CS32 Spring 2021

Week 9

TA: Manoj Reddy

LA: Katherine Zhang

Outline

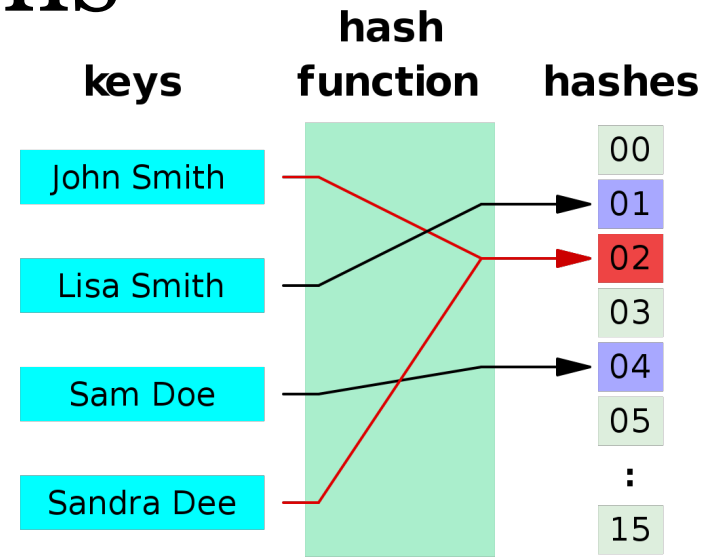
- HashTables

Hash Tables

- Motivation:
  - If balanced BST insert/lookup: O(log n)
  - Efficient way to search for data (*Ideally O(1) for insert/lookup*)
- Based on the concept of Hashing function
  - f(<input>) → <output>
  - input can be any sequence of bytes
  - output is a number with a defined range (but independent of bucket size)
- % (modulo operation/remainder) converts ‘output’ to the desired range
  - [0,n-1], if %n is performed

Properties of Hashing Functions

- Map from a larger to smaller space
- F(x) is always the same for given x
- F(x1) & F(x2) can hash to the same value
  - Collision (Many to one function)
- Properties of a good hashing functions
  - Quick to compute
  - Minimize collisions
  - Avalanche Effect
  - Hard to reverse-engineer (cryptography)
- Examples:
  - int: Identity function
  - string: Additive hash
- C++ has in-built hash functions
- Widely used:
  - Authentication
  - Network integrity

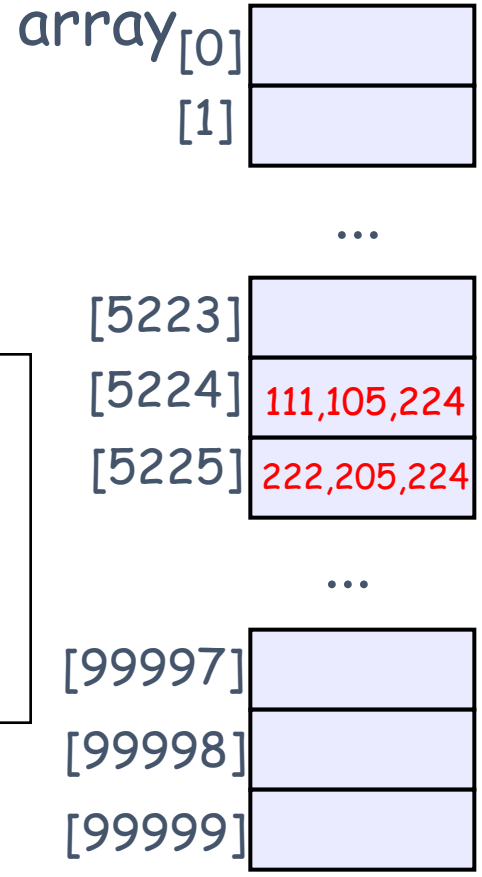


Algorithm and variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Max message size (bits)	Word size (bits)	Rounds	Bitwise operations	Collisions found	Example Performance (MiB/s)[8]
MD5 (as reference)		128	128	512	$2^{64} - 1$	32	64	and,or,xor,rot	Yes	335
SHA-0		160	160	512	$2^{64} - 1$	32	80	and,or,xor,rot	Yes	-
SHA-1		160	160	512	$2^{64} - 1$	32	80	and,or,xor,rot	Theoretical attack (2^{61})[9]	192
SHA-2	*SHA-224* *SHA-256*	224 256	256	512	$2^{64} - 1$	32	64	and,or,xor,shr,rot	None	139
SHA-2	*SHA-384* *SHA-512* *SHA-512/224* *SHA-512/256*	384 512 224 256	512	1024	$2^{128} - 1$	64	80	and,or,xor,shr,rot	None	154
SHA-3		224/256/384/512	1600 (5×5 array of 64-bit words)	1152/1088/832/576		64	24	and,xor,not,rot	None	

Closed Hash Table with Linear Probing

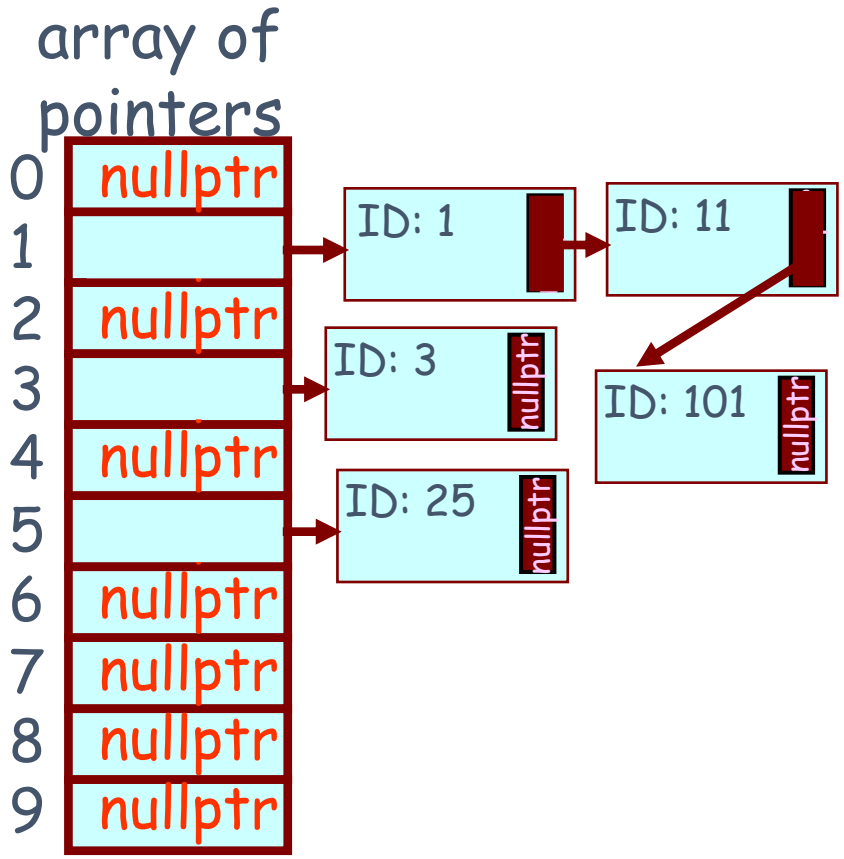
- Closed – Fixed size & Linear Probing – Keep searching until an empty bucket
- Lookup(x)
  - Hash x to get bucket number
  - Search linearly until empty bucket (wrap around)
- Insert(x)
  - Hash x to get bucket number
  - Insert value at the next available empty bucket
- Implementation
  - Array of struct Bucket
- Deletion possible, but not recommended
- What is the worse-case time complexity of lookup/insertion?

```
struct Bucket
{
   int val;
   …//Other data
   bool used;
};
```



Open Hash Table

- Able to handle any input size
- Allows deletions
- Each bucket is a LinkedList of stored items
- Lookup(x)
  - Hash x to get the appropriate bucket
  - Traverse through the entire LinkedList
- Insert(x)
  - Hash x to get the appropriate bucket
  - Insert into the LinkedList
- Delete(x)
  - Hash x to get the appropriate bucket
  - Delete from the LinkedList



Load Factor

- $L = \frac{\#\ of\ items}{\#\ of\ buckets}$
- Increasing the total number of buckets will decrease the load factor (L), hence decrease average number of tries
- Tradeoff: May end up wasting lots of memory
- Try to choose a prime number of buckets
  - More even distribution and fewer collision

Hash T…es

…ees

In fact, if you want to expand your hash table's size you basically have to create a whole new one*:

1. Allocate a whole new array with more buckets
2. Rehash every value from the original table into the new table
3. Free the original table

Speed		
Simplicity	Easy to implement	…plex to …ement
Max Size	Closed: Limited by array size Open: Not limited, but high load impacts performance	Unlimited size
Space Efficiency	Wastes a lot of space if you have a large hash table holding few items	Only uses as much memory as needed (one node per item inserted)
Ordering	No ordering (random)	Alphabetical ordering

unordered_map

- Hash based version of map
  - #include <unordered_map>
- Key is hashed
- Questions:
  - How would the struct look like?
  - Implement the insert and lookup methods

LA Worksheet