|  |  |
| --- | --- |
| **Binary Search Trees**   * you can ensure a balanced tree if its self-balanced, amortized, or if the data is random. * *Traversals*   + pre order: root, left, right   void preorder (int \* root)  if (root != NULL)  print root->data  preorder(root->left)  preorder(root->right)   * + in order: left, root. right   void inorder(int\*root)  if(root!=NULL)  inorder(root->left)  print root->data  inorder(root->right)   * + post order : left, right, root   void postorder(int\*root)  if(root==NULL)  return  postorder(root->left)  postorder(root->right)  print root->data   * *Deleting a node*    + leaf node – set it to NULL and parent’s child is NULL   + node w/ 1 child:     - left: replace node w left child     - right: replace node w right child   + node w 2 children     - replace node w min value of right subtree * destructor   + post order   destructorHelper(node \*n)  if(n->left != NULL)  destructorHelper(n->left)  if(n->right != NULL)  destructorHelper(n->right)  delete n;   * Red and Black Trees   + a node is either red or black   + root is black   + every leaf node is black, empty and has NULL children   + for every node in the tree, all paths to a descendent leaf node must pass thru the same number of black nodes   **Graphs**   * complete graph: when all vertices are connected * Breadth First O(V+E)   + works well on unweighted and undirected graphs   + Traverse: use queue, run until empty, scan adjacency list of each item in Q, and add unvisited to Q. mark every checked node as visited   + Search: add distance (n->adj[x].v->dist = n->dist+1) * Depth First O(V+E)   + recursive: given a vertex, mark as visited, and call recursively w first non visited item from adj list * ADT:   + Insert vertex: allocate new vertex dynamically and add pointer to vertex vector   + add edge – undirected graph, add one in each direction to each vertex’s adj list. * Dijkstra O(ElogV)   + maintains a solved list   + vertex gets added to solved list if it is the min of all the unsolved vertices in the adj list of all vertices in solved list   + terminates as end vertex gets marked as solved   **Hash Table**   * mod of sum: sum all values in string and mod by array length * multiplicative: sum all values, multiply by a constant decimal (13/32), keep fractional part (hash = fract(hash)) and multiply by table size * collision resolution   + open addressing:     - linear probing     - quadratic probing   + chaining     - linked list     - BST   **Graph Functions**  void Graph::addVertex(string name)  {  vertex \*newVertex = new vertex;  newVertex->name = name;  vertices.push\_back(newVertex);  }  void Graph::addEdge(string v1, string v2)  {  for(int i = 0; i < vertices.size(); i++)  {  if(vertices[i]->name == v1)  {  for(int j = 0; j < vertices.size(); j++)  {  if(vertices[j]->name == v2 && i != j)  {  adjVertex e0;  e0.v = vertices[j];  vertices[i]->adj.push\_back(e0);  adjVertex e1;  e1.v = vertices[i];  vertices[j]->adj.push\_back(e1);  }  }  }  }  }  void Graph::displayEdges()  {  for(int i = 0; i < vertices.size(); i++)  {  cout << vertices[i]->name << " --> ";  for (int j = 0; j < vertices[i]->adj.size(); j++)  {  cout << vertices[i]->adj[j].v->name << " ";  }  cout << endl;  }  }  void Graph::breadthFirstTraverse(string sourceVertex)  {  vertex \*vStart;  for (int i = 0; i < vertices.size(); i++)  {  if (vertices[i]->name == sourceVertex)  {  vStart = vertices[i];  }  }  vStart->visited = true;  queue<vertex\*> q;  q.push(vStart);  cout<< "Starting vertex (root): " << vStart->name << "->";  while(!q.empty())  {  vertex \*n = q.front();  q.pop();  for(int i = 0; i < n->adj.size(); i++)  {  vertex \*temp = n->adj[i].v;  if(temp->visited == false)  {  temp->visited = true;  temp->distance = n->distance + 1;  cout << n->adj[i].v->name <<"("<< n->adj[i].v->distance <<")"<< " ";  q.push(temp);  }  }  }  }  void DFTraversal(vertex \*n)  {  n->visited = true;  for(int x = 0; x < n->adj.size(); x++ )  {    if(n->adj[x].v->visited == false)  {  n->adj[x].v->visited = true;  DFTraversal(n->adj[x].v);  }  }  } | **BST functions**  *Add node*  Node\* BST:: addNodeHelper(Node\* currNode, int data)  {  if(currNode == NULL){  return createNode(data);  }  else if(currNode->key < data){  currNode->right = addNodeHelper(currNode->right,data);  }  else if(currNode->key > data){  currNode->left = addNodeHelper(currNode->left,data);  }  return currNode;  }  void BST:: addNode(int data)  {  root = addNodeHelper(root, data);  cout<<data<<" has been added"<<endl;  }  *Delete node* (when calling it set root equal to function)  Node\* BST::getMinValueNode(Node\* currNode){  if(currNode->left == NULL){  return currNode;  }  return getMinValueNode(currNode->left);  }  Node\* BST::deleteNode(Node \*currNode, int value)  {  if(currNode == NULL)  {  return NULL;  }  else if(value < currNode->key)  {  currNode->left = deleteNode(currNode->left, value);  }  else if(value > currNode->key)  {  currNode->right = deleteNode(currNode->right, value);  }  else  {  if(currNode->left == NULL && currNode->right == NULL)  {  delete currNode;  currNode = NULL;  }  else if(currNode->left == NULL)  {  Node \*temp = currNode;  currNode = currNode->right;  delete temp;  }  else if(currNode->right == NULL)  {  Node \*temp = currNode;  currNode = currNode->left;  delete temp;  }  else  {  ///Replace with Minimum from right subtree  Node \*temp = getMinValueNode(currNode->right);  currNode->key = temp->key;  deleteNode(temp, temp->key);  }  }  return currNode;  }  *Search node*  Node\* BST::searchKeyHelper(Node\* currNode, int data){  if(currNode == NULL)  return NULL;  if(currNode->key == data)  return currNode;  if(currNode->key > data)  return searchKeyHelper(currNode->left, data);  return searchKeyHelper (currNode->right, data);  }  bool BST::searchKey(int key){  Node\* tree = searchKeyHelper(root, key);  if(tree != NULL) {  return true;  }  cout<<"Key not present in the tree"<<endl;  return false;  }  void depthFirstRecurse(vertex \*n)  {  n->visited = true;  for(int x = 0; x < (int)n->adj.size(); x++ )  {    if(n->adj[x].v->visited == false)  {  n->adj[x].v->visited = true;  cout<< n->adj[x].v->name << " --> " ;  depthFirstRecurse(n->adj[x].v);  }  }  }  void Graph::depthFirstTraversal(string sourceVertex)  {  vertex \*vStart;  for(int i = 0; i < (int)vertices.size(); i++)  {  if(vertices[i]->name == sourceVertex)  {  vStart = vertices[i];  }  }    cout<< vStart->name << " --> " ;  depthFirstRecurse(vStart);  cout << "Done ";  }  bool isSolved(vector<vertex\*> solvedList, string key)  {  for(int i = 0; i < (int)solvedList.size(); i++)  {  if(solvedList[i]->name == key)  {  return true;  }  }  return false;  }  vertex\* Graph::DijkstraAlgorithm(string start, string end)  {  vertex \*vStart;  vertex \*vEnd;  for(int i = 0; i < (int)vertices.size(); i++)  {  if(vertices[i]->name == start)  {  vStart = vertices[i];  }  if(vertices[i]->name == end)  {  vEnd = vertices[i];  }  }  vector<vertex\*> solvedList;  solvedList.push\_back(vStart);  while(!isSolved(solvedList, vEnd->name))  {  int minDist = INT\_MAX;  vertex \*solvedV = NULL;  for(int i = 0; i < (int)solvedList.size(); i++)  {  vertex \*s = solvedList[i];  for(int j = 0; j < (int)s->adj.size(); j++)  {  if(!isSolved(solvedList, s->adj[j].v->name))  {  int dist = s->distance + s->adj[j].weight;  if(dist < minDist)  {  solvedV = s->adj[j].v;  minDist = dist;  s->adj[j].v->pred = s;  }  }  }  }  solvedV->distance = minDist;  solvedList.push\_back(solvedV);    }  return vEnd;  } |