A dictionary ADT implements the following operations:

- **Insert(x)**: puts the item x into the dictionary
- **Delete(x)**: deletes the item x from the dictionary
- **IsIn(x)**: returns true iff the item x is in the dictionary

Frequently, we think of the items being stored in the dictionary as keys.

The keys typically have records associated with them which are carried around with the key but not used by the ADT implementation.

Thus we can implement functions like:

- **Insert(k,r)**: puts the item (k,r) into the dictionary if the key k is not already there, otherwise returns an error
- **Delete(k)**: deletes the item with key k from the dictionary
- **Lookup(k)**: returns the item (k,r) if k is in the dictionary, otherwise returns null
Implementing Dictionaries

- The simplest way to implement a dictionary ADT is with a linked list
- Let $l$ be a linked list data structure, assume we have the following operations defined for $l$
  - `head(l)`: returns a pointer to the head of the list
  - `next(p)`: given a pointer $p$ into the list, returns a pointer to the next element in the list if such exists, null otherwise
  - `previous(p)`: given a pointer $p$ into the list, returns a pointer to the previous element in the list if such exists, null otherwise
  - `key(p)`: given a pointer into the list, returns the key value of that item
  - `record(p)`: given a pointer into the list, returns the record value of that item

In-Class Exercise

- Q1: Write the operation `Lookup(k)` which returns a pointer to the item with key $k$ if it is in the dictionary or null otherwise
- Q2: Write the operation `Insert(k, r)`
- Q3: Write the operation `Delete(k)`
- Q4: For a dictionary with $n$ elements, what is the runtime of all of these operations for the linked list data structure?

In-Class Exercise

- Q5: Describe how you would use this dictionary ADT to count the number of occurrences of each word in an online book.
- Q6: If $m$ is the total number of words in the online book, and $n$ is the number of unique words, what is the runtime of the algorithm for the previous question?

Dictionaries

- This linked list implementation of dictionaries is very slow
- Q: Can we do better?
- A: Yes, with hash tables, AVL trees, etc
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

Direct Address Functions

- DA-Search(T,k){ return T[k];}
- DA-Insert(T,x){ T[key(x)] = x;}
- DA-Delete(T,x){ T[key(x)] = NIL;}

Each of these operations takes $O(1)$ time

Direct Addressing

- Suppose universe of keys is $U = \{0, 1, \ldots, m - 1\}$, where $m$ is not too large
- Assume no two elements have the same key
- We use an array $T[0..m - 1]$ to store the keys
- Slot $k$ contains the elem with key $k$

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, $E(n_{h(k)}) = \sum_{i=0}^{m-1} (1/m)n_i = n/m = \alpha$

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?

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 naturaly numbers)

Multiplication Method

- $h(k) = \lfloor m \cdot (kA \mod 1) \rfloor$
- $kA \mod 1$ 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, ..., \(m - 1\))

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 \(h(k) + ih_2(k)\)