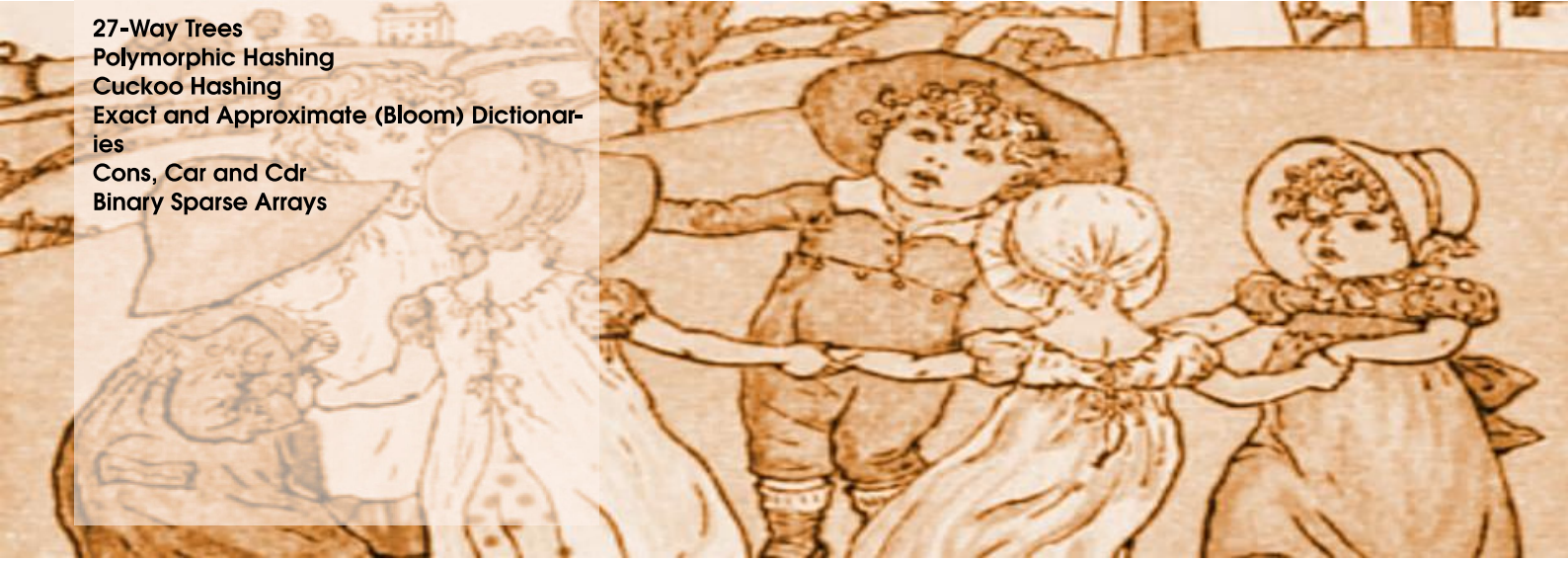
27-Way Trees
Polymorphic Hashing
Cuckoo Hashing
Exact and Approximate (Bloom) Dictionaries
Cons, Car and Cdr
Binary Sparse Arrays



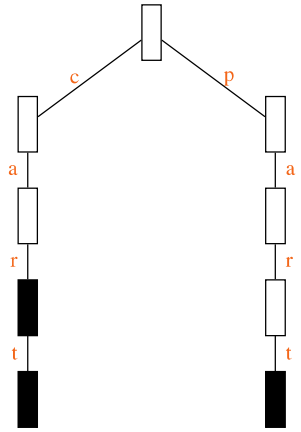
11. ADTs II & Data Structures

11.1 27-Way Trees

The Dictionary Abstract Data Type (ADT) allows words to be stored in such that they may be efficiently checked later, to ensure that words have been spelled correctly etc. There are many ways of implementing the Dictionary ADT (binary trees, hash tables and so on), but here we look at a tree structure which has a dynamic collection of nodes.
The 27-Way Tree has 27 downward links, each corresponding to one of the letters $a \ldots z$ and one to the apostrophe character '. This has many similarities with *tries*:

www `https://en.wikipedia.org/wiki/Trie`

Any word can be stored in the tree. You start at the top node, and, given the first character follow the relevant downward pointer. So, if the first character is 'a' you follow the first pointer down from the first node to another. If the next letter is 's' then you follow the 19^{th} downwards pointer from that node to the following one. Each pointer corresponds to a letter (or the apostrophe), and each level of the tree corresponds to the length of a word.



In the above diagram, the words 'car', 'cart' and 'part' have been stored in such a 27-Way Tree. Since 'car' is a substring of 'cart' we have to find a way of making it clear where valid word

ending are. In the diagram, black-filled nodes show such *terminal* nodes. The string 'ca' is not a valid word since the block immediately afterwards has not been marked as such. The word 'par' is not valid either, since it has never been added to the dictionary (but could, in principle, be added later).

Exercise 11.1.1 Write an ADT to implement the `t27.h` interface given. Write the source file `t27.c` such that the driver file `driver.c` works correctly. Ensure that this all works with the `Makefile` which, as with `t27.h` **must be** used **unaltered**. The standard operations are :

```
dict* dict_init(void);
bool dict_addword(dict* p, const char* str);
dict* dict_spell(const dict* p, const char* str);
int dict_nodecount(const dict* p);
int dict_wordcount(const dict* p);
int dict_mostcommon(const dict* p);
void dict_free(dict** p);
```

The function `dict_addword()` allows a string to be inserted into the tree, and the node after marked as a terminal node.
`dict_spell()` returns a pointer to the terminal node corresponding to the string (if it has been inserted) and `NULL` otherwise.
`dict_nodecount()` returns the total number of nodes which are part of the tree, and `dict_wordcount()` returns the total number of words which have been inserted into the tree **including** duplicates. A frequency counter is used in each node to keep a track of words which have been added multiple times. In this case this counter is incremented, but no new nodes are created.
`dict_mostcommon()` returns the number of times the most common word has been inserted.
The above functions are the standard assignment and are worth 60%. There are two additional advanced functions for you to implement, each worth up to 10% of the assignemnt:

```
unsigned dict_cmp(dict* p1, dict* p2);
```

which counts the least number of nodes you must traverse to move from one node to another in the tree. In the above figure, there are 7 steps from 'car' to 'part'. The other is:

```
void dict_autocomplete(const dict* p, const char* wd, char* ret);
```

which returns the substring corresponding to the additional letters required to get from the `wd` to the most frequent word below it. In the figure, to autocomplete 'car' would be the word 'cart', so the additional letters required are simply 't', which is stored in `ret`.

- We've provided the structure that **must** be used for this assignment. Do not attempt to change it. All functionality required is possible using this structure.
- For questions such as "but what about hyphenated words?" etc., look in the `driver.c` file. If we haven't tested for it in there, make a sensible decision as to what should happen. Provided that this file still operates correctly you can decide some of the edge-cases for yourself.
- Even if you don't get all the functions to work correctly, make sure the code still compiles by writing 'dummy' functions as placeholders, even if some of the assertions fail. I'll test the basic functionality separately from the two advanced ones.

- This assignment is brand new. If minor typos etc. are uncovered as you complete the assignment we may update this documentation and/or the online files provided. Please check regularly to make sure you are using the most recent version.

This is worth 80% **of the marks** (60% + 10% + 10%)

The manner in which you've been asked to implement the functionality above is **very** specific. However, exactly the same functionality (adding/spell-checking etc.) could be implemented in very different ways (linked lists, BSTs, hashing etc.). As an extension, write a 'rival' version to implement the standard functionality of these functions without using a 27-Way Tree. This should still ensure that some of the standard functionality in `driver.c` can be compiled against it. This means you can ignore `dict_autocomplete()`, `dict_cmp()` and `dict_nodecount()` since these analyse something very specific about the 'shape' of the 27-Way Tree data structure. Use your own Makefile to achieve this, and put them in the sub-directory `Extension`.

If you do this, also submit `extension.txt` which details what you have done, the motivation for it, and how well it works in practice (or not), and comparing it with the original tree. This discussion (a few hundred words at most) is as (if not more) important than the code itself. You may wish to discuss which of the two methods is better, in terms of actual speed, complexity, memory usage, testability etc.
This worth 20% **of the marks.**

■

11.2 Polymorphic Hashing

Polymorphism is the concept of writing functions (or ADTs), without needing to specify which particular type is being used/stored. To understand the quicksort algorithm, for instance, doesn't really require you to know whether you're using integers, doubles or some other type. C is not very good at dealing with polymorphism - you'd need something like Java or C++ for that. However, it does allow the use of `void*` pointers for us to approximate it.

Exercise 11.2.1 Here we write an implementation of a polymorphic associative array using hashing, see `assoc.h`. Both the key & data to be used by the hash function are of unknown types, so we will simply store void pointers to both of these, and the user of the associative array (and **not** the ADT itself) will be responsible for creating and maintaining such memory, and also ensuring it doesn't change when in use by the associative array. In the `testassoc.c` file we show two uses for such a type : a simple string example (to find the longest word in English that is also a valid (but different) word when spelled backwards), and a simple integer example where we keep a record of how many unique random numbers in a range are chosen.

Since we do not know **what** the type of the key is, we need to be careful when comparing or copying keys. Therefore, in the `assoc_init(int keysize)` function, the user has to pass the size of the key used (e.g. `sizeof(int)`) or in the case or strings, the special value 0. Now we can use `memcmp()` and `memcpy()`, or in the case of strings, `strcmp()` and `strcpy()` for dealing with the keys. In the case of data, the ADT only ever needs to return a pointer to the data (not process it) via `assoc_lookup()`, so its size is not important.

Your hash table used to implement the ADT should be resizeable, and you may use open-addressing/double-hashing or separate chaining to deal with collisions. Make no assumptions about the maximum size of the array, and make the initial size of the array small e.g. 16 or 17 (a prime is useful if you're double hashing).. You can use any hash function you wish, but if