

Homework #5: Solutions

Problem 1. The problem here is to keep the operations of insert and delete running in logarithmic time under the new key storage scheme. (The new operation runs in constant time, since it is just a matter of adding the increment to the key of the root—well, actually, not the root, but the sentinel in position 0, i.e., the “parent” of the root.)

Delete offers no particular problem: the root is removed; the last element of the heap needs to have the actual value of its key calculated—but that’s easily done by going up from the last element to the root, adding each delta as we go; then it can be sifted down from the top, using the difference between it and the root as first key, then reducing that difference after each comparison by subtracting the delta found in the node with which the previous comparison was made. Since an exchange modifies the parent of both children, the delta value of the untouched child must be updated. Assuming a sentinel in position 0 (the only element with an actual key value) and one in position $n + 1$, we get something like this:

```

/* return root */
item = heap[1].item;
/* calculate delta of last element with respect to root */
lastdelta = heap[n].key;
i = n/2;
while i > 1 {
    lastdelta = lastdelta+heap[i].key;
    i = i/2;
}
/* move end sentinel */
lastitem = heap[n].item
heap[n] = heap[n+1];
/* sift down last element */
i = 1;
while ((2*i <= n-1) &&
    (lastdelta > heap[2*i].key || lastdelta > heap[2*i+1].key)) {
    if (heap[2*i].key < heap[2*i+1].key) {
        mini = 2*i;
        maxi = mini+1;
    }
    else {
        maxi = 2*i;
        mini = maxi+1;
    }
    heap[i] = heap[mini];
    lastdelta = lastdelta+heap[i].key;
    heap[maxi].key = heap[maxi].key - heap[mini].key;
    i = mini;
}
/* found the place: insert item and adjust its key */
heap[i].item = lastitem;
heap[i].key = lastdelta+heap[i].key;
/* reflect size change */
n = n-1;

```

Insert is very similar. The new element has an actual key, whereas its putative parent and ancestors all have only a delta, except for the root's sentinel. So we begin by computing the actual key value of the putative parent, by ascending all the way to the root and summing the deltas, then use this value and the key of the new element to calculate the initial delta (often negative) for the new element and begin the sift-up. The sift-up terminates when the delta of the new element becomes positive—at each level we increment its delta by that of the node over which the sift-up proceeds. Again, each exchange must update the key of the untouched child. We get something like this:

```

i = (n+1)/2;
parentkey = heap[i].key;
while i > 0 {
    i = i/2;
    parentkey = parentkey+heap[i].key;
}
/* initial delta is difference to putative parent */
delta = newkey-parentkey;
/* move sentinel */
n = n+1;
heap[n+1] = heap[n];
/* sift up new item */
i = n;
while delta < 0 {
    heap[i] = heap[i/2];
    delta = delta+heap[i].key;
    if odd(i)
        heap[i-1].key = heap[i-1].key - heap[i].key;
    else
        heap[i+1].key = heap[i-1].key - heap[i].key;
    i = i/2;
}
heap[i] = newitem;

```

Insert and Deletemin still run in logarithmic time—although now each operation will always take time proportional to the height of the heap, no matter what the circumstances.

Problem 2.

Consider the following variation on double hashing for keys that have a total ordering. When attempting to insert an item, x , into the table, we begin by hashing x with our first hash function h_1 ; if location $h_1(x)$ is empty, we insert x there. If $h_1(x)$ contains some key y , however, instead of automatically bouncing to the next location in the probe sequence for x (which would be location $h_1(x) + h_2(x)$ as used in double hashing), we first compare keys x and y ; if x is the larger key, we bounce x , compute $h_2(x)$, and probe location $h_1(x) + h_2(x)$; on the other hand, if x is the smaller key, we dislodge y , i.e., we insert x in location $h_1(x)$ and send y bouncing along its probe sequence, i.e, we next examine location $h_1(x) + h_2(y)$. At every collision, we repeat the procedure, sending the larger key bouncing.

Let the size of the table be n , the number of items in the table m , and the loading factor $\alpha = m/n$.

- Prove that every insertion must eventually terminate.

An item can only bounce over smaller items; otherwise it must come to rest, either in an empty slot or by dislodging a larger item. Hence each item has a fixed number of other items over which it could bounce, so only a finite number of bounces will occur before all items have come to rest.

- How many bounces *could* it take before an insertion terminates?

In a table with m items, the new item could bounce m times and stop in an empty slot; it could also bounce fewer times and dislodge some item, which then starts on its own bounce sequence. Of course, as we proceed from dislodging to dislodging, keys must increase, since an item can only dislodge one with a larger key. Thus the worst case is to dislodge every item in turn, starting with the smallest item in the table; this requires that the item inserted be smaller than any currently within the table. In the worst case, the probe path of each successive item takes it over all items smaller than it is before coming to the next item to be dislodged. The i th item (in decreasing order of keys) can bounce over $(i - 1)$ items and so we could have as many as $\sum_{i=1}^m i$ or $\Theta(m^2)$ bounces.

- What is the average number of bounces before insertion terminates? (To derive this, assume a model similar to the one we used for perfect hashing; rather than computing $h_1(x) + h_2(y)$, we compute the next $h_i(y)$, which is independent of the previous hash functions h_1, h_2 , etc. This is of course an approximation of the real scheme, but double hashing of any kind is too hard to analyze exactly.)

The key is to realize that our new scheme always creates exactly the same table for the same collection of items (with each item in the same slot), regardless of the order in which the items are inserted; next we note that, if the items are inserted in increasing order of keys, then none will ever be dislodged and our scheme will behave exactly like regular double hashing. Thus the analysis for double hashing (which did not depend on the order in which the items were inserted) applies to our scheme as well and the average number of bounces is simply $1/(1 - \alpha)$. We need only prove that our first assertion is correct. We prove it by contradiction. Suppose that there are at least two different configurations of the table created by different orderings of the items and let x be the smallest key that appears in different locations in these two configurations. Then examine the probe sequence for x . By definition, all locations that appear before x in the probe sequence for x contain keys smaller than x ; but by assumption, all keys smaller than x appear in the same position in both configurations. Hence x sits in the first location in its probe sequence that comes after the smaller keys, which is the same location in both configurations, a contradiction.

An alternate approach to the whole affair is to use amortization. It is clear that an insertion may use fewer or more bounces than the number of probes that a successful search for the same item would require; the first occurs when the item, after being inserted, is dislodged by later insertions and so moved farther away from its initial hashing address, while the second occurs when an item is newly inserted, causing other items to be dislodged and so additional bounces beyond those need to carry the new item to its current resting place. However, if we sum all of the bounces used in all of the insertions, we note that the resulting sum equals the sum of all the probes needed to retrieve each of the items in the complete table! To see this, simply note that the number of probes needed to retrieve an item in the complete table is simply the number of times that the item has been bounced away from its initial hashing address, which is the number of bounces for the item when it was first inserted plus the total number of additional bounces that it was subjected to when dislodged by later insertions. Since the total number of probes for searching for each item in turn equals the total number of bounces for inserting all items, the average number of probes in successful search equals the average number of bounces in insertion.

- What can you now say about the average behavior of successful search? unsuccessful search?

Searching can make use of the fact that each probe sequence now has a type of ordering: on the probe sequence of item x , only items smaller than x can be encountered before x

itself. (Note that items along the probe sequence are not in sorted order; the only assertion we can make is that above.) Therefore, when searching for x in the table, we can abort the search as soon as an item larger than x is encountered—not just when hitting an empty location. Thus an unsuccessful search takes exactly the same number of probes as a successful one! This is the major difference between the regular double hashing scheme and our current scheme: in regular double hashing, unsuccessful search can be much worse than successful search.

- How would you delete an item in such a table?

Not at all... Or, at least, with great difficulty and to very little purpose. Since we now use keys to decide when to stop a search, we cannot remove a key from the table at all. We can mark the key as deleted, but we cannot remove it nor can we re-use the location for a new key, unless the new key is no larger than the old key—otherwise, it is always possible that the new key would invalidate the ordering property on at least one of the probe sequences passing through this location.

- So what do you think is the purpose behind such a scheme?

Primarily, to improve unsuccessful search. The scheme could also be used for other purposes: since, on the average, smaller keys end up closer to their original hash locations than larger ones, we could assign keys to items based on the expected frequency of retrieval and thus ensure that the most commonly wanted items are the easiest to retrieve.

The scheme is called *ordered hashing* and is due to Amble and Knuth (*Computer Journal* **17**, 2, May 1974, pp. 135–142). It remains one of the best two schemes known to date for hashing with open addressing (the other is variously known as Brent's variation or tree-structured hashing and is rather more complex, with much larger insertion costs).