CS-GY 6043 Lecture 2
Dynamic arrays with Insert/Delete, Binomial Heaps, Fibonacci Heaps

Sept. 14, 2023

Scribed by Ching Huang

Dynamic Arrays

In the previous lecture and homework, we looked at the amortized analysis of dynamic arrays that support the operations of insert and multipop. In this lecture, we look at the amortized analysis of dynamic arrays that support the operations of insert and delete.
In the following sections, n is the number of elements in the array, s is the size of the array, and α is the load factor, which represents how full the array is. $\alpha = n/s$.

Insert

A dynamic array that only supports the insert operation and doubles in size when the table is full has a potential function of $2n - s$ where n is the number of elements in the array and s is the size of the array. The amortized cost of insert is O(1).

Insert and Delete

A dynamic array that supports both insert and delete has the constraint of keeping its load factor within some constant limits. In the set up of this problem, the array is always at least 1/4 full, so $1/4 \leq \alpha \leq 1$. The array doubles when an insertion happen at $\alpha = 1$, and the array shrinks when a deletion happen at $\alpha = 1/4$.
Doubling and shrinking the table are costly, so the idea is to accumulate a bit of potential from every insertion and deletion. The potential function ϕ used is the following:

$$\phi = \begin{cases} 2n - s, & \text{if } \alpha \geq 1/2 \\ s/2 - n, & \text{if } \alpha < 1/2 \end{cases}$$

This will result in amortized cost of O(1) for each operation.

Binomial Heaps

In this section, we will be focusing on binomial heaps, a data structure that supports the operations of a priority queue. In comparison to regular binary heaps, binomial heaps can perform merging more efficiently.

Priority Queue(PQ)

Each item in a priority queue carries some sort of *data* and a key x that represents its priority. The following describes the operations of a priority queue that acts as a min-heap. (The smaller the x, the higher the priority.)
Operations:

- Insert(x, *data*) : inserts an item with *data* and a key x
- FindMin() : identifies the item with smallest x but doesn't remove it from the data structure
- ExtractMin() : removes and returns the item with the smallest x
- DecreaseKey(*this*, newKey) : decreases the x of the item signified by *this* to newKey
- Delete(*this*) : deletes the item signified by *this*

Additional Operation:

- Merge(PQ1, PQ2) : merges two priority queues

Binomial Trees

Structure: Binomial trees are trees made by linking smaller binomial trees. The simplest binomial tree contains 1 node and is one with order 0. A binomial tree with order k is made by linking two binomial trees of order $k-1$.

Implementation: Binomial trees are usually implemented using the (left child, and right sibling) form. This means that each node only stores its leftmost child. The leftmost child will keep track of all the other children of the parent node, its siblings, in a linked-list form. Each node in a binomial tree also store its degree. To link two binomial trees of the same order together, the root of one tree is chosen as the new root, and the root of the other tree becomes the left child of the new root. The original left child of the new root becomes the right sibling of the new left child.

Properties of binomial tree of order k (B_k):

- size = 2^k
- height = k

- root degree $= k$, degree of its children are $k-1, k-2, k-3, ..., 0$ in order.
- number of items at level l is $\binom{k}{l}$

Proofs of properties by induction:

Size of B_k is 2^k:
Size of $B_0 = 2^0$, since it only contains 1 node.
Size of $B_{k+1} = 2\cdot$ size of $B_k = 2 \cdot 2^k = 2^{k+1}$

Height of B_k is k:
Height of $B_0 = 0$
Height of $B_{k+1} = 1 +$ Height of $B_k = 1 + k$

The root of B_k has k children, and its children has degree $k-1, k-2, k-3, ..., 0$ in order from left to right:
Degree of root of B_0 is 0, and the root has no children.
The root of B_k has degree k. By the way that a binomial tree is constructed, the root of B_{k+1} has degree $k+1$. The left most child of the root of B_{k+1} is the root of B_k, and its other children are the same children it has before linking. By the inductive hypothesis, the children of the root of B_k has degree $k-1$, $k-2$, ..., 0 in order from left to right. Thus the children of B_{k+1} has degree k, $k-1$, $k-2$, ..., 0 from left to right.

Number of items at level l is $\binom{k}{l}$:
Number of items at level $0 = \binom{k}{0} = 1$
Number of items at level j in B_{k+1}
$=$ number of items at level j in B_k + number of items at level $j-1$ in B_k
$= \binom{k}{j} + \binom{k}{j-1}$
$= \binom{k+1}{j}$

Heap-ordered Trees

A heap-ordered tree (min-heap) is a tree where Key(node)$\geq$ Key(parent node) for all nodes in the tree.

Binomial Heaps

Definition:

- A collection of heap-ordered binomial trees.
- Trees are stored in a root list, in order of strictly increasing degrees.
- No 2 trees have the same degree.

Properties of a binomial heap with n nodes:

- Each tree in a binomial tree has a degree < log$n$.
- There are at most log$n$ trees.
- The binary representation of n tells you how many trees there are and which ones. Ex. Binary representation of 13 is 1101. This means that there is a degree 0, an degree 2, and a degree 3 binomial tree in a binomial heap of 13 nodes.

Internal operations to understand how binomial heaps work:

- Link(x, y): Links two heap-ordered binomial trees, x and y together. x and y must have the same degree. This operation is O(1).

  Link(x, y) is performed by
  1. comparing the keys of root of x and root of y, to determine which node should be the root of the new binomial tree
  2. making the root of the other tree the new left child of the chosen root and the original left child of the chosen root the right sibling of the the new left child
- List-Merge(H1, H2): Merges two binomial heaps together into one sorted linked-list of binomial tree roots. This operation is O(log$n$), where n is the total number of nodes after the merge.
- Union(H1, H2): Merges two binomial heaps together into one binomial heap. This is implemented by calling List-Merge(H1, H2) and repeatedly calling Link(x,y) on the result of List-Merge(H1,H2) to merge trees of the same order together. The time complexity of this operation is O(log$n$), where n is the total number of nodes.

How do binomial heaps support operations of a priority queue?

- Insert(x, $data$) : Make a binomial heap containing only one binomial tree of order 0. This binomial heap will contain only one node, which is the item to be inserted. Use Union(H1, H2) to merge this new binomial heap with the original binomial heap. This operation is O(log$n$).
- FindMin() : Scan the list of tree roots for the item with the smallest key. This operation is O(log$n$), where n is the total number of nodes in the heap.
- ExtractMin() : Call FindMin() to find the item with the smallest key in the list of roots. Remove the item and reconnect the list of roots. Since the item is the root of a binomial tree, by removing the item, we are left with a linked-list of binomial tree roots in decreasing order (Property of binomial trees). Sort the roots in the linked-list in increasing order to create a new binomial heap. Call Union(H1, H2) to merge the new binomial heap with the original binomial heap. This operation is O(log$n$).

- DecreaseKey(*this*, newKey) : Bubble-up. Find the item signified by *this*. Compare its key with its parent's key. If its key is smaller, swap the parent and the item. Continue to do this until the item's key is greater than or equal to its parent's or when item becomes the root of a binomial tree. This operation is O(log$n$).

- Delete(*this*) : Find the item signified by *this*. Call DecreaseKey(*this*, newKey) and set the newKey to be negative infinity. Call ExtractMin(). This operation is O(log$n$).

Fibonacci Heaps

In amortized cost analysis, Fibonacci heaps can achieve the operations Extract-Min and DecreaseKey in O(log$n$) time and every other priority queue operation in O(1) time.

Structure: A Fibonacci heap is made up of heap-ordered trees. The tree roots are stored in a circular doubly linked-list in no particular order. A min pointer that points to the item with the minimum key in the heap is stored. Each node stores a variable/bit that signifies whether it is marked.

Potential function for amortized analysis:

ϕ(H) = t(H) + 2m(H)

t(H): size of the tree root list

m(H): number of marked nodes

How do Fibonacci heaps support operations of a priority queue?

- Insert(x, *data*): Create a tree with a single node and add it to the root list. Then update the min pointer.

- Merge(H1, H2): Concatenates 2 circular root lists together and update the min pointer.

- ExtractMin(): uses a function that consolidates trees/nodes...
  More details will be given in the next lecture.