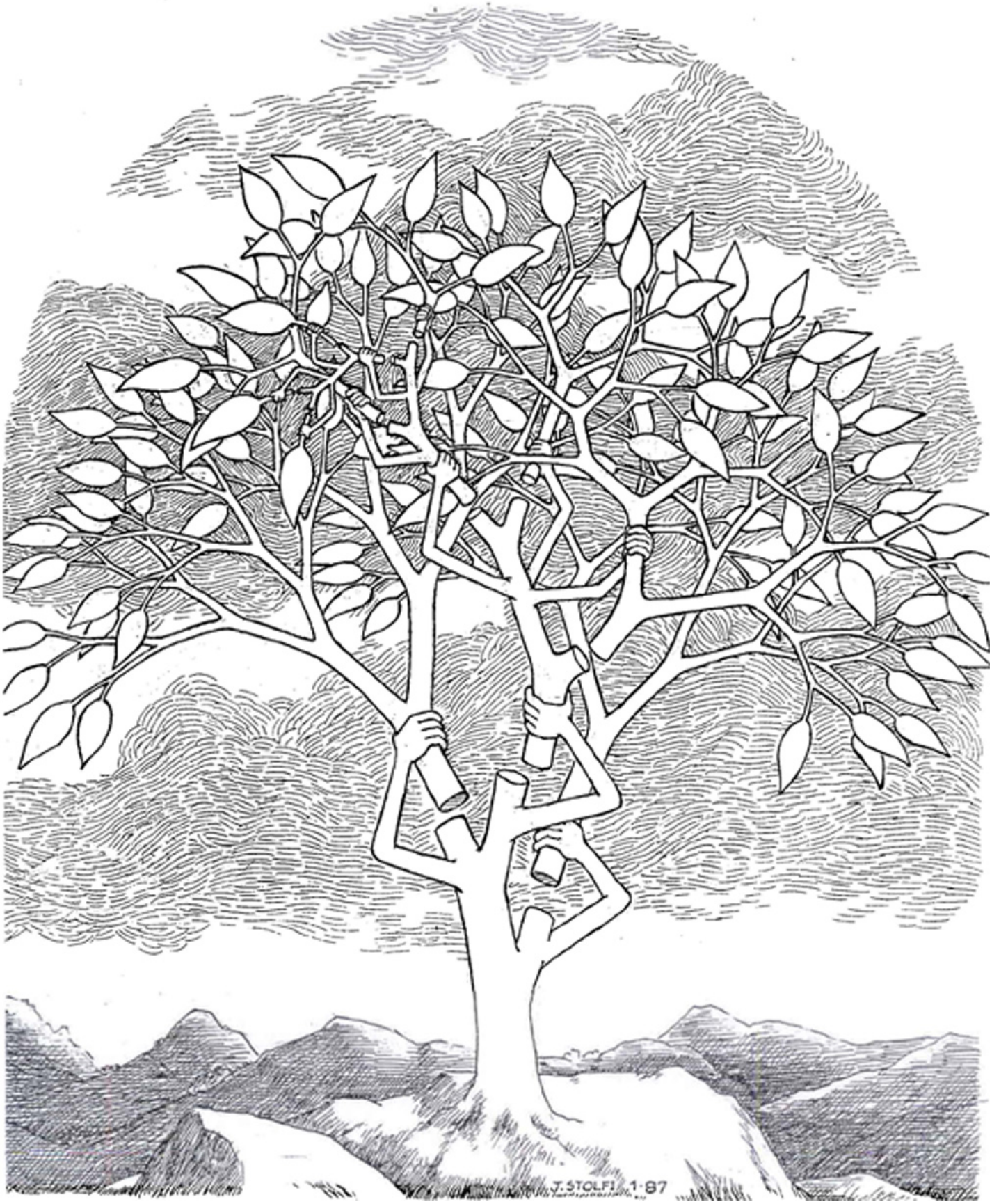
Binomial and Fibonacci heap



Index

Introduction

Binomial heaps are data structures implemented as a collection of binomial trees, (A binomial tree of order K can be constructed from two trees of order (K-1)). They can implement several methods:

Min, Insert, Union, ExtractMin, DecreaseKey and Delete.

Fibonacci heaps are similar to binomial heaps, but they have better performances in the mean of the Amortized analysis, This methods have a cost of O(1) except for ExtractMin and Delete (O(lg n))

Fibonacci heaps are used to improve the cost of Dijikstra and Prim.

We implemented first the algorithms of Binomial and Fibonacci. We then used ExtractMin of fibonacci so as to implement Prim and Dijikstra.

We have made a bonus algorithm , Kruskal who is also a "Minimum Spanning Tree " algorithm.

Rapport d'algorithmique

Supervisor :

Professor Amin Shokrollahi

Mr. Mahdi Cheraghchi

Students:

Chekir Ali

Mohamed Slim Slama

We have made two versions for our report. A Frensh version and an English one .
Our english is not as good as our french, and we hope that our english version would be clear.
So iIf you have a doubt in the english version please don't hesitated to take a look in the french one.

Thank you.

I. Dijkstra

1.1 DIJKSTRA ET FIBONACCI :

DONNEES :
G = (V,E) avec V = n et E = m.
V représente les sommets, et E les arêtes de pondération non négative.

BUT :
Déterminer un plus court chemin de s a v avec s != v.

1.1.1 L'Algorithmique DE DIJKSTRA :

L'algorithme utilise les méthodes suivantes :

Dijkstra(G,Poids,sdeb)
1 Initialisation(G,sdeb)
2 P := ensemble vide
3 Q := ensemble de tous les nœuds
4 tant que Q n'est pas un ensemble vide
5 faire s1 := Trouve_min(Q)
6 P := P union {s1}
7 pour chaque nœud s2 voisin de s1
8 faire maj_distances(s1,s2)

★cette méthode nous permet d'elaborer l'algorithme de dijkstra.

maj_distances(s1,s2)
1 si d[s2] > d[s1] + Poids(s1,s2)
2 alors d[s2] := d[s1] + Poids(s1,s2)
3 prédecesseur[s2] := s1 /*on fait passer le chemin par s1*/

★La méthode Maj_distance nous amène une Mise à jour des distances entre s1 et s2 .

Initialisation(G,sdeb)
1 pour chaque point s de G
2 faire d[s] := infini
3 prédecesseur[s] := 0 /*car on ne connait au départ aucun chemin entre s et sdeb*/
4 d[sdeb] := 0 /*sdeb est le point le plus proche de sdeb!*/

★initialisation de l'énoncé.

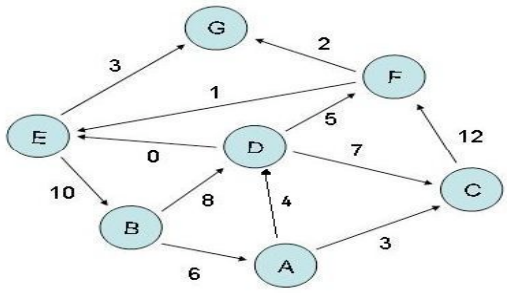
1.1.2 Cout de l'algorithme :

Si nous stockons les distances provisoires dans un tableau , la recherche du minimum coûte O(n) et l'algorithme a une complexitée de O(n^2) indépendamment de la densitée du graphe car l'ensemble des opérations de maj_distance s'effectue en O(n) si le graphe est dense,
Cette complexité est optimale, car il faut bien examiner toutes les arêtes si nous utilisons un heap classique en maintenant pour chaque sommet un pointeur sur le nœud.

L'opération maj_distance coûterait O(m.log(n)) et la détermination d'un sommet v avec d[v] = min sachant que le nœud n'est pas marqué nous coûterait O(n.log(n)) (ce qui est meilleur pour des graphes peu dense mais moins bon pour les graphes dense).

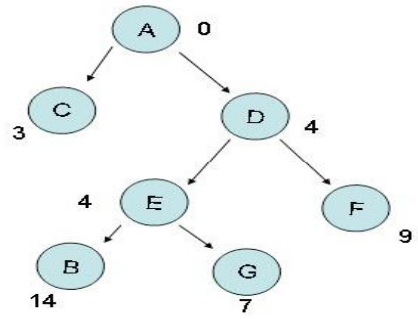
1.1.3 Mecanisme l'algorithme de DIJKSTRA :

SOIT LE GRAPHE SUIVANT :



★Appliquons l'algorithme de Dijkstra à partir du point A pour trouver les plus courts chemins :

	S	A	B	C	D	E	F	G
Initialisation	-	**0**,-	00,A	00,A	00,A	00,A	00,A	00,A
1er itération	{A}	**0**,-	00,A	**3**,A	4,A	00,A	00,A	00,A
2ème itération	{A, C}	**0**,-	00,A	**3**,A	**4**,A	00,A	15,C	00,A
3ème itération	{A, C D}	**0**,-	00,A	**3**,A	**4**,A	**4**,D	9,D	00,A
4ème itération	{A,C,D,E}	**0**,-	14,E	**3**,A	**4**,A	**4**,D	9,D	**7**,E
5ème itération	{A,C,D,E,G}	**0**,-	14,E	**3**,A	**4**,A	**4**,D	**9**,D	**7**,E
6ème itération	{A,C,D,E,G F}	**0**,-	**14**,E	**3**,A	**4**,A	**4**,D	**9**,D	**7**,E
7ème itération	{A,C,D,E,G,F,B}	**0**,-	**14**,E	**3**,A	**4**,A	**4**,D	**9**,D	**7**,E



★Donc pour avoir le chemin le plus court pour aller de A vers B, il suffit de suivre le chemin :

A->D->E->B

Dans notre implémentation, le OUTPUT consiste en l'affichage pour chaque nœud du nœud précédent ainsi que la distance nécessaire pour y accéder.

1.1.4 Overture :

Si nous utilisons un heap de fibonacci, on obtiendrait une complexité en O(n.log(n) + m).

1.2 DIJKSTRA avec FIBONACCI :

Pour résumer, l'algorithme de Dijkstra consiste à trouver le plus court chemin d'un sommet donné à tous les nœuds de l'arbre.
Dans notre étude, nous allons implémenter cet algorithme en utilisant Fibonacci, plus précisément la méthode extractmin() qui n'a un coût que de O(log(n)).

Pour l'implémentation de l'algorithme de Dijkstra, nous avons créé trois classes une classe principale Dijkstra une autre classe liens et enfin la classe Graphenode.

On crée d'abord les nœuds et les liens. Chaque nœud contient un fibnode qu'on insérera dans un fibheap, qui nous servira pour la méthode extractmin de Fibonacci, un entier dgre déterminant le nombre de voisin, la Key, un entier distance, dans lequel on stockera la distance minimum de ce noeud par rapport au nœud traiter, il contient aussi un vecteur de nœud qui contient tous ses voisins et enfin un Graphenode pred déterminant le nœud qui le précède dans l'arbre couvrant minimum.
Pour être plus clair, nous allons raisonner de la façon suivante dans chaque nœud nous allons remplir les distance et le pred pour ainsi trouver l'arbre couvrant minimum.
Ainsi en utilisant la méthode extractmin de Fibonacci on diminue le temps de parcours.

La classe lien contient un entier déterminant le poids de l'arête, et un Graphenode qui détermine le nœud au bout de l'arête.

La méthode initialisation prend un vecteur de nœud en paramètre et le rempli. On ne l'utilise que pour le premier pas de notre algorithme.

La méthode maj_distance change la distance des nœuds voisins de telle sorte à en garder la distance minimum.

1.2.1 Fonctionnement :

D'abord, on initialise les distances du nœud traité avec ses voisins, Pour cela on utilise initialisation().
Ensuite, on crée un heap de fibonacci avec les fib de tous les nœuds.
On entre après dans une boucle while sur le nombre de sommet, on extrait dès lors le minimum du heap déjà crée , et l'on étudie ce sommet et tous ses voisins avec maj_distance.

II. L'Algorithme de PRIM:

L'algorithme de Prim est implémenté de la même manière que Dijkstra, mais une petite différence subsiste c'est que la distance rechercher n'est pas d'un sommet source mais de telle sorte à déterminer un arbre couvrant minimum.
Les arêtes du graphe de prim ne sont pas orientés contrairement a celles de dijkstra.

Un changement est ainsi nécessaire dans la méthode maj_distance Pour déterminer la distance il faut juste prendre en compte le poids de l'arête étudier sans y additionner la distance de l'arête en question.
Un autre problème surgi c'est qu'avant de faire maj_distance il faut vérifier que le nœud a étudié n'est pas contenu dans le vecteur P dont on remplie au fur et au mesure qu'on travaille sur les différents minimums afin d'éviter les cycles.

III. L'Algorithme de KRUSKAL:

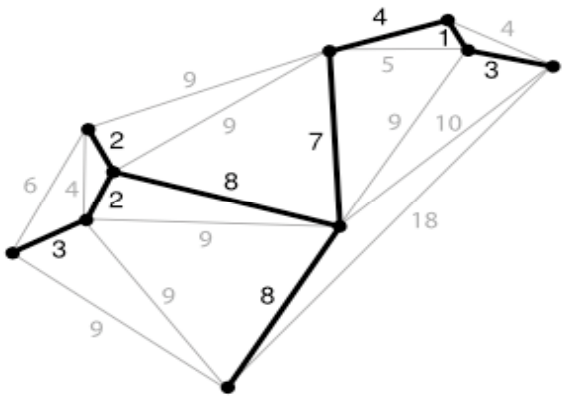
L'algorithme de kruskal consiste à déterminer l'arbre couvrant minimal. Pour ceci on range les arêtes par ordre croissant, on prend les arêtes un par un dans cet ordre si elle ne crée pas un cycle.

Chaque nœud à un entier « appartenance » qui détermine les sommets avec lesquels il est déjà lié avec les arêtes précédemment choisies.

Pour voir s'ils créent un cycle, à chaque fois qu'on insère une arête on étudie les nœuds concernés s'ils ont la même appartenance on n'insère pas l'arête, si les appartenances sont différentes nous insérons

l'arête et nous mettons tous les nœuds de deux arbre dans un même arbre avec une même « appartenance », ensuite on recommence…

KRUSKAL (G,w)
1 E := ø
2 pour chaque sommet v de G
3 faire CRÉER-ENSEMBLE (v)
4 trier les arêtes de G par ordre croissant de poids w
5 pour chaque arête (u,v) de G prise par ordre de poids croissant
6 faire si ENSEMBLE-REPRÉSENTATIF (u) ≠ ENSEMBLE-REPRÉSENTATIF (v)
7 alors ajouter l'arête (u,v) à l'ensemble E
8 UNION (u,v)
9 retourner E



Dans notre implémentation, le OUTPUT consiste en l'affichage, pour chaque arête prise, du nœud précédent ainsi que le suivant (composant l'arête) et la distance nécessaire pour y accéder.

IV. Binomial heap

Un heap binomial est en fait un ensemble d'arbres binomiaux. Un arbre binomial est défini comme suit:
* L'arbre binomial d'ordre 0 est un simple nœud
* L'arbre binomial d'ordre k possède une racine de degré k et ses fils sont racines d'arbre binomiaux d'ordre k-1, k-2, ..., 2, 1, 0 (dans cet ordre)

Les Heaps binomiaux sont des collections d'arbres. Les arbres y sont rooted de façon ordonnée selon leur dégrées.
Comme les Heaps fibonacci, les opérations se référant à un nœud donné implique l'envoie de ce nœud en paramètre. C'est d'ailleurs pourquoi l'on a implémenter les méthodes VectHeap et VectTree qui nous ont permis un accès simple au nœud désirée.

Un heap binomial est implémenté en tant qu'ensemble de tas binomiaux satisfaisant aux propriétés suivantes:
* Chaque arbre binomial du tas possède une structure ordonnée: la clé de chaque nœud est supérieure ou égale à celle de son parent.
* Pour tout j entier naturel, il existe au plus un tas binomial d'ordre j (en effet $2^{k-1} + 2^{k-1} = 2^k$).

La seconde propriété implique qu'un heap binomial contenant n éléments consiste en au plus ln n + 1 arbres binomiaux. En fait, le nombre et les ordres de ces arbres est déterminé de manière unique par le nombre n d'éléments: chaque tas binomial correspond au bit 1 dans l'écriture binaire du nombre n. Par exemple, 13 correspond à 1101 en binaire, $2^3 + 2^2 + 2^0$, et donc le tas binomial à 13 éléments consistera en 3 arbres binomiaux d'ordres respectifs 3, 2 et 0 (cf. figure ci-dessous) Les racines des arbre binomiaux sont stockés dans une liste indicée par l'ordre des l'arbre.

Pour un arbre binomial Bk , il y'a 2^k nœuds, et ainsi la hauteur de l'arbre est de k, et il existe (k.i) nœuds a une profondeur de i = 0,1,2,…,k.

4.1 Structures :

Node :

Chaque nœud à un pointeur vers son nœud père ainsi qu'un de ses frères « sibling », et un autre vers l'un de ses fils. Le degré d'un nœud

indique le nombre de fils. Les fils sont reliés entre eux de façon linéaire et avec un simple lien (pointeur vers le nœud voisin « sibling »).
Cette façon de procéder nous permet de supprimer un nœud en O(logn).

Heap :

Les roots de tous les arbres dans les Heaps sont reliés entre eux . Cette listes contenant tous les roots est appelée root list. le heap du binomial est constitué que d'un pointeur vers le head, nœud se situant à l'extrémité de la root liste, ainsi on envoi juste le head pour travailler sur le heap.

4.2 Cout des opérations :

	binaire (pire)	**binomial** (pire)	**skew** (amorti)	**Fibonacci** (amorti)
deleteMin	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)
insert	O(log *n*)	O(log *n*)	O(1)	O(1)
merge	O(*n*)	O(log *n*)	O(1)	O(1)
decreaseKey	O(log *n*)	O(log *n*)	O(log *n*)	O(1)

Pour insérer un nouvel élément dans un heap, on crée un nouveau heap contenant uniquement cet élément que nous fusionnons ensuite avec le heap initial, et ce en O(ln n).

Pour trouver le plus petit élément dans un heap, nous avons besoin de trouver le minimum dans la root liste (dont le nombre maximal de nœud est de ln n), ce qui ce fait un coût en O(ln n).

Pour supprimer le plus petit élément du heap, il suffit de le localiser dans la root liste : on obtient alors une liste de ses fils, que l'on transforme en un autre heap binomial qu'on fusionne au heap initial précédent.

Lorsqu'on diminue la Key d'un élément, elle peut devenir plus petite que celle de son père, violant les règle du binomial heap. Dans ce cas, nous échangeons l'élément avec son père (plus précisément les Key), et ainsi de suite jusqu'à ce que l'arbre soit bien ordonné. Chaque arbre binomial ayant pour taille maximale ln n, l'opération est en O(ln n).

Enfin pour supprimer un noeud, on lui donne pour clé moins l'infini (mais, dans notre programme, on lui assigne juste un nombre négatif de l'ordre de -9 ce qui nous suffit, en effet il est plus petite que toute les autres clés...) puis on effectue extrait le min , c'est à dire lui-même.

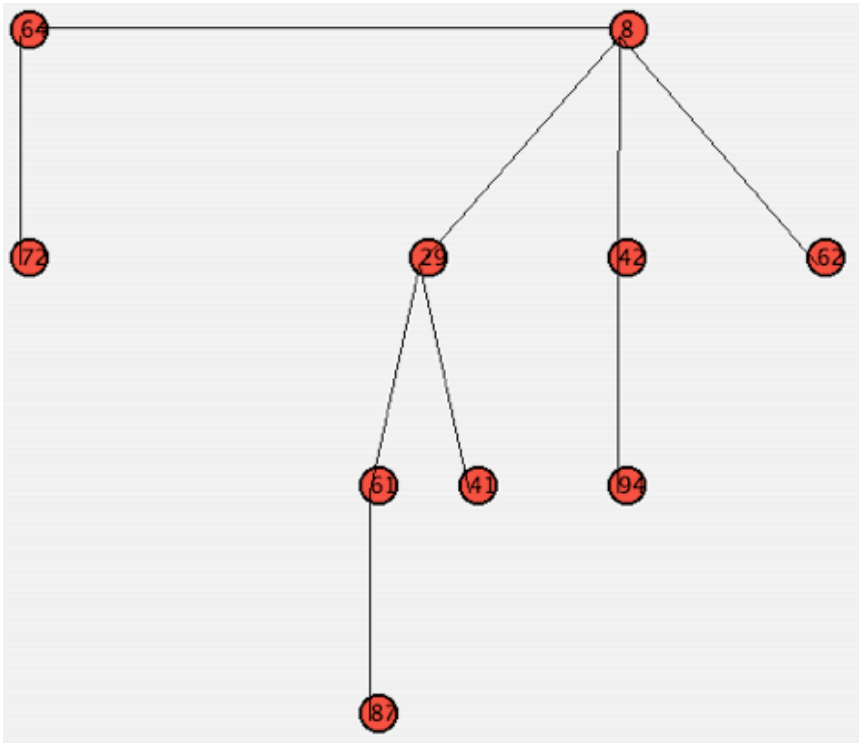
4.3 Les differents algorithme en bref :

- Make-Heap : crée est retourne un nouveau heap ne contenant pas d'élément.
- Insert(H,x) : insert le nœud x dans le heap H .
- Minimum(H) : retourne un pointeur sur le nœud qui a le key minimal.
- Extract-Min(H) : efface le nœud de Key minimum et retourne un pointeur sur le nœud.
- Union(H1,H2) : crée et retourne un nouveau heap qui contient tous les nœuds du heap de H1 et de H2. Les deux heap sont ainsi détruit par cette opération.
- Decrease-Key(H,x,k) : assigne au nœud x une nouvelle valeur k qui ne doit pas être plus grande que la valeur initiale.
- Delete(H,x) : efface le nœud x du heap.

4.4 Les algorithmes plus en détails :

La méthode VectHeap() nous sert à insérer tous les nœuds dans un vecteur vect reçu en paramètre. Cette méthode reçoit en paramètre un nœud N (qui se trouve être le head) et un vecteur de Node vect. Nous parcourrons tous les voisins de N en faisant appelle a chaque fois a la méthode VectTree().

La méthode VectTree() est récursive, elle nous permet d'insérer tous les éléments d'un arbre dans un vecteur vect reçu en paramètre. Cette méthode reçoit aussi en paramètre un entier i et un nœud N. Nous avons utilisé le modèle d'une queue FIFO pour le parcours des nœuds.



EXEMPLE D'EXCUTION:

HEAP INITIAL:---------->

LES ETAPES:
Etape 1 : extractmin
Etape 2 : decreaseKey node de clef 72 réduit à 28
Etape 3 : delete le noeud de clef 28
Etape 4 : insert node de clef 12

B-HEAP-UNION :

```
Binomial-Heap-Union(H1,H2)
H := Make-Binomial-Heap()
head[H] := Binomial-Heap-Merge(H1,H2)
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
  then return H
prev-x := NIL
x := head[H]
next-x := sibling[x]
while next-x <> NIL
    do if (degree[x] <> degree[next-x]) or
          (sibling[next-x] <> NIL
          and degree[sibling[next-x]] = degree[x])
        then prev-x := x
             x := next-x
        else if key[x] <= key[next-x]
             then sibling[x] := sibling[next-x]
                  Binomial-Link(next-x,x)
             else if prev-x = NIL
                  then head[H] = next-x
                  else sibling[prev-x] := next-x
```

```
Binomial-link(x,next-x)
      x := next-x
     next-x := sibling[x]
return H
```

★Explication de l'implémentation :

La méthode BHeapUnion() nous permet d'unir deux heap, elle reçoit en paramètre deux noeuds H1 et H2 (qui sont les heads des deux heap). Nous faisons appelle à la méthode BHeapMerge. Ensuite si nous avons deux roots de même degré on utilise Blink().

B-HEAP-MERGE :

```
Binomial-HeapMerge(H1,H2)
a = head[H1]
b = head[H2]
head[H1] = Min-Degree(a, b)
if head[H1] = NIL
   return
if head[H1] = b
  then b = a
a = head[H1]
while b <> NIL
   do if sibling[a] = NIL
       then sibling[a] = b
          return
       else if degree[sibling[a]] < degree[b]
            then a = sibling[a]
            else c = sibling[b]
               sibling[b] = sibling[a]
               sibling[a] = b
               a = sibling[a]
               b = c
```

★Explication de l'implémentation :

La méthode BHeapMerge() merges les roots listes des deux head de heap envoyer en paramètre H1 et H2 dans un même heap.

B-LINK :

```
Binomial-Link(y,z)
p[y] := z
sibling[y] := child[z]
child[z] := y
degree[z] := degree[z] + 1
```

★Explication de l'implémentation :

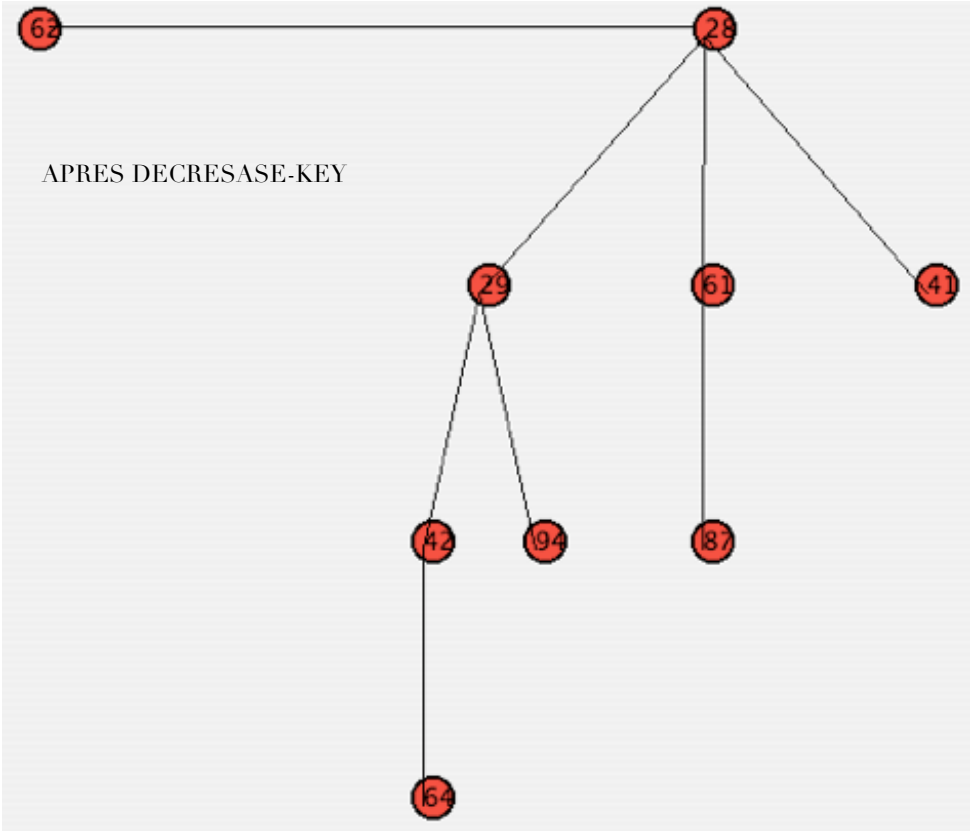
La méthode Blink() reçoit en paramètre deux node y et z, elle consiste à mettre z comme père de y tout en augmentant le degré de z

B-DECREASE-KEY :

```
Binomial-Heap-Decrease-Key(H,x,k)
if k > key[x]
  then error "hew key is greater than current key"
key[x] := k
y := x
z := p[y]
while z <> NIL and key[y] < key[z]
   do exchange key[y] and key[z]
     if y and z have satellite fields, exchange them, too.
     y := z
     z := p[y]
```

★Explication de l'implémentation :

La méthode BinomialHeapDecreasekey() reçoit en paramètre le nœud à changer ainsi que la valeur qu'on veut lui attribuer.
Pour vérifier si le nœud est bien dans le heap, on envoie en paramètre le head et on insert dans le vect (envoyer aussi en paramètre) tous les nœuds a l'aide de vectheap(). Si après la diminution, la key du nœud est inférieur a celle du nœud père alors on échange la key des deux nœuds et ainsi de suite…

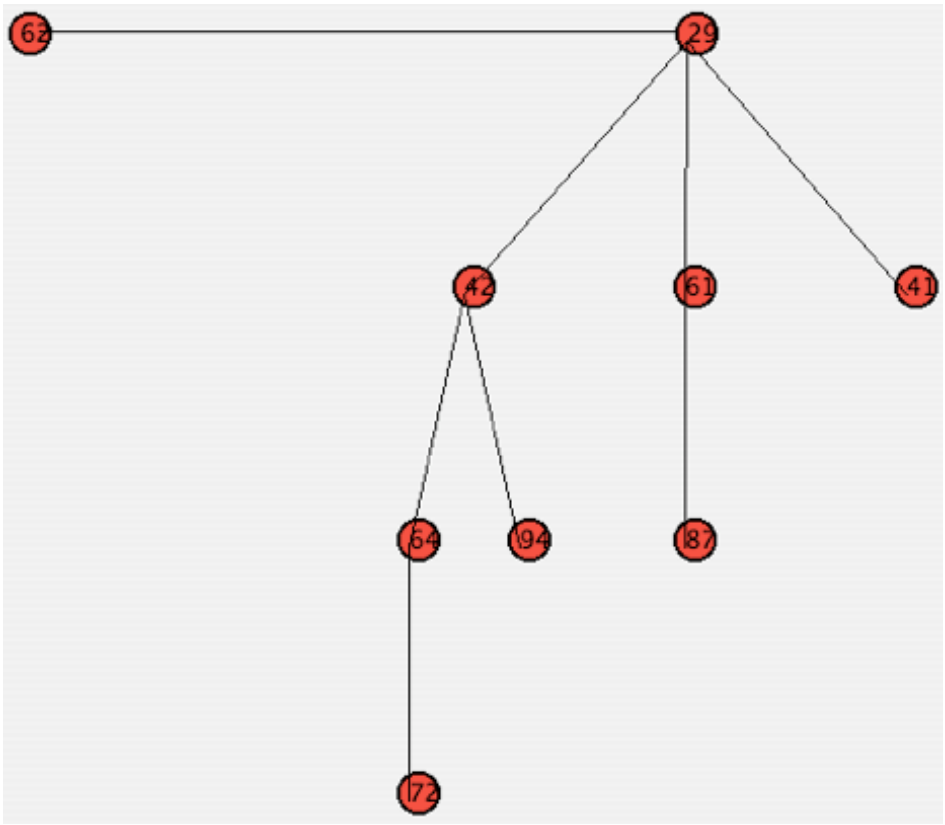


EXTRACT-MIN :

Binomial-Heap-Extract-Min(H)
find the root x with the minimum key in the root list of H,
and remove x from the root list of H
H' := Make-Binomial-Heap()
reverse the order of the linked list of x's children
and set head[H'] to point to the head of the resulting list
H := Binomial-Heap-Union(H,H')
return x

★Explication de l'implémentation :
La méthode ExtractMin() extrait le minimum en envoyant en paramètre le head du heap. On cherche le minimum en parcourant à l'aide d'une boucle while les nœuds de la root liste. Si le minimum se trouve être le head on pose son sibling comme head du heap,
Sinon on élimine directement le min,Enfin on insère ses fils (si il y'en a) dans le root liste à l'aide de BHeapUnion().

APRES EXTRACT-MIN NOUS AVONS LE GRAPHE SUIVANT:

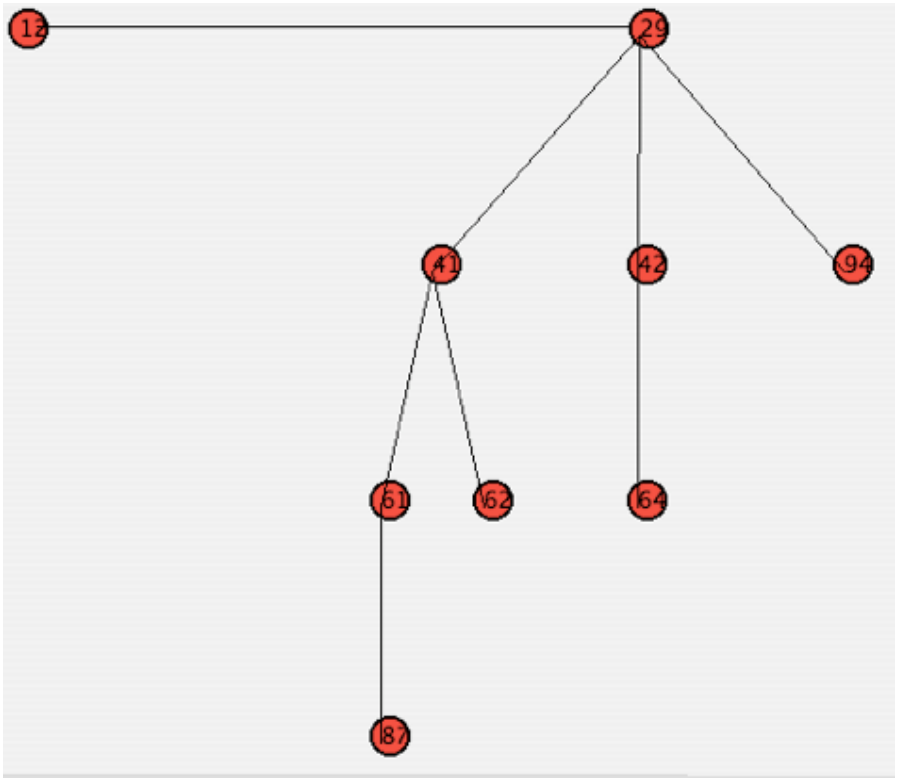


INSERT :

Binomial-Heap-Insert(H,x)
H' := Make-Binomial-Heap()
p[x] := NIL
child[x] := NIL
sibling[x] := NIL
degree[x] := 0
head[H'] := x
H := Binomial-Heap-Union(H,H')

★Explication de l'implémentation :
La méthode insert s'effectue de la façon suivante : on crée un nouveau heap contenant uniquement cet élément que nous fusionnons ensuite avec le heap initial

APRES INSERT NOUS AVONS LE GRAPHE SUIVANT:

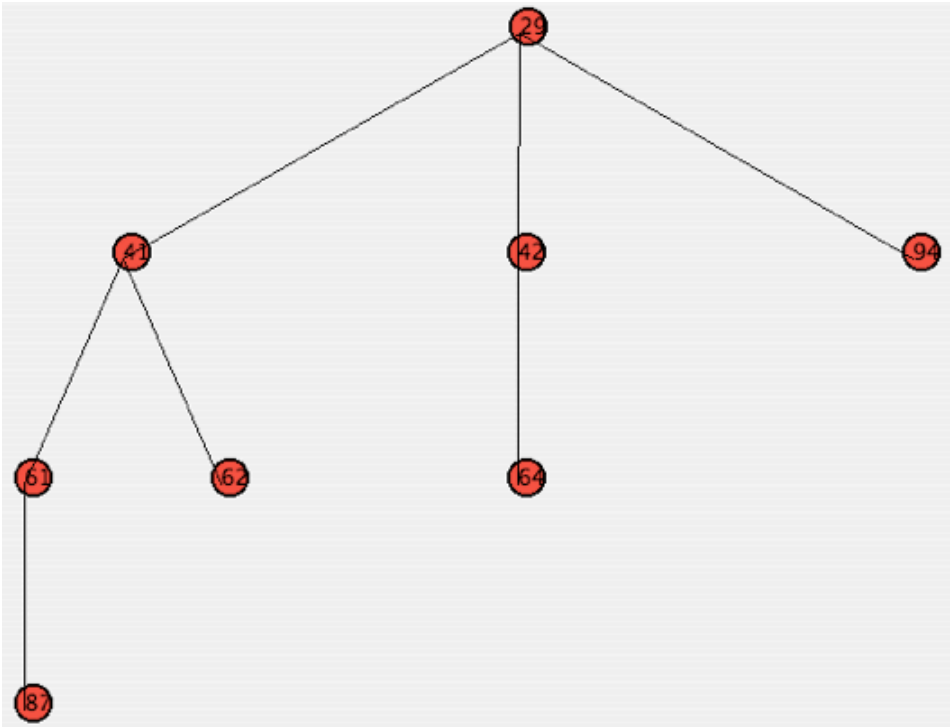


DELETE

Binomial-Heap-Delete(H,x)
Binomial-Heap-Decrease-Key(H,x,-infinity)
Binomial-Heap-Extract-Min(H)

★Explication de l'implémentation :
La méthode delete s'effectue de la manière suivante : on lui donne pour clé moins l'infini (mais, dans notre programme, on lui assigne juste un nombre négatif de l'ordre de -9 ce qui nous suffit, en effet il est plus petite que toute les autres clés...) puis on effectue extrait le min, c'est à dire on l'extrait de la root liste.

APRES DELETE NOUS AVONS LE GRAPHE SUIVANT:


29
41
42
94
61
62
64
87

V. Fibonacci Heap

Comme les Binomial Heaps, les Fibonacci Heaps sont des collections d'arbres. Les arbres dans Fibonacci sont rooted mais ne sont pas ordonnés.

Ils ont été conçu de sorte à respecter des temps constant en tenant compte de l'amortized analysis. Les Binomials Heaps exécutent des opérations comme Insert, Extract-min ou Decrease-Key en O(lg n). Fibonnaci exécute ces mêmes opérations en des temps plus raisonnable en effet toutes les actions ne nécessitant pas la suppression d'un nœud sont exécutées en O(1). Ce qui donne au Fibonacci Heaps un avantage certain.

Le réel apport des Fibonacci heaps ce fait surtout ressentir lorsque les algorithmes nécessitent peu d'opérations impliquant Decrease-Key.

Comme les Binomials Heaps , les opérations se référant à un nœud donné implique l'envoie de ce nœud en paramètre. C'est d'ailleurs pourquoi l'on a implémenter les méthodes VectHeap et VectTree qui nous ont permis un accès simple au nœud désirée.

On va d'abord brièvement décrire les structures qui nous ont permis de comprendre et d'implémenter les différents algorithmes. Puis nous allons présenter ces méthodes une à une en focalisant à la fois sur la partie algorithmique mais aussi sur son implémentation.

5.1 Structures :

Node (fibnode):

Chaque nœud à un pointeur vers son nœud père et un autre vers l'un de ses fils. Le degré d'un nœud indique le nombre de fils. Les fils sont reliés entre eux de façon circulaire et avec un double lien (pointeur vers le nœud à sa droite right et à sa gauche left).
Cette façon de procéder nous permet de supprimer un nœud en O(1) ainsi que de concaténer en un temps constant.
On utilise aussi le champ mark (booléen) qui va nous indiquer si ce nœud à perdu un fils lors d'une opération précédente.

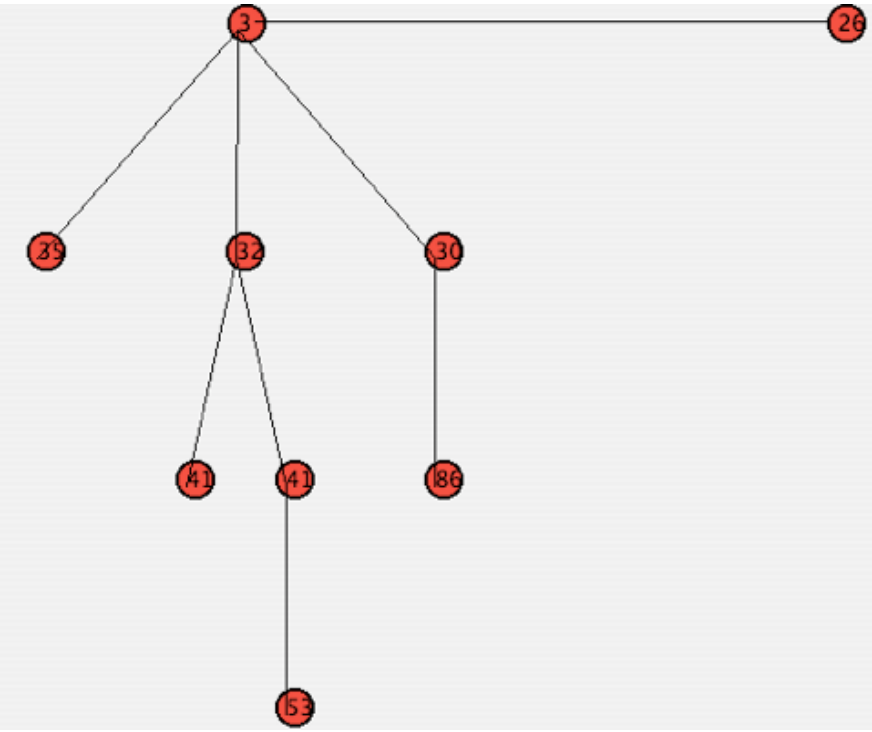
Heap (fibHeap):

Les roots de tous les arbres dans les Fibonacci Heaps sont reliés entre eux , doublement liées de façon circulaire. Cette listes contenant tous les roots est appelée root list . Fibonnaci Heaps ont deux variables. D'abord un pointeur ver le min (H.min), nœud ayant la plus petite Key ainsi que le nombre de nœuds dans ce Heap.

5.2 Algorithmes:

AVEC UN EXEMPLE D'EXCUTION:

HEAP INITIAL:



LES ETAPES:
Etape 1 : extractmin
Etape 2 : decreaseKey node de clef 53 réduit à 9
Etape 3 : delete le noeud de clef 86
Etape 4 : insert node de clef 5.

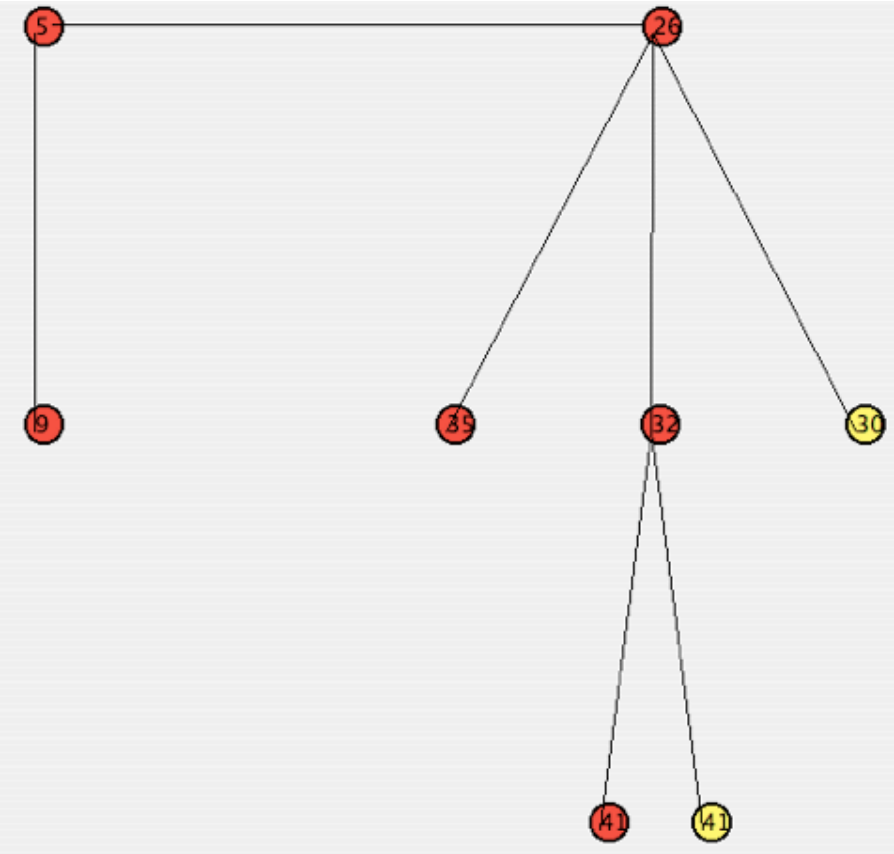
FIB-HEAP-INSERT(H,X):

1 Degree[x]←0;
2 P[x] ← null ;
3 child[x] ←null;
4 left[x]← x;
5 right[x]←x;
6 mark[x] ← false
7 concatenate the root list containing x with root list H
8 if min[h] = null or Key[x]<Key[min[H]]
9 then min[H]←x
10 n[H] ← n[H] + 1

Cette algorithme est assez simple , il ne fait qu'insérer x dans la root list de H. Redéfini le min si nécessaire, et enfin il incrémente le nombre de nœuds dans le Heap.
L'opération insert dans notre implémentation doit être suivi de extractmin qui n'est utile que pour consolider (seulement apres insert).

★Explication de l'implémentation :

Insère le nœud x à la droite du minimum, on incrémente le nombre de nœuds du heap.



FIB-HEAP-UNION (H1,H2):

1 H←MAKE-FIB-HEAP()
2 min[H] ← min[H1]
3 concatenate the root list of H2 With the root list of H.
4 if(min[H1] = null) or min[H2] != nul and min[H2]<min[H1]
5 then min[H] ← min[H2]
6 n[H] ← n[H1]+n[H2]
7 free H1 and H2
8 return

Make-fib-heap crée un Heap , il s'agit ensuite (ligne 3) de créer une root list commune à H et H2. On réactualise ensuite le minimum. Et enfin on incrémente le nombre de nœuds.

★Explication de l'implémentation :

La méthode FibHeapUnion() unis deux fibheaps. Elle fait appel à la méthode concatenate() , méthode qui va unir les deus heaps .

5.2.1 Extracting the minimum node :

FIB-HEAP-EXTRACT-MIN(H):

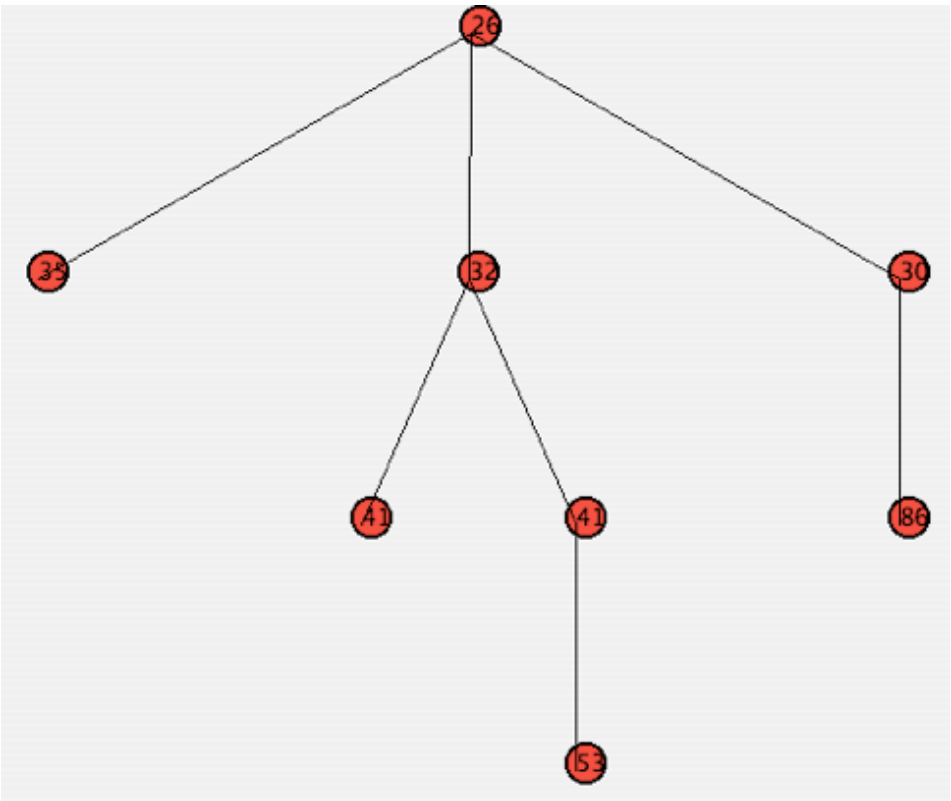
```
1 Z ← min[H]
2 if z != null
3 then for each child x of z
4       do add x to the root list of H
5        p[x] ← null
6       remove z from the root list of H
7        if z = right[z]
8 then min [H] ← null
9 else min[H] ← right[z]
10 consolidate(H)
11 n[H] ← n[H] – 1
12 return z
```

CONSOLIDATE(H):

```
1 for i ← o to D(n[H])
2       do A[i] ← null
3 for each node w in the root list of H
4       do x ← w
5          d ← degrée[x]
6           wile A[d] != null
7                do y ← A[d]
8                  if Key[x]>Key[y]
9                     then Exchange x et y
10                  FIB-HEAP-LINK(H,y,x);
11                  A[d] ←null
12                  d ← d+1
13          A[d]←x
14min[H]←null
15for i←0 to D(n[H])
16      do if A[i]!=null
17          then add A[i] to the root list of H
18               if (min[H] = null or Key[A[i]]<Key[min[H]]
19                 then min[H]←A[i]
```

FIB-HEAP-LINK(H,Y,X)

1 remove y from. The root list of H.
2 maker y a Child of x, incrementing degré[x]
3 mark[y]←FALSE



Apres l'extraction du minimum
Lors de l'apel à EXTRACT-MIN aprés INSERT , cette méthode élabore en premier lieu CONDOLIDATE .
Elle extrait ensuite le minimum .

Extracting the minimum node est l'une des opérations qui nous a paru la plus complexe dans son implémentation .

Elle procède de la manière suivante :

Après l'initialisation , chaque fils de x va être rajouté à la root list.

On va ensuite supprimer le nœud z de H. On actualise le minimum et l'on appelle consolidate.

Consolidate est le cœur de cet algorithme, il va donner une nouvelle hiérarchie à notre heap.

Consolidate procède de manière répétitive, il crée un tableau où les indices vont être interprétés comme des degrés. Par exemple le nœud à la case 5 possède un degré de 5. Toutes les cases vont d'abord être initialisées à nul. Il va ensuite rechercher deux nœuds ayant le même degré dans la root list . Dans la boucle wile si la case correspondant au nœud actuel n'est pas vide (un nœud ayant ce même degré a été inséré).

il va appeler FIB-HEAP-LINK où le nœud de plus grand Key va être mis comme fils de l'autre nœud (ce dernier va bien évidemment être supprimé du root list). Consolidate va ainsi procéder jusqu'à ce que tous les nœuds du root list soient de degré différent.

★Explication de l'implémentation :

La méthode FibHeapextractMin est en elle même assez courte. En effet un appel à FibextractNode va d'abord extraire le nœud minimum (en unissant le nœud à gauche et à droite grâce à fibunion()) . Elle va ensuite faire appel a consolidate().

La méthode consolidate donne une nouvelle hiérarchie à notre heap. Consolidate() initialise un nouveau vecteur de taille Max(degree) des nœuds dans la root list. Chaque nœud du root list va être insérer dans le vecteur dans la case correspondant à son degré. Les indices du vecteur peuvent être vu comme le degré.
Si la case où l'on veut insérer le nœud est déjà occupée(deux nœuds du root list on le même degré) on va fusionner les deux nœuds à l'aide de FibHeapLink() . (le nœud ayant le plus grand degré va devenir le fils de l'autre nœud).
Le nœud ainsi obtenu à un degré incrémenter de 1. Il va être insérer dans le vecteur de la même manière que précédemment.

Les nœuds du nouveau vecteur vont être inséré un à un dans la root list.

La méthode consolidate() va donc nous donné après les modifications des Tree , un Heap où tous les nœuds dans la root list ont un degré différents.

DECREASING A KEY

Fib-heap-decrease-Key(H,x,k):

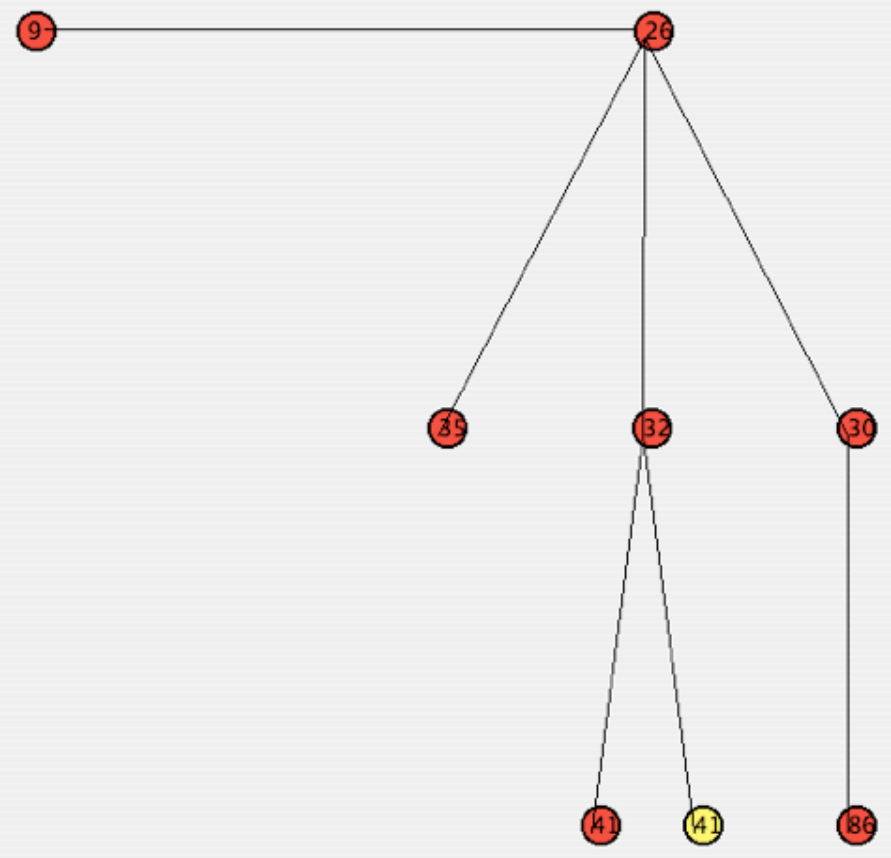
```
1 if k > Key[x]
2         then errer «new Key is greater than current Key  »
3 Key[x]←k
4 y←p[x]
5 if y != nul and Key[x] < Key[y]
6    then Cut(H,x,y)
7            Cascading-Cut(H,y)
8 if Key[x] < Key[min[h]]
9   then min[H]←x
```

CUT(H,x,y):

```
1 remove x from the child list of y,decrementing degree[y]
2 add x to the root list of H
3 p[x] ← nul
4 mark[x]←False
```

Cascading-Cut(H,y):

```
1 z ← p[y]
2 if z != nul
3    then if mark[y] = False
4          then mark[y]←True
5           else Cut(H,y,z)
6         Cascading-cut(H,z)0
```



Après s'être assurer que la nouvelle Key est inférieur au Key actuel. Si le nœud appartiens à la root list où que la valeur du nœud père est inférieur à la valeur introduite , il n'y aura pas de changement.

Si ce n'est pas le cas , des changement vont être opérés, On commence par appeler cut sur x. x va être monter dans la root list et va être détacher de son père.

Le champ mark nous ai d'une très grande utilité dans cette algorithme. En effet tant que marque serra à true les méthode cascading cut et cut vont s'exécuter de manière récursif

★Explication de l'implémentation :

La méthode decerease Key permet de diminuer la Key d'un fibnode donnée pour cela la méthode reçoit en paramètre un fibheap, un fibnode et un entier déterminants la valeur qu'on veut affecter au fibnode envoyer.

Pour résumer le code, on envoie en paramètre le fibnode pour lequel nous voulons diminuer la Key et pour cela on test si, après le changement la Key du père du fibnode diminué est plus petite que celui-ci, si c'est le cas, on élabore un cut entre le père et le fils et ensuite un cascadingcut sur le père.

On évalue enfin si la diminution nous donne un nœud de clef inférieur au minimum si c'est le cas nous modifions le minimum.

La méthode cut :
La méthode cut() prend en paramètre un fibheap, et deux fibnode, x et y. On extrait à l'aide de la méthode fibgetnode() le nœud x et on diminue le degré d'y (qui se trouve être le père de x), enfin, on met x sur la liste des roots à l'aide de la méthode fibAddnode().

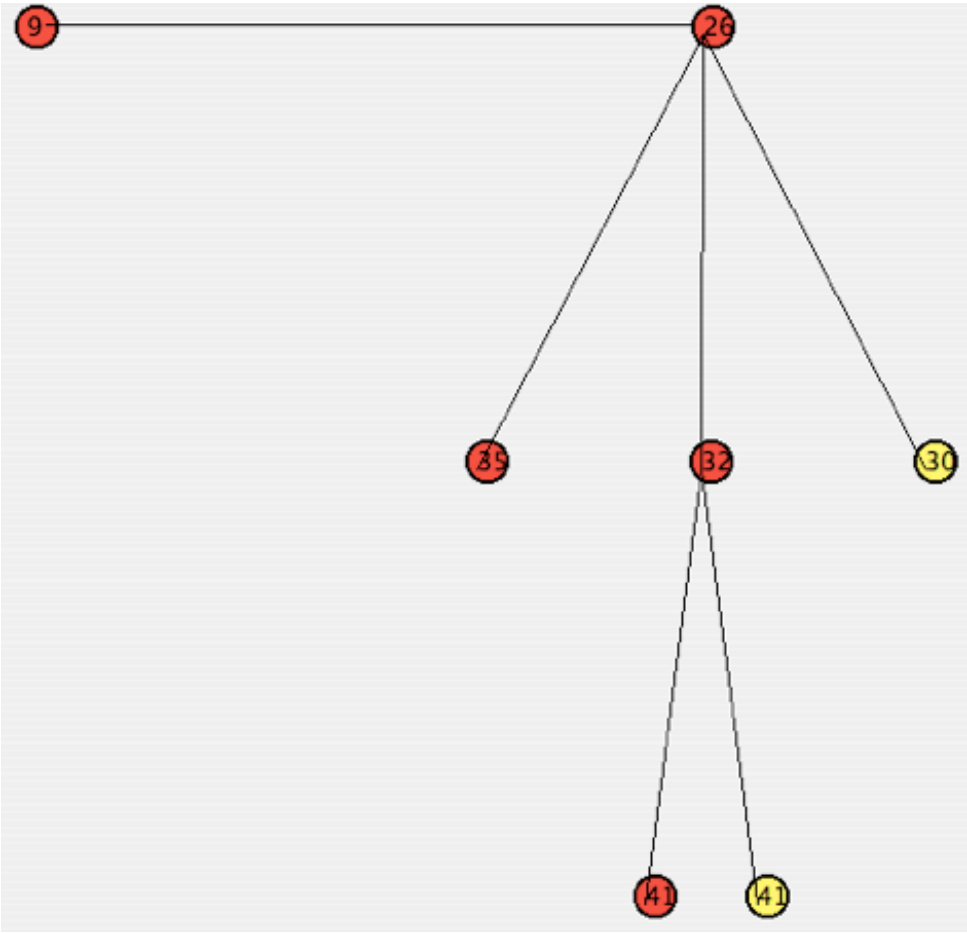
La méthode fibgetnode ;
La méthode fibgetnode() nous permettons d'extraire le nœud envoyé en paramètre mais tout en ne conservant que les liens du nœud extrait avec ses fils.

La méthode fibAddnode :
La méthode fibAddnode() reçoit en paramètre deux fibnode N et Z , celle-ci nous permet d'insérer le nœud N à gauche du nœud Z (lors de l'appelle à cette méthode dans cut(), Z sera notre minimum, c'est-à-dire H.min).

La méthode cascadingCut :
La méthode cascadingCut() reçoit en paramètre un fibheap H ainsi qu'un fibnode y, si le père d'y existe et n'est pas marqué, on le marque si au contraire le père d'y est marqué on élabore un cut entre y est son père ensuite on réitère la méthode en envoyant en paramètre le père d'y.

FIB-HEAP-DELETE(H,X):

1 FIB-HEAP-DEVREASE-KEY(h,X,-INF);
2 FIB-HEAP-EXTRACT-MIN(H);



La méthode VectHeap();

Cette méthodes nous est très utile. Elle va parcourir tous notre heap, pour chaque nœud du root list elle va appeler VectTree() . VectTree() est une méthode qui va de manière récursif parcourir tous les nœuds en les rajoutant dans un vecteur.

VectHeap va donc nous générer un vecteur contenant tous les nœuds des Heaps.

5.3 Amortized analysis

5.3.1 The potential method :

La potential méthode est une méthode qui gère le coût des opérations d'un algorithme. La méthode potentiel est associée avec toute la structure et non avec un seul objet donné de cette structure.
Posons Do l'état initial , Di l'état résultant après la ieme opération et Ci le coût de la ieme opération.

Une potential function phi, transforme une structure de donnée en un nombre réel $\phi(Di)$.
Le coût amorti noté C'î pour la ieme opération est alors donné par :

$C'i = ci + \phi(Di) - \phi(D(i-1))$

Le coût totale va alors être donné par

$$\sum_{i=1}^{n} (\ C'i = \sum_{i=1}^{n(ci)} (ci + \phi(Di) - \phi(D(i-1) \ )$$

Si l'on peut trouver un $\phi(Dn) >= \phi(Do)$, alors la somme ci serra une majoration du coût totale actuel.

Intuitivement si $\phi(Di) - \phi(D(i-1)) > 0$, alors C'i va représenter une surcharge à l'opération i sinon il va représenter une sous-charge. La potential méthode peut être vu comme un apport d'énergie , permettant de subvenir aux opérations à exécuter.
Il faut noter que différentes potential function peuvent être utilisés , aboutissant à des coûts d'amortissement différents certes supérieur au coût réel .

Pour un fibonacci heap on indique par T(H) Le nombre d'arbres, et par m(H) le nombre de nœuds marqués.

Le potentiel d'un fibonacci heap est alors donné par :

$\phi(H) = t(H) + 2m(H)$

Le potentiel de départ est nul , on commence sans heap et ce potentiel ne serra ainsi jamais négatif.

FIB-HEAP-INSERT(H,X):

Supposons que le heap avait t(H) arbres , après insert il aura t(H') = t(H) + 1;
Et il garde le nombre de nœud marqués : m(H') = m(H).
La différence de potentiel serra donc égal à :
((t(H)+1) + 2m(H)) - (t(H)+2m(H)) = 1 .
Comme le coût actuel est de O(1) , le coût amorti serra de O(1)+1 = O(1).

FIB-HEAP-UNION (H1,H2):

$\phi(H) - (\phi(H1) + \phi(H2))$
= (t(H) + 2 m(H)) - ((t(H1)+2m(H1)) + (t(H2)+ 2m(H2))) .
= 0.
t(H) = t(H1) + t(H2) et m(H) = m(H1) + m(H2). Comme l'union ne fait qu'unir les deux heaps.

Le coût amorti est donc : O(1).

FIB-HEAP-EXTRACT-MIN(H):

Un coût de O(D(n)) vient du fait qu'il y a au plus D(n) fils pour le nœud minimum et dans consolidate des deux boucles for. Il reste à analyser la boucle centrale de consolidate. Il faut noter qu'il y a au plus dans la root list D(n) + t(H) - 1, (t(h) de départ plus le nombre de fils du minimum moins le minimum lui même).
Le temps actuel dans extractmin est O(D(n) + t(H)).

Au plus D(n)+1 nœuds restent dans le root, et pas de nœuds en plus ont marqués alors le potentiel est (D(n) + 1) + 2 m(H).

O(D(n) + t(H)) +((D(n)+1) + 2 m(H))) - (t(H) +2 m(H))).
= O(D(n)) + O(t(H)) - t(H)
= O(D(n))

Or D(n) = O(lg n).

Le temps d'amortissement d'extractmin est alors O(lg n).

Fib-heap-decrease-Key(H,x,k):

Le coût d'amortissement de decreaseKey est seulement O(1).
Sans L'exécution des méthodes cascading et cascading cut decreaseKey à un temps de O(1).
Chaque appel de Cascading à un temps d'exécution en O(1), ainsi si il y a c appel, cela va devenir O(c).
La différence de potentiels : Chaque appel a cascading cut va couper un nœud marqué (a part le dernier appel).
Il y aura alors t(h)+c arbres et au plus m(h)-c+2 nœuds marqués.
(c - 1 nœuds de démarqués par les appels cascading cut).
((T(H)+c)+2(m(H)-c+2)) -(t(H)+2m(H)) = 4 - c.
decrease-Key a donc un coût de O(c) + 4 - c = O(1).

VI. Interface graphique

5.1 Généralité :

Lors de l'exécution de notre programme. Une fenêtre proposant divers menu apparaît.
Un premier menu sous " Programme " permet de facilite l'utilisation du programme.
Les menus "BinomialHeap" et "FibonacciMenu" offrent à l'utilisateur divers manières pour accéder
à chacune des méthodes.

" InsérerHeap"/"Insérerfibheap" : Laisse à l'utilisateur le choix du nombre de nœuds ainsi
que de leurs valeurs.

" SampleHeap"/" SamplefibHeap": donne un accès rapide à un exemple déjà créer.

" aleatoireHeap"/"aleatoirefibHeap ": permet la création d'une manière aléatoire des Heaps
, l'utilisateur n'a qu'à choisir le nombre de nœuds qu'il désire.

"Créer2heap"/"Créer2fibsheaps" : création de deux Heaps séparer permettant de vérifier
le bon fonctionnement de la méthode Union.

Dans Bin, la méthode action performed permet de gérer tous les boutons . Elle intéragit avec es classes Fibonacci et Binomial.

5.2 Public class bin extends Jframe implements ActionListener :

La class bin hérite de Jframe . La première méthode afficheMenuBar() est la méthode qui affiche en premier lieu notre fenêtre avec ces différents menus et sous menus.

Il faut noter que notre programme intéragit avec l'utilisateur grâce à des fenêtres Pop-up . Déférentes méthodes askSize() , askNode() , askNodeChoisi() , valeurDek() et afficherVersion() permettent de capturer l'information introduite par l'utilisateur et lui affiche les résultats.

La méthodes actionPerformed() est la méthode qui permet de gérer toute l'interface graphique.

ActionPerformed fait le lien entre l'interface graphique et les class Binomial et Fibonacci . Il s'appuie sur afficher() et fibafficher() qui font appelle aux class Canvasheap et Canvasfheap pour afficher le résultat.

5.3 La méthode action performed :

Comme mentionné avant, l'utilisateur pourra accéder aux fonctionnalités de notre programme de quatre manières.
Après la sélection d'une de ses méthodes un nouveau panel vient remplacer l'image d'accueil. Dans ce nouveau panel on distingue huit boutons, ainsi que le heap souhaité. Ces boutons correspondent aux différentes méthodes associées à chaque clou.

Contrairement aux autres boutons, « Union » ne s'exécute qu'après avoir choisi dans les menus « créer deux heaps » / « créer deux fib-heaps ». Ce bouton fait appel aux méthodes FibHeapUnion () / BHeap Union () des classes fibonnacci et binomial.
Le bouton « Minimum » fait apparaître une fenêtre pop-up avec la valeur du Minimum. Alors qu'elle est immédiate dans le cas du fibonnacci, une boucle sur la root list est nécessaire dans le cas binomial.
Le bouton « Insert » : Une fenêtre pop-up demande à l'utilisateur la valeur du nœud à insérer, les méthodes FibHeapInsert () / BHeapUnion () sont utilisées.

Le bouton extract Min enlève la plus petite valeur du graphe grâce aux méthodes FibHeap ExtraxtMin () / ExtraxtMin ()
Le bouton decrease Key :
Le bouton delete :

5.4 Les arbres dans l'interface graphique :

Les classes canvasfheap ainsi que la classe canvasheap permettent d'implémenter un heap graphiquement. Ces deux classes procèdent de manière identique. La méthode paint() appelle la méthode drawheap.
Dreawheap commence par appeler gethauteurtree méthode récursif permettant d'obtenir la hauteur de notre heap, dans la première boucle while. En plus des appels à gethauteurtree on calcule le nombre de nœuds dans la root list. On obtient ainsi les dimensions de notre graphique. La deuxième boucle while fait appel à drawtree pour chaque node du root list.
Drawtree, s'appuyant sur Node (qui permet réellement de dessiner un nœud), nous permet de façon récursive de dessiner tous les nœuds d'un arbre (Node) et le segment entre père et fils. Lors de l'appel à Drawtree le nœud reçu en paramètre va être dessiné, par la suite un appel à drawtree à tous ces fils va être effectué.
On envoie en paramètre écartLargeur() et les coordonnées du nœud père. Un trait pourra ainsi les relier.

Conclusion

« Relaxed Fibonacci heaps : An alternative to Fibonacci heaps with worst case rather than amortized time bounds »

Les Monceaux de fibonacci relachés offrent les mêmes performance que les Monceaux de Fibonacci et cela au Worst Case.

En effet Chandrasekhar Boyati et C. Pandu Rangan du Indian Institute of Technology on d'abord trouvés une structure qui répondait à ses normes . Ceux-ci présentent donc un nouveau type de monceau offrant des performances intéressantes au pire cas pour la plupart des opérations. Cependant ils n'ont pu obtenir ces même performance pour ExtractMin et Delete.

Gerth Stolting Bordal obtiendra à moins d'un an d'intervalle, une tout autre structure qui atteint des performance en O(log n) aux "Worst Case" pour ces deux opérations.

Q : le monceau
N : Un noeud de type I
M : Un noeud de type II
R : La racine
WN : Le poids du noeud N
P : Un pointeur sur le dernier noeud parent à avoir perdu un enfant. Initialement P = R
wN: L'augmentation de poids de N dû à la perte de son dernier enfant.
n: Le nombre d'enfants du monceau

Le noeud
Le noeud de type I du monceau de Fibonacci relaxé est le suivant :
Élément : La clé du noeud
Type : Le type du noeud (I ou II)
Degré : Le nombre d'enfants de type I de ce noeud
Perdu : Le nombre d'enfants perdus de type I de ce noeud
Un pointeur sur une liste doublement chaînée des enfants de type I
Un pointeur sur un noeud de type II (s'il y en a)

Structures secondaires
Un tableau A : Chaque élément i du tableau pointe sur le premier enfant de degré i de R'
Un tableau B : Chaque élément i du tableau est vrai si et seulement si le nombre d'enfants
de degré i est pair.
Une liste LP : Une liste chaînée des paires de noeuds enfants de R' et de même degré.
Une liste LL : Une liste chaînée des noeuds ayant un champ perdu > 1
Une liste LM : Une liste chaînée de tous les noeuds de type II autre que R'

Ces noeuds possédent aussi des contraintes (voire doc annexe pour plus d'information.)