21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2022**

Lecture 5: Binomial Queue

Lecturer: Deshi Ye *Scribe: D. Ye*

5.1 Overview

Recall that the leftist and skew heaps support merging, insertion, and deleteMin all effectively in $O(\log N)$ time per operation (or in amortized). In this lecture, we show that binomial queues support all three operations in $O(logN)$ worst-case time per operation, but insertions take constant time on average.

A binomial queue is not a heap-ordered tree, but rather a collection of heap-ordered trees, known as a forest. Each heap-ordered tree is a binomial tree.

A binomial tree of height 0 is a one-node tree. A binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}. See Fig. 5.1 for an example.

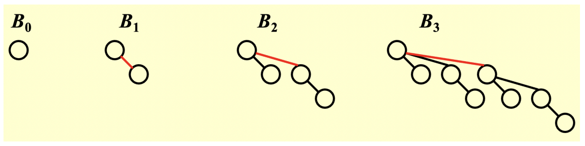


Figure 5.1: The binomial trees B_0 through B_3

Lemma 5.1 *A binomial tree B_k has the following properties:*

1. *The height of B_k is k.*
2. *B_k has exactly 2^k nodes.*
3. *The number of nodes at depth d is the binomial coefficience $\binom{k}{d}$.*
4. *B_k consists of a root with k children, which are $B_0, B_1, \ldots, B_{k-1}$.*
5. *Binomial tree with n nodes has height $O(\log n)$.*

Proof: The proof is by induction on k. For each property, the basis is the binomial tree B_0. It is easy to check each property holds for B_0.

For the inductive step, we assume that the lemma holds for B_{k-1}.

1. Because of the way in which the two copies of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}. By the inductive hypothesis, this maximum depth is $(k-1)+1=k$.

2. Binomial tree B_k consists of two copies of B_{k-1}, and so B_k has $2^{k-1}+2^{k-1}=2^k$ nodes.
3. Let $D(k,i)$ be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in one B_{k-1} appears in B_k is at depth i, while a node at depth i of another B_{k-1} is at depth $i+1$. Thus, the number of nodes at depth i in B_k is the number of nodes at depth i in B_{k-1} plus the number of nodes at depth $i-1$ in B_{k-1}.

$$\begin{aligned} D(k,i) &= D(k-1,i)+D(k-1,i-1) \\ &= \binom{k-1}{i}+\binom{k-1}{i-1} \\ &= \binom{k}{i} \end{aligned}$$

4. The root of B_k has degree one more than B_{k-1}. By the inductive hypothesis, B_k consists of a root with k children. Note that one heap B_{k-1} has children $B_0, B_1, \ldots, B_{k-2}$. When B_{k-1} is linked to B_{k-1}, therefore, the children of the resulting root are roots of $B_0, B_1, \ldots, B_{k-1}$.
5. A binomial tree with n nodes. By Lemma 5.1, we could suppose that $n=2^k$ and it is a binomial tree B_k. Clearly, the height of B_k is k, which is $O(\log n)$.

■

A binomial queue satisfies the B_k structure, heap order, and one binomial tree for each height. A priority queue of any size can be uniquely represented by a collection of binomial trees.

Example: Represent a priority queue of size 13 by a collection of binomial trees. We use the binary resenation of 13, $13 = 2^0 + 02^1 + 2^2 + 2^3 = 1101_2$.

In the following, we discuss the operations of a binomial queue: FindMin, DeleteMin, Insert, Merge, DecreaseKey.

5.2 Operations

5.2.1 FindMin

The minimum key is in one of the roots. There are at most $\lceil \log N \rceil$ roots, hence the FindMin takes at most of $O(\log N)$, where N is the number of nodes in the binomial queue. Note that we can remember the minimum and update whenever it is changed. Then this operation FindMin will take $O(1)$.

5.2.2 Merge

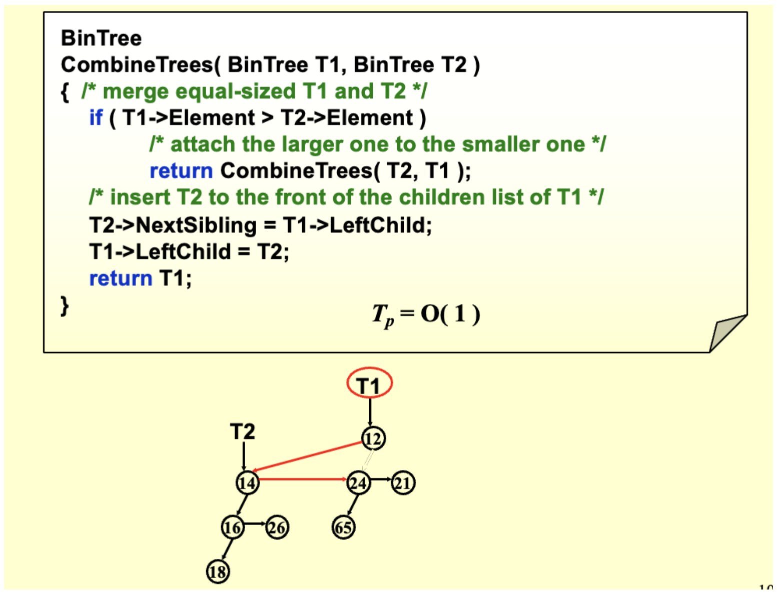
Merge operation is analogous to adding two binary numbers. The binomial trees are ordered in non-decreasing height. The procedure of Merging is kept combining two equal-sized trees starting from the tree of the smallest size. That is, if there have two or three equal-sized binomial trees of B_k, combine two of them into a binomial tree of B_{k+1}, until at most one tree of any height left.

The pseudo-code of Merge is attached here.

```
BinQueue  Merge( BinQueue H1, BinQueue H2 )
{   BinTree T1, T2, Carry = NULL;
```

```
    int i, j;
    if ( H1->CurrentSize + H2-> CurrentSize > Capacity )  ErrorMessage();
    H1->CurrentSize += H2-> CurrentSize;
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
        switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
        case 0: /* 000 */
        case 1: /* 001 */  break;
        case 2: /* 010 */  H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
        case 4: /* 100 */  H1->TheTrees[i] = Carry; Carry = NULL; break;
        case 3: /* 011 */  Carry = CombineTrees( T1, T2 );
                        H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
        case 5: /* 101 */  Carry = CombineTrees( T1, Carry );
                        H1->TheTrees[i] = NULL; break;
        case 6: /* 110 */  Carry = CombineTrees( T2, Carry );
                        H2->TheTrees[i] = NULL; break;
        case 7: /* 111 */  H1->TheTrees[i] = Carry;
                        Carry = CombineTrees( T1, T2 );
                        H2->TheTrees[i] = NULL; break;
        } /* end switch */
    } /* end for-loop */
    return H1;
}
```

Claim: When the process of Merge operation ends, the forest has at most one tree of any height.



```
BinTree
CombineTrees( BinTree T1, BinTree T2 )
{ /* merge equal-sized T1 and T2 */
    if ( T1->Element > T2->Element )
            /* attach the larger one to the smaller one */
            return CombineTrees( T2, T1 );
    /* insert T2 to the front of the children list of T1 */
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}
```

$T_p = O(1)$

Figure 5.2: Combine two equal-sized binomial tree

Merging two equal-sized binomial trees takes constant time (see Fig. 5.2), and there are $O(\log N)$ binomial trees, the merge takes $O(\log N)$ time in the worst case.

5.2.3 Insert

Insert is a special case of Merge operation, hence, it takes $O(\log N)$ time in the worst case.

Theorem 5.2 *Performing N Inserts on an initially empty binomial queue will take $O(N)$ worst-case time. Hence the average time is constant.*

Proof: We suppose that it takes unit time to combine two equal-sized binomial trees. Let Φ_i be the number of trees after the ith insertion ($\Phi_0 = 0$). Let C_i be the actual cost of the ith insertion, and $\hat{C}_i$ the amortized cost, respectively. To insert a new item into a binomial queue, we make the new item into a 1-heap and then apply the Merge operation.

Let c be the number of trees merged during the ith insertion. Then $C_i = 1 + c$, i.e., the number of trees merged plus 1. At the end of ith insert, we know c trees will disappear. The net potential changes $\Delta\Phi = \Phi_i - \Phi_{i-1} = 1 - c$. Thus $\hat{C}_i = C_i + \Phi_i - \Phi_{i-1} = c + 1 + (1 - c) = 2$.

The total amortized running time is

$$2N = \sum_{i=1}^{N} \hat{C}_i = \sum_{i=1}^{N} C_i + \Phi_N - \Phi_0$$

Since $\Phi_N - \Phi_0 \geq 0$, we have $\sum_{i=1}^{N} C_i \leq 2N$, and the average time is $O(1)$.

Another proof by the aggregation method. Let n be the size of the current binomial queue. When we insert a new node in the queue of size n, the cost is i when the binary representation of n with i 1's at the end, but $(i+1)$-th position is 0 (We have exactly $N/2^i$ numbers among all the N numbers). The total cost is $\sum_{i=1}^{N} i\frac{N}{2^i} \leq 2N$. One can check it by the following calculation.

$$y = \sum_{i=1}^{N} \frac{i}{2^i} = 1/2 + 2/2^2 + 3/2^3 + \ldots + N/2^N$$

$$2y = 1 + 2/2 + 3/2^2 + \ldots + N/2^{N-1}$$

We got

$$y = 1 + 1/2 + 1/2^2 + \ldots + 1/2^{N-1} - N/2^N = 2 - 1/2^{N-1} - N/2^N.$$

■

5.2.4 DecreaseKey

The operation DecreaseKey(H, x, k): Given a handle to an element x in H, decrease its key to k. Suppose x is in binomial tree B_k. Repeatedly exchange x with its parent until heap order is restored. The running time is the height of B_k, which is $O(\log N)$, where N is the number of items in H.

5.2.5 Implementation

A binomial queue is an array of binomial trees. Each binomial tree (subtrees) uses the Left-child-next-sibling with linked lists (children are ordered by the size of their subtrees), and the subtrees are stored in the array in decreasing sizes. The Fig. 5.3 shows how the binomial queue H is represented. The root list is a linked list of roots in order of non-increasing degree. Each binomial tree is stored in the left-child, next-sibling representation. The sibling field has a different meaning for roots than for non-roots. If x is a root, then sibling[x] points to the next root in the root list.

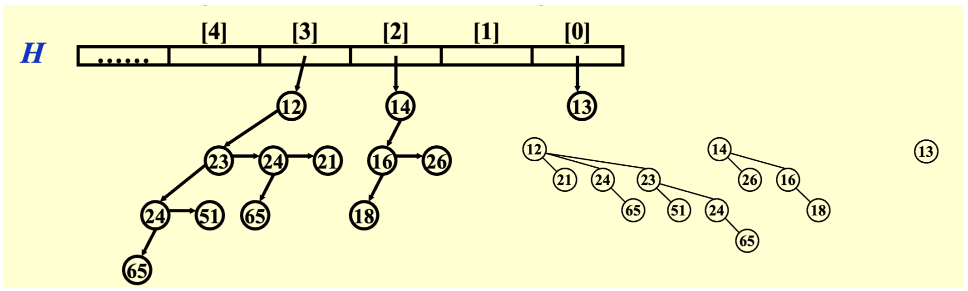


Figure 5.3: Representation of binomial queue H

For further details, we refers to read the textbook [2, 1].

5.3 Dijkstra's Algorithm

We will see how heaps can speedup the Dijkstra's Algorithm for the singe-source shortest path problem.

Theorem 5.3 *(Dijkstra Running Time (Heap-Based)) For every directed graph $G = (V, E)$, every starting vertex s, and every choice of nonnegative edge lengths, Dijkstra's original shortest path algorithm does not use a priority queue, and runs in $O(n^2)$ time. The heap-based implementation of Dijkstra requires $O(1)$ make-heap, $|V|$ Insertions, $|V|$ DeleteMins, and $|E|$ DecreaseKeys. If we used Fibonacci heap to implement it, then the running time is $O(m + n \log n)$, where $m = |E|$ and $n = |V|$.*

Pseudocode of Dijkstra's algorithm (Heap-Based)

```
for every vertex $v$ in $V$ do
    $d_v \leftarrow \infty$; $p_v \leftarrow null$
    Insert($Q, v, d_v$)  /* initialize vertex priority in the priority queue */
$d_s \leftarrow 0$; Decrease($Q, s, d_s$)  /* update priority of $s$ with $d_s$ */
$V_T \leftarrow \emptyset$
for $i \leftarrow 0$ to $|V| - 1$ do
    $u^* \leftarrow DeleteMin(Q)$  /* delete the minimum priority element */
    $V_T \leftarrow V_t \bigcup \{u^*\}$
    for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$;   $p_u \leftarrow u^*$
            Decrease($Q, u, d_u$)
```

Theorem 5.4 *(Prim-MST Running Time (Heap-Based)) For every directed graph $G = (V, E)$, the heap-based implementation of Prim's algorithm requires $O(1)$ make-heap, $|V|$ Insertions, $|V|$ DeleteMins, and $|E|$ DecreaseKeys. If we used Fibonacci heap to implement it, then the running time is $O(m + n \log n)$, where $m = |E|$ and $n = |V|$.*

```
High-level pseudocode:

Algorithm: Prim-MST (G)
Input: Graph G=(V,E) with edge-weights.

1.   Initialize MST to vertex 0.
2.   priority[0] = 0
3.   For all other vertices, set priority[i] = infinity
4.   Initialize prioritySet to all vertices;
5.   while prioritySet.notEmpty()
6.       v = remove minimal-priority vertex from prioritySet;
7.       for each neighbor u of v
9.           w = weight of edge (v, u)
8.           if w < priority[u]
9.               priority[u] = w        // Decrease the priority.
10.          endif
11.      endfor
12.  endwhile

Output: A minimum spanning tree of the graph G.
```

5.4 Heaps

The running time of heap operations are shown in Table 5.1, where the running time of skew heap and the Fibonacci heap is the amortized cost.

Table 5.1: Running time of Heap Operations

	Leftist	**Skew heap**	**Binomial heap**	**Fibonacci heap**	**Link list**
Make heap	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
FindMin	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n)$
Union (Meld)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
DeleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
DecreaseKey			$O(\log n)$	$O(1)$	$O(1)$

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] M. A. Weiss. *Data Structures and Algorithm Analysis in C (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.