Hashing

EECS 233

Hashing

- We have learned several data structures that allow us to store and search for data items using their key fields.
  - Unsorted lists and arrays - linear search complexity
  - Sorted arrays - log complexity for search; linear for inserts
  - Various trees (AVL, B-trees, Heaps)
    - Generally logarithmic time complexity
    - Rich operations:
      - Insert, remove, search, find max/min, top-k

Taking Stock…

- The efficiency of the different data structures and operations:

Data structure	Search for an item	Insert an item
List (unsorted array)		
List (sorted array)		
List (linked list)		
Binary search tree	?	
Balanced search tree (AVL, B-trees)		
Heaps		

- Can we do better than logarithmic complexity?

- Hash tables and hashing techniques may allow us to search, insert, and delete an item in sub-logarithmic (average-case) time, namely, O(1)

Storing Objects in an Array: Keys and Indexes

- Key: a (unique) attribute of object
- Index: a position of object in an array
- Unsorted arrays:
  - Indexes are independent of keys
  - Searching is dumb and inefficient
- Sorted arrays:
  - Indexes are related to keys
  - Searching is smart (no longer blind) and efficient (logarithmic)
- The hashing idea: treat the key as an index!
  - Searching is simple and takes almost no time (constant) = genius
- Example: storing grades about students in this class
  - Give each student a unique key (integer from 1-150).
  - Store the student records in an array, in the position determined by the key
  - We can perform both search/update and insertion in $O(1)$ time (for up to 150 students).

Hash Functions

- Problem with student grades:
  - A student will have different keys in different classes
  - What if we used a social security number?
  - Nice, but a humongous array!
- In many real-world problems, the key attribute has semantic meaning
  - Cannot be arbitrarily assigned
  - Phone book: key is person's name, not a unique number
  - Web cache: key is a URL
- To handle these problems, we use a *hash function*
  - Convert ("map") keys into array indices
    - Domain: the keys
    - Range: integers in [0, size-of-array)
  - The word "hash": to chop into small pieces (Merriam-Webster)
    - Chopping large domain space into small number of array cells

Hash Tables

- Example: student list
  - Index = SSN mod 150
- Problem: multiple keys mapping to the same index
  - Two students with SSN 150-00-0001 and 150-00-0151
    - 150000001 mod 150 = 150000151 mod 150 = 1
  - We need techniques to handle such cases called *collisions*
- The resulting data structure is known as a *hash table*
  - Hash function
  - Array
  - Procedures and data structures to handle collisions
- Operations:
  - Insert
  - Remove
  - Contains (search)
  - isEmpty, makeEmpty, etc.
  - No findMax!

Hash Functions Desiderata

- Example 1: Salary as key
  - 10-workers shop
  - Keep employees record in hash table with (salary mod 10) hash function
  - What if all salaries are multiple of 10K? - Bad function!
  - What if all salaries are multiple of 2K? - Better function!
  - Random salaries - Good function!

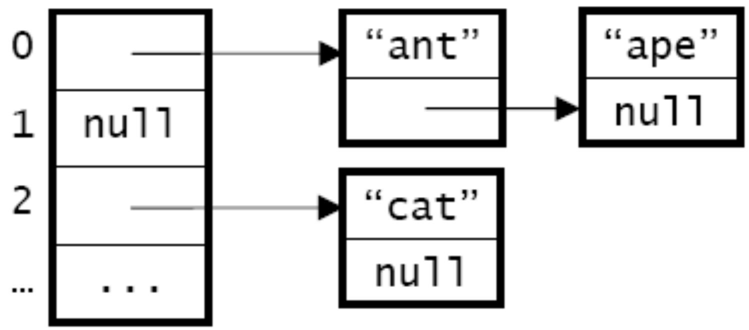
- Example 2: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - h(key) = (the byte sum of all characters) mode table-size
    - example: h("cat") = ('c' + 'a' + 't') mod 500000
    - But: assume words are mostly up to 20 character-long
    - Max sum is 127*20 = 2540; any larger table is of no use!

- Requirements for good hash functions:
  - Full table size utilization
  - Even ("uniform") key mapping throughout the table
    - Utilizing known key distribution in the objects
    - Making hash function "random" - the distribution of keys is independent of distribution of indexes they map to

Handling Collisions with Chaining

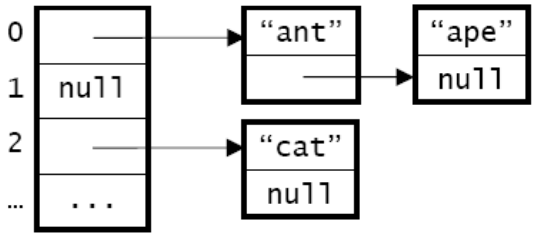
- If multiple items are assigned the same hash code, “chain” them together. Each position in the hash table serves as a *bucket* that is able to store multiple data items.

- Implementation: a linked list
  - disadvantage: memory overhead for the references



- Alternative implementation: each bucket is itself an array (or points to an array)
  - disadvantages:
    - large buckets can waste memory
    - a bucket may become full; *overflow* occurs when we try to add an item to a full bucket

Operations with Chaining



- Search for an item
  - Need to traverse the corresponding linked list or array
  - To guarantee short lists, the number of hash table slots should be of the same order as the total number of items
  - *Load factor:* ratio of the number of items to the number of hash table slots
- Inserting an item
  - Insertion in a linked list or array
  - New list node must be allocated
  - Array size may need to be dynamically adjusted
- Removing an item
  - The size of an almost-empty array needs to be adjusted
  - List node must be garbage-collected

Handling Collisions with Open Addressing

- When the position assigned by the hash function is occupied, find another open position - a process called *probing*.
  - Example: h(key) = <character encoding of first char> – <encoding of ‘a’>
  - “wasp” has a hash code of 22, but it ends up in position 23, because position 22 is occupied.

- The hash table also performs probing to search for an item.
  - example: when searching for “wasp”, we look in position 22 and then look in position 23
  - we can only stop a search when we reach an empty position

0	“ant”
1	
2	“cat”
3	
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	
24	“yak”
25	“zebra”

Linear Probing for Open Addressing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.
  - Example:
    - “ape” (h = 0) would be placed in position 1, because position 0 is already full.
    - “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
    - where would “whale” end up?

- Advantage: if there is an open position, linear probing will eventually find it.

- Disadvantage: “clusters” of occupied positions develop
  - Increases the lengths of subsequent probes.
  - As load factor increases, both search and insert times look increasingly linear!

0	“ant”
1	
2	“cat”
3	
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Quadratic Probing for Open Addressing

- Probe sequence: h(key), h(key) + 1, h(key) + 4, h(key) + 9, …, wrapping around as necessary.
  - the offsets are perfect squares: $h + 1^2$, $h + 2^2$, $h + 3^2$, …
  - Example:
    - "ape" (h = 0): try 0, 0 + 1 – open!
    - "bear" (h = 1): try 1, 1 + 1, 1 + 4 – open!
    - "whale"?

- Advantage: reduces clustering

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Theorem: If the load factor is less than 50% and the table size is prime, a new item can always be inserted

Double Hashing for Open Addressing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - Probe sequence: h1, h1 + h2, h1 + 2*h2, …
- Example:
  - h1 = our previous function h
  - h2 = number of characters in the string
  - "ape" (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - "bear" (h1 = 1, h2 = 4): try 1 – open!
  - "whale"?
- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - Theorem: will find an open position if there is one, provided the table size is a prime number.
  - Disadvantage: the need for two hash functions

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Removing Items with Open Addressing

- Consider the following scenario using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”

0	“ant”
1	ape
2	“cat”
3	
4	“emu”
5	
…	. . .
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Removing Items with Open Addressing

- Consider the following scenario using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item, but “bear” is further down in the table

  - Cannot tell if it is not in the table

0	“ant”
1	ape
2	“cat”
3	
4	“emu”
5	
…	. . .
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

- Any difference with quadratic probing or double hashing?

Removing Items with Open Addressing

- When we remove an item from a position, we need to leave a special value in that position to indicate that an item was removed.
- Three types of positions: occupied, empty, “removed”.
- When we search for a key, we stop probing when we encounter an empty position, but not when we encounter a removed position.
  - How to search for “ape”?
  - How to search for “bear”?
- We can insert items in either empty or removed positions.

0	“ant”
1	REMOVED
2	“cat”
3	“bear”
4	“emu”
5	
…	. . .
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”