INF3105 – Arbres binaires de recherches

Jaël Champagne Gareau

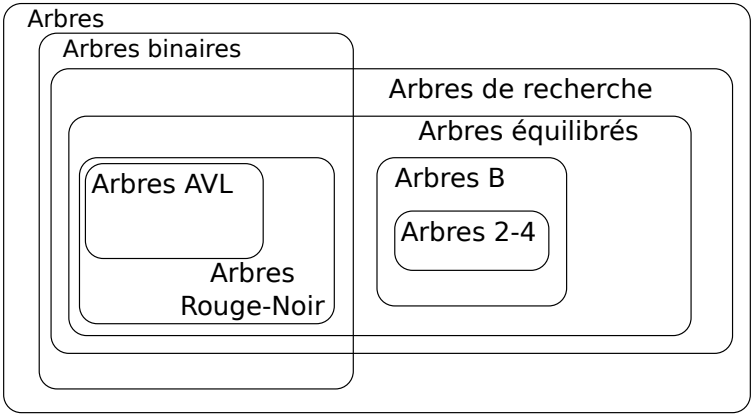
Université du Québec à Montréal (UQAM)

Été 2024

http://cria2.uqam.ca/INF3105/

UQÀM

Arbres étudiés en INF3105



Arbre binaire

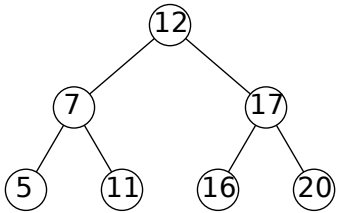
- Cas particulier d'un arbre.
- Chaque noeud a au plus deux enfants.
- Parfois, on nomme les enfants : gauche et droite.

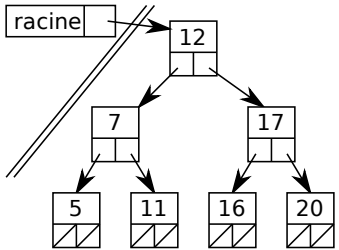
Propriétés

- Nombre minimal de noeuds : $h+1$.
- Nombre maximal de noeuds : $2^{h+1}-1$.

Représentation d'un arbre binaire

```
1  template <class T> class ArbreBinaire {
2    public:
3      ArbreBinaire();
4      ~ArbreBinaire();
5      void vider();
6    private:
7      struct Noeud {
8          T contenu;
9          Noeud* gauche;
10         Noeud* droite;
11     };
12     Noeud* racine;
13     vider(Noeud*);
14 };
```





Constructeur / Destructeur

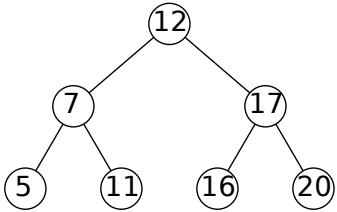
```
1  template <class T>
2  ArbreBinaire::ArbreBinaire()
3      : racine(nullptr) {}
4
5  template <class T>
6  ArbreBinaire::~ArbreBinaire() {
7      vider(); // sera fait plus tard
8  }
```

Arbre de recherche

- Cas particulier d'arbre.
- Relation d'ordre : les enfants d'un parents sont ordonnés afin de faciliter la recherche.
- But : recherche efficace des nœuds.
- Utile pour représenter et stocker des ensembles.
- L'ordre d'insertion des éléments n'affecte pas le contenu abstrait de l'arbre.

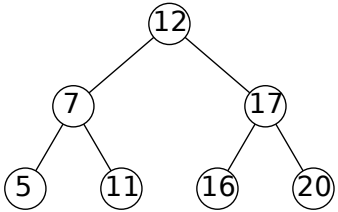
Arbre binaire de recherche (ABR)

- Cas particulier d'arbre binaire.
- Cas particulier d'arbre de recherche.
- Noms des enfants : gauche et droite.
- Nécessite une relation d'ordre.
- Organisation des nœuds :
  - *gauche* $<$ *parent*.
  - *parent* $<$ *droite*.
  - Évidemment : *gauche* $<$ *droite*.
- Généralisable à $\leq$ au lieu de $<$.
- L'arbre ci-droit représente l'ensemble {5, 7, 11, 12, 16, 17, 20}.



Parcours en inordre

```
1  template <class T>
2  void ArbreBinaire<T>::afficher_inordre() {
3      afficher_inordre(racine);
4  }
5
6  template <class T>
7  void ArbreBinaire<T>::afficher_inordre(Noeud* n) {
8      if(n == nullptr) return;
9      afficher_inordre(n->gauche);
10     std::cout << n->contenu << " "; // traitement
11     afficher_inordre(n->droite);
12 }
```



- Un parcours en inordre visite les éléments contenus dans les nœuds de l'arbre dans leur ordre naturel.

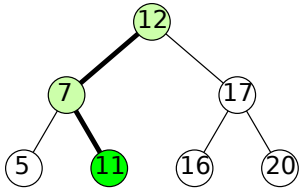
Classe ArbreBinRech

```
1  template <class T>
2  class ArbreBinRech : public ArbreBinaire {
3    public:
4      bool contient(const T& element) const;
5      const T* rechercher(const T& element) const;
6      void inserer(const T& element);
7      void enlever(const T& element);
8    private:
9      const T* rechercher(Noeud* n, const T& element) const;
10     void inserer(Noeud*& n, const T& element);
11     void enlever(Noeud*& n, const T& element);
12 };
```

- Le type T requiert un opérateur <.

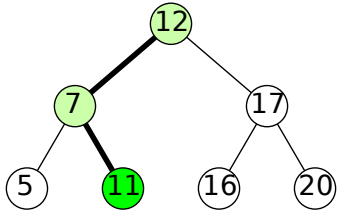
Recherche

```
1  template <class T>
2  bool ArbreBinRech<T>::contient(const T& element) const {
3      return rechercher(racine, element) != nullptr;
4  }
5
6  template <class T>
7  const T* ArbreBinRech<T>::rechercher(Noeud* n, const T& e) const {
8      if(n == nullptr)
9          return nullptr;
10     else if(e == n->contenu)
11         return &(n->contenu);
12     else if(e < n->contenu)
13         return rechercher(n->gauche, e);
14     else // e > n->contenu
15         return rechercher(n->droite, e);
16 }
```



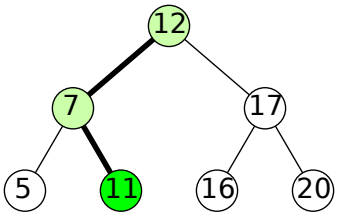
Exercice 1 : Recherche sans opérateurs == et >

```
1  template <class T>
2  bool ArbreBinRech<T>::contient(const T& element) const {
3      return rechercher(racine, element) != nullptr;
4  }
5
6  template <class T>
7  const T* ArbreBinRech<T>::rechercher(Noeud* n, const T& e) const {
8
9
10
11
12
13
14
15 }
```



Exercice 1 : Recherche sans opérateurs == et >

```
1  template <class T>
2  bool ArbreBinRech<T>::contient(const T& element) const {
3      return recherche(racine, element) != nullptr;
4  }
5
6  template <class T>
7  const T* ArbreBinRech<T>::rechercher(Noeud*n, const T& e) const {
8      if(n == nullptr)
9          return nullptr;
10     if(e < n->contenu)
11         return rechercher(n->gauche, e);
12     if(n->contenu < e)
13         return rechercher(n->droite, e);
14     return &(n->contenu); // n->contenu == e
15 }
```



Complexité de la recherche

- Cas moyen ?
- Pire cas ?

Complexité de la recherche

- Cas moyen ?
  - Idéalement, les deux sous-arbres d'un nœud devraient avoir autant de nœuds (arbre équilibré).
  - Ainsi, on espère couper par deux la recherche à chaque itération.
  - Le nombre de fois qu'on peut diviser n par deux est $\log_2 n$.
  - Donc : $O(\log n)$.
- Pire cas ?
  - Si l'arbre n'est pas équilibré, la recherche n'est pas coupée par deux à chaque itération.
  - Le pire arbre a une hauteur $h = n - 1$
  - Donc, la complexité dans le pire cas est $O(h) = O(n)$.

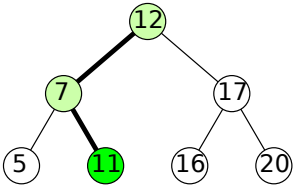
Exercice 2 : Version non récursive

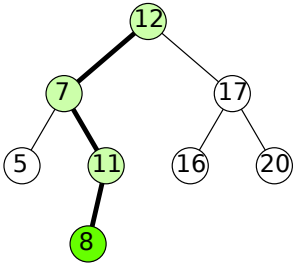
```
1  template <class T>
2  bool ArbreBinRech<T>::contient(const T& element) const {
3      return rechercher(element) != nullptr;
4  }
5
6  template <class T>
7  const T* ArbreBinRech<T>::rechercher(const T& element) const {
8
9
10
11
12
13
14
15
16 }
```

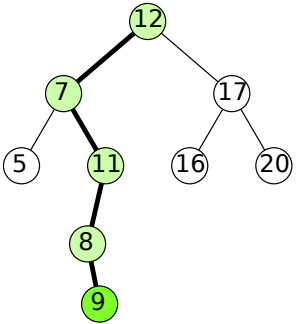
Exercice 2 : Version non récursive

```
1  template <class T>
2  bool ArbreBinRech<T>::contient(const T& element) const {
3      return rechercher(element) != nullptr;
4  }
5
6  template <class T>
7  const T* ArbreBinRech<T>::rechercher(const T& element) const {
8      Noeud* n = racine;
9      while(n != nullptr) {
10       if(element < n->contenu) n = n->gauche;
11       else if(n->contenu < element) n = n->droite;
12       else return &n->contenu;
13     }
14     return nullptr;
15 }
```

Insertion





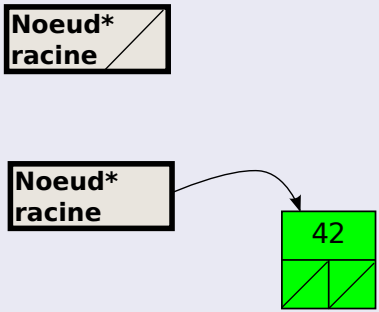


Ébauche code insertion

Ébauche 1

```
1  template <class T>
2  void ArbreBinRech<T>::inserer(const T& e) {
3      // cas de la racine (arbre vide)
4      if(racine == nullptr)
5          racine = new Noeud(e);
6      else {
7          // ...
8      }
9  }
```

Exemple : arbre.insert(42)

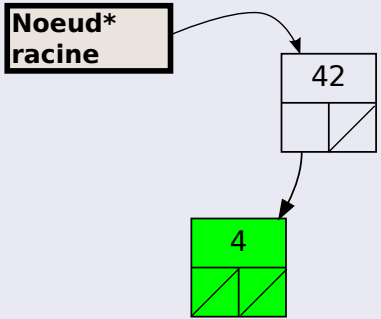


Ébauche code insertion

Ébauche 2

```
1  template <class T> // fonction public
2  void ArbreBinRech<T>::inserer(const T& e) {
3      if(racine == nullptr) racine = new Noeud(e);
4      else inserer(racine, e);
5  }
6
7  template <class T> // fonction private
8  void ArbreBinRech<T>::inserer(Noeud* n, const T& e) {
9      if(e == n->contenu) n->contenu = e;
10     else if(e < n->contenu) {
11         if(n->gauche == nullptr) n->gauche = new Noeud(e);
12         else inserer(n->gauche, e);
13     } else {
14         if(n->droite == nullptr) n->droite = new Noeud(e);
15         else inserer(n->droite, e);
16     }
17 }
```

Exemple : arbre.insert(4)



Ébauche code insertion

Ébauche 2

```
1  template <class T> // fonction public
2  void ArbreBinRech<T>::inserer(const T& e) {
3     if(racine == nullptr) racine = new Noeud(e);
4     else inserer(racine, e);
5  }
6
7  template <class T> // fonction private
8  void ArbreBinRech<T>::inserer(Noeud* n, const T& e) {
9     if(e == n->contenu) n->contenu = e;
10    else if(e < n->contenu) {
11       if(n->gauche == nullptr) n->gauche = new Noeud(e);
12       else inserer(n->gauche, e);
13    } else {
14       if(n->droite == nullptr) n->droite = new Noeud(e);
15       else inserer(n->droite, e);
16    }
17 }
```

Ébauche 3 (code unifié)

```
1  template <class T> // fonction public
2  void ArbreBinRech<T>::inserer(const T& e) {
3     inserer(racine, e);
4  }
5
6  template <class T>  // fonction private
7  void ArbreBinRech<T>::inserer(Noeud*& n, const T& e) {
8     if(n == nullptr)
9        n = new Noeud(e);
10    else if(e == n->contenu)
11       n->contenu = e;
12    else if(e<n->contenu)
13       inserer(n->gauche, e);
14    else
15       inserer(n->droite, e);
16 }
```

Complexité de l'insertion

- Cas moyen ? $O(\log n)$
- Pire cas ? $O(n)$

Enlèvement

- Repérer le nœud (comme une recherche).
- Cas 1 : Si une feuille : simplement enlever le nœud.
- Cas 2 : Si un seul enfant : reconnecter l'enfant avec le parent.
- Cas 3 : Si deux enfants : emprunter le max à gauche ou le min à droit qui sont garantis d'être soit une feuille (cas 1) ou un noeud avec au plus un enfant (cas 2).

Complexité de l'enlèvement

- Cas moyen ? $O(\log n)$
- Pire cas ? $O(n)$

Exercice 1 : Vider

```
1  template <class T>
2  ArbreBinaire<T>::~ArbreBinaire() {
3      vider();
4  }
5
6  template <class T>
7  void ArbreBinaire<T>::vider() {
8
9
10
11
12
13 }
```

Exercice 1 : Vider

```
1  template <class T>
2  void ArbreBinaire<T>::vider() {
3      vider(racine);
4      racine = nullptr;
5  }
6
7  template <class T>
8  void ArbreBinaire<T>::vider(Noeud* n) {
9      if(n == nullptr) return;
10     vider(n->gauche);
11     vider(n->droite);
12     delete n;
13 }
```

Exercice 2 : trouver les constructeur(s) et opérateur(s) requis

```
1 int main() {
2     ArbreBinRech<int> arbre1;
3     arbre1.inserer(3);
4     arbre1.inserer(2);
5     arbre1.inserer(1);
6     ArbreBinRech<int> arbre2(arbre1);
7     ArbreBinRech<int> arbre3;
8     arbre1 = arbre3;
9 }
```

Exercice 2 : constructeurs et opérateurs requis

```
1 int main() {
2   ArbreBinRech<int> arbre1; // constructeur sans argument
3   arbre1.inserer(3);
4   arbre1.inserer(2);
5   arbre1.inserer(1);
6   ArbreBinRech<int> arbre2(arbre1); // constructeur par copie
7   ArbreBinRech<int> arbre3;
8   arbre1 = arbre3; // operateur d'affectation (operator =)
9 } // destructeur
```

Exercice 2 : Révision de la déclaration

```
1  template <class T>
2  class ArbreBinaire {
3    public:
4      ArbreBinaire();
5      ~ArbreBinaire();
6      ArbreBinaire(const ArbreBinaire&);
7      // ...
8      ArbreBinaire& operator = (const ArbreBinaire&);
9      // ...
10   private:
11     void copier(const Noeud* src, Noeud*& dst);
12     // ...
13 };
```

Exercice 2 : Constructeur par copie

```
1  template <class T>
2  ArbreBinaire<T>::ArbreBinaire(const ArbreBinaire& autre) : racine(nullptr) {
3      copier(autre.racine, racine);
4  }
5
6  template <class T>
7  void ArbreBinaire<T>::copier(const Noeud* src, Noeud*& dst) {
8      if(src == nullptr) return;
9      dst = new Noeud(src->contenu);
10     copier(src->gauche, dst->gauche);
11     copier(src->droite, dst->droite);
12 }
```

Exercice 2 : Operateur d'affectation =

```
1  template <class T>
2  ArbreBinaire& ArbreBinaire::operator=(const ArbreBinaire& autre) {
3      if(this == &autre) return *this;
4      vider();
5      copier(autre.racine, racine);
6      return *this;
7  }
8
9  template <class T>
10 void ArbreBinaire<T>::copier(const Noeud* src, Noeud*& dst) const {
11     if(src == nullptr) return;
12     dst = new Noeud(src->contenu);
13     copier(src->gauche, dst->gauche);
14     copier(src->droite, dst->droite);
15 }
```

Exercice 3 : Simulez

```
1  void f1(ArbreBinRech<int> a, int i) {
2      a.inserer(i);
3  }
4
5  ArbreBinRech<int> f2(ArbreBinRech<int> a, int i) {
6      a.inserer(i);
7      return a;
8  }
9
10 int main() {
11     ArbreBinRech<int> arbre1;
12     f1(arbre1, 1);
13     ArbreBinRech<int> arbre2;
14     arbre2 = f2(arbre1, 2);
15     ArbreBinRech<int> arbre3 = f2(arbre1, 3);
16 }
```