|  |  |
| --- | --- |
| **Destructor Prototype:** ~Table();  **Function:**  template <class T> void Table<T>::~Table() {  for (unsigned int j = 0; j < rows; j++) {  delete [] values[j];  }  delete [] values;  } | **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];  }  }  } |
| 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; |
| **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. | **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! |
| **(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); |
| **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  C:\Users\drumhbT430s\Desktop\Capture2.JPG Mid-Traversal: Left Root Right | **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; |