Decrease Key and Delete Node Operations on a Fibonacci Heap

Alright, let's dive into two important things you can do with a Fibonacci heap:

1. **Decrease a key**: This is like saying, "Hey, let's make this number smaller!" You have a number in your heap, and you want to make it even smaller. For example, if you have a list of scores in a game and you find a way to improve your score, you'd use this operation to update it.

2. **Delete a node**: Sometimes, you want to remove a particular thing from your heap. Maybe you're playing a game, and you want to remove a player from the scoreboard. This operation lets you do just that. You point to the thing you want to get rid of, and poof, it's gone.

Decreasing a Key

Alright, let's understand how to decrease a key in a Fibonacci heap. Imagine you have a collection of numbers, and you want to make one of those numbers smaller. Here's how you do it:

1. **Decrease-Key**: First, you pick the specific number you want to make smaller. Let's call this number "x," and you change its value to the new, smaller value, which we'll call "k."
2. If "x" has a parent (that's like saying it's not an only child), and if its parent's value is greater than the new value "k" you've given to "x," then you do two things:
    - You perform a **"Cut"** operation on "x," which basically means you detach it from its current position and make it a separate tree.
    - You also perform a **"Cascading-Cut"** operation on the parent of "x." This helps clean things up if the parent has lost a child.
3. Lastly, if the value of "x" after decreasing it is smaller than the current minimum value in your heap, you update the minimum value to be "x."

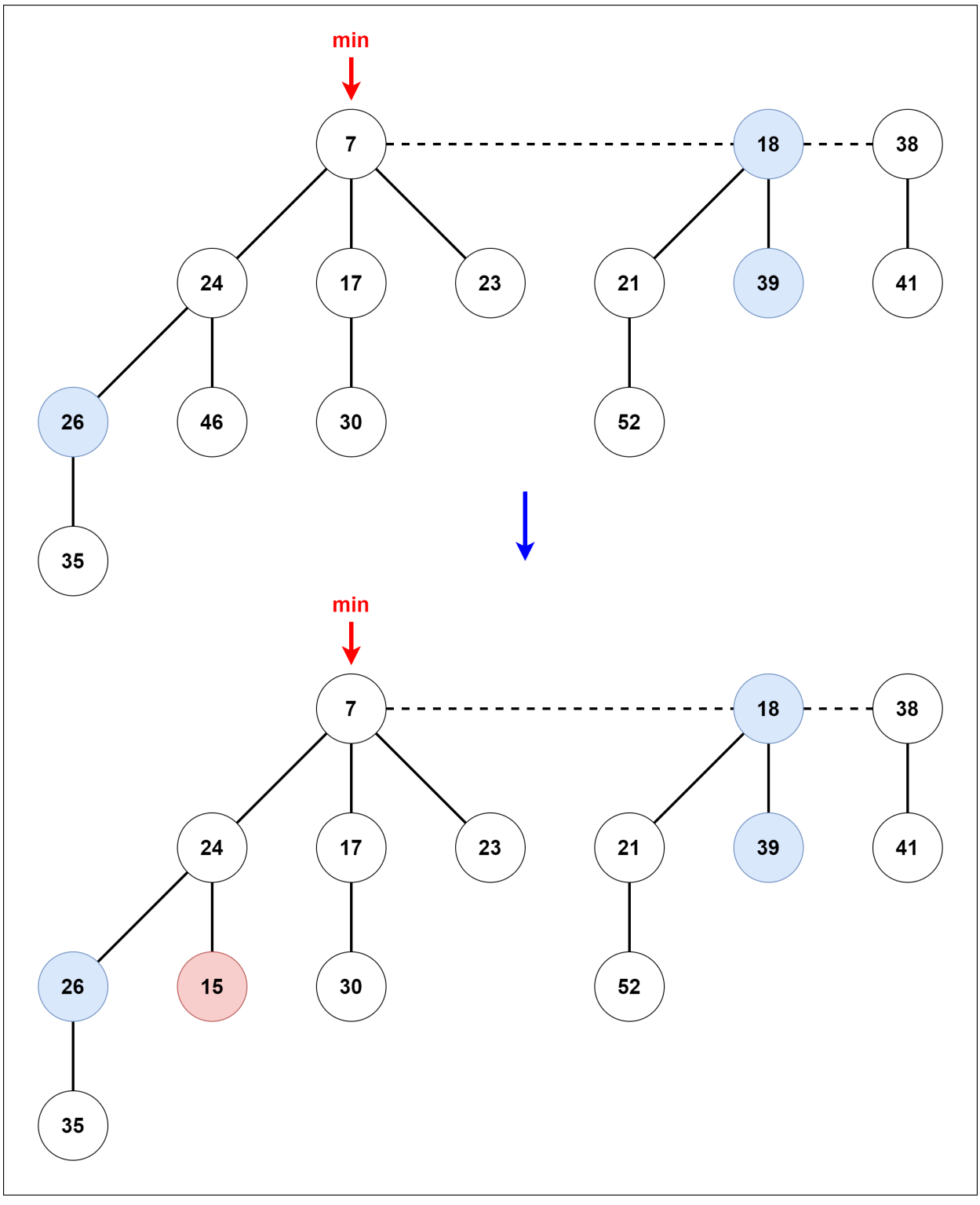
So, in simple terms, "Decrease-Key" allows you to make a number smaller in your heap, and it also helps maintain the structure of the heap properly.

Decrease Key Example

Let's take a practical example to understand how to decrease a key in a Fibonacci heap:

Example: Decreasing 46 to 15.

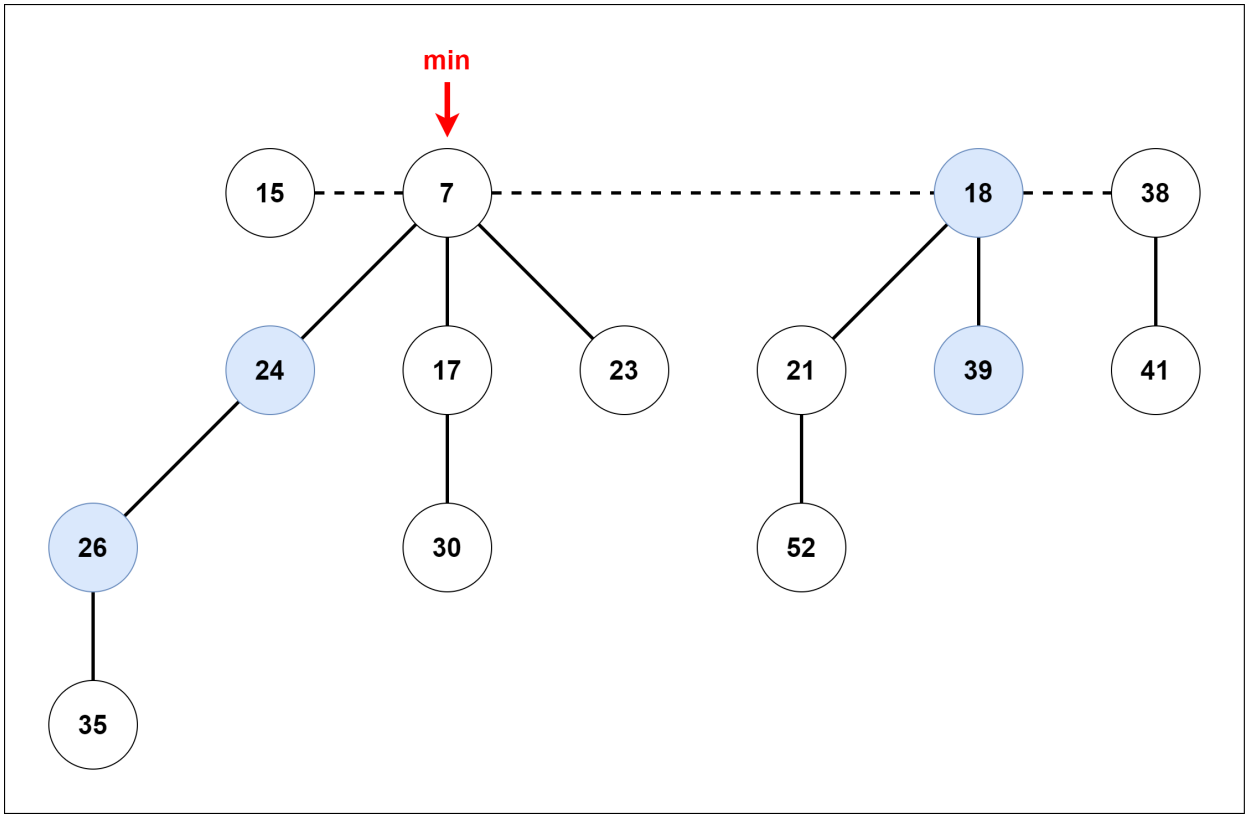
1. First, we have our heap with various numbers. We want to decrease the value of 46 to 15.



2. We perform the "Decrease-Key" operation by changing the value of 46 to 15.

Now, let's get into the details of the operations involved:

- **Cut Part**: Since the parent of the node containing 46 is not null (that means it has a parent), and the value 15 is less than the value of its parent (which is 24), we perform a "Cut" operation. This means we remove the node containing 46 from its current position in the tree and add it to the list of trees at the top level, which we call the root list.

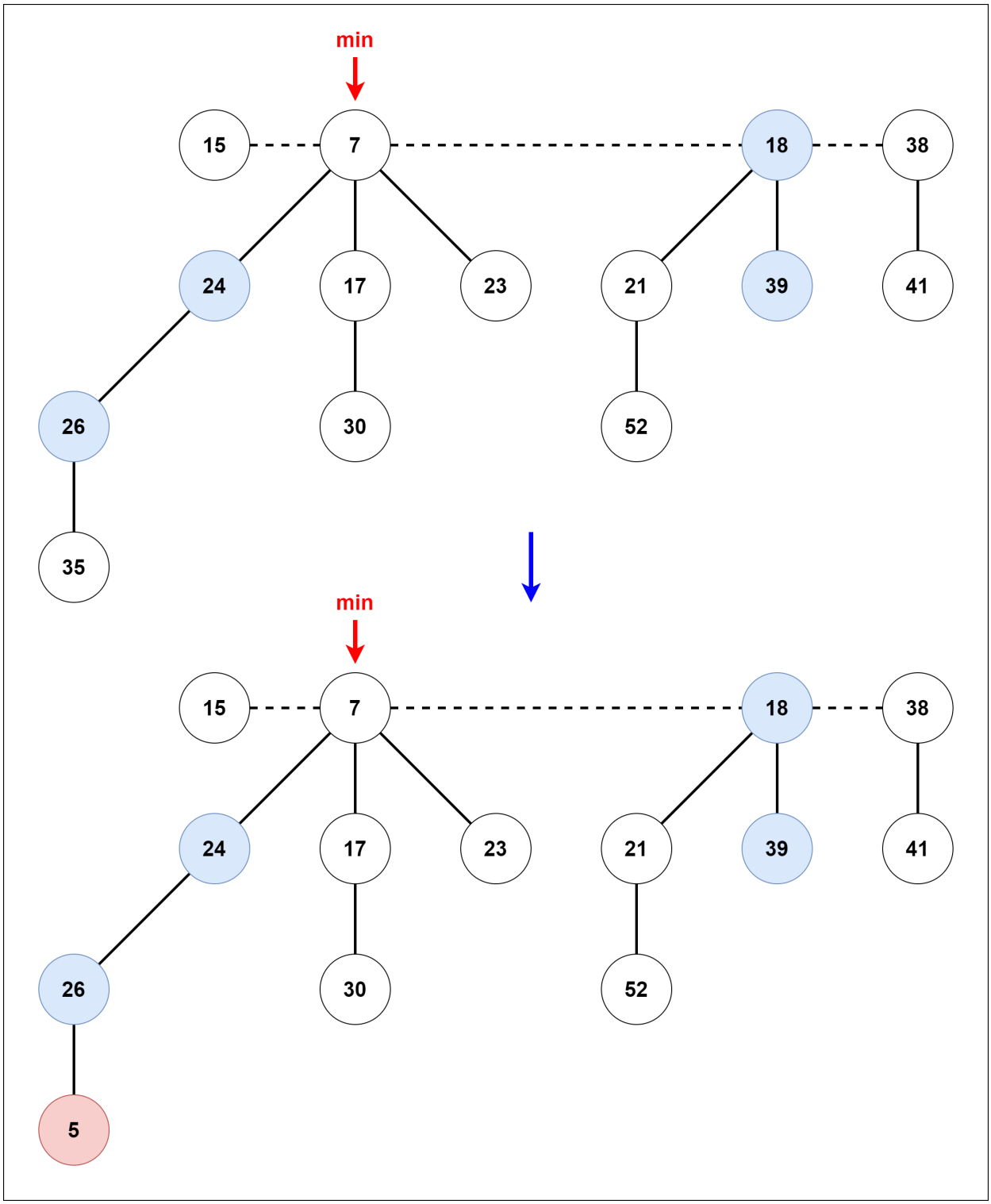


- **Cascading-Cut Part**: Additionally, we mark the parent, which in this case is the node containing 24. This is done because if a marked node loses a child, it becomes marked itself. If it loses another child, it gets cut as well. This process continues recursively up the tree until an unmarked parent is found.

So, in our example, we've not only decreased the key from 46 to 15 but also adjusted the structure of our Fibonacci heap by cutting and marking nodes where necessary.

Example: Decreasing 35 to 5

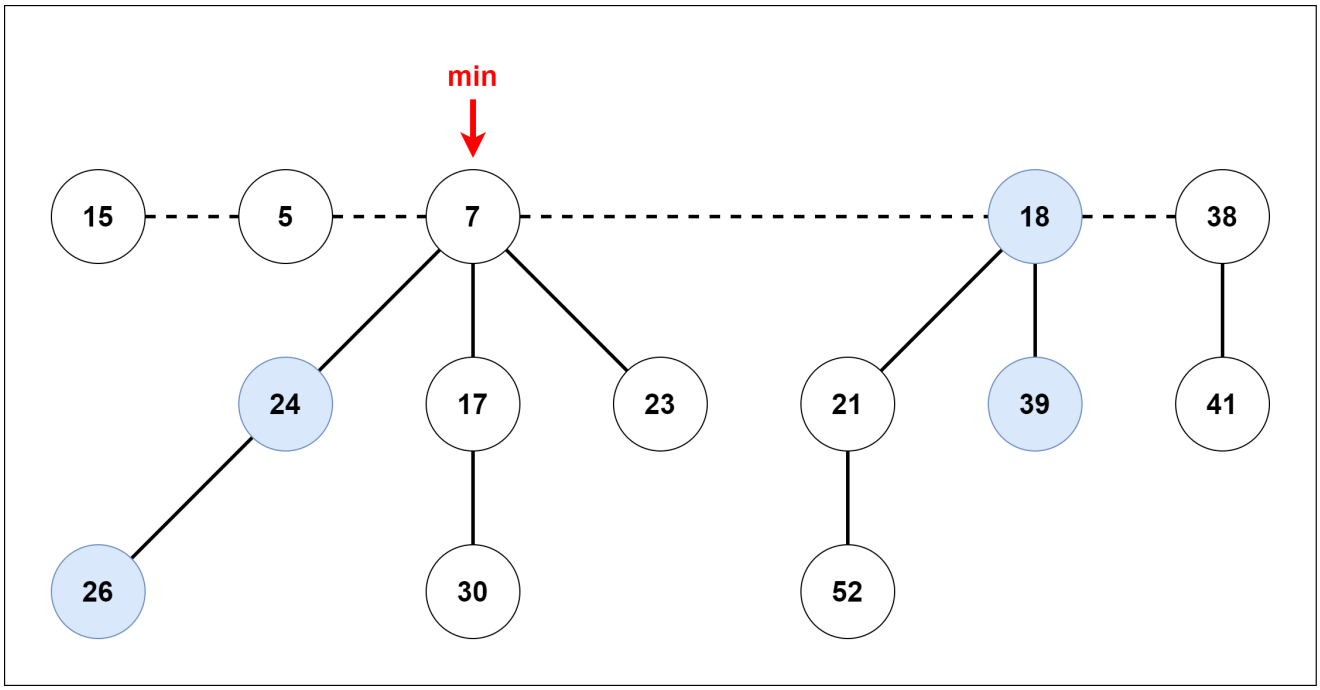
1. We start with our Fibonacci heap, and we want to decrease the value of 35 to 5.



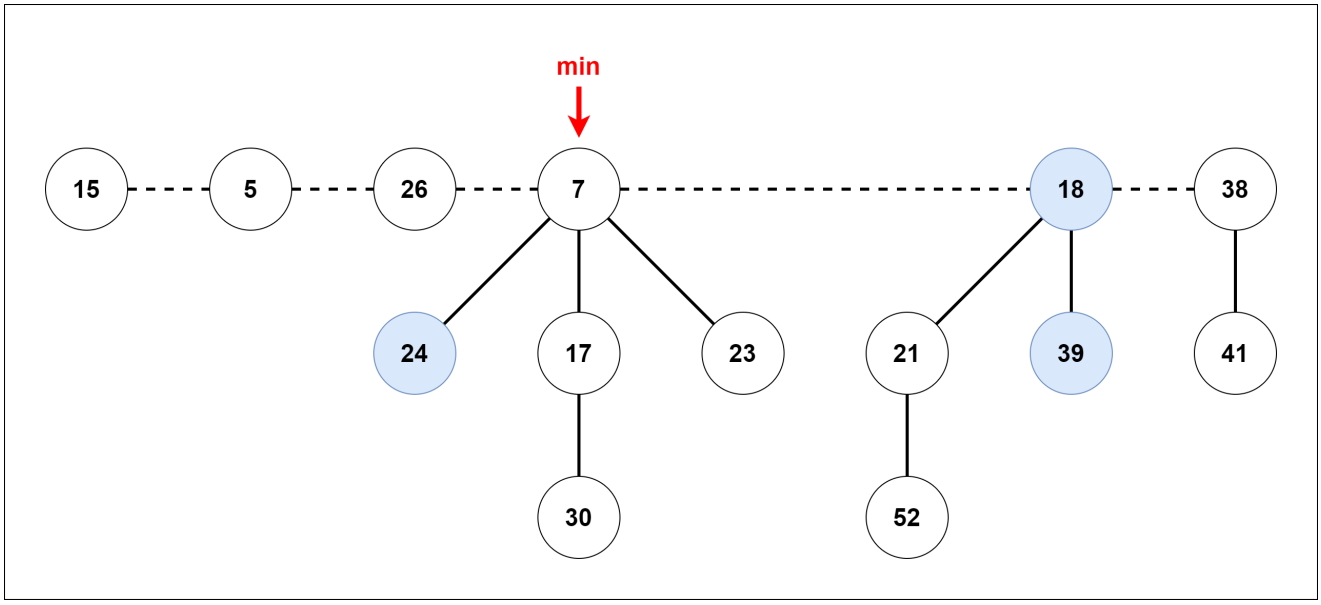
2. We perform the "Decrease-Key" operation by changing the value of 35 to 5.

Now, let's understand the steps involved:

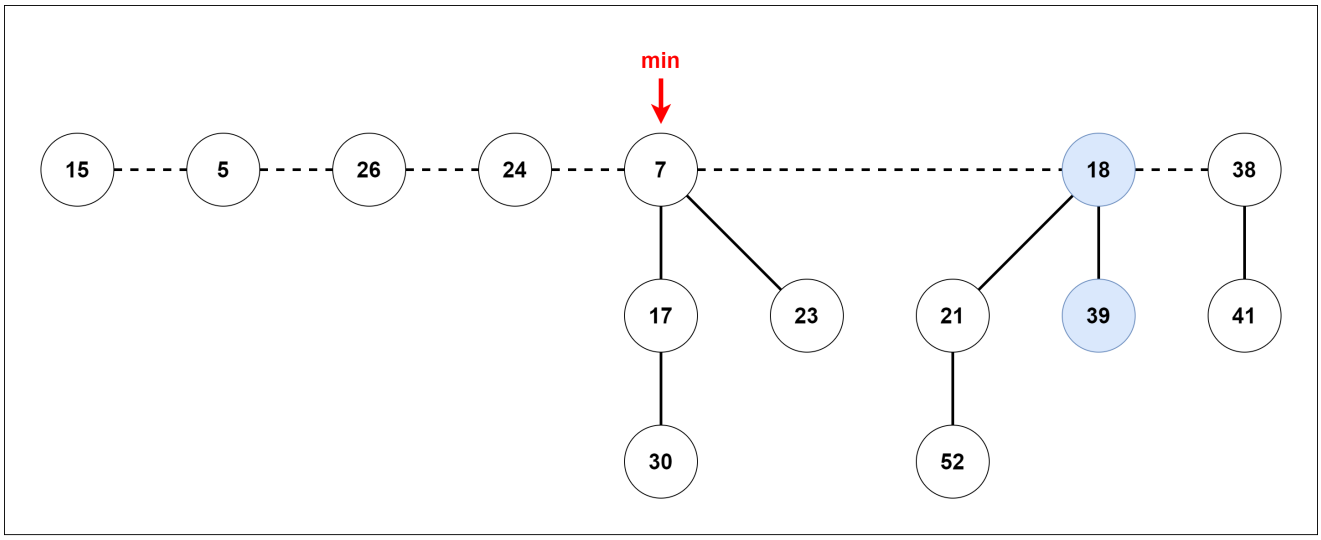
- **Cut Part**: The parent of the node with 35 is not null (which means it has a parent), and since 5 is smaller than its parent (which is 26), we perform a "Cut" operation. This means we remove the node with 35 from its current position in the tree and add it to the list of trees at the top level, known as the root list.



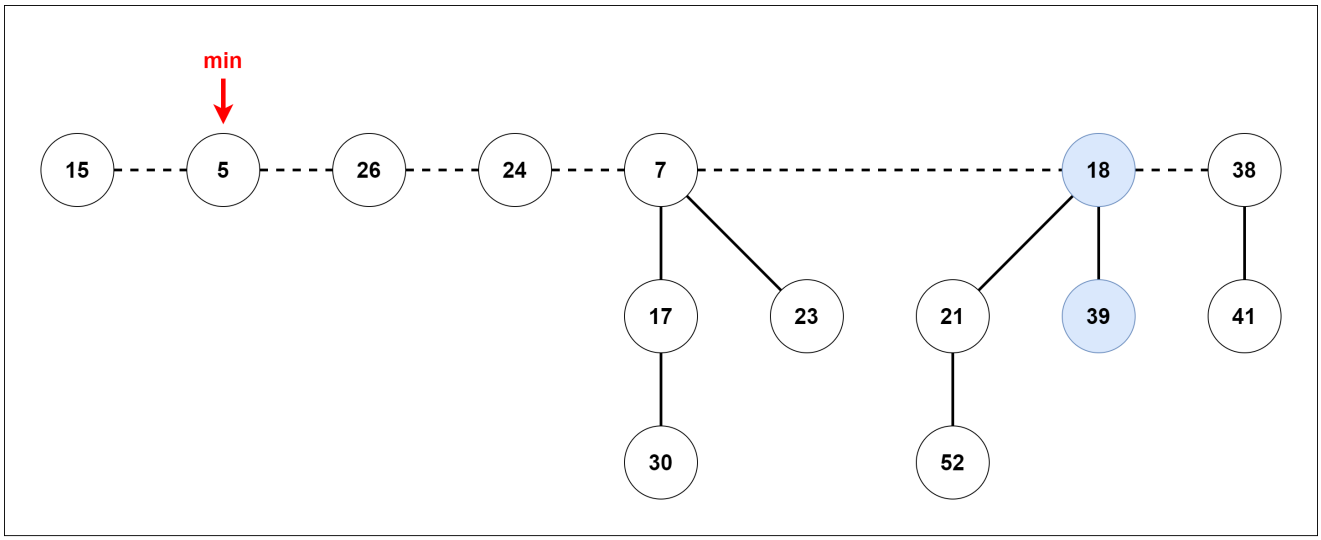
- **Cascading-Cut Part**: Now, because we cut the node with 35, we mark its parent, which is the node with 26. Marking is done to keep track of which nodes have lost children. If a marked node loses another child, it gets cut as well. So, we perform the "Cut" operation on the node with 26, removing it from its current position and adding it to the root list. Then, we mark it as unmarked because it's now in the root list.



- This process continues up the tree. Since the node with 24 is also marked, we again perform "Cut" and "Cascading-Cut" operations on it and its parent, resulting in adjustments to the structure of our Fibonacci heap.



- Finally, after these operations, we find that 5 is smaller than the current minimum value (which was 7), so we mark 5 as the new minimum.



So, through this example, we've successfully decreased the key from 35 to 5 in our Fibonacci heap and managed the structure accordingly.

Deleting a Node

Let's learn how to delete a node in a Fibonacci heap. It involves a couple of key operations: "decrease-key" and "extract-min." Here are the steps:

- **Step 1:** We begin with a node, let's call it "k," that we want to delete from the Fibonacci heap.
- **Step 2:** First, we perform the "decrease-key" operation on this node "k." What we do here is decrease its value to the lowest possible value, which is negative infinity, represented as "-∞."
- **Step 3:** Once we've made this node's value as small as it can be, we proceed to the "extract-min" operation. This operation removes the minimum node from the heap.

Since we've made node "k" have the smallest possible value, it will now be considered the minimum node in the heap.

So, by applying these two operations, "decrease-key" and "extract-min," we effectively delete the node "k" from the Fibonacci heap.

Decrease key and Delete node operations in C++

```
// Operations on a Fibonacci heap in C++

#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

struct node
{
  int n;
  int degree;
  node *parent;
  node *child;
  node *left;
  node *right;
  char mark;

  char C;
};

class FibonacciHeap
{
private:
  int nH;

  node *H;

public:
  node *InitializeHeap();
  int Fibonnaci_link(node *, node *, node *);
  node *Create_node(int);
  node *Insert(node *, node *);
  node *Union(node *, node *);
  node *Extract_Min(node *);
  int Consolidate(node *);
  int Display(node *);
  node *Find(node *, int);
```

```
  int Decrease_key(node *, int, int);
  int Delete_key(node *, int);
  int Cut(node *, node *, node *);
  int Cascase_cut(node *, node *);
  FibonacciHeap() { H = InitializeHeap(); }
};

node *FibonacciHeap::InitializeHeap()
{
  node *np;
  np = NULL;
  return np;
}

node *FibonacciHeap::Create_node(int value)
{
  node *x = new node;
  x->n = value;
  return x;
}

node *FibonacciHeap::Insert(node *H, node *x)
{
  x->degree = 0;
  x->parent = NULL;
  x->child = NULL;
  x->left = x;
  x->right = x;
  x->mark = 'F';
  x->C = 'N';
  if (H != NULL)
  {
    (H->left)->right = x;
    x->right = H;
    x->left = H->left;
    H->left = x;
    if (x->n < H->n)
      H = x;
  }
  else
  {
    H = x;
  }
  nH = nH + 1;
  return H;
}

int FibonacciHeap::Fibonnaci_link(node *H1, node *y, node *z)
{
```

```
  (y->left)->right = y->right;
  (y->right)->left = y->left;
  if (z->right == z)
    H1 = z;
  y->left = y;
  y->right = y;
  y->parent = z;

  if (z->child == NULL)
    z->child = y;

  y->right = z->child;
  y->left = (z->child)->left;
  ((z->child)->left)->right = y;
  (z->child)->left = y;

  if (y->n < (z->child)->n)
    z->child = y;
  z->degree++;
}

node *FibonacciHeap::Union(node *H1, node *H2)
{
  node *np;
  node *H = InitializeHeap();
  H = H1;
  (H->left)->right = H2;
  (H2->left)->right = H;
  np = H->left;
  H->left = H2->left;
  H2->left = np;
  return H;
}

int FibonacciHeap::Display(node *H)
{
  node *p = H;
  if (p == NULL)
  {
    cout << "The Heap is Empty" << endl;
    return 0;
  }
  cout << "The root nodes of Heap are: " << endl;

  do
  {
    cout << p->n;
    p = p->right;
    if (p != H)
```

```
    {
      cout << "-->";
    }
  } while (p != H && p->right != NULL);
  cout << endl;
}

node *FibonacciHeap::Extract_Min(node *H1)
{
  node *p;
  node *ptr;
  node *z = H1;
  p = z;
  ptr = z;
  if (z == NULL)
    return z;

  node *x;
  node *np;

  x = NULL;

  if (z->child != NULL)
    x = z->child;

  if (x != NULL)
  {
    ptr = x;
    do
    {
      np = x->right;
      (H1->left)->right = x;
      x->right = H1;
      x->left = H1->left;
      H1->left = x;
      if (x->n < H1->n)
        H1 = x;

      x->parent = NULL;
      x = np;
    } while (np != ptr);
  }

  (z->left)->right = z->right;
  (z->right)->left = z->left;
  H1 = z->right;

  if (z == z->right && z->child == NULL)
    H = NULL;
```

```
  else
  {
    H1 = z->right;
    Consolidate(H1);
  }
  nH = nH - 1;
  return p;
}

int FibonacciHeap::Consolidate(node *H1)
{
  int d, i;
  float f = (log(nH)) / (log(2));
  int D = f;
  node *A[D];

  for (i = 0; i <= D; i++)
    A[i] = NULL;

  node *x = H1;
  node *y;
  node *np;
  node *pt = x;

  do
  {
    pt = pt->right;

    d = x->degree;

    while (A[d] != NULL)

    {
      y = A[d];

      if (x->n > y->n)

      {
        np = x;

        x = y;

        y = np;
      }

      if (y == H1)
        H1 = x;
      Fibonnaci_link(H1, y, x);
```

```
      if (x->right == x)
        H1 = x;
      A[d] = NULL;
      d = d + 1;
    }

    A[d] = x;
    x = x->right;

  }

  while (x != H1);
  H = NULL;
  for (int j = 0; j <= D; j++)
  {
    if (A[j] != NULL)
    {
      A[j]->left = A[j];
      A[j]->right = A[j];
      if (H != NULL)
      {
        (H->left)->right = A[j];
        A[j]->right = H;
        A[j]->left = H->left;
        H->left = A[j];
        if (A[j]->n < H->n)
          H = A[j];
      }
      else
      {
        H = A[j];
      }
      if (H == NULL)
        H = A[j];
      else if (A[j]->n < H->n)
        H = A[j];
    }
  }
}

int FibonacciHeap::Decrease_key(node *H1, int x, int k)
{
  node *y;
  if (H1 == NULL)
  {
    cout << "The Heap is Empty" << endl;
    return 0;
  }
  node *ptr = Find(H1, x);
```

```
  if (ptr == NULL)
  {
    cout << "Node not found in the Heap" << endl;
    return 1;
  }

  if (ptr->n < k)
  {
    cout << "Entered key greater than current key" << endl;
    return 0;
  }
  ptr->n = k;
  y = ptr->parent;
  if (y != NULL && ptr->n < y->n)
  {
    Cut(H1, ptr, y);
    Cascase_cut(H1, y);
  }

  if (ptr->n < H->n)
    H = ptr;

  return 0;
}

int FibonacciHeap::Cut(node *H1, node *x, node *y)

{
  if (x == x->right)
    y->child = NULL;
  (x->left)->right = x->right;
  (x->right)->left = x->left;
  if (x == y->child)
    y->child = x->right;
  y->degree = y->degree - 1;
  x->right = x;
  x->left = x;
  (H1->left)->right = x;
  x->right = H1;
  x->left = H1->left;
  H1->left = x;
  x->parent = NULL;
  x->mark = 'F';
}

int FibonacciHeap::Cascase_cut(node *H1, node *y)
{
  node *z = y->parent;
  if (z != NULL)
```

```
  {
    if (y->mark == 'F')
    {
      y->mark = 'T';
    }
    else

    {
      Cut(H1, y, z);
      Cascase_cut(H1, z);
    }
  }
}

node *FibonacciHeap::Find(node *H, int k)
{
  node *x = H;
  x->C = 'Y';
  node *p = NULL;
  if (x->n == k)
  {
    p = x;
    x->C = 'N';
    return p;
  }

  if (p == NULL)
  {
    if (x->child != NULL)
      p = Find(x->child, k);
    if ((x->right)->C != 'Y')
      p = Find(x->right, k);
  }

  x->C = 'N';
  return p;
}

int FibonacciHeap::Delete_key(node *H1, int k)
{
  node *np = NULL;
  int t;
  t = Decrease_key(H1, k, -5000);
  if (!t)
    np = Extract_Min(H);
  if (np != NULL)
    cout << "Key Deleted" << endl;
  else
    cout << "Key not Deleted" << endl;
```

```
  return 0;
}

int main()
{
  int n, m, l;
  FibonacciHeap fh;
  node *p;
  node *H;
  H = fh.InitializeHeap();

  p = fh.Create_node(7);
  H = fh.Insert(H, p);
  p = fh.Create_node(17);
  H = fh.Insert(H, p);
  p = fh.Create_node(26);
  H = fh.Insert(H, p);
  p = fh.Create_node(1);
  H = fh.Insert(H, p);

  fh.Display(H);

  p = fh.Extract_Min(H);
  if (p != NULL)
    cout << "The node with minimum key: " << p->n << endl;
  else
    cout << "Heap is empty" << endl;

  m = 26;
  l = 16;
  fh.Decrease_key(H, m, l);

  m = 16;
  fh.Delete_key(H, m);
}
```

Complexities

Now, let's talk about the complexities, the time it takes for these operations in a Fibonacci heap:

Decrease Key: When we decrease the key of a node, it's a very efficient operation. In terms of time complexity, we can say it's O(1). That means it takes constant time, no matter how big the heap is.

Delete Node: Now, when it comes to deleting a node, it's a bit different. The time it takes depends on the size of our heap. For this operation, it has a time complexity of O(log n). So, the time it takes to delete a node grows logarithmically with the size of the heap.

So, to sum it up, decreasing a key is really fast and takes constant time, O(1). Deleting a node, on the other hand, takes a bit longer and grows logarithmically with the size of the heap, O(log n).

Operation	Time Complexity
Decrease Key	O(1)
Delete Node	O(log n)