ST 表、堆（左偏树）、LCA

徐宇飞

BIT ACM

2023 年 7 月 25 日

ST 表是基于倍增思想、处理区间查询的一种数据结构。

ST 表是基于倍增思想、处理区间查询的一种数据结构。
由于 ST 表的查询过程中，会用两个部分重叠的区间计算整个区间，所以要求处理的问题是可重复贡献的问题。

ST 表是基于倍增思想、处理区间查询的一种数据结构。
由于 ST 表的查询过程中，会用两个部分重叠的区间计算整个区间，所以要求处理的问题是可重复贡献的问题。

可重复贡献问题

可重复贡献问题是对于运算 opt，运算的性质满足 $x opt x = x$，则对应的区间询问就是一个可重复的贡献问题，例如：最大值满足 $max(x, x) = x, min(x, x) = x$，最大公因数满足 $gcd(x, x) = x$，因此 RMQ 问题和 GCD 问题就是一个可重复贡献的问题

ST 表的具体实现

以区间最小值为例
先预处理一个二维数组 f[][]，其中 f[i][j] 表示 $[i, i+2^j-1]$ 这个区间中的最小值，即从第 i 个元素开始的连续 2^j 个元素中的最小值。
f[i][j] 有递推关系式

$$f[i][j] = min(f[i][j-1], f[i+(1<<(j-1))][j-1])$$

ST 表的具体实现

预处理过程

```
int a[N]; \\要查询的数组
int f[N][D]; \\ST表预处理的数组

for(int i = 1; i < N; i++) f[i][0] = a[i];
for(int j = 1; j < D; j++) {
    for(int i = 1; i < N; i++) {
        f[i][j] = min(f[i][j-1], f[i+(1<<(j-1))][j-1]);
    }
}
```

ST 表的具体实现

预处理过程

```
int a[N]; \\要查询的数组
int f[N][D]; \\ST表预处理的数组

for(int i = 1; i < N; i++) f[i][0] = a[i];
for(int j = 1; j < D; j++) {
    for(int i = 1; i < N; i++) {
        f[i][j] = min(f[i][j-1], f[i+(1<<(j-1))][j-1]);
    }
}
```

由于 D 只需要取到 $\lceil \log N \rceil$，因此复杂度为 $O(n \log n)$

ST 表的具体实现

使用 ST 表查询任意区间 $[l, r]$ 的最小值
令 L 为使 $2^L <= r - l + 1$ 的最小整数，则 $[l, r]$ 可以被拆成两个部分重叠的区间 $[l, l + 2^L - 1]$ 和 $[r - 2^L + 1, r]$ 这两个部分重叠的区间，这两个区间的最小值分别是 $f[l][L]$ 和 $f[r - 2^L + 1][L]$，这两个值取最小就是 $[l, r]$ 这个区间的最小值。

ST 表的具体实现

使用 ST 表查询任意区间 $[l, r]$ 的最小值
令 L 为使 $2^L <= r - l + 1$ 的最小整数，则 $[l, r]$ 可以被拆成两个部分重叠的区间 $[l, l + 2^L - 1]$ 和 $[r - 2^L + 1, r]$ 这两个部分重叠的区间，这两个区间的最小值分别是 $f[l][L]$ 和 $f[r - 2^L + 1][L]$，这两个值取最小就是 $[l, r]$ 这个区间的最小值。

查询过程

```
int query(int l, int r) {
    int L = 0;
    while((1<<(L+1)) <= r - l + 1) L++;
    return min(f[l][L], f[r-(1<<L)+1][L]);
}
```

堆是一种数据结构，在 STL 中有封装好的堆——优先队列。
优先队列可以高效的插入元素、查询最大元素、删除最大元素。

堆是一种数据结构，在 STL 中有封装好的堆——优先队列。
优先队列可以高效的插入元素、查询最大元素、删除最大元素。

```
#include <queue>
using namespace std;

priority_queue<int> pq; //默认是大根堆
                        //即堆顶元素是最大的元素

pq.push(x); //插入元素
pq.top(); //堆顶元素
pq.pop(); //删除堆顶元素
pq.empty(); //判断是否非空
```

STL 中的优先队列是用完全二叉堆实现的。
左偏树也可以用来实现堆的功能，除了上述的功能，它还可以高效地实现两个堆的合并（$O(logn)$）。

STL 中的优先队列是用完全二叉堆实现的。
左偏树也可以用来实现堆的功能，除了上述的功能，它还可以高效地实现两个堆的合并（$O(logn)$）。
左偏树中的每个节点要存储以下信息：

- 该节点的权值 val
- 父节点的编号 fa
- 左右儿子节点的编号 ls，rs
- 该节点的距离 dist

节点的距离

某个节点被称为外节点，仅当这个节点的左子树或右子树为空。某一个节点的距离即该节点到与其最近的外节点经过的边数。易得，外节点的距离为 0，空节点距离为 -1。

左偏树需要满足的性质

1. 堆的性质，即对于任意节点 p，$val(p) \le val(ls), val(rs)$。（假设定义的是小根堆）

左偏树需要满足的性质

1. 堆的性质，即对于任意节点 p, $val(p) \leq val(ls), val(rs)$。（假设定义的是小根堆）
2. $dist(ls) \geq dist(rs)$，即左子树距离一定大于等于右子树距离，这也是左偏树这一名字的由来。

左偏树需要满足的性质

1. 堆的性质，即对于任意节点 p, $val(p) \leq val(ls), val(rs)$。（假设定义的是小根堆）
2. $dist(ls) \geq dist(rs)$，即左子树距离一定大于等于右子树距离，这也是左偏树这一名字的由来。
3. $dist(x) = dist(rs) + 1$，也就是说，任意节点的距离等于其右儿子的距离 $+1$。

左偏树核心操作：合并

左偏树的合并是一个递推操作。
设 x，y 是两个要合并的左偏树的根节点编号。不妨设 x 的权值较小，否则可以交换 x 和 y。

- 我们将 x 作为被插入树，y 作为插入树，每次合并就是先把 y 和 x 的右儿子合并, 再将合并后的新树的根节点作为 x 的右儿子。item 边界条件是当 x 和 y 代表的树中有一棵为空时，直接返回另一棵非空的树。

左偏树核心操作：合并

左偏树的合并是一个递推操作。
设 x，y 是两个要合并的左偏树的根节点编号。不妨设 x 的权值较小，否则可以交换 x 和 y。

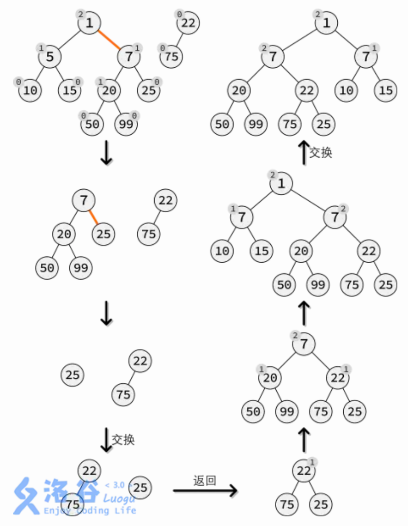
- 我们将 x 作为被插入树，y 作为插入树，每次合并就是先把 y 和 x 的右儿子合并, 再将合并后的新树的根节点作为 x 的右儿子。item 边界条件是当 x 和 y 代表的树中有一棵为空时，直接返回另一棵非空的树。
- 因此如果 x 的右儿子为空时，相当于直接将 y 作为 x 的右儿子。

左偏树核心操作：合并

左偏树的合并是一个递推操作。
设 x，y 是两个要合并的左偏树的根节点编号。不妨设 x 的权值较小，否则可以交换 x 和 y。

- 我们将 x 作为被插入树，y 作为插入树，每次合并就是先把 y 和 x 的右儿子合并, 再将合并后的新树的根节点作为 x 的右儿子。item 边界条件是当 x 和 y 代表的树中有一棵为空时，直接返回另一棵非空的树。
- 因此如果 x 的右儿子为空时，相当于直接将 y 作为 x 的右儿子。
- 特别要注意的是，在每次插入完成后，还需要检查左右儿子的距离是否满足左偏树的要求，如果不满足，还需要交换左右儿子。

左偏树核心操作：合并



合并操作

```
node t[N]; //节点数组
int Merge(int x,int y)
{
    if(!x||!y)return x+y;
    if(t[x].val>t[y].val||(t[x].val==t[y].val&&x>y))
        swap(x,y);
    int &ul=t[x].ls,&ur=t[x].rs;
    ur=Merge(ur,y);
    t[ur].fa=x;
    if(t[ul].dist<t[ur].dist)swap(ul,ur);
    t[x].dist=t[ur].dist+1;
    return x;
}
```

左偏树的其他操作都可以用合并操作来间接实现。

左偏树的其他操作都可以用合并操作来间接实现。

插入节点

单个节点也可以看成一棵左偏树，直接将节点和原树合并即可。

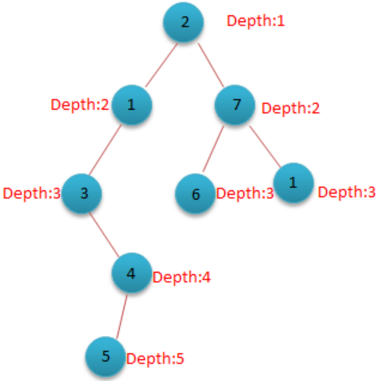
删除根节点

将根节点的权值赋为 -1，然后合并左右子树，维护相关信息即可。

```
	inline void Erase(int x)
	{
		int ul=t[x].ls,ur=t[x].rs;
		t[x].val=-1;t[ul].fa=0;t[ur].fa=0;
		merge(ul,ur);
	}
```

最近公共祖先（LCA）。LCA(u, v) 指的是 u、v 的公共祖先中深度最深的祖先。

最近公共祖先（LCA）。LCA(u, v) 指的是 u、v 的公共祖先中深度最深的祖先。



u，v 两个节点有两种情况：

1. u，v 两个节点深度相同
2. u，v 两个节点深度不同

u，v 两个节点有两种情况：

1. u，v 两个节点深度相同
2. u，v 两个节点深度不同

第二种情况我们可以转化成第一种情况：不妨设 u 的深度比 v 更大，我们可以让 u 一直向它的父节点走，直到走到与 v 深度相同的 u'。容易发现 $lca(u,v) = lca(u',v)$。

u，v 两个节点有两种情况：

1. u，v 两个节点深度相同
2. u，v 两个节点深度不同

第二种情况我们可以转化成第一种情况：不妨设 u 的深度比 v 更大，我们可以让 u 一直向它的父节点走，直到走到与 v 深度相同的 u'。容易发现 $lca(u, v) = lca(u', v)$。
对于第一种情况，我们考虑一种暴力的做法，我们将 u，v 同时向上走，直到 u，v 走到一个相同的节点。那么这个节点就是 u，v 的 lca。

上述暴力的方法已经是正确的，但我们可以用树上倍增的思想将复杂度优化到 $O(logn)$。

上述暴力的方法已经是正确的，但我们可以用树上倍增的思想将复杂度优化到 $O(logn)$。
发现我们每次都是向上走一步。假设我们总共要向上走 k 步，那么我们第一次可以走 2^i 步，其中 2^i 是小于等于 k 的最小 2 的整数幂。这样最多只要走 $logk$ 次。

于是，我们需要先预处理出每个节点向上走 2^i 步后会到达那个节点以及每个节点的深度。
令 f[i][j] 代表第 i 个节点向上走 2^j 步后到达的节点，有关系式：

$$f[i][j] = f[f[i][j-1]][j-1]$$

于是，我们需要先预处理出每个节点向上走 2^i 步后会到达那个节点以及每个节点的深度。
令 f[i][j] 代表第 i 个节点向上走 2^j 步后到达的节点，有关系式：

$$f[i][j] = f[f[i][j-1]][j-1]$$

预处理过程

```
int dep[N];
int f[N][D];
void init(int u, int fa) {
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for(int d = 1; d < D; d++) f[u][d] = f[f[u][d-1]][d
        -1];

    for(int v: v是u的子节点) init(v, u);
}
```

然后我们就可以用上述过程求 LCA 了。

求 LCA

```cpp
int lca(int u, int v) {
    if(dep[u] < dep[v]) swap(u, v);

    //将第二种情况先转化为第一种情况
    for(int d = D - 1; d >= 0; d--) {
        if(dep[u] - (1<<d) >= dep[v]) u = f[u][d];
    }
    //处理第二种情况
    if(u == v) return u;
    for(int d = D - 1; d >= 0; d--) {
        if(f[u][d] != f[v][d]) {
            u = f[u][d];
            v = f[v][d];
        }
    }
    return f[u][0];
}
```