Project 2

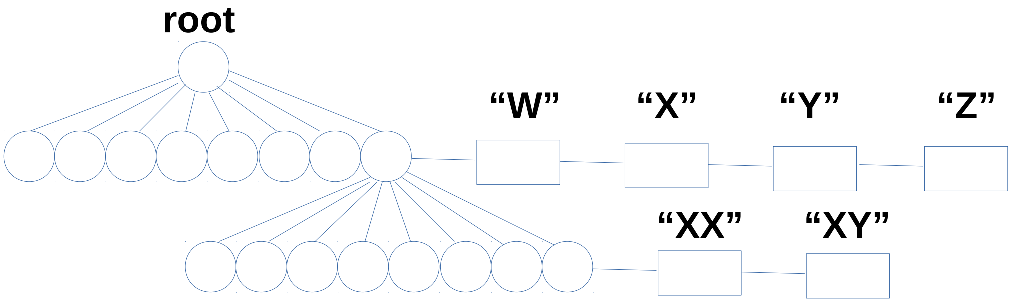
<T9 Trie>

CSC-17C
Rachel Scherer
Date: 10/31/18

Introduction

Inspired by the “Hashing” final exam question, “T9 Trie” is a data structure that’s inspired from the T9 predictive text technology used in older cell phones without on-screen keyboards.

This data structure is an *8-ary* tree that contains a linked list in each node. Each node is a number, 2-9 (represented internally as 0-7), and each link contains a word, an integer which represents the “priority” of the word (more frequently used words appear closer to the front of the linked list), and a pointer to the next link. This allows node 4 in 7→2→4 to contain a linked list Rag→ Sag (etc).



The core functionality of this program is to print word that best fits a given set of numbers, 2-9. Using the chart above, searching “99” should return “XX”, the highest priority (or most frequently used) word in that node.

Summary

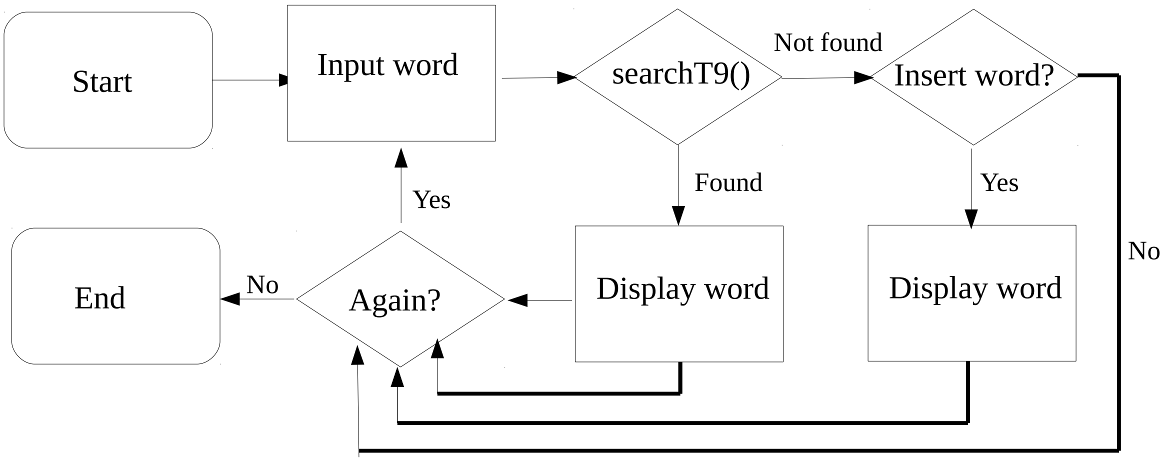
Project size: 817 lines of code (1087 Lines including comments and whitespace)

If a large enough dictionary of words were to be inserted into this data structure, I can see this working just as well as the traditional T9 predictive text used in older cell phones. Given more time, it would be nice to automatically have it set the most-recently queried words to the front of the linked list in its given node.

Description

This project was created as a means of getting more comfortable with trees and recursive functions.

Flow Chart (main.cpp)



Variables

Type	Variable Name	Description	Location
Trie*	NewTrie	Hold Trie in main()	main()
String	Input	User input	Main()
String	Word	Returned from searchT9()	main()
String	Hash	Returned from getHash()	main()
Ifstream	Fin	File i/o	insertDictionary()
system_error&	e	Error handling	InsertDictionary()
String	Word	Fin reads into this variable and gets inserted into trie	insertDictionary()
String	Word	Holds word	Trie::struct Link
Integer	Priority	Determines placement in Link*	Trie::struct Link
Link*	Next	Pointer to next link	Trie::struct Link
Node*	Child[8]	Pointers to children	Trie::struct Node
Link*	Head	Pointer to link in node	Trie::struct Node

Node*	Root	Root of trie	Trie
Link*	Temp	Iterates through link	destroySubTree()
queue<int>	Hash	Returned from getHashQueue to iterate through trie	insert()
Link*	NewLink	Inserted into node’s linked list	insert()
Link*	Temp	Iterates through link	insert()
Int	Val	Returned from hash.front()	insert()
queue<int>	Hash	Returned from getHashQueue to iterate through trie	remove()
Link*	temp	Iterates through link	remove()
Link*	prev	Iterates through link	remove()
Link*”	dltLnk	Iterates through link	remove()
Int	CurrentNum	Temporarily holds each number in “input” and pushes to “hashQueue”	searchT9()
queue<int>	HashQueue	Used to iterate through Trie	searchT9()
Int	Val	Returned from hash.front()	searchT9Rec()
queue<int>	Hash	Returned from getHashQueue to iterate through trie	searchWord()
Link*	Temp	Used to iterate through Link	searchWordRec()
Int	val	Returned from hash.front()	searchWordRec()
Node*	Temp	Holds current node in loop	breadth()
Link*	TempLink	Holds current link in loop	breadth()
queue<int>	HashQueue	Used to iterate through Trie	getHashQueue()
String	Hash	Returned from getHash()	getHashQueue()
Int	Val	Temporarily holds each number in “input” and pushes to “hashQueue”	getHashQueue
String	Hash	Holds converted key (2-9 or 0-7)	getHash()
queue<int>	Hash	Returned from getHashQueue()	setPriority()
Link*	Temp	Iterates through link	setPriortyRec()
Int	Val	Returned from hash.front()	setPriortyRec()
queue<int>	Hash	Returned from getHashQueue()	prioritize()
Link*	prev	Iterates through link	prioritizeRec()
Link*	moveFront	Moves link to front	prioritizeRec()
Int	Val	Returned from hash.front()	prioritizeRec()
Link*	prev	Iterates through link	deprioritizeRec()
Link*	moveBack	Moves link to back	deprioritizeRec()
Int	Val	Returned from hash.front()	deprioritizeRec()

Psuedocode - main.cpp

```
Int main
    New Trie newTrie
    call insertDictionary()
    string input
    Do
        Do
            print “Enter numbers you’d like to input”
            Enter input
        While input is not valid
        Call searchT9Trie(input), assign to word
        If word is blank
            print “Word not found”
            Do
                print “Would you like to insert one?”
                Enter input
            While input is not valid
            If input is yes
                Do
                    print “What would you like to insert?
                    Enter input
                While input is not valid
                call insert(int)
                call getHash(input) and print
        else
            print word
        Do
            print “Is there another number you’d like to be read?
            Enter input
        While input is not valid
    While input = yes
    Call getHash(“HelloWorld”), assign to hash
    Call insert(“HelloWorld”)
    Display hash
    Call searchT9(hash) and print word
    If searchWord(“HelloWorld”) is true
        print true
    else
        print false
    Call remove(“HelloWorld”)
    if searchWord(“HelloWorld”) is true
        print true
    else
        print false
    print preOrder()
    print postOrder()
    print breadth()
    delete newTrie
    return 0
}
```

```
insertDictionary(Trie)
    ifstream fin
    try
        open fin
    catch system error
        print error message
    string word
    if fin is open
        counter = 0
        While word reads from fin
```

```
            Increment counter
            Call insert(word, counter)
    call insert(“HelloDrLehr)
    Close fin
```

Psuedocode – Trie.cpp

Constructor

```
    node root;
    loop I from 0 to 8
        root’s 8 children are null
    root’s head is null
```

Destructor

```
    Call destroySubTree(root)
```

destroySubTrie(Node)

```
    If node does not exist
        return
    else
        loop I from 0 to 8
        call destroySubTrie for the 8 children
        call destroyWordLinks for node’s head
        delete node;
```

destoryWordLinks(Link)

```
    If head does not exist
        return
    else
        Link temp = head
        While temp exists
            temp = temp’s pointer
            delete head
            head = temp
```

insert(word,priority)

```
    queue of ints hash = getHashQueue(word)
    If hash is empty
        return
    else
        call insertRec(head,word,priority,queue)
```

insertRec(node,word,priority,queue)

```
    If node does not exist
        create new node
        Loop I from 0 to 8
            node’s 8 children are null
        Node’s head is null
    If hash is empty
        if there’s no head
            create new head
            set head’s priority to priority
            set head’s word to word
            set head’s next link to null
        else
            create new link
            if head’s priority is >= priority
```

```
                set newLink’s next pointer to head
                set head headLink
                set head’s word to word
                set head’s next link to null
            else
                create new link temp
                while temp’s next link exists and priority > temp’s next link’s priority
                    set temp to temp’s next link
                set newLink’s next link to temp’s next link
                set temp’s next link to newLink
                set newLink’s priority to priority
                set newLink’s word to word
    else
        set val to hash’s front int
        pop front from hash;
        call insertRec with node’s val
```

remove(word)

```
queue of ints hash = getHashQueue(word)
    If hash is empty
        return
    else
        call removeRec(head,word,priority,queue)
```

removeRec(node,word,priority,queue)

```
    If node does not exist
        return
    else
        If hash is empty
            if there’s no head
                return
            else if head’s word = word
                create new link temp
                set temp to head
                delete head
                set head to temp’s next link
            else if head’s next link doesn’t exist and head doesn’t equal word
                return
            else
                create new link prev
                set prev to head
                while prev’s next link exists and that word doesn’t equal word
                    set prev to prev’s next link
                if prev’s next word is word
                    create new link dltLnk
                    set dltLnk to prev’s next link
                    set prev’s next link to dltLnk’s next link
                    delete dltLnk
                    set dltLnk to null
                else
                    return
        else
            set val to hash’s front int
            pop front from hash;
            call insertRec with node’s val
```

searchT9(input)

```
    int currentNum
    queue of ints hashQueue
    for each letter in input
        set currentNum to that letter
        push currentNum to hashQueue
    return searchT9Rec(root,hashQueue)

searchT9Rec(node,hashQueue)
    if there’s no node
        return
    else
        if hashQueue is empty
            if there’s no head
                return
            return head’s word
        else
            set val to hashQueue’s front
            pop hashQueue’s front
            call searchT9Rec with node’s child

searchWord(input)
    call getHashQueue and set it equal to queue of ints, hash
    if the hash is empty
        return false
    else
        return searchWordRec(root,input,hash)

searchWordRec(node,word,hashQueue)
    if there’s no node
        return false
    else
        if hashQueue is empty
            if node’s head doesn’t exist
                return false
            else
                new Link temp
                set temp to node’s head
                while temp exists
                    if temp’s word = word
                        return true
                    set temp to temp’s next link
                return false
        else
            set val to hashQueue’s front
            pop hashQueue’s front
            call searchWordRec using val

preOrder
    call preorder(root)

preOrder(node)
    if node doesn’t exist
        return
    else
        if node’s head exists
            print node’s head’s word
        for I from 0 to 8
            call preorder for each of the nine children

postOrder
    call preorder(root)

postOrder(node)
    if node doesn’t exist
        return
    else
```

```
        for I from 0 to 8
            call preorder for each of the nine children
        if node’s head exists
            print node’s head’s word

breadth
    new Node temp
    new Link tempLink
    set newQueue to a queue of nodes
    push root to newQueue
    while newQueue’s size > 0
        set temp to newQueue’s front
    if temp’s head exists
        set tempLink to temp’s head
    while tempLink exists
        print tempLink’s word
        set tempLink to tempLink’s next link
    pop from newQueue
    for I from 0 to 8
        if temp’s child exists
        push that child to newQueue

getHashQueue(key)
    queue of ints hashQueue
    set string hash to getHash(key)
    int val
    for I from 0 to hash’s length
        set val to that character – 48
        push val to hashQueue
    return hashQueue
getHash(key,adjustFlag)
    set hash to “”
    for I from 0 to key’s length
        if that character is not A-Z
            return “”
        if adjustFlag is false
            if that character equals ‘a’,’b’, or ‘c’
                append 0 to hash
            if that character equals ‘d’,’e’, or ‘f’
                append 1 to hash
            if that character equals ‘g’,’h’, or ‘ci’
                append 2 to hash
            if that character equals ‘j’,’k’, or ‘l’
                append 3 to hash
            if that character equals ‘m’,’n’, or ‘o’
                append 4 to hash
            if that character equals ‘p’,’q’, ‘r’, or ‘s’
                append 5 to hash
            if that character equals ‘t’,’u’, or ‘v’
                append 6 to hash
            if that character equals ‘w’,’x’, ‘y’, or ‘z’
                append 7 to hash
        if adjustFlag is true
            if that character equals ‘a’,’b’, or ‘c’
                append 2 to hash
            if that character equals ‘d’,’e’, or ‘f’
                append 3 to hash
            if that character equals ‘g’,’h’, or ‘ci’
                append 4 to hash
            if that character equals ‘j’,’k’, or ‘l’
                append 5 to hash
            if that character equals ‘m’,’n’, or ‘o’
```

```
                append 6 to hash
            if that character equals ‘p’,’q’, ‘r’, or ‘s’
                append 7 to hash
            if that character equals ‘t’,’u’, or ‘v’
                append 8 to hash
            if that character equals ‘w’,’x’, ‘y’, or ‘z’
                append 9 to hash
    return hash
```

setPriority(string,priority)

```
    call getHashQueue and set is equal to queue of ints hash
    if hash is empty
        return
    else
        call setPriority(root,word,priority,hash)
```

setPriorityRec(node,word,priority,hash)

```
    if node doesn’t exist
        return
    else
        if hashQueue is empty
            if node’s head doesn’t exist
                return
            else
                new Link temp
                set temp to head
                while temp texists
                    if temp’s word = word
                        set temp’s priority to priority
                        return
                    set temp to temp’s next Link

        else
            set val to hashQueue’s front
            pop from hashQueue’s front
            call setPriority using node’s child
```

prioritize(word)

```
    call getHashQueue and set is equal to queue of ints hash
    if hash is empty
        return
    else
        call prioritizeRec(root,word,hash)
```

prioritizeRec(node,word,hash)

```
    if node doesn’t exist
        return
    else
        if hashQueue is empty
            if node’s head doesn’t exist
                return
            else if head’s word = word
                return
            else if head’s next link doesn’t exist and head’s word doesn’t equal word
                return
            else
                new link prev
                set prev to node’s head
                while prev’s next link exists and prev’s next word doesn’t equal word
                    set prev to prev’s next link
                if prev’s next word = word
                    new link moveFront
                    set moveFront to prev’s next link
                    set prev’s next link to moveFront’s next link
                    set moveFront’s next link to node’s head
                    set node’s head to moveFront
                    set node’s head’s priority to 0
                else
                    return
        else
            set val to hashQueue’s front
            pop from hashQueue’s front
            call prioritizeRec using node’s child

deprioritize(word)
    call getHashQueue and set is equal to queue of ints hash
    if hash is empty
        return
    else
        call deprioritizeRec(root,word,hash)

deprioritizeRec(node,word,hash)
    if node doesn’t exist
        return
    else
        if hashQueue is empty
            if node’s head doesn’t exist
                return
        else
            new link prev
                set prev to node’s head
                while prev’s next link exists and prev’s next word doesn’t equal word
                    set prev to prev’s next link
                if prev’s next word = word
                    new link moveBack
                    set moveBack to prev’s next link
                    set prev’s next link to moveBack’s next link
                    while moveBack exists
                        set MoveBack to moveBack’s next link
                    set moveBack’s next Link to null
                else
                    return
        else
            set val to hashQueue’s front
            pop from hashQueue’s front
            call deprioritizeRec using node’s child
```

Contents of main.cpp

```
 /*
 * File:   main.cpp
 * Author: rachel
 *
 * Created on December 13, 2018, 9:34 PM
 */

//System Libraries
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <sstream>
```

```
using namespace std;

//User Libraries
#include "Trie.h"

//Function prototypes
void insertDictionary(Trie*);
bool validateInputInt(string);
bool validateInputYN(string);
bool validateYN(string);
bool validateInputAlpha(string);
int countDigits(int);

int main(int argc, char** argv)
{
    Trie* newTrie = new Trie;

    //Inserting 1000 words in the Trie as a base
    insertDictionary(newTrie);

    //Allow the user to search and insert words in the trie.
    string input;
    do
    {
        do
        {
            cout << "\nPlease enter what numbers you'd like read using T9! (ex: 43556375347) ";
            cin >> input;
        }while(!validateInputInt(input));
        string word = newTrie->searchT9(input);
        if(word=="")
        {
            cout << "\nWord not found! ";
            do
            {
                cout << "Would you like to insert one? (y/n) ";
                cin >> input;
            }while(!validateInputYN(input));
            if(tolower(input[0])=='y')
            {
                do
                {
                    cout << "\nWhat is the word you'd like to insert? ";
                    cin >> input;
                }while(!validateInputAlpha(input));
                newTrie->insert(input, 1);
                cout << "\nThe numeric code for this word is: " << newTrie->getHash(input,true);
            }
        }
        else
        {
            cout << "\nYour word is: " << word << "\n";
        }
        do
        {
            cout << "\nIs there another number you'd like to be read? (y/n) ";
            cin >> input;
        }while(!validateInputYN(input));
    }while(validateYN(input));

    //Test "getHash"
    string hash = newTrie->getHash("HelloWorld",true);
    //Test "insert"
    cout <<"\nInserting 'HelloWorld' into the Trie...";
    newTrie->insert("HelloWorld",true);

    cout <<"\nHash for 'HelloWorld' = " << hash;

    //Test "searchT9"
    cout << "\nSearching by hash...";
    cout << "\nHash " << hash << " points to " << newTrie->searchT9(hash);

    //Test "searchWord"
    cout << "\nSearching by name... ";
    if(newTrie->searchWord("HelloWorld"))
    {
        cout << "Returns true.";
    }
    else
    {
        cout << "Returns false.";
    }

    //Test "remove"
    cout << "\nDeleting 'HelloWorld'... ";
    newTrie->remove("HelloWorld");

    //Test searchWord again after removing
    cout << "\nSearching again...";
    if(newTrie->searchWord("HelloWorld"))
    {
        cout << "Returns true.";
    }
    else
    {
        cout << "Returns false.";
    }

    //Test preorder output
    cout << "\n\n-------------------PREORDER-------------------\n";
    newTrie->preorder();

    //Test postorder output
    cout << "\n-------------------POSTORDER-------------------\n";
    newTrie->postorder();

    //Test breadth output
    cout << "\n-------------------BREADTH-------------------\n";
    newTrie->breadth();

    //Delete the trie
    delete newTrie;

    //Exit program
    return 0;
}

/**
 * Inserts "1000Words.txt" into the Trie
 * @param trie
 */
void insertDictionary(Trie* trie)
{
    ifstream fin;

    //Open the file
    try
    {
        fin.open("1000Words.txt");
    }
    catch(system_error& e)
    {
        cerr << e.code().message() << "\n";
    }

    //Read from the file, setting the line number as the priority of the word since it's sorted by
common use.
    string word;
    if(fin.is_open())
    {
        int counter = 0;
        while(fin >> word)
        {
            trie->insert(word,++counter);
        }
    }

    //Insert my own word in too
    trie->insert("HelloDrLehr",1);

    //Close file
    fin.close();
}

/**
 * Returns true if input only contains digits from 2-9
 * @param input
 * @return
 */
bool validateInputInt(string input)
{
    for(int i=0;i<input.length();i++)
    {
        if(!isdigit(input[i]))
        {
            cout << "Error! Invalid input.\n";
            return false;
        }
        if((int)input[i] < 50 || (int)input[i] > 57)
        {
            cout << "Error! Digits must be 2-9.\n";
            return false;
        }
    }
    return true;
}

/**
 * Returns true if input is either 'y' or 'n'
 * @param input
 * @return
 */
bool validateInputYN(string input)
{
    if(tolower(input[0]) != 'y' && tolower(input[0]) != 'n')
    {
        cout << "Error! Invalid input.\n";
        return false;
    }
    return true;
}

/**
 * Returns true if input is 'y'
 * @param input
 * @return
 */
bool validateYN(string input)
{
    if(tolower(input[0]) == 'y') return true;
    else return false;
}

/**
 * Returns true if input only contains letters
 * @param input
 * @return
 */
bool validateInputAlpha(string input)
{
    for(int i=0;i<input.length();i++)
    {
        if(!isalpha(input[i]))
        {
            cout << "Error! Must contain letters only.\n";
            return false;
        }
    }
    return true;
}
```

Contents of Trie.h

```
/*
 * File:   Trie.h
 * Author: rachel
 *
 * Created on December 13, 2018, 9:38 PM
 */

#ifndef TRIE_H
#define TRIE_H

//System Libraries
#include <cstdlib>
#include <queue>
#include <string>
#include <iostream>
using namespace std;

class Trie
{

private:
    //Link stores the words associated with a node
    struct Link
    {
        string word;
        int priority;
        Link* next;
    };

    //Stores the numbers
    struct Node
    {
        Node* child[8];
        Link* head;
    };

    //Root of Trie
    Node* root;

    //Getters
    queue<int> getHashQueue(string);                    //Places string into queue of ints

    //Recursive add/delete functions
    void insertRec(Node*&,string,int,queue<int>);       //Inserts word into Node on Trie
        void removeRec(Node*&,string,queue<int>);       //Removes word from Node on Trie

    //Recursive destroyer helper functions
    void destroySubTree(Node*&);                        //Uses postorder traversal to delete Trie
    void destroyWordLinks(Link*);                       //Deletes Linked List from Trie

    //Recursive print functions
    void preorder(Node*);                               //Displays contents using preorder traversal
    void postorder(Node*);                              //Displays contents using postorder traversal

    //Recursive search functions
    string searchT9Rec(Node*&,queue<int>);              //Searches Trie by number (returns string)
    bool searchWordRec(Node*&,string,queue<int>);       //Searches Trie by string (returns boolean)

    //Recursive setters
    void setPriorityRec(Node*&,string,int,queue<int>);  //Changes priority of word
    void prioritizeRec(Node*&,string,queue<int>);       //Brings word to front of list in its node
    void deprioritizeRec(Node*&,string,queue<int>);     //Brings word to end of list in its node

public:
    //Constructor
    Trie();

    //Destructor
    ~Trie();

    //Print functions
    void preorder();                                    //Calls recursive preorder function
    void postorder();                                   //Calls recursive postorder function
    void breadth();                                     //Displays contents using breadth traversal

    //Add/remove functions
    void insert(string,int);                            //Calls recursive insert function
    void remove(string);                                //Calls revursive remove function

    //Search functions
    string searchT9(string);                            //Calls recursive search function
    bool searchWord(string);                            //Calls recursive search function

    //Getters
    string getHash(string,bool);                        //Returns hash
    //Setters
    void setPriority(string,int);                       //Calls recursive setPriority function
    void prioritize(string);                            //Calls recursive prioritize function
    void deprioritize(string);                          //Calls recursive deprioritize function
};
#endif /* TRIE_H */
```

Contents of Trie.cpp

```
/*
 * File:   Trie.cpp
 * Author: rachel
 *
 * Created on December 13, 2018, 9:38 PM
 */

//User libraries
#include "Trie.h"

//Default constructor
Trie::Trie()
{
    root = new Node;
    for(int i=0;i<8;i++)
    {
        root->child[i] = NULL;
    }
    root->head = NULL;
}

//Destructor
Trie::~Trie()
{
    destroySubTree(root);
}

/**
 * Destructor helper function that destroys nodes using postorder traversal
 * @param node
 */
void Trie::destroySubTree(Node*& node)
{
    if(!node)
    {
        return;
    }
    else
    {
        for(int i=0;i<8;i++)
        {
            destroySubTree(node->child[i]);
        }
        destroyWordLinks(node->head);
        delete node;
    }
}

/**
 * Destructor helper function that deletes the linked list associated with a node.
 * @param head
 */
void Trie::destroyWordLinks(Link* head)
{
    if(!head)
    {
        return;
    }
    else
    {
        Link* temp = new Link;
        temp = head;
        while(temp)
        {
            temp = temp->next;
            delete head;
            head = temp;
        }
    }
}

/**
 * Gets hash, then calls recursive insert function using the root as an argument
 * @param word
 * @param priority
 */
void Trie::insert(string word, int priority)
{
    queue<int> hash = getHashQueue(word);
    if(hash.empty())
    {
        return;
    }
    else
    {
    insertRec(root,word,priority,hash);
    }
}

/**
 * Recursively inserts a word into an existing node, or creates a new node if it doesn't already exist
 * @param node
 * @param word
 * @param priority
 * @param hash
 */
void Trie::insertRec(Node*& node, string word, int priority, queue<int>hash){
    if(!node)
    {
        node = new Node;
        for(int i=0;i<8;i++)
        {
            node->child[i] = NULL;
        }
        node->head = NULL;
    }
    if(hash.empty())
    {
        if(!node->head)
        {
            node->head = new Link;
            node->head->priority = priority;
            node->head->word = word;
            node->head->next = NULL;
        }
        else
        {
            Link* newLink = new Link;
            if(node->head->priority >= priority)
            {
                newLink->next = node->head;
                node->head = newLink;
                newLink->word = word;
                newLink->priority = priority;
            }
            else
            {
                Link* temp = new Link;
                while(temp->next && priority > temp->next->priority)
                {
                    temp = temp->next;
                }
                newLink->next = temp->next;
                temp->next = newLink;
                newLink->priority = priority;
                newLink->word = word;
            }
        }
    }
    else
    {
        int val = hash.front();
        hash.pop();
        insertRec(node->child[val],word,priority,hash);
    }
}

/**
 * Gets hash, then calls recursive remove function using the root as an argument
 * @param word
 */
void Trie::remove(string word)
{
    queue<int> hash = getHashQueue(word);
    if(hash.empty())
    {
        return;
    }
    else
    {
    removeRec(root,word,hash);
    }
}

/**
 * Recursively removes a word from a node
 * @param node
 * @param word
 * @param hash
 */
void Trie::removeRec(Node*& node, string word, queue<int> hash)
{
    if(!node)
    {
        cout << "'" << word << "'" << " not found - cannot delete.\n";
    }
    else
    {
        if(hash.empty())
        {
            if(!node->head)
            {
                cout << "'" << word << "'" << " not found - cannot delete.\n";
            }
            else if(node->head->word == word)
            {
                Link* temp = new Link;
                temp = node->head;
                delete node->head;
                node->head = temp->next;
            }
            else if(!node->head->next && node->head->word != word)
            {
                cout << "'" << word << "'" << " not found - cannot delete.\n";
            }
            else
            {
                Link* prev = new Link;
                prev = node->head;
                while(prev->next && prev->next->word != word)
                {
                    prev = prev->next;
                }
                if(prev->next->word == word)
                {
                    Link* dltLnk = new Link;
                    dltLnk = prev->next;
                    prev->next = dltLnk->next;
                    delete dltLnk;
                    dltLnk = NULL;
                }
                else
                {
                    cout << "'" << word << "'" << " not found - cannot delete.\n";
                }
            }
        }
        else
        {
            int val = hash.front();
            hash.pop();
            removeRec(node->child[val],word,hash);
        }
    }
}

/**
 * Using the input as the hash, calls recursive searchT9 function
 * @param input
 * @return
 */
string Trie::searchT9(string input)
{
    int currentNum;
    queue<int> hashQueue;
    for(int i=0;i<input.size();i++)
    {
        currentNum = (int)input[i] - 50;
        hashQueue.push(currentNum);
    }
    string word = searchT9Rec(root,hashQueue);
    return word;
}

/**
 * Returns head of link list associated with a given node using recursion
 * @param node
 * @param hashQueue
 * @return
 */
string Trie::searchT9Rec(Node*& node, queue<int> hashQueue)
{
    if(!node)
    {
        return "";
    }
    else
    {
        if(hashQueue.empty())
        {
            if(!node->head)
            {
                return "";
            }
            return node->head->word;
        }
        else
        {
            int val = hashQueue.front();
            hashQueue.pop();
            return searchT9Rec(node->child[val],hashQueue);
        }
    }
}

/**
 * Gets hash, then calls recursive searchWord function using the root as an argument
 * @param input
 * @return
 */
bool Trie::searchWord(string input)
{
    queue<int> hash = getHashQueue(input);
    if(hash.empty())
    {
        return false;
    }
    else
    {
    return searchWordRec(root,input,hash);
    }
}

/**
 * Recursively searches for a given node using hashQueue. If the node exists, searches for "word" in
linked list.
 * @param node
 * @param word
 * @param hashQueue
 * @return
 */
bool Trie::searchWordRec(Node*& node, string word, queue<int> hashQueue)
{
    if(!node)
    {
        return false;
    }
    else
    {
        if(hashQueue.empty())
        {
            if(!node->head)
            {
                return false;
            }
            else
            {
                Link* temp = new Link;
                temp = node->head;
                while(temp)
                {
                    if(temp->word == word)
                    {
                        return true;
                    }
                    temp = temp->next;
                }
                return false;
            }
        }
        else
        {
            int val = hashQueue.front();
            hashQueue.pop();
            return searchWordRec(node->child[val],word,hashQueue);
        }
    }
}

/**
 * Calls recursive preorder function using root as an argument.
 */
void Trie::preorder()
{
    preorder(root);
}

/**
 * Recursively prints node, then iterates through the children.
 * @param node
 */
void Trie::preorder(Node* node)
{
    if(!node)
    {
        return;
    }
    else
    {
        if(node->head)
        {
            cout << node->head->word << "\n";
        }
        for(int i=0;i<8;i++)
        {
            preorder(node->child[i]);
        }
    }
}

/**
 * Calls recursive postorder function using root as an argument.
 */
void Trie::postorder()
{
    postorder(root);
}

/**
 * Recursively iterates through the children, then prints node.
 * @param node
 */
void Trie::postorder(Node* node)
{
    if(!node)
    {
        return;
    }
    else
    {
        for(int i=0;i<8;i++)
        {
            preorder(node->child[i]);
        }
        if(node->head)
        {
            cout << node->head->word << "\n";
        }
    }
}

/**
 * Uses a queue to print upper level nodes first, then the leaves
 */
void Trie::breadth()
{
    Node* temp = new Node;
    Link* tempLink = new Link;
    queue<Node*> newQueue;
    newQueue.push(root);
    while(newQueue.size()>0)
    {
        temp = newQueue.front();
        if(temp->head)
        {
            tempLink = temp->head;
        }
        while(tempLink)
        {
            cout << tempLink->word << "\n";
            tempLink = tempLink->next;
        }
        newQueue.pop();
        for(int i=0;i<8;i++)
        {
            if(temp->child[i])
            {
                newQueue.push(temp->child[i]);
            }
        }
    }
}

/**
 * Calls getHash(), then puts it in a queue
 * @param key
 * @return
 */
queue<int> Trie::getHashQueue(string key)
{
    queue<int> hashQueue;
    string hash = getHash(key,false);
    int val;
    //Create a queue of integers to hold the hash
    for(int i=0;i<hash.length();i++)
    {
        val = (int)hash[i] - 48;
        hashQueue.push(val);
    }
    //Return queue
    return hashQueue;
}

/**
 * Using phone number-letter pairings, returns integers 2-9 if flag is false and 0-7 if true.
 * @param key
 * @param adjustFlag
 * @return
 */
string Trie::getHash(string key, bool adjustFlag)
{
    string hash = "";
    for(int i=0;i<key.length();i++)
    {
        if(!isalpha(key[i]))
        {
            cout << "Error! Must contain letters only.\n";
            return "";
        }
        if(!adjustFlag)
        {
            if(toupper(key[i])=='A' || toupper(key[i])=='B' || toupper(key[i])=='C')
            {
                hash+="0";
            }
            if(toupper(key[i])=='D' || toupper(key[i])=='E' || toupper(key[i])=='F')
            {
                hash+="1";
            }
            if(toupper(key[i])=='G' || toupper(key[i])=='H' || toupper(key[i])=='I')
            {
                hash+="2";
            }
            if(toupper(key[i])=='J' || toupper(key[i])=='K' || toupper(key[i])=='L')
            {
                hash+="3";
            }
            if(toupper(key[i])=='M' || toupper(key[i])=='N' || toupper(key[i])=='O')
            {
                hash+="4";
            }
            if(toupper(key[i])=='P' || toupper(key[i])=='Q' || toupper(key[i])=='R' ||
toupper(key[i])=='S')
            {
                hash+="5";
            }
            if(toupper(key[i])=='T' || toupper(key[i])=='U' || toupper(key[i])=='V')
            {
                hash+="6";
            }
            if(toupper(key[i])=='W' || toupper(key[i])=='X' || toupper(key[i])=='Y' ||
toupper(key[i])=='Z')
            {
                hash+="7";
            }
        }
        else
        {
            if(toupper(key[i])=='A' || toupper(key[i])=='B' || toupper(key[i])=='C')
            {
                hash+="2";
            }
            if(toupper(key[i])=='D' || toupper(key[i])=='E' || toupper(key[i])=='F')
            {
                hash+="3";
            }
            if(toupper(key[i])=='G' || toupper(key[i])=='H' || toupper(key[i])=='I')
            {
                hash+="4";
            }
            if(toupper(key[i])=='J' || toupper(key[i])=='K' || toupper(key[i])=='L')
            {
                hash+="5";
            }
            if(toupper(key[i])=='M' || toupper(key[i])=='N' || toupper(key[i])=='O')
            {
                hash+="6";
            }
            if(toupper(key[i])=='P' || toupper(key[i])=='Q' || toupper(key[i])=='R' ||
toupper(key[i])=='S')
            {
                hash+="7";
            }
            if(toupper(key[i])=='T' || toupper(key[i])=='U' || toupper(key[i])=='V')
            {
                hash+="8";
            }
            if(toupper(key[i])=='W' || toupper(key[i])=='X' || toupper(key[i])=='Y' ||
toupper(key[i])=='Z')
            {
                hash+="9";
            }
        }
    }
    return hash;
}
```

```
/**
 * Gets hash, then calls recursive setPriorty function using root as an argument
 * @param word
 * @param priority
 */
void Trie::setPriority(string word, int priority)
{
    queue<int> hash = getHashQueue(word);
    if(hash.empty())
    {
        return;
    }
    else
    {
        setPriorityRec(root,word,priority,hash);
    }
}

/**
 * Sets the priority of a given word to move its place in a node's linked list.
 * @param node
 * @param word
 * @param priority
 * @param hashQueue
 */
void Trie::setPriorityRec(Node*& node, string word, int priority, queue<int> hashQueue)
{
    if(!node)
    {
        cout << "'" << word << "'" << " not found - cannot set priority.\n";
    }
    else
    {
        if(hashQueue.empty())
        {
            if(!node->head)
            {
                cout << "'" << word << "'" << " not found - cannot set priority.\n";
            }
            else
            {
                Link* temp = new Link;
                temp = node->head;
                while(temp)
                {
                    if(temp->word == word)
                    {
                        temp->priority = priority;
                        return;
                    }
                    temp = temp->next;
                }
            }
        }
        else
        {
            int val = hashQueue.front();
            hashQueue.pop();
            setPriorityRec(node->child[val],word,priority,hashQueue);
        }
    }
}
```

```
/**
 * Gets hash, then calls recursive prioritize function using root as an argument
 * @param word
 */
void Trie::prioritize(string word)
{
    queue<int> hash = getHashQueue(word);
    if(hash.empty())
    {
        return;
    }
    else
    {
        prioritizeRec(root,word,hash);
    }
}

/**
 * Moves the position of a given word to the head of a linked list
 * @param node
 * @param word
 * @param hashQueue
 */
void Trie::prioritizeRec(Node*& node, string word, queue<int> hashQueue)
{
    if(!node)
    {
        return;
    }
    else
    {
        if(hashQueue.empty())
        {
            if(!node->head)
            {
                cout << "'" << word << "'" << " not found - cannot prioritize.\n";
            }
            else if(node->head->word == word)
            {
                cout << "'" << word << "'" << " is already prioritized.\n";
            }
            else if(!node->head->next && node->head->word != word)
            {
                cout << "'" << word << "'" << " not found - cannot prioritize.\n";
            }
            else
            {
                Link* prev = new Link;
                prev = node->head;
                while(prev->next && prev->next->word != word)
                {
                    prev = prev->next;
                }
                if(prev->next->word == word)
                {
                    Link* moveFront = new Link;
                    moveFront = prev->next;
                    prev->next = moveFront->next;
                    moveFront->next = node->head;
                    node->head = moveFront;
                    node->head->priority = 0;
                }
                else
                {
                    cout << "'" << word << "'" << " not found - cannot prioritize.\n";
                }
            }
        }
        else
        {
            int val = hashQueue.front();
            hashQueue.pop();
            prioritizeRec(node->child[val],word,hashQueue);
        }
    }
}

/**
 * Gets hash, then calls recursive prioritize function using root as an argument
 * @param word
 */
void Trie::deprioritize(string word)
{
    queue<int> hash = getHashQueue(word);
    if(hash.empty())
    {
        return;
    }
    else
    {
        deprioritizeRec(root,word,hash);
    }
}

/**
 * Moves the position of a given word to the end of a linked list
 * @param node
 * @param word
 * @param hashQueue
 */
void Trie::deprioritizeRec(Node*& node, string word, queue<int> hashQueue)
{
    if(!node)
    {
        return;
    }
    else
    {
        if(hashQueue.empty())
        {
            if(!node->head)
            {
                cout << "'" << word << "'" << " not found - cannot deprioritize.\n";
            }
            else if(!node->head->next)
            {
                if(node->head->word != word)
                {
                    cout << "'" << word << "'" << " not found - cannot deprioritize.\n";
                }
                else
                {
                    cout << "'" << word << "'" << " already deprioritized.\n";
                }
            }
            else
            {
                Link* prev = new Link;
                prev = node->head;
                while(prev->next && prev->next->word != word)
                {
                    prev = prev->next;
                }
                if(prev->next->word == word)
                {
                    Link* moveBack = new Link;
                    moveBack = prev->next;
                    prev->next = moveBack->next;
                    while(moveBack)
                    {
                        moveBack = moveBack->next;
                    }
                    moveBack->next = NULL;
                }
                else
                {
                    cout << "'" << word << "'" << " not found - cannot deprioritize.\n";
                }
            }
        }
        else
        {
            int val = hashQueue.front();
            hashQueue.pop();
            deprioritizeRec(node->child[val],word,hashQueue);
        }
    }
}
```