W4231: Analysis of Algorithms

10/5/1999 (revised 10/6/1999)

- More hashing
- Binomial heaps

Lectures Next Week

No Lecture Tues. October 12 and Thurs. October 14.

Extra lecture ($2\,\frac{1}{2}$ hours) on Mon. October 11 in 1127 Mudd, 11:30am-2:00pm.

Office Hours and Homework

Extra office hours on Fri. October 8, 3-4pm.

No office hours on Thurs. October 14.

Homework due in class next Monday, or to Dario, in his office, next Tues. 3-4pm.

Midterm

Midterm is tentatively scheduled for Tues. Oct. 19, 7-9pm.

An alternate date/time will be set for (CVN and in-class) students with serious conflicts with the scheduled one (three so far).

The syllabus for the midterm includes everything so far, plus Binomial heaps (today) and the notion of amortization (next time). No Fibonacci heaps.

Definition of Universal Hashing

A random hash function

$$h : \{1, \ldots, M\} \to \{1, \ldots, m\}$$

is universal if for every two different inputs x and y,

$$\mathbf{Pr}[h(x) = h(y)] \leq 1/m$$

Analysis of Use of Universal Hashing

Let $y_1, \ldots, y_n$ be an arbitrary set of keys that have been inserted in the table. Let x be an arbitrary key.

Call C the random variable (depending on the choice of h) that counts the number of collisions between $h(x)$ and $h(y_1), \ldots, h(y_n)$. I.e. $C = |\{j : h(y_j) = h(x)\}|$.

Call C_y the random variable (depending on the choice of h) that is 1 if $h(x) = h(y)$, and 0 otherwise.

Then for every y

$$\begin{aligned}\mathbf{E}[C_y] &= 0 \cdot \mathbf{Pr}[h(x) \neq h(y)] + 1 \cdot \mathbf{Pr}[h(x) = h(y)] \\ &= \mathbf{Pr}[h(x) = h(y)] \leq \frac{1}{m}\end{aligned}$$

Since $C = \sum_{i=1}^{n} C_{y_i}$, we have

$\mathbf{E}[C] = \mathbf{E}[\sum_{i=1}^{n} C_{y_i}] = \sum_{i=1}^{n} \mathbf{E}[C_{y_i}] \leq n/m = \alpha$

The running time for `delete` or `find` on x is $O(C)$, so the average running time is $O(\alpha)$.

Interpretation

If we build the table with a random universal hash function,

no matter which sequence of keys we have to insert, and which key we are considering at a given moment,

the average (over h) of the time needed to do a `find` or a `delete` on x is $O(n/m)$, the *load factor*.

The same as having random inputs.

Applications of Hashing

- The Set data structure is useful in several applications. In compilers it is often implemented with hash functions.
- In the *static dictionary* problem, we are interested in building in one shot a data structure for n given elements (with a possibly huge preprocessing time) and then only want to do `find`. Worst-case constant-time implementations are possible using universal hashing as fundamental primitive.

- It is possible to come up with random hash functions such that given a value v it is very hard to come up with an x such that $h(x) = v$ (e.g. MD5 from RSA),

  In order to sign an email, it is sufficient to sign a hash of the email.
- Sometimes *distributed caching* (e.g. for web servers) is done using hashing. A hash function distributes URLs among machines dedicated to caching. Hashing based on sums mod m can be a disaster.

Hashing, Conclusions

Hashing works very well in practice.

Typically it is not needed to pick h at random, and experimental analysis with typical inputs will help find values of m for which the mod hash function is good. Random choices give stronger guarantees of efficiency.

Tolerable downside of hashing: high worst case complexity.

Sometimes intolerable downside of hashing: need to know max number n of elements to be put in the table, so we can choose m. If we over-estimate, collisions become frequent and we may need to reconstruct from scratch a bigger table.

If we under-estimate we waste space. We may need to construct from scratch a smaller table and free the memory of the bigger one.

Even assuming periodical reconstructions, if done in the right way, one can maintain constant *amortized* complexity on average.

Binomial Heaps

Implement the abstract data structure Priority Queue, with the `union` operation.

Operations in the abstract data structure:

- H:= `create`(); Create returns an empty data structure.
- p:= `insert`(x, H); puts element x in H and returns a pointer to it.
- p= `find-min`(H); gives a pointer to the element of H with the minimum key.

- `delete-min`(H); removes from H the elements having smallest key.
- H:= `union`(H_1, H_2); creates a new data structure containing all the elements of H_1 and H_2.
- `decrease-key`(p, k); changes the key of the element pointed by p to k.

Note: There is no operation to find (a pointer to) an element with a given key.

Main result: all operations in $O(\log n)$ worst case time.

Note: AVL trees and red-black trees (that we will not see) can

implement all the operations in $O(\log n)$ time except `union`().

Binomial Tree

A binomial heap is a forest of binomial trees.

We better start with defining binomial trees.

There is a binomial tree for every integer ≥ 0.
Inductive definition:

- A node is the binomial tree B_0.
- Given two binomial trees B_i, join them by making the root of one of them be the first child of the other. This gives B_{i+1}.

Properties of B_k

- It has 2^k nodes.
- It has depth k (depth $=$ number of edges in the longest path from the root to a leaf).
- The root has k children, which is the largest degree in the graph.
- There are $\binom{k}{i}$ nodes at level i in the tree. (Hence the name.)

Representing a binomial tree

Each node is a record containing:

- An element of the data type we want to store.
- Pointers to: father, leftmost child, next sibling in the left-to-right order.

The tree itself can be represented as a record containing a pointer to the root, and an integer field (for a tree B_k the integer field will be k).

We will concentrate on binomial trees that are *heap ordered*:

the key stored in a node is $\geq$ than the keys stored at the children.

Merging two binomial trees of same size

Consider the operation

$T = \texttt{tree} - \texttt{join}(T_1, T_2)$ that given two heap-ordered binomial trees T_1, T_2 (of the same size), returns a heap-ordered binomial tree containing the union of T_1 and T_2.

Can do in $O(1)$ time.

Binomial Heap

A binomial heap H is a forest of binomial trees.

It is represented as a list of the roots.

[More precisely, a list of records, each record containing a pointer to a root, and an integer field telling the size of the tree.]

The list is ordered by increased size of the trees.

Elements in the binomial trees are heap-ordered (the value of the key at a node is larger than the value of the keys at the children).

The binomial trees have all different parameters k.

Structure and the `find-min` Operation

If there are n items in H, no tree can have parameter k bigger than $\lfloor \log n \rfloor$ (since B_k has 2^k nodes).

Since all the trees are different, there are at most $\lfloor \log n \rfloor$ trees.

The minimum is one of the roots; hence the minimum can be found in $O(\log n)$ time by scanning the list of roots.

Union

Join the lists of roots of H_1 and H_2. Similar to the Merge() phase of MergeSort, but when we find a duplicate elements we do a `tree-join()` operation.

At most a `tree-join()` for every element in the lists. Total $O(\log n_1 + \log n_2) = O(\log n)$ time.

Insert

To implement `insert`(x, H)

1. Create a binomial heap H' that contains only x. $O(1)$ time.
2. Let H be Union(H, H'). $O(\log n)$ time.

Total: $O(\log n)$ time.

`decrease-key`(p, k, H)

Reduce the value of the key of the element pointed by p to k.

If x becames smaller than the father, exchange with the father, and proceed recursively to "bubble" x up.

Same as with standard heaps.

Each swap is constant time. At most as many swaps as the depth. $O(\log n)$ time.

`delete-min`(H)

The minimum is always the root of some tree, T.

Remove the tree T from the heap H.

Delete the root from T, and make a new binomial heap H' containing the subtrees.

Do a `union`(H, H').

Total time: $O(\log n)$.

`delete`(p, H)

This operation is not required by the description of the abstract data structure, but is implementable with the operators we have so far.

Do a `decrease-key`(p, k, H), choosing k such that after the decrease the key of the element pointed by p is smaller than the current minimum of H.

Do a `delete`(H).