To represent the dynamic set, we use an array, or direct-address table, denoted by \( T[0 \ldots m - 1] \), 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. \( T[x\text{.key}] = x \) 

**DIRECT-ADDRESS-DELETE** \((T, x)\)

1. \( T[x\text{.key}] = \text{NIL} \)

All operations are \( \Theta(1) \).

[Note: not counting the time to create the table.]

For hash tables (under certain assumptions), search, insert, & delete are \( \Theta(1) \) on average.
Suppose keys are banner ids
- 8 digit ids
Direct addressing wouldn't make a table of size
100,000,000
not very practical, mostly empty.

Maybe not a bad idea for social security numbers.
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.
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:

- **CHAINED-HASH-INSERT** $(T, x)$
  
  1. insert $x$ at the head of list $T[h(x\text{.key})]$  

- **CHAINED-HASH-SEARCH** $(T, k)$
  
  1. search for an element with key $k$ in list $T[h(k)]$  

- **CHAINED-HASH-DELETE** $(T, x)$
  
  1. delete $x$ from the list $T[h(x\text{.key})]$
Chaining Illustration

Dynamic Set 1
Dynamic Set 2
Direct Addressing 1
Direct Addressing 2
Hashing 1
Hashing 2
Chaining 1
▷ Chaining 2
Chaining 3
Chaining 4
Hash Functions
Open Address 1
Open Address 2
Open Address 3
Open Address 4
Open Address 5

**U** (universe of keys)

**K** (actual keys)

- **k₁**
- **k₂**
- **k₃**
- **k₄**
- **k₅**
- **k₆**
- **k₇**
- **k₈**
- **k₉**

**T**

- **k₁**
- **k₄**
- **k₅**
- **k₇**
- **k₃**
- **k₆**

Keys are: 2, 3, 5, 7, 11, 13, 17, 19

Hash table with 10 slots

\[ h(k) = k \mod 10 \]

Load factor is: \( \frac{8}{10} \)

\( n = 8 \), \( m = 10 \)
Given a hash table $T$ with $m$ slots that stores $n$ elements, we define the \textit{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.

For open addressing, require $\alpha \leq 1$, prefer around $\frac{1}{2}$, an alternative to chaining.

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 \textit{simple uniform hashing}. Do what we can to make this true.

Not true for longer ids, need to take some care.

Mostly true for about 123 ids, to choose hash function.

Can analyze what happens when assumption is true.

Ensure as much as possible to be close to assumption.