Let $U = \{u_1, \ldots, u_n\}$ be a finite set of Boolean variables. If u is a variable, then u and $\bar{u}$ are *literals*. A *clause* over U is a set of literals over U that are joined together by the Boolean *or*. A statement in *conjunctive normal form* is a finite collection of clauses joined together by the Boolean *and*. A *truth assignment* for U is a function $t: U \to \{T, F\}$. If $t(u) = T$, then u is "true" under that assignment; otherwise, u is "false". A literal u is true (false) under an assignment if and only if the literal $\bar{u}$ is false (true). A statement in conjunctive normal form is *satisfiable* if there is a truth assignment that simultaneously makes each clause true and so makes the entire statement true under the usual interpretation of Boolean *and* ($\wedge$) and *or* ($\vee$). The *satisfiability problem* is specified as follows: given a set U of variables and a conjunctive normal form statement over U, is that statement satisfiable? An equivalent formulation is to consider the set of all conjunctive normal form statements that are satisfiable and to ask if a given statement is in that set.

Consider the following examples over the set $U = \{p, q, r, s\}$:

(i) $(p \vee q \vee r) \wedge (\bar{p} \vee \bar{q} \vee \bar{s}) \wedge (p \vee \bar{r} \vee s) \wedge (q \vee r \vee s)$;

(ii) $(p \vee \bar{p} \vee q) \wedge (p \vee r \vee \bar{r}) \vee (q \vee r \vee s \vee \bar{s})$;

(iii) $(p \vee q) \wedge (\bar{p} \vee q) \wedge (p \vee \bar{q}) \wedge (\bar{p} \vee \bar{q})$. Example (i) is satisfiable by means of the truth assignment $f(p) = T, f(q) = T, f(s) = F, f(r) = F$. Example (ii) is satisfiable under every truth assignment–it is a tautology. Example (iii) is not satisfiable under any truth assignment–it is a contradiction.

The problem of determining whether a given conjunctive normal form statement is or is not satisfiable is a "decision" problem: one must decide whether the given statement is a member of the set of all satisfiable statements.

undirected graph

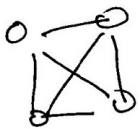
k - clique problem ($k \geq 2$)

input: an undirected graph
$G = (V, E)$

output: yes if $\exists$ a clique k in G of size k

no otherwise

3-clique



4-clique.

k - clique : a complete graph that consists of k nodes

3 CNF SAT reduces to k-clique problem

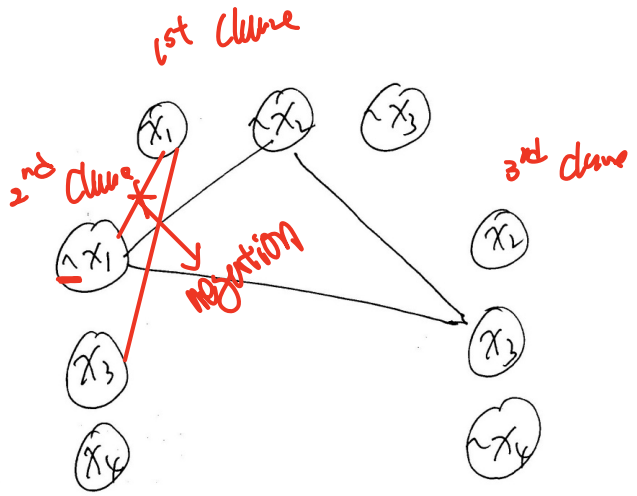
$f = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge$
$(\neg x_1 \vee x_3 \vee x_4) \wedge$
$(x_2 \vee x_3 \vee \neg x_4)$

Literals ⇒ nodes

edges exist between a pair of nodes a, b if

① each belongs to a different clause

② not negating each other



Computational complexity measures how much time and/or memory space is needed as a function of the input size. Let TIME[$t(n)$] be the set of problems that can be solved by algorithms that perform at most $O(t(n))$ steps for inputs of size n. The complexity class polynomial time (P) is the set of problems that are solvable in time at most some polynomial in n.

$$\mathrm{P} = \bigcup_{k=1}^{\infty} \mathrm{TIME}[n^k]$$

Some important computational problems appear to require more than polynomial time. An interesting class of such problems is contained in nondeterministic polynomial time (NP). A nondeterministic computation is one that may make arbitrary choices as it works. If any of these choices leads to an accept state, then we say the input is accepted. As an example, let us consider the three-colorability problem. A nondeterministic algorithm traverses the input graph, arbitrarily assigning to each vertex a color: red, yellow, or blue. Then it checks whether each edge joins vertices of different colors. If so, it accepts.

(3^n) possibilities

one characterization proved by
Fagin Theorem

there are three unary relations—Red (R), Yellow (Y), and Blue (B)—defined on the universe of vertices. It goes on to say that every vertex has some color and no two adjacent vertices have the same color.

$$\Psi_3 \equiv (\exists R)(\exists Y)(\exists B)(\forall x)$$
$$[(R(x) \vee Y(x) \vee B(x)) \wedge (\forall y)(E(x, y) \rightarrow$$
$$\neg (R(x) \wedge R(y)) \wedge \neg (Y(x) \wedge Y(y))$$
$$\wedge \neg (B(x) \wedge B(y)))]$$

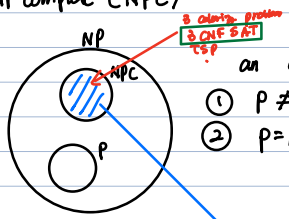
A second-order existential formula (SO $\exists$) begins with second-order existential quantifiers and is followed by a first-order formula. As an

Descriptive complexity began with the following theorem of Ronald Fagin. Fagin's theorem says that NP is equal to the set of problems describable in second-order, existential logic. Observe that Fagin's theorem characterizes the complexity class NP purely by logic, with no mention of machines or time.

Theorem 1 [1]. *A set of structures T is in* NP *iff there exists a second-order existential formula, Φ such that $T = \{\mathcal{A} | \mathcal{A} \models \Phi\}$.*

ch 34

NP complete (NPC)



an assumption

① $P \neq NP$

② $P = NP$

"hardest" in NP

| if any one of problems in NPC can be solvable efficiently
| then $P = NP$

Notion called "NP-hard"

Definition

A problem $x \in NPC$

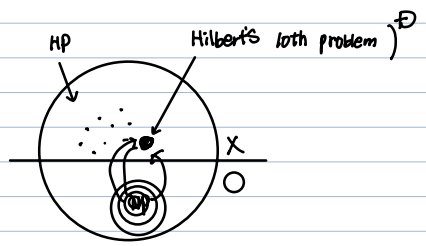
① $x \in NP$ reducible

NP-Hard ⟶ ② $\forall_{Y \in NP}\ Y \leq_p x$

ex1) Sudoku $\leq$ G. coloring

ex2) and MBGM $\leq$ MFP

ex3) 3CNFSAT $\leq$ k-diq



Hilbert's 10th problem

input: a Diophantine equation (polynomial Equation in which every coeff is an integer)

Output: yes if has integral (integer) root

No, otherwise

Ex | $3x^2 + 4xy + z = 6$

$f(x,y,z) = 3x^2 + 4xy + z - 6$

$x^2 + 2x + 1 = 0$

$(x+1)^2 = 0 \quad x = -1$ → Return Yes

2 property

NPC → ①

②

$p = 0$

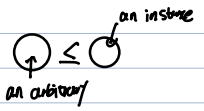
{ P | P has integral roots }

$\leq_p$: polynomial time reducability has "transitivity"

Fact if $A \leq_p B$, $B \leq_p C$ then $A \leq_p C$

For $\forall_{Y \in NP}$ $Y \leq_p D$

Fact $\forall_{Y \in NP}$ $Y \leq_p$ 3CNF-SAT



3CNF-SAT $\leq_p D$

Any variable $x \longrightarrow x$

$\wedge x \longrightarrow 1-x$

$\vee \longrightarrow \cdot$

$\wedge \longrightarrow +$

$(\ ) \longrightarrow (\ )^2$

$= 0$

$Y = (x \vee y) \wedge (x \vee \sim y) \wedge (\sim x \vee \sim y)$ 2 CNF

$Y = (x \vee y \vee \sim x) \wedge (x \vee \sim y \vee \sim x) \wedge (\sim x \vee \sim y \vee z)$ 3CNF-SAT $\in$ NPC

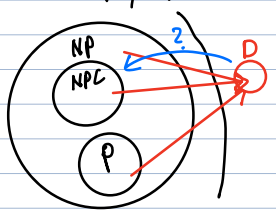
$x = T$, $y = F$, $z = T$ $\forall_Y \in NP$

$\Downarrow$

$(x \cdot y \cdot (1-x))^2 + (x \cdot 1-y \cdot 1-x)^2 + ((1-z) \cdot (1-y) \cdot z)^2 = 0$

$x = 1$ (0) $y = 1$ (0) $z = 1$ or 0 (0) $\rightarrow$ one possibility out of 8

unlike NPC, NP-'hard' can get out of NP



All NPC is NP-hard

NP-hard $= \mathcal{I}$

$\forall_{Y \in NP} \leq_p \mathcal{I}$

Chapter 34

NP Complete problems – “hardest” problems in NP

ment of the theory of NP-completeness. Loosely speaking, this theory identifies a set of computational problems that are as hard as NP. That is, the fate of the P versus NP question lies with each of these problems: If any of these problems is easy to solve then so are all problems in NP. Thus, showing that a problem is NP-complete provides evidence to its intractability (assuming, of course, P different than NP). Indeed, demonstrating the NP-completeness of a problem not only the conjectured intractability of a problem but rather also its “richness,” in the sense that the problem is rich enough to encode any other problem in NP. The use of the term “encoding” is justified by the exact meaning of NP-completeness, which in turn establishes relations between different computational problems (without referring to their absolute complexity).

Diophantine – not NP-Complete (unsolvable)

$Diophantine = \{p \mid p \text{ is a polynomial with an integral root}\}$
The mapping reduction 3-SAT $\leq_P$ *Diophantine* is achieved as follows.
Consider a 3-CNF formula φ:

- Each literal of φ, x, maps to integer variable x.
- Each literal $\overline{x}$ maps to expression $(1 - x)$.
- Each $\vee$ in a clause maps to . (times).
- Each clause is mapped as above, and then *squared.*
- Each $\wedge$ maps to $+$.
- The resulting equation is set to 0.
- Example: map

$$\varphi = (x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee \overline{y})$$

to

$$E = (x.y)^2 + (x.(1-y))^2 + ((1-x).(1-y))^2 = 0.$$

CH4

Recursive alg
D & C
$T(n) = \square$

① Recursion Tree method : merge sort
② Substitution method : "Guess and test" method QS
③ master's theorem

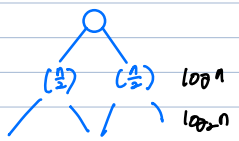
Substitution method

1. Guess a solution
2. Verify that solution is correct

Ex $T(n) = b$ for $(n < 2)$

$T(n) = 2T(\frac{n}{2}) + bn\lg n$ $(n \geq 2)$ — division, combine

a recurrence (eq)



cf $T(n) = 2T(\frac{n}{2}) + c \cdot n$ $\Rightarrow$ MS
$T(n) = O(n \lg n)$

Assume that $T(n) \leq c \cdot n \lg n$ | Guess ~ $T(\frac{n}{2}) \leq c \cdot \frac{n}{2} \lg \frac{n}{2}$

Verification

$T(\frac{n}{2}) \rightarrow \boxed{T(n)}$?

$T(n) = 2T(\frac{n}{2}) + bn\lg n$

$T(n) \leq 2 \cdot c \cdot \frac{n}{2} \cdot \lg \frac{n}{2} + bn\lg n$ — want to show this

$T(n) \leq c \cdot n(\lg n - 1) + bn\lg n \leq c \cdot n \lg n$

$= c(\lg n - 1) + bn\lg n \leq c \cdot n\lg n$

$= c\lg n - c + bn\lg n \leq c\lg n$

$= bn\lg n \leq c$

✗ wrong

$T(\frac{n}{2}) \rightarrow \boxed{T(n)}$?

Assume that $T(n) = O(n \lg^2 n)$

$T(\frac{n}{2}) \leq c \cdot \frac{n}{2} \cdot \lg^2 \frac{n}{2}$

$T(n) = 2 \cdot T(\frac{n}{2}) + bn\lg n$ — want to show this

$\leq 2 \cdot \frac{c}{2} \lg^2 \frac{n}{2} + bn\lg n \leq c \cdot n \lg^2 n$

$\lg^2 \frac{n}{2} c + bn\lg n \leq c\lg^2 n$

$c(\lg \frac{n}{2})^2 + bn\lg n \leq c \cdot (\lg n)^2$

$c(\lg n - 1)^2 + bn\lg n \leq c \cdot (\lg n)^2$

$c(\lg^2 n - 2\lg n + 1) + bn\lg n \leq c\lg^2 n$

$-2c\lg n + c + bn\lg n \leq 0$

$-2c\lg n + c + bn\lg n \leq 0$

$c \leq (2c - b)\lg n$ — $b \leq 2c$, $2c - b > 0$

NO matter this — as n increase

Chapter 4 recurrences – substitution method

1. Guess a solution [this part requires some experiences, exercises, etc]
2. Prove the solution is correct

case $T(n) = b$ *for* $n < 2$*):*

$$T(n) = 2T(n/2) + bn \log n.$$

This looks very similar to the recurrence relation for the merge-sort routine, so we might make as our first guess the following:

$$\textit{First guess: } T(n) \leq cn \log n,$$

case $T(n) = b$ for $n < 2$):

$$T(n) = 2T(n/2) + bn\log n.$$

This looks very similar to the recurrence relation for the merge-sort routine, so we might make as our first guess the following:

First guess: $T(n) \leq cn\log n,$

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n. \end{aligned}$$

not working!

case $T(n) = b$ for $n < 2$):

$$T(n) = 2T(n/2) + bn\log n.$$

This looks very similar to the recurrence relation for the merge-sort routine, so we might make as our first guess the following:

$$\textit{First guess: } T(n) \leq cn\log n,$$

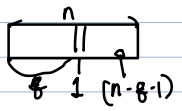
$$\textit{Better guess: } T(n) \leq cn\log^2 n,$$

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log^2 n - 2\log n + 1) + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n, \end{aligned}$$

provided $c \geq b$. Thus, we have shown that $T(n)$ is indeed $O(n\log^2 n)$.

7.4.2 p.6

max (Quick sort)



$T(n) = T(n-q-1) + T(q) + c \cdot n$

$\max_{0 \le q \le n-1} (T(n-q-1) + T(q) + c \cdot n) \quad O(n^2)$

$0 \le q \le n-1$

$T(n) \le c \cdot n^2$ — Guess

$n-q-1 < n$, $q < n$ so it's okay

$T(n) \le c \cdot (n-q-1)^2 + c \cdot q^2 + cn$

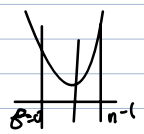
$0 \le q \le n-1$

$c \cdot (n^2 + q^2 + 1 - 2nq + 2q - 2n) + cq^2 + cn$

$2cq^2 + \ldots$

$T(n)' = \ldots = 0$

has a longest value when $q = 0$ or $q = n-1$



Worst time complexity of Quicksort – section 7.4

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

where the parameter q ranges from 0 to $n-1$ because the procedure PARTITION produces two subproblems with total size $n-1$. We guess that $T(n) \le cn^2$ for some constant c. Substituting this guess into recurrence (7.1), we obtain

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

The expression $q^2 + (n-q-1)^2$ achieves a maximum over the parameter's range $0 \le q \le n-1$ at either endpoint, as can be seen since the second derivative of the expression with respect to q is positive (see Exercise 7.4-3). This observation gives us the bound $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

master method – section 4.3

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

$$T(n) = 9T(n/3) + n\ .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

$$T(n) = 3T(n/4) + n \lg n ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

$$T(n) = 2T(n/2) + n \lg n ,$$

even though it has the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. It might seem that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio

chapter 11 hash table – an example of a dictionary

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler for a computer language maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$.

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)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

Each of these operations is fast: only $O(1)$ time is required.

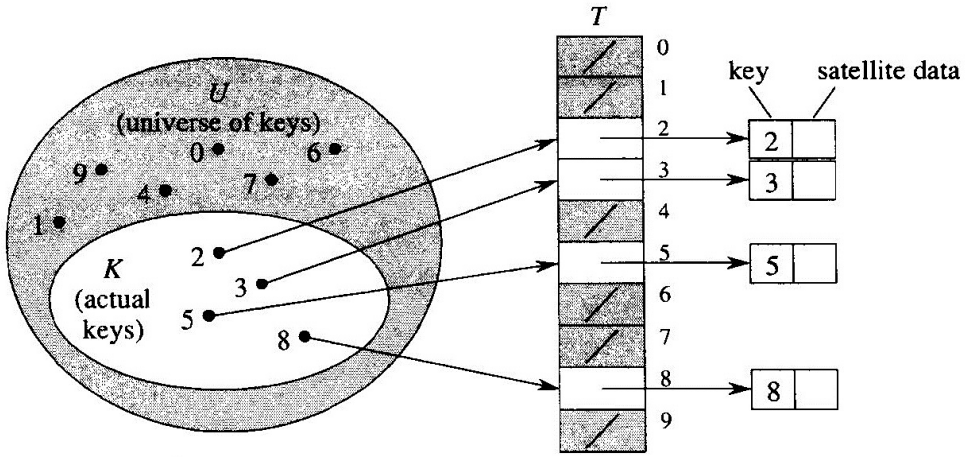


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

Direct address table – space / time trade-off

Time – O(1) : very fast (worst case complexity)
Space – impractical

 a hash table

Time – O(1) : expected time
Space – “more” efficient than direct address table

h : U -> {0,1,…,m-1}

Collision – two different keys may hash to the same slot

The difficulty with direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirements can be reduced to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $O(1)$ time. The only catch is that this bound is for the *average time*, whereas for direct addressing it holds for the *worst-case time*.

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*** $T[0..m-1]$:

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

We say that an element with key k ***hashes*** to slot $h(k)$; we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The point of

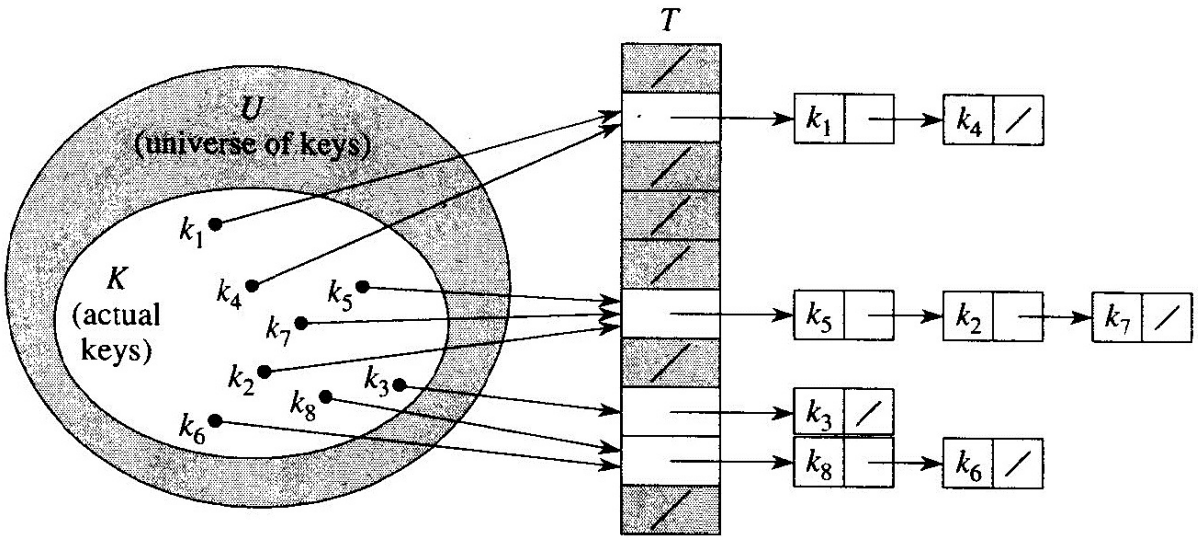


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Collision resolution methods

(1) Chaining – maintain a linked list – insertion / deletion O(1) time, worst complexity

CHAINED-HASH-INSERT(T, x)
insert x at the head of list $T[h(key[x])]$

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

CHAINED-HASH-DELETE(T, x)
delete x from the list $T[h(key[x])]$

(2) Open addressing – find a next available slot (computing a probe sequence)

Search – chaining [expected time – O(1) time]

Theorem 11.1

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

load factor = (number of keys) / (number of slots)

simple uniform hashing – each key is equally likely to be hashed to any one of m slots AND for any two keys a and b, where a hashes to is independent of where b hashes to

11.3 hash function

division method

$h(k) = k \bmod m$

m should not be a power of 2.

Ex. 1100 = 12
10100 = 20
11100 = 28
101100 = ~~48~~ 44

$h(k) = k \bmod 8$

12 mod 8 = 4
20 mod 8 = 4
28 mod 8 = 4
~~48~~ 44 mod 8 = 04

multiplication method

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

$kA \bmod 1$

$= kA - \lfloor kA \rfloor$

Ex. $k = 12$, $A = 0.6$

$0 < A < 1$

$7.2 - 7$

$= 0.2$

In ***open addressing***, all elements are stored in 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 the desired element is found or it is clear that the element is not in the table. There are no lists and no elements stored outside the table, as there are in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; the load factor α can never exceed 1.

hash function is extended

$h: U \times \{0, 1, 2, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

$h(\uparrow, \uparrow)$

key, probe #

$(h\langle k, 0\rangle, h\langle k, 1\rangle, \ldots, h\langle k, m-1\rangle)$

~ probe sequence

insert / search – O(n)

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

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

deletion should be done carefully

3 ways to compute probe sequences

In our analysis, we make the assumption of ***uniform hashing***: we assume that each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence. Uniform hashing generalizes the notion of simple uniform

linear probing, quadratic probing, double hashing

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

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m-1$. Given key k, the first slot probed is $T[h'(k)]$, i.e., the

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m , \tag{11.5}$$

where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m - 1$. The initial position probed is $T[h'(k)]$; later positions probed

Double hashing is 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

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions. The initial probe is to position

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. (See Exercise 11.4-3.) A convenient way to ensure this

search – expected time [very fast!]

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Theorem 11.8

Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.