Hash Tables

Suppose we have a list of stolen credit card numbers and we would like to rapidly check if a particular credit card number is in our list.

The best binary tree lookup (using trees or arrays) take $O(\log n)$.

If the last 4 digits of the card numbers are randomly distributed we could make an array of length 10000 and store the card number at the location indicated by the last 4 digits.

If none of the cards in our list have the same last 4 digits we can find the card (if it is in our list) by looking at the location indicated by the last 4 digits of the card we are searching for.

The average time for this lookup is $O(1)$.

The worst case can be $O(n)$ if all cards have the same last 4 digits.
Hash Tables

Hash tables generally support methods to
  insert a new item in the table
  remove an item from the table
  enumerate all of the items in the table

The item inserted in a hash table can be a key-value pair
so the hash table can serve as a mapping from keys to values
Hash Tables - Collisions

If two cards have the same last 4 digits then we say that a collision has occurred.

All hash table implementations must deal with collisions.

When a collision occurs, something special must be done to resolve the conflict.

Techniques for dealing with collisions include:

- Open hashing
- Double hashing
- Chained hashing

Often a prime number is chosen for the table length to help distribute items randomly in the hash table.
Hash function

All hash tables need a hash function (two hash functions for double hashing)

A hash function takes an item and returns a number

The hash function should be designed to map items to different integers as much as possible to avoid collisions

If the item is a number, dividing by the table length and taking the remainder is frequently used as a hash function (this works well if the table size is a prime number)

If the item has many parts (e.g., an array) then the hash function should depend on all (or many) parts
Table loading - load factor

Table loading is the ratio of the number of entries to the table size

When the table loading is small there are few collisions and the item is generally found at the location specified by the hash function

When the table loading approaches 100% collisions are more frequent and performance is lower

For good performance the table size should be large

To save memory the table size should be small

A loading factor of 75% is a good balance between speed and memory
Open hashing

In open hashing we just keep looking in successive locations until we find a free spot (wrapping around to the beginning of the table if we reach the end)

This works ok when the table has few entries but when the table gets close to 100% full, large clusters grow and lookups are slower because a large cluster may have to be searched to find a given entry

deleting an entry requires either marking the entry so that a future search will keep searching past the deleted item or moving appropriate entries to fill the gap if necessary
Double hashing

In double hashing we call a second hash function to get an increment. We then step through the table by this increment until we find a free spot (wrapping around to the beginning of the table if we reach the end).

This reduces the tendency for clusters to develop and improves performance when the hash table fills up. Deleting an entry requires marking the entry so that a future search will keep searching past the deleted item.
Chained hashing

In chained hashing we put all the items at the same table location (using something like a linked list). Clusters are not an issue with chained hashing but the list at the table location must still be searched. Deleting an item just means removing the item from the list at the table entry.
Linked Lists

A linked list consists of nodes
Each node contains data and a link to the next node of the list
Many elements can be chained together by using the link
The last node of the list needs a special marker in its link
null can be used for the marker
Typical implementation

```java
public class MyList {
    private int data;
    private MyList next;

    public MyList( int data, MyList next) {
        this.data = data;
        this.next = next;
    }

    public int getData() {
        return data;
    }

    public MyList getNext() {
        return next;
    }

    public void setData( int newData ) {
        data = newData;
    }

    public void setNext( MyList newNext ) {
        next = newNext;
    }
}
```
Implementing a Stack

We just need a pointer to a linked list:

```java
public class MyStack {
    private MyList stack;

    public MyStack() {
        stack = null;
    }

    public boolean isEmpty() {
        return stack == null;
    }

    public void push( int data ) {
        stack = new MyList( data, stack );
    }

    public int pop() {
        int value = stack.getData();
        stack = stack.getNext();
        return value;
    }
} // MyStack
```
Implementing a Queue - head and tail

```java
public class MyQueue {
    private MyList front, back;

    public MyQueue() {
        front = null;
        back = null;
    }

    public boolean isEmpty() {
        return front == null;
    }

    public void enqueue( int data ) {
        MyList node = new MyList( data, null );
        if( back != null ) {
            back.setNext( node );
        } else {
            front = node;
        }
        back = node;
    }

    public int dequeue() {
        int value = front.getData();
        front = front.getNext();
        if( front == null ) {
            back = null;
        }
        return value;
    }

    public void push( int data ) {
        front = new MyList( data, front );
        if( back == null ) {
            back = front;
        }
    }
}
```

// MyQueue
```
public class MyTest {
    public static void main( String args[] ) {
        MyStack s = new MyStack();
        MyQueue q = new MyQueue();

        for( int i = 0; i < 10; i++ ) {
            s.push( i );
            q.push( i );
            q.enqueue( i );
        }

        System.out.print("Stack:");
        while( !s.isEmpty() ) {
            System.out.print( " " + s.pop() );
        }
        System.out.println();

        System.out.print("Queue:");
        while( !q.isEmpty() ) {
            System.out.print( " " + q.dequeue() );
        }
        System.out.println();
    }
} // MyTest
Processing a list
    traversing a linked list
    Step to next item
cursor = cursor.GetNext()
    Iterate through a linked list
for( cursor = head;
    cursor != null;
    cursor = cursor.getNext() )

    search for data
    search for nth
    length
    sum
    Copying a list
    Reversing a list
Processing Program 1

```java
public class MyTest1 {
    public static void main( String args[] ) {
        MyList list1
            =new MyList(1,new MyList(2,new MyList(3,null)));
        MyList list2 = copy( list1 );
        for(MyList p = list2; p != null; p=p.getNext()) {
            System.out.println( p.getData() );
        }
        MyList list3 = reverse( list1 );
        for(MyList p = list3; p != null; p=p.getNext()) {
            System.out.println( p.getData() );
        }
    }

    public static MyList copy( MyList list ) {  
        MyList head = null, tail = null;
        for(MyList p = list; p != null; p = p.getNext()){
            MyList next = new MyList(p.getData(), null);
            if (tail == null ) {
                head = next;
            } else {
                tail.setNext( next );
            }
            tail = next;
        }
        return head;
    }

    public static MyList reverse( MyList list ) {  
        MyList temp = null;
        for(MyList p = list; p != null; p = p.getNext()){
            temp = new MyList( p.getData(), temp );
        }
        return temp;
    }
} // MyTest1
```
public class MyTest2 {
    public static void main( String args[] ) {
        MyList list1
            =new MyList(1,new MyList(2,new MyList(3,null)));

        MyList list2 = smashReverse( list1 );
        for(MyList p = list2; p != null; p=p.getNext() ) {
            System.out.println( p.getData() );
        }
    }
}

displacement

public static MyList smashReverse( MyList list ) {
    MyList list1 = list, list2 = null;

    while( list1 != null ) {
        MyList temp = list1.getNext();
        list1.setNext( list2 );
        list2 = list1;
        list1 = temp;
    }
    return list2;
}

} // MyTest2
Doubly Linked Lists

A Doubly Linked List has nodes with two references - one to the next node and one to the previous node.

Trick - make the reference to a doubly linked list another list element.

This trick eliminates many special tests for the beginning or end of the list.
Arrays vs Linked Lists vs Doubly Linked Lists

Arrays are better at random access
Linked lists are better at adding and removing at a cursor
Doubly linked lists are better for a cursor that must move in both directions
Resizing can be inefficient for arrays