|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Trees** | | | |
| struct tree\_node {  int data;  struct tree\_node \*left;  struct tree\_node \*right;  }; | struct tree\_node\* create\_node(int val) {  // Allocate space for the node, set the fields.  struct tree\_node\* temp;  temp = (struct tree\_node\*)malloc(sizeof(struct tree\_node));  temp->data = val;  temp->left = NULL;  temp->right = NULL;  return temp; // Return a pointer to the created node.  } | | |
| // O(log *n* 🡪 *n*)  struct tree\_node\* delete(struct tree\_node\* root, int value) {  struct tree\_node \*delnode, \*new\_del\_node, \*save\_node;  struct tree\_node \*par;  int save\_val;  delnode = findNode(root, value); // Get node to delete.  par = parent(root, delnode); // Get the parent of node.  // Case where the node to delete is a leaf node.  if (isLeaf(delnode)) {  // Deleting the only node in the tree.  if (par == NULL) {  free(root); // free the memory for the node.  return NULL;  }  // Deletes the node if it's a left child.  if (value < par->data) {  free(par->left); // Free the memory for the node.  par->left = NULL;  }  // Deletes the node if it's a right child.  else {  free(par->right); // Free the memory for the node.  par->right = NULL;  }  return root; // Return the root of the new tree.  }  // Case where the node to be deleted only has left  if (hasOnlyLeftChild(delnode)) {  // Deleting the root node of the tree.  if (par == NULL) {  save\_node = delnode->left;  free(delnode); // Free the node to delete.  return save\_node; // Return new root node.  }  // Deletes the node if it's a left child.  if (value < par->data) {  save\_node = par->left; // Save the node to delete.  par->left = par->left->left; //Readjust parent pointer.  free(save\_node); // Free memory for the deleted node.  }  // Deletes the node if it's a right child.  else {  save\_node = par->right; // Save the node to delete.  par->right = par->right->left; // Readjust parent  free(save\_node); // Free memory for the deleted node.  }  return root; // Return root of tree after the deletion.  }  // Case where the deleted node only has a right child.  if (hasOnlyRightChild(delnode)) {  // Node to delete is the root node.  if (par == NULL) {  save\_node = delnode->right;  free(delnode);  return save\_node;  }  // Deletes the node if it is a left child.  if (value < par->data) {  save\_node = par->left;  par->left = par->left->right;  free(save\_node);  }  // Deletes the node if it is a right child.  else {  save\_node = par->right;  par->right = par->right->right;  free(save\_node);  }  return root;  }  // Find the new physical node to delete.  new\_del\_node = minVal(delnode->right);  save\_val = new\_del\_node->data;  delete(root, save\_val); // Now, delete the proper value.  // Restore the data to the original node to be deleted.  delnode->data = save\_val;  return root;  } | // Insertion [O(*n* 🡪 log2*n*)]  struct tree\_node\* insert(struct tree\_node \*root, struct tree\_node \*element) {  // Inserting into an empty tree.  if (root == NULL)  return element;  else {  // element should be inserted to the right.  if (element->data > root->data) {  // There is a right subtree to insert the node.  if (root->right != NULL)  root->right = insert(root->right, element);  // Place the node directly to the right of root.  else  root->right = element;  }  // element should be inserted to the left.  else {  // There is a left subtree to insert the node.  if (root->left != NULL)  root->left = insert(root->left, element);  // Place the node directly to the left of root.  else  root->left = element;  }  // Return root pointer of updated tree.  return root;  }  } | // O(log *n* 🡪 *n*)  struct tree\_node\* findNode(struct tree\_node \*current\_ptr, int value) {  // Check if there are nodes in the tree.  if (current\_ptr != NULL) {  // Found the value at the root.  if (current\_ptr->data == value)  return current\_ptr;  // Search to the left.  if (value < current\_ptr->data)  return findNode(current\_ptr->left, value);  // Search to the right.  else  return findNode(current\_ptr->right, value);  }  else  return NULL; // No node found.  } | |
| **Heaps** | | |
| struct heap {  struct lecture\*\* heaparray;  int capacity;  int size;  }; | | struct heapStruct\* initHeap() {  struct heapStruct\* h;  h = (struct heapStruct\*)(malloc(sizeof(struct heapStruct)));  h->capacity = SIZE;  h->heaparray = (int\*)malloc(sizeof(int)\*(SIZE+1));  h->size = 0;  return h;  } |
| void percolateDown(struct heapStruct \*h, int index) {  int min;  // Only try to internal nodes.  if ((2\*index+1) <= h->size) {  // Find the minimum value  min = minimum(h->heaparray[2\*index], 2\*index, h->heaparray[2\*index+1], 2\*index+1);  // If value is less than current move down.  if (h->heaparray[index] > h->heaparray[min]) {  swap(h, index, min);  // This part is recursive and allows us to continue percolating down the element.  percolateDown(h, min);  }  }  // Current element has exactly 1 left child.  else if (h->size == 2\*index) {  // Compare the current item to only child.  // No recursive call is needed since the child of this node is a leaf.  if (h->heaparray[index] > h->heaparray[2\*index])  swap(h, index, 2\*index);  }  } | | void percolateUp(struct heapStruct \*h, int index) {  // Only if node isn't the root.  if (index > 1) {  // See if our current node is smaller than its parent.  if (h->heaparray[index/2] > h->heaparray[index]) {  // Move node up one level.  Sw ap(h, index, index/2);  // Check if needs to be done again.  percolateUp(h, index/2);  }  }  } |
| // O(log *n*)  int removeMin(struct heapStruct \*h) {  int retval;  // We can only remove an element, if one exists in the heap!  if (h->size > 0) {  // Where the minimum is stored.  retval = h->heaparray[1];  // Copy last value into top slot.  h->heaparray[1] = h->heaparray[h->size];.  h->size--;  // Move down to rightful spot.  percolateDown(h, 1);  return retval;  }  // No value to return, failure with a -1.  else  return -1;  } | | // O(log *n*)  int insert(struct heapStruct \*h, int value) {  int\* temp;  int\* throwaway;  int i;  // Array is full, realloc  if (h->size == h->capacity) {  h->heaparray = (int\*)realloc(h->heaparray, sizeof(int)\*(2\*h->capacity+1));  // Realloc failed  if (h->heaparray == NULL) return 0;  // Double capacity.  h->capacity \*= 2;  }  // Adjust necessary components of h, and then move the inserted item into its appropriate location.  h->size++;  h->heaparray[h->size] = value;  percolateUp(h, h->size);  } |