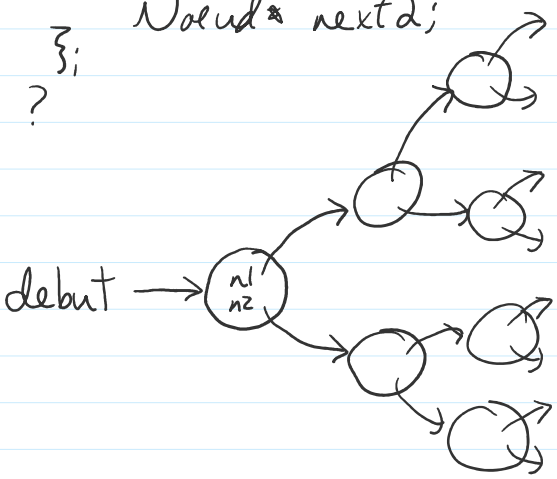
Arbres

30 septembre 2024 14:05

Il se passe quoi si on fait

```
struct Noeud{
    TYPE val;
    Noeud* next1;
    Noeud* next2;
};
```

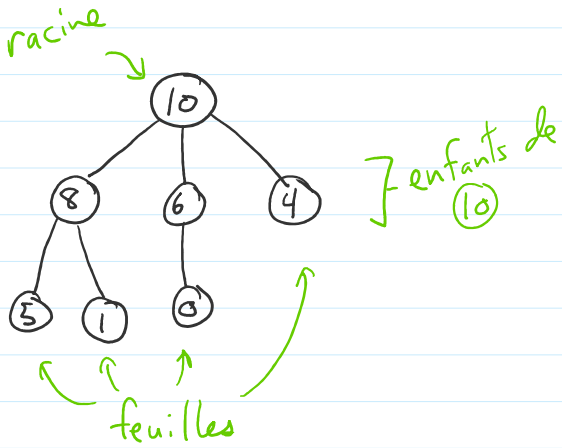
?



On a un arbre !

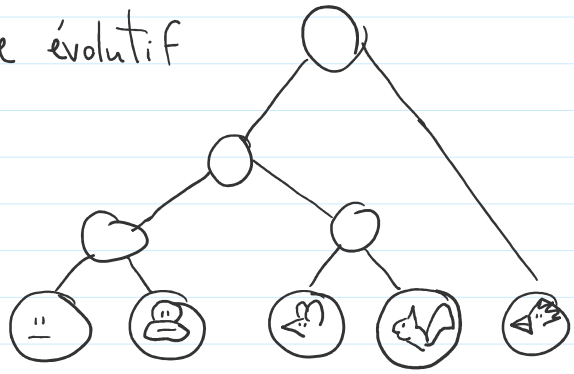
Arbre

- stocke elts avec une hiérarchie
- un noeud de départ, qui est appelé la racine.
  - attention: le concept d'arbre non-enraciné existe (IFT436)
- les successeurs d'un noeud sont appelés ses enfants,
  - enracinent un arbre
  - un noeud a 0, 1, ou plusieurs enfants
- chaque noeud a un parent (pas plus), sauf la racine qui n'a pas de parent
- feuille = un noeud avec 0 enfant



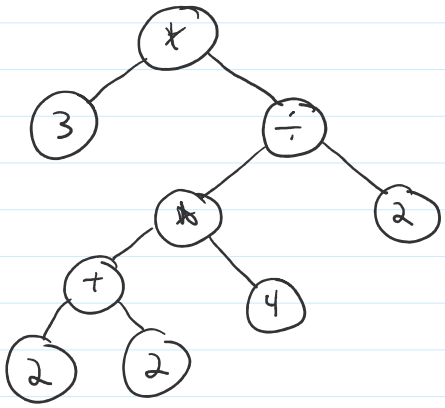
- noeud interne = noeud qui n'est pas une feuille

ex: arbre évolutif



expression math

$$3 * ((2+2) * 4) \div 2$$



Utilisation:

```
calculer(Noeud* v){

    si v->est_feuille()
        return v->val;

    int r1 = calculer(v->enfant1);
    int r2 = calculer(v->enfant2);

    return v->appliquer_op(r1, r2);
}
```

Appel initial: `calculer(racine);`

Représentation d'un arbre

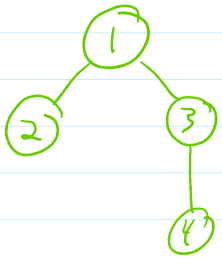
① Utiliser Noeud directement (et être prudent avec les new)

```
void main(){

    Noeud* racine = new Noeud();
    racine->val() = 1;
    racine->add_enfant(2);

    Noeud* v = racine->add_enfant(3);
    v->add_enfant(4);

    delete racine;
}
```




```
template <typename T = int>
class Noeud{
private:
    T val;
    vector<Noeud*> enfants;
    Noeud* parent;

public:

    Noeud(Noeud* parent = nullptr){
        this->parent = parent;
    }

    TYPE& val(){ return val; }

    bool est_feuille(){
        return enfants.empty();
    }

    bool est_racine(){
        // ??
```

```
bool est_racine(){
    return !parent;     //wat?
}

Noeud* add_enfant(T& val){
    Noeud* v=new Noeud(this);
    v->val()=val;
    enfants.push_back(v);
    return v;
}

size_t get_nb_enf(){return enfants.size();}

Noeud* get_enf(size_t i){return enfants[i];}

~Noeud(){
    for(size_t i=0; i<enfants.size(); ++i)
        delete enfants[i];
    enfants.clean();
}

};
```

② Encapsuler dans une classe Arbre (ou Tree)

```
...
class Arbre{
    class Noeud{
    ...
    };
    Noeud* racine;
    ...
```

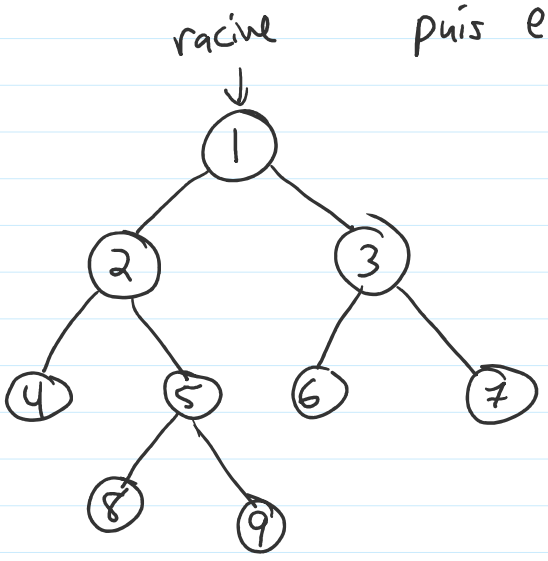
```
    Noeud* get_racine(){ return racine; }
}
```

- Utile si votre SDD utilise un arbre à l'interne seulement
- Pour manipuler l'arbre, pas utile.

- Un arbre n'ordonne pas les elts
- On peut ordonner les noeuds selon leur <u>ordre de visite</u> dans un parcours.

- Types de parcours:

1) Post-ordre: quand on arrive à un noeud, on visite ses enfants, puis ensuite le noeud.




```
void postordre(Noeud* v){
    for(int i=0; i<v->get_nbenf(); ++i){
        postordre(v->get_enfant(i));
    }
    cout<<v->val<<" "; //visiter v
}
```

Appel init: postordre(racine);

4-8-9-5-2-6-7-3-1 Ordre des noeuds

2) Pré-ordre: on visite le noeud, puis ses enfants.

```
void preordre(v){
    cout<<v->val;
    for(i=0..v->get_nb_enf()){
        preordre(v->get_enfant(i));
    }
}
```

1-2-4-5-8-9-3-6-7

3) in-ordre: si chaque noeud a 0 ou 2 enfants,
on visite
- l'enfant gauche
- le noeud
- l'enfant droit

```
void inordre(v){
    if(v->get_nb_enf()>0)
        inordre(v->get_enf(0));
    cout<<v->val;
    if(...)
        inordre(v->get_enf(1));
}
```

4-2-8-5-9-6-3-7