Destructor Prototype: ~Table();

Function:

```
template <class T> void Table<T>::~Table() {
  for (unsigned int j = 0; j < rows; j++) {
    delete [] values[j];
  }
  delete [] values;
}
```

```
if (ptr_->right != NULL) {
        ptr_ = ptr_->right;
        while (ptr_->left != NULL) {
            ptr_ = ptr_->left;
        }
    }
    else {
        while (ptr_->parent != NULL && ptr_-
>parent->right == ptr_) {
            ptr_ = ptr_->parent;
        }
        ptr_ = ptr_->parent;
    }
    return *this;
```

Copy Constructor Prototype:

```
Table(const Table& t) { copy(t); }
```

Assignment Operator Prototype:

```
const Table& operator=(const Table& t);
```

Functions:

```
//Assgn 1 Tble 2 another, avoid self-assgnment
template <class T> const Table<T>&
Table<T>::operator=(const Table<T>& v) {
  if (this != &v) {
    destroy();
    this->copy(v); //Copy is below
  }
  return *this;
}
//Create the Tble as a copy of the given Tble
template <class T> void Table<T>::copy(const Table<T>& v)
{
  this->create(v.rows,v.cols);
  for (unsigned int i = 0; i < rows; i++) {
    for (unsigned int j = 0; j < cols; j++) {
      values[i][j] = v.values[i][j];
    }
  }
}
```

Const(antly screwing up consts):

-- Const objects can only be used by const member functions

-- In classes, if const at end of member function prototype then it does not change any member variables.

```
if (&r != this) {
    this->destroy_rope(root);
    root = this->copy_rope(r.root,NULL);
    size_ = r.size_;
  }
  return *this;
```

Order Notation:

-- O(1), a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.

-- O(log n), a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.

-- O(n), a.k.a. LINEAR. e.g., sum up a list.

-- O(n log n), e.g., sorting.

-- O(n^(1/2)), O(n^3), O(n^k), a.k.a. POLYNOMIAL, find the closest pair

-- O(2^n), O(kn), a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.

-- O(N * M), nested for loops.

(How to abuse the) Sort (function and get away with it):

```
#include <algorithm>
//function prototype for sorting & sort call example
bool by_total_snowfall(const Snow &a, const Snow &b);
sort(container.begin(), container.end(),by_total_snowfall);
```

Iterators (Abed)/Reverse Iterators (Evil Abed):

-- use dereference operator to access value at iterator (*)

-- use select/dereference operator to access member functions (itr->member()).

-- reverse_iterator increments backwards, find beginning reverse itr with .rbegin() and the .rend().

--*itr for value

-- itr->func() is the same as (*itr).func()

-- Iterators have de-increment/increment!

STD::FIND:

```
#include <algorithm>
std::find(container.begin(), container.end(), value);
```

Standard Library Containers:

Arrays: Can be dynamically created, fixed size, has [], created by type[size], int t[] = {4,5,3,2,2}, has size, iterator stuff, etc.

std::string: Container of chars, has iterator stuff, size(), [], can append with +=, push_back/pop_back, insert, erase.

std::vector: Has [], push/pop_back, insert, eras, and iterator stuff. Can access iterator with v.begin() + int.

std::list: Has iterator stuff, push/pop _ back/front, .front() and .back() for element access, no []! Not connected

Erase & Insert:

```
var.erase(iterator position);
//erases the object at position, returns next
var.insert(iterator position, val);
//inserts val in container before position
container<type>::iterator for itr
```

std::map: Keys need operator<, tree based (red and black), log(n). Insert takes pair, returns pair <iter, bool>. Find takes in value, returns iter or end. Erase takes iter or key, or range of iter.

std::set: Insert takes key, returns pair iter bool. Erase takes key returns int. Find takes key, returns iter (end if not).

Recursion Example:

```
int intpow(int n, int p) {
   if (p == 0) {
      return 1;
   } else {
      return n * intpow(n, p-1);
   }
}
void countdown(int n) {
   std::cout << n << std::endl;
   if (n == 0) return;
   else countdown(n-1);
}
```

Operators:

```
+,-,*, /, %, >, <, !=, ==, +=, -=,*=, /=, %=
Also! Don't forget you can ++i and --i.
```

Assignment Operator Special: (:)

```
TrainCar(char t, int w) : type(t), weight(w), prev(NULL){
    //other function stuff can go here
}
```

Recursive Print Data:

```
void PrintData(Node *head) {
    if (head == NULL) return; //(!head) works
    std::cout << head->value << " ";
    PrintData(head->next);
}
```

Mirror:

```
void destroy(TriNode *n) {
// base case
2if (n == NULL) return;
// recursively delete the children
destroy (n->left);
destroy (n->middle);
destroy (n->right);
// then delete this node
delete n;
}
// helper function
TriNode* copy_mirror(TriNode *n) {
// base case
if (n == NULL) return NULL;
// create a new node on the heap
TriNode *tmp = new TriNode(n->val);
// copy, swapping left and right
tmp->left = copy_mirror(n->right);
tmp->middle = copy_mirror(n->middle);
tmp->right = copy_mirror(n->left);
return tmp;
}
// primary function
void make_symmetric(TriNode* n) {
// base case
if (n == NULL) return;
// clobber existing structure on right side of tree
destroy(n->right);
// replace it with a mirror image copy
n->right = copy_mirror(n->left);
// recurse on the middle branch of the tree
make_symmetric(n->middle);
}
```

TREE NAVIGATION (In order of consideration)

Pre-Traversal: Root Left Right

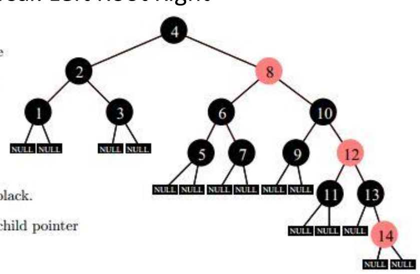
Post-Traversal: Left Right Root

Mid-Traversal: Left Root Right

18.6 Red-Black Trees

In addition to the binary search tree properties, the following red-black tree properties are maintained throughout all modifications to the data structure:

1. Each node is either red or black.
2. The NULL child pointers are black.
3. Both children of every red node are black. Thus, the parent of a red node must also be black.
4. All paths from a particular node to a NULL child pointer contain the same number of black nodes.



Binary Tree to Linked List:

```
template <class T>
void binarytree_to_linkedlist(DualNode<T> *root, DualNode<T>* &head, DualNode<T>* &tail) {
// base case
if (root == NULL) {
head = tail = NULL;
return; }
// temporary variables
DualNode<T> *l_head, *l_tail, *r_head, *r_tail;
// recursive calls
binarytree_to_linkedlist(root->leftprev,l_head,l_tail);
binarytree_to_linkedlist(root->rightnext,r_head,r_tail);
// the root comes first in prefix traversal
head = root;
head->leftprev = NULL;
// after that comes the left tree (if it exists)
if (l_head == NULL) {
l_tail = head;
} else {
head->rightnext = l_head;
l_head->leftprev = head; }
// then the right tree
// make sure the tail is set appropriately!
if (r_head == NULL) {
tail = l_tail;
} else {
l_tail->rightnext = r_head;
r_head->leftprev = l_tail;
tail = r_tail;
} }
```

Swivel:

```
template <class T>
void left_swivel(Node<T>* &input) {
assert (input != NULL && input->left != NULL);
Node<T> *orig = input;
Node<T> *repl = input->left;
Node<T> *parent = input->parent;
Node<T> *mid = input->left->right;
input = repl;
repl->parent = parent;
orig->parent = repl;
repl->right = orig;
orig->left = mid;
if (mid != NULL) mid->parent = orig;
}
```

Sorting with a Set:

```
std::set<int> data;
int num;
// read in the data, store in a set
for (int i = 0; i < n; i++) {
std::cin >> num;
data.insert(num);
}
// output directly from the set (will be sorted!)
std::set<int>::iterator itr = data.end();
while (itr != data.begin()) {
itr--;
std::cout << *itr << " "; }
std::cout << std::endl;
```