第十四章 单链表

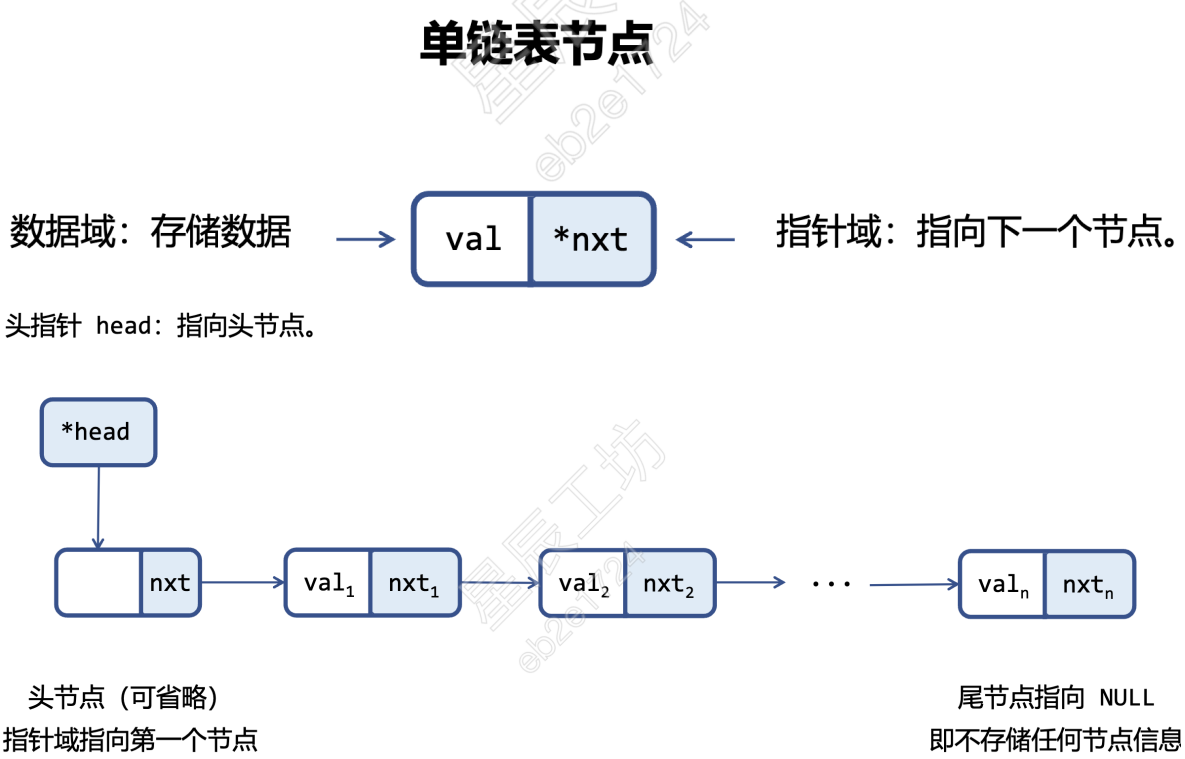
0.标准模版

1.线性数据结构

- 线性数据结构是最常用的数据结构，元素之间存在看一对一的线性关系，要注意，数据结构的实现和原理是分开的，要辨别当前讲的内容是原理还是实现。
- 从存储方式可分为/实现方式
  - 顺序存储结构/顺序表：其中顺序存储结构中用一组地址连续的存储单元依次存储元素，它的特点是逻辑关系上相邻的两个元素 在物理位置上也相邻， 因此可以随机存取表中的任一元素；
  - 链式存储结构/链表：链式存储结构中元素随机存储，每个元素都有指针用于指向自己的直接后继元素，常见的单链表、 双向链表、 循环链表采用链式存储
- 从概念上可以区分数组，链表，栈，队列。

2. 单链表的定义

- 单链表的定义 链表是一种非常重要的数据结构，它可以动态地分配内存空间，实现高效的数据操作，单链表属于链表的一种
  单链表由一系列的节点构成，每个节点包含了数据和一个指向下一个节点的指针



关于头节点和头指针：
头指针：

1. 头指针是链表的必要元素
2. 无论链表是否为空，头指针均不为空；
3. 头指针是指向链表第一个节点的指针、若链表有头节点， 则是指向头节点的指针 头节点：
4. 头节点不是链表的必须要素；
5. 头节点是为了操作的统一和方便而设立的，放在第一个节点之前，其数据域一般不存储数据，但也可以用来存放链表的长度
6. 有了头节点， 对第一节点执行删除、 插入等操作时，不用实时更新头指针的指向，对第一节点的操作也能和其他节点的操作统一

3. 单链表的模拟

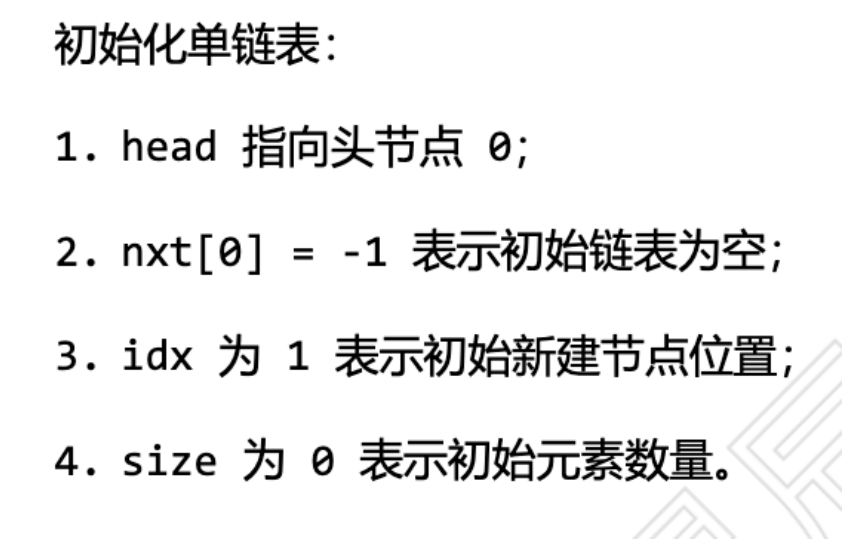
- 使用静态数组进行模拟

  使用静态数组进行模拟，根据单链表基本结构：

  1. 定义 max_size 为链表最大长度；
  2. 定义 head 存储头节点地址模拟头指针；
  3. 定义 val 数组存储节点的值，模拟数据域；
  4. 定义 nxt 数组存储节点间的指向关系，模拟指针域；
  5. 定义 idx 存储新节点位置；
  6. 定义 size 存储元素数量。

```
const int max_size = 1001;
int head;
int val[max_size];
int nxt[max_size];
int idx;
int size;
```

- 初始化单链表



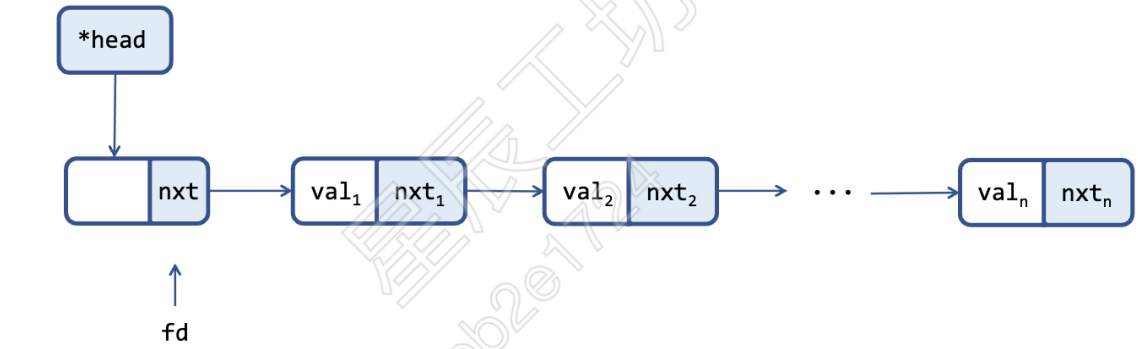

```
void init()
{

    head = 0;
    nxt[0] = -1'
    idx = 1;
    size = 0;
}
```

空链表：当链表中仅有头指针和头节点或者仅有头指针的时候，此时链表为空。

4. 单链表的查找与插入

- 单链表的查找



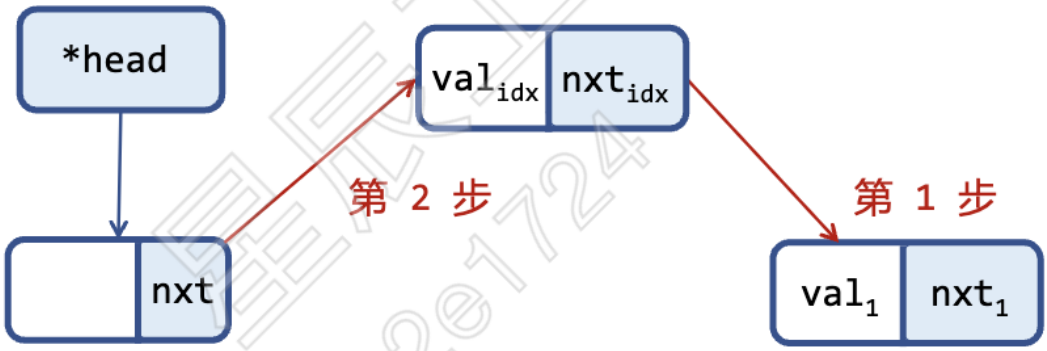
fd 模拟指针从头节点开始查找

传入 pos：

1. 查找位置不存在时，做出提示；
2. 否则从头节点开始查找；
3. 找到第 pos 个节点后停止；
4. 返回第 pos 个节点的位置。

```
// 查找节点
int find(int pos)
{
    if (pos > size || pos < 0){
        cout << "查找位置不存在";
        return -1;
    }
    else{
        int fd = head;
        for (int i = 1; i <= pos; i++){
            fd = nxt[fd];
        }
        return fd;
    }
}
```

- 单链表的插入
  - 头插法



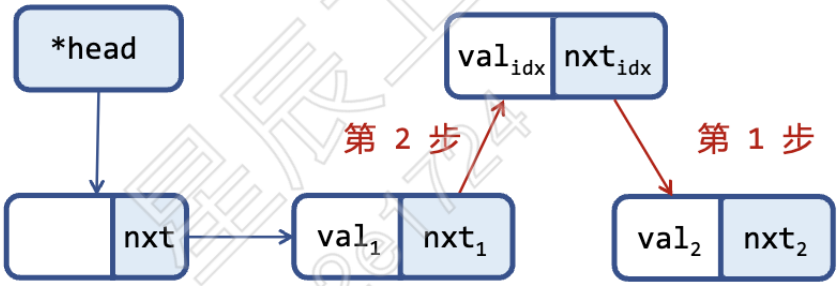
头插法：将新节点插入到头部。

1. 做出插入位置的判断；
2. idx 为新节点位置；
3. 存储的值保存在新节点位置。
4. 将新节点的 nxt 指向头节点的 nxt；
5. 将 head 的 nxt 指向新节点，作为新的第一个节点；
6. idx++ 作为下一个新节点位置；
7. size++。

```
// 头插法
void push_front(int x){
    if (idx > max_size - 1)
        cout << "存储位置已超过范围" << endl;
    else{
        val[idx] = x;
        nxt[idx] = nxt[head];
        nxt[head] = idx;
        idx++;
        size++;
    }
}
```

- 中间插入法

例如：插入到第 2 个节点前。



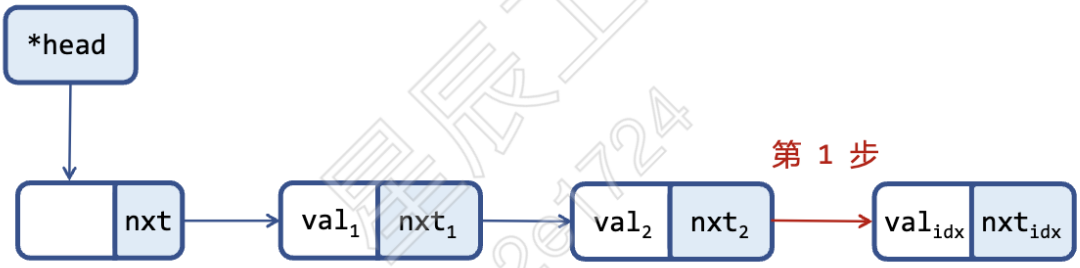
中间插入法：将新节点插入到节点之前。

1. 做出插入位置的判断；
2. 插入到节点之前，通过循环找到第 pos-1 个节点；
3. 保存新节点存储的值；
4. 将新节点的 nxt 先指向第 pos 节点；
5. 再将 pos-1 节点的 nxt 指向 idx；
6. idx++ 作为下一个新节点位置；
7. size++。

```
// 中间插入
void insert(int pos, int x){
    if (pos > size || pos <= 0)
        cout << "插入位置不存在" << endl;
    else if (idx > max_size - 1)
        cout << "存储位置已超过范围" << endl;
    else{
        int temp = find(pos - 1);
        val[idx] = x;
        nxt[idx] = nxt[temp];
        nxt[temp] = idx;
        idx++;
        size++;
    }
}
```

- 尾插法

尾插法：将新节点插入到尾部



尾插法：将新节点插入末尾。

1. 做出插入位置的判断；
2. 找到尾节点；
3. 保存新节点存储的值；
4. 将尾节点的 nxt 指向新节点；
5. 新节点的 nxt 指向 -1；
6. idx++ 作为下一个新节点位置
7. size++。

```
// 尾插法
void push_back(int x){
    if (idx > max_size - 1)
        cout << "存储位置已超过范围" << endl;
    else{
        int temp = head;
        while (nxt[temp] != -1)
            temp = nxt[temp];
        val[idx] = x;
        nxt[temp] = idx;
        nxt[idx] = -1;
        idx++;
        size++;
    }
}
```