CMP-5014Y Data Structures and Algorithms

Geoff McKeown

Tries

A data Structure for the efficient storage of dictionaries of keys

Lecture Objectives

- To introduce a data structure for representing dictionaries containing many keys with common prefixes.
- To discuss both a linked implementation and an array implementation.

Introduction

Suppose each key in K is a finite string

$$a_1 a_2 a_3 \cdots a_n$$

$a_i \in A$, $i = 1, \ldots, n$ $(n \geq 1)$, where A is an ordered set (alphabet).

Let ε denote the null string.

Definition A *trie*, t, for some $S \subset K$ is a tree; either it is empty, $\emptyset$, or it has the following properties:

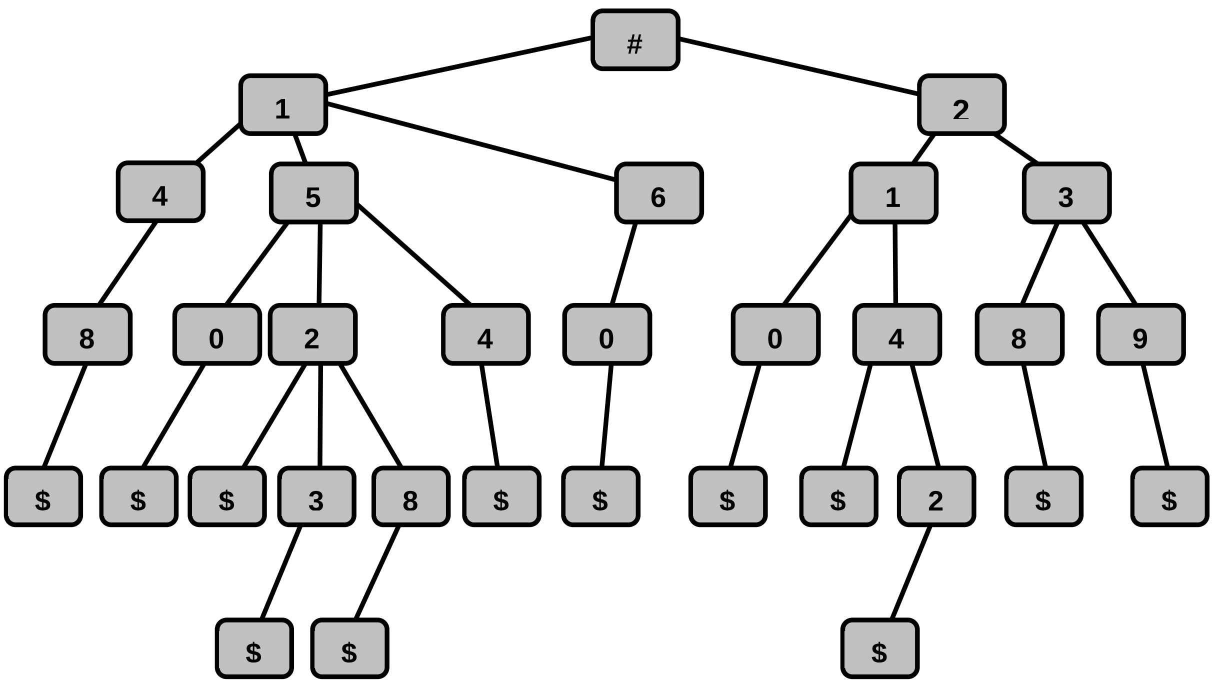
- the root contains a special symbol, $\sharp \notin A$;
- each leaf node contains a special end-of-key symbol, $\$ \notin A$;
- every other node contains an element of A such that

$$a_1 a_2 \cdots a_n \in S \text{ iff } \sharp a_1 a_2 \cdots a_n \$ \text{ is a path in } t.$$

Example

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$S = \{148, 150, 152, 1523, 1528, 154, 160, 210, 214, 2142, 238, 239\}$$



- ◇ Every path between the root and a leaf corresponds to a key in S.
- ◇ A trie is an appropriate representation when the combined length of all distinct prefixes in a set of keys, S, is small compared to the total length of all keys in S.
- ◇ Maximum number of children of a non-leaf node is $m = |A| + 1$.

ADT TRIE(A)

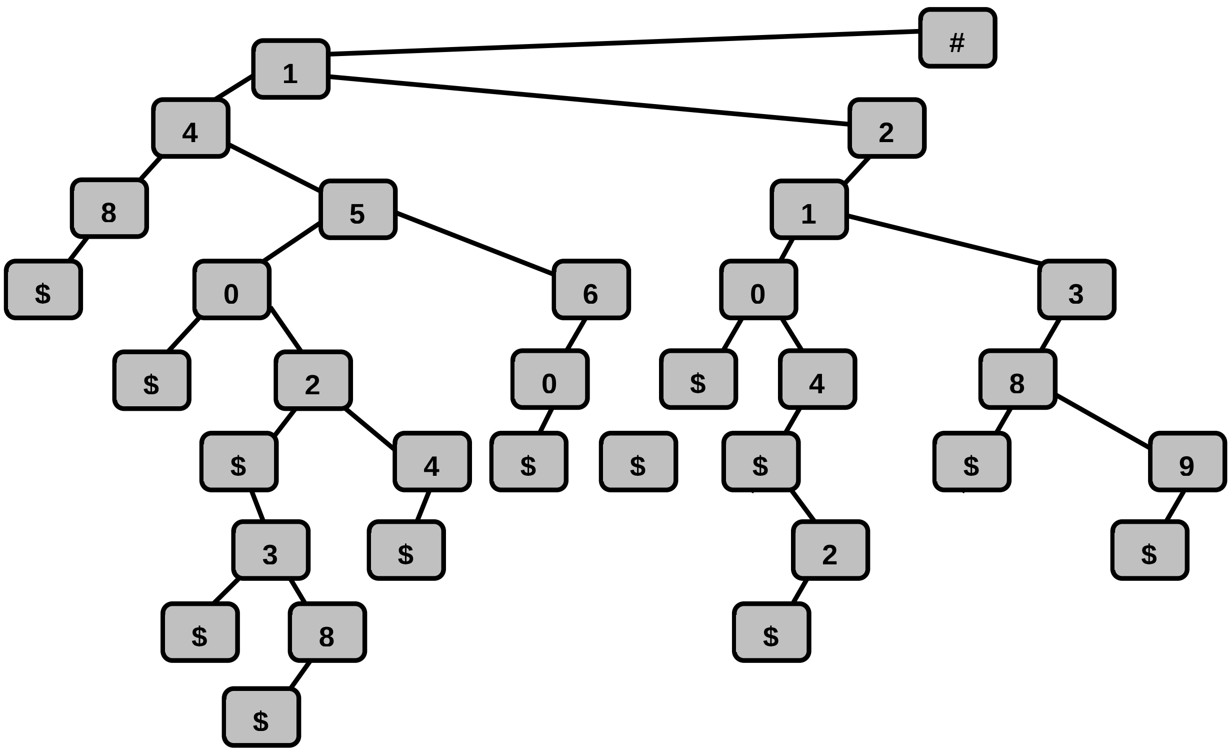
Java Interface

```
package TriePkg;

public interface ADT_Trie
{
    public boolean isEmptyTrie( );

    public boolean search( Trie_Key k );

    public void insertKey( Trie_Key k );

    public void deleteKey( Trie_Key k );
}
```

Linked List Implementation

First represent the trie as a binary tree.

- ◇ left child in the binary tree corresponds to leftmost child in the trie;
- ◇ right child in the binary tree corresponds to leftmost sibling in the trie.



◇ Keys are represented by character strings:

▷ the subset of characters that can be used as symbols in a key is assumed to have been specified, as is the character to be used as the end-of-key symbol

for example, might take the decimal digits

‘0’, ‘1’, . . . , ‘9’

as the key symbols, and any non-digit as the end-of-key symbol.

Implementation of ADT_Trie

```
package TriePkg;

public class Trie_Key
{
   String symbol_String;

   static final char EOK = '$';

   public Trie_Key( )
      { symbol_String = "";}

   public Trie_Key( String s )
      { symbol_String = s ; }

   public Trie_Key( Trie_Key k )
      { symbol_String = k.symbol_String; }

   public static String digits = "0123456789";

   public static void setSymbols( String s )
      { digits = s; }

   boolean isNullKey( )
      { return symbol_String.length( ) == 0; }
```

```
   boolean isValidTrie_Key( )
   {
      if ( isNullKey( ) ) return true;
      boolean found = true;
      for ( int i = 0; i < symbol_String.length( )
                            && found; i++ )
      {
         found = false;
         char c = symbol_String.charAt( i );
         for ( int j = 0; j < digits.length( )
                            && !found; j++ )
            if( c == digits.charAt( j ) )
               found = true;
      }
      return found;
   }
   public String toString()
      { return symbol_String; }
   char head( )
      { return symbol_String.charAt( 1 ); }
   Trie_Key tail( )
   { return new Trie_Key( symbol_String.substring( 1,
                            symbol_String.length( ) ) );}
}
// End of class Trie_Key
```

```
package TriePkg;
class TrieNode
{
// Data members
   TrieNode left;
   TrieNode right;
   char symbol;
// Constuctors
   TrieNode( )
      { this( null ); }
   TrieNode( char c )
      { this( c, null, null ); }
   TrieNode( char c, TrieNode lt, TrieNode rt )
      { symbol = c; left = lt; right = rt; }
   public String toString()
      { return( symbol ); }
```

```
static boolean search( TrieNode t, Trie_Key k )
{
   if ( t == null )}
      return false;
   char s = t.symbol;
   if( k.isNullKey() )
      if ( s == EOK )
         return true;
      else
         return false;
   else // k is not the null key
   {
      char kh = k.head( );
      if ( ( s == EOK ) || ( s < kh ) )
         return search( t.right, k );
      else
      {
         if ( s > kh )
            return false;
         else
            return search( t.left, k.tail( ) );
      }
   }
}
```

```
   static void insert( Trie_Key k, TrieNode t )
      ;// not implemented
   static void delete( Trie_Key k, TrieNode t )
      ;// not implemented
   }
// end of class TrieNode
```

```
package TriePkg;
public class Trie implements ADT_Trie
{
// Trie has only one data member
   TrieNode root;
// Constructors
public Trie( )
   { root = null; } // This corresponds to mkEmptyTrie
public Trie( char c )
   { root = new TrieNode( c ); }
public boolean isEmptyTrie( )
   { return root == null; }
```

```
public boolean search( Trie_Key k )
{
   if ( !k.isValidTrie_Key() )
   {
      System.out.println( "handle invalid key error");
      return false;
   }
else
   return TrieNode.search( root, k );
}
```

```
   public insertKey( Trie_Key k )
      {
         if ( !k.isValidTrie_Key() )
            System.out.println(
                        "handle invalid key error");
         else
            root = TrieNode.insert( k, root );
      }
   public deleteKey( Trie_Key k )
   {
      if ( !k.isValidTrie_Key() )
         System.out.println(
                        "handle invalid key error");
      else
         root = TrieNode.delete( k, root );
   }
}
   // End of class Trie
```

Complexity

◇ In the worst-case, searching for a key of length n takes $O(nm)$ time

  ▷ $m = |A| + 1$, the size of the alphabet plus 1 for \$.

◇ If no node in a trie has too many children,

  ▷ the set of keys, S, is said to be *sparse*

then the use of m in the search-time complexity bound may be a big over-estimate.

◇ If the number of children is generally nearer to m than to 1, then the set of keys is said to be *dense*.

Pruning Straggly Branches

◇ If a long branch leads to a single key, we can coalesce the branch.

◇ For example, if key 135689 is the only key with the prefix 135, once the nodes for 135 have been matched, we can go immediately to the complete key.

Array Implementation of a Trie

Example

180, 185, 1867, 195, 207, 217, 2174, 21749, 217493, 226, 27, 274, 278, 279, 2796, 281, 284, 285, 286, 287, 288, 294, 307, 768.

Insert 180:

	1
0	
1	180
2	
3	
4	
5	
6	
7	
8	
9	
$	

Insert 185:

	1	2	3
0			180
1	(2)		
2			
3			
4			
5			185
6			
7			
8		(3)	
9			
$			

Insert 1867 and then 195:

	1	2	3
0			180
1	(2)		
2			
3			
4			
5			185
6			1867
7			
8		(3)	
9		195	
$			

Insert 207:

	1	2	3
0			180
1	(2)		
2	207		
3			
4			
5			185
6			1867
7			
8		(3)	
9		195	
$			

Insert 217:

	1	2	3	4
0			180	207
1	(2)			217
2	(4)			
3				
4				
5			185	
6			1867	
7				
8		(3)		
9		195		
$				

Insert 2174:

	1	2	3	4	5	6
0			180	207		
1	(2)			(5)		
2	(4)					
3						
4						2174
5			185			
6			1867			
7					(6)	
8		(3)				
9		195				
$						217

Insert 21749:

	1	2	3	4	5	6	7
0			180	207			
1	(2)			(5)			
2	(4)						
3							
4						(7)	
5			185				
6			1867				
7					(6)		
8		(3)					
9		195					21749
$						217	2174

$\vdots \qquad \vdots \qquad \vdots$

Finally:

	1	2	3	4	5	6	7	8	9	10	11
0			180	207							
1	(2)			(5)							281
2	(4)			226							
3	307							217493			
4						(7)			274		284
5			185								285
6			1867							2796	286
7	768			(9)	(6)						287
8		(3)		(11)					278		288
9		195		294			(8)		(10)		
$						217	2174	21749	27	279	

Searching for a Key

- Given $k = a_1 a_2 \ldots a_n$;

- let T be the array implementing the trie:

  - if $T[a_1, 1]$ is a key entry, this means there is only one key with prefix a_1

    – if k matches the entry, the search is successful, otherwise, k is not present in the trie;

  - if $T[a_1, 1]$ gives another column index in T,

    $T[a_1, 1] = j$, say

    then column j represents all keys in the trie prefixed by a_1, so goto $T[a_2, j]$, etc.

Complexity

- In the worst-case, to search for a key of length n, we access n elements in the array, T.
- Time for each access is $O(1)$.
- Worst-case search-time is therefore $O(n)$.
- Worst-case storage complexity is $O(Nn_{\max})$:
  - $N = |S|$, the size of the set of keys;
  - $n_{\max}$ is the maximum length of a key;
  - worst-case is for *sparse* set of keys
    - e.g. 2 keys each of length 100 differing only in their final digit use 100 columns in the array;
  - for a dense set of keys, on average a column will contain many keys, as well as cursors to other columns
    - e.g. if on average, 50% of the entries in a column are keys, then storage complexity is $O(N)$.
- The array method is therefore suitable for implementing a trie for which the set of keys is dense.