IFT339
Structures de données

Thème 9 : Structures de recherche

Aïda Ouangraoua
Département d'informatique

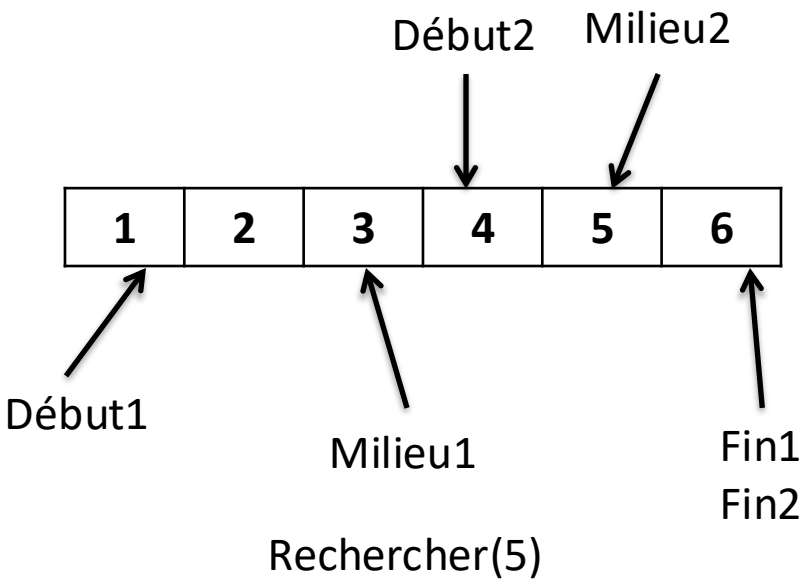
UNIVERSITÉ DE
SHERBROOKE

Skip_list, Arbres binaires de recherche

- Objectif: recherche et insertion dans un conteneur en O(log(n))
- Idée : réduire l'espace de recherche de moitié à chaque étape
- Inspiration : avec Array, si le conteneur est trié, on peut faire une recherche dichotomique en O(log(n)), en localisant le milieu du conteneur en O(1)
  - Possible de localiser le milieu en O(1) car les éléments sont contigus
  - Comment faire dans le cas où les éléments sont non contigus ? (ex. List)

Recherche dichotomique

- Array (trié) : O(log(n))



- Comment faire de même avec une List == > Skip_list

```
Algo B : Entier x Tableau de n entiers → Index (Entier)
    Entree : x, tab
    pas_trouve ← Vrai
    debut ← 0
    fin ← n-1
    milieu ← (debut+fin)/2
    Tant que pas_trouve == Vrai
        Si tab[milieu] == x
            position ← milieu
            pas_trouve ← Faux
        Sinon Si x < tab[milieu]
            fin ← milieu -1
        Sinon
            debut ← milieu +1
        milieu ← (debut+fin)/2
```

Skip_list

- Série de List $L_0, L_1, L_2, \ldots, L_k$ telles que $L_k \subset L_{k-1} \subset \ldots \subset L_0$

L_3 Début				–				9			–					Fin
L_2 Début		–		5		–		9	–	11			–			Fin
L_1 Début	–	1	–	5	–	7	–	9	–	11	–	15	–			Fin
L_0 Début	–	0	–	1	–	2	–	5	–	6	–	7	–	8	–	9	–	10	–	11	–	12	–	15	–	16	–	Fin

- Exemple : Recherche(16) en O(log(n))

L_3 Début ➔ 9

L_2 9 ➔ 11

L_1 11 ➔ 15

L_0 15 ➔ 16

Skip_list

- Série de List L_0, L_1, L_2, …, L_k telles que L_k C L_{k-1} C … C L_0

L_3 Début				–				9			–					Fin
L_2 Début		–		5		–		9	–	11			–			Fin
L_1 Début	–	1	–	5	–	7	–	9	–	11	–	15	–			Fin
L_0 Début	–	0	–	1	–	2	–	5	–	6	–	7	–	8	–	9	–	10	–	11	–	12	–	15	–	16	–	Fin

- Exemple : Recherche(x) en O(log(n))

  À partir de la position i = Début de L_k

  Répéter:

  Si i.suivant[k] <= x

  i = i.suivant[k] // avancer

  Sinon

  k = k-1 // descendre d'un niveau

Skip_list

- Dans le cas idéal, chaque L_i contient $n/2^i$

L_3 Début	–				–				9				–			Fin
L_2 Début	–		5		–		9		–	11			–			Fin
L_1 Début	–	1	–	5	–	7	–	9	–	11	–	15	–	Fin		
L_0 Début	–	0	–	1	–	2	–	5	–	6	–	7	–	8	–	9
	–	10	–	11	–	12	–	15	–	16	–	Fin				

- Hauteur de la Skip_list : nombre de List

- Hauteur d'un élément h(x): nombre de List le contenant

- Au moment de l'ajout d'un élément x, il faut déterminer sa hauteur
  - On utilise un algorithme probabiliste tel que:
    Prob(h(x) = 1) = ½ ; Prob(h(x) = 2) = 1/4 ; ….; Prob(h(x) = i) = $1/2^i$;

Skip_list

- Dans le cas idéal, chaque L_i contient $n/2^i$

L_3 Début				–				9			–					Fin
L_2 Début		–		5		–		9	–	11			–			Fin
L_1 Début	–	1	–	5	–	7	–	9	–	11	–	15	–	Fin		
L_0 Début	–	0	–	1	–	2	–	5	–	6	–	7	–	8	–	9
	–	10	–	11	–	12	–	15	–	16	–	Fin				

- Au moment de l'ajout d'un élément x, il faut déterminer sa hauteur
  - On utilise un algorithme probabiliste tel que:

    Prob(h(x) = 1) = ½ ; Prob(h(x) = 2) = 1/4 ; ….; Prob(h(x) = i) = $1/2^i$;

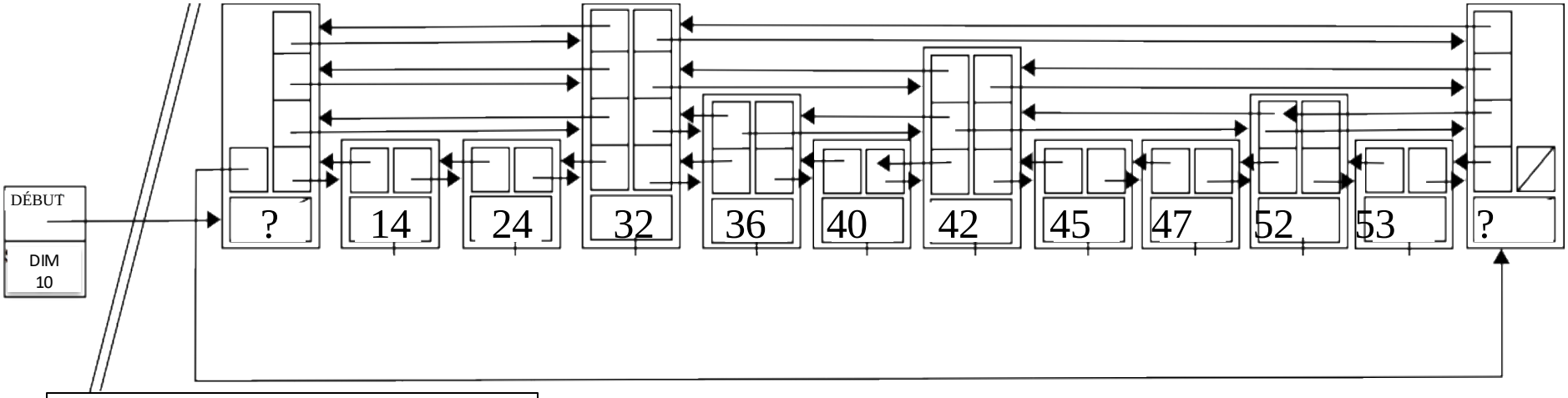
    Tirer au hasard 0 ou 1 tant que l'on tire 0

    h(x) = nombre de tirages

    0 ➔ h(x) = 1 ; 0 1 ➔ h(x) = 2 ; 0 0 1 ➔ h(x) = 3 ; …

Skip_list

- Représentation




```
class cell{
 private:
  type val;
  size_t hauteur
  cell** prec;
  cell** suiv;
 public:
   cell(type&, size_t h);
   cell(type&);}
```

```
class skip_list{
 private:
  cell *debut;
  size_t dim;
 public:
  …
}
```

Skip_list : Prototypes des opérateurs

❑ **Constructeurs**

skip_list : Ø → skip_list& // par défaut
skip_list : skip_list& → skip_list& // par copie

❑ **Destructeur**

~skip_list : Ø → Ø // fait appel à la fonction clear()

❑ **Affectateur**

operator= : skip_list& → Ø // copie le paramètre dans l'objet appelant

Skip_list : Prototypes des opérateurs

❑ **Modificateurs**

swap : skip_list& → Ø // échange le paramètre avec l'objet appelant

insert: cell*, type& → cell* //ajoute un élément à une position (privée)

insert: type& → iterator //ajoute un élément

erase: cell* → cell* //retire un élément à une position (privée)

erase: type& → iterator //retire un élément

❑ **Recherche/ localisation**

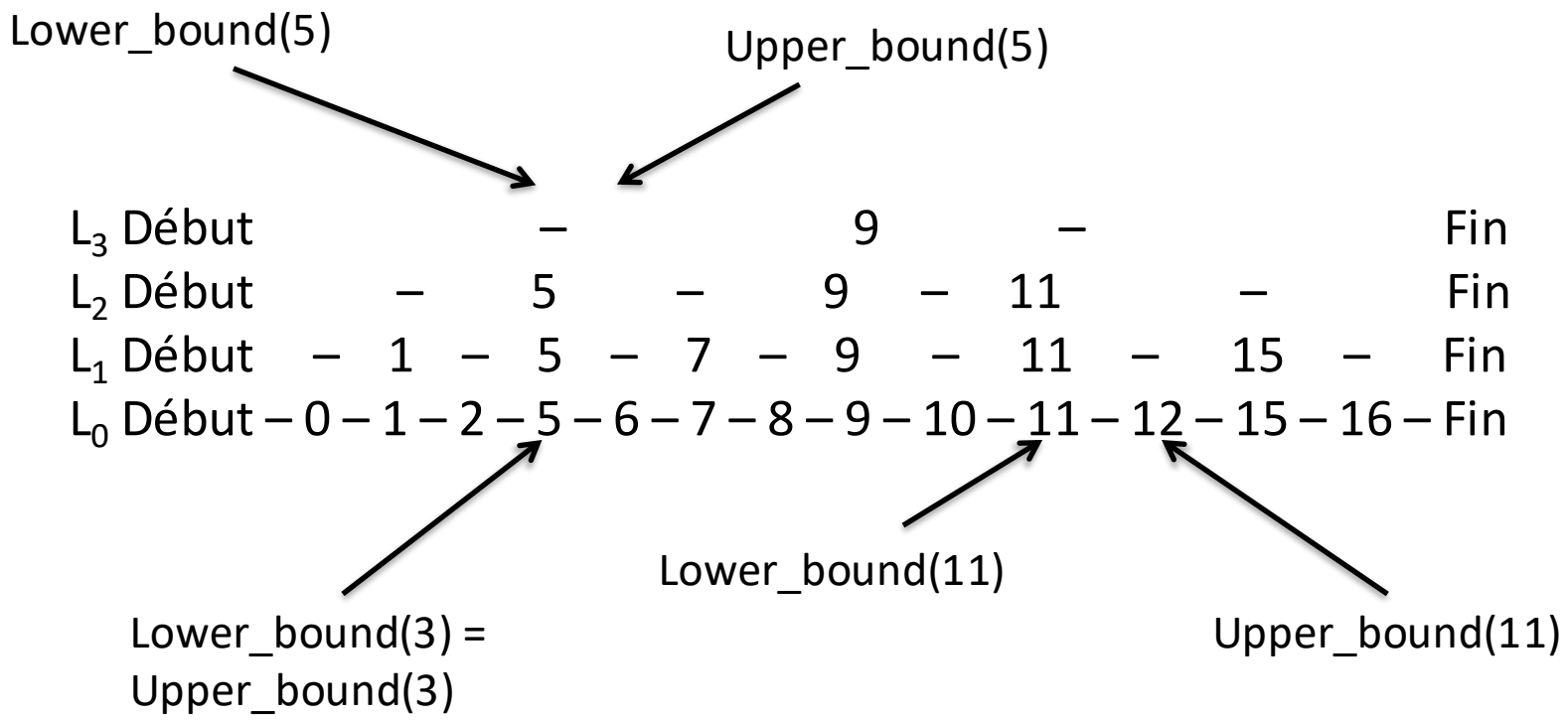
lower_bound: type& → cell* //premier élément >= x (privée)

upper_bound: type& → cell* //premier élément > x (privée)

search: type& → cell* (privée)

search: type& → iterator

Skip_list : lower_bound et upper_bound



Skip_list : lower_bound et upper_bound

- cell* lower_bound(type x)

```
cell* c  = debut;
size_t k = debut->suiv.size()
for(size_t i = k-1 ; i >= 0; i--) {
     while(c->suiv[i] ->val < x)          // <= x (pour upper_bound)
          c = c->suiv[i];
}
 c = c->suiv[0];
 return c;
```

Skip_list : insert

- cell* insert(cell* c, type x)

```
size_t h = calculer_aleatoirement_hauteur();
cell* n = new cell(x,h);
size_t k = debut->suiv.size()
if(h > k)
        augmenter_hauteur(h);  //ajouter les List L_k, L_{k+1}, …, L_{h(x)-1}
//ajouter n entre c->prec[0] et c
```

- cell* insert(type x)

```
cell* c = lower_bound(x)
 return insert(c,x);
```

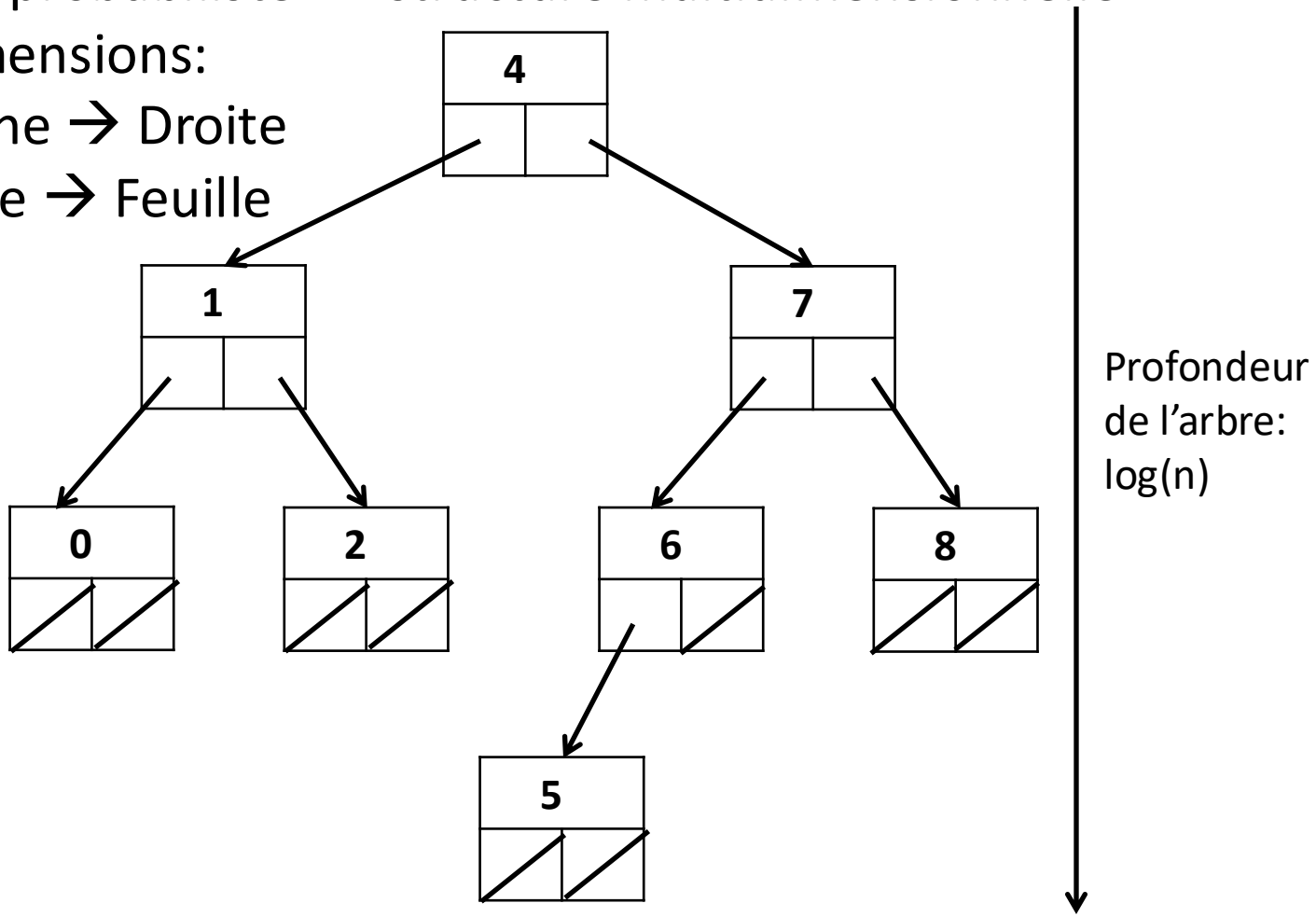
Skip_list : search

- cell* search(type x)

```
cell* c = lower_bound(x)
if(c->val == x)
    return c;
else
    return nullptr;
```

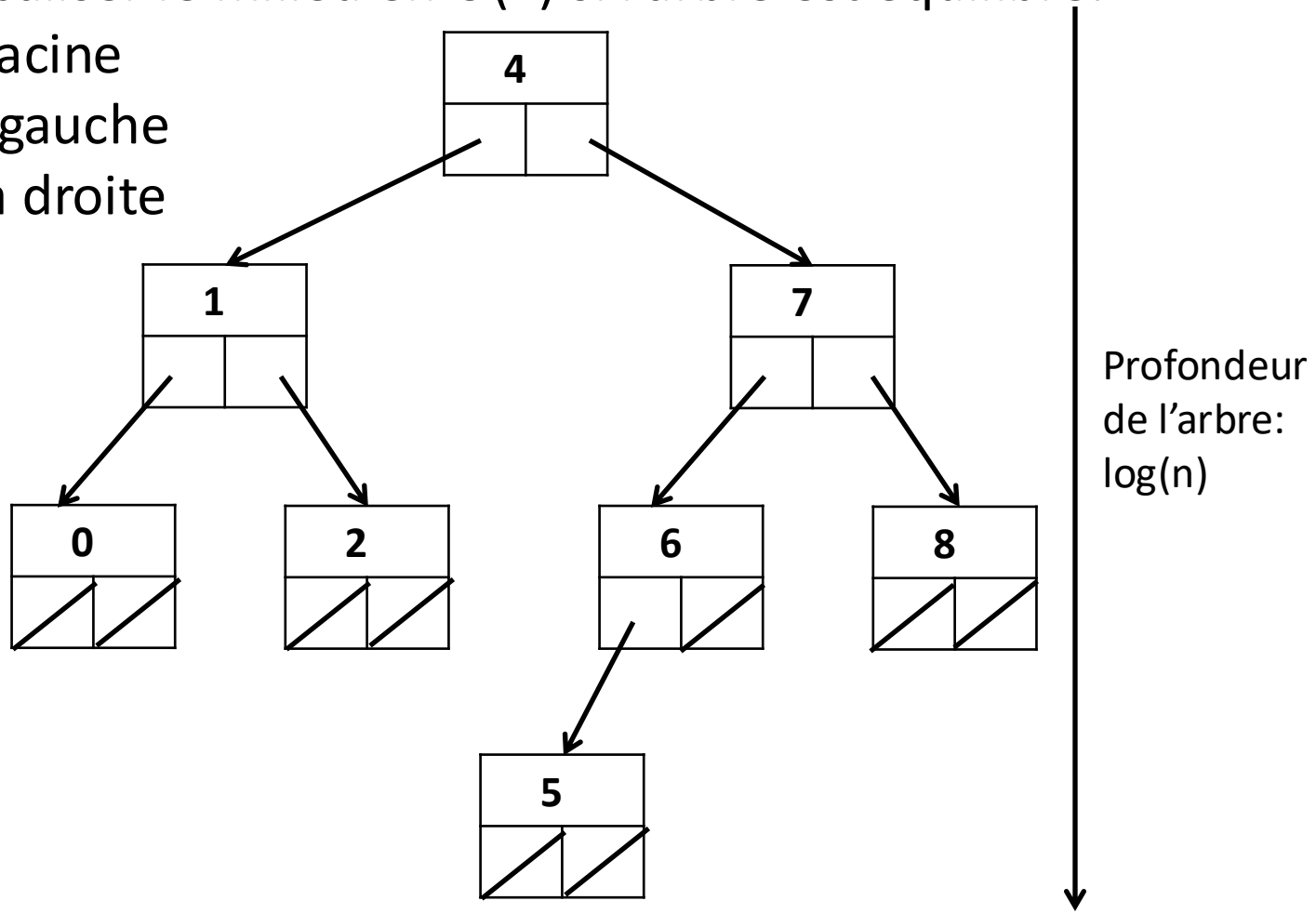
Arbre binaire de recherche

- Skip_list : opération en O(log(n)) (structure linéaire, solution probabiliste)
- Solution non probabiliste → structure multidimensionnelle
  - Deux dimensions:
    - Gauche → Droite
    - Racine → Feuille



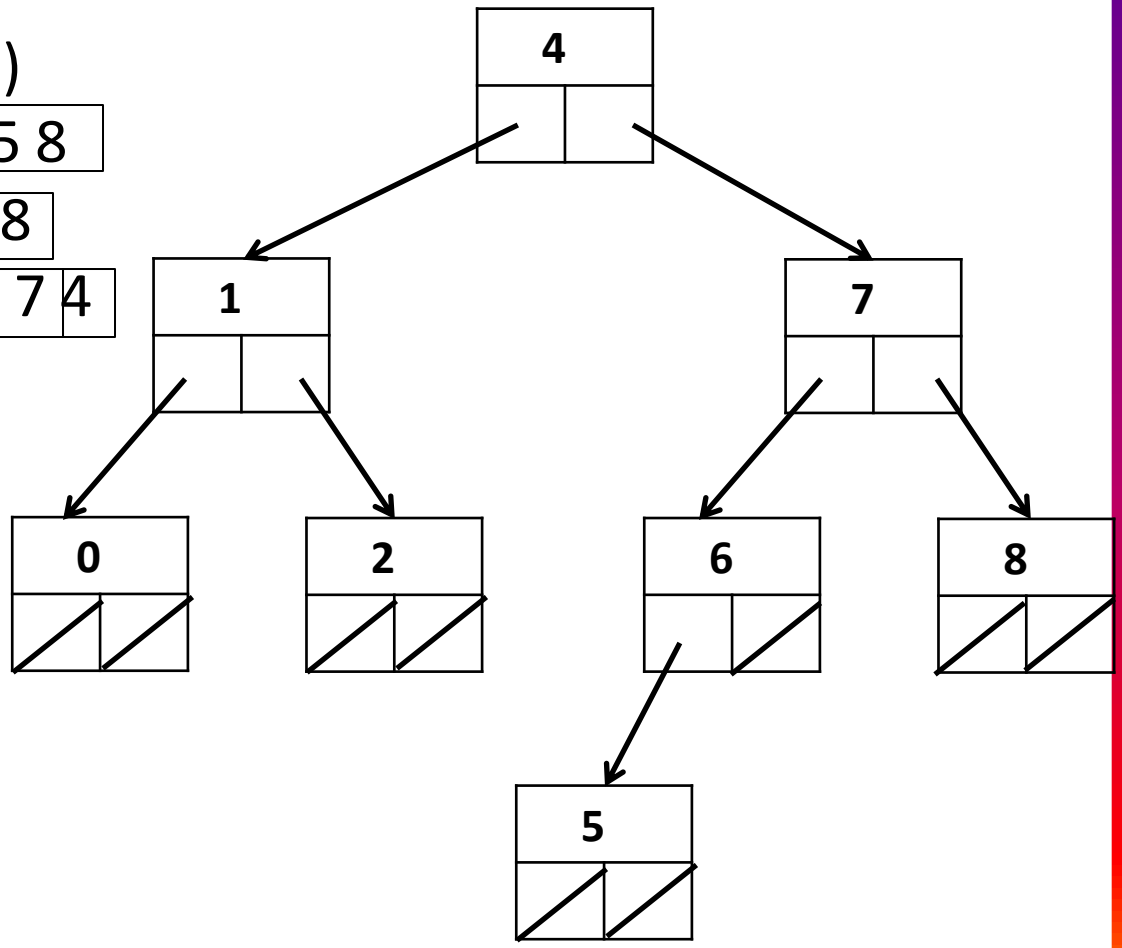
Arbre binaire de recherche

- ABR : tel que chaque nœud a au plus deux enfants: un enfant gauche et/ou un enfant droit
- Permet de localiser le milieu en O(1) si l'arbre est équilibré:
  - milieu à la racine
  - plus petit à gauche
  - plus grand à droite



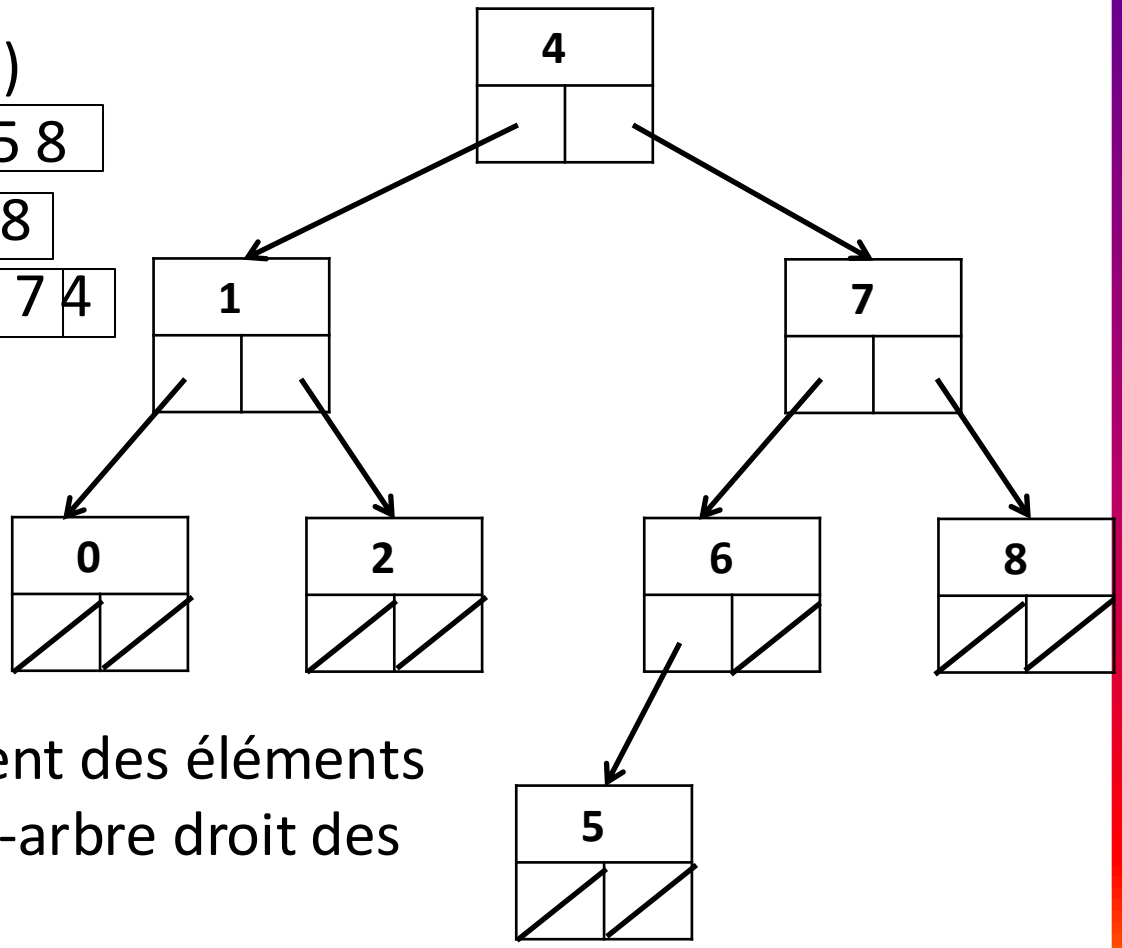
Arbre binaire de recherche

- ABR : tel que chaque nœud a au plus deux enfants: un enfant gauche et/ou un enfant droit
- Plusieurs types de parcours possible:
  - En largeur (horizontal)
  - En profondeur (vertical)
    - Prefixe: 4 1 0 2 7 6 5 8
    - Infixe: 0 1 2 4 5 6 7 8
    - Postfixe: 0 2 1 5 6 8 7 4



Arbre binaire de recherche

- ABR : tel que chaque nœud au plus deux enfants: un enfant gauche et/ou un enfant droit
- Plusieurs types de parcours possible:
  - En largeur (horizontal)
  - En profondeur (vertical)
    - Prefixe: 4 1 0 2 7 6 5 8
    - Infixe: 0 1 2 4 5 6 7 8
    - Postfixe: 0 2 1 5 6 8 7 4



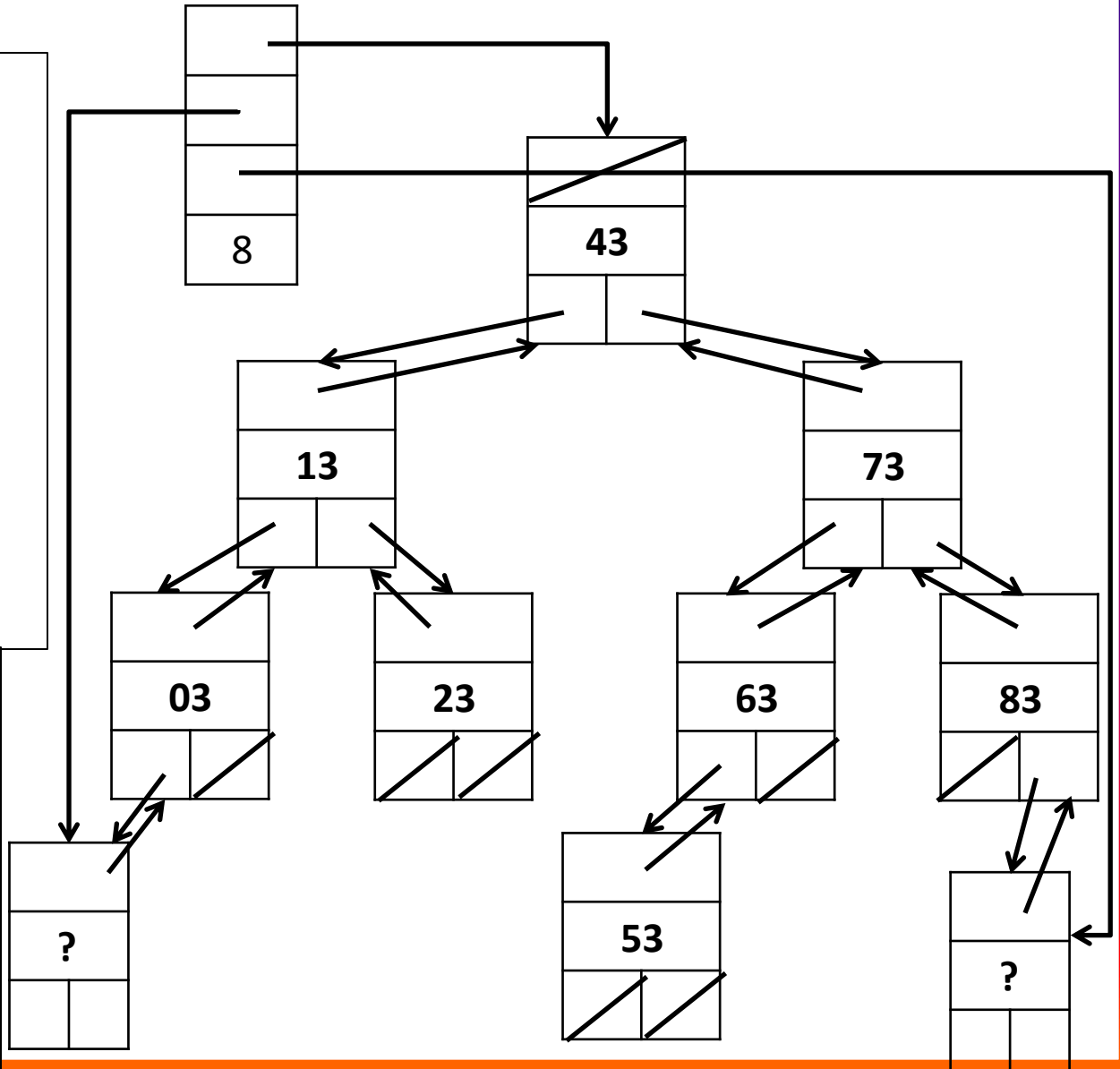
- Dans un ABR, l'ordre infixe, correspond à l'ordre des éléments, i.e. :
  pour tout nœud x,
  le sous-arbre gauche contient des éléments plus petits que x, et le sous-arbre droit des éléments plus grands.

Arbre binaire de recherche

- Représentation

```
class noeud{
 private:
  type val;
  nœud* parent;
  nœud* gauche;
  nœud* droit;
 public:
   noeud(type&);
   noeud(type&, nœud*,
         nœud*, nœud* );}
```

```
class abr{
 private:
  noeud *racine, debut,
fin;
  size_t dim;
 public:
  …
```



ABR : Prototypes des opérateurs

❑ **Constructeurs**

abr : Ø → abr& // par défaut
abr : abr& → abr& // par copie

❑ **Destructeur**

~abr : Ø → Ø // fait appel à la fonction clear()

❑ **Affectateur**

operator= : abr& → Ø // copie le paramètre dans l'objet appelant

ABR : Prototypes des opérateurs

❑ **Modificateurs**

swap : abr& → Ø // échange le paramètre avec l'objet appelant
insert: noeud*, type& → noeud* //ajoute un élément à une position (privée)
insert: type& → noeud* //ajoute un élément (privée)
insert: type& → iterator //ajoute un élément
erase: noeud*→ noeud* //retire un élément à une position (privée)
erase: : type& → noeud* //retire un élément (privée)
erase: type& → iterator //retire un élément

❑ **Recherche/ localisation**

lower_bound: type& → noeud* //premier élément >= x (privée)
upper_bound: type& → noeud* //premier élément > x (privée)
search: type& → nœud* (privée)
search: nœud*, type& → nœud* (privée)
search: type& → iterator
previous: noeud*→ noeud* * (privée)
next: noeud*→ noeud* * (privée)

ABR : search

- noeud* search (nœud* n, type x)

```
nœud* c;
if( n == nullptr or n->val == x)
    c = n;
elif(n->val < x)
    c = search (n->droit ,x);
else
    c = search (n->gauche ,x);
return c;
```

ABR : search

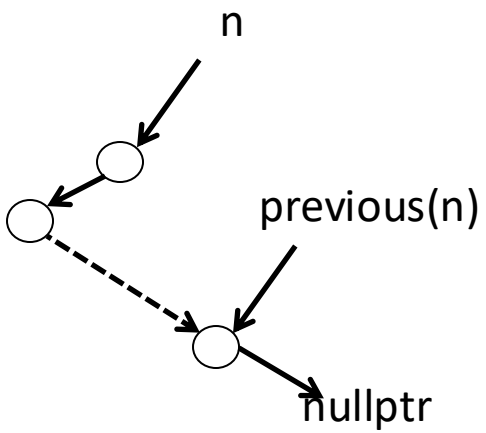
- noeud* search (type x)

```
return search(racine,x);
```

- noeud* search (type x)

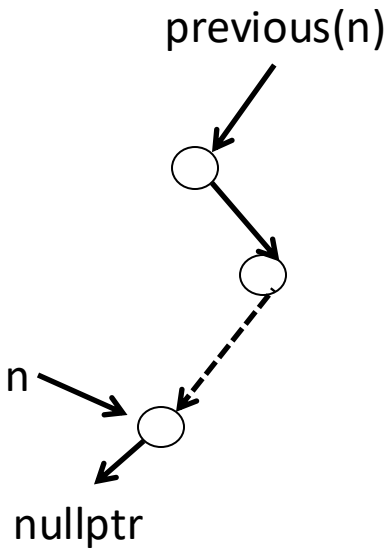
```
nœud* n = racine;
while(n != nullptr and n->val != x){
        if(n->val < x)
            n = n->droit;
        else
            n = n->gauche;
 return n;
```

ABR : previous, next



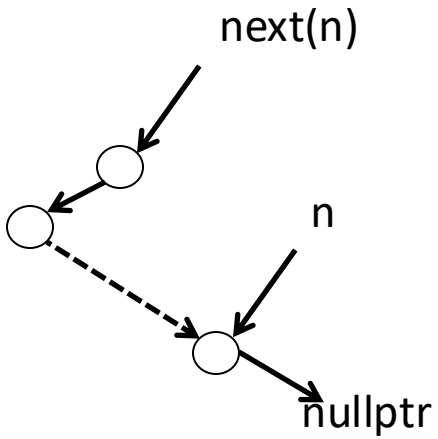
- noeud* previous (nœud* n)

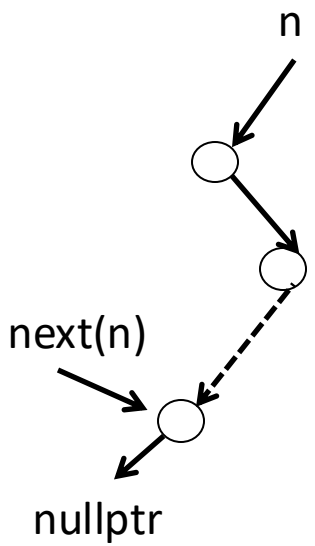
```
    nœud* p;
    if(n->gauche != nullptr){  //cas (1) : a un enfant gauche; on doit descendre
        p = n->gauche;
        while(p->droit != nullptr)
            p = p->droit;
    } else{    //cas(2) : pas d'enfant gauche; on doit remonter
        prevp = n; p = n->parent;
        while(p != nullptr and p->droit != prevp){
            prevp = p;
            p = p->parent;
        }
    }
    return p;
```



ABR : previous, next

- noeud* next (nœud* n)
  //symétrique de previous
  //en inversant gauche et droit





ABR : lower_bound,upper_bound

- noeud* lower_bound (type x) //dernier nœud où l'on descend à gauche

```
nœud* n = racine;
nœud* lower = fin;
while(n != nullptr){
        if(n->val >= x){
            lower = n;
            n = n->gauche;}
         else
            n = n->droit;
}
return lower;
```

ABR : lower_bound,upper_bound

- noeud* upper_bound (type x) //dernier nœud où l'on descend à gauche

```
nœud* n = racine;
nœud* upper = fin;
while(n != nullptr){
      if(n->val > x){
           upper = n;
           n = n->gauche;}
       else
           n = n->droit;
}
 return upper;
```

ABR : insert

- noeud* insert (nœud* n, type x)

```
nœud* nouveau = new nœud(x);
if (n->gauche == nullptr){// cas(1): pas d'enfant gauche
        n->gauche  = nouveau;
        nouveau->parent = n;
}
else{// cas(2): a un d'enfant gauche
        nœud* prev = previous(n);
        prev->droit  = nouveau;
        nouveau->parent = prev;
}
return nouveau;
```

On insère toujours une feuille

ABR : insert

- noeud* insert (type x)

```
nœud* n = lower_bound(x);
return insert(n,x);
```

On insère toujours une feuille

ABR : erase

- noeud* erase (nœud* n)

```
nœud* c = next(n);
if(n->gauche == nullptr and n->droit == nullptr)//aucun enfant
    //supprimer n de l'arbre
elif(n->gauche != nullptr)//un enfant gauche
    //remplacer la valeur de n par celle de previous(n)
    // supprimer previous(n) de l'arbre
else//un enfant droit
    //remplacer la valeur de n par celle de next(n)
    // supprimer next(n) de l'arbre
    c = n;
return c;
```

On supprime toujours une feuille

ABR : erase

- noeud* erase (type x)

```
nœud* n = search(x);
if(n != nullptr)
        return erase(n);
return n;
```

On supprime toujours une feuille