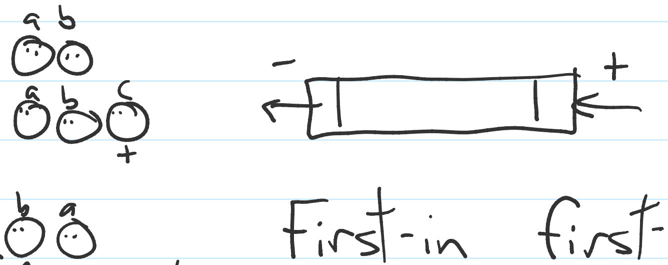
Listes/files/piles et listes chaînées

23 septembre 2024 12:59

TDA liste: ajout/suppression au début/milieu/fin
accès à l'elt i
get taille

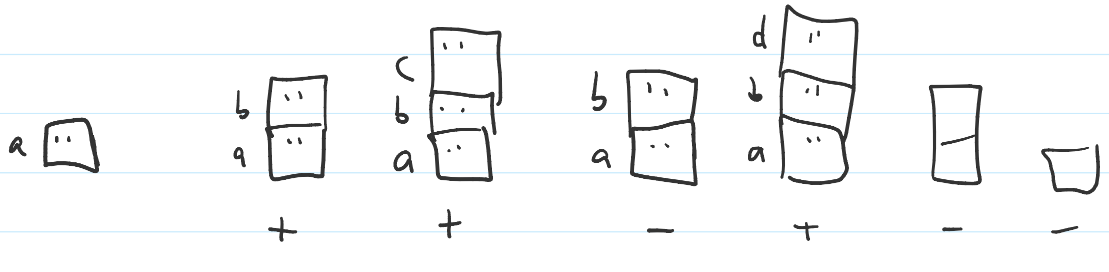
file: ajouter à la fin (enfiler/enqueue)
voir début
retirer elt au début (défiler, dequeue)



First-in first-out (fifo)

double-file: ajout début/fin (deque)
delete début/fin

pile: ajouter au sommet
voir sommet
retirer sommet



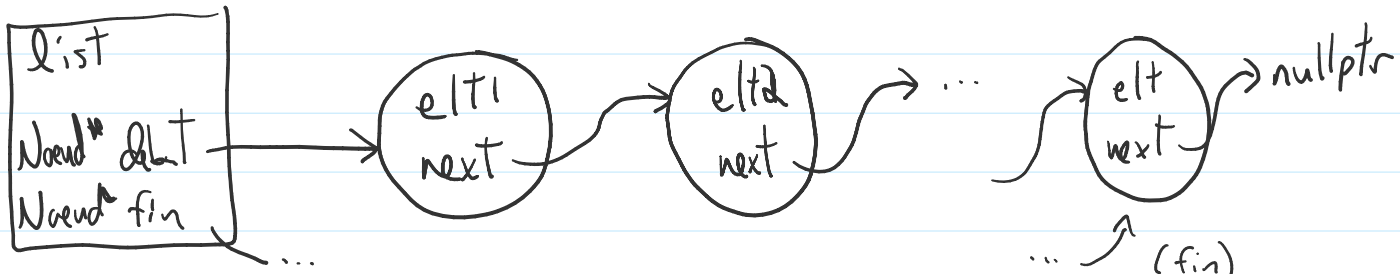
last-in first-out (LIFO)

Implémentations: tableau
tableau circulaire (deque)
→ liste chaînée (linked list)

Tableau circulaire (deque)
→ liste chaînée (linked list)

<u>Liste chaînée</u>: structure dans laquelle chaque elt a un pointeur vers l'elt suivant.

Objet <u>Noeud</u> pour contenir elt+ptr
↳ cellule dans notes de cours

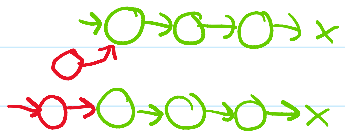


- Pour encapsuler, on peut faire une classe interne.

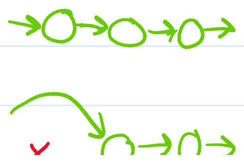
```
template <typename T>
class list{
private:
        struct Noeud{
            T elt;
            Noeud* next;
        };
        Noeud* debut;
        Noeud* fin;
        int nbelem;

public:
        list(){
```

```
public:
    list(){
        nbelem = 0;
        debut = fin = nullptr;
    }

    void push_back(const T& elt){

        Noeud* n = new Noeud();    //ToDo: delete

        n->elt = elt;
        n->next = nullptr;

        if (!debut)
            debut = fin = n;
        else {
            fin->next = n;
        }
    }
```

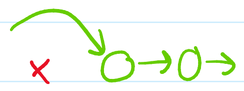
temps O(1)



```
    void push_front( elt){

        Noeud* n = new Noeud();
        n->elt = elt;
        n->next = debut ;    //ok si vide

        debut = n;
        if (!fin)
            fin = n;
    }

    void pop_front(){
```

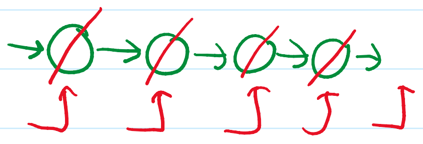


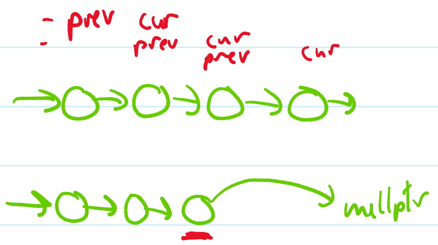
```
void pop_front(){
    if(!debut) return;

    Noeud* old = debut;
    debut = old->next;
    delete old;

    if (fin == old)
        fin = debut;
}
```



```
~list(){
    if (debut){
        Noeud* cur = debut;
        while (cur){
            Noeud* tmp = cur;
            cur = cur->next;
            delete tmp;
        }
    }
}
```

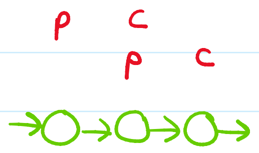
$O(n)$





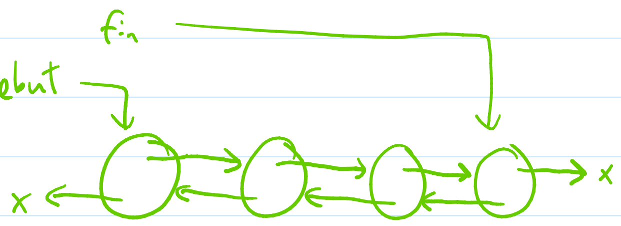
```
void pop_back(){
    if (!debut) return
    if (debut == fin){
        delete debut;
    if (debut==fin){
        delete debut;
        debut = fin = nullptr;
    }

    Noeud* prev = debut;
    Noeud* cur = debut→next;

    while (cur→next){
        prev = cur;
        cur = cur→next;
    }
    delete cur;
    fin = p;
    p→next = nullptr;
}
```

O(n)



- Pour un pop_back en O(1), liste doublement chaînée

```
struct Noeud{
    T elt;
    Noeud* next;
    Noeud* prev;
}
```

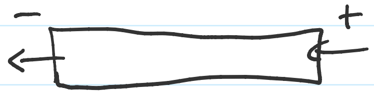


```
//Todo: ajuster push_back, push_front, autres

void pop_back(){
    if (!debut) return
    if (debut==fin){ ... }
    if (!debut) return
    if (debut == fin){ ... }

    Noeud* prev = fin->prev;
    prev->next = nullptr;
    delete fin;
    fin = prev;
}
```

O(1)

Implémentations:

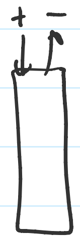
file:



pop_front
push_back

liste simplement chaînée

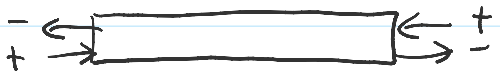
pile:



push_front
pop_front

liste simplement chaînée

deque:



liste doublement chaînée

liste: · même chose
· insérer/supprimer au milieu
↳ itérateurs

[illegible]

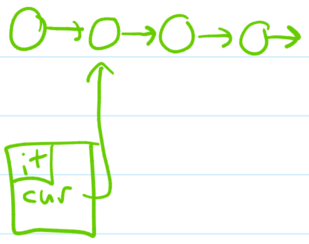
↳ itérateurs

- accès à l'élément i

```
T& operator[](size_t i){          //pas dans SL

    size_t cpt=0;
    Noeud* cur= debut;
    while (cpt < i){
        cur =cur→next;
        cpt++;
    }
    return cur→elt;
}
```

$\Theta(n)$

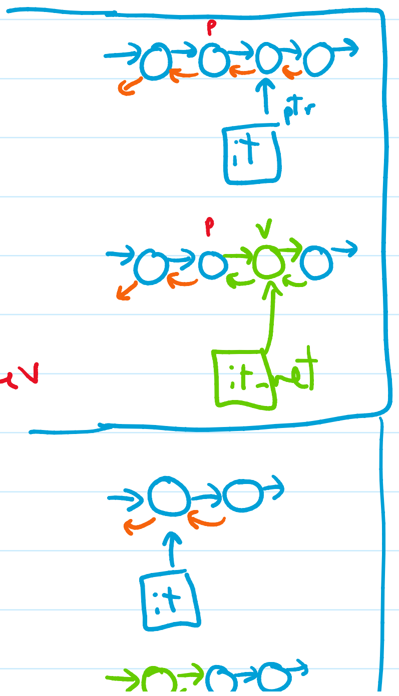
- Itération:

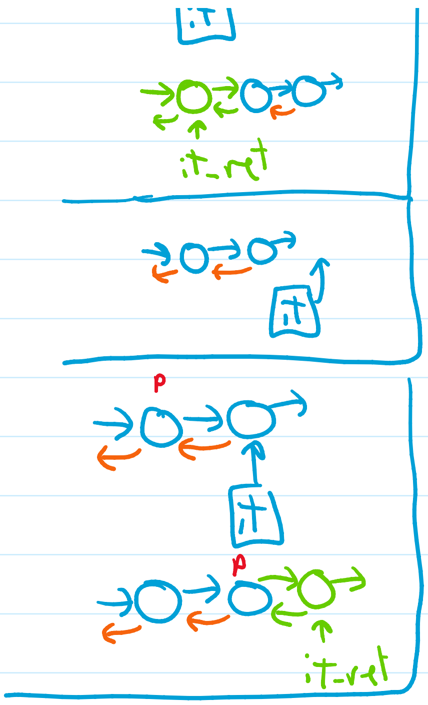


```
class list::iterator{
private:
        Noeud* cur;
public:
        operator++(){
          cur =cur→next;
        }
        operator--(){
          cur = cur→prev;
        }
        operator*(){
          return cur→elt;
        }
        operator==(src){
```

```
        operator==(src){
            return src.cur==cur;
        }
}


iterator list::begin(){
    iterator it;
    it.cur=debut;
    return it;
}

iterator list::end(){
    iterator it;
    it.cur=nullptr;
    return it;
}


void insert(iterator& it, T& val){
    //on suppose une liste doublement chaînée
    if (!it.ptr){ push_back(val); return; }
    Noeud* v=new Noeud();
    v->elt = val;


    Noeud* prev=it.ptr->prev;

    if (prev) prev->next=v;

    v->prev=prev;
    v->next=it.ptr;
```

Tous les cas





```
v → next = it.ptr;
it.ptr → prev = v;
if (!prev) debut = v;
```

$O(1)$

```
iterator it_ret;
it_ret.ptr = v;
return it_ret;
}
```

• Version qui délègue à push_back

```
iterator insert(iterator& it, TYPE& val){

    if(!it.ptr){
        push_back(val);
        return iterator(fin);
    }
    Noeud* oldfin = fin;
    Noeud* next = it.ptr → next;
    it.ptr → next = nullptr;
    fin = it.ptr;
    push_back(val);

    iterator it_ret;
    it_ret.ptr = fin;
    fin → next = next;
    next → prev = fin;
    fin = old_fin;
```

$O(1)$

```
        next→prev = fin;
        fin = old_fin;
        return it_ret;
    }


    TYPE& operator[](size_t i){

        Noeud* v = debut;
        size_t cpt = 0;

        while (cpt != i){
            v = v→next;
            ++cpt;
        }
        return v→elt;
    }
```

$O(n)$