Data Structures in Python 3

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2019/index.html

Tries – C-like implementation I

- As in C
  - trie is an array of trie nodes
  - each node is an array
    - indexed by symbols
    - array values correspond to indices of child nodes

	a	b	. . .	y	z	flag
0						
1						
⋮			⋮			

Tries – C-like implementation II

```
if __name__ == '__main__': # can use code in this file via import (?)
    if len(sys.argv) != 2 : # == argc
        # Note the printf-like syntax below
        sys.exit('Usage: %s <dict file name>' % sys.argv[0])

    # CAUTION:
    # trie = [ [ 0 ] * (NUM_SYMBOLS+1) ] * INIT_SIZE WILL NOT WORK
    # The above creates copies of the SAME list, so when you modify
    # one of them they all change.
    # See
    https://stackoverflow.com/questions/6667201/how-to-define-a-two-dimensi
    trie = [ [ 0 ] * (NUM_SYMBOLS+1) for i in range(INIT_SIZE) ]

    with open(sys.argv[1]) as f : # with => don't need a f.close()
        for line in f: # ASSUMPTION: one word per line
            insert_string(trie, line.strip())
```

Tries – C-like implementation III

```
def insert_string(trie, s) :
    index = 0
    for c in s.lower() :
        i = ord(c) - ord('a')    # get ASCII value
        if i < 0 or i >= NUM_SYMBOLS :
            # print("Unexpected character %c in string %s" % (c, s))
            continue
        if trie[index][i] == 0 : # need new node
            new_index = get_new_node(trie)
            trie[index][i] = new_index
            index = new_index
        else :
            index = trie[index][i]
    trie[index][NUM_SYMBOLS] += 1
```

Tries – C-like implementation IV

```
def get_new_node(trie) :
    global num_nodes, max_nodes
    if num_nodes == max_nodes :
        trie.extend([ [ 0 ] * (NUM_SYMBOLS+1) for i in range(INIT_SIZE)
    ])
        max_nodes += INIT_SIZE
    num_nodes += 1
    return num_nodes - 1
```

Tries++ I

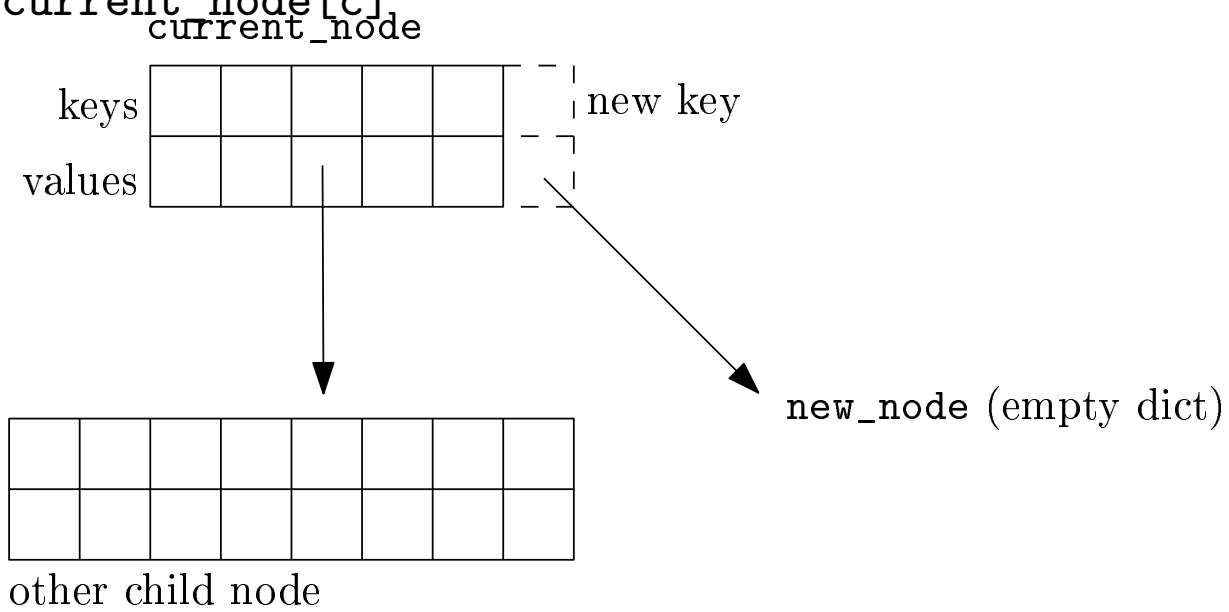
- Each trie node is a dictionary (list of key, value pairs)
- Keys in each dictionary: symbols in your alphabet
- Value for any key: reference (pointer) to child node (another dictionary)

Initialisation

```
trie = { } # Just the first node
```

Tries++ – Insertion I

```
def insert_string(trie, s) :
    current_node = trie
    for c in s.lower() :
        if c not in current_node : # c is not a key in this dictionary
            new_node = {}
            current_node[c] = new_node
            current_node = new_node
        else :
            current_node = current_node[c]
```



Tries++ – Searching I

```
def search(trie, s) :
    current_node = trie
    for c in s.lower() :
        if c not in current_node :
            return False
        else :
            current_node = current_node[c]
```

```
    while True:
        s = input("Enter word to search for: ")
        if s == "quit" : # Why can you not use "is" ?
            break
        if search(unpickled_trie, s) :
            print("Word found")
        else :
            print("Word not found")
```

Additional modules

pickle https://docs.python.org/3/library/pickle.html

collections.namedtuple https://docs.python.org/3/library/collections.html#collections.namedtuple

Problems I

1. Rewrite in Python the implementation of heaps provided to you. Check your implementation by using it to write a program that heapsorts a list of integers provided as command-line argumets
2. Implement binary trees using `namedtuple`. Test your implementation by writing preorder / inorder / postorder traversal routines.
3. Informally compare the times taken by the C and Python implementations to process `/usr/share/dict/words` (and write `dict.h`, if applicable).