Outline: hash tables and Huffman trees

- hash tables with chained hashing
- hash tables with open addressing
- Huffman trees
- Huffman coding
- implementation of Huffman coding

hash table collisions

- there are two main sources of collision in a hash table:
  - hash function collision: two different keys hash to the same integer
    - if adding all the letters, “edo” and “doe” both hash to 24, which is an inevitable collision
      - the example function is not a very good hash function!
      - a cryptographic hash function makes it very hard to find data that collides
  - hash table array index collision: two different integers, modulo the hash table length, give the same index
    - when adding all the letters of the keys “edo”, “hello”, and “world”, table size 11 has no collisions, table size 3 has collisions
    - so resizing the hash table may change the number of hash table collisions
      - but cannot easily predict what table size wold remove all collisions

hash table load factor

- the load factor of a hash table is the number of elements divided by the table size
- as the load factor approaches 100%, collisions are more likely

ways to handle hash table collisions

- an array location can only hold one item, so what to do in case of a collision?
- three main solutions:
  - can increase (e.g. double plus one) the size of the array until all collisions go away
    - only works if the hash function itself has no collisions, and may use a lot of memory
  - can have each array element refer to a linked list rather than a single element: chained hashing
  - can look for another, unused place in the same array: open address hashing

hash table runtime analysis

- three main solutions:
  - increase the size of the array until collisions go away
  - chained hashing: each array element refer to a linked list of values
  - open address hashing: look for an array location with no element
- in-class exercise: assuming few collisions, what is the average runtime of find and add for each of the above strategies?
- in-class exercise: assuming many collisions, what is the worst-case runtime of find and add for each of the above strategies?

chained hashing

- chaining or chained hashing: each array element refers to a linked list of elements

```
LinkedNode<T> hashTable[]
```

- with a really bad hash function (e.g. a function that always returns the same integer) all the data is stored in one linked list
- with good hash functions, most lists will be short
- the load factor can exceed 100%, because an array element can refer to any number of values
- requires dynamic memory allocation
  - fine for Java, less interesting for languages such as C or for code inside the operating system
- over half the storage is for links rather than data, so inefficient for very large hash tables
- when there is no concern about space usage and about dynamic memory allocation, chained hashing is the preferred method for resolving collisions

open addressing

- when inserting a value in a hash table, if the slot (array location) indicated by the hash function is full, insert it into another slot
- this works until the hash table is full (100% load factor), i.e. it works as long as there are open slots
- the probe sequence determines where to look next when there is a collision
- when looking up a value in a hash table, the same probe sequence must be followed as when inserting

probing in open addressing

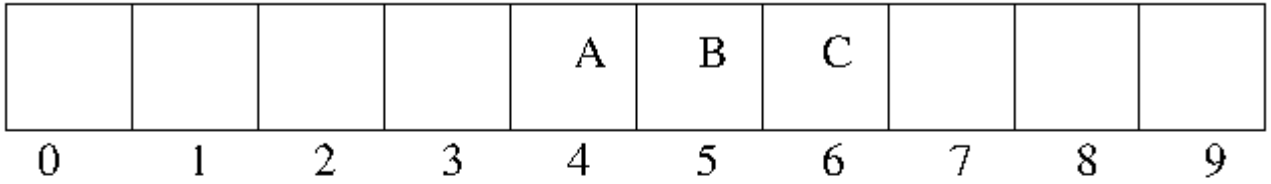
- assume a collision at index i
- linear probing: look at subsequent locations for an open slot: i, i + 1, i + 2, i + 3, i + 4, ...
- double hashing: a second hash function determines the step size: if the second hash function gives me h, look in locations i, i + h, i + 2h, i + 3h, i + 4h, … (mod the table size)
- double hashing avoids the bunching up of data at popular indices of the hash table
  - as long as any collisions are on the table index rather than the hash functions

open addressing with deletion

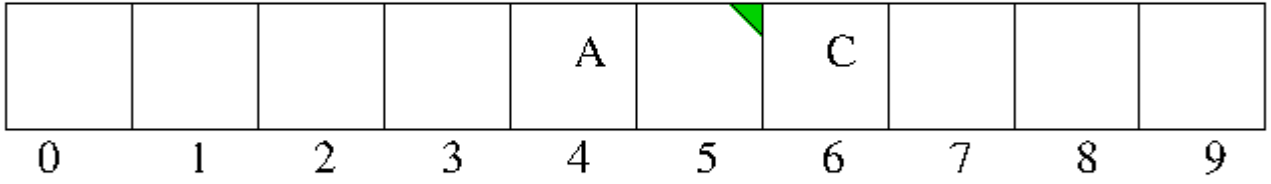
- some hash tables only allow adding values, while others also allow removing values
- when removing a value from a hash table with open addressing, the hash table must record that the element was removed, so future searches can keep looking when they reach the slot of a deleted element
- each slot must record whether it is empty, full, or deleted
- a slot can only be empty until the first time a value is inserted, after which it can only be full or deleted
  - which means searches may have to look at many slots

open addressing with deletion: example

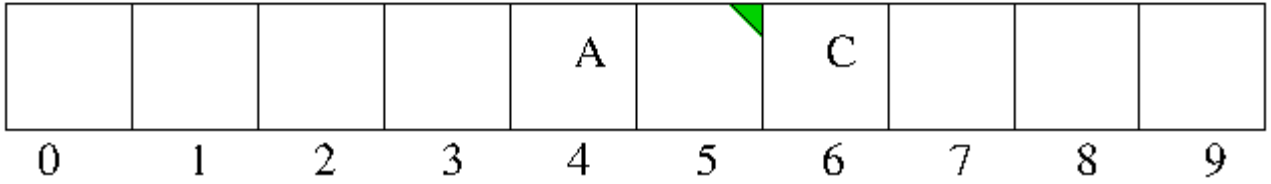
inserting: A with hash 24, B with hash 55, C with hash 14



remove B



search for C, starting from array index 4



linear probing (linear hashing)

- increment the index (modulo the array size) until a free slot is found
- with unevenly-distributed keys and hash functions that mostly give numbers close to each other, all the values are stored next to each other in the array
- this may lead to long times for search, insertion, and removal

open addressing table size

- load factor cannot exceed 100%, so the table must have at least as many slots as the number of stored elements
- if the table size is a prime number, linear hashing or double hashing will visit the entire table before giving up
- otherwise, for example double hashing in a table of size 100, if the step size is 10, will only visit 1/10th of the slots
- if the table size is ever changed, each element must be reinserted using the recomputed hash function and the new table size
  - since just copying the old array would map an element to the wrong index:

    24 modulo 11 = 2, but 24 modulo 23 = 1
- for the same amount of data, the overall storage used is less than chained hashing

Huffman coding

- suppose you wanted to compress a text file, that is, represent it using the least possible number of bits
- suppose also you want to use a unique string of bits for each character -- that is, we are not going to encode entire words
- it would be good to use fewer bits for characters that occur frequently, and as many bits as needed for other characters
- The encoding length is minimized by using shorter bit strings to encode more commonly-used characters, and longer bit strings to encode less frequently-used characters

- Huffman coding is an algorithm for assigning variable-length bit strings to symbols so as to minimize the length of the encoded string

Huffman coding example

- for example, in the string "hello world", I could use the following encoding:

  h 11000

  e 11001

  l 0

  o 10

  ' ' (space) 11010

  w 11011

  r 1110

  d 1111

- “hello world” has 3 l’s and 2 o’s, so these get shorter encodings than the other characters
- in-class exercise: what does the bit string (25 bits) "1100 0100 1111 1101 0111 1101 0111 0" represent?
- in comparison, the above string takes 45 bits if each character is represented with a constant 5 bits/character, 63 bits using 7-bit ASCII, and 72 bits using UTF-8
  - 5 bits/character which is the smallest number of bits to represent all 26 characters and space
- so Huffman coding can save space! the compressed string uses 25 / 72 = 35% of the bits of the (normal) UTF-8 encoding.

Algorithm for building a Huffman coding tree

1. make a list of all symbols with their frequencies
2. sort the list so symbols with lower frequency are in front
3. if the list only has one element, the element is the root of the tree and we are done
4. remove the first two elements from the list and put them as subtrees of a new node – a new binary tree
5. add the frequencies of the two subtrees to give the frequency of this binary tree
6. insert this tree in the right place in the sorted list
7. return to step 3

- in-class exercise: do this for the characters in “hello world”
  - 3 L, 2 O, and one each of H, E, space, W, R, and D

Using a Huffman coding tree to encode a string of characters

- to encode
  - find the leaf with the next character to encode
  - starting from the root to this leaf, place a 0 bit for each time you take the left subtree, and a 1 bit for each time you take the right subtree
- in-class exercise: use this algorithm and the tree from the last exercise to encode the string "wow"
- can do the above search once for each character, put the results in a table, then use the table to do the actual encoding quickly

Using a Huffman coding tree to decode a string of bits

- to decode:
  - use each bit to go left (0 bit) or right (1 bit) in the tree
  - if you have reached a leaf, put this character into the result string, then
  - start at the root of the tree with the next bit
- in-class exercise: use this algorithm and the above tree to decode as many bits as possible of the bit string "1100 0100 1101 0111 1101 0111 0"

Compression using Huffman coding

- an encoding based on letter frequencies in one string (or in a large sample) can be used for encoding many different strings
- if so, a single copy of the table (tree) can be kept, and Huffman coding is guaranteed to do no worse than fixed-length encoding as long as all the strings have the same character frequencies
- otherwise, a separate table (tree) is needed for each compression, and the table has to be included in the count of bytes to be stored or transmitted after compression
- in such cases, Huffman coding might actually give a somewhat larger size than the original
- in practice, even including the table, Huffman coding is usually worthwhile for sufficiently long strings in natural languages, which have lots of redundancy and different letter frequencies

An implementation of Huffman coding: data structures

- many possible implementations, this is one (textbook, chapter 6.1 and 6.7)
- use several data structures:
  - a priority queue to hold the sorted data (could be built using a heap)
  - each priority queue element refers to a tree node
  - an interior tree node has no value, but has two children
  - a leaf node has a value, but has no children
  - each node has a frequency of occurrence, which is used as a priority in the queue (low priority returned first)

An implementation of Huffman coding: algorithms

- begin by computing the frequency of each value
  - perhaps by using a hash table
- once the frequency of values is known, insert each value into the priority queue, using its frequency as a priority
- remove the front two elements from the queue, create an interior node to refer to these two elements, and insert the interior node back into the queue with, as priority, the sum of the priorities of the two nodes
- once there is only one element left in the queue, this is the huffman tree
- a further step would be to build an encoding table from the tree
- in-class exercise: do this given the string "there is no place like home"