Fenwick Tree and Segment Tree

# Scenario

**应用：**维护数组中的连续范围的[a, b]的属性P(a, b)

**条件：**存在操作符 +, 对任意c: a <= c <= b, P(a, b) = P(a, c) + P(a, b)

**典型问题：**

* **Range sum:** 属性是range sum, 操作符是加法
* **Range addition:** 属性是包含[a, b]的所有range, +是并操作
* **Weighted RAM:** 属性是[a, b]返回内的权重和， 操作符是加法

# Overview

**Fenwick Tree (Binary Index Tree):**

* update element, retrieve prefix sum (cumsum)
* log(N)时间写
* log(N)时间读
* 空间开销O(N) （如果不需要存储原始数据，为N)
* 动态增加capacity只需要resize数组

**ZKW Segment Tree**

* 三种使用模式：  
  (1) update element, retrieve range sum (2) update range sum, retrive element (3) update element, retrieve range whose sum >= given value
* log(N)时间修改单个元素
* log(N)时间检索range sum
* 空间开销O(N) (树的最后一层存储了原始数据）
* 动态增加capacity需要在resize数组以后移动数据，amortized cost为O(1)

**比较：**

* Fenwick Tree是多叉树，自顶向下search比较难
* Fenwick Tree主要针对prefix sum，如果要作range sum必须有合法有效的减法operator
* Fenwick Tree解释起来没有Segment Tree容易
* ZKW Segment Tree 占的空间大一点点（需要是2的整数次，resize的逻辑复杂一点点）

**几个例子：**

* Range sum query - mutable (Leetcode 307)
* Range sum query 2D - mutable (Leetcode 308)
* Range addition (Leetcode 370，原题其实不需要，但如果followup数据mutable怎么办，则需要）
* Weighted RAM (Two sigma面经）

# 

# Fenwick Tree

设计思想：通过把cumsum的值分散保存在idx在树中的路径上，并用位运算映射树结构，使检索更新cumsum快速简单

例子：下图圆圈中蓝字为下标idx，黑字为值tree[idx]。红线指向后继next，黑线指向parent.

结点关系映射：

第k层的index有k位1.

黑线－从子结点到父结点index的映射: **parent(n) = n & (n-1)**，即最后一位置0.

红线－从结点到它后继结点的映射： **next(n) = n + (n & -n)**. 左移最后一位1,或把最后一段0111...11结构变为1000...00

例子：

**idx** 0 1 2 3 4 5 6 7 8 9 10 11 12

(binary) 0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100

parent[idx] - 0 0 2 0 4 4 6 0 8 8 10 8

(binary) - 0 0 10 0 100 100 110 0 1000 1000 1010 1000

next[idx] - 2 4 4 8 6 8 8 - 10 12 12 -

(binary) - 10 100 100 1000 1q0 1000 1000 - 1010 1100 1100 -

对后继映射的进一步解释：

后继结点定义为下一个兄弟结点，或如果当前结点n是父亲的最右结点，后继是n第一个非最右结点祖先的后继结点。

对第一种情况，注意到(n&-n)取了n最右一位就可以得到结论。对第二种情况，注意到所有最右结点都以x0 11...1100..00结尾，它的祖先结点中最低的不是最右结点的key是x0 10...0，而它的右结点值是x1 00...1100..00，正好还是n + (n & -n)。

数据映射

tree[idx] = cumsum[idx] - cumsum[parent[idx]]

例如上图对应下表：

**idx** 0 1 2 3 4 5 6 7 8 9 10 11 12

input[idx] 2 1 1 3 2 3 4 5 6 7 8 9

cumsum[idx] 0 2 3 4 7 9 12 16 21 27 34 42 51

**tree[idx]** 0 2 3 1 7 2 5 4 9 6 13 8 30

初始化：tree[idx]初始化为长为数组长度+1的全零数组。

检索：检索cumsum只需把node n+1路径上所有结点加起来，求range sum就是两个cumsum之差。

更新input[n]：假设增加d, 影响的元素是结点n+1以及它到根结点路径上所有结点的后续兄弟结点。这些元素都需要加d

例如更新input[4]，则需更新结点5(对应input[4]), 6, 8.

最后注意Fenwick tree不直接存储data，因此如果问题需要access data方便起见最好另存一份。

|  |
| --- |
| **class NumArray** {  public:  **NumArray(vector<int> &nums)** : data(nums), tree(nums.size()+1, 0) {  for (int i = 0; i < data.size(); ++i)  for (int key = i+1; key < tree.size(); key += (key&-key)) tree[key] += data[i];  }  **void update(int i, int val)** {  int diff = val - data[i];  for (int key = i+1; key < tree.size(); key += (key&-key)) tree[key] += diff;  data[i] = val;  }  **int sumRange(int i, int j)** {  return cumSum(j) - cumSum(i-1);  }  private:  int cumSum(int i) {  if (i < 0) return 0;  int sum = 0;  for (int key = i + 1; key > 0; key &= (key-1)) sum += tree[key];  return sum;  }  vector<int> tree;  vector<int> data;  }; |

# ZKW Segment Tree

线段树是一棵二叉树。每个结点对应一个区间[l,r),结点的两个子结点把这个区间的值分为均匀的两段。每个结点还存储这个区间里的sum。根结点对应[0, n)。

这里我们使用数组存储一棵二叉完全树。为index方便，我们把n扩展到2的整数次幂M = 2k. 上面的例子对应的Segment Tree如下图。

5

**[4,6)**

51

**[0,16)**

21

**[0,8)**

30

**[8,16)**

7

**[0,4)**

14

**[4,8)**

3

**[0,2)**

1

**[2,3)**

3

**[5,6)**

30

**[8,12)**

9

**[6,8)**

6

**[8,9)**

17

**[10,12)**

9

**[11,12)**

2

**[0,1)**

1

**[1,2)**

3

**[3,4)**

2

**[4,5)**

4

**[6,7)**

5

**[7,8)**

7

**[9,10)**

8

**[10,11)**

4

**[2,4)**

13

**[8,10)**

0

**[12,16)**

0

**[12,14)**

0

**[14,16)**

**1(1)**

**2(10)**

**3(11)**

**4(100)**

**5(101)**

**6(101)**

**7(111)**

**8(1000)**

**9(1001)**

**10(1010)**

**11(1011)**

**12(1100)**

**13(1101)**

**14(1110)**

**15(1111)**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28,29**

**30,31**

其中的红字表示结点的index。

结点关系映射：注意到index的父结点是子结点的前缀。因此

* parent(n) = n>>1;
* left\_child(n) = n\*2;
* right\_child(n) = n\*2+1;

数据映射：

define BASE = next\_power\_of\_two(size of input)

tree[n] = tree[left\_child[n)] + tree[right\_child(n)] for n < BASE

tree[n] = input[n-BASE) for n >= BASE

初始化：

这样构造树时，则把输入复制从tree[M]开始的空间。然后从tree[M-1]开始，用tree[n] = tree[n\*2] + tree[n\*2+1]构造。

检索：

当我们需要检索区间(j, k)时(即区间[j+1, k-1])，相关结点都在j, k的最小公共子树里，然后寻找到j的path上所有左结点的右兄弟之和，和到k的path上所有右结点的左兄弟之和。我们只需要迭代指向j, k的父亲，直到两者指向同一结点。

更新：当更新A[k]时，只要相应更新根结点到结点[k,k]路径上所有结点的value.这通过迭代访问parent(n)即可做到。

边界条件：开区间访问需要检索A[-1]和A[end+1]，因此我们增加两个元素，同时把index向右平移1.

|  |
| --- |
| inline unsigned int next\_power\_of\_two(unsigned int v) {  --v;  v |= (v >> 1);  v |= (v >> 2);  v |= (v >> 4);  v |= (v >> 8);  v |= (v >> 16);  return v+1;  }  **class NumArray** {  public:  **NumArray(vector<int> &nums)** : M(next\_power\_of\_two(nums.size()+2)) {  tree.resize(M<<1, 0);  copy(nums.begin(), nums.end(), tree.begin()+M+1);  for (int k = M-1; k>0; k--) tree[k] = tree[k<<1] + tree[k<<1|1];  }  **void update(int i, int val)** {  i++; //number is 1 based  int diff = val - tree[M+i];  for (int key = M+i; key > 0; key >>= 1) tree[key] += diff;  }  **int sumRange(int i, int j)** {  int sum = 0;  for (i += M, j += M+2; i != j^1; i >>= 1, j >>= 1) {  if (!(i&1)) sum += tree[i^1];  if (j&1) sum += tree[j^1];  }  return sum;  }  private:  vector<int> tree; //tree stored in array, size is M\*2  int M; //smallest power of 2 greater or equal than nums.size()+2  }; |

# 扩展到高维度

**例子：range sum query 2D**

|  |
| --- |
| Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).  Note:  The matrix is only modifiable by the update function.  You may assume the number of calls to update and sumRegion function is distributed evenly.  You may assume that row1 ≤ row2 and col1 ≤ col2. |
| Example: Given matrix = [  [3, 0, 1, 4, 2],  [5, 6, 3, 2, 1],  [1, 2, 0, 1, 5],  [4, 1, 0, 1, 7],  [1, 0, 3, 0, 5]  ]  sumRegion(2, 1, 4, 3) -> 8  update(3, 2, 2)  sumRegion(2, 1, 4, 3) -> 10 |

**Solution: Segment tree**

把1D的Segment Tree（或Fenwick Tree）扩展到2D。办法是类似1D segment tree划分结点。

**Step 3**

**Step 1**

**Step 2**

数据结构：上左图中红色区域表示数组，蓝色区域表示tree的数据，记录某个2D region的sum，黄色块和红蓝线表示结点间的父子关系。不难看出这个2D矩阵的每行、每列都是一个1D Segment tree

初始化： 首先把数据copy到矩阵右下角，然后先按行初始化data块左边的tree数组，然后按列更新前一半行的tree 数组。如上右图。

更新：

如上左图。但更新结点时因为data是2D的。因此结点(i, j)的更新要传播要所有i的祖先和j的祖先的笛卡尔积上。

导致的变化在于每个结点需要更新log(m)\*log(n)个结点，

检索：

如上右图，首先对每行做range sum。然后对每列做range sum

类似的，把fenwick tree扩展到2D也可以解决这个问题

|  |
| --- |
| inline unsigned int next\_power\_of\_two(unsigned int v) {  --v;  v |= (v >> 1);  v |= (v >> 2);  v |= (v >> 4);  v |= (v >> 8);  v |= (v >> 16);  return v+1;  }    **class NumMatrix** {  public:  **NumMatrix(vector<vector<int>> &matrix)**  : BASE0(next\_power\_of\_two(matrix.size()+2)), BASE1(next\_power\_of\_two(matrix.size()+2)),  tree(BASE0 \* 2, vector<int>(BASE1 \* 2, 0)) {  for (int i = 0; i < matrix.size(); ++i) {  copy(matrix[i].begin(), matrix[i].end(), tree[BASE0+i+1].begin()+BASE1+1);  for (int j = BASE1 -1; j>0; --j)  tree[BASE0+i+1][j] = tree[BASE0+i+1][j<<1]+tree[BASE0+i+1][j<<1|1];  }    for (int i = BASE0-1; i >0; --i)  for (int j = BASE1+matrix[0].size(); j >0; --j)  tree[i][j] = tree[i<<1][j] + tree[i<<1|1][j];  }  **void update(int row, int col, int val)** {  ++row, ++col; //1 based index  int diff = val - tree[BASE0+row][BASE1+col];  for (int k = BASE0+row; k > 0; k >>=1)  for (int j = BASE1+col; j >=0; j>>=1) tree[k][j] += diff;  }  **int sumRegion(int row1, int col1, int row2, int col2)** {  int sum = 0;  for (int i1 = row1+BASE0, i2 = row2+BASE0+2; i1^i2^1; i1>>=1, i2>>=1) {  if (!(i1&1)) sum += sumRow(i1^1, col1, col2);  if (i2&1) sum += sumRow(i2^1, col1, col2);  }  }  private:  inline int sumRow(int i, int col1, int col2) {  vector& t = tree[i];  int sum = 0;  for (int j1 = col1+BASE1, j2 = col2+BASE1+2; j1^j2^1; j1>>=1, j2>>=1) {  if (!(j1&1)) sum += t[j1^1];  if (j2&1) sum += t[j2^1];  }  return sum;  }    const int BASE0, BASE1; //number of leaf nodes in each dimension  vector<vector<int>> tree;  }; |

# 写Range，读Element

**例子： Range addition**

|  |
| --- |
| Assume you have an array of length n initialized with all 0's and are given k update operations.  Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc.  Return the modified array after all k operations were executed. |
| Example:  Given:  length = 5,  updates = [  [1, 3, 2],  [2, 4, 3],  [0, 2, -2]  ]  Output: [-2, 0, 3, 5, 3] |

**Solution: Segment tree**

把segment tree的读写操作互换，modify range时更新相关非叶结点，retrieve元素的值时访问对应叶子结点到根结点路径上的node之和。

|  |
| --- |
| inline unsigned int next\_power\_of\_two(unsigned int v) {  --v;  v |= (v >> 1);  v |= (v >> 2);  v |= (v >> 4);  v |= (v >> 8);  v |= (v >> 16);  return v+1;  }  **vector<int> getModifiedArray(int length, vector<vector<int>>& updates)** {  int M = next\_power\_of\_two(length+2);  vector<int> tree(M\*2, 0);  for (vector<int>& v : updates) {  int val = v[2];  for (int i = v[0]+M, j = v[1]+M+2; i ^ j ^ 1; i >>= 1, j >>=1 ) {  if (!(i&1)) tree[i^1] += val;  if (j&1) tree[j^1] += val;  }  }    vector<int> result(length, 0);  for (int i = 1; i <= length; ++i) {  int sum = 0;  for (int j = M+i; j > 0; j >>= 1) sum += tree[j];  result[i-1] = sum;  }  return result;  } |

# 动态分配空间

**例子：Weighted RAM(Two Sigma面经题)** 让设计一个数据结构，要求可以存储Object-Weight Pair，实现如下几个接口：1） Update；2） Insert；3） Remove；4） GetRandom. 第四个方法是实现的重点。这个GetRandom的方法是随机地返回一个Object，要求概率满足：此object的weight / 所有weight的和。 楼主想的是HashMap的Key存Object，Value存weight，这样可以轻松实现Update，Insert和remove的功能。至于GetRandom这个方法，楼主是用了一个辅助的Array，用来表明每个Object对应的区间，然后用随机数获得某个index，最后看看这个index在哪个区间，然后就返回该对象。但是缺点是：每每更新，插入或者remove掉某个object的时候，这个辅助的array都要重新计算，有没有更好的方法来解决此题？貌似面试官一直在提normalization 和Denormalization。不能理解面试官要的是什么.

Source: <http://www.1point3acres.com/bbs/thread-198541-1-1.html>

**Solution:**

用一个segment tree的结构来存weights。一个queue用来存放树里available的位置。两个unordered\_map用来存segment tree的叶子结点和object之间的双向mapping。

随机采样：

产生一个[1, sum of weights]范围内的数。搜索cumsum(weight)数组的lower\_bound。这可以通过对segment tree作二分搜索得到。

数据添加、删除、更新

维护queue和unordered\_map的操作比较直观。

对segment tree的修改体现在删除时把原来非0的叶子结点置0. 添加时需要检查是否当前capacity已满，如果已满则需要先resize数组。然后update queue指定的叶子位置就好。

|  |
| --- |
| **class weighted\_RAM** {  public:  **weighted\_RAM() : M(128), tree(M\*2, 0)** {  for (int k = 1; k < M; ++k) loc.push(k); //free spot from 1 to M-1  }    **void add(const string &str, int w)** {  if (loc.empty()) resize();  int i = loc.front();  loc.pop();  update\_leaf(i, w);  idx\_to\_obj[i] = str;  obj\_to\_idx[str] = i;  }    **void update(const string &str, int w)** {  int i = obj\_to\_idx[str];  update\_leaf(i, w-tree[M+i]);  }    **void remove(const string &str)** {  int i = obj\_to\_idx[str];  loc.push(i);  update\_leaf(i, -tree[M+i]);  idx\_to\_obj.erase(i);  obj\_to\_idx.erase(str);  }  **string get\_random()** {  default\_random\_engine eng;  int key = (eng())%tree[1]+1;  int i = sum\_search(key+1);  return idx\_to\_obj[i];  }  private:  void resize() {  tree.resize(M\*4, 0);  for (int base = M; base; base >>= 1) //move current nodes to new locations  rotate(tree.begin()+base, tree.begin()+base\*2, tree.begin()+base\*3);  tree[1]=tree[2]; //new root  for (int k = M; k < M\*2; ++k) loc.push(k);  M<<=1;  }    int sum\_search(int w) {  int i = 1;  do {  i <<= 1;  if (w > tree[i]) w -= tree[i++];  } while (i < M);  return i-M;  }  void update\_leaf(int i, int diff) {  for (int key = M+i; key > 0; key >>= 1) tree[key] += diff;  }  private:  int M; //smallest power of 2 greater or equal than nums.size()+2  vector<int> tree; //tree stored in array, size is M\*2  queue<int> loc; //free spots  unordered\_map<int, string> idx\_to\_obj;  unordered\_map<string, int> obj\_to\_idx;  }; |