data structures for sets of information

where is $x$?

add $x$

remove $x$

what is the smallest?

what is the largest?

given $x$, what is the next one?

given $x$, what is the previous one?
linear data structures

stack

queue

today: linked list
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
implementations using arrays

the determined size might be much too large or too small
lists: properties

also a linear data structure

any data can be in a list

nodes with access to the next one (and the previous one)
node in a singly linked list

a node $v$ contains:

a data element $v.key$ also written $v.element$

a pointer $v.next$ to the next node
singly linked lists

$L.head$ points to the first node

if list empty, then $L.head = \text{nil}$

possibly also:
variable $L.tail$ points to the last node

possibly also:
variable $L.size$ indicates number of nodes
lists: possible operations

accessor operations:
  • first()
  • last()
  • before(n)
  • after(n)

query operations:
  • isFirst(n)
  • isLast(n)
  • inList(n)

(n is a node)
lists: more possible operations

generic operations:
  • size()
  • isEmpty()

update operations:
  • replaceElement(n, d)
  • insertBefore(n, d)
  • insertAfter(n, d)
  • insertFirst(d)
  • insertLast(d)
  • remove(n)

(n and m are nodes; d is data)

many possible operations, so we need to be specific in our ADT
size: different possibilities

if we have a global variable size: return that one in $\mathcal{O}(1)$

otherwise: in $\mathcal{O}(n)$

**Algorithm** $\text{size}(L)$:

1. $s := 0$
2. $m := L\text{.head}$
3. while $m \neq \text{nil}$ do
   1. $s := s + 1$
   2. $m := m\text{.next}$
4. return $s$
insertAfter: example

\[
\text{first} \rightarrow n \rightarrow \text{last} \rightarrow \emptyset
\]

\[
\text{first} \rightarrow n \rightarrow \text{last} \rightarrow \emptyset
\]
Algorithm insertAfter\((n, d)\):

new Node \(x\)

\(x.key := d\)

\(x.next := n.next\)

\(n.next := x\)

if \(n\) was the last node then now \(x\) is the last node

the size increases by one
remove a node

how to remove a node?

what is a special case
Algorithm listSearch($L, d$):

\[
\begin{align*}
x & := L.\text{head} \\
\text{while } x \neq \text{nil} \text{ and } x.\text{element} \neq d \text{ do} \\
& \quad x := x.\text{next} \\
\text{return } x
\end{align*}
\]

input: a list and a key, output: node containing the key or nil
search using sentinels

sentinel is a dummy object that simplifies dealing with boundaries

$L.nil$ represents nil, and is as the start

$L.nil.next$ points to the head of the list

empty list is just the sentinel $L.nil$

$L.head$ no longer necessary
list search
without sentinels:

Algorithm listSearch\((L, d)\):
\[
x := L\cdot head
\]
\[
\textbf{while } x \neq \text{nil} \textbf{ and } x\cdot key \neq d \textbf{ do}
\]
\[
x := x\cdot next
\]
\[
\text{return } x
\]

with sentinels:

Algorithm listSearchSen\((L, d)\):
\[
x := L\cdot nil\cdot next
\]
\[
\textbf{while } x \neq L\cdot nil \textbf{ and } x\cdot key \neq d \textbf{ do}
\]
\[
x := x\cdot next
\]
\[
\text{return } x
\]
## singly linked lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>first, (last, )after</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>before</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>replaceElement, swapElements</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAfter</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertBefore</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

**NB:** size, last, insertLast, remove
a node $v$ contains:

- a data element $v.key$
- a link to the next node $v.next$
- a link to the previous node $v.prev$
doubly linked lists

can also be done without sentinel node

*L.nil* a sentinel node

*L.nil.next* points to the head (as for singly linked)

*L.nil.prev* points to the tail (new)

empty lists consists of just the sentinel *L.nil*,

then both *L.nil.next* and *L.nil.prev* point to *L.nil*
insertAfter in doubly linked list in $\mathcal{O}(1)$
delete in doubly linked list
without sentinel:

**Algorithm** listDelete($L, x$):

if $x . prev \neq \text{nil}$ then

$x . prev . next := x . next$

else

$L . head := x . next$

if $x . next \neq \text{nil}$ then

$x . next . prev := x . prev$

with sentinels

**Algorithm** listDeleteSet($L, x$):

$x . prev . next := x . next$

$x . next . prev := x . prev$
doubly linked lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(), isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>first, last, after</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>before</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>replaceElement(,,) swapElements</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertFirst(), insertLast()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAfter</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertBefore</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove()</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
again sets containing data: dictionaries

- insert
- search
- delete

example: table of identifiers in programming language

we would like to do those operations in $O(1)$

compute address instead of search for item
hashing

hash table is an effective data structure for implementing dictionaries
worst-case for operations usually in $\Theta(n)$ with $n$ number of items
in practice often much better, search even in $O(1)$
hash table generalizes array where we access address $i$ in $O(1)$
applications of hashing in compilers and cryptography
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
direct-address table

universe of keys: \( U = \{0, \ldots, m - 1\} \) with \( m \) small

use array of length \( m \): \( T[0 \ldots (m - 1)] \)

what is stored in \( T[k] \)?
direct-address table

universe of keys: $U = \{0, \ldots, m - 1\}$ with $m$ small

use array of length $m$: $T[0 \ldots (m - 1)]$

what is stored in $T[k]$?

either nil if there is no item with key $k$

or a pointer $x$ to the record with key $x.key = i$ and satellite data $x.element$
direct-address table: example

keys: integers in \( \{0, \ldots, m - 1\} \)

elements: (here) courses

item \( i = (k, e) \) is stored in array \([0 \ldots (m - 1)]\)
operations for direct-address table

\[
\text{insert}(S, x) \text{ add } x \text{ in set } S \\
T[x.\text{key}] := x
\]

\[
\text{delete}(S, x) \text{ remove } x \text{ from set } S \\
T[x.\text{key}] := \text{nil}
\]

\[
\text{search}(S, k) \text{ search for } \text{key } k \text{ in } S \\
\text{return } T[k]
\]

\( x \) is pointer to record with key \( x.\text{key} \) and satellite data \( x.\text{element} \)
analysis of direct-address table

inserting, deleting, searching all in $O(1)$
drawbacks of direct-address table

if universe of keys $U$ is large we need a lot of storage
also if we actually use only a small subset of $U$

keys must be integers

so: hashing
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
hash tables

hash function maps keys to indices (slots) 0, \ldots, m - 1 of a hash table
so \( h : U \rightarrow \{0, \ldots, m - 1\} \)

usually more keys than indices: \( |U| >> m \)

element with key \( k \) hashes to slot \( h(k) \)
hashing: example

keys are first names, elements are phone numbers

hash function: length modulo 5

Alice
John
Sue
collisions

problem: different keys may be hashed to the same slots
so $h(k) = h(k')$ with $k \neq k'$
this is called a collision

because number of keys $|U|$ larger than number of slots $m$
the hash function $h$ cannot be injective
do we often have collisions?

for $p$ items and a hash table of size $m$:

$m^p$ possibilities for a hash function

if $p = 8$ and $m = 10$ already $10^8$ possibilities

there are $\frac{m!}{(m-p)!}$ possibilities for hashing without collision

if $p = 8$ and $m = 10$ then $3 \cdot 4 \cdot \ldots \cdot 10$ such possibilities
collisions: probabilities

birthday paradox

for 23 people the probability that everyone has a unique birthday is \(< \frac{1}{2}\)

that is: for \(p = 23\) and \(m = 366\) the probability of collision is \(\geq \frac{1}{2}\)
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
solving collisions using chaining

create a **list** for each slot

link records in the same slot into a list

slot in hash table points to head of a linked list

(nil if list is empty)
chaining: example

hash function is month of birth modulo 5

drawback: pointer structures are expensive
for insertion: in $O(1)$

for deletion: in $O(1)$ if lists are doubly-linked

for searching:
if every key hashes to the same slot then just (long) linked list

in $O(n)$ with $n$ size of the dictionary
chaining: average case analysis

assumption:
key is hashed to any arbitrary slot, independent of other keys

we have \( n \) keys and \( m \) slots

probability of \( h(k) = h(k') \) is \( \frac{1}{m} \)

expected length of list at \( T[h(k)] \) is \( \frac{n}{m} \)

this is called the load factor \( \alpha = \frac{n}{m} \)

expected search time for unsuccessful case:

computing \( h(k) \) and accessing the slot, and cost of searching list

so \( \Theta(1 + \alpha) \)
also average-case successful search in $\Theta(1 + \alpha)$

if $\alpha \in \mathcal{O}(1)$ (constant!) then expected search time in $\Theta(1)$

$n \in \mathcal{O}(m)$ also expected search time in $\Theta(1)$

(number of slots proportional to number of keys)

if hash table is too small it does not work properly!
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
choosing a hash function

in view of the assumption in the analysis: what is a good hash function?
distributes keys uniformly and seemingly randomly
regularity of keys distribution should not affect uniformity
hash values are easy to compute: in $O(1)$
(properties can be hard to check)
hash functions: division method

a key $k$ is hashed to $k \mod m$

**pro:** easy to compute

**contra:** not good for all values of $m$

do not take $m$ with small divisor
division method: bad choices for \( m \)

if \( m \) and all keys even then we use half of the slots

\( m = 100 \) and we store 50, 100, 150, 200, 250

\[
egin{align*}
50 & \mapsto 50 \\
100 & \mapsto 100 \\
150 & \mapsto 50 \\
200 & \mapsto 100 \\
250 & \mapsto 50
\end{align*}
\]

\( m = 2^4 = 16 \) and we store 0, 32, 64, 96

\[
egin{align*}
0000000 & = 0 \mapsto 16 \\
0010000 & = 32 \mapsto 16 \\
0100000 & = 64 \mapsto 16 \\
0110000 & = 96 \mapsto 16
\end{align*}
\]

heuristics: take \( m \) prime not too close to a power of 2 or 10 (common bases)
Choose a constant $A$ with $0 < A < 1$.

Compute $k \cdot A$.

Take its fractional part: $k \cdot A - \lfloor k \cdot A \rfloor$.

Multiply the result with number of slots $m$.

Take floor.

A key $k$ is hashed to $\lfloor (m \cdot (k \cdot c - \lfloor k \cdot c \rfloor)) \rfloor$. 
multiplication method: example
multiplication method: which $A$ are good?

Advantage: value of $A$ is not critical, but some are better than others. An optimal value of $A$ depends on the data being hashed.

Knuth recommends the golden ratio (see book p264)

$$A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887\ldots$$
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
open addressing

alternative to chaining for solving collisions

we make a probe sequence

\[ h : U \times \{0, \ldots, m - 1\} \rightarrow \{0, \ldots, m - 1\} \]

deletion is difficult
open addressing: example

hash key $k = 387$ with table partly full
open addressing: linear probing

next probe: try the next address modulo $m$

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

we get clustering!

and removal difficult
quadratic probing

\[ h(k, i) = h'(k) + a \cdot i + b \cdot i^2 \mod m \]
open addressing: double hashing

we use two hash functions

\[ h(k, i) = h_1(k) + i \cdot h_2(k) \mod m \]

search for a key doing:

\[ h_1(k) + 0 \cdot h_2(k) \mod m \]
\[ h_1(k) + 1 \cdot h_2(k) \mod m \]
\[ h_1(k) + 2 \cdot h_2(k) \mod m \]
\[ \vdots \]
double hashing: example

\[
\begin{align*}
    h_1(k) &= k \mod m \\
    h_2(k) &= 1 + (k \mod (m - 2))
\end{align*}
\]
double hashing: example

\[ m = 13, \ h(k) = k \mod 13, \ h'(k) = 7 - (k \mod 7) \]

<table>
<thead>
<tr>
<th></th>
<th>(h(k))</th>
<th>(h'(k))</th>
<th>try</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td>5, 10</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>4</td>
<td>5, 9, 13</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
double hashing: analysis

probe sequence: \( h(k, 0), h(k, 1), \ldots, h(k, m - 1) \)

assumption:
each key is equally likely to have any one of the \( m! \) permutations as its probe sequence regardless of what happens to the other keys

assumption:
load factor \( \alpha = \frac{n}{m} < 1 \)
expected number of probes for unsuccessful search

probe 1: with probability $\frac{n}{m}$ collision, so go to probe 2

probe 2: with probability $\frac{n-1}{m-1}$ collision, so go to probe 3

probe 3: with probability $\frac{n-2}{m-2}$ collision, so go to probe 4

note: $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$

expected number of probes:

$$1 + \frac{n}{m}(1 + \frac{n-1}{m-1}(1 + \frac{n-2}{m-2}(\ldots))) \leq 1 + \alpha(1 + \alpha(1 + \alpha(\ldots))) \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$
double hashing: remarks

if $\alpha < 1$ constant then expected number of probes in $O(1)$

if table is full for 50% then we expect 2 probes

if table is full for 90% then we expect 10 probes
universal hashing

a hash function is selected randomly at run-time
from a family of good hash functions
then good expected performance
overview

- linked lists
- direct-address table
- hash tables
- chaining
- hash functions
- open addressing
- material
Donald Knuth: The art of computer programming
volume Sorting and Searching