Module 5:
Dictionary ADT (Hash table)

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

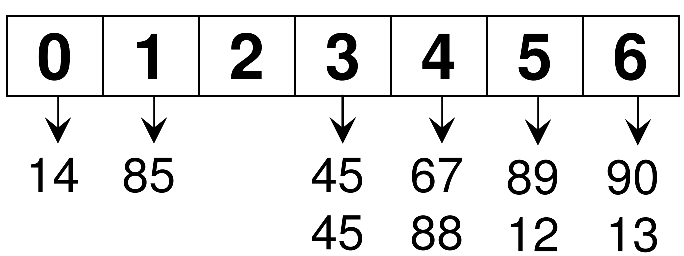
Dictionary ADT

- Models data as a collection of key-value pairs.
  - The Keys are unique
  - A value has a unique key and is accessed in the dictionary using that key
- Operations
  - Insert: The addition of a key-value pair
  - Delete: The removal of a key-value pair
  - Find: Lookup the existence of a key-value pair
  - isEmpty: Whether the dictionary is empty
- Implementations: Hash tables, Binary search trees

Hash table

- Maps the elements (values) of a collection to a unique key and stores them as key-value pairs.
- Hash table of size m (where m is the number of unique keys, ranging from 0 to m-1) uses a hash function H(v) = v mod m
- The hash value (a.k.a. hash index) for an element v is H(v) = v mod m and corresponds to one of the keys of the hash table.
- The size of the Hash table is typically a prime integer.
- Example: Consider a hash table of size 7. Its hash function is H(v) = v mod 7.
- Let an array A = {45, 67, 89, 45, 85, 12, 88, 90, 13, 14}

Value, v	45	67	89	45	85	12	88	90	13	14
H(v) = v mod 7	3	4	5	3	1	5	4	6	6	0



We will implement Hash table as an array of singly linked lists

Space-Time Tradeoff

- Note: At the worst case, there could be only one linked list in the hash table (i.e., all the elements map to the same key).
- On average, we expect the ‘n’ elements to be evenly divided across the ‘m’ keys, so that the length of a linked list is n/m. Nevertheless, for a hash table of certain size (m), ‘n’ is the only variable.
- Space complexity: $\Theta(n)$
  - For an array of ‘n’ elements, we need to allocate space for ‘n’ nodes (plus the ‘m’ head node) across the ‘m’ linked lists.
  - Since usually, n >> m, we just consider the overhead associated with storing the ‘n’ nodes
- Time complexity:
  - Insert/Delete/Lookup: O(n), we may have to traverse the entire linked list
  - isEmpty: O(m), we have to check whether each index in the Hash table has an empty linked list or not.

Example: Number of Comparisons

Array, A = {45, 23, 11, 78, 89, 44, 22, 28, 41, 30}
H(v) = v mod 7

Hash table

0	1	2	3	4	5	6	Successful Search, # comparisons
28	78	23	45	11	89	41	1
	22	44					2
		30					3

Average Number of Comparisons for a Successful Search (Hash table)

$$= \frac{(7*1) + (2*2) + (1*3)}{10} = \frac{14}{10} = \mathbf{1.4}$$

Worst Case Number of Comparisons for a Successful Search (Hash table) = 3

Worst Case Number of Comparisons for an Unsuccessful Search (Hash table) = 3

Example: Number of Comparisons

Array, A = {45, 23, 11, 78, 89, 44, 22, 28, 41, 30}
H(v) = v mod 7

Hash table

0	1	2	3	4	5	6	Successful Search, # comparisons
28	78	23	45	11	89	41	1
	22	44					2
		30					3

Average Number of Comparisons for a Successful Search (Hash table)

$$= \frac{(7*1) + (2*2) + (1*3)}{10} = \frac{14}{10} = \mathbf{1.4}$$

Average Number of Comparisons for a Successful Search (Array)

$$= \frac{1 + 2 + 3 + \ldots + 10}{10} = \frac{10*11/2}{10} = \mathbf{5.5}$$

Worst Case Number of Comparisons	Hash table	Array
For a Successful Search	3	10
For an unsuccessful Search	3	10

Review of Singly Linked List

Class List C++

```
private:
        Node *headPtr;

public:
        List(){
                headPtr = new Node();
                headPtr->setNextNodePtr(0);
        }

        Node* getHeadPtr(){
                return headPtr;
        }

        bool isEmpty(){

                if (headPtr->getNextNodePtr() == 0)
                        return true;

                return false;
        }
```

Class List Java

```
private  Node headPtr;

public List(){
        headPtr = new Node();
        headPtr.setNextNodePtr(null);
}


public Node getHeadPtr(){
        return headPtr;
}

public boolean isEmpty(){

        if (headPtr.getNextNodePtr() == null)
                return true;

        return false;
}
```

insert(int) C++

```
void insert(int data){
	Node* currentNodePtr = headPtr->getNextNodePtr();
	Node* prevNodePtr = headPtr;

	while (currentNodePtr != 0){
		prevNodePtr = currentNodePtr;
		currentNodePtr = currentNodePtr->getNextNodePtr();
	}

	Node* newNodePtr = new Node();
	newNodePtr->setData(data);
	newNodePtr->setNextNodePtr(0);
	prevNodePtr->setNextNodePtr(newNodePtr);
}
```

insert(int) Java

```
public void insert(int data){

	Node currentNodePtr = headPtr.getNextNodePtr();
	Node prevNodePtr = headPtr;

	while (currentNodePtr != null){
		prevNodePtr = currentNodePtr;
		currentNodePtr = currentNodePtr.getNextNodePtr();
	}

	Node newNodePtr = new Node();
	newNodePtr.setData(data);
	newNodePtr.setNextNodePtr(null);
	prevNodePtr.setNextNodePtr(newNodePtr);

}
```

deleteElement(int)
C++

```
bool deleteElement(int deleteData){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        Node* nextNodePtr = headPtr;

        while (currentNodePtr != 0){

                if (currentNodePtr->getData() == deleteData){
                        nextNodePtr = currentNodePtr->getNextNodePtr();
                        prevNodePtr->setNextNodePtr(nextNodePtr);
                        return true;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }
        return false;
}
```

deleteElement(int) Java

```
public boolean deleteElement(int data){

        Node currentNodePtr = headPtr.getNextNodePtr();
        Node prevNodePtr = headPtr;
        Node nextNodePtr = headPtr;

        while (currentNodePtr != null){

                if (currentNodePtr.getData() == data){
                        nextNodePtr = currentNodePtr.getNextNodePtr();
                        prevNodePtr.setNextNodePtr(nextNodePtr);
                        return true;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();
        }

        return false;
}
```

```
int countList(){
        Node* currentNodePtr = headPtr->getNextNodePtr();
        int numElements = 0;

        while (currentNodePtr != 0){
                numElements++;
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }
        return numElements;
}
```

C++

```
void IterativePrint(){
        Node* currentNodePtr = headPtr->getNextNodePtr();

        while (currentNodePtr != 0){
                cout << currentNodePtr->getData() << " ";
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }
        cout << endl;
}
```

```java
public int countList(){
	Node currentNodePtr = headPtr.getNextNodePtr();
	int numElements = 0;

	while (currentNodePtr != null){
		numElements++;
		currentNodePtr = currentNodePtr.getNextNodePtr();
	}

	return numElements;
}
```

```java
public void IterativePrint(){
	Node currentNodePtr = headPtr.getNextNodePtr();

	while (currentNodePtr != null){
		System.out.print(currentNodePtr.getData()+" ");
		currentNodePtr = currentNodePtr.getNextNodePtr();
	}

	System.out.println();
}
```

Java

C++

```
bool containsElement(int searchData){
        Node* currentNodePtr = headPtr->getNextNodePtr();

        while (currentNodePtr != 0){

                if (currentNodePtr->getData() == searchData)
                        return true;

                currentNodePtr = currentNodePtr->getNextNodePtr();
        }
        return false;
}
```

Java

```
public boolean containsElement(int data){
        Node currentNodePtr = headPtr.getNextNodePtr();

        while (currentNodePtr != null){

                if (currentNodePtr.getData() == data)
                        return true;

                currentNodePtr = currentNodePtr.getNextNodePtr();
        }
        return false;
}
```

Hash table Implementation (Code: 5.1)

Class Hashtable C++

```
private:
        List* listArray;
        int tableSize;

public:
        Hashtable(int size){
                tableSize = size;
                listArray = new List[size];
        }

        int getTableSize(){
                return tableSize;
        }

        void insert(int data){
                int hashIndex = data % tableSize;
                listArray[hashIndex].insert(data);
        }
```

Class Hashtable Java

```
private List[] listArray;
private int tableSize;

public Hashtable(int size){
        tableSize = size;
        listArray = new List[size];
        for (int index = 0; index < size; index++)
                listArray[index] = new List();
}

public int getTableSize(){
        return tableSize;
}

public void insert(int data){
        int hashIndex = data % tableSize;
        listArray[hashIndex].insert(data);
}
```

Class Hashtable
C++

```
void deleteElement(int data){
        int hashIndex = data % tableSize;
        while (listArray[hashIndex].deleteElement(data));
}
```

To delete all the entries for the data in the Linked List

```
bool hasElement(int data){

        int hashIndex = data % tableSize;
        return listArray[hashIndex].containsElement(data);

}

void printHashTable(){
        for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){
                cout << "Hash Index: " << hashIndex << " : " ;
                listArray[hashIndex].IterativePrint();
        }
}
```

```
void deleteElement(int data){
        int hashIndex = data % tableSize;
        //while (listArray[hashIndex].deleteElement(data));
        listArray[hashIndex].deleteElement(data);
}
```

Just to delete the first occurrence of the data in the Linked List

Class Hashtable Java

```
public void deleteElement(int data){
        int hashIndex = data % tableSize;
        while (listArray[hashIndex].deleteElement(data));
}
```

To delete all the entries for the data in the Linked List

```
public boolean hasElement(int data){
        int hashIndex = data % tableSize;
        return listArray[hashIndex].containsElement(data);
}

public void printHashTable(){
        for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){
                System.out.print("Hash Index: " + hashIndex + " : ");
                listArray[hashIndex].IterativePrint();
        }
}
```

```
void deleteElement(int data){
        int hashIndex = data % tableSize;
        //while (listArray[hashIndex].deleteElement(data));
        listArray[hashIndex].deleteElement(data);
}
```

Just to delete the first occurrence of the data in the Linked List

Sample Applications of Hash table

- To test whether a test sequence is a permutation of a given sequence
- To print the unique elements in an array
- To find the union of two linked lists

Permutation Check (Code 5.2)

- Given a sequence of integers (as a string)
  - Use string tokenizer to parse the string, extract the individual integers and store them in a hash table
- Given the test sequence of integers (also a string)
  - Use string tokenizer to parse the string and extract the individual integers
    - For each integer in the test sequence, check if it is in the hash table.
      - If so, delete it (just delete the first occurrence of the integer in the appropriate linked list of the hash table)
      - Otherwise, STOP and say the test sequence is not a permutation of the original sequence
    - If the hash table is empty after the test sequence is processed, print that the test sequence is a permutation of the original sequence; otherwise, it is not a permutation of the original.

Hashtable: isEmpty( )

Code 5.2: C++

```
bool isEmpty(){

        for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){

                if (!listArray[hashIndex].isEmpty())
                        return false;
        }
                                // If even one linked list is not empty, return false
                                // (i.e., the hash table is not empty)
        return true;            // If all the linked lists are empty, return true
}                               // (i.e., the hash table is empty)
```

Code 5.2: Java

```
public    boolean isEmpty(){

        for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){

                if (!listArray[hashIndex].isEmpty())
                        return false;
        }

        return true;
}
```

```
string integerSequence;                                   // Input the original integer sequence as a
cout << "Enter the integer sequence: ";                   // comma separated list
getline(cin, integerSequence);                            // Example: 45, 23, 12, 23, 90

string testSequence;                                      // Input the test sequence likewise
cout << "Enter the test sequence for permutation: ";      // Example: 12, 23, 45, 90, 23
getline(cin, testSequence);

int hashTableSize;
cout << "Enter the size of the hash table: ";             // Input the size of the hash table and
cin >> hashTableSize;                                     // Initialize a hash table of that size
Hashtable hashTable(hashTableSize);

char* integerArray = new char[integerSequence.length()+1];
strcpy(integerArray, integerSequence.c_str());
                                                          // Code for tokenizing the original
char* cptr = strtok(integerArray, ", ");                  // integer sequence and extracting
                                                          // the individual integers
while (cptr != 0){
        string token(cptr);
        int value = stoi(token);

        hashTable.insert(value);                          // Insert the extracted integers in the hash table

        cptr = strtok(NULL, ", ");
}

cout << endl;
```

Code 5.2 (C++)

Code 5.2 (C++)

```
hashTable.printHashTable();   // Print the contents of the hash table

char* testArray = new char[testSequence.length()+1];   // Code to string tokenize the
strcpy(testArray, testSequence.c_str());               // test integer sequence and extract
                                                       // the individual integers
char* tptr = strtok(testArray, ", ");

while (tptr != 0){
                                          // Check whether a test integer is in the
    string token(tptr);                   // hash table; if so, delete it.
    int testValue = stoi(token);          // If not in the hash table, print the test
                                          // sequence is not a permuted seq. STOP!
    if (hashTable.hasElement(testValue))
        hashTable.deleteElement(testValue);
    else{
        cout << testSequence << " is not a permuted sequence of " << integerSequence << endl;
        return 0;   // equivalent to stopping the program right away!
    }

    tptr = strtok(NULL, ", ");
  // If the hash table is empty when we finish processing the test sequence, it implies
}  // there exists an element in the original sequence for every element in the test
   // sequence and all of these elements were deleted. Hence, they are a permutation
if (hashTable.isEmpty())
    cout << testSequence << " is a permuted sequence of " << integerSequence << endl;
else
    cout << testSequence << " is not a permuted sequence of " << integerSequence << endl;
```

Code 5.2 (Java)

```
Scanner input = new Scanner(System.in);

String integerSequence;
System.out.print("Enter the integer sequence: ");
integerSequence = input.nextLine();      // Input the original integer sequence as a
                                         // comma separated list
String testSequence;                     // Example: 45, 23, 12, 23, 90
System.out.print("Enter the test sequence for permutation: ");
testSequence = input.nextLine();         // Input the test sequence likewise
                                         // Example: 12, 23, 45, 90, 23
int hashTableSize;
System.out.print("Enter the size of the hash table: ");  // Input the size of the
hashTableSize = input.nextInt();                         // hash table and
Hashtable hashTable = new Hashtable(hashTableSize);      // Initialize a hash table of
                                                         // that size

StringTokenizer stk = new StringTokenizer(integerSequence, ", ");
while (stk.hasMoreTokens()){
        int value = Integer.parseInt(stk.nextToken());
        hashTable.insert(value);         // Code for tokenizing the original
}                                        // integer sequence and extracting
                                         // the individual integers

System.out.println();    // Insert the extracted integers in the hash table
```

Code 5.2 (Java)

```
hashTable.printHashTable();     // Print the contents of the hash table

stk = new StringTokenizer(testSequence, ", ");          // Code to string tokenize the
while (stk.hasMoreTokens()){                            // test integer sequence and
        int testValue = Integer.parseInt(stk.nextToken());  // extract the individual integers
        if (hashTable.hasElement(testValue))
                hashTable.deleteElement(testValue);
        else{
         System.out.println(testSequence + " is not a permuted sequence of " + integerSequence);
                return;
        }
}
```

// equivalent to stopping
// the program right away!

// Check whether a test integer is in the
// hash table; if so, delete it.
// If not in the hash table, print the test
// sequence is not a permuted seq. STOP!

```
if (hashTable.isEmpty())
        System.out.println(testSequence +" is a permuted sequence of " + integerSequence);
else
        System.out.println(testSequence +" is not a permuted sequence of " + integerSequence);
```

// If the hash table is empty when we finish processing the test sequence, it implies
// there exists an element in the original sequence for every element in the test
// sequence and all of these elements were deleted. Hence, they are a permutation

Printing the Unique Elements (Code 5.3)

- Given an array A[0…n-1] that may have elements appearing more than once, we could use the hash table to store the unique elements and print them.
- For every element A[i], with $0 \le i \le n-1$, we store the element A[i] in the hash table the first time we come across it as well as print it.
- Hence, for every element A[i], with $0 \le i \le n-1$, we check if A[i] is already in the hash table.
  - If A[i] is not already in the hash table, it implies A[i] has not been seen before: so, we print it out as well as insert in the hash table.
  - If A[i] is already in the hash table, it implies we have already printed it out and should not be printed again.
- The time complexity of the algorithm is dependent on the time to check whether a particular element is in the hash table or not for all values of the array index. If this could be done in $\Theta(1)$ time per element, the asymptotic time complexity of the algorithm is $\Theta(n)$.

Code 5.3 (C++)

```
int numElements;
cout << "Enter the number of elements you want to store in the array: ";
cin >> numElements;        // Input the number of elements to store in the array

int maxValue;
cout << "Enter the maximum value for an element: ";
cin >> maxValue;           // Input the maximum value for an element

int hashTableSize;
cout << "Enter the size of the hash table: ";
cin >> hashTableSize;      // Input the number of indexes in the hash table

srand(time(NULL));         // Initialize the random number generator

                                   // Generate the random elements
int array[numElements];            // and store them in the array as well as
cout << "Elements generated: ";    // print them
for (int index = 0; index < numElements; index++){
        array[index] = rand() % maxValue;
        cout << array[index] << " ";
}

cout << endl;
```

```
Hashtable hashTable(hashTableSize);     // Initialize the
                                        // Hash table

for (int index = 0; index < numElements; index++){

        if (!hashTable.hasElement(array[index])){
                cout << array[index] << " ";
                hashTable.insert(array[index]);
        }
}

cout << endl;
```

// For an element array[index]: check if it is in the
// hashTable. If it is not there, it implies the element
// has not been seen before and print it as well as add
// it to the hashTable.
// If the element is already there in the hashTable,
// it implies it is the duplicate entry for the element and
// is not printed.

Code 5.3 (C++)

Code 5.3 (Java)

```
Scanner input = new Scanner(System.in);
                    // Create an object of class Scanner to get inputs from the user
int numElements;
System.out.print("Enter the number of elements you want to store in the array: ");
numElements = input.nextInt(); // Input the number of elements to store in the array

int maxValue;
System.out.print("Enter the maximum value for an element: ");
maxValue = input.nextInt();     // Input the maximum value for an element

int hashTableSize;
System.out.print("Enter the size of the hash table: ");
hashTableSize = input.nextInt(); // Input the number of indexes in the hash table

Random randGen = new Random(System.currentTimeMillis());
                                // Initialize the random number generator
int array[] = new int[numElements];                // Generate the random elements
System.out.print("Elements generated: ");    // and store them in the array as well as
for (int index = 0; index < numElements; index++){                    // print them
        array[index] = randGen.nextInt(maxValue);
        System.out.print(array[index] + " ");
}

System.out.println();
```

Code 5.3 (Java)

```
// Initialize the Hash table
Hashtable hashTable = new Hashtable(hashTableSize);

for (int index = 0; index < numElements; index++){

        if (!hashTable.hasElement(array[index])){
                System.out.print(array[index] + " ");
                hashTable.insert(array[index]);
        }
}

System.out.println();
```

// For an element array[index]: check if it is in the
// hashTable. If it is not there, it implies the element
// has not been seen before and print it as well as add
// it to the hashTable.
// If the element is already there in the hashTable,
// it implies it is the duplicate entry for the element and
// is not printed.

Find the Union of two Linked Lists (Code 5.4)

- Let L1 and L2 be the two linked lists and unionList be the union of the two lists
- unionList is initially empty and an element is to be included only once in this list (i.e., elements with duplicate entries in L1 or L2 are included only once in the unionList).
- We populate the contents of L1 in a hash table and unionList:
  - An element is added to both the hash table and unionList if the element is not in the hash table.
- We go through list L2, element by element. If an element in L2 is not in the hash table (it implies the element is not in L1 and unionList either), then it is included in the hash table as well as the unionList.

Code 5.4 (C++)

```
int numElements;
cout << "Enter the number of elements you want to store in the two lists: ";
cin >> numElements;
                    // Assume the two linked lists have the same number of elements
int maxValue; // The maximum value that could be for an element in the two lists
cout << "Enter the maximum value for an element: ";              // is also the same
cin >> maxValue;

int hashTableSize;
cout << "Enter the size of the hash table: "; // Input the number of indexes for the
cin >> hashTableSize;                         // Hash table

srand(time(NULL)); // Initialize the random number generator

List firstList;
cout << "Elements generated for the first list: ";  // generate random integers and
for (int index = 0; index < numElements; index++){  // populate the firstList
        int value = rand() % maxValue;
        firstList.insert(value);
}
firstList.IterativePrint();

List secondList;
cout << "Elements generated for the second list: "; // generate random integers and
for (int index = 0; index < numElements; index++){  // populate the secondList
        int value = rand() % maxValue;
        secondList.insert(value);
}
secondList.IterativePrint();
```

Code 5.4 (C++)

```
List unionList;      // The unionList will have only unique
Hashtable hashTable(hashTableSize);                 // elements

for (int index = 0; index < numElements; index++){
        int value = firstList.read(index);
                                              // Scan through the firstList and
        if (!hashTable.hasElement(value)){    // insert an element in both the
                hashTable.insert(value);      // hash table and unionList if the
                unionList.insert(value);      // element is not already in
        }                                     // the hash table
}

for (int index = 0; index < numElements; index++){
        int value = secondList.read(index);   // Scan through the secondList
                                              // If an element in the secondList
        if (!hashTable.hasElement(value)){    // is not in the hash table, it implies
                hashTable.insert(value);      // the element is not in the unionList
                unionList.insert(value);      // either and hence add the element
        }                                     // to both the unionList and the
}                                             // hash table

cout << "Elements in the union list: ";
unionList.IterativePrint();
```

Code 5.4 (Java)

```
Scanner input = new Scanner(System.in);

int numElements;
System.out.print("Enter the number of elements you want to store in the two lists: ");
numElements = input.nextInt();      // Assume the two linked lists have the same
                                    // number of elements. The maximum value that
int maxValue;                       // could be for an element in the two lists is also
System.out.print("Enter the maximum value for an element: ");
maxValue = input.nextInt();                                        // the same
                                    // Input the number of indexes for the
int hashTableSize;                  // Hash table
System.out.print("Enter the size of the hash table: ");
hashTableSize = input.nextInt();
                          // Initialize the random number generator
Random randGen = new Random(System.currentTimeMillis());

List firstList = new List();
System.out.print("Elements generated for the first list: ");
for (int index = 0; index < numElements; index++){
        int value = randGen.nextInt(maxValue);      // generate random integers and
        firstList.insert(value);                    // populate the firstList
}
firstList.IterativePrint();

List secondList = new List();
System.out.print("Elements generated for the second list: ");
for (int index = 0; index < numElements; index++){  // generate random integers and
        int value = randGen.nextInt(maxValue);      // populate the secondList
        secondList.insert(value);
}
secondList.IterativePrint();
```

Code 5.4 (Java)

```
List unionList = new List();
Hashtable hashTable = new Hashtable(hashTableSize);
                                    // The unionList will have only unique
                                    // elements
for (int index = 0; index < numElements; index++){
        int value = firstList.read(index);

        if (!hashTable.hasElement(value)){     // Scan through the firstList and
                hashTable.insert(value);       // insert an element in both the
                unionList.insert(value);       // hash table and unionList if the
        }                                      // element is not already in
}                                              // the hash table

for (int index = 0; index < numElements; index++){
        int value = secondList.read(index);    // Scan through the secondList
                                               // If an element in the secondList
        if (!hashTable.hasElement(value)){     // is not in the hash table, it implies
                hashTable.insert(value);       // the element is not in the unionList
                unionList.insert(value);       // either and hence add the element
        }                                      // to both the unionList and the
}                                              // hash table

System.out.print("Elements in the union list: ");
unionList.IterativePrint();
```