数据结构

- 树状数组
  - 树上数组上二分
- 线段树
  - 可持久化线段树
  - 线段树合并/分裂
- 字典树
- 树链剖分
- 稀疏表
- Link Cut Tree
- 珂朵莉树
- 虚树

树状数组

树状数组是最为小巧实用的数据结构之一，能在$\mathcal{O}(\log n)$的时间复杂度内进行单点修改+区间查询。通过维护差分数组也可以实现区间修改+单点查询。

树状数组通过前缀和相减来完成区间操作，所以要求维护的信息具有可减性，否则无法使用树状数组维护。

```
template<typename T=int> struct Fenwick {
    int size=0;
    vector<T> tr;

    int lowbit(int x) { return x&(-x); }

    void update(T &aim,const T &val) { aim+=val; }

    void add(int pos,T val) {
        while(pos<=size) update(tr[pos],val),pos+=lowbit(pos);
    }

    T query(int pos) {
        T res{};
        while(pos) update(res,tr[pos]),pos-=lowbit(pos);
        return res;
    }

    Fenwick(int size):size(size) { tr.resize(size+1); }
};
```

树上数组上二分

类似线段树，我们可以在树上数组上进行二分，从高位向低位枚举即可。

权值树状数组求第k大的例子：

```
int kth(int k){
    int pos=0;
    for(int i=bit;~i;i--)
        if(pos+(1<<i)<N&&tr[pos+(1<<i)]<k)
            pos+=1<<i,k-=tr[pos];
    return pos+1-M;
}
```

线段树

线段树能够灵活地维护区间信息，区间修改与查询均为$\mathcal{O}(\log n)$，常数较大。

```
struct SegmentTree {
    #define lch (u<<1)
    #define rch (u<<1|1)
    constexpr static int MAXSIZE=N;

    struct Node {
        int l,r;

    } tr[MAXSIZE<<2];

    void pushup(int u) {

    }

    void pushdn(int u) {

    }

    void modify(int u,int l,int r,int val) {
        if(tr[u].l>=l&&tr[u].r<=r) {}
        else {
            pushdn(u);
            int mid=tr[u].l+tr[u].r>>1;
            if(mid>=l) modify(lch, l, r, val);
            if(mid<r) modify(rch, l, r, val);
            pushup(u);
        }
    }

    int query(int u,int l,int r) {
        if(tr[u].l>=l&&tr[u].r<=r) {}
        else {
            pushdn(u);
            int mid=tr[u].l+tr[u].r>>1;

            if(mid>=l) query(lch, l, r);
```

```
            if(mid<r) query(rch, l, r);
        }
    }

    void build(int u,int l,int r) {
        tr[u]={l,r};
        if(l==r) {}
        else {
            int mid=l+r>>1;
            build(lch, l, mid);
            build(rch, mid+1, r);
            pushup(u);
        }
    }

    #undef lch
    #undef rch
} sgt;
```

可持久化线段树

通过记录每次修改变化的节点，可以在保存历史信息的同时，大幅地压缩空间复杂度。

区间第k大值查询的例子：

```
struct PersistentSegmentTree {

    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    constexpr static int MAX_SIZE=N*20*2;

    struct Node {
        int ch[2];
        int cnt;
    } tr[MAX_SIZE];
    int idx;

    int new_node() {
        // assert(idx<MAX_SIZE);
        return ++idx;
    }

    void pushup(int u) {
        tr[u].cnt=tr[lch].cnt+tr[rch].cnt;
    }

    void modify(int &u,int v,int l,int r,int p) {
        u=new_node();
        tr[u]=tr[v];
        if(l==r) tr[u].cnt++;
        else {
```

```
            int mid=l+r>>1;
            if(p<=mid) modify(lch, tr[v].ch[0], l, mid, p);
            else modify(rch, tr[v].ch[1], mid+1, r, p);
            pushup(u);
        }
    }

    int kth(int u,int v,int l,int r,int k) {
        if(l==r) return l;
        int mid=l+r>>1;
        int lcnt=tr[lch].cnt-tr[tr[v].ch[0]].cnt;
        if(lcnt>=k) return kth(lch, tr[v].ch[0], l, mid, k);
        return kth(rch, tr[v].ch[1], mid+1, r, k-lcnt);
    }

    void build(int &u,int l,int r) {
        u=new_node();
        tr[u]={l,r};
        if(l!=r) {
            int mid=l+r>>1;
            build(lch,l,mid);
            build(rch,mid+1,r);
        }
    }

    #undef lch
    #undef rch

} sgt;
```

线段树合并/分裂

设区间大小为 n ，分裂次数为 m 。无论以何种顺序合并与分裂，时间复杂度均为 $\mathcal{O}((n+m)\log n)$ ，空间复杂度 $\mathcal{O}((n+m)\log n)$ （常数=1），如果在合并时保留子树结构，则需要多一倍的空间。

```
struct MergeSplitSegmentTree {

    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    constexpr static int MAX_SIZE=1e7+10;

    struct Node {
        int ch[2];
        int cnt;
    } tr[MAX_SIZE];
    int idx;

    int new_node() {
        // assert(idx<MAX_SIZE);
        return ++idx;
    }
```

```
    void pushup(int u) {
        if(lch&&rch) ;
        else if(lch) ;
        else if(rch) ;
    }

    // remember to pushdn laze tag
    void pushdn(int u) {
        if(lch) ;
        if(rch) ;

    }

    void merge(int &u,int v) {
        if(!u&&!v) return;
        if(!u||!v) u=u|v;
        else {
            pushdn(u);pushdn(v);
            merge(lch,tr[v].ch[0]);
            merge(rch,tr[v].ch[1]);
            pushup(u);
        }
    }

    // k][k+1
    void split(int &u,int &v,int l,int r,int k) {
        if(!u||k>=r) return;
        if(k<l) swap(u,v);
        else {
            v=new_node();
            int mid=l+r>>1;
            if(k<=mid) swap(rch,tr[v].ch[1]);
            pushdn(u);
            if(k<mid) split(lch, tr[v].ch[0], l, mid, k);
            else split(rch, tr[v].ch[1], mid+1, r, k);
            pushup(u),pushup(v);
        }
    }

    int kth(int u,int l,int r,int k) {
        if(tr[u].cnt<k) return -1;
        if(l==r) return l;
        int mid=l+r>>1;
        pushdn(u);
        if(tr[lch].cnt>=k) return kth(lch, l, mid, k);
        return kth(rch, mid+1, r, k-tr[lch].cnt);
    }

    void build(int &u,int l,int r,int p) {
        u=new_node();
        if(l==r) ;
        else {
            int mid=l+r>>1;
```

```
            if(p<=mid) build(lch,l,mid,p);
            else build(rch,mid+1,r,p);
            pushup(u);
        }
    }

    #undef lch
    #undef rch

} sgt;
```

字典树

```
struct Trie {
    constexpr static int A=26,B='a';
    struct Node {
        int ch[A];
        int cnt;
    };
    vector<Node> tr;

    int new_node() { tr.push_back({}); return tr.size()-1; }

    int extend(int u,int x) {
        if(!tr[u].ch[x-B]) tr[u].ch[x-B]=new_node();
        tr[tr[u].ch[x-B]].cnt++;
        return tr[u].ch[x-B];
    }

    template<typename T> void insert(const T &s) {
        int u=0;
        for(auto x:s) u=extend(u, x);
    }

    void clear() { tr.clear(); new_node(); }
    Trie() { clear(); }
    Trie(int size) { tr.reserve(size); clear(); }
} trie;
```

树链剖分

重链剖分能将树上路径转为$\mathcal{O}(\log n)$级别的连续区间，从而将树上问题转化为区间问题。预处理时间复杂度$\mathcal{O}(n)$,单次路径剖分时间复杂度$\mathcal{O}(\log n)$。

```
namespace hpd {
    using PII=pair<int,int>;
    constexpr int N=1e5+10; // ***
    int id[N],w[N],nw[N],cnt;
    int dep[N],sz[N],top[N],p[N],hch[N];
```

```
    vector<int> adj[N];

    void dfs1(int u,int fa,int d) {
        dep[u]=d,p[u]=fa,sz[u]=1;
        for(int v:adj[u]) {
            if(v==fa) continue;
            dfs1(v,u,d+1);
            sz[u]+=sz[v];
            if(sz[hch[u]]<sz[v]) hch[u]=v;
        }
    }

    void dfs2(int u,int t) {
        id[u]=++cnt,nw[cnt]=w[u],top[u]=t;
        if(!hch[u]) return;
        dfs2(hch[u],t);
        for(int v:adj[u])
            if(v!=p[u]&&v!=hch[u]) dfs2(v,v);
    }

    int lca(int x,int y) {
        while(top[x]!=top[y]) {
            if(dep[top[x]]<dep[top[y]]) swap(x,y);
            x=p[top[x]];
        }
        if(dep[x]<dep[y]) swap(x,y);
        return y;
    }

    vector<PII> decompose(int x,int y) {
        vector<PII> res;
        while(top[x]!=top[y]) {
            if(dep[top[x]]<dep[top[y]]) swap(x,y);
            res.emplace_back(id[top[x]],id[x]);
            x=p[top[x]];
        }
        if(dep[x]<dep[y]) swap(x,y);
        res.emplace_back(id[y],id[x]);
        return res;
    }

    PII decompose(int x) {
        return { id[x],id[x]+sz[x]-1 };
    }

    void init() {
        dfs1(1,-1,1); dfs2(1,1);
    }

    void clear(int n) {
        cnt=0;
        fill(hch, hch+n+1, 0);
    }
}
```

稀疏表

倍增维护区间最大值。

```
template<int MAX_SIZE,typename T=int> struct SparseTable {
    constexpr static int M=__lg(MAX_SIZE);
    T arr[MAX_SIZE],st[M][MAX_SIZE];

    void build(int n) {
        for(int i=1;i<=n;i++) st[0][i]=arr[i];
        for(int k=1,t=1<<k;k<M;k++,t<<=1)
            for(int i=1,j=i+t-1,mid=i+t/2;j<=n;i++,j++,mid++)
                st[k][i]=max(st[k-1][i],st[k-1][mid]);
    }

    T query(int l,int r) {
        if(r<l) return 0;
        int k=__lg(r-l+1);
        return max(st[k][l],st[k][r-(1<<k)+1]);
    }
};
```

Link Cut Tree

LCT用来解决动态树问题，加删边/提取树上路径的复杂度均为$\mathcal{O}(\log n)$，常数巨大。

编号从1开始。

```
struct LinkCutTree {

    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    #define wch(u) (tr[tr[u].p].ch[1]==u)
    constexpr static int MAX_SIZE=1e5+10;

    struct Node {
        int ch[2],p;
        bool rev;

    } tr[MAX_SIZE];
    int stk[MAX_SIZE];

    bool is_root(int u) {
        return tr[tr[u].p].ch[0]!=u&&tr[tr[u].p].ch[1]!=u;
    }

    void pushup(int u) {
```

```
    }

    void pushrev(int u) {
        swap(lch,rch);
        tr[u].rev^=1;
    }

    void pushdn(int u) {
        if(tr[u].rev) pushrev(lch),pushrev(rch),tr[u].rev=0;
    }

    void rotate(int x) {
        int y=tr[x].p,z=tr[y].p,k=wch(x);
        if(!is_root(y)) tr[z].ch[wch(y)]=x;
        tr[y].ch[k]=tr[x].ch[!k],tr[tr[y].ch[k]].p=y;
        tr[x].ch[!k]=y,tr[y].p=x,tr[x].p=z;
        pushup(y),pushup(x);
    }

    void splay(int u) {
        int top=0,fa=u;
        stk[++top]=fa;
        while(!is_root(fa)) stk[++top]=fa=tr[fa].p;
        while(top) pushdn(stk[top--]);
        for(;!is_root(u);rotate(u))
            if(!is_root(fa=tr[u].p)) rotate(wch(u)==wch(fa)?fa:u);
    }

    void access(int u) {
        int t=u;
        for(int v=0;u;v=u,u=tr[u].p)
            splay(u),rch=v,pushup(u);
        splay(t);
    }

    void make_root(int u) {
        access(u);
        pushrev(u);
    }

    int split(int u,int v) {
        make_root(u);
        access(v);
        return v;
    }

    int find_root(int u) {
        access(u);
        while(lch) pushdn(u),u=lch;
        splay(u);
        return u;
    }

    void link(int u,int v) {
```

```
        make_root(u);
        if(find_root(v)!=u) tr[u].p=v;
    }

    void cut(int u,int v) {
        make_root(u);
        if(find_root(v)==u&&rch==v&&!tr[v].ch[0])
            rch=tr[v].p=0,pushup(u);
    }

    void modify(int u,int val) {
        splay(u);

        pushup(u);
    }

    #undef lch
    #undef rch
    #undef wch

} lct;
```

珂朵莉树

珂朵莉树通过暴力地合并set中信息相同的点来压缩时间复杂度，在保证数据随机的前提下，其时间复杂度为$\mathcal{O}(n\log^2 n)$。

```
struct ChthollyTree {
    struct Node {
        int l,r,v;
        Node(int L,int R,int V) : l(L),r(R),v(V) {}
        bool operator< (const Node &x) const {
            return l<x.l;
        }
    };
    set<Node> st;

    auto split(int pos){
        auto it=st.lower_bound(Node(pos,pos,0));
        if(it!=st.end()&&it->l==pos) return it;
        it=prev(it);
        auto [l,r,v]=*it;
        st.erase(it);
        st.insert(Node(l,pos-1,v));
        return st.insert(Node(pos,r,v)).first;
    }

    void assign(int l,int r,int v){
        auto end=split(r+1),begin=split(l);
        st.erase(begin,end);
        st.insert(Node(l,r,v));
```

```
    }
} odt;
```

虚树

能在 $\mathcal{O}(k\log n)$ 时间内提取树上的 k 个关键点建成一棵新树,并且新树的点数不超过 $2k$。

```
namespace vt {
    constexpr int N=1e5+10,M=__lg(N); // ***
    vector<int> vt[N],adj[N];
    int stk[N],top,id[N],idx;
    int fa[N][M+1],dep[N];
    bool key[N];

    void lca_init(int u,int p) {
        dep[u]=dep[p]+1;
        for(int v:adj[u]) {
            if(v==p) continue;
            fa[v][0]=u;
            for(int i=1;i<=M;i++)
                fa[v][i]=fa[fa[v][i-1]][i-1];
            lca_init(v,u);
        }
    }

    int lca(int u,int v) {
        if(dep[u]<dep[v]) swap(u,v);
        for(int k=M;~k;k--)
            if(dep[fa[u][k]]>=dep[v])
                u=fa[u][k];
        if(u==v) return u;
        for(int k=M;~k;k--)
            if(fa[u][k]!=fa[v][k])
                u=fa[u][k],v=fa[v][k];
        return fa[u][0];
    }

    void relabel(int u,int fa) {
        id[u]=++idx;
        for(int v:adj[u]) if(v!=fa) relabel(v, u);
    }

    void build(vector<int> &vec) {
        sort(vec.begin(),vec.end(),[](int x,int y) {
            return id[x]<id[y];
        });

        // TODO clearup dirt memory
        auto clear=[&](int u) {
            vt[u].clear();
            key[u]=0;
```

```
        };

        auto add=[&](int u,int v) {
            vt[u].emplace_back(v);
        };

        clear(1);
        stk[top=0]=1;
        for(int u:vec) {
            if(u==1) continue;
            int p=lca(u,stk[top]);
            if(p!=stk[top]) {
                while(id[p]<id[stk[top-1]])
                    add(stk[top-1],stk[top]),top--;
                if(id[p]!=id[stk[top-1]])
                    clear(p),add(p,stk[top]),stk[top]=p;
                else add(p,stk[top--]);
            }
            clear(u);
            stk[++top]=u;
            key[u]=1;
        }
        for(int i=0;i<top;i++) add(stk[i],stk[i+1]);
    }

    void init() {
        lca_init(1, 0);
        relabel(1, 0);
    }

    void clear(int n) {
        idx=0;
        for(int i=0;i<=n;i++) adj[i].clear();
    }
}
```