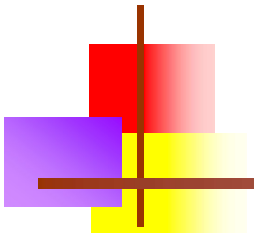
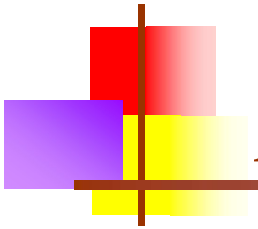


Ternary Search Tries

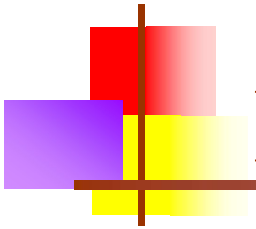
Professor Peter Brass
City College of New York

Kanhar Munshi, 2012



Approach

1. Some computational problems (1 minute)
2. Introduction to tries (1-2 minutes)
3. Introduction to ternary tries (5-7 minutes)
    a. Insert (3)
    b. Search (1)
    c. Partial Search (1)
    d. Draw (1)
4. Solutions to above Problems. (1 minute)
5. Q&A (1 minute)



Practical: Spell Check

- How do you build a spell check function where the vocabulary is greater than the size of the computer.
  - Bloom filter
    - Can validate incorrect spellings with 100% probability
      - May mark a correct spelling as incorrect
        - Which may be acceptable for a spell check function
    - **Needs to be resized if word list grows.**
    - **Cannot suggest alternative spellings**

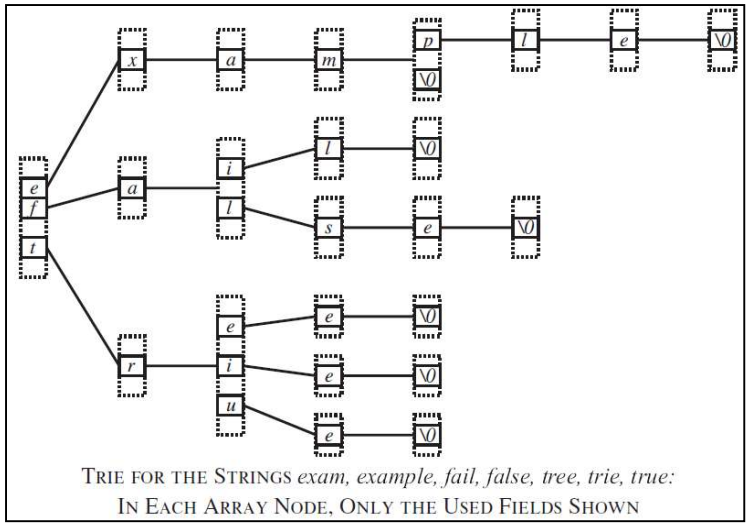
Practical: Concordance

- Store all the occurrences of each word in a collection of words:-
  - Each word contains a list of positions, where it occurred in the text, or even in multiple texts

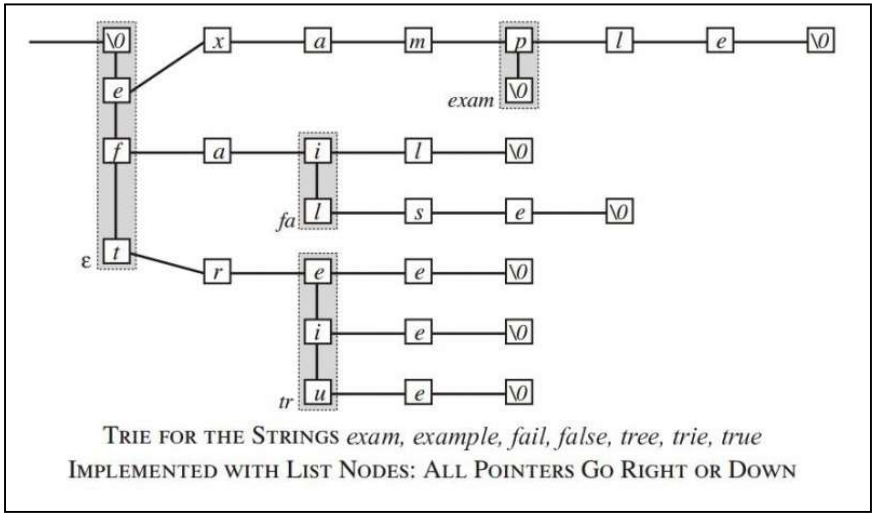
s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!		
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	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				

Standard Tries: Introduction

- The standard trie for a set of strings S is an ordered tree such that:
  - Each node **excluding the root** is labeled with a character
  - There can be |A| children of each node
    - The children are alphabetically ordered.
  - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings S = { **e**xam, example, **f**ail, false, **t**ree, trie, true }
  - Trie with 256 pointers
  - Trie with Linked list of variable number of pointers.



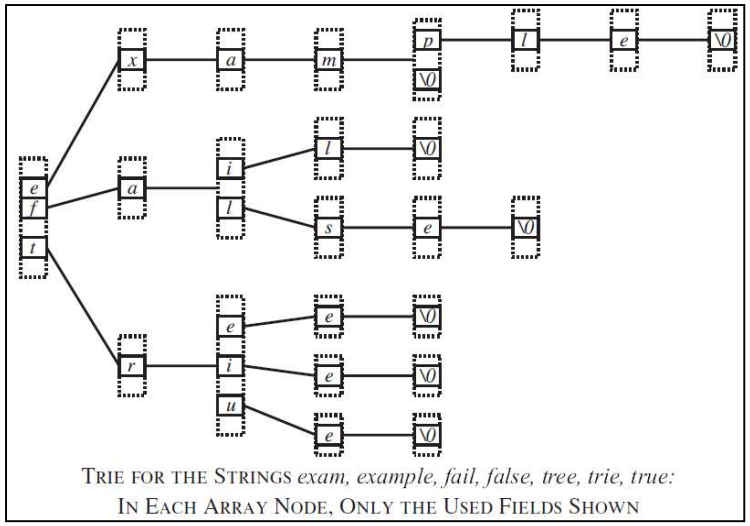
TRIE FOR THE STRINGS *exam, example, fail, false, tree, trie, true:*
IN EACH ARRAY NODE, ONLY THE USED FIELDS SHOWN



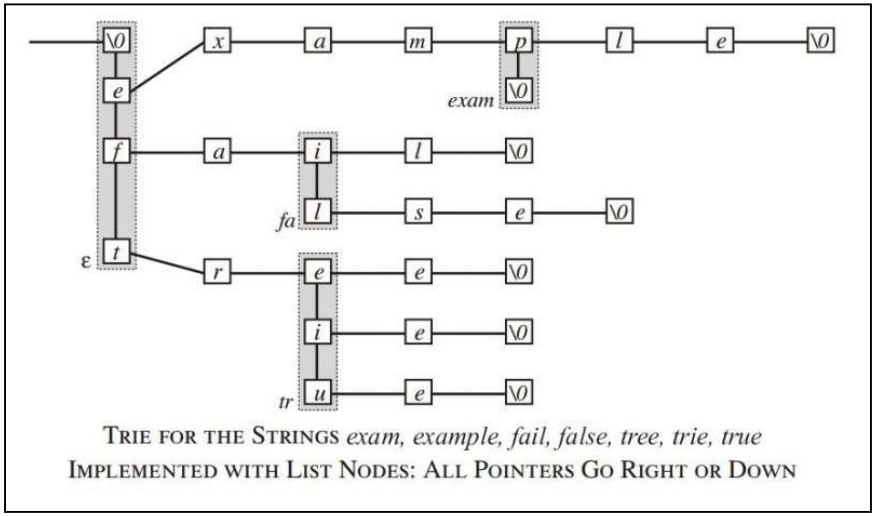
TRIE FOR THE STRINGS *exam, example, fail, false, tree, trie, true*
IMPLEMENTED WITH LIST NODES: ALL POINTERS GO RIGHT OR DOWN

Standard Tries: Optimizations

- Use Linked list instead of |A| pointers
- Use HBT instead of linked list
- Use HBT globally for entire trie.
- Alphabet Reduction
- Or …use a totally different approach.



TRIE FOR THE STRINGS *exam, example, fail, false, tree, trie, true:*
IN EACH ARRAY NODE, ONLY THE USED FIELDS SHOWN



TRIE FOR THE STRINGS *exam, example, fail, false, tree, trie, true*
IMPLEMENTED WITH LIST NODES: ALL POINTERS GO RIGHT OR DOWN

Ternary Tries: Introduction

- The standard trie for a set of strings S over an alphabet A is an ordered tree such that:
  - Each node **including the root** is labeled with a character
  - There are only three children of each node
    - Lower, Equal, Higher
  - The paths from the external nodes to the root yield the strings of S
    - The leaf nodes can contain an object or object(S) that have as their key the string formed in navigating from root to the leaf.
  - Benefits:-
    - Find is independent of the size of the alphabet
    - Sorted list of words, traversal (in order post order), partial finds

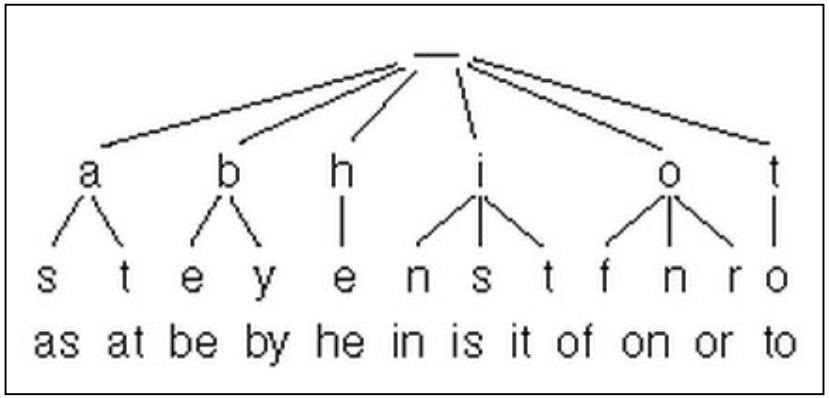


Figure 1: Trie for 12 words

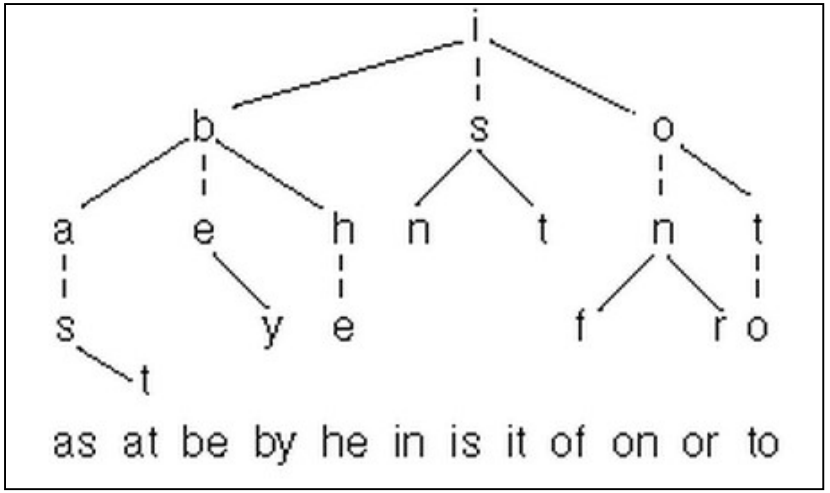
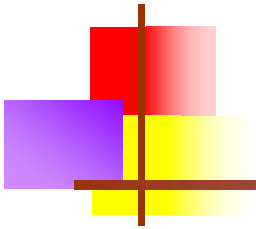
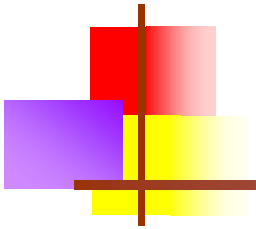


Figure 2: Ternary Search Trie for the same 12 words



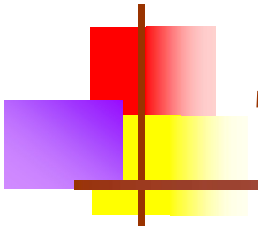
TST: Advantages

- Ternary trees do not incur extra overhead for insertion or successful searches.
- Ternary trees gracefully grow and shrink; hash tables need to be rebuilt after large size changes.
- Ternary trees support advanced searches, such as partial-match and near-neighbor search.
- Ternary trees support many other operations, such as traversal to report items in sorted order.



TST: Disadvantages

- Not many ☺
- Harder to visualize
- Seemingly harder to implement.



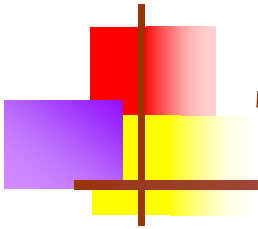
TST: Definition

```
typedef struct t_node_t
{
    char key;
    struct t_node_t *lower,
                    *equal,
                    *highr;
} ternary_trie;
```

```
ternary_trie* tst_insert(ternary_trie* p,        char *key, char *value);

int  tst_search_regular( ternary_trie* root,     char *s);
void tst_search_partial( ternary_trie* root,     char *s);
void tst_neighbor_search(ternary_trie* root,     char *s, int d);

void tst_draw(ternary_trie* p_trie, int depth, int pOnlyWords);
```



TST: tst_insert

```
ternary_trie* tst_insert(ternary_trie* p_trie, char *p_insert, char *p_cur_char)
{
    if (p_trie == 0)
    {
        p_trie = (ternary_trie*) malloc(sizeof(ternary_trie));
        p_trie->key = *p_cur_char;
        p_trie->lower = 0;
        p_trie->equal = 0;
        p_trie->highr = 0;
    }

    if (*p_cur_char < p_trie->key)
        p_trie->lower = tst_insert(p_trie->lower, p_insert, p_cur_char);
    else if (*p_cur_char > p_trie->key)
        p_trie->highr = tst_insert(p_trie->highr, p_insert, p_cur_char);
    else if (*p_cur_char == p_trie->key)
    {
        if (*p_cur_char != 0)
            p_trie->equal = tst_insert(p_trie->equal,p_insert, ++p_cur_char);
        else
            p_trie->equal = (ternary_trie*) p_insert;
    }

    return p_trie;
}
```

TST: Creation 1

```
ternary_trie* tst_insert(ternary_trie* p_trie, char *p_insert, char *p_cur_char)
{
    if (p_trie == 0)
    {
        p_trie = (ternary_trie*) malloc(sizeof(ternary_trie));
        p_trie->key = *p_cur_char;
        p_trie->lower = 0;
        p_trie->equal = 0;
        p_trie->highr = 0;
    }

    if (*p_cur_char < p_trie->key)
        p_trie->lower = tst_insert(p_trie->lower, p_insert, p_cur_char);
    else if (*p_cur_char > p_trie->key)
        p_trie->highr = tst_insert(p_trie->highr, p_insert, p_cur_char);
    else if (*p_cur_char == p_trie->key)
    {
        if (*p_cur_char != 0)
            p_trie->equal = tst_insert(p_trie->equal,p_insert, ++p_cur_char);
        else
            p_trie->equal = (ternary_trie*) p_insert;
    }

    return p_trie;
}
```

```
Printing entire Ternary Search Tree
t
   r
      e
         e
            \tree
```

Step1: Insert ‘tree’

```
Printing entire Ternary Search Tree
t
   r
      e
         i
            e
               \trie
         e
            \tree
```

Step2: Insert ‘trie’

TST: Visual

Words(w)=
{
exam,
example,
fail,
false,
tree,
trie,
true
}

Fig 1: Input set of words

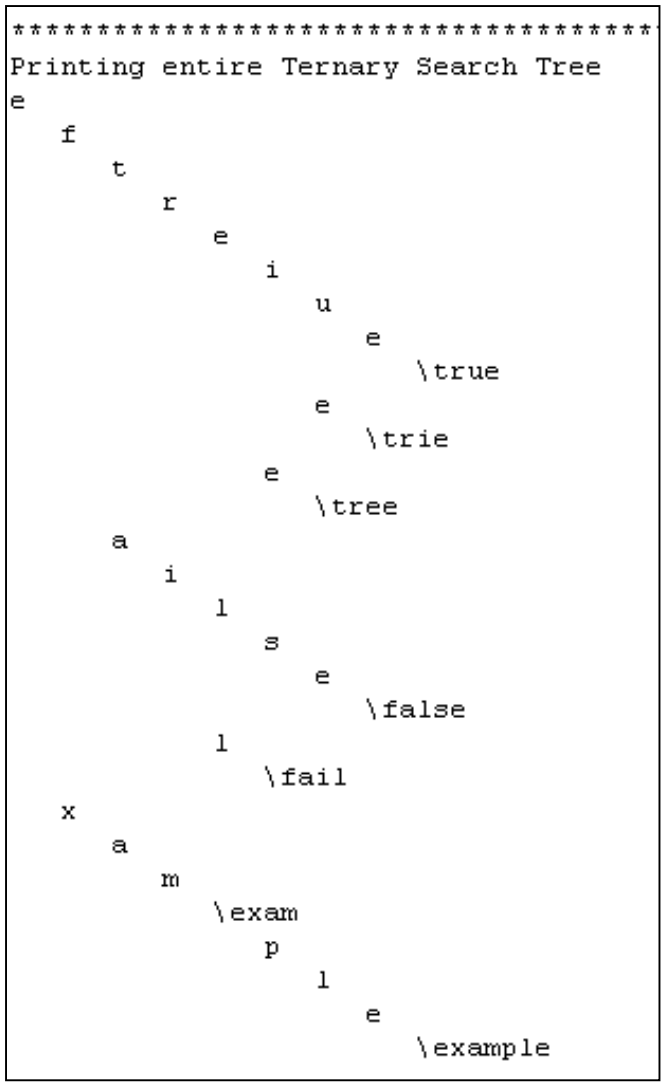


Fig 2: Associated Ternary Search Trie

TST: search_regular

```
int tst_search_regular(ternary_trie* p_root,char *p_key)
{
    ternary_trie* p = p_root;

    while (p)
    {
        if (*p_key < p->key)
            p = p->lower;
        else if (*p_key > p->key)
            p = p->highr;
        else
        {
            if (*p_key++ == 0)
            {
                printf("%s Found\n", p_key);
                return 1;
            }
            else
                p = p->equal;
        }
    }

    printf("%s Not Found\n", p_key);
    return 0;
}
```

```
**************************************
Printing entire Ternary Search Tree
e
   f
      t
         r
            e
               i
                  u
                     e
                        \true
                  e
                     \trie
               e
                  \tree
      a
         i
            l
               s
                  e
                     \false
            l
               \fail
   x
      a
         m
            \exam
               p
                  l
                     e
                        \example
```

```
Testing Regular Search:-
'tries' Not Found
'example' Found
```

TST: tst_search_partial

```
void tst_search_partial(ternary_trie* p_root,char *p_key, int p_len)
{
    ternary_trie* p = p_root;

    printf("Searching for matches to Regex: %s*\n", p_key);

    while (p)
    {
        if (*p_key < p->key)
            p = p->lower;
        else if (*p_key > p->key)
            p = p->highr;
        else
        {
            if (--p_len <= 0)
                tst_draw(p, 0, TRUE);

            if (*p_key++ != 0)
                p = p->equal;
        }
    }
}
```

```
**************************************
Printing entire Ternary Search Tree
e
   f
      t
         r
            e
               i
                  u
                     e
                        \true
                  e
                     \trie
               e
                  \tree
      a
         i
            l
               s
                  e
                     \false
            l
               \fail
   x
      a
         m
            \exam
               p
                  l
                     e
                        \example
```

```
Testing Partial Search:-
Searching for matches to Regex: fa*
false
fail
Searching for matches to Regex: tr*
true
trie
tree
```

TST: Draw and Print

```
void tst_draw(ternary_trie* p_trie, int depth, int pOnlyWords)
{
    if (!p_trie)
        return;

    if (p_trie->key == (int)'\0')
    {
        if (p_trie->equal)
        {
            if (pOnlyWords)
                printf("%s\n", (char *) p_trie->equal);
            else
            {
                padding ( ' ', depth*3 );
                printf("\\");
                printf("%s\n", (char *) p_trie->equal);
            }
        }
        if (p_trie->highr)
            tst_draw(p_trie->highr,   depth+1, pOnlyWords);
        if (p_trie->lower)
            tst_draw(p_trie->lower,   depth+1, pOnlyWords);

        return;
    }
    else
    {
        if (!pOnlyWords)
        {
            padding ( ' ', depth*3 );
            printf("%c\n", (char) p_trie->key);
        }
    }

    tst_draw(p_trie->highr,   depth+1, pOnlyWords);
    tst_draw(p_trie->equal,   depth+1, pOnlyWords);
    tst_draw(p_trie->lower,   depth+1, pOnlyWords);
}
```

False

True

Fig 1: Pre-order traversal of the tree (Root, Right, Middle, Left). Note: Changing to Root, Left, Middle, Right would give us an alphabetical ordered list in Fig 3

```
**************************************
Printing entire Ternary Search Tree
e
   f
      t
         r
            e
               i
                  u
                     e
                        \true
                  e
                     \trie
               e
                  \tree
      a
         i
            l
               s
                  e
                     \false
            l
               \fail
   x
      a
         m
            \exam
               p
                  l
                     e
                        \example
```

Fig 2: Pretty print Ternary Trie with leaf nodes

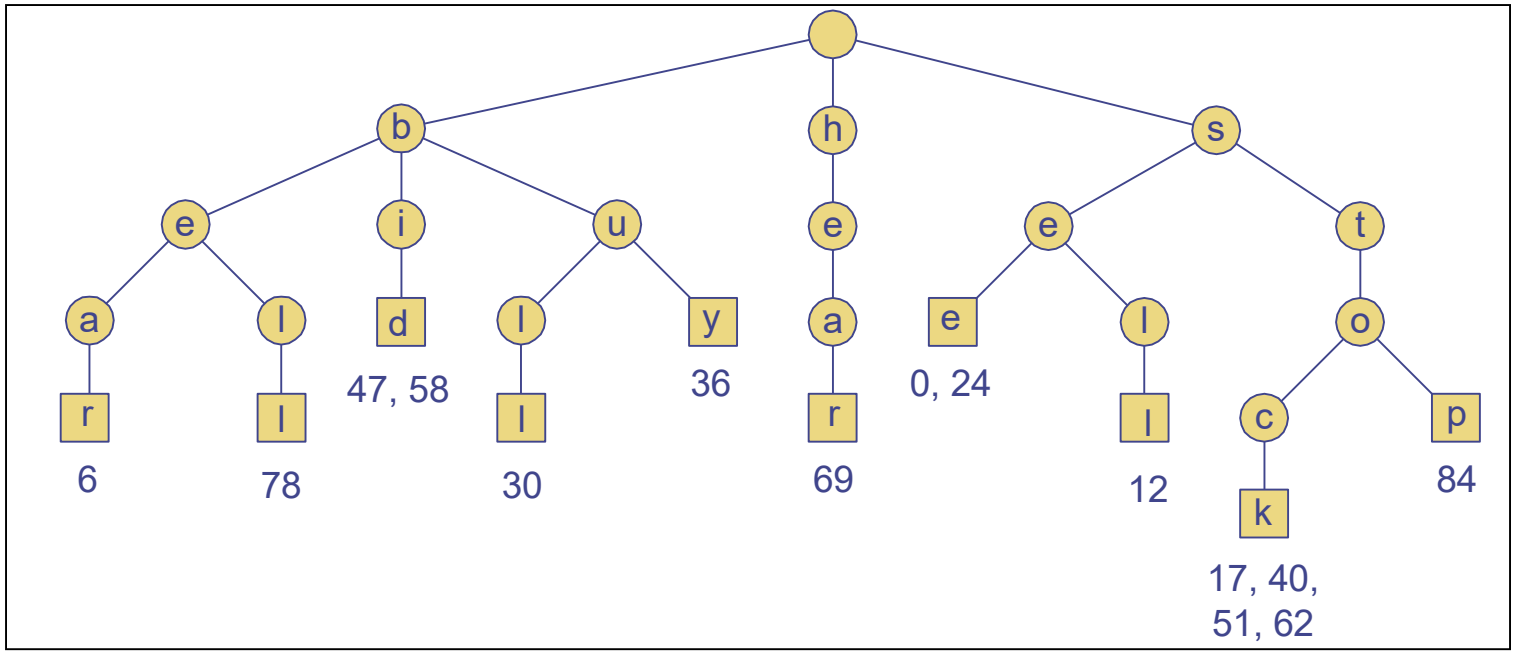
```
Printing all words sorted in Reverse alphabetical:-
true
trie
tree
false
fail
exam
example
```

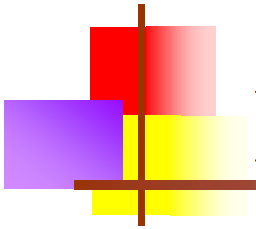
Fig 3: Traverse all words (Print in linear time)

Practical: Concordance

- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text

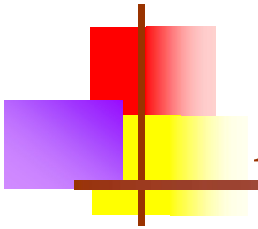
s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!		
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	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				





Practical: Spell Check

- How do you build a spell check function where the vocabulary is greater than the size of the computer.
  - TST
    - Low storage overhead.
    - Nearest neighbor search. Useful for correcting misspellings
    - Can print vocabulary in sorted order.



Appendix:

- Chapter 8, Section 1, Advanced Data Structures, Peter Brass.
- "Fast Algorithms for Sorting and Searching Strings" by Bentley & Sedgewick (1997). File:Bentley 1997.pdf
- http://c2.com/cgi/wiki?TernarySearchTree
- http://callisto.nsu.ru/documentation/CSIR/Algo/ternary_trees/Dr_%20Dobb's%20Journal%20April%201998%20Ternary%20SearchTrees.htm
- http://www.cs.princeton.edu/~rs/strings/