CS/IS F211
Data Structures & Algorithms

15-Mar-14

1

N-ary Trees

- Traversal(s) and Applications

Trees – Operations and Representation

- Operations
  - TREE createTree(int maxChildren)
    - // Use an invalid argument for arbitrary branching
  - TREE isEmptyTree(TREE)
  - Element rootVal(TREE)
  - Iterator getChildren(TREE)
    - // Define an iterator to access children.
  - TREE makeTree(Element rootVal, TREE *children)
- Representation
  - Each node is represented as a record: <*Value, Children*>
    - List of *Children* is usually an array or a linked list.
      - When do you use a linked list?

Trees - Representation

- N-ary trees

```
typedef struct _node  *TREE;
struct _node {
    Element val;
    TREE  children[N];         // N is a constant
    int numCh;                 // actual number of children
};
```

Alternative: Allocate dynamically!

- Arbitrary Branching

```
typedef struct _node  *TREE;
struct _node {
    Element val;
    TREE  *children;
};   // children is head of a linked list
```

Trees – Traversals - DFT

- Depth First Traversal
  - Traverse one path (from root to leaf) completely before starting on another path
- Algorithm:

```
dfsTree(Tree t)
{
    if t is empty return;
    visit root;              // do what you have to!
    for each c in children of t { dfsTree(c); }
}
```

Trees – Traversals – DFT [2]

- Depth First Traversal – Recursive Implementation

```
dfsTree(Tree t)
{
        if (isEmptyTree(t)) return;
        visit(rootVal(t));              // do what you have to!
        Iterator Ch = getChildren(t);
         while (hasMoreElements(Ch)) {
                dfsTree(getNextElement(Ch));
         }
}
```

- Recursive call is inside a loop – how to eliminate this?

TREES – TRAVERSALS – DFT [3]

- Depth First Traversal – (Naïve) Iterative Implementation

```
dfsTree(Tree t)
{    Stack st = createStack();
     BEGIN:  if (isEmptyTree(t)) return;
     visit(rootVal(t));           // do what you have to!
     Iterator Ch = children(t);
     while (hasMoreElements(Ch)) {
             st = push(getNextElement(Ch), st);
      }
      if (!isEmptyStack(st)) {
           t = top(st); st = pop(st);   goto BEGIN;
      }
}
```

Can we avoid pushing all children on stack?

Trees – Traversals – DFT [4]

- Depth First Traversal – Iterative Implementation

```
dfsTree(Tree t)
{   if (isEmptyTree(t))  return;
    Stack st = createStack();

    st = push(getChildren(t),  st);
     while (!isEmptyStack(st)) {
          tCh = top(st);
            if (hasMoreElements(tCh)) {
               t = getNextElement(tCh);

               st = push(getChildren(t), st);
          } else {  st = pop(st);
          }
       }
}
```

Where should the visits occur?

Store the iterator for a node's children instead of storing all children.

The Iterator is a pointer to a node in a linked list or an (integer) index and starting address of an array.

Trees – Traversals – DFT [5]

- Depth First Traversal – Iterative Implementation

```
dfsTree(Tree t)
{   if (isEmptyTree(t))  return;
    Stack st = createStack();
    visit(rootVal(t));
    st = push(getChildren(t),  st);
     while (!isEmptyStack(st)) {
            tCh = top(st);
              if (hasMoreElements(tCh)) {
                t = getNextElement(tCh);
                visit(rootVal(t));
                st = push(getChildren(t), st);
            } else {  st = pop(st);
            }
       }
}
```

Visit a node and then push its iterator on stack.

When all children of a node are visited pop the corresponding iterator!

Tree Traversals – BFT

- Breadth First Traversal (a.k.a. Level Order Traversal)
  - Traverse one level (of depth) completely before starting on a lower level
- Algorithm:

```
bfsTree(Tree t) {
  if (!isEmpty(t)) { visit(rootVal(t));  bfsLevel("children of t"); }
}

bfsLevel(Set remSet) {
    copy remSet into cSet
    for each c in cSet {
        visit(c);
        remSet = remSet - { c } U getChildren(c);
    }
    bfsLevel(newRemSet);
}
```

Representation for Set?

No ordered queries

First-in-First out – Why?

A FIFO Queue is a natural choice

This is a tail call

Tree Traversals – BFT [2]

```
bfsTree(Tree t) {
  if (!isEmpty(t)) {
    Queue q = createQ();  q = addQ(t, q); bfsLevel(q);   }
}
 bfsLevel(Queue q) {
   while (!isEmpty(q)) {
        t = getQ(q);  q = deleteQ(q);
        visit(t);
        Iterator ch = getChildren(t);
      while (hasMoreElements(ch))
                  q = addQ(getNextElement(ch), q);
   }
}
```

Faster addition?

Queue of Iterators:

may increase some work for getQ

TREES - TRAVERSALS

- Time Complexity of DFT and BFT
  - O(n)
- Space Complexity
  - DFT: Size of stack
    - Height of the tree
  - BFT: Size of queue
    - Maximum # nodes in two consecutive levels
      - Exercise: Make it more precise!
- Under what conditions does the size of the queue (in BFT) get larger than the size of the stack (in DFT)?
  - Give a comparative characterization so that one can choose DFT or BFT
    - for a particular application and/or a given (class of) tree(s)!

Trees – Applications – File Systems

- Implement the following using traversal:
  - (Unix command) find
  - (Unix Command) cp – R
    - Look up the man pages to understand the commands.
    - Look up man pages (for *dirent* / *readdir*)
    - Choice of traversal
      - DFT or BFT?? Why??
  - (Windows Explorer) Navigation
    - Expand "on click"
  - Game Playing (e.g. Chess)
    - Find the next steps (given current board position)
    - Best First Traversal : Find the best next steps and expand them further
- Exercise:
  - Identify other day-to-day (computational) examples!

STRINGS

- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}

- Let $\boldsymbol{P}$ be a string of size $\boldsymbol{m}$
  - A substring $\boldsymbol{P}[\boldsymbol{i} \mathbin{..} \boldsymbol{j}]$ of $\boldsymbol{P}$ is the subsequence of $\boldsymbol{P}$ consisting of the characters with ranks between $\boldsymbol{i}$ and $\boldsymbol{j}$
  - A prefix of $\boldsymbol{P}$ is a substring of the type $\boldsymbol{P}[0 \mathbin{..} \boldsymbol{i}]$
  - A suffix of $\boldsymbol{P}$ is a substring of the type $\boldsymbol{P}[\boldsymbol{i} \mathbin{..} \boldsymbol{m} - 1]$
- Given strings $\boldsymbol{T}$ (text) and $\boldsymbol{P}$ (pattern), the pattern matching problem consists of finding a substring of $\boldsymbol{T}$ equal to $\boldsymbol{P}$
- Applications:
  - Text editors
  - Search engines
  - Biological research

PREPROCESSING

- Preprocessing the pattern speeds up pattern matching queries
  - After preprocessing the pattern, KMP's algorithm performs pattern matching in time proportional to the text size
  - Every search ∞ size of text
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
  - Preprocess text
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
  - pattern matching queries ∞ the pattern size

OUTLINE

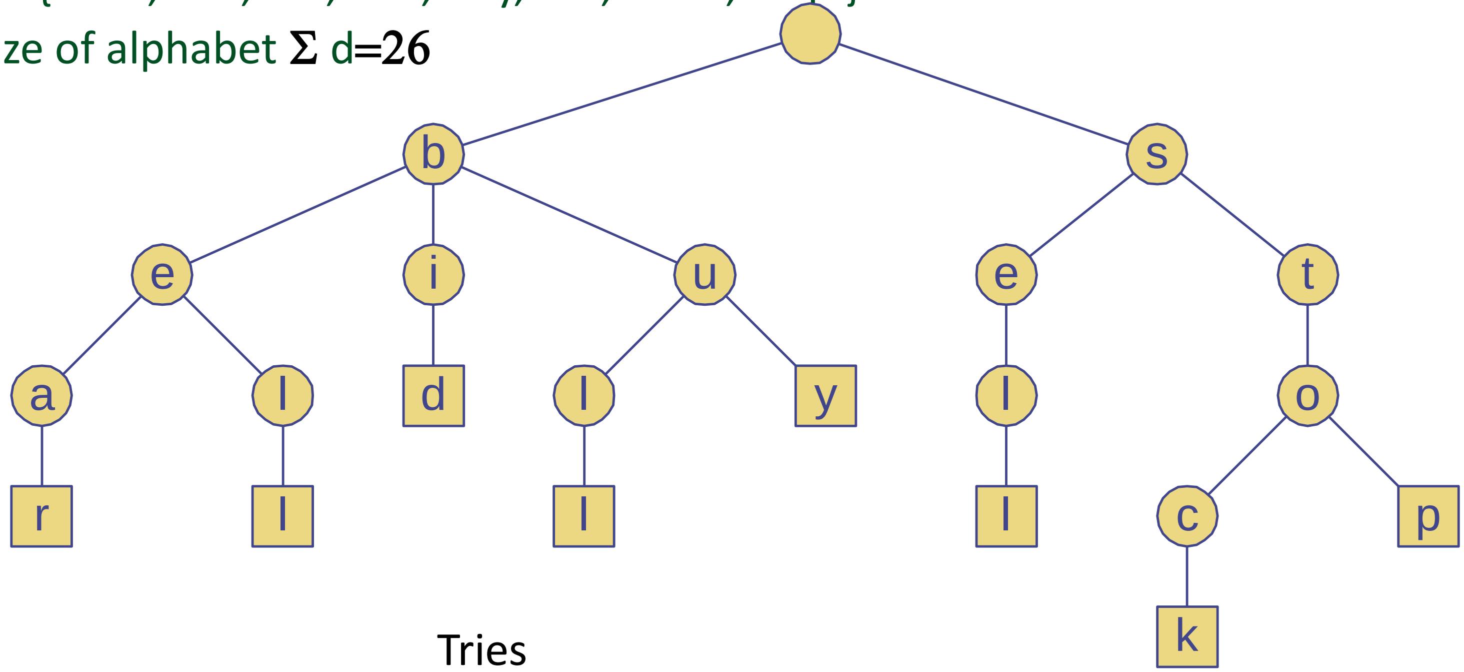
- Standard tries
- Compressed tries
  - Space efficient way of storing standard tries
- Suffix trees

Standard Trie (1)

- The standard trie for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered (left to right)
  - The paths from the root node to the external node yield the strings of S
- Example: standard trie for the set of strings

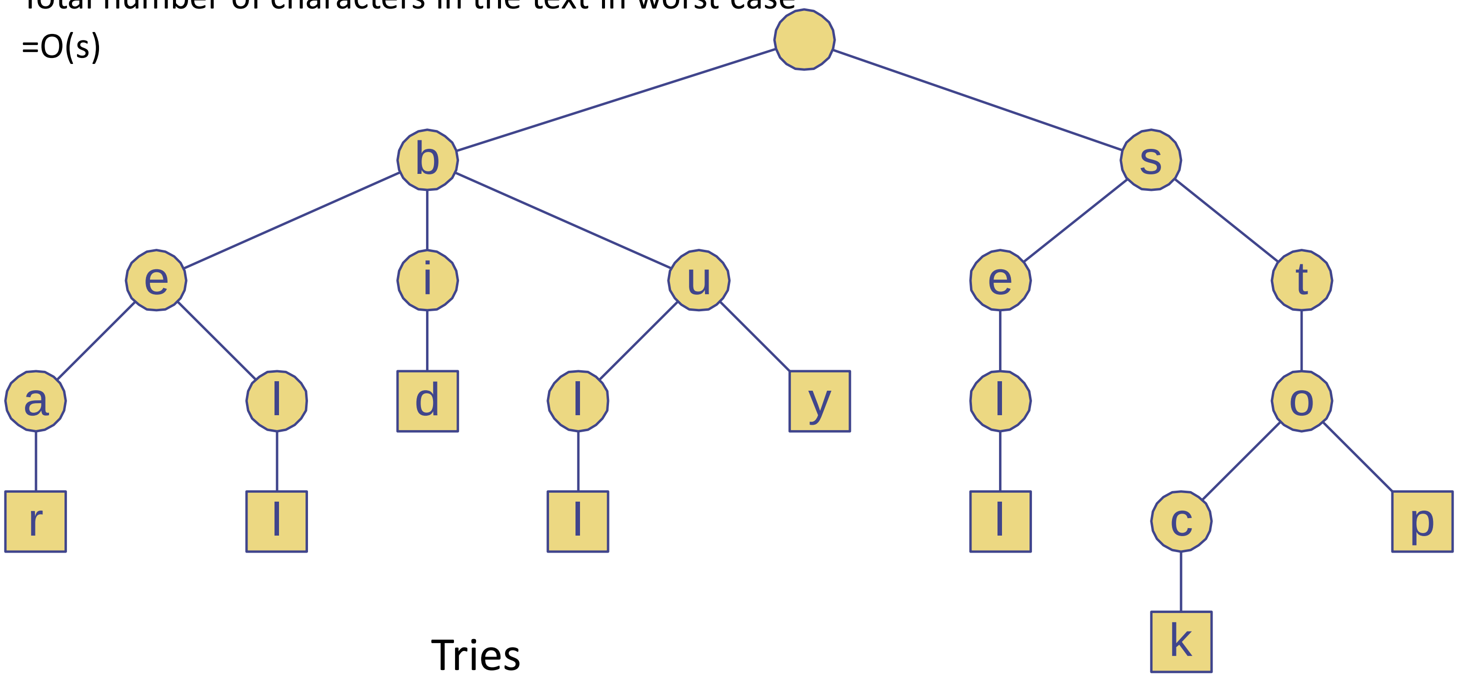
  S = { bear, bell, bid, bull, buy, sell, stock, stop }

  Size of alphabet Σ d=26



STANDARD TRIE (1)

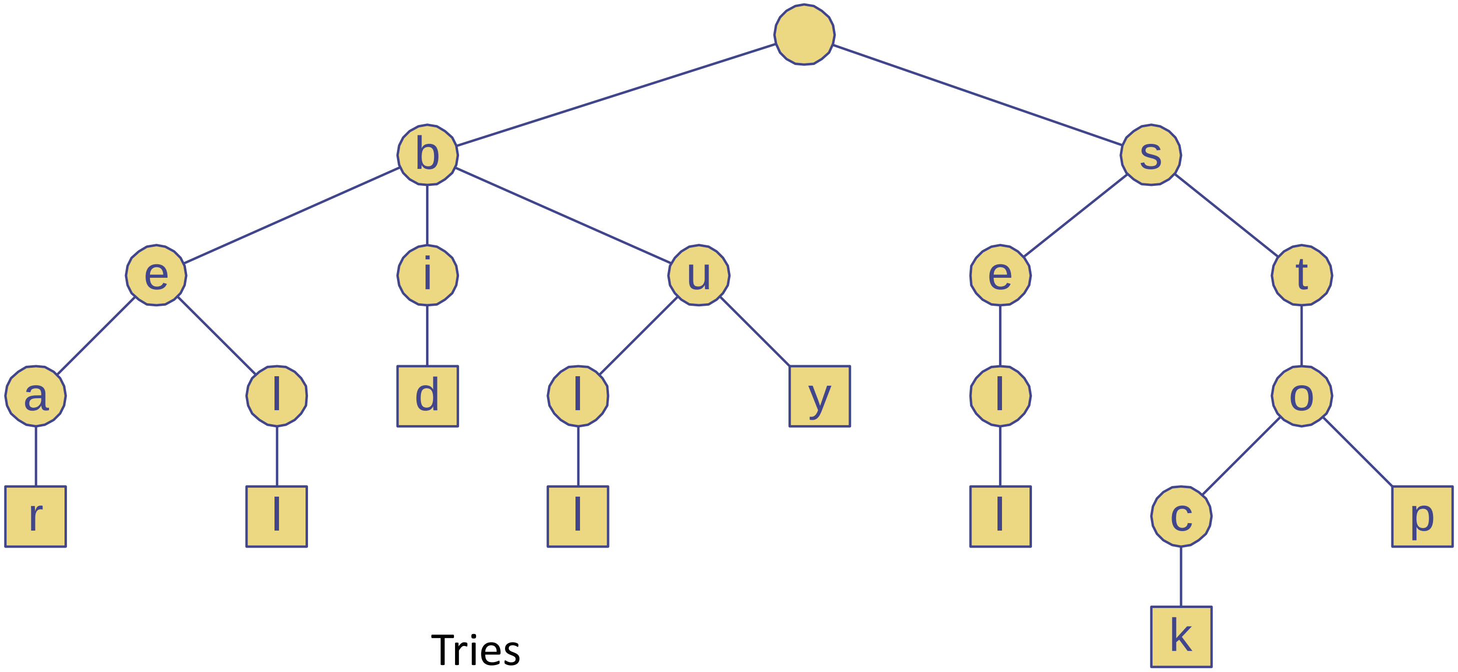
- Each node can have up to 26 children.
  - Represent children: Array or linked list?
- How much time it takes to search a given word?
  - 26*length of the word
  - =O(d*m) where m is the size of the word to be searched.
- How much space?
  - Total number of characters in the text in worst case
  - =O(s)



Tries

STANDARD TRIE (2)

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  - n total size of the strings in S
  - m size of the string parameter of the operation
  - d size of the alphabet



Applications of Standard Tries

- Matching in O(m) where m is the size of the word.
- Word matching
  - Find the first occurrence of word in the text
- Prefix matching
  - Find the first occurrence of the longest prefix of word in the text

Word Matching with a Trie

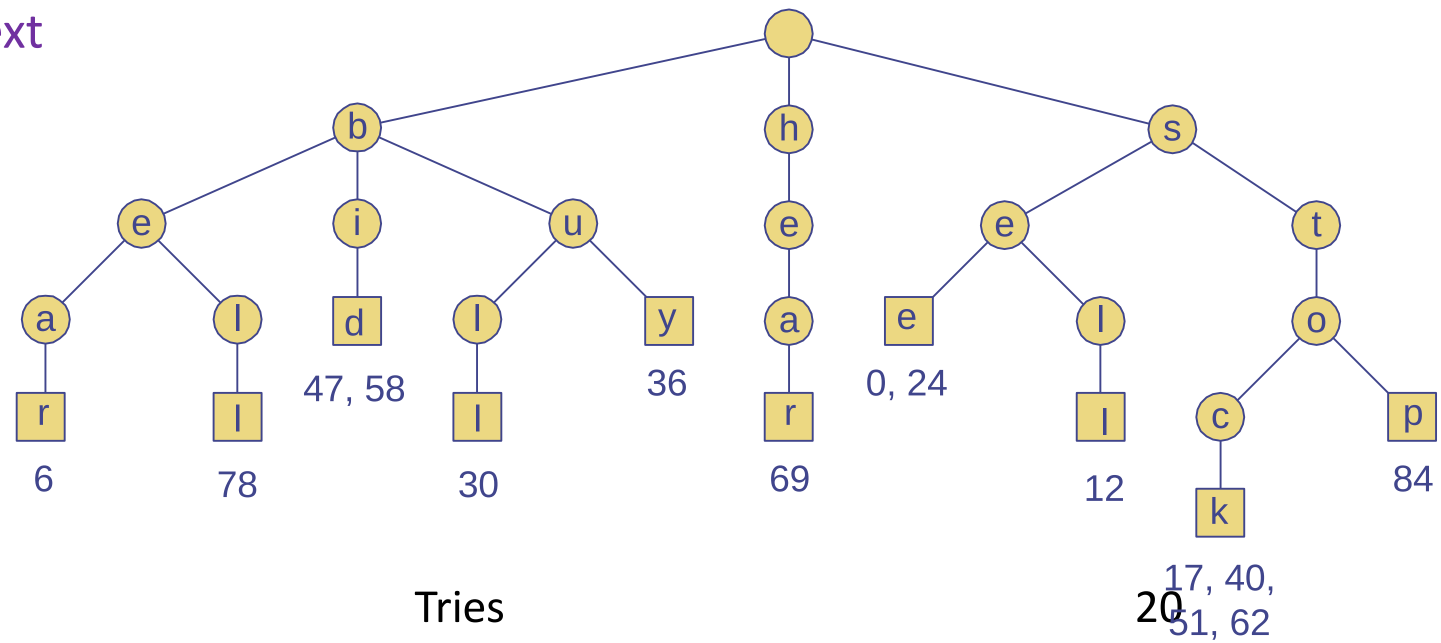
- Multiple occurrences.
- Each leaf stores the occurrences of the associated word in the text

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!	
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88

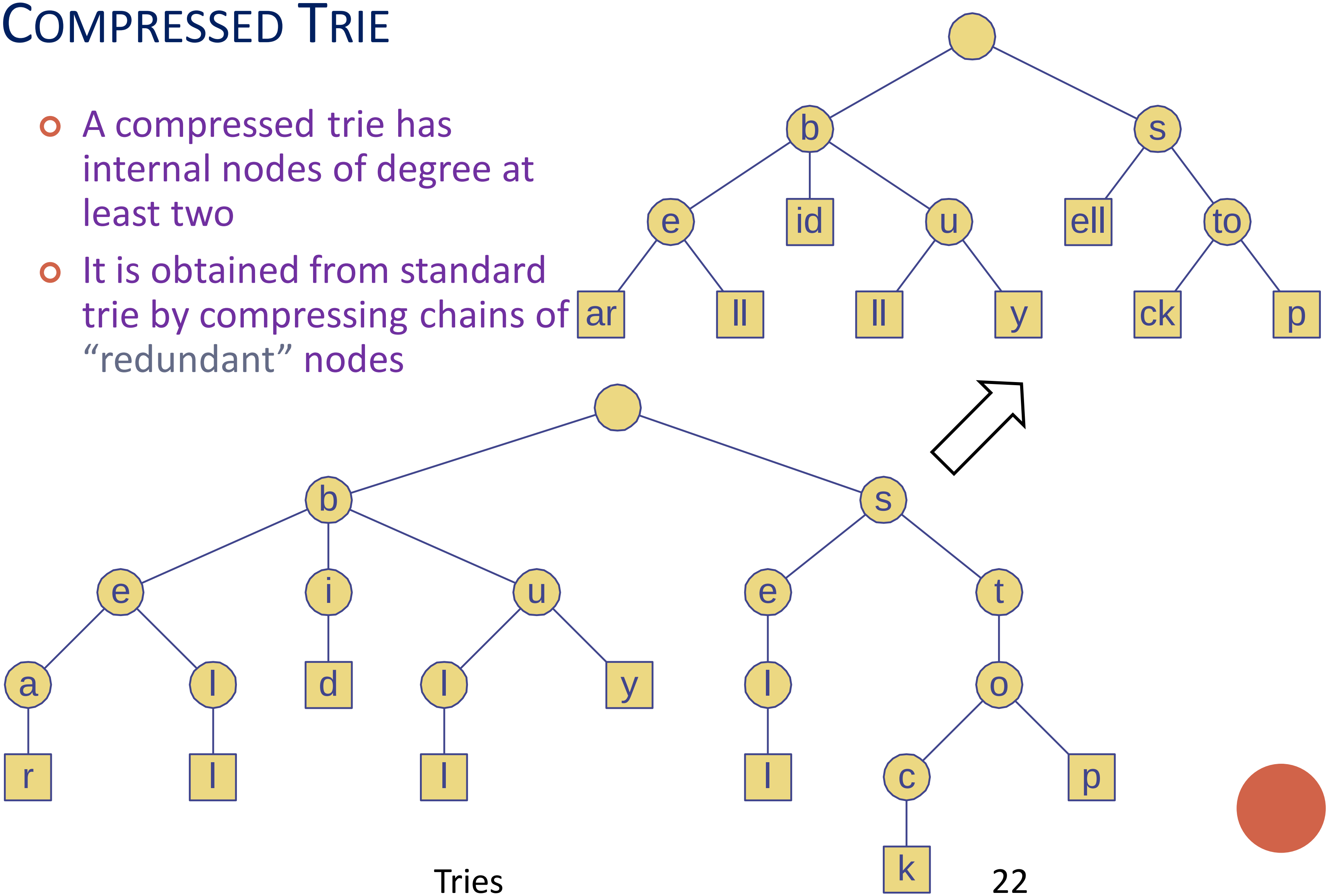


Pattern Matching

- Arbitrary pattern which may not be a word?
- Ull? Ock?
- Suffix trees.

Compressed Trie

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes



Compressed Tries

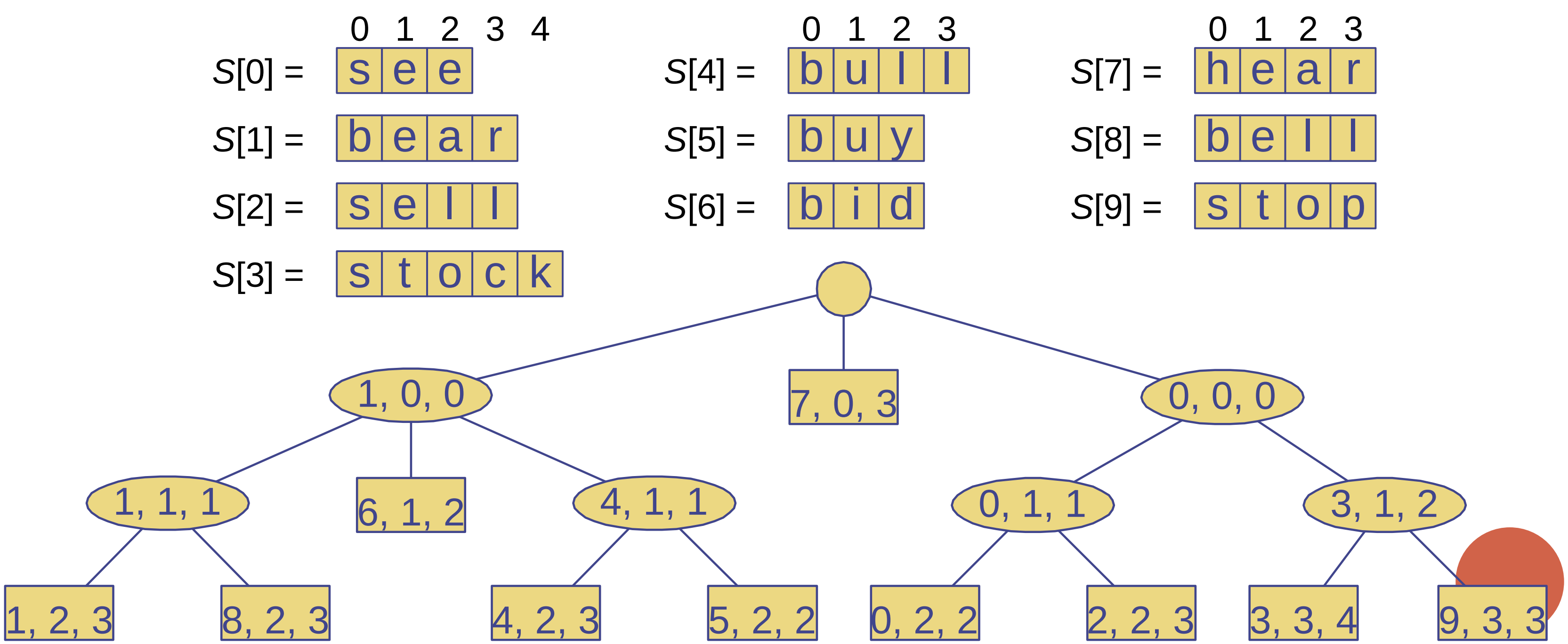
- A tree in which every internal node has at least two children has utmost L-1 internal nodes where L is the number of leaves.
- Number of nodes in a compressed trie is proportional to number of strings, not length of all strings.

COMPRESSED TRIES

- But now nodes store not a character but a label.
- Store index range instead of a label.

Compact Representation

- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(s)$ space, where s is the number of strings in the array
  - Serves as an auxiliary index structure



TRIES APPLICATIONS

- Search Engine
  - Trie: index of all searchable words
  - Each leaf is associated with a word and has pointer to a list of URLs or documents which contain this word.
  - Trie is kept in main memory.
  - Set operations union, intersection.

TRIES APPLICATIONS

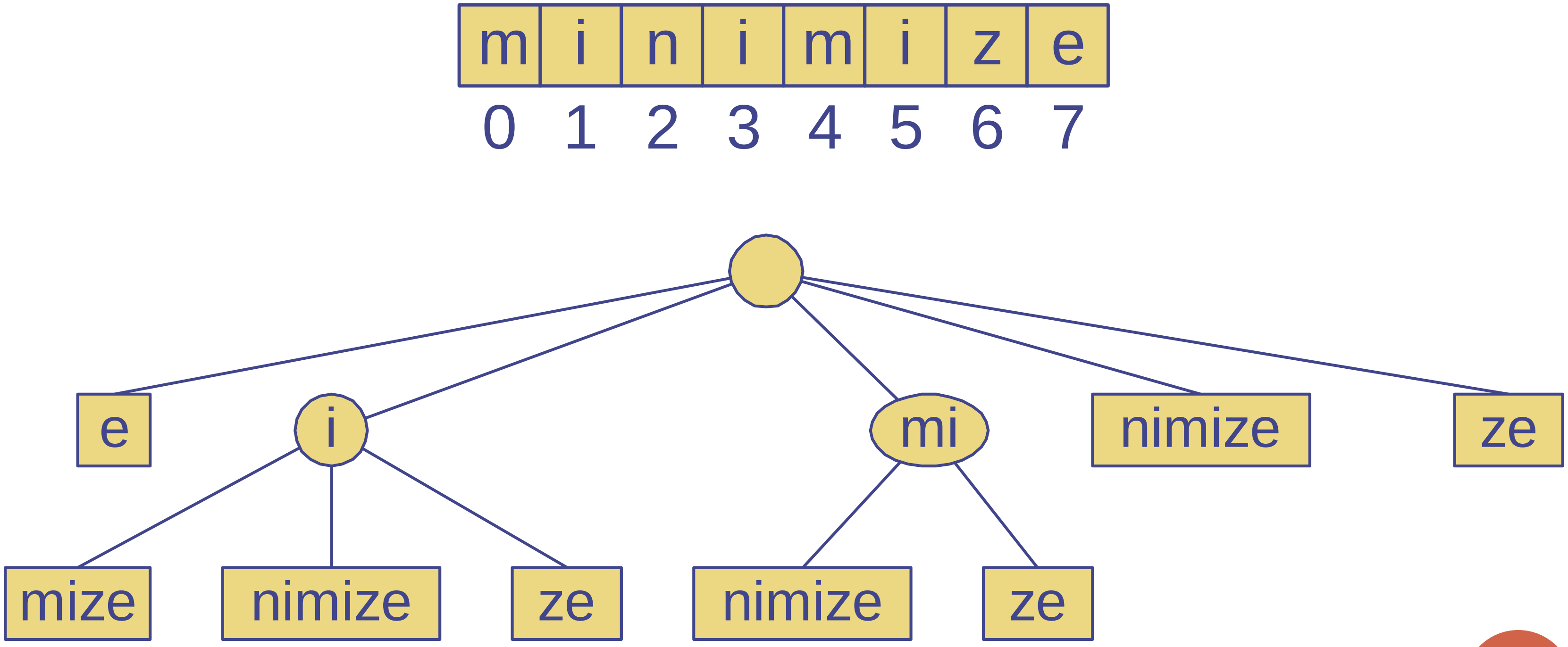
- Routers
  - Routing table contains IP prefix, interface
  - 2^32 addresses
  - Prefixes are stored, not individual addresses.
  - Match destination address with longest prefix.
  - Routers use alphabet of {0,1}

Pattern Matching

- Arbitrary pattern which may not be a word?
- Ull? Ock?
- Suffix trees.

SUFFIX TREE

- The suffix trie of a string X is the compressed trie of all the suffixes of X

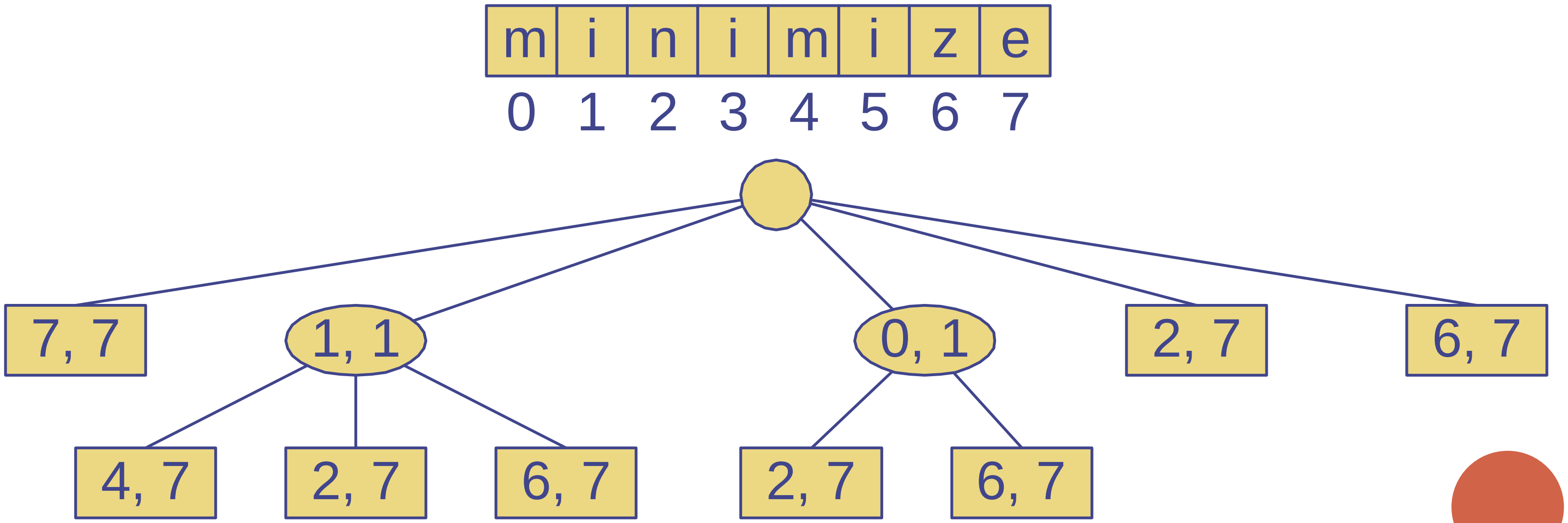


Suffix tree

- Suffix tree is made for entire text not for every word.
- Space: O(s) where s is the size of the text.
- Compressed trie size= number of strings.
- Here number of suffixes is equal to number of characters.

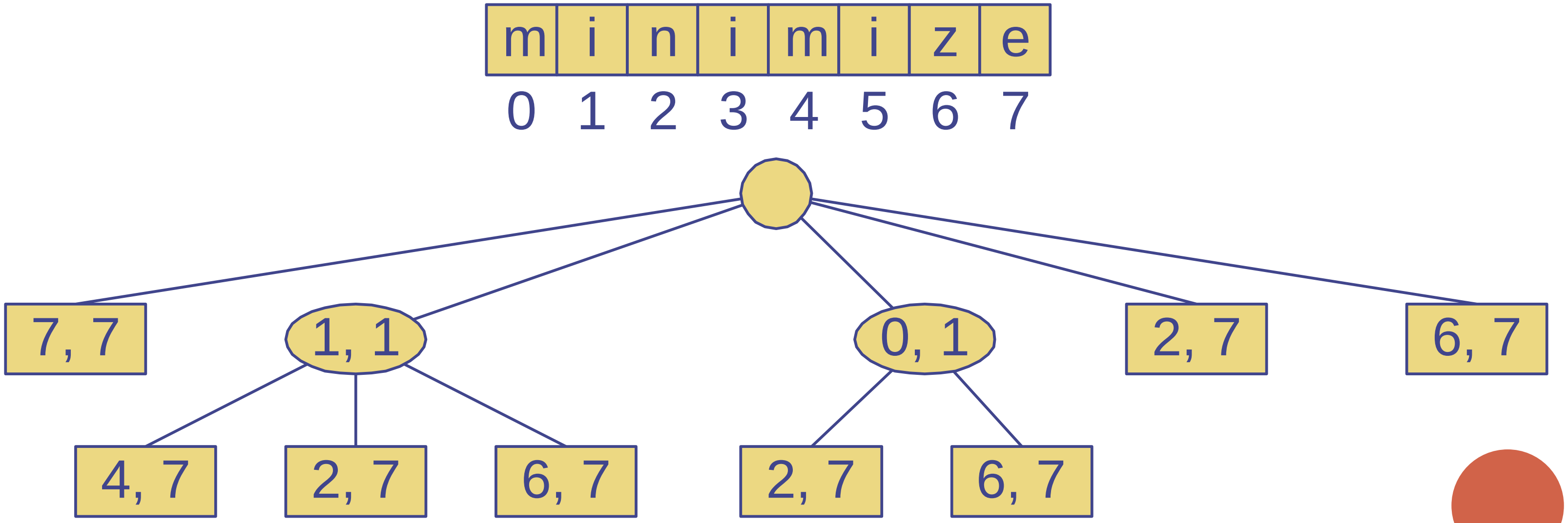
SUFFIX TREE

- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern



SUFFIX TREE

- If a suffix is pre-fix of another suffix, then use a terminating special character, because that suffix will not endup at leaf node.



SUFFIX TREE

- Building a suffix tree will take O(n) time.

SUFFIX TREE- APPLICATIONS

- Searching for a substring in O(m) time.
- Longest Repeated Substring in O(s) time.
- Longest Common Substring
  - str$str#
- Polindromes
  - Str$reverse(str)#