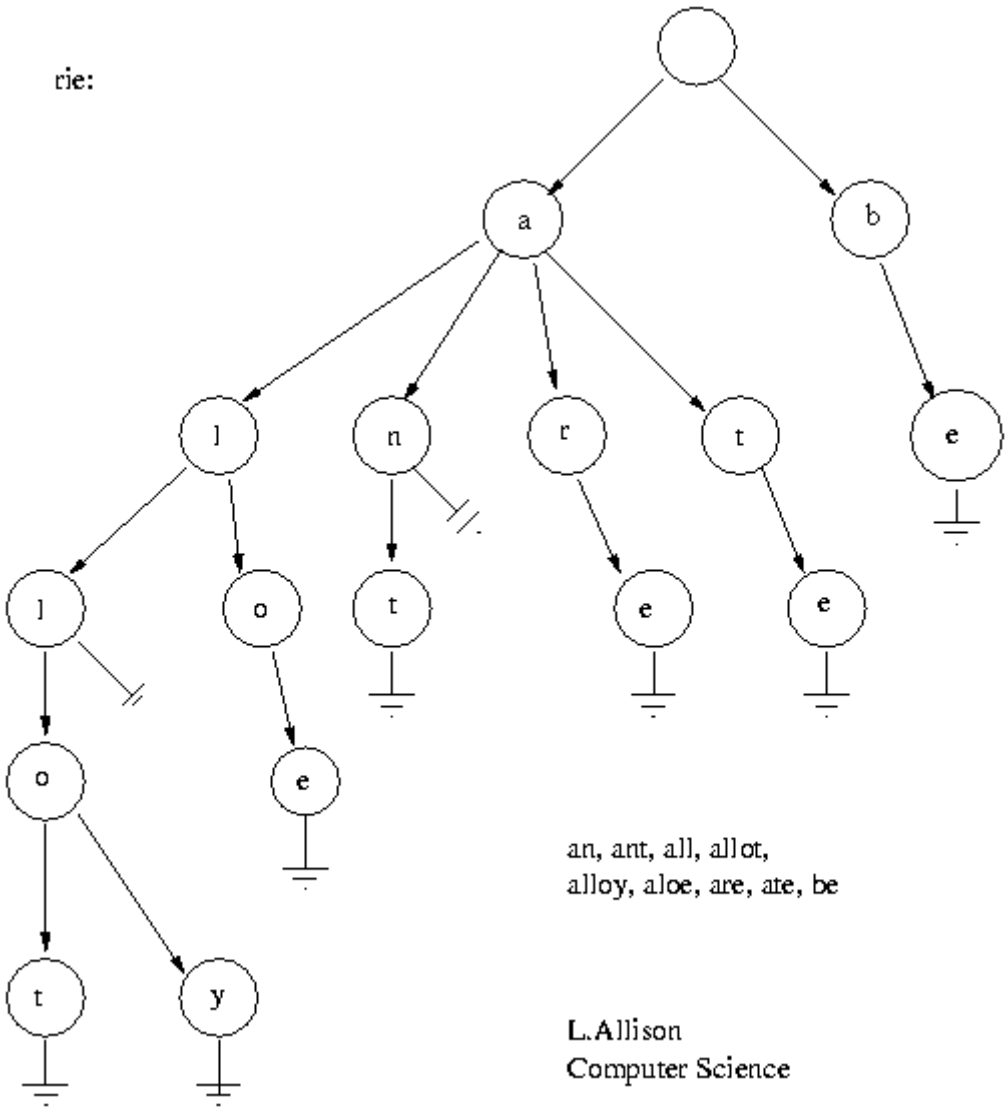
Дисциплина: *СД и ООП* **Тема:** *Префиксно дърво*
Специалност: *КН 1 курс, 3 и 4 групи* **Дата**: *02.06.2010 г.*

Retrieved from http://en.wikipedia.org/wiki/Trie: *In computer science , a **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.*



*The following are the main advantages of **trie**s over binary search trees:*

- *Looking up keys is faster. Looking up a key of length m takes worst case O(m) time. A BST takes O(log n) time, where n is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys. Also, the simple operations **trie**s use during lookup, such as array indexing using a character, are fast on real machines*
- ***Trie**s can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys*
- ***Trie**s help with longest-prefix matching , where we wish to find the key sharing the longest possible prefix of characters all unique*

*As mentioned, a **trie** has a number of advantages over binary search trees. A **trie** can also be used to replace a hash table, over which it has the following advantages:*

•*Looking up data in a **trie** is faster in the worst case, O(1) time, compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time*

•*There are no collisions of different keys in a **trie***

•*Buckets in a **trie** which are analogous to hash table buckets that store key collisions are only necessary if a single key is associated with more than one value*

•*There is no need to provide a hash function or to change hash functions as more keys are added to a **trie***

•*A **trie** can provide an alphabetical ordering of the entries by key*

***Trie**s do have some drawbacks as well:*

•***Trie**s can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory*

•*It is not easy to represent all keys as strings, such as floating point numbers, which can have multiple string representations for the same floating point number, e.g. 1, 1.0, 1.00, +1.0, etc.*

•***Trie**s are frequently less space-efficient than hash tables*

•*Unlike hash tables, **trie**s are generally not already available in programming language toolkits*

Applications:

•***Dictionary representation.** A common application of a **trie** is storing a dictionary, such as one found on a mobile telephone*

•***Trie**s are also well suited for implementing approximate matching algorithms, including those used in spell checking software*

***Algorithms:** The following pseudo-code represents the general algorithm for determining whether a given string is in a **trie**. Note that children is an array of a node's children; and we say that a "terminal" node is one which contains a valid word.*

```
function find(node, key) {
        if (key is an empty string) {  # base case
                return is node terminal?
        } else {  # recursive case
                c = first character in key  # this works because key is not empty
                tail = key minus the first character
                child = node.children[c]
                if (child is null) {  # unable to recurse, although key is non-empty
                        return false
                } else {
                        return find(child, tail)
                }
        }
}
```

***Sorting:** Lexicographic sorting of a set of keys can be accomplished with a simple **trie**-based algorithm as follows:*

1.*Insert all keys in a **trie***

2.*Output all keys in the **trie** by means of pre-order traversal, which results in output that is in lexicographically increasing order, or post-order traversal, which results in output that is in lexicographically decreasing order*

Задача: Реализация на *префиксно дърво:*

1.Класът **PrefixTree** е реализация на *префиксно дърво от ключове (обекти на класа **String**), съставени от малките латински букви*:

```
public class PrefixTree implements Tree {
        //Data
```

```
protected char content;
protected Object marker;   //The marker denotes the end of a key
protected PrefixTree[] child;

//Constructors
public PrefixTree() {
        content = ' ';        //Root contains blank - represents empty string
        marker = null;
        child = new PrefixTree[26];          //alphabet = "abcdefghijklmnopqrstuvwxyz"
}

public PrefixTree(int i) {
        content = (char)('a' + i);
        marker = null;
        child = new PrefixTree[26];          //alphabet = "abcdefghijklmnopqrstuvwxyz"
}

//Access methods
public char character() { return content; }
public boolean endOfKey() { return marker != null ; }
public Object data() { return marker; }
public PrefixTree[] child() { return child; }

//Protected methods
protected PrefixTree insert(String s) {
        PrefixTree current = this;
        PrefixTree result = null;

        if(s.length() == 0)                  //For empty String
                current.marker = new Boolean(true);

        System.out.println("     Inserted characters:");
        for(int i = 0; i < s.length(); i++) {
                if(current.child[s.charAt(i)-'a'] != null) {
                        current = current.child[s.charAt(i)-'a'];
                        System.out.print(current.content + "   ");
                }

                else {
                        result = new PrefixTree((int)(s.charAt(i)-'a'));
                        current.child[s.charAt(i)-'a'] = result;
                        current = current.child[s.charAt(i)-'a'];
                        System.out.print(current.content + "   ");
                }

                if(i == s.length()-1) {
                        current.marker = new Boolean(true);
                        result = current;
                }
        }

        System.out.println("\nFinished inserting the word: " + s + "\n");
        return result;
}

protected PrefixTree search(String s) {
        PrefixTree current = this;
        System.out.println("\n     Searching for string: " + s);

        while(current != null) {
                for(int i = 0; i < s.length(); i++) {
                        if(current.child[s.charAt(i)-'a'] == null) {
                                System.out.println("Cannot find string: " + s);
                                return null;
                        }
```

```
                        else {
                                current = current.child[s.charAt(i)-'a'];
                                System.out.println("Found character: " + current.content);
                        }
                }

                if (current.marker != null) {
                        System.out.println("Found string: " + s);
                        return current;
                }
                else {
                        System.out.println("Cannot find string: " + s +"(only present as a substring)");
                        return null;
                }
        }

        return null;
}

//Tree methods implementation
public void add(Object value) { insert((String)value); }
public boolean contains(Object value) { return search((String)value) != null; }
public boolean isEmpty() { return size() == 0; }

public int height() {
        PrefixTree[] child = child();
        int max = -1;
        for(int i = 0; i < child.length; i++) {
                PrefixTree t = child[i];
                if(t != null) {
                        int h = t.height();
                        if(max < h) max = h;
                }
        }
        return max + 1;
}

public int numberOfLiefs() { throw new UnsupportedOperationException(); }
public Object clone() { throw new UnsupportedOperationException(); }
public Object remove(Object value) { throw new UnsupportedOperationException(); }
public int size() { throw new UnsupportedOperationException(); }
public void clear() { throw new UnsupportedOperationException(); }

public java.util.Iterator iterator() { return treePreorderIterator(); }

public String toString() {
        String result = "";
        java.util.Iterator<String> it = iterator();
        while(it.hasNext())
                result = result + it.next() + "\n";
        return result;
}

//Iterators
public java.util.Iterator treeBreadthFirstIterator() { return new PrefixTreeBreadthFirstIterator(this); }
public java.util.Iterator treePreorderIterator() { return new PrefixTreePreorderIterator(this); }

//Other methods
public String characters() { throw new UnsupportedOperationException(); }
}
```

2.Класът **PrefixTreePreorderIterator** представя итератор за *обхождане на ключовете на префиксно дърво (низходящо обхождане)*:

```
import java.util.*;

public class PrefixTreePreorderIterator implements Iterator<String> {
	//Data
	LinkedList<String> list;
	Iterator<String> it;

	//Constructor
	public PrefixTreePreorderIterator(PrefixTree tree) {
		list = new LinkedList<String>();
		preorder(tree,"");
		reset();
	}

	//Private method
	private void preorder(PrefixTree tree,String s) {
		//Root visit
		s = s + tree.character();
		if(tree.endOfKey()) {
			if(s.length() == 0) list.addLast("");
			else list.addLast(s.substring(1));
		}

		//Subtrees visit
		PrefixTree[] child = tree.child();
		for(int i = 0; i < child.length; i++) {
			tree = child[i];
			if(tree != null)
				preorder(tree,s);
		}
	}

	//Iterator methods
	public boolean hasNext() { return it.hasNext(); }
	public String next() { return it.next(); }
	public void reset() { it = list.iterator(); }
	public void remove() { it.remove(); }
}
```

3.Да се реализира клас **PrefixTreeBreadthFirstIterator** за *обхождане на символите на префиксно дърво по нива*

Тестване: Класът **TestPrefixTree** чете думите an, ant, all, allor, alloy, aloe, are, ate и be от текстов файл testPrefixTreeData.txt и създава префиксно дърво, представено на стр.1. Обхожда дървото с итераторите от т.2 и т.3 и записва резултата във файла testPrefixTreeResult.txt.