ALGORITHMS AND DATA STRUCTURES

HASH TABLES

IMPLEMENTATION OF ASSOCIATIVE CONTAINERS

- What data structure is more suitable to implement a version of these containers?
  - `std::set<data_type>`
  - `std::map<key_type, value_type>`

IMPLEMENTATION OF ASSOCIATIVE CONTAINERS

- What data structure is more suitable to implement a version of these containers?
  - `std::set<data_type>`
  - `std::map<key_type, value_type>`
- What are the best complexities we can aspire to?
  - Quadratic, linear, linearithmic, logarithmic, constant…

ARRAY-BASED IMPLEMENTATION OF MAP

- Let us say you want to implement a version of `std::map<key_type, value_type>` using an array
  - How to go about implementing a `map<string, string>`, for instance?
  - Basic operations
    - `find`, `insert`, and `remove`

```
template <typename KeyType, typename ValueType>
class Map {
public:
    Map();
    ~Map();
    int size();
    bool isEmpty();
    void clear();
    void put(KeyType key, ValueType value);
    ValueType get(KeyType key);
    bool containsKey(KeyType key);

private:
    // private stuff
};
```

```
template <typename KeyType, typename ValueType>
class Map {
public:
    // public stuff

private:
    struct KeyValuePair {
        KeyType key;
        ValueType value;
    };

    KeyValuePair *array;
    int capacity;
    int count;

    void expandCapacity();
    int findKey(KeyType key);
};
```

```
template <typename KeyType, typename ValueType>
void Map<KeyType,ValueType>::put(KeyType key,
                                 ValueType value) {
    int index = findKey(key);
    if (index == -1) {
        if (count == capacity) expandCapacity();
        index = count++;
        array[index].key = key;
    }
    array[index].value = value;
}

template <typename KeyType, typename ValueType>
ValueType Map<KeyType,ValueType>::get(KeyType key) {
    int index = findKey(key);
    if (index == -1) error("get: No value for key");

    return array[index].value;
}
```

```
template <typename KeyType, typename ValueType>
bool Map<KeyType,ValueType>::containsKey(KeyType key) {
    return findKey(key) != -1;
}

// all relevant methods depend on this method
template <typename KeyType, typename ValueType>
int Map<KeyType,ValueType>::findKey(KeyType key) {
    for (int i = 0; i < count; i++) {
        if (array[i].key == key) return i;
    }
    return -1;
}
```

ARRAY-BASED IMPLEMENTATION OF MAP

- All of the most relevant methods rely on `findkey()`
  - `findkey()`: linear search method, which makes
    - `get`, `put`, and `containsKey` all $O(N)$
  - binary search could improve `get` and `containsKey` to $O(\log N)$, but not `put` that is still $O(N)$, why?
- We need to propose a more efficient strategy
  - one way to do that is with lookup tables…

LOOKUP TABLES

- are programming structures to obtain a desired value in a quick lookup, **quick?**
  - by computing the appropriate index in a table
  - are typically an array or a grid
- replaces runtime computation with a simpler array indexing operation
- Example: thumb-tabs in a dictionary or simple lookup array

LOOKUP TABLES

- **Example**: Translate two-letter codes into city names
    - What data structure to use?
        - Take advantage of the two-letter structure of the key

AK	Alaska	HI	Hawaii	ME	Maine	NJ	New Jersey	SD	South Dakota
AL	Alabama	IA	Iowa	MI	Michigan	NM	New Mexico	TN	Tennessee
AR	Arkansas	ID	Idaho	MN	Minnesota	NV	Nevada	TX	Texas
AZ	Arizona	IL	Illinois	MO	Missouri	NY	New York	UT	Utah
CA	California	IN	Indiana	MS	Mississippi	OH	Ohio	VA	Virginia
CO	Colorado	KS	Kansas	MT	Montana	OK	Oklahoma	VT	Vermont
CT	Connecticut	KY	Kentucky	NC	North Carolina	OR	Oregon	WA	Washington
DE	Delaware	LA	Louisiana	ND	North Dakota	PA	Pennsylvania	WI	Wisconsin
FL	Florida	MA	Massachusetts	NE	Nebraska	RI	Rhode Island	WV	West Virginia
GA	Georgia	MD	Maryland	NH	New Hampshire	SC	South Carolina	WY	Wyoming

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

	A	B	C	D	E	F	G	H	I	. . .
A										0
B										1
C	California									2
D					Delaware					3
E										4
F										5
G	Georgia									6
H									Hawaii	7
I	Iowa			Idaho						8
J										9
K										10
L	Louisiana									11
M	Massachusetts			Maryland	Maine				Michigan	12
N			North Carolina	North Dakota	Nebraska			New Hampshire		13
O								Ohio		14
P	Pennsylvania									15
Q										16
R									Rhode Island	17
S			South Carolina	South Dakota						18
T										19
U										20
V	Virginia									21
W	Washington								Wisconsin	22
X										23
Y										4
Z										25
	0	1	2	3	4	5	6	7	8	

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

LOOKUP TABLES

- We could use `map<string, string>`
  - Array-based implementation?
  - Try a lookup table!
    - consider the two-letter code ‘XY’ as coordinates `(X,Y)` in a two-dimensional array that contains city names
    - use X and Y as indices in a two-dimensional grid
    - remember to convert letters (`char`s) to integers

LOOKUP TABLES

- Works the same as looking up in an array
  - Simple arithmetics and then looking up
  - It then runs in constant time $O(1)$
- How to generalize beyond two-letter codes?

LOOKUP TABLES

- How to improve this piece of code?

```
std::string getStateName(string key,
                         Grid<string> & grid) {
    char row = key[0] - 'A';
    char col = key[1] - 'A';
    if (!grid.inBounds(row, col) ||
        grid[row][col] == "") {
        error("No state name for " + key);
    }
    return grid[row][col];
}
```

- How to generalize beyond two-letter codes? **Hashing!**

HASH TABLES

- are abstract data structures that allows to perform
  - `find`
  - `insert`
  - `remove`

  $\left.\right\}$ operations in $O(1)$ average time
- are associative arrays that maps `keys` to `values`
- can be used to efficiently implement a `map`

HASHING

- is a technique used for performing insertions, deletions, and searches in *constant average time*
- is a computational strategy that operates as follows
  1. Take a function (**hash function**) to transform a key into an integer (**hash code** [of that key])
  2. Use the hash code as a starting point as you search for a matching key in the table.
- A **hash table** is an implementation of `map` that uses hashing

HASH TABLES

- can be considered as a generalization of arrays
- is a data structure of fixed size that contains a collection of items
- can be implemented using `vectors`, pointers combined with `trees`, `lists`, etc.
- come in different flavors
  - chaining hash tables (today)
  - probing hash tables (exercise)

HASH TABLES: CHAINING

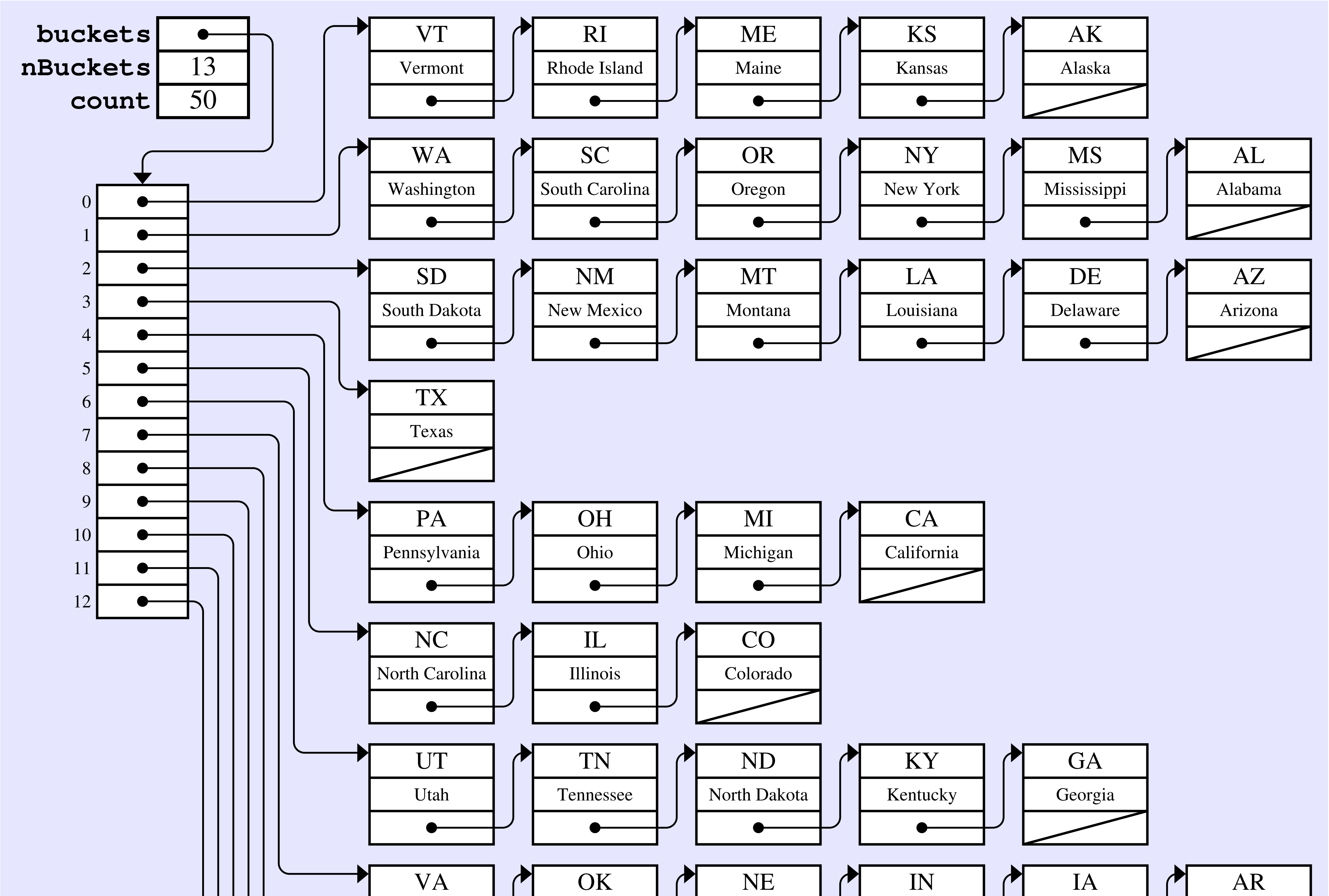
- Each hash code serves as an index into an array of linked lists
  - each list is called a **bucket**
- The right bucket is found by invoking the hash function to the key

```
int bucket = hashCode(key) % nBuckets;
```

- `bucket` is an index to the array of lists (the key hashes to a bucket)
- When two or more different keys hash to the same bucket is called **collision**

HASH TABLES: CHAINING

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.



```
template <typename KeyType, typename ValueType>
class HashMap {
   struct Cell {
        KeyType key;
        ValueType value;
        Cell *link;
    };

    Cell **buckets;
    int nBuckets;
    int count;

    Cell *findCell(int bucket, ValueType key) {
        Cell *cp = buckets[bucket];
        while (cp != NULL && key != cp->key)
            cp = cp->link;
        return cp;
    }
};
```

```
const int INITIAL_BUCKET_COUNT = 101;

template <typename KeyType, typename ValueType>
HashMap<KeyType,ValueType>::HashMap() {
    nBuckets = INITIAL_BUCKET_COUNT;
    buckets = new Cell*[nBuckets];

    for (int i = 0; i < nBuckets; i++)
        buckets[i] = NULL;

    count = 0;
}

template <typename KeyType, typename ValueType>
HashMap<KeyType,ValueType>::~HashMap() {
    clear();
}
```

```
template <typename KeyType, typename ValueType>
void HashMap<KeyType,ValueType>::clear() {
    for (int i = 0; i < nBuckets; i++) {
        Cell *cp = buckets[i];

        while (cp != NULL) {
            Cell *oldCell = cp;
            cp = cp->link;
            delete oldCell;
        }
    }
    count = 0;
}
```

```
template <typename KeyType, typename ValueType>
void HashMap<KeyType,ValueType>::put(KeyType key,
                                     ValueType val) {
    int bucket = hashCode(key) % nBuckets;
    Cell *cp = findCell(bucket, key);

    if (cp == NULL) {
        cp = new Cell;
        cp->key = key;
        cp->link = buckets[bucket];
        buckets[bucket] = cp;
        count++;
    }
    cp->value = val;
}
```

HASH MAP BY CHAINING: IMPLEMENTATION

```
template <typename KeyType, typename ValueType>
ValueType HashMap<KeyType,ValueType>::get(KeyType key)
{
    int bucket = hashCode(key) % nBuckets;
    Cell *cp = findCell(bucket, key);
    if (cp == NULL) error("get: no value for key");

    return cp->value;
}

template <typename KeyType, typename ValueType>
bool HashMap<KeyType,ValueType>::containsKey(KeyType key)
{
    int bucket = hashCode(key) % nBuckets;

    return findCell(bucket, key) != NULL;
}
```

NUMBER OF BUCKETS

- As the hash function, also change the probability of collisions
  - if `nBuckets` is small $\longmapsto$ more collisions
  - if `nBuckets` is large $\longmapsto$ waste resources
- It is key to compare to the number of entries
- **Load factor** of the hash table: $\lambda = \frac{N_{\text{keys}}}{N_{\text{buckets}}} = \frac{n}{m}$

`time-space tradeoff`

NUMBER OF BUCKETS

- A nice sweet spot for $\boldsymbol{\lambda \approx 0.7}$ leading to $O(1)$ operations
- Most of the times it is not possible to determine the number of entries
  - Empirically obtain an estimate for `nBuckets`
- **Rehashing**: Dynamically change `nBuckets` as λ grows too large (exercise)
  - might impact performance but not too much if done sparsely

HASH TABLES: CHAINING

- **Theorem 1**: In a hash table in which collisions are resolved by chaining, an *unsuccessful* search takes $\Theta(1 + \lambda)$ average-case time, under the assumption of simple uniform hashing and constant-time hash code
- **Theorem 2**: In a hash table in which collisions are resolved by chaining, a *successful* search takes $\Theta(1 + \lambda)$ average-case time, under the assumption of simple uniform hashing and constant-time hash code

Simple uniform hashing:
Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of simple uniform hashing.

HASH TABLES: CHAINING

- Meaning: If $n = O(m) \rightarrow \alpha = O(1)$
  - Searching takes constant time *on average*
  - Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time (for doubly LL)
  - **ALL** dictionary operations in $O(1)$ time *on average*
- What is the difference with lookup tables, in terms of space and time complexities?

HASH FUNCTION: HASHCODE

- lies at the heart of the implementation
  1. must be computable in constant time (i.e., independent of the number of items in the hash table)
  2. must distribute its items uniformly among the array slots or buckets (sample uniform hashing)
- does not have to necessarily take advantage of all the information provided by the key

HASH FUNCTION: HASHCODE

- Universal hash functions guarantee a uniformly distributed occupation of buckets
- is usually defined as a free function, not part of the `HashMap` interface
- has to be as unpredictable as possible for a given set of keys

HASH FUNCTION: HASHCODE

- Every hash function will profoundly affect the efficiency of the implementation
- Although the correctness of the implementation is not affected by the unpredictability of the hash function
- It is often a good idea to choose the size of the table to be a primer number

HASH FUNCTION: HASHCODE

- Unpredictable, less collisions

```
const int HASH_SEED = 5381;
const int HASH_MULTIPLIER = 33;
const int HASH_MASK = unsigned(-1) >> 1;

int hashCode(string str) {
    unsigned hash = HASH_SEED;
    int n = str.length();
    for (int i = 0; i < n; i++)
        hash = HASH_MULTIPLIER * hash + str[i];

    return int(hash & HASH_MASK);
}
```

HASH FUNCTION: HASHCODE

- This one would produce more collisions

```
int hashCode(string str) {
    int hash = 0;
    int n = str.length();

    for (int i = 0; i < n; i++)
        hash += str[i];

    return hash;
}
```

```
act, cat, tac
```