Course Code : MCS-021
Course Title : Data and File Structures
Assignment Number : MCA(2)/021/Assign/13
Maximum Marks : 100
Weightage : 25%

This assignment has four questions which carry 80 marks. Answer all the questions. Each question carries 20 marks. You may use illustrations and diagrams to enhance the explanations. Please go through the guidelines regarding assignments given in the Programme Guide. Ensure that you don't copy the program from course material or any other source. All the implementations should be in C language.

Question 1:
Write an algorithm for the implementation of Doubly Linked Lists.

Solution :

Alogrithm :

Step 1: Read the list operation.

Step 2: If operation is Create then process the following steps.

1.
Create the new node and allocate memory for the new node.

2.
Read the data in the new node.

3.
Assign Flink and Blink as NULL.

4.
Assign current as new.

Step 3: If operation is Insertion do the following steps.

i)
Check insertion at beginning is true then

1.
Create the new node and allocate memory for the new node

2.
Read the data in the new node.

3.
Move the current node to start position

4.
Assign new->Blink =Null

5.
Assign new->Flink=current

6.
Now assign current->Blink=new

ii)
Insertion at between nodes is true then

1.
Create the new node and allocate memory for the new node

2.
Read the data in the new node.

3.
Move the current node required position

4.
Assign new node’s Blink=current.

5.
Assign new node’s Flink=current->Flink

6.

Assign current node’s Flink =new

iii)
Insertion at between nodes is true then

1.
Create the new node and allocate memory for the new node.

2.
Read the data in the new node.

3.
Move the current node to end position.

4.
Assign current node’s Flink =new.

5.
Assign new’s Blink as current.

6.
Assign new’s Flink as NULL.

7.
Move end pointer to new node.

Step 4: If operation is deletion then do the following steps. i). Check deletion at beginning is true then.
1.
Move current pointer to start position.

2.
Move the start pointer next node in the link.

3.
Assign start’s Blink as NULL.

4.
Reallocate current node from memory.

ii)
Check deletion between two nodes is true

1.
Move the current pointer to required position.

2.
Assign current’s->Blink->Flink as current’s Flink.

3.
Assign currents’s->Flink->Blink as current’s Blink.

4.
Reallocate current from memory.

iii)
Check deletion at end is true

1.
Move the current pointer to end position.

2.
Assign current’s->Blink->Flink as Null.

3.
Reallocate current from memory.

4.
Reallocate current from memory. Step 5: If the operation is traversing. i)
Check deletion at end is true

1.
Move the current pointer to end position.

2.
Repeat the following steps until current becomes NULL.

3.
Display the data.

4. Current=current-
>Flink.

ii)
If Backward traversing then

1.
Move the current to end position.

2.
Repeat the following steps until current becomes NULL.

3.
Display the data.

4. Current=current-
>Flink.

QUEUE-EMPTY
if L.head == NIL return
True else return False

QUEUE(x):
if L.head == NIL: x.prev = NIL L.head =
x else
cur = L.head
while cur.next != NIL cur = cur.next
cur.next = x x.prev = cur x.next = NIL

6.7 Multiple Queues Using Single Array

As we have discussed that one of the application of queues is categorization of data is possible. And multiple queues can be used to store variety of data. We can implement multiple queues using single dimensional arrays.

In a one dimensional array, multiple queues can be placed. Insertion from its rear end and deletion from its front end can be possible for desired queue. You can visualize this idea with the help of Fig. 6.10 given below.

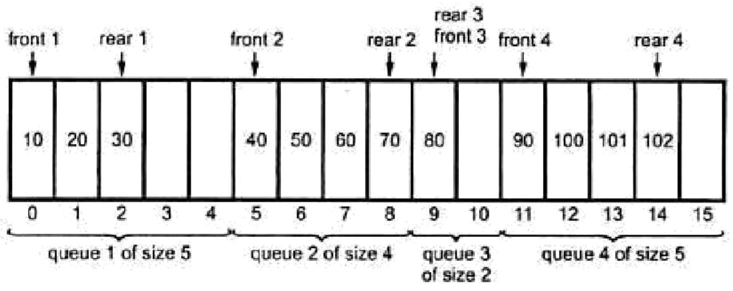

Fig. 6.10 Multiple queues using single array

As shown in the Fig. 6.10, there are four queues having their own front and rear positioned at appropriate points in a single dimensional array. We can perform insertion and deletion of any element for any queue. We can declare the messages "queue full" and "queue empty" at appropriate situations for that particular queue.

DEQUEUE():

x = L.head L.head = x.next
x.next.prev = L.head return x

Question 2:

Implement multiple queues in a single dimensional array. Write algorithms for various queue operations for them .

Solution :

Question 3:

Write a note of not more than 5 pages summarizing the latest research in the area of "Searching Algorithms". Refer to various journals and other online resources. Indicate them in your assignment.

Solution :

.
.
.

The Fish-Search Algorithm

A typical example of dynamic search is WebGlimpse [Manber et al. 97] that allows users to dynamically search sub-areas of the Web predefined by a given "hop" parameter (i.e. depth in the Web viewed as a graph). The fish-search [De Bra et al. 94] proposes a more refined paradigm that can be explained by the following metaphor. It compares search agents exploring the Web to a school of fish in the sea. When food (relevant information) is found, fish (search agents) reproduce and continue looking for food, in the absence of food (no relevant information) or when the water is polluted (poor bandwidth), they die. The key principle of the algorithm is the following: it takes as input a seed URL and a search query, and dynamically builds a priority list (initialized to the seed URL) of the next URLs (hereafter called nodes) to be explored. At each step the first node is popped from the list and processed. As each document's text becomes available, it is analyzed by a scoring component evaluating whether it is relevant or irrelevant to the search query (1-0value) and, based on that score, a heuristic decides whether to pursue the exploration in that direction or not: Whenever a document source is fetched, it is scanned for links. The nodes pointed to by these links (denoted "children") are each assigned a depth value. If the parent is relevant, the depth of the children is set to some predefined value. Otherwise, the depth of the children is set to be one less than the depth of the parent. When the depth reaches zero, the direction is dropped and none of its children is inserted into the list.

Children whose depth is greater than 0 are inserted in the priority list according to the following heuristics:

1.
the

are added to the head of the
list 2.

the first width children of an irrelevant node are added to the list right after the last child of a relevant node
3.

the rest of the children are added to the tail (to be handled only if time permits) 4. We propose several improvements to the original fish-search algorithm in order to overcome these limitations. They result in a new algorithm, called the "shark search" algorithm, that while using the same simple metaphor, leads to the discovery of more relevant information in the same exploration time.
5.

One immediate improvement is to use, instead of the binary (relevant/irrelevant) evaluation of document relevance, what we will call hereafter a similarity engine in order to evaluate the relevance of documents to a given query. Such an engine analyzes two documents dynamically and returns a "fuzzy" score, i.e., a score between 0 and 1 (0 for no similarity whatsoever, 1 for perfect "conceptual" match) rather than a binary value. A straightforward method for building such an engine is to apply the usual vector space model [Salton & McGill 83]. It has been shown that these kinds of techniques give better results than simple string- or regular-expression matching [Salton 89, p306]. The similarity algorithm can be seen as orthogonal to the fish-search algorithm. We will assume in the rest of the paper that such an engine is available and that for any pair query, document, (q,d), it returns a similarity score sim(q,d) between 0 and 1.

6.

This first improvement has a direct impact on the priority list. We use a "fuzzy" relevance score, giving the child an inherited score, thus preferring the children of a node that has a better score. This information can be propagated down the descendants chain, thus boosting the importance of the grandchildren of a relevant node over the grandchildren of an irrelevant node.

The children of an irrelevant node use their parent's inherited score for calculating their own inherited score -

"fading" factor [Marchiori 97]. Thus, suppose documents X and Y were analyzed and that X has higher score. Suppose further that the children of both X and Y have a null score, and the algorithm now has to select the most promising of their grandchildren. Since both their scores are multiplied g

further away from X, into the great-grandchildren zone, would bring Y's descendants back into inheritance is much more natural than the Boolean approach of the fish-search.

7.

A more significant improvement consists of refining the calculation of the potential score of the children not only by propagating ancestral relevance scores deeper down the hierarchy, but also by making use of the meta-information contained in the links to documents. We propose to use the hints that appear in the parent document, regarding a child. The anchor text of the link is the author's way to hint as to the subject of the linked document. A surfer on the Web, encountering a page with a large amount of links, will use the anchor text of the links in order to decide how to proceed. Some automated tools (see [Iwazume et al. 96]) also take advantage of that information.

This approach however can fail in poorly styled HTML documents, in which anchor texts consist only of "click here" or "jump there", or when the anchor information is a picture without ALT information.

To remedy this problem we suggest using the close textual context of the link, and combining the information extracted from it with the information extracted from the anchor text. To reduce the risk of mistakenly giving a high potential score to a node due to a textual context that actually belongs to another node linked in the same paragraph (context boundaries are difficult to identify), we suggest a small fix. If a node has a relevant anchor text, the score of the textual context is set to the maximum (value of 1), thus giving it an edge over neighboring links. We claim that a policy that selects the children with the most promising anchor and anchor context information is preferable to arbitrarily selecting the first set of children.

Open Mass Spectrometry Search Algorithm

Large numbers of MS/MS peptide spectra generated in proteomics experiments require efficient, sensitive and specific algorithms for peptide identification. In the Open Mass Spectrometry Search Algorithm (OMSSA), specificity is calculated by a classic probability score using an explicit model for matching experimental spectra to sequences. At default thresholds, OMSSA matches more spectra from a standard protein cocktail than a comparable algorithm. OMSSA is designed to be faster than published algorithms in searching large MS/MS datasets.

The Shark-Search Algorithm

We propose several improvements to the original fish-search algorithm in order to overcome these limitations. They result in a new algorithm, called the "shark search" algorithm, that while using the same simple metaphor, leads to the discovery of more relevant information in the same exploration time.

One immediate improvement is to use, instead of the binary (relevant/irrelevant) evaluation of document relevance, what we will call hereafter a similarity engine in order to evaluate the relevance of documents to a given query. Such an engine analyzes two documents dynamically and returns a "fuzzy" score, i.e., a score between 0 and 1 (0 for no similarity whatsoever, 1 for perfect

"conceptual" match) rather than a binary value. A straightforward method for building such an engine is to apply the usual vector space model [Salton & McGill 83]. It has been shown that these kinds of techniques give better results than simple string- or regular- expression matching [Salton 89, p306]. The similarity algorithm can be seen as orthogonal to the fish-search algorithm. We will assume in the rest of the paper that such an engine is available and that for any pair query, document, (q,d), it returns a similarity score

sim(q,d) between 0 and 1.

This first improvement has a direct impact on the priority list. We use a "fuzzy" relevance score, giving the child an inherited score, thus preferring the children of a node that has a better score. This information can be propagated down the descendants chain, thus boosting the importance of the grandchildren of a relevant node over the grandchildren of an irrelevant node. The children of an irrelevant node use their parent's inherited score for calculating their own inherited score - by ing" factor

[Marchiori 97]. Thus, suppose documents X and Y were analyzed and that X has higher score.

Suppose further that the children of both X and Y have a null score, and the algorithm now has to select the most promising of their grandchildren. Since

shark search chooses X's grandchildren first (which seems intuitively correct). Stepping further away from X, into the great-grandchildren zone, would bring Y's descendants back into consideration, given an app

more natural than the Boolean approach of the fish-search.

A more significant improvement consists of refining the calculation of the potential score of the children not only by propagating ancestral relevance scores deeper down the hierarchy, but also by making use of the meta-information contained in the links to documents. We propose to use the hints that appear in the parent document, regarding a child. The anchor text of the link is the author's way to hint as to the subject of the linked document. A surfer on the Web, encountering a page with a large amount of links, will use the anchor text of the links in order to decide how to proceed. Some automated tools (see [Iwazume et al. 96]) also take advantage of that information.

This approach however can fail in poorly styled HTML documents, in which anchor texts consist only of "click here" or "jump there", or when the anchor information is a picture without ALT information.

To remedy this problem we suggest using the close textual context of the link, and combining the information extracted from it with the information extracted from the anchor text. To reduce the risk of mistakenly giving a high potential score to a node due to a textual context that actually belongs to another node linked in the same paragraph (context boundaries are difficult to identify), we suggest a small fix. If a node has a relevant anchor text, the score of the textual context is set to the maximum (value of 1), thus giving it an edge over neighboring links. We claim that a policy that selects the children with the most promising anchor and anchor context information is preferable to arbitrarily selecting the first set of children.

Question 4:
What are the applications of Tries?

Solution :

APPLICATIONS OF TRIES

Tries find numerous applications. Some of them are explained as follow s:

1 **- Tries in the Web search engines:** Tries can be used to store and maintain the index words and the relevant Web site address. In the Web search process the users query the index words, the search engine has to fetch the appropriate links as results. In this process the search engines maintain the table of index words to the Web link or Website address. To maintain a sequence of words as the index , the search engines follow the tries procedure for quick retrieval of appropriate information.

2 - **Symbol table maintenance :** A symbol table is a data structur e used in the language translators such as compilers or interpreters. A symbol table consists of the address location, syntax and semantic information of the identifier , type and scope of the ident ifier . To maintain all the identifiers' information, the compilers or interpreters use tries procedure in the form of hash tables.

As a replacement for other data structures

As mentioned, a trie has a number of advantages over binary search trees.[4] A trie can also be used to replace a hash table, over which it has the following advantages:

Looking up data in a trie is faster in the worst case, O(m) time (where m is the length of a search string), compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash.

There are no collisions of different keys in a trie.

Buckets in a trie which are analogous to hash table buckets that store key collisions are necessary only if a single key is associated with more than one value.

There is no need to provide a hash function or to change hash functions as more keys are added to a trie. A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory.[5]

Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.

Some tries can require more space than a hash table, as memory may be allocated for each character in the search string, rather than a single chunk of memory for the whole entry, as in most hash tables.

Dictionary representation

A common application of a trie is storing a predictive text or autocomplete dictionary, such as found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e. storage of information auxiliary to each word is not required), a minimal deterministic acyclic finite state automaton would use less space than a trie. This is because an acyclic deterministic finite automaton can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored.

Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking and hyphenation[2] software.

Algorithms[edit source | editbeta]

We can describe lookup (and membership) easily. Given a recursive trie type, storing an optional value at each node, and a list of children tries, indexed by the next character, (here, represented as a Haskell data type):

```
data  Trie  a  =
Trie  { value ::
Maybe a
, children :: [(Char,Trie a)] }
```

We can look up a value in the trie as follows:

```
find :: String -> Trie a -> Maybe a find [] t = value t
find (k:ks) t = case lookup k (children t) of


Nothing -> Nothing
Just ct -> find ks ct
```

In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm

in Python (here, specifically for testing membership). Note that children is map of a node's children; and we say that a "terminal" node is one which contains a valid word.

```
def find(node, key): for char in key:
if char not in node.children: return
None else:
node = node.children[char] return node.value
```

A C version

```
#include<stdio.h>
#include<malloc.h>

typedef struct trie
{




}trie;

int words; int prefixes;
struct trie *edges[26];


trie * initialize(trie *node)
{
int i; if(node==NULL)
node=(trie *)malloc(sizeof(trie)); node-
>words=0; node->prefixes=0;

for(i=0;i<26;i++)

node->edges[i]=NULL; return node;
}

trie * addWord(trie *ver,char *str)
{
```

```
printf("%c",str[0]);
if(str[0]=='')
{






}
else
{

ver->words=ver->words+1;


ver->prefixes=(ver->prefixes)+1; char
k; k=str[0]; str++;
int index=k-'a'; if(ver-
>edges[index]==NULL)
{
ver->edges[index]=initialize(ver->edges[index]);
}
ver->edges[index]=addWord(ver->edges[index],str);
}
return ver;
}

int countWords(trie *ver,char *str)
{
if(str[0]=='') return
ver->words;

else
{




int k=str[0]-'a'; str++;



if(ver->edges[k]==NULL)
return 0;
return countWords(ver->edges[k],str);
}
}
```

```
int countPrefix(trie *ver,char *str)
{
if(str[0]=='')
return ver->prefixes;

else
{


}
}





int k=str[0]-'a'; str++;
if(ver->edges[k]==NULL)
return 0;
return countPrefix(ver->edges[k],str);

int main()
{
trie *start=NULL; start=initialize(start); int
ch=1; while(ch)
{

printf("1. Insert a word "); printf("2. Count words"); printf("3. Count prefixes"); printf("0.
Exit"); printf("your choice: "); scanf("%d",&ch); char input[1000]; switch(ch)

{

case 1:

printf("a word to insert: ");
scanf("%s",input);
start=addWord(start,input);
break;


case 2:


printf("a word to count words: ");
scanf("%s",input);
printf("%d",countWords(start,input));
break;


case 3:


printf("a word to count prefixes: ");
scanf("%s",input);
printf("%d",countPrefix(start,input));
break;

}
}
return 0 ;
}
```