Content:

- Implementation of Dictionaries using:
  - Ordered Search Tables
  - Hash tables
  - Skip Lists

ADT Map

- Domain: sets of pairs \(<k, i>\)
  where \(k\) key, \(i\) information,
  the sets are partial functions on keys!
- Operations:
  - \(size\), \(isEmpty\), \(get(k)\), \(put(k,v)\), \(remove(k)\)
- Examples:
  - Examination list: personal number used as key, identifier list created by a compiler.
  - Static Maps: no updates allowed
  - Dynamic Maps: updates are allowed

ADT Dictionary

- Domain: sets of pairs \(<k, i>\)
  where \(k\) key, \(i\) information
  Is ADT Map a special case of ADT Dictionary?
- Operations:
  - \(size\), \(isEmpty\), \(find(k)\), \(findAll(k)\), \(insert(k,v)\), \(remove(k,v)\)
- Example:
  - Telephone directory (multiple numbers allowed)
  - Static Dictionary: no updates allowed
  - Dynamic Dictionary: updates are allowed
**Implementations: Map, Dictionary**

- **Table/Array** – seq. of memory chunks of equal size
  - Unordered: no particular order between \( T[i] \) and \( T[i+1] \)
  - Ordered: ...but here \( T[i] < T[i+1] \) [Goodrich/Tamassia 9.3.3]

- **Linked Lists** (from Lecture 2)
  - Unordered [Goodrich/Tamassia 9.3.1]
  - Ordered

- **Hashing** [Goodrich/Tamassia 9.2]

- **Skip Lists** [Goodrich/Tamassia 9.4]

- **(Binary) Search Trees** [Goodrich/Tamassia 10]
  Lecture 5

---

**Table representations of a Dictionary (1)**

**Unordered table:**

lookUp by **linear search**

- unsuccessful lookup: \( n \) comparisons \( \rightarrow O(n) \) time
- successful lookup, but worst case: \( n \) comparisons \( \rightarrow O(n) \) time
- successful lookup, average case with uniform distribution of requests:

\[
\frac{1}{n} (1 + 2 + \ldots + (n-1) + n) = \frac{n+1}{2} \text{ comparisons } \rightarrow O(n) \text{ time}
\]

---

**Table representations of a Dictionary (2)**

**Ordered table:**

**LookUp by binary search:**

\[
\text{Function binLookup(table } T[min..max], \text{ key } k) \\
\text{ if (min>max) then return NULL} \\
\text{ mid } \leftarrow \frac{min+max}{2} \\
\text{ if } k = \text{key}(T[mid]) \\
\text{ then return } T[mid] \\
\text{ else if } k < \text{key}(T[mid]) \\
\text{ then return binLookup(} T[min..mid-1) \\
\text{ else return binLookup(} T[mid+1..max)
\]

---

**Analysis of binLookup (binary search)**

- First iteration:

- ...second:

- ...third:

...and very fast we will find the key!

How fast? Better than \( O(n) \)?
Analysis of binary search (cont.)

Function \texttt{binLookup} (table \texttt{T}[\texttt{min}..\texttt{max}], key \texttt{k})

- if (\texttt{min} > \texttt{max}) then return \texttt{NULL}
- \texttt{mid} ← \left\lfloor \frac{\texttt{min} + \texttt{max}}{2} \right\rfloor
- if \texttt{k} = \texttt{key(T[mid])} then return \texttt{T[mid]}
- else if \texttt{k} < \texttt{key(T[mid])} then return \texttt{binLookup(T[\texttt{min}..\texttt{mid}-1])}
- else return \texttt{binLookup(T[\texttt{mid}+1..\texttt{max}])}

Hypothesis:

- \texttt{T(1)} = \(c\)
- \texttt{T(2)} = \(c + T(1) = 2c\)
- \texttt{T(4)} = \(c + T(2) = 3c\)
- \texttt{T(n)} = \(c + T(n/2) = \ldots\)?

Proof:

- \texttt{T(2n)} = \(c + T(n) = c + (c + \log(n)c) = 2c + \log(n)c = (2 + \log(n))c\)
- \texttt{= \log(2) + \log(n)c = \log(2) + \log(2n)c = (1 + \log(2n))c = c + \log(2n)c\)

...thus \texttt{binSearch \in O(\log(n))}

LookUp of Map/Dictionary in \(O(\log(n))\) ...ok?

We can do better with HASH TABLES

- Idea:
  - Given a small table to store elements in...
  - ...for each element find a suitable table index!

  - Find a function \(h: \text{key} \to i \in [0..\text{max}]\)
    - such that \(k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)\)
  - Store each key-element pair as \(<k, e>\) in \(T[h(k_i)]\)

Hash Table – Collision Resolution

Two principles for handling collisions:

1. Chaining: keep conflicting data in linked lists
   - Separate Chaining: Keep a linked list of the colliding ones outside the table
   - Coalesced Chaining: Store all items inside the table
2. Open Addressing: Store all items inside the table, and the index to use at collision is determined by an algorithm
Hashing with Separate Chaining: Example

- Hash table of size: 13
- Hash function $h$ with $h(k) = k \mod 13$
- Store 10 integer keys: 54, 10, 18, 25, 28, 41, 38, 36, 12, 90

Hashing with Separate Chaining: LookUp

Given: key $k$, hash table $T$, hash function $h$
1. compute $h(k)$
2. search for $k$ in the list pointed by $T[h(k)]$

Notation: probe = one access to the linked list data structure
- 1 probe for accessing the list header (if nonempty)
- 1 + 1 probes for accessing the contents of the first element
- 1 + 2 probes for accessing the contents of the second element
- ...
A probe (just pointer dereferencing) takes constant time. How many probes $P$ are needed to retrieve a hash table entry?

Separate Chaining: Unsuccessful LookUp

- $n$ data items
- $m$ positions in the table

Worst case:
- all items have the same hash value: $P = 1 + n$

Average case:
- Hash values equally distributed among $m$:
- Average length $\alpha$ of the list: $\alpha = n/m$
  $\alpha = n/m$ is called the load factor
- $P = 1 + \alpha$

Separate Chaining: Successful LookUp

Average case: Expected number $P$ of probes for given key $k$...
- Access to $T[h(k)]$ (beginning of a list $L$): 1
- Traversing $L \rightarrow k$ found after: $(|L| + 1)/2$
- Expected (or average) $|L|$
corresponds to $\alpha$, thus: $P = \alpha/2 + 3/2$
Coalesced Chaining: items inside table (1)

First step:
- store first element in table
- keep rest in separate lists

The increase in space consumption is acceptable if key fields are small or hash table is quite full.

Coalesced Chaining: items inside table (2)

Place data items in the table
- Extend them with pointers
- Resolve collisions by using the first free slot

Chains may contain keys with different hash values...
...but all keys with the same hash value appears in the same chain
+ better space utilization
- table may become full

Open Addressing

- Store all elements inside the table
- Use a fix algorithm to find a free slot
  Sequential / Linear Probing
  - desired hash index \( j = h(k) \)
  - in case of conflict go to the next free position
  - If at end of the table, go to the beginning...
  - Close positions rapidly filled (primary clustering)
  - How to remove(k) ??

Open Addressing – how to remove()

The element to remove may be part of a collision chain – can we tell?

If it is part of a chain, it can’t be removed entirely!
- Since all keys are stored, re-hash all remaining data?
- Scan elements after, re-hash or compact when suitable, stop at first free slot... ??
- Ignore – insert a “deleted” marker if the next slot is non-empty...
Double Hashing – or what to do on crash

- Second hashing function \( h_2 \) computes increments in case of conflicts
- Increment beyond the table is taken modulo \( m = \text{tableSize} \)

Linear probing is double hashing with \( h_2(k) = 1 \)

Requirements on \( h_2 \):
- \( h_2(k) \neq 0 \) for all \( k \)
- For each \( k \), \( h_2(k) \) has no common divisor with \( m \)
  \( \Rightarrow \) all table positions can be reached

A common choice: \( h_2(k) = q - (k \mod q) \) for \( q < m \) prime, \( m \) prime (i.e., pick a prime less than the table size!)

What is a good hash function?

Assume \( k \) is a natural number.

Hashing should give a uniform distribution of hash values, but this depends on the distribution of keys in the data to be hashed.

Example:
- hashing last names in a (Swedish) student group
- hash function: the ASCII-value of the last character
  Bad choice: majority of the names ends with \( n \)

Hashing by Modulo-Division

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

- Avoid:
  - \( m = 2^d \): hashing gives \( d \) least bits of \( k \)
  - \( m = 10^d \): hashing of decimal numbers gives last \( d \) digits

- Prime numbers suggested for \( m \)
- Check samples of real data to experiment with hash parameters

Skip Lists

- A hierarchical linked list...
- A probabilistic alternative for implementation of ADT Dictionary
- Insertion uses randomization ("coin flipping")
- Good expected-case performance
- Worst-case performance in skip lists is unlikely
  (>250 items, the risk of a search time more than 3 times the expected is below \( 10^{-6} \))
Skip List data structure

- Special keys: $\leftrightarrow$ and $\infty$ ...smaller/larger than any real key...
- A list head with info about max size (height) of the list...
- Several levels of linked lists, less dense higher up
  - Level 0: all keys in a linked list between $\leftrightarrow$ and $\infty$ in order by the '<' -relation
  - Level 1: at $\leftrightarrow$ and $\infty$ there is a common start node and end node for all levels, and at some internal nodes a bridge between the levels... (a "tower")...where we can switch to a more fine grained list

Skip Lists – searching...
When searching for a key $k$:
- Follow the list of the highest level...
  - Stop before we pass a $k_i > k$ (we risk to overshoot what we’re looking for)
  - If found, return it, otherwise...
- We have stopped at a level:
  - We have found the key?
  - No, switch to next lower level (via the "last tower") and continue searching.
- Returns: The largest key $k_i \leq k$ (which might be $\infty$)

Similarity with binary search – but for lists
- Example: LookUp(18)
Skip List - Insert

- Insert(x):
  - P := LookUp(x) ...we have a position P in the list
  - If (P.value < x):
    - ...insert new list cell after P
    - "toss a coin" to decide how high this "tower" should be:
      - while ("toss a coin" = yes) do
        - increase the tower one step

Skip List – Delete ...and properties

- Similar to Insert:
  - Search
  - If found, remove and fix the link between the towers
- Worst case execution time
  - of LookUp, insert and delete on a skip list of n items is $O(n)$
- Expected execution time (assuming uniform distribution of keys) is $O(\log n)$ if search starts at a height $\lceil \log n \rceil$
- You can play with Skip List animation
  - http://iamwww.unibe.ch/~wenger/DA/SkipList/