Insertion With Open Addressing

Idea: keep probing until you find a free slot:
OPEN-PROBE-INSERT(T, x)

```
y ← h(key[x], 0)
i ← 0
while T[y] ≠ NIL
    do i ← i + 1
        y ← h(key[x], i)
T[y] ← 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 keep checking if an item should be moved back until you find an empty slot
- Deleting this way may take 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.
  - This makes people uncomfortable about using this approach.

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(given cell is empty) $= 1 - \alpha$.
- E(#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(#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 $\textsc{Parent}$, $\textsc{Left}$, $\textsc{Right}$
- 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 $\textsc{Left}(i)$ and $\textsc{Right}(i)$ 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 ← LEFT(i)
r ← RIGHT(i)
if l ≤ heapsize[A] and A[l] > A[i]
then largest ← l
else largest ← i
if r ≤ heapsize[A] and A[r] > A[largest]
then largest ← r
if largest ≠ 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.

- We can do that by running HEAPIFY from the bottom up

BUILD-HEAP(A)

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

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 are dealing with much smaller subtrees:

$$\sum_{k=0}^{\lg n} (n/2^k)ck \le cn \sum_{k=0}^{\infty} (k/2^k) = 2cn$$

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 $A[n]$, and then making $A[1..n-1]$ into a heap (as in HEAPSORT).

HEAP-EXTRACT-MAX(A)

if $heap\text{-}size[A] < 1$
  then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[heap\text{-}size[A]]$
$heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
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)

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$i \leftarrow heap\text{-}size[A]$
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 $i \leftarrow$ *length*[A] **downto** 2
  do exchange $A[1] \leftrightarrow A[i]$
    heapsize[A] $\leftarrow$ *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 $O(\lg n)$

Binary Search Trees

Heaps are good for insertion, deletion, searching.

Priority heaps are good for minimum/maximum.

Binary search trees (BSTs) are a useful data structure to implement dictionary operations, min, max, successor, predecessor.

- basic operations take time O(height tree)
- randomly built BST with n nodes has height $\lg(n)$
- will consider variants of BSTs – *red-black trees* – that are guaranteed to have height $O(\lg n)$
- Another variant, *B-trees*, are used in databases
- lots of other variants
  - splay trees
  - AVL trees
  - persistent trees
- The great number of variants is an indication of the importance of BSTs.

The Binary-Search Tree Property

A binary search tree is a binary tree where each node has key, parent, left child, right child

- $p[x] =$ NIL for the root
- *left*[x], *right*[x] may be NIL

The keys must satisfy the *binary-search-tree* (BST) property:

> If y is a node in the left subtree of x, then $key[y] \leq key[x]$
> If y is a node in the right subtree of x, then $key[y] \geq key[x]$

Note: This property makes sense only if the keys are totally ordered

Searching a Binary Search Tree

Searching is easy because of the BST property:

TREE-SEARCH(x, k) [x is a pointer to a node]

if $x =$ NIL or $k = key[x]$
  then return x
if $k < key[x]$
  then return TREE-SEARCH(*left*[x], k)
  else return TREE-SEARCH(*right*[x], k)

- This tells us whether k appears in the subtree rooted at x
- running time: $O(h(x))$, where $h(x)$ is the height of x

Here is a non-recursive version:

ITERATIVE-TREE-SEARCH(x, k)

while $x \neq$ NIL and $k \neq key[x]$
  do if $k < key[x]$
    then $x \leftarrow left[x]$
    else $x \leftarrow right[x]$
return x

Minimum and maximum

Min and max are easy: just go all the way to the left/right:

TREE-MINIMUM(x) [x is a pointer to a node]

```
while left[x] ≠ NIL
    do x ← left[x]
return x
```

TREE-MAXIMUM(x)

```
while right[x] ≠ NIL
    do x ← right[x]
return x
```


Successor and Predecessor

The successor of x is the element with the next-biggest key

- May want successor if you want to list keys in increasing order
- Again, this makes sense only if keys are totally ordered

Where is the successor of x located?

1. If x has a right child, then it's the leftmost node of the subtree rooted at the right child.
   - Clearly this is the successor of x in the subtree rooted at x
   - Work up the tree by induction from x to show that this remains true
2. If x has no right child, and x is the left child of its parent, then the successor is the parent
   - Again, need to argue by induction up the tree that this is right
3. If x is the right child of its parent, find the lowest ancestor of x which is the left child of its parent

TREE-SUCCESSOR(x)

```
if right[x] ≠ NIL
  then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
    do x ← y
       y ← p[y]
return y
```

- TREE-PREDECESSOR works the same way
- Both run in time $O(h)$:
  - We either go up the tree or down the tree

Insertion

Inserting z is straightforward:

- We insert z at a leaf
- Figure out which one by starting at the root and making comparisons

TREE-INSERT(T, z)

```
y ← NIL
x ← root[T]        [y is the parent of x]
while x ≠ NIL
    do y ← x
       if key[z] < key[x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```

Insertion clearly runs in time $O(h)$

Deletion in BSTs

Deleting z is the trickiest operation. There are three cases:

1. z has no children: easy – just delete z
2. z has one child: easy – delete z; child of z becomes child of z's parent
   - we still maintain the BST property
3. if z has two children
   - Find z's successor z'
     - this will be the leftmost element in the subtree rooted at *right*$[z]$
   - recursively delete z'
     - this is easy because z' has at most one child (no left child)
   - Replace z by z'
   - This maintains the BST property

Tree-Delete(T, z)

if *left*$[z]$ = NIL or *right*$[x]$ = NIL
  then $y \leftarrow z$
  else $y \leftarrow$ Tree-Successor$[z]$
    [y is the node that gets spliced out]
if *left*$[y] \neq$ NIL
  then $x \leftarrow$ *left*$[y]$
  else $x \leftarrow$ *right*$[y]$
    [x is the unique successor of y (or NIL)]
if $x \neq$ NIL
  then $p[x] \leftarrow p[y]$
if $p[y]$ = NIL
  then *root*$[T] \leftarrow x$
  else if y = *left*$[p[y]]$
      then *left*$[p[y]] \leftarrow x$
      else *right*$[p[y]] \leftarrow x$
if $y \neq z$
  key$[z] \leftarrow$ *key*$[y]$
    [also copy other fields, if there are any]

Again, the running time is $O(h)$.

The Height of a Random BST

All the algorithms run in time $O(h)$.

What's h for an n-node tree?

- best case: $\lg(n)$ – if the tree is perfectly balanced
- worst case: $O(n)$ – if the tree is completely unbalanced

What can we expect on average?

Let's assume the tree is built up by starting with an empty tree and inserting n elements.

- it's very hard to analyze what happens if we have inserts + deletes
  - deletes could unbalance a tree—if a node has two children, we delete from the right subtree.

If the n elements are in increasing or decreasing order, then we have a completely unbalanced tree.

- This can be a serious problem in practice
- Running time $O(n)$ is not acceptable
- Red-black trees solve that problem

If all the $n!$ permutations of the trees are equally likely, then the expected height of the tree is $O(\lg n)$.

Using a BST for Sorting

Can sort using a BST by doing an *inorder* traversal

- first left subtree, then root, then right subtree

Inorder-Tree-Walk(x) [walk through subtree rooted at x]

if $x \neq$ NIL
  then Inorder-Tree-Walk(*left*$[x]$)
      print *key*$[x]$
      Inorder-Tree-Walk(*right*$[x]$)

Analysis: first need to build the BST by inserting elements to be sorted. This takes expected time

$$O(\lg(1)) + \cdots + O(\lg n) = O(n \lg n)$$

The tree walk then takes time $O(n)$.

Balanced Search Trees

The BSTs just presented only have expected height $O(\lg n)$. There are a number of variants which are *guaranteed* to have height $O(\lg n)$:

- red-black trees (CLR; Chapter 14)
- AVL trees
- ...

Keeping the tree balanced requires (lots of) additional overhead, although the basic ideas remain the same.