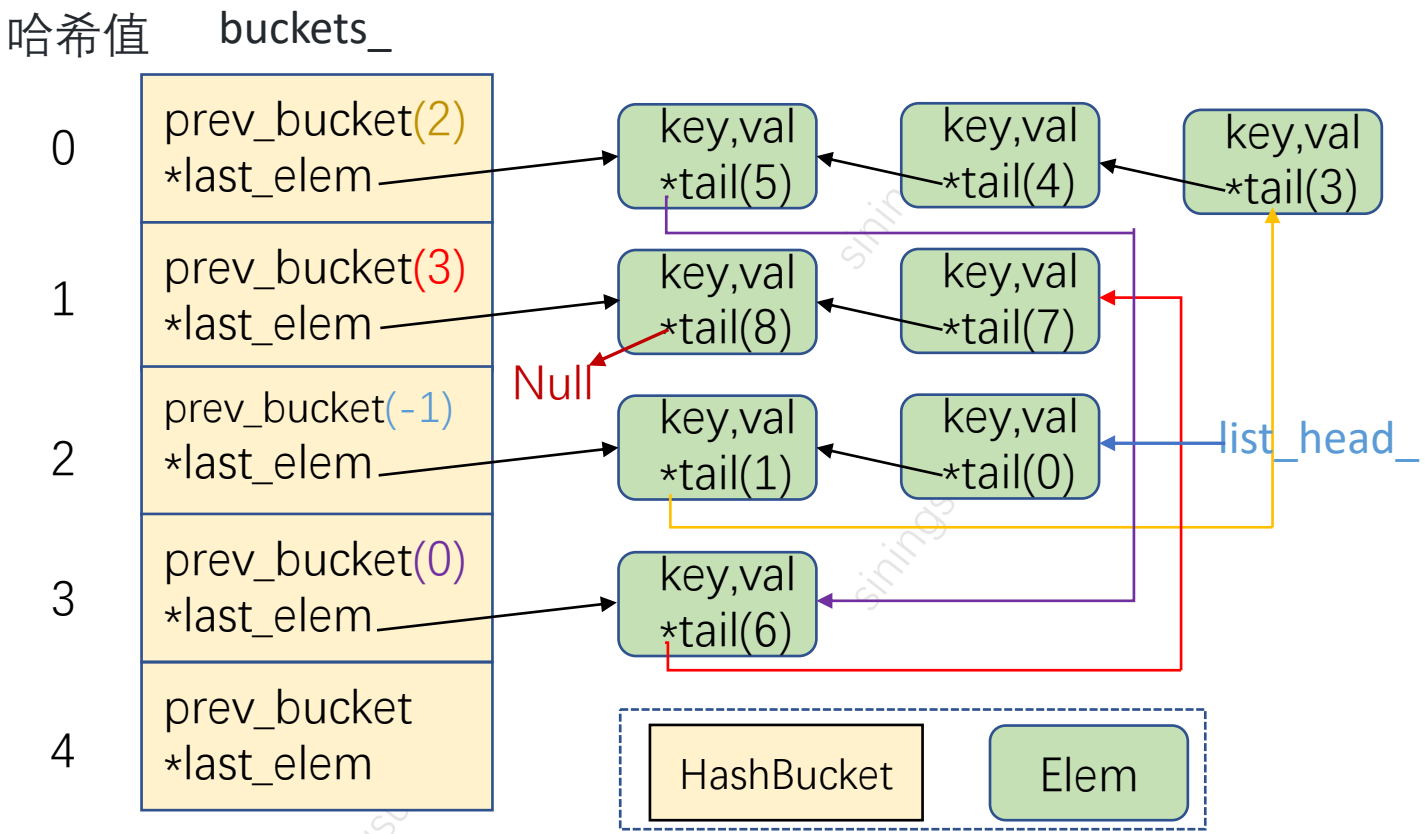
Kaldi FasterDecoder 中的HashList

https://github.com/kaldi-asr/kaldi/blob/master/src/util/hash-list.h

https://github.com/kaldi-asr/kaldi/blob/master/src/util/hash-list-inl.h



HashList

HashList是Kaldi专门为解码器设计的一种数据结构，他是哈希表和单项链表的一个结合，在解码这个任务上，使用HashList的好处大概有这么几点：

1. 使用一个HashList就可以维护SimpleDecoder中的prev_toks_和cur_toks_两个token表
2. 进行了分块内存管理，解码过程中不需要频繁的申请和释放内存

HashList中定义了一个HashBucket概念，所有哈希值相同的元素可以认为在同一个HashBucket里的单向链表

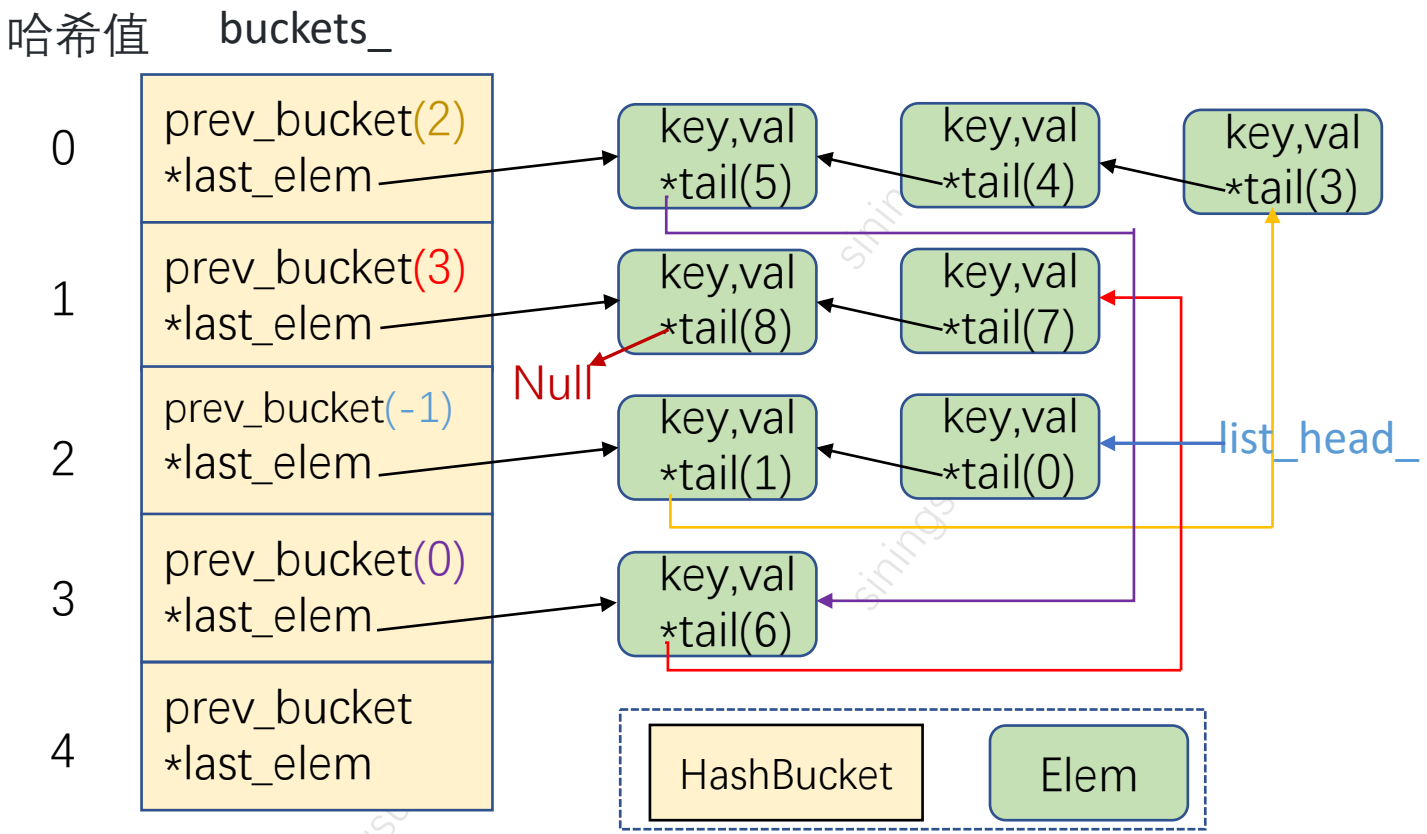
左图给出了一个HashList的示意图，并在图上标注了HashList中每个成员变量表示的意义

```
132     Elem *freed_head_;  // head of list of currently freed ele
133     // allocation]
134
135     std::vector<Elem*> allocated_;  // list of allocated block
```

所有的Elem实际都存储在allocated_里面，freed_head_指向allocated_里面被分配但是还没被使用的Elem

看起来这个数据结构很复杂！一步一步的解析这个表，其实就会发现很简单！

简单来说，这就是一个以list_head_为头的单项链表，链表中的某些元素可以通过哈希值直接访问！



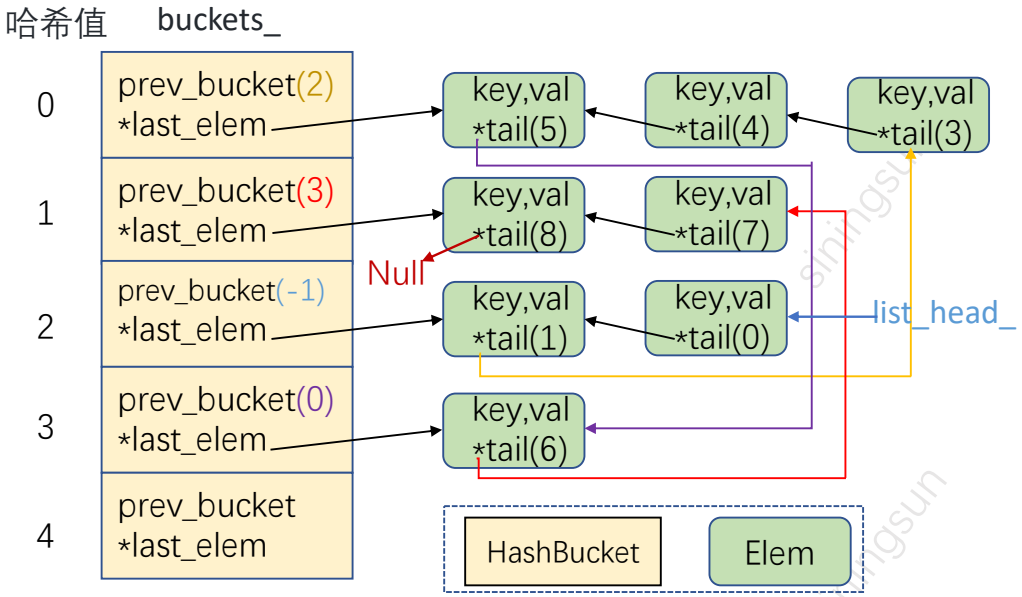
HashList中元素的访问与插入Insert

具体到解码图中使用HashList的时候，Elem中的key就是StateId，val就是Token。SimpleDecoder中已经提到，某个StateId里面在当前时刻只会记录得分最优的Token，所有在HashList中查找一个Elem，如果链表中存在这个StateId的Elem，就会比较两个Elem的cost，选取得分最优的留下。

不要将Elem的tail指针和Token的prev_混淆，Elem中的val变量就是Token，每个Token都有一个prev_指针指向上一步的Token

下面具体的来解析一下左图的HashList这个结构

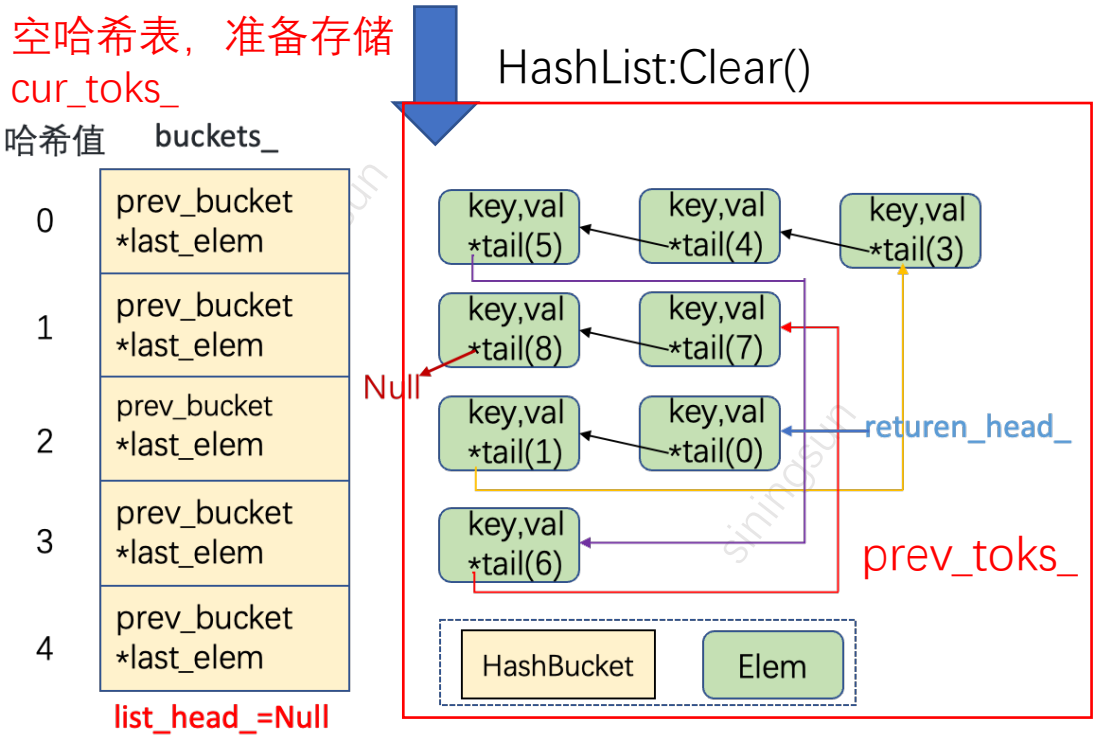
1. 首先看从list_head_出发，沿着tail指针，可以把整个链表都访问到！比如(1)这个Elem，它的tail指针（黄色箭头）指向了第0个bucket的链表的头，也就是（3）这个Elem；继续沿着（3）->（4）->（5），（5）这个Elem的tail指针（紫色箭头）指向了第3个bucket的链表头，也就是（6）这个Elem。
2. 再来看我们如果想插入某一个<StateId, Token>到这个HashList中，假设根据StateId计算出来的哈希值是0，那就应该插入到哈希值为0对应的bucket的链表中；此时我们需要遍历bucket0这个链表看StateId对应的Elem是不是存在；而从bucket0这个链表中，我们只能找到链表的最后一个Elem，就是last_elem指向的元素（5），但是我们无法从（5）访问其他元素，这时候就需要知道bucket0这个链表的头指针，也就是（3）这个元素；我们看到prev_bucket_里面是2，它表示这个bucket的头指针可以通过bucket2的last_elem指向的Elem的tail来获得！如果prev_bucket_=-1，那么通过list_head_就可以访问到当前bucket的所有元素



HashList的Clear操作

HashList的Clear操作，其实并不是删除HashList中的所有的元素，而是将哈希表和以list_head_为头指针的单链表断开连接，把单链表的list_head_返回给解码器（55-57）。而断开二者的链接也很简单，遍历所有的bucket，把每个bucket的last_elem指向Null就可以（49-53）

每次解码新的一帧数据的时候，首先进行HashList的Clear操作之后得到链表头，这个链表里存储的内容就是SimpleDecoder里面的prev_toks_；此时哈希表已经空了，cur_toks_重新存入HashList里即可！




```
45  template<class I, class T>
46  typename HashList<I, T>::Elem* HashList<I, T>::Clear() {
47    // Clears the hashtable and gives ownership of the currently contained list
48    // to the user.
49    for (size_t cur_bucket = bucket_list_tail_;
50         cur_bucket != static_cast<size_t>(-1);
51         cur_bucket = buckets_[cur_bucket].prev_bucket) {
52      buckets_[cur_bucket].last_elem = NULL;  // this is how we indicate "empty".
53    }
54    bucket_list_tail_ = static_cast<size_t>(-1);
55    Elem *ans = list_head_;
56    list_head_ = NULL;
57    return ans;
58  }
59
```

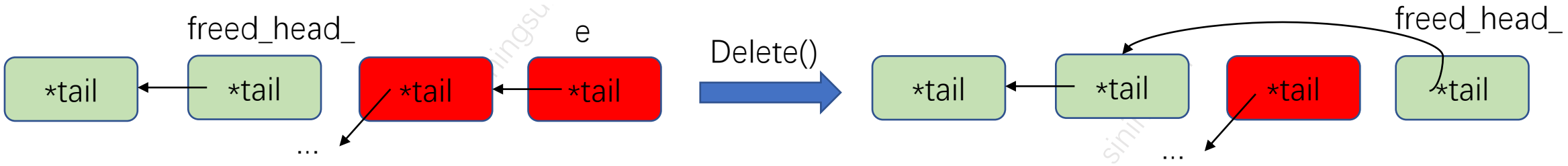
```
88  template<class I, class T>
89  inline typename HashList<I, T>::Elem* HashList<I, T>::New() {
90    if (freed_head_) {
91      Elem *ans = freed_head_;
92      freed_head_ = freed_head_->tail;
93      return ans;
94    } else {
95      Elem *tmp = new Elem[allocate_block_size_];
96      for (size_t i = 0; i+1 < allocate_block_size_; i++)
97        tmp[i].tail = tmp+i+1;
98      tmp[allocate_block_size_-1].tail = NULL;
99      freed_head_ = tmp;
100     allocated_.push_back(tmp);
101     return this->New();
102   }
103 }
```

HashList的New和Delete

HashList的New操作，返回一个Elem的指针，New方法内不一定会新开辟内存。第90-93，如果freed_head_不是Null，表示目前allocated_内还有可用空间来存放Elem，freed_head_ 沿着tail后移就可以获得一个开辟但是未被使用的空间；
如果freed_head_为Null，表示目前没有足够的使用空间，95行新开辟一个大小为allocate_block_size_的类型为Elem的空间，并且形成链表（95-98），将新分配的空间push_back到allocated_里面，freed_head_指向新分配到空间的开始地方；
对101行，递归的调用了New，此时freed_head_不是Null，又返回到90-93行代码。

```
65  template<class I, class T>
66  inline void HashList<I, T>::Delete(Elem *e) {
67    e->tail = freed_head_;
68    freed_head_ = e;
69  }
```

HashList的Delete的操作，并不是释放内存，删除某个元素e，具体的做法就是



End