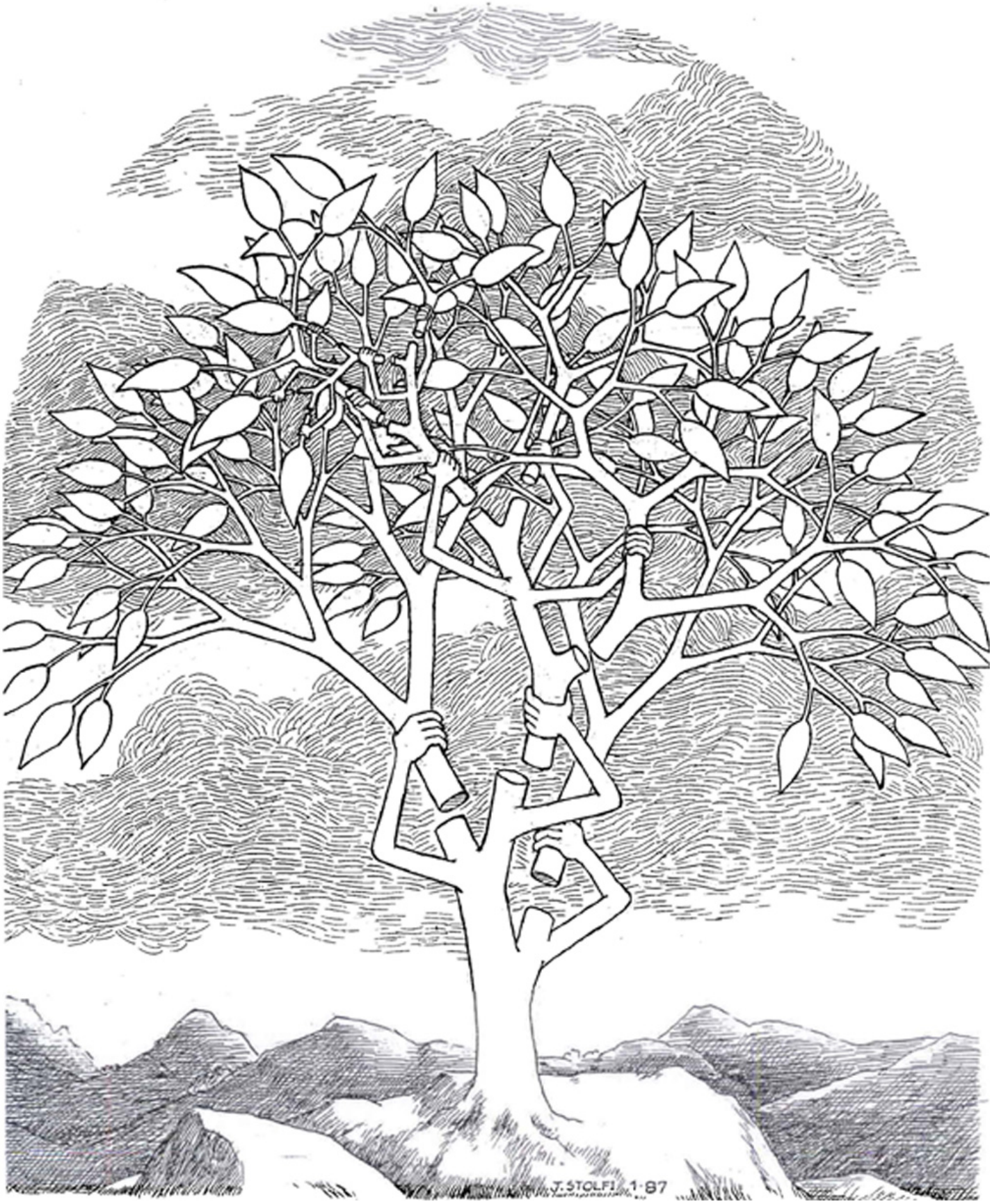
Binomial and Fibonacci heap



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,howevere it figured that they had a better performance in what regards the amortized analysis, These 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 :

Data:
G = (V,E) with V = n and E = m.
V stands for the tops , and E represents the edge with non-négative weights.

Purpose of this work:
Determin the shortest between 2 distinctive points

1.1.1 The DIJKSTRA algorithm:

The algorithm uses the following methodes :

```
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)
```

★This method allows us to elaborate the Dijskistra algorithm

```
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*/
```

★This method Maj_distance allows for a distance update
Between s1 and 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!*/
```

★initialization of the topic.

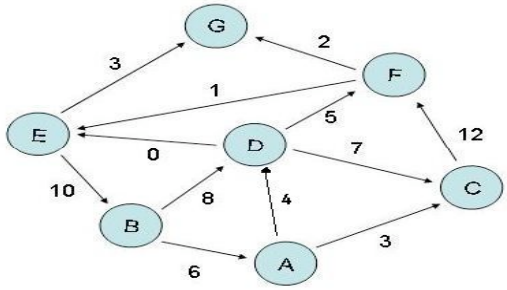
1.1.2 Algorithme cost :

If the intermediary distances are saved in a table , looking for the minimum would cost O(n) and the algorithm has a O(n^2) complexity indépendently from The graph density because all the operations from maj_distance are done on O(n) when the graph is dense.

This complexity is optimal given that all lines must be checked up. If a normal heap is used with a pointer maintained on every node the operation maj_distance would cost O(m.log(n)) and the determination of a summet v with d[v] = min (given that the node is not marked) would cost O(n.log(n)) (which is better for graphs with a low density).

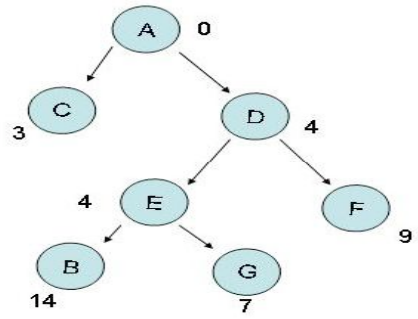
1.1.3 The DIJKSTRA algorithm mecanism:

Let the followong graph



★Let's apply the Dijkistra algorithm starting from point A in order to find the shortest way :

	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



★To obtain the shortest way from A to B, one needs to follow the following way :

A->D->E->B

In our implementation, the OUTPUT consists in showing for every node, the precedent node and the distance that separates them.

1.1.4 Opening:

If a Fibonaci heap is used we would obtain a complexity in
O(n.log(n) + m)

1.2. DIJKSTRA with FIBONACCI :

To sum up , The Dijkstra algorihm consists in finding the shortest way between one node and all the others.
The algorithm is going to be implemented using Fibonaci and more precisely the extractmin() method which only costs O(log(n)).
Three classes were created in order to implement the Djikstra algorithm

- A main class Dijkstra
- A class for the links
- The graph node class

First we create the node and the links. Every link contains a Fibnode that will be put into a fib-heap.
This will help us for the Fibonaci extractmin method.
An integer determining the number of neighbours , The
Key, a distance integer, in wich the minimum distance between this node and the treated one will be stocked.
It also contains a vector of nodes which is made of all the neighboring nodes
And a Graphenode pred détermining the node that precedes in the tree that looks for the minimum.

In order to be clearer, we will start thinking in the following way:
Every node will be filled with distances and a pred in order to find the tree containing the minimum distance.

Therefore using the Fobonaci extramin we reduce the process time
The link class contains an integer determining the weight of the line and a graphnode that determines thenode at the end of the line.
The initialization takes a vector of nodes as parameters and fills it. It is only used for the first step in our algorithm.
The maj_distance method modifies the distance between nodes in a way that keeps the distance minimal.

1.2.1 Functionning :

The distance between the treated node and its neighbours is initialized first.
For this purpose we use initialisation()
A Fibonaci heap with all the nodes fibs is created afterwards.
A While procedure is then implemented in order to assess the number of summets.
The minimum heap already created is then extracted
We check out the summit and all its neighbours using maj_distance

II. The PRIM Algorithm

Prim and dijksta are implemented in the same way but a tiny difference resides.
It is about the distance calculated is not for a source summit but for the tree containing the minimum.
The lines of the prim graph are not oriented at the contrary of that of Dijkstra.
A transformation is then necessary in the maj_distance method. In order to determine
The distance, one only needs to take into account the weight of line studied without additioning the distance the distance to the line.
Another problem appears, it is about the necessity of a verification that the node to be studied is not containes in the vector P that is filled all along the process in irder to avoid cycles

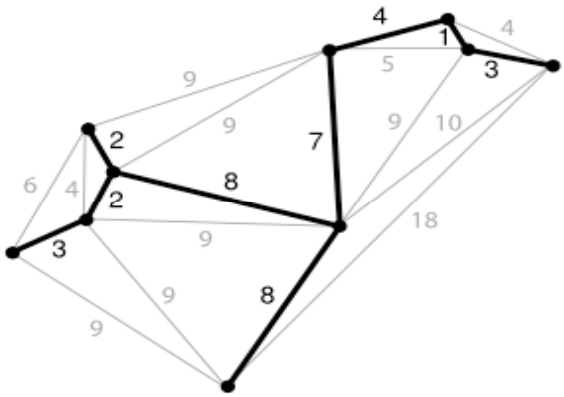
III. L'Algorithme DE KRUSKAL:

The kruskal algorithm consists in determining the tree that contains the minimum.
The lines are ranked with regards to their size. The lines are considered one by one so that no cycles would be triggered.

Each node has an integer "appurtenance" that determines the summits that links it to the lines.

To see whether cycles are created or not , each time a line is incorporated, the related nodes are studied in order to check whether they have the same "appurtenance". If this is true, the line is not incorporated. We also include all nodes from two different trees in a same tree with the same « appartenance ». The process is repeated many times

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



In our implementation, the OUTPUT consists in showing for each considered line, the precedent and the following node, and the distance necessary to get into it.

IV. Binomial heap

A binomial heap is actually a set of binomial trees.
A binomial tree is defined in the following way :
* The o binomial tree is a simple node
* the K binomial tree has a k degrees root and its "childs" are the roots
k-1, k-2, ..., 2, 1, 0 binomial trees (in the presented order ordre)
Binomial heaps are sets of trees. The trees are rooted in an ordered way following the degrees.
Like the fibonaci heaps, the operations refering to a given node implies the sending of the nodes parameters.
And that is why the VectHeap method and vecttree were implemented. That allowed us to have a simple access to the desired node.
A binomial heap is implemented as a set of binomials satisfying the following properties:

* Each tree has an ordered structure: Each node has a key equal or superior to that of its parent.
* For every natiral integer j, there can't be more than one set of binomials which order is j.
The second property implies that a binomial heap containing n elements consists in no more than ln(n+1) binomial trees.
Actually, these trees numbers and orders are determined in a unique way by the n elements.
Each binomial set corresponds to a bite 1 in the binary writing of the figure n.
For example, 13 corresponds to 1101 in binary, $2^3 + 2^2 + 2^0$.
The binomial set with 13 elements will consist in 3 binomial trees which orders will respectively be
3,2,0.
The binomial trees roots are stocked in a list indexed by the trees order.
For a binomial tree Bk, there are 2^k nodes and the height of the tree would be K.
There are (K, i) nodes which depth is i=0,1,2…k

4.1 Structures :

Node :

Each node has a pointer towards the fostering node as well as a pointer towards its siblings and a pointer towards is childs. The node's degree indicates the number of the childs. The latters are related between them in a linear way and with a simple link. Proceeding this way allows us to eliminate a node in O(logn).

Heap :

Each tree's roots in a heap are related between them.
The list contains every root and is called rootlist. The binomial heap is only made of a pointer towards the head. A node that is located at the end of a rootlist. The head is then sent to work on the heap.

4.2 Operation costs :

	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)

To include a new element in a heap, we create a new heap just containing This element that we merge with the initial heap and this in O(ln n).

To find the smallest element in a heap, we need to find the minimum in a rootlist which contains a maximal number of nodes of ln(n). this costs in O(ln n).

To delete the smallest element in the heap, it is sufficient to locate it in the root list .
A list of it childs is then edited and transformed into another binomial heap that we merge with the initial one.

When an element's key is reduced, it becomes smaller than that of it « father »'s keys.
We act that way until the a good order is established in the tree. Each binomial tree having a maximal size ln(n)
the operation will be in O(ln n).

To delete the node, it is given a (-minus infinite) key . In our programme, a negative number is assigned with a -9 order which is enough to extract the minimum. On proceed to the minimum extraction, that is ti say the node itself.

4.3 Les differents algorithme en bref :

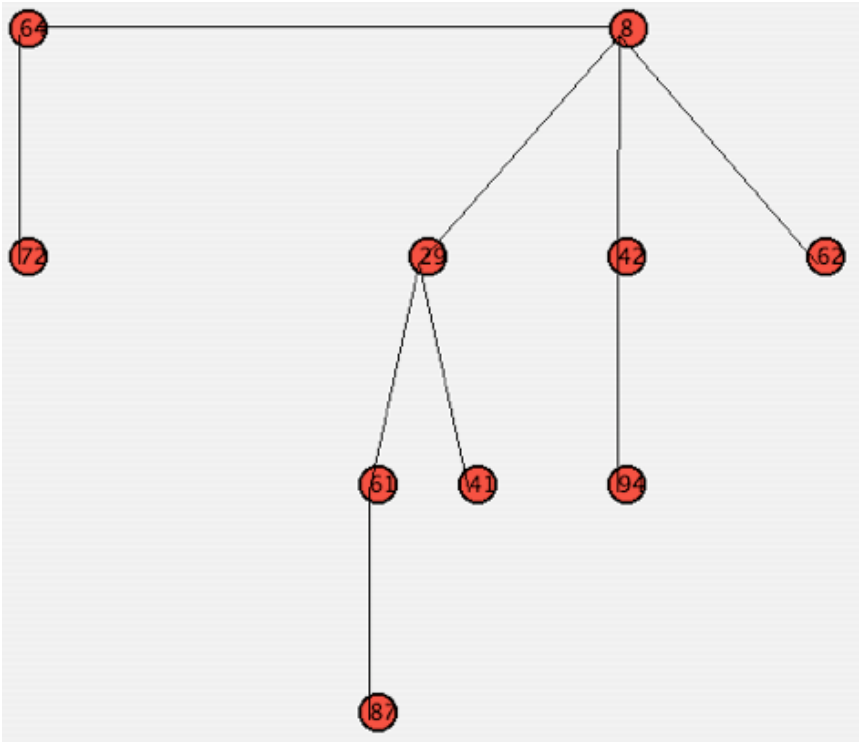
- Make-Heap : creates and returns a new heap which is empty.
- Insert(H,x) : includes the node x in the heap H.
- Minimum(H) : returns a pointer on the node that has the minimal key.
- Extract-Min(H) : deletes the node witha minimal key and returns a pointer on the node.
- Union(H1,H2) : creates et returns a new heap that contains all the nodes of a H1and H2 heap.
  This opration destructs both heaps.
- Decrease-Key(H,x,k) : assigns a new value k to the node s which shound't be greater than the former one.
- Delete(H,x) : deletes the node x from the heap.

4.4 More details:

The VectHeap() method is used to include all nodes in a vector with parameters.
This method receives a node N as a parameter (it figures out to be the head) and a vector from Node vect.
We go through all N neighbours calling the vecttree method each time.
The vecttree method is recursive, it allows us to include all elements from a tree in a vector vect received as a parameter. This method receives an integer I and a node N. We have used a FiFO queue model in order to go through these nodes.

AN EXECUTION EXAMPLE

HEAP INTIAL:---------->



The steps:

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
```

★Accounting for the implementation :

The BHeapUnion() method allows us to merge 2 heaps. It takes 2 nodes H1 and H2 as parameters.
They are the heads of both heaps.
We call the BHeapmerge method.
if the roots have the same degrees we use 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
```

★Accounting for the implementation
The BHeapMerge()
The méthod BHeapMerge() merges the roots lists from the heaps'heads and sends H1 and H2 as parameters in the qame heap.

B-LINK :

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

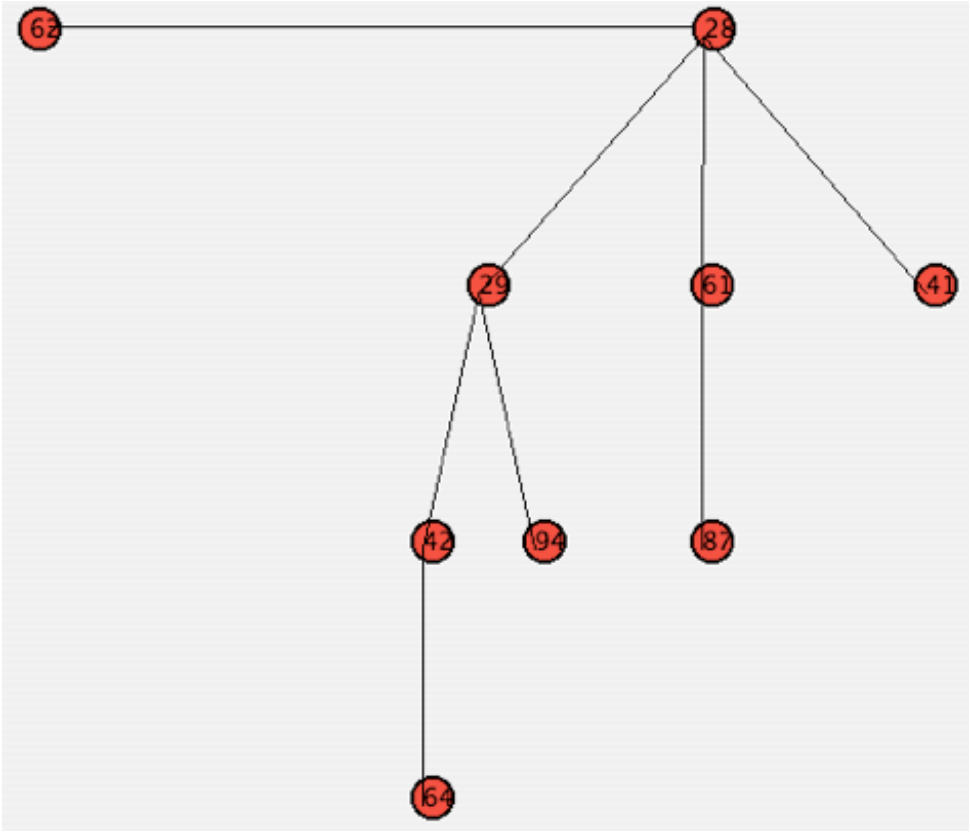
★Accounting for the implementation
Two nodes y and z are set as parameters for the La méthode BHeapMerge() . It consistis in putting z as a father for y while increasing the degree of z.

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]**

★Accounting for the implementation

The node that has to be changed and the desired value are set as a parameter for the BinomialHeapDecreasekey().
To see wether the node is in the heap or not, the head is sent as a parameter. All the nodes are included in the vect using the vectheap. If the node's key is still smaller than that of the father after the reduction, then the keys are exchanged and so on.



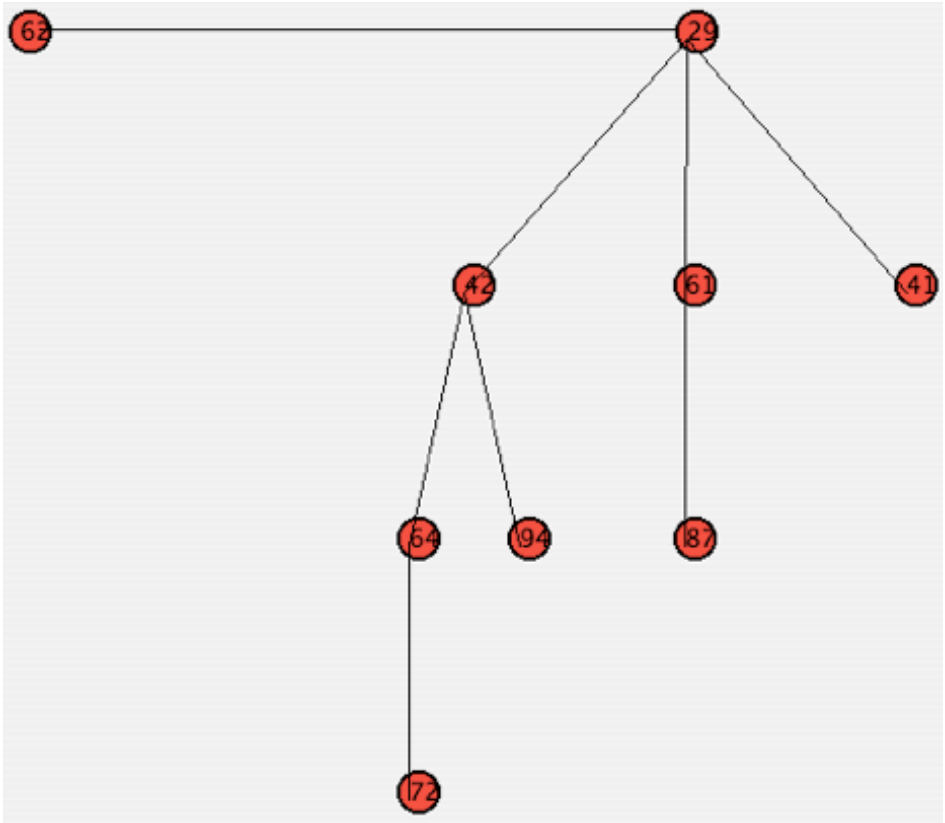
EXTRACT-MIN :

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

★Accounting for the implementation:

The ExtractMin() method extracts the minimum by sending the head of the heap as a parameter.
Using a While procedure we go through the nodes of the root -liste. If the minimum is the head, its sibling is considered as the head of the heap. Otherwise the minimum is directly eliminates. The BHeapUnion() is used to include the sibling (if they exist) in the rootlist

EXTRACT-MIN allows us to have the following graph:

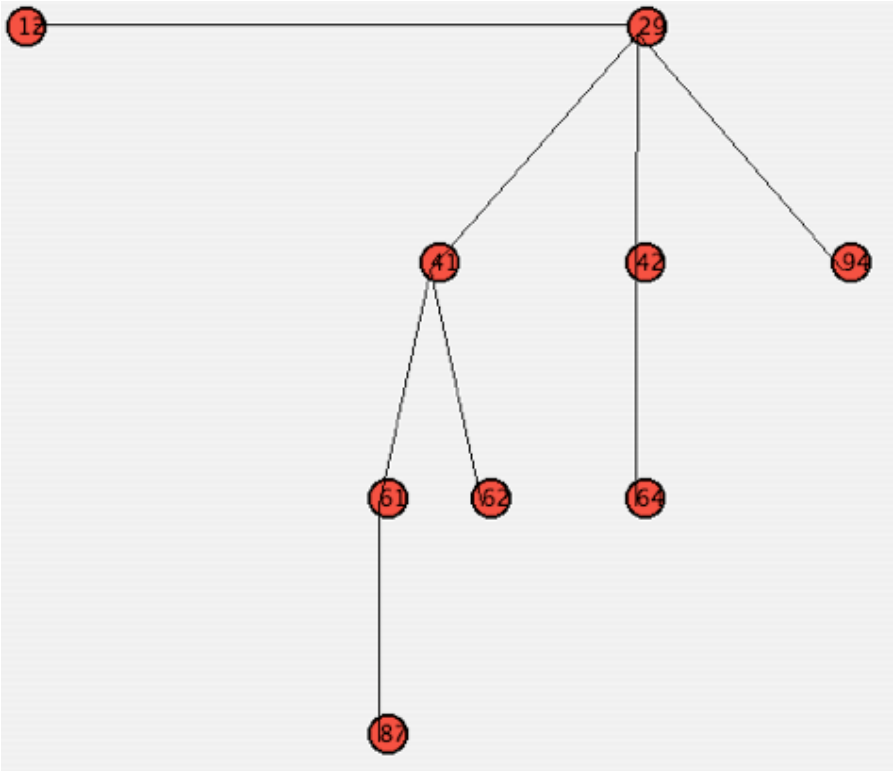


INSERT :

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')

★Accounting for the implementation:
The insert methode works this way : A new heap is created containing an element that is merged with the initial heap.

The Graph comes after the insert function:

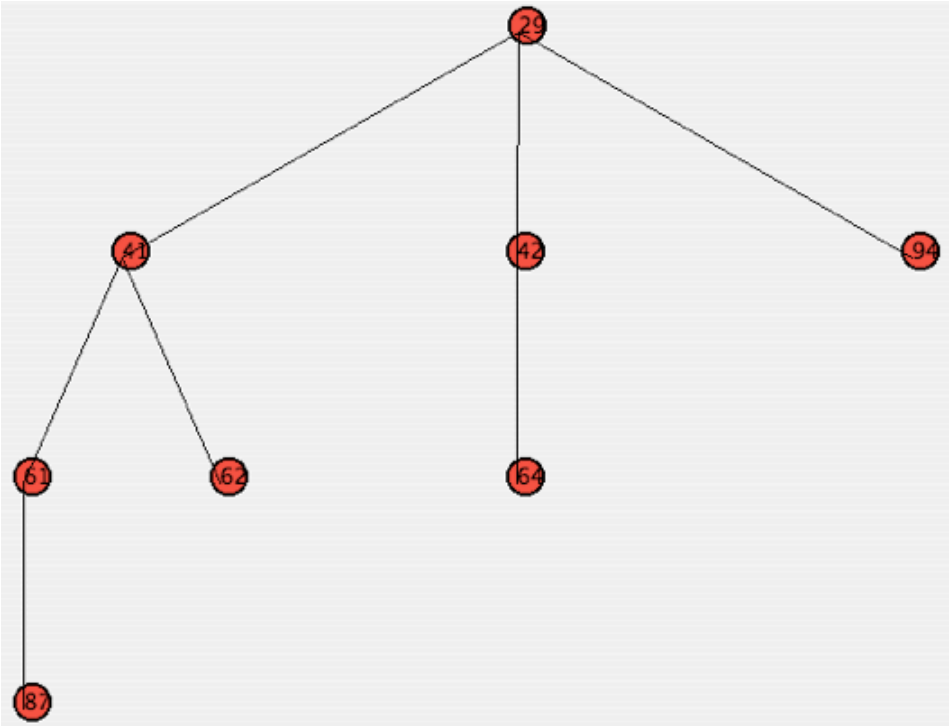


DELETE

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

★Accounting for the implementation:
The delete method goes this way : minus infinite is given as a key (-9 is effectively assigned in the program).
Extract min is then used. It is extracted from the rootliste.

The Graph comes after the delete function:



V. Fibonacci Heap

As for the Binomial heaps, the Fibonaci heaps are tree collection.
The trees are rooted but are not ranked.
They are conceived in order to respect constant times taking into account the amortized analysis.
The binomial heaps execute operation such as Insert, Extract-min or Decrease-Key in O(lg n).
Fibonaci executes these operations in shorter times. All actions that do not need deleting a node are executed in O(1)and that gives to the Fibonaci heaps a real advantage.
What Fibonaci brings is more interesting when the algorithms need few operations using "decrease key".
The operations referring to a given node trigger the sending of this node as a parameter. That is why the methods Vectheap and Vecttree that allowed us to have a simple access to the desired node have been implemented.

The structures that allowed us to understand and implement the different algorithms are going to described briefly.
We will present these methods one by one focusing on the algorithmic part and its implementation.

5.1 Structures :

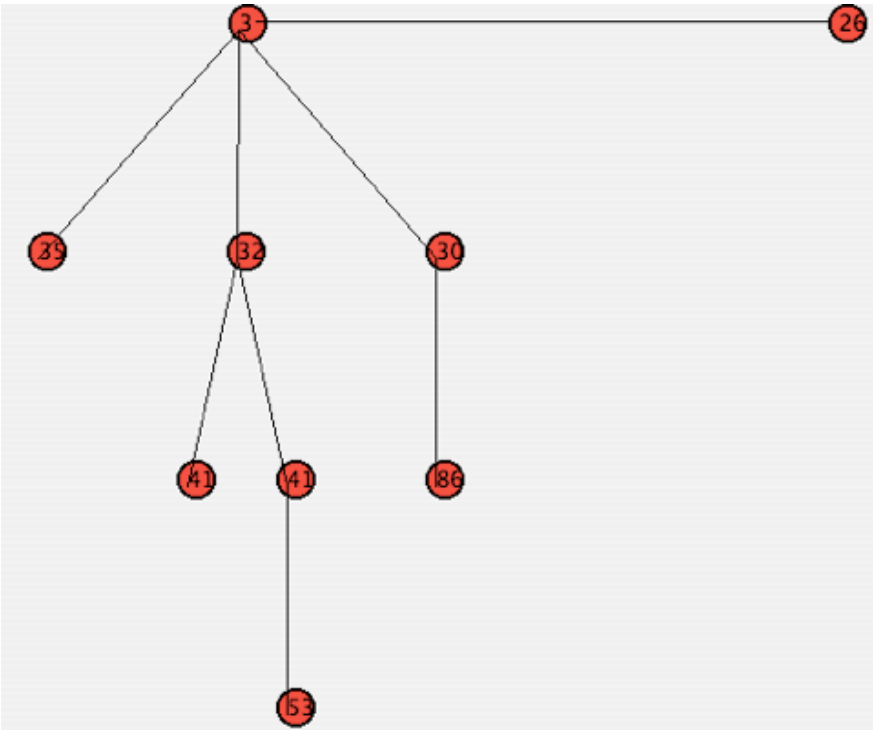
Node (fibnode):
Each node has a pointer towards its father and another toward one of its childs.
The node degree indicates the number of its childs. The siblings are related in a cicular way with a double link.
A pointer towards the node at its right and another at it left. This way of proceeding allows us to delete a node in O(1)
And to append in a constant time.
A boolen mark is also used in order to indicate if the node has lost its childs during the process.

Heap (fibHeap):
All the trees roots are related in the Fibonacci Heaps.
The list containing all the roots is called the rootlist.
The fibonaci heaps have two variables: first a pointer towards min(H, min), a node with the smallest key and the number of nodes in the heap.

5.2 Algorithmes:

AVEC UN EXEMPLE D'EXCUTION:

HEAP INITIAL:

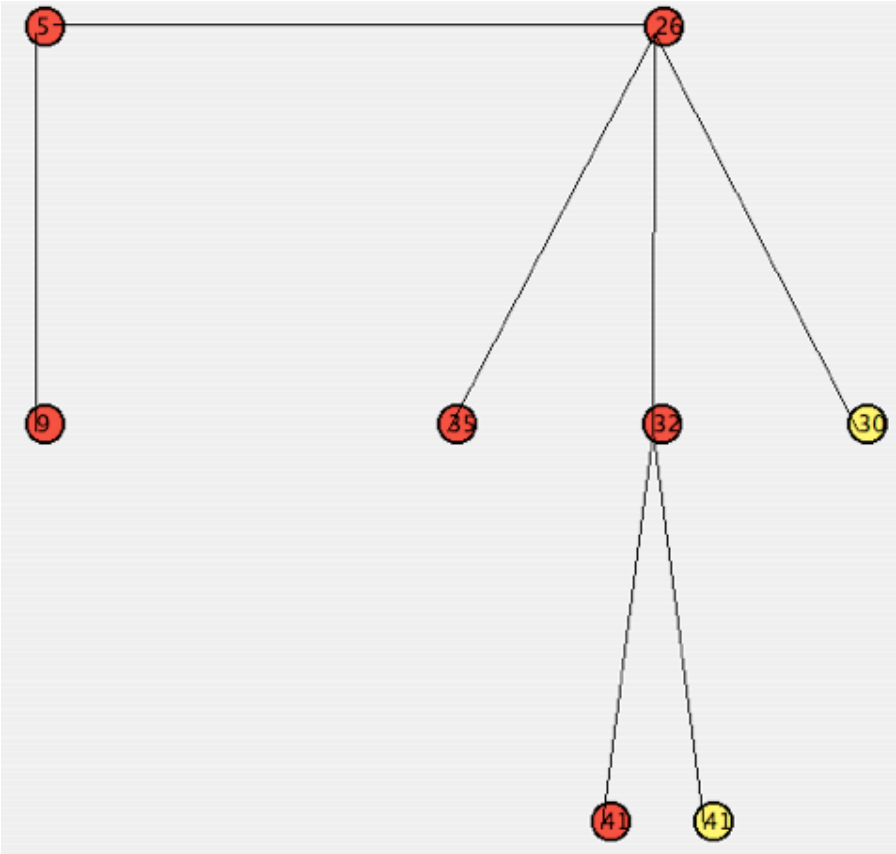


LES ETAPES:
Etape 1 : extractmin
Etape 2 : decreaseKey node de clef 53 réduit à 9
Etape 3 : delete le noeud de clef 86
Etape 4 : insert node de clef 5.

FIB-HEAP-INSERT(H,X):

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

This algorithm is quite simple, it only includes x in the root list H, it redefines the min if necessary, and finaly it increments the number of nodes in the heap.
The operation "insert" has to be followed by extramin in our mplementation:
This is useful for consolidation.
Accounting for the implementation:
Include a node x at the right of the minimum. The number of nodes in the heap is incremented.



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 creates a heap. A rootlist common to H1 and H2 is created then. The minimun is actualised again.

★Accounting for the implementation:

FibHeapUnion() method gathers 2 fibheaps.
It calls the concatenate method that will join both heaps.

5.2.1 Extracting the minimum node :

FIB-HEAP-EXTRACT-MIN(H):

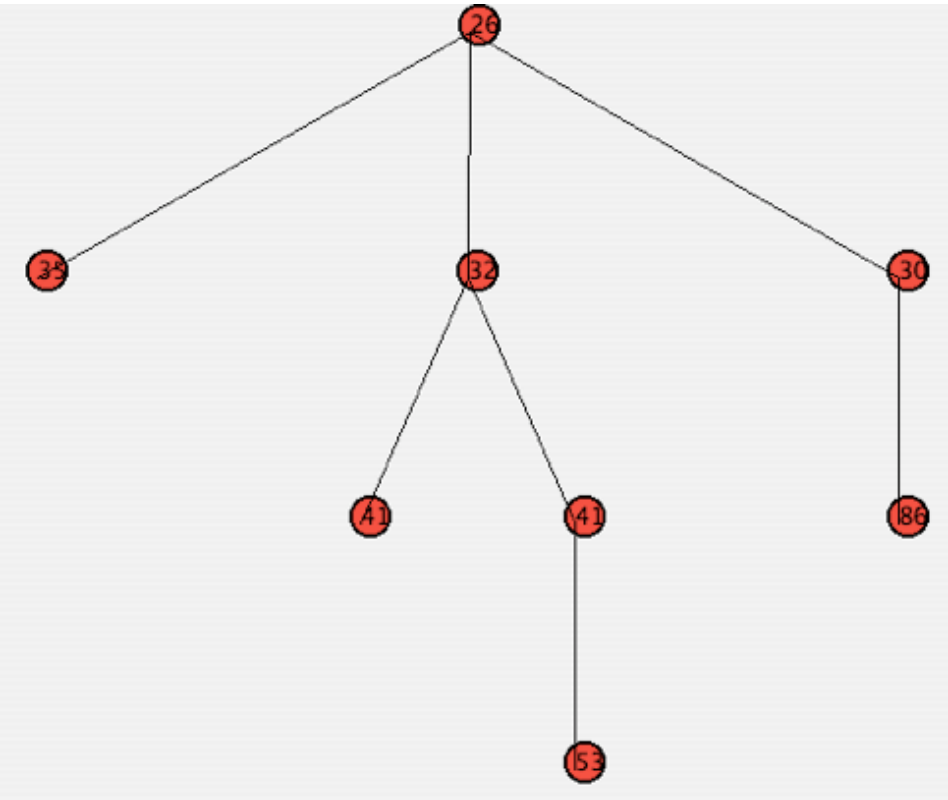
```
1 Z ← min[H]
2 if z != null
3 then for each child x of z
4       do add x to the root list of H
5        p[x] ← null
6       remove z from the root list of H
7        if z = right[z]
8 then min [H] ← null
9 else min[H] ← right[z]
10 consolidate(H)
11 n[H] ← n[H] – 1
12 return z
```

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




```
After extracting the minimum,
while calling Extract min
After Insert the method elaborates “Consolidate” in a first step
then calls “extract min”
```

Extracting the minimum node is one of the most complicated operations in terms of implementation .

It proceeds in the following way :

After the initialization, each x's son will be added to the root list.
The z node from H will be deleted aferwards. The minimum is actualized and "Consolidate" is called.
Consolidate is the heart of this algorithm, it will give the heap a new hierarchy.

Consolidate proceeds in a repetitive way. It creates a table in which all indexes are interpreted as degrees.
For example the node in case five has five degrees.
All the cases are going to be initialized with 0.

Nodes having the same degree are going to be looked for in the root list .
In the while procedure if the case corresponding to the node is not empty (a node having the same degree has been already incorporated) then will FIB-HEAP-LINK is going to be called. The node with the greatest key will be put into it as an childs of the other node (the former will be obviously deleted from the rootlist)
Consolidate will proceed until all nodes from the rootlist have different degrees.

★Accounting for the implementation:

The FibHeapextractMin method is quite short in itself.
First FibextractNode is called for to extract the minimal node.
Fibunion will merge the nodes at the righ and left sides.
Consolidate will then be called. It gives a new hierarchy to the heap.

Consolidate () will initialize a new vector with nodes from the rootlist which size will be max(degree) .
Each node from the rootlist will be included in the vector in the case corresponding to its degree.
The vector indexes will be considered as degrees.
If the case in which the node should be included is already occupied (2 nodes from the rootlist have the same degree). Both nodes will be merged using FibHeapLink().
The node with the greatest degree will be fostered by the other node.
The node obtained thus has an incremented degree of 1.
It will be included in the vector as It has been done before.
The nodes from thenew vector will be included one by one in the root list.
The consolidate method will give a heap in which all nodes in the root list have different degrees

DECREASING A KEY

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

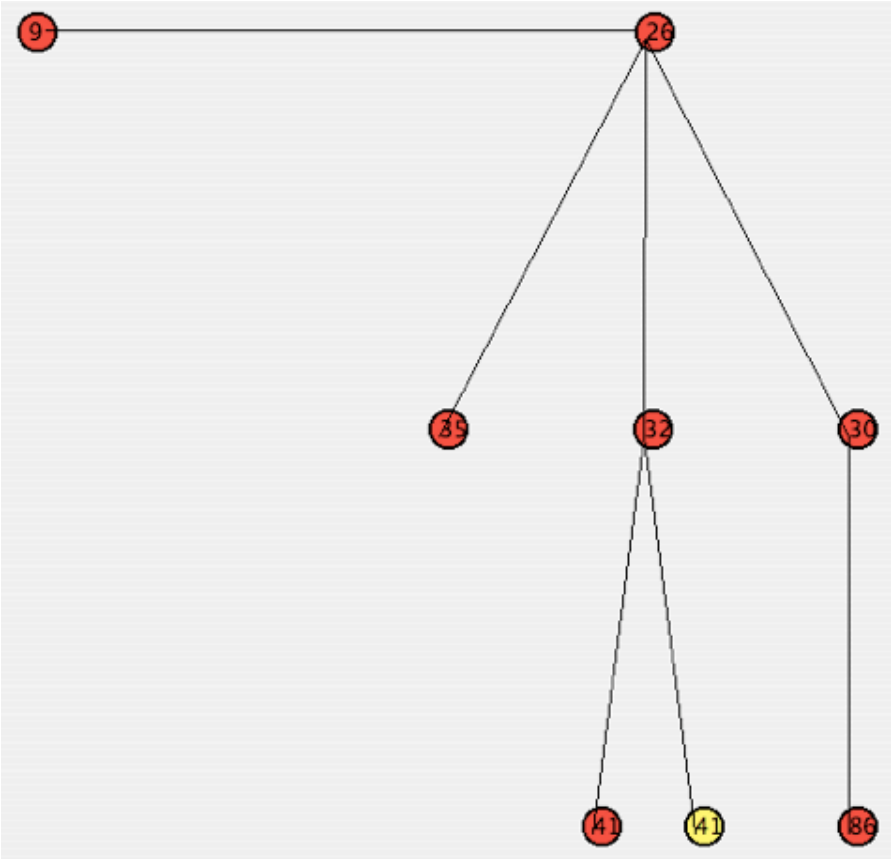
```
1 if k > Key[x]
2         then errer «new Key is greater than current Key  »
3 Key[x]←k
4 y←p[x]
5 if y != nul and Key[x] < Key[y]
6    then Cut(H,x,y)
7             Cascading-Cut(H,y)
8 if Key[x] < Key[min[h]]
9   then min[H]←x
```

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
```



Now that it has been proven that the new key is smaller than the actual one .
There will be no changes when the node belongs to the rootlist or when the value of the node is smaller to the one introduced. Otherwise, there will be changes.
First we call cut on x. X is going to be introduced in the rootlist et is going to be separated from its father.

Le champ mark is really useful for this algorithm. In deed while mark is true the cascading cut method and cut will be executed in a recursive way

★Accounting for the implementation:

The decrease key method allows to reduce the key by a given fibnode. The fibnode is given as a parameter . An integer determining the value desired to the fibnode. To turn the program shorter, the fibnode (the key of which has to be reduced) is sent as a parameter. After changing the father's key that is related to to the reduced fibnode. If it is smaller than the former one, a cut is put between the father and the son and afterwards a cascadingnut is introduced on the father.
We assess wether the reduction gives us a node with a smaller key. If it is true we modify the minimum.

The cut Method:
It takes a fibheap as a parameter et 2 fibnodes x and y. Using the fibgetnode method we extract the x node and reduce the degree of y. (which is x father). At the end x is included in the rootlist using fibbaddnode.

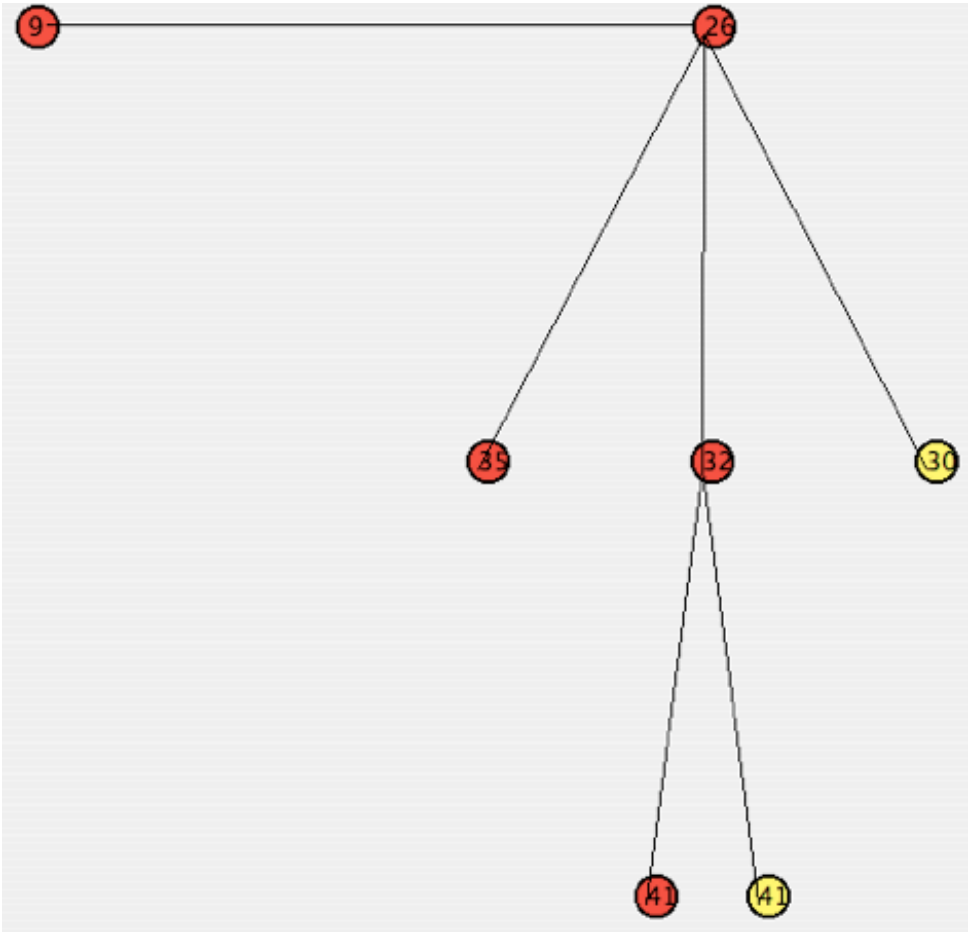
The fibgetnode method:
The Fibgetnode method allows us to extract the node sent as a paramater while keeping the links of the extracted node with its sons.

The Fibaddnode method:
The Fibaddnode method get two fibnodes N and as parameters.
It allows us to include the N node at the left and the Z node at the right
When we call this method in the cut, Z is going to be the minimum that means H min.

The Cascadingcut Method:
The cascadingcut method takes a fibheap H as a parameter as well as a fib-node y.
If y's Father exists and is not marked, it will be marked.
If y's father is marked, we elaborate a cut between y and its father.
The method is repeated while sending y's father.

FIB-HEAP-DELETE(H,X):

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



La méthode VectHeap();

This method is very helpful. It will go through the heap. For each node from the rootlist, it is going to call VectTree.
VectTree() is a method taht is going through all nodes in a recursive way adding it into a vector.
VectHeap is going to generate a vector containing all nodes from the heaps.

5.3 Amortized analysis

5.3.1 The potential method :

The potential method manages the algorithm operation cost . It is related to the whole structure et not to a simple object of that structure.
Let D_0 stand for the intial state, Di Stands for the state resulting of the ith operation and Ci the cost of the ith operation.
A potential function $\phi()$ turns a data structure in a real number $\phi(Di)$
The amortized cost C'i for the ith operation will then be:

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

And the total cost :

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

if there is $\phi(Dn) >= \phi(Do)$, then the sum ci will be greater thanthe actual total cost.
Intuitively if $\phi(Di) - \phi(D(i-1)) > 0$, then C'i will represent an overwight for the operation i. Otherwise it will represent an underweight.
The potential method can be considered as a gain in effort allowing to come to the needs of the operations that need to be executed. Different potential methods can be used that will lead to different costs of execution, No doubt greater than the real one.

T(h) stands for the number of trees in a Fibonacci heap. m(H) the number of marked nodes.

A fibonacci heap potential is given as :

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

The departure potential is null. There is no heap at the beginning and the potential will never be negative.

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. In the root list there is at most (D(n) + t(h) - 1) nodes , t(H) from the initial tree and we add the minimum's number of children minus the minimum itself.
The actual time in extractmin is O(D(n) + t(H)).

At most D(n)+1 modes will be kept in the root,
No other nodes are marked and the potential will then be
(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).

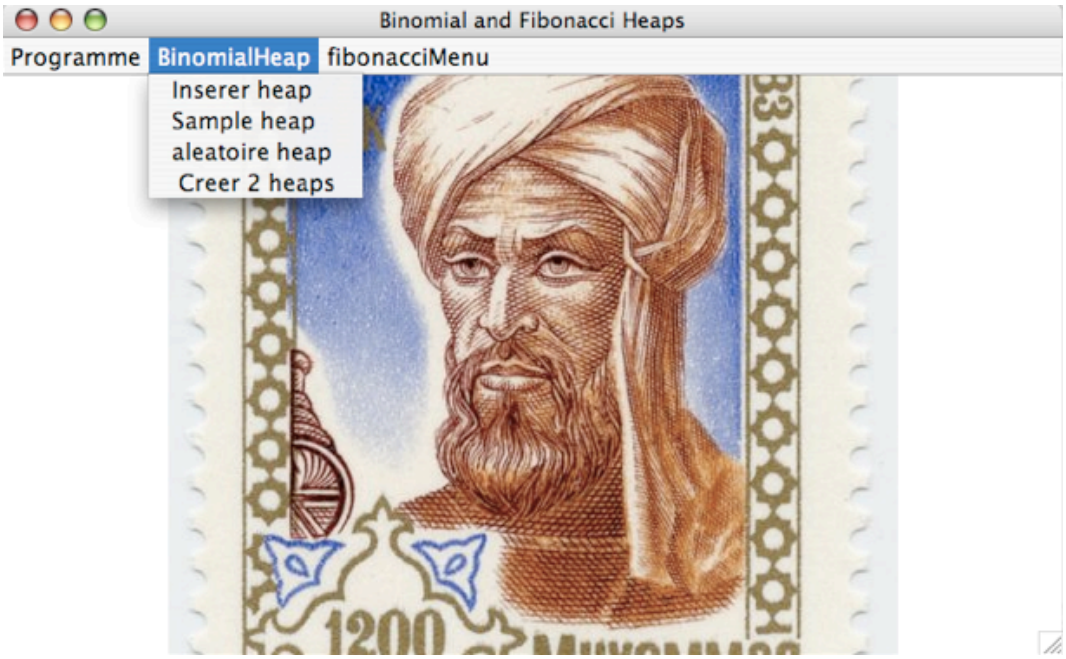
The extractmin time is then going to be O(lg n).

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

The decreaseKey amotizing cost will only be O(1).
Without executing the cascading method and cascading cut decreaseKey at a O(1). Time
Ithe cascading call has a O(1), execution time. If there are c calls it going to be O(c).
The potential difference: Each call of a cascading cut is going to cut a marked node (except the last call)
There will be t(h)+c trees and at most m(h)-c+2 marked nodes.

(c - 1 demarked nodes from the cascading cut call).
((T(H) + c) + 2(m(H) - c + 2)) -(t(H) + 2m(H)) = 4 - c.
then the decrease-Key has a cost of O(c) + 4 - c = O(1).

VI. Interface graphique



5.1 Généralité :

During the program execution a window with different menus appears. First menu under « Programme » makes the use of the program easier. The "BinomialHeap" and "FibonacciMenu" offer to the user different ways to get in.
For each method

" InsérerHeap"/"Insérerfibheap" gives the user the choice of the number of nodes and the values wanted.

" SampleHeap"/" SamplefibHeap": allows a quick access an example that has already been created

" aleatoireHeap"/"aleatoirefibHeap ": allows the creation of a hazardus way for the heaps.
The user has to choose the number of nodes desired.

"Créer2heap"/"Créer2fibsheaps" : creates 2 separate heaps which allows for a verification of the well functioning of the union method
In bin, the action performed method allows to manages all the buttons.
It interatcts with the Fibonaci and binomial classes.

5.2 Public class bin extends Jframe implements ActionListener :

The bin class inherits from Jframe.
The afficheMenuBar() method shows the window with its different menus and submenus.

Our programme interacts with the user thanks to a pop up window.
Different methods méthodes askSize() , askNode() , askNodeChoisi() , valeurDek() et afficherVersion() allow to capture the information introduced by the user and shows the results.
The actionPerformed() method allows to manage all the graphic interface.
ActionPerformed makes the link between the graphic interface and the Binomial Fibonacci classes . It stands on afficher() and fibafficher() which call Canvasheap and Canvasfheap to show the results.

5.3 The action performed method:

As it has been mentionned above, the user will be able to access to the programme functionalities in 4 different ways.
After the selection of one the methods a new panel will replace the welcome image.
Eight buttons can be seen and a desired heap. These buttons coorespond to different methods related to each nail.
At the contrary of the "Union" can only be executed after choosing « créer deux heaps » / « créer deux fib-heaps ». in the menu. From the fibonnacci et binomial classes
The minimum button shows a pop up window with the minimum value.
This immediate in the Fibonaci case but needs a rootlist in the other case.
A pop up window asks the user the value of the node he wants to include; the FibHeapInsert () / BHeapUnion () methods are used.
The "extract Min button" takes rthe smallest values out of the graph thanks to FibHeap ExtraxtMin () / ExtraxtMin ()
Le bouton decrease Key :
Le bouton delete :

5.4 The trees in the graphic interface:

The canvasfheap and canvasheap classes allow to implement a heap graphicly.
Both classes proceed in the same way.
The paint method calls the drawheap method.
Dreawheap first calls the gethauteurtree, a recursive method that allows to obtain the hight in the heap. It is done at the level of the first while procedure.
The dimensions of the graph are thus obtained. The second while procedure calls drawtree for every node in the rootlist.
Drawtree is based on node (which draws a node). It allows to draw the nodes in the tree and a line between the father an dits offspring. When Drawtree is called the node taken as a parameter will be drawn . The same procedure will be done for the offspring
Ecart largeur is sent as a parameter and the father's node coordinates. They will be linked by a line

Conclusion

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

The relaxed fibonaci heap have in the worst case the same performance as the Fibonaci heaps

Chandrasekhar Boyati et C. Pandu Rangan from Indian Institute of Technology have first found a structure responding to their norms
. they present a new type of heaps with a better performance for each operation.
Neverthe less they could'nt have these performances for ExtractMin et Delete.
Less than one year later Gerth Stolting Bordal will find a totally different structure that reaches performances in en O(log n) in the wors cases for all operations

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.)

We would like to thank Mr Mahdi Cheraghchi and the professor Mr Shokrollahi.