## Pointers

Storing, data type of pointers and variables must be the same

| &var | Returns address of memory location of variable |
|---|---|
| data_type *pointer; | Initialize. Put * in front of name |
| *pointer | Returns value at the memory location stored by pointer |

Array variables are actually pointers to the first element in the array. The amount that your pointer moves from an arithmetic operation

| *array | Returns first element in array |
|---|---|
| *(array+2) | Returns third element in array |

◎ Variables that you declare are stored in a memory location in your computer
◎ The address of these memory locations can be stored in pointers
◎ Addresses are in hexadecimal

## Iterators

```
//Append ::iterator to your data type declaration
to create an iterator of
that data type
vector<int>::iterator it; // declares an iterator
of vector<int>
// loops from the start to end of vi
for(vector<int>::iterator it = vi.begin(); it !=
vi.end(); it++)
    cout << *it << " "; // outputs 1 2 3 4
deque<int> d;
deque<int>::iterator it;
it = d.begin(); //Points to first element
it++; // Points to next element
it = d.end(); // Points to Last element
it--; // Points to previous element
cout << *it; // outputs the element it is pointing
```

Iterators are essentially pointers to an STL data structure

## Maps

```
map<string, int> M;
M["hello"] = 2;
M["asd"] = 986;
M.count("asd"); // returns 1
M.count("doesnt_exist"); // returns 0
M.size();
// Check for the existence of some key in the map -
O(log N)
it = M.find("asd"); //returns the iterator to
"asd"
it = M.upper_bound("aaa");
it = M.lower_bound("aaa");
if (it == M.end())
    cout << "Does not exist\n";
//Iteration
for (auto it = mp.begin(); it != mp.end(); ++it) {
    cout << it.first << ", " << it.second << "\n";
}
```

A data structure that takes in any data[key]
Gives you the associated value stored through O(log N) magic

Best used when you need to lookup certain values in O(log N) time that are associated with other values

## Queue

```
queue<int> q;
q.push(5); // Inserts/ Enqueue element at the back
of the queue
q.front(); // Returns element atb the front of the
queue
q.pop // Removes (Dequeues) element from the front
of the queue
q.empty(); // Returns boolean value of whether
queue is empty
```

First In First Out data structure where elements can only be added to the back and accessed at the front

---

## Priority Queue

```
priority_queue<data_type> pq; // Largest at top
priority_queue<data_type, vector<data_type>,
greater<data_type> >
pq; // Smallest at top
pq.push(5); // pushes element into it. Duplicates
are allowed
pq.top() // Returns largest or smallest element
pq.pop() // Removes largest or smallest element
pq.size(); // Returns size
pq.empty(); Check if queue is empty
```

Like a queue except that only the element with the greatest priority (eg. largest/smallest) can be accessed

## Fenwick tree

```
//Below here can mix & match
long long ft[100001]; // note: this fenwick tree
is 1-indexed.
////PU-
PQ////////////////////////////////////////////
//////
void fenwick_update(int pos, long long value) {
    while (pos <= N) {
        //cout<<"Fenwick Updating: "<<pos<<","<-
<value<<endl;
        ft[pos] += value;

        pos += pos&-pos;
    }
}
long long fenwick_query(int pos) {
    long long sum = 0;
    while (pos) { // while p > 0
        sum += ft[pos];
        pos -= pos&-pos;
    }
    return sum;
}
////RU-
PQ////////////////////////////////////////////
//////
void fenwick_range_update(int pos_a, int pos_b, int
val){
    //TLE way
```

## Fenwick tree (cont)

```
    //for (int i=pos_a;i<=pos_b;i++){fenwick_upd-
ate(i, val);}
    fenwick_update(pos_a, val);
    fenwick_update(pos_b+1, -val);
}
////PURQ///////////////
////////////////////////////////////////
long long fenwick_range_query(int pos_a, int
pos_b) {
    return fenwick_query(pos_b) - fenwick_query(p-
os_a-1);
}
////RU-
RQ//////////////////////////////////////////////
//////
long long B1[100001];long long B2[100001];
void base_update(long long *ft, int pos, long long
value){
  //Add largest power of 2 dividing x / Last set
bit in number x
  for (; pos <= N; pos += pos&(-pos))
    ft[pos] += value;
}
void rurq_range_update(int a, int b,long long v){
  base_update(B1, a, v);
  base_update(B1, b + 1, -v);
  base_update(B2, a, v * (a-1));
  base_update(B2, b + 1, -v * b);
}
void rurq_point_update(int a, long long v){
    rurq_range_update(a,a,v);
}
long long base_query(long long *ft,int b){
  long long sum = 0;
  for(; b > 0; b -= b&(-b))
    sum += ft[b];
  return sum;
}
// Return sum A[1...b]
long long rurq_query(int b){
```

C

By **Hackin7**
cheatography.com/hackin7/

Published 12th December, 2018.
Last updated 27th December, 2019.
Page 2 of 5.

Sponsored by CrosswordCheats.com
Learn to solve cryptic crosswords!
http://crosswordcheats.com

### Fenwick tree (cont)

```
  return base_query(B1, b) * b - base_query(B2, b);
}
//Return sum A[a...b]
long long rurq_range_query(int a,int b){
  return rurq_query(b) - rurq_query(a-1);
}
```

### Pair

```
// Initialise
pair<data_type_1, data_type_2> variable;
// OR
pair<data_type_1, data_type_2> variable = make_p-
air(value1,value2);
// Store values
variable.first = value;
variable second = value;
// Retrieve values
cout <<variable first << " " << variable.second <<
endl;
//Nesting pairs
pair<int, pair<int, int> > a;
a.first = 5;
a.second.first = 6;
a.second.second = 7;
```

Stores a pair of values

### Stack

```
stack<int> s;
s.push(5); // push an element onto the stack -
O(1)
s.pop(); // pop an element from the stack - O(1)
s.top(); // access the element at the top of the
stack - O(1)
s.empty(); // whether stack is empty - O(1)
```

First-In-Last-Out data structure
Only Element at the top can be accessed / removed

### Vector

```
// Initialize
vector<data_type> v;
v.push_back(value); // Add element to back
v.pop_back() // Remove last element
v.clear(); // Remove all elements
v[index] // Return element of index
v.back(); // Return last element
v.size(); // Return Size of vector
v.empty() // Return boolean value of whether
vector is empty
```

Like arrays but re-sizable. You can add and remove any number of elements from any position.

### Sets and Multisets

```
set<int> s; set<int>::iterator it;
multiset<int> s; multiset<int>::iterator it;
s.insert(10);
it = s.find(8)
it = s.upper_bound(7);
it = s.lower_bound(7);
s.erase(10); //Remove element from set
s.erase(it) //Can also use iterators
s.empty();
s.clear();
// to loop through a set
for(it = s.begin(); it != s.end(); it++)
    cout << *it << " "; // outputs 2 7 10
```

In a set: All elements are sorted and no duplicates
Multisets can store duplicates though

## Deque

```
deque<int> d;
// access an element / modify an element (0-indexed
as well) - O(1)
d[0] = 2; // change deque from {5, 10} to {2, 10}
d[0]; // returns a value of 2
d.back(); // get back (last) element - O(1)
d.front(); // get front (first) element - O(1)
d.clear() // Remove all elements
d.push_back(5); // add an element to the back -
O(1)
d.push_front(10); // add an element to the front -
O(1)
d.pop_back(); // remove the back (last) element -
O(1)
d.pop_front(); // remove the front (first) element
- O(1)
d.size(); //Return size
d.empty // Whether queue is empty
```

A stack and queue combined.

...or a vector that and be pushed and popped from the front as well.

Deque = Double Ended Queue!

## Segment Tree

```
struct node {
    int start, end, mid, val, lazyadd;
    node left, right;

    node(int _s, int _e) {
        //Range of values stored
        start = _s; end = _e; mid = (start+end)/2;
        //Min value stored
        val = 0; lazyadd = 0;
        if (start!=end) {
            left = new node(start,mid);
            right = new node(mid+1,end);
        }
    }

    int value(){
        if (start==end){
            val += lazyadd;lazyadd = 0;return val;
```

## Segment Tree (cont)

```
        }else{
            val += lazyadd;
            // Propagate Lazyadd
            right->lazyadd += lazyadd;
            left->lazyadd += lazyadd;
            lazyadd = 0;
            return val;
        }
    }

    void addRange(int lower_bound, int upper_-
bound, int val_to_add){
        if (start == lower_bound && end == upper_-
bound){
            lazyadd += val_to_add;
        }else{
            if (lower_bound > mid){
                right->addRange(lower_bound,
upper_bound, val_to_add);
            }else if (upper_bound <= mid){
                left->addRange(lower_bound,
upper_bound, val_to_add);
            }else{
                left->addRange(lower_bound, mid,
val_to_add);
                right->addRange(mid+1, upper_-
bound, val_to_add);
            }
            val = min(left->value(), right->va-
lue());
        }
    }

    // Update position to new_value // O(log N)
    void update(int pos, int new_val) { //position
x to new value
        if (start==end) { val=new_val; return; }
        if (pos>mid) right->update(pos, new_val);
        if (pos<=mid) left->update(pos, new_val);
        val = min(left->val, right->val);
    }

    // Range Minimum Query // O(log N)
```

By Hackin7
cheatography.com/hackin7/

Published 12th December, 2018.
Last updated 27th December, 2019.
Page 4 of 5.

### Segment Tree (cont)

```
    int rangeMinimumQuery(int lower_bound, int
upper_bound) {
        //cout<<"Node:"<<start<<" "<<end<<" "<<mid<<"
"<<val<<endl;
        //If Query Range Corresponds////////////////
        if (start==lower_bound && end==upper_bound){
            return value();
        }
        //Query Right Tree if range only lies there
        else if (lower_bound > mid){
            return right->rangeMinimumQuery(lower-
_bound, upper_bound);
        }
        //Query Left Tree if range only lies there
        else if (upper_bound <= mid){
            return left->rangeMinimumQuery(lower_-
bound, upper_bound);
        }
        //Query both ranges as range spans both trees
        else{
            return min(left->rangeMinimumQuery(lower-
_bound, mid),right->rangeMinimumQuery(mid+1, upper_-
bound));
        }
        //E-
nd///////////////////////////////////////
    }

} *root;
void init(int N){
    root = new node(0, N-1); // creates seg tree of
size n
}
void update(int P, int V){
    root->update(P,V);
}
int query(int A, int B){
    int val = root->rangeMinimumQuery(A,B);
    return val;
}
```