**skip list（跳表）**

文／littlersmall（简书作者）  
原文链接：http://www.jianshu.com/p/fd4a4248cf2d  
著作权归作者所有，转载请联系作者获得授权，并标注“简书作者”。

第一次看到这种数据结构还是刚接触ocean base架构的时候。粗略扫了几眼，以为是一个简单的二级索引，没有仔细考虑就略过了。后来去北京出差，经神夜路点播，遂明白这种链表式结构的简约而不简单，有一种四两拨千斤的优雅。

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.  
--William Pugh

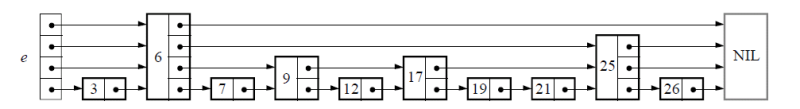
相比于红黑树，B树，AVL树，跳表的实现相当简单，同时，由于其多维链表的特性，使得跳表可以支持无锁的多读一写。（链表的多读一写无锁实现方式这里就不展开了）。  
不同于B树，跳表的平衡性依靠随机算法，在正常情况下，该结构的查找，插入，删除的时间复杂度都是logN。  
先从一维链表开始，我们知道在链表中查找一个元素I的话，需要将整个链表遍历一次。

http://upload-images.jianshu.io/upload_images/1397675-186fe5469317cced.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240

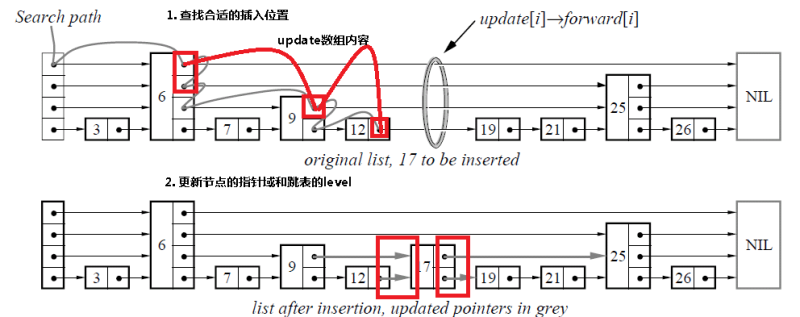
如果是说链表是排序的，并且节点中还存储了指向前面第二个节点的指针的话，那么在查找一个节点时，仅仅需要遍历N/2个节点即可。

http://upload-images.jianshu.io/upload_images/1397675-0139030353ec9c89.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240

这基本上就是跳表的核心思想，其实也是一种通过“空间来换取时间”的一个算法。  
下面我们来看一个4层跳表示例：



查找时首先从高层开始查找，之后逐渐降低层次靠近数据，完成定位。  
插入操作：



由于跳表数据结构整体上是有序的，所以在插入时，需要首先查找到合适的位置，然后就是修改指针（和链表中操作类似），然后更新跳表的level变量。  
补充一个数据节点层次确定算法：

int height = 1;

while (height < kMaxHeight && ((rnd\_.Next() % kBranching) == 0))

{

height++;

}

可以发现层级越高的节点越少，因此跳表整体的指针开销并不高。相比于同级别的树形实现，跳表具有更快的速度，更低的空间开销和更简单的实现。

/\*

·\* skipList.h

·\*

·\* ·Created on: 2013年8月7日

·\* · · ·Author: sigh.xy

·\*/

#ifndef SKIPLIST\_H\_

#define SKIPLIST\_H\_

#include <iostream>

#include <stack>

//rand

#include<stdlib.h>

template <class Key = int, class Value = int>

class Node

{

public:

Key key;

Value value;

Node(Key k, Value v) : key(k), value(v){}

Node(){}

};

template <class Key = int, class Value = int>

class Element

{

Node<Key, Value> node;

Element\*\* next;

public:

Element() : next(NULL) {}

Element(Node<Key, Value> node, int level)

{

this->node = node;

next = new Element\*[level];

for (int i = 0; i < level; i++)

{

next[i] = NULL;

}

}

void setNext(int place, Element\* nElement)

{

next[place] = nElement;

}

Element\* & getNext(int place)

{

return next[place];

}

Key getKey()

{

return node.key;

}

Value getValue()

{

return node.value;

}

~Element()

{

if (next)

{

delete[] next;

}

}

};

//declare

template <class Key, class Value>

class SkipIterator;

template <class Key = int, class Value = int, int MAXLEVEL = 4>

class SkipList

{

//head

Element<Key, Value>\*\* head;

int randLevel(int level = MAXLEVEL);

void findWay(Key key, std::stack<Element<Key, Value>\*\* >& pStack);

public:

typedef SkipIterator<Key, Value> Iterator;

SkipList()

{

head = new Element<Key, Value>\*[MAXLEVEL];

for (int i = 0; i < MAXLEVEL; i++)

{

head[i] = NULL;

}

}

Value find(Key key);

bool insert(Key key, Value value);

Iterator begin()

{

return head[0];

}

Iterator end()

{

return Iterator();

}

//another kind of insert

//bool delKey(Key key);

~SkipList()

{

Element<Key, Value>\* cur = head[0];

while (cur)

{

Element<Key, Value>\* tmp = cur;

cur = cur->getNext(0);

delete tmp;

}

}

};

//查找数据

template <class Key, class Value, int MAXLEVEL>

Value SkipList<Key, Value, MAXLEVEL>::find(const Key key)

{

if (NULL == head)

{

return (Value) 0;

}

//std::cout << "ok" << std::endl;

int rawL = MAXLEVEL - 1;

Element<Key, Value>\* cur = NULL;

//find the first < place

while (rawL >= 0)

{

if (head[rawL] && head[rawL]->getKey() == key)

{

return head[rawL]->getValue();

}

else if (head[rawL] && head[rawL]->getKey() < key)

{

cur = head[rawL];

break;

}

rawL--;

}

//std::cout << "rawL = " << rawL << std::endl;

while (rawL >= 0)

{

if (cur && cur->getKey() == key)

{

return cur->getValue();

}

else if (cur->getNext(rawL) && cur->getNext(rawL)->getKey() <= key)

{

cur = cur->getNext(rawL);

}

else

{

rawL--;

}

}

return (Value) 0;

}

//通过栈记录查找路径，用于插入操作。

template <class Key, class Value, int MAXLEVEL>

void SkipList<Key, Value, MAXLEVEL>::findWay(Key key,

std::stack<Element<Key, Value>\*\* >& pStack)

{

int rawL = MAXLEVEL - 1;

Element<Key, Value>\* cur = NULL;

//find the first < place

while (rawL >= 0)

{

if (head[rawL] && head[rawL]->getKey() < key)

{

cur = head[rawL];

break;

}

pStack.push(&head[rawL]);

rawL--;

}

while (rawL >= 0)

{

if (cur->getNext(rawL) && cur->getNext(rawL)->getKey() <= key)

{

cur = cur->getNext(rawL);

}

else

{

pStack.push(&cur->getNext(rawL));

rawL--;

}

}

}

//插入操作

template <class Key, class Value, int MAXLEVEL>

bool SkipList<Key, Value, MAXLEVEL>::insert(Key key, Value value)

{

int level = randLevel();

Element<Key, Value>\* element =

new Element<Key, Value>(Node<Key, Value>(key, value), level);

std::stack<Element<Key, Value>\*\*> pStack;

findWay(key, pStack);

for (int i = 0; i < level; i++)

{

element->getNext(i) = \*pStack.top();

\*(pStack.top()) = element;

//std::cout << "head = " << head[0]->getValue() << std::endl;

pStack.pop();

}

//std::cout << head[0]->getNext(0) << std::endl;

return true;

}

template <class Key, class Value, int MAXLEVEL>

int SkipList<Key, Value, MAXLEVEL>::randLevel(int level)

{

int height = 1;

while (height < MAXLEVEL && ((rand() % level) == 0))

{

height++;

}

return height;

}

//iterator

template <class Key, class Value>

class SkipIterator

{

private:

Element<Key, Value>\* element;

public:

typedef Element<Key, Value> EType;

SkipIterator(Element<Key, Value>\* e) : element(e) {}

SkipIterator()

{

element = NULL;

}

EType& operator\*()

{

return \*element;

}

void operator++()

{

element = element->getNext(0);

}

void operator++(int)

{

++\*this;

}

bool operator!= (SkipIterator<Key, Value> right)

{

return this->element != right.element;

}

};

#endif /\* SKIPLIST\_H\_ \*/

基于模版的某种实现方式没有做很严格的测试，因此不保证正确性。