Today’s Outline

- Hash Tables
- Trees

Direct Addressing Problem

- If universe $U$ is large, storing the array $T$ may be impractical
- Also much space can be wasted in $T$ if number of objects stored is small
- Q: Can we do better?
- A: Yes we can trade time for space

Hash Tables

- “Key” Idea: An element with key $k$ is stored in slot $h(k)$, where $h$ is a hash function mapping $U$ into the set $\{0, \ldots, m-1\}$
- Main problem: Two keys can now hash to the same slot
- Q: How do we resolve this problem?
- A1: Try to prevent it by hashing keys to “random” slots and making the table large enough
- A2: Chaining
- A3: Open Addressing
Chained Hash

In chaining, all elements that hash to the same slot are put in a linked list.

CH-Insert(T, x) { Insert x at the head of list T[h(key(x))]; }  
CH-Search(T, k) { search for elem with key k in list T[h(k)]; }  
CH-Delete(T, x) { delete x from the list T[h(key(x))]; }

Analysis

• CH-Insert and CH-Delete take $O(1)$ time if the list is doubly linked and there are no duplicate keys
• Q: How long does CH-Search take?
• A: It depends. In particular, depends on the load factor, $\alpha = n/m$ (i.e. average number of elems in a list)

CH-Search Analysis

• Worst case analysis: everyone hashes to one slot so $\Theta(n)$
• For average case, make the simple uniform hashing assumption: any given elem is equally likely to hash into any of the $m$ slots, indep. of the other elems
• Let $n_i$ be a random variable giving the length of the list at the $i$-th slot
• Then time to do a search for key $k$ is $1 + n_{h(k)}$

• Q: What is $E(n_{h(k)})$?
• A: We know that $h(k)$ is uniformly distributed among $\{0, \ldots, m-1\}$
• Thus, $E(n_{h(k)}) = \sum_{i=0}^{m-1} (1/m) n_i = n/m = \alpha$
Hash Functions

- Want each key to be equally likely to hash to any of the $m$ slots, independently of the other keys
- Key idea is to use the hash function to “break up” any patterns that might exist in the data
- We will always assume a key is a natural number (can e.g. easily convert strings to natural numbers)

Division Method

- $h(k) = k \mod m$
- Want $m$ to be a prime number, which is not too close to a power of 2
- Why?

Multiplication Method

- $h(k) = \lfloor m \times (kA \mod 1) \rfloor$
- $kA \mod 1$ means the fractional part of $kA$
- Advantage: value of $m$ is not critical, need not be a prime
- $A = (\sqrt{5} - 1)/2$ works well in practice

Open Addressing

- All elements are stored in the hash table, there are no separate linked lists
- When we do a search, we probe the hash table until we find an empty slot
- Sequence of probes depends on the key
- Thus hash function maps from a key to a “probe sequence” (i.e. a permutation of the numbers $0, \ldots, m - 1$)
In general, for open addressing, the hash function depends on both the key to be inserted and the probe number.

Thus for a key $k$, we get the probe sequence $h(k, 0), h(k, 1), \ldots, h(k, m-1)$.

If we use open addressing, the hash table can never fill up i.e. the load factor $\alpha$ can never exceed 1.

An advantage of open addressing is that it avoids pointers and the overhead of storing lists in each slot of the table.

This freed up memory can be used to create more slots in the table which can reduce the load-factor and potentially speed up retrieval time.

A disadvantage is that deletion is difficult. If deletions occur in the hash table, chaining is usually used.

**OA-Insert**

```c
OA-Insert(T,k){
i = 0;
repeat {
j = h(k,i);
if (T[j] = nil){
    T[j] = k;
    return j;
}
else i++;
} until (i==m);
}
```

**OA-Search**

```c
OA-Insert(T,k){
i = 0;
repeat {
j = h(k,i);
if (T[j] = k){
    return j;
}
else i++;
} until (T[j]==nil or i==m);
}
```
OA-Delete

- Deletion from an open-address hash table is difficult
- When we delete a key from slot $i$, we can’t just mark that slot as empty by storing nil there
- The problem is that this would make it impossible to find some key $k$ during whose insertion we probed slot $i$ and found it occupied

OA-Delete

- One solution is to mark the slot by storing in it the value “DELETED”
- Then we modify OA-Insert to treat such a slot as if it were empty so that something can be stored in it
- OA-Search passes over these special slots while searching
- Note that if we use this trick, search times are no longer dependent on the load-factor $\alpha$ (for this reason, chaining is more commonly used when keys must be deleted)

Implementation

- To analyze open-address hashing, we make the assumption of uniform hashing: we assume that each key is equally likely to have any of the $m!$ permutations of $\{0, 1, \ldots, m - 1\}$ as its probe sequence
- True uniform hashing is difficult to implement, so in practice, we generally use one of three approximations on the next slide

Implementations

- All positions are taken modulo $m$, and $i$ ranges from 1 to $m - 1$
  - Linear Probing: Initial probe is to position $h(k)$, successive probes are to positions $h(k) + i$,
  - Quadratic Probing: Initial probes is to position $h(k)$, successive probes are to position $h(k) + c_1i + c_2i^2$
  - Double Hashing: Initial probe is to position $h(k)$, successive probes are to positions $h(k) + ih_2(k)$
Analysis

- Recall that the load factor, \( \alpha \), is the number of elements stored in the hash table, \( n \), divided by the total number of slots \( m \).
- In open-address hashing, we have at most one element per slot so \( \alpha < 1 \).
- We assume uniform hashing i.e. each probe maps to essentially a random slot in the table.
- We can show that the expected time for insertions is at most \( 1/\alpha \), the expected time for an unsuccessful search is \( 1/(1 - \alpha) \) and the expected time for a successful search is \( (1/\alpha) \ln[1/(1 - \alpha)] \).

Hash Tables Wrapup

Hash Tables implement the Dictionary ADT, namely:

- Insert(\( x \)) - \( O(1) \) expected time, \( \Theta(n) \) worst case
- Lookup(\( x \)) - \( O(1) \) expected time, \( \Theta(n) \) worst case
- Delete(\( x \)) - \( O(1) \) expected time, \( \Theta(n) \) worst case

Binary Search Trees

- Binary Search Trees are another data structure for implementing the dictionary ADT.

Red-Black Trees

Red-Black trees (a kind of binary tree) also implement the Dictionary ADT, namely:

- Insert(\( x \)) - \( O(\log n) \) time
- Lookup(\( x \)) - \( O(\log n) \) time
- Delete(\( x \)) - \( O(\log n) \) time
Why BST?

- Q: When would you use a Search Tree?
- A1: When need a hard guarantee on the worst case run times (e.g. “mission critical” code)
- A2: When want something more dynamic than a hash table (e.g. don’t want to have to enlarge a hash table when the load factor gets too large)
- A3: Search trees can implement some other important operations...

Search Tree Operations

- Insert
- Lookup
- Delete
- Minimum/Maximum
- Predecessor/Successor

What is a BST?

- It’s a binary tree
- Each node holds a key and record field, and a pointer to left and right children
- *Binary Search Tree Property* is maintained

Binary Search Tree Property

- Let \( x \) be a node in a binary search tree. If \( y \) is a node in the left subtree of \( x \), then \( \text{key}(y) \leq \text{key}(x) \). If \( y \) is a node in the right subtree of \( x \) then \( \text{key}(x) \leq \text{key}(y) \)
Inorder Tree-Walk

Inorder-TW(x)

if (x is not nil)
    Inorder-TW(left(x));
    print key(x);
    Inorder-TW(right(x));
}
Analysis
• Correctness?
• Run time?

Search in BT
Tree-Search(x,k){
  if (x=nil) or (k = key(x)){
    return x;
  }
  if (k<key(x)){
    return Tree-Search(left(x),k);
  }else{
    return Tree-Search(right(x),k);
  }
}

Analysis
• Let $h$ be the height of the tree
• The run time is $O(h)$
• Correctness???

In-Class Exercise
• Q1: What is the loop invariant for Tree-Search?
• Q2: What is Initialization?
• Q3: Maintenance?
• Q4: Termination?