Search Methods

- The main operation used by previously covered search methods was the comparison of keys
  - In sequential search the data structure is searched in turn to determine the location of the element being sought: O(n)
  - In binary search, a sorted table is divided successively into halves, and a key comparison determines if an element is found: O(lg n)
  - In a binary search tree the search is limited to the right or left subtree based on key comparisons: O(lg n)
- Is there another approach to storing and retrieving data, that does not rely on searching?

Hashing

- We can calculate the location of the data in the table
  - If the key is known, the data can be accessed directly
  - Search for an item no longer depends on *n* – it is constant: O(1)
- This process of assigning a key to a location is called hashing
- Hashing is accomplished via a hash function, *h*
  - The function must be able to convert the data type being stored (integer, string, object, record,…) to a numeric table index
  - The element is subsequently retrieved by computing its location
- While in theory hashing gives constant time O(1) access, in practice this is seldom possible, and can at best only be approximated

Hashing - When?

- Along with B-Trees, hashing is one of the primary methods used to store large databases on disk
- When should hashing be used?
  - For exact match queries
  - To answer questions like “What data, if any, has this key?”
- What is hashing not good for?
  - Finding ranges of keys
  - Finding the minimum or maximum key
  - Processing data in key order

C++ Data Structures that use Hashing

- Sets (and multisets, ordered and unordered)

```
set<int> set1;                                      // declare a set of integers (ordered by default)
set1.insert(10);                                    // insert the value 10
set1.insert(3);                                     // insert the value 3
set<int>::iterator it;                              // declare iterator to set of ints
for (it = set1.begin(); it != set1.end(); ++it)     // use it to loop through
        cout << *it << " ";                         // outputs in order: 3 10
```

- Maps (and multimaps, ordered and unordered)

```
map<string,int> map1;                               /* map with string keys and int values
                                                       NOTE: stored as pairs */
map1["abc"]=100;                                    // insert key = "abc" , value = 100
map1["b"]=200;                                      // insert key = "b" , value = 200
map<string,int>::iterator it = map1.find("b");      // declare iterator and look for key b
if (it != map1.end())                               // if the key is found
        cout << it->second << endl;                 // output the value (second part of pair)
```

- How are the above stored/retrieved/deleted?
  - Mainly in tables (arrays) indexed by a value from a hash function

Hash Function

- Assumption: data is stored in a table that holds *m* items
  - The size of the hash table (usually an array) is part of the data structure, and can significantly impact its performance
  - *m* is usually a large prime number, or a nonprime that has no prime factors below 20
  - Items are indexed into locations 0 through *m*-1 of the table
- The hash function maps a data key to a table index
  - It should be simple to calculate (to keep computing cost low)
  - It should distribute keys uniformly over the index range
- If *h* transforms every possible key to a different index, *h* is said to be a perfect hash function
  - The hash table must have as many slots as there are keys
  - In reality, since all keys are generally not known, some approximation is required

Common Hash Functions

- Most hashing methods involve modulo division, in order to calculate a valid table index (between 0 and m-1)
  - $h(k) = h_k$ (function application) followed by h_k *mod* m to index the hash table
  - This approach works well when little is known about keys
- Common hash functions, *h*
  - Direct key mod – used when the key is an integer: k_i *mod* m
    - Example: if the key is 511, in a table of size 367 (a prime #), the key would map to index 144 (511 % 367)
  - Converting alphanumeric characters in strings to their ASCII values, and adding them (or combining in them another way)
    - "*Data*" = 68+97+116+97 = 378; in a hash table of size 367 (a prime #) "*Data*" would map to index 11 (378 % 367)

Common Hash Functions - Folding

- ***Shift folding*** – the key is split into parts which are then combined in some way (for example, by adding)
  - A social security #, say 123-45-6789, may be split into 3 parts: 123, 456, and 789 – adding these yields 1368
  - If m = 367 this value maps to index 267 (1368 % 367)
- ***Boundary folding*** - key is split into parts, alternating parts are reversed, and the parts combined
  - Using the same SS# above, the parts are 123, 654, 789, and sum to 1566; if m = 367 this maps to index 98
- The combination method is not restricted to addition
- Folding is often combined with other methods, before modding by m (i.e., convert to ASCII, then fold, then mod)

Common Hash Functions – Partial Keys

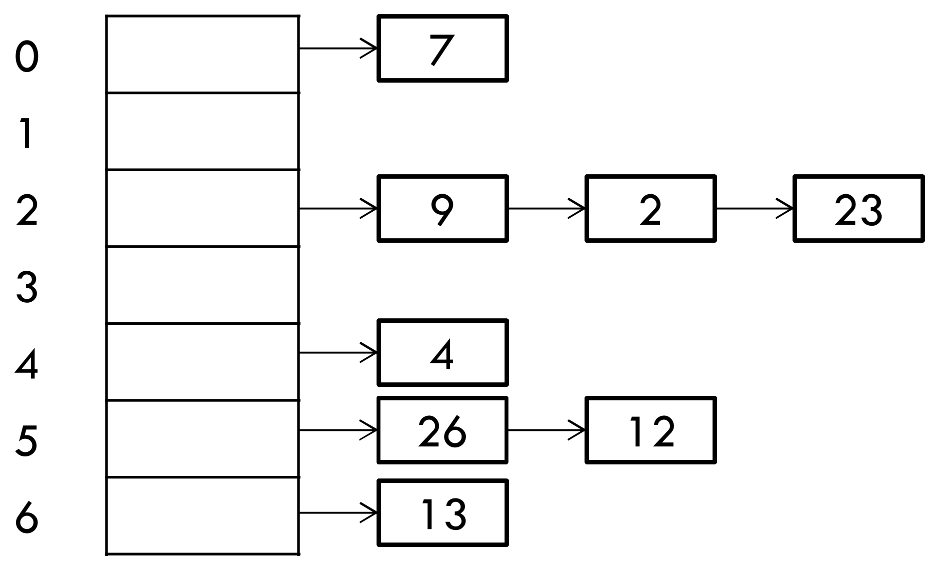
- The ends of numbers often change little, resulting in non-uniform key distribution – other digits may improve this
- Mid-square
  - The number is squared, and the middle part is the key to be modded; Ex: $3{,}121^2 = 9{,}740{,}641$ (the key used is 406)
  - Mid-square is often used after converting a number to binary
- Extraction
  - Uses only part of the key – parts are chosen and combined
  - Ex: Using the earlier SS# example (123-45-6789)
    - First-4: *1234*; Last-4: *6789*; First-2 + Last-2: *1289*; etc.
  - Care must be taken to choose well: First-4 yields little diversity in a geographical region

Collisions

- All hash function looked at so far allow the possibility that multiple keys hash to the same location in the table
- Examples: (assume table size $m = 367$)
  - Direct: $h(511) = h(878) = 144$
  - ASCII sum: *h("Data") = h("Dang") = 378 = 11*
  - Folding: $h(123\text{-}45\text{-}6789) = h(314\text{-}94\text{-}5109) = 1368$
- Collisions happen when a key hashes to an index (initially its home position) already occupied by another element
- The approach used to resolve collisions is the collision resolution policy - a number of such methods exist

Separate Chaining

- *Separate chaining* (also called open hashing) – This policy stores data items outside the hash table
  - Hash table contains pointers to linked lists, rather than data
  - Items are hashed to a location in the array, and then inserted into the linked list at that slot
- Example: Table size, $m = 7$; Insert 26, 13, 9, 4, 2, 7, 12, 23



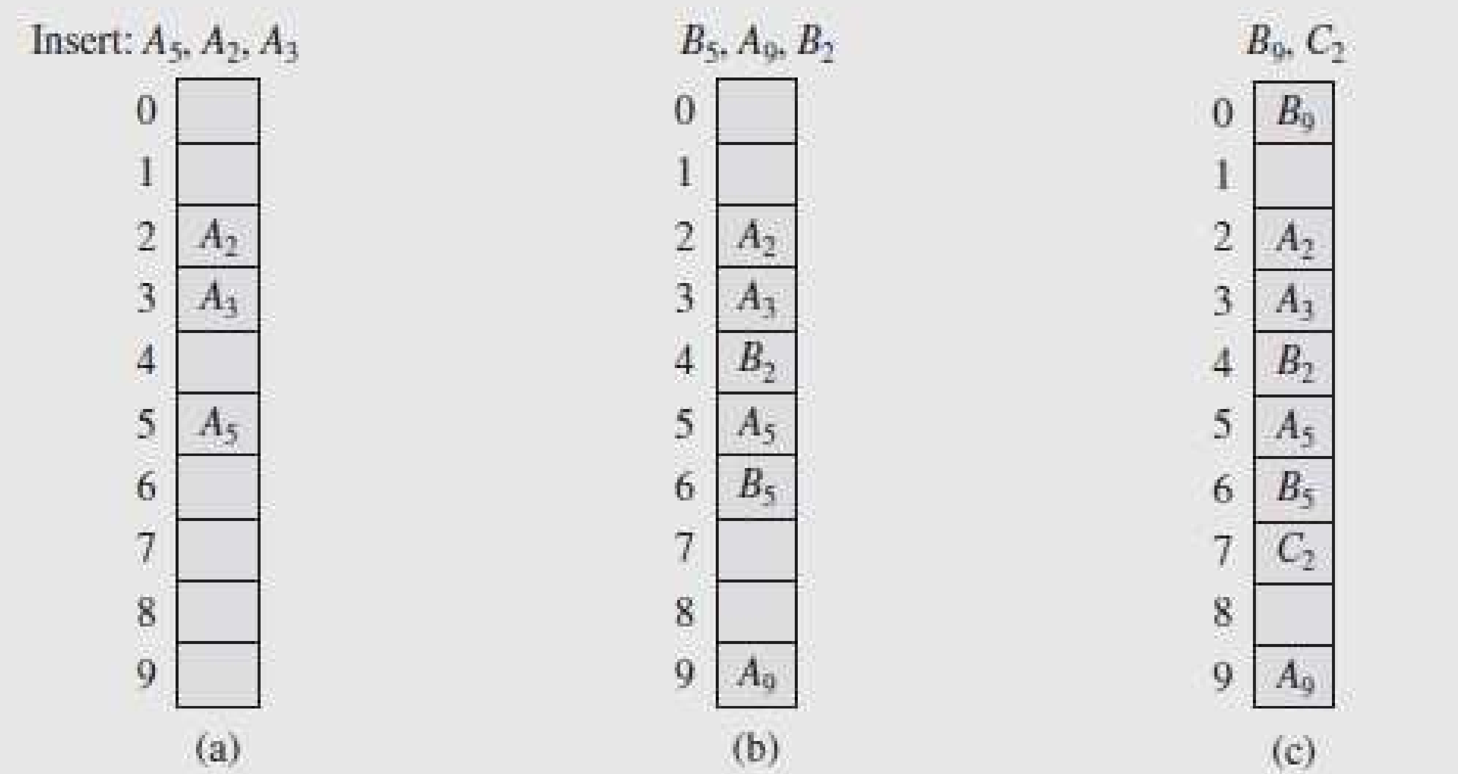
Open Addressing Hashing

- ***Open addressing*** hashing (not to be confused with open hashing) is an approach that allows data to be stored anywhere in the hash table
- Since all data is kept inside the table, this is a form of ***closed hashing***
- Collisions are resolved by finding an open position other than that hashed
- If the position h(k) is occupied, positions are tried in a probing sequence:
  - norm(h(k) + p(1)), norm(h(k) + p(2)), . . . , norm(h(k) + p(*i*)), . . .
- Function p is called a probing function, and *i* is the probe for the *i*th try
- norm is a normalization function, often modulo division by the table size
- Probing is done until an open location is found, the index wraps around to the original hash location, or all possibilities are tried (hash table full)
- Multiple methods for probing are possible: linear and quadratic probing, and double hashing (uses a second hash function to generate the next try)

Linear Probing

- In linear probing, the probing function is simply p(*i*) = *i*
  - The *i*th attempt to store the key adds *i* to the home position
- If there's a collision at *that* location, the next position is tried, and so on
- If the end of the table is reached without finding an empty slot, the search "wraps" to the start of the table
- In the worst case, the search terminates in the location that precedes the home position
- The algorithm must have a way of detecting when the table is full

Linear Probing Example



Subscripts indicate the home position of the data item being inserted

What problems might there be with this approach?

Alternate probing functions

- Linear probing tends to lead to clustering in the table
  - If a cluster is created, it has a tendency to grow, and the larger the cluster, the greater likelihood that it grows more
  - This weakens the performance of storing/retrieving data
- Alternate functions include
  - Quadratic probing – A commonly used function results in the probing sequence: h(k) + i^2 , h(k) – i^2 for i = 1, 2, . . . (m – 1)/2
  - Ex: Let m = 19 and key hashes to 9 (e.g., h(k) = 9) – the probing sequence is

    9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11,… (i=1, 2, 3, 4, 5, 6, …)
    - Add probe to h(k) and mod by m Ex: probe 6^2: 9+6^2=45 … 45%19=7
    - Subtract probe from h(k) and mod by m: Ex: probe 6^2: 9-6^2=-27 … -27%19=11
  - Random probing sequence (as generated by a random seeded function)

    Probes to add in turn may: 2, 16, 9, 25, … - Seeding ensures sequence is recoverable
- Alternate probing reduces clustering but does not eliminate them

Double Hashing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:
    - $h_1(x) = x \bmod 11$
    - $h_2(x) = 7 - (x \bmod 7)$
  - Insert keys: 58, 14, 91
    - 58 mod 11 = 3
    - 14 mod 11 = 3 → 3+7=10
    - 91 mod 11 = 3 → 3+7, 3+2*7 mod 11=6

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Double Hashing Example

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
	76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5	10%7 = 3	55%7 = 6
				5 - (47%5) = 3		5 - (55%5) = 5
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

Searching a Hash Table

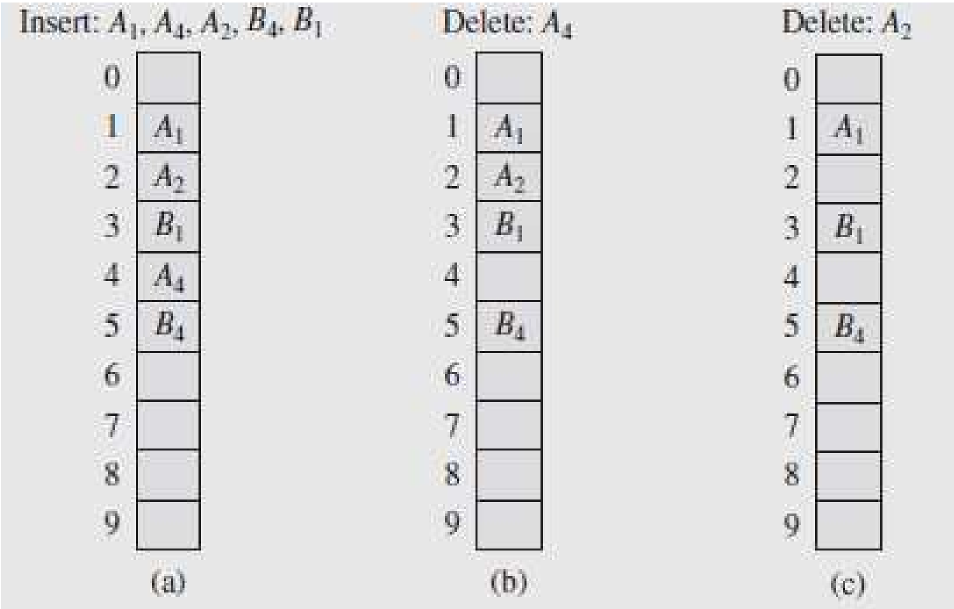
- Searching for an item in a hash table that uses separate chaining
  - Hash into the table and search the linked list until the item is found or the end of the list is reached
- Searching for an item in a hash table that employs probing can be straightforward
  - Try the home position
  - If the item is not found, try the probes in turn
  - If all probes are exhausted, the item is not in the table
  - This method may fail, if care is not taken during key deletion

Deletion

- How can data be removed from a hash table?
- If chaining is used, the deletion of an element entails deleting the node from the linked list holding that element
- For hash tables that use probing, deletion must involve care, given the handling of collisions
- The figure on the next slide illustrates the problems that may arise when searching for a node that was not inserted in its home position

Impact of Deletion on Searching (Probing)

- In (a) B_4 was not inserted in the home position (A_4 had that spot)
- In (b), after A_4 is deleted, attempts to find B_4 check location 4, which is empty – this could be interpreted as B_4 is not in the table
- A similar situation occurs in (c), when A_2 is deleted, causing searches for B_1 to stop at position 2



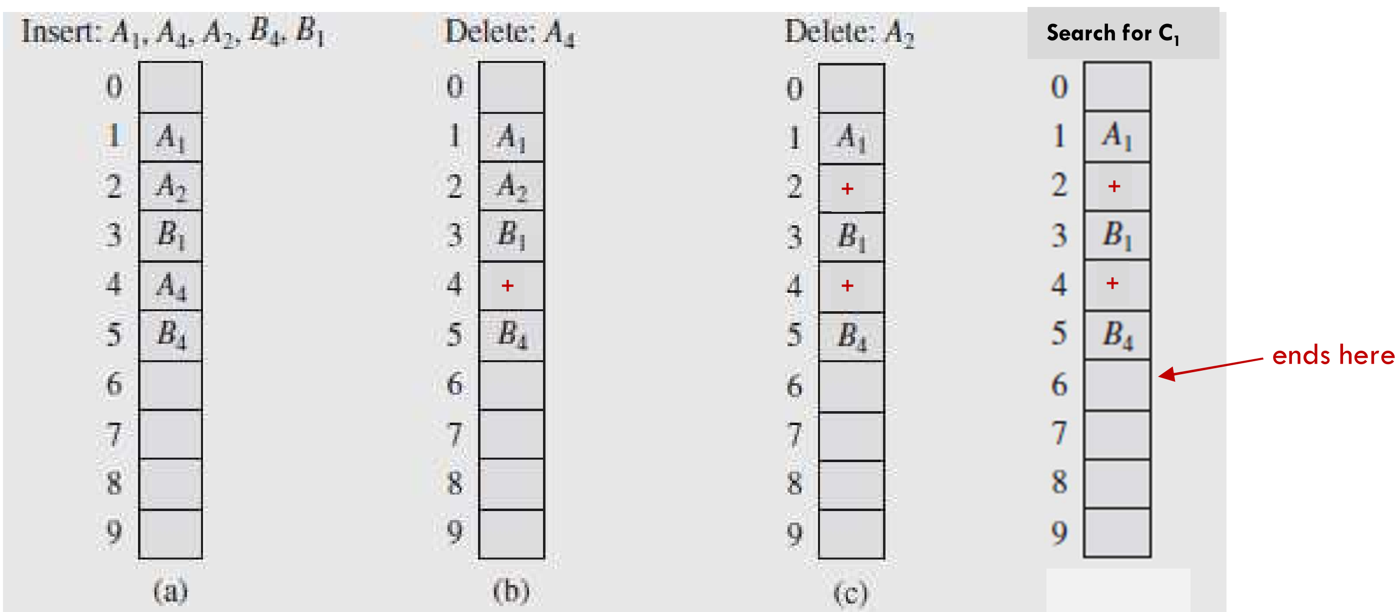
Adapted from Data Structures and Algorithms in C++, Fourth Edition (Drozdek)

Key Deletion in Closed Hashing

- While the previous example used linear probing, this deletion method applies to any closed hashing collision resolution policy
- The solution is to mark deleted keys with some type of indicator that the keys are not valid (often called a tombstone)
- In this way, searches for elements won't terminate prematurely
- When new keys are inserted, they can overwrite tombstones
- The drawback to this is when the table has far more deletions than insertions it will become overloaded with tombstones, slowing down search times, since every probe must be tested
- The table needs to be purged periodically by moving undeleted elements to cells occupied by deleted elements
- Those cells containing deleted elements not overwritten can then be marked as available

Marking Deleted Data

- When A_4 is deleted, a tombstone is placed at that index
- An attempt to find B_4 checks location 4, which has a tombstone, so the first probe is added to B_4's home position, and a search is made for B_4 at *that* location
- The situation after A_2 is deleted is similar, with respect to B_1
- Using this method, an item is declared "*not found*" only when a non-tombstone, empty location is found, or all viable probes are exhausted (i.e., the table is full)



Adapted from Data Structures and Algorithms in C++, Fourth Edition (Drozdek)

Rehashing

- If the table gets too full or has many tombstones, the running time will suffer and insertions might fail
- This can happen if there are too many removals mixed with insertions in the table
- A solution is

1. Build another hash table, that is about twice as large, with an associated new hash function
2. Iterate through the original hash table, computing the new hash value for each (non-tombstone) element and inserting it in the new table.

- There are algorithms for rehashing – Ex: *cuckoo* hashing

Rehashing Example

- Linear probing in use
- Order of insertion:
  13, 15, 24, 6
- After key 23 is added, the hash table is ~70% full, and rehashing is triggered (by user setting)
- New table size is prime number > 2m

0	6
1	15
2	23
3	24
4	
5	
6	13

m = 7
H(k) = k mod 7

Rehash →

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

m = 17
H(k) = k mod 17