Ush 1 Single linked list

// Program Dictionary

// Program that uses linked list to maintain a dictionary

#include <iostream.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

#include <string.h>

#include <ctype.h>

class dict

{

private :

struct node

{

char data[20] ;

char m[100] ;

int mcount ;

struct node \* link ;

} \*dic[22] ;

public :

dict( ) ;

void store ();

int search ( char \* );

void show( ) ;

void deldic( ) ;

} ;

// initialises data member

dict :: dict()

{

for ( int i = 0 ; i < 22 ; i++ )

dic[i] = NULL ;

}

// stores word with their meanings in the dictionary

void dict :: store ( )

{

char \* str,char \* meaning

int i, j = toupper ( str[0] ) - 65 ;

node \*r, \*temp = dic[j], \*q ;

char ch = 'y' ;

q = new node ;

strcpy ( q -> data, str ) ;

strcpy ( q -> m, meaning ) ;

q -> link = NULL ;

//question refers to here

}

// searches for given word in dictionary

int dict :: search ( char \*str )

{

struct node \*n ;

char temp1[20] ;

char temp2[20] ;

int i ;

n = dic[toupper ( str[0] ) - 65] ;

strcpy ( temp2, str ) ;

strupr ( temp2 ) ;

while ( n != NULL )

{

strcpy ( temp1, n -> data ) ;

if ( strcmp ( strupr ( temp1 ), temp2 ) == 0 )

{

cout << "\n" << n -> data << "\t\t" << n -> m ;

}

n = n -> link ;

}

return 0 ;

}

// displays word and its meaning

void dict :: show( )

{

node \*n ;

int i, j ;

cout << "Word\t\tMeaning\n" ;

for ( i = 0 ; i <= 30 ; i++ )

cout << "-" ;

for ( i = 0 ; i <= 22 ; i++ )

{

n = dic[i] ;

while ( n != NULL )

{

cout << "\n" << n -> data << "\t\t" << n -> m ;

n = n -> link ;

}}}

void main( )

{

char word[20] ;

int ch ;

int i ;

dict d ;

char temp ;

clrscr( ) ;

while ( 1 )

{

clrscr( ) ;

cout << "\n\t\tDictionary\n" ;

cout << "\t\t1.Search Word.\n" ;

cout << "\t\t2.Show Dictionary.\n" ;

cout << "\t\t0.Exit." ;

cout << "\n\n\t\tYour Choice " ;

cin >> ch ;

switch ( ch )

{

case 1 :

cout << "\nEnter the word to search : " ;

cin >> word ;

i = d.search ( word ) ;

if ( ! i )

cout << "Word does not exists." ;

cout << "\nPress any key to continue..." ;

cin >> temp ;

break ;

case 2 :

d.show( ) ;

cout << "\nPress any key to continue..." ;

cin >> temp ;

break ;

case 0 :

exit ( 0 ) ;

break ;

default :

cout << "\nWrong Choice" ;

cout << "\nPress any key to continue..." ;

cin >> temp ;

}

}

}

//Complexity: O(n2)

Ush 1- Binary Tree

#include <stdio.h>

#include<iostream.h>

#include <conio.h>

#include <string.h>

#include <stdlib.h>

#include<dos.h>

#define LEFT 1

#define RIGHT 2

struct node

{

char word[20],meaning[100];

node \*left,\*right;

};

node \*maketree(char[],char[]);

node\* treefromfile();

void filefromtree(node\*);

void addword(node\*,char[],char[]);

void seperateword(char[],char[],char[]);

void displayall(node\*);

node\* bsearch(node\*,char[]);

void showmenu();

FILE \*file\_ptr;

void prog()

{

clrscr();

char word[20],meaning[100];

int menuchoice;

node \*temp;

temp=treefromfile();

if(temp==NULL)

{

printf("

File does not exist or dictionary is empty...");

getch();

}

while(1)

{

clrscr();

showmenu();

scanf("

%d",&menuchoice);

switch(menuchoice)

{

case 1:printf("

Enter word : ");

scanf("%s",word);

printf("

Enter meaning : " );

flushall();

gets(meaning);

if(temp==NULL)

temp=maketree(word,meaning);

else

addword(temp,word,meaning);

break;

case 2:if(temp==NULL)

printf("

The dictionary is empty...");

else

{

printf("

Find meaning of : ");

flushall();

gets(word);

node \*t;

t=bsearch(temp,word);

if(t==NULL)

printf("

Word not found...");

else

{

printf("

%s : ",t->word);

puts(t->meaning);

}

}

getch();

break;

case 3:if(temp==NULL)

printf("

Dictionary is empty...");

else

displayall(temp);

getch();

break;

case 4:filefromtree(temp);

exit(1);

break;

default:cout<<"

Enter Again";

delay(1000);

prog();

break;

}

}

}

void showmenu()

{

printf("

COMPUTER DICTIONARY");

printf("

[1]. Add a word.");

printf("

[2]. Find meaning.");

printf("

[3]. Display all.");

printf("

[4]. Save and Close.

Enter Choice");

}

node\* treefromfile()

{

node \*ptree=NULL;

char word[20],meaning[100],str[120],\*i;

int flags=0;

file\_ptr=fopen("C:\dict.anu","r");

if(file\_ptr==NULL)

ptree=NULL;

else

{

while(!feof(file\_ptr))

{

i=fgets(str,120,file\_ptr);

if(i==NULL)

break;

seperateword(str,word,meaning);

if(flags==0)

{

ptree=maketree(word,meaning);

flags=1;

}

else

addword(ptree,word,meaning);

}

fclose(file\_ptr);

}

return ptree;

}

node\* maketree(char w[],char m[])

{

node \*p;

p=new node;

strcpy(p->word,w);

strcpy(p->meaning,m);

p->left=NULL;

p->right=NULL;

return p;

}

void seperateword(char str[],char w[],char m[])

{

int i,j;

for(i=0;str[i]!=' ';i++)

w[i]=str[i];

w[i++]=NULL; //Append the null and skip the space.

for(j=0;str[i]!='

';i++,j++)

{

m[j]=str[i];

}

m[j]=NULL;

}

void addword(node \*tree,char word[],char meaning[])

{

node \*p,\*q;

p=q=tree;

while(strcmp(word,p->word)!=0 && p!=NULL)

{

q=p;

if(strcmp(word,p->word)<0)

p=p->left;

else

p=p->right;

}

if(strcmp(word,q->word)==0)

{

printf("

This word already exists...");

delay(1000);

}

else if(strcmp(word,q->word)<0)

q->left=maketree(word,meaning);

else

q->right=maketree(word,meaning);

}

node\* bsearch(node \*tree,char word[])

{

node \*q;

q=tree;

while(q!=NULL)

{

//p=q;

if(strcmp(word,q->word)<0)

q=q->left;

else if(strcmp(word,q->word)>0)

q=q->right;

if(strcmp(word,q->word)==0)

break;

}

return q;

}

void filefromtree(node \*tree)

{

void travandwrite(node\*);

file\_ptr=fopen("C:\dict.anu","w");

if(file\_ptr==NULL)

{

printf("

Cannot open file for writing data...");

}

else //if(tree==NULL)

{

if(tree!=NULL)

{

travandwrite(tree);

}

fclose(file\_ptr); //Close the file anyway.

}

}

void travandwrite(node \*tree)

{

if(tree!=NULL)

{

fprintf(file\_ptr,"%s %s

",tree->word,tree->meaning);

travandwrite(tree->left);

travandwrite(tree->right);

}

}

void displayall(node \*tree)

{

if(tree!=NULL)

{

displayall(tree->left);

printf("%s : %s

",tree->word,tree->meaning);

displayall(tree->right);

}

}

void intro()

{

int i;

clrscr();

gotoxy(20,20);

cout<<"DICTIONARY LOADING";

for(i=0;i<50;i++)

{

gotoxy(15+i,21);

cout<<"???";

gotoxy(20,22);

cout<<2\*i<<"% completed";

delay(150);

}

gotoxy(20,20);

cout<<"DICTIONARY LOADING COMPLETED";

clrscr();

}

void main()

{

clrscr();

intro();

prog();

}

//Complexity : O(n2)

Ush 1- Hash Table

#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

# define max 10

typedef struct list

{

int data;

struct list \*next;

} node\_type;

node\_type \*ptr[max],\*root[max],\*temp[max];

class Dictionary

{

public:

int index;

Dictionary();

void insert(int);

void search(int);

void delete\_ele(int);

};

Dictionary::Dictionary()

{

index=-1;

for(int i=0; i<max; i++)

{

root[i]=NULL;

ptr[i]=NULL;

temp[i]=NULL;

}

}

void Dictionary::insert(int key)

{

index=int(key%max);

ptr[index]=(node\_type\*)malloc(sizeof(node\_type));

ptr[index]->data=key;

if(root[index]==NULL)

{

root[index]=ptr[index];

root[index]->next=NULL;

temp[index]=ptr[index];

}

else

{

temp[index]=root[index];

while(temp[index]->next!=NULL)

temp[index]=temp[index]->next;

temp[index]->next=ptr[index];

}

}

void Dictionary::search(int key)

{

int flag=0;

index=int(key%max);

temp[index]=root[index];

while(temp[index]!=NULL)

{

if(temp[index]->data==key)

{

cout<<"\nSearch key is found!!";

flag=1;

break;

}

else temp[index]=temp[index]->next;

}

if (flag==0)

cout<<"\nsearch key not found.......";

}

void Dictionary::delete\_ele(int key)

{

index=int(key%max);

temp[index]=root[index];

while(temp[index]->data!=key && temp[index]!=NULL)

{

ptr[index]=temp[index];

temp[index]=temp[index]->next;

}

ptr[index]->next=temp[index]->next;

cout<<"\n"<<temp[index]->data<<" has been deleted.";

temp[index]->data=-1;

temp[index]=NULL;

free(temp[index]);

}

main()

{

int val,ch,n,num;

char c;

Dictionary d;

do

{

cout<<"\nMENU:\n1.Create";

cout<<"\n2.Search for a value\n3.Delete an value";

cout<<"\nEnter your choice:";

cin>>ch;

switch(ch)

{

case 1:

cout<<"\nEnter the number of elements to be inserted:";

cin>>n;

cout<<"\nEnter the elements to be inserted:";

for(int i=0; i<n; i++)

{

cin>>num;

d.insert(num);

}

break;

case 2:

cout<<"\nEnter the element to be searched:";

cin>>n;

d.search(n);

case 3:

cout<<"\nEnter the element to be deleted:";

cin>>n;

d.delete\_ele(n);

break;

default:

cout<<"\nInvalid Choice.";

}

cout<<"\nEnter y to Continue:";

cin>>c;

}

while(c=='y');

getch();

}

//Complexity: O(n)

Ush 2 Web Crawler

# -\*- coding: utf-8 -\*-

# filename: crawler.py

import sqlite3

import urllib2

from HTMLParser import HTMLParser

from urlparse import urlparse

class HREFParser(HTMLParser):

"""

Parser that extracts hrefs

"""

hrefs = set()

def handle\_starttag(self, tag, attrs):

if tag == 'a':

dict\_attrs = dict(attrs)

if dict\_attrs.get('href'):

self.hrefs.add(dict\_attrs['href'])

def get\_local\_links(html, domain):

"""

Read through HTML content and returns a tuple of links

internal to the given domain

"""

hrefs = set()

parser = HREFParser()

parser.feed(html)

for href in parser.hrefs:

u\_parse = urlparse(href)

if href.startswith('/'):

# purposefully using path, no query, no hash

hrefs.add(u\_parse.path)

else:

# only keep the local urls

if u\_parse.netloc == domain:

hrefs.add(u\_parse.path)

return hrefs

class CrawlerCache(object):

"""

Crawler data caching per relative URL and domain.

"""

def \_\_init\_\_(self, db\_file):

self.conn = sqlite3.connect(db\_file)

c = self.conn.cursor()

c.execute('''CREATE TABLE IF NOT EXISTS sites

(domain text, url text, content text)''')

self.conn.commit()

self.cursor = self.conn.cursor()

def set(self, domain, url, data):

"""

store the content for a given domain and relative url

"""

self.cursor.execute("INSERT INTO sites VALUES (?,?,?)",

(domain, url, data))

self.conn.commit()

def get(self, domain, url):

"""

return the content for a given domain and relative url

"""

self.cursor.execute("SELECT content FROM sites WHERE domain=? and url=?",

(domain, url))

row = self.cursor.fetchone()

if row:

return row[0]

def get\_urls(self, domain):

"""

return all the URLS within a domain

"""

self.cursor.execute("SELECT url FROM sites WHERE domain=?", (domain,))

# could use fetchone and yield but I want to release

# my cursor after the call. I could have create a new cursor tho.

# ...Oh well

return [row[0] for row in self.cursor.fetchall()]

class Crawler(object):

def \_\_init\_\_(self, cache=None, depth=2):

"""

depth: how many time it will bounce from page one (optional)

cache: a basic cache controller (optional)

"""

self.depth = depth

self.content = {}

self.cache = cache

def crawl(self, url, no\_cache=None):

"""

url: where we start crawling, should be a complete URL like

'http://www.intel.com/news/'

no\_cache: function returning True if the url should be refreshed

"""

u\_parse = urlparse(url)

self.domain = u\_parse.netloc

self.content[self.domain] = {}

self.scheme = u\_parse.scheme

self.no\_cache = no\_cache

self.\_crawl([u\_parse.path], self.depth)

def set(self, url, html):

self.content[self.domain][url] = html

if self.is\_cacheable(url):

self.cache.set(self.domain, url, html)

def get(self, url):

page = None

if self.is\_cacheable(url):

page = self.cache.get(self.domain, url)

if page is None:

page = self.curl(url)

else:

print "cached url... [%s] %s" % (self.domain, url)

return page

def is\_cacheable(self, url):

return self.cache and self.no\_cache and not self.no\_cache(url)

def \_crawl(self, urls, max\_depth):

n\_urls = set()

if max\_depth:

for url in urls:

# do not crawl twice the same page

if url not in self.content:

html = self.get(url)

self.set(url, html)

n\_urls = n\_urls.union(get\_local\_links(html, self.domain))

self.\_crawl(n\_urls, max\_depth-1)

def curl(self, url):

"""

return content at url.

return empty string if response raise an HTTPError (not found, 500...)

"""

try:

print "retrieving url... [%s] %s" % (self.domain, url)

req = urllib2.Request('%s://%s%s' % (self.scheme, self.domain, url))

response = urllib2.urlopen(req)

return response.read().decode('ascii', 'ignore')

except urllib2.HTTPError, e:

print "error [%s] %s: %s" % (self.domain, url, e)

return ''

//Complexity: O(n)