Data Structure For A Contact List

Terry Yin

November 6, 2014

A phone contact list is a collection of contact. The common collection data types are, (Brookshear 2011, p.342):

- **List** represents a linear collection. Items can be accessed in sequence or by index.
- **Set** represents a collection of unique items. There cannot be duplicated items, and there's no order in the collection.
- **Map** or dictionary/hash, represents a key-value collection, where items can be accessed by a unique key.
- **Tree** is a collection that preserve the relationship amoung the items in a tree structure. A tree makes it easy to access a sub group of items. A tree also builds short paths to access any item in the tree.

These are the very general collection types. And when we talk about data structures, there are at least two different concerns. On concern is the abstract behaviour. The other concern is the underlying implementation details. In this paper, we mostly talk only about the abstract behaviour of the data type. This paper will use the phone contact list as an example to show the thinking process of selecting a proper data struture based on different requirement.

1 Data Type for A Contact

We need to define a basic datatype for the contact list. It's a **user-defined data type** (Brookshear 2011, p.368). We call it `Contact`. The requirement for the contact is:

- Contacts should have some predefined fields, including firt name, last name and phone number.
- It should have self-defined fields per contact

We will ignore the self-defined fileds to keep things simple, because our focus in the paper is on the collection. To define the `Contact` type in Python:

```python
class Contact:
    def __init__(self, first_name, last_name, phone_number):
        self.first_name = first_name
        self.last_name = last_name
        self.phone_number = phone_number

    # we define the __repr__ function to display the content of the contact
    def __repr__(self):
        return "<contact> %s %s: %s" % (self.first_name, self.last_name, self.phone_number)
```

2 Basic User Scenarios: Linear Access

Basically, a phone contact list need to:

- User adds a new contact
- User updates a contact information
- User browses the contacts in alphabetic order

To fulfill the basic user scenarios, a basic list would be enough, because it's simple.

```python
contact_list = list()
contact_list.append(Contact("Terry", "Yin", "119"))
contact_list.append(Contact("Teddy", "Bear", "000"))
contact_list.append(Contact("Jeff", "Dean", "911"))
print(contact_list)
```

```
[<contact> Terry Yin: 119, <contact> Teddy Bear: 000, <contact> Jeff Dean: 911]
```

Because we inserted "Terry" first and then "Jeff", the list is not sorted in alphabetic order. To browse the contacts in alphabetic order, we have two choices:

1. Insert the new contact in the right place when adding contact.
2. Sort the list before browsing.

As browsing is a much more frequent operation than adding, we may choose option 1 to make the browsing faster and easier.

3 More User Scenarios: Access With Keys

A typical phone contact list will also do:

- User searches by name
- User deletes a contact by selecting the name

These behaviours can be implemented with a list as well. But maybe not efficient enough. It looks like a map is an easier solution. A map in Python is called a `dict`, Wikipedia (2014*a*).

```python
contact_dict = dict()
contact_dict["Terry"] = Contact("Terry", "Yin", "119")
contact_dict["Teddy"] = Contact("Teddy", "Bear", "000")
contact_dict["Jeff"] = Contact("Jeff", "Dean", "911")
print(contact_dict)
print(contact_dict["Terry"])
```

```
{'Teddy': <contact> Teddy Bear: 000, 'Jeff': <contact> Jeff Dean: 911, 'Terry': <contact> Terry Yin: 11
<contact> Terry Yin: 119
```

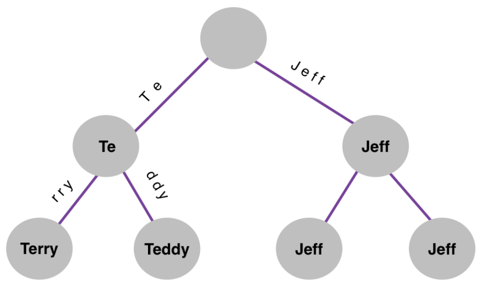
One of the drawback of using map (with current requirement) is that map doesn't keep the order information. So in order to browser the contact list in alphabetic order, we need to sort all the keys of the map first.

4 More User Scenarios: Duplicated Entries

The contact list on the phone I'm using right now (Samsung S4) can also do:

- User adds a duplicated contact
  - Given a contact already exists in the contact list
  - When user adds a contact with the same name
  - Then new contact should be added sucessfully
  - And both contacts should exist in the contact list
- Deleting a contact should not delete other contacts with the same name
- As user is typing the name, the phone should list the contacts with the given prefix already typed dynamically.

Now a map or dict doesn't fit the need any more, because there will be duplicated keys. A map cannot directly have duplicated keys. Considering user also need to look up in the contact list by prefix of name, a good choice could be a **Trie** Wikipedia (2014*b*). A trie is a special type of tree. It forms the tree structure based on the prefix of the keys. It has similar time complexity as a map, but it also provides the subset of all the prefixes, and it's also ordered. A trie is also easier to be modified to allow duplicated keys.



Most programming languages doesn't support trie data structure directly. The trie will be like:

```python
from collections import defaultdict

class Contacts_Trie:

    def __init__(self):
        self.contacts = list()
        self.sub_notes = defaultdict(Contacts_Trie)

    def add(self, name, contact):
        if name == "":
            self.contacts.append(contact)
        else:
            self.sub_notes[name[0]].add(name[1:], contact)

    def contacts_with_prefix(self, prefix):
        if prefix == "":
            return self._get_all_contacts()
        else:
            return self.sub_notes[prefix[0]].contacts_with_prefix(prefix[1:])

    def _get_all_contacts(self):
        result = self.contacts
        for key in sorted(self.sub_notes.keys()):
            result += self.sub_notes[key]._get_all_contacts()
        return result
```

Below is an example and test for using the trie. As you can see, we can get all the contacts who's name have prefix "Te", and duplicated names are allowed.

```python
contacts = Contacts_Trie()
contacts.add("Terry", Contact("Terry", "Yin", "119"))
contacts.add("Teddy", Contact("Teddy", "Bear", "000"))
contacts.add("Terry", Contact("Terry", "Again", "111"))
contacts.add("Jeff", Contact("Jeff", "Dea", "911"))
print contacts.contacts_with_prefix("Te")
```

```
[<contact> Teddy Bear: 000, <contact> Terry Yin: 119, <contact> Terry Again: 111]
```

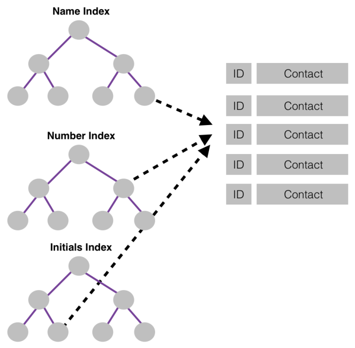
As now duplicated names are allowed, we cannot use name to identity contact entry any more. So a new field `id` need to be introduced to the `Contact` type. `id` is a unique number to identify a contact.

5 Even More User Scenarios: Mulitiple Indexes

The contact list on my phone can also do:

- Incoming call with contact name
- User searches with phone number
- User searches with initals

Now there are more than one way to search in the contact list. Our solution with the trie to store the contacts is not enough. One possible solution is to have **multiple index trees**. Each node in an index tree has a reference to the contact item, which probably is stored in a list.



One of the drawbacks of this solution is adding and deleting an item from the contact list will need to update many index trees. By doing so, we sacrifice the performance for adding and deleting to improve the performance for looking up.

6 Conclusion

This is a great mental exercise for studying data structure. But, after this long discussion, what is the best choice for the data stucture of the contact list? My answer is **I don't know**. And I'm pretty sure **my current design is wrong**.

Why do I know I'm wrong? **Because I've always been wrong**. Experiences tell me, I've never created the right design just by one try for any non-trival software.

However, I have tried my best to speculate about the design of this contact list. I will only know where is wrong in my design when and after I implement my speculative ideas. And when there's requirement change, my design will be wrong again.

References

Brookshear, J. G. (2011), *Computer science: an overview*, Paul Muljadi.

Wikipedia (2014*a*), 'Associative array — wikipedia, the free encyclopedia'. [Online; accessed 6-November-2014].
URL: *http://en.wikipedia.org/w/index.php?title=Associative_array&oldid=630919453*

Wikipedia (2014*b*), 'Trie — wikipedia, the free encyclopedia'. [Online; accessed 6-November-2014].
URL: *http://en.wikipedia.org/w/index.php?title=Trie&oldid=616659272*