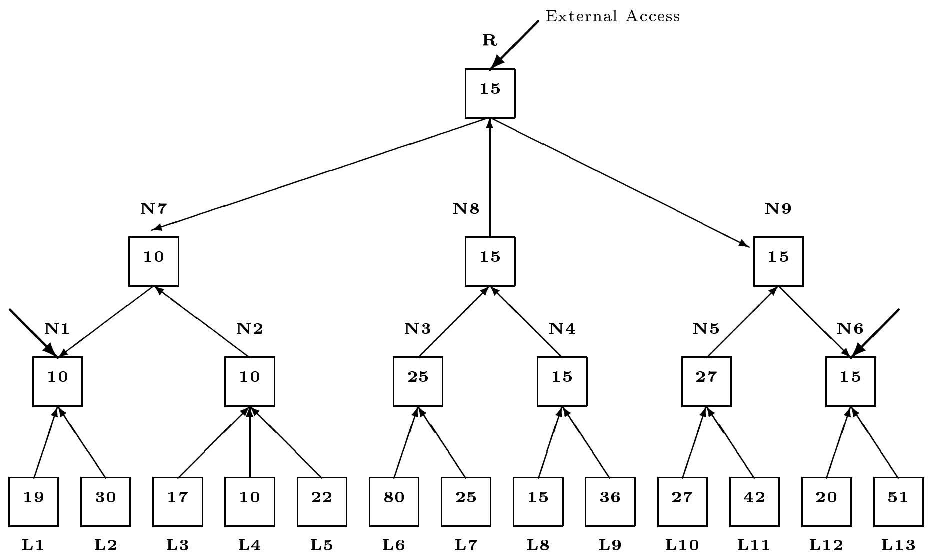

Figure 1

we present an improved version of our data structure that supports *push, pop, inject* and *eject* in $O(1)$ worst case time without changing the complexity of the other operations.

2 The Amortized $O(1)$ Time Data Structure

2.1 Structural Properties

Our data structure shares the following structural properties of 2-4 trees :

1. Every internal node has 2 or 3 or 4 sons. We will say that a node satisfies the *degree property* if it satisfies this condition.

2. All the leaves are at the same level.

2.2 Organisation of the Leaves

Every data item is present in some leaf and every leaf contains one data item. The tree is so arranged that if we go through all the leaves of the tree from left to right, we get the order in which the data items are present in the deque, with the item in the left-most leaf corresponding to the head of the sequence and the item in the right-most leaf corresponding to the tail.

2.3 Organisation of the Internal Nodes

2.3.1 Classification of the Internal Nodes

For convinience of discussion, we will first classify the internal nodes of our tree into three categories :

1. The *root.* (R in figure 1)
2. The *exterior nodes,* which are the internal nodes in the right-most and left-most paths other than the root. (N_1, N_6, N_7 and N_9 in figure 1)
3. The *interior nodes,* which are the rest of the internal nodes. (N_2, N_3, N_4, N_5 and N_8 in figure 1)

2.3.2 Graphical Representation

Consider the graph representation of this tree. We will define a direction[1] for every edge of this graph. We will say that a node M *points* to a node N if there is a directed edge from M to N. We define that —

1. Every leaf *points* to its parent.
2. Every interior node *points* to its parent.
3. The root *points* to its left-most and right-most sons, if they are not leaves.
4. Every exterior node, except the left-most one and the right-most one, *points* to its exterior son.

For example, these directions are explicitly shown in figure 1. The graph thus defined above is obviously a directed acyclic graph.

2.3.3 Contents of the Internal Nodes

For an internal node N, let S_N be the set of nodes that *point* to N. In every internal node N of the tree, we will store (a pointer to) an item as follows :

$$\begin{aligned} item(N) &= \min\{item(M) \mid M \in S_N\},\ if\ S_N\ is\ not\ empty, \\ &= \textbf{infinity},\ if\ S_N\ is\ empty. \end{aligned}$$

Observe that S_N can be empty only when N is the root and has only two exterior sons. We will say that an internal node satisfies the *content property* if it satisfies the above equation.

[1]These directions are just for our conceptual understanding and have nothing to do with the way the nodes are hooked up in the tree.

2.4 External Access

Our tree itself is accessible from outside *via* three pointers, one for the *root*, and one each for the *left-most and right-most exterior nodes.*
For example, the tree in figure 1 is accessed through pointers to N_1, N_6 and R.

The Algorithms and their Analysis

We shall present algorithms for the non-degenerate cases only, that is, when the tree has more then three internal nodes. The degenerate cases can be handled individually, since the number of such cases is finite.

2.5 The Potential Function

Corresponding to each configuration of our data structure, we define a potenial[2] Φ as follows :

$$\begin{aligned} \Phi &= \Theta(\#\ nodes\ with\ 2\ sons + 2 \times \#\ nodes\ with\ 4\ sons) \\ &+ \Theta(height\ of\ the\ tree) \end{aligned}$$

The amortized cost of an operation is defined as the sum of *its actual cost* and *the increase in potential of the data structure due to the operation.*
Since the potential is initially zero and is always positive, the total amortized cost after n operations is an upper bound on the total actual cost.

2.6 Find Minimum

Since the only nodes in the graph with out-degree zero are the left-most and the right-most exterior nodes, at least one of them has to contain the minimum item. Hence, it takes at most one comparision to find the minimum. Thus the time complexity of this operation is $O(1)$, in the worst case.

2.7 Push, Pop, Inject, Eject

From the left-right symmetry in the properties of our tree, we know that both push and inject are identical operations, and so are pop and eject. Hence, we will discuss the operations of insertion and deletion from the right end of the deque only.

[2]Note that our potential function is only a slight modification of the potential function of the 2-4 trees, to accomodate the analysis of the *concatenate* operation.

For convinience of discussion, we will be using the following function :

Algorithm 1 : Update N, where N is an internal node

1. Update the item stored in N such that it satisfies the *content property*, as discussed in Section 2.3.3.

The structural adjustments performed on our tree are exactly identical to those performed on a 2-4 tree following every insertion or deletion. However, with every node split, or node merge, or borrowing of a child by a node from a brother, we will also do a constant time updating of internal nodes as discussed below :

1. When an exterior node N is split into an interior node N_1 and an exterior node N_2, then update N_1, P and N_2, in that order, where P is the parent of N_1 and N_2.
2. When an exterior node is merged with its brother to form a node N, then update P and N, in that order, where P is the parent of N.
3. When an exterior node N_2 borrows a son from a brother N_1, then update N_1 and P, in that order, where P is the parent of N_1 and N_2.

It is easy to see the correctness of the following lemma from the above discussion.

Lemma 1 *If every internal node of the tree satisfies the* content property *before an internal node N splits or merges or borrows a son, then we can restore the* content property *of every internal node by doing a constant time updating of internal nodes (as discussed above) after the split or merge or borrowing, provided that neither N nor its parent is the root.*

We will now present the algorithm for insertion and deletion from the right end.

Algorithm 2 : Insert (or delete) an item at the right end

1. Insert the data item as the right-most leaf L of the tree.
   (Or, delete the right-most leaf L of the tree.)
2. Perform structural adjustments on the tree just as is done in 2-4 trees, accompanied with the updating of internal nodes as discussed above.
3. If no structural adjustments were required in the above step, then let N be the right-most exterior node. Else let N be the parent of the last node that was split, in case of insertion. (In case of deletion, let N be the parent of the last node that was merged, or parent of the last node that borrowed a son. If the last merged node happens to be the root, then let N be the root itself).
   If N is the root then
   (a) Update all the internal nodes in the left-most path, starting from the root till the left-most exterior node.
   (b) Update all the internal nodes in the right-most path, starting from the root till the right-most exterior node.

Time Complexity Analysis

Let there be k node splits or node merges in Step 2. Since the first step takes $O(1)$ time and the rest of the steps take $O(k)$ time each, the actual cost of the operation is $O(k)$.

But, the corresponding decrease in potential is $\Theta(k)$, just as in case of 2-4 trees. Thus, choosing appropiate values for the constants in the potential function, the amortized costs of each of these operations turn out to be $O(1)$.

However, the worst case cost is $O(\log n)$.

2.8 Concatenate two deques

Let us assume, we have to concatenate two deques D_1 and D_2, with the head of D_2 following the tail of D_1 in the final deque. Let T_1 and T_2 be the corresponding trees of heights h_1 and h_2 and with roots R_1 and R_2 respectively, where T_1 is to be attached to the left of T_2. There might be two possibilities :

Case 1 : $h_1 \neq h_2$

Without loss of generality, say $h_1 > h_2$. In this case, concatenation can also be treated as an insertion, not at the leaf level, but at a higher level such that the leaves in the final tree are all at the same level. However, since the nodes in the left-most path of T_2 and some of the nodes in the right-most path of T_1 are converted from exterior nodes to interior nodes, certain nodes in the new tree must be updated. Also, the tree has to be adjusted, to satisfy the structural properties.

Algorithm 3 : Concatenate T_2 to the right of T_1, where $h_1 > h_2$

1. Insert the root R_2 of T_2 as the son of a node M in the right-most path of T_1 such that the leaves in the final tree are all of the same level.
2. Update all the internal nodes in the left-most path of T_2, starting from the left-most exterior node till the root.
3. Update all the internal nodes in the right-most path of T_1 starting from the right-most exterior node till the node M.
4. If M is the root of T_1 then update all the internal nodes in the left-most path of T_1, starting from the root till the left-most exterior node.
5. If M violates the degree property, then execute Algorithm 2 from Step 2 onwards to structurally adjust the new tree.
6. Update all the internal nodes in the right-most path of the new tree starting from the node R_2 till the right-most exterior node.

Case 2: $h_1 = h_2$

In this case, we create a new root R with a dummy item such that $value(R) = \infty$, and make the roots of T_1 and T_2 sons of R. As in the previous case, we also have to update the right-most and left-most paths of T_1 and T_2.

Algorithm 4 : Concatenate T_2 to the right of T_1, where $h_1 = h_2$

1. Create a new root R, with $value(R) = \infty$.
2. Make the roots R_1 and R_2 of T_1 and T_2 sons of R.
3. Update all the internal nodes in the right-most path of T_1, starting from the right-most exterior node till the root R.
4. Update all the internal nodes in the left-most path of T_2, starting from the left-most exterior node till the root R.
5. Update all the internal nodes in the left-most path of T_1, starting from the root R till the left-most exterior node.
6. Update all the internal nodes in the right-most path of T_2, starting from the root R till the right-most exterior node.

Time Complexity Analysis

Let k be the number of node splits, if any, for structurally adjusting the tree. Then, the actual cost of the operation is obviously $O(h_2) + O(k)$.

But, the corresponding decrease in potential is $\Theta(h_2) + \Theta(k)$, since in place of two trees of heights h_1 and h_2, we finally have only one tree of height h_1 (or $h_1 + 1$). Thus the amortized cost of *concatenate* opeartion turns out to be $O(1)$.

However, the worst case cost is $O(\log n)$.

Theorem 1 *Deques with heap order can be implemented to support each of the operations push, pop, inject, eject and concatenate two deques in $O(1)$ amortized case time and the operation find minimum in $O(1)$ worst case time, even when the number of deques is arbitrarily large.*

3 Improved Data Structure to support Push, Pop, Inject and Eject in $O(1)$ Worst Case Time

Since the operations *push, pop, inject,* and *eject* cause insertions or deletions only at the right or left end of the tree, we can improve the efficiency of the data structure by performing the necessary structural adjustments and updating of internal nodes lazily over the next few operations.

To effect this, we will allow the exterior nodes to temporarily have at least one and at most five sons. We will also allow one internal node each in the right-most and left-most paths to violate the *content property.*

Algorithm 5 : Insert (or delete) an item at the right (or left) end

1. Do the insertion (or deletion) at the right (or left) end of the tree.
2. Structurally adjust the lowest node N in the right-most (or left-most) path that violates the *degree property*.
3. // N_L and N_R are the two nodes that may violate the content property.
   // Initially, N_L and N_R are respectively the left-most and right-most leaves.
   If the parent of node N is the root then
   (a) Structurally adjust the other exterior son of the root, if it is violates the *degree property*.
   (b) Structurally adjust the root, if it is violates the *degree property*.
   (c) Set $N_L \leftarrow$ left son of root and $N_R \leftarrow$ right son of root.
4. If N_L and N_R are not leaves then
   (a) Update N_L and N_R
   (b) Set $N_L \leftarrow$ left son of N_L and $N_R \leftarrow$ right son of N_R.

As an illustration of the above algorithm, consider a tree in which the right-most exterior node N_1 and some of its immediate ancestors have four sons each. Let N_{i+1} represent the parent of N_i. Let there be a series of insertions from the right end into this tree. The following table shows the number of sons of some of the right exterior nodes of the tree after each insertion. Note how we do not immediately split all the ancestors but instead 'lazily' split only the lowest node with every insertion.

	N_1	N_2	N_3	N_4	N_5
Initial	4	4	4	4	4
Insert	5	4	4	4	4
Split	3	5	4	4	4
Insert	4	5	4	4	4
Split	4	3	5	4	4
Insert	5	3	5	4	4
Split	3	4	5	4	4
Insert	4	4	5	4	4
Split	4	4	3	5	4

We will demonstrate the correctness of the above algorithm through the following lemmas :

Lemma 2 *Every exterior node will have at least 1 son and at most 5 sons after every operation.*

To prove this, we can think of the above algorithm as follows. After every insertion or deletion at the right end, the right-most exterior node gets a *chance* to adjust itself. If required, it uses the *chance* to adjust itself. Else it passes the *chance* to its parent, which in turn either uses the *chance* or passes it upwards, and so on. To prove the lemma, it suffices to prove that *every node in the right-most path and the left-most path gets at least one* chance *to adjust itself between two consecutive insertion or deletion operations on it.*

Let the above statement be true for an exterior node N_h at a height h in the tree. We will then prove the statement for its parent N_{h+1}.

Observe that an insertion into N_{h+1} corresponds to splitting of N_h and a deletion from N_{h+1} corresponds to merging of N_h with its brother. But after every node split or node merge, N_h will have exactly 3 sons. Hence, before the next node split or node merge of N_h, there have to be at least two insertions or deletions on N_h. Between these two insertions or deletions, N_h will get a *chance* to adjust itself which it does not require since it will have at least 2 sons and at most 4 sons. Hence, it will pass this *chance* to N_{h+1} which can thus adjust itself between every two operations on it. And since the above statement the trivially true for the right-most and left-most exterior nodes, it follows that it is true for all exterior nodes.

Lemma 3 *The lowest exterior node that violates the* degree property *in the right-most or left-most path can be located in constant time by maintaining for each of the two exterior paths, a stack of all the exterior nodes in that path that violate the* degree property*, with the lowest such node at the top of the stack. This ensures that the above algorithm takes $O(1)$ worst case time.*

Lemma 4 *Once the parent of node N in the above algorithm becomes the root, it takes only h operations for N_L and N_R to become leaves while it takes $O(2^h)$ operations for the root to be disturbed again, where h is the height of the tree. This ensures that at any time there are at most two internal nodes (namely N_L and N_R) that violate the* content property*, and hence, find minimum can still be implemented in $O(1)$ worst case time.*

Lemma 5 *Concatenate operation can still be implemented in $O(1)$ amortized time with the following algorithm.*

Algorithm 6 : Concatenate two deques T_1 and T_2

1. Structurally adjust both the trees so that all the internal nodes satisfy the *degree property.* Also, update the exterior nodes in both the trees, starting from N_R or N_L to right-most or left-most exterior nodes.
2. Execute the algorithm discussed in Section 2.8.

Since total work done in m operations by the algorithms for the improved data structure is no more than the total work done by the algorithms for the amortized data structure, the same amortized $O(1)$ bound holds here also.

Theorem 2 *Deques with heap order can be implemented to support each of the operations push, pop, inject, eject and find minimum in $O(1)$ worst case time and the operation concatenate two deques in $O(1)$ amortized time, even when the number of deques is arbitrarily large. Moreover the operations like given the actual position of a key,* insert a key next to it, or delete it, or change its value *can also be implemented in $O(\log n)$ worst case time with the same data structure.*

References

[1] Adam L. Buchsbaum, Rajamani Sundar and Robert E.Tarjan. *"Data structural bootstrapping, linear path compression and concatenable heap ordered double ended queues"*. Proc. **33rd** Ann. Symp. on Foundations of Computer Science, 40-49 (1992)

[2] G. Diehr and B. Faaland, *"Optimal pagination of B-trees with variable length items"*. Comm. ACM **27**, 241-247 (1984)

[3] Hanai Gajeuska and Robert E. Tarjan. *"Deque with heap order"*. IPL **22**, 197-200 (1986)

[4] R. Cole and A. Siegel, *"River routing every which way, but loose"*. Proc. **25th** Ann. IEEE Symp. on Foundations of Computer Science, 65-73 (1984)

[5] S. R. Kosaraju, *"Real-time simulation of concatenatable double-ended queues by double-ended queues.* Proc. **11th** Ann. ACM Symp. on Theory of Computing, 346-351 (1979)