Lecture 2 Advanced Data Structures

CS-GY 6043: Design & Analysis of Algorithms II
Sept. 14th, 2023
Scribed by Kevin Tan

1. Dynamic Arrays Supporting `Delete`

- Let load factor $\alpha = \frac{n}{s}$, where n is the number of elements in the array and s is the size of the array. We want to prevent α from being too small, i.e., having $\alpha \geq c$ where c is a constant.
- We know the amortized cost of insertions is $O(1)$, and $\alpha \geq \frac{1}{2}$ always holds when only `insert` is supported because we double the array when it is full $(\alpha = \frac{1}{2})$.
- When deleting array elements causing $\alpha < \frac{1}{2}$, if we shrink the array immediately, possible subsequent `insert` and `delete` operations will cause performance "thrashing" because of unnecessary costs of expanding and shrinking.
- If we shrink the array when $\alpha < \frac{1}{4}$, then we can have $O(1)$ amortized cost for `delete`. The potential function is

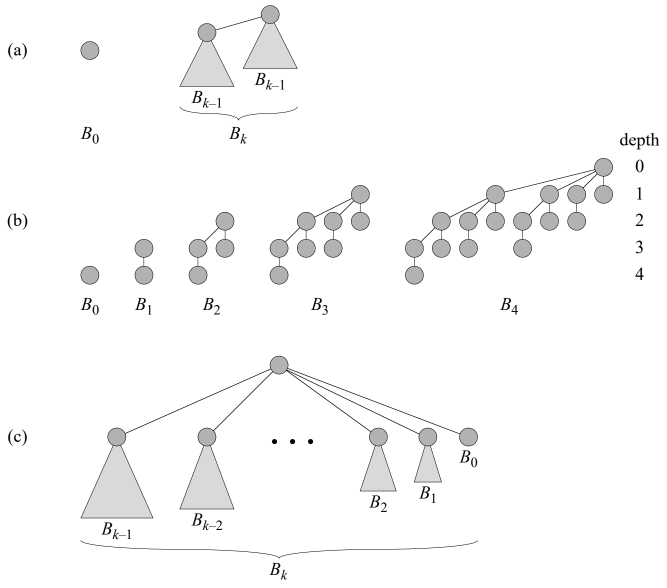
$$\varphi = \begin{cases} 2n - s, & \alpha \geq \frac{1}{2}; \\ \frac{s}{2} - n, & \alpha < \frac{1}{2}. \end{cases}$$

2. Priority Queues (PQs)

- PQ is an Abstract Data Type (ADT) that supports the following operations:
    - `Insert(x, data)`: insert `data` into the PQ with key `x`.
    - `ExtractMin()`: remove and return the item with the smallest key.
    - `FindMin()`: peek and return the item with the smallest key.
    - `DecreaseKey(this, newkey)`: decrease the key of the current item that `this` pointer is pointing at.
    - `Delete(this)`: delete an item that `this` pointer is pointing at.
- A binary implicit heap is usually used to implement PQs, as all operations (except for `FindMin` that has $O(1)$ cost) have $O(\log n)$ cost, where n is the number of elements. Theoretically, typical applications include Prim's Minimal Spanning Tree (MST) algorithm and Dijkstra's shortest path algorithm that require lots of `DecreaseKey`.
- However, if we want to merge two priority queues, the implementation using binary implicit heaps costs $O(n)$. We can achieve $O(\log n)$ using binomial heaps, which will be introduced below.

3. Binomial Trees

- Binomial trees have recursive structures. An illustration from the textbook is shown below, where (a) reflects the recursive definition.



- Let $B_0, B_1, B_2, ...$ be binomial trees, which are ordered trees. For each B_k, it has the following property:
  1. Its size is 2^k.
  2. Its height is k.
  3. The number of nodes on level i is $\binom{k}{i}$.
  4. Its degree of the root is k, with degrees of the children being $k-1, k-2, k-3, \ldots, 0$ from left to right.
- Proof by induction:
  1. $\text{size}(B_0) = 2^0 = 1$, $\text{size}(B_{k+1}) = 2 \times \text{size}(B_k) = 2 \times 2^k = 2^{k+1}$.
  2. $\text{height}(B_0) = 0$, $\text{height}(B_{k+1}) = 1 + \text{height}(B_k) = 1 + k$.
  3. $\binom{0}{i} = 1$, $\binom{k+1}{i} = \binom{k}{i} + \binom{k}{i-1}$.
  4. $\text{degree}(B_0) = 0$, $\text{degree}(B_{k+1}) = 1 + \text{degree}(B_k) = 1 + k$.

4. Binomial Heaps

- A heap-ordered tree is a tree where the key of a node is no less than the key of its parent node.
- A binomial heap is a collection of heap-ordered binomial trees.
  - No two trees have the same degree.
  - Trees are stored in a root list in increasing order of degrees.
- Each bit of the binary representation of the number of nodes n corresponds to the presence (1) or absence (0) of a tree with a degree of the corresponding bit order. For example, a tree of $13 = 1101_2$ nodes includes trees B_3, B_2, B_0 with $2^3 = 8, 2^2 = 4, 2^0 = 1$ nodes, respectively.
- Operations include:

- `Link(x, y)`: It links two heap-ordered binomial trees with the same degree together. The tree with the smaller key becomes the new root, and the other tree becomes its leftmost child. The degree of the root goes up by 1. The cost is $O(1)$.
- `ListMerge(H1, H2)`: It merges two root lists in increasing order of degrees. The cost is $O(\log n)$ because each heap has at most $\lfloor \log n \rfloor + 1$ roots, and they are in the same order.
- `Union(H1, H2)`: It first calls `ListMerge` and then repeatedly calls `Link` until no trees have the same degree. Since the root list remains in increasing order of degrees, it only needs to be traversed once; therefore, the cost is $O(\log n)$.
- `Insert(x, data)`: It first creates a binomial heap with only one binomial tree of degree 0 and then `Union` it into the original binomial heap. The cost is $O(\log n)$.
- `ExtractMin()`: It returns and removes the root with the smallest key. Then, it calls `Union` to merge the children of the removed root into the heap. Attention is needed to the calling sequence of `Link` in `Union` as it may be reversed so the children are `link`ed in increasing order of degrees. The cost is $O(\log n)$ because the children form a binomial heap.
- `DecreaseKey(this, newkey)`: It changes the key of the current item that `this` pointer is pointing at and swaps the item and the parent until the parent has a smaller key or the item becomes the root of a binomial tree (in the last case, the new root should be inserted to the right position to maintain the increasing order of the root list). The cost is $O(\log n)$ because the height of the tree is at most $\log n$.
- `Delete(this)`: It first calls `DecreaseKey` to set the key of the current item that `this` pointer is pointing at to $-\infty$ and then calls `ExtractMin` to remove the item. The cost is $O(\log n)$.

5. Fibonacci Heaps

- A Fibonacci heap is a list of trees (tree roots) where each tree is heap-ordered, but children are not ordered. Children at the same level are circularly doubly linked. A pointer `min` always points at the node with the minimum key. Some nodes are marked.
- Fibonacci heaps involve amortized analysis. The potential function is

$$\varphi(H) = t(H) + 2m(H)$$

  where t is the size of the tree root list, and m is the number of marked nodes. Let $D(n)$ be the max degree of any node in any Fibonacci heap of size that is no more than n. The amortized cost of `ExtractMin` and `DecreaseKey` remains $O(\log n)$, but all other operations cost only $O(1)$. We will later show $D(n) = O(\log n)$.
- Operations include:
  - `Insert(x, data)`: It adds a small tree $\{x\}$ to the root list and updates the `min` pointer. It costs $O(1)$.
  - `Merge(H1, H2)`: It merges two root lists and updates the `min` pointer. It costs $O(1)$.
  - `Consolidate()`: [to be covered next lecture]
  - `ExtractMin()`: [to be covered next lecture]