Programmation C/C++ Labo B (séances 7&8)	Structures de données en C++

Objectif	Implémenter la même fonctionnalité en utilisant différentes structures de données, et comparer la performance des diverses implémentations.
Matériel	Station Linux avec gcc et XEmacs Fichiers sources disponibles sur http://diwww.epfl.ch/w3lsp/teaching/prog3ssc/
Durée	6 heures

Introduction

Ce laboratoire vous propose de voir en pratique l'impact de divers facteurs sur la performance d'un programme. Vous allez donc améliorer les performances de deux exemples que vous avez vus au cours : la liste chainée et la recherche dans un dictionnaire. Pour chacun de ces exercices, nous vous fournissons le code de l'implémentation de base et vous donnons des indications pour optimiser leur performance.

Exercice 1. Liste chainée

Les allocations mémoire sont des opérations lentes. Il est donc préférable d'allouer beaucoup de mémoire en une fois plutôt que de faire de nombreuses petites allocations. Certains programmes qui font beaucoup d'allocations mémoire peuvent donc être limités par celles-ci plutôt que par la vitesse du processeur ou l'efficacité de l'algorithme utilisé. Dans de tels cas, la performance du programme peut être largement améliorée en essayant de réutiliser des éléments déjà alloués plutôt que de les créer et de les détruire à chaque utilisation.

L'implémentation de liste chainée vue au cours souffre de ce problème : chaque fois que l'on ajoute un élément à la liste, un nouveau nœud est créé. De même, chaque nœud supprimé de la liste est détruit. Plutôt que de créer et de détruire des nœuds à tour de bras, l'idée est d'avoir une réserve de nœuds inutilisés à disposition. Lorsqu'on ajoute un élément dans la liste, on peut ainsi prendre un nœud dans la réserve et l'ajouter dans la liste. Le nœud retournera dans la réserve lorsque l'élément qu'il représente sera supprimé de la liste.

Une telle liste chainée doit donc maintenir deux sous-listes : l'une contenant les éléments déjà ajoutés à la liste chainée, et l'autre contenant les nœuds qui ne sont pas utilisés. Un certain nombre de nœuds sont créés lors de la création de la liste chainée, et sont placés dans une liste qui fait office de réserve. On ajoute donc un élément dans la liste chainée en déplaçant un nœud depuis la liste de nœuds disponibles vers la liste de nœuds utilisés. Comme les nœuds ne font que passer d'une liste à l'autre, le nombre total de nœuds dans les deux listes est constant et aucune allocation ou déallocation n'est nécessaire après la création de la liste chainée.

Mise en place

Téléchargez les fichiers *timer.h* et *linkedlist.cpp* sur le site web du cours. Le fichier *timer.h* fournit une classe *Timer* contenant une méthode statique *get()*. Cette méthode retourne le nombre de secondes écoulées depuis un instant donné. On peut donc mesurer le temps d'exécution d'une portion de code en appelant *Timer::get()* au début et à la fin de cette portion, puis en prenant la différence entre les deux mesures.

Le fichier *linkedlist.cpp* fournit une implémentation complète d'une liste chainée telle que présentée au cours. La fonction *main()* crée une liste, ajoute des éléments en utilisant *pushBack()*, vide la liste, la remplit à nouveau en utilisant *pushFront()* et quitte. Le temps d'exécution des diverses opération est mesuré et affiché.

```
Exemple d'exécution de linkedlist

> g++ -Wall -o linkedlist linkedlist.cpp
> ./linkedlist 10000000

Allocation time: 0.000003 s
Adding elements at end: 0.909220 s (10000000 elements)
Clearing all elements: 0.425926 s
Adding elements at front: 0.596544 s (10000000 elements)
```

Question 1: Expérimentez en mesurant le temps d'exécution en fonction du nombre d'éléments ajoutés dans la liste. Comment varient les temps affichés lorsque l'on modifie le nombre d'éléments ajoutés ? Essayez avec 10'000, 100'000, 1 million et 10 millions d'éléments.

Le fichier *timer.h* contient en fait deux implémentations de la classe *Timer*, encadrées par des instructions *#ifdef*, *#else* et *#endif*. Les fonctions de mesure de temps ne sont pas les mêmes sous Windows et sous Unix (Cygwin, Linux et Mac OS X sont tous assimilables à des systèmes Unix), ce qui implique que le code qui compile sur une plateforme ne compile pas sur l'autre. On utilise donc la compilation conditionnelle pour savoir sur quelle plateforme on se trouve et différencier les cas.

La compilation conditionnelle se base sur les instructions du préprocesseur (qui sont traitées avant la compilation) pour savoir si une partie de code doit être compilée ou pas. Le programme contenu dans *ifdef.c* en illustre le fonctionnement avec un exemple simple. Dans le cas de la classe *Timer*, on utilise le fait que tous les compilateurs pour Windows définissent une valeur WIN32. On utilise donc cette valeur pour différencier les systèmes et compiler le code correct dans tous les cas.

Cette technique est donc très pratique pour produire du code multi-platforme, i.e. du code qui puisse être compilé et exécuté indépendamment du système d'exploitation.

Programme à rendre

Rendez un fichier *preallocatedlinkedlist.cpp* qui implémente une liste chainée utilisant dix millions d'éléments préalloués. La fonction *main()* contenue dans ce fichier sera identique à celle de *linkedlist.cpp*.

Question 2: Donnez les temps d'exécution de votre programme, en utilisant le même nombre d'éléments qu'à la question 1. Comment varient les temps en fonction du nombre d'éléments ajoutés ? Comparez ces temps avec ceux du programme original. Sont-ils meilleurs ? De quel facteur ? Expliquez les variations constatées.

Indications d'implémentation

Très peu de modifications sont nécessaires pour passer de la liste chainée telle qu'implémentée dans *linkedlist.cpp* et celle que nous vous demandons d'implémenter. Il est donc plus rapide de compléter le code que de le réécrire. En gardant la structure de *linkedlist.cpp*, seules quelques méthodes doivent être modifiées pour obtenir le résultat voulu, mais libre à vous d'utilisez une autre architecture plus efficace ou plus intuitive. Seul le code de la fonction *main()* ne doit pas être modifié.

Exercice 2. Dictionnaire de mots

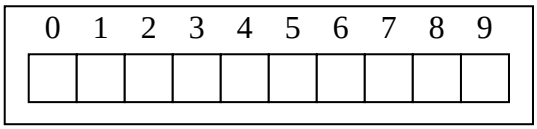
Le second exercice de ce laboratoire se base sur la recherche de mots dans un dictionnaire. La version qui vous a été présentée au cours était simple, mais elle est très lente et consomme inutilement de la mémoire en utilisant un tableau de taille fixe. Prenons comme exemple un dictionnaire qui stocke des nombres sous forme de chaines de caractères :

Entrée 0	1	2	4			124
Entrée 1	2	1	2			212
Entrée 2	1	2	4	9		1249
Entrée 3	2	9				29
Entrée 4						

Pour chercher si un nombre se trouve dans le dictionnaire, nous sommes obligés de parcourir tous les nombres qui y sont stockés. De plus, on voit qu'il y a de nombreuses cases vides, donc de l'espace mémoire qui est inutilisé.

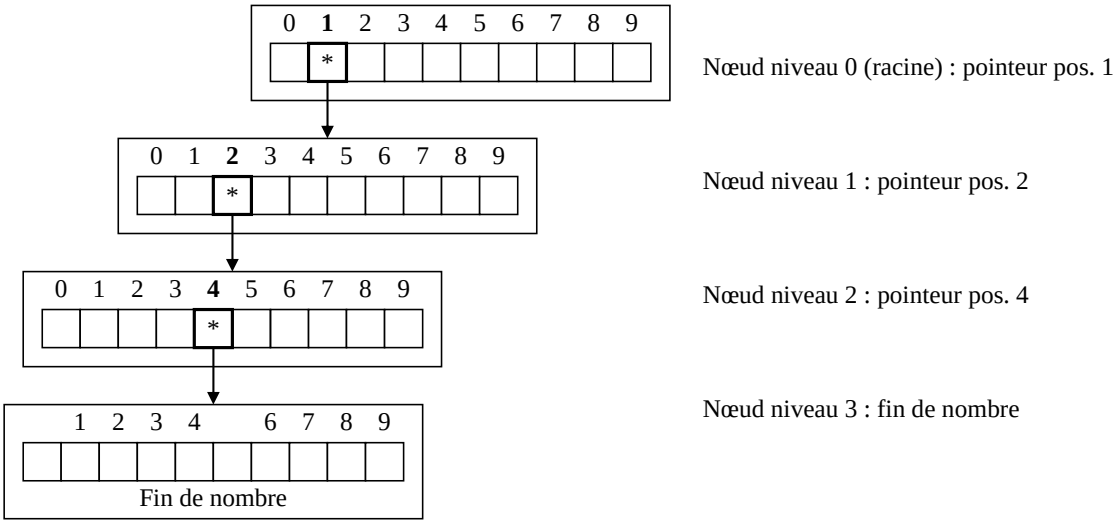
Ces deux caractéristiques peuvent être améliorées en implémentant notre dictionnaire sous forme d'arbre. L'arbre est constitué de nœuds représentant chacun un chiffre. Si un chiffre suit un autre chiffre dans un nombre, son nœud est un fils du nœud du chiffre qui le précède. Chaque chemin de l'arbre représente donc un nombre stocké dans le dictionnaire.

En pratique, chaque nœud contient un tableau de pointeurs vers d'autres nœuds. Le nombre d'éléments du tableau de pointeurs est égal au nombre de lettres de l'alphabet que nous utilisons. Dans notre exemple, il y a dix éléments puisqu'il y a dix chiffres.

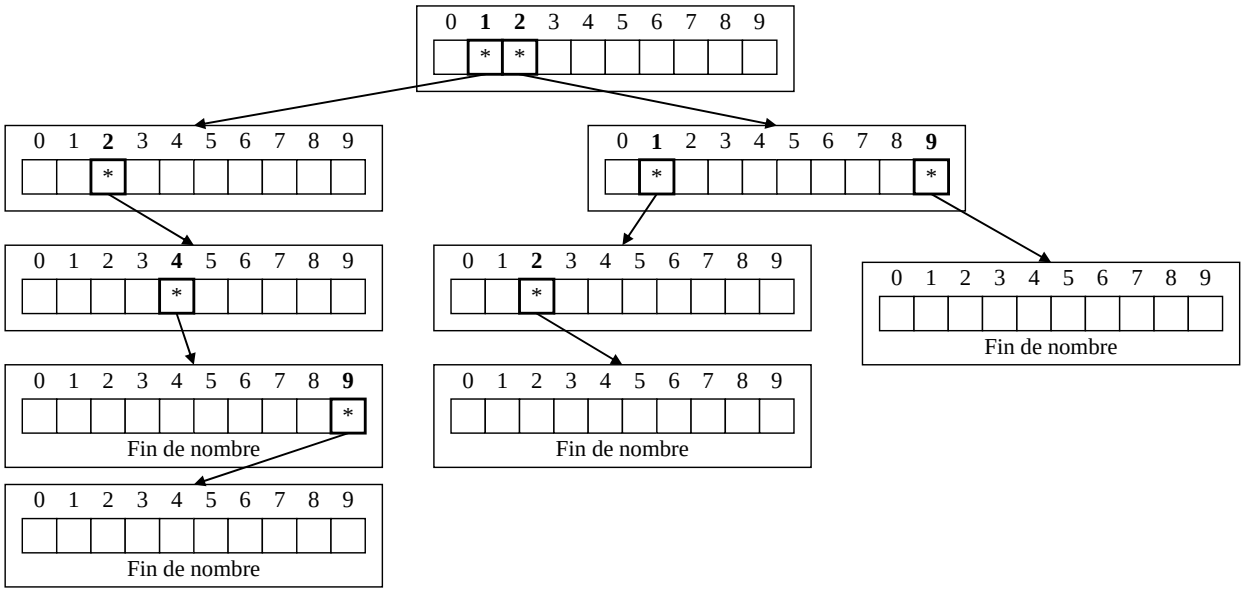


Un nombre est représenté en chainant les chiffres les uns aux autres, à partir d'un nœud racine. Le chiffre que représente un nœud est spécifié par la position à laquelle est stocké le pointeur vers le nœud successeur. La figure suivante représente le chainage de 1, 2 et 4 au nœud racine pour représenter le nombre 124.

Le pointeur vers le nœud suivant est stocké à la position 1 dans le tableau de pointeurs du nœud racine, donc on a au départ le chiffre 1. Le pointeur suivant est stocké dans la case 2 du tableau de pointeurs pointé par le nœud 1, donc le nœud 2 représente le chiffre 2. Le nœud 3 représente le chiffre 4, dernier chiffre du nombre.



On crée un arbre simplement en attachant plusieurs nœuds fils à un nœud, comme indiqué dans la figure ci-dessous. Les nombres stockés sont les mêmes que ceux du dictionnaire linéaire : 124, 212, 1249 et 29. Ce mode de stockage économise de la mémoire en ne stockant qu'une seule fois chaque préfixe: par exemple, le stockage du nombre 1249 réutilise les mêmes nœuds que pour le nombre 124. Il n'est ainsi pas suffisant qu'un nœud n'ait pas de fils pour qu'il termine un nombre. C'est pour cette raison que l'on indique la terminaison d'un nombre de manière explicite.



La recherche d'un nombre dans un tel dictionnaire consiste ainsi à trouver dans l'arbre un chemin partant de la racine dont les nœuds représentent les chiffres du nombre, en vérifiant que le dernier nœud trouvé représente effectivement une fin de nombre.

Le stockage de mots est réalisé de manière identique, sauf que chaque nœud peut avoir jusqu'à vingt-six fils (un par lettre) au lieu de dix (un par chiffre) lorsqu'on stocke des nombres.

Le but de cet exercice est donc d'implémenter les fonctionnalités du dictionnaire présenté au cours en stockant les mots dans un arbre au lieu de les stocker dans un tableau préalloué.

Mise en place

Commencez par télécharger sur le site web du cours le fichier *lineardict.cpp*, ainsi que les listes de mots. Ce programme est très proche de l'exemple du cours et stocke une liste de mots dans un tampon (ou buffer) alloué de manière statique avant de chercher des informations dans le dictionnaire. Compilez le programme et expérimentez avec différentes listes de mots et divers patterns de recherche.

Exemple d'exécution de lineardict

```
> g++ -Wall -o lineardict lineardict.cpp
> ./lineardict list-full.txt 3 t.me
636 3-letter words
tame
time
tome
```

Téléchargez aussi le fichier *timer.h* permettant de faire des mesures de temps. Mesurer des temps d'exécutions peut paraitre simple, mais il y a plusieurs choses auxquelles il faut faire attention. L'affichage sur le terminal est une opération très lente, ce qui fausse les résultats de mesure. De plus lorsque les opérations mesurées sont très courtes, leur durée et facilement influencée par des facteurs externes, comme les tâches de fond du système ou la résolution de la fonction de mesure de temps. Pour que les mesures soient significatives, elles doivent donc porter sur un grand nombre de répétitions.

Le code de mesure du programme *lineardict* prend en compte ces contraintes. Afin de ne pas devoir maintenir plusieurs versions du programme, on utilise la compilation conditionnelle pour compiler différentes versions du code selon que l'on veuille faire des mesures de performances ou non. On utilise la définition *MEASURE_PERF* à cet effet. Si *MEASURE_PERF* est définit, on inclut le fichier *timer.h*, on n'affiche pas les mots trouvés lors de la recherche de pattern, et on effectue un grand nombre de fois les diverses opérations de recherche. *MEASURE_PERF* peut être définie de deux façons: en utilisant l'instruction du préprocesseur *#define MEASURE_PERF*, ou en spécifiant la définition sur la ligne de commande du compilateur. (Si vous ne l'avez pas déjà fait, expérimentez avec la compilation conditionnelle en utilisant le code d'exemple *ifdef.c*.)

```
Exemple d'ecécution de lineardict, avec mesure de temps

> g++ -Wall -o lineardict lineardict.cpp -DMEASURE_PERF
> ./lineardict list-full.txt 3 t.me

Count time (3-letter words): 2.47198 s
Pattern matching time: 2.79419 s
```

Téléchargez maintenant les fichiers *treedictmain.cpp* et *treedictionary.h* sur le site web du cours. Le fichier source *treedictmain.cpp* contient une fonction *main()* se comportant exactement comme celle de *lineardict.cpp*. Afin de pouvoir utiliser la classe *TreeDictionary*, *treedictmain.cpp* inclut le fichier *treedictionary.h*. Celui-ci est un fichier en-tête définissant les classes *TreeDictionary* (représentant le dictionnaire) et *CharNode* qui représente un nœud de l'arbre.

Programme à rendre

Vous devez rendre deux fichiers (merci d'utiliser les mêmes noms):

- *treedictionary.h*
- *treedictionary.cpp*

En compilant ensemble *treedictionary.cpp* et *treedictmain.cpp*, vous devez obtenir un exécutable ayant exactement le même comportement que le dictionnaire linéaire.

Le but de cet exercice est d'implémenter un dictionnaire en arbre efficace. Vous n'êtes donc pas obligés de suivre les indications données ci-après. Libre à vous de modifier le contenu de *treedictionary.h* pour améliorer l'efficacité de votre implémentation. La seule contrainte est que votre code doit pouvoir être compilé avec la fonction *main()* officielle contenue dans *treedictmain.cpp*.

Une fois que tous les fichiers auront été rendus, nous les compilerons en utilisant la même fonction *main()* pour tous les groupes. Chaque projet sera compilé deux fois, avec et sans définition de *MEASURE_PERF*. Dans les deux cas, l'affichage devra être identique à celui de l'implémentation de référence fournie par *lineardict.cpp*. Les temps d'exécution seront ensuite mesurés et comparés.

Question 3: Comparez les temps d'exécution de votre programme avec ceux de *lineardict* et commentez les résultats obtenus. Faites vos mesures avec différentes listes de mots et avec les patterns '..es' et 'a.es'. Comment varient les temps d'exécution pour chacun des programmes ? Pouvez-vous expliquer pourquoi ?

Question 4: Estimez la quantité de mémoire nécessaire pour stocker le dictionnaire pour les deux programmes. Détaillez votre calcul.

Indications d'implémentation

La première chose à faire est d'avoir un programme qui compile, afin que vous puissiez tester les différentes méthodes individuellement. A cette fin, créez un fichier *treedictionary.cpp* qui inclut le fichier en-tête et implémente les méthodes qui y sont déclarées. Le corps des méthodes n'a aucune importance pour le moment, il doit juste être présent pour satisfaire le compilateur. Vous aurez peut-être aussi envie d'implémenter votre propre fonction *main()* pour mieux tester votre programme. Etant donné que vous aurez plusieurs versions de programme à tester, vous gagnerez du temps en utilisant un *makefile*.

Les makefiles permettent d'automatiser le processus de compilation. Vous trouverez sur le site web du cours un document qui décrit leur utilisation. Utilisez différentes règles pour générer les programmes avec et sans *MEASURE_PERF*, et avec l'une ou l'autre des fonctions *main()*.

Voyons maintenant le détail des différentes méthodes. Les indications ci-dessous se basent sur les prototypes fournis dans *treedictionary.h*, mais ne sont pas exhaustives. Une fois de plus, vous êtes libres d'ajouter, de supprimer ou de modifier ces méthodes (ou même les classes), dans la mesure où votre code peut être compilé avec la fonction *main()* officielle.

Classe *CharNode*

Un objet *CharNode* représente un nœud de l'arbre qui constitue le dictionnaire. Chaque nœud peut avoir jusqu'à vingt-six fils dont les adresses sont stockées dans un tableau de taille fixe. Un booléen est utilisé pour indiquer si le nœud courant marque la fin d'un mot ou non.

```
class CharNode
{
public:
    CharNode *sons[26];
    bool wordEnd;

public:
    CharNode();

    virtual ~CharNode();

    void findSuffix(const char *pattern, char *result, int currentDepth);

    int countWords(int suffixLength);
};
```

Le constructeur alloue et initialise le tableau de pointeurs vers les nœuds fils. Comme il n'y a pas de raison de vouloir conserver la fin d'un mot si on supprime une lettre qu'il contient, le destructeur d'un nœud ne doit pas seulement libérer sa propre mémoire, mais doit aussi détruire tous les fils.

La classe *CharNode* spécifie aussi deux méthodes de parcours d'arbre, utiles surtout si vous décidez d'implémenter les parcours d'arbre de manière récursive. La première, *findSuffix()*, recherche un pattern dans le sous-arbre dont la racine est le nœud courant. Elle peut fonctionner de manière récursive. Comme les nœuds ne connaissent pas leur parent, deux

arguments supplémentaires sont passés en paramètre: *result* stocke le mot trouvé jusqu'à présent, et *currentDepth* indique à quelle profondeur se trouve le nœud courant. A chaque itération, la méthode cherche un fils qui corresponde au prochain caractère du pattern à identifier, et modifie *result* en conséquence. Lorsqu'un mot complet correspondant au pattern a été trouvé, celui-ci est imprimé, pour autant que *MEASURE_PERF* ne soit pas définit. Les mots sont affichés au fur et à mesure qu'ils sont trouvés, donc aucune valeur de retour n'est nécessaire. La deuxième méthode retourne le nombre de mots qui se terminent à la profondeur donnée. Elle peut fonctionner de manière récursive ou non.

Classe *TreeDictionary*

Cette classe construit et maintient un dictionnaire à partir de nœuds *CharNode*. Comme chaque nœud est référencé par son parent, il est suffisant de stocker le nœud racine (qui est le seul sans parent) pour pouvoir atteindre le reste de l'arbre.

```
class TreeDictionary
{
private:
    CharNode *root;

public:
    TreeDictionary();

    virtual ~TreeDictionary();

    void addWord(const char *word);

    void matchPattern(const char *pattern);

    int countWords(int wordLength);
};
```

La construction n'a rien de particulier. La destruction du dictionnaire entraîne la destruction de tous les nœuds qu'il contient.

La méthode *addWord()* ajoute un mot dans l'arbre en créant les nœuds nécessaires.

La méthode *matchPattern()* trouve les mots respectant le pattern donné dans le dictionnaire. Dans sa forme récursive, elle se base sur la méthode *CharNode::findSuffix()*. Comme les mots sont affichés au fur et à mesure qu'ils sont trouvés, aucune valeur de retour n'est nécessaire.

La méthode *countWords()* compte le nombre de mots d'une longueur donnée. Dans sa forme récursive, elle se base sur la méthode *CharNode::countWords()* pour implémenter la fonctionnalité requise.

Annexe 1. timer.h

```
#ifndef TIMER_H
#define TIMER_H

// Define a helper class to measure time
// Two versions are provided for UNIX and Windows platforms
#ifndef WIN32

// UNIX code
#include <sys/time.h>
class Timer {
public:
    // Returns the number of seconds since the Epoch
    // (00:00:00 UTC, January 1, 1970)
    static double get() {
            struct timeval t;      // Use 'man gettimeofday' for info on timeval
            gettimeofday(&t,NULL); // structure and gettimeofday function
            return t.tv_sec+((double)t.tv_usec)/1000000;
    }
};
```

```
#else
// Windows code
#include <windows.h>
class Timer {
public:
    // Returns the number of seconds since the sytem was started
    // Time wraps around to zero every 49.7 days
    static double get() {
            // DWORD is an unsigned int
            DWORD t=GetTickCount();// GetTickCount returns a number of milliseconds
            return (double)t/1000.0;
    }
};
#endif
#endif
```

Annexe 2. linkedlist.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include "timer.h"

class LinkedListNode {
public:
    int el;               // Value of local element
    LinkedListNode *next; // Pointer to next list element
};

class LinkedList {
private:
    LinkedListNode *head, *tail; // Pointers to first and last nodes in list
    int size;                    // Size of list
public:
    LinkedList() { head=NULL; tail=NULL; size=0; }

    virtual ~LinkedList() { clear(); }

    // Add an element at the end of the list
    void pushBack(int el) {
            LinkedListNode *lln=new LinkedListNode();
            lln->el=el;
            if(head==NULL) // If list is empty
                    head=lln;
            else
                    tail->next=lln;
            tail=lln;
            size++;
    }

    // Add an element at the beginning of the list
    void pushFront(int el) {
            LinkedListNode *lln=new LinkedListNode();
            lln->el=el;
            if(head==NULL) // If list is empty
                    tail=lln;
            lln->next=head;
            head=lln;
            size++;
    }

    // Retrieve an element at a specified position in the linked list
    // no check is done that position is valid
    int elementAt(int pos) {
            LinkedListNode *current=head;
            for(int i=0;i<pos;i++)
                    current=current->next;
            return current->el;
    }

    // Return the number of elements in the linked list
    int getSize() { return size; }

    // Delete all nodes in the list
    void clear() {
            LinkedListNode *current=head;
            while(current!=NULL) {
```

```
                LinkedListNode *tmp=current;
                current=current->next;
                delete tmp;
        }
        head=NULL;
        tail=NULL;
        size=0;
    }

    // Print all elements
    void print() {
        LinkedListNode *current=head;
        while(current!=NULL) {
                printf("%d\n",current->el);
                current=current->next;
        }
    }
};

int main(int argc,char *argv[]) {
    if(argc!=2) {
        printf("Please specify nb elements to add\n");
        exit(1);
    }
    int elementsToAdd=atoi(argv[1]);
    double st=Timer::get();
    LinkedList ll;
    double et=Timer::get();
    printf("Allocation time: %f s\n",et-st);
    st=Timer::get();
    for(int i=0;i<elementsToAdd;i++) // Add elements at end of list
        ll.pushBack(i);
     et=Timer::get();
    printf("Adding elements at end: %f s (%d elements)\n",et-st,ll.getSize());
#ifdef DEBUG
    ll.print();
#endif
    st=Timer::get();
    ll.clear(); // Remove all elements from list
    et=Timer::get();
    printf("Clearing all elements: %f s\n",et-st);

    st=Timer::get();
    for(int i=0;i<elementsToAdd;i++) // Add elements at front of list
        ll.pushFront(i);
    et=Timer::get();
    printf("Adding elements at front: %f s (%d elements)\n",et-st,ll.getSize());
#ifdef DEBUG
    ll.print();
#endif
}
```

Annexe 3. lineardict.cpp

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

// Uncomment the following to measure the application performance
//#define MEASURE_PERF
#ifdef MEASURE_PERF
#include "timer.h" // Time measurement function
#endif

#define MAX_WORD_LENGTH 64
#define MAX_NB_WORDS 100000

// Matches a string against a given pattern. It prints the string if a match
// is found, or simply returns if lengths or content do not match
// The '.' is seen as a wild card that matches any character
// example: t.m. matches time, tome and temp; t.me matches time and tome
void matchPattern(const char *pattern, const char *word, size_t patternLength) {
    // Return if lengths or content do not match
    if(patternLength!=strlen(word))
        return;
    for(size_t i=0;i<patternLength;++i)
        if(pattern[i]!='.' && pattern[i]!=word[i])
```

```
                    return;
    #ifndef MEASURE_PERF
        // If we get here, we found a match
        std::cout << word << std::endl;
    #endif
    }

    int main(int argc, char *argv[]) {
        if(argc!=4) {
          std::cout<<"Usage: "<<argv[0] << " wordList wordLength pattern" << std::endl;
           return 1;
        }
        FILE *file=fopen(argv[1],"r");
        if(file==NULL) { // Check that the file could be opened
                std::cout << "Could not open file " << argv[1] << std::endl;
                return 1;
        }
        //  Words  are  stored  in  a  large  array  (capacity  MAX_NB_WORDS  words  of
MAX_WORD_LENGTH characters)
        char *words=(char*)malloc(MAX_WORD_LENGTH*MAX_NB_WORDS*sizeof(char));
        // Loops until the buffer is full or stream cannot be read any more
        int wordCount=0;
        while(wordCount<MAX_NB_WORDS
                && fgets(words+wordCount*MAX_WORD_LENGTH,MAX_WORD_LENGTH,file)!=NULL)
        {
                // Remove line ending characters, and check that line is not empty
                if(strtok(words+wordCount*MAX_WORD_LENGTH,"\r\n")!=NULL)
                        wordCount++;
        }
        //  Count number of words with wordLength characters
        int countN=0,i;
        int wordLength=atoi(argv[2]);
    #ifndef MEASURE_PERF
        for(i=0;i<wordCount*MAX_WORD_LENGTH;i+=MAX_WORD_LENGTH)
                if((int)strlen(words+i)==wordLength)
                        countN++;
        std::cout << countN << " " << wordLength << "-letter words" << std::endl;
    #else
        int j;
        double st=Timer::get();
        for(j=0;j<1000;++j) {
                countN=0;
                for(i=0;i<wordCount*MAX_WORD_LENGTH;i+=MAX_WORD_LENGTH)
                        if(strlen(words+i)==wordLength)
                                countN++;
        }
        double et=Timer::get();
        std::cout << "Count time (" << wordLength << "-letter words): "
                  << et-st << " s" << std::endl;
    #endif
        size_t len=strlen(argv[3]); // Precompute pattern length
    #ifndef MEASURE_PERF
        for(i=0;i<wordCount*MAX_WORD_LENGTH;i+=MAX_WORD_LENGTH)
                matchPattern(argv[3],words+i,len);
    #else
        st=Timer::get();
        for(j=0;j<1000;++j)
                for(i=0;i<wordCount*MAX_WORD_LENGTH;i+=MAX_WORD_LENGTH)
                        matchPattern(argv[3],words+i,len);
        et=Timer::get();
        std::cout << "Pattern matching time: " << et-st << " s" << std::endl;
    #endif
        return 0;
    }
```

Annexe 4. treedictionary.h

```
    #ifndef TREEDICTIONARY_H
    #define TREEDICTIONARY_H

    class CharNode {
    public:
        CharNode *sons[26]; // A fixed array containing pointers to sons
        bool wordEnd;       // Whether this character terminates a word

    public:
```

```
    CharNode(); // Constructor

    virtual ~CharNode(); // Destructor deletes all sons

    // Recursively find words that match the specified pattern. 'result' stores the
    // string found so far; 'currentDepth' indicates the current depth in the tree
    void findSuffix(const char *pattern, char *result, int currentDepth);

    // Recursively count the number of words that are complete at the given depth
    int countWords(int suffixLength);
};

class TreeDictionary {
private:
    CharNode *root; // The root of the tree
public:
    TreeDictionary(); // Constructor

    virtual ~TreeDictionary(); // Destructor

    // Add a word into the dictionary
    void addWord(const char *word);

    // Find words in the dictionary that match the specified pattern
    void matchPattern(const char *pattern);

    // Counts the number of words that contain the specified number of characters
    int countWords(int wordLength);
};
#endif
```

Annexe 5. treedictmain.cpp

```
#include <stdio.h>
#include <string>
#include "treedictionary.h"
#include "timer.h"

int main(int argc, char *argv[]) {
    if(argc!=4) {
            printf("Usage: %s wordsFile wordLength pattern\n",argv[0]);
            return 1;
    }
    //  Open file, C style
    FILE *fptr=fopen(argv[1],"r");
    if(fptr==NULL) {
            // File could not be opened, print error message
            printf("Could not open file %s",argv[1]);
            return 1;
    }
    //  Fill data structure
    TreeDictionary dict;
    char buf[64]; // Assume no word is longer than 64 characters
    // Scanning with %s ignores \r and \n so we don't need to handle line endings
    while(fscanf(fptr,"%s",buf)!=EOF)
            dict.addWord(buf);
    //  Count number of words with given length
    int wordLength=atoi(argv[2]);
#ifndef MEASURE_PERF
    int count=dict.countWords(wordLength);
    printf("%d %d-letter words\n",count,wordLength);
    //  Match pattern
    dict.matchPattern(argv[3]);
#else
    double st=Timer::get();
    int count;
    for(int j=0;j<1000;++j)
            count=dict.countWords(wordLength);
    printf("Count time (%d-letter words): %f s\n",wordLength,Timer::get()-st);
    st=Timer::get();
    for(int j=0;j<1000;++j)
            dict.matchPattern(argv[3]);
    printf("Pattern matching time: %f s\n",Timer::get()-st);
#endif
    return 0;
}
```