|  |  |
| --- | --- |
| **Data Structures & Algorithms**  Diploma in IT, CSF  Year 2 (2024/25) Semester 4 | **Week 6** |
| **1-2 Hours** |
| **Tutorial 6 – Hash Tables** | |

1. Devise a *perfect* hash function for the following domains. What is the range of your function?
2. The set of integers in [-100, 100]

|  |
| --- |
| Fixed range of 200 + 1  Hash function should be h(x) = x + 100, where X is any real number between [-100, 100]  Range of hash function is [0,200] |

1. The set of 3-letter English words

|  |
| --- |
| 26 alphabet characters, 26^3 = 17576  H(word1word2word3) = (w1 x 26^2) + (w2 x 26^2) + (w3 x 26^0) |

1. The set of tic-tac-toe positions

|  |
| --- |
|  |

1. The specification of the Dictionary ADT implemented using a Hash Table with separate chaining is given below.

|  |
| --- |
| // Dictionary.h - - Specification of Dictionary ADT  #include<string>  #include<iostream>  using namespace std;  const int MAX\_SIZE = 100;  typedef string ItemType;  typedef int KeyType;  struct Node  {  KeyType key; // search key  ItemType item; // data item  Node \*next; // pointer pointing to next item  };  class Dictionary  {  private:  Node \*items[MAX\_SIZE];  int size; // number of items in the Dictionary  public:  // constructor  Dictionary();  // destructor  ~Dictionary();  int hash(KeyType key);  // add a new item with the specified key to the Dictionary  bool add(KeyType newKey, ItemType newItem);  // remove an item with the specified key in the Dictionary  void remove(KeyType key);  // get an item with the specified key in the Dictionary (retrieve)  ItemType get(KeyType key);  // check if a specified key is in the Dictionary  bool contains(KeyType key);  // check if the Dictionary is empty  bool isEmpty();  // check the size of the Dictionary  int getLength();    //------------------- Other useful functions -----------------  // display the items in the Dictionary  void print();  }; |

Implement the following operations of the Dictionary ADT

1. **int hash(KeyType key);**

|  |
| --- |
| int Dictionary::hash(KeyType key) {          return key % MAX\_SIZE;      } |

1. **bool add(KeyType key, ItemType item);**

|  |
| --- |
| // add a new item with the specified key to the Dictionary      bool Dictionary::add(KeyType newKey, ItemType newItem) {          // get the index of the new key          int index = hash(newKey);          // create a new node          Node \*newNode = new Node;          // assign the key and item to the new node          newNode->key = newKey;          // assign the item to the new node          newNode->item = newItem;          // assign the next pointer to the new node          newNode->next = nullptr;          // if the index is empty          if (items[index] == nullptr) {              // assign the new node to the index              items[index] = newNode;              size++;              return true;          } else {              // Create a pointer to the current node residing in the index              Node \*current = items[index];              // Traverse the linked list until the end              while (current != nullptr) {                  // if the key already exists                  if (current->key == newKey) {                      // delete the new node, clear the memory                      delete newNode;                      return false;                  }                  // if at the end of list, attach new node                  if (current->next == nullptr) {                      current->next = newNode;                      size++;                      return true;                  }                  // move to the next node                  current = current->next;              }          }          // If failed delete node in case of memory leak          delete newNode;          return false;      } |

1. **bool remove(KeyType key);**

|  |
| --- |
| // remove an item with the specified key in the Dictionary      void Dictionary::remove(KeyType key){          // get the index of the key          int index = hash(key);          // Get the current node          Node \*current = items[index];          // Get the previous node          Node \*previous = nullptr;          // Traverse the linked list          while (current != nullptr) {              // If statement to check if the key is found              if (current->key == key) {                  // If the previous node is null, check if it is the first node                  if (previous == nullptr) {                      // If it is the first node, assign the next node to index                      items[index] = current->next;                  } else {                      // If it is not the first node, assign the next node to the previous node                      // This changes the pointer of next for previous node, to be the same pointer of next for current node (skips over current node)                      previous->next = current->next;                  }                  delete current;                  size--;                  return;              }              // Move to the next node              previous = current;              current = current->next;          }          cerr << "Key not found" << endl;      } |