CS29206 Systems Programming Laboratory, Spring 2022–2023

Programming Assignment 2

Last date of submission: 12–February–2023

Topic: Building projects with makefiles

Overview

This assignment deals with the development of some utilities on English words, and has three components.

- **A library for storing and working with lists of strings:** A trie is a popular and efficient data structure for this purpose. Prepare a shared library `libtrie.so` that implements some basic trie operations (initialize, search, insert, and list). Here, all strings consist only of lower-case roman letters a–z.
- **A library for some word-based utilities:** Another shared library `libwordutils.so` is to be designed. This library uses a trie as a dictionary (to be interpreted as a word list) of English words (only those with lower-case letters). The dictionary is to be populated by reading the words from a text file `words.txt` to be supplied to you. The library should implement functions to investigate possibilities of new word generation by adding letters to the beginning and to the end of a string, and also by permuting the letters of a string. The two libraries should be built independently of one another.
- **Two application programs:** The first application program `maxchain.c` is intended to find a longest chain of valid English words such that each word in the chain is obtained by appending a single letter at one of the two ends of the previous word in the chain. For example, the following chain consists of six valid English words. All single letters are considered valid words.

  o → or → ore → core → score → scored

  Here is a longer chain.

  t → at → ate → rate → irate → pirate → pirates

  The second application program `subwords.c` reads a string S from the user. Let l be the length of S. The program finds and prints all $(l-1)$-letter words that can be formed from the letters of S. For example, if the user enters `osptp`, then the valid 4-letter English subwords are

  opts, post, pots, spot, stop, tops, pops.

  Words like *toss* cannot be built from `osptp` (the input supplies only one s, whereas *toss* uses two).

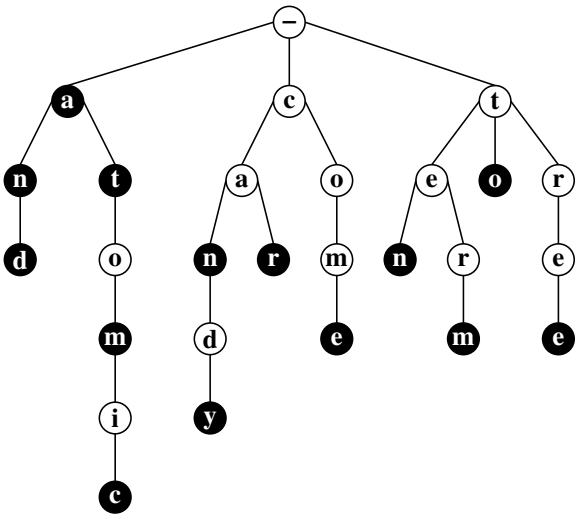
The files in this assignment should be organized as follows. A brief description is provided here. Detailed explanations appear in the rest of this document. Stick to this organization strictly from the very beginning.

```
A2/                         [Project root directory]
├── makefile                [Top–level makefile]
├── dict/                   [Subdirectory storing a default word list]
│   └── words.txt           [This file will be supplied to you]
├── trie/                   [Subdirectory for the trie library]
│   ├── makefle             [For building libtrie.so]
│   ├── trie.h
│   └── trie.c
├── util/                   [Subdirectory for the wordutils library]
│   ├── makefile            [For building libwordutils.so]
│   ├── dicttype.h
│   ├── dictload.h
│   ├── dictload.c
│   ├── dictutils.h
│   ├── dictutils.c
│   └── wordutils.h
└── apps/                   [Subdirectory for the application programs]
    ├── makefile            [For compiling and running the application programs]
    ├── maxchain.c          [First application program]
    └── subwords.c          [Second application program]
```

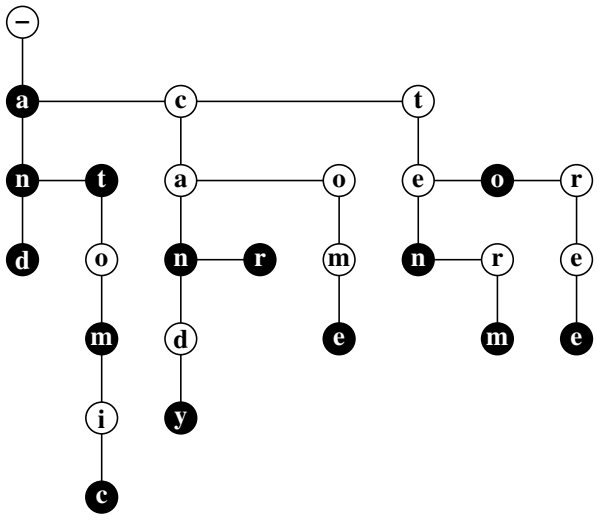
Write the entire project in plane C. Do not use any global or static variables.

Tries

A trie (derived from *retrieval*, also called a prefix tree) facilitates a compact storage of strings in the main memory. Part (a) of the following figure shows an example. Each dark node indicates the end of a string stored in the trie. The root node labelled as – is a dummy node. The labels read from the root node to a marked node (except the – stored at the root) define a string stored in the dictionary.



(a) A trie storing the words a, an, and, at, atom, atomic, can, candy, car, come, ten, term, to, tree



(b) The first–child–next–sibling representation of the trie

The first-child-next-sibling representation of the trie is shown in Part (b) of the figure. The vertical links are the first-child links, whereas the horizontal links are the next-sibling links. You should go for this representation of a trie. Do **not** use an array of 26 pointers at every node, because that is too wasteful of space, particularly when you work with real-life dictionaries containing hundreds of thousands of words.

For the trie library, define two data types `trienode` and `trie`. Each node consists of a character label, an end-of-word marker, and two pointers (first-child and next-sibling). Parent pointers are not needed for these applications. Each list of children linked by the next-sibling pointers must be kept (alphabetically) sorted without any duplicate labels. Implement the following functions for tries.

- `trieinit()` creates an empty trie containing only the root node.
- `triesearch(T,S)` finds whether the string `S` is stored in the trie `T`. The function returns 1 if the search succeeds, or 0 otherwise. Notice that `S` is stored in `T` if and only if there is a path from the root to a node labelled by `S` (except the – at the root) and the last node on this path is marked.
- `trieinsert(T,S)` inserts a string `S` in a trie `T`. This has no effect if `S` is already stored in `T`.
- `listall(T)` prints an alphabetic listing of all the words stored in the trie `T` (one word in one line).

Write a `makefile` to build the dynamic library `libtrie.so`. All includes in this assignment must be made in the `#include <...>` format, so you need to set `CFLAGS` in all the makefiles appropriately.

The word utilities

These utilities work on dictionaries (word lists) implemented as tries. Pretend (as in real-life situations) that you do not have access to the source code of the trie library. (Do you have or need the source code of the math library?) You have access only to the header file `trie.h` and the precompiled library `libtrie.so`. Build the utilities library `libwordutils.so` based upon this assumption. In particular, you must work only with the functions provided by the trie library. Make no attempt to manipulate a trie by any other means.

As a first level of abstraction, redefine the `trie` data structure as `dict` in the header file `dicttype.h`.

A `dict` should be populated by a list of words before anything is done on the word list. This is to be done by the module `dictload`. A file `words.txt` will be supplied to you. Keep it in the `dict` directory, and call it `DFLT_DICT` (the default dictionary). Write a function `loaddfltdict()` to read this file line by line, and prepare a `dict` from the strings read. Write another function `loaddict(fname)` where the `dict` is to be built from the explicitly specified dictionary file `fname`. This will allow the users to work with other dictionaries.

Then, write the utilities module **dictutils**. This consists of the following functions.

- **addbefore(D,S)** takes a **dict D** and a string **S** as parameters (**S** need not store a valid English word). The function should return a single string consisting of all single letters that can be added before **S** to obtain valid English words. For example, the string lab can be pre-augmented only as blab, flab, and slab, so the function would return the string **"bfs"**.
- **addafter(D,S)** takes a **dict D** and a string **S** as parameters (**S** need not store a valid English word). The function should return a single string consisting of all single letters that can be added after **S** to obtain valid English words. For example, the string pla can be post-augmented only as plan, plap, plat, and play, so the function would return the string **"npty"**.
- **anagrams(D,S)** takes a **dict D** and a string **S** as parameters (**S** need not store a valid English word). The function returns a dynamically allocated array of all permutations of **S**, that are valid English words according to the dictionary **D**. The array of strings should be null-terminated (similar to **argv**). For example, if you supply **opst** as **S**, then all the valid anagrams are opts, post, pots, spot, stop, and tops, so the array of strings to be returned by the function should be

  { "opts", "post", "pots", "spot", "stop", "tops", NULL }.

  Each valid anagram must appear only once in the output array. Be careful, because **S** may contain repeated letters. You are not required to output the strings in sorted order (no harm if you do so).

For the convenience of the users, write a single header file **wordutils.h** that serves the sole purpose of including all the *useful* header files of the wordutils library. Write a **makefile** to build the dynamic library **libwordutils.so**. Do not define any make dependencies on the trie library sources.

The application programs

Write two application programs in the **apps/** subdirectory.

maxchain.c finds the longest chain of valid words, that can be obtained by adding single letters to the beginning or end of words. Use the library calls **addbefore** and **addafter** to stage an exhaustive search. Print the longest chain in the following format (this is not the longest chain anyway).

```
    a
==> at
==> ate
==> rate
==> irate
==> pirate
==> pirates
```

subwords.c reads a short string S from the user. Let the length of S be $l \geqslant 2$. All $(l-1)$-letter permutations of S stored in the dictionary are printed (each only once). Use the library call **anagrams** to do this. The listing need not to be alphabetically sorted. Here is a sample transcript from a run of the program.

```
Enter a lower-case string: osptp
opts
post
pots
spot
stop
tops
pops
7 subwords found
```

Write a **makefile** in the root directory **A2/** of the project. **make libraries** should build the two dynamic libraries by recursing into the subdirectories **trie/** and **util/**. **make maxchain** should compile and run the application program **apps/maxchain.c**, and **make subwords** should compile and run the application program **apps/subwords.c**. Use another **makefile** in the **apps** directory.

All the makefiles should have appropriate cleaning targets to remove the object, library, and executable files. When you are done, **make** the cleaning, and submit the entire project as a compressed archive **A2.zip** or **A2.tgz** consisting only of the files mentioned on the first page and organized exactly as instructed there.