Dictionary: design choices

R. Inkulu
http://www.iitg.ac.in/rinkulu/

Introduction

```
typedef struct {
   char *word;
   char *meaning;
} Entry;
```

- assume that each word has only one meaning
- intend to build a data structure to store the collection of word-meaning pairs

  objective: faster insert, delete, and accessing of words and their meanings

An array of entries

```
Entry entryTab[2000];   //entryTab[] is global

int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- disadv: all elements of *Entry*[] may not be used; hence, waste of space

An array of pointers to entries

```
Entry *entryTab[2000];  //entryTab[] is global

int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- disadv: number of word-meaning pairs that can be stored needs to be decided at pre-compile-time

A dynamic array of entries

```
Entry *entryTab;  //entrytab is global

int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- disadv: contiguous allocation of huge number of bytes; cost associated with realloc

A dynamic array of pointers to entries

```
Entry **entryTab;  //entrytab is global

int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- adv: need based allocation of objects of Entry

With a linked list

```
Node *entryNode;  //entryNode is global
   //entryNode points to a Node of linked-list type

int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- disadv: worst-case search time grows with the list size

Chained hash table: a collection of linked lists

```
Node *hashtab[26];  //hashtab[] is global
   //a member of an instance of Node points to
       //an instance of Entry

int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- a given word is placed in one among the twenty six linked-lists based on its first character

- disadv: considering that many words start with few specific characters (ex. a, s, t), the distribution of words to groups could be skewed

Chained hash table: more reasonable hash function

```
Node *hashTab[1000];  //hashTab[] is global

int getHashValue(const char *word);
int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
int remove(const char *word);
```

- compute hash values based on the ASCII values of all the characters of a given word (sophisticated hash functions are not part of this course)

Worst-case time complexity with chained hash table implementation

letting n be the number of words stored in the hash table,

- getMeaning $O(n)$

  given that the used hash function is just a *heuristic*

- remove $O(n)$

- insert $O(1)$

- getHashValue $O(1)$

  assuming that the number of characters in a word are $O(1)$

Using a binary search tree (a.k.a. BST)

```
struct TreeNodeA {
    Entry *entryObj;
    struct TreeNodeA *left;
    struct TreeNodeA *right; };
typedef struct TreeNodeA TreeNode;
TreeNode *rootNode;  //rootNode is global
int insert(const char *word, const char *meaning);
const char *getMeaning(const char *word);
void printAll(TreeNode *node);
   //inorder, preorder, and postoder
```

- at most two *child nodes* for any node v, one is termed as the *left child* of v and the other *right child* of v
- *key* corresp. to left (resp. right) subtree of any node v is less (resp. greater) than the key of v
- *root node*; *levels*; *height*
- printing in *inorder* yields *sorted* listing of dictionary contents

assume that there are no removal of entries (no time to cover remove operation)

Worst-case time complexity with BST implementation

letting n be the number of words stored in BST,

- each operation takes $O(n)$ time in the worst-case
- however, given a balanced-BST, each operation takes only $O(\lg n)$ time

  in other words, $O(\lg n)$ is achievable provided that if we can *maintain the balance* while inserting and deleting

  → doable! let us save "how?" to another course :-)

homework: analyze the worst-case time complexity of each operation of every implementation of Dictionary
homework: redesign appropriate data structures and the code to take into account that each word may have more than one meaning