Algorithm Template

AQP

January 26, 2019

Contents

1 Dynamic programming

2 DataStructure

2.1 SparseTable

```
struct SparseTable {
public:
  SparseTable(int* data, int n) {
    int size_ = n;
    min_ = (int**)malloc(sizeof(int*)*size_);
    max_ = (int**)malloc(sizeof(int*)*size_);
    for(int i = 0; i < size_; ++i) {
      min_[i] = (int*)malloc(sizeof(int)*(int
      )ceil(log(size_)));
      max_[i] = (int*)malloc(sizeof(int)*(int
      )ceil(log(size_)));
      min_[i][0] = data[i];
      max_[i][0] = data[i];
    }
    for(int j = 1; (1<<j) <= size_; ++j)
      for(int i = 0; i + (1<<j) - 1 < size_;
      ++i) {
        min_[i][j] = std::min(min_[i][j-1],
        min_[i + (1<<(j-1))][j-1]);
        max_[i][j] = std::max(max_[i][j-1],
        max_[i + (1<<(j-1))][j-1]);
      }
  }

  ~SparseTable() {
    for(int i = 0; i < size_; ++i) {
      free(min_[i]);
      free(max_[i]);
    }
    free(min_);
    free(max_);
  }

  int Query(int l, int r) {
    int k = 0;
    while((1<<(k+1)) <= r-l+1) ++k;

    //return min(min_[l][k], min_[r-(1<<k)+1][k]);
    return max(max_[l][k], max_[r-(1<<k)+1][k
    ]) - min(min_[l][k], min_[r-(1<<k)+1][k
    ]);
  }

private:
  size_t size_;
  int** min_;
  int** max_;
};
```

2.2 Treap

```
struct Treap {  // 树堆
  struct Node {
    Node(int x): v(x) { ch[0] = ch[1] = NULL;
    r = rand(); s = 1;}

    int cmp(int x) const {
      if(x == v) return -1;
      return x < v ? 0 : 1;
    }

    void Maintain() {
    // for rank tree
      s = 1;
      if(ch[0] != NULL) s += ch[0]->s;
      if(ch[1] != NULL) s += ch[1]->s;
    }

    Node *ch[2];       // 左右子树
    int r;
    // 优先级，r 越大优先级越高
    int v;             // 值
    int s;
    // for rank tree, 表示以此节点为根的子树的节点数
  };

  void Rotate(Node* &o, int d) {
  // 引用 o 表示可以上 o 指向别的地址
    Node* k = o->ch[d^1];
    // d 为 0 或 1, d^1 表示 1-d
    o->ch[d^1] = k->ch[d];
    k->ch[d] = o;
    o->Maintain();     // for rank tree
    k->Maintain();
    o = k;
  }

  void Insert(Node* &o, int x) {
    if(o == NULL) {
      o = new Node(x);
    }
    else {
      int d = o->cmp(x);
      Insert(o->ch[d], x);
```

```
      if(o->ch[d]->r > o->r) Rotate(o, d^1);
    }
  }

  void Remove(Node* &o, int x) {
    int d = o->cmp(x);
    if(d == -1) {
      if(o->ch[0] == NULL) o = o->ch[1];
      else if(o->ch[1] == NULL) o = o->ch[0];
      else {
        d = (o->ch[0]->r > o->ch[1]->r ? 1 :
        0);
        Rotate(o, d);
        Remove(o->ch[d], x);
      }
    }
    else {
      Remove(o->ch[d], x);
    }
  }

  int Find(Node *o, int x) {
    while(o != NULL) {
      int d = o->cmp(x);
      if(d == -1) return 1;
      else o == o->ch[d];
    }
    return 0;
  }

};
```