Two main data structures

- Stack

Stack is a typical data structure which has a feature of **first in last out**
It has 4 main functions

1. push(self,x)-- add element x to the top of the stack
2. pop(self) -- remove and return the top element from the stack, an error would occur if the stack is empty.
3. top(self)--return a reference to the top element of the stak, without removing it, an error occurs of the stack is empty.
4. **bool**(self)-- return **True(a boolean value)** if the stack contains something.

- The python use

```
class Astack:
    def __init__(self):
       self.a = []
    def __bool__(self):
       return self.a != []
    def push(self, x):
       self.a.append(x)
    def pop(self):
       x = self.top()
       del self.a[-1]
       return x
```

An example of application -- check the matching

```
def paren_match(s):
    stack = Astack()
    for c in s:
        if c in ('(', '[', '{'):
            stack.push(c)
        elif c in (')', '}', ']'):
            if not stack or stack.pop() and c not in ('()', '{}', '[]'):
                return False
    return not stack #This statement means after comparation, the stack is empty
and it is true.
```

- Queues

Feature: First in first out.
Usually, a Queue has 4 main functions similar to stack.

1. `push_back(self,x)` ---Add element x to the end fo the queue

2. `pop(self)` --- remove and return the first element from the queue, and error occurs if the queue is empty.
3. `top(self)` --- return a reference to the first element of the queue, without remiving it, an error occurs if the queue is empty.
4. `__bool__(self)`--- return True if the queue contains some elements, Flase if empty

We can build a queue by using link-list

```
class Node:
    def __init__(self):
        self.elm = None
        self.nxt = None
class Lqueue:
    def __init__(self):
        self.head = self.dummy = Node(None,None)
    def __bool__(self):
        return self.head is not self.dummy
    def push_back(self,x):
        self.dummy.elm, self.dummy.nxt = x, Node(None,None)
        self.dummy = self.dummy.nxt
        #Interesting thing is that we can sue 2 ways to build a queue, one is build the connection at first, amnother is use psuh_back()
    def top(self):
        if not self:
            raise IndexError
        return self.head.elm
    def pop(self):
        x = self.top()
        self.head = self.head.nxt
        return x
    def __iter__(self):
        p = self.head
        while p is not self.dummy:
            yield p.elm
            p = p.nxt
```

Also we can use **Circular Array** to finish this task.

```
class Aqueue:
    def __init__(self):
        self.a = [None]*16
        self.n = len(self.a)
        self.head = self.tail = 0
    def __bool__(self):
        return self.head != self.tail
    def __len__(self):
        return (self.tail - self.head)%self.n#Give the defination that in a fixed array, there can be a changable queue which has a head and tail.
    def __iter__(self):
        yield from (self.a[(self.head + i)%self.n] for i in range(len(self)))
    def top(self):
        if not self:
            raise IndexError
        return self.a[self.head]
    def pop(self):
        x = self.top()
        self.a[self.head] = None
        self.head = (self.head + 1)%self.n
        return x
    def push_back(self,x):
        l = len(self)
        if l + 1 == self.n:
            self.a[self.tail:self.tail] = [None]*self.n
            self.n = len(self.a)
            self.head = (self.tail - l)%self.n
        self.a[self.tail] = x
        self.tail = (self.tail+1)%self.n
```

Circular Doubly Linked Lists and Deques

A list has a node and a pointer, the doubly linked list has 2 pointers, one for the next node, one for the previous node.

- insert a node in dll

```
def insert(p,q):
    p.nxt, p.pre = q, q.pre
    p.pre.nxt = p.nxt.pre = p
```

The advantages of DLL:

1. Nodes ar both ends are immediately accessible
2. Insertions and deletions at both ends are very efficient, independent to the length of the list
3. To add an element at teh first position, we insert it before `dummy.nxt`
4. To add an element ate the last position, we insert it before `dummy`
5. To remove an elment at teh first position, we delete `dummy.nxt`
6. To remove an element at the last position, we delete `dummy.pre`

Before the building of DLL, we put the linked list first.

```
class listNode():
    def __init__(self,x):
        self.val = x#The value of node
        self.nxt = None#The reference of next node

n0 = listNode(1)#Build the nodes
n1 = listNode(2)
```

```
n2 = listNode(3)
n0.nxt = n1#Build the references
n1.nxt = n2
n2.nxt = None
```

Then is the DLL buiding.

```
class DLL():
    def __init__(self,x):
        self.val = x
        self.nxt = self.pre = self
n0 = DLL(1)
n1 = DLL(2)
n2 = DLL(3)
n3 = DLL(4)
n0.nxt = n1
n1.nxt = n2
n2.nxt = n3
n3.nxt = n0
n0.pre = n3
n1.pre = n0
n2.pre = n1
n3.pre = n2
#Following are some common operations of Doulbe Linked List
def insert_elm(x,q):
    p = DLL(x)
    insert(p,q)
def delete(p):
    p.pre.nxt = p.nxt
    p.nxt.pre = p.pre
    p.nxt = p.pre = p
    return DLL(p)
def __bool__(self):
    return self.dummy.nxt is not self.dummy
def check_empty(self):
    if not self:
        raise IndexError
def __iter__(self):#Overview the link list
    p = self.dummy.nxt
    while p is not self.dummy:
        yield p.elm
        p = p.nxt
def __reversed__(self):#Get the reversed list
    p = self.dummy.pre
    while p is not self.dummy:
        yield p.elm
        p = p.pre
def push(self,x):
    inser_elm(x, self.dummy.nxt)
def pop(self):
    self.check_empty()
    x = delete(self.dummy.nxt)
```

```
        return x
    def top(self):
        self.check_empty()
        return self.dummy.nxt.elm
    def push_back(self,x):
        insert_elm(x,self.dummy)
    def pop_back(self):
        self.check_empty()
        x= delete_elm(self.dummy.pre)
        return x
    def back(self):
        self.check_empty()
        return self.dummy.pre.elm
```

Recursion

Recursion is the general term for the practice of defining a nobject in terms of itself or of part of itself

- A recursive or inductive definition of a function consists of 2 steps

1. Basis Step: Specify the value of the function at initial values
2. Recursive Step: Give a rule for finding its value at an integer fro mits values at smaller integers

Recurrence Relations

A recurrence relation for the sequence $\{a_n\}$ is an equation that expresses the term a_n in terms of some previous terms, namely a_0, a_1 ..., of the sequence for some positive integer n.

A famouse example: Fibonacci sequence.

Fibonnacci sequence f_0,f_1, f_2 is a recurrence relation with the initial terms f_0 = 0 and f_1 = 1with the following recurrence equation : $f_n = f_{n-1} + f_{n-2}$,when n belongs to {2,3,4......} In this example, the most inportant thing is find 2 variables to store the values of f_{n-1} and f_{n-2} and apply them in code. But we can see the most direct solution first:

```
def fit(n):
    if(n < 2):
        return n
    else:
        return fit(n - 1) + fit(n - 2)
```

In this function, the fit() will be executed many times which is not effcient. So we often use the following steps:

```
int fib(int n){
    return additiveSequence(n, 0, 1);
}
```

```
int additiveSequence(int n, int t0, int t1){
    if (n==0) return t0;
    if (n==1) return t1;
    return additiveSequence(n-1, t1, t0+t1);
}
/*use c language*/
```

Josephus Problem

During the JEwish-Roman war, Flavius Josephus, a famous historian of the first century, was among a band of 41 Jewish rebels trapped in a cave by the Romans. Preferring suivide to capture, the rebels decided to form a circle and, proceeding around it, to kill every third remaining person until no one was left. But Josephus, together his friend, wanted to avoid being killed. So he quickly calculated where he and his friend should stand in the vicious circle.

```
public class JosephusProblem {
    public static int flag = 0;
    public static void main (String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入一个数字：");
        int number = input.nextInt();
        josephus_1(number);
        System.out.println("幸运者为1：" + flag);
    }
     private static void josephus_1(int numbers) {
        //如果只有一个人的情况下，那么这一个人存活J(K) = 1
        //如果有偶数的情况下，那么情况为J(2 * K) = 2 * J(K) - 1
        //如果有奇数的情况下，那么情况为J(2 * K + 1) = 2 * J(K) + 1
        if (numbers == 1) {
            flag = 1;
        } else if (numbers % 2 == 0) {
            josephus_1(numbers / 2);
            flag = flag * 2 - 1;
        } else if (numbers % 2 != 0) {
            josephus_1((numbers - 1) / 2);
            flag = flag * 2 + 1;
        }
```

Tree

- A tree is either empty or a root node r which contains an element,adn zero or more non-empty subtrees, and there is an edge, which is directional,going from node r to the root node of each subtree.

    1. The root of each subtree is called a child of r, and r is the parent of each child.The number of children that a node is called its degree.
    2. Nodes with the same parent are siblings
    3. A node with no children(0-degree) is called a leaf, or an external node;otherwise it is called an internal node

- Pre-order traversal: The recursion solution

```
def preorderTraversal(root):
    def preorder(self,root, res):
        if root == None:
            return
        res.append(root.val)
        self.preorder(root.left, res)
        self.preorder(root.right, res)
    res = []
    preorder(root, res)
    return res
```

C++

```
class Solutioin{
    public:
    void inorder(TreeNode* root, vector<int>& res){
        if(! root){
            return;
        }
        res.push_back(root->val);
        inorder(root->left, res);
        inorder(root->right, res);
    }
    vector<int> ubirderTraversal(TreeNode* root){
        vector<int> res;
        inorder(root, res);
        return res;
    }
}
```

java

```
class solution{
    public List<Integer> inorderTraversal(TreeNode root){
        List<Integer> res = new ArrayList<lnteger>();
        inorder(root, res);
        return res;
    }
    public void inorder(TreeNode root, List<Integer> res){
        if(root == null){
            return;
        }
        res.add(root.val);
        inorder(root.left, res);
        inorder(root.right,res);
    }
}
```

C

```
void inorder(struct TreeNode* root, int* res, int* resSize){
    if(!root)
}
```

- Binary Search Tree Binary Search Tree is a special tree which is ordered by the values of numbers.
    - Order rules: From the first element, we add it as the root node. Add next value at the left side of it is less than the root value, and right side if it is larger than that. After the building ,the leftest element must be the smallest element and rightest is the largest element.
    - AVL tree AVL tree is one type of the Binary Search Tree. The most significant feature is that the

HashMap

HashMap is a special data structure with amazing efficiency. If we know the "Key", the searching effiency will be $O(1)$.

python implement

```
""" 初始化哈希表 """
mapp = {}# In python, dictinary is based on hashmap.

""" 添加操作 """
# 在哈希表中添加键值对 (key, value)
mapp[12836] = "小哈"
mapp[15937] = "小啰"
mapp[16750] = "小算"
mapp[13276] = "小法"
mapp[10583] = "小鸭"

""" 查询操作 """
# 向哈希表输入键 key ，得到值 value
name = mapp[15937]

""" 删除操作 """
# 在哈希表中删除键值对 (key, value)
mapp.pop(10583)
```

java implement

```
/* 初始化哈希表 */
Map<Integer, String> map = new HashMap<>();
```

```
/* 添加操作 */
// 在哈希表中添加键值对 (key, value)
map.put(12836, "小哈");
map.put(15937, "小啰");
map.put(16750, "小算");
map.put(13276, "小法");
map.put(10583, "小鸭");

/* 查询操作 */
// 向哈希表输入键 key ，得到值 value
String name = map.get(15937);

/* 删除操作 */
// 在哈希表中删除键值对 (key, value)
map.remove(10583);
```

C++ implement

```
/* 初始化哈希表 */
unordered_map<int, string> map;

/* 添加操作 */
// 在哈希表中添加键值对 (key, value)
map[12836] = "小哈";
map[15937] = "小啰";
map[16750] = "小算";
map[13276] = "小法";
map[10583] = "小鸭";

/* 查询操作 */
// 向哈希表输入键 key ，得到值 value
string name = map[15937];

/* 删除操作 */
// 在哈希表中删除键值对 (key, value)
map.erase(10583);
```

There are 3 ways to overview the hashmap:**traversal the Key-Point, traversal Keys, traversal points**

python implement

```
""" 遍历哈希表 """
# 遍历键值对 key->value
for key, value in mapp.items():
    print(key, "->", value)
# 单独遍历键 key
for key in mapp.keys():
    print(key)
```

```
# 单独遍历值 value
for value in mapp.values():
    print(value)
```

java implement

```
/* 遍历哈希表 */
// 遍历键值对 key->value
for (Map.Entry <Integer, String> kv: map.entrySet()) {
    System.out.println(kv.getKey() + " -> " + kv.getValue());
}
// 单独遍历键 key
for (int key: map.keySet()) {
    System.out.println(key);
}
// 单独遍历值 value
for (String val: map.values()) {
    System.out.println(val);
}
```

C++ implement

```
/* 遍历哈希表 */
// 遍历键值对 key->value
for (auto kv: map) {
    cout << kv.first << " -> " << kv.second << endl;
}
// 单独遍历键 key
for (auto key: map) {
    cout << key.first << endl;
}
// 单独遍历值 value
for (auto val: map) {
    cout << val.second << endl;
}
```

Hash function

The basic data structure of HashMap is called **Bucket**. Arrays, linklist or Binary search tree can make buckets. In hashmap, the most important thing is **mapping**. So if we want to get some specified value in map, we should do the following steps:

1. Calculate the index throught hash function
2. Get the relative value through index.

A simple hash funcion

```java
/* 键值对 int->String */
class Entry {
    public int key;
    public String val;
    public Entry(int key, String val) {
        this.key = key;
        this.val = val;
    }
}

/* 基于数组简易实现的哈希表 */
class ArrayHashMap {
    private List<Entry> bucket;
    public ArrayHashMap() {
        // 初始化一个长度为 100 的桶（数组）
        bucket = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            bucket.add(null);
        }
    }

    /* 哈希函数 */
    private int hashFunc(int key) {
        int index = key % 100;
        return index;
    }

    /* 查询操作 */
    public String get(int key) {
        int index = hashFunc(key);
        Entry pair = bucket.get(index);
        if (pair == null) return null;
        return pair.val;
    }

    /* 添加操作 */
    public void put(int key, String val) {
        Entry pair = new Entry(key, val);
        int index = hashFunc(key);
        bucket.set(index, pair);
    }

    /* 删除操作 */
    public void remove(int key) {
        int index = hashFunc(key);
        // 置为 null ，代表删除
        bucket.set(index, null);
    }

    /* 获取所有键值对 */
    public List<Entry> entrySet() {
        List<Entry> entrySet = new ArrayList<>();
        for (Entry pair : bucket) {
            if (pair != null)
                entrySet.add(pair);
        }
        return entrySet;
    }

    /* 获取所有键 */
    public List<Integer> keySet() {
        List<Integer> keySet = new ArrayList<>();
        for (Entry pair : bucket) {
            if (pair != null)
                keySet.add(pair.key);
        }
        return keySet;
    }

    /* 获取所有值 */
    public List<String> valueSet() {
        List<String> valueSet = new ArrayList<>();
        for (Entry pair : bucket) {
            if (pair != null)
                valueSet.add(pair.val);
        }
        return valueSet;
    }

    /* 打印哈希表 */
    public void print() {
        for (Entry kv: entrySet()) {
            System.out.println(kv.key + " -> " + kv.val);
        }
    }
}
```

Hash conflict

If the hash funtion culculate 2 same indexes, we can it hash conflict