CPSC 131
Data Structures Concepts
Unordered Containers / Hash Tables

Dr. Anand Panangadan

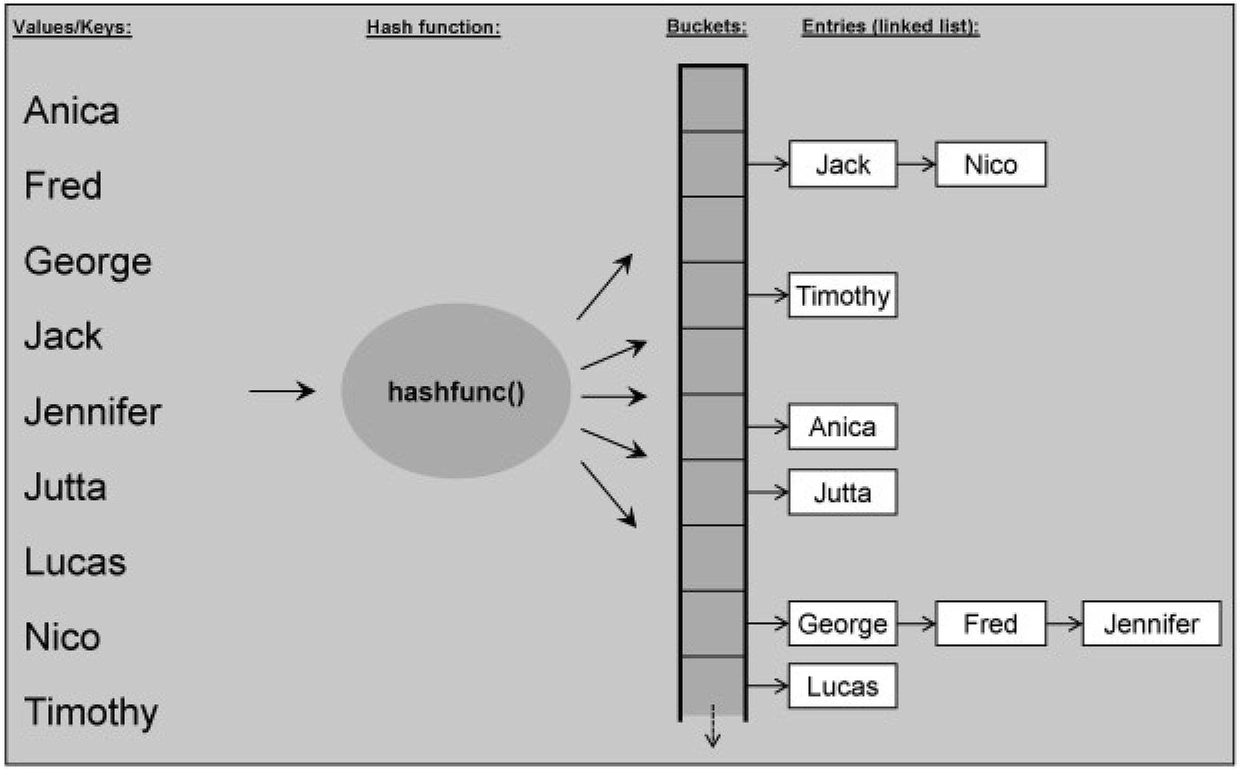
apanangadan@fullerton.edu

Dr. Thomas Bettens

TLBettens@fullerton.edu

Unordered Containers / Hash Tables

Josuttis, The C++ Standard Library



Elements have no defined order

Finding an element is faster than associative containers

Unordered Containers

Josuttis, The C++ Standard Library

- In unordered containers, elements have no defined order
  - If you insert three elements, they might have any order when you iterate over all the elements in the container.
  - If you insert a fourth element, the order of the elements previously inserted might change.
  - The only important fact is that a specific element is somewhere in the container.
  - Even when you have two containers with equal elements inside, the order might be different.
  - Think of it as like a bag.

Unordered Containers

Josuttis, The C++ Standard Library

- Unordered containers are typically implemented as a hash table.
  - Internally, the container is an array of linked lists.
- Using a hash function, the position of an element in the array gets processed.
  - The goal is that each element has its own position so that you have fast access to each element, provided that the hash function is fast.
  - Multiple elements might have the same position because such a fast perfect hash function is not always possible or might require that the array consumes a huge amount of memory.
  - For this reason, the elements in the array are linked lists so that you can store more than one element at each array position.

Unordered Containers

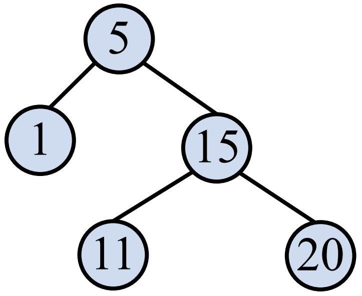
Josuttis, The C++ Standard Library

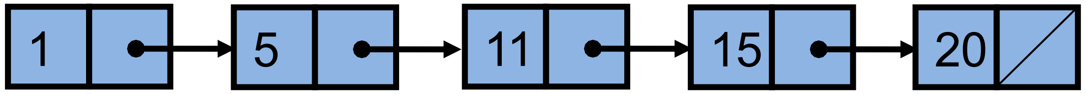
Major advantage of unordered containers

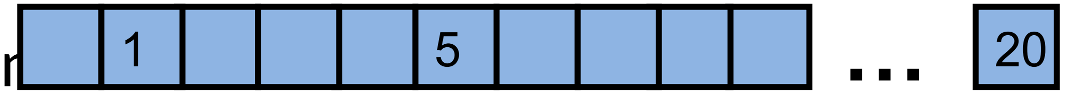
- Finding an element with a specific value is even faster than for associative containers.
  - The use of unordered containers provides amortized constant complexity, provided that you have a good hash function.
  - However, providing a good hash function is not easy

A constant time data structure?

- How long does it take to find an entry in a data structure?
  - Linked lists: O(n)
  - Balanced trees (AVL): O(log n)
- Can we get to O(1)?
- An array can get values in O(1) ***if***
  - Keys are the same as array index

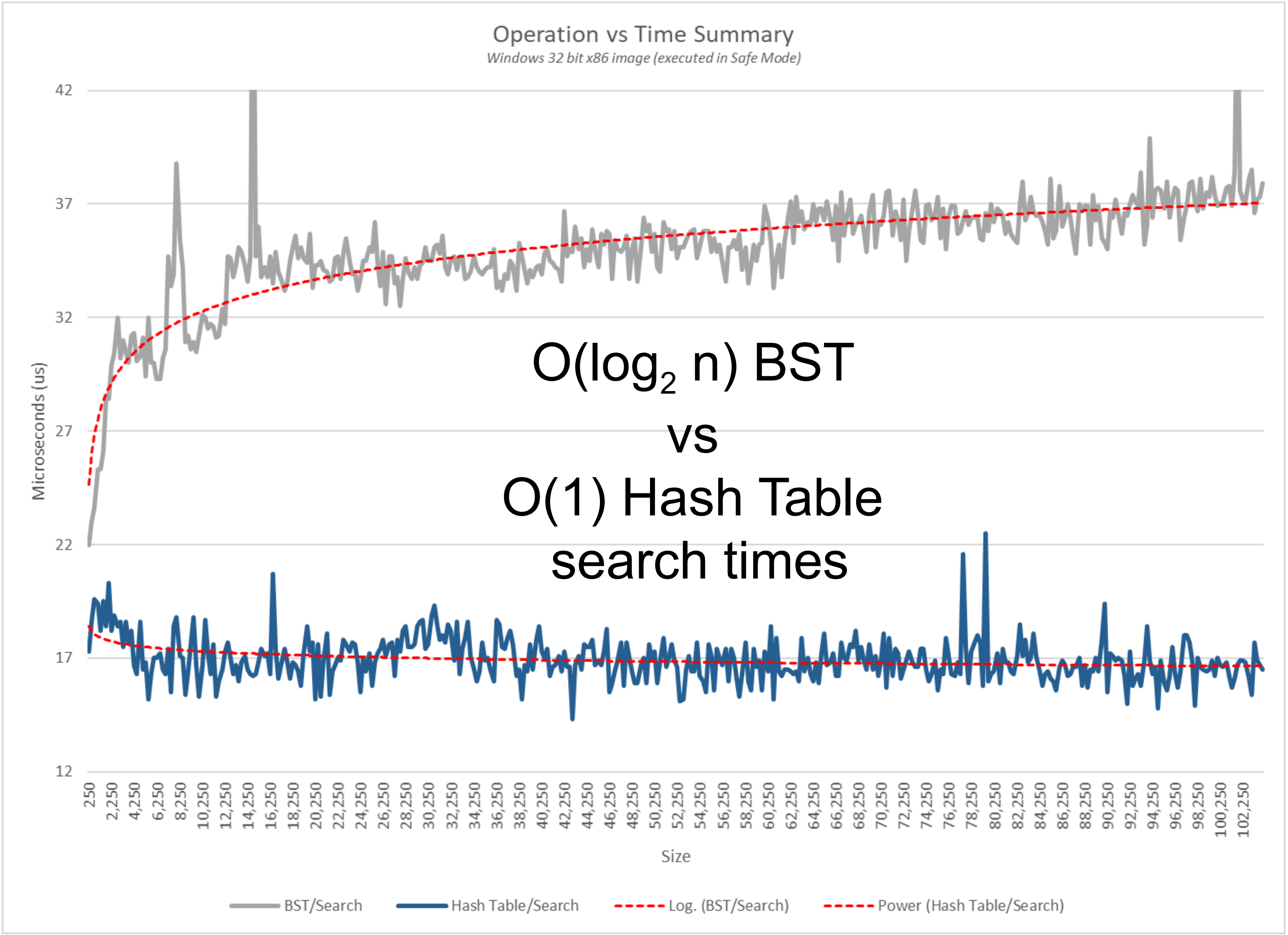






A constant time data structure?

- An array can get values in O(1) ***if***
    - Keys are the same as array index
- Disadvantages:
    - Requires keys be unique integers in the range 0,1,…, N-1
    - wastes a lot of space if the number of entries are much smaller than N
- The hash table is an attempt to reach O(1) by:
    - converting keys into codes, which may not be unique
    - compressing codes into indexes within a reduced storage space

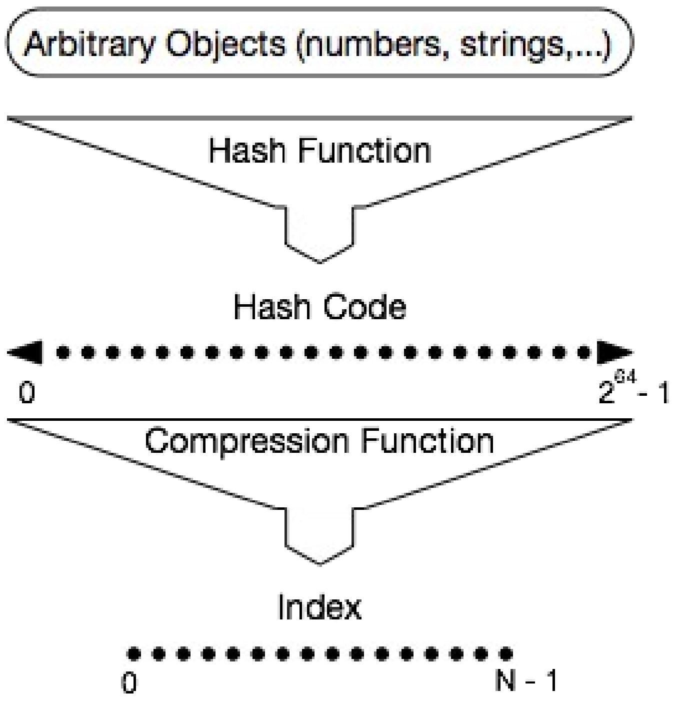


$O(\log_2 n)$ BST vs $O(1)$ Hash Table search times

Main idea

Convert any data type into an array index

Hashing function

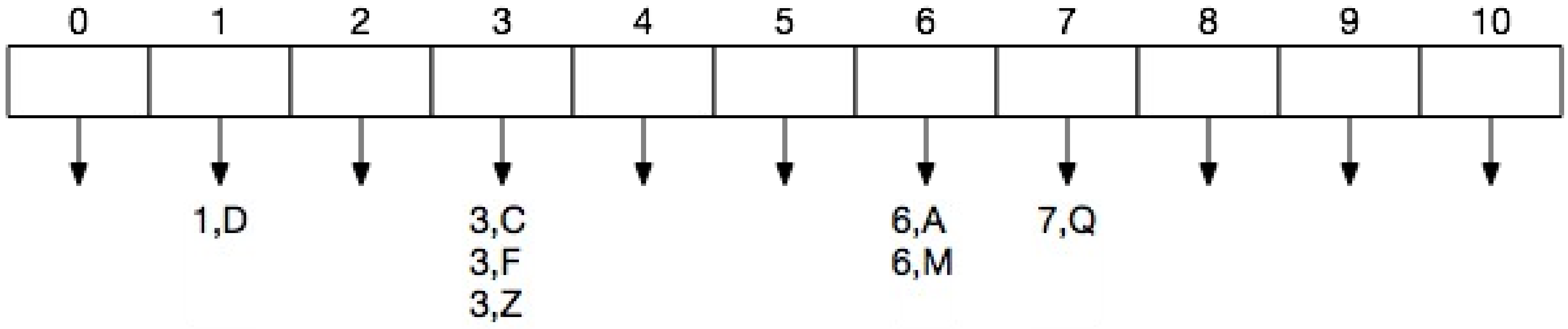


Hashtable

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

(Key, value) pairs

- Reminder:
  - Keys are associated with values
  - For simplicity, only showing keys in figures



Compression function

Division method

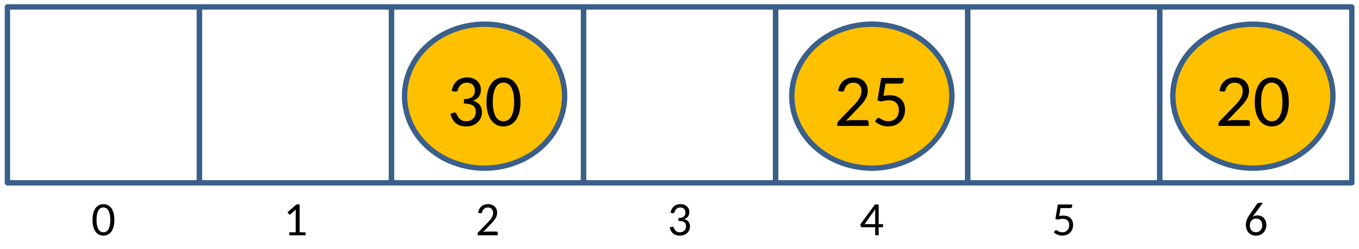
- Let N be the size of the array
- To get array index of hash code k, do
- Index = k % N
  - Take remainder after dividing k by N
- Simple and commonly used

Example:

- Let N=7
- Key 20 goes into array index 20 % 7 = 6
  - array[6] = 20

Compression function

Insert keys 20, 25, 30 into a table of size 7



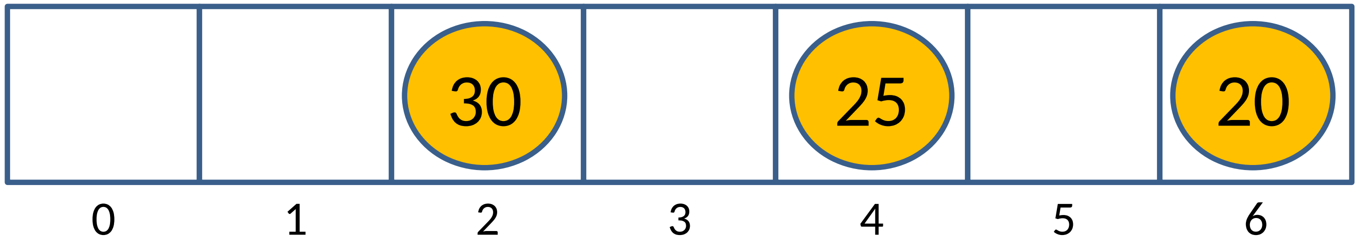
Compression function

Searching is similar to insert

search(25):

1. Calculate index(25) = 25%7 = 4
2. Look in array[4]

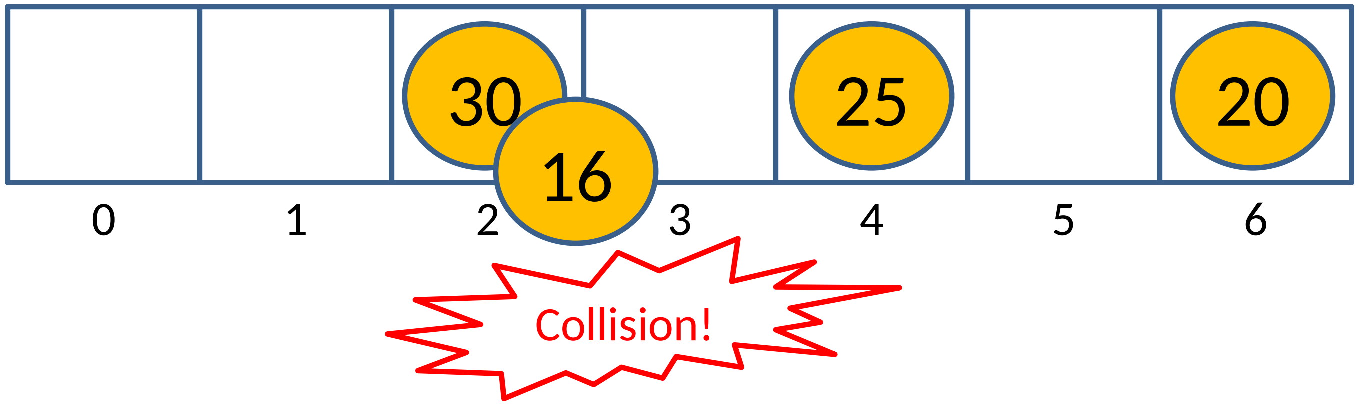
Cost of insert/search = O(1)



Compression function

Inserted keys 20, 25, 30 into a table of size 7

Insert key 16

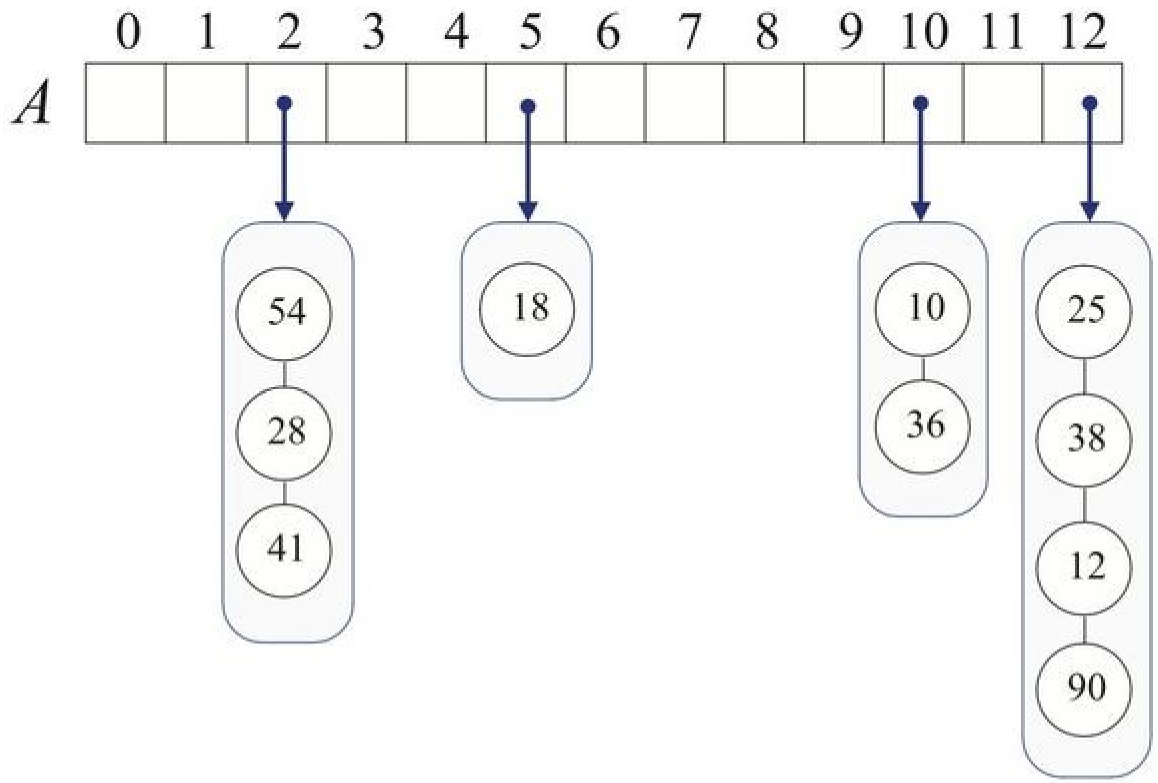


Collisions

- When two *different* keys get assigned to the *same* table index
  - 30 % 7 = 16 % 7 = 2
- Solutions to deal with collisions
  - Chaining
  - Probing
    - Linear probing
    - Quadratic probing
  - Prevent collisions completely – *Direct Hashing*

Chaining

Have each bucket hold a list (or a vector) of keys



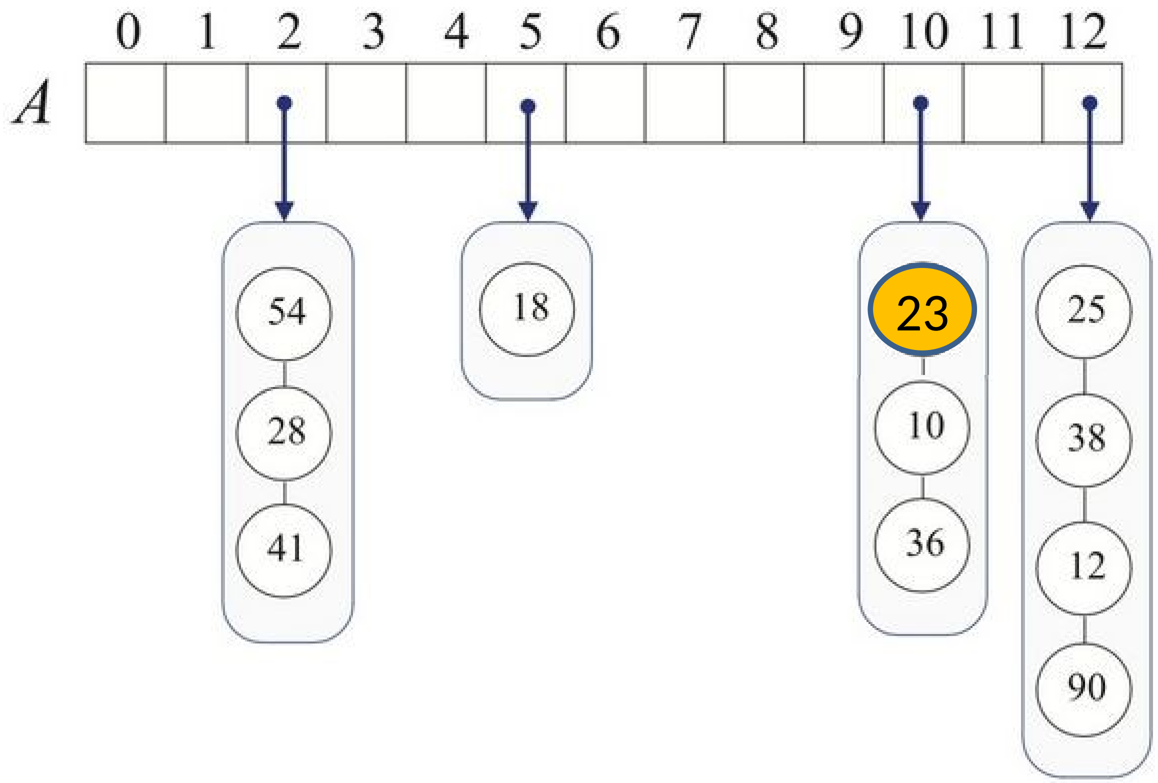
Chaining

Have each bucket hold a list (or a vector) of keys

- find(): calculate hash, search bucket’s list for key
- insert(): calculate hash, insert key into bucket’s list.
- remove(): calculate hash, remove key from bucket’s list

Chaining

Insert(23)? *Note that N=13*



Chaining

Have each bucket hold a list (or a vector) of keys

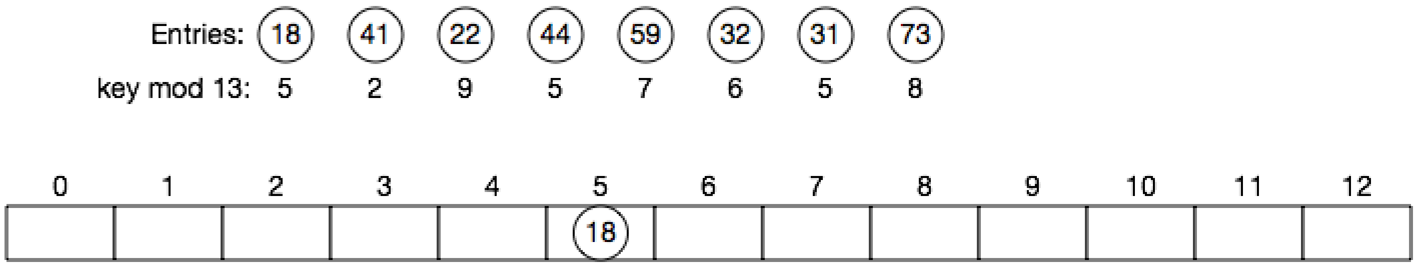
- find(): calculate hash, search bucket's list for key
  - O(n) worst-case
- insert(): calculate hash, insert key into bucket's list.
  - O(1) worst-case if duplicates allowed
  - O(n) worst-case if duplicates not allowed
- remove(): calculate hash, remove key from bucket's list
  - O(n) worst-case

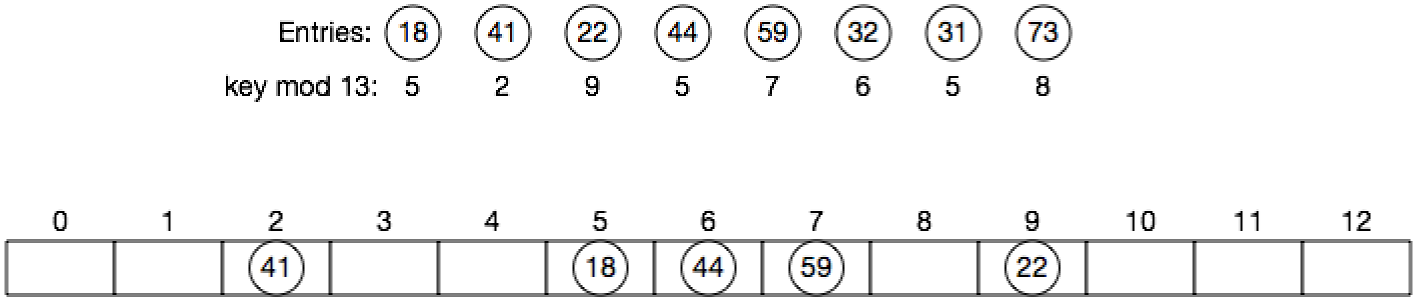
Linear probing

- Only have a single array
- If bucket is occupied, search forward for a free bucket
- Search is circular
    - when end of table is reached, wrap around to beginning
- Search fails if starting point is reached

Resolving collisions with linear probing

A hash table of size 13, compression function: key % 13




Entries: 18 41 22 44 59 32 31 73
key mod 13: 5 2 9 5 7 6 5 8
0 1 2 3 4 5 6 7 8 9 10 11 12
41
18
44
59
22

Values of “empty” cells

- Two kinds of empty cells
  - Empty since start (E1)
  - Empty after removal of an element (E2)
- Initially, all cells have value E1

Search with Linear Probing

- Consider a hash table A that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell (E1) is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    $i \leftarrow h(k)$
    $n \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c == \text{E1}$
            return not found
        else if $c.key() == k$
            return $c.value()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $n \leftarrow n + 1$
    until $n == N$
    return not found
```

Insert with Linear Probing

- insert(k, $value$)
  - We start at cell $h(k)$
  - We probe consecutive locations until an empty cell (E1 or E2) is found, or
    - N cells have been unsuccessfully probed

```
Algorithm insert(k, value)

    $i \leftarrow h(k)$
    $n \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c == \text{E1}$ or $\text{E2}$
            $A[i].\text{key} \leftarrow k$
            $A[i].\text{value} \leftarrow value$
            break
        else
            $i \leftarrow (i + 1) \bmod N$
            $n \leftarrow n + 1$
    until $n == N$
```

Remove with Linear Probing

- remove(***k***)
  - We start at cell ***h***(***k***)
  - We probe consecutive locations until until one of the following occurs
    - An item with key ***k*** is found, or
    - An empty cell (E1) is found, or
    - *N* cells have been unsuccessfully probed

```
Algorithm remove(k)
    i ← h(k)
    n ← 0
    repeat
        c ← A[i]
        if c == E1
            return not found
        else if c.key () == k
            A[i] ← E2
            return
        else
            i ← (i + 1) mod N
        n ← n + 1
    until n == N
```

Performance of Linear Probing

- Colliding items lump together, causing future collisions to cause a longer sequence of probes
- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
  - The worst case occurs when all the keys inserted into the map collide

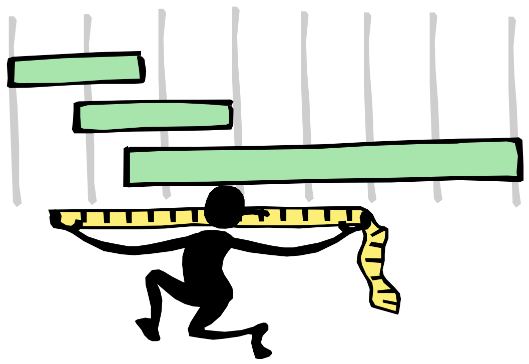
- Load factor of a hash table $\alpha = n/N$
  - How full is the hash table?

- The load factor affects the performance of a hash table
- Assuming that the hash values are like random numbers, the expected number of probes for an insertion with linear probing is
  $$1 / (1 - \alpha)$$

- Recommendation: keep $\alpha < 0.5$
  - At least half the table must be empty
- Then, expected cost < 1/(1-0.5) = O(1)

Performance of Hashing

- The expected running time of all the operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Recommendations:
  - keep $\alpha < 0.5$ for linear probing
  - keep $\alpha < 0.9$ for separate chaining
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches



Quadratic probing

- Same approach as linear probing but probe sequence is different
- Linear probing sequence ():
  - index = H, H+1, H+2, H+3, …

- Quadratic probing sequence:

  - c1 and c2 are constants that are given
    - For instance: c1=1, c2=1

Quadratic probing

- Example:
  - Let table size

- Probe sequence for key=45:

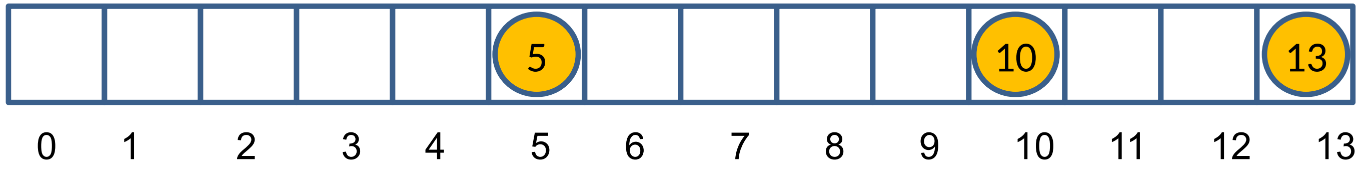
  - First probe:

  - Then probe:

  - Then probe:

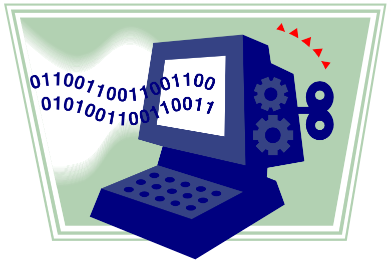
Direct Hashing

- Can prevent collisions completely, if:
  - Table is large enough that every key gets its own index
  - Array index = key
  - A true O(1) data structure



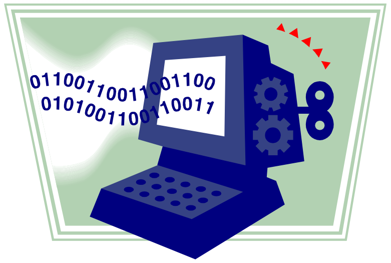
- Limitations of direct hashing
  1. All keys must be non-negative integers
  2. The hash table's size equals the largest key value plus 1, which may be very large

Hash Codes



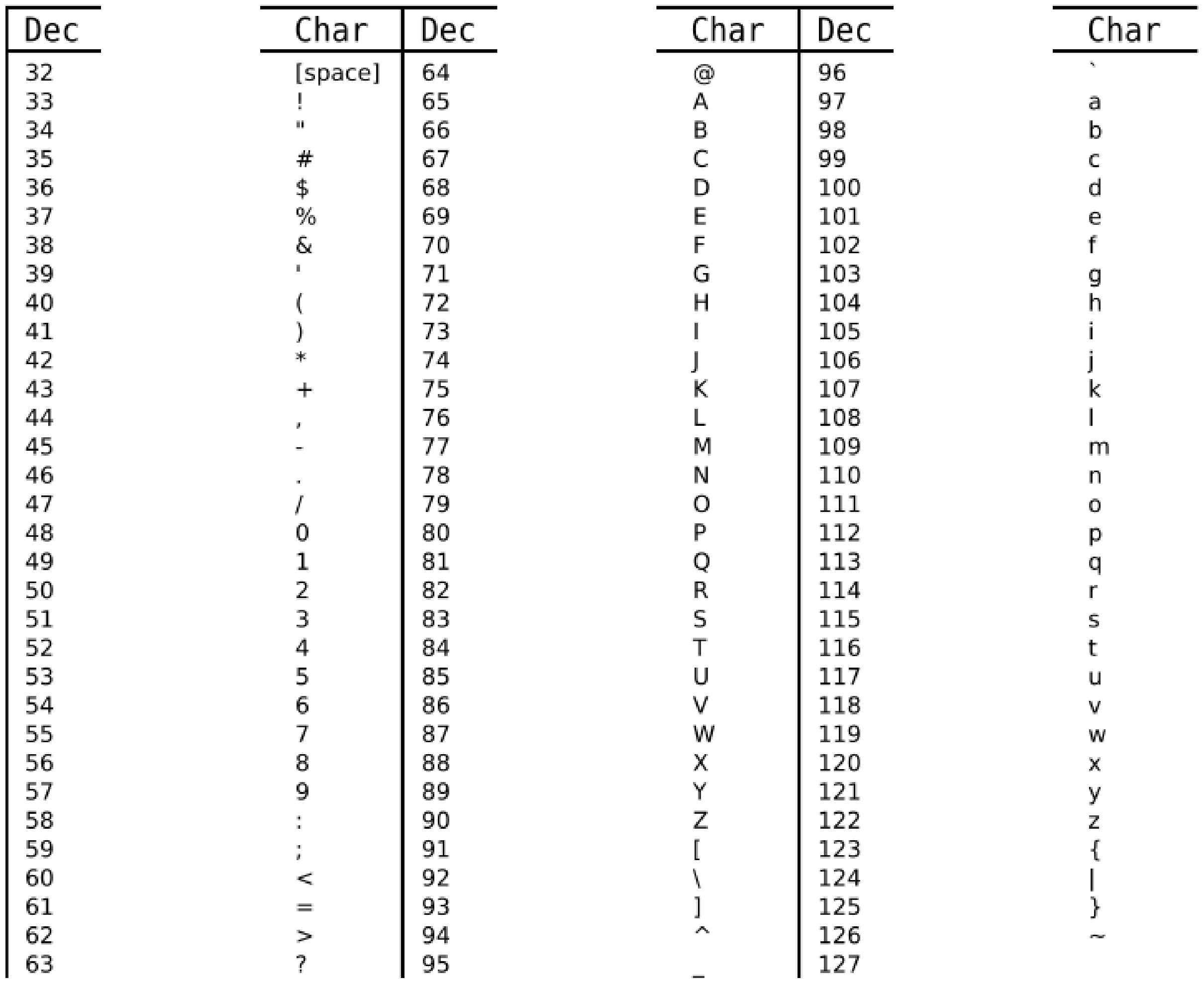
- How do we deal with non-integer keys?
  - Char
  - Strings
- The goal of a hash function is to “disperse” the keys in an apparently random way

Hash Codes



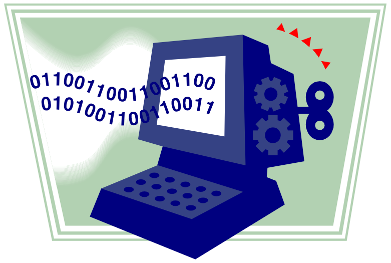
- ASCII code
  - A number for every character
  - “A” = 65; “B” = 66, “C” = 67, …, “Z”=90
  - “a” = 97; “b” = 98, “c” = 99, …, “z”=122

ASCII table



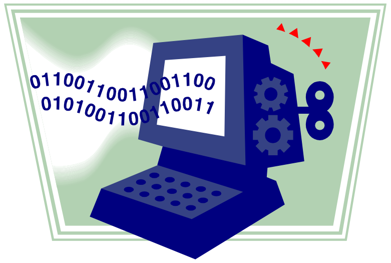
Dec	Char	Dec	Char	Dec	Char
32	[space]	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	

Hash Codes



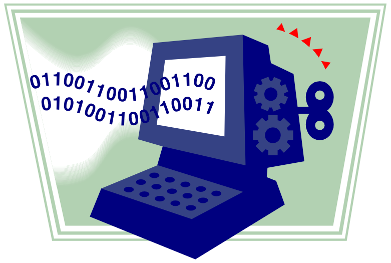
- Component sum
  - If the key is longer than an integer
  - Partition the key into components of fixed length (e.g., 16 or 32 bits) and sum the components (ignoring overflows)
  - Can we do this for strings?
    - “A” = 65
    - “AB” = 65 + 66 = 131
    - “ABE” = 65 + 66 + 69 = 200

Hash Codes



- Component sum for strings
  - “A” = 65
  - “AB” = 65 + 66 = 131
  - “ABE” = 65 + 66 + 69 = 200
- What is the issue?
  - Order does not matter
  - “RAMON” and “NORMA” have the same hash code

Hash codes



Polynomial accumulation

- Again, split key into components (x_0, x_1 ,…)
- But instead of just adding,
- Treat components as the coefficients of a polynomial:

$$x_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2} a + x_{k-1}$$

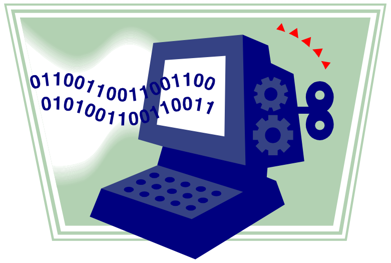
"a" is some number

- Easier to write code when rewritten like this:

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \ldots a(x_2 + a(x_1 + a\, x_0)) \ldots))$$

N	O	R	M	A
x_0	x_1	x_2	x_3	x_4

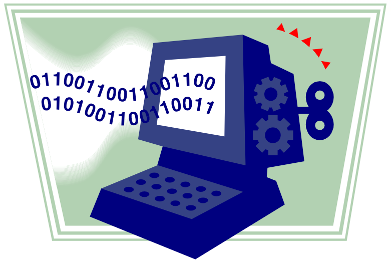
Hash codes



C++ implementation of Polynomial accumulation

```
size_t polynomial_hash (string word) {
    const int a = 33;
    size_t hash = 0;
    for (int i = 0; i < word.size(); i++)
       hash = hash*a + word[i];
    return hash;
}
```

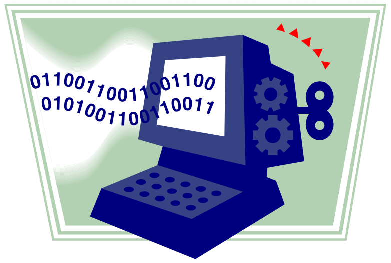
Hash codes



Polynomial accumulation

- Especially suitable for strings
- Choice of $a = 33$ gives at most 6 collisions on a set of 50,000 English words

Hash Codes



- Integer cast
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type
    - byte, short, int, float in C++

hash table in C++

- Usage similar to a map

```
#include <unordered_map>

std::unordered_map<std::string, double>  gpaRecord;

gpaRecord["Allen"] = 3.42;    // new element inserted
gpaRecord["Beth" ] = 3.5;     // new element inserted
cout << gpaRecord["Allen"];   // existing element read
```

- "unordered_map containers are faster than map containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements."

hash table in C++

- `std::unordered_map`
- Using the operator[key] automatically inserts (key, default value) if key is not found!
  - Same as in std::map
  - Check for key using find() and end iterator

```
std::unordered_map<std::string, double>  gpaRecord;

if( gpaRecord.find("Allen") != gpaRecord.end() )
{
  cout << gpaRecord[“Allen"];
}
```