Hash Tables

Dynamic Set Operations
Direct Address Tables
Hash Tables
Collision Resolution by Chaining
Simple Uniform Hashing Assumption
Hash Functions
Open Addressing

Main Dynamic Set Operations

**SEARCH(S, k)**
A query that, given a set \( S \) and a key value \( k \), returns a pointer \( x \) to an element in \( S \) such that \( x.key = k \), or NIL if no such element belongs to \( S \).

**INSERT(S, x)**
A modifying operation that augments the set \( S \) with the element pointed to by \( x \). We usually assume that any attributes in element \( x \) needed by the set implementation have already been initialized.

**DELETE(S, x)**
A modifying operation that, given a pointer \( x \) to an element in the set \( S \), removes \( x \) from \( S \). (Note that this operation takes a pointer to an element \( x \), not a key value.)

Other Dynamic Set Operations

**MINIMUM(S)**
A query on a totally ordered set \( S \) that returns a pointer to the element of \( S \) with the smallest key.

**MAXIMUM(S)**
A query on a totally ordered set \( S \) that returns a pointer to the element of \( S \) with the largest key.

**SUCCESSOR(S, x)**
A query that, given an element \( x \) whose key is from a totally ordered set \( S \), returns a pointer to the next larger element in \( S \), or NIL if \( x \) is the maximum element.

**PREDECESSOR(S, x)**
A query that, given an element \( x \) whose key is from a totally ordered set \( S \), returns a pointer to the next smaller element in \( S \), or NIL if \( x \) is the minimum element.
Direct Addressing

To represent the dynamic set, we use an array, or direct-address table, denoted by $T[0..m-1]$, in which each position, or slot, corresponds to a key in the universe $U$. Figure 11.1 illustrates the approach; slot $k$ points to an element in the set with key $k$. If the set contains no element with key $k$, then $T[k] = \text{NIL}$.

The dictionary operations are trivial to implement:

DIRECT-ADDRESS-SEARCH($T, k$)
1 return $T[k]

DIRECT-ADDRESS-INSERT($T, x$)
1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE($T, x$)
1 $T[x.key] = \text{NIL}$

Hashing Basics

With direct addressing, an element with key $k$ is stored in slot $k$. With hashing, this element is stored in slot $h(k)$; that is, we use a hash function $h$ to compute the slot from the key $k$. Here, $h$ maps the universe $U$ of keys into the slots of a hash table $T[0..m-1]$:

$h : U \rightarrow \{0, 1, \ldots, m-1\}$

There is one hitch: two keys may hash to the same slot. We call this situation a collision. Fortunately, we have effective techniques for resolving the conflict created by collisions.

Hashing Illustration

![Hashing Illustration Diagram]

$U$ (universe of keys)

$K$ (actual keys)

The diagram illustrates a hash table $T$ with keys $k_1, k_2, k_3, k_4, k_5$. The hash function $h$ maps keys to slots in the table.
Collision Resolution by Chaining

In chaining, we place all the elements that hash to the same slot into the same linked list, as Figure 11.3 shows. Slot \( j \) contains a pointer to the head of the list of all stored elements that hash to \( j \); if there are no such elements, slot \( j \) contains NIL.

The dictionary operations on a hash table \( T \) are easy to implement when collisions are resolved by chaining:

\[
\begin{align*}
\text{CHAINED-Hash-Insert}(T, x) & \quad 1 \quad \text{insert } x \text{ at the head of list } T[h(x, \text{key})] \\
\text{CHAINED-Hash-Search}(T, k) & \quad 1 \quad \text{search for an element with key } k \text{ in list } T[h(k)] \\
\text{CHAINED-Hash-Delete}(T, x) & \quad 1 \quad \text{delete } x \text{ from the list } T[h(x, \text{key})]
\end{align*}
\]

Chaining Analysis Definitions

Given a hash table \( T \) with \( m \) slots that stores \( n \) elements, we define the load factor \( \alpha \) for \( T \) as \( n/m \), that is, the average number of elements stored in a chain. Our analysis will be in terms of \( \alpha \), which can be less than, equal to, or greater than 1. We assume that any given element is equally likely to hash into any of the \( m \) slots, independently of where any other element has hashed to. We call this the assumption of simple uniform hashing.

Chaining Analysis Theorems

**Theorem 11.1**
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time \( \Theta(1+\alpha) \), under the assumption of simple uniform hashing.

**Theorem 11.2**
In a hash table in which collisions are resolved by chaining, a successful search takes average-case time \( \Theta(1+\alpha) \), under the assumption of simple uniform hashing.
**Hash Functions**

For example, if we know that the keys are random real numbers $k$ independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of simple uniform hashing.

In the **division method** for creating hash functions, we map a key $k$ into one of $m$ slots by taking the remainder of $k$ divided by $m$. That is, the hash function is

$$h(k) = k \mod m.$$  

The **multiplication method** for creating hash functions operates in two steps. First, we multiply the key $k$ by a constant $A$ in the range $0 < A < 1$ and extract the fractional part of $kA$. Then, we multiply this value by $m$ and take the floor of the result. In short, the hash function is

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

where “$kA \mod 1$” means the fractional part of $kA$, that is, $kA - \lfloor kA \rfloor$.

---

**Collision Resolution by Open Addressing**

In **open addressing**, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m - 1$ (which requires $\Theta(n)$ search time), the sequence of positions probed **depends upon the key being inserted**. To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\}.$$  

---

**Open Addressing Hash-Insert**

```
OPEN-INSERT(T, k)
1 i = 0
2 repeat
3 j = h(k, i)
4 if T[j] == NIL
5 T[j] = k
6 return j
7 else i = i + 1
8 until i == m
9 error “hash table overflow”
```

---

**Open Addressing Hash-Search**

```
OPEN-SEARCH(T, k)
1 i = 0
2 repeat
3 j = h(k, i)
4 if T[j] == k
5 return j
6 i = i + 1
7 until T[j] == NIL or i == m
8 return NIL
```
Linear Probing

Given an ordinary hash function \( h' : U \rightarrow \{0, 1, \ldots, m - 1\} \), which we refer to as an auxiliary hash function, the method of linear probing uses the hash function

\[
h(k, i) = (h'(k) + i) \mod m
\]

for \( i = 0, 1, \ldots, m - 1 \). Given key \( k \), we first probe \( T[h'(k)] \), i.e., the slot given by the auxiliary hash function. We next probe slot \( T[h'(k) + 1] \), and so on up to slot \( T[m - 1] \). Then we wrap around to slots \( T[0], T[1], \ldots \) until we finally probe slot \( T[h'(k) - 1] \). Because the initial probe determines the entire probe sequence, there are only \( m \) distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by \( i \) full slots gets filled next with probability \( (i + 1)/m \). Long runs of occupied slots tend to get longer, and the average search time increases.

Other Probing Techniques

**Quadratic probing** uses a hash function of the form

\[
h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m
\]

where \( h' \) is an auxiliary hash function, \( c_1 \) and \( c_2 \) are positive auxiliary constants.

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. **Double hashing** uses a hash function of the form

\[
h(k, i) = (h_1(k) + ih_2(k)) \mod m
\]

where both \( h_1 \) and \( h_2 \) are auxiliary hash functions.