What is a Dictionary?

Container class
• Stores key-element pairs (like priority queue)

Allows “look-up” (find) operation

Allows insertion/removal of elements

May be unordered or ordered

Dictionary Keys

Must support equality operator
• For ordered dictionary, also support comparator operator
  — useful for finding neighboring elements

Sometimes required to be unique

Dictionary Examples

Natural language dictionary
• word is key
• element contains word, definition, pronunciation, etc.

Web pages
• URL is key
• html or other file is element

Any typical database (e.g. student record)
• has one or more search keys
• each key may require own organizational dictionary

Unordered Dictionary ADT

findElement(k): Return element with key k
insertItem(k,e): Insert element e with key k
removeElement(k): Remove element with key k

Special sentinel, NO_SUCH_KEY returned when no element with key is present

Log File

Store key-element pairs in unsorted sequence
Always insert using insertLast( )
• $O(1)$ time

findElement( ) by traversing entire list
• $O(n)$ time

Good when inserts are common and finds are rare (e.g. archiving data records)
• number of searches = $O(1) \rightarrow O(n)$ total time
• number of searches = $O(n) \rightarrow O(n^2)$ total time
Hash Table

Provides efficient implementation of unordered dictionary
- Insert, remove, and find all $O(1)$ expected time

Bucket array
- Provides storage for elements

Hash function
- Maps keys to buckets (ranks)
- For each operation, evaluate hash function to find location of item

Bucket Array

Each array element holds 1 or more dictionary elements

Capacity is number of array elements

Load is percent of capacity used
- $N$ is capacity of hash table
- $n$ is size of dictionary
- $n/N$ is load of hash table

Collision is mapping of multiple dictionary elements to the same array element

Simplest Hash Table

Keys are unique integers in range $[0, N-1]$

Trivial hash function
- $h(k) = k$

Uses $O(N)$ space (can be very large)
- okay if $N = O(n)$
- bad if key can be any 32-bit integer
  - table has $2^{32}$ entries = 4 gigaentries

find(), insert(), and remove() all take $O(1)$ time

Hash Function

Maps each key to an array rank
- $h(k): K \rightarrow R$
- array rank is integer in $[0, N-1]$

Decomposed into two parts
- hash code generation
  - converts key to an integer
- compression map
  - converts integer hash code to valid rank

Generating Hash Codes: Java's Object.hashCode()

generates integer for any object

generates same integer for two objects as long as equals() method evaluates to true
- different instances with same value are not equal according to Object.equals()
  - won’t always give expected hashing behavior

exact method is implementation dependent

“Good” hash function

Want to “spread out” values to avoid collisions
Ideally, keys act as random distribution of ranks
- Probability($h(k) = i$) = $1/N$ for all $i$ in $[0, N-1]$
- Expected keys in bucket $i$ is $n/N$
  - this is $O(1)$ if $n = O(N)$
If no collision, operations are $O(1)$
- so expected time is $O(1)$ for all operations

Note: worst case time is still $O(n)$
Generating Hash Codes:  
**Cast to Integer**  
Works well if key is byte, short, or char type  
• can use Float.floatToIntBits() for floats  

Disadvantages  
• High order bits ignored for longs/doubles  
  —May result in collisions  
• Cannot handle more complex keys  

Generating Hash Codes:  
**Polynomial Hash Codes**  
Multiply each component by some constant to a power  
\[ h(x_0, x_1, x_2, ..., x_{k-1}) = \sum_{i=0}^{k-1} a_i x_i \]  
\[ = x_0 + a(x_1 + a(x_2 + ... + a(x_{k-1}))) \]  
• Makes hash code dependent on order of components  

Disadvantages  
• \( k-1 \) multiplies in hash evaluation  
• Choice of \( a \) makes big difference in “goodness” of hash function  

Generating Hash Codes:  
**Summing Components**  
Add up multiple integers to get a single integer  
• Ignore overflows  
\[ h(x_0, x_1, x_2, ..., x_{k-1}) = \sum_{i=0}^{k-1} x_i \]  

Examples  
• Long or double may be converted to two ints (high order and low order) and summed  
• Strings may be broken into multiple characters and summed  

Disadvantage  
• Ordering of integers is ignored  
  —May result in collisions  

Generating Hash Codes:  
**Cyclic Shift**  
Cyclic Shift Hash Codes  
• Rotates bits of current code by some number of positions before adding each new component  
\[ h(x_0, x_1, x_2, ..., x_{k-1}) = \text{rotate}(x_{k-1} + \text{rotate}(x_{k-2} + ... + x_1 + \text{rotate}(x_0))) \]  
• no multiplication  
  —only addition and bitwise shifts and ORs  

Disadvantages  
• Choice of rotation size still makes big difference in “goodness”  

Compression Maps  
**Division Method**  
• \( h(k) = |k| \mod N \)  
• \( N \) works best if it is a prime number  
• Even then, multiples of \( N \) map to same position  
  —\( h(iN) = 0, h(i(N+j)) = j \mod N \)  

MAD (multiply, add, and divide) Method  
• \( h(k) = |ak+b| \mod N \)  
  —\( h(iN) = |aiN+b| \mod N = b \mod N \)  
  —\( h(i(N+j)) = |aiN+aj+b| \mod N \)  
  = \( |aj+b| \mod N \)  
• Not clear that this is much better...  

Collision Handling:  
**Chaining**  
For each bucket, store a sequence of elements that map to the bucket  
• effectively a much smaller, auxiliary dictionary  
  Linearly search sequence to find correct element
Chaining Example

\[ N = 7, \quad h(k) = |k| \mod N \]

Insert 19 36 5 21 -4 26 14  
\[(\text{load} = 1)\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
& & & & & & \\
21 & 36 & & & & & \\
14 & & & 26 & & & \\
\end{array}
\]

Collision Handling: Open Addressing

Store only 1 element per bucket

- No addition space, but requires smaller load

If multiple elements map to same bucket, use some method to find empty bucket

- Linear probing
  \[ h'(k) = (h(k) + j) \mod N \quad j = 0, 1, 2, 3, \ldots \]
  - Keep adding 1 to rank to find empty bucket

- Quadratic probing
  \[ h'(k) = (h(k) + j^2) \mod N \quad j = 0, 1, 2, 3, \ldots \]

- Double hashing
  \[ h'(k) = (h(k) + j^a h''(k)) \mod N \quad j = 0, 1, 2, 3, \ldots \]

Linear Probing Example

\[ N = 7, \quad h(k) = |k| \mod N \]

Insert 19 36 5 21 -4 26 14  
\[(\text{load} = 1)\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
& & & & & & \\
21 & 36 & & & & & \\
14 & & & 26 & & & \\
\end{array}
\]

Other Open Addressing Difficulties

Searching

- For NO_SUCH_KEY, must search until empty bucket found

Removing

- Cannot just empty the bucket
  - could disconnect colliding keys

- Easiest method is setting with special DELETED_KEY sentinal
  - insert() can reuse bucket
  - find() must continue searching beyond bucket

Rehashing

When load of hash table gets too large

- Allocate new hash table
- Generate new hash function
- Re-hash old elements into new table
- Time cost may be amortized as in dynamic array
  - must increase size by \( O(n) \) each time

Ordered Dictionary ADT

Unordered Dictionary ADT plus:

- closestKeyBefore(k): returns key preceding k
- closestElemBefore(k): returns element preceding k
- closestKeyAfter(k): returns key following k
- closestElemAfter(k): returns element following
Ordered Dictionaries

Simplest type is “lookup table”
- Store elements in sorted vector
- Insert takes O(n)
- Find takes O(log n)
  — use binary search

Skip Lists

Based on stacked set of linked lists (a hierarchy of lists)
- \(S_h, S_{h-1}, \ldots, S_0\): \(h\) is height of skip list
- \(S_h\) is entire dictionary
- \(S_i\) contains a subset of \(S_{i+1}\)
  — Each element of \(S_i\) is 50% likely to appear in \(S_{i+1}\)

Provides expected bounds of \(O(\log n)\) for find, insert, and remove
- with “high probability” - uses randomization

Example Skip List

Each skip list/row is a level
Each column is a tower
- links connect elements within level or tower

Searching (findElem)

Begin at left end of highest level
1. Scan forward while key ≤ search key
2. If level > 0, drop down to next level. Goto 1.

Search Example

findElement(50)

Insertion

1. Find insert position in level 0 using search and do insert
2. Flip coin. If heads, insert in level \(i+1\), and repeat 2 until tails
Insert Example

\[ \text{insertItem}(42, \text{elem}) \]

- \text{random()} \text{ returns: } H, H, T, \ldots

Removal

1. Find element in level 0 using search
2. Remove from level 0 and follow tower to remove from all levels

Basic Analysis - height (drop down)

Probability that given item appears in level \( i \):
\[ \frac{1}{2^i} : \frac{1}{2} \times \frac{1}{2} \times \ldots \times \frac{1}{2} \]

Probability that level \( i \) has at least 1 element:
\[ P_i \leq n \times \frac{1}{2^i} = n/2^i \]

\[ P_{\text{log}_n} \leq n/2^{\log n} = n/2^{\log_2 n} = n/n^3 = 1/n^2 \]

So height is < \( 3\log n \) with high probability

- Expected height is \( O(\log n) \)

Basic Analysis - scan forward

Imagine scan in reverse direction
Each element scanned has 1/2 chance of having an element in tower above it

Expected number of elements scanned before going up tower is 2 = \( O(1) \)

Search Analysis

Number of drop down steps is \( O(\log n) \)
Number of scan forward steps is \( O(\log n) \)
Total expected search time is \( O(\log n) \)

Same applies to insert and remove

Worst case: \( O(h + n) \)
In Class Example

Work in groups of 2-3

Assume calls to random() return:

H T T H H T T H . . .

Create skip list with these inserts:

10 15 12 5 20 17 25

What is maximum height for any sequence of inserts? Why?

What is expected search time for this random() distribution? Why?