CSCI 470: Binary Search Trees (Insertion & Deletion), Hashing

Vijay Chaudhary

Oct 04, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. BST: Querying search
2. BST: Insertion & Deletion
3. Hashing

BST: Querying search

BST: Querying Search

We will go over TREE-SEARCH, MAXIMUM, MINIMUM, and SUCCESSOR. In this section, we shall examine these operations and show how to support each one in time $O(h)$ on any binary search tree of height h.

BST: Searching

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else
6        return TREE-SEARCH(x.right, k)
```

Runtime ?

BST: Searching iteratively

ITERATIVE-TREE-SEARCH(x, k)

```
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else
5              x = x.right
6  return x
```

Runtime ?

BST: minimum and maximum

TREE-MINIMUM(x)

```
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

Runtime ?

TREE-MAXIMUM(x)

```
1  while x.right ≠ NIL
2        x = x.right
3  return x
```

Runtime ?

BST: Successor

TREE-SUCCESSOR(x)

```
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

Runtime ?

BST: Insertion & Deletion

Figure: reference

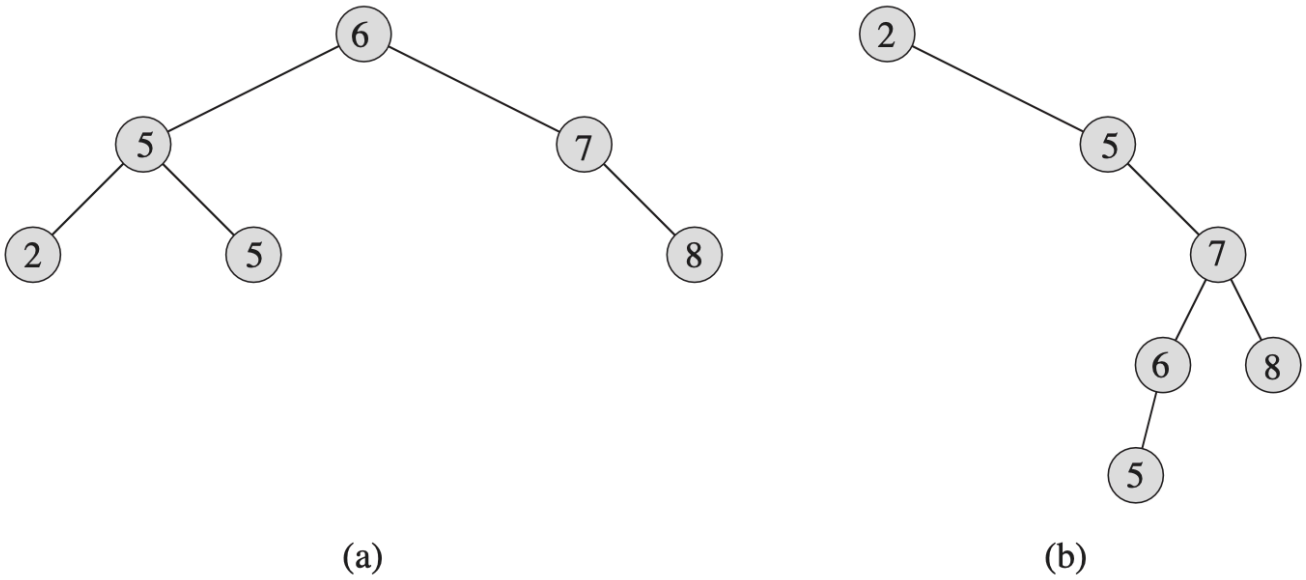


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

Insertion

TREE-INSERT(T, z)

```
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z   // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

Insertion: Figure

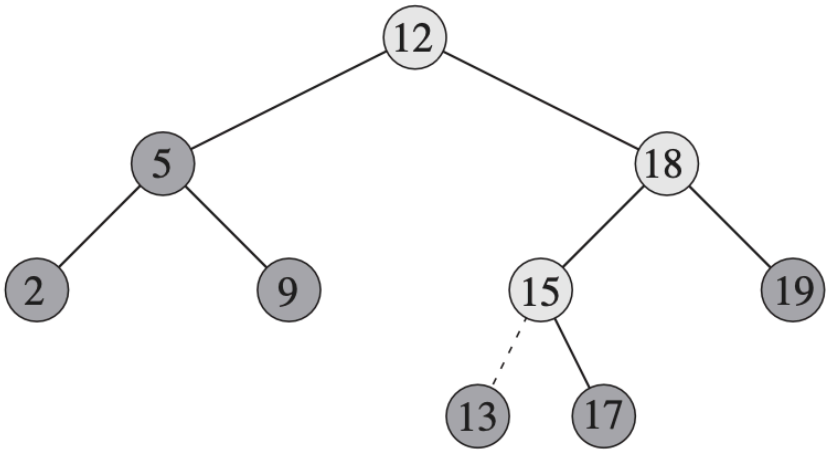


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Deletion cases

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.

Deletion cases

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.

Deletion cases

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, then we find z's successor y - which must be in z's right substree - and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. This case is the tricky one because, **it matters whether y is z's right child**.

Deletion

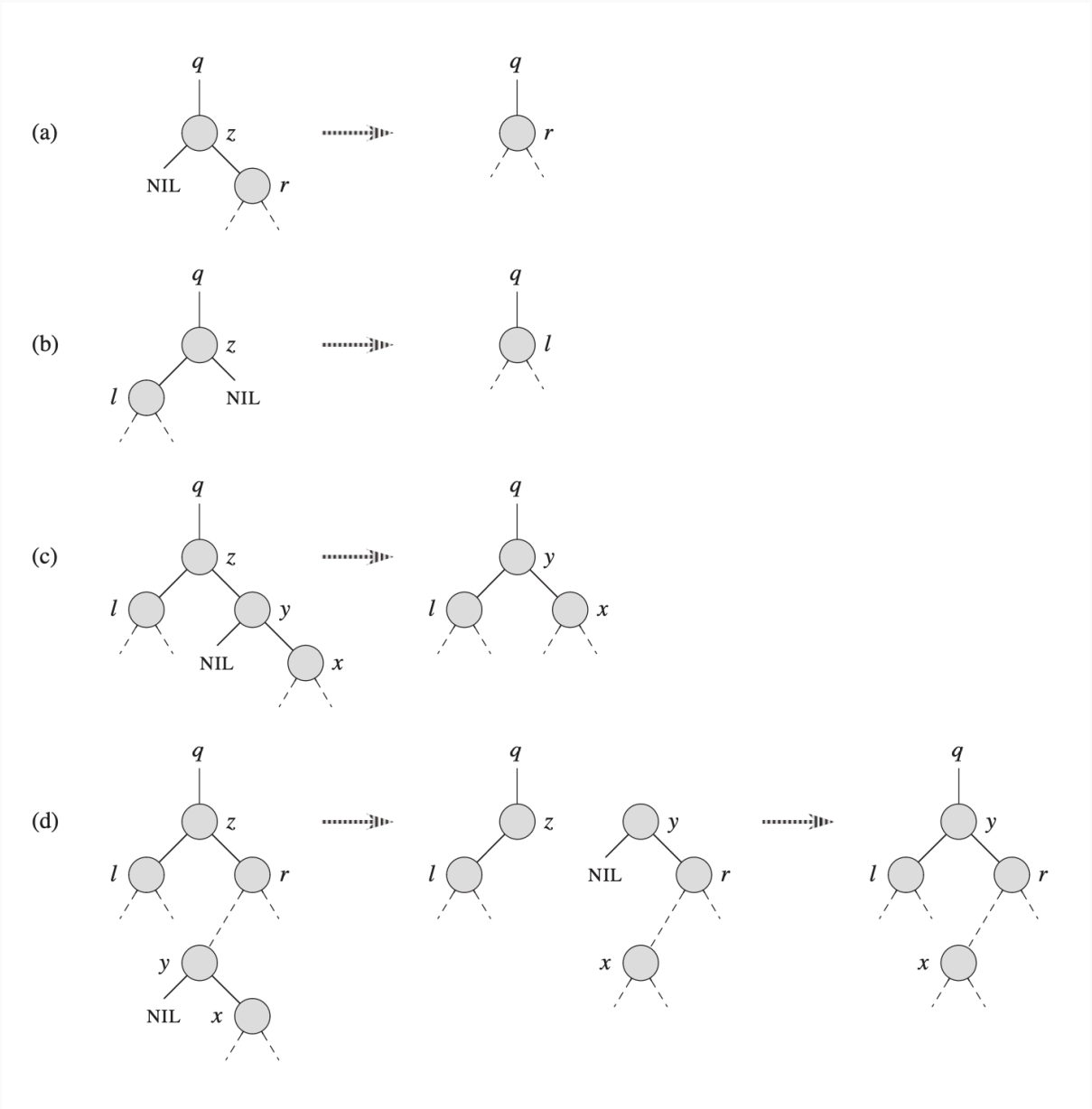
TRANSPLANT(T, u, v)

```
1  if u.p == NIL // u is the root.
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

Deletion (continued)

```
TREE-DELETE(T, z)
 1  if z.left == NIL // Case 1
 2        TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL // Case 2
 4        TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSPLANT(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSPLANT(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

Deletion: Figure



BST Deletion: Runtime

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs $O(h)$ time on a tree of height h.

Hashing

Direct address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Direct address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- We assume that the maximum value is not too large.

Direct address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- We assume that the maximum value is not too large.
- We also assume that no two elements have the same key.

Direct Address Table: Figure

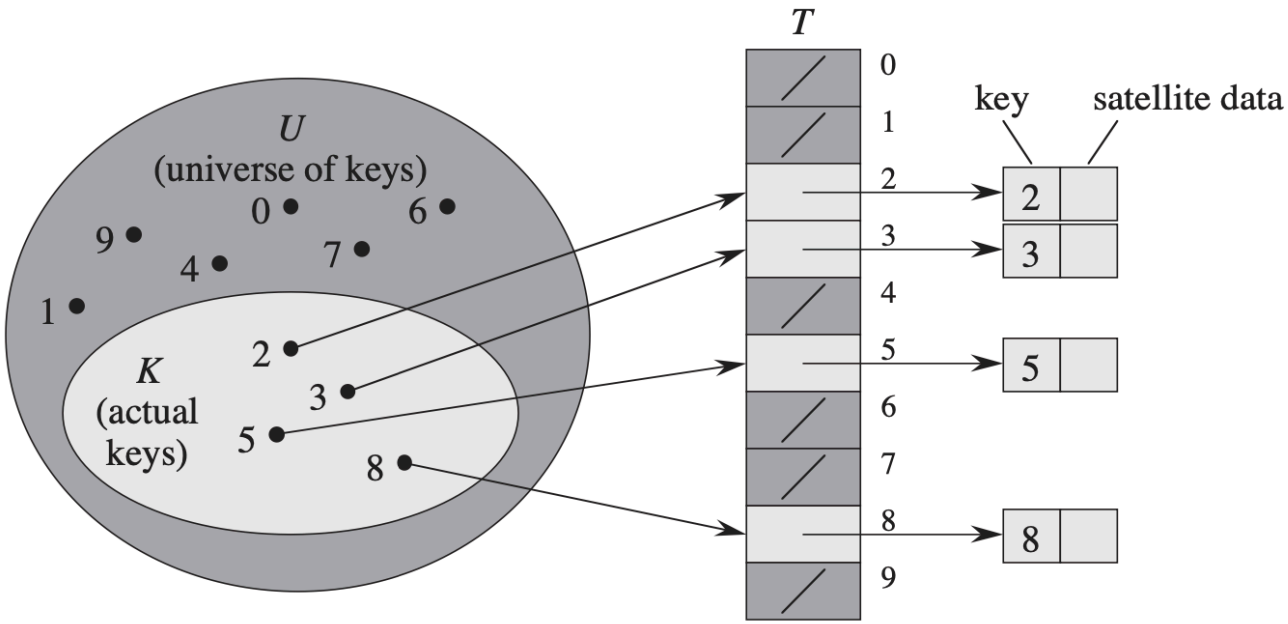


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Direct address tables

- With $U = \{0, 1, ..., m - 1\}$, we can use an array for **direct-address table** such as $T[0 \,..\, m - 1]$, in which each position, or **slot**, corresponds to a key in the universe U.
- Think of the array to keep the counts in counting sort, where we used the index to keep the count of the element. Similarly, we are using each position in the array as a slot that correspond to the key.
- Either we can use the $T[k]$ to store the key itself or we can have a pointer at $T[k]$ that points to the key. In either case, we need to find a way to represent when the slot is empty.

Dictionary operations

DIRECT-ADDRESS-SEARCH

```
1  return T[k]
```

DIRECT-ADDRESS-INSERT

```
1  T[x.key] = x
```

DIRECT-ADDRESS-DELETE

```
1  T[x.key] = NIL
```

Pros and cons of direct address tables

Pros:

- Implementation is trivial.
- insertion, deletion, and lookup can be done at $O(1)$.

Cons:

- Table will be very sparse if there is a large gap between the maximum value and second largest value.
- Only useful if the maximum value is very small.

Hash tables

- 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.

Hash tables

- 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.
- $h : U \to \{0, 1, ..., m-1\}$ into the slots of a **hash table** $T[0..m-1]$, where the size of $m \ll |U|$.

Hash tables

- 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.
- $h : U \to \{0, 1, ..., m-1\}$ into the slots of a **hash table** $T[0..m-1]$, where the size of $m \ll |U|$.
- We say that an element with key k ***hashes*** to slot $h(k)$ is the **hash value** of key k.

Hash tables

- 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.
- $h : U \to \{0, 1, ..., m - 1\}$ into the slots of a **hash table** $T[0 \,..\, m - 1]$, where the size of $m \ll |U|$.
- We say that an element with key k ***hashes*** to slot $h(k)$ is the **hash value** of key k.
- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.

Hash tables

- 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.
- $h : U \to \{0, 1, ..., m-1\}$ into the slots of a **hash table** $T[0..m-1]$, where the size of $m \ll |U|$.
- We say that an element with key k ***hashes*** to slot $h(k)$ is the **hash value** of key k.
- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.
- Since $|U| > m$, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible.

Hash Table: figure

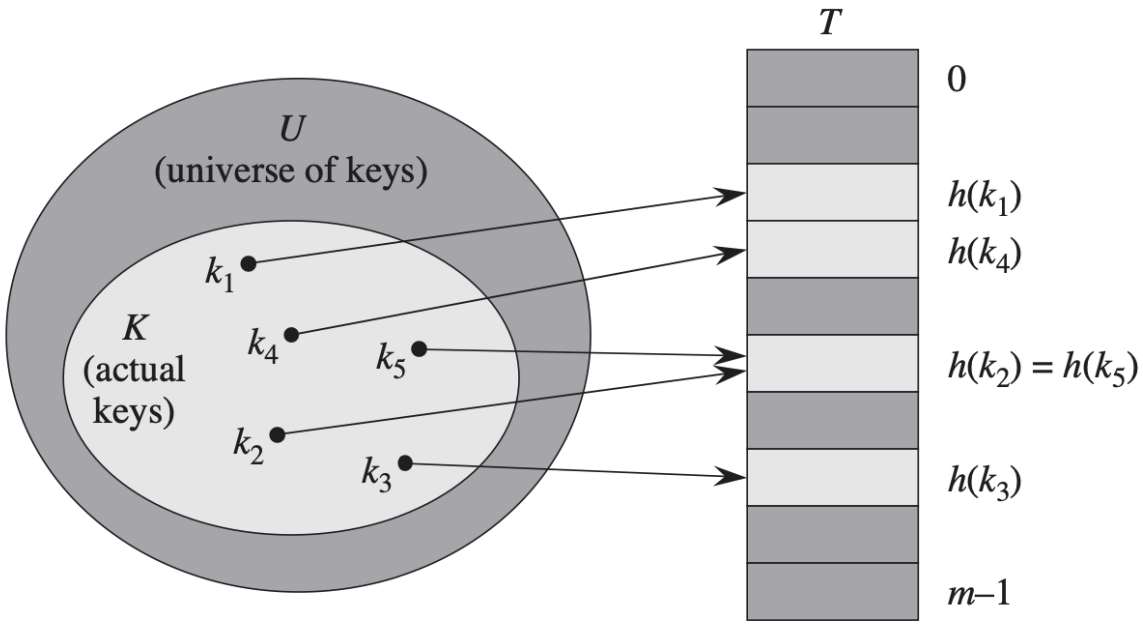


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

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.

Hash Table: figure

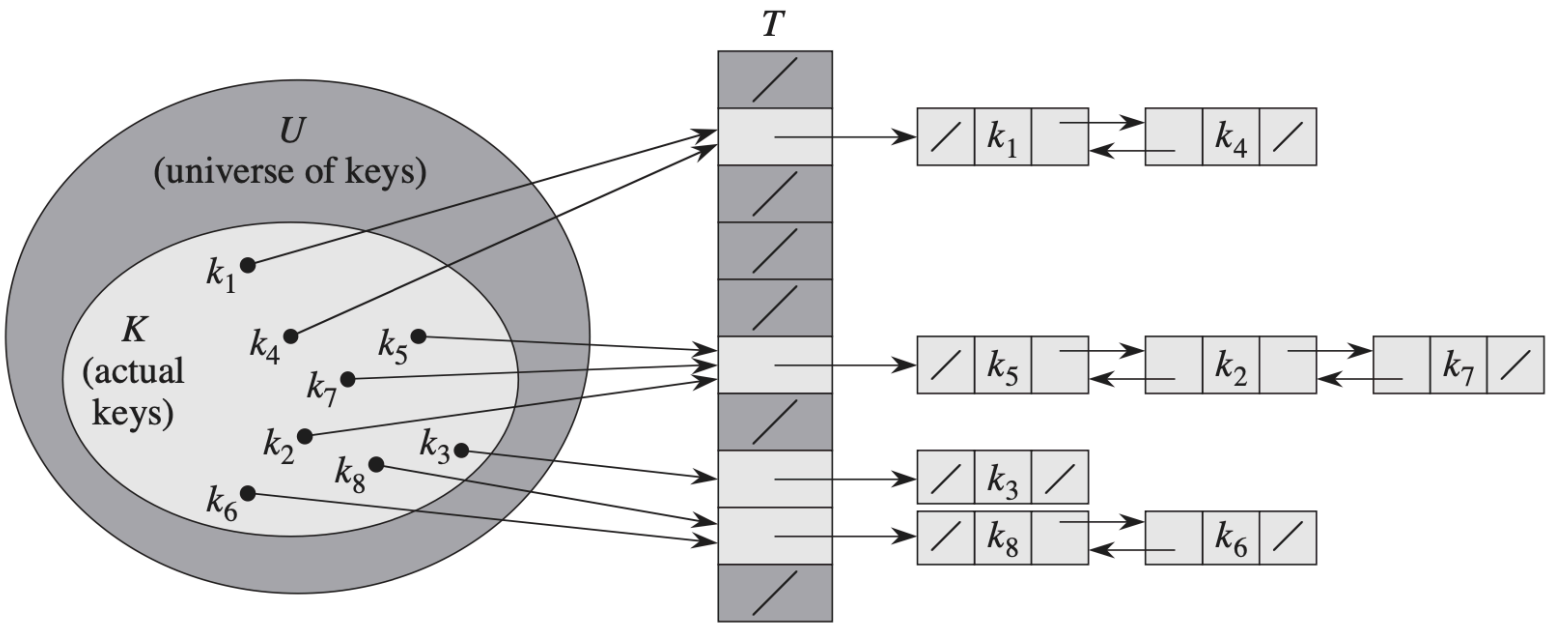


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

Dictionary operations on a hash table T

CHAINED-HASH-INSERT

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE

1 delete x from the list $T[h(x.key)]$