Data Structures

Lesson 2

Topics

- STL Data Structures
- Non-STL Data Structures

Static Array

```
int array1[10];
int array2[10] = {0,1,2,3,4,5,6,7,8,9}
int array3[10] = {1} // [1,0,0,0,0,0,0,0,0,0]
int array4[10] = {0} // [0,0,0,0,0,0,0,0,0,0]

cin >> array1[0]; //Get an integer
int* array5 = new int[*array1] //Dynamic array allocation

for(i = 0; i <= 10; ++i){ //Get an entire array
  cin >> array2[i];
}

Int array6[2][3] = {{1,2,3},{4,5,6}}
```

Vector

- Dynamic array – basic operations are $O(1)$ amortized
- Can be used as list, queue, stack
- Common functions:
  - []
  - push_back()
  - pop_back()
  - insert()
  - erase()
  - size()
  - empty()
  - clear() //Use between test cases
  - resize() //Use between test cases

Vector

```
vector<int> arr1(5, 0); // [0,0,0,0,0]
arr1[0] = 1; // [1,0,0,0,0]
arr1.push_back(3); // [1,0,0,0,0,3]
vector<int> arr2 = arr1; //copy
arr1.pop_back(3); // [1,0,0,0,0]
auto it = arr1.begin();
it++; it++;
arr1.insert(it,arr2.begin(),arr2.end()) // [1,0,1,0,0,0,0,3,0,0,0]
arr1.erase(arr1.begin()) // [0,1,0,0,0,0,3,0,0,0]
```

Set

- Based on Balanced Binary Search Tree (Balanced-BST)
  - Basic operations - $O(\log n)$
  - Objects must support “<“ operator
- Common Functions:
  - insert()
  - search()
  - erase()
  - size()
  - empty()
  - min/max – Not functions, see code
  - lower_bound()/upper_bound()

Set

rbrgin++ go reverse
always set is sorted

```
set<int> set1 = {3,14,15,92,65,35,89,79,32,38};
int min = *set1.begin(); // 3
int max = *set1.rbegin(); // 92
cout << *set1.lower_bound(15); // 15
cout << *set1.lower_bound(16); // 32
```

map

is also sorted by the first value

- Similar to set, but supports key-value operations

```
map<int, char> map1 = {{1, 'a'}, {3, 'c'}, {2, 'b'}};
    for(auto it: map1){ cout << it.first << it.second; } //abc
    map1[3] = 'C';
    cout << map1[3]; // C
    map1[4] = 'd';
    map1[5] = 'e';
    auto itlow = map1.lower_bound(2); //inclusive
    auto itupp = map1.upper_bound(4); //exclusive
    cout << itlow->second; // b
    cout << itupp->second; // e
    map1.erase(itlow,itupp);
    for(auto it: map1){ cout << it.first << it.second;} //ae
```

multiset/multimap

- multiset/multimap – allows key repetition
- Also Balanced-BST
- No [] operator

Set and Map variations

not sorted is implemented by hashmap

- unordered_set/unordered_map
- Based on Hash Table
  - Basic opertaions – $O(1)$ (on average)
  - Objects must support “()“ operator (implementing an hash function)
  - And “==“ operator.
- Not recommended for complex data types in competitive programing

unordered_set/unordered_map

- Based on Hash Table
  - Basic opertaions – $O(1)$ (on average)
  - Objects must support “()“ operator (implementing an hash function)
  - And “==“ operator.
- Not recommended for complex data types in competitive programing

unordered_set/unordered_map

```
unordered_map<string, int> ht;
ht["key"] = 5;
cout << ht["key"]; // 5    insert the key to the hashmap
if (ht.find("key") != ht.end())    find return iterator if it founded else return iterator to end
   cout << "key found";
else
   cout << "key not found";
cout << "second=" << ht["second"];
// Implicitly create an item (val=0) for non-existant keys.
ht[”third"]++;
// Increase by 1 if the key  already exists in the hash table. If not,
insert the key and set the value as 1.
for (auto& p : ht) //Order unknown
{ cout << "key=" <<p.first << " " << "val="<< p.second << endl; }
```

priority_queue

- Min/Max heap
- Common functions:
  - top()
  - pop()
  - push()
  - empty

priority_queue

```
vector<int> arr = { 5,3,6,3,2,1,1 };
priority_queue<int> max_heap(arr.begin(),arr.end()); // O(n)
while (!max_heap.empty()) { // 6,5,3,3,2,1,1
cout << max_heap.top();
max_heap.pop(); // no return val
}
priority_queue<int, vector<int>, greater<int> > min_heap;    min heap
min_heap.push(6);min_heap.push(8);
min_heap.push(1);min_heap.push(2); // O(log n)
while (!min_heap.empty()) // 1,2,6,8
{
cout << min_heap.top() << endl;
min_heap.pop();
}
```

Break
&
Competition

bitset

- An efficient Boolean array with logic operations
- Common functions:
  - reset()
  - set()
  - test()
  - count()
  - any()
  - all()
  - none()
  - &,|,^,~ operators (And, Or, Xor, Not)
  - >>, << (Bit shift)

bitset

```
bitset<8> bset1; // [0,0,0,0,0,0,0,0]
bset1.set(); // [1,1,1,1,1,1,1,1]
bset1.reset(3); bset1.reset(5); // [1,1,1,0,1,0,1,1]
bset1.flip(6); // [1,1,1,0,1,0,0,1]
cout << bset1.none(); // false   all of them is zero
cout << bset1.any(); // true   any of them is one
cout << bset1.all(); // false   all of them is one
bitset<8> bset2; // [0,0,0,0,0,0,0,0]
bset2.set(0); // [1,0,0,0,0,0,0,0]
bset2 = (bset2 >> 3) | (bset2 >> 5) // [0,0,0,1,0,1,0,0]
cout << (bset1 & bset2).none(); // true
cout << (bset1  | bset2).all(); // true
cout << ((~bset1)  ^ bset2).any(); // false
```

bitmask

- If the number of flags is small we can use integers instead of bitset
- Advantage: Much faster
- Disadvantage: No built in interface

```
int b1 = 10; // 00001010
int b2 = 18; // 00010010
int b3 = b1 & b2; // 00000010
int b4 = b1 | (b3<<4); // 00101010
for(int i = 0; i < 8; ++i){
  if(b4 & 1<<i) cout << "true";
  else          cout << "false";
} // false,true,false,true,false,true,false,false
```

Union-Find

- **Goal**: Given a collection of items $\{1, \ldots, n\}$, construct an efficient data structure which supports the following operations:
  - **Find**(i): Determine which subset a particular element is in. By comparing the result of two Find operations, one can determine whether two elements are in the same subset.
    Union(i, j): Join two subsets into a single subset.
- Union-Find is a part of the Kruskal MST algorithm.

Union-Find

- Naïve implementation – Graph of items
  - Union(i, j) adds an edge i and j, Find(i) determines subset ID using DFS.
  - Worst case time complexity for $O(n^2)$
- Better solution - Disjoint-set forests
  - Union(i, j), Find(i) in $O(\log^* n)$ amortized – almost constant!
- Implementation:
  - Inverted trees for the disjoint-set forests are represented using parent arrays.
  - Refer to [DS1]() or [Wikipedia]() and the attached .cpp file for more info.

Do Union(0, 1)

```
  1  2  3
 /
0
```

Do Union(1, 2)

```
  1   3
 / \
0   2
```

Do Union(2, 3)

```
   1
 / | \
0  2  3
```

Union-Find

```
struct unionfind
{
    vector<int> rank;
    vector<int> parent;
    unionfind(int size)    {
        rank=vector<int>(size,0);
        parent=vector<int>(size);
        for(int i=0;i<size;i++)
            parent[i]=i;
    }
    int find(int x){
      int tmp=x;
      while(x!=parent[x]) x=parent[x];
      while(tmp!=x)      {
            int remember=parent[tmp];
              parent[tmp]=x;
            tmp=remember; }
```

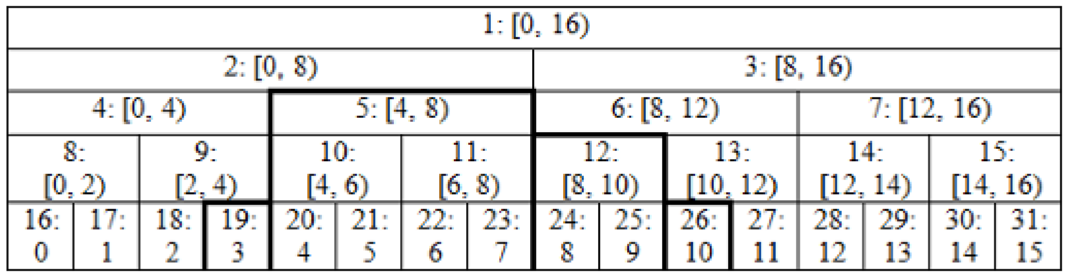
```
        return x;
    }
    void Union(int p, int q){
        p = find(p);
        q = find(q);
        if(q==p)
        {
            return;
        }
        if(rank[p] < rank[q]) parent[p]
= q;
        else parent[q] = p;
        if(rank[p] == rank[q])
rank[p]++;
    }
};
```

Segment tree

- **Problem**: Given an array of numbers $(a_1, \dots, a_n)$, construct an efficient data structure which supports the following operations:
  - **Add**(i, x): Add x to the i-th cell $(a_i \leftarrow a_i + x)$
  - **Max**(i, j): Return $\max\{a_i, \dots, a_j\}$

- Naïve approach - Use an array.
  - Add(i, x) in $O(1)$, Max(i, j) complexity is $O(n)$.
  - Worst case complexity for m queries is $O(mn)$.
- Better: Use a segment tree!
  - Add(i, x), Max(i, j) are computed in $O(\log n)$.
  - Worst case complexity for m queries is $O(m \log n)$.

Segment Tree



- Can be easily adapted to compute the sum/xor/max/min of all elements in a given segment.
- Implementation:
  - Example is included in the attached .cpp file.
  - Refer to Wikipedia for more info.
  - The following blog post contains a brief introduction and examples: http://codeforces.com/blog/entry/18051

Segment Tree

```
const int N = 1e5;  // limit for array size
int n;  // array size
int t[2 * N];

void build() {  // build the tree
    for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}
void modify(int p, int value) {  // set value at position p
    for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}
int query(int l, int r) {  // sum on interval [l, r)
    int res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l&1) res += t[l++];
        if (r&1) res += t[--r];
    }
    return res;
}
```

Fenwick tree (BIT)

- Similar to a Segment Tree, but even quicker to implement.
- Only supports the **Sum**$(0, j)$ sums (i.e prefix sums).

- For more info-
  - https://www.hackerearth.com/practice/data-structures/advanced-data-structures/fenwick-binary-indexed-trees/tutorial/
  - https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

Fenwick tree (BIT)

```
int BIT[1000] = {0}, a[1000], n;

void update(int x, int val)
{
      for(; x <= n; x += x&-x)
        BIT[x] += val;
}
int query(int x)
{
     int sum = 0;
     for(; x > 0; x -= x&-x)
        sum += BIT[x];
     return sum;
}
```