Rendu du projet de programmation orientée objet 2A MINDS

Mohamed Fyras Telmini

Fait en décembre 2020

1 Projet 1: ensembles

1.1 L'objet "ensemble":

L'objet "ensemble" contient 3 attributs:

1. Taille: capacité du tableau
2. Tab: tableau qui contient les éléments de l'ensemble
3. Indice: indice de la dernière case remplie, cet attribut est utile pour les méthodes que j'ai implémenté

```
class Ensemble
{   private:
    int taille; //
    int* tab; //tal
    int indice=0;
```

Figure 1: Attributs de l'ensemble

1.2 Les méthodes:

1.2.1 Constructeurs:

J'ai choisi de créer deux constructeurs, un qui crée un ensemble vide, l'autre qui crée un ensemble de taille donnée. Pour le premier constructeur, il n'a pas d'utilité tant que l'attribut "taille" n'a pas de valeur par défaut autre que 0.

1.2.2 Destructeur:

J'ai utilisé le destructeur par défaut " Ensemble(){}"

```
Ensemble(); //Constructeur d'
Ensemble(int n=0):taille(n){
tab = new int [taille];} //Co
```

Figure 2: Constructeurs l'ensemble

1.2.3 Affichage:

La méthode d'affichage vérifie si l'ensemble est vide et renvoie "ensemble vide" si c'est le cas. Sinon elle affiche les éléments du tableau un par un. L'affichage s'arrête à la dernière case remplie, ce choix me semble plus logique car il ne sert à rien d'afficher les cases vides et cela peut saturer l'affichage.

```
void Affiche(){ //affichage
    cout << "{";
    if (indice==0) {cout<<"ensemble vide";} //véri
    else{
        for (int i=0;i<min(indice,taille)-1;i++){
                cout<< tab[i]<<",";
                                    };
        cout<<tab[min(indice,taille)-1];} //affich
    cout<<"}"<<endl;
     };
```

Figure 3: Méthode d'affichage

1.2.4 Appartient:

Cette méthode est un intermédiaire que j'ai utilisé avant l'ajout, elle prend en entrée un entier, et retourne "true" si l'entier est présent dans l'attribut "tab" et "false" sinon. La recherche se fait par un simple parcourt de "tab" de 0 à "indice" (c'est obsolète de parcourir les cases vides).

```
bool Appartient(int n){
    for(int i=0;i<indice;i++)
    {
        if (tab[i]==n){
        return(true);}      //r
    }
    return(false);
}
```

Figure 4: Méthode de vérification d'appartenance

1.2.5 Ajout:

Avant d'ajouter un élément, la méthode vérifie si l'ensemble est saturé. Sinon, elle vérifie si l'élément appartient deja dans l'ensemble moyennant la méthode précédente qui retourne "true" si il y'a appartenance et "false" sinon.
Sinon, elle ajoute l'élément dans la case de l'attribut "tab" d'indice "indice" et décale l'attribut "indice" de 1.
à noter aussi que j'ai créé une copie de cette méthode appelée "ajoute-interne" qui ne fait pas un affichage en cas d'erreurs, elle est utilisée dans certaines méthodes qui vont suivre au lieu de l'ajout normal.

```
void Ajoute(int n){ //ajoute un entier à l'ensemble
    if (Appartient(n)){
            cout<< n<<" deja dans l'ensemble"<<endl;} //vérifie si l'entier exi
    else if ((indice==taille)&&(!(indice==0))){cout<<"ensemble plein"<<endl;} /
    else{
    tab[indice]=n; //ajoute l'élément
        indice++;}; //avance l'indice
};
```

Figure 5: Méthode d'ajout

Tests: La figure 9 est le résultat des tests des méthodes que je vient d'expliciter.

```
cout << "Test du constructeur:" << endl;

Ensemble E(2);
E.Affiche();
cout << "Test de l'ajout: " <<endl;
E.Ajoute(3);
E.Affiche();
E.Ajoute(3);
E.Affiche();
E.Ajoute(1);
E.Affiche();
E.Ajoute(4);
E.Affiche();
```

```
Test du constructeur:
{ensemble vide}
Test de l'ajout:
{3}
3 deja dans l'ensemble
{3}
{3,1}
ensemble plein
{3,1}
```

Figure 6: Tests de constructeurs/affichage/ajout

1.2.6 Méthode d'inclusion:

Cette méthode prend en entrée un ensemble (E1 disons), et elle vérifie si E1 est inclus dans l'ensemble qui lui fait appel (E2 disons). Elle retourne "true" si il y'a inclusion et "false" sinon. Un cas particulier se présente lorsque E1 est de cardinal (par rapport aux case deja remplies et non par rapport à la capacité totale) supérieur à E2, il ne peut pas y avoir d'inclusions donc elle retourne

directement "false". Sinon, la méthode fait le parcours des éléments de E1 (à ce stade on est sur qu'il est de cardinal plus petit) et cherche si il y'a un élément qui n'appartient pas à E2 (moyennant la méthode appartient). Si un élément n'appartient pas, elle retourne directement "false" et la boucle s'arrête. Si la boucle se termine alors elle retourne "true". J'ai choisi cette approche car elle me semble être la plus optimale.

```
bool Inclut(Ensemble a){ //vérifie l'inclusion de l'ensemble "a" d
    int i=0;
    if (a.indice>taille) {return(false);}; //un ensemble de cardin
    while (i<a.indice){if (!(Appartient(a.tab[i]))) return(false);
        i++;
    }
    return(true); //si tout les éléments sont inclus alors l'ensem

};
```

Figure 7: Méthode d'inclusion

1.2.7 Méthode d'intersection

Cette méthode prend en entrée un ensemble (E1) et retourne un ensemble (E3) qui est l'intersection de E1 avec l'ensemble qui a fait appel à la méthode (E2). Il existe un cas particulier où il y'a inclusion entre E1 et E2, l'intersection est alors l'un d'eux qui est de cardinal plus petit. Si il n'y a pas d'inclusion, alors la méthode crée un ensemble de taille de l'ensemble de cardinal plus petit (l'intersection entre deux ensembles ne peut pas être de cardinal plus grand que le min de leurs deux cardinaux). Elle parcourt E2 élément par élément et fait appel à la méthode ajoute dès qu'un élément de E2 appartient à E1 (moyennant la méthode d'appartenance). J'ai choisi cette approche car elle me semble être la plus optimale en calcul et en mémoire.

1.2.8 Méthode d'union

Cette méthode prend en entrée un ensemble (E1) et retourne un ensemble qui est l'union de cet ensemble et de l'ensemble qui lui fait appel (E2). Un cas particulier est celui où il y'a inclusion entre les deux ensembles, alors leurs union est l'ensemble de cardinal plus grand entre eux. Sinon la méthode crée un ensemble de taille (Card(E1)+Card(E2)-Card(intersection(E1,E2))) et fait le parcours de E2 élément par élément, elle l'ajoute en sa totalité à l'ensemble d'union. Puis elle fait le parcours de E1 élément par élément, et ajoute les éléments qui ne sont pas déjà dans l'union (la vérification se fait automatiquement dans la méthode d'ajout, ici j'utilise ajoute-interne pour ne pas avoir un affichage si un élément existe déjà). Cette approche me semble être la plus optimale en calcul et mémoire.

```
Ensemble Intersection(Ensemble a){ //retourne un ensemble qui est
if (Inclut(a)) { return(a);};           //cas particulier: si un
if (a.Inclut(*this)) {return(*this);}
else{
    Ensemble E(min(indice,a.indice)); //création d'un ensemble de
    for(int i=0;i<a.taille;i++){
        if (Appartient(a.tab[i])) {E.Ajoute_interne(a.tab[i]);};
    };
    return(E); //ensemble de sortie
}



};
```

Figure 8: Méthode d'intersection

```
Ensemble Union(Ensemble a){
    if (Inclut(a)) {return(*this);} //cas particulier
else if (a.Inclut(*this)) {return(a);}
else{
Ensemble E(indice+a.indice-Intersection(a).indice); /
            //ajout de chaque ensemble à la sortie, l
            for(int i=0; i<a.indice;i++)
            {
                E.Ajoute_interne(a.tab[i]);};

            for(int i=0; i<indice;i++)
            {
                E.Ajoute_interne(tab[i]);};
return(E); //ensemble de sortie
}};
```

Figure 9: Méthode d'union

Tests de l'inclusion, intersection et union: Les tests commencent par la création et remplissage de nouveaux ensembles. Les tests sont explicités dans la figure 11 et les résultats dans la figure 24

```
cout<<"Exemples d'ensembles:"<<endl;
cout<<"E:";
E.Affiche();
cout << "E4:";
E4.Affiche();
cout << "E5:";
E5.Affiche();
```

```
Exemples d'ensembles:
E:{3,1}
E4:{5,4,1,3}
E5:{9,8,4,6}
```

Figure 10: Ensembles créés pour les tests

```
//début tests
cout<<"Test d'inclusion:"<<endl;
cout<<"E4 inclut dans E? " <<E.Inclut(E4)<<endl;
cout<<"E inclut dans E4? " <<E4.Inclut(E)<<endl;
cout<<"E inclut dans E5? " <<E5.Inclut(E)<<endl;

cout<<"Test d'intersection"<<endl;
cout << "intersection entre E et E4: ";
E.Intersection(E4).Affiche();
cout << "intersection entre E et E5: ";
E.Intersection(E5).Affiche();
cout << "intersection entre E5 et E: ";
E5.Intersection(E).Affiche();
cout << "intersection entre E5 et E4: ";
E5.Intersection(E4).Affiche();
cout << "intersection entre E4 et E5: ";
E4.Intersection(E5).Affiche();

cout<<"Test d'union"<<endl;
cout << "union entre E et E4: ";
E4.Union(E).Affiche();
cout << "union entre E et E5: ";
E5.Union(E).Affiche();
cout << "union entre E4 et E5: ";
E5.Union(E4).Affiche();
cout << "union entre E5 et E4: ";
E4.Union(E5).Affiche();
```

Figure 11: Codes des tests de l'inclusion, intersection et union

```
Test d'inclusion:
E4 inclut dans E? 0
E inclut dans E4? 1
E inclut dans E5? 0
Test d'intersection
intersection entre E et E4: {3,1}
intersection entre E et E5: {ensemble vide}
intersection entre E5 et E: {ensemble vide}
intersection entre E5 et E4: {4}
intersection entre E4 et E5: {4}
Test d'union
union entre E et E4: {5,4,1,3}
union entre E et E5: {3,1,9,8,4,6}
union entre E4 et E5: {5,4,1,3,9,8,6}
union entre E5 et E4: {9,8,4,6,5,1,3}
```

Figure 12: Résultats des tests de l'inclusion, intersection et union

2 Projet 2: Liste de couples

2.1 L'objet couple

2.1.1 Attributs

1. Clef: clef du couple
2. Valeur: valeur du couple

2.1.2 Méthodes

1. Constructeurs: constructeur d'un couple (0,0) et d'un couple (clé,valeur)
2. get-clef: retourne la clé du couple, puisque celle-ci est en "private"
3. get-val: retourne la valeur du couple, puisque celle-ci est en "private"
4. affiche: affichage d'un couple
5. Destructeur: J'ai utilisé le destructeur par défaut " Couple(){}"

```
class Couple
{   private:
    int clef; //clef du couple
    float valeur; //valeur associée
    public:
    Couple(){clef=0; valeur=0;}; //constructeur de coup
    Couple(int a,float b):clef(a),valeur(b){}; //constr
    int get_clef(){return clef;}; //récupération de la
    float get_val(){return valeur;}; //récupération de
    void afficher(){cout<<"{"<<clef<<";"<<valeur<<"}";}
    ~Couple(){}; //destructeur par défaut
};
```

Figure 13: L'objet couple

2.2 L'objet noeud

2.2.1 Attributs

1. contenu: couple associé au noeud
2. suivant:pointeur sur le noeud suivant

Les attributs ne sont pas en "private" pour éviter certaines erreurs, je sais qu'elles auraient dû l'être mais j'ai préféré avoir un programme stable que de risquer des erreurs.

2.2.2 Méthodes

1. constructeurs: constructeur d'un noeud avec couple et sans suivant, constructeur d'un noeud avec couple et suivant
2. get-couple: retourne le couple du noeud
3. get-suivant: retourne le suivant du noeud
4. affichage: afficher un noeud, afficher-fin est utile pour l'objet liste pour afficher correctement le noeud final
5. Destructeur: J'ai utilisé le destructeur par défaut " Noeud(){}"

```
class Noeud
{
    public:
    Couple contenu; //contenu
    Noeud* suivant; //pointeur sur le noeud suivant



    Noeud(Couple c):contenu(c),suivant(NULL){}; //cons
    Noeud(Couple c, Noeud* n):contenu(c),suivant(n){};
    //ces deux fonctions sont obsolètes puisque les at
    Couple get_cont(){return contenu;}; //récupère le
    Noeud* get_suiv(){return suivant;}; //récupère le
    //
    void afficher(){contenu.afficher(); //affichage
    cout<<",";}
    void afficherfin(){contenu.afficher();} //affichag
    ~Noeud(){}; //destructeur par défaut
};
```

Figure 14: L'objet noeud

2.3 L'objet Liste

2.3.1 Attributs

1. debut: 1er élément de la liste
2. fin: dernier élément de la liste
3. taille: taille de la liste

2.3.2 Méthodes

2.3.3 Constructeurs:

J'ai employé 3 constructeurs:

1. Constructeur d'une liste vide
2. Constructeur d'une liste avec début et fin et taille
3. Constructeur par copie

```
Liste(){debut=NULL; //constructeur d'une li
fin=NULL;
taille=0;};
Liste(Noeud* n1,Noeud* n2,int n3){debut=n1;
fin=n2;
taille=n3;};
Liste (const Liste & L){taille = L.taille;
debut = L.debut;
fin = L.fin;}
```

Figure 15: Constructeurs de la liste de couples

2.3.4 Méthodes de récupération d'attributs:

1. get-taille: retourne la longeur de la liste
2. get-debut: retourne le 1er noeud de la liste

```
int get_taille(){return taille;};
Noeud* get_debut(){return debut;};
```

Figure 16: Récupérateurs d'attributs

2.3.5 Méthode de vérification de l'existence de la clef:

Cette méthode est utile pour l'ajout d'éléments et la recherche d'un élément étant donné sa clef. Elle prend en entrée un noeud et crée un noeud de parcours égal au début de la liste, puis parcours chaque noeud de la liste en cherchant une égalité de sa clef avec la clef de l'entrée. Si il y'a égalité elle retourne 'true' sinon 'false'.

```
bool verifclef(Noeud* n){        //vérifie l'existance de la c
Noeud* deb=debut; //variable de parcours
    for(int i=0;i<taille;i++)
    {   if((deb->contenu.get_clef()==n->contenu.get_clef()))
    {
        return(true); //retourne true si il y a égalité
    }
       if (!(deb->suivant==NULL)){ //mise à jour du pointeu
    deb=deb->suivant;}
    }
    return(false); //fin de boucle = pas d'égalité

}
```

Figure 17: Vérification de l'existence d'une clef associée à un noeud

2.3.6 Méthode d'ajout de noeud

cette méthode permet d'insérer un noeud dans la liste selon l'indice croissant des clefs des noeuds de la liste. Elle prend en entrée le noeud à ajouter. Elle vérifie si la liste est vide et si elle l'est, elle insère le noeud au début.
Si la liste n'est pas vide, elle fait le parcours de la liste en vérifiant à chaque itération si la clef de l'entrée est inférieure à la clef de l'indice de parcours. J'ai isolé le cas où la clef de l'entrée est inférieure à la clef du début pour éviter des erreurs. J'ai créé deux noeuds de parcours pour toujours avoir accès au "précédent" de l'élément courant pour permettre l'insertion du noeud entre les deux. L'insertion se fait par modification des attributs "suivant" des noeuds concernés. Si le parcours se termine sans insertion, alors le noeud d'entrée est inséré à la fin de la liste. à noter que chaque ajout augmente l'attribut "taille" de la liste de 1.

2.3.7 Affichage

Cette méthode permet l'affichage du contenu de la liste moyennant un noeud de parcours, l'affichage de chaque noeud se fait par la méthode "afficher" des noeuds, l'affichage du dernier élément est isolé pour ne pas avoir une virgule après lui.

2.3.8 Valeur d'un noeud associé à une clef (Rechercher)

C'est la méthode "recherche" demandée par l'énoncé, j'ai juste employé un nom différent. Cette méthode prend en entrée un entier "clef" et fait le parcours de la liste moyennant un noeud de parcours. Si il y'a égalité entre l'entrée et la clef du noeud, elle retourne la valeur qui lui est associée (float). Si le parcours se termine sans égalité, alors la clef n'existe pas et la fonction fait un affichage de l'erreur et retourne NULL.

```
    void append(Noeud* n){ //ajoute un noeud à la liste
    if (debut==NULL){ //si la liste est vide
            debut=n;
            n->suivant=NULL;
            fin=n;
            taille++; return;}
    else if((verifclef(n))){cout<<"clef existante"<<endl; return;} //si la clef existe deja, n'ajoute pas
    else { //si la liste n'est pas vide et que la clef n'existe pas deja
            if (n->contenu.get_clef() < debut->get_cont().get_clef()) { n->suivant=debut; debut=n ; taille++; return;}
            else{ //si la clef n'est pas inférieure au 1er élément
                Noeud* indice; //noeud de parcourus
                Noeud* inter; //noeud qui garde le précédent du noeud de parcours
                indice=debut; //initialisation de l'indice de parcours
                        while(indice !=fin){ //parcours tant que l'indice n'est pas égal au dernier noeud de la liste
                            inter=indice; //récupération du précédent
                            indice=indice->suivant; //avance de l'indice de parcours
                                if (n->contenu.get_clef() < indice->get_cont().get_clef()) { //vérification si la clef
                                        inter->suivant=n; //si oui, insérer le noeud entre inter et indice
                                        n->suivant=indice;
                                        taille++; return; } //fin de la boucle si il y'a insertion
                                        };
                indice->suivant=n; //sinon, l'élément doit être inséré à la fin de la liste
                fin=n;
                taille++;
                }
}; //fin append
```

Figure 18: Ajout d'un noeud à la liste

```
void afficher(){ //affichage
    Noeud* temp; //noeud de parcour
    temp=debut; //initialisation
    cout<<"[";
    for(int i=0;i<taille-1;i++){ //
     temp->afficher();
     temp=temp->suivant;


    }
    temp->afficherfin(); //affichag
    cout<<"]"<<endl;
    }
```

Figure 19: Affichage

2.3.9 Suppression d'un noeud étant donné la clef

Cette méthode prend en entrée un entier "clef", elle vérifie l'existence de cette cléf moyennant la méthode "valeur" qui retourne NULL si la clef n'existe pas. Puis si la clef existe, la méthode fait le parcours de la liste moyennant des noeuds de parcours (même logique que l'ajout). J'ai isolé le cas de suppression du 1er indice pour éviter des erreurs. La suppression se fait en reliant le noeud qui précède le noeud à supprimer en celui qui succède ce dernier (suivant du précédent devient égal au suivant du noeud à supprimer).

```
float valeur(int n) //récupération de la
{   Noeud* deb=debut; //initialisation du
    for(int i=0;i<taille;i++) //boucle de
    {   if((deb->contenu.get_clef()==n))
    {
        return(deb->contenu.get_val()); /
    }
       if (!(deb->suivant==NULL)){ //véri
    deb=deb->suivant;}
    }
    cout<<"clef inexistante"<<endl; //sir
    return(NULL);
}
```

Figure 20: Récupération de la valeur d'un noeud associé à une clef (Rechercher)

```
void supprime(int n){
    if (!(valeur(n)!=NULL)){return;} //si la clef n'existe pas, ne fait rien
    else{ //si la clef existe
        Noeud* deb; //initialisation des noeuds de parcours
        Noeud* f;
        Noeud* interm;
        deb=debut;
        if(deb->contenu.get_clef()==n){debut=deb->suivant;taille--; return;}
        interm=debut; //initialisation des noeuds de parcours
        deb=deb->suivant;
        for (int i=0;i<taille-1;i++) //boucle de recherche du noeud avant la
        {
            if ((deb->contenu.get_clef()==n)) //si égalité des clefs
            {
             interm->suivant=deb->suivant;taille--;return; //on "coupe" le n
            };

            deb=deb->suivant; //mise à jour des noeuds de parcours
            interm=interm->suivant;
        };
    };
}
```

Figure 21: Suppression d'un noeud associé à une clef

2.3.10 Destructeur

J'ai utilisé le destructeur par défaut " Liste(){}"

2.3.11 Tests

```
//création de couples
Couple cp1(4,1.2);
Couple cp2(2,18);
Couple cp3(6,4.3);
Couple cp4(3,1552);
Couple cp5(3,123);
Couple cp6(5,7.3);
//création de noeuds
Noeud n1(cp1);
Noeud n2(cp2);
Noeud n3(cp3);
Noeud n4(cp4);
Noeud n5(cp5);
Noeud n6(cp6);
//création de liste
Liste a;
```

Figure 22: Initialisations des tests

```
cout<<"test d'ajout et affichage"<<endl;
//test d'ajout et affichage
a.append(&n1);
a.afficher();
a.append(&n2);
a.afficher();
a.append(&n3);
a.afficher();
a.append(&n4);
a.afficher();
a.append(&n5);
a.afficher();
a.append(&n6);
a.afficher();
//test du constructeur par copie
cout<<"test du constructeur par copie"<<endl;
Liste b(a);
b.afficher();
cout<<"test de suppression (supprime 2 puis supprime 3)"<<endl;
//test de suppression
a.supprime(2);
a.afficher();
a.supprime(3);
a.afficher();
cout<<"test de recherche (recherche 5, 4, 2)"<<endl;
//test de recherche
cout<<a.valeur(5)<<endl;
cout<<a.valeur(4)<<endl;
cout<<a.valeur(2)<<endl;
```

Figure 23: Code des tests

Figure 24: Résultats des tests