Binomial Queue

☞ **Structure:**

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.

What i

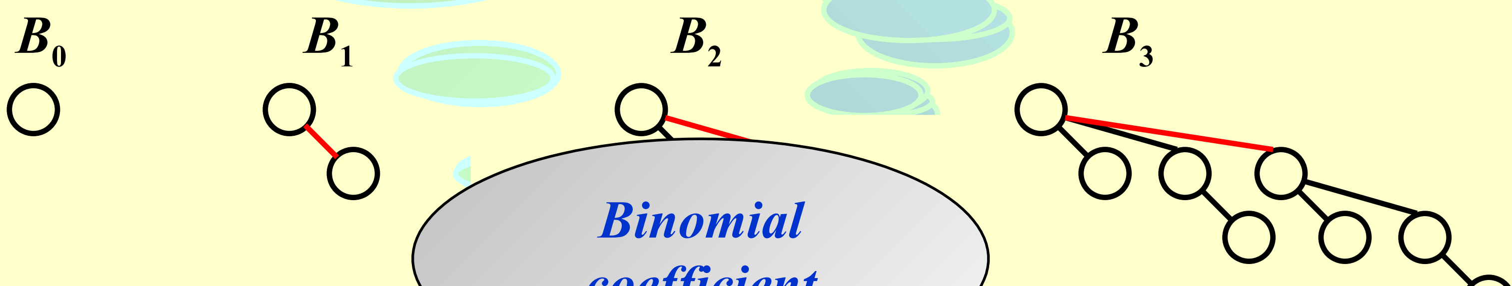
A binomial tree of height 0 is a one-node tree.

Then what is the avera

A binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}.

Constant!

So O(log N) for an insertion is NOT good enough!



Binomial coefficient

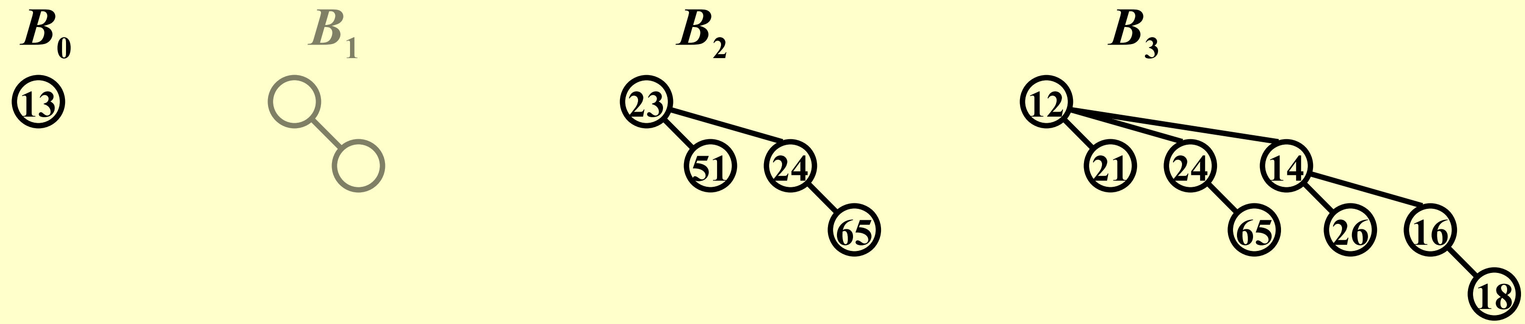
Observation: B_k co… t with $\underline{\;k\;}$ children, which are $\underline{B_0, B_1, \ldots, B_{k-1}}$. B_k has exactly $\underline{\;2^k\;}$ nodes. The number of nodes at depth d is $\underline{\binom{k}{d}}$.

B_k structure + heap order + one binomial tree for each height

➡ **A priority queue of any size can be uniquely represented by a collection of binomial trees.**

【 Example 】 Represent a priority queue of size 13 by a collection of binomial trees.

Solution: $13 = 2^0 + 0 \times 2^1 + 2^2 + 2^3 = 1101_2$

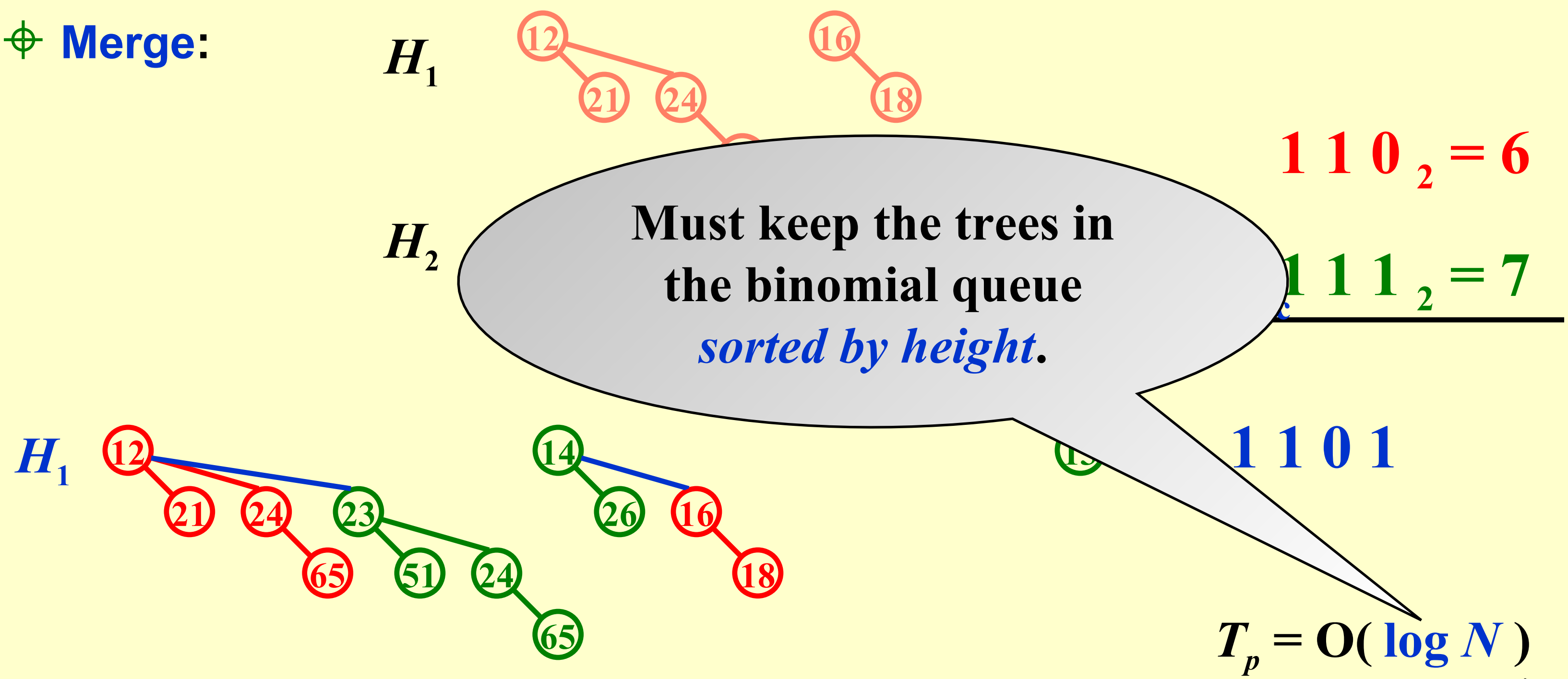


☞ Operations:

⌖ **FindMin**: The minimum key is in one of the **roots**.
There are at most $\lceil \log N \rceil$ roots, hence $T_p = O(\log N)$.

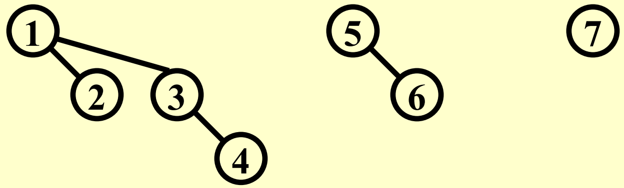
Note: We can remember the minimum and update whenever it is changed. Then this operation will take O(1).

⌖ **Merge**:



⌖ **Insert**: a special case for merging.

【 Example 】 Insert 1, 2, 3, 4, 5, 6, 7 into an initially empty queue.



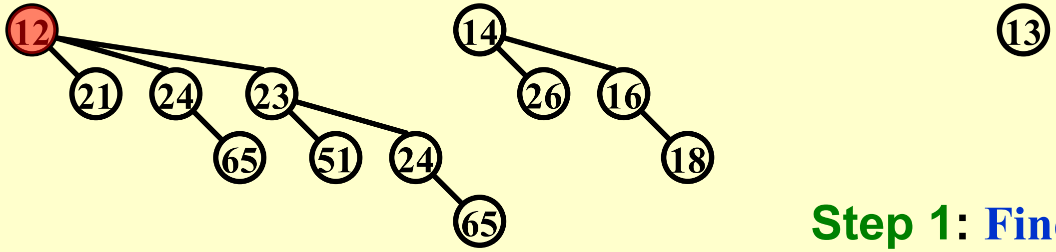
Note:

If the smallest nonexistent binomial tree is B_i, then $T_p = Const \cdot (i + 1)$.

Performing N **Inserts** on an initially empty binomial queue will take $O(N)$ worst-case time. Hence the average time is constant.

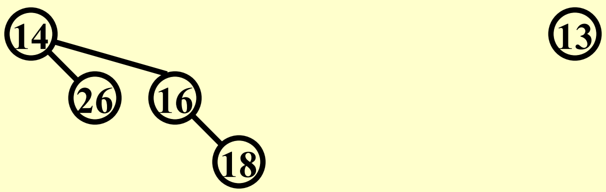
⌖ **DeleteMin (H):**

H



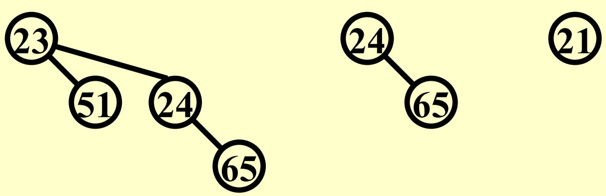
Step 1: FindMin in B_k
/* O(log *N*) */

H'



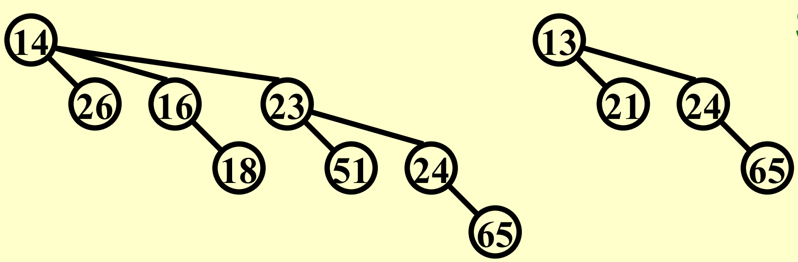
Step 2: Remove B_k from *H*
/* O(1) */

H''



Step 3: Remove root from B_k
/* O(log *N*) */

H

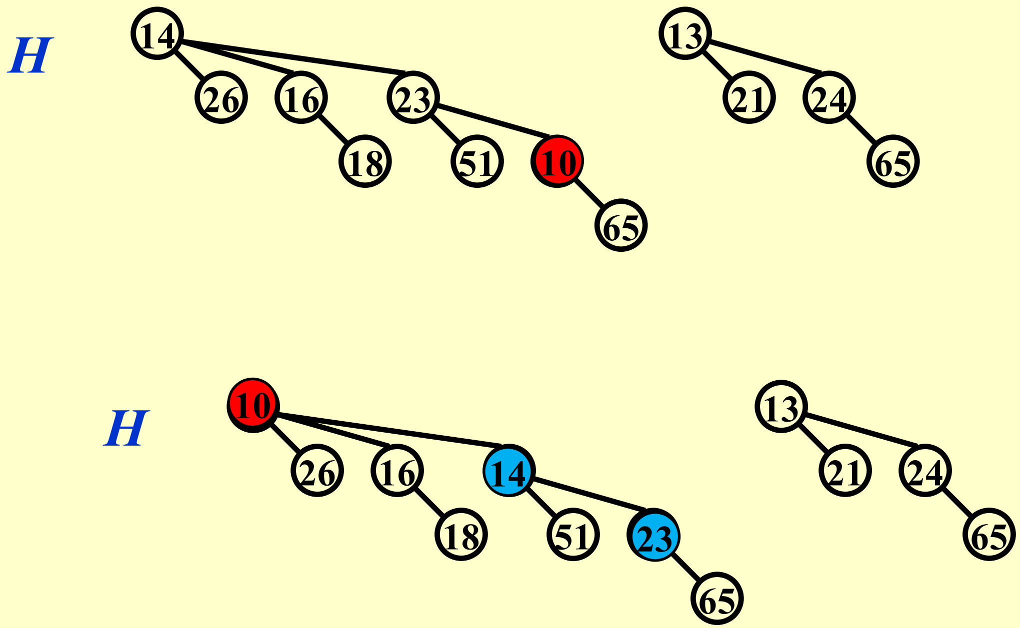


Step 4: Merge (*H'*, *H''*)
/* O(log *N*) */

⌖ Decrease key (H):

Given a handle to an element x in H, decrease its key to k.

- Suppose x is in binomial tree Bk.
- Repeatedly exchange x with its parent until heap order is restored.



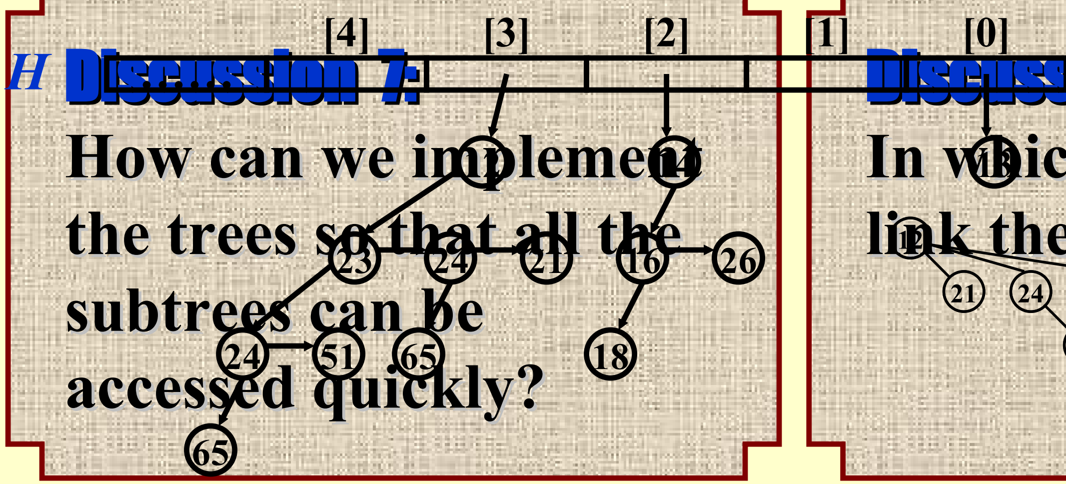
☞ Implementation:

Binomial queue = array of binomial trees

Operation	Property	Solution
DeleteMin	Find all the subtrees quickly	Left-child-next-sibling with linked lists
Merge	The children are ordered by their sizes	The new tree will be the largest. Hence maintain the subtrees in decreasing sizes

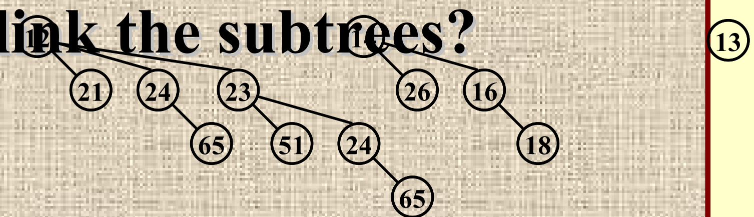
Discussion 7:

How can we implement the trees so that all the subtrees can be accessed quickly?



Discussion 8:

In which order must we link the subtrees?



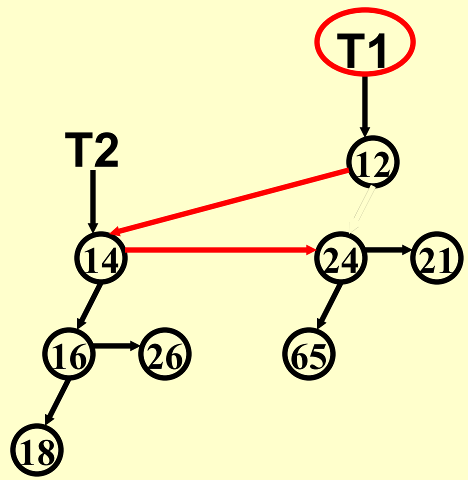
```
typedef struct BinNode *Position;
typedef struct Collection *BinQueue;
typedef struct BinNode *BinTree;  /* missing from p.176 */

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

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

```
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;
}
```

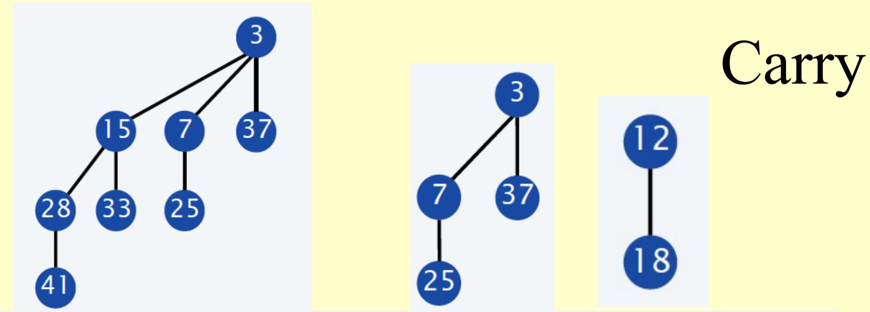
$T_p = O(1)$

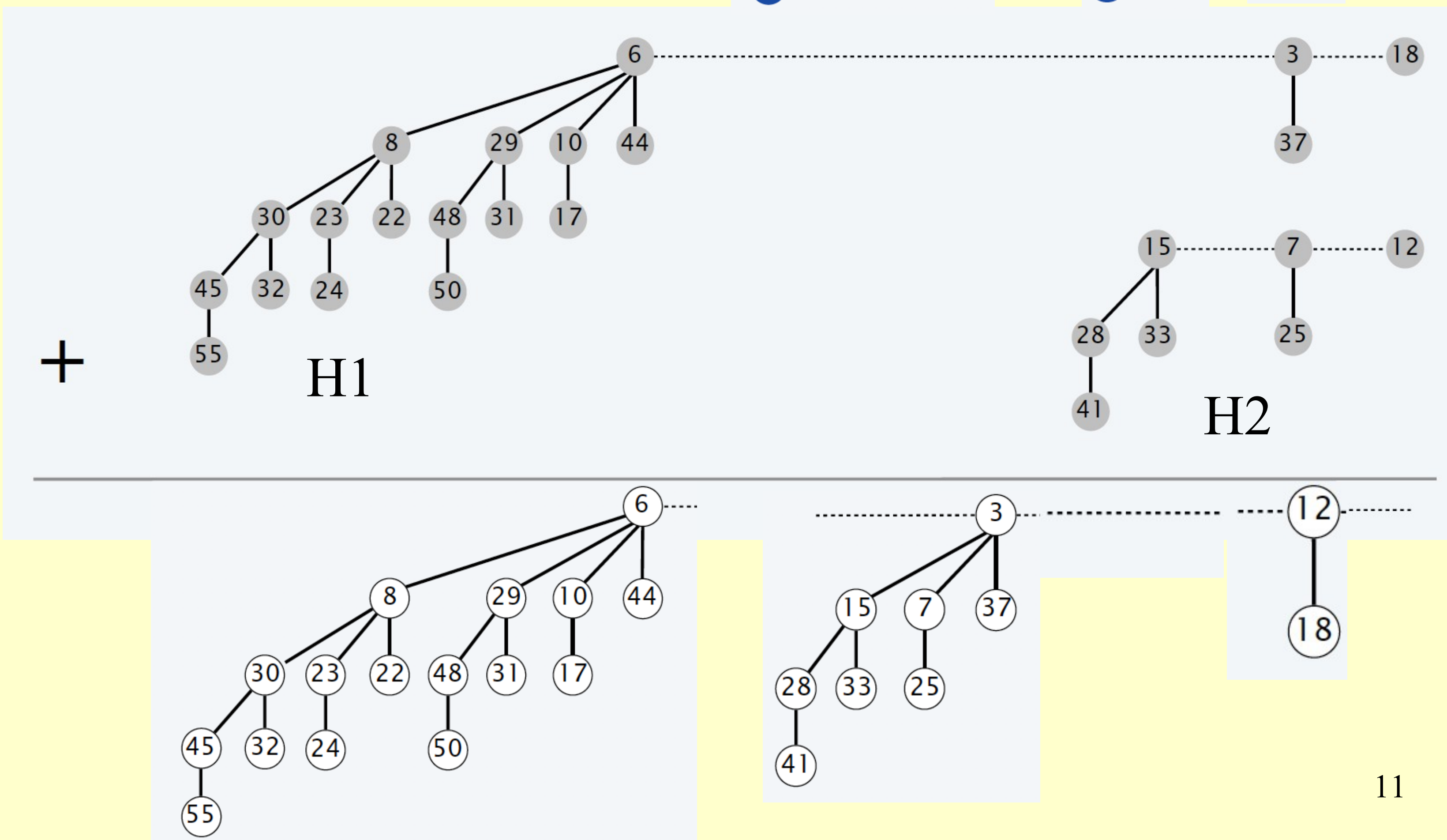


Merge(BinQueue H1, BinQueue H2)

19 + 7 = 26

		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0





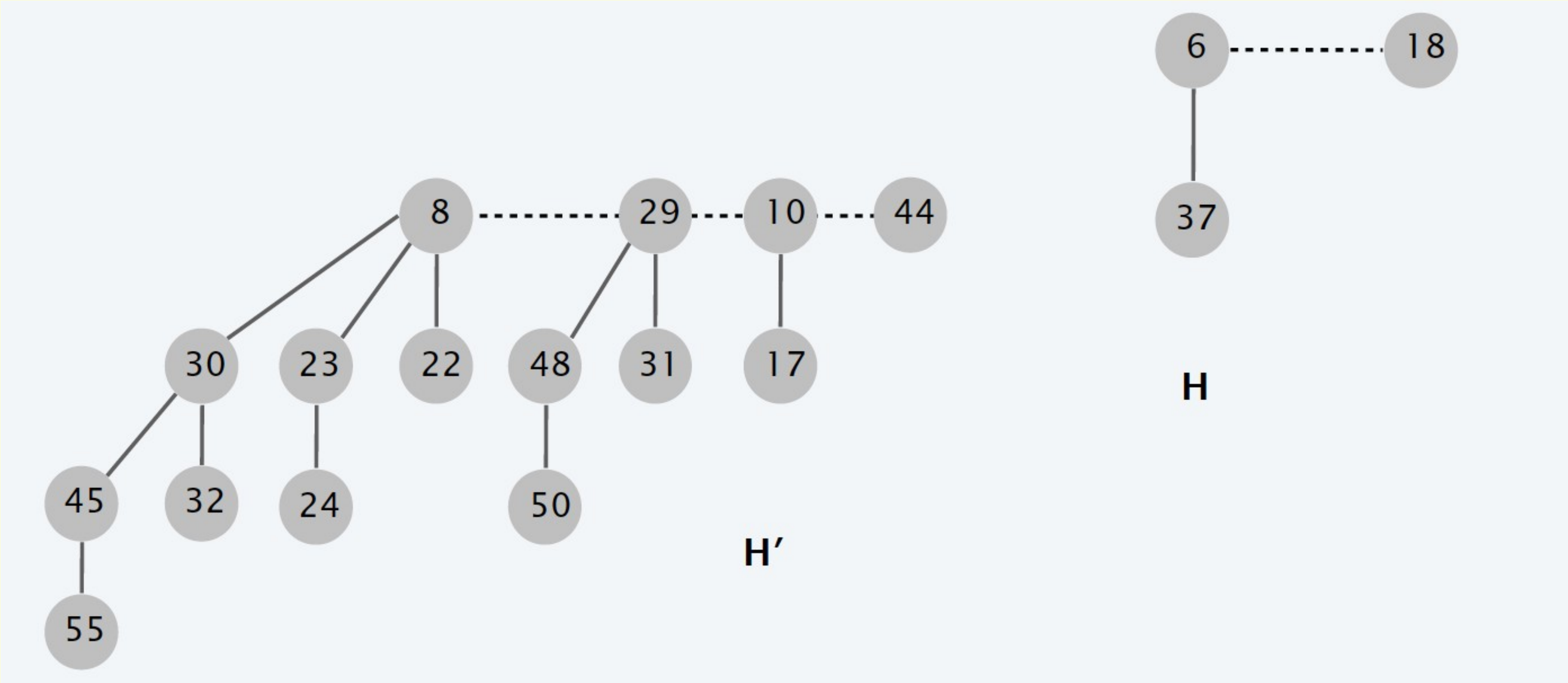
```
BinQueue  Merge( BinQueue H1, BinQueue H2 )
{   BinTree T1, T2, Carry = NULL;
    int i, j;
    if ( H1->CurrentSize + H2-> CurrentSize > Capacity )  ErrorMessage();
    H1->CurrentSize += H2-> CurrentSize;
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
       T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
       switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
    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;
}
```

/* assign each digit to a tree */

Carry	T2	T1

DeleteMin(BinQueue H)

- Find root x with min key in root list of H, and delete.
- H' ← broken binomial trees.
- H ← Merge(H', H).



```
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; }

     for ( i = 0; i < MaxTrees; i++) {  /* Step 1: find the minimum item */
        if( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) { 
     MinItem = H->TheTrees[i]->El    ent;  MinTree = i;    } /* end if */
     } /* end for-i-loop */
     DeletedTree = H->TheTrees[ MinTr
     H->TheTrees[ MinTree ] = NUL
     OldRoot = DeletedTree;   /* Step
     DeletedTree = DeletedTree->LeftChi
     DeletedQueue = Initialize();   /* Step 3.2: create H"  */
     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;
     H = Merge( H, DeletedQueue ); /* Step 4: merge H' and H" */
     return MinItem;
}
```

This can be replaced by
the actual number of roots

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

Proof 1 (Aggregate):

Insert. How much work to insert a new node x?

- If $n =$ ....0, then only 1 credit.
- If $n =$ ....01, then only 2 credits.
- If $n =$ ....011, then only 3 credits.
- If $n =$ ....0111, then only 4 credits.

$\times\frac{1}{16}+\ldots)$

Theorem. Starting from an empty binomial heap, a sequence of n consecutive INSERT operations takes $O(n)$ time.

Pf. $(n / 2)\,(1) + (n / 4)(2) + (n / 8)(3) + \ldots \;\le\; 2\,n.$ ▪

$$\sum_{i=1}^{k} \frac{i}{2^i} = 2 - \frac{k}{2^k} - \frac{1}{2^{k-1}} \le 2$$

Proof 2: **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 (number of trees merged + 1)

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

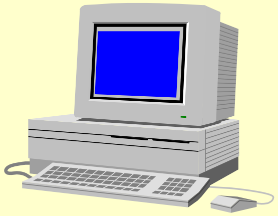
$C_i + (\Phi_i - \Phi_{i-1}) = 2$ for all $i = 1, 2, \ldots, N$

1 – number of trees merged

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

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

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



Research Project 2
Shortest Path Algorithm with Heaps (26)

DIMACS

Center for Discrete Mathematics & Theoretical Computer Science
Founded as a National Science Foundation Science and Technology Center

9th DIMACS Implementation Challenge - Shortest Paths

This project requires you to implement Dijkstra's algorithm based on a min-priority queue, such as a *Fibonacci heap*. The goal of the project is to find the best data structure for Dijkstra's algorithm.

Detailed requirements can be downloaded from

https://pintia.cn/

Reference:

Data Structure and Algorithm Analysis in C (2nd Edition)：Ch.5， p.170-180； Ch.11， p.430-435； ***M.A.Weiss 著、陈越改编，人民邮件出版社， 2005***

Introduction to Algorithms, 3rd Edition: Ch.19, p. 505-530； ***Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. The MIT Press. 2009***