Développement efficace (R3.02)
Les tas (heap)

Marin Bougeret
LIRMM, IUT/Université de Montpellier

Outline

Contexte

Hypothèse

On considère des éléments e_i de type T qui savent se comparer selon un ordre donné :

- en maths on écrira $e_1 \leq e_2$
- en JAVA on écrira $e_1.compareTo(e_2) \leq 0$

But

Ecrire une structure de données, **le tas/heap** qui sait faire principalement et rapidement ($\mathcal{O}(log(n))$)

- `add()`
- `getTop()`/`removeTop()` obtention/suppression d'un élément maximal pour $\leq$

Essais naïfs

Tentative 1 : maintenir un tableau trié croissant

- `add()`
  - tout re-trier : $\mathcal{O}(n(log(n)))$ trop long !
  - autres idées ?
- `getTop()`/`removeTop()` : on pourrait maintenir un entier `size` qui compte le nombre d'entiers dans la structure , les éléments seraient stockés de `t[0]` à `t[size-1]`

Essais naïfs

Tentative 1 : maintenir un tableau trié croissant

- `add()`
  - tout re-trier : $\mathcal{O}(n(log(n)))$ trop long !
  - autres idées ?
- `getTop()`/`removeTop()` : on pourrait maintenir un entier `size` qui compte le nombre d'entiers dans la structure , les éléments seraient stockés de `t[0]` à `t[size-1]`

Essais naïfs

Tentative 1 : maintenir un tableau trié croissant

- `add()`
  - tout re-trier : $\mathcal{O}(n(log(n)))$ trop long !
  - autres idées ?
- `getTop()`/`removeTop()` : on pourrait maintenir un entier `size` qui compte le nombre d'entiers dans la structure , les éléments seraient stockés de `t[0]` à `t[size-1]`

Tentative 2 : utiliser un arbre de recherche équilibré

- `add()` `getTop()`/`removeTop()` en $\mathcal{O}(h)$ (ainsi que `contains(T e)`), où h est la hauteur de l'arbre.
- Il faudrait s'assurer que h reste en $\mathcal{O}(log(n))$:
  - **les arbres de recherche équilibrés** (balanced search trees) permettent d'obtenir $h \leq c log(n)$ avec c constante
  - avec les tas, on va avoir $h = \lfloor log(n+1) \rfloor$ qui est la meilleure borne atteignable (mais on ne sera pas faire `contains(T e)` rapidement)

Outline

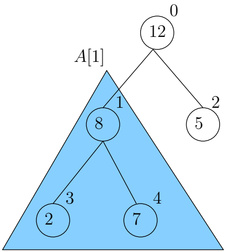
Notations

On considère un arbre binaire enraciné A contenant des éléments de type T dans ses sommets. Pour tout sommet $i \in V(A)$ on note :

- $val(i)$ l'élément contenu dans le sommet i
- $p(i)$ la profondeur de i
- $filsG(i)$ et $filsD(i)$ le numéro du fils gauche et droit de i
- $pere(i)$ numéro du père de i
- $A[i]$ le sous arbre enraciné en i

Exemple avec $T =$ les entiers

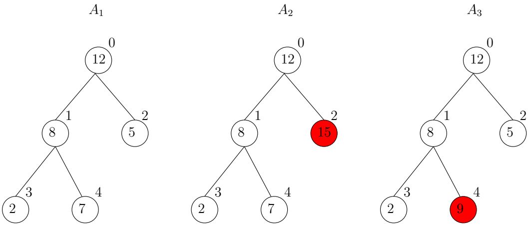
- $val(2) = 5$
- $p(0) = 1$, $p(4) = 2$
- $filsG(1) = 3$
- $pere(4) = 1$



Propriété du maximum

On dit qu'un arbre A a <u>la propriété du maximum</u> ssi pour tout sommet $i \in V(A)$, i est plus grand que tous les éléments sous lui

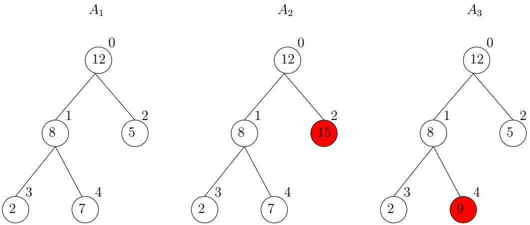
A_1 a la propriété du maximum, A_2 et A_3 ne l'ont pas.



Propriété du maximum

On dit qu'un arbre A a la propriété du maximum ssi pour tout sommet $i \in V(A)$, $val(i) \geq val(j)$ pour tout $j \in V(A[i])$

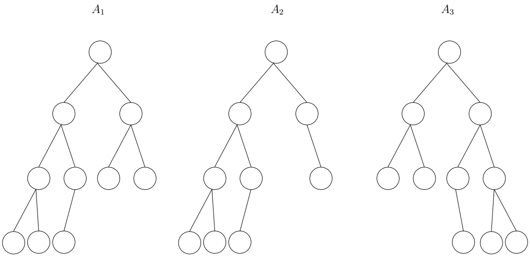
A_1 a la propriété du maximum, A_2 et A_3 ne l'ont pas.



Propriété de semi-complet

On dit qu'un arbre A est semi-complet ssi tous les niveaux sont pleins, sauf peut être le plus profond qui est tassé à gauche

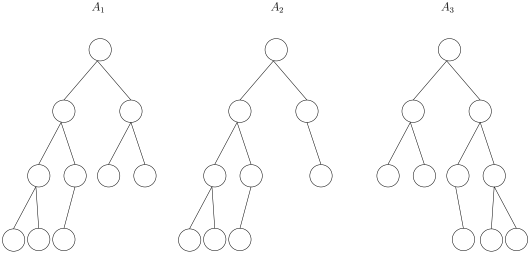
A_1 est semi-complet, A_2 et A_3 ne le sont pas.



Propriété de semi-complet

On dit qu'un arbre A est <u>semi-complet</u> ssi pour tout $p \geq 1$, il y a 2^{p-1} sommets de profondeur p, et .. tassage (on laisse tomber la définition formelle :=)

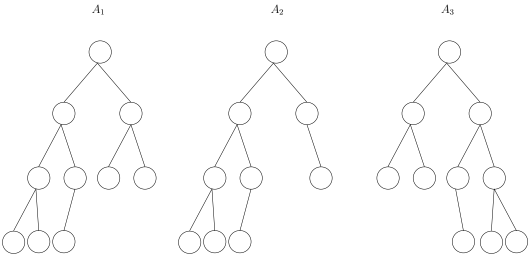
A_1 est semi-complet, A_2 et A_3 ne le sont pas.



Propriété de semi-complet

On dit qu'un arbre A est <u>semi-complet</u> ssi tous les niveaux sont pleins, sauf peut être le plus profond qui est tassé à gauche

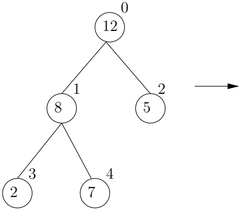
A_1 est semi-complet, A_2 et A_3 ne le sont pas.



Semi-complet : conséquences

Un arbre semi-complet se stocke facilement dans un tableau :

- on numérote les sommets par niveaux, de gauche à droite
- on crée un tableau `t` de grande taille
- l'élément du sommet *i* est stocké dans la case `t[i]`
- on ajoute un entier `size` pour se rappeler de la taille



$size = 5$

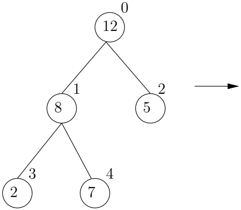
$t =$

12	8	5	2	7					

Semi-complet : conséquences

Un arbre semi-complet se stocke facilement dans un tableau :

- on numérote les sommets par niveaux, de gauche à droite
- on crée un tableau `t` de grande taille
- l'élément du sommet *i* est stocké dans la case `t[i]`
- on ajoute un entier `size` pour se rappeler de la taille



$size = 5$

$t =$

12	8	5	2	7					

En TP, on utilisera une `ArrayList` pour `t`

Semi-complet : conséquences

Hauteur d'un semi-complet

- Soit A semi-complet à n éléments. Alors $h(A) = \lfloor \log(n+1) \rfloor$.
- C'est la plus petite hauteur possible pour un arbre binaire : pour tout arbre binaire B à n éléments, $h(B) \geq \lfloor \log(n+1) \rfloor$

Preuve:

Observation

Un arbre binaire complet de hauteur h contient $c_h = 1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1$ sommets

Semi-complet : conséquences

Hauteur d'un semi-complet

- Soit A semi-complet à n éléments. Alors $h(A) = \lfloor \log(n+1) \rfloor$.
- C'est la plus petite hauteur possible pour un arbre binaire : pour tout arbre binaire B à n éléments, $h(B) \geq \lfloor \log(n+1) \rfloor$

Preuve: Conséquences

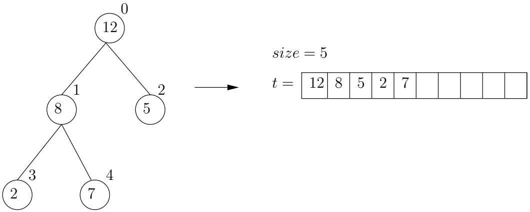
- Soit A semi-complet à n éléments.
- On a $c_{h-1} < n$ (A contient strictement plus que le complet de hauteur $h-1$)
- Donc $2^{h-1} < n+1$, impliquant $h < log(n+1) + 1 \Rightarrow h \leq \lfloor \log(n+1) \rfloor$.
- Si l'on a un arbre binaire B de hauteur h qui contient n éléments, alors $n \leq 2^h - 1$
- donc $\log(n+1) \leq h$, et $\lfloor \log(n+1) \rfloor \leq h$

Semi-complet : conséquences

Propriété de cette numérotation

Pour tout sommet i,

- $filsG(i) = 2i + 1$
- $filsD(i) = 2i + 2$
- $pere(i) = \lfloor \frac{i-1}{2} \rfloor$



Semi-complet : conséquences

Preuve 1.

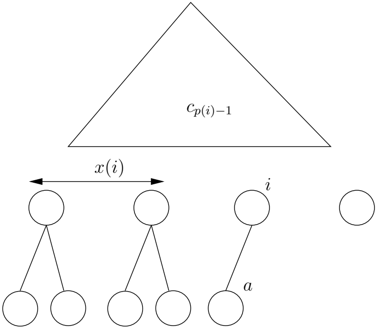
Semi-complet : conséquences

Preuve 1.

Preuve 1.

Soit i un sommet, on a :

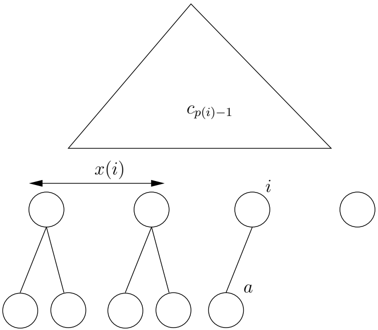
- $i = c_{p(i)-1} + x(i)$
- $a = c_{p(i)} + 2x(i)$



Preuve 1.

Soit i un sommet, on a :

- $i = (2^{p(i)} - 1) + x(i)$
- $a = (2^{(p(i)+1} - 1) + 2x(i)$
- donc $a = 2i + 1$, et
  $b = a + 1 = 2i + 2$.

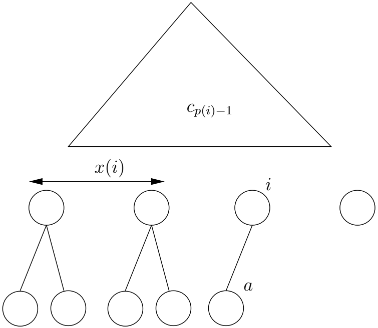


Preuve 1.

On a donc

- $filsG(i) = 2i + 1$
- $filsD(i) = 2i + 2$

Ce qui implique $pere(x) = \lfloor \frac{x-1}{2} \rfloor$

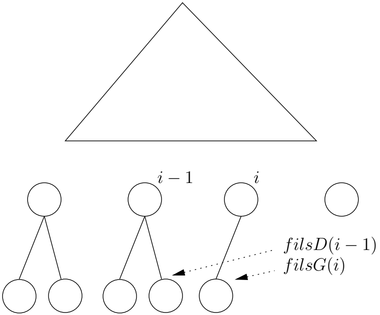


Semi-complet : conséquences

Preuve 2.

Observation

Les fils de i sont numérotés juste après les fils de $i-1$.
Autrement dit: $filsG(i) = filsD(i-1) + 1$



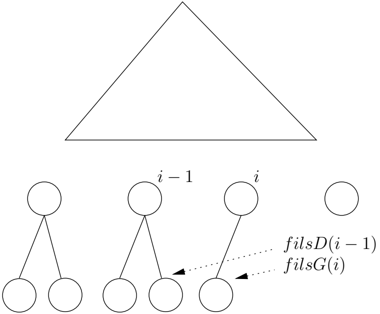
Semi-complet : conséquences

Preuve 2.

Observation

Les fils de i sont numérotés juste après les fils de $i-1$.
Autrement dit: $\mathit{filsG}(i) = \mathit{filsD}(i-1)+1$

Par récurrence :



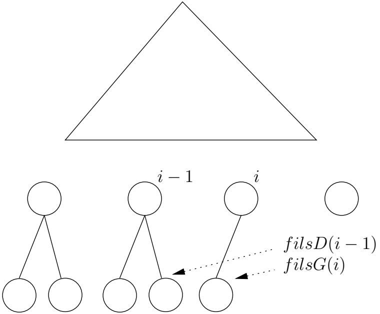
Preuve 2.

Observation

Les fils de i sont numérotés juste après les fils de $i-1$.
Autrement dit: $filsG(i) = filsD(i-1)+1$

Par récurrence :

- $i = 0$ évident



Semi-complet : conséquences

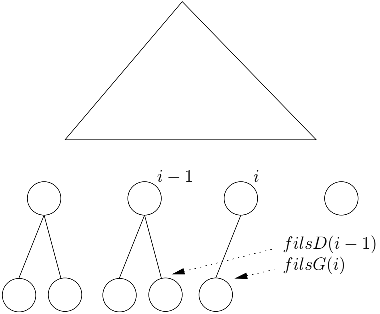
Preuve 2.

Observation

Les fils de i sont numérotés juste après les fils de $i-1$.
Autrement dit: $\mathit{filsG}(i) = \mathit{filsD}(i-1)+1$

Par récurrence :

- Prop vraie pour $i-1 \Rightarrow$ vraie pour i:
  - par hyp de rec: $\mathit{filsD}(i-1) = 2(i-1)+2$
  - et donc $\mathit{filsG}(i) = 2(i-1)+3 = 2i+1$
  - et $\mathit{filsD}(i) = \mathit{filsG}(i)+1$



Semi-complet : conséquences

Conséquence de la propriété de numérotation

- On peut naviguer dans l'arbre aussi facilement que si l'on avait les pointeurs habituels sur les fils et le père!
- Mais d'ailleurs, qu'apporte le stockage par tableau par rapport à la classe `Arbre` récursive habituelle avec ses pointeurs ?
  $\rightarrow$ WAIT, YOU'LL SEE!!

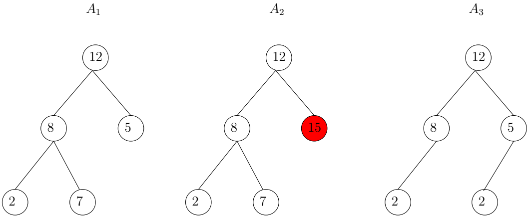
Exemple jouet d'une méthode dans la classe `Heap`

```
public void print(int i){
  //0 <= i < size(), affiche les elements de A
      [i]
  System.out.println(t.get(i)+ "␣");
        if(left(i)<size()) //left(i)=2i+1
              print(left(i));
        if(right(i)<size()) //right(i)=2i+2
              print(right(i));
}
```

Définition

Un <u>max-tas (heap)</u> est un arbre A semi-complet et ayant la propriété du maximum.

- A_1 est un tas
- A_2 n'est pas un tas (semi-complet, n'a pas la propriété du maximum)
- A_3 n'est pas un tas (pas semi-complet, a la propriété du maximum)



Tas

Définition

Un max-tas (heap) est un arbre A semi-complet et ayant la propriété du maximum.

Définition

Un min-tas (heap) est un arbre A semi-complet et ayant la propriété du minimum (élément petits en haut). Tout est symétrique, on ne parlera donc dans ce cours que des max-tas.

Tas : implémentation en TP

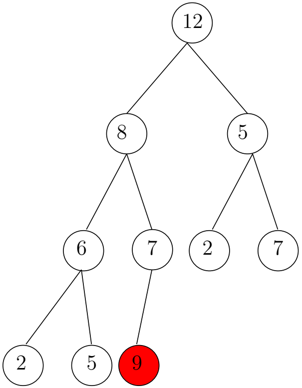
```
public class Heap<T extends Comparable<T>>{
  private ArrayList<T> t;
  //plus besoin de se souvenir de la taille
      : t.size() est en O(1)

  public void add(T e){
     ..
  }
  public T removeTop(){
     ..
  }
  public void hasChangedPriority(T e){
     ..
  }
}
```

Voyons maintenant comment implémenter les méthodes!

Outline

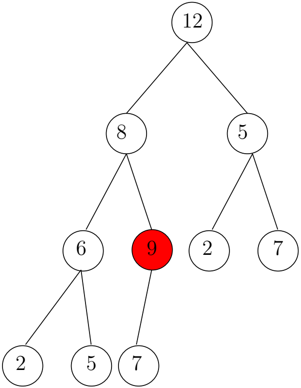
add(T e)

```
public void add(T e){
  ajouter e dans la derniere
     case de t
  heapifyUp(t.size()-1)
}

public void heapifyUp(int i){
  //prerequis : cf ci-dessous
  //action : modifie this
     pour en refaire un tas
  //startegie : faire
     remonter (par des
     echanges) l'element
     initialement en t[i]
     tant qu'il est plus
     grand que son pere
}
```



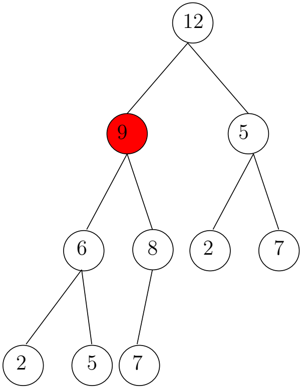
add(T e)

```
public void add(T e){
  ajouter e dans la derniere
     case de t
  heapifyUp(t.size()-1)
}

public void heapifyUp(int i){
  //prerequis : cf ci-dessous
  //action : modifie this
     pour en refaire un tas
  //startegie : faire
     remonter (par des
     echanges) l'element
     initialement en t[i]
     tant qu'il est plus
     grand que son pere
}
```



add(T e)

```
public void add(T e){
  ajouter e dans la derniere
     case de t
  heapifyUp(t.size()-1)
}

public void heapifyUp(int i){
  //prerequis : cf ci-dessous
  //action : modifie this
     pour en refaire un tas
  //startegie : faire
     remonter (par des
     echanges) l'element
     initialement en t[i]
     tant qu'il est plus
     grand que son pere
}
```

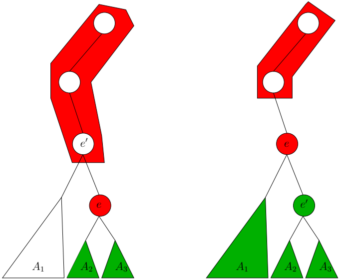


L'algorithme précédent fonctionne car il maintient les invariants suivants à chaque échange :

- semi-complétude : la structure ne change pas (l'arbre reste donc semi-complet)
- propriété du max :
  - e est bien $\geq$ à tous ses descendants
  - les seuls problèmes potentiels sont entre e et ses ascendants

Preuve de l'invariant pour la propriété du max. Supposons que:

- $e \geq$ aux éléments de A_2, A_3
- les seuls problèmes potentiels sont entre e et ses ascendants

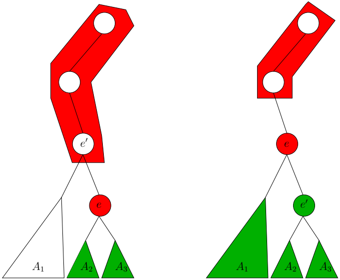


add(T e)

L'algorithme précédent fonctionne car il maintient les invariants suivants à chaque échange :

- semi-complétude : la structure ne change pas (l'arbre reste donc semi-complet)
- propriété du max :
  - e est bien $\geq$ à tous ses descendants
  - les seuls problèmes potentiels sont entre e et ses ascendants

On échange e et e' car $e \geq e'$

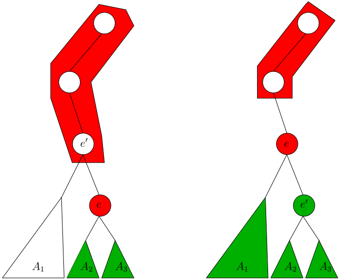


add(T e)

L'algorithme précédent fonctionne car il maintient les invariants suivants à chaque échange :

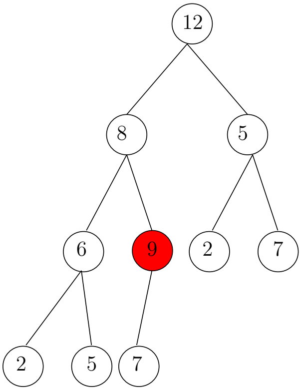
- semi-complétude : la structure ne change pas (l'arbre reste donc semi-complet)
- propriété du max :
  - e est bien $\geq$ à tous ses descendants
  - les seuls problèmes potentiels sont entre e et ses ascendants

- $e \geq$ aux éléments son nouveau sous arbre droit
- $e \geq$ aux éléments de A_1 (car $e \geq e' \geq$ éléments de A_1)
- on ne crée pas de problèmes d'ordre ailleurs

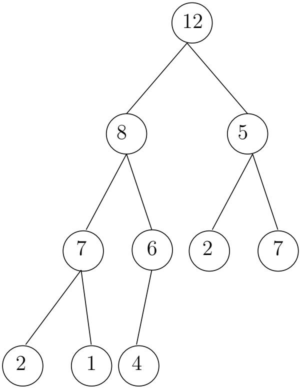


add(T e)

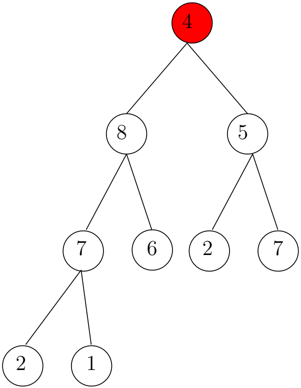
```
public void add(T e){
  ajouter e dans la derniere
     case de t
  heapifyUp(t.size()-1)
}

public void heapifyUp(int i){
  //prerequis :
  //this semi-complet
  //seuls problemes d'ordre
     potentiel entre element
     en i et ses ascendants

}
```



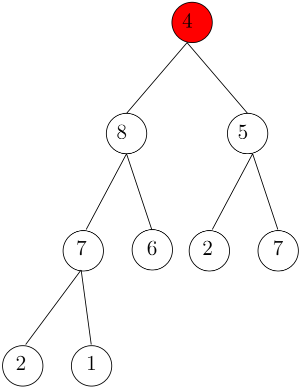
```
public T removeTop(){
  res = t[0]
  échanger t[0] et t[t.size()-1]
  t.remove(t.size()-1) //O(1)
  heapifyDown(0)
}

public void heapifyDown(int i){
  //prerequis : cf ci-dessous
  //action : modifie this pour
     en refaire un tas
  //startegie : fait descendre (
     par des echanges)
  //l'element initialement en
  //t[i] tant qu'il est plus
     petit que des éléments de
     ses sous arbres
}
```



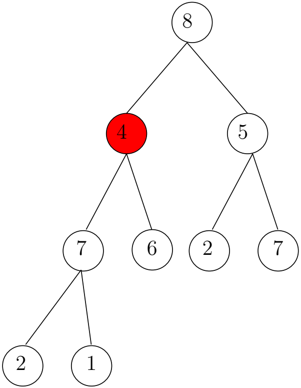
```
public T removeTop(){
  res = t[0]
  échanger t[0] et t[t.size()-1]
  t.remove(t.size()-1) //O(1)
  heapifyDown(0)
}

public void heapifyDown(int i){
  //prerequis : cf ci-dessous
  //action : modifie this pour
     en refaire un tas
  //startegie : fait descendre (
     par des echanges)
  //l'element initialement en
  //t[i] tant qu'il est plus
     petit que des éléments de
     ses sous arbres
}
```



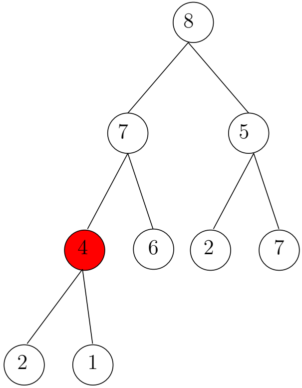
Attention, à chaque étape, il faut échanger l'élément avec le maximum de ses deux fils



Attention, à chaque étape, il faut échanger l'élément avec le maximum de ses deux fils



Attention, à chaque étape, il faut échanger l'élément avec le maximum de ses deux fils



`removeTop()`

L'algorithme précédent fonctionne car il maintient les invariants suivants à chaque échange :

- semi-complétude : la structure ne change pas (l'arbre reste donc semi-complet)
- propriété du max :
  - e est bien $\leq$ à tous ses ascendants
  - les seuls problèmes potentiels sont entre e et ses descendants

Question laissée en suspens

- "Mais d'ailleurs, qu'apporte le stockage par tableau par rapport à la classe `Arbre` récursive habituelle avec ses pointeurs ?"

$\rightarrow$ On aurait été bien embêté pour faire (en $\mathcal{O}(1)$) l'équivalent de "je commence par ajouter/supprimer l'élément dans la dernière case du tableau"

Outline

Tris par tas (heap sort)

Tris par tas

Etant donné un tableau d'entiers t à trier par ordre croissant :

1. créer un tas h et y ajouter tous les $t[i]$ dans h (complexité $\mathcal{O}(nlog(n))$)
2. exécuter n fois $h.removeTop()$ (complexité $\mathcal{O}(nlog(n))$)

Remarques

- on rappelle que $\mathcal{O}(nlog(n))$ est la meilleure complexité possible pour un tri dans le cas général
- l'étape 1 peut même se faire en $\mathcal{O}(n)$, mais ne changerait pas la complexité totale

Priority Queue

Structure permettant de stocker des couples (e, p) d' éléments e `T` avec une priorité $p \in \mathbb{R}$, et fournissant

- `void add(T e, double p)`
- `T removeTop()` et `T getTop()`

Application : gestion des processus dans un OS

- A chaque fin de tranche de temps, l'OS doit sélectionner la prochaine tâche élue, qui sera celle de priorité maximum

$\rightarrow$ Il suffit d'utiliser une `PriorityQ<Tache>`

Priority Queue

Structure permettant de stocker des couples (e, p) d' éléments e `T` avec une priorité $p \in \mathbb{R}$, et fournissant

- `void add(T e, double p)`
- `T removeTop()` et `T getTop()`

Implémentation d'une priority queue

- il suffit d'utiliser un tas
- attention : un tas à besoin d'éléments comparables, alors qu'ici `T` est un type quelconque
- solution : écrire une classe `ElemWithPriority<T>` servant à stocker un élément et sa priorité, qui implémente l'interface `Comparable`

$\rightarrow$ la file de priorité se contente de construire les couples et de les ranger dans son tas (cf TP)

Algorithme de Dijkstra

Problème du plus court chemin

- Entrée : $G = (V, E)$ avec des poids strictement positifs sur les arêtes, et deux sommets s et t
- Sortie : un $s - t$ chemin C (et un chemin vide si un tel chemin n'existe pas)
- But : minimiser $l(C)$, la longueur du chemin (somme poids des arêtes)

- il existe beaucoup d'algorithmes polynomiaux pour résoudre ce problème
- nous allons étudier celui de Dijkstra, dont une implémentation classique utilise les files de priorités (et donc les tas)

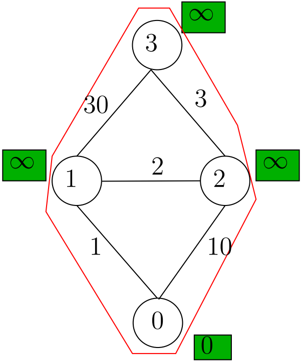
Variables importantes :

- un tableau `dist` : `dist[i]` mémorise la meilleure distance trouvée pour aller de `s` à `i`
- un tableau `prec` : `prec[i]` mémorise de quel sommet on vient pour atteindre `i` (dans le meilleur chemin actuel trouvé pour aller de `s` à `i`)
- un ensemble `avoir` : contenant les numéros de sommets restant à examiner

Algorithme de Dijkstra

Boucle principale :

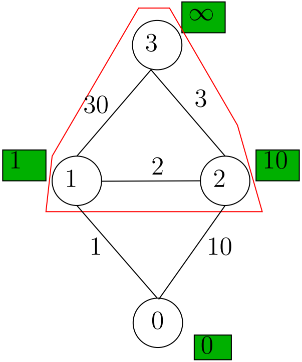
- retirer un sommet `u` de `avoir` ayant la plus petite `dist`
- mettre à jour les voisins de `u`



Algorithme de Dijkstra

Boucle principale :

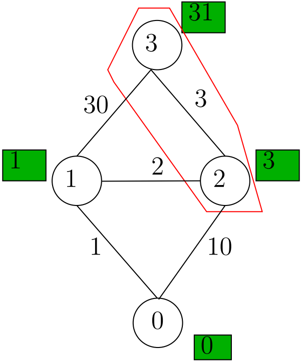
- retirer un sommet `u` de `avoir` ayant la plus petite `dist`
- mettre à jour les voisins de `u`



Algorithme de Dijkstra

Boucle principale :

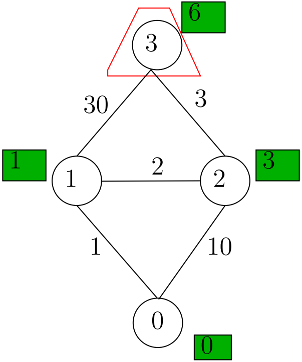
- retirer un sommet `u` de `avoir` ayant la plus petite `dist`
- mettre à jour les voisins de `u`



Algorithme de Dijkstra

Boucle principale :

- retirer un sommet `u` de `avoir` ayant la plus petite `dist`
- mettre à jour les voisins de `u`



Algorithme de Dijkstra

Boucle principale :

```
public ArrayList<Integer> dijkstra(int s, int t)
  .. init dist, prec, avoir (avec tous sommets)

  while (!trouve && avoir.size() > 0) {
    int u = enlever de avoir un sommet v ayant
          la plus petit dist
    if (u == t)
           trouve = true;
     else
           pour tous les voisins v de u
              if (dist[u] + c(uv) < dist[v])
                 prec[v] = u;

  }
  .. retourner chemin si existe grâce à prec
```

Algorithme de Dijkstra

Complexité

- cela dépend de la structure de donnée utilisée pour `avoir`
- analysons avec un tableau naïf, puis avec une priority queue

Avec un tableau

- au plus n itérations de la boucle
- une itération coûte $t_{extraireMin} + t_{miseJourVois}$
- $t_{extraireMin} = \mathcal{O}(n)$ et $t_{miseJourVois} = \mathcal{O}(n)$
- Donc version naïve en $\mathcal{O}(n^2)$

Avec une priority queue?

Regardons comment adapter l'algorithme.

Algorithme de Dijkstra

Complexité

- cela dépend de la structure de donnée utilisée pour `avoir`
- analysons avec un tableau naïf, puis avec une priority queue

Avec un tableau

- au plus n itérations de la boucle
- une itération coûte $t_{extraireMin} + t_{miseJourVois}$
- $t_{extraireMin} = \mathcal{O}(n)$ et $t_{miseJourVois} = \mathcal{O}(n)$
- Donc version naïve en $\mathcal{O}(n^2)$

Avec une priority queue?

- la priority queue stocke des couples (i, d) avec i sommet et $d = dist[i]$ (distance courante pour i)
- on utilise une min priority queue (et pas max priority queue)

```
public ArrayList<Integer> dijkstra(int s, int t)
  .. init dist, prec, avoir (avec tous sommets)

  while (!trouve && avoir.size() > 0) {
    int u = enlever de avoir un sommet v ayant
        la plus petit dist //log(n) avec
        priorityQ
    if (u == t)
         trouve = true;
     else
         pour tous les voisins v de u
           if (dist[u] + c(uv) < dist[v])
           prec[v] = u;
           dist[v] = dist[u] + c(uv);
           //on a envie de faire :
           add.changePriority(v,dist[u] + c(uv));

  }
  .. retourner chemin si existe grâce à prec
```

Algorithme de Dijkstra

Problème

- la priority queue que nous avons considérée ne sait pas faire `changePriority(elem,newPrio)`
- on pourrait le rajouter, mais une implémentation naïve nécessiterait $\mathcal{O}(n)$ (rien que pour retrouver l'élément dans le tas!)
- une solution en $\mathcal{O}(log(n))$ existe, mais nécessite d'ajouter une `Map` dans le `Heap` pour se souvenir d'où sont les éléments
- nous allons contourner le problème en insérant une nouvelle fois le sommet v (avec sa nouvelle priorité $dist[u] + c(uv)$)
- cela va créer des doublons dans la file, mais cela ne posera pas de problèmes car on retirera d'abord le sommet v avec sa plus petite priorité : cf ex suivant

Algorithme de Dijkstra

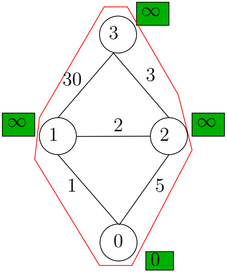
Boucle principale :

- retirer de la file un sommet `u` de `avoir` de min dist.
- vérifier si `u` a déjà été traité (donc on s'ajoute un autre tableau de booléens pour s'en souvenir)
- si c'est la première fois que l'on voit `u`:
  - ~~mettre à jour les voisins de `u`~~

Algorithme de Dijkstra

Boucle principale :

- retirer de la file un sommet `u` de `avoir` de min dist.
- vérifier si `u` a déjà été traité (donc on s'ajoute un autre tableau de booléens pour s'en souvenir)
- si c'est la première fois que l'on voit `u`:
  - insérer les voisins de `u` avec leur nouvelle distance (peut créer des doublons)

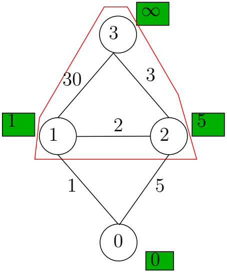


File de priorité :
$(0, 0), (1, \infty), (2, \infty), (3, \infty)$

Algorithme de Dijkstra

Boucle principale :

- retirer de la file un sommet `u` de `avoir` de min dist.
- vérifier si `u` a déjà été traité (donc on s'ajoute un autre tableau de booléens pour s'en souvenir)
- si c'est la première fois que l'on voit `u`:
  - insérer les voisins de `u` avec leur nouvelle distance (peut créer des doublons)

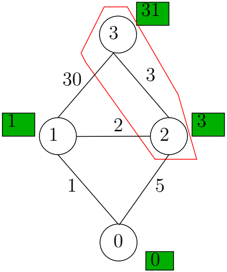


File de priorité :
$(1, 1), (2, 5), (1, \infty), (2, \infty), (3, \infty)$

Algorithme de Dijkstra

Boucle principale :

- retirer de la file un sommet `u` de `avoir` de min dist.
- vérifier si `u` a déjà été traité (donc on s'ajoute un autre tableau de booléens pour s'en souvenir)
- si c'est la première fois que l'on voit `u`:
  - insérer les voisins de `u` avec leur nouvelle distance (peut créer des doublons)

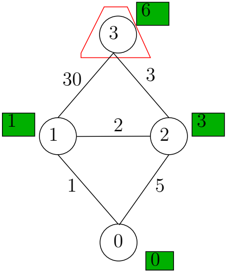


File de priorité :
$(2,3),(2,5),(3,31),(1,\infty),$
$(2,\infty),(3,\infty)$

Algorithme de Dijkstra

Boucle principale :

- retirer de la file un sommet `u` de `avoir` de min dist.
- vérifier si `u` a déjà été traité (donc on s'ajoute un autre tableau de booléens pour s'en souvenir)
- si c'est la première fois que l'on voit `u`:
  - insérer les voisins de `u` avec leur nouvelle distance (peut créer des doublons)

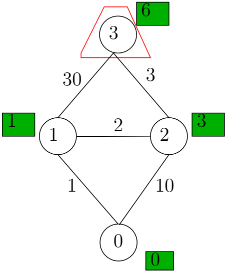


File de priorité :
$(2,5),(3,6),(3,31),(1,\infty),$
$(2,\infty),(3,\infty)$
On va retirer $(2,5)$, et ne rien faire car 2 est déjà traité.

Algorithme de Dijkstra

Boucle principale :

- retirer de la file un sommet `u` de `avoir` de min dist.
- vérifier si `u` a déjà été traité (donc on s'ajoute un autre tableau de booléens pour s'en souvenir)
- si c'est la première fois que l'on voit `u`:
  - insérer les voisins de `u` avec leur nouvelle distance (peut créer des doublons)



File de priorité :
$(3, 6), (3, 31), (1, \infty),$
$(2, \infty), (3, \infty)$

Algorithme de Dijkstra

Complexité avec une priority queue

- au plus n "vraies" itérations de la boucle (où l'on fait un traitement), et m itérations "bidons" où on retire un sommet déjà traité et où on ne fait rien

- une itération où l'on traite le sommet u coûte $t_u = t_{extraireMin} + d(u)(\mathcal{O}(1) + t_{add})$ avec $d(u)$ degré du sommet u
- $t_{extraireMin} = \mathcal{O}(log(n))$ et $t_{add} = \mathcal{O}(log(n))$
- une itération où l'on traite le sommet u coûte $t_u = \mathcal{O}(log(n)) + d(u)(\mathcal{O}(log(n)))$

Algorithme de Dijkstra

Complexité avec une priority queue

- au plus n "vraies" itérations de la boucle (où l'on fait un traitement), et m itérations "bidons" où on retire un sommet déjà traité et où on ne fait rien

- une itération où l'on traite le sommet u coûte $t_u = \mathcal{O}(log(n)) + d(u)(\mathcal{O}(log(n)))$

Complexité avec une priority queue

- au plus n "vraies" itérations de la boucle (où l'on fait un traitement), et m itérations "bidons" où on retire un sommet déjà traité et où on ne fait rien

- une itération où l'on traite le sommet u coûte $t_u = \mathcal{O}(log(n)) + d(u)(\mathcal{O}(log(n)))$

Donc version avec priority queue en (on ne compte pas les initialisations) :

$$sum_{u \in V(G)} t_u + m\mathcal{O}(1)$$

$$\sum_{u \in V(G)} \Big(\mathcal{O}(log(n)) + d(u)(\mathcal{O}(log(n)))\Big) + \mathcal{O}(m)$$

$$\mathcal{O}(nlog(n)) + 2m\mathcal{O}(log(n)) + \mathcal{O}(m)$$

$$\mathcal{O}(nlog(n)) + \mathcal{O}(mlog(n))$$

$$\mathcal{O}(mlog(n))$$

Conclusion

- naïf: $\mathcal{O}(n^2)$
- priority queue: $\mathcal{O}(mlog(n))$
- (améliorable en $\mathcal{O}(m + nlog(n))$ avec un Fibonacci Heap)

Comment se décider ?

- cela dépend de m
- pour un graphe connexe $n - 1 \leq m \leq \frac{n(n-1)}{2} = \mathcal{O}(n^2)$
- quand le graphe est peu dense (m plus proche de n que de n^2), choisir une priority queue

Conclusion de ce cours

Une structure de données : les tas

- la hauteur de l'arbre sous-jacent est optimale et petite : $h = \lfloor \log(n+1) \rfloor$
- cela permet `add`, `removeTop`, et `remove` en $\mathcal{O}(log(n))$

Exemples d'application

- tri par tas
- file de priorité
- Disjkstra