# Search Engine Lite

Created By: Ilija Brdar & David Drvar

# Adding to Trie

```
{make cached words dict CW}
{make fileInfo FI}
for all w in words do
    if w in CW then
        CW[w].occurence += 1
    else
        CW[w] = FI
        Trie.add(w, FI)
```

```
add_word(word, FI)
    curr := root
    for all char in word do
        element := Node(char)
        for all child in curr.children
            if child.char is char then
                curr := child
                found := True
                break
        if not found then
            {add node to curr's children}
            curr := element
curr.isEnd = True
curr.addFile(FI)
```

# Searching in Trie

## Complexity

**Adding:** $O(AN)$
**Searching:** $O(AN)$

A – alphabet size + 9 digits
N – word length

```
search_trie(word)
    curr := root
    for all char in word
        for all node in curr.children
            if char is node.char then
                curr := node
                found = True
                break
        If not found then
            return              {didn't find}
    if curr.isEnd then
        return curr.files       {found}
    else
        return                  {didn't find}
```

# Graph structure

- Directed
- Dictionary of vertices – key is file address

# Vertex structure

- List of incoming vertices
- List of outgoing vertices
- ID - file's address

# Adding an edge to Graph

```
if origin not in self.vert_list then
        add_vertex(origin)
if destination not in self.vert_list then
        add_vertex(destination)

vert_list[origin].add_outgoing(vert_list[destination])
vert_list[destination].add_incoming(vert_list[origin])
```

# Set - structure and methods

Implemented using Python dictionary

- key – file address
- value – number of word occurrences within the file

| 3 main methods : | Complexity |
|---|---|
| • union | O(n) |
| • intersection | O(n) |
| • difference | O(n) |

# 1. Union

```
for file in other_set do
        self.add(file, other_set[file])
```

# 2. Intersection

```
files_to_be_removed := []
for file in my_set do
        if file in other_set then
                add a file in the original set and sum up the number of word apperreances
        else
                files_to_be_removed.append(file)

for file in files_to_be_removed do
        remove those files from the original set that other_set doesn’t contain
```

# 3. Difference

```
to_be_removed := []

for file in self.my_set do
        if file in other_set then
                to_be_removed.append(file)

for file in to_be_removed do
        self.remove(file)
```

# Parsing Simple Queries

Defining simple queries:

- token OP token
- NOT_OP token

token: it can be any string of chars

OP = {AND, OR, NOT, and, or, not,
And, Or, Not, AnD, oR,…}

NOT_OP: subset of OP containing
'not' tokens

```
parseQuery(query)
    critera := query.split()
    if query[0] == 'not' then
        if len(criteria) != 2 then ERROR
        if criteria[1] in OP then ERROR
    if any token from OP in criteria then
        if len(criteria) != 3 then ERROR
        if criteria[0] or criteria[1] in OP then
            ERROR
        else
            flag := True
{if there are duplicates, get rid of them}
```

# Query executing

```
fill_sets(set1, set2, crit)
    FIs := search_trie(crit[0])
    {for not every file from the path}
    set1.add(FI) for all FI from FIs
    FIs := serach_trie(crit[2])
    set2.add(FI) for all FI from FIs
```

```
execute(T, crit, path)
    if crit[0] == 'not' then
        fill_sets(set1, set2, crit)
        ret set1.difference(set2)
    else if any token from OP in crit then
        fill_sets(set1, set2, crit)
        ret set1.union(set2) if crit[1] == 'and'
        ret set1.difference(ser2) if crit[1] == 'not'
        ret set1.intersection(set) otherwise
    else
        for all word in crit
            fill_set(set, crit)
            result = result.union(set)
        ret result
```

# Computing a pagerank

$$rank = \sum words\ in\ a\ vertex + 0.7 \times \sum incoming\ edges + 0.9 \times \sum words\ in\ verteces\ that\ point\ to\ the\ vertex$$

value of an edge from a vertex that contains at least one searched word is 3, instead of 1

a higher rank of pages that contain all searched word is implemented by summation of word occurrences in *union* and *intersection* in *set*

# Heapsort

- max heap – value in a parent node is greater than the values in its two children nodes
- array based implementation
- (key, value) where key is file's rank and value file address
- adding elements from result_set to the heap and removing the max element one by one and forming the sorted result_set
- Time complexity O (nlogn)

## Adding to heap

*append a new node to the end of array*

upheap(len(data) – 1)

## Upheap

parent_index = (index – 1) // 2

**if** data[index] < data[parent_index] **then**

    return

swap(index, parent_index)

upheap(parent_index)

# Removing max from heap

*swap root and the last element*

*pop the last element from heap*

downheap(0)

# Downheap

```
left_child := index * 2 + 2
right_child := index *2 + 1
max_index := index
if data[left_child] > data[index] then
        max_index := left_child
if data[right_child] > data[index] then
        max_index := right_child

if max_index != index then
        swap(max_index, index)
        downheap(max_index)
```

# Pagination

```
input page length
index := 0
print(result_set[index:page_length])
index := page_length
while True do
        char := input('A for the previous page, D for the next, X for changing the page length and Q for exit')
        if char == 'A' then
                index := index – 2 * page_length
                print(result_set[index : index + page_length])
        else if char == 'D' then
                print(result_set[index : index + page_length])
                index := index + page_length
        else if char == 'X' then
                input new page length and print result_set from the beginning
        else if char == 'Q' then
                break
```

# Parsing Advanced Queries

- C-like logical expressions
- *Parglare* library
- Flex & Bison Grammar (saves priority)
- IR Classes (or, and, not)
- Action dictionary (using lambda functions)

**class** orNode

    left_child := None

    right_child := None

**class** notNode

    child := None

# Parsing Advanced Queries

## Grammar part

```
OrExpression
   : AndExpression
   | OrExpression '||'
     AndExpression
   ;
```

## Implementation part

```
"OrExpression" : [
   lambda _, n: n[0],
   lambda _, n: orNode(n[0], n[1])
]
```

# Advanced queries - evaluating an IR tree

```
if node is leaf then
    return node
else
    left := evaluate_tree(node.left)
    right := evaluate_tree(node.right)
    if node is NotNode then
        if right is leaf then
            form criteria and execute basic query
        else if right is set then
            {make set including all files}
            result_set := set.difference(right)
    else if node is AndNode then
        if left and right are leaves then
            form criteria and execute basic query
        else if left and right are sets then
            result_set := left.intersection(right)
        else if one is a set and the other is leaf then
            form criteria and execute basic query for a leaf
            result_set := left.intersection(right)
    else if node is OrNode then
        if left and right are leaves then
            form criteria and execute basic query
        else if left and right are sets then
            result_set := left.union(right)
        else if one is set and other is leaf then
            form criteria and execute basic query for a leaf
            result_set := left.union(right)
```

# THANK YOU FOR YOUR ATTENTION!