Insertion With Open Addressing

Idea: keep probing until you find a free slot:
Open-Probe-Insert(T, x)

$y \leftarrow h(key[x], 0)$
$i \leftarrow 0$
while $T[y] \neq$ NIL
    do $i \leftarrow i + 1$
        $y \leftarrow h(key[x], i)$
$T[y] \leftarrow x$

Searching is similar:

- Terminate when you find the element you're looking for or an empty slot.

Deletion is tricky:

- Problem: if you delete, for example, $T(h(k, 2))$, you have to move back key k' in position $T(h(k, 3))$ if $h(k, 0) = h(k', 0)$. Similarly, may have to move back key in position $T(h(k, 4)$, $T(h(k, 5))$, ...
  - If you don't move it back, then searching won't work right.
- Have to check all items to see if they should be moved back.
- Deleting this way takes time $O(n)$.
- Alternative: just mark element as "deleted"
  - Then don't have to move back anything
  - HASH-INSERT can still use empty slot.
  - But now search time not just dependent on load factor.

As a result, in applications with deletion, chaining is more commonly used.

Linear Probing

The most obvious thing to do if a slot is already occupied is to search through the table sequentially until we find an empty slot. This is *linear probing*:

$$h(k, i) = h'(k) + i \bmod m$$

- h' is an arbitrary hash function
- start at $h'(k)$ and search forward

Naive analysis: Suppose probes are independent and the load factor is α ($\alpha < 1$ for open addressing).

- $\Pr(\text{given cell is empty}) = 1 - \alpha$.
- $E(\#\text{probes to find empty cell}) = 1/(1 - \alpha)$.

What happens in practice: *primary clustering.*

- Runs of occupied slots build up
- The expected number of probes in an unsuccessful/successful search is actually more like

$$\frac{1}{2}(1 + 1/(1-\alpha)^2) \ / \ \frac{1}{2}(1 + 1/(1-\alpha))$$

- This is not so bad if $\alpha = .5$; degrades badly if α is close to 1.

Quadratic Probing

In *quadratic probing*

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- h' is the initial hash function
- c_1, c_2 are constants
- $c_2 \neq 0$ (or else we're basically doing linear probing)

In practice, quadratic probing is much better than linear probing

- Still causes *secondary clustering*
  - $h'(k) = h'(k')$ implies that the probe sequences for k and k' are the same
- This is only a problem with high load factors

Double Hashing

In double hashing, the probe sequence depends on k

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Must have $h_2(k)$ *relatively prime* to m

- $\gcd(h_2(k), m) = 1$

Otherwise we don't probe the whole hash table.

- If gcd $= d$, we probe only $1/d$ of the hash table
- If $m = 600$, $h_2(k) = 6$, probe only 100 elements

Can guarantee gcd $= 1$ if

- m is a prime, $h_2(k) < m$
- m is a power of 2, $h_2(k)$ is odd

Analysis of Open-Address Hashing

$E(\#\text{probes in an unsuccessful search}) = 1/(1-\alpha)$

- Assuming all search sequences equally likely
- somewhat better in a successful search

Expected time for insertion: $1/(1-\alpha)$

- Insertion is more or less like an unsuccessful search

Hashing: Summary

Hashing is very useful in practice. Typically we use

- Hashing with chaining, with a load factor ~ 1
- Open-address hashing with quadratic probing and a load factor of $< .5$
  - load factors aren't comparable; we can afford a bigger table with open-address hashing

Lots of applications:

- in compilers, to keep track of declared variables in code
  - only need insert and search
- in game programs to keep track of positions
- in spell-checkers to detect misspelled words
  - can prehash dictionary

Priority Queues

Hashing is great for insertion, deletion, searching (all roughly constant time).

- But with hashing can't take max/min

If all you want to do is insert, delete, max, the *priority queues* are a good choice.

Operations for priority queues:

- INSERT(S, x): insert x into S
  - put a new job in the queue
- MAXIMUM(S): get element of S with largest key
  - Examining next job
- EXTRACT-MAX(S): remove and return element of S with largest key
  - Perform next job (and remove it from queue)

Priority queues are used to model queues/waiting lines.

Heaps

A good way of implementing a priority queue is by using a *heap*.

A *(binary) heap* data structure is an array.

- It's a way of representing a tree
- For an index i:
  - $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
  - $\textsc{Left}(i) = 2i$
  - $\textsc{Right}(i) = 2i + 1$
- If i is represented in binary, can easily compute Parent, Left, Right
- Heaps satisfy the *heap property:*

$$A[\textsc{Parent}(i)] \geq A[i]$$

  That means that heaps are (sort of) sorted

Given an array A, there may a heap in an initial subarray of A:

- *length*$[A]$ is the number of elements in A
- *heap-size*$[A] \leq$ *length*$[A]$ is the number of elements in the heap stored in A.

Heap Operations: Heapify

We want to be able to perform certain operations to manipulate heaps:

- HEAPIFY: makes the tree rooted at i a heap, if the trees rooted at LEFT(i) and RIGHT(i) are heaps.
  - Problem: $A[i]$ may be smaller than its children, violating the heap property.
  - Solution: switch $A[i]$ with the appropriate child

HEAPIFY$((A, i))$

$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
if $l \leq heapsize[A]$ and $A[l] > A[i]$
then $largest \leftarrow l$
else $largest \leftarrow i$
if $r \leq heapsize[A]$ and $A[r] > A[largest]$
then $largest \leftarrow r$
if $largest \neq i$
then exchange $A[i]$ with $A[largest]$
        HEAPIFY$(A, largest)$

Running Time of Heapify

Let $T(n)$ be the worst-case running time of $\textsc{Heapify}(A, i)$ if the subtree rooted at i has n elements.

- In the worst case, need to run Heapify on a child of i + do a constant amount of other work
- A child of i may be the root of a tree with as many as $2n/3$ children. Therefore:

$$T(n) \leq T(2n/3) + \Theta(1)$$

- By the master theorem, $T(n) = \Theta(\lg n)$.
- Alternatively, on a tree of height h, the running time of Heapify is $\Theta(h)$
  - The *height* of a tree is the length of the longest path from the root to a leaf.
  - The height of a binary tree with n nodes is $\lg n$.

Heap Operations: Building a Heap

Given an array of elements, we want to make a heap out of them.

- Run Heapify from the bottom up!

Build-Heap(A)

$heap\text{-}size(A) \leftarrow length(A)$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
  do Heapify(A, i)

Running time of Build-Heap

- Clearly $O(n \lg n)$: We call Heapify $n/2$ times.
- Can get a better upper bound, since for most of the calls, we have much smaller subtrees.
  - Claim: we call Heapify at most $\lceil n/2^{k+1} \rceil$ times on a tree of height k (with roughly 2^k nodes).

$$\sum_{k=1}^{\lfloor \lg n \rfloor} (\lceil n/2^{k+1})\rceil) ck \leq cn \sum_{k=0}^{\infty} (k/2^k) = 2cn$$

Thus, Build-Heap runs in linear time.

Calculating the sum

We can prove by induction on N that

$$\sum_{x=0}^{N} x^k = (1 - x^{N+1})/(1 - x)$$

Therefore:

$$\sum_{x=0}^{\infty} x^k = 1/(1 - x), \text{if } x < 1$$

Now differentiate both sides to get

$$\sum_{x=0}^{\infty} kx^{k-1} = 1/(1 - x)^2$$

Multiply both sides by x:

$$\sum_{x=0}^{\infty} kx^k = x/(1 - x)^2$$

Substitute $x = 1/2$:

$$\sum_{x=0}^{\infty} k/2^k = 2$$

Implementing a Priority Queue With a Heap

Suppose elements of S are stored in a heap A.

- Implement Maximum(S) with Heap-Maximum: return $A[1]$
  - Running time: $\Theta(1)$

Implement Extract-Max by returning $A[1]$, switching $A[1]$ and $A[n]$ ($n = \textit{heap-size}[A]$), and then making $A[1..n-1]$ into a heap.

Heap-Extract-Max(A)

```
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
Heapify(A, 1)
return max
```

Running time of Heap-Extract-Max: $\Theta(\lg n)$

- One call to *Heapify* + constant amount of other work

What about insertion?

- Put new element at the bottom of the heap and then percolate it up until it gets to the proper place.

HEAP-INSERT(A, x)

$\textit{heap-size}[A] \leftarrow \textit{heap-size}[A] + 1$
$i \leftarrow \textit{heap-size}[A]$
while $i > 1$ and $A[\text{PARENT}(i)] < x$
    do $A[i] \leftarrow A[\text{PARENT}(i)]$
        $i \leftarrow \text{PARENT}(i)$
$A[i] \leftarrow x$

- Running time of HEAP-INSERT: $\Theta(\lg n)$
    - We go through the loop at most $\lg n$ times, as we go from the leaf to the root

Heapsort

We can also use heaps for sorting:

If we build a heap using Build-Heap, the heap property guarantees

- the largest element will be first.
- the two subtrees of the root are heaps

Now if we switch the first and last elements:

- the last element is the largest (which is what we want in a sorted array)
- since the children of the root are still heaps, we can use Heapify

Heapsort(A)

Build-Heap(A)
for $i \leftarrow \textit{length}[A]$ **downto** 2
    do exchange $A[1] \leftrightarrow A[i]$
        $\textit{heapsize}[A] \leftarrow \textit{heapsize}[A] - 1$
        Heapify($A, 1$)

Running time of Heapsort is $O(n \lg n)$

- One call to Build-Heap: $O(n)$
- $n - 1$ calls to Heapify, each one is $O(\lg n)$