binomial queue

1.why?

average time for **insertions** for leftist or skew heaps? O(LogN)

插入N个元素总时间O(N),build heap O(N)时间，平均时间是常数

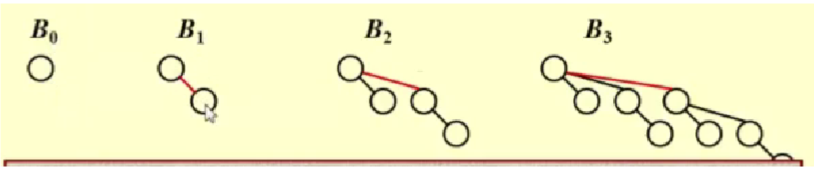
=>Log N不是一个好的时间，引入binomial

2.definition

A **binomial queue** is not a heap-ordered tree, but rather a collection of **heap-ordered** trees, known as a **forest**. Each heap-ordered tree is a binomial tree.

binomial tree:

A binomial tree of height 0 is a one-node tree.
A binomial tree, Bk, of height k is formed by attaching a binomial tree, Bk – 1, to the root of another binomial tree, Bk – 1.

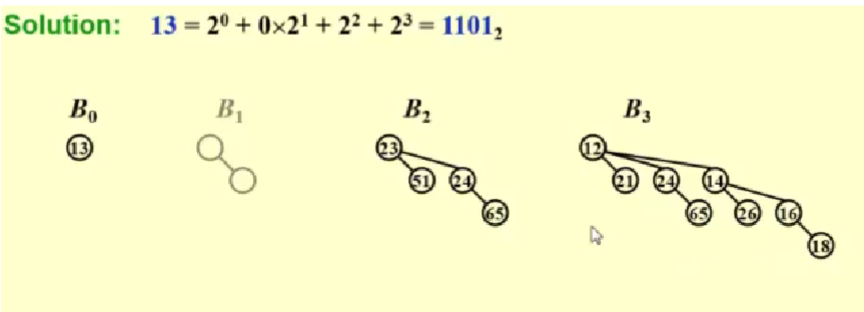


Bk 有2^k 个节点，root有k个节点，depth d 有C k（下标） d（上标）个节点 d:depth

3.inference

任何一个优先队列都可以用一系列binomial tree **唯一**表示

eg: size=13



4.operations

4.1 FindMin

scan root and find the smallest

N个结点，最多logN （向上取整）个root，

可以存储最小值，需要更新时更新一下，时间复杂度O(1);

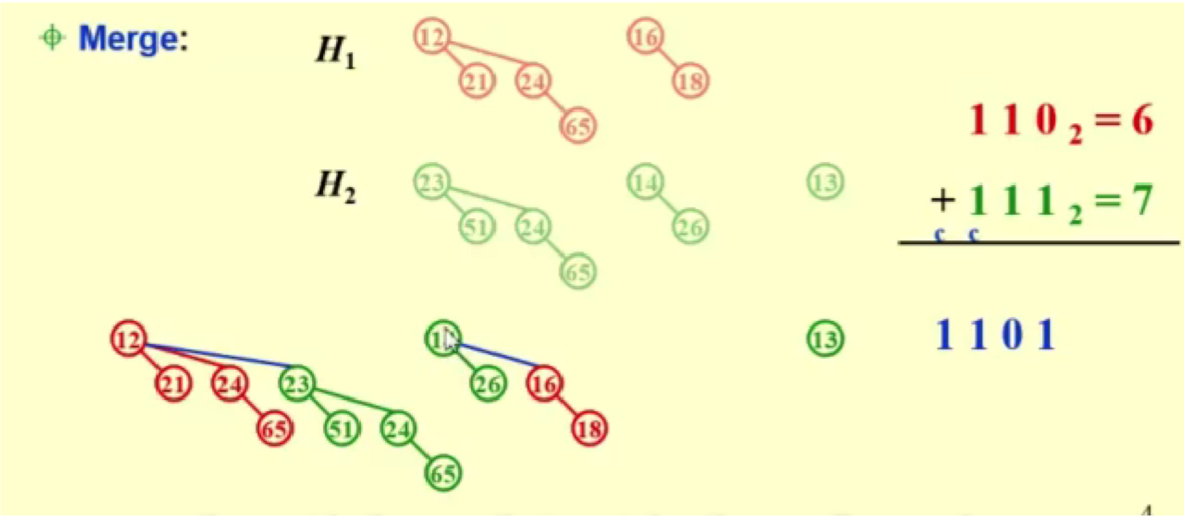
4.2 Delete Min(H)

update 的时间复杂度O(Log N),所以不存储最小值也一样，时间复杂度就是 O(Log N)

step:

1. findmin in H O(LogN)
2. remove the Bk, 获得k个子树 O(1)
3. remove root from Bk O(LogN) 最多有k个子节点，k最大是O(LogN)
4. 和forest里其他tree 进行merge O(LogN)

4.3 merge

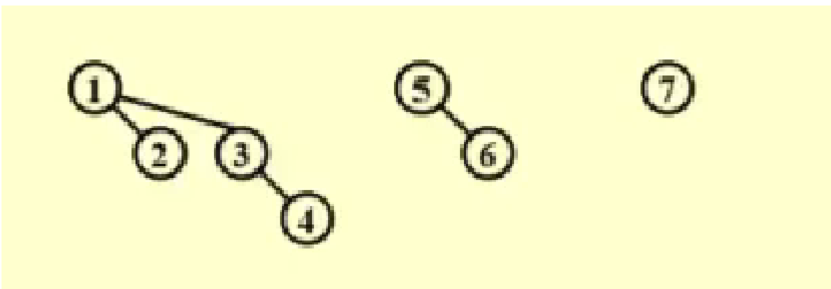


思路与加法相似，根据height从低到高开始merge,进位

所以存储binomial queue按照size排序存，而不是按照root大小

4.4 insert

1，2，3，4，5，6，7

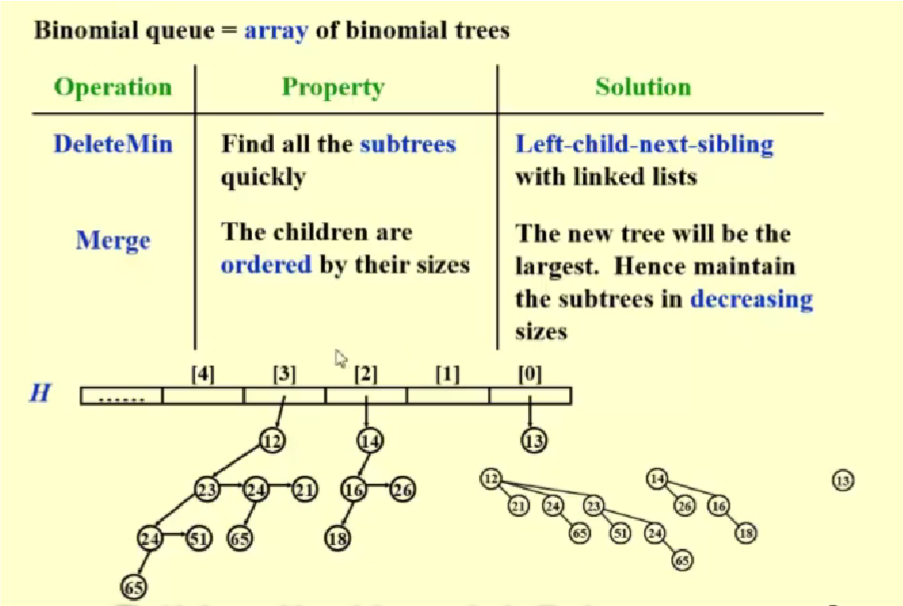


插入的最坏时间复杂度是O(N)

4、implementation

left-child next-sibling: 左找到最大子树，右找到各种sibling

decreasing order: merge two tree 不用遍历寻找size of largest

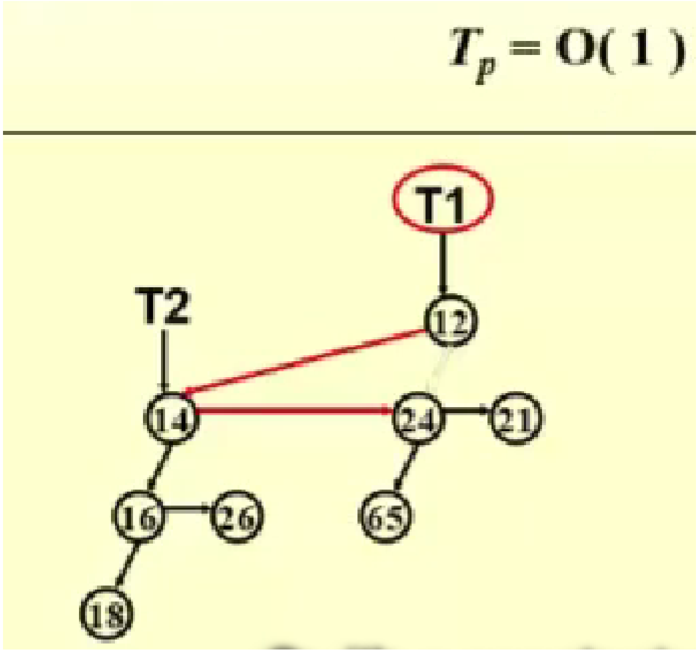



```c
//basic structure
typedef struct BinNode *Position;
typedef struct Collection *BinQueue;
typedef struct BinNode *BinTree;  /* missing from p.176 */

struct BinNode
{
    ElementType     Element;
    Position        LeftChild;
    Position        NextSibling;
};

struct Collection
{
    int         CurrentSize;  /* total number of nodes */
    BinTree     TheTrees[ MaxTrees ];
};
```

```c
//combine
BinTree
CombineTrees( BinTree T1, BinTree T2 )
{  /* merge equal-sized T1 and T2 */
    if ( T1->Element > T2->Element )
        /* attach the larger one to the smaller one */
        return CombineTrees( T2, T1 );
    /* insert T2 to the front of the children list of T1 */
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}
```



时间复杂度O(1)

```
//merge
BinQueue  Merge( BinQueue H1, BinQueue H2 )
{   BinTree T1, T2, Carry = NULL;
    int i, j;
 //step1:error checking
    if ( H1->CurrentSize + H2-> CurrentSize > Capacity )  ErrorMessage();
 //step2:size change
    H1->CurrentSize += H2-> CurrentSize;
 //step3:i 0~logN 用j *=2实现
 //merge tree one by one
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
        switch( 4*!!Carry + 2*!!T2 + !!T1 ) {  // !! : null 0 ;not null 1
              //4: 第三位    2:第二位  1:第一位
        case 0: /* 000 */
        case 1: /* 001 */  break;
        case 2: /* 010 */  H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
        case 4: /* 100 */  H1->TheTrees[i] = Carry; Carry = NULL; break;
        case 3: /* 011 */  Carry = CombineTrees( T1, T2 );
                        H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
        case 5: /* 101 */  Carry = CombineTrees( T1, Carry );
                        H1->TheTrees[i] = NULL; break;
        case 6: /* 110 */  Carry = CombineTrees( T2, Carry );
                        H2->TheTrees[i] = NULL; break;
        case 7: /* 111 */  H1->TheTrees[i] = Carry;
                        Carry = CombineTrees( T1, T2 );
                        H2->TheTrees[i] = NULL; break;
        } /* end switch */
    } /* end for-loop */
    return H1;
}
```

delete min

```
ElementType  DeleteMin( BinQueue H )
```

```
{   BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem = Infinity;  /* the minimum item to be returned */
    int i, j, MinTree; /* MinTree is the index of the tree with the minimum item
*/

    if ( IsEmpty( H ) )  {  PrintErrorMessage();  return Infinity; }

    /* Step 1: find the minimum item */
    for ( i = 0; i < MaxTrees; i++) {
        if( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) {
        MinItem = H->TheTrees[i]->Element;  MinTree = i;    } /* end if */
    } /* end for-i-loop */
    DeletedTree = H->TheTrees[ MinTree ];

    /* Step 2: remove the MinTree from H => H’ */
    H->TheTrees[ MinTree ] = NULL;

    /* Step 3.1: remove the root */
    OldRoot = DeletedTree;
    DeletedTree = DeletedTree->LeftChild;   free(OldRoot);

    /* Step 3.2: create H” */
    DeletedQueue = Initialize();
    DeletedQueue->CurrentSize = ( 1<<MinTree ) - 1;  /* 2^MinTree - 1 */
    for ( j = MinTree - 1; j >= 0; j-- ) {
        DeletedQueue->TheTrees[j] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[j]->NextSibling = NULL;
    } /* end for-j-loop */

    H->CurrentSize  -= DeletedQueue->CurrentSize + 1;

    /* Step 4: merge H’ and H” */
    H = Merge( H, DeletedQueue );
    return MinItem;
}
```

5.anylisis

A binomial queue of N elements can be built by N successive insertions in O(N) time.

proof1 : aggregate分析法

steps in total :N

links in total: k in average 2^k steps

Proof 1 (Aggregate):

B_0	/*step = 1 */
B_1	/*step = 1, link = 1 */
B_1 B_0	/*step = 1*/
B_2	/*step = 1, link = 2 */
B_2 B_0	/*step = 1*/
B_2 B_1	/*step = 1, link = 1*/
B_2 B_1 B_0	/*step = 1*/
B_3	/*step = 1, link = 3*/
B_3 B_0	/*step = 1*/
…… …… ……	

Total steps = N

Total links =

$$N\left(\frac{1}{4}+2\times\frac{1}{8}+3\times\frac{1}{16}+\ldots\right)$$
$$=O(N)$$

proof2: potential method

An insertion that costs c units results in a net increase of 2 – c trees in the forest.

C_i ::= cost of the ith insertion

Φ_i ::= number of trees *after* the ith insertion ($\Phi_0 = 0$)

$C_i + (\Phi_i - \Phi_{i-1}) = 2$ for all $i = 1, 2, \ldots, N$

Add all these equations up $\Rightarrow \sum_{i=1}^{N} C_i + \Phi_N - \Phi_0 = 2N$

$$\sum_{i=1}^{N} C_i = 2N - \Phi_N \le 2N = O(N)$$

■

$T_{worst} = O(\log N)$, but $T_{amortized} = 2$

Expensive insertions，remove trees, while cheap ones create trees.