

# Priority Queue

*Stack & Queue* 是优先级队列的一种特例，优先级完全取决于插入顺序

PQ每次只关心优先级最高的那个元素，所以只要维护偏序关系即可，而不用维护全序关系

- 尝试使用线性结构

```
//use vector
getMax() // traverse, O(n)
delMax() // traverse and delete , O(n) + O(n-r), (r means the rank of Max)
insert() // push_back() , O(1)
```

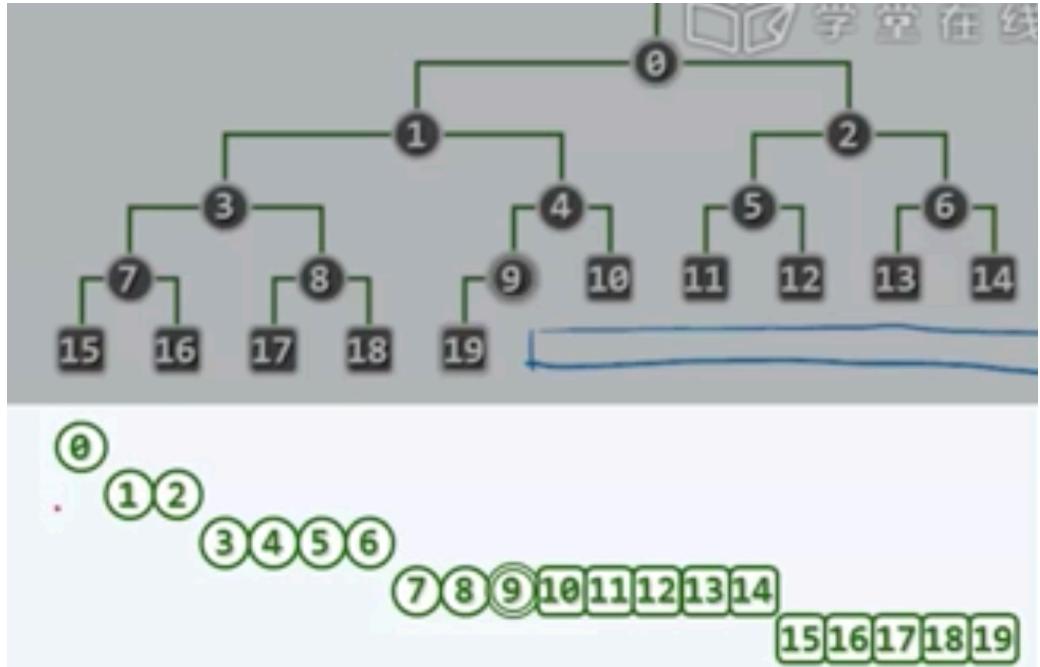
对优先级最高的元素操作效率过低

- 尝试使用半线性结构

```
//use BBST(Balanced Binary Search Tree)
getMax() // O(1)
delMax() , insert() //O(logN)
```

但是BBST对于PQ所需要的操作过于复杂强大，没有必要

- Combine the advantages
  - 逻辑上使用完全二叉树(完全二叉堆)
  - 物理上使用vector
  - 顺序同层次遍历顺序
- 有下标之间有如下关系
  - $0(2^{1-1} - 1)$ 一定是根节点第1层,  $1(2^{2-1} - 1)$ 、 $2$ 一定是第2层,  $3(2^{3-1} - 1)$ 、 $4$ 、 $5$ 、 $6$ 一定是第3层.....
  - 寻找父亲: 对齐(-1)后回溯(/2):  
 $((2^{n+1} - 1) - 1)/2 = 2^n - 1$ (偶数时因为整除的性质被视作奇数)
  - 寻找孩子: 下溯(\*2)后对齐(左孩子+1, 右孩子+2)
  - 寻找父亲相当于在层压缩所以-1, 寻找孩子时相当于层在膨胀所以+1



```
#define Parent(i) ( (i-1)/2 )
#define Lchild(i) ( i*2 + 1 )
#define Rchild(i) ( i*2 + 2 )
```

- 从类的定义也可以看出combine的思想

```
template <typename T>
class PQ_Binary_Heap : public PQ , public
vector<T> {

};
```

- 根据PQ的定义，父亲的优先级必然 $\geq$ 他的两个孩子，所以根元素即为Max

```
template <typename T>
T PQ_Binary_Heap<T>::getMax(){ return at(0) ;
}
```

- 插入时插入在末端，不断上滤以恢复树的优先级排序

```
template <typename T>
void PQ_Binary_Heap<T>::insert(T element ){
    push_back(element);
    percolate(size() - 1);
}

template <typename T>
```

```

void PQ_Binary_Heap<T>::percolateup(size_t
rank) {
    while (Parent(rank)) {
        size_t p = Parent(rank);
        if (at(rank) <= at(p)) break;
        std::swap(at(rank), at(p));
        rank = p;
    }
}

// O(logN) the height of the tree

```

- 删除时删掉堆顶后用末元素填补以保持结构性，再不断percolatedown以恢复堆序性

```

template <typename T>
void PQ_Binary_Heap<T>::delMax() {
    at(0) = back();
    pop_back();
    percolaterdown(0);
}

template <typename T>
void PQ_Binary_Heap<T>::percolaterdown(size_t
rank) {
    size_t largerRank = 0;
    while(Lchild(rank) < size()) {
        if (Rchild(rank) < size()) {

```

```

        largerRank = at(Lchild(rank)) >
at(Rchild(rank)) ? Lchild(rank) :
Rchild(rank);
    }
else{
    largerRank = Lchild;
}
size_t largestrank = at(rank) >
at(largerRank) ? rank : largestrank ;
if (largerRank == rank) break ;
std::swap(at(rank) , at(largerRank));
rank = largerRank;
}
}

```

- 批量建堆(heapification)
  - 蛮力算法：自上而下的percolateup

```

for (int i = 0 ; i < size() ; i++){
    percolateup(i);
}

```

$$T = \sum_{i=0}^{n-1} \text{depth}(\text{node}_i) = O(n \log n)$$

$n$ 个节点，每个节点上滤所需时间约为 $\log n$

- Floyd算法：自下而上的percolatedown(从最后一个内部节点开始)

```
for(int i = Lastinternal(n) ; i >= 0 ; i--){  
    percolatedown(i);  
}
```

$$T = \sum_{i=\text{lastInternalRank}}^0 \text{height}(\text{node}_i) = O(n), \text{ 证明请上网查阅}$$

- 比较：

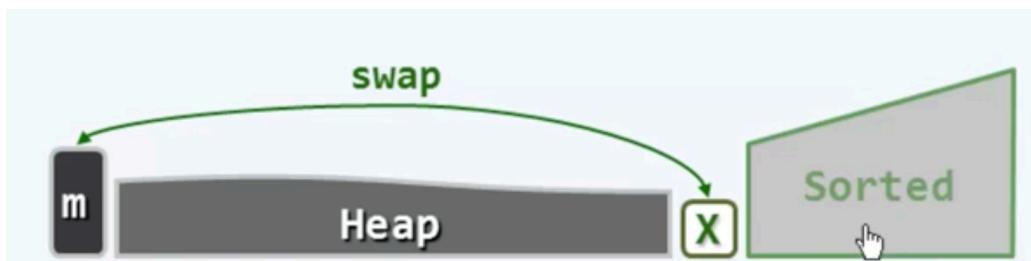
- 在一棵树中犹如社会收入金字塔，高收入人群少（树的顶部节点少），中低收入人群多（树的中底部节点多）
- 蛮力算法自上而下percolateup，使得占少数的节点开销少，占大部分的节点开销大，就如同对少数富人少征税，对多数穷人多征税。
- Floyd算法自下而上percolatedown，使得占多数的节点开销少，占少数的节点开销大，对富人多征税，对穷人少征税。

- 蛮力使用了不合理的复杂度分配策略所以导致了低效 ( $O(n \log n)$  可以完成快排获得全序序列了，而PQ中只需维持偏序关系，故而认为蛮力算法低效)

## Heap Sort(堆排序)

$$T(n) = O(n) + O(n \log n)$$

- flody建堆算法  $O(n)$ , 每次取出堆顶元素  $O(1)$ , 恢复堆序性  $O(\log n)$ , 共要取  $n$  次
- 空间复杂度也有优势, 就地解决, 仅需  $S(n)$



```
void HeapSort(vector<T>::iterator start ,
vector<T>::iterator end , vector<T> & v){
    heapify(start , end , v);
    while(v.size() != 0){
        v._head[end--] = v.popMax();
    }
}
```

```
//STL solution
//has nothing to do with heapsort
```

```
#include <iostream>
#include <algorithm> //std::make_heap,
std::pop_heap, std::push_heap, std::sort_heap
#include <vector>
using std::cout ;
using std::endl;
using std::vector;
using std::make_heap;
using std::pop_heap;
using std::push_heap;
using std::sort_heap;

template <typename T>
void printVector(const vector<T> & v){
    for (auto bg = v.begin() ; bg != v.end() ;
bg++){
        cout << " [ " << *bg << " ]" ;
    }
    cout << endl;
}

int main(){
    int array [10] =
{345,34,56,546,6,547,23,1,2,3};
    vector<int> max_heap (std::begin(array) ,
std::end(array));
```

```
vector<int> min_heap (std::begin(array) ,  
std::end(array));  
cout << "orginal vector :";  
printVector(max_heap);  
cout << "max heap \t:";  
make_heap(max_heap.begin() ,  
max_heap.end());  
printVector(max_heap);  
make_heap(min_heap.begin() , min_heap.end())  
, [](int a , int b)->bool { return a > b;});  
cout << "min heap \t:";  
printVector(min_heap);  
  
cout << "-----" << endl;  
cout << "max heap \t:";  
pop_heap(max_heap.begin() ,  
max_heap.end());  
printVector(max_heap);  
cout << "getMax : " << max_heap.back()<<  
, the max_heap.pop_back()" << endl;  
max_heap.pop_back();  
cout << "-----" << endl;  
cout << "max heap \t:";  
printVector(max_heap);  
cout << "max_heap.push_back(999)\n";  
max_heap.push_back(999);  
cout << "max heap \t:";
```

```

    printVector(max_heap);
    push_heap(max_heap.begin() ,
max_heap.end());
    cout << "max heap \t:";
    printVector(max_heap);

    cout << "-----" << endl;

    cout << "after sorting" << endl;
    cout << "max heap \t:";
    sort_heap(max_heap.begin() ,
max_heap.end());
    printVector(max_heap);
    sort_heap(min_heap.begin() , min_heap.end())
, [](int a , int b)->bool { return a > b;});
    cout << "min heap \t:";
    printVector(min_heap);

    return 0;
}

```

see more at [cppreference](#) and [stackoverflow](#)

```

#include <stdlib.h>

int heapsort(void *base, size_t nmemb, size_t
size, int (compar)(const void *, const void
*));

```

```
/*
base
Points to initial element of array to be
sorted.

nmemb
Is the number of objects in the array.

size
Is the size of each object in memory.

compar
Is the sorting function to use.

*/
```

## 优先队列的其他实现方式（堆的变种）

### 左式堆

便于堆合并操作

- 依次将B中元素插入A中，  $T(m, n) = O(m \log(n))$
- 直接将两堆相接，然后Flody建堆 $O(n + m)$ ,但是该方法并没有利用原来已经给出的A和B中的偏序关系的信息

息，所以一定可以更快

## 左式堆(leftist heap)

- 保持堆序性
  - 使得在合并时，只需调整很少的节点
  - 所有节点靠左，合并时直接向右（偏序关系的再一次体现，只要求父亲严格大于等于孩子，而不在乎兄弟之间的关系）
- 破坏了结构性
  - 但是堆序性才是堆结构的本质要求
  - 结构性必要时可以舍弃
  - 由于左式堆结构性已经破坏，元素排列不在是紧凑的，故继续使用 **vector** 作为物理基础则会导致空间浪费,改用灵活的树形结构加以实现

## NPL(Null Path Length)

引入假想的外部节点，消除1度的节点，转化为真二叉树

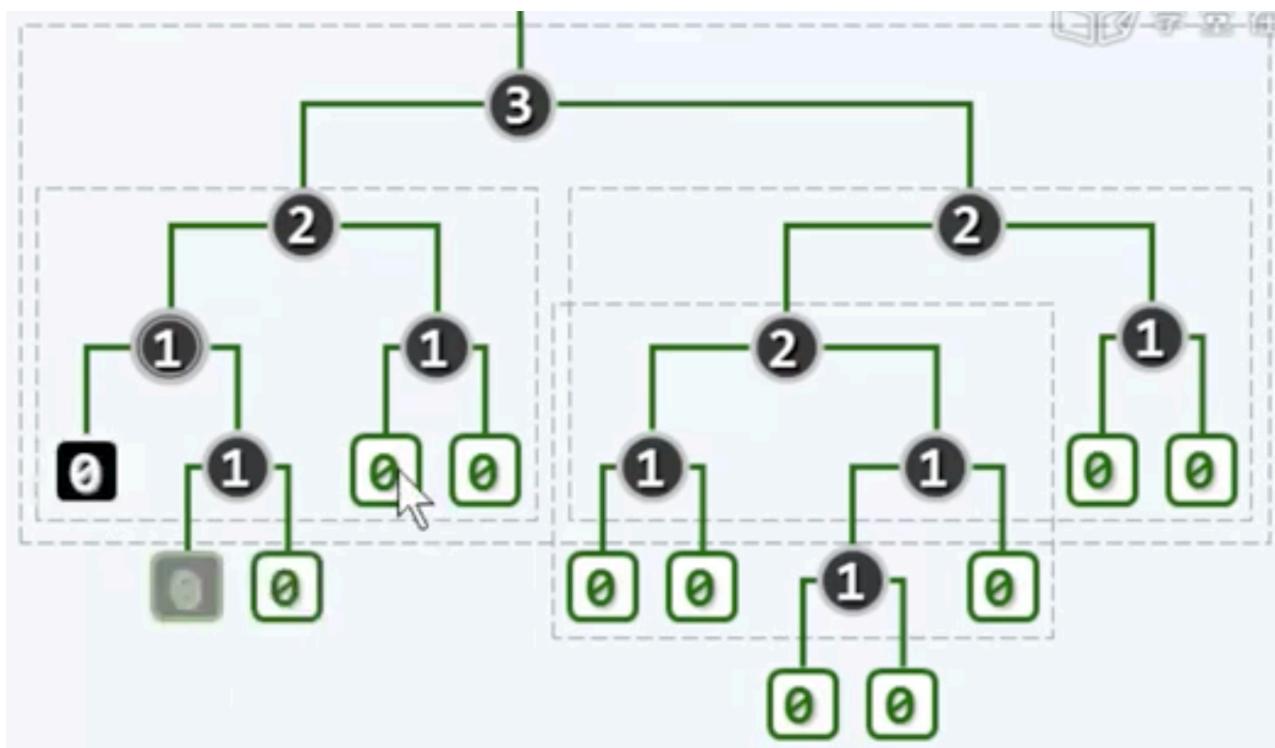
NPL是用来度量堆的倾斜性的

```

typedef struct node{...} Node;
int NPL(Node n){
    if (n == NULL) return 0;
    else return (1 + min(NPL(n.left) ,
NPL(n.right)));
}

```

$NPL(n)$  也是节点  $n$  到达叶节点的最短距离



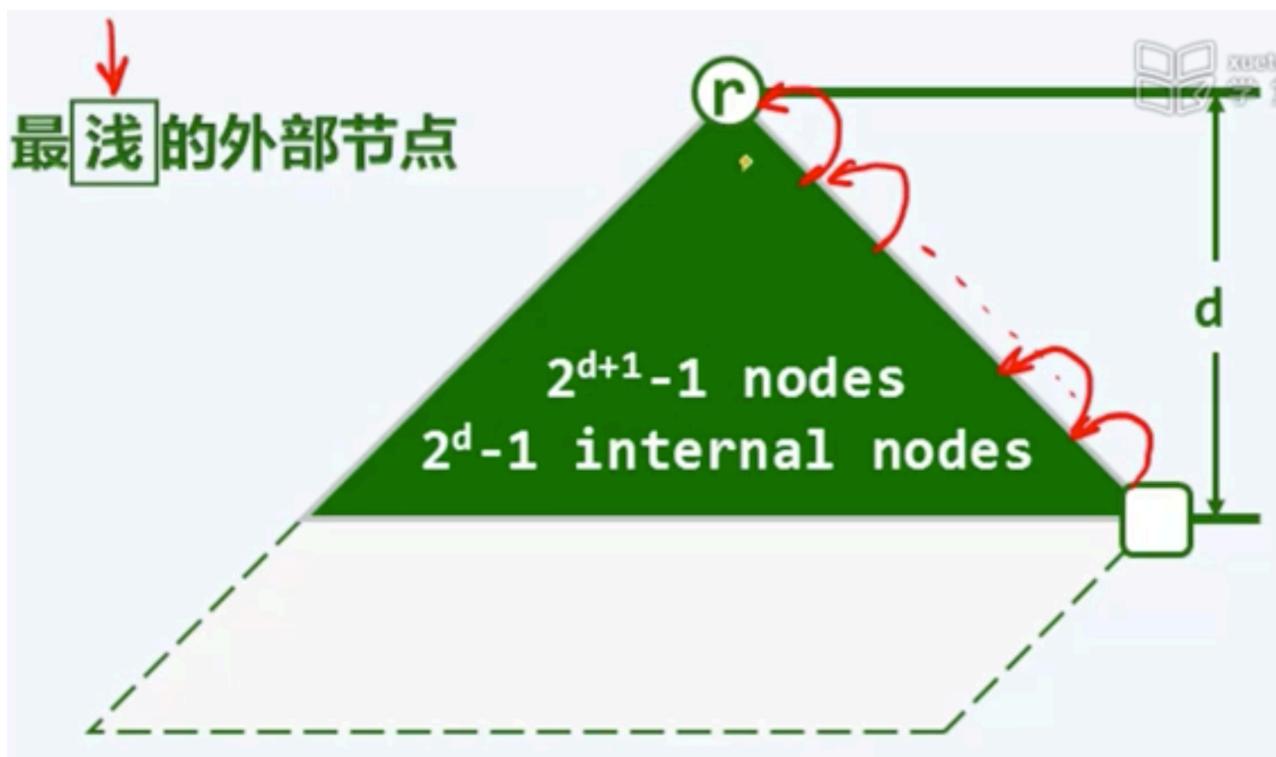
## 左倾性

- 对于堆内的每一个节点，都有  $NPL(\text{node-} \rightarrow \text{left}) \geq NPL(\text{node-} \rightarrow \text{right})$
- 由定义自然有  $NPL(\text{node}) = NPL(\text{node-} \rightarrow \text{right}) + 1$
- 左式堆只是更倾向将节点分布在左侧，并非完全分布

在左侧

## 右侧链

- 假设根节点  $NPL(r) = d$ , 则有以  $r$  为根形成的极大满子树
  - 它共有节点  $2^{d+1} - 1 = n$
  - 有内部节点  $2^d - 1$



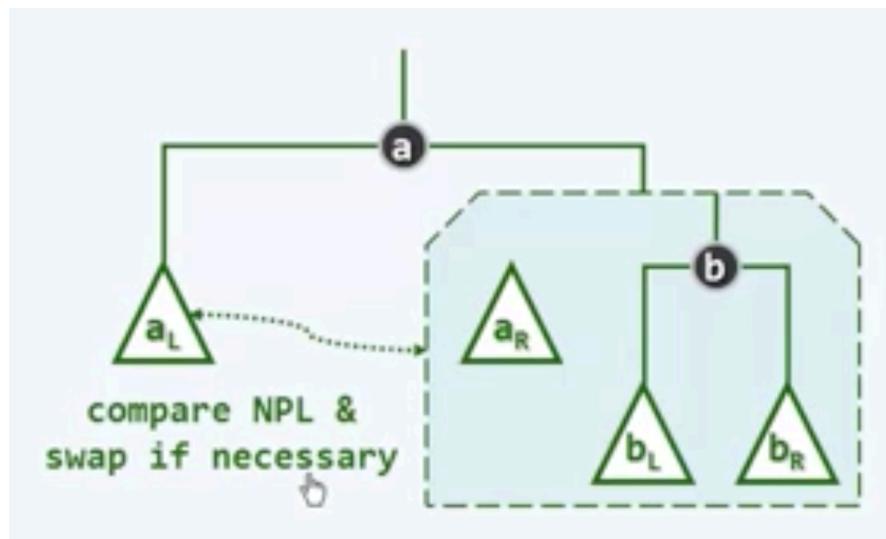
- 此时若只沿着右侧链操作的话，右侧链上共有节点  $d + 1$  个， $d \leq \log_2(n + 1) - 1$ , 仅有  $\log n$  的复杂度

## 实现

```

template <typename T>
class LeftistHeap : public PQ<T> , public
BinaryTree<T>{
    .....
    static LeftistHeap* merge(LeftistHeap * a ,
LeftistHeap * b);
}

```



```

template <typename T>
LeftistHeap* LeftistHeap<T>::merge(LeftistHeap
* a , LeftistHeap * b){
    //recursive base
    if (a == nullptr) return b;
    if (b == nullptr) return a;

    //maintain the property of heap
    if (a->root->data < b->root->data)
swap(a,b);
}

```

```

    //merge right subheap of a with b
    a->root->right = merge(a->root->right , b-
>root);

    //maintain the property of leftisheap
    if (a->root->left == nullptr || a->root-
>left->npl < a->root->right->npl) swap(a->root-
>left , a->root->right);
    a->npl = a->right->npl==0 ? 1 : a->right-
>npl + 1 ;

    //return merged leftisheap
    return a;
}

```

因为算法只在右侧链上操作，所以具有 $O(\log(n))$ 的复杂度

此后，插入和删除（最大/小元）的操作均可基于merge实现

- 插入：将待插入的节点视作仅有一个人数的左式堆，合并两堆
- 删除：摘除根节点，合并左右子堆

