Week 8
Hash tables

General remarks

- We discuss data structures in general.
- Followed by basic techniques of hash tables.

Reading from CLRS for week 8

1. The introduction to Part III on using sets on a computer.
2. Chapter 11, Sections 11.1, 11.2, 11.3.

Sets

The most fundamental mathematical notion is that of a set:

- We have the possibility to determine the elements of a set.
- And we can form sets by some set-defining property.

Now to bring the eternal and infinite world of mathematics to a computer, we need to take care of

- construction and destruction of objects
- handling of objects by value or reference
- naming (interfaces)
- order issues (sets are unordered, but in computation there is always order).

For this, CLRS uses the ADT of "dynamic sets", consisting of $1 + 2 + 4 = 7$ operations.

Recall: ADT means "abstract data type" — values (like "sets") and how to operate with them.

Elements of a dynamic sets

A "dynamic set" might contain

- **pointers** (or iterators) to objects,
- or the **objects** themselves (in Java this can be only integers and other primitive types, in C++ this is possible for every type of object).

Whatever the objects in a dynamic set are, *access* (especially for changing them) is only possible via a pointer (or iterator).

- For *insertion* into a dynamic set, we must be given the object itself, and we obtain back the pointer (iterator, handle) to the copy of that object in the dynamic set.
- For *deletion* from a dynamic set, we typically have the pointer of an object (already) in the dynamic set, and we want to delete that (very specific) object.

For *searching*, we typically have only given some "key information", and we want to search for some element in the dynamic set, which fits this (key) information.

Keys

Besides objects (which become "elements" once they are in the set) and pointers, CLRS uses the notion of a **key** to identify an object:

> If the object is for example a record of personal attributes, then the name or some form of ID can be used as a key.

Often they keys are used for sorting.

> For example for that database of personal attributes, we might sort it according to alphabetical sorting of names.

Dynamic sets: elementship and search

With sets S we can "ask" whether "$x \in S$" is true. This is the most basic set-operation, and the equivalent for dynamic sets is the operation

- $\mathrm{SEARCH}(S, k)$ for key k and dynamic set S, returning either a pointer (iterator) to an object in S with key k, or NIL if there is no such object.

We require the ability to extract the key from an object in S, and to compare keys for equality.

1. Storing S via an array (or a list), SEARCH can be performed in $O(|S|)$ time (that is, *linear time*) by simple sequential search.
2. To do faster than this, typically in time $O(\log(|S|))$ (*logarithmic time*), under various circumstances, is a major aim of data structures for dynamic sets.

Dynamic sets: modifying operations

With the following operations we can build and change dynamic sets:

- $\mathrm{INSERT}(S, x)$ inserts an object into dynamic set S, where x is either a pointer or the object itself.
- $\mathrm{DELETE}(S, x)$ deletes an object from dynamic set S, where here x is a pointer (iterator) into S.

Note that the "x" in DELETE is of a different nature than the "x" in INSERT: It is a pointer (iterator!) *into* the (dynamic) set (and thus these two x are of different type).

The most important application of INSERT is for creating a dynamic set:

To create a set S of size n, call $\mathrm{INSERT}(S, -)$ n-times.

We always have INSERT, while we might not have DELETE.

Dynamic sets: using order

Often it is assumed that a *linear order* is given on the keys:

- So besides "$k == k'$?"
- we now can ask "$k \leq k'$?".

In practice using strict orders "$<$" is more common, however this creates some (necessary) technical complications, which in this module we won't be much concerned about (we discuss issues when the need arises).

(These complications have to do with the notion of "equality", since "$<$" includes "not equal". Considering Java, recall that there (lacking operator overloading and lacking the ability to distinguish between "object" and "pointer") you have two operations for checking "equality": "==" for object-identity and "`.equals()`" for object-equality, and now we needed *appropriate* object-equality (*consistent with the algorithms*).)

Dynamic sets: four further operations

Given a linear order on the objects (to be put into the set), we have the following four additional operations:

- $\mathrm{MINIMUM}(S)$ returns a pointer (iterator) to the element with the smallest key
- $\mathrm{MAXIMUM}(S)$ returns a pointer (iterator) to the element with the largest key
- $\mathrm{SUCCESSOR}(S, x)$, where x is a pointer (iterator) into S, returns the next element in S w.r.t. the order on the keys
- $\mathrm{PREDECESSOR}(S, x)$, where x is a pointer (iterator) into S, returns the previous element in S.

Operations for computing successors and predecessors can fail (there is no successor resp. predecessor iff we are already at the end resp. beginning), and in such cases we return NIL.

Using sorting algorithms: static case

Dynamic sets can be realised using sorting, where we have to assume a linear order on the keys.

If the set S with n elements is to be built only once, at the beginning, from a sequence of elements, then storing the elements in an array and sorting them, using for example MERGE-SORT with time complexity $O(n \cdot \log n)$, is a good option:

- SEARCH then takes time $O(\log n)$ (using binary search)
- while each of MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR takes constant time.

However we are concerned here with the *dynamic* case, where insertions and deletions are used in unpredictable ways. If we not assume that building the set is done (once and for all) at the beginning, but we want to have insertion and deletion, then the case is much less favourable.

Using sorting algorithms: dynamic case

- Now INSERTION and DELETION take time $O(n)$, since at the given insertion/deletion index, shifting of the other elements appropriately is needed;
- while the five other (non-modifying) operations still take logarithmic resp. constant time.

For practical applications, the linear complexity of insertion and deletion is not acceptable. And we also see that most of the intelligence of sophisticated search algorithms is blown out of the window here, and only some form of INSERTION-SORT (in combination with binary search) survived.

It could be said that data structures for dynamic sets try to introduce some of the intelligent methods into the dynamic framework, making insertion and deletion more efficient (necessarily at the cost of making the other operations (somewhat) less efficient).

Special cases of dynamic sets

Often not all of the operations for dynamic sets are needed, opening up the possibilities of specialised and more efficient implementations. Three important cases are as follows:

Buffer Only INSERTION, SHOW-SELECTED-ELEMENT (like TOP) and DELETE-SELECTED-ELEMENT; special cases are "stacks" and "queues".

Priority queue Only INSERTION, MINIMUM resp. MAXIMUM (for min- resp. max-priority queues) and DELETE-MIN resp. DELETE-MAX (i.e., we have special queues, where the selected element is given by a linear order on the keys).

Dictionary Only INSERTION, DELETION and SEARCH (i.e., no order-requirements).

In other words:

Special cases of dynamic sets (cont.)

- Buffers (stacks, queues, and priority queues) have the notion of a "selected element", which they can show and delete (but general deletion is not possible); searching in the general sense is not possible.
- Dictionaries can perform arbitrary deletions and searches, but they have no order on the elements (and thus also no "special elements").

Insertion is always possible (given that there is enough space).

Dictionaries

We are now concentrating on *dictionaries*, with their three operations:

1. INSERT(x) (input is pointer x to element to be inserted)
2. SEARCH(k) (input is key k, returns a pointer)
3. DELETE(x) (input is pointer x to element to be deleted)

Via binary search trees, which generalise binary search, we get such a dictionary:

- We actually get all seven operations for dynamic sets (assuming a linear order on the keys) from binary search trees, all in time logarithmic in the size of the set.
- *Hashing* is a technique **specialised** for dictionaries (not supporting the four order-related operations).
- It usually is faster for dictionaries — goal is constant time.

Dictionaries

We are now concentrating on *dictionaries*, with their three operations:

1. INSERT(x) (input is pointer x to element to be inserted)
2. SEARCH(k) (input is key k, returns a pointer)
3. DELETE(x) (input is pointer x to element to be deleted)

Via binary search trees, which generalise binary search, we get such a dictionary:

- We actually get all seven operations for dynamic sets (assuming a linear order on the keys) from binary search trees, all in time logarithmic in the size of the set.
- *Hashing* is a technique **specialised** for dictionaries (not supporting the four order-related operations).
- It usually is faster for dictionaries — goal is constant time.

Applications of dictionaries

A standard application is for example in a compiler:

1. We have many different "identifiers", for variables, functions and classes for example.
2. For such an identifier, for example the class-name `BreadthFirst`, a lot of information needs to be stored.
3. The dictionary now translates the identifier, a character sequence e.g. "BreadthFirst", into a pointer to the data associated with this class.

Indeed dictionaries are everywhere — it's always there when you have to associate data to some "keys"!

Can you think of some examples?

The fastest implementation: keys as array indices

With binary search trees we can achieve worst-case *logarithmic time* for the three dictionary operations.

We now want **constant time** for the three operations — on average, and if we provide enough space.

The basic idea for this is to use arrays:

If we can use the keys as array indices, we are done (mostly).

Hashing is the process of handling arbitrary *key-spaces* $\mathbb{K}$ as if they were array indices.

The fastest implementation: keys as array indices

With binary search trees we can achieve worst-case *logarithmic time* for the three dictionary operations.

We now want **constant time** for the three operations — on average, and if we provide enough space.

The basic idea for this is to use arrays:

If we can use the keys as array indices,
we are done (mostly).

Hashing is the process of handling arbitrary *key-spaces* $\mathbb{K}$ as if they were array indices.

The fastest implementation: keys as array indices

With binary search trees we can achieve worst-case *logarithmic time* for the three dictionary operations.

We now want **constant time** for the three operations — on average, and if we provide enough space.

The basic idea for this is to use arrays:

If we can use the keys as array indices,
we are done (mostly).

Hashing is the process of handling arbitrary *key-spaces* $\mathbb{K}$ as if they were array indices.

The basic idea of hashing

We consider the key-space $\mathbb{K}$ (an arbitrary set), and we can use an array of length m. The basic idea is to use a hash function

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

which translates key-values into indices.

- The simplest case is when h is *injective*, i.e., maps different keys to different indices.
- Injective hash functions are called **perfect**.
- For that to be possible we need $|\mathbb{K}| \leq m$, i.e., there are at most m different keys.
- Otherwise we have to handle **collisions**, that is, different keys map to the same index, i.e., there are $k, k' \in \mathbb{K}$, $k \neq k'$, with $h(k) = h(k')$.

The simplest case of hashing

The simplest case of hashing is when for h we can use the identity, that is,

- the keys are natural numbers ≥ 0, i.e., $\mathbb{K} \subset \{0, 1, 2, \ldots\}$;
- within a feasible range, i.e., $m = \max(\mathbb{K}) + 1$ is not too large (note that in general $\max(\mathbb{K})$, i.e., the maximum of possible indices, is much larger than $|\mathbb{K}|$).

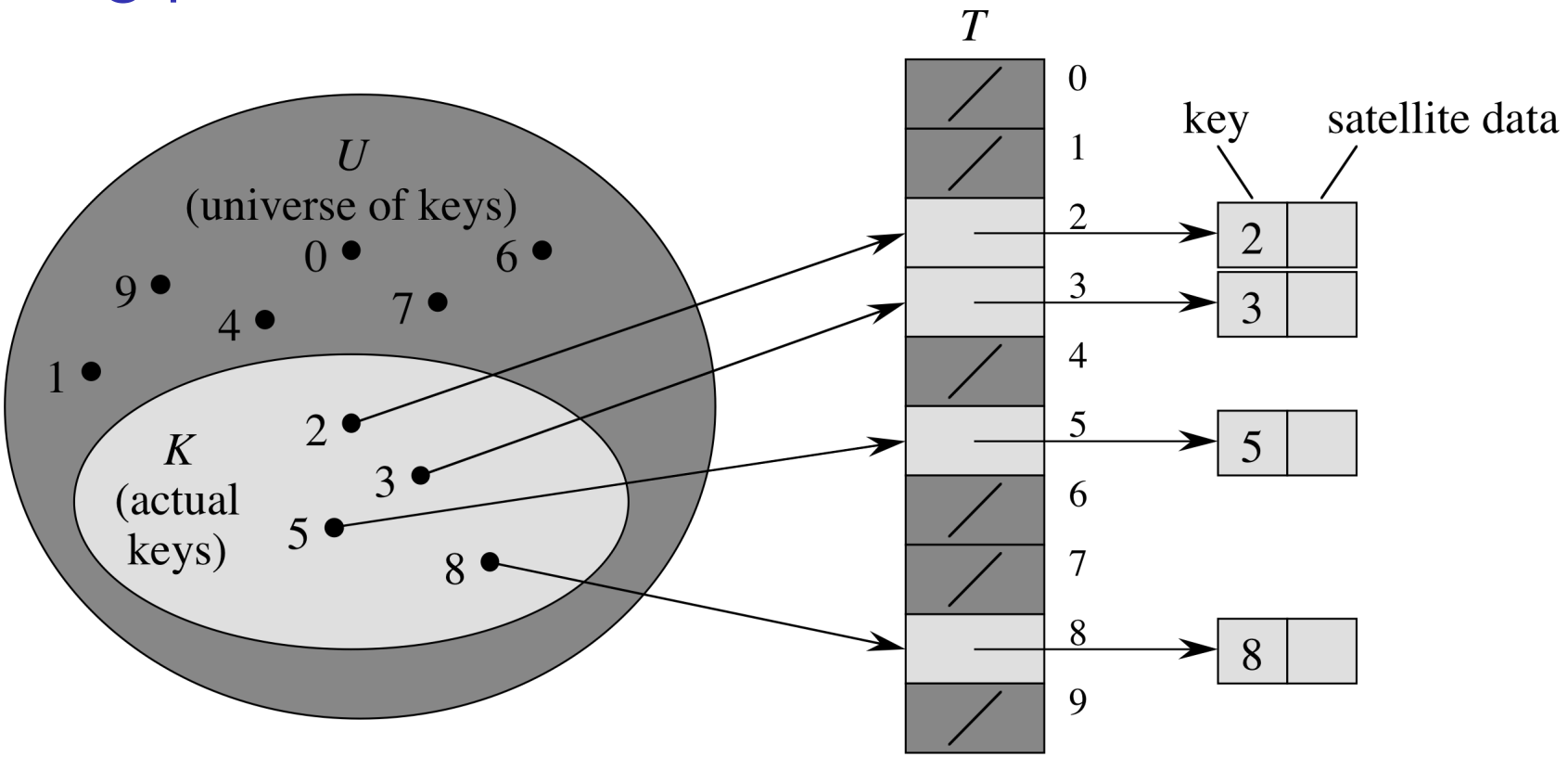
The array is called a **direct-address (hash) table**; the book uses the letter T (for "table").

Key or not

A basic problem is how to show that an element is *not* there:

1. Conceptually simplest is to use pointers, where then the NIL-pointer shows that the element is not there.
2. Alternatively we can use a special "singular" key-value.
3. Or for example an additional boolean array.

Using pointers



How to implement a dynamic set by a direct-address table T: The keys of the key-space $\mathbb{K} = \{0, 1, \ldots, 9\} = U$ are used as indices in the table. The empty slots in the table contain NIL.

The basics of the implementation

```
Search(T,k)
1   return T[k]

Insert(T,x)
1   T[x.key] = x

Delete(T,x)
1   T[x.key] = NIL
```

Examples where some simple translation is needed

If $\mathbb{K}$ is small, then we typically can find a nice *injective* (i.e., perfect) hash function h, for example:

- The keys are integers in a known range — just move them.
- The keys are (arbitrary) images with 20 pixels — use binary encoding.
- Do you know other examples?

In principle we can always use an injective hash function:

- If we have enough memory, then this is very fast.
- However in practice this is often not feasible, for example for strings, and so we need to handle "collisions", that is, cases where the hash function yields the same index for different keys.

Examples where some simple translation is needed

If $\mathbb{K}$ is small, then we typically can find a nice *injective* (i.e., perfect) hash function h, for example:

- The keys are integers in a known range — just move them.
- The keys are (arbitrary) images with 20 pixels — use binary encoding.
- Do you know other examples?

In principle we can always use an injective hash function:

- If we have enough memory, then this is very fast.
- However in practice this is often not feasible, for example for strings, and so we need to handle "collisions", that is, cases where the hash function yields the same index for different keys.

General hash tables

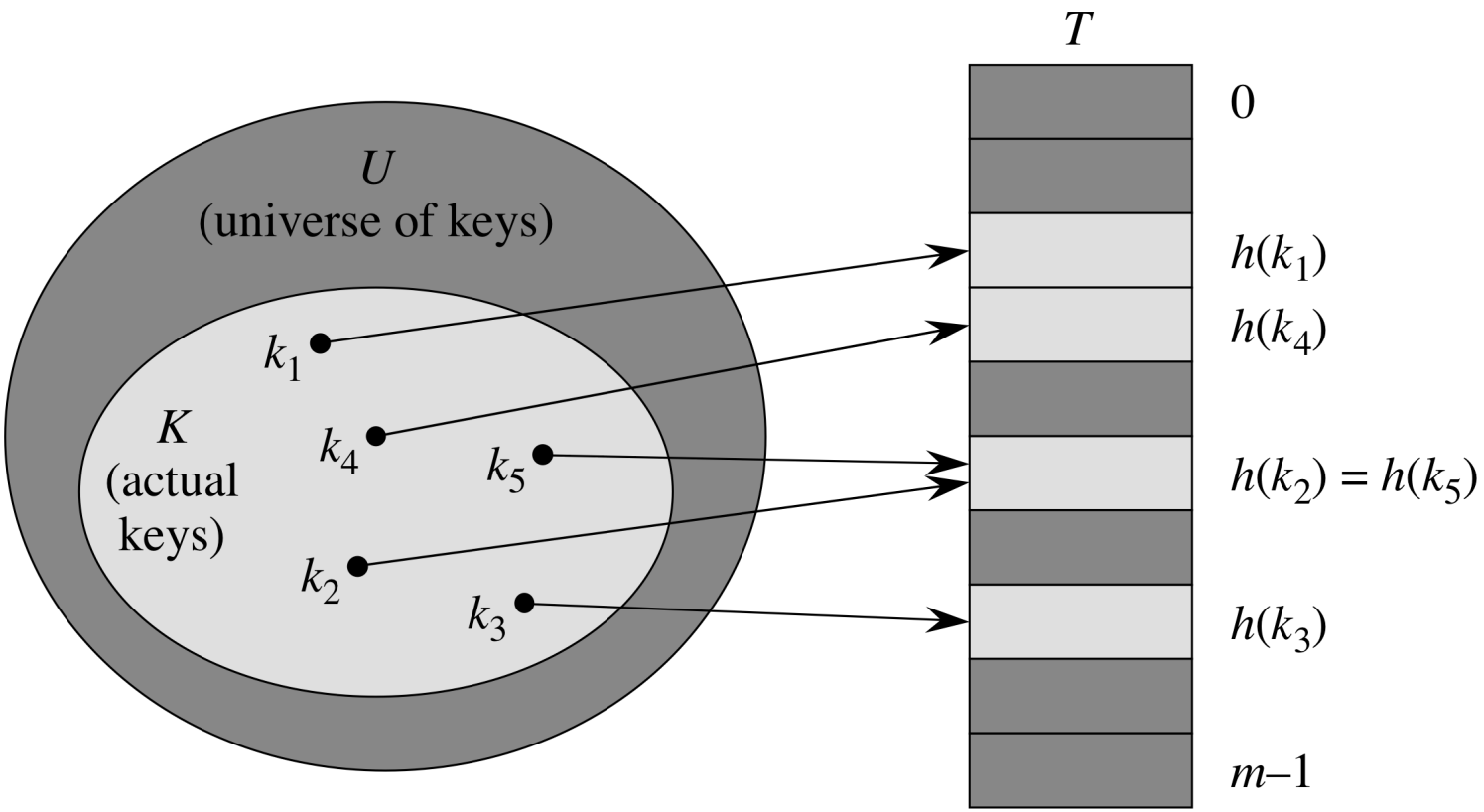
As said already, the general idea of "hashing" is to use a "hash function"

$$h : \mathbb{K} \to \{0, \ldots, m-1\}$$

(here using "$\mathbb{K}$" instead of "U" as in the book).

- m is the **size** of the hash table.
- An element with key k **hashes** to **slot** $h(k)$.
- $h(k)$ is the **hash value** of key k.

General hash tables (cont.)



We see that we have a collision for keys k_2 and k_5.

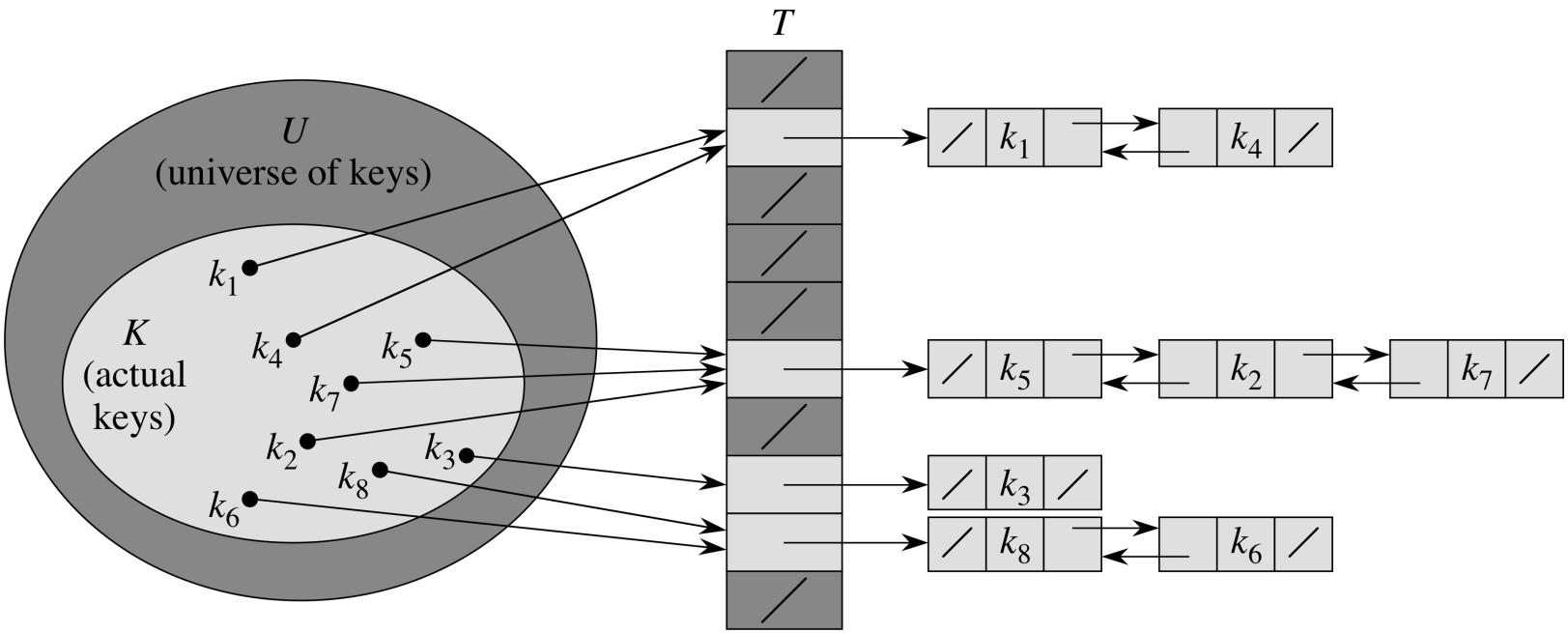
How to handle collisions?

In general, $|\mathbb{K}|$ is much bigger than m:

- The hash function should be as "random" as possible.
- That is, it should hash "unpredictably".
- That is, it should be independent of our choices of keys.
- That is, we should get as few collisions as possible.

However we have to handle collisions nevertheless!

Using linked lists



Collision resolution by "chaining"

Put all elements that hash to the same slot
into a linked list.

- 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.
- Singly linked lists can be used if we do not want to delete elements.

The basics of the implementation

Search(T,k)

return result of search for key k in list T[h(k)]

Run time: linear in the length of the list of elements in slot $h(k)$.

Insert(T,x)

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

Run time: constant

Delete(T,x)

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

Run time: constant

The basics of the implementation

Search(T,k)

return result of search for key k in list T[h(k)]

Run time: linear in the length of the list of elements in slot $h(k)$.

Insert(T,x)

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

Run time: constant

Delete(T,x)

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

Run time: constant

The basics of the implementation

Search(T,k)

return result of search for key k in list T[h(k)]

Run time: linear in the length of the list of elements in slot $h(k)$.

Insert(T,x)

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

Run time: constant

Delete(T,x)

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

Run time: constant

The basics of the implementation

Search(T,k)

return result of search for key k in list T[h(k)]

Run time: linear in the length of the list of elements in slot $h(k)$.

Insert(T,x)

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

Run time: constant

Delete(T,x)

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

Run time: constant

Analysis

What is the time-complexity of SEARCH ?

Worst-case is when all n keys hash to the same slot, and we get just a single list of length n; so the worst-case time is $\Theta(n)$, plus time to compute the hash function.

How can we treat **average-case performance** ?

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the m slots.
- Analysis is in terms of the **load factor** $\alpha := \frac{n}{m}$.
- We assume computation of the hash function takes constant time.

Theorem 1

Search takes expected time $\Theta(1 + \alpha)$.

Analysis

What is the time-complexity of SEARCH ?

Worst-case is when all n keys hash to the same slot, and we get just a single list of length n; so the worst-case time is $\Theta(n)$, plus time to compute the hash function.

How can we treat **average-case performance** ?

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the m slots.
- Analysis is in terms of the **load factor** $\alpha := \frac{n}{m}$.
- We assume computation of the hash function takes constant time.

Theorem 1

Search takes expected time $\Theta(1 + \alpha)$.

Analysis

What is the time-complexity of SEARCH ?

Worst-case is when all n keys hash to the same slot, and we get just a single list of length n; so the worst-case time is $\Theta(n)$, plus time to compute the hash function.

How can we treat **average-case performance** ?

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the m slots.
- Analysis is in terms of the **load factor** $\alpha := \frac{n}{m}$.
- We assume computation of the hash function takes constant time.

Theorem 1

Search takes expected time $\Theta(1 + \alpha)$.

What makes a good hash function?

What are the conditions for a "good" hash function

$$h : \mathbb{K} \to \{0, \ldots, m-1\}?$$

1. By definition, h must be a *function*, that is, for the same key it must return always the same hash value.
2. Now, ideally, the hash function satisfies the assumption of **simple uniform hashing** — is this possible?

Actually, "simple uniform hashing" is an assumption on the interplay between the hash function and the probability distribution that keys are drawn from:

In practice we might not know
the probability distribution of the keys —

since we have a hash *function*, in the worst-case we can always pick the keys to hash into the same slot!

What makes a good hash function?

What are the conditions for a "good" hash function

$$h : \mathbb{K} \to \{0, \ldots, m-1\}?$$

1. By definition, h must be a *function*, that is, for the same key it must return always the same hash value.
2. Now, ideally, the hash function satisfies the assumption of **simple uniform hashing** — is this possible?

Actually, "simple uniform hashing" is an assumption on the interplay between the hash function and the probability distribution that keys are drawn from:

In practice we might not know
the probability distribution of the keys —

since we have a hash *function*, in the worst-case we can always pick the keys to hash into the same slot!

Using knowledge about the keys

Qualitative information about the distribution of keys may be useful in the design process of a hash function:

- In the compiler-example, closely related symbols, e.g. “pt” and “pts” (“pointer” and “pointers”), often occur in the same program.
- So a good hash function would minimise the chance that such variants hash to the same slot.

The general guideline is to compute the hash values in a way that is independent of any patterns in the data.

Various general methods for constructing hash functions have been investigated. A glimpse into that theory follows.

Keys as natural numbers

To obtain a general overview, we assume that the keys are natural numbers.

Thus for us the first step in designing a hash-function for a given use-case is to interprete the keys as natural numbers:

- A general tool is to use a "radix system".
- Recall that in a radix-system with base $b \geq 2$ we have digits $0, \ldots, b-1$, and a sequence $d_1, \ldots, d_n$ of digits denotes the natural number $\sum_{i=1}^{n} d_i \cdot b^{n-i}$.
- So for example $(1,1,0)_2$ yields $0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 6$.
- And $(2,1,2)_3$ yields $2 \cdot 3^0 + 1 \cdot 3^1 + 2 \cdot 3^2 = 23$.

To get a natural number from the key, for the key-space $\mathbb{K}$ one has to find a suitable base b, such that keys can be interpreted as sequences of digits to base b.

Keys as natural numbers (cont.)

For example consider the hashing of strings: If we consider an ASCII-string, then we have $b = 128$, and for example "CLRS" yields $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141764947$.

We see that we must be prepared to handle large numbers, especially if we consider the now dominant character-encoding http://en.wikipedia.org/wiki/UTF-8, which extends ASCII, and for which $b = 2^{32} = 4294967296$ would be appropriate.

In practice, one would avoid the inefficient detour through large numbers, but one would integrate the translation to numbers and the hashing of numbers, to avoid the large numbers altogether.

For hashing of strings, of course canned solutions exist — however in general there is no best solution, and the situation at hand should be studied.

The division method

Now our keys are natural numbers, i.e., $\mathbb{K} \subseteq \{0, 1, 2, \ldots\}$.

The easiest hash function $h : \mathbb{K} \to \{0, \ldots, m-1\}$ is given by

$$h(k) := k \bmod m.$$

Recall that in C-based languages (like C, C++, Java) the mod-operator, that is, the *remainder-operator*, is computed by the %-operator, and that in such languages for non-negative integers x, y we have

```
x % y == x - (x / y) * y.
```

For example

1. $3 \bmod 2 = 1$, $6 \bmod 2 = 0$
2. $7 \bmod 5 = 2$, $29 \bmod 5 = 4$.

We always have

$$x \bmod y \in \{0, \ldots, y-1\}.$$

On a good choice of the modulus m

One would think that the choice of m is just dictated by the given storage space:

- However, one typically doesn't care whether we have, say, 10000 bytes more or less.
- So we have some space to manoeuvre.
- And that space is needed to make the remainder-function a good hash function!

A general advice is:

Choose a **prime number** m
which is not "too close" to an exact power of 2.

In general, the remainder-function yields a good hash function, assuming that a prime number is chosen that is unrelated to any patterns in the distribution of the keys.

On a good choice of the modulus m (cont.)

For an example, consider $n = 2000$, where we are willing to examine on average 3 elements in an unsuccessful search. So assuming a good hash function, the table size should be around $m \approx 2000/3 = 666.\overline{6}$.

- The first prime number greater than $2000/3$ is 673.
- The closest powers of 2 are $2^9 = 512$ and $2^{10} = 1024$.
- So the choice is reasonable.
- One could also choose, say, 701.

Universal hashing

In practice hashing by division (i.e., using an appropriate remainder-function as hash function) performs well, given that we choose the modulus "well", but we do not have a guarantee for that:

Suppose that a malicious adversary, who gets to choose the keys to be hashed, has seen your hashing program and knows the hash function in advance. Then he could choose keys that all hash to the same slot, giving worst-case behaviour.

One way to defeat the adversary is to use a different hash function each time. You choose one at random at the beginning of your program. Unless the adversary knows how you'll be randomly choosing which hash function to use, he cannot intentionally defeat you.

However, just because we choose a hash function randomly, that doesn't mean it's a good hash function. What we want is to

Universal hashing (cont.)

randomly choose a single hash function from a set of good candidates.

- Such schemes exist under the name of "universal hashing" (for example by extending the division method!).
- They need to use true random bits, which one might get from the operating system, but in practice a pseudo-random generator should suffice.

Using chaining and universal hashing,
the expected time for each SEARCH operation is $O(1)$,
when using only $O(m)$ such operations.

The "expectation" (i.e., averaging) concerns only the random choice of the hash function, while the key sequence is fixed but arbitrary.

Universal hashing (cont.)

So we get, as desired, constant run-time on average *in general* (without assumptions like simple uniform hashing).

However, for practical purposes specialised hash functions are preferred, since universal hashing has a large overhead.

Perfect (i.e., collision-free) hash functions

Consider the case where the key-space is the set of strings of length n, using only the characters "$+, -, 0$".

1. How large is the key-space?
2. Can you find a bijective hash function (this is now not only perfect, but also **minimal**)?

We have three characters, so $|\mathbb{K}| = 3^n$.

To map these strings of length n, over three characters, to the interval

$$0, 1, \ldots, 3^n - 1$$

Consider the example $n = 2$, so $3^2 = 9$ tuples to "rank":

$$00,\ 0-,\ 0+,\ -0,\ --,\ -+.+0,\ +-,\ ++ \mapsto 0, \ldots, 8$$

Using the "digits" $x_i \in \{0, 1, 2\}$, the formula is

$$(x_0, \ldots, x_{n-1}) \mapsto \sum_{i=1}^{n-1} x_i \cdot 3^i.$$

Perfect (i.e., collision-free) hash functions

Consider the case where the key-space is the set of strings of length n, using only the characters "$+, -, 0$".

1. How large is the key-space?
2. Can you find a bijective hash function (this is now not only perfect, but also **minimal**)?

We have three characters, so $|\mathbb{K}| = 3^n$.

To map these strings of length n, over three characters, to the interval

$$0, 1, \ldots, 3^n - 1$$

Consider the example $n = 2$, so $3^2 = 9$ tuples to "rank":

$$00,\ 0-,\ 0+,\ -0,\ --,\ -+.+0,\ +-,\ ++ \mapsto 0, \ldots, 8$$

Using the "digits" $x_i \in \{0, 1, 2\}$, the formula is

$$(x_0, \ldots, x_{n-1}) \mapsto \sum_{i=1}^{n-1} x_i \cdot 3^i.$$

Perfect (i.e., collision-free) hash functions

Consider the case where the key-space is the set of strings of length n, using only the characters "$+, -, 0$".

1. How large is the key-space?
2. Can you find a bijective hash function (this is now not only perfect, but also **minimal**)?

We have three characters, so $|\mathbb{K}| = 3^n$.

To map these strings of length n, over three characters, to the interval

$$0, 1, \ldots, 3^n - 1$$

Consider the example $n = 2$, so $3^2 = 9$ tuples to "rank":

$$00,\ 0-,\ 0+,\ -0,\ --,\ -+.+0,\ +-,\ ++ \mapsto 0, \ldots, 8$$

Using the "digits" $x_i \in \{0, 1, 2\}$, the formula is

$$(x_0, \ldots, x_{n-1}) \mapsto \sum_{i=1}^{n-1} x_i \cdot 3^i.$$

Example for hashing by chaining and division

Demonstrate what happens when we insert the keys

$$5, 28, 19, 15, 20, 33, 12, 17, 10$$

into a hash table with collisions resolved by chaining, using the division method with $m = 9$ for the hash function.

1. $5 \mapsto 5$
2. $28 \mapsto 1$
3. $19 \mapsto 1$
4. $15 \mapsto 6$
5. $20 \mapsto 2$
6. $33 \mapsto 6$
7. $12 \mapsto 3$
8. $17 \mapsto 8$
9. $10 \mapsto 1$

Example for hashing by chaining and division

Demonstrate what happens when we insert the keys

$$5, 28, 19, 15, 20, 33, 12, 17, 10$$

into a hash table with collisions resolved by chaining, using the division method with $m = 9$ for the hash function.

1. $5 \mapsto 5$
2. $28 \mapsto 1$
3. $19 \mapsto 1$
4. $15 \mapsto 6$
5. $20 \mapsto 2$
6. $33 \mapsto 6$
7. $12 \mapsto 3$
8. $17 \mapsto 8$
9. $10 \mapsto 1$

Analysis of hashing with chaining

Under what circumstances does hashing with chaining have constant run-time for all three dictionary-operations?

When we have **simple uniform hashing**
and a constant load factor.

Analysis of hashing with chaining

Under what circumstances does hashing with chaining have constant run-time for all three dictionary-operations?

When we have **simple uniform hashing**
and a constant load factor.

Division method

Why is actually choosing for m a power of 2 or a non-prime a bad choice ?

It's about **patterns** in your data:

1. Choosing prime numbers you minimise the chance of hitting an **additive pattern**.
   Iff m is a prime number, then arithmetic progressions

   $$a, a + b, a + 2b, a + 3b, \ldots$$

   for m not a divisor of b always cover m different slots before cycling.
2. Staying away from the powers of 2 is a simple precaution to reduce the chances of hitting a **multiplicative pattern**.

Division method

Why is actually choosing for m a power of 2 or a non-prime a bad choice ?

It's about **patterns** in your data:

1. Choosing prime numbers you minimise the chance of hitting an **additive pattern**.
   Iff m is a prime number, then arithmetic progressions

   $$a, a + b, a + 2b, a + 3b, \ldots$$

   for m not a divisor of b always cover m different slots before cycling.
2. Staying away from the powers of 2 is a simple precaution to reduce the chances of hitting a **multiplicative pattern**.

General hash functions

Assume you have a good hash function for strings: Can you derive hash functions for other data types?

Poor man's hash function: just represent the data via strings.

General hash functions

Assume you have a good hash function for strings: Can you derive hash functions for other data types?

Poor man's hash function: just represent the data via strings.