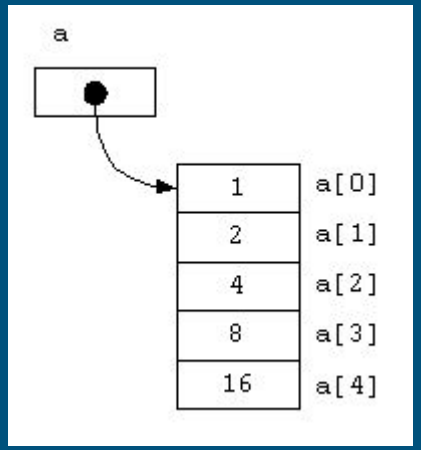
Arrays & Hash Maps

Kevin Yu

Arrays

Array is a data structure consisting of a collection of elements, of same memory size, each identified by at least one array index.




```
// define and instantiate an array
int* arr = new int[n];

// Access and modify the element in the array
arr[0] = 10;
arr[1] = 20;
```

Static vs Dynamic (resizable) Arrays

Static Array

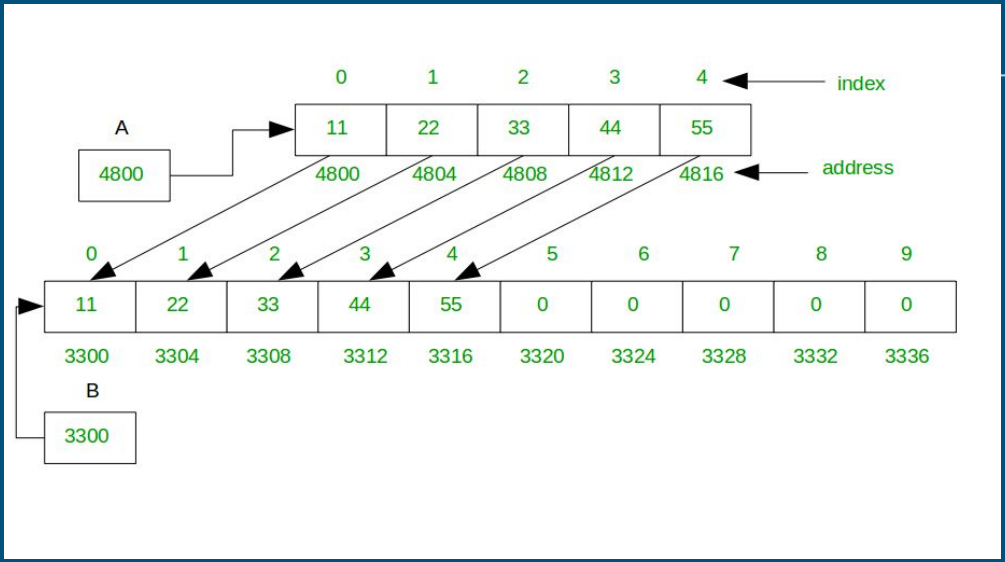
- Fixed size

Dynamic Arrays:

- Dynamic size
- Aka, vectors in C++

```
// declare and instantiate an array
int arr[5] = {1, 2, 3, 4, 5};


// declare and initialize a vector
#include <vector>

std::vector<int> v = {1, 2, 3, 4, 5};

// updating an element in a vector
v[1] = 20;

// safer version
v.at(1) = 20;      // throws error if index is invalid
```

How do dynamic (resizable) arrays work?



Size - number of elements currently stored

Capacity - maximum number of elements it can hold

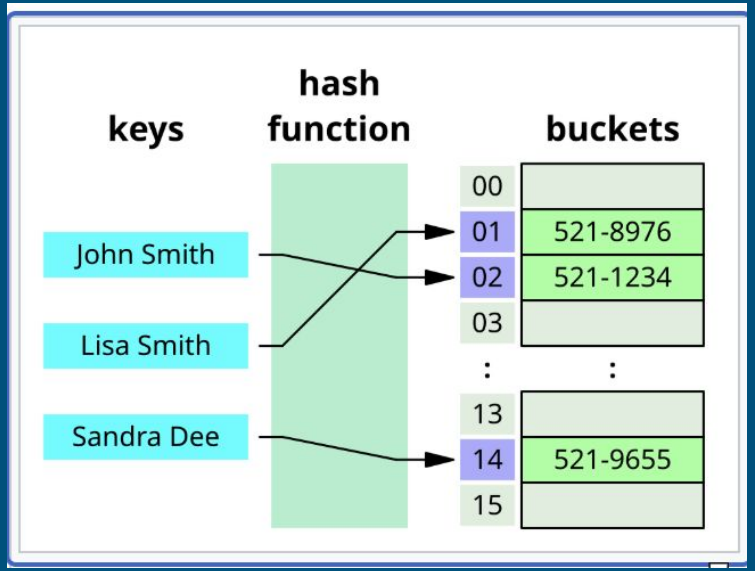
growth factor - multiplicative value used when resizing the underlying array. Usually it is 2.

Vector’s Operation Time Complexity

Operation	Syntax	Complexity
Access	`v[i]` / `v.at(i)`	O(1)
Add at end	`v.push_back(x)`	Amortized O(1)
Add at index	`v.insert(v.begin()+i, x)`	O(n)
Remove last	`v.pop_back()`	O(1)
Remove at index	`v.erase(v.begin()+i)`	O(n)

Hash Maps

A hash map is a data structure that stores **key-value pairs**. It works like a dictionary or map, where each key is linked to a value. It uses array underneath.



Hash Maps

```cpp
#include <unordered_map>
#include <string>

// declare the hash map
std::unordered_map<std::string, int> mp;

mp["apple"] = 5;   // add key-value pair
int x = mp["apple"]; // access value by key, x = 5
```

Hash Sets

Hash sets are similar to hash maps, but they only have value and don’t have keys.

```cpp
#include <iostream>
#include <unordered_set>
#include <string>

int main()
{
    std::unordered_set<std::string> fruits = {"apple", "banana", "orange"};

    fruits.insert("mango"); // add element
    fruits.erase("banana"); // remove element

    if (fruits.count("apple")) // check existence
        std::cout << "Apple is available.\n";

    // iterate over elements
    for (const auto &fruit : fruits)
        std::cout << fruit << " ";
}
```

How does hashing work?

Unicode:

Unicode = One Big Dictionary
"This letter or character = this number"

Example of a basic hash function

Algorithm -
1. Sum up all the unicode value of the key
2. Get the "index" by taking the modulo of sum of key

Example:
Put the key-value pair in below map
key = "bob", value = 67
Map size = 10

1. Get sum of all the unicode value of key
   "b" = 98 in unicode
   "o" = 111 in unicode
   sum = 98 + 111 + 98 = 307
2. Get the "index" by taking the modulo
   index = sum % map_size
   index = 307 % 10 = 7

Question
Where would you place the below key-value pair?

"alice" : 95

symbol

number representation

Code	Glyph	Decimal	Octal	Description
U+0061	a	97	0141	Latin Small Letter A
U+0062	b	98	0142	Latin Small Letter B
U+0063	c	99	0143	Latin Small Letter C
U+0064	d	100	0144	Latin Small Letter D
U+0065	e	101	0145	Latin Small Letter E
U+0066	f	102	0146	Latin Small Letter F
U+0067	g	103	0147	Latin Small Letter G
U+0068	h	104	0150	Latin Small Letter H
U+0069	i	105	0151	Latin Small Letter I
U+006A	j	106	0152	Latin Small Letter J
U+006B	k	107	0153	Latin Small Letter K
U+006C	l	108	0154	Latin Small Letter L
U+006D	m	109	0155	Latin Small Letter M
U+006E	n	110	0156	Latin Small Letter N
U+006F	o	111	0157	Latin Small Letter O
U+0070	p	112	0160	Latin Small Letter P
U+0071	q	113	0161	Latin Small Letter Q
U+0072	r	114	0162	Latin Small Letter R
U+0073	s	115	0163	Latin Small Letter S
U+0074	t	116	0164	Latin Small Letter T
U+0075	u	117	0165	Latin Small Letter U
U+0076	v	118	0166	Latin Small Letter V
U+0077	w	119	0167	Latin Small Letter W

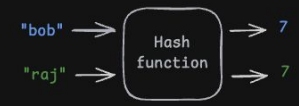
Collisions

Example of a basic hash function

Algorithm -
1. Sum up all the unicode value of the key
2. Get the "index" by taking the modulo of sum of key

Example:
Put the key-value pair in below map
key = "raj", value = 55
Map size = 10

1. Get sum of all the unicode value of key
   "r" = 114
   "a" = 97
   "j" = 106
   sum = 114 + 97 + 106 = 317
2. Get the "index" by taking the modulo
   index = sum % map_size
   index = 317 % 10 = 7

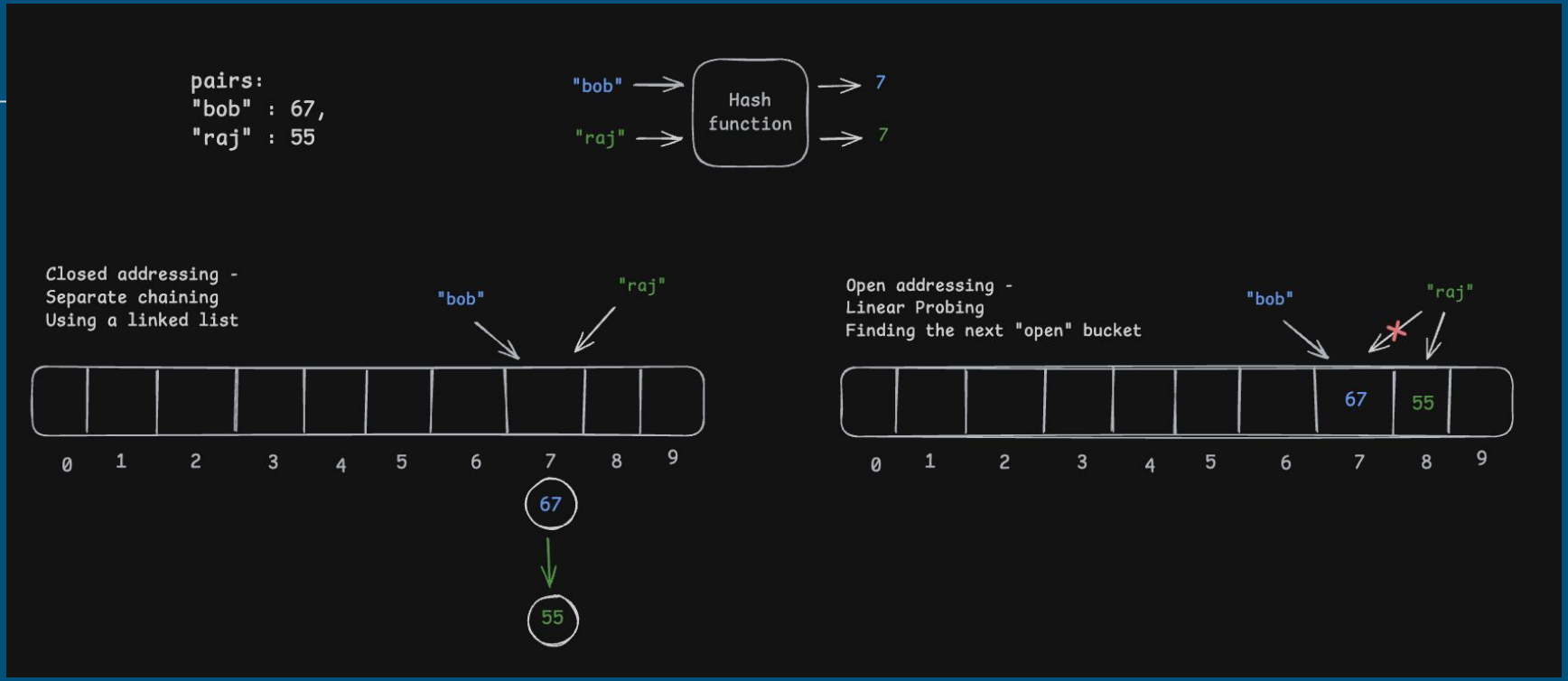


symbol

number representation

Code	Glyph	Decimal	Octal	Description
U+0061	a	97	0141	Latin Small Letter A
U+0062	b	98	0142	Latin Small Letter B
U+0063	c	99	0143	Latin Small Letter C
U+0064	d	100	0144	Latin Small Letter D
U+0065	e	101	0145	Latin Small Letter E
U+0066	f	102	0146	Latin Small Letter F
U+0067	g	103	0147	Latin Small Letter G
U+0068	h	104	0150	Latin Small Letter H
U+0069	i	105	0151	Latin Small Letter I
U+006A	j	106	0152	Latin Small Letter J
U+006B	k	107	0153	Latin Small Letter K
U+006C	l	108	0154	Latin Small Letter L
U+006D	m	109	0155	Latin Small Letter M
U+006E	n	110	0156	Latin Small Letter N
U+006F	o	111	0157	Latin Small Letter O
U+0070	p	112	0160	Latin Small Letter P
U+0071	q	113	0161	Latin Small Letter Q
U+0072	r	114	0162	Latin Small Letter R
U+0073	s	115	0163	Latin Small Letter S
U+0074	t	116	0164	Latin Small Letter T
U+0075	u	117	0165	Latin Small Letter U
U+0076	v	118	0166	Latin Small Letter V
U+0077	w	119	0167	Latin Small Letter W

Collisions Resolution



Hash Map’s Operation Time Complexity

Operation	Average Case	Worst Case	Notes
Insert (put)	O(1)	O(n)	Average O(1) due to direct index access, but worst case O(n) if many keys collide (all go in same bucket).
Search (get / containsKey)	O(1)	O(n)	Same reasoning as insert. Collisions increase lookup time.
Delete (remove)	O(1)	O(n)	Need to locate the key first; average O(1), worst case O(n) if all keys are in same bucket.

Hash Maps vs Dynamic Arrays (Vectors)

Operation	`vector`	`unordered_map` (average case)
Access by index/key	O(1)	O(1)
Search for value/key	O(n)	O(1)
Insert at end / by key	O(1) amortized	O(1)
Insert at beginning / middle	O(n)	O(1)
Delete at end / by key	O(1)	O(1)
Delete at beginning / middle	O(n)	O(1)
Iteration over all elements	O(n)	O(n)
Memory layout	Contiguous	Spread across buckets
Ordered?	Yes	No (unordered)

Thank You