

array



```
a = array("f", [0]*100)
a[0] = 1234
a[99] = 5678
```

Random deletion	$O(N)$
Random insertion	$O(N)$
Random access	$O(1)$
Mixed data types	No

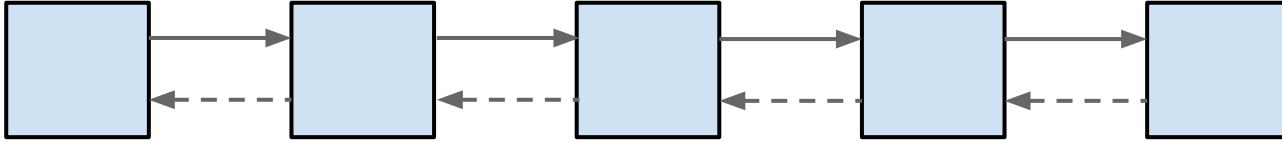
list



```
a = list()  
a.append( 5 )
```

Random insertion / deletion	$O(N)$
Random access	$O(1)$
Mixed data types	Yes

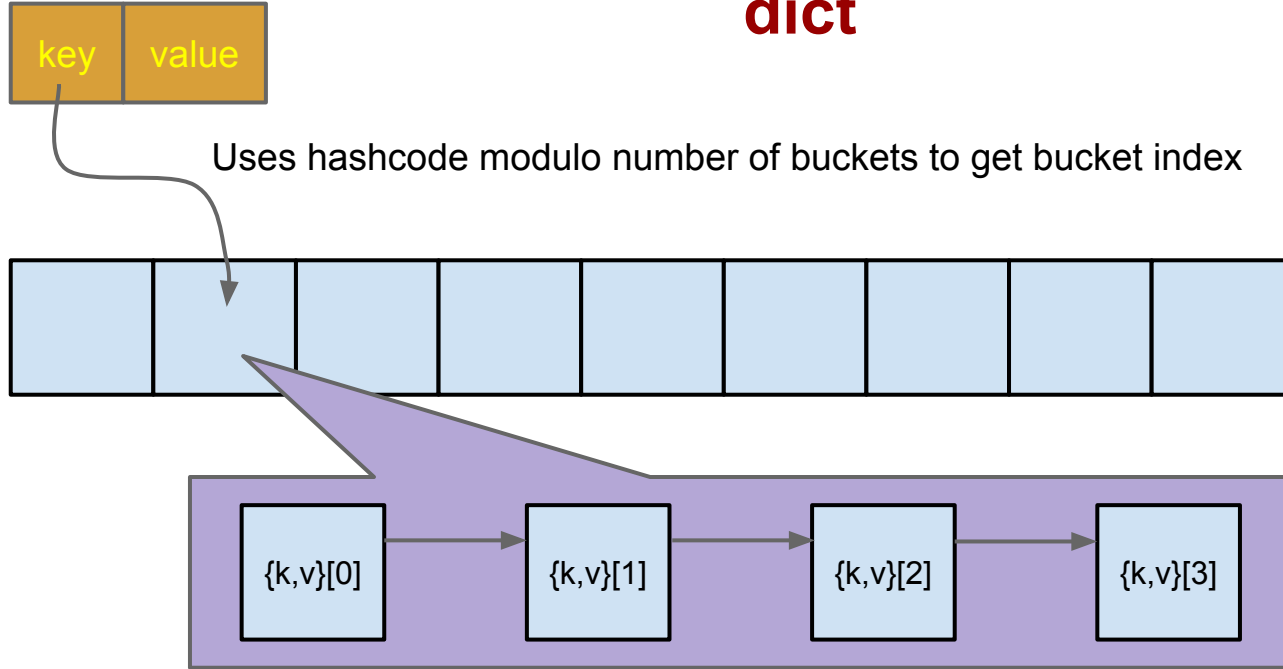
deque



```
from collections import deque
d = deque()
d.append( 5 )
d.popleft()
```

Append / delete from ends	$O(1)$
Random insertion / deletion after traversal	$O(1)$
Random access	$O(N)$
Mixed data types	Yes

dict



Within bucket key/value mapping is unique and stored in a linked list. The search walks this list until *equals* method returns a match.

The idea is to find a hash code that quickly distributes key / value pairs evenly across buckets.

Use of gperf - Perfect hash creator

- From the Linux or Unix command line
 - vi tickers.dat
 - IBM
 - DELL
 - gperf < tickers.dat
- This is what we get:

```
const char *
in_word_set (str, len)
    register const char *str;
    register unsigned int len;
{
    static const char * wordlist[] =
    {
        "", "", "",
        "IBM",
        "DELL"
    };

    if (len <= MAX_WORD_LENGTH && len >= MIN_WORD_LENGTH)
    {
        register int key = hash (str, len);

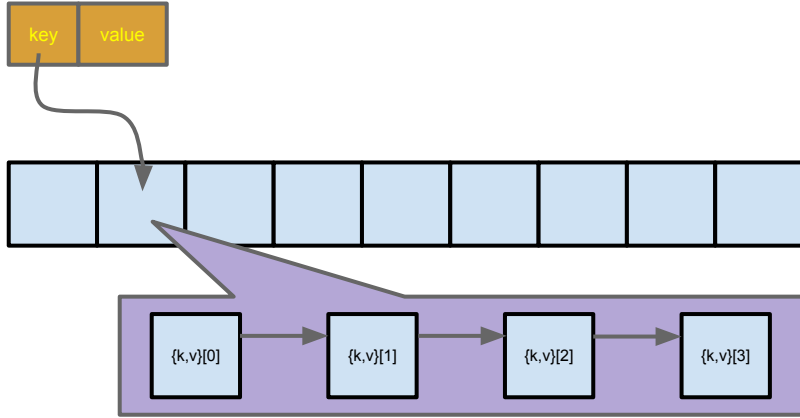
        if (key <= MAX_HASH_VALUE && key >= 0)
        {
            register const char *s = wordlist[key];

            if (*str == *s && !strcmp (str + 1, s + 1))
                return s;
        }
    }

    return 0;
}
lees-air:~ minddrill$ █
```

gperf is smart. It realized that the two tickers we used, IBM and DELL have different lengths, so it made an array of five elements and put IBM in the third place and DELL in the fourth. In other words, the hashing function is simple: Take the length of the ticker and use it as an index into an array.

dict



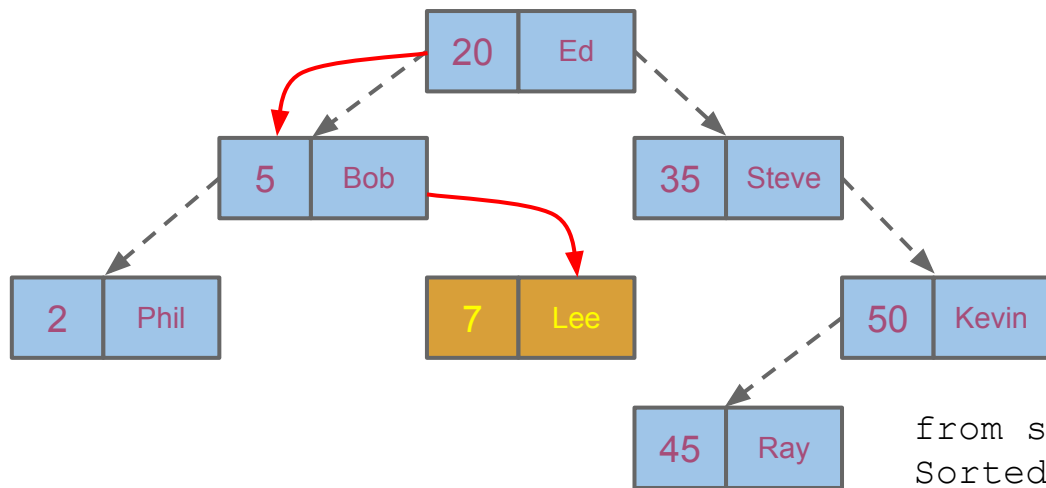
```
d = dict()  
d[ "name" ] = "Lee"  
d[ 5 ] = 32
```

Random access	$O(1)$
Random delete	$O(1)$
Random insert	$O(1)$
Mixed data types	Yes
Hashcode and equals	Required

In some cases, there is a difference between the concepts of computational complexity and computing time. This data structure has low computational complexity but not low computing time. The calculation of the hash and the traversal of the intra-bucket linked lists takes time. Hence, it is computationally simple but time intensive. You wouldn't want to use this data structure to, for example, implement an array, although you could.

SortedDict

The closest we can get to a standard tree map implementation in Python – without rolling our own – is the sorted containers class, SortedDict. A binary tree is always in order. Balanced binary trees allow rapid searching and $O(\log(N))$ insertion and growth.



Random access	$O(\log(N))$
Node disconnect	$O(1)$ for binary tree but...
Random insert	$O(\log(N))$
Mixed data types	Yes
Tail insertion, deletion	Can be modified for $O(1)$
Comparator	Required

```
from sortedcontainers import  
SortedDict  
d = SortedDict()  
d[ "name" ] = "Lee"  
d[ 5 ] = 32
```






set and tree set

- A set is a dict but without values. It contains only keys. The complexity is the same.
- There is no standard tree set implementation but a tree set is, to a tree exactly what a set is to a dictionary: keys but not values

CircularList - No standard implementation

	F						L	
addLast	F							L
remFirst, addLast	L	F						
remFirst, addLast		L	F					
remFirst, remLast	L			F				

CircularList

Deletion from ends	$O(1)$	
Growth at ends	$O(1)$	
Random access	$O(1)$	
Random deletion / insertion	$O(N)$	
Mixed data types	Yes	

As we are often in the position of using these features of a class to manage queues of data, the circular list should be of particular interest to us.