

Binomial Queue

二项队列

priority queue: heap insert $O(\log N)$ \rightarrow 均摊 const
built $O(N)$ \rightarrow

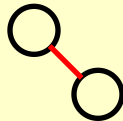
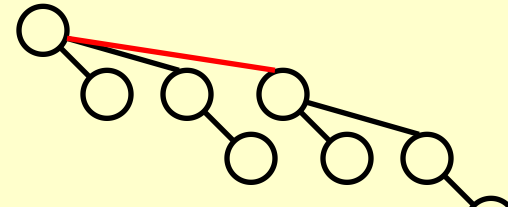
二项树的组合

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**. **Constant!**

What is the cost? So $O(\log N)$ for an insertion is **NOT** good enough!

A binomial tree of height **0** is a one-node tree. A binomial tree of height **k** is formed by attaching a binomial tree, B_{k-1} , to the root of another binomial tree, B_{k-1} .

 B_0  B_1  B_2  B_3 

Binomial
coefficient

Observation: B_k consists of a root with k children, which are B_0, B_1, \dots, B_{k-1} . B_k has exactly 2^k nodes. The number of nodes at depth d is $\binom{k}{d}$.

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

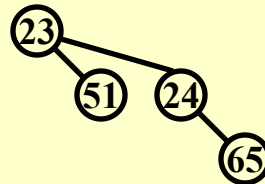
B_0

⑬

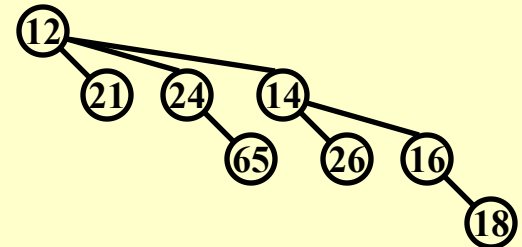
B_1



B_2



B_3

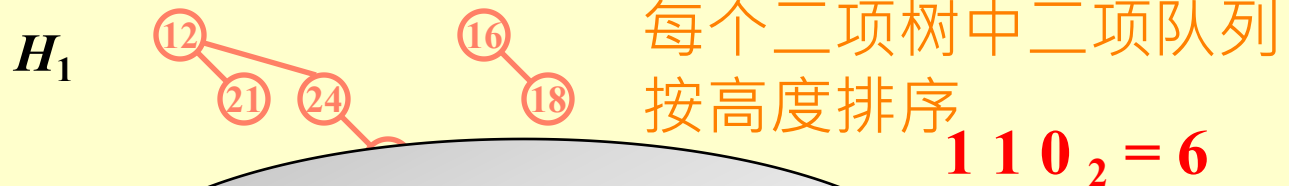


Operations:

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

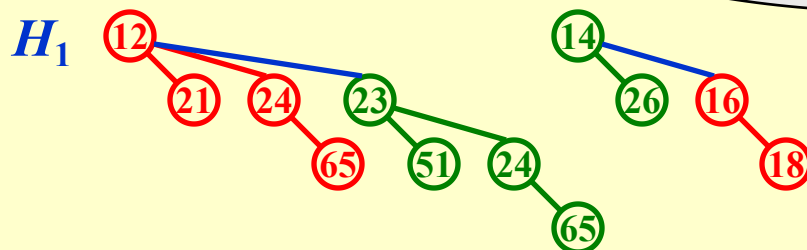
Note: We can remember the minimum and update whenever it is changed. Then this operation will take $O(1)$.

⊕ Merge:



H_2

Must keep the trees in the binomial queue
sorted by height.

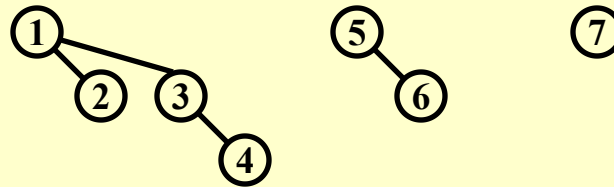


$1\ 1\ 0\ 1$

$$T_p = O(\log N)$$

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

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



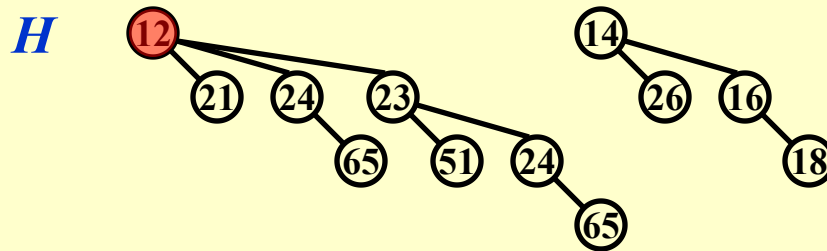
Note:

If the smallest nonexistent binomial tree is B_i , then

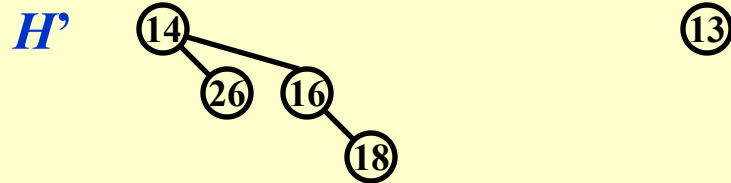
$$T_p = \text{Const} \cdot (i + 1).$$

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

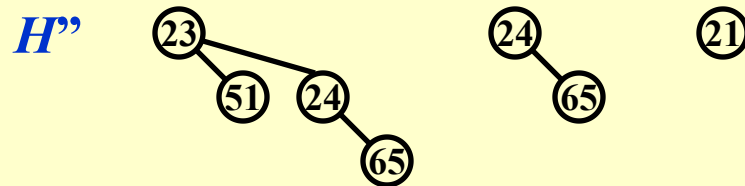
⊕ **DeleteMin** (H):



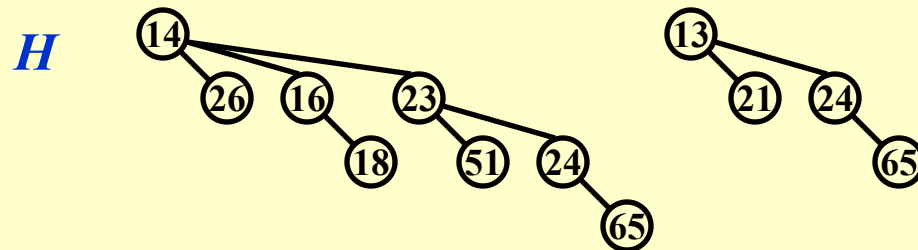
Step 1: **FindMin** in B_k
 $/* O(\log N) */$



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



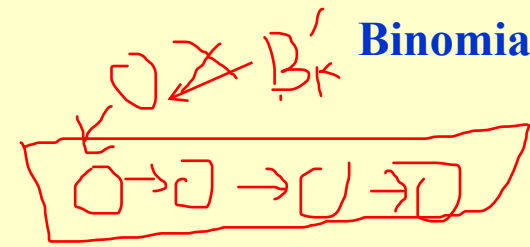
Step 3: Remove root from B_k
 $/* O(\log N) */$



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

Implementation:

Binomial queue = **array** of binomial trees



| Operation | Property | Solution |
|-----------|--|--|
| DeleteMin | Find all the subtrees quickly | 左儿子右兄弟 Left-child-next-sibling with linked lists |
| Merge | The children are ordered by their 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?

从大到小 降序维护子树
方便更新左儿子，否则合并时还要通过右兄弟找到
size最大的子树

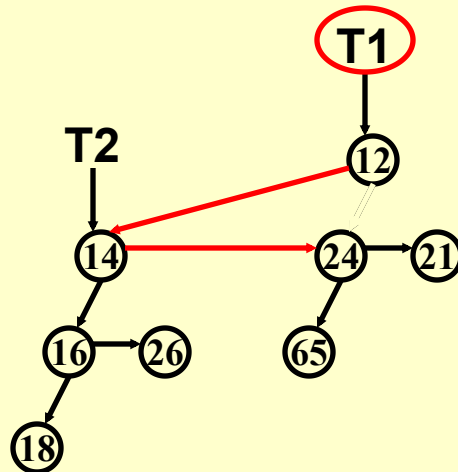
```
typedef struct BinNode *Position;  
typedef struct Collection *BinQueue;  
typedef struct BinNode *BinTree; /* missing from p.176 */
```

```
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;****}**

$$T_p = O(1)$$



```
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 ) {
```

j判断容量是否超过

```
    T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
```

```
    switch( 4*!!Carry + 2*!!T2 + !!T1 ) { /* assign each digit to a tree */
```

```
      case 0: /* 000 */
```

```
      case 1: /* 001 */ break;
```

进位

| | | |
|-------|----|----|
| Carry | T2 | T1 |
|-------|----|----|

```
      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;
```

判断三棵树分别存在否

```
    } /* end switch */
```

```
  } /* end for-loop */
```

```
  return H1;
```

```
}
```

谁和谁合并 合并后放哪 进位还是

NULL转
Bool

```

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]->Element; MinTree = i; /* end if */
        } /* end for-i-loop */
    }
    DeletedTree = H->TheTrees[MinTree];
    H->TheTrees[MinTree] = NULL; /* Step 3.1: delete H' */
    OldRoot = DeletedTree; /* Step 3.1.1: free OldRoot */
    DeletedTree = DeletedTree->LeftChild; free(OldRoot);
    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. 每次最差是 $O(\log N)$

Proof 1 (Aggregate): 聚合

+1 B_0 $/*step = 1 */$ 连接操作 Total steps = N

+0 B_1 $/*step = 1, link = 1 */$

+1 $B_1 \ B_0$ $/*step = 1 */$

-1 B_2 $/*step = 1, link = 2 */$

+1 $B_2 \ B_0$ $/*step = 1 */$

$B_2 \ B_1$ $/*step = 1, link = 1 */$

$B_2 \ B_1 \ B_0$ $/*step = 1 */$

-2 B_3 $/*step = 1, link = 3 */$

$B_3 \ B_0$ $/*step = 1 */$

... ..

Total links =

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

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

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 i th insertion

$\Phi_i ::=$ number of trees *after* the i th insertion ($\Phi_0 = 0$)

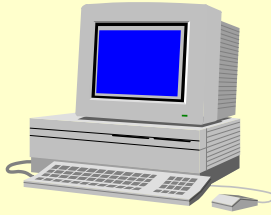
$$C_i + (\Phi_i - \Phi_{i-1}) = 2 \quad \text{for all } i = 1, 2, \dots, N$$

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

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

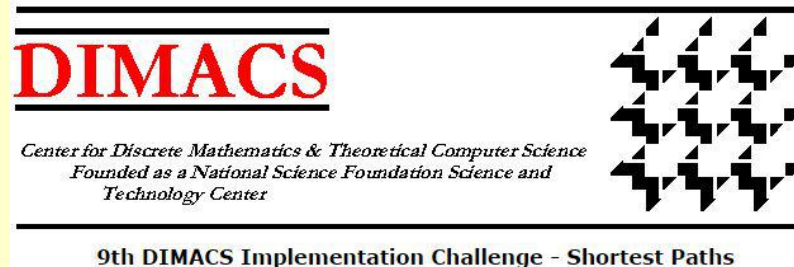


$$T_{\text{worst}} = O(\log N), \text{ but } T_{\text{amortized}} = 2$$



Research Project 1

Shortest Path Algorithm with Heaps (26)



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, 人民邮电出版社, 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*