|  |  |
| --- | --- |
| *  Undirected graph is connected if there is a path between every pair of vertices *  Connected, directed graph is called strongly connected *  Complete graph has an edge between every pair of vertices   unordered\_map<string, double> grades; <- hash table to map student name to grades  insert functions takes a pair of object as a parameter:  pair<string, double> p(“Some student name”, 86.5);  grades.insert(p);  can insert/update using []  **Retrieve:**  double grade = grades[studentName];  Insert/update:  grades[studentName] = 98.6;  Count:  See if an item is in a set, if not return 0  Unweighted path length is N-1, number of edges on path  Unweighted shortest-path problem: O(|E|+|V|)  Weighted shortest-path problem   * No negative edges: O(|E| log |V|) * Negative edges: O(|E|∙|V|) * Acyclic graphs: O(|E|+|V|)   Weighted graph   * Dijkstra: O(|E| log |V|) * Negative weights: O(|E|·|V|)   struct Vertex  {  List adj;  bool known;  DistType dist;  Vertex panth;};  void dijkstra(Vertex s)  {  for each vertex v{  v.dist = infinity;  v.known = false;}  s.dist = 0  for (;;){  Vertex v = smallest unknown distance vertex;  if (v == NOT\_A\_VERTEX){  break;}  v.known = true;  for each Vertex w adjacent to v  if (!w.known)  {if (v.dist + cvw < w.dist){  decrease(w.dist to v.dist + cvw);  w.path = v;}}}}  void printPath(Vertex v)  {  If(v.path != NOT\_A\_VERTEX)  {  printPath(v.path);  cout<< “to “;  }  Cout<<v;  }  Suffix trie:  The first one is a whole word.  Like :  Banana 0  Anana 1  Nana 2  Ana 3  Na 4  A 5 | Patricia tree:  Write the whole string out first  And instead of a trie tree where you make a node for every character,  Make a node of the full word, then split if they share something in common  Advantage : save more space, due to less node,  Patricia is a little bit faster because it is more compact  TRIE:  Struct node  {  Bool is\_end;  Int prefix\_count;  Struct node\* child[alphabet\_size];  } \*head;  Void init()  {  Head = new node();  Head->prefix\_count = 0;  Head->is\_end = false;  }  Void insert(string word)  {  Node\*current = head;  Current->prefix\_count++;  For(unsigned int I =0; i< word.length(); ++i)  {  Int letter = (int) word[i] – (int)’a’;  If(current-> child[letter] == NULL)  Current->child[letter] = new node();  Current-> child[letter] -> prefix\_count++;  Current = current->child[letter];  }  Current->is\_end = true;  }  Bool search(string word)  {  Node\* current = head;  For(unsigned int i=0; i< word.length(); ++i)  {  If(current->child[((int)word[i] – (int)’a’)] == NULL) return false  Current = current-> child[((int)word[i] – (int)’a’)];  }  Return current->is\_end;  }  void Trie::deleteWord(string word)  {  Node \* currentNode = root;  Node \* lastWord = root;  for (int i = 0; i < word.size(); ++i)  {  char currentChar = tolower(word.at(i));  int index = currentChar - 'a';  assert(index >= 0);  *// if the current node has the current character as one of its decendants*  if (currentNode->children[index] != NULL)  currentNode = currentNode->children[index];  *// the current node doesn't have the current character which means the word is not in the trie*  else  return;  if (i == word.size() - 1 && currentNode->end)  currentNode->end = **false**;  }  }  void alphabetize(Node \* node, string prefix = "")  {  if (node->end)  cout << prefix << endl;  for (int i = 0; i < 26; ++i)  {  if (node->children[i] != NULL)  {  string currentString = prefix + node->children[i]->value;  alphabetize(node->children[i], currentString);  }  }  } |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |





