CHAPTER - 11

```
public interface Dictionary
{
  public Object get(Object key);
  public Object put(Object key, Object theElement);
  public Object remove(Object key);
}
```

Program 11.1 The interface Dictionary

```
public Object get(Object theKey)
{
  SortedChainNode currentNode = firstNode;

  // search for match with theKey
  while (currentNode != null &&
         currentNode.key.compareTo(theKey) < 0)
    currentNode = currentNode.next;

  // verify match
  if (currentNode != null &&
                currentNode.key.equals(theKey))
    // yes, found match
    return currentNode.element;

  // no match
  return null;
}
```

Program 11.2 The method SortedChain.get

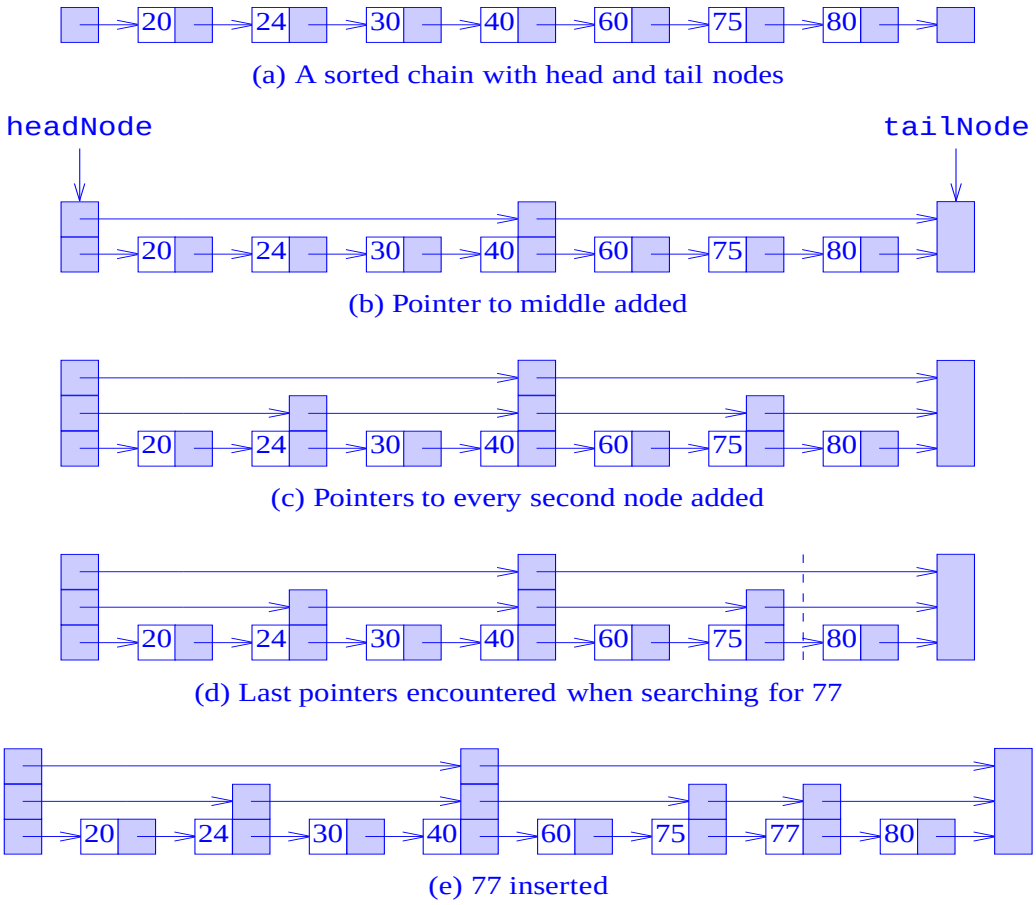

```
public Object put(Object theKey, Object theElement)
{
  SortedChainNode p = firstNode,
                  tp = null; // tp trails p

  // move tp so that theElement can be inserted
  // after tp
  while(p != null && p.key.compareTo(theKey) < 0)
  {
    tp = p;
    p = p.next;
  }

  // check if there is a matching element
  if (p != null && p.key.equals(theKey))
  {// replace old element
    Object elementToReturn = p.element;
    p.element = theElement;
    return elementToReturn;
  }
```

```
    // no match, set up node for theElement
    SortedChainNode q = new SortedChainNode(theKey,
                                theElement, p);

    // insert node just after tp
    if (tp == null) firstNode = q;
    else tp.next = q;

    size++;
    return null;
}
```

Program 11.3 The method SortedChain.put


```
public Object remove(Object theKey)
{
  SortedChainNode p = firstNode,
                  tp = null; // tp trails p
  // search for match with theKey
  while(p != null && p.key.compareTo(theKey) < 0){
    tp = p;
    p = p.next;}
  // verify match
  if (p != null && p.key.equals(theKey))
  {// found a match
    Object e = p.element; // the matching element
    // remove p from the chain
    if (tp == null)
     firstNode = p.next;//p is first node
    else tp.next = p.next;
    size--;
    return e;}
  // no matching element to remove
  return null;
}
```

Program 11.4 The method SortedChain.remove



(a) A sorted chain with head and tail nodes

(b) Pointer to middle added

(c) Pointers to every second node added

(d) Last pointers encountered when searching for 77

(e) 77 inserted

Figure 11.1 Fast searching of a sorted chain


```
protected static class SkipNode
{
  // data members
  protected Comparable key;
  protected Object element;
  protected SkipNode [] next;

  // constructor
  protected SkipNode(Object theKey,
    Object theElement, int size)
  {
    key = (Comparable) theKey;
    element = theElement;
    next = new SkipNode [size];
  }
}
```

Program 11.5 The class SkipNode


```
// probability used to decide level number
protected float prob;
// max permissible chain level
protected int maxLevel;
// max current nonempty chain
protected int levels;
// current number of elements
protected int size;
protected Comparable tailKey; // a large key
protected SkipNode headNode; // head node
protected SkipNode tailNode; // tail node
// last node seen on each level
protected SkipNode [] last;
// needed for random numbers
protected Random r;
```

Program 11.6 The data members of SkipList


```
public SkipList(Comparable largeKey,
                int maxElements, float theProb)
{
  prob = theProb;
  maxLevel = (int) Math.round(Math.log(maxElements)
                     / Math.log(1/prob)) - 1;
  // size an levels have default initial value 0
  tailKey = largeKey;

  // create head & tail nodes and last array
  headNode= new SkipNode (null,null,maxLevel+1);
  tailNode = new SkipNode (tailKey,null,0);
  last = new SkipNode [maxLevel+1];

  // headNode points to tailNode at all levels
  // initially
  for (int i = 0; i <= maxLevel; i++)
     headNode.next[i] = tailNode;

// initialize random number generator
  r = new Random();
}
```

Program 11.7 Constructor


```
public Object get(Object theKey)
{
  if (tailKey.compareTo(theKey) <= 0)
    return null; // no matching element possible

  // position p just before possible node with
  // theKey
  SkipNode p = headNode;
  for (int i = levels; i >= 0; i--)//go down levels
    // follow level i
    while (p.next[i].key.compareTo(theKey) < 0)
      p = p.next[i];// pointers

  // check if next node has theKey
  if (p.next[0].key.equals(theKey))
    return p.next[0].element;
  return null; // no matching element
}
```

Program 11.8 The method SkipList.get


```
int level()
{
  int lev = 0;
  while (r.nextFloat() <= prob)
    lev++;
  return (lev <= maxLevel) ?  lev :  maxLevel;
}
```

Program 11.9 Method to assign a level number


```
SkipNode search(Object theKey)
{
  // position p just before possible node with
  // theKey
  SkipNode p = headNode;
  for (int i = levels; i >= 0; i--)
  {
    while (p.next[i].key.compareTo(theKey) < 0)
      p = p.next[i];
    last[i] = p; // last level i node seen
  }
  return (p.next[0]);
}
```

Program 11.10 Method to search a skip list and save the last node encountered at each level


```
public Object put(Object theKey, Object theElement)
{
  // key too large
  if (tailKey.compareTo(theKey) <= 0)
    throw new IllegalArgumentException("key is
                                        too large");
  // see if element with theKey already present
  SkipNode p = search(theKey);
  if (p.key.equals(theKey))
  {// update p.element
    Object elementToReturn = p.element;
    p.element = theElement;
    return elementToReturn;
  }
  // not present, determine level for new node
  int lev = level(); // level of new node
```

```
  // fix lev to be <= levels + 1
  if (lev > levels)
  {
    lev = ++levels;
    last[lev] = headNode;
  }

  // get and insert new node just after p
  SkipNode y = new SkipNode (theKey, theElement,
                                     lev + 1);
  for (int i = 0; i <= lev; i++)
  {// insert into level i chain
    y.next[i] = last[i].next[i];
    last[i].next[i] = y;
  }
  size++;
  return null;
}
```

Program 11.11 Skip list insertion


```
public Object remove(Object theKey)
{
  if (tailKey.compareTo(theKey) <= 0) // too large
    return null;
  // see if matching element present
  SkipNode p = search(theKey);
  if (!p.key.equals(theKey)) // not present
    return null;
  // delete node from skip list
  for (int i = 0; i <= levels &&
              last[i].next[i] == p; i++)
    last[i].next[i] = p.next[i];
  // update Levels
  while (levels > 0 &&
       headNode.next[levels] == tailNode)
    levels--;
  size--;
  return p.element;
}
```

Program 11.12 Removing an element from a skip list


```
public static long threeToLong(String s)
{
  // leftmost char
  long answer = s.charAt(0);
  // shift left 16 bits and add in next char
  answer = (answer << 16) + s.charAt(1);
  // shift left 16 bits and add in next char
  return (answer << 16) + s.charAt(2);
}
```

Program 11.13 Converting a three-character string to a long integer

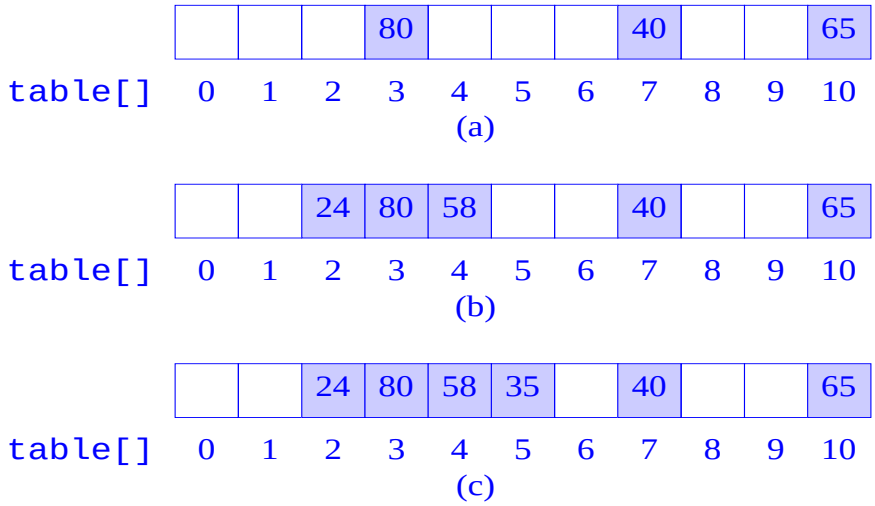


Figure 11.2 Hash tables


```
public static int integer(String s)
{
  //number of characters in s
  int length = s.length();
  int answer = 0;
  if (length % 2 == 1)
  {// length is odd
    answer = s.charAt(length - 1);
    length--;
  }

  // length is now even
  for (int i = 0; i < length; i += 2)
  {// do two characters at a time
    answer += s.charAt(i);
    answer += ((int) s.charAt(i + 1)) << 16;
  }
  return (answer < 0) ?  -answer :  answer;
}
```

Program 11.14 Converting a string into a nonunique integer


```
// data members of HashTable
protected int divisor;// hash function divisor
protected HashEntry [] table; // hash table array
protected int size;// number of elements in table

// constructor
public HashTable(int theDivisor)
{
  divisor = theDivisor;
  // allocate hash table array
  table = new HashEntry [divisor];
}
```

Program 11.15 Data members and constructor for HashTable


```
private int search(Object theKey)
{
   // home bucket
  int i = Math.abs(theKey.hashCode()) % divisor;
  int j = i; // start at home bucket
  do
  {
    if (table[j] == null ||
            table[j].key.equals(theKey))
      return j;
    j = (j + 1) % divisor; // next bucket
  } while (j != i);//returned to home bucket?

  return j; // table full
}
```

Program 11.16 The method HashTable.search

```
public Object get(Object theKey)
{
  // search the table
  int b = search(theKey);

  // see if a match ws found at table[b]
  if (table[b] == null ||
          !table[b].key.equals(theKey))
    return null;        // no match

  return table[b].element; // matching element
}
```

Program 11.17 The method HashTable.get

```
public Object put(Object theKey, Object theElement)
{
  // search the table for a matching element
  int b = search(theKey);
  // check if matching element found
  if (table[b] == null) {
    // no matching element and table not full
    table[b] = new HashEntry(theKey, theElement);
    size++;
    return null; }
  else
  {// check if duplicate or table full
    if (table[b].key.equals(theKey))
    {// duplicate, change table[b].element
      Object elementToReturn = table[b].element;
      table[b].element = theElement;
      return elementToReturn; }
    else // table is full
      throw new IllegalArgumentException("table
                                  is full");
  }
}
```

Program 11.18 Insertion into a hash table

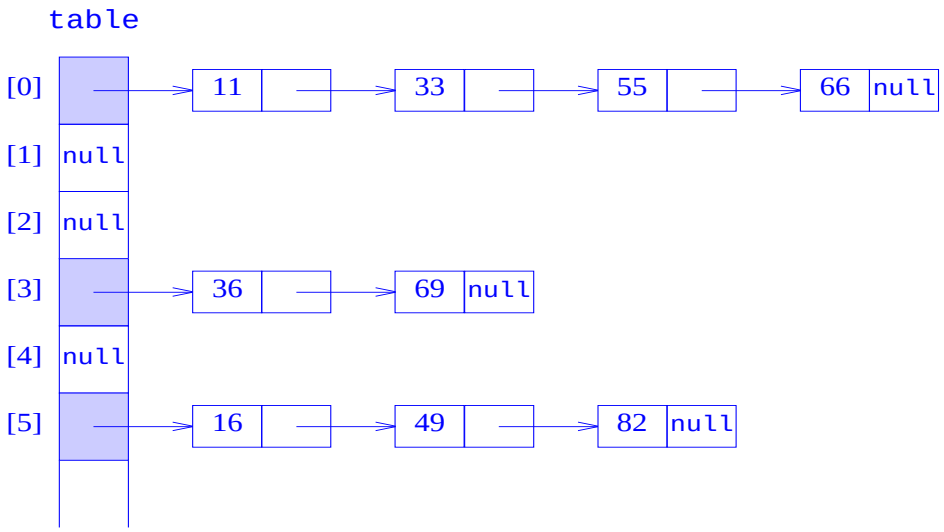
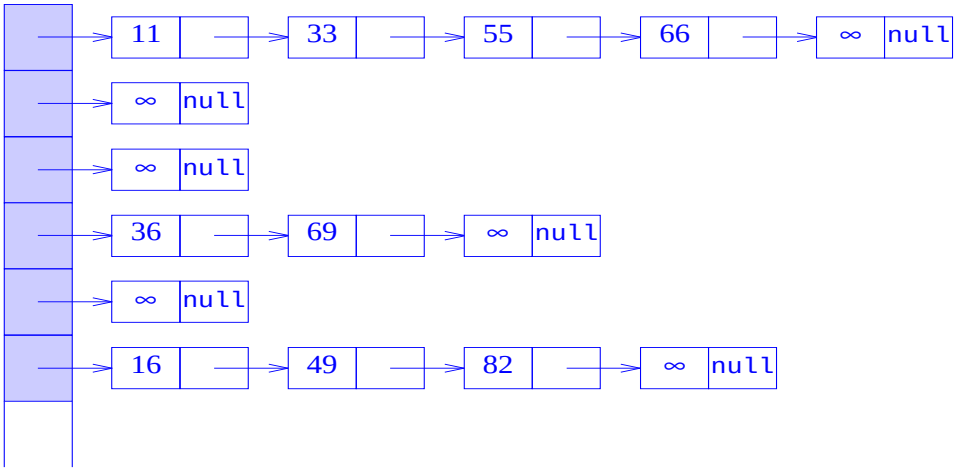


Figure 11.3 A chained hash table


```
public Object get(Object theKey)
  {return table[Math.abs(theKey.hashCode()) %
                        divisor].get(theKey);}
public Object put(Object theKey, Object theElement)
{
  // home bucket
  int b = Math.abs(theKey.hashCode()) % divisor;
  Object elementToReturn = table[b].put(theKey,
                                        theElement);
  if (elementToReturn == null) size++;//new key
  return elementToReturn;
}
public Object remove(Object theKey)
{
  Object x = table[Math.abs(theKey.hashCode())
              % divisor].remove(theKey);
  if (x != null) size--;
  return x;
}
```

Program 11.19 get, put and remove methods for a chained hash table



∞ denotes large key

Figure 11.4 Hash table with tail nodes

Figure 11.5 LZW compression


```
private static void setFiles(String [] args)
                              throws IOException
{
  String inputFile, outputFile;
  // see if file name provided
  if (args.length >= 2)
    inputFile = args[1];
  else
  {// input file name not provided, ask for it
    System.out.println("Enter name of file to
                        compress");
    MyInputStream keyboard = new MyInputStream();
    inputFile = keyboard.readString();
  }
  // establish input and output streams
  in = new BufferedInputStream(new
          FileInputStream(inputFile));
  outputFile = inputFile + ".zzz";
  out = new BufferedOutputStream(new
          FileOutputStream(outputFile));
}
```

Program 11.20 Establish input and output streams

code	0	1	2	3	4	5	6	7
key	a	b	0a	2b	1b	4b	5a	3a

Figure 11.6 Modified LZW compression dictionary for aaabbbbbbaabaaba

```
/**output 1 byte and save remaining half byte*/
private static void output(int pcode)
                throws IOException
{
  int c, d;
  if (bitsLeftOver)
  {// half byte remains from before
    d = pcode & MASK1; // right BYTE_SIZE bits
    c = (leftOver << EXCESS)+(pcode >> BYTE_SIZE);
    out.write(c);
    out.write(d);
    bitsLeftOver = false;
  }
  else
  {// no bits remain from before
    leftOver = pcode&MASK2;//right EXCESS bits
    c = pcode >> EXCESS;
    out.write(c);
    bitsLeftOver = true;
  }
}
```

Program 11.21 Output a code


```
private static void compress() throws IOException
{
  // define and initialize the code dictionary
  HashChains h = new HashChains(D);
  for (int i = 0; i < ALPHA; i++)
    // initialize code table
    h.put(new MyInteger(i), new MyInteger(i));
  int codesUsed = ALPHA;
  // input and compress
  int c = in.read(); // first byte of input
  if (c != -1)
  {// input file is not empty
    int pcode = c;
    c = in.read();// second byte
    while (c != -1)   // not at end of file
    {// process byte c
      int k = (pcode << BYTE_SIZE) + c;
      // see if code for k is in the dictionary
      MyInteger e = (MyInteger) h.get(new
                                MyInteger(k));
```

```
      if (e == null)
      {// k is not in the table
        output(pcode);
        // create new code
        if (codesUsed < MAX_CODES)
          h.put(new MyInteger((pcode<<BYTE_SIZE)+c),
              new MyInteger(codesUsed++));
        pcode = c;
      }
      else pcode = e.intValue();
      c = in.read();
    }
    // output last code(s)
    output(pcode);
    if (bitsLeftOver)
      out.write(leftOver << EXCESS);
  }
  in.close();
  out.close();
}
```

Program 11.22 LZW compressor


```
public class Compress
{
  // class data members
  // constants
  final static int D = 4099;//hash function divisor
  final static int MAX_CODES = 4096; // 2^12
  final static int BYTE_SIZE = 8;
  final static int EXCESS = 4;// 12 - ByteSize
  final static int ALPHA = 256;// 2^ByteSize
  final static int MASK1 = 255;// ALPHA - 1
  final static int MASK2 = 15;//2^EXCESS - 1
  // variables
  // code bits yet to be output
  static int leftOver;
  static boolean bitsLeftOver;
  static BufferedInputStream in;
  static BufferedOutputStream out;
```

```
  // other methods come here

  public static void main(String [] args)
                                  throws IOException
  {
    setFiles(args);
    compress();
  }
}
```

Program 11.23 Data members and main method of Compress


```
/** output the byte sequence that corresponds
to code */
private static void output(int code)
                                  throws IOException
{
  size = -1;
  while (code >= ALPHA)
  {// suffix is in the dictionary
    s[++size] = h[code].suffix;
    code = h[code].prefix;
  }
  s[++size] = code; // code < ALPHA

  // decompressed string is s[size] ...  s[0]
  for (int i = size; i >= 0; i--)
    out.write(s[i]);
}
```

Program 11.24 Compute $text$(code)

```
/** @return next code from compressed file
@return -1 if there is no next code */
private static int getCode() throws IOException
{
  int c = in.read();
  if (c == -1) return -1; // no more codes
  // see if any left over bits from before
  // if yes, concatenate with left over bits
  int code;
  if (bitsLeftOver)
    code = (leftOver << BYTE_SIZE) + c;
  else
  {//no left over bits,need more bits to complete
  // code
    int d = in.read(); // another byte
    code = (c << EXCESS) + (d >> EXCESS);
    leftOver = d & MASK; // save unused bits
  }
  bitsLeftOver = !bitsLeftOver;
  return code;
}
```

Program 11.25 Extracting codes from a compressed file


```
private static void decompress() throws IOException
{
  int codesUsed = ALPHA; // codes used so far
  s = new int [MAX_CODES];
  h = new Element [MAX_CODES];

  // input and decompress
  int pcode = getCode(), // previous code
     ccode;              // current code

  if (pcode >= 0)
  {// input file is not empty
    s[0] = pcode;   // byte for pcode
    out.write(s[0]);
    size = 0; // s[size] is first character of
           // last string output

    do
    {// get another code
      ccode = getCode();
      if (ccode < 0) break; // no more codes
```

```
        if (ccode < codesUsed)
        {// ccode is defined
          output(ccode);
          if (codesUsed < MAX_CODES)
            // create new code
            h[codesUsed++] = new Element(pcode,
                                         s[size]);
        }
        else
        {// special case, undefined code
          h[codesUsed++]=new Element(pcode,s[size]);
          output(ccode);
        }
        pcode = ccode;
     } while(true);
   }
   out.close();
   in.close();
}
```

Program 11.26 LZW decompressor


```
public class Decompress
{
  // top-level member class Element defined here

  // class data members
  // constants
  final static int MAX_CODES = 4096; // 2^12
  final static int BYTE_SIZE = 8;
  final static int EXCESS = 4;   // 12 - ByteSize
  final static int ALPHA = 256;  // 2^ByteSize
  final static int MASK = 15;//2^EXCESS-1
  // variables
  static int [] s; // used to reconstruct text
  static int size; // size of reconstructed text
  static Element [] h;// dictionary
  static int leftOver;//input bits yet to be output
  static boolean bitsLeftOver;
  static BufferedInputStream in;
  static BufferedOutputStream out;
```

```
  // other methods defined here

  public static void main(String [] args)
                                  throws IOException
  {
    setFiles(args);
    decompress();
  }
}
```

Program 11.27 Data members and method main of Decompress

