4.栈和队列

1队列

列队包含一个数组和两个变量，头与尾
允许在队列的head进行删除操作，tail进行插入操作
原则是 First in ,first out .
是学习广度优先搜索以及列队优化的Bellman-Ford最短路算法的核心数据结构。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int q[30]={6,3,1,7,5,8,9,2,4},head,tail;
    printf("Could you tell me your QQ secretly ?\n");
    head=0;
    tail=9; //tail指向队尾的后一个位置，方便
    while(head<tail) //判断非空
    {
        printf("%d",q[head]);
        head++;
        q[tail]=q[head];
        tail++;
        head++;
    }
    return 0;
}
```

为了更加直观，我们可以把队列封装为结构体类型实现上述功能

```
#include <stdio.h>
#include <stdlib.h>
struct queue
{
    int date[100];
    int head ;
    int tail ;
};
int main ()
{
    struct queue q ;
    int i ;//初始化列队
    q.head=1;
    q.tail=1;
    for(i=1;i<=9;i++)
    {
        scanf("%d",&q.date[q.tail]);
        q.tail++;
    }
    while(q.head<q.tail)
    {
        printf("%d ",q.date[q.head]);
        q.head++;
        q.date[q.tail]=q.date[q.head];
        q.tail++;
        q.head++;
    }
    return 0 ;
}
```

2.栈

特点 后进后出
例1 回文字符

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char a[101],s[101];
    int i ,len , next ,top ,mid;
    gets(a);
    len=strlen(a);
    mid=len/2-1;//求字符串的中点
    top=0;//栈的初始化
    for(i=0;i<=mid;i++)
    {
        top++;
        s[top]=a[i];
    }//mid前入栈
    if(len%2==0)
        next=mid+1;
    else
        next = mid+2;
    //next为需要匹配的字符下标
    for(i=next;i<=len-1;i++)
    {
        if(a[i]!=s[top])
            break;
        top--;
    }
    if(top==0)
        printf("YES");
    else
        printf("NO");
    return 0 ;
}
```

remark:
最特别要注意的是top++的顺序什么的因为最后末尾了还要--才等于0

5.3树

资料来源

1.引子——动态规划

（1）Fibonacci

1.递归实现

```
int F(int i )
  {
    if(i<1) return 0;
    if(i==1) return 1;
  return F(i-1)+F(i-2);
  }
```

程序虽然紧凑精致，但是指数级的运行时间却让它不可用。

2.自底向上的动态规划.

```
F(0)=0;F(1)=1;
F(U)=F(i-1)+F(1-2);
```

我们使用可以按从最小开始的顺序计算所有函数值来求任何类似函数的值，在每一步使用先前已经计算的值来计算出当前值。它的运算是线性级别的

3.自顶向下的动态规划

```
int F(int i )
  {
    int t ;
  if(knownF[i]!=-1) return knownF[i];
  if(i==0) return 0;
  if(i==1) return 1;
  if(i>1) t=F(i-1)+F(i-2);
  return knownF[i]=t;
  }
```

我们一般使用自定向下的动态规划：1更自然2计算子序列问题能自己解决3可能不需要计算全部子问题的解。

2.树

（1）二叉树

A binary tree is a structure comprising nodes, where each node has the following 3 components:

1. Data element: Stores any kind of data in the node
2. Left pointer: Points to the tree on the left side of node
3. Right pointer: Points to the tree on the right side of the node

Commonly-used terminologies

- **Root**: Top node in a tree
- **Child**: Nodes that are next to each other and connected downwards
- **Parent**: Converse notion of child
- **Siblings**: Nodes with the same parent
- **Descendant**: Node reachable by repeated proceeding from parent to child
- **Ancestor**: Node reachable by repeated proceeding from child to parent.
- **Leaf**: Node with no children
- **Internal node**: Node with at least one child
- **External node**: Node with no children

Structure code of a tree node

In programming, trees are declared as follows:

```
    struct node
    {
         int data;                 //Data element
         struct node * left;          //Pointer to left
node
         struct node * right;         //Pointer to right
node
    };
```

Creating nodes

Simple node

```
    struct node root;
```

Pointer to a node

```
    struct node * root;
    root=(node * )malloc(sizeof(node));
```

In this case, you must explicitly allocate the memory of the node type to the pointer (preferred method).

Utility function returning node

```
struct node * newnode(int element)
{
    struct node * temp=(node * )malloc(sizeof(node));
    temp->data=element;
    temp->left=temp->right=NULL;
    return temp;
}
```

Maximum depth/height of a tree

The idea is to do a post-order traversal and maintain two variables to store the left depth and right depth and return max of both the depths.

```
int maxDepth(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
         /* compute the depth of each subtree */
          int lDepth = maxDepth(node->left);
          int rDepth = maxDepth(node->right);

          /* use the larger one */
          if (lDepth > rDepth)
                return(lDepth+1);
          else
               return(rDepth+1);
   }
}
```

Time complexity

O(n)O(n)

Application of trees

1. a Manipulate hierarchical data
2. Make information easy to search (see tree traversal)
3. Manipulate sorted lists of data
4. Use as a workflow for compositing digital images for visual effects
5. Use in router algorithms

Contributed by: Vaibhav Tulsyan

例子：反转二叉树，背包问题