Ch11. Skip Lists and Hashing (for Dictionary)

Bird's-Eye View (0)

- Chapter 9: Stack
  - A kind of Linear list & LIFO(last-in-first-out) structure
  - Insertion and removal from one end

- Chapter 10: Queue
  - A kind of Linear list & FIFO(first-in-first-out) structure
  - Insertion and deletion occur at different ends of the linear list

- Chapter 11: Skip Lists & Hashing
  - Chains augmented with additional forward pointers
  - Popular technique for random access to records

Bird's-Eye View (1)

- Define the concept of Dictionary
- Skip list for Dictionary
  - Chains augmented with additional forward pointers
  - Employ a randomization technique
    - To determine
      - Which chain nodes are to be augmented
      - How many additional pointers are to be placed in the node
    - To search, insert, remove element: O(log n) time

- Hashing for Dictionary
  - Usage of randomization to search, insert, remove elements at 0(1) time
- Hashing Application
    - Text compression: Lampel-Ziv-Welch algorithm
    - Text decompression

Bird's-Eye View (2)

- Dictionary Implementation

Method	Worst Case			Excepted		
	Search	Insert	Removed	Search	Insert	Remove
Sorted array	θ(log n)	θ(n)	θ(n)	θ(log n)	θ(n)	θ(n)
Sorted chain	θ(n)	θ(n)	θ(n)	θ(n)	θ(n)	θ(n)
Skip lists	θ(n)	θ(n)	θ(n)	θ(log n)	θ(log n)	θ(log n)
Hash tables	θ(n)	θ(n)	θ(n)	θ(1)	θ(1)	θ(1)

  - Skip lists is better than hashing when frequently outputting all elements in sorted order or search by element rank

- Hashing in Java
  - java.util.HashTable, java.util.HashMap, and java.util.Hashset

Table of Contents

- Definition: Dictionary
- Linear List Representation
- Skip Lists Representation
- Hash Table Representation
  - Hashing concepts
  - Collision Solutions
- Hashing Application
  - Text Compression

Dictionary (1)

- A collection of pairs of the form (*k*,*e*)
  - k : a key
  - e : the element associates with the key k
  - Pairs have different keys
- Operations
  - Get the element associated with a specified key
  - Insert or put an element with a specified key
  - Delete or remove an element with a specified key
- Intuitively, dictionary is a mini database

Key	Element
db	Data Base
ds	Data Structure
ai	Artificial Intelligence

Dictionary (2)

- A dictionary with duplicates
  - Keys are not required to be distinct
  - Need to have a rule to eliminate the ambiguity
    - Get operation
      - Get any element or Get all elements
    - Remove operation
      - Remove the element specified by user or arbitrarily any one element

- Sequential access
  - Elements are retrieved 1 by 1 in an ascending order of keys

The Abstract Data Type: Dictionary

AbstractDataType *Dictionary* {

instances

collection of elements with distinct keys

operations

get(k) : return the element with key k;

put(k, x) : put the element x whose key is k into the dictionary and return the old element associated with k;

remove(k) : remove the element with key k and return it;

}

The interface: Dictionary

```
public interface Dictionary {

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

Table of Contents

- Definition: Dictionary
- Linear List Representation
- Skip Lists Representation
- Hash Table Representation
- Hashing Application
  - Text Compression

Dictionary by Linear List

- Interface LinearList {
  isEmpty(); size(); get(index); indexOf(x);
  remove(index); add(theIndex, x); output(); }
- Interface Dictionary {
  get(Object key) ;
  public Object put(Object key, Object theElement) ;
  public Object remove(Object key) ; }

- L = (e_0, e_1, e_2, ….. , e_{n-1})
  - Each e_i is a pair (key, element)
  - The e_is are in ascending order of key

- 2 kinds of representations
  - The class *SortedArrayList* as array-based
  - The class *SortedChain* as linked

Array-based Dictionary

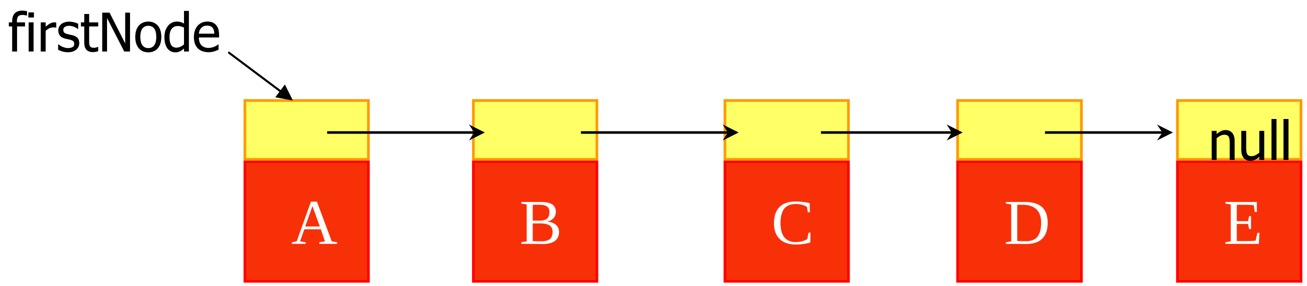
- The class SortedArrayList for array-based dictionary

A	B	C	D	E										

- Time complexity of operations
  - Get : O(log n)
    - by binary search
  - Insert : O(log n) + O(n)
    - by binary search & move at most n elements to right
  - Remove : O(log n) + O(n)
    - by binary search & move at most n elements to left

Linked-List based Dictionary

- The class SortedChain for linked-list based dictionary



- Time complexity of operations
  - Get : O(n)
  - Insert : O(n) + O(1) (put at proper place)
  - Remove : O(n)

- No binary search in a sorted chain!

get() in SortedChain for Dictionary (1)

```
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))
        return currentNode.element;
      // no match
      return null;
}
```

put() in SortedChain for Dictionary: (2)

```
/** insert an element with the specified key
  * overwrite old element if there is already an element with the given key
  * @ return old element (if any) with key theKey */
 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);
  if (tp == null)  firstNode = q; // insert node just after tp
   else tp.next = q;
   size++;
   return null;
}
```

remove() in SortedChain for Dictionary (3)

```
/** @return matching element and remove it
  * @return null if no matching element */
public Object remove(Object theKey)  {
    SortedChainNode p = firstNode,  tp = null;    // tp trails p
    while (p != null && p.key.compareTo(theKey) < 0) // search for match with theKey
           {  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;  } //end of if
   return null;  // no matching element to remove
} //end of remove()
```

Table of Contents

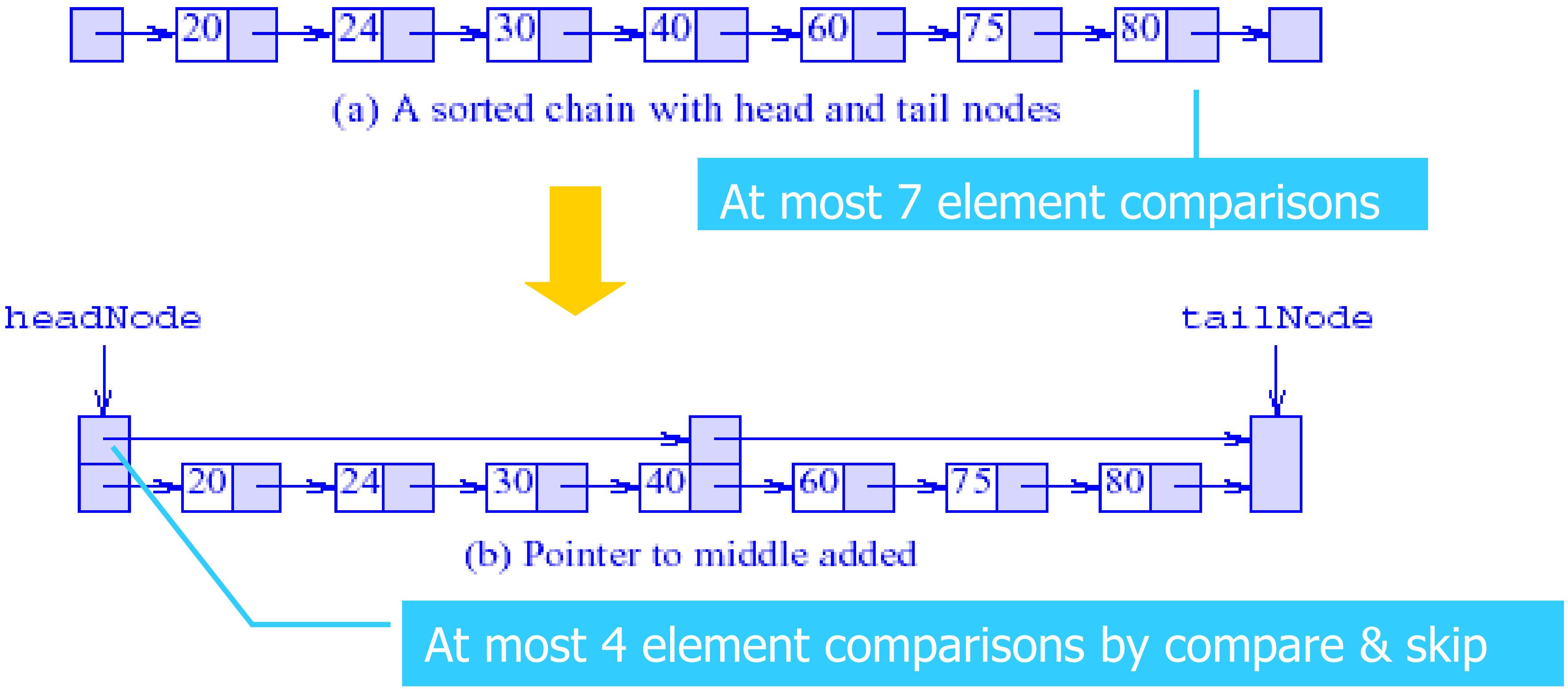
The Ideal Skip-List (1)

- In n-element dictionary which is a sorted chain, to search any element e_i
  - N element comparisons are needed
  - The number of comparisons can be reduced to n/2 + 1 with help of middle point
    - Compare with the middle point
    - If e_i < middle point, search only the left half
    - Else, search only the right half

- Adding some more data structure for a middle point can save the number of comparisons!
  - YES, Simulate the binary searching in a sorted chain with some more data structure

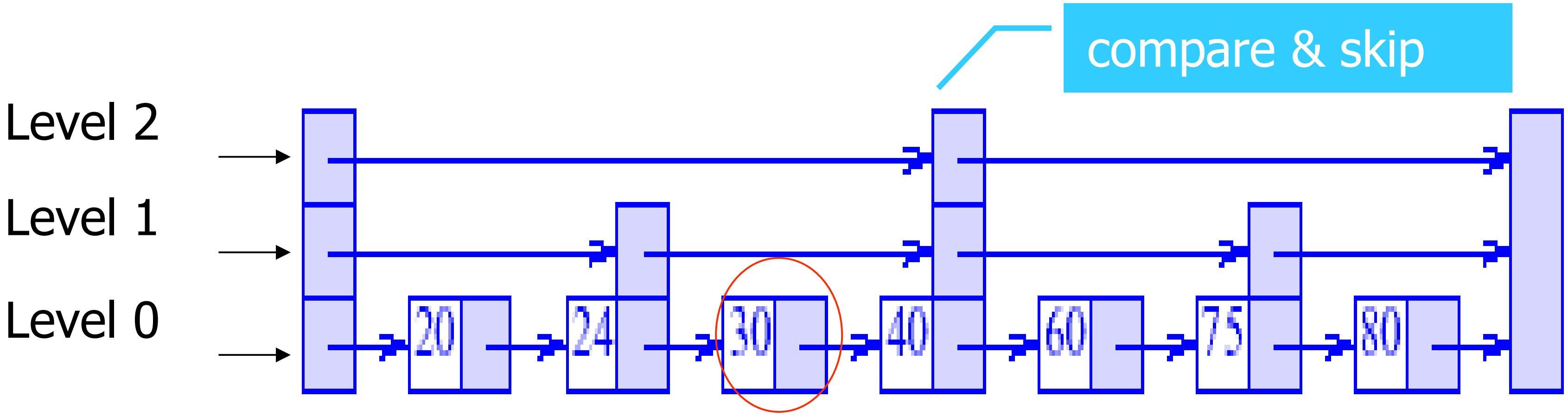
The Ideal Skip-List (2)

- Example : Consider the seven-element sorted chain



The Ideal Skip-List (3)

- Example(Cont.)
  - By keeping pointers to the middle elements of each half, we can keep on reducing the number of element comparisons



(c) Pointers to every second node added

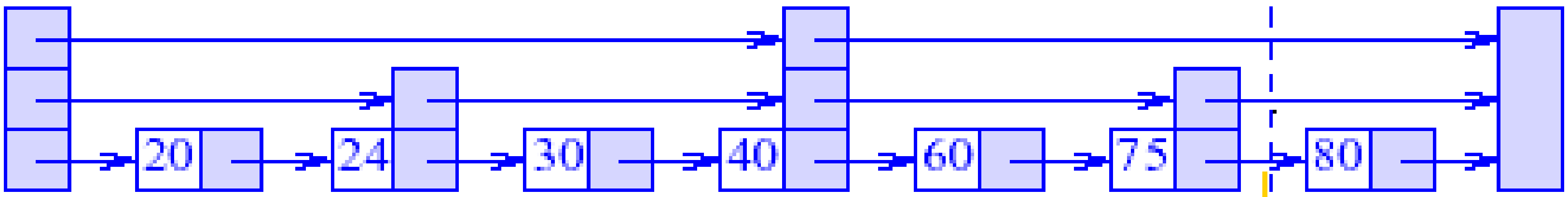
  - For example, the search value is 30

The Ideal Skip-List (4)

- Skip List
  - The level 0 chain includes all N elements
  - The level i chain
    - Includes every 2^i th element
    - Comprises a subset of the elements in the level i −1 chain
    - $N / 2^i$ elements are located in the level i
  - Legend: An element is a level i element iff it is in the chains for level 0 through i duplicately and not on the level i+1 chain
  - A regular skip list structure is the previous figure (c)
    - but we cannot maintain the ideal structure when insertion/deletion occur without doing 0(n) work

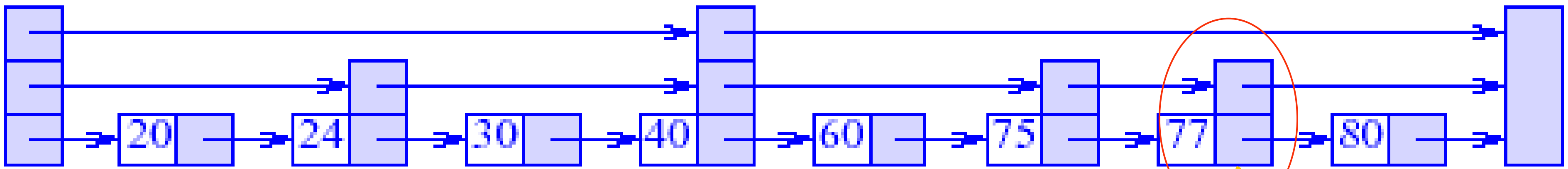
Insert in Skip-list

- Example(Cont.) : Consider the insert for an element 77



(d) Last pointers encountered when searching for 77

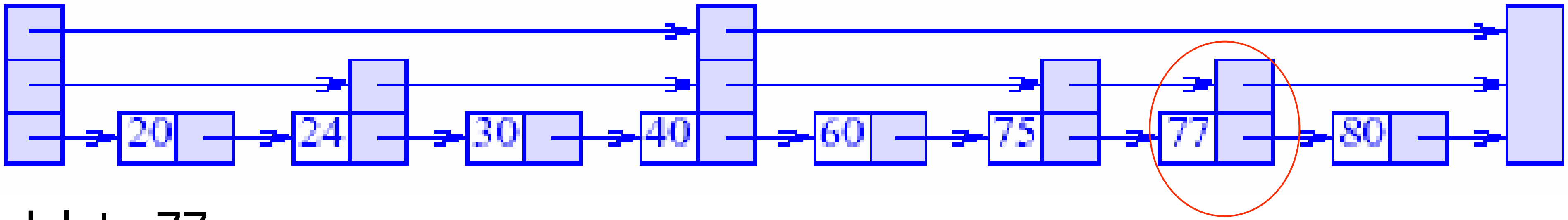
At most 3 element comparisons



(e) 77 inserted

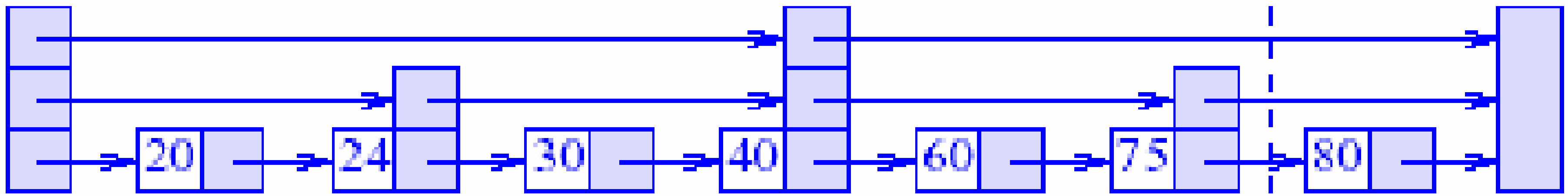
The element 77 may be in level 0 or level 1 or level 2 ➔ which one?

Delete in SkipList



delete 77

1. Search for 77
2. The encountered pointers are the level 2 in “40” and the level 1,0 in “75”
3. Level 0,1 pointers are to be changed to point to the element after 77



Assigning Levels in SkipList (1)

- We want to keep having the ideal skip-list structure, but
  - We better attempt to approximate the regular skip list structure
  - Assigning a proper level to the new element is an important issue!

- Properties of SkipList (When p elements are assigned to the next level)
  - Probability that the new element is assigned at level 0: $p^0 = 1$
  - Probability that the new element is assigned at level 1: $p^1 = 1/2$
  - Probability that the new element is assigned at level i: $p^i = (½)^i$

  - For a general p, the num of chain levels = $\lfloor \log_{1/p} n \rfloor + 1$
  - The level i chain comprises every (1/p) th element of the level i –1 chain

Assigning Levels in SkipList (2)

- One way: Level assigning by a uniform random number(URN) generator
  - URN generates real number R such that $0 \leq R \leq 1$
  - The Probability that the new element is assigned at level 1: P
  - The probability of "the element on the level i –1 is also on level i": P
- Level assigning process by URN when inserting an element
  - If R is $\leq p$, assign the new element on the level 1 chain
  - If the new R is $\leq p$, assign the new element on the level 2 chain
  - Until the new R > *p*, continue this process

- Shortcomings of URN
  - Assigned level number may be greater than $\log_{1/p} N$
    - To prevent this possibility, set an upper limit
  - Sometimes alter the level assignment of element
    - If the new element was assigned the level 9 and there are no level 3, 4, … , 8 elements prior to and following the insertion

Assigning Levels in SkipList (3)

- An alternative way of assigning level
  - Divide the range of values that the URN outputs into several segments
    - The 1st segment ⊆ 1 - 1/p of the range
      - 1/p of the whole elements go to the next level
    - The 2nd segment ⊆ 1/p X (1 – 1/p) of the range
    - And so on
  - If the random number in the *i* th segment, the inserted element is a level *i* – 1 element

1st segment	2nd	3rd	4th	…

The class SkipNode of SkipList

Head node : fields for the maximum num of level chains

```
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]; //size = i + 1 for level i node
   }
}
```

Data members of SkipList

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

Interface Comparable (1)

- Java.lang.Comparable
- The Comparable interface imposes a total ordering on the objects of each class that implements it
  - The ordering is referred to as the class's natural ordering
  - The class's compareTo method is referred to as its natural comparison method
- Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort)
  - Objects that implement this interface can be used as keys in a sorted elements in a sorted set, without the need to specify a comparator

Interface Comparable (2)

- The compareTo(Object o) method
  - The sole member of the Comparable interface, and not a member of Object
  - Compares this object with the specified object for order
  - Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object

```
public class MyInteger  implements Comparable {
  private int value;
  public MyInteger (int theValue) {value = theValue;}
  public int compareTo(Object o){
    int x = ((MyInteger)o).value;
    if (value < x) return -1;
    if (value == x) return 0;
    return 1;
  }
}
```

constructor() of SkipList

```
/* * create an empty skip list : 0(maxlevel)
   * largekey: used as key in tail node    * all elements must have a smaller key than "largekey"
   * maxElements: largest no of elements to be stored in the dictionary
   * theProb: probability that element on one level is also on the next level      */
public SkipList (Comparable largeKey, int maxElements,  float theProb) {
prob = theProb;
maxLevel = (int) Math.round( Math.log(maxElements) / Math.log(1/prob) ) - 1;
tailKey = largeKey;  // size and levels have default initial value 0
 // 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;
  r = new Random();  // initialize random number generator
 }
```

get() of SkipList

```
/** @return element with specified key & @return null if no matching element */
 public Object get(Object theKey)  {
   if (tailKey.compareTo(theKey) <= 0) return null;  // not possible

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

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

 } //end of get() function
```

level() of SkipList

- The method put() will first invoke level() to assign a level number and search() to search the skip list
- level() is using a random number generator

```
/** @return a random level number <= maxLevel */
int level()   {
  int lev = 0;
  while (r.nextFloat() <= prob)
    lev++;
  return (lev <= maxLevel) ? lev : maxLevel;
}
```

search() of SkipList

```
/** search for theKey saving last nodes seen at each level in the array
  * last @return node that might contain theKey */
 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: a set of pointers last[2], last[1], last[0]
   }
   return (p.next[0]);
 }
```

put() of SkipList (1)

```
/** insert an element with the specified key
  * overwrite old element if there is already an element with the given key
  * @return old element (if any) with key theKey
  * @throws IllegalArgumentException when theKey >= largeKey = tailKey */
 public Object put(Object theKey, Object theElement)  {
    if  (tailKey.compareTo(theKey) <= 0)    // key too large
        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
```

put() of SkipList (2)

```
int lev = level(); // level of new node
// fix lev to be less than  levels + 1
if (lev > levels) {
        lev = ++levels;
        last[lev] = headNode;
}  // get and insert a new node just after p
SkipNode y = new SkipNode (theKey, theElement, lev + 1);

// insert the new element into level i chain
for (int i = 0; i <= lev; i++) {
   y.next[i] = last[i].next[i];
   last[i].next[i] = y;
}
size++;
return null;

}
```

remove() of SkipList

```
/** @return matching element and remove it
  * @return null if no matching element */
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;

for (int i = 0; i <= levels && last[i].next[i] == p; i++) // delete node from skip list
        last[i].next[i] = p.next[i];
 while (levels > 0 && headNode.next[levels] == tailNode) // update Levels
        levels--;
 size--;
 return p.element;

} //end of remove() function
```

Other Issues of SkipList

- The codes of other methods are similar to those of the class *Chain*
  - size() / isEmpty() / elements() / iterator()
- The SkipList iterator iterator()
  - Can provide sequential access in sorted order in $\theta(1)$ time per element accessed
- Complexity
  - get(), put(), remove()
    - O(n + maxLevel) where n is the number of elements
  - Space complexity
    - Worst case space: O(n * MaxLevel) for pointers
    - On the average, the expected number of pointers

      $n\sum_i p^i = n\,(1 + p + p^2 \,..) = n * 1/(1 - p)$

Table of Contents

- Dictionaries
- Linear List Representation
- Skip Lists Representation
- Hash Table Representation
- Hashing Application
  - Text Compression

Hash Table Representation

- **Hashing Concepts**
- **Pitfalls of Hashing**
- **Good Hash Functions**
- **Collision Resolutions**
  - **Linear probing**
  - **Random probing**
  - **Hashing with Chaining**

Hashing Concepts (1)

- Use hash table to store dictionary pairs
- Use a hash function f()
  - Map keys into index in a hash table
  - Element *e* has a key k and is stored in table[*f(k)*]
- Complexity
  - To initialize an empty dictionary
    - O(*b*) time where *b* is the number of positions
  - To perform get(), put(), and remove() operation
    - $\Theta(1)$ time

A Simple Hashing Scheme

Hashing Concepts (2)

- If the range in key is so large, maintaining a table for each possible key value in key range is impractical

- Example : Consider the student records dictionary
  - There are 100 students
  - Key field is student ID with Range [100000, 999999] of Key (ex: 234966, 887654,….)
  - Suppose hash function f(k) = k – 100000
  - The length of table is 900,000: table[0, 899999]

- It doesn't make sense to use a table with 900,000 for only 100 students
- If we want to have a table with 100 slots, we need to have a hashing function which maps student IDs into table entry numbers (0..99).

Hashing Concepts (3)

- Buckets : Each position of the table
  - The number of buckets = the table length D
  - Index of hash table entries: 0 ~~ D – 1

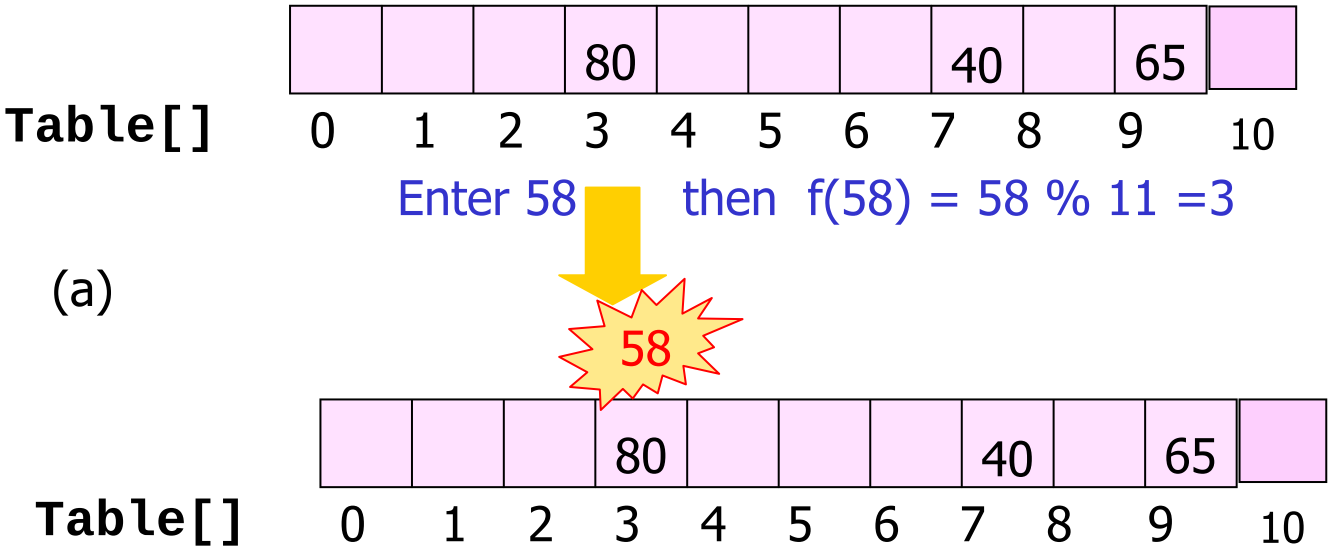
- The Division-based Hash Function: f(k) = k % D

- Home bucket : f(k) for the element whose key is *k*
  - If D =11, key =3, then the home bucket address is f(3) = (3 % 11) = 3

- The no of slots in the bucket = the no of elements that a bucket holds
  - If there are 3 slots in a bucket, only upto 3 elements can be stored in a bucket

Pitfalls of Hashing (1)

- A collision occurs whenever two different keys have the same home buckets
  - To resolve, there are linear probing, random probing, chaining, etc
  - Example : D =11, each bucket has one slot



- An overflow occurs when there is no room left in the home bucket

Pitfall of Hashing (2)

- Consider a primary key consisting of a string of 12 letters and a table with 100,000 slots.
  - Since $26^{12} >> 10^{5}$, So synonyms (collisions) are inevitable!

- If M = number of records, N = number of available slots,
  P(k) = probability of k records hashing to the same slot

then $P(k) = \binom{M}{K} \left(\frac{1}{N}\right) X \left(1-\frac{1}{N}\right) \approx \frac{f^k}{e^{k} * k!}$ where f is the loading factor M/N

- As f → 1, we know that p(0) →1/e and p(1) → 1/e.

The other (1 - 1/e) of the records must hash into (1 - 2/e) of the slots, for an average of 2.4 slot. So many synonyms!!

Good Hash Functions

- A uniform hash function distributes the approximately same number of keys from the key range per bucket
- For every key range [0, r], if r > 1 and d > 1 , f(k) = k % d is a uniform hash function if some buckets get $\lfloor r/d \rfloor$ keys and other buckets get $\lceil r/d \rceil$ keys
- The ideal choice d is a *prime number* or has *no prime factors less than 20*
- Convert nonintegral keys to integers for use by a division hash function
  - Integral type: int, long, char
  - Nonintegral type: string, double
- Object.hashCode() of Java returns an integer
  - S.hashCode() where s may be a String, Double, etc

“Integer to String” Method

```
// Convert a string into a unique integer
public static int integer (String s)  {
    int length = s.length(); //number of characters in s
    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; //shifting by 16 bits
    }
    return (answer < 0) ? – answer : answer;
}
```

Collision Resolutions

- **Linear Probing**
- **Random Probing**
- **Hashing with Chaining**
- **Rehashing**
- **….**

search() in Linear Probing

- Linear probing: search the table for the next available bucket sequentially in case of collisions
  - Regard the table as circular list

- search(k) {

  Compute f(k);

  Look at the table[f(k)];

  If the element in table[f(k)] has the key k, return the bucket address table[f(k)]

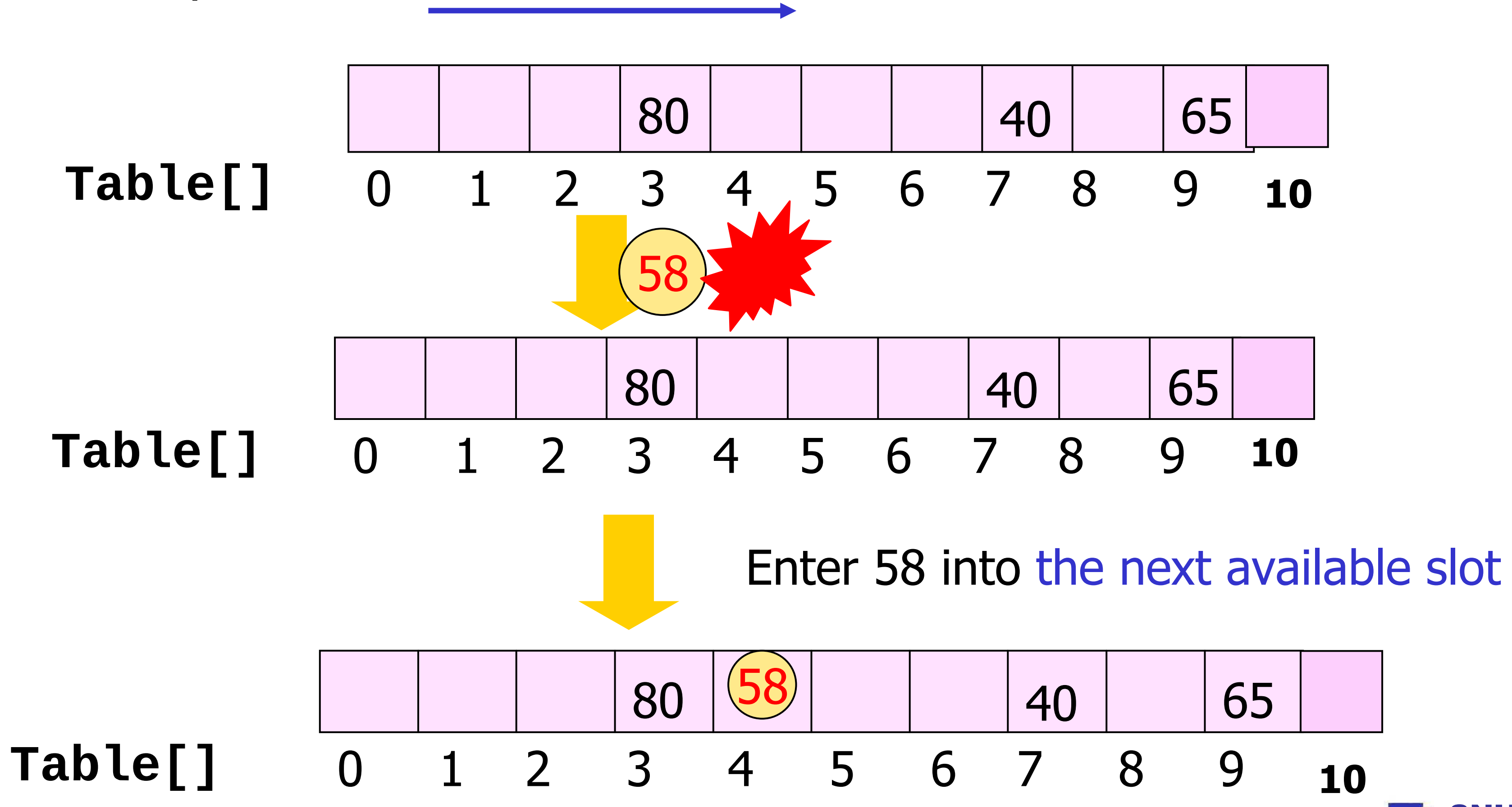
  Otherwise search the next available bucket in a circular manner

  }

- Search() is always ahead of get(), put(), remove()

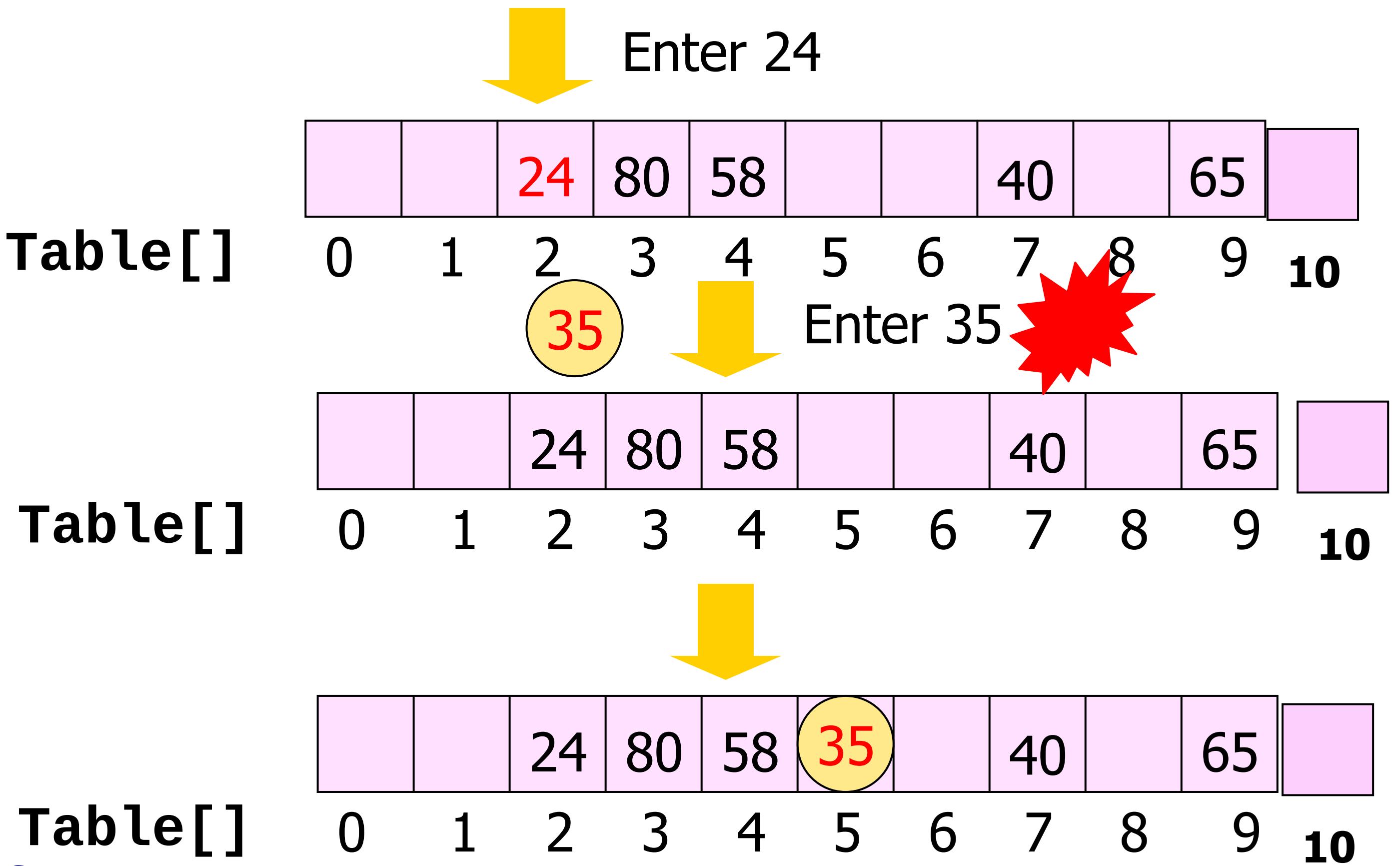
put() in Linear Probing (1)

- Example, enter 58, 24, 35, 98 in order with k % 11



put() in Linear Probing (2)

- Example: enter 58, 24, 35, 98 in order with k % 11



put() in Linear Probing (3)

- Example: enter 58, 24, 35, 98 in order with k % 11

Enter 98

Table[]	0	1	2	3	4	5	6	7	8	9	**10**
			24	80	58	35		40		65	98

- At some point we need to double the array

11	12	13…..		20	22
			…		

get() in Linear Probing

- Search(k) will return an address B with the following manner

  Compute f(k);

  Look at the table[f(k)];

  If the element in table[f(k)] has the key k,

  return the bucket address table[f(k)]

  Otherwise

  search the next available bucket in a circular manner

- Then return table[B].element

remove() in Linear Probing

- Several elements must be moved after removing an element
  - So, remove() should check the natural address of many elements

- Search steps for elements to move
  - Begin just after the bucket vacated by the removed element

    { Proceed to successive buckets;

      Check the natural address of the element;

      If the bucket is the home bucket for the element, move the element;

      until

      - **either reach to an empty bucket**
      - **or return to the bucket from which the deletion took place**

    }

HashTable with Linear Probing (1)

- HashEntry is for the pairs stored in a bucket

```
protected static class HashEntry {
    // data members
    protected Object key;
    protected Object element;
    // constructors
    private HashEntry() {}
    private HashEntry(Object theKey, Object theElement) {
        key = theKey;
        element = theElement; }
}
```

- The hash table structure
  - If the number of slots per bucket = 1, 1D array table[b] of type HashEntry
  - If the number of slots per bucket more than 1,
    - 2D array table[b][s] of type HashEntry
    - 1D array of type bucket having (key, element) pairs

HashTable with Linear Probing (2)

```
public class HashTable {
// data members of HashTable
 protected int              divisor;   // hash function divisor
 protected HashEntry []  table;      // hash table array,  one record slot for one hash bucket
 protected int              size;       // number of elements in table

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

}
```

search() with Linear Probing

```
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
  }
```

get() with Linear Probing

```
/** @return element with specified key
  * @return null if no matching element */
public Object get (Object theKey)   { // search the table
   int b = search(theKey);

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

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

put() with Linear Probing

```
/*  * insert an element with the specified key; overwrite old element if the old one has the given key
    * @throws IllegalArgumentException when the table is full
    * @return old element (if any) with key theKey */
 public Object put (Object theKey, Object theElement) {
      // search the table for a matching element
      int b = search(theKey);
      if (table[b] == null) {  // check if matching element found
         // 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 & return the old one
                Object elementToReturn = table[b].element;
                table[b].element = theElement;
                return elementToReturn; }
              else /* table is full*/    throw new IllegalArgumentException("table is full");
      } //end of else
}
```

Analysis of Linear Probing

- 2 variables for average performance when n is large
  - U_n = the average number of buckets examined during an unsuccessful search
  - S_n = the average number of buckets examined during an successful search
  - The smaller U_n & S_n, the better

- For linear probing
  - $U_n \approx ½ (1+1/(1- \alpha)^2)$
  - $S_n \approx ½ (1 + 1/(1 - \alpha))$
  - Where the loading factor α = n / b (n = no of elements, b = no of buckets)
  - Better try to keep $\alpha \leq 0.75$
  - The number of buckets should be 33% bigger than the number of elements

Random Probing

- When an overflow occurs, search for a free bucket in a random manner
  - Assign a new bucket address from a psudo-random number generator for the new element when collision happens
  - Input for the psudo-random number generator is the current address

- Linear Probing
  - Jump to the next position by "1"
- Random Probing
  - Jump to the next position by a random number

skip

Analysis of Random Probing (1)

- Probability theory
  - Let *p* be the probability that a certain event occurs
  - The expected number of independent trials needed for that event to occur is *1/p*
- The formula for U_n is derived as follows
  - When the loading density is α = n/b
    - the probability that any bucket is occupied is also same
  - The probability(p) that a bucket is empty = 1- α
  - The expected number of buckets examined
    - $U_n \approx 1/p = 1/(1 - \alpha)$

skip

Analysis of Random Probing (2)

- The formula for S_n is derived as follows
  - When the ith element is inserted,
    - the item is inserted into the empty bucket where the unsuccessful search terminates
    - The loading factor = (i -1) / b
  - The expected number of buckets examined for searching the ith element
    - $1 / (1 - ((i-1) / b))$

skip

Analysis of Random Probing (3)

- The formula for S_n is derived as follows

$$S_n \approx \frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-\frac{i-1}{b}} == -\frac{1}{\alpha}\log_e(1-\alpha)$$

- When the number of examined buckets is concerned,
  - Linear probing incurs a performance penalty relative to random probing (remember $S_n \approx$ ½ (1 + 1/(1 −α)))

- When α = 0.9, linear probing needs 50.5 bucket searches while random probing needs 10 bucket searches

skip

Analysis of Random Probing (4)

- Why do we not use random probing?
  - Computing the next random number (random probing) takes more time than examining several buckets (linear probing)
  - Random probing searches the table in a random fashion, it pays a run time penalty because of the cache effect
  - If the loading factor is close to 1, random probing is better, but linear probing is popular otherwise.

Choosing a Divisor D (1)

- $f(k) = k \% D$ and $\alpha = n/b$
- Can determine D & b using the formulas U_n & S_n
  - Determine the largest α
  - Obtain the smallest permissible value for b from *n* and α
  - Find the smallest integer for D
    - that is at least as large as this value of b
    - that is a prime or has no factors smaller than 20

- Example: Suppose we want $U_n <= 50.5$, $S_n <= 4$ & 1000 elements in linear probing
  - From $U_n = ½(1+1/(1-\alpha)^2)$, we get $\alpha <= 0.9$
  - From $S_n = ½(1 + 1/(1 - \alpha))$, we get $4 >= 0.5 + 1/(2(1-\alpha))$
  - Thus, we require $\alpha <= \min\{0.9, 6/7\} = 6/7 = n / b$ (where n = 1000)
  - Hence b should be at least $n/\alpha = 1167$ which is a suitable value for D

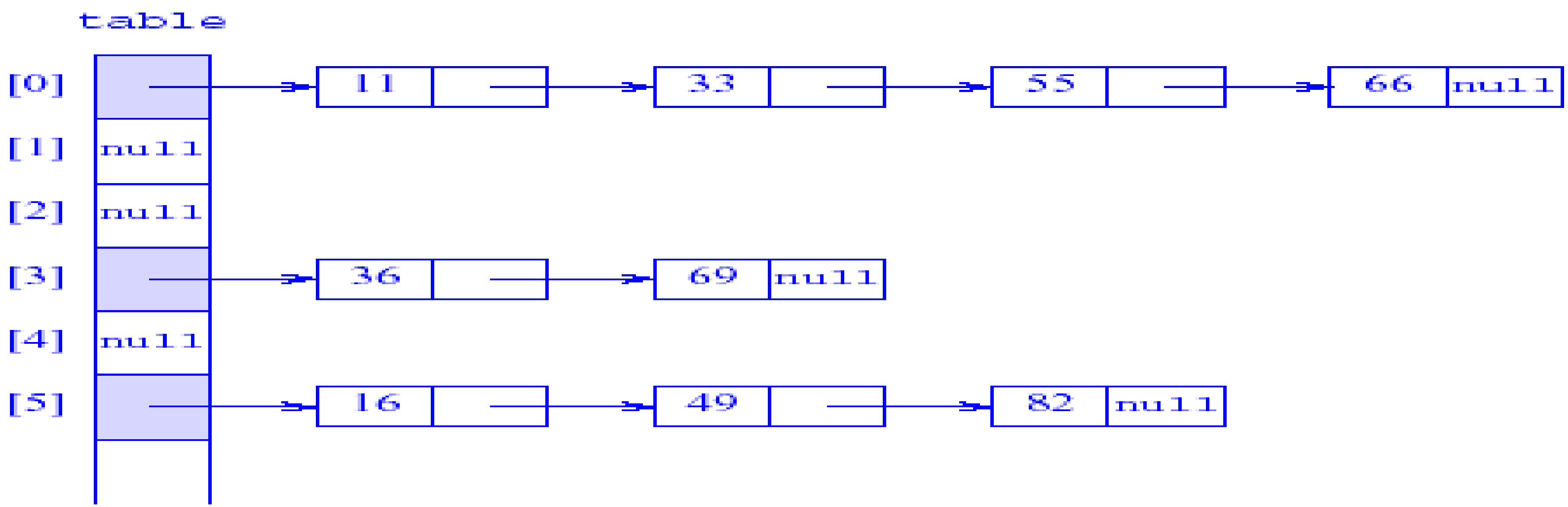
Choosing a Divisor D (2)

- Another simple way to compute D
  - L: the maximum amount of space available for the hash table
  - Find the largest $D \leq L$ that is either a prime or has no factor smaller than 20
  - Ex: Suppose 530 buckets in the hash table, 529 would be good for D & b
    - 23* 23 = 529

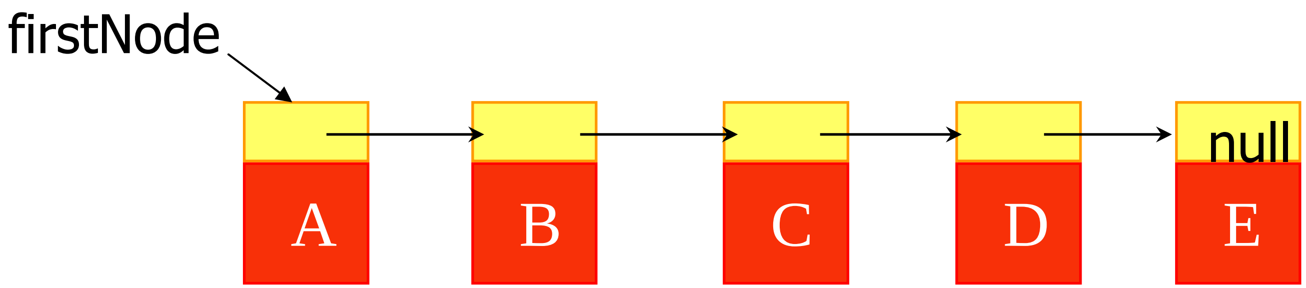
- Finding a prime number which is less than a very big number is a difficult task?

Hashing with chains

- The class HashChain maintain chains of elements that have the same home bucket
  - Each bucket has space for just a pointer "first node"
  - All elements are kept on chains in ascending order (SortedChainNode)
  - table[0:divisor-1] is a type of SortedChain class



Remember SortedChain Class



- table[] SortedChain;
- table[1].get() ➔ SortedChain.get()
- table[1].put() ➔ SortedChain.put()
- table[1].remove() ➔ SortedChain.remove()

get() in HashChain

**** The class HashChain implements a dictionary using 1D array table[o:n] of sorted chains**

table[].get()

: Compute the home bucket, k%D, for the key

: Search the chain to which this bucket points

```
/** @return element with specified key
  * @return null if no matching element */
 public Object get (Object theKey) {

    return table[Math.abs(theKey.hashCode())% divisor].get(theKey);
 }
```

put() in HashChain

table[].put()

: Verify that the table does not already have an element with the same key

```
/** insert an element with the specified key
 * overwrite old element if the element has the given key
 * @return old element (if any) with key 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;
}
```

remove() in HashChain

table[].remove()

: Access the home bucket chain

: Search this chain for an element with given key

: Delete the element

```
/** @return matching element and remove it
  * @return null if no matching element */
public Object remove(Object theKey)  {
   Object x = table[Math.abs(theKey.hashCode()) % divisor].remove(theKey);
   if (x != null) size--;
   return x;
 }
```

HashChain with a tail node (1)

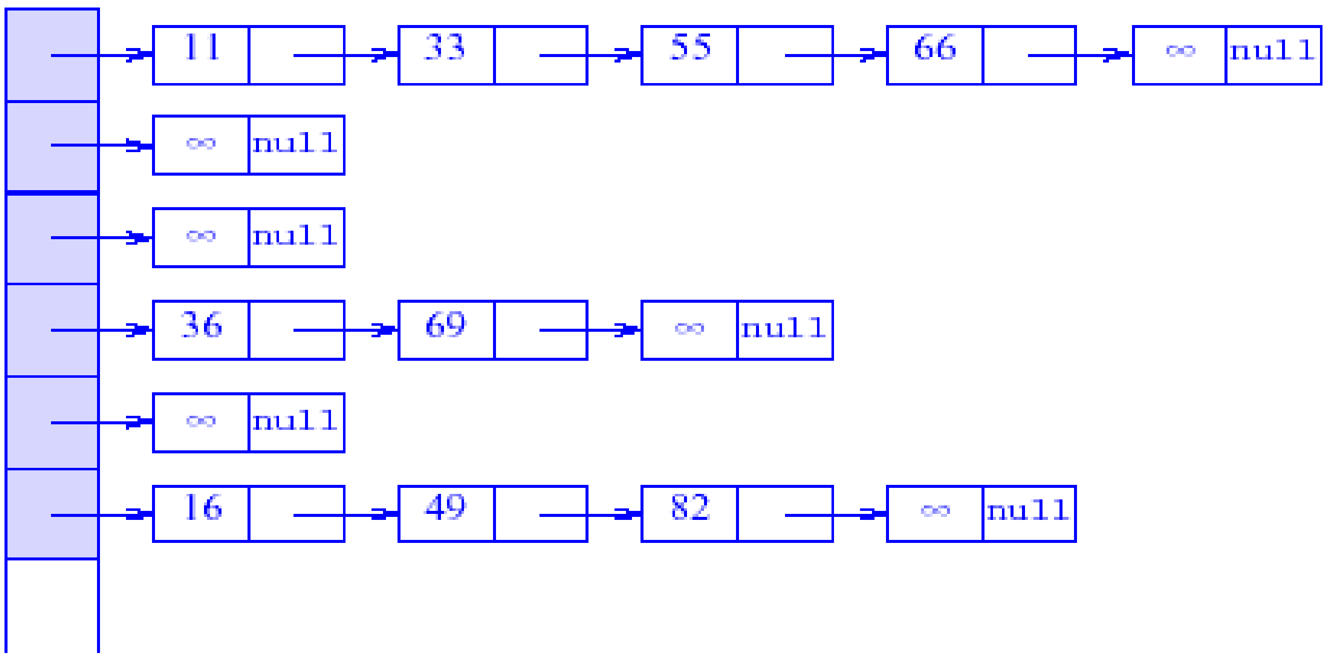
- table[].get(k): go to the next element if this element is less than the given key and the next pointer is not null

- Adding a tail node to the end of each chain can improve performance slightly
  - Put the largest key into the tail node
  - The tail node can eliminate most the checks against *null* that are used in the codes for the methods of *SortedChain*
    - table[].get(k): (currentNode != NULL)

- The constant Integer.MAX_VALUE

HashChain with a tail node (2)

- Example of hash table with tail nodes



∞ denotes large key

SNU IDB Lab.

Chaining vs. Linear Probing (1)

- Space requirements
  - Linear Probing $\leq$ Chaining
- Time complexities
  - The derivation of U_n of Chaining
    - For an i-node chain, i+1 possibilities for the range in which the search key falls
    - If each of these possibilities happens with equal probability, the average number of nodes that get examined in an unsuccessful search is

      $(1/i+1)(i+\sum^{i}_{j=1} j) = (i(i+3))/2(i+1)$ when $i \geq 1$
  - On average, the expected length of a chain = n/b = @,

    we substitute i with alpha when alpha ≥ 1

    Then, $U_n = (@(@ + 3)) / 2(@ + 1)$ in Chaining
  - Remember $U_n = ½(1+1/(1- @)^2)$ in linear hashing

Chaining vs. Linear Probing (2)

- The derivation of S_n in Chaining
  - ✓ When ith identifier is inserted, have to examine 1+(i-1)/b nodes
  - ✓ If each of n identifiers is searched for with equal probability,

    $S_n = 1/n (\sum^{n}_{i=1}\{1+(i-1)/b\}) \rightarrow 1+(@ /2)$ in Chaining
  - Remember $S_n = ½(1 + 1/(1 – @))$ in linear probing

- Comparing the above formulas, "chaining" generally examines a smaller number of buckets than "linear and random probing"

Hashing vs. Skip Lists (1)

- Both utilize a randomization process
  - Skip Lists: assign a level to an element at insertion
  - Hashing: assign a bucket to randomly distribute the bucket assignments for the different elements being inserted

- Average case operations: skip list (O(logN)) vs. hashing (O(1))

- Worst case operations

	Time complexity	Space complexity
Skip lists	Θ(n+maxLevel)	maxLevel * (n+1) for pointers
Hashing	Θ(n)	D + n for pointers

Hashing vs. Skip Lists (2)

- To output the elements in ascending order of value
  - Skip List : Linear time by going down the level 0 chain
  - Chained Hash Table :

    $\theta(D)$ (to collect) +

    $O(n\log D)$ (to combine the chains in ascending order of key)

- Other operations such as get or remove the element with largest or smallest value
  - A hash table is more expensive than a skip list

Table of Contents

Table of Contents

- Hashing Application – Text Compression
  - LZW Compression
  - Implementation of LZW Compression
  - LZW Decompression
  - Implementation of LZW Decompression
  - Performance Evaluation

LZW Compression (1)

- Character : one of the standard 256 ASCII characters which 1byte each

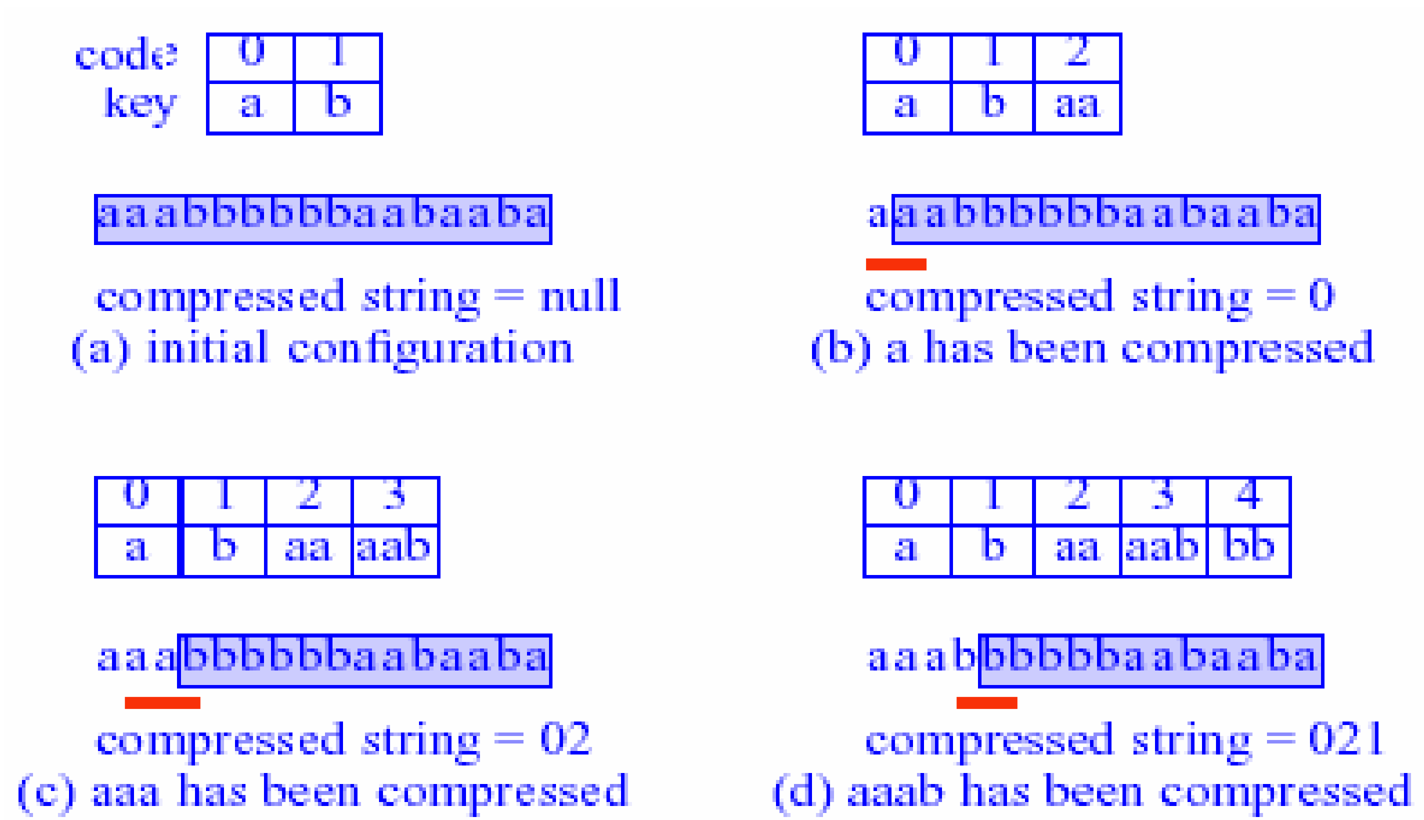
- LZW compression method (Lampel-Ziv-Welch algorithm)
  - Maps strings of text characters into numeric codes
  - The mapping is stored in a dictionary
    - Each dictionary entry has *key* and *code*
    - Key : the character string represented by code

- The LZW compressor repeatedly do this LZW rule
  - Find the longest prefix "p" of the unencoded part of S that is in the dictionary
  - Output its code
  - If there is a next character *c* in S, *pc* is assigned the next code
  - If pc is not in the dictionary, insert pc into the dictionary

- The dictionary in LZW compressor can be implemented with a hashchain
  - But it is somehow difficult to think LZW compression as Hashing application!

LZW Compression (2)

- For example, S = aaabbbbbbaabaaba is to be compressed



LZW Compression (3)

0	1	2	3	4	5
a	b	aa	aab	bb	bbb

aaabbbbbbaabaaba

compressed string = 0214

(e) aaabbb has been compressed

0	1	2	3	4	5	6
a	b	aa	aab	bb	bbb	bbba

aaabbbbbbaabaaba

compressed string = 02145

(f) aaabbbbbb has been compressed

0	1	2	3	4	5	6	7
a	b	aa	aab	bb	bbb	bbba	aaba

aaabbbbbbaabaaba

compressed string = 021453 7

(g) aaabbbbbbaab has been compressed

- String S is encoded as the string 0214537 and the code table disappears!
- Similarly the code table is reconstructed during decompression

Table of Contents

The class Compress

Class Compress {

Methods :

setFiles() : open the input and output files
output () : output a byte of the compressed file
compress() : read bytes of the input file and determine their output code
main() : a main method

}

Establish Input / Output Streams (1)

- Input: a text file
- Output: a binary file (compress file)
- If the input file name is input_File,
  - then the output file name is to be input_File.zzz

- Program: Compress.java
- Compile: javac Compress.java
- Command line
  - java Compress input_File

Establish Input / Output Streams (2)

```
/** create input and output streams */
 private static void setFiles (String [] argv) throws IOException  {
   String inputFile, outputFile;
   // see if file name provided
   if (argv.length >= 2)   inputFile = argv[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 with input buffering each disk access brings
   // in a buffer load of data rather than a single byte
   in = new BufferedInputStream ( new FileInputStream(inputFile));
   outputFile = inputFile + ".zzz";
   out = new BufferedOutputStream ( new FileOutputStream(outputFile));
}
```

Dictionary in Compress

- Modefined LZW compression dictionary for aaabbbbbbaabaaba
- Code = 12 bits Key = 20 bits = 12 bits (code) + 8 bits (character)

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

- The dictionary may be represented as a chained hash table
  - HashChains h = new HashChains(D);
    - Divisor D = 4099
- The dictionary can be an array as shown in Decompress.

output() in Compress

```
/** 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,MISK1=255
     //EXCESS = 4, BYTE_SIZE = 8
     c = (leftOver << EXCESS) + (pcode >> BYTE_SIZE);
     out.write(c);
     out.write(d);
     bitsLeftOver = false;   }  //end of if

      else{  // no bits remain from before
       leftOver = pcode & MASK2;  // right EXCESS bits, MASK2=15
       c = pcode >> EXCESS;
       out.write(c);
       bitsLeftOver = true; }
}
```

compress() in Compress (1)

```
 /** Lempel-Ziv-Welch compressor */
private static void compress() throws IOException   {
    // define and initialize the code dictionary
  HashChains h = new HashChains(D);   // HashChain Dictionary!!!!!
  for (int i = 0; i < ALPHA; i++)   // initialize code table
       h.put(new MyInteger(i), new MyInteger(i));
  int codesUsed = ALPHA;  //ALPHA = 256

  // input and compress
  int c = in.read();          // first byte of input
  if (c != -1) {  // input file is not empty
       int pcode = c;
```

compress() in Compress (2)

```
c = in.read();      // second byte
while (c != -1)    {  / / process byte c until not at the end of file
        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);
          if (codesUsed < MAX_CODES)  // create new code in the dictinary
             h.put(new MyInteger((pcode << BYTE_SIZE) + c), new MyInteger(codesUsed++));
          pcode = c;  }
      else  pcode = e.intValue();
       c = in.read();
  } //end of while
  output(pcode); // output last code(s)
  if (bitsLeftOver)  out.write(leftOver << EXCESS);
}
in.close();
out.close();
}
```

Data Members & Methods in Compress

```
public class Compress { // constants & variables
 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
 static int                    leftOver;       // code bits yet to be output
 static boolean                bitsLeftOver;
 static BufferedInputStream     in;
 static BufferedOutputStream   out;
  //other methods come here:  output(), getCode(), compress()
 public static void main(String [] argv)
                 throws IOException{ setFiles(argv); compress();}
} //end of class Compress
```

Table of Contents

- Application – Text Compression
  - LZW Compression
  - Implementation of LZW Compression
  - LZW Decompression
  - Implementation of LZW Decompression
  - Performance Evaluation

LZW Decompression (1)

- For decompression
  - Input the codes one at a time
  - Replace them by texts

- Way of the code to text mapping
  - The code assigned for single-character texts are entered into the dictionary
  - With a given code, search for an entry of dictionary
  - Replace the first code in compressed file to a single character
  - For all other codes *p*, consider
    - The case that the code *p* is in the dictionary
    - The case that the code *p* is not in the dictionary

LZW Decompression (2)

- Case when code *p* is in the dictionary
  - From dictionary, extract the text *text(p)*
    - *text(p)* : the corresponding text
  - Output it
  - If the code that precedes *p* is *q*, enter the pair (nextcode, text(q)fc(p)) into the directory
    - fc(p) : the first character of text(p)

- Case when code *p* is not in the dictionary
  - The code-to-text mapping for *p* is text(q)fc(q) where q is the code that precedes p
  - Output it
  - Enter the pair (nextcode, text(q)fc(q)) into the directory

LZW Decompression (3)

- For example, decompress the compressed coded 0214537
  1. Initialize the dictionary with the pairs (0,a), (1,b)
  2. The first code 0 ➔ output the text a
  3. Code 2 ➔ It is undefined → previous code 0, so text(2) = text(0)fc(0) = aa
     ➔ output text aa and add (2, aa) into the dictionary
  4. Code 1 ➔ output text b and add (3, text(2)fc(1)) = (3, aab) into the dictionary
  5. Code 4 ➔ It is undefined ➔ previous code 1, so text(4) = text(1)fc(1) =bb
     ➔ output text bb and add (4, bb) into the dictionary
  6. Code 5 ➔ It is undefined ➔ previous code 4, so text(5) = text(4)fc(4) = bbb
     ➔ output text bbb and add (4,bbb) into the dictionary
  7. Code 3 ➔ output aab and add (6, text(5)fc(3)) = (6,bbba) into the dictionary
  8. Code 7 ➔ It is undefined ➔ previous code 3, so text(7) = text(3)fc(3) = aaba
     ➔ add (7,aaba) into the dictionary and output aaba

The decompressed text: a+aa+b+bb+bbb+aab+aaba ➔ aaabbbbbbaabaaba

Table of Contents

Dictionary in Decompress (1)

- Implement as the class Decompress
  - Decompress.setFiles is similar to Compress.setFiles
- Dictionary Organization
  - Store the prefix code and the suffix separately as two integers
  - Array dictionary using the class Element

```
private static class Element {
    // data members
    private int prefix;
    private int suffix;

    // constructor
    private Element( int thePrefix, int theSuffix) {
      prefix = thePrefix;
      suffix = theSuffix;
    }
  }
```

Dictionary in Decompress (2)

```
/**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]);
  }
```

getCode() in Decompress

Reverse the process employed by the method output() in Compress

```
/** @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 leftover bits from before
    // if yes, concatenate with leftover bits
    int code;
    if (bitsLeftOver)   code = (leftOver << BYTE_SIZE) + c;
    else {  // no leftover 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;
  }
```

decompress() in Decompress (1)

```
/** Lempel-Ziv-Welch decompressor */
 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
```

decompress() in Decompress (2)

```
do{ ccode = getCode(); // get another code
      if (ccode < 0) break;    // no more codes
      if (ccode < codesUsed) { /* ccode is defined  */
          output(ccode);
          if (codesUsed < MAX_CODES)  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);

} //end of if pcode>=0
 out.close();
 in.close();
} //end of decompress()
```

Data Members & Methods in Decompress

```
public class Decompress  {    // constants and variables
        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
        static int [] s;                      // used to reconstruct text
        static int size;                      // size of reconstructed text
        static Element [] h;                  // array  dictionary!!!
        static int leftOver;                  // input bits yet to be output
        static boolean bitsLeftOver;
        static BufferedInputStream    in;
        static BufferedOutputStream out;
       //    other methods defined here : output(),  getCode(), decompress(),
        public static void main(String [] argv) throws IOException {
                setFiles(argv);
                decompress();    }
} //end of class Decompress
```

Table of Contents

Performance Evaluation

- Our LZW program : compress a 33772byte ASCII file to 18765 bytes
  - Compression ratio = 1.8
- Zip : compress 33772byte ASCII file to 11041 bytes
  - Compression ratio = 3.1
- Commercial compression programs couple methods such as LZW compression and other compression methods
- We should not expect a raw LZW compressor to match the performance of a commercial compressor

Summary (0)

- Chapter 9: Stack
  - A kind of Linear list & LIFO(last-in-first-out) structure
  - Insertion and removal from one end
- Chapter 10: Queue
  - A kind of Linear list & FIFO(first-in-first-out) structure
  - Insertion and deletion occur at different ends of the linear list
- Chapter 11: Skip Lists & Hashing
  - Chains augmented with additional forward pointers
  - Popular technique for random access to records

Summary (1)

- Define the concept of Dictionary
- Skip list for Dictionary
  - Chains augmented with additional forward pointers
  - Employ a randomization technique
    - To determine
      - Which chain nodes are to be augmented
      - How many additional pointers are to be placed in the node
    - To search, insert, remove element: O(log n) time
- Hashing for Dictionary
  - Usage of randomization to search, insert, remove elements at 0(1) time
- Hashing Application
    - Text compression: Lampel-Ziv-Welch algorithm
    - Text decompression

Summary (2)

- Comparison of performance (Dictionary Implementation)

Method	Worst Case			Excepted		
	Search	Insert	Removed	Search	Insert	Remove
Sorted array	θ(log n)	θ(n)	θ(n)	θ(log n)	θ(n)	θ(n)
Sorted chain	θ(n)	θ(n)	θ(n)	θ(n)	θ(n)	θ(n)
Skip lists	θ(n)	θ(n)	θ(n)	θ(log n)	θ(log n)	θ(log n)
Hash tables	θ(n)	θ(n)	θ(n)	θ(1)	θ(1)	θ(1)

  - Skip lists is better than hashing when frequently outputting all elements in sorted order or search by element rank

- Hashing in Java:

  java.util.HashTable, java.util.HashMap, and java.util.Hashset

JDK class: java.util.Hashtable

public interface ***Hashtable*** extends Dictionary {

constructors

Hashtable(): Constructs an empty hash table with initial size 11

Hashtable(int cap): Constructs an empty hash table with initial size *cap*

methods

Object get(Object key): Returns the value to which *key* is mapped

Object put(Object key, Object value): Maps *key* to *value*

}