DATA STRUCTURES AND ALGORITHMS

Lecture 14

Learning outcome 6

Dictionaries

- Dictionary (associative array, map, symbol table) is a container that contains a collection of pairs (key, value) and which provides operations:
  - Adding a new pair
  - Pair removal
  - Modify the value of an existing pair (but not the key)
  - Retrieving value by the key (emphasis)
- Some programming languages (Python) have built-in types
- The two main directions of dictionary implementation are:
  - Hash tables
  - Binary search trees (learning outcome 4)

Comparison of dictionary implementations

- Emphasis on quick search, insert and delete

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	O(1)	O(*n*)	O(1)	O(*n*)	O(1)	O(*n*)	No
Self-balancing binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	Yes
unbalanced binary search tree	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	Yes
Sequential container of key-value pairs (e.g. association list)	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)	O(*n*)	No

Dictionaries by trees	Dictionaries by hash tables
Items are sorted	Items are not sorted
Consumes less memory	Consumes more memory
Better if there are insertions and / or deletions	Better if search dominates
Guaranteed performance	Performance may vary

Taken from en.wikipedia.org

ALGEBRA

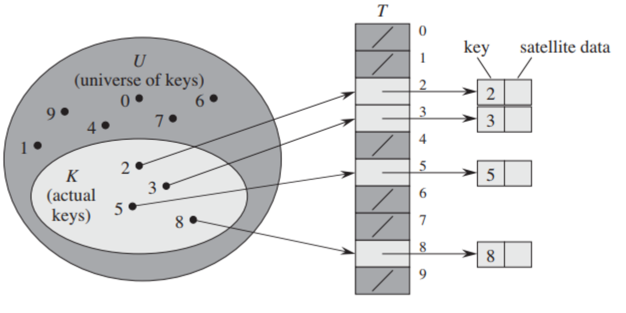
DIRECT ADDRESS TABLES

ALGEBRA

Introduction

- Direct-address tables are a special, simple case of hash tables
  - Great if we have a relatively small number of unique keys and if the difference between the smallest and largest key is not too big
  - Not used in practice
- The basis of direct address tables is the array
  - A position in the array on a certain index is called a slot or a bucket
  - Each possible key has one place reserved in the array
    - The reserved place index is equal to the key

Example



- Variations:
  - Sometimes we store data in the slot instead of the pointer
  - Sometimes we don't keep the key at all because the index itself is actually the key
  - An empty slot needs to be defined correctly

Taken from Cormen et al: Introduction to Algorithms

Operations

- To implement three dictionary operations:
  - SEARCH(key)
    - return array[key]
  - INSERT(key, value)
    - array[key] ← value
  - DELETE(key)
    - array[key] ← NULL
- What is the complexity of each operation?
  - $O(1)$
  - Fantastic, but impractical for real conditions: large, probably underused array is required (e.g. keys 2, 7 and 545,000)

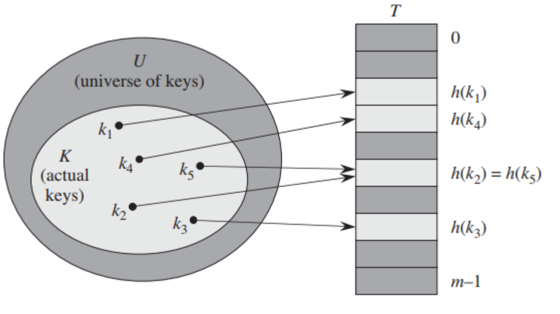
HASH TABLES

Hash tables

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	No

- Hash tables have constant complexity only in the best and average case
  - In the worst case, they give unacceptable linear complexity
- Idea: place the element with the key in the index ***h*(*key*)**
  - We use the hash function *h* to calculate the index from the key
  - This allows us to have an array that is significantly smaller than the total number of keys
- A hash function can convert multiple keys to the same index, which is called a collision

Example of a hash table



- Find the collision

Taken from Cormen et al: Introduction to Algorithms

Hash functions

- A hash function is a function that converts the key to an index of the array
  - On one side of the spectrum are direct address tables
    - ***h*(*key*) = *key***
      - Index is equal to key (each key has a "reserved" place)
    - Best possible performance
    - Large, potentially underutilized arrays are needed
  - The other extreme would be function ***h*(*key*) = *x***
    - Converts all keys to the same index *x* (collisions)
- Good hash functions are between these extremes
  - They evenly convert keys to indexes so that approximately the same number of keys are converted to each index

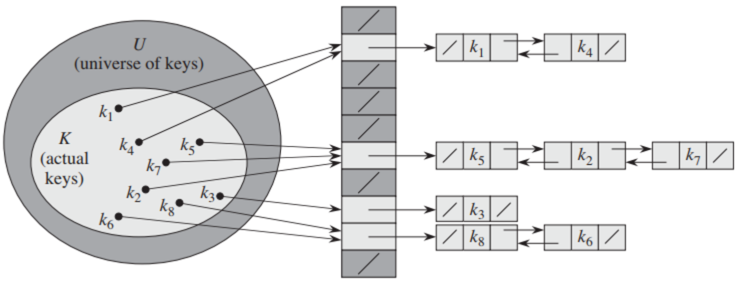
Collision resolution

- Since the hash function can convert multiple keys to the same index, it is necessary to take this into account
- The main ways to resolve collisions:
  - For tables with direct addressing, collision is avoided by function *h*(*key*) = *key*
  - Chaining
  - Open addressing
  - ...

CHAINING

Chaining

- When chaining, we put all the keys that are hashed in the same slot in the linked list that belongs to that slot
  - Each slot contains either a linked list or nullptr



Taken from Cormen et al: Introduction to Algorithms

Operations

- Let's look at how to implement three dictionary operations on a chaining hash table:
  - SEARCH(key)
    - Calculate the hash value based on the key
    - Find the value with the given key from the list in that slot
  - INSERT(key, value)
    - Calculate the hash value based on the key
    - Insert the key and value into the list in that slot
  - DELETE(key)
    - Calculate the hash value based on the key
    - Remove the value with that key from the list in that slot

Example

- Suppose we have a hash table with an array of 7 elements and a hash function: h (key) = key % 7. Let's draw what the hash table looks like after inserting data:

```
1    Danijel Subašić
2    Lovre Kalinić
3    Dominik Livaković
4    Vedran Ćorluka
6    Domagoj Vida
7    Ivan Strinić
8    Šime Vrsaljko
9    Josip Pivarić
10   Tin Jedvaj
11   Dejan Lovren
12   Matej Mitrović
13   Luka Modrić (C)
14   Ivan Rakitić
15   Ivan Perišić
16   Mateo Kovačić
17   Marko Rog
18   Marcelo Brozović
19   Milan Badelj
20   Mario Mandžukić
21   Nikola Kalinić
22   Andrej Kramarić
23   Marko Pjaca
```

Performance (1/2)

- Insertion performance:
  - If we allow double keys: $O(1)$
  - If we no not allow double keys: $O(k)$, where k is the number of elements in the slot
    - We need to inspect all the elements in the slot and check if there already is a key
  - If we know the key is unique: $O(1)$
- Delete performance:
  - In general: $O(k)$
  - If we have an iterator on the element and if the list is double linked: $O(1)$

Performance (2/2)

- Search performance:
  - We define the load factor: $\alpha = \frac{n}{m}$
    - n is the number of elements
    - m is the number of slots
  - Prerequisite 1: as long as the load factor is around or below 1, the search will be on average $\Theta(1)$
    - $n << m$ => α goes to 0 => unused space is growing
    - $n >> m$ => α grows => performance is falling
    - $n \approx m$ => α around 1 => optimum
  - Prerequisite 2: the hash function does a good job
    - Otherwise, the load factor does not play any role

Problem

- Let's take keys from 1 to 100 and place them in our own simple implementation of a hash table with chaining (the value should be the square of the key). We use the hash function *h*(*key*) = *key* % 31. Let's display the distribution of keys by buckets and demonstrate the search.

Solution

```
int main() {
    hash_table ht;
    for (int i = 1; i <= 100; i++) {
         ht.insert(i, i*i);
    }

    ht.print();

    int n;
    cout << "Enter number: ";
    cin >> n;
    cout << "Its square is: " << ht.search(n) << endl;

    return 0;
}
```

Solution

```
struct entry {
    int key;
    int value;
    entry(int key, int value) {
        this->key = key;
        this -> value = value;
    }
};

class hash_table {
private:
    list<entry> ARRAY[31];
    int h(int key);
public:
    void insert(int key, int value);
    int search(int key);
    void print();
};
```


Solution

```
int hash_table::h(int key) {
    return key % 31;
}

void hash_table::insert(int key, int value) {
    int slot = h(key);
    ARRAY[slot].push_back(entry(key, value));
}

void hash_table::print() {
    for (int i = 0; i < 31; i++) {
        cout << "Slot " << i << ": ";
        for (auto it = ARRAY[i].begin(); it !=
                                ARRAY[i].end(); ++it) {
            cout << "key=" << it->key << " ";
        }
        cout << endl;
    }
}
```

Solution

```
int hash_table::search(int key) {
    int slot = h(key);
    for (auto it = ARRAY[slot].begin(); it !=
                                ARRAY[slot].end(); ++it) {
        if (it->key == key) {
            return it->value;
        }
    }
    return -1;
}
```