Iterators: std::list::iterator i,j;

```
std::advance(i,2); // takes an iterator and moves it n units. no return value
                   // Time complexity: O(n)

std::distance(i,j); // takes 2 iterators and returns the distance between them
                    // Time complexity: O(n)

std::begin(array); // takes in an array and returns a pointer to the first element
                   // in the array.

std::end(array); // take in an array and returns an iterator past the last element.

std::next(i,3); // returns an iterator n positions away from i.
                // Time complexity: O(n)

std::prev(i,2); // returns an iterator n positions away from i.
                // Time complexity: O(n)
```

Lists: std::list my_list = {0,1,2,3};

```
my_list.begin(); // returns an iterator to the beginning of the list
                 // Time complexity: O(1)
my_list.end(); // returns an iterator past the end of the list
               // Time complexity: O(1)

// Same with my_list.rbegin() & my_list.rend(). rbegin() returns
// an iterator to the end of the list while .rend() returns past
// the beginning of the list.
// you can add a 'c' to all 4 functions to return a const iterator.
// my_list.cbegin(), my_list.cend(), my_list.crbegin(), my_list.crend()
```

Modifiers

```
my_list.push_front(-1); // takes in a value and returns nothing.
                        // Time complexity: O(1)
```

```
my_list.pop_front(); // deletes the first element of the list.
                     // Time complexity: O(1)
```

```
my_list.push_back(4); // takes in a value and returns nothing.
                      // Time complexity: O(1)
```

```
my_list.pop_back(); // delete the last element of the list.
                    // Time complexity: O(1)
```

```
my_list.insert(iterator, value, n); // Time complexity: O(n)
// inserts the value n times at the iterator.
// returns an iterator at the last inserted value.
```

```
my_list.erase(iterator1, iterator2); // Time complexity: O(n)
                                     // n being distance between 2 iterators.
// if 1 iterator is passing in just deletes that value. else, it deletes
// everything in between including the first but not the last.
// returns an iterator next of the last deleted value
```

```
my_list.clear(); // deletes everything in the list.
                 // Time complexity: O(n) n being list size.
```

Vectors: std::vector<int> my_vec = {0,1,2,3,4};

```
my_vec.begin(); // returns an iterator to the begining of vector
my_vec.end(); // returns an iterator past the end of vector.
// same with .rbegin() & .rend().
// returns an iterator to the end of vector. returns an iterator past
// the beginning.
// adding a C makes it const.
// .cbegin(), .cend(), .crbegin(), .crend()
```

Element access

```
my_vec[int]; // returns the value // Time complexity: O(1)
```

```
my_vec.at(int); // at that index, set a number to that index.
                // returns nothing. // Time complexity: O(1)
```

```
my_vec.front(); // returns a refrence to the first value of vector
                // Time complexity: O(1)
```

```
my_vec.back(); // returns a refrence to the last value of vector
               // Time complexity: O(1)
```

Modifiers:

```
my_vec.push_back(value); // adds value to the back of vector.
                         // returns nothing // Time complexity: O(1)
```

```
my_vec.pop_back(); // deletes the last element of the vector
                   // returns nothing // Time complexity: O(1)
```

```
my_vec.insert(iterator, value, n); // Time complexity: O(n) n being
                                   // vec size
// inserts value n times at the iterator location.
// returns an iterator at the last value added
```

```
my_vec.erase(iterator1, iterator2); // Time complexity: O(n+m)
// n being distance between iterator1 & iterator2
// m being number of values that need shifted due to the change
// deletes all values between iterator1 & iterator2. Including
// iterator1, excluding iterator2
// returns an iterator after iterator2.

std::vector<int>::iterator i = my_vec.begin();
std::advance(i,2);                                  = 1, 3
std::cout << i[1] << ' ' << i[-1] << std::endl;
// this means you can subscript iterators
// only for vectors.
```

Rewriting push_back, pop_back, push_front, pop_front, insert, and delete.

```
Void list<T>:: push_back(const T &v) {
    Node<T>* newp = new Node<T>(v);
    // if the list is empty
    if (!tail) {
        head = tail = newp;
    } else {
        newp -> prev = tail;
        tail -> next = newp;
        tail = newp;
    }
    size++;
}
Void list<T>:: push_front(const T& v) {
    Node<T>* newp = new Node<T>(v);
    if (!head) {
        head = tail = newp;
    } else {
        head -> prev = newp;
        newp -> next = head;
        head = newp;
    }
    size++;
}
```

```
Void list<T>:: pop_back() {
    if (tail -> prev = NULL) {
        delete tail;
        head = tail = NULL;
    } else {
        Node<T>* newhead = tail -> prev;
        delete tail;
        tail = newhead;
        newhead -> next = NULL;
    }
    Size--;
}
Void list<T>:: pop_front() {
    if (head -> next = NULL) {
        delete head;
        head = tail = NULL;
    } else {
        Node<T>* newhead = head -> next;
        delete head;
        newhead -> prev = NULL;
        head = newhead;
    }
    Size--;
}
```

```
typename list<T>::iterator
list<T>:: insert(iterator i, const T& v) {
    Node<T>* p = new Node<T>(v);
    p -> prev = i.ptr -> prev;
    p -> next = i.ptr;
    i.ptr -> prev = p;
    if (i.ptr == head) {
        head = p;
    } else {
        p -> prev -> next = p;
    }
    size++;
    return iterator(p);
}
```

```
Void list<T>:: destory_list {
    Node<T>* temp;
    while (head != NULL) {
        temp = head;
        head = temp -> next;
        delete temp;
        size--;
    }
}
```

Recursion:

```
int find_path (int x, int y) {
    if (x == 0 || y == 0) { // base case
        return 1;
    }
    return find_path(x-1, y) + find_path(x, y-1);
} // finds all possible paths from a location to 0,0
```