数据结构

数据结构:
data structure
是带有结构的数据元素的集合。
结构指的是数据之间的相互关系，即数据的组织形式，结构中的数据元素称为节点。

- 相关概念
  - 数据（data）
    - 描述客观事物的数/字符以及能输入计算机中并被计算机处理的符号的集合。
  - 数据元素（data element）
    - 数据的基本单位
  - 数据对象（data object）
    - 具有相同性质的数据元素的集合，是数据的一个子集
  - 数据结构的包含的三个方面
    - 逻辑结构

      数据元素之间的逻辑或抽象关系，从逻辑关系上描述数据，与数据的存储结构无关，独立于计算机。可以看作是具体问题抽象处理的数学模型。

      - 线性结构

        数据元素之间存在着一对一的关系，且结构中仅有一个开始节点和一个终端节点，其余节点都是仅有一个直接前趋和一个直接后继。
        例如：链表

      - 非线性结构

        数据元素之间存在着一对多或者多对多的关系。
        例如树结构，网结构，图结构

    - 物理结构

      数据元素及其关系在计算机内从存储方式。

      - 存储结构
        - 顺序存储结构

          按逻辑关系顺序存储

        - 链式存储结构

          在内存中按指针链接存储

      - 存储方法
        - 顺序存储方法

          把逻辑上相邻的节点存储到物理结构上也相邻的连续存储单元里，由此得到的存储结构称为顺序存储结构。
          通常借助程序设计语言的数组来描述。

- 链接存储方法

  用一组不一定连续的存储单元存储逻辑上相邻的元素，元素见的逻辑关系是由附加的指针域来表示，由此得到的存储结构称为链式存储结构。

- 索引存储方法

  索引存储方法通常是在存储元素信息的同时，还建立附加的索引表。
  表中的索引项一般形式是：关键字，地址
  关键字是唯一标志一个元素的一个数据项或多个数据项的组合。

- 散列存储方法

  散列存储方法的基本思想是根据元素的关键字直接计算出该元素的存储地址。

- 数据的运算

  即对数据元素施加的操作。

表

线性表

定义

- 1.有且只有一个开头元素，没有前趋，只有一个直接后继
- 2.有且只有一个终端元素，没有后继，仅有一个直接前趋
- 3.其余元素称为内部元素，仅有一个直接前趋和一个直接后继

运算

常见的基本运算

- 1.构造空表
- 2.求表长
- 3.根据索引查找
- 4.根据值查找
- 5.插入
- 6.删除

存储

顺序存储

线性表的顺序存储指的是将线性表的数据元素按其逻辑次序依次存入一组地址连续的存储单元里，用这种方法存储的线性表称为顺序表。

元素之间的关系

假设线性表中所有元素的类型都是相同的，且每个元素占用d个存储单元，其中第一个单元的存储地址就是该元素的存储位置

- a(i+1)跟a(i)关系

$$LOC(a_{i+1}) = LOC(a_i) + d$$

- a^i的存储位置

$$LOC(a_i) = LOC(a_1) + (i-1) * d$$

- 运算
  - 插入元素
    - 后面的元素需要后移
      - 假设pi是在第i个元素之前插入一个元素的概率，则在长度为n的线性表中插入一个节点是所需要移动元素的期望值（平均数）

$$E_{is}(n) = \sum_{i=1}^{n+1} p_i(n-i+1)$$

      - 在等概率下插入，需要移动元素的平均次数

$$E_{is}(n) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2}$$

  - 删除
    - 后面的元素需要前移
      - qi为删除发第i个元素的概率，删除一个元素的平均移动次数

$$E_{de}(n) = \sum_{i=1}^{n} q_i(n-i) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

- 链式存储

  链式存储结构存储线性表数据元素的存储空间可能是连续的，也可能是不连续的，因而链式表的节点是不可以存取存取的。

  - 单向链表
    - 域
      - 数据域

        存储数据元素的域称为数据域

      - 指针域

        存储直接后继存储地址的域称为指针域

    - 运算
      - 创建
        - 头插法
          - 先生成终点节点，后续的节点插入链表头
        - 尾插法

- 固定头，新节点接上个的尾部
- 查找
  - 按结点序号查
  - 按结点值找
- 插入
- 删除

- 循环链表

  循环链表是链式存储结构的另一种形式。
  其特点是单链表中最后一个结点的指针域不为空，而是指向链表头节点，使整个链表构成一个环。

- 双向链表
  - pre
    - data
      - next

- 栈(Stack)
  - 定义
    - 栈是限定在表的一端进行插入和删除运算的线性表，通常将插入、删除的一端称为栈顶，另一端称为栈第。不含元素的空表称为空栈。
      - 像桶、LIFO(Last In First Out) 后进先出
  - 运算
    - 构造空栈
      - InitStack
    - 判断栈是否为空
      - StackEmpty
    - 判断栈是否已满（满栈）
      - StackFull
    - 进栈（入栈-插入）
      - Push
    - 退栈（出栈-删除）
      - Pop
    - 取栈顶元素
      - GetTop
  - 存储结构
    - 顺序存储

- 顺序栈
  - 数组实现
    - 栈底-数组最低端，下标为0
    - 栈顶-常用整形变量top来值时栈顶位置-通常称top为栈顶指针
  - 运算

```
顺序栈定义：
#define StackSize 100 //栈空间
typedef char DataType;
typedef struct{
      DataType data[StackSize]; //存放结点
      int top;   //栈顶指针
}SeqStack;
SeqStack S;
```

    - 1.置空栈

```
void InitStack(SeqStack * S)
{
      S->top=-1;    //空栈栈顶指针不能是0
}
```

    - 2.判栈空

```
int StackEmpty(SeqStack * S)
{
      return S->==-1;
}
```

    - 3.判栈满

```
int StackFull(SeqStack * S)
{
      return S->top == StackSize - 1;
}
```

    - 4.进栈（入栈）

```
void Push(SeqStack * S,DataType x)
{
      if(StackFull(S))
            printf("stack overflow");
      else{
            S->top = S-> + 1;
            S->data[S->top]=x;
      }
}
```

- 5.退栈（弹栈）

```
DataType Pop(SeqStack * S)
{
    if(StackEmpty(S)){
        printf("stack underflow");
        exit(0);
    }else{
        return S->data[S->top--];
    }
}
```

- 6.取栈顶（不改变指针）

```
DataType GetTop(SeqStack * S)
{
    if(SeqStackEmpty(S)){
        printf("stack empty");
    }else{
        return S->data[S->top];
    }
}
```

- 链式存储

  可以解决由顺序存储分配固定空间所产生的一处和空间浪费问题

  - 链栈
    - 链表实现
      - 插入跟删除操作仅限在表头（栈顶）进行
    - 运算

```
链栈定义：
#define StackSize 100 //栈空间
typedef char DataType;
typedef struct stacknode{
    DataType data; //存放结点
    struct stacknode * next;     //栈顶指针
}StackNode;
typedef StackNode * LinkStack;
LinkStack top;
```

- 1.判栈空

```
int StackEmpty(LinkStack * top)
{
    return top == NULL;
}
```

- 2.进栈（入栈）

```
void Push(LinkStack top,DataType x)
{
    StackNode * p;
    p=(StackNode *)malloc(sizeof(StackNode));
    p->data=x;
    p->next=top;
    top = p;
    return top;
}
```

- 3.退栈（弹栈）

```
void Pop(LinkStack top,DataType x)
{
    StackNode * p = top;
    if(StackEmpty(top))
        printf("stack empty");
    else{
        *x = p->data;
        top=p->next;
        free(p);   //删除p指向的结点
        return top;
    }
}
```

- 4.取栈顶

```
DataType GetTop(LinkStack top)
{
    if(SeqStackEmpty(top)){
        printf("stack empty");
    }else{
        return top->data;
    }
}
```

- 队列（Queue）
  - 定义
    - 队列也是一种操作受限的线性表，它只允许在表的一端进行元素插入，而在另一端进行元素删除。允许插入的一端称为队尾（rear），允许删除的一端称为队头（front）
      - 像排队、先进先出（First In First Out，FIFO）
  - 运算
    - 1.置空队列（构造一个队列）
    - 2.判队空QueueEmpty
    - 3.入队列EnQueue
    - 4.出队列DeQueue
    - 5.取队头GetFront

- 存储结构
  - 顺序存储
    - 顺序队列

      由于队列的队头和队尾的位置是变化的，因此需要设置两个指针front和rear分别指示队头和队尾元素在表中的位置，初始值为0.

      ```
      #define QueueSize 1000
      typedef struct{
            DataType data[QueueSize];
            int front,rear;
      } SeqQueue;
      SeqQueue Q;
      ```

      - 数组实现
        - 入队时，将新元素插入rear所指的位置，再将rear+1
        - 出队时，将front+1，并返回被删除的元素
      - 循环队列

        ```
        #define QueueSize 100
        typedef struct{
              DataType data[QueueSize];
              int front,rear;
        } CirQueue;
        ```

        - 定义
          - 为了解决数组的溢出和浪费问题，可以把数据看出一个环，这样定义的循环队列，数组能被有效利用
        - 运算
          - 1.置空队列（构造一个队列）

            ```
            void InitQueue(CirQueue * Q)
            {
                  Q->front=Q->rear=0;
            }
            ```

          - 2.判队空QueueEmpty

            ```
            int QueueEmpty(CirQueue * Q)
            {
                  return Q->rear==Q->front;
            }
            ```

            - 队满：尾指针追上头指针
            - 对空：头指针追上尾指针
          - 3.判队满

            ```
            int QueueFull(CirQueue * Q)
            {
                  return (Q->rear + 1) % QueueSize == Q->front;
            }
            ```

- 4.入队列EnQueue

```
void EnQueue(CirQueue * Q)
{
    //插入队尾
    if(QueueFull(Q))
        printf("Queue overflow");
    else{
        Q->data[Q->rear]=x;
        Q->rear=(Q->rear+1) % QueueSize;
    }
}
```

- 5.出队列DeQueue

```
DataType DeQueue(CirQueue * Q)
{
    //取队头的值,删除队头元素
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        x = Q->data[Q->front];
        Q->front=(Q->front+1) % QueueSize;
        return x;
    }
}
```

- 6.取队头GetFront

```
DataType GetFront(CirQueue * Q)
{
    //取队头的值
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        return Q->data[Q->front];
    }
}
```

- 链式存储
  - 链队列

```
限制在表头删除、表尾插入操作的单链表
#define QueueSize 1000
typedef struct qnode{
    DataType data[QueueSize];
    int qnode * next;
} QueueNode;
typedef struct{
    QueueNode * front;
    QueueNode * rear;
} LinkQueue;
LinkQueue Q;
```

    - 链表实现
      - 一个队列由一个头指针和一个尾指针唯一确定

- 运算
  - 1.置空队列（构造一个队列）

```
void InitQueue(LinkQueue * Q)
{
    //头节点
    Q->front=(QueueNode *)malloc(sizeof(QueueNode));
    //尾指针也指向头节点
    Q->rear=Q->front;
    Q->rear->next=NULL;
}
```

  - 2.判队空QueueEmpty

```
int QueueEmpty(LinkQueue Q)
{
    //头尾指针相等队列为空
    return Q->rear=Q->front;
}
```

  - 3.入队列EnQueue

```
void EnQueue(LinkQueue*Q,DataType x)
{
    //将x插入队尾
    QueueNode * p = (QueueNode *)malloc(sizeof(QueueNode));
    p->data = x;
    p->next = NULL;
    Q->rear->next = p;
    Q->rear=p;
}
```

  - 4.出队列DeQueue

```
DataType DeQueue(LinkQueue * Q)
{
    //取队头的值,删除队头元素
    QueueNode * p;
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        p = Q->front;
        Q->front=Q->front->next;
        free(p);
        return (Q->front->data);
    }
}
```

- 5.取队头GetFront

```
DataType GetFront(LinkQueue * Q)
{
    //取队头的值
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        return Q->front->next->data;
    }
}
```

- 树
- 散列
- 图