CSCI 204: Data Structures & Algorithms

Revised by Xiannong Meng based on textbook author's notes

Hash Maps Implementation and Applications

Revised based on textbook author's notes.

Table Size

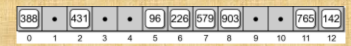
- How big should a hash table be?
  - If we know the max number of keys.
    - create it big enough to hold all of the keys.
  - In most instances, we don't know the number of keys.
- Most probing techniques work best when the table size is a prime number.

Rehashing

- We can start with a small table and expand it as needed.
  - Similar to the approach used with the array.
- **load factor** – the ratio between the number of keys and the size of the table.
  - A hash table should be expanded before the load factor reaches 80%.

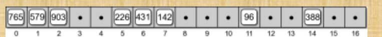
Rehashing Example

- After creating a larger array for the table, we can not simply copy the original keys to the new table.



- We must rebuild or rehash the entire table.

```
h(765)  => 0              h(579)  => 1
h(431)  => 6              h(226)  => 5
h(96)   => 11             h(903)  => 2
h(142)  => 6   => 7       h(388)  => 14
```



Expansion Size

- Size of the expansion depends on the application.
- Good rule of thumb is to at least double its size.
- Two common approaches:
  - double the size of the table, then search for the first larger prime number.
  - double the size of the table and add one to ensure M is odd.

Efficiency Analysis

- Depends on:
  - the hash function
  - size of the table
  - type of collision resolution probe
- Once an empty slot is located, adding or deleting a key can be done in O(1) time.
- The time required to perform the search is the main contributor to the overall time of all ops.

Efficiency Analysis

- Best case: O(1)
  - The key maps directly to the correct entry.
  - There are no collisions.
- Worst case: O(m)
  - Assume there are *n* keys stored in a table of size *m*.
  - The probe has to visit every entry in the table.

Efficiency Analysis

- While hashing appears to be no better than a basic linear search or binary search in worst case, hashing is very efficient in the average case with load factor < 0.8. (Table shows the data for M == 13.)
- Remember linear search O(n), binary search O(log n) and log 13 is about 3.7, hashing is O(1).

Load Factor	0.25	0.5	0.67	0.8	0.99
Successful search:					
Linear probe	1.17	1.50	2.02	3.00	50.50
Quadratic probe	1.66	2.00	2.39	2.90	6.71
Unsuccessful search:					
Linear probe	1.39	2.50	5.09	13.00	5000.50
Quadratic probe	1.33	2.00	3.03	5.00	100.00

Hash Functions

- The efficiency of hashing depends in large part on the selection of a good hash function.
  - A "perfect" function will map every key to a different table entry.
    - This is seldom achieved except in special cases.
  - A "good" hash function distributes the keys evenly across the range of table entries.

Function Guidelines

- Important guidelines to consider in designing a hash function.
  - Computation should be simple.
  - Resulting index can not be random.
  - Every part of a multi-part key should contribute.
  - Table size should be a prime number.

Common Hash Functions

- **Division** – simplest for integer values.

```
h(key) = key % M
```

- **Truncation** – some columns in the key are ignored.
  - Example: assume keys composed of 7 digits.
  - Use the 1st, 3rd, 6th digits to form an index (M = 1000).

Common Hash Functions

- **Folding** – key is split into multiple parts then combined into a single value.
  - Given the key value 4873152, split it into three smaller values (48, 73, 152).
  - Add the values together and use with division.

Hashing Strings

- Strings can also be stored in a hash table.
  - Convert to an integer value that can be used with the division or truncation methods.
- Simplest approach: sum the ASCII values of individual characters.
  - Short strings will not hash to larger table entries.
- Better approach: use a polynomial.

$$S_0 a^{n-1} + S_1 a^{n-2} + \cdots + S_{n-3} a^2 + S_{n-2} a + S_{n-1}$$

The HashMap ADT

- Hash tables are commonly used to implement a map or dictionary.
  - Same as the Map ADT.
  - Keys must be hashable.
- Python's dictionary is implemented using a hash table.

HashMap Implementation

- Hash table:
  - Initial size: M = 7
  - Must expand as needed.
  - Load factor: 2/3
  - Expansion size: 2M + 1
- Entries:

```
class _MapEntry :
  def __init__( self, key, value ):
    self.key = key
    self.value = value
```

HashMap Implementation

- Use double hashing:
  - Hash function:

```
h(key) = |hash(key)| % M
```

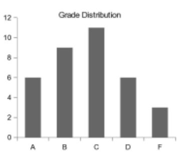
  - Probe function:

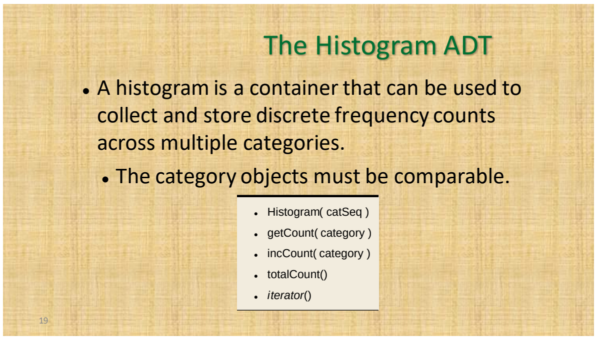
```
hp(key) = 1 + |hash(key)| % (M - 2)
```

- `hash()` is Python's built-in `hash()` function.
  - Takes a built-in type as the key and returns an int value that can be used with division method.

Application: Histograms

- Graphical chart of tabulated frequencies.
  - Very common in statistics.
  - Used to show the distribution of data

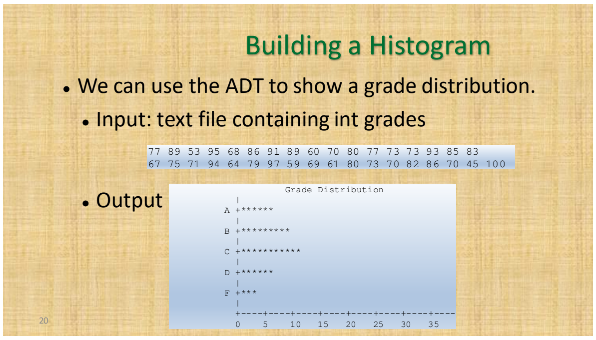




The Histogram ADT

- A histogram is a container that can be used to collect and store discrete frequency counts across multiple categories.
  - The category objects must be comparable.

- Histogram(catSeq)
- getCount(category)
- incCount(category)
- totalCount()
- *iterator*()



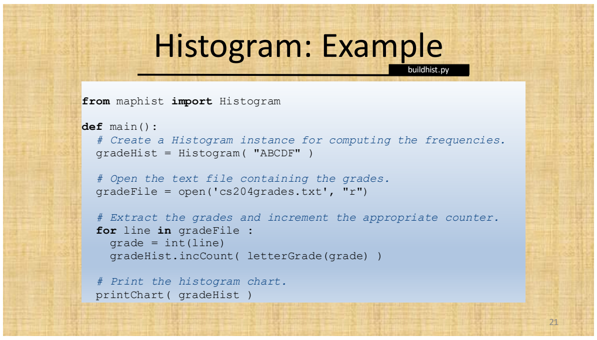
Building a Histogram

- We can use the ADT to show a grade distribution.
  - Input: text file containing int grades

```
77 89 53 95 68 86 91 89 60 70 80 77 73 73 93 85 83
67 75 71 94 64 79 97 59 69 61 80 73 70 82 86 70 45 100
```

  - Output

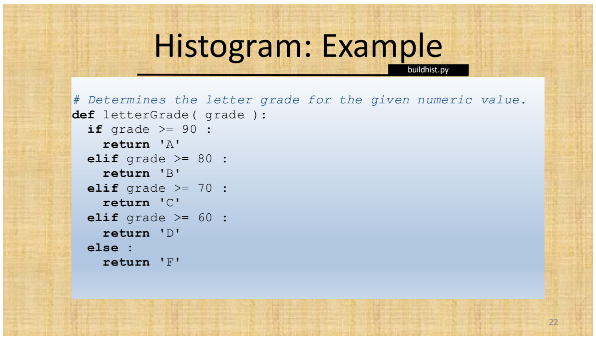
```
              Grade Distribution
  |
A +******
  |
B +*********
  |
C +***********
  |
D +******
  |
F +***
  |
  +----+----+----+----+----+----+----+----
  0    5    10   15   20   25   30   35
```



Histogram: Example

buildhist.py

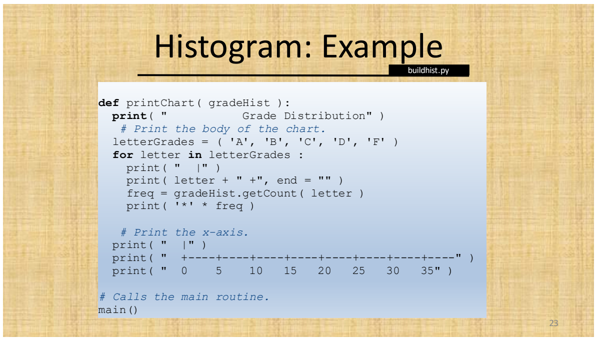
```
from maphist import Histogram

def main():
  # Create a Histogram instance for computing the frequencies.
  gradeHist = Histogram( "ABCDF" )

  # Open the text file containing the grades.
  gradeFile = open('cs204grades.txt', "r")

  # Extract the grades and increment the appropriate counter.
  for line in gradeFile :
    grade = int(line)
    gradeHist.incCount( letterGrade(grade) )

  # Print the histogram chart.
  printChart( gradeHist )
```



Histogram: Example

buildhist.py

```
# Determines the letter grade for the given numeric value.
def letterGrade( grade ):
  if grade >= 90 :
     return 'A'
  elif grade >= 80 :
     return 'B'
  elif grade >= 70 :
     return 'C'
  elif grade >= 60 :
     return 'D'
  else :
     return 'F'
```



Histogram: Example

buildhist.py

```
def printChart( gradeHist ):
  print( "           Grade Distribution" )
   # Print the body of the chart.
  letterGrades = ( 'A', 'B', 'C', 'D', 'F' )
  for letter in letterGrades :
    print( "  |" )
    print( letter + " +", end = "" )
    freq = gradeHist.getCount( letter )
    print( '*' * freq )

   # Print the x-axis.
  print( "  |" )
  print( "  +----+----+----+----+----+----+----+----" )
  print( "  0    5    10   15   20   25   30   35" )

# Calls the main routine.
main()
```