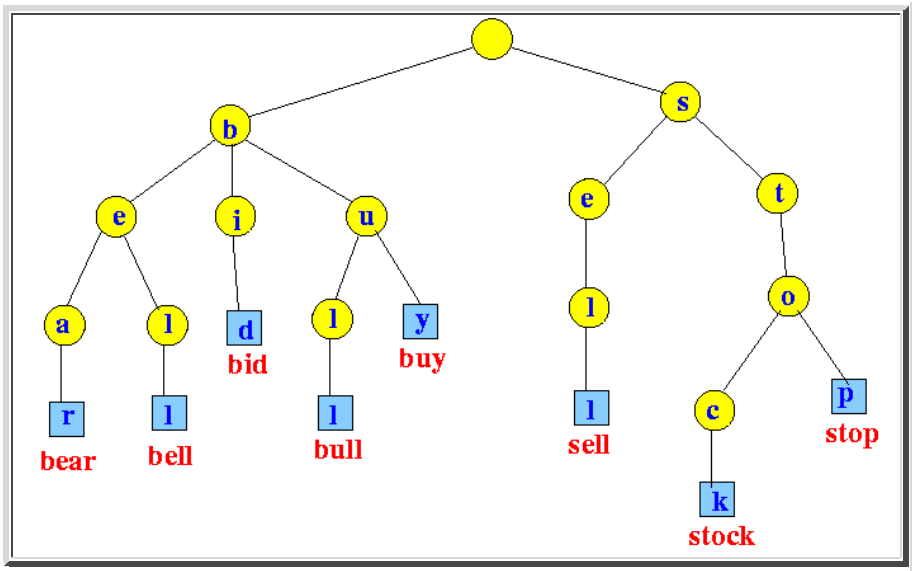
The *standard* trie

- **The *trie***
  - **Trie:**
    - The term **trie** comes from (**information**) **retrieval**.
    - Following the etymology, the inventor, **Edward Fredkin**, pronounces it as **"tree"**.
    - However, it is often pronounced as **"try"**.
  - **What is** a **trie**:
    - The ***trie* data structure (abstract data type)** is an **specialized (very efficient)** implementation of an **(ordered) index** for ***text based* keys**

- **The *standard* Trie data structure**
  - **Definitions:**
    - **Alphabet** = a **set** of ***characters***
    - Let **S** = a ***set*** of **s strings** (= **keys**) over an ***alphabet* Σ**
    - A **trie T** that store the **keys** in **S** is a structure where:
      - Each **node** of **T** (***except*** the **root node**) is **labeled** with a **character c ∈ Σ**
        - The **root node** has **no label** !!!
      - Each **internal node** of **T** can have **≤ |Σ| # of keys**
      - The **keys** are stored in **alphabetical order** inside an ***internal* node**
      - The **trie T** has **s *external* nodes**
        - Each ***external* node** is associated with **one string** in **S**

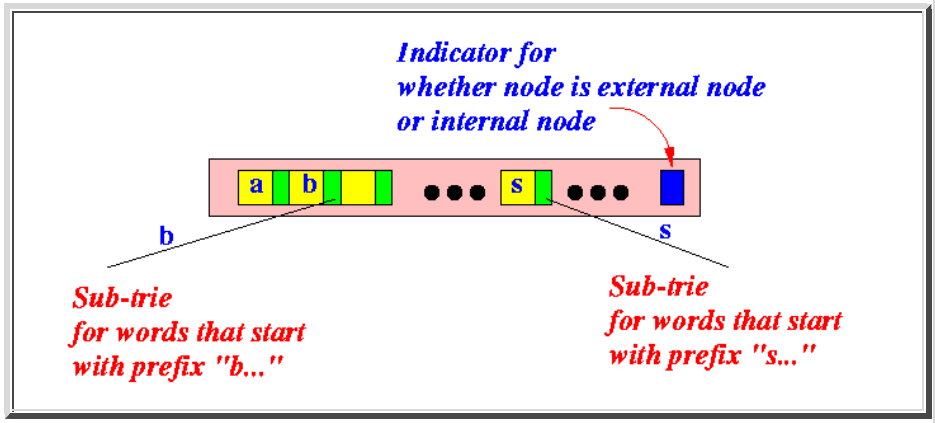
          (And **index (= location) information** are stored for these **strings**)
      - The ***path*** from the ***root node*** to an ***external* node** yields exactly **one string in S**
  - **Example:**
    - **S = { bear, bell, bid, bull, buy, sell, stock, stop }** (**s** = 8)

      Trie:



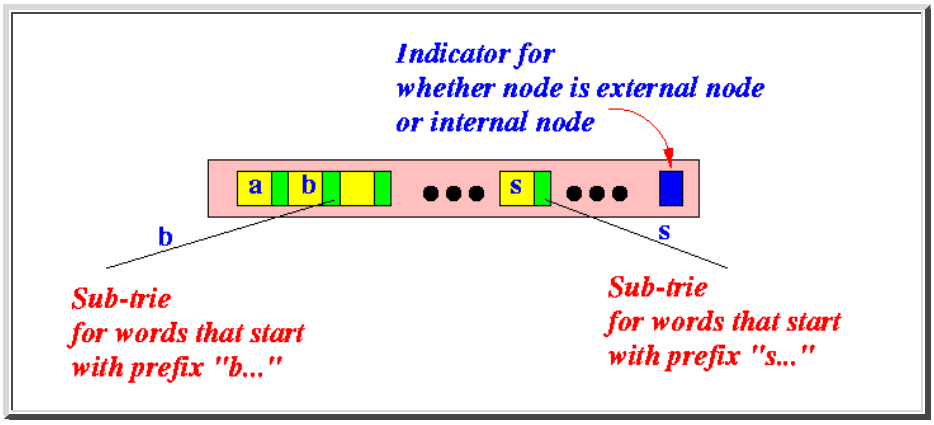
- **Note:**
  - There are **8** leaf nodes !!

- **How to *implementat* a trie**
  - There are **many ways** to **implement** a **trie**
    - Use an **array** of **references**
      - Each **array element** represent one **letter** of the **alphabet**
      - Each **array reference** will point to a **sub-trie** that corresponds to **strings** that ***starts*** with the **corresponding letter**

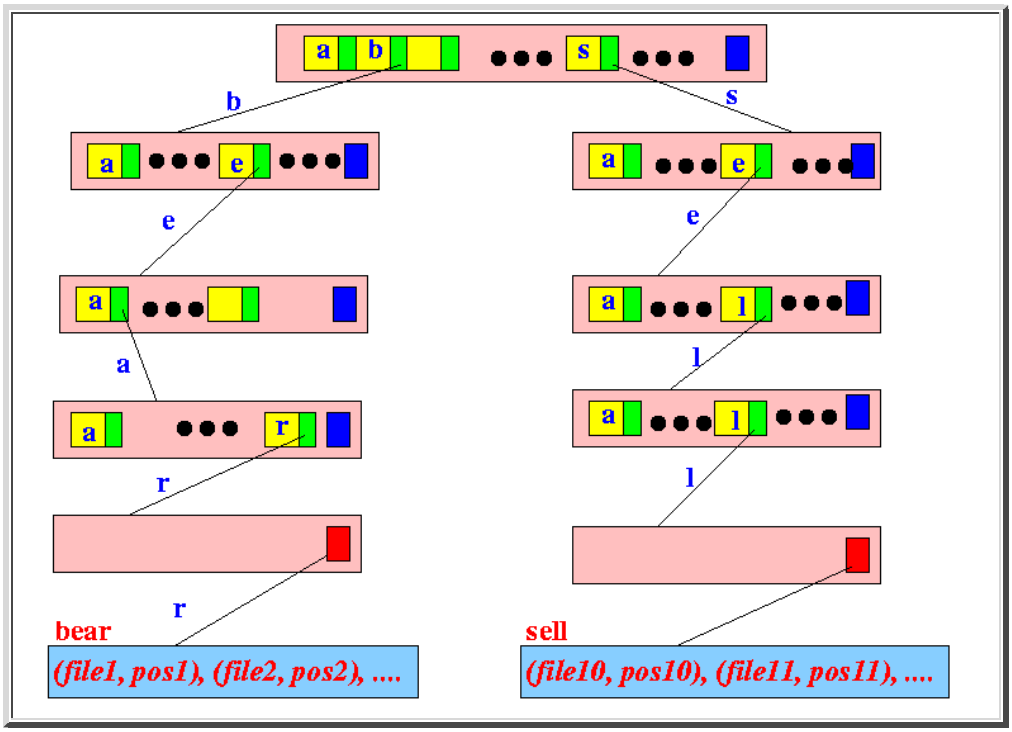


    - Use a **binary tree**
      - This implementation is called a ***bitwise* trie**

- **Implementing a *trie* using an *array* of references**
  - **Array implementation:**
    - If the **trie** is **dense *and/or* alphabet Σ** is **small** (e.g., **Σ = {a, b, c, ..., z}**), you can use an **array** to store the **labels**
    - **Node structure:**



Example:



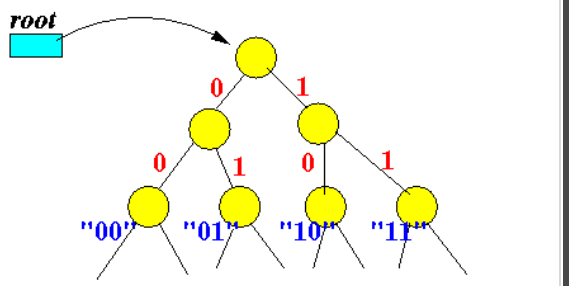
- **Implementing a *trie* using a *binary* tree**
  - The **binary tree implementation**:

- The **binary tree** implementation of a **trie** is known as a ***bitwise* trie**
- The **implementation** uses the ***alphabet***: **Σ = {0, 1}**
  - In other words, the implementation stores a **sequences of bits**
- The **keys** are **read** as a **sequence of bits**

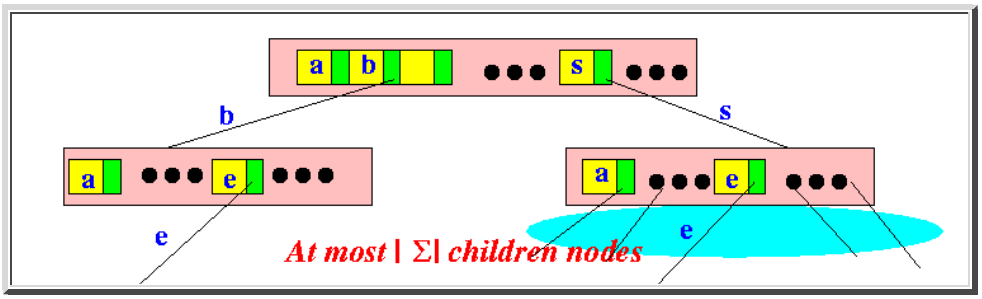
Example:

```
bear =  'b'    'e'    'a'    'r'
   = 01100010 01100101 01100001 01110010
```

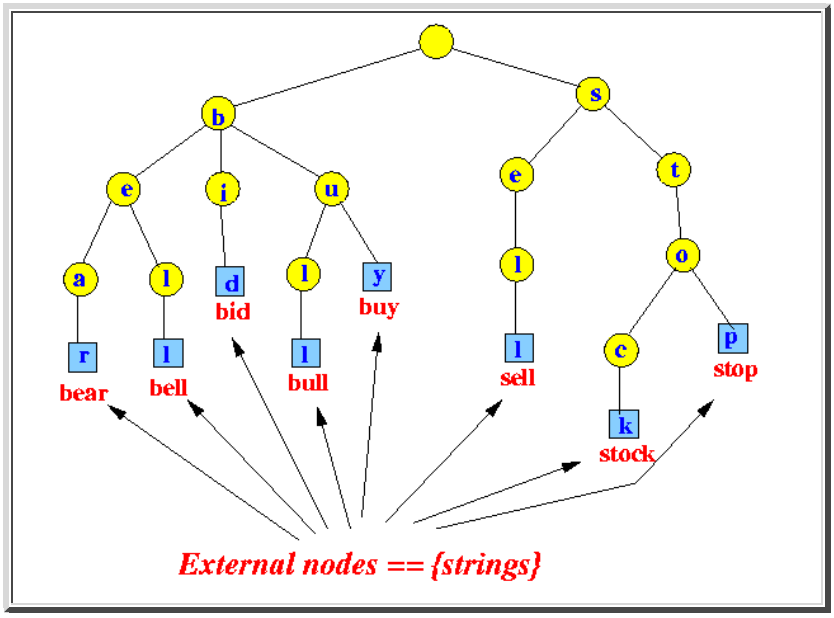
- A ***bitwise* trie** is a **binary tree**:



- **Structural properties of the *standard* trie**
  - **Properties of the *standard* trie:**
    - Every ***internal* node** has **≤ |Σ| children**

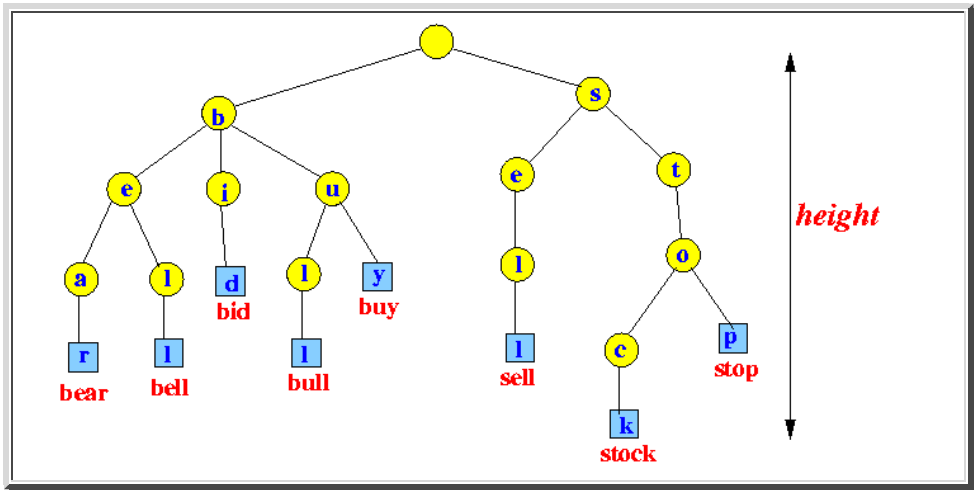


    - This follows from the **way** that the **trie** is **constructed**
    - The **trie T** on the **set S** with **s strings (keys)** has ***exactly* s *external* nodes**



- This is because a **path** from the ***root node*** to **one *external* node** corresponds to **1 key**

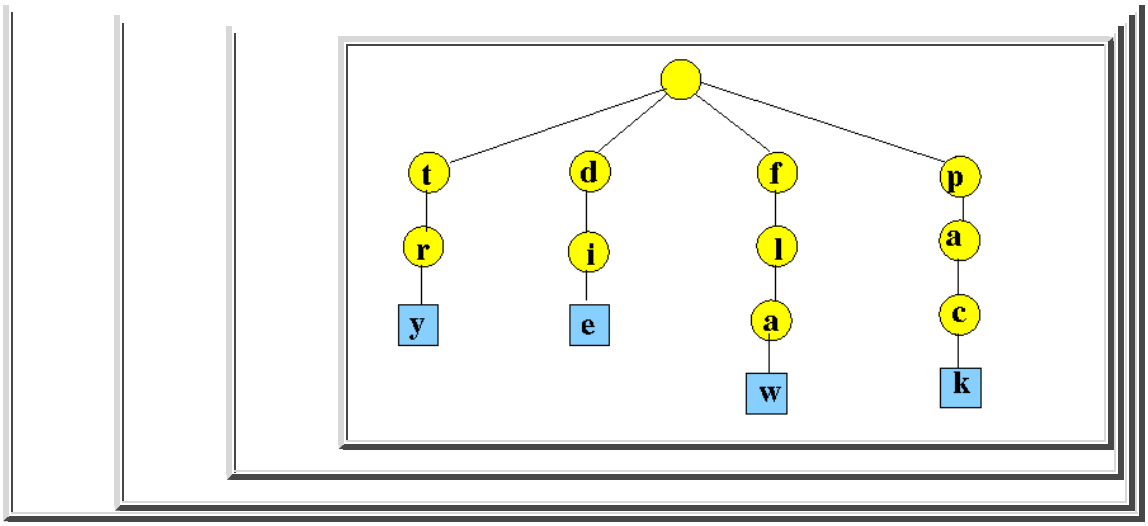
- The **height** of the **trie T** on the **set S** = the **length of the *longest* string ∈ S**



- This is because a **path** from the ***root node*** to **one *external* node** corresponds to **1 key**
- The **longest path** = **longest key ∈ S**

- The **number of nodes** in the **trie T** on the **set S** = **O(n)**, where **n** = **# characters in the strings ∈ S**
  - In the **worst case**, **every character** in the **keys** are ***different***

    E.g.:
    - **S = { try, die, flaw, pack }**

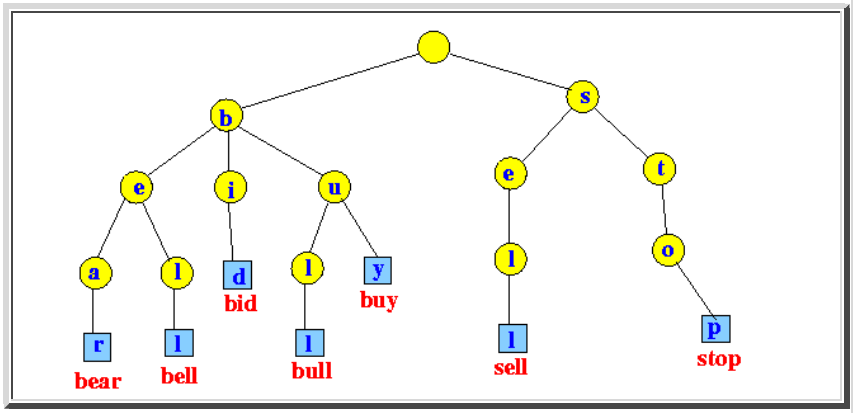


- **Inserting into a *standard* trie**
  - **High level description:**

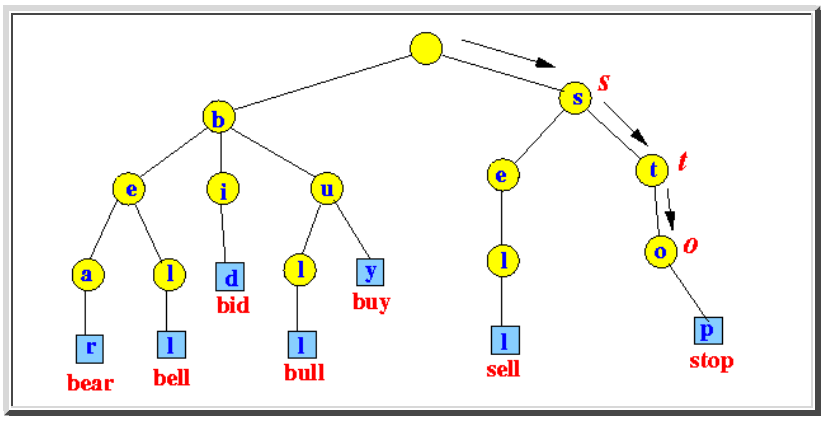
```
p = root;     // Start at the root

for ( each character c in the keyword ) do
{
  if ( c is already stored of the sub-trie )
    traverse to that subtrie
  else
    create a new node for c
}

insert value into leaf node
```

Example:

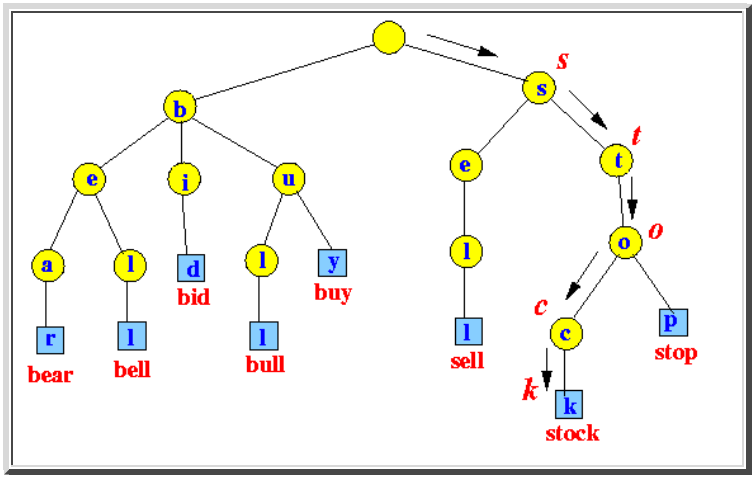
- Insert **stock** in this **trie**:



- **First** we traverse the **prefix** that is already **stored** in the trie: **"sto"**:



- **Then** we create **nodes** for letters are are **not** in the trie: **"ck"**:



- **Psuedo code:**

```
Value put( Key k, Value v )
{
  int i;
  TrieNode p;

  p = root;

  for ( i = 0; i < k.length(); i++ )
  {
    nextChar = k[i];

    if ( nextChar ∈ p.char[] )
    {
      nextLink = link associated with the char entry;
      p = nextLink;          // Traverse
    }
    else
    {
      p = new TrieNode();       // Create new node

      insert "nextChar" into p;
    }
  }

  /* ----------------------------------------------------
    When the while loop ends, we have found or created
```

```
    the external node for the key "k"
    ---------------------------------------------------------- */
  insert v into node p;
}
```

- **Advantages of Tries over an *ordinary* map**
  - **Advantages:**

    - **Looking up data** in a ***trie*** is in the **worst case** = **O(m)**, where **m = length of the key**
      - Look up in a **map** is **O(lg(n))** where **n = # entries** !!!

        So a **trie** has performance levels that is similar to a **hash table** !!!

    - Unlike a **hash table**, a **trie** can provide an **alphabetical *ordering*** of the **entries by key**

      (I.e., A **trie** implements an **ordered map** while a **hash table** cannot !)

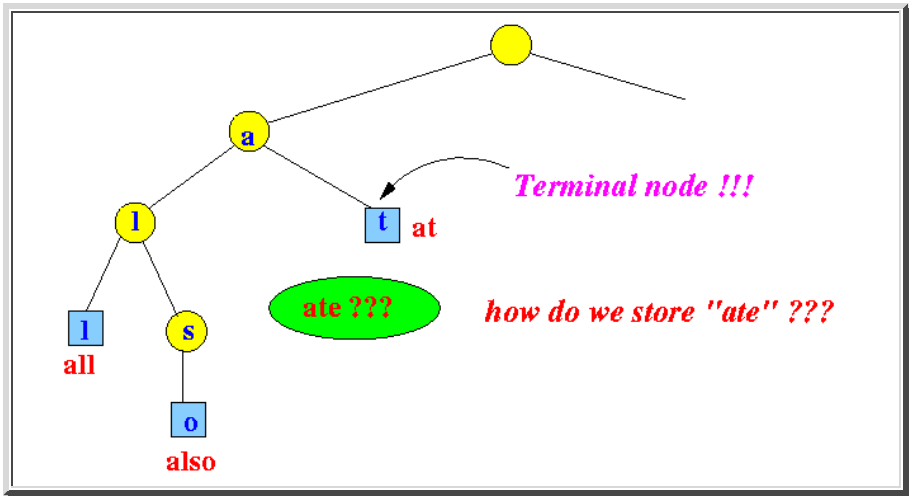
- **Handling keys that are *prefixes* of another key**
  - The **standard trie** has the **property** that:

    - ***Only*** the ***external*** **nodes** can store **information**

      (The **path** formed by the **internal nodes** represents the **key**)

  - When a **key (string)** is a **prefix** of **another key**, the **path** of the **first key** would **end** in an **internal node**

    Example: **at** and **ate**



  - **Solution:**

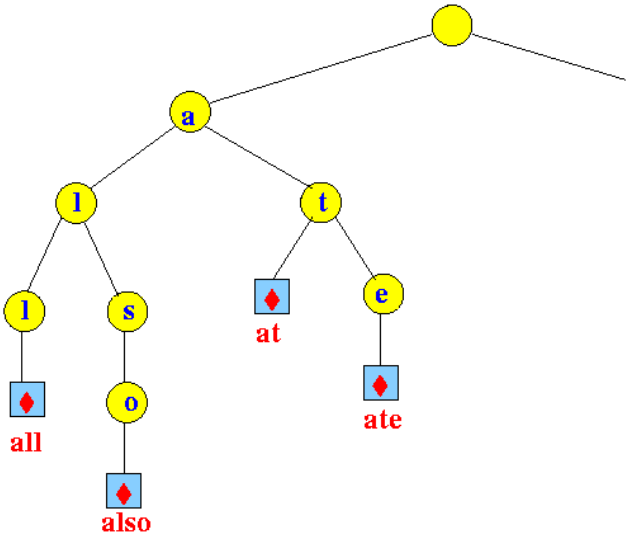
    - **Add** a **special termination symbol** ◊ to the **alphabet** Σ

- The **termination symbol** ◊ has the ***lower* value** in the **alphabet**

  I.e.,: **termination symbol** ◊ ***preceeds* *every* character** in the **alphabet Σ** !!

- We **append** the **termination symbol** ◊ to each **keyword** stored in the **trie**

Example:



- **Note:**
  - We **typically** use the **NUL character '\0'** as **termination symbol**

- **External material**
  - I found an implementation of **trie** here:
    - http://simplestcodings.blogspot.com/2012/11/trie-implementation-in-c.html

    and modified it to **print** the internal structure...

    Caveat:

    - The author of this program uses a ***non*-standard way** to represent nodes.

      (I don't want to go into the details of **his** implementation. I do like his code because it can be easily adapted to print out the structure of the **trie**)

  - **Example Program:** (Demo above code) ***Example***
    - The **trie.c** Prog file: click here
    - The **trie.h** Header file: click here
    - A test program **main.c** Prog file: click here

    How to run the program:

- **Right click** on link(s) and **save** in a scratch directory
- To compile: gcc -o main main.c trie.c
- To run: main

(*Edited*) Output:

```
(`*',0)  (`b',-1)  (`e',-1)  (`a',-1)  (`r',-1)  (`*',36)   "bear"

=========================================================
(`*',0)  (`b',-1)  (`e',-1)  (`a',-1)  (`r',-1)  (`*',36)   "bear"

                   (`l',-1)  (`l',-1)  (`*',14)   "bell"

=========================================================
(`*',0)  (`b',-1)  (`e',-1)  (`a',-1)  (`r',-1)  (`*',36)   "bear"

                   (`l',-1)  (`l',-1)  (`*',14)   "bell"

                   (`*',-1)  (`*',4)             "be"

=========================================================
(`*',0)  (`b',-1)  (`e',-1)  (`a',-1)  (`r',-1)  (`*',36)   "bear"

                   (`l',-1)  (`l',-1)  (`*',14)   "bell"

                   (`*',-1)  (`*',4)             "be"

            (`u',-1)  (`l',-1)  (`l',-1)  (`*',40)   "bull"

=========================================================
(`*',0)  (`b',-1)  (`e',-1)  (`a',-1)  (`r',-1)  (`*',36)   "bear"

                   (`l',-1)  (`l',-1)  (`*',14)   "bell"

                   (`*',-1)  (`*',4)             "be"

            (`u',-1)  (`l',-1)  (`l',-1)  (`*',40)   "bull"

                   (`y',-1)  (`*',40)            "buy"
```

- **An array implementaion of the standard trie**
  - I found an implementation of the **standard trie** that uses an **array** as described in the notes above:
    - http://cristibalas.wordpress.com/2008/05/11/generic-string-trie-implementation/

  - The **node structure** is as follows:

```
typedef struct sTRIE_NODE
{
  char is_final;       // 1 if the node is an external node
                       //   ==> void *data will contain data stored
                       // 0 if node is an internal node
                       //   ==> use nxt[charCode] to search further

  void *data;          // user data that is stored with this node
                       // Only use this field if  is_final == 1

  unsigned char childs; // number of childs. useful for fast deletion
                       // (If childs becomes 0, we delete this node)

  struct sTRIE_NODE *nxt[256]; // child pointers for every ASCII char !

} TRIE_NODE;
```

- Here is how you **search** the **trie**:

```
/* ================================================================
  trie_search_nodes( node, str, pos )

     Search for the string  str[pos...]
     in sub-trie rooted at node "node"
  ================================================================ */
int trie_search_nodes(TRIE_NODE *node, char *str, int pos)
{
  unsigned char ch = (unsigned char)str[pos]; // Use this character to search
                                  // in node "node"


  if( ch == '\0' )    // string is ended
  {
    if( node->is_final == 1 )
    {
      // We reached an external node ==> the string is found !

      crt_search_data = node->data;    // Get data assoc. with keyword
      return 1;                    // Return FOUND indication
    }
    else
    {
      return 0;                    // Return NOT FOUND indication
    }
  }
  else
  {
    if( node->nxt[ch] != NULL )        // More characters in string
    {
      return trie_search_nodes(node->nxt[ch], str, pos + 1);
                  // Search further - using next character position
    }
    else
    {
      return 0;                    // not found
    }
  }

}
```

- **Example Program:** (Demo above code) ***Example***
  - The **C** Prog file (implements the **trie**): click here
  - The **header** file: click here
